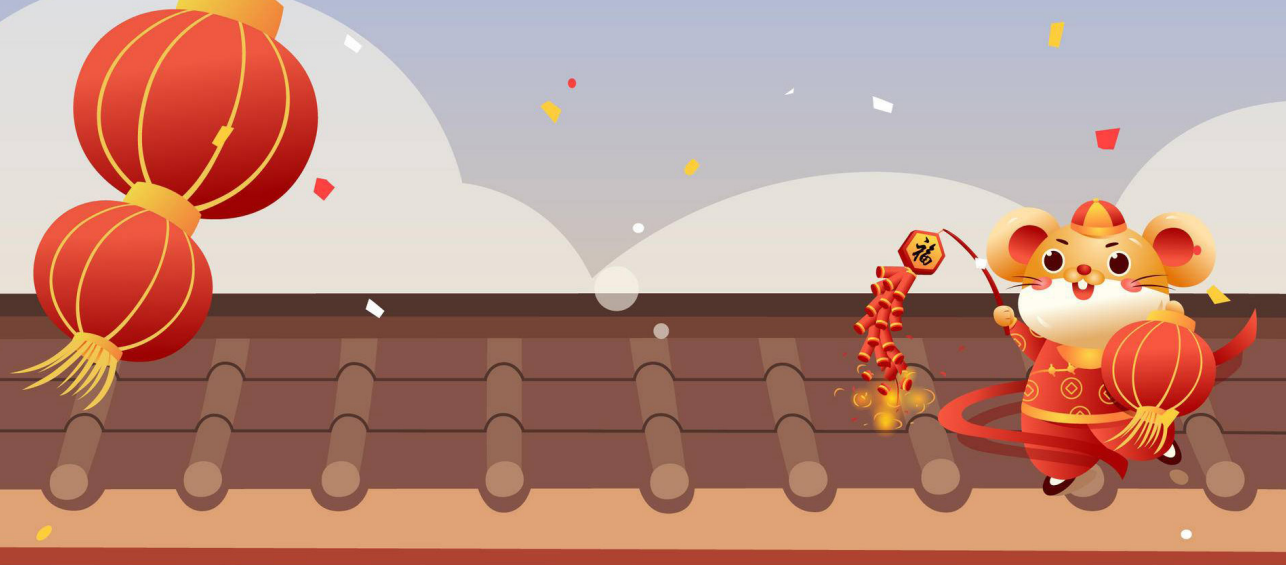


美团点评
2019 技术年货

CODE A BETTER LIFE

美团点评

微信扫码关注技术团队公众号

tech.meituan.com
美团技术博客

新年
快乐

序

新年将至，年味渐浓。

美团点评技术年货如期而至。

从 2013 年 12 月 4 日发布第一篇文章，一直到今天，美团技术团队官方博客已经走过了 6 个春秋。

截止目前，我们总共发布 376 篇技术文章，微信公众号（meituan-tech）的关注者也超过 20 万。

由衷地感谢大家一直以来对我们的鼓励和陪伴！

2020 年春节到来之际，我们精选美团技术博客几十篇技术干货以及数篇国际顶会论文，整理制作成一本厚达 900 多页的电子书，作为新年礼物赠送给大家。

这本电子书内容覆盖前端、后台、算法、数据等多个领域，希望小伙伴们在新的一年里收获满满，成长多多。

也欢迎大家转给有相同兴趣的同事、朋友，一起切磋，共同成长。

最后，祝大家合家欢乐，和和美美，锦簇花团！

目录

序 iii

前端 1

动态化 2

MTFlexbox 自动化埋点探索 2

Litho 在美团动态化方案 MTFlexbox 中的实践 14

开源 React Native 组件库 beeshell 2.0 发布 28

React Native 在美团外卖客户端的实践 61

代码质量与安全 85

Android 静态代码扫描效率优化与实践 85

Probe：Android 线上 OOM 问题定位组件 115

活动 Web 页面人机识别验证的探索与实践 135

React Native 工程中 TSLint 静态检查工具的探索之路 144

ESLint 在中大型团队的应用实践 165

App 流程管理及实践 182

美团 iOS 工程 zsource 命令背后的那些事儿 182

客户端单周发版下的多分支自动化管理与实践 190

美团外卖前端容器化演进实践 198

Bifrost 微前端框架及其在美团闪购中的实践 219

基本功 235

Litho 的使用及原理剖析 235

Android 兼容 Java 8 语法特性的原理分析 252

美团外卖商家端视频探索之旅 270

后台 290

基本功 291

Java 魔法类：Unsafe 应用解析 291

Java 动态追踪技术探究 308

字节码增强技术探索 320

JVM CPU Profiler 技术原理及源码深度解析 347

Java 动态调试技术原理及实践 370

从 ReentrantLock 的实现看 AQS 的原理及应用 399

架构 435

美团点评 Kubernetes 集群管理实践 435

美团集群调度系统 HULK 技术演进 450

保障 IDC 安全：分布式 HIDS 集群架构设计 463

Leaf：美团分布式 ID 生成服务开源 486

美团大规模微服务通信框架及治理体系 OCTO 核心组件开源 493

美团下一代服务治理系统 OCTO2.0 的探索与实践 499

实践与经验总结 515

XGBoost 缺失值引发的问题及其深度分析 515

Spring Boot 引起的“堆外内存泄漏”排查及经验总结 524

美团点评效果广告实验配置平台的设计与实现 536

根因分析初探：一种报警聚类算法在业务系统的落地实施 547

全链路压测自动化实践 565

降低软件复杂性一般原则和方法 580

算法 592

美团 BERT 的探索和实践 593

深度学习在搜索业务中的探索与实践 616

大众点评搜索基于知识图谱的深度学习排序实践 648

大众点评信息流基于文本生成的创意优化实践 671

AI Challenger 2018：细粒度用户评论情感分析冠军思路总结 695

WSDM Cup 2019 自然语言推理任务获奖解题思路 704

深度学习在美团配送 ETA 预估中的探索与实践 716

配送交付时间轻量级预估实践 729

ICDAR 2019 论文：自然场景文字定位技术详解 745

CVPR 2019 轨迹预测竞赛冠军方法总结 757

顶会论文：基于神经网络 StarNet 的行人轨迹交互预测算法 764

大数据 774

Hadoop YARN：调度性能优化实践 775

将军令：数据安全平台建设实践 794

OneData 建设探索之路：SaaS 收银运营数仓建设 812

研发团队资源成本优化实践 829

Jupyter 在美团民宿的应用实践 843

人物志 865

MIT 科技创新“远见者”：美团 NLP 负责人王仲远 865

美团技术委员会前端通道主席洪磊：爱折腾的斜杠青年 881

论文精选 890

The 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems 891

2019 IEEE/RSJ International Conference on Intelligent Robots and Systems 900

2019 INFORMS Annual Meeting 906

2019 International Joint Conference on Artificial Intelligence 915

The 25th ACM SIGKDD Conference on Knowledge Discovery and Data Mining 922

The Web Conference 2019 932

The 15th International Conference on Document Analysis and Recognition 943

前端

溪流汇成海，梦站成山脉。

十余年，前端领域从打包在后端工程中的一段 HTML 和 JS 代码逐渐演化出具有完整技术体系的全新篇章。

十余年，美团点评前端团队已成为美团点评业务发展的重要技术支撑。

十余年，美团点评也成为了全球最大的生活服务电子商务平台。

十余年的技术积累，是美团技术团队继续成长的有力后盾。过去的一年间，美团技术博客继续践行「分享美团点评优质技术内容、传递美团点评技术文化」的宗旨，从美团点评内部提炼不少优质内容，分享出来与业界同仁一起学习交流。其中前端领域发布数十篇优质文章，包括动态化、代码质量及安全、App 流程管理及实践、前端基本功等。

希望同业界同仁一起交流切磋，共同成长！

动态化

MTFlexbox 自动化埋点探索

叶梓

背景

跨平台动态化技术是目前移动互联网领域的重点关注方向，它既能节约人力，又能实现业务快速上线的需求。经过十年的发展，美团 App 已经变成了一个承载众多业务的超级平台，众多的业务方对业务形态的快速迭代和更新提出了越来越高的要求。传统移动端“静态”的开发方式存在一系列问题，比如包体积增长过快、线上 Bug 修复困难、发版周期长等，已经不能满足高速发展的业务需要。因此，美团平台自研了一套跨平台动态化方案——MTFlexbox。

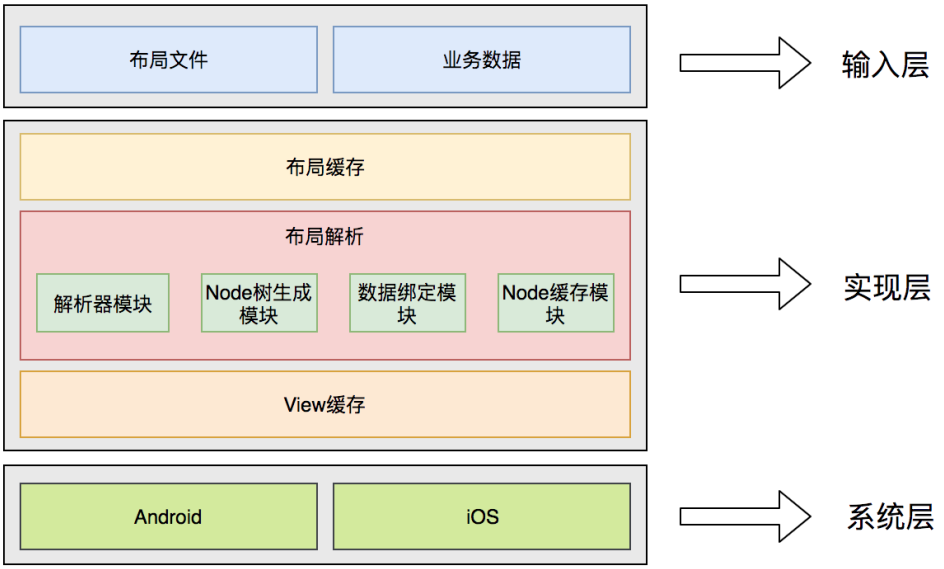
目前，MTFlexbox 已经广泛应用于美团首页、搜索、外卖等多个业务场景，并且已稳定运行两年有余。在 MTFlexbox 规范下，只需要写一份布局文件，就可以适用多端。在实际开发中，客户端开发同学开发布局的同时也要添加好埋点信息，帮助产品同学来评估上线后的效果。但现有布局埋点存在成本过高、准确率较低等痛点，为了解决这些问题，我们充分了解数据组开发人员和产品对数据统计的诉求，结合对 MTFlexbox 原理的深入理解，围绕 MTFlexbox 的埋点上报做了很多持续、有针对性的自动化工作，帮助多个项目的效率得到了显著提升。本文主要介绍美团在 MTFlexbox 自动化埋点方向所进行的一些探索，希望对大家能够有所帮助。

MTFlexbox 介绍

MTFlexbox 原理

MTFlexbox 是美团内部一套非常成熟的跨平台动态化解决方案，遵循了 CSS3 中提出的 Flexbox 规范，抹平了多平台的差异。MTFlexbox 首先按照 Flexbox 的规

范，定义了一套三端统一的 XML 布局文件，并将布局文件上传至后台；客户端下载带有布局文件的 JSON 数据后，解析布局并绑定 JSON 数据，最终交由 Native 渲染成视图。MTFlexbox 的整体架构图如下所示：



MTFlexbox 架构图

如果要用一句话来解释 MTFlexbox 的原理，就是按照约定的规则将 XML 内容映射成 Native 布局。从 Android 开发者的角度想，可以认为是把传统 XML 布局文件由内置改成从网络下发，实现展示样式动态改变的效果。上图第一层是 MTFlexbox 需要的输入，包括 XML 布局文件和展示的业务数据。其中 XML 布局文件中包括 UI 标签和埋点信息，每一种类型的埋点信息都作为一种属性和某一个 UI 标签相绑定。展示的业务数据可以通过后台下发或者写死在本地。为了将 XML 文件与具体的 View 进行解耦，MTFlexbox 在 XML 与 View 之间增加了一层 Node 层，即先将 XML 解析成 Node 树，再将 Node 树解析成 View 树。MTFlexbox 共有 3 层缓存：对 XML 文件的缓存、对 Node 节点的缓存、对 View 的缓存。其中缓存 View 指的是缓存一个 XML 创建的 View，通常只会缓存 rootView。在 Node 树生成了 View 树并绑定 JSON 数据后，才会最终渲染成 Native 控件。

MTFlexbox 适用场景

MTFlexbox 基本上支持 Native 上常用的基础控件的展示，对有 UI 定制化的需求支持度很高。但 MTFlexbox 的 XML 布局需要在运行前编写完成，只支持简单的三元表达式，逻辑能力有限。因此，MTFlexbox 特别适合布局样式复杂、变动频繁但交互简单的业务场景。例如美团 App 首页、搜索结果页等。这些业务场景都具备以下两个特点：

面向多业务方：各业务方有自己的个性化丰富样式，且不同时期可能需要不同的样式。

交互简单：点击跳转完成流量输送的简单交互。

下面是 MTFlexbox 使用场景的一些截图：

首页模板

搜索模板

MTFlexbox 自动化埋点前期工作

在美团实际的业务场景中，卡片的点击、曝光和加载数据是分析一个新产品形态上线效果好坏的最基本方式之一。相对应的，客户端的数据采集方式是洞察对于模块的点击、曝光和加载事件，然后结合上下文环境，比如页面标识、模块标识等，最后使用埋点上报工具和业务字段一起进行上报。MTFlexbox 作为模块级别的动态布局 UI 展示框架，对于数据采集方式的支持也是必不可少的。MTFlexbox 针对数据采集

的方式，做了以下两件事：

制定了一套双端统一的埋点标准化规范。

埋点类型定义成 Tag 标签属性，写入布局文件中。

MTFlexbox 结合美团自研的客户端数据上报工具，定义了多个专门针对埋点的特有属性字段，主要类型如下：

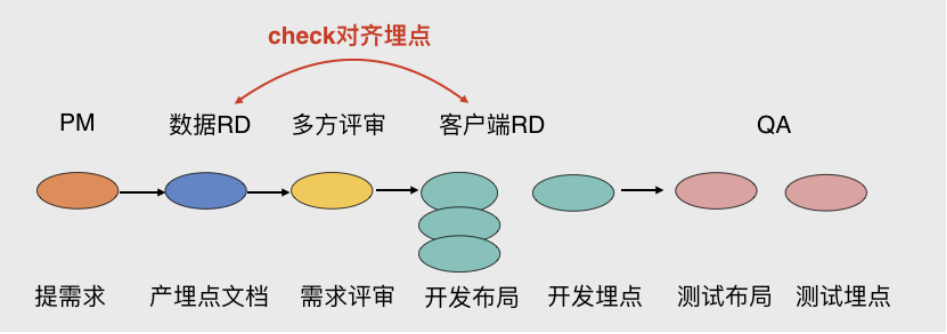
属性名	埋点类型
click-mge2-report	点击2.5埋点
click-mge4-report	点击4.0埋点
click-tag-report	点击Tag埋点
load-meg2-report	加载2.5埋点
load-meg4-report	加载4.0埋点
see-mge2-report	看见2.5埋点
see-mge4-report	看见4.0埋点

客户端开发人员在编写布局文件时，可以根据具体的产品需求，对不同控件的标签添加埋点属性，并且写入需要上报的业务字段。这样可以达到与 Native 埋点相同的效果，并且双端只需要配置一份埋点。以 see-mge4-report 埋点为例，布局埋点代码如下：

```
<Container style="width:360pt;justify-content:center;" >
    <Var name="see_MGE4" type="json">
            {
            "bid":"xxxxx",
            "cid":"yyyyy",
            "lab":{
                "isDynamic":true,
                "gather_index":"{extra.gather_index}",
                "index":"{extra.index}"
```

```
                }
            }
    </Var>
    <Container
              see-mge4-report="{see_MGE4}"
              click-url="{business.iUrl}"
              visibility="{{display.itemDynamic.
imageUrl}?visible:displaynone}" >

            <Img style="width:331pt;height:106pt;justify-content:center;"
                border-radius="5pt"
                scale-type="center-crop"
                src="{display.itemDynamic.imageUrl}"
                background="#FFF8F8F8" />
    </Container>
</Container>
```

MTFlexbox 动态化研发流程



MTFlexbox 动态化研发流程

从上述 MTFlexbox 动态化研发流程图中可以看出，数据需求和产品需求需要客户端开发人员在同一份布局文件中耦合在一起去实现，而且埋点属性和布局控件相绑定。这就导致在埋点过程中会出现很多问题，总结如下：

埋点成本过高

- 沟通成本较高：对于一个新的产品需求，首先产品需要将埋点需求提给数据组，数据同学理解了产品需求后产出埋点文档；然后产品、数据同学、客户端开发同学三方进行需求评审和埋点评审，沟通埋点需要上报的字段和时机等细节。

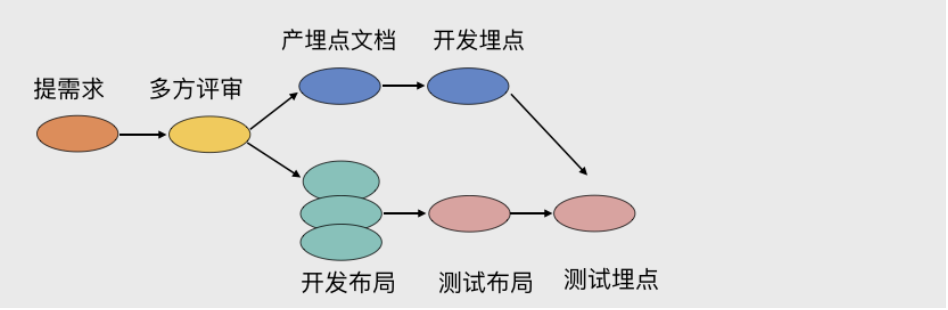
很多时候，一次沟通不到位，还需要反复沟通或者重新沟通，直到产品、数据同学和客户端开发人员三方对需求和埋点的理解一致为止。平均一个 5PD（5PD 指 5 个工作日）的需求需要消耗数据同学和客户端开发人员各 1PD 的时间来进行理解和沟通。

- 开发成本过高：客户端开发人员在编写 XML 布局文件时，往往要花 30% 左右的时间进行手动埋点和自测校验。

埋点线上事故多

- 因整个埋点缺乏自动化的埋点校验和预警机制，一旦开发人员出现人为的失误，导致错埋、漏埋现象，都有可能引发严重的线上故障。例如，客户端开发人员手动埋点时，出现人为失误引入了错误数据；产品验收阶段需要修改布局样式，客户端开发人员会出现“仅修改布局而遗漏埋点”的问题。

鉴于 MTFlexbox 存在埋点成本过高和线上问题较多的突出问题，我们迫切的希望通过一些手段来最大程度的规避和解决这类问题。埋点成本过高的原因在于 MTFlexbox 将布局和埋点耦合在一起编写，客户端开发人员需要做的事情过于“杂”和“多”。找到了这个痛点，很容易想到将埋点上报和布局编写解耦，让客户端开发人员只负责编写布局，数据同学只负责埋点配置，以此降低开发和沟通成本；同时通过自动化埋点校验手段提升埋点准确率，优化流程，减少线上事故的产生。基于此，产出我们理想的布局和埋点解耦之后的动态化研发流程，如下图所示：



新的动态化流程

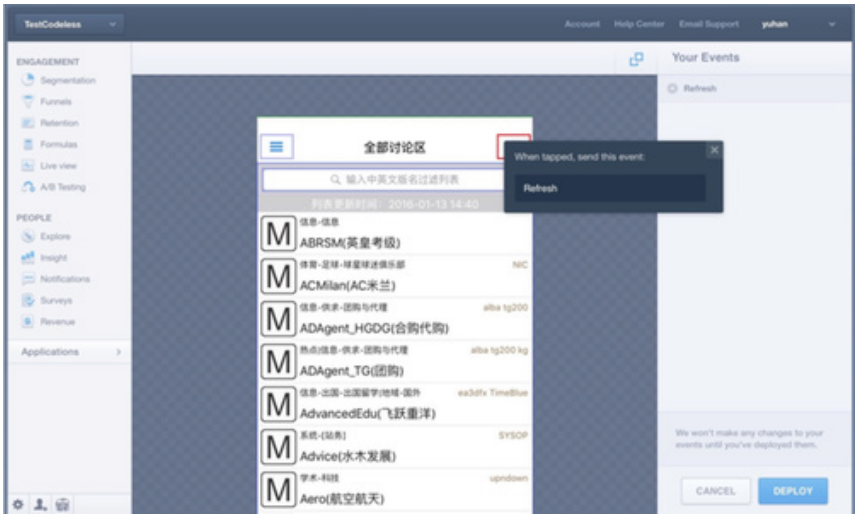
业内自动化埋点方案调研与参考

美团外卖前端无痕埋点实践

外卖团队在他们原有代码埋点方案的基础上，演化出了一套轻量的、声明式的前端埋点方案。详细内容可以参考博客:《美团点评前端无痕埋点实践》。此方案通过声明式埋点的方式实现了埋点代码与业务逻辑的解耦，并且支持对通用的业务数据的自动化上报。但此方案不能完全实现自动化埋点，并且实现成本较高。

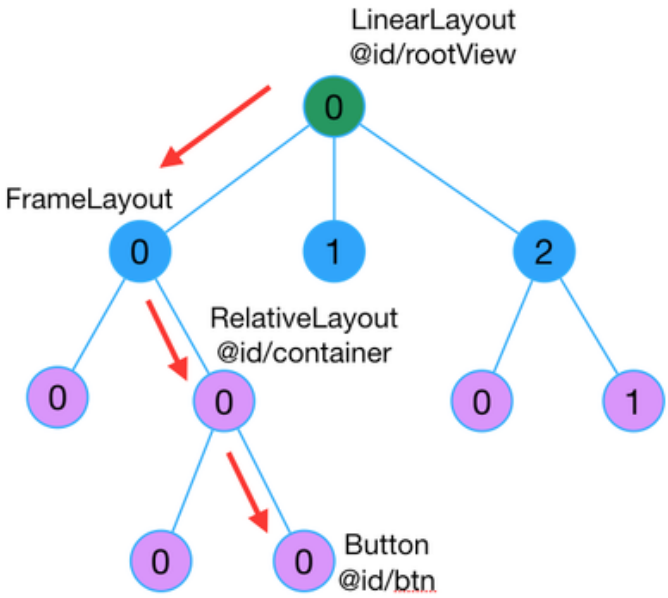
Mixpanel

Mixpanel 是一个已经商业化的可视化埋点方案，采用了截屏的方式在 IDE 中完成控件的圈选操作，体验较好值得我们借鉴。不过该方案主要面向非技术人员，不支持上报业务字段数据。



HubbleData

HubbleData 是网易开发的一个洞察用户行为的数据分析系统，提供一套完整的数据解决方案。



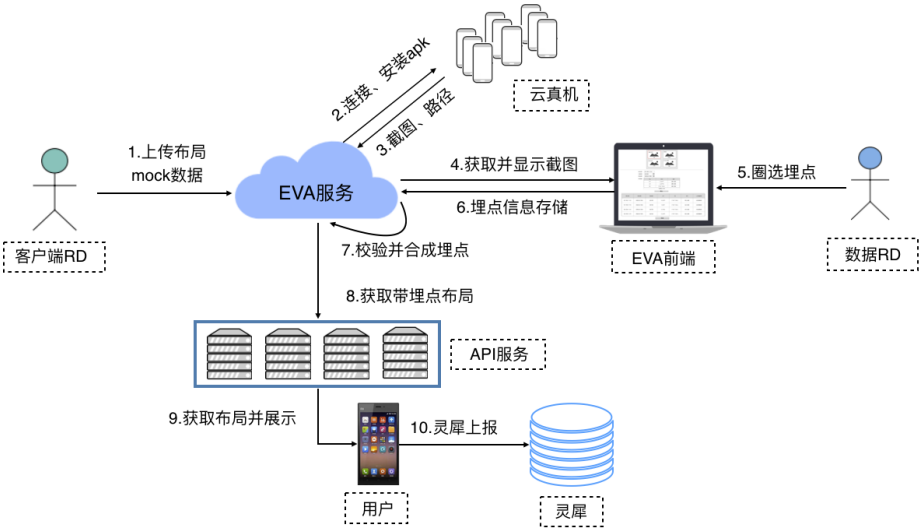
网易对 XPath 做了优化，主要体现在 View 索引的计算上：

- 原始 XPath 计算方式：每个 ViewGroup 下的所有 View 作为一个数组，索引从 0 开始。例如上图 Button 控件的 XPath 标识为：LinearLayout[0]/FrameLayout[0]/RelativeLayout[1]/Button[1]
- 网易 XPath 计算方式：每个 ViewGroup 下的所有 View 先按控件类型分类，然后再把每个类型中的控件按照数组的方式，从 0 开始。例如上图 Button 控件的 XPath 标识为：LinearLayout[0]#rootView/FrameLayout[0]/RelativeLayout[0]#container/Button[0]#btn

但是网易的这次优化，并没有解决由于同类型控件位置变更而引发的埋点错误问题，根源在于控件唯一标识不够准确。同时网易的修改仅限控件的一些固有属性，并没有搜集到更有价值的业务数据。

结合上述四种方案的优缺点，自动化埋点需要具备的几个条件，即：简洁直接的流程、友好可视化的前端配置界面、业务字段的可配置化、埋点有效性的检测。我们的方案就是基于这几个目标而诞生的。

我们的方案



整体流程

MTFlexbox 自动化埋点的核心流程，分为以下五步：

1. 客户端开发人员根据设计稿开发 XML 样式文件，自测通过后将 XML 样式文件与接口数据上传至 MTFlexbox 管理后台。
2. MTFlexbox 管理后台自动连接远程移动设备，并发送布局处理命令。远程移动设备将布局渲染结束后，抓取截图和布局层级信息（包括控件父子关系、控件位置、大小等信息）并上传至管理后台。
3. 前端页面从后台拿到 DPath 路径信息、坐标信息和截图信息，提供一套可视化的界面供数据同学进行模块内任一控件的埋点圈选配置。数据同学根据自身的需求，从目录树中圈选出自己希望配置埋点的控件。如下图所示，右侧模块中会出现红圈将选中的控件标出。

目录树圈选控件

4. 选中某个控件之后，数据同学对该控件进行埋点配置，元素类型支持当前元素和同类元素。其中同类元素可以节省数据同学对于同一种类型的控件的多次配置。对于已经圈选出的控件，列表中会详细展示出相关的信息，并附上控件对于的位置截图，能够方便数据 RD 定位埋点的控件具体位置。

埋点配置

5. MTFlexbox 管理后台根据前端上报的埋点信息，生成包含业务埋点的 XML 样式文件，供 C 端业务方后台调用。

```
<?xml version="1.0" encoding="UTF-8"?>
<Container>
    <Var auto-mge="true" name="ff510aa110844bb78c0b86fb04b26460"
type="json">
        {
            "bid" : "xxxxx",
            "cid" : "ssssss",
            "lab" : {
                "index" : "{_index}",
            }
        }
    </Var>

    <!-- 整个容器 -->
    <Container background="#FFFFFF" border-radius="10pt"
        click-mge4-report="{ff510aa110844bb78c0b86fb04b26460}"
        click-url="{_iUrl}" padding-left="10pt" padding-right="10pt">
        <!-- 左半部分 -->
        <Container style="flex-direction:column;justify-content:flex-
start;margin-top:15pt;">
```

6. 当客户端请求业务后台时，业务后台将包含业务埋点的 XML 样式文件下发给客户端，客户端根据配置完成埋点信息上报。

总结与展望

目前 MTFlexbox 自动化埋点方案已经使用在美团首页、大搜等业务中，整体埋点成本降低了 80%，上线后且无埋点故障。在此埋点方案的实现过程中，我们也踩了很多在设计之初没有预想到的坑，遇到了一些难点，详细设计问题和解决方案稍后的博客中的详细介绍，敬请关注美团技术团队公众号。

目前，我们基于 MTFlexbox 实现了 View 与埋点的自动化绑定，后期我们规划通过规范标准化后台下发的数据，包括业务数据和埋点数据，进而实现埋点数据的动态化下发和自动化绑定，进一步节省在埋点配置阶段和测试阶段的人力投入。

参考资料

[1] 网易 HubbleData 之 Android 无埋点实践
[2] 商业化埋点实现方案 mixpanel
[3] 美团点评前端无痕埋点实践

作者简介

叶梓、腾飞、田贝、张颖，美团终端业务研发团队研发工程师。

招聘信息

美团终端业务研发团队的职责是保障平台业务高效、稳定迭代的同时，持续优化用户体验和研发效率。团队负责的业务主要包括美团首页、美团搜索等千万级 DAU 高频业务以及分享、账号、音 / 视频等基础业务，支撑和对接外卖、酒店等 30 多个业务方。团队通过动态化能力建设，加快业务上线速度，帮助产品（PM）快速验证业务选型，做出业务决策；架构 / 服务标准化体系建设，提升前后端以及平台与业务线的沟通、合作效率；业务监控和体验优化，有效保障核心业务服务成功率的同时，提升用户使用美团 App 过程中的稳定性和流畅性。团队开发技术栈包括 Android、iOS、ReactNative、Flexbox 等。

美团终端业务研发团队是一个活力四射、对技术充满激情的团队，现诚聘 Android、iOS 工程师，欢迎有兴趣的同学投简历至 tech@meituan.com。

Litho 在美团动态化方案 MTFlexbox 中的实践

少宽　腾飞　叶梓

MTFlexbox

MTFlexbox 是美团内部应用的非常成熟的一种跨平台动态化解决方案，它遵循了 CSS3 中提出的 Flexbox 规范来抹平多平台的差异。MTFlexbox 适用于重展示、轻交互的业务场景，与现有 HTML、React Native、Weex 等跨平台方案相比，MTFlexbox 具备着性能高、渲染速度快、兼容性高、原生功能支持度高等优势。但其缺点在于不支持复杂的交互逻辑，不适合复杂交互的业务场景。目前，MTFlexbox 已经广泛应用在美团首页、搜索、外卖等重要业务场景。本文主要介绍在 MTFlexbox 中使用 Litho 优化性能的实践经验。

MTFlexbox 的原理

MTFlexbox 首先定义一份跨平台统一的 DSL 布局描述文件，前端通过“所见即所得”的编辑器编辑产生布局，客户端下载布局文件后，根据布局中的描述绑定 JSON 数据，并最终完成视图的渲染。MTFlexbox 框架图如下图所示：

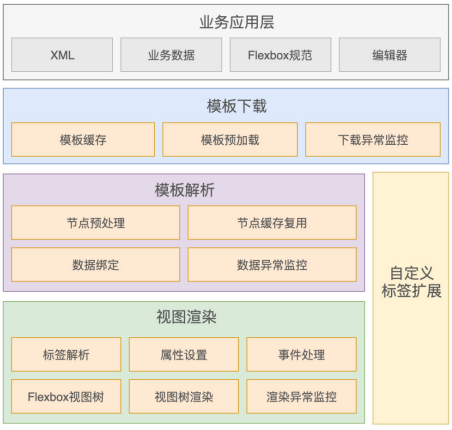


图 1　MTFlexbox 的架构

图中分为五层，分别是：

- 业务应用层：业务使用 MTFlexbox 的编辑器定义符合 Flexbox 规范的 DSL 文件（XML 模版）。
- 模版下载：负责 XML 模版下载相关的工作，包括模版缓存、预加载和异常监控等。
- 模版解析：负责模版解析相关的工作，包括标签节点的预处理、数据绑定、标签节点的缓存复用和数据异常监控等。
- 视图渲染：负责视图渲染相关的工作，包括把标签结点按照 Flexbox 规范解析成 Native 视图，并完成视图属性的设置、点击曝光事件的处理、视图渲染、异常监控等。
- 自定义标签扩展：提供支持业务扩展自定义标签的能力。

鉴于本篇博客主要涉及渲染相关的内容，下面将着重介绍 MTFlexbox 从模版解析到渲染的过程。如下图所示，MTFlexbox 首先会把 XML 模版解析成 Java 中的标签树，然后和 JSON 数据绑定结合成一颗具有完整数据信息的节点树。至此，模版解析工作就完成了。解析完成的节点树会交给视图引擎进行 Native 视图树的创建和渲染。

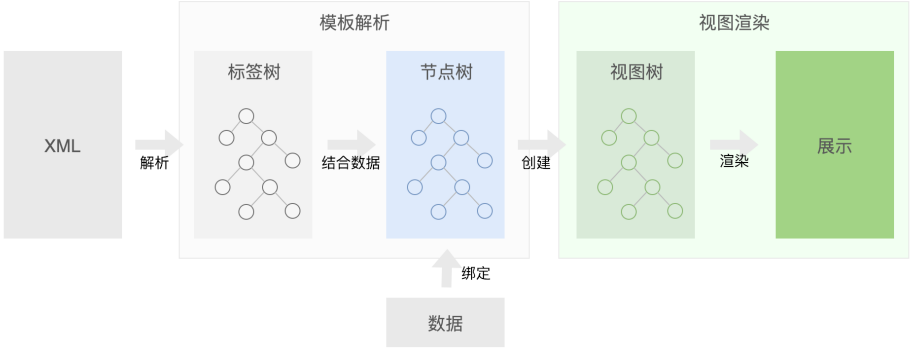


图 2　视图模版从解析到渲染

MTFlexbox 在美团动态化实践中面临的挑战

随着 MTFlexbox 在美团内部被广泛使用，我们遇到了两个问题：

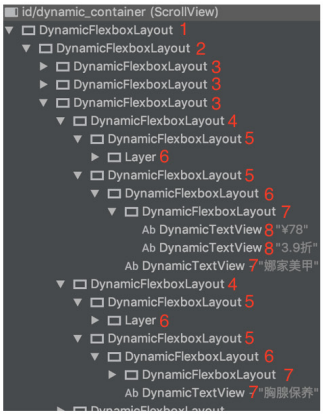
- 复杂视图因层级过深，导致滑动卡顿问题。
- 生成视图耗时过长，导致滑动卡顿问题。

问题一：视图层级过深

原因分析

MTFlexbox 使用的是 Flexbox 布局，Flexbox 布局可以理解成 Android LinearLayout 布局的一种扩展。Flexbox 在布局过程中使用到大量的布局嵌套，如果布局酷炫复杂，无疑会出现布局层级过深、视图树遍历耗时、绘制耗时等问题，最终引发滑动卡顿。下图是美团正在使用的一个模版的视图层级情况（布局最深处有 8 层）：

显示布局边界下的视图效果



视图树

图 3　模版布局层级效果

影响

布局层级过深在布局的计算和渲染过程中会导致过多的递归调用，影响视图的绘制效率，引发页面滑动 FPS 下降问题，这会直接影响到用户体验。

问题二：生成视图耗时过长

原因分析

视图生成耗时原因如下图所示：RecyclerView 在使用 MTFlexbox 布局条目时，需要对条目模版进行下载并解析生成节点树，这样会导致生成视图的过程耗时过长。为了提高视图生成速度，我们增加了复用机制，但是滑动过程中，如果遇到新的布局样式仍然需要重新下载和解析。另外，MTFlexbox 绑定的数据是未经解析的 JSON 字符串，所以也要比正常情况下的数据绑定更耗时一些。 正是上面两个原因，导致了 MTFlexbox 生成视图耗时过长的问题，这也会导致滑动时 FPS 出现突然下降的现象，产生卡顿感。

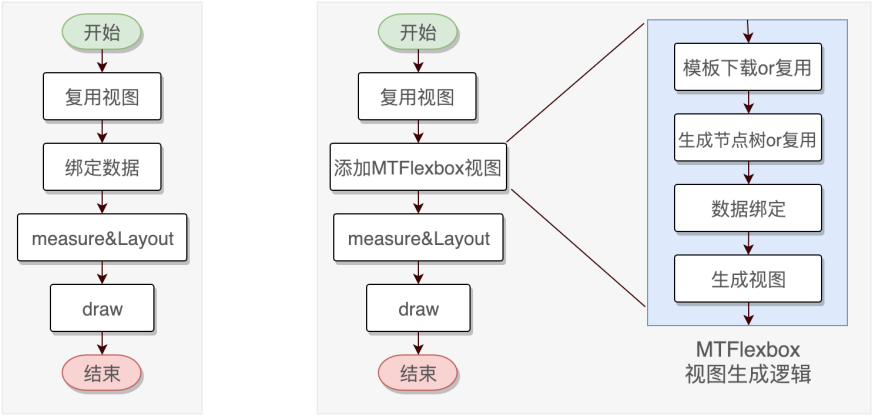


图 4　视图生成耗时原因分析

影响

由于视图的创建会阻塞主线程，创建视图耗时过长会导致 RecyclerView 列表滑动时卡顿感明显，也严重影响到了用户体验。

Litho

Litho 原理

Litho 是一套声明式 UI 框架，或者说是一个渲染引擎，它主要优化复杂 RecyclerView 列表的滑动性能问题。Litho 实现了视图的细粒度复用、异步计算布局和扁平化视图，可以显著提升滑动性能，减少 RecyclerView 滑动时的内存占用。详细介绍可以参考美团技术团队之前发布的另一篇博客：Litho 的使用及原理剖析。

Litho 的优势

通过对 Litho 原理的了解，我们可以看到 Litho 主要针对 RecyclerView 复杂滑动列表做了以下几点优化：

- 视图的细粒度复用，可以减少一定程度的内存占用。
- 异步计算布局，把测量和布局放到异步线程进行。
- 扁平化视图，把复杂的布局拍成极致的扁平效果，优化复杂列表滑动时由布局计算导致的卡顿问题。

扁平化视图刚好可以优化 MTFlexbox 遇到的视图层级过深的问题。异步计算布局虽然不能直接解决 MTFlexbox 生成视图耗时过长问题，但是给问题的解决提供了新的思路——异步提前完成视图创建。而且使用 Litho 还能带来一定程度的内存优化。所以如何将 Litho 应用到 MTFlexbox 中，进而来解决 MTFlexbox 现存的问题，是我们解下来要讨论的重点。

Litho + MTFlexbox 是怎么解决上述两个问题的?

解决问题一：视图层级过深问题

Litho 实现了布局的扁平化，所以最直接的方式就是使用 Litho 来替换 MTFlexbox 现有的视图引擎。视图引擎最主要的作用，是把 XML 文件解析出来的节点树变成 Litho 可以展示的视图，所以视图引擎替换的主要工作是把节点树转换成 Litho 能展示的视图。如下图所示。由于 Litho 使用的是组件化思想，需要先把节点转化成组

件，再把组件树设置给 LithoView，而 LithoView 是 Litho 用于兼容原生 View 的容器，它负责把 Litho 和系统视图引擎桥接起来。

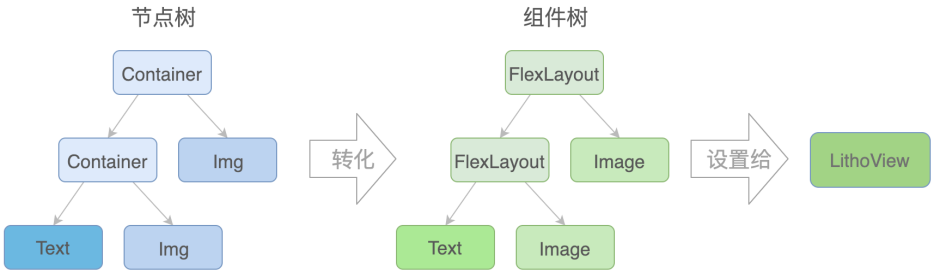


图 5　Litho 视图引擎从节点到视图的转换

不过视图引擎的替换并不是一帆风顺的，我们在替换过程中也遇到了 4 个比较大的挑战。

难点一：复用视图无法更新数据问题

问题描述：完成了节点树到组件树的转化以后，我们发现了一个严重的问题——复用的视图无法应用新的数据。

问题分析：当数据发生变化后，MTFlexbox 的节点树会对比新旧数据的变更，确定哪些结点需要更新并通知到具体的视图节点，然后更新显示内容（例如：新数据相比旧数据改变了 Text，那么只有 Text 对应的节点会通知对应的视图去更新内容）。Litho 组件的 Prop 属性是不允许更改的，而 Litho 组件中绝大多数属性都是 Prop 属性。

解决方案

方案一：使用 State 属性全局替换所有组件的 Prop 属性。这种方式的优点在于替换方式相对简单直接，缺点是侵入性强，替换工作量巨大且不符合 Litho 的思想（尽可能少的去改变组件的状态）。这种方案不是最优解，我们要降低侵入，简单快捷地实现数据更新，于是就产生了方案二，具体如下图所示。

方案二：封装一套 Updater 组件，用于创建真正展示的组件。Updater 组通过 State 属性监听对应节点的数据变更，当节点数据变化时，可以触发对应节点的更新。

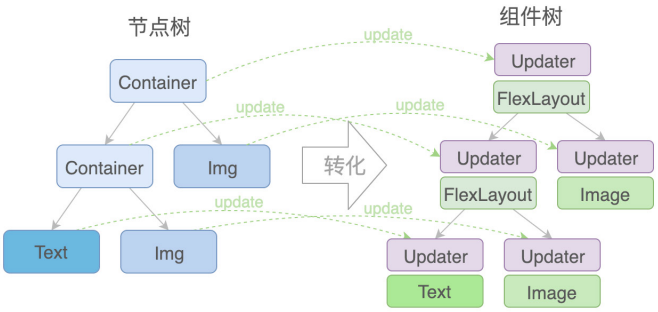


图 6　数据更新问题初版解决方案

但在后来的实践过程中，我们发现 Litho 整个组件树中只要有一个组件有状态更新，便会重新计算整个布局，而每次数据更新少说也会有几十个节点发生变化。频繁的重复计算反而导致性能变得很差。在经过了多种尝试以后，我们找到了最优的解决方案：

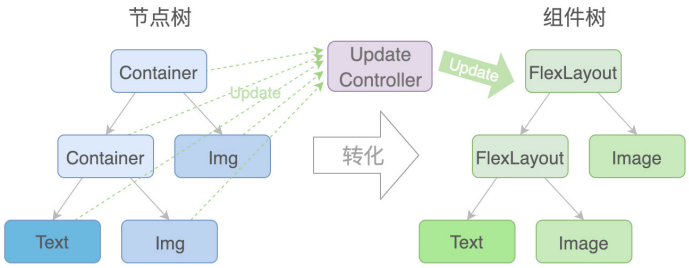


图 7　数据更新问题最终解决方案

如上图所示，状态更新控制器负责整个视图所有节点的更新操作。在所有数据都更新完成以后，统一交由状态更新控制器触发一遍组件更新。

难点二：Litho 不支持层叠布局问题

MTFlexbox 并没有完全严格的使用 Flexbox 布局规范，为了简单实现层叠效果，MTFlexbox 自定义了一种新布局规范——Layer 布局。Layer 布局具有以下两个特点：

- 特点一：Layer 的子视图在 z 轴上依次层叠展示。
- 特点二：Layer 的子视图默认且只能充满父布局。

原因分析：由于 Litho 严格遵守 Flexbox 布局规范，所以没有现成的 Layer 组件。

解决方案：自己实现 Layer 组件，满足第一个特点很容易，Flexbox 本身就支持层叠展示，只需要把子视图设为绝对布局就可以了。但是让子视图默认充满父布局就没有那么简单了，Flexbox 布局中没有任何一个属性可以达到这个效果。在经过了若干次组合多个属性的尝试以后，还是没能找到解决方案。既然 Layer 并不是 Flexbox 布局的规范，那么我们局限在 Flexbox 的束缚下，怕是很难找到完美的解决方案。那么，能不能在 Litho 中绕过 Flexbox 的约束，自己实现 Layer 效果呢？想在 Litho 中突破 Flexbox 布局的束缚，就需要了解 Litho 是如何使用 Flexbox 的。

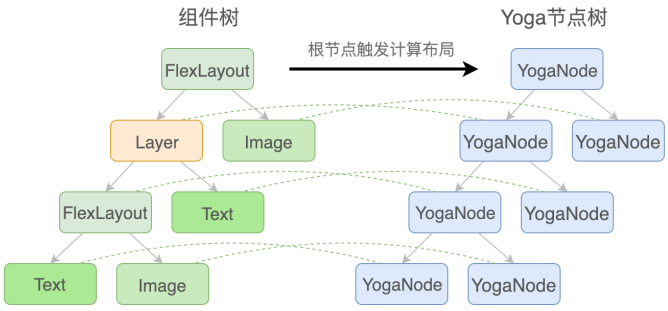


图 8　Litho 的布局计算原理

如上图，Litho 的 Flexbox 布局是由 Yoga 负责布局计算的。每一个 Litho 组件都会对应一个 Yoga 节点。但 Yoga 的布局计算过程是由根节点去统一触发的，子节点没有办法知道自己对应的 Yoga 节点是何时开始计算，及何时计算结束。这样以来，我们就没有时机去感知到 Layer 组件的布局是否计算完成，也就没有办法在 Layer 组件计算完成后去控制 Layer 子节点的计算。为了解决这个问题，我们做了两件事：

- 添加布局计算完成的回调，在布局计算完成后由根节点逐层通知子节点计算完成的消息。

- 拆分 Yoga 节点树，由 Layer 自己来控制子节点的计算。

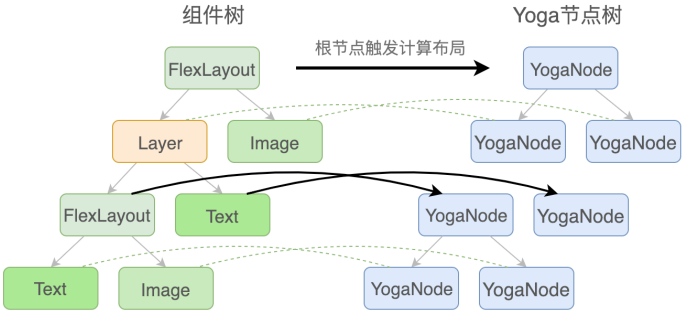


图 9　Layer 布局的实现原理

如上图所示，把 Layer 组件伪造成叶子节点，不把 Layer 组件的子节点设置给 Yoga，这样一个 Yoga 中的布局树就被 Layer 组件切割开了。当根节点计算完成以后，通知到 Layer 组件，Layer 组件再依次去设置子节点的宽高和位置属性，并触发子节点去完成各自子节点的布局计算。这样就完美地实现了 Layer 的布局效果。

难点三：Litho 图片组件不支持使用网络图片问题

原因分析：Litho 的组件是一个属性的集合，Litho 期望我们在组件创建时便确定了所有属性的值，所以 Litho 不支持网络图的展示。如果要支持从网络下载图片，就意味着图片组件用来展示的内容会发生变化。所以 Litho 自带的图片组件并不支持使用网络图片。

解决方案

方案一：用 State 属性解决网络图片下载带来的展示内容变化问题。我们在实践中发现，State 属性的更新会导致整个布局重新计算，其实替换图片资源不会导致图片组件的大小位置发生变化，根本不需要重新计算布局。为了减少使用 State 属性导致布局计算频繁的问题，就摒弃了这种方案。

方案二：Litho 官方额外提供的异步下载图片组件 FrescoImage 中使用的是图片代理方式。FrescoImage 使用 DraweeDrawable 来绘制视图，而 DraweeDrawable 实际上并不具备图片渲染的能力，只是在内部保存了一个真正的 Drawable 来负责渲

染。所以，DraweeDrawable 本质上是对真正要展示的图片做了一层代理，当从网络上下载下来真正要展示的图片后，只需要通过替换代理图片就可以完成视图的更新。美团下载图片使用的是 Glide，只需要按照这个思路实现自己的 GlideDrawable 就好了。

难点四：自定义标签扩展的接口不兼容问题

MTFlexbox 支持自定义标签的扩展，所以我们在完成基本视图标签的 Litho 实现以后，还需要支持自定义 Tag 的扩展，才算完成视图引擎的替换工作。

原因分析：MTFlexbox 在设计自定义标签接口时，只提供了允许使用 View 完成视图扩展的接口，但是 Litho 实现的视图引擎是使用组件作为视图单元和 MTFlexbox 对接的，所以接口不能兼容。

解决方案

方案一：重新提供使用 Litho 组件完成视图扩展的接口。其缺点是，需要 MTFlexbox 的使用方重新实现已经支持了的自定义标签，工作量较大，所以这种方案被抛弃了。

方案二：Litho 中使用业务方已经扩展好的 View。其优点是使用方对视图引擎的替换无感知。那么，怎样才能在 Litho 中使用业务方已经扩展好的 View 呢？可以先看下面这张图。

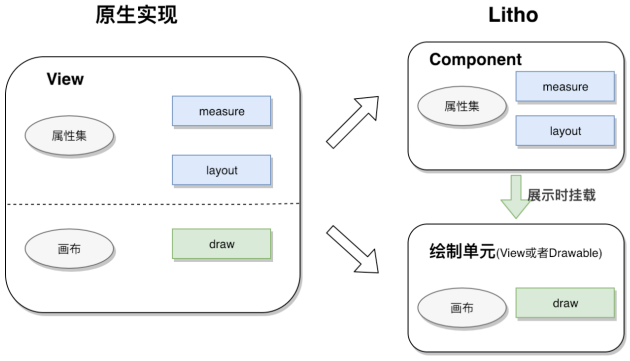


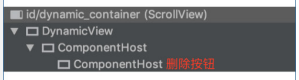
图 10　Litho 对 View 功能的拆分

我们可以简单的理解成 Litho 对 Android 的 View 做了一个功能拆分，把属性和

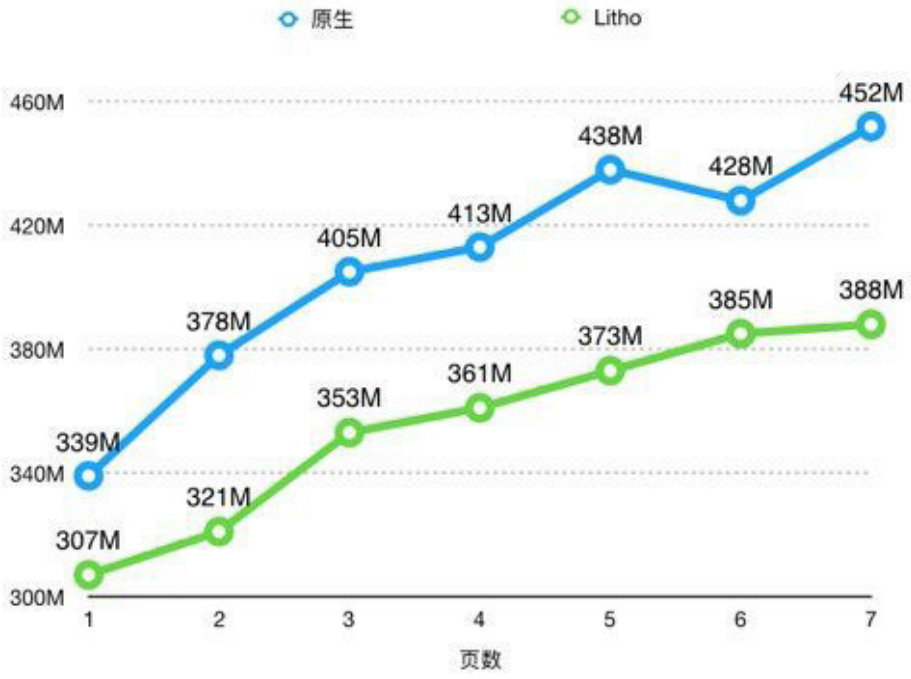
布局计算的能力放在了组件里面，每一种组件对应一个绘制单元来专门负责绘制。那么对于使用方扩展的标签，我们可以定义一个通用组件来统一承接。在挂载绘制单元时，再去调用使用方扩展的视图去绘制。

优化效果

至此，视图引擎的替换就完成了，整个视图引擎的替换做到了使用方无感知。完美解决了 MTFlexbox 视图层级深的问题，顺带还优化了部分性能。下面是布局层级优化效果的对比，可以看到相同样式下，使用 Litho 引擎实现的视图比使用 MTFlexbox 原生引擎的视图层级要浅很多。

视图引擎	显示布局边界下的视图效果	视图树
Litho		
原生		

除此之外，还有我们的内存优化成果。下图是美团首页使用 MTFlexbox 时，内存占用随滑动页数（一页为 20 条数据）增加而变化的趋势图。可以看到，使用 Litho 引擎实现的 MTFlexbox 比使用原生引擎的 MTFlexbox 在内存占用上能有 30M 以上的优化。

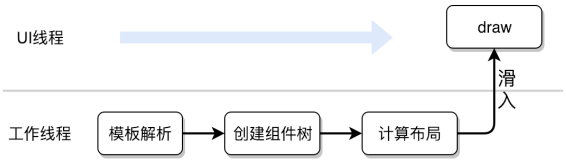


解决问题二：生成视图耗时过长

上文提到导致生成视图耗时过长的有两个原因：

（1）MTFlexbox 对布局模版的下载和解析耗时。（2）MTFlexbox 绑定时解析数据的耗时。

上文“自定义标签扩展的接口不兼容问题”中介绍过 Litho 的组件能够独立完成布局计算。另外，Litho 组件是轻量级的，所以我们直接把 Litho 组件作为 RecyclerView 适配器的数据源。这样就需要在数据解析时提前完成组件的创建，而组件的创建需要用到 MTFlexbox 的整个解析过程，也就是说，我们把 MTFlexbox 导致视图生成耗时过长的过程提前在数据层异步完成了。这样就不需要等到视图要展示时再去解析，从而规避了视图生成耗时过长的问题。具体的原理，可以参见 Litho 的使用及原理剖析一文中的 3.2 节“异步布局”。



如上图所示，在异步线程中提前完成 MTFlexbox 布局到 Litho 组件的转换。当视图真正要展示时，只需要把组件设置给 LithoView 就可以了。

优化效果

使用 Litho 引擎实现的滑动列表，在连续滑动过程中不会出现 FPS 波动问题，而使用 MTFlexbox 原生引擎实现的滑动列表则波动明显。（数据采集自魅蓝 2 手机，中低端手机优化效果明显）

总结

经过一段时间的实践，Litho + MTFlexbox 给美团 App 在性能指标上带来了较大的提升。但是还有很多问题待完善，我们后续还会针对以下几点进一步提升效果：

- 利用 Litho 组件属性不可变的特点，将提前异步布局进一步扩展为提前渲染出位图，在绘制时直接展示位图，可以进一步提升绘制效率。
- 优化 RecyclerView 相关的 API，降低侵入性。
- 解决有点击事件、埋点事件等属性的视图需要降级成 View 才能使功能生效的问题，进一步优化视图层级。

参考资料

Litho 官网　Flexbox 规范

作者简介

少宽、腾飞、叶梓，美团终端业务研发团队开发工程师。

招聘信息

美团终端业务研发团队的职责是保障平台业务高效、稳定迭代的同时，持续优化用户体验和研发效率。团队负责的业务主要包括美团首页、美团搜索等千万级 DAU 高频业务以及分享、账号、音 / 视频等基础业务，支撑和对接外卖、酒店等 30 多个业务方。

团队通过动态化能力建设，加快业务上线速度，帮助产品团队快速验证业务选型，做出业务决策；通过架构 / 服务标准化体系建设，提升前后端以及平台与业务线的沟通、合作效率；业务监控和体验优化，有效保障核心业务服务成功率的同时，提升用户使用美团 App 过程中的稳定性和流畅性。团队开发技术栈包括 Android、iOS、ReactNative、Flexbox 等。

美团终端业务研发团队现诚聘 Android、iOS 工程师，欢迎有兴趣的同学投简历至：tech@meituan.com（注明：美团终端业务研发团队）。

开源 React Native 组件库 beeshell 2.0 发布

小龙　宋鹏

引言

随着 React Native（以下简称 RN）技术的出现，大前端的发展趋势已经势不可挡，跨平台技术因其通用性、低成本、高效率的特点，逐渐成为行业追捧的热点。

为了进一步降低开发成本、提升产品迭代的效率，美团开始推广使用 RN 技术。随之而来，相关业务方开始提出对 RN 组件库的诉求。2018 年 11 月，公司内部发起了 RN 组件库建设，旨在提供全公司共用的组件库。鉴于我们团队在开源 beeshell 1.0（建议阅读 1.0 版本推广文章）时，积累了丰富的经验，于是就加入到了公司级 RN 组件库的项目共建中。完成组件库开发后，我们决定将共建的成果贡献出来，并以 beeshell 升级 2.0 的形式进行开源，共计开源 38（33 个组件与 5 个工具）个功能。在服务社区的同时，也希望借助社区的力量，进一步完善组件库。

图 D

beeshell 2.0 效果图

背景

前端开发（包括 Web 与 Native 开发）是图形用户界面（GUI）开发的一种。纵观各种 Web 以及 Native 产品界面，发现它们都是由一些基本组件（控件、元素）组合而成。受益于组件化、模块化的开发方式，前端 MV* 框架（如：AngularJS、React、Vue.js 等）也得以蓬勃发展。借助这些组件化框架，前端开发构建用户界面的过程，本质上是：开发组件，处理组件间的组合与通信。

这样看来，如果实现一套通用的组件并有效复用，便可以大幅减少开发组件的工作，进而提升前端开发的效率。各个业务方对 RN 组件库的诉求，目的也在于此：降低成本，提升效率。

- 然而，公司内不同事业部的业务场景和产品功能不尽相同，如何通过一套组件，来有效的支撑外卖、配送、酒旅及其他事业部的业务需求？这无疑是对组件库提出了更高的要求：
- UI 风格一致性。在同一个业务中，各个组件要有一致的 UI 风格，保证用户体验、塑造品牌形象。
- 通用性。可以支持不同的业务方，可以灵活定制不同的业务需求，最大化组件复用率，减少重复开发。
- 易用性。组件的功能、行为表现符合开发者的直观感受，易于学习和使用、减轻记忆负担；功能丰富，可以支持多种业务场景，支持特定业务场景的多种情况。
- 稳定性。RN 组件库需要同时支持 iOS、Android、Web 三个平台，组件要在三个平台可用、可靠、稳定。

简而言之，组件库的目标是通用、易用、稳定、灵活。

系统设计

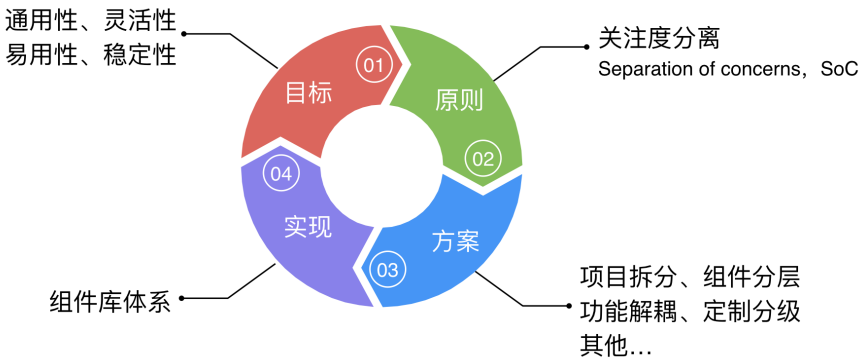


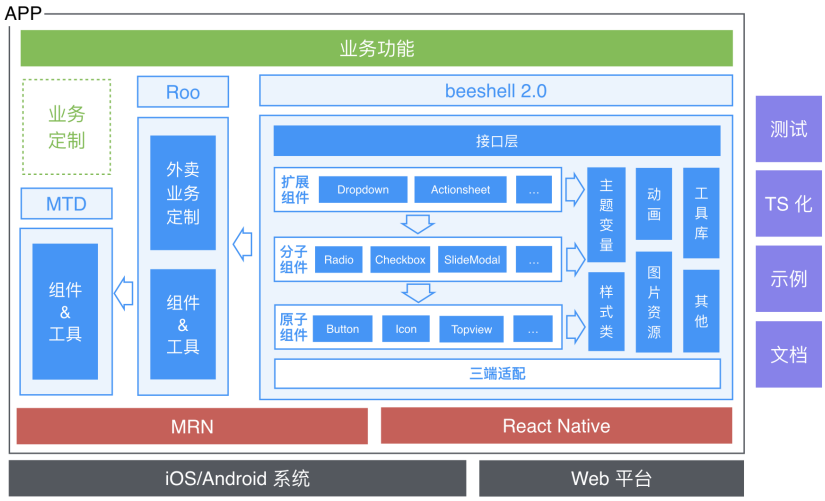
图 H

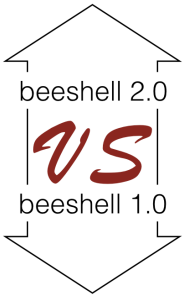
我们的目标是提供一套通用、易用、稳定、灵活的组件。然而，对一个组件来说，如果通用性、灵活性强，则易用性、稳定性势必较差。如何合理的处理这个矛盾呢?

为解决这个矛盾，我们使用了“关注度分离”的设计原则：首先将组件库需要支持的功能与特性进行分解；进而仔细研究特性的不同侧面（关注点），并提供相应的解决方案；最后整合各独立方案，解决冲突部分，形成整体的解决方案。

“关注度分离”的设计原则，贯穿整个组件库的设计与实现，是组件库的核心思想。以该原则为基础，衍生出了项目拆分、组件分层、功能解耦等具体方案，实现了一个组件库体系，保证了可以兼顾到相互矛盾的两个方面，实现最终目标。

架构升级


APP
业务功能
业务定制
MTD
组件&工具
Roo
外卖业务定制
组件&工具
beeshell 2.0
接口层
扩展组件
Dropdown
Actionsheet
...
分子组件
Radio
Checkbox
SlideModal
...
原子组件
Button
Icon
Topview
...
主题变量
样式类
动画
图片资源
工具库
其他
三端适配
测试
TS 化
示例
文档
MRN
React Native
iOS/Android 系统
Web 平台


beeshell 2.0
VS
beeshell 1.0

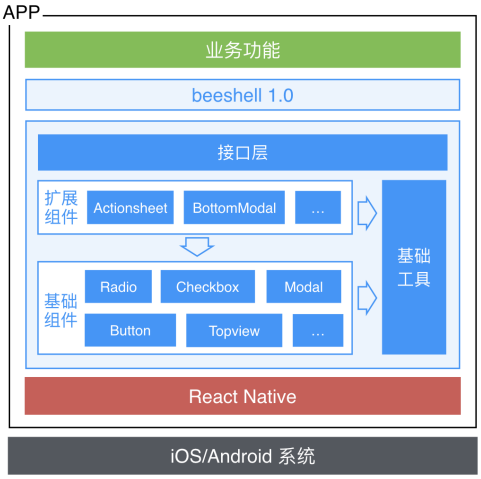

APP
业务功能
beeshell 1.0
接口层
扩展组件
Actionsheet
BottomModal
...
基础组件
Radio
Checkbox
Modal
Button
Topview
...
基础工具
React Native
iOS/Android 系统

图 F

beeshell 2.0 与 1.0 的架构在整体上保持一致，共分成四层：业务层、组件库（体系）、RN 层、系统层。而 beeshell 2.0 的架构升级，则主要体现在第二层与第三层：

第二层：组件库体系

组件库由 1.0 的一个项目演变成 2.0 的三个项目（版本），形成组件库体系。具体包含：MTD 公司通用版本、Roo 外卖定制版本和 beeshell 开源版本。

这种项目拆分的方式，符合“关注度分离”的设计原则，三个版本有各自不同的关注点：

- MTD 的关注点是通用性、灵活性，所以提供的是基础、通用的组件。组件的扩展能力极强，可以满足多个业务方的定制化需求。
- Roo 是对 MTD 的继承与扩展，定制了外卖业务的 UI 风格与功能，通用性减弱，功能性和业务性增强，关注易用性、业务性、一致性。
- beeshell（准确说是 2.0 版本）是对 Roo 的继承与扩展，与 Roo 相比，去除了过于业务化的组件与功能，纳入并整合社区的需求，关注通用性、易用性、稳定性。

组件库体系的三个版本，内部的架构设计一致，本文只详细介绍 beeshell 开源版本。

组件库使用了分层的架构风格，分成了接口层、组件层、工具层以及三端适配：

- 接口层。汇总所有组件，统一输出，使用全部引入的方式，方便组件使用。另外，为了避免引入过多无用组件，引起资源包过大，也支持组件的按需引入。
- 组件层。细分为原子、分子、扩展组件。 与 beeshell 1.0 相比，我们对组件在更细的粒度上进行拆分。同时，层次划分也更加精细、明确。如上图 F 所示：基础组件细分为分子、原子组件。这样，组件的继承、组合方式更加灵活，能够最大化代码复用。而且，原子、分子、扩展组件，各层次组件各司其职，“关注度分离”，兼顾通用性和易用性。

- 工具层。与 beeshell 1.0 相比，工具更加完备。不但新增了树形结构处理器、校验器等；而且工具的独立性更强，与组件完全解耦，与组件配合实现功能。这样，一方面，使得组件实现更加简洁，提升了组件的可维护性；另一方面，可以将工具提供给用户，方便用户开发，提升组件库的功能性、易用性。而且，工具与组件的解耦遵循“关注度分离”的原则。
- 三端适配。RN 在整体上实现了跨平台，iOS、Android、Web 三端使用一套代码，但是在一些细节方面，例如：特殊 API 的支持、相对位置计算等，各个平台有较大差异。为了更好的支持三端，保证跨平台稳定性，还需要在这一层做很多适配工作。

第三层：RN 层

这一层新增了 MRN，MRN 是对 RN 的二次封装，与 RN 底层实现保持一致。组件库在两个平台的表现一致。

MRN 是基于开源的 RN 框架改造并完善而成的一套动态化方案。MRN 从三个方面弥补 RN 的不足：

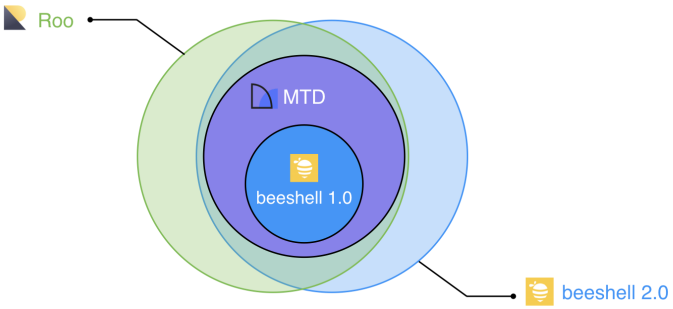
- 动态化能力。RN 本身不支持热更新，MRN 借助公司内发布平台，实现包的上传发布、下载更新、下线回滚等操作。
- 双端（未来三端）复用问题。从设计原则上保证开发者对平台的差异无明显感知。
- 保障措施。在开发保障方面：提供脚手架工具、模版工程、开发文档等，方便开发者日常的 MRN 开发工作。提供多级分包机制，业务内部和业务之间能够灵活组织代码；在运行保障方面，提供运行环境隔离，使得不同业务之间页面运行无干扰。提供数据采集和指标大盘，及时发现运行中的问题，同时提供降级能力以应对紧急情况。

除此之外，整个组件库体系，还具备以下特点：更加完善的测试方案；丰富的代码示例；使用 TypeScript（以下简称 TS）语言进行开发，充分利用 TS 的类型定义与检查；针对每个组件都有详细的文档说明。

协作模式

这里通过结构和流程两个方面，详细介绍 beeshell 1.0 与 beeshell 2.0 以及 MTD、Roo 的关系。三个版本之间通过 Git Fork 建立依赖关系，使用源码依赖的方式实现项目拆分。对于用户而言，不同版本的相同组件，底层依赖与实现都是一致的。

结构



流程

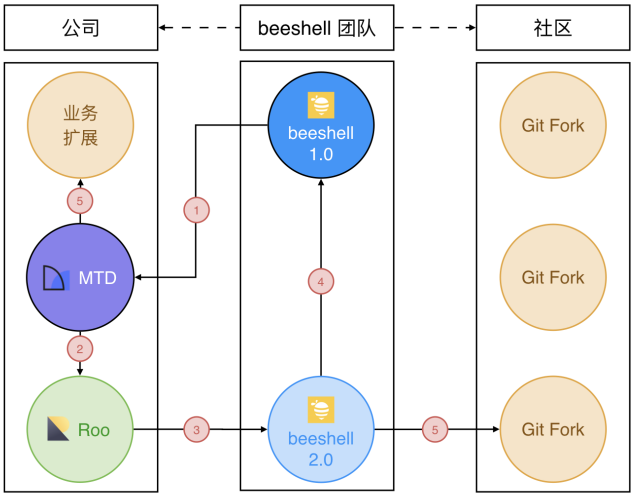


图 X

结构方面：MTD 包含 beeshell 1.0 的全部内容，并进行了组件数量的扩充、组件功能的增强；Roo 包含 MTD，进行了定制与业务扩展；beeshell 2.0 与 Roo 基本一致，去除业务部分，纳入社区需求。

流程方面：

- 首先，我们在 beeshell 1.0 的开发以及开源中，积累了丰富的经验。在建设 MTD 公司通用版组件库时，贡献了 50% 的组件；同时，贡献了许多设计模式与思路，大大加速了组件库的建设。
- 其次，在 MTD 建设完成后，为了更加方便、快速的接入外卖的相关业务，以 MTD 为基础，定制了外卖主题的组件库 Roo，提升了组件库的业务功能性和易用性。外卖相关的业务项目，在接入 Roo 后直接使用，无需再进行主题的定制与调整，在一定程度上节省开发成本。
- 第三，我们将共建的成果贡献出来，以 Roo 为基础，升级 beeshell 2.0 并开源。将部分过于业务化的组件移除，纳入了社区的相关需求，保证组件库的通用性、易用性与稳定性。
- 最后，对于公司内部，各个业务方可以以 MTD 为基础进行扩展，定制自己的业务主题组件库（Roo 就是第一个业务扩展）；对于社区，各个开发者可以根据实际的业务需求，以 beeshell 为基础，定制扩展组件库。

综上所述，我们以 beeshell 开发团队为桥梁，建立了美团公司与开源社区之间进行技术交流的通道，美团公司、beeshell 团队以及社区，可以在技术上互帮互助，共同建设、进步。

方案实现

UI 风格一致性

UI 风格一致性的重要性在于，对内可以保持平台统一性、提升团队效率、打磨细节体验；对外可以塑造品牌形象、减轻用户学习成本、保持产品的体验一致性。

UI 风格一致性的关键要素有很多，包括色彩、排版、字体、图标以及交互操作等。可以归纳为两类：样式一致性和动效一致性。

beeshell 延用了 Roo（袋鼠 UI）的 UI 设计规范，其内容涵盖了 PC 端与移动端、Web 平台与 RN 平台，对 UI 与交互给出了详细的视觉规范，旨在保证外卖事业部，全部产品的 UI 一致性。UI 规范的技术实现方式如下：

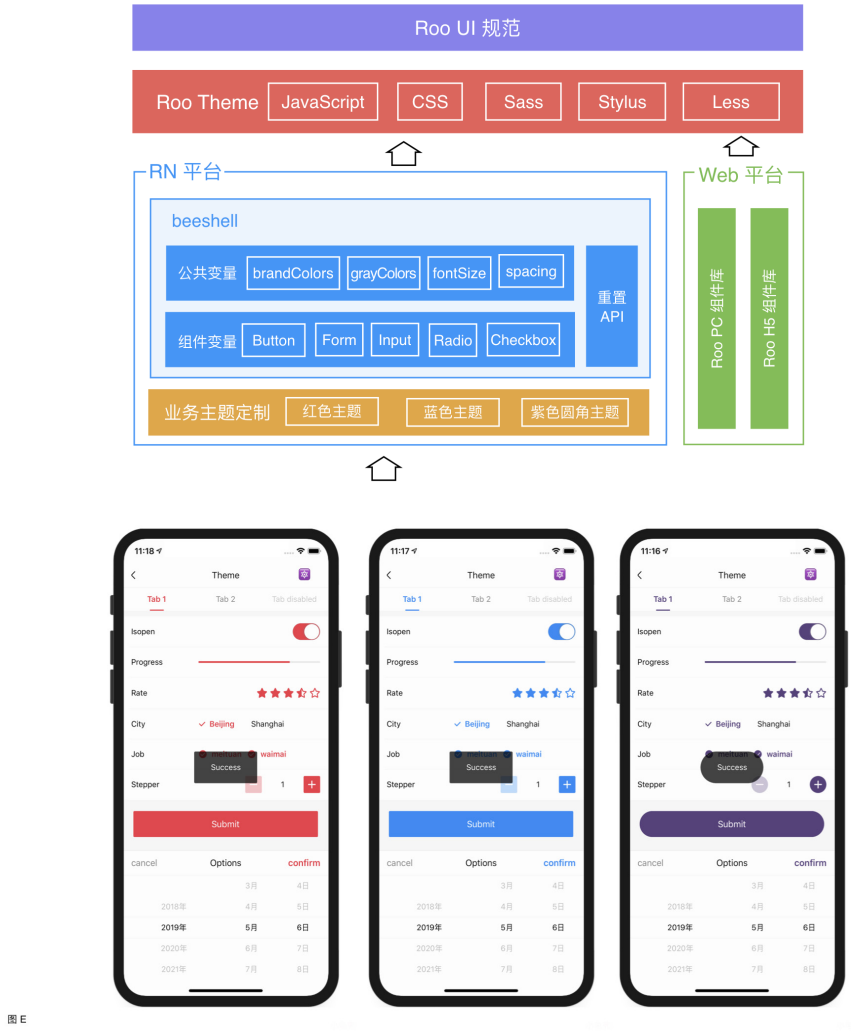


图 E

Roo Theme 向上实现了 UI 规范具体内容，将设计规范统一收敛，向下输出主题变量、组件样式类、通用样式工具等，供各个组件库以及业务方使用。

Roo Theme 主要使用 SasS 预处理器实现，提供了各个组件的统一样式类。为了满足不同技术栈的需求，主题变量的输出提供了 JavaScript（以下简称 JS）、CSS、Less、SasS、Stylus 多种语言，不同平台、技术栈可以根据情况自由选择。

beeshell 基于 Roo Theme 输出的 JS 主题变量，通过样式和动效两个方面，保证了 UI 一致性。

- 样式一致性。通过继承、扩展 Roo Theme，定义全局性的主题变量，用于组件的样式部分定义。主题变量范围涉及品牌色、灰度、字体尺寸、间距以及组件级变量，为组件以及组件之间样式一致性，提供了全面的保障。同时，组件库提供了自定义主题变量的接口，可以重置相关变量的值，对 UI 风格进行一致性调整，实现一键换肤。另外，使用“内部样式 < 主题 < 扩展样式”的样式优先级覆盖策略，保证了样式部分的定制能力（在下文“定制化能力分级设计”章节中详细介绍）。
- 动效一致性。一方面，依赖主题变量中定义的动画开关变量（主要考虑到一些低端 Android 机器的性能问题），用户可以关闭某个组件的动画；另一方面，依赖组件库的良好分层设计，我们将动画类独立实现，这样可以使用策略模式，将动画方便的集成到任意组件中。

下文详细介绍样式一致性和动效一致性：

样式一致性

样式一致性，可以从色彩和排版两个方面来保证。

首先，介绍下色彩部分。在 App 应用中，色彩元素扮演的角色仅次于功能。人与计算机的互动，主要是与图形用户界面的交互，而色彩在该交互中起着关键作用。它可以帮助用户查看和理解界面内容，与正确的元素互动，并了解相关操作。每个 App 都会有一套配色方案，并在主要区域使用其基础色彩。

正因为有无数种色彩组合的可能，在设计一个 App 时，人们的配色方案也有无数种选择。本文不纠结于如何选择一个好的配色方案，而是介绍一个配色方案应该具有哪些元素。一套完整的配色方案，应该包括品牌主色、品牌功能色、中性色。本文以 beeshell 的配色方案举例说明。

色彩：品牌主色

品牌主色应该是应用中出现最频繁的颜色，通常用来强调 UI 中关键部分的颜色。beeshell 的品牌主色色值为 #ffd800，如下图所示：

color-1	color-2	color-3	color-4	color-5	color-6	color-7	color-8	color-9	color-10
#ffffe6	#fffda3	#fff87a	#fff152	#ffe629	#ffd800	#d9b100	#b38c00	#8c6900	#664900

通常，一个产品的 UI 只会有一个品牌主色。然而，像 beeshell 这种品牌主色色值较浅的情况，一个品牌主色并不能够支撑所有的应用场景。此时，可以通过加深主色的方式，再增加几个色值，beeshell 的品牌主色还包括一个加深的色值 #ffa000，用于某些组件的激活状态，如下图所示：

color-1	color-2	color-3	color-4	color-5	color-6	color-7	color-8	color-9	color-10
#fffae6	#ffeaa3	#ffdc7a	#ffcb52	#ffb829	#ffa000	#d98200	#b36500	#8c4b00	#663300

对于品牌主色的个数，需要根据色值的情况而定，不必过于拘泥规则，只要能有一致性的用户体验即可。

色彩：品牌功能色

beeshell 的功能色内容与使用场景如下图所示：

Info 一般信息	#188afa
Success 成功	#61cb28
Warning 警告	#ff8400
Danger 危险	#f23244

图 U

这四个色值，分别用于一般信息、成功、警告、失败这四种业务场景，以及从这四种业务场景所衍生出的场景。在一定程度上，保证色彩的一致性。

色彩：中性色

beeshell 的中性色（灰度）的内容与使用场景如下图所示：

GrayBase 标题、正文	GrayDarker 副标题	GrayDark 补充信息	Gray 取消、禁用	GrayLight —	GrayLighter —	GrayLightest —
#111111	#333333	#555555	#888888	#aaaaaa	#cccccc	#ebebeb

图 G

中性色应用于界面主体文本的颜色，灰度范围的确定，可以大大提升色彩的一致性。

接下来介绍下排版，具体可以分为字体、间距、边线。

排版：字体

beeshell 的字体尺寸（Font Size）集，是基于 12、14、16、20 和 28 的排版比例，如下图所示：

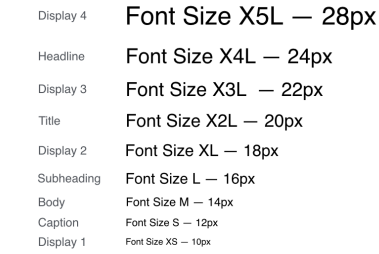


图 B

这样的排版比例，可以使得界面的文字内容更加协调、流畅，进而提升了排版的一致性。

对于字重（Font Weight），beeshell 只使用正常（Normal）和加粗（Bold）两种：Normal 用于一般文本；Bold 用于强调，吸引用户注意力。只使用这两种字重，也避免了因为不同字体家族（Font Family），对字重的支持范围不同，而导致视觉差异。

除了字体尺寸和字重，影响排版的还有字体行高（Line Height）。为了达到适当的可读性和阅读流畅性，可以根据字体的大小和粗细，设定字体行高。默认情况下，RN 应用：行高 = 字体大小 * 1.2。如下图所示：

图 L

beeshell 使用了默认的字体行高，在一定程度保证了可读性和排版的一致性。

排版：间距

间距是 UI 元素与元素之间、父元素与子元素之间的空白区域，一个应用排版风格一致性，很大程度取决于间距。一个组件的最终宽高，应该由内容、内边距以及边框决定，是自适应的，而不应该直接定义宽高。

对于同一个 App，间距应该在一个合适的范围取值，可以通过定义『小号间距』、『中号间距』、『大号间距』等来划分信息层次。例如 beeshell 的 Button 组件，有三种尺寸。实现效果如下图所示：

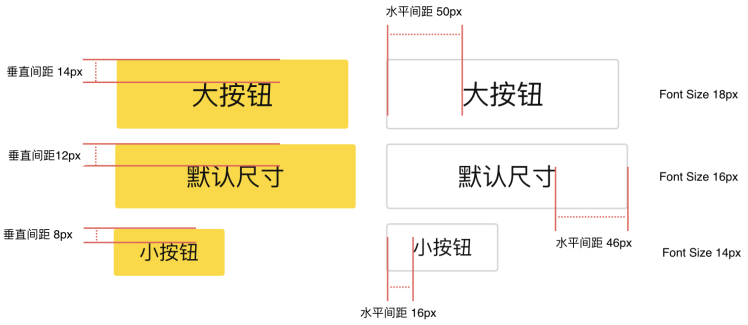


图 S

排版：边线

边线（边框）部分，需要统一元素的边框宽度、颜色和圆角，边线虽然对 UI 风格的影响较小，但是不可或缺。beeshell 使用的边框宽度为一个物理像素，使用 RN 提供的 StyleSheet.hairlineWidth 接口实现；定义了三种灰度的边框颜色；主要使用 2px 的圆角。

最后，样式的一致性，还涉及到图标、布局等相关内容，因为 beeshell 对这些内容的涉及较少，这里不做详细介绍。

动效一致性

动效展示了应用的组织方式和功能。

动效可以：

- 引导用户在视图中的视觉焦点。
- 提示用户完成手势操作后会发生什么。
- 暗示元素间的等级和空间关系。
- 让用户忽视系统背后发生的事情（比如抓取内容、或加载下一个视图）。
- 使应用更有个性、更优雅、体验更加一致。

beeshell 组件库基于 Animated 进行了二次封装，提供 FadeAnimated 和 SlideAnimated 两个动画类，支持淡入淡出动画和滑动动画，可以使用策略模式集成到任何组件中。

beeshell 将逐渐在所有的组件集成这两种动画，保证动效的一致性。下文展示已经实现了动画的组件，先睹为快。

Button 组件使用 FadeAnimated 类实现动画，动效如下图所示：

Modal 组件使用 FadeAnimated 类实现动画，动效如下图所示：

Dropdown 组件使用 SlideAnimated 类实现动画，动效如下图所示：

定制化能力分级设计

要开发一套全公司共用的组件，需要同时满足酒旅、外卖 C 端、外卖 B 端以及外卖 M 端等业务的需求，这对组件的定制化能力提出了更高的要求。

为了进一步增强组件的定制化能力，提升组件的通用性、灵活性；同时，避免属性（API）的无节制增加，进而导致组件难以维护，我们设计了定制化能力分级的策略。

这里以一个常见的业务场景：底部上滑弹框，来举例说明分级设计。

图 M

如上图所示：第一个例子比较通用、规范。“区域文字内容”的文案与样式需要支持自定义；第二个例子，需要支持多行文字以及图标，即“区域内容”需要支持自定义；第三个例子，自定义的重点，由区域以及区域内部，转移到区域之间的布局，“区域布局”需要支持自定义。

区域文字内容、区域内容、区域布局，三个层面的灵活性，可以涵盖一个业务场景所有定制化需求了。以当前业务场景为例，来讲解如何使用定制化能力分级设计，完成全部的定制化需求。

首先实现了一个 BottomModal 底部弹框组件，然后开始进行定制化设计。

第一级定制化：定制主题变量

“完成”文本的颜色，使用的是主题变量定义的品牌主色（Brand Primary Dark），beeshell 默认的品牌主色为黄色。

通过组件库提供的自定义主题变量接口，可以修改品牌主色的色值，进而修改了“完成”文本的颜色。同理，可修改“取消”、“标题”的颜色。

修改品牌主色，影响范围很大，所有组件的色彩风格统一变化。如果我只想把文本的颜色改成红色，但是并不想修改品牌主色，应该如何定制呢？可以使用第二级定制化。

第二级定制化：提供定制属性

这里提供 LabelText（类型为 String）和 LabelTextStyle（类型为 TextStyle）属性，如下图所示：

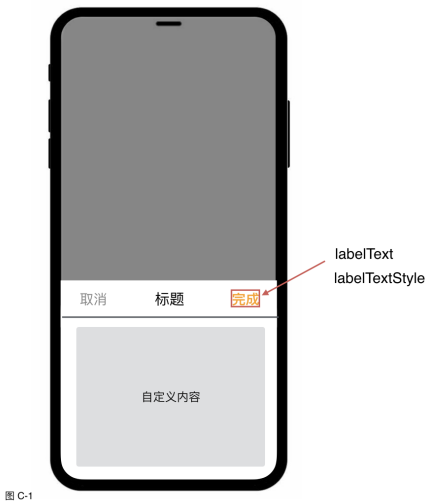


图 C-1

代码实现为：

```
<Text style={this.props.labelTextStyle}>{this.props.labelText || '完成
'}</Text>
```

LabelText 用于定制文案内容，将 LabelTextStyle 整体暴露出来，而不是只暴

露颜色单个属性，这样处有两点好处：

- 开发者都熟悉 Style 这个名称与用法，但并不知道 xxxColor 是什么，组件更加易用。
- Style 不仅可以定制 Color，还支持 fontSize、fontWeight 等属性，定制能力更强。

以上两级主要是样式部分的定制，使用了样式优先级覆盖的策略，扩展样式（labelTextStyle）覆盖主题，主题覆盖内部样式。

下一步，就是对于多行文字、图标的支持。

第三级定制化：开放渲染区域

提供 Label 属性，类型为 ReactElement 对象，任意定制 UI，如下图所示：

图 C-2

到这里，区域以及区域内部的定制化需求，就都可以满足了。但是区域布局的定制化，因为布局情况太多，并不容易实现。

如果再提供几个属性，用于定制布局方式、头部边框样式、底部按钮，按照这种方式，属性会无节制增加，势必造成组件难以维护，易用性也会大打折扣。那应该如何实现？我们设计了第四级。

第四级定制化：继承 / 组合基类

在 beeshell 1.0 的开源推广文章中也有讲到过，我们在组件库开发之初，对常见组件，进行了全面的梳理。在比较细的粒度，对组件进行拆分，以继承的方式，层层依赖，以功能渐进式增强的方式，实现各个组件。

这样使得开发者，可以在任意层级上继承、组合组件，进行定制化开发，提供了极强的扩展能力。具体实现如下：

首先，组件库实现一个 SlideModal 滑动弹框组件。这是一个比较底层、基础的组件，功能相对少，支持多个方向的滑动动画，内容完全由开发者自定义，通用性、定制化能力极强。实现效果如下：

然后，组件库实现了 BottomModal 组件，继承 SlideModal，固定滑动的方向和开始位置，弹框内容横向拉伸至全屏、纵向自适应，功能增强而定制化能力减弱。实现效果如下：

前文已经讲到，产品需求已经超出了 BottomModal 定制化的能力，强行实现只会带来不良后果。所以，我的方式是组合使用 SlideModal，开发一个新的组件，也就是第四级定制化。新组件的实现效果如下：

第四级定制化，是使用了新的思路，不再盲目的增加一个组件的功能，来帮助开发者满足产品需求，而是提供了基础工具。基础工具实现了底层、复杂的部分，表现层的部分则让渡给开发者，用户自己实现，“授人以鱼不如授人以渔”。

对比业界的开源 RN 组件库，也没有几个可以达到第四级的定制化能力。

beeshell 通过四个级别的定制化的能力，可以轻松搞定所有的产品的需求。

总之，定制化能力分级设计，是对定制化需求进行分类，针对每一类的定制化需求，设计了相应的策略，最终合成一个完整的解决方案，符合“关注度分离”的设计原则。

功能丰富

功能丰富旨在支持多种业务场景，支持特定业务场景的多种情况，进而提升组件库的易用性。功能丰富通过两个方面保证：组件丰富度、单个组件的功能丰富度。

组件丰富度

为了保证组件丰富度，我们通过多种渠道（既有组件整理、业务组件提取、参考标杆项目）来收集组件。最终规划了包括通用类、导航类、数据录入类、数据展示类、操作反馈类、基础工具在内的 6 个大类，共 50 多个功能，旨在覆盖尽可能多的业务场景。

目前，beeshell 已经实现了 38 个功能，与业界标杆项目的对比情况如下图：

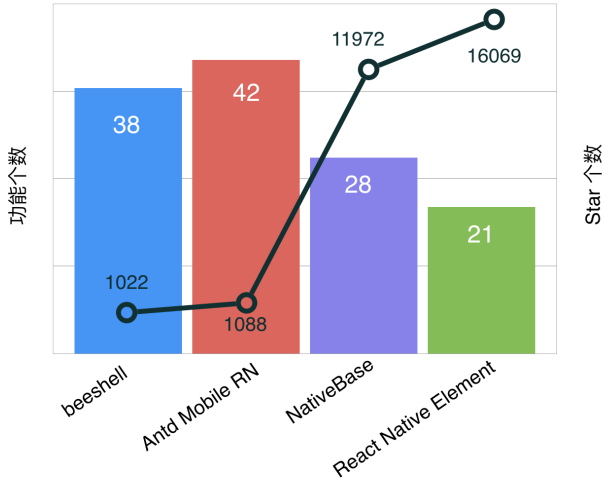


图 K

虽然，beeshell 的组件数量还比不上 Antd Mobile RN（用不了多久也会超过），但已经超过 NativeBase 和 React Native Element。beeshell 在组件数量上有很大优势，可以支持更多的业务场景，且支持全部引入和按需引入，用户无需担心打包过多无用代码的问题。

功能丰富度

功能丰富度针对的是单个组件所支持的功能，旨在覆盖特定业务场景的多种情况。

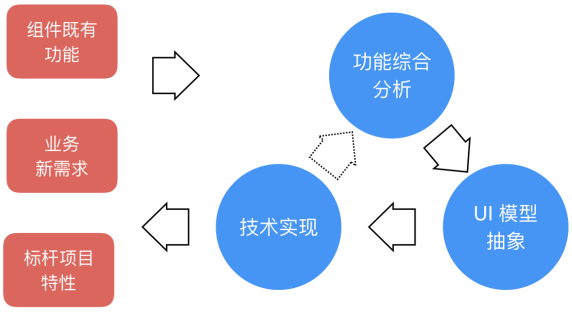


图 O

对于一个组件的功能丰富度，我们通过多方式收集、功能综合分析、UI 模型抽象、技术实现、验证反馈几个步骤来保证。

- 多方式收集。通过多种方式来收集组件功能，收集方式包括：组件既有功能、业务新需求、标杆项目特性。
- 功能综合分析。对收集的功能进行全面、综合分析与考虑，得出组件需要支持的功能特性。
- UI 模型抽象。对组件功能进行抽象设计，根据 UI 模型，明确抽象设计的合理性和有效性。
- 技术实现。根据平台特性、技术选型来完成代码实现。
- 验证反馈。组件实现后，会应用到业务或者示例工程来验证，如果组件并不能满足需求，需要重复执行以上几步。

下文以 SlideModal 组件的实现，举例说明：

首先，通过多种方式，收集到的滑动弹框场景如下：

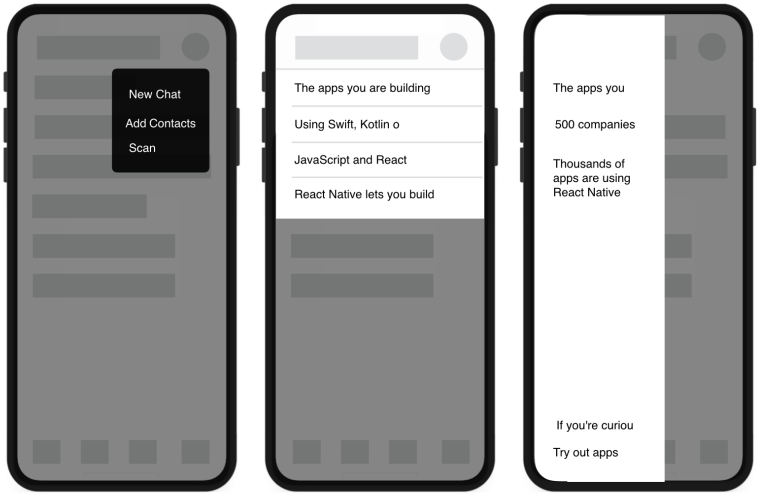


图 Q

其次，综合分析得出，SlideModal 组件需要支持的功能有：弹出位置自定义、滑动方向自定义、全屏 / 半屏自定义。

然后，进行 UI 模型抽象，抽象设计如下图所示：

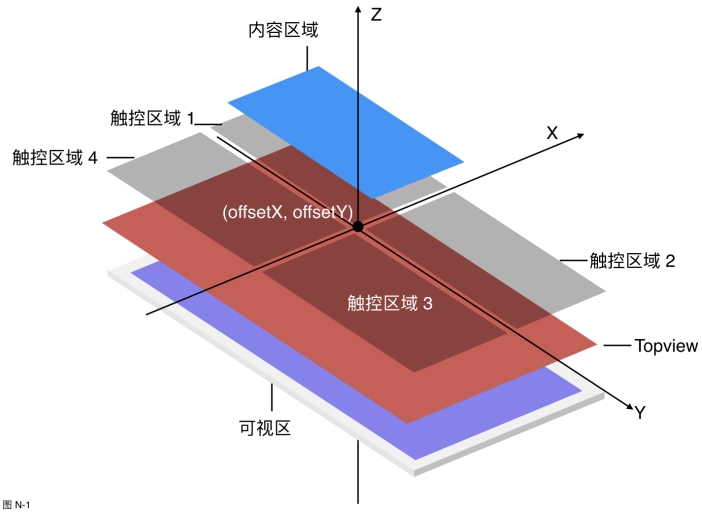


图 N-1

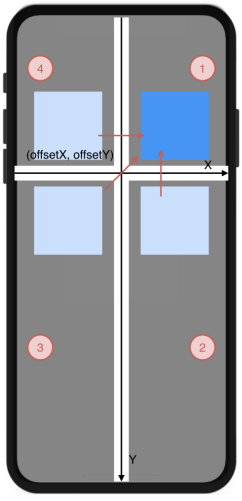


图 N-2

由 UI 模型得出，SlideModal 组件通过 (offsetX, offsetY) 坐标值来定义弹出位置；为了支持全屏 / 半屏效果，将屏幕分割为 4 个区域，分别自定义触控效果（阻断点击或者击穿）；弹框内容在一个区域展示，每个区域有 3 种滑动方向（图 N-2），共支持 12 种滑动动效。

最后，是基于 RN 平台的技术实现，并经过大量业务场景的验证反馈。Slide-

Modal 组件的最终实现效果如下：

对比业界开源 RN 组件库，针对滑动弹框场景，没有几个可以超过 SlideModal 的业务支持能力。

SlideModal 组件只提供底层支持，如果要应用到真实的业务场景，还需要基

于该组件进一步开发。beeshell 也提供了更高层次的定制组件，例如：Dropdown、Popover 等，可以直接使用。

除了 SlideModal 之外，还实现了其他功能强大的组件：Slider 滑块组件，支持纵向和横向滑动；Rate 评分组件，实现一套滑动评分的机制，支持定制任意 UI 元素。由于篇幅有限，在此不再赘述。

易用性提升

组件易用性的提升，通过命名、文档和示例这三个方面来保证。

命名

命名包括组件名、属性与方法名。

一个组件，实际上就是 Web 页面或者 App 中的元素、控件，通常因为原生控件的能力薄弱，而进行二次封装。所以组件名与原生控件名的名称，尽量保持一致。例如，Form 与 HTML Form 标签一致，Switch 从 iOS 控件 UISwitch 中得来。这样的命名，可以给与开发者更加直观的感受，通过名称就能知道组件大概的 UI 与功能，降低学习和使用的成本。

属性与方法的命名，既要考虑原生控件的属性名，又要考虑组件库命名的一致性。例如，表单录入的相关组件，包括 Input、Radio、Checkbox、Switch 等，组件的值要统一使用 Value 命名，值变化的回调使用 onChange，选中状态使用 Checked 布尔类型。这样符合用户的直观感受，更加易用，降低使用成本。

常用属性名举例如下：

属性名	类型	描述
Style	ViewStyle/TextStyle	组件样式，通常作为组件的第一个子节点的样式属性
Data	Any[]	数据源，数据源的元素通常是对象 { label: string, value: any, [props: string]: any } Label 作为展示文案，Value 作为元素唯一标志，以及其他属性
Value	Any	值
onChange	Function	值变化回调
onPress	Function	点击事件
renderItem	Function	自定义渲染项

文档

文档规定了统一的格式，旨在全方位介绍组件，方便开发者使用，格式内容如下：

- 组件名称。
- 组件描述。
- 引入方式，包括全部、按需两种引入方式。
- 示例演示，动图与静图。
- 示例代码，使用伪代码，言简意赅，能说明使用方式即可，同时，附有完整示例代码的链接。
- API 说明，分成 Props 和 Methods 两部分。
  - Props 包含 Name | Type | Required | Default | Description。
  - Methods 格式借鉴 RN 官方文档格式。

示例

beeshell 组件库遵循“关注度分离”的设计原则，对组件进行细粒度的拆分，进行了分类、分层，以及基础工具与组件实现的功能解耦。

这些设计虽然大大提升了组件库的灵活性，但是在一定程度上，对开发者提出了更高的要求。开发者需要理解各个组件与工具，灵活的组合各个元素，才能更好的完成业务需求。

为了方便开发者，更有效、合理的使用组件，我们将会实现一些经典的业务场景，以示例的形式开源出来。借助美团的平台服务，为用户提供在线演示的功能，用户可以下载美团 App（iOS 与 Android 都可以），扫描下图二维码，即可快速体验各种应用场景。

测试

代码开发的目标有两个：第一个是实现需求，第二个是保证研发质量。研发质量对公共组件库来说，尤为重要。测试是为了提高代码质量和可维护性，是实现代码开发第二个目标的一种方法。

beeshell 1.0 中已经集成了“黑盒测试”与“白盒测试”。beeshell 2.0 在原有的基础上，进行了一定程度的优化，代码的可靠性与安全性，仍然保持最高级别，而测试覆盖率则由原来的 70% 提升到了 80% 以上。

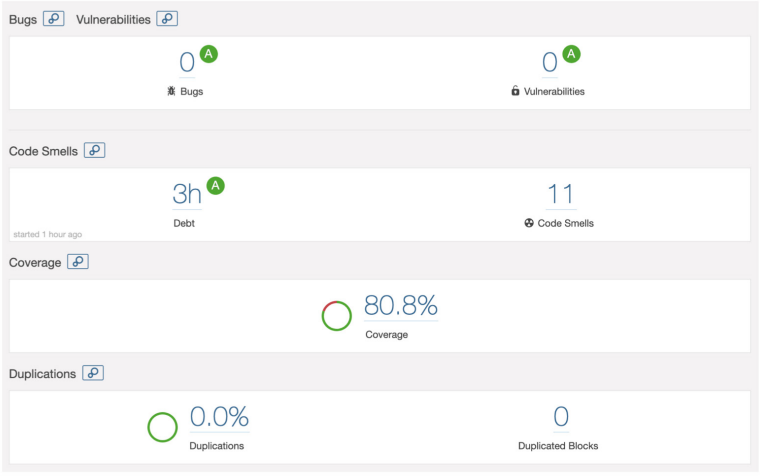
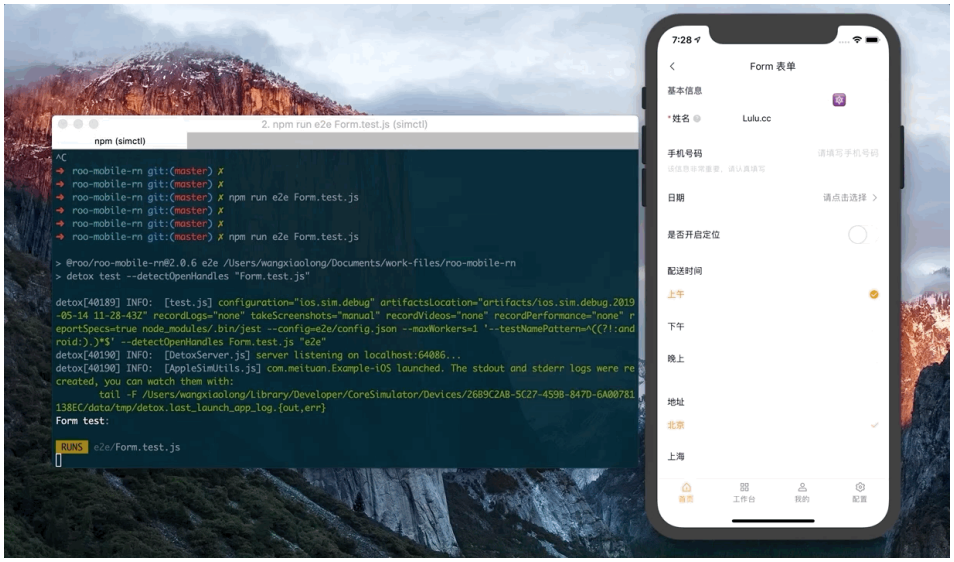


图 A

SonarQube 的分析统计结果

不仅如此，beeshell 2.0 在测试领域继续探索，集成了“灰盒测试”（基于开源方案 Detox 实现）。

> 灰盒测试，是介于白盒测试与黑盒测试之间的一种测试，灰盒测试多用于集成测试阶段，不仅关注输出、输入的正确性，同时也关注程序内部的情况。灰盒测试不像白盒那样详细、完整，但又比黑盒测试更关注程序的内部逻辑，常常是通过一些表征性的现象、事件、标志来判断内部的运行状态。



灰盒测试效果

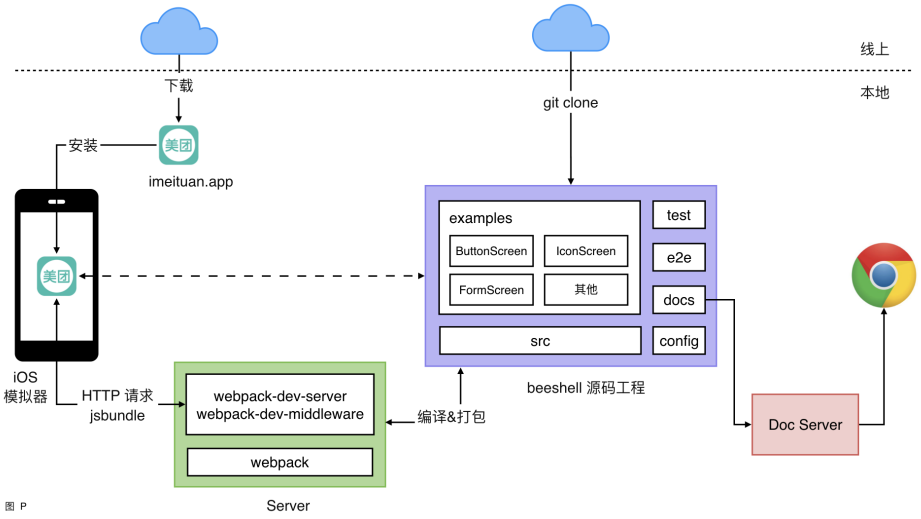
通过黑盒测试、白盒测试、灰盒测试，三种测试方案，更加全面的保证组件库的代码质量，大大提高了代码可维护性。

开发调试

受益于 MRN 平台，JS 代码与 Native 代码得以完全分离。

beeshell 源码工程，包含了包括组件源码、示例代码、测试文件在内的全部 JS 代码，Native 部分则只负责打包生成容器（本文以美团 APP 举例说明），通过下载并安装 .app（iOS）或者 .apk（Android）文件至模拟器，直接加载本地服务提供的 jsbundle，快速进入开发调试。

前端开发再也不用关心 Native 的部分，无需耗时耗力的维护 Native 环境、依赖，极大降低了前端开发成本。



开发调试流程

未来规划

我们的目标是把 beeshell 建设成为一个全面且完备的组件库，不仅会不断丰富 JS 组件，而且会不断加强复合组件去支持更多的底层功能。

我们为组件库发展规划了三个阶段：

- 第一阶段，beeshell 1.0 版本，开源 20+ 组件，主要提供基础功能。
- 第二阶段，对我们在开发 React Native 应用几年时间积累的组件进行整理，同时参考业界的标杆项目，开源 50+ 组件。
- 第三阶段，调研移动端 APP 常用的功能与场景，分析与整理，然后在 beeshell 中实现，开源 100+ 组件。

此次 beeshell 2.0 升级，共计开源 38 个功能，而且已经详细的规划了另外的 15+ 组件，也会在近期开源，目前处于第二阶段的收尾阶段。

第三阶段的建设，也在紧锣密鼓的筹备当中，要实现 100+ 组件任务十分艰巨，希望大家踊跃参与，共同建设。

开源相关

Github 地址

beeshell

核心贡献者

小龙，泽楠，轶超，宋鹏，孟谦

参考资料

- beeshell 1.0 开源推广文章
- MATERIAL DESIGN

团队介绍

外卖事业部终端团队，负责的多个终端和平台直接连接亿万用户、数百万商家和几万个运营人员，目标是在保障业务高稳定、高可用的同时，持续提升用户体验和研发效率。 – 在用户方向上，构建了全链路的高可用体系，客户端、Web 前端和小程序等多终端的可用性在 99% 左右；跨多端高复用的局部动态化框架在首页、广告、营销等核心路径的落地，提升了 30% 的研发效率；

– 在商家方向上，从提高进程优先级、VoIP Push 拉活、doze 等方面进行保活定制，并提供了 Shark、短链和 Push 等多条触达通道，订单到达率提升至 98% 以上；

– 在运营方向上，通过标准化研发流程、建设组件库和 Node 服务以及前端应用的管理与页面配置等提升 10% 的研发效率。

招聘信息

外卖事业部终端团队是由一群活力四射，对技术饱含热情，平均年龄不超过 26 岁的人共同组成。在这里，你可以看到大家对技术的追求，对产品的雕琢，对团队的认同，一切都是那么自然；在这里，你可以做一个纯粹的 FE，写写 JS，打磨一下 CSS；也可以做一个精致的猪猪女孩（男孩），优雅的调试 Android 和 iOS 代码。当然，你绝对可以做一个霸道总裁，肆意的拥抱跨端方案，Hybrid，Flutter，React Native 等，都是你的新战场。我们正在持续努力成为一个面向未来编程的团队，而这里还缺一个你。欢迎志同道合的同学发送简历到：tech@meituan.com（邮件标题注明：外卖事业部终端团队）。

React Native 在美团外卖客户端的实践

晓飞、唐笛、维康

MRN 简介

MRN（Meituan React Native）是基于开源的 React Native 框架改造并完善而成的一套动态化方案，在开发体验上基本能与原生 RN 保持一致，同时从业务需求的角度满足从开发、构建、测试、部署、运维的工程化需要。解决了一系列痛点问题：客户端版本审核及更新效率低、Android/iOS/Web 三端开发技术方案不一致、公共需求重复劳动、需求排期不敏捷、集成成本高等。目前 MRN 已接入美团数十个 App，在核心框架及生态工具上有超过百位代码贡献者，（每天）的总 PV 超过 1 亿次。

在项目成立之初，MRN 使用当时最新的 React Native 0.54.3 作为基础版本，然后进行了一系列的改造。React Native 官方稳定版已经升至 0.60.5，对 MRN 页面的质量性能、开发者体验都有了巨大的提升，包括 JSI 替换桥进行 JS 和 Native 通信、JS 引擎替换、React Hooks 等功能。最近，MRN 也做了一些升级适配和深度优化，在相关基础建设、融合过程、优化手段等方面，我们进行了很多的探索和思考，后续这些内容会陆续放出，希望能给大家一些启发。本文主要分享美团外卖 App 在业务实践和技术探索过程中的经验。

背景

美团外卖自 2013 年创建以来，一直处于高速发展期。美团外卖所承载的业务，也从单一的餐饮业务，发展到餐饮、超市、生鲜、果蔬、药品、鲜花、蛋糕、跑腿等十多个大品类业务。伴随着业务的快速发展，我们深切地感受到 3 个痛点：

（1）业务要求快速发版试错和原生迭代周期长

美团外卖业务长期使用 H5+Native 的技术栈。由于原生应用需要依托于应用市场进行更新，这样的话，每次产品的更新必须依赖用户的主动更新，使得版本的迭代

周期变得很长，无法实现快速发版快速试错的诉求。H5 虽然具备随时发布的能力，但受限于内核的影响，平台兼容性并不好，性能也较差，而且调用 Native 的能力也受限，往往只能满足一定范围内的产品需求。

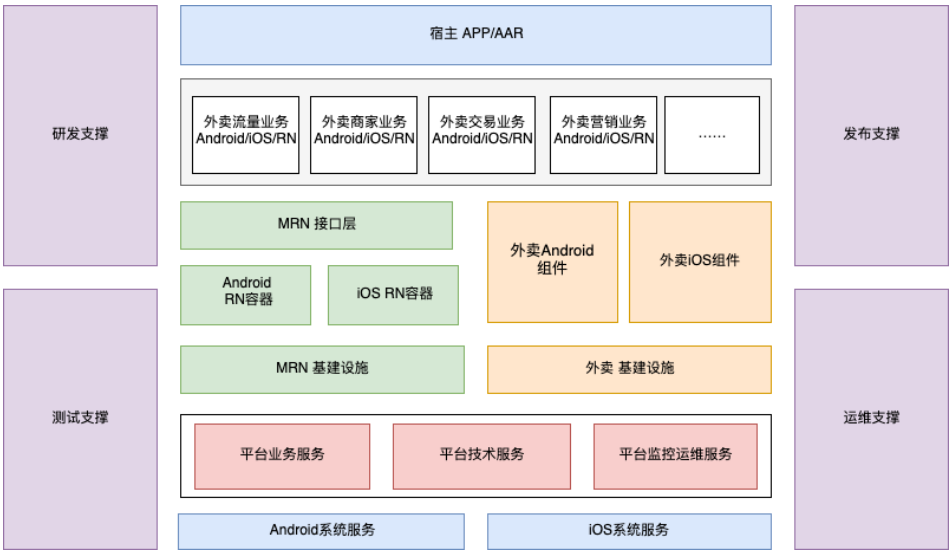
（2）有限的客户端研发资源无法满足日益增长的业务

业务的快速发展对客户端的开发效率不断提出挑战。如何通过技术手段，在有限的客户端人力资源下，支持更多的业务需求，解决有限的研发资源跟不断变大的业务需求量之间的矛盾呢？试想，如果能逐渐地磨平 Android 和 iOS 开发技术栈带来的问题，支持一套代码在 2 个平台上线，理论上人效可以提升一倍，支持的业务需求也可以提升一倍。

（3）业务持续增长带来的安装包的大幅增长

业务的快速迭代，功能的持续增加，美团外卖客户端安装包也在持续增加，从 2018 年到 2019 年，已经累计增长 140%。如果没有切实有效的技术手段，安装包将变得越发臃肿。 业务层面的这些痛点，也在不断地督促我们去反思：到底有没有一种框架可以解决这些问题。2015 年，Facebook 发布了非常具有颠覆性的 React Native（简称 RN）框架。从名字上就可以看出，这属于一种混合式开发的模式。RN 使用 Native 来渲染，JS 来编码，从而实现了跨平台开发、快速编译、快速发布、高效渲染和布局。作为一种跨平台的移动应用开发框架，RN 的特性非常符合我们的诉求。我们也在一直积极地探索 RN 相关的技术，并且基于 RN 在脚手架、组件库、预加载、分包构建、发布运维等多个维度进行了全面的定制及优化，大幅提升了 RN 的开发及发布运维效率，还打造了更适应于美团的 MRN 技术体系。从 2018 年开始，美团外卖客户端团队开始尝试使用 MRN 框架来解决业务层面的一系列问题。经过一年多的实践，我们积累了一些经验和结论，希望相关的经验和结论能够帮助到更多的个人或技术团队。

外卖混合式架构



上图是外卖 App 引入 MRN 后的架构全景图，接下来我们会从下到上、从左到右逐步介绍：

- 最下层是 Android/iOS 系统服务层，因为 MRN 是跨端的，所以需要引入这一层。相对单一平台来说，由于 MRN 的引入，整个 App 的架构不可避免地需要考虑 Android 和 iOS 平台本身的差异性。
- 倒数第二层是平台服务层，这一层相对与单一平台来说，并没有太大区别。
- 再往上一层是 MRN 基建层，这一层的工作主要是：(1) 尽可能地屏蔽 Android 和 iOS 系统的差异性；(2) 打通已有的平台基建能力，让上层业务不能感知到差异。
- 再上一层是业务组件层，这一层相对于单一平台来说，区别不大，主要是增加了 Android 和 iOS 的 RN 容器，同时业务组件是可以被 RN 调用的。
- 继续往上是 MRN 接口层，该层的主要任务是尽可能地屏蔽 Android 和 iOS 组件之间的差异，让上层页面使用的 RN 接口保持一致。
- 最后是业务层，这一层是用户可直接接触到的页面，页面的实现可以是

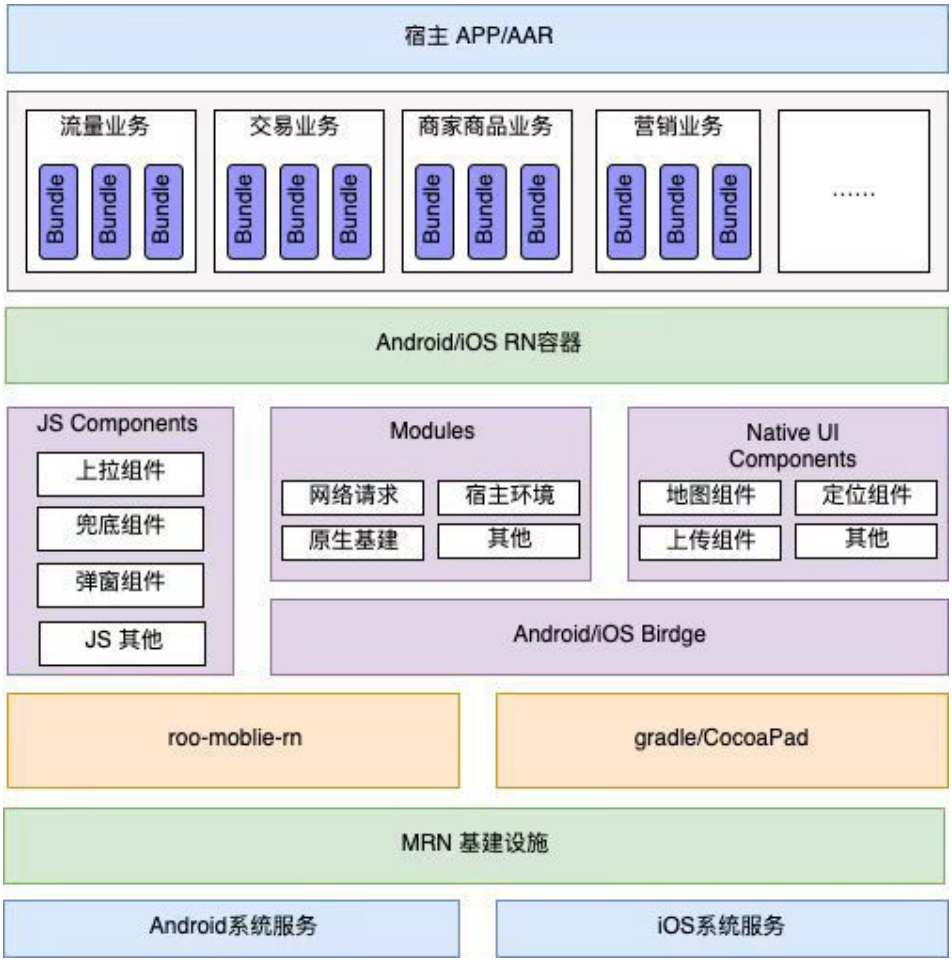
Android/iOS/RN。

- 左上角是研发支撑，主要包括代码规范、代码检查工具、Debug 插件、准入规范、准入检查工具、代码模板插件等。这块相对于单一平台来说，主要的差异体现在：由于编译器和语言不同，使用的工具有所区别，但工具要做的事情基本是一致的。
- 左下角是测试支撑，主要包括 UI 自动化测试、自测覆盖率检查、AppMock 工具、业务自测小助手、性能测试、云测平台等。这块相对于单一平台来说，基本也是一致的，主要的差异同研发支撑，主要是语言不同，使用的工具有所区别。
- 右上角是发布支撑，主要包括打包 Bundle 和 APK、打包检查、发布检查、发布 Bundle 和 APK 等。这块相对于单一平台来说，保持了打包发布平台的一致性，区别在于：需在原有的基础上，增加 MRN 的打包发布环节。
- 右下角是运维支撑，主要包括基建成功率监控、业务成功率监控、线上问题追踪、网络降级等。这块相对于单一平台来说，保持了一致性，区别在于：需在原有的基础上，增加 MRN 的监控运维。

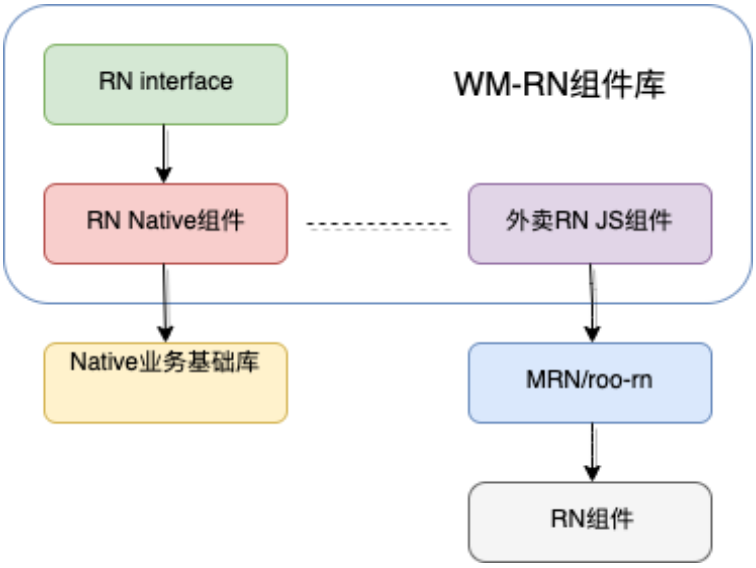
研发测试支撑

外卖业务 MRN 组件架构

RN 官方对双端只提供了 30 多个常用组件，与成熟的 Native 开发相比，天壤之别。所以我们在开发的过程中面临的一个很重要问题就是组件的缺失。于是，MRN 团队基于 RN 组件进行了丰富，引入了一些优秀的开源组件，但是源于外卖业务的特殊性，一方面需要业务定制，另一方面部分组件依然缺失。所以为了减少重复代码，提升外卖客户端 MRN 的研发效率，建设外卖组件库就变得非常有必要。



上图是我们外卖组件库的架构图，最底层依赖 Android 和 iOS 的原生服务；然后是 MRN 基建层，用于抹平 Android 和 iOS 系统之间的差异；再上一层则是外卖组件库及其依赖，如平台组件库和打包服务，组件库分为两类：纯 JS 组件和包含 JS 和 Native 的复合组件。再上一层则是 Android 和 iOS 的 MRN 容器，它提供了上层 Bundle 的运行环境。整个组件的架构思路，是利用中间层来屏蔽平台的差异，尽可能地使用 JS 组件，减少对原生组件的依赖。这样可以有效地减少上层业务开发时对平台的理解。接下来，我们主要讲一下 WM-RN 组件库：



如上图所示，WM-RN 组件库主要包含三部分：RN interface、RN Native 组件、外卖 RN JS 组件。RN Interface 主要包括 Native 组件的 Bridge 部分和 Native 组件在 JS 侧的封装，封装一层的好处是方便调用 Native 暴露出的接口，也可以用来抹平 Android 和 iOS 系统间的差异；RN Native 组件分为 Android 和 iOS 两端，依赖各自的业务模块，为 RN 提供外卖 Native 的业务能力，如购物车服务、广告服务；外卖 RN JS 组件则是纯 JS 实现，内部兼容外卖 App 与美团外卖频道间的差异、Android 和 iOS 平台间的差异，依赖现有的 MRN 组件库和外卖开源 Beeshell 组件库，减少组件的开发成本；从工程的物理结构来看，建议将 Native 组件、RN Interface 放在一个仓库进行管理，主要是因为 Native 与 JS 侧的很多通信都是通过字符串来匹配的，放在一起方便双端与 JS 侧的接口统一对齐，发布时也会更加方便。目前，外卖组件库已经扩展了几十个业务组件，支持了线上近百个 MRN 页面。

Native/MRN/H5 选型标准

目前，美团外卖 App 存在三种技术栈：Native、MRN、H5，面对业务持续增长和安装包不断变大的压力，选择合适的技术栈显得尤为重要。H5 在性能和用户体验方面相比 Native 和基于 Native 渲染的 RN 相对弱一些，所以目前大部分 H5 页面只

是用来承载需求变更频繁、需要即时上线的活动页面。那么 MRN 和 Native 的界限是什么呢？当有一个新的页面产生时，我们应该如何做取舍？通过实践，我们逐渐摸索了一套选型规则，如下：

- Native 选型规则，强交互（同时存在 2 种及以上手势操作），无法用二元函数描述的复杂动效，对用户体验要求极致的页面，类似首页、点菜页、提单页等。
- 对于强交互或强动画，MRN 技术栈支持效果不理想，不建议使用。其他情况下，建议使用 MRN。
- H5 适用于需要外链展示的轻展示页面，比如向外投放活动的运营页面等等。

具体选型细节可参考下表：

需求类型	细类	举例	Native	MRN	H5
展示类	静态展示类	列表、纯展示	不推荐	推荐	不推荐
	简单交互页	交互简单，业务复杂	不推荐	推荐	不推荐
	千人千面		不推荐	推荐	不推荐
交互类	复杂交互功能（可以RN配合Native）	键盘、嵌套滚动、频繁精细滑动、滑动冲突	推荐	不推荐	不推荐
	一般动画	轮播图、弹窗、进度条等	不推荐	推荐	不推荐
	精细动画	下拉刷新动画、加载动画	推荐	不推荐	不支持
组件类	通用业务类组件	商家相册	不推荐	推荐	不推荐
	页面内多个Native组件	公用部分不多，大部分都是Native组件堆砌的	推荐	不推荐	不支持
	原生能力组件	图片选择、地图定位等	推荐	不推荐	不支持
全局模块	高性能模块	购物车拖拽、用户频繁加减菜	推荐	不推荐	不支持
	强交互模块	智能点餐	推荐	不推荐	不支持
外链	需要外链展示	需要嵌入没有MRN环境的App	不支持	不支持	推荐

发布运维支撑

发布运维是一个成熟的软件项目中非常核心的部分，它保证了整个项目能够高效且稳定地运转。建立一个稳定可靠的发布运维体系是我们建设整个外卖 MRN 技术

体系的重要目标。但发布运维的建设上下游牵扯了众多基建：拥有一个合理的工程结构对发布运维来说至关重要。如果工程结构臃肿且混乱，将会引起的一系列的权限问题、管理维护问题，这样会严重制约整个发布运维体系的效率。所以 MRN 的工程架构演进优化也是发布运维体系建设的重要组成部分。

MRN 分库 & 工程结构演进

业务分库

任何一个大型、长期的前端技术项目，良好的工程结构都是研发发布支撑中非常核心的部分。从 2018 年 10 月份，外卖正式启动 MRN 项目以来，面临涉及近百个 MRN 和几十人参与的大规模 MRN 应用计划。从项目初期，我们就开始寻找一个非常适合开发维护的工程结构。

在最开始的时候，我们的目标是快速验证及落地，使用了一个 Git 库与一个 Talos 项目（美团自研发布系统）去承接所有页面的开发及发布工作，同时对权限进行了收缩，保证初期阶段的安全发布。然而随着页面的增多，每个版本的发布压力逐渐增大。发布 SOP 上的三大关键节点权限：Git 库操作权限、Talos 的发布权限、美团自研的线上降级系统 Horn 权限，互不相关，负责人也各异，导致发布时常因各个节点的权限审批问题，严重阻塞效率。

随着项目的大规模铺开，我们的页面数量、合并上线次数与初期已不可同日而语。为了解决逐渐臃肿的代码仓库问题及发布效率问题，我们将庞大而臃肿的 RN 库根据业务维度和维护团队拆分成了 4 个业务库，分别是订单业务、流量业务、商家业务、营销业务，并确认各库的主 R，建立对应的 Talos 项目，而主 R 也是对应 Talos 项目的负责人。同时所有的主 R 都有 MRN 灰度脚本的管控权限。这样一来，MRN 的工程结构和 Native 的工程结构完全对齐，每个责任人都非常明确自己的职责，不会来回地穿插在不同的业务之间，同时业务库任意页面的发布权限都进行了集中，RD 只需要了解业务的负责人，即可找到对应的主 R 完成这个业务的所有相关工作。

工程结构

在项目初期，对于每个库的工程结构，美团内部比较流行的工程结构有两种：一个是适合小型业务开发的单工程多 Bundle 方案，另一个是相对更适合中大型业务开发的多工程多 Bundle 方案。

单工程单 Bundle 方案

顾名思义，单工程单 Bundle 方案的意思就是一个前端工程承载所有的业务代码，最终的产物也只有一个 RN Bundle。通过入参决定具体加载哪个页面。

对于业务不多，参与人不多的团队，使用单工程单 Bundle 的方式即可快速完成开发、发布。因为通过一次发布就可以完成整个发布的工作，但是带来的弊端也是不可接受的：因为所有业务都耦合在一起，每次更新都会“牵一发而动全身”，增大了问题的隐患。如果多个业务需求同时提测的时候，在团队配合上也是一个极大的挑战，因为新版本号会覆盖旧版本号，导致两个需求提测时会出现相互覆盖的情况。所以我们在立项之初就排除了这种方案。

多工程多 Bundle 方案

多工程多 Bundle 方案的意思就是一个 Git 库中存放了多个页面文件夹，各个文件夹是完全独立的关系，各自是一个完整的前端工程。拥有自己独立的 MRN 配置信息、package.json、组件、Lint 配置等（如下图所示）。每个页面文件夹都输出一个独立的 RN Bundle。

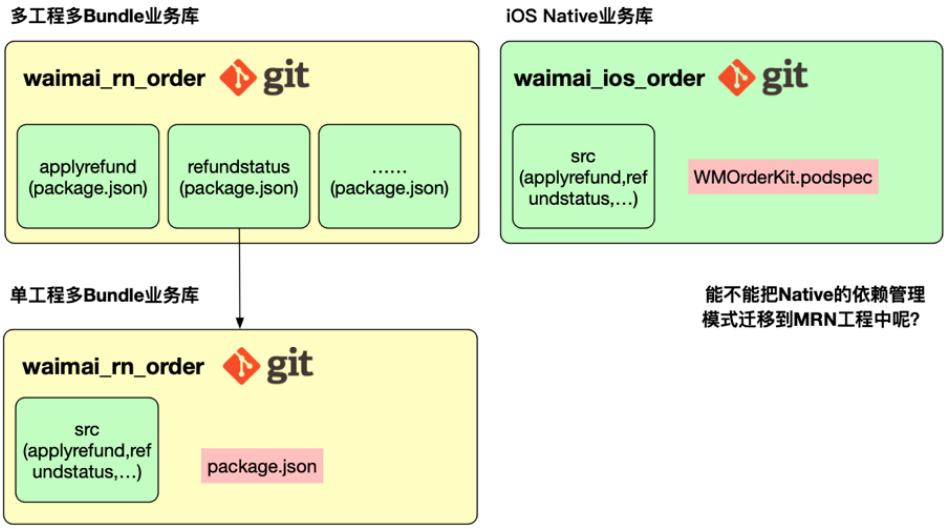
相比于单工程单 Bundle 方案，多工程多 Bundle 方案将页面进行解耦，使之基本可以满足中大型 MRN 项目的需求。在外卖 MRN 项目初期，一直都使用着这样的工程结构进行开发。但是我们也为之付出了相应的代价，即每个页面的依赖都需要对应 RD 去维护升级，依赖碎片化的问题日趋严重。同时在工程级别的管控，如统一 Lint 规则、Git Hook 等也变得更加复杂。

多工程多 Bundle 方案 => 单工程多 Bundle 方案

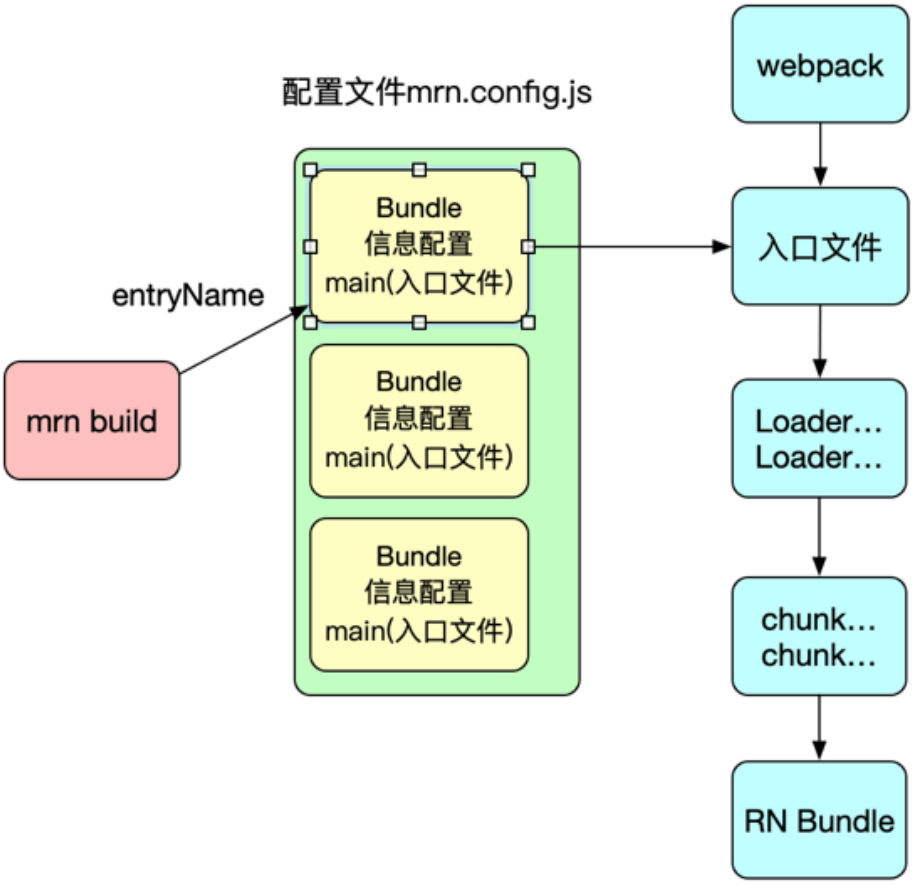
随着外卖 MRN 页面规模以及参与人规模的进一步增大，多工程多 Bundle 方案的缺点日益凸显。特别对于那些前端技术底子相对薄弱的团队来说，依赖管理问题会

变得很头疼。在这种情况下，单工程多 Bundle 的方案就应运而生了。

核心思路也很简单：观察一下单工程单 Bundle 方案和多工程多 Bundle 方案的优缺点可知，单工程单 Bundle 依赖管理方便的优点主要来自于“单工程”，而多工程多 Bundle 的业务解耦的优点主要来自于“多 Bundle”。所以结合这两种工程方案的核心优点，就可以设计一种新方案：单工程多 Bundle。即用一个工程去承接所有的页面代码，但是又可以让每个页面输出独立的 RN Bundle 来保证互不影响。其实，这种方式类似于 Native 一个静态库的管理，如下图所示：



通过分析 MRN 的打包原理可知，MRN 通过一个配置文件配置了一个 Bundle 的所有业务信息以及 mrn-pack2 的打包入口。所以我们只需要让配置文件支持多份 Bundle 信息的配置，通过打包命令与参数选择正确的 mrn-pack2 打包入口，即可打出我们最终所需要的业务 Bundle。如下图所示：



核心优势：

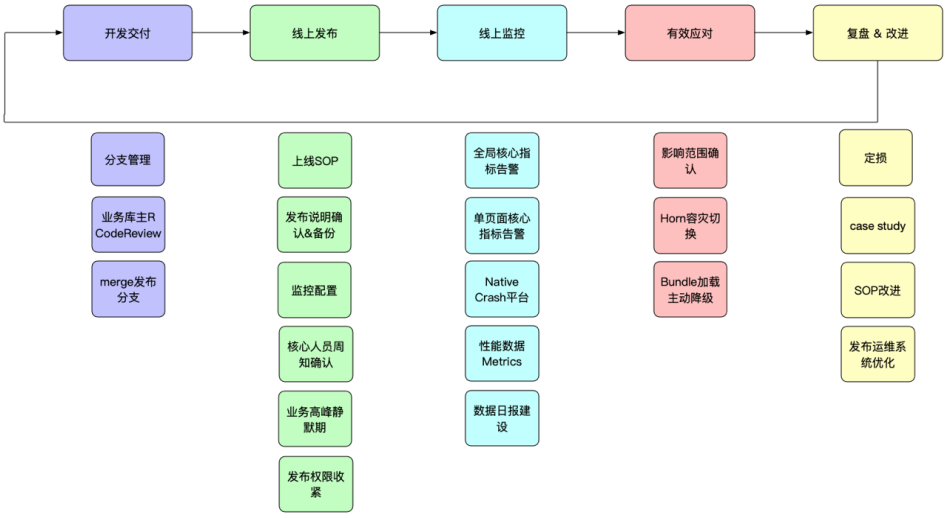
- 整个工程采用一个 package.json，管理业务库中所有的依赖。这样可以有效地解决各自页面去管理自己依赖时，必然产生的依赖版本碎片化问题，避免同一依赖库因为版本不一样，而导致页面表现不一样的问题。
- 从依赖角度去规范各自页面的使用工具规范，如 A 页面使用某一种三方库来实现某种功能，B 页面使用另一种三方库也实现了同一种功能，单一依赖管理就可以从库依赖的角度强制做技术选型，减少各个页面的实现差异，从而降低维护成本。
- 让业务同学可以更加专心地开发业务代码，不用关心复杂的依赖问题，大大提升了开发效率。

实现了工程级别的管控，如 Pre-Commit，脚手架方案管理将变得更加便捷。

这种工程组织形式也成为了 MRN 工程结构的最佳实践，而且美团内部也有多个团队采用了这种解决方案。目前已支撑超过几百个页面的开发和维护工作。

外卖发布运维体系

下图展示了我们的发布运维全景，共覆盖了开发交付、线上发布、线上监控、有效应对、复盘改进等五大模块。接下来我们会逐一进行介绍。



（1）开发交付

开发阶段，需求 RD 完成开发，提交到 Git 库的发布分支。对应的业务库主 R 角色（通常由 RN 经验较丰富的工程师来承担）进行 CodeReview，确认无误之后会执行代码的合并操作。顺便说一下，这也是外卖 RN 质量保障长征路的第一步。

（2）线上发布

合入发布分支之后，就可以正式启动一次 RN Bundle 发布。这里我们借助了美团内部的 Talos 完成整个发布过程，Talos 的发布模板与插件流水线规范了一次发布需要的所有操作，核心步骤包括发布准备（Git 拉代码、环境参数确认、本次发布说明填写）、发布自检（依赖问题检查、Lint、单元测试）、正式打包（Build、版本号自更新）、产物上传测试环境（测试 / 线上环境隔离、测试环境进行测试），双重确认

（QA、Leader 确认发布）、产物上传线上环境等等。

产物上传线上环境，实际上是上传到了美团内部的 CD 平台 –Eva。在 Eva 上，我们可以借助 RN Bundle 的发布配置去约束发布 App 的版本号、SDK 版本等，以及具体的发布比例及地区，去满足我们不同的发布需求。最终执行发布操作，将 RN Bundle 上传到 CDN 服务器，供用户下载，完成整个发布流程。

（3）运维监控

发布之后，运维是重中之重。首先我们的运维难点在于我们的业务横跨两个平台——美团 App 与外卖 App。由于它们在基建、扩展、网络部分都存在差异，所以我们选取指标的维度不仅要从业务出发，还要增加全局的维度，来确保外卖平台 MRN 的正常运转。基于这个层面的思考，我们选取了一系列 RN 核心指标（在下面的章节会详细列举），进行了全方位的监控。目前外卖客户端，已经做到分钟级监控、小时级监控和日级别监控等三档监控。

在监控手段上，首先我们使用了美团开源的 Cat 告警平台（这部分已经通过 Talos 插件完全自动化配置），确保当核心指标在线上出现波动、异常的时候，相关 RD、QA 以及业务负责人可以及时接受到报警，并由对应的 RD 主 R 负责，快速进入到“有效应对”的环节。同时为了能够分阶段、更好地处理问题，我们将核心指标报警分为【P1】与【P0】两个级别，分别代表“提高警觉，确认问题”与“大事不好，马上处理”。保障了一个问题出现之后能够及时被发现并快速进行处理。

除了监控报警手段之外，我们还会借鉴客户端高可用性保障的经验。用一些日常运维的手段去发现问题。比如使用灰度小助手、数据日报等手段从宏观角度主动去发现存在隐患的指标，及时治理，避免问题。

（4）有效应对

根据“墨菲定律”：如果事情有变坏的可能，不管这种可能性有多小，它总会发生。即便我们在发布管控和线上监控上做的再充分，线上问题最终还是无法避免的。所以当通过线上告、客诉等手段发现线上问题之后，我们需要及时的应对问题、解决问题，把问题带来的影响降低到最小，并以最快的速度恢复对用户的服务。

在有效应对的方面，我们主要靠两种手段。第一种是存在 B 方案兜底的情况，使

用 Horn 灰度配置，关掉 MRN 开关，短时间内恢复成 Native 页面或者 H5 页面继续为用户提供服务，同时通知相关 RD 和 QA 快速定位问题，及时修复，验证并上线。第二种是无兜底方案的情况，CDN 服务器（Eva）上撤掉问题 Bundle，实现版本回滚，接下来的问题定位过程跟手段保持一致。

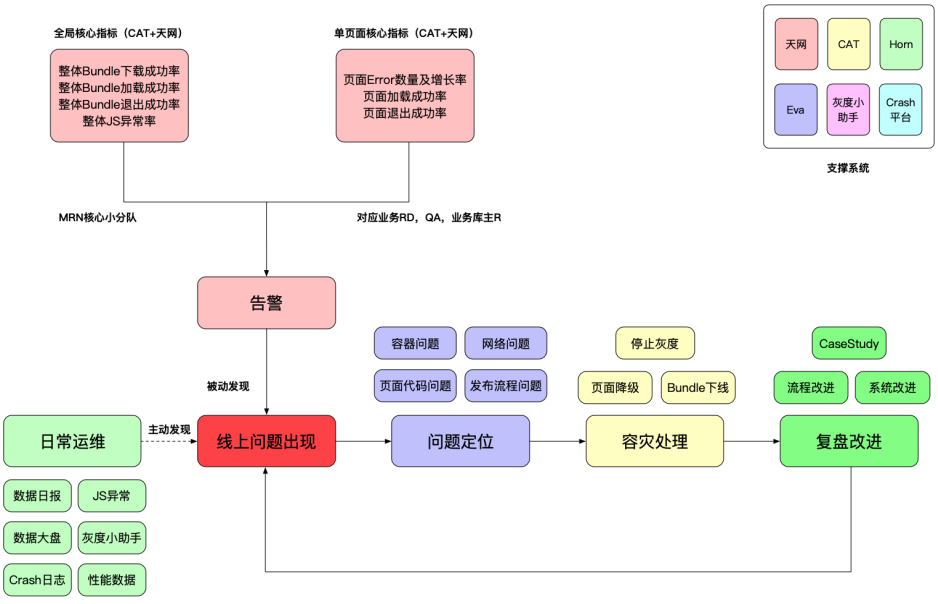
这两种备案保障了外卖 MRN 业务的整体高可用性。

（5）复盘改进

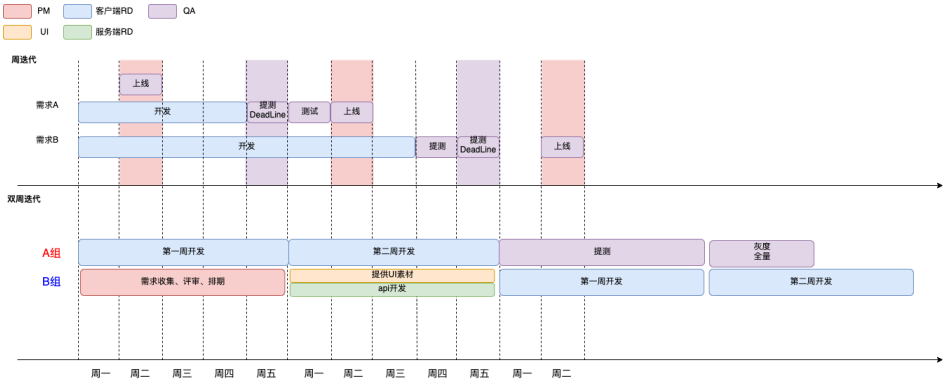
在以上四个大环节中，问题可能会出现在任意一个环节。除了及时发现问题与解决问题，我们还需要尽力避免问题。这一点主要是靠我们内部的例会、复盘会，对典型问题进行 Review，将问题进行归类，包括复盘流程规范问题、操作失误问题、框架 Bug 等，并力图通过规范流程、系统优化来尽力地避免问题。

在外卖 MRN 项目实施过程中，我们共推动了二十多项规范流程、系统优化等措施，大大保障了整体服务的稳定性。

最后，我们用一张图对外卖监控运维体系做一个总结，帮助大家有一个全局的认知。



混合式架构流程



针对混合式架构的流程，目前外卖技术团队采用的是正常双周版本迭代流程 + 周迭代上线流程。MRN 页面既可以跟版迭代，也可以不跟版迭代，这样可以有效地减少流程的复杂度和降低 QA 的测试成本，而周迭代流程可以有效地利用 MRN 动态发版的灵活性。混合式开发和原生开发应尽量保持时间节点和已有流程的一致。这种设计的好处在于，一方面随着动态化的比例越来越高，版本迭代将可以无限拉长，另一方面从双周迭代逐渐演变成周迭代的切换成本也得到大幅的降低。详细可分为下面几个阶段：

评审阶段

业务评审阶段在原有的流程上，增加了技术选型阶段。在技术选型时，明确是否会存在需要使用 MRN 页面的情况，如果页面可以完全不涉及到 Native 部分即可完成，就可以进入周迭代的发版流程。如果需求用 MRN 实现，但是又涉及到 Native 部分，仍然走周迭代的上线流程。除正常开发需求的时间外，RD 需综合考虑到双端上的适配成本。

开发阶段

客户端以周维度进行开发，每周确定下周可提测的内容，根据提测内容是否为动态化的业务、下周是否在版本迭代周期内，决定跟版发布或周发布。

提测阶段

提测前，为了保证 MRN 页面的提测质量，RD 首先需要按照 QA 提供的测试用例提前发现适配问题。提测时需要在提测邮件中注明：（1）提测的 Bundle 名称和对应的版本号 ；（2）标明哪些组件涉及 Native 模块 ；（3）依赖变更情况，如是否升级了基础库，升级后的影响范围；（4）重点测试点的建议。

上线阶段

MRN 由于其可动态发布的特性可以跟版发布，也可不跟版发布，但上线时间和灰度时间节点都保持了一致。不过版本还是动态发版，都默认周二上线，周四全量。

- 跟版发布：默认只对当前版本生效，需在双周迭代三轮提测节点，周二当天将 Bundle 上线服务器，MRN 的灰度开关全量打开。通过周四 App 的发版灰度比例来控制 MRN 的灰度比例，上线时需配置报警和灰度助手监控，实时掌握 MRN 的线上数据。
- 不跟版发布：也同样以周四作为全量发布窗口，Bundle 需在周二时上线指定线上版本，指定 QA 白名单。测试通过后，在周三按照比例逐步灰度，周四正式全量，和跟版发布一样，上线时需要配置报警和监控。

架构总结

引入 MRN 后，相对单平台而言，架构层级上，我们增加了 2 个 MRN 中间层去屏蔽 Android 和 iOS 平台、原生组件之间的差异。这样做的目的是为了让上层业务开发者可以很快地使用框架进行业务开发，完全不用关心平台和组件间的差异。通过引入 MRN 技术栈，带来的好处很明显：

（1）使用 MRN 实现的页面理论上可以实现一套代码，部署到不同平台上，开发效率得到大幅度提升。（2）采用 MRN 框架，无论是加载性能还是页面滑动性的用户体验上，都会比原来 H5 的方式要好。（3）部分页面具备了快速编译、快速发布的能力。

但一个硬币总有两面，混合式架构增加了架构的复杂度，使得原本只要考虑一个平台的事情，逐渐转变成需要考虑三个平台，另外 Android 本身具备碎片化的问题，

这使得混合式架构的适配问题较为突出。当出现问题时，我们的第一反应由“这是什么问题”变成“它是否存在于两个平台，还是只在一个平台上?”、“如果仅在一个平台上，是在原生代码还是 React Native 代码出了问题?”、“历史版本的 MRN 是否存在问题，是否需要修复”、“修复的效果在 Android 和 iOS 上的表现是否一样”，这些问题增加了定位和修复工作的复杂性。另外，MRN 的适应场景也是有限的，并非所有的业务和页面都适合改造成 MRN，如何做选择也需要进行有效的判断，从而增加了决策成本。

针对上述问题，我们的建议是：

（1）减少分歧

- 在研发、测试、发布和运维环节，MRN 的页面尽可能对齐 Native 原有的环节，减少团队理解的成本。
- 在 Debug 开发环境下，利用页面浮层提示技术栈使用情况；Release 环境下，利用工具、MRN 自动化报表，及时的让开发同学明确知道是 Native 页面还是 MRN 页面，减少确认。
- MRN 页面尽可能地避免原生组件的使用，而使用纯 JS 代码实现，供 MRN 页面使用的原生组件的需要高质量的提供，减少下层组件的问题。
- 默认只修复当前的版本，出现严重问题时才考虑修复历史版本，减少多版本带来的复杂度提升。

（2）技术栈明确边界

- 做好 Native 和 MRN 技术栈使用的边界，尽可能用简单的选型标准，让合适的场景选用合适的技术栈，从而保证业务整体的可用性，让用户体验依然如初。

（3）单技术栈转向多技术栈团队

- 培养全栈工程师，当团队的同学都具备 iOS、Android 和 MRN 多个技术栈能力时，将会有效地提升开发的效率，短期内可选择 iOS、Android 和 MRN 工程师结伴编程的策略。

可用性体系

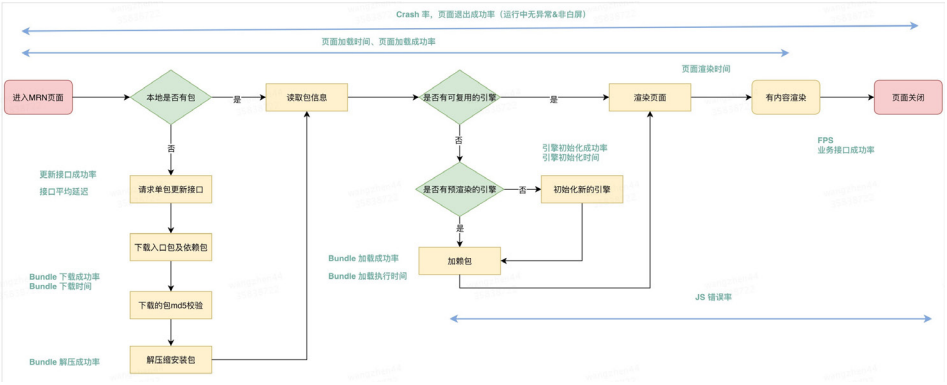
正如在“监控运维”章节中所讲到的那样，线上运维是我们工作的重中之重。这个章节我们就讲一下我们对于监控指标的选取。鉴于外卖业务的特殊性，除了美团的外卖频道之外，外卖业务还需要运行在独立的外卖 App 上。如下图所示：

外卖 App 经过多年的发展，目前已逐渐成为一个平台级应用，承接了 C 端、闪购、跑腿等多个业务。与美团 App 相比，它们之间在很多基础建设、扩展、网络部分都存在差异。所以在监控核心指标的选取上，我们除了保证 C 端 MRN 业务在美团以及外卖两端的高可用性，还需要保证外卖 App 平台本身基建的稳定性，从而保证运转在外卖 App 上所有 MRN 业务的高可用性。

而从监控的大分类上来讲，我们分为了【 可用性指标 】以及【 性能指标 】，它们分别关注业务本身的可用性，以及页面的性能与用户体验。接下来，我们就依次进行讲解。

MRN 可用性指标

可用性指标也是我们关注的关键指标，它直接决定了我们的 MRN 页面是否能够正确、稳定地为用户提供服务。通过 MRN Bundle 加载全景，我们可以确定整个包加载的几个关键节点。可以说，MRN 业务的可用性就是取决于这些关键节点的成功率。



下载链路

MRN 是一个动态化的框架，所有的 MRN Bundle 都是从 CDN 节点上远程下载。所以下载成功是 MRN 业务可用的先决条件。有些普通的业务方是不需要关注这个指标的，而外卖 App 可能会因为网络库基建，出现启动下载线程拥堵、DNS 劫持等问题，所以我们把下载成功率作为外卖 App 监控的全局指标。目前，外卖 App 的下载成功率长期稳定在 99.9% 左右。

加载链路

加载链路可以细分为初始化引擎部分以及业务 Bundle 加载部分。前者跟基建有关，代表从引擎创建到加载完 Common 包加载成功这段的成功率。这部分主要依赖 MRN SDK 的稳定性，从我们的日报上看，稳定性基本保持在 99.99% 以上。

而业务 Bundle 加载成功率（MRN PageLoad Success），是 MRN 页面创建到业务视图内容渲染过程中，没有发生错误的比例。它与跟拉包时网络情况、MRN 框架稳定性和业务 JS 代码都有关系。这也是我们关注的核心指标，因为它直接决定了我们某个页面是否可以渲染成功，所以我们把这个指标同时列为了外卖 App 监控告警的全局指标与单 Bundle 告警的指标。目前，整个外卖业务的 Bundle 加载成功率稳定在 99.9% 以上。

使用链路

Bundle 加载成功之后，页面成功被渲染。但是在使用的过程中，可能会因为 JS

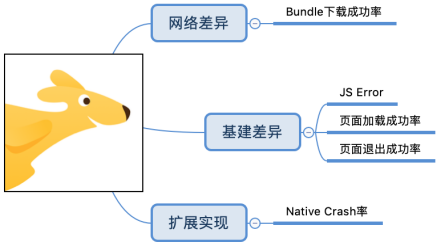
代码，Native 代码的 Bug 出现 JS Error、Native Crash 等问题，这样给用户带来的直观反馈就是应用闪退、页面白屏等，造成了服务的不可用。所以在使用链路上出现问题率，基本也可以直观反映出一个 RN 页面的质量以及它当前的运行状况。

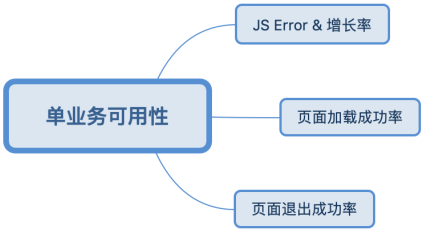
在使用链路上，我们主要关注的是 JS Error 率、JS Error 个数以及页面退出成功率（MRN PageExit Success）等。

JS Error 很好理解，由于 RN 是由 JS 驱动的框架，所以一个页面的 JS Error 率基本上可以综合反映出一个页面的可用性、稳定性或者基建的稳定性，故我们同样把这个指标同时列为了外卖 App 监控告警的全局指标与单 Bundle 告警的指标。我们用上报上来的 JS Error 数量做分子，该页面的 PV 做分母，计算一个页面的 JS 错误率，当 JS Error 个数短时间内极速升高或者 JS Error 率有大幅上升时，就会触发我们的 JS Error 告警。目前外卖大盘的 JS Error 率保持在万分之一左右，略低于 Native Crash 率。

页面退出成功率（MRN PageExit Success），理解起来不如前面的指标那么简单，因为它表示的是用户在退出 MRN 页面时，业务视图内容已成功渲染的比例。它会包含所有已知和未知的异常，但是用户进入页面后快速退出的场景，也会被错误的统计在其中，因为用户退出时可能页面尚在加载中。相比于 JS Error，它是一个更加综合的指标，基本上涵盖了加载失败、渲染白屏、使用时出现错误等多个异常场景，基本上可以反映出一次 MRN 业务的单次可用性，相比于之前的指标会更加严格。我们把这个指标同时列为了外卖 App 监控告警的全局指标与单 Bundle 告警的指标。我们希望它永远能保持在 99.9% 以上，否则就会触发告警。目前外卖大盘的 MRN PageExit Success 基本稳定在万分之三左右，我们最终的目标是希望稳定在万分之一左右。

最后，我们希望通过两个“脑图”快速回顾一下外卖全局监控与单业务监控关注的核心指标。





全局监控　　单业务监控

MRN 性能指标

除了可用性指标，性能指标也是我们重点关注的内容。如果加载时间过长，就会大大增加用户离开页面的概率。而页面卡顿，也会影响用户在使用层面的体验，从而引发客诉或者业务损失。

根据 Bundle 加载全链路图，我们也可以把性能指标分为两个大类，一个是加载时耗时与使用时性能指标。前者主要关注 Bundle 从 Load 到渲染整个链路的耗时，后者主要关注使用时的性能指标，在这里主要是指页面的 FPS。

加载链路耗时

如上述所说，整个加载链路分为引擎初始化的时间以及 Bundle 本身加载及渲染的时间的时间。

引擎初始化的时间在整条链路上占比是最长的，因为初始化的时候会加载比一般业务代码大得多的 CommonJS。经过观察，这部分的时间总体表现较差，在 iOS 上 50 分位和 90 分位分别是 0.3s 和 0.7s。在 Android 上表现更差，50 分位和 90 分位分别是 1.3s 和 1.8s。不过目前 MRN 已经使用了预加载方案，即在 App 刚启动时就初始化一个 JS 引擎，等实际使用时，直接复用该引擎即可，大大缩短了首次 Bundle 的整体加载时间。

页面加载时间和页面渲染时间是我们关注的第二类指标，从加载链路图也可以发现，页面加载时间代表从开始加载 Bundle 到 RN 内容渲染成功的整条时间，而页面渲染时间则是它的子集，代表 Bundle 解析完毕，从 JS StartApplication 开始加载

组件到渲染出第一帧的时间（iOS 和 Android 的统计口径不同）。区分这两项指标也可以更好地分析整个加载链路上的瓶颈在哪，有助于针对性的做性能优化。

以外卖 iOS 50 分位为例，我们发现页面整体的加载时间在 400ms 左右，JS 渲染时间只需要 100ms 左右，主要的性能瓶颈在 Bundle 加载以及 JS Bundle 的解析部分，这也是我们接下来需要重点研究课题。

使用时 FPS

衡量用户使用体验比较直观的一个指标就是 FPS，较高的 FPS 会让用户更加顺畅地体验功能，完成操作。

目前，MRN 在外卖侧业务总体落地页面复杂度适中，遇到复杂动画也使用了 BindingX 来提升性能。通过监控，外卖侧的页面总体表现良好，在 iOS 上几近满帧，在 Android 上表现稍差，平均在 55 帧左右，较深的视图层级与较低的 JS-Native 的通信效率都是 MRN FPS 的杀手。如何提升 MRN 特别是在 Android 上的页面性能也是我们下一阶段研究的课题。

目前，外卖性能指标 50 分位的性能指标基本满足线上需求，但是 90 分位的表现不尽如人意，特别是较低的 FPS 以及过长的页面加载时间。革命尚未成功，同志仍需努力。

效率衡量

引入 MRN，提升了本地的开发效率，但同时也增加了工程的复杂度，所以总体来说真的能提升实际开发效率吗？在完成几十个 RN 页面的开发后，总结了一些公式，希望可以给其他团队一些结论性的参考。首先设定三个方面去考量：人效提升、代码复用、维护成本衡量，将外卖的所有 MRN 页面加在一起，取平均值，可以得出较为准确的结论：

- 人效提升计算公式：∑（Android Native 总人日 +iOS Native 总人日 −RN 总人日）/ ∑（Android Native 总人日 +iOS Native 总人日）
- 代码复用率计算公式：∑（RN 行数 − 平台分支判断代码块）/ ∑（RN 行数 +Android native+iOS Native）

- 维护成本计算公式：∑（Android Native 原生总行数 +iOS Native 页面总行数 −RN 页面总行数）/ ∑（Android Native 页面总行数 +iOS Native 页面总行数）

根据页面的交互程度去进一步的划分，得到如下的表格：

页面	人效提升	代码复用率	维护成本
纯展示页面	60%	100%	81.77%
简单交互页面	58%	88.69%	55.16%
复杂交互页面	55%	73.99%	46.60%
总计	57.70%	87.56%	61.18%

如表所示：人效提升的方面，主要取决于页面是否存在复杂的交互，如果页面存在复杂交互，就会不可避免的导致涉及到 Native 的双端原生开发，如部分交互需要 Native Module 实现，最终的人效提升将大打折扣。而对于涉及较少的 Native Module 和展示型的页面，MRN 存在较大优势。但大家会很奇怪这种结果，为什么人效提升会大于 50%？ 逻辑上 Android 和 iOS 双端复用后，提升的效率理论上最大应该是 50%。这是由于 RN bundle 的热加载极大地节省了 Native 的编译时间，这一部分相对原生开发效率大概能提升 20% 以上，使得最终的人效提升大于 50%。双端复用率方面，对于纯展示型的页面，大概率可以完全由 JS 实现，双端复用率可以达到 100%，后续双端只需维护一份 JS 代码即可，极大的降低了维护成本。对于一些交互复杂的页面，需双端各自封装对应的 Native Module 实现，复用率下降，维护成本变高。

总结

随着业务的快速发展，工程复杂度的不断提升，在没有外力的情况下，开发效率必然会持续下降。如何在资源有限的情况下不断提升开发效率是一个永恒的话题。美团外卖客户端通过借助美团基建 MRN，推动混合式架构来提升效率。截至目前，美

团外卖业务已经有 60 多个 RN 页面上线，每天的 PV 高达上千万，为用户提供了稳定可靠的服务。

混合式开发带来的不仅仅是技术层面的挑战，更是对团队成员、团队组织能力的挑战。MRN 虽然能够做到跨端，但是有时候仍然需要针对特定平台单独编写代码来解决问题，这就间接要求工程师必须熟悉三个平台，团队也必须有效组织各技术栈人才共同协作，才能真正用好 MRN。

参考文献

- 京东 618：RN 框架在京东无线端的实践
- React Native 架构分析
- 点我达骑手 Weex 最佳实践
- State of React Native 2018
- 使用 React Native 的五个理由
- iOS 开发是否要采用 React Native
- 开源 React Native 组件库 beeshell 2.0 发布
- ESLint 在中大型团队的应用实践
- CAT 3.0 开源发布，支持多语言客户端及多项性能提升

作者简介

晓飞、唐笛、维康，均为美团外卖前端团队研发工程师。

招聘信息

美团外卖长期招聘 Android、iOS、FE 高级 / 资深工程师和技术专家，Base 北京、上海、成都，欢迎有兴趣的同学投递简历到 tech@meituan.com（邮件标题注明：美团外卖前端团队）。

代码质量与安全

Android 静态代码扫描效率优化与实践

肖鸿耀

DevOps 实践中，我们在 CI(Continuous Integration) 持续集成过程主要包含了代码提交、静态检测、单元测试、编译打包环节。其中静态代码检测可以在编码规范，代码缺陷，性能等问题上提前预知，从而保证项目的交付质量。Android 项目常用的静态扫描工具包括 CheckStyle、Lint、FindBugs 等，为降低接入成本，美团点评集团内部孵化了静态代码扫描插件，集合了以上常用的扫描工具。项目初期引入集团内部基建时我们接入了代码扫描插件，在 PR(Pull Request) 流程中借助 Jenkins 插件来触发自动化构建，从而达到监控代码质量的目的。初期单次构建耗时平均在 1~2min 左右，对研发效率影响甚少。但是随着时间推移，代码量随业务倍增，项目也开始使用 Flavor 来满足复杂的需求，这使得我们的单次 PR 构建达到了 8~9min 左右，其中静态代码扫描的时长约占 50%，持续集成效率不高，对我们的研发效率带来了挑战。

针对以上的背景和问题，我们思考以下几个问题：

思考一：现有插件包含的扫描工具是否都是必需的?

扫描工具对比

为了验证扫描工具的必要性，我们关心以下一些维度：

- 扫码侧重点，对比各个工具分别能针对解决什么类型的问题；
- 内置规则种类，列举各个工具提供的能力覆盖范围；
- 扫描对象，对比各个工具针对什么样的文件类型扫描；
- 原理简介，简单介绍各个工具的扫描原理；

- 优缺点，简单对比各个工具扫描效率、扩展性、定制性、全面性上的表现。

维度	CheckStyle	FindBugs	Lint
扫描侧重点	代码风格，编程规范，圈复杂度等	针对Java工程，Java代码的编码习惯、糟糕代码、性能以及安全等问题。	针对Android工程，检查 Android 项目源文件是否有潜在的错误，以及在正确性、安全性、性能、易用性、无障碍性和国际化方面是否需要优化改进。
内置规则种类	[100+] 检测规则	[300+] 检测规则	300+ 检测规则
扫描对象	源代码文件	Java字节码	Manifest文件、XML、Java、Kotlin、Java字节码、Gradle文件、Proguard文件、Property文件、图片资源
原理简介	使用Antlr库对源码文件做词语法分析生成抽象语法树，遍历整个语法树匹配检测规则	基于BCEL库通过扫描字节码完成代码检查，主要做缺陷模式匹配和数据流分析	基于抽象语法树分析，经历了LOMBOK-AST、PSI、UAST三种语法分析器
优点	对规范类规则检查很细致，速度快、轻量、针对代码风格有优势，耗时相对较少	针对字节码，对JDK定制化程度高、能发现Java代码中的一些潜在错误和缺陷	官方支持、检测全面、扩展性极强，支持自定义规则、配套工具完善
缺点	检查的规则相对简单，无法检查潜在Bug	定制规则门槛高，需要了解字节码，依赖编译代码，扫描比较耗时	如果检测了字节码文件，依赖编译代码，并且全量检测比较耗时，同时API变动比较大，目前还在不断优化迭代

注：FindBugs 只支持 Java1.0~1.8，已经被 SpotBugs 替代。鉴于部分老项目并没有迁移到 Java8，目前我们并没有使用 SpotBugs 代替 FindBugs 的原因如下，详情参考官方文档。 同时，SpotBugs 的作者也在讨论是否让 SpotBugs 支持老的 Java 版本，结论是不提供支持。

Supported Java version

SpotBugs is built by JDK8, and run on JRE8 and newer versions.

SpotBugs can scan bytecode (class files) generated by JDK8 and newer versions. However, support for Java 11 and newer is still experimental. Visit issue tracker to find known problems.

SpotBugs does not support bytecode (class files) generated by outdated JDK such as 10, 9, 7 and older versions.

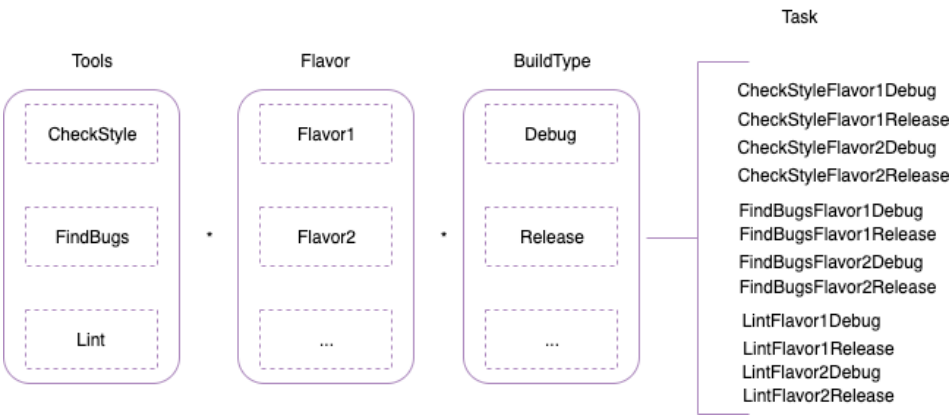
经过以上的对比分析我们发现，工具的诞生都能针对性解决某一领域问题。CheckStyle 的扫描速度快效率高，对代码风格和圈复杂度支持友好；FindBugs 针对 Java 代码潜在问题，能帮助我们发现编码上的一些错误实践以及部分安全问题和性能问题；Lint 是官方深度定制，功能极其强大，且可定制性和扩展性以及全面性都表现良好。所以综合考虑，针对思考一，我们的结论是整合三种扫描工具，充分利用每一个工具的领域特性。

思考二：是否可以优化扫描过程？

既然选择了整合这几种工具，我们面临的挑战是整合工具后扫描效率的问题，首先来分析目前的插件到底耗时在哪里。

静态代码扫描耗时分析

Android 项目的构建依赖 Gradle 工具，一次构建过程实际上是执行所有的 Gradle Task。由于 Gradle 的特性，在构建时各个 Module 都需要执行 CheckStyle、FindBugs、Lint 相关的 Task。对于 Android 来说，Task 的数量还与其构建变体 Variant 有关，其中 Variant = Flavor * BuildType。所以一个 Module 执行的相关任务可以由以下公式来描述：Flavor * BuildType *（Lint，CheckStyle，Findbugs），其中 * 为笛卡尔积。如下图所示：



可以看到，一次构建全量扫描执行的 Task 跟 Varint 个数正相关。对于现有工程的任务，我们可以看一下目前各个任务的耗时情况：（以实际开发中某一次扫描为例）

通过对 Task 耗时排序，主要的耗时体现在 FindBugs 和 Lint 对每一个 Module 的扫描任务上，CheckStyle 任务并不占主要影响。整体来看，除了工具本身的扫描时间外，耗时主要分为多 Module、多 Variant 带来的任务数量耗时。

优化思路分析

对于工具本身的扫描时间，一方面受工具自身扫描算法和检测规则的影响，另一方面也跟扫描的文件数量相关。针对源码类型的工具比如 CheckStyle 和 Lint，需要经过词法分析、语法分析生成抽象语法树，再遍历抽象语法树跟定义的检测规则去匹配；而针对字节码文件的工具 FindBugs，需要先编译源码成 Class 文件，再通过

BCEL 分析字节码指令并与探测器规则匹配。如果要在工具本身算法上去寻找优化点，代价比较大也不一定能找到有效思路，投入产出比不高，所以我们把精力放在减少 Module 和 Variant 带来的影响上。

从上面的耗时分析可以知道，Module 和 Variant 数直接影响任务数量，一次 PR 提交的场景是多样的，比如多 Module 多 Variant 都有修改，所以要考虑这些都修改的场景。先分析一个 Module 多 Variant 的场景，考虑到不同的 Variant 下源代码有一定差异，并且 FindBugs 扫描针对的是 Class 文件，不同的 Variant 都需要编译后才能扫描，直接对多 Variant 做处理比较复杂。我们可以简化问题，用以空间换时间的方式，在提交 PR 的时候根据 Variant 用不同的 Jenkins Job 来执行每一个 Variant 的扫描任务。所以接下来的问题就转变为如何优化在扫描单个 Variant 的时候多 Module 任务带来的耗时。

对于 Module 数而言，我们可以将其抽取成组件，拆分到独立仓库，将扫描任务拆分到各自仓库的变动时期，以 aar 的形式集成到主项目来减少 Module 带来的任务数。那对于剩下的 Module 如何优化呢？无论是哪一种工具，都是对其输入文件进行处理，CheckStyle 对 Java 源代码文件处理，FindBugs 对 Java 字节码文件处理，如果我们可以通过一次任务收集到所有 Module 的源码文件和编译后的字节码文件，我们就可以减少多 Module 的任务了。所以对于全量扫描，我们的主要目标是来解决如何一次性收集所有 Module 的目标文件。

思考三：是否支持增量扫描？

上面的优化思路都是基于全量扫描的，解决的是多 Module 多 Variant 带来的任务数量耗时。前面提到，工具本身的扫描时间也跟扫描的文件数量有关，那么是否可以从扫描的文件数量来入手呢？考虑平时的开发场景，提交 PR 时只是部分文件修改，我们没必要把那些没修改过的存量文件再参与扫描，而只针对修改的增量文件扫描，这样能很大程度降低无效扫描带来的效率问题。有了思路，那么我们考虑以下几个问题：

- 如何收集增量文件，包括源码文件和 Class 文件?
- 现在业界是否有增量扫描的方案，可行性如何，是否适用我们现状?
- 各个扫描工具如何来支持增量文件的扫描?

根据上面的分析与思考路径，接下来我们详细介绍如何解决上述问题。

全量扫描优化

搜集所有 Module 目标文件集

获取所有 Module 目标文件集，首先要找出哪些 Module 参与了扫描。一个 Module 工程在 Gradle 构建系统中被描述为一个“Project”，那么我们只需要找出主工程依赖的所有 Project 即可。由于依赖配置的多样性，我们可以选择在某些 Variant 下依赖不同的 Module，所以获取参与一次构建时与当前 Variant 相关的 Project 对象，我们可以用如下方式：

```
static Set<Project> collectDepProject(Project project, BaseVariant
variant, Set<Project> result
= null) {
  if (result == null) {
    result = new HashSet<>()
  }
  Set taskSet = variant.javaCompiler.taskDependencies.
getDependencies(variant.javaCompiler)
  taskSet.each { Task task ->
    if (task.project != project && hasAndroidPlugin(task.project)) {
      result.add(task.project)
      BaseVariant childVariant = getVariant(task.project)
      if (childVariant.name == variant.name || "${variant.
flavorName}${childVariant.buildType.
name}".toLowerCase() == variant.name.toLowerCase()) {
        collectDepProject(task.project, childVariant, result)
      }
    }
  }
  return result
}
```

目前文件集分为两类，一类是源码文件，另一类是字节码文件，分别可以如下处理：

```
projectSet.each { targetProject ->
  if (targetProject.plugins.hasPlugin(CodeDetectorPlugin) && GradleUtils.
hasAndroidPlugin(targetProject)) {
    GradleUtils.getAndroidExtension(targetProject).sourceSets.all {
AndroidSourceSet sourceSet ->
      if (!sourceSet.name.startsWith("test") && !sourceSet.name.
startsWith(SdkConstants.FD_TEST)) {
        source sourceSet.java.srcDirs
      }
    }
  }
}
```

注：上面的 Source 是 CheckStyle Task 的属性，用其来指定扫描的文件集合；

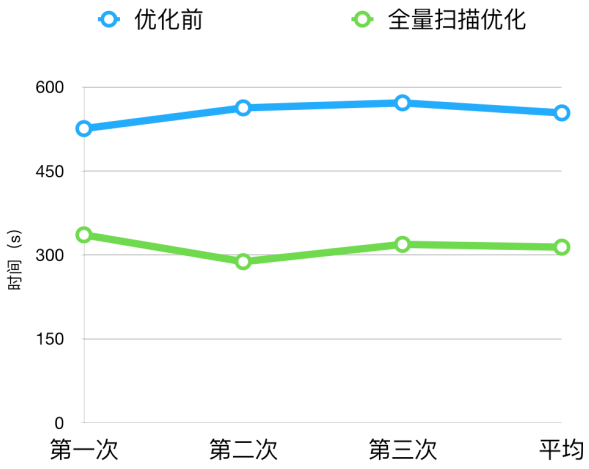
```
// 排除掉一些模板代码 class 文件
static final Collection<String> defaultExcludes =
(androidDataBindingExcludes +
androidExcludes + butterKnifeExcludes + dagger2Excludes).asImmutable()

List<ConfigurableFileTree> allClassesFileTree = new ArrayList<>()
ConfigurableFileTree currentProjectClassesDir = project.fileTree(dir:
variant.javaCompile.
destinationDir, excludes: defaultExcludes)
allClassesFileTree.add(currentProjectClassesDir)
GradleUtils.collectDepProject(project, variant).each { targetProject ->
  if (targetProject.plugins.hasPlugin(CodeDetectorPlugin) && GradleUtils.
hasAndroidPlugin(targetProject)) {
    // 可能有的工程没有 Flavor 只有 buildType
     GradleUtils.getAndroidVariants(targetProject).each { BaseVariant
targetProjectVariant ->
      if (targetProjectVariant.name == variant.name ||
"${targetProjectVariant.name}".
toLowerCase() == variant.buildType.name.toLowerCase()) {
         allClassesFileTree.add(targetProject.fileTree(dir:
targetProjectVariant.javaCompile.
destinationDir, excludes: defaultExcludes))
      }
    }
  }
}
```

注：收集到字节码文件集后，可以用通过 FindBugsTask 的 Class 属性指定扫描，后文会详细介绍 FindBugs Task 相关属性。

对于 Lint 工具而言，相应的 Lint Task 并没有相关属性可以指定扫描文件，所以在全量扫描上，我们暂时没有针对 Lint 做优化。

全量扫描优化数据

通过对 CheckStyle 和 FindBugs 全量扫描的优化，我们将整体扫描时间由原来的 9min 降低到了 5min 左右。



增量扫描优化

由前面的思考分析我们知道，并不是所有的文件每次都需要参与扫描，所以我们可以通过增量扫描的方式来提高扫描效率。

增量扫描技术调研

在做具体技术方案之前，我们先调研一下业界的现有方案，调研如下：

工具	现有方案	备注	初步调研方案
checkStyle	https://github.com/yangziwen/diff-checkstyle	基于git命令对比提交记录的差异获取差异文件	可以从配置参数入手，指定扫描的Java源文件集合
FindBugs	暂无	暂无此方面资料	可以从配置参数入手，指定扫描的Class文件集合
Lint	Lint增量扫描（基于Android Gradle Plugin 2.X实现）	未开源，Android Gradle Plugin 3.x以后实现方式不一样，需要单独处理	需要了解内部实现原理，自定义增量扫描Client

针对 Lint，我们可以借鉴现有实现思路，同时深入分析扫描原理，在 3.x 版本上寻找出增量扫描的解决方案。对于 CheckStyle 和 FindBugs，我们需要了解工具的相关配置参数，为其指定特定的差异文件集合。

注：业界有一些增量扫描的案例，例如 diff_cover，此工具主要是对单元测试整体覆盖率的检测，以增量代码覆盖率作为一个指标来衡量项目的质量，但是这跟我们的静态代码分析的需求不太符合。它有一个比较好的思路是找出差异的代码行来分析覆盖率，粒度比较细。但是对于静态代码扫描，仅仅的差异行不足以完成上下文的语义分析，尤其是针对 FindBugs 这类需要分析字节码的工具，获取的差异行还需要经过编译成 Class 文件才能进行分析，方案并不可取。

寻找增量修改文件

增量扫描的第一步是获取待扫描的目标文件。我们可以通过 git diff 命令来获取差异文件，值得注意的是对于删除的文件和重命名的文件需要忽略，我们更关心新增和修改的文件，并且只需要获取差异文件的路径就好了。举个例子：git diff –name-only –diff-filter=dr commitHash1 commitHash2，以上命令意思是对比两次提交记录的差异文件并获取路径，过滤删除和重命名的文件。对于寻找本地仓库的差异文件上面的命令已经足够了，但是对于 PR 的情况还有一些复杂，需要对比本地代码与远程仓库目标分支的差异。集团的代码管理工具在 Jenkins 上有相应的插件，该插件默认提供了几个参数，我们需要用到以下两个：– ${targetBranch}：需要合入代码的目标分支地址；– ${sourceCommitHash}：需要提交的代码 hash 值；

通过这两个参数执行以下一系列命令来获取与远程目标分支的差异文件。

```
git remote add upstream ${upstreamGitUrl}
git fetch upstream ${targetBranch}
git diff --name-only --diff-filter=dr $sourceCommitHash
upstream/$targetBranch
```

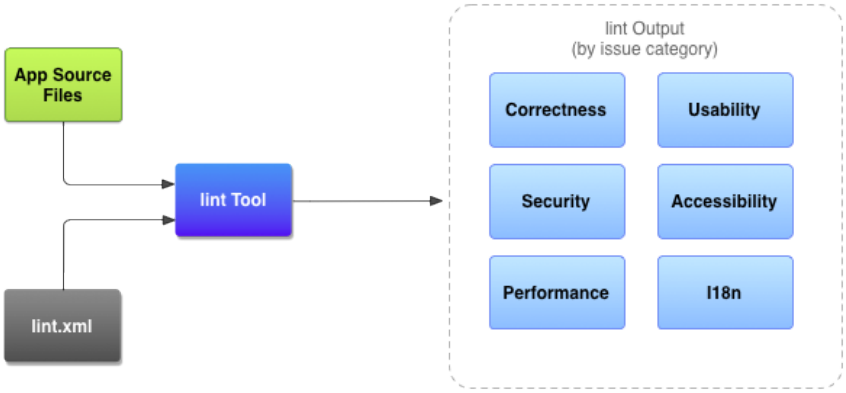
1. 配置远程分支别名为 UpStream，其中 upstreamGitUrl 可以在插件提供的配置属性中设置；
2. 获取远程目标分支的更新；

3. 比较分支差异获取文件路径。

通过以上方式，我们找到了增量修改文件集。

Lint 扫描原理分析

在分析 Lint 增量扫描原理之前，先介绍一下 Lint 扫描的工作流程：



App Source Files

项目中的源文件，包括 Java、XML、资源文件、proGuard 等。

lint.xml

用于配置希望排除的任何 Lint 检查以及自定义问题严重级别，一般各个项目都会根据自身项目情况自定义的 lint.xml 来排除一些检查项。

lint Tool

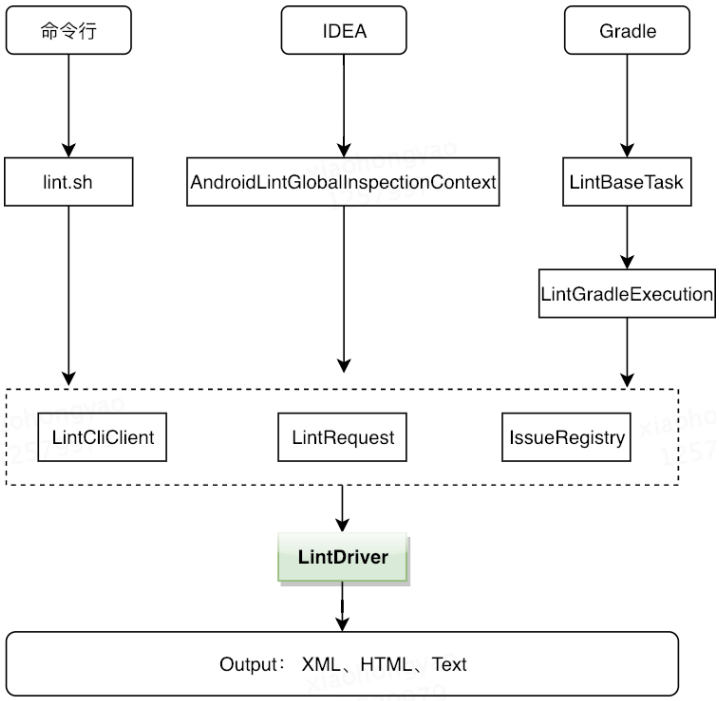
一套完整的扫描工具用于对 Android 的代码结构进行分析，可以通过命令行、IDEA、Gradle 命令三种方式运行 lint 工具。

lint Output

Lint 扫描的输出结果。

从上面可以看出，Lint Tool 就像一个加工厂，对投入进来的原料（源代码）进行加工处理（各种检测器分析），得到最终的产品（扫描结果）。Lint Tool 作为一个扫

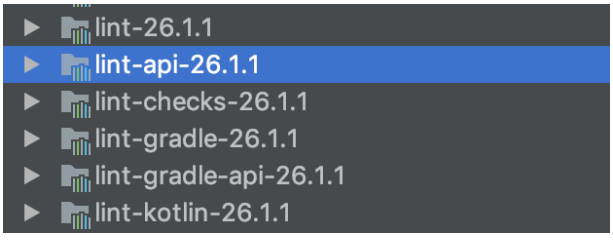
描工具集，有多种使用方式。Android 为我们提供了三种运行方式，分别是命令行、IDEA、Gradle 任务。这三种方式最终都殊途同归，通过 LintDriver 来实现扫描。如下图所示：



为了方便查看源码，新建一个工程，在 build.gradle 脚本中，添加如下依赖：

```
compile 'com.android.tools.build:gradle:3.1.1'
compile 'com.android.tools.lint:lint-gradle:26.1.1'
```

我们可以得到如下所示的依赖：



lint-api-26.1.1

Lint 工具集的一个封装，实现了一组 API 接口，用于启动 Lint；

lint-checks-26.1.1

一组内建的检测器，用于对这种描述好 Issue 进行分析处理；

lint-26.1.1

可以看做是依赖上面两个 jar 形成的一个基于命令行的封装接口形成的脚手架工程，我们的命令行、Gradle 任务都是继承自这个 jar 包中相关类来做的实现；

lint-gradle-26.1.1

可以看做是针对 Gradle 任务这种运行方式，基于 lint-26.1.1 做了一些封装类；

lint-gradle-api-26.1.1

真正 Gradle Lint 任务在执行时调用的入口；

在理解清楚了以上几个 jar 的关系和作用之后，我们可以发现 Lint 的核心库其实是前三个依赖。后面两个其实是基于脚手架，对 Gradle 这种运行方式做的封装。最核心的逻辑在 LintDriver 的 Analyze 方法中。

```
fun analyze() {

    ... 省略部分代码 ...

    for (project in projects) {
        fireEvent(EventType.REGISTERED_PROJECT, project = project)
    }
    registerCustomDetectors(projects)

    ... 省略部分代码 ...

    try {
        for (project in projects) {
            phase = 1

            val main = request.getMainProject(project)

            // The set of available detectors varies between projects
            computeDetectors(project)

            if (applicableDetectors.isEmpty()) {
```

```
                // No detectors enabled in this project: skip it
                continue
            }

            checkProject(project, main)
            if (isCanceled) {
                break
            }

            runExtraPhases(project, main)
        }
    } catch (throwable: Throwable) {
        // Process canceled etc
        if (!handleDetectorError(null, this, throwable)) {
            cancel()
        }
    }
    ... 省略部分代码 ...
}
```

主要是以下三个重要步骤：

- registerCustomDetectors(projects)

Lint 为我们提供了许多内建的检测器，除此之外我们还可以自定义一些检测器，这些都需要注册进 Lint 工具用于对目标文件进行扫描。这个方法主要做以下几件事情：

1. 遍历每一个 Project 和它的依赖 Library 工程，通过 client.findRuleJars 来找出自定义的 jar 包；
2. 通过 client.findGlobalRuleJars 找出全局的自定义 jar 包，可以作用于每一个 Android 工程；
3. 从找到的 jarFiles 列表中，解析出自定义的规则，并与内建的 Registry 一起合并为 CompositeIssueRegistry；需要注意的是，自定义的 Lint 的 jar 包存放位置是 build/intermediaters/lint 目录，如果是需要每一个工程都生效，则存放位置为 ~/.android/lint/。

- computeDetectors(project)

这一步主要用来收集当前工程所有可用的检测器。

checkProject(project, main) 接下来这一步是最为关键的一步。在此方法中，

调用 runFileDetectors 来进行文件扫描。Lint 支持的扫描文件类型很多，因为是官方支持，所以针对 Android 工程支持的比较友好。一次 Lint 任务运行时，Lint 的扫描范围主要由 Scope 来描述。具体表现在：

```
fun infer(projects: Collection<Project>?): EnumSet<Scope> {
            if (projects == null || projects.isEmpty()) {
                return Scope.ALL
            }

            // Infer the scope
            var scope = EnumSet.noneOf(Scope::class.java)
            for (project in projects) {
                val subset = project.subset
                if (subset != null) {
                    for (file in subset) {
                        val name = file.name
                        if (name == ANDROID_MANIFEST_XML) {
                            scope.add(MANIFEST)
                        } else if (name.endsWith(DOT_XML)) {
                            scope.add(RESOURCE_FILE)
                        } else if (name.endsWith(DOT_JAVA) || name.
endsWith(DOT_KT)) {
                            scope.add(JAVA_FILE)
                        } else if (name.endsWith(DOT_CLASS)) {
                            scope.add(CLASS_FILE)
                        } else if (name.endsWith(DOT_GRADLE)) {
                            scope.add(GRADLE_FILE)
                        } else if (name == OLD_PROGUARD_FILE || name ==
FN_PROJECT_PROGUARD_FILE)
{
                            scope.add(PROGUARD_FILE)
                        } else if (name.endsWith(DOT_PROPERTIES)) {
                            scope.add(PROPERTY_FILE)
                        } else if (name.endsWith(DOT_PNG)) {
                            scope.add(BINARY_RESOURCE_FILE)
                        } else if (name == RES_FOLDER || file.parent ==
RES_FOLDER) {
                            scope.add(ALL_RESOURCE_FILES)
                            scope.add(RESOURCE_FILE)
                            scope.add(BINARY_RESOURCE_FILE)
                            scope.add(RESOURCE_FOLDER)
                        }
                    }
                } else {
                    // Specified a full project: just use the full
project scope
                    scope = Scope.ALL
```

```
                    break
                }
            }
    }
```

可以看到，如果 Project 的 Subset 为 Null，Scope 就为 Scope.ALL，表示本次扫描会针对能检测的所有范围，相应地在扫描时也会用到所有全部的 Detector 来扫描文件。

如果 Project 的 Subset 不为 Null，就遍历 Subset 的集合，找出 Subset 中的文件分别对应哪些范围。其实到这里我们已经可以知道，Subset 就是我们增量扫描的突破点。接下来我们看一下 runFileDetectors：

```
if(scope.contains(Scope.JAVA_FILE)||scope.contains(Scope.ALL_JAVA_FILES)){
  val checks = union(scopeDetectors[Scope.JAVA_
FILE],scopeDetectors[Scope.ALL_JAVA_FILES])
  if (checks != null && !checks.isEmpty()) {
    val files = project.subset
    if (files != null) {
      checkIndividualJavaFiles(project, main, checks, files)
    } else {
      val sourceFolders = project.javaSourceFolders
      val testFolders = if (scope.contains(Scope.TEST_SOURCES))
      project.testSourceFolders
      else
      emptyList<File> ()
      val generatedFolders = if (isCheckGeneratedSources)
      project.generatedSourceFolders
      else
      emptyList<File> ()
      checkJava(project, main, sourceFolders, testFolders,
generatedFolders, checks)
    }
  }
}
```

这里更加明确，如果 project.subset 不为空，就对单独的 Java 文件扫描，否则，就对源码文件和测试目录以及自动生成的代码目录进行扫描。整个 runFileDetectors 的扫描顺序入下：

1. Scope.MANIFEST
2. Scope.ALL_RESOURCE_FILES)|| scope.contains(Scope.RESOURCE_FILE) || scope.contains(Scope.RESOURCE_FOLDER) || scope.contains(Scope.BINARY_RESOURCE_FILE)
3. scope.contains(Scope.JAVA_FILE) || scope.contains(Scope.ALL_JAVA_FILES)
4. scope.contains(Scope.CLASS_FILE) || scope.contains(Scope.ALL_CLASS_FILES) || scope.contains(Scope.JAVA_LIBRARIES)
5. scope.contains(Scope.GRADLE_FILE)
6. scope.contains(Scope.OTHER)
7. scope.contains(Scope.PROGUARD_FILE)
8. scope.contains(Scope.PROPERTY_FILE)

与官方文档的描述顺序一致。

现在我们已经知道，增量扫描的突破点其实是需要构造 project.subset 对象。

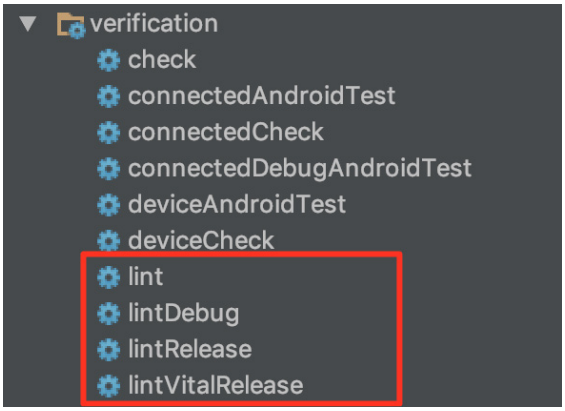
```
/**
     * Adds the given file to the list of files which should be
checked in this
     * project. If no files are added, the whole project will be
checked.
     *
     * @param file the file to be checked
     */
    public void addFile(@NonNull File file) {
        if (files == null) {
            files = new ArrayList<>();
        }
        files.add(file);
    }

    /**
     * The list of files to be checked in this project. If null, the
whole
     * project should be checked.
     *
     * @return the subset of files to be checked, or null for the
whole project
     */
```

```
    @Nullable
    public List<File> getSubset() {
        return files;
    }
```

注释也很明确的说明了只要 Files 不为 Null，就会扫描指定文件，否则扫描整个工程。

Lint 增量扫描 Gradle 任务实现

前面分析了如何获取差异文件以及增量扫描的原理，分析的重点还是侧重在 Lint 工具本身的实现机制上。接下来分析，在 Gradle 中如何实现一个增量扫描任务。大家知道，通过执行 ./gradlew lint 命令来执行 Lint 静态代码检测任务。创建一个新的 Android 工程，在 Gradle 任务列表中可以在 Verification 这个组下面找到几个 Lint 任务，如下所示：



这几个任务就是 Android Gradle 插件在加载的时候默认创建的。分别对应于以下几个 Task：

- lint->LintGlobalTask：由 TaskManager 创建；
- lintDebug、lintRelease、lintVitalRelease->LintPerVariantTask：由 ApplicationTaskManager 或者 LibraryTaskManager 创建，其中 lintVitalRelease 只在 release 下生成；

所以，在 Android Gradle 插件中，应用于 Lint 的任务分别为 LintGlobalTask 和 LintPerVariantTask。他们的区别是前者执行的是扫描所有 Variant，后者执行只针对单独的 Variant。而我们的增量扫描任务其实是跟 Variant 无关的，因为我们会把所有差异文件都收集到。无论是 LintGlobalTask 或者是 LintPerVariantTask，都继承自 LintBaseTask。最终的扫描任务在 LintGradleExecution 的 runLint 方法中执行，这个类位于 lint-gradle-26.1.1 中，前面提到这个库是基于 Lint 的 API 针对 Gradle 任务做的一些封装。

```
/** Runs lint on the given variant and returns the set of warnings */
    private Pair<List<Warning>, LintBaseline> runLint(
            @Nullable Variant variant,
            @NonNull VariantInputs variantInputs,
            boolean report, boolean isAndroid) {
        IssueRegistry registry = createIssueRegistry(isAndroid);
        LintCliFlags flags = new LintCliFlags();
        LintGradleClient client =
                new LintGradleClient(
                        descriptor.getGradlePluginVersion(),
                        registry,
                        flags,
                        descriptor.getProject(),
                        descriptor.getSdkHome(),
                        variant,
                        variantInputs,
                        descriptor.getBuildTools(),
                        isAndroid);
        boolean fatalOnly = descriptor.isFatalOnly();
        if (fatalOnly) {
            flags.setFatalOnly(true);
        }
        LintOptions lintOptions = descriptor.getLintOptions();
        if (lintOptions != null) {
            syncOptions(
                    lintOptions,
                    client,
                    flags,
                    variant,
                    descriptor.getProject(),
                    descriptor.getReportsDir(),
                    report,
                    fatalOnly);
        } else {
            // Set up some default reporters
```

```
            flags.getReporters().add(Reporter.createTextReporter(client,
flags, null,
                    new PrintWriter(System.out, true), false));
            File html = validateOutputFile(createOutputPath(descriptor.
getProject(), null, ".html",
                    null, flags.isFatalOnly()));
            File xml = validateOutputFile(createOutputPath(descriptor.
getProject(), null, DOT_XML,
                    null, flags.isFatalOnly()));
            try {
                flags.getReporters().add(Reporter.
createHtmlReporter(client, html, flags));
                flags.getReporters().add(Reporter.
createXmlReporter(client, xml, false));
            } catch (IOException e) {
                throw new GradleException(e.getMessage(), e);
            }
        }
        if (!report || fatalOnly) {
            flags.setQuiet(true);
        }
        flags.setWriteBaselineIfMissing(report && !fatalOnly);

        Pair<List<Warning>, LintBaseline> warnings;
        try {
            warnings = client.run(registry);
        } catch (IOException e) {
            throw new GradleException("Invalid arguments.", e);
        }

        if (report && client.haveErrors() && flags.isSetExitCode()) {
            abort(client, warnings.getFirst(), isAndroid);
        }

        return warnings;
    }
```

我们在这个方法中看到了 warnings = client.run(registry)，这就是 Lint 扫描得到的结果集。总结一下这个方法中做了哪些准备工作用于 Lint 扫描：1. 创建 IssueRegistry，包含了 Lint 内建的 BuiltinIssueRegistry；2. 创建 LintCliFlags；3. 创建 LintGradleClient，这里面传入了一大堆参数，都是从 Gradle Android 插件的运行环境中获得；4. 同步 LintOptions，这一步是将我们在 build.gralde 中配置的一些 Lint 相关的 DSL 属性，同步设置给 LintCliFlags，给真正的 Lint 扫描核心库使

用；5. 执行 Client 的 Run 方法，开始扫描。

扫描的过程上面的原理部分已经分析了，现在我们思考一下如何构造增量扫描的任务。我们已经分析到扫描的关键点是 client.run(registry)，所以我们需要构造一个 Client 来执行扫描。一个想法是通过反射来获取 Client 的各个参数，当然这个思路是可行的，我们也验证过实现了一个用反射方式构造的 Client。但是反射这种方式有个问题是丢失了从 Gradle 任务执行到调用 Lint API 开始扫描这一过程中做的其他事情，侵入性比较高，所以我们最终采用继承 LintBaseTask 自行实现增量扫描任务的方式。

FindBugs 扫描简介

FindBugs 是一个静态分析工具，它检查类或者 JAR 文件，通过 Apache 的 BCEL 库来分析 Class，将字节码与一组缺陷模式进行对比以发现问题。FindBugs 自身定义了一套缺陷模式，目前的版本 3.0.1 内置了总计 300 多种缺陷，详细可参考官方文档。FindBugs 作为一个扫描的工具集，可以非常灵活的集成在各种编译工具中。接下来，我们主要分析在 Gradle 中 FindBugs 的相关内容。

Gradle FindBugs 任务属性分析

在 Gradle 的内置任务中，有一个 FindBugs 的 Task，我们看一下官方文档对 Gradle 属性的描述。

选几个比较重要的属性介绍：

- Classes 该属性表示我们要分析的 Class 文件集合，通常我们会把编译结果的 Class 目录用于扫描。
- Classpath 分析目标集合中的 Class 需要用到的所有相关的 Classes 路径，但是并不会分析它们自身，只用于扫描。
- Effort 包含 MIN，Default，MAX，级别越高，分析得越严谨越耗时。
- findBugs ClasspathFinbugs 库相关的依赖路径，用于配置扫描的引擎库。
- reportLevel 报告级别，分为 Low，Medium，High。如果为 Low，所有

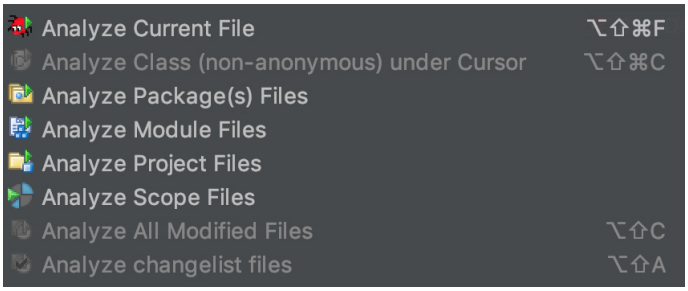
Bug 都报告，如果为 High，仅报告 High 优先级。

- Reports 扫描结果存放路径。

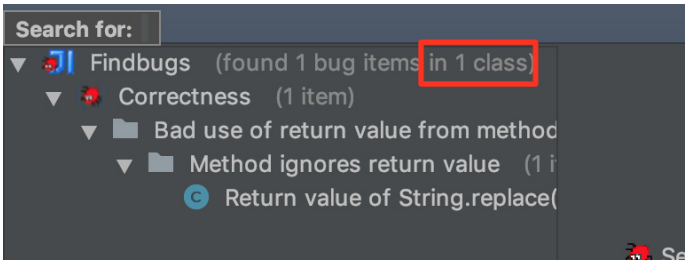
通过以上属性解释，不难发现要 FindBugs 增量扫描，只需要指定 Classes 的文件集合就可以了。

FindBugs 任务增量扫描分析

在做增量扫描任务之前，我们先来看一下 FindBugs IDEA 插件是如何进行单个文件扫描的。



我们选择 Analyze Current File 对当前文件进行扫描，扫描结果如下所示：



可以看到确实只扫描了一个文件。那么扫描到底使用了哪些输入数据呢，我们可以通过扫描结果的提示清楚看到：

这里我们能看到很多有用的信息：

- 源码目录列表，包含了工程中的 Java 目录，res 目录，以及编译过程中生成的一些类目录；
- 需要分析的目标 Class 集合，为编译后的 Build 目录下的当前 Java 文件对应的 Class 文件；

- Aux Classpath Entries, 表示分析上面的目标文件需要用到的类路径。

所以，根据 IDEA 的扫描结果来看，我们在做增量扫描的时候需要解决上面这几个属性的获取。在前面我们分析的属性是 Gradle 在 FindBugs lib 的基础上，定义的一套对应的 Task 属性。真正的 FinBugs 属性我们可以通过官方文档或者源码中查到。

配置 AuxClasspath

前文提到，ClassPath 是用来分析目标文件需要用到的相关依赖 Class，但本身并不会被分析，所以我们需要尽可能全的找到所有的依赖库，否则在扫描的时候会报依赖的类库找不到。

```
FileCollection buildClasses = project.fileTree(dir: "${project.
buildDir}/intermediates/classes/${variant.flavorName}/${variant.
buildType.name}",includes: classIncludes)

FileCollection targetClasspath = project.files()
GradleUtils.collectDepProject(project, variant).each { targetProject ->
    GradleUtils.getAndroidVariants(targetProject).each { targetVariant ->
        if (targetVariant.name.capitalize().equalsIgnoreCase(variant.
name.capitalize())) {
            targetClasspath += targetVariant.javaCompile.classpath
        }
    }
}

classpath = variant.javaCompile.classpath + targetClasspath +
buildClasses
```

FindBugs 增量扫描误报优化

对于增量文件扫描，参与的少数文件扫描在某些模式规则上可能会出现误判，但是全量扫描不会有问题，因为参与分析的目标文件是全集。举一个例子：

```
class A {
  public static String buildTime = "";
  ....
}
```

静态变量 buildTime 会被认为应该加上 Final，但是其实其他类会对这个变量赋值。如果单独扫描类 A 文件，就会报缺陷 BUG_TYPE_MS_SHOULD_BE_FINAL。我们通过 Findbugs-IDEA 插件来扫描验证，也同样会有一样的问题。要解决此类问题，需要找到谁依赖了类 A，并且一同参与扫描，同时也需要找出类 A 依赖了哪些文件，简单来说：需要找出与类 A 有直接关联的类。为了解决这个问题，我们通过 ASM 来找出相关的依赖，具体如下：

```
void findAllScanClasses(ConfigurableFileTree allClass) {
    allScanFiles = [] as HashSet
    String buildClassDir = "${project.buildDir}/$FINDBUGS_ANALYSIS_
DIR/$FINDBUGS_
ANALYSIS_DIR_ORIGIN"

    Set<File> moduleClassFiles = allClass.files
    for (File file : moduleClassFiles) {
        String[] splitPath = file.absolutePath.split("$FINDBUGS_
ANALYSIS_DIR/$FINDBUGS_
ANALYSIS_DIR_ORIGIN/")
        if (splitPath.length > 1) {
            String className = getFileNameNoFlag(splitPath[1],'.')
            String innerClassPrefix = ""
            if (className.contains('$')) {
                innerClassPrefix = className.split('\\$')[0]
            }
            if (diffClassNamePath.contains(className) ||
diffClassNamePath.contains(innerClassPrefix))
{
                allScanFiles.add(file)
            } else {
                Iterable<String> classToResolve = new
ArrayList<String>()
                classToResolve.add(file.absolutePath)
                Set<File> dependencyClasses = Dependencies.
findClassDependencies(project, new
ClassAcceptor(), buildClassDir, classToResolve)
                for (File dependencyClass : dependencyClasses) {
                    if (diffClassNamePath.
contains(getPackagePathName(dependencyClass))) {
                        allScanFiles.add(file)
                        break
                    }
                }
            }
        }
    }
}
```

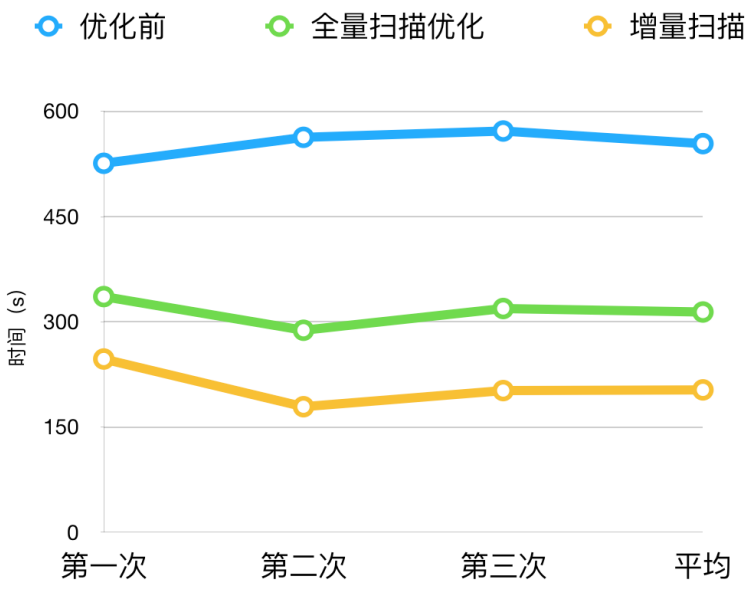
通过以上方式，我们可以解决一些增量扫描时出现的误报情况，相比 IDEA 工具，我们更进一步降低了扫描部分文件的误报率。

CheckStyle 增量扫描

相比而言，CheckStyle 的增量扫描就比较简单了。CheckStyle 对源码扫描，根据官方文档各个属性的描述，我们发现只要指定 Source 属性的值就可以指定扫描的目标文件。

```
void configureIncrementScanSource() {
    boolean isCheckPR = false
    DiffFileFinder diffFileFinder

    if (project.hasProperty(CodeDetectorExtension.CHECK_PR)) {
        isCheckPR = project.getProperties().get(CodeDetectorExtension.
CHECK_PR)
    }

    if (isCheckPR) {
        diffFileFinder = new DiffFileFinderHelper.PRDiffFileFinder()
    } else {
        diffFileFinder = new DiffFileFinderHelper.LocalDiffFileFinder()
    }

    source diffFileFinder.findDiffFiles(project)

    if (getSource().isEmpty()) {
        println '没有找到差异 java 文件，跳过 checkStyle 检测'
    }
}
```

经过全量扫描和增量扫描的优化，我们整个扫描效率得到了很大提升，一次 PR 构建扫描效率整体提升 50%+。优化数据如下：



扫描工具通用性

解决了扫描效率问题，我们想怎么让更多的工程能低成本的使用这个扫描插件。对于一个已经存在的工程，如果没有使用过静态代码扫描，我们希望在接入扫描插件后续新增的代码能够保证其经过增量扫描没有问题。而老的存量代码，由于代码量过大增量扫描并没有效率上的优势，我们希望可以使用全量扫描逐步解决存量代码存在的问题。同时，为了配置工具的灵活，也提供配置来让接入方自己决定选择接入哪些工具。这样可以让扫描工具同时覆盖到新老项目，保证其通用。所以，要同时支持配置使用增量或者全量扫描任务，并且提供灵活的选择接入哪些扫描工具。

扫描完整性保证

前面提到过，在 FindBugs 增量扫描可能会出现因为参与分析的目标文件集不全导致的某类匹配规则误报，所以在保证扫描效率的同时，也要保证扫描的完整性和准确性。我们的策略是以增量扫描为主，全量扫描为辅，PR 提交使用增量扫描提高效率，在 CI 配置 Daily Build 使用全量扫描保证扫描完整和不遗漏。

我们在自己的项目中实践配置如下：

```
apply plugin: 'code-detector'

codeDetector {
    // 配置静态代码检测报告的存放位置
    reportRelativePath = rootProject.file('reports')

    /**
     * 远程仓库地址，用于配置提交 pr 时增量检测
     */
    upstreamGitUrl = "ssh://git@xxxxxxxx.git"

    checkStyleConfig {
        /**
         * 开启或关闭 CheckStyle 检测
         * 开启：true
         * 关闭：false
         */
        enable = true
        /**
         * 出错后是否要终止检查
         * 终止：false
         * 不终止：true。配置成不终止的话 CheckStyleTask 不会失败，也不会拷贝错误
报告
         */
        ignoreFailures = false
        /**
         * 是否在日志中展示违规信息
         * 显示：true
         * 不显示：false
         */
        showViolations = true
        /**
         * 统一配置自定义的 checkstyle.xml 和 checkstyle.xsl 的 uri
         * 配置路径为：
         *      "${checkStyleUri}/checkstyle.xml"
         *      "${checkStyleUri}/checkstyle.xsl"
         *
         * 默认为 null，使用 CodeDetector 中的默认配置
         */
        checkStyleUri = rootProject.file('codequality/checkstyle')
    }

    findBugsConfig {
        /**
         * 开启或关闭 Findbugs 检测
         * 开启：true
```

```
            * 关闭：false
            */
           enable = true
           /**
            * 可选项，设置分析工作的等级，默认值为 max
            * min, default, or max. max 分析更严谨，报告的 bug 更多 . min 略微少些
            */
           effort = "max"
           /**
            * 可选项，默认值为 high
            * low, medium, high. 如果是 low 的话，那么报告所有的 bug
            */
           reportLevel = "high"
           /**
            * 统一配置自定义的 findbugs_include.xml 和 findbugs_exclude.xml 的 uri
            * 配置路径为：
            *       "${findBugsUri}/findbugs_include.xml"
            *       "${findBugsUri}/findbugs_exclude.xml"
            * 默认为 null，使用 CodeDetector 中的默认配置
            */
           findBugsUri = rootProject.file('codequality/findbugs')
       }

       lintConfig {

           /**
            * 开启或关闭 lint 检测
            * 开启：true
            * 关闭：false
            */
           enable = true

           /**
            * 统一配置自定义的 lint.xml 和 retrolambda_lint.xml 的 uri
            * 配置路径为：
            *       "${lintConfigUri}/lint.xml"
            *       "${lintConfigUri}/retrolambda_lint.xml"
            * 默认为 null，使用 CodeDetector 中的默认配置
            */
           lintConfigUri = rootProject.file('codequality/lint')
       }
   }
```

我们希望扫描插件可以灵活指定增量扫描还是全量扫描以应对不同的使用场景，比如已存在项目的接入、新项目的接入、打包时的检测等。

执行脚本示例：

```
./gradlew ":${appModuleName}:assemble${ultimateVariantName}"
-PdetectorEnable=true
-PcheckStyleIncrement=true -PlintIncrement=true -PfindBugsIncrement=true
-PcheckPR=${checkPR} -PsourceCommitHash=${sourceCommitHash}
-PtargetBranch=${targetBranch} --stacktrace
```

希望一次任务可以暴露所有扫描工具发现的问题，当某一个工具扫描到问题后不终止任务，如果是本地运行在发现问题后可以自动打开浏览器方便查看问题原因。

```
def finalizedTaskArray = [lintTask,checkStyleTask,findbugsTask]
checkCodeTask.finalizedBy finalizedTaskArray

"open ${reportPath}".execute()
```

为了保证提交的 PR 不会引起打包问题影响包的交付，在 PR 时触发的任务实际为打包任务，我们将静态代码扫描任务挂接在打包任务中。由于我们的项目是多 Flavor 构建，在 CI 上我们将触发多个 Job 同时执行对应 Flavor 的增量扫描和打包任务。同时为了保证代码扫描的完整性，我们在真正的打包 Job 上执行全量扫描。

本文主要介绍了在静态代码扫描优化方面的一些思路与实践，并重点探讨了对 Lint、Finbugs、CheckStyle 增量扫描的一些尝试。通过对扫描插件的优化，我们在代码扫描的效率上得到了提升，同时在实践过程中我们也积累了自定义 Lint 检测规则的方案，未来我们将配合基础设施标准化建设，结合静态扫描插件制定一些标准化检测规则来更好的保证我们的代码规范以及质量。

参考资料

- CheckStyle 插件文档
- FindBugs 插件文档
- Android Lint 开发介绍
- Lint 增量扫描
- FindBugs 配置属性介绍
- 美团外卖 Android Lint 代码检查实践
- Gradle 源码
- 静态代码检测工具对比
- Android Gradle 插件源码

作者简介

鸿耀，美团餐饮生态技术团队研发工程师。

招聘

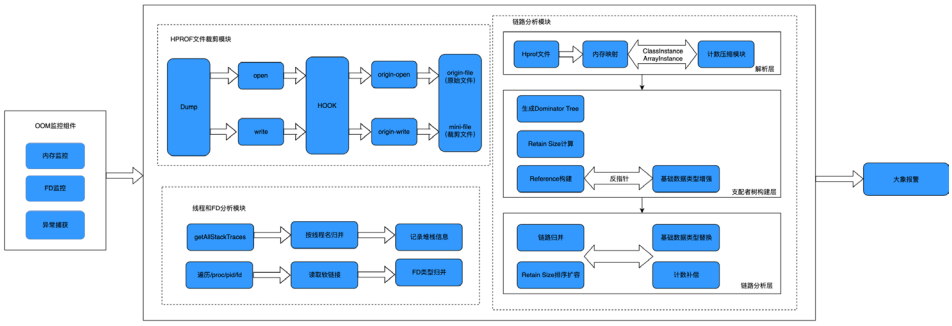
美团餐饮生态技术团队诚招 Android、Java 后端高级 / 资深工程师和技术专家，Base 成都，欢迎有兴趣的同学投递简历到 tech@meituan.com。

Probe：Android 线上 OOM 问题定位组件

逢搏　毅然　永刚

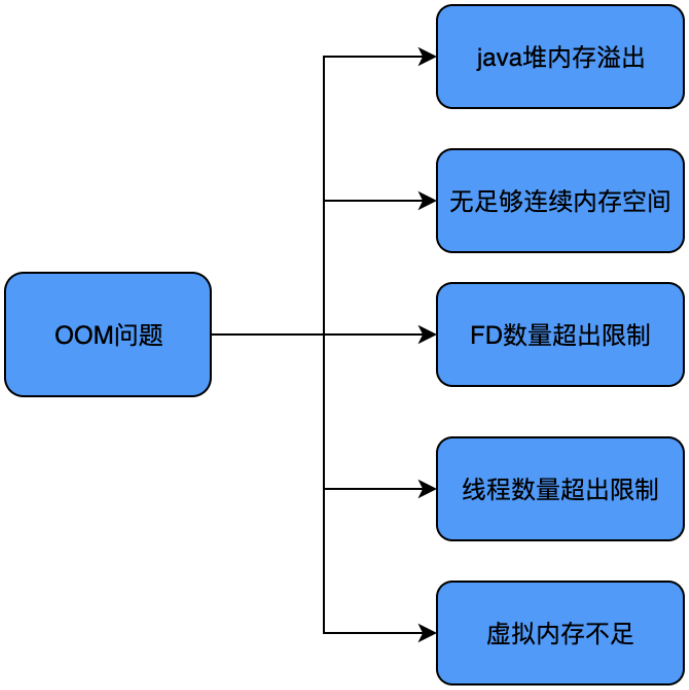
配送骑手端 App 是骑手用于完成配送履约的应用，帮助骑手完成接单、到店、取货及送达，提供各种不同的运力服务，也是整个外卖闭环中的重要节点。由于配送业务的特性，骑手 App 对于应用稳定性的要求非常高，体现 App 稳定性的一个重要数据就是 Crash 率，而在众多 Crash 中最棘手最难定位的就是 OOM 问题。对于骑手端 App 而言，每天骑手都会长时间的使用 App 进行配送，而在长时间的使用过程中，App 中所有的内存泄漏都会慢慢累积在内存中，最后就容易导致 OOM，从而影响骑手的配送效率，进而影响整个外卖业务。

于是我们构建了用于快速定位线上 OOM 问题的组件——Probe，下图是 Probe 组件架构，本文主要分享 Probe 组件是如何对线上 OOM 问题进行快速定位的。



OOM 原因分析

要定位 OOM 问题，首先需要弄明白 Android 中有哪些原因会导致 OOM，Android 中导致 OOM 的原因主要可以划分为以下几个类型：



Android 虚拟机最终抛出 OutOfMemoryError 的代码位于 /art/runtime/thread.cc。

```
void Thread::ThrowOutOfMemoryError(const char* msg)
参数 msg 携带了 OOM 时的错误信息
```

下面两个地方都会调用上面方法抛出 OutOfMemoryError 错误，这也是 Android 中发生 OOM 的主要原因。

堆内存分配失败

系统源码文件：/art/runtime/gc/heap.cc

```
void Heap::ThrowOutOfMemoryError(Thread* self, size_t byte_count,
AllocatorType allocator_
type)
抛出时的错误信息：
    oss << "Failed to allocate a " << byte_count << " byte allocation
with " << total_bytes_free
<< " free bytes and " << PrettySize(GetFreeMemoryUntilOOME()) << " until OOM";
```

这是在进行堆内存分配时抛出的 OOM 错误，这里也可以细分成两种不同的类型：

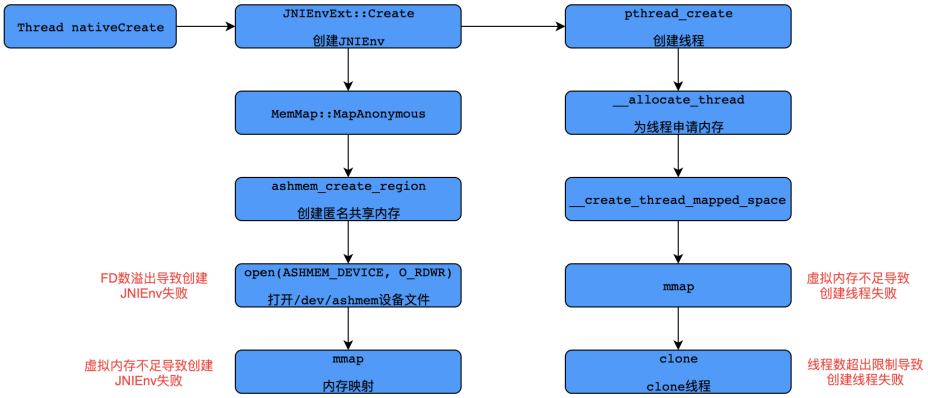
1. 为对象分配内存时达到进程的内存上限。由 Runtime.getRuntime.Max-Memory() 可以得到 Android 中每个进程被系统分配的内存上限，当进程占用内存达到这个上限时就会发生 OOM，这也是 Android 中最常见的 OOM 类型。
2. 没有足够大小的连续地址空间。这种情况一般是进程中存在大量的内存碎片导致的，其堆栈信息会比第一种 OOM 堆栈多出一段信息：failed due to fragmentation (required continguous free “<< required_bytes << “bytes for a new buffer where largest contiguous free” << largest_continuous_free_pages << “ bytes)”；其详细代码在 art/runtime/gc/allocator/rosalloc.cc 中，这里不作详述。

创建线程失败

系统源码文件：/art/runtime/thread.cc

```
void Thread::CreateNativeThread(JNIEnv* env, jobject java_peer, size_t
stack_size, bool is_
daemon)
抛出时的错误信息：
    "Could not allocate JNI Env"
  或者
    StringPrintf("pthread_create (%s stack) failed: %s",
PrettySize(stack_size).c_str(),
strerror(pthread_create_result)));
```

这是创建线程时抛出的 OOM 错误，且有多种错误信息。源码这里不展开详述了，下面是根据源码整理的 Android 中创建线程的步骤，其中两个关键节点是创建 JNIEnv 结构体和创建线程，而这两步均有可能抛出 OOM。



创建 JNI 失败

创建 JNIEnv 可以归为两个步骤：

- 通过 Andorid 的匿名共享内存（Anonymous Shared Memory）分配 4KB（一个 page）内核态内存。
- 再通过 Linux 的 mmap 调用映射到用户态虚拟内存地址空间。

第一步创建匿名共享内存时，需要打开 /dev/ashmem 文件，所以需要一个 FD（文件描述符）。此时，如果创建的 FD 数已经达到上限，则会导致创建 JNIEnv 失败，抛出错误信息如下：

```
E/art: ashmem_create_region failed for 'indirect ref table': Too many
open files
 java.lang.OutOfMemoryError: Could not allocate JNI Env
   at java.lang.Thread.nativeCreate(Native Method)
   at java.lang.Thread.start(Thread.java:730)
```

第二步调用 mmap 时，如果进程虚拟内存地址空间耗尽，也会导致创建 JNIEnv 失败，抛出错误信息如下：

```
E/art: Failed anonymous mmap(0x0, 8192, 0x3, 0x2, 116, 0): Operation
not permitted. See
process maps in the log.
java.lang.OutOfMemoryError: Could not allocate JNI Env
  at java.lang.Thread.nativeCreate(Native Method)
```

```
    at java.lang.Thread.start(Thread.java:1063)
```

创建线程失败

创建线程也可以归纳为两个步骤：

1. 调用 mmap 分配栈内存。这里 mmap flag 中指定了 MAP_ANONYMOUS，即匿名内存映射。这是在 Linux 中分配大块内存的常用方式。其分配的是虚拟内存，对应页的物理内存并不会立即分配，而是在用到的时候触发内核的缺页中断，然后中断处理函数再分配物理内存。
2. 调用 clone 方法进行线程创建。

第一步分配栈内存失败是由于进程的虚拟内存不足，抛出错误信息如下：

```
W/libc: pthread_create failed: couldn't allocate 1073152-bytes mapped
space: Out of memory
W/tch.crowdsourc: Throwing OutOfMemoryError with VmSize  4191668 kB
"pthread_create
(1040KB stack) failed: Try again"
java.lang.OutOfMemoryError: pthread_create (1040KB stack) failed: Try
again
        at java.lang.Thread.nativeCreate(Native Method)
        at java.lang.Thread.start(Thread.java:753)
```

第二步 clone 方法失败是因为线程数超出了限制，抛出错误信息如下：

```
W/libc: pthread_create failed: clone failed: Out of memory
W/art: Throwing OutOfMemoryError "pthread_create (1040KB stack) failed:
Out of memory"
java.lang.OutOfMemoryError: pthread_create (1040KB stack) failed: Out
of memory
  at java.lang.Thread.nativeCreate(Native Method)
  at java.lang.Thread.start(Thread.java:1078)
```

OOM 问题定位

在分析清楚 OOM 问题的原因之后，我们对于线上的 OOM 问题就可以做到对症下药。而针对 OOM 问题，我们可以根据堆栈信息的特征来确定这是哪一个类型的

OOM，下面分别介绍使用 Probe 组件是如何去定位线上发生的每一种类型的 OOM 问题的。

堆内存不足

Android 中最常见的 OOM 就是 Java 堆内存不足，对于堆内存不足导致的 OOM 问题，发生 Crash 时的堆栈信息往往只是“压死骆驼的最后一根稻草”，它并不能有效帮助我们准确地定位到问题。

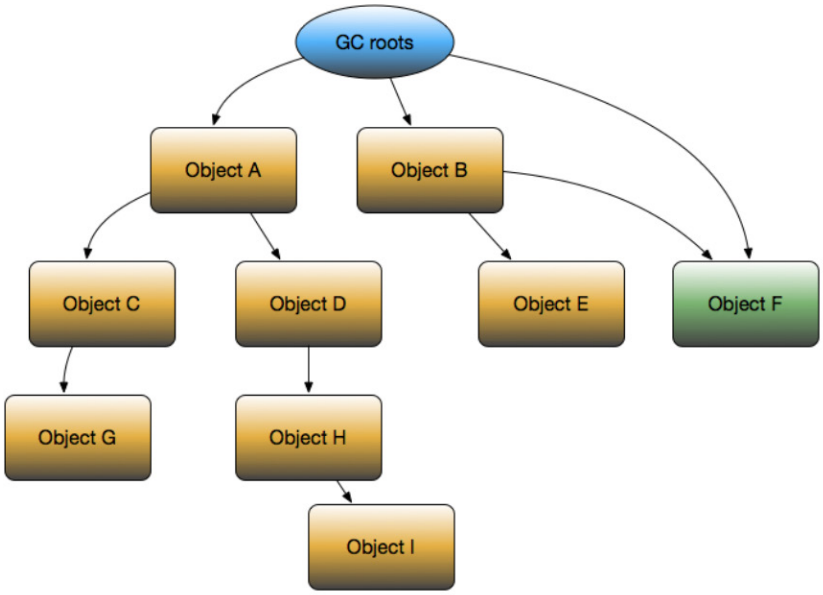
堆内存分配失败，通常说明进程中大部分的内存已经被占用了，且不能被垃圾回收器回收，一般来说此时内存占用都存在一些问题，例如内存泄漏等。要想定位到问题所在，就需要知道进程中的内存都被哪些对象占用，以及这些对象的引用链路。而这些信息都可以在 Java 内存快照文件中得到，调用 Debug.dumpHprof-Data(String fileName) 函数就可以得到当前进程的 Java 内存快照文件（即 HPROF 文件）。所以，关键在于要获得进程的内存快照，由于 dump 函数比较耗时，在发生 OOM 之后再去执行 dump 操作，很可能无法得到完整的内存快照文件。

于是 Probe 对于线上场景做了内存监控，在一个后台线程中每隔 1S 去获取当前进程的内存占用（通过 Runtime.getRuntime.totalMemory()-Runtime.getRuntime.freeMemory() 计算得到），当内存占用达到设定的阈值时（阈值根据当前系统分配给应用的最大内存计算），就去执行 dump 函数，得到内存快照文件。

在得到内存快照文件之后，我们有两种思路，一种想法是直接将 HPROF 文件回传到服务器，我们拿到文件后就可以使用分析工具进行分析。另一种想法是在用户手机上直接分析 HPROF 文件，将分析完得到的分析结果回传给服务器。但这两种方案都存在着一些问题，下面分别介绍我们在这两种思路的实践过程中遇到的挑战和对应的解决方案。

线上分析

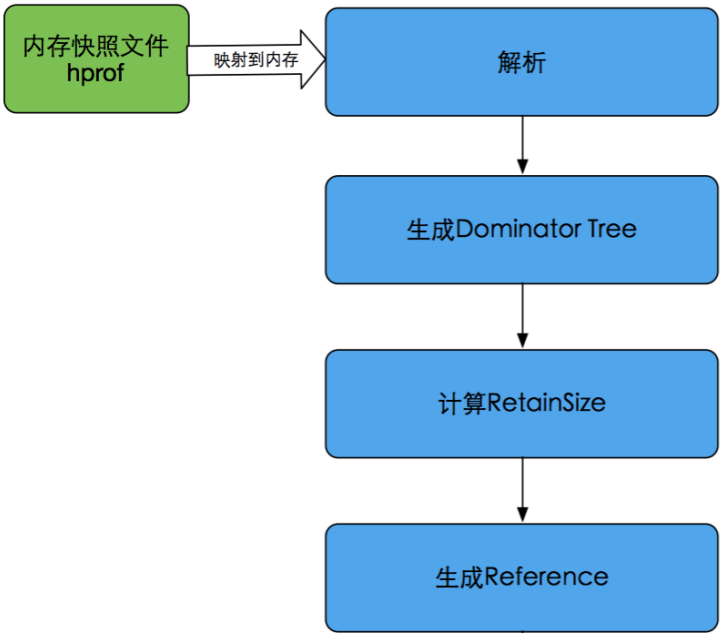
首先，我们介绍几个基本概念：



- **Dominator**：从 GC Roots 到达某一个对象时，必须经过的对象，称为该对象的 Dominator。例如在上图中，B 就是 E 的 Dominator，而 B 却不是 F 的 Dominator。
- **ShallowSize**：对象自身占用的内存大小，不包括它引用的对象。
- **RetainSize**：对象自身的 ShallowSize 和对象所支配的（可直接或间接引用到的）对象的 ShallowSize 总和，就是该对象 GC 之后能回收的内存总和。例如上图中，D 的 RetainSize 就是 D、H、I 三者的 ShallowSize 之和。

JVM 在进行 GC 的时候会进行可达性分析，当一个对象到 GC Roots 没有任何引用链相连（用图论的话来说，就是从 GC Roots 到这个对象不可达）时，则证明此对象是可回收的。

Github 上有一个开源项目 HAHA 库，用于自动解析和分析 Java 内存快照文件（即 HPROF 文件）。下面是 HAHA 库的分析步骤：



于是我们尝试在 App 中去新开一个进程使用 HAHA 库分析 HPROF 文件，在线下测试过程中遇到了几个问题，下面逐一进行叙述。

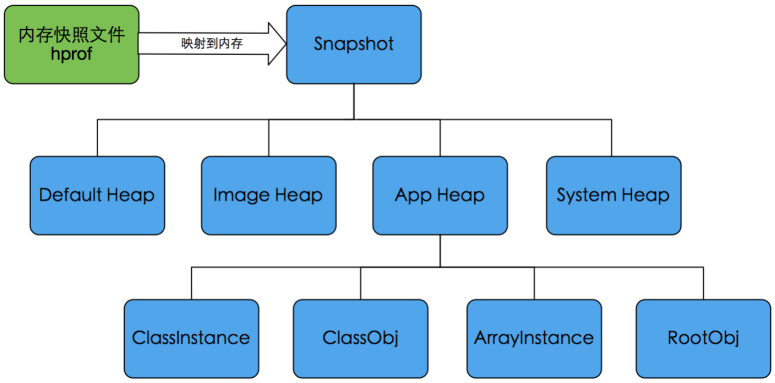
分析进程自身 OOM

测试时遇到的最大问题就是分析进程自身经常会发生 OOM，导致分析失败。为了弄清楚分析进程为什么会占用这么大内存，我们做了两个对比实验：

- 在一个最大可用内存 256MB 的手机上，让一个成员变量申请特别大的一块内存 200 多 MB，人造 OOM，Dump 内存，分析，内存快照文件达到 250 多 MB，分析进程占用内存并不大，为 70MB 左右。
- 在一个最大可用内存 256MB 的手机上，添加 200 万个小对象（72 字节），人造 OOM，Dump 内存，分析，内存快照文件达到 250 多 MB，分析进程占用内存增长很快，在解析时就发生 OOM 了。

实验说明，分析进程占用内存与 HPROF 文件中的 Instance 数量是正相关的，在将 HPROF 文件映射到内存中解析时，如果 Instance 的数量太大，就会导致 OOM。

HPROF 文件映射到内存中会被解析成 Snapshot 对象（如下图所示），它构建了一颗对象引用关系树，我们可以在这颗树中查询各个 Object 的信息，包括 Class 信息、内存地址、持有的引用以及被持有引用的关系。

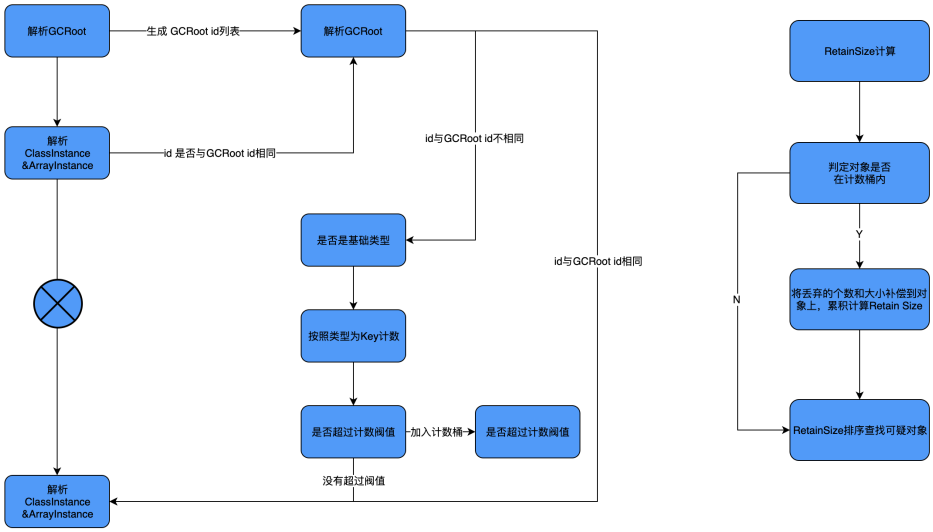


HPROF 文件映射到内存的过程：

```
// 1. 构建内存映射的 HprofBuffer 针对大文件的一种快速的读取方式，其原理是将文件流的通
道与  ByteBuffer 建立起关联，并只在真正发生读取时才从磁盘读取内容出来。
HprofBuffer buffer = new MemoryMappedFileBuffer(heapDumpFile);
// 2. 构造 Hprof 解析器
HprofParser parser = new HprofParser(buffer);
// 3. 获取快照
Snapshot snapshot = parser.parse();
// 4. 去重 gcRoots
deduplicateGcRoots(snapshot);
```

为了解决分析进程 OOM 的问题，我们在 HprofParser 的解析逻辑中加入了计数压缩逻辑（如下图），目的是在文件映射过程去控制 Instance 的数量。在解析过程中对于 ClassInstance 和 ArrayInstance，以类型为 key 进行计数，当同一类型的 Instance 数量超过阈值时，则不再向 Snapshot 中添加该类型的 Instance，只是记录 Intsance 被丢弃的数量和 Instance 大小。这样就可以控制住每一种类型的 Instance 数量，减少了分析进程的内存占用，在很大程度上避免了分析进程自身的 OOM 问题。既然我们在解析时丢弃了一部分 Instance，后面就得把丢弃的这部分找补回来，所以在计算 RetainSize 时我们会进行计数桶补偿，即把之前丢弃的相同类

型的 Instance 数量和大小都补偿到这个对象上，累积去计算 RetainSize。

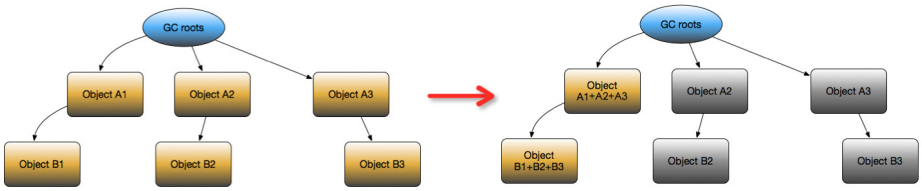


链路分析时间过长

在线下测试过程中还遇到了一个问题，就是在手机上进行链路分析的耗时太长。

使用 HAHA 算法在 PC 上可以快速地对所有对象都进行链路分析，但是在手机上由于性能的局限性，如果对所有对象都进行链路分析会导致分析耗时非常长。

考虑到 RetainSize 越大的对象对内存的影响也越大，即 RetainSize 比较大的那部分 Instance 是最有可能造成 OOM 的“元凶”。

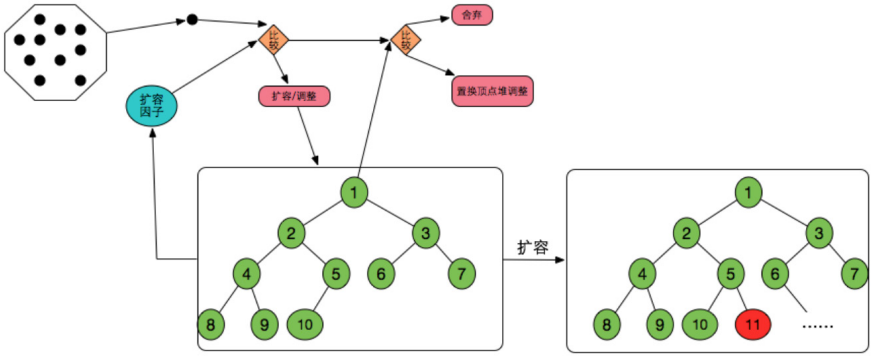


我们在生成 Reference 之后，做了一步链路归并（如上图），即对于同一个对象的不同 Instance，如果其底下的引用链路中的对象类型也相同，则进行归并，并记录 Instance 的个数和每个 Instance 的 RetainSize。

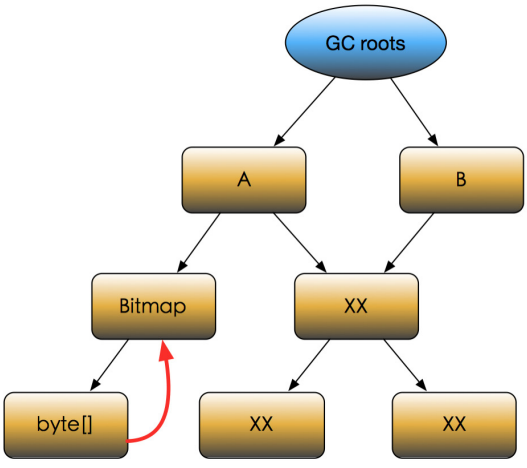
然后对归并后的 Instance 按 RetainSize 进行排序，取出 TOP N 的 Instance，

其中在排序过程中我们会对 N 的值进行动态调整，保证 RetainSize 达到一定阈值的 Instance 都能被发现。对于这些 Instance 才进行最后的链路分析，这样就能大大缩短分析时长。

排序过程：创建一个初始容量为 5 的集合，往里添加 Instance 后进行排序，然后遍历后面的 Instance，当 Instance 的 RetainSize 大于总共消耗内存大小的 5% 时，进行扩容，并重新排序。当 Instance 的 RetainSize 大于现有集合中的最小值时，进行替换，并重新排序。

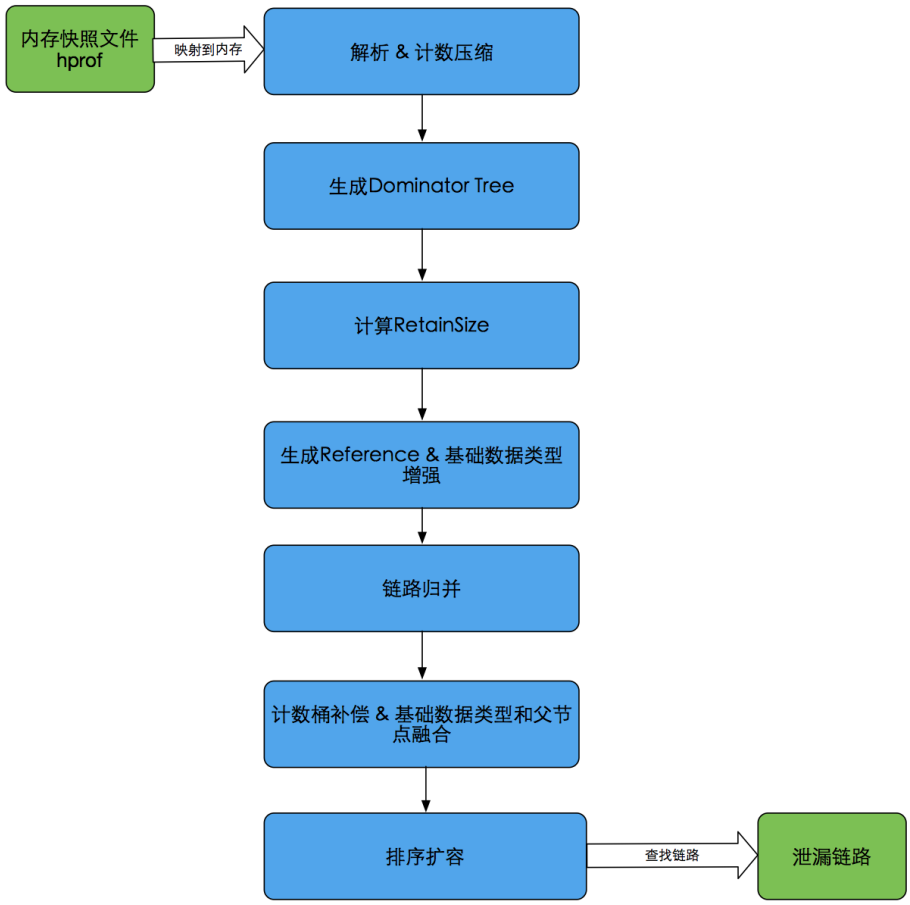


基础类型检测不到



为了解决 HAHA 算法中检测不到基础类型泄漏的问题，我们在遍历堆中的 Instance 时，如果发现是 ArrayInstance，且是 byte 类型时，将它自身舍弃掉，并将它的 RetainSize 加在它的父 Instance 上，然后用父 Instance 进行后面的排序。

至此，我们对 HAHA 的原始算法做了诸多优化（如下图），很大程度解决了分析进程自身 OOM 问题、分析时间过长问题以及基础类型检测不到的问题。



针对线上堆内存不足问题，Probe 最后会自动分析出 RetainSize 大小 Top N 对象到 GC Roots 的链路，上报给服务器，进行报警。下面是一个线上案例，这里截取了上报的链路分析结果中的一部分，完整的分析结果就是多个这样的组合。在第一段链路分析可以看到，有个 Bitmap 对象占用了 2MB 左右的内存，根据链路定

位到代码，修复了 Bitmap 泄漏问题。第二段链路分析反映的是一个 Timer 泄漏问题，可以看出内存中存在 4 个这样的 Instance，每个 Instance 的 Retain Size 是 595634，所以这个问题会泄漏的内存大小是 4*595634=2.27MB。

裁剪回捞 HPROF 文件

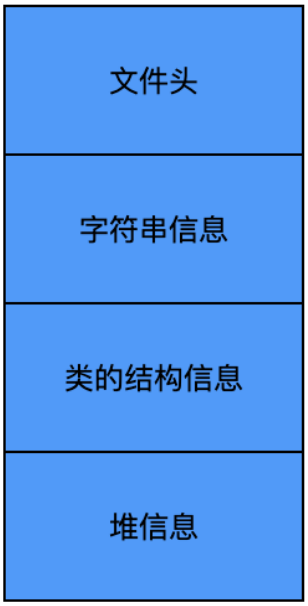
在 Probe 上线分析方案之后，发现尽管我们做了很多优化，但是受到手机自身性能的约束，线上分析的成功率也只有 65%。

于是，我们对另一种思路即回捞 HPROF 文件后本地分析进行了探索，这种方案最大的问题就是线上流量问题，因为 HPROF 文件动辄几百 MB，如果直接进行上传，势必会对用户的流量消耗带来巨大影响。

使用这种方案的关键点就在于减少上传的 HPROF 文件大小，减少文件大小首先想到的就是压缩，不过只是做压缩的话，文件还是太大。接下来，我们就考虑几百 MB 的文件内容是否都是我们需要的，是否可以对文件进行裁剪。我们希望对 HPROF 无用的信息进行裁剪，只保留我们关心的数据，就需要先了解 HPROF 文件的格式：

Debug.dumpHprofData() 其内部调用的是 VMDebug 的同名函数，层层深入最终可以找到 /art/runtime/hprof/hprof.cc，HPROF 的生成操作基本都是在这里执行的，结合 HAHA 库代码阅读 hrpof.cc 的源码。

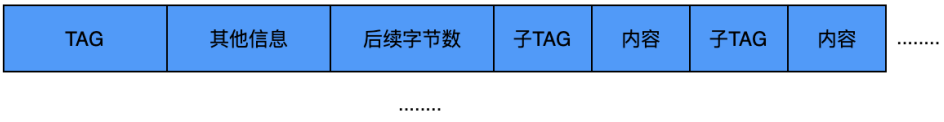
HPROF 文件的大体格式如下：



一个 HPROF 文件主要分为这四部分：

- 文件头。
- 字符串信息：保存着所有的字符串，在解析的时候通过索引 id 被引用。
- 类的结构信息：是所有 Class 的结构信息，包括内部的变量布局，父类的信息等等。
- 堆信息：即我们关心的内存占用与对象引用的详细信息。

其中我们最关心的堆信息是由若干个相同格式的元素组成，这些元素的大体格式如下图：



每个元素都有个 TAG 用来标识自己的身份，而后续字节数则表示元素的内容长度。元素携带的内容则是若干个子元素组合而成，通过子 TAG 来标识身份。

具体的 TAG 和身份的对应关系可以在 hrpof.cc 源码中找到，这里不进行展开。

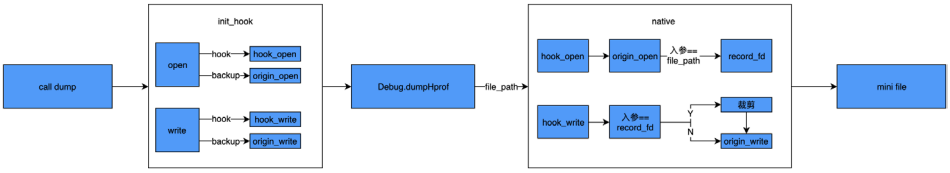
```
// Values for the first byte of HEAP_DUMP and HEAP_DUMP_SEGMENT records:
enum HprofHeapTag {
  // Traditional.
  HPROF_ROOT_UNKNOWN = 0xFF,
  HPROF_ROOT_JNI_GLOBAL = 0x01,
  HPROF_ROOT_JNI_LOCAL = 0x02,
  HPROF_ROOT_JAVA_FRAME = 0x03,
  HPROF_ROOT_NATIVE_STACK = 0x04,
  HPROF_ROOT_STICKY_CLASS = 0x05,
  HPROF_ROOT_THREAD_BLOCK = 0x06,
  HPROF_ROOT_MONITOR_USED = 0x07,
  HPROF_ROOT_THREAD_OBJECT = 0x08,
  HPROF_CLASS_DUMP = 0x20,
  HPROF_INSTANCE_DUMP = 0x21,
  HPROF_OBJECT_ARRAY_DUMP = 0x22,
  HPROF_PRIMITIVE_ARRAY_DUMP = 0x23,

  // Android.
  HPROF_HEAP_DUMP_INFO = 0xfe,
  HPROF_ROOT_INTERNED_STRING = 0x89,
  HPROF_ROOT_FINALIZING = 0x8a,  // Obsolete.
  HPROF_ROOT_DEBUGGER = 0x8b,
  HPROF_ROOT_REFERENCE_CLEANUP = 0x8c,  // Obsolete.
  HPROF_ROOT_VM_INTERNAL = 0x8d,
  HPROF_ROOT_JNI_MONITOR = 0x8e,
  HPROF_UNREACHABLE = 0x90,  // Obsolete.
  HPROF_PRIMITIVE_ARRAY_NODATA_DUMP = 0xc3,  // Obsolete.
};
```

弄清楚了文件格式，接下来需要确定裁剪内容。经过思考，我们决定裁减掉全部基本类型数组的值，原因是我们的使用场景一般是排查内存泄漏以及 OOM，只关心对象间的引用关系以及对象大小即可，很多时候对于值并不是很在意，所以裁减掉这部分的内容不会对后续的分析造成影响。

最后需要确定裁剪方案。先是尝试了 dump 后在 Java 层进行裁剪，发现效率很低，很多时候这一套操作下来需要 20s。然后又尝试了 dump 后在 Native 层进行裁剪，这样做效率是高了点，但依然达不到预期。

经过思考，如果能够在 dump 的过程中筛选出哪些内容是需要保留的，哪些内容是需要裁剪的，需要裁剪的内容直接不写入文件，这样整个流程的性能和效率绝对是最高的。

为了实现这个想法，我们使用了 GOT 表 Hook 技术（不展开介绍）。有了 Hook 手段，但是还没有找到合适的 Hook 点。通过阅读 hrpof.cc 的源码，发现最适合的点就是在写入文件时，拿到字节流进行裁剪操作，然后把有用的信息写入文件。于是项目最终的结构如下图：



我们对 IO 的关键函数 open 和 write 进行 Hook。Hook 方案使用的是爱奇艺开源的 xHook 库。

在执行 dump 的准备阶段，我们会调用 Native 层的 open 函数获得一个文件句柄，但实际执行时会进入到 Hook 层中，然后将返回的 FD 保存下来，用作 write 时匹配。

在 dump 开始时，系统会不断的调用 write 函数将内容写入到文件中。由于我们的 Hook 是以 so 为目标的，系统运行时也会有许多写文件的操作，所以我们需要对前面保存的 FD 进行匹配。若 FD 匹配成功则进行裁剪，否则直接调用 origin-write 进行写入操作。

流程结束后，就会得到裁剪后的 mini-file，裁剪后的文件大小只有原始文件大小的十分之一左右，用于线上可以节省大部分的流量消耗。拿到 mini-file 后，我们将裁剪部分的位置填上字节 0 来进行恢复，这样就可以使用传统工具打开进行分析了。

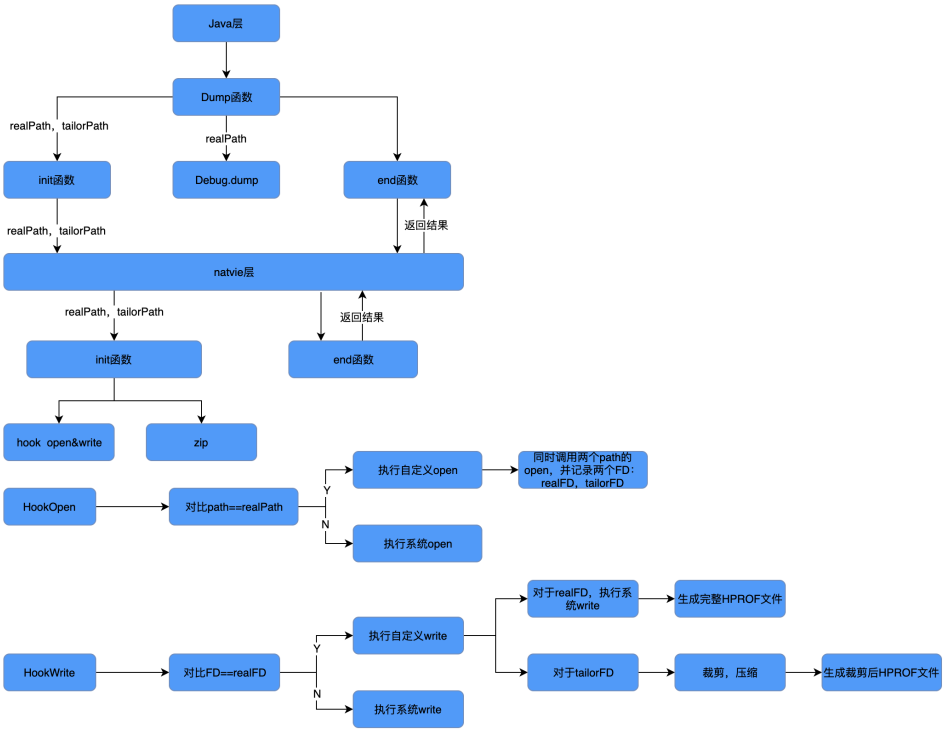
原始 HPROF 文件和裁剪后再恢复的 HPROF 文件分别在 Android Studio 中打开，发现裁剪再恢复的 HPROF 文件打开后，只是看不到对象中的基础数据类型值，而整个的结构、对象的分布以及引用链路等与原始 HPROF 文件是完全一致的。事实证明裁剪方案不会影响后续对堆内存的链路分析。

方案融合

由于目前裁剪方案在部分机型上（主要是 Android 7.X 系统）不起作用，所以在 Probe 中同时使用了这两种方案，对两种方案进行了融合。即通过一次 dump 操作

得到两份 HPROF 文件，一份原始文件用于下次启动时分析，一份裁剪后的文件用于上传服务器。

Probe 的最终方案实现如下图，主要是在调用 dump 函数之前先将两个文件路径（希望生成的原始文件路径和裁剪文件路径）传到 Native 层，Native 层记录下两个文件路径，并对 open 和 write 函数进行 Hook。hookopen 函数主要是通过 open 函数传入的 path 和之前记录的 path 比对，如果相同，我们就会同时调用之前记录的两个 path 的 open，并记录下两个 FD，如果不相同则直接调原生 open 函数。hookwrite 函数主要是通过传入的 FD 与之前 hookopen 中记录的 FD 比对，如果相同会先对原始文件对应的 FD 执行原生 write，然后对裁剪文件对应的 FD 执行我们自定义的 write，进行裁剪压缩。这样再传入原始文件路径调用系统的 dump 函数，就能够同时得到一份完整的 HPROF 文件和一份裁剪后的 HPROF 文件。



线程数超出限制

对于创建线程失败导致的 OOM，Probe 会获取当前进程所占用的虚拟内存、进程中的线程数量、每个线程的信息（线程名、所属线程组、堆栈信息）以及系统的线程数限制，并将这些信息上传用于分析问题。

/proc/sys/kernel/threads-max 规定了每个进程创建线程数目的上限。在华为的部分机型上，这个上限被修改的很低（大约 500），比较容易出现线程数溢出的问题，而大部分手机这个限制都很大（一般为 1W 多）。在这些手机上创建线程失败大多都是因为虚拟内存空间耗尽导致的，进程所使用的虚拟内存可以查看 /proc/pid/status 的 VmPeak/VmSize 记录。

然后通过 Thread.getAllStackTraces() 可以得到进程中的所有线程以及对应的堆栈信息。

一般来说，当进程中线程数异常增多时，都是某一类线程被大量的重复创建。所以我们只需要定位到这类线程的创建时机，就能知道问题所在。如果线程是有自定义名称的，那么直接就可以在代码中搜索到创建线程的位置，从而定位问题，如果线程创建时没有指定名称，那么就需要通过该线程的堆栈信息来辅助定位。下面这个例子，就是一个“crowdSource msg”的线程被大量重复创建，在代码中搜索名称很快就查出了问题。针对这类线程问题推荐的做法就是在项目中统一使用线程池，可以很大程度上避免线程数的溢出问题。

线程信息：

```
thread name: Thread[nio_tunnel_handler,5,main]   count: 1
thread name: Thread[OkHttp Dispatcher,5,main]   count: 30
thread name: Thread[process_read_thread,5,main]   count: 4
thread name: Thread[Jit thread pool worker thread 0,5,main]   count: 1
thread name: Thread[crowdSource msg,5,main]   count: 202
thread name: Thread[Timer-4,5,main]   count: 1
thread name: Thread[mqt_js,5,main]   count: 1

threadnames:Thread[Thread-5,5,main] count:1
trace:
java.lang.Object.wait(NativeMethod)
com.dianping.networklog.d.run(UnknownSource:28)
```

FD 数超出限制

前面介绍了，当进程中的 FD 数量达到最大限制时，再去新建线程，在创建 JNIEnv 时会抛出 OOM 错误。但是 FD 数量超出限制除了会导致创建线程抛出 OOM 以外，还会导致很多其它的异常，为了能够统一处理这类 FD 数量溢出的问题，Probe 中对进程中的 FD 数量做了监控。在后台启动一个线程，每隔 1s 读取一次当前进程创建的 FD 数量，当检测到 FD 数量达到阈值时（FD 最大限制的 95%），读取当前进程的所有 FD 信息归并后上报。

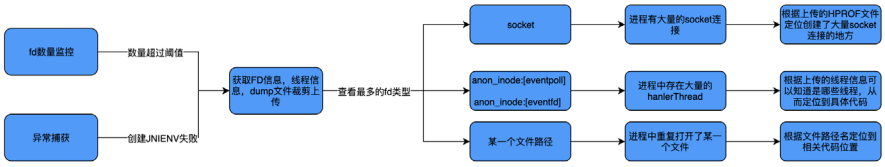
在 /proc/pid/limits 描述着 Linux 系统对对应进程的限制，其中 Max open files 就代表可创建 FD 的最大数目。

进程中创建的 FD 记录在 /proc/pid/fd 中，通过遍历 /proc/pid/fd，可以得到 FD 的信息。

获取 FD 信息：

```
File fdFile=new File("/proc/" + Process.myPid() + "/fd");
File[] files = fdFile.listFiles();
int length = files.length; // 即进程中的 fd 数量
for (int i = 0; i < length ; i++) {
  if (Build.VERSION.SDK_INT >= 21) {
         Os.readlink(files[i].getAbsolutePath()); // 得到软链接实际指向的文件
     } else {
      //6.0 以下系统可以通过执行 readlink 命令去得到软连接实际指向文件，但是耗时较久
  }
}
```

得到进程中所有的 FD 信息后，我们会先按照 FD 的类型进行一个归并，FD 的用途主要有打开文件、创建 socket 连接、创建 handlerThread 等。



比如像下面这个例子中，就是 anon_inode:[eventpoll] 和 anon_inode:[eventfd] 的数量异常的多，说明进程中很可能是启动了大量的 handlerThread，再结合回传

上来的线程信息就能快速定位到问题代码的具体位置。

FD 溢出案例：

FD 信息：

```
anon_inode:[eventpoll]   count: 381
anon_inode:[eventfd]   count: 381
pipe   count  26
socket   count  32
/system/framework/framework-res.apk   count: 1
 ..........
Thread 信息:
thread name: Thread[Jit thread pool worker thread 0,5,main]   count: 1
thread name: Thread[mtqq handler,5,main]   count: 302
thread name: Thread[Timer-4,5,main]   count: 1
thread name: Thread[mqt_js,5,main]   count: 1
```

总结

Probe 目前能够有效定位线上 Java 堆内存不足、FD 泄漏以及线程溢出的 OOM 问题。骑手 Android 端使用 Probe 组件解决了很多线上的 OOM 问题，将线上 OOM Crash 率从最高峰的 2‰降低到了现在的 0.02‰左右。我们后续也会继续完善 Probe 组件，例如 HPROF 文件裁剪方案对 7.X 系统的兼容以及 Native 层的内存问题定位。

作者简介

逢搏，美团配送 App 团队研发工程师。
毅然，美团配送 App 团队高级技术专家。
永刚，美团平台监控团队研发工程师。

招聘信息

美团配送 App 团队，负责美团骑手、美团众包、美团跑腿等配送相关 App 的研发，涉及技术领域包括但不限于 App 的稳定性建设、App 性能监控和优化、大前端跨平台动态化、App 安全。对上述领域感兴趣的请联系 tech#meituan.com。

活动 Web 页面人机识别验证的探索与实践

益国

在电商行业，线上的营销活动特别多。在移动互联网时代，一般为了活动的快速上线和内容的即时更新，大部分的业务场景仍然通过 Web 页面来承载。但由于 Web 页面天生“环境透明”，相较于移动客户端页面在安全性上存在更大的挑战。本文主要以移动端 Web 页面为基础来讲述如何提升页面安全性。

活动 Web 页面的安全挑战

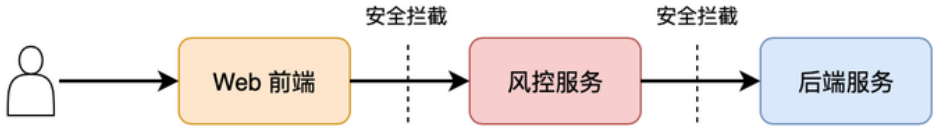
对于营销活动类的 Web 页面，领券、领红包、抽奖等活动方式很常见。此类活动对于普通用户来说大多数时候就是“拼手气”，而对于非正常用户来说，可以通过直接刷活动 API 接口的“作弊”方式来提升“手气”。这样的话，对普通用户来说，就变得很不公平。

对于活动运营的主办方来说，如果风控措施做的不好，这类刷接口的“拼手气”方式可能会对企业造成较大的损失。如本来计划按 7 天发放的红包，在上线 1 天就被刷光了，活动的营销成本就会被意外提升。主办方想发放给用户的满减券、红包，却大部分被黄牛使用自动脚本刷走，而真正想参与活动的用，却无法享受活动优惠。

终端用户到底是人还是机器，网络请求是否为真实用户发起，是否存在安全漏洞并且已被“羊毛党”恶意利用等等，这些都是运营主办方要担心的问题。

安全防范的基本流程

为了提升活动 Web 页面的安全性，通常会引入专业的风控服务。引入风控服务后，安全防护的流程大致如图所示。



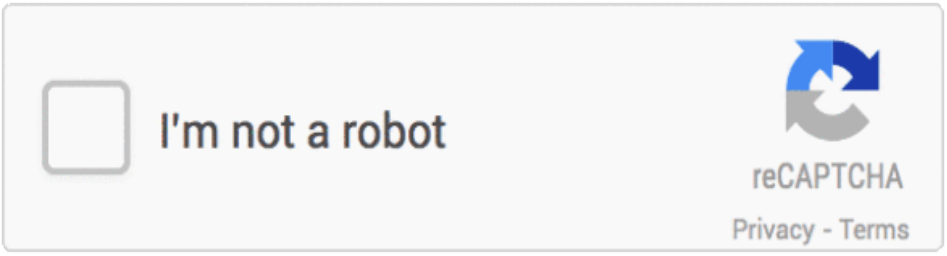
- Web 前端：用户通过 Web 页面来参与活动，同时 Web 前端也会收集用于人机识别验证的用户交互行为数据。由于不同终端（移动端 H5 页面和 PC 端页面）交互形式不同，收集用户交互行为数据的侧重点也会有所不同。
- 风控服务：一般大公司都会有专业的风控团队来提供风控服务，在美团内部有智能反爬系统来基于地理位置、IP 地址等大数据来提供频次限制、黑白名单限制等常规的基础风控拦截服务。甚至还有依托于海量的全业务场景的用户大数据，使用贝叶斯模型、神经网络等来构建专业度较深的服务。风控服务可以为 Web 前端提供通用的独立验证 SDK：验证码、滑块验证等区分人机的“图灵验证”，也可以为服务端提供 Web API 接口的验证等。
- 后端业务服务：负责处理活动业务逻辑，如给用户发券、发红包，处理用户抽奖等。请求需要经过风控服务的验证，确保其安全性，然后再来处理实际业务逻辑，通常，在处理完实际业务逻辑时，还会有针对业务本身的风控防范。

对于活动 Web 页面来说，加入的风控服务主要为了做人机识别验证。在人机识别验证的专业领域上，我们可以先看看业界巨头 Google 是怎么做的。

Google 如何处理人机验证

Google 使用的人机验证服务是著名的 reCAPTCHA（Completely Automated Public Turing Test To Tell Computers and Humans Apart，区分人机的全自动图灵测试系统），也是应用最广的验证码系统。早年的 reCAPTCHA 验证码是这样的：

如今的 reCAPTCHA 已经不再需要人工输入难以识别的字符，它会检测用户的终端环境，追踪用户的鼠标轨迹，只需用户点击“我不是机器人”就能进行人机验证（reCAPTCHA 骗用户进行数据标注而进行 AI 训练的验证另说）。



reCAPTCHA 的验证方式从早先的输入字符到现在的轻点按钮，在用户体验上，有了较大的提升。

而在活动场景中引入人机识别验证，如果只是简单粗暴地增加验证码，或者只是像 reCAPTCHA 那样增加点击“我不是机器人”的验证，都会牺牲用户体验，降低用户参加活动的积极性。

Google 的普通 Web 页面的浏览和有强交互的活动 Web 页面虽是不同的业务场景，但对于活动 Web 页面来说，强交互恰好为人机识别验证提供了用户交互行为数据收集的契机。

人机识别验证的技术挑战

理想的方案是在用户无感知的情况下做人机识别验证，这样既确保了安全又对用户体验无损伤。

从实际的业务场景出发再结合 Web 本身的环境，如果想实现理想的方案，可能会面临如下的技术挑战：

- （1）需要根据用户的使用场景来定制人机识别验证的算法：Web 前端负责收集、上报用户交互行为数据，风控服务端校验上报的数据是否符合正常的用户行为逻辑。
- （2）确保 Web 前端和风控服务端之间通信和数据传输的安全性。

- （3）确保上述两大挑战中提到的逻辑和算法不会被代码反编译来破解。

在上述的三个挑战中，（1）已经实现了人机识别验证的功能，而（2）和（3）都是为了确保人机识别验证不被破解而做的安全防范。接下来，本文会分别针对这三个技术挑战来说明如何设计技术方案。

挑战一：根据用户使用场景来定制人机识别验证算法

先来分析一下用户的使用场景，正常用户参与活动的步骤是用户进入活动页面后，会有短暂的停留，然后点击按钮参与活动。这里所说的“参与活动”，最终都会在活动页面发起一个接口的请求。如果是非正常用户，可以直接跳过以上的实际动作而去直接请求参与活动的接口。

那么区别于正常用户和非正常用户就是那些被跳过的动作，对实际动作进一步归纳如下：

1. 进入页面。
2. 短暂的停留。
3. 滚动页面。
4. 点击按钮。

以上的动作又可以分为必需的操作和可选的操作。对这一连串动作产生的日志数据进行收集，在请求参与活动的接口时，将这些数据提交至后端，验证其合法性。这就是一个简单的人机识别验证。

在验证动作的合法性时，需要考虑到这些动作数据是不是能被轻易模拟。另外，动作的发生应该有一条时间线，可以给每个动作都增加一个时间戳，比如点击按钮肯定是在进入页面之后发生的。

一些特定的动作的日志数据也会有合理的区间，进入页面的动作如果以 JS 资源加载的时间为基准，那么加载时间可能大于 100 毫秒，小于 5 秒。而对于移动端的按钮点击，点击时记录的坐标值也会有对应的合理区间，这些合理的区间会根据实际的环境和情况来进行设置。

除此之外，设备环境的数据也可以进行收集，包括用户参与活动时使用的终端类型、浏览器的类型、浏览器是否为客户端的容器等，如果使用了客户端，客户端是否会携带特殊的标识等。

最后，还可以收集一些“无效”的数据，这些数据用于障人耳目，验证算法会将其忽略。尽管收集数据的动作是透明的，但是验证数据合法性不是透明的，攻击者无法知道，验证的算法中怎么区分哪些是有效、哪些是无效。这已经有点“蜜罐数据”的意思了。

挑战二：确保通信的安全性

收集的敏感数据要发送给风控服务端，进而确保通信过程的安全。

1. Web API 接口不能被中途拦截和篡改，通信协议使用 HTTPS 是最基本的要求；同时还要让服务端生成唯一的 Token，在通信过程中都要携带该 Token。
2. 接口携带的敏感数据不能是明文的，敏感数据要进行加密，这样攻击者无法通过网络抓包来详细了解敏感数据的内容。

Token 的设计

Token 是一个简短的字符串，主要为了确保通信的安全。用户进入活动 Web 页面后，请求参与活动的接口之前，会从服务端获取 Token。该 Token 的生成算法要确保 Token 的唯一性，通过接口或 Cookie 传递给前端，然后，前端在真正请求参与活动的接口时需要带上该 Token，风控服务端需要验证 Token 的合法性。也就是说，Token 由服务端生成，传给前端，前端再原封不动的回传给服务端。一旦加入了 Token 的步骤，攻击者就不能直接去请求参与活动的接口了。

Token 由风控服务端基于用户的身份，根据一定的算法来生成，无法伪造，为了提升安全等级，Token 需要具有时效性，比如 10 分钟。可以使用 Redis 这类缓存服务来存储 Token，使用用户身份标识和 Token 建立 KV 映射表，并设置过期时间为 10 分钟。

虽然前端在 Cookie 中可以获取到 Token，但是前端不能对 Token 做持久化的

缓存。一旦在 Cookie 中获取到了 Token，那么前端可以立即从 Cookie 中删除该 Token，这样能尽量确保 Token 的安全性和时效性。Token 存储在 Redis 中，也不会因为用户在参与活动时频繁的切换页面请求，而对服务造成太大的压力。

另外，Token 还可以有更多的用处：

- 标识参与活动用户的有效性。
- 敏感数据对称加密时生成动态密钥。
- API 接口的数字签名。

敏感数据加密

通信时，传递的敏感数据可以使用常见的对称加密算法进行加密。

为了提升加密的安全等级，加密时的密钥可以动态生成，前端和风控服务端约定好动态密钥的生成规则即可。加密的算法和密钥也要确保不被暴露。

通过对敏感数据加密，攻击者在不了解敏感数据内容的前提下就更别提模拟构造请求内容了。

挑战三：化解纸老虎的尴尬

有经验的 Web 开发者看到这里，可能已经开始质疑了：在透明的前端环境中折腾安全不是白折腾吗？这就好比费了很大的劲却只是造了一个“纸老虎”，质疑是有道理的，但是且慢，通过一些安全机制的加强是可以让“纸老虎”尽可能的逼真。

本文一再提及的 Web 环境的透明性，是因为在实际的生产环境中的问题：前端的代码在压缩后，通过使用浏览器自带的格式化工具和断点工具，仍然具备一定的可读性，花点时间仍然可以理解代码的逻辑，这就给攻击者提供了大好的代码反编译机会。

如果要化解“纸老虎”的尴尬，就要对前端的代码进行混淆。

前端代码混淆

前端的 JS 代码压缩工具基本都是对变量、函数名称等进行缩短，压缩对于混淆的作用是比较弱。除了对代码进行压缩，还需要进行专门的混淆。

对代码进行混淆可以降低可读性，混淆工具有条件的话最好自研，开源的工具要慎用。或者基于 Uglify.js 来自定义混淆的规则，混淆程度越高可读性就越低。

代码混淆也需要把握一个度，太复杂的混淆可能会让代码无法运行，也有可能会影响本身的执行效率。同时还需要兼顾混淆后的代码体积，混淆前后的体积不能有太大的差距，合理的混淆程度很重要。

```
!function(e,u,A,l,p,O,k,g,h,j,v,d,w,x,U,m,F,I,D,E,R,C,P,S,T,q,z,B,G,H,r){module.exports=function(e)
{return function e(r,a,t){if(!(r instanceof Object))return r;var n;switch(Object[p][l][A](r)[u](0,-k))
{case g:n=[];break;case h:n=new Date(r[j]());break;case v:n=new RegExp(r);break;case d:break;case
w:case x:case U:case m:case F:case I:case D:case E:case R:n=r[C]();break;default:n={}}if(a[P](r),t[P]
(n),r instanceof Array)for(var c=S;c<r[T];c++)n[c]=e(r[c],a,t);for(var s=Object[z](r)[q](),i=Object[z]
(n)[q](),o=S;o<s[T];o++){var y=s[o];if(i[T]>S&&y===i[S])i[B]();else if(Object[p][G][A](r,y)){var
f=r[y],b=a[H](f);n[y]=b!==-k?t[b]:e(r[y],a,t)}}return n}(e,[],[])}}
(0,"slice","call","toString","prototype",8,1,"Array","Date","getTime","RegExp","Function","Uint8Array",
"Uint8ClampedArray","Uint16Array","Uint32Array","Int8Array","Int16Array","Int32Array","Float32Array","F
loat64Array","subarray","push",0,"length","sort","keys","shift","hasOwnProperty","indexOf");
```

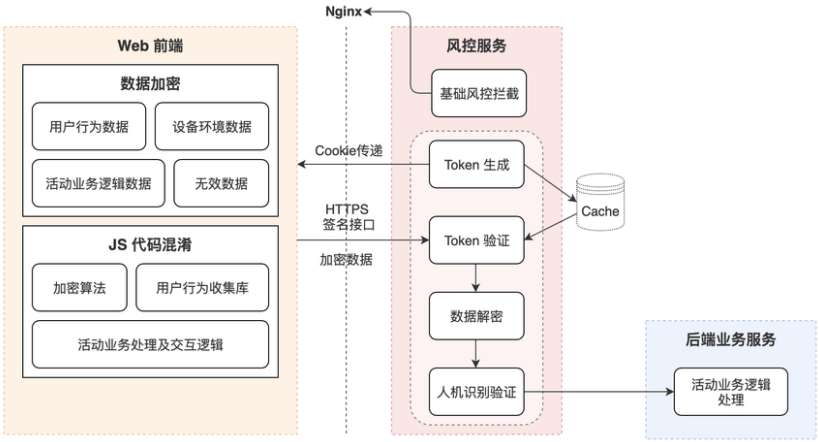
断点工具的防范会更麻烦些。在使用断点工具时通常都会导致代码延迟执行，而正常的业务逻辑都会立即执行，这是一个可以利用的点，可以考虑在代码执行间隔上来防范断点工具。

通过代码混淆和对代码进行特殊的处理，可以让格式化工具和断点工具变得没有用武之地。唯一有些小遗憾，就是处理后的代码也不能正常使用 Source Map 的功能了。

有了代码混淆，反编译的成本会非常高，这样“纸老虎”已经变得很逼真了。

技术方案设计

在讲解完如何解决关键的技术挑战后，就可以把相应的方案串起来，然后设计成一套可以实施的技术方案了。相对理想的技术方案架构图如下：



下面会按步骤来讲解技术方案的处理流程：

Step 0　基础风控拦截

基础风控拦截是上面提到的频次、名单等的拦截限制，在 Nginx 层就能直接实施拦截。如果发现是恶意请求，直接将请求过滤返回 403，这是初步的拦截，用户在请求 Web 页面的时候就开始起作用了。

Step 1　风控服务端生成 Token 后传给前端

Step 0 可能还没进入到活动 Web 页面，进入活动 Web 页面后才真正开始人机识别验证的流程，前端会先开始获取 Token。

Step 2　前端生成敏感数据

敏感数据应包含用户交互行为数据、设备环境数据、活动业务逻辑数据以及无效数据。

Step 3　使用 HTTPS 的签名接口发送数据

Token 可以作为 Authorization 的值添加到 Header 中，数据接口的签名可以有效防止 CSRF 的攻击。

Step 4　数据接口的校验

风控服务端收到请求后，会先验证数据接口签名中的 Token 是否有效。验证完 Token，才会对敏感数据进行解密。数据解密成功，再进一步对人机识别的数据合法

性进行校验。

Step 5 业务逻辑的处理

前面的步骤为了做人机识别验证，这些验证不涉及到业务逻辑。在所有这些验证都通过后，后端业务服务才会开始处理实际的活动业务逻辑。处理完活动业务逻辑，最终才会返回用户参与活动的结果。

总结

为了提升活动 Web 页面的安全性，使用了各种各样的技术方案，我们将这些技术方案组合起来才能发挥安全防范的作用，如果其中某个环节处理不当，都可能会被当作漏洞来利用，从而导致整个验证方案被攻破。

为了验证技术方案的有效性，可以持续观察活动 API 接口的请求成功率。从请求成功率的数据中进一步分析“误伤”和“拦截”的数据，以进一步确定是否要对方案进行调优。

通过上述的人机识别验证的组合方案，可以大幅提升活动 Web 页面的安全性。在活动 Web 页面应作为一个标准化的安全防范流程，除了美团，像淘宝和天猫也有类似的流程。由于活动运营的环节和方法多且复杂，仅仅提升了 Web 页面也不敢保证就是铁板一块，安全需要关注的环节还很多，安全攻防是一项长期的“拉锯升级战”，安全防范措施也需要持续地优化升级。

参考资料

- https://www.google.com/recaptcha/intro/v3.html
- https://segmentfault.com/a/1190000006226236
- https://www.freebuf.com/articles/web/102269.html

作者简介

益国，美团点评 Web 前端开发工程师。2015 年加入美团，曾先后负责过风控前端 SDK 和活动运营平台的研发，现负责大数据平台的研发工作。

React Native 工程中 TSLint 静态检查工具的探索之路

家正

背景

建立的代码规范没人遵守，项目中遍地风格迥异的代码，你会不会抓狂?

通过测试用例的程序还会出现 Bug，而原因仅仅是自己犯下的低级错误，你会不会抓狂?

某种代码写法存在问题导致崩溃时，只能全工程检查代码，这需要人工花费大量时间 Review 代码，你会不会抓狂?

以上这些问题，可以通过静态检查有效地缓解!

静态检查（Static Program Analysis）主要是以不运行程序的方式对于程序源代码进行检查分析的技术，而与之相反的就是动态检查（Dynamic Program Analysis），通过实际运行程序输入测试数据产生预期结果的技术。通过代码静态检查，我们可以快速定位代码的错误与缺陷，可以减少逐行阅读代码浪费的时间，可以（根据需要）快速扫描代码中可能存在的漏洞等。代码静态检查可以在代码的规范性、安全性、可靠性、可维护性等方面起到重要作用。

在客户端中，Android 可以使用 CheckStyle、Lint、Findbugs、PMD 等工具，iOS 可以使用 Clang Static Analyzer、OCLint 等工具。而在 React Native 的开发过程中，针对于 JavaScript 的 ESLint，与 TypeScript 的 TSLint，则成为了主要代码静态检查的工具。本文将按照使用 TSLint 的原因、使用 TSLint 的方法、自定义 TSLint 的步骤进行探究分析。

一、使用 TSLint 的原因

在客户端团队进入 React Native 项目的开发过程中，面临着如下问题：

1. 由于大家从客户端转入到 React Native 开发过程中，容易出现低级语法错误；
2. 开发者之前从事 Android、iOS、前端等工作，因此代码风格不同，导致项目代码风格不统一；
3. 客户端效果不一致，有可能 Android 端显示正常、iOS 端显示异常，或者相反的情况出现。

虽然以上问题可以通过多次不断将雷点标记出，并不断地分享经验与强化代码 Review 过程等方式来进行缓解，但是仍面临着 React Native 开发者掌握的技术水平千差万别，知识分享传播的速度缓慢等问题，既导致了开发成本的不断增加和开发效率持续低下的问题，还难以避免一个坑被踩了多次的情况出现。这时急需一款可以满足以下目标的工具：

1. 可检测代码低级语法错误；
2. 规范项目代码风格；
3. 根据需要可自定义检查代码的逻辑；
4. 工具使用者可以“傻瓜式”的接入部署到开发 IDE 环境；
5. 可以快速高效地将检查工具最新检查逻辑同步到开发 IDE 环境中；
6. 对于检查出的问题可以快速定位。

根据上述要求的描述，静态检查工具 TSLint 可以较为有效地达成目标。

二、TSLint 介绍

TSLint 是硅谷企业 Palantir 的一个项目，它是一款可以检查 TypeScript 代码可读性、可维护性以及功能性错误的静态检查工具，当前许多编辑器（Editors）和构建系统（Build Systems）支持这一工具，同时支持自定义编写 Lint 规则、配置、格式化等。

当前 TSLint 已经包含了上百条规则，这些规则构筑了当前 TSLint 检查的基础。在代码开发阶段中，通过这些配置好的规则可以给工程一个完整的检查，并随时可以提示出可能存在的问题。本文内容参考了 TSLint 官方文档 https://palantir.github.io/tslint/。

TSLint 常见规则

以下规则主要来源于 TSLint 规则，是某些规则的简单介绍。

规则	含义	错误示例

class-name	class的命名规则，主要是首字母大写的pascal命名法	interface someInterface {}
		~~ [0] //提示命名不规范
		class Another_Invalid_Class_Name {
		~~ [0] //提示命名不规范
		}

no-unsafe-finally	不许在finally里面使用return\continue\break\throws	function() {
		try {
		} finally {
		return;
		~~ [return] //提示不许在finally里面使用return
		}
		}

align 可配置参数 "arguments", "elements", "members", "parameters", "statements	垂直对齐，用于代码风格	var invalidConstructorArgsAlignment = new foo(10, "abcd"); ~~~~~~~ [arguments are not aligned] //提示未对齐
no-empty	不允许空块出现	if (x === 1) {} ~~ [block is empty] //提示块内为空

no-switch-case-fall-through	不允许switch语句中直接穿过case，意思为必须有break，防止没有break，导致代码进入下个case里面去	witch (foo) {
		case 1:
		bar();
		case 2:
		~~ [expected a 'break' before 'case'] //提示在case关键字前要有break
		bar();
		bar();
		case 3:
		~~ [expected a 'break' before 'case']
		case 4:
		default:
		break;
		}

...	...	...

TSLint 规则示例

常用 TSLint 规则包

上述 2.1 所列出的规则来源于 Palantir 官方 TSLint 规则。实际还有多种，可能会用到的有以下：

TSLint规则包	规则包内容
tslint	Palantir官方推出的规则，是最基础的标准规则包
tslint-react	Palantir官方推出的规则，主要用于React & JSX的检查
tslint-microsoft-contrib	微软官方推出的规则，用于微软项目的一个TSLint规则包
tslint-consistent-codestyle	主要用于统一代码风格的规则包
tslint-config-airbnb	适用于Airbnb 代码风格统一的工程，其中此工程没有新规则，主要是基于tslint-consistent-codestyle、tslint-eslint-rules、tslint-microsoft-contrib这三个规则包进行配置，根据自己需求开启或者关闭规则
tslint-eslint-rules	主要是补充对应于ESLint规则中TSLint规则缺失的部分
...	...

TSLint 规则示例

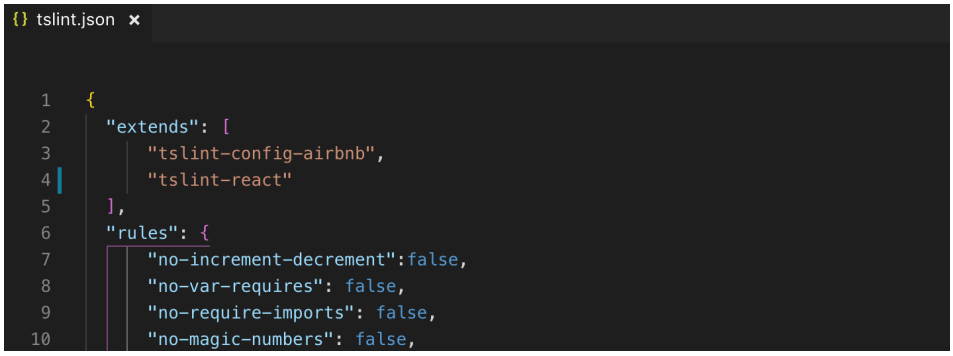
我们在项目的规则配置过程中，一般采用上述规则包其中一种或者若干种同时配置，那如何配置呢？请看下文。

三、如何进行 TSLint 规则配置与检查

首先，在工程 package.json 文件中配置 TSLint 包：

TSLint 规则示例

在根目录中的 tslint.json 文件中可以根据需要配置已有规则，例如：



TSLint 规则示例

其中 extends 数组内放置继承的 TSLint 规则包，上图包括了 airbnb 配置的规则包、tslint-react 的规则包，而 rules 用于配置规则的开关。

TSLint 规则目前只有 true 和 false 的选项，这导致了结果要么正常，要么报错 ERROR，而不会出现 WARNING 等警告。

有些时候，虽然配置某些规则开启，但是某个文件内可能会关闭某些甚至全部规则检查，这时候可以通过规则注释来配置，如：

```
/* tslint:disable */
```

上述注释表示本文件自此注释所在行开始，以下的所有区域关闭 TSLint 规则检查。

```
/* tslint:enable */
```

上述注释表示本文件自此注释所在行开始，以下的所有区域开启 TSLint 规则检查。

```
/* tslint:disable:rule1 rule2 rule3... */
```

上述注释表示本文件自此注释所在行开始，以下的所有区域关闭规则 rule1 rule2 rule3…的检查。

```
/* tslint:enable:rule1 rule2 rule3... */
```

上述注释表示本文件自此注释所在行开始，以下的所有区域开启规则 rule1 rule2 rule3…的检查。

```
// tslint:disable-next-line
```

上述注释表示此注释所在行的下一行关闭 TSLint 规则检查。

```
someCode(); // tslint:disable-line
```

上述注释表示此注释所在行关闭 TSLint 规则检查。

```
// tslint:disable-next-line:rule1 rule2 rule3...
```

上述注释表示此注释所在行的下一行关闭规则 rule1 rule2 rule3…的检查检查。

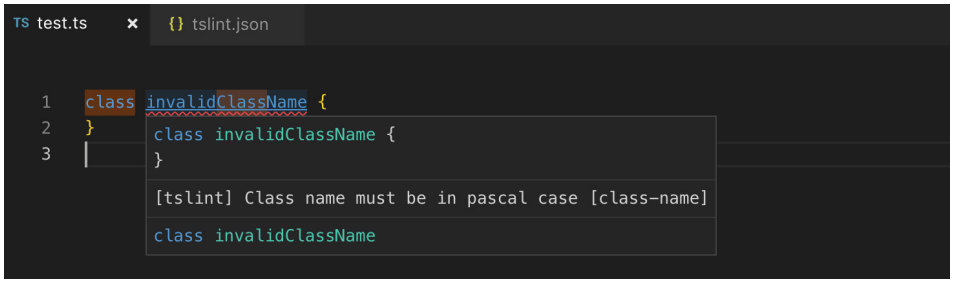
以上配置信息，这里具体参考了 https://palantir.github.io/tslint/usage/rule-flags/。

本地检查

在完成工程配置后，需要下载所需要依赖包，要在工程所在根目录使用 npm install 命令完成下载依赖包。

IDE 环境提示

在完成下载依赖包后，IDE 环境可以根据对应配置文件进行提示，可以实时地提示出存在问题代码的错误信息，以 VSCode 为例：



TSLint 规则示例

本地命令检查

VSCode 目前还有继续完善的空间，如果部分文件未在窗口打开的情况下，可能存在其中错误未提示出的情况，这时候，我们可以通过本地命令进行全工程的检查，在 React Native 工程的根目录下，通过以下命令行执行：

```
tslint --project tsconfig.json --config tslint.json
```

（此命令如果不正确运行，可在之前加入 ./node_modules/.bin/）即为：

```
./node_modules/.bin/tslint --project tsconfig.json --config tslint.json
```

从而会提示出类似以下错误的信息：

```
src/Components/test.ts[1, 7]: Class name must be in pascal case
```

在线 CI 检查

本地进行代码检查的过程也会存在被人遗忘的可能性，通过技术的保障，可以避免人为遗忘，作为代码提交的标准流程，通过 CI 检查后再合并代码，可以有效避免代码错误的问题。CI 系统可以为理解为一个云端的环境，环境配置与本地一致，在这种情况下，可以生成与本地一致的报告，在美团内部可以使用基于 Jenkins 的 Castle CI 系统，生成结果与本地结果一致：

TSLint 规则示例

其他方式

代码检查不止局限上述阶段，在代码 commit、pull request、打包等阶段均可触发。

- 代码 commit 阶段，通过 Hook 方式可以触发代码检查，可以有效地将在线 CI 检查阶段强制提前，基本保证了在线 CI 检查的完全正确性。
- 代码 pull request 阶段，通过在线 CI 检查可以触发代码检查，可以有效保证合入分支尤其是主分支的正确性。
- 代码打包阶段，通过在线 CI 检查可以触发代码检查，可以有效保证打包代码的正确性。

四、自定义编写 TSLint 规则

为什么要自定义 TSLint 规则

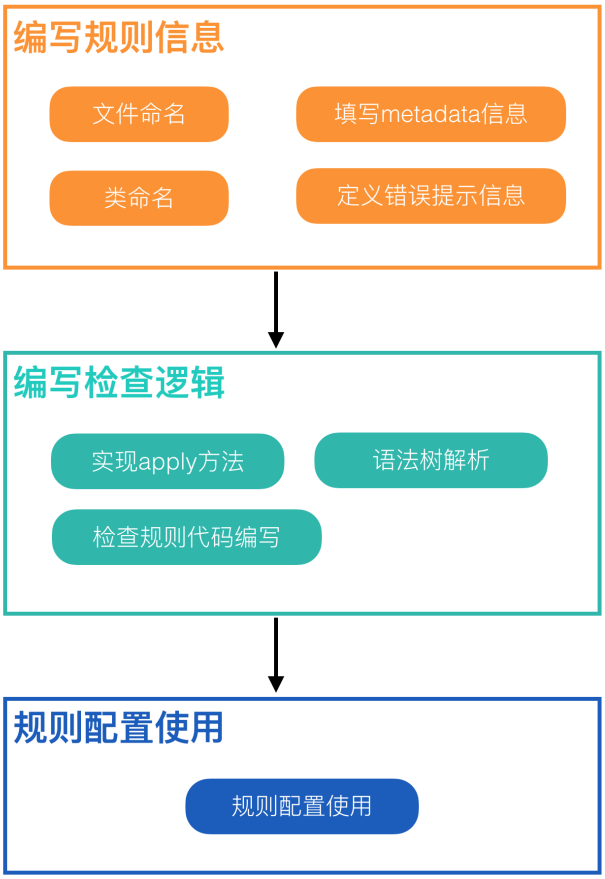
当前的 TSLint 规则虽然涵盖了比较普遍问题的一些代码检查，但是实践中还是存在一些问题的：

1. 团队中的个性化需求难以满足。例如，saga 中的异步函数需要在最外层加 try-catch，且 catch 块中需要加异常上报，这个明显在官方的 TSLint 规则无法实现，为此需要自定义的开发。
2. 官方规则的开启与配置不符合当前团队情况。

基于以上原因其他团队也有自定义 TSLint 的先例，例如上文提到的 tslint-microsoft-contrib、tslint-eslint-rules 等。

自定义规则步骤

那自定义 TSLint 大概需要什么步骤呢，首先规则文件根据规范进行按部就班的编写规则信息，然后根据代码检查逻辑对语法树进行分析并编写逻辑代码，这也是自定义规则的核心部分了，最后就是自定义规则的使用了。



TSLint 规则示例

自定义规则的示例直接参考官方的规则是最直接的，我们能这里参考一个比较简单的规则”class-name”。

“class-name”规则上文已经提到，它的意思是对类命名进行规范，当团队中类相关的命名不规范，会导致项目代码风格不统一甚至其他出现的问题，而”-

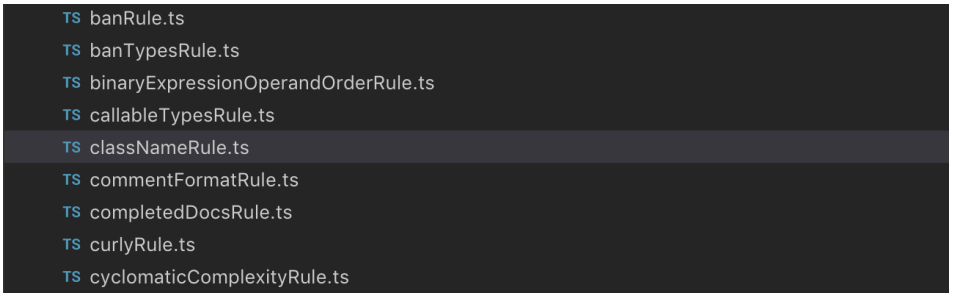
class-name”规则可以有效解决这个问题。我们可以看下具体的源码文件：https://github.com/palantir/tslint/blob/master/src/rules/classNameRule.ts。

然后将分步对此自定义规则进行讲解。

```
import { isClassLikeDeclaration, isInterfaceDeclaration } from "tsutils";
import * as ts from "typescript";

import * as Lint from "../index";
import { isUpperCase } from "../utils";

export class Rule extends Lint.Rules.AbstractRule {
    /* tslint:disable:object-literal-sort-keys */
    public static metadata: Lint.IRuleMetadata = {
        ruleName: "class-name",
        description: "Enforces PascalCased class and interface names.",
        rationale: "Makes it easy to differentiate classes from regular variables at a glance.",
        optionsDescription: "Not configurable.",
        options: null,
        optionExamples: [true],
        type: "style",
        typescriptOnly: false,
    };
    /* tslint:enable:object-literal-sort-keys */

    public static FAILURE_STRING = "Class name must be in pascal case";

    public apply(sourceFile: ts.SourceFile): Lint.RuleFailure[] {
        return this.applyWithFunction(sourceFile, walk);
    }
}

function walk(ctx: Lint.WalkContext<void>) {
    return ts.forEachChild(ctx.sourceFile, function cb(node: ts.Node): void {
        if (isClassLikeDeclaration(node) && node.name !== undefined ||
            isInterfaceDeclaration(node)) {
            if (!isPascalCased(node.name!.text)) {
                ctx.addFailureAtNode(node.name!, Rule.FAILURE_STRING);
            }
        }
        return ts.forEachChild(node, cb);
    });
}

function isPascalCased(name: string): boolean {
    return isUpperCase(name[0]) && !name.includes("_");
}
```

TSLint 规则示例

第一步，文件命名



TSLint 规则示例

规则命名必须是符合以下 2 个规则：

1. 驼峰命名。
2. 以 'Rule' 为后缀。

第二步，类命名

规则的类名是 Rule，并且要继承 Lint.Rules.AbstractRule 这个类型，当然也可能有继承 TypedRule 这个类的时候，但是我们通过阅读源码发现，其实它也是继承自 Lint.Rules.AbstractRule 这个类。

```
import { AbstractRule } from "./abstractRule";
import { ITypedRule, RuleFailure } from "./rule";

export abstract class TypedRule extends AbstractRule implements ITypedRule {

    public apply(): RuleFailure[] {
        // if no program is given to the linter, show an error
        showWarningOnce(`Warning: The '${this.ruleName}' rule requires type information.`);
        return [];
    }

    public abstract applyWithProgram(sourceFile: ts.SourceFile, program: ts.Program): RuleFailure[];
}
```

TSLint 规则示例

第三步，填写 metadata 信息

metadata 包含了配置参数，定义了规则的信息以及配置规则的定义。

- ruleName 是规则名，使用烤串命名法，一般是将类名转为烤串命名格式。
- description 一个简短的规则说明。
- descriptionDetails 详细的规则说明。
- rationale 理论基础。
- options 配置参数形式，如果没有可以配置为 null。
- optionExamples 参数范例 ，如没有参数无需配置。
- typescriptOnly true/false 是否只适用于 TypeScript。
- hasFix true/false 是否带有修复方式。
- requiresTypeInfo 是否需要类型信息。
- optionsDescrition options 的介绍。
- type 规则的类型。

规则类型有四种，分别为：”functionality”、”maintainability”、”style”、”type-script”。

- functionality ：针对于语句问题以及功能问题。
- maintainability：主要以代码简洁、可读、可维护为目标的规则。
- style：以维护代码风格基本统一的规则。
- typescript：针对于 TypeScript 进行提示。

第四步，定义错误提示信息

```
public static FAILURE_STRING = "Class name must be in pascal case";
```

TSLint 错误信息

这个主要是在检查出问题的时候进行提示的文字，并不局限于使用一个静态变量的形式，但是大部分官方规则都是这么编写，这里对此进行介绍，防止引起歧义。

第五步，实现 apply 方法

apply 主要是进行静态检查的核心方法，通过返回 applyWithFunction 方法或者返回 applyWithWalker 来进行代码检查，其实 applyWithFunction 方法与

applyWithWalker 方法的主要区别在于 applyWithWalker 可以通过 IWalker 实现一个自定义的 IWalker 类，区别如下：

```
public applyWithWalker(walker: IWalker): RuleFailure[] {
    walker.walk(walker.getSourceFile());
    return walker.getFailures();
}

public isEnabled(): boolean {
    return this.ruleSeverity !== "off";
}

protected applyWithFunction(sourceFile: ts.SourceFile, walkFn: (ctx: WalkContext<void>) => void): RuleFailure[];
protected applyWithFunction<T>(sourceFile: ts.SourceFile, walkFn: (ctx: WalkContext<T>) => void, options: NoInfer<T>): RuleFailure[];
protected applyWithFunction<T, U>(
    sourceFile: ts.SourceFile,
    walkFn: (ctx: WalkContext<T>, programOrChecker: U) => void,
    options: NoInfer<T>,
    checker: NoInfer<U>,
): RuleFailure[];
protected applyWithFunction<T, U>(
    sourceFile: ts.SourceFile,
    walkFn: (ctx: WalkContext<T | void>, programOrChecker?: U) => void,
    options?: T,
    programOrChecker?: U,
): RuleFailure[] {
    const ctx = new WalkContext(sourceFile, this.ruleName, options);
    walkFn(ctx, programOrChecker);
    return ctx.failures;
}
```

TSLint

其中实现 IWalker 的抽象类 AbstractWalker 里面也继承了 WalkContext，

```
import * as ts from "typescript";

import { RuleFailure } from "../rule/rule";
import { WalkContext } from "./walkContext";

export interface IWalker {
    getSourceFile(): ts.SourceFile;
    walk(sourceFile: ts.SourceFile): void;
    getFailures(): RuleFailure[];
}

export abstract class AbstractWalker<T> extends WalkContext<T> implements IWalker {
    public abstract walk(sourceFile: ts.SourceFile): void;

    public getSourceFile() {
        return this.sourceFile;
    }

    public getFailures() {
        return this.failures;
    }
}
```

TSLint

而这个 WalkContext 就是上面提到的 applyWithFunction 的内部实现类。

```
protected applyWithFunction(sourceFile: ts.SourceFile, walkFn: (ctx: WalkContext<void>) => void): RuleFailure[];
protected applyWithFunction<T>(sourceFile: ts.SourceFile, walkFn: (ctx: WalkContext<T>) => void, options: NoInfer<T>): RuleFailure[];
protected applyWithFunction<T, U>(
    sourceFile: ts.SourceFile,
    walkFn: (ctx: WalkContext<T>, programOrChecker: U) => void,
    options: NoInfer<T>,
    checker: NoInfer<U>,
): RuleFailure[];
protected applyWithFunction<T, U>(
    sourceFile: ts.SourceFile,
    walkFn: (ctx: WalkContext<T | void>, programOrChecker?: U) => void,
    options?: T,
    programOrChecker?: U,
): RuleFailure[] {
    const ctx = new WalkContext(sourceFile, this.ruleName, options);
    walkFn(ctx, programOrChecker);
    return ctx.failures;
}
```

TSLint

第六步，语法树解析

无论是 applyWithFunction 方法还是 applyWithWalker 方法中的 IWalker 实现都传入了 sourceFile 这个参数，这个相当于文件的根节点，然后通过 ts.forEach-Child 方法遍历整个语法树节点。

这里有两个查看 AST 语法树的工具：

- AST Explorer：https://astexplorer.net/

  对应源码：https://github.com/fkling/astexplorer
- TypeScript AST Viewer：https://ts-ast-viewer.com/

  对应源码：https://github.com/dsherret/ts-ast-viewer

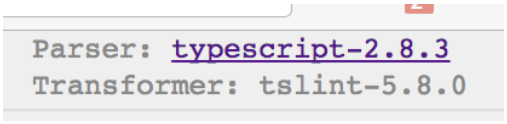
AST Explorer

优点：

在 AST Explorer 可以高亮显示所选中代码对应的 AST 语法树信息。

缺点：

1. 不能选择对应版本的解析器，导致显示的语法树代码版本固定。



TSLint

2. 语法树显示的信息相对较少。

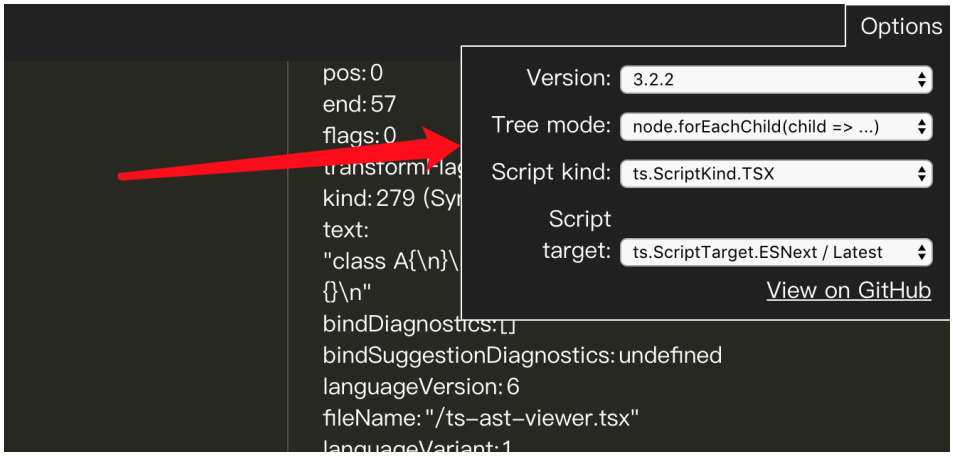
```
kind: 203
name: Identifier = $node {
   pos: 41
   end: 53
   flags: 0
   escapedText: "invalidName"
   text: "invalidName"
   kind: 71
   *leadingComments: undefined
   *trailingComments: undefined
}
typeParameters: undefined
```

TSLint

TypeScript AST Viewer

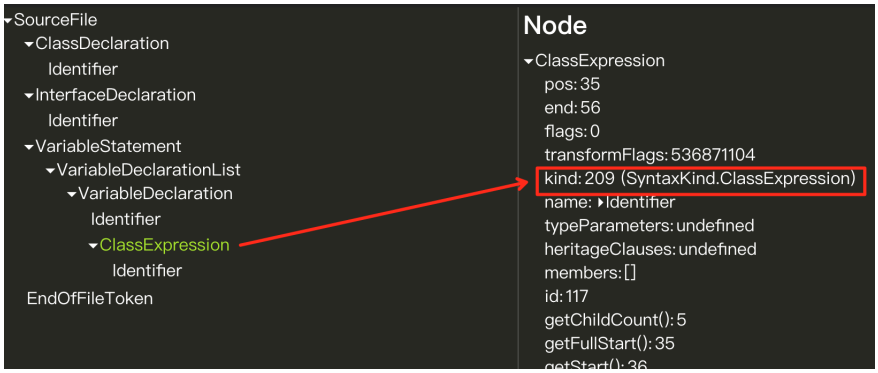
优点：

1. 解析器对应版本可以动态选择：



TSLint

2. 语法树显示的信息不仅显示对应的数字代码，还可为对应的实际信息：



TSLint

每个版本对应对 kind 信息数值可能会变动，但是对应的枚举名字是固定的，如下图：

typescript.d.ts

```
61    }
62    enum SyntaxKind {
63        Unknown = 0,
64        EndOfFileToken = 1,
65        SingleLineCommentTrivia = 2,
66        MultiLineCommentTrivia = 3,
67        NewLineTrivia = 4,
68        WhitespaceTrivia = 5,
69        ShebangTrivia = 6,
70        ConflictMarkerTrivia = 7,
71        NumericLiteral = 8,
72        StringLiteral = 9,
73        JsxText = 10,
74        JsxTextAllWhiteSpaces = 11,
75        RegularExpressionLiteral = 12,
76        NoSubstitutionTemplateLiteral = 13,
77        TemplateHead = 14,
78        TemplateMiddle = 15,
79        TemplateTail = 16,
80        OpenBraceToken = 17,
81        CloseBraceToken = 18,
82        OpenParenToken = 19,
83        CloseParenToken = 20,
84        OpenBracketToken = 21,
85        CloseBracketToken = 22,
```

TSLint

从而这个工具可以避免频繁根据其数值查找对应信息。

缺点：不能高亮显示代码对应的 AST 语法树区域，定位效率较低。

综上，通过同时使用上述两个工具定位分析，可以有效地提高分析效率。

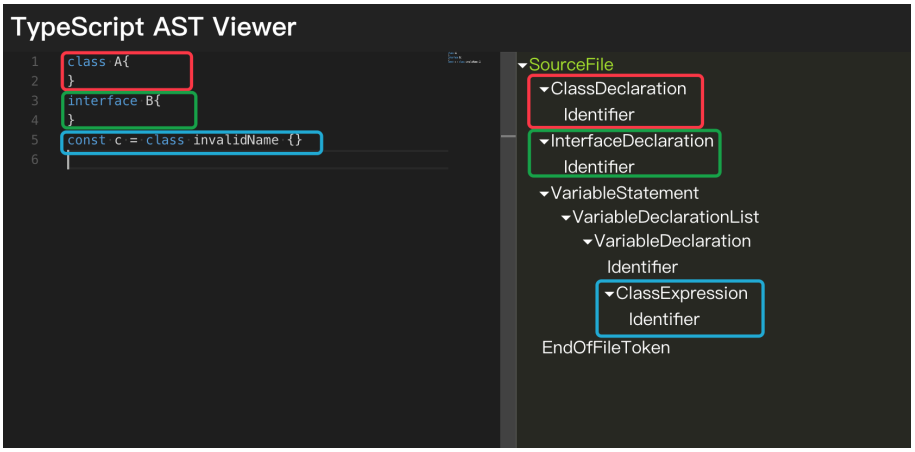
第七步，检查规则代码编写

通过 ts.forEachChild 方法对于语法树所有的节点进行遍历，在遍历的方法里可以实现自己的逻辑，其中节点的类为 ts.Node：

```
interface Node extends TextRange {
    kind: SyntaxKind;
    flags: NodeFlags;
    decorators?: NodeArray<Decorator>;
    modifiers?: ModifiersArray;
    parent?: Node;
}
```

TSLint

其中 kind 为当前节点的类型，当然 Node 是所有节点的基类，它的实现还包括 Statement、Expression、Declaration 等，回到开头这个 "class-name" 规则，我们的所有声明类主要是 class 与 interface 关键字，分别对应 ClassExpression、ClassDeclaration、InterfaceDeclaration，我们可以通过上步提到的 AST 语法树工具，在语法树中看到其为一一对应的。



TSLint

在规则代码中主要通过 isClassLikeDeclaration、isInterfaceDeclaration 这两个方法进行判断的。

```
function walk(ctx: Lint.WalkContext<void>) {
    return ts.forEachChild(ctx.sourceFile, function cb(node: ts.Node): void {
        if (isClassLikeDeclaration(node) && node.name !== undefined ||
            isInterfaceDeclaration(node)) {
            if (!isPascalCased(node.name!.text)) {
                ctx.addFailureAtNode(node.name!, Rule.FAILURE_STRING);
            }
        }
        return ts.forEachChild(node, cb);
    });
}
```

TSLint

其中 isClassLikeDeclaration、isInterfaceDeclaration 对应的方法我们可以在 node.js 文件中找到：

```
function isClassLikeDeclaration(node) {
    return node.kind === ts.SyntaxKind.ClassDeclaration ||
        node.kind === ts.SyntaxKind.ClassExpression;
}
exports.isClassLikeDeclaration = isClassLikeDeclaration;
```

TSLint

```
function isInterfaceDeclaration(node) {
    return node.kind === ts.SyntaxKind.InterfaceDeclaration;
}
exports.isInterfaceDeclaration = isInterfaceDeclaration;
```

TSLint

判断是对应的类型时，调用 addFailureAtNode 方法把错误信息和节点传入，当然还可以调用 addFailureAt、addFailure 方法。

```
export class WalkContext<T> {
    public readonly failures: RuleFailure[] = [];

    constructor(public readonly sourceFile: ts.SourceFile, public readonly ruleName: string, public readonly options: T) {}

    /** Add a failure with any arbitrary span. Prefer `addFailureAtNode` if possible. */
    public addFailureAt(start: number, width: number, failure: string, fix?: Fix) {
        this.addFailure(start, start + width, failure, fix);
    }

    public addFailure(start: number, end: number, failure: string, fix?: Fix) {
        const fileLength = this.sourceFile.end;
        this.failures.push(
            new RuleFailure(this.sourceFile, Math.min(start, fileLength), Math.min(end, fileLength), failure, this.ruleName, fix),
        );
    }

    /** Add a failure using a node's span. */
    public addFailureAtNode(node: ts.Node, failure: string, fix?: Fix) {
        this.addFailure(node.getStart(this.sourceFile), node.getEnd(), failure, fix);
    }
```

TSLint

最终这个规则编写结束了，有一点再次强调下，因为每个版本所对应的类型代码可能不相同，当判断 kind 的时候，一定不要直接使用各个类型对应的数字。

第八步，规则配置使用

完成规则代码后，是 ts 后缀的文件，而 ts 规则文件实际还是要用 js 文件，这时候我们需要用命令将 ts 转化为 js 文件：

```
tsc ./src/*.ts --outDir dist
```

将 ts 规则生成到 dist 文件夹（这个文件夹命名用户自定），然后在 tslint.json 文件中配置生成的规则文件即可。

```
{} tslint.json ×
1  {
2    "rulesDirectory": "./dist",
3    "rules": {
4      "check-if-number-zero": true
5    }
6  }
7
```

TSLint

之后在项目的根目录里面，使用以下命令既可进行检查：

```
tslint --project tsconfig.json --config tslint.json
```

同时为了未来新增规则以及规则配置的更好的操作性，建议可以封装到自己的规则包，以便与规则的管理与传播。

总结

TSLint 的优点：

1. 速度快。相对于动态代码检查，检查速度较快，现有项目无论是在本地检查，还是在 CI 检查，对于由十余个页面组成的 React Native 工程，可以在 1 到 2 分钟内完成；
2. 灵活。通过配置规则，可以有效地避免常见代码错误与潜在的 Bug；
3. 易扩展。通过编写配置自定义规则，可以及时准确快速查找出代码中特定风险点。

TSLint 缺点：

1. 规则的结果只有对与错两种等级结果，没有警告等级的的提示结果；
2. 无法直接报告规则报错数量，只能依赖其他手段统计；
3. TSLint 规则针对于当前单一文件可以有效地通过语法树进行分析判定，但对于引用到的其他文件中的变量、类、方法等，则难以通过 AST 语法树进行判定。

使用结果及分析

在美团，有十余个页面的单个工程首次接入 TSLint 后，检查出的问题有近百条。但是由于开启的规则不同，配置规则包的差异，检查后的数量可能为几十条到几千条甚至更多。现在已开发十余条自定义规则，在单个工程内，处理优化了数百处可能存在问题的代码。最终 TSLint 接入了相关 React Native 开发团队，成为了代码提交阶段的必要步骤。

通过团队内部的验证，文章开头遇到的问题得到了有效地缓解，目标基本达到预期。TSLint 在 React Native 开发过程中既保证了代码风格的统一，又保证了 React

Native 开发人员的开发质量，避免了许多低级错误，有效地节省了问题排查和人员沟通的成本。

同时利用自定义规则，能够将一些兼容性问题在内的个性化问题进行总结与预防，提高了开发效率，不用花费大量时间查找问题代码，又避免了在一个问题上跌倒多次的情况出现。对于不同经验的开发者而言，不仅可以进行友好的提示，也可以帮助快速地定位问题，将一个人遇到的经验教训，用极低的成本扩散到其他团队之中，将开发状态从“亡羊补牢”进化到“防患未然”。

作者简介

家正，美团点评 Android 高级工程师。2017 年加入美团点评，负责美团大交通的业务开发。

ESLint 在中大型团队的应用实践

宋鹏

引言

代码规范是软件开发领域经久不衰的话题，几乎所有工程师在开发过程中都会遇到，并或多或少会思考过这一问题。随着前端应用的大型化和复杂化，越来越多的前端工程师和团队开始重视 JavaScript 代码规范。得益于前端开源社区的繁盛，当下已经有几种较为成熟的 JavaScript 代码规范检查工具，包括 JSLint、JSHint、ESLint、FECS 等等。本文主要介绍目前较为通用的方案——ESLint，它是一款插件化的 JavaScript 代码静态检查工具，其核心是通过对代码解析得到的 AST（Abstract Syntax Tree，抽象语法树）进行模式匹配，定位不符合约定规范的代码。

ESLint 的使用并不复杂。依照 ESLint 的文档安装相关依赖，可以根据个人 / 团队的代码风格进行配置，即可通过命令行工具或借助编辑器集成的 ESLint 功能对工程代码进行静态检查，发现和修复不符合规范的代码。如果想降低配置成本，也可以直接使用开源配置方案，例如 eslint-config-airbnb 或 eslint-config-standard。

对于独立开发者，或者执行力较强、技术场景较为单一的小型团队而言，直接使用 ESLint 及其生态提供的一些标准方案，可以用较低成本来实现 JavaScript 代码规范的落地。如果再搭配一些辅助工具（例如 **husky** 和 **lint-staged**），整个流程会更加顺畅。但对于数十人的大型前端团队来说，面向数百个前端工程，规模化地应用统一的 JavaScript 代码规范，问题就会变得较为复杂。如果直接利用现有的开源配置方案，可能会使工作事倍功半。

问题分析

规模化应用统一的 ESLint 代码规范，会涌现各类问题，根源在于大型团队和小团队（或独立开发者）的差异性：

技术层面上：

- **技术场景更加广泛：**对于大型团队，其开发场景一般不会局限在传统 Web 领域内，往往还会涉及 Node.js、React Native、小程序、桌面应用（例如 Electron）等更广泛的技术场景。
- **技术选型更加分散：**团队内工程技术选型往往并不统一，如 React/Vue、JavaScript/TypeScript 等。
- **工程数量的增加和工程方案离散化导致 ESLint 方案的复杂度提升：**这样会进一步增加工程接入成本、升级成本和方案维护成本。

在团队层面，随着人员的增加和组织结构的复杂化：

- 人员风格差异性更大、沟通协调成本更高。
- 方案宣导更难触达，难以保证规范执行的落实。
- 执行状况和效果难以统计和分析。

因为存在诸多差异，我们在设计具体方案时，需要考虑和解决更多问题，以保证规范的落实。针对上述分析，我们梳理了以下需要解决的问题：

- 如何制定统一的代码规范和对应的 ESLint 配置?
  - **场景支撑：**如何实现对场景差异的支持？如何保证不同场景间一致部分（例如 JavaScript 基础语法）的规范一致性?
  - **技术选型支撑：**如何在支撑不同技术选型的前提下，保证基础规则（例如缩进）的一致性?
  - **可维护性：**具体到规则配置上，能否提升可复用性？在方案升级迭代时成本是否可控?
- 如何保证代码规范的执行?
  - 人员的增加和组织结构的复杂化，会导致基于管理的执行把控失效，这种情况应该如何保证代码规范的执行质量?
- 如何降低应用成本?
  - 在工程数量增加、工程方案离散化的情况，降低方案的接入、升级和执行

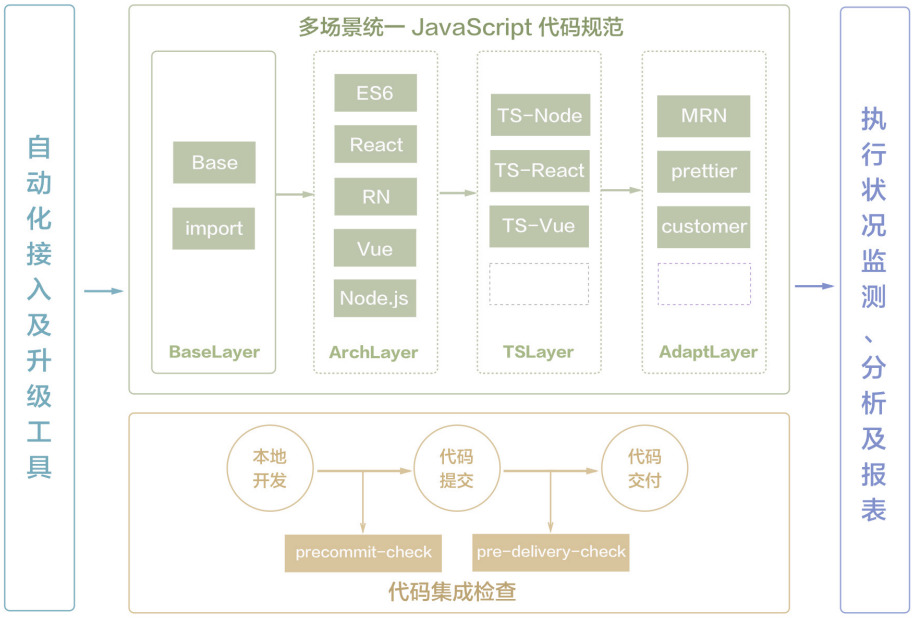
成本能节约大量的人力，同时也有利于方案落地推进。

- 如何及时了解规范应用状况和效果?

解决方案

为了能在团队内实现 JavaScript 代码规范的统一，在分析和思考团队规模化应用存在的问题后，我们设计了一套完整的技术解决方案。该方案包括多场景统一的 ESLint 规则配置、代码集成交付检查、自动化接入工具、执行状况监测分析等四个模块。通过各个模块协调配合，共同解决上文提出的问题，在降低维护成本、提升执行效率的同时，也保障了代码规范的统一。

整体方案的设计如下图所示：



1. **多场景统一的 JavaScript 规范：**该模块是整个方案的核心，借助 ESLint 的特性，通过分层分类的结构设计，在保证基础规则一致性的同时，实现了对不同场景、技术选型的支撑。
2. **代码集成交付检查：**该模块是方案落地执行的保障，将代码静态检查集成到持续交付工作流中。具体设计实现上，在保证交付质量的同时，也通过定制集

成检查工具降低了开发者的应用执行成本。

3. **自动化接入和升级方案：**通过命令行工具提供“一键”接入 / 升级能力，同时集成到团队脚手架中，大大降低了工程接入和维护的成本。
4. **执行状况监测分析：**通过对工具运行和代码集成交付检查过程进行埋点、检查结果收集和分析，了解方案的应用状态和效果。

方案实现

上文中提出的问题，通过各模块的协调配合能够得到有效地解决，但具体到各个模块的实现，仍然需要进一步深入思考，以设计出更加合理的实现方案。本章将对方案的四个核心模块进行详细介绍。

通用 ESLint 配置方案

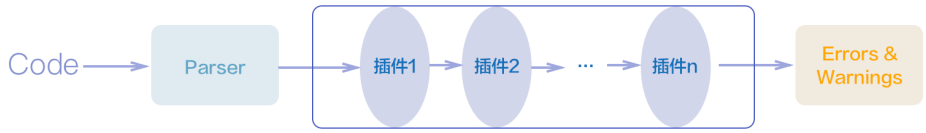
这一模块主要借助 ESLint 的基础特性，采用分层分类的结构设计，提供多场景、多技术方案的通用配置方案，并使方案具备易维护、易扩展的特性。

ESLint 特性简介

在进行 ESLint 配置方案设计前，我们先看一下 ESLint 的一些特点。

1. 插件化

下图简单地描述了 ESLint 的工作过程：



ESLint 的能力更像一个引擎，通过提供的基础检测能力和模式约束，推动代码检测流程的运转。原始代码经过解析器的解析，在管道中逐一经过所有规则的检查，最终检测出所有不符合规范的代码，并输出为报告。借助插件化的设计，不但可以对所有的规则进行独立的控制，还可以定制和引入新的规则。ESLint 本身并未和解析器强绑定，我们可以使用不同的解析器进行原始代码解析，例如可以使用

babel-eslint 支持更新版本、不同阶段的 ES 语法，支持 JSX 等特殊语法，甚至可以借助 **@typescript-eslint/parser** 支持 **TypeScript** 语言的检查。

2. 配置能力全面、可层叠、可共享

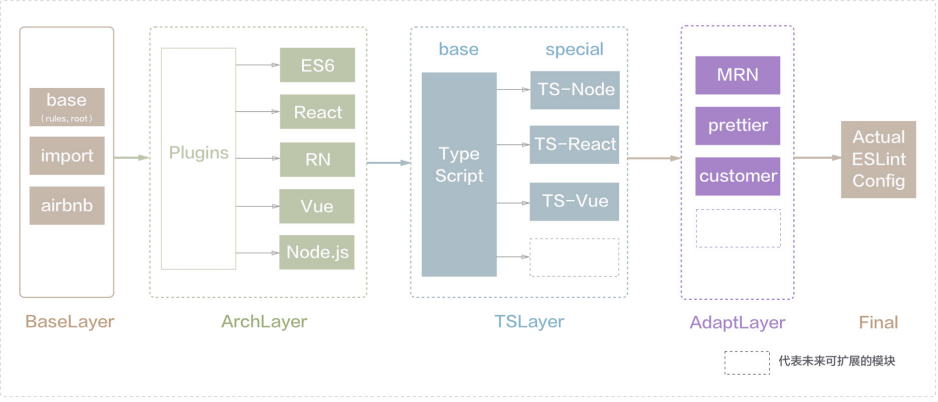
ESLint 提供了全面、灵活的配置能力，可以对解析器、规则、环境、全局变量等进行配置；可以快速引入另一份配置，和当前配置层叠组合为新的配置；还可以将配置好的规则集发布为 npm 包，在工程内快速应用。

3. 社区生态较为成熟

开源社区中基于 ESLint 的项目非常多，既有针对各种场景、框架的插件，也有各种 ESLint 规则配置方案，基本可以涵盖前端开发的所有场景。

规范配置方案设计

基于 ESLint 的插件化、可层叠配置特性，以及面向各种场景、框架的开源方案，我们设计了如下图所示的 ESLint 配置架构：



该配置架构采用了分层、分类的结构，其中：

- **基础层：**制定统一的基础语法和格式规范，提供通用的代码风格和语法规则配置，例如缩进、尾逗号等等。
- **框架支撑层（可选）：**提供对通用的一些技术场景、框架的支持，包括 Node.js、React、Vue、React Native 等；这一层借助开源社区的各种插件进行配

置，并对各种框架的规则都进行了一定的调整。

- **TypeScript 层（可选）：**这一层借助 typescript-eslint，提供对 TypeScript 的支持。
- **适配层（可选）：**提供对特殊场景的定制化支持，例如 MRN（美团内部的 React Native 定制化方案）、配合 prettier 使用、或者某些团队的特殊规则诉求。

具体的实际项目中，可以灵活的选择各层级、各类型的搭配，获得和项目匹配的 ESLint 规则集。例如，对于使用 TypeScript 语言的 React 项目，可以将基础层、框架层的 React 分支、以及 TypeScript 支撑层的 React 分支层叠到一起，最终形成适用于该项目的 ESLint 配置。如果项目不再使用 TypeScript 语言，只需要将 ts-react 这一层去掉即可。

最终，形成了如下所示的 ESLint 配置集：

考虑到维护、升级和应用成本，我们最终选择将所有配置放到一个 npm 包中，而不是每种类型分别设置。仍以使用 TypeScript 语言的 React 项目为例，只需在工程中进行如下配置：

```
// 需要安装 typescript、eslint-plugin-react、@typescript-eslint 等插件
module.exports = {
  root: true,
  extends: [
    // 因为基础层是必备的，所以框架层默认引入了对应的基础层，不需再单独引入 eslintrc.
base.js
    'eslint-config-xxx/eslintrc.react.js',
    'eslint-config-xxx/eslintrc.typescript-react.js'
  ]
}
```

这种通过分层、分类的结构设计，还有利于后期的维护：

- 对基础层的修改，只需修改一处即会全局生效。
- 对非基础层某一部分的调整不会产生关联性的影响。
- 如需扩展对某一类型的支持，只需关注这一类型的特殊规则配置。

众所周知，TypeScript 类型的项目使用 TSLint 进行代码检查，也是一种简单、便捷的方案。但在本方案中我们依旧选择了：eslint + @typescript-eslint/parser + @typescript-eslint/eslint-plugin 的组合方案。主要有以下几点原因：

- ESLint 的规则配置更加详细全面，覆盖更加广泛。
- 采用了分层分类的架构，能够保证即使框架或语言不同，也能在基本语法、风格层面保持规则的一致性，这样有利于团队内不同技术选型项目的风格统一。
- @typescript-eslint 方案持续迭代，问题响应非常迅速，对 TSLint 相关的规则基本提供了对等的实现。

根据最新消息，TypeScript 在 2019 路线图 中明确表明后续对 Lint 工具的支持和建设会以对 ESLint 进行适配的方式为主。

代码集成检查

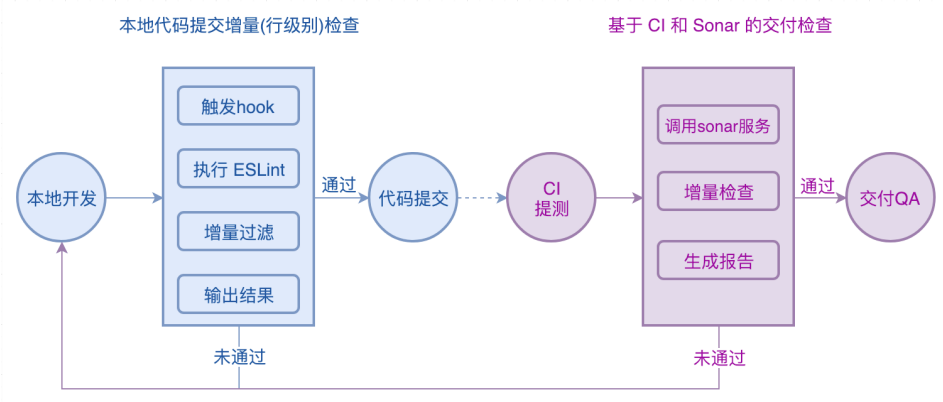
基于团队对工程化基础设施的建设，将代码规范静态检查与开发工作流集成，保证代码规范的落实。

通常而言，工程接入 ESLint 后，可以在开发的同时借助编辑器集成的 ESLint 检查提示能力（例如 VSCode 的 ESLint 插件），实时发现和修改不符合规范的语法错误和风格问题。但这仍不能避免因一些主观因素或疏漏造成的规范执行不到位，所以我们考虑在开发工作流的特定节点自动执行代码静态检查，阻断不合规范代码的提交或交付。

集成静态检查的开发工作流节点有很多，我们主要参考以下两种方案：

- **代码提交检查：**在代码 Commit 时，通过 githook 触发 ESLint 检查。其优点在于能实时响应开发者的动作，给出反馈，快速定位和修复问题；缺陷在于开发者可以主动跳过检查。
- **代码交付检查：**在代码交付（借助 CI 系统的交付流程功能）时，在代码检测平台中对代码进行 ESLint 检查，检测不通过则阻断交付。其优点在于能够强制执行，可在线追踪检测报告；缺陷在于离开发者的开发环境太“远”，开发者响应处理成本较高。

如果将两者进行结合，可能会事半功倍，效果如下图所示：



常用的代码提交检查方法一般是 husky 与 lint-staged 结合，在代码 Commit 时，通过 githook 触发对 git 暂存区文件的检查。但考虑到团队现有工程数量庞大、存在大量行数较多的文件，虽然 lint-staged 策略能够降低部分成本，但仍稍显不足。为此，我们对该方法进行优化，定制了本地代码提交检查工具 precommit-eslint，其核心特点是：

- 将增量检查执行到代码行这一粒度，支持 Warn 和 Error 两个检查级别。
- 只需将工具安装为工程的依赖，无需任何配置。
- 减少了 pre-commit hook 中植入脚本的侵入性。
- 进行了执行状况埋点和采集。

使用效果如下图所示：

```
→  waimai_mfe_ems git:(feature/WAIMAISP-8833) ✗ git commit
**start eslint**


/Users/songpeng/mtcode/mfe/contract/waimai_mfe_ems/src/pages/taskList/constants.ts
  10:1   error  Missing return type on function  @typescript-eslint/explicit-function-return-type
  39:70  error  Missing semicolon                semi

✖ 2 problems (2 errors, 0 warnings)


eslint check info collected.

**eslint failed**
```

在美团，我们使用自主开发的 CI 系统，并在独立部署的 Sonar 系统上定制化实现了相应规则，基本可以满足诉求，这里就不再赘述。对于独立的团队，基于 ESLint 提供的工具，可以很容易的实现使用 Node 快速搭建一个代码检测服务或平台，大家有兴趣不妨一试。

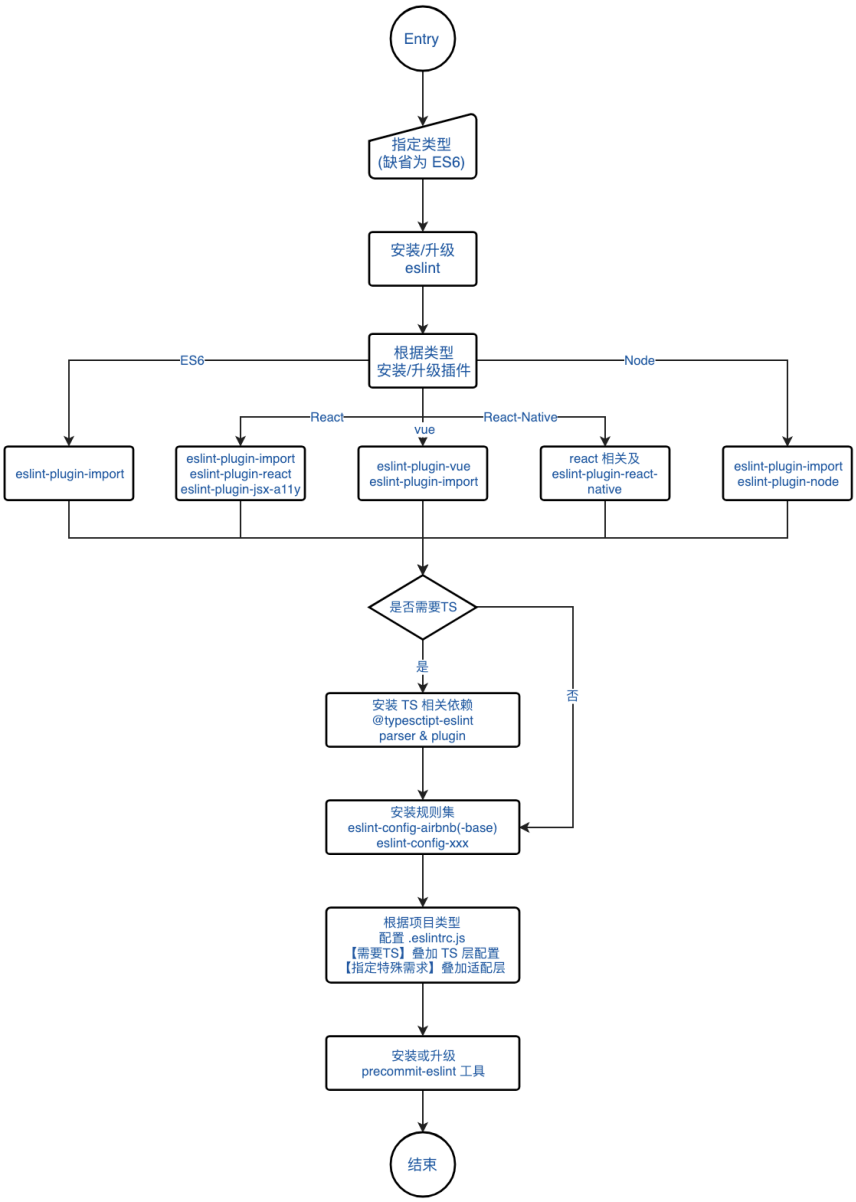
自动化接入工具

这个模块主要通过 CLI 工具提供方案自动化接入的能力，降低工程接入和升级的成本。如果不借助自动化工具，在工程中接入上述方案还是有一定的工作量和复杂度的，大致步骤如下：

1. 安装 Eslint。
2. 根据项目类型安装对应的 ESLint 规则配置 npm 包。
3. 根据项目类型安装相关的插件、解析器等。
4. 根据项目类型配置 .eslintrc 文件。
5. 安装代码提交检查工具。
6. 配置 package.json。
7. 测试及修复问题。

在这个过程中，特别需要注意依赖的版本问题：依赖之间的版本兼容性，例如 typescript 和 @typescript-eslint/parser 之间的兼容性；依赖对规则的支持性，比如某个版本的插件中去除了对某个规则的支持，但规则配置中仍然配置了该规则，此时配置就会失效。对于 ESLint 不熟悉的开发者而言，在配置的过程中都会或多或少遇到兼容性、解析异常、规则无效等问题，反复排查和定位问题会浪费大量的精力。

因此，在设计开发自动化接入工具时，我们综合考虑了操作步骤、依赖版本、规则集和工程方案的兼容性，设计了如下的工作流程：

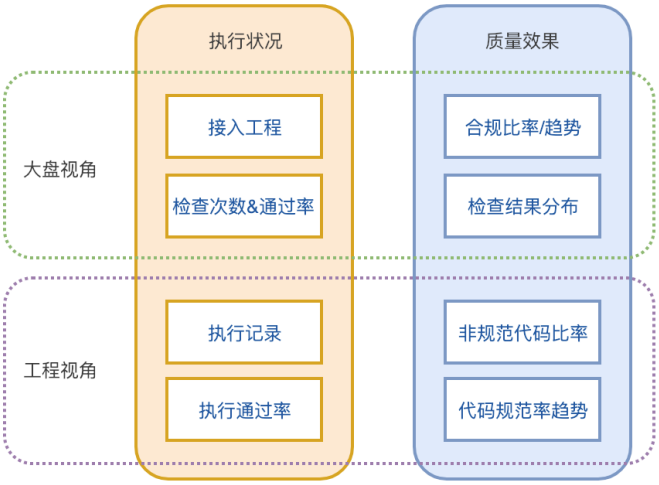


该工具流程简单，不管什么开发场景和框架选型，繁琐的接入流程都可以简化为一条命令，需要配合工程方案升级时同样如此。如下图所示，执行该命令后项目就完成了 ESLint 的接入，使用统一的规则规范编码，同是在代码提交时自动进行增量检查：

```
➜ test git:(master) ✗ eslint-init --react --ts --neverAutoInstall
正在安装 eslint 相关依赖 ...
eslint@5.12.1
babel-eslint@10.0.1
eslint-plugin-import@2.15.0
eslint-plugin-react@7.12.4
eslint-plugin-jsx-a11y@6.2.0
eslint-config-airbnb@17.1.0
@typescript-eslint/parser@1.1.1
@typescript-eslint/eslint-plugin@1.1.1
eslint 依赖安装完成
正在配置 eslint...
正在安装 eslint 配置集
eslint 配置集安装完成
检测到该项目尚无 eslintrc.js 配置文件
复制标准 eslintrc.js 配置模板到项目空间...
eslint配置完成
如果该项目中已经存在 eslintrc.js 之外的其他eslint配置文件，可以删除~
默认使用工程目录下的 .tsconfig.json 文件帮助 typescript 的检验
如配置文件非该路径，请自行配置：https://github.com/typescript-eslint/typescript-esli
nt/tree/master/packages/parser#configuration。
eslint 配置完成
正在设置持续集成检查方案
开始配置package.json...
持续集成检查方案配置成功
eslint初始化完成，happy coding~
```

埋点与统计分析

统计分析的主要目的是掌握方案应用执行状况和效果，理论上应当支持工程和大盘两个视角，如下图所示：

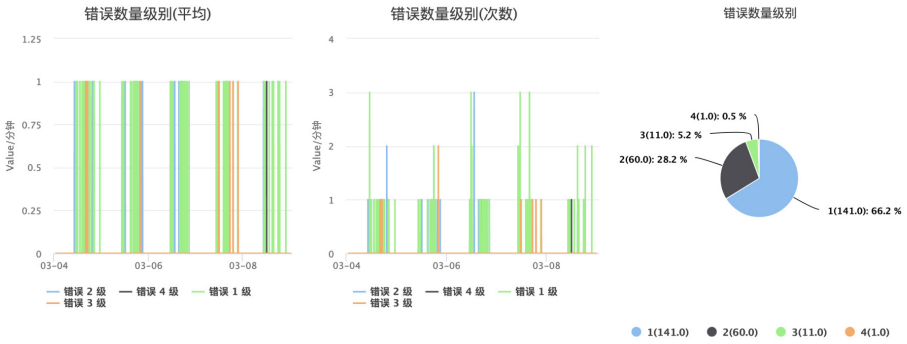


执行情况分析其实并不复杂，核心是信息采集和分析。在本方案中，信息采集通过 precommit-eslint 工具实现：在 git commit 触发本地代码检查后，脚本会把检查结果（包括检查是否通过、错误或警告信息的数量级别等）上报；信息的统计分析借助日志上报分析平台实现，美团使用的是 CAT 平台（如果团队或公司没有专门的平台，可以在上文提到的代码检测服务平台中实现这部分功能）。为了便于数据的聚合分析，我们将一次代码提交检查中出现的问题数量进行了分级：

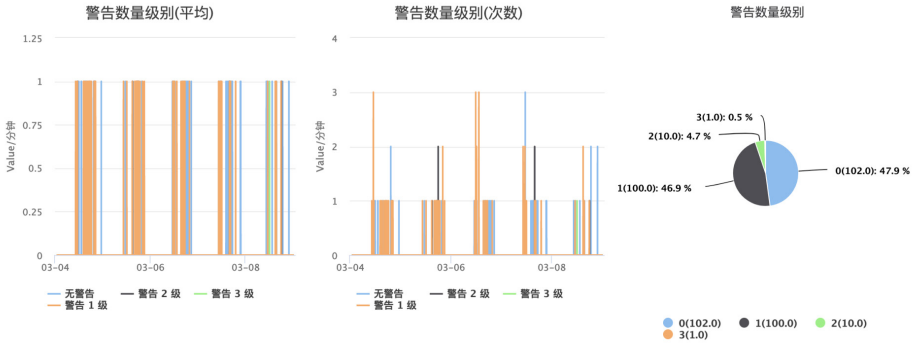
- 检查通过：检查无代码规范错误。
- 错误 1 级：检查出代码规范错误数量小于 10 个。
- 错误 2 级：检查出代码规范错误数量在 10 – 100 个之间。
- 错误 3 级：检查出代码规范错误数量在 100 – 1000 个之间。
- 错误 4 级：检查出代码规范错误数量大于 1000 个。

比如下图中，**201903 第一周**的代码提交检查结果统计（综合采样率 0.2），很明显，所有检查失败的提交中，错误数量在 10 个以内的占比最大，修复成本不高。

1. 提交检查异常分布（仅筛选检查未通过信息）



2. 提交检查警告信息分析



除此之外，还可以对单一工程，在更细的时间粒度上去观察提交检查的执行情况。

效果质量主要分析工程质量的变化：一方面可以通过代码检查执行通过率变化趋势、检查结果分布去看持续的生产流程中，代码质量是否有所提升；另一方面，由于代码检查采用增量模式，需要对工程代码进行整体分析，得到工程整体的不规范代码占比及变化趋势，从而从工程维度分析判断质量效果（涉及到权限相关问题，目前团队中未采用工程分析的方法）。具体的分析会在方案应用效果中一并进行介绍。

方案应用

除了上述整体方案外，为保证开发者使用更方便，我们还进行了一些配套工作：

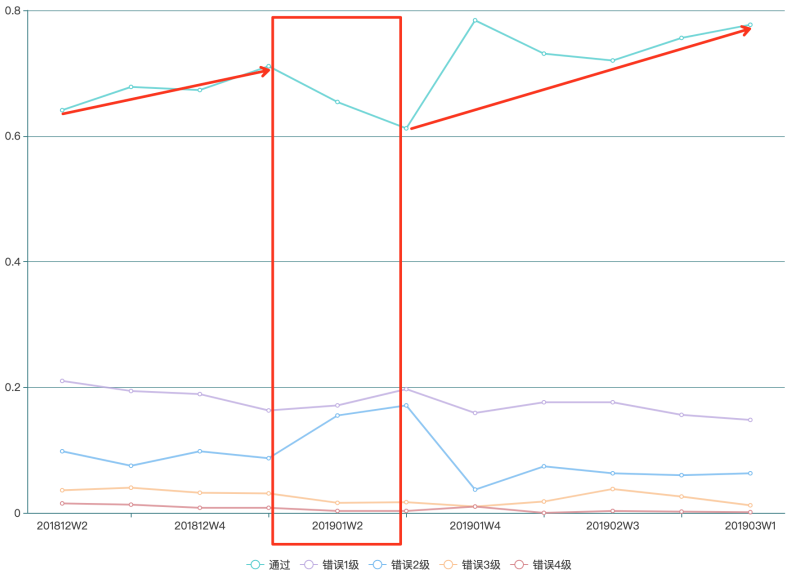
- 持续维护升级：以每月一版的方式持续迭代升级，解决应用中的问题、规则争议，以及支持新的规则或方案。
- 工程化集成：整套方案可以无缝接入到各个团队的脚手架工具中，自动成为团队的默认方案，在工程初始化阶段即可完成接入。
- 官网建设：提供详细的使用文档，包括规则信息、接入方法，并且对每个版本提供规则、环境依赖、changeLog 等详细说明。
- 常见使用问题：更新维护 FAQ，帮助后续接入者快速查找并解决问题。

目前，该套方案已经接入美团外卖、餐饮平台、闪购、榛果、金融等多个团队，基于埋点统计分析，我们（基于 2019 年 2 月份最后一周统计数据分析，综合采样率

0.2）得到了如下数据：

- 截止到 2019 年 2 月底，该方案已接入超过 200 个前端工程。
- 集成检查（增量）每天执行接近 1000 次。
- 集成检查（增量）平均每天检查出错误约 20000–25000 处。
- 集成检查代码质量：平均通过率为 75.562%，错误 1 级的比率为 15.644%，在所有未通过检查中占比 64.015%。

同时，我们持续统计上述数据的变化趋势，跟踪代码质量提升效果，以 2018 年 12 月到 2019 年 3 月的数据为例（截止 2019 年 3 月第一周，以周为时间统计尺度）：



从图中可以看出，最近三个月检查通过率整体呈上升趋势，但 2019 年 1 月的第 2 周和第 3 周集成检查通过率有明显下降。分析项目信息发现，在 2019 年 1 月的第 2 周有一批新项目接入，代码检查规范检查出几十个错误。但整体来看，目前集成检查通过率基本稳定在 75% – 80%，从变化趋势看仍有上升空间。

方案实施之后，我们做了一个用户调研，发现整体方案的运营正在发挥着正向的作用。一方面，在一定程度上提升了多人协作的效率，无论是共同维护一个工程还是

在多个工程间切换，避免了代码风格不一致带来的可读性成本和格式化风险；另一方面，会帮助大家发现和避免一些简单的语法错误。

规划和思考

该方案已经稳定应用，除了现有功能，我们还在思考是否可以更进一步的优化，提供更丰富的能力。由此规划了一些仍未落地的方向：

1. 扩展支持 HTML 和 CSS 的代码风格检查：虽然近几年前端框架、组件库的建设一定程度上减少了业务开发中（尤其是中后台业务）对 HTML 和 CSS 的需求，但是规范 HTML 和 CSS 的代码风格仍是必要的。基于此，可以用同样的思路将 HTML 和 CSS 的代码静态检查方案集成到当前的方案中，不再局限于 JavaScript（或 TypeScript）。
2. 进一步的封装：目前整体方案会将所有依赖和配置暴露在工程内，如果将其完全封装在一个工具内会更便于应用，但难点在于兼顾灵活性、对编辑器的支持等问题。
3. 增加工程维度的代码质量趋势分析：目前代码检查策略是增量检查，可以对接入的工程定期全量检查，基于时间线分析工程的代码质量变化趋势。
4. 进一步深入分析检查结果和统计数据，发现一些潜在问题，为推动开发质量提升提供辅助，如：
    - 统计开发者在工程中关闭或调整的规则，分析占比较高的规则被关闭的原因，进而调整规则或推动规则的执行。
    - 统计分布检查出错误的规则分布，梳理出最常出问题的代码规则，发布对应的最佳实践或手册。

以上是美团外卖团队在 ESLint 方案规模化应用过程中的一些实践，欢迎大家提出建议，一起沟通交流。

作者简介

宋鹏，美团外卖事业部终端研发工程师。

团队介绍

美团外卖事业部终端团队，负责的多个终端和平台直接连接亿万用户、数百万商家和几万名运营与销售，目标是在保障业务高稳定、高可用的同时，持续提升用户体验和研发效率。

在用户方向上，构建了全链路的高可用体系，客户端、Web 前端和小程序等多终端的可用性在 99% 左右；跨多端高复用的局部动态化框架在首页、广告、营销等核心路径的落地，提升了 30% 的研发效率；

在商家方向上，从提高进程优先级、VoIP Push 拉活、doze 等方面进行保活定制，并提供了 Shark、短链和 Push 等多条触达通道，订单到达率提升至 98% 以上；

在运营方向上，通过标准化研发流程、建设组件库和 Node 服务以及前端应用的管理与页面配置等提升 10% 的研发效率。

团队有多个岗位正在招聘，欢迎加入我们，联系邮箱 tech@meituan.com ，注明“外卖终端团队”。

App流程管理及实践

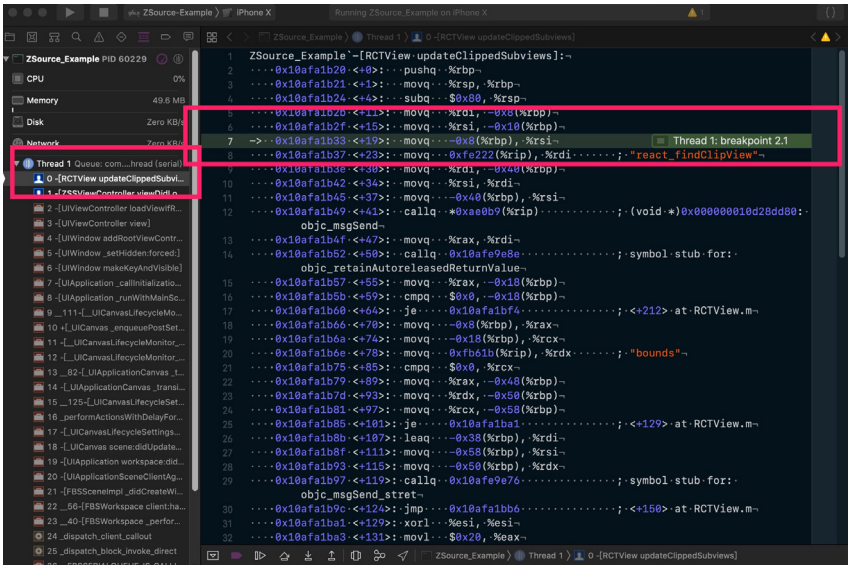
美团 iOS 工程 zsource 命令背后的那些事儿

宇杰

zsource 命令是什么?

美团 App 在 2015 年就已经基于 CocoaPods 完成了组件化的工作。在组件化的改造过程中，为了能够加速整体工程的构建速度，我们对需要集成进美团 App 的组件进行了二进制化，同时提供一个叫做 cocoapods-binary 的 CocoaPods 插件来支持本地工程使用二进制。因此，美团 App 的开发者在集成开发时，除了自己正在开发的组件，其他的组件都以二进制的形式存在。

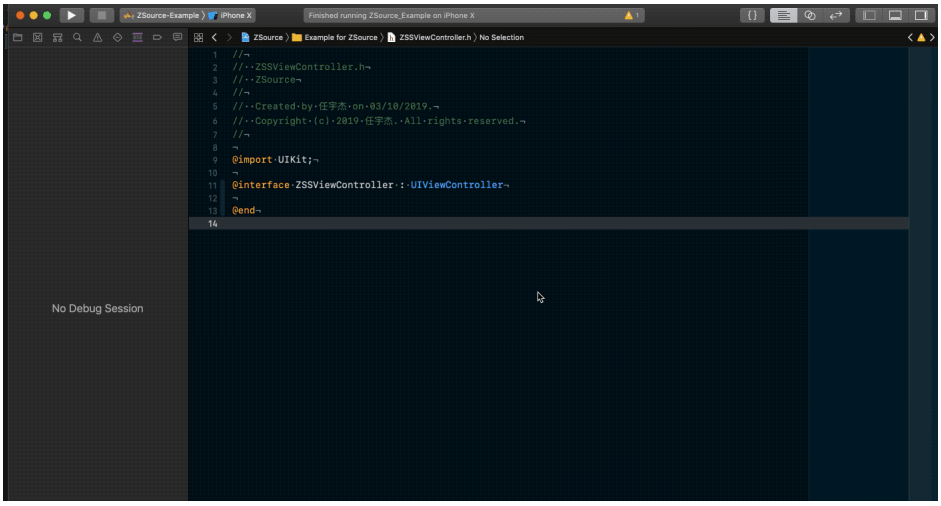
使用二进制，虽然会给工程带来构建速度的提升，但是会带来一个新的问题：在调试工程时，那些使用二进制的组件，无法像源码调试那样看到足够丰富的调试信息。例如，如果程序在二进制组件的代码中崩溃，我们只能看到该组件的堆栈信息和一些不明所以的汇编代码：



程序断点在二进制组件的代码中时的样子

和业界大多的组件化方案类似，美团 App 的组件化方案也提供了将一个组件从二进制切换到源码的机制。美团工程的开发者能够使用一系列配置和命令来切换组件的源码和二进制状态，但每次切换都需要重新执行 pod install。这种方式在组件化的初期是没有什么问题的。但随着美团 App 的组件数量不断增长，即便是只切换一个组件的状态，单次 pod install 的时间也增长到了分钟级。而且这种方式每切换一次就必须重新编译运行一次 App，在追查一些偶现崩溃问题时，开发体验非常不友好，也不利于崩溃问题的快速定位分析。

为了解决以上提到的这些问题，我们利用 CocoaPods 的插件机制，为 CocoaPods 的 pod 命令增加了 zsource 子命令，开发者可以在使用二进制构建工程的同时，非常快速地将一个组件调出源码进行调试，具体的使用效果可以看一下如下的屏幕录制：



zsource 的实际使用过程

zsource 命令的开发始末

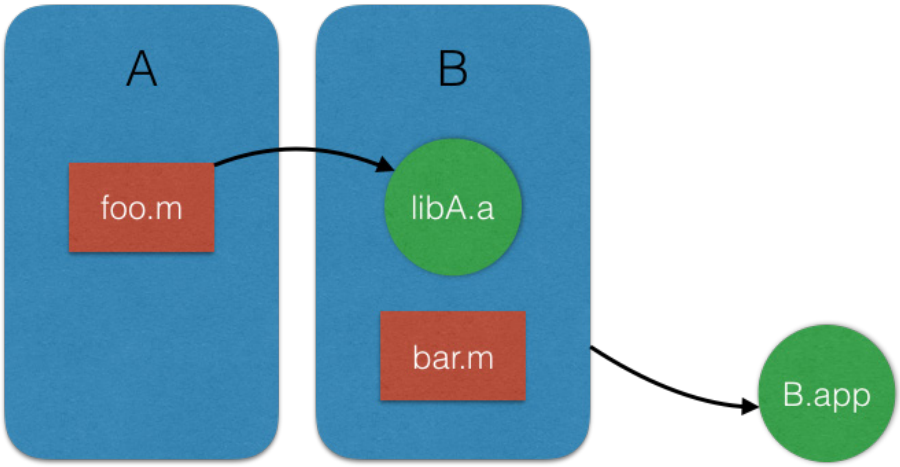
在推出 zsource 功能后，很多同学都对 zsource 背后的技术原理十分感兴趣。其实 zsource 整个功能的开发流程也十分的有趣，就像小说一样，分为几个不同的时期：

- 原理猜想
- 查阅资料
- 简单粗暴的尝试
- 柳暗花明
- 工程化

原理猜想

如果让我们猜想 Xcode 断点调试功能的实现原理，可能大部分人都会猜这样一种可能：Xcode 在编译 Debug 版本的二进制过程中，在二进制中某个字段存储了该二进制所对应的源码的文件地址。当我们在 Xcode 中打断点进行调试的时候，Xcode 会根据二进制中这个字段中存储的源码文件地址，打开对应的源码文件，并在 UI 上展示该源码文件。

道理好像没有什么问题，但是事实是这样吗？在某次团建回国的航班上，我们组成威和志宇两位同学在提出这种猜想后，拿出电脑，做了一个这样的小实验：



实验说明

实验中，他们分别创建了两个 Xcode 工程 A 和 B，工程 A 会产出一个二进制 libA.a。工程 B 中会接将 A 的产出 libA.a 拖到工程中，然后设置 A 中代码的符号断点，然后编译运行。结果发现，当断点断在 A 中的代码时，Xcode 会直接跳转到 A

的源文件中，并且可以继续增加断点以及正常的单步调试。

通过这个实验，成威和志宇同学确定了猜想的正确性。那么接下来需要做的，就是确定二进制中，这个源文件地址信息具体藏在哪一个字段中。

查阅资料

我们都知道苹果的 Mach-O 二进制文件使用的是 DWARF 这种格式来存放调试相关的数据的。但因为我们很难从这个问题中提炼几个精确的关键词在搜索引擎中检索，所以很难通过简单的几次检索就获取到我们想要的答案：二进制这个字段的名称，在初期甚至无法确定这个字段应该是从 Mach-O 的资料中检索还是从 DWARF 的资料中检索。

在没有太好的搜索结果的情况下，我们一度曾经想尝试去从头去啃一啃找到的一些二进制相关的文档：

- osx-abi-macho-file-format-reference
- Introduction to the DWARF Debugging Format
- DWARF 1.1.0 Reference

简单粗暴的尝试

然而，由于对二进制格式不是那么熟悉，也不太了解二进制相关的词汇和概念，所以阅读文档的速度就非常缓慢。

不过，技术的有趣之处就在于，有时候你可以基于我们的猜想，任意去尝试，跳过艰辛的文档阅读过程。在文档阅读遇到挫折后，我们猜想，二进制中很有可能也是用字符来存储这些源码信息的，那么如果我们就把二进制当做字符来看，是不是能搜到一些东西呢?

于是我们试着做了一个比较简单的二进制文件，二进制文件中仅仅包含一个 ZSCViewController，然后用 xxd 这个命令尝试读取二进制中的内容，考虑到 xxd 的输出会折行，我们选取了 ZSCViewController 字符串的子串进行过滤：

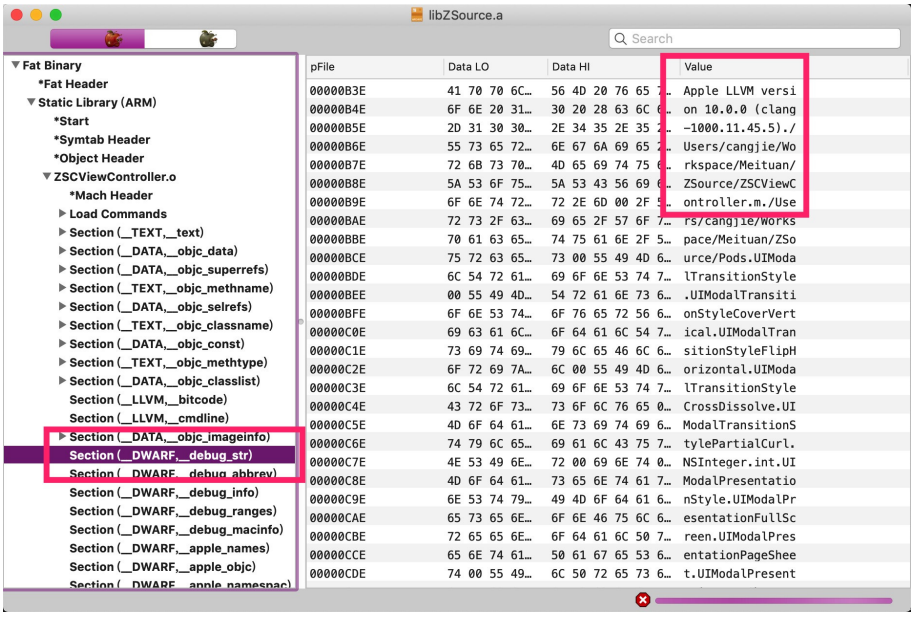
```
xxd ./libZSource.a | grep -C 5 'ZSCViewControlle'
```

果真得到了一些结果：

```
xxd ./libZSource.a | grep -C 5 'ZSCViewControlle'
00001dd0: 564d 2076 6572 7369 6f6e 2031 302e 302e  VM version 10.0.
00001de0: 3020 2863 6c61 6e67 2d31 3030 302e 3131  0 (clang-1000.11
00001df0: 2e34 352e 3529 002f 5573 6572 732f 6361  .45.5)./Users/ca
00001e00: 6e67 6a69 652f 576f 726b 7370 6163 652f  ngjie/Workspace/
00001e10: 4d65 6974 7561 6e2f 5a53 6f75 7263 652f  Meituan/ZSource/
00001e20: 5a53 4356 6965 7743 6f6e 7472 6f6c 6c65  ZSCViewControlle
00001e30: 722e 6d00 2f55 7365 7273 2f63 616e 676a  r.m./Users/cangj
00001e40: 6965 2f57 6f72 6b73 7061 6365 2f4d 6569  ie/Workspace/Mei
00001e50: 7475 616e 2f5a 536f 7572 6365 2f50 6f64  tuan/ZSource/Pod
00001e60: 7300 5549 4d6f 6461 6c54 7261 6e73 6974  s.UIModalTransit
00001e70: 696f 6e53 7479 6c65 0055 494d 6f64 616c  ionStyle.UIModal
```

xxd 命令的输出结果

通过这个实验，我们确定了二进制中源码文件的路径确实是用普通的字符来存储的；紧接着，我们用 MachOViewer 来查看二进制文件，以获取到更友好的二进制信息。利用 MachOViewer，我们可以发现这些信息都存在了二进制的“__debug_str”Section 中。



MacOViewer 的结果

虽然还是不确定这个地址所对应的字段叫什么，但研究到这里，我们还是有所进

展的，最起码我们可以假定这个路径一定是紧跟在“Apple LLVM version 10.0.0 ”字符后面的，然后利用一些读取 Mach-O 的 Ruby 库，比如 ruby-macho，基于这个假定来读取这个路径，为这个特性的工具化提供一丝可能性。

柳暗花明

简单的尝试没有得到想要的答案，但透过 Section 的名字，可以确定源码文件的路径信息和 DWARF 有关。

长时间和 CI 打交道的经验告诉我们，对于每一种二进制格式，苹果公司都会提供一个可以专门用于解析的命令行工具，所以我们就尝试找了找有没有解析二进制中 DWARF 格式的命令行工具。

功夫不负有心人，我们找到了 dwarfdump，那么用它来看看之前的那个二进制文件：

```
dwarfdump ./libZSource.a | grep 'ZSCViewContro'
```

果然有了更好的输出：

```
AT_name( "/Users/cangjie/Workspace/Meituan/ZSource/ZSCViewController.m" )
    AT_name( "ZSCViewController" )
    AT_decl_file( "/Users/cangjie/Workspace/Meituan/ZSource/ZSCViewController.h" )
    AT_decl_file( "/Users/cangjie/Workspace/Meituan/ZSource/ZSCViewController.m" )
    AT_decl_file( "/Users/cangjie/Workspace/Meituan/ZSource/ZSCViewController.m" )
    AT_name( "-[ZSCViewController viewDidLoad]" )
    AT_decl_file( "/Users/cangjie/Workspace/Meituan/ZSource/ZSCViewController.m" )
        AT_type( {0x000011bd} ( const ZSCViewController* ) )
    AT_type( {0x000011c2} ( ZSCViewController* ) )
    AT_type( {0x000001e5} ( ZSCViewController ) )
```

dwarfdump 的输出

这里我们注意到了 AT_name 这个字段名。拿着这个字段名，去前面给出的 DWARF 1.1.0 Reference 文档中查阅，我们可以得知：

> An AT_name attribute whose value is a null-terminated string containing the full or relative path name of the primary source file from which the compilation unit was derived.

进一步查询，我们可以找到另一个和他类似的字段 —— AT_comp_dir：

> An AT_comp_dir attribute whose value is a null-terminated string containing the current working directory of the compilation command that produced this compilation unit in whatever form makes sense Forelax the host system.

看起来，这两个字段就是我们所苦苦追寻的答案了。

工程化

通过实验，以及找到的这两个字段的描述，我们基本可以确定，即便工程是使用二进制构建，只要二进制 AT_name 字段中的路径存在对应的源码文件，App 一样可以使用源码进行断点调试。这种调试方式除了修改源码再次构建不能生效以外，其他的调试场景都和直接使用源码构建无异。考虑到我们日常的调试场景绝大多数都只需要查看其他组件的源码，并不需要修改，把这个功能工程化还是非常有意义的。

那接下来的事情就比较简单了：

1. 首先，我们需要确定大部分美团使用的组件二进制的编译目录是相同的。这样就方便我们在本地某个路径下统一管理下载的源码文件。
2. 接下来，我们通过 dwarfdump 这个命令获取源码文件应该在的路径，然后通过给 CocoaPods 增加命令，将源码文件下载并放置在对应的路径中。

幸运的是，查看完美团 App 的几百个组件后，我们发现只有少数近一年内没有制作过二进制的组件路径比较不同，其他都相同，因此可以先忽略这一小部分组件。如果这部分组件需要支持该功能，只要再制作一次二进制即可。

确定方案以后，写代码就很简单了，最终我们利用 CocoaPods，提供了 zsource 的三个命令：

```
  pod zsource
Usage:

    $ pod zsource COMMAND

      [cocoapods-binary] 通过将二进制对应源码放置在临时目录中，让二进制出现断点时可以跳到对应的源码。 该命令想法由 Zangchengwei 以及
      wangZhiyu 提出，故名为 zsource

Commands:

    + add       [cocoapods-binary] 在不删除二进制的情况下为某个组件添加源码调试能力，多个组件名称用空格分隔
    + clean     [cocoapods-binary] 删除所有已经下载的源码
    + list      [cocoapods-binary] 展示所有已经下载的源码以及其大小

Options:

    --silent    Show nothing
    --verbose   Show more debugging information
    --no-ansi   Show output without ANSI codes
    --help      Show help banner of specified command
```

pod zsource 命令

总结

zsource 功能整体的开发过程基本上都是基于一个个的猜想和实验来完成的，整体的开发上线过程实际上只花了两个晚上。但如果在没有基础知识的情况下，选择把上文中提到的参考资料都看懂后再动手，可能会花费更多的时间。这一个有趣的验证过程也充分说明，有时候我们可以不拘泥于冗长的文档以及资料，通过类似逆向工程的方式，非常快速地拿到我们需要的答案。此时我们再回过头去看文档，可能会获得比直接看文档更好的效果。

最后，非常感谢成威老师和志宇同学对技术的崇高追求，即便在飞机上，也愿意拿出电脑验证自己的猜想，为 zsource 后续的工程化落地提供了更多的可能。

作者简介

宇杰，美团 iOS 工程师，2016 年加入美团，先后参与美团 App 持续集成平台建设、美团 App ReactNative 平台化等工作。目前在参与美团 App 工程效率提升和 Flutter 应用的相关工作。

客户端单周发版下的多分支自动化管理与实践

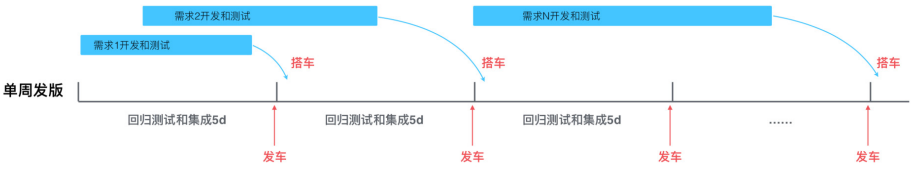
王坤

背景

目前，互联网产品呈现出高频优化迭代的趋势，需求方希望尽早地看到结果，并给予及时反馈，所以技术团队需要用“小步快跑”的姿势来做产品，尽早地交付新版本。基于以上背景，美团客户端研发平台适时地推行了单周发版的迭代策略。单周版本迭代的优点可以概括为三个方面：更快地验证产品创意是否符合预期，更灵活地上线节奏，更早地修复线上 Bug。

首先说一下美团平台的发版策略，主要变更点是由之前的每四周发一版改为每周都有发版。具体对比如下：

- （旧）三周迭代指的是 2 周开发 +1 周半测试，依赖固定的排期和测试时间，如果错过排期，则需要等待至少 20 天方可跟着下个版本迭代发布，线上验证产品效果的时间偏长。具体示例描述如下：



- （新）单周版本迭代指一周一发版，单周迭代版本排期、测试不再依赖固定时间节点，需求开发并测试完成，就可以搭乘最近一周的发版“小火车”，跟版发布直接上线。对于一般需求而言，这将会大大缩短迭代时间。

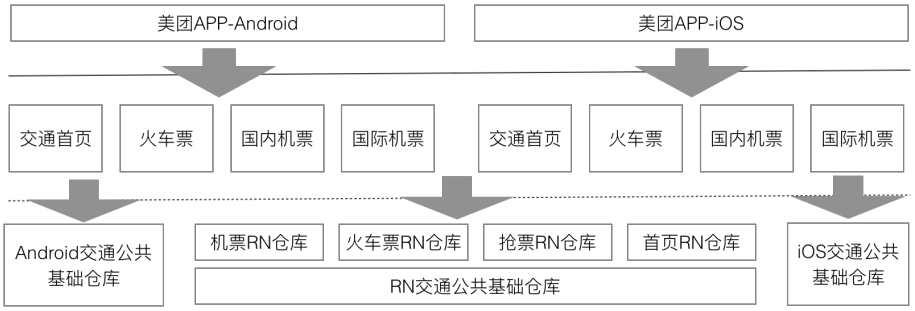
业务方研发人员的痛点

在之前按月发版的迭代节奏下，基本上所有的需求都属于串行开发，每个版本的开发流程比较固定。从“评审 – 开发 – 提测 – 灰度 – 上线”各个环节都处于一个固定的时间点来顺序执行，开发人力资源的协调方式也相对简单。全面推进单周发版之后，并不能把所有需求压缩到 5 天之内开发完成，而是会存在大量的并行开发的场景，之前的固定时间节点全部被打破，由固定周期变成了动态化调配，这给业务方的需求管理和研发人员人力管理都带来了指数式复杂度的提升。一旦进入并行开发，需求之间会产生冲突和依赖关系，版本代码也会随之产生冲突和依赖，这也大大提高了开发过程中的分支管理成本，如何规范化管理分支，降低分支冲突，把控代码质量，是本文接下来要讨论的重点。

下面描述了几种典型的单周发版带来的问题：

- **业务需求开发周期不固定，会存在大量的多版本、多需求并行开发。平台只提供了单周发版的基础策略，每 5 天发一版，业务方完成需求即可搭车发版。**
  对于各业务方来说，需求开发往往并不是都能在 5 天内完成，一般需求在 5~10 天左右，在之前串行发版模式下这个问题其实也存在，但并不突出，在单周发版的前提下，都要面临跨版本开发，意味着多个版本多个需求会同步并行开发，这给业务方的分支管理带来了极大的挑战。

- **业务方架构复杂，仓库依赖多，单周发版分支创建合并维护成本大。**
  交通业务线涉及火车票、国内机票、国际机票多条业务线，代码仓库除了业务线的独立仓库，还有交通首页，交通公共仓库，RN 仓库等多个仓库，Android 端 6 个 Git 仓库，iOS 端 5 个仓库，RN5 个仓库，共计 16 个 Git 仓库。
  多仓库频繁发版分支代码存在安全风险，容易漏合代码，冲掉线上代码。



交通业务线仓库结构示意图

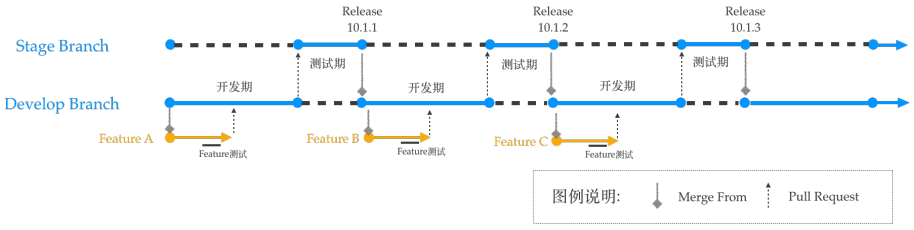
- **业务线自身的公共基础库需求变动频繁。也需要具备单周发版的能力。**

  例如交通公共基础仓库，承载了很多交通业务线的 UI 功能组件，这些公共组件的业务变化频繁，公共基础仓库变化的同时，可能会对使用组件的业务产生影响，需要同步的升级发版。美团平台的策略是公共服务组件每四个小版本统一升级一次，但对业务方自身组件这种策略限制较大，还是需要公共组件也要具备随时发版的能力。

单周发版分支管理解决方案

针对上面提出的问题，交通客户端团队通过技术培训、流程优化、关键点检测、自动化处理等方式保证分支代码的安全。技术培训主要是加强技术人员分支管理的基本知识，Git 的正确使用方法，这里不做过多描述。本文主要讨论关键点检测，以及如何进行自动化的分支管理。

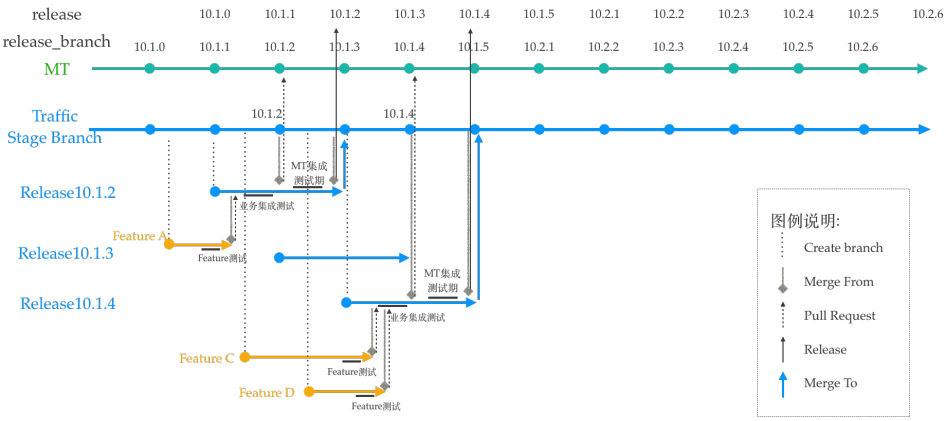
在实施单周发版之前，业务方代码仓库只有两个分支，Develop 分支，即开发分支；Stage 分支，即发版分支；开发流程基本在串行开发模式，每个版本 10 天开发，8 天测试，然后进入下一版本的开发。



之前的串行发版模式

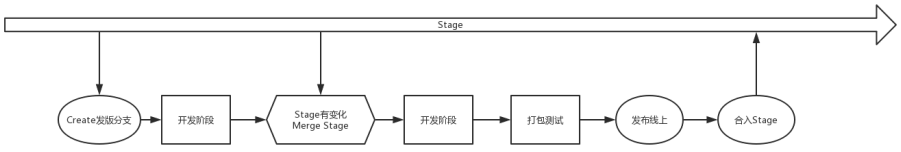
这种方式只能适用于节奏固定的长周期开发方式，对于多版本并行开发来说，有点力不从心，显然已经不能承载当前更灵活的发版节奏。

针对这些问题，我们推出了如下分支管理结构。总的来说，就是废除之前作为开发分支的 Develop 分支，建立对应的 Release 发版分支，每个版本打包从 Release 分支直接打包；同时 Stage 分支不再承担打包职责，而是作为一个主干分支实时同步所有已发布上线的功能，Stage 分支更像一个“母体”，孵化出 Release 分支和其它 Feature 分支；当 Release 分支测试通过、并且发版上线之后，再合入到 Stage 分支，此时所有正在开发中的其它分支都需要同步 Stage 分支的最新代码，保证下一个即将发布的版本的功能代码的完整性。



交通业务单周发版分支管理模型示意图

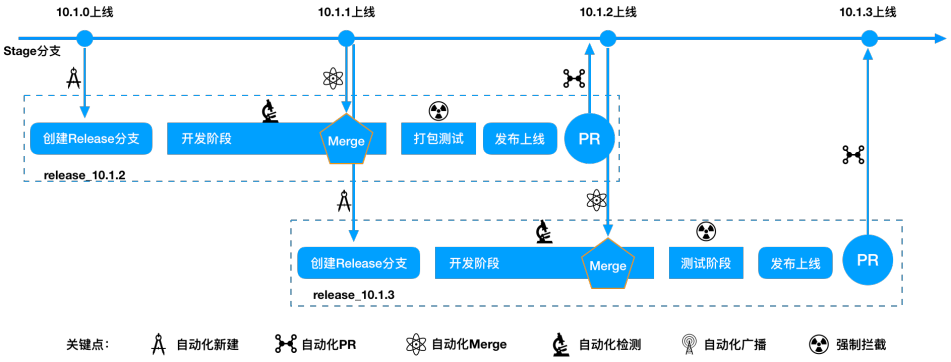
上面的流程描述可能有些复杂，下面是简化的流程图，每个版本都有自己的生命周期：



交通业务单周发版分支生命周期

1. 从 Stage 创建一个 Release 分支；
2. 进入开发阶段；
3. 如果 Stage 分支有变化，同步 Stage 分支；
4. 打包测试；
5. 测试通过，发布线上；
6. 发布线上之后，合回 Stage 分支。

为了适应单周发版，新的开发流程也引入了很多新的挑战。例如下图所示的一个 Branch 分支中涉及的六个关键点：创建分支、合入主干、主干变化通知、Merge 主干变化、检测主干同步、未同步拦截，除了这些还要考虑多仓库同步操作的问题，还有热修复版本的管理方式的问题。能否把这些关键节点合理的规范和把控起来，是我们当前应对多分支并行开发的主要难点：



交通业务单周发版关键节点

如何更高效的解决这些问题呢？结合我们当前使用的工具：Git + Atlassian Stash 代码仓库管理工具；Jenkins Build 打包工具；大象（美团内部通讯工具）内网通信工具。目前这三个开发工具已经非常成熟、稳定，并且接口丰富易于扩展，我们需要配合当前单周发版的分支管理模式，利用这些工具来进行扩展开发，正所谓“要站在巨人的肩膀上”。

1. 创建分支 Release 分支如何创建，何时创建，分支命名规范定义如何约束?

创建 Release 分支，本质上是从 Stage 新建一个分支，当前提前一个周期创建新的发版分支，例如在 10.1.1 版本 Release 后，创建 10.1.3 版本的分支，此时 10.1.2 版本处于开发测试阶段。业务方所有的分支命名和平台的分支命名保持一致，采用 Release/x.x.x 的格式，但同时需要升级成为即将发布的 Release 版本号，例如 10.1.3。

现在交通业务线多达十几个仓库，每个仓库每周都要操作一次需要耗费大量人力。之前每个分支的创建都是通过 Stash 或者手工创建，能不能自动化批处理的创建呢? 答案是肯定的。对此，我们采用了 Jenkins 的方式，需要建立一个 Jenkins Job, 基本原理就是通过命令行的方式进行 Branch 的创建，然后通过 Job 管理，批处理建立所有仓库的 Release 分支，这样就收敛了 Branch 的创建，即采用统一的命名规范，并且同时升级版本号。这就解决了创建分支的难点，实现了自动化创建分支，并且实现了规范化命名。

2. 如何知道 Stage 分支有变化，变化后需要做什么，不做会怎样?

一个好的开发习惯，就是每天写码之前都同步主分支，但是还是需要一个机制来确保同步。这里做了三个措施来确保各个分支和 Stage 是保持同步的：一个通知，一个警告，一次拦截。这三个步骤解决主干变化通知、检测主干同步、未同步拦截的问题。

一个通知：具体路径如下，建立了一个内部推送公众账号和一个 Jenkins 监听 Job，当所有交通业务仓库 Stage 分支有代码改动，通知所有对应的开发人员，该仓库有代码变化，请及时合入。

一次警告：本地开发过程中，每次提交代码到远端仓库时，会触发一个 Stage 分支代码同步检测的脚本，如果发现未同步，会通过内部通讯系统通知提交者存在未同步主分支问题。但这里目前并不做强制拦截，保证分支代码开发的整体流畅性。

最终拦截：在开发分支打包的过程中强制拦截，最终功能代码上线还是需要打包操作。在打包操作时统一收口，由于之前打包也是在 Jenkins 上来完成的，这里我们也是通过在打包 Jenkins 上接入了分支合并检测的插件，这样每次打包时会再次检测

和主分支的同步情况。如果发现未同步则打包失败，确保每次发版都包含当前线上已有代码的功能，防止新版本丢失功能。

3. 如何合并分支，如何保证漏合?

和上面提到的第一个如何创建分支的问题类似，通过 Jenkins Job 来进行批量操作，可以一键创建所有分支的 Pull Request；在每个版本发版之前，统一进行一次打包，合入美团的主分支，防止多个仓库有漏合的情况。

4. 公共基础库版本策略?

公共基础和业务分支保持同样的策略，通过批处理脚本同时建立分支，合并分支，监听分支变化，需要注意的是，每次版本升级，公共基础库也需要同步的打包，并且强制业务库升级。不然，如果基础仓库存在接口变动，有的业务升级了，有的业务没升级，最终会导致无法合入主分支，进而无法打出 App 包。

5. 热修复的版本管理策略?

热修复确实是一种非常规的处理方式。从原则上来讲，热修复需要在对应的 Release 分支上进行修改，然后把修改合入 Stage 分支，同时需要同步到其它正在开发的分支。实际的处理需要根据具体情况来分析，是否需要对线上多个版本热修复。如果多版本都要修复就不能再合入 Stage 分支，否则会导致 Stage 分支冲突，如果把 Stage 分支合入需要热修复的其它分支，会把线上当前最新代码带入历史旧版本，会导致版本兼容性问题。最终执行起来可能还是对热修复版本进行单独处理，不一定要进行 Stage 主分支的同步，热修复的版本管理成本会比较高，更多的需要人工介入。

未来展望

目前整体的分支发版流程已经基本完成，现在已经稳定运行了 10 个小版本，同时没有出现因为分支管理问题而引发的线上问题。

不过，当前整体流程的自动化程度还有待提高，每周需要人工去触发，很多代码合并过程中的冲突问题还需要人工去解决。未来我们希望能够自动化地根据平台的版本号自动创建分支，并且对于一些简单的冲突问题拥有自动化的处理能力。随着单周

发版的不断成熟，未来对于持续交付能力也将不断提升，发版节奏可以不限于单周，一周两版或是更快的发版节奏也成为一种新的可能。

作者简介

王坤，美团客户端开发工程师，2016 年加入美团，目前主要负责大交通业务的客户端架构、版本管理及相关工作。

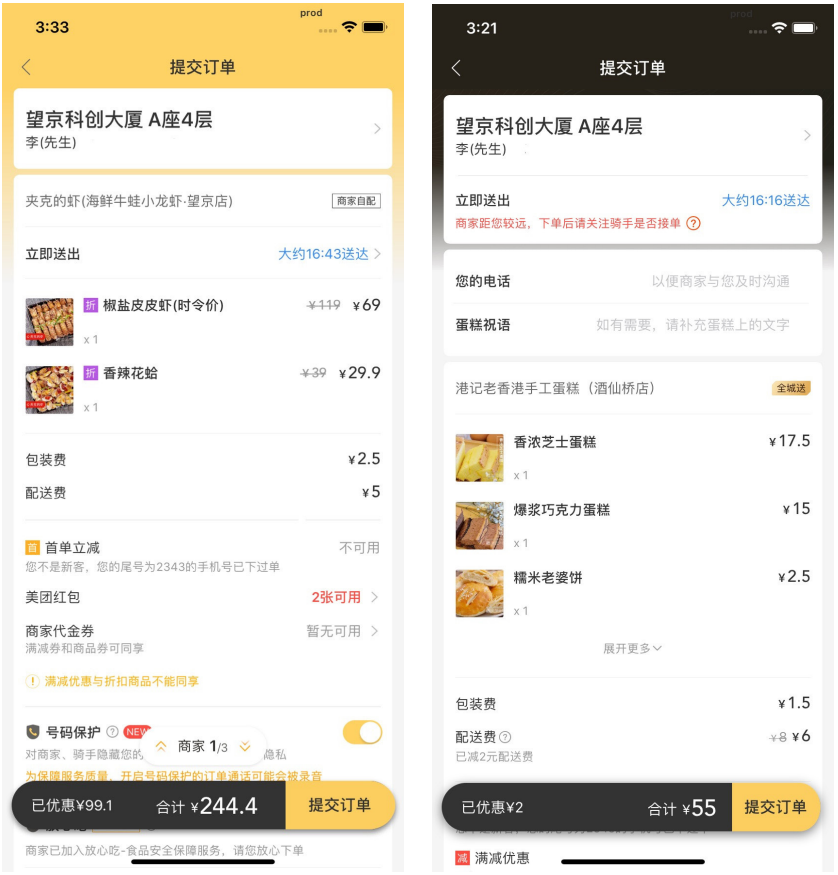
美团外卖前端容器化演进实践

李肖　廷瑞　彦平　同同

背景

提单页的位置

提单页是美团外卖交易链路中非常关键的一个页面。外卖下单的所有入口，包括首页商家列表、订单列表页再来一单、二级频道页的今日推荐等，最终都会进入提单页，在确认各项信息之后，点击提交订单按钮，完成最终下单操作。



所支撑的业务

虽然提单页的代码统一放在外卖代码仓库中，但根据业务发展的需要，提单页上的模块分别由不同的业务部门去负责维护。主要包括以下业务方：

外卖侧业务

- 提单页绝大部分模块的需求开发和日常维护都是由外卖侧的研发同学在负责，包括地址模块、商家商品信息模块、折扣信息模块、准时宝、隐私号、发票备注等。

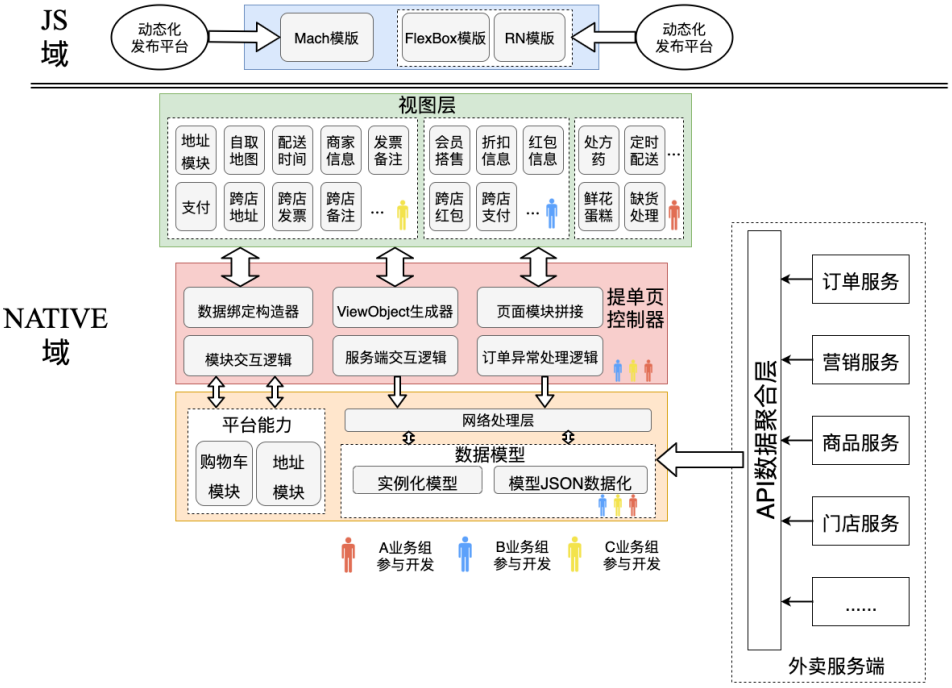
闪购侧业务

- 当从商超等频道进入提单页时，提单页生成的是闪购侧订单，闪购侧的订单在

配送方式、红包、下单路径上都与外卖订单有所区别，但又依赖于外卖的基础功能模块，因此与外卖侧功能存在严重的耦合问题。

其他业务

- 提单页上的部分模块对动态化配置能力有着很高的要求，这些模块使用 Mach 等动态化模版来实现相关的业务逻辑，由专门的业务组负责开发和维护。



容器化改造前提单页架构与开发人员分工

随着业务的不断迭代，提单页的模块也越来越多，逻辑的耦合也越来越重。现在提单页的 UI 展示模块已经超过 30 个，这些模块的展示与否基本上通过服务端的下发数据来决定。在不同的订单类型下，提单页所展示元素的差异越来越大，很多模块的代码已经不适合统一放在一起维护，代码拆分的需求十分强烈。此外，客户端包体积是衡量客户端性能的重要指标，为了解决业务发展带来的提单页代码量急剧增长的问题，同时实现页面元素的动态配置，我们希望能够实现提单页的动态化，而动态化需要基于容器来实现，所以我们提出了提单页的容器方案。

问题和挑战

提单页的容器化与外卖首页的动态化有以下几点不同：

1. 提单页整体动态化的需求不是很强烈，并且 API 改造的成本比较高，因此 API 接口字段保持不变，需要在客户端层面去做转换。
2. 首页模块基本仅作为展示用途，提单页模块的交互逻辑要复杂一些，比如发票模块，进入二级页面操作完成后还要更新提单页的数据。
3. 首页模块的 UI 展示各模块之间是完全独立的，而提单页的模块是根据功能聚合在一个组，这些模块条件出现的位置不同，展示的样式也不一致，如下图备注发票模块所示，最上层和最底层的模块上都带有圆角，所以提单页需要外层再添加一个模块组。

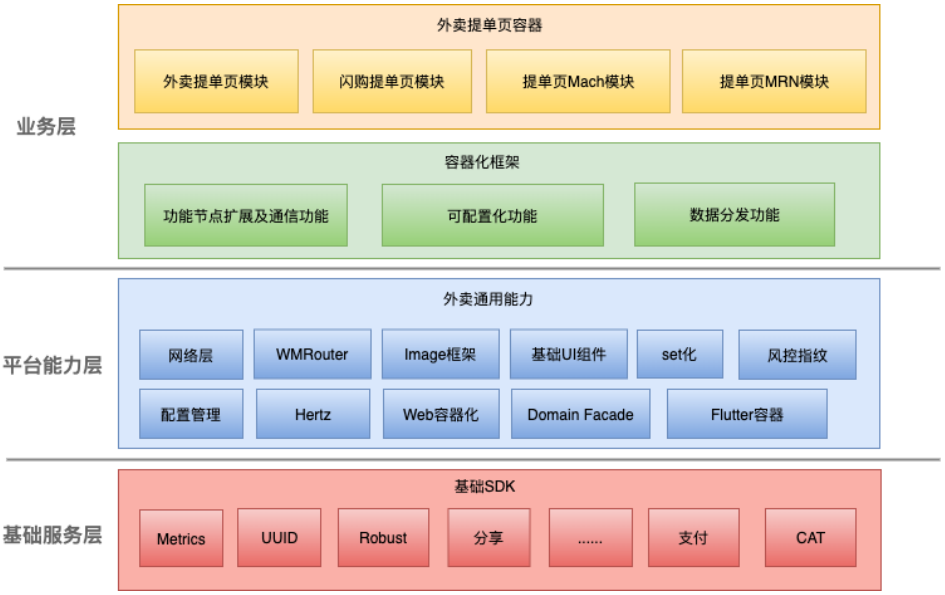
容器化后的提单页，需要实现模块之间的互相无感知，根据服务端的下发数据，客户端可以将闪购代码仓库内的模块和外卖代码仓库内的模块拼接起来组成完整的提单页展示给用户。当用户在提单页完成一系列操作时，各模块可以提供必要的参数给服务端。要想实现这一点，我们需要考虑以下几个问题：

- 模块注册问题，如何在无直接依赖的情况下，让提单页获取页面可用模块。
- API 数据分发问题，如何将服务端字段转换为模块可用数据，同时不侵入到模块这一层。
- 通信问题，模块之间如何实现联动效果。

- 页面更新和复用问题，在提单页刷新时如何提交数据给服务端以及如何完成模块的更新。

设计方案

1. 容器化整体的架构图设计

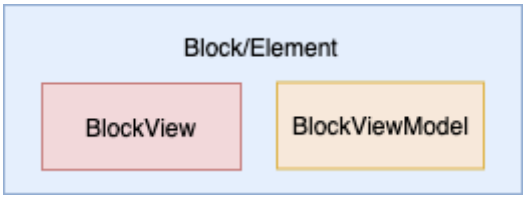


容器化是我们在外卖平台化之后对多方业务能力的支持和扩展，在不改变 API 数据源等前提下，我们保证其具有动态可配置化的能力。为了更好地支撑业务，我们在业务层面抽离出来容器化框架层，其所提供三个部分的核心功能：1. 功能节点扩展及通信功能；2. 可配置化功能；3. 数据分发功能。在最上层业务容器中，目前所支持外卖提单页面模块、闪购提单页模块、提单页 Mach（外卖动态化模板）模块、提单页 MRN（RN 页面）模块四种不同的业务。

1.1 概念解释

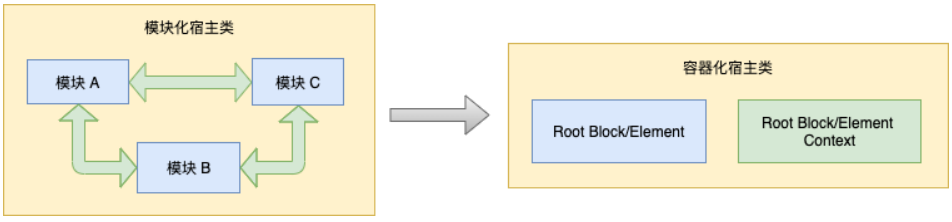
在容器化框架设计过程中，我们引入了一些新的定义，比如在 Android 端引入了 Block 的概念，这里的 Block 是一个功能模块的简称。在提单页页面中，我们可以理解为一个 Block 对应一个功能条目。在 iOS 端有与之对应的概念 Element（由于

两者没有差异，下文陈述中用 Block 代指两者）。



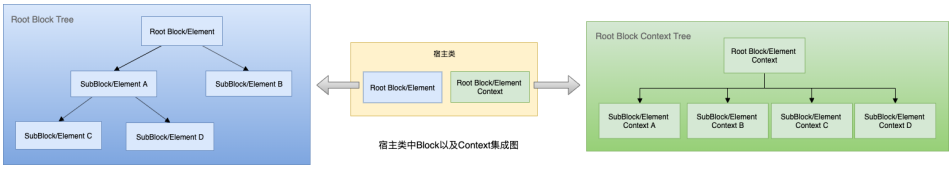
Block 有两种类型：其一是普通的 Block，其包含 BlockView（视图层）和 BlockViewModel（数据层）。BlockView（视图层）用来展示具体的视图以及内部的业务逻辑；BlockViewModel（数据层），用来数据解析。其二是 LogicBlock，是没有视图的 Block，单纯地用来做数据业务处理。

1.2 整体概述

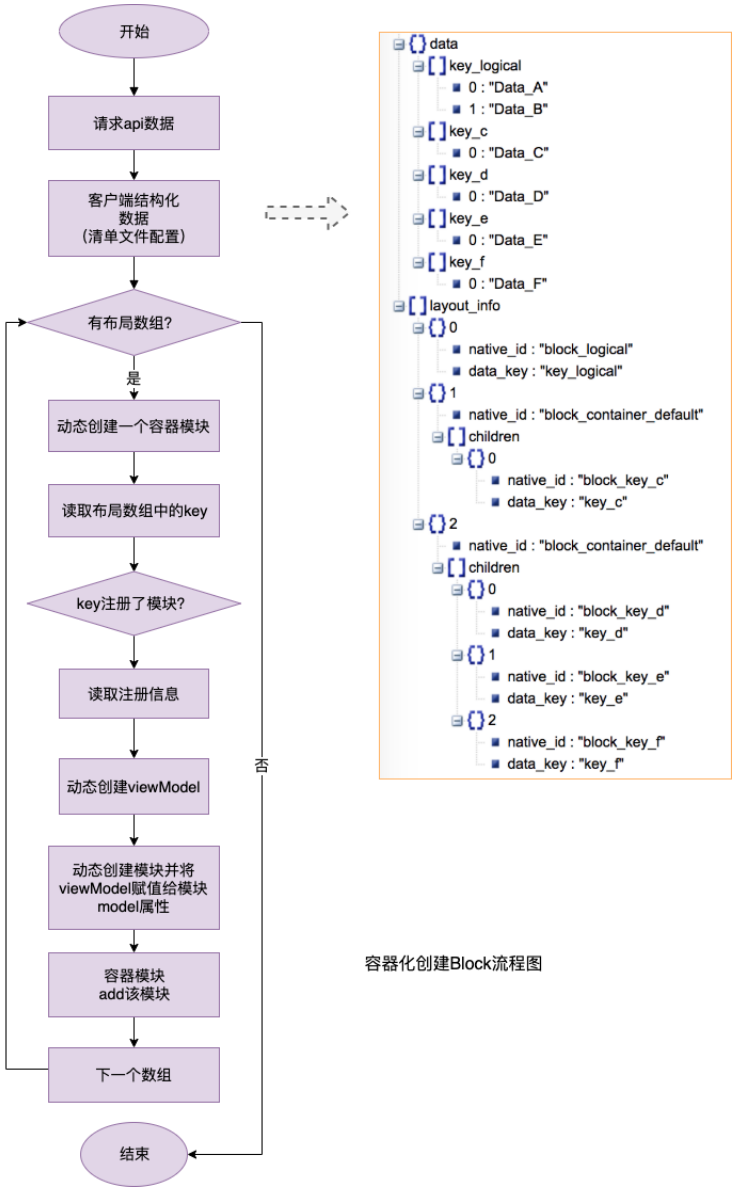


在容器化之前，我们的业务大多是模块化的结构，模块化宿主类是承载所有模块化的管理类，各个模块之间通过宿主类或者控制器进行数据交互。但在容器化改造中，我们将之前宿主类中管理的模块进行拆解，并重新定义了宿主类的职责。在容器化宿主类中，我们将不再持有各个功能模块的引用，而只要持有 Root Block 这一个实例，就可以完成对所有功能模块的管理。而 Root Block Context 则用来处理父 Block 与子 Block 之间的通信以及子 Block 之间的通信。

1.3 核心功能

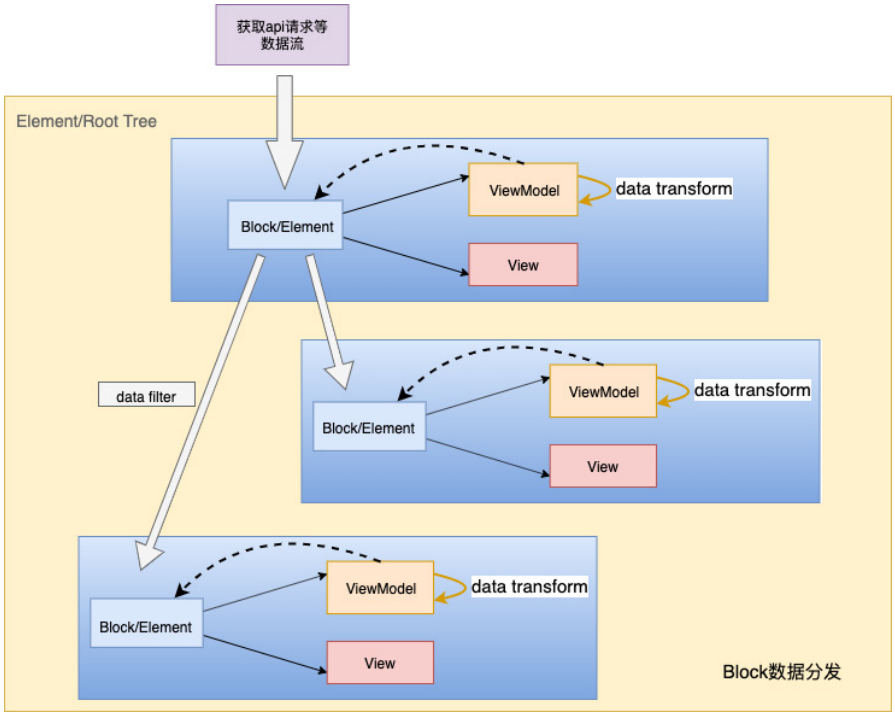


第一部分功能节点扩展及通信功能。主要是目前页面的集成和通信关系，其中 Root Block 是 Block Tree 的根节点，下面会挂载一些 SubBlock 子节点，Root Block 会控制整体的数据流的分发以及整体样式；Root Block Context 可以理解为上下文环境或通信的总线。每个模块都有自己的 Context，来维护自己向外部提供数据以及业务逻辑的能力，这些子 Context 会统一注册到 Root Context 中进行管理维护。



容器化创建Block流程图

第二部分可配置化功能。在发起数据请求成功之后，客户端根据注册的 Key 以及接收到的数据，动态创建 Block 的容器化能力。遍历解析数据以及配置文件，先动态创建 viewModel，将创建好的 viewModel 绑定到生成的 Block 模块上，动态添加到 Root Block 中。多业务方在完全不用相互感知的情况下，完成对新增模块的开发。



第三部分数据分发。既将解析之后的数据，由 Root Block 节点进行数据分发到各个子 Block，各子 Block 的 BlockViewModel 在更新数据之后并回传到 Block 中，Block 用更新后的数据更新 View 的展示。其中，数据可以自动完成分发，也可以手动的接管数据流进行相应的处理。

2. Block 注册问题

2.1 Android 注册的设计方案

Android 是在编译时期，通过 APT（注解处理器）的方式，将在指定模块上的注解信息和 Block 类关联起来，生成 Block 类对应的工厂类，然后将这些工厂类存在全局的 Map 集合中，并在运行时进行初始化操作。

```
@DynamicBinder(nativeId = "block_key_d", viewModel = blockDViewModel.
class, modelType = blockDInfo.class)
```

NativeID 是用来标识 Block 块的唯一 Key，viewModel 是用来绑定 View 视图的数据层，modelType 对应着 API 的数据 Model。

2.2 iOS 注册的设计方案

iOS 使用 Kylin 注册，Kylin 是美团平台开发的基建库，利用 Clang 提供的 section() 函数，在编译时 Kylin 将 {kylin_Key,kylin_Data} 格式的数据写入到可执行文件的特定数据段中，运行期就可以通过读取指定的 Key 值获取相应的数据。 使用这种方式，注册代码分散在每个组件内部。注册内容：组件 native_id、Element 名称、viewModel，其含义同上。

```
PGA_ELEMENT_REGISTER(order_action, OrderActionElement, OrderActionViewModel)
```

注册宏：

```
#define PGA_ELEMENT_REGISTER(NATIVE_ID, PGA_ELEMENT, PGA_VIEW_MODEL)  \
   KLN_STRING_EXPORT("AppKey_"#NATIVE_ID"", "{ \""#PGA_VIEW_MODEL"\"
: \""#PGA_ELEMENT"\"}");
```

3. API 数据结构化

由于 API 下发数据的不规范性，需要将数据按照 data_key 这种数据模式的方式进行整理，然后在获取数据之后，按照规则进行数据解析并创建相应的功能 Block。

目前 API 数据返回的格式：

```
{
  "data":{
        "xxx_pay_by_friend": true,
        "xxx_by_friend_tip": " 发给微信好友，让 TA 买单 ",
        "xxx_by_friend_bubble_desc": " 找微信好友买单 ",
        "xxx_friend_order": false
  }
  "code":0,
  "msg":""
}
```

由于这种格式是平铺分散的，没有将特定功能点的字段聚合在一起表示，不利于我们动态地将数据 Model 与 Block 绑定在一起。

需要我们将一个模块的数据统一在一个 JSON 对象中，整理之后 API 数据返回的格式如下：

```
{
  "data":{
       "pay_by_friend":{//key
          "xxx_pay_by_friend": true,
          "xxx_by_friend_tip": " 发给微信朋友，让 TA 买单 ",
          "xxx_by_friend_bubble_desc": " 找微信好友买单 ",
          "xxx_friend_order": false
       }
  }
  "code":0,
  "msg":""
}
```

将平铺的 API 数据整理成定制的结构化数据，将 Key 作为唯一的标识，那么就可以方便地用来对应指定模块化 Block 中所需的数据 Model。

布局及位置信息会对应相应的模块视图层，这由另外的 layoutInfo 字段给出。数组中的每条元素对应每一个 Block 模块 ，其中 native_id 的值是唯一的且与上面 Block 在注册时候的 Key 保持一致，data_key 的值映射上面整理之后的 API 数据的 Key，这样在编译时期生成 Block 的时候，就可以动态地关联相应的 ViewModel 以及数据模型。

```
{
  "layout_info":[
      {"native_id":"order_pay_by_friend","data_key":"pay_by_friend"},
      {
        "native_id":"block_container_default",// 容器组
        "children":[
          {"native_id":"order_flower_cake","data_key":"flower_cake"}
        ]
      }
    ]
}
```

当然，这里可以以组为维度将一些功能相似的模块聚合在一起，native_id 的含

义同上，Children 是子 Block 结点的数组。

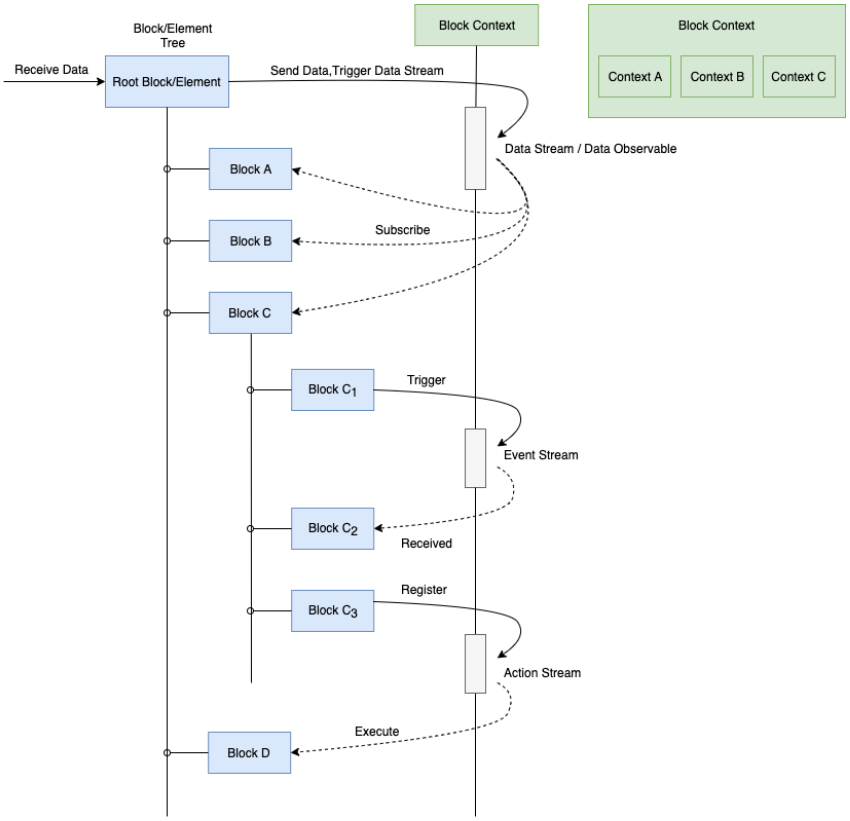
4. 模块间通信问题

由于之前模块化的时候，我们通过中间类的方式承载各个业务模块的通信逻辑。以 Android 为例，我们将多个子模块之间需要通信的逻辑，用接口的方式抛到 Activity 层，由 Activity 层进行业务逻辑的实现，但是由于子模块众多，最终导致该类的膨胀和模块的高耦合性，难以进行扩展和维护。

在容器化设计的时候，为了更好地使各个业务之间进行通信，降低耦合性，我们引入了 BlockContext，同上所述，理解为通信总线。

每个 Block 都有自己的 BlockContext，各个 BlockContext 汇总到 Root Block Context 中去实现，最终，各个 Block 就可以通过 BlockContext 进行数据传递。

整体的通信分发图如下：



图中展示的两种数据方式

4.1 Command 数据交互方式

将所需要的数据包装成事件，在指定的位置驱动事件的执行进而拿到需要的数据。

```
// 声明事件容器
private SupplierCommand<Object> mSupplierCommand = new SupplierCommand<>();
@Override
public SupplierCommand<Object> getSupplierCommand() {
    return mSupplierCommand;
}
// 注册实现
context().getSupplierCommand().registerCommand(new Supplier() {
   @Override
    public Object run() {
    }
});
// 获取相应的 Object 对象
context().getSupplierCommand().execute();
```

4.2 Event 数据交互方式

利用观察者的方式，订阅相应的事件，通过主动触发，从而完成数据分发等不同操作。

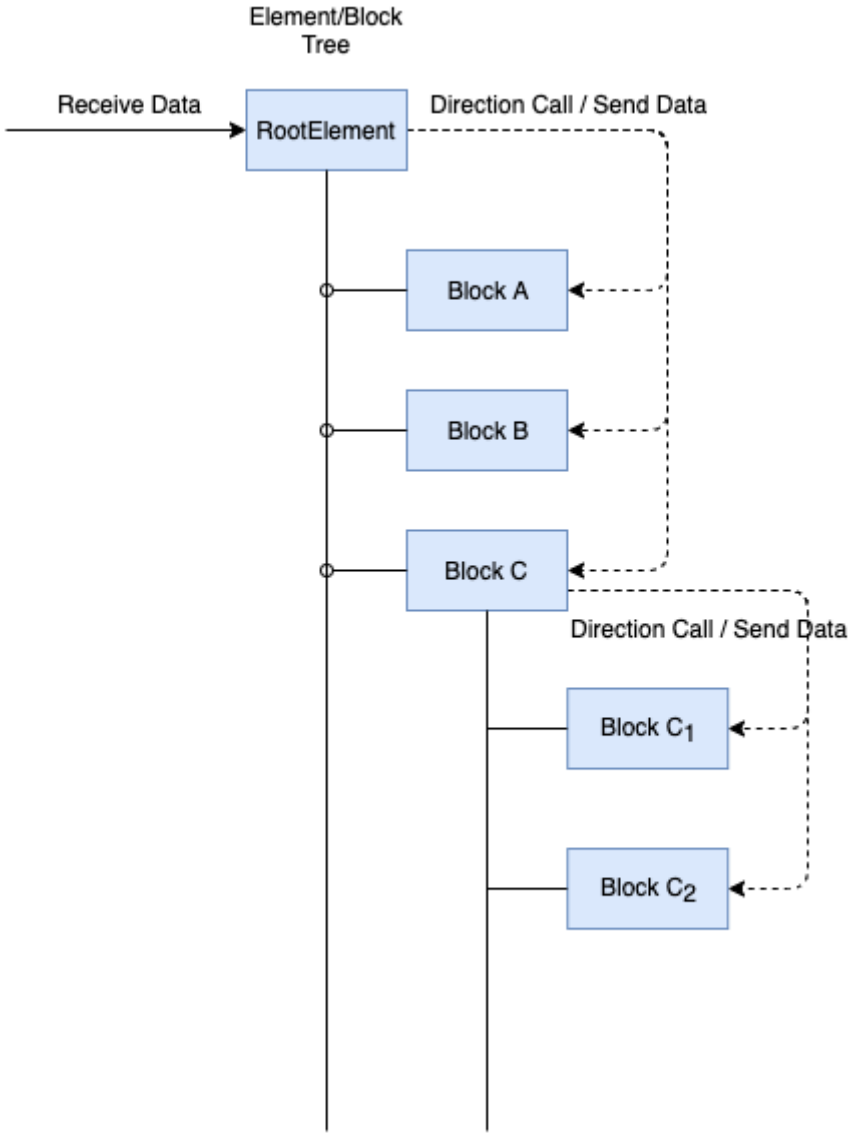
```
// 声明事件容器
private SupplierEvent mSupplierEvent = new SupplierEvent();
@Override
public SupplierEvent supplierResponseEvent() {
  return mSupplierEvent;
}
// 实现订阅
context().supplierResponseEvent().subscribe(new Action() {
   @Override
   public void action() {
   }
});
// 触发相应的操作
context().supplierResponseEvent().trigger();
```

5. Block 页面数据分发问题

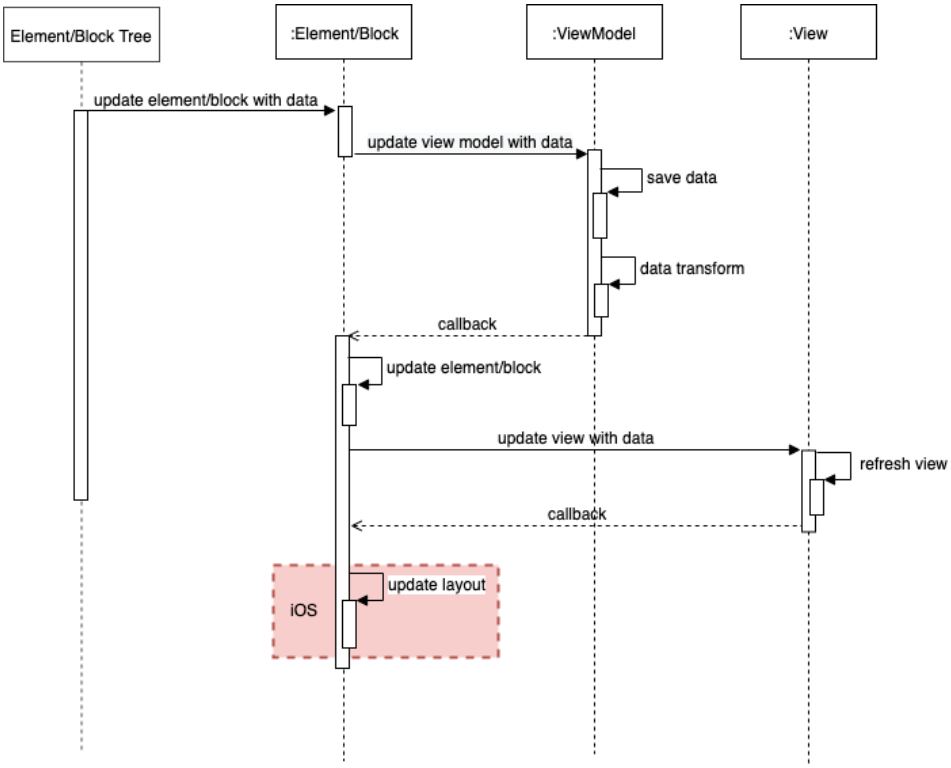
5.1 数据分发问题

Root Block 在接收数据的之后，会按照 Block 结点进行数据的分发。父 Block

将数据逐次的分发给子 Block。



Block Tree 数据分发逻辑简介图



Block 页面的刷新流程时序图

5.2 Block 创建的顺序

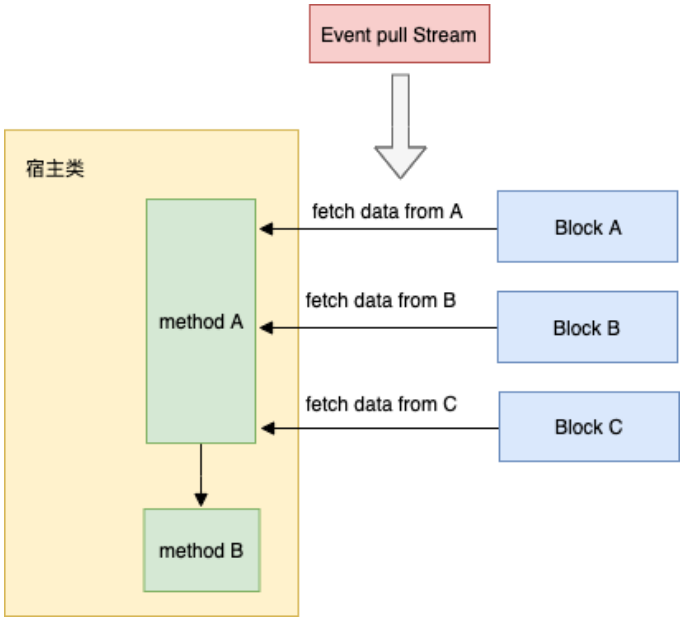
Block 创建的顺序由 API 结构化数据中的 layoutInfo 数组来决定，layoutInfo 数组的具体格式如第三节 API 数据结构化中内容所示。容器化后的提单页会根据 layoutInfo 数组的顺序，依次创建对应 native_id 的 Block 模块。因此，对于一些基础公共模块（比如 wm_confirm_order_logical 对应的 Block），我们可以将其放在 layoutInfo 数组的最前面让其提前加载，保证负责 UI 展示的 Block 创建时数据可用。

5.3 数据拉取问题

由于提单页的模块比较多，在页面曝光、页面刷新或提交请求时，需要从指定的模块获取相应的数据，作为请求的入参，那么如何做成在不感知其他业务方模块的情况下，完成数据的组装呢?

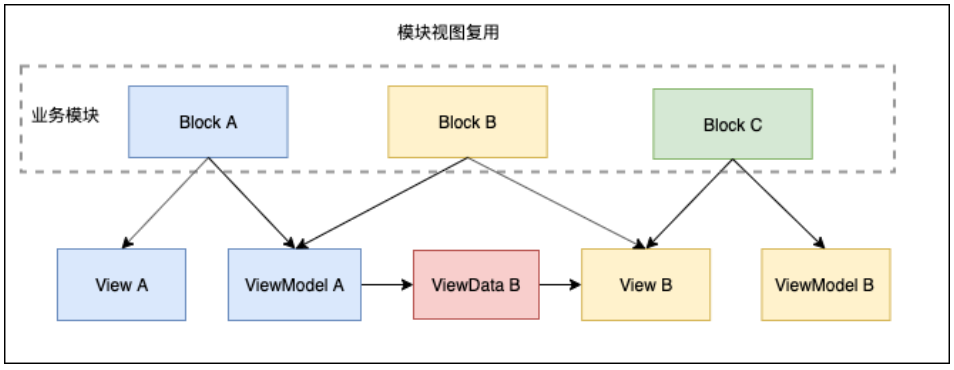
如上面的通信设计思路，我们利用 Event 数据交互方式，从各个模块中将需要

的数据取出来，完成数据的拼装。其中不同业务场景提取数据需要的校验工作，也分散在各个模块中进行处理。最终，即使在物理层面上隔离了对 Block 的感知，但是依然可以完成对请求所需数据的获取。



6. Block 页面的复用问题

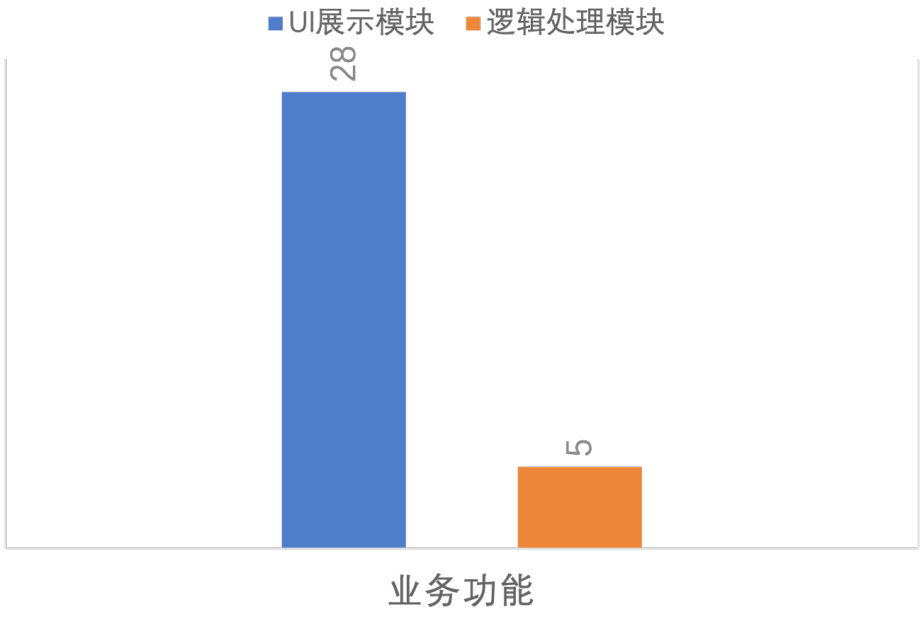
在实际的开发中，有些 Block 的页面 View 大致上相似，但是逻辑上有些细微的差异，为了快速开发，我们在设计上复用了其视图。Block、BlockView 以及 ViewModel 的关系：一个 Block 对应一个 ViewModel 和一个 BlockView，一个 ViewModel 和一个 BlockView 可以对应多个 Block。

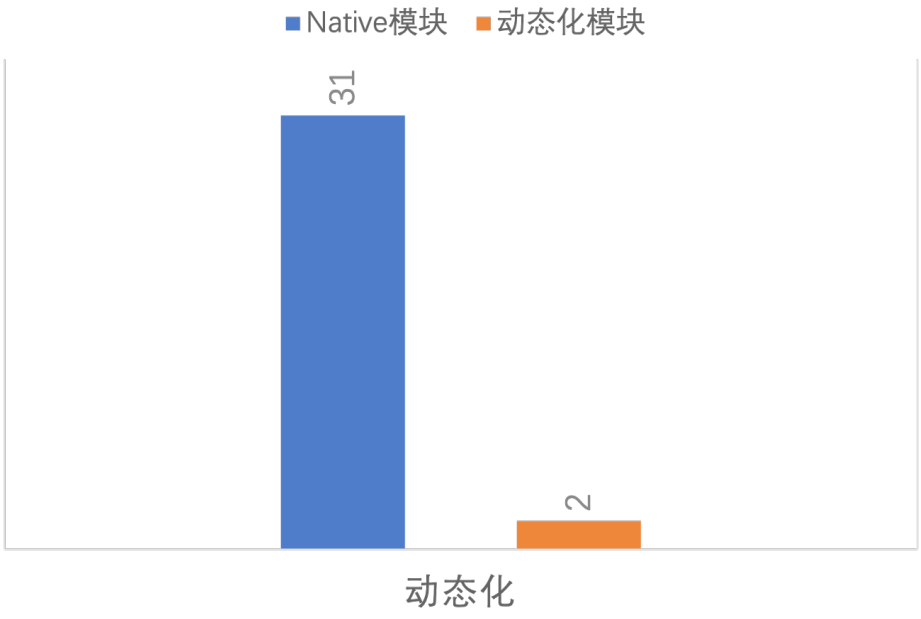


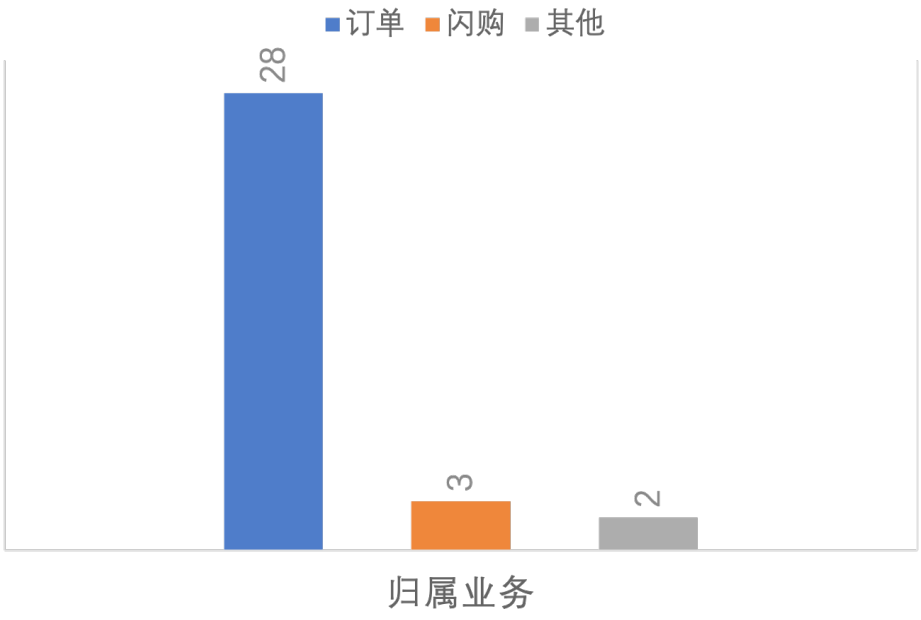
计算机界有一句名言：“计算机科学领域的任何问题都可以通过增加一个中间层来解决。”（原始版本出自计算机科学家 David Wheeler）相似的，为了视图层的复用，屏蔽数据层的差异，我们在数据层的逻辑中转部分引入一个中间层 ViewData，ViewData 是为了更好地适配数据模型以及区别视图展示上的差异，这样就大大提高了代码的复用率。

收益

在开发过程中，我们将 iOS 和 Android 系统的模块进行了对齐和统一，容器化完成之后，两端同一 NativeID 对应的模块展示着相同的 UI 数据，也具有完全一样的业务逻辑。经过容器化后的提单页，相关代码被划分到了 33 个模块当中，这些模块分别承担着不同的职责。这里按照模块的业务功能、所采用的技术栈和所属业务线将这些容器化后的模块进行划分，得到如下的柱状直方图：







容器化之后的提单页完全由各模块组成，这些模块可以负责 UI 展示，也可以不展示任何 UI 模块，单纯地处理业务逻辑。模块内部的开发方案也可以根据业务形态

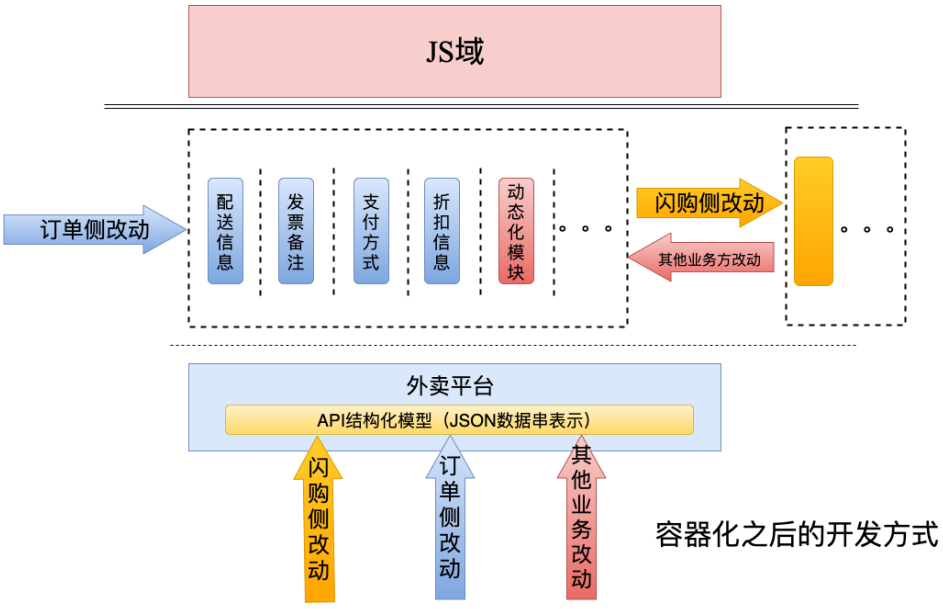
自由选择，相互之间做到了完全无感知。这些优点为后续提单页的业务迭代和技术优化都提供了很大的空间。

解耦的收益

开发效率提升

容器化之前的提单页，页面各部分共享同一个数据模型，服务端接口数据返回后，在提单页控制器内进行数据的更新、过滤和二次加工之后，再分发给页面上的各模块。当不同的 RD 同时开发提单页的需求时，这些放置在一起的业务逻辑会提高 RD 的开发成本，另外很容易出现代码层面的冲突，在需要 RD 手动解决的同时，也很容易因为开发流程的不规范出现 Bug。

容器化之后的提单页，开发模式也相应发生了改变，RD 在开发过程中，不会感知到别的模块内业务需求的改动，各业务可以在各自的容器内进行有效的推进迭代，而不用担心会影响到其他业务，从而让多人协作开发效率得到一定的提升。

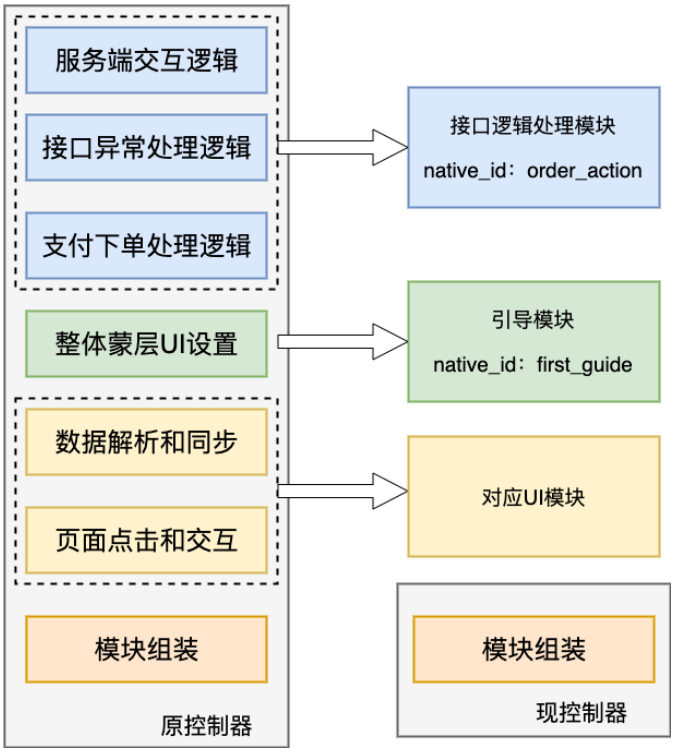


容器化之后的开发方式

控制器瘦身

在客户端业务开发的层面，MVC 架构得到了广泛应用。容器化重构之前的提单页，虽然也以模块化思想为基础做过多次重构，但是依然深受 MVC 思想的影响，在提单页控制器内保留了大量业务逻辑的代码。这些业务逻辑的代码最终会在业务迭代过程中逐渐变得臃肿和不可控，最终成为下一次代码重构计划中的业务背景。

本次提单页的容器化改造彻底解决了这一问题。基于 PGA 框架，包括接口异常处理、数据模型传递和二级页面跳转等业务逻辑代码都被收入到对应的 Element 和 Block 中，改造后的提单页中已经不存在业务逻辑相关的代码，彻底杜绝再次出现臃肿页面 VC 的可能。经统计，iOS 侧提单页控制器的代码行数从 2894 行减少到 289 行，控制器类中仅包含 Block 组装的业务逻辑。



包体积减少

提单页承载着美团的外卖业务和闪购业务，在未进行容器化之前，两个业务方需

要同时向订单库提交代码，在订单库整体“瘦身”的过程中，我们发现这种开发模式让包大小优化的工作多次出现反复，并且统计指标也难以统一和对齐。对提单页进行容器化改造之后，外卖和闪购分别维护各自模块内的代码，相互之间无依赖，闪购侧可以直接在自己的代码仓库内完成提单页模块的新增和修改，不需要再给外卖代码仓库提 PR，也就不会对外卖侧的包大小统计产生影响。

动态化的收益

动态化是整个外卖业务的发展方向。提单页的动态化建立在容器化的基础之上，在完成容器化之后，就具备了动态化的基础。当前提单页的动态化，所指的主要是模块层级的动态化，提单页的各模块展示顺序、展示与否，都可以完全由根据服务端下发的数据决定，各模块可以自由地进行组合、拼装，实现提单页的动态配置。

两端对齐的收益

之前因为历史原因，提单页很多的功能模块，Android 和 iOS 在实现上大相径庭，完全不一样的实现让两端在新业务需求到来时，在与服务端接口对接、开发工时和开发方案上都存在很大的差异，这些差异点对产品需求的排期、开发和测试上线上都产生了很多负面的影响。

提单页在容器化后的另外一项收益，就是 Android 和 iOS 在模块层级的代码实现，完成了统一。借助于 PGA 框架和 Element 注册机制，Android 和 iOS 具有大致相同的模块结构，相同 native_id 的模块获取的 API 接口返回字段完全一致；在页面请求接口数据时，相同 ID 的模块也提供同样的数据字段。在后续的开发过程中，两端对 API 接口字段的请求趋于一致，可以最大程度地减少因为两端不一致带来的合作方开发成本，也可以在一定程度上减轻下游的测试压力。

总结与展望

外卖客户端一直在推动核心页面的标准化，同时一直在探索尝试让核心页面也具备动态化能力。提单页作为下单路径上的核心页面，在 PGA 框架的基础上完成了容器化重构。至此，外卖首页、点菜页和提单页在页面这一层级都统一使用 PGA 框架

实现。统一化和标准化之后，可以让编程风格趋于一致，代码结构在不同平台保持统一，在后续的需求开发中，可以有效减少因为两端实现不一致出现的隐性开发成本。

提单页在容器化之后，让区域动态化的技术演进更容易推进。模块之间的解耦让不同模块可以自由选择模块内使用的技术栈而不会对其他模块产生影响。对于提单页的部分模块，完全可以通过 Mach 或者 RN 等动态化方案来实现，通过区域动态化来进一步减少开发成本，提高业务需求的开发效率。

在提单页之后，客户端会继续推进订单状态页使用 PGA 框架实现容器化，让标准化框架对用户下单路径上的核心页面实现 100% 覆盖。同时积极在提单页的商家商品信息展示、放心吃、准时保等模块探索页面的部分区域动态化，进一步缩减包大小，提高开发效率。

附录

1. Mach (马赫) 是外卖终端组自研的多终端跨平台级的局部动态化技术。
2. MRN 是美团基于 React-native 0.54.3 进行的二次封装，抹平了两端上的差异，并且提供了一些基础库和组件库供业务开发同学使用。
3. Metrics 是美团平台团队和外卖团队，开发的新一代 App 性能采集、监控、统计平台。
4. Hertz (赫兹) 是一个自动化的性能采集与监控 SDK，可以在开发、测试、灰度、运维各阶段，采集性能指标、检测卡顿、测量页面加载时间，帮助开发者监控和定位性能问题。

作者简介

李肖、廷瑞、彦平、同同均为美团外卖团队工程师。

招聘信息

美团外卖长期招聘 Android、iOS、FE 高级 / 资深工程师和技术专家，Base 北京、上海、成都，欢迎有兴趣的同学投递简历到 tech@meituan.com。

Bifrost 微前端框架及其在美团闪购中的实践

雨甫

Bifrost（英 [ˈbi:frɔst]）原意彩虹桥，北欧神话中是连通天地的一条通道。而在漫威电影《雷神》中，Bifrost 是神域——阿斯加德（Asgard）的出入口，神域的人通过它自由穿梭于“九界”（指九个平行的宇宙）之间。借用“彩虹桥”的寓意，我们希望 Bifrost 可以成为前端不同 SPA（Single Page Application）系统之间的桥梁，使得不同的单页应用可以用这种方式实现功能的自由聚合 / 拆分。

项目背景

立项之初，闪购赋能企管平台（以下简称“企管平台”）仅仅是面向单个商家的 CRM 管理系统，采用常规的 Vue 单页应用方式来实现。随着项目的推进，它的定位逐渐发生了变化，从一个单一业务的载体逐渐变成了面向多种场景的商家管理平台。另一方面，由于系统由多个前端团队共同开发维护，越来越多的问题随之浮现：

- 异地协作时，信息同步不及时引起的代码冲突以及修改公共组件引入的 Bug。
- 不同的商家针对同一个页面存在定制化的需求。
- 已经实现的一些功能需要集成到企管平台中来。

因此，我们希望构建一个更高维度的解耦方案，使我们能够在开发阶段把互不干涉的模块拆成一个个类似后端微服务架构那样的子系统，各自迭代，在运行时集成为一个能够覆盖上述各种使用场景的完整系统。

方案选型

首先，我们整理了核心诉求，按优先级排序如下：

- 希望异地开发时不同的模块能够独立开发、独立部署。
- 对已在线上运行的项目，希望能够低成本地接入企管平台，而不需要对开发、部署流程做大规模的改动。

- 各个子系统独立运行，互不影响，但允许我们在开发阶段与其他子系统进行联调。
- 保持单页应用的体验。
- 由于现有项目都是基于 Vue 技术栈开发，因此，我们的框架并不需要做到技术栈无关，只要满足 Vue 的项目即可。

基于以上这些诉求，我们调研了目前市面上常用的微前端方案，最常见的方案有：

- 基于 Nginx 的路由分发。
- 使用 Iframe 将页面嵌入。

除此之外，还有美团集团内部的微前端实践——美团 HR 系统（用微前端方式搭建类单页应用）和业界比较知名的微前端框架——SingleSPA。

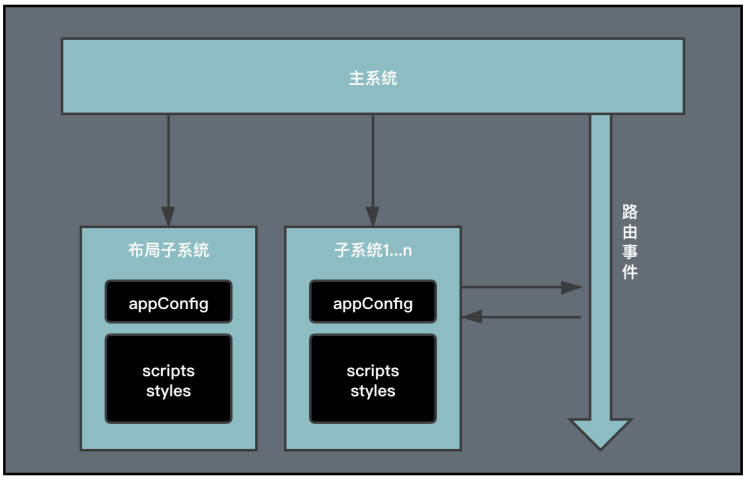
这些方案的优劣整理如下：

方案	实现	优势	缺点
基于Nginx的路由分发	根据路由前缀的不同将用户的请求指向不同的HTML入口文件。	实现简单，且各个子系统可以采用完全不同的技术栈进行开发。	页面跳转时，可能会闪动、白屏，无法做到单页应用的体验；新系统加入时，需要修改线上Nginx配置，风险较大。
Iframe嵌入	在现有系统中直接嵌入Iframe。	实现简单、子系统加载时依然保持单页应用体验。	页面刷新之后，无法保持子系统当前的路由状态；Iframe的适配存在一定问题。
HR系统方案	按模块对系统进行拆分。各个模块独立开发、构建、部署，在运行时，通过Webpack动态加载的方式进行集成。	能够保证单页应用的体验，且经过多次迭代，已经是一个经过生产验证的成熟解决方案。	本质上还是一个单页应用，各个子模块无法独立运行；同其他子系统进行联调的成本较高；新增子系统需要修改Nginx配置，存在一定风险。
SingleSPA	异步加载子系统的入口文件，并动态为子系统创建挂载点，Bifrost也参考了它的实现方式。	目前唯一落地的真正的微前端框架，可以做到技术栈无关。	子系统之间难以联调；需要修改目前的部署流程。

从用户体验角度出发，Nginx 和 Iframe 首先被否决；HR 系统的方案需要对现有的项目进行改造，把不同团队目前开发的项目整合到同一个单页应用中，在项目快速迭代的过程中，成本过高，所以也被否掉。SingleSPA 看起来完美，但它没有照顾到实际生产环境中的开发、部署的差异性，并不是 Product-Ready。综合多种因素考虑，我们最终决定采用自研的方式来实现微前端化 Bifrost。

核心架构

Bifrost 框架在设计的时候参考了 SingleSPA 的思路，将系统了分为主系统和子系统。

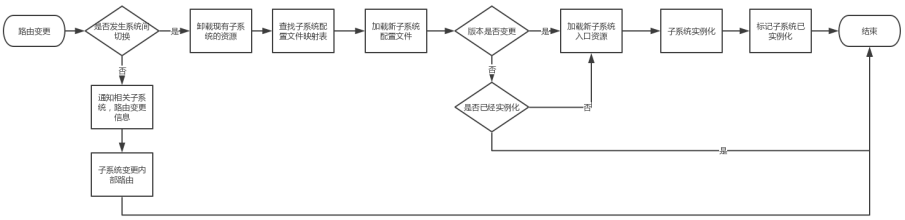


主系统是用来控制子系统的调度中心，职责包括：

- 维护子系统的注册表。
- 管理各个子系统的生命周期。
- 传递路由信息。
- 加载子项目的入口资源。
- 为子系统的实例提供挂载点。

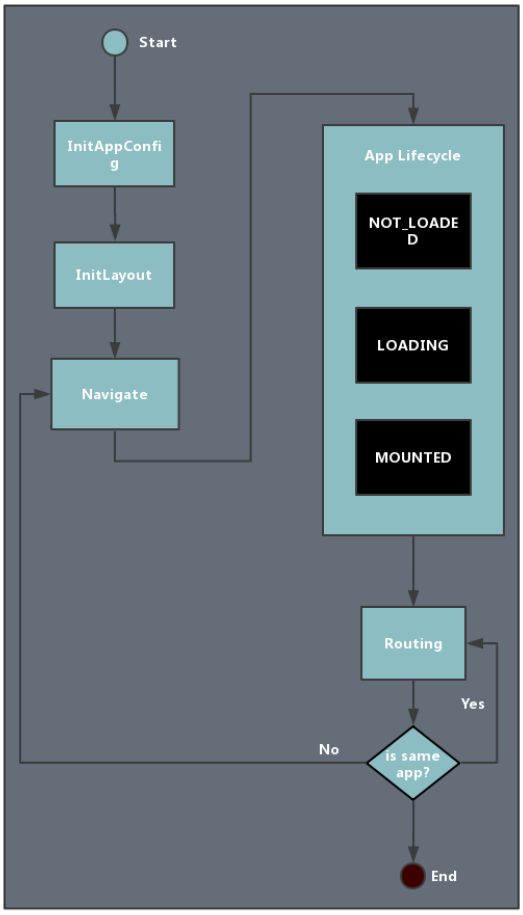
子系统只负责业务逻辑的实现。如果进一步细分的话，子系统可以分为业务子系统、实现公共菜单子系统、导航布局子系统，其中布局子系统会先于业务子系统加载。

Bifrost 采用路由消息分发的方式来控制子系统的加载和跳转。主系统维护了一条路由消息总线，当路由发生变化时，子系统会将路由事件推送给路由总线，然后由路由总线决定加载 / 跳转的目标子系统。如果路由不需要切换子系统，则交由当前子系统进行处理。



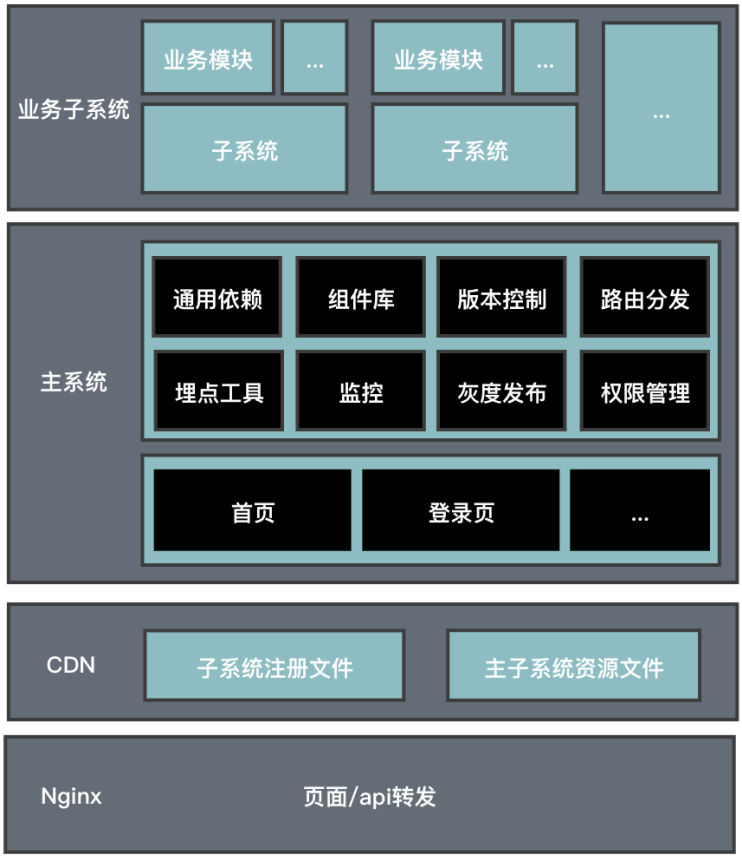
如果子系统发生切换，主系统会在 DOM 中添加对应子系统的挂载节点，并异步加载系统的静态资源。由于子系统都是完整的 Vue 实例，当子系统的代码加载并执行之后，子系统就会自动在其对应的挂载节点上渲染相应的内容。

整个系统的生命周期如下图所示：



具体实现

基于 Bifrost 实现的项目架构如下图所示：



这里，我们主要关注主系统、业务子系统和布局子系统的实现。

主系统

主系统的逻辑比较简单，主要是实例化 Bifrost 中定义的 Platform 对象，并注册各个子系统。子系统的注册信息包括：

- AppName：子系统名，与系统的路由前缀保持对应，同时也会作为子系统在 DOM 中挂载节点的 ID。

- Domain：非必填，如果出现多个路由前缀都对应同一个子系统，可以通过 Domain 进行映射。
- ConfigPath：对应子系统配置文件的 URL。

一个简单的主系统实现如下：

```
import { Platform } from '@sfe/bifrost'

new Platform({
  layoutFrame: {
    render () {
      // render layout
    }
  },
  appRegister: [
    { appName: 'app1', configPath: '/path/to/app1/config.js' }
  ]
}).start()
```

业务子系统

在设计方案时，我们始终保持一个理念，就是保证对业务代码的零侵入，因此业务系统改造的工作量很小。代码层面，只需要把原本子系统的初始化流程放到 AppContainer 对象的 Mounted 回调函数里即可：

```
import { AppContainer } from '@sfe/bifrost'
import Vue from 'vue'
import VueRouter from 'vue-router'

Vue.use(VueRouter)
const router = new VueRouter({})
new AppContainer({
  appName: 'app1',
  router,
  mounted () {
    return new Vue({
      router,
      components: { App },
      template: '<App/>'
    }).$mount()
  }
}).start()
```

另外，还需要修改子系统的构建流程，构建完成之后生成一个包含子系统入口资源信息的配置文件。一个典型的配置文件如下：

```
((callback) => callback({
  scripts: [
    '/js/chunk-vendors.dee65310.js','/js/home.b822227c.js'
  ],
  styles: [
    '/css/chunk-vendors.e7f4dbac.css','/css/home.285dac42.css'
  ]
}))(configLoadedCb.crm)
```

- 此处我们实现了 @sfe/bifrost-config-plugin 插件，在 Webpack 构建脚本中引入该插件就可以自动生成项目对应的配置文件。配置文件是一个立即执行函数，主系统可以通过 JSONP 的方式读取配置文件中的内容。

在实际生产环境中，我们可以将子系统发布到任意 CDN，只要能够保证配置文件的 URL 始终不变，那么无需依赖任何服务，主系统就可以感知到子系统的发布。

布局子系统

布局子系统是用来实现菜单和导航栏的 Vue 工程，本质上和一般的业务子系统没有区别。只需要注意，布局子系统使用的是 LayoutContainer 而非 AppContainer 进行包装。

```
import { LayoutContainer } from '@sfe/bifrost'
import Vue from 'vue'

import App from './app'

new LayoutContainer({
  appName: 'layout',
  router,
  onInit ({ appSlot, callback }) {
    Vue.config.productionTip = false
    const app = new Vue({
      el: '#app',
      router,
      store,
      render: (h) => (
        <App appSlot={appSlot} />
```

```
        )
      })
      callback()
    }
  }).start()
```

布局子系统作为主系统的一部分，既可以放在主系统中去实现，也可以像其他子系统一样通过异步的方式去加载。在我们的项目中，结合了上面两种方式（布局子系统既可以为作为常规的 Vue 项目构建，也可以发布成 NPM 包），每次发布时，会同时发布布局的静态资源和 NPM 包。主系统通过 NPM 包的方式引入布局子系统，将它打包到项目中，避免线上运行时，额外加载布局子系统的资源，减小项目体积，加快渲染速度。

本地开发时，我们则会通过 Bifrost 定义的 MockPlatform 异步加载布局子系统的静态资源，保证线上 / 线下运行效果的一致性，方便本地联调。

工程实践

代码层面的改动虽然不多，但要在实际的生产环境中落地，还需要解决一系列老生常谈的问题，包括：

- 本地开发时，如何保证与线上实际运行效果的一致性?
- 如何实现全局状态管理和子系统之间的通信?
- 如何对公共依赖和公共模块进行管理?
- 发布部署流程需要怎样调整?

根据闪购业务实践，我们总结了一套适用于 Bifrost 的解决方案。

本地联调

采用微前端的方式意味着子系统的完全隔离，这给我们的开发带来了一系列困扰：

- 本地开发时，无法看到当前开发的功能在主系统中实际运行的效果。
- 子系统之间有时会存在跳转关系，在开发阶段难以验证这种跳转逻辑的正确性。

为了解决这些问题，Bifrost 定义了 MockPlatform。MockPlatform 的思路很简单：既然主系统可以动态加载线上的子系统，那么我们只需要在开发时，模拟主系统的运行方式，去加载其他子系统的线上资源，之后就可以像调用后端 API 一样同各个子系统进行联调了。这也就解释了为什么布局子系统在输出 NPM 包的同时还维护了一份静态资源。

MockPlatform 的 API 同 Platform 对象的 API 是一致的，开发时，我们只需要按照主系统的方式引用布局或业务子系统的配置文件 URL 即可：

```
// ...others...

new AppContainer({
  // ...others...
  runDevPlatform: process.env.NODE_ENV === 'development', // 只在开发环境
下启动 mock platform
  devPlatformConfig: {
    layoutFrame: {
      mode: 'remote',
      configPath: 'path/to/layout/config.js'
    },
    appRegister: [{
      appName: 'app2',
      mode: 'remote',
      configPath: 'path/to/app2/config.js'
    }]
  },
  // ...others...
}).start()
```

借助 MockPlatform，我们项目在开发阶段的感受和开发普通的单页应用没有任何差异，如果某个我们依赖的子系统更新了功能，只需要让对应的 RD 发布一下，就可以在本地看到它的最新效果。

全局状态

除了本地联调，全局通信也是微前端项目中绕不开的一个话题。由于我们所有项目采用的都是 Vue 技术栈，所以会选择基于 Vuex 来实现全局通信。Bifrost 的主系统会维护一个全局的 Vuex Store，用于保存全局状态。

当子系统希望监听全局状态时，子系统并不是直接订阅全局 Store（Vue 的依赖收集机制也决定了子系统无法响应全局 Store 的变化），而是借助 Bifrost 提供的 syncGlobalStore 函数来订阅全局 Store。调用该函数后，任何全局状态的变化都会被同步到本地 Store 的 Global 命名空间下。之后，就可以像普通的单页应用那样，调用 Vuex 的 mapState 方法实现和全局状态的双向绑定。

```
import { AppContainer, syncGlobalStore } from '@sfe/bifrost'
import Vue from 'vue'
import Vuex from 'vuex'
// ...others...
Vue.use(Vuex)
const store = new Vuex.Store({})
new AppContainer({
  mounted () {
    // 同步全局 store 状态
    syncGlobalStore(store)

    // ...others...
    return new Vue({
      store,
      router,
      components: { App },
      template: '<App/>'
    }).$mount()
  }
}).start()
```

如果子系统自身的状态需要共享，Bifrost 还会提供 installGlobalModule 函数。该函数会将当前子系统需要共享的状态挂载到全局 Store 下，其他子系统可以通过前面提到的方式来同步这些状态。虽然 Bifrost 提供了子系统通信的能力，但在实际拆分子系统时，应该尽量避免这种情况发生。如果两个子系统之间需要频繁通信，那就应该考虑把他们划分到同一个子系统。

公共依赖

由于各个子系统都需要集成到企管平台，为了保证体验的一致性，大家都是基于同样的组件库进行开发。几乎所有项目都会依赖 lodash、Moment 等基础库，因此如果不对公共依赖进行管理，项目会加载大量冗余代码。

针对这个问题，我们采用的是 Webpack External 方式来解决。构建时，各个子系统会将公共依赖排除，主系统会打包一份包含所有这些公共依赖的 DLL 文件。子系统在运行时，直接从全局引用对应的依赖。如果子系统希望使用某些库的特定版本，也可以选择不排除这些依赖项。这在子系统希望升级某些依赖库的时候显得极为有用：通过子系统的局部升级，可以限制依赖库升级的影响范围，避免造成全局影响。

DLL 文件会包含大部分公共依赖，但有一个例外——我们不会将 Vue 打到 DLL 文件中。因为在实际开发中，很多库都喜欢向 Vue 的原型链上挂载方法和属性。如果不同团队开发时挂载的内容恰好用到同一个字段，就会带来不可预知的影响。

模块复用

除了底层的依赖，我们还需要考虑对公共的业务模块和工具函数进行复用。在企管平台，我们为公共业务组件库和公共函数库创建独立的 Git 工程，然后将所有的子系统和公共模块通过 Git Submodule 的方式引入到主系统的工程中。主系统采用 Lerna 的方式组织代码，各个子系统在开发时，可以通过软链直接引用到本地公共模块的代码，实现公共模块的复用。当公共模块发生更新，直接调用 Lerna Publish 就可以同时更新所有子系统 package.json 中依赖版本。

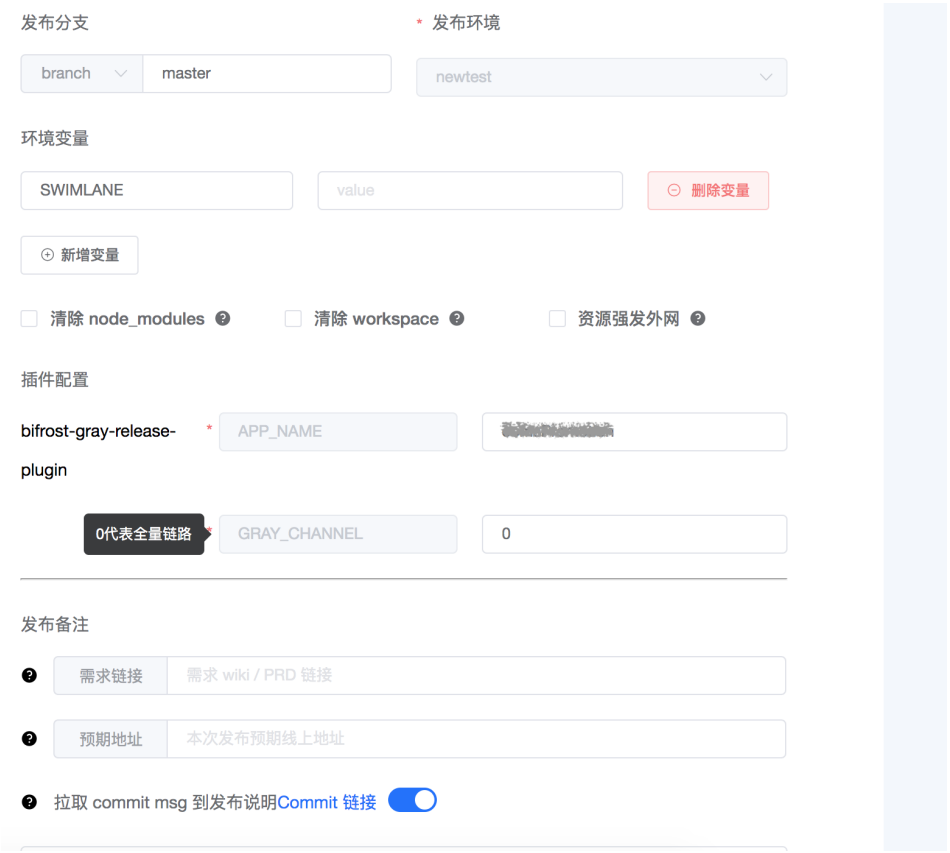
发布及部署流程

前面提到，主系统采用的是 JSONP 方式加载子系统的配置文件，整个发布过程都只需要发布静态资源，因此，Talos（美团内部自研的持续集成平台）提供的前端静态资源发布的能力就可以满足我们的需求。每次发布时，只需要构建有更新的项目，并将打包后的静态资源上传到 CDN 即可。

版本控制

采用微前端架构还有一个额外的好处：在 Nginx 和实际的业务层之间，多了一层主系统，我们可以像客户端一样，动态决定需要加载的子系统版本。基于此，我们实现了子系统的版本控制和定向灰度功能。发布时，我们通过参数确定本次发布是否是

灰度版本。在发布成功后，会记录本次发布的灰度信息、版本和配置文件 URL 等信息。



主系统每次启动时，首先会调用接口确定当前用户所处的链路（全量 / 灰度），再根据链路信息加载相应的子系统。我们记录了每次发布的资源 URL，所以也支持子系统的版本切换。只需要在版本服务中修改各条链路上需要激活的子系统版本，就可以轻松实现子系统版本切换。

埋点及错误上报

这里我们主要讨论 Bifrost 框架的埋点方案。在 Bifrost 项目中，可以借助主系统提供的一系列钩子函数实现针对子系统的埋点，包括：onAppLoading、onAppLoaded、onAppRouting、onError。每当子系统发生切换都会调用 onAppRouting 函数，因此我们可以在这里记录子系统加载的次数（PV）。onAppLoading 和 onAppLoaded 则会在子系统初次加载时调用，通过计算 Loading 和 Loaded 成功率的比值，我们可以得到子系统加载的成功率。子系统加载失败时，会调用 onError 函数，帮助排查子系统加载失败的原因。

收益

今年年初，我们对企管平台进行了微前端改造，目前系统已经在线上平稳运行半年时间，支持上百个零售商品牌，上千家门店业务的运转。

采用微前端架构，给我们项目带来的好处是显而易见的：

- 实现了异地合作开发时的完全解藕。采用微前端架构之后，两地团队在开发过程中再也没有遇到代码冲突的问题。
- 避免了单页应用发展成“巨石”应用。目前，企管平台总共实现了上百个页面，采用微前端的方式进行划分后，每个子系统包含的页面都不超过三十个，子系统的可维护性得到大大提高。
- 今年企管平台经历了两次大的组件库版本升级。第一次升级时，项目还是单页

应用，我们在暂停业务开发的基础上，耗费了大约一周的时间对所有的页面进行回归验证、完成升级。第二次升级时，我们已经完成了项目的微前端改造，可以通过增量的方式，先升级不常用的子系统，验证通过后再升级其他子系统。这样既不用中断正常的业务开发，也保证了依赖库升级时的影响范围和风险可控。

不是“银弹”

当然，同所有的架构方案一样，微前端这种模式也存在一些折衷和妥协。在获得低耦合和灵活性的同时，也引入了额外的复杂度。在微前端项目中，我们需要考虑多个工程的规范和代码质量的统一，需要引入更多的自动化工具来管理项目的发布部署流程，还需要处理多个前端工程运行在同一个域名下引起的 Cookie 覆盖等问题。

因此，在采用微前端架构之前，建议大家要谨慎的评估自己的项目是否真的适合采用微前端的方式，避免盲目引入微前端导致项目难以维护，得不偿失。

我们认为，如果项目中存在以下两个场景，比较适合采用微前端架构：

- 功能模块较多，且各个功能模块相对较为独立的中后台系统。
- 项目存在大量历史遗留问题，希望在保留已有功能的基础上，开发新的功能模块。

其他大部分项目都可以通过调整代码结构，构建单页应用，甚至采用最传统的多页应用等方式来进行优化、调整，从来达到降低耦合的目的。微前端并不是“银弹”。

期许

- 从去年 12 月立项至今，Bifrost 经历了近一年的迭代，发布了 2 个大版本和 38 个小版本。诞生之初，Bifrost 仅仅是针对企管平台这个特定业务场景的微前端方案。如今，已进化为面向 Vue 技术栈的通用微前端框架。期间，我们围绕 Bifrost，逐步完善了整个微前端技术体系的建设，实现了 Bifrost 主 / 子系统的脚手架工程和命令行工具、子系统的管理平台、灰度发布功能等一系列

平台和工具，完成了 Bifrost 微前端生态的雏形。

当然，Bifrost 依然还有很多可以提升的地方。未来，我们将会从以下几个方面进一步完善 Bifrost：

- 提供更加完善的前端微服务治理工具。
- 实现 JS 和 CSS 沙盒。
- 支持更多的技术栈。

结语

随着前端工程的日益复杂，我们对可扩展的前端架构的诉求也变得更加强烈。微前端作为一种前端解藕的方案，自然更加频繁地被大家所提及和应用。另一方面，虽然网上已经有了很多关于微前端的讨论，但依然缺乏真正落地到生产环境的案例。因此，我们希望通过对闪购团队近半年在微前端方案上的实践分享，帮助大家对微前端从概念到应用有一个更加清晰的认识，也期待与大家一起交流，碰撞出更多的火花。

作者简介

雨甫，美团闪购前端研发工程师。

招聘

美团闪购是美团点评旗下的零售到家业务，闪购专注于为消费者提供丰富、便捷的零售品类选择和及时配送服务，为零售商家提供线上、线下的整体解决方案，助力商家提升经营效率。
用户通过手机下单即可快速买到周边各类商家提供的丰富商品，涵盖食材生鲜、超市便利、鲜花绿植、母婴用品和健康护理等众多品类。美团闪购与美团外卖共享配送网络，平均 30 分钟配送上门，24 小时无间断配送，打造全品类一站式零售到家平台。
美团闪购前端团队诚招高级前端开发、前端开发专家。欢迎各位大佬的加入，共同打造极致的 LBS 电商体验。感兴趣同学可投递简历至：tech@meituan.com（邮件标题注明：美团闪购前端团队）

基本功

Litho 的使用及原理剖析

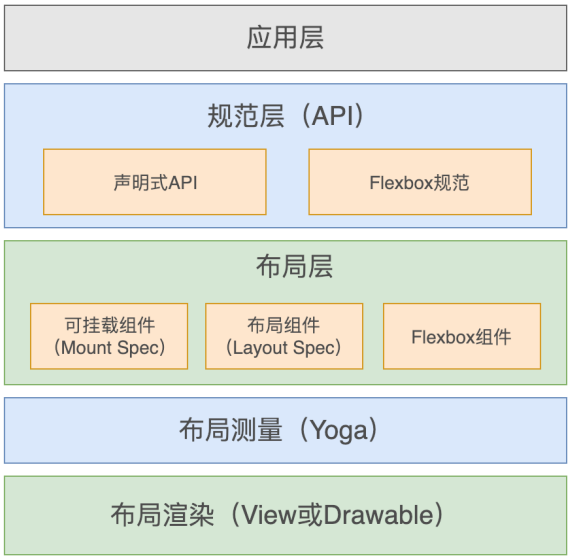
少宽　张颖

1. 什么是 Litho？

Litho 是 Facebook 推出的一套高效构建 Android UI 的声明式框架，主要目的是提升 RecyclerView 复杂列表的滑动性能和降低内存占用。下面是 Litho 官网的介绍：

> Litho is a declarative framework for building efficient user interfaces (UI) on Android. It allows you to write highly-optimized Android views through a simple functional API based on Java annotations. It was primarily built to implement complex scrollable UIs based on RecyclerView. With Litho, you build your UI in terms of components instead of interacting directly with traditional Android views. A component is essentially a function that takes immutable inputs, called props, and returns a component hierarchy describing your user interface.
>
> Litho 是高效构建 Android UI 的声明式框架，通过注解 API 创建高优的 Android 视图，非常适用于基于 Recyclerview 的复杂滚动列表。Litho 使用一系列组件构建视图，代替了 Android 传统视图交互方式。组件本质上是一个函数，它接受名为 Props 的不可变输入，并返回描述用户界面的组件层次结构。

Litho 是一套完全不同于传统 Android 的 UI 框架，它继承了 Facebook 一向大胆创新的风格，突破性地在 Android 上实现了 React 风格的 UI 框架。架构图如下：



应用层：上层 Android 应用接入层。

规范层（API）：允许用户使用声明式的 API（注解）来构建符合 Flexbox 规范的布局。

布局层：Litho 使用可挂载组件、布局组件和 Flexbox 组件来构建布局，其中可挂载组件和布局组件允许用户使用规范来定义，各个组件的具体用法下面的组件规范中会详细介绍。在 Litho 中每一个组件都是一个独立的功能模块。Litho 的组件和 React 的组件相类似，也具有属性和状态的概念，通过状态的变更来控制组件的展示样式。

布局测量：Litho 使用 Yoga 来完成组件布局的异步或同步（可根据场景定制）测量和计算，实现了布局的扁平化。

布局渲染：Litho 不仅支持使用 View 来渲染视图，还可以使用更轻量的 Drawable 来渲染视图。Litho 实现了大量使用 Drawable 来渲染的基础组件，可以进一步拍平布局。

除了上面提到的扁平化布局，Litho 还实现了布局的细粒度复用和异步计算布局的能力，对于这些功能的实现在 Litho 的特性及原理剖析中详细介绍。下面先介绍一

下大家比较关心的 Litho 使用方法。

2. Litho 的使用

Litho 的使用方式相比于传统的 Android 来说有些另类，它抛弃了通过 XML 定义布局的方式，采用声明式的组件在 Java 中构建布局。

Litho 和原生 Android 在使用上的区别

Android 传统布局：首先在资源文件 res/layout 目录下定义布局文件 xx.xml，然后在 Activity 或 Fragment 中引用布局文件生成视图，示例如下：

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="Hello World"
  android:textAlignment="center"
  android:textColor="#666666"
  android:textSize="40dp" />
```

```
public class MainActivity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.helloworld);
  }
}
```

Litho 布局：Litho 抛弃了 Android 原生的布局方式，通过组件方式构建布局生成视图，示例如下：

```
public class MainActivity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
      ComponentContext context = new ComponentContext(this);
      final Text.Builder builder = Text.create(context);
      final Component = builder.text("Hello World")
                .textSizeDip(40)
                .textColor(Color.parseColor("#666666"))
                .textAlignment(Layout.Alignment.ALIGN_CENTER)
```

```
                .build();
        LithoView view = LithoView.create(context, component);
        setContentView(view);
    }
}
```

Litho 自定义视图

Litho 中的视图单元叫做 Component，可以直观的翻译为“组件”，它的设计理念来自于 React 组件化的思想。每个组件持有描述一个视图单元所必须的属性和状态，用于视图布局的计算工作。视图最终的绘制工作是由组件指定的绘制单元（View 或者 Drawable）来完成的。

Litho 组件的创建方式也和原生 View 的创建方式有着很大的区别。Litho 使用注解定义了一系列的规范，我们需要使用 Litho 的注解来定义自己的组件生成规则，最终由 Litho 在编译期自动编译生成真正的组件。

组件规范

Litho 提供了两种类型的组件规范，分别是 Layout Spec 规范和 Mount Spec 规范。下面分别介绍两种规范的使用方式：

Layout Spec 规范：用于生成布局类型组件的规范，布局组件在逻辑上等同于 Android 中的 ViewGroup，用于组织其他组件构成一个布局。它要求我们必须使用 @LayoutSpec 注解来注明，并实现一个标注了 @OnCreateLayout 注解的方法。示例如下：

```
@LayoutSpec
class HelloComponentSpec {
  @OnCreateLayout
  static Component onCreateLayout(ComponentContext c, @Prop String name) {
    return Column.create(c)
      .child(Text.create(c)
        .text("Hello, " + name)
        .textSizeRes(R.dimen.my_text_size)
        .textColor(Color.BLACK)
        .paddingDip(ALL, 10)
        .build())
      .child(Image.create(c)
        .drawableRes(R.drawable.welcome)
```

```
            .scaleType(ImageView.ScaleType.CENTER_CROP)
            .build())
          .build();
    }
  }
```

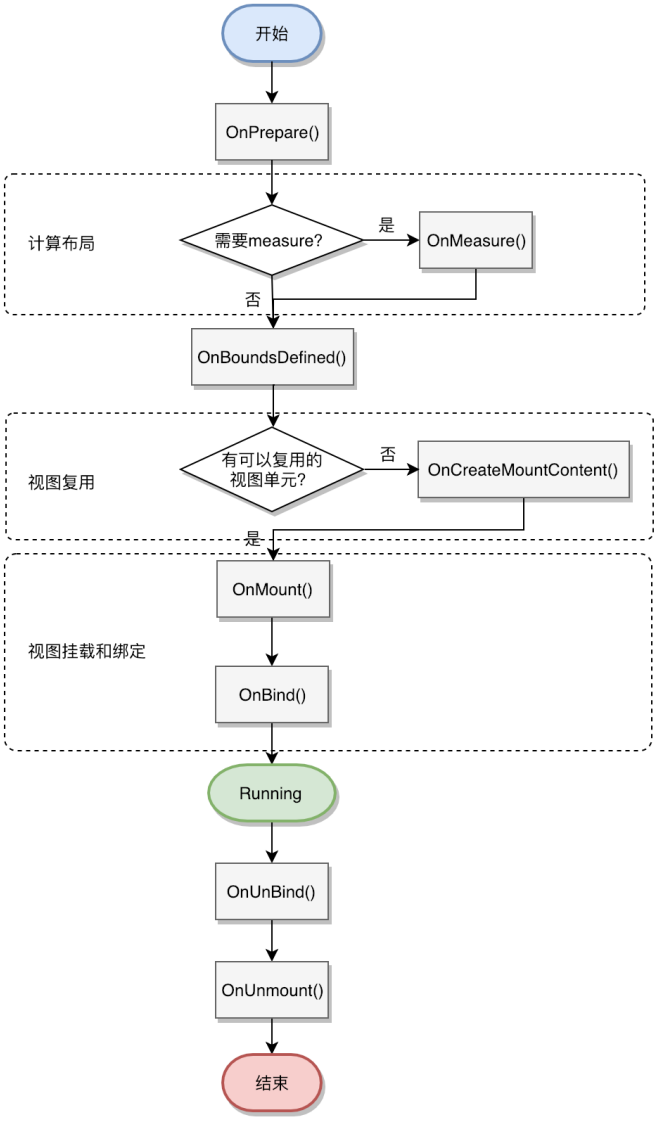
最终 Litho 会在编译时生成一个名为 HelloComponent 的组件。

```
  public final class HelloComponent extends Component {

    @Prop(resType = ResType.NONE,optional = false) String name;

    private HelloComponent() {
      super();
    }

    @Override
    protected Component onCreateLayout(ComponentContext c) {
      return (Component) HelloComponentSpec.
onCreateLayout((ComponentContext) c, (String)
name);
    }

    ...

    public static Builder create(ComponentContext context, int
defStyleAttr, int defStyleRes) {
      Builder builder = sBuilderPool.acquire();
      if (builder == null) {
        builder = new Builder();
      }
    HelloComponent instance = new HelloComponent();
      builder.init(context, defStyleAttr, defStyleRes, instance);
      return builder;
    }

    public static class Builder extends Component.Builder<Builder> {
      private static final String[] REQUIRED_PROPS_NAMES = new String[]
{"name"};
      private static final int REQUIRED_PROPS_COUNT = 1;
      HelloComponent mHelloComponent;

      ...

      public Builder name(String name) {
       this.mHelloComponent.name = name;
       mRequired.set(0);
```

```
        return this;
      }

      @Override
      public HelloComponent build() {
        checkArgs(REQUIRED_PROPS_COUNT, mRequired, REQUIRED_PROPS_NAMES);
        HelloComponent helloComponentRef = mHelloComponent;
        release();
        return helloComponentRef;
      }
    }
  }
```

Mount Spec 规范：用来生成可挂载类型组件的规范，用来生成渲染具体 View 或者 Drawable 的组件。同样，它必须使用 @MountSpec 注解来标注，并至少实现一个标注了 @onCreateMountContent 的方法。Mount Spec 相比于 Layout Spec 更复杂一些，它拥有自己的生命周期：

- @OnPrepare，准备阶段，进行一些初始化操作。
- @OnMeasure，负责布局的计算。
- @OnBoundsDefined，在布局计算完成后挂载视图前做一些操作。
- @OnCreateMountContent，创建需要挂载的视图。
- @OnMount，挂载视图，完成布局相关的设置。
- @OnBind，绑定视图，完成数据和视图的绑定。
- @OnUnBind，解绑视图，主要用于重置视图的数据相关的属性，防止出现复用问题。
- @OnUnmount，卸载视图，主要用于重置视图的布局相关的属性，防止出现复用问题。



除了上述两种组件类型，Litho 中还有一种特殊的组件——Layout，它不能使用规范来生成。Layout 是 Litho 中的容器组件，类似于 Android 中的 ViewGroup，但是只能使用 Flexbox 的规范。它可以包含子组件节点，是 Litho 各组件连接的纽带。Layout 组件只是 Yoga 在 Litho 中的代理，组件的所有布局相关的属性都会直接设置给 Yoga，并由 Yoga 完成布局的计算。Litho 实现了两个 Layout 组件 Row 和

Column，分别对应 Flexbox 中的行和列。

Litho 的属性

在 Litho 中属性分为两种，不可变属性称为 Props，可变属性称为 State，下面分别介绍一下两种属性：

Props 属性：组件中使用 @Prop 注解标注的参数集合，具有单向性和不可变性。下面通过一个简单的例子了解一下如何在组件中定义和使用 Props 属性：

```
@MountSpec
class MyComponentSpec {

    @OnPrepare
    static void onPrepare(
      ComponentContext c,
      @Prop(optional = true) String prop1) {
      ...
    }

    @OnMount
    static void onMount(
      ComponentContext c,
      SomeDrawable convertDrawable,
      @Prop(optional = true) String prop1,
      @Prop int prop2) {
        if (prop1 != null) {
        ...
        }
    }
}
```

在上面的代码中，共使用了三次 Prop 注解，分别标注 prop1 和 prop2 两个变量，即定义了 prop1 和 prop2 两个属性。Litho 会在自动编译生成的 MyComponent 类的 Builder 类中生成这两个属性的同名方法。按照如下代码，便可以去使用上面定义的属性：

```
MyComponent.create(c)
  .prop1("My prop 1")
  .prop2(256)
  .build();
```

State 属性：意为“状态”属性，State 属性虽然可变，但是其变化由组件内部控制，例如：输入框、Checkbox 等都是由组件内部去感知用户行为，并更新组件的 State 属性。所以一个组件一旦创建，我们便无法通过任何外部设置去更改它的属性。组件的 State 属性虽然不允许像 Props 属性那样去显式设置，但是我们可以定义一个单独的 Props 属性来当做某个 State 属性的初始值。

3. Litho 的特性及原理剖析

Litho 官网首页通过 4 个段落重点介绍了 Litho 的 4 个特性。

声明式组件

Litho 采用声明式的 API 来定义 UI 组件，组件通过一组不可变的属性来描述 UI。这种组件化的思想灵感来源于 React，关于声明式组件的用法上面已经详细介绍过了。

传统 Android 布局因为 UI 与逻辑分离，所以开发工具都有强大的预览功能，方便开发者调整布局。而 Litho 采用 React 组件化的思想，通过组件连接了逻辑与布局 UI，虽然 Litho 也提供了对 Stetho 的支持，借助于 Chrome 开发者工具对界面进行调试，不过使用起来并没有那么方便。

异步布局

Android 系统在绘制时为了防止页面错乱，页面所有 View 的测量 (Measure)、布局 (Layout) 以及绘制 (Draw) 都是在 UI 线程中完成的。当页面 UI 非常复杂、视图层级较深时，难免 Measure 和 Layout 的时间会过长，从而导致页面渲染时候丢帧出现卡顿情况。Litho 为解决该问题，提出了异步布局的思想，利用 CPU 的闲置时间提前在异步线程中完成 Measure 和 Layout 的过程，仅在 UI 线程中完成绘制工作。当然，Litho 只是提供了异步布局的能力，它主要使用在 RecyclerView 等可以提前知道下一个视图长什么样子的场景。

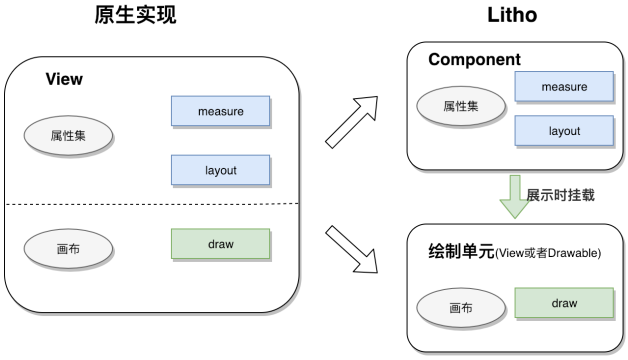
异步布局原理剖析

针对 RecyclerView 等滑动列表，由于可以提前知道接下来要展示的一个甚至多

个条目的视图样式，所以只要提前创建好下一个或多个条目的视图，就可以提前完成视图的布局工作。

那么 Android 原生为什么不支持异步布局呢？主要有以下两个原因：

- View 的属性是可变的，只要属性发生变化就可能导致布局变化，因此需要重新计算布局，那么提前计算布局的意义就不大了。而 Litho 组件的属性是不可变的，所以对于一个组件来说，它的布局计算结果是唯一且不变的。
- 提前异步布局就意味着要提前创建好接下来要用到的一个或者多个条目的视图，而 Android 原生的 View 作为视图单元，不仅包含一个视图的所有属性，而且还负责视图的绘制工作。如果要在绘制前提前去计算布局，就需要预先去持有大量未展示的 View 实例，大大增加内存占用。反观 Litho 的组件则没有这个问题，Litho 的组件只是视图属性的一个集合，仅负责计算布局，绘制工作由指定的绘制单元来完成，相比与传统的 View 显然 Litho 的组件要轻量的多。所以在 Litho 中，提前创建好接下来要用到的多个条目的组件，并不会带来性能问题，甚至还可以直接把组件当成滑动列表的数据源。如下图所示：



扁平化的视图

使用 Litho 布局，我们可以得到一个极致扁平的视图效果。它可以减少渲染时的递归调用，加快渲染速度。

下面是同一个视图在 Android 和 Litho 实现下的视图层级效果对比。可以看到，

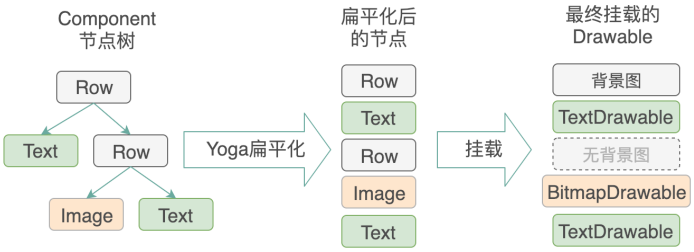
同样的样式，使用 Litho 实现的布局要比使用 Android 原生实现的布局更加扁平。

扁平化视图原理剖析

Litho 使用 Flexbox 来创建布局，最终生成带有层级结构的组件树。然后 Litho 对布局层级进行了两次优化。

- 使用了 Yoga 来进行布局计算，Yoga 会将 Flexbox 的相对布局转成绝对布局。经过 Yoga 处理后的布局没有了原来的布局层级，变成了只有一层。虽然不能解决过度绘制的问题，但是可以有效地减少渲染时的递归调用。
- 前面介绍过 Litho 的视图渲染由绘制单元来完成，绘制单元可以是 View 或者更加轻量的 Drawable，Litho 自己实现了一系列挂载 Drawable 的基本视图组件。通过使用 Drawable 可以减少内存占用，同时相比于 View，Android 无法检查出 Drawable 的视图层级，这样可以使视图效果看起来更加扁平。

原理如下图所示，Litho 会先把组件树拍平成没有层级的列表，然后使用 Drawable 来绘制对应的视图单元。



Litho 使用 Drawable 代替 View 能带来多少好处呢？ Drawable 和 View 的区别在于前者不能和用户交互，只能展示，因此 Drawable 不会像 View 那样持有很多变量和引用，所以 Drawable 比 View 从内存上看要轻量很多。举个例子：50 个同样展示“Hello world”的 TextView 和 TextDrawable 在内存占比上，前者几乎是后者的 8 倍。对比图如下，Shallow Size 表示对象自身占用的内存大小。

Class Name	Allocations	Deallocations	Total Count	Shallow Size
TextView *(android.widget)*	0	0	50	40,450
TextTrieMap$Node *(android.icu.impl)*	0	0	1	24
TextTrieMap *(android.icu.impl)*	0	0	1	13
TextPaint *(android.text)*	0	0	104	14,144
TextLine$DecorationInfo *(android.text)*	0	0	1	26
TextLine *(android.text)*	0	0	1	78
TextLayoutBuilder$Params *(com.facebook.fbui.textl*	0	0	1	95
TextLayoutBuilder *(com.facebook.fbui.textlayoutbui*	0	0	1	38
TextDrawable *(com.facebook.litho.widget)*	0	0	50	5,450

绘制单元的降级策略

由于 Drawable 不具有交互能力，所以对于使用 Drawable 无法实现的交互场景，Litho 会自动降级成 View。主要有以下几种场景：

- 有监听点击事件。
- 限制子视图绘出父布局。
- 有监听焦点变化。
- 有设置 Tag。
- 有监听触摸事件。
- 有光影效果。

对于以上场景的使用请仔细考虑，过多的使用会导致 Litho 的层级优化效果变差。

对比 Android 的约束布局

为了解决布局嵌套问题，Android 推出了约束布局（ConstraintLayout），使用约束布局也可以达到扁平化视图的目的，那么使用 Litho 的好处是什么呢?

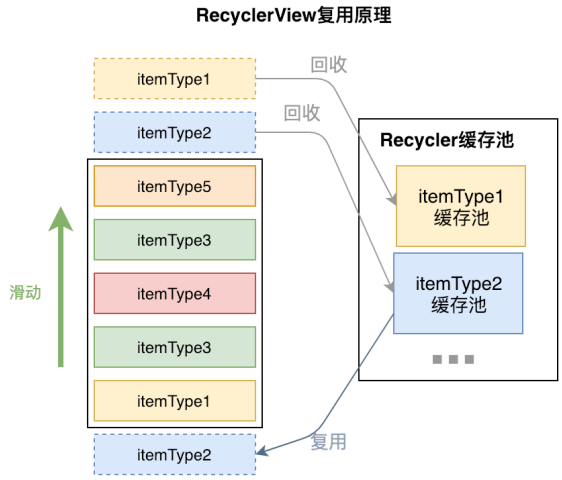
Litho 可以更好地实现复杂布局。约束布局虽然可以实现扁平效果，但是它使用了大量的约束来固定视图的位置。随着布局复杂程度的增加，约束条件变得越来越多，可读性也变得越来越差。而 Litho 则是对 Flexbox 布局进行的扁平化处理，所以实际使用的还是 Flexbox 布局，对于复杂的布局 Flexbox 布局可读性更高。

细粒度的复用

Litho 中的所有组件都可以被回收，并在任何位置进行复用。这种细粒度的复用方式可以极大地提高内存使用率，尤其适用于复杂滑动列表，内存优化非常明显。

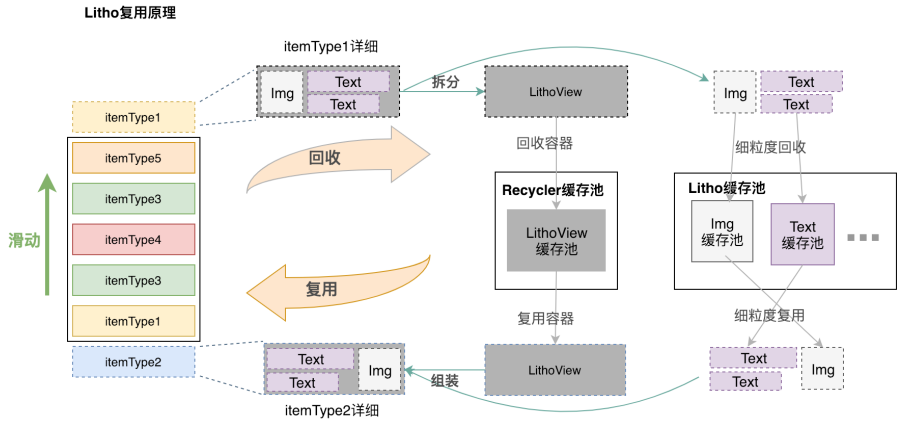
原生 RecyclerView 复用原理剖析

原生的 RecyclerView 视图按模板类型进行存储并复用，也就是说模板类型越多，所需存储的模板种类也就越多，导致内存占用越来越大。原理如下图。滑出屏幕的 itemType1 和 itemType2 都会在 Recycler 缓存池保存，等待后面滑进屏幕的条目的复用。



细粒度复用优化内存原理剖析

在 Litho 中，item 在回收前，会把 LithoView 中挂载的各个绘制单元拆分出来（解绑），由 Litho 自己的缓存池去分类回收，在展示前由 LithoView 按照组件树的样式组装（挂载）各个绘制单元，这样就达到了细粒度复用的目的。原理如下图。滑出屏幕的 itemType1 会被拆分成一个个的视图单元。LithoView 容器由 Recycler 缓存池回收，其他视图单元由 Litho 的缓存池分类回收。



使用细粒度复用的 RecyclerView 的缓存池不再需要区分模板类型来缓存大量的视图模板，只需要缓存 LithoView 容器。细粒度回收的视图单元数量要远远小于原来缓存在各个视图模板中的视图单元数量。

实践

美团对 Litho 进行了二次开发，在美团的 MTFlexbox 动态化实现方案（简称动态布局）中把 Litho 作为底层 UI 渲染引擎来使用。通过动态布局的预览工具，为 Litho 提供实时预览能力，同时可以有效发挥 Litho 的性能优化效果。

目前 Litho+ 动态布局的实现方案已经应用在了美团 App 中，给美团 App 带来了不错的性能提升。后续博主会详细介绍 Litho+ 动态布局在美团性能优化的实践方案。

首页部分卡片　　搜索部分卡片

使用 Litho+ 动态布局实现的部分卡片

内存数据

由于 Litho 中使用了大量 Drawable 替换 View，并且实现了视图单元的细粒度复用，因此复杂列表滑动时内存优化比较明显。美团首页内存占用随滑动页数变化走势图如下。随着一页一页地滑动，内存优化了 30M 以上。（数据采集自 Vivo x20 手机内存占用情况）

FPS 数据

FPS 的提升主要得益于 Litho 的异步布局能力，提前计算布局可以减少滑动时的帧率波动，所以滑动过程较平稳，不会有高低起伏的卡顿感。（数据采集自魅蓝 2 手机一段时间内连续 fps 的波动情况）

5. 总结

Litho 相对于传统 Android 是颠覆式的，它采用了 React 的思路，使用声明式的 API 来编写 UI。相比于传统 Android，确实在性能优化上有很大的进步，但是如果完全使用 Litho 开发一款应用，需要自己实现很多组件，而 Litho 的组件需要在编译时生成，实时预览方面也有所欠缺。相对于直接使用 Litho 的高成本，把 Litho 封装成 Flexbox 布局的底层渲染引擎是个不错的选择。

6. 参考资料

1. Litho 官网

2. 说一说 Facebook 开源的 Litho
3. React 官网
4. Yoga 官网

7. 作者简介

何少宽，美团 Android 开发工程师，2015 年加入美团，负责美团平台终端业务研发工作。
张颖，美团 Android 开发工程师，2017 年加入美团，负责美团平台终端业务研发工作。

Android 兼容 Java 8 语法特性的原理分析

元合　朝旭

本文主要阐述了 Lambda 表达式及其底层实现（invokedynamic 指令）的原理、Android 第三方插件 RetroLambda 对其的支持过程、Android 官方最新的 dex 编译器 D8 对其的编译支持。通过对这三个方面的跟踪分析，以 Java 8 的代表性特性——Lambda 表达式为着眼点，将 Android 如何兼容 Java8 的过程分享给大家。

Java 8 概述

Java 8 是 Java 开发语言非常重要的一个版本。Oracle 从 2014 年 3 月 18 日发布 Java 8，从该版本起，Java 开始支持函数式编程。特别是吸收了运行在 JVM 上的 Scala、Groovy 等动态脚本语言的特性之后，Java 8 在语言的表达力、简洁性两个方面有了很大的提高。

Java 8 的主要语言特性改进概括起来包括以下几点：

- Lambda 表达（函数闭包）
- 函数式接口（@FunctionalInterface）
- Stream API（通过流式调用支持 map、filter 等高阶函数）
- 方法引用（使用 :: 关键字将函数转化为对象）
- 默认方法（抽象接口中允许存在 default 修饰的非抽象方法）
- 类型注解和重复注解

其中 Lambda 表达、函数式接口、方法引用三个特性为 Java 带来了函数式编程的风格；而 Stream 实现了 map、filter、reduce 等常见的高阶函数，数据源囊括了数组、集合、IO 通道等，这些又为 Java 带来了流式编程或者说链式编程的风格，以上这些风格让 Java 变得越来越现代化和易用。

Android 和 Java 关系

其实 Java 在 Android 的快速发展过程中扮演着非常重要的角色，无论是作为开发语言（Java）、开发 Framework（Android-SDK 引用了 80% 的 JDK-API），还是开发工具（Eclipse or Android Studio）。这些都和 Java 有着千丝万缕的关系。不过可能是受到与 Oracle 的法律诉讼的影响，Google 在 Android 上针对 Java 的升级一直都不是很积极：

- Android 从 1.0 一直升级到 4.4，迭代了将近 19 个 Android 版本，才在 4.4 版本中支持了 Java 7。
- 然后从 Android 4.4 版本开始算起，一直到 Android N(7.0) 共 4 个 Android 版本，才在 Jack/Jill 工具链勉强支持了 Java 8。但由于 Jack/Jill 工具链在构建流程中舍弃了原有 Java 字节码的体系，导致大量既有的技术沉淀无法应用，致使许多 App 工程放弃了接入。
- 最后直到 Android P(9.0) 版本，Google 才在 Android Studio 3.x 中通过新增的 D8 dex 编译器正式支持了 Java 8，但部分 API 并不能全版本支持。

可谓“历经坎坷”。特别是 Rx 大行其道的今天，Rx 配合 Java 8 特性 Lambda 带来简洁、高效的开发体验，更是让 Android Developer 望眼欲穿。

接下来，本文将从技术原理层面，来分析一下 Android 是如何支持 Java 8 的。

Lambda 表达式

想要更好的理解 Android 对 Java 8 的支持过程，Lambda 表达式这一代表性的“语法糖”是一个非常不错的切入点。所以，我们首先需要搞清楚 Lambda 表达式到底是什么？其底层的实现原理又是什么？

Lambda 表达式是 Java 支持函数式编程的基础，也可以称之为闭包。简单来说，就是在 Java 语法层面允许将函数当作方法的参数，函数可以当做对象。任一 Lambda 表达式都有且只有一个函数式接口与之对应，从这个角度来看，也可以说是该函数式接口的实例化。

Lambda 表达式

通用格式：

```
语法格式:
(parameters) -> expression  或者  (parameters) -> { statements; }

格式理解:
( 对应函数式接口的参数列表 ) -> { 对应函数接口的实现方法 }
```

简单范例：

```
package com.j8sample;

public class J8Sample {

    public static void main(String arg[]) {

        Runnable runnable = () -> System.out.println("xixi"); // Lambda表达式1
        new Thread(runnable).start();

        new Thread(() -> {
            System.out.println("haha"); // Lambda表达式2
        }).start();
    }
}
```

```
@FunctionalInterface
public interface Runnable {
    /**
     * When an object implementing interface <code>Runnable</code> is used
     * to create a thread, starting the thread causes the object's
     * <code>run</code> method to be called in that separately executing
     * thread.
     * <p>
     * The general contract of the method <code>run</code> is that it may
     * take any action whatsoever.
     *
     * @see     java.lang.Thread#run()
     */
    public abstract void run();
}
```

说明：

- Lambda 表达式中 () 对应的是函数式接口 –run 方法的参数列表。
- Lambda 表达式中 System.out.println(“xixi”) / System.out.println(“ha–ha”)，在运行时会是具体的 run 方法实现。

Lambda 表达式原理

针对实例中的代码，我们来看下编译之后的字节码：

```
javac J8Sample.java  ->  J8Sample.class

javap -c -p J8Sample.class
```

```
Compiled from "J8Sample.java"
public class com.j8sample.J8Sample {
  public com.j8sample.J8Sample();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: invokedynamic #2,  0              // InvokeDynamic #0:run:()Ljava/lang/Runnable;
       5: astore_1
       6: new           #3                  // class java/lang/Thread
       9: dup
      10: aload_1
      11: invokespecial #4                  // Method java/lang/Thread."<init>":(Ljava/lang/Runnable;)V
      14: invokevirtual #5                  // Method java/lang/Thread.start:()V
      17: new           #3                  // class java/lang/Thread
      20: dup
      21: invokedynamic #6,  0              // InvokeDynamic #1:run:()Ljava/lang/Runnable;
      26: invokespecial #4                  // Method java/lang/Thread."<init>":(Ljava/lang/Runnable;)V
      29: invokevirtual #5                  // Method java/lang/Thread.start:()V
      32: return

  private static void lambda$main$1();
    Code:
       0: getstatic     #7                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #8                  // String haha
       5: invokevirtual #9                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return

  private static void lambda$main$0();
    Code:
       0: getstatic     #7                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #10                 // String xixi
       5: invokevirtual #9                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
}
```

从字节码中我们可以看到：

- 实例中 Lambda 表达式 1 变成了字节码代码块中 Line 11 的 0: invokedynamic #2, 0 // InvokeDynamic #0:run:()Ljava/lang/Runnable。
- 实例中 Lambda 表达式 2 变成了字节码代码块中 Line 20 的 21: invokedynamic #6, 0 // InvokeDynamic #1:run:()Ljava/lang/Runnable。

可见，Lambda 表达式在虚拟机层面上，是通过一种名为 invokedynamic 字节码指令来实现的。那么 invokedynamic 又是何方神圣呢?

invokedynamic 指令解读

invokedynamic 指令是 Java 7 中新增的字节码调用指令，作为 Java 支持动态类

型语言的改进之一，跟 invokevirtual、invokestatic、invokeinterface、invokespecial 四大指令一起构成了虚拟机层面各种 Java 方法的分配调用指令集。区别在于：

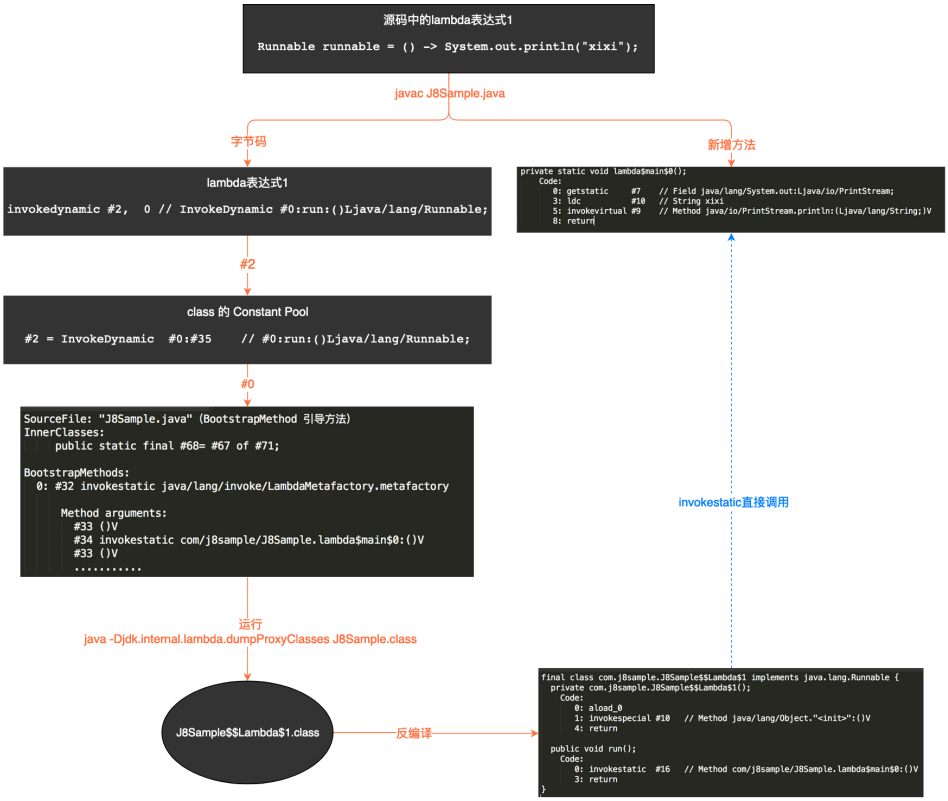
- 后四种指令，在编译期间生成的 class 文件中，通过常量池 (Constant Pool) 的 MethodRef 常量已经固定了目标方法的符号信息（方法所属者及其类型，方法名字、参数顺序和类型、返回值）。虚拟机使用符号信息能直接解释出具体的方法，直接调用。
- 而 invokedynamic 指令在编译期间生成的 class 文件中，对应常量池 (Constant Pool) 的 Invokedynamic_Info 常量存储的符号信息中并没有方法所属者及其类型，替代的是 BootstapMethod 信息。在运行时，通过引导方法 BootstrapMethod 机制动态确定方法的所属者和类型。这一特点也非常契合动态类型语言只有在运行期间才能确定类型的特征。

那么，invokedynamic 如何通过引导方法找到所属者及其类型？我们依然结合前面的 J8Sample 实例：

```
javap -v J8Sample.class
```

```
Constant pool:  (class 常量池)
   #1 = Methodref          #12.#30        // java/lang/Object."<init>":()V
   #2 = InvokeDynamic      #0:#35         // #0:run:()Ljava/lang/Runnable;
   #3 = Class              #36            // java/lang/Thread
   #4 = Methodref          #3.#37         // java/lang/Thread."<init>":(Ljava/lang/Runnable;)V
   #5 = Methodref          #3.#38         // java/lang/Thread.start:()V
   #6 = InvokeDynamic      #1:#35         // #1:run:()Ljava/lang/Runnable;
   #7 = Fieldref           #40.#41        // java/lang/System.out:Ljava/io/PrintStream;
   #8 = String             #42            // haha
   #9 = Methodref          #43.#44        // java/io/PrintStream.println:(Ljava/lang/String;)V
  #10 = String             #45            // xixi
  #11 = Class              #46            // com/j8sample/J8Sample
  #12 = Class              #47            // java/lang/Object
  #13 = Utf8               <init>
  ..........

SourceFile: "J8Sample.java" (BootstrapMethod 引导方法)
InnerClasses:
     public static final #68= #67 of #71;
    //Lookup=class java/lang/invoke/MethodHandles$Lookup of class java/lang/invoke/MethodHandles
BootstrapMethods:
  0: #32 invokestatic java/lang/invoke/LambdaMetafactory.metafactory:(Ljava/lang/invoke/MethodHandles$Lookup;
                                                                      Ljava/lang/String;
                                                                      Ljava/lang/invoke/MethodType;
                                                                      Ljava/lang/invoke/MethodType;
                                                                      Ljava/lang/invoke/MethodHandle;
                                                                      Ljava/lang/invoke/MethodType;)
                                                                      Ljava/lang/invoke/CallSite;
      Method arguments:
        #33 ()V
        #34 invokestatic com/j8sample/J8Sample.lambda$main$0:()V
        #33 ()V
        ...........
```

结合 J8Sample.class 字节码，并对 invokedynamic 指令调用过程进行跟踪分析。总结如下：



依据上图 invokedynamic 调用步骤，我们一步一步做一个分析讲解。

步骤 1　选取 J8Sample.java 源码中 Lambda 表达式 1：

Runnable runnable = () -> System.out.println("xixi"); // lambda 表达式 1

步骤 2　通过 javac J8Sample.java 编译得到 J8Sample.class 之后，

Lambda 表达式 1 变成：0: invokedynamic #2, 0 // InvokeDynamic #0:run:()Ljava/lang/Runnable;

对应在 J8Sample.class 中发现了新增的私有静态方法：

```
private static void lambda$main$0();
  Code:
     0: getstatic     #7                  // Field java/lang/System.out:Ljava/io/PrintStream;
     3: ldc           #10                 // String xixi
     5: invokevirtual #9                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
     8: return
```

步骤 3　针对表达式 1 的字节码分析 #2 对应的是 class 文件中的常量池：

#2 = InvokeDynamic #0:#35 // #0:run:()Ljava/lang/Runnable;

注意，这里 InvokeDynamic 不是指令，代表的是 Constant_InvokeDynamic_Info 结构。

步骤 4　结构后面紧跟的 #0 标识的是 class 文件中的 BootstrapMethod 区域中引导方法的索引：

```
BootstrapMethods:
  0: #32 invokestatic java/lang/invoke/LambdaMetafactory.metafactory:(Ljava/lang/invoke/MethodHandles$Lookup;
                                                                      Ljava/lang/String;
                                                                      Ljava/lang/invoke/MethodType;
                                                                      Ljava/lang/invoke/MethodType;
                                                                      Ljava/lang/invoke/MethodHandle;
                                                                      Ljava/lang/invoke/MethodType;)Ljava/lang/invoke/CallSite;
    Method arguments:
      #33 ()V
      #34 invokestatic com/j8sample/J8Sample.lambda$main$0:()V
      #33 ()V
```

步骤 5　引导方法中的 java/lang/invoke/LambdaMetafactory.metafactory 才是 invokedynamic 指令的关键：

```
public static CallSite metafactory(MethodHandles.Lookup caller,
                                   String invokedName,
                                   MethodType invokedType,
                                   MethodType samMethodType,
                                   MethodHandle implMethod,
                                   MethodType instantiatedMethodType)
        throws LambdaConversionException {
    AbstractValidatingLambdaMetafactory mf;
    mf = new InnerClassLambdaMetafactory(caller, invokedType,
                                         invokedName, samMethodType,
                                         implMethod, instantiatedMethodType,
                                          isSerializable: false, EMPTY_CLASS_ARRAY, EMPTY_MT_ARRAY);
    mf.validateMetafactoryArgs();
    return mf.buildCallSite();
}
```

```
public InnerClassLambdaMetafactory(MethodHandles.Lookup caller,
                                   MethodType invokedType,
                                   String samMethodName,
                                   MethodType samMethodType,
                                   MethodHandle implMethod,
                                   MethodType instantiatedMethodType,
                                   boolean isSerializable,
                                   Class<?>[] markerInterfaces,
                                   MethodType[] additionalBridges)
        throws LambdaConversionException {
    super(caller, invokedType, samMethodName, samMethodType,
          implMethod, instantiatedMethodType,
          isSerializable, markerInterfaces, additionalBridges);
    implMethodClassName = implDefiningClass.getName().replace( oldChar: '.', newChar: '/');
    implMethodName = implInfo.getName();
    implMethodDesc = implMethodType.toMethodDescriptorString();
    implMethodReturnClass = (implKind == MethodHandleInfo.REF_newInvokeSpecial)
            ? implDefiningClass
            : implMethodType.returnType();
    constructorType = invokedType.changeReturnType(Void.TYPE);
    lambdaClassName = targetClass.getName().replace( oldChar: '.', newChar: '/') + "$$Lambda$" + counter.incrementAndGet();
    cw = new ClassWriter(ClassWriter.COMPUTE_MAXS);
    int parameterCount = invokedType.parameterCount();
    if (parameterCount > 0) {
        argNames = new String[parameterCount];
        argDescs = new String[parameterCount];
        for (int i = 0; i < parameterCount; i++) {
            argNames[i] = "arg$" + (i + 1);
            argDescs[i] = BytecodeDescriptor.unparse(invokedType.parameterType(i));
        }
    } else {
        argNames = argDescs = EMPTY_STRING_ARRAY;
    }
}
```

该方法会在运行时，在内存中动态生成一个实现 Lambda 表达式对应函数式接口的实例类型，并在接口的实现方法中调用步骤 2 中新增的静态私有方法。

步骤 6　使用 java -Djdk.internal.lambda.dumpProxyClasses J8Sample.class 运行一下，可以内存中动态生成的类型输出到本地：

```
public InnerClassLambdaMetafactory(MethodHandles.Lookup caller,
                                   MethodType invokedType,
                                   String samMethodName,
                                   MethodType samMethodType,
                                   MethodHandle implMethod,
                                   MethodType instantiatedMethodType,
                                   boolean isSerializable,
                                   Class<?>[] markerInterfaces,
                                   MethodType[] additionalBridges)
        throws LambdaConversionException {
    super(caller, invokedType, samMethodName, samMethodType,
          implMethod, instantiatedMethodType,
          isSerializable, markerInterfaces, additionalBridges);
    implMethodClassName = implDefiningClass.getName().replace( oldChar: '.', newChar: '/');
    implMethodName = implInfo.getName();
    implMethodDesc = implMethodType.toMethodDescriptorString();
    implMethodReturnClass = (implKind == MethodHandleInfo.REF_newInvokeSpecial)
            ? implDefiningClass
            : implMethodType.returnType();
    constructorType = invokedType.changeReturnType(Void.TYPE);
    lambdaClassName = targetClass.getName().replace( oldChar: '.', newChar: '/') + "$$Lambda$" + counter.incrementAndGet();
    cw = new ClassWriter(ClassWriter.COMPUTE_MAXS);
    int parameterCount = invokedType.parameterCount();
    if (parameterCount > 0) {
        argNames = new String[parameterCount];
        argDescs = new String[parameterCount];
        for (int i = 0; i < parameterCount; i++) {
            argNames[i] = "arg$" + (i + 1);
            argDescs[i] = BytecodeDescriptor.unparse(invokedType.parameterType(i));
        }
    } else {
        argNames = argDescs = EMPTY_STRING_ARRAY;
    }
}
```

步骤 7　通过 javap -p -c J8Sample\$\$Lambda\$1.class 反编译一下，可以看到生成类的实现：

```
final class com.j8sample.J8Sample$$Lambda$1 implements java.lang.Runnable {
  private com.j8sample.J8Sample$$Lambda$1();
    Code:
       0: aload_0
       1: invokespecial #10                 // Method java/lang/Object."<init>":()V
       4: return

  public void run();
    Code:
       0: invokestatic  #16                 // Method com/j8sample/J8Sample.lambda$main$0:()V
       3: return
}
```

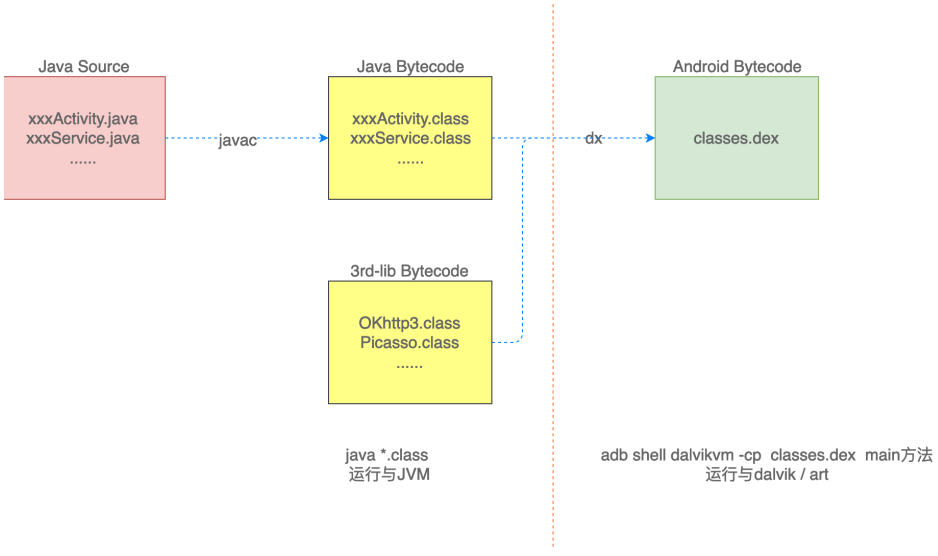
在 run 方法中使用了 invokestatic 指令，直接调用了 J8Sample.lambda$-main$0 这个在编译期间生成的静态私有方法。

至此，上面 7 个步骤就是 Lambda 表达式在 Java 的底层的实现原理。Android 针对这些实现会怎么处理呢?

Android 不能直接支持

回到 Android 系统上，Java-Bytecode（JVM 字节码）是不能直接运行在 Android 系统上的，需要转换成 Android-Bytecode（Dalvik/ART 字节码）。

如图：

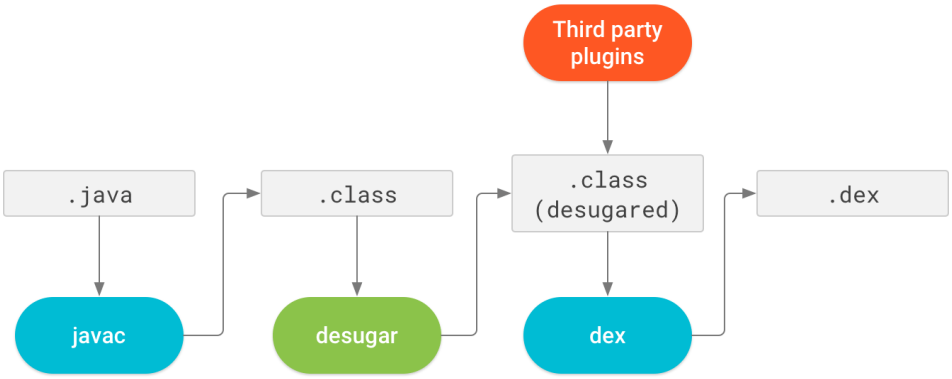


通过 Lambda 这节，我们知道 Java 底层是通过 invokedynamic 指令来实现，由于 Dalvik/ART 并没有支持 invokedynamic 指令或者对应的替代功能。简单的来说，就是 Android 的 dex 编译器不支持 invokedynamic 指令，导致 Android 不能直接支持 Java 8。

Android 间接支持

既然不能直接支持，那就只能在 Java-Bytecode 转换到 Android-Bytecode 这一过程中想办法，间接支持。这个间接支持的过程我们统称为 Desugar（脱糖）过程。

官方流程图：

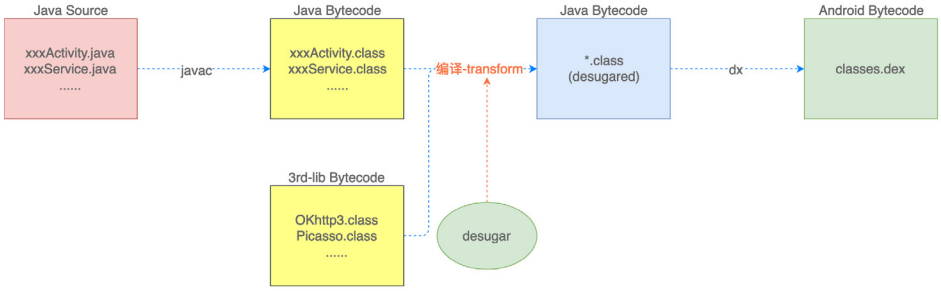


当前，无论是 RetroLambda，还是 Google 的 Jack & Jill 工具，还是最新的 D8 dex 编译器：

- 流程方面：都是按照如上图所示的官方流程进行 Desugar 的。
- 原理方面：却是参照 Lambda 在 Java 底层的实现，并将这些实现移至到 RetroLambda 插件或者 Jack、D8 编译器工具中。

下面我们逐个分析解读一下。

Android 间接支持之 RetroLambda



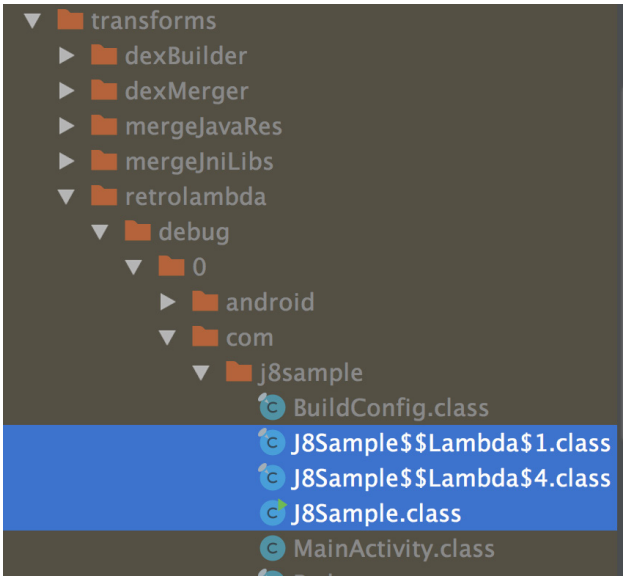
如图所示，RetroLambda 的 Desugar 过程发生在 javac 将源码编译完成之后，dx 工具进行 dex 编译之前。

RetroLambda Desugar

其实就是参照 invokedynamic 指令解读一节中的步骤 5，根据 java/lang/invoke/LambdaMetafactory.metafactory 方法，直接将原本在运行时生成在内存中的 J8Sample\$\$Lambda\$1.class，在 javac 编译结束之后，dx 编译 dex 之前，直接生成到本地，并使用生成的 J8Sample\$\$Lambda\$1 类修改 J8Sample.class 字节码文件，将 J8Sample.class 中的 invokedynamic 指令替换成 invokestatic 指令。

将实例中的 J8Sample.java 放到一个配置了 Retrolambda 的 Android 工程中：

AndroidStudio -> Build -> make project 编译之后：



app:transformClassesWithRetrolambdaForDebug 任务发生在 app:compileDebugJavaWithJavac（对应 javac）之后，和 app:transformDexArchiveWithDexMergerForDebug（对应 dx）之前，同时在 build/intermediates/transforms/retrolambda 下面生产如图所示的 class 文件。

J8Sample.class 和 J8Sample$$Lambda$1.class 反编译之后的代码如下：

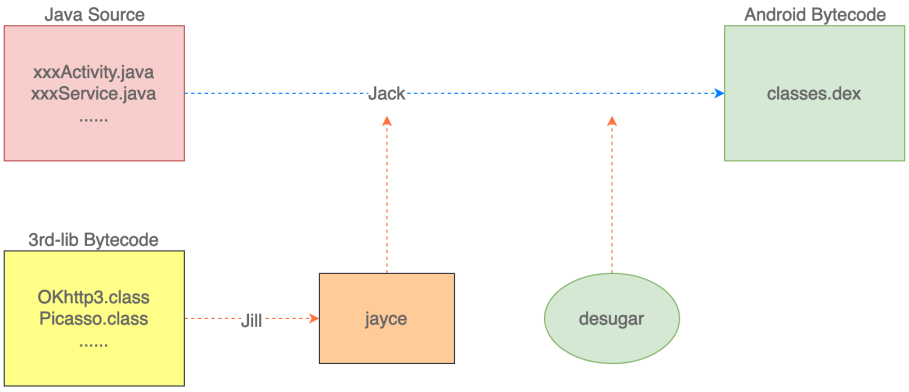
```
package com.j8sample;

public class J8Sample {
    public J8Sample() {
    }

    public static void main(String[] arg) {
        Runnable runnable = J8Sample$$Lambda$1.lambdaFactory$();
        (new Thread(runnable)).start();
        (new Thread(J8Sample$$Lambda$4.lambdaFactory$())).start();
    }
}
```

```
package com.j8sample;

// $FF: synthetic class
final class J8Sample$$Lambda$1 implements Runnable {
    private static final J8Sample$$Lambda$1 instance = new J8Sample$$Lambda$1();

    private J8Sample$$Lambda$1() {
    }

    public void run() {
        J8Sample.lambda$main$0();
    }

    public static Runnable lambdaFactory$() { return instance; }
}
```

通过反编译代码，可以看出 J8Sample.class 中 Lambda 表达式已经被我们熟悉的 1.7or1.6 的语句所替代。

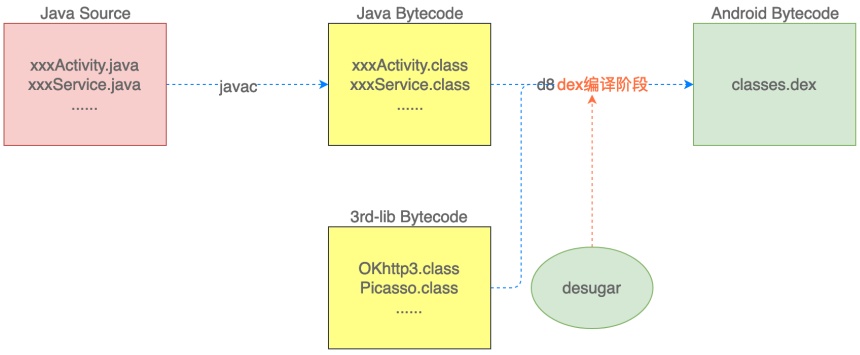
注意：右图中 J8Sample.lambda$main$0() 方法在左图中没有显示出来，但是 J8Sample.class 字节码确实是存在的。

Android 间接支持之 Jack&Jill 工具



Jack 是基于 Eclipse 的 ecj 编译开发的，Jill 是基于 ASM4 开发的。Jack&Jill 工具链是 Google 在 Android N(7.0) 发布的，用于替换 javac&dx 的工具链，并且在 jack 过程内置了 Desugar 过程。

但是在 Android P(9.0) 的时候将 Jack&Jill 工具链废弃了，被 javac&D8 工具链替代了。这里就不做 Desugar 具体分析了。



D8 是 Android P(9.0) 新增的 dex 编译器。并在 Android Studio 3.1 版本中默认使用 D8 作为 dex 的默认编译器。

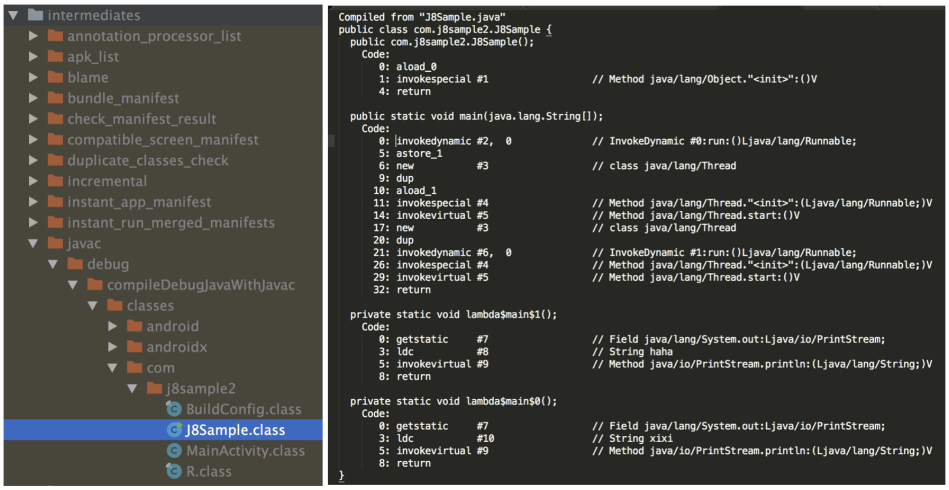
D8 Desugar

如图所示，Desugar 过程放在了 D8 的内部，由 Android Studio 这个 IDE 来实现这个转换，原理基本和 RetroLambda 是一样。

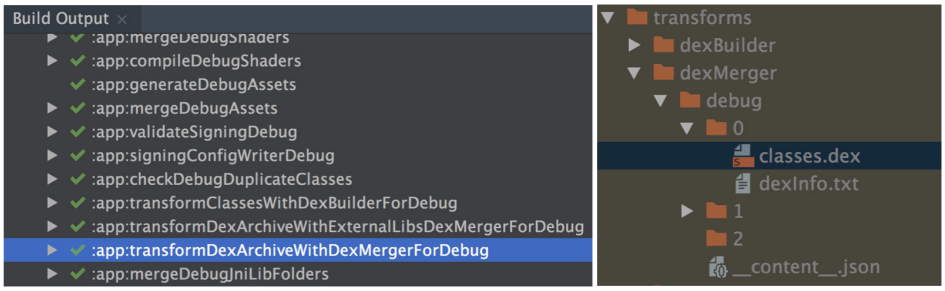
本质上也是参照 java/lang/invoke/LambdaMetafactory.metafactory 方法直接将原本在运行时生成在内存中的 J8Sample\$\$Lambda\$1.class，在 D8 的编译 dex 期间，直接生成并写入到 dex 文件中。

同样，将实例中的 J8Sample.java 放到支持 D8 的 Android 工程中：

同样，AndroidStudio -> Build -> make project 编译之后：



javac 编译之后的 J8Sample.class 还是使用 invokedynamic 指令，即这一步并没有 Desugar：



在 app:transformDexArchiveWithDexMergerForDebug（对应 dx）任务之后，再对应 build/intermediates/transforms/dexMerger 目录找第 0 个 classex.dex。

执行 $ANDROID_HOME/build-tools/28.0.3/dexdump -d classes.dex >> dexInfo.txt 拿到 dex 信息。

还是选取实例中 Lambda 表达式 1：Runnable runnable = () -> System.out.println("xixi"); 来进行分析。

这个 dexIno.txt 文件非常大，有 1.4M，我们通过 com.J8Smaple2.J8Sample 找到我们 J8Sample 在 dex 中位置。

```
Class #175            -
  Class descriptor  : 'Lcom/j8sample2/J8Sample;'
  Access flags      : 0x0001 (PUBLIC)
  Superclass        : 'Ljava/lang/Object;'
  Interfaces        -
  Static fields     -
  Instance fields   -
  Direct methods    -
    #0              : (in Lcom/j8sample2/J8Sample;)
      name          : '<init>'
      type          : '()V'
      access        : 0x10001 (PUBLIC CONSTRUCTOR)
      code          -
      registers     : 1
      ins           : 1
      outs          : 1
      insns size    : 4 16-bit code units
011bac:                                        |[011bac] com.j8sample2.J8Sample.<init>:()V
011bbc: 7010 e100 0000                         |0000: invoke-direct {v0}, Ljava/lang/Object;
011bc2: 0e00                                   |0003: return-void
```

新增方法：

```
|[011bc4] com.j8sample2.J8Sample.lambda$main$0:()V
|0000: sget-object v0, Ljava/lang/System;.out:Ljava/io/PrintStream; // field@154c
|0002: const-string v1, "xixi" // string@0829
|0004: invoke-virtual {v0, v1}, Ljava/io/PrintStream;.println:(Ljava/lang/String;)V
|0007: return-void
```

J8Sample.main 方法：

```
|[011c04] com.j8sample2.J8Sample.main:([Ljava/lang/String;)V
|0000: sget-object v0, Lcom/j8sample2/-$$Lambda$J8Sample$jWmuYH0zEF070TKXrjBFgnnqOKc;.INSTANCE:L
|0002: new-instance v1, Ljava/lang/Thread; // type@00cd
|0004: invoke-direct {v1, v0}, Ljava/lang/Thread;.<init>:(Ljava/lang/Runnable;)V // method@00e2
|0007: invoke-virtual {v1}, Ljava/lang/Thread;.start:()V // method@00e3
|000a: new-instance v1, Ljava/lang/Thread; // type@00cd
|000c: sget-object v2, Lcom/j8sample2/-$$Lambda$J8Sample$WGbOO3DKt0AUOe3xQIFKehlVuRk;.INSTANCE:L
|000e: invoke-direct {v1, v2}, Ljava/lang/Thread;.<init>:(Ljava/lang/Runnable;)V // method@00e2
|0011: invoke-virtual {v1}, Ljava/lang/Thread;.start:()V // method@00e3
|0014: return-void
```

图中选中部分，对应就是 Lambda 表达式 1 desugar 之后的内容。

翻译成 Java 的话就变成了：new Lcom/j8sample2/-$$Lambda$J8Sample$jWmuYH0zEF070TKXrjBFgnnqOKc 这个生成类的一个对象。

类 Lcom/j8sample2/-$$Lambda$J8Sample$jWmuYH0zEF070TKXrjBFgnnqOKc 对应前面的生成的 J8Sample$$Lambda$1 类型，只不过数字 1 变成了 Hash 值。

```
Class #172            -
  Class descriptor  : 'Lcom/j8sample2/-$$Lambda$J8Sample$jWmuYH0zEF070TKXrjBFgnnqOKc;'
  Access flags      : 0x1011 (PUBLIC FINAL SYNTHETIC)
  Superclass        : 'Ljava/lang/Object;'
  Interfaces        -
    #0              : 'Ljava/lang/Runnable;'
```

实现 Interface Ljava/lang/Runnable。Lcom/j8sample2/-$$Lambda$J8Sample$jWmuYH0zEF070TKXrjBFgnnqOKc.run 方法：

```
|[011b08] com.j8sample2.-..Lambda.J8Sample.jWmuYH0zEF070TKXrjBFgnnqOKc.run:()V
|0000: invoke-static {}, Lcom/j8sample2/J8Sample;.lambda$main$0:()V // method@00c9
|0003: return-void
```

到这里，是不是和前面 RetroLambda 就一样了。

总结

至此，Lambda 及其 invokedynamic 指令、RetroLambda 插件、D8 编译器各自的原理分析都已经结束了。

相比较 Lambda 在 Java8 自己内部的实现：即运行时，在内存中动态生成关联的函数式接口的实例类型，通过 BSM- 引导方法找到该内存类（字节码层面的反射）。

在 Android 上的其他三种 Desugar 方式，原理都是一样的，区别在于时机不同：

1. RetroLambda 将函数式接口对应的实例类型的生产过程，放在 javac 编译之后，dx 编译之前，并动态修改了表达式所属的字节码文件。
2. Jack&Jill 是直接将接口对应的实例类型，直接 jack 过程中生成，并编译进了 dex 文件。
3. D8 的过程是在 dex 编译过程中，直接在内存生成接口对应的实例类型，并将生成的类型直接写入生成的 dex 文件中。

探讨

无论是 RetroLambda，还是 D8，对 Java8 的特性也不是全都支持。

Java8 新增的许多 API（例如：新的 DataAPI），就 D8 编译器而言，只有在 Android P(9.0) 版本中能直接运行。低于 9.0 就不行了。如何能够全版本支持 Java 8。D8 还有很长的一段路要走。

如果我们在低版本需要使用新的 API，目前可以采取将这些 API 打包进去的临时办法。

写到这里，肯定有人要提出，为什么不直接使用 Kotlin 呢？确实 Kotlin 对 Lambda 表达式、函数引用等特性都做了很好的支持，但是现实的情况中，Kotlin 很难取代 Android 中的 Java。新业务、新工程还相对容易，对老业务来说，尤其是经过多年沉淀，工程结构复杂，迁移改造带来的收益，往往远远小于迁移改造带来的成本和不可控之风险。Kotlin 和 Java 同时存在的情况，长期来看是一个必然的结果。

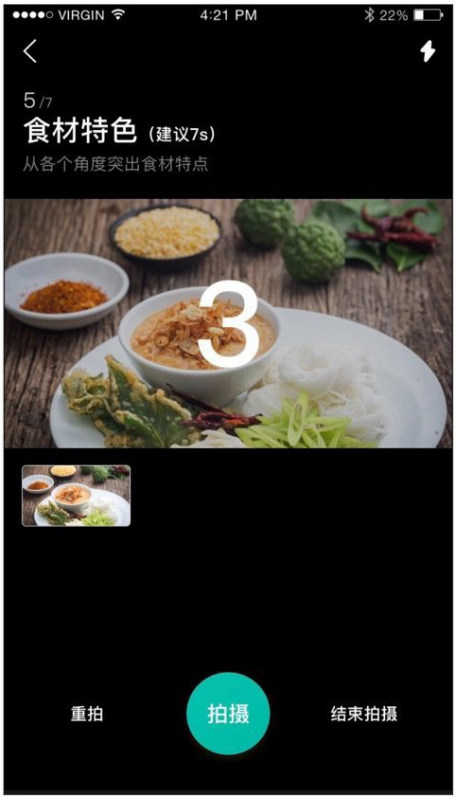
至于 Java 8 的其他特性呢，D8 是如何实现的，也可以按照上面类似的方式去分析，甚至可以结合 Kotlin 实现的方式，一探究竟。

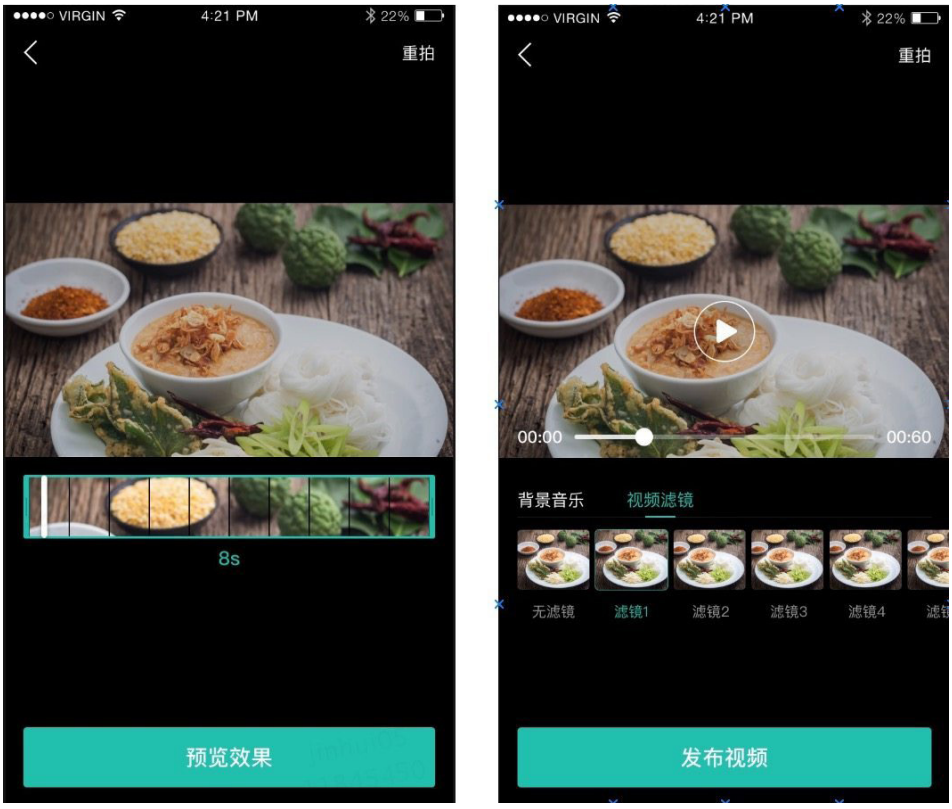
美团外卖商家端视频探索之旅

金辉　李琼

背景

美团外卖至今已迅猛发展了六年，随着外卖业务量级与日俱增，单一的文字和图片已无法满足商家的需求，商家迫切需要更丰富的商品描述手段吸引用户，增加流量，进而提高下单转化率和下单量。商品视频的引入，在一定程度上可以提升商品信息描述丰富度，以更加直观的方式为商家引流，增加收益。为此，商家端引入了视频功能，进行了一系列视频功能开发，核心功能包含视频处理（混音，滤镜，加水印，动画等）、视频拍摄、合成等，最终效果图如下所示：





自视频功能上线后，每周视频样本量及使用视频的商家量大幅增加，视频录制成功率达 99.533%，视频处理成功率 98.818%，音频处理成功率 99.959%，Crash 率稳定在 0.1‰，稳定性高且可用性强。目前，视频功能已在蜜蜂 App、闪购业务和商家业务上使用。

对于视频链路的开发，我们经历了方案选型、架构设计及优化、业务实践、功能测试、监控运维、更新维护等各个环节，核心环节如下图所示。在开发过程中，我们遇到了各种技术问题和挑战，下文会针对遇到的问题、挑战，及其解决方案进行重点阐述。

方案选型

在方案选型时，重点对核心流程和视频格式进行选型。我们以功能覆盖度、稳定性及效率、可定制性、成本及开源性做为核心指标，从而衡量方案的高可用性和可行性。

1. 核心流程选型

视频开发涉及的核心流程包括播放、录制、合成、裁剪、后期处理（编解码、滤镜、混音、动画、水印）等。结合商家端业务场景，我们有针对性的进行方案调研。重点调研了业界现有方案，如阿里的云视频点播方案、腾讯云视频点播方案、大众点评 App 的 UGC 方案，及其它的一些第三方开源方案等，并进行了整体匹配度的对比，如下图所示：

方案	能力	是否开源	收费	SDK大小	稳定性/性能	说明	结论
腾讯	丰富	否	10000/年	15M左右	高可靠，性能优 兼容性好	稳定性强，功能丰富，性能效率好，兼容性优	优点：软硬编码支持，稳定强，功能丰富 缺点：收费，定制化难度大
阿里	丰富	否	49000/年	20M左右	高可靠，性能优	功能丰富，稳定性强，性能效率好，安全性高	优点：性能优，安全性高 缺点：成本高，功能略臃肿，定制化难度大
点评方案	满足基本	否	否	集成方式待定	不保证	支持部分功能，业务场景有差异	优势：沟通协作良好可共建 缺点：业务场景差异大，部分功能不支持
其他开源	弱	是	否	待定	差	无完善的开源库，杂乱	开发代价太大，但可作参考

阿里和腾讯的云视频点播方案比较成熟，集成度高，且能力丰富，稳定性及效率也很高。但两者成本较高，需要收费，且 SDK 大小均在 15M 以上，对于我们的业务场景来说有些过于臃肿，定制性较弱，无法迅速的支持我们做定制性扩展。

当时的点评 App UGC 方案，基础能力是满足的，但因业务场景差异：

- 比如外卖的视频拍摄功能要求在竖屏下保证 16:9 的视频宽高比，这就需要对原有的采集区域进行截取，视频段落的裁剪支持不够等，业务场景的差异导致了实现方案存在巨大的差异，故放弃了点评 App UGC 方案。其他的一些开源方案（比如 Grafika 等），也无法满足要求，这里不再一一赘述。

通过技术调研和分析，吸取各开源项目的优点，并参考点评 App UGC、Google CTS 方案，对核心流程做了最终的方案选型，打造一个适合我们业务场景的方案，如下表所示：

核心能力	方案	效果
视频播放	ijkplayer+AndroidVideoCache	兼容性强 支持边缓存边播
视频录制	MediaRecorder+MediaCodec	采集区域的裁剪
视频合成	mp4parser	多文件拼接
视频裁剪	MediaCodec	误差微秒级
后期处理	MediaCodec，OpenGL，AAC	滤镜 动画 贴纸 混音等

2. 视频格式选型

文件格式	视频编码格式	视频分辨率	帧率	码率	音频压缩编码	音频采样率	音频码率
mp4	H.264	比例16：9	30fps	1200kb/s	AAC	44.1khz	128kb/s

- 采用 H.264 的视频协议：H.264 的标准成熟稳定，普及率高。其最大的优势是具有很高的数据压缩比率，在同等图像质量的条件下，H.264 的压缩比是 MPEG-2 的 2 倍以上，是 MPEG-4 的 1.5 ~ 2 倍。
- 采用 AAC 的音频协议：AAC 是一种专为声音数据设计的文件压缩格式。它采用了全新的算法进行编码，是新一代的音频有损压缩技术，具有更加高效，更具有”性价比“的特点。

整体架构

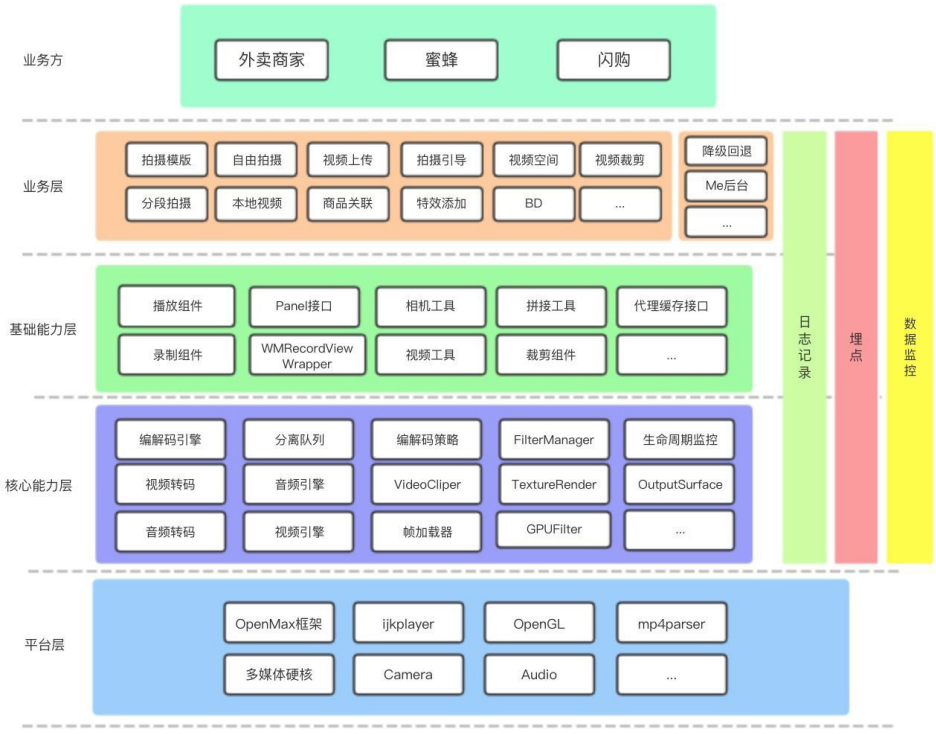
我们整体的架构设计，用以满足业务扩展和平台化需要，可复用、可扩展，且可快速接入。架构采用分层设计，基础能力和组件进行下沉，业务和视频能力做分离，最大化降低业务方的接入成本，三方业务只需要接入视频基础 SDK，直接使用相关能力组件或者工具即可。

整体架构分为四层，分别为平台层、核心能力层、基础组件层、业务层。

- **平台层**：依赖系统提供的平台能力，比如 Camera、OpenGL、MediaCodec 和 MediaMuxer 等，也包括引入的平台能力，比如 ijkplayer 播放器、mp4parser 。
- **核心能力层**：该层提供了视频服务的核心能力，包括音视频编解码、音视频的转码引擎、滤镜渲染能力等 。
- **基础能力层**：暴露了基础组件和能力，提供了播放、裁剪、录屏等基础组件和对应的基础工具类，并提供了可定制的播放面板，可定制的缓存接口等。
- **业务层**：包括段落拍摄、自由拍摄、视频空间、拍摄模版预览及加载等。

我们的视频能力层对业务层是透明的，业务层与能力层隔离，并对业务层提供了部分定制化的接口支持，这样的设计降低了业务方的接入成本，并方便业务方的

扩展，比如支持蜜蜂 App 的播放面板定制，还支持缓存策略、编解码策略的可定制。整体设计如下图所示：



实践经验

在视频开发实践中，因业务场景的复杂性，我们遇到了多种问题和挑战。下面以核心功能为基点，围绕各功能遇到的问题做详细介绍。

视频播放

播放器是视频播放基础。针对播放器，我们进行了一系列的方案调研和选择。在此环节，遇到的挑战如下：

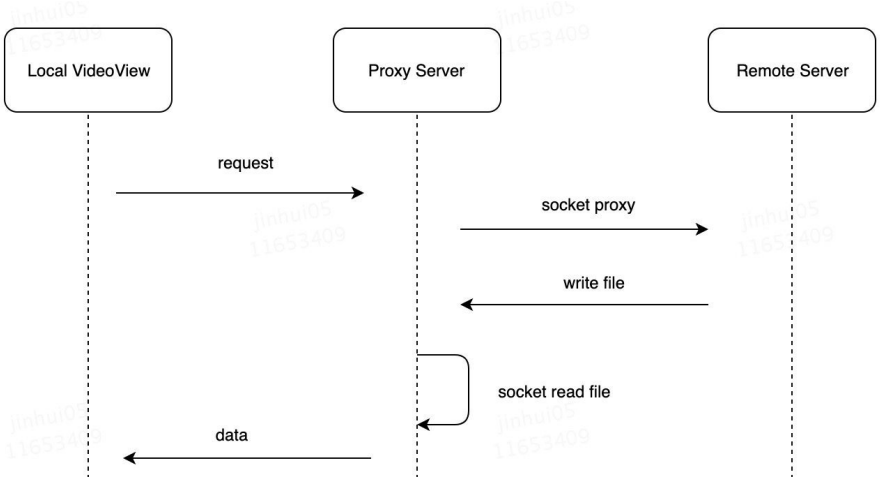
1. 兼容性问题

2. 缓存问题

针对兼容性问题，Android 有原生的 MediaPlayer，但其版本兼容问题偏多且支持格式有限，而我们需要支持播放本地视频，本地视频格式又无法控制，故该方案被舍弃。ijkplayer 基于 FFmpeg，与 MediaPlayer 相比，优点比较突出：具备跨平台能力，支持 Android 与 iOS；提供了类似 MediaPlayer 的 API，可兼容不同版本；可实现软硬解码自由切换，拥有 FFmpeg 的能力，支持多种流媒体协议。基于上述原因，我们最终决定选用 ijkplayer。

但紧接着我们又发现 ijkplayer 本身不支持边缓存边播放，频繁的加载视频导致耗费大量的流量，且在弱网或者 3G 网络下很容易导致播放卡顿，所以这里就衍生出了缓存的问题。

针对缓存问题，我们引入 AndroidVideoCache 的技术方案，利用本地的代理去请求数据，先本地保存文件缓存，客户端通过 Socket 读取本地的文件缓存进行视频播放，这样就做到了边播放边缓存的策略，流程如下图：



此外，我们还对 AndroidVideoCache 做了一些技术改造：

- 优化缓存策略。针对缓存策略的单一性，支持有限的最大文件数和文件大小问

题，我们调整为由业务方可以动态定制缓存策略；

- 解决内存泄露隐患。对其页面退出时请求不关闭会导致的内存泄露，我们为其添加了完整的生命周期监控，解决了内存泄露问题。

视频录制

在视频拍摄的时候，最为常用的方式是采用 MediaRecorder+Camera 技术，采集摄像头可见区域。但因我们的业务场景要求视频采集的时候，只录制采集区域的部分区域且比例保持宽高比 16:9，在保证预览图像不拉伸的情况下，只能对完整的采集区域做裁剪，这无形增加了开发难度和挑战。通过大量的资料分析，我们重点调研了有两种方案：

1. Camera+AudioRecord+MediaCodec+Surface
2. MediaRecorder+MediaCodec

方案 1 需要 Camera 采集 YUV 帧，进行截取采集，最后再将 YUV 帧和 PCM 帧进行编码生成 mp4 文件，虽然其效率高，但存在不可把控的风险。

方案 2 综合评估后是改造风险最小的。 综合成本和风险考量，我们保守的采用了方案 2，该方案是对裁剪区域进行坐标换算（如果用前置摄像头拍摄录制视频，会出现预览画面和录制的视频是镜像的问题，需要处理）。当录制完视频后，生成了 mp4 文件，用 MediaCodec 对其编码，在编码阶段再利用 OpenGL 做内容区域的裁剪来实现。但该方案又引发了如下挑战：

（1）对焦问题

因我们对采集区域做了裁剪，引发了点触对焦问题。比如用户点击了相机预览画面，正常情况下会触发相机的对焦动作，但是用户的点击区域只是预览画面的部分区域，这就导致了相机的对焦区域错乱，不能正常进行对焦。后期经过问题排查，对点触区域再次进行相应的坐标变换，最终得到正确的对焦区域。

（2）兼容适配

我们的视频录制利用 MediaRecorder，在获取配置信息时，由于 Android 碎片

化问题，不同的设备支持的配置信息不同，所以就会出现设备适配问题。

```
                        // VIVO Y66 模版拍摄时候，播放某些有问题的视频文件的同
时去录制视频，会导致 MediaServer 挂掉的问题
        // 发现将 1080P 尺寸的配置降低到 720P 即可避免此问题
        // 但是 720P 尺寸的配置下，又存在绿边问题，因此再降到 480
        if(isVIVOY66() && mMediaServerDied) {
            return getCamcorderProfile(CamcorderProfile.QUALITY_480P);
        }

        //SM-C9000，在 1280 x 720 分辨率时有一条绿边。网上有种说法是 GPU 对数据
进行了优化，
使得 GPU 产生的图像分辨率
        // 和常规分辨率存在微小差异，造成图像色彩混乱，修复后存在绿边问题。
        // 测试发现，降低分辨率或者升高分辨率都可以绕开这个问题。
        if (VideoAdapt.MODEL_SM_C9000.equals(Build.MODEL)) {
            return getCamcorderProfile(CamcorderProfile.QUALITY_HIGH);
        }

        // 优先选择 1080 P 的配置
        CamcorderProfile camcorderProfile =
getCamcorderProfile(CamcorderProfile.
QUALITY_1080P);
        if (camcorderProfile == null) {
            camcorderProfile = getCamcorderProfile(CamcorderProfile.
QUALITY_720P);
        }
        // 某些机型上这个 QUALITY_HIGH 有点问题，可能通过这个参数拿到的配置是
1080p，所
以这里也可能拿不到
        if (camcorderProfile == null) {
            camcorderProfile = getCamcorderProfile(CamcorderProfile.
QUALITY_HIGH);
        }
        // 兜底
        if (camcorderProfile == null) {
            camcorderProfile = getCamcorderProfile(CamcorderProfile.
QUALITY_480P);
        }
```

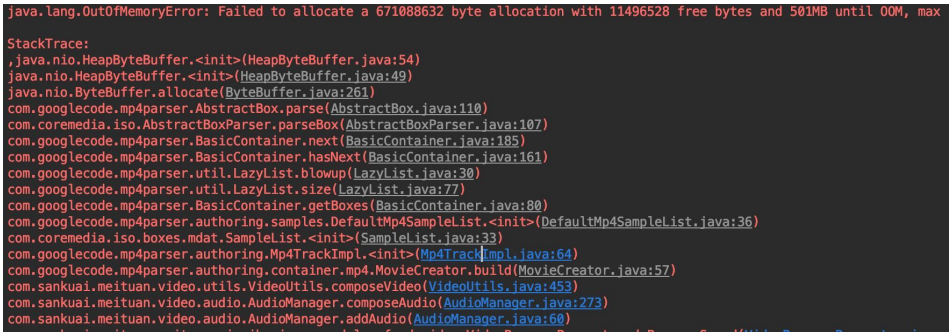
视频合成

我们的视频拍摄有段落拍摄这种场景，商家可根据事先下载的模板进行分段拍摄，最后会对每一段的视频做拼接，拼接成一个完整的 mp4 文件。mp4 由若干个 Box 组成，所有数据都封装在 Box 中，且 Box 可再包含 Box 的被称为 Container

Box。mp4 中 Track 表示一个视频或音频序列，是 Sample 的集合，而 Sample 又可分为 Video Smaple 和 Audio Sample。Video Smaple 代表一帧或一组连续视频帧，Audio Sample 即为一段连续的压缩音频数据。（详见 mp4 文件结构。）

基于上面的业务场景需要，视频合成的基础能力我们采用 mp4parser 技术实现（也可用 FFmpeg 等其他手段）。mp4parser 在拼接视频时，先将视频的音轨和视频轨进行分离，然后进行视频和音频轨的追加，最终将合成后的视频轨和音频轨放入容器里（这里的容器就是 mp4 的 Box）。采用 mp4parser 技术简单高效，API 设计简洁清晰，满足需求。

但我们发现某些被编码或处理过的 mp4 文件可能会存在特殊的 Box，并且 mp4parser 是不支持的。经过源码分析和原因推导，发现当遇到这种特殊格式的 Box 时，会申请分配一个比较大的空间用来存放数据，很容易造成 OOM（内存溢出），见下图所示。于是，我们对这种拼接场景下做了有效规避，仅在段落拍摄下使用 mp4parser 的拼接功能，保证我们处理过的文件不会包含这种特殊的 Box。



视频裁剪

我们刚开始采用 mp4parser 技术完成视频裁剪，在实践中发现其精度误差存在很大的问题，甚至会影响正常的业务需求。比如我们禁止裁剪出 3s 以下的视频，但是由于 mp4parser 产生的精度误差，导致 4-5s 的视频很容易裁剪出少于 3s 的视频。究其原因，mp4parser 只能在关键帧（又称 I 帧，在视频编码中是一种自带全部信息的独立帧）进行切割，这样就可能存在一些问题。比如在视频截取的起始

时间位置并不是关键帧，因此会造成误差，无法保证精度而且是秒级误差。以下为 mp4parser 裁剪的关键代码：

```
public static double correctTimeToSyncSample(Track track, double
cutHere, boolean next) {
        double[] timeOfSyncSamples = new double[track.getSyncSamples().
length];
        long currentSample = 0;
        double currentTime = 0;
        for (int i = 0; i < track.getSampleDurations().length; i++) {
            long delta = track.getSampleDurations()[i];
            int index = Arrays.binarySearch(track.getSyncSamples(),
currentSample + 1);
            if (index >= 0) {
                timeOfSyncSamples[index] = currentTime;
            }
            currentTime += ((double) delta / (double) track.
getTrackMetaData().getTimescale());
            currentSample++;
        }
        double previous = 0;
        for (double timeOfSyncSample : timeOfSyncSamples) {
            if (timeOfSyncSample > cutHere) {
                if (next) {
                    return timeOfSyncSample;
                } else {
                    return previous;
                }
            }
            previous = timeOfSyncSample;
        }
        return timeOfSyncSamples[timeOfSyncSamples.length - 1];
}
```

为了解决精度问题，我们废弃了 mp4parser，采用 MediaCodec 的方案，虽然该方案会增加复杂度，但是误差精度大大降低。

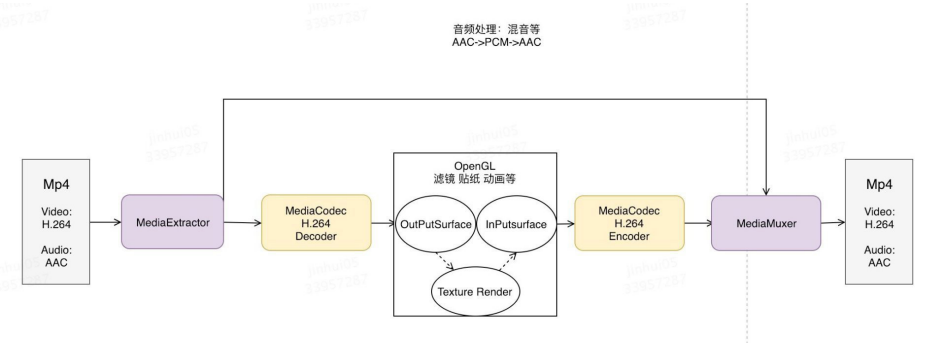
方案具体实施如下：先获得目标时间的上一帧信息，对视频解码，然后根据起始时间和截取时长进行切割，最后将裁剪后的音视频信息进行压缩编码，再封装进 mp4 容器中，这样我们的裁剪精度从秒级误差降低到微秒级误差，大大提高了容错率。

视频处理

视频处理是整个视频能力最核心的部分，会涉及硬编解码（遵循 OpenMAX 框

架）、OpenGL、音频处理等相关能力。

下图是视频处理的核心流程，会先将音视频做分离，并行处理音视频的编解码，并加入特效处理，最后合成进一个 mp4 文件中。



在实践过程中，我们遇到了一些需要特别注意的问题，比如开发时遇到的坑，严重的兼容性问题（包括硬件兼容性和系统版本兼容性问题）等。下面重点讲几个有代表性的问题。

1. 偶数宽高的编解码器

视频经过编码后输出特定宽高的视频文件时出现了如下错误，信息里仅提示了 Colorformat 错误，具体如下：

```
19624-19828 I/OMXClient: Treble IOmx obtained
19624-19820 I/VideoTrackTranscoder: mEncoder.name = OMX.qcom.video.encoder.avc
19624-19820 I/VideoTrackTranscoder: mOutputFormat :  {color-format=39, i-frame-interval=10, mime=video/avc, width=1080, bitrate=358
19624-19828 W/OMXUtils: do not know color format 0x7fa30c06 = 2141391878
19624-19828 W/OMXUtils: do not know color format 0x7fa30c04 = 2141391876
    do not know color format 0x7fa30c08 = 2141391880
    do not know color format 0x7fa30c07 = 2141391879
19624-19828 W/OMXUtils: do not know color format 0x7f000789 = 2130708361
19624-19828 E/ACodec: [OMX.qcom.video.encoder.avc] does not support color format 39
    [OMX.qcom.video.encoder.avc] configureCodec returning error -61
    signalError(omxError 0x80001001, internalError -61)
19624-19827 E/MediaCodec: Codec reported err 0xfffffffc3, actionCode 0, while in state 3
19624-19820 E/MediaCodec: configure failed with err 0xfffffffc3, resetting...
19624-19828 I/OMXClient: Treble IOmx obtained
19624-19820 W/System.err: android.media.MediaCodec$CodecException: Error 0xfffffffc3
        at android.media.MediaCodec.native_configure(Native Method)
        at android.media.MediaCodec.configure(MediaCodec.java:1943)
        at android.media.MediaCodec.configure(MediaCodec.java:1872)
        at com.dianping.video.videofilter.transcoder.engine.VideoTrackTranscoder.setup(VideoTrackTranscoder.java:104)
```

查阅大量资料，也没能解释清楚这个异常的存在。基于日志错误信息，并通过系统源码定位，也只是发现了是和设置的参数不兼容导致的。经过反复的试错，最后确认是部分编解码器只支持偶数的视频宽高，所以我们对视频的宽高做了偶数限制。引

起该问题的核心代码如下：

```
status_t ACodec::setupVideoEncoder(const char *mime, const sp<AMessage> &msg,
        sp<AMessage> &outputFormat, sp<AMessage> &inputFormat) {
    if (!msg->findInt32("color-format", &tmp)) {
        return INVALID_OPERATION;
    }
    OMX_COLOR_FORMATTYPE colorFormat =
        static_cast<OMX_COLOR_FORMATTYPE>(tmp);
    status_t err = setVideoPortFormatType(
            kPortIndexInput, OMX_VIDEO_CodingUnused, colorFormat);
    if (err != OK) {
        ALOGE("[%s] does not support color format %d",
              mComponentName.c_str(), colorFormat);
        return err;
    }
    .......
}
status_t ACodec::setVideoPortFormatType(OMX_U32 portIndex,OMX_VIDEO_
CODINGTYPE
compressionFormat,
        OMX_COLOR_FORMATTYPE colorFormat,bool usingNativeBuffers) {
    ......
    for (OMX_U32 index = 0; index <= kMaxIndicesToCheck; ++index) {
        format.nIndex = index;
        status_t err = mOMX->getParameter(
                mNode, OMX_IndexParamVideoPortFormat,
                &format, sizeof(format));
        if (err != OK) {
            return err;
        }
     ......
}
```

2. 颜色格式

我们在处理视频帧的时候，一开始获得的是从 Camera 读取到的基本的 YUV 格式数据，如果给编码器设置 YUV 帧格式，需要考虑 YUV 的颜色格式。这是因为 YUV 根据其采样比例，UV 分量的排列顺序有很多种不同的颜色格式，Android 也支持不同的 YUV 格式，如果颜色格式不对，会导致花屏等问题。

3. 16 位对齐

这也是硬编码中老生常谈的问题了，因为 H264 编码需要 16*16 的编码块大小。

如果一开始设置输出的视频宽高没有进行 16 字节对齐，在某些设备（华为，三星等）就会出现绿边，或者花屏。

4. 二次渲染

4.1 视频旋转

在最后的视频处理阶段，用户可以实时的看到加滤镜后的视频效果。这就需要对原始的视频帧进行二次处理，然后在播放器的 Surface 上渲染。首先我们需要 OpenGL 的渲染环境（通过 OpenGL 的固有流程创建），渲染环境完成后就可以对视频的帧数据进行二次处理了。通过 SurfaceTexture 的 updateTexImage 接口，可将视频流中最新的帧数据更新到对应的 GL 纹理，再操作 GL 纹理进行滤镜、动画等处理。在处理视频帧数据的时候，首先遇到的是角度问题。在正常播放下（不利用 OpenGL 处理情况下）通过设置 TextureView 的角度（和视频的角度做转换）就可以解决，但是加了滤镜后这一方案就失效了。原因是视频的原始数据经过纹理处理再渲染到 Surface 上，单纯设置 TextureView 的角度就失效了，解决方案就是对 OpenGL 传入的纹理坐标做相应的旋转（依据视频的本身的角度）。

4.2 渲染停滞

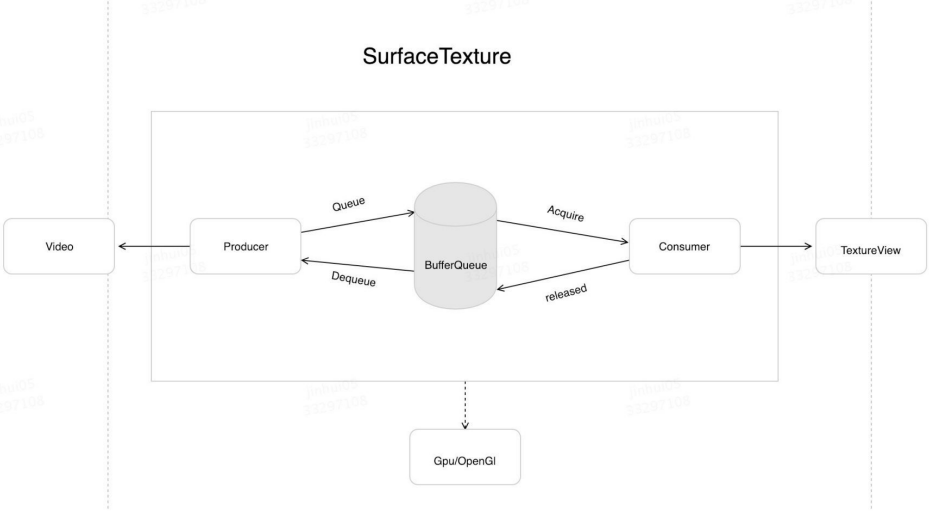
视频在二次渲染后会出现偶现的画面停滞现象，主要是 SurfaceTexture 的 OnFrameAvailableListener 不返回数据了。该问题的根本原因是 GPU 的渲染和视频帧的读取不同步，进而导致 SurfaceTexture 的底层核心 BufferQueue 读取 Buffer 出了问题。下面我们通过 BufferQueue 的机制和核心源码深入研究下：

首先从二次渲染的工作流程入手。从图像流（来自 Camera 预览、视频解码、GL 绘制场景等）中获得帧数据，此时 OnFrameAvailableListener 会回调。再调用 updateTexImage()，会根据内容流中最近的图像更新 SurfaceTexture 对应的 GL 纹理对象。我们再对纹理对象做处理，比如添加滤镜等效果。SurfaceTexture 底层核心管理者是 BufferQueue，本身基于生产者消费者模式。

BufferQueue 管理的 Buffer 状态分为：FREE，DEQUEUED，QUEUED，ACQUIRED，SHARED。当 Producer 需要填充数据时，需要先 Dequeue 一

个 Free 状态的 Buffer，此时 Buffer 的状态为 DEQUEUED，成功后持有者为 Producer。随后 Producer 填充数据完毕后，进行 Queue 操作，Buffer 状态流转为 QUEUED，且 Owner 变为 BufferQueue，同时会回调 BufferQueue 持有的 ConsumerListener 的 onFrameAvailable，进而通知 Consumer 可对数据进行二次处理了。Consumer 先通过 Acquire 操作，获取处于 QUEUED 状态的 Buffer，此时 Owner 为 Consumer。当 Consumer 消费完 Buffer 后，会执行 Release，该 Buffer 会流转回 BufferQueue 以便重用。BufferQueue 核心数据为 GraphicBuffer，而 GraphicBuffer 会根据场景、申请的内存大小、申请方式等的不同而有所不同。

SurfaceTexture 的核心流程如下图：



通过上图可知，我们的 Producer 是 Video，填充视频帧后，再对纹理进行特效处理（滤镜等），最后再渲染出来。前面我们分析了 BufferQueue 的工作流程，但是在 Producer 要填充数据，执行 dequeueBuffer 操作时，如果有 Buffer 已经 QUEUED，且申请的 dequeuedCount 大于 mMaxDequeuedBufferCount，就不会再继续申请 Free Buffer 了，Producer 就无法 DequeueBuffer，也就导致 onFrameAvailable 无法最终调用，核心源码如下：

```
status_t BufferQueueProducer::dequeueBuffer(int
*outSlot,sp<android::Fence> *outFence,
uint32_t width, uint32_t height,
       PixelFormat format, uint32_t usage,FrameEventHistoryDelta*
outTimestamps) {
       ......
       int found = BufferItem::INVALID_BUFFER_SLOT;
       while (found == BufferItem::INVALID_BUFFER_SLOT) {
            status_t status =
waitForFreeSlotThenRelock(FreeSlotCaller::Dequeue,
                      & found);
            if (status != NO_ERROR) {
                return status;
            }
        }
        ......
}
status_t BufferQueueProducer::waitForFreeSlotThenRelock(FreeSlotCaller
caller,
                    int*found) const{
        ......
        while (tryAgain) {
            int dequeuedCount = 0;
            int acquiredCount = 0;
            for (int s : mCore -> mActiveBuffers) {
                if (mSlots[s].mBufferState.isDequeued()) {
                    ++dequeuedCount;
                }
                if (mSlots[s].mBufferState.isAcquired()) {
                    ++acquiredCount;
                }
            }
            // Producers are not allowed to dequeue more than
            // mMaxDequeuedBufferCount buffers.
            // This check is only done if a buffer has already been
queued
            if (mCore -> mBufferHasBeenQueued &&
                    dequeuedCount >= mCore -> mMaxDequeuedBufferCount) {
                BQ_LOGE("%s: attempting to exceed the max dequeued
buffer count "
                        "(%d)", callerString, mCore ->
mMaxDequeuedBufferCount);
                return INVALID_OPERATION;
            }
        }
        .......
 }
```

5. 码流适配

视频的监控体系发现，Android 9.0 的系统出现大量的编解码失败问题，错误信息都是相同的。在 MediaCodec 的 Configure 时候出异常了，主要原因是我们强制使用了 CQ 码流，Android 9.0 以前并无问题，但 9.0 及以后对 CQ 码流增加了新的校验机制而我们没有适配。核心流程代码如下：

```
status_t ACodec::configureCodec(
        const char *mime, const sp<AMessage> &msg) {
       .......
      if (encoder) {
        if (mIsVideo || mIsImage) {
          if (!findVideoBitrateControlInfo(msg, &bitrateMode, &bitrate,
&quality)) {
                return INVALID_OPERATION;
            }
      } else if (strcasecmp(mime, MEDIA_MIMETYPE_AUDIO_FLAC)
           && !msg->findInt32("bitrate", &bitrate)) {
          return INVALID_OPERATION;
      }
   }
   .......
}
static bool findVideoBitrateControlInfo(const sp<AMessage> &msg,
        OMX_VIDEO_CONTROLRATETYPE *mode, int32_t *bitrate, int32_t
*quality) {
    *mode = getVideoBitrateMode(msg);
    bool isCQ = (*mode == OMX_Video_ControlRateConstantQuality);
    return (!isCQ && msg->findInt32("bitrate", bitrate))
         || (isCQ && msg->findInt32("quality", quality));
}
9.0 前并无对 CQ 码流的强校验，如果不支持该码流也会使用默认支持的码流，
static OMX_VIDEO_CONTROLRATETYPE getBitrateMode(const sp<AMessage>
&msg) {
    int32_t tmp;
    if (!msg->findInt32("bitrate-mode", &tmp)) {
        return OMX_Video_ControlRateVariable;
    }
    return static_cast<OMX_VIDEO_CONTROLRATETYPE>(tmp);
}
```

关于码流还有个问题就是如果通过系统的接口 isBitrateModeSupported（int mode），判断是否支持该码流可能会出现误判，究其原因是 framework 层写死了该返回值，而并没有从硬件层或从 media_codecs.xml 去获取该值。关于码流各硬件厂商

支持的差异性，可能谷歌也认为码流的兼容性太碎片化，不建议用非默认的码流。

6. 音频处理

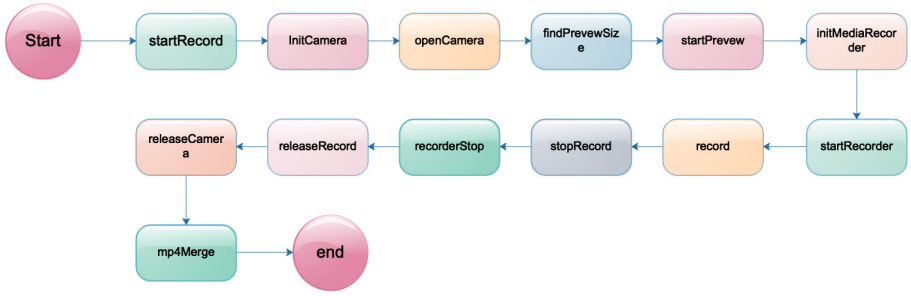
音频处理还括对音频的混音，消声等操作。在混音操作的时候，还要注意音频文件的单声道转换等问题。

其实视频问题总结起来，大部分是都会牵扯到编解码（尤其是使用硬编码），需要大量的适配工作（以上也只是部分问题，碎片化还是很严峻的），所以就需要兜底容错方案，比如加入软编。

线上监控

视频功能引入了埋点，日志，链路监控等技术手段进行线上的监控，我们可以针对监控结果进行降级或维护更新。埋点更多的是产品维度的数据收集，日志是辅助定位问题的，而链路监控则可以做到监控预警。我们加了拍摄流程，音视频处理，视频上传流程的全链路监控，整个链路如果任何一个节点出问题都认为是整个链路的失败，若失败次数超过阈值就会通过大象或邮件进行报警，我们在适配 Andorid 9.0 码流问题时，最早发现也是由于链路监控的预警。所有全链路的成功率目标值均为 98%，若成功率低于 92% 的目标阈值就会触发报警，我们会根据报警的信息和日志定位分析，该异常的影响范围，再根据影响范围确定是否热修复或者降级。如下以拍摄流程为例，其链路各核心节点的监控：

拍摄流程全链路，如下图（各关键节点监控）：



容灾降级

视频功能目前只支持粗粒度的降级策略。我们在视频入口处做了开关控制，关掉后所有的视频功能都无法使用。我们通过线上监控到视频的稳定性和成功率在特定机型无法保证，导致影响用户正常的使用商家端 App，我们支持针对特定设备做降级。后续我们可以做更细粒度的降级策略，比如根据 P0 级功能做降级，或者编解码策略的降级等

维护更新

视频功能上线后，经历了几个稳定的版本，保持着较高的成功率，但近期收到了 sniffer 的邮件报警，发现视频处理链路的失败次数明显增多，通过 sniffer 收集的信息发现大部分都是 Android 9.0 的问题（也就是上面讲的 Android 9.0 码流适配的问题），我们在商家端 5.2 版本进行了修复，该问题解决后我们的视频处理链路成功率也恢复到了 98% 以上。

总结和规划

视频功能上线后，稳定性、内存、CPU 等一些相关指标数据比较理想，我们建设的视频监控体系，也支撑着视频核心业务的监控，一些异常报警也让我们及时发现问题并迅速对异常进行维护更新，但视频技术栈也是远比本文介绍的要庞大，怎么提高秒播率，怎么提高编解码效率，还有硬编解码过程中可能造成的花屏，绿边等问题都是挑战，需要更深入的研究解决。

未来我们会继续致力于提高视频处理的兼容性和效率，优化现有流程，我们会对音频和视频处理合并处理，也会引入软编和自定义编解码算法。

美团外卖大前端团队将来也会继续致力于提高用户的体验，并且会将在实践过程中遇到的问题进行总结，沉底技术，积极的和大家分享，如果你也对视频感兴趣，欢迎加入我们。

参考资料

1. Android 开发者官网
2. Google CTS
3. Grafika
4. BufferQueue 原理介绍
5. MediaCodec 原理
6. 微信 Android 视频编码爬过的坑
7. mp4 文件结构
8. AndroidVideoCache 代理策略
9. ijkplayer
10. mp4parser
11. GPUImage

作者简介

金辉、**李琼**，美团外卖商家终端研发工程师。

招聘信息

美团外卖商家终端研发团队的主要职责是为商家提供稳定可靠的生产经营工具，在保障稳定的需求迭代的基础之上，持续优化 APP、PC 和 H5 的性能和用户体验，并不断优化提升团队的研发效率。团队主要负责的业务主要包括外卖订单、商品管理、门店装修、服务市场、门店运营、三方会话、蓝牙打印、自动接单、视频、语音和实时消息触达等基础业务，支撑整个外卖链路的高可用性及稳定发展。

团队通过架构演进及平台化体系化建设，有效支撑业务发展，提升了业务的可靠性和安全性；通过大规模落地跨平台和动态化技术，加快了业务迭代效率，帮助产品（PM）加快产品方案的落地及上线；通过监控容灾体系建设，有效保障业务的高可用性和稳定性；通过性能优化建设，保证 APP 的流畅性和良好用户体验。团队开发的技术栈包括 Android、iOS、React、Flutter 和 React Native。

后台

所谓「万变不离其宗」。无论前端框架如何在演进、变化，后台总能像「中流砥柱」一样，承载着各种「滔天巨浪」。

2019 年，美团技术博客共发布 18 篇后台的文章，我们分成了基本功、架构、实践与经验的总结等 3 个版块进行展示。《字节码增强技术探索》、《Java 动态追踪技术探究》，带领我们探究了技术底层的实现原理；《美团点评 Kubernetes 集群管理实践》、《美团集群调度系统 HULK 技术演进》，展现了对成千上万台机器如何进行管理的艺术。

「泰山不让土壤，故能成其大；河海不择细流，故能就其深」，因后台涉及的领域比较广、比较杂，任何一个想在这个领域取得成就的工程师，都要具有非凡的恒心与耐力，这些文章只是「后台」体系的冰山一角，我们需要学习的还有很多很多。这是我们的「宿命」，也是我们的「使命」。

这 18 篇文章，献给那些默默抗住亿万并发、管理数万台机器、操纵千百个数据库的后台工程师们，感谢你们的负重前行。

基本功

Java 魔法类：Unsafe 应用解析

璐璐

前言

Unsafe 是位于 sun.misc 包下的一个类，主要提供一些用于执行低级别、不安全操作的方法，如直接访问系统内存资源、自主管理内存资源等，这些方法在提升 Java 运行效率、增强 Java 语言底层资源操作能力方面起到了很大的作用。但由于 Unsafe 类使 Java 语言拥有了类似 C 语言指针一样操作内存空间的能力，这无疑也增加了程序发生相关指针问题的风险。在程序中过度、不正确使用 Unsafe 类会使得程序出错的概率变大，使得 Java 这种安全的语言变得不再“安全”，因此对 Unsafe 的使用一定要慎重。

注：本文对 sun.misc.Unsafe 公共 API 功能及相关应用场景进行介绍。

基本介绍

如下 Unsafe 源码所示，Unsafe 类为一单例实现，提供静态方法 getUnsafe 获取 Unsafe 实例，当且仅当调用 getUnsafe 方法的类为引导类加载器所加载时才合法，否则抛出 SecurityException 异常。

```
public final class Unsafe {
  // 单例对象
  private static final Unsafe theUnsafe;

  private Unsafe() {
  }
  @CallerSensitive
  public static Unsafe getUnsafe() {
    Class var0 = Reflection.getCallerClass();
```

```
    // 仅在引导类加载器 `BootstrapClassLoader` 加载时才合法
    if(!VM.isSystemDomainLoader(var0.getClassLoader())) {
      throw new SecurityException("Unsafe");
    } else {
      return theUnsafe;
    }
  }
}
```

那如若想使用这个类，该如何获取其实例？有如下两个可行方案。

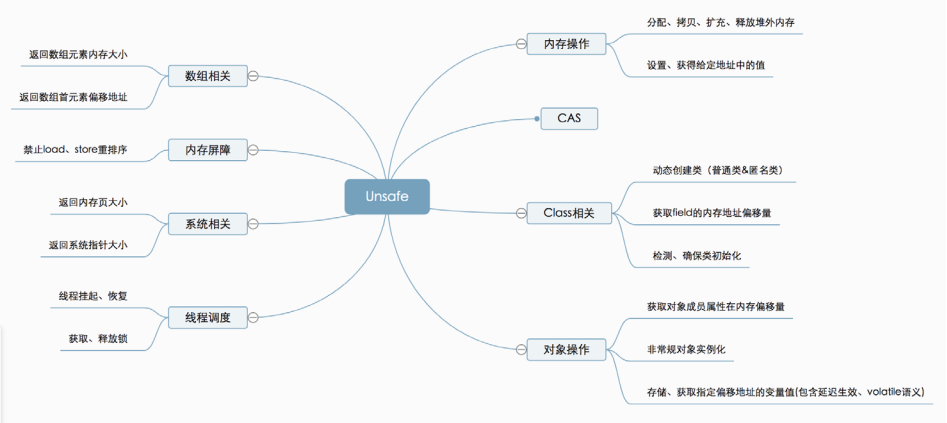
其一，从 getUnsafe 方法的使用限制条件出发，通过 Java 命令行命令 -Xbootclasspath/a 把调用 Unsafe 相关方法的类 A 所在 jar 包路径追加到默认的 bootstrap 路径中，使得 A 被引导类加载器加载，从而通过 Unsafe.getUnsafe 方法安全的获取 Unsafe 实例。

```
java -Xbootclasspath/a: ${path}   // 其中 path 为调用 Unsafe 相关方法的类所在
jar 包路径
```

其二，通过反射获取单例对象 theUnsafe。

```
private static Unsafe reflectGetUnsafe() {
    try {
      Field field = Unsafe.class.getDeclaredField("theUnsafe");
      field.setAccessible(true);
      return (Unsafe) field.get(null);
    } catch (Exception e) {
      log.error(e.getMessage(), e);
      return null;
    }
}
```

功能介绍



如上图所示，Unsafe 提供的 API 大致可分为内存操作、CAS、Class 相关、对象操作、线程调度、系统信息获取、内存屏障、数组操作等几类，下面将对其相关方法和应用场景进行详细介绍。

内存操作

这部分主要包含堆外内存的分配、拷贝、释放、给定地址值操作等方法。

```
// 分配内存，相当于 C++ 的 malloc 函数
public native long allocateMemory(long bytes);
// 扩充内存
public native long reallocateMemory(long address, long bytes);
// 释放内存
public native void freeMemory(long address);
// 在给定的内存块中设置值
public native void setMemory(Object o, long offset, long bytes, byte value);
// 内存拷贝
public native void copyMemory(Object srcBase, long srcOffset, Object
destBase, long
destOffset, long bytes);
// 获取给定地址值，忽略修饰限定符的访问限制。与此类似操作还有：getInt，getDouble，
getLong，getChar 等
public native Object getObject(Object o, long offset);
// 为给定地址设置值，忽略修饰限定符的访问限制，与此类似操作还有：
putInt,putDouble，
putLong，putChar 等
public native void putObject(Object o, long offset, Object x);
```

```
// 获取给定地址的 byte 类型的值（当且仅当该内存地址为 allocateMemory 分配时，此方法结
果为确定的）
public native byte getByte(long address);
// 为给定地址设置 byte 类型的值（当且仅当该内存地址为 allocateMemory 分配时，此方法结
果才是确定的）
public native void putByte(long address, byte x);
```

通常，我们在 Java 中创建的对象都处于堆内内存（heap）中，堆内内存是由 JVM 所管控的 Java 进程内存，并且它们遵循 JVM 的内存管理机制，JVM 会采用垃圾回收机制统一管理堆内存。与之相对的是堆外内存，存在于 JVM 管控之外的内存区域，Java 中对堆外内存的操作，依赖于 Unsafe 提供的操作堆外内存的 native 方法。

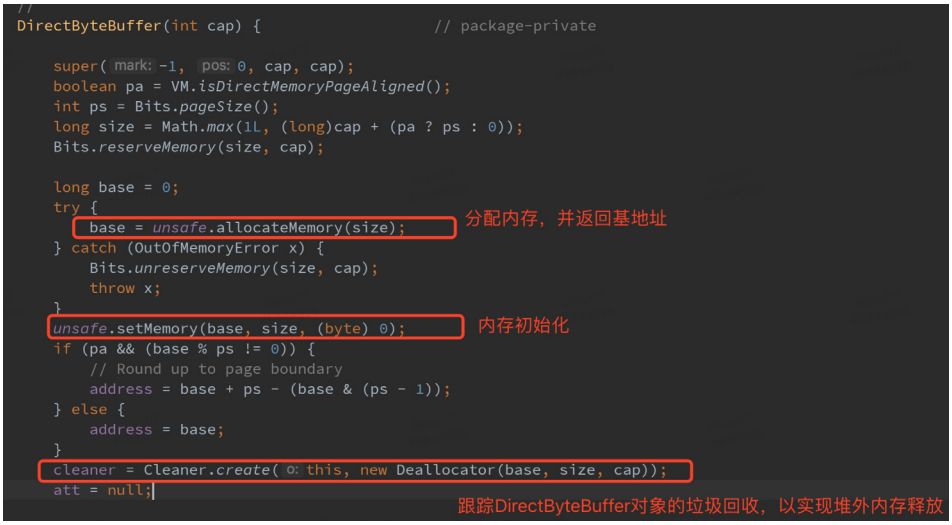
使用堆外内存的原因

- 对垃圾回收停顿的改善。由于堆外内存是直接受操作系统管理而不是 JVM，所以当我们使用堆外内存时，即可保持较小的堆内内存规模。从而在 GC 时减少回收停顿对于应用的影响。
- 提升程序 I/O 操作的性能。通常在 I/O 通信过程中，会存在堆内内存到堆外内存的数据拷贝操作，对于需要频繁进行内存间数据拷贝且生命周期较短的暂存数据，都建议存储到堆外内存。

典型应用

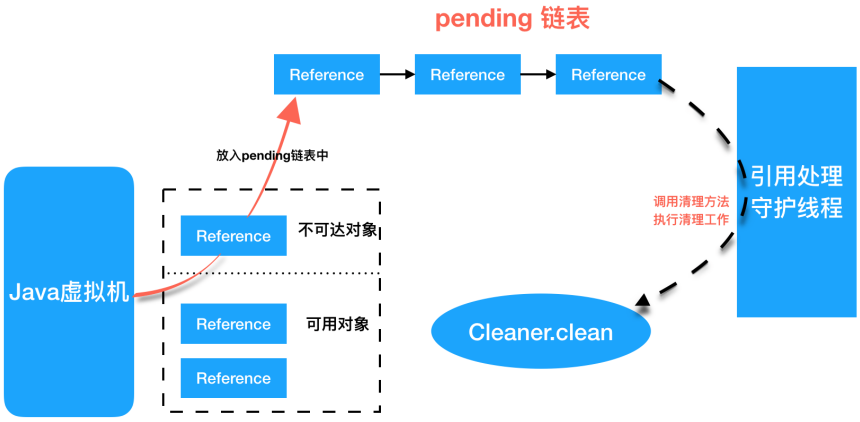
DirectByteBuffer 是 Java 用于实现堆外内存的一个重要类，通常用在通信过程中做缓冲池，如在 Netty、MINA 等 NIO 框架中应用广泛。DirectByteBuffer 对于堆外内存的创建、使用、销毁等逻辑均由 Unsafe 提供的堆外内存 API 来实现。

下图为 DirectByteBuffer 构造函数，创建 DirectByteBuffer 的时候，通过 Unsafe.allocateMemory 分配内存、Unsafe.setMemory 进行内存初始化，而后构建 Cleaner 对象用于跟踪 DirectByteBuffer 对象的垃圾回收，以实现当 DirectByteBuffer 被垃圾回收时，分配的堆外内存一起被释放。

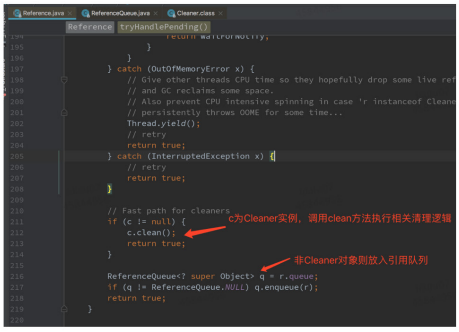


那么如何通过构建垃圾回收追踪对象 Cleaner 实现堆外内存释放呢?

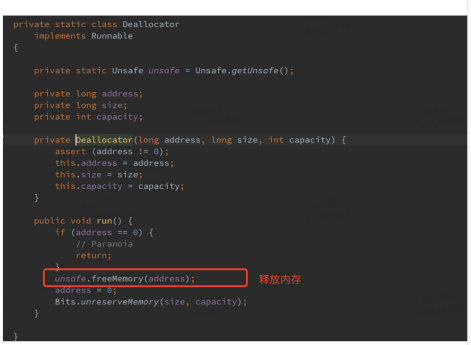
Cleaner 继承自 Java 四大引用类型之一的虚引用 PhantomReference（众所周知，无法通过虚引用获取与之关联的对象实例，且当对象仅被虚引用引用时，在任何发生 GC 的时候，其均可被回收），通常 PhantomReference 与引用队列 ReferenceQueue 结合使用，可以实现虚引用关联对象被垃圾回收时能够进行系统通知、资源清理等功能。如下图所示，当某个被 Cleaner 引用的对象将被回收时，JVM 垃圾收集器会将此对象的引用放入到对象引用中的 pending 链表中，等待 Reference-Handler 进行相关处理。其中，Reference-Handler 为一个拥有最高优先级的守护线程，会循环不断的处理 pending 链表中的对象引用，执行 Cleaner 的 clean 方法进行相关清理工作。



所以当 DirectByteBuffer 仅被 Cleaner 引用（即为虚引用）时，其可以在任意 GC 时段被回收。当 DirectByteBuffer 实例对象被回收时，在 Reference-Handler 线程操作中，会调用 Cleaner 的 clean 方法根据创建 Cleaner 时传入的 Deallocator 来进行堆外内存的释放。



图一



图二

CAS 相关

如下源代码释义所示，这部分主要为 CAS 相关操作的方法。

```
/**
     *  CAS
 * @param o          包含要修改 field 的对象
```

```
     * @param offset     对象中某 field 的偏移量
     * @param expected   期望值
     * @param update     更新值
     * @return           true | false
     */
    public final native boolean compareAndSwapObject(Object o, long
    offset,  Object expected,
    Object update);

    public final native boolean compareAndSwapInt(Object o, long offset,
    int expected,int
    update);

    public final native boolean compareAndSwapLong(Object o, long offset,
    long expected,
    long update);
```

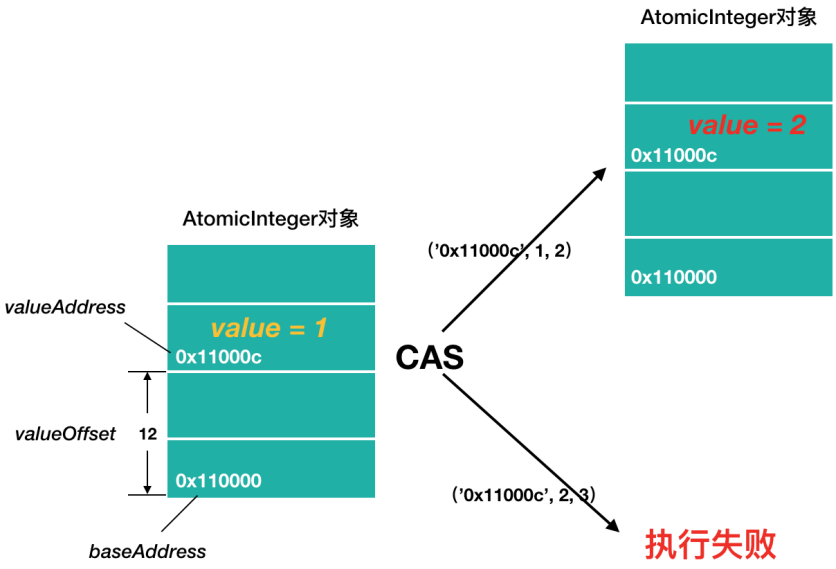
什么是 CAS？即比较并替换，实现并发算法时常用到的一种技术。CAS 操作包含三个操作数——内存位置、预期原值及新值。执行 CAS 操作的时候，将内存位置的值与预期原值比较，如果相匹配，那么处理器会自动将该位置值更新为新值，否则，处理器不做任何操作。我们都知道，CAS 是一条 CPU 的原子指令（cmpxchg 指令），不会造成所谓的数据不一致问题，Unsafe 提供的 CAS 方法（如 compareAndSwapXXX）底层实现即为 CPU 指令 cmpxchg。

典型应用

CAS 在 java.util.concurrent.atomic 相关类、Java AQS、CurrentHashMap 等实现上有非常广泛的应用。如下图所示，AtomicInteger 的实现中，静态字段 valueOffset 即为字段 value 的内存偏移地址，valueOffset 的值在 AtomicInteger 初始化时，在静态代码块中通过 Unsafe 的 objectFieldOffset 方法获取。在 AtomicInteger 中提供的线程安全方法中，通过字段 valueOffset 的值可以定位到 AtomicInteger 对象中 value 的内存地址，从而可以根据 CAS 实现对 value 字段的原子操作。

```
public class AtomicInteger extends Number implements java.io.Serializable {
    private static final long serialVersionUID = 6214790243416807050L;

    // setup to use Unsafe.compareAndSwapInt for updates
    private static final Unsafe unsafe = Unsafe.getUnsafe();
    private static final long valueOffset;

    static {
        try {
            valueOffset = unsafe.objectFieldOffset
                (AtomicInteger.class.getDeclaredField( name: "value"));
        } catch (Exception ex) { throw new Error(ex); }
    }

    private volatile int value;
```

下图为某个 AtomicInteger 对象自增操作前后的内存示意图，对象的基地址 baseAddress=“0x110000”，通过 baseAddress+valueOffset 得到 value 的内存地址 valueAddress=“0x11000c”；然后通过 CAS 进行原子性的更新操作，成功则返回，否则继续重试，直到更新成功为止。



线程调度

这部分，包括线程挂起、恢复、锁机制等方法。

```
// 取消阻塞线程
public native void unpark(Object thread);
// 阻塞线程
public native void park(boolean isAbsolute, long time);
// 获得对象锁（可重入锁）
@Deprecated
public native void monitorEnter(Object o);
// 释放对象锁
@Deprecated
public native void monitorExit(Object o);
// 尝试获取对象锁
@Deprecated
public native boolean tryMonitorEnter(Object o);
```

如上源码说明中，方法 park、unpark 即可实现线程的挂起与恢复，将一个线程进行挂起是通过 park 方法实现的，调用 park 方法后，线程将一直阻塞直到超时或者中断等条件出现；unpark 可以终止一个挂起的线程，使其恢复正常。

典型应用

Java 锁和同步器框架的核心类 AbstractQueuedSynchronizer，就是通过调用 LockSupport.park() 和 LockSupport.unpark() 实现线程的阻塞和唤醒的，而 LockSupport 的 park、unpark 方法实际是调用 Unsafe 的 park、unpark 方式来实现。

Class 相关

此部分主要提供 Class 和它的静态字段的操作相关方法，包含静态字段内存定位、定义类、定义匿名类、检验 & 确保初始化等。

```
// 获取给定静态字段的内存地址偏移量，这个值对于给定的字段是唯一且固定不变的
public native long staticFieldOffset(Field f);
// 获取一个静态类中给定字段的对象指针
public native Object staticFieldBase(Field f);
// 判断是否需要初始化一个类，通常在获取一个类的静态属性的时候（因为一个类如果没初始化，
它的静态属性也不会初始化）使用。 当且仅当 ensureClassInitialized 方法不生效时返回 false。
public native boolean shouldBeInitialized(Class<?> c);
// 检测给定的类是否已经初始化。通常在获取一个类的静态属性的时候（因为一个类如果没初始化，它的静态属性也不会初始化）使用。
```

```
public native void ensureClassInitialized(Class<?> c);
// 定义一个类，此方法会跳过 JVM 的所有安全检查，默认情况下，ClassLoader（类加载器）
和 ProtectionDomain（保护域）实例来源于调用者
public native Class<?> defineClass(String name, byte[] b, int off, int
len, ClassLoader loader,
ProtectionDomain protectionDomain);
// 定义一个匿名类
public native Class<?> defineAnonymousClass(Class<?> hostClass, byte[]
data, Object[]
cpPatches);
```

典型应用

从 Java 8 开始，JDK 使用 invokedynamic 及 VM Anonymous Class 结合来实现 Java 语言层面上的 Lambda 表达式。

- **invokedynamic**：invokedynamic 是 Java 7 为了实现在 JVM 上运行动态语言而引入的一条新的虚拟机指令，它可以实现在运行期动态解析出调用点限定符所引用的方法，然后再执行该方法，invokedynamic 指令的分派逻辑是由用户设定的引导方法决定。
- **VM Anonymous Class**：可以看做是一种模板机制，针对于程序动态生成很多结构相同、仅若干常量不同的类时，可以先创建包含常量占位符的模板类，而后通过 Unsafe.defineAnonymousClass 方法定义具体类时填充模板的占位符生成具体的匿名类。生成的匿名类不显式挂在任何 ClassLoader 下面，只要当该类没有存在的实例对象、且没有强引用来引用该类的 Class 对象时，该类就会被 GC 回收。故而 VM Anonymous Class 相比于 Java 语言层面的匿名内部类无需通过 ClassClassLoader 进行类加载且更易回收。

在 Lambda 表达式实现中，通过 invokedynamic 指令调用引导方法生成调用点，在此过程中，会通过 ASM 动态生成字节码，而后利用 Unsafe 的 defineAnonymousClass 方法定义实现相应的函数式接口的匿名类，然后再实例化此匿名类，并返回与此匿名类中函数式方法的方法句柄关联的调用点；而后可以通过此调用点实现调用相应 Lambda 表达式定义逻辑的功能。下面以如下图所示的 Test

类来举例说明。

```java
import java.util.function.Consumer;


public class Test {

    public static void main(String[] args) throws Exception {
        Consumer<String> consumer = s -> System.out.println(s);
        consumer.accept( t: "lambda");
    }
}
```

Test 类编译后的 class 文件反编译后的结果如下图一所示（删除了对本文说明无意义的部分），我们可以从中看到 main 方法的指令实现、invokedynamic 指令调用的引导方法 BootstrapMethods、及静态方法 lambda$main$0（实现了 Lambda 表达式中字符串打印逻辑）等。在引导方法执行过程中，会通过 Unsafe.defineAnonymousClass 生成如下图二所示的实现 Consumer 接口的匿名类。其中，accept 方法通过调用 Test 类中的静态方法 lambda$main$0 来实现 Lambda 表达式中定义的逻辑。而后执行语句 consumer.accept("lambda") 其实就是调用下图二所示的匿名类的 accept 方法。

```
Compiled from "Test.java"
public class com.sankuai.meituan.trippackage.api.Test {

  public static void main(java.lang.String[]) throws java.lang.Exception;
    Code:
       0: invokedynamic #2,  0      // InvokeDynamic #0:accept:()Ljava/util/function/Consumer;
       5: astore_1
       6: aload_1
       7: ldc           #3          // String lambda
       9: invokeinterface #4,  2    // InterfaceMethod java/util/function/Consumer.accept:(Ljava/lang/Object;)V
      14: return

  private static void lambda$main$0(java.lang.String);
    Code:
       0: getstatic     #5          // Field java/lang/System.out:Ljava/io/PrintStream;
       3: aload_0
       4: invokevirtual #6          // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       7: return

  SourceFile: "Test.java"
  InnerClasses:
       public static final #75= #74 of #78; //Lookup=class java/lang/invoke/MethodHandles$Lookup
                                             of class java/lang/invoke/MethodHandles
  BootstrapMethods:
    0: #38 invokestatic java/lang/invoke/LambdaMetafactory.metafactory:(Ljava/lang/invoke/MethodHandles$Lookup;
           Ljava/lang/String;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodHandle;
           Ljava/lang/invoke/MethodType;)Ljava/lang/invoke/CallSite;
      Method arguments:
        #39 (Ljava/lang/Object;)V
        #40 invokestatic com/sankuai/meituan/trippackage/api/Test.lambda$main$0:(Ljava/lang/String;)V
        #41 (Ljava/lang/String;)V
}
```

引导方法

图一

```java
import java.lang.invoke.LambdaForm.Hidden;
import java.util.function.Consumer;

// $FF: synthetic class
final class Test$$Lambda$1 implements Consumer {
    private Test$$Lambda$1() {
    }

    @Hidden
    public void accept(Object var1) {
        Test.lambda$main$0((String)var1);
    }
}
```

图二

对象操作

此部分主要包含对象成员属性相关操作及非常规的对象实例化方式等相关方法。

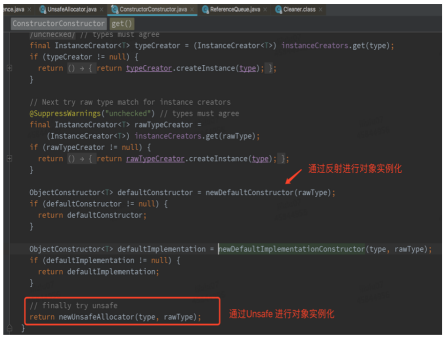
```
// 返回对象成员属性在内存地址相对于此对象的内存地址的偏移量
public native long objectFieldOffset(Field f);
// 获得给定对象的指定地址偏移量的值，与此类似操作还有：getInt，getDouble，getLong，
getChar 等
public native Object getObject(Object o, long offset);
// 给定对象的指定地址偏移量设值，与此类似操作还有：putInt，putDouble，putLong，
putChar 等
public native void putObject(Object o, long offset, Object x);
// 从对象的指定偏移量处获取变量的引用，使用 volatile 的加载语义
public native Object getObjectVolatile(Object o, long offset);
// 存储变量的引用到对象的指定的偏移量处，使用 volatile 的存储语义
public native void putObjectVolatile(Object o, long offset, Object x);
// 有序、延迟版本的 putObjectVolatile 方法，不保证值的改变被其他线程立即看到。只
有在 field 被 volatile 修饰符修饰时有效
public native void putOrderedObject(Object o, long offset, Object x);
// 绕过构造方法、初始化代码来创建对象
public native Object allocateInstance(Class<?> cls) throws
InstantiationException;
```

典型应用

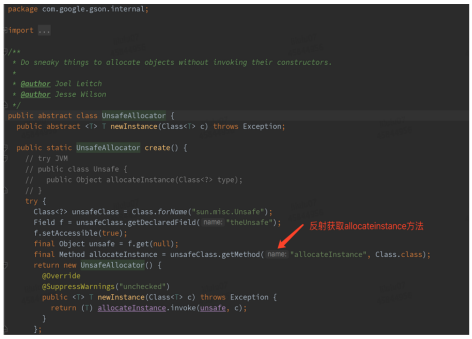
- **常规对象实例化方式**：我们通常所用到的创建对象的方式，从本质上来讲，都是通过 new 机制来实现对象的创建。但是，new 机制有个特点就是当类只提供有参的构造函数且无显示声明无参构造函数时，则必须使用有参构造函数进行对象构造，而使用有参构造函数时，必须传递相应个数的参数才能完成对象实例化。
- **非常规的实例化方式**：而 Unsafe 中提供 allocateInstance 方法，仅通过 Class 对象就可以创建此类的实例对象，而且不需要调用其构造函数、初始化代码、JVM 安全检查等。它抑制修饰符检测，也就是即使构造器是 private 修饰的也能通过此方法实例化，只需提类对象即可创建相应的对象。由于这种特性，allocateInstance 在 java.lang.invoke、Objenesis（提供绕过类构造器的对象生成方式）、Gson（反序列化时用到）中都有相应的应用。

如下图所示，在 Gson 反序列化时，如果类有默认构造函数，则通过反射调

用默认构造函数创建实例，否则通过 UnsafeAllocator 来实现对象实例的构造，UnsafeAllocator 通过调用 Unsafe 的 allocateInstance 实现对象的实例化，保证在目标类无默认构造函数时，反序列化不够影响。



图一



图二

数组相关

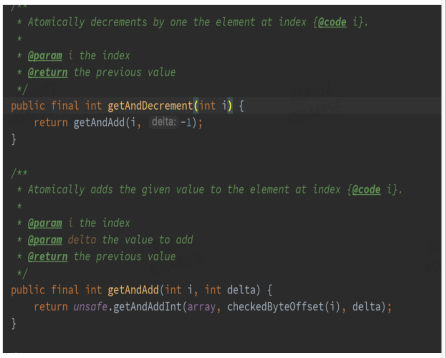
这部分主要介绍与数据操作相关的 arrayBaseOffset 与 arrayIndexScale 这两个方法，两者配合起来使用，即可定位数组中每个元素在内存中的位置。

```
// 返回数组中第一个元素的偏移地址
public native int arrayBaseOffset(Class<?> arrayClass);
// 返回数组中一个元素占用的大小
public native int arrayIndexScale(Class<?> arrayClass);
```

典型应用

这两个与数据操作相关的方法，在 java.util.concurrent.atomic 包下的 AtomicIntegerArray（可以实现对 Integer 数组中每个元素的原子性操作）中有典型的应用，如下图 AtomicIntegerArray 源码所示，通过 Unsafe 的 arrayBaseOffset、arrayIndexScale 分别获取数组首元素的偏移地址 base 及单个元素大小因子 scale。后续相关原子性操作，均依赖于这两个值进行数组中元素的定位，如下图二所示的 getAndAdd 方法即通过 checkedByteOffset 方法获取某数组元素的偏移地址，而后通过 CAS 实现原子性操作。

图一



图二

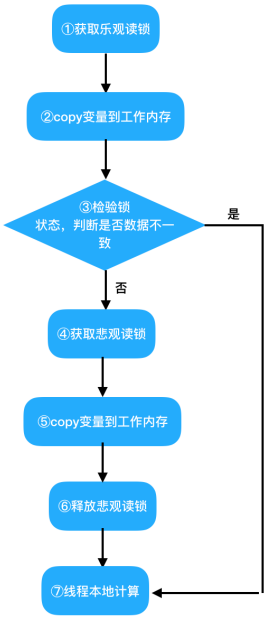
内存屏障

在 Java 8 中引入，用于定义内存屏障（也称内存栅栏，内存栅障，屏障指令等，是一类同步屏障指令，是 CPU 或编译器在对内存随机访问的操作中的一个同步点，使得此点之前的所有读写操作都执行后才可以开始执行此点之后的操作），避免代码重排序。

```
// 内存屏障，禁止 load 操作重排序。屏障前的 load 操作不能被重排序到屏障后，屏障后的 load 操作不能被重排序到屏障前
public native void loadFence();
// 内存屏障，禁止 store 操作重排序。屏障前的 store 操作不能被重排序到屏障后，屏障后的 store 操作不能被重排序到屏障前
public native void storeFence();
// 内存屏障，禁止 load、store 操作重排序
public native void fullFence();
```

典型应用

在 Java 8 中引入了一种锁的新机制——StampedLock，它可以看成是读写锁的一个改进版本。StampedLock 提供了一种乐观读锁的实现，这种乐观读锁类似于无锁的操作，完全不会阻塞写线程获取写锁，从而缓解读多写少时写线程“饥饿”现象。由于 StampedLock 提供的乐观读锁不阻塞写线程获取读锁，当线程共享变量从主内存 load 到线程工作内存时，会存在数据不一致问题，所以当使用 StampedLock 的乐观读锁时，需要遵从如下图用例中使用的模式来确保数据的一致性。




```
class Point {

    private double x, y;

    private final StampedLock sl = new StampedLock();

    void move(double deltaX, double deltaY) {
        long stamp = sl.writeLock();    // 使用写锁-独占操作
        try {
            x += deltaX;
            y += deltaY;
        } finally {
            sl.unlockWrite(stamp);
        }
    }
    double distanceFromOrigin() {
        long stamp = sl.tryOptimisticRead();          // ①
        double currentX = x, currentY = y;            // ②
        if (!sl.validate(stamp)) {                    // ③
            stamp = sl.readLock();                    // ④
            try {
                currentX = x;                         // ⑤
                currentY = y;
            } finally {
                sl.unlockRead(stamp);                 // ⑥
            }
        }
        return Math.sqrt(currentX * currentX + currentY * currentY); // ⑦
    }
}
```

如上图用例所示计算坐标点 Point 对象，包含点移动方法 move 及计算此点到原点的距离的方法 distanceFromOrigin。在方法 distanceFromOrigin 中，首先，通过 tryOptimisticRead 方法获取乐观读标记；然后从主内存中加载点的坐标值 (x,y)；而后通过 StampedLock 的 validate 方法校验锁状态，判断坐标点 (x,y) 从主内存加载到线程工作内存过程中，主内存的值是否已被其他线程通过 move 方法修改，如果 validate 返回值为 true，证明 (x, y) 的值未被修改，可参与后续计算；否则，需加悲观读锁，再次从主内存加载 (x,y) 的最新值，然后再进行距离计算。其中，校验锁状态这步操作至关重要，需要判断锁状态是否发生改变，从而判断之前 copy 到线程工作内存中的值是否与主内存的值存在不一致。

下图为 StampedLock.validate 方法的源码实现，通过锁标记与相关常量进行位运算、比较来校验锁状态，在校验逻辑之前，会通过 Unsafe 的 loadFence 方法加入一个 load 内存屏障，目的是避免上图用例中步骤②和 StampedLock.validate 中锁状态校验运算发生重排序导致锁状态校验不准确的问题。

```
/**
 * Returns true if the lock has not been exclusively acquired
 * since issuance of the given stamp. Always returns false if the
 * stamp is zero. Always returns true if the stamp represents a
 * currently held lock. Invoking this method with a value not
 * obtained from {@link #tryOptimisticRead} or a locking method
 * for this lock has no defined effect or result.
 *
 * @param stamp a stamp
 * @return {@code true} if the lock has not been exclusively acquired
 * since issuance of the given stamp; else false
 */
public boolean validate(long stamp) {
    U.loadFence();                    load 内存屏障
    return (stamp & SBITS) == (state & SBITS);
}

/**
 * If the lock state matches the given stamp, releases the
```

系统相关

这部分包含两个获取系统相关信息的方法。

```
// 返回系统指针的大小。返回值为 4（32 位系统）或 8（64 位系统）。
public native int addressSize();
// 内存页的大小，此值为 2 的幂次方。
public native int pageSize();
```

典型应用

如下图所示的代码片段，为 java.nio 下的工具类 Bits 中计算待申请内存所需内存页数量的静态方法，其依赖于 Unsafe 中 pageSize 方法获取系统内存页大小实现后续计算逻辑。

```
    private static int pageSize = -1;

    static int pageSize() {
        if (pageSize == -1)
            pageSize = unsafe().pageSize();
        return pageSize;
    }

    static int pageCount(long size) {
        return (int)(size + (long)pageSize() - 1L) / pageSize();
    }
```

结语

本文对 Java 中的 sun.misc.Unsafe 的用法及应用场景进行了基本介绍，我们可以看到 Unsafe 提供了很多便捷、有趣的 API 方法。即便如此，由于 Unsafe 中包含大量自主操作内存的方法，如若使用不当，会对程序带来许多不可控的灾难。因此对它的使用我们需要慎之又慎。

参考资料

- OpenJDK Unsafe source
- Java Magic. Part 4: sun.misc.Unsafe
- JVM crashes at libjvm.so
- Java 中神奇的双刃剑 - Unsafe
- JVM 源码分析之堆外内存完全解读
- 堆外内存 之 DirectByteBuffer 详解
- 《深入理解 Java 虚拟机（第 2 版）》

作者简介

璐璐，美团点评 Java 开发工程师。2017 年加入美团点评，负责美团点评境内度假的后端开发。

Java 动态追踪技术探究

高扬

引子

在遥远的希艾斯星球爪哇国塞沃城中，两名年轻的程序员正在为一件事情苦恼，程序出问题了，一时看不出问题出在哪里，于是有了以下对话：

“Debug 一下吧。”

“线上机器，没开 Debug 端口。”

“看日志，看看请求值和返回值分别是什么?”

“那段代码没打印日志。”

“改代码，加日志，重新发布一次。”

“怀疑是线程池的问题，重启会破坏现场。”

长达几十秒的沉默之后：“据说，排查问题的最高境界，就是只通过 Review 代码来发现问题。”

比几十秒长几十倍的沉默之后：“我轮询了那段代码一十七遍之后，终于得出一个结论。”

“结论是?”

“我还没到达只通过 Review 代码就能发现问题的至高境界。”

从 JSP 说起

对于大多数 Java 程序员来说，早期的时候，都会接触到一个叫做 JSP（Java Server Pages）的技术。虽然这种技术，在前后端代码分离、前后端逻辑分离、前后端组织架构分离的今天来看，已经过时了，但是其中还是有一些有意思的东西，值得拿出来说一说。

当时刚刚处于 Java 入门时期的我们，大多数精力似乎都放在了 JSP 的页面展示效果上了：

“这个表格显示的行数不对”

“原来是 for 循环写的有问题，改一下，刷新页面再试一遍”

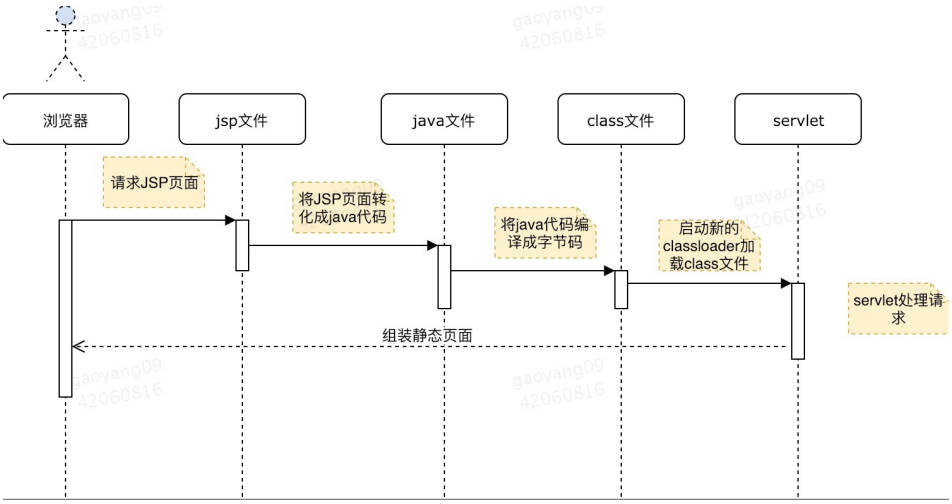
“嗯，好了，表格显示没问题了，但是，登录人的姓名没取到啊，是不是 Sesstion 获取有问题?”

“有可能，我再改一下，一会儿再刷新试试”

……

在一遍一遍修改代码刷新浏览器页面重试的时候，我们自己也许并没有注意到一件很酷的事情：我们修改完代码，居然只是简单地刷新一遍浏览器页面，修改就生效了，整个过程并没有重启 JVM。按照我们的常识，Java 程序一般都是在启动时加载类文件，如果都像 JSP 这样修改完代码，不用重启就生效的话，那文章开头的问题就可以解决了啊：Java 文件中加一段日志打印的代码，不重启就生效，既不破坏现场，又可以定位问题。忍不住试一试：修改、编译、替换 class 文件。额，不行，新改的代码并没有生效。那为什么偏偏 JSP 可以呢? 让我们先来看看 JSP 的运行原理。

当我们打开浏览器，请求访问一个 JSP 文件的时候，整个过程是这样的：



JSP 文件处理过程

JSP 文件修改过后，之所以能及时生效，是因为 Web 容器（Tomcat）会检查请求的 JSP 文件是否被更改过。如果发生过更改，那么就将 JSP 文件重新解析翻译成

一个新的 Sevlet 类，并加载到 JVM 中。之后的请求，都会由这个新的 Servet 来处理。这里有个问题，根据 Java 的类加载机制，在同一个 ClassLoader 中，类是不允许重复的。为了绕开这个限制，Web 容器每次都会创建一个新的 ClassLoader 实例，来加载新编译的 Servlet 类。之后的请求都会由这个新的 Servlet 来处理，这样就实现了新旧 JSP 的切换。

HTTP 服务是无状态的，所以 JSP 的场景基本上都是一次性消费，这种通过创建新的 ClassLoader 来“替换”class 的做法行得通，但是对于其他应用，比如 Spring 框架，即便这样做了，对象多数是单例，对于内存中已经创建好的对象，我们无法通过这种创建新的 ClassLoader 实例的方法来修改对象行为。

我就是想不重启应用加个日志打印，就这么难吗?

Java 对象行为

既然 JSP 的办法行不通，那我们来看看还有没有其他的办法。仔细想想，我们会发现，文章开头的问题本质上是动态改变内存中已存在对象的行为的问题。所以，我们得先弄清楚 JVM 中和对象行为有关的地方在哪里，有没有更改的可能性。

我们都知道，对象使用两种东西来描述事物：行为和属性。举个例子：

```
public class Person{
  private int age;
  private String name;
  public void speak(String str) {
    System.out.println(str);
  }
  public Person(int age, String name) {
    this.age = age;
    this.name = name;
  }
}
```

上面 Person 类中 age 和 name 是属性，speak 是行为。对象是类的事例，每个对象的属性都属于对象本身，但是每个对象的行为却是公共的。举个例子，比如我们现在基于 Person 类创建了两个对象，personA 和 personB：

```
Person personA = new Person(43, "lixunhuan");

personA.speak(" 我是李寻欢 ");

Person personB = new Person(23, "afei");

personB.speak(" 我是阿飞 ");
```

personA 和 personB 有各自的姓名和年龄，但是有共同的行为：speak。想象一下，如果我们是 Java 语言的设计者，我们会怎么存储对象的行为和属性呢?

“很简单，属性跟着对象走，每个对象都存一份。行为是公共的东西，抽离出来，单独放到一个地方。”

“咦? 抽离出公共的部分，跟代码复用好像啊。”

“大道至简，很多东西本来都是殊途同归。”

也就是说，第一步我们首先得找到存储对象行为的这个公共的地方。一番搜索之后，我们发现这样一段描述：

> Method area is created on virtual machine startup, shared among all Java virtual machine threads and it is logically part of heap area. It stores per-class structures such as the run-time constant pool, field and method data, and the code for methods and constructors.

Java 的对象行为（方法、函数）是存储在方法区的。

“方法区中的数据从哪来?”

“方法区中的数据是类加载时从 class 文件中提取出来的。”

“class 文件从哪来?”

“从 Java 或者其他符合 JVM 规范的源代码中编译而来。”

“源代码从哪来?”

“废话，当然是手写!”

“倒着推，手写没问题，编译没问题，至于加载……有没有办法加载一个已经加载过的类呢？如果有的话，我们就能修改字节码中目标方法所在的区域，然后重新加载这个类，这样方法区中的对象行为（方法）就被改变了，而且不改变对象的属性，也不影响已经存在对象的状态，那么就可以搞定这个问题了。可是，这岂不是违背了 JVM 的类加载原理？毕竟我们不想改变 ClassLoader。”

“少年，可以去看看 java.lang.instrument.Instrumentation。”

java.lang.instrument.Instrumentation

看完文档之后，我们发现这么两个接口：redefineClasses 和 retransformClasses。一个是重新定义 class，一个是修改 class。这两个大同小异，看 reDefineClasses 的说明：

> This method is used to replace the definition of a class without reference to the existing class file bytes, as one might do when recompiling from source for fix-and-continue debugging. Where the existing class file bytes are to be transformed (for example in bytecode instrumentation) retransformClasses should be used.

都是替换已经存在的 class 文件，redefineClasses 是自己提供字节码文件替换掉已存在的 class 文件，retransformClasses 是在已存在的字节码文件上修改后再替换之。

当然，运行时直接替换类很不安全。比如新的 class 文件引用了一个不存在的类，或者把某个类的一个 field 给删除了等等，这些情况都会引发异常。所以如文档中所言，instrument 存在诸多的限制：

> The redefinition may change method bodies, the constant pool and attributes. The redefinition must not add, remove or rename fields or methods, change the signatures of methods, or change inheritance. These restrictions maybe be lifted in future versions. The class file bytes are not checked, verified and installed until after the transforma-

tions have been applied, if the resultant bytes are in error this method will throw an exception.

我们能做的基本上也就是简单修改方法内的一些行为，这对于我们开头的问题，打印一段日志来说，已经足够了。当然，我们除了通过 reTransform 来打印日志，还能做很多其他非常有用的事情，这个下文会进行介绍。

那怎么得到我们需要的 class 文件呢？一个最简单的方法，是把修改后的 Java 文件重新编译一遍得到 class 文件，然后调用 redefineClasses 替换。但是对于没有（或者拿不到，或者不方便修改）源码的文件我们应该怎么办呢？其实对于 JVM 来说，不管是 Java 也好，Scala 也好，任何一种符合 JVM 规范的语言的源代码，都可以编译成 class 文件。JVM 的操作对象是 class 文件，而不是源码。所以，从这种意义上来讲，我们可以说“JVM 跟语言无关”。既然如此，不管有没有源码，其实我们只需要修改 class 文件就行了。

直接操作字节码

Java 是软件开发人员能读懂的语言，class 字节码是 JVM 能读懂的语言，class 字节码最终会被 JVM 解释成机器能读懂的语言。无论哪种语言，都是人创造的。所以，理论上（实际上也确实如此）人能读懂上述任何一种语言，既然能读懂，自然能修改。只要我们愿意，我们完全可以跳过 Java 编译器，直接写字节码文件，只不过这并不符合时代的发展罢了，毕竟高级语言设计之始就是为我们人类所服务，其开发效率也比机器语言高很多。

对于人类来说，字节码文件的可读性远远没有 Java 代码高。尽管如此，还是有一些杰出的程序员们创造出了可以用来直接编辑字节码的框架，提供接口可以让我们方便地操作字节码文件，进行注入修改类的方法，动态创造一个新的类等等操作。其中最著名的框架应该就是 ASM 了，cglib、Spring 等框架中对于字节码的操作就建立在 ASM 之上。

我们都知道，Spring 的 AOP 是基于动态代理实现的，Spring 会在运行时动态创建代理类，代理类中引用被代理类，在被代理的方法执行前后进行一些神秘的

操作。那么，Spring 是怎么在运行时创建代理类的呢？动态代理的美妙之处，就在于我们不必手动为每个需要被代理的类写代理类代码，Spring 在运行时会根据需要动态地创造出一个类，这里创造的过程并非通过字符串写 Java 文件，然后编译成 class 文件，然后加载。Spring 会直接“创造”一个 class 文件，然后加载，创造 class 文件的工具，就是 ASM 了。

到这里，我们知道了用 ASM 框架直接操作 class 文件，在类中加一段打印日志的代码，然后调用 retransformClasses 就可以了。

BTrace

截止到目前，我们都是停留在理论描述的层面。那么如何进行实现呢？先来看几个问题：

1. 在我们的工程中，谁来做这个寻找字节码，修改字节码，然后 reTransform 的动作呢？我们并非先知，不可能知道未来有没有可能遇到文章开头的这种问题。考虑到性价比，我们也不可能在每个工程中都开发一段专门做这些修改字节码、重新加载字节码的代码。
2. 如果 JVM 不在本地，在远程呢？
3. 如果连 ASM 都不会用呢？能不能更通用一些，更“傻瓜”一些。

幸运的是，因为有 BTrace 的存在，我们不必自己写一套这样的工具了。什么是 BTrace 呢？ BTrace 已经开源，项目描述极其简短：

A safe, dynamic tracing tool for the Java platform.

BTrace 是基于 Java 语言的一个安全的、可提供动态追踪服务的工具。BTrace 基于 ASM、Java Attach Api、Instruments 开发，为用户提供了很多注解。依靠这些注解，我们可以编写 BTrace 脚本（简单的 Java 代码）达到我们想要的效果，而不必深陷于 ASM 对字节码的操作中不可自拔。

看 BTrace 官方提供的一个简单例子：拦截所有 java.io 包中所有类中以 read 开头的方法，打印类名、方法名和参数名。当程序 IO 负载比较高的时候，就可以从输

出的信息中看到是哪些类所引起，是不是很方便?

```
package com.sun.btrace.samples;

import com.sun.btrace.annotations.*;
import com.sun.btrace.AnyType;
import static com.sun.btrace.BTraceUtils.*;

/**
 * This sample demonstrates regular expression
 * probe matching and getting input arguments
 * as an array - so that any overload variant
 * can be traced in "one place". This example
 * traces any "readXX" method on any class in
 * java.io package. Probed class, method and arg
 * array is printed in the action.
 */
@BTrace public class ArgArray {
    @OnMethod(
        clazz="/java\\.io\\..*/",
        method="/read.*/"
    )
    public static void anyRead(@ProbeClassName String pcn, @
ProbeMethodName String
pmn, AnyType[] args) {
        println(pcn);
        println(pmn);
        printArray(args);
    }
}
```

再来看另一个例子：每隔 2 秒打印截止到当前创建过的线程数。

```
package com.sun.btrace.samples;

import com.sun.btrace.annotations.*;
import static com.sun.btrace.BTraceUtils.*;
import com.sun.btrace.annotations.Export;

/**
 * This sample creates a jvmstat counter and
 * increments it everytime Thread.start() is
 * called. This thread count may be accessed
 * from outside the process. The @Export annotated
 * fields are mapped to jvmstat counters. The counter
 * name is "btrace." + <className> + "." + <fieldName>
 */
```

```
@BTrace public class ThreadCounter {

    // create a jvmstat counter using @Export
    @Export private static long count;

    @OnMethod(
        clazz="java.lang.Thread",
        method="start"
    )
    public static void onnewThread(@Self Thread t) {
        // updating counter is easy. Just assign to
        // the static field!
        count++;
    }

    @OnTimer(2000)
    public static void ontimer() {
        // we can access counter as "count" as well
        // as from jvmstat counter directly.
        println(count);
        // or equivalently ...
        println(Counters.perfLong("btrace.com.sun.btrace.samples.
ThreadCounter.count"));
    }
}
```

看了上面的用法是不是有所启发？忍不住冒出来许多想法。比如查看 HashMap 什么时候会触发 rehash，以及此时容器中有多少元素等等。

有了 BTrace，文章开头的问题可以得到完美的解决。至于 BTrace 具体有哪些功能，脚本怎么写，这些 Git 上 BTrace 工程中有大量的说明和举例，网上介绍 BTrace 用法的文章更是恒河沙数，这里就不再赘述了。

我们明白了原理，又有好用的工具支持，剩下的就是发挥我们的创造力了，只需在合适的场景下合理地进行使用即可。

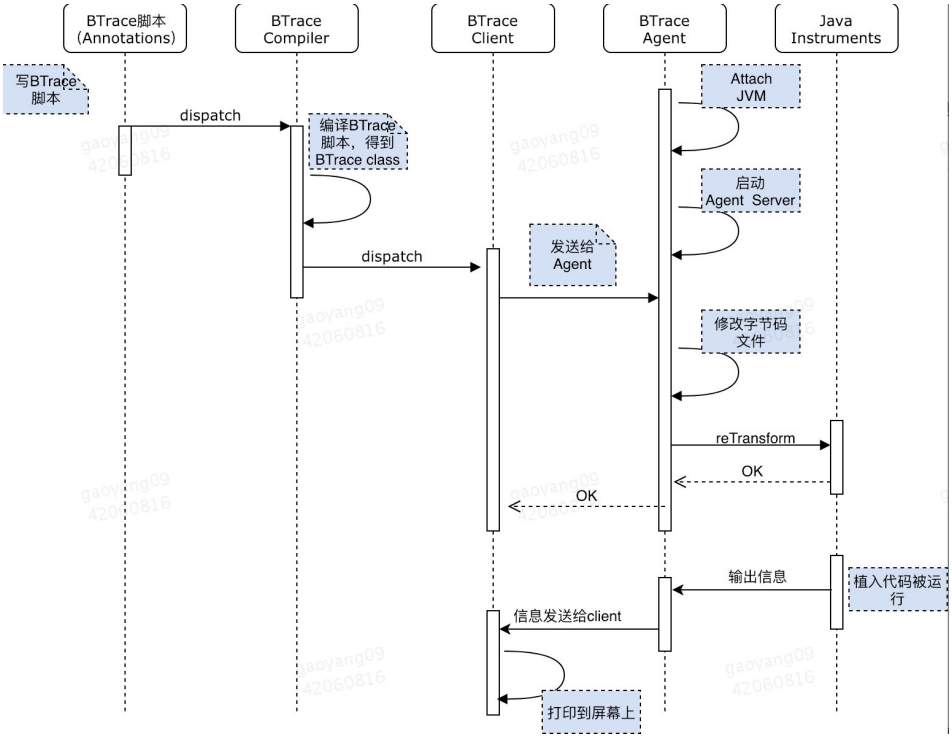
既然 BTrace 能解决上面我们提到的所有问题，那么 BTrace 的架构是怎样的呢？

BTrace 主要有下面几个模块：

1. BTrace 脚本：利用 BTrace 定义的注解，我们可以很方便地根据需要进行脚本的开发。

2. Compiler：将 BTrace 脚本编译成 BTrace class 文件。
3. Client：将 class 文件发送到 Agent。
4. Agent：基于 Java 的 Attach Api，Agent 可以动态附着到一个运行的 JVM 上，然后开启一个 BTrace Server，接收 client 发过来的 BTrace 脚本；解析脚本，然后根据脚本中的规则找到要修改的类；修改字节码后，调用 Java Instrument 的 reTransform 接口，完成对对象行为的修改并使之生效。

整个 BTrace 的架构大致如下：



BTrace 工作流程

BTrace 最终借 Instruments 实现 class 的替换。如上文所说，出于安全考虑，Instruments 在使用上存在诸多的限制，BTrace 也不例外。BTrace 对 JVM 来说是“只读的”，因此 BTrace 脚本的限制如下：

1. 不允许创建对象
2. 不允许创建数组
3. 不允许抛异常
4. 不允许 catch 异常
5. 不允许随意调用其他对象或者类的方法，只允许调用 com.sun.btrace.BTraceUtils 中提供的静态方法（一些数据处理和信息输出工具）
6. 不允许改变类的属性
7. 不允许有成员变量和方法，只允许存在 **static public void** 方法
8. 不允许有内部类、嵌套类
9. 不允许有同步方法和同步块
10. 不允许有循环
11. 不允许随意继承其他类（当然，java.lang.Object 除外）
12. 不允许实现接口
13. 不允许使用 assert
14. 不允许使用 Class 对象

如此多的限制，其实可以理解。BTrace 要做的是，虽然修改了字节码，但是除了输出需要的信息外，对整个程序的正常运行并没有影响。

Arthas

BTrace 脚本在使用上有一定的学习成本，如果能把一些常用的功能封装起来，对外直接提供简单的命令即可操作的话，那就再好不过了。阿里的工程师们早已想到这一点，就在去年（2018 年 9 月份），阿里巴巴开源了自己的 Java 诊断工具——Arthas。Arthas 提供简单的命令行操作，功能强大。究其背后的技术原理，和本文中提到的大致无二。Arthas 的文档很全面，想详细了解的话可以戳这里。

本文旨在说明 Java 动态追踪技术的来龙去脉，掌握技术背后的原理之后，只要愿意，各位读者也可以开发出自己的“冰封王座”出来。

尾声：三生万物

现在，让我们试着站在更高的地方“俯瞰”这些问题。

Java 的 Instruments 给运行时的动态追踪留下了希望，Attach API 则给运行时动态追踪提供了“出入口”，ASM 则大大方便了“人类”操作 Java 字节码的操作。

基于 Instruments 和 Attach API 前辈们创造出了诸如 JProfiler、Jvisualvm、BTrace、Arthas 这样的工具。以 ASM 为基础发展出了 cglib、动态代理，继而是应用广泛的 Spring AOP。

Java 是静态语言，运行时不允许改变数据结构。然而，Java 5 引入 Instruments，Java 6 引入 Attach API 之后，事情开始变得不一样了。虽然存在诸多限制，然而，在前辈们的努力下，仅仅是利用预留的近似于“只读”的这一点点狭小的空间，仍然创造出了各种大放异彩的技术，极大地提高了软件开发人员定位问题的效率。

计算机应该是人类有史以来最伟大的发明之一，从电磁感应磁生电，到高低电压模拟 0 和 1 的比特，再到二进制表示出几种基本类型，再到基本类型表示出无穷的对象，最后无穷的对象组合交互模拟现实生活乃至整个宇宙。

两千五百年前，《道德经》有言：“道生一，一生二，二生三，三生万物。”

两千五百年后，计算机的发展过程也大抵如此吧。

作者简介

高扬，2017 年加入美团打车，负责美团打车结算系统的开发。

字节码增强技术探索

赵泽恩

1. 字节码

1.1 什么是字节码?

Java 之所以可以“一次编译，到处运行”，一是因为 JVM 针对各种操作系统、平台都进行了定制，二是因为无论在什么平台，都可以编译生成固定格式的字节码（.class 文件）供 JVM 使用。因此，也可以看出字节码对于 Java 生态的重要性。之所以被称之为字节码，是因为字节码文件由十六进制值组成，而 JVM 以两个十六进制值为一组，即以字节为单位进行读取。在 Java 中一般是用 javac 命令编译源代码为字节码文件，一个 .java 文件从编译到运行的示例如图 1 所示。

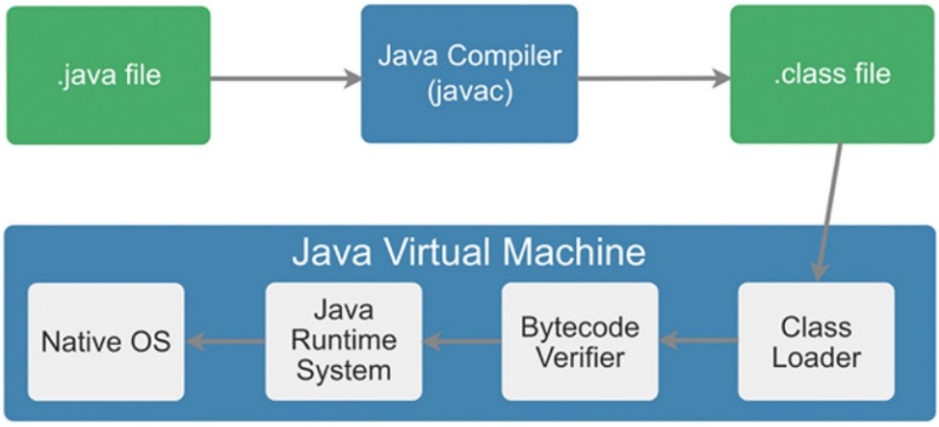


图 1　Java 运行示意图

对于开发人员，了解字节码可以更准确、直观地理解 Java 语言中更深层次的东西，比如通过字节码，可以很直观地看到 Volatile 关键字如何在字节码上生效。另外，字节码增强技术在 Spring AOP、各种 ORM 框架、热部署中的应用屡见不鲜，深入理解其原理对于我们来说大有裨益。除此之外，由于 JVM 规范的存在，只要最

终可以生成符合规范的字节码就可以在 JVM 上运行，因此这就给了各种运行在 JVM 上的语言（如 Scala、Groovy、Kotlin）一种契机，可以扩展 Java 所没有的特性或者实现各种语法糖。理解字节码后再学习这些语言，可以“逆流而上”，从字节码视角看它的设计思路，学习起来也“易如反掌”。

本文重点着眼于字节码增强技术，从字节码开始逐层向上，由 JVM 字节码操作集合到 Java 中操作字节码的框架，再到我们熟悉的各类框架原理及应用，也都会一一进行介绍。

1.2 字节码结构

.java 文件通过 javac 编译后将得到一个 .class 文件，比如编写一个简单的 ByteCodeDemo 类，如下图 2 的左侧部分：

```
1  public class ByteCodeDemo {
2      private int a = 1;
3
4      public int add() {
5          int b = 2;
6          int c = a + b;
7          System.out.println(c);
8          return c;
9      }
10 }
11
```

```
CA FE BA BE 00 00 00 34 00 24 0A 00 06 00 16 09 00 05 00 17 09 00 18 00 19 0A 00
1A 00 1B 07 00 1C 07 00 1D 01 00 01 61 01 00 01 49 01 00 06 3C 69 6E 69 74 3E 01
00 03 28 29 56 01 00 04 43 6F 64 65 01 00 0F 4C 69 6E 65 4E 75 6D 62 65 72 54 61
62 6C 65 01 00 12 4C 6F 63 61 6C 56 61 72 69 61 62 6C 65 54 61 62 6C 65 01 00 04
74 68 69 73 01 00 1F 4C 6D 65 69 74 75 61 6E 2F 62 79 74 65 63 6F 64 65 2F 42 79
74 65 43 6F 64 65 44 65 6D 6F 3B 01 00 03 61 64 64 01 00 03 28 29 49 01 00 01 62
01 00 01 63 01 00 0A 53 6F 75 72 63 65 46 69 6C 65 01 00 11 42 79 74 65 43 6F 64
65 44 65 6D 6F 2E 6A 61 76 61 0C 00 09 00 0A 0C 00 07 00 08 07 00 1E 0C 00 1F 00
20 07 00 21 0C 00 22 00 23 01 00 1D 6D 65 69 74 75 61 6E 2F 62 79 74 65 63 6F 64
65 2F 42 79 74 65 43 6F 64 65 44 65 6D 6F 01 00 10 6A 61 76 61 2F 6C 61 6E 67 2F
4F 62 6A 65 63 74 01 00 10 6A 61 76 61 2F 6C 61 6E 67 2F 53 79 73 74 65 6D 01 00
03 6F 75 74 01 00 15 4C 6A 61 76 61 2F 69 6F 2F 50 72 69 6E 74 53 74 72 65 61 6D
3B 01 00 13 6A 61 76 61 2F 69 6F 2F 50 72 69 6E 74 53 74 72 65 61 6D 01 00 07 70
72 69 6E 74 6C 6E 01 00 04 28 49 29 56 00 21 00 05 00 06 00 00 00 01 00 02 00 07
00 08 00 00 00 02 00 01 00 09 00 0A 00 01 00 0B 00 00 00 38 00 02 00 01 00 00 00
0A 2A B7 00 01 2A 04 B5 00 02 B1 00 00 00 02 00 0C 00 00 00 0A 00 02 00 00 00 07
00 04 00 08 00 0D 00 00 00 0C 00 01 00 00 00 0A 00 0E 00 0F 00 00 00 01 00 10 00
11 00 01 00 0B 00 00 00 5C 00 02 00 03 00 00 00 12 05 3C 2A B4 00 02 1B 60 3D B2
00 03 1C B6 00 04 1C AC 00 00 00 02 00 0C 00 00 00 12 00 04 00 00 00 0B 00 02 00
0C 00 09 00 0D 00 10 00 0E 00 0D 00 00 00 20 00 03 00 00 00 12 00 0E 00 0F 00 00
00 02 00 10 00 12 00 08 00 01 00 09 00 09 00 13 00 08 00 02 00 01 00 14 00 00 00
02 00 15
```

图 2　示例代码（左侧）及对应的字节码（右侧）

编译后生成 ByteCodeDemo.class 文件，打开后是一堆十六进制数，按字节为单位进行分割后展示如图 2 右侧部分所示。上文提及过，JVM 对于字节码是有规范要求的，那么看似杂乱的十六进制符合什么结构呢？ JVM 规范要求每一个字节码文件都要由十部分按照固定的顺序组成，整体结构如图 3 所示。接下来我们将一一介绍这十部分：

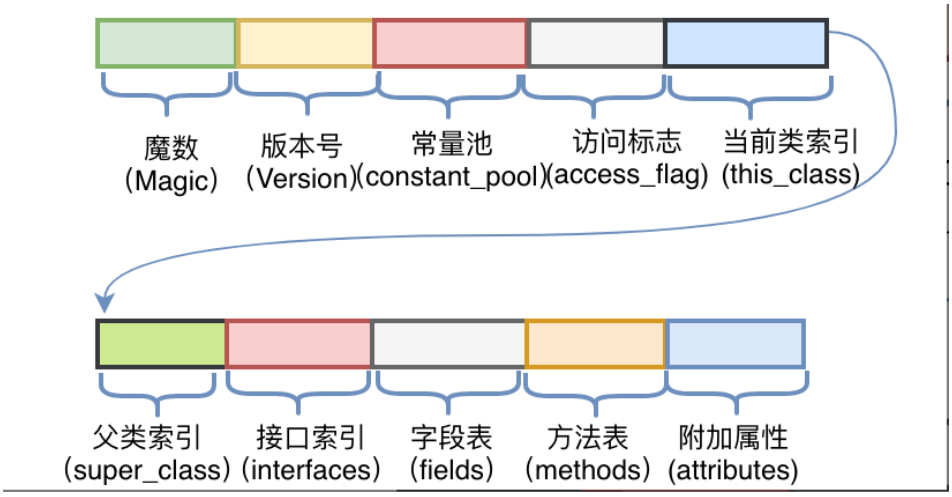


图 3　JVM 规定的字节码结构

（1）魔数（Magic Number）

所有的 .class 文件的前四个字节都是魔数，魔数的固定值为：0xCAFEBABE。魔数放在文件开头，JVM 可以根据文件的开头来判断这个文件是否可能是一个 .class 文件，如果是，才会继续进行之后的操作。

有趣的是，魔数的固定值是 Java 之父 James Gosling 制定的，为 CafeBabe（咖啡宝贝），而 Java 的图标为一杯咖啡。

（2）版本号

版本号为魔数之后的 4 个字节，前两个字节表示次版本号（Minor Version），后两个字节表示主版本号（Major Version）。上图 2 中版本号为“00 00 00 34”，次版本号转化为十进制为 0，主版本号转化为十进制为 52，在 Oracle 官网中查询序号 52 对应的主版本号为 1.8，所以编译该文件的 Java 版本号为 1.8.0。

（3）常量池（Constant Pool）

紧接着主版本号之后的字节为常量池入口。常量池中存储两类常量：字面量与符号引用。字面量为代码中声明为 Final 的常量值，符号引用如类和接口的全局限定名、字段的名称和描述符、方法的名称和描述符。常量池整体上分为两部分：常量池计数器以及常量池数据区，如下图 4 所示。

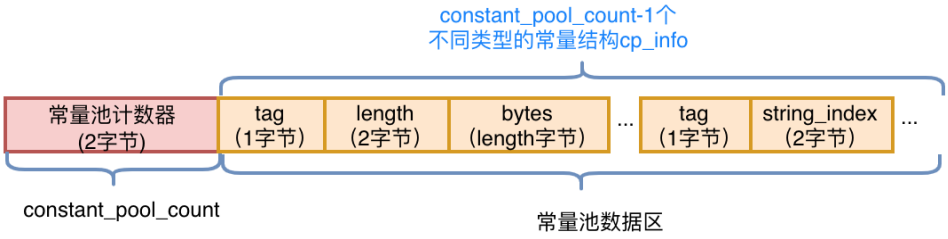


图 4　常量池的结构

- 常量池计数器（constant_pool_count）：由于常量的数量不固定，所以需要先放置两个字节来表示常量池容量计数值。图 2 中示例代码的字节码前 10 个字节如下图 5 所示，将十六进制的 24 转化为十进制值为 36，排除掉下标“0”，也就是说，这个类文件中共有 35 个常量。

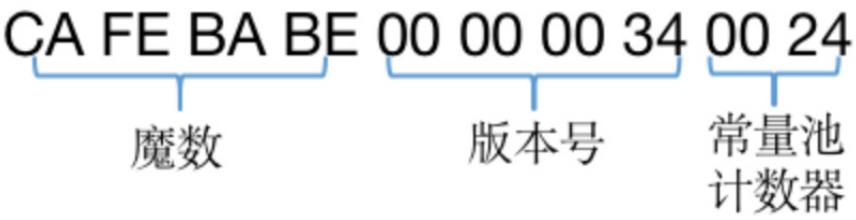


图 5　前十个字节及含义

- 常量池数据区：数据区是由（constant_pool_count−1）个 cp_info 结构组成，一个 cp_info 结构对应一个常量。在字节码中共有 14 种类型的 cp_info（如下图 6 所示），每种类型的结构都是固定的。

常量	项目	类型	描述
CONSTANT_Utf8_info	tag	1byte	值为 1
	length	2byte	UTF-8 编码的字符串占用的字符数
	bytes	1byte	长度为 length 的 UTF-8 编码的字符串
CONSTANT_Integer_info	tag	1byte	值为 3
	bytes	4byte	按照高位在前存储的 int 值
CONSTANT_Float_info	tag	1byte	值为 4
	bytes	4byte	按照高位在前存储的 float 值
CONSTANT_Long_info	tag	1byte	值为 5
	bytes	8byte	按照高位在前存储 long 值
CONSTANT_Double_info	tag	1byte	值为 6
	bytes	8byte	按照高位在前存储 double 值
CONSTANT_Class_info	tag	1byte	值为 7
	index	2byte	指向全限定名常量项的索引
CONSTANT_String_info	tag	1byte	值为 8
	index	2byte	指向字符串字面量的索引
CONSTANT_Fieldref_info	tag	1byte	值为 9
	index	2byte	指向声明字段的类或者接口描述符 CONSTANT_Class_info 的索引项
	index	2byte	指向字段描述符 CONSTANT_NameAndType 的索引项
CONSTANT_Methodref_info	tag	1byte	值为 10
	index	2byte	指向声明方法的类描述符 CONSTANT_Class_info 的索引项
	index	2byte	指向名称及类型描述符 CONSTANT_NameAndType 的索引项
CONSTANT_InterfaceMethodref_info	tag	1byte	值为 11
	index	2byte	指向声明方法的接口描述符 CONSTANT_Class_info 的索引项
	index	2byte	指向名称及类型描述符 CONSTANT_NameAndType 的索引项
CONSTANT_NameAndType_info	tag	1byte	值为 12
	index	2byte	指向该字段或方法名称常量项的索引
	index	2byte	指向该字段或方法描述符常量项的索引
CONSTANT_MethodHandle_info	tag	1byte	值为 15
	reference_kind	1byte	值必须在 1~9 之间，它决定了方法句柄的类型方法句柄类型的值表示方法句柄的字节码行为
	reference_index	2byte	值必须是对常量池的有效索引
CONSTANT_MethodType_info	tag	1byte	值为 16
	descriptor_index	2byte	值必须是对常量池的有效索引，常量池在该索引处的项必须是 CONSTANT_Utf8_info 结构，表示方法的描述符
CONSTANT_InvokeDynamic_info	tag	1byte	值为 18
	bootstarp_method_attr_index	2byte	值必须是对当前 Class 文件中引导方法表的 bootstrap_methods[]数组的有效索引
	name_and_type_index	2byte	值必须是对当前常量池的有效索引，常量池在该索引处的项必须是 CONSTANT_NameAndType_info 结构，表示方法名和方法描述符

图 6　各类型的 cp_info

具体以 CONSTANT_utf8_info 为例，它的结构如下图 7 左侧所示。首先一个字节“tag”，它的值取自上图 6 中对应项的 Tag，由于它的类型是 utf8_info，所以值为“01”。接下来两个字节标识该字符串的长度 Length，然后 Length 个字节为这个字符串具体的值。从图 2 中的字节码摘取一个 cp_info 结构，如下图 7 右侧所示。将它翻译过来后，其含义为：该常量类型为 utf8 字符串，长度为一字节，数据为“a”。

长度	名称	值
1字节	tag	01 对应图5中utf8_info的“标志”栏中的值
2字节	length	该utf8字符串的长度
length个字节	bytes	length个字节的具体数据

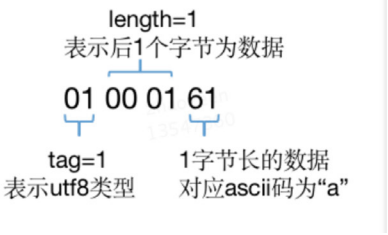


图 7　CONSTANT_utf8_info 的结构（左）及示例（右）

其他类型的 cp_info 结构在本文不再赘述，整体结构大同小异，都是先通过 Tag 来标识类型，然后后续 n 个字节来描述长度和（或）数据。先知其所以然，以后可以通过 javap -verbose ByteCodeDemo 命令，查看 JVM 反编译后的完整常量池，如下图 8 所示。可以看到反编译结果将每一个 cp_info 结构的类型和值都很明确地呈现了出来。

```
 #1 = Methodref          #6.#22         // java/lang/Object."<init>":()V
 #2 = Fieldref           #5.#23         // meituan/bytecode/ByteCodeDemo.a:I
 #3 = Fieldref           #24.#25        // java/lang/System.out:Ljava/io/PrintStream;
 #4 = Methodref          #26.#27        // java/io/PrintStream.println:(I)V
 #5 = Class              #28            // meituan/bytecode/ByteCodeDemo
 #6 = Class              #29            // java/lang/Object
 #7 = Utf8               a
 #8 = Utf8               I
 #9 = Utf8               <init>
#10 = Utf8               ()V
#11 = Utf8               Code
#12 = Utf8               LineNumberTable
#13 = Utf8               LocalVariableTable
#14 = Utf8               this
#15 = Utf8               Lmeituan/bytecode/ByteCodeDemo;
#16 = Utf8               add
#17 = Utf8               ()I
#18 = Utf8               b
#19 = Utf8               c
#20 = Utf8               SourceFile
#21 = Utf8               ByteCodeDemo.java
#22 = NameAndType        #9:#10         // "<init>":()V
#23 = NameAndType        #7:#8          // a:I
#24 = Class              #30            // java/lang/System
#25 = NameAndType        #31:#32        // out:Ljava/io/PrintStream;
#26 = Class              #33            // java/io/PrintStream
#27 = NameAndType        #34:#35        // println:(I)V
#28 = Utf8               meituan/bytecode/ByteCodeDemo
#29 = Utf8               java/lang/Object
#30 = Utf8               java/lang/System
#31 = Utf8               out
#32 = Utf8               Ljava/io/PrintStream;
#33 = Utf8               java/io/PrintStream
#34 = Utf8               println
```

图 8　常量池反编译结果

（4）访问标志

常量池结束之后的两个字节，描述该 Class 是类还是接口，以及是否被 Public、Abstract、Final 等修饰符修饰。JVM 规范规定了如下图 9 的访问标志（Access_Flag）。需要注意的是，JVM 并没有穷举所有的访问标志，而是使用按位或操作来进行描述的，比如某个类的修饰符为 Public Final，则对应的访问修饰符的值为 ACC_PUBLIC | ACC_FINAL，即 0x0001 | 0x0010=0x0011。

标志名称	标志值	含义
ACC_PUBLIC	0x0001	字段是否为public
ACC_PRIVATE	0x0002	字段是否为private
ACC_PROTECTED	0x0004	字段是否为protected
ACC_STATIC	0x0008	字段是否为static
ACC_FINAL	0x0010	字段是否为final
ACC_VOLATILE	0x0040	字段是否为volatile
ACC_TRANSTENT	0x0080	字段是否为transient
ACC_SYNCHETIC	0x1000	字段是否为由编译器自动产生
ACC_ENUM	0x4000	字段是否为enum

图 9　访问标志

（5）当前类名

访问标志后的两个字节，描述的是当前类的全限定名。这两个字节保存的值为常量池中的索引值，根据索引值就能在常量池中找到这个类的全限定名。

（6）父类名称

当前类名后的两个字节，描述父类的全限定名，同上，保存的也是常量池中的索引值。

（7）接口信息

父类名称后为两字节的接口计数器，描述了该类或父类实现的接口数量。紧接着的 n 个字节是所有接口名称的字符串常量的索引值。

（8）字段表

字段表用于描述类和接口中声明的变量，包含类级别的变量以及实例变量，但是不包含方法内部声明的局部变量。字段表也分为两部分，第一部分为两个字节，描述字段个数；第二部分是每个字段的详细信息 fields_info。字段表结构如下图所示：

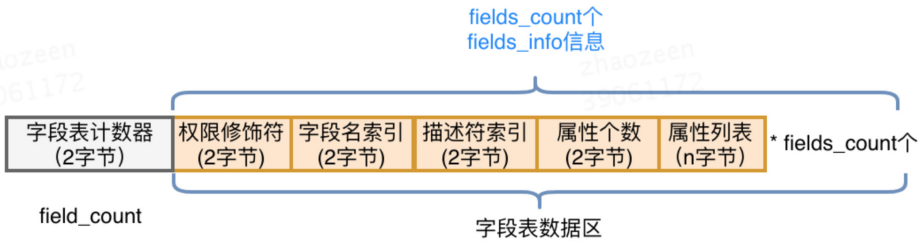


图 10　字段表结构

以图 2 中字节码的字段表为例，如下图 11 所示。其中字段的访问标志查图 9，0002 对应为 Private。通过索引下标在图 8 中常量池分别得到字段名为“a”，描述符为“I”（代表 int）。综上，就可以唯一确定出一个类中声明的变量 private int a。

图 11　字段表示例

（9）方法表

字段表结束后为方法表，方法表也是由两部分组成，第一部分为两个字节描述方法的个数；第二部分为每个方法的详细信息。方法的详细信息较为复杂，包括方法的访问标志、方法名、方法的描述符以及方法的属性，如下图所示：

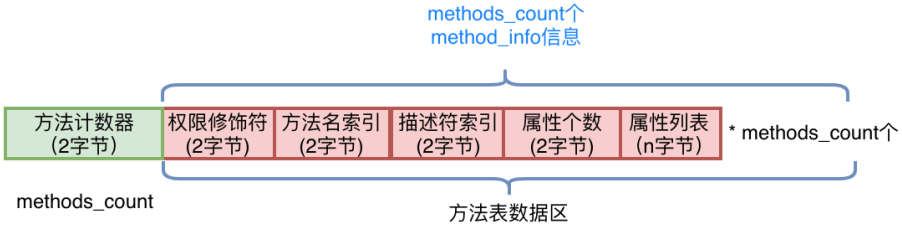


图 12　方法表结构

方法的权限修饰符依然可以通过图 9 的值查询得到，方法名和方法的描述符都是常量池中的索引值，可以通过索引值在常量池中找到。而“方法的属性”这一部分较为复杂，直接借助 javap -verbose 将其反编译为人可以读懂的信息进行解读，如图 13 所示。可以看到属性中包括以下三个部分：

```
public int add();                                   //方法名
  descriptor: ()I                                   //方法的描述符
  flags: ACC_PUBLIC                                 //方法的访问标志
  Code:                                             //Code开始
    stack=2, locals=3, args_size=1                  //最大栈深度、局部变量数、参数列表size
       0: iconst_2                                  //0-17为.java中的代码对应的操作码
       1: istore_1
       2: aload_0
       3: getfield      #2                         // 引用常量池#2的值: Field a:I
       6: iload_1
       7: iadd
       8: istore_2
       9: getstatic     #3                        // 引用常量池#3的值: Field
                                                  //java/lang/System.out:Ljava/io/PrintStrea
      12: iload_2
      13: invokevirtual #4                         //引用常量池#4的值
                                                   //Method java/io/PrintStream.println:(I)V
      16: iload_2
      17: ireturn
    LineNumberTable:                               //行号表，将上面的操作码与.java中的行号做对应
      line 11: 0
      line 12: 2
      line 13: 9
      line 14: 16
    LocalVariableTable:                            //本地变量表，包含局部变量的定义
      Start  Length  Slot  Name   Signature
          0      18     0  this   Lmeituan/bytecode/ByteCodeDemo;    //slot=0为this
          2      16     1     b   I
          9       9     2     c   I
```

图 13　反编译后的方法表

- “Code 区”：源代码对应的 JVM 指令操作码，在进行字节码增强时重点操作的就是“Code 区”这一部分。
- “LineNumberTable”：行号表，将 Code 区的操作码和源代码中的行号对应，Debug 时会起到作用（源代码走一行，需要走多少个 JVM 指令操作码）。
- “LocalVariableTable”：本地变量表，包含 This 和局部变量，之所以可以在每一个方法内部都可以调用 This，是因为 JVM 将 This 作为每一个方法的第一个参数隐式进行传入。当然，这是针对非 Static 方法而言。

（10）附加属性表

字节码的最后一部分，该项存放了在该文件中类或接口所定义属性的基本信息。

1.3 字节码操作集合

在上图 13 中，Code 区的红色编号 0 ~ 17，就是 .java 中的方法源代码编译后让 JVM 真正执行的操作码。为了帮助人们理解，反编译后看到的是十六进制操作码所对应的助记符，十六进制值操作码与助记符的对应关系，以及每一个操作码的用处可以查看 Oracle 官方文档进行了解，在需要用到时进行查阅即可。比如上图中第一个助记符为 iconst_2，对应到图 2 中的字节码为 0x05，用处是将 int 值 2 压入操作数栈中。以此类推，对 0~17 的助记符理解后，就是完整的 add() 方法的实现。

1.4 操作数栈和字节码

JVM 的指令集是基于栈而不是寄存器，基于栈可以具备很好的跨平台性（因为寄存器指令集往往和硬件挂钩），但缺点在于，要完成同样的操作，基于栈的实现需要更多指令才能完成（因为栈只是一个 FILO 结构，需要频繁压栈出栈）。另外，由于栈是在内存实现的，而寄存器是在 CPU 的高速缓存区，相较而言，基于栈的速度要慢很多，这也是为了跨平台性而做出的牺牲。

我们在上文所说的操作码或者操作集合，其实控制的就是这个 JVM 的操作数栈。为了更直观地感受操作码是如何控制操作数栈的，以及理解常量池、变量表的作用，将 add() 方法的对操作数栈的操作制作为 GIF，如下图 14 所示，图中仅截取了常量池中被引用的部分，以指令 iconst_2 开始到 ireturn 结束，与图 13 中 Code 区

0~17 的指令一一对应：

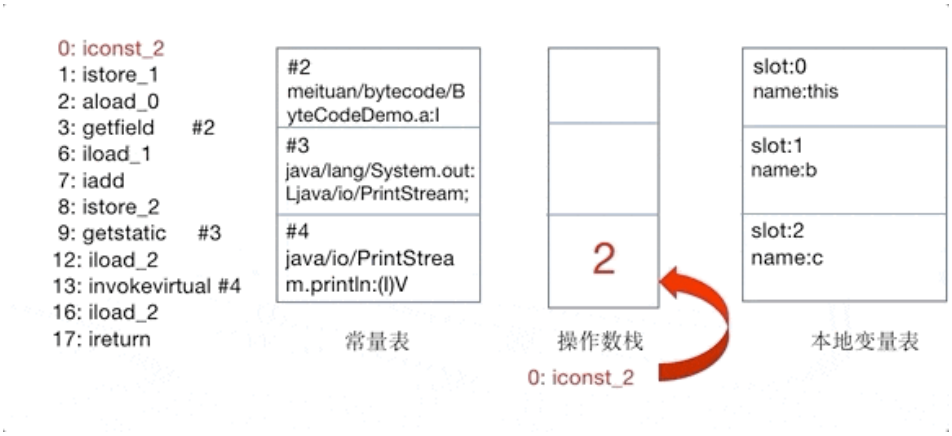


图 14　控制操作数栈示意图

1.5　查看字节码工具

如果每次查看反编译后的字节码都使用 javap 命令的话，好非常繁琐。这里推荐一个 Idea 插件：jclasslib。使用效果如图 15 所示，代码编译后在菜单栏”View”中选择”Show Bytecode With jclasslib”，可以很直观地看到当前字节码文件的类信息、常量池、方法区等信息。

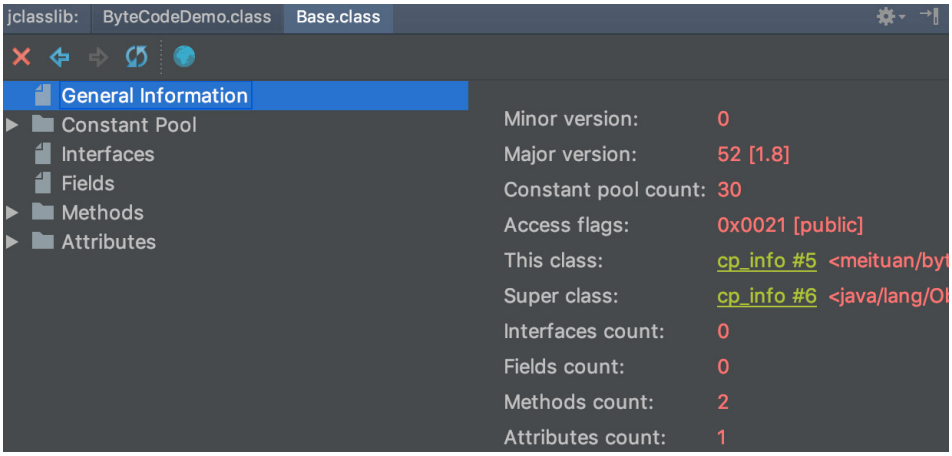


图 15　jclasslib 查看字节码

2. 字节码增强

在上文中，着重介绍了字节码的结构，这为我们了解字节码增强技术的实现打下了基础。字节码增强技术就是一类对现有字节码进行修改或者动态生成全新字节码文件的技术。接下来，我们将从最直接操纵字节码的实现方式开始深入进行剖析。

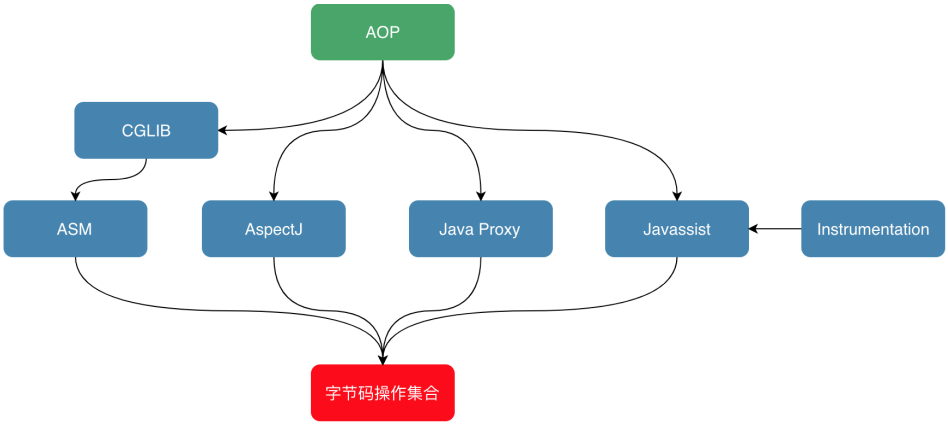


图 16　字节码增强技术

2.1　ASM

对于需要手动操纵字节码的需求，可以使用 ASM，它可以直接生产 .class 字节码文件，也可以在类被加载入 JVM 之前动态修改类行为（如下图 17 所示）。ASM 的应用场景有 AOP（Cglib 就是基于 ASM）、热部署、修改其他 jar 包中的类等。当然，涉及到如此底层的步骤，实现起来也比较麻烦。接下来，本文将介绍 ASM 的两种 API，并用 ASM 来实现一个比较粗糙的 AOP。但在此之前，为了让大家更快地理解 ASM 的处理流程，强烈建议读者先对访问者模式进行了解。简单来说，访问者模式主要用于修改或操作一些数据结构比较稳定的数据，而通过第一章，我们知道字节码文件的结构是由 JVM 固定的，所以很适合利用访问者模式对字节码文件进行修改。

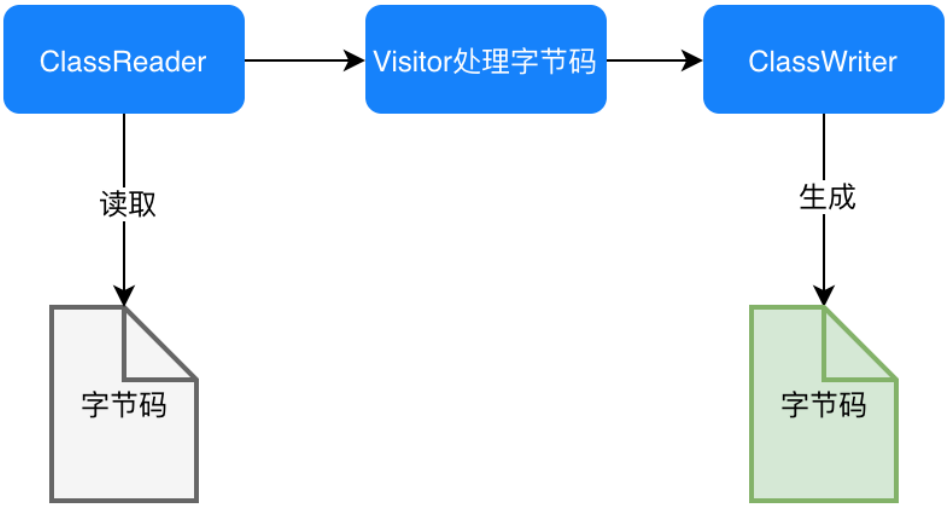


图 17　ASM 修改字节码

2.1.1　ASM API

2.1.1.1　核心 API

ASM Core API 可以类比解析 XML 文件中的 SAX 方式，不需要把这个类的整个结构读取进来，就可以用流式的方法来处理字节码文件。好处是非常节约内存，但是编程难度较大。然而出于性能考虑，一般情况下编程都使用 Core API。在 Core API 中有以下几个关键类：

- ClassReader：用于读取已经编译好的 .class 文件。
- ClassWriter：用于重新构建编译后的类，如修改类名、属性以及方法，也可以生成新的类的字节码文件。
- 各种 Visitor 类：如上所述，CoreAPI 根据字节码从上到下依次处理，对于字节码文件中不同的区域有不同的 Visitor，比如用于访问方法的 MethodVisitor、用于访问类变量的 FieldVisitor、用于访问注解的 AnnotationVisitor 等。为了实现 AOP，重点要使用的是 MethodVisitor。

2.1.1.2　树形 API

ASM Tree API 可以类比解析 XML 文件中的 DOM 方式，把整个类的结构读

取到内存中，缺点是消耗内存多，但是编程比较简单。TreeApi 不同于 CoreAPI，TreeAPI 通过各种 Node 类来映射字节码的各个区域，类比 DOM 节点，就可以很好地理解这种编程方式。

2.1.2 直接利用 ASM 实现 AOP

利用 ASM 的 CoreAPI 来增强类。这里不纠结于 AOP 的专业名词如切片、通知，只实现在方法调用前、后增加逻辑，通俗易懂且方便理解。首先定义需要被增强的 Base 类：其中只包含一个 process() 方法，方法内输出一行“process”。增强后，我们期望的是，方法执行前输出“start”，之后输出”end”。

```
public class Base {
    public void process(){
        System.out.println("process");
    }
}
```

为了利用 ASM 实现 AOP，需要定义两个类：一个是 MyClassVisitor 类，用于对字节码的 visit 以及修改；另一个是 Generator 类，在这个类中定义 ClassReader 和 ClassWriter，其中的逻辑是，classReader 读取字节码，然后交给 MyClassVisitor 类处理，处理完成后由 ClassWriter 写字节码并将旧的字节码替换掉。Generator 类较简单，我们先看一下它的实现，如下所示，然后重点解释 MyClassVisitor 类。

```
import org.objectweb.asm.ClassReader;
import org.objectweb.asm.ClassVisitor;
import org.objectweb.asm.ClassWriter;

public class Generator {
    public static void main(String[] args) throws Exception {
            //读取
        ClassReader classReader = new ClassReader("meituan/bytecode/asm/
Base");
        ClassWriter classWriter = new ClassWriter(ClassWriter.COMPUTE_
MAXS);
        //处理
        ClassVisitor classVisitor = new MyClassVisitor(classWriter);
        classReader.accept(classVisitor, ClassReader.SKIP_DEBUG);
```

```
        byte[] data = classWriter.toByteArray();
        // 输出
        File f = new File("operation-server/target/classes/meituan/
bytecode/asm/Base.class");
        FileOutputStream fout = new FileOutputStream(f);
        fout.write(data);
        fout.close();
        System.out.println("now generator cc success!!!!!");
    }
}
```

MyClassVisitor 继承自 ClassVisitor，用于对字节码的观察。它还包含一个内部类 MyMethodVisitor，继承自 MethodVisitor 用于对类内方法的观察，它的整体代码如下：

```
import org.objectweb.asm.ClassVisitor;
import org.objectweb.asm.MethodVisitor;
import org.objectweb.asm.Opcodes;

public class MyClassVisitor extends ClassVisitor implements Opcodes {
    public MyClassVisitor(ClassVisitor cv) {
        super(ASM5, cv);
    }
    @Override
    public void visit(int version, int access, String name, String
signature,
                      String superName, String[] interfaces) {
        cv.visit(version, access, name, signature, superName,
interfaces);
    }
    @Override
    public MethodVisitor visitMethod(int access, String name, String
desc, String signature,
String[] exceptions) {
        MethodVisitor mv = cv.visitMethod(access, name, desc, signature,
                exceptions);
        //Base 类中有两个方法：无参构造以及 process 方法，这里不增强构造方法
        if (!name.equals("<init>") && mv != null) {
            mv = new MyMethodVisitor(mv);
        }
        return mv;
    }
    class MyMethodVisitor extends MethodVisitor implements Opcodes {
        public MyMethodVisitor(MethodVisitor mv) {
            super(Opcodes.ASM5, mv);
        }
```

```
        @Override
        public void visitCode() {
            super.visitCode();
            mv.visitFieldInsn(GETSTATIC, "java/lang/System", "out",
"Ljava/io/PrintStream;");
            mv.visitLdcInsn("start");
            mv.visitMethodInsn(INVOKEVIRTUAL, "java/io/PrintStream",
"println", "(Ljava/lang/
String;)V", false);
        }
        @Override
        public void visitInsn(int opcode) {
            if ((opcode >= Opcodes.IRETURN && opcode <= Opcodes.RETURN)
                    || opcode == Opcodes.ATHROW) {
                // 方法在返回之前，打印 "end"
                mv.visitFieldInsn(GETSTATIC, "java/lang/System", "out",
"Ljava/io/PrintStream;");
                mv.visitLdcInsn("end");
                mv.visitMethodInsn(INVOKEVIRTUAL, "java/io/PrintStream",
"println", "(Ljava/lang/
String;)V", false);
            }
            mv.visitInsn(opcode);
        }
    }
}
```

利用这个类就可以实现对字节码的修改。详细解读其中的代码，对字节码做修改的步骤是：

- 首先通过 MyClassVisitor 类中的 visitMethod 方法，判断当前字节码读到哪一个方法了。跳过构造方法 <init> 后，将需要被增强的方法交给内部类 MyMethodVisitor 来进行处理。
- 接下来，进入内部类 MyMethodVisitor 中的 visitCode 方法，它会在 ASM 开始访问某一个方法的 Code 区时被调用，重写 visitCode 方法，将 AOP 中的前置逻辑就放在这里。
- MyMethodVisitor 继续读取字节码指令，每当 ASM 访问到无参数指令时，都会调用 MyMethodVisitor 中的 visitInsn 方法。我们判断了当前指令是否为无参数的“return”指令，如果是就在它的前面添加一些指令，也就是将 AOP

的后置逻辑放在该方法中。

- 综上，重写 MyMethodVisitor 中的两个方法，就可以实现 AOP 了，而重写方法时就需要用 ASM 的写法，手动写入或者修改字节码。通过调用 methodVisitor 的 visitXXXXInsn() 方法就可以实现字节码的插入，XXXX 对应相应的操作码助记符类型，比如 mv.visitLdcInsn(“end”) 对应的操作码就是 ldc“end”，即将字符串“end”压入栈。

完成这两个 visitor 类后，运行 Generator 中的 main 方法完成对 Base 类的字节码增强，增强后的结果可以在编译后的 target 文件夹中找到 Base.class 文件进行查看，可以看到反编译后的代码已经改变了（如图 18 左侧所示）。然后写一个测试类 MyTest，在其中 new Base()，并调用 base.process() 方法，可以看到下图右侧所示的 AOP 实现效果：

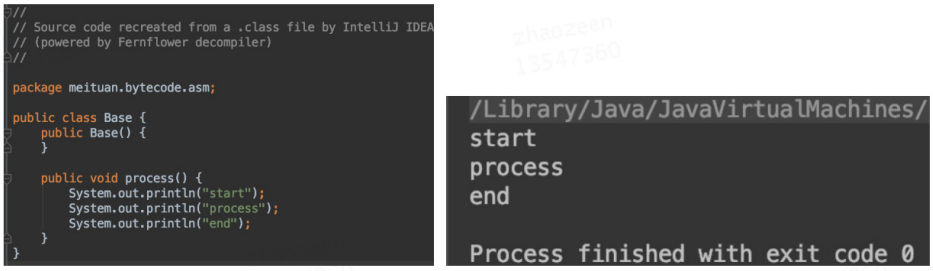


图 18　ASM 实现 AOP 的效果

2.1.3　ASM 工具

利用 ASM 手写字节码时，需要利用一系列 visitXXXXInsn() 方法来写对应的助记符，所以需要先将每一行源代码转化为一个个的助记符，然后通过 ASM 的语法转换为 visitXXXXInsn() 这种写法。第一步将源码转化为助记符就已经够麻烦了，不熟悉字节码操作集合的话，需要我们将代码编译后再反编译，才能得到源代码对应的助记符。第二步利用 ASM 写字节码时，如何传参也很令人头疼。ASM 社区也知道这两个问题，所以提供了工具 ASM ByteCode Outline。

安装后，右键选择“Show Bytecode Outline”，在新标签页中选择“ASMi-

fied”这个 tab，如图 19 所示，就可以看到这个类中的代码对应的 ASM 写法了。图中上下两个红框分别对应 AOP 中的前置逻辑于后置逻辑，将这两块直接复制到 visitor 中的 visitMethod() 以及 visitInsn() 方法中，就可以了。

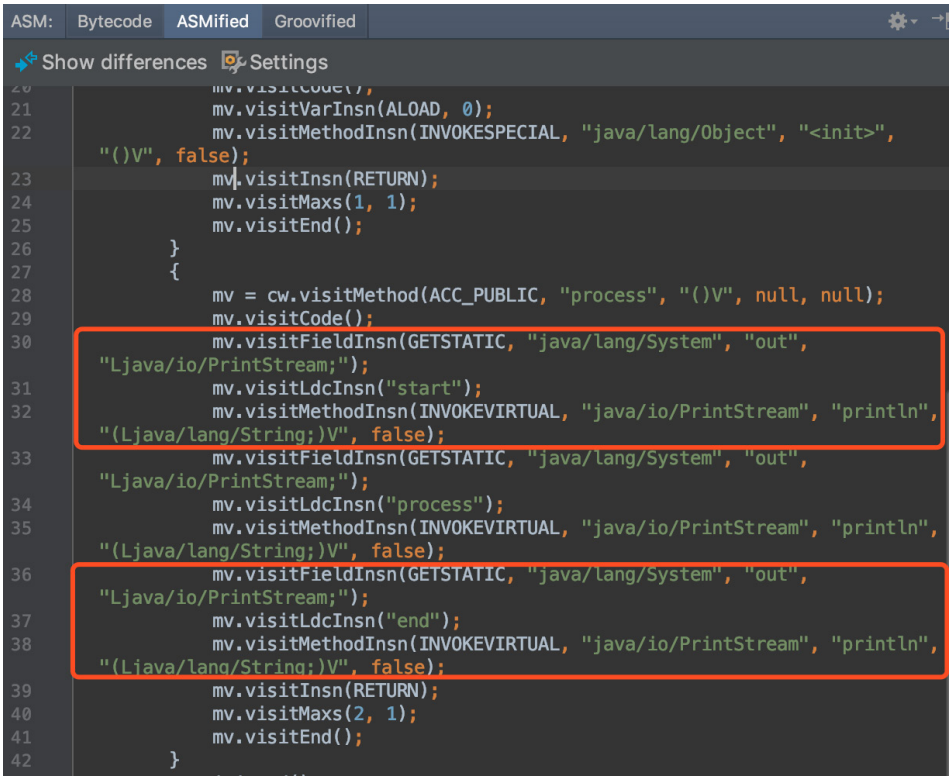


图 19　ASM Bytecode Outline

2.2　Javassist

ASM 是在指令层次上操作字节码的，阅读上文后，我们的直观感受是在指令层次上操作字节码的框架实现起来比较晦涩。故除此之外，我们再简单介绍另外一类框架：强调源代码层次操作字节码的框架 Javassist。

利用 Javassist 实现字节码增强时，可以无须关注字节码刻板的结构，其优点就在于编程简单。直接使用 java 编码的形式，而不需要了解虚拟机指令，就能动态改变类的结构或者动态生成类。其中最重要的是 ClassPool、CtClass、CtMethod、

CtField 这四个类：

- CtClass（compile-time class）：编译时类信息，它是一个 class 文件在代码中的抽象表现形式，可以通过一个类的全限定名来获取一个 CtClass 对象，用来表示这个类文件。
- ClassPool：从开发视角来看，ClassPool 是一张保存 CtClass 信息的 HashTable，key 为类名，value 为类名对应的 CtClass 对象。当我们需要对某个类进行修改时，就是通过 pool.getCtClass(“className”) 方法从 pool 中获取到相应的 CtClass。
- CtMethod、CtField：这两个比较好理解，对应的是类中的方法和属性。

了解这四个类后，我们可以写一个小 Demo 来展示 Javassist 简单、快速的特点。我们依然是对 Base 中的 process() 方法做增强，在方法调用前后分别输出”start” 和”end”，实现代码如下。我们需要做的就是从 pool 中获取到相应的 CtClass 对象和其中的方法，然后执行 method.insertBefore 和 insertAfter 方法，参数为要插入的 Java 代码，再以字符串的形式传入即可，实现起来也极为简单。

```
import com.meituan.mtrace.agent.javassist.*;

public class JavassistTest {
    public static void main(String[] args) throws NotFoundException,
CannotCompileException, IllegalAccessException, InstantiationException,
IOException {
        ClassPool cp = ClassPool.getDefault();
        CtClass cc = cp.get("meituan.bytecode.javassist.Base");
        CtMethod m = cc.getDeclaredMethod("process");
        m.insertBefore("{ System.out.println(\"start\"); }");
        m.insertAfter("{ System.out.println(\"end\"); }");
        Class c = cc.toClass();
        cc.writeFile("/Users/zen/projects");
        Base h = (Base)c.newInstance();
        h.process();
    }
}
```

3. 运行时类的重载

3.1 问题引出

上一章重点介绍了两种不同类型的字节码操作框架，且都利用它们实现了较为粗糙的 AOP。其实，为了方便大家理解字节码增强技术，在上文中我们避重就轻将 ASM 实现 AOP 的过程分为了两个 main 方法：第一个是利用 MyClassVisitor 对已编译好的 class 文件进行修改，第二个是 new 对象并调用。这期间并不涉及到 JVM 运行时对类的重加载，而是在第一个 main 方法中，通过 ASM 对已编译类的字节码进行替换，在第二个 main 方法中，直接使用已替换好的新类信息。另外在 Javassist 的实现中，我们也只加载了一次 Base 类，也不涉及到运行时重加载类。

如果我们在一个 JVM 中，先加载了一个类，然后又对其进行字节码增强并重新加载会发生什么呢？模拟这种情况，只需要我们在上文中 Javassist 的 Demo 中 main() 方法的第一行添加 Base b=new Base()，即在增强前就先让 JVM 加载 Base 类，然后在执行到 c.toClass() 方法时会抛出错误，如下图 20 所示。跟进 c.toClass() 方法中，我们会发现它是在最后调用了 ClassLoader 的 native 方法 defineClass() 时报错。也就是说，JVM 是不允许在运行时动态重载一个类的。

图 20　运行时重复 load 类的错误信息

显然，如果只能在类加载前对类进行强化，那字节码增强技术的使用场景就变得很窄了。我们期望的效果是：在一个持续运行并已经加载了所有类的 JVM 中，还能利用字节码增强技术对其中的类行为做替换并重新加载。为了模拟这种情况，我们将 Base 类做改写，在其中编写 main 方法，每五秒调用一次 process() 方法，在 process() 方法中输出一行“process”。

我们的目的就是，在 JVM 运行中的时候，将 process() 方法做替换，在其前后分别打印“start”和“end”。也就是在运行中时，每五秒打印的内容由”process”

变为打印”start process end”。那如何解决 JVM 不允许运行时重加载类信息的问题呢？为了达到这个目的，我们接下来一一来介绍需要借助的 Java 类库。

```
import java.lang.management.ManagementFactory;

public class Base {
    public static void main(String[] args) {
        String name = ManagementFactory.getRuntimeMXBean().getName();
        String s = name.split("@")[0];
        //打印当前 Pid
        System.out.println("pid:"+s);
        while (true) {
            try {
                Thread.sleep(5000L);
            } catch (Exception e) {
                break;
            }
            process();
        }
    }

    public static void process() {
        System.out.println("process");
    }
}
```

3.2 Instrument

instrument 是 JVM 提供的一个可以修改已加载类的类库，专门为 Java 语言编写的插桩服务提供支持。它需要依赖 JVMTI 的 Attach API 机制实现，JVMTI 这一部分，我们将在下一小节进行介绍。在 JDK 1.6 以前，instrument 只能在 JVM 刚启动开始加载类时生效，而在 JDK 1.6 之后，instrument 支持了在运行时对类定义的修改。要使用 instrument 的类修改功能，我们需要实现它提供的 ClassFileTransformer 接口，定义一个类文件转换器。接口中的 transform() 方法会在类文件被加载时调用，而在 transform 方法里，我们可以利用上文中的 ASM 或 Javassist 对传入的字节码进行改写或替换，生成新的字节码数组后返回。

我们定义一个实现了 ClassFileTransformer 接口的类 TestTransformer，依然在其中利用 Javassist 对 Base 类中的 process() 方法进行增强，在前后分别打印

“start”和“end”，代码如下：

```
import java.lang.instrument.ClassFileTransformer;

public class TestTransformer implements ClassFileTransformer {
    @Override
    public byte[] transform(ClassLoader loader, String className,
Class<?> classBeingRedefined,
ProtectionDomain protectionDomain, byte[] classfileBuffer) {
        System.out.println("Transforming " + className);
        try {
            ClassPool cp = ClassPool.getDefault();
            CtClass cc = cp.get("meituan.bytecode.jvmti.Base");
            CtMethod m = cc.getDeclaredMethod("process");
            m.insertBefore("{ System.out.println(\"start\"); }");
            m.insertAfter("{ System.out.println(\"end\"); }");
            return cc.toBytecode();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

现在有了 Transformer，那么它要如何注入到正在运行的 JVM 呢？还需要定义一个 Agent，借助 Agent 的能力将 Instrument 注入到 JVM 中。我们将在下一小节介绍 Agent，现在要介绍的是 Agent 中用到的另一个类 Instrumentation。在 JDK 1.6 之后，Instrumentation 可以做启动后的 Instrument、本地代码（Native Code）的 Instrument，以及动态改变 Classpath 等等。我们可以向 Instrumentation 中添加上文中定义的 Transformer，并指定要被重加载的类，代码如下所示。这样，当 Agent 被 Attach 到一个 JVM 中时，就会执行类字节码替换并重载入 JVM 的操作。

```
import java.lang.instrument.Instrumentation;

public class TestAgent {
    public static void agentmain(String args, Instrumentation inst) {
        // 指定我们自己定义的 Transformer，在其中利用 Javassist 做字节码替换
        inst.addTransformer(new TestTransformer(), true);
        try {
            // 重定义类并载入新的字节码
            inst.retransformClasses(Base.class);
            System.out.println("Agent Load Done.");
```

```
        } catch (Exception e) {
            System.out.println("agent load failed!");
        }
    }
}
```

3.3 JVMTI & Agent & Attach API

上一小节中，我们给出了 Agent 类的代码，追根溯源需要先介绍 JPDA（Java Platform Debugger Architecture）。如果 JVM 启动时开启了 JPDA，那么类是允许被重新加载的。在这种情况下，已被加载的旧版本类信息可以被卸载，然后重新加载新版本的类。正如 JDPA 名称中的 Debugger，JDPA 其实是一套用于调试 Java 程序的标准，任何 JDK 都必须实现该标准。

JPDA 定义了一整套完整的体系，它将调试体系分为三部分，并规定了三者之间的通信接口。三部分由低到高分别是 Java 虚拟机工具接口（JVMTI），Java 调试协议（JDWP）以及 Java 调试接口（JDI），三者之间的关系如下图所示：

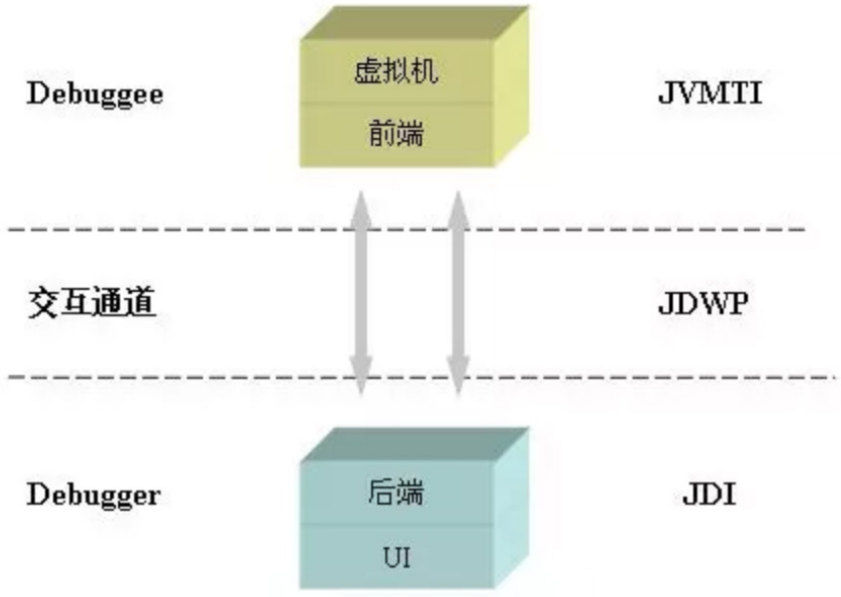


图 21 JPDA

现在回到正题，我们可以借助 JVMTI 的一部分能力，帮助动态重载类信息。JVM TI（JVM TOOL INTERFACE，JVM 工具接口）是 JVM 提供的一套对 JVM 进行操作的工具接口。通过 JVMTI，可以实现对 JVM 的多种操作，它通过接口注册各种事件勾子，在 JVM 事件触发时，同时触发预定义的勾子，以实现对各个 JVM 事件的响应，事件包括类文件加载、异常产生与捕获、线程启动和结束、进入和退出临界区、成员变量修改、GC 开始和结束、方法调用进入和退出、临界区竞争与等待、VM 启动与退出等等。

而 Agent 就是 JVMTI 的一种实现，Agent 有两种启动方式，一是随 Java 进程启动而启动，经常见到的 java -agentlib 就是这种方式；二是运行时载入，通过 attach API，将模块（jar 包）动态地 Attach 到指定进程 id 的 Java 进程内。

Attach API 的作用是提供 JVM 进程间通信的能力，比如说我们为了让另外一个 JVM 进程把线上服务的线程 Dump 出来，会运行 jstack 或 jmap 的进程，并传递 pid 的参数，告诉它要对哪个进程进行线程 Dump，这就是 Attach API 做的事情。在下面，我们将通过 Attach API 的 loadAgent() 方法，将打包好的 Agent jar 包动态 Attach 到目标 JVM 上。具体实现起来的步骤如下：

- 定义 Agent，并在其中实现 AgentMain 方法，如上一小节中定义的代码块 7 中的 TestAgent 类；
- 然后将 TestAgent 类打成一个包含 MANIFEST.MF 的 jar 包，其中 MANIFEST.MF 文件中将 Agent-Class 属性指定为 TestAgent 的全限定名，如下图所示；

```
Manifest-Version: 1.0
Agent-Class: meituan.bytecode.jvmti.TestAgent
Created-By: zhaozeen
Can-Redefine-Classes: true
Can-Retransform-Classes: true
Boot-Class-Path: javassist-3.20.0-GA.jar
```

图 22　Manifest.mf

- 最后利用 Attach API，将我们打包好的 jar 包 Attach 到指定的 JVM pid 上，代码如下：

```
import com.sun.tools.attach.VirtualMachine;

public class Attacher {
    public static void main(String[] args) throws
AttachNotSupportedException, IOException,
AgentLoadException, AgentInitializationException {
        // 传入目标 JVM pid
        VirtualMachine vm = VirtualMachine.attach("39333");
        vm.loadAgent("/Users/zen/operation_server_jar/operation-
server.jar");
    }
}
```

- 由于在 MANIFEST.MF 中指定了 Agent-Class，所以在 Attach 后，目标 JVM 在运行时会走到 TestAgent 类中定义的 agentmain() 方法，而在这个方法中，我们利用 Instrumentation，将指定类的字节码通过定义的类转化器 TestTransformer 做了 Base 类的字节码替换（通过 javassist），并完成了类的重新加载。由此，我们达成了“在 JVM 运行时，改变类的字节码并重新载入类信息”的目的。

以下为运行时重新载入类的效果：先运行 Base 中的 main() 方法，启动一个 JVM，可以在控制台看到每隔五秒输出一次”process”。接着执行 Attacher 中的 main() 方法，并将上一个 JVM 的 pid 传入。此时回到上一个 main() 方法的控制台，可以看到现在每隔五秒输出”process”前后会分别输出”start”和”end”，也就是说完成了运行时的字节码增强，并重新载入了这个类。

```
process
process
process
process
objc[32480]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_151.jdk/
WARNING: javassist-3.20.0-GA.jar not added to bootstrap class loader search: Illegal argument or not JAR file
Transforming meituan/bytecode/jvmti/Base
Agent Load Done.
start
process
end
start
process
end
```

图 23　运行时重载入类的效果

3.4 使用场景

至此，字节码增强技术的可使用范围就不再局限于 JVM 加载类前了。通过上述几个类库，我们可以在运行时对 JVM 中的类进行修改并重载了。通过这种手段，可以做的事情就变得很多了：

- 热部署：不部署服务而对线上服务做修改，可以做打点、增加日志等操作。
- Mock：测试时候对某些服务做 Mock。
- 性能诊断工具：比如 bTrace 就是利用 Instrument，实现无侵入地跟踪一个正在运行的 JVM，监控到类和方法级别的状态信息。

4. 总结

字节码增强技术相当于是一把打开运行时 JVM 的钥匙，利用它可以动态地对运行中的程序做修改，也可以跟踪 JVM 运行中程序的状态。此外，我们平时使用的动态代理、AOP 也与字节码增强密切相关，它们实质上还是利用各种手段生成符合规范的字节码文件。综上所述，掌握字节码增强后可以高效地定位并快速修复一些棘手的问题（如线上性能问题、方法出现不可控的出入参需要紧急加日志等问题），也可以在开发中减少冗余代码，大大提高开发效率。

5. 参考文献

- 《ASM4-Guide》
- Oracle:The class File Format
- Oracle:The Java Virtual Machine Instruction Set
- javassist tutorial
- JVM Tool Interface - Version 1.2

作者简介

泽恩，美团点评研发工程师。

招聘信息

美团到店住宿业务研发团队负责美团酒店核心业务系统建设，致力于通过技术践行“帮大家住得更好”的使命。美团酒店屡次刷新行业记录，最近 12 个月酒店预订间夜量达到 3 个亿，单日

入住间夜量峰值突破 280 万。团队的愿景是：建设打造旅游住宿行业一流的技术架构，从质量、安全、效率、性能多角度保障系统高速发展。

美团到店事业群住宿业务研发团队现诚聘后台开发工程师 / 技术专家，欢迎有兴趣的同学投简历至：tech@meituan.com（注明：美团到店事业群住宿业务研发团队）

JVM CPU Profiler 技术原理及源码深度解析

业祥　继东

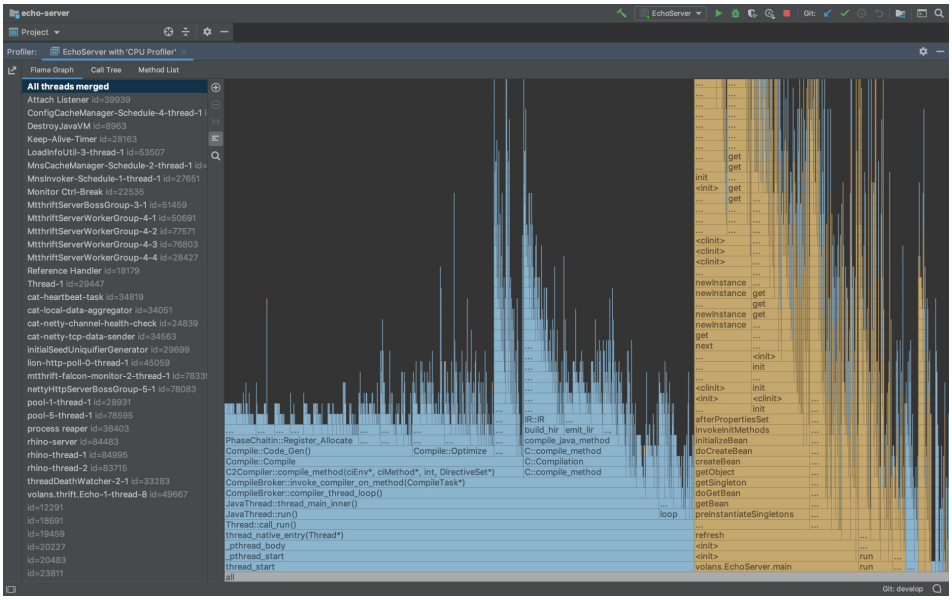
研发人员在遇到线上报警或需要优化系统性能时，常常需要分析程序运行行为和性能瓶颈。Profiling 技术是一种在应用运行时收集程序相关信息的动态分析手段，常用的 JVM Profiler 可以从多个方面对程序进行动态分析，如 CPU、Memory、Thread、Classes、GC 等，其中 CPU Profiling 的应用最为广泛。CPU Profiling 经常被用于分析代码的执行热点，如“哪个方法占用 CPU 的执行时间最长”、“每个方法占用 CPU 的比例是多少”等等，通过 CPU Profiling 得到上述相关信息后，研发人员就可以轻松针对热点瓶颈进行分析和性能优化，进而突破性能瓶颈，大幅提升系统的吞吐量。

本文介绍了 JVM 平台上 CPU Profiler 的实现原理，希望能帮助读者在使用类似工具的同时也能清楚其内部的技术实现。

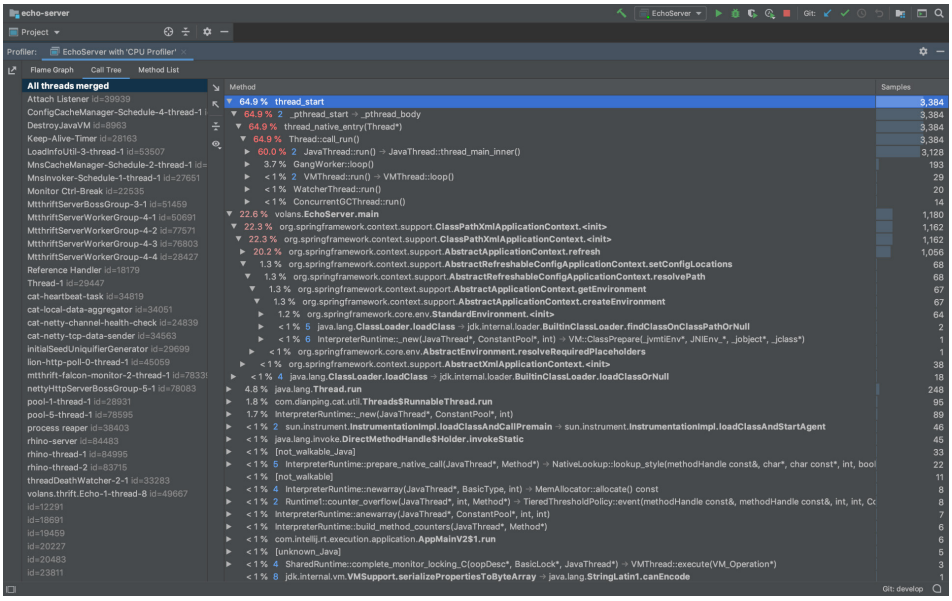
CPU Profiler 简介

社区实现的 JVM Profiler 很多，比如已经商用且功能强大的 JProfiler，也有免费开源的产品，如 JVM-Profiler，功能各有所长。我们日常使用的 Intellij IDEA 最新版内部也集成了一个简单好用的 Profiler，详细的介绍参见官方 Blog。

在用 IDEA 打开需要诊断的 Java 项目后，在“Preferences -> Build, Execution, Deployment -> Java Profiler”界面添加一个“CPU Profiler”，然后回到项目，单击右上角的“Run with Profiler”启动项目并开始 CPU Profiling 过程。一定时间后（推荐 5min），在 Profiler 界面点击“Stop Profiling and Show Results”，即可看到 Profiling 的结果，包含火焰图和调用树，如下图所示：



Intellij IDEA – 性能火焰图



Intellij IDEA – 调用堆栈树

火焰图是根据调用栈的样本集生成的可视化性能分析图，《如何读懂火焰图?》一

文对火焰图进行了不错的讲解，大家可以参考一下。简而言之，看火焰图时我们需要关注“平顶”，因为那里就是我们程序的 CPU 热点。调用树是另一种可视化分析的手段，与火焰图一样，也是根据同一份样本集而生成，按需选择即可。

这里要说明一下，因为我们没有在项目中引入任何依赖，仅仅是“Run with Profiler”，Profiler 就能获取我们程序运行时的信息。这个功能其实是通过 JVM Agent 实现的，为了更好地帮助大家系统性的了解它，我们在这里先对 JVM Agent 做个简单的介绍。

JVM Agent 简介

JVM Agent 是一个按一定规则编写的特殊程序库，可以在启动阶段通过命令行参数传递给 JVM，作为一个伴生库与目标 JVM 运行在同一个进程中。在 Agent 中可以通过固定的接口获取 JVM 进程内的相关信息。Agent 既可以是用 C/C++/Rust 编写的 JVMTI Agent，也可以是用 Java 编写的 Java Agent。

执行 Java 命令，我们可以看到 Agent 相关的命令行参数：

```
Plain Text
    -agentlib:< 库名 >[=< 选项 >]
                  加载本机代理库 < 库名 >, 例如 -agentlib:jdwp
                  另请参阅 -agentlib:jdwp=help
    -agentpath:< 路径名 >[=< 选项 >]
                  按完整路径名加载本机代理库
    -javaagent:<jar 路径 >[=< 选项 >]
                  加载 Java 编程语言代理，请参阅 java.lang.instrument
```

JVMTI Agent

JVMTI（JVM Tool Interface）是 JVM 提供的一套标准的 C/C++ 编程接口，是实现 Debugger、Profiler、Monitor、Thread Analyser 等工具的统一基础，在主流 Java 虚拟机中都有实现。

当我们要基于 JVMTI 实现一个 Agent 时，需要实现如下入口函数：

```
// $JAVA_HOME/include/jvmti.h

JNIEXPORT jint JNICALL Agent_OnLoad(JavaVM *vm, char *options, void *reserved);
```

使用 C/C++ 实现该函数，并将代码编译为动态连接库（Linux 上是 .so），通过 -agentpath 参数将库的完整路径传递给 Java 进程，JVM 就会在启动阶段的合适时机执行该函数。在函数内部，我们可以通过 JavaVM 指针参数拿到 JNI 和 JVMTI 的函数指针表，这样我们就拥有了与 JVM 进行各种复杂交互的能力。

更多 JVMTI 相关的细节可以参考官方文档。

Java Agent

在很多场景下，我们没有必要必须使用 C/C++ 来开发 JVMTI Agent，因为成本高且不易维护。JVM 自身基于 JVMTI 封装了一套 Java 的 Instrument API 接口，允许使用 Java 语言开发 Java Agent（只是一个 jar 包），大大降低了 Agent 的开发成本。社区开源的产品如 Greys、Arthas、JVM-Sandbox、JVM-Profiler 等都是纯 Java 编写的，也是以 Java Agent 形式来运行。

在 Java Agent 中，我们需要在 jar 包的 MANIFEST.MF 中将 Premain-Class 指定为一个入口类，并在该入口类中实现如下方法：

```
public static void premain(String args, Instrumentation ins) {
    // implement
}
```

这样打包出来的 jar 就是一个 Java Agent，可以通过 -javaagent 参数将 jar 传递给 Java 进程伴随启动，JVM 同样会在启动阶段的合适时机执行该方法。

在该方法内部，参数 Instrumentation 接口提供了 Retransform Classes 的能力，我们利用该接口就可以对宿主进程的 Class 进行修改，实现方法耗时统计、故障注入、Trace 等功能。Instrumentation 接口提供的能力较为单一，仅与 Class 字节码操作相关，但由于我们现在已经处于宿主进程环境内，就可以利用 JMX 直接获取宿主进程的内存、线程、锁等信息。无论是 Instrument API 还是 JMX，它们内部仍是统一基于 JVMTI 来实现。

更多 Instrument API 相关的细节可以参考官方文档。

CPU Profiler 原理解析

在了解完 Profiler 如何以 Agent 的形式执行后，我们可以开始尝试构造一个简单的 CPU Profiler。但在此之前，还有必要了解下 CPU Profiling 技术的两种实现方式及其区别。

Sampling vs Instrumentation

使用过 JProfiler 的同学应该都知道，JProfiler 的 CPU Profiling 功能提供了两种方式选项：Sampling 和 Instrumentation，它们也是实现 CPU Profiler 的两种手段。

Sampling 方式顾名思义，基于对 StackTrace 的“采样”进行实现，核心原理如下：

1. 引入 Profiler 依赖，或直接利用 Agent 技术注入目标 JVM 进程并启动 Profiler。
2. 启动一个采样定时器，以固定的采样频率每隔一段时间（毫秒级）对所有线程的调用栈进行 Dump。
3. 汇总并统计每次调用栈的 Dump 结果，在一定时间内采到足够的样本后，导出统计结果，内容是每个方法被采样到的次数及方法的调用关系。

Instrumentation 则是利用 Instrument API，对所有必要的 Class 进行字节码增强，在进入每个方法前进行埋点，方法执行结束后统计本次方法执行耗时，最终进行汇总。二者都能得到想要的结果，那么它们有什么区别呢？或者说，孰优孰劣？

Instrumentation 方式对几乎所有方法添加了额外的 AOP 逻辑，这会导致对线上服务造成巨额的性能影响，但其优势是：绝对精准的方法调用次数、调用时间统计。

Sampling 方式基于无侵入的额外线程对所有线程的调用栈快照进行固定频率抽样，相对前者来说它的性能开销很低。但由于它基于“采样”的模式，以及 JVM 固有的只能在安全点（Safe Point）进行采样的“缺陷”，会导致统计结果存在一定的

偏差。譬如说：某些方法执行时间极短，但执行频率很高，真实占用了大量的 CPU Time，但 Sampling Profiler 的采样周期不能无限调小，这会导致性能开销骤增，所以会导致大量的样本调用栈中并不存在刚才提到的”高频小方法“，进而导致最终结果无法反映真实的 CPU 热点。更多 Sampling 相关的问题可以参考《Why (Most) Sampling Java Profilers Are Fucking Terrible》。

具体到“孰优孰劣”的问题层面，这两种实现技术并没有非常明显的高下之判，只有在分场景讨论下才有意义。Sampling 由于低开销的特性，更适合用在 CPU 密集型的应用中，以及不可接受大量性能开销的线上服务中。而 Instrumentation 则更适合用在 I/O 密集的应用中、对性能开销不敏感以及确实需要精确统计的场景中。社区的 Profiler 更多的是基于 Sampling 来实现，本文也是基于 Sampling 来进行讲解。

基于 Java Agent + JMX 实现

一个最简单的 Sampling CPU Profiler 可以用 Java Agent + JMX 方式来实现。以 Java Agent 为入口，进入目标 JVM 进程后开启一个 ScheduledExecutorService，定时利用 JMX 的 threadMXBean.dumpAllThreads() 来导出所有线程的 StackTrace，最终汇总并导出即可。

Uber 的 JVM-Profiler 实现原理也是如此，关键部分代码如下：

```
// com/uber/profiling/profilers/StacktraceCollectorProfiler.java

/*
 * StacktraceCollectorProfiler 等同于文中所述 CpuProfiler，仅命名偏好不同而已
 * jvm-profiler 的 CpuProfiler 指代的是 CpuLoad 指标的 Profiler
 */

// 实现了 Profiler 接口，外部由统一的 ScheduledExecutorService 对所有 Profiler 定时执行
@Override
public void profile() {
    ThreadInfo[] threadInfos = threadMXBean.dumpAllThreads(false, false);
    // ...
    for (ThreadInfo threadInfo : threadInfos) {
        String threadName = threadInfo.getThreadName();
        // ...
```

```
            StackTraceElement[] stackTraceElements = threadInfo.
    getStackTrace();
            // ...
            for (int i = stackTraceElements.length - 1; i >= 0; i--) {
                StackTraceElement stackTraceElement = stackTraceElements[i];
                // ...
            }
            // ...
        }
    }
```

Uber 提供的定时器默认 Interval 是 100ms，对于 CPU Profiler 来说，这略显粗糙。但由于 dumpAllThreads() 的执行开销不容小觑，Interval 不宜设置的过小，所以该方法的 CPU Profiling 结果会存在不小的误差。

JVM-Profiler 的优点在于支持多种指标的 Profiling（StackTrace、CPUBusy、Memory、I/O、Method），且支持将 Profiling 结果通过 Kafka 上报回中心 Server 进行分析，也即支持集群诊断。

基于 JVMTI + GetStackTrace 实现

使用 Java 实现 Profiler 相对较简单，但也存在一些问题，譬如说 Java Agent 代码与业务代码共享 AppClassLoader，被 JVM 直接加载的 agent.jar 如果引入了第三方依赖，可能会对业务 Class 造成污染。截止发稿时，JVM-Profiler 都存在这个问题，它引入了 Kafka-Client、http-Client、Jackson 等组件，如果与业务代码中的组件版本发生冲突，可能会引发未知错误。Greys/Arthas/JVM-Sandbox 的解决方式是分离入口与核心代码，使用定制的 ClassLoader 加载核心代码，避免影响业务代码。

在更底层的 C/C++ 层面，我们可以直接对接 JVMTI 接口，使用原生 C API 对 JVM 进行操作，功能更丰富更强大，但开发效率偏低。基于上节同样的原理开发 CPU Profiler，使用 JVMTI 需要进行如下这些步骤：

1. 编写 Agent_OnLoad()，在入口通过 JNI 的 JavaVM* 指针的 GetEnv() 函数拿到 JVMTI 的 jvmtiEnv 指针：

```
// agent.c

JNIEXPORT jint JNICALL Agent_OnLoad(JavaVM *vm, char *options, void
*reserved) {
    jvmtiEnv *jvmti;
    (*vm)->GetEnv((void **)&jvmti, JVMTI_VERSION_1_0);
    // ...
    return JNI_OK;
}
```

2. 开启一个线程定时循环，定时使用 jvmtiEnv 指针配合调用如下几个 JVMTI 函数：

```
// 获取所有线程的 jthread
jvmtiError GetAllThreads(jvmtiEnv *env, jint *threads_count_ptr,
jthread **threads_ptr);

// 根据 jthread 获取该线程信息（name、daemon、priority...）
jvmtiError GetThreadInfo(jvmtiEnv *env, jthread thread,
jvmtiThreadInfo* info_ptr);

// 根据 jthread 获取该线程调用栈
jvmtiError GetStackTrace(jvmtiEnv *env,
                         jthread thread,
                         jint start_depth,
                         jint max_frame_count,
                         jvmtiFrameInfo *frame_buffer,
                         jint *count_ptr);
```

主逻辑大致是：首先调用 GetAllThreads() 获取所有线程的“句柄”jthread，然后遍历根据 jthread 调用 GetThreadInfo() 获取线程信息，按线程名过滤掉不需要的线程后，继续遍历根据 jthread 调用 GetStackTrace() 获取线程的调用栈。

3. 在 Buffer 中保存每一次的采样结果，最终生成必要的统计数据即可。

按如上步骤即可实现基于 JVMTI 的 CPU Profiler。但需要说明的是，即便是基于原生 JVMTI 接口使用 GetStackTrace() 的方式获取调用栈，也存在与 JMX 相同的问题——只能在安全点（Safe Point）进行采样。

SafePoint Bias 问题

基于 Sampling 的 CPU Profiler 通过采集程序在不同时间点的调用栈样本来近似地推算出热点方法，因此，从理论上来讲 Sampling CPU Profiler 必须遵循以下两个原则：

1. 样本必须足够多。
2. 程序中所有正在运行的代码点都必须以相同的概率被 Profiler 采样。

如果只能在安全点采样，就违背了第二条原则。因为我们只能采集到位于安全点时刻的调用栈快照，意味着某些代码可能永远没有机会被采样，即使它真实耗费了大量的 CPU 执行时间，这种现象被称为“SafePoint Bias”。

上文我们提到，基于 JMX 与基于 JVMTI 的 Profiler 实现都存在 SafePoint Bias，但一个值得了解的细节是：单独来说，JVMTI 的 GetStackTrace() 函数并不需要在 Caller 的安全点执行，但当调用 GetStackTrace() 获取其他线程的调用栈时，必须等待，直到目标线程进入安全点；而且，GetStackTrace() 仅能通过单独的线程同步定时调用，不能在 UNIX 信号处理器的 Handler 中被异步调用。综合来说，GetStackTrace() 存在与 JMX 一样的 SafePoint Bias。更多安全点相关的知识可以参考《Safepoints: Meaning, Side Effects and Overheads》。

那么，如何避免 SafePoint Bias？社区提供了一种 Hack 思路——AsyncGetCallTrace。

基于 JVMTI + AsyncGetCallTrace 实现

如上节所述，假如我们拥有一个函数可以获取当前线程的调用栈且不受安全点干扰，另外它还支持在 UNIX 信号处理器中被异步调用，那么我们只需注册一个 UNIX 信号处理器，在 Handler 中调用该函数获取当前线程的调用栈即可。由于 UNIX 信号会被发送给进程的随机一线程进行处理，因此最终信号会均匀分布在所有线程上，也就均匀获取了所有线程的调用栈样本。

OracleJDK/OpenJDK 内部提供了这么一个函数——AsyncGetCallTrace，它

的原型如下：

```
// 栈帧
typedef struct {
 jint lineno;
 jmethodID method_id;
} AGCT_CallFrame;

// 调用栈
typedef struct {
    JNIEnv *env;
    jint num_frames;
    AGCT_CallFrame *frames;
} AGCT_CallTrace;

// 根据 ucontext 将调用栈填充进 trace 指针
void AsyncGetCallTrace(AGCT_CallTrace *trace, jint depth, void
*ucontext);
```

通过原型可以看到，该函数的使用方式非常简洁，直接通过 ucontext 就能获取到完整的 Java 调用栈。

顾名思义，AsyncGetCallTrace 是“async”的，不受安全点影响，这样的话采样就可能发生在任何时间，包括 Native 代码执行期间、GC 期间等，在这时我们是无法获取 Java 调用栈的，AGCT_CallTrace 的 num_frames 字段正常情况下标识了获取到的调用栈深度，但在如前所述的异常情况下它就表示为负数，最常见的 −2 代表此刻正在 GC。

由于 AsyncGetCallTrace 非标准 JVMTI 函数，因此我们无法在 jvmti.h 中找到该函数声明，且由于其目标文件也早已链接进 JVM 二进制文件中，所以无法通过简单的声明来获取该函数的地址，这需要通过一些 Trick 方式来解决。简单说，Agent 最终是作为动态链接库加载到目标 JVM 进程的地址空间中，因此可以在 Agent_OnLoad 内通过 glibc 提供的 dlsym() 函数拿到当前地址空间（即目标 JVM 进程地址空间）名为“AsyncGetCallTrace”的符号地址。这样就拿到了该函数的指针，按照上述原型进行类型转换后，就可以正常调用了。

通过 AsyncGetCallTrace 实现 CPU Profiler 的大致流程：

1. 编写 Agent_OnLoad()，在入口拿到 jvmtiEnv 和 AsyncGetCallTrace 指针，获取 AsyncGetCallTrace 方式如下：

```
typedef void (*AsyncGetCallTrace)(AGCT_CallTrace *traces, jint depth,
void *ucontext);
// ...
AsyncGetCallTrace agct_ptr = (AsyncGetCallTrace)dlsym(RTLD_DEFAULT,
"AsyncGetCallTrace");
if (agct_ptr == NULL) {
    void *libjvm = dlopen("libjvm.so", RTLD_NOW);
    if (!libjvm) {
        // 处理 dlerror()...
    }
    agct_ptr = (AsyncGetCallTrace)dlsym(libjvm, "AsyncGetCallTrace");
}
```

2. 在 OnLoad 阶段，我们还需要做一件事，即注册 OnClassLoad 和 OnClassPrepare 这两个 Hook，原因是 jmethodID 是延迟分配的，使用 AGCT 获取 Traces 依赖预先分配好的数据。我们在 OnClassPrepare 的 CallBack 中尝试获取该 Class 的所有 Methods，这样就使 JVMTI 提前分配了所有方法的 jmethodID，如下所示：

```
void JNICALL OnClassLoad(jvmtiEnv *jvmti, JNIEnv* jni, jthread thread,
jclass klass) {}

void JNICALL OnClassPrepare(jvmtiEnv *jvmti, JNIEnv *jni, jthread
thread, jclass klass) {
    jint method_count;
    jmethodID *methods;
    jvmti->GetClassMethods(klass, &method_count, &methods);
    delete [] methods;
}

// ...

jvmtiEventCallbacks callbacks = {0};
callbacks.ClassLoad = OnClassLoad;
callbacks.ClassPrepare = OnClassPrepare;
jvmti->SetEventCallbacks(&callbacks, sizeof(callbacks));
jvmti->SetEventNotificationMode(JVMTI_ENABLE, JVMTI_EVENT_CLASS_LOAD,
NULL);
jvmti->SetEventNotificationMode(JVMTI_ENABLE, JVMTI_EVENT_CLASS_PREPARE, NULL);
```

3. 利用 SIGPROF 信号来进行定时采样：

```
// 这里信号 handler 传进来的 ucontext 即 AsyncGetCallTrace 需要的 ucontext
void signal_handler(int signo, siginfo_t *siginfo, void *ucontext) {
    // 使用 AsyncCallTrace 进行采样，注意处理 num_frames 为负的异常情况
}

// ...

// 注册 SIGPROF 信号的 handler
struct sigaction sa;
sigemptyset(&sa.sa_mask);
sa.sa_sigaction = signal_handler;
sa.sa_flags = SA_RESTART | SA_SIGINFO;
sigaction(SIGPROF, &sa, NULL);

// 定时产生 SIGPROF 信号
// interval 是 nanoseconds 表示的采样间隔，AsyncGetCallTrace 相对于同步采样来说可以适当高频一些
long sec = interval / 1000000000;
long usec = (interval % 1000000000) / 1000;
struct itimerval tv = {{sec, usec}, {sec, usec}};
setitimer(ITIMER_PROF, &tv, NULL);
```

4. 在 Buffer 中保存每一次的采样结果，最终生成必要的统计数据即可。

按如上步骤即可实现基于 AsyncGetCallTrace 的 CPU Profiler，这是社区中目前性能开销最低、相对效率最高的 CPU Profiler 实现方式，在 Linux 环境下结合 perf_events 还能做到同时采样 Java 栈与 Native 栈，也就能同时分析 Native 代码中存在的性能热点。该方式的典型开源实现有 Async-Profiler 和 Honest-Profiler，Async-Profiler 实现质量较高，感兴趣的话建议大家阅读参考文章。有趣的是，IntelliJ IDEA 内置的 Java Profiler，其实就是 Async-Profiler 的包装。更多关于 AsyncGetCallTrace 的内容，大家可以参考《The Pros and Cons of AsyncGetCallTrace Profilers》。

生成性能火焰图

现在我们拥有了采样调用栈的能力，但是调用栈样本集是以二维数组的数据结构形式存在于内存中的，如何将其转换为可视化的火焰图呢?

火焰图通常是一个 svg 文件，部分优秀项目可以根据文本文件自动生成火焰图文

件，仅对文本文件的格式有一定要求。FlameGraph 项目的核心只是一个 Perl 脚本，可以根据我们提供的调用栈文本生成相应的火焰图 svg 文件。调用栈的文本格式相当简单，如下所示：

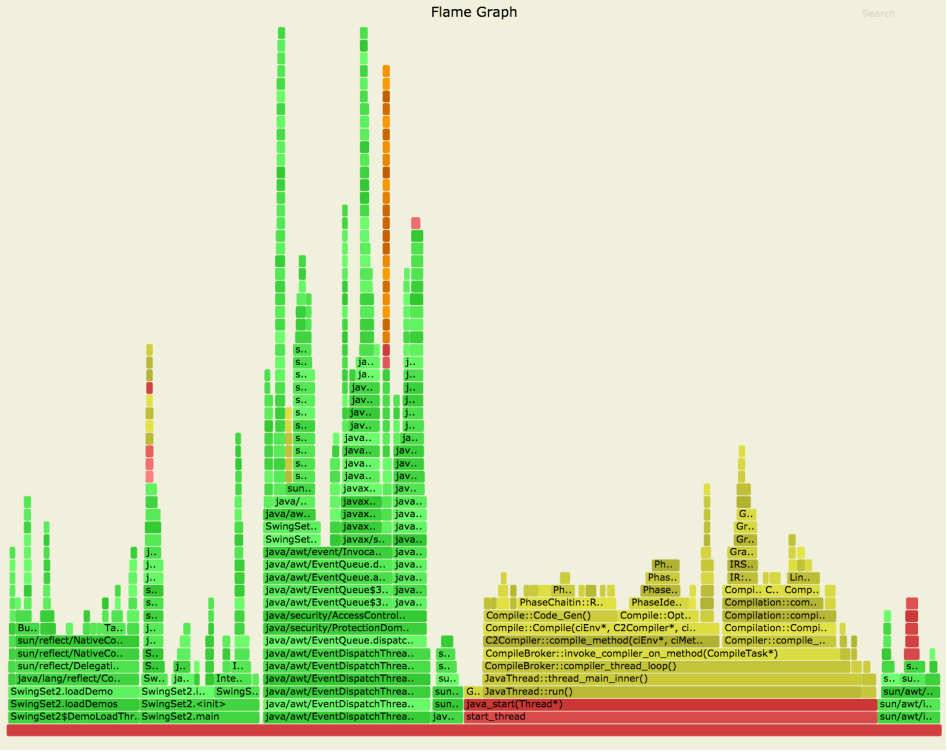
```
base_func;func1;func2;func3 10
base_func;funca;funcb 15
```

将我们采样到的调用栈样本集进行整合后，需输出如上所示的文本格式。每一行代表一“类“调用栈，空格左边是调用栈的方法名排列，以分号分割，左栈底右栈顶，空格右边是该样本出现的次数。

将样本文件交给 flamegraph.pl 脚本执行，就能输出相应的火焰图了：

```
$ flamegraph.pl stacktraces.txt > stacktraces.svg
```

效果如下图所示：



通过 flamegraph.pl 生成的火焰图

HotSpot 的 Dynamic Attach 机制解析

到目前为止，我们已经了解了 CPU Profiler 完整的工作原理，然而使用过 JProfiler/Arthas 的同学可能会有疑问，很多情况下可以直接对线上运行中的服务进行 Profling，并不需要在 Java 进程的启动参数添加 Agent 参数，这是通过什么手段做到的？答案是 Dynamic Attach。

JDK 在 1.6 以后提供了 Attach API，允许向运行中的 JVM 进程添加 Agent，这项手段被广泛使用在各种 Profiler 和字节码增强工具中，其官方简介如下：

> This is a Sun extension that allows a tool to ‘attach’ to another process running Java code and launch a JVM TI agent or a java.lang.instrument agent in that process.

总的来说，Dynamic Attach 是 HotSpot 提供的一种特殊能力，它允许一个进程向另一个运行中的 JVM 进程发送一些命令并执行，命令并不限于加载 Agent，还包括 Dump 内存、Dump 线程等等。

通过 sun.tools 进行 Attach

Attach 虽然是 HotSpot 提供的能力，但 JDK 在 Java 层面也对其做了封装。

前文已经提到，对于 Java Agent 来说，PreMain 方法在 Agent 作为启动参数运行的时候执行，其实我们还可以额外实现一个 AgentMain 方法，并在 MANIFEST.MF 中将 Agent-Class 指定为该 Class：

```
public static void agentmain(String args, Instrumentation ins) {
    // implement
}
```

这样打包出来的 jar，既可以作为 -javaagent 参数启动，也可以被 Attach 到运行中的目标 JVM 进程。JDK 已经封装了简单的 API 让我们直接 Attach 一个 Java Agent，下面以 Arthas 中的代码进行演示：

```
// com/taobao/arthas/core/Arthas.java
```

```
import com.sun.tools.attach.VirtualMachine;
import com.sun.tools.attach.VirtualMachineDescriptor;

// ...

private void attachAgent(Configure configure) throws Exception {
    VirtualMachineDescriptor virtualMachineDescriptor = null;

    // 拿到所有 JVM 进程，找出目标进程
    for (VirtualMachineDescriptor descriptor : VirtualMachine.list()) {
        String pid = descriptor.id();
        if (pid.equals(Integer.toString(configure.getJavaPid()))) {
            virtualMachineDescriptor = descriptor;
        }
    }
    VirtualMachine virtualMachine = null;
    try {
        // 针对某个 JVM 进程调用 VirtualMachine.attach() 方法，拿到
VirtualMachine 实例
        if (null == virtualMachineDescriptor) {
            virtualMachine = VirtualMachine.attach("" + configure.
getJavaPid());
        } else {
            virtualMachine = VirtualMachine.
attach(virtualMachineDescriptor);
        }

        // ...

        // 调用 VirtualMachine#loadAgent()，将 arthasAgentPath 指定的 jar
attach 到目标 JVM
进程中
        // 第二个参数为 attach 参数，即 agentmain 的首个 String 参数 args
        virtualMachine.loadAgent(arthasAgentPath, configure.
getArthasCore() + ";" + configure.
toString());
    } finally {
        if (null != virtualMachine) {
            // 调用 VirtualMachine#detach() 释放
            virtualMachine.detach();
        }
    }
}
```

直接对 HotSpot 进行 Attach

sun.tools 封装的 API 足够简单易用，但只能使用 Java 编写，也只能用在 Java Agent 上，因此有些时候我们必须手工对 JVM 进程直接进行 Attach。对于 JVMTI，除了 Agent_OnLoad() 之外，我们还需实现一个 Agent_OnAttach() 函数，当将 JVMTI Agent Attach 到目标进程时，从该函数开始执行：

```
// $JAVA_HOME/include/jvmti.h

JNIEXPORT jint JNICALL Agent_OnAttach(JavaVM *vm, char *options, void
*reserved);
```

下面我们以 Async-Profiler 中的 jattach 源码为线索，探究一下如何利用 Attach 机制给运行中的 JVM 进程发送命令。jattach 是 Async-Profiler 提供的一个 Driver，使用方式比较直观：

```
Usage:
    jattach <pid> <cmd> [args ...]
Args:
    <pid>   目标 JVM 进程的进程 ID
    <cmd>   要执行的命令
    <args>  命令参数
```

使用方式如：

```
$ jattach 1234 load /absolute/path/to/agent/libagent.so true
```

执行上述命令，libagent.so 就被加载到 ID 为 1234 的 JVM 进程中并开始执行 Agent_OnAttach 函数了。有一点需要注意，执行 Attach 的进程 euid 及 egid，与被 Attach 的目标 JVM 进程必须相同。接下来开始分析 jattach 源码。

如下所示的 Main 函数描述了一次 Attach 的整体流程：

```
// async-profiler/src/jattach/jattach.c

int main(int argc, char** argv) {
    // 解析命令行参数
    // 检查 euid 与 egid
```

```
    // ...

    if (!check_socket(nspid) && !start_attach_mechanism(pid, nspid)) {
        perror("Could not start attach mechanism");
        return 1;
    }

    int fd = connect_socket(nspid);
    if (fd == -1) {
        perror("Could not connect to socket");
        return 1;
    }

    printf("Connected to remote JVM\n");
    if (!write_command(fd, argc - 2, argv + 2)) {
        perror("Error writing to socket");
        close(fd);
        return 1;
    }
    printf("Response code = ");
    fflush(stdout);

    int result = read_response(fd);
    close(fd);
    return result;
}
```

忽略掉命令行参数解析与检查 euid 和 egid 的过程。jattach 首先调用了 check_socket 函数进行了“socket 检查?”，check_socket 源码如下：

```
// async-profiler/src/jattach/jattach.c

// Check if remote JVM has already opened socket for Dynamic Attach
static int check_socket(int pid) {
    char path[MAX_PATH];
    snprintf(path, MAX_PATH, "%s/.java_pid%d", get_temp_directory(),
pid); // get_temp_
directory() 在 Linux 下固定返回 "/tmp"
    struct stat stats;
    return stat(path, &stats) == 0 && S_ISSOCK(stats.st_mode);
}
```

我们知道，UNIX 操作系统提供了一种基于文件的 Socket 接口，称为“UNIX Socket”（一种常用的进程间通信方式）。在该函数中使用 S_ISSOCK 宏来判断该文

件是否被绑定到了 UNIX Socket，如此看来，“/tmp/.java_pid”文件很有可能就是外部进程与 JVM 进程间通信的桥梁。

查阅官方文档，得到如下描述：

> The attach listener thread then communicates with the source JVM in an OS dependent manner: - On Solaris, the Doors IPC mechanism is used. The door is attached to a file in the file system so that clients can access it. - On Linux, a Unix domain socket is used. This socket is bound to a file in the filesystem so that clients can access it. - On Windows, the created thread is given the name of a pipe which is served by the client. The result of the operations are written to this pipe by the target JVM.

证明了我们的猜想是正确的。目前为止 check_socket 函数的作用很容易理解了：判断外部进程与目标 JVM 进程之间是否已经建立了 UNIX Socket 连接。

回到 Main 函数，在使用 check_socket 确定连接尚未建立后，紧接着调用 start_attach_mechanism 函数，函数名很直观地描述了它的作用，源码如下：

```
// async-profiler/src/jattach/jattach.c

// Force remote JVM to start Attach listener.
// HotSpot will start Attach listener in response to SIGQUIT if it
sees .attach_pid file
static int start_attach_mechanism(int pid, int nspid) {
    char path[MAX_PATH];
    snprintf(path, MAX_PATH, "/proc/%d/cwd/.attach_pid%d", nspid,
nspid);

    int fd = creat(path, 0660);
    if (fd == -1 || (close(fd) == 0 && !check_file_owner(path))) {
        // Failed to create attach trigger in current directory. Retry
in /tmp
        snprintf(path, MAX_PATH, "%s/.attach_pid%d", get_temp_
directory(), nspid);
        fd = creat(path, 0660);
        if (fd == -1) {
            return 0;
        }
        close(fd);
```

```
    }

    // We have to still use the host namespace pid here for the kill() call
    kill(pid, SIGQUIT);

    // Start with 20 ms sleep and increment delay each iteration
    struct timespec ts = {0, 20000000};
    int result;
    do {
        nanosleep(&ts, NULL);
        result = check_socket(nspid);
    } while (!result && (ts.tv_nsec += 20000000) < 300000000);

    unlink(path);
    return result;
}
```

start_attach_mechanism 函数首先创建了一个名为“/tmp/.attach_pid”的空文件，然后向目标 JVM 进程发送了一个 SIGQUIT 信号，这个信号似乎触发了 JVM 的某种机制？紧接着，start_attach_mechanism 函数开始陷入了一种等待，每 20ms 调用一次 check_socket 函数检查连接是否被建立，如果等了 300ms 还没有成功就放弃。函数的最后调用 Unlink 删掉 .attach_pid 文件并返回。

如此看来，HotSpot 似乎提供了一种特殊的机制，只要给它发送一个 SIGQUIT 信号，并预先准备好 .attach_pid 文件，HotSpot 会主动创建一个地址为“/tmp/.java_pid”的 UNIX Socket，接下来主动 Connect 这个地址即可建立连接执行命令。

查阅文档，得到如下描述：

> Dynamic attach has an attach listener thread in the target JVM. This is a thread that is started when the first attach request occurs. On Linux and Solaris, the client creates a file named .attach_pid(pid) and sends a SIGQUIT to the target JVM process. The existence of this file causes the SIGQUIT handler in HotSpot to start the attach listener thread. On Windows, the client uses the Win32 CreateRemoteThread function to create a new thread in the target process.

这样一来就很明确了，在 Linux 上我们只需创建一个“/tmp/.attach_pid”文

件，并向目标 JVM 进程发送一个 SIGQUIT 信号，HotSpot 就会开始监听“/tmp/.java_pid”地址上的 UNIX Socket，接收并执行相关 Attach 的命令。至于为什么一定要创建 .attach_pid 文件才可以触发 Attach Listener 的创建，经查阅资料，我们得到了两种说法：一是 JVM 不止接收从外部 Attach 进程发送的 SIGQUIT 信号，必须配合外部进程创建的外部文件才能确定这是一次 Attach 请求；二是为了安全。

继续看 jattach 的源码，果不其然，它调用了 connect_socket 函数对“/tmp/.java_pid”进行连接，connect_socket 源码如下：

```
// async-profiler/src/jattach/jattach.c

// Connect to UNIX domain socket created by JVM for Dynamic Attach
static int connect_socket(int pid) {
    int fd = socket(PF_UNIX, SOCK_STREAM, 0);
    if (fd == -1) {
        return -1;
    }

    struct sockaddr_un addr;
    addr.sun_family = AF_UNIX;
    snprintf(addr.sun_path, sizeof(addr.sun_path), "%s/.java_pid%d",
get_temp_directory(), pid);

    if (connect(fd, (struct sockaddr*)&addr, sizeof(addr)) == -1) {
        close(fd);
        return -1;
    }
    return fd;
}
```

一个很普通的 Socket 创建函数，返回 Socket 文件描述符。

回到 Main 函数，主流程紧接着调用 write_command 函数向该 Socket 写入了从命令行传进来的参数，并且调用 read_response 函数接收从目标 JVM 进程返回的数据。两个很常见的 Socket 读写函数，源码如下：

```
// async-profiler/src/jattach/jattach.c

// Send command with arguments to socket
static int write_command(int fd, int argc, char** argv) {
    // Protocol version
```

```
    if (write(fd, "1", 2) <= 0) {
        return 0;
    }

    int i;
    for (i = 0; i < 4; i++) {
        const char* arg = i < argc ? argv[i] : "";
        if (write(fd, arg, strlen(arg) + 1) <= 0) {
            return 0;
        }
    }
    return 1;
}

// Mirror response from remote JVM to stdout
static int read_response(int fd) {
    char buf[8192];
    ssize_t bytes = read(fd, buf, sizeof(buf) - 1);
    if (bytes <= 0) {
        perror("Error reading response");
        return 1;
    }

    // First line of response is the command result code
    buf[bytes] = 0;
    int result = atoi(buf);

    do {
        fwrite(buf, 1, bytes, stdout);
        bytes = read(fd, buf, sizeof(buf));
    } while (bytes > 0);
    return result;
}
```

浏览 write_command 函数就可知外部进程与目标 JVM 进程之间发送的数据格式相当简单，基本如下所示：

```
<PROTOCOL VERSION>\0<COMMAND>\0<ARG1>\0<ARG2>\0<ARG3>\0
```

以先前我们使用的 Load 命令为例，发送给 HotSpot 时格式如下：

```
1\0load\0/absolute/path/to/agent/libagent.so\0true\0\0
```

至此，我们已经了解了如何手工对 JVM 进程直接进行 Attach。

Attach 补充介绍

Load 命令仅仅是 HotSpot 所支持的诸多命令中的一种，用于动态加载基于 JVMTI 的 Agent，完整的命令表如下所示：

```
static AttachOperationFunctionInfo funcs[] = {
  { "agentProperties",  get_agent_properties },
  { "datadump",         data_dump },
  { "dumpheap",         dump_heap },
  { "load",             JvmtiExport::load_agent_library },
  { "properties",       get_system_properties },
  { "threaddump",       thread_dump },
  { "inspectheap",      heap_inspection },
  { "setflag",          set_flag },
  { "printflag",        print_flag },
  { "jcmd",             jcmd },
  { NULL,               NULL }
};
```

读者可以尝试下 threaddump 命令，然后对相同的进程进行 jstack，对比观察输出，其实是完全相同的，其它命令大家可以自行进行探索。

总结

总的来说，善用各类 Profiler 是提升性能优化效率的一把利器，了解 Profiler 本身的实现原理更能帮助我们避免对工具的各种误用。CPU Profiler 所依赖的 Attach、JVMTI、Instrumentation、JMX 等皆是 JVM 平台比较通用的技术，在此基础上，我们去实现 Memory Profiler、Thread Profiler、GC Analyzer 等工具也没有想象中那么神秘和复杂了。

参考资料

- JVM Tool Interface
- The Pros and Cons of AsyncGetCallTrace Profilers
- Why (Most) Sampling Java Profilers Are Fucking Terrible
- Safepoints: Meaning, Side Effects and Overheads
- Serviceability in HotSpot
- 如何读懂火焰图?
- IntelliJ IDEA 2018.3 EAP: Git Submodules, JVM Profiler (macOS and Linux) and more

作者简介

业祥，继东，美团基础架构部 / 服务框架组工程师。

团队信息

美团点评基础架构团队诚招高级、资深技术专家，Base 北京、上海。我们致力于建设美团点评全公司统一的高并发高性能分布式基础架构平台，涵盖数据库、分布式监控、服务治理、高性能通信、消息中间件、基础存储、容器化、集群调度等基础架构主要的技术领域。欢迎有兴趣的同学投送简历至：tech@meituan.com。

Java 动态调试技术原理及实践

胡健

断点调试是我们最常使用的调试手段，它可以获取到方法执行过程中的变量信息，并可以观察到方法的执行路径。但断点调试会在断点位置停顿，使得整个应用停止响应。在线上停顿应用是致命的，动态调试技术给了我们创造新的调试模式的想象空间。本文将研究 Java 语言中的动态调试技术，首先概括 Java 动态调试所涉及的技术基础，接着介绍我们在 Java 动态调试领域的思考及实践，通过结合实际业务场景，设计并实现了一种具备动态性的断点调试工具 Java-debug-tool，显著提高了故障排查效率。

JVMTI（JVM Tool Interface）是 Java 虚拟机对外提供的 Native 编程接口，通过 JVMTI，外部进程可以获取到运行时 JVM 的诸多信息，比如线程、GC 等。Agent 是一个运行在目标 JVM 的特定程序，它的职责是负责从目标 JVM 中获取数据，然后将数据传递给外部进程。加载 Agent 的时机可以是目标 JVM 启动之时，也可以是在目标 JVM 运行时进行加载，而在目标 JVM 运行时进行 Agent 加载具备动态性，对于时机未知的 Debug 场景来说非常实用。下面将详细分析 Java Agent 技术的实现细节。

2.1 Agent 的实现模式

JVMTI 是一套 Native 接口，在 Java SE 5 之前，要实现一个 Agent 只能通过编写 Native 代码来实现。从 Java SE 5 开始，可以使用 Java 的 Instrumentation 接口（java.lang.instrument）来编写 Agent。无论是通过 Native 的方式还是通过 Java Instrumentation 接口的方式来编写 Agent，它们的工作都是借助 JVMTI 来进行完成，下面介绍通过 Java Instrumentation 接口编写 Agent 的方法。

2.1.1 通过 Java Instrumentation API

- 实现 Agent 启动方法

Java Agent 支持目标 JVM 启动时加载，也支持在目标 JVM 运行时加载，这两种不同的加载模式会使用不同的入口函数，如果需要在目标 JVM 启动的同时加载 Agent，那么可以选择实现下面的方法：

```
[1] public static void premain(String agentArgs, Instrumentation inst);
[2] public static void premain(String agentArgs);
```

JVM 将首先寻找 [1]，如果没有发现 [1]，再寻找 [2]。如果希望在目标 JVM 运行时加载 Agent，则需要实现下面的方法：

```
[1] public static void agentmain(String agentArgs, Instrumentation inst);
[2] public static void agentmain(String agentArgs);
```

这两组方法的第一个参数 AgentArgs 是随同“- javaagent”一起传入的程序参数，如果这个字符串代表了多个参数，就需要自己解析这些参数。inst 是 Instrumentation 类型的对象，是 JVM 自动传入的，我们可以拿这个参数进行类增强等操作。

- 指定 Main-Class

Agent 需要打包成一个 jar 包，在 ManiFest 属性中指定“Premain-Class”或者“Agent-Class”：

```
Premain-Class: class
Agent-Class: class
```

- 挂载到目标 JVM

将编写的 Agent 打成 jar 包后，就可以挂载到目标 JVM 上去了。如果选择在目标 JVM 启动时加载 Agent，则可以使用“-javaagent:[=]“，具体的使用方法可以使用“Java -Help”来查看。如果想要在运行时挂载 Agent 到目标 JVM，就需要做一些额外的开发了。

com.sun.tools.attach.VirtualMachine 这个类代表一个 JVM 抽象，可以通过这个类找到目标 JVM，并且将 Agent 挂载到目标 JVM 上。下面是使用 com.sun.tools.attach.VirtualMachine 进行动态挂载 Agent 的一般实现：

```
    private void attachAgentToTargetJVM() throws Exception {
        List<VirtualMachineDescriptor> virtualMachineDescriptors =
VirtualMachine.list();
        VirtualMachineDescriptor targetVM = null;
        for (VirtualMachineDescriptor descriptor :
virtualMachineDescriptors) {
            if (descriptor.id().equals(configure.getPid())) {
                targetVM = descriptor;
                break;
            }
        }
        if (targetVM == null) {
            throw new IllegalArgumentException("could not find the
target jvm by process id:" +
configure.getPid());
        }
        VirtualMachine virtualMachine = null;
        try {
            virtualMachine = VirtualMachine.attach(targetVM);
            virtualMachine.loadAgent("{agent}", "{params}");
        } catch (Exception e) {
            if (virtualMachine != null) {
                virtualMachine.detach();
            }
        }
    }
```

首先通过指定的进程 ID 找到目标 JVM，然后通过 Attach 挂载到目标 JVM 上，执行加载 Agent 操作。VirtualMachine 的 Attach 方法就是用来将 Agent 挂载到目标 JVM 上去的，而 Detach 则是将 Agent 从目标 JVM 卸载。关于 Agent 是如何挂载到目标 JVM 上的具体技术细节，将在下文中进行分析。

2.2 启动时加载 Agent

2.2.1 参数解析

创建 JVM 时，JVM 会进行参数解析，即解析那些用来配置 JVM 启动的参数，比如堆大小、GC 等；本文主要关注解析的参数为 -agentlib、-agentpath、-javaagent，这几个参数用来指定 Agent，JVM 会根据这几个参数加载 Agent。下面来分析一下 JVM 是如何解析这几个参数的。

```
  // -agentlib and -agentpath
  if (match_option(option, "-agentlib:", &tail) ||
          (is_absolute_path = match_option(option, "-agentpath:",
&tail))) {
      if(tail != NULL) {
        const char* pos = strchr(tail, '=');
        size_t len = (pos == NULL) ? strlen(tail) : pos - tail;
        char* name = strncpy(NEW_C_HEAP_ARRAY(char, len + 1,
mtArguments), tail, len);
        name[len] = '\0';
        char *options = NULL;
        if(pos != NULL) {
          options = os::strdup_check_oom(pos + 1, mtArguments);
        }
#if !INCLUDE_JVMTI
        if (valid_jdwp_agent(name, is_absolute_path)) {
          jio_fprintf(defaultStream::error_stream(),
            "Debugging agents are not supported in this VM\n");
          return JNI_ERR;
        }
#endif // !INCLUDE_JVMTI
        add_init_agent(name, options, is_absolute_path);
      }
    // -javaagent
    } else if (match_option(option, "-javaagent:", &tail)) {
#if !INCLUDE_JVMTI
      jio_fprintf(defaultStream::error_stream(),
        "Instrumentation agents are not supported in this VM\n");
      return JNI_ERR;
#else
      if (tail != NULL) {
        size_t length = strlen(tail) + 1;
        char *options = NEW_C_HEAP_ARRAY(char, length, mtArguments);
        jio_snprintf(options, length, "%s", tail);
        add_init_agent("instrument", options, false);
        // java agents need module java.instrument
        if (!create_numbered_property("jdk.module.addmods", "java.
instrument", addmods_
count++)) {
          return JNI_ENOMEM;
        }
      }
#endif // !INCLUDE_JVMTI
    }
```

上面的代码片段截取自 hotspot/src/share/vm/runtime/arguments.cpp 中的 Arguments::parse_each_vm_init_arg(const JavaVMInitArgs* args, bool* patch_

mod_javabase, Flag::Flags origin) 函数，该函数用来解析一个具体的 JVM 参数。这段代码的主要功能是解析出需要加载的 Agent 路径，然后调用 add_init_agent 函数进行解析结果的存储。下面先看一下 add_init_agent 函数的具体实现：

```
  // -agentlib and -agentpath arguments
  static AgentLibraryList _agentList;
  static void add_init_agent(const char* name, char* options, bool
absolute_path)
    { _agentList.add(new AgentLibrary(name, options, absolute_path,
NULL)); }
```

AgentLibraryList 是一个简单的链表结构，add_init_agent 函数将解析好的、需要加载的 Agent 添加到这个链表中，等待后续的处理。

这里需要注意，解析 -javaagent 参数有一些特别之处，这个参数用来指定一个我们通过 Java Instrumentation API 来编写的 Agent，Java Instrumentation API 底层依赖的是 JVMTI，对 -JavaAgent 的处理也说明了这一点，在调用 add_init_agent 函数时第一个参数是“instrument”，关于加载 Agent 这个问题在下一小节进行展开。到此，我们知道在启动 JVM 时指定的 Agent 已经被 JVM 解析完存放在了一个链表结构中。下面来分析一下 JVM 是如何加载这些 Agent 的。

2.2.2 执行加载操作

在创建 JVM 进程的函数中，解析完 JVM 参数之后，下面的这段代码和加载 Agent 相关：

```
  // Launch -agentlib/-agentpath and converted -Xrun agents
  if (Arguments::init_agents_at_startup()) {
    create_vm_init_agents();
  }
  static bool init_agents_at_startup() {
    return !_agentList.is_empty();
  }
```

当 JVM 判断出上一小节中解析出来的 Agent 不为空的时候，就要去调用函数 create_vm_init_agents 来加载 Agent，下面来分析一下 create_vm_init_agents 函数是如何加载 Agent 的。

```
void Threads::create_vm_init_agents() {
  AgentLibrary* agent;
  for (agent = Arguments::agents(); agent != NULL; agent = agent->next())
{
    OnLoadEntry_t  on_load_entry = lookup_agent_on_load(agent);
    if (on_load_entry != NULL) {
      // Invoke the Agent_OnLoad function
      jint err = (*on_load_entry)(&main_vm, agent->options(), NULL);
    }
  }
}
```

create_vm_init_agents 这个函数通过遍历 Agent 链表来逐个加载 Agent。通过这段代码可以看出，首先通过 lookup_agent_on_load 来加载 Agent 并且找到 Agent_OnLoad 函数，这个函数是 Agent 的入口函数。如果没找到这个函数，则认为是加载了一个不合法的 Agent，则什么也不做，否则调用这个函数，这样 Agent 的代码就开始执行起来了。对于使用 Java Instrumentation API 来编写 Agent 的方式来说，在解析阶段观察到在 add_init_agent 函数里面传递进去的是一个叫做”instrument”的字符串，其实这是一个动态链接库。在 Linux 里面，这个库叫做 libinstrument.so，在 BSD 系统中叫做 libinstrument.dylib，该动态链接库在 {JAVA_HOME}/jre/lib/ 目录下。

2.2.3 instrument 动态链接库

libinstrument 用来支持使用 Java Instrumentation API 来编写 Agent，在 libinstrument 中有一个非常重要的类称为：JPLISAgent（Java Programming Language Instrumentation Services Agent），它的作用是初始化所有通过 Java Instrumentation API 编写的 Agent，并且也承担着通过 JVMTI 实现 Java Instrumentation 中暴露 API 的责任。

我们已经知道，在 JVM 启动的时候，JVM 会通过 –javaagent 参数加载 Agent。最开始加载的是 libinstrument 动态链接库，然后在动态链接库里面找到 JVMTI 的入口方法：Agent_OnLoad。下面就来分析一下在 libinstrument 动态链接库中，Agent_OnLoad 函数是怎么实现的。

```
JNIEXPORT jint JNICALL
DEF_Agent_OnLoad(JavaVM *vm, char *tail, void * reserved) {
    initerror = createNewJPLISAgent(vm, &agent);
    if ( initerror == JPLIS_INIT_ERROR_NONE ) {
        if (parseArgumentTail(tail, &jarfile, &options) != 0) {
            fprintf(stderr, "-javaagent: memory allocation failure.\n");
            return JNI_ERR;
        }
        attributes = readAttributes(jarfile);
        premainClass = getAttribute(attributes, "Premain-Class");
        /* Save the jarfile name */
        agent->mJarfile = jarfile;
        /*
         * Convert JAR attributes into agent capabilities
         */
        convertCapabilityAttributes(attributes, agent);
        /*
         * Track (record) the agent class name and options data
         */
        initerror = recordCommandLineData(agent, premainClass, options);
    }
    return result;
}
```

上述代码片段是经过精简的 libinstrument 中 Agent_OnLoad 实现的，大概的流程就是：先创建一个 JPLISAgent，然后将 ManiFest 中设定的一些参数解析出来，比如（Premain-Class）等。创建了 JPLISAgent 之后，调用 initializeJPLISAgent 对这个 Agent 进行初始化操作。跟进 initializeJPLISAgent 看一下是如何初始化的：

```
JPLISInitializationError initializeJPLISAgent(JPLISAgent *agent, JavaVM
*vm, jvmtiEnv *jvmtienv)
{
    /* check what capabilities are available */
    checkCapabilities(agent);
    /* check phase - if live phase then we don't need the VMInit event */
    jvmtierror = (*jvmtienv)->GetPhase(jvmtienv, &phase);
    /* now turn on the VMInit event */
    if ( jvmtierror == JVMTI_ERROR_NONE ) {
        jvmtiEventCallbacks callbacks;
        memset(&callbacks, 0, sizeof(callbacks));
        callbacks.VMInit = &eventHandlerVMInit;
        jvmtierror = (*jvmtienv)->SetEventCallbacks(jvmtienv,&callbacks,
sizeof(callbacks));
    }
    if ( jvmtierror == JVMTI_ERROR_NONE ) {
```

```
        jvmtierror = (*jvmtienv)->SetEventNotificationMode(jvmtienv,JVM
TI_ENABLE,JVMTI_EVENT_
VM_INIT,NULL);
    }
    return (jvmtierror == JVMTI_ERROR_NONE)? JPLIS_INIT_ERROR_NONE :
JPLIS_INIT_ERROR_
FAILURE;
}
```

这里，我们关注 callbacks.VMInit = &eventHandlerVMInit; 这行代码，这里设置了一个 VMInit 事件的回调函数，表示在 JVM 初始化的时候会回调 eventHandlerVMInit 函数。下面来看一下这个函数的实现细节，猜测就是在这里调用了 Premain 方法：

```
void JNICALL  eventHandlerVMInit( jvmtiEnv *jvmtienv,JNIEnv
*jnienv,jthread thread) {
   // ...
   success = processJavaStart( environment->mAgent, jnienv);
  // ...
}
jboolean  processJavaStart(JPLISAgent *agent,JNIEnv *jnienv) {
    result = createInstrumentationImpl(jnienv, agent);
    /*
     *  Load the Java agent, and call the premain.
     */
    if ( result ) {
        result = startJavaAgent(agent, jnienv, agent->mAgentClassName,
agent->mOptionsString,
agent->mPremainCaller);
    }
    return result;
}
jboolean startJavaAgent( JPLISAgent *agent,JNIEnv *jnienv,const char
*classname,const char
*optionsString,jmethodID agentMainMethod) {
  // ...
  invokeJavaAgentMainMethod(jnienv,agent->mInstrumentationImpl,agentMainM
ethod,
classNameObject,optionsStringObject);
  // ...
}
```

看到这里，Instrument 已经实例化，invokeJavaAgentMainMethod 这个方法将我们的 premain 方法执行起来了。接着，我们就可以根据 Instrument 实例来做我

们想要做的事情了。

2.3 运行时加载 Agent

比起 JVM 启动时加载 Agent，运行时加载 Agent 就比较有诱惑力了，因为运行时加载 Agent 的能力给我们提供了很强的动态性，我们可以在需要的时候加载 Agent 来进行一些工作。因为是动态的，我们可以按照需求来加载所需要的 Agent，下面来分析一下动态加载 Agent 的相关技术细节。

2.3.1 AttachListener

Attach 机制通过 Attach Listener 线程来进行相关事务的处理，下面来看一下 Attach Listener 线程是如何初始化的。

```
// Starts the Attach Listener thread
void AttachListener::init() {
  // 创建线程相关部分代码被去掉了
  const char thread_name[] = "Attach Listener";
  Handle string = java_lang_String::create_from_str(thread_name, THREAD);
  { MutexLocker mu(Threads_lock);
    JavaThread* listener_thread = new JavaThread(&attach_listener_thread_
entry);
    // ...
  }
}
```

我们知道，一个线程启动之后都需要指定一个入口来执行代码，Attach Listener 线程的入口是 attach_listener_thread_entry，下面看一下这个函数的具体实现：

```
static void attach_listener_thread_entry(JavaThread* thread, TRAPS) {
  AttachListener::set_initialized();
  for (;;) {
      AttachOperation* op = AttachListener::dequeue();
      // find the function to dispatch too
      AttachOperationFunctionInfo* info = NULL;
      for (int i=0; funcs[i].name != NULL; i++) {
        const char* name = funcs[i].name;
        if (strcmp(op->name(), name) == 0) {
          info = &(funcs[i]); break;
        }}
       // dispatch to the function that implements this operation
```

```
        res = (info->func)(op, &st);
      //...
    }
}
```

整个函数执行逻辑，大概是这样的：

- 拉取一个需要执行的任务：AttachListener::dequeue。
- 查询匹配的命令处理函数。
- 执行匹配到的命令执行函数。

其中第二步里面存在一个命令函数表，整个表如下：

```
static AttachOperationFunctionInfo funcs[] = {
  { "agentProperties",  get_agent_properties },
  { "datadump",         data_dump },
  { "dumpheap",         dump_heap },
  { "load",             load_agent },
  { "properties",       get_system_properties },
  { "threaddump",       thread_dump },
  { "inspectheap",      heap_inspection },
  { "setflag",          set_flag },
  { "printflag",        print_flag },
  { "jcmd",             jcmd },
  { NULL,               NULL }
};
```

对于加载 Agent 来说，命令就是“load”。现在，我们知道了 Attach Listener 大概的工作模式，但是还是不太清楚任务从哪来，这个秘密就藏在 AttachListener::dequeue 这行代码里面，接下来我们来分析一下 dequeue 这个函数：

```
LinuxAttachOperation* LinuxAttachListener::dequeue() {
  for (;;) {
    // wait for client to connect
    struct sockaddr addr;
    socklen_t len = sizeof(addr);
    RESTARTABLE(::accept(listener(), &addr, &len), s);
    // get the credentials of the peer and check the effective uid/guid
    // - check with jeff on this.
    struct ucred cred_info;
    socklen_t optlen = sizeof(cred_info);
    if (::getsockopt(s, SOL_SOCKET, SO_PEERCRED, (void*)&cred_info,
&optlen) == -1) {
```

```
        ::close(s);
        continue;
      }
      // peer credential look okay so we read the request
      LinuxAttachOperation* op = read_request(s);
      return op;
    }
  }
```

这是 Linux 上的实现，不同的操作系统实现方式不太一样。上面的代码表面，Attach Listener 在某个端口监听着，通过 accept 来接收一个连接，然后从这个连接里面将请求读取出来，然后将请求包装成一个 AttachOperation 类型的对象，之后就会从表里查询对应的处理函数，然后进行处理。

Attach Listener 使用一种被称为“懒加载”的策略进行初始化，也就是说，JVM 启动的时候 Attach Listener 并不一定会启动起来。下面我们来分析一下这种“懒加载”策略的具体实现方案。

```
  // Start Attach Listener if +StartAttachListener or it can't be
started lazily
  if (!DisableAttachMechanism) {
    AttachListener::vm_start();
    if (StartAttachListener || AttachListener::init_at_startup()) {
      AttachListener::init();
    }
  }
// Attach Listener is started lazily except in the case when
// +ReduseSignalUsage is used
bool AttachListener::init_at_startup() {
  if (ReduceSignalUsage) {
    return true;
  } else {
    return false;
  }
}
```

上面的代码截取自 create_vm 函数，DisableAttachMechanism、StartAttachListener 和 ReduceSignalUsage 这三个变量默认都是 false，所以 AttachListener::init(); 这行代码不会在 create_vm 的时候执行，而 vm_start 会执行。下面来看一下这个函数的实现细节：

```
void AttachListener::vm_start() {
  char fn[UNIX_PATH_MAX];
  struct stat64 st;
  int ret;
  int n = snprintf(fn, UNIX_PATH_MAX, "%s/.java_pid%d",
           os::get_temp_directory(), os::current_process_id());
  assert(n < (int)UNIX_PATH_MAX, "java_pid file name buffer overflow");
  RESTARTABLE(::stat64(fn, &st), ret);
  if (ret == 0) {
    ret = ::unlink(fn);
    if (ret == -1) {
      log_debug(attach)("Failed to remove stale attach pid file at %s", fn);
    }
  }
}
```

这是在 Linux 上的实现，是将 /tmp/ 目录下的 .java_pid{pid} 文件删除，后面在创建 Attach Listener 线程的时候会创建出来这个文件。上面说到，AttachListener::init() 这行代码不会在 create_vm 的时候执行，这行代码的实现已经在上文中分析了，就是创建 Attach Listener 线程，并监听其他 JVM 的命令请求。现在来分析一下这行代码是什么时候被调用的，也就是“懒加载”到底是怎么加载起来的。

```
  // Signal Dispatcher needs to be started before VMInit event is posted
  os::signal_init();
```

这是 create_vm 中的一段代码，看起来跟信号相关，其实 Attach 机制就是使用信号来实现“懒加载“的。下面我们来仔细地分析一下这个过程。

```
void os::signal_init() {
  if (!ReduceSignalUsage) {
    // Setup JavaThread for processing signals
    EXCEPTION_MARK;
    Klass* k = SystemDictionary::resolve_or_fail(vmSymbols::java_lang_
Thread(), true, CHECK);
    instanceKlassHandle klass (THREAD, k);
    instanceHandle thread_oop = klass->allocate_instance_handle(CHECK);
    const char thread_name[] = "Signal Dispatcher";
    Handle string = java_lang_String::create_from_str(thread_name,
CHECK);
    // Initialize thread_oop to put it into the system threadGroup
    Handle thread_group (THREAD, Universe::system_thread_group());
    JavaValue result(T_VOID);
```

```
    JavaCalls::call_special(&result, thread_oop,klass,vmSymbols::object_
initializer_name(),vmSymbols::threadgroup_string_void_signature(),
                            thread_group,string,CHECK);
    KlassHandle group(THREAD, SystemDictionary::ThreadGroup_klass());
    JavaCalls::call_special(&result,thread_group,group,vmSymbols::add_
method_name(),vmSymbols::thread_void_signature(),thread_oop,CHECK);
    os::signal_init_pd();
    { MutexLocker mu(Threads_lock);
      JavaThread* signal_thread = new JavaThread(&signal_thread_entry);
     // ...
    }
    // Handle ^BREAK
    os::signal(SIGBREAK, os::user_handler());
  }
}
```

JVM 创建了一个新的进程来实现信号处理，这个线程叫“Signal Dispatcher”，一个线程创建之后需要有一个入口，“Signal Dispatcher”的入口是 signal_thread_entry：

```
263
264       switch (sig) {
265         case SIGBREAK: {
266           // Check if the signal is a trigger to start the Attach Listener - in that
267           // case don't print stack traces.
268           if (!DisableAttachMechanism && AttachListener::is_init_trigger()) {
269             continue;
270           }
```

这段代码截取自 signal_thread_entry 函数，截取中的内容是和 Attach 机制信号处理相关的代码。这段代码的意思是，当接收到“SIGBREAK”信号，就执行接下来的代码，这个信号是需要 Attach 到 JVM 上的信号发出来，这个后面会再分析。我们先来看一句关键的代码：AttachListener::is_init_trigger()：

```
bool AttachListener::is_init_trigger() {
  if (init_at_startup() || is_initialized()) {
    return false;               // initialized at startup or already
initialized
  }
  char fn[PATH_MAX+1];
  sprintf(fn, ".attach_pid%d", os::current_process_id());
  int ret;
  struct stat64 st;
  RESTARTABLE(::stat64(fn, &st), ret);
  if (ret == -1) {
```

```
    log_trace(attach)("Failed to find attach file: %s, trying
alternate", fn);
    snprintf(fn, sizeof(fn), "%s/.attach_pid%d", os::get_temp_
directory(), os::current_process_
id());
    RESTARTABLE(::stat64(fn, &st), ret);
  }
  if (ret == 0) {
    // simple check to avoid starting the attach mechanism when
    // a bogus user creates the file
    if (st.st_uid == geteuid()) {
      init();
      return true;
    }
  }
  return false;
}
```

首先检查了一下是否在 JVM 启动时启动了 Attach Listener，或者是否已经启动过。如果没有，才继续执行，在 /tmp 目录下创建一个叫做 .attach_pid%d 的文件，然后执行 AttachListener 的 init 函数，这个函数就是用来创建 Attach Listener 线程的函数，上面已经提到多次并进行了分析。到此，我们知道 Attach 机制的奥秘所在，也就是 Attach Listener 线程的创建依靠 Signal Dispatcher 线程，Signal Dispatcher 是用来处理信号的线程，当 Signal Dispatcher 线程接收到“SIGBREAK”信号之后，就会执行初始化 Attach Listener 的工作。

2.3.2 运行时加载 Agent 的实现

我们继续分析，到底是如何将一个 Agent 挂载到运行着的目标 JVM 上，在上文中提到了一段代码，用来进行运行时挂载 Agent，可以参考上文中展示的关于“attachAgentToTargetJvm”方法的代码。这个方法里面的关键是调用 VirtualMachine 的 attach 方法进行 Agent 挂载的功能。下面我们就来分析一下 VirtualMachine 的 attach 方法具体是怎么实现的。

```
    public static VirtualMachine attach(String var0) throws
AttachNotSupportedException,
IOException {
        if (var0 == null) {
            throw new NullPointerException("id cannot be null");
```

```
        } else {
            List var1 = AttachProvider.providers();
            if (var1.size() == 0) {
                throw new AttachNotSupportedException("no providers
installed");
            } else {
                AttachNotSupportedException var2 = null;
                Iterator var3 = var1.iterator();
                while(var3.hasNext()) {
                    AttachProvider var4 = (AttachProvider)var3.next();
                    try {
                        return var4.attachVirtualMachine(var0);
                    } catch (AttachNotSupportedException var6) {
                        var2 = var6;
                    }
                }
                throw var2;
            }
        }
    }
```

这个方法通过 attachVirtualMachine 方法进行 attach 操作，在 MacOS 系统中，AttachProvider 的实现类是 BsdAttachProvider。我们来看一下 BsdAttach-Provider 的 attachVirtualMachine 方法是如何实现的：

```
public VirtualMachine attachVirtualMachine(String var1) throws
AttachNotSupportedException, IOException {
        this.checkAttachPermission();
        this.testAttachable(var1);
        return new BsdVirtualMachine(this, var1);
    }
BsdVirtualMachine(AttachProvider var1, String var2) throws
AttachNotSupportedException,
IOException {
        int var3 = Integer.parseInt(var2);
        this.path = this.findSocketFile(var3);
        if (this.path == null) {
            File var4 = new File(tmpdir, ".attach_pid" + var3);
            createAttachFile(var4.getPath());
            try {
                sendQuitTo(var3);
                int var5 = 0;
                long var6 = 200L;
                int var8 = (int)(this.attachTimeout() / var6);
                do {
                    try {
```

```
                        Thread.sleep(var6);
                    } catch (InterruptedException var21) {
                        ;
                    }
                    this.path = this.findSocketFile(var3);
                    ++var5;
                } while(var5 <= var8 && this.path == null);
            } finally {
                var4.delete();
            }
        }
        int var24 = socket();
        connect(var24, this.path);
    }
    private String findSocketFile(int var1) {
        String var2 = ".java_pid" + var1;
        File var3 = new File(tmpdir, var2);
        return var3.exists() ? var3.getPath() : null;
    }
```

findSocketFile 方法用来查询目标 JVM 上是否已经启动了 Attach Listener，它通过检查”tmp/“目录下是否存在 java_pid{pid} 来进行实现。如果已经存在了，则说明 Attach 机制已经准备就绪，可以接受客户端的命令了，这个时候客户端就可以通过 connect 连接到目标 JVM 进行命令的发送，比如可以发送“load”命令来加载 Agent。如果 java_pid{pid} 文件还不存在，则需要通过 sendQuitTo 方法向目标 JVM 发送一个“SIGBREAK”信号，让它初始化 Attach Listener 线程并准备接受客户端连接。可以看到，发送了信号之后客户端会循环等待 java_pid{pid} 这个文件，之后再通过 connect 连接到目标 JVM 上。

2.3.3 load 命令的实现

下面来分析一下，“load”命令在 JVM 层面的实现：

```
static jint load_agent(AttachOperation* op, outputStream* out) {
  // get agent name and options
  const char* agent = op->arg(0);
  const char* absParam = op->arg(1);
  const char* options = op->arg(2);
  // If loading a java agent then need to ensure that the java.
instrument module is loaded
  if (strcmp(agent, "instrument") == 0) {
```

```
    Thread* THREAD = Thread::current();
    ResourceMark rm(THREAD);
    HandleMark hm(THREAD);
    JavaValue result(T_OBJECT);
    Handle h_module_name = java_lang_String::create_from_str("java.
instrument", THREAD);
    JavaCalls::call_static(&result,SystemDictionary::module_Modules_
klass(),vmSymbols::loadModule_name(),
                           vmSymbols::loadModule_signature(),h_module_
name,THREAD);
  }
  return JvmtiExport::load_agent_library(agent, absParam, options, out);
}
```

这个函数先确保加载了 java.instrument 模块，之后真正执行 Agent 加载的函数是 load_agent_library，这个函数的套路就是加载 Agent 动态链接库，如果是通过 Java instrument API 实现的 Agent，则加载的是 libinstrument 动态链接库，然后通过 libinstrument 里面的代码实现运行 agentmain 方法的逻辑，这一部分内容和 libinstrument 实现 premain 方法运行的逻辑其实差不多，这里不再做分析。至此，我们对 Java Agent 技术已经有了一个全面而细致的了解。

3.1 动态字节码修改的限制

上文中已经详细分析了 Agent 技术的实现，我们使用 Java Instrumentation API 来完成动态类修改的功能，在 Instrumentation 接口中，通过 addTransformer 方法来增加一个类转换器，类转换器由类 ClassFileTransformer 接口实现。ClassFileTransformer 接口中唯一的方法 transform 用于实现类转换，当类被加载的时候，就会调用 transform 方法，进行类转换。在运行时，我们可以通过 Instrumentation 的 redefineClasses 方法进行类重定义，在方法上有一段注释需要特别注意：

```
     * The redefinition may change method bodies, the constant pool and
attributes.
     * The redefinition must not add, remove or rename fields or methods,
change the
     * signatures of methods, or change inheritance.  These restrictions
maybe be
```

```
     * lifted in future versions.  The class file bytes are not checked,
verified and installed
     * until after the transformations have been applied, if the
resultant bytes are in
     * error this method will throw an exception.
```

这里面提到，我们不可以增加、删除或者重命名字段和方法，改变方法的签名或者类的继承关系。认识到这一点很重要，当我们通过 ASM 获取到增强的字节码之后，如果增强后的字节码没有遵守这些规则，那么调用 redefineClasses 方法来进行类的重定义就会失败。那 redefineClasses 方法具体是怎么实现类的重定义的呢？它对运行时的 JVM 会造成什么样的影响呢？下面来分析 redefineClasses 的实现细节。

3.2 重定义类字节码的实现细节

上文中我们提到，libinstrument 动态链接库中，JPLISAgent 不仅实现了 Agent 入口代码执行的路由，而且还是 Java 代码与 JVMTI 之间的一道桥梁。我们在 Java 代码中调用 Java Instrumentation API 的 redefineClasses，其实会调用 libinstrument 中的相关代码，我们来分析一下这条路径。

```
    public void redefineClasses(ClassDefinition... var1) throws
ClassNotFoundException {
        if (!this.isRedefineClassesSupported()) {
            throw new UnsupportedOperationException("redefineClasses is
not supported in this environment");
        } else if (var1 == null) {
            throw new NullPointerException("null passed as
'definitions' in redefineClasses");
        } else {
            for(int var2 = 0; var2 < var1.length; ++var2) {
                if (var1[var2] == null) {
                    throw new NullPointerException("element of
'definitions' is null in redefineClasses");
                }
            }
            if (var1.length != 0) {
                this.redefineClasses0(this.mNativeAgent, var1);
            }
        }
    }
```

```
    private native void redefineClasses0(long var1, ClassDefinition[]
var3) throws
ClassNotFoundException;
```

这是 InstrumentationImpl 中的 redefineClasses 实现，该方法的具体实现依赖一个 Native 方法 redefineClasses()，我们可以在 libinstrument 中找到这个 Native 方法的实现：

```
JNIEXPORT void JNICALL Java_sun_instrument_InstrumentationImpl_
redefineClasses0
  (JNIEnv * jnienv, jobject implThis, jlong agent, jobjectArray
classDefinitions) {
    redefineClasses(jnienv, (JPLISAgent*)(intptr_t)agent,
classDefinitions);
}
```

redefineClasses 这个函数的实现比较复杂，代码很长。下面是一段关键的代码片段：

```
if (!errorOccurred) {
    jvmtiError  errorCode = JVMTI_ERROR_NONE;
    errorCode = (*jvmtienv)->RedefineClasses(jvmtienv, numDefs, classDefs);
    if (errorCode == JVMTI_ERROR_WRONG_PHASE) {
        /* insulate caller from the wrong phase error */
        errorCode = JVMTI_ERROR_NONE;
    } else {
        errorOccurred = (errorCode != JVMTI_ERROR_NONE);
        if ( errorOccurred ) {
            createAndThrowThrowableFromJVMTIErrorCode(jnienv, errorCode);
        }
    }
}
```

可以看到，其实是调用了 JVMTI 的 RetransformClasses 函数来完成类的重定义细节。

```
// class_count - pre-checked to be greater than or equal to 0
// class_definitions - pre-checked for NULL
jvmtiError JvmtiEnv::RedefineClasses(jint class_count, const
jvmtiClassDefinition* class_
definitions) {
//TODO: add locking
  VM_RedefineClasses op(class_count, class_definitions, jvmti_class_
load_kind_redefine);
  VMThread::execute(&op);
```

```
    return (op.check_error());
  } /* end RedefineClasses */
```

重定义类的请求会被 JVM 包装成一个 VM_RedefineClasses 类型的 VM_Operation，VM_Operation 是 JVM 内部的一些操作的基类，包括 GC 操作等。VM_Operation 由 VMThread 来执行，新的 VM_Operation 操作会被添加到 VMThread 的运行队列中去，VMThread 会不断从队列里面拉取 VM_Operation 并调用其 doit 等函数执行具体的操作。VM_RedefineClasses 函数的流程较为复杂，下面是 VM_RedefineClasses 的大致流程：

- 加载新的字节码，合并常量池，并且对新的字节码进行校验工作

```
    // Load the caller's new class definition(s) into _scratch_classes.
    // Constant pool merging work is done here as needed. Also calls
    // compare_and_normalize_class_versions() to verify the class
    // definition(s).
    jvmtiError load_new_class_versions(TRAPS);
```

- 清除方法上的断点

```
    // Remove all breakpoints in methods of this class
    JvmtiBreakpoints& jvmti_breakpoints = JvmtiCurrentBreakpoints::get_
  jvmti_breakpoints();
    jvmti_breakpoints.clearall_in_class_at_safepoint(the_class());
```

- JIT 逆优化

```
    // Deoptimize all compiled code that depends on this class
    flush_dependent_code(the_class, THREAD);
```

- 进行字节码替换工作，需要进行更新类 itable/vtable 等操作
- 进行类重定义通知

```
    SystemDictionary::notice_modification();
```

VM_RedefineClasses 实现比较复杂的，详细实现可以参考 RedefineClasses 的实现。

Java-debug-tool 是一个使用 Java Instrument API 来实现的动态调试工具，它通过在目标 JVM 上启动一个 TcpServer 来和调试客户端通信。调试客户端通过命令行来发送调试命令给 TcpServer，TcpServer 中有专门用来处理命令的 handler，handler 处理完命令之后会将结果发送回客户端，客户端通过处理将调试结果展示出来。下面将详细介绍 Java-debug-tool 的整体设计和实现。

4.1 Java-debug-tool 整体架构

Java-debug-tool 包括一个 Java Agent 和一个用于处理调试命令的核心 API，核心 API 通过一个自定义的类加载器加载进来，以保证目标 JVM 的类不会被污染。整体上 Java-debug-tool 的设计是一个 Client-Server 的架构，命令客户端需要完整的完成一个命令之后才能继续执行下一个调试命令。Java-debug-tool 支持多人同时进行调试，下面是整体架构图：

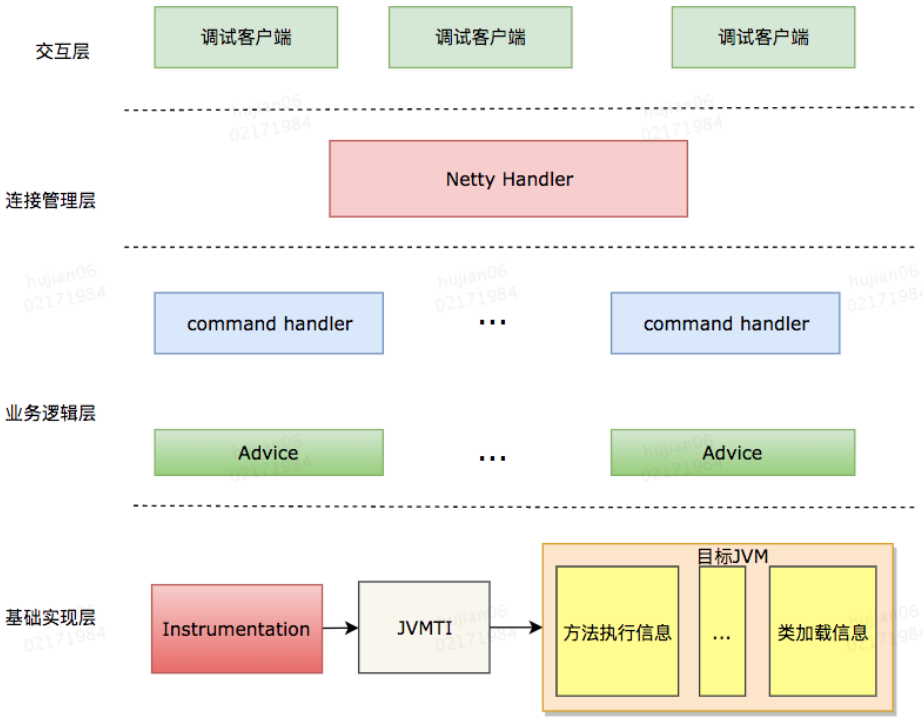


图 1

下面对每一层做简单介绍：

- 交互层：负责将程序员的输入转换成调试交互协议，并且将调试信息呈现出来。
- 连接管理层：负责管理客户端连接，从连接中读调试协议数据并解码，对调试结果编码并将其写到连接中去；同时将那些超时未活动的连接关闭。
- 业务逻辑层：实现调试命令处理，包括命令分发、数据收集、数据处理等过程。
- 基础实现层：Java-debug-tool 实现的底层依赖，通过 Java Instrumentation 提供的 API 进行类查找、类重定义等能力，Java Instrumentation 底层依赖 JVMTI 来完成具体的功能。

在 Agent 被挂载到目标 JVM 上之后，Java-debug-tool 会安排一个 Spy 在目标 JVM 内活动，这个 Spy 负责将目标 JVM 内部的相关调试数据转移到命令处理模块，命令处理模块会处理这些数据，然后给客户端返回调试结果。命令处理模块会增强目标类的字节码来达到数据获取的目的，多个客户端可以共享一份增强过的字节码，无需重复增强。下面从 Java-debug-tool 的字节码增强方案、命令设计与实现等角度详细说明。

4.2 Java-debug-tool 的字节码增强方案

Java-debug-tool 使用字节码增强来获取到方法运行时的信息，比如方法入参、出参等，可以在不同的字节码位置进行增强，这种行为可以称为“插桩”，每个“桩”用于获取数据并将他转储出去。Java-debug-tool 具备强大的插桩能力，不同的桩负责获取不同类别的数据，下面是 Java-debug-tool 目前所支持的“桩”：

- 方法进入点：用于获取方法入参信息。
- Fields 获取点 1：在方法执行前获取到对象的字段信息。
- 变量存储点：获取局部变量信息。
- Fields 获取点 2：在方法退出前获取到对象的字段信息。
- 方法退出点：用于获取方法返回值。
- 抛出异常点：用于获取方法抛出的异常信息。

通过上面这些代码桩，Java-debug-tool 可以收集到丰富的方法执行信息，经过处理可以返回更加可视化的调试结果。

4.2.1 字节码增强

Java-debug-tool 在实现上使用了 ASM 工具来进行字节码增强，并且每个插桩点都可以进行配置，如果不想要什么信息，则没必要进行对应的插桩操作。这种可配置的设计是非常有必要的，因为有时候我们仅仅是想要知道方法的入参和出参，但 Java-debug-tool 却给我们返回了所有的调试信息，这样我们就得在众多的输出中找到我们所关注的内容。如果可以进行配置，则除了入参点和出参点外其他的桩都不插，那么就可以快速看到我们想要的调试数据，这种设计的本质是为了让调试者更加专注。下面是 Java-debug-tool 的字节码增强工作方式：

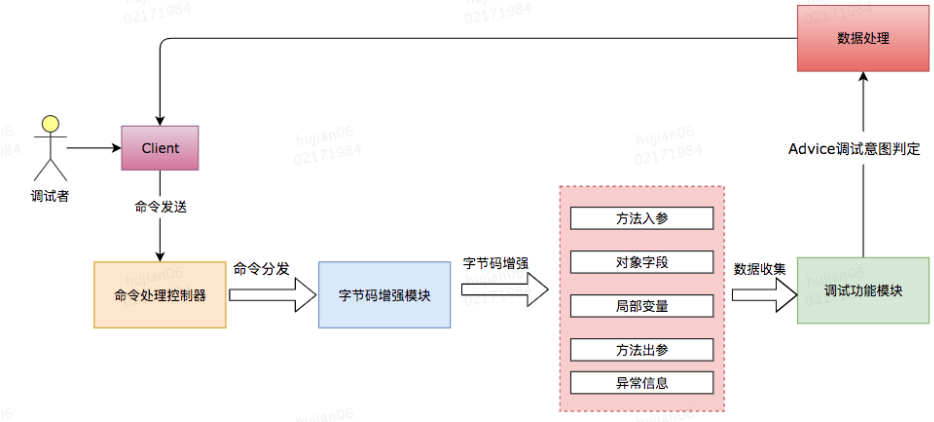


图 2

如图 2 所示，当调试者发出调试命令之后，Java-debug-tool 会识别命令并判断是否需要进行字节码增强，如果命令需要增强字节码，则判断当前类 + 当前方法是否已经被增强过。上文已经提到，字节码替换是有一定损耗的，这种具有损耗的操作发生的次数越少越好，所以字节码替换操作会被记录起来，后续命令直接使用即可，不需要重复进行字节码增强，字节码增强还涉及多个调试客户端的协同工作问题，当一个客户端增强了一个类的字节码之后，这个客户端就锁定了该字节码，其他客户端

变成只读，无法对该类进行字节码增强，只有当持有锁的客户端主动释放锁或者断开连接之后，其他客户端才能继续增强该类的字节码。

字节码增强模块收到字节码增强请求之后，会判断每个增强点是否需要插桩，这个判断的根据就是上文提到的插桩配置，之后字节码增强模块会生成新的字节码，Java-debug-tool 将执行字节码替换操作，之后就可以进行调试数据收集了。

经过字节码增强之后，原来的方法中会插入收集运行时数据的代码，这些代码在方法被调用的时候执行，获取到诸如方法入参、局部变量等信息，这些信息将传递给数据收集装置进行处理。数据收集的工作通过 Advice 完成，每个客户端同一时间只能注册一个 Advice 到 Java-debug-tool 调试模块上，多个客户端可以同时注册自己的 Advice 到调试模块上。Advice 负责收集数据并进行判断，如果当前数据符合调试命令的要求，Java-debug-tool 就会卸载这个 Advice，Advice 的数据就会被转移到 Java-debug-tool 的命令结果处理模块进行处理，并将结果发送到客户端。

4.2.2 Advice 的工作方式

Advice 是调试数据收集器，不同的调试策略会对应不同的 Advice。Advice 是工作在目标 JVM 的线程内部的，它需要轻量级和高效，意味着 Advice 不能做太过于复杂的事情，它的核心接口“match”用来判断本次收集到的调试数据是否满足调试需求。如果满足，那么 Java-debug-tool 就会将其卸载，否则会继续让他收集调试数据，这种“加载 Advice”->“卸载 Advice”的工作模式具备很好的灵活性。

关于 Advice，需要说明的另外一点就是线程安全，因为它加载之后会运行在目标 JVM 的线程中，目标 JVM 的方法极有可能是多线程访问的，这也就是说，Advice 需要有能力处理多个线程同时访问方法的能力，如果 Advice 处理不当，则可能会收集到杂乱无章的调试数据。下面的图片展示了 Advice 和 Java-debug-tool 调试分析模块、目标方法执行以及调试客户端等模块的关系。

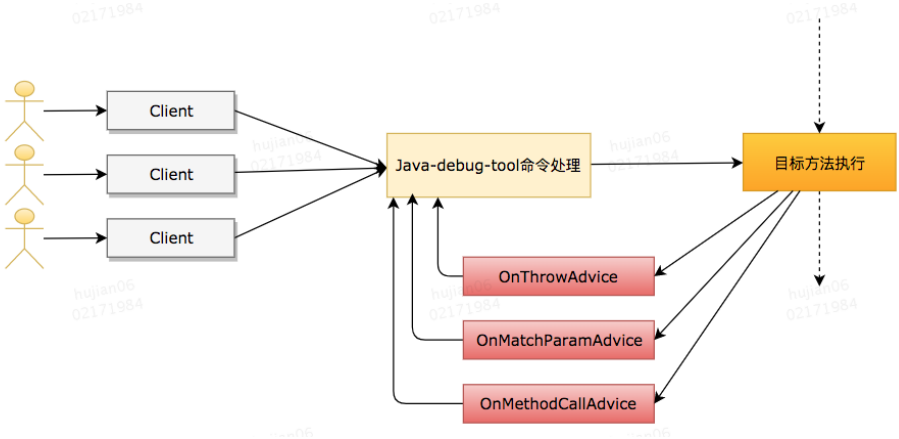


图 3

Advice 的首次挂载由 Java-debug-tool 的命令处理器完成，当一次调试数据收集完成之后，调试数据处理模块会自动卸载 Advice，然后进行判断，如果调试数据符合 Advice 的策略，则直接将数据交由数据处理模块进行处理，否则会清空调试数据，并再次将 Advice 挂载到目标方法上去，等待下一次调试数据。非首次挂载由调试数据处理模块进行，它借助 Advice 按需取数据，如果不符合需求，则继续挂载 Advice 来获取数据，否则对调试数据进行处理并返回给客户端。

4.3 Java-debug-tool 的命令设计与实现

4.3.1 命令执行

上文已经完整的描述了 Java-debug-tool 的设计以及核心技术方案，本小节将详细介绍 Java-debug-tool 的命令设计与实现。首先需要将一个调试命令的执行流程描述清楚，下面是一张用来表示命令请求处理流程的图片：

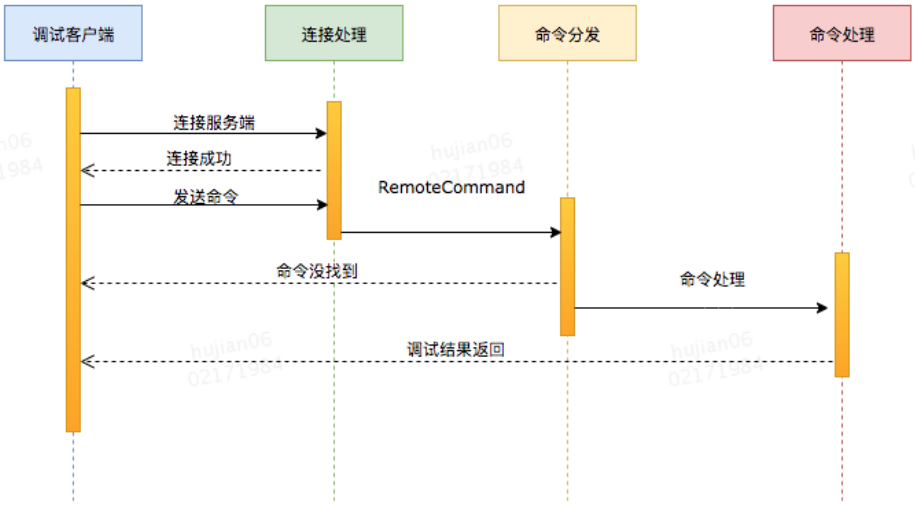


图 4

图 4 简单的描述了 Java-debug-tool 的命令处理方式，客户端连接到服务端之后，会进行一些协议解析、协议认证、协议填充等工作，之后将进行命令分发。服务端如果发现客户端的命令不合法，则会立即返回错误信息，否则再进行命令处理。命令处理属于典型的三段式处理，前置命令处理、命令处理以及后置命令处理，同时会对命令处理过程中的异常信息进行捕获处理，三段式处理的好处是命令处理被拆成了多个阶段，多个阶段负责不同的职责。前置命令处理用来做一些命令权限控制的工作，并填充一些类似命令处理开始时间戳等信息，命令处理就是通过字节码增强，挂载 Advice 进行数据收集，再经过数据处理来产生命令结果的过程，后置处理则用来处理一些连接关闭、字节码解锁等事项。

Java-debug-tool 允许客户端设置一个命令执行超时时间，超过这个时间则认为命令没有结果，如果客户端没有设置自己的超时时间，就使用默认的超时时间进行超时控制。Java-debug-tool 通过设计了两阶段的超时检测机制来实现命令执行超时功能：首先，第一阶段超时触发，则 Java-debug-tool 会友好的警告命令处理模块处理时间已经超时，需要立即停止命令执行，这允许命令自己做一些现场清理工作，当然需要命令执行线程自己感知到这种超时警告；当第二阶段超时触发，则

Java-debug-tool 认为命令必须结束执行，会强行打断命令执行线程。超时机制的目的是为了不让命令执行太长时间，命令如果长时间没有收集到调试数据，则应该停止执行，并思考是否调试了一个错误的方法。当然，超时机制还可以定期清理那些因为未知原因断开连接的客户端持有的调试资源，比如字节码锁。

4.3.2 获取方法执行视图

Java-debug-tool 通过下面的信息来向调试者呈现出一次方法执行的视图：

- 正在调试的方法信息。
- 方法调用堆栈。
- 调试耗时，包括对目标 JVM 造成的 STW 时间。
- 方法入参，包括入参的类型及参数值。
- 方法的执行路径。
- 代码执行耗时。
- 局部变量信息。
- 方法返回结果。
- 方法抛出的异常。
- 对象字段值快照。

图 5 展示了 Java-debug-tool 获取到正在运行的方法的执行视图的信息。

```
127.0.0.1:11234>mt -c R -m call -t watch -i #p0+#p1<10 -timeout 10 -s fd

-----------------------------------------------------------------------------------------------
命令                    : mt
命令执行Round           : 3
客户端ID                : 10000
客户端类型              : client:1
协议版本                : version:1
命令耗时                : 427 (ms)
STW时间                 : 0 (ms)
-----------------------------------------------------------------------------------------------
[R.call] by invoking:Thread[Thread-0,5,main]
with params
[
[0] @class:java.lang.Integer -> 2,
[1] @class:java.lang.Integer -> 4,
[2] @class:C -> {"@class":"C","a":0}
]
[1 ms] (34) [sa = 1]
[0 ms] (35)
[0 ms] (36) [ii = 0]
[0 ms] (37) [ij = 0]
[0 ms] (38) [jk = 1]
[0 ms] (39) [f = 1.0]
[0 ms] (40) [d = 0.123]
[0 ms] (41) [name = hello2,4]
[0 ms] (42) [list = [5]]
[0 ms] (43)
[0 ms] (47)
[0 ms] (50)
[0 ms] (51)
[0 ms] (53)
[0 ms] (57)
return value:[1]  at line:57 with cost:4 ms

Before Invoking Method
ClassField:
     |_ hello[world]@java.lang.String
ClassField:
     |_ input[11]@int

Before Exiting Method
ClassField:
     |_ hello[world]@java.lang.String
ClassField:
     |_ input[6]@int

-----------------------------------------------------------------------------------------------
```

图 5

4.4 Java-debug-tool 与同类产品对比分析

Java-debug-tool 的同类产品主要是 greys，其他类似的工具大部分都是基于 greys 进行的二次开发，所以直接选择 greys 来和 Java-debug-tool 进行对比。

对比项	greys	Java-debug-tool	说明
多调试客户端支持	支持	支持	都是用观察者模式来支持多调试客户端同时调试；
方法增强	入参/返回/抛出异常	入参/执行路径/局部变量/出参/抛出异常/对象字段	相比greys，Java-debug-tool有更多的插桩选择，意味着可以获取到更丰富的调试信息
工具使用难度	适中，但是命令较多，需要配和多个命令进行调试	适中，调试集中在很少几个命令上	因为是基于shell的交互模式，两种工具都存在一定的使用难度
类隔离	自定义类加载器加载工具代码与目标JVM的类进行隔离	使用自定义类加载器	这样对目标JVM不会产生类污染
Agent加载模式	运行时加载	运行时加载	这也符合"动态调试"的场景需求
交互模式	命令行	命令行	greys的调试服务端是一个使用NIO实现的tcpServer，Java-debug-tool使用Netty实现调试服务端
命令数量	10+	5+	greys提供了大量的命令，而Java-debug-tool则专注于方法级别的链路追踪，没有提供太多的命令
表达式支持	支持，使用OGNL表达式；	支持，使用Spring表达式	Java-debug-tool的表达式仅在方法追踪时有用；
方法追踪调试信息获取	指定具体的调试信息，比如方法退出前；tt命令可以记录方法的执行信息；	执行mt命令，默认即可获取到包括方法入参、执行路径、执行耗时（包括单行耗时）、方法出参、抛出的异常等丰富的信息；	Java-debug-tool的mt命令和单步调试更类似，区别就是Java-debug-tool没有断点；
对象打印	如果类没有覆盖toString方法则不能很好的获取到对象的信息；	当检测到类没有覆盖toString方法之后，对象会被打印成json；	
命令特点	命令多，每个命令各司其职	命令较少，mt命令相当于聚合了多个greys的命令，调试信息较为丰富；	
网络连接管理	一个客户端连接idle一段时间会被关闭	一个客户端连接idle一段时间会被关闭，可配置服务端是否在所有连接都关闭的情况下关闭服务；	
语言	Java	Java	

本文详细剖析了 Java 动态调试关键技术的实现细节，并介绍了我们基于 Java 动态调试技术结合实际故障排查场景进行的一点探索实践；动态调试技术为研发人员进行线上问题排查提供了一种新的思路，我们基于动态调试技术解决了传统断点调试存在的问题，使得可以将断点调试这种技术应用在线上，以线下调试的思维来进行线上调试，提高问题排查效率。

参考文献

- ASM 4 guide
- Java Virtual Machine Specification
- JVM Tool Interface
- alibaba arthas
- openjdk

作者简介

胡健，美团到店餐饮研发中心研发工程师。

招聘

美团到店餐饮研发中心支撑了美团核心业务 – 餐饮团购业务。团队核心成员并肩作战与美团一起打赢了千团大战，实力爆表，更有来自知名互联网公司的各路大牛加盟支持。团队正在经历从优秀到卓越的蜕变，挑战多，发展空间大。期待优秀的你加入我们，共同努力，让所有人吃的更好、生活更好！ 美团到店餐饮研发中心诚聘 Java 高级工程师、架构师、专家，欢迎有兴趣的同学投递简历到 tech@meituan.com（邮件标题注明：美团到店餐饮研发中心）

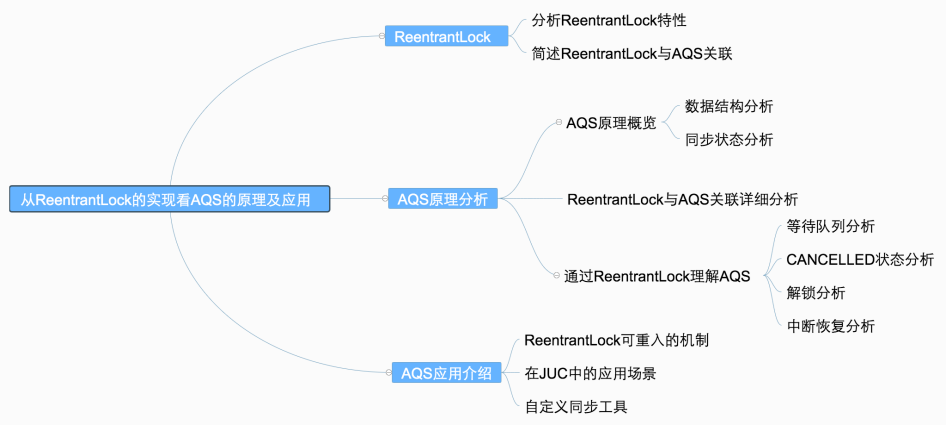
从 ReentrantLock 的实现看 AQS 的原理及应用

李卓

前言

Java 中的大部分同步类（Lock、Semaphore、ReentrantLock 等）都是基于 AbstractQueuedSynchronizer（简称为 AQS）实现的。AQS 是一种提供了原子式管理同步状态、阻塞和唤醒线程功能以及队列模型的简单框架。本文会从应用层逐渐深入到原理层，并通过 ReentrantLock 的基本特性和 ReentrantLock 与 AQS 的关联，来深入解读 AQS 相关独占锁的知识点，同时采取问答的模式来帮助大家理解 AQS。由于篇幅原因，本篇文章主要阐述 AQS 中独占锁的逻辑和 Sync Queue，不讲述包含共享锁和 Condition Queue 的部分（本篇文章核心为 AQS 原理剖析，只是简单介绍了 ReentrantLock，感兴趣同学可以阅读一下 ReentrantLock 的源码）。

下面列出本篇文章的大纲和思路，以便于大家更好地理解：



1. ReentrantLock

1.1 ReentrantLock 特性概览

ReentrantLock 意思为可重入锁，指的是一个线程能够对一个临界资源重复加锁。为了帮助大家更好地理解 ReentrantLock 的特性，我们先将 ReentrantLock 跟常用的 Synchronized 进行比较，其特性如下（蓝色部分为本篇文章主要剖析的点）：

	ReentrantLock	Synchronized
锁实现机制	依赖AQS	监视器模式
灵活性	支持响应中断、超时、尝试获取锁	不灵活
释放形式	必须显示调用unlock()释放锁	自动释放监视器
锁类型	公平锁&非公平锁	非公平锁
条件队列	可关联多个条件队列	关联一个条件队列
可重入性	可重入	可重入

下面通过伪代码，进行更加直观的比较：

```
// **************************Synchronized 的使用方式
**************************
// 1.用于代码块
synchronized (this) {}
// 2.用于对象
synchronized (object) {}
// 3.用于方法
public synchronized void test () {}
// 4.可重入
for (int i = 0; i < 100; i++) {
      synchronized (this) {}
}
// **************************ReentrantLock 的使用方式
**************************
```

```
public void test () throw Exception {
      // 1. 初始化选择公平锁、非公平锁
      ReentrantLock lock = new ReentrantLock(true);
      // 2. 可用于代码块
      lock.lock();
      try {
            try {
                  // 3. 支持多种加锁方式，比较灵活； 具有可重入特性
                  if(lock.tryLock(100, TimeUnit.MILLISECONDS)){ }
            } finally {
                  // 4. 手动释放锁
                  lock.unlock()
            }
      } finally {
            lock.unlock();
      }
}
```

1.2 ReentrantLock 与 AQS 的关联

通过上文我们已经了解，ReentrantLock 支持公平锁和非公平锁（关于公平锁和非公平锁的原理分析，可参考《不可不说的 Java“锁”事》），并且 ReentrantLock 的底层就是由 AQS 来实现的。那么 ReentrantLock 是如何通过公平锁和非公平锁与 AQS 关联起来呢？ 我们着重从这两者的加锁过程来理解一下它们与 AQS 之间的关系（加锁过程中与 AQS 的关联比较明显，解锁流程后续会介绍）。

非公平锁源码中的加锁流程如下：

```
// java.util.concurrent.locks.ReentrantLock#NonfairSync

// 非公平锁
static final class NonfairSync extends Sync {
      ...
      final void lock() {
            if (compareAndSetState(0, 1))
                  setExclusiveOwnerThread(Thread.currentThread());
            else
                  acquire(1);
            }
  ...
}
```

这块代码的含义为：

- 若通过 CAS 设置变量 State（同步状态）成功，也就是获取锁成功，则将当前线程设置为独占线程。
- 若通过 CAS 设置变量 State（同步状态）失败，也就是获取锁失败，则进入 Acquire 方法进行后续处理。

第一步很好理解，但第二步获取锁失败后，后续的处理策略是怎么样的呢？这块可能会有以下思考：

- 某个线程获取锁失败的后续流程是什么呢？有以下两种可能：

（1）将当前线程获锁结果设置为失败，获取锁流程结束。这种设计会极大降低系统的并发度，并不满足我们实际的需求。所以就需要下面这种流程，也就是 AQS 框架的处理流程。

（2）存在某种排队等候机制，线程继续等待，仍然保留获取锁的可能，获取锁流程仍在继续。

- 对于问题 1 的第二种情况，既然说到了排队等候机制，那么就一定会有某种队列形成，这样的队列是什么数据结构呢？
- 处于排队等候机制中的线程，什么时候可以有机会获取锁呢？
- 如果处于排队等候机制中的线程一直无法获取锁，还是需要一直等待吗，还是有别的策略来解决这一问题？

带着非公平锁的这些问题，再看下公平锁源码中获锁的方式：

```
// java.util.concurrent.locks.ReentrantLock#FairSync

static final class FairSync extends Sync {
  ...
      final void lock() {
              acquire(1);
      }
  ...
}
```

看到这块代码，我们可能会存在这种疑问：Lock 函数通过 Acquire 方法进行加

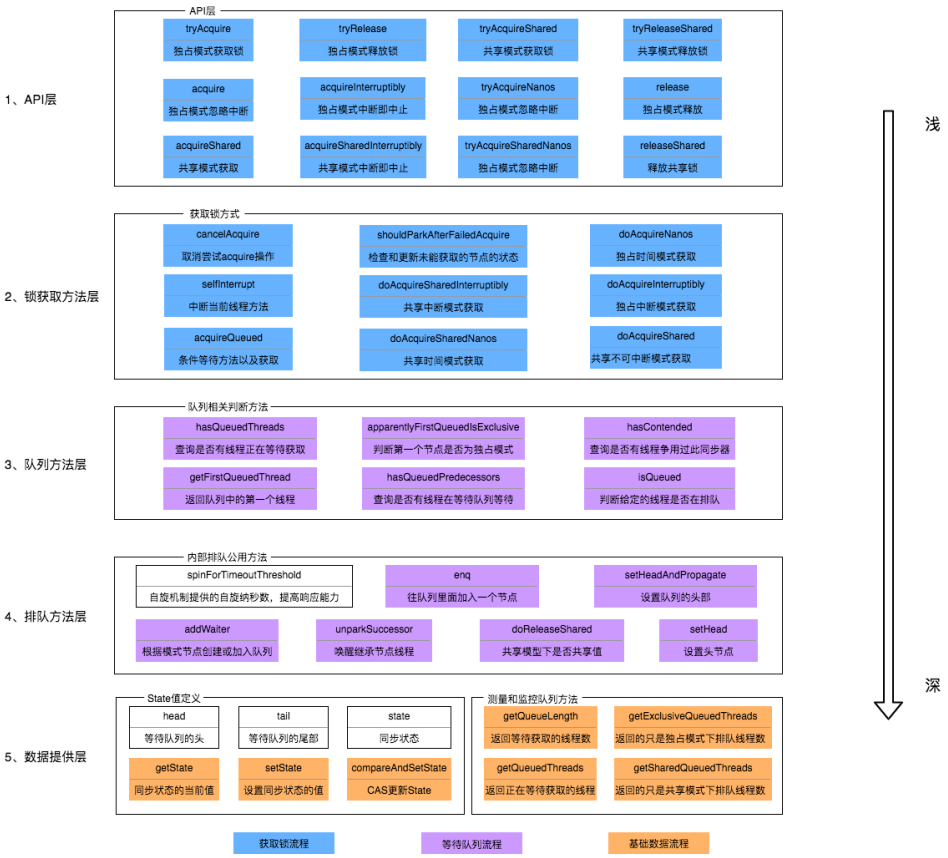
锁，但是具体是如何加锁的呢?

结合公平锁和非公平锁的加锁流程，虽然流程上有一定的不同，但是都调用了 Acquire 方法，而 Acquire 方法是 FairSync 和 UnfairSync 的父类 AQS 中的核心方法。

对于上边提到的问题，其实在 ReentrantLock 类源码中都无法解答，而这些问题的答案，都是位于 Acquire 方法所在的类 AbstractQueuedSynchronizer 中，也就是本文的核心——AQS。下面我们会对 AQS 以及 ReentrantLock 和 AQS 的关联做详细介绍（相关问题答案会在 2.3.5 小节中解答）。

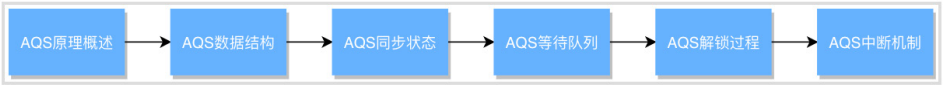
2. AQS

首先，我们通过下面的架构图来整体了解一下 AQS 框架：



- 上图中有颜色的为 Method，无颜色的为 Attribution。
- 总的来说，AQS 框架共分为五层，自上而下由浅入深，从 AQS 对外暴露的 API 到底层基础数据。
- 当有自定义同步器接入时，只需重写第一层所需要的部分方法即可，不需要关注底层具体的实现流程。当自定义同步器进行加锁或者解锁操作时，先经过第一层的 API 进入 AQS 内部方法，然后经过第二层进行锁的获取，接着对于获取锁失败的流程，进入第三层和第四层的等待队列处理，而这些处理方式均依赖于第五层的基础数据提供层。

下面我们会从整体到细节，从流程到方法逐一剖析 AQS 框架，主要分析过程如下：

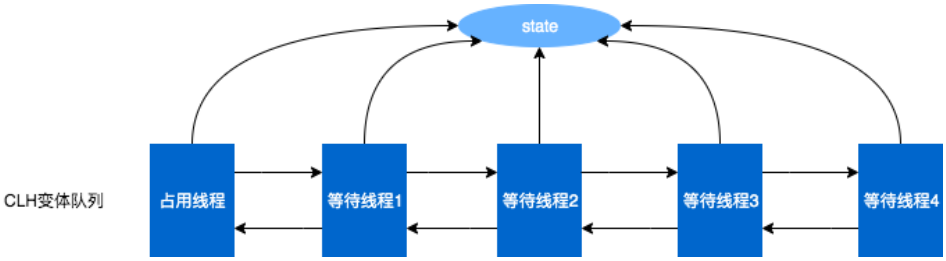


2.1 原理概览

AQS 核心思想是，如果被请求的共享资源空闲，那么就将当前请求资源的线程设置为有效的工作线程，将共享资源设置为锁定状态；如果共享资源被占用，就需要一定的阻塞等待唤醒机制来保证锁分配。这个机制主要用的是 CLH 队列的变体实现的，将暂时获取不到锁的线程加入到队列中。

CLH：Craig、Landin and Hagersten 队列，是单向链表，AQS 中的队列是 CLH 变体的虚拟双向队列（FIFO），AQS 是通过将每条请求共享资源的线程封装成一个节点来实现锁的分配。

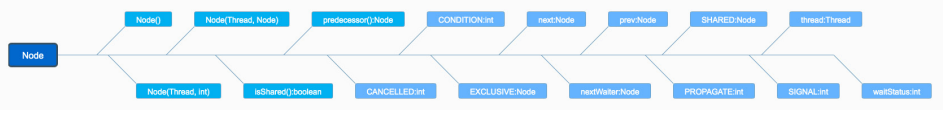
主要原理图如下：



AQS 使用一个 Volatile 的 int 类型的成员变量来表示同步状态，通过内置的 FIFO 队列来完成资源获取的排队工作，通过 CAS 完成对 State 值的修改。

2.1.1 AQS 数据结构

先来看下 AQS 中最基本的数据结构——Node，Node 即为上面 CLH 变体队列中的节点。



解释一下几个方法和属性值的含义：

方法和属性值	含义
waitStatus	当前节点在队列中的状态
thread	表示处于该节点的线程
prev	前驱指针
predecessor	返回前驱节点，没有的话抛出 npe
nextWaiter	指向下一个处于 CONDITION 状态的节点（由于本篇文章不讲述 Condition Queue 队列，这个指针不多介绍）
next	后继指针

线程两种锁的模式：

模式	含义
SHARED	表示线程以共享的模式等待锁
EXCLUSIVE	表示线程正在以独占的方式等待锁

waitStatus 有下面几个枚举值：

枚举	含义
0	当一个 Node 被初始化的时候的默认值
CANCELLED	为 1，表示线程获取锁的请求已经取消了
CONDITION	为 -2，表示节点在等待队列中，节点线程等待唤醒
PROPAGATE	为 -3，当前线程处在 SHARED 情况下，该字段才会使用
SIGNAL	为 -1，表示线程已经准备好了，就等资源释放了

2.1.2 同步状态 State

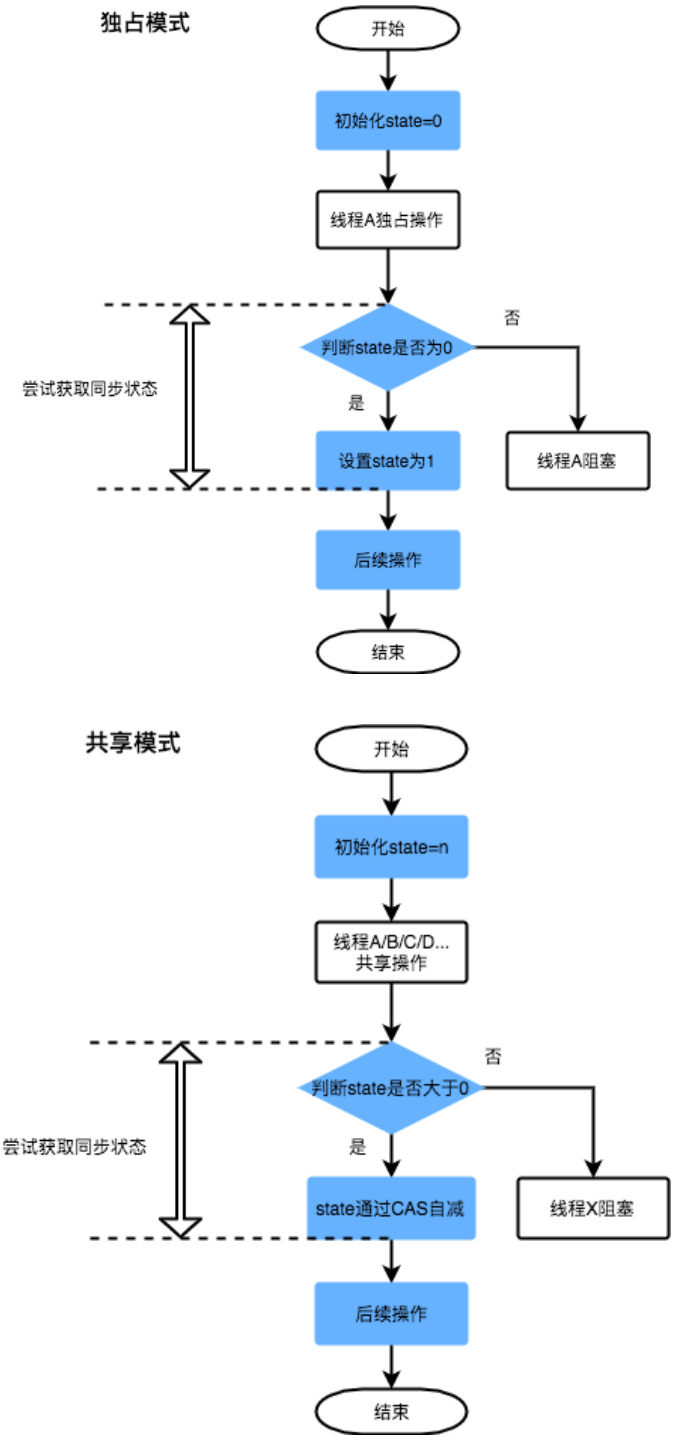
在了解数据结构后，接下来了解一下 AQS 的同步状态——State。AQS 中维护了一个名为 state 的字段，意为同步状态，是由 Volatile 修饰的，用于展示当前临界资源的获锁情况。

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

private volatile int state;
```

下面提供了几个访问这个字段的方法：

方法名	描述
protected final int getState()	获取 State 的值
protected final void setState(int newState)	设置 State 的值
protected final boolean compareAndSetState(int expect, int update)	使用 CAS 方式更新 State

这几个方法都是 Final 修饰的，说明子类中无法重写它们。我们可以通过修改 State 字段表示的同步状态来实现多线程的独占模式和共享模式（加锁过程）。


独占模式
开始
初始化state=0
线程A独占操作
判断state是否为0
否
是
尝试获取同步状态
设置state为1
线程A阻塞
后续操作
结束
共享模式
开始
初始化state=n
线程A/B/C/D...
共享操作
判断state是否大于0
否
是
尝试获取同步状态
state通过CAS自减
线程X阻塞
后续操作
结束

对于我们自定义的同步工具，需要自定义获取同步状态和释放状态的方式，也就是 AQS 架构图中的第一层：API 层。

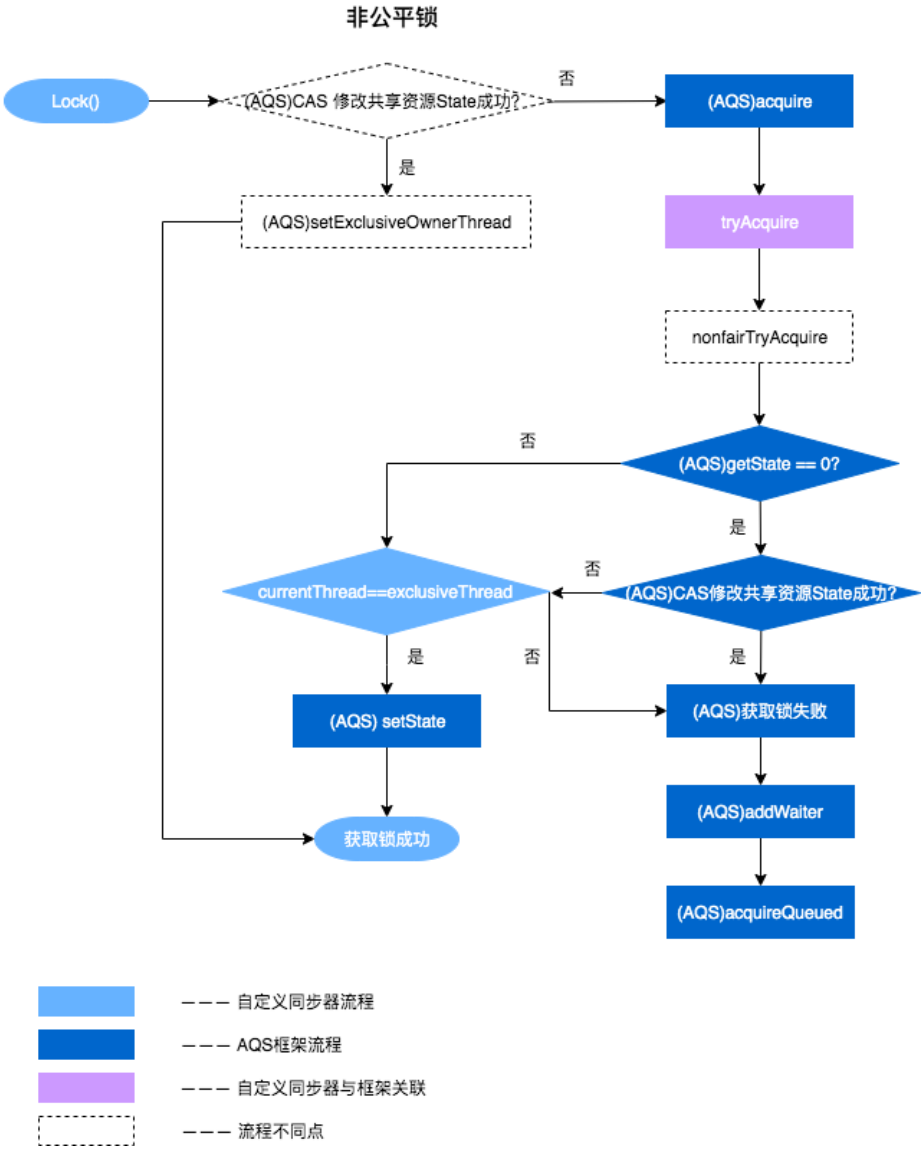
2.2 AQS 重要方法与 ReentrantLock 的关联

从架构图中可以得知，AQS 提供了大量用于自定义同步器实现的 Protected 方法。自定义同步器实现的相关方法也只是为了通过修改 State 字段来实现多线程的独占模式或者共享模式。自定义同步器需要实现以下方法（ReentrantLock 需要实现的方法如下，并不是全部）：

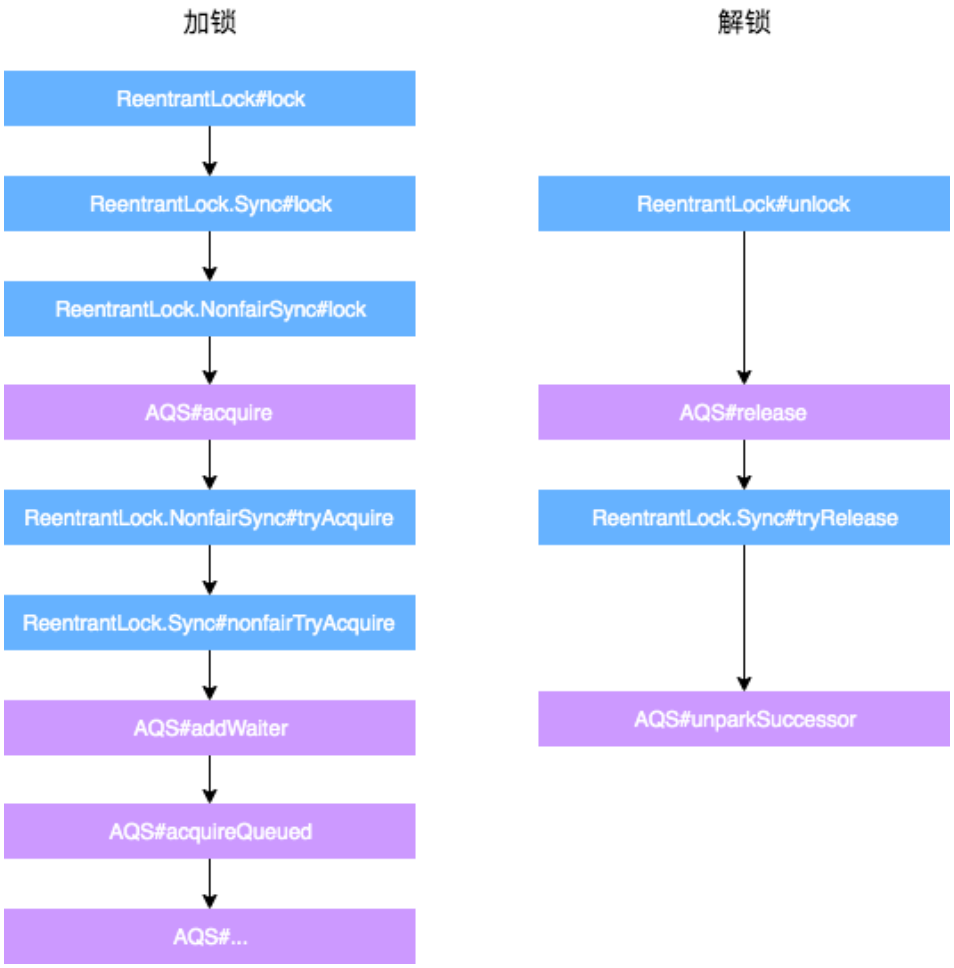
方法名	描述
protected boolean isHeldExclusively()	该线程是否正在独占资源。只有用到 Condition 才需要去实现它。
protected boolean tryAcquire(int arg)	独占方式。arg 为获取锁的次数，尝试获取资源，成功则返回 True，失败则返回 False。
protected boolean tryRelease(int arg)	独占方式。arg 为释放锁的次数，尝试释放资源，成功则返回 True，失败则返回 False。
protected int tryAcquireShared(int arg)	共享方式。arg 为获取锁的次数，尝试获取资源。负数表示失败；0 表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。
protected boolean tryReleaseShared(int arg)	共享方式。arg 为释放锁的次数，尝试释放资源，如果释放后允许唤醒后续等待结点返回 True，否则返回 False。

一般来说，自定义同步器要么是独占方式，要么是共享方式，它们也只需实现 tryAcquire-tryRelease、tryAcquireShared-tryReleaseShared 中的一种即可。AQS 也支持自定义同步器同时实现独占和共享两种方式，如 ReentrantReadWriteLock。ReentrantLock 是独占锁，所以实现了 tryAcquire-tryRelease。

以非公平锁为例，这里主要阐述一下非公平锁与 AQS 之间方法的关联之处，具体每一处核心方法的作用会在文章后面详细进行阐述。



为了帮助大家理解 ReentrantLock 和 AQS 之间方法的交互过程，以非公平锁为例，我们将加锁和解锁的交互流程单独拎出来强调一下，以便于对后续内容的理解。



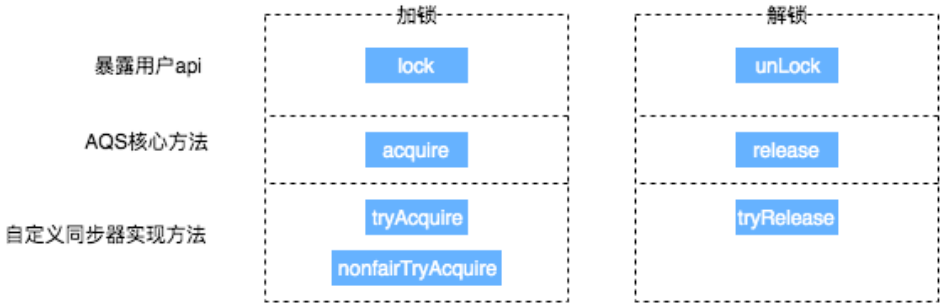
加锁：

- 通过 ReentrantLock 的加锁方法 Lock 进行加锁操作。
- 会调用到内部类 Sync 的 Lock 方法，由于 Sync#lock 是抽象方法，根据 ReentrantLock 初始化选择的公平锁和非公平锁，执行相关内部类的 Lock 方法，本质上都会执行 AQS 的 Acquire 方法。
- AQS 的 Acquire 方法会执行 tryAcquire 方法，但是由于 tryAcquire 需要自定义同步器实现，因此执行了 ReentrantLock 中的 tryAcquire 方法，由于 ReentrantLock 是通过公平锁和非公平锁内部类实现的 tryAcquire 方法，因

此会根据锁类型不同，执行不同的 tryAcquire。

- tryAcquire 是获取锁逻辑，获取失败后，会执行框架 AQS 的后续逻辑，跟 ReentrantLock 自定义同步器无关。
- 解锁：
- 通过 ReentrantLock 的解锁方法 Unlock 进行解锁。
- Unlock 会调用内部类 Sync 的 Release 方法，该方法继承于 AQS。
- Release 中会调用 tryRelease 方法，tryRelease 需要自定义同步器实现，tryRelease 只在 ReentrantLock 中的 Sync 实现，因此可以看出，释放锁的过程，并不区分是否为公平锁。
- 释放成功后，所有处理由 AQS 框架完成，与自定义同步器无关。

通过上面的描述，大概可以总结出 ReentrantLock 加锁解锁时 API 层核心方法的映射关系。



2.3 通过 ReentrantLock 理解 AQS

ReentrantLock 中公平锁和非公平锁在底层是相同的，这里以非公平锁为例进行分析。

在非公平锁中，有一段这样的代码：

```
// java.util.concurrent.locks.ReentrantLock

static final class NonfairSync extends Sync {
      ...
      final void lock() {
```

```
                if (compareAndSetState(0, 1))
                        setExclusiveOwnerThread(Thread.currentThread());
                else
                        acquire(1);
        }
    ...
}
```

看一下这个 Acquire 是怎么写的：

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

public final void acquire(int arg) {
        if (!tryAcquire(arg) && acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
                selfInterrupt();
}
```

再看一下 tryAcquire 方法：

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

protected boolean tryAcquire(int arg) {
        throw new UnsupportedOperationException();
}
```

可以看出，这里只是 AQS 的简单实现，具体获取锁的实现方法是由各自的公平锁和非公平锁单独实现的（以 ReentrantLock 为例）。如果该方法返回了 True，则说明当前线程获取锁成功，就不用往后执行了；如果获取失败，就需要加入到等待队列中。下面会详细解释线程是何时以及怎样被加入进等待队列中的。

2.3.1 线程加入等待队列

2.3.1.1 加入队列的时机

当执行 Acquire(1) 时，会通过 tryAcquire 获取锁。在这种情况下，如果获取锁失败，就会调用 addWaiter 加入到等待队列中去。

2.3.1.2 如何加入队列

获取锁失败后，会执行 addWaiter(Node.EXCLUSIVE) 加入等待队列，具体实现方法如下：

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

private Node addWaiter(Node mode) {
      Node node = new Node(Thread.currentThread(), mode);
      // Try the fast path of enq; backup to full enq on failure
      Node pred = tail;
      if (pred != null) {
              node.prev = pred;
              if (compareAndSetTail(pred, node)) {
                      pred.next = node;
                      return node;
              }
      }
      enq(node);
      return node;
}
private final boolean compareAndSetTail(Node expect, Node update) {
      return unsafe.compareAndSwapObject(this, tailOffset, expect, update);
}
```

主要的流程如下：

- 通过当前的线程和锁模式新建一个节点。
- Pred 指针指向尾节点 Tail。
- 将 New 中 Node 的 Prev 指针指向 Pred。
- 通过 compareAndSetTail 方法，完成尾节点的设置。这个方法主要是对 tailOffset 和 Expect 进行比较，如果 tailOffset 的 Node 和 Expect 的 Node 地址是相同的，那么设置 Tail 的值为 Update 的值。

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

static {
      try {
              stateOffset = unsafe.objectFieldOffset(AbstractQueuedSynchronizer.
class.getDeclaredField("state"));
              headOffset = unsafe.
objectFieldOffset(AbstractQueuedSynchronizer.class.
getDeclaredField("head"));
              tailOffset = unsafe.
objectFieldOffset(AbstractQueuedSynchronizer.class.
getDeclaredField("tail"));
              waitStatusOffset = unsafe.objectFieldOffset(Node.class.
getDeclaredField("waitStatus"));
```

```
                nextOffset = unsafe.objectFieldOffset(Node.class.
getDeclaredField("next"));
        } catch (Exception ex) {
      throw new Error(ex);
    }
}
```

从 AQS 的静态代码块可以看出，都是获取一个对象的属性相对于该对象在内存当中的偏移量，这样我们就可以根据这个偏移量在对象内存当中找到这个属性。tailOffset 指的是 tail 对应的偏移量，所以这个时候会将 new 出来的 Node 置为当前队列的尾节点。同时，由于是双向链表，也需要将前一个节点指向尾节点。

- 如果 Pred 指针是 Null（说明等待队列中没有元素），或者当前 Pred 指针和 Tail 指向的位置不同（说明被别的线程已经修改），就需要看一下 Enq 的方法。

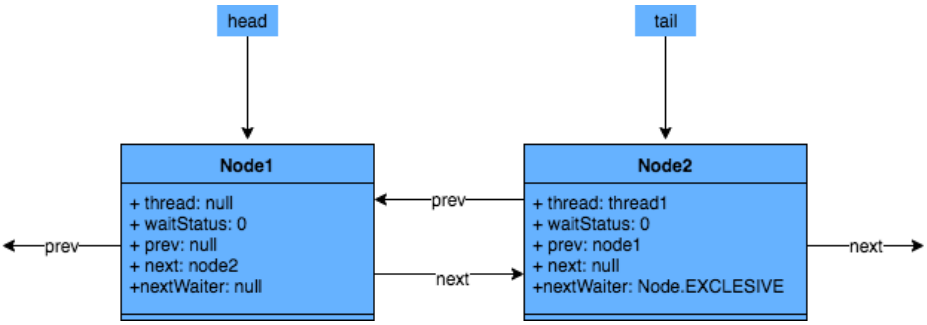
```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

private Node enq(final Node node) {
        for (;;) {
                Node t = tail;
                if (t == null) { // Must initialize
                        if (compareAndSetHead(new Node()))
                                tail = head;
                } else {
                        node.prev = t;
                        if (compareAndSetTail(t, node)) {
                                t.next = node;
                                return t;
                        }
                }
        }
}
```

如果没有被初始化，需要进行初始化一个头结点出来。但请注意，初始化的头结点并不是当前线程节点，而是调用了无参构造函数的节点。如果经历了初始化或者并发导致队列中有元素，则与之前的方法相同。其实，addWaiter 就是一个在双端链表添加尾节点的操作，需要注意的是，双端链表的头结点是一个无参构造函数的头结点。

总结一下，线程获取锁的时候，过程大体如下：

1. 当没有线程获取到锁时，线程 1 获取锁成功。

2. 线程 2 申请锁，但是锁被线程 1 占有。



3. 如果再有线程要获取锁，依次在队列中往后排队即可。

回到上边的代码，hasQueuedPredecessors 是公平锁加锁时判断等待队列中是否存在有效节点的方法。如果返回 False，说明当前线程可以争取共享资源；如果返回 True，说明队列中存在有效节点，当前线程必须加入到等待队列中。

```
// java.util.concurrent.locks.ReentrantLock

public final boolean hasQueuedPredecessors() {
      // The correctness of this depends on head being initialized
      // before tail and on head.next being accurate if the current
      // thread is first in queue.
      Node t = tail; // Read fields in reverse initialization order
      Node h = head;
      Node s;
      return h != t && ((s = h.next) == null || s.thread != Thread.
currentThread());
}
```

看到这里，我们理解一下 h != t && ((s = h.next) == null || s.thread != Thread.currentThread()); 为什么要判断的头结点的下一个节点？第一个节点储存的数据是什么？

双向链表中，第一个节点为虚节点，其实并不存储任何信息，只是占位。真正的第一个有数据的节点，是在第二个节点开始的。当 h != t 时：如果 (s =

h.next) == null，等待队列正在有线程进行初始化，但只是进行到了 Tail 指向 Head，没有将 Head 指向 Tail，此时队列中有元素，需要返回 True（这块具体见下边代码分析）。 如果 (s = h.next) != null，说明此时队列中至少有一个有效节点。如果此时 s.thread == Thread.currentThread()，说明等待队列的第一个有效节点中的线程与当前线程相同，那么当前线程是可以获取资源的；如果 s.thread != Thread.currentThread()，说明等待队列的第一个有效节点线程与当前线程不同，当前线程必须加入进等待队列。

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer#enq

if (t == null) { // Must initialize
      if (compareAndSetHead(new Node()))
              tail = head;
} else {
      node.prev = t;
      if (compareAndSetTail(t, node)) {
              t.next = node;
              return t;
      }
}
```

节点入队不是原子操作，所以会出现短暂的 head != tail，此时 Tail 指向最后一个节点，而且 Tail 指向 Head。如果 Head 没有指向 Tail（可见 5、6、7 行），这种情况下也需要将相关线程加入队列中。所以这块代码是为了解决极端情况下的并发问题。

2.3.1.3 等待队列中线程出队列时机

回到最初的源码：

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

public final void acquire(int arg) {
      if (!tryAcquire(arg) && acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
              selfInterrupt();
}
```

上文解释了 addWaiter 方法，这个方法其实就是把对应的线程以 Node 的数据结构形式加入到双端队列里，返回的是一个包含该线程的 Node。而这个 Node 会作

为参数，进入到 acquireQueued 方法中。acquireQueued 方法可以对排队中的线程进行“获锁”操作。

总的来说，一个线程获取锁失败了，被放入等待队列，acquireQueued 会把放入队列中的线程不断去获取锁，直到获取成功或者不再需要获取（中断）。

下面我们从“何时出队列?”和“如何出队列?”两个方向来分析一下 acquireQueued 源码：

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

final boolean acquireQueued(final Node node, int arg) {
      // 标记是否成功拿到资源
      boolean failed = true;
      try {
              // 标记等待过程中是否中断过
              boolean interrupted = false;
              // 开始自旋，要么获取锁，要么中断
              for (;;) {
                      // 获取当前节点的前驱节点
                      final Node p = node.predecessor();
                      // 如果 p 是头结点，说明当前节点在真实数据队列的首部，就尝试
获取锁（别忘了头结点是虚节点）
                      if (p == head && tryAcquire(arg)) {
                              // 获取锁成功，头指针移动到当前 node
                              setHead(node);
                              p.next = null; // help GC
                              failed = false;
                              return interrupted;
                      }
                      // 说明 p 为头节点且当前没有获取到锁（可能是非公平锁被抢占了）
或者是 p 不为头结点，这个时候就要判断当前 node 是否要被阻塞（被阻塞条件：前驱节点的
waitStatus 为 -1），防止无限循环浪费资源。具体两个方法下面细细分析
                      if (shouldParkAfterFailedAcquire(p, node) &&
parkAndCheckInterrupt())
                              interrupted = true;
              }
      } finally {
              if (failed)
                      cancelAcquire(node);
      }
}
```

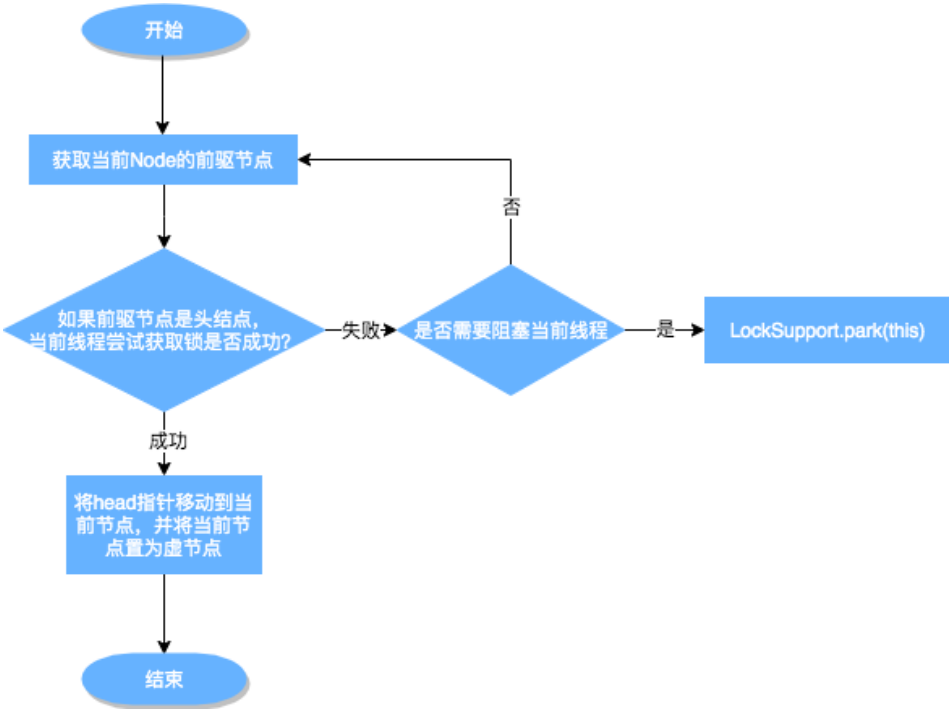
注：setHead 方法是把当前节点置为虚节点，但并没有修改 waitStatus，因为它是一直需要用的数据。

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

private void setHead(Node node) {
      head = node;
      node.thread = null;
      node.prev = null;
}

// java.util.concurrent.locks.AbstractQueuedSynchronizer

// 靠前驱节点判断当前线程是否应该被阻塞
private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {
      // 获取头结点的节点状态
      int ws = pred.waitStatus;
      // 说明头结点处于唤醒状态
      if (ws == Node.SIGNAL)
              return true;
      // 通过枚举值我们知道 waitStatus>0 是取消状态
      if (ws > 0) {
              do {
                      // 循环向前查找取消节点，把取消节点从队列中剔除
                      node.prev = pred = pred.prev;
              } while (pred.waitStatus > 0);
              pred.next = node;
      } else {
              // 设置前任节点等待状态为 SIGNAL
              compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
      }
      return false;
}
```

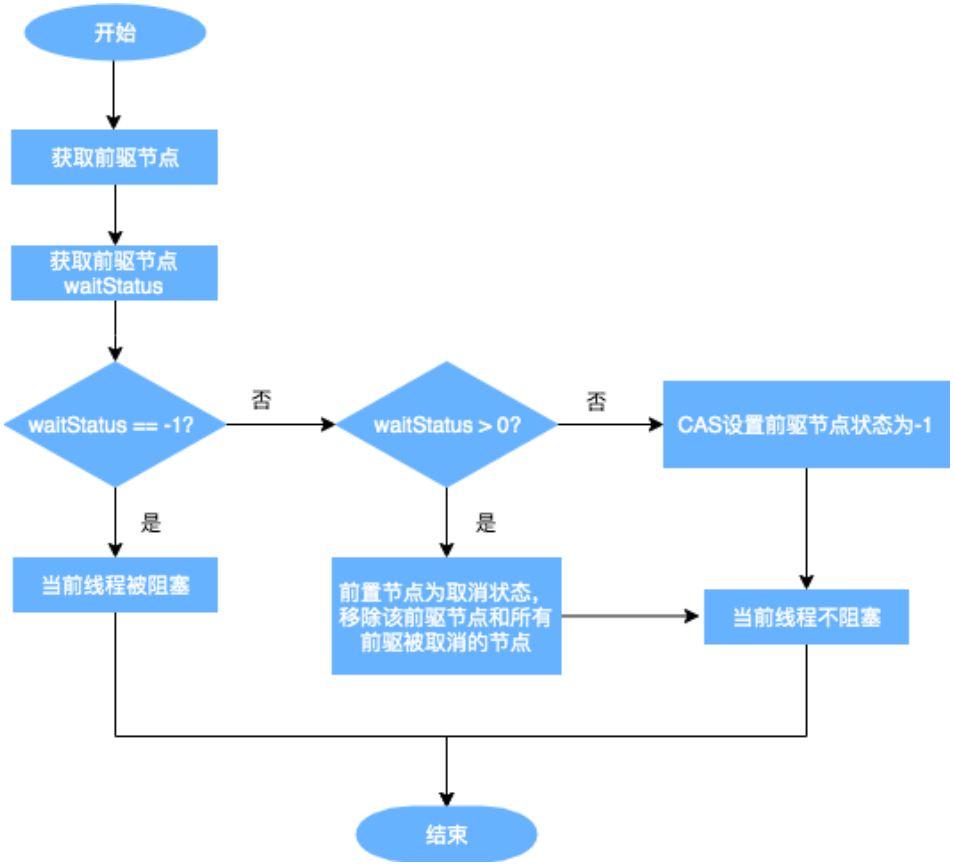
parkAndCheckInterrupt 主要用于挂起当前线程，阻塞调用栈，返回当前线程的中断状态。

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

private final boolean parkAndCheckInterrupt() {
    LockSupport.park(this);
    return Thread.interrupted();
}
```

上述方法的流程图如下：



从上图可以看出，跳出当前循环的条件是当“前置节点是头结点，且当前线程获取锁成功”。为了防止因死循环导致 CPU 资源被浪费，我们会判断前置节点的状态来决定是否要将当前线程挂起，具体挂起流程用流程图表示如下（shouldParkAfter-FailedAcquire 流程）：



从队列中释放节点的疑虑打消了，那么又有新问题了：

- shouldParkAfterFailedAcquire 中取消节点是怎么生成的呢？什么时候会把一个节点的 waitStatus 设置为 −1？
- 是在什么时间释放节点通知到被挂起的线程呢?

2.3.2 CANCELLED 状态节点生成

acquireQueued 方法中的 Finally 代码：

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

final boolean acquireQueued(final Node node, int arg) {
      boolean failed = true;
      try {
```

```
    ...
            for (;;) {
                    final Node p = node.predecessor();
                    if (p == head && tryAcquire(arg)) {
                            ...
                            failed = false;
        ...
                    }
                    ...
      } finally {
            if (failed)
                    cancelAcquire(node);
            }
}
```

通过 cancelAcquire 方法，将 Node 的状态标记为 CANCELLED。接下来，我们逐行来分析这个方法的原理：

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

private void cancelAcquire(Node node) {
  // 将无效节点过滤
      if (node == null)
              return;
  // 设置该节点不关联任何线程，也就是虚节点
      node.thread = null;
      Node pred = node.prev;
  // 通过前驱节点，跳过取消状态的 node
      while (pred.waitStatus > 0)
              node.prev = pred = pred.prev;
  // 获取过滤后的前驱节点的后继节点
      Node predNext = pred.next;
  // 把当前 node 的状态设置为 CANCELLED
      node.waitStatus = Node.CANCELLED;
  // 如果当前节点是尾节点，将从后往前的第一个非取消状态的节点设置为尾节点
  // 更新失败的话，则进入 else，如果更新成功，将 tail 的后继节点设置为 null
      if (node == tail && compareAndSetTail(node, pred)) {
              compareAndSetNext(pred, predNext, null);
      } else {
              int ws;
    // 如果当前节点不是 head 的后继节点，1：判断当前节点前驱节点的是否为 SIGNAL，
2：如果不是，则把前驱节点设置为 SINGAL 看是否成功
    // 如果 1 和 2 中有一个为 true，再判断当前节点的线程是否为 null
    // 如果上述条件都满足，把当前节点的前驱节点的后继指针指向当前节点的后继节点
              if (pred != head && ((ws = pred.waitStatus) == Node.SIGNAL
|| (ws <= 0 &&
compareAndSetWaitStatus(pred, ws, Node.SIGNAL))) && pred.thread != null)
```

```
{
                        Node next = node.next;
                        if (next != null && next.waitStatus <= 0)
                                compareAndSetNext(pred, predNext, next);
                } else {
        // 如果当前节点是 head 的后继节点，或者上述条件不满足，那就唤醒当前节点的后
继节点
                        unparkSuccessor(node);
                }
                node.next = node; // help GC
        }
}
```

当前的流程：

- 获取当前节点的前驱节点，如果前驱节点的状态是 CANCELLED，那就一直往前遍历，找到第一个 waitStatus <= 0 的节点，将找到的 Pred 节点和当前 Node 关联，将当前 Node 设置为 CANCELLED。

根据当前节点的位置，考虑以下三种情况：

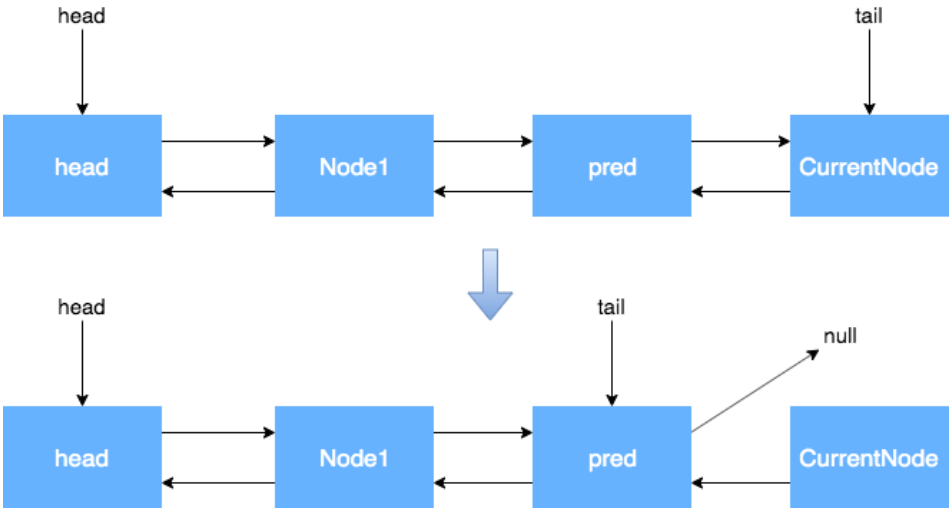
（1）当前节点是尾节点。

（2）当前节点是 Head 的后继节点。

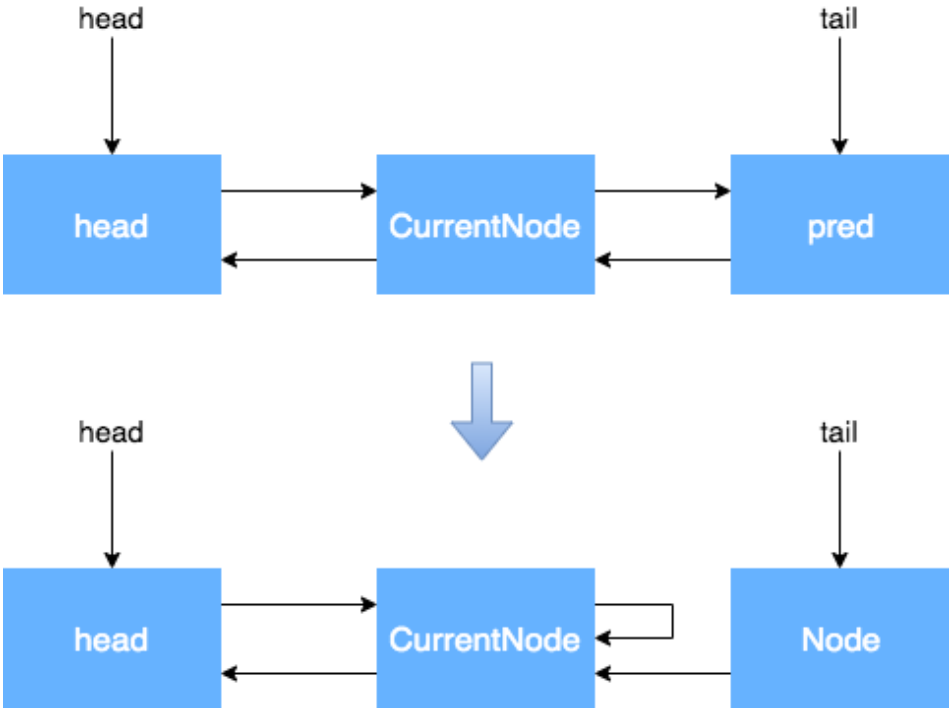
（3）当前节点不是 Head 的后继节点，也不是尾节点。

根据上述第二条，我们来分析每一种情况的流程。

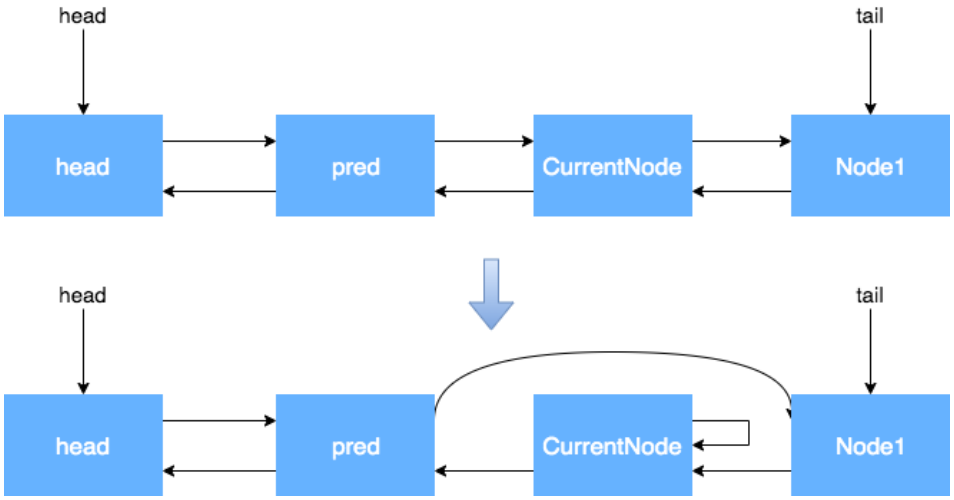
当前节点是尾节点。



当前节点是 Head 的后继节点。



当前节点不是 Head 的后继节点，也不是尾节点。



通过上面的流程，我们对于 CANCELLED 节点状态的产生和变化已经有了大致的了解，但是为什么所有的变化都是对 Next 指针进行了操作，而没有对 Prev 指针进行操作呢？什么情况下会对 Prev 指针进行操作？

- 执行 cancelAcquire 的时候，当前节点的前置节点可能已经从队列中出去了（已经执行过 Try 代码块中的 shouldParkAfterFailedAcquire 方法了），如果此时修改 Prev 指针，有可能会导致 Prev 指向另一个已经移除队列的 Node，因此这块变化 Prev 指针不安全。 shouldParkAfterFailedAcquire 方法中，会执行下面的代码，其实就是在处理 Prev 指针。shouldParkAfterFailedAcquire 是获取锁失败的情况下才会执行，进入该方法后，说明共享资源已被获取，当前节点之前的节点都不会出现变化，因此这个时候变更 Prev 指针比较安全。

```
do {
      node.prev = pred = pred.prev;
} while (pred.waitStatus > 0);
```

2.3.3 如何解锁

我们已经剖析了加锁过程中的基本流程，接下来再对解锁的基本流程进行分析。

由于 ReentrantLock 在解锁的时候，并不区分公平锁和非公平锁，所以我们直接看解锁的源码：

```
// java.util.concurrent.locks.ReentrantLock

public void unlock() {
      sync.release(1);
}
```

可以看到，本质释放锁的地方，是通过框架来完成的。

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

public final boolean release(int arg) {
      if (tryRelease(arg)) {
            Node h = head;
            if (h != null && h.waitStatus != 0)
                  unparkSuccessor(h);
            return true;
      }
      return false;
}
```

在 ReentrantLock 里面的公平锁和非公平锁的父类 Sync 定义了可重入锁的释放锁机制。

```
// java.util.concurrent.locks.ReentrantLock.Sync

// 方法返回当前锁是不是没有被线程持有
protected final boolean tryRelease(int releases) {
      // 减少可重入次数
      int c = getState() - releases;
      // 当前线程不是持有锁的线程，抛出异常
      if (Thread.currentThread() != getExclusiveOwnerThread())
            throw new IllegalMonitorStateException();
      boolean free = false;
      // 如果持有线程全部释放，将当前独占锁所有线程设置为 null，并更新 state
      if (c == 0) {
            free = true;
            setExclusiveOwnerThread(null);
      }
      setState(c);
      return free;
}
```

我们来解释下述源码：

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

public final boolean release(int arg) {
      // 上边自定义的 tryRelease 如果返回 true，说明该锁没有被任何线程持有
      if (tryRelease(arg)) {
              // 获取头结点
              Node h = head;
              // 头结点不为空并且头结点的 waitStatus 不是初始化节点情况，解除线
程挂起状态
              if (h != null && h.waitStatus != 0)
                      unparkSuccessor(h);
              return true;
      }
      return false;
}
```

这里的判断条件为什么是 h != null && h.waitStatus != 0？

h == null Head 还没初始化。初始情况下，head == null，第一个节点入队，Head 会被初始化一个虚拟节点。所以说，这里如果还没来得及入队，就会出现 head == null 的情况。

h != null && waitStatus == 0 表明后继节点对应的线程仍在运行中，不需要唤醒。

h != null && waitStatus < 0 表明后继节点可能被阻塞了，需要唤醒。

再看一下 unparkSuccessor 方法：

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

private void unparkSuccessor(Node node) {
      // 获取头结点 waitStatus
      int ws = node.waitStatus;
      if (ws < 0)
              compareAndSetWaitStatus(node, ws, 0);
      // 获取当前节点的下一个节点
      Node s = node.next;
      // 如果下个节点是 null 或者下个节点被 cancelled，就找到队列最开始的非
cancelled 的节点
      if (s == null || s.waitStatus > 0) {
              s = null;
              // 就从尾部节点开始找，到队首，找到队列第一个 waitStatus<0 的节点。
```

```
                for (Node t = tail; t != null && t != node; t = t.prev)
                        if (t.waitStatus <= 0)
                                s = t;
        }
        // 如果当前节点的下个节点不为空，而且状态 <=0，就把当前节点 unpark
        if (s != null)
                LockSupport.unpark(s.thread);
}
```

为什么要从后往前找第一个非 Cancelled 的节点呢？原因如下。

之前的 addWaiter 方法：

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

private Node addWaiter(Node mode) {
        Node node = new Node(Thread.currentThread(), mode);
        // Try the fast path of enq; backup to full enq on failure
        Node pred = tail;
        if (pred != null) {
                node.prev = pred;
                if (compareAndSetTail(pred, node)) {
                        pred.next = node;
                        return node;
                }
        }
        enq(node);
        return node;
}
```

我们从这里可以看到，节点入队并不是原子操作，也就是说，node.prev = pred; compareAndSetTail(pred, node) 这两个地方可以看作 Tail 入队的原子操作，但是此时 pred.next = node; 还没执行，如果这个时候执行了 unparkSuccessor 方法，就没办法从前往后找了，所以需要从后往前找。还有一点原因，在产生 CANCELLED 状态节点的时候，先断开的是 Next 指针，Prev 指针并未断开，因此也是必须要从后往前遍历才能够遍历完全部的 Node。

综上所述，如果是从前往后找，由于极端情况下入队的非原子操作和 CANCELLED 节点产生过程中断开 Next 指针的操作，可能会导致无法遍历所有的节点。所以，唤醒对应的线程后，对应的线程就会继续往下执行。继续执行

acquireQueued 方法以后，中断如何处理?

2.3.4 中断恢复后的执行流程

唤醒后，会执行 return Thread.interrupted();，这个函数返回的是当前执行线程的中断状态，并清除。

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

private final boolean parkAndCheckInterrupt() {
      LockSupport.park(this);
      return Thread.interrupted();
}
```

再回到 acquireQueued 代码，当 parkAndCheckInterrupt 返回 True 或者 False 的时候，interrupted 的值不同，但都会执行下次循环。如果这个时候获取锁成功，就会把当前 interrupted 返回。

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

final boolean acquireQueued(final Node node, int arg) {
      boolean failed = true;
      try {
              boolean interrupted = false;
              for (;;) {
                      final Node p = node.predecessor();
                      if (p == head && tryAcquire(arg)) {
                              setHead(node);
                              p.next = null; // help GC
                              failed = false;
                              return interrupted;
                      }
                      if (shouldParkAfterFailedAcquire(p, node) &&
parkAndCheckInterrupt())
                              interrupted = true;
                      }
      } finally {
              if (failed)
                      cancelAcquire(node);
      }
}
```

如果 acquireQueued 为 True，就会执行 selfInterrupt 方法。

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

static void selfInterrupt() {
      Thread.currentThread().interrupt();
}
```

该方法其实是为了中断线程。但为什么获取了锁以后还要中断线程呢？这部分属于 Java 提供的协作式中断知识内容，感兴趣同学可以查阅一下。这里简单介绍一下：

1. 当中断线程被唤醒时，并不知道被唤醒的原因，可能是当前线程在等待中被中断，也可能是释放了锁以后被唤醒。因此我们通过 Thread.interrupted() 方法检查中断标记（该方法返回了当前线程的中断状态，并将当前线程的中断标识设置为 False），并记录下来，如果发现该线程被中断过，就再中断一次。
2. 线程在等待资源的过程中被唤醒，唤醒后还是会不断地去尝试获取锁，直到抢到锁为止。也就是说，在整个流程中，并不响应中断，只是记录中断记录。最后抢到锁返回了，那么如果被中断过的话，就需要补充一次中断。

这里的处理方式主要是运用线程池中基本运作单元 Worder 中的 runWorker，通过 Thread.interrupted() 进行额外的判断处理，感兴趣的同学可以看下 ThreadPoolExecutor 源码。

2.3.5 小结

我们在 1.3 小节中提出了一些问题，现在来回答一下。

Q：某个线程获取锁失败的后续流程是什么呢?

A：存在某种排队等候机制，线程继续等待，仍然保留获取锁的可能，获取锁流程仍在继续。

Q：既然说到了排队等候机制，那么就一定会有某种队列形成，这样的队列是什么数据结构呢?

A：是 CLH 变体的 FIFO 双端队列。

Q：处于排队等候机制中的线程，什么时候可以有机会获取锁呢?

A：可以详细看下 2.3.1.3 小节。

Q：如果处于排队等候机制中的线程一直无法获取锁，需要一直等待么？还是有别的策略来解决这一问题?

A：线程所在节点的状态会变成取消状态，取消状态的节点会从队列中释放，具体可见 2.3.2 小节。

Q：Lock 函数通过 Acquire 方法进行加锁，但是具体是如何加锁的呢?

A：AQS 的 Acquire 会调用 tryAcquire 方法，tryAcquire 由各个自定义同步器实现，通过 tryAcquire 完成加锁过程。

3. AQS 应用

3.1 ReentrantLock 的可重入应用

ReentrantLock 的可重入性是 AQS 很好的应用之一，在了解完上述知识点以后，我们很容易得知 ReentrantLock 实现可重入的方法。在 ReentrantLock 里面，不管是公平锁还是非公平锁，都有一段逻辑。

公平锁：

```
// java.util.concurrent.locks.ReentrantLock.FairSync#tryAcquire

if (c == 0) {
      if (!hasQueuedPredecessors() && compareAndSetState(0, acquires)) {
              setExclusiveOwnerThread(current);
              return true;
      }
}
else if (current == getExclusiveOwnerThread()) {
      int nextc = c + acquires;
      if (nextc < 0)
              throw new Error("Maximum lock count exceeded");
      setState(nextc);
      return true;
}
```

非公平锁：

```
// java.util.concurrent.locks.ReentrantLock.Sync#nonfairTryAcquire

if (c == 0) {
      if (compareAndSetState(0, acquires)){
```

```
                setExclusiveOwnerThread(current);
                return true;
        }
}
else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0) // overflow
                throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
}
```

从上面这两段都可以看到，有一个同步状态 State 来控制整体可重入的情况。State 是 Volatile 修饰的，用于保证一定的可见性和有序性。

```
// java.util.concurrent.locks.AbstractQueuedSynchronizer

private volatile int state;
```

接下来看 State 这个字段主要的过程：

1. State 初始化的时候为 0，表示没有任何线程持有锁。
2. 当有线程持有该锁时，值就会在原来的基础上 +1，同一个线程多次获得锁是，就会多次 +1，这里就是可重入的概念。
3. 解锁也是对这个字段 −1，一直到 0，此线程对锁释放。

3.2 JUC 中的应用场景

除了上边 ReentrantLock 的可重入性的应用，AQS 作为并发编程的框架，为很多其他同步工具提供了良好的解决方案。下面列出了 JUC 中的几种同步工具，大体介绍一下 AQS 的应用场景：

同步工具	同步工具与AQS的关联
ReentrantLock	使用 AQS 保存锁重复持有的次数。当一个线程获取锁时，ReentrantLock 记录当前获得锁的线程标识，用于检测是否重复获取，以及错误线程试图解锁操作时异常情况的处理。
Semaphore	使用 AQS 同步状态来保存信号量的当前计数。tryRelease 会增加计数，acquireShared 会减少计数。
CountDownLatch	使用 AQS 同步状态来表示计数。计数为 0 时，所有的 Acquire 操作（CountDownLatch 的 await 方法）才可以通过。

同步工具	同步工具与AQS的关联
ReentrantRead-WriteLock	使用 AQS 同步状态中的 16 位保存写锁持有的次数，剩下的 16 位用于保存读锁的持有次数。
ThreadPoolExec-utor	Worker 利用 AQS 同步状态实现对独占线程变量的设置（tryAcquire 和 tryRelease）。

3.3 自定义同步工具

了解 AQS 基本原理以后，按照上面所说的 AQS 知识点，自己实现一个同步工具。

```
public class LeeLock  {

    private static class Sync extends AbstractQueuedSynchronizer {
        @Override
        protected boolean tryAcquire (int arg) {
            return compareAndSetState(0, 1);
        }

        @Override
        protected boolean tryRelease (int arg) {
            setState(0);
            return true;
        }

        @Override
        protected boolean isHeldExclusively () {
            return getState() == 1;
        }
    }

    private Sync sync = new Sync();

    public void lock () {
        sync.acquire(1);
    }

    public void unlock () {
        sync.release(1);
    }
}
```

通过我们自己定义的 Lock 完成一定的同步功能。

```
public class LeeMain {

    static int count = 0;
    static LeeLock leeLock = new LeeLock();

    public static void main (String[] args) throws InterruptedException {

        Runnable runnable = new Runnable() {
            @Override
            public void run () {
                try {
                    leeLock.lock();
                    for (int i = 0; i < 10000; i++) {
                        count++;
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                } finally {
                    leeLock.unlock();
                }

            }
        };
        Thread thread1 = new Thread(runnable);
        Thread thread2 = new Thread(runnable);
        thread1.start();
        thread2.start();
        thread1.join();
        thread2.join();
        System.out.println(count);
    }
}
```

上述代码每次运行结果都会是 20000。通过简单的几行代码就能实现同步功能，这就是 AQS 的强大之处。

总结

我们日常开发中使用并发的场景太多，但是对并发内部的基本框架原理了解的人却不多。由于篇幅原因，本文仅介绍了可重入锁 ReentrantLock 的原理和 AQS 原理，希望能够成为大家了解 AQS 和 ReentrantLock 等同步器的“敲门砖”。

参考资料

- Lea D. The java. util. concurrent synchronizer framework[J]. Science of Computer Programming, 2005, 58(3): 293-309.
- 《Java 并发编程实战》
- 不可不说的 Java“锁”事

作者简介

李卓，美团点评住宿度假研发中心 Java 研发工程师，2018 年加入美团点评。

招聘信息

美团到店住宿门票业务研发团队负责美团酒店和门票核心业务系统建设。

- 美团酒店屡次刷新行业记录，最近 12 个月酒店预订间夜量达到 3 个亿，单日入住间夜量峰值突破 300 万，单季度间夜突破 1 亿间。
- 美团门票 2018 年出票量达到一亿张，成为国内门票预订规模顶尖的平台。技术团队的愿景是：建设打造旅游住宿行业一流的技术架构，从质量、安全、效率、性能多角度保障系统高速发展。

美团到店住宿门票业务研发团队期待优秀的技术伙伴加入，欢迎投简历至：tech@meituan.com（邮件标题注明：美团到店住宿门票业务研发团队）

架构

美团点评 Kubernetes 集群管理实践

国梁

背景

作为国内领先的生活服务平台，美团点评很多业务都具有非常显著、规律的“高峰”和“低谷”特征。尤其遇到节假日或促销活动，流量还会在短时间内出现爆发式的增长。这对集群中心的资源弹性和可用性有非常高的要求，同时也会使系统在支撑业务流量时的复杂度和成本支出呈现指数级增长。而我们需要做的，就是利用有限的资源最大化地提升集群的吞吐能力，以保障用户体验。

本文将介绍美团点评 Kubernetes 集群管理与使用实践，包括美团点评集群管理与调度系统介绍、Kubernetes 管理与实践、Kubernetes 优化与改造以及资源管理与优化等。

美团点评集群管理与调度系统

美团点评在集群管理和资源优化这条道路上已经“摸爬滚打”多年。2013 年，开始构建基于传统虚拟化技术的资源交付方式；2015 年 7 月，开始建立完善的集群管理与调度系统——HULK，目标是推动美团点评服务容器化；2016 年，完成基于 Docker 容器技术自研实现了弹性伸缩能力，来提升交付速度和应对快速扩缩容的需求，实现弹性扩容、缩容，提升资源利用率，提升业务运维效率，合理有效的降低企业 IT 运维成本；2018 年，开始基于 Kubernetes 来进行资源管理和调度，进一步提升资源的使用效率。

美团点评集群管理与调度平台演进

最初，美团点评通过基于 Docker 容器技术自研实现了弹性伸缩能力，主要是为了解决基于虚拟化技术的管理及部署机制在应对服务快速扩容、缩容需求时存在的诸多不足。例如资源实例创建慢、无法统一运行环境、实例部署和交付流程长、资源回收效率低、弹性能力差等等。经过调研与测试，结合业界的实践经验，我们决定基于 Docker 容器技术自研集群管理与调度系统，有效应对快速扩缩容的需求，提升资源的利用效率。我们把它叫做“绿巨人”——HULK，这个阶段可以看作是 HULK1.0。

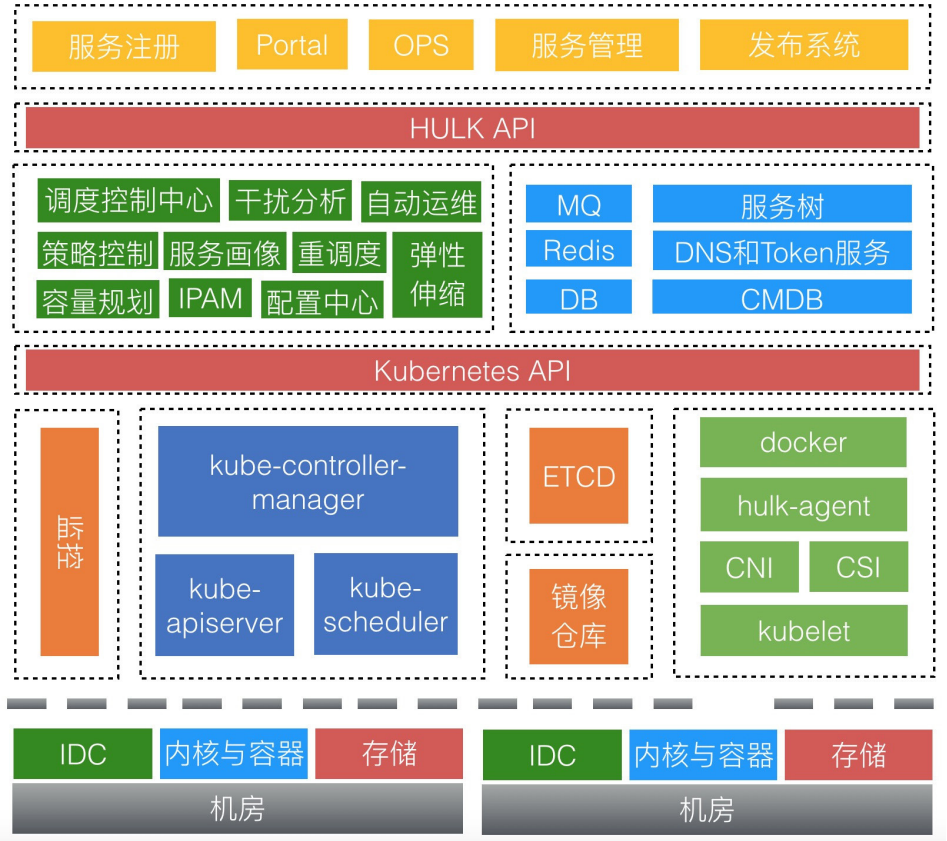
之后，在生产环境中经过不断摸索和尝试，我们逐渐意识到，仅仅满足于集群的弹性伸缩能力是不够的，成本和效率肯定是未来必将面临且更为棘手的问题。我们吸取了 2 年来 HULK 1.0 的开发和运维经验，在架构和支撑系统层面做了进一步优化和改进，并借助于生态和开源的力量来为 HULK 赋能，即引入了开源的集群管理与调度系统 Kubernetes，期望能进一步提升集群管理、运行的效率和稳定性，同时降低资源成本。所以我们从自研平台转向了开源的 Kubernetes 系统，并基于 Kubernetes 系统打造了更加智能化的集群管理与调度系统——HULK2.0。

架构全览

在架构层面，HULK2.0 如何能与上层业务和底层 Kubernetes 平台更好地分层和解耦，是我们在设计之初就优先考虑的问题。我们期望它既要能对业务使用友好，又能最大限度地发挥 Kubernetes 的调度能力，使得业务层和使用方毋需关注资源关系细节，所求即所得；同时使发布、配置、计费、负载等逻辑层与底层的 Kubernetes 平台解耦分层，并保持兼容原生 Kubernetes API 来访问 Kubernetes

集群。从而可以借助于统一的、主流的、符合业界规范的标准，来解决美团点评基础架构面临的复杂的、多样的、不统一的管理需求。

架构介绍



HULK2.0 架构图

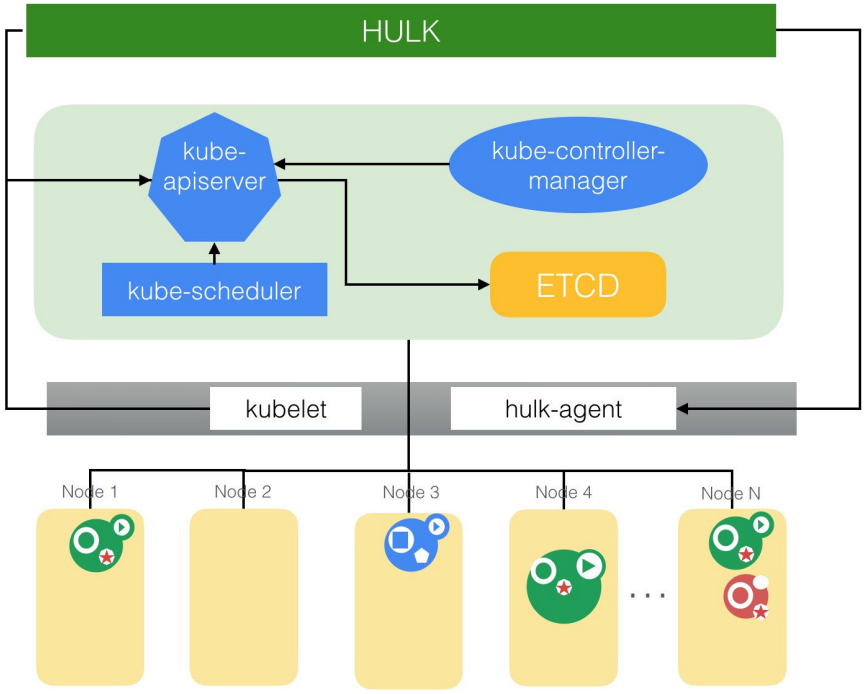
自上而下来看，美团集群管理与调度平台面向全公司服务，有各个主要业务线、统一的 OPS 平台以及 Portal 平台，HULK 不可能针对每个平台定制化接口和解决方案，所以需要将多样的业务和需求抽象收敛，最终统一通过 HULK API 来屏蔽 HULK 系统的细节，做到 HULK 与上层业务方的解耦。HULK API 是对业务层和资源需求的抽象，是外界访问 HULK 的唯一途径。

解决了上层的问题后，我们再来看与下层 Kubernetes 平台的解耦。HULK 接到上层资源请求后，首先要进行一系列的初始化工作，包括参数校验、资源余量、IP 和 Hostname 的分配等等，之后向 Kubernetes 平台实际申请分配机器资源，最终将资源交付给用户，Kubernetes API 进一步将资源需求收敛和转换，让我们可以借助于 Kubernetes 的资源管理优势。Kubernetes API 旨在收敛 HULK 的资源管理逻辑并与业界主流对齐。此外，因为完全兼容 Kubernetes API，可以让我们借助社区和生态的力量，共同建设和探索。

可以看到，HULK API 和 Kubernetes API 将我们整个系统分为三层，这样可以让每一层都专注于各自的模块。

Kubernetes 管理与实践

为什么会选择 Kubernetes 呢？ Kubernetes 并不是市面上唯一的集群管理平台（其他如 Docker Swarm 或 Mesos），之所以选择它，除了它本身优秀的架构设计，我们更加看重的是 Kubernetes 提供的不是一个解决方案，而是一个平台和一种能力。这种能力能够让我们真正基于美团点评的实际情况来扩展，同时能够依赖和复用多年来的技术积累，给予我们更多选择的自由，包括我们可以快速地部署应用程序，而无须面对传统平台所具有的风险，动态地扩展应用程序以及更好的资源分配策略。



HULK–Kubernetes 架构图

Kubernetes 集群作为整个 HULK 集群资源管理与平台的基础，需求是稳定性和可扩展性，风险可控性和集群吞吐能力。

集群运营现状

- 集群规模：10 万 + 级别线上实例，多地域部署，还在不断快速增长中。
- 业务的监控告警：集群对应用的启动和状态数据进行采集，container-init 自动集成业务监控信息，业务程序毋需关注，做到可插拔、可配置。
- 资源的健康告警：从资源的角度对 Node、Pod 和 Container 等重要数据监控采集，及时发现它们的状态信息，例如 Node 不可用、Container 不断重启等等。
- 定时巡检与对账：每天自动对所有宿主机进行状态检查，包括剩余磁盘量（数据卷）、D 进程数量、宿主机状态等，并对 AppKey 扩容数据和实际的 Pod 和容器数据同步校验，及时发现不一致情况。

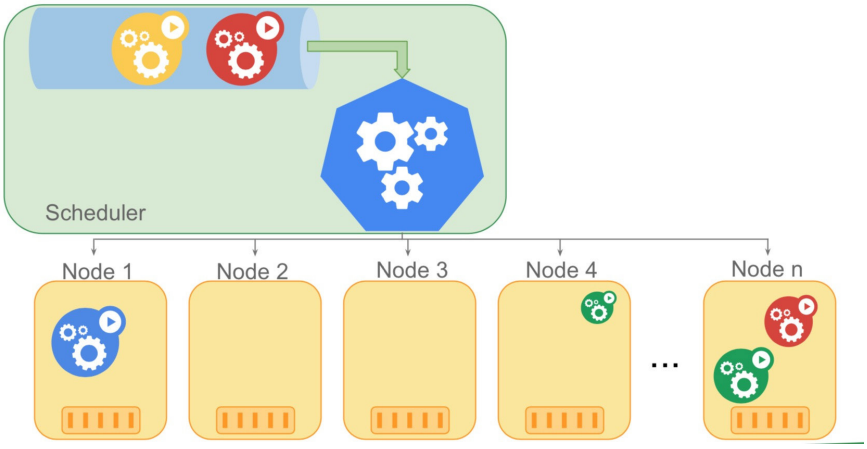
- 集群数据可视化：对当前集群状态，包括宿主机资源状态、服务数、Pod 数、容器化率、服务状态、扩缩容数据等等可视化；并提供了界面化的服务配置、宿主机下线以及 Pod 迁移操作入口。
- 容量规划与预测：提前感知集群资源状态，预先准备资源；基于规则和机器学习的方式感知流量和高峰，保证业务正常、稳定、高效地运行。

Kubernetes 优化与改造

kube-scheduler 性能优化

我们有集群在使用 1.6 版本的调度器，随着集群规模的不断增长，旧版本的 Kubernetes 调度器（1.10 之前版本）在性能和稳定性的问题逐渐凸显，由于调度器的吞吐量低，导致业务扩容超时失败，在规模近 3000 台的集群上，一次 Pod 的调度耗时在 5s 左右。Kubernetes 的调度器是队列化的调度器模型，一旦扩容高峰等待的 Pod 数量过多就会导致后面 Pod 的扩容超时。为此，我们对调度器性能进行了大幅度的优化，并取得了非常明显的提升，根据我们的实际生产环境验证，性能比优化前提升了 400% 以上。

Kubernetes 调度器工作模型如下：



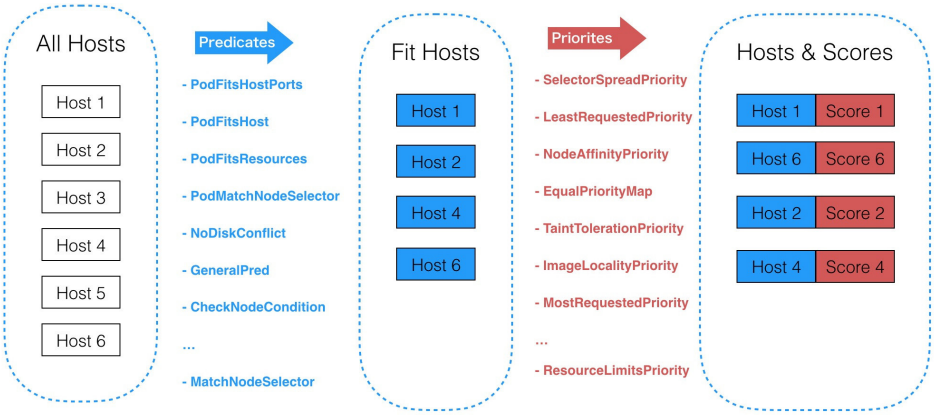
kube-scheduler 示意图

（kubernetes 调度器，图片来源于网络）

预选失败中断机制

一次调度过程在判断一个 Node 是否可作为目标机器时，主要分为三个阶段：

- 预选阶段：硬性条件，过滤掉不满足条件的节点，这个过程称为 Predicates。这是固定先后顺序的一系列过滤条件，任何一个 Predicate 不符合则放弃该 Node。
- 优选阶段：软性条件，对通过的节点按照优先级排序，称之为 Priorities。每一个 Priority 都是一个影响因素，都有一定的权重。
- 选定阶段：从优选列表中选择优先级最高的节点，称为 Select。选择的 Node 即为最终部署 Pod 的机器。

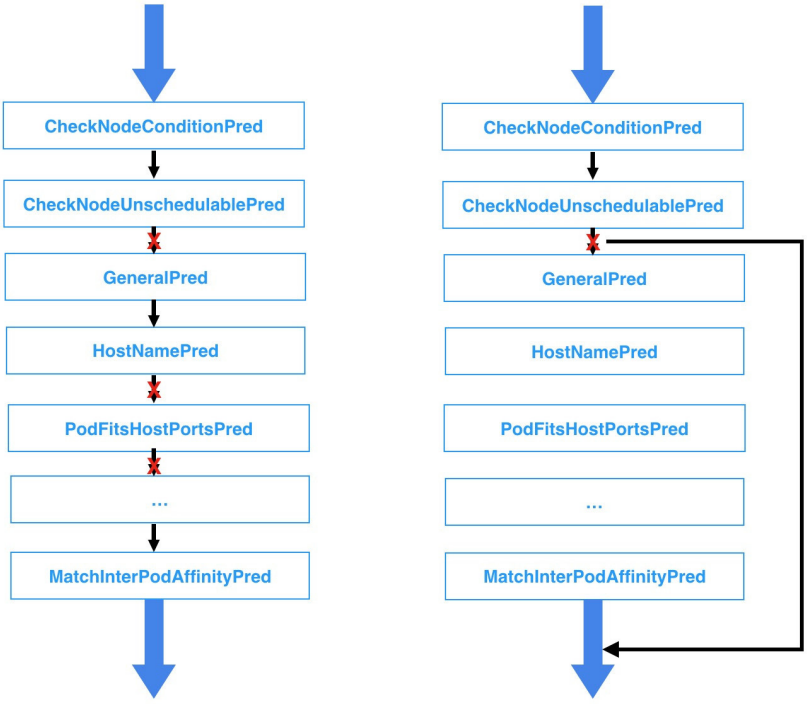


kube-scheduler 调度过程

通过深入分析调度过程可以发现，调度器在预选阶段即使已经知道当前 Node 不符合某个过滤条件仍然会继续判断后续的过滤条件是否符合。试想如果有上万台 Node 节点，这些判断逻辑会浪费很多计算时间，这也是调度器性能低下的一个重要因素。

为此，我们提出了“预选失败中断机制”，即一旦某个预选条件不满足，那么该 Node 即被立即放弃，后面的预选条件不再做判断计算，从而大大减少了计算量，调

度性能也大大提升。如下图所示：



kube-scheduler 的 Predicates 过程

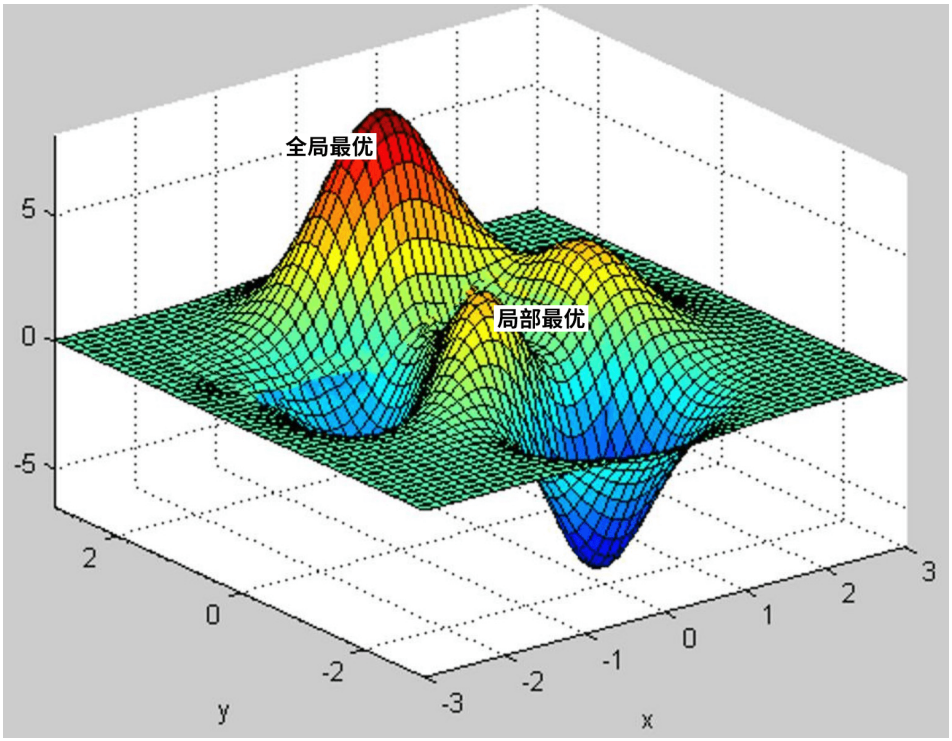
我们把该项优化贡献给了 Kubernetes 社区（详见 PR），增加了 alwaysCheckAllPredicates 策略选项，并在 Kubernetes1.10 版本发布并开始作为默认的调度策略，当然你也可以通过设置 alwaysCheckAllPredicates=true 使用原先的调度策略。

在实际测试中，调度器至少可以提升 40% 的性能，如果你目前在使用的 Kube-scheduler 的版本低于 1.10，那么建议你尝试升级到新的版本。

局部最优解

对于优化问题尤其是最优化问题，我们总希望找到全局最优的解或策略，但是当问题的复杂度过高，要考虑的因素和处理的信息量过多时，我们往往会倾向于接受局部最优解，因为局部最优解的质量不一定都是差的。尤其是当我们有确定的评判标

准，同时标明得出的解是可以接受的话，通常会接收局部最优的结果。这样，从成本、效率等多方面考虑，才是我们在实际工程中真正会采取的策略。



kube-scheduler 的局部最优解（图片来源于网络）

当前调度策略中，每次调度调度器都会遍历集群中所有的 Node，以便找出最优的节点，这在调度领域称之为 BestFit 算法。但是在生产环境中，我们是选取最优 Node 还是次优 Node，其实并没有特别大的区别和影响，有时候我们还是会避免选取最优的 Node（例如我们集群为了解决新上线机器后频繁在该机器上创建应用的问题，就将最优解随机化）。换句话说，找出局部最优解就能满足需求。

假设集群一共 1000 个 Node，一次调度过程 PodA，这其中有 700 个 Node 都能通过 Predicates（预选阶段），那么我们就会把所有的 Node 遍历并找出这 700 个 Node，然后经过得分排序找出最优的 Node 节点 NodeX。但是采用局部最优算法，即我们认为只要能找出 N 个 Node，并在这 N 个 Node 中选择得分最高的 Node 即

能满足需求，比如默认找出 100 个可以通过 Predicates（预选阶段）的 Node 即可，最优解就在这 100 个 Node 中选择。当然全局最优解 NodeX 也可能不在这 100 个 Node 中，但是我们在这 100 个 Node 中选择最优的 NodeY 也能满足要求。最好的情况是遍历 100 个 Node 就找出这 100 个 Node，也可能遍历了 200 个或者 300 个 Node 等等，这样我们可以大大减少计算时间，同时也不会对我们的调度结果产生太大的影响。

局部最优的策略是我们与社区合作共同完成的，这里面还涉及到如何做到公平调度和计算任务优化的细节（详见 PR1，PR2），该项优化在 Kubernetes 1.12 版本中发布，并作为当前默认调度策略，可以大幅度提升调度性能，尤其在大规模集群中的提升，效果非常明显。

kubelet 改造

风险可控性

前面提到，稳定性和风险可控性对大规模集群管理来说非常重要。从架构上来看，Kubelet 是离真实业务最近的集群管理组件，我们知道社区版本的 Kubelet 对本机资源管理有着很大的自主性，试想一下，如果某个业务正在运行，但是 Kubelet 由于出发了驱逐策略而把这个业务的容器干掉了会发生什么？这在我们的集群中是不应该发生的，所以需要收敛和封锁 Kubelet 的自决策能力，它对本机上业务容器的操作都应该从上层平台发起。

容器重启策略

Kernel 升级是日常的运维操作，在通过重启宿主机来升级 Kernel 版本的时候，我们发现宿主机重启后，上面的容器无法自愈或者自愈后版本不对，这会引发业务的不满，也造成了我们不小的运维压力。后来我们为 Kubelet 增加了一个重启策略（Reuse），同时保留了原生重启策略（Rebuild），保证容器系统盘和数据盘的信息都能保留，宿主机重启后容器也能自愈。

IP 状态保持

根据美团点评的网络环境，我们自研了 CNI 插件，并通过基于 Pod 唯一标识来申请和复用 IP。做到了应用 IP 在 Pod 迁移和容器重启之后也能复用，为业务上线和运维带来了不少的收益。

限制驱逐策略

我们知道 Kubelet 拥有节点自动修复的能力，例如在发现异常容器或不合规容器后，会对它们进行驱逐删除操作，这对于我们来说风险太大，我们允许容器在一些次要因素方面可以不合规。例如当 Kubelet 发现当前宿主机上容器个数比设置的最大容器个数大时，会挑选驱逐和删除某些容器，虽然正常情况下不会轻易发生这种问题，但是我们也需要对此进行控制，降低此类风险。

可扩展性

资源调配

在 Kubelet 的扩展性方面我们增强了资源的可操作性，例如为容器绑定 Numa 从而提升应用的稳定性；根据应用等级为容器设置 CPUShare，从而调整调度权重；为容器绑定 CPUSet 等等。

增强容器

我们打通并增强了业务对容器的配置能力，支持业务给自己的容器扩展 ulimit、io limit、pid limit、swap 等参数的同时也增强容器之间的隔离能力。

应用原地升级

大家都知道，Kubernetes 默认只要 Pod 的关键信息有改动，例如镜像信息，就会出发 Pod 的重建和替换，这在生产环境中代价是很大的，一方面 IP 和 HostName 会发生改变，另一方面频繁的重建也给集群管理带来了更多的压力，甚至还可能导致无法调度成功。为了解决该问题，我们打通了自上而下的应用原地升级功能，即可以动态高效地修改应用的信息，并能在原地（宿主机）进行升级。

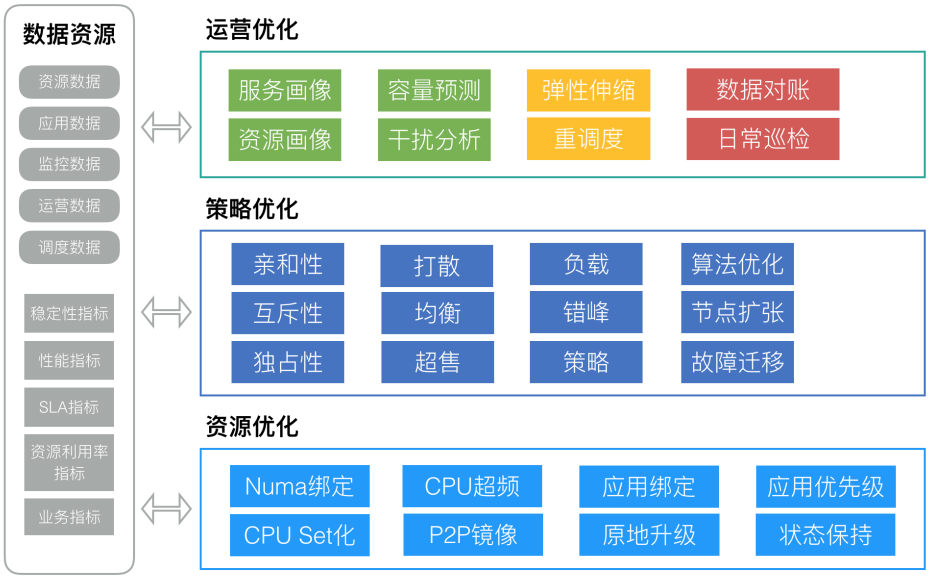
镜像分发

镜像分发是影响容器扩容时长的一个重要环节，我们采取了一系列手段来优化，

保证镜像分发效率高且稳定：

- 跨 Site 同步：保证服务器总能从就近的镜像仓库拉取到扩容用的镜像，减少拉取时间，降低跨 Site 带宽消耗。
- 基础镜像预分发：美团点评的基础镜像是构建业务镜像的公共镜像。业务镜像层是业务的应用代码，通常比基础镜像小很多。在容器扩容的时候如果基础镜像已经在本地，就只需要拉取业务镜像的部分，可以明显的加快扩容速度。为达到这样的效果，我们会把基础镜像事先分发到所有的服务器上。
- P2P 镜像分发：基础镜像预分发在有些场景会导致上千个服务器同时从镜像仓库拉取镜像，对镜像仓库服务和带宽带来很大的压力。因此我们开发了镜像 P2P 分发的功能，服务器不仅能从镜像仓库中拉取镜像，还能从其他服务器上获取镜像的分片。

资源管理与优化



资源管理与优化

优化关键技术

- 服务画像：对应用的 CPU、内存、网络、磁盘和网络 I/O 容量和负载画像，了解应用的特征、资源规格和应用类型以及不同时间对资源的真实使用，然后从服务角度和时间维度进行相关性分析，从而进行整体调度和部署优化。
- 亲和性和互斥性：哪些应用放在一起使整体计算能力比较少而吞吐能力比较高，它们就存在一定亲和性；反之如果应用之间存在资源竞争或相互影响，则它们之间就存在着互斥性。
- 场景优先：美团点评的业务大都是基本稳定的场景，所以场景划分很有必要。例如一类业务对延迟非常敏感，即使在高峰时刻也不允许有太多的资源竞争产生，这种场景就要避免和减少资源竞争引起的延迟，保证资源充足；一类业务在有些时间段需要的 CPU 资源可能会突破配置的上限，我们通过 CPU Set 化的方式让这类业务共享这部分资源，以便能够突破申请规格的机器资源限制，不仅服务能够获得更高的性能表现，同时也把空闲的资源利用了起来，资源使用率进一步提升。
- 弹性伸缩：应用部署做到流量预测、自动伸缩、基于规则的高低峰伸缩以及基于机器学习的伸缩机制。
- 精细化资源调配：基于资源共享和隔离技术做到了精细化的资源调度和分配，例如 Numa 绑定、任务优先级、CPU Set 化等等。

策略优化

调度策略的主要作用在两方面，一方面是按照既定策略部署目标机器；二是能做到集群资源的排布最优。

- 亲和性：有调用关系和依赖的应用，或哪些应用放在一起能使整体计算能力比较少、吞吐能力比较高，这些应用间就存在一定亲和性。我们的 CPU Set 化即是利用了对 CPU 的偏好构建应用的亲和性约束，让不同 CPU 偏好的应用互补。

- 互斥性：跟亲和性相对，主要是对有竞争关系或业务干扰的应用在调度时尽量分开部署。
- 应用优先级：应用优先级的划分是为我们解决资源竞争提供了前提。当前当容器发生资源竞争时，我们无法决策究竟应该让谁获得资源，当有了应用优先级的概念后，我们可以做到，在调度层，限制单台宿主机上重要应用的个数，减少单机的资源竞争，也为单机底层解决资源竞争提供可能；在宿主机层，根据应用优先级分配资源，保证重要应用的资源充足，同时也可运行低优先级应用。
- 打散性：应用的打散主要是为了容灾，在这里分为不同级别的打散。我们提供了不同级别的打散粒度，包括宿主机、Tor、机房、Zone 等等。
- 隔离与独占：这是一类特殊的应用，必须是独立使用一台宿主机或虚拟机隔离环境部署，例如搜索团队的业务。
- 特殊资源：特殊资源是满足某些业务对 GPU、SSD、特殊网卡等特殊硬件需求。

在线集群优化

在线集群资源的优化问题，不像离线集群那样可以通过预知资源需求从而达到非常好的效果，由于未来需求的未知性，在线集群很难在资源排布上达到离线集群的效果。针对在线集群的问题，我们从上层调度到底层的资源使用都采取了一系列的优化。

- Numa 绑定：主要是解决业务侧反馈服务不稳定的问题，通过绑定 Numa，将同一个应用的 CPU 和 Memory 绑定到最合适的 Numa Node 上，减少跨 Node 访问的开销，提升应用性能。
- CPU Set 化：将一组特性互补的应用绑定在同一组 CPU 上，从而让他们能充分使用 CPU 资源。
- 应用错峰：基于服务画像数据为应用错开高峰，减少资源竞争和相互干扰，提升业务 SLA。
- 重调度：资源排布优化，用更少的资源提升业务性能和 SLA；解决碎片问题，

提升资源的分配率。

- 干扰分析：基于业务监控数据指标和容器信息判断哪些容器有异常，提升业务 SLA，发现并处理异常应用。

结束语

当前，在以下几个方面我们正在积极探索：

- 在线 – 离线业务混合部署，进一步提升资源使用效率。
- 智能化调度，业务流量和资源使用感知调度，提升服务 SLA。
- 高性能、强隔离和更安全的容器技术。

作者简介

国梁，美团点评基础研发平台集群调度中心高级工程师。

招聘信息

美团点评基础研发平台集群调度中心，致力于打造高效的业界领先的集群管理与调度平台，通过企业级集群管理平台建设业界领先的云化解决方案，提高集群管理能力和稳定性，同时降低 IT 成本，加速公司的创新发展。同时随着 Kubernetes 已经成为业界的事实标准，美团点评也在逐步拥抱社区，参与开源并且在集群调度领域已经取得很大进展，也期待和业界同仁一起努力，共同提高集群管理和调度能力，降低整个行业的 IT 成本，协同创新发展。美团点评基础研发平台长期招聘集群管理与调度、弹性调度、Kubernetes 以及 Linux 内核方面的人才，有兴趣的同学可以发送简历到 tech@meituan.com 。

美团集群调度系统 HULK 技术演进

涂扬

本文根据美团基础架构部 / 弹性策略团队负责人涂扬在 2019 QCon（全球软件开发大会）上的演讲内容整理而成。本文涉及 Kubernetes 集群管理技术，美团相关的技术实践可参考此前发布的《美团点评 Kubernetes 集群管理实践》。

一、背景

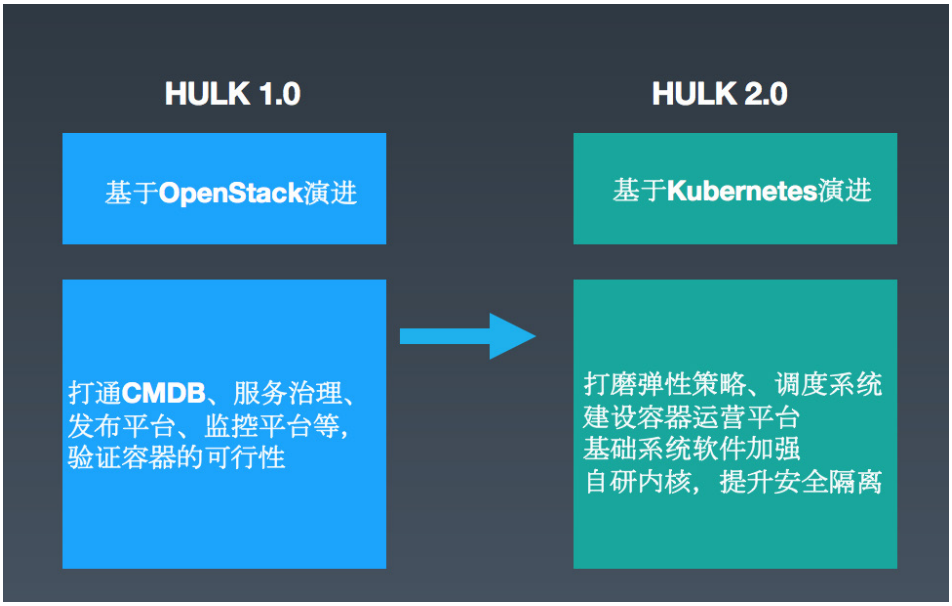
HULK 是美团的容器集群管理平台。在 HULK 之前，美团的在线服务大部分部署都是在 VM 上，在此期间，我们遇到了很大的挑战，主要包括以下两点：

- 环境配置信息不一致：部分业务线下验证正常，但线上验证却不正常。
- 业务扩容流程长：从申请机器、资源审核到服务部署，需要 5 分钟才能完成。

因为美团很多业务都具有明显的高低峰特性，大家一般会根据最高峰的流量情况来部署机器资源，然而在业务低峰期的时候，往往用不了那么多的资源。在这种背景下，我们希望打造一个容器集群管理平台来解决上述的痛点问题，于是 HULK 项目就应运而生了。

HULK 平台包含容器以及弹性调度系统，容器可以统一运行环境、提升交付效率，而弹性调度可以提升业务的资源利用率。在漫威里有个叫 HULK 的英雄，在情绪激动的时候会变成“绿巨人”，情绪平稳后则恢复人身，这一点跟我们容器的”弹性伸缩“特性比较相像，所以我们的系统就取名为”HULK“。

总的来讲，美团 HULK 的演进可以分为 1.0 和 2.0 两个阶段，如下图所示：



在早期，HULK 1.0 是基于 OpenStack 演进的一个集群调度系统版本。这个阶段工作的重点是将容器和美团的基础设施进行融合，比如打通 CMDB 系统、公司内部的服务治理平台、发布平台以及监控平台等等，并验证容器在生产环境的可行性。2018 年，基础架构部将底层的 OpenStack 升级为容器编排标准 Kubernetes，然后我们把这个版本称之为 HULK 2.0，新版本还针对在 1.0 运营过程中遇到的一些问题，对系统专门进行了优化和打磨，主要包括以下几个方面：

- 进一步打磨了弹性策略和调度系统。
- 构建了一站式容器运营平台。
- 对基础系统软件进行加强，自研内核，提升安全隔离能力。

截止发稿时，美团生产环境超过 1 万个应用在使用容器，容器数过 10 万。

二、HULK2.0 集群调度系统总体架构图

上图中，最上层是集群调度系统对接的各个平台，包括服务治理、发布平台、测试部署平台、CMDB 系统、监控平台等，我们将这些系统打通，业务就可以无感知地从 VM 迁移到容器中。其中：

- **容器弹性**：可以让接入的业务按需使用容器实例。
- **服务画像**：负责应用运行情况的搜集和统计，如 CPU/IO 使用、服务高峰期、上下游等信息，为弹性伸缩、调度系统提供支持。
- **容器编排和镜像管理**：负责对实例进行调度与应用实例构建。

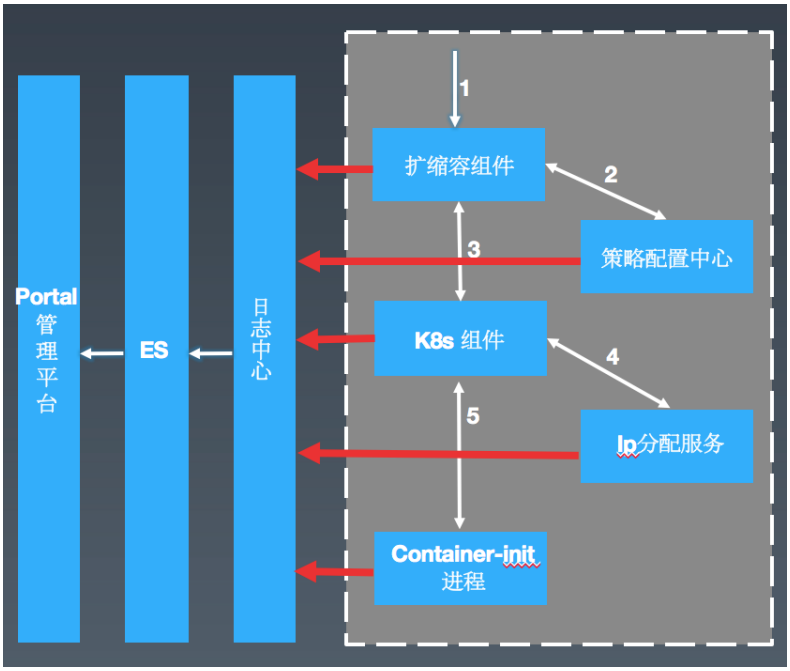
最底层的 HULK Agent 是我们在每个 Node 上的代理程序。此前，在美团技术团队官方博客上，我们也分享过底层的镜像管理和容器运行时相关内容，参见《美团容器技术研发实践》一文。而本文将重点阐述容器编排（调度系统）和容器弹性（弹性伸缩平台），以及团队遇到的一些问题以及对应的解决方案，希望对大家能有所启发。

三、调度系统痛点、解法

3.1 业务扩缩容异常

痛点：集群运维人员排查成本较高。

为了解决这个问题，我们可以先看一下调度系统的简化版架构，如下图所示：



可以看到，一次扩缩容请求基本上会经历以下这些流程：

a. 用户或者上层系统发起扩缩容请求。

b. 扩缩容组件从策略配置中心获取对应服务的配置信息。

c. 将对应的配置信息提交到美团自研的一个 API 服务（扩展的 K8s 组件），然后 K8s 各 Master 组件就按照原生的工作流程开始 Work。

d. 当某个实例调度到具体的 Node 上的时候，开始通过 IP 分配服务获取对应的 Hostname 和 IP。

e. Container-init 是一号进程，在容器内部拉起各个 Agent，然后启动应用程序。针对已经标准化接入的应用，会自动进行服务注册，从而承载流量。

而这些模块是由美团内部的不同同学分别进行维护，每次遇到问题时，就需要多个同学分别核对日志信息。可想而知，这种排查问题的方式的成本会有多高。

解法：类似于分布式调用链中的 traceId，每次扩缩容会生成一个 TaskId，我们在关键链路上进行打点的同时带上 TaskId，并按照约定的格式统一接入到美团点评

日志中心，然后在可视化平台 HULK Portal 进行展示。

落地效果：

- **问题排查提效**：之前排查类似问题，多人累计耗时平均需要半个小时。目前，1 个管理员通过可视化的界面即可达到分钟级定位到问题。
- **系统瓶颈可视化**：全链路上每个时段的平均耗时信息一览无遗。

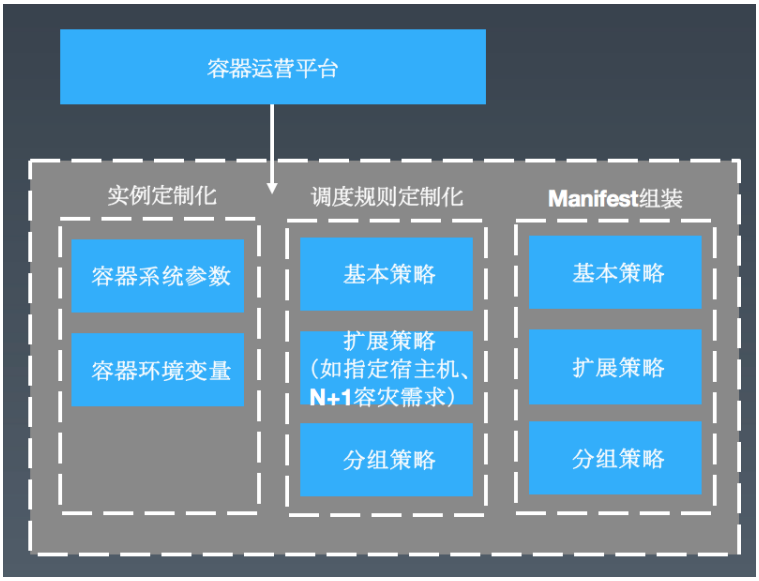
3.2 业务定制化需求

痛点：每次业务的特殊配置都可能变更核心链路代码，导致整体系统的灵活性不够。

具体业务场景如下：

- 业务希望能够去设置一些系统参数，比如开启 swap，设置 memlock、ulimit 等。
- 环境变量配置，比如应用名、ZooKeeper 地址等。

解法：建设一体化的调度策略配置中心，通过调度策略配置中心，可定制化调度规则。

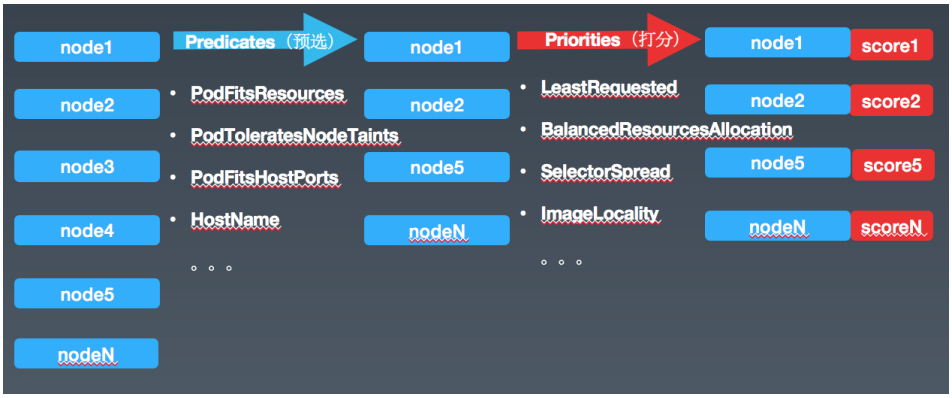


- 实例基本配置，比如业务想给机器加 Set 化、泳道标识。
- 实例的扩展配置：如部分业务，比如某些服务想将实例部署在包含特定硬件的宿主机，会对核心业务有 N+1 的容灾需求，并且还需要将实例部署在不同的 IDC 上。
- 相同配置的应用可以创建一个组，将应用和组进行关联。

在策略配置中心，我们会将这些策略进行 Manifest 组装，然后转换成 Kubernetes 可识别的 YAML 文件。

落地效果：实现了平台自动化配置，运维人员得到解放。

3.3 调度策略优化



接下来，介绍一下 Kubernetes 调度器 Scheduler 的默认行为：它启动之后，会一直监听 ApiServer，通过 ApiServer 去查看未 Bind 的 Pod 列表，然后根据特定的算法和策略选出一个合适的 Node，并进行 Bind 操作。具体的调度策略分为两个阶段：Predicates 预选阶段和 Priorities 打分阶段。

Predicates 预选阶段（一堆的预选条件）：PodFitsResources 检查是否有足够的资源（比如 CPU、内存）来满足一个 Pod 的运行需求，如果不满足，就直接过滤掉这个 Node。

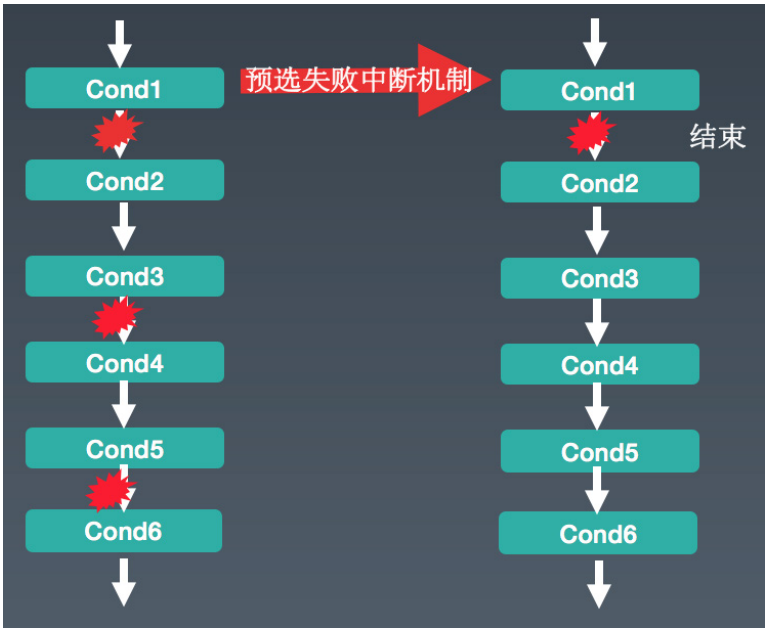
Priorities 打分阶段（一堆的优先级函数）：

- LeastRequested：CPU 和内存具有相同的权重，资源空闲比越高的节点得分越高。
- BalancedResourcesAllocation：CPU 和内存使用率越接近的节点得分越高。

将以上优先级函数算出来的值加权平均算出来一个得分（0-10），分数越高，节点越优。

痛点一：当集群达到 3000 台规模的时候，一次 Pod 调度耗时 5s 左右（K8s 1.6 版本）。如果在预选阶段，当前 Node 不符合过滤条件，依然会判断后续的过滤条件是否符合。假设有上万台 Node 节点，这种判断逻辑便会浪费较多时间，造成调度器的性能下降。

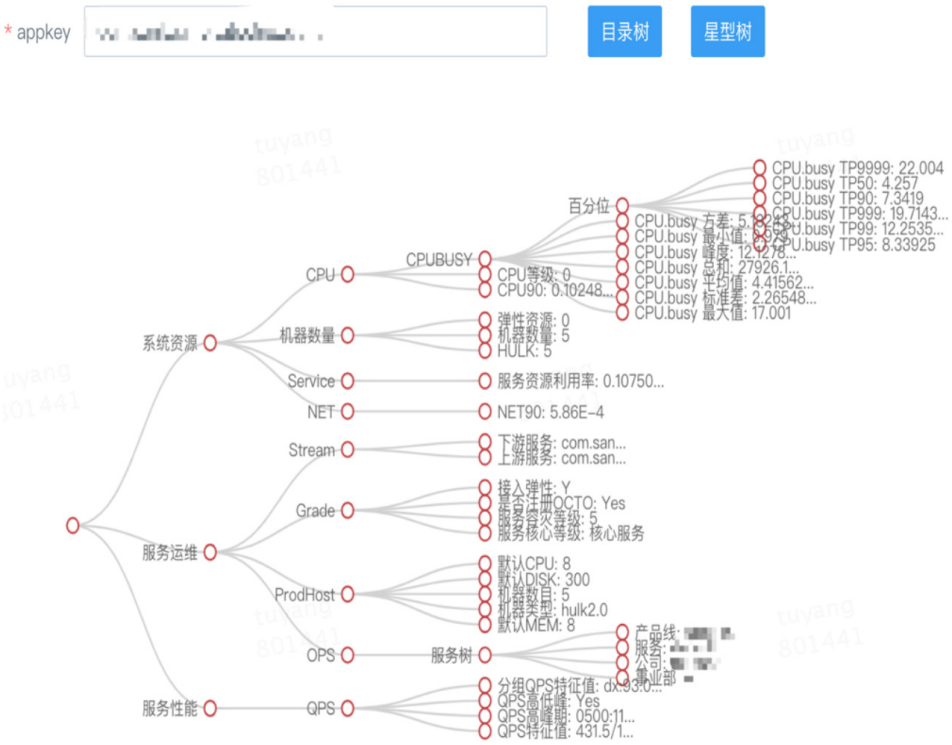
解法：当前 Node 中，如果遇到一个预选条件不满足（比较像是短路径原则），就将这个 Node 过滤掉，大大减少了计算量，调度性能也得到大幅提升。



成效：生产环境验证，提升了 40% 的性能。这个方案目前已经成为社区 1.10 版本默认的调度策略，技术细节可以参考 GitHub 上的 PR。

痛点二：资源利用率最大化和服务 SLA 保障之间的权衡。

解法：我们基于服务的行为数据构建了服务画像系统，下图是我们针对某个应用进行服务画像后的树图展现。



调度前：可以将有调用关系的 Pod 设置亲和性，竞争相同资源的 Pod 设置反亲和性，相同宿主机上最多包含 N 个核心应用。 **调度后**：经过上述规则调度后，在宿主机上如果依然出现了资源竞争，优先保障高优先级应用的 SLA。

3.4 重编排问题

痛点：

（1）容器重启 / 迁移场景：

- 容器和系统盘的信息丢失。
- 容器的 IP 变更。

（2）驱逐场景：Kubelet 会自动杀死一些违例容器，但有可能是非常核心的业务。

解法：

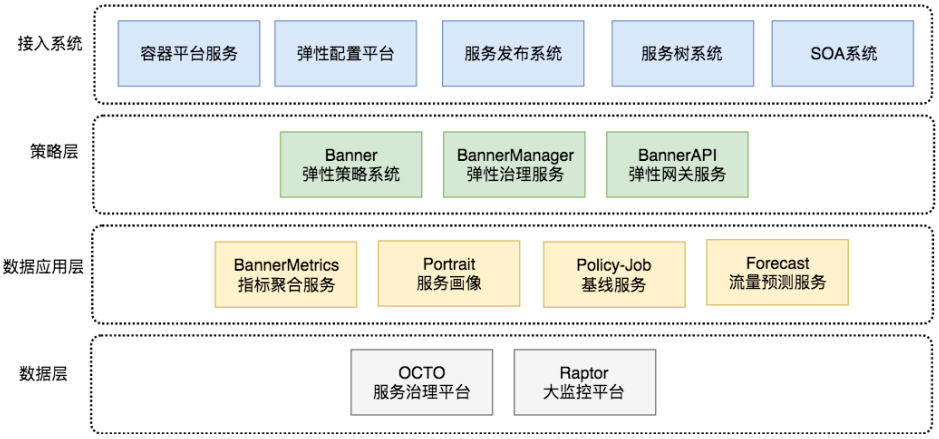
（1）容器重启 / 迁移场景：

- 新增 Reuse 策略，保留原生重启策略（Rebuild）。
- 定制化 CNI 插件，基于 Pod 标识申请和复用 IP。

（2）关闭原生的驱逐策略，通过外部组件来做决策。

四、弹性伸缩平台痛点、解法

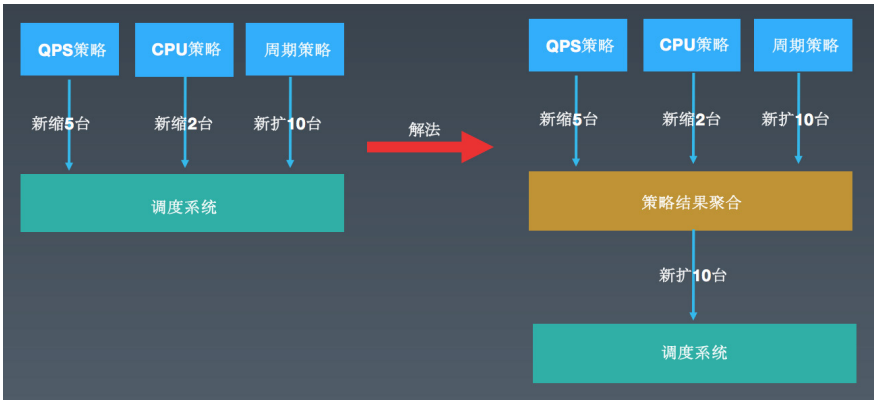
弹性伸缩平台整体架构图如下：



注：Raptor 是美团点评内部的大监控平台，整合了 CAT、Falcon 等监控产品。

在弹性伸缩平台演进的过程中，我们主要遇到了以下 5 个问题。

4.1 多策略决策不一致



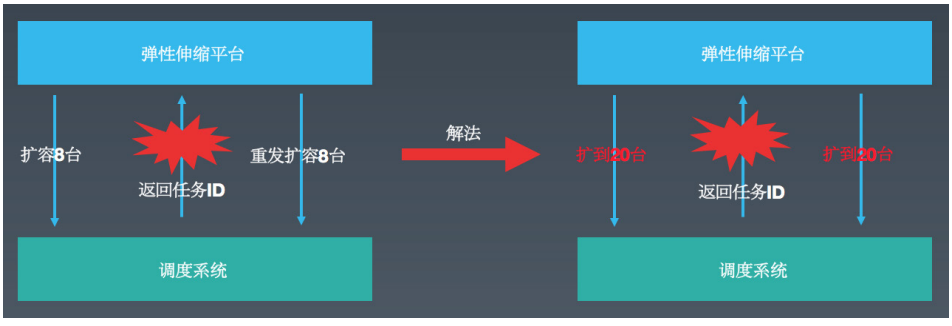
如上图所示，一个业务配置了 2 条监控策略和 1 条周期策略：

- **监控策略**：当某个指标（比如 QPS、CPU）超过阈值上限后开始扩容，低于阈值下限后开始缩容。
- **周期策略**：在某个固定的时间开始扩容，另外一个固定的时间开始缩容。

早期的设计是各条策略独自决策，扩容顺序有可能是：缩 5 台、缩 2 台、扩 10 台，也有可能是：扩 10 台、缩 5 台、缩 2 台，就可能造成一些无效的扩缩行为。

解法：增加了一个聚合层（或者把它称之为策略协商层），提供一些聚合策略：默认策略（多扩少缩）和权重策略（权重高的来决策扩缩行为），减少了大量的无效扩缩现象。

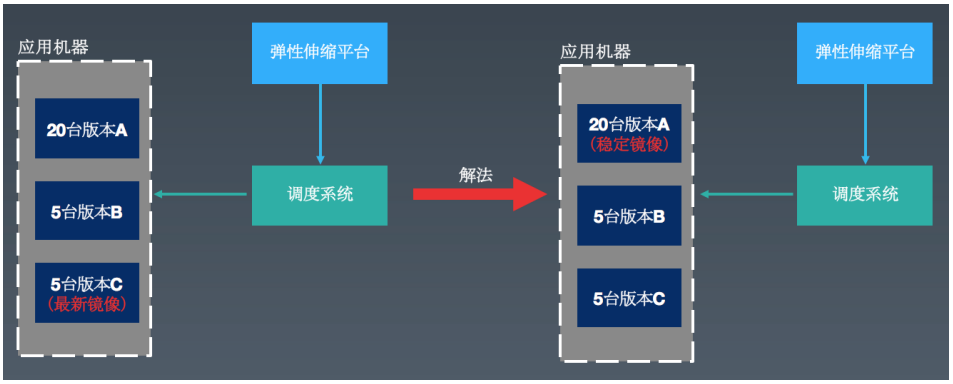
4.2 扩缩不幂等



如上图所示，聚合层发起具体扩缩容的时候，因之前采用的是增量扩容方式，在一些场景下会出现频繁扩缩现象。比如，原先 12 台，这个时候弹性伸缩平台告诉调度系统要扩容 8 台，在返回 TaskId 的过程中超时或保存 TaskId 失败了，这个时候弹性伸缩平台会继续发起扩容 8 台的操作，最后导致服务下有 28 台实例（不幂等）。

解法：采用按目标扩容方式，直接告诉对端，希望能扩容到 20 台，避免了短时间内的频繁扩缩容现象。

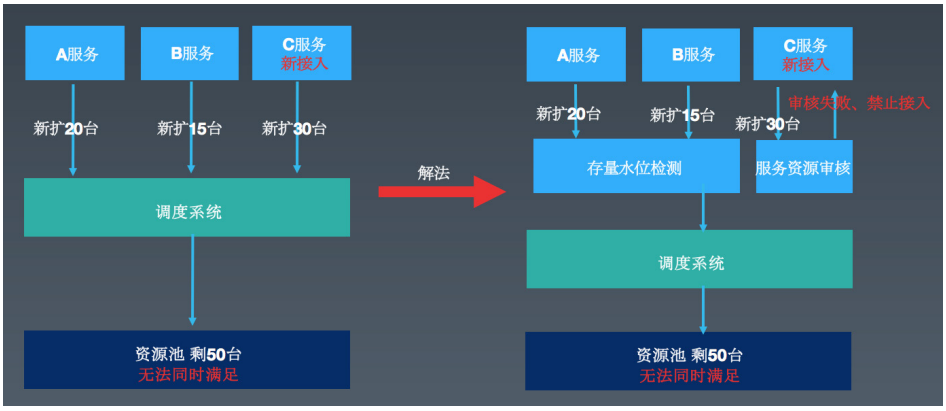
4.3 线上代码多版本



如上图所示，一个业务线上有 30 台机器，存在 3 个版本（A、B、C）。之前我们弹性扩容的做法是采用业务构建的最新镜像进行扩容，但在实际生产环境运行过程中却遇到问题。比如一些业务构建的最新镜像是用来做小流量测试的，本身的稳定性没有保障，高峰期扩容的时候会提升这个版本在线上机器中的比例，低峰期的时候又把之前稳定版本给缩容了，经过一段时间的频繁扩缩之后，最后线上遗留的实例可能都存在问题。

解法：基于约定优于配置原则，我们采用业务的稳定镜像（采用灰度发布流程将线上所有实例均覆盖过一遍的镜像，会自动标记为稳定镜像）进行扩容，这样就比较好地解决了这个问题。

4.4 资源保障问题

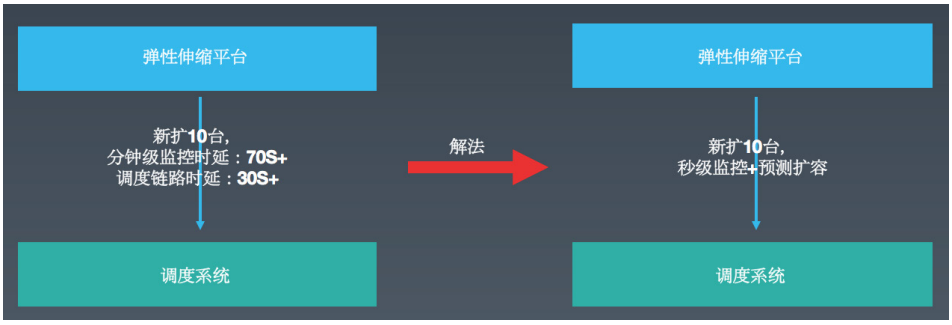


如上图所示，存量中有 2 个服务，一个需要扩容 20 台，一个需要扩容 15 台，这个时候如果新接入一个服务，同一时间需要扩容 30 台，但是资源池只剩余 50 台实例了。这个时候就意味着，谁先扩容谁就可以获得资源保障，后发起的请求就无法获得资源保障。

解法：

（1）**存量资源水位检测：**当存量资源的使用水位超过阈值的时候，比如达到 80% 的时候会有报警，告诉我们需要做资源补充操作。（2）**增量服务弹性资源预估：**如果这个服务通过预判算法评估，接入之后可能会导致存量服务的扩容得不到保障，则拒绝或者补充资源后，再让这个业务接入。

4.5 端到端时效问题



如图所示，我们的分钟级监控时延（比如 1:00:00~1:01:00 的监控数据，大概需要到 1:01:10 后可将采集到的所有数据聚合完成）是 70s+，调度链路时延是 30s+，整体需要上 100s+，在生产环境的业务往往会比较关注扩容时延。

解法：监控系统这块已经建设秒级监控功能。基于这些做法都属于后验性扩容，存在一定的延迟性，目前我们也在探索基于历史行为数据进行服务预测，在监控指标达到扩容阈值前的 1~2 分钟进行提前扩容。

五、经验总结

技术侧：

- 开源产品“本土化”：原生的 Kubernetes 需要和内部已有的基础设施，如服务树、发布系统、服务治理平台、监控系统等做融合，才能更容易在公司内进行落地。
- 调度决策：增量的调度均使用新策略来进行规范化，存量的可采用重调度器进行治理。
- 弹性伸缩：公有云在弹性伸缩这块是没有 SLA 保障的，但是做内部私有云，就需要做好扩容成功率、端到端时延这两块的 SLA 保障。

业务侧：

- 业务迁移：建设了全自动化迁移平台，帮助业务从 VM 自动迁移到容器，极大地降低了因迁移而带来的人力投入。
- 业务成本：使用 HULK 可较好地提升业务运维效率（HULK 具备资源利用率更高、弹性扩容、一键扩容等特点），降低了业务成本。

作者简介

涂扬，美团点评技术专家，现任基础架构部弹性策略团队负责人。

招聘信息

美团点评基础架构团队诚招高级、资深技术专家，Base 北京、上海。我们致力于建设美团点评全公司统一的高并发高性能分布式基础架构平台，涵盖数据库、分布式监控、服务治理、高性能通信、消息中间件、基础存储、容器化、集群调度等基础架构主要的技术领域。欢迎有兴趣的同学投送简历到 tech@meituan.com（邮件标题注明：基础架构部弹性策略团队）

保障 IDC 安全：分布式 HIDS 集群架构设计

陈驰

背景

近年来，互联网上安全事件频发，企业信息安全越来越受到重视，而 IDC 服务器安全又是纵深防御体系中的重要一环。保障 IDC 安全，常用的是基于主机型入侵检测系统 Host-based Intrusion Detection System，即 HIDS。在 HIDS 面对几十万台甚至上百万台规模的 IDC 环境时，系统架构该如何设计呢？复杂的服务器环境，网络环境，巨大的数据量给我们带来了哪些技术挑战呢?

需求描述

对于 HIDS 产品，我们安全部门的产品经理提出了以下需求：

1. 满足 50W-100W 服务器量级的 IDC 规模。
2. 部署在高并发服务器生产环境，要求 Agent 低性能低损耗。
3. 广泛的部署兼容性。
4. 偏向应用层和用户态入侵检测（可以和内核态检测部分解耦）。
5. 针对利用主机 Agent 排查漏洞的最急需场景提供基本的能力，可以实现海量环境下快速查找系统漏洞。
6. Agent 跟 Server 的配置下发通道安全。
7. 配置信息读取写入需要鉴权。
8. 配置变更历史记录。
9. Agent 插件具备自更新功能。

分析需求

首先，服务器业务进程优先级高，HIDS Agent 进程自己可以终止，但不能影响宿

主机的主要业务，这是第一要点，那么业务需要具备熔断功能，并具备自我恢复能力。

其次，进程保活、维持心跳、实时获取新指令能力，百万台 Agent 的全量控制时间一定要短。举个极端的例子，当 Agent 出现紧急情况，需要全量停止时，那么全量停止的命令下发，需要在 1–2 分钟内完成，甚至 30 秒、20 秒内完成。这些将会是很大的技术挑战。

还有对配置动态更新，日志级别控制，细分精确控制到每个 Agent 上的每个 HIDS 子进程，能自由地控制每个进程的启停，每个 Agent 的参数，也能精确的感知每台 Agent 的上线、下线情况。

同时，Agent 本身是安全 Agent，安全的因素也要考虑进去，包括通信通道的安全性，配置管理的安全性等等。

最后，服务端也要有一致性保障、可用性保障，对于大量 Agent 的管理，必须能实现任务分摊，并行处理任务，且保证数据的一致性。考虑到公司规模不断地扩大，业务不断地增多，特别是美团和大众点评合并后，面对的各种操作系统问题，产品还要具备良好的兼容性、可维护性等。

总结下来，产品架构要符合以下特性：

1. 集群高可用。
2. 分布式，去中心化。
3. 配置一致性，配置多版本可追溯。
4. 分治与汇总。
5. 兼容部署各种 Linux 服务器，只维护一个版本。
6. 节省资源，占用较少的 CPU、内存。
7. 精确的熔断限流。
8. 服务器数量规模达到百万级的集群负载能力。

技术难点

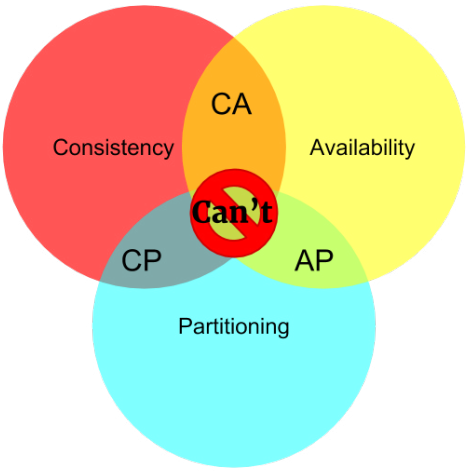
在列出产品需要实现的功能点、技术点后，再来分析下遇到的技术挑战，包括不限于以下几点：

- 资源限制，较小的 CPU、内存。
- 五十万甚至一百万台服务器的 Agent 处理控制问题。
- 量级大了后，集群控制带来的控制效率，响应延迟，数据一致性问题。
- 量级大了后，数据传输对整个服务器内网带来的流量冲击问题。
- 量级大了后，运行环境更复杂，Agent 异常表现的感知问题。
- 量级大了后，业务日志、程序运行日志的传输、存储问题，被监控业务访问量突增带来监控数据联动突增，对内网带宽，存储集群的爆发压力问题。

我们可以看到，技术难点几乎都是**服务器到达一定量级**带来的，对于大量的服务，集群分布式是业界常见的解决方案。

架构设计与技术选型

对于管理 Agent 的服务端来说，要实现高可用、容灾设计，那么一定要做多机房部署，就一定会遇到数据一致性问题。那么数据的存储，就要考虑分布式存储组件。 分布式数据存储中，存在一个定理叫 CAP 定理：



CAP-theorem.png

CAP 的解释

关于 CAP 定理，分为以下三点：

- 一致性（Consistency）：分布式数据库的数据保持一致。
- 可用性（Availability）：任何一个节点宕机，其他节点可以继续对外提供服务。
- 分区容错性（网络分区）Partition Tolerance：一个数据库所在的机器坏了，如硬盘坏了，数据丢失了，可以添加一台机器，然后从其他正常的机器把备份的数据同步过来。

根据定理，分布式系统只能满足三项中的两项而不可能满足全部三项。理解 CAP 定理的最简单方式是想象两个节点分处分区两侧。允许至少一个节点更新状态会导致数据不一致，即丧失了 Consistency。如果为了保证数据一致性，将分区一侧的节点设置为不可用，那么又丧失了 Availability。除非两个节点可以互相通信，才能既保证 Consistency 又保证 Availability，这又会导致丧失 Partition Tolerance。

参见：CAP Theorem。

CAP 的选择

为了容灾上设计，集群节点的部署，会选择的异地多机房，所以「Partition tolerance」是不可能避免的。**那么可选的是 AP 与 CP。**

在 HIDS 集群的场景里，各个 Agent 对集群持续可用性没有非常强的要求，在短暂时间内，是可以出现异常，出现无法通讯的情况。但最终状态必须要一致，不能存在集群下发关停指令，而出现个别 Agent 不听从集群控制的情况出现。所以，我们需要一个满足 CP 的产品。

满足 CP 的产品选择

在开源社区中，比较出名的几款满足 CP 的产品，比如 etcd、ZooKeeper、Consul 等。我们需要根据几款产品的特点，根据我们需求来选择符合我们需求的产品。

插一句，网上很多人说 Consul 是 AP 产品，这是个错误的描述。既然 Consul 支持分布式部署，那么一定会出现「网络分区」的问题，那么一定要支持「Partition tolerance」。另外，在 consul 的官网上自己也提到了这点 Consul uses a CP architecture, favoring consistency over availability.

> Consul is opinionated in its usage while Serf is a more flexible and general purpose tool. In CAP terms, Consul uses a CP architecture, favoring consistency over availability. Serf is an AP system and sacrifices consistency for availability. This means Consul cannot operate if the central servers cannot form a quorum while Serf will continue to function under almost all circumstances.

etcd、ZooKeeper、Consul 对比

借用 etcd 官网上 etcd 与 ZooKeeper 和 Consul 的比较图。

	ETCD	ZOOKEEPER	CONSUL	NEWSQL (CLOUD SPANNER, COCKROACHDB, TIDB)
Concurrency Primitives	Lock RPCs, Election RPCs, command line locks, command line elections, recipes in go	External curator recipes in Java	Native lock API	Rare, if any
Linearizable Reads	Yes	No	Yes	Sometimes
Multi-version Concurrency Control	Yes	No	No	Sometimes
Transactions	Field compares, Read, Write	Version checks, Write	Field compare, Lock, Read, Write	SQL-style
Change Notification	Historical and current key intervals	Current keys and directories	Current keys and prefixes	Triggers (sometimes)
User permissions	Role based	ACLs	ACLs	Varies (per-table GRANT, per-database roles)
HTTP/JSON API	Yes	No	Yes	Rarely
Membership Reconfiguration	Yes	>3.5.0	Yes	Yes
Maximum reliable database size	Several gigabytes	Hundreds of megabytes (sometimes several gigabytes)	Hundreds of MBs	Terabytes+
Minimum read linearization latency	Network RTT	No read linearization	RTT + fsync	Clock barriers (atomic, NTP)

etcd-ZooKeeper-Consul

在我们 HIDS Agent 的需求中，除了基本的服务发现 、配置同步 、配置多版本控制 、变更通知等基本需求外，我们还有基于产品安全性上的考虑，比如传输通道加密、用户权限控制、角色管理、基于 Key 的权限设定等，这点 etcd 比较符合我们要求。很多大型公司都在使用，比如 Kubernetes、AWS、OpenStack、Azure、Google Cloud、Huawei Cloud 等，并且 etcd 的社区支持非常好。基于这几点因

素，我们选择 etcd 作为 HIDS 的分布式集群管理。

选择 etcd

对于 etcd 在项目中的应用，我们分别使用不同的 API 接口实现对应的业务需求，按照业务划分如下：

- Watch 机制来实现配置变更下发，任务下发的实时获取机制。
- 脑裂问题在 etcd 中不存在，etcd 集群的选举，只有投票达到 N/2+1 以上，才会选做 Leader，来保证数据一致性。另外一个网络分区的 Member 节点将无主。
- 语言亲和性，也是 Golang 开发的，Client SDK 库稳定可用。
- Key 存储的数据结构支持范围性的 Key 操作。
- User、Role 权限设定不同读写权限，来控制 Key 操作，避免其他客户端修改其他 Key 的信息。
- TLS 来保证通道信息传递安全。
- Txn 分布式事务 API 配合 Compare API 来确定主机上线的 Key 唯一性。
- Lease 租约机制，过期 Key 释放，更好的感知主机下线信息。
- etcd 底层 Key 的存储为 BTree 结构，查找时间复杂度为 O（$\log_8 n$），百万级甚至千万级 Key 的查找耗时区别不大。

etcd Key 的设计

前缀按角色设定：

- Server 配置下发使用 /hids/server/config/{hostname}/master。
- Agent 注册上线使用 /hids/agent/master/{hostname}。
- Plugin 配置获取使用 /hids/agent/config/{hostname}/plugin/ID/conf_name。

Server Watch /hids/server/config/{hostname}/master，实现 Agent 主机上线的瞬间感知。Agent Watch /hids/server/config/{hostname}/ 来获取配置变更，

任务下发。Agent 注册的 Key 带有 Lease Id，并启用 keepalive，下线后瞬间感知。（异常下线，会有 1/3 的 keepalive 时间延迟）

关于 Key 的权限，根据不同前缀，设定不同 Role 权限。赋值给不同的 User，来实现对 Key 的权限控制。

etcd 集群管理

在 etcd 节点容灾考虑，考虑 DNS 故障时，节点会选择部署在多个城市，多个机房，以我们服务器机房选择来看，在大部分机房都有一个节点，综合承载需求，我们选择了 N 台服务器部署在个别重要机房，来满足负载、容灾需求。但对于 etcd 这种分布式一致性强的组件来说，每个写操作都需要 N/2−1 的节点确认变更，才会将写请求写入数据库中，再同步到各个节点，那么意味着节点越多，需要确认的网络请求越多，耗时越多，反而会影响集群节点性能。这点，我们后续将提升单个服务器性能，以及牺牲部分容灾性来提升集群处理速度。

客户端填写的 IP 列表，包含域名、IP。IP 用来规避 DNS 故障，域名用来做 Member 节点更新。最好不要使用 Discover 方案，避免对内网 DNS 服务器产生较大压力。

同时，在配置 etcd 节点的地址时，也要考虑到内网 DNS 故障的场景，地址填写会混合 IP、域名两种形式。

1. IP 的地址，便于规避内网 DNS 故障。
2. 域名形式，便于做个别节点更替或扩容。

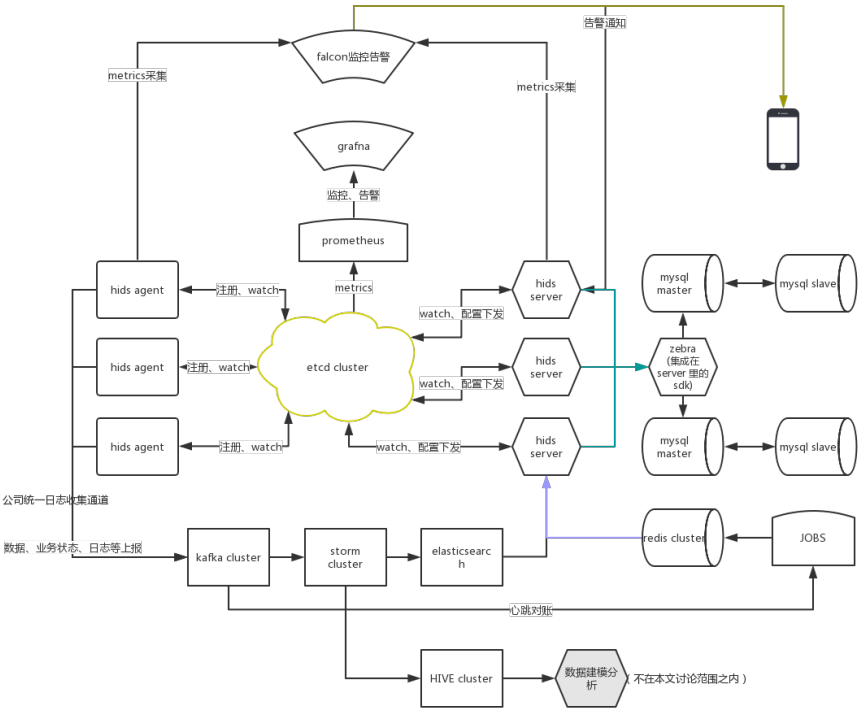
我们在设计产品架构时，为了安全性，开启了 TLS 证书认证，当节点变更时，证书的生成也同样要考虑到上面两种方案的影响，证书里需要包含固定 IP，以及 DNS 域名范围的两种格式。

etcd Cluster 节点扩容

节点扩容，官方手册上也有完整的方案，etcd 的 Client 里实现了健康检测与故障迁移，能自动的迁移到节点 IP 列表中的其他可用 IP。也能定时更新 etcd Node

List，对于 etcd Cluster 的集群节点变更来说，不存在问题。需要我们注意的是，TLS 证书的兼容。

分布式 HIDS 集群架构图



hids-cluster-architecture

集群核心组件高可用，所有 Agent、Server 都依赖集群，都可以无缝扩展，且不影响整个集群的稳定性。即使 Server 全部宕机，也不影响所有 Agent 的继续工作。

在以后 Server 版本升级时，Agent 不会中断，也不会带来雪崩式的影响。etcd 集群可以做到单节点升级，一直到整个集群升级，各个组件全都解耦。

编程语言选择

考虑到公司服务器量大，业务复杂，需求环境多变，操作系统可能包括各种

Linux 以及 Windows 等。为了保证系统的兼容性，我们选择了 Golang 作为开发语言，它具备以下特点：

1. 可以静态编译，直接通过 syscall 来运行，不依赖 libc，兼容性高，可以在所有 Linux 上执行，部署便捷。
2. 静态编译语言，能将简单的错误在编译前就发现。
3. 具备良好的 GC 机制，占用系统资源少，开发成本低。
4. 容器化的很多产品都是 Golang 编写，比如 Kubernetes、Docker 等。
5. etcd 项目也是 Golang 编写，类库、测试用例可以直接用，SDK 支持快速。
6. 良好的 CSP 并发模型支持，高效的协程调度机制。

产品架构大方向

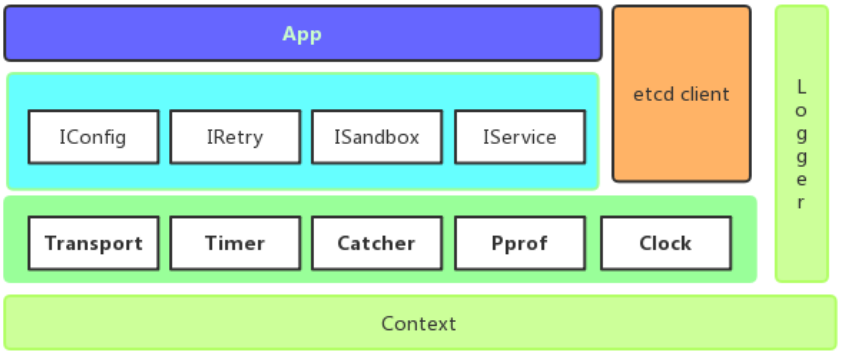
HIDS 产品研发完成后，部署的服务都运行着各种业务的服务器，业务的重要性排在第一，我们产品的功能排在后面。为此，确定了几个产品的大方向：

- 高可用，数据一致，可横向扩展。
- 容灾性好，能应对机房级的网络故障。
- 兼容性好，只维护一个版本的 Agent。
- 依赖低，不依赖任何动态链接库。
- 侵入性低，不做 Hook，不做系统类库更改。
- 熔断降级可靠，宁可自己挂掉，也不影响业务 。

产品实现

篇幅限制，仅讨论框架设计、熔断限流、监控告警、自我恢复以及产品实现上的主进程与进程监控。

框架设计



hids-framework

如上图，在框架的设计上，封装常用类库，抽象化定义 Interface，剥离 etcd Client，全局化 Logger，抽象化 App 的启动、退出方法。使得各模块（以下简称 App）只需要实现自己的业务即可，可以方便快捷的进行逻辑编写，无需关心底层实现、配置来源、重试次数、熔断方案等等。

沙箱隔离

考虑到子进程不能无限的增长下去，那么必然有一个进程包含多个模块的功能，各 App 之间既能使用公用底层组件（Logger、etcd Client 等），又能让彼此之间互不影响，这里进行了沙箱化处理，各个属性对象仅在各 App 的 sandbox 里生效。同样能实现了 App 进程的性能熔断，停止所有的业务逻辑功能，但又能具有基本的自我恢复功能。

IConfig

对各 App 的配置抽象化处理，实现 IConfig 的共有方法接口，用于对配置的函数调用，比如 Check 的检测方法，检测配置合法性，检测配置的最大值、最小值范围，规避使用人员配置不在合理范围内的情况，从而避免带来的风险。

框架底层用 Reflect 来处理 JSON 配置，解析读取填写的配置项，跟 Config 对象对比，填充到对应 Struct 的属性上，允许 JSON 配置里只填写变化的配置，没填

写的配置项，则使用 Config 对应 Struct 的默认配置。便于灵活处理配置信息。

```
type IConfig interface {
      Check() error // 检测配置合法性
}

func ConfigLoad(confByte []byte, config IConfig) (IConfig, error) {
...
// 反射生成临时的 IConfig
      var confTmp IConfig
      confTmp = reflect.New(reflect.ValueOf(config).Elem().Type()).
Interface().(IConfig)
...

      // 反射 confTmp 的属性
      confTmpReflect := reflect.TypeOf(confTmp).Elem()
      confTmpReflectV := reflect.ValueOf(confTmp).Elem()

      // 反射 config IConfig
      configReflect := reflect.TypeOf(config).Elem()
      configReflectV := reflect.ValueOf(config).Elem()
...
      for i = 0; i < num; i++ {
    // 遍历处理每个 Field
              envStructTmp := configReflect.Field(i)
      // 根据配置中的项，来覆盖默认值
      if envStructTmp.Type == confStructTmp.Type {
        configReflectV.FieldByName(envStructTmp.Name).
Set(confTmpReflectV.Field(i))
```

Timer、Clock 调度

在业务数据产生时，很多地方需要记录时间，时间的获取也会产生很多系统调用。尤其是在每秒钟产生成千上万个事件，这些事件都需要调用获取时间接口，进行 clock_gettime 等系统调用，会大大增加系统 CPU 负载。 而很多事件产生时间的准确性要求不高，精确到秒，或者几百个毫秒即可，那么框架里实现了一个颗粒度符合需求的（比如 100ms、200ms、或者 1s 等）间隔时间更新的时钟，即满足事件对时间的需求，又减少了系统调用。

同样，在有些 Ticker 场景中，Ticker 的间隔颗粒要求不高时，也可以合并成一个 Ticker，减少对 CPU 时钟的调用。

Catcher

在多协程场景下，会用到很多协程来处理程序，对于个别协程的 panic 错误，上层线程要有一个良好的捕获机制，能将协程错误抛出去，并能恢复运行，不要让进程崩溃退出，提高程序的稳定性。

抽象接口

框架底层抽象化封装 Sandbox 的 Init、Run、Shutdown 接口，规范各 App 的对外接口，让 App 的初始化、运行、停止等操作都标准化。App 的模块业务逻辑，不需要关注 PID 文件管理，不关注与集群通讯，不关心与父进程通讯等通用操作，只需要实现自己的业务逻辑即可。App 与框架的统一控制，采用 Context 包以及 Sync.Cond 等条件锁作为同步控制条件，来同步 App 与框架的生命周期，同步多协程之间同步，并实现 App 的安全退出，保证数据不丢失。

限流

网络 IO

- 限制数据上报速度。
- 队列存储数据任务列表。
- 大于队列长度数据丢弃。
- 丢弃数据总数计数。
- 计数信息作为心跳状态数据上报到日志中心，用于数据对账。

磁盘 IO

程序运行日志，对日志级别划分，参考 /usr/include/sys/syslog.h：

- LOG_EMERG
- LOG_ALERT
- LOG_CRIT
- LOG_ERR
- LOG_WARNING

- LOG_NOTICE
- LOG_INFO
- LOG_DEBUG

在代码编写时，根据需求选用级别。级别越低日志量越大，重要程度越低，越不需要发送至日志中心，写入本地磁盘。那么在异常情况排查时，方便参考。

日志文件大小控制，分 2 个文件，每个文件不超过固定大小，比如 20M、50M 等。并且，对两个文件进行来回写，避免日志写满磁盘的情况。

IRetry

为了加强 Agent 的鲁棒性，不能因为某些 RPC 动作失败后导致整体功能不可用，一般会有重试功能。Agent 跟 etcd Cluster 也是 TCP 长连接（HTTP2），当节点重启更换或网络卡顿等异常时，Agent 会重连，那么重连的频率控制，不能是死循环般的重试。假设服务器内网交换机因内网流量较大产生抖动，触发了 Agent 重连机制，不断的重连又加重了交换机的负担，造成雪崩效应，这种设计必须要避免。 在每次重试后，需要做一定的回退机制，常见的指数级回退，比如如下设计，在规避雪崩场景下，又能保障 Agent 的鲁棒性，设定最大重试间隔，也避免了 Agent 失控的问题。

```
// 网络库重试 Interface
type INetRetry interface {
      // 开始连接函数
      Connect() error
      String() string
      // 获取最大重试次数
      GetMaxRetry() uint
     ...
}
// 底层实现
func (this *Context) Retry(netRetry INetRetry) error {
...
    maxRetries = netRetry.GetMaxRetry() // 最大重试次数
    hashMod = netRetry.GetHashMod()
for {
      if c.shutting {
              return errors.New("c.shutting is true...")
```

```
        }
        if maxRetries > 0 && retries >= maxRetries {
                c.logger.Debug("Abandoning %s after %d retries.",
netRetry.String(), retries)
                return errors.New(" 超过最大重试次数 ")
        }
...
        if e := netRetry.Connect(); e != nil {
                delay = 1 << retries
                if delay == 0 {
                        delay = 1
                }
                delay = delay * hashInterval
...
                c.logger.Emerg("Trying %s after %d seconds ,
retries:%d,error:%v", netRetry.String(), delay, retries, e)
                time.Sleep(time.Second * time.Duration(delay))
        }
...
}
```

事件拆分

百万台 IDC 规模的 Agent 部署，在任务执行、集群通讯或对宿主机产生资源影响时，务必要错峰进行，根据每台主机的唯一特征取模，拆分执行，避免造成雪崩效应。

监控告警

古时候，行军打仗时，提倡「兵马未动，粮草先行」，无疑是冷兵器时代决定胜负走向的重要因素。做产品也是，尤其是大型产品，要对自己运行状况有详细的掌控，做好监控告警，才能确保产品的成功。

对于 etcd 集群的监控，组件本身提供了 Metrics 数据输出接口，官方推荐了 Prometheus 来采集数据，使用 Grafana 来做聚合计算、图标绘制，我们做了 Alert 的接口开发，对接了公司的告警系统，实现 IM、短信、电话告警。

Agent 数量感知，依赖 Watch 数字，实时准确感知。

如下图，来自产品刚开始灰度时的某一时刻截图，Active Streams（即 etcd Watch 的 Key 数量）即为对应 Agent 数量，每次灰度的产品数量。因为该操作，是

Agent 直接与集群通讯，并且每个 Agent 只 Watch 一个 Key。且集群数据具备唯一性、一致性，远比心跳日志的处理要准确的多。

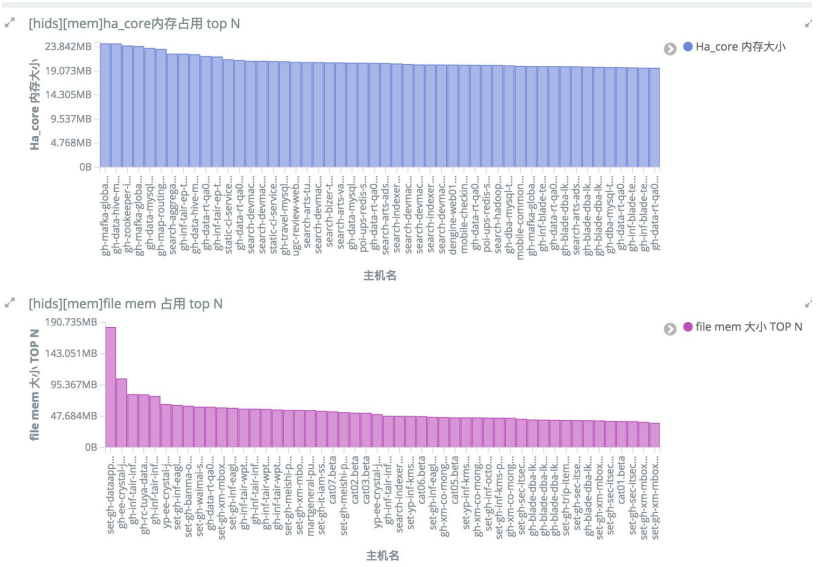
etcd-Grafana-Watcher-Monitor

etcd 集群 Members 之间健康状况监控

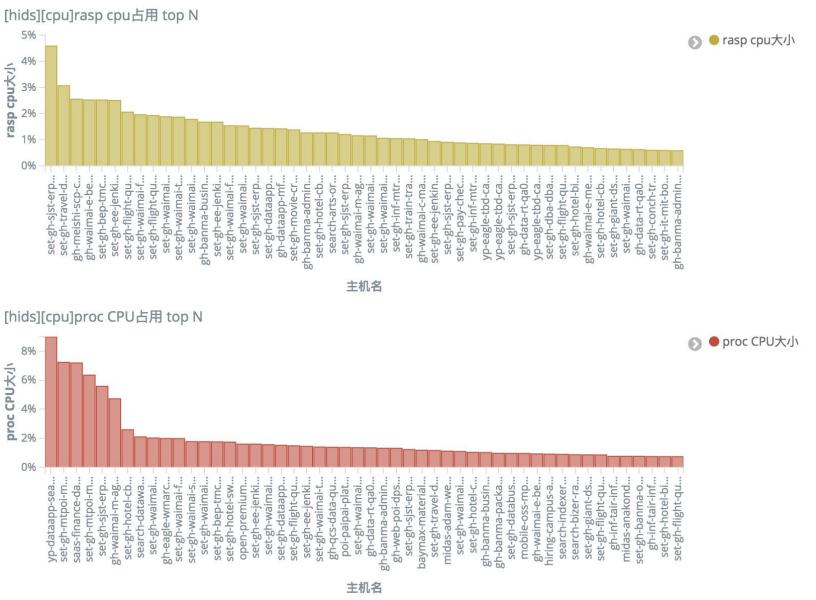
etcd-Grafana-GC-Heap-Objects

用于监控管理 etcd 集群的状况，包括 Member 节点之间数据同步，Leader 选举次数，投票发起次数，各节点的内存申请状况，GC 情况等，对集群的健康状况做全面掌控。

程序运行状态监控告警



agent-mem-es



agent-cpu-es

全量监控 Agent 的资源占用情况，统计每天使用最大 CPU\ 内存的主机 Agent，确定问题的影响范围，及时做策略调整，避免影响到业务服务的运行。并在后续版本上逐步做调整优化。

百万台服务器，日志告警量非常大，这个级别的告警信息的筛选、聚合是必不可少的。减少无用告警，让研发运维人员疲于奔命，也避免无用告警导致研发人员放松了警惕，前期忽略个例告警，先解决主要矛盾。

- 告警信息分级，告警信息细分 ID。
- 根据告警级别过滤，根据告警 ID 聚合告警，来发现同类型错误。
- 根据告警信息的所在机房、项目组、产品线等维度来聚合告警，来发现同类型错误。

数据采集告警

- 单机数据数据大小、总量的历史数据对比告警。
- 按机房、项目组、产品线等维度的大小、总量等维度的历史数据对比告警。
- 数据采集大小、总量的对账功能，判断经过一系列处理流程的日志是否丢失的监控告警。

熔断

- 针对单机 Agent 使用资源大小的阈值熔断，CPU 使用率，连续 N 次触发大于等于 5%，则进行保护性熔断，退出所有业务逻辑，以保护主机的业务程序优先。
- Master 进程进入空闲状态，等待第二次时间 Ticker 到来，决定是否恢复运行。
- 各个 App 基于业务层面的监控熔断策略。

灰度管理

在前面的配置管理中的 etcd Key 设计里，已经细分到每个主机（即每个 Agent）一个 Key。那么，服务端的管理，只要区分该主机所属机房、环境、群组、产品线即可，那么，我们的管理 Agent 的颗粒度可以精确到每个主机，也就是支持任意纬度

的灰度发布管理与命令下发。

数据上报通道

组件名为 log_agent ，是公司内部统一日志上报组件，会部署在每一台 VM、Docker 上。主机上所有业务均可将日志发送至该组件。 log_agent 会将日志上报到 Kafka 集群中，经过处理后，落入 Hive 集群中。（细节不在本篇讨论范围）

主进程

主进程实现跟 etcd 集群通信，管理整个 Agent 的配置下发与命令下发；管理各个子模块的启动与停止；管理各个子模块的 CPU、内存占用情况，对资源超标进行进行熔断处理，让出资源，保证业务进程的运行。

插件化管理其他模块，多进程模式，便于提高产品灵活性，可更简便的更新启动子模块，不会因为个别模块插件的功能、BUG 导致整个 Agent 崩溃。

进程监控

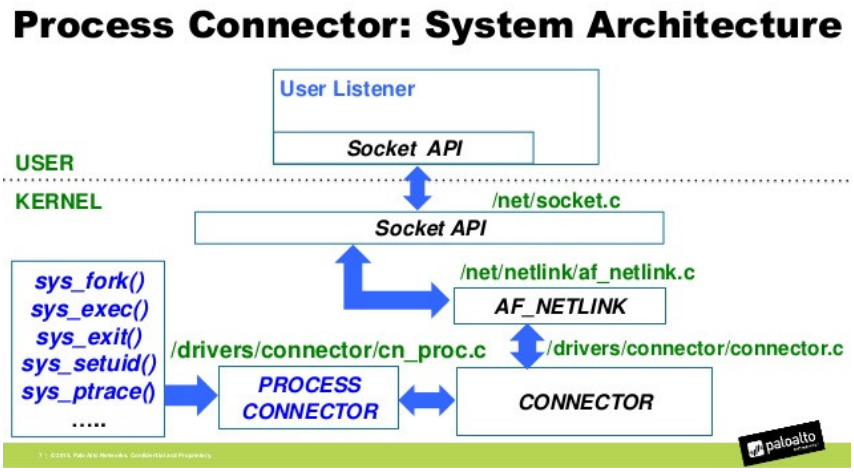
方案选择

我们在研发这产品时，做了很多关于 linux 进程创建监控的调研，不限于安全产品，大约有下面三种技术方案：

方案	Docker兼容性	开发难度	数据准确性	系统侵入性
cn_proc	不支持 Docker	一般	存在内核拿到的 PID，在 /proc/ 下丢失的情况	无
Audit	不支持 Docker	一般	同 cn_proc	弱，但依赖 Auditd
Hook	定制	高	精确	强

对于公司的所有服务器来说，几十万台都是已经在运行的服务器，新上的任何产品，都尽量避免对服务器有影响，更何况是所有服务器都要部署的 Agent。 意味着我们在选择系统侵入性来说，优先选择最小侵入性的方案。

对于 Netlink 的方案原理，可以参考这张图（来自 :kernel-proc-connector-and-containers）



process-connector

系统侵入性比较

- cn_proc 跟 Autid 在「系统侵入性」和「数据准确性」来说，cn_proc 方案更好，而且使用 CPU、内存等资源情况，更可控。
- Hook 的方案，对系统侵入性太高了，尤其是这种最底层做 HOOK syscall 的做法，万一测试不充分，在特定环境下，有一定的概率会出现 Bug，而在百万 IDC 的规模下，这将成为大面积事件，可能会造成重大事故。

兼容性上比较

- cn_proc 不兼容 Docker，这个可以在宿主机上部署来解决。
- Hook 的方案，需要针对每种 Linux 的发行版做定制，维护成本较高，且不符合长远目标（收购外部公司时遇到各式各样操作系统问题）

数据准确性比较

在大量 PID 创建的场景，比如 Docker 的宿主机上，内核返回 PID 时，因为 PID 返回非常多非常快，很多进程启动后，立刻消失了，另外一个线程都还没去读取 /proc/，进程都丢失了，场景常出现在 Bash 执行某些命令。

最终，我们选择 Linux Kernel Netlink 接口的 cn_proc 指令作为我们进程监控

方案，借助对 Bash 命令的收集，作为该方案的补充。当然，仍然存在丢数据的情况，但我们为了系统稳定性，产品侵入性低等业务需求，牺牲了一些安全性上的保障。

对于 Docker 的场景，采用宿主机运行，捕获数据，关联到 Docker 容器，上报到日志中心的做法来实现。

遇到的问题

内核 Netlink 发送数据卡住

内核返回数据太快，用户态 ParseNetlinkMessage 解析读取太慢，导致用户态网络 Buff 占满，内核不再发送数据给用户态，进程空闲。对于这个问题，我们在用户态做了队列控制，确保解析时间的问题不会影响到内核发送数据。对于队列的长度，我们做了定值限制，生产速度大于消费速度的话，可以丢弃一些数据，来保证业务正常运行，并且来控制进程的内存增长问题。

疑似“内存泄露”问题

在一台 Docker 的宿主机上，运行了 50 个 Docker 实例，每个 Docker 都运行了复杂的业务场景，频繁的创建进程，在最初的产品实现上，启动时大约 10M 内存占用，一天后达到 200M 的情况。

经过我们 Debug 分析发现，在 ParseNetlinkMessage 处理内核发出的消息时，PID 频繁创建带来内存频繁申请，对象频繁实例化，占用大量内存。同时，在 Golang GC 时，扫描、清理动作带来大量 CPU 消耗。在代码中，发现对于 **linux/connector.h** 里的 struct cb_msg、**linux/cn_proc.h** 里的 struct proc_event 结构体频繁创建，带来内存申请等问题，以及 Golang 的 GC 特性，内存申请后，不会在 GC 时立刻归还操作系统，而是在后台任务里，逐渐的归还到操作系统，见：debug.FreeOSMemory

> FreeOSMemory forces a garbage collection followed by an attempt to return as much memory to the operating system as possible. (Even if this is not called, the runtime gradually returns memory to the operating system in a background task.)

但在这个业务场景里，大量频繁的创建 PID，频繁的申请内存，创建对象，那么

申请速度远远大于释放速度，自然内存就一直堆积。

从文档中可以看出，FreeOSMemory 的方法可以将内存归还给操作系统，但我们并没有采用这种方案，因为它治标不治本，没法解决内存频繁申请频繁创建的问题，也不能降低 CPU 使用率。

为了解决这个问题，我们采用了 sync.Pool 的内置对象池方式，来复用回收对象，避免对象频繁创建，减少内存占用情况，在针对几个频繁创建的对象做对象池化后，同样的测试环境，内存稳定控制在 15M 左右。

大量对象的复用，也减少了对象的数量，同样的，在 Golang GC 运行时，也减少了对象的扫描数量、回收数量，降低了 CPU 使用率。

项目进展

在产品的研发过程中，也遇到了一些问题，比如：

1. etcd Client Lease Keepalive 的 Bug。
2. Agent 进程资源限制的 Cgroup 触发几次内核 Bug。
3. Docker 宿主机上瞬时大量进程创建的性能问题。
4. 网络监控模块在处理 Nginx 反向代理时，动辄几十万 TCP 链接的网络数据获取压力。
5. 个别进程打开了 10W 以上的 fd。

方法一定比困难多，但方法不是拍脑袋想出来的，一定要深入探索问题的根本原因，找到系统性的修复方法，具备高可用、高性能、监控告警、熔断限流等功能后，对于出现的问题，能够提前发现，将故障影响最小化，提前做处理。在应对产品运营过程中遇到的各种问题时，逢山开路，遇水搭桥，都可以从容的应对。

经过我们一年的努力，已经部署了除了个别特殊业务线之外的其他所有服务器，数量达几十万台，产品稳定运行。在数据完整性、准确性上，还有待提高，在精细化运营上，需要多做改进。

本篇更多的是研发角度上软件架构上的设计，关于安全事件分析、数据建模、运营策略等方面的经验和技巧，未来将会由其他同学进行分享，敬请期待。

总结

我们在研发这款产品过程中，也看到了网上开源了几款同类产品，也了解了他们的设计思路，发现很多产品都是把主要方向放在了单个模块的实现上，而忽略了产品架构上的重要性。

比如，有的产品使用了 syscall hook 这种侵入性高的方案来保障数据完整性，使得对系统侵入性非常高，Hook 代码的稳定性，也严重影响了操作系统内核的稳定。同时，Hook 代码也缺少了监控熔断的措施，在几十万服务器规模的场景下部署，潜在的风险可能让安全部门无法接受，甚至是致命的。

这种设计，可能在服务器量级小时，对于出现的问题多花点时间也能逐个进行维护，但应对几十万甚至上百万台服务器时，对维护成本、稳定性、监控熔断等都是很大的技术挑战。同时，在研发上，也很难实现产品的快速迭代，而这种方式带来的影响，几乎都会导致内核宕机之类致命问题。这种事故，使用服务器的业务方很难进行接受，势必会影响产品的研发速度、推进速度；影响同事（SRE 运维等）对产品的信心，进而对后续产品的推进带来很大的阻力。

以上是笔者站在研发角度，从可用性、可靠性、可控性、监控熔断等角度做的架构设计与框架设计，分享的产品研发思路。

笔者认为大规模的服务器安全防护产品，首先需要考虑的是架构的稳定性、监控告警的实时性、熔断限流的准确性等因素，其次再考虑安全数据的完整性、检测方案的可靠性、检测模型的精确性等因素。

九层之台，起于累土。只有打好基础，才能运筹帷幄，决胜千里之外。

参考资料

1. https://en.wikipedia.org/wiki/CAP_theorem
2. https://www.consul.io/intro/vs/serf.html
3. https://golang.org/src/runtime/debug/garbage.go?h=FreeOSMemory#L99
4. https://www.ibm.com/developerworks/cn/linux/l-connector/
5. https://www.kernel.org/doc/
6. https://coreos.com/etcd/docs/latest/

作者简介

陈驰，美团点评技术专家，2017 年加入美团，十年以上互联网产品研发经验，专注于分布式系统架构设计，目前主要从事安全防御产品研发工作。

关于美团安全

美团安全部的大多数核心人员，拥有多年互联网以及安全领域实践经验，很多同学参与过大型互联网公司的安全体系建设，其中也不乏全球化安全运营人才，具备百万级 IDC 规模攻防对抗的经验。安全部也不乏 CVE“挖掘圣手”，有受邀在 Black Hat 等国际顶级会议发言的讲者，当然还有很多漂亮的运营妹子。

目前，美团安全部涉及的技术包括渗透测试、Web 防护、二进制安全、内核安全、分布式开发、大数据分析、安全算法等等，同时还有全球合规与隐私保护等策略制定。我们正在建设一套百万级 IDC 规模、数十万终端接入的移动办公网络自适应安全体系，这套体系构建于零信任架构之上，横跨多种云基础设施，包括网络层、虚拟化 / 容器层、Server 软件层（内核态 / 用户态）、语言虚拟机层（JVM/JS V8）、Web 应用层、数据访问层等，并能够基于“大数据 + 机器学习”技术构建全自动的安全事件感知系统，努力打造成业界最前沿的内置式安全架构和纵深防御体系。

随着美团的高速发展，业务复杂度不断提升，安全部门面临更多的机遇和挑战。我们希望将更多代表业界最佳实践的安全项目落地，同时为更多的安全从业者提供一个广阔的发展平台，并提供更多在安全新兴领域不断探索的机会。

招聘信息

美团安全部正在招募 Web& 二进制攻防、后台 & 系统开发、机器学习 & 算法等各路小伙伴。

如果你想加入我们，欢迎简历请发至邮箱 zhaoyan17@meituan.com

具体职位信息可参考这里：https://mp.weixin.qq.com/s/ynEq5LqQ2uBcEaHCu7Tsiw

美团安全应急响应中心 MTSRC 主页：security.meituan.com

Leaf：美团分布式 ID 生成服务开源

志桐

Leaf 是美团基础研发平台推出的一个分布式 ID 生成服务，名字取自德国哲学家、数学家莱布尼茨的一句话："There are no two identical leaves in the world."Leaf 具备高可靠、低延迟、全局唯一等特点。目前已经广泛应用于美团金融、美团外卖、美团酒旅等多个部门。具体的技术细节，可参考此前美团技术博客的一篇文章：《Leaf 美团分布式 ID 生成服务》。近日，Leaf 项目已经在 Github 上开源：https://github.com/Meituan-Dianping/Leaf，希望能和更多的技术同行一起交流、共建。

Leaf 特性

Leaf 在设计之初就秉承着几点要求：

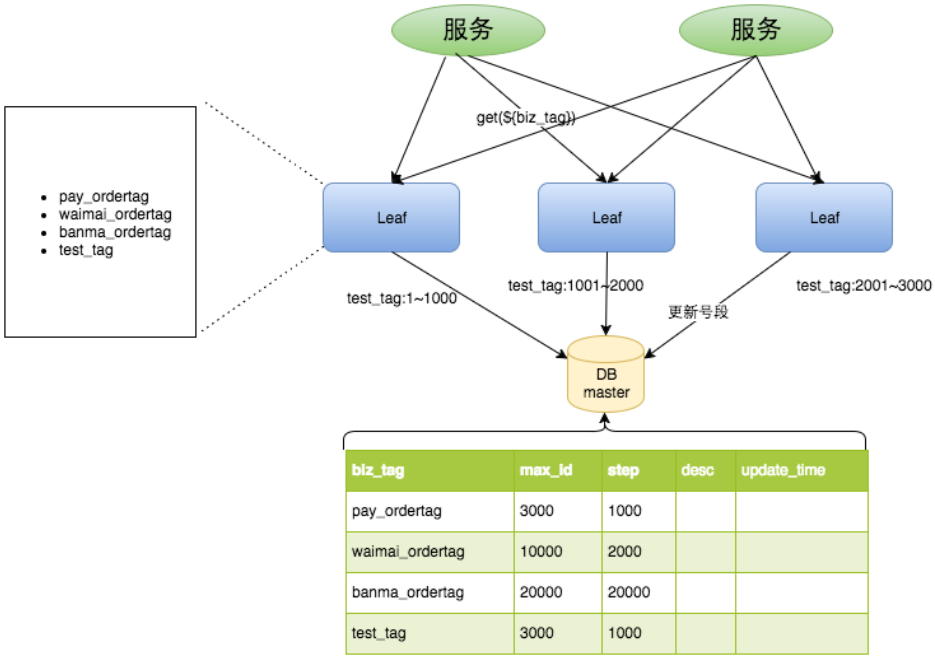
1. 全局唯一，绝对不会出现重复的 ID，且 ID 整体趋势递增。
2. 高可用，服务完全基于分布式架构，即使 MySQL 宕机，也能容忍一段时间的数据库不可用。
3. 高并发低延时，在 CentOS 4C8G 的虚拟机上，远程调用 QPS 可达 5W+，TP99 在 1ms 内。
4. 接入简单，直接通过公司 RPC 服务或者 HTTP 调用即可接入。

Leaf 诞生

Leaf 第一个版本采用了预分发的方式生成 ID，即可以在 DB 之上挂 N 个 Server，每个 Server 启动时，都会去 DB 拿固定长度的 ID List。这样就做到了完全基于分布式的架构，同时因为 ID 是由内存分发，所以也可以做到很高效。接下来是数据持久化问题，Leaf 每次去 DB 拿固定长度的 ID List，然后把最大的 ID 持久化下来，也就是并非每个 ID 都做持久化，仅仅持久化一批 ID 中最大的那一个。这个方式

有点像游戏里的定期存档功能，只不过存档的是未来某个时间下发给用户的 ID，这样极大地减轻了 DB 持久化的压力。

整个服务的具体处理过程如下：



biz_tag	max_id	step	desc	update_time
pay_ordertag	3000	1000		
waimai_ordertag	10000	2000		
banma_ordertag	20000	20000		
test_tag	3000	1000		

- Leaf Server 1：从 DB 加载号段 [1，1000]。
- Leaf Server 2：从 DB 加载号段 [1001，2000]。
- Leaf Server 3：从 DB 加载号段 [2001，3000]。

用户通过 Round-robin 的方式调用 Leaf Server 的各个服务，所以某一个 Client 获取到的 ID 序列可能是：1，1001，2001，2，1002，2002……也可能是：1，2，1001，2001，2002，2003，3，4……当某个 Leaf Server 号段用完之后，下一次请求就会从 DB 中加载新的号段，这样保证了每次加载的号段是递增的。

Leaf 数据库中的号段表格式如下：

```
+-------------+--------------+------+-----+--------------------+------
-----------------------+
| Field       | Type         | Null | Key | Default            | Extra
|
```

```
+-------------+--------------+------+-----+-------------------+------
----------------------+
| biz_tag     | varchar(128) | NO   | PRI |                   |
|
| max_id      | bigint(20)   | NO   |     | 1                 |
|
| step        | int(11)      | NO   |     | NULL              |
|
| desc        | varchar(256) | YES  |     | NULL              |
|
| update_time | timestamp    | NO   |     | CURRENT_TIMESTAMP | on update
CURRENT_
TIMESTAMP |
+-------------+--------------+------+-----+-------------------+------
----------------------+
```

Leaf Server 加载号段的 SQL 语句如下：

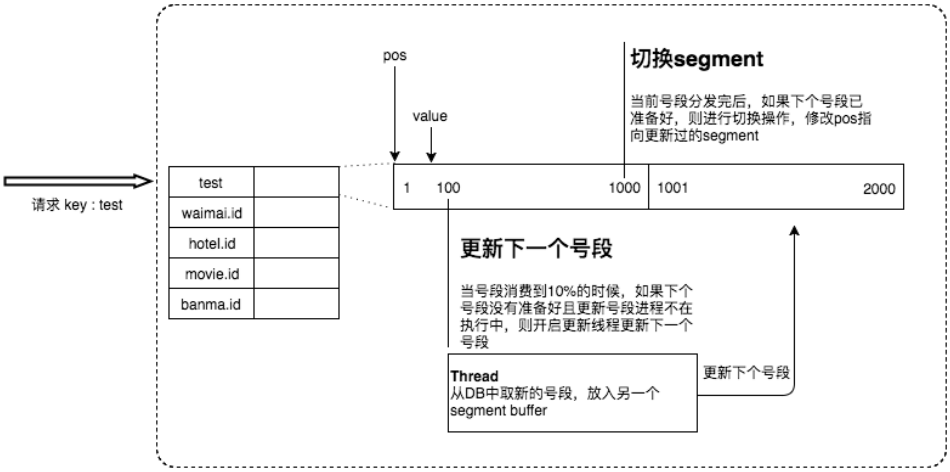
```
Begin
UPDATE table SET max_id=max_id+step WHERE biz_tag=xxx
SELECT tag, max_id, step FROM table WHERE biz_tag=xxx
Commit
```

整体上，V1 版本实现比较简单，主要是为了尽快解决业务层 DB 压力的问题，而快速迭代出的一个版本。因而在生产环境中，也发现了些问题。比如：

1. 在更新 DB 的时候会出现耗时尖刺，系统最大耗时取决于更新 DB 号段的时间。
2. 当更新 DB 号段的时候，如果 DB 宕机或者发生主从切换，会导致一段时间的服务不可用。

Leaf 双 Buffer 优化

为了解决这两个问题，Leaf 采用了异步更新的策略，同时通过双 Buffer 的方式，保证无论何时 DB 出现问题，都能有一个 Buffer 的号段可以正常对外提供服务，只要 DB 在一个 Buffer 的下发的周期内恢复，就不会影响整个 Leaf 的可用性。



这个版本代码在线上稳定运行了半年左右，Leaf 又遇到了新的问题：

1. 号段长度始终是固定的，假如 Leaf 本来能在 DB 不可用的情况下，维持 10 分钟正常工作，那么如果流量增加 10 倍就只能维持 1 分钟正常工作了。
2. 号段长度设置的过长，导致缓存中的号段迟迟消耗不完，进而导致更新 DB 的新号段与前一次下发的号段 ID 跨度过大。

Leaf 动态调整 Step

假设**服务 QPS 为 Q，号段长度为 L，号段更新周期为 T**，那么 **Q * T = L**。最开始 L 长度是固定的，导致随着 Q 的增长，T 会越来越小。但是 Leaf 本质的需求是**希望 T 是固定的**。那么如果 L 可以和 Q 正相关的话，T 就可以趋近一个定值了。所以 Leaf 每次更新号段的时候，根据上一次更新号段的周期 T 和号段长度 step，来决定下一次的号段长度 nextStep：

- T < 15min，nextStep = step * 2
- 15min < T < 30min，nextStep = step
- T > 30min，nextStep = step / 2

至此，满足了号段消耗稳定趋于某个时间区间的需求。当然，面对瞬时流量几

十、几百倍的暴增，该种方案仍不能满足可以容忍数据库在一段时间不可用、系统仍能稳定运行的需求。因为本质上来讲，Leaf 虽然在 DB 层做了些容错方案，但是号段方式的 ID 下发，最终还是需要强依赖 DB。

MySQL 高可用

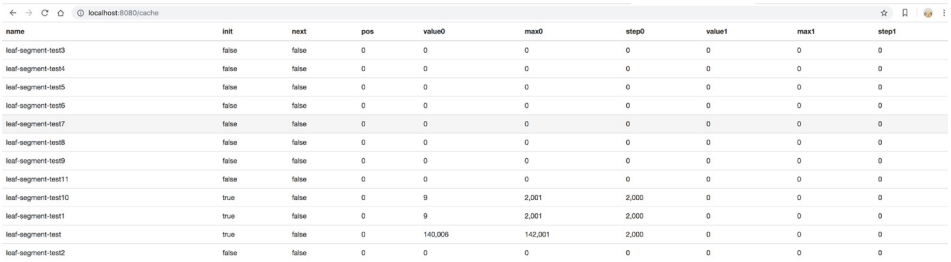
在 MySQL 这一层，Leaf 目前采取了半同步的方式同步数据，通过公司 DB 中间件 Zebra 加 MHA 做的主从切换。未来追求完全的强一致，会考虑切换到 MySQL Group Replication。

现阶段由于公司数据库强一致的特性还在演进中，Leaf 采用了一个临时方案来保证机房断网场景下的数据一致性：

- 多机房部署数据库，每个机房一个实例，保证都是跨机房同步数据。
- 半同步超时时间设置到无限大，防止半同步方式退化为异步复制。

Leaf 监控

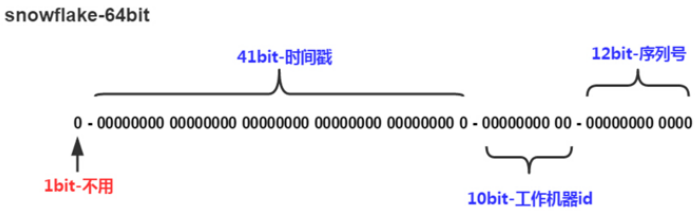
针对服务自身的监控，Leaf 提供了 Web 层的内存数据映射界面，可以实时看到所有号段的下发状态。比如每个号段双 buffer 的使用情况，当前 ID 下发到了哪个位置等信息都可以在 Web 界面上查看。



name	init	next	pos	value0	max0	step0	value1	max1	step1
leaf-segment-test3	false	false	0	0	0	0	0	0	0
leaf-segment-test4	false	false	0	0	0	0	0	0	0
leaf-segment-test5	false	false	0	0	0	0	0	0	0
leaf-segment-test6	false	false	0	0	0	0	0	0	0
leaf-segment-test7	false	false	0	0	0	0	0	0	0
leaf-segment-test8	false	false	0	0	0	0	0	0	0
leaf-segment-test9	false	false	0	0	0	0	0	0	0
leaf-segment-test11	false	false	0	0	0	0	0	0	0
leaf-segment-test10	true	false	0	9	2,001	2,000	0	0	0
leaf-segment-test1	true	false	0	9	2,001	2,000	0	0	0
leaf-segment-test	true	false	0	140,006	142,001	2,000	0	0	0
leaf-segment-test2	false	false	0	0	0	0	0	0	0

Leaf Snowflake

Snowflake，Twitter 开源的一种分布式 ID 生成算法。基于 64 位数实现，下图为 Snowflake 算法的 ID 构成图。



- 第 1 位置为 0。
- 第 2–42 位是相对时间戳，通过当前时间戳减去一个固定的历史时间戳生成。
- 第 43–52 位是机器号 workerID，每个 Server 的机器 ID 不同。
- 第 53–64 位是自增 ID。

这样通过时间 + 机器号 + 自增 ID 的组合来实现了完全分布式的 ID 下发。

在这里，Leaf 提供了 Java 版本的实现，同时对 Zookeeper 生成机器号做了弱依赖处理，即使 Zookeeper 有问题，也不会影响服务。Leaf 在第一次从 Zookeeper 拿取 workerID 后，会在本机文件系统上缓存一个 workerID 文件。即使 ZooKeeper 出现问题，同时恰好机器也在重启，也能保证服务的正常运行。这样做到了对第三方组件的弱依赖，一定程度上提高了 SLA。

未来规划

- 号段加载优化：Leaf 目前重启后的第一次请求还是会同步加载 MySQL，之所以这么做而非服务初始化加载号段的原因，主要是 MySQL 中的 Leaf Key 并非一定都被这个 Leaf 服务节点所加载，如果每个 Leaf 节点都在初始化加载所有的 Leaf Key 会导致号段的大量浪费。因此，未来会在 Leaf 服务 Shutdown 时，备份这个服务节点近一天使用过的 Leaf Key 列表，这样重启后会预先从 MySQL 加载 Key List 中的号段。
- 单调递增：简易的方式，是只要保证同一时间、同一个 Leaf Key 都从一个 Leaf 服务节点获取 ID，即可保证递增。需要注意的问题是 Leaf 服务节点切换时，旧 Leaf 服务用过的号段需要废弃。路由逻辑，可采用主备的模型或者每个 Leaf Key 配置路由表的方式来实现。

关于开源

分布式 ID 生成的方案有很多种，Leaf 开源版本提供了两种 ID 的生成方式：

- 号段模式：低位趋势增长，较少的 ID 号段浪费，能够容忍 MySQL 的短时间不可用。
- Snowflake 模式：完全分布式，ID 有语义。

读者可以按需选择适合自身业务场景的 ID 下发方式。希望美团的方案能给予大家一些帮助，同时也希望各位能够一起交流、共建。

Leaf 项目 Github 地址：https://github.com/Meituan-Dianping/Leaf。

如有任何疑问和问题，欢迎提交至 Github issues。

美团大规模微服务通信框架及治理体系 OCTO 核心组件开源

舒超　张翔

微服务通信框架及治理平台 OCTO 作为美团基础架构设施的重要组成部分，目前已广泛应用于公司技术线，稳定承载上万应用、日均支撑千亿级的调用。业务基于 OCTO 提供的标准化技术方案，能够轻松实现服务注册 / 发现、负载均衡、容错处理、降级熔断、灰度发布、调用数据可视化等服务治理功能。

现在我们将 OCTO 的核心组件 OCTO-RPC、OCTO-NS、OCTO-Portal 开源，欢迎大家使用和共建。OCTO-RPC、OCTO-NS、OCTO-Portal 深入了解。

背景

OCTO 项目始于 2014 年底，当时美团正处在新业务拓展期，服务数量不断增长，服务间调用拓扑日益复杂，逐渐暴露出了一些典型的问题：

- 研发效率低：缺乏标准的服务治理框架和组件，不少团队“重复造轮子”造成资源浪费，研发质量参差不齐，系统整体可用性不高。
- 运维成本高：缺乏完善、便捷的体系化运维手段，难以进行准确的监控告警，难以对涉及多级链路的故障快速定位。
- 服务运营难：服务指标数据收集困难，缺乏自动化深度分析，难以满足业务快速对指标进行评估决策的要求。

针对这些痛点问题，提升研发效率和质量，降低运营成本，对于支撑业务快速稳定发展显得尤为重要。因此，基础架构团队研发了公司级统一的分布式微服务通信框架及治理平台——OCTO。OCTO 是英文单词章鱼（Octopus）的缩写，章鱼众多的触手表征 OCTO 平台能触达治理海量的服务，涵盖服务治理领域的各个部分，因此取名。自平台推出以来，在全公司各业务线被广泛进行使用，显著提升了全公司的技术研发效率与运营质量，稳定支撑着美团业务的高速发展。

有别于传统的开源服务治理组件，OCTO 提供了体系化的服务治理方案，从通信框架到注册中心、从文件配置到埋点告警、从灰度链路到数据报表，立体化的提升微服务运营能力、赋能业务。此外，奉行“策略下沉”的设计思想，OCTO 剥离传统通信框架具备的服务治理策略功能，下沉到代理组件实现，有效提升通信框架的稳定性、降低业务系统额外开销。历经海量调用验证的 OCTO，依托低耦合、模块化、单元化系统架构，能够有效满足各类复杂业务场景需求，实现异地多活等容灾目标。

功能特性

针对暴露出的问题，OCTO 围绕包括定义、开发、测试、部署、运维、优化、下线在内的服务的全生命周期，打造了一系列组件和系统，方便研发同学专注于自身业务逻辑开发的同时，又能享受完善的服务治理功能。例如，我们在开发阶段提供了高性能、高可用、功能模块化的通信框架供业务便捷使用；在测试阶段提供了 SET 化、泳道、服务分组等各种灰度策略供业务无损试错；在运维阶段提供机器级、框架级、业务级等层级分明的监控告警供业务快速定位问题；在优化阶段提供了详尽的服务指标供业务自定义分析改进。

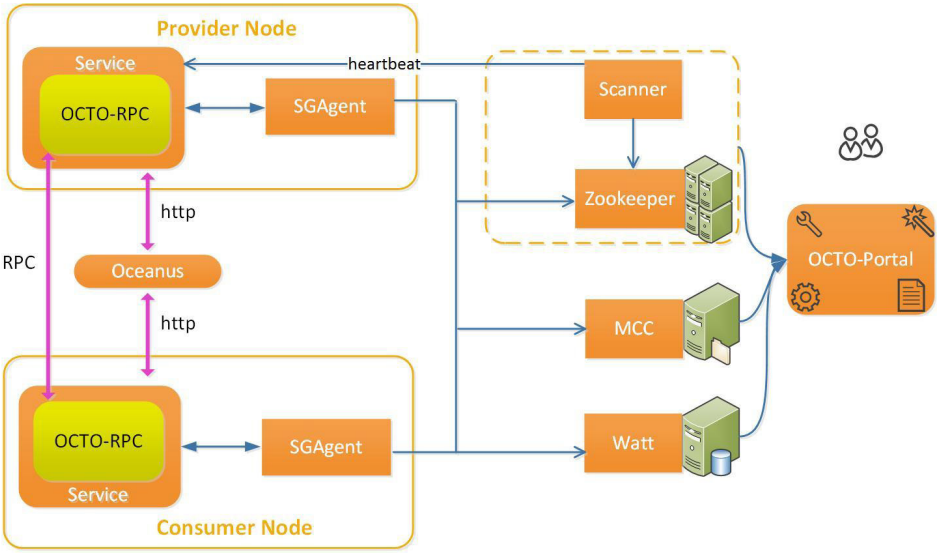
OCTO 主要功能特性包括但不限于：

- 命名服务：服务注册 / 发现。
- 服务管理：服务状态监测；服务启动、停止；服务负载均衡。
- 容错处理：实时屏蔽异常的服务，自动调配请求流量。
- 流量分发：灰度发布、节点动态流量分配等场景。
- 数据可视化：服务调用统计上报分析，提供清晰的数据图表展示，直观定位服务间依赖关系。
- 服务分组：支持服务动态自动归组与不同场景下的自定义分组，解决在多机房场景下跨机房调用穿透、沙盒测试等问题。
- 服务监控报警：支持服务与接口级别多指标、多维度的监控，支持多种报警方式。
- 统一配置管理：支持服务配置统一管理，灵活设置不同环境间差异，支持历史

版本，配置项变更后实时下发。

- 分布式服务跟踪：轻松诊断服务访问慢、异常抖动等问题。
- 过载保护：灵活定义分配给上游服务消费者的配额，当服务调用量超出最大阈值时，基于不同服务消费者进行 QoS 区分，触发流控进行过载保护。
- 服务访问控制：支持多粒度、多阶段的鉴权方式。

OCTO 整体架构



OCTO 架构图

- OCTO-RPC（开源 Java/C++）：分布式 RPC 通信框架，支持 Java、C++、Node.js 等多种语言。
- SGAgent：部署在各服务节点，承担服务注册 / 发现、动态路由解析、负载均衡、配置传输、性能数据上报等功能。
- Oceanus（待开源）：HTTP 定制化路由负载器，具体可参考 Oceanus：美团 HTTP 流量定制化路由的实践一文。
- OCTO-NS：包括 SDK（Java/C++）、基础代理 SGAgent、命名服务缓存 NSC、健康检查服务 Scanner 等组件，提供命名服务，服务注册等信息的存

储，服务状态检测的扫描等功能。

- Watt（待开源）：统计链路信息，计算服务各类指标作为监控报警依据。
- MCC（待开源）：统一配置中心，可进行服务配置的管理下发。
- OCTO-Portal：服务注册、管理、诊断、配置、配额等功能的一站式管理平台。

OCTO 开源组件

OCTO 首批开源的核心组件包括：分布式服务通信框架（OCTO-RPC）、服务注册中心（OCTO-NS）、服务治理平台（OCTO-Portal）。未来，我们还将持续开源更多的组件与功能。

分布式服务通信框架（OCTO-RPC）

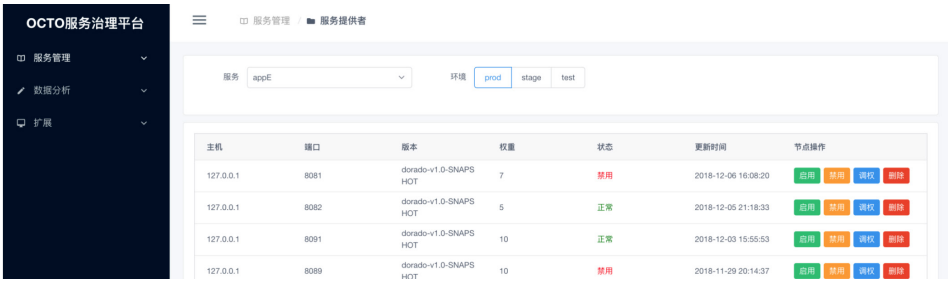
分布式服务通信框架是 OCTO 的重要组成部分，具备高性能、高可用、易扩展、易接入等特点，已覆盖美团 90% 以上的服务（Java/C++），支撑每天千亿级别的调用量。目前 Java 和 C++ 版本已经开源，更多技术实现详见：OCTO-RPC。

服务注册中心（OCTO-NS）

服务注册中心基于服务描述信息，实现服务注册 / 发现、配置管理、路由分组、负载均衡、健康检测等功能，搭配服务治理平台能够更便捷地进行服务节点数据的可视化运营。目前开源出来的子模块包括 SDK（Java/C++）、基础代理 SGAgent、命名服务缓存 NSC、健康检查服务 Scanner。更多技术实现详见：OCTO-NS。

服务治理平台（OCTO-Portal）

服务治理的一站式平台，为服务关注方提供服务节点管理、性能数据分析、全链路跟踪诊断等服务治理核心能力。更多技术实现详见：OCTO-Portal。



OCTO-Portal 示意图

关于开源

在过去四年中，OCTO 是美团在架构中间件领域研发的一个重要技术项目，在复杂多元业务、大规模并发调用的场景下已被充分验证。目前 OCTO 支撑了美团大规模微服务每天千亿级的调用，接口调用成功率达到了 99.999%，是全公司 SOA、服务治理的核心基础。我们期望通过将其开源，不断反馈给社区、贡献给行业。同时，我们希望在行业优秀工程师的帮助下，OCTO 平台能得到更快地升级更新迭代。欢迎大家多提宝贵意见和建议。

未来规划

为了进一步支撑美团业务飞速发展的需求，同时对标业界先进的服务治理理念与实践，未来一段时间内，OCTO 将在以下几方面规划演进：

1. **命名服务 AP 化**：弱化强一致性，聚焦异常状态下注册中心的可用性。
2. **框架反应式编程：**RPC 框架提供反应式编程支持，帮助业务构建高性能异步无阻塞的服务。
3. **Service Mesh：**将现有命名服务等功能与控制面 / 数据面结合，以服务网格的思路进一步演进 OCTO 服务治理体系。

作者简介

舒超，2015 年加入美团，高级技术专家，基础开发负责人。

张翔，2017 年加入美团，现负责公司命名服务和通信框架的研发。

招聘信息

美团 OCTO 服务治理团队诚招 C++/Java 高级工程师、技术专家。我们致力于研发公司级、业界领先的基础架构组件，研发范围涵盖分布式框架、命名服务、Service Mesh 等技术领域。欢迎有兴趣的同学投送简历至 tech@meituan.com。

美团下一代服务治理系统 OCTO2.0 的探索与实践

郭继东

本文根据美团基础架构部服务治理团队工程师郭继东在 2019 QCon（全球软件开发大会）上的演讲内容整理而成，主要阐述美团大规模治理体系结合 Service Mesh 演进的探索实践，希望对从事此领域的同学有所帮助。

一、OCTO 现状分析

OCTO 是美团标准化的服务治理基础设施，治理能力统一、性能及易用性表现优异、治理能力生态丰富，已广泛应用于美团各事业线。关于 OCTO 的现状，可整体概括为：

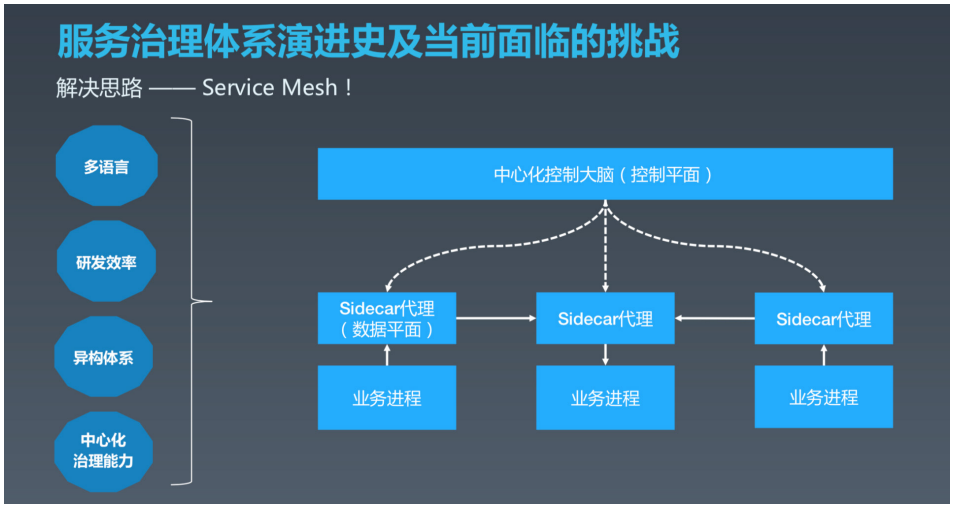
- 已成为美团高度统一的服务治理技术栈，覆盖了公司 90% 的应用，日均调用超万亿次。
- 经历过大规模的技术考验，覆盖数万个服务 / 数十万个节点。
- 协同周边治理生态提供的治理能力较为丰富，包含但不限于 SET 化、链路级复杂路由、全链路压测、鉴权加密、限流熔断等治理能力。
- 一套系统支撑着多元业务，覆盖公司所有事业线。

目前美团已经具备了相对完善的治理体系，但仍有较多的痛点及挑战：

- 对多语言支持不够好。美团技术栈使用的语言主要是 Java，占比到达 80% 以上，上面介绍的诸多治理能力也集中在 Java 体系。 但美团同时还有其他近 10 种后台服务语言在使用，这些语言的治理生态均十分薄弱，同时在多元业务的模式下必然会有增长的多语言需求，为每一种语言都建设一套完善的治理体系成本很高，也不太可能落地。
- 中间件和业务绑定在一起，制约着彼此迭代。一般来说，核心的治理能力主要由通信框架承载，虽然做到了逻辑隔离，但中间件的逻辑不可避免会和业务

在物理上耦合在一起。这种模式下，中间件引入 Bug 需要所有业务配合升级，这对业务的研发效率也会造成损害；新特性的发布也依赖业务逐个升级，不具备自主的控制能力。

- 异构治理体系技术融合成本很高。
- 治理决策比较分散。每个节点只能根据自己的状态进行决策，无法与其他节点协同仲裁。



针对以上痛点，我们考虑依托于 Service Mesh 解决。Service Mesh 模式下会为每个业务实例部署一个 Sidecar 代理，所有进出应用的业务流量统一由 Sidecar 承载，同时服务治理的工作也主要由 Sidecar 执行，而所有的 Sidecar 由统一的中心化控制大脑控制面来进行全局管控。这种模式如何解决上述四个问题的呢?

- Service Mesh 模式下，各语言的通信框架一般仅负责编解码，而编解码的逻辑往往是不变的。核心的治理功能（如路由、限流等）主要由 Sidecar 代理和控制大脑协同完成，从而实现一套治理体系，所有语言通用。
- 中间件易变的逻辑尽量下沉到 Sidecar 和控制大脑中，后续升级中间件基本不需要业务配合。SDK 主要包含很轻薄且不易变的逻辑，从而实现了业务和中间件的解耦。
- 新融入的异构技术体系可以通过轻薄的 SDK 接入美团治理体系（技术体系难

兼容，本质是它们各自有独立的运行规范，在 Service Mesh 模式下运行规范核心内容就是控制面和 Sidecar），目前美团线上也有这样的案例。

- 控制大脑集中掌控了所有节点的信息，进而可以做一些全局最优的决策，比如服务预热、根据负载动态调整路由等能力。

总结一下，在当前治理体系进行 Mesh 化改造可以进一步提升治理能力，美团也将 Mesh 化改造后的 OCTO 定义为下一代服务治理系统 OCTO2.0（内部名字是 OCTO Mesh）。

二、技术选型及架构设计

2.1 OCTO Mesh 技术选型

美团的 Service Mesh 建设起步于 2018 年底，当时所面临一个核心问题是整体方案最关键的考量应该关注哪几个方面。在启动设计阶段时，我们有一个非常明确的意识：在大规模、同时治理能力丰富的前提下进行 Mesh 改造需要考虑的问题，与治理体系相对薄弱且期望依托于 Service Mesh 丰富治理能力的考量点，还是有非常大的差异的。总结下来，技术选型需要重点关注以下四个方面：

- OCTO 体系已经历近 5 年的迭代，形成了一系列的标准与规范，进行 Service Mesh 改造治理体系架构的升级范围会很大，在确保技术方案可以落地的同时，也要屏蔽技术升级或只需要业务做很低成本的改动。
- 治理能力不能减弱，在保证对齐的基础上逐渐提供更精细化、更易用的运营能力。
- 能应对超大规模的挑战，技术方案务必能确保支撑当前量级甚至当前 N 倍的增量，系统自身也不能成为整个治理体系的瓶颈。
- 尽量与社区保持亲和，一定程度上与社区协同演进。

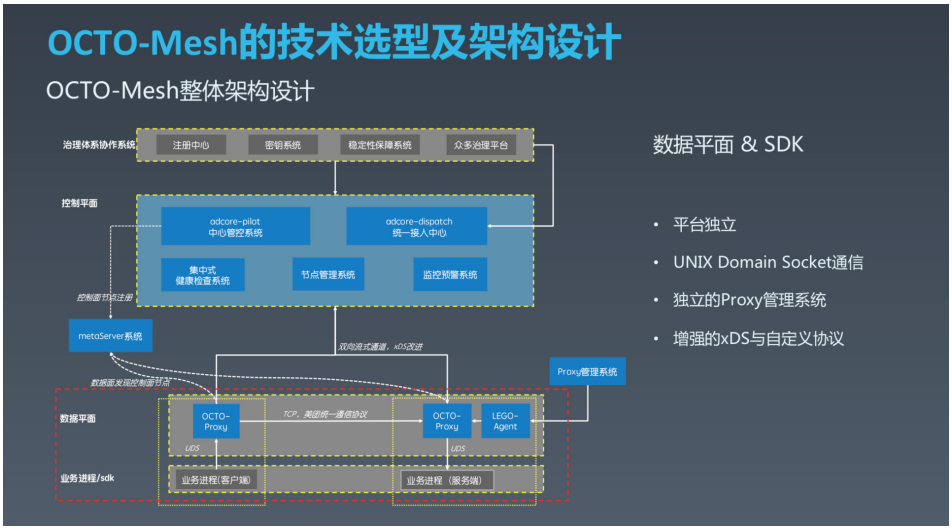
针对上述考量，我们选择的方式是数据面基于 Envoy 二次开发，控制面自研为主。

数据面方面，当时 Envoy 有机会成为数据面的事实标准，同时 Filter 模式及 xDS 的设计对扩展比较友好，未来功能的丰富、性能优化也与标准关系较弱。

控制面自研为主的决策需要考量的内容就比较复杂了，总体而言需要考虑如下几个方面：

- 截止发稿前，美团容器化主要采用富容器的模式，这种模式下强行与 Istio 及 Kubernetes 的数据模型匹配改造成本极高，同时 Istio API 也尚未确定。
- 截止发稿前，Istio 在集群规模变大时较容易出现性能问题，无法支撑美团数万应用、数十万节点的的体量，同时数十万节点规模的 Kubernetes 集群也需要持续优化探索。
- Istio 的功能无法满足 OCTO 复杂精细的治理需求，如流量录制回放压测、更复杂的路由策略等。
- 项目启动时非容器应用占比较高，技术方案需要兼容存量非容器应用。

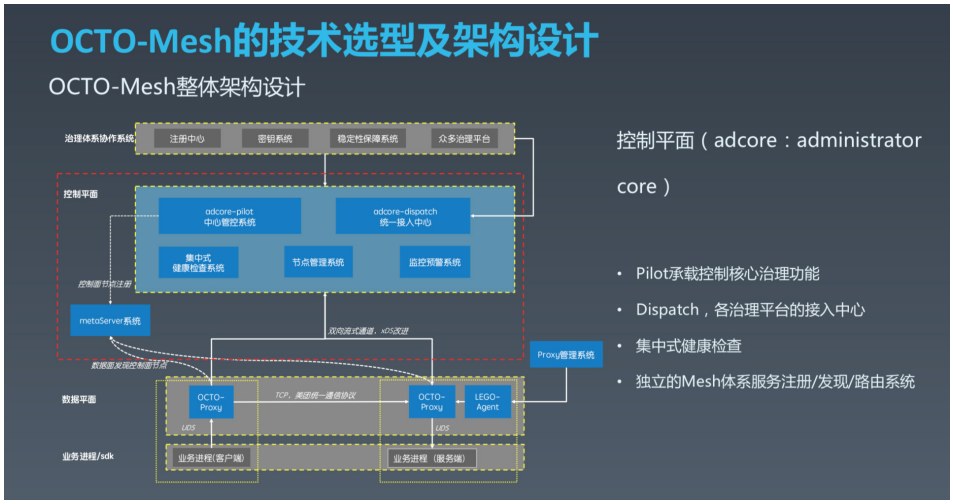
2.2 OCTO Mesh 架构设计



上面这张图展示了 OCTO Mesh 的整体架构。从下至上来看，逻辑上分为业务进程及通信框架 SDK 层、数据平面层、控制平面层、治理体系协作的所有周边生态层。

- 先来重点介绍下业务进程及 SDK 层、数据平面层：
- OCTO Proxy（数据面 Sidecar 代理内部叫 OCTO Proxy）与业务进程采用 1 对 1 的方式部署。
- OCTO Proxy 与业务进程采用 UNIX Domain Socket 做进程间通信（这里没有选择使用 Istio 默认的 iptables 流量劫持，主要考虑美团内部基本是使用的统一化私有协议通信，富容器模式没有用 Kubernetes 的命名服务模型，iptables 管理起来会很复杂，而 iptables 复杂后性能会出现较高的损耗。）；OCTO Proxy 间跨节点采用 TCP 通信，采用和进程间同样的协议，保证了客户端和服务端具备独立升级的能力。
- 为了提升效率同时减少人为错误，我们独立建设了 OCTO Proxy 管理系统，部署在每个实例上的 LEGO Agent 负责 OCTO Proxy 的保活和热升级，类似于 Istio 的 Pilot Agent，这种方式可以将人工干预降到较低，提升运维效率。

- 数据面与控制面通过双向流式通信。路由部分交互方式是增强语义的 xDS，增强语义是因为当前的 xDS 无法满足美团更复杂的路由需求；除路由外，该通道承载着众多的治理功能的指令及配置下发，我们设计了一系列的自定义协议。



控制面（内部名字是 Adcore）自研为主，整体分为：Adcore Pilot、Adcore Dispatcher、集中式健康检查系统、节点管理模块、监控预警模块。此外独立建设了统一元数据管理及 Mesh 体系内的服务注册发现系统 Meta Server 模块。每个模块的具体职责如下：

- Adcore Pilot 是个独立集群，模块承载着大部分核心治理功能的管控，相当于整个系统的大脑，也是直接与数据面交互的模块。
- Adcore Dispatcher 也是独立集群，该模块是供治理体系协作的众多子系统便捷接入 Mesh 体系的接入中心。
- 不同于 Envoy 的 P2P 节点健康检查模式，OCTO Mesh 体系使用的是集中式健康检查。
- 控制面会节点管理系统负责采集每个节点的运行时信息，并根据节点的状态做全局性的最优治理的决策和执行。
- 监控预警系统是保障 Mesh 自身稳定性而建设的模块，实现了自身的可观测性，当出现故障时能快速定位，同时也会对整个系统做实时巡检。

- 与 Istio 基于 Kubernetes 来做寻址和元数据管理不同，OCTO Mesh 由独立的 Meta Server 负责 Mesh 自身众多元信息的管理和命名服务。

三、关键设计解析

大规模治理体系 Mesh 化建设成功落地的关键点有：

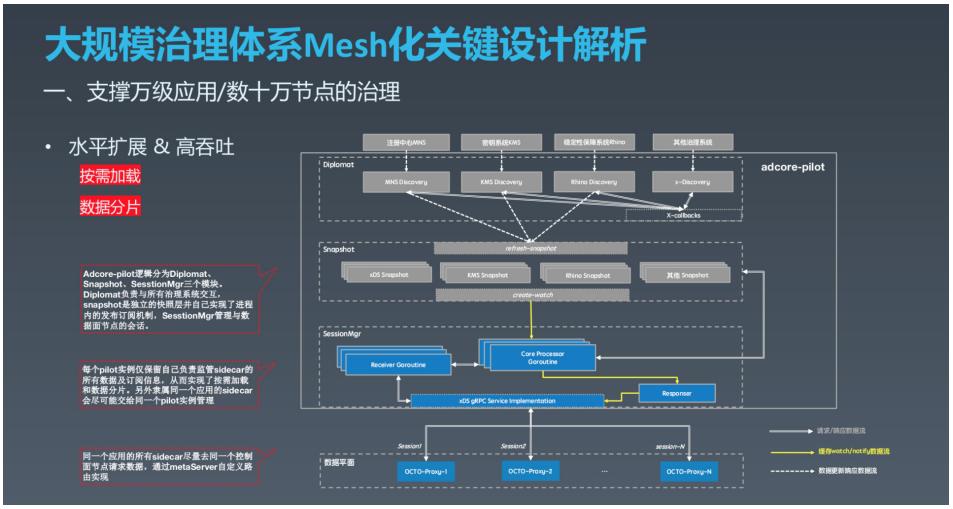
- 系统水平扩展能力方面，可以支撑数万应用 / 百万级节点的治理。
- 功能扩展性方面，可以支持各类异构治理子系统融合打通。
- 能应对 Mesh 化改造后链路复杂的可用性、可靠性要求。
- 具备成熟完善的 Mesh 运维体系。

围绕这四点，便可以在系统能力、治理能力、稳定性、运营效率方面支撑美团当前多倍体量的新架构落地。

3.1 大规模系统 Mesh 化系统能力建设

对于社区 Istio 方案，要想实现超大规模应用集群落地，需要完成较多的技术改造。主要是因为 Istio 水平扩展能力相对薄弱，内部冗余操作较多，整体稳定性建设较为薄弱。针对上述问题，我们的解决思路如下：

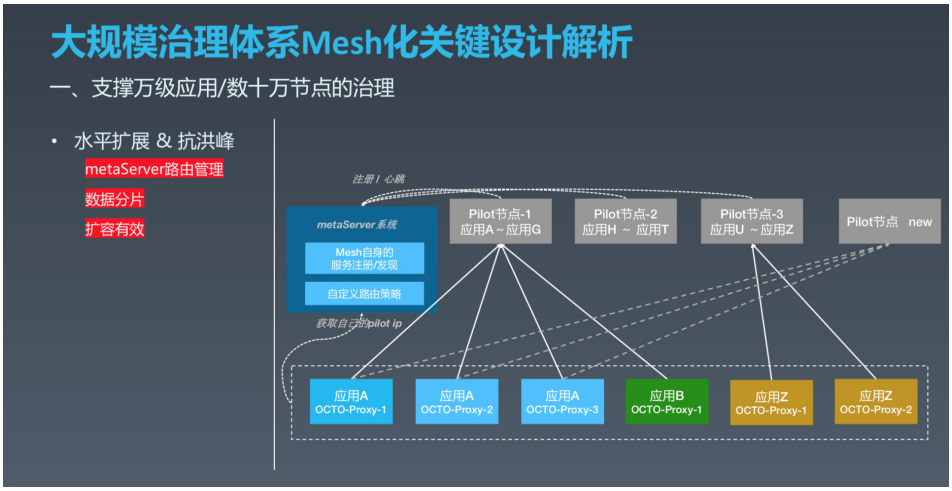
- 控制面每个节点并不承载所有治理数据，系统整体做水平扩展，在此基础上提升每个实例的整体吞吐量和性能。
- 当出现机房断网等异常情况时，可以应对瞬时流量骤增的能力。
- 只做必要的 P2P 模式健康检查，配合集中式健康检查进行百万级节点管理。



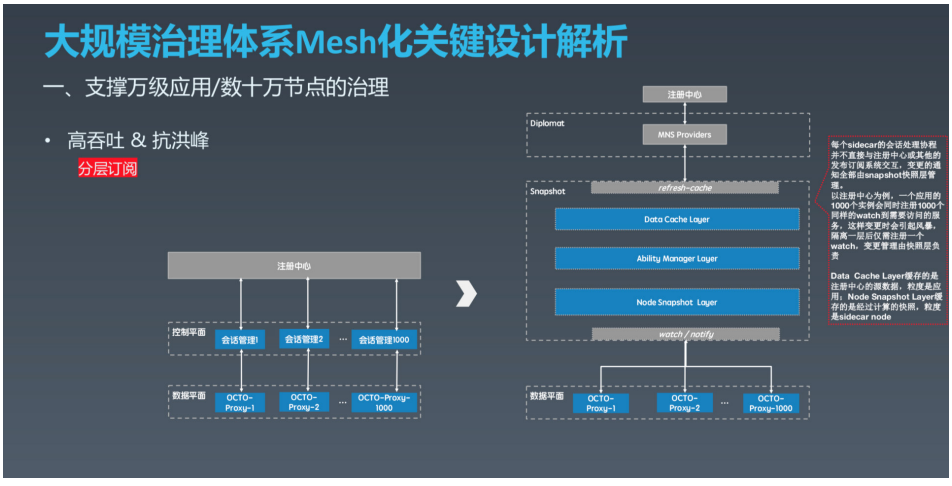
按需加载和数据分片主要由 Adcore Pilot 配合 Meta Server 实现。

Pilot 的逻辑架构分为 SessionMgr、Snapshot、Diplomat 三个部分，其中 SessionMgr 管理每个数据面会话的全生命周期、会话的创建、交互及销毁等一系列动作及流程；Snapshot 维护数据最新的一致性快照，对下将资源的更新同步给 SessionMgr 处理，对上响应各平台的数据变更通知，进行计算并将存在关联关系的一组数据做快照缓存。Diplomat 模块负责与服务治理系统的众多平台对接，只有该模块会与第三方平台直接产生依赖。

控制面每个 Pilot 节点并不会把整个注册中心及其他数据都加载进来，而是按需加载自己管控的 Sidecar 所需要的相关治理数据，即从 SessionMgr 请求的应用所负责的相关治理数据，以及该应用关注的对端服务注册信息。另外同一个应用的所有 OCTO Proxy 应该由同一个 Pilot 实例管控，否则全局状态下又容易趋近于全量了。具体是怎么实现的呢？答案是 Meta Server，自己实现控制面机器服务发现的同时精细化控制路由规则，从而在应用层面实现了数据分片。



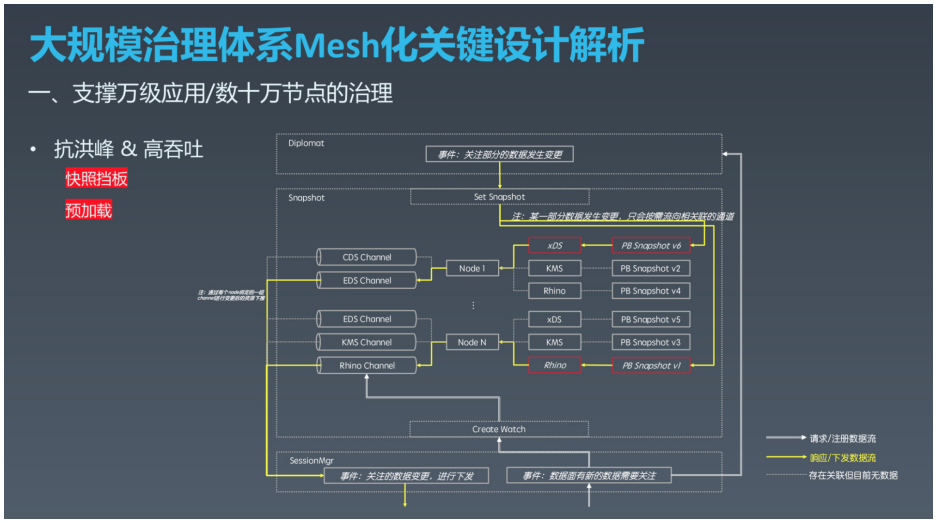
Meta Server 管控每个 Pilot 节点负责应用 OCTO Proxy 的归属关系。当 Pilot 实例启动会注册到 Meta Server，此后定时发送心跳进行续租，长时间心跳异常会自动剔除。在 Meta Server 内部实现了较为复杂的一致性哈希策略，会综合节点的应用、机房、负载等信息进行分组。当一个 Pilot 节点异常或发布时，隶属该 Pilot 的 OCTO Proxy 都会有规律的连接到接替节点，而不会全局随机连接对后端注册中心造成风暴。当异常或发布后的节点恢复后，划分出去的 OCTO Proxy 又会有规则的重新归属当前 Pilot 实例管理。对于关注节点特别多的应用 OCTO Proxy，也可以独立部署 Pilot，通过 Meta Server 统一进行路由管理。



Mesh 体系的命名服务需要 Pilot 与注册中心打通，常规的实现方式如左图所示（以 Zookeeper 为例），每个 OCTO Proxy 与 Pilot 建立会话时，作为客户端角色会向注册中心订阅自身所关注的服务端变更监听器，假设这个服务需要访问 100 个应用，则至少需要注册 100 个 Watcher 。假设该应用存在 1000 个实例同时运行，就会注册 100*1000 = 100000 个 Watcher，超过 1000 个节点的应用在美团内部还是蛮多的。另外还有很多应用关注的对端节点相同，会造成大量的冗余监听。当规模较大后，网络抖动或业务集中发布时，很容易引发风暴效应把控制面和后端的注册中心打挂。

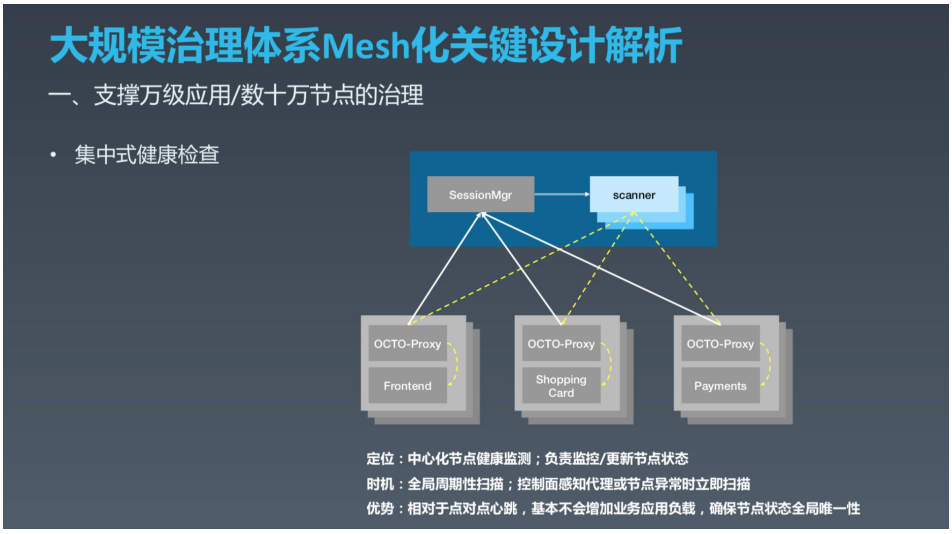
针对这个问题，我们采用分层订阅的方案解决。每个 OCTO Proxy 的会话并不直接与注册中心或其他的发布订阅系统交互，变更的通知全部由 Snapshot 快照层管理。Snapshot 内部又划分为 3 层，Data Cache 层对接并缓存注册中心及其他系统的原始数据，粒度是应用；Node Snapshot 层则是保留经过计算的节点粒度的数据；Ability Manager 层内部会做索引和映射的管理，当注册中心存在节点状态变更时，会通过索引将变更推送给关注变更的 OCTO Proxy。

对于刚刚提到的场景，隔离一层后 1000 个节点仅需注册 100 个 Watcher，一个 Watcher 变更后仅会有一条变更信息到 Data Cache 层，再根据索引向 1000 个 OCTO Proxy 通知，从而极大的降低了注册中心及 Pilot 的负载。



Snapshot 层除了减少不必要交互提升性能外，也会将计算后的数据格式化缓存下来，一方面瞬时大量相同的请求会在快照层被缓存挡住，另一方面也便于将存在关联的数据统一打包到一起，避免并发问题。这里参考了 Envoy-Control-Plane 的设计，Envoy-Control-Plane 会将包含 xDS 的所有数据全部打包在一起，而我们是将数据隔离开，如路由、鉴权完全独立，当路由数据变更时不会去拉取并更新鉴权信息。

预加载主要目的是提升服务冷启动性能，Meta Server 的路由规则由我们制定，所以这里提前在 Pilot 节点中加载好最新的数据，当业务进程启动时，Proxy 就可以立即从 Snapshot 中获取到数据，避免了首次访问慢的问题。



Istio 默认每个 Envoy 代理对整个集群中所有其余 Envoy 进行 P2P 健康检测，当集群有 N 个节点时，一个检测周期内（往往不会很长）就需要做 N 的平方次检测，另外当集群规模变大时所有节点的负载就会相应提高，这都将成为扩展部署的极大障碍。

不同于全集群扫描，美团采用了集中式的健康检查方式，同时配合必要的 P2P 检测。具体实现方式是：由中心服务 Scanner 监测所有节点的状态，当 Scanner 主动检测到节点异常或 Pilot 感知连接变化通知 Scanner 扫描确认节点异常时，Pilot 立刻通过 eDS 更新节点状态给 Proxy，这种模式下检测周期内仅需要检测 N 次。

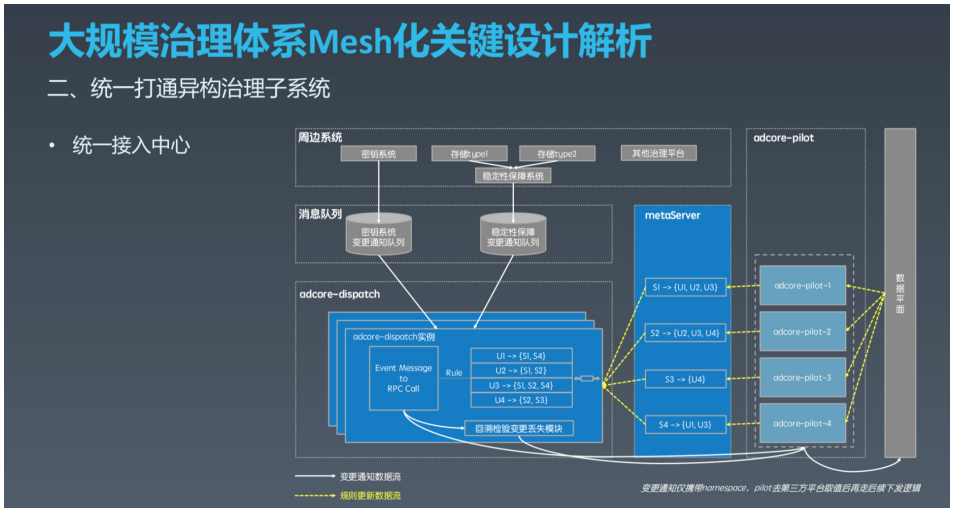
Google 的 Traffic Director 也采用了类似的设计，但大规模使用需要一些技巧：第一个是为了避免机房自治的影响而选择了同机房检测方式，第二个是为了减少中心检测机器因自己 GC 或网络异常造成误判，而采用了 Double Check 的机制。

此外除了集中健康检查，还会对频繁失败的对端进行心跳探测，根据探测结果进行降权或摘除操作提升成功率。

3.2 异构治理系统融合设计

OCTO Mesh 需要对齐当前体系的核心治理能力，这就不可避免的将 Mesh 与治理生态的所有周边子系统打通。Istio 和 Kubernetes 将所有的数据存储、发布订阅机制都依赖 Etcd 统一实现，但美团的 10 余个治理子系统功能各异、存储各异、发布订阅模式各异，呈现出明显的异构特征，如果接入一个功能就需要平台进行存储或其他大规模改造，这样是完全不可行的。一个思路是由一个模块来解耦治理子系统与 Pilot ，这个模块承载所有的变更并将这个变更下发给 Pilot，但这种方式也有一些问题需要考虑，之前介绍每个 Pilot 节点关注的数据并不同，而且分片的规则也可能时刻变化，有一套机制能将消息发送给关注的 Pilot 节点。

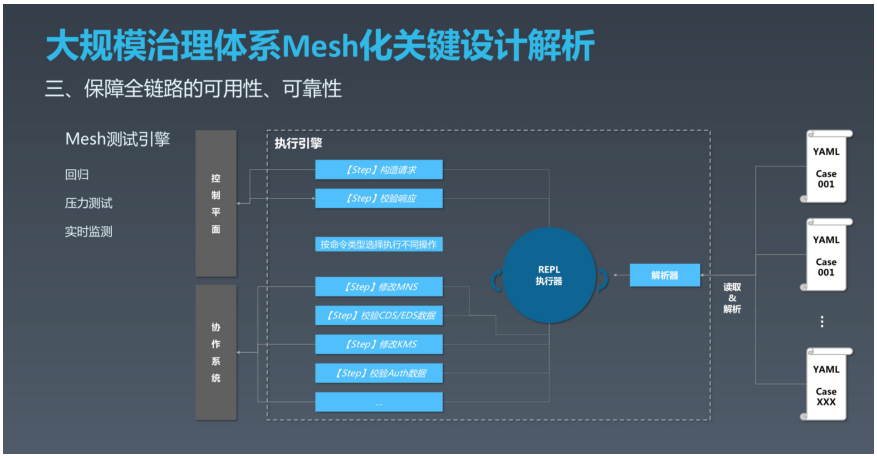
总体而言需要实现三个子目标：打通所有系统，治理能力对齐；快速应对未来新系统的接入；变更发送给关注节点。我们解法是：独立的统一接入中心，屏蔽所有异构系统的存储、发布订阅机制；Meta Server 承担实时分片规则的元数据管理。



具体执行机制如上图所示：各系统变更时使用客户端将变更通知推送到消息队列，只推送变更但不包含具体值（当 Pilot 接收到变更通知是会主动 Fetch 全量数据，这种方式一方面确保 Mafka 的消息足够小，另一方面多个变更不需要在队列中保序解决版本冲突问题）；Adcore Dispatcher 消费信息并根据索引将变更推送到关注的 Pilot 机器，当 Pilot 管控的 Proxy 变更时会同步给 Meta Server，Meta Server 实时将索引关系更新并同步给 Dispatcher。为了解决 Pilot 与应用的映射变更间隙出现消息丢失，Dispatcher 使用回溯检验变更丢失的模式进行补偿，以提升系统的可靠性。

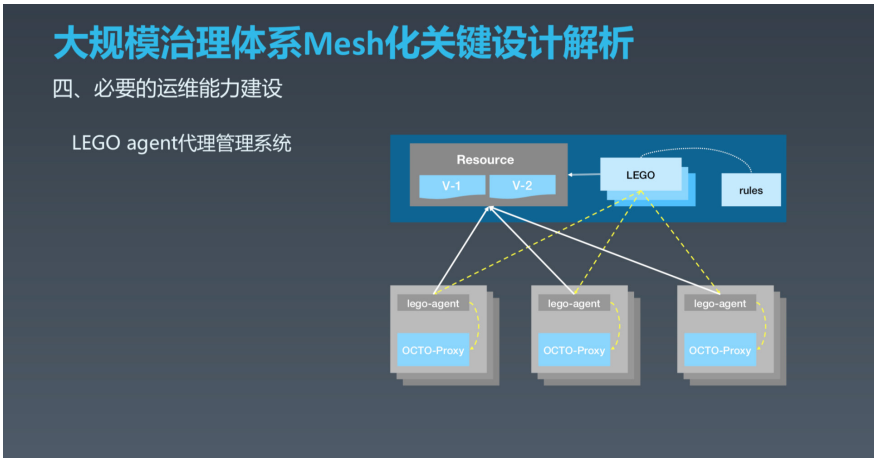
3.3 稳定性保障设计

Service Mesh 改造的系统避不开“新”和“复杂”两个特征，其中任意一个特征都可能会给系统带来稳定性风险，所以必须提前做好整个链路的可用性及可靠性建设，才能游刃有余的推广。美团主要是围绕控制故障影响范围、异常实时自愈、可实时回滚、柔性可用、提升自身可观测性及回归能力进行建设。



这里单独介绍控制面的测试问题，这块业界可借鉴的内容不多。xDS 双向通信比较复杂，很难像传统接口那样进行功能测试，定制多个 Envoy 来模拟数据面进行测试成本也很高。 我们开发了 Mock-Sidecar 来模拟真正数据面的行为来对控制面进行测试，对于控制面来说它跟数据面毫无区别。Mock-Sidecar 把数据面的整体行为拆分为一个个可组合的 Step，机制与策略分离。执行引擎就是所谓的机制，只需要按步骤执行 Step 即可。YAML 文件就是 Step 的组合，用于描述策略。我们人工构造各种 YAML 来模拟真正 Sidecar 的行为，对控制面进行回归验证，同时不同 YAML 文件执行是并行的，可以进行压力测试。

3.4 运维体系设计



为了应对未来百万级 Proxy 的运维压力，美团独立建设了 OCTO Proxy 运维系统 LEGO，除 Proxy 保活外也统一集中控制发版。具体的操作流程是：运维人员在 LEGO 平台发版，确定发版的范围及版本，新版本资源内容上传至资源仓库，并更新规则及发版范围至 DB，发升级指令下发至所要发布的范围，收到发版命令机器的 LEGO Agent 去资源仓库拉取要更新的版本（中间如果有失败，会有主动 Poll 机制保证升级成功），新版本下载成功后，由 LEGO Agent 启动新版的 OCTO Proxy。

四、总结与展望

4.1 经验总结

- 服务治理建设应该围绕体系标准化、易用性、高性能三个方面开展。
- 大规模治理体系 Mesh 化应该关注以下内容：
  - 适配公司技术体系比新潮技术更重要，重点关注容器化 & 治理体系兼容打通。
  - 建设系统化的稳定性保障体系及运维体系。
- OCTO Mesh 控制面 4 大法宝：Meta Server 管控 Mesh 内部服务注册发现及元数据、分层分片设计、统一接入中心解耦并打通 Mesh 与现有治理子系统、集中式健康检查。

4.2 未来展望

未来，我们会继续在 OCTO Mesh 道路上探索，包括但不限于以下几个方面：

- 完善体系：逐渐丰富的 OCTO Mesh 治理体系，探索其他流量类型，全面提升服务治理效率。
- 大规模落地：持续打造健壮的 OCTO Mesh 治理体系，稳步推动在公司的大规模落地。
- 中心化治理能力探索：新治理模式的中心化管控下，全局最优治理能力探索。

作者简介

继东，基础架构部服务治理团队。

招聘信息

美团点评基础架构团队诚招高级、资深技术专家，Base 北京、上海。我们致力于建设美团点评全公司统一的高并发高性能分布式基础架构平台，涵盖数据库、分布式监控、服务治理、高性能通信、消息中间件、基础存储、容器化、集群调度等基础架构主要的技术领域。欢迎有兴趣的同学投送简历到 tech@meituan.com（邮件标题注明：美团点评基础架构团队）。

实践与经验总结

XGBoost 缺失值引发的问题及其深度分析

李兆军

1. 背景

XGBoost 模型作为机器学习中的一大“杀器”，被广泛应用于数据科学竞赛和工业领域，XGBoost 官方也提供了可运行于各种平台和环境的对应代码，如适用于 Spark 分布式训练的 XGBoost on Spark。然而，在 XGBoost on Spark 的官方实现中，却存在一个因 XGBoost 缺失值和 Spark 稀疏表示机制而带来的不稳定问题。

事情起源于美团内部某机器学习平台使用方同学的反馈，在该平台上训练出的 XGBoost 模型，使用同一个模型、同一份测试数据，在本地调用（Java 引擎）与平台（Spark 引擎）计算的结果不一致。但是该同学在本地运行两种引擎（Python 引擎和 Java 引擎）进行测试，两者的执行结果是一致的。因此质疑平台的 XGBoost 预测结果会不会有问题?

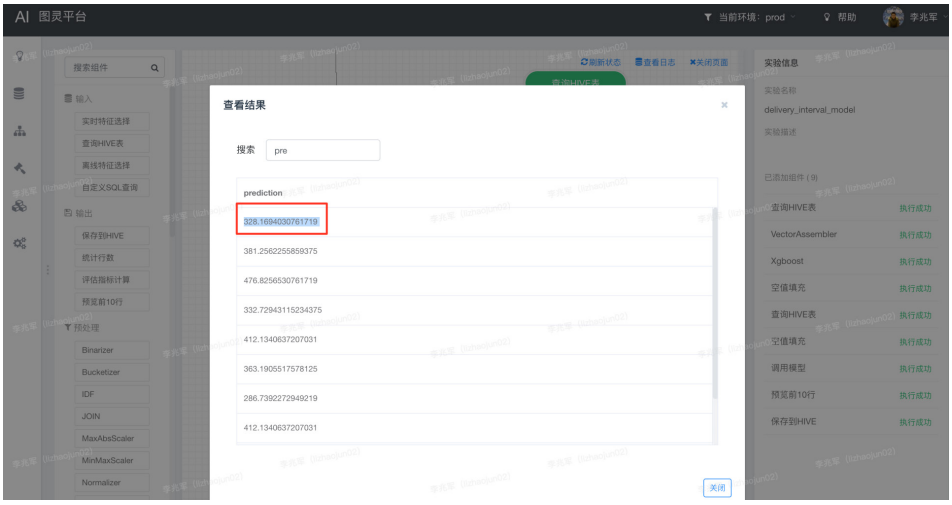
该平台对 XGBoost 模型进行过多次定向优化，在 XGBoost 模型测试时，并没有出现过本地调用（Java 引擎）与平台（Spark 引擎）计算结果不一致的情形。而且平台上运行的版本，和该同学本地使用的版本，都来源于 Dmlc 的官方版本，JNI 底层调用的应该是同一份代码，理论上，结果应该是完全一致的，但实际中却不同。

从该同学给出的测试代码上，并没有发现什么问题：

```
// 测试结果中的一行，41 列
double[] input = new double[]{1, 2, 5, 0, 0, 6.666666666666667, 31.14,
29.28, 0, 1.303333,
2.8555, 2.37, 701, 463, 3.989, 3.85, 14400.5, 15.79, 11.45, 0.915,
7.05, 5.5, 0.023333, 0.0365,
0.0275, 0.123333, 0.4645, 0.12, 15.082, 14.48, 0, 31.8425, 29.1,
7.7325, 3, 5.88, 1.08, 0, 0, 0,
32];
// 转化为 float[]
```

```
float[] testInput = new float[input.length];
for(int i = 0, total = input.length; i < total; i++){
  testInput[i] = new Double(input[i]).floatValue();
}
// 加载模型
Booster booster = XGBoost.loadModel("${model}");
// 转为 DMatrix，一行，41 列
DMatrix testMat = new DMatrix(testInput, 1, 41);
// 调用模型
float[][] predicts = booster.predict(testMat);
```

上述代码在本地执行的结果是 333.67892，而平台上执行的结果却是 328.1694030761719。



两次结果怎么会不一样，问题出现在哪里呢?

2. 执行结果不一致问题排查历程

如何排查？首先想到排查方向就是，两种处理方式中输入的字段类型会不会不一致。如果两种输入中字段类型不一致，或者小数精度不同，那结果出现不同就是可解释的了。仔细分析模型的输入，注意到数组中有一个 6.666666666666667，是不是它的原因?

一个个 Debug 仔细比对两侧的输入数据及其字段类型，完全一致。

这就排除了两种方式处理时，字段类型和精度不一致的问题。

第二个排查思路是，XGBoost on Spark 按照模型的功能，提供了 XGBoostClassifier 和 XGBoostRegressor 两个上层 API，这两个上层 API 在 JNI 的基础上，加入了很多超参数，封装了很多上层能力。会不会是在这两种封装过程中，新加入的某些超参数对输入结果有着特殊的处理，从而导致结果不一致?

与反馈此问题的同学沟通后得知，其 Python 代码中设置的超参数与平台设置的完全一致。仔细检查 XGBoostClassifier 和 XGBoostRegressor 的源代码，两者对输出结果并没有做任何特殊处理。

再次排除了 XGBoost on Spark 超参数封装问题。

再一次检查模型的输入，这次的排查思路是，检查一下模型的输入中有没有特殊的数值，比方说，NaN、-1、0 等。果然，输入数组中有好几个 0 出现，会不会是因为缺失值处理的问题?

快速找到两个引擎的源码，**发现两者对缺失值的处理真的不一致!**

XGBoost4j 中缺失值的处理

XGBoost4j 缺失值的处理过程发生在构造 DMatrix 过程中，默认将 0.0f 设置为缺失值：

```
  /**
   * create DMatrix from dense matrix
   *
   * @param data data values
   * @param nrow number of rows
   * @param ncol number of columns
   * @throws XGBoostError native error
   */
  public DMatrix(float[] data, int nrow, int ncol) throws XGBoostError {
    long[] out = new long[1];

    //0.0f 作为 missing 的值
    XGBoostJNI.checkCall(XGBoostJNI.XGDMatrixCreateFromMat(data, nrow,
ncol, 0.0f, out));

    handle = out[0];
  }
```

XGBoost on Spark 中缺失值的处理

而 xgboost on Spark 将 NaN 作为默认的缺失值。

```
  /**
   * @return A tuple of the booster and the metrics used to build
training summary
   */
  @throws(classOf[XGBoostError])
  def trainDistributed(
      trainingDataIn: RDD[XGBLabeledPoint],
      params: Map[String, Any],
      round: Int,
      nWorkers: Int,
      obj: ObjectiveTrait = null,
      eval: EvalTrait = null,
      useExternalMemory: Boolean = false,

      //NaN 作为 missing 的值
      missing: Float = Float.NaN,

      hasGroup: Boolean = false): (Booster, Map[String, Array[Float]])
= {
      //...
      }
```

也就是说，本地 Java 调用构造 DMatrix 时，如果不设置缺失值，默认值 0 被当作缺失值进行处理。而在 XGBoost on Spark 中，默认 NaN 会被为缺失值。原来 Java 引擎和 XGBoost on Spark 引擎默认的缺失值并不一样。而平台和该同学调用时，都没有设置缺失值，造成两个引擎执行结果不一致的原因，就是因为缺失值不一致！

修改测试代码，在 Java 引擎代码上设置缺失值为 NaN，执行结果为 328.1694，与平台计算结果完全一致。

```
    // 测试结果中的一行，41 列
    double[] input = new double[]{1, 2, 5, 0, 0, 6.666666666666667,
31.14, 29.28, 0, 1.303333,
2.8555, 2.37, 701, 463, 3.989, 3.85, 14400.5, 15.79, 11.45, 0.915,
7.05, 5.5, 0.023333, 0.0365,
0.0275, 0.123333, 0.4645, 0.12, 15.082, 14.48, 0, 31.8425, 29.1,
7.7325, 3, 5.88, 1.08, 0, 0, 0,
32];
    float[] testInput = new float[input.length];
    for(int i = 0, total = input.length; i < total; i++){
      testInput[i] = new Double(input[i]).floatValue();
    }
```

```
Booster booster = XGBoost.loadModel("${model}");
// 一行，41 列
DMatrix testMat = new DMatrix(testInput, 1, 41, Float.NaN);
float[][] predicts = booster.predict(testMat);
```

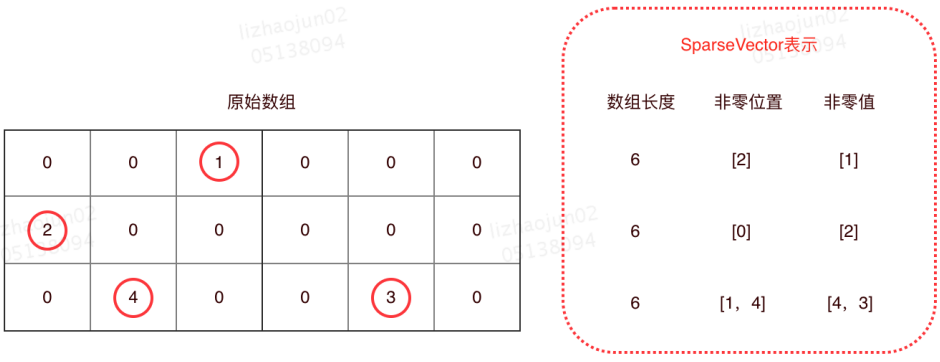
3. XGBoost on Spark 源码中缺失值引入的不稳定问题

然而，事情并没有这么简单。

Spark ML 中还有隐藏的缺失值处理逻辑：SparseVector，即稀疏向量。

- SparseVector 和 DenseVector 都用于表示一个向量，两者之间仅仅是存储结构的不同。
- 其中，DenseVector 就是普通的 Vector 存储，按序存储 Vector 中的每一个值。
- 而 SparseVector 是稀疏的表示，用于向量中 0 值非常多场景下数据的存储。
- SparseVector 的存储方式是：仅仅记录所有非 0 值，忽略掉所有 0 值。具体来说，用一个数组记录所有非 0 值的位置，另一个数组记录上述位置所对应的数值。有了上述两个数组，再加上当前向量的总长度，即可将原始的数组还原回来。
- 因此，对于 0 值非常多的一组数据，SparseVector 能大幅节省存储空间。

SparseVector 存储示例见下图：



如上图所示，SparseVector 中不保存数组中值为 0 的部分，仅仅记录非 0 值。因此对于值为 0 的位置其实不占用存储空间。下述代码是 Spark ML 中 VectorAssembler 的实现代码，从代码中可见，如果数值是 0，在 SparseVector 中是不进行记录的。

```
    private[feature] def assemble(vv: Any*): Vector = {
    val indices = ArrayBuilder.make[Int]
    val values = ArrayBuilder.make[Double]
    var cur = 0
    vv.foreach {
      case v: Double =>

        //0 不进行保存
        if (v != 0.0) {

          indices += cur
          values += v
        }
        cur += 1
      case vec: Vector =>
        vec.foreachActive { case (i, v) =>

          //0 不进行保存
          if (v != 0.0) {

            indices += cur + i
            values += v
          }
        }
        cur += vec.size
      case null =>
        throw new SparkException("Values to assemble cannot be null.")
      case o =>
        throw new SparkException(s"$o of type ${o.getClass.getName} is
not supported.")
    }
    Vectors.sparse(cur, indices.result(), values.result()).compressed
    }
```

不占用存储空间的值，也是某种意义上的一种缺失值。SparseVector 作为 Spark ML 中的数组的保存格式，被所有的算法组件使用，包括 XGBoost on Spark。而事实上 XGBoost on Spark 也的确将 Sparse Vector 中的 0 值直接当作缺失值进行处理：

```
    val instances: RDD[XGBLabeledPoint] = dataset.select(
      col($(featuresCol)),
      col($(labelCol)).cast(FloatType),
      baseMargin.cast(FloatType),
      weight.cast(FloatType)
    ).rdd.map { case Row(features: Vector, label: Float, baseMargin:
Float, weight: Float) =>
      val (indices, values) = features match {

        //SparseVector 格式，仅仅将非 0 的值放入 XGBoost 计算
        case v: SparseVector => (v.indices, v.values.map(_.toFloat))

        case v: DenseVector => (null, v.values.map(_.toFloat))
      }
      XGBLabeledPoint(label, indices, values, baseMargin = baseMargin,
weight = weight)
    }
```

XGBoost on Spark 将 SparseVector 中的 0 值作为缺失值**为什么会引入不稳定的问题呢?**

重点来了，Spark ML 中对 Vector 类型的存储是有优化的，它会自动根据 Vector 数组中的内容选择是存储为 SparseVector，还是 DenseVector。也就是说，一个 Vector 类型的字段，在 Spark 保存时，同一列会有两种保存格式：SparseVector 和 DenseVector。而且对于一份数据中的某一列，两种格式是同时存在的，有些行是 Sparse 表示，有些行是 Dense 表示。选择使用哪种格式表示通过下述代码计算得到：

```
  /**
   * Returns a vector in either dense or sparse format, whichever
uses less storage.
   */
  @Since("2.0.0")
  def compressed: Vector = {
    val nnz = numNonzeros
    // A dense vector needs 8 * size + 8 bytes, while a sparse vector
needs 12 * nnz + 20 bytes.
    if (1.5 * (nnz + 1.0) < size) {
      toSparse
    } else {
      toDense
    }
  }
```

在 XGBoost on Spark 场景下，默认将 Float.NaN 作为缺失值。如果数据集中的某一行存储结构是 DenseVector，实际执行时，该行的缺失值是 Float.NaN。而如果数据集中的某一行存储结构是 SparseVector，由于 XGBoost on Spark 仅仅使用了 SparseVector 中的非 0 值，也就导致该行数据的缺失值是 Float.NaN 和 0。

也就是说，如果数据集中某一行数据适合存储为 DenseVector，则 XGBoost 处理时，该行的缺失值为 Float.NaN。而如果该行数据适合存储为 SparseVector，则 XGBoost 处理时，该行的缺失值为 Float.NaN 和 0。

即，数据集中一部分数据会以 Float.NaN 和 0 作为缺失值，另一部分数据会以 Float.NaN 作为缺失值！ 也就是说在 XGBoost on Spark 中，0 值会因为底层数据存储结构的不同，同时会有两种含义，而底层的存储结构是完全由数据集决定的。

因为线上 Serving 时，只能设置一个缺失值，因此被选为 SparseVector 格式的测试集，可能会导致线上 Serving 时，计算结果与期望结果不符。

4. 问题解决

查了一下 XGBoost on Spark 的最新源码，依然没解决这个问题。

赶紧把这个问题反馈给 XGBoost on Spark，同时修改了我们自己的 XGBoost on Spark 代码。

```
    val instances: RDD[XGBLabeledPoint] = dataset.select(
       col($(featuresCol)),
       col($(labelCol)).cast(FloatType),
       baseMargin.cast(FloatType),
       weight.cast(FloatType)
     ).rdd.map { case Row(features: Vector, label: Float, baseMargin:
Float, weight: Float) =>

       // 这里需要对原来代码的返回格式进行修改
       val values = features match {

         //SparseVector 的数据，先转成 Dense
         case v: SparseVector => v.toArray.map(_.toFloat)

         case v: DenseVector => v.values.map(_.toFloat)
       }
       XGBLabeledPoint(label, null, values, baseMargin = baseMargin,
```

```
weight = weight)
    }
    /**
     * Converts a [[Vector]] to a data point with a dummy label.
     *
     * This is needed for constructing a [[ml.dmlc.xgboost4j.scala.
DMatrix]]
     * for prediction.
     */
    def asXGB: XGBLabeledPoint = v match {
      case v: DenseVector =>
        XGBLabeledPoint(0.0f, null, v.values.map(_.toFloat))
      case v: SparseVector =>

        //SparseVector 的数据，先转成 Dense
        XGBLabeledPoint(0.0f, null, v.toArray.map(_.toFloat))

    }
```

问题得到解决，而且用新代码训练出来的模型，评价指标还会有些许提升，也算是意外之喜。

希望本文对遇到 XGBoost 缺失值问题的同学能够有所帮助，也欢迎大家一起交流讨论。

作者简介

兆军，美团配送事业部算法平台团队技术专家。

招聘信息

美团配送事业部算法平台团队，负责美团一站式大规模机器学习平台图灵平台的建设。围绕算法整个生命周期，利用可视化拖拽方式定义模型训练和预测流程，提供强大的模型管理、线上模型预测和特征服务能力，提供多维立体的 AB 分流支持和线上效果评估支持。团队的使命是为算法相关同学提供统一的，端到端的，一站式自助服务平台，帮助算法同学降低算法研发复杂度，提升算法迭代效率。

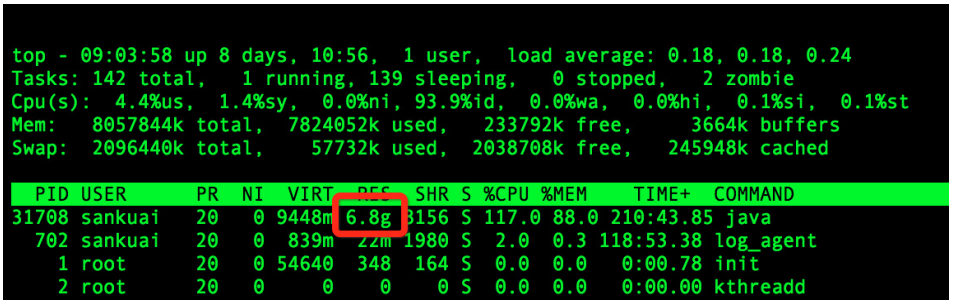
现面向数据工程，数据开发，算法工程，算法应用等领域招聘资深研发工程师 / 技术专家 / 方向负责人（机器学习平台 / 算法平台），欢迎有兴趣的同学一起加入，简历可投递至：tech@meituan.com（注明：美团配送事业部）

Spring Boot 引起的“堆外内存泄漏”排查及经验总结

纪兵

背景

为了更好地实现对项目的管理，我们将组内一个项目迁移到 MDP 框架（基于 Spring Boot），随后我们就发现系统会频繁报出 Swap 区域使用量过高的异常。笔者被叫去帮忙查看原因，发现配置了 4G 堆内内存，但是实际使用的物理内存竟然高达 7G，确实不正常。JVM 参数配置是“-XX:MetaspaceSize=256M -XX:MaxMetaspaceSize=256M -XX:+AlwaysPreTouch -XX:ReservedCodeCacheSize=128m -XX:InitialCodeCacheSize=128m, -Xss512k -Xmx4g -Xms4g,-XX:+UseG1GC -XX:G1HeapRegionSize=4M”，实际使用的物理内存如下图所示：



top 命令显示的内存情况

排查过程

1. 使用 Java 层面的工具定位内存区域（堆内内存、Code 区域或者使用 unsafe.allocateMemory 和 DirectByteBuffer 申请的堆外内存）

笔者在项目中添加 -XX:NativeMemoryTracking=detailJVM 参数重启项目，使用命令 jcmd pid VM.native_memory detail 查看到的内存分布如下：

```
Native Memory Tracking:

Total: reserved=6478032KB, committed=5440716KB
-                 Java Heap (reserved=4194304KB, committed=4194304KB)
                            (mmap: reserved=4194304KB, committed=4194304KB)

-                     Class (reserved=1140931KB, committed=103619KB)
                            (classes #16301)
                            (malloc=2243KB #40399)
                            (mmap: reserved=1138688KB, committed=101376KB)

-                    Thread (reserved=570885KB, committed=570885KB)
                            (thread #555)
                            (stack: reserved=569512KB, committed=569512KB)
                            (malloc=722KB #2777)
                            (arena=650KB #1109)

-                      Code (reserved=146450KB, committed=146450KB)
                            (malloc=13330KB #16372)
                            (mmap: reserved=133120KB, committed=133120KB)

-                        GC (reserved=205314KB, committed=205314KB)
                            (malloc=16898KB #26279)
                            (mmap: reserved=188416KB, committed=188416KB)

-                  Compiler (reserved=1921KB, committed=1921KB)
                            (malloc=1790KB #2122)
                            (arena=131KB #3)

-                  Internal (reserved=189371KB, committed=189367KB)
                            (malloc=189335KB #74055)
                            (mmap: reserved=36KB, committed=32KB)

-                    Symbol (reserved=22818KB, committed=22818KB)
                            (malloc=18911KB #189399)
                            (arena=3908KB #1)

-    Native Memory Tracking (reserved=5822KB, committed=5822KB)
                            (malloc=254KB #3754)
                            (tracking overhead=5569KB)

-               Arena Chunk (reserved=215KB, committed=215KB)
                            (malloc=215KB)

Virtual memory map:
```

jcmd 显示的内存情况

发现命令显示的 committed 的内存小于物理内存，因为 jcmd 命令显示的内存包含堆内内存、Code 区域、通过 unsafe.allocateMemory 和 DirectByteBuffer 申请的内存，但是不包含其他 Native Code（C 代码）申请的堆外内存。所以猜测是使用 Native Code 申请内存所导致的问题。

为了防止误判，笔者使用了 pmap 查看内存分布，发现大量的 64M 的地址；而这些地址空间不在 jcmd 命令所给出的地址空间里面，基本上就断定就是这些 64M 的内存所导致。

```
[sankuai@yp-hotel-cbs-poisummary02 ~]$ pmap -x 15573 | sort -k 3 -n -r
total kB         11456816 6439012 6423480
00000006c0000000 4205696 3188384 3188384 rw---   [ anon ]
00007f758a16c000  131072  131072  131072 rwx--   [ anon ]
00007f7444000000  131060  129628  129628 rw---   [ anon ]
00007f74c8000000   65536   65536   65536 rw---   [ anon ]
00007f7468000000   65536   65536   65536 rw---   [ anon ]
00007f748c000000   65508   64364   64364 rw---   [ anon ]
00007f7464000000   65520   64344   64344 rw---   [ anon ]
00007f7484000000   65512   64332   64332 rw---   [ anon ]
00007f74bc000000   65516   64296   64296 rw---   [ anon ]
00007f74ac000000   65524   64260   64260 rw---   [ anon ]
00007f74d4000000   65508   64256   64256 rw---   [ anon ]
00007f7474000000   65508   64248   64248 rw---   [ anon ]
00007f74a8000000   65512   64244   64244 rw---   [ anon ]
00007f74a4000000   65508   64224   64224 rw---   [ anon ]
00007f7490000000   65528   64224   64224 rw---   [ anon ]
00007f74b4000000   65516   64220   64220 rw---   [ anon ]
00007f7478000000   65508   64208   64208 rw---   [ anon ]
00007f7460000000   65512   64208   64208 rw---   [ anon ]
00007f74b8000000   65532   64204   64204 rw---   [ anon ]
00007f744c000000   65524   64204   64204 rw---   [ anon ]
00007f7458000000   65520   64200   64200 rw---   [ anon ]
00007f7454000000   65532   64200   64200 rw---   [ anon ]
00007f74c4000000   65508   64196   64196 rw---   [ anon ]
00007f745c000000   65516   64196   64196 rw---   [ anon ]
00007f74d0000000   65508   64188   64188 rw---   [ anon ]
00007f747c000000   65508   64188   64188 rw---   [ anon ]
00007f7470000000   65508   64188   64188 rw---   [ anon ]
00007f7488000000   65508   64180   64180 rw---   [ anon ]
00007f74b0000000   65516   64172   64172 rw---   [ anon ]
00007f746c000000   65512   64172   64172 rw---   [ anon ]
00007f74a0000000   65512   64164   64164 rw---   [ anon ]
00007f7450000000   65516   64156   64156 rw---   [ anon ]
00007f7494000000   65508   64152   64152 rw---   [ anon ]
00007f749c000000   65508   64148   64148 rw---   [ anon ]
00007f74cc000000   65508   64140   64140 rw---   [ anon ]
00007f7480000000   65520   64112   64112 rw---   [ anon ]
```

pmap 显示的内存情况

2. 使用系统层面的工具定位堆外内存

因为笔者已经基本上确定是 Native Code 所引起，而 Java 层面的工具不便于排查此类问题，只能使用系统层面的工具去定位问题。

首先，使用了 gperftools 去定位问题

gperftools 的使用方法可以参考 gperftools，gperftools 的监控如下：

```
ap-gperftools/heap.hprof_21187.0030.heap (2796 MB currently in use)
kuai/heap-gperftools/heap.hprof_21187.0030.heap
ap-gperftools/heap.hprof_21187.0031.heap (2896 MB currently in use)
kuai/heap-gperftools/heap.hprof_21187.0031.heap
ap-gperftools/heap.hprof_21187.0032.heap (2996 MB currently in use)
kuai/heap-gperftools/heap.hprof_21187.0032.heap
ap-gperftools/heap.hprof_21187.0033.heap (3096 MB currently in use)
kuai/heap-gperftools/heap.hprof_21187.0033.heap
ap-gperftools/heap.hprof_21187.0034.heap (9043 MB allocated cumulatively, 640 MB currently in use)
kuai/heap-gperftools/heap.hprof_21187.0034.heap
ap-gperftools/heap.hprof_21187.0035.heap (10068 MB allocated cumulatively, 729 MB currently in use)
kuai/heap-gperftools/heap.hprof_21187.0035.heap
ap-gperftools/heap.hprof_21187.0036.heap (11106 MB allocated cumulatively, 762 MB currently in use)
kuai/heap-gperftools/heap.hprof_21187.0036.heap
```

gperftools 监控

从上图可以看出：使用 malloc 申请的的内存最高到 3G 之后就释放了，之后始终维持在 700M-800M。笔者第一反应是：难道 Native Code 中没有使用 malloc 申请，直接使用 mmap/brk 申请的?（gperftools 原理就使用动态链接的方式替换了操作系统默认的内存分配器（glibc）。）

然后，使用 strace 去追踪系统调用

因为使用 gperftools 没有追踪到这些内存，于是直接使用命令“strace -f -e”brk,mmap,munmap” -p pid”追踪向 OS 申请内存请求，但是并没有发现有可疑内存申请。strace 监控如下图所示：

```
] mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f60704f9000
] munmap(0x7f60704f9000, 4096) = 0
] mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f60704f9000
] munmap(0x7f60704f9000, 4096) = 0
] mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f60704f9000
] munmap(0x7f60704f9000, 4096) = 0
] mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f60704f9000
] munmap(0x7f60704f9000, 4096) = 0
] mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f60704f9000
] munmap(0x7f60704f9000, 4096) = 0
] mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f60704f9000
] munmap(0x7f60704f9000, 4096) = 0
```

strace 监控

接着，使用 GDB 去 dump 可疑内存

因为使用 strace 没有追踪到可疑内存申请；于是想着看看内存中的情况。就是直接使用命令 gdp -pid pid 进入 GDB 之后，然后使用命令 dump memory mem.bin startAddress endAddressdump 内存，其中 startAddress 和 endAddress 可以从 /proc/pid/smaps 中查找。然后使用 strings mem.bin 查看 dump 的内容，如下：

```
StackMapTable
SourceFile
InnerClasses
rorsimulate/internal/org/apache/commons/lang3/tuple/ImmutablePair
<L:Ljava/lang/Object;R:Ljava/lang/Object;>Lcom/meituan/trip/errorsimulate/internal/org/apache/commons/lang3/tuple/Pair<TL
Kcom/meituan/trip/errorsimulate/internal/org/apache/commons/lang3/tuple/Pair
ImmutablePair.java
serialVersionUID
left
Ljava/lang/Object;
right
|(Ljava/lang/Object;Ljava/lang/Object;)Lcom/meituan/trip/errorsimulate/internal/org/apache/commons/lang3/tuple/ImmutableP
<init>
'(Ljava/lang/Object;Ljava/lang/Object;)V
this
^Lcom/meituan/trip/errorsimulate/internal/org/apache/commons/lang3/tuple/ImmutablePair<TL;TR;>;
VLcom/meituan/trip/errorsimulate/internal/org/apache/commons/lang3/tuple/ImmutablePair;
getLeft
```

gperftools 监控

从内容上来看，像是解压后的 JAR 包信息。读取 JAR 包信息应该是在项目启动的时候，那么在项目启动之后使用 strace 作用就不是很大了。所以应该在项目启动的时候使用 strace，而不是启动完成之后。

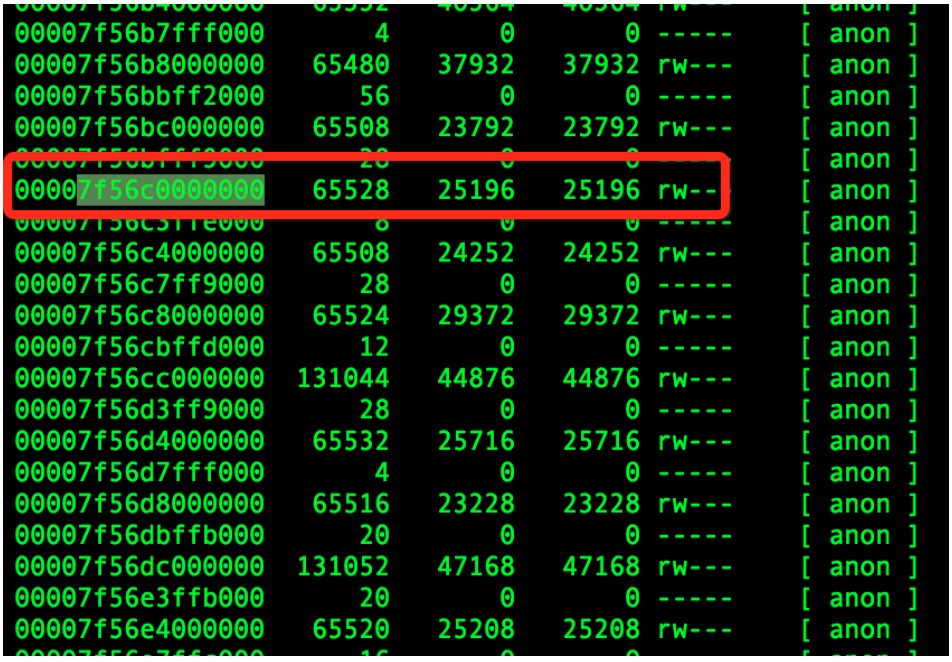
再次，项目启动时使用 strace 去追踪系统调用

项目启动使用 strace 追踪系统调用，发现确实申请了很多 64M 的内存空间，截图如下：

```
[pid 16681] mmap(NULL, 134217728, PROT_NONE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_NORESERVE, -1, 0) = 0x7f56bc000000
[pid 16681] munmap(0x7f56c0000000, 67108864) = 0
Process 16977 attached
[pid 16681] mmap(0x7f56c8000000, 67108864, PROT_NONE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_NORESERVE, -1, 0) = 0x7f56c0000000
[pid 16681] munmap(0x7f56b8000000, 67108864) = 0
```

strace 监控

使用该 mmap 申请的地址空间在 pmap 对应如下：

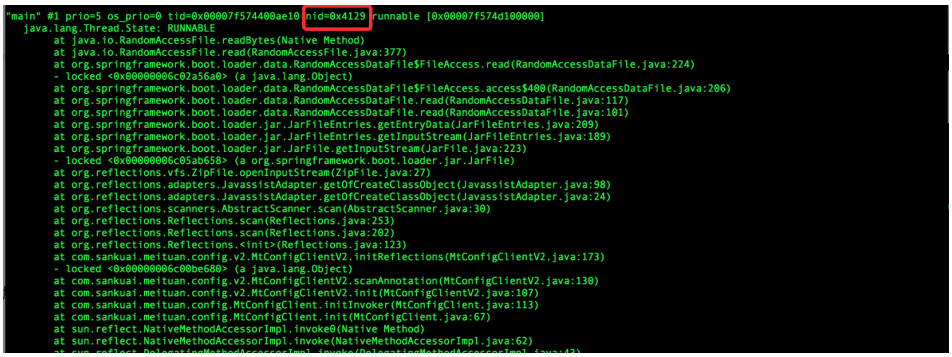


```
00007f56b7fff000      4      0      0 -----  [ anon ]
00007f56b8000000  65480  37932  37932 rw---  [ anon ]
00007f56bbff2000     56      0      0 -----  [ anon ]
00007f56bc000000  65508  23792  23792 rw---  [ anon ]
00007f56bfff9000     28      0      0 -----  [ anon ]
00007f56c0000000  65528  25196  25196 rw---  [ anon ]
00007f56c3ffe000      8      0      0 -----  [ anon ]
00007f56c4000000  65508  24252  24252 rw---  [ anon ]
00007f56c7ff9000     28      0      0 -----  [ anon ]
00007f56c8000000  65524  29372  29372 rw---  [ anon ]
00007f56cbffd000     12      0      0 -----  [ anon ]
00007f56cc000000 131044  44876  44876 rw---  [ anon ]
00007f56d3ff9000     28      0      0 -----  [ anon ]
00007f56d4000000  65532  25716  25716 rw---  [ anon ]
00007f56d7fff000      4      0      0 -----  [ anon ]
00007f56d8000000  65516  23228  23228 rw---  [ anon ]
00007f56dbffb000     20      0      0 -----  [ anon ]
00007f56dc000000 131052  47168  47168 rw---  [ anon ]
00007f56e3ffb000     20      0      0 -----  [ anon ]
00007f56e4000000  65520  25208  25208 rw---  [ anon ]
```

strace 申请内容对应的 pmap 地址空间

最后，使用 jstack 去查看对应的线程

因为 strace 命令中已经显示申请内存的线程 ID。直接使用命令 jstack pid 去查看线程栈，找到对应的线程栈（注意 10 进制和 16 进制转换）如下：

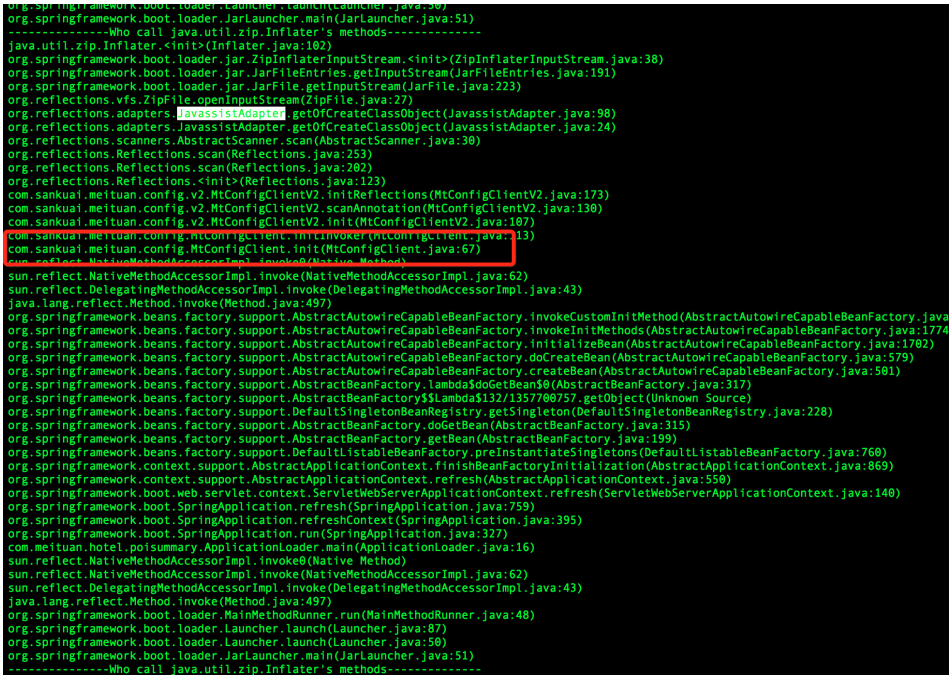


```
"main" #1 prio=5 os_prio=0 tid=0x00007f574400ae10 nid=0x4129 runnable [0x00007f574d100000]
   java.lang.Thread.State: RUNNABLE
        at java.io.RandomAccessFile.readBytes(Native Method)
        at java.io.RandomAccessFile.read(RandomAccessFile.java:377)
        at org.springframework.boot.loader.data.RandomAccessDataFile$FileAccess.read(RandomAccessDataFile.java:224)
        - locked <0x00000006c02a56a0> (a java.lang.Object)
        at org.springframework.boot.loader.data.RandomAccessDataFile$FileAccess.access$400(RandomAccessDataFile.java:206)
        at org.springframework.boot.loader.data.RandomAccessDataFile.read(RandomAccessDataFile.java:117)
        at org.springframework.boot.loader.data.RandomAccessDataFile.read(RandomAccessDataFile.java:101)
        at org.springframework.boot.loader.jar.JarFileEntries.getEntryData(JarFileEntries.java:209)
        at org.springframework.boot.loader.jar.JarFileEntries.getInputStream(JarFileEntries.java:189)
        at org.springframework.boot.loader.jar.JarFile.getInputStream(JarFile.java:223)
        - locked <0x00000006c05ab658> (a org.springframework.boot.loader.jar.JarFile)
        at org.reflections.vfs.ZipFile.openInputStream(ZipFile.java:27)
        at org.reflections.adapters.JavassistAdapter.getOfCreateClassObject(JavassistAdapter.java:98)
        at org.reflections.adapters.JavassistAdapter.getOfCreateClassObject(JavassistAdapter.java:24)
        at org.reflections.scanners.AbstractScanner.scan(AbstractScanner.java:30)
        at org.reflections.Reflections.scan(Reflections.java:253)
        at org.reflections.Reflections.scan(Reflections.java:202)
        at org.reflections.Reflections.<init>(Reflections.java:123)
        at com.sankuai.meituan.config.v2.MtConfigClientV2.initReflections(MtConfigClientV2.java:173)
        - locked <0x00000006c00be680> (a java.lang.Object)
        at com.sankuai.meituan.config.v2.MtConfigClientV2.scanAnnotation(MtConfigClientV2.java:130)
        at com.sankuai.meituan.config.v2.MtConfigClientV2.init(MtConfigClientV2.java:107)
        at com.sankuai.meituan.config.MtConfigClient.initInvoker(MtConfigClient.java:113)
        at com.sankuai.meituan.config.MtConfigClient.init(MtConfigClient.java:67)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
```

strace 申请空间的线程栈

这里基本上就可以看出问题来了：MCC（美团统一配置中心）使用了 Reflec-

tions 进行扫包，底层使用了 Spring Boot 去加载 JAR。因为解压 JAR 使用 Inflater 类，需要用到堆外内存，然后使用 Btrace 去追踪这个类，栈如下：



btrace 追踪栈

然后查看使用 MCC 的地方，发现没有配置扫包路径，默认是扫描所有的包。于是修改代码，配置扫包路径，发布上线后内存问题解决。

3. 为什么堆外内存没有释放掉呢?

虽然问题已经解决了，但是有几个疑问：

- 为什么使用旧的框架没有问题?
- 为什么堆外内存没有释放?
- 为什么内存大小都是 64M，JAR 大小不可能这么大，而且都是一样大?
- 为什么 gperftools 最终显示使用的的内存大小是 700M 左右，解压包真的没有使用 malloc 申请内存吗?

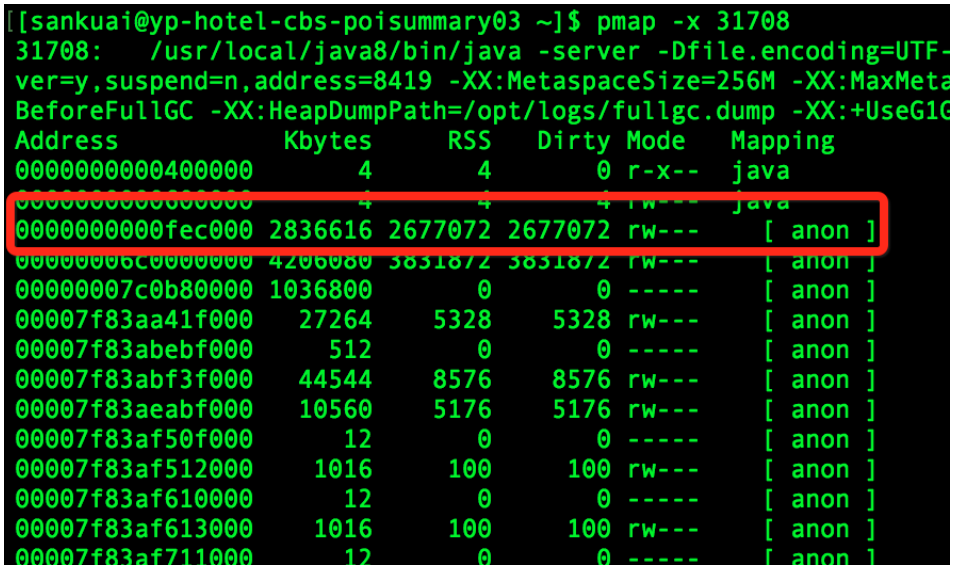
带着疑问，笔者直接看了一下 Spring Boot Loader 那一块的源码。发现 Spring Boot 对 Java JDK 的 InflaterInputStream 进行了包装并且使用了 Inflater，而 Inflater 本身用于解压 JAR 包的需要用到堆外内存。而包装之后的类 ZipInflaterInputStream 没有释放 Inflater 持有的堆外内存。于是笔者以为找到了原因，立马向 Spring Boot 社区反馈了这个 bug。但是反馈之后，笔者就发现 Inflater 这个对象本身实现了 finalize 方法，在这个方法中有调用释放堆外内存的逻辑。也就是说 Spring Boot 依赖于 GC 释放堆外内存。

笔者使用 jmap 查看堆内对象时，发现已经基本上没有 Inflater 这个对象了。于是就怀疑 GC 的时候，没有调用 finalize。带着这样的怀疑，笔者把 Inflater 进行包装在 Spring Boot Loader 里面替换成自己包装的 Inflater，在 finalize 进行打点监控，结果 finalize 方法确实被调用了。于是笔者又去看了 Inflater 对应的 C 代码，发现初始化的使用了 malloc 申请内存，end 的时候也调用了 free 去释放内存。

此刻，笔者只能怀疑 free 的时候没有真正释放内存，便把 Spring Boot 包装的 InflaterInputStream 替换成 Java JDK 自带的，发现替换之后，内存问题也得以解决了。

这时，再返过来看 gperftools 的内存分布情况，发现使用 Spring Boot 时，内存使用一直在增加，突然某个点内存使用下降了好多（使用量直接由 3G 降为 700M 左右）。这个点应该就是 GC 引起的，内存应该释放了，但是在操作系统层面并没有看到内存变化，那是不是没有释放到操作系统，被内存分配器持有了呢?

继续探究，发现系统默认的内存分配器（glibc 2.12 版本）和使用 gperftools 内存地址分布差别很明显，2.5G 地址使用 smaps 发现它是属于 Native Stack。内存地址分布如下：



```
[[sankuai@yp-hotel-cbs-poisummary03 ~]$ pmap -x 31708
31708:   /usr/local/java8/bin/java -server -Dfile.encoding=UTF-
ver=y,suspend=n,address=8419 -XX:MetaspaceSize=256M -XX:MaxMeta
BeforeFullGC -XX:HeapDumpPath=/opt/logs/fullgc.dump -XX:+UseG1G
Address           Kbytes     RSS   Dirty Mode   Mapping
0000000000400000       4       4       0 r-x--  java
0000000000600000       4       4       4 rw---  java
0000000000fec000 2836616 2677072 2677072 rw---    [ anon ]
00000006c0000000 4206080 3831872 3831872 rw---    [ anon ]
00000007c0b80000 1036800       0       0 -----    [ anon ]
00007f83aa41f000   27264    5328    5328 rw---    [ anon ]
00007f83abebf000     512       0       0 -----    [ anon ]
00007f83abf3f000   44544    8576    8576 rw---    [ anon ]
00007f83aeabf000   10560    5176    5176 rw---    [ anon ]
00007f83af50f000      12       0       0 -----    [ anon ]
00007f83af512000    1016     100     100 rw---    [ anon ]
00007f83af610000      12       0       0 -----    [ anon ]
00007f83af613000    1016     100     100 rw---    [ anon ]
00007f83af711000      12       0       0 -----    [ anon ]
```

gperftools 显示的内存地址分布

到此，基本上可以确定是内存分配器在捣鬼；搜索了一下 glibc 64M，发现 glibc 从 2.11 开始对每个线程引入内存池（64 位机器大小就是 64M 内存），原文如下：

- An enhanced dynamic memory allocation (malloc) behaviour enabling higher scalability across many sockets and cores. This is achieved by assigning threads their own memory pools and by avoiding locking in some situations. The amount of additional memory used for the memory pools (if any) can be controlled using the environment variables MALLOC_ARENA_TEST and MALLOC_ARENA_MAX. MALLOC_ARENA_TEST specifies that a test for the number of cores is performed once the number of memory pools reaches this value. MALLOC_ARENA_MAX sets the maximum number of memory pools used, regardless of the number of cores.

glib 内存池说明

按照文中所说去修改 MALLOC_ARENA_MAX 环境变量，发现没什么效果。查看 tcmalloc（gperftools 使用的内存分配器）也使用了内存池方式。

为了验证是内存池搞的鬼，笔者就简单写个不带内存池的内存分配器。使用命令 gcc zjbmalloc.c -fPIC -shared -o zjbmalloc.so 生成动态库，然后使用 export LD_PRELOAD=zjbmalloc.so 替换掉 glibc 的内存分配器。其中代码 Demo 如下：

```
#include<sys/mman.h>
#include<stdlib.h>
#include<string.h>
#include<stdio.h>
// 作者使用的 64 位机器，sizeof(size_t) 也就是 sizeof(long)
void* malloc ( size_t size )
{
   long* ptr = mmap( 0, size + sizeof(long), PROT_READ | PROT_WRITE,
MAP_PRIVATE | MAP_
ANONYMOUS, 0, 0 );
   if (ptr == MAP_FAILED) {
      return NULL;
   }
   *ptr = size;                        // First 8 bytes contain length.
   return (void*)(&ptr[1]);            // Memory that is after length
variable
}

void *calloc(size_t n, size_t size) {
 void* ptr = malloc(n * size);
 if (ptr == NULL) {
      return NULL;
 }
 memset(ptr, 0, n * size);
 return ptr;
}
void *realloc(void *ptr, size_t size)
{
 if (size == 0) {
      free(ptr);
      return NULL;
 }
 if (ptr == NULL) {
      return malloc(size);
 }
 long *plen = (long*)ptr;
 plen--;                             // Reach top of memory
 long len = *plen;
 if (size <= len) {
      return ptr;
 }
 void* rptr = malloc(size);
 if (rptr == NULL) {
      free(ptr);
      return NULL;
 }
 rptr = memcpy(rptr, ptr, len);
 free(ptr);
 return rptr;
```

```
}

void free (void* ptr )
{
   if (ptr == NULL) {
       return;
   }
   long *plen = (long*)ptr;
   plen--;                          // Reach top of memory
   long len = *plen;                // Read length
   munmap((void*)plen, len + sizeof(long));
}
```

通过在自定义分配器当中埋点可以发现其实程序启动之后应用实际申请的堆外内存始终在 700M-800M 之间，gperftools 监控显示内存使用量也是在 700M-800M 左右。但是从操作系统角度来看进程占用的内存差别很大（这里只是监控堆外内存）。

笔者做了一下测试，使用不同分配器进行不同程度的扫包，占用的内存如下：

内存分配器名称	没有扫全包	一次扫描全包	两次扫描全包
glibc	750M	1.5G	2.3G
tcmalloc	900M	2.0G	2.77g
自定义	1.7G	1.7G	1.7G

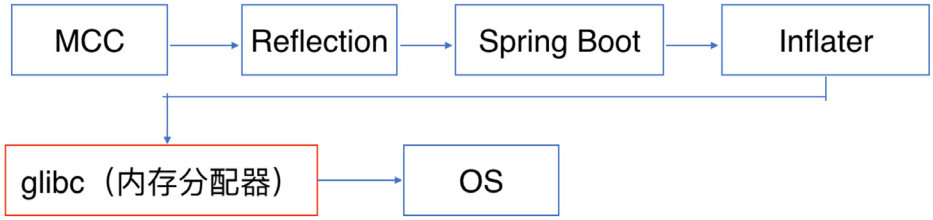
内存测试对比

为什么自定义的 malloc 申请 800M，最终占用的物理内存在 1.7G 呢?

因为自定义内存分配器采用的是 mmap 分配内存，mmap 分配内存按需向上取整到整数个页，所以存在着巨大的空间浪费。通过监控发现最终申请的页面数目在 536k 个左右，那实际上向系统申请的内存等于 512k * 4k（pagesize）= 2G。 为什么这个数据大于 1.7G 呢?

因为操作系统采取的是延迟分配的方式，通过 mmap 向系统申请内存的时候，系统仅仅返回内存地址并没有分配真实的物理内存。只有在真正使用的时候，系统产生一个缺页中断，然后再分配实际的物理 Page。

总结



流程图

整个内存分配的流程如上图所示。MCC 扫包的默认配置是扫描所有的 JAR 包。在扫描包的时候，Spring Boot 不会主动去释放堆外内存，导致在扫描阶段，堆外内存占用量一直持续飙升。当发生 GC 的时候，Spring Boot 依赖于 finalize 机制去释放了堆外内存；但是 glibc 为了性能考虑，并没有真正把内存归返到操作系统，而是留下来放入内存池了，导致应用层以为发生了“内存泄漏”。所以修改 MCC 的配置路径为特定的 JAR 包，问题解决。笔者在发表这篇文章时，发现 Spring Boot 的最新版本（2.0.5.RELEASE）已经做了修改，在 ZipInflaterInputStream 主动释放了堆外内存不再依赖 GC；所以 Spring Boot 升级到最新版本，这个问题也可以得到解决。

参考资料

1. GNU C Library (glibc)
2. Native Memory Tracking
3. Spring Boot
4. gperftools
5. Btrace

作者简介

纪兵，2015 年加入美团，目前主要从事酒店 C 端相关的工作。

美团点评效果广告实验配置平台的设计与实现

哲琪　仓魁　刘铮

一、背景

效果广告的主要特点之一是可量化，即广告系统的所有业务指标都是可以计算并通过数字进行展示的。因此，可以通过业务指标来表示广告系统的迭代效果。那如何在全量上线前确认迭代的结果呢？通用的方法是采用 AB 实验（如图 1）。所谓 AB 实验，是指单个变量具有两个版本 A 和 B 的随机实验。在实际应用中，是一种比较单个（或多个）变量多个版本的方法，通常是通过测试受试者对多个版本的反应，并确定多个版本中的哪个更有效。Google 工程师在 2000 年进行了首次 AB 实验，试图确定在其搜索引擎结果页上显示的最佳结果数。到了 2011 年，Google 进行了 7,000 多次不同的 AB 实验。现在很多公司使用“设计实验”的方法来制定营销决策，期望在实验样本上可以得到积极的转化结果，并且随着工具和专业知识在实验领域的发展，AB 实验已成为越来越普遍的一种做法。

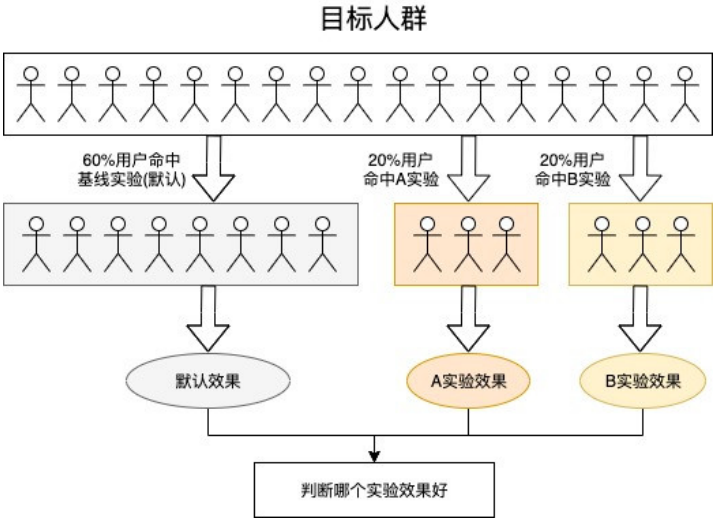


图 1　什么是 AB 实验

我们都知道，机器学习在广告投放中的作用是举足轻重的，广告收入的提升离不开算法模型的优化迭代，因此算法模型的迭代也需要进行 AB 实验。除了算法模型的迭代之外，工程中较大规模的重构和优化也需要 AB 实验来验证效果的有效性和正确性。此外，目前在大部分应用中，应用参数配置采用最多是单键值的配置方式，这种配置方式的确满足了大部分配置的需要，但是在结合业务需求的情况下，使用起来可能会很乏力。

因此，我们需要搭建一个平台（Wedge），来满足算法、工程在迭代过程中的实验需求，并且满足在不同流量下应用参数配置的需求。

二、方案设计

目标

Wedge 平台的目标如下：

- 支持各类算法实验场景，可灵活支持后续的功能扩展。
- 实验配置、使用、效果回收等全链路对使用者透明，降低解释成本。
- 提供不同流量下应用参数的配置，降低参数解析成本。
- 支持版本控制，可快速回滚。
- 提供简洁、易用的操作界面。

设计思路

在《Overlapping Experiment Infrastructure: More, Better, Faster Experimentation》中，Google 给出了一套通用的分层实验解决方案。我们以此为蓝本，结合美团点评效果广告的 LBS 特性，针对不同的业务场景，实现了更适合日常迭代的实验配置框架。目前，该框架已在搜索广告投放系统上全量上线。

实验分类

基于 Google 分层实验平台，结合实际需求进行了以下实验分类。

根据实验种类分类

- **水平实验**：类似于 Overlapping Layer 中的实验，是属于同个“层”的实验，实验是互斥的，在同一“层”上实验可以理解为是同一种实验，例如：关键词“层”表示这一层的实验都是关键词相关的，该层上存在实验 H1 和 H2，那么流量绝对不会同时命中 H1 和 H2。
- **垂直实验**：类似于 Non-overlapping Layer 中的实验，分布于不同“层”之间，实验是不互斥的，例如在关键词“层”和 CTR“层”上在相同的分桶上配置了实验 V1 和 V2，那么流量可以同时命中 V1 和 V2。
- **条件实验**：表示进入某“层”的实验需要满足某些条件，水平实验和垂直实验都可以是条件实验。
- 根据流量类别分类
- 这种分类主要了为了用户体验，使平台在操作上更加的简单、易用：
- **普通实验**：最基本的实验，根据流量类别进行配置。
- **引用实验**：流量分类是整个配置中心基础，但实际上存在一些实验是跨流量了，而引用实验则可以配置在不同的流量种类中。
- **全局实验**：可以理解为特殊的引用实验，全局实验在所有流量上都生效。

架构图

图 2 为整体架构图，比较便于大家理解，我们可以看到整体架构分为四层：

- **Web 层**：提供平台 UI，负责应用参数配置、实验配置、实验效果查看以及其他。
- **服务层**：提供权限控制、实验管理、拉取实验效果等功能。
- **存储层**：主要是数据存储功能。
- **业务层**：业务层结合 SDK 完成获取实验参数和获取应用参数的功能。

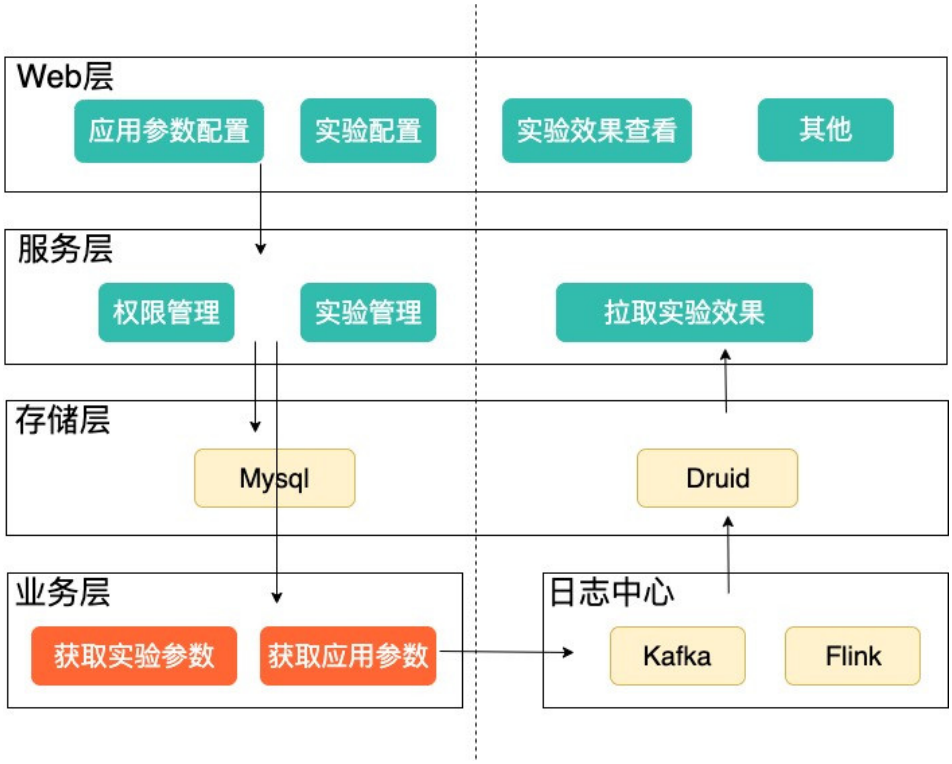


图 2　架构图

三、模型设计

1. 分流模型

实验模型

如图 3 所示，整体模型分为以下几个部分：

- App：表示一个应用，不同的应用对应的实验配置完全不同，首先从 App 上进行区分可以更加明确实验的归属。
- Scene：表示某一类流量的集合，例如：搜索、美食筛选、到综筛选等，在这些流量上配置的实验互不干扰。
- Layer：表示某一层（种）实验的集合。例如可以将在 matching 上做的实验放入 Matching Layer 中。流量命中时依次进入每个 Layer 获取实验配置参数，

此时的 Layer 更像一个抽象概念，与具体的业务或者逻辑相关。

- **条件 Layer**：是一种更加精细的流量控制方式，表示某一流量的某个或者某几个参数在满足一定条件下才会进行实验。进一步说就是相同 Scene 下，某一流量的参数 A 满足条件一时，采用一种实验配置策略；满足条件二时，采用另一种实验配置策略，那可以分为两层，如图 3 所示的 Layer_3 和 Layer_4。例如：某流量需要在城市北京单独做实验，这种情况下，可以分为参数相同但是先决条件（即城市）互斥的两个 Layer。此时的 Layer 在抽象的基础上更加的具体化。
- **Exp**：表示具体实验，包括实验的分桶、实验参数、是否为垂直流量等等。

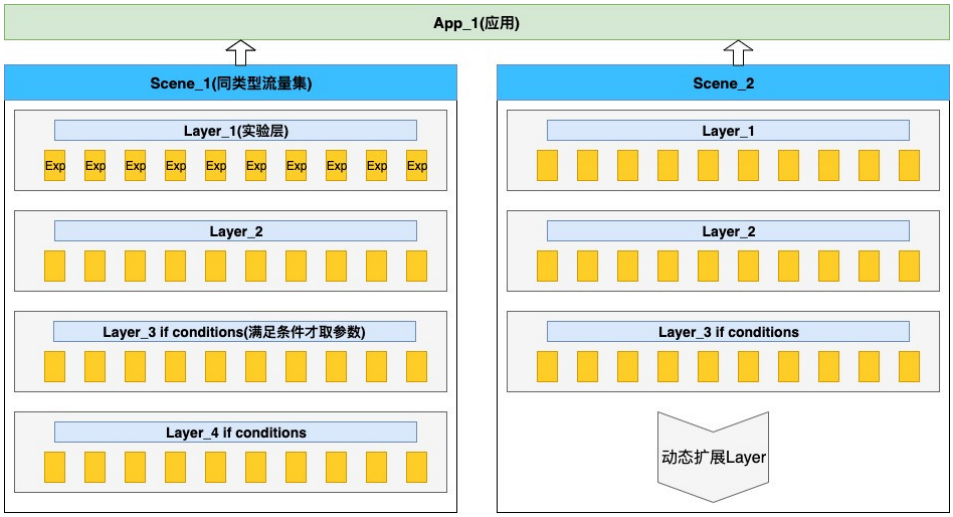


图 3　实验模型

水平、垂直分流模型

如图 4 所示，水平、垂直分流模型分为以下两个部分：

- **仅包含水平实验**：最基础的实验需求，全部实验独占一个 Layer，每个实验覆盖若干个桶，例如图 4 中的 Layer_1，将流量分为 10 份，包含三个实验，这三个实验分别占用 3、3、4 份流量。
- **同时包含水平、垂直实验**：一个 Layer 中同时包含垂直、水平两种类型的实

验。例如图 4 中的 Layer_2 和 Layer_3，将最后的 4 份流量用来做垂直实验，包含两个垂直实验，分别是 Exp_6 和 Exp_7。

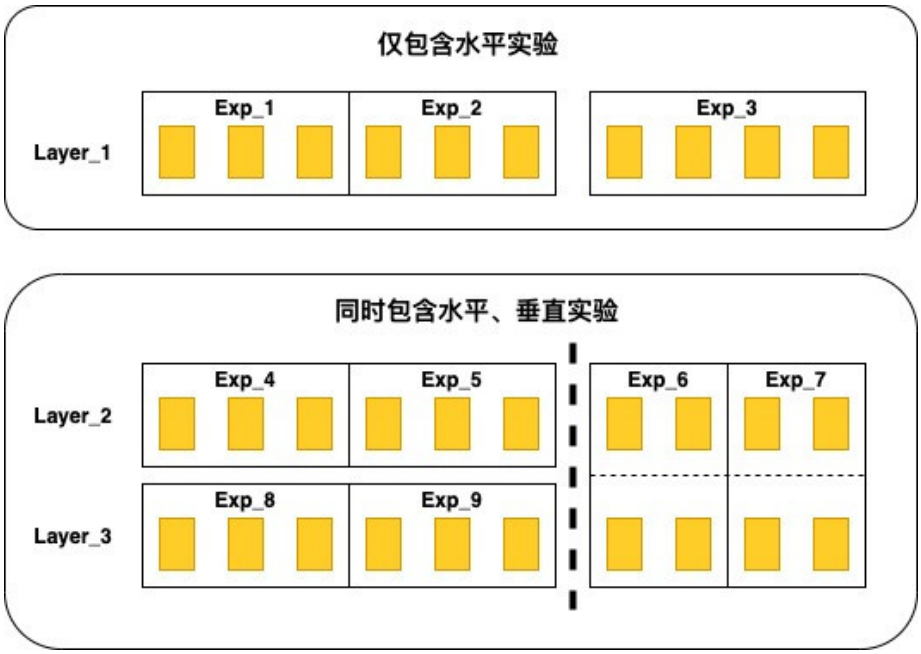


图 4　水平、垂直分流模型

2. 实验命中模型

实验命中模型是指，当一个请求过来时，返回全局统一的实验参数。所有的请求都会平均地落入每一个分桶中，并且不同的 Layer 之间能够保证流量的正交。

名词解释

Bucket	分桶
流量正交	流量在不同Layer之间的分桶是完全相互独立的
Hash优先级	表示计算Hash值的先后顺序，用于垂直和水平实验切分
Hash因子	分桶的唯一标识
Hash串	计算Hash的字符串
取模数	Hash值计算之后的除数

- Hash 优先级：在实验命中过程中，第一次 Hash 首先判断命中垂直流量，如果没有命中，则进行第二次 Hash 再判断水平流量。
- Hash 因子：目前美团侧一般情况下为 uuid，点评侧为 dpid。
- 垂直流量 Hash 串：Hash 因子 +scene_id。
- 水平流量 Hash 串：Hash 因子 +scene_id+layer_id+layer_name。
- 取模数：在 Hash 过程中，垂直流量按照总 Bucket（默认取值 100）取模；水平流量按照总 Bucket 数减去垂直流量 Bucket 数取模。这样的命中模型能保证无论是垂直的 Bucket，还是水平的 Bucket 都是全局的 1%。

实例解析

以最复杂的流量分配为例，如图 5，水平、垂直流量各占全局 50% 流量。

水平流量上包含两个实验：Exp_1、Exp_2 各占全局 20% 流量，还有 10% 流量未分配实验，垂直流量与水平流量相同。

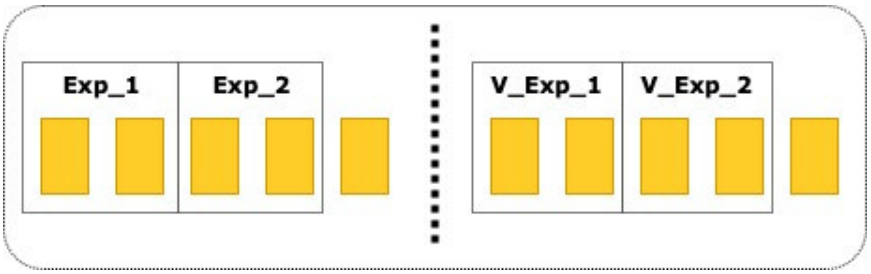


图 5　流量分配示例

典型实验命中如图 6 所示：

- 当一个请求过来时，命中顺序按照 Hash 优先级为先垂直流量后水平流量。
- 首先利用垂直流量 Hash 串进行 Hash 并取模得到一个 hashNum，如果命中了 50 ~ 99，就会认为命中了垂直流量，直接返回 V_Exp_1、V_Exp_2 或者未命中。
- 如果没有命中 50 ~ 99，则认为命中了水平流量。再利用水平 Hash 因子进行 Hash 并取模得到一个 hashNum，根据配置直接返回 Exp_1、Exp_2 或者未命中。

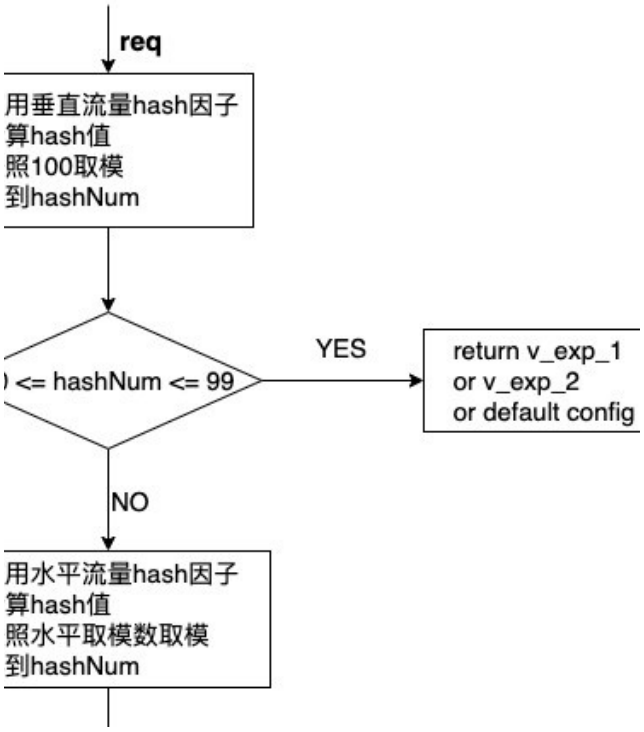


图 6　实验命中流程

应用参数模型

应用参数模型如图 7 所示，分为两层结构：

- **全局参数**：所有流量都能拿到的参数。
- **同类型流量参数**：相同类型的流量能拿到的参数。

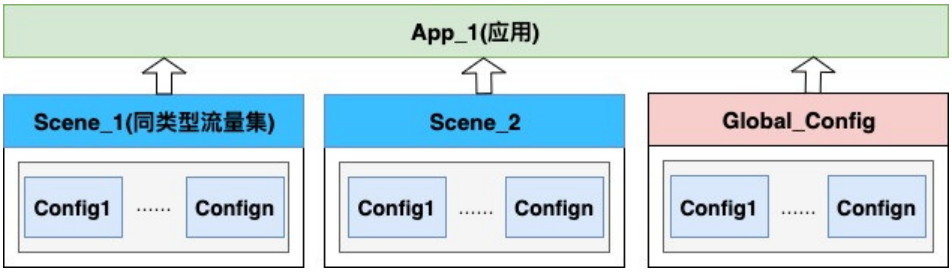


图 7　应用参数模型

值得注意的是，应用参数的两个层级以及之前提到的实验参数可能出现重复的情况，在出现重复的情况下参数的优先级为：实验参数 > 同类型流量参数 > 全局参数。

3. 回滚模型

很明显，平台的分流模型是层次结构的，所以在数据设计上也是每一层一个table。这样就会出现一个问题，一般的参数配置回滚都是单值的回滚，但是存在多个表的情况下没有办法这么简单地去回滚。

因此，我们设计了如下的回滚模型：

1. 首先在配置发布时，会将所有修改的表名、列名、列类型、列新旧值、修改类型存入表中。
2. 回滚时获取上次发布的所有修改的表名、列名、列类型、列新旧值、修改类型，反向操作数据库，达到回滚的目的。

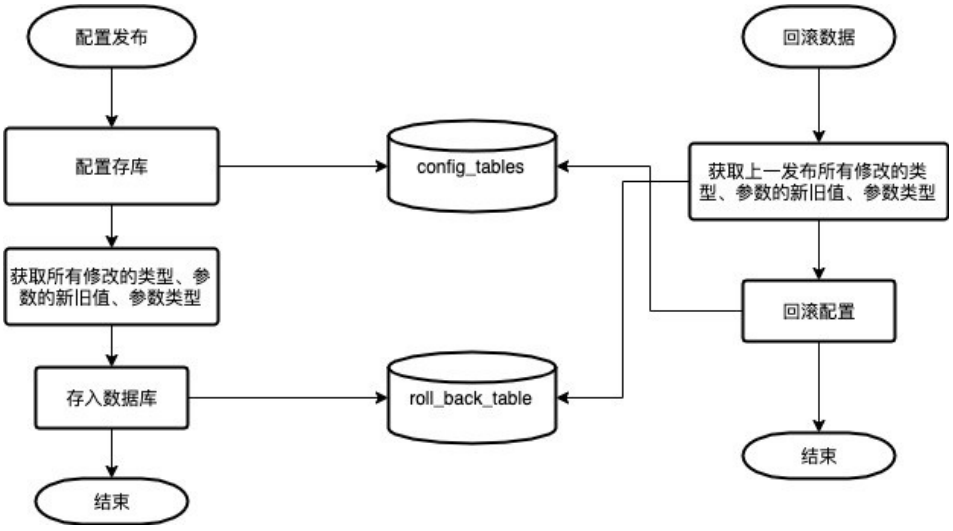


图 8　回滚模型

数据库表的设计如下：

列名	数据类型	说明
operation_id	BIGINT	表示配置发布的ID
old_value	TEXT	旧值
column_name	VARCHAR	列名
new_value	TEXT	新值
operation_step	INT	修改的数据表的顺序
table_name	VARCHAR	修改的数据表名
operation_type	INT	修改类型，包括增加、修改、删除
value_type	INT	值类型

四、AB 实验实时效果

实时效果指标包含实际 CTR、预估 CTR、请求 PV、广告密度、有效曝光、RPS 等，由于很多数据分布在不同的日志中，所以需要实时处理。

广告的打点一般分为请求（PV）打点、SPV（Server PV）打点、CPV（Client PV）曝光打点和 CPV 点击打点，在所有打点中都会包含一个流量的 requestId 和命中的实验路径。根据 requestId 和命中的实验路径可以将所有的日志进行 join，得到一个 request 中需要的所有数据，然后将数据存入 Durid 中。

计算各项效果指标，就是在日志 join 后带有实验路径的数据上做 OLAP。为了支持高效的实时查询，平台采用时序数据库 Druid 作为底层存储。Druid 作为分布式时序数据库，提供了丰富的 OLAP 能力和强悍的性能。Druid 将数据分为时间戳、维度和指标三个部分，其中维度多用于过滤，指标用于聚合和计算等。平台将数据中的实验路径同其他用于过滤的字段一同作为维度，结合时间戳和指标字段，完成指定标签的广告效果指标计算。

五、总结

目前 Wedge 平台已经完全上线，除了满足目标之外，还带来了以下的成果：

- 配合完成算法工程架构调整、更新流大版本升级。
- 算法各方向之间可以任意做垂直实验，满足算法灵活、快速迭代。
- 具备配置审核功能，保证配置发布的正确性。

六 . 作者简介

哲琪、仓魁、刘铮，美团点评效果广告引擎团队工程师。

七 . 招聘信息

美团点评效果广告引擎团队，招募出色的后端开发工程师。我们希望您：具有扎实的后端服务开发功底，能够熟练使用 Java 或 C++ 开发语言，致力于互联网技术领域。

有兴趣的同学，欢迎投递简历至：tech@meituan.com（邮件标题注明：美团点评效果广告引擎团队）

根因分析初探：一种报警聚类算法在业务系统的落地实施

刘场　千钊

背景

众所周知，日志是记录应用程序运行状态的一种重要工具，在业务服务中，日志更是十分重要。通常情况下，日志主要是记录关键执行点、程序执行错误时的现场信息等。系统出现故障时，运维人员一般先查看错误日志，定位故障原因。当业务流量小、逻辑复杂度低时，应用出现故障时错误日志一般较少，运维人员一般能够根据错误日志迅速定位到问题。但是，随着业务逻辑的迭代，系统接入的依赖服务不断增多，引入的组件不断增多，当系统出现故障时（如 Bug 被触发、依赖服务超时等等），错误日志的量级会急剧增加。极端情况下甚至出现“疯狂报错”的现象，这时候错误日志的内容会存在相互掩埋、相互影响的问题，运维人员面对报错一时难以理清逻辑，有时甚至顾此失彼，没能第一时间解决最核心的问题。

错误日志是系统报警的一种，实际生产中，运维人员能够收到的报警信息多种多样。如果在报警流出现的时候，通过处理程序，将报警进行聚类，整理出一段时间内的报警摘要，那么运维人员就可以在摘要信息的帮助下，先对当前的故障有一个大致的轮廓，再结合技术知识与业务知识定位故障的根本原因。

围绕上面描述的问题，以及对于报警聚类处理的分析假设，本文主要做了以下事情：

1. 选定聚类算法，简单描述了算法的基本原理，并给出了针对报警日志聚类的一种具体的实现方案。
2. 在分布式业务服务的系统下构造了三种不同实验场景，验证了算法的效果，并且对算法的不足进行分析阐述。

目标

对一段时间内的报警进行聚类处理，将具有相同根因的报警归纳为能够涵盖报警内容的泛化报警（Generalized Alarms），最终形成仅有几条泛化报警的报警摘要。如下图 1 所示意。

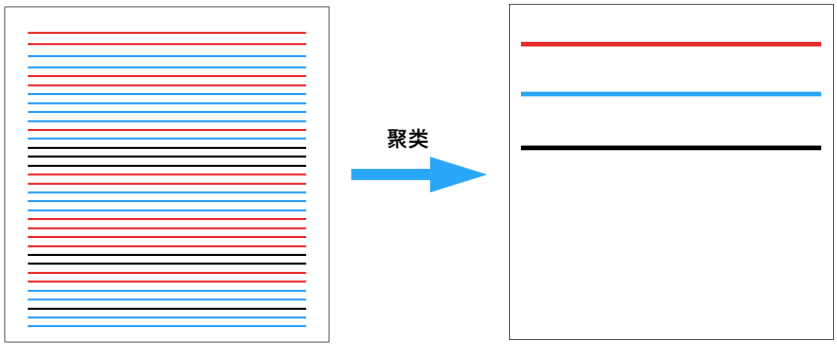


图 1

我们希望这些泛化报警既要具有很强的概括性，同时尽可能地保留细节。这样运维人员在收到报警时，便能快速定位到故障的大致方向，从而提高故障排查的效率。

设计

如图 2 所示，异常报警根因分析的设计大致分为四个部分：收集报警信息、提取报警信息的关键特征、聚类处理、展示报警摘要。

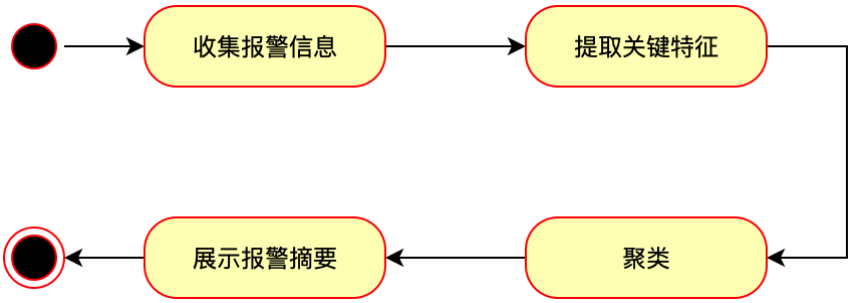


图 2

算法选择

聚类算法采用论文“Clustering Intrusion Detection Alarms to Support Root Cause Analysis [KLAUS JULISCH, 2002]”中描述的根因分析算法。该算法基于一个假设：将报警日志集群经过泛化，得到的泛化报警能够表示报警集群的主要特征。以下面的例子来说明，有如下的几条报警日志：

```
server_room_a-biz_tag-online02 Thrift get deal ProductType deal error.
server_room_b-biz_tag-offline01 Pigeon query deal info error.
server_room_a-biz_tag-offline01 Http query deal info error.
server_room_a-biz_tag-online01 Thrift query deal info error.
server_room_b-biz_tag-offline02 Thrift get deal ProductType deal error.
```

我们可以将这几条报警抽象为：“全部服务器 网络调用 故障”，该泛化报警包含的范围较广；也可以抽象为：“server_room_a 服务器 网络调用 产品信息获取失败”和“server_room_b 服务器 RPC 获取产品类型信息失败”，此时包含的范围较小。当然也可以用其他层次的抽象来表达这个报警集群。

我们可以观察到，抽象层次越高，细节越少，但是它能包含的范围就越大；反之，抽象层次越低，则可能无用信息越多，包含的范围就越小。

这种抽象的层次关系可以用一些有向无环图（DAG）来表达，如图 3 所示：

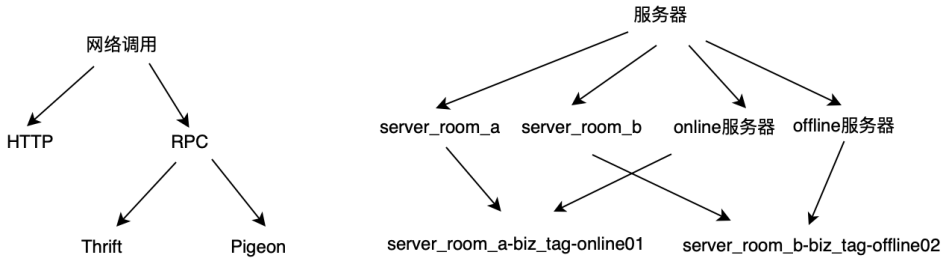


图 3　泛化层次结构示例

为了确定报警聚类泛化的程度，我们需要先了解一些定义：

- 属性（Attribute）：构成报警日志的某一类信息，如机器、环境、时间等，文中用 Ai 表示。

- 值域（Domain）：属性 Ai 的域（即取值范围），文中用 Dom(Ai) 表示。
- 泛化层次结构（Generalization Hierarchy）：对于每个 Ai 都有一个对应的泛化层次结构，文中用 Gi 表示。
- 不相似度（Dissimilarity）：定义为 d(a1, a2)。它接受两个报警 a1、a2 作为输入，并返回一个数值量，表示这两个报警不相似的程度。与相似度相反，当 d(a1, a2) 较小时，表示报警 a1 和报警 a2 相似。为了计算不相似度，需要用户定义泛化层次结构。

为了计算 d(a1, a2)，我们先定义两个属性的不相似度。令 x1、x2 为某个属性 Ai 的两个不同的值，那么 x1、x2 的不相似度为：在泛化层次结构 Gi 中，通过一个公共点父节点 p 连接 x1、x2 的最短路径长度。即 d(x1, x2) := min{d(x1, p) + d(x2, p) | p ∈ Gi, x1 ⊴ p, x2 ⊴ p}。例如在图 3 的泛化层次结构中，d（“Thrift”, “Pigeon”) = d(“RPC”, “Thrift”) + d(“RPC”, “Pigeon”) = 1 + 1 = 2。

对于两个报警 a1、a2，其计算方式为：

$$d(\mathbf{a_1}, \mathbf{a_2}) := \sum_{i=1}^{n} d(\mathbf{a_1}[A_i], \mathbf{a_2}[A_i])$$

公式 1

例如：a1 = (“server_room_b-biz_tag-offline02”, “Thrift”), a2 = (“server_room_a-biz_tag-online01”, “Pigeon”), 则 d(a1, a2) = d(“server_room_b-biz_tag-offline02”, “server_room_a-biz_tag-online01”) + d((“Thrift”, “Pigeon”) = d(“server_room_b-biz_tag-offline02”, “服务器”) + d(“server_room_a-biz_tag-online01”, “服 务 器 ”) + d(“RPC”, “Thrift”) + d(“RPC”, “Pigeon”) = 2 + 2 + 1 + 1 = 6。

我们用 C 表示报警集合，g 是 C 的一个泛化表示，即满足 ∀ a ∈ C, a ⊴ g。以报警集合{“dx-trip-package-api02 Thrift get deal list error.”, “dx-trip-package-api01 Thrift get deal list error.”}为例，“dx 服务器 thrift 调用 获取产品信息失败”是一个泛化表示，“服务器 网络调用 获取产品信息失败”也是一个泛化表

示。对于某个报警聚类来说，我们希望获得既能够涵盖它的集合又有最具象化的表达的泛化表示。为了解决这个问题，定义以下两个指标：

$$\bar{d}(\mathbf{g},\mathcal{C}) \;:=\; 1/|\mathcal{C}| \times \sum_{\mathbf{a}\in\mathcal{C}} d(\mathbf{g},\mathbf{a})$$

$$H(\mathcal{C}) \;:=\; \min\{\,\bar{d}(\mathbf{g},\mathcal{C}) \,|\, \mathbf{g} \in \mathbf{X}_{i=1}^{n}\mathrm{Dom}(A_i),\; \forall \mathbf{a} \in \mathcal{C} : \mathbf{a} \trianglelefteq \mathbf{g}\,\}$$

公式 2

H© 值最小时对应的 g，就是我们要找的最适合的泛化表示，我们称 g 为 C 的“覆盖”(Cover)。

基于以上的概念，将报警日志聚类问题定义为：定义 L 为一个日志集合，min_size 为一个预设的常量，Gi(i = 1, 2, 3……n) 为属性 Ai 的泛化层次结构，目标是找到一个 L 的子集 C，满足 |C| >= min_size，且 H© 值最小。min_size 是用来控制抽象程度的，极端情况下如果 min_size 与 L 集合的大小一样，那么我们只能使用终极抽象了，而如果 min_size = 1，则每个报警日志是它自己的抽象。找到一个聚类之后，我们可以去除这些元素，然后在 L 剩下的集合里找其他的聚类。

不幸的是，这是个 NP 完全问题，因此论文提出了一种启发式算法，该算法满足 |C| >= min_size，使 H© 值尽量小。

算法描述

1. 算法假设所有的泛化层次结构 Gi 都是树，这样每个报警集群都有一个唯一的、最顶层的泛化结果。
2. 将 L 定义为一个原始的报警日志集合，算法选择一个属性 Ai，将 L 中所有报警的 Ai 值替换为 Gi 中 Ai 的父值，通过这一操作不断对报警进行泛化。
3. 持续步骤 2 的操作，直到找到一个覆盖报警数量大于 min_size 的泛化报警为止。
4. 输出步骤 3 中找到的报警。

算法伪代码如下所示：

```
输入：报警日志集合 L，min_size，每个属性的泛化层次结构 G1,.....,Gn
输出：所有符合条件的泛化报警
T := L;                // 将报警日志集合保存至表 T
for all alarms a in T do
    a[count] := 1;    // "count" 属性用于记录 a 当前覆盖的报警数量
while ∀a ∈ T : a[count] < min_size do {
    使用启发算法选择一个属性 Ai;
    for all alarms a in T do
        a[Ai] := parent of a[Ai] in Gi;
        while identical alarms a, a' exist do
            Set a[count] := a[count] + a'[count];
            delete a' from T;
}
```

其中第 7 行的启发算法为：

```
首先计算 Ai 对应的 Fi
fi(v) := SELECT sum(count) FROM T WHERE Ai = v    // 统计在 Ai 属性上值为 v 的报警的数量
Fi := max{fi(v) | v ∈ Dom(Ai)}
选择 Fi 值最小的属性 Ai
```

这里的逻辑是：如果有一个报警 a 满足 a[count]>= min_size，那么对于所有属性 Ai，均能满足 Fi >= fi(a[Ai]) >= min_size。反过来说，如果有一个属性 Ai 的 Fi 值小于 min_size，那么 a[count] 就不可能大于 min_size。所以选择 Fi 值最小的属性 Ai 进行泛化，有助于尽快达到聚类的条件。

此外，关于 min_size 的选择，如果选择了一个过大的 min_size，那么会迫使算法合并具有不同根源的报警。另一方面，如果过小，那么聚类可能会提前结束，具有相同根源的报警可能会出现在不同的聚类中。

因此，设置一个初始值，可以记作 ms0。定义一个较小的值 $\varepsilon(0 < \varepsilon < 1)$，当 min_size 取值为 ms0、ms0 * (1 − ε)、ms0 * (1 + ε) 时的聚类结果相同时，我们就说此时聚类是 ε− 鲁棒的。如果不相同，则使 ms1 = ms0 * (1 − ε)，重复这个测试，直到找到一个鲁棒的最小值。

需要注意的是，ε− 鲁棒性与特定的报警日志相关。因此，给定的最小值，可能

相对于一个报警日志来说是鲁棒的，而对于另一个报警日志来说是不鲁棒的。

实现

1. 提取报警特征

根据线上问题排查的经验，运维人员通常关注的指标包括时间、机器（机房、环境）、异常来源、报警日志文本提示、故障所在位置（代码行数、接口、类）、Case 相关的特殊 ID（订单号、产品编号、用户 ID 等等）等。

但是，我们的实际应用场景都是线上准实时场景，时间间隔比较短，因此我们不需要关注时间。同时，Case 相关的特殊 ID 不符合我们希望获得一个抽象描述的要求，因此也无需关注此项指标。

综上，我们选择的特征包括：机房、环境、异常来源、报警日志文本关键内容、故障所在位置（接口、类）共 5 个。

2. 算法实现

（1）提取关键特征

我们的数据来源是日志中心已经格式化过的报警日志信息，这些信息主要包含：报警日志产生的时间、服务标记、在代码中的位置、日志内容等。

- 故障所在位置

优先查找是否有异常堆栈，如存在则查找第一个本地代码的位置；如果不存在，则取日志打印位置。

- 异常来源

获得故障所在位置后，优先使用此信息确定异常报警的来源（需要预先定义词典支持）；如不能获取，则在日志内容中根据关键字匹配（需要预先定义词典支持）。

- 报警日志文本关键内容

优先查找是否有异常堆栈，如存在，则查找最后一个异常（通常为真正的故障原因）；如不能获取，则在日志中查找是否存在“code=……,message=……”这样形式的错误提示；如不能获取，则取日志内容的第一行内容（以换行符为界），并去除其中可能存在的 Case 相关的提示信息

- 提取“机房和环境”这两个指标比较简单，在此不做赘述。

（2）聚类算法

算法的执行，我们以图 4 来表示。

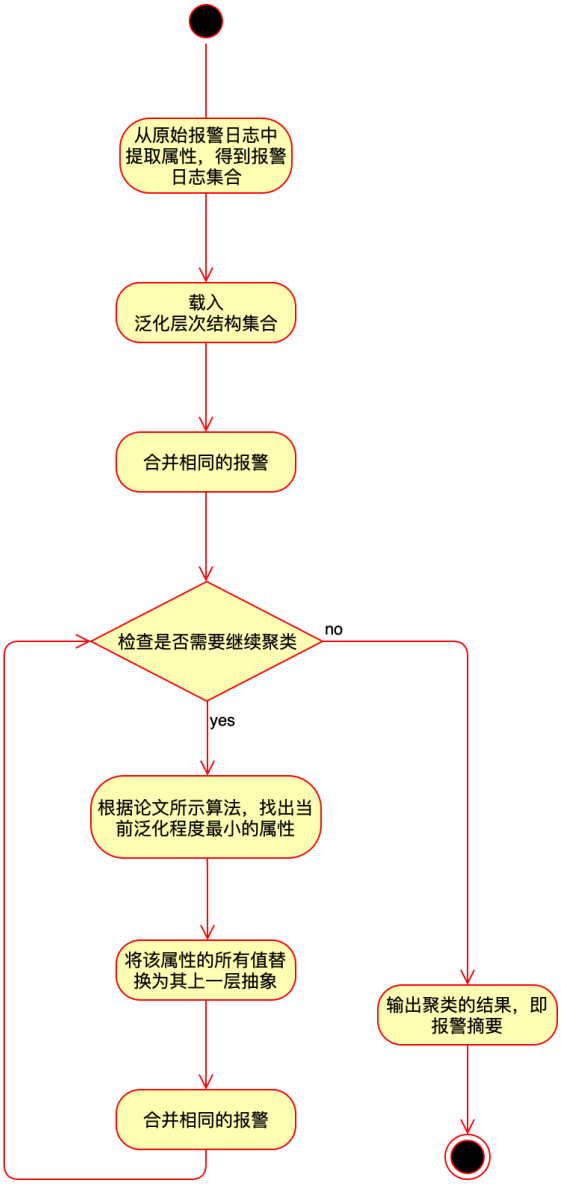


图 4　报警日志聚类流程图

（3）min_size 选择

考虑到日志数据中可能包含种类极多，且根据小规模数据实验表明，min_size = 1/5 * 报警日志数量时，算法已经有较好的表现，再高会增加过度聚合的风险，因此我们取 min_size = 1/5 * 报警日志数量，ε 参考论文中的实验，取 0.05。

（4）聚类停止条件

考虑到部分场景下，报警日志可能较少，因此 min_size 的值也较少，此时聚类已无太大意义，因此设定聚类停止条件为：聚类结果的报警摘要数量小于等于 20 或已经存在某个类别的 count 值达到 min_size 的阈值，即停止聚类。

3. 泛化层次结构

泛化层次结构，用于记录属性的泛化关系，是泛化时向上抽象的依据，需要预先定义。

根据实验所用项目的实际使用环境，我们定义的泛化层次结构如下：

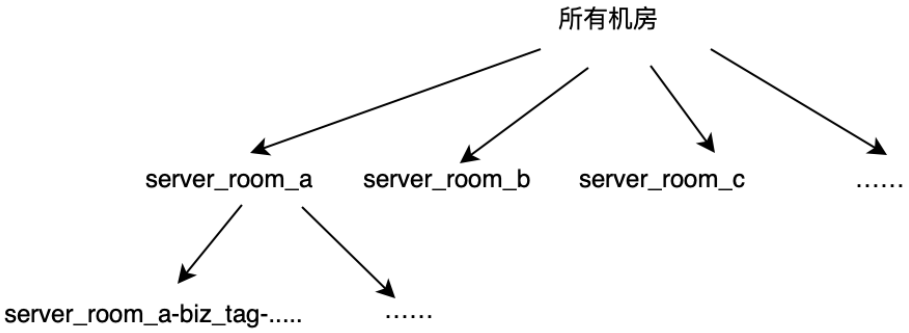


图 5　机房泛化层次结构

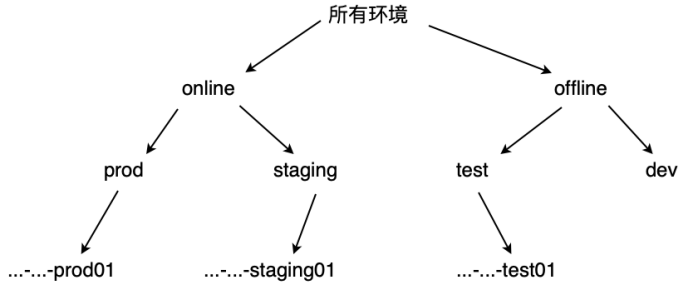


图 6　环境泛化层次结构

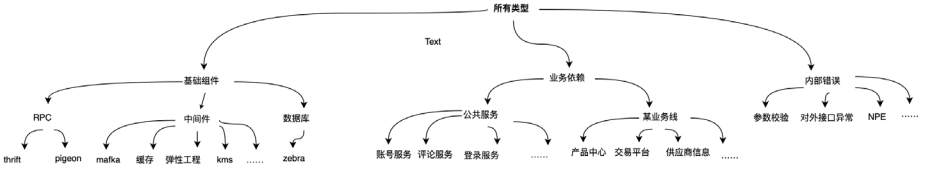


图 7　错误来源泛化层次结构

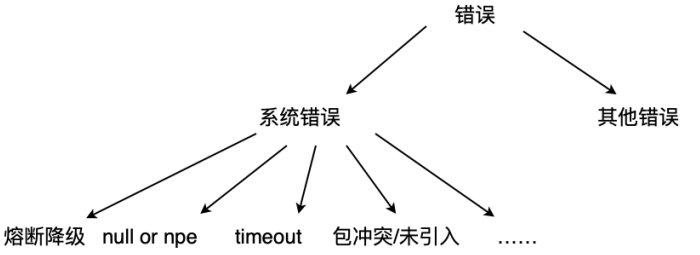


图 8　日志文本摘要泛化层次结构

“故障所在位置”此属性无需泛化层次结构，每次泛化时直接按照包路径向上层截断，直到系统包名。

实验

以下三个实验均使用 C 端 API 系统。

1. 单依赖故障

实验材料来自于线上某业务系统真实故障时所产生的大量报警日志。

- 环境：线上
- 故障原因：产品中心线上单机故障
- 报警日志数量：939 条

部分原始报警日志如图 9 所示，初次观察时，很难理出头绪。

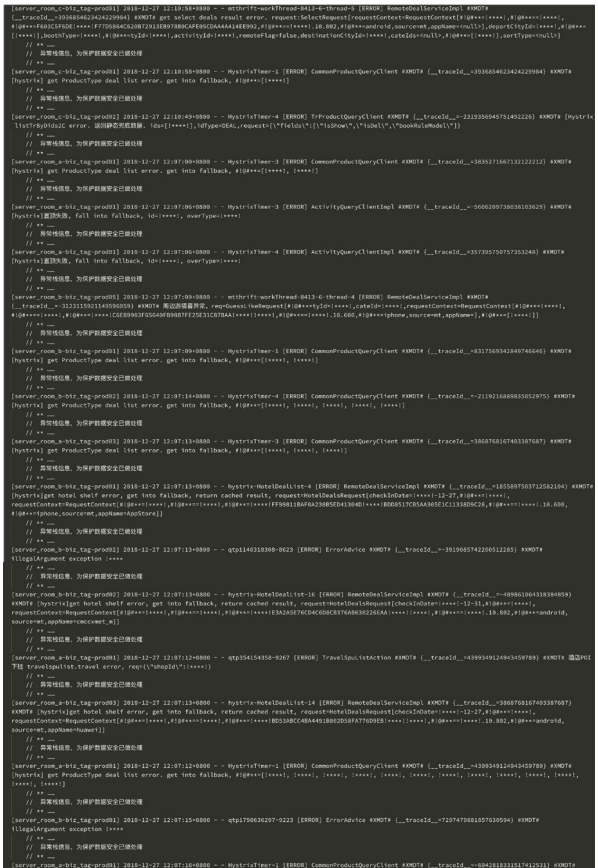


图 9　单依赖故障报警日志节选

经过聚类后的报警摘要如表 1 所示：

ID	Server Room	Error Source	Environment	Position（为保证数据安全，类路径已做处理）	Summary（为保证数据安全，部分类路径已做处理）	Count
1	所有机房	产品中心	Prod	com.*.*.*.CommonProductQueryClient	com.netflix.hystrix.exception.HystrixTimeoutException: commonQueryClient.getProductType execution timeout after waiting for 150ms.	249
2	所有机房	业务插件	Prod	com.*.*.*.PluginRegistry.lambda	java.lang.IllegalArgumentException: 未找到业务插件 : 所有产品类型	240
3	所有机房	产品中心	Prod	com.*.*.*.TrProductQueryClient	com.netflix.hystrix.exception.HystrixTimeoutException: TrQueryClient.listTrByDids2C execution timeout after waiting for 1000ms.	145
4	所有机房	对外接口（猜喜 / 货架 / 目的地）	Prod	com.*.*.*.RemoteDealServiceImpl	com.netflix.hystrix.exception.HystrixTimeoutException: ScenicDealList.listDealsByScenic execution timeout after waiting for 300ms.	89
5	所有机房	产品中心	Prod	com.*.*.*.CommonProductQueryClient	com.netflix.hystrix.exception.HystrixTimeoutException: commonQueryClient.listTrByDids2C execution timeout after waiting for 1000ms.	29
6	所有机房	产品中心	Prod	com.*.*.*.ActivityQueryClientImpl	com.netflix.hystrix.exception.HystrixTimeoutException: commonQueryClient.getBusinessLicense execution timeout after waiting for 100ms.	21

ID	Server Room	Error Source	Environment	Position（为保证数据安全，类路径已做处理）	Summary（为保证数据安全，部分类路径已做处理）	Count
7	所有机房	产品中心	prod	com.*.*.*.CommonProductQueryClient	com.netflix.hystrix.exception.HystrixTimeoutException: commonQueryClient.getBusinessLicense execution timeout after waiting for 100ms.	21
8	所有机房	对外接口（猜喜 / 货架 / 目的地）	Prod	com.*.*.*.RemoteDealServiceImpl	com.netflix.hystrix.exception.HystrixTimeoutException: HotelDealList.hotelShelf execution timeout after waiting for 500ms.	17
9	所有机房	产品中心	Prod	com.*.*.*.TrProductQueryClient	Caused by: java.lang.InterruptedException	16
10	所有机房	产品中心	Prod	com.*.*.*.TrProductQueryClient	Caused by: java.lang.InterruptedException	13

我们可以看到前三条报警摘要的 Count 远超其他报警摘要，并且它们指明了故障主要发生在产品中心的接口。

2. 无相关的多依赖同时故障

实验材料为利用故障注入工具，在 Staging 环境模拟运营置顶服务和 A/B 测试服务同时产生故障的场景。

- 环境：Staging（使用线上录制流量和压测平台模拟线上正常流量环境）
- 模拟故障原因：置顶与 A/B 测试接口大量超时
- 报警日志数量：527 条

部分原始报警日志如图 10 所示：

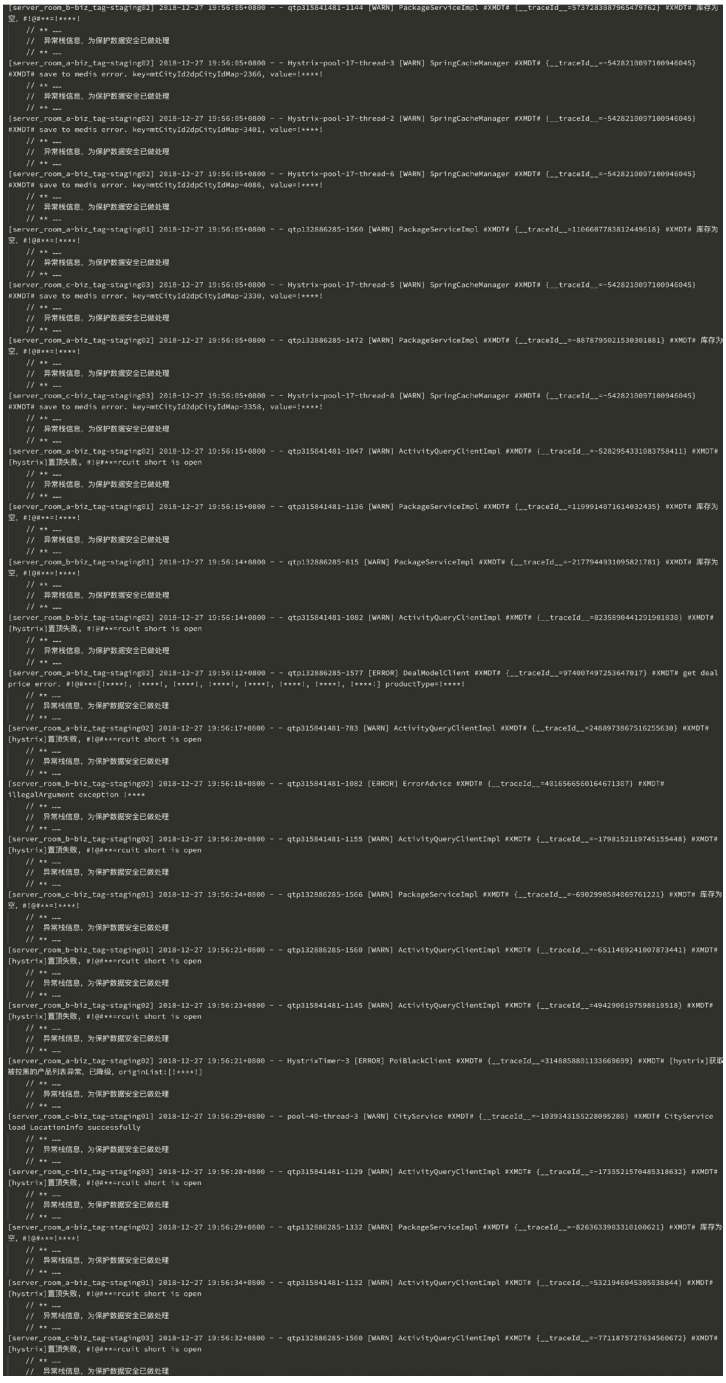


图 10　无相关的多依赖同时故障报警日志节选

经过聚类后的报警摘要如表 2 所示：

ID	Server Room	Error Source	Environment	Position（为保证数据安全，类路径已做处理）	Summary（为保证数据安全，部分类路径已做处理）	Count
1	所有机房	运营活动	Staging	com.*.*.*.ActivityQueryClientImpl	[hystrix] 置顶失败 , circuit short is open	291
2	所有机房	A/B 测试	Staging	com.*.*.*.AbExperimentClient	[hystrix] tripExperiment error, circuit short is open	105
3	所有机房	缓存	Staging	com.*.*.*.CacheClientFacade	com.netflix.hystrix.exception.HystrixTimeoutException: c-cache-rpc.common_deal_base.rpc execution timeout after waiting for 1000ms.	15
4	所有机房	产品信息	Staging	com.*.*.*.queryDealModel	Caused by: com.meituan.service.mobile.mtthrift.netty.exception.RequestTimeoutException: request timeout	14
5	所有机房	产品中心	Staging	com.*.*.*.CommonProductQueryClient	com.netflix.hystrix.exception.HystrixTimeoutException: commonQueryClient.getBusinessLicense execution timeout after waiting for 100ms.	9
6	所有机房	产品中心	Staging	com.*.*.*.getOrderForm	java.lang.IllegalArgumentException: 产品无库存	7
7	所有机房	弹性工程	Staging	com.*.*.*.PreSaleChatClient	com.netflix.hystrix.exception.HystrixTimeoutException: CustomerService.PreSaleChat execution timeout after waiting for 50ms.	7
8	所有机房	缓存	Staging	com.*.*.*.SpringCacheManager	Caused by: java.net.SocketTimeoutException: Read timed out	7
9	所有机房	产品信息	Staging	com.*.*.*.queryDetailUrlVO	java.lang.IllegalArgumentException: 未知的产品类型	2
10	所有机房	产品信息	Staging	com.*.*.*.queryDetailUrlVO	java.lang.IllegalArgumentException: 无法获取链接地址	1

从上表可以看到，前两条报警摘要符合本次试验的预期，定位到了故障发生的原因。说明在多故障的情况下，算法也有较好的效果。

3. 中间件与相关依赖同时故障

实验材料为利用故障注入工具，在 Staging 环境模拟产品中心服务和缓存服务同时产生超时故障的场景。

- 环境：Staging（使用线上录制流量和压测平台模拟线上正常流量环境）
- 模拟故障原因：产品中心所有接口超时，所有缓存服务超时
- 报警日志数量：2165

部分原始报警日志如图 11 所示：

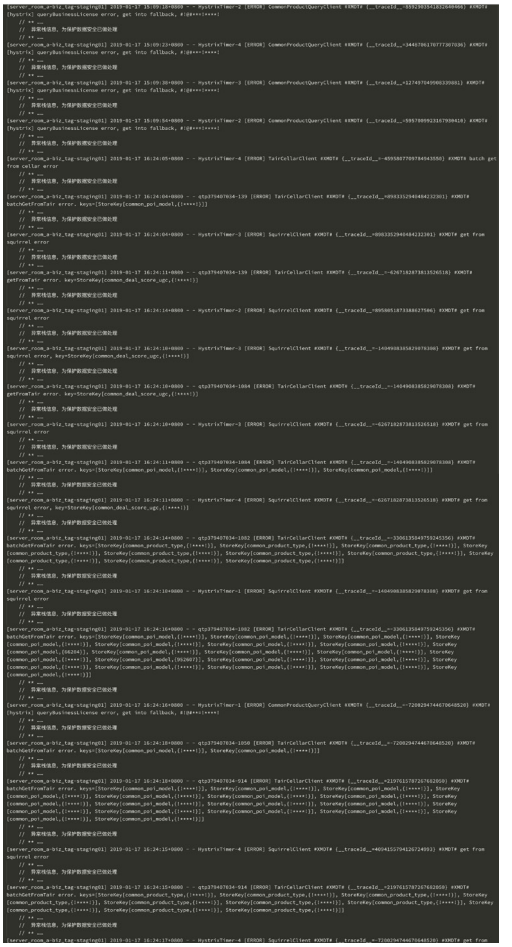


图 11　中间件与相关依赖同时故障报警日志节选

经过聚类后的报警摘要如表 3 所示：

ID	Server Room	Error Source	Environment	Position（为保证数据安全，类路径已做处理）	Summary（为保证数据安全，部分类路径已做处理）	Count
1	所有机房	Squirrel	Staging	com.*.*.*.cache	Timeout	491
2	所有机房	Cellar	Staging	com.*.*.*.cache	Timeout	285
3	所有机房	Squirrel	Staging	com.*.*.*.TdcServiceImpl	Other Exception	149
4	所有机房	评论	Staging	com.*.*.*.cache	Timeout	147
5	所有机房	Cellar	Staging	com.*.*.*.TdcServiceImpl	Other Exception	143
6	所有机房	Squirrel	Staging	com.*.*.*.PoiManagerImpl	熔断	112
7	所有机房	产品中心	Staging	com.*.*.*.CommonProductQueryClient	Other Exception	89
8	所有机房	评论	Staging	com.*.*.*.TrDealProcessor	Other Exception	83
9	所有机房	评论	Staging	com.*.*.*.poi.PoiInfoImpl	Other Exception	82
10	所有机房	产品中心	Staging	com.*.*.*.client	Timeout	74

从上表可以看到，缓存（Squirrel 和 Cellar 双缓存）超时最多，产品中心的超时相对较少，这是因为我们系统针对产品中心的部分接口做了兜底处理，当超时发生时后先查缓存，如果缓存查不到会穿透调用一个离线信息缓存系统，因此产品中心超时总体较少。

综合上述三个实验得出结果，算法对于报警日志的泛化是具有一定效果。在所进行实验的三个场景中，均能够定位到关键问题。但是依然存在一些不足，报警摘要中，有的经过泛化的信息过于笼统（比如 Other Exception）。

经过分析，我们发现主要的原因有：其一，对于错误信息中关键字段的提取，在一定程度上决定了向上泛化的准确度。其二，系统本身日志设计存在一定的局限性。

同时，在利用这个泛化后的报警摘要进行分析时，需要使用者具备相应领域的知识。

未来规划

本文所关注的工作，主要在于验证聚类算法效果，还有一些方向可以继续完善和优化：

1. 日志内容的深度分析。本文仅对报警日志做了简单的关键字提取和人工标记，未涉及太多文本分析的内容。我们可以通过使用文本分类、文本特征向量相似度等，提高日志内容分析的准确度，提升泛化效果。
2. 多种聚类算法综合使用。本文仅探讨了处理系统错误日志时表现较好的聚类算法，针对系统中多种不同类型的报警，未来也可以配合其他聚类算法（如K-Means）共同对报警进行处理，优化聚合效果。
3. 自适应报警阈值。除了对报警聚类，我们还可以通过对监控指标的时序分析，动态管理报警阈值，提高告警的质量和及时性，减少误报和漏告数量。

参考资料

1. Julisch, Klaus. “Clustering intrusion detection alarms to support root cause analysis.” ACM transactions on information and system security (TISSEC) 6.4 (2003): 443-471.
2. https://en.wikipedia.org/wiki/Cluster_analysis

作者简介

刘琀，美团点评后端工程师。2017 年加入美团点评，负责美团点评境内度假的业务开发。
干钊，美团点评后端工程师。2017 年加入美团点评，负责美团点评境内度假的业务开发。

全链路压测自动化实践

欧龙

背景与意义

境内度假是一个低频、与节假日典型相关的业务，流量在节假日较平日会上涨五到十几倍，会给生产系统带来非常大的风险。因此，在 2018 年春节前，我们把整个境内度假业务接入了全链路压测，来系统性地评估容量和发现隐患，最终确保了春节期间系统的稳定。

在整个过程中，我们意识到，全链路压测在整个系统稳定性建设中占有核心重要的位置，也是最有效的方案。结合实际业务节假日的频率（基本平均一个月一次），如果能够把它作为稳定性保障的常规手段，我们的系统质量也能够得到很好的保障。同时，为了解决周期常态化压测过程中人力成本高、多个团队重复工作、压测安全不可控，风险高等痛点，我们提出了全链路压测自动化的设想。

通过对压测实施的具体动作做统一的梳理，在压测各个阶段推进标准化和自动化，尽力提升全流程的执行效率，最终达到常态化的目标，如下图 1 所示：

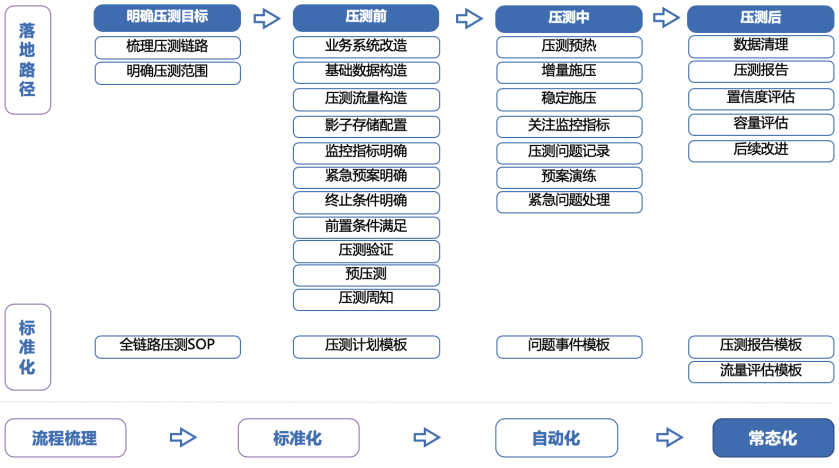


图 1　自动化落地整体思路

另外，在全链路压测的整个周期中，压测安全和压测有效性也是需要一直关注的质量属性。基于这些思考，如下图 2 所示，我们把压测自动化需要解决的关键问题进行了归类和分解：

- 基础流程如何自动化，提高人效；
- 如何自动做好压测验证，保障压测安全；
- 压测置信度量化如何计算，保证压测有效。

图 2　问题分析

最终，基于美团基础的压测平台（Quake 在整个系统，主要提供流量录制、回放、施压的功能），设计并实现了全链路自动化压测系统，为不同业务实施全链路压测提效，并确保压测安全。该系统：

- 提供链路梳理工具，能够自动构建压测入口链路完整的依赖信息，辅助链路梳理；
- 支持链路标注和配置功能，对于无需压测触达的依赖接口，可以通过配置化手段，完成相关接口的 Mock 配置，不用在业务代码中嵌入压测判断逻辑；
- 提供抽象的数据构造接口，通过平台，用户可以配置任意的数据构造逻辑和流程；
- 在压测前 / 压测中，自动对压测服务和流量做多项校验，保障压测安全性；
- 在平日，基于压测计划提供周期性小流量的压测校验，使得业务迭代变更带来

的压测安全风险被尽早发现；

- 提供压测计划管理功能，通过系统自动调度和控制施压过程，解放人力；同时强制前置预压测，也提高了安全性；
- 一键压测，自动生成报告，收集链路入口和告警信息，提供问题记录和跟进功能。

系统设计

系统总体设计

图 3　系统总体逻辑架构

系统的总体逻辑架构，如图 3 所示，主要包括链路构建 / 比对、事件 / 指标收集、链路治理、压测配置管理、压测验证检查、数据构造、压测计划管理、报告输出等功能模块。通过这些模块，为全链路压测的整个流程提供支持，尽力降低业务部门使用全链路压测的门槛和成本。

链路构建 / 比对：负责服务接口方法调用链路的构建、更新、存储。

链路治理：基于构建的链路关系，提供链路中核心依赖、出口 Mock 接口等标注、上下游分析、展示，以及出口 Mock 的配置等功能。

压测配置管理：自动发现注册服务的 Mafka/Cellar/Squirrel/Zebra 的压测配置，

辅助压测方核查和配置相关配置项。

压测验证检查：确保系统可压测，通过多种校验手段和机制设计，来保证压测的安全性。

数据构造：为不同业务压测实施准备基础和流量数据。

压测计划管理：设定压测执行计划，并依赖“压测控制”模块，自动调度整个压测执行过程。

故障诊断：依据收集的关键业务 / 服务指标、报警等信息，判断分析服务是否异常，以及是否终止压测。

置信度评估：从数据覆盖、链路覆盖、技术指标等维度评估压测结果的置信度，即与真实流量情况下各评估维度的相似性。

非功能性需求说明：

- 可扩展性
  - 能够兼容不同业务线数据构造逻辑的差异性。
  - 能够支持不同的流量录制方式。
- 安全性
  - 集成 SSO，按用户所属团队分组，展示所属的压测服务信息。对关键操作留存操作日志。
  - 压测验证检查，是确保压测安全的关键。支持周期性压测验证，能发现待压测服务可压测性随时间的退化。
- 可重用性
  - 长远看，链路构建、事件 / 指标收集 / 故障诊断等模块，在稳定性领域是可重用的基础设施，按独立通用模块建设。

约束说明：

- 基于 Quake 搭建，流量的录制、回放、施压等依赖 Quake。

以下对部分关键模块设计做详细介绍。

链路治理模块设计

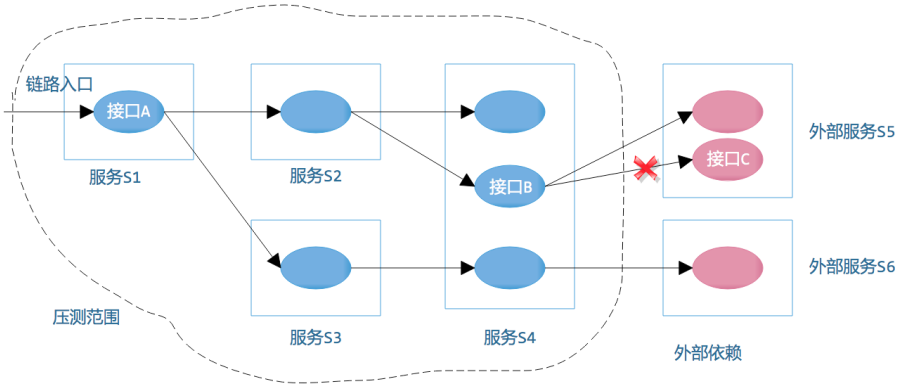


图 4　链路治理示意图

链路治理模块是基于链路构建模块实现的。链路构建模块，底层是以闭包表的方式存储两个维度（服务和接口）的链路关系的，会周期自动地构建或更新。

链路治理模块主要提供链路入口选取、链路标注、服务出口分析、出口 Mock 配置等功能。如图 4 所示，注册压测的服务构成了压测服务的范围，也就确定了各个链路的边界。通过系统自动构建的树结构方式的链路关系，可以辅助压测方对整个链路的梳理，它解决了以往链路梳理靠翻代码等低效手段，缺少全链路视角无法做到完备梳理等问题。

图 5　出口 Mock 配置化

同时，针对整个压测范围，依赖接口可以做人工标注。哪些需要 Mock，哪些不需要 Mock，如此压测特有的链路信息能够得到持续的维护。

对于需要 Mock 的外部接口（如图 4 中的接口 C），待压测系统通过引入专有 SDK 的方式，获得出口配置化 Mock 的能力。如图 5 所示，这里使用了美团酒旅 Mock 平台的基础能力，采用 JVM-Sandbox 作为 AOP 工具，对配置的需要 Mock 的外部接口做动态能力增强。在接口调用时，判断是否是压测流量，是的话走 Mock 逻辑，做模拟时延处理，返回提前配置的响应数据。这样的话，第一，简化了出口 Mock 的操作，业务代码里 Mock 逻辑 0 侵入；第二，把之前本地 Mock 与借助 Mockserver 的两种解决方案用一种方案替代，便于统一管理；第三，在实际压测时，平台还可以通过 SDK 收集 Mock 逻辑执行的数据，自动与后台标注的 Mock 数据对比，来确保应该被 Mock 的出口确实被 Mock 掉。

数据构造模块设计

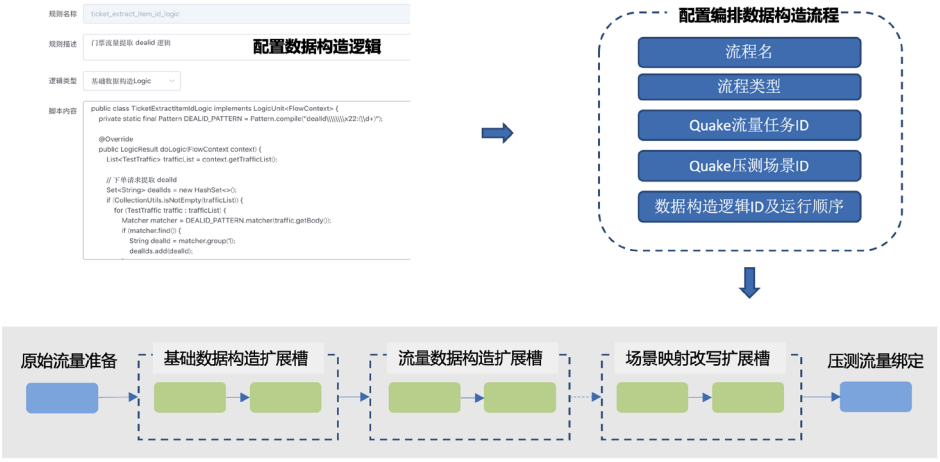


图 6　数据构造

数据构造模块是为了解决不同业务对于基础数据和流量数据的差异化构造流程。提出了两个关键的概念：数据构造逻辑和数据构造流程。数据构造逻辑，是数据构造的细粒度可复用的基本单元，由一段 Java 代码表示。平台提供统一抽象的数据构造接口，基于 Java 动态编译技术，开发了一个 Java 版的脚本引擎，支持构造逻辑的

在线编辑与更新。同时，基于美团 RPC 中间件泛化调用能力，构建了泛化调用工具，帮助用户把外部基础数据构造接口的调用集成到一个数据构造逻辑中。

数据构造流程，定义了压测基础数据和流量数据生成的整个流程。通过与 Quake 的交互，获取原始真实的线上数据；构建了一个简版的流程引擎，在统一设定的流程中，如图 6 所示，通过在标准扩展槽中，配置不同类型的数据构造逻辑和执行顺序，来定义整个数据构造执行的流程；最后，把构造的流量数据与 Quake 压测场景绑定，作为后续 Quake 压测施压中，场景回放流量的来源。

通过这样的设计，能够支持任意数据构造逻辑，通用灵活。同时集成了 Quake 已有的流量录制功能，一键执行数据构造流程，大大地提升了效率。

压测验证模块设计

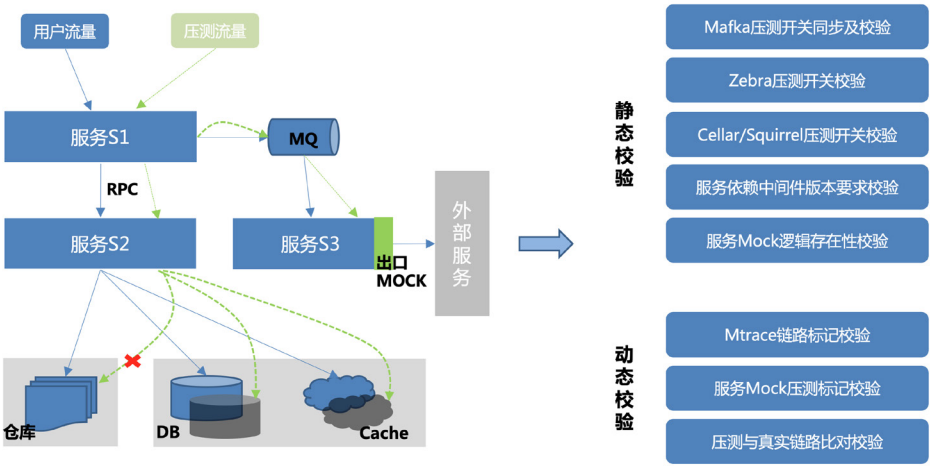


图 7　美团服务压测验证示意

对于压测安全性的保障，一直是自动化的难点。之前的经验多是在非生产环境压测或预压测过程中，依靠不同服务相关负责人的人工确认。这里针对压测验证，提供两条新的思考角度：一个是从待压测服务系统可压测性的角度看；一个是从压测流量特征的角度看。对于第一个角度，一个服务支持压测需要满足压测数据和流量的隔离。对于不同的系统生态，需要满足的点是不同的，对于美团生态下的服务，可压测的条件包括组件版本支持压测、影子存储配置符合预期等等。

从这些条件出发，就可以得到下面这些静态的校验项：

- 服务依赖中间件版本要求校验；
- Zebra 压测配置校验；
- Cellar/Squirrel 压测配置校验；
- Mafka 压测开关同步及校验；
- 服务 Mock 逻辑存在性校验。

而从第二个角度来看，就是关注压测流量下会产生哪些特有的流量特征数据，通过这些特有的数据来确保压测的安全性。这里主要有三类数据：美团分布式追踪系统（MTrace）中调用链路的压测标记数据（正常的压测链路应该是一直带有压测标记，直到压测范围的边界节点，可参考图 4）；标记 Mock 的外部接口被调用时，上报的运行数据；基于监控系统得到的压测流量特有的监控数据。利用这些数据，我们设计了三种动态的校验项，发现压测标记丢失、Mock 出口被调用等异常情况：

- MTrace 链路标记校验，从压测链路入口出发，收集压测链路信息，校验压测标记信息传递是否符合预期。

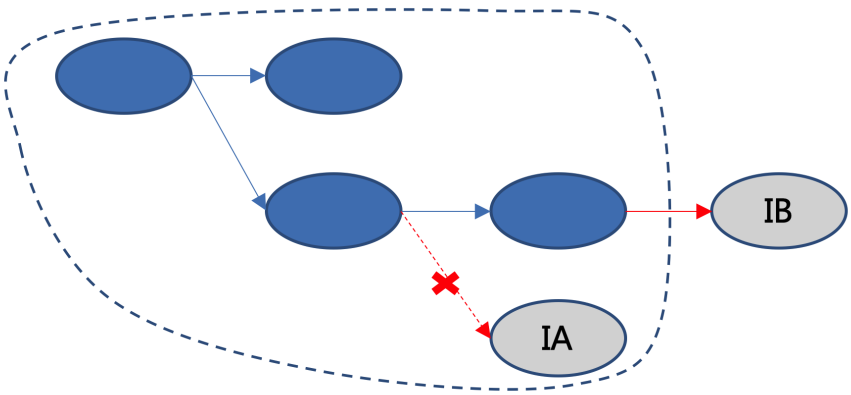


图 8　MTrace 链路标记校验示意

- 服务 Mock 逻辑压测标记校验，通过增强的校验逻辑，把执行信息上报到平

台，与 Mock 配置时的标注数据对比验证。

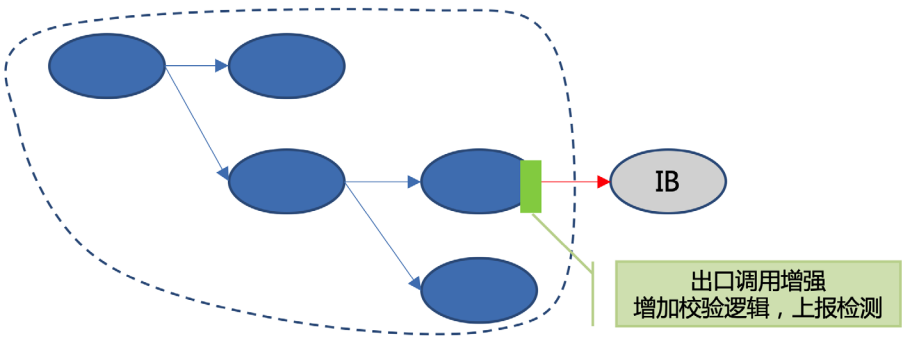


图 9　服务 Mock 压测校验示意

- 压测与真实链路比对校验，利用链路治理模块构建链路的能力，采集压测监控数据重构链路，与真实链路对比验证。

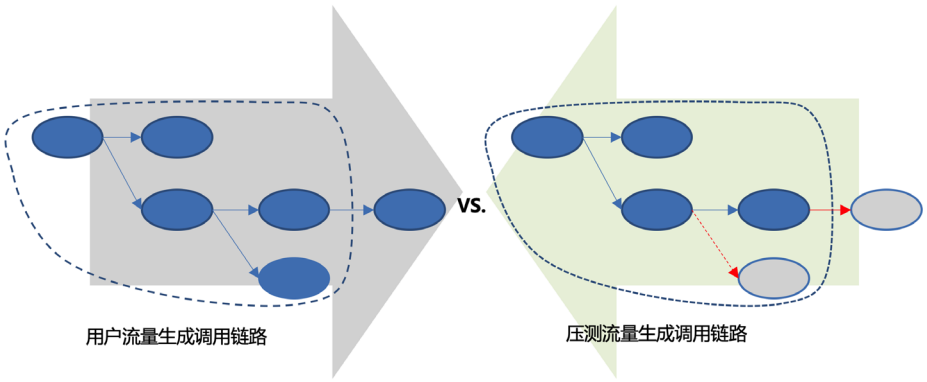


图 10　压测与真实链路对比示意

除了明确静态和动态两类压测校验规则，在具体流程安排上，在压测时和平日两个时期执行这些规则。既能把压测校验的压力分散到平时，也能尽快地发现服务因代码迭代引入的新风险。

在压测时，通过强制前置预压测的流程设计以及静态 / 动态压测校验项的自动执行，保障安全这个事情。校验不通过，给出告警，甚至在允许的情况下直接终止设定

的压测计划。

在平日，通过执行周期性小流量压测校验，在施压过程中对 QPS 做个位数的精细控制，以尽量小的代价快速发现压测范围内压测安全性的退化。

压测计划管理模块设计

压测计划管理模块，提供压测计划的提前设定，然后模块能够自动调度和控制整个施压过程。如图 11 所示，这里的压测计划是多个压测场景的组合，包含 QPS 的增长计划等信息，主要分为预压测和正式压测两个阶段。压测计划的自动实施，能够解决尤其多场景组合压测，操作耗时多、多场景压测 QPS 无法同步变更、压测方无法兼顾操作和观测等问题，提升了效率。同时，在压测计划执行状态机里，预压测正常执行完成，状态才能迁移到正式压测的开始状态，提高了压测安全性。

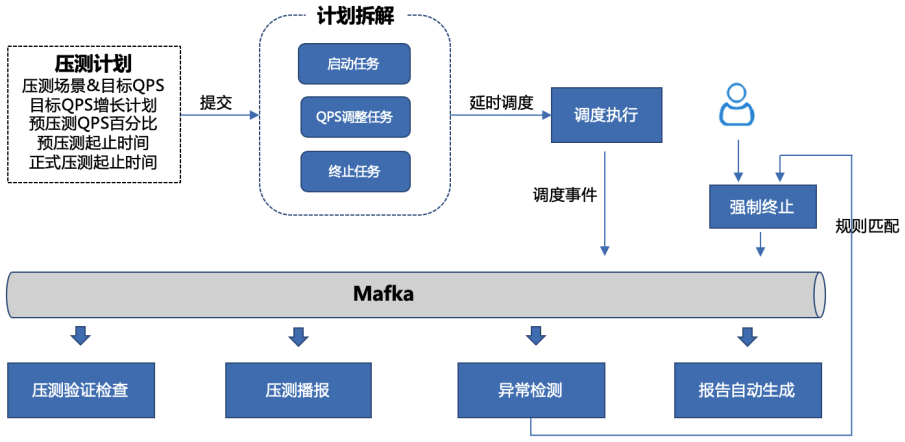


图 11　压测计划执行

从图 11 可以看到，压测计划模块，是整个自动化压测的核心，协同起了各个模块。通过具体的计划任务执行产生的事件，触发了压测验证检查、压测进展播报、收集压测监控 / 告警等数据，来检测服务是否异常，并根据配置来终止压测，能够故障时及时止损。最后，报告生成模块收到压测终止事件，汇总各种信息，自动生成包括压测基本信息等多维度信息的压测报告，节省了一些压测后分析的时间。

案例分享

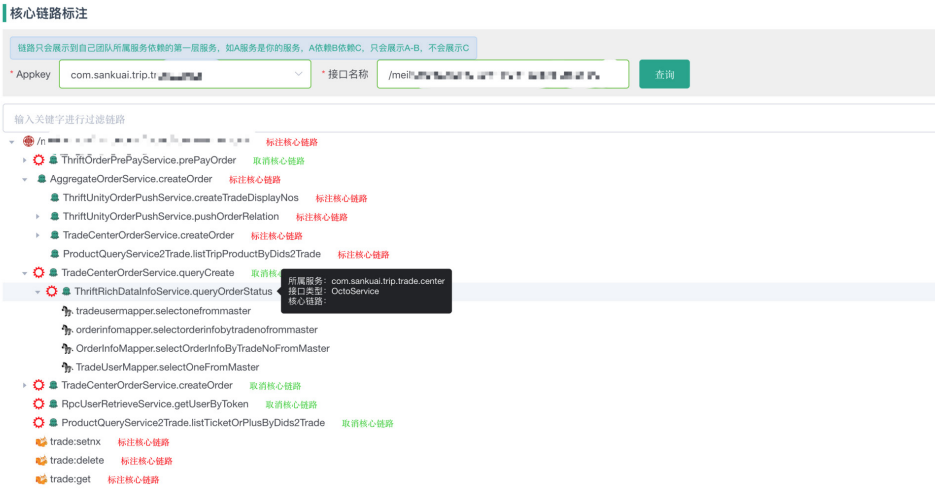
以下以实际压测的过程来做个案例分享。

团队 / 服务注册

- 设定实施压测的虚拟团队和压测覆盖范围的应用服务。

链路治理

- 选定压测链路入口，可以得到入口以下的接口链路关系树，便于梳理。
- 明确需要 Mock 的外部接口，并做配置，参考“链路治理模块设计”一节。

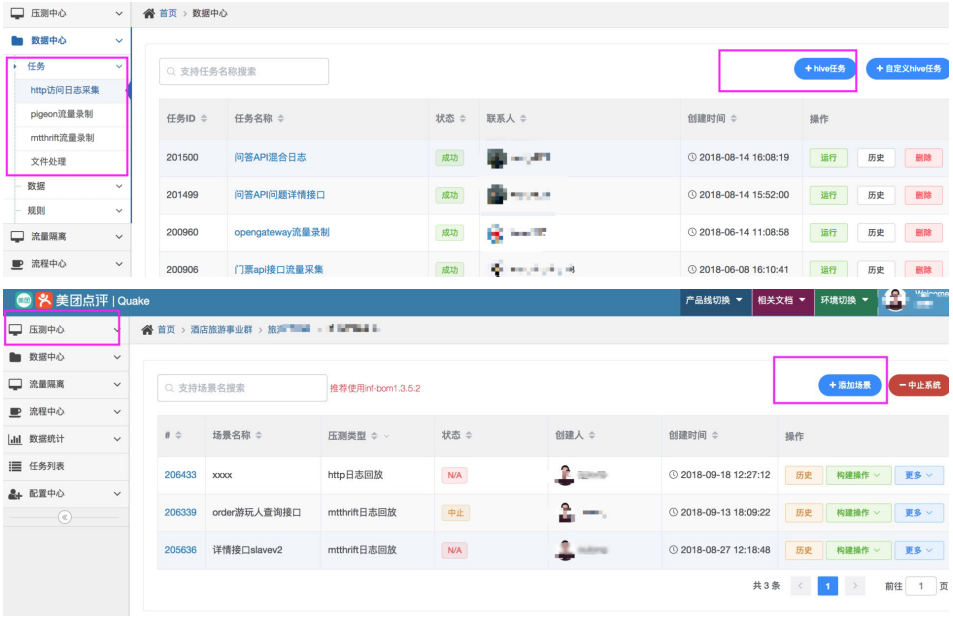


应用改造与压测配置

- 对待接入压测应用改造，满足“服务的可压测条件”，参考图 7。
- 压测应用依赖中间件配置，系统依据构建的链路信息，能够自动发现。提供统一配置和核对的页面功能。

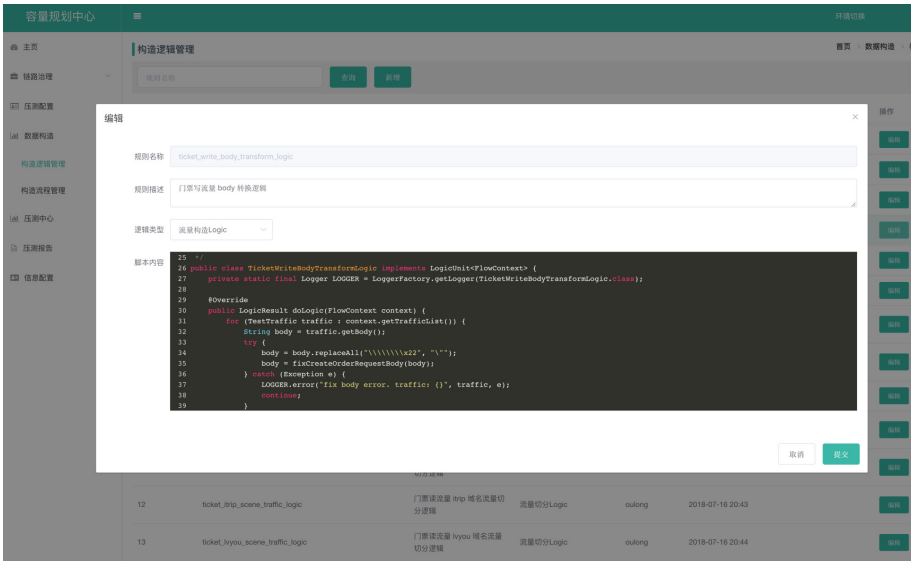
Quake 准备

- 压测自动化系统是基于 Quake 构建的，流量录制、回放、施压等依赖于此。因此需要到 Quake 上配置流量录制的“流量任务”和压测执行的“压测场景”。

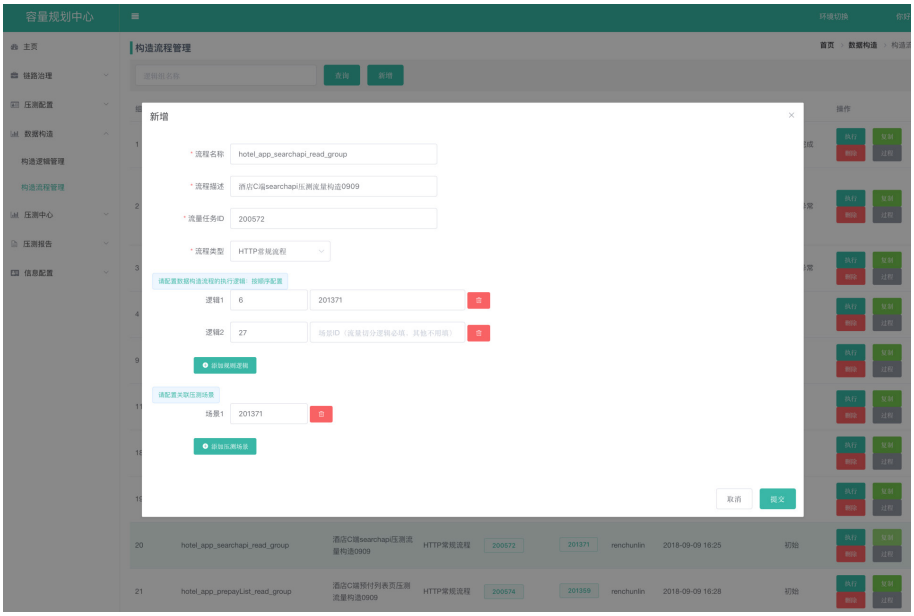


数据构造

- 配置数据构造逻辑，当然已有的逻辑都是可复用的单元，可以先查看已有逻辑是否能满足自己的需要。

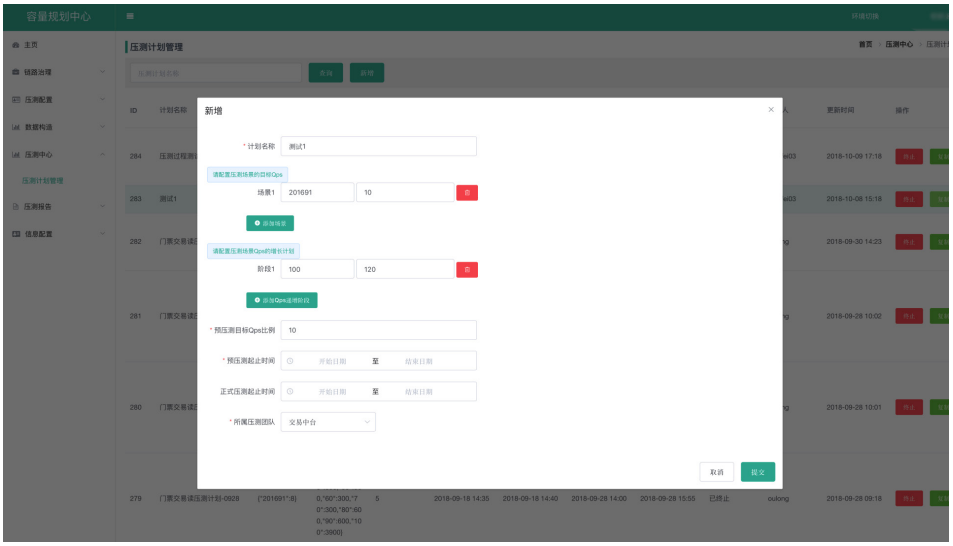


- 配置数据构造流程。

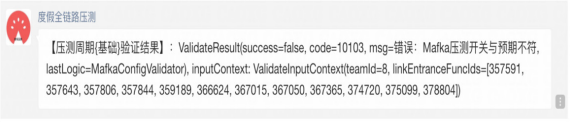


压测实施

- 设定压测计划，到启动时间，系统会自动启动压测。



- 压测中，注意关注压测验证校验的告警信息，及时处理。



- 压测后，可查看压测报告。记录和跟进发现的问题。

总结与展望

目前，压测自动化系统已经投入使用，美团酒店和境内度假的全部团队已经接入，有效地提升了压测效率。后续会在两个大方向上持续建设升级，一个是把全链路压测放到“容量评估与优化”领域来看，不仅关注整体系统的稳定性，同时也期望兼顾成本的平衡；另一个是与稳定性其他子领域的生态集成，比如故障演练、弹性伸缩等等，在更多场景发挥压测的作用。最后，通过这些努力，使得线上系统的稳定性成为一个确定性的事情。

参考资料

[1] 全链路压测平台（Quake）在美团中的实践
[2] 阿里 JVM-Sandbox
[3] Dubbo 的泛化调用
[4] Java 的动态编译

作者简介

欧龙，美团研发工程师，2013 年加入美团，目前主要负责境内度假交易稳定性建设等工作。

降低软件复杂性一般原则和方法

邹政华

一、前言

斯坦福教授、Tcl 语言发明者 John Ousterhout 的著作《A Philosophy of Software Design》[1]，自出版以来，好评如潮。按照 IT 图书出版的惯例，如果冠名为“实践”，书中内容关注的是某项技术的细节和技巧；冠名为“艺术”，内容可能是记录一件优秀作品的设计过程和经验；而冠名为“哲学”，则是一些通用的原则和方法论，这些原则方法论串起来，能够形成一个体系。正如”知行合一”、“世界是由原子构成的”、“我思故我在”，这些耳熟能详的句子能够一定程度上代表背后的人物和思想。用一句话概括《A Philosophy of Software Design》，软件设计的核心在于降低复杂性。

本篇文章是围绕着“降低复杂性”这个主题展开的，很多重要的结论来源于 John Ousterhout，笔者觉得很有共鸣，就做了一些相关话题的延伸、补充了一些实例。虽说是”一般原则“，也不意味着是绝对的真理，整理出来，只是为了引发大家对软件设计的思考。

二、如何定义复杂性

关于复杂性，尚无统一的定义，从不同的角度可以给出不同的答案。可以用数量来度量，比如芯片集成的电子器件越多越复杂 (不一定对)；按层次性 [2] 度量，复杂度在于层次的递归性和不可分解性。在信息论中，使用熵来度量信息的不确定性。

John Ousterhout 选择从认知的负担和开发工作量的角度来定义软件的复杂性，并且给出了一个复杂度量公式：

$$C = \sum_{p} c_p t_p$$

子模块的复杂度 cp 乘以该模块对应的开发时间权重值 tp，累加后得到系统的整体复杂度 C。系统整体的复杂度并不简单等于所有子模块复杂度的累加，还要考虑该模块的开发维护所花费的时间在整体中的占比（对应权重值 tp）。也就是说，即使某个模块非常复杂，如果很少使用或修改，也不会对系统的整体复杂度造成大的影响。

子模块的复杂度 cp 是一个经验值，它关注几个现象：

- 修改扩散，修改时有连锁反应。
- 认知负担，开发人员需要多长时间来理解功能模块。
- 不可知（Unknown Unknowns），开发人员在接到任务时，不知道从哪里入手。

造成复杂的原因一般是代码依赖和晦涩 (Obscurity)。其中，依赖是指某部分代码不能被独立地修改和理解，必定会牵涉到其他代码。代码晦涩，是指从代码中难以找到重要信息。

三、解决复杂性的一般原则

首先，互联网行业的软件系统，很难一开始就做出完美的设计，通过一个个功能模块衍生迭代，系统才会逐步成型；对于现存的系统，也很难通过一个大动作，一劳永逸地解决所有问题。系统设计是需要持续投入的工作，通过细节的积累，最终得到一个完善的系统。因此，好的设计是日拱一卒的结果，在日常工作中要重视设计和细节的改进。

其次，专业化分工和代码复用促成了软件生产率的提升。比如硬件工程师、软件工程师（底层、应用、不同编程语言）可以在无需了解对方技术背景的情况下进行合作开发；同一领域服务可以支撑不同的上层应用逻辑等等。其背后的思想，无非是通过将系统分成若干个水平层、明确每一层的角色和分工，来降低单个层次的复杂性。同时，每个层次只要给相邻层提供一致的接口，可以用不同的方法实现，这就为软件重用提供了支持。分层是解决复杂性问题的重要原则。

第三，与分层类似，分模块是从垂直方向来分解系统。分模块最常见的应用场景，是如今广泛流行的微服务。分模块降低了单模块的复杂性，但是也会引入新的复

杂性，例如模块与模块的交互，后面的章节会讨论这个问题。这里，我们将第三个原则确定为分模块。

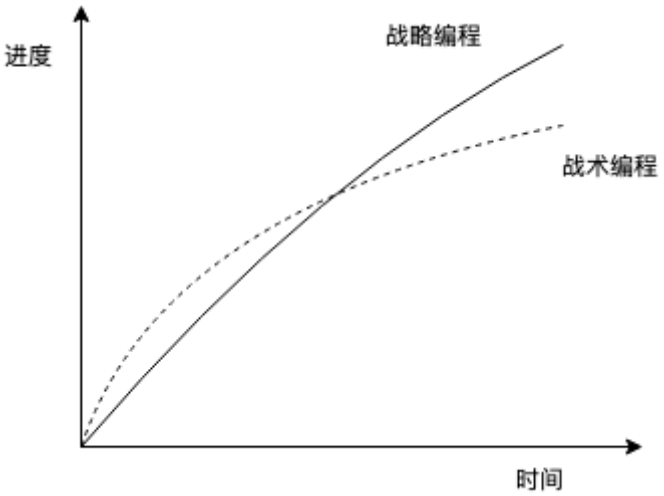
最后，代码能够描述程序的工作流程和结果，却很难描述开发人员的思路，而注释和文档可以。此外，通过注释和文档，开发人员在不阅读实现代码的情况下，就可以理解程序的功能，注释间接促成了代码抽象。好的注释能够帮助解决软件复杂性问题，尤其是认知负担和不可知问题（Unknown Unknowns）。

四、解决复杂性之日拱一卒

4.1 拒绝战术编程

战术编程致力于完成任务，新增加特性或者修改 Bug 时，能解决问题就好。这种工作方式，会逐渐增加系统的复杂性。如果系统复杂到难以维护时，再去重构会花费大量的时间，很可能会影响新功能的迭代。

战略编程，是指重视设计并愿意投入时间，短时间内可能会降低工作效率，但是长期看，会增加系统的可维护性和迭代效率。



设计系统时，很难在开始阶段就面面俱到。好的设计应该体现在一个个小的模块上，修改 bug 时，也应该抱着设计新系统的心态，完工后让人感觉不到“修补”的痕迹。经过累积，最终形成一个完善的系统。从长期看，对于中大型的系统，将日常开

发时间的 10–15% 用于设计是值得的。有一种观点认为，创业公司需要追求业务迭代速度和节省成本，可以容忍糟糕的设计，这是用错误的方法去追求正确的目标。降低开发成本最有效的方式是雇佣优秀的工程师，而不是在设计上做妥协。

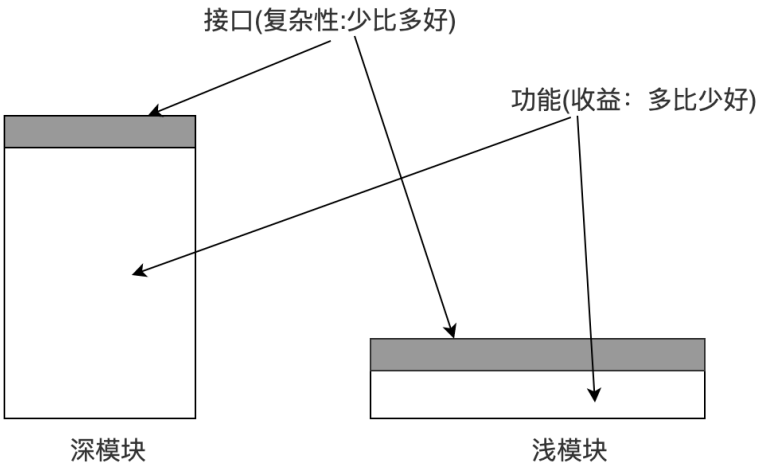
4.2 设计两次

为一个类、模块或者系统的设计提供两套或更多方案，有利于我们找到最佳设计。以我们日常的技术方案设计为例，技术方案本质上需要回答两个问题，其一，为什么该方案可行？ 其二，在已有资源限制下，为什么该方案是最优的？为了回答第一个问题，我们需要在技术方案里补充架构图、接口设计和时间人力估算。而要回答第二个问题，需要我们在关键点或争议处提供二到三种方案，并给出建议方案，这样才有说服力。通常情况下，我们会花费很多的时间准备第一个问题，而忽略第二个问题。其实，回答好第二个问题很重要，大型软件的设计已经复杂到没人能够一次就想到最佳方案，一个仅仅“可行”的方案，可能会给系统增加额外的复杂性。对聪明人来说，接受这点更困难，因为他们习惯于“一次搞定问题”。但是聪明人迟早也会碰到自己的瓶颈，在低水平问题上徘徊，不如花费更多时间思考，去解决真正有挑战性的问题。

五、解决复杂性之分层

5.1 层次和抽象

软件系统由不同的层次组成，层次之间通过接口来交互。在严格分层的系统里，内部的层只对相邻的层次可见，这样就可以将一个复杂问题分解成增量步骤序列。由于每一层最多影响两层，也给维护带来了很大的便利。分层系统最有名的实例是 TCP/IP 网络模型。



在分层系统里，每一层应该具有不同的抽象。TCP/IP 模型中，应用层的抽象是用户接口和交互；传输层的抽象是端口和应用之间的数据传输；网络层的抽象是基于 IP 的寻址和数据传输；链路层的抽象是适配和虚拟硬件设备。如果不同的层具有相同的抽象，可能存在层次边界不清晰的问题。

5.2 复杂性下沉

不应该让用户直面系统的复杂性，即便有额外的工作量，开发人员也应当尽量让用户使用更简单。如果一定要在某个层次处理复杂性，这个层次越低越好。举个例子，Thrift 接口调用时，数据传输失败需要引入自动重试机制，重试的策略显然在 Thrift 内部封装更合适，开放给用户（下游开发人员）会增加额外的使用负担。与之类似的是系统里随处可见的配置参数（通常写在 XML 文件里），在编程中应当尽量避免这种情况，用户（下游开发人员）一般很难决定哪个参数是最优的，如果一定要开放参数配置，最好给定一个默认值。

复杂性下沉，并不是说把所有功能下移到一个层次，过犹不及。如果复杂性跟下层的功能相关，或者下移后，能大大下降其他层次或整体的复杂性，则下移。

5.3 异常处理

异常和错误处理是造成软件复杂的罪魁祸首之一。有些开发人员错误的认为处理和上报的错误越多越好，这会导致过度防御性的编程。如果开发人员捕获了异常并不知道如何处理，直接往上层扔，这就违背了封装原则。

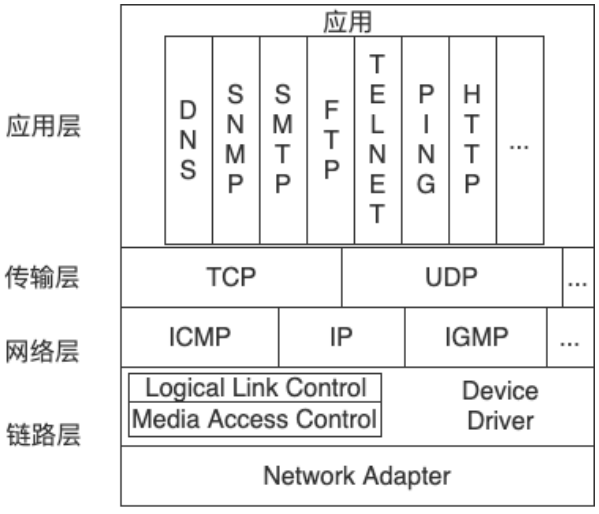
降低复杂度的一个原则就是尽可能减少需要处理异常的可能性。而最佳实践就是确保错误终结，例如删除一个并不存在的文件，与其上报文件不存在的异常，不如什么都不做。确保文件不存在就好了，上层逻辑不但不会被影响，还会因为不需要处理额外的异常而变得简单。

六、解决复杂性之分模块

分模块是解决复杂性的重要方法。理想情况下，模块之间应该是相互隔离的，开发人员面对具体的任务，只需要接触和了解整个系统的一小部分，而无需了解或改动其他模块。

6.1 深模块和浅模块

深模块 (Deep Module) 指的是拥有强大功能和简单接口的模块。深模块是抽象的最佳实践，通过排除模块内部不重要的信息，让用户更容易理解和使用。



TCP/IP网络模型

Unix 操作系统文件 I/O 是典型的深模块，以 Open 函数为例，接口接受文件名为参数，返回文件描述符。但是这个接口的背后，是几百行的实现代码，用来处理文件存储、权限控制、并发控制、存储介质等等，这些对用户是不可见的。

```
int open(const char* path, int flags, mode_t permissions);
```

与深模块相对的是浅模块 (Shallow Module)，功能简单，接口复杂。通常情况下，浅模块无助于解决复杂性。因为他们提供的收益（功能）被学习和使用成本抵消了。以 Java I/O 为例，从 I/O 中读取对象时，需要同时创建三个对象 FileInput-Stream、BufferedInputStream、ObjectInputStream，其中前两个创建后不会被直接使用，这就给开发人员造成了额外的负担。默认情况下，开发人员无需感知到 BufferedInputStream，缓冲功能有助于改善文件 I/O 性能，是个很有用的特性，可以合并到文件 I/O 对象里。假如我们想放弃缓冲功能，文件 I/O 也可以设计成提供对应的定制选项。

```
FileInputStream fileStream = new FileInputStream(fileName);
BufferedInputStream bufferedStream = new BufferedInputStream(fileStream);
ObjectInputStream objectStream = new ObjectInputStream(bufferedStream);
```

关于浅模块有一些争议，大多数情况是因为浅模块是不得不接受的既定事实，而不见得是因为合理性。当然也有例外，比如领域驱动设计里的防腐层，系统在与外部系统对接时，会单独建立一个服务或模块去适配，用来保证原有系统技术栈的统一和稳定性。

6.2 通用和专用

设计新模块时，应该设计成通用模块还是专用模块？一种观点认为通用模块满足多种场景，在未来遇到预期外的需求时，可以节省时间。另外一种观点则认为，未来的需求很难预测，没必要引入用不到的特性，专用模块可以快速满足当前的需求，等有后续需求时再重构成通用的模块也不迟。

以上两种思路都有道理，实际操作的时候可以采用两种方式各自的优点，即在功能实现上满足当前的需求，便于快速实现；接口设计通用化，为未来留下余量。

举个例子。

```
void backspace(Cursor cursor);
void delete(Cursor cursor);
void deleteSelection(Selection selection);

// 以上三个函数可以合并为一个更通用的函数
void delete(Position start, Position end);
```

设计通用性接口需要权衡，既要满足当前的需求，同时在通用性方面不要过度设计。一些可供参考的标准：

- 满足当前需求最简单的接口是什么？在不减少功能的前提下，减少方法的数量，意味着接口的通用性提升了。
- 接口使用的场景有多少？如果接口只有一个特定的场景，可以将多个这样的接口合并成通用接口。
- 满足当前需求情况下，接口的易用性？如果接口很难使用，意味着我们可能过度设计了，需要拆分。

6.3 信息隐藏

信息隐藏是指，程序的设计思路以及内部逻辑应当包含在模块内部，对其他模块不可见。如果一个模块隐藏了很多信息，说明这个模块在提供很多功能的同时又简化了接口，符合前面提到的深模块理念。软件设计领域有个技巧，定义一个”大”类有助于实现信息隐藏。这里的“大”类指的是，如果要实现某功能，将该功能相关的信息都封装进一个类里面。

信息隐藏在降低复杂性方面主要有两个作用：一是简化模块接口，将模块功能以更简单、更抽象的方式表现出来，降低开发人员的认知负担；二是减少模块间的依赖，使得系统迭代更轻量。举个例子，如何从 B+ 树中存取信息是一些数据库索引的核心功能，但是数据库开发人员将这些信息隐藏了起来，同时提供简单的对外交互接口，也就是 SQL 脚本，使得产品和运营同学也能很快地上手。并且，因为有足够的抽象，数据库可以在保持外部兼容的情况下，将索引切换到散列或其他数据结构。

与信息隐藏相对的是信息暴露，表现为：设计决策体现在多个模块，造成不同模块间的依赖。举个例子，两个类能处理同类型的文件。这种情况下，可以合并这两个类，或者提炼出一个新类（参考《重构》[3] 一书）。工程师应当尽量减少外部模块需要的信息量。

6.4 拆分和合并

两个功能，应该放在一起还是分开？“不管黑猫白猫”，能降低复杂性就好。这里有一些可以借鉴的设计思路：

- 共享信息的模块应当合并，比如两个模块都依赖某个配置项。
- 可以简化接口时合并，这样可以避免客户同时调用多个模块来完成某个功能。
- 可以消除重复时合并，比如抽离重复的代码到一个单独的方法中。
- 通用代码和专用代码分离，如果模块的部分功能可以通用，建议和专用部分分离。举个例子，在实际的系统设计中，我们会将专用模块放在上层，通用模块放在下层以供复用。

七、解决复杂性之注释

注释可以记录开发人员的设计思路和程序功能，降低开发人员的认知负担和解决不可知 (Unkown Unkowns) 问题，让代码更容易维护。通常情况下，在程序的整个生命周期里，编码只占了少部分，大量时间花在了后续的维护上。有经验的工程师懂得这个道理，通常也会产出更高质量的注释和文档。

注释也可以作为系统设计的工具，如果只需要简单的注释就可以描述模块的设计思路和功能，说明这个模块的设计是良好的。另一方面，如果模块很难注释，说明模块没有好的抽象。

7.1 注释的误区

关于注释，很多开发者存在一些认识上的误区，也是造成大家不愿意写注释的原因。比如“好代码是自注释的”、”没有时间“、“现有的注释都没有用，为什么还要浪费时间”等等。这些观点是站不住脚的。“好代码是自注释的”只在某些场景下是

合理的，比如为变量和方法选择合适的名称，可以不用单独注释。但是更多的情况，代码很难体现开发人员的设计思路。此外，如果用户只能通过读代码来理解模块的使用，说明代码里没有抽象。好的注释可以极大地提升系统的可维护性，获取长期的效率，不存在“没有时间”一说。注释也是一种可以习得的技能，一旦习得，就可以在后续的工作中应用，这就解决了“注释没有用”的问题。

7.2 使用注释提升系统可维护性

注释应当能提供代码之外额外的信息，重视 What 和 Why，而不是代码是如何实现的 (How)，最好不要简单地使用代码中出现过的单词。

根据抽象程度，注释可以分为低层注释和高层注释，低层次的注释用来增加精确度，补充完善程序的信息，比如变量的单位、控制条件的边界、值是否允许为空、是否需要释放资源等。高层次注释抛弃细节，只从整体上帮助读者理解代码的功能和结构。这种类型的注释更好维护，如果代码修改不影响整体的功能，注释就无需更新。在实际工作中，需要兼顾细节和抽象。低层注释拆散与对应的实现代码放在一起，高层注释一般用于描述接口。

注释先行，注释应该作为设计过程的一部分，写注释最好的时机是在开发的开始环节，这不仅会产生更好的文档，也会帮助产生好的设计，同时减少写文档带来的痛苦。开发人员推迟写注释的理由通常是：代码还在修改中，提前写注释到时候还得再改一遍。这样的话就会衍生两个问题：

- 首先，推迟注释通常意味着根本就没有注释。一旦决定推迟，很容易引发连锁反应，等到代码稳定后，也不会有注释这回事。这时候再想添加注释，就得专门抽出时间，客观条件可能不会允许这么做。
- 其次，就算我们足够自律抽出专门时间去写注释，注释的质量也不会很好。我们潜意识中觉得代码已经写完了，急于开展下一个项目，只是象征性地添加一些注释，无法准确复现当时的设计思路。

避免重复的注释。如果有重复注释，开发人员很难找到所有的注释去更新。解决方法是，可以找到醒目的地方存放注释文档，然后在代码处注明去查阅对应文档的地

址。如果程序已经在外部文档中注释过了，不要在程序内部再注释了，添加注释的引用就可以了。

注释属于代码，而不是提交记录。一种错误的做法是将功能注释放在提交记录里，而不是放在对应代码文件里。因为开发人员通常不会去代码提交记录里去查看程序的功能描述，很不方便。

7.3 使用注释改善系统设计

良好的设计基础是提供好的抽象，在开始编码前编写注释，可以帮助我们提炼模块的核心要素：模块或对象中最重要的功能和属性。这个过程促进我们去思考，而不是简单地堆砌代码。另一方面，注释也能够帮助我们检查自己的模块设计是否合理，正如前文中提到，深模块提供简单的接口和强大的功能，如果接口注释冗长复杂，通常意味着接口也很复杂；注释简单，意味着接口也很简单。在设计的早期注意和解决这些问题，会为我们带来长期的收益。

八、后记

John Ousterhout 累计写过 25 万行代码，是 3 个操作系统的重要贡献者，这些原则可以视为作者编程经验的总结。有经验的工程师看到这些观点会有共鸣，一些著作如《代码大全》、《领域驱动设计》也会有类似的观点。本文中提到的原则和方法具有一定实操和指导价值，对于很难有定论的问题，也可以在实践中去探索。

关于原则和方法论，既不必刻意拔高，也不要嗤之以鼻。指导实践的不是更多的实践，而是实践后的总结和思考。应用原则和方法论实质是借鉴已有的经验，可以减少我们自行摸索的时间。探索新的方法可以帮助我们适应新的场景，但是新方法本身需要经过时间检验。

九、参考文档

- John Ousterhout. A Philosophy of Software Design. Yaknyam Press, 2018.
- 梅拉尼 · 米歇尔 . 复杂 . 湖南科学技术出版社 , 2016.
- Martin Fowler. Refactoring: Improving the Design of Existing Code (2nd Edition) . Addison-Wesley Signature Series, 2018.

作者简介

政华，顺谱，陶鑫，美团打车调度系统工程团队工程师。

招聘信息

美团打车调度系统工程团队诚招高级工程师 / 技术专家，我们的目标，是与算法、数据团队密切协作，建设高性能、高可用、可配置的打车调度引擎 , 为用户提供更好的出行体验。欢迎有兴趣的同学发送简历到 tech@meituan.com (邮件标题注明：打车调度系统工程团队)。

算法

用能力支撑了每天数十亿次的交易量；

用实力攀登算法领域最高学术殿堂；

用技术绘制美团点评最美的 AI 全景图。

美团算法团队正在构建的 AI 相关技术囊括了视觉、语音、自然语言处理、机器学习、知识图谱等。以美团 / 大众点评 App 搜索、推荐为核心，面向外卖配送的策略、调度算法、定价系统，延伸到无人配送的自动驾驶、智能耳机里的语音识别、人脸识别，再到连接用户端的客服系统，连接商家端的金融体系、供应链系统也汇聚了美团点评正在构建的庞大知识图谱……

探索无止境，实践出真知。美团点评算法团队孜孜不倦的学习与探索，并在实际业务场景中引入新技术，总结新领悟。

美团 BERT 的探索和实践

杨扬　佳昊　金刚

2018 年，自然语言处理（Natural Language Processing，NLP）领域最激动人心的进展莫过于**预训练语言模型**，包括基于 RNN 的 ELMo[1] 和 ULMFiT[2]，基于 Transformer[3] 的 OpenAI GPT[4] 及 Google BERT[5] 等。下图 1 回顾了近年来预训练语言模型的发展史以及最新的进展。预训练语言模型的成功，证明了我们可以从海量的无标注文本中学到潜在的语义信息，而无需为每一项下游 NLP 任务单独标注大量训练数据。此外，预训练语言模型的成功也开创了 NLP 研究的新范式 [6]，即首先使用大量无监督语料进行语言模型预训练（Pre-training），再使用少量标注语料进行微调（Fine-tuning）来完成具体 NLP 任务（分类、序列标注、句间关系判断和机器阅读理解等）。

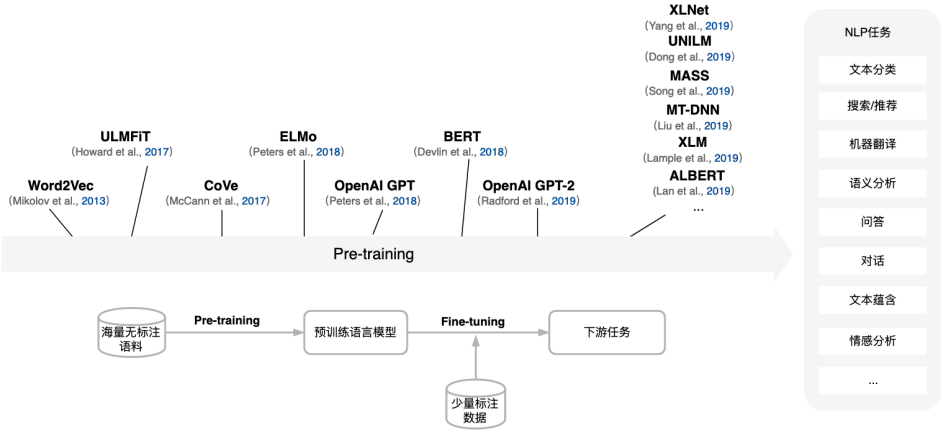


图 1　NLP Pre-training and Fine-tuning 新范式及相关扩展工作

所谓的“预训练”，其实并不是什么新概念，这种“Pre-training and Fine-tuning”的方法在图像领域早有应用。2009 年，邓嘉、李飞飞等人在 CVPR 2009 发布了 ImageNet 数据集 [7]，其中 120 万张图像分为 1000 个类别。基于 ImageNet，以图像分类为目标使用深度卷积神经网络（如常见的 ResNet、VCG、Inception 等）

进行预训练，得到的模型称为预训练模型。针对目标检测或者语义分割等任务，基于这些预训练模型，通过一组新的全连接层与预训练模型进行拼接，利用少量标注数据进行微调，将预训练模型学习到的图像分类能力迁移到新的目标任务。预训练的方式在图像领域取得了广泛的成功，比如有学者将 ImageNet 上学习得到的特征表示用于 PSACAL VOC 上的物体检测，将检测率提高了 20%[8]。

他山之石，可以攻玉。图像领域预训练的成功也启发了 NLP 领域研究，深度学习时代广泛使用的词向量（即词嵌入，Word Embedding）即属于 NLP 预训练工作。使用深度神经网络进行 NLP 模型训练时，首先需要将待处理文本转为词向量作为神经网络输入，词向量的效果会影响到最后模型效果。词向量的效果主要取决于训练语料的大小，很多 NLP 任务中有限的标注语料不足以训练出足够好的词向量，通常使用跟当前任务无关的大规模未标注语料进行词向量预训练，因此预训练的另一个好处是能增强模型的泛化能力。目前，大部分 NLP 深度学习任务中都会使用预训练好的词向量（如 Word2Vec[9] 和 GloVe[10] 等）进行网络初始化（而非随机初始化），从而加快网络的收敛速度。

预训练词向量通常只编码词汇间的关系，对上下文信息考虑不足，且无法处理一词多义问题。如“bank”一词，根据上下文语境不同，可能表示“银行”，也可能表示“岸边”，却对应相同的词向量，这样显然是不合理的。为了更好的考虑单词的上下文信息，Context2Vec[11] 使用两个双向长短时记忆网络（Long Short Term Memory，LSTM）[12] 来分别编码每个单词左到右（Left-to-Right）和右到左（Right-to-Left）的上下文信息。类似地，ELMo 也是基于大量文本训练深层双向 LSTM 网络结构的语言模型。ELMo 在词向量的学习中考虑深层网络不同层的信息，并加入到单词的最终 Embedding 表示中，在多个 NLP 任务中取得了提升。ELMo 这种使用预训练语言模型的词向量作为特征输入到下游目标任务中，被称为 Feature-based 方法。

另一种方法是微调（Fine-tuning）。GPT、BERT 和后续的预训练工作都属于这一范畴，直接在深层 Transformer 网络上进行语言模型训练，收敛后针对下游目标任务进行微调，不需要再为目标任务设计 Task-specific 网络从头训练。关于

NLP 领域的预训练发展史，张俊林博士写过一篇很详实的介绍 [13]，本文不再赘述。

Google AI 团队提出的预训练语言模型 BERT（Bidirectional Encoder Representations from Transformers），在 11 项自然语言理解任务上刷新了最好指标，可以说是近年来 NLP 领域取得的最重大的进展之一。BERT 论文也斩获 NLP 领域顶会 NAACL 2019 的最佳论文奖，BERT 的成功也启发了大量的后续工作，不断刷新了 NLP 领域各个任务的最佳水平。有 NLP 学者宣称，属于 NLP 的 ImageNet 时代已经来临 [14]。

美团点评作为中国领先的生活服务电子商务平台，涵盖搜索、推荐、广告、配送等多种业务场景，几乎涉及到各种类型的自然语言处理任务。以大众点评为例，迄今为止积累了近 40 亿条文本 UGC，如何高效而准确地完成对海量 UGC 的自然语言理解和处理是美团点评技术团队面临的挑战之一。美团点评 NLP 团队一直紧跟业界前沿技术，开展了基于美团点评业务数据的预训练研究工作，训练了更适配美团点评业务场景的 MT-BERT 模型，通过微调将 MT-BERT 落地到多个业务场景中，并取得了不错的业务效果。

BERT 是基于 Transformer 的深度双向语言表征模型，基本结构如图 2 所示，本质上是利用 Transformer 结构构造了一个多层双向的 Encoder 网络。Transformer 是 Google 在 2017 年提出的基于自注意力机制（Self-attention）的深层模型，在包括机器翻译在内的多项 NLP 任务上效果显著，超过 RNN 且训练速度更快。不到一年时间内，Transformer 已经取代 RNN 成为神经网络机器翻译的 State-Of-The-Art（SOTA）模型，包括谷歌、微软、百度、阿里、腾讯等公司的线上机器翻译模型都已替换为 Transformer 模型。关于 Transformer 的详细介绍可以参考 Google 论文《Attention is all you need》[3]。

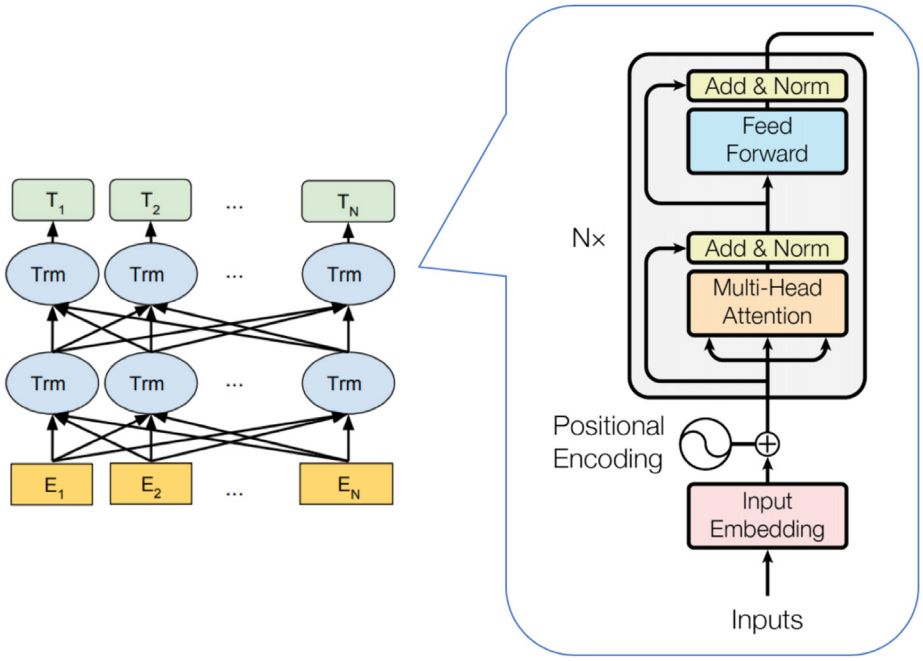


图 2　BERT 及 Transformer 网络结构示意图

模型结构

如表 1 所示，根据参数设置的不同，Google 论文中提出了 Base 和 Large 两种 BERT 模型。

表1　BERT Base和Large模型参数对比

模型	Layers	Hidden Size	Attention Head	参数数量
Base	12	768	12	110M
Large	24	1024	16	340M

输入表示

针对不同的任务，BERT 模型的输入可以是单句或者句对。对于每一个输入的 Token，它的表征由其对应的词表征（Token Embedding）、段表征（Segment Embedding）和位置表征（Position Embedding）相加产生，如图 3 所示：

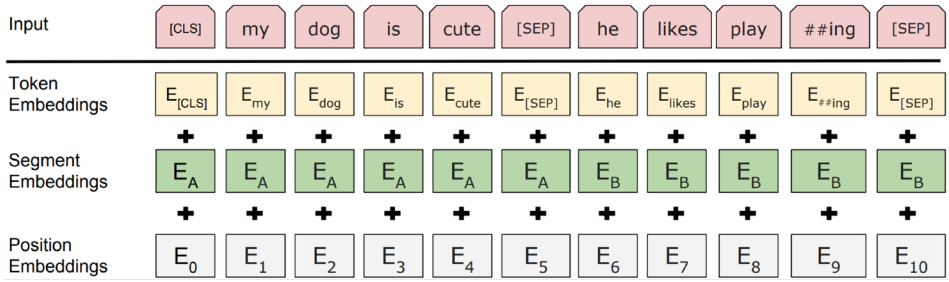


图 3　BERT 模型的输入表示

- 对于英文模型，使用了 Wordpiece 模型来产生 Subword 从而减小词表规模；对于中文模型，直接训练基于字的模型。
- 模型输入需要附加一个起始 Token，记为 [CLS]，对应最终的 Hidden State（即 Transformer 的输出）可以用来表征整个句子，用于下游的分类任务。
- 模型能够处理句间关系。为区别两个句子，用一个特殊标记符 [SEP] 进行分隔，另外针对不同的句子，将学习到的 Segment Embeddings 加到每个 Token 的 Embedding 上。
- 对于单句输入，只有一种 Segment Embedding；对于句对输入，会有两种 Segment Embedding。

预训练目标

BERT 预训练过程包含两个不同的预训练任务，分别是 Masked Language Model 和 Next Sentence Prediction 任务。

Masked Language Model（MLM）

通过随机掩盖一些词（替换为统一标记符 [MASK]），然后预测这些被遮盖的词来训练双向语言模型，并且使每个词的表征参考上下文信息。

这样做会产生两个缺点：（1）会造成预训练和微调时的不一致，因为在微调时 [MASK] 总是不可见的；（2）由于每个 Batch 中只有 15% 的词会被预测，因此模型的收敛速度比起单向的语言模型会慢，训练花费的时间会更长。对于第一个缺点的解

决办法是，把 80% 需要被替换成 [MASK] 的词进行替换，10% 的随机替换为其他词，10% 保留原词。由于 Transformer Encoder 并不知道哪个词需要被预测，哪个词是被随机替换的，这样就强迫每个词的表达需要参照上下文信息。对于第二个缺点目前没有有效的解决办法，但是从提升收益的角度来看，付出的代价是值得的。

Next Sentence Prediction（NSP）

为了训练一个理解句子间关系的模型，引入一个下一句预测任务。这一任务的训练语料可以从语料库中抽取句子对包括两个句子 A 和 B 来进行生成，其中 50% 的概率 B 是 A 的下一个句子，50% 的概率 B 是语料中的一个随机句子。NSP 任务预测 B 是否是 A 的下一句。NSP 的目的是获取句子间的信息，这点是语言模型无法直接捕捉的。

Google 的论文结果表明，这个简单的任务对问答和自然语言推理任务十分有益，但是后续一些新的研究 [15] 发现，去掉 NSP 任务之后模型效果没有下降甚至还有提升。我们在预训练过程中也发现 NSP 任务的准确率经过 1–2 个 Epoch 训练后就能达到 98%–99%，去掉 NSP 任务之后对模型效果并不会有太大的影响。

数据 & 算力

Google 发布的英文 BERT 模型使用了 BooksCorpus（800M 词汇量）和英文 Wikipedia（2500M 词汇量）进行预训练，所需的计算量非常庞大。BERT 论文中指出，Google AI 团队使用了算力强大的 Cloud TPU 进行 BERT 的训练，BERT Base 和 Large 模型分别使用 4 台 Cloud TPU（16 张 TPU）和 16 台 Cloud TPU（64 张 TPU）训练了 4 天（100 万步迭代，40 个 Epoch）。但是，当前国内互联网公司主要使用 Nvidia 的 GPU 进行深度学习模型训练，因此 BERT 的预训练对于 GPU 资源提出了很高的要求。

美团 BERT（MT-BERT）的探索分为四个阶段：（1）开启混合精度实现训练加速；（2）在通用中文语料基础上加入大量美团点评业务语料进行模型预训练，完成领域迁移；（3）预训练过程中尝试融入知识图谱中的实体信息；（4）通过在业务数据上进行微调，支持不同类型的业务需求。MT-BERT 整体技术框架如图 4 所示：

图 4　MT-BERT 整体技术框架

基于美团点评 AFO 平台的分布式训练

正如前文所述，BERT 预训练对于算力有着极大要求，我们使用的是美团内部开发的 AFO[16]（AI Framework On Yarn）框架进行 MT-BERT 预训练。AFO 框架基于 YARN 实现数千张 GPU 卡的灵活调度，同时提供基于 Horovod 的分布式训练方案，以及作业弹性伸缩与容错等能力。Horovod 是 Uber 开源的深度学习工具 [17]，它的发展吸取了 Facebook《一小时训练 ImageNet》论文 [18] 与百度 Ring Allreduce[19] 的优点，可为用户实现分布式训练提供帮助。根据 Uber 官方分别使用标准分布式 TensorFlow 和 Horovod 两种方案，分布式训练 Inception V3 和 ResNet-101 TensorFlow 模型的实验验证显示，随着 GPU 的数量增大，Horovod 性能损失远小于 TensorFlow，且训练速度可达到标准分布式 TensorFlow 的近两倍。相比于 Tensorflow 分布式框架，Horovod 在数百张卡的规模上依然可以保证稳定的加速比，具备非常好的扩展性。

Horovod 框架的并行计算主要用到了两种分布式计算技术：控制层的 Open

MPI 和数据层的 Nvidia NCCL。控制层面的主要作用是同步各个 Rank（节点），因为每个节点的运算速度不一样，运算完每一个 Step 的时间也不一样。如果没有一个同步机制，就不可能对所有的节点进行梯度平均。Horovod 在控制层面上设计了一个主从模式，Rank 0 为 Master 节点，Rank1-n 为 Worker 节点，每个 Worker 节点上都有一个消息队列，而在 Master 节点上除了一个消息队列，还有一个消息 Map。每当计算框架发来通信请求时，比如要执行 Allreduce，Horovod 并不直接执行 MPI，而是封装了这个消息并推入自己的消息队列，交给后台线程去处理。后台线程采用定时轮询的方式访问自己的消息队列，如果非空，Woker 会将自己收到的所有 Tensor 通信请求都发给 Master。因为是同步 MPI，所以每个节点会阻塞等待 MPI 完成。Master 收到 Worker 的消息后，会记录到自己的消息 Map 中。如果一个 Tensor 的通信请求出现了 n 次，也就意味着，所有的节点都已经发出了对该 Tensor 的通信请求，那么这个 Tensor 就需要且能够进行通信。Master 节点会挑选出所有符合要求的 Tensor 进行 MPI 通信。不符合要求的 Tensor 继续留在消息 Map 中，等待条件满足。决定了 Tensor 以后，Master 又会将可以进行通信的 Tensor 名字和顺序发还给各个节点，通知各个节点可以进行 Allreduce 运算。

混合精度加速

当前深度学习模型训练过程基本采用单精度（Float 32）和双精度（Double）数据类型，受限于显存大小，当网络规模很大时 Batch Size 就会很小。Batch Size 过小一方面容易导致网络学习过程不稳定而影响模型最终效果，另一方面也降低了数据吞吐效率，影响训练速度。为了加速训练及减少显存开销，Baidu Research 和 Nvidia 在 ICLR 2018 论文中 [20] 合作提出了一种 Float32（FP32）和 Float16（FP16）混合精度训练的方法，并且在图像分类和检测、语音识别和语言模型任务上进行了有效验证。Nvidia 的 Pascal 和 Volta 系列显卡除了支持标准的单精度计算外，也支持了低精度的计算，比如最新的 Tesla V100 硬件支持了 FP16 的计算加速，P4 和 P40 支持 INT8 的计算加速，而且低精度计算的峰值要远高于单精浮点的计算峰值。

为了进一步加快 MT-BERT 预训练和推理速度，我们实验了混合精度训练方式。混合精度训练指的是 FP32 和 FP16 混合的训练方式，使用混合精度训练可以加速训练过程并且减少显存开销，同时兼顾 FP32 的稳定性和 FP16 的速度。在模型计算过程中使用 FP16 加速计算过程，模型训练过程中权重会存储成 FP32 格式（FP32 Master-weights），参数更新时采用 FP32 类型。利用 FP32 Master-weights 在 FP32 数据类型下进行参数更新可有效避免溢出。 此外，一些网络的梯度大部分在 FP16 的表示范围之外，需要对梯度进行放大使其可以在 FP16 的表示范围内，因此进一步采用 Loss Scaling 策略通过对 Loss 进行放缩，使得在反向传播过程中梯度在 FP16 的表示范围内。

为了提高预训练效率，我们在 MT-BERT 预训练中采用了混合精度训练方式。

加速效果

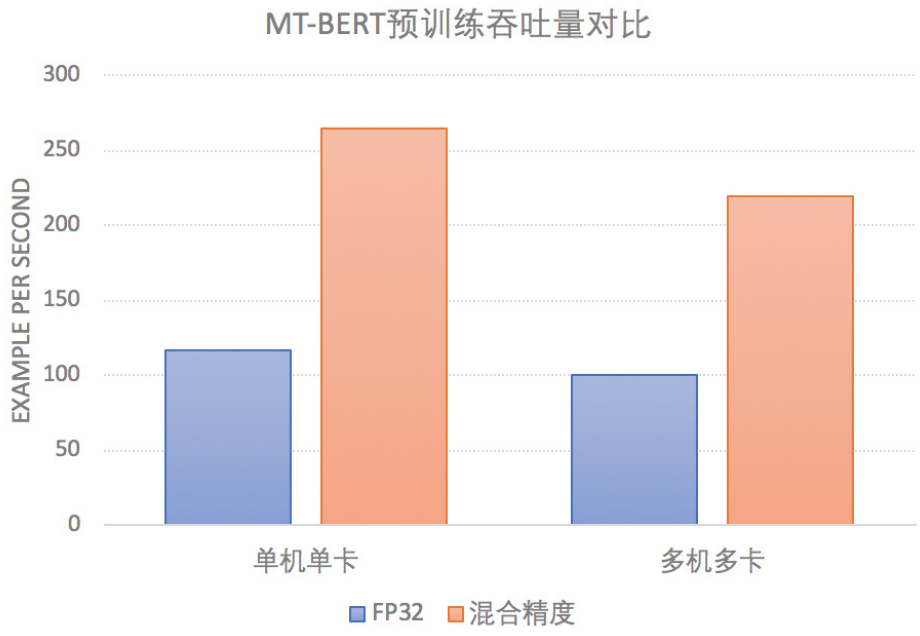


图 5　MT-BERT 开启混合精度在 Nvidia V100 上训练吞吐量的对比
（Tensorflow 1.12；Cuda 10.0；Horovod 0.15.2）

如图 5 所示，开启混合精度的训练方式在单机单卡和多机多卡环境下显著提升了训练速度。为了验证混合精度模型会不会影响最终效果，我们分别在美团点评业务和通用 Benchmark 数据集上进行了微调实验，结果见表 2 和表 3。

表2　开启混合精度训练的MT-BERT模型在美团点评业务Benchmark上效果对比

数据集	Metric	MT-BERT FP32	MT-BERT 混合精度	Google BERT
细粒度情感分析	Macro-F1	72.04%	72.25%	71.63%
Query 意图分类	F1	93.27%	93.13%	92.68%
Query 成分分析（NER）	F1	91.46%	91.05%	90.66%

表3　开启混合精度训练的MT-BERT模型在中文通用Benchmark上效果对比

数据集	Metric	MT-BERT FP32	MT-BERT 混合精度	Google BERT
MSRA-NER	F1	95.89%	95.75%	95.76%
LCQMC	Accuracy	86.74%	85.87%	86.06%
ChnSentiCorp	Accuracy	95.00%	94.92%	92.25%
NLPCC-DBQA	MRR	94.07%	93.24%	93.55%
XNLI	Accuracy	78.10%	76.57%	77.47%

通过表 2 和表 3 结果可以发现，开启混合精度训练的 MT-BERT 模型并没有影响效果，反而训练速度提升了 2 倍多。

领域自适应

Google 发布的中文 BERT 模型是基于中文维基百科数据训练得到，属于通用领域预训练语言模型。由于美团点评积累了大量业务语料，比如用户撰写的 UGC 评论和商家商品的文本描述数据，为了充分发挥领域数据的优势，我们考虑在 Google 中文 BERT 模型上加入领域数据继续训练进行领域自适应（Domain Adaptation），使得模型更加匹配我们的业务场景。实践证明，这种 Domain-aware Continual Training 方式，有效地改进了 BERT 模型在下游任务中的表现。由于 Google 未发布中文 BERT Large 模型，我们也从头预训练了中文 MT-BERT Large 模型。

我们选择了 5 个中文 Benchmark 任务以及 3 个美团点评业务 Benchmark 在内的 8 个数据集对模型效果进行验证。实验结果如表 4 所示，MT-BERT 在通用 Benchmark 和美团点评业务 Benchmark 上都取得了更好的效果。

表4 MT-BERT模型和Google BERT模型在8个Benchmark上的效果对比

Benchmark	Metric	Google BERT	MT-BERT
MSRA-NER	F1	95.76%	95.89%
LCQMC	Accuracy	86.06%	86.74%
ChnSentiCorp	Accuracy	92.25%	95.00%
NLPCC-DBQA	MRR	93.55%	94.07%
XNLI	Accuracy	77.47%	78.10%
细粒度情感分析	Macro-F1	71.63%	72.04%
Query 意图分类	F1	92.68%	93.27%
Query 成分分析（NER）	F1	90.66%	91.46%

知识融入

BERT 在自然语言理解任务上取得了巨大的成功，但也存在着一些不足。其一是常识（Common Sense）的缺失。人类日常活动需要大量的常识背景知识支持，BERT 学习到的是样本空间的特征、表征，可以看作是大型的文本匹配模型，而大量的背景常识是隐式且模糊的，很难在预训练数据中进行体现。其二是缺乏对语义的理解。模型并未理解数据中蕴含的语义知识，缺乏推理能力。在美团点评搜索场景中，需要首先对用户输入的 Query 进行意图识别，以确保召回结果的准确性。比如，对于“宫保鸡丁”和“宫保鸡丁酱料”两个 Query，二者的 BERT 语义表征非常接近，但是蕴含的搜索意图却截然不同。前者是菜品意图，即用户想去饭店消费，而后者则是商品意图，即用户想要从超市购买酱料。在这种场景下，BERT 模型很难像正常人一样做出正确的推理判断。

为了处理上述情况，我们尝试在 MT-BERT 预训练过程中融入知识图谱信息。知识图谱可以组织现实世界中的知识，描述客观概念、实体、关系。这种基于符号语义的计算模型，可以为 BERT 提供先验知识，使其具备一定的常识和推理能力。在我们团队之前的技术文章 [21] 中，介绍了 NLP 中心构建的大规模的餐饮娱乐知识图

谱——美团大脑。我们通过 Knowledge-aware Masking 方法将“美团大脑”的实体知识融入到 MT-BERT 预训练中。

BERT 在进行语义建模时，主要聚焦最原始的单字信息，却很少对实体进行建模。具体地，BERT 为了训练深层双向的语言表征，采用了 Masked LM（MLM）训练策略。该策略类似于传统的完形填空任务，即在输入端，随机地“遮蔽”掉部分单字，在输出端，让模型预测出这些被“遮蔽”的单字。模型在最初并不知道要预测哪些单字，因此它输出的每个单字的嵌入表示，都涵盖了上下文的语义信息，以便把被“掩盖”的单字准确的预测出来。

图 6 左侧展示了 BERT 模型的 MLM 任务。输入句子是“全聚德做的烤鸭久负盛名”。其中，“聚”，“的”，“久”3 个字在输入时被随机遮蔽，模型预训练过程中需要对这 3 个遮蔽位做出预测。

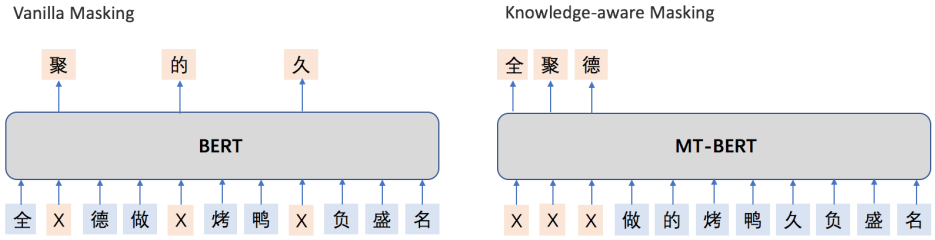


图 6　MT-BERT 默认 Masking 策略和 Whole Word Masking 策略对比

BERT 模型通过字的搭配（比如“全 X 德”），很容易推测出被“掩盖”字信息（“德”），但这种做法只学习到了实体内单字之间共现关系，并没有学习到实体的整体语义表示。因此，我们使用 Knowledge-aware Masking 的方法来预训练 MT-BERT。具体的做法是，输入仍然是字，但在随机”遮蔽”时，不再选择遮蔽单字，而是选择“遮蔽”实体对应的词。这需要我们在预训练之前，对语料做分词，并将分词结果和图谱实体对齐。图 6 右侧展示了 Knowledge-aware Masking 策略，“全聚德”被随机“遮蔽”。MT-BERT 需要根据“烤鸭”，“久负盛名”等信息，准确的预测出“全聚德”。通过这种方式，MT-BERT 可以学到“全聚德”这个实体的语义

表示，以及它跟上下文其他实体之间的关联，增强了模型语义表征能力。基于美团大脑中已有实体信息，我们在 MT-BERT 训练中使用了 Knowledge-aware Masking 策略，实验证明在细粒度情感分析任务上取得了显著提升。

表5 MT-BERT在细粒度情感分析数据集上效果

模型	Macro-F1
BERT (Vanilla masking)	72.04%
MT-BERT (Knowledge-aware Masking)	72.48%

模型轻量化

BERT 模型效果拔群，在多项自然语言理解任务上实现了最佳效果，但是由于其深层的网络结构和庞大的参数量，如果要部署上线，还面临很大挑战。以 Query 意图分类为例，我们基于 MT-BERT 模型微调了意图分类模型，协调工程团队进行了 1000QPS 压测实验，部署 30 张 GPU 线上卡参与运算，在线服务的 TP999 高达 50ms 之多，难以满足上线要求。

为了减少模型响应时间，满足上线要求，业内主要有三种模型轻量化方案。

- 低精度量化。在模型训练和推理中使用低精度（FP16 甚至 INT8、二值网络）表示取代原有精度（FP32）表示。
- 模型裁剪和剪枝。减少模型层数和参数规模。
- 模型蒸馏。通过知识蒸馏方法 [22] 基于原始 BERT 模型蒸馏出符合上线要求的小模型。

在美团点评搜索 Query 意图分类任务中，我们优先尝试了模型裁剪的方案。由于搜索 Query 长度较短（通常不超过 16 个汉字），整个 Sequence 包含的语义信息有限，裁剪掉几层 Transformer 结构对模型的语义表征能力不会有太大影响，同时又能大幅减少模型参数量和推理时间。经过实验验证，在微调过程中，我们将 MT-BERT 模型裁剪为 4 层 Transfomer 结构（MT-BERT-MINI，MBM），实验效果如图 7 所示。可以发现，Query 分类场景下，裁剪后的 MBM 没有产生较大影响。由

于减少了一些不必要的参数运算，在美食和酒店两个场景下，效果还有小幅的提升。

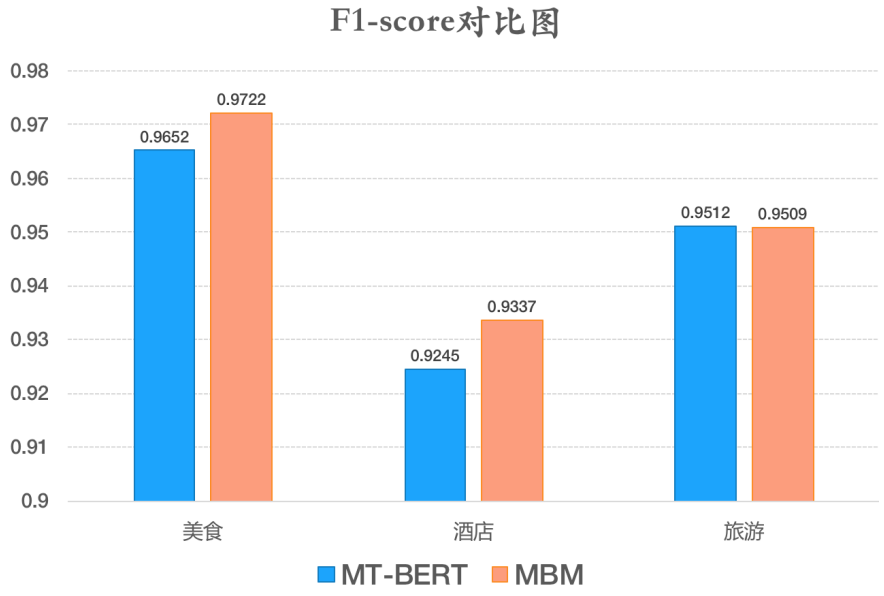


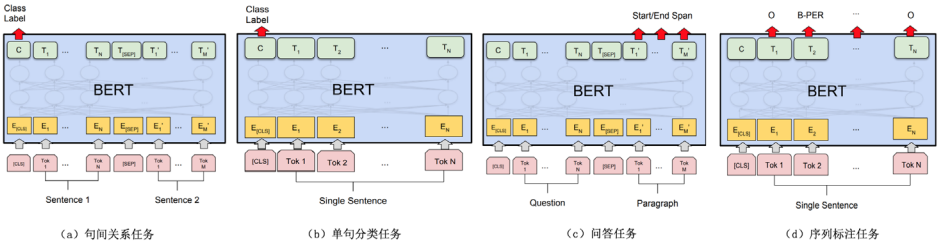
图 7　裁剪前后 MT-BERT 模型在 Query 意图分类数据集上 F1 对比

MBM 在同等压测条件下，压测服务的 TP999 达到了 12-14ms，满足搜索上线要求。除了模型裁剪，为了支持更多线上需求，我们还在进行模型蒸馏实验，蒸馏后的 6 层 MT-BERT 模型在大多数下游任务中都没有显著的效果损失。值得一提的是，BERT 模型轻量化是 BERT 相关研究的重要方向，最近 Google 公布了最新 ALBERT 模型（A Lite BERT）[23]，在减少模型参数量的同时在自然语言理解数据集 GLUE 上刷新了 SOTA。

图 8 展示了基于 BERT 模型微调可以支持的任务类型，包括句对分类、单句分类、问答（机器阅读理解）和序列标注任务。

1. 句对分类任务和单句分类任务是句子级别的任务。预训练中的 NSP 任务使得 BERT 中的“[CLS]”位置的输出包含了整个句子对（句子）的信息，我们利用其在有标注的数据上微调模型，给出预测结果。
2. 问答和序列标注任务都属于词级别的任务。预训练中的 MLM 任务使得每个

Token 位置的输出都包含了丰富的上下文语境以及 Token 本身的信息，我们对 BERT 的每个 Token 的输出都做一次分类，在有标注的数据上微调模型并给出预测。



（a）句间关系任务　（b）单句分类任务　（c）问答任务　（d）序列标注任务

图 8　BERT 微调支持的任务类型

基于 MT-BERT 的微调，我们支持了美团搜索和点评搜索的多个下游任务，包括单句分类任务、句间关系任务和序列标注任务等等。

单句分类

细粒度情感分析

美团点评作为生活服务平台，积累了大量真实用户评论。对用户评论的细粒度情感分析在深刻理解商家和用户、挖掘用户情感等方面有至关重要的价值，并且在互联网行业已有广泛应用，如个性化推荐、智能搜索、产品反馈、业务安全等领域。为了更全面更真实的描述商家各属性情况，细粒度情感分析需要判断评论文本在各个属性上的情感倾向（即正面、负面、中立）。为了优化美团点评业务场景下的细粒度情感分析效果，NLP 中心标注了包含 6 大类 20 个细粒度要素的高质量数据集，标注过程中采用严格的多人标注机制保证标注质量，并在 AI Challenger 2018 细粒度情感分析比赛中作为比赛数据集验证了效果，吸引了学术界和工业届大量队伍参赛。

针对细粒度情感分析任务，我们设计了基于 MT-BERT 的多任务分类模型，模型结构如图 9 所示。模型架构整体分为两部分：一部分是各情感维度的参数共享层（Share Layers），另一部分为各情感维度的参数独享层（Task-specific Layers）。其中参数共享层采用了 MT-BERT 预训练语言模型得到文本的上下文表征。MT-BERT 依赖其深层网络结构以及海量数据预训练，可以更好的表征上下文信息，尤其

擅长提取深层次的语义信息。 参数独享层采用多路并行的 Attention+Softmax 组合结构，对文本在各个属性上的情感倾向进行分类预测。通过 MT-BERT 优化后的细粒度情感分析模型在 Macro-F1 上取得了显著提升。

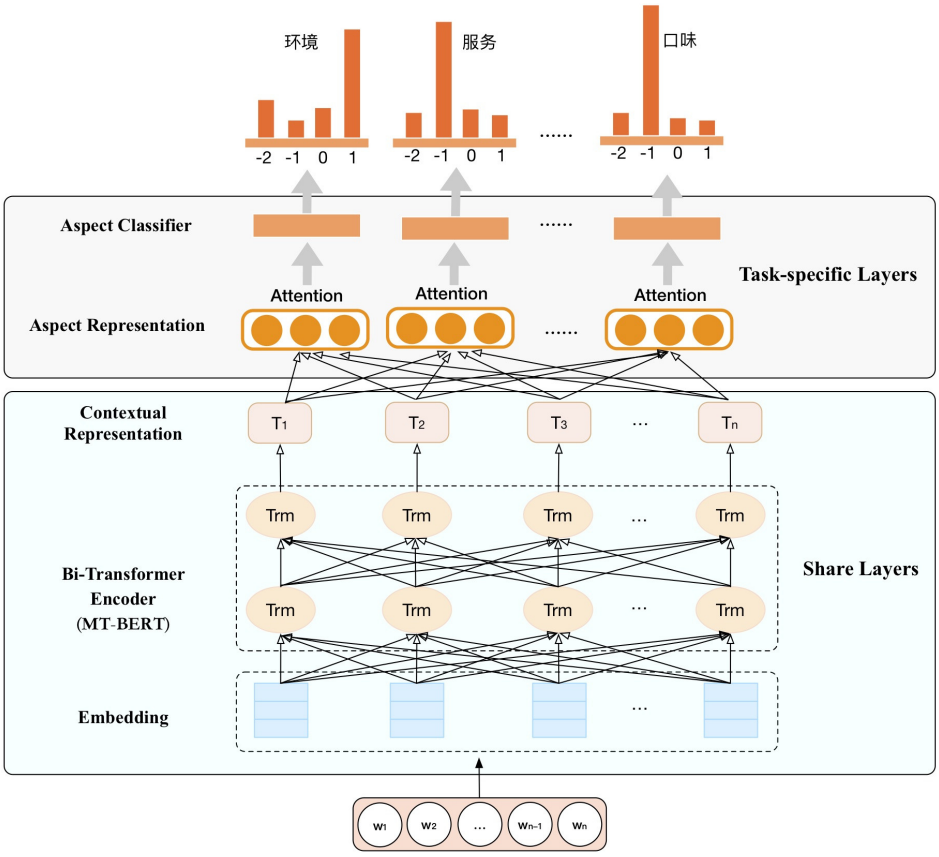


图 9　基于 MT-BERT 的多任务细粒度情感分析模型架构

细粒度情感分析的重要应用场景之一是大众点评的精选点评模块，如图 10 所示。精选点评模块作为点评 App 用户查看高质量评论的入口，其中精选点评标签承载着结构化内容聚合的作用，支撑着用户高效查找目标 UGC 内容的需求。细粒度情感分析能够从不同的维度去挖掘评论的情感倾向。基于细粒度情感分析的情感标签能够较好地帮助用户筛选查看，同时外露更多的 POI 信息，帮助用户高效的从评论中获取消费指南。

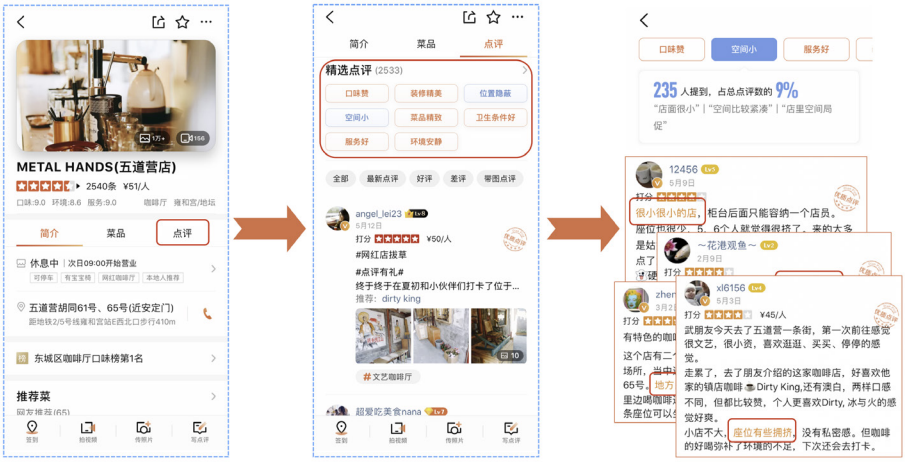
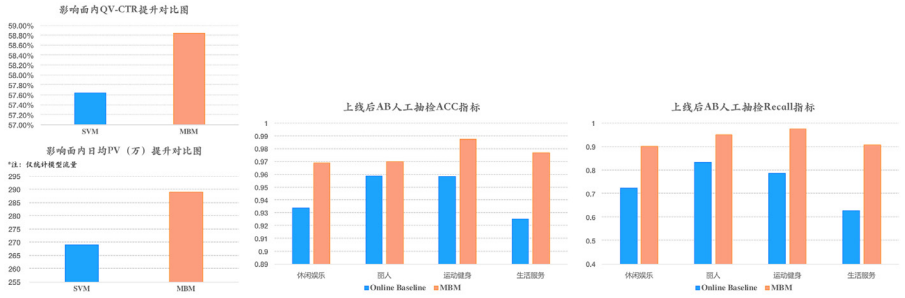


图 10　大众点评精选点评模块产品形态

Query 意图分类

在美团点评的搜索架构中，Deep Query Understanding（DQU）都是重要的前置模块之一。对于用户 Query，需要首先对用户搜索意图进行识别，如美食、酒店、演出等等。我们跟内部的团队合作，尝试了直接使用 MT-BERT 作为 Query 意图分类模型。为了保证模型在线 Inference 时间，我们使用裁剪后的 4 层 MT-BERT 模型（MT-BERT-MINI，MBM 模型）上线进行 Query 意图的在线意图识别，取得的业务效果如图 11 所示：



(a)线上收益指标（以美食频道为例）　　(b) 线下准召指标（以部分频道为例）

图 11　MBM 模型的业务效果

同时对于搜索日志中的高频 Query，我们将预测结果以词典方式上传到缓存，进一步减少模型在线预测的 QPS 压力。MBM 累计支持了美团点评搜索 17 个业务频道的 Query 意图识别模型，相比原有模型，均有显著的提升，每个频道的识别精确度都达到 95% 以上。MBM 模型上线后，提升了搜索针对 Query 文本的意图识别能力，为下游的搜索的召回、排序及展示、频道流量报表、用户认知报表、Bad Case 归因等系统提供了更好的支持。

推荐理由场景化分类

推荐理由是点评搜索智能中心数据挖掘团队基于大众点评 UGC 为每个 POI 生产的自然语言可解释性理由。对于搜索以及推荐列表展示出来的每一个商家，我们会用一句自然语言文本来突出商家的特色和卖点，从而让用户能够对展示结果有所感知，“知其然，更知其所以然”。近年来，可解释的搜索系统越来越受到关注，给用户展示商品或内容的同时透出解释性理由，正在成为业界通行做法，这样不仅能提升系统的透明度，还能提高用户对平台的信任和接受程度，进而提升用户体验效果。在美团点评的搜索推荐场景中，推荐理由有着广泛的应用场景，起到解释展示、亮点推荐、场景化承载和个性化体现的重要作用，目前已经有 46 个业务方接入了推荐理由服务。

对于不同的业务场景，对推荐理由会有不同的要求。在外卖搜索场景下，用户可能更为关注菜品和配送速度，不太关注餐馆的就餐环境和空间，这种情况下只保留符合外卖场景的推荐理由进行展示。同样地，在酒店搜索场景下，用户可能更为关注酒店特色相关的推荐理由（如交通是否方便，酒店是否近海近景区等）。

我们通过内部合作，为业务方提供符合不同场景需求的推荐理由服务。推荐理由场景化分类，即给定不同业务场景定义，为每个场景标注少量数据，我们可以基于 MT-BERT 进行单句分类微调，微调方式如图 8(b) 所示。

图 12　外卖和酒店场景下推荐理由

句间关系

句间关系任务是对两个短语或者句子之间的关系进行分类，常见句间关系任务如自然语言推理（Natural Language Inference, NLI）、语义相似度判断（Semantic Textual Similarity，STS）等。

Query 改写是在搜索引擎中对用户搜索 Query 进行同义改写，改善搜索召回结果的一种方法。在美团和点评搜索场景中，通常一个商户或者菜品会有不同的表达方式，例如“火锅”也称为“涮锅”。有时不同的词语表述相同的用户意图，例如“婚纱摄影”和“婚纱照”，“配眼镜”和“眼镜店”。Query 改写可以在不改变用户意图的情况下，尽可能多的召回满足用户意图的搜索结果，提升用户的搜索体验。为了

减少误改写，增加准确率，需要对改写后 Query 和原 Query 做语义一致性判断，只有语义一致的 Query 改写对才能上线生效。Query 语义一致性检测属于 STS 任务。我们通过 MT-BERT 微调任务来判断改写后 Query 语义是否发生漂移，微调方式如图 8(a) 所示，把原始 Query 和改写 Query 构成句子对，即“[CLS] text_a [SEP] text_b [SEP]”的形式，送入到 MT-BERT 中，通过“[CLS]”判断两个 Query 之间关系。实验证明，基于 MT-BERT 微调的方案在 Benchmark 上准确率和召回率都超过原先的 XGBoost 分类模型。

序列标注

序列标注是 NLP 基础任务之一，给定一个序列，对序列中的每个元素做一个标记，或者说给每一个元素打一个标签，如中文命名实体识别、中文分词和词性标注等任务都属于序列标注的范畴。命名实体识别（Named Entity Recognition，NER），是指识别文本中具有特定意义的实体，主要包括人名、地名、机构名、专有名词等，以及时间、数量、货币、比例数值等文字。

在美团点评业务场景下，NER 主要需求包括搜索 Query 成分分析，UGC 文本中的特定实体（标签）识别 / 抽取，以及客服对话中的槽位识别等。NLP 中心和酒店搜索算法团队合作，基于 MT-BERT 微调来优化酒店搜索 Query 成分分析任务。酒店 Query 成分分析任务中，需要识别出 Query 中城市、地标、商圈、品牌等不同成分，用于确定后续的召回策略。

在酒店搜索 Query 成分分析中，我们对标签采用“BME”编码格式，即对一个实体，第一个字需要预测成实体的开始 B，最后一个字需要预测成实体的结束 E，中间部分则为 M。以图 13 中酒店搜索 Query 成分分析为例，对于 Query“北京昆泰酒店”，成分分析模型需要将“北京”识别成地点，而“昆泰酒店”识别成 POI。MT-BERT 预测高频酒店 Query 成分后通过缓存提供线上服务，结合后续召回策略，显著提升了酒店搜索的订单转化率。

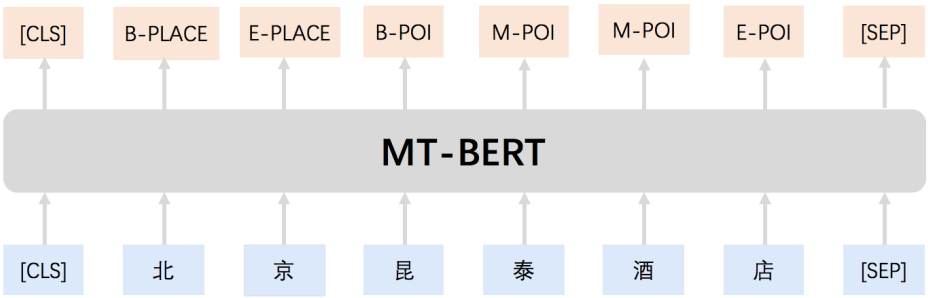


图 13　酒店 Query 的成分分析

一站式 MT-BERT 训练和推理平台建设

为了降低业务方算法同学使用 MT-BERT 门槛，我们开发了 MT-BERT 一站式训练和推理平台，一期支持短文本分类和句间关系分类两种任务，目前已在美团内部开放试用。

基于一站式平台，业务方算法同学上传业务训练数据和选择初始 MT-BERT 模型之后，可以提交微调任务，微调任务会自动分配到 AFO 集群空闲 GPU 卡上自动运行和进行效果验证，训练好的模型可以导出进行部署上线。

融入知识图谱的 MT-BERT 预训练

正如前文所述，尽管在海量无监督语料上进行预训练语言模型取得了很大的成功，但其也存在着一定的不足。BERT 模型通过在大量语料的训练可以判断一句话是否通顺，但是却不理解这句话的语义，通过将美团大脑等知识图谱中的一些结构化先验知识融入到 MT-BERT 中，使其更好地对生活服务场景进行语义建模，是需要进一步探索的方向。

MT-BERT 模型的轻量化和小型化

MT-BERT 模型在各个 NLU 任务上取得了惊人的效果，由于其复杂的网络结构和庞大的参数量，在真实工业场景下上线面临很大的挑战。如何在保持模型效果的前提下，精简模型结构和参数已经成为当前热门研究方向。我们团队在低精度量化、模型裁剪和知识蒸馏上已经做了初步尝试，但是如何针对不同的任务类型选择最合适的

模型轻量化方案，还需要进一步的研究和探索。

参考文献

[1] Peters, Matthew E., et al. “Deep contextualized word representations.” arXiv preprint arXiv:1802.05365 (2018).

[2] Howard, Jeremy, and Sebastian Ruder. “Universal language model fine-tuning for text classification.” arXiv preprint arXiv:1801.06146 (2018).

[3] Vaswani, Ashish, et al. “Attention is all you need.” Advances in neural information processing systems. 2017.

[4] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving Language Understanding by Generative Pre-Training. Technical report, OpenAI.

[5] Devlin, Jacob, et al. “Bert: Pre-training of deep bidirectional transformers for language understanding.” arXiv preprint arXiv:1810.04805 (2018).

[6] Ming Zhou. “The Bright Future of ACL/NLP.” Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics. (2019).

[7] Deng, Jia, et al. “Imagenet: A large-scale hierarchical image database.” 2009 IEEE conference on computer vision and pattern recognition. Ieee, (2009).

[8] Girshick, Ross, et al. “Rich feature hierarchies for accurate object detection and semantic segmentation.” Proceedings of the IEEE conference on computer vision and pattern recognition. 2014.

[9] Mikolov, Tomas, et al. “Distributed representations of words and phrases and their compositionality.” Advances in neural information processing systems. 2013.

[10] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global vectors for word representation. In EMNLP.

[11] Oren Melamud, Jacob Goldberger, and Ido Dagan.2016. context2vec: Learning generic context embedding with bidirectional lstm. In CoNLL.

[12] Hochreiter, Sepp, and J ü rgen Schmidhuber. “Long short-term memory.” Neural computation 9.8 (1997): 1735-1780.

[13] 张俊林 . 从 Word Embedding 到 BERT 模型—自然语言处理中的预训练技术发展史 . https://zhuanlan.zhihu.com/p/49271699

[14] Sebastion Ruder. “NLP’s ImageNet moment has arrived.” http://ruder.io/nlp-imagenet/. (2019)

[15] Liu, Yinhan, et al. “Roberta: A robustly optimized BERT pretraining approach.” arXiv preprint arXiv:1907.11692 (2019).

[16] 郑坤 . 使用 TensorFlow 训练 WDL 模型性能问题定位与调优 . https://tech.meituan.com/2018/04/08/tensorflow-performance-bottleneck-analysis-on-hadoop.html

[17] Uber. “Meet Horovod: Uber’s Open Source Distributed Deep Learning Framework for TensorFlow”. https://eng.uber.com/horovod/

[18] Goyal, Priya, et al. “Accurate, large minibatch sgd: Training imagenet in 1 hour.” arXiv preprint arXiv:1706.02677 (2017).

[19] Baidu. https://github.com/baidu-research/baidu-allreduce
[20] Micikevicius, Paulius, et al. “Mixed precision training.” arXiv preprint arXiv:1710.03740 (2017).
[21] 仲远，富峥等 . 美团餐饮娱乐知识图谱——美团大脑揭秘 . https://tech.meituan.com/2018/11/22/meituan-brain-nlp-01.html
[22] Hinton, Geoffrey, Oriol Vinyals, and Jeff Dean. “Distilling the knowledge in a neural network.” arXiv preprint arXiv:1503.02531 (2015).
[23] Google. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. https://openreview.net/pdf?id=H1eA7AEtvS. (2019)

作者简介

杨扬，佳昊，礼斌，任磊，峻辰，玉昆，张欢，金刚，王超，王珺，富峥，仲远，都来自美团搜索与 NLP 部。

招聘信息

搜索与 NLP 部是美团人工智能技术研发的核心团队，致力于打造高性能、高扩展的搜索引擎和领先的自然语言处理核心技术和服务能力，依托搜索排序、NLP（自然语言处理）、Deep Learning（深度学习）、Knowledge Graph（知识图谱）等技术，处理美团海量文本数据，打通餐饮、旅行、休闲娱乐等本地生活服务各个场景数据，不断加深对用户、场景、查询和服务的理解，高效地支撑形态各样的生活服务搜索，解决搜索场景下的多意图、个性化、时效性问题，给用户极致的搜索体验，构建美团知识图谱，搭建通用 NLP Service，为美团各项业务提供智能的文本语义理解服务。我们的团队既注重 AI 技术的落地，也开展中长期的搜索、NLP 及知识图谱基础研究。目前项目及业务包括搜索引擎研发、知识图谱、智能客服、语音语义搜索、文章评论语义理解、智能助理等。

美团搜索与 NLP 部诚招智能对话算法专家、推荐算法专家、知识图谱算法专家、NLP 算法专家、数据挖掘专家，以及搜索引擎架构师、高级后台研发工程师、前端技术开发资深工程师、测试工程师、数据工程师等众多岗位，欢迎有兴趣的同学，投递简历至：tech@meituan.com（邮件标题注明：岗位名称 + 美团搜索与 NLP 部）。

深度学习在搜索业务中的探索与实践

艺涛

本文根据美团高级技术专家翟艺涛在 2018 QCon 全球软件开发大会上的演讲内容整理而成，内容有修改。

引言

2018 年 12 月 31 日，美团酒店单日入住间夜突破 200 万，再次创下行业的新纪录，而酒店搜索在其中起到了非常重要的作用。本文会首先介绍一下酒店搜索的业务特点，作为 O2O 搜索的一种，酒店搜索和传统的搜索排序相比存在很大的不同。第二部分介绍深度学习在酒店搜索 NLP 中的应用。第三部分会介绍深度排序模型在酒店搜索的演进路线，因为酒店业务的特点和历史原因，美团酒店搜索的模型演进路线可能跟大部分公司都不太一样。最后一部分是总结。

酒店搜索的业务特点

最大的连接器

美团的使命是帮大家“Eat Better，Live Better”，所做的事情就是连接人与服务。用户在美团平台可以找到他们所需要的服务，商家在美团可以售卖自己提供

的服务，而搜索在其中扮演的角色就是“连接器”。大部分用户通过美团 App 找酒店是从搜索开始的，搜索贡献了大部分的订单，是最大的流量入口。在美团首页点击“酒店住宿”图标，就会进入上图右侧的搜索入口，用户可以选择城市和入住时间并发起搜索。

查询引导

推荐

搜索&排序

酒店搜索技术团队的工作不仅有搜索排序，还有查询引导、推荐等工作，查询引导如搜索智能提示、查询纠错等。之所以还有推荐的工作，是因为很多用户在发起搜索时不带查询词，本质上属于推荐，此外还有特定场景下针对少无结果的推荐等。本文主要介绍搜索排序这方面的工作。

不同搜索对比

现在，大家对搜索都很熟悉，常见的有网页搜索，比如 Google、百度、搜狗等；商品搜索，像天猫、淘宝、京东等；还有就是 O2O（Online To Offline）的搜索，典型的就是酒店的搜索。虽然都是搜索，但是用户使用搜索的目的并不相同，包括找信息、找商品、找服务等等，不同搜索之间也存在很大的差别。

	酒店搜索（O2O）	网页搜索	商品搜索
目标	交易	相关性	交易
个性化	高	低	高
结构化数据	是	否	是
位置约束	高	低	低
供给约束	区域	无	全国

上图对不同搜索进行了简单对比，可以从 5 个维度展开。首先是目标维度。因为用户是来找信息，网页搜索重点是保证查询结果和用户意图的相关性，而在商品搜索和酒店搜索中，用户的主要目的是查找商品或服务，最终达成交易，目标上有较大区别。用户使用不同搜索的目的不同，从而导致不同搜索对个性化程度的要求不同。交易属性的搜索，包括商品搜索和酒店搜索，对个性化程度的要求都比较高，因为不同用户的消费水平不同，偏好也不一样。

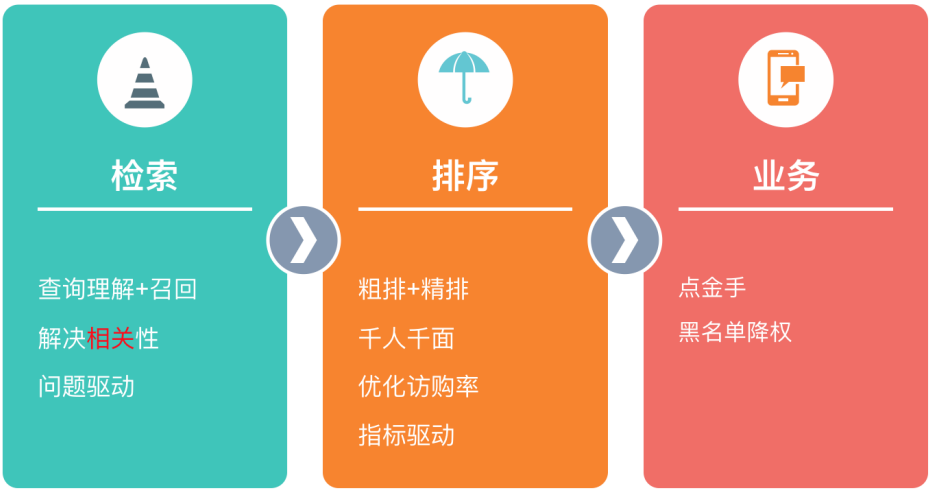
在技术层面上，也存在很多不同点。网页搜索会索引全网的数据，这些数据不是它自己生产，数据来源非常多样，包括新闻、下载页、视频页、音乐页等各种不同的形态，所以整个数据是非结构化的，差异也很大。这意味着网页搜索需要拥有两种技术能力，数据抓取能力和数据解析能力，它们需要抓取网页并解析形成结构化数据。在这个层面上，酒店搜索和商品搜索相对就“幸福”一些，因为数据都是商家提交的结构化数据，相对来说更加规范。

此外，酒店作为一种 O2O 的服务，用户在线上（Online）下单，最终需要到线下（Offline）去消费，所以就有一个位置上的约束，而位置的约束也就导致出现供给侧的约束，供给只能在某个特定位置附近。比如北京大学方圆几公里之内的酒店。这两点约束在网页搜索和商品搜索中就不用考虑，网页可以无限次的进行阅读。商品搜索得益于快递业的快速发展，在北京也可以买到来自浙江的商品，供给侧的约束比较小。

介绍完不同搜索产品的特点，接下来看不同搜索产品的优化目标。通用搜索的优化目标是相关性，评价指标是 DCG、NDCG、MAP 等这些指标，要求查询结果和用户意图相关。对商品搜索来说，不同电商平台的优化目标不太一样，有的目标是最大化 GMV，有的目标是最大化点击率，这些在技术上都可以实现。

而对酒店搜索而言，因为它属于 O2O 的业务形态，线上下单，线下消费，这就要求搜索结果必须和用户的查询意图“强相关”。这个“强相关”包括两层含义，显性相关和隐性相关。举个例子，用户搜索“北京大学”，那么他的诉求很明确，就是要找“北京大学”附近的酒店，这种属于用户明确告诉平台自己的位置诉求。但是，如果用户在本地搜索“七天”，即使用户没有明确说明酒店的具体位置，我们也知道，用户可能想找的是距离自己比较近的“七天酒店”，这时候就需要建模用户的隐性位置诉求。

美团是一个交易平台，大部分用户使用美团是为了达成交易，所以要优化用户的购买体验。刻画用户购买体验的核心业务指标是访购率，用来描述用户在美团是否顺畅的完成了购买，需要优化访购率这个指标。总结一下，酒店搜索不仅要解决相关性，尽量优化用户购买体验、优化访购率等指标，同时还要照顾到业务诉求。



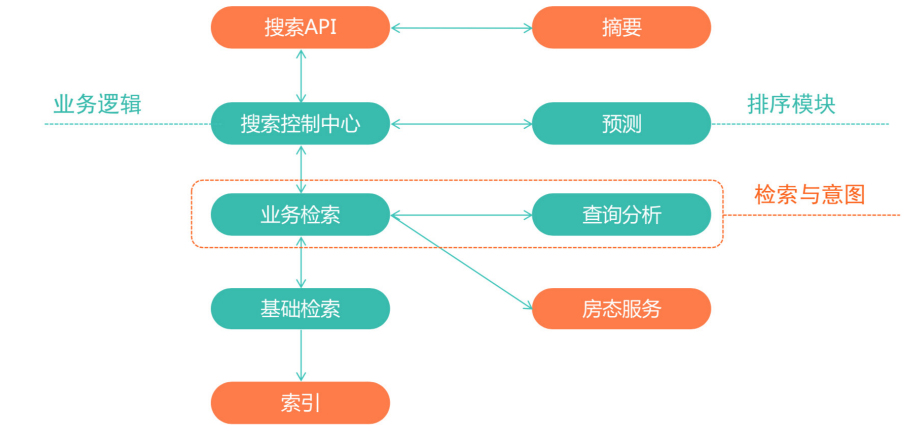
根据上面的分析，酒店搜索的整个搜索框架就可以拆分成三大模块：检索、排序以及业务规则。检索层包括查询理解和召回两部分，主要解决相关性问题。查询理解做的事情就是理解用户意图，召回根据用户意图来召回相关的酒店，两者强耦合，需要放在一起。检索的核心是语义理解，比如用户搜索“北京大学”，平台就知道用户想找的是“北京大学附近的酒店”，所以这个模块的优化方式是问题驱动，不断地发现问题、解决问题来进行迭代。

接下来，从检索模块检索出来的酒店都已经是满足用户需求的酒店了。还是上面“北京大学”的那个例子，检索模块已经检索出来几百家“北京大学”附近的酒店，这些都是和用户的查询词“北京大学”相关的，怎么把用户最有可能购买的酒店排到前面呢？这就是排序模块要做的事情。

排序模块使用机器学习和深度学习的技术提供“千人千面”的排序结果，如果是经常预定经济连锁型酒店的用户，排序模块就把经济连锁型酒店排到前面。针对消费水平比较高，对酒店要求比较高的用户，排序模块就把高档酒店排到前面，对每个用户都可以做到个性化定制。排序属于典型的技术驱动模块，优化目标是访购率，用这个技术指标驱动技术团队不断进行迭代和优化。

最后是业务层面，比如有些商家会在美团上刷单作弊，针对这些商家需要做降权处理。

整体框架

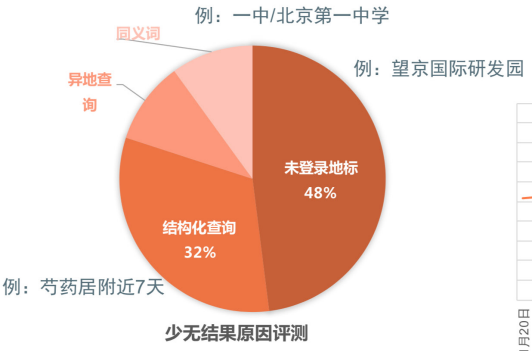


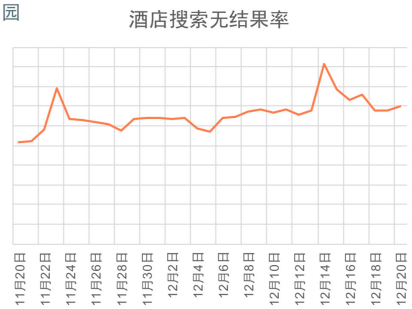
上图是搜索的整体框架，这里详细描述下调用过程：

- 搜索 API 负责接收用户的查询词并发送给搜索控制中心。
- 控制中心把接收到的查询请求发送到检索与意图模块，搜索词会先经过查询分析模块做用户的查询意图分析，分析完之后，会把用户的查询意图分析结果传回去给业务检索模块，业务检索模块根据意图识别结果形成查询条件，然后去基础检索端查询结果。
- 基础检索访问索引得到查询结果后，再把结果返回给上层。
- 业务检索模块获取基础的检索结果后，会调用一些外部服务如房态服务过滤一些满房的酒店，再把结果返回给控制中心。
- 此时，控制中心得到的都是和用户查询意图强相关的结果，这时就需要利用机器学习技术做排序。通过预测模块对每个酒店做访购率预测，控制中心获取预测模块的排序结果后，再根据业务逻辑做一些调整，最终返回结果给搜索 API。

可以看到，模块划分和前文描述的思想一致，检索模块主要解决用户意图识别和召回问题，也就是解决相关性。预测模块做访购率预测，业务逻辑放在搜索控制中心实现。接下来会介绍一下意图理解和排序模块中涉及的一些深度学习技术。

查询理解：问题

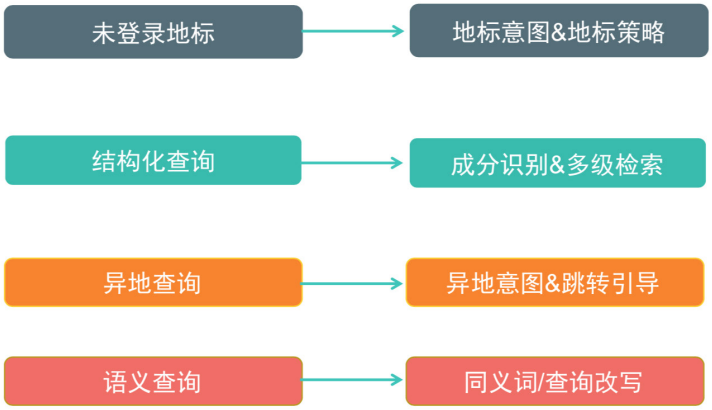




先来看下查询理解的问题，这个模块通过数据分析和 Case 分析，不断的发现问题、解决问题来迭代优化。之前的评测发现少无结果的原因，主要包括以下几种：

- 地标词：比如用户搜索“望京国际研发园”，但是后台没有一家酒店包含“望京国际研发园”这几个字，其实用户想找的是望京国际研发园附近的酒店。
- 结构化查询：比如芍药居附近 7 天，酒店描述信息中没有“附近”这个词，搜索体验就比较差。这种需要对查询词做成分识别，丢掉不重要的词，并且对不用类别的 Term 走不同的检索域。
- 异地查询：用户在北京搜索“大雁塔”没有结果，其实用户的真实意图是西安大雁塔附近的酒店，这种需要做异地需求识别并进行异地跳转。
- 同义词：在北京搜索“一中”和搜索“北京第一中学”，其实都是同一个意思，需要挖掘同义词。

解决方案



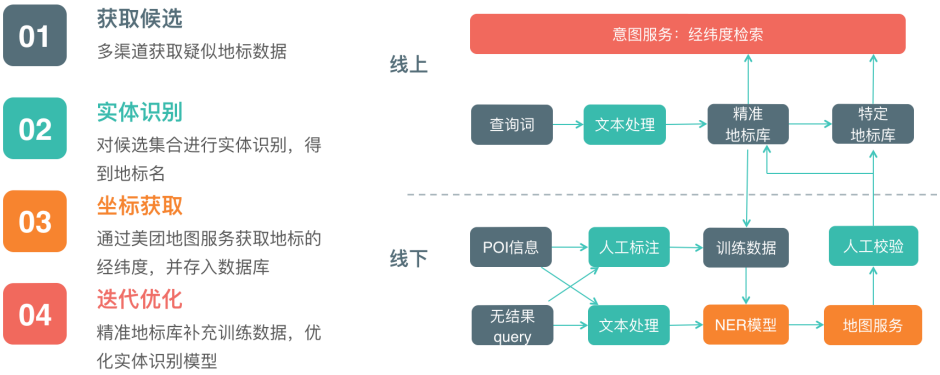
针对这几类问题，我们分别作了以下工作：

- 针对地标词问题，提供地标意图识别和地标策略，把地标类别的查询词改成按经纬度进行画圈检索。
- 针对结构化查询的问题，我们对查询词做了成分识别，设计了少无结果时的多级检索架构。
- 针对异地查询的问题，做异地意图识别和异地的跳转引导。
- 针对语义查询的问题，做同义词和查询改写。

这里的每一个模块都用到了机器学习和深度学习的技术，本文挑选两个酒店搜索中比较特殊的问题进行介绍。

地标问题是 O2O 搜索的一个典型问题，在网页搜索和商品搜索中都较少出现此类问题。当用户搜索类似“望京国际研发园”这种查询词的时候，因为搜索的相关性是根据文本计算的，需要酒店描述中有相关文字，如果酒店的描述信息中没有这个词，那就检索不出来。比如昆泰酒店，虽然就在望京国际研发园旁边，但是它的描述信息中并没有出现“望京国际研发园”，所以就无法检索出来，这会导致用户体验较差。

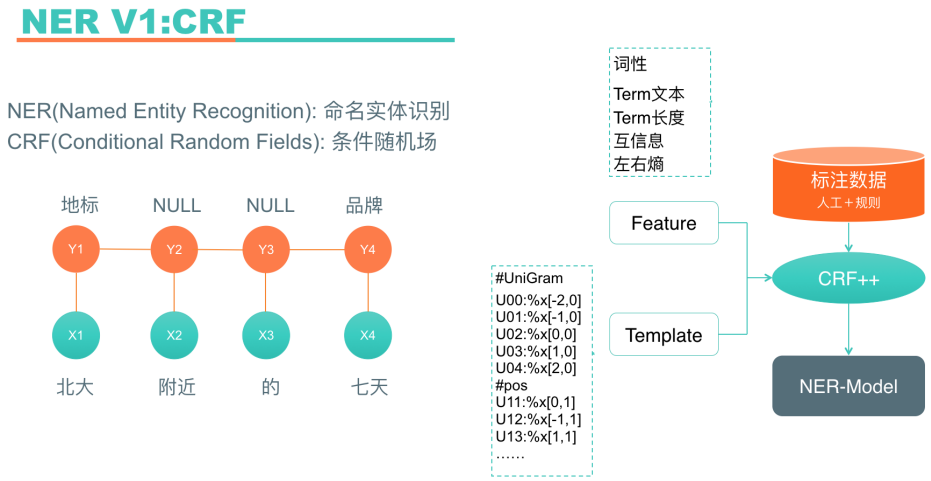
经过分析，我们发现有一类查询词是针对特定地点的搜索，用户的诉求是找特定地点附近的酒店，这种情况下走文本匹配大概率是没有结果的。这个问题的解法是针对这种类型的查询词，从“文本匹配”改成“坐标匹配”，首先分析查询词是不是有地标意图，如果是的话就不走文本匹配了，改走坐标匹配，检索出来这个坐标附近的酒店就可以了。这时就产生了两个问题：第一，怎么确定哪些查询词有地标意图；第二，怎么获取经纬度信息。



针对这个问题，我们做了地标策略，步骤如下：

- 多渠道获取可能包含地标词的候选集，这些候选集包括用户少无结果的查询词，以及一些酒店提供的描述信息。
- 对候选集合进行命名实体识别（NER，Named Entity Recognition），可以得到各个命名实体的类型，标识为“地标”类型的就是疑似地标词。
- 把疑似地标词放到美团地图服务中获取经纬度，经过人工校验无误后，存入线上数据库中；线上来查询请求时，先会去匹配精准地标库，如果匹配成功，说明这个查询词是地标意图，这时就不走文本检索了，直接在意图服务层走经纬度检索。
- 经过人工校验的精准地标库补充到 NER 模型的训练数据中，持续优化 NER 模型。

这里提到了 NER 模型，下面对它做一下详细的介绍。

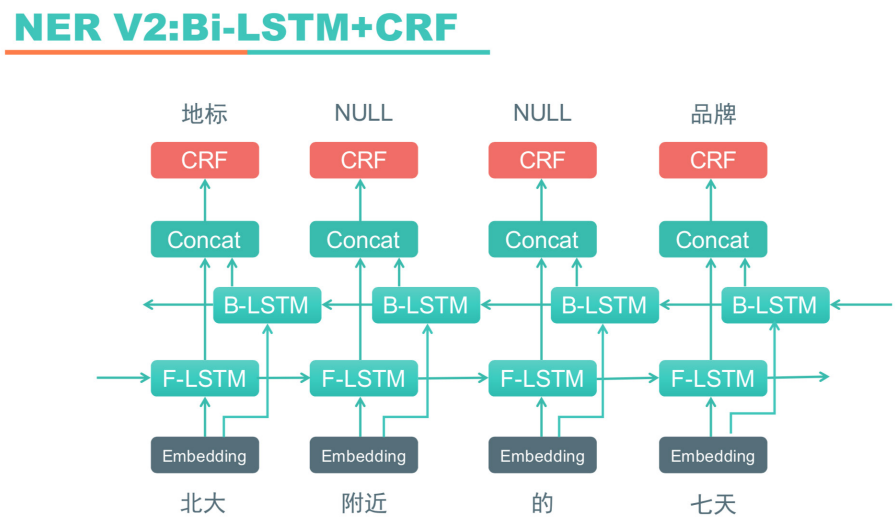


John Lafferty et al. Conditional random fields: Probabilistic models for segmenting and labeling sequence data.ICML2001

NER 是命名实体识别，是机器学习中的序列标注问题，比如输入“北大附近的七天”，就会标注出来每个词的成分，这里“北大”是地标，“七天”是酒店品牌。这里的类别是根据业务特点自己定义的，酒店业务中有地标、品牌、商圈等不同的类别。与分类问题相比，序列标注问题中当前的预测标签不仅与当前的输入特征相关，还与前后的预测标签相关，即预测标签序列之间有强相互依赖关系。

解决序列标注问题的经典模型是 CRF（Conditional Random Field，条件随机场），也是我们刚开始尝试的模型。条件随机场可以看做是逻辑回归的序列化版本，逻辑回归是用于分类的对数线性模型，条件随机场是用于序列化标注的对数线性模型，可以看做是考虑了上下文的分类模型。

机器学习问题的求解就是“数据 + 模型 + 特征”，数据方面先根据业务特点定义了几种实体类别，然后通过“人工 + 规则”的方法标注了一批数据。特征方面提取了包括词性、Term 文本特征等，还定义了一些特征模板，特征模板是 CRF 中人工定义的一些二值函数，通过这些二值函数，可以挖掘命名实体内部以及上下文的构成特点。标注数据、模型、特征都有了，就可以训练 CRF 模型，这是线上 NER 问题的第一版模型。

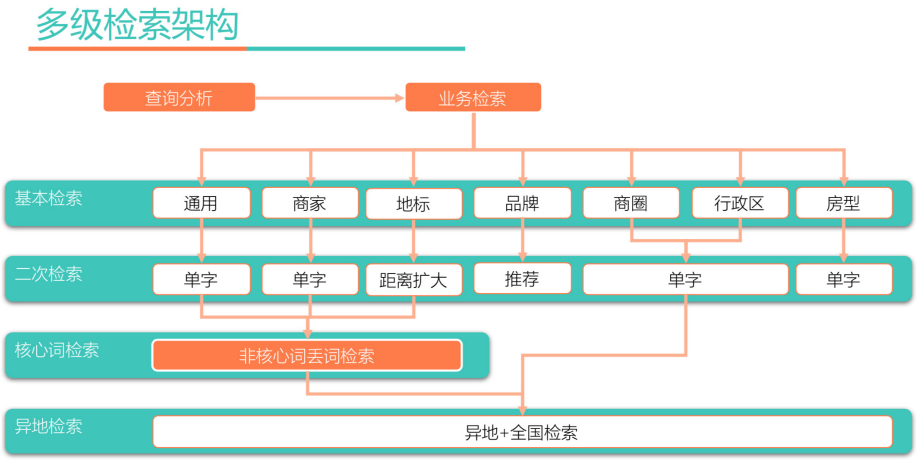


Guillaume Lample et al Neural architectures for named entity recognition. NAACL2016

随着深度学习的发展，用 Word Embedding 词向量作为输入，叠加神经网络单元的方法渐渐成为 NLP 领域新的研究方向。基于双向 LSTM（Long Short-Term Memory）+CRF 的方法成为 NER 的主流方法，这种方法采用双向 LSTM 单元作为特征提取器替代原有的人工特征，不需要专门的领域知识，框架也通用。Embedding 输入也有多种形式，可以是词向量，可以是字向量，也可以是字向量和词向量的拼接。

我们尝试了双向 LSTM+CRF，并在实际应用中做了些改动：由于在 CRF 阶段已经积累了一批人工特征，实验发现把这些特征加上效果更好。加了人工特征的双向 LSTM+CRF 是酒店搜索 NER 问题的主模型。

当然，针对 LSTM+CRF 的方法已经有了很多的改进，比如还有一种 NER 的方法是融合 CNN+LSTM+CRF，主要改进点是多了一个 CNN 模块来提取字级别的特征。CNN 的输入是字级别的 Embedding，通过卷积和池化等操作来提取字级别的特征，然后和词的 Embedding 拼接起来放入 LSTM。这种方法在两个公开数据集上面取得了最好的结果，也是未来尝试的方向之一。



为了解决少无结果的问题，我们设计了多级检索架构，如上图所示，主要分 4 个层次：基本检索、二次检索、核心词检索和异地检索。

- 基本检索会根据查询词的意图选择特定的检索策略，比如地标意图走经纬度检索，品牌意图只检索品牌域和商家名。
- 基本检索少无结果会进行二次检索，二次检索也是分意图的，不同意图类型会有不同的检索策略，地标意图是经纬度检索的，二次检索的时候就需要扩大检索半径；品牌意图的查询词，因为很多品牌在一些城市没有开店，比如香格里拉在很多小城市并没有开店，这时比较好的做法，是推荐给用户该城市最好的酒店。
- 如果还是少无结果，会走核心词检索，只保留核心词检索一遍。丢掉非核心词有多种方式，一种是删除一些运营定义的无意义词，一种是保留 NER 模型识别出来的主要实体类型。此外还有一个 TermWeight 的模型，对每个词都有一个重要性的权重，可以把一些不重要的词丢掉。
- 在还没有结果的情况下，会选择”异地 + 全国“检索，即更换城市或者在全国范围内进行检索。

多级检索架构上线后，线上的无结果率就大幅度降低了。

排序

排序其实是一个典型的技术问题，业界应用比较广泛的有广告排序和推荐排序，广告排序比如 Google 和百度的关键字广告排序，今日头条、腾讯的展示广告排序。推荐排序比如快手、抖音这些短视频平台，以及各大 App、浏览器的信息流。广告排序和推荐排序优化的目标都是点击率，技术栈也比较相似，包括 LR/FTRL、FM/FFM、GBDT、DNN 等模型。

跟以上两种排序应用相比，酒店排序有自己的业务特点，因为美团酒店具有 LBS 属性和交易属性，天生自带很多连续特征，如酒店价格、酒店评分、酒店离用户的距离等，这些连续特征是决定用户购买行为的最重要因素。优化目标也不一样，大部分场景下酒店搜索的优化目标是访购率，部分场景下优化目标是点击率。在技术层面，酒店排序整体的技术栈和广告、推荐比较相似，都可以使用 LR/FTRL、FM/FFM、GBDT、DNN 等模型。

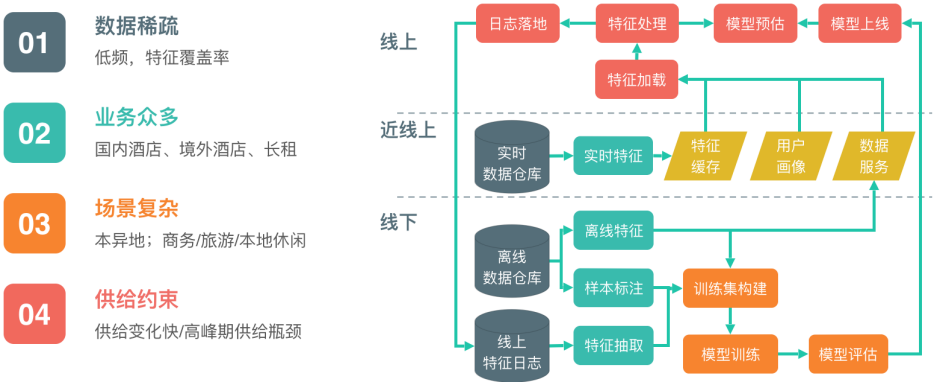
面临的挑战

具体到酒店排序工作，我们面临一些不一样的挑战，主要包括以下 4 点：

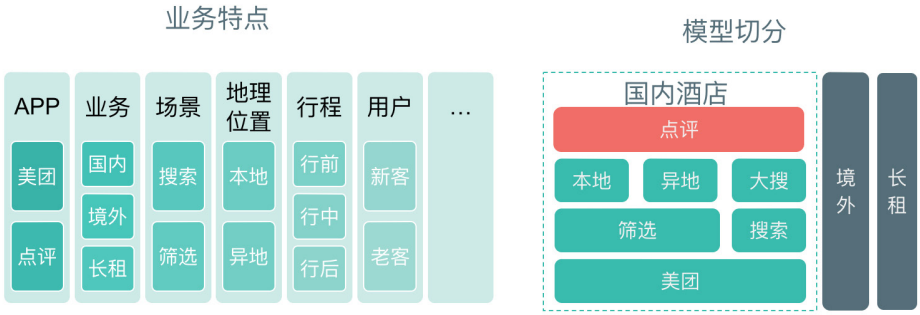
1. 数据稀疏。住酒店本身是一种低频行为，大部分用户一年也就住一两次，导致很多特征的覆盖率比较低。
2. 业务众多。美团酒店包括国内酒店业务、境外酒店业务，以及长租、钟点房

等业务，同时有美团和点评两个不同的 App。

3. 场景复杂。按照用户的位置可以分成本地和异地，按照用户的诉求可以分成商务、旅游、本地休闲等几大类，这些用户之间差异很明显。比如商务用户会有大量复购行为，典型例子是美团员工的出差场景，美团在上海和北京各有一个总部，如果美团的同学去上海出差，大概率会在公司差旅标准内选一家离公司近的酒店，从而会在同一家酒店产生大量的复购行为；但是如果是一个旅游用户，他就很少反复去同一个地方。
4. 供给约束。酒店行业供给的变化很快，一个酒店只有那么多房间，一天能提供的间夜量是固定的，全部订出的话，用户提价也不会提供新的房间，这种情况在劳动节、国庆这种节假日特别明显。



上图右侧是排序的整体架构图，分为线下、线上和近线上三个部分。在线下部分，主要做离线的模型调优和评估，线上部分做预测。这里比较特别的是近线上部分，我们在实时层面做了大量的工作，包括用户的实时行为、酒店实时价格、实时库存等等，以应对供给变化快的特点。



这里介绍一个业务特点导致的比较独特的问题：**模型切分**。美团酒店有很多业务场景，包括国内酒店、境外酒店、长租、钟点房等；还有两个 App，美团 App 和大众点评 App；还有搜索和筛选两种场景，搜索带查询词，筛选没有查询词，两种场景差异较大；从地理位置维度，还可以分成本地和异地两种场景。

面对这么多的业务场景，第一个问题就是模型怎么设计，是用统一的大模型，还是分成很多不同的小模型？我们可以用一个大模型 Cover 所有的场景，用特征来区分不同场景的差异，好处是统一模型维护和优化成本低。也可以划分很多小模型，这里有一个比较好的比喻，多个专科专家会诊，胜过一个全科医生。切分模型后，可以避免差异较大的业务之间互相影响，也方便对特殊场景进行专门的优化。

在模型切分上，主要考虑三个因素：

- 第一，业务之间的差异性。比如长租和境外差异很大，国内酒店和境外业务差异也很大，这种需要拆分。
- 第二，细分后的数据量。场景分的越细，数据量就越小，会导致两个问题，一是特征的覆盖率进一步降低；二是数据量变小后，不利于后续的模型迭代，一些复杂模型对数据量有很高的要求。我们做过尝试，国内酒店场景下，美团和大众点评两个 App 数据量都很大，而且用户也很不一样，所以做了模型拆分；但是境外酒店，因为本身是新业务数据量较小，就没有再进行细分。
- 第三，一切以线上指标为准。我们会做大量的实验，看当前数据量下怎么拆分效果更好，比如美团 App 的国内酒店，我们发现把搜索和筛选拆开后，效果

更好；筛选因为数据量特别大，拆分成本、异地效果也更好，但是如果搜索场景拆分成本地、异地模型就没有额外收益了。最终，一切都要以线上的实际表现为准。

模型演进

接下来介绍一下排序模型的演进过程，因为业务特点及历史原因，酒店搜索的排序模型走了一条不一样的演进路线。大家可以看业界其他公司点击率模型的演进，很多都是从 LR/FTRL 开始，然后进化到 FM/FFM，或者用 GBDT+LR 搞定特征组合，然后开始 Wide&Deep。

酒店搜索的演进就不太一样。酒店业务天生自带大量连续特征，如酒店价格、酒店和用户的距离、酒店评分等，因此初始阶段使用了对连续特征比较友好的树模型。在探索深度排序模型的时候，因为已经有了大量优化过的连续特征，导致我们的整个思路也不太一样，主要是借鉴一些模型的思想，结合业务特点做尝试，下面逐一进行介绍。

非线性

GBDT的改进，通过树节点的分裂实现非线性

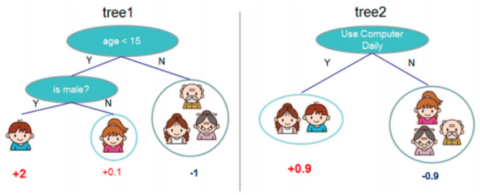
特征组合

树的层次结构实现不同特征的自动组合

适合酒店业务特点

连续特征多，如酒店评分、价格、距离等

XGB:eXtreme Gradient Boosting



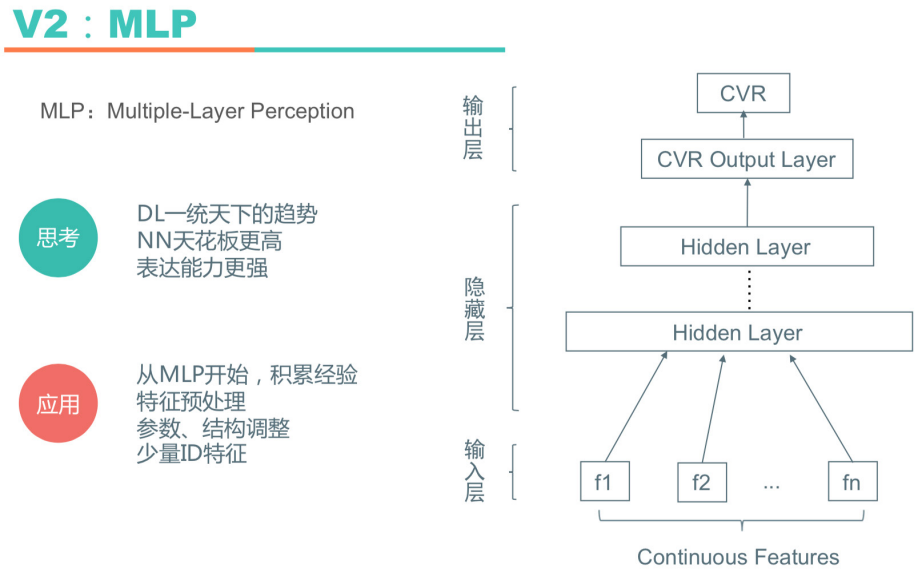
T Chen, C Guestrin. Xgboost: A scalable tree boosting system. KDD2016

初始阶段线上使用的模型是 XGB(XGBoost, eXtreme Gradient Boosting)。作为 GBDT 的改进，XGB 实现了非线性和自动的特征组合。树节点的分裂其实就实现了非线性，树的层次结构实现了不同特征的自动组合，而且树模型对特征的包容性非常好，树的分裂通过判断相对大小来实现，不需要对特征做特殊处理，适合连续特征。

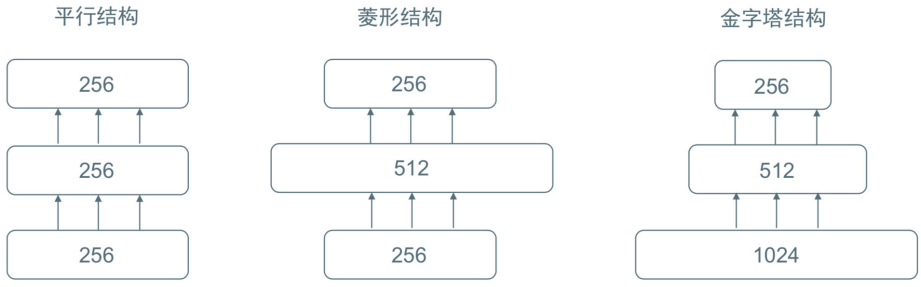
树模型的这些特点确实很适合酒店这种连续特征多的场景，至今为止，XGB 都是数据量较小场景下的主模型。但是树模型优化到后期遇到了瓶颈，比如特征工程收益变小、增大数据量没有额外收益等，此外树模型不适合做在线学习的问题愈发严重。酒店用户在劳动节、国庆节等节假日行为有较大不同，这时需要快速更新模型，我们尝试过只更新最后几棵树的做法，效果不佳。考虑到未来进一步的业务发展，有必要做模型升级。

模型探索的原则是从简单到复杂，逐步积累经验，所以首先尝试了结构比较简单的 MLP(Multiple-Layer Perception) 多层感知机，也就是全连接神经网络。神经网络是一种比树模型“天花板”更高的模型，“天花板”更高两层意思：第一层意思，可以优化提升的空间更大，比如可以进行在线学习，可以做多目标学习；第二层意思，模型的容量更大，“胃口”更大，可以“吃下”更多数据。此外它的表达能力也更强，可以拟合任何函数，网络结构和参数可以调整的空间也更大。但是它的优点同时也是它的缺点，因为它的网络结构、参数等可以调整的空间更大，神经网需要做很

多的参数和网络结构层面的调整。



上图是 MLP 的网络结构图，包含输入层、若干个隐藏层、输出层。在很长一段时间内，在特征相同的情况下，MLP 效果不如 XGB，所以有段时间线上使用的是 XGB 和 MLP 的融合模型。后来经过大量的网络结构调整和参数调整，调参经验越来越丰富，MLP 才逐步超越 XGB。这里额外说明一下，酒店搜索中有少量的 ID 类特征，在第一版 MLP 里 ID 类特征是直接当做连续特征处理的。比如城市 ID，ID 的序关系有一定的物理意义，大城市 ID 普遍较小，小城市开城晚一些，ID 较大。

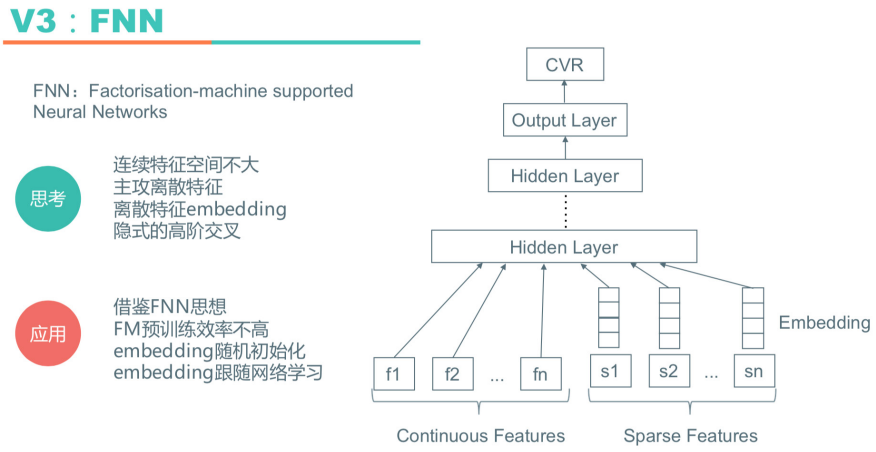


在 MLP 阶段我们对网络结构做了大量实验，尝试过三种网络结构：平行结构、

菱形结构、金字塔结构。在很多论文中提到三者相比平行结构效果最好，但是因为酒店搜索的数据不太一样，实验发现金字塔结构效果最好，即上图最右边的“1024-512-256”的网络结构。同时还实验了不同网络层数对效果的影响，实验发现 3-6 层的网络效果较好，更深的网络没有额外收益而且线上响应时间会变慢，后面各种模型探索都是基于 3 到 6 层的金字塔网络结构进行尝试。

MLP 上线之后，我们开始思考接下来的探索方向。在树模型阶段，酒店搜索组就在连续特征上做了很多探索，连续特征方面很难有比较大的提升空间；同时业界的研究重点也放在离散特征方面，所以离散特征应该是下一步的重点方向。

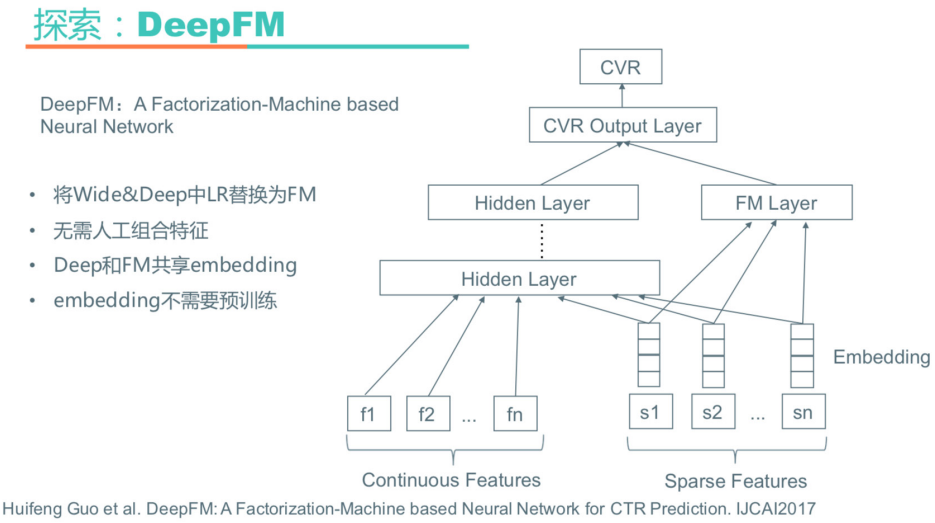
深度排序模型对离散特征的处理有两大类方法，一类是对离散特征做 Embedding，这样离散特征就可以表示成连续的向量放到神经网络中去，另一类是 Wide&Deep，把离散特征直接加到 Wide 侧。我们先尝试了第一种，即对离散特征做 Embedding 的方法，借鉴的是 FNN 的思想。其实离散特征做 Embedding 的想法很早就出现了，FM 就是把离散特征表示成 K 维向量，通过把高维离散特征表示成低维向量增加模型泛化能力。



Weinan Zhang et al. Deep Learning over Multi-Field Categorical Data: A Case Study on User Response Prediction. ECIR 2016

实际使用中，我们稍微做了一些改动，实验中发现使用 FM 预训练的效率不高，所以尝试了不做预训练直接把 Embedding 随机初始化，然后让 Embedding 跟随网络一起学习，实验结果发现比 FM 预训练效果还要好一点。最后的做法是没有用 FM

做预训练，让 Embedding 随机初始化并随网络学习，上图是线上的 V3 模型。



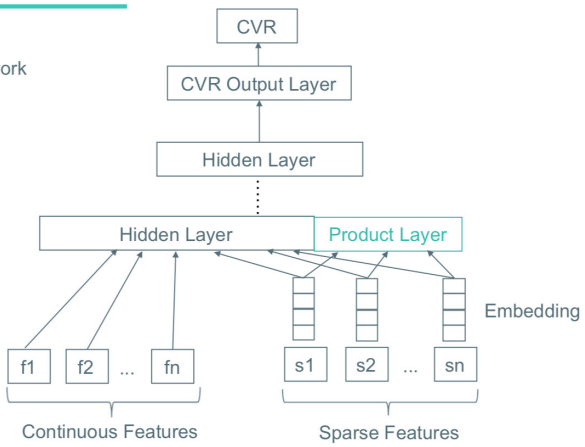
FNN 的成功上线证明离散特征 Embedding 这个方向值得深挖，所以我们接着实验了 DeepFM。DeepFM 相对于 Wide&Deep 的改进，非常类似于 FM 相对 LR 的改进，都认为 LR 部分的人工组合特征是个耗时耗力的事情，而 FM 模块可以通过向量内积的方式直接求出二阶组合特征。DeepFM 使用 FM 替换了 Wide&Deep 中的 LR，离散特征的 Embedding 同时“喂”给神经网和 FM，这部分 Embedding 是共享的，Embedding 在网络的优化过程中自动学习，不需要做预训练，同时 FM Layer 包含了一阶特征和二阶的组合特征，表达能力更强。我们尝试了 DeepFM，线下有提升线上波动提升，并没有达到上线的标准，最终没有全量。

尽管 DeepFM 没有成功上线，但这并没有动摇我们对 Embedding 的信心，接下来尝试了 PNN。PNN 的网络重点在 Product 上面，在点击率预估中，认为特征之间的关系更多是一种 And“且”的关系，而非 Add“加”的关系，例如性别为男且用华为手机的人，他定酒店时属于商务出行场景的概率更高。

探索：PNN

PNN：Product-based Neural Network

- 特征之间‘and’关系
- 显式的2阶特征组合
- 隐式的高阶特征组合
- 内积和外积两种方式



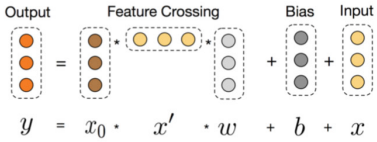
Yanru Qu et al. Product-based neural networks for user response prediction. ICDM2016

PNN 使用了 Product Layer 进行显式的二阶特征组合。上图右边是 PNN 的网络结构图，依然对离散特征做 Embedding，Embedding 向量同时送往隐层和 Product 层，Product 通过内积或者外积的方式，对特征做显式的二阶交叉，之后再送入神经网的隐层，这样可以做到显式的二阶组合和隐式的高阶特征组合。特征交叉基于乘法的运算实现，有两种方式：内积和外积。我们尝试了内积的方式，线下略有提升线上也是波动提升，没有达到上线标准，所以最终也没有全量上线。

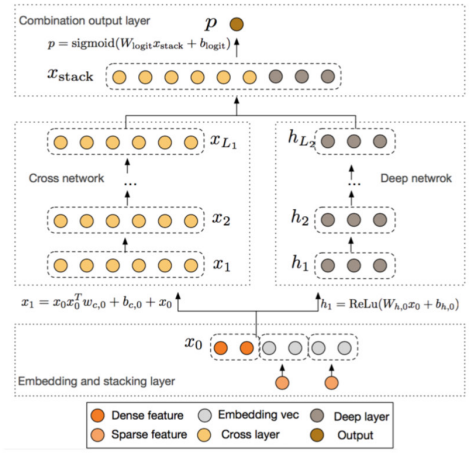
探索：DCN

DCN：Deep & Cross network

- 引入Cross network
- 显式高阶特征交叉
- Cross网络和Deep网络并行
- 显示交叉&隐式交叉
- “元素”级交叉



$$y = x_0 \cdot x' \cdot w + b + x$$



Ruoxi Wang, et al. Deep & Cross Network for Ad Click Predictions. ADKDD2017

PNN 之后我们认为 Embedding 还可以再尝试一下，于是又尝试了 DCN（Deep&Cross Network）。DCN 引入了一个 Cross Network 进行显式的高阶特征交叉。上图右边是论文中的图，可以看到 Deep&Cross 中用了两种网络，Deep 网络和 Cross 网络，两种网络并行，输入都一样，在最后一层再 Stack 到一起。

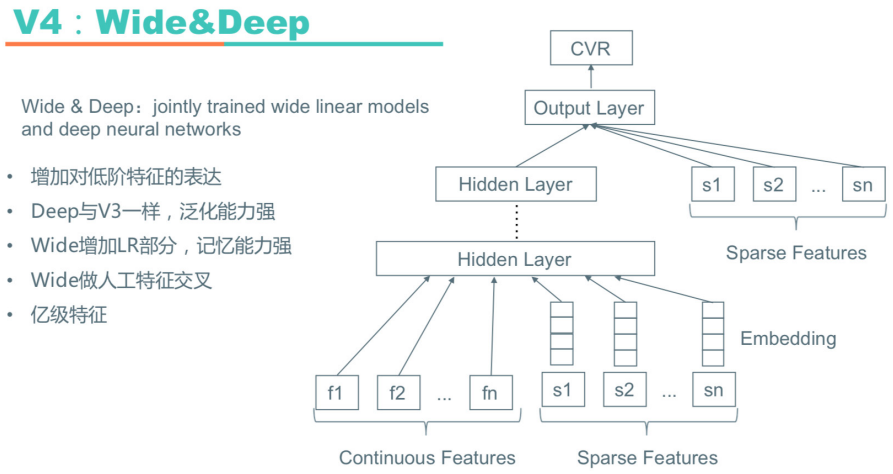
Deep 网络和前面几种网络一样，包括连续特征和离散特征的 Embedding，Cross 网络是 DCN 的特色，在 Cross 网络里面，通过巧妙的设计实现了特征之间的显式高阶交叉。看上图左下角的 Cross 结构示意，这里的 x 是每一层的输入，也就是上一层的输出。Feature Crossing 部分包括了原始输入 x0、本层输入 x 的转置、权重 w 三项，三项相乘其实就做了本层输入和原始输入的特征交叉，x1 就包含了二阶的交叉信息，x2 就包含了三阶的交叉信息，就可以通过控制 Cross 的层数显式控制交叉的阶数。

不得不说，DCN 在理论上很漂亮，我们也尝试了一下。但是很可惜，线下有提升线上波动提升，依然未能达到上线的标准，最终未能全量上线。

经过 DeepFM、PNN、DCN 的洗礼，促使我们开始反思，为什么在学术上特别有效的模型，反而在酒店搜索场景下不能全量上线呢？它们在线下都有提升，在线上也有提升，但是线上提升较小且有波动。

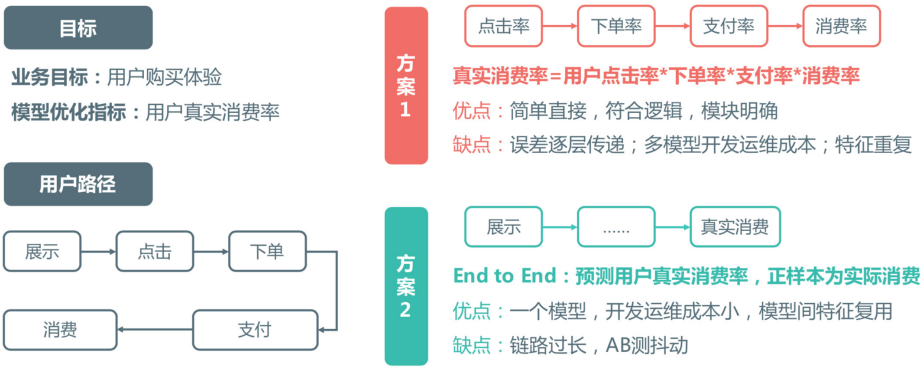
经过认真分析我们发现可能有两个原因：第一，连续特征的影响，XGB 时代尝试了 600 多种连续特征，实际线上使用的连续特征接近 400 种，这部分特征太强了；第二，离散特征太少，离散特征只有百万级别，但是 Embedding 特别适合离散特征多的情况。接下来方向就很明确了：补离散特征的课。

最终，我们还是把目光转回 Wide&Deep。Wide&Deep 同时训练一个 Wide 侧的线性模型和一个 Deep 侧的神经网络，Wide 部分提供了记忆能力，关注用户有过的历史行为，Deep 部分提供了泛化能力，关注一些没有历史行为的 Item。之前的工作主要集中在 Deep 测，对低阶特征的表达存在缺失，所以我们添加了 LR 模块以增加对低阶特征的表达，Deep 部分和之前的 V3 一样。刚开始只用了少量的 ID 类特征，效果一般，后来加了大量人工的交叉特征，特征维度达到了亿级别后效果才得到很好的提升。下图是我们的 V4 模型：



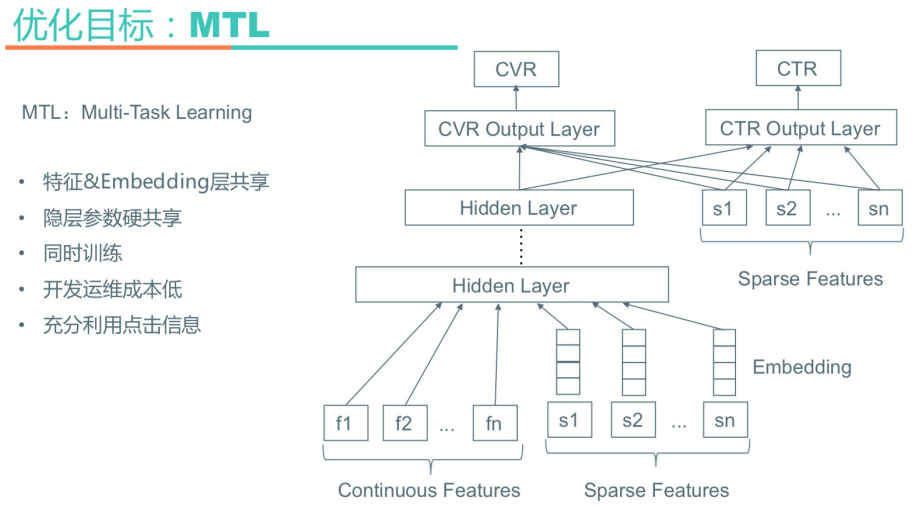
Heng-Tze Cheng et al. 2016. Wide & deep learning for recommender systems. 2016

接下来介绍一下优化目标的迭代过程（后面讲 MTL 会涉及这部分内容）。酒店搜索的业务目标是优化用户的购买体验，模型的优化指标是用户的真实消费率，怎么优化这个目标呢？通过分析用户的行为路径可以把用户的行为拆解成“展示 -> 点击 -> 下单 -> 支付 -> 消费”等 5 个环节，这其中每个环节都可能存在用户流失，比如有些用户支付完成后，因为部分商家确认比较慢，用户等不及就取消了。



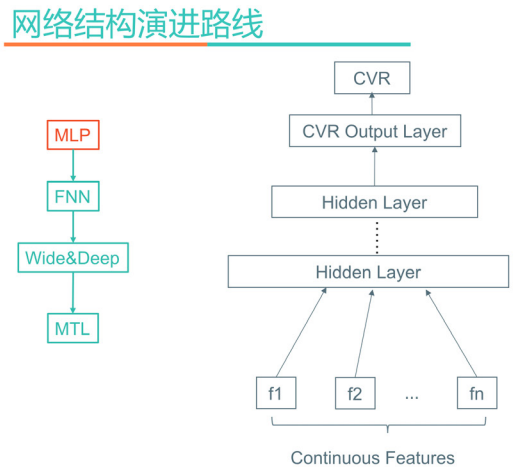
刚开始我们采用了方案 1，对每一个环节建模（真实消费率 = 用户点击率 × 下单率 × 支付率 × 消费率）。优点是非常简单直接且符合逻辑，每个模块分工明确，容易确认问题出在哪里。缺点也很明显，首先是特征重复，4 个模型在用户维度和商

家维度的特征全部一样，其次模型之间是相乘关系且层数过多，容易导致误差逐层传递，此外 4 个模型也增加了运维成本。后来慢慢进化到了方案 2 的“End to End”方式，直接预测用户的真实消费率，这时只需要把正样本设定为实际消费的样本，一个模型就够了，开发和运维成本较小，模型间特征也可以复用，缺点就是链路比较长，上线时经常遇到 AB 测抖动问题。



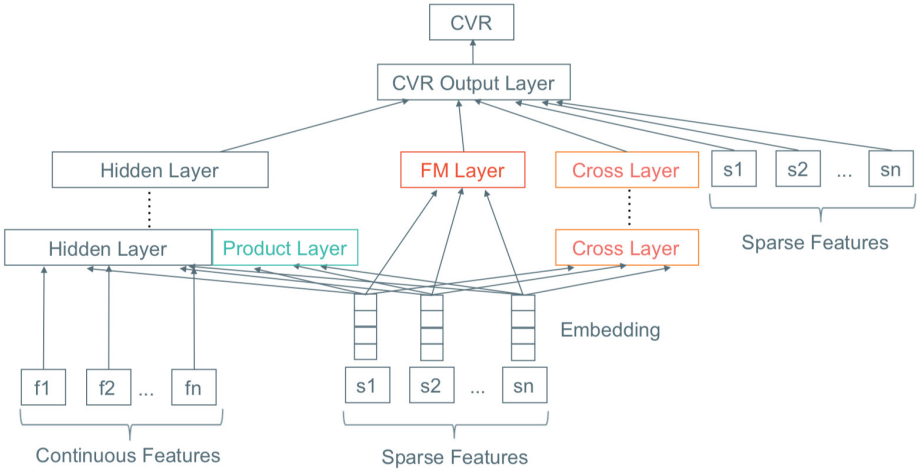
模型切换到神经网络后就可以做多任务学习了，之前树模型时代只预测“End to End”真实访购率，神经网络则可以通过多任务学习同时预测 CTR 展示点击率和 CVR 点击消费率。多任务学习通过硬共享的方式同时训练两个网络，特征、Embedding 层、隐层参数都是共享的，只在输出层区分不同的任务。上图是酒店搜索当前线上的模型，基于 Wide&Deep 做的多任务学习。

网络结构演进路线



上图是酒店搜索排序的深度排序模型演进路线，从 MLP 开始，通过对离散特征做 Embedding 进化到 FNN，中间尝试过 DeepFM、PNN、DCN 等模型，后来加入了 Wide 层进化到 Wide&Deep，现在的版本是一个 MTL 版的 Wide&Deep，每个模块都是累加上去的。

除了上面提到的模型，我们还探索过这个：



这是我们自己设计的混合网络，它融合了 FNN、DeepFM、PNN、DCN、Wide&Deep 等不同网络的优点，同时实现了一阶特征、显式二阶特征组合、显式高阶特征组合、隐式高阶特征组合等，有兴趣的同学可以尝试一下。

不同模型实验结果

不同模型实验结果

模型	离线AUC	线上效果（访购率）
XGBoost	Baseline	Baseline
MLP	+18BP	+0.34%
MLP+XGBoost	+32BP	+0.79%
FNN	+33BP	+0.73%
DeepFM	+39BP	波动提升，未全量
Deep&Cross	+24BP	波动提升，未全量
Wide&Deep	+29BP	无线上实验
Wide&Deep（组合特征）	+54BP	+1.26%

上图是不同模型的实验结果，这里的 BP 是基点（Basis Point），1BP=0.01%。XGB 是 Baseline，MLP 经过很长时间的调试才超过 XGB，MLP 和 XGB 融合模型的效果也很好，不过为了方便维护，最终还是用 FNN 替换了融合模型。Wide&Deep 在开始阶段，提升并没有特别多，后来加了组合特征后效果才好起来。我们 Embedding 上面的尝试，包括 DeepFM、Deep&Cross 等，线下都有提升，线上波动有提升，但是未能达到上线的标准，最终未能全量。

连续特征

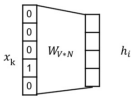
累积分布归一化：$x' = \int_{-\infty}^{x} f(x)dx$

标准化：$x' = \frac{x-\mu}{\sigma}$

根号、对号变换：$x' = \sqrt{x}$　$x' = log_2^x$

离散特征

ID-Embedding：



ID-Combine：$\emptyset_k(x) = \Pi x_i^{c_{ki}}, c_{ki} \in (0,1)$

缺失值

缺失值参数化：$x' = w_{miss} \cdot g(x) + w_{hit} \cdot f(x)$

$g(x) = \begin{cases} 0, x_{正常} \\ 1, x_{缺失} \end{cases}, f(x) = \begin{cases} x, x_{正常} \\ 0, x_{缺失} \end{cases}$

处理方法	AUC
标准化	Baseline
累积分布归一化	+15BP
标准化+累积分布归一化	+26BP
根号+对数等手工变化	+7BP
ID-COMBINE	+25BP
ID-Embedding	+23BP
缺失值处理	+27BP

在特征预处理方面对连续特征尝试了累计分布归一化、标准化，以及手工变换如根号变换、对数变换等；累积分布归一化其实就是做特征分桶，因为连续特征多且分布范围很广，累积分布归一化对酒店搜索的场景比较有效。

离散特征方面尝试了特征 Embedding 及离散特征交叉组合，分别对应 FNN 和 Wide&Deep。这里特别提一下缺失值参数化，因为酒店业务是一种低频业务，特征覆盖率低，大量样本存在特征缺失的情况，如果对缺失特征学一个权重，非缺失值学一个权重效果较好。

激活函数

ReLU　$f(x) = \max(0, x)$

Sigmoid　$f(x) = \frac{1}{1+e^{-x}}$

Leaky_ReLU　$f(x) = \begin{cases} x & , x \geq 0 \\ 0.01x & , x < 0 \end{cases}$

ELU　$f(x) = \begin{cases} x & , x > 0 \\ \alpha(e^x - 1) & , x \leq 0 \end{cases}$

优化器

Adagrad

RMSProp

Adam

$g_t = \nabla_{\theta_{t-1}} f(\theta_{t-1})$

$\Delta\theta_t = -\eta * g_t$

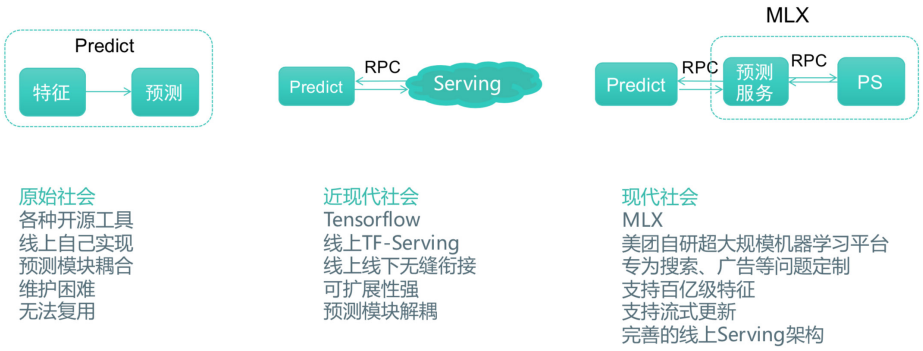
SGD

其他　BN/Dropout/学习率/隐层

激活函数	优化器	AUC
Sigmoid	Adagrad	-300BP
ReLU	Adagrad	Baseline
ELU	Adagrad	-2BP
LeakyReLU	Adagrad	+5BP
ReLU	RMSProp	+10BP
ReLU	Adam	+15BP

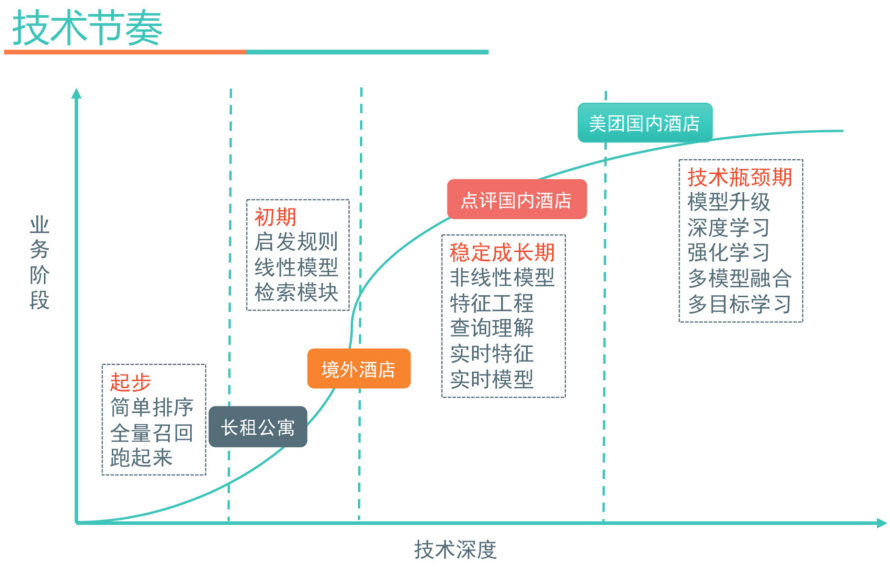
参数调优方面分别尝试了激活函数、优化器等。激活函数尝试过 Sigmoid、

ReLU、Leaky_ReLU、ELU 等；优化器也实验过 Adagrad、Rmsprop、Adam 等；从实验效果看，激活函数 ReLU+Adam 效果最好。刚开始时，加了 Batch Normalization 层和 Dropout 层，后来发现去掉后效果更好，可能和酒店搜索的数据量及数据特点有关。网络结构和隐层数方面用的是 3 到 6 层的金字塔网络。学习率方面的经验是学习率小点比较好，但是会导致训练变慢，需要找到一个平衡点。



下面介绍深度排序模型线上 Serving 架构的演化过程，初始阶段组内同学各自探索，用过各种开源工具如 Keras、TensorFlow 等，线上分别自己实现，预测代码和其他代码都放一起，维护困难且无法复用。

后来组内决定一起探索，大家统一使用 TensorFlow，线上用 TF-Serving，线上线下可以做到无缝衔接，预测代码和特征模块也解耦了。现在则全面转向 MLX 平台，MLX 是美团自研的超大规模机器学习平台，专为搜索、推荐、广告等排序问题定制，支持百亿级特征和流式更新，有完善的线上 Serving 架构，极大地解放了算法同学的生产力。



最后介绍一下我们对搜索排序技术节奏的一些理解，简单来说就是在不同阶段做不同的事情。

在上图中，横轴表示技术深度，越往右技术难度越大，人力投入越大，对人的要求也越高。纵轴是业务阶段。业务阶段对技术的影响包括两方面，数据量和业务价值。数据量的大小，可以决定该做什么事情，因为有些技术在数据量小的时候意义不大；业务价值就更不用说了，业务价值越大越值得“重兵投入”。

- 起步阶段：起步阶段，还没有数据，这时候做简单排序就好，比如纯按价格排序或者距离排序，目的是让整个流程快速地跑起来，能提供最基本的服务。比如 2017 年，美团的长租业务当时就处于起步阶段。
- 业务初期：随着业务的发展，就进入了业务发展初期，订单数慢慢增长，也有了一些数据，这时候可以增加一些启发式规则或者简单的线性模型，检索模型也可以加上。但是由于数据量还比较小，没必要部署很复杂的模型。
- 稳定成长期：业务进一步发展后，就进入了稳定成长期，这时候订单量已经很大了，数据量也非常大了，这段时间是“补课”的时候，可以把意图理解的模块加上，排序模型也会进化到非线性模型比如 XGB，会做大量的特征工程，

实时特征以及实时模型，在这个阶段特征工程收益巨大。

- 技术瓶颈期：这个阶段的特点是基本的东西都已经做完了，在原有的技术框架下效果提升变的困难。这时需要做升级，比如将传统语义模型升级成深度语义模型，开始尝试深度排序模型，并且开始探索强化学习、多模型融合、多目标学习等。

中国有句俗话叫“杀鸡焉用牛刀”，比喻办小事情，何必花费大力气，也就是不要小题大做。其实做技术也一样，不同业务阶段不同数据量适合用不同的技术方案，没有必要过度追求先进的技术和高大上的模型，根据业务特点和业务阶段选择最匹配的技术方案才是最好的。我们认为，**没有最好的模型，只有合适的场景**。

总结

酒店搜索作为 O2O 搜索的一种，和传统的搜索排序相比有很多不同之处，既要解决搜索的相关性问题，又要提供“千人千面”的排序结果，优化用户购买体验，还要满足业务需求。通过合理的模块划分可以把这三大类问题解耦，检索、排序、业务三个技术模块各司其职。在检索和意图理解层面，我们做了地标策略、NER 模型和多级检索架构来保证查询结果的相关性；排序模型上结合酒店搜索的业务特点，借鉴业界先进思想，尝试了多种不同的深度排序模型，走出了一条不一样的模型演进路线。同时通过控制技术节奏，整体把握不同业务的技术选型和迭代节奏，对不同阶段的业务匹配不同的技术方案，只选对的，不选贵的。

参考文献

[1] John Lafferty et al. Conditional random fields: Probabilistic models for segmenting and labeling sequence data.ICML2001.
[2] Guillaume Lample et al Neural architectures for named entity recognition. NAACL2016.
[3] Zhiheng Huang, Wei Xu, and Kai Yu. 2015.
[4] Bidirectional LSTM-CRF models for sequence tagging. arXiv preprint arXiv:1508.01991.
[5] Xuezhe Ma et al.End-to-end Sequence Labeling via Bi-directional LSTM-CNNs-CRF.ACL2016.

[6] T Chen, C Guestrin. XGBoost: A scalable tree boosting system. KDD2016.
[7] Weinan Zhang et al. Deep Learning over Multi-Field Categorical Data: A Case Study on User Response Prediction. ECIR 2016.
[8] Huifeng Guo et al. DeepFM: A Factorization-Machine based Neural Network for CTR Prediction. IJCAI2017.
[9] Yanru Qu et al. Product-based neural networks for user response prediction. ICDM2016.
[10] Heng-Tze Cheng et al. 2016. Wide & deep learning for recommender systems. 2016.In Proceedings of the 1st Workshop on Deep Learning for Recommender Systems.
[11] Ruoxi Wang et al. Deep & Cross Network for Ad Click Predictions. ADKDD2017.

作者简介

艺涛，美团高级技术专家，2016 年加入美团，现负责美团酒店业务搜索排序技术。2010 年毕业于中科院计算所，曾在网易有道等公司工作，先后从事网页搜索、购物搜索、计算广告等方向的研发工作。曾荣获“Kaggle 卫星图像分类大赛”亚军，QCon 明星讲师。

大众点评搜索基于知识图谱的深度学习排序实践

非易　祝升　汤彪　张弓　仲远

1. 引言

挑战与思路

搜索是大众点评 App 上用户进行信息查找的最大入口，是连接用户和信息的重要纽带。而用户搜索的方式和场景非常多样，并且由于对接业务种类多，流量差异大，为大众点评搜索（下文简称点评搜索）带来了巨大的挑战，具体体现在如下几个方面：

1. **意图多样**：用户查找的信息类型和方式多样。信息类型包括 POI、榜单、UGC、攻略、达人等。以找店为例，查找方式包括按距离、按热度、按菜品和按地理位置等多种方式。例如用户按照品牌进行搜索时，大概率是需要寻找距离最近或者常去的某家分店；但用户搜索菜品时，会对菜品推荐人数更加敏感，而距离因素会弱化。
2. **业务多样**：不同业务之间，用户的使用频率、选择难度以及业务诉求均不一样。例如家装场景用户使用频次很低，行为非常稀疏，距离因素弱，并且选择周期可能会很长；而美食多为即时消费场景，用户行为数据多，距离敏感。
3. **用户类型多样**：不同的用户对价格、距离、口味以及偏好的类目之间差异很大；搜索需要能深度挖掘到用户的各种偏好，实现定制化的“千人千面”的搜索。
4. **LBS 的搜索**：相比电商和通用搜索，LBS 的升维效应极大地增加了搜索场景的复杂性。例如对于旅游用户和常驻地用户来说，前者在搜索美食的时候可能会更加关心当地的知名特色商户，而对于距离相对不敏感。

上述的各项特性，叠加上时间、空间、场景等维度，使得点评搜索面临比通

用搜索引擎更加独特的挑战。而解决这些挑战的方法，就需要升级 NLP（Natural Language Processing，自然语言处理）技术，进行深度查询理解以及深度评价分析，并依赖知识图谱技术和深度学习技术对搜索架构进行整体升级。在美团 NLP 中心以及大众点评搜索智能中心两个团队的紧密合作之下，经过短短半年时间，点评搜索核心 KPI 在高位基础上仍然大幅提升，是过去一年半涨幅的六倍之多，提前半年完成全年目标。

基于知识图谱的搜索架构重塑

美团 NLP 中心正在构建全世界最大的餐饮娱乐知识图谱——美团大脑（相关信息请参见《美团大脑：知识图谱的建模方法及其应用》）。它充分挖掘关联各个场景数据，用 NLP 技术让机器“阅读”用户公开评论，理解用户在菜品、价格、服务、环境等方面的喜好，构建人、店、商品、场景之间的知识关联，从而形成一个“知识大脑”[1]。通过将知识图谱信息加入到搜索各个流程中，我们对点评搜索的整体架构进行了升级重塑，图 1 为点评搜索基于知识图谱搭建的 5 层搜索架构。本篇文章是“美团大脑”系列文章第二篇（系列首篇文章请参见《美团餐饮娱乐知识图谱——美团大脑揭秘》），主要介绍点评搜索 5 层架构中核心排序层的演变过程，文章主要分为如下 3 个部分：

1. 核心排序从传统机器学习模型到大规模深度学习模型的演进。
2. 搜索场景深度学习排序模型的特征工程实践。
3. 适用于搜索场景的深度学习 Listwise 排序算法——LambdaDNN。

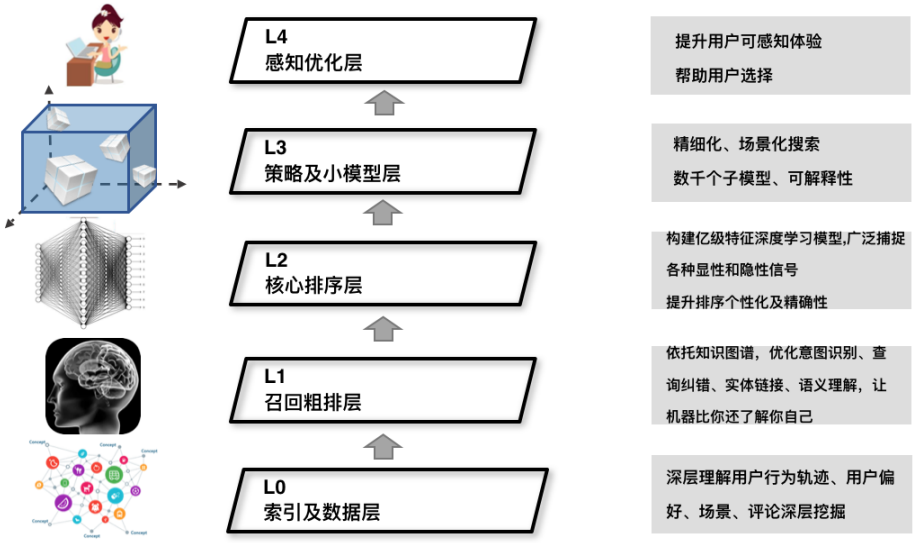


图 1　基于知识图谱的点评搜索 5 层架构

2. 排序模型探索与实践

搜索排序问题在机器学习领域有一个单独的分支，Learning to Rank（L2R）。主要分类如下：

1. 根据样本生成方法和 Loss Function 的不同，L2R 可以分为 Pointwise、Pairwise、Listwise。
2. 按照模型结构划分，可以分为线性排序模型、树模型、深度学习模型，他们之间的组合（GBDT+LR，Deep&Wide 等）。

在排序模型方面，点评搜索也经历了业界比较普遍的迭代过程：从早期的线性模型 LR，到引入自动二阶交叉特征的 FM 和 FFM，到非线性树模型 GBDT 和 GBDT+LR，到最近全面迁移至大规模深度学习排序模型。下面先简单介绍下传统机器学习模型（LR、FM、GBDT）的应用和优缺点，然后详细介绍深度模型的探索实践过程。

传统机器学习模型

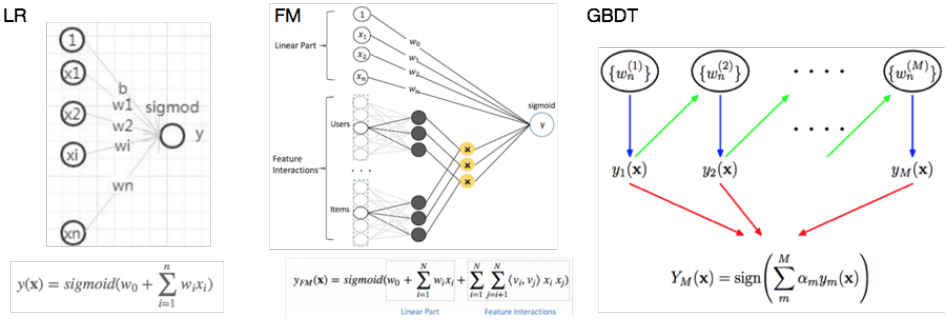


图 2　几种传统机器学习模型结构

1. LR 可以视作单层单节点的线性网络结构。模型优点是可解释性强。通常而言，良好的解释性是工业界应用实践比较注重的一个指标，它意味着更好的可控性，同时也能指导工程师去分析问题优化模型。但是 LR 需要依赖大量的人工特征挖掘投入，有限的特征组合自然无法提供较强的表达能力。
2. FM 可以看做是在 LR 的基础上增加了一部分二阶交叉项。引入自动的交叉特征有助于减少人工挖掘的投入，同时增加模型的非线性，捕捉更多信息。FM 能够自动学习两两特征间的关系，但更高量级的特征交叉仍然无法满足。
3. GBDT 是一个 Boosting 的模型，通过组合多个弱模型逐步拟合残差得到一个强模型。树模型具有天然的优势，能够很好的挖掘组合高阶统计特征，兼具较优的可解释性。GBDT 的主要缺陷是依赖连续型的统计特征，对于高维度稀疏特征、时间序列特征不能很好的处理。

深度神经网络模型

随着业务的发展，在传统模型上取得指标收益变得愈发困难。同时业务的复杂性要求我们引入海量用户历史数据，超大规模知识图谱特征等多维度信息源，以实现精准个性化的排序。因此我们从 2018 年下半年开始，全力推进 L2 核心排序层的主模型迁移至深度学习排序模型。深度模型优势体现在如下几个方面：

1. **强大的模型拟合能力**：深度学习网络包含多个隐藏层和隐藏结点，配合上非线

性的激活函数，理论上可以拟合任何函数，因此十分适用于点评搜索这种复杂的场景。

2. **强大的特征表征和泛化能力**：深度学习模型可以处理很多传统模型无法处理的特征。例如深度网络可以直接中从海量训练样本中学习到高维稀疏 ID 的隐含信息，并通过 Embedding 的方式去表征；另外对于文本、序列特征以及图像特征，深度网络均有对应的结构或者单元去处理。
3. **自动组合和发现特征的能力**：华为提出的 DeepFM，以及 Google 提出的 DeepCrossNetwork 可以自动进行特征组合，代替大量人工组合特征的工作。

下图是我们基于 Google 提出的 Wide&Deep 模型搭建的网络结构 [2]。其中 Wide 部分输入的是 LR、GBDT 阶段常用的一些细粒度统计特征。通过较长周期统计的高频行为特征，能够提供很好的记忆能力。Deep 部分通过深层的神经网络学习 Low-Order、高纬度稀疏的 Categorical 型特征，拟合样本中的长尾部分，发现新的特征组合，提高模型的泛化能力。同时对于文本、头图等传统机器学习模型难以刻画的特征，我们可以通过 End-to-End 的方式，利用相应的子网络模型进行预处理表示，然后进行融合学习。

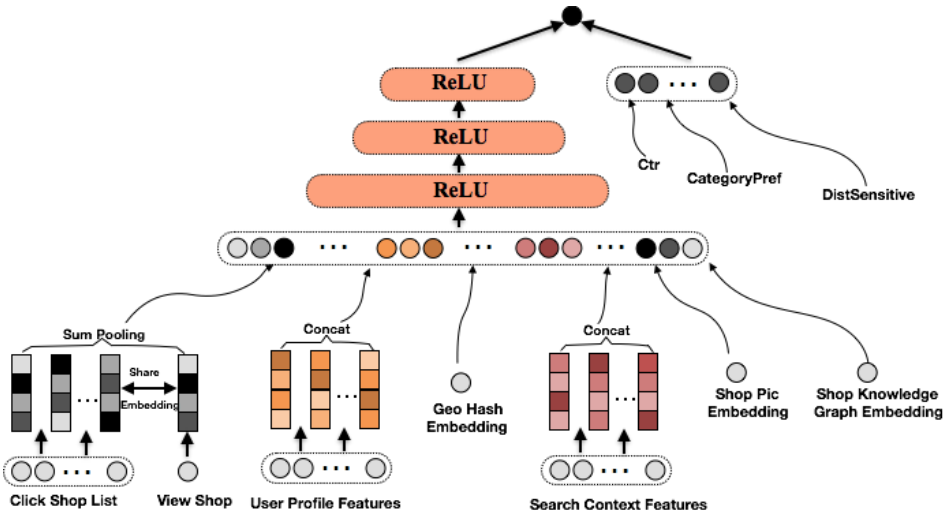


图 3　Deep&Wide 模型结构图

3. 搜索深度排序模型的特征工程实践

深度学习的横空出世，将算法工程师从很多人工挖掘和组合特征的事情中解放出来。甚至有一种论调，专做特征工程的算法工程师可能面临着失业的风险。但是深度学习的自动特征学习目前主要集中体现在 CV 领域，CV 领域的特征数据是图片的像素点——稠密的低阶特征，深度学习通过卷积层这个强力工具，可以自动对低阶特征进行组合和变换，相比之前人工定义的图像特征从效果上来说确实更加显著。在 NLP 领域因为 Transformer 的出现，在自动特征挖掘上也有了长足的进步，BERT 利用 Transformer 在多个 NLP Task 中取得了 State-of-The-Art 的效果。

但是对于 CTR 预估和排序学习的领域，目前深度学习尚未在自动特征挖掘上对人工特征工程形成碾压之势，因此人工特征工程依然很重要。当然，深度学习在特征工程上与传统模型的特征工程也存在着一些区别，我们的工作主要集中在如下几个方面。

3.1 特征预处理

- **特征归一化**：深度网络的学习几乎都是基于反向传播，而此类梯度优化的方法对于特征的尺度非常敏感。因此，需要对特征进行归一化或者标准化以促使模型更好的收敛。
- **特征离散化**：工业界一般很少直接使用连续值作为特征，而是将特征离散化后再输入到模型中。一方面因为离散化特征对于异常值具有更好的鲁棒性，其次可以为特征引入非线性的能力。并且，离散化可以更好的进行 Embedding，我们主要使用如下两种离散化方法：
  - 等频分桶：按样本频率进行等频切分，缺失值可以选择给一个默认桶值或者单独设置分桶。
  - 树模型分桶：等频离散化的方式在特征分布特别不均匀的时候效果往往不好。此时可以利用单特征结合 Label 训练树模型，以树的分叉点做为切分值，相应的叶子节点作为桶号。
- **特征组合**：基于业务场景对基础特征进行组合，形成更丰富的行为表征，为模型提供先验信息，可加速模型的收敛速度。典型示例如下：

- 用户性别与类目之间的交叉特征，能够刻画出不同性别的用户在类目上的偏好差异，比如男性用户可能会较少关注“丽人”相关的商户。
- 时间与类目之间的交叉特征，能够刻画出不同类目商户在时间上的差异，例如，酒吧在夜间会更容易被点击。

3.2 万物皆可 Embedding

深度学习最大的魅力在于其强大的特征表征能力，在点评搜索场景下，我们有海量的用户行为数据，有丰富的商户 UGC 信息以及美团大脑提供的多维度细粒度标签数据。我们利用深度学习将这些信息 Embedding 到多个向量空间中，通过 Embedding 去表征用户的个性化偏好和商户的精准画像。同时向量化的 Embedding 也便于深度模型进一步的泛化、组合以及进行相似度的计算。

3.2.1 用户行为序列的 Embedding

用户行为序列（搜索词序列、点击商户序列、筛选行为序列）包含了用户丰富的偏好信息。例如用户筛选了“距离优先”时，我们能够知道当前用户很有可能是一个即时消费的场景，并且对距离较为敏感。行为序列特征一般有如下图所示的三种接入方式：

- **Pooling**：序列 Embedding 后接入 Sum/Average Pooling 层。此方式接入成本低，但忽略了行为的时序关系。
- **RNN**：LSTM/GRU 接入，利用循环网络进行聚合。此方式能够考虑行为序列的时序关系；代价是增大了模型复杂度，影响线上预测性能。
- **Attention**：序列 Embedding 后引入 Attention 机制，表现为加权的 Sum Pooling；相比 LSTM/GRU 计算开销更低 [4]。

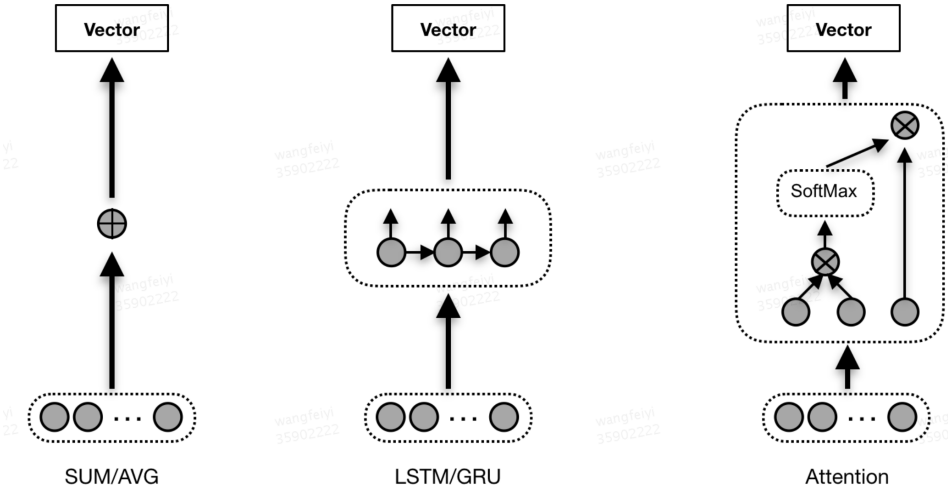


图 4　行为序列特征接入的几种方法

同时，为了突显用户长期偏好和短期偏好对于排序的不同影响，我们按照时间维度对行为序列进行了划分：Session、半小时、一天、一周等粒度，也在线上取得了收益。

3.2.2　用户 ID 的 Embedding

一种更常见的刻画用户偏好的方式，是直接将用户 ID 经过 Embedding 后作为特征接入到模型中，但是最后上线的效果却不尽如人意。通过分析用户的行为数据，我们发现相当一部分用户 ID 的行为数据较为稀疏，导致用户 ID 的 Embedding 没有充分收敛，未能充分刻画用户的偏好信息。

Airbnb 发表在 KDD 2018 上的文章为这种问题提供了一种解决思路 [9]——利用用户基础画像和行为数据对用户 ID 进行聚类。Airbnb 的主要场景是为旅游用户提供民宿短租服务，一般用户一年旅游的次数在 1–2 次之间，因此 Airbnb 的用户行为数据相比点评搜索会更为稀疏一些。

Buckets	1	2	3	4	5	6	7	8
Market	SF	NYC	LA	HK	PHL	AUS	LV	...
Language	en	es	fr	jp	ru	ko	de	...
Device Type	Mac	Msft	Andr	Ipad	Tablet	Iphone	...	
Full Profile	Yes	No						
Profile Photo	Yes	No						
Num Bookings	0	1	2-7	8+				
$ per Night	<40	40-55	56-69	70-83	84-100	101-129	130-189	190+
$ per Guest	<21	21-27	28-34	35-42	43-52	53-75	76+	
Capacity	<2	2-2.6	2.7-3	3.1-4	4.1-6	6.1+		
Num Reviews	<1	1-3.5	3.6-10	> 10				
Listing 5 Star %	0-40	41-60	61-90	90+				
Guest 5 Star %	0-40	41-60	61-90	90+				

图 5　按照用户画像和行为信息聚类

如上图所示，将用户画像特征和行为特征进行离散分桶，拼接特征名和所属桶号，得到的聚类 ID 为：US_lt1_pn3_pg3_r3_5s4_c2_b1_bd2_bt2_nu3。

我们也采取了类似 Airbnb 的方案，稀疏性的问题得到了很好的解决，并且这样做还获得了一些额外的收益。大众点评作为一个本地化的生活信息服务平台，大部分用户的行为都集中自己的常驻地，导致用户到达一个新地方时，排序个性化明显不足。通过这种聚类的方式，将异地有相同行为的用户聚集在一起，也能解决一部分跨站的个性化问题。

3.2.3　商户信息 Embedding

商户 Embedding 除了可以直接将商户 ID 加入模型中之外，美团大脑也利用深度学习技术对 UGC 进行大量挖掘，对商家的口味、特色等细粒度情感进行充分刻画，例如下图所示的“好停车”、“菜品精致”、“愿意再次光顾”等标签。

图 6　美团大脑提供的商家细粒度情感标签

这些信息与单纯的商户星级、点评数相比，刻画的角度更多，粒度也更细。我们将这些标签也进行 Embedding 并输入到模型中：

- **直连**：将标签特征做 Pooling 后直接输入模型。这种接入方式适合端到端的学习方式；但受输入层大小限制，只能取 Top 的标签，容易损失抽象实体信息。
- **分组直连**：类似于直连接入的方式，但是先对标签进行分类，如菜品 / 风格 / 口味等类别；每个分类取 Top N 的实体后进行 Pooling 生成不同维度的语义向量。与不分组的直连相比，能够保留更多抽象信息。

- **子模型接入**：可以利用 DSSM 模型，以标签作为商户输入学习商户的 Embedding 表达。此种方式能够最大化保留标签的抽象信息，但是线上实现和计算成本较高。

3.2.4 加速 Embedding 特征的收敛

在我们的深度学习排序模型中，除了 Embedding 特征，也存在大量 Query、Shop 和用户维度的强记忆特征，能够很快收敛。而 Embedding 特征是更为稀疏的弱特征，收敛速度较慢，为了加速 Embedding 特征的收敛，我们尝试了如下几种方案：

- **低频过滤**：针对出现频率较低的特征进行过滤，可以很大程度上减少参数量，避免过拟合。
- **预训练**：利用多类模型对稀疏 Embedding 特征进行预训练，然后进入模型进行微调：
  - 通过无监督模型如 Word2vec、Fasttext 对用户 – 商户点击关系建模，生成共现关系下的商户 Embedding。
  - 利用 DSSM 等监督模型对 Query– 商户点击行为建模得到 Query 和商户的 Embedding。
- **Multi-Task**：针对稀疏的 Embedding 特征，单独设置一个子损失函数，如下图所示。此时 Embedding 特征的更新依赖两个损失函数的梯度，而子损失函数脱离了对强特征的依赖，可以加快 Embedding 特征的收敛。

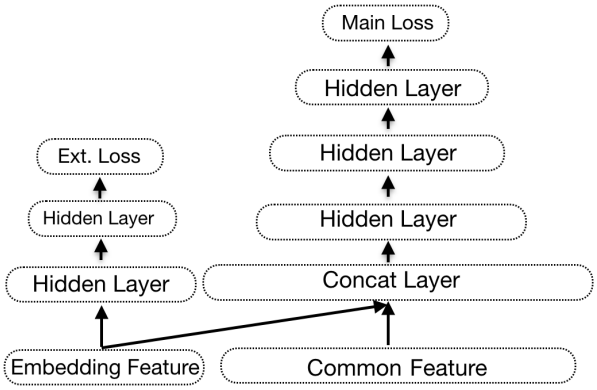


图 7　Multi-Task 加速 Embedding 特征收敛

3.3　图片特征

图片在搜索结果页中占据了很大的展示面积，图片质量的好坏会直接影响用户的体验和点击，而点评商户首图来自于商户和用户上传的图片，质量参差不齐。因此，图片特征也是排序模型中较为重要的一类。目前点评搜索主要用了以下几类图片特征：

- **基础特征**：提取图片的亮度、色度饱和度等基础信息，进行特征离散化后得到图片基础特征。
- **泛化特征**：使用 ResNet50 进行图片特征提取 [3]，通过聚类得到图片的泛化特征。
- **质量特征**：使用自研的图片质量模型，提取中间层输出，作为图片质量的 Embedding 特征。
- **标签特征**：提取图片是否是食物、环境、价目表、Logo 等作为图片分类和标签特征。

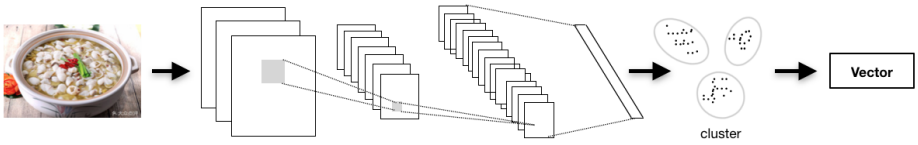


图 8　图片特征接入

4. 适用于搜索场景的深度学习 Listwise 排序算法：LambdaDNN

4.1 搜索业务指标与模型优化目标的 Gap

通常模型的预测目标与业务指标总会存在一些 Gap。如果模型的预测目标越贴近业务目标，越能保证模型优化的同时业务指标也能够有相应的提升；反之则会出现模型离线指标提升，但线上关键业务指标提升不明显，甚至出现负向的问题。工业届大部分深度学习排序采用 Pointwise 的 Log Loss 作为损失函数，与搜索业务指标有较大的 Gap。体现在如下两个方面：

1. 搜索业务常用的指标有 QV_CTR 或者 SSR(Session Success Rate)，更关心的是用户搜索的成功率（有没有发生点击行为）；而 Pointwise 的 Log Loss 更多是关注单个 Item 的点击率。
2. 搜索业务更关心排在页面头部结果的好坏，而 Pointwise 的方法则对于所有位置的样本一视同仁。

图 9　Pointwise 和 Listwise 优化目标的区别

基于上述理由，我们对于深度学习模型的损失函数进行了优化。

4.2 优化目标改进：从 Log Loss 到 NDCG

为了让排序模型的优化目标尽量贴近搜索业务指标，需要按照 Query 计算损失，且不同位置的样本具有不同的权重。搜索系统常用的指标 NDCG(Normalized Discounted Cumulative Gain) 相较于 Log Loss 显然更贴近搜索业务的要求，NDCG 计算公式如下：

$$NDCG@k(l)=\frac{1}{Z_k}\sum_{j=1}^{k}G(l_j)\eta(j)$$

累加部分为 DCG(Discounted Cumulative Gain) 表示按照位置折损的收益，对于 Query 下的结果列表 l，函数 G 表示对应 Doc 的相关度分值，通常取指数函数，即 G(lj)=2lj−1（lj 表示的是相关度水平，如 {0，1，2}）；函数 η 即位置折损，一般采用 η(j)=1/log(j+1)，Doc 与 Query 的相关度越高且位置越靠前则 DCG 值会越大。另外，通常我们仅关注排序列表页前 k 位的效果，Zk 表示 DCG@k 的可能最大值，以此进行归一化处理后得到的就是 NDCG@k。

问题在于 NDCG 是一个处处非平滑的函数，直接以它为目标函数进行优化是不可行的。LambdaRank 提供了一种思路：绕过目标函数本身，直接构造一个特殊的梯度，按照梯度的方向修正模型参数，最终能达到拟合 NDCG 的方法 [6]。因此，如果我们能将该梯度通过深度网络进行反向传播，则能训练一个优化 NDCG 的深度网络，该梯度我们称之为 Lambda 梯度，通过该梯度构造出的深度学习网络称之为 LambdaDNN。

要了解 Lambda 梯度需要引入 LambdaRank。LambdaRank 模型是通过 Pairwise 来构造的，通常将同 Query 下有点击样本和无点击样本构造成一个样本 Pair。模型的基本假设如下式所示，令 Pij 为同一个 Query 下 Doci 相比 Docj 更相关的概率，其中 si 和 sj 分别为 Doci 和 Docj 的模型得分：

$$P_{ij} \equiv P(U_i \rhd U_j) \equiv \frac{1}{1+e^{-\sigma(s_i-s_j)}}$$

使用交叉熵为损失函数，令 Sij 表示样本 Pair 的真实标记，当 Doci 比 Docj 更相关时（即 Doci 有被用户点击，而 Docj 没有被点击），有 Sij=1，否则为 −1；则损失函数可以表示为：

$$C = \frac{1}{2}(1-S_{ij})\sigma(s_i-s_j) + \log(1+e^{-\sigma(s_i-s_j)})$$

在构造样本 Pair 时，我们可以始终令 i 为更相关的文档，此时始终有 Sij ≡ 1，代入上式并进行求导，则损失函数的梯度为：

$$\lambda_{ij} = \frac{\partial C(s_i-s_j)}{\partial s_i} = \frac{-\sigma}{1+e^{\sigma(s_i-s_j)}}$$

到目前为止，损失函数的计算过程中并未考虑样本所在的位置信息。因此进一步对梯度进行改造，考虑 Doci 和 Docj 交换位置时的 NDCG 值变化，下式即为前述的 Lambda 梯度。可以证明，通过此种方式构造出来的梯度经过迭代更新，最终可以达到优化 NDCG 的目的。

$$\lambda_{ij} = \frac{\partial C(s_i-s_j)}{\partial s_i} = \frac{-\sigma}{1+e^{\sigma(s_i-s_j)}}|\Delta_{NDCG}|$$

Lambda 梯度的物理意义如下图所示。其中蓝色表示更相关（用户点击过）的文档，则 Lambda 梯度更倾向于位置靠上的 Doc 得到的提升更大（如红色箭头所示）。有了 Lambda 梯度的计算方法，训练中我们利用深度网络预测同 Query 下的 Doc 得分，根据用户实际点击 Doc 的情况计算 Lambda 梯度并反向传播回深度网络，则可以得到一个直接预测 NDCG 的深度网络。

图 10　Lambda 梯度的物理意义

4.3　LambdaDNN 的工程实施

我们利用 TensorFlow 分布式框架训练 LambdaDNN 模型。如前文所述，Lambda 梯度需要对同 Query 下的样本进行计算，但是正常情况下所有的样本是随机 Shuffle 到各个 Worker 的。因此我们需要对样本进行预处理：

1. 通过 QueryId 进行 Shuffle，将同一个 Query 的样本聚合在一起，同一个 Query 的样本打包进一个 TFRecord。
2. 由于每次请求 Query 召回的 Doc 数不一样，对于可变 Size 的 Query 样本在拉取数据进行训练时需要注意，TF 会自动补齐 Mini-Batch 内每个样本大小一致，导致输入数据中存在大量无意义的默认值样本。这里我们提供两点处理方式：
   - MR 过程中对 Key 进行处理，使得多个 Query 的样本聚合在一起，然后在训练的时候进行动态切分。
   - 读取到补齐的样本，根据设定的补齐标记获取索引位，去除补齐数据。

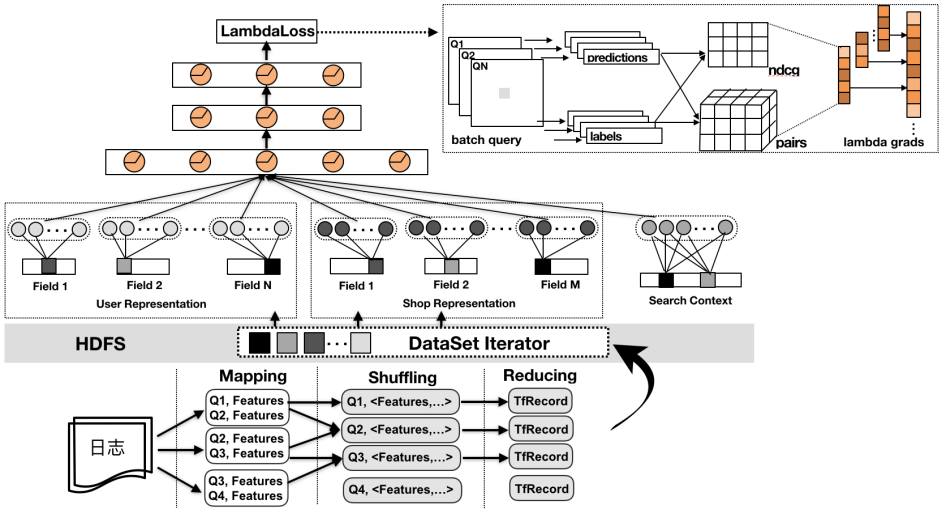


图 11　Lambda 梯度的分布式实现

为了提升训练效率，我们与基础研发平台数据平台中心紧密协同，一起探索并验证了多项优化操作：

1. 将 ID 类特征的映射等操作一并在预处理中完成，减少多轮 Training 过程中的重复计算。
2. 将样本转 TfRecord，利用 RecordDataSet 方式读取数据并计算处理，Worker 的计算性能大概提升了 10 倍。
3. Concat 多个 Categorical 特征，组合成 Multi-Hot 的 Tensor 进行一次 Embedding_Lookup 操作，减少 Map 操作的同时有助于参数做分片存储计算。
4. 稀疏 Tensor 在计算梯度以及正则化处理时保留索引值，仅对有数值的部分进行更新操作。
5. 多个 PS 服务器间进行分片存储大规模 Tensor 变量，减少 Worker 同步更新的通讯压力，减少更新阻塞，达到更平滑的梯度更新效果。

整体下来，对于 30 亿左右的样本量、上亿级别的特征维度，一轮迭代大概在半小时内完成。适当的增加并行计算的资源，可以达到分钟级的训练任务。

4.4 进一步改进优化目标

NDCG 的计算公式中，折损的权重是随着位置呈指数变化的。然而实际曝光点击率随位置变化的曲线与 NDCG 的理论折损值存在着较大的差异。

对于移动端的场景来说，用户在下拉滑动列表进行浏览时，视觉的焦点会随着滑屏、翻页而发生变动。例如用户翻到第二页时，往往会重新聚焦，因此，会发现第二页头部的曝光点击率实际上是高于第一页尾部位置的。我们尝试了两种方案去微调 NDCG 中的指数位置折损：

1. **根据实际曝光点击率拟合折损曲线**：根据实际统计到的曝光点击率数据，拟合公式替代 NDCG 中的指数折损公式，绘制的曲线如图 12 所示。
2. **计算 Position Bias 作为位置折损**：Position Bias 在业界有较多的讨论，其中 [7][8] 将用户点击商户的过程分为观察和点击两个步骤：a. 用户需要首先看到该商户，而看到商户的概率取决于所在的位置；b. 看到商户后点击商户的概率只与商户的相关性有关。步骤 a 计算的概率即为 Position Bias，这块内容可以讨论的东西很多，这里不再详述。

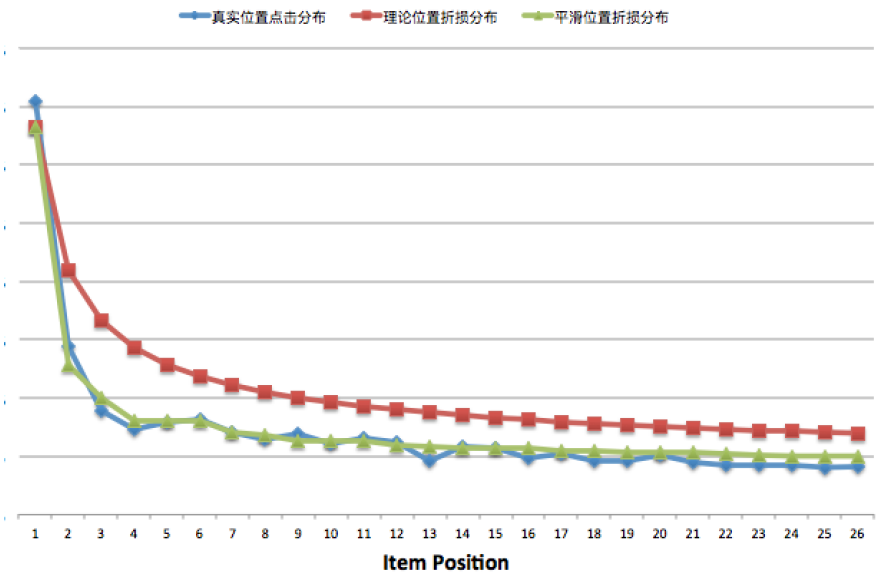


图 12　真实位置折损与理论折损的差别

经过上述对 NDCG 计算改造训练出的 LambdaDNN 模型，相较 Base 树模型和 Pointwise DNN 模型，在业务指标上有了非常显著的提升。

模型	NDCG@10
LambdaMart	0.7141
DNN	0.7378(+3.32%)
LambdaDNN	0.7485(+4.82%)

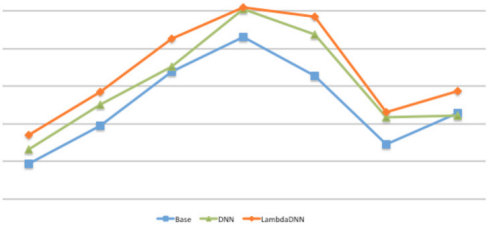


图 13　LambdaDNN 离线 NDCG 指标与线上 PvCtr 效果对比

4.5　Lambda 深度排序框架

Lambda 梯度除了与 DNN 网络相结合外，事实上可以与绝大部分常见的网络结构相结合。为了进一步学习到更多交叉特征，我们在 LambdaDNN 的基础上分别尝试了 LambdaDeepFM 和 LambdaDCN 网络；其中 DCN 网络是一种加入 Cross 的并行网络结构，交叉的网络每一层的输出特征与第一层的原始输入特征进行显性的两两交叉，相当于每一层学习特征交叉的映射去拟合层之间的残差。

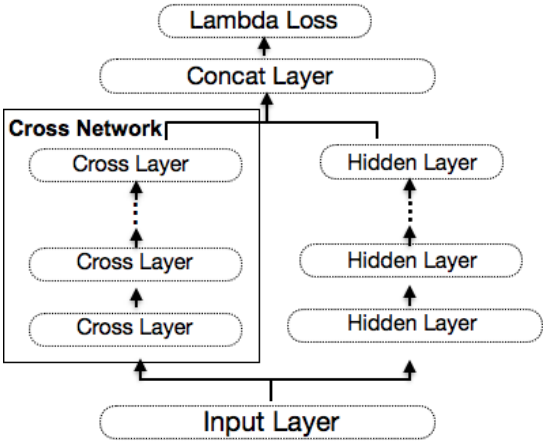


图 14　DCN 模型结构

离线的对比实验表明，Lambda 梯度与 DCN 网络结合之后充分发挥了 DCN 网络的特点，简洁的多项式交叉设计有效地提升模型的训练效果。NDCG 指标对比效

果如下图所示：

模型	NDCG@10
DNN	0.7378
DCN	**0.7410(+0.43%)**
LambdaDNN	**0.7485(+1.45%)**
LambdaDCN	**0.7496(+1.61%)**

图 15　Lambda Loss 与 DCN 网络结果的效果

5. 深度学习排序诊断系统

深度学习排序模型虽然给业务指标带来了大幅度的提升，但由于深度学习模型的“黑盒属性”导致了巨大的解释性成本，也给搜索业务带来了一些问题：

1. **日常搜索 Bad Case 无法快速响应**：搜索业务日常需要应对大量来自于用户、业务和老板们的“灵魂拷问”，“为何这个排序是这样的”，“为什么这家商户质量跟我差不多，但是会排在我的前面”。刚切换到深度学习排序模型的时候，我们对于这样的问题显得手足无措，需要花费大量的时间去定位问题。
2. **无法从 Bad Case 中学习总结规律持续优化**：如果不明白为什么排序模型会得出一个很坏的排序结果，自然也无法定位模型到底出了什么问题，也就无法根据 Bad Case 总结规律，从而确定模型和特征将来的优化方向。
3. **模型和特征是否充分学习无从得知**：新挖掘一些特征之后，通常我们会根据离线评测指标是否有提升决定特征是否上线。但是，即使一个有提升的特征，我们也无法知道这个特征是否性能足够好。例如，模型拟合的距离特征，会不会在特定的距离段出现距离越远反而打分越高的情况。

这些问题都会潜在带来一些用户无法理解的排序结果。我们需要对深度排序模型清晰地诊断并解释。

关于机器学习模型的可解释性研究，业界已经有了一些探索。Lime(Local

Interpretable Model-Agnostic Explanations) 是其中的一种，如下图所示：通过对单个样本的特征生成扰动产生近邻样本，观察模型的预测行为。根据这些扰动的数据点距离原始数据的距离分配权重，基于它们学习得到一个可解释的模型和预测结果[5]。举个例子，如果需要解释一个情感分类模型是如何预测“我讨厌这部电影”为负面情感的，我们通过丢掉部分词或者乱序构造一些样本预测情感，最终会发现，决定“我讨厌这部电影”为负面情感的是因为“讨厌”这个词。

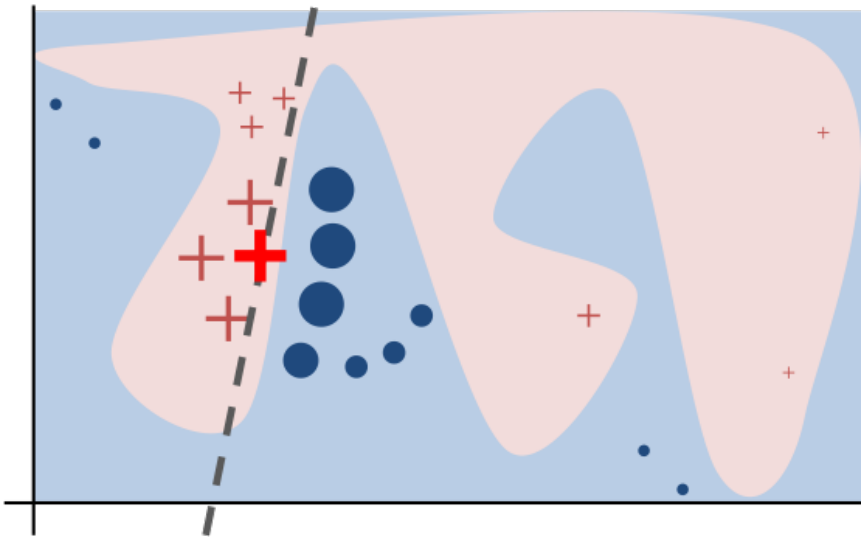


图 16　Lime 解释器的工作原理

基于 Lime 解释器的思想，我们开发了一套深度模型解释器工具——雅典娜系统。目前雅典娜系统支持两种工作模式，Pairwise 和 Listwise 模式：

1. Pairwise 模式用来解释同一个列表中两个结果之间的相对排序。通过对样本的特征进行重新赋值或者替换等操作，观察样本打分和排序位次的变化趋势，诊断出当前样本排序是否符合预期。如下图所示，通过右侧的特征位次面板可以快速诊断出为什么“南京大牌档”的排序比“金时代顺风港湾”要更靠前。第一行的特征位次信息显示，若将“金时代顺风港湾”的 1.3km 的距离特征用“南京大牌档”的 0.2km 的距离特征进行替换，排序位次将上升 10 位；由此得出，“南京大牌档”排在前面的决定性因素是因为距离近。

2. Listwise 模式与 Lime 的工作模式基本类似，通过整个列表的样本生成扰动样本，训练线性分类器模型输出特征重要度，从而达到对模型进行解释的目的。

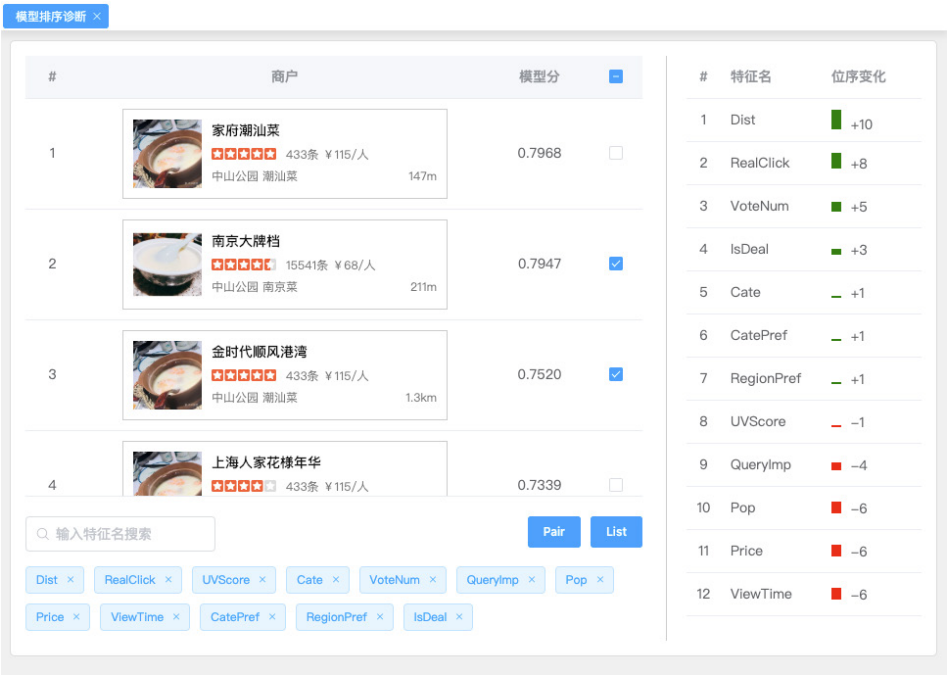


图 17　深度学习排序诊断系统：雅典娜

6. 总结与展望

2018 年下半年，点评搜索完成了从树模型到大规模深度学习排序模型的全面升级。团队在深度学习特征工程、模型结构、优化目标以及工程实践上都进行了一些探索，在核心指标上取得了较为显著的收益。当然，未来依然有不少可以探索的点。

在特征层面，大量知识图谱提供的标签信息尚未充分挖掘。从使用方式上看，简单以文本标签的形式接入，损失了知识图谱的结构信息，因此，Graph Embedding 也是未来需要尝试的方向。同时团队也会利用 BERT 在 Query 和商户文本的深层语义表达上做一些工作。

模型结构层面，目前线上依然以全连接的 DNN 网络结构为主，但 DNN 网络结构在低秩数据的学习上不如 DeepFM 和 DCN。目前 LambdaDeepFM 和

LambdaDCN 在离线上已经取得了收益，未来会在网络结构上做进一步优化。

在模型优化目标上，Lambda Loss 计算损失的时候，只会考虑 Query 内部有点击和无点击的样本对，大量无点击的 Query 被丢弃，同时，同一个用户短时间内在不同 Query 下的行为也包含着一些信息可以利用。因此，目前团队正在探索综合考虑 Log Loss 和 Lambda Loss 的模型，通过 Multi-Task 和按照不同维度 Shuffle 样本让模型充分学习，目前我们已经在线下取得了一些收益。

最后，近期 Google 开源的 TF Ranking 提出的 Groupwise 模型也对我们有一些启发。目前绝大部分的 Listwise 方法只是体现在模型训练阶段，在打分预测阶段依然是 Pointwise 的，即只会考虑当前商户相关的特征，而不会考虑列表上下文的结果，未来我们也会在这个方向上进行一些探索。

参考资料

1. 美团大脑：知识图谱的建模方法及其应用
2. Wide & Deep Learning for Recommender Systems
3. Deep Residual Learning for Image Recognition
4. Attention Is All You Need
5. Local Interpretable Model-Agnostic Explanations: LIME
6. From RankNet to LambdaRank to LambdaMART: An Overview
7. A Novel Algorithm for Unbiased Learning to Rank
8. Unbiased Learning-to-Rank with Biased Feedback
9. Real-time Personalization using Embeddings for Search Ranking at Airbnb

作者简介

非易，2016 年加入美团点评，高级算法工程师，目前主要负责点评搜索核心排序层的研发工作。

祝升，2016 年加入美团点评，高级算法工程师，目前负责点评搜索核心排序层的研发工作。

汤彪，2013 年加入美团点评，高级算法专家，点评平台搜索技术负责人，致力于深层次查询理解和大规模深度学习排序的技术落地。

张弓，2012 年加入美团点评，美团点评研究员。目前主要负责点评搜索业务演进，及集团搜索公共服务平台建设。

仲远，博士，美团 AI 平台部 NLP 中心负责人，点评搜索智能中心负责人。在国际顶级学术会议发表论文 30 余篇，获得 ICDE 2015 最佳论文奖，并是 ACL 2016 Tutorial“Understanding Short Texts”主讲人，出版学术专著 3 部，获得美国专利 5 项。此前，博士曾担任微软亚洲研究院主管研究员，以及美国 Facebook 公司 Research Scientist。曾负责微软研究院知识图谱、对话机器人项目和 Facebook 产品级 NLP Service。

大众点评信息流基于文本生成的创意优化实践

忆纯　杨肖　明海　众一　扬威　凤阳

1. 引言

信息流是目前大众点评除搜索之外的第二大用户获取信息的入口，以优质内容来辅助用户消费决策并引导发现品质生活。整个大众点评信息流（下文简称点评信息流）围绕个性化推荐去连接用户和信息，把更好的内容推荐给需要的用户。信息流推荐系统涉及内容挖掘、召回、精排、重排、创意等多层机制和排序。本文主要围绕创意部分的工作展开，并选取其中重要的文本创意优化做介绍，分为三个部分：第一部分阐述几个重点问题，包括创意优化是什么，为什么做，以及挑战在哪里；第二部分讲述领域内的应用及技术进展；第三部分介绍我们创意优化的实践，最后做个总结。

什么是创意优化

创意是一个宽泛的概念，它作为一种信息载体对受众展现，可以是文本、图像、视频等任何单一或多类间的组合，如新闻的标题就是经典的创意载体。而创意优化，作为一种方法，指在原有基础上进一步挖掘和激活资源组合方式进而提升资源的价值。在互联网领域产品中，往往表现为通过优化创意载体来提升技术指标、业务目标的过程，在信息流中落地重点包括三个方向：

1. **文本创意**：在文本方面，既包括了面向内容的摘要标题、排版改写等，也包括面向商户的推荐文案及内容化聚合页。它们都广泛地应用了文本表示和文本生成等技术，也是本文的主要方向。
2. **图像创意**：图像方面涉及到首图或首帧的优选、图像的动态裁剪，以及图像的二次生成等。
3. **其他创意**：包括多类展示理由（如社交关系等）、元素创意在内的额外补充信息。

核心目标与推荐问题相似，提升包括点击率、转化率在内的通用指标，同时需要

兼顾考量产品的阅读体验包括内容的导向性等。关于“阅读体验”的部分，这里不作展开。

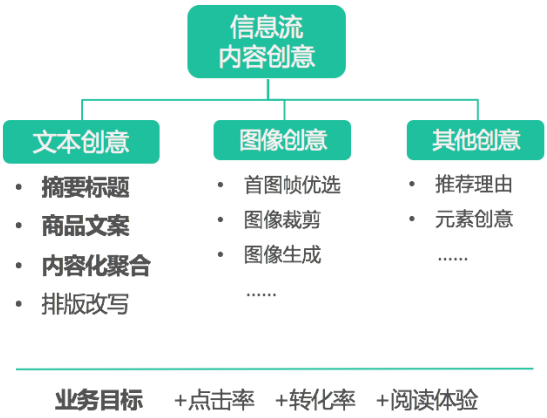


图 1　创意优化的整体应用

为什么要做文本生成

首先文本创意本身为重要的业务发展赋能。在互联网下半场，大众点评平台（下称点评平台）通过内容化去提升用户停留时长，各类分发内容类型在不停地增加，通过优化创意来提升内容的受众价值是必由之路。其次，目前很多内容类型还主要依赖运营维护，运营内容天然存在覆盖少、成本高的问题，无法完全承接需要内容化改造的场景。最后，近几年深度学习在 NLP（Natural Language Processing，自然语言处理）的不同子领域均取得了重大突破。更重要的是，点评平台历经多年，积淀了大量可用的内容数据。从技术层面来说，我们也有能力提供系统化的文本创意生成的解决方案。

对此，我们从文本创意面向对象的角度定义了两类应用形态，分别是面向内容的摘要标题，以及面向商户的推荐文案与内容化聚合页。前者主要应用信息流各主要内容场景，后者则主要应用在信息流广告等内容化场景。这里提前做下产品的简单介绍，帮助大家建立一个立体化的感知。

- **摘要标题**：顾名思义，就是针对某条分发内容生成摘要作标题展示。点评内容

源非常多样，但超过 95% 内容并没有原生标题，同时原生标题质量和多样性等差异也极大。

- **商户文案**：生成有关单个商户核心卖点的描述，一般形式为一句话的短文案。
- **内容聚合**：生成完整的内容页包括标题及多条文案的短篇推荐理由，不同于单商户文案的是，既需要考虑商户的相关性，又要保证理由的多样性。

图 2　文本创意的应用场景

最后需要明确的是，我们做文本创意优化最大的初心，是希望通过创意这个载体显式地连接用户、商户和内容。我们能够知道用户关注什么，知道哪些内容说什么，如何引导用户看，知道哪些商户好、好在哪里，将信息的推荐更进一步。而非为了生成而生成。

面临的挑战

文本创意优化，在业务和技术上分别面临着不同的挑战。首先业务侧，启动创意优化需要两个基础前提：

- 第一，衔接好创意优化与业务目标，因为并不是所有的创意都能优化，也不是所有创意优化都能带来预期的业务价值，方向不对则易蹚坑。
- 第二，创意优化转化为最优化问题，有一定的 Gap。其不同于很多分类排序问

题，本身相对主观，所谓“一千个人眼中有一千个哈姆雷特”，创意优化能不能达到预期的业务目标，这个转化非常关键。

其次，在技术层面，业界不同的应用都面临不一样的挑战，并且尝试和实践对应的解决方案。对文本创意生成来说，我们面临的最大的挑战包括以下三点：

- **带受限的生成** 生成一段流畅的文本并非难事，关键在于根据不同的场景和目标能控制它说什么、怎么说。这是目前挑战相对较大的一类问题，在我们的应用场景中都面临这个挑战。
- **业务导向** 生成能够提升业务指标、贴合业务目标的内容。为此，对内容源、内容表示与建模上提出了更高的要求。
- **高效稳定** 这里有两层含义，第一层是高效，即模型训练预测的效果和效率；第二层是稳定，线上系统应用，需要具备很高的准确率和一套完善的质量提升方案。

2. 文本生成问题综述

我们整体的技术方案演进，可以视作近两年 NLP 领域在深度学习推动下发展的一个缩影。所以在展开之前，先谈一谈整个领域的应用及技术进展。

2.1 相关领域应用

在学界相关领域，文本生成被称为 NLG，其相关任务目标是根据输入数据生成自然语言的文本。而我们在 NLP 领域使用更多的一般是 NLU（Nature Language Understanding 自然语言理解）类任务，如文本分类、命名实体识别等，NLU 的目标则是将自然语言文本转化成结构化数据。NLU 和 NLG 两者表向上是一对相反的过程，但其实是紧密相连的，甚至目前很多 NLU 的任务都受到了生成式模型中表示方法的启发，它们更多只在最终任务上有所区别。

文本生成也是一个较宽泛的概念，如下图所示，广义上只要输出是自然语言文本的各类任务都属于这个范畴。但从不同的输入端可以划分出多种领域应用，从应用相对成熟的连接人和语言的 NMT（神经机器翻译），到 2019 年初，能续写短篇故事的 GPT2 都属于 Text2Text 任务。给定结构化数据比如某些信息事件，来生成

文本比如赛事新闻的属于 Data2Text 类任务，我们的商户文案也属此类。另外还有 Image2Text 等，这块也逐渐在出现一些具有一定可用性又让人眼前一亮的应用，比如各种形式的看图说话。

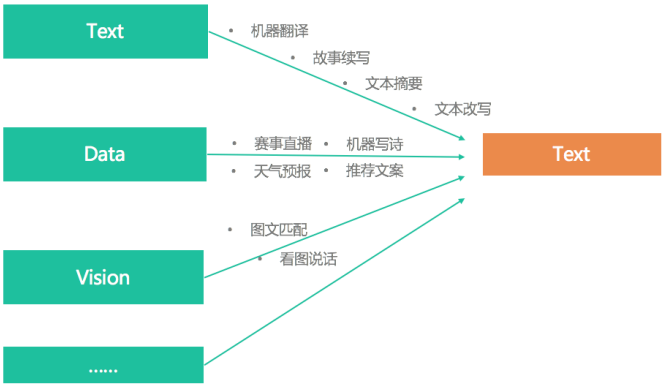


图 3　从输入端划分文本生成的领域应用

2.2　相关技术与进展

文本生成包含文本表示和文本生成两个关键的部分，它们既可以独立建模，也可以通过框架完成端到端的训练。

文本生成

文本生成要解决的一个关键问题，是根据给定的信息如何生成一段文本句子。这是一个简单输入复杂输出的任务，问题的复杂度太大，至今在准确和泛化上都没有兼顾的非常好的方法。2014 年提出的 Seq2Seq Model，是解决这类问题一个非常通用的思路，本质是将输入句子或其中的词 Token 做 Embedding 后，输入循环神经网络中作为源句的表示，这一部分称为 Encoder；另一部分生成端在每一个位置同样通过循环神经网络，循环输出对应的 Token，这一部分称为 Decoder。通过两个循环神经网络连接 Encoder 和 Decoder，可以将两个平行表示连接起来。

另外一个非常重要的，就是 Attention 机制，其本质思想是获取两端的某种权重关系，即在 Decoder 端生成的词和 Encoder 端的某些信息更相关。它也同样可以处理多模态的问题，比如 Image2Text 任务，通过 CNN 等将图片做一个关键特征的向

量表示，将这个表示输出到类似的 Decoder 中去解码输出文本，视频语音等也使用同样的方式（如下图所示）。

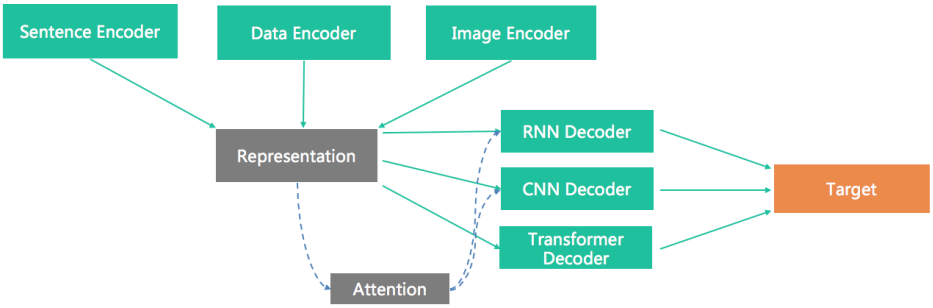


图 4　Seq2Seq 结构图示

可见 Encoder-Decoder 是一个非常通用的框架，它同样深入应用到了文本生成的三种主流方法，分别是规划式、抽取式和生成式，下面看下这几类方法各自的优劣势：

- **规划式**：根据结构化的信息，通过语法规则、树形规则等方式规划生成进文本中，可以抽象为三个阶段。宏观规划解决“说什么内容”，微观规划解决“怎么说”，包括语法句子粒度的规划，以及最后的表层优化对结果进行微调。其优势是控制力极强、准确率较高，特别适合新闻播报等模版化场景。而劣势是很难做到端到端的优化，损失信息上限也不高。
- **抽取式**：顾名思义，在原文信息中抽取一部分作为输出。可以通过编码端的表征在解码端转化为多种不同的分类任务，来实现端到端的优化。其优势在于：能降低复杂度，较好控制与原文的相关性。而劣势在于：容易受原文的束缚，泛化能力不强。
- **生成式**：通过编码端的表征，在解码端完成序列生成的任务，可以实现完全的端到端优化，可以完成多模态的任务。其在泛化能力上具有压倒性优势，但劣势是控制难度极大，建模复杂度也很高。

目前的主流的评估方法主要基于数据和人工评测。基于数据可以从不同角度衡量和训练目标文本的相近程度，如基于 N-Gram 匹配的 BLUE 和 ROUGE 等，基于字符编辑距离（Edit Distance）等，以及基于内容 Coverage 率的 Jarcard 距离等。基于数据的评测，在机器翻译等有明确标注的场景下具有很大的意义，这也是机器翻译领域最先有所突破的重要原因。但对于我们创意优化的场景来说，意义并不大，我们更重要的是优化业务目标，多以线上的实际效果为导向，并辅以人工评测。

另外，值得一提的是，近两年也逐渐涌现了很多利用 GAN（Generative Adversarial Networks，生成对抗网络）的相关方法，来解决文本生成泛化性多样性的问题。有不少思路非常有趣，也值得尝试，只是 GAN 对于 NLP 的文本生成这类离散输出任务在效果评测指标层面，与传统的 Seq2Seq 模型还存在一定的差距，可视为一类具有潜力的技术方向。

文本表示

前文提到，在 Encoder 端包括有些模型在 Decoder 端都需要对句子进行建模，那如何设计一个比较好的模型做表示，既可以让终端任务完成分类、序列生成，也可以做语义推理、相似度匹配等等，就是非常重要的一个部分。那在表示方面，整个 2018 年有两方面非常重要的工作进展：

- **Contextual Embedding**：该方向包括一系列工作，如最佳论文 Elmo(Embeddings from Language Models)，OpenAI 的 GPT(Generative Pre-Training)，以及谷歌大力出奇迹的 BERT(Bidirectional Encoder Representations from Transformers)。解决的核心问题，是如何利用大量的没标注的文本数据学到一个预训练的模型，并通过通过这个模型辅助在不同的有标注任务上更好地完成目标。传统 NLP 任务深度模型，往往并不能通过持续增加深度来获取效果的提升，但是在表示层面增加深度，却往往可以对句子做更好的表征，它的核心思想是利用 Embedding 来表征上下文的的信息。但是这个想法可以通过很多种方式来实现，比如 ELMo，通过双向的 LSTM 拼接后，可以同时得到含上下文信息的 Embedding。而 Transformer 则在

Encoder 和 Decoder 两端，都将 Attention 机制都应用到了极致，通过序列间全位置的直连，可以高效叠加多层（12 层），来完成句子的表征。这类方法可以将不同的终端任务做一个统一的表示，大大简化了建模抽象的复杂度。我们的表示也经历了从 RNN 到拥抱 Attention 的过程。

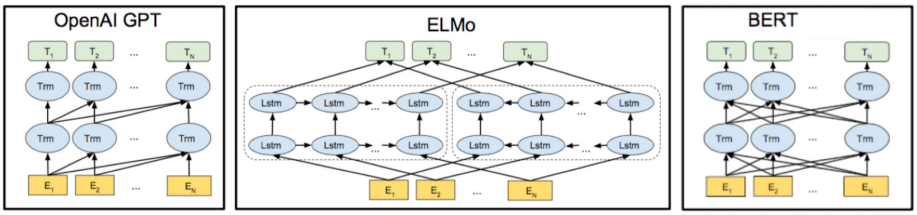


图 5　GPT ELMo BERT 模型结构

- **Tree-Based Embedding**：另外一个流派则是通过树形结构进行建模，包括很多方式如传统的语法树，在语法结构上做 Tree Base 的 RNN，用根结点的 Embedding 即可作为上下文的表征。Tree 本身可以通过构造的方式，也可以通过学习的方式（比如强化学习）来进行构建。最终 Task 效果，既和树的结构（包括深度）有关，也受“表示”学习的能力影响，调优难度比较大。在我们的场景中，人工评测效果并不是很好，仍有很大继续探索的空间。

3. 探索与实践

该部分介绍从 2017 年底至今，我们基于文本生成来进行文本创意优化的一些探索和实践。

3.1　内容源

启动文本生成，首先要了解内容本身，数据的数量和质量对我们的任务重要性无须赘述，这是一切模型的基础。目前我们使用到的数据和大致方法包括：

- **平台渠道**：用户评价、用户笔记、Push、攻略、视频内容、榜单、团单等等。
- **第三方渠道**：合作获取了很多第三方平台的内容来补缺，同时运营侧辅助创意撰写和标注了大量内容，他们同样贡献了可观的数据量。

- **标注数据**：最稀缺的永远是标注数据，尤其是符合业务目标的标注。为此，我们在冷启动阶段设计了 E&E（Explore and Exploit，探索与利用）机制，有意识地积累线上标注，同时尽量引入更多第三方的标注源。

但这些内容的不同特点，也带来了不同的挑战：

- **内容多样**：前面提到的这些内容的结构化程度各不相同，长短差异也极大，对内容表示提出了很高的要求。
- **质量不一**：源内容非常丰富，但事实上质量、质感远远没有达到理想的标准。尤其是占绝对大头的 UGC 的内容，不做好两端的质控将极大影响业务目标的优化，甚至会造成体验问题。
- **聚焦商户**：平台 99% 以上的内容，都以商户作为核心载体，这个对商户的理解和表示同样提出了很高的要求，尤其是在内容化升级的场景下。
- **场景差异**：不同的场景、不同的应用，对模型能力的侧重和优化目标不一样。比如内容和商户，前者要求要有很高的准确率，同时保证优化线上效果；后者更多的是要求有较强的泛化性，并对质感进行优化。

图 7　双平台内容特点与挑战

3.2　基础能力模块

所以，文本创意优化要在业务侧落地产生效果，还需应用到 NLP 领域诸多方向的技术。下图是抽象的整个文本生成应用的基础能力模块，包括用于源和端质量控制的文本质量层，构建 Context 表示的文本表示层，以及面向业务优化的端到端模型

层，其中很多技术应用了公司其他兄弟团队包括内容挖掘组、NLP 中心、离线计算组的出色成果。如针对负面内容过滤的情感分析，多项针对性的文本分类，针对商户表示的标签挖掘等，在这里特别向他们表示感谢。

图 8　文本生成应用的基础能力模块

3.3　信息流标题实践

双平台的内容需要在信息流分发，在创意上最先优化的就是标题，这是用户仅能看到两个要素之一（另一个为首图），而我们超过 95% 的内容并没有原生标题，同时原生标题也存在诸如多样性差非场景导向等问题，还有二次优化的空间。

但是，有两点比较大的挑战，在不同任务上具象可能不一样。它们的本质并没有改变，部分也是业界难点：

- **1. 两个受限条件**：第一，需要以线上点击率转化率为优化目标，线上没效果，写的再好意义都不大；第二，需要与原文强相关，并且容错空间极小，一出现就是 Case。
- **2. 优化评估困难**：第一，模型目标和业务目标间存在天然 Gap；第二，标注数据极度稀缺，离线训练和线上实际预测样本数量之间，往往差距百倍。

对此，我们通过抽取式和生成式的相结合互补的方式，并在流程和模型结构上着手进行解决。

抽取式标题

抽取式方法在用户内容上有比较明显的优势：首先控制力极强，对源内容相关性好，改变用户行文较少，也不容易造成体验问题，可以直接在句子级别做端到端优化。对此，我们把整个标题建模转变为一个中短文本分类的问题，但也无法规避上文提到两个大挑战，具体表现在：

- 在优化评估上，首先标题创意衡量的主观性很强，线上 Feeds 的标注数据也易受到其他因素的影响，比如推荐排序本身；其次，训练预测数据量差异造成 OOV 问题非常突出，分类任务叠加噪音效果提升非常困难。对此，我们重点在语义 + 词级的方向上来对点击 / 转化率做建模，同时辅以线上 E&E 选优的机制来持续获取标注对，并提升在线自动纠错的能力。
- 在受限上，抽取式虽然能直接在 Seq 级别对业务目标做优化，但有时候也须兼顾阅读体验，否则会形成一些“标题党”，亦或造成与原文相关性差的问题。对此，我们抽象了预处理和质量模型，来通用化处理文本创意内容的质控，独立了一个召回模块负责体验保障。并在模型结构上来对原文做独立表示，后又引入了 Topic Feature Context 来做针对性控制。

整个抽取式的流程，可以抽象为四个环节 + 一个在线机制：

图 9　抽取式生成标题流程

1. 源数据在内容中台完成可分发分析后，针对具体内容，进行系统化插件式的预处理，包括分句拼句、繁简转换、大小写归一等，并进行依存分析。
2. 而后将所有可选内容作质量评估，包括情感过滤、敏感过滤等通用过滤，以

及规则判别等涉及表情、冗余字符处理与语法改写的二次基础优化。

3. 在召回模块中，通过实体识别 +TF-IDF 打分等方式来评估候选内容标题基础信息质量，并通过阈值召回来保证基础阅读体验，从而避免一些极端的 Bad Case。
4. 最后，针对候选标题直接做句子级别的点击 / 转化率预估，负责质感、相关性及最终的业务目标的优化。为此，我们先后尝试了诸多模型结构来解决不同问题，下面重点在这方面做下介绍。

我们第一版 Bi-LSTM + Attention 整个结构并不复杂。我们的输入层是 PreTrain 的 Word Embedding，经过双向 LSTM 给到 Attention 层，Dropout 后全连接，套一个交叉熵的 Sigmod，输出判别，但它的意义非常明显，既可以对整句序列做双向语义的建模，同时可以通过注意力矩阵来对词级进行加权。这个在线上来看，无论是对体感还是点击转化率都较召回打分的原始版本，有了巨大提升。而后，我们还在这个 Base 模型基础上，尝试添加过 ELMo 的 Loss，在模型的第一层双向 LSTM 进行基于 ELMo Loss 的 Pre Train 作为初始化结果，在线上指标也有小幅的提升。

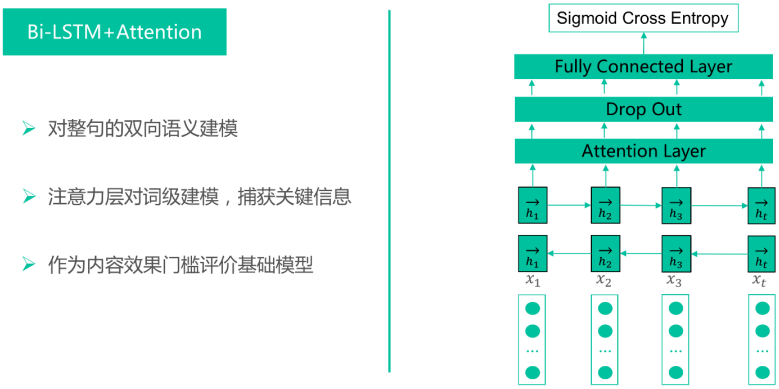


图 10　Bi-LSTM + Attention

但是上述这个结构，将中短文本脱离原文独立建模，显然无法更好地兼顾原文受限这个条件。一个表现，就是容易出现“标题党”、原文不相关等对体验造成影响

的问题。对此，我们在原文与候选标题结合的表示建模方面，做了不少探索，其中以CNN+Bi-LSTM+Attention 的基模型为代表，但其在相关性建模受原文本身长度的影响较大，而且训练效率也不理想。

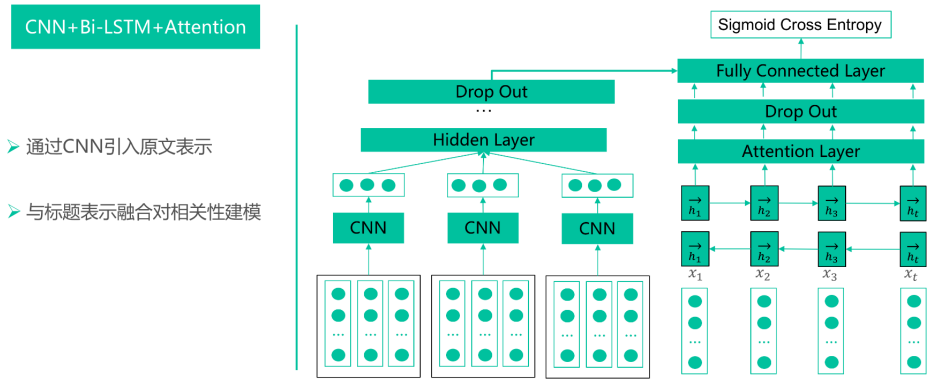


图 11　CNN+Bi-LSTM + Attention

经过一段时间的探索分析，在原文受限问题上，最终既通过深度模型来表征深层的语义，也辅以更多的特征工程，如属性、Topic 等挖掘特征我们统称为 Context，来表征用户能感知到的浅层信息，“两条腿走路”才能被更好的学习，这个在文案生成和标题生成的探索中反过来为抽取式提供了借鉴。

在效率上，我们整体替换了 RNN-LSTM 的循环结构，采用了谷歌那时新提出的自注意力的机制，来解决原文表征训练效率和长依赖问题。采用这个结构在效果和效率上又有了较大的提升。主要问题是，我们的 Context 信息如何更好地建模到 Self-Attention 的结构中。它与生成式模型结构非常类似，在下文生成式部分有所介绍。

另外，需要说明的一点是，除非有两个点以上的巨大提升，一般我们并不会以离线评测指标来评价模型好坏。因为前面提到，我们的标注数据存在不同程度的扰动，而且只是线上预测很小的一个子集，无法避免的与线上存在一定的 Gap，所以我们更关注的是模型影响的基础体验（人工检测通过率即非 Bad Case 率），效率表现（训练预测的时效）最重要的还是线上实际的业务效果。在我们这几个版本的迭代中，这三

个方面都分别获得了不同程度的优化，尤其是包括点击率、总点击量等在内的业务指标，都累计获得了 10% 以上的提升。

	V0.1 TF-IDF	V1.0 Bi-LSTM+Attn	V1.5 CNN+Bi-LSTM+Attn	V2.0 TopicFeature+Self-Attn
业务优化（CTR/CVR/点击曝光）	**+0%**	**+12%**	**+2%**	**+4%**
基础体验（检验通过率）	*+0%*	*+10%*	*+5%*	*+8%*
效率优化（训练预测时效）	*+0%*	*+0%*	*-100%*	*+200%*

图 13　效果数据

受限生成式标题

抽取式标题在包括业务指标和基础体验上都获取了不错的效果，但仍有明显的瓶颈。第一，没有完全脱离原文，尤其在大量质量欠优内容下无法实现创意的二次优化；第二，更好的通过创意这个载体显式的连接用户、商户和内容，这个是生成式标题可以有能力实现的，也是必由之路。

生成式标题，可以抽象描述为：在给定上文并在一定受限条件下，预估下个词的概率的问题。在信息流标题场景，抽取式会面临的问题生成式全部会继承，且在受限优化上面临更大的挑战：

- 原文受限，首先只有表示并学习到原文的语义意图才能更好的控制标题生成，这个本身在 NLU 就是难点，在生成式中就更为突出；其次，标注数据稀缺，原文＋标题对的数据极少，而大部分又存在于长文章。为了保证控制和泛化性，我们初期将标题剥离原文独立建模，通过 Context 衔接，这样能引入更多的非标数据，并在逐步完成积累的情况下，才开始尝试做原文的深度语义表示。
- 优化评估，受限生成式对训练语料的数量和质量要求高很多，首先要保证基础的语义学习也要保证生成端的质量；其次，生成式本质作为语言模型无法在句子层面对业务目标直接做优化，这中间还存在一道 Gap。

在表示上，前面已经提到，我们经历过目标单独建模和结合原文建模的过程，主要原因还是在于仅针对 Target 的理解去构建 Context 衔接，非常容易出现原文相关性问题。所以我们在描述的泛化性方向也做了不少的尝试，比如尽可能地描述广而泛

主题。诸如“魔都是轻易俘获人心的聚餐胜地”，因为只面向上海的商户，内容符合聚餐主题，泛化能力很强，但仍然不能作为一个普适的方案解决问题。

下图为我们一个有初步成效的 RNN-Base 的 Seq2Seq 模型的整体结构。Encoder 端使用的是，包括前面提到的主题（包括商户信息）表示以及原文的双向语义表示，两部分的拼接构成的 Context，输出给注意力层。Decoder 端生成文本时，通过注意力机制学习主题和原文表示的权重关系，这个结构也完整应用到了文案生成，其中控制结构会在文案中展开介绍。

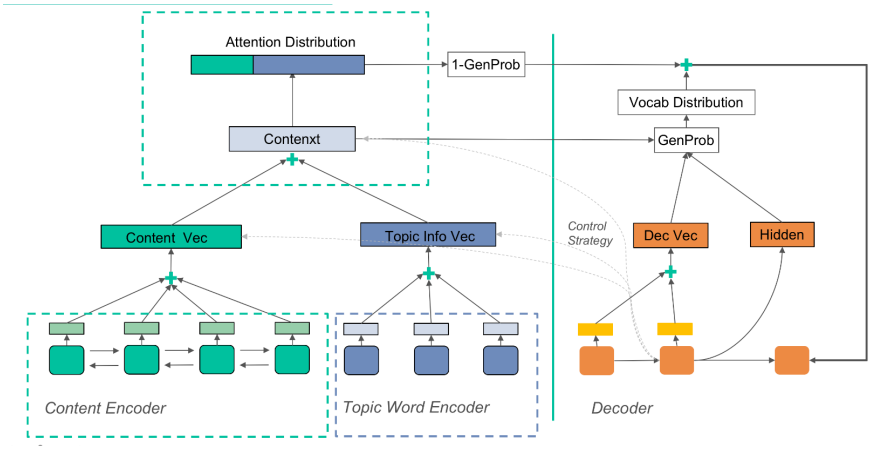


图 14　LSTM Attention Based Seq2Seq 模型结构

在序列建模上，我们经历了一个从 RNN 到自注意力的过程。简单介绍下，序列建模一个核心要点是如何建模序列间的长依赖关系。影响它的重要因素是，信号在网络正向和反向计算中传递的长度（也就是计算次数），较长的依赖关系消失越严重。而在自注意力结构中，每一层都直接与前一层的所有位置直接连接，因此依赖长度均为 O(1)，最大程度保留了序列间的依赖关系。

可以看到，Encoder 包括两部分，一部分是 Source 原文，一部分是基于原文和商户理解的主题 Context，两者共同组成。为此，我们借鉴了 NMT 的一部分研究思想，调整了 Transformer 的结构，在原结构上额外引入了 Context Encoder，并且在 Encoder 和 Decoder 端加入了 Context 的 Attention 层，来强化模型捕捉 Context 信息的能力。

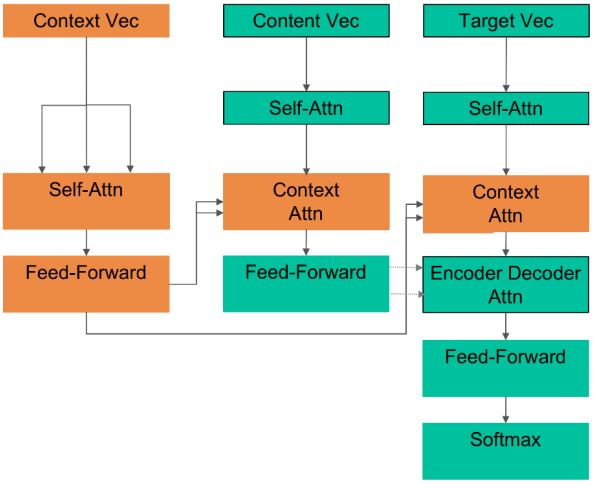


图 15　Transformer Based Seq2Seq Model

我们在生成式方向探索过程中，对低质内容的标题生成，在线上获得了接近 10% 的效果提升，但仍有很多值得进一步的尝试和深挖的空间。

抽取与生成 Combine

在我们的场景中，有两种 Combine 的思路，一个是以业务效果为导向的偏工程化方法，另外一个是我们正在探索的一种 Copy 方法。

工程化的思想非常简洁，在推荐问题上扩充候选，是提升效果的一个可行途径，那生成内容即作为新增的候选集之一，参与整体的预估排序。这个方法能保证最终线上效果不会是负向的，实际上也取得了一定的提升。

另一种方法也是学业界研究的子方向之一，即 Copy 机制，我们也在做重点探索，这里仅作思路的介绍，不再进行展开。

使用 Copy 机制的原始目的，是为了解决生成式的 OOV（超出词表范围）问题。但对于我们的场景来说，大部分的“内容 – 标题”对数据是来自于抽取式，即我们很多标题数据，其实参考了原文。那如何继承这个参考机制，针对业务目标学习何时 Copy 以及 Copy 什么，来更优雅地发挥生成式的优势，就是我们探索 Copy 方法的初衷。我们的方向是对 Copy 和 Generate 概率做独立建模，其中重点解决在受限情况下的“Where To Point”问题。

业务指标与生成式目标的 Gap

我们知道生成式模型其本质是一个 Language Model，它的训练目标是最小化 Word 级别的交叉熵 Loss，而最终我们的需要评价的其实是业务相关的句子级别点击率，这就导致了训练目标和业务指标不一致。

解决这个问题，在我们的场景中有三个可行的方向，第一是在 Context 中显式地标注抽取式模型的 Label，让模型学习到两者的差异；第二是在预测 Decoder 的 Beam Search 计算概率的同时，添加一个打分控制函数；第三则是在训练的 Decoder 中，建立一个全局损失函数参与训练，类似于 NMT 中增加的 Coverage Loss。

考虑到稳定性和实现成本，我们最终尝试了第一和第二种方式，其中第二种方式还是从商户文案迁移过来的，也会在下文进行介绍。在线上，这个尝试并没有在 Combine 的基础上取得更好的效果，但同样值得更加深入的探索。

在线 E&E 机制

最后，介绍一下前面提到过的标题 E&E（Explore and Exploit，探索与利用）机制，用来持续获取标注数据，并提升在线自动纠错的能力。我们采用了一种贪心的 Epsilon Greedy 策略，并做了一点修改，类似经典的 Epsilon 算法，区别是引入创意状态，根据状态将 Epsilon 分成多级。目的是将比较好的创意可以分配给较大概率的流量，而不是均分，差的就淘汰，以此来提升效率。在初期优化阶段，这种方式发挥了很大的作用。

具体我们根据标题和图片的历史表现和默认相比，将状态分成 7 档，从上到下效果表现依次递减，流量分配比例也依次降低，这样可以保证整个系统在样本有噪音的情况下实现线上纠偏。

创意状态	划分原则	Epsilon
Win	曝光充足，且CTR置信大于默认创意	高：70%
Prewin	曝光不充足，且CTR大于默认创意	中：20%
Default	默认创意	
Notsure	曝光不充足	低：10%
Candidate	无曝光	
Prefail	曝光不充足，且CTR低于默认创意	
Fail	曝光充足，且CTR置信小于默认创意	0%

图 17　在线 E&E 选优

3.4　商户文案实践

文案作为一个常见的创意形式，在 O2O 以商户为主要载体的场景下有三点需要：第一，赋予商户以内容调性，丰富创意；第二，通过内容化扩展投放的场景；最后，赋能平台的内容化升级，主要业务目标包括点击率、页面穿透率等等。

文案生成和标题生成能够通用整体的生成模型框架，可以归为 Data2Text 类任务，最大区别是由文案的载体”商户”所决定。不同于内容，准确性的要求低很多，复杂度也大大降低，但同时为泛化能力提出了更高的要求，也带来了与内容生成不同的问题。首先在表示上，对商户的结构化理解变得尤其关键；其次在控制上，有 D2T 任务特有且非常重要的控制要求。前文也提到了生成一段文本从来不是难点，重要的是如何按照不同要求控制 Seq 生成的同时，保证很好的泛化性。下文也会分别介绍卖点控制、风格控制、多样性控制控制等几个控制方法。实现这样的控制，也有很多不同的思路。

商户表示

商户的表示抽象为 Context，如下图中所示，主要分两部分。

第一部分来源于商户的自身理解，一部分则来源于目标文本，两部分有一定交集。其中商户理解的数据为卖点或者 Topic，在初期，为了挖掘商户卖点和 Topic，我们主要使用成本较低、无需标注的 LDA。但是它的准确性相对不可控，同时对产出的卖点主题仍需要进行人工的选择，以便作为新的标注，辅助后续扩展有监督的任

务。我们通过 Key 和 Value 两个 Field，来对卖点和主题进行共同表达（也存在很多只有 Value 的情况），比如下图这个商户“菜品”是个 Key，“雪蟹”是 Value，“约会”则仅是 Value。随着时间的推移，后续我们逐渐利用平台商户标签和图谱信息，来扩展商户卖点的覆盖，以此丰富我们的输入信息。该部分在内容挖掘和 NLP 知识图谱的相关介绍中都有涉及，这里不再进行展开。

第二部分目标文本来源，特意添加这部分进入 Context，主要有三方面原因：

- 第一，仅仅依靠商户理解的 Context，在训练过程中 Loss 下降极慢，并且最终预测生成多样性不理想。本质原因是，目标文本内容与商户卖点、主题间的相关性远远不够。通过不同商户的集合来学习到这个表示关系，非常困难。
- 第二，拓宽可用数据范围，不受商户评论这类有天然标注对的数据限制，从商户衔接扩展到卖点衔接，引入更多的泛化描述数据，比如各类运营文案等等。
- 第三，这也是更为重要的一点，能够间接地实现卖点选择的能力，这个会在下文进行介绍。

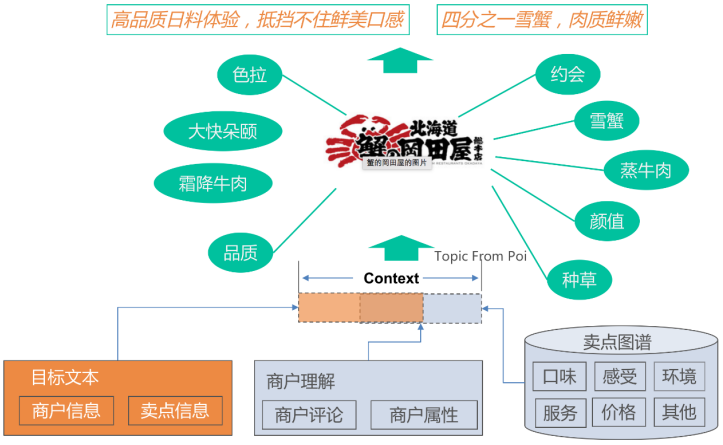


图 18　商户表示

控制端实现

控制，在解码端表现为两类，一类我们称之为 Hard Constrained（强控制），即在数据端给定（或没有给定）的信息，一定要在解码端进行（或不进行）相应描述，这

个适用于地域类目等不能出错的信息。比如这家商户在上海，生成时不能出现除上海以外的地域信息，否则容易造成歧义。另一类称之为 Soft Constrained（弱控制），不同于 NMT 问题，在文案生成上即便是完全相同的输入，不同的输出都是允许的，比如同一商户，最终的文案可以选择不同的卖点去描述不同的内容。

这类同属受限优化的问题，前文提到过有两个思路方向：第一，通过构建机制来让模型自己学习到目标；第二，在 Decoder 的 Beam Search 阶段动态地加入所需的控制目标。我们使用两者相结合的方法，来完成最终的不同控制的实现。

- **两端机制设计**：在具体机制实现上，主要依赖在 Input Context 和 Output Decoder 两端同时生效，让 Context 的 Hard Constrained 来源于 Output，从而使 Model 能够自动学习到强受限关系；而 Soft Constrained 则通过贝叶斯采样的方法，动态添加进 Context，从而帮助 Model 提升泛化能力。
- **Decoder 控制**：简单介绍下 Beam Search，前面提到过，文本生成的预测过程是按 Word 级进行的，每轮预测的候选是整个词汇空间，而往往一般的词表都是十万以上的量级。如果生成序列序列长度为 N，最终候选序列就有十万的 N 次方种可能，这在计算和存储上绝不可行。这时候，就需要使用到 Beam Search 方法，每一步保留最优的前 K（K 一般为 2）个最大概率序列，其他则被剪枝，本质上可以视作一个压缩版的维特比解码。

我们在预测 Beam Search 阶段，除了计算模型概率外，额外增加下图中绿色部分的 Fuction。输入为之前已生成的序列，具体计算逻辑取决于控制目标，可以自由实现。

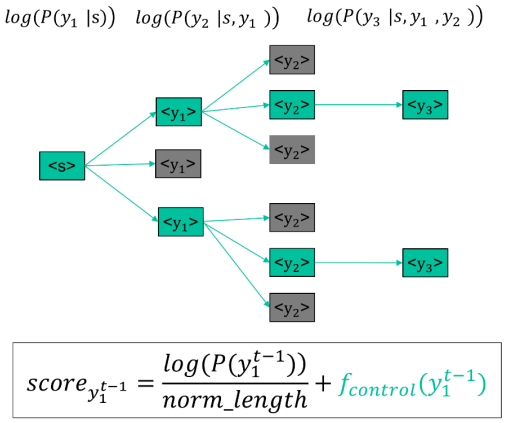


图 19　Decoder Beam_Search 控制

下面简单介绍两个重要的控制实现：

- **卖点控制**：这是最重要的一个控制机制，我们整理了涉及到 Hard Constrained 的卖点和实体，重要的如地域、品类等，在目标理解过程中直接加入 Context。对于 Soft Constrained，我们通过卖点的共现计算一个简单的条件概率，并将卖点依此条件概率随机添加进 Context 中，从而让模型通过注意力学习到受限关系。最后在 Decoder fuction 部分，我们新增了一个 Hard&Soft Constrained 的匹配打分项，参与最终的概率计算。最终的实际结果，也非常符合我们的预期。
- **风格控制**：实现方法和卖点控制非常相似，只是这里的风格，其实是通过不同内容之间的差异来间接进行实现。比如大众点评头条、PGC 类的内容与 UGC 类的的写作风格，就存在极大的差异。那么在文案上，比如聚合页标题上可能更需要 PGC 的风格，而聚合页内容上则需要 UGC 的风格。这样的内容属性，即可作为一个 Context 的控制信号，让模型捕获。

3.5　内容聚合

多样性控制

多样性，在文案生成上是一个比较重要和普遍的问题，尤其对于同一个店铺、同

一个卖点或主题同时生成 N 条内容的聚合页来说，更为突出。本质原因是，在解码预测 Beam Search 时永远选择概率最大的序列，并不考虑多样性。但是如果预测时采用 Decoder 概率 Random Search 的方法，则在通顺度上会存在比较大的问题。

对此，我们直接对全局结果进行优化，在预测时把一个聚合页 Context 放到同一个 batch 中，batch_size 即为文案条数，对已经生成序列上进行实体重复检测和 n-gram 重复检测，将检测判重的加一个惩罚性打分，这个简单的思想已经能非常好的解决多样性问题。

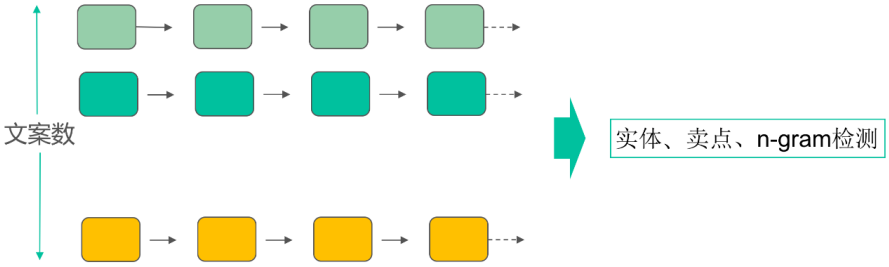


图 20　聚合页多样性控制

4. 动态创意

目前，很多搜索推荐等排序优化场景，都会将创意信息作为特征工程一部分添加进精排或召回模型。那如果把创意优化近似为一个内容级创意排序问题，也可以无缝衔接常用的 Wide&Deep、DNN、FNN 等 CTR 预估模型。但是这之前，需要明确一点非常重要的问题，即它与推荐精排模型的差异，它们之间甚至可能会相互影响，对此，提供下我们的思考。

与精排模型的差异

- 第一，精排模型能否一并完成创意的排序，答案显然是肯定的。但它的复杂度决定了能 Cover 候选集的上限，性能上往往接受不了叉乘创意带来的倍数增长。但此非问题的关键。
- 第二，创意层排序在精排层之前还是之后，直接影响了创意模型的复杂度，也

间接决定了其效果的上限，以及它对精排模型可能的影响程度，从而可能带来全局的影响。此没有最佳实践，视场景权衡。

- 第三，精排模型与创意排序业务目标一致，但实现方式不同。精排模型通过全局排序的最优化来提升业务指标，而创意优化则是通过动态提升内容受众价值来提升业务指标。

最后，我们回到用户视角，当用户在浏览信息流时，其实看到的只有创意本身（标题、图片、作者等信息），但用户却能从中感知到背后的诸多隐含信息，也就是 CTR 预估中的重要内容 / 商户类特征，诸如类目、场景、商户属性等。这个现象背后的本质在于，创意可以表征很多高阶的结构化信息。

基于这一点，在创意优化的特征工程上，方向就很明确了：强化 User/Context，弱化 Item/POI，通过创意表征，来间接学习到弱化的信息从而实现创意层面的最优排序。该部分工作不仅仅涉及到文本，在本文中不再展开。

用户兴趣与文本生成结合的可能性

动态创意为文本生成提供了全新的空间，也提出了更高的要求。动态创意提升受众价值，不仅仅只能通过排序来实现，在正篇介绍的最后部分，我们抛出一个可能性的问题，供各位同行和同学一起思考。也希望能看到更多业界的方案和实践，共同进步。

5. 总结与展望

整个 2018 年，大众点评信息流在核心指标上取得了显著的突破。创意优化作为其中的一部分，在一些方面进行了很多探索，也在效果指标上取得了较为显著的收益。不过，未来的突破，更加任重而道远。

2018 年至 2019 年初，NLP 的各个子领域涌现了非常多令人惊喜的成果，并且这些成果已经落地到业界实践上。这是一个非常好的趋势，也预示着在应用层面会有越来越多的突破。比如 2019 年初，能够续写短篇小说的 GPT2 问世，虽然它真实的泛化能力还未可知，但让我们真切看到了在内容受限下高质量内容生成的可能性。

最后，回到初心，我们希望通过创意的载体显式地连接用户、商户和内容。我们能了解用户关注什么，知道某些内容表达什么，获知哪些商户好，好在哪里，将信息的推荐更进一步。

参考资料

[1] Context-aware Natural Language Generation with Recurrent Neural Networks. arXiv preprint arXiv:1611.09900.
[2] Attention Is All You Need. arXiv preprint arXiv:1706.03762.
[3] Universal Transformers. arXiv preprint arXiv:1807.03819.
[4] A Convolutional Encoder Model for Neural Machine Translation. arXiv preprint arXiv:1611.02344.
[5] Don't Give Me the Details, Just the Summary! Topic-Aware Convolutional Neural Networks for Extreme Summarization. arXiv preprint arXiv:1808.08745.
[6] Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805.
[7] ELMO：Deep contextualized word representations. arXiv preprint arXiv:1802.05365.
[8] openAI GPT：Improving Language Understanding by Generative Pre-Training.
[9] Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473.
[10] Tensor2Tensor for Neural Machine Translation. arXiv preprint arXiv:1803.07416.
[11] A Convolutional Encoder Model for Neural Machine Translation. arXiv preprint arXiv:1611.02344.
[12] Sequence-to-Sequence Learning as Beam-Search Optimization. arXiv preprint arXiv:1606.02960.
[13] A Deep Reinforced Model For Abstractive Summarization. arXiv preprint arXiv:1705.04304.
[14] SeqGAN: Sequence Generative Adversarial Nets with Policy Gradient. arXiv preprint arXiv:1609.05473.
[15] Generating sequences with recurrent neural networks. CoRR,abs/1308.0850.

作者简介

忆纯，2015 年加入美团点评，算法专家，目前负责点评信息流内容创意工作。
杨肖，博士，2016 年加入美团点评，高级算法专家，点评推荐智能中心内容团队负责人。
明海，2016 年加入美团点评，美团点评研究员，点评推荐智能中心团队负责人。
众一，2016 年加入美团点评，算法研发工程师，目前主要负责点评信息流创意相关算法研发工作。
扬威，2018 年初加入美团点评，算法研发工程师，目前主要负责点评信息流动态创意相关算法研发工作。
凤阳，2016 年加入美团点评，算法研发工程师，目前主要负责点评信息流内容运营算法优化的工作。

AI Challenger 2018：细粒度用户评论情感分析冠军思路总结

程惠阁

2018 年 8 月 -12 月，由美团点评、创新工场、搜狗、美图联合主办的“AI Challenger 2018 全球 AI 挑战赛”历经三个多月的激烈角逐，冠军团队从来自全球 81 个国家、1000 多所大学和公司的过万支参赛团队中脱颖而出。其中“后厂村静静”团队 – 由毕业于北京大学的程惠阁（现已入职美团点评）单人组队，勇夺“细粒度用户评论情感分类”赛道的冠军。本文系程惠阁对于本次参赛的思路总结和经验分享，希望对大家能够有所帮助和启发。

背景

在 2018 全球 AI 挑战赛中，美团点评主要负责了其中两个颇具挑战的主赛道赛题：细粒度用户评论情感分析和无人驾驶视觉感知。其中 NLP 中心负责的细粒度用户评论情感分析赛道，最受欢迎，参赛队伍报名数量最多，约占整个报名团队的五分之一。

细粒度用户评论情感分析赛道提供了 6 大类、20 个细分类的中文情感评论数据，标注规模难度之大，在 NLP 语料特别是文本分类相关语料中都属于相当罕见，这份数据有着极其重要的科研学术以及工业应用价值。

赛题简介

在线评论的细粒度情感分析对于深刻理解商家和用户、挖掘用户情感等方面有至关重要的价值，并且在互联网行业有极其广泛的应用，主要用于个性化推荐、智能搜索、产品反馈、业务安全等。本次比赛我们提供了一个高质量的海量数据集，共包含 6 大类 20 个细粒度要素的情感倾向。参赛人员需根据标注的细粒度要素的情感倾向建立算法，对用户评论进行情感挖掘，组委将通过计算参赛者提交预测值和场景真实值之间的误差确定预测正确率，评估所提交的预测算法。

1. 工具介绍

在本次比赛中，采用了自己开发的一个训练框架，来统一处理 TensorFlow 和 PyTorch 的模型。在模型代码应用方面，主要基于香港科技大学开源的 RNet 和 MnemonicReader 做了相应修改。在比赛后期，还加入了一个基于 BERT 的模型，从而提升了一些集成的效果。

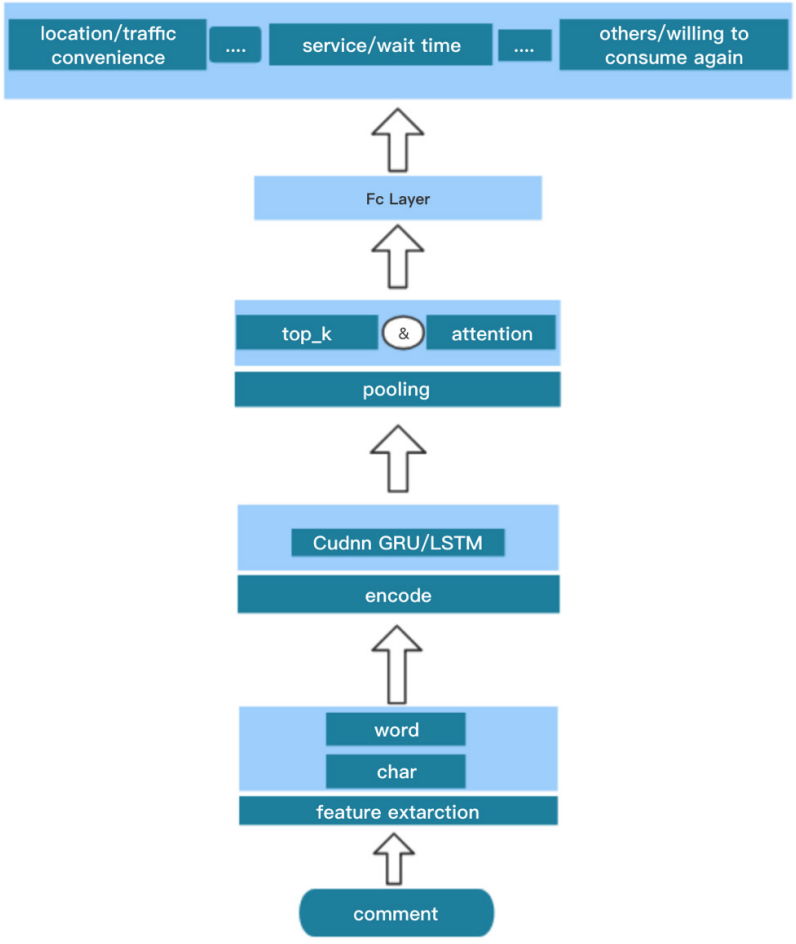
2. 整体思路

整体将该问题看作 20 个 Aspect 的情感多分类问题，采用了传统的文本分类方法，基于 LSTM 建模文本，End2End 多 Aspect 统一训练。

文本分类是业界一个较为成熟的问题，在 2018 年 2 月份，我参加了 Kaggle 的“作弊文本分类”比赛，当时的冠军团队主要依靠基于翻译的数据增强方法获得了成功。2018 年反作弊工作中的一些实践经验，让我意识到，数据是提升文本分类效果的第一关键。因此，我第一时间在网络上寻找到了较大规模的大众点评评论语料，在 Kaggle 比赛的时候，NLP 的语言模型预训练还没有出现，而随着 ELMo 之类模型的成功，也很期待尝试一下预训练语言模型在这个数据集合上的整体效果。

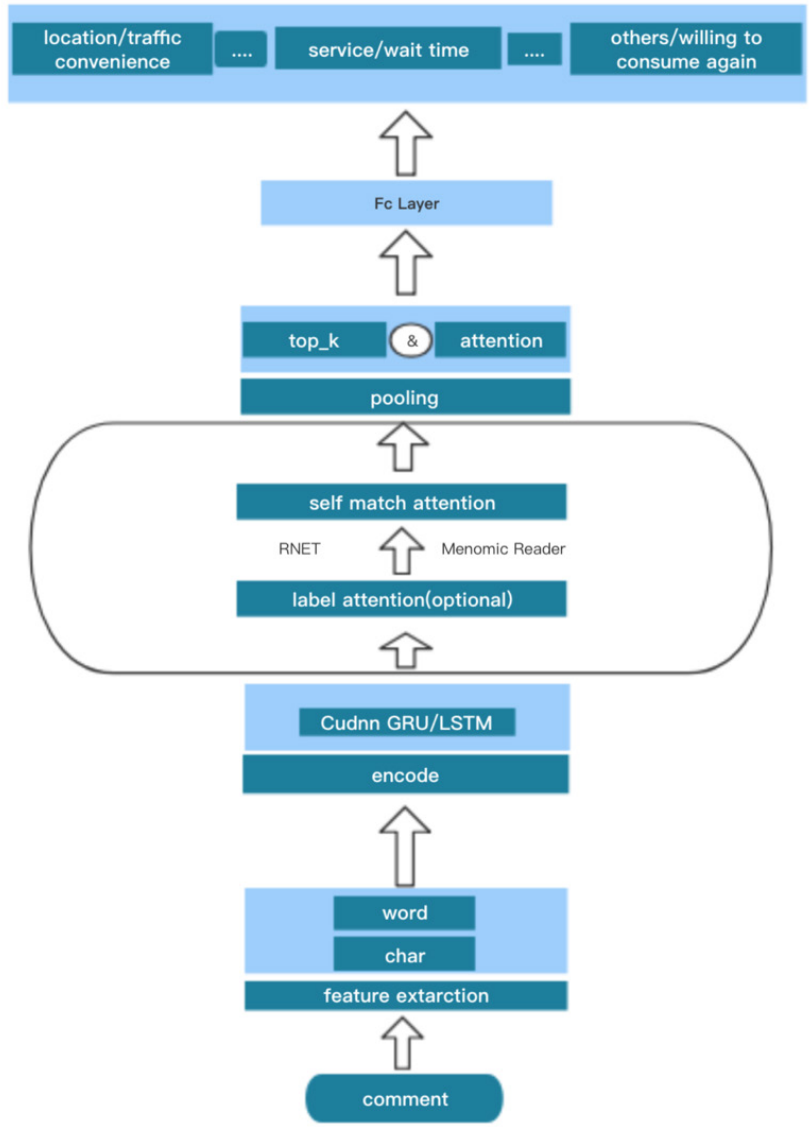
3. 基础模型思路

首先，尝试了不使用预训练语言模型的基础模型，基于 Kaggle Toxic 比赛的经验，直接使用了当时表现最好的 LSTM Encode + Pooling 作为基线模型。在 Kaggle 的比赛中，大家实验的普遍结果是针对中长文本的分类任务的最佳单模型，都是基于 RNN(LSTM/GRU) 或者部分基于 RNN 的模型，比如 RCNN、Capsule + RNN 这样的模型，而其他的模型，比如单纯的 CNN 结构相对表现较差，主要可能是因为 RNN 模型能更好地捕获相对较长距离的顺序信息。



4. 模型层面优化

在基线模型的基础上，效仿阅读理解常见的做法，增加了 Self Attention 层（计算文本到文本自身的 Attention 权重），并将 Attention 之后的输出和原始 LSTM 输出，采用 Gate(RNet) 或者 Semantic Fusion(MnemonicReader) 的方式进行融合。



5. 模型细节处理

更宽的参数更多的模型效果更好

- LSTM 效果好于 GRU。
- Hidden size 400 > 200 > 100。
- Topk Pooling + Attention Pooling 的效果好于单独的 Max 或者 Attention Pooling。
- 共享层前置，Pooling 层 和最后 Fc 层不同 aspect 参数独占效果更好（来自赛后实验，以及其他团队经验）。

这里推测主要原因：是这个数据集有 20 个 Aspect，每个 Aspect 分 4 个不同的类别，所需要的参数相对较多。

三角学习率调节效果最佳

- 参考 BERT 开源代码的学习率设置带来较大效果提升。

采用 Word + Char 的词建模方式

- 这种建模方式能结合分词和字符粒度切分的好处，最大限度避免词汇 UNK 带来的损失。
- 注意对比 Kaggle Toxic 比赛那次比赛是英文语料，对应英文，当时的实验结果是 Word + Ngram 的建模效果更好，收敛更快，所以针对不同 NLP 任务，我们需要具体进行分析。

采用尽可能大的词表

和其他团队相比，我采用了更大的词表 14.4W（Jieba 分词），19.8W（Sentence Piece Unigram 分词），依靠外部大众点评评论数据基于 fastText 预训练词向量，能够支持更大的词表。同时为了避免训练过拟合，采用了只 Finetune 训练中高频的词对低频词固定词向量的处理方式。

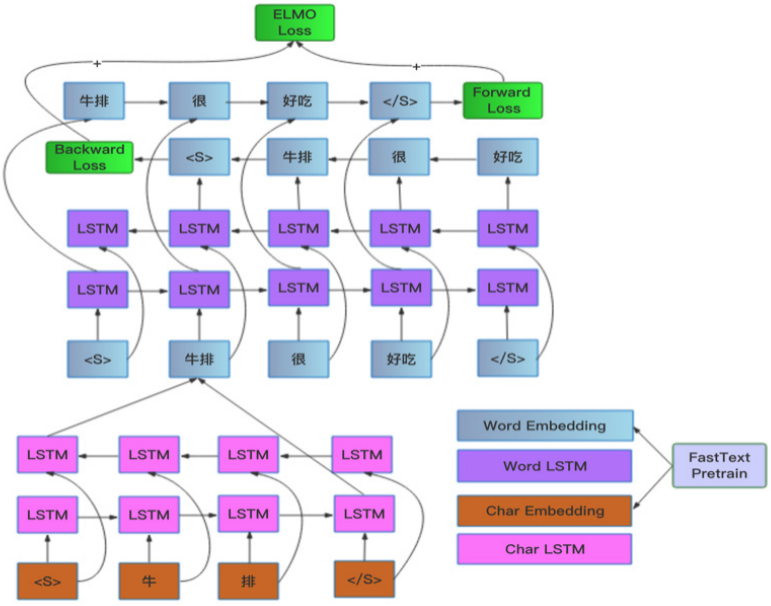
最开始，预计情感相关的词汇相对较少，不需要较大的词表，但是实验过程中发现更大的词表相对地能够提升性能，前提是利用较多的外部数据去比较好的刻画训练数据中低频词的向量。在理论上，我们可以采用一个尽可能大的词表在预测过程中去

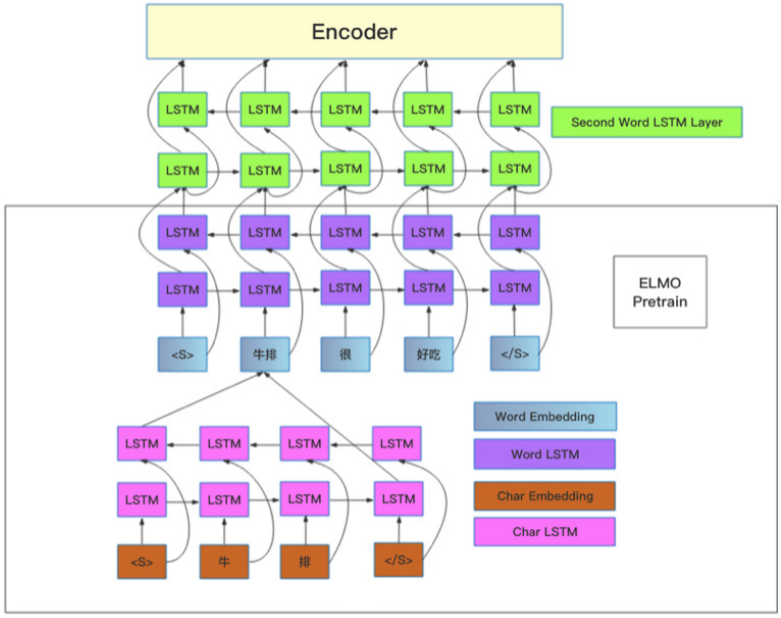
尽可能的减少 UNK 的存在（有论文的结论是对应 UNK 不同的词赋于不同随机向量效果，好于一个固定的 UNK 向量。这里类似，如果我们赋予一个基于无监督外部数据，通过语言模型训练得到的向量则效果更好）。

6. 预训练语言模型

这部分是模型效果提升的关键，这里采用了 ELMo Loss。在简单尝试了官方的 ELMo 版本之后，感觉速度相对比较慢，为此，采用了自己实现的一个简化版的 ELMo，实质上只使用了 ELMo 的 Loss 部分。

在当前双层 LSTM Encoder 的基础上，采用了最小代价的 ELMo 引入，也就是对当前模型的第一层 LSTM 进行基于 ELMo Loss 的预训练，而 Finetune 的时候，模型结构和之前完全不变，只是第一层 LSTM 以及词向量部分采用的 ELMo 预训练的初始化结果，另外在 ELMo 的训练过程中，也采用了基于 fastText 的词向量参数初始化。这个设计使得 ELMo 训练以及 Finetune 训练的收敛，都加快了很多，只需要大概 1 小时的 ELMo 训练，就能在下游任务产生明显受益。值得一提的是，ELMo 和 Self Attention 的搭配在这个数据集合效果非常好。





7. 模型集成

为了取得更好的模型多样性，采用了多种粒度的分词方式，在 Jieba 分词的主要模型基础上，同时引入了基于 SentencePiece 的多种粒度分词。SentencePiece 分词能带来更短的句子长度，但是分词错误相对 Jieba 略多，容易过拟合，因此采用了只 Finetune Char 向量，固定词向量的策略来避免过拟合。多种粒度的分词配合 Word + Char 的建模方式带来了很好的模型多样性。

此外，模型维度的多样性来源自 RNet 结构和 MnemonicReader 结构，以及 BERT 模型的结构的不同。

在模型选择的时候选取了平均 F1 值最优的轮次模型，集成的时候采用了按 Aspect 效果分开加权集成的方式（权重来自 Valid 数据的 F1 分值排序）。基于以上的多样性策略，只需要 7 个单模型集成就能取得较好的效果。

8. 关于 BERT

在实验中基于 Char 的 BERT 单模型，在本次比赛中并没有取得比 ELMo 更好的效果，受限于 512 的长度和只基于 Char 的限制，目前看起来 BERT 模型在这个

数据集合更容易过拟合，Train Loss 下降较快，对应 Valid Loss 效果变差。相信通过适当的优化 BERT 模型能取得更好的效果。

9. 后续优化

F1 的优化是一个有意思的方向。本次比赛中，没有对此做特殊处理，考虑到 F1 是一个全局优化值，如果基于 Batch 强化学习，每个 Batch 可能很难拟合稀有样本分布。

BERT 的进一步优化。因为 BERT 出现之前，基于 Transformer 的模型在长文本分类效果大都是差于基于 LSTM 的模型的，所以如果我们按照 BERT 的 Loss 去预训练基于 LSTM 而不是 Transformer 的模型，在分类问题层面的效果如何？另外，在这个数据集合基于 Transformer 的 BERT，能否取得比 ELMo 更好的分类效果？

对话 AI Challenger 2018 冠军：程惠阁

Q：谈谈对本次参赛的感受？

程惠阁：作为一个多年的算法从业者，我真实的感受到在 AI 时代，技术更新非常之快，比如席卷而来的 ELMo、BERT 等预训练语言模型在工业界影响力之大。包括美团在内的很多公司都快速跟进并上线，而且取得了很好收益，因此技术人员时刻保持学习的心态是非常重要的。

而比赛和工作存在很大的不同，比赛相对更加单纯明确，比赛可以使我在最短时间去学习实验验证一些新的技术，而在标准数据集合验证有效的模型策略，往往在工作中也有实际的价值。对于比赛以及工作中的模型开发，我觉得比较重要的一点首先要做好细致的模型验证部分，在此基础上逐步开发迭代模型才有意义。比如在这次比赛中，我从一开始就监控了包括整体以及各个 Aspect 的包括 F1、AUC、Loss 等等各项指标。

Q：对学习算法的新同学有哪些建议？

程惠阁：如果有时间，可以系统地学习一些名校的深度学习相关的课程，还有很重要的一点，就是实践，我们可以参加去学校项目或者去大公司实习，当然也可以利

用 AI Challenger、Kaggle 这样的竞赛平台进行实践。

Q：为什么会选择参加细粒度用户评论情感分类这个赛道？

程惠阁：因为我之前参加过类似的比赛，并且做过文本分类相关的工作，对这个赛道的赛题也比较感兴趣。

Q：本次比赛最有成就感的事情是什么？

程惠阁：不断迭代提升效果带来的成就感吧，特别是简化版 ELMo 带来的效果提升。

Q：参赛过程中，有哪些收获和成长？

程惠阁：作为一个 TensorFlow 重度用户，我学会了使用 PyTorch 并且体验到 PyTorch 带来的优雅与高效。体验到了预训练语言模型的威力。在比赛中和比赛后，我也收获了很多志同道合的朋友，和他们的交流学习，也帮助我提高了很多。

更重要的是，因为这次比赛，我加入了美团点评这个大家庭，入职这段时间，让我真切地感受到美团点评为了提升用户体验，为了让用户吃的更好，生活更好，在技术方面做了大量的投入。

WSDM Cup 2019 自然语言推理任务获奖解题思路

帅朋

WSDM（Web Search and Data Mining，读音为 Wisdom）是业界公认的高质量学术会议，注重前沿技术在工业界的落地应用，与 SIGIR 一起被称为信息检索领域的 Top2。

刚刚在墨尔本结束的第 12 届 WSDM 大会传来一个好消息，由美团搜索与 NLP 部 NLP 中心的刘帅朋、刘硕和任磊三位同学组成的 Travel 团队，在 WSDM Cup 2019 大赛“真假新闻甄别任务”中获得了第二名的好成绩。队长刘帅朋受邀于 2 月 15 日代表团队在会上作口头技术报告，向全球同行展示了来自美团点评的解决方案。本文将详细介绍他们本次获奖的解决方案。

1. 背景

信息技术的飞速发展，催生了数据量的爆炸式增长。技术的进步也使得了人们获取信息的方式变得更加便捷，然而任何技术都是一把“双刃剑”，信息技术在为人们的学习、工作和生活提供便利的同时，也对人类社会健康持续的发展带来了一些新的威胁。目前亟需解决的一个问题，就是如何有效识别网络中大量存在的“虚假新闻”。

虚假新闻传播了很多不准确甚至虚构的信息，对整个线上资讯的生态造成了很大的破坏，而且虚假新闻会对读者造成误导，干扰正常的社会舆论，严重的危害了整个社会的安定与和谐。因此，本届 WSDM Cup 的一个重要议题就是研究如何实现对虚假新闻的准确甄别，该议题也吸引了全球众多数据科学家的参与。

虽然美团点评的主营业务与在线资讯存在一些差异，但本任务涉及的算法原理是通用的，而且在美团业务场景中也可以有很多可以落地，例如虚假评论识别、智能客服中使用的问答技术、NLP 平台中使用的文本相似度计算技术、广告匹配等。于是，Travel 团队通过对任务进行分析，将该问题转化为 NLP 领域的“自然语言推理”(NLI) 任务，即判断给定的两段文本间的逻辑蕴含关系。因此，基于对任务较为深入理解和平时的技术积累，他们提出了一种解决方案——一种基于多层次深度模型融合框架的虚假新闻甄别技术，该技术以最近 NLP 领域炙手可热的 BERT 为基础模型，并在此基础上提出了一种多层次的模型集成技术。

2. 数据分析

为了客观地衡量算法模型的效果，本届大会组织方提供了一个大型新闻数据集，该数据集包含 32 万多个训练样本和 8 万多个测试样本，这些数据样本均取材于互联网上真实的数据。每个样本包含有两个新闻标题组成的标题对，其中标题对类别标签包括 Agreed、Disagreed、Unrelated 等 3 种。他们的任务就是对测试样本的标签类别进行预测。

“磨刀不误砍柴功”，在一开始，Travel 团队并没有急于搭建模型，而是先对数据进行了全面的统计分析。他们认为，如果能够通过分析发现数据的一些特性，就会有助于后续采取针对性的策略。

首先，他们统计了训练数据中的类别分布情况，如图 1 所示，Unrelated 类别占比最大，接近 70%；而 Disagreed 类占比最小，不到 3%。训练数据存在严重的类别不均衡问题，如果直接用这样的训练数据训练模型，这会导致模型对占比较大类的学习比较充分，而对占比较小的类别学习不充分，从而使模型向类别大的类别进行偏移，存在较严重的过拟合问题。后面也会介绍他们针对该问题提出的对应解决方案。

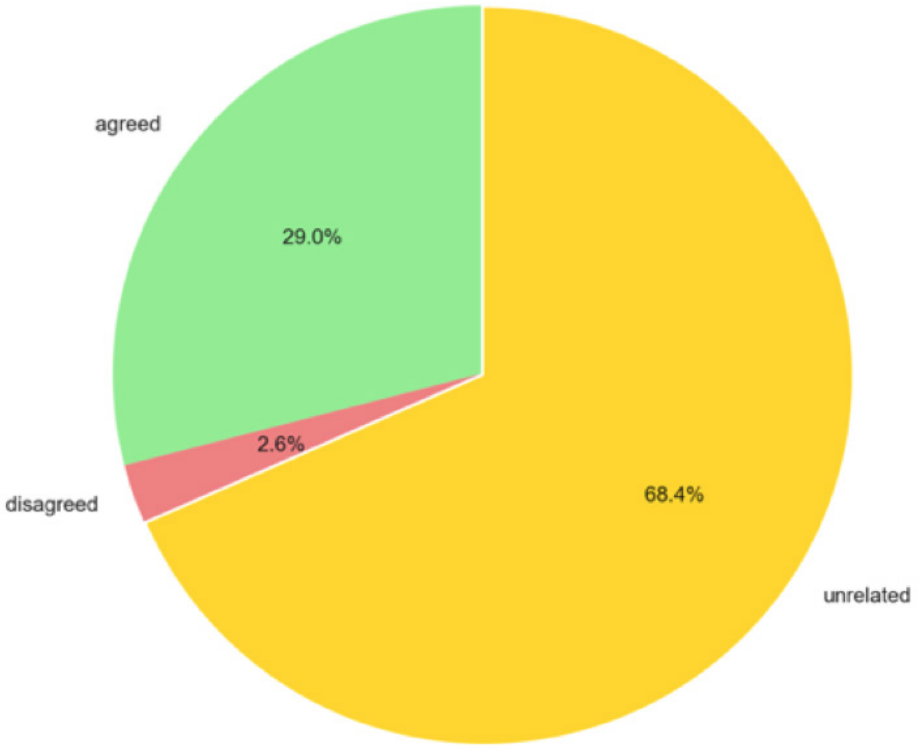


图 1　数据集中类别分布情况

然后，Travel 团队对训练数据的文本长度分布情况进行了统计，如图 2 所示，不同类别的文本长度分布基本保持一致，同时绝大多数文本长度分布在 20 ~ 100 内。这些统计信息对于后面模型调参有着很大的帮助。

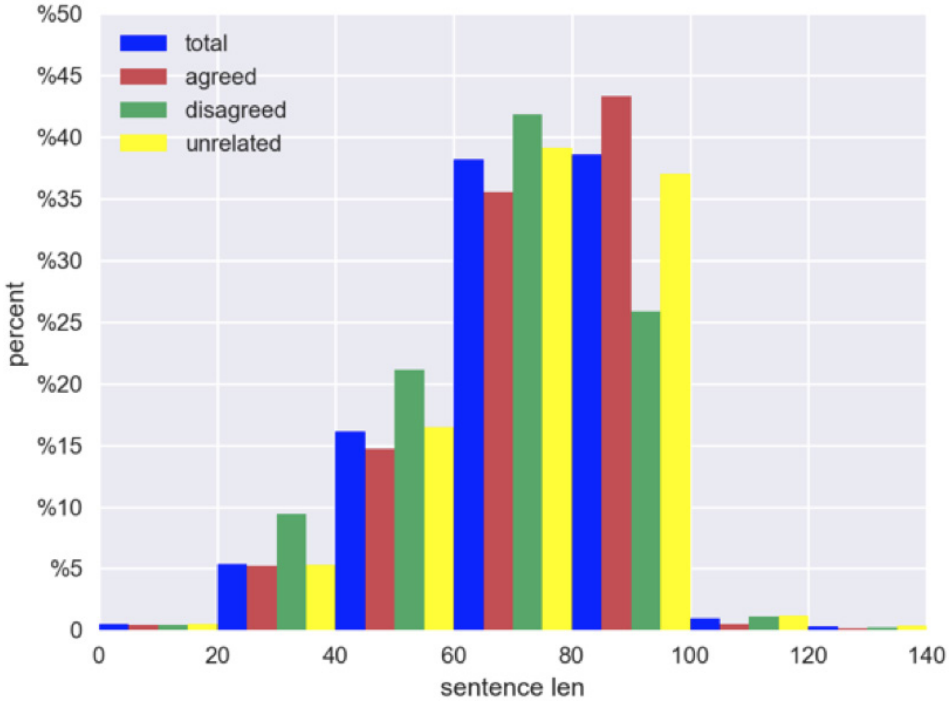


图 2　数据集中文本长度分布情况

3. 数据的预处理与数据增强

本着“数据决定模型的上限，模型优化只是不断地逼近这个上限”的想法，接下来，Travel 团队对数据进行了一系列的处理。

在数据分析时，他们发现训练数据存在一定的噪声，如果不进行人工干预，将会影响模型的学习效果。比如新闻文本语料中简体与繁体共存，这会加大模型的学习难度。因此，他们对数据进行繁体转简体的处理。同时，过滤掉了对分类没有任何作用的停用词，从而降低了噪声。

此外，上文提到训练数据中，存在严重的样本不均衡问题，如果不对该问题做针对性的处理，则会严重制约模型效果指标的提升。通过对数据进行了大量的分析后，他们提出了一个简单有效的缓解样本不均衡问题的方法，**基于标签传播的数据增强方法**。具体方法如图 3 所示：

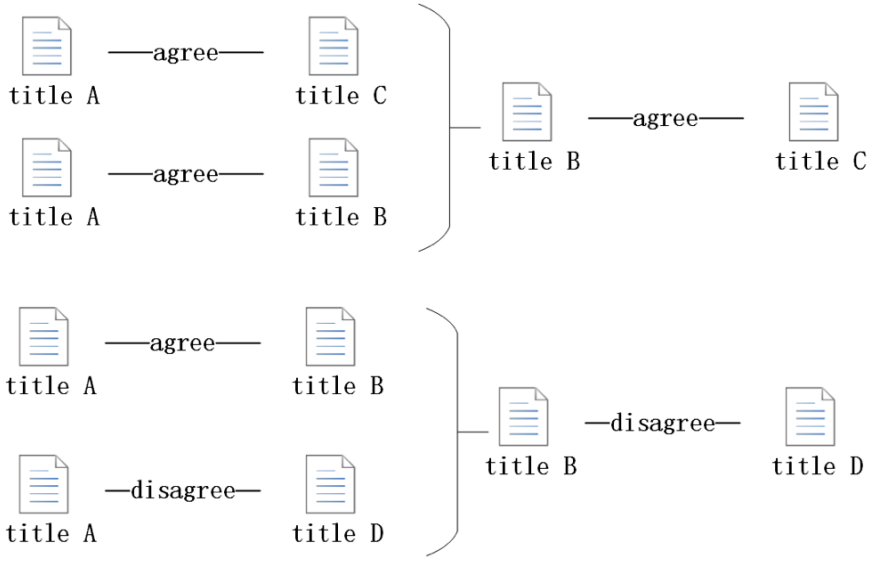


图 3　数据增强策略

如果标题 A 与标题 B 一致，而标题 A 与标题 C 一致，那么可以得出结论，标题 B 与标题 C 一致。同理，如果标题 A 与标题 B 一致，而标题 A 与标题 D 不一致，那么可以得出结论，标题 B 与标题 D 也不一致。此外，Travel 团队还通过将新闻对中的两条文本相互交换位置，来扩充训练数据集。

4. 基础模型

BERT 是 Google 最新推出的基于双向 Transformer 的大规模预训练语言模型，在 11 项 NLP 任务中夺得 SOTA 结果，引爆了整个 NLP 界。BERT 取得成功的一个关键因素是 Transformer 的强大特征提取能力。Transformer 可以利用 Self-Attention 机制实现快速并行训练，改进了 RNN 最被人所诟病的“训练慢”的缺点，可以高效地对海量数据进行快速建模。同时，BERT 拥有多层注意力结构（12 层或 24 层），并且在每个层中都包含有多个“头”（12 头或 16 头）。由于模型的权重不在层与层之间共享，一个 BERT 模型相当于拥有 12 × 12=224 或 24 × 16=384 种不同的注意力机制，不同层能够提取不同层次的文本或语义特征，这可以让 BERT 具有超强的文本表征能力。

本赛题作为典型的自然语言推理（NLI）任务，需要提取新闻标题的高级语义特征，BERT 的超强文本表征能力正好本赛题所需要的。基于上述考虑，Travel 团队的基础模型就采用了 BERT 模型，其中 BERT 网络结构如图 4 所示：

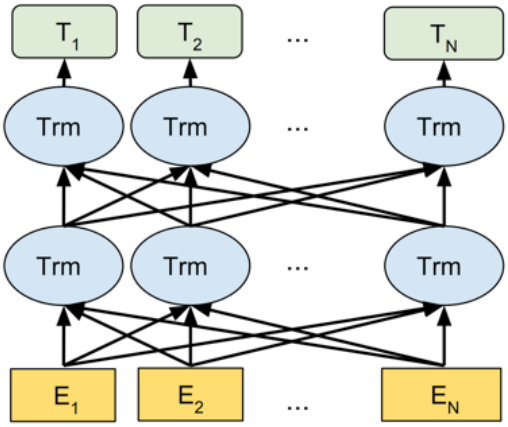


图 4　BERT 网络结构图

在比赛中，Travel 团队在增强后的训练数据上对 Google 预训练 BERT 模型进行了微调（Finetune），使用了如图 5 所示的方式。为了让后面模型融合增加模型的多样性，他们同时 Finetune 了中文版本和英文版本。

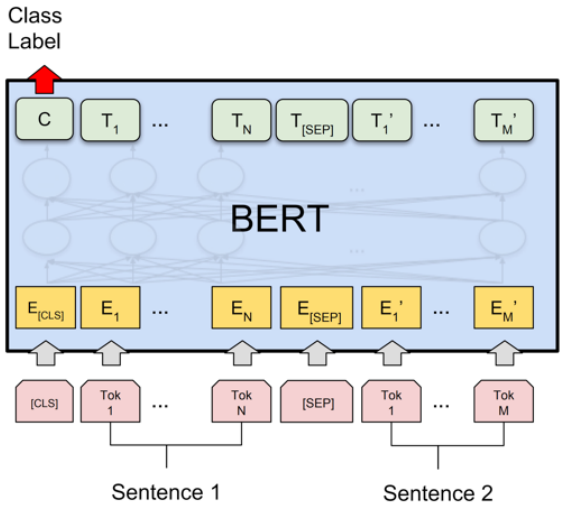


图 5　基于 BERT 的假新闻分类模型结构

5. 多层次深度模型融合框架

模型融合，是指对已有的多个基模型按照一定的策略进行集成以提升模型效果的一种技术，常见的技术包括 Voting、Averaging、Blending、Stacking 等等。这些模型融合技术在前人的许多工作中得到了应用并且取得了不错的效果，然而任何一种技术只有在适用场景下才能发挥出最好的效果，例如 Voting、Averaging 技术的融合策略较为简单，一般来说效果提升不是非常大，但优点是计算逻辑简单、计算复杂度低、算法效率高；而 Stacking 技术融合策略较复杂，一般来说效果提升比较明显，但缺点是算法计算复杂度高，对计算资源的要求较苛刻。

本任务使用的基模型为 BERT，该模型虽然拥有非常强大的表征建模能力，但同时 BERT 的网络结构复杂，包含的参数众多，计算复杂度很高，即使使用了专用的 GPU 计算资源，其训练速度也是比较慢的，因此这就要求在对 BERT 模型融合时不能直接使用 Stacking 这种高计算复杂度的技术，因此我们选择了 Blending 这种计算复杂度相对较低、融合效果相对较好的融合技术对基模型 BERT 做融合。

同时，Travel 团队借鉴了神经网络中网络分层的设计思想来设计模型融合框架，他们想既然神经网络可以通过增加网络深度来提升模型的效果，那么在模型融合中是否也可以通过增加模型融合的层数来提升模型融合的效果呢？基于这一设想，他们提出了一种多层次深度模型融合框架，该框架通过增加模型的层数进而提升了融合的深度，最终取得了更好的融合效果。

具体来说，他们的框架包括三个层次，共进行了两次模型融合。第一层采用 Blending 策略进行模型训练和预测，在具体实践中，他们选定了 25 个不同的 BERT 模型作为基模型；第二层采用 5 折的 Stacking 策略对 25 个基模型进行第一次融合，这里他们选用了支持向量机（SVM）、逻辑回归（LR）、K 近邻（KNN）、朴素贝叶斯（NB），这些传统的机器学习模型，既保留了训练速度快的优点，也保证了模型间的差异性，为后续融合提供了效率和效果的保证；第三层采用了一个线性的 LR 模型，进行第二次模型融合并且生成了最终的结果。模型融合的架构如图 6 所示：

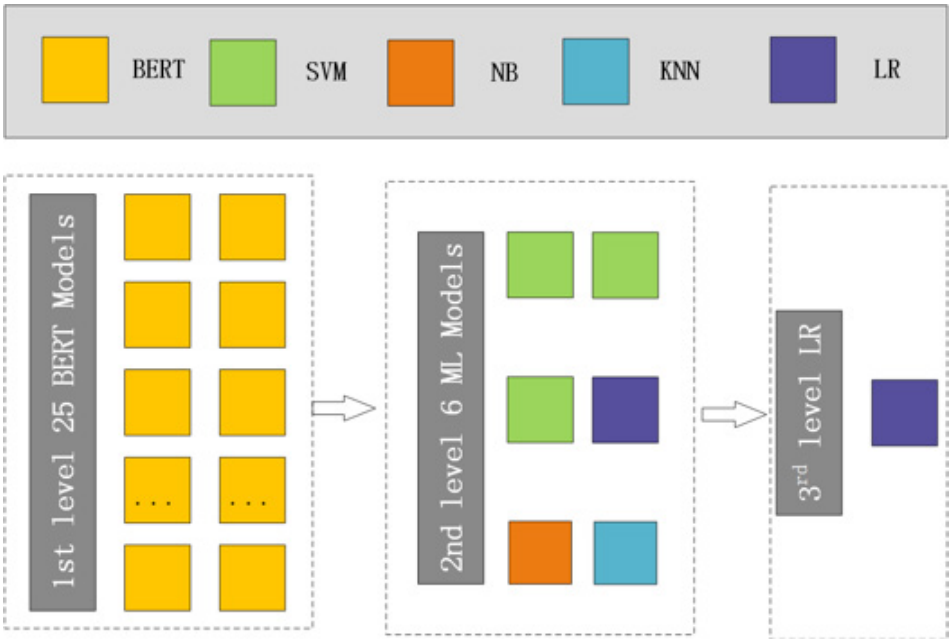


图 6　模型融合架构

整体方案模型训练分为三个阶段，如图 7 所示：

- 第一个阶段，将训练数据划分为两部分，分别为 Train Data 和 Val Data。Train Data 用于训练 BERT 模型，用训练好的 BERT 模型分别预测 Val Data 和 Test Data。将不同 BERT 模型预测的 Val Data 和 Test Data 的结果分别进行合并，可以得到一份新的训练数据 New Train Data 和一份新的测试数据 New Test Data。
- 第二阶段，将上一阶段的 New Train Data 作为训练数据，New Test Data 作为测试数据。本阶段将 New Train Data 均匀的划分为 5 份，使用“留一法”训练 5 个 SVM 模型，用这 5 个模型分别去预测剩下的一份训练数据和测试数据，将 5 份预测的训练数据合并，可以得到一份新的训练数据 NewTrainingData2，将 5 份预测的测试数据采用均值法合并，得到一份新的测试数据 NewTestData2。同样的方法再分别训练 LR、KNN、NB 等模型。
- 第三阶段，将上一阶段的 NewTrainingData2 作为训练数据，NewTestDa-

ta2 作为测试数据，重新训练一个 LR 模型，预测 NewTestData2 的结果作为最终的预测结果。为了防止过拟合，本阶段采用 5 折交叉验证的训练方式。

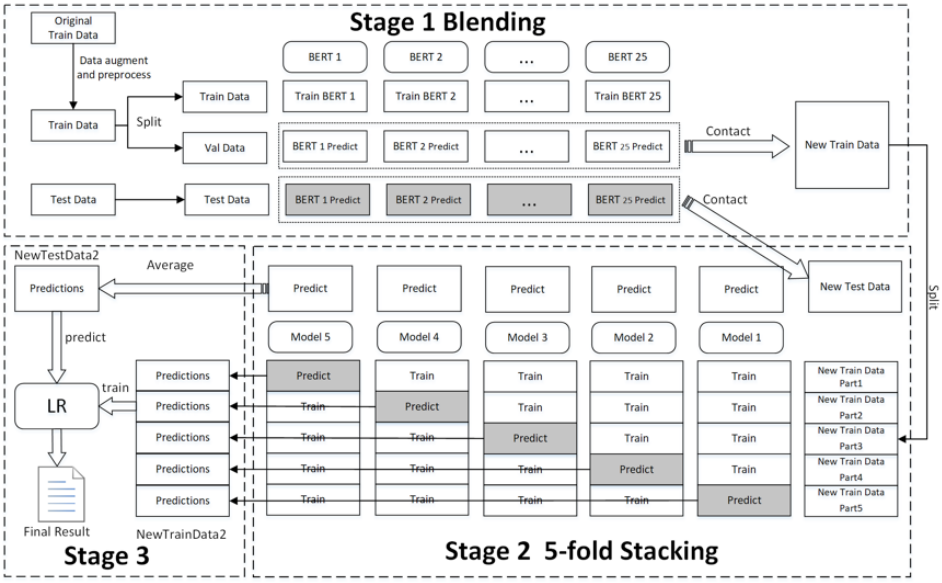


图 7　假新闻分类方案的整体架构和训练流程

6. 实验

6.1　评价指标

为了缓解数据集中存在的类别分布不均衡问题，本任务使用带权重的准确率作为衡量模型效果的评价指标，其定义如下所示：

$$weightedAccuracy(y, \hat{y}, \omega) = \frac{1}{n} \sum_{i=1}^{n} \frac{\omega_i (y_i = \hat{y})}{\sum \omega_i}$$

其中，y 为样本的真实类别标签，$\hat{y}$ 为模型的预测结果，ωi 为数据集中第 i 个样本的权重，其权重值与类别相关，其中 Agreed 类别的权重为 1/15，Disagreed 类别的权重为 1/5，Unrelated 类别的权重为 1/16。

6.2 实验结果

在官方测试集上，Travel 团队的最优单模型的准确率达到 0.86750，25 个 BERT 模型简单平均融合后准确率达 0.87700（+0.95PP），25 个 BERT 模型结果以加权平均的形式融合后准确率达 0.87702（+0.952PP），他们提出的多层次模型融合技术准确率达 0.88156（+1.406PP）。实践证明，美团 NLP 中心的经验融合模型在假新闻分类任务上取得了较大的效果提升。

Table 1: Performance of Various Models

Model	Weighted Acc on Private LB
Best Single base model	0.86750
Averaging of 25 BERT	0.87700
Weighted Averaging of 25 BERT	0.87702
Our Empirical Ensemble Model	0.88156

图 8　效果提升

7. 总结与展望

本文主要对解决方案中使用的关键技术进行了介绍，比如数据增强、数据预处理、多层模型融合策略等，这些方法在实践中证明可以有效的提升预测的准确率。由于参赛时间所限，还有很多思路没有来及尝试，例如美团使用的 BERT 预训练模型是基于维基百科数据训练而得到的，而维基百科跟新闻在语言层面也存在较大的差异，所以可以将现有的 BERT 在新闻数据上进行持续地训练，从而使其能够对新闻数据具有更好的表征能。

参考文献

[1] Dagan, Ido, Oren Glickman, and Bernardo Magnini. 2006. The PASCAL recognising textual entailment challenge, Machine learning challenges. evaluating predictive uncertainty, visual object classification, and recognising tectual entailment. Springer, Berlin, Heidelberg, 177–190.

[2] Bowman S R, Angeli G, Potts C, et al. 2015. A large annotated corpus for learning natural language inference. In proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP).

[3] Adina Williams, Nikita Nangia, and Samuel R Bowman. 2018. A broad-coverage challenge corpus for sentence understanding through inference. In NAACL.
[4] Rajpurkar P, Zhang J, Lopyrev K, et al. 2016. Squad: 100,000+ questions for machine comprehension of text. arXiv preprint arXiv:1606.05250.
[5] Luisa Bentivogli, Bernardo Magnini, Ido Dagan, Hoa Trang Dang, and Danilo Giampiccolo. 2009. The fifth PASCAL recognizing textual entailment challenge. In TAC. NIST.
[6] Hector J Levesque, Ernest Davis, and Leora Morgenstern. 2011. The winograd schema challenge. In Aaai spring symposium: Logical formalizations of commonsense reasoning, volume 46, page 47.
[7] Bowman, Samuel R., et al. 2015. "A large annotated corpus for learning natural language inference." arXiv preprint arXiv:1508.05326.
[8] Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., & Bowman, S. R. 2018. GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding. arXiv preprint arXiv:1804.07461.
[9] Chen, Q., Zhu, X., Ling, Z., Wei, S., Jiang, H., & Inkpen, D. 2016. Enhanced lstm for natural language inference. arXiv preprint arXiv:1609.06038.
[10] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding with unsupervised learning. Technical report, OpenAI.
[11] Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805.
[12] David H. Wolpert. 1992. Stacked generalization. Neural Networks (1992). https: // doi.org/10.1016/S0893- 6080(05)80023- 1.

作者简介

刘帅朋，硕士，美团点评搜索与 NLP 部 NLP 中心高级算法工程师，目前主要从事 NLU 相关工作。曾任中科院自动化研究所研究助理，主持研发的智能法律助理课题获 CCTV-1 频道大型人工智能节目《机智过人第二季》报道。

刘硕，硕士，美团点评搜索与 NLP 部 NLP 中心智能客服算法工程师，目前主要从事智能客服对话平台中离线挖掘相关工作。

任磊，硕士，美团点评搜索与 NLP 部 NLP 中心知识图谱算法工程师，目前主要从事美团大脑情感计算以及 BERT 应用相关工作。

会星，博士，担任美团点评搜索与 NLP 部 NLP 中心的研究员，智能客服团队负责人。目前主要负责美团智能客服业务及智能客服平台的建设。在此之前，会星在阿里达摩院语音实验室作为智能语音对话交互专家，主要负责主导的产品有斑马智行语音交互系统，YunOS 语音助理等，推动了阿里智能对话交互体系建设。

富峥，博士，担任美团点评搜索与 NLP 部 NLP 中心的研究员，带领知识图谱算法团队。目前主要负责美团大脑项目，围绕美团吃喝玩乐场景打造的知识图谱及其应用，能够打通餐饮、旅行、休闲娱乐等各个场景数据，为美团各场景业务提供更加智能的服务。张富峥博士在知识图

谱、个性化推荐、用户画像、时空数据挖掘等领域展开了众多的创新性研究，并在相关领域的顶级会议和期刊上发表 30 余篇论文，如 KDD、WWW、AAAI、IJCAI、TKDE、TIST 等，曾获 ICDM2013 最佳论文大奖，出版学术专著 1 部。

仲远，博士，美团点评搜索与 NLP 部负责人。在国际顶级学术会议发表论文 30 余篇，获得 ICDE 2015 最佳论文奖，并是 ACL 2016 Tutorial“Understanding Short Texts”主讲人，出版学术专著 3 部，获得美国专利 5 项。此前，博士曾担任微软亚洲研究院主管研究员，以及美国 Facebook 公司 Research Scientist。曾负责微软研究院知识图谱、对话机器人项目和 Facebook 产品级 NLP Service。

深度学习在美团配送 ETA 预估中的探索与实践

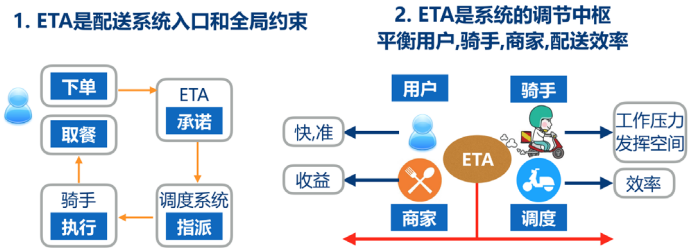
基泽　周越　显杰

1. 背景

ETA（Estimated Time of Arrival，“预计送达时间”），即用户下单后，配送人员在多长时间内将外卖送达到用户手中。送达时间预测的结果，将会以”预计送达时间”的形式，展现在用户的客户端页面上，是配送系统中非常重要的参数，直接影响了用户的下单意愿、运力调度、骑手考核，进而影响配送系统整体成本和用户体验。

对于整个配送系统而言，ETA 既是配送系统的入口和全局约束，又是系统的调节中枢。具体体现在：

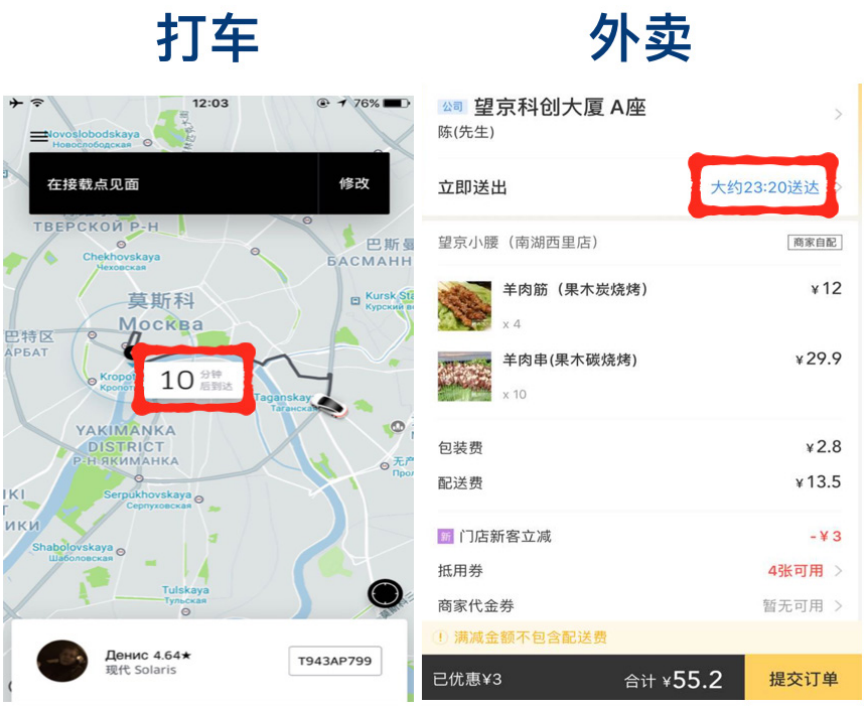
- ETA 在用户下单时刻就需要被展现，这个预估时长继而会贯穿整个订单生命周期，首先在用户侧给予准时性的承诺，接着被调度系统用作订单指派的依据及约束，而骑手则会按照这个 ETA 时间执行订单的配送，配送是否准时还会作为骑手的工作考核结果。
- ETA 作为系统的调节中枢，需要平衡用户 – 骑手 – 商家 – 配送效率。从用户的诉求出发，尽可能快和准时，从骑手的角度出发，太短会给骑手极大压力。从调度角度出发，太长或太短都会影响配送效率。而从商家角度出发，都希望订单被尽可能派发出去，因为这关系到商家的收入。



ETA 在配送系统中作用

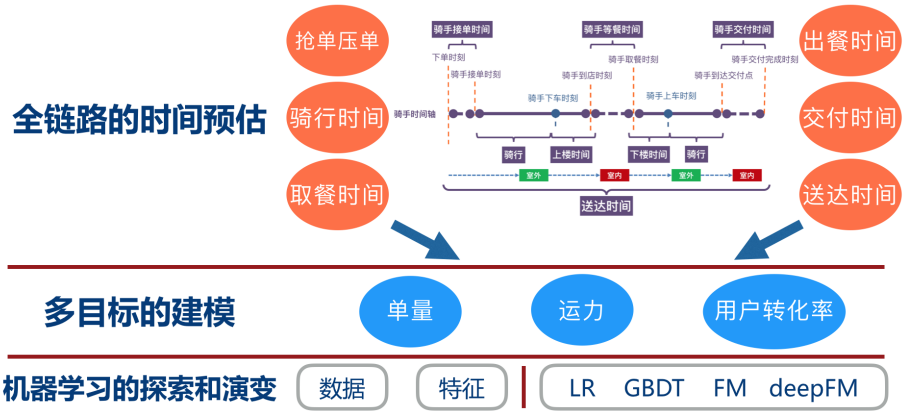
在这样多维度的约束之下，外卖配送的 ETA 的建模和估计会变得更加复杂。与打车场景中的 ETA 做对比，外卖场景的 ETA 面临如下的挑战：

- 外卖场景中 ETA 是对客户履约承诺的重要组成部分，无论是用户还是骑手，对于 ETA 准确性的要求非常高。而在打车场景，用户更加关心是否能打到车，ETA 仅提供一个参考，司机端对其准确性也不是特别在意。
- 由于外卖 ETA 承担着承诺履约的责任，因此是否能够按照 ETA 准时送达，也是外卖骑手考核的指标、配送系统整体的重要指标；承诺一旦给出，系统调度和骑手都要尽力保证准时送达。因此过短的 ETA 会给骑手带来极大的压力，并降低调度合单能力、增加配送成本；过长的 ETA 又会很大程度影响用户体验。
- 外卖场景中 ETA 包含更多环节，骑手全程完成履约过程，其中包括到达商家、商家出餐、等待取餐、路径规划、不同楼宇交付等较多的环节，且较高的合单率使得订单间的流程互相耦合，不确定性很大，做出合理的估计也有更高难度。



外卖及打车中的 ETA

下图是骑手履约全过程的时间轴，过程中涉及各种时长参数，可以看到有十几个节点，其中关键时长达到七个。这些时长涉及多方，比如骑手（接 – 到 – 取 – 送）、商户（出餐）、用户（交付），要经历室内室外的场景转换，因此挑战性非常高。对于 ETA 建模，不光是简单一个时间的预估，更需要的是全链路的时间预估，同时更需要兼顾”单量 – 运力 – 用户转化率”转化率之间的平衡。配送 ETA 的演变包括了数据、特征层面的持续改进，也包括了模型层面一路从 LR–XGB–FM–DeepFM– 自定义结构的演变。



ETA 的探索与演变

具体 ETA 在整个配送业务中的位置及配送业务的整体机器学习实践，请参看《机器学习在美团配送系统的实践：用技术还原真实世界》。

2. 业务流程迭代中的模型改进

2.1 基础模型迭代及选择

与大部分 CTR 模型的迭代路径相似，配送 ETA 模型的业务迭代经历了 LR–> 树模型 –>Embedding–>DeepFM–> 针对性结构修改的路径。特征层面也进行不断迭代和丰富。

- 模型维度从最初考虑特征线性组合，到树模型做稠密特征的融合，到 Embedding 考虑 ID 类特征的融合，以及 FM 机制低秩分解后二阶特征组合，最终通

过业务指标需求，对模型进行针对性调整。

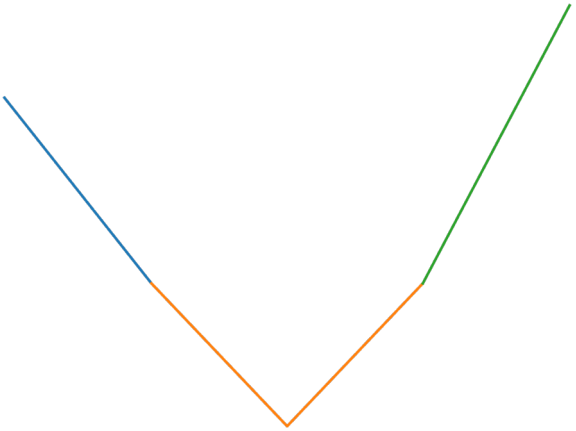
- 特征维度逐步丰富到用户画像 / 骑手画像 / 商家画像 / 地址特征 / 轨迹特征 / 区域特征 / 时间特征 / 时序特征 / 订单特征等维度。

目前版本模型在比较了 Wide&Deep、DeepFM、AFM 等常用模型后，考虑到计算性能及效果，最终选择了 DeepFM 作为初步的 Base 模型。整个 DeepFM 模型特征 Embedding 化后，在 FM（Factorization Machine）基础上，进一步加入 deep 部分，分别针对稀疏及稠密特征做针对性融合。FM 部分通过隐变量内积方式考虑一阶及二阶的特征融合，DNN 部分通过 Feed-Forward 学习高阶特征融合。模型训练过程中采取了 Learning Decay/Clip Gradient/ 求解器选择 /Dropout/ 激活函数选择等，在此不做赘述。

2.2 损失函数

在 ETA 预估场景下，准时率及置信度是比较重要的业务指标。初步尝试将 Square 的损失函数换成 Absolute 的损失函数，从直观上更为切合 MAE 相比 ME 更为严苛的约束。在适当 Learning Decay 下，结果收敛且稳定。

同时，在迭代中考虑到相同的 ETA 承诺时间下，在前后 N 分钟限制下，早到 1min 优于晚到 1min，损失函数的设计希望整体的预估结果能够尽量前倾。对于提前部分，适当降低数值惩罚。对于迟到部分，适当增大数值惩罚。进行多次调试设计后，最终确定以前后 N 分钟以及原点作为 3 个分段点。在原先 absolute 函数优化的基础上，在前段设计 1.2 倍斜率 absolute 函数，后段设计 1.8 倍斜率 absolute 函数，以便让结果整体往中心收敛，且预估结果更倾向于提前送达，对于 ETA 各项指标均有较大幅度提升。



损失函数

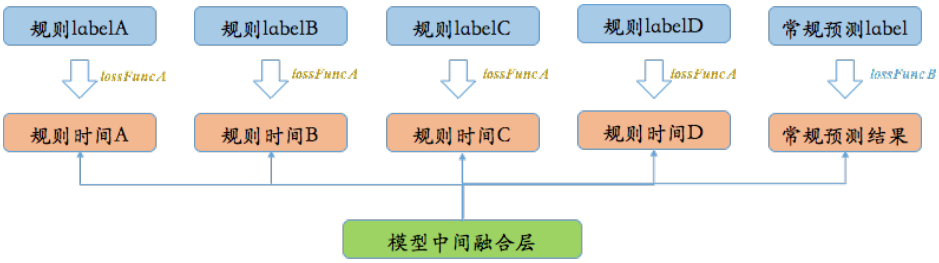
2.3 业务规则融入模型

目前的业务架构是”模型 + 规则”，在模型预估一个 ETA 值之后，针对特定业务场景，会有特定业务规则时间叠加以满足特定场景需求，各项规则由业务指标多次迭代产生。这里产生了模型和规则整体优化的割裂，在模型时间和规则时间分开优化后，即模型训练时并不能考虑到规则时间的影响，而规则时间在一年之中不同时间段，会产生不同的浮动，在经过一段时间重复迭代后，会加大割裂程度。

在尝试了不同方案后，最终将整体规则写入到了 TF 模型中，在 TF 模型内部调整整体规则参数。

- 对于简单的 (a*b+c)*d 等规则，可以将规则逻辑直接用 TF 的 OP 算子来实现，比如当 b、d 为定值时，则 a、c 为可学习的参数。
- 对于过于复杂的规则部分，则可以借助一定的模型结构，通过模型的拟合来代替，过多复杂 OP 算子嵌套并不容易同时优化。

通过调节不同的拟合部分及参数，将多个规则完全在 TF 模型中实现。最终对业务指标具备很大提升效果，且通过对部分定值参数的更改，具备部分人工干涉模型能力。



多目标补时结构

在这里，整体架构就简化为多目标预估的架构，这里采用多任务架构中常用的 Shared Parameters 的结构，训练时按比例采取不同的交替训练策略。结构上从最下面的模型中间融合层出发，分别在 TF 内实现常规预测结构及多个规则时间结构，而其对应的 Label 则仍然从常规的历史值和规则时间值中来，这样考虑了以下几点：

- 模型预估时，已充分考虑到规则对整体结果的影响（例如多个规则的叠加效应），作为整体一起考虑。
- 规则时间作为辅助 Label 传入模型，对于模型收敛及 Regularization，起到进一步作用。
- 针对不同的目标预估，采取不同的 Loss，方便进行针对性优化，进一步提升效果。

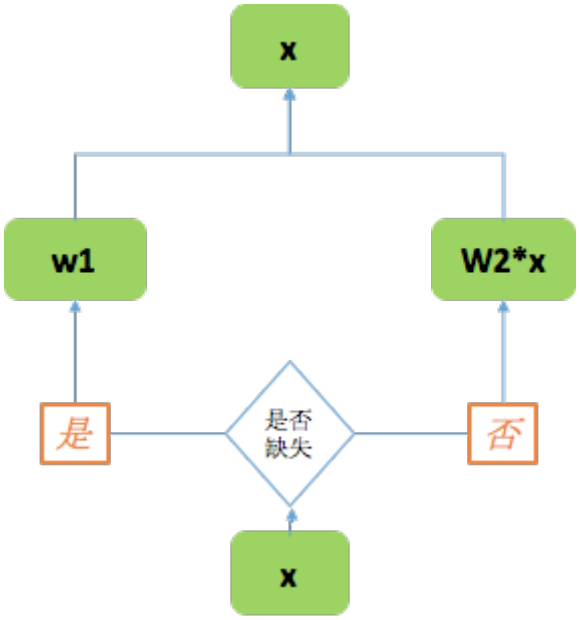
模型结构在进行预估目标调整尝试中：

- 尝试过固定共享网络部分及不固定共享部分参数，不固定共享参数效果明显。
- 通常情况下激活函数差异不大，但在共享层到独立目标层中，不同的激活函数差异很大。

2.4 缺失值处理

在模型处理中，特征层面不可避免存在一定的缺失值，而对于缺失值的处理，完全借鉴了《美团“猜你喜欢”深度学习排序模型实践》文章中的方法。对于特征 x 进入 TF 模型，进行判断，如果是缺失值，则设置 w1 参数，如果不是缺失值则进入模型数值为 w2*x，这里将 w1 和 w2 作为可学习参数，同时放入网络进行训练。以此

方法来代替均值 / 零值等作为缺失值的方法。



缺失值处理

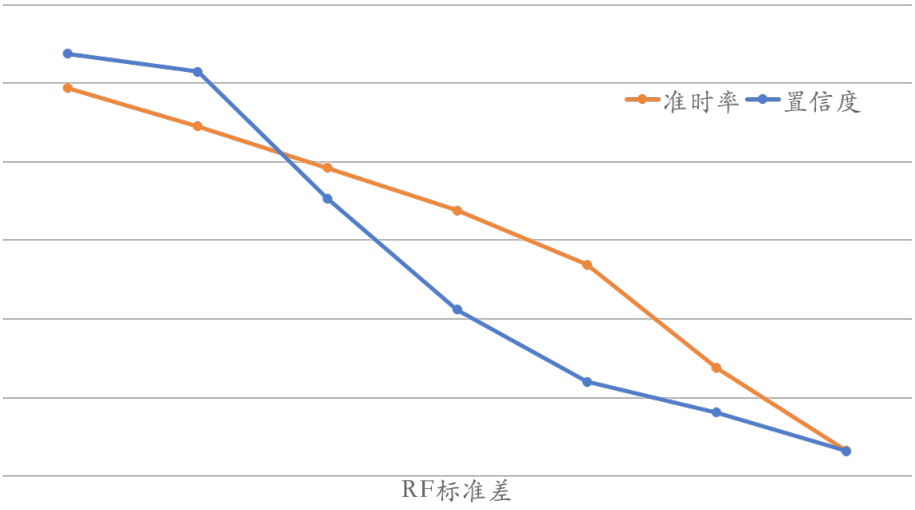
3. 长尾问题优化

3.1 模型预估结果 + 长尾规则补时

基础模型学习的是整体的统计分布，但对于一些长尾情形的学习并不充分，体现在长尾情形下预估时间偏短（由于 ETA 拥有考核骑手的功能，预估偏短对骑手而言意味着很大的伤害）。故将长尾拆解成两部分来分析：

- 业务长尾，即整体样本分布造成的长尾。主要体现在距离、价格等维度。距离越远，价格越高，实际送达时间越长，但样本占比越少，模型在这一部分上的表现整体都偏短。
- 模型长尾，即由于模型自身对预估值的不确定性造成的长尾。模型学习的是整体的统计分布，但不是对每个样本的预估都有“信心”。实践中采用 RF 多棵决策树输出的标准差来衡量不确定性。RF 模型生成的决策树是独立的，每棵

树都可以看成是一个专家，多个专家共同打分，打分的标准差实际上就衡量了专家们的“分歧”程度（以及对预估的“信心”程度）。从下图也可以看出来，随着 RF 标准差的增加，模型的置信度和准时率均在下降。



模型长尾因子

在上述拆解下，采用补时规则来解决长尾预估偏短的问题：长尾规则补时为 < 业务长尾因子，模型长尾因子 > 组合。其中业务长尾因子为距离、价格等业务因素，模型长尾因子为 RF 标准差。最终的 ETA 策略即为模型预估结果 + 长尾规则补时。

4. 工程开发实践

4.1 训练部分实践

整体训练流程

对于线下训练，采取如下训练流程：

Spark 原始数据整合 -> Spark 生成 TFRecord -> 数据并行训练 -> Tensor-Flow Serving 线下 GPU 评估 -> CPU Inference 线上预测

整个例行训练亿级数据多轮 Epoch 下流程持续约 4 小时，其中 TF 训练中，考虑到 TF 实际计算效率并不是很高，有很大比例在数据 IO 部分，通过 Spark 生成

TFRecord 部分，在此可将速度加速约 3.6 倍。而在数据并行训练部分，16 卡内的并行度扩展基本接近线性，具备良好的扩展性。由于 PS 上参数量并未达到单机无法承受，暂时未对参数在 PS 上进行切分。Serving 线下 GPU 评估部分，是整个流程中的非必需项，虽然在训练过程中 Chief Worker 设置 Valid 集合可有一定的指标，但对全量线下，通过 Spark 数据调用 Serving GPU 的评估具备短时间内完成全部流程能力，且可以指定大量复杂自定义指标。

数据并行训练方式

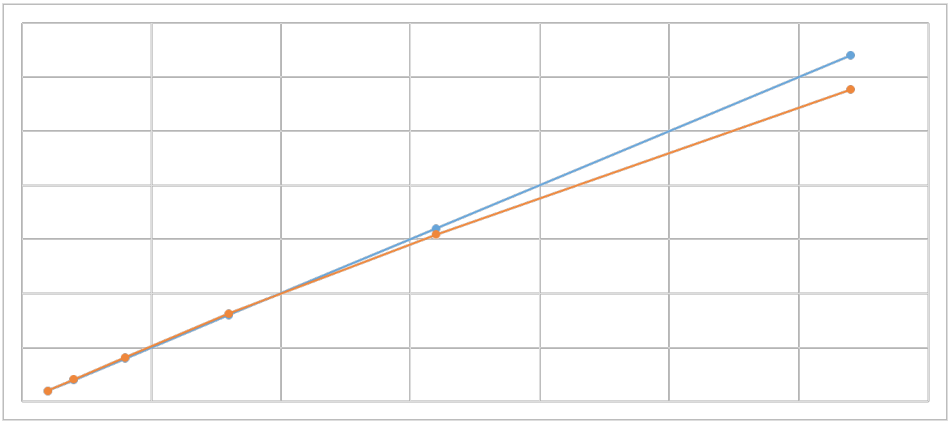
整个模型的训练在美团的 AFO 平台上进行，先后尝试分布式方案及单机多卡方案。考虑到生产及结果稳定性，目前线上模型生产采用单机多卡方案进行例行训练。

- 分布式方案：

采用 TF 自带的 PS-Worker 架构，异步数据并行方式，利用 tf.train.MonitoredTrainingSession 协调整个训练过程。整个模型参数存储于 PS，每个 Step 上每个 Worker 拉取数据进行数据并行计算，同时将梯度返回，完成一次更新。目前的模型单 Worker 吞吐 1~2W/s，亿级数据几轮 Epoch 耗时在几小时内完成。同时测试该模型在平台上的加速比，大约在 16 块内，计算能力随着 Worker 数目线性增加，16 卡后略微出现分离。在目前的业务实践中，基本上 4-6 块卡可以短时间内完成例行的训练任务。

- 单机多卡方案：

采用 PS-Worker 的方案在平台上具备不错的扩展性，但是也存在一定的弊端，使用 RPC 的通讯很容易受到其他任务的影响，整个的训练过程受到最慢 Worker 的影响，同时异步更新方式对结果也存在一定的波动。对此，在线上生产中，最终选取单机多卡的方案，牺牲一定的扩展性，带来整体训练效果和训练速度的稳定性。单机多卡方案采取多 GPU 手动指定 OP 的 Device，同时在各个 Device 内完成变量共享，最后综合 Loss 与梯度，将 Grad 更新到模型参数中。



加速比曲线

TF 模型集成预处理

模型训练过程中，ID 类特征低频过滤需要用到 Vocab 词表，连续型特征都需要进行归一化。这里会产生大量的预处理文件，在线下处理流程中很容易在 Spark 中处理成 Libsvm 格式，然后载入到模型中进行训练。但是在线上预测时，需要在工程开发端载入多个词表及连续型特征的归一化预处理文件（avg/std 值文件等），同时由于模型是按天更新，存在不同日期版本的对齐问题。

为了简化工程开发中的难度，在模型训练时，考虑将所有的预处理文件写入 TF 计算图之中，每次在线预测只要输入最原始的特征，不经过工程预处理，直接可得到结果：

- 对于 ID 类特征，需要进行低频过滤，然后制作成词表，TF 模型读入词表的 list_arr，每次 inference 通过 ph_vals，得到对应词表的 ph_idx。

```
tf_look_up = tf.constant(list_arr, dtype=tf.int64)
table = tf.contrib.lookup.HashTable(tf.contrib.lookup.
KeyValueTensorInitializer(tf_look_up, idx_range), 0)
ph_idx = table.lookup(ph_vals) + idx_bias
```

对于连续型特征，在 Spark 处理完得到 avg/std 值后，直接写入 TF 模型计算图中，作为 constant 节点，每个 ph_in 经过两个节点，得到相应 ph_out。

```
constant_avg = tf.constant(feat_avg, dtype=tf.float32, shape=[feat_dim],
name="avg")
constant_std = tf.constant(feat_std, dtype=tf.float32, shape=[feat_dim],
name="std")
ph_out = (ph_in - constant_avg) / constant_std
```

4.2 TF 模型线上预测

配送机器学习平台内置了模型管理平台，对算法训练产出的模型进行统一管理和调度，管理线上模型所用的版本，并支持模型版本的切换和回退，同时也支持节点模型版本状态的管理。

ETA 使用的 DeepFM 模型用 TensorFlow 训练，生成 SavedModel 格式的模型，需要模型管理平台支持 Tensorflow SavedModel 格式。

实现方案 S 线上服务加载 TensorFlow SavedModel 模型有多种实现方案：

- 自行搭建 TensorFlow Serving CPU 服务，通过 gRPC API 或 RESTful API 提供服务，该方案实现比较简单，但无法与现有的基于 Thrift 的模型管理平台兼容。
- 使用美团 AFO GPU 平台提供的 TensorFlow Serving 服务。
- 在模型管理平台中通过 JNI 调用 TensorFlow 提供的 Java API TensorFlow Java API，完成模型管理平台对 SavedModel 格式的支持。

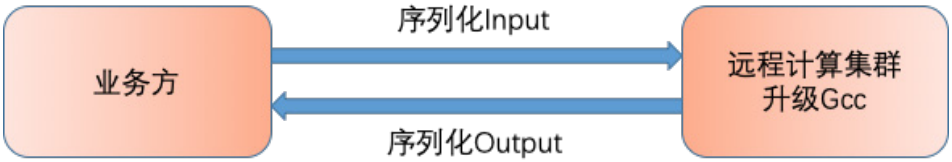
最终采用 TensorFlow Java API 加载 SavedModel 在 CPU 上做预测，测试 batch=1 时预测时间在 1ms 以内，选择方案 3 作为实现方案。

远程计算模式

TensorFlow Java API 的底层 C++ 动态链接库对 libstdc++.so 的版本有要求，需要 GCC 版本不低于 4.8.3，而目前线上服务的 CPU 机器大部分系统为 CentOS 6, 默认自带 GCC 版本为 4.4.7。如果每台线上业务方服务器都支持 TensorFlow SavedModel 本地计算的话，需要把几千台服务器统一升级 GCC 版本，工作量比较大而且可能会产生其他风险。

因此，我们重新申请了几十台远程计算服务器，业务方服务器只需要把 Input 数

据序列化后传给 TensorFlow Remote 集群，Remote 集群计算完后再将 Output 序列化后返回给业务方。这样只需要对几十台计算服务器升级就可以了。



在线序列化

线上性能

模型上线后，支持了多个业务方的算法需求，远程集群计算时间的 TP99 基本上在 5ms 以内，可以满足业务方的计算需求。

线上效果

总结与展望

模型落地并上线后，对业务指标带来较大的提升。后续将会进一步根据业务优化模型，进一步提升效果：

- 将会进一步丰富多目标学习框架，将取餐、送餐、交付、调度等整个配送生命周期内的过程在模型层面考虑，对订单生命周期内多个目标进行建模，同时提升模型可解释性。
- 模型融合特征层面的进一步升级，在 Embedding 以外，通过更多的 LSTM/CNN/ 自设计结构对特征进行更好的融合。
- 特征层面的进一步丰富。

作者简介

基泽，美团点评技术专家，目前负责配送算法策略部机器学习组策略迭代工作。
周越，2017 年加入美团配送事业部算法策略组，主要负责 ETA 策略开发。
显杰，美团点评技术专家，2018 年加入美团，目前主要负责配送算法数据平台深度学习相关的研发工作。

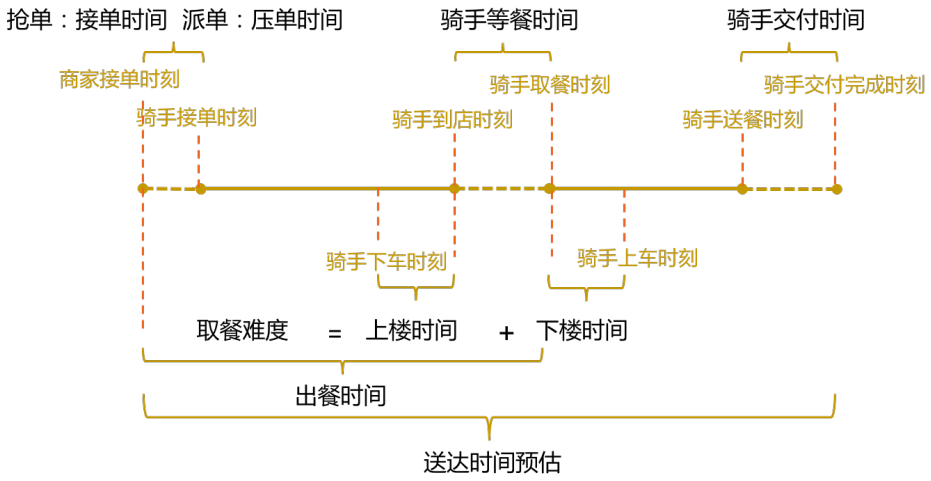
配送交付时间轻量级预估实践

基泽　闫聪

1. 背景

可能很多同学都不知道，从打开美团 App 点一份外卖开始，然后在半小时内就可以从骑手小哥手中拿到温热的饭菜，这中间涉及的环节有多么复杂。而美团配送技术团队的核心任务，就是将每天来自祖国各地的数千万份订单，迅速调度几十万骑手小哥按照最优路线，并以最快的速度送到大家手中。

在这种场景下，骑手的交付时间，即骑手到达用户附近下车后多久能送到用户手中 ，就是一个非常重要的环节。下图是一个订单在整个配送链路的时间构成，时间轴最右部分描述了交付环节在整个配送环节中的位置。交付时间衡量的是骑手送餐时的交付难度，包括从骑手到达用户楼宇附近，到将餐品交付到用户手中的整个时间。



交付时间的衡量是非常有挑战的一件事，因为骑手在送餐交付到用户手中时会碰到不同的问题，例如：骑手一次送餐给楼宇内多个用户，骑手对于特定楼宇寻址特别困难，骑手在交付楼宇附近只能步行，老旧小区没有电梯，写字楼无法上楼，或者难

以等到电梯等等。交付时间预估需要具备刻画交付难度的能力，在定价、调度等多个场景中被广泛使用。例如根据交付难度来确定是否调节骑手邮资，根据交付难度来确定是否调节配送运单的顺序，从而避免超时等等。总的来说，交付时间预估是配送业务基础服务的重要一环。

但是，交付时间预估存在如下的困难：

- 输入信息较少，且多为非数值型数据，目前能够被用来预估的仅有如下维度特征：交付地址、交付点的经纬度、区域、城市，适配常规机器学习模型需要重新整理且容易丢失信息。
- 计算性能要求很高。由于是基础服务，会被大量的服务调用，需要性能 TP99 保证在 10ms 以内，整个算法平均响应时间需要控制在 5ms 内，其中包括数据处理及 RPC 的时间。且该标准为 CPU 环境下的性能要求，而非 GPU 下的性能要求。

	Name	Total	Failure	Failure%	Log View	Max	Avg	90Line	95Line	99Line	99.9Line
[:: show ::]	TOTAL	1,225,797	0	0.0000%	L S	85.0	3.8	0.0	0.0	0.0	0.0
[:: show ::]	V3:1	595,184	0	0.0000%	L S	80.0	4.3	8.1	8.8	10.1	23.7
[:: show ::]	V3P2:1	397,471	0	0.0000%	L S	85.0	3.7	5.0	5.1	6.0	18.1
[:: show ::]	V2:1	233,142	0	0.0000%	L S	45.0	2.5	3.4	3.7	4.0	6.6

上图为部分版本所对应的性能，平响时间均在 5ms 内，TP99 基本在 10ms 内

总结起来，交付时间预估的问题，在于需要使用轻量级的解决方案来处理多种数据形式的非数值型数据，并提取有效信息量，得到相对准确的结果。在相同效果的前提下，我们更倾向于性能更优的方案。

在本文中，我们介绍了交付时间预估迭代的三个版本，分别为基于地址结构的树模型、向量召回方案以及轻量级的 End-to-End 的深度学习网络。同时介绍了如何在性能和指标之间取舍，以及模型策略迭代的中间历程，希望能给从事相关工作的同学们有所启发和帮助。

2. 技术迭代路径

首先，在交付时间预估的技术迭代上，我们主要经历了三个大版本的改动，每一

版本在 5ms 计算性能的约束下，追求轻量化的解决方案，在兼顾提升效果的基础上，不显著增加性能的消耗。

本章节分别叙述了 3 个模型的迭代路径，包括技术选型、关键方案及最终效果。

2.1 树模型

技术选型

最早也是最容易被考虑到的是利用规则，核心思路是利用树结构衡量地址相似性，尽可能在相似的交付地址上积聚结构化数据，然后利用局部的回归策略，得到相对充裕的回归逻辑，而未能达到回归策略要求的则走兜底的策略。

为了快速聚积局部数据，树模型是一个较为合适的解决方案，树的规则解析能够有效地聚集数据，同时一个层级并不深的树，在计算速度上，具备足够的优势，能够在较短的时间内，得到相对不错的解决方案。

观察用户填写地址以及联系实际中地址的层级结构，不难发现，一个地址可以由四级结构组成：地址主干词（addr）、楼宇号（building）、单元号（unit）、楼层（floor）。其中的地址主干词在实际中可能对应于小区名或者学校名等地标名称。例如望京花园 1 号楼 2 单元 5 楼，解析为（望京花园，1 号楼，2 单元，5 楼）。通过分析，实际交付时长与楼层高低呈正相关关系，且不同交付地址的交付时长随楼层增加的变化幅度也有所区别，所以可以使用线性回归模型拟合楼层信息和交付时长的关系，而地址主干词、楼宇号、单元号作为其层级索引。但用户填写的地址中并不一定包含完整的四级结构，就会存在一定比例的缺失，所以利用这样的层级结构构建成一棵树，然后充分利用上一层已知的信息进行预估。预测时，只需根据结点的分支找到对应的模型即可，如果缺失，使用上一层结构进行预测。对于没有达到训练模型要求数据量的地址，使用其所在的区域平均交付时长作为交付时长的预估结果，这部分也可以看作区域信息，作为树结构的根节点。

迭代路径

整体的思路是基于离散特征训练树模型，在树的结点上基于楼层训练线性回归模型。树结点训练分裂规则：（1）数据量大于阈值；（2）分裂后 MAE（平均绝对误差）的

和小于分裂前。考虑到数据的时效性，采用加权线性回归增加近期数据的权重。

2.2 树模型 + 向量召回方案

技术选型

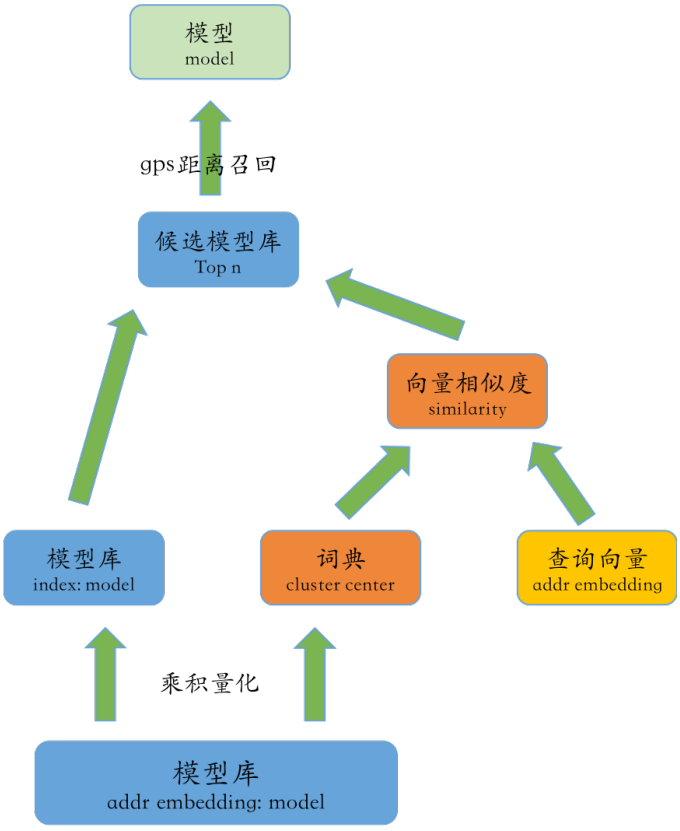
向量召回作为主流的召回方案之一，被业界广泛使用，在使用 LSH、PQ 乘积量化等常用开源工具基础上，高维向量召回性能通常在毫秒量级。

而从算法上考虑，树模型中 NLP 地址解析结果能够达到模型使用要求的仅为 70%+，剩余 20%+ 的地址无法通过训练得到的模型从而只能走降级策略。利用高维向量来表达语义相似性，即利用向量来表达地址相似性，从而用相似数据对应的模型来替代相似但未被召回数据，将地址主干词进行 Embedding 后，摆脱主干词完全匹配的低鲁棒性。

例如，在地址上可能会出现【7 天酒店晋阳街店】数据量比较充足，但【7 天连锁酒店太原高新区晋阳街店】数据量不充足从而无法训练模型的案例，这可能是同一个交付位置。我们希望尽可能扩大地址解析的成功率。

迭代路径

整个技术路径较为清晰简单，即利用 Word2Vec 将 charLevel 字符进行 Embedding，获得该地址的向量表示，并且融入 GPS 位置信息，设计相应兜底策略。



向量召回方案决策路径

最终效果

比较大地提升了整体策略的召回率，提升了 12.20pp，对于未被上一版本树模型召回的地址，指标有了显著的提升，其中 ME 下降 87.14s，MAE 下降 38.13s，1min 绝对偏差率减小 14.01pp，2min 绝对偏差率减小 18.45pp，3min 绝对偏差率减小 15.90pp。

2.3 End-to-End 轻量化深度学习方案

技术选型

在树模型的基础上，迭代到向量召回方案，整个模型的召回率有了较大幅度的增长，但仍然不是 100%。分析发现，召回率提升的障碍在于 NLP 对于地址解析的覆盖率。

整个方案的出发点：

从模型复杂度考虑，同样仅仅使用地址信息的话，在提升模型 VC 维的基础上，使用其他的模型方案至少可以持平树模型的效果，如果在这基础上还能融入其他信息，那么对于原模型的基线，还能有进一步的提升。

考虑到不仅仅需要使用地址数据，同时需要使用 GPS 数据、大量 ID 类的 Embedding，对于各类非数值类型的处理灵活性考虑，采用深度学习的方案，来保证多源且多类型特征能在同一个优化体系下优化学习。

工程上需要考虑的点：

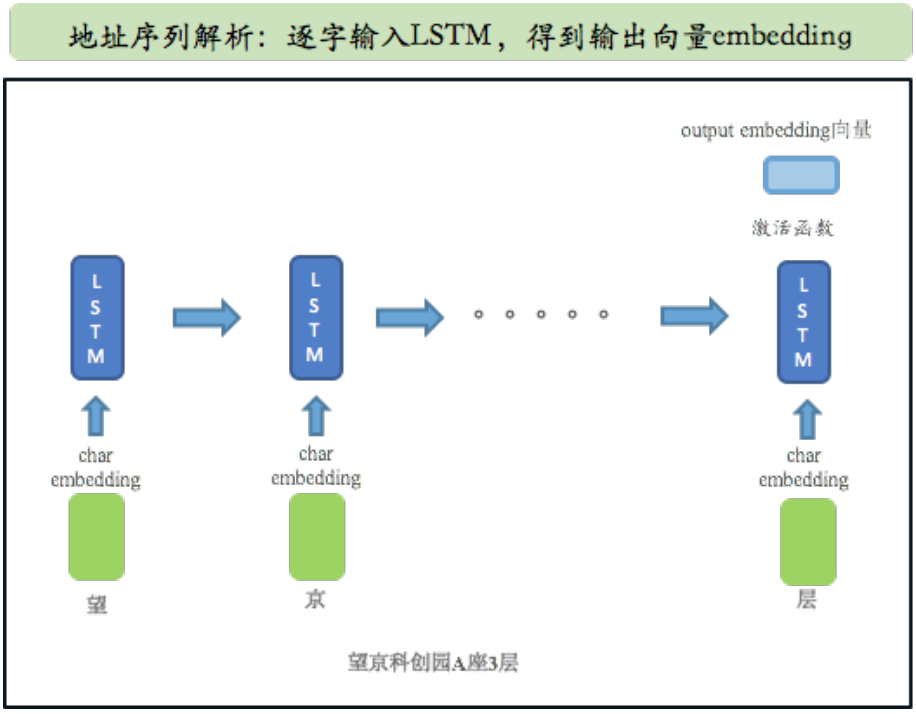
交付模型作为基础模型，被广泛应用在路径构造、定价、ETA 等各个业务中频繁调用，在树模型版本中，对于性能的要求为平均响应时间 5ms，TP99 在 10ms 左右，本方案需要考虑沿袭原业务的性能，不能显著增加计算耗时。

交付模型的难点在于非数值型特征多，信息获取形式的多样化，当前的瓶颈并不在于模型的复杂度低。如果可以轻量地获取信息及融合，没必要对 Fusion 后的信息做较重的处理方案。

所以整体的设计思路为：利用深度学习融合非数值型特征，在简单 Fusion 的基础上，直接得到输出结构，对于组件的选择，尽可能选用 Flops 较低的设计。该设计背后意图是，在充分使用原始输入信息，在尽可能避免信息损失的基础上，将非数值型的信息融入进去。并将信息充分融合，直接对接所需要的目标。而选用的融合组件结构尽可能保证高性能，且具备较高学习效率。这里分别针对地址选用了较为 Robust 的 LSTM，针对 GPS 选用了自定义的双线性 Embedding，兼顾性能和效果。

迭代路径

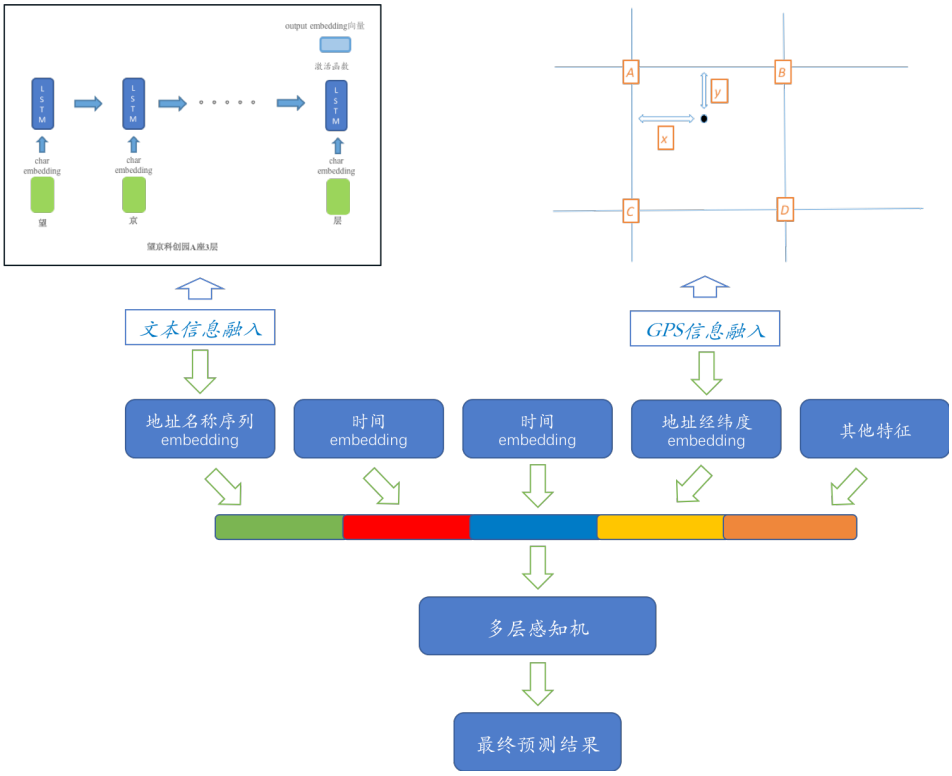
开始采用端到端的深度学习模型，这里首先需要解决的是覆盖率问题，直接采用 LSTM 读取 charLevel 的地址数据，经过全连接层直接输出交付时间。作为第一版本的数据，该版本数据基本持平树模型效果，但对于树模型未召回的 20% 数据，有了较大的提升。



地址信息输入 charLevel 模型

在采用 charLevel 的地址奏效后，我们开始采用加入用户地址 GPS 的信息，由于 GPS 为经纬度信息，非数值型数据，我们使用一种基于地理位置格点的双线性插值方法进行 Embedding。该方案具备一定的扩展性，对不同的 GPS 均能合理得到 Embedding 向量，同时具备平滑特性，对于多对偏移较小的 GPS 点能够很好的进行支持。

最终方案将地址 Embedding 后，以及 GPS 点的 Embedding 化后，加入下单时间、城市 ID、区域 ID 等特征后，再进行特征融合及变换，得到交付模型的时间预估输出。整个模型是一个端到端的训练，所有参数均为 Trainable。



模型结构示意图

扩展组件

在证实 End-to-End 路径可行后，我们开始进行扩展组件建设，包括自定义损失函数、数据采样修正、全国模型统一等操作，得到一系列正向效果，并开发上线。

特征重要性分析

对于深度学习模型，我们有一系列特征重要性评估方案，这里采用依次进行 Feature Permutation 的方式，作为评估模型特征重要性的方式。

考虑 GPS 经纬度和用户地址存在较大程度的信息重叠，评估结果如下。Shuffle 后，用户地址的特征重要性高于 GPS 经纬度的特征重要性。加入 GPS 后 ME 下降不如地址信息明显，主要是地址信息包含一定冗余信息（下文会分析），而其他信息的影响则可以忽略不计。

特征	特征重要排名
用户地址	1
GPS 经纬度	2
其他特征	……

注：在配送的其他案例中，商户 GPS 的经纬度重要性 >> 用户地址重要性 >> 用户 GPS 的经纬度重要性，该特征重要性仅仅为本案例特征重要性排序，不同学习目标下可能会有比较明显差别。

最终效果

End-to-End 深度学习模型的最终效果较为显著：对于树模型及向量召回方案的最痛点，覆盖率得到彻底解决，覆盖率提升到 100%。ME 下降 4.96s，MAE 下降 8.17s，1min 绝对偏差率减小 2.38pp，2min 绝对偏差率减小 5.08pp，3min 绝对偏差率减小 3.46pp。同时，对于之前树模型及向量召回方案未能覆盖到的运单，提升则更为明显。

3. 模型相关分析

在整个技术迭代的过程中，由于整个解决方案对于性能有着较为苛刻的要求，需要单独对方案性能进行分析。本章节对向量召回方案及深度学习方案进行了相应的性能分析，以便在线下确认性能指标，最终保证上线后性能均达到要求。下文分别着重介绍了向量匹配的工具 Faiss 以及 TensorFlow Operation 算子的选取，还有对于整体性能的影响。

同时对比 End-to-End 生成向量与 Word2Vec 生成向量的质量区别，对于相关项目具备一定的借鉴意义。

3.1 向量召回性能

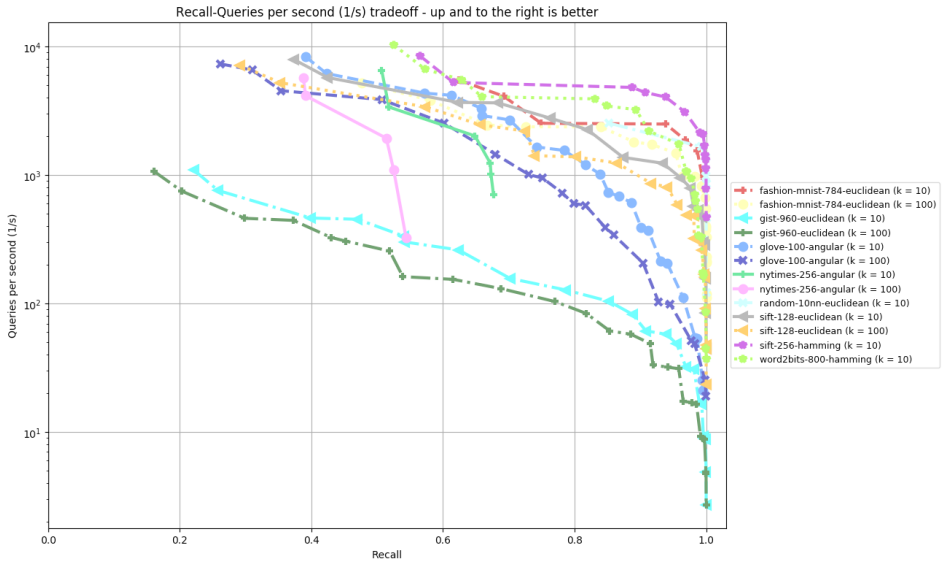
最近邻搜索（Nearest Neighbor Search）指的是在高维度空间内找到与查询点最近点的问题。在数据样本小的时候，通过线性搜索就能满足需求，但随着数据量的增加，如达到上百万、上亿点时候，倾向于将数据结构化表示来更加精确地表达向量信息。

此时近似最近邻搜索 ANN（Approximate Nearest Neighbor）是一个可参考的技术，它能在近似召回一部分之后，再进行线性搜索，平衡效率和精度。目前大体上有以下 3 类主流方法：基于树的方法，如 K-D 树等；基于哈希的方法，例如 LSH；基于矢量量化的方法，例如 PQ 乘积量化。在工业检索系统中，乘积量化是使用较多的一种索引方法。

针对向量召回的工具，存在大量的开源实现，在技术选型的过程中，我们参照 ANN-Benchmarks 以及 Erikbern/ANN-Benchmarks 中的性能评测结果。在众多 ANN 相关的工具包内，考虑到性能、内存、召回精度等因素，同时可以支持 GPU，在向量召回方案的测试中，选择以 Faiss 作为 Benchmark。

Faiss 是 FaceBook 在 2017 年开源的一个用于稠密向量高效相似性搜索和密集向量聚类的库，能够在给定内存使用下，在速度和精度之间权衡。可以在提供多种检索方式的同时，具备 C++/Python 等多个接口，也对大部分算法支持 GPU 实现。

下图为 Faiss 测评曲线：



交付时间模型召回的性能测试如下，可以达到性能需求。

- 召回候选集数量：8W 条向量【由于采用了 GPS 距离作为距离限制，故召回

测试采用 8W 数量级】

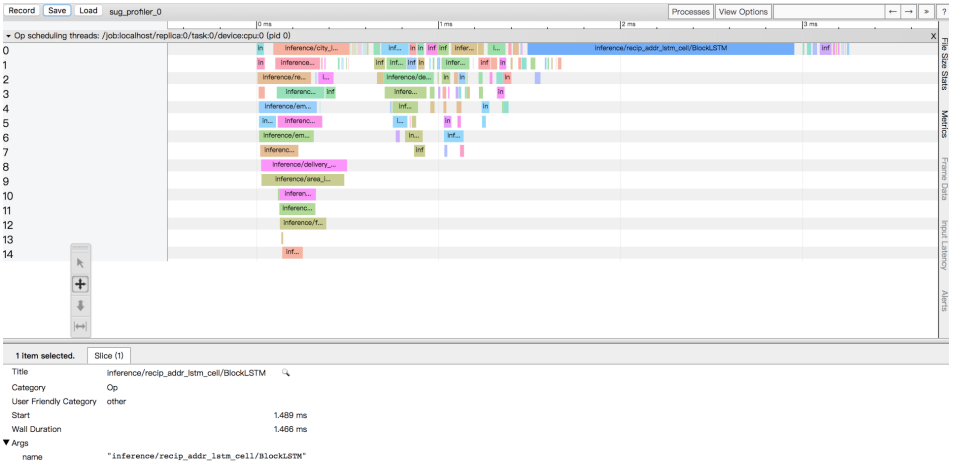
- 测试机器：Mac 本机 CPU【CPU 已满足性能，故不再测试 GPU】

单位 (ms)	最近邻数量	mean	90分位数	max	建索引
IndexFlatL2 dimention=128	1	2.63	2.73	10.83	12.80
	100	2.66	2.69	63.82	
	1000	3.14	3.27	21.54	
IndexIVFFlat L2距离 nlist=100 dimention=128 nprobe=10	1	0.65	0.67	5.59	234.52
	100	0.70	0.72	5.72	
	1000	1.06	1.08	11.95	
IndexIVFFlat Cosine距离 nlist=100 dimention=128 nprobe=10	1	0.36	0.38	5.47	153.99
	100	0.39	0.40	14.41	
	1000	0.61	0.63	5.99	
IndexFlatIP dimention=128	1	2.37	2.20	9.45	17.53
	100	2.92	3.08	17.59	
	1000	3.25	3.29	28.12	

3.2 序列模块性能

在 TensorFlow 系统中，以 C API 为界限，将系统划分为【前端】和【后端】两个子系统，前端扮演 Client 角色，完成计算图的构造，然后由 Protobuf 发送给后端启动计算图计算。计算图的基础单元是 OP，代表的是某种操作的抽象。在 TensorFlow 中，考虑到实现的不同，不同 OP 算子的选择，对于计算性能具有较大影响。

为了评测深度学习交付模型的性能瓶颈，首先对整个模型进行 Profile，下图即为 Profile 后的 Timeline，其中整个计算大部分消耗在序列模块处理部分，即下图中的蓝色部分。故需要对序列模块的计算性能进行 OP 算子的加速。



考虑到序列处理的需求，评估使用了 LSTM/GRU/SRU 等模块，同时在 TensorFlow 中，LSTM 也存在多种实现形式，包括 BasicLSTMCell、LSTMCell、LSTMBlockCell、LSTMBlockFusedCell 和 CuDNNLSTM 等实现，由于整个交付模型运行在 CPU 上，故排除 CuDNNLSTM，同时设置了全连接层 FullyConnect 加入评估。

从评估中可以发现，全连接层速度最快，但是对于序列处理会损失 2.3pp 效果，其余的序列模型效果差异不大，但不同的 OP 实现对结果影响较大。原生的 BasicLSTM 性能较差，contrib 下的 LSTMBlockFusedCell 性能最好，GRU/SRU 在该场景下未取得显著优势。

这是 LSTMBlockFusedCell 的官方说明，其核心实现是将 LSTM 的 Loop 合并为一个 OP，调用时候整个 Timeline 上更为紧凑，同时节约时间和内存：

> This is an extremely efficient LSTM implementation, that uses a single TF op for the entire LSTM. It should be both faster and more memory-efficient than LSTMBlockCell defined above.

以下是序列模块的性能测试：

- 环境：Tensorflow1.10.0，CentOS 7。
- 测试方法：CPU inference 1000 次，取最长的地址序列，求平均时间。

- 结论：LSTMBlockFused 实现性能最佳。【FullyConnect 性能最快，但对性能有损失】

注：在评估中，不仅仅包括了序列模型，也包括了其他功能模块，故参数量及模型大小按照总体模型而言

lstm结构OP	时间(ms)	FLOPs	可训练参数量	模型大小(MB)	效果差异
Fully Connect	1.18	27.83M	7.00M	29.1	−2.3pp
SRU	4.00	27.96M	7.06M	29.4	差异不显著
GRU Block	3.64	28.02M	7.10M	29.6	差异不显著
GRU	4.44	28.02M	7.10M	29.6	差异不显著
LSTMBlockFused	2.48	28.09M	7.13M	29.7	差异不显著
LSTM Block	4.34	28.09M	7.13M	29.7	差异不显著
LSTM	4.85	28.09M	7.13M	29.7	差异不显著
BasicLSTM	4.92	28.09M	7.13M	29.7	差异不显著

3.3 向量效果分析

将向量召回与深度学习模型进行横向比较，二者中间过程均生成了高维向量。不难发现，二者具备一定的相似性，这里就引发了我们的思考：

相较于向量召回，深度学习模型带来的提升主要来自于哪里?

有监督的 lstm 学习到的 Embedding 向量与自监督的 Word2Vec 得到的向量在地址相似性计算中有多大差别，孰优孰劣?

首先，我们分析第一个问题，End-to-End 模型提升主要来自哪里?

ME	MAE	1min绝对偏差率	2min绝对偏差率	3min绝对偏差率
End-to-End 模型 − Word2Vec 模型	4.14	−0.45	−0.31%	0.05%

我们直接将 End-to-End 模型得到的 char embedding 抽取出来，直接放入到 Word2Vec 方案内，取代 Word2Vec 生成的 char embedding，再进行向量召回的评估。结果如下表所示，单独抽取出来的 char embedding 在向量召回方案中，表现与 Word2Vec 生成的向量基本一致，并没有明显的优势。

注：

- 1min 绝对偏差率定义：|pred−label|<=60s
- 2min 绝对偏差率定义：|pred−label|<=120s
- 3min 绝对偏差率定义：|pred−label|<=180s

此时的变量有 2 个方面：

- a）对于 charLevel 地址的学习结构不同，一个为 Word2Vec，一个为 LSTM
- b）输入信息的不同，Word2Vec 的信息输入仅仅为地址主干词，而 End−to−End 的信息输入则包括了地址主干词、地址附属信息、GPS 等其他信息。

注：

- 完整地址：卓玛护肤造型（洞庭湖店）（洞庭湖路与天山路交叉路口卓玛护肤造型）
- 地址主干词：卓玛护肤造型店
- 地址附属信息：（洞庭湖店）（洞庭湖路与天山路交叉路口卓玛护肤造型）

为了排除第二方面的因素，即 b 的因素，使用地址主干词作为输入，而不用地址附属信息和其他模型结构的输入，保持模型输入跟 Word2Vec 一致。在测试集上，模型的效果比完整地址有明显的下降，MAE 增大约 15s。同时将 char embedding 提取出来，取代 Word2Vec 方案的 char embedding，效果反而变差了。结合 2.3 节中的特征重要性，可知，深度学习模型带来的提升主要来自对地址中冗余信息（相较于向量召回）的利用，其次是多个新特征的加入。另外，对比两个 End−to−End 模型的效果，地址附属信息中也包含着对匹配地址有用的信息。

ME	MAE	1min绝对偏差率	2min绝对偏差率	3min绝对偏差率
End−to−End 模型 − Word2Vec 模型	−1.28	0.64	0.90%	0.85%

针对第二个问题，有监督的 End−to−End 学习到的 Embedding 向量，与自监督的 Word2Vec 得到的向量在地址相似性计算中有多大差别，孰优孰劣?

采用地址主干词代替完整地址，作为 End−to−End 模型的输入进行训练，其他

信息均保持不变。使用地址主干词训练得到的 Embedding 向量，套用到向量召回方案中。

从评估结果来看，对于不同的阈值，End-to-End 的表现差异相对 Word2Vec 较小。相同阈值下，End-to-End 召回率更高，但是效果不如 Word2Vec。

从相似计算结果看，End-to-End 模型会把一些语义不相关但是交付时间相近的地址，映射到同一个向量空间，而 Word2Vec 则是学习一个更通用的文本向量表示。

> 例如，以下两个交付地址会被认为向量距离相近，但事实上只是交付时间相近：南内环西街与西苑南路交叉口金昌盛国会 <=> 辰憬家园迎泽西大街西苑南路路口林香斋酒店

如果想要针对更为复杂的目标和引入更多信息，可以使用 End-to-End 框架；只是计算文本相似性，从实验结果看，Word2Vec 更好一些。同时，通过查看 Case 也可以发现，End-to-End 更关注结果相似性，从而召回一部分语义上完全不相关的向量。两个模型目标上的不同，从而导致了结果的差异。

4. 总结与展望

在本篇中，依次展示了在配送交付场景下的三次模型策略迭代过程，以及在较为苛刻性能要求限制下，如何用轻量化的方案不断提高召回率及效果。同时，对迭代过程中的性能进行简单的分析及衡量，这对相关的项目也具备一定的借鉴意义，最后对 Word2Vec 及 End-to-End 生成的向量进行了比较。

事实上，本文中提及的向量召回及深度学习融合非数值型特征的方案，已经在业界被广泛使用。但对于差异化的场景，本文仍具备一定的借鉴价值，特别是对于订单 – 骑手匹配、订单 – 订单匹配等非搜索推荐领域的场景化应用，以及 TF OP 算子的选用及分析、Embedding 生成方式带来的差异，希望能够给大家提供一些思路和启发。

5. 关联阅读

交付时间预估与 ETA 预估及配送其他业务关系：

- 交付时间预估是 ETA 预估中的重要一环，关于 ETA 预估，请参见《深度学习在美团配送 ETA 预估中的探索与实践》。
- 具体 ETA 在整个配送业务中的位置及配送业务的整体机器学习实践，请参看《机器学习在美团配送系统的实践：用技术还原真实世界》。

6. 作者简介

基泽，美团点评技术专家
闫聪，美团点评算法工程师

7. 招聘信息

美团配送 AI 团队支撑全球领先的即时配送网络——美团配送，并建立了行业领先的美团智能配送系统。美团配送 AI 团队主要来自一线互联网公司以及知名科研机构，研发实力和氛围卓越。目前美团配送业务仍处于快速发展期，新的场景、新的技术难题不断涌现，成长空间巨大。
美团配送 AI 团队现诚聘算法策略工程师、机器学习工程师和运筹优化工程师，欢迎有兴趣的小伙伴投递简历至：tech@meituan.com（邮件标题注明：美团配送 AI 团队）

ICDAR 2019 论文：自然场景文字定位技术详解

刘曦

自然场景文字定位是文字识别中非常重要的一部分。与通用的物体检测相比，文字定位更具挑战性，文字在长宽比、尺度和方向上有更大范围的变化。针对这些问题，本文介绍一种融合文字片段及金字塔网络的场景文字定位方法。该方法将特征金字塔机制应用到单步多框检测器以处理不同尺度文字，同时检测多个文字片段以及学习出文字片段之间 8-neighbor 连接关系，最后通过 8-neighbor 连接关系将文字片段连接起来，实现对不同方向和长宽比的文字定位。此外，针对文字通常较小特点，扩大检测网络中 backbone 模型深层特征图，以获得更好性能。

本文提出的方法已发表在文档分析与识别国际会议 ICDAR2019 (International Conference on Document Analysis and Recognition) 上，审稿人评论该方法为“As it is of more practical uses”，认可了它的实用性。

ICDAR 是由国际模式识别学会 (IAPR) 组织的专业会议之一，专注于文本领域的识别与应用。ICDAR 大会每两年举办一次，目前已发展成文字识别领域的旗舰学术会议。为了提高自然场景的文本检测和识别水平，国际文档分析和识别会议（ICDAR）于 2003 年设立了鲁棒文本阅读竞赛（“Robust Reading Competitions”）。至今已有来自 89 个国家的 3500 多支队伍参与。ICDAR 2019 将于今年 9 月 20-25 日在澳大利亚悉尼举办。 美团今年联合国内外知名科研机构和学者，提出了”中文门脸招牌文字识别”比赛（ICDAR 2019 Robust Reading Challenge onReading Chinese Text on Signboards）。

背景

自然场景图像中的文字识别已被广泛应用在现实生活中，例如拍照翻译，自动驾驶，图像检索和增强现实等，因此也有越来越多的专家学者对其进行研究。自然场景文字定位是指对场景图像中所有文本的精确定位，是自然场景文字识别中第一步也是最重要的一步。由于自然场景下文本颜色、大小、宽高比、字体、方向、光照条件和

背景等具有较大变化（如图 1），因此它是非常具有挑战性的。

图 1　自然场景文字图片

深度学习技术在物体识别和检测等计算机视觉任务方面已经取得了很大进展。许多最先进的基于卷积神经网络（CNN）的目标检测框架，如 Faster RCNN、SSD 和 FPN[1] 等，已被用来解决文本检测问题并且性能远超传统方法。

深度卷积神经网络是一个多层级网络结构，浅层特征图具有高分辨率及小感受野，深层特征图具有低分辨率及大感受野。具有小感受野的浅层特征点对于小目标比较敏感，适合于小目标检测，但是浅层特征具有较少的语义信息，与深层特征相比具有较弱的辨别力，导致小文本定位的性能较差。另一方面，场景文字总是具有夸张的长宽比（例如一个很长的英文单词或者一条中文长句）以及旋转角度（例如基于美学考虑），通用物体检测框架如 Faster RCNN 和 SSD 是无法回归较大长宽比的矩形和旋转矩形。

围绕上面描述的两个问题，本文主要做了以下事情：

1. 为了处理不同尺度的文本，借鉴特征金字塔网络思路，将具有较强判别能力的深层特征与浅层特征相结合，实现在各个层面都具有丰富语义的特征金字塔。另外，当较深层中的小对象丢失时，特征金字塔网络仍可能无法检测到小对象，深层的上下文信息无法增强浅层特征。我们额外扩大了深层的特征图，以更准确地识别小文本。
2. 我们不直接回归文本行，而是将文本行分解为较小的局部可检测的文字片段，并通过深度卷积网络进行学习，最后将所有文字片段连接起来生成最终的文本行。

现有方法

最新的基于深度神经网络的文本定位算法大致可以分为两大类：（1）基于分割的文本定位；（2）基于回归的文本定位。

（1）基于分割的文本定位

当前基于分割的文本定位方法大都受到完全卷积网络（FCN [2]）的启发。全卷积网络（FCN, fully convolutional network），是去除了全连接 (fc) 层的基础网络，最初是用于实现语义分割任务。由于 FCN 网络最后一层特征图的像素分辨率较高，而图文识别任务中需要依赖清晰的文字笔画来区分不同字符（特别是汉字），所以 FCN 网络很适合用来提取文本特征。当 FCN 被用于图文识别任务时，最后一层特征图中每个像素将被分成文字行（前景）和非文字行（背景）两个类别。

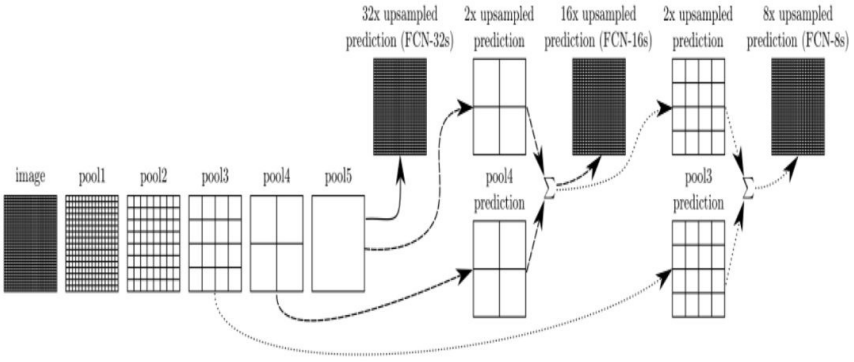


图 2　全卷积网络

（2）基于回归的文本定位

Textboxes [3] 是经典的也是最常用的基于回归的文本定位方法，它基于 SSD 框架，训练方式是端到端，运行速度也较快。为了适应文本行细长型特点，特征层也用长条形卷积核代替了其他模型中常见的正方形卷积核。为了防止漏检文本行，还在垂直方向增加了候选框数量。为了检测大小不同的字符块，在多个尺度的特征图上并行预测文本框，然后对预测结果做 NMS 过滤。

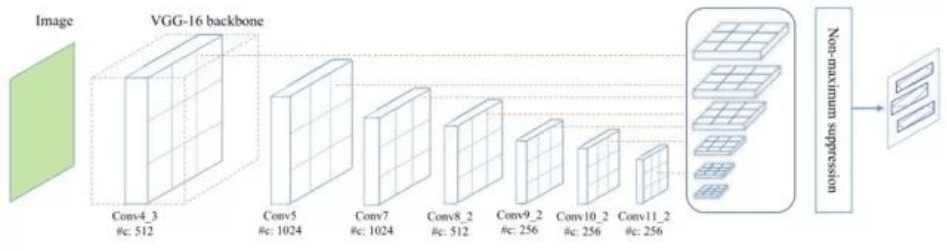


图 3　Textboxes 框架

提出方法

我们的方法也是基于 SSD，整体框架如图 4。为了应对多尺度文字尤其是小文字，对高层特征图进行间隔采样，以保持高层特征图分辨率。同时借鉴特征金字塔网络相关思路，将高层特征图上采样与底层特征叠加，构建一个新的多层级金字塔特征

图（图 4 蓝色框部分）。此外，为了处理各种方向文字，在不同尺度的特征图上预测文字片段以及片段之间的连接关系，然后对预测出的文字片段和连接关系进行组合，得到最终文本框。下面将具体介绍方法。

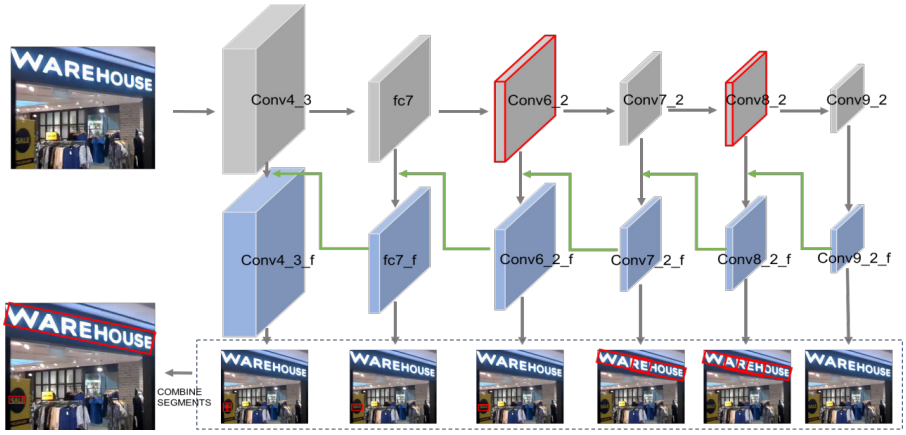


图 4　我们方法框架

（1）扩大高层特征图

深度卷积神经网络通常是逐层下采样，这对于物体分类来说是有效的，但是对于检测任务来说是有损害的。基于时间和性能的权衡考量，我们对卷积网络中最后几层特征进行间隔采样，如图 5，从 Conv6_2 层开始下采样，Conv7_2 层保持原分辨率，Conv8_2 层再下采样。

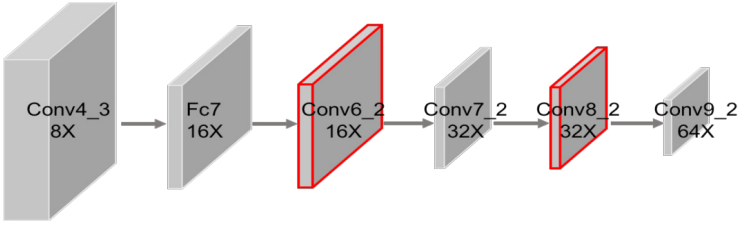


图 5　扩大特征图

（2）构建特征金字塔

虽然通过扩大深度特征图的设计可以更好地检测小文本，但较小的文本仍然难

以检测。为了更好地检测较小的文本，进一步增强较浅层（例如图 5 中 conv4_3，Fc7）的特征。我们通过融合高层和低层的特征构建了一个新的特征金字塔（图 4 中蓝色部分：conv4_3_f,fc7_f,conv6_2_f,conv7_2_f,conv8_2_f 和 conv9_2_f），新的金字塔特征具有更强辨别力和语义丰富性。

高层和低层特征融合策略如图 6 所示，高层特征图先进行上采样使之与低层特征图相同大小，然后与低层特征图进行叠加，叠加后的特征图再连接一个 3*3 卷积，获得固定维度的特征图，我们设定固定维度 d=256。

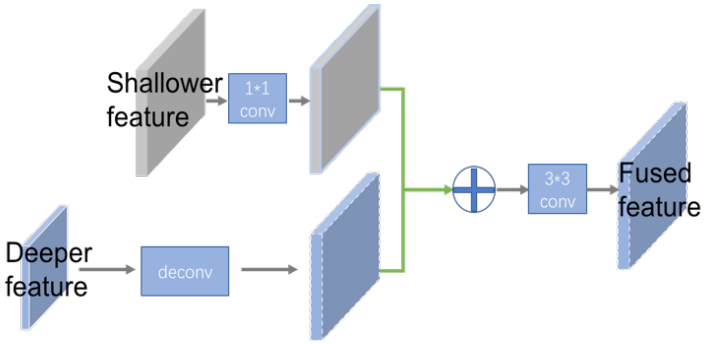


图 6　构建特征金字塔模块

（3）预测文字片段及片段之间连接关系

如图 7，先将每个文字词切割为更易检测的有方向的小文字块（segment），然后用邻近连接（link）将各个小文字块连接成词。这种方案方便于识别长度变化范围很大的、带方向的词和文本行，它不会象 Faster-RCNN 等方案因为候选框长宽比例原因检测不出长文本行，而且处理速度很快。

图 7　小文字块和近邻连接

基于第（2）小节构建的特征金字塔特征图，将每层特征图上特征点用于检测小文字块和文字块连接关系。如图 8，连接关系可以分为八种，上、下、左、右、左上、右上、左下、右下，同一层特征图、或者相邻层特征图上的小文字块都有可能被连接入同一个词中，换句话说，位置邻近、并且尺寸接近的文字块都有可能被预测到同一词中。

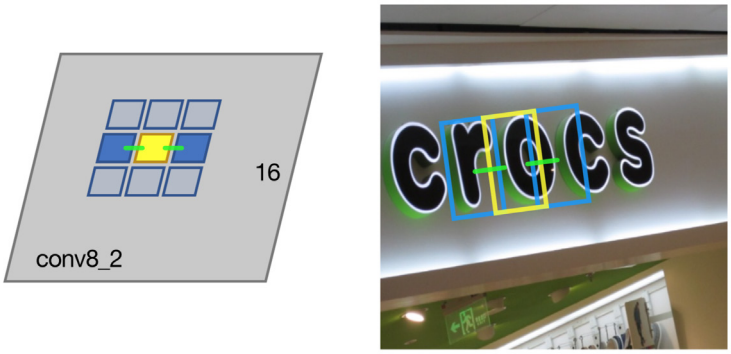


图 8　连接关系示意图

最后基于检测出的小文字块以及文字块连接，组合出文本框（如图 9），具体组合过程如下：

（a）将所有具有连接关系的小文字块组合起来，得到若干小文字块组；（b）对于每组小文字块，找到一条直线能最好的拟合组内所有小文字块中心点；（c）将组内所有小文字块的中心点投影到该直线上，找出距离最远的两个中心点 A 和 B；（d）最终文字框中心点为（A+B）/2，方向为直线斜率，宽度为 A，B 两点直线距离加上 A，B 两点的平均宽度，高度为所有小文字块的平均高度。

图 9　小文字块连接示意图

实验及应用

我们在两个公开数据集上（ICDAR2013，ICDAR2015）对方法进行评测。其中 ICDAR2013 数据集，训练图片 229 张，测试图片 233 张；ICDAR2015 数据集，训练图片 1000 张，测试图片 500 张，它们都来自于自然场景下相机拍摄的图片。

（1）我们首先对比了扩大高层特征图与不扩大高层特征图的性能比较，并在基础上对比加入特征金字塔后的性能比较，在 ICDAR2015 数据集上实验，结果如表 1：

表 1　方法中不同模块有效性验证

Method	Precision	Recall	F-measure
Baseline	79.5	73.4	76.3
扩大高层特征图	81	76.3	78.6
金字塔+扩大高层特征图	88	76.8	82

“baseline”方法是 ssd 框架 + 预测文字片段及片段之间连接关系模块，“扩大高层特征图”是在 baseline 方法基础上对高层特征图进行扩大，“金字塔 + 扩大高层特征图”是在 baseline 方法基础上对高层特征图进行扩大 并且加入特征金字塔。从表 1 中不难发现，扩大高层特征图可以带来精度和召回的提升，尤其是召回有近 3 个点的提升（73.4->76.3），这很好理解，因为更大的特征图产生更多的特征点以及预测结果；在此基础上再加入金字塔机制，精度获得显著提升，说明金字塔结构极大增强低层特征判别能力。

（2）我们也和其他方法也做了比较，具体见表 2 和表 3：

表 2　ICDAR2013 数据集与其他方法比较

Method	Precision	Recall	F-measure
CTPN	93	83	87.7
TextBoxes++	88	74	81
PixelLink	84.4	82.3	83.3
SegLink	87.7	83.0	85.3
OUR	92.4	83.8	87.9

表 3　ICDAR2015 数据集与其他方法比较

Method	Precision	Recall	F	FPS
SegLink	73.1	76.8	75	-
East	80.5	72.8	76.4	6.5
RRPN	82.2	73.2	77.4	-
TextBoxes++	87.2	76.7	81.7	11.6
PixelLink	82.9	81.7	82.3	7.3
OUR	88	76.8	82	10.3

从上表中可以看出，我们的方法在时间和精度上取得很好的权衡。在 ICDAR2015 数据集上，虽然性能不及 PixelLink，但是 FPS 要远高于它；而相比 TextBoxes++，虽然 FPS 略低于它，但是精度更高。图 10 给出一些文字定位结果示例。

图 10　文字定位结果示意图

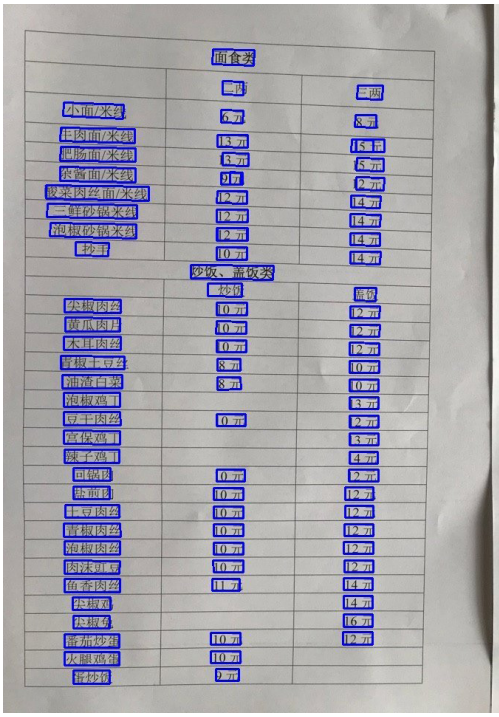
（3）此外，本方法也落地应用于实际业务场景菜单识别中。菜单上文字通常较小、较密，菜名文字可长可短，以及由于拍摄角度导致文字方向倾斜等。如图 11 所示，方法能很好的解决以上问题（小文字、密集文字行、长文本、不同方向）；并且在 500 张真实商家菜单图片上进行评测，相比 SegLink 方法，性能明显提升（近 5 个点提升）。

表 4　菜单测试结果

500菜单测试集	Precision	Recall	F-measure
改进前	85.1	84.7	84.9
改进后	89.5	90.3	89.9


白切鸡 45元/半只
白切鸡 85元/只
农家小炒肉 28元
凉瓜炒牛肉 28元
凉瓜炒蛋 18元
尖椒炒牛肉 28元
青瓜炒腰球 28元
菜心炒牛肉 28元
姜葱炒鳝片 28元
姜葱炒田鸡 28元
香辣牛肉 33元
香辣鸡杂 28元
红烧茄子 20元
韭黄炒蛋 18元
酸辣土豆丝 18元
豆豉鲮鱼炒凉瓜 18元
凉瓜炒咸蛋黄 18元


儿童套餐 19元
双人套餐 33元
全家福套餐 49元
单人套餐
吉祥二宝套餐
鸡腿堡+皇家鲜虾堡+皇家蟹黄堡 27元
阳光小吃组合
香辣炸翅+皇家鸡米花+热恋爆米花+可乐 25元


面食类
一两
三两
小面/米线 6元 8元
牛肉面/米线 13元 15元
肥肠面/米线 5元 5元
浓酱面/米线 9元 2元
酸菜肉丝面/米线 12元 14元
三鲜砂锅米线 12元 14元
泡椒砂锅米线 12元 14元
抄手 10元 14元
炒饭、盖饭类
炒饭
盖饭
尖椒肉丝 10元 12元
黄瓜肉片 10元 12元
木耳肉丝 10元 12元
青椒土豆丝 8元 10元
油渣白菜 8元 10元
泡椒鸡丁 3元
豆干肉丝 10元 2元
宫保鸡丁 3元
辣子鸡丁 4元
回锅肉 10元 2元
鱼香肉丝 10元 12元
土豆肉丝 10元 12元
青椒肉丝 10元 12元
泡椒肉丝 10元 12元
肉沫豆豉 10元 12元
鱼香肉丝 11元 14元
尖椒鸡 14元
尖椒鱼 16元
番茄炒蛋 10元 12元
火腿鸡蛋 10元
蛋炒饭 9元

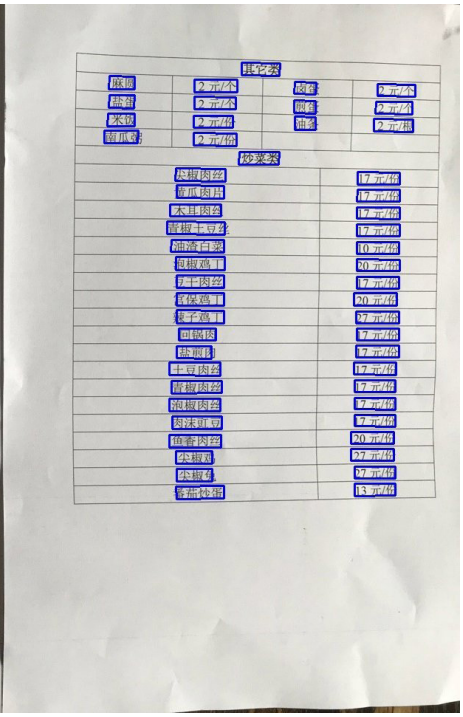

其它类
麻团 2元/个
盐蛋 2元/个
米饭 2元/份
酱爪爪 2元/份
凤爪 2元/个
鹌鹑蛋 2元/个
咖喱 2元/份
炒菜类
尖椒肉丝 17元/份
黄瓜肉片 17元/份
木耳肉丝 17元/份
青椒土豆丝 17元/份
油渣白菜 10元/份
泡椒鸡丁 20元/份
土豆肉丝 17元/份
宫保鸡丁 20元/份
辣子鸡丁 27元/份
回锅肉 17元/份
咕咾肉 17元/份
土豆肉丝 17元/份
青椒肉丝 17元/份
泡椒肉丝 17元/份
肉沫豆豉 17元/份
鱼香肉丝 20元/份
尖椒鸡 27元/份
尖椒鱼 27元/份
番茄炒蛋 13元/份

图 11　菜单文字定位结果示意图

结论

本文我们提出了一个高效的场景文本检测框架。针对文字特点，我们扩大高层特征图尺寸并构建了一个特征金字塔，以更适用于不同比例文本，同时通过检测文本片段和片段连接关系来处理长文本和定向文本。实验结果表明该框架快速且准确，在 ICDAR2013 和 ICDAR2015 数据集上获得了不错结果，同时应用到公司实际业务场景菜单识别上，获得明显性能提升。下一步，受实例分割的方法 PixelLink [4] 的启发，我们也考虑将文本片段进一步细化到像素级，同时融合检测和分割方法各自优缺点，构建联合检测和分割的文字定位框架。

参考文献

Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, Serge Belongie. “Feature Pyramid Networks for Object Detection.” arXiv preprint. arXiv: 1612.03144, 2017.

J. Long, E. Shelhamer, and T. Darrell. “Fully convolutional networks for semantic segmentation.” In CVPR, 2015.

M. Liao, B. Shi, and X. Bai. “Textboxes++: A single-shot oriented scene text detector.” IEEE Trans. on Image Processing, vol. 27, no. 8, 2018.
D. Deng, H. Liu, X. Li, and D. Cai. “Pixellink: Detecting scene text via instance segmentation.” In AAAI, pages 6773 - 6780, 2018.

作者简介

刘曦，美团视觉图像中心文字识别组算法专家。

招聘信息

美团视觉图像中心文字识别组：针对美团各项业务如商家入驻资质审核、网页信息合规审核等需求，对证照、票据、菜单、网图等图片类型开展文字识别研发工作。利用高性能文字识别功能，帮助业务方和商家实现自动审核 、自动录入，显著提升人效、降低成本，改善体验。

欢迎计算机视觉相关及相关领域小伙伴加入我们，简历可发邮件至 tech@meituan.com（邮件标题注明：美团视觉图像中心文字识别组）。

CVPR 2019 轨迹预测竞赛冠军方法总结

李鑫

背景

CVPR 2019 是机器视觉方向最重要的学术会议，本届大会共吸引了来自全世界各地共计 5160 篇论文，共接收 1294 篇论文，投稿数量和接受数量都创下了历史新高，其中与自动驾驶相关的论文、项目和展商也是扎堆亮相，成为本次会议的“新宠”。

障碍物轨迹预测挑战赛（Trajectory Prediction Challenge）隶属于 CVPR 2019 Workshop on Autonomous Driving — Beyond Single Frame Perception（自动驾驶研讨会），由百度研究院机器人与自动驾驶实验室举办，侧重于自动驾驶中的多帧感知，预测和自动驾驶规划，旨在聚集来自学术界和工业界的研究人员和工程

师，讨论自动驾驶中的计算机视觉应用。美团无人配送与视觉团队此项比赛获得了第一名。

在该比赛中，参赛队伍需要根据每个障碍物过去 3 秒的运动轨迹，预测出它在未来 3 秒的轨迹。障碍物共有四种类型，包括行人、自行车、大型机动车、小型机动车。每种障碍物的轨迹用轨迹上的采样点来表示，采样的频率是 2 赫兹。美团的方法最终以 1.3425 的成绩取得该比赛的第一名，同时我们也在研讨会现场分享了算法和模型的思路。

赛题简介

轨迹预测竞赛数据来源于在北京搜集的包含复杂交通灯和路况的真实道路数据，用于竞赛的标注数据是基于摄像头数据和雷达数据人工标注而来，其中包含各种车辆、行人、自行车等机动车和非机动车。

训练数据：每个道路数据文件包含一分钟的障碍物数据，采样频率为每秒 2 赫兹，每行标注数据包含障碍物的 ID、类别、位置、大小、朝向信息。

测试数据：每个道路数据文件包含 3 秒的障碍物数据，采样频率为每秒 2 赫兹，目标是预测未来 3 秒的障碍物位置。

评价指标

平均位移误差：Average displacement error（ADE），每个预测位置和每个真值位置之间的平均欧式距离差值。

终点位移误差：Final displacement error（FDE），终点预测位置和终点真值位置之间的平均欧式距离差值。

由于该数据集包含不同类型的障碍物轨迹数据，所以采用根据类别加权求和的指标来进行评价。

$$WSADE = D_v \cdot ADE_{\mathrm{v}} + D_p \cdot ADE_{\mathrm{p}} + D_b \cdot ADE_{\mathrm{b}},$$

$$WSFDE = D_v \cdot FDE_{\mathrm{v}} + D_p \cdot FDE_{\mathrm{p}} + D_b \cdot FDE_{\mathrm{b}},$$

现有方法

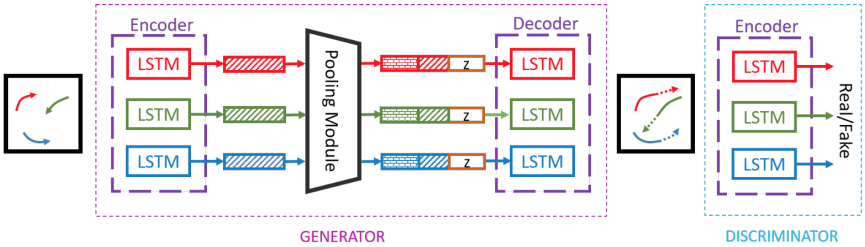
这次竞赛要解决的预测问题不依赖地图和其他交通信号等信息，属于基于非结构化数据预测问题，这类问题现在主流的方法主要根据交互性将其区分为两类：1. **独立预测**，2. **依赖预测**。

独立预测是只基于障碍物历史运动轨迹给出未来的行驶轨迹，依赖预测是会考虑当前帧和历史帧的所有障碍物的交互信息来预测所有障碍物未来的行为。

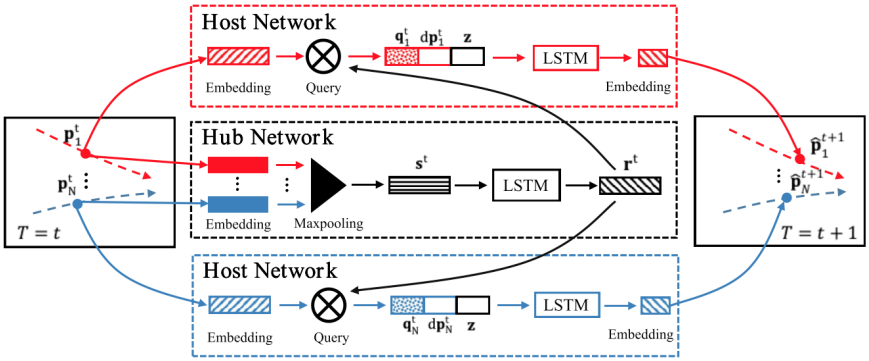
考虑交互信息的依赖预测，是当前学术界研究比较多的一类问题。但是经调研总结，我们发现其更多的是在研究单一类别的交互，比如在高速公路上都是车辆，那预测这些车辆之间的交互；再比如在人行道上预测行人的交互轨迹。预测所有类别障碍物的之间的交互的方法很少。

以下是做行人交互预测的两个方法模型：

方法 1. Social GAN，分别对每个障碍车输入进行 Encoder，然后通过一个统一的 Pooling 模块提取交互信息，再单独进行预测。



方法 2. StarNet，使用一个星型的 LSTM 网络，使用 Hub 网络提取所有障碍物的交互信息，然后再输出给每个 Host 网络独立预测每个障碍物的轨迹。

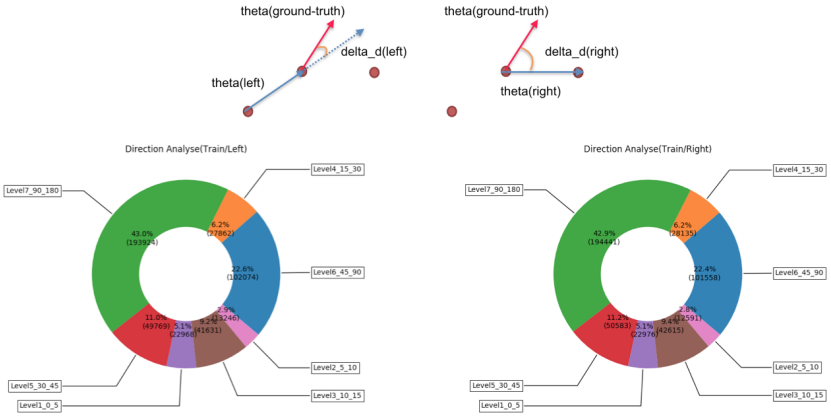


我们的方法

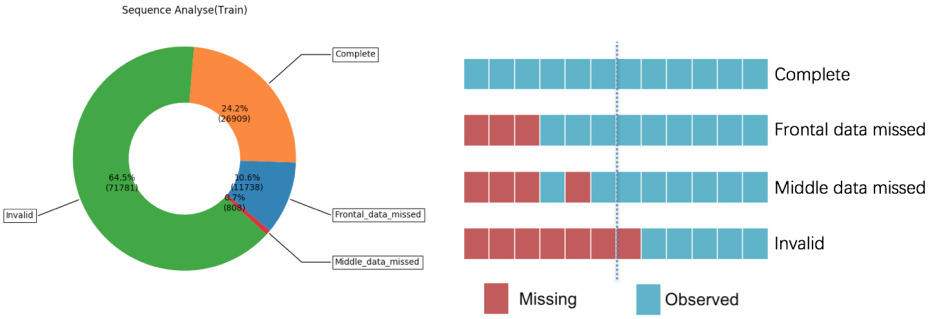
数据分析

拿到赛题之后，我们首先对训练数据做了分析，由于最终的目标是预测障碍物测位置，所以标注数据中的障碍物大小信息不太重要，只要根据类别来进行预测即可。

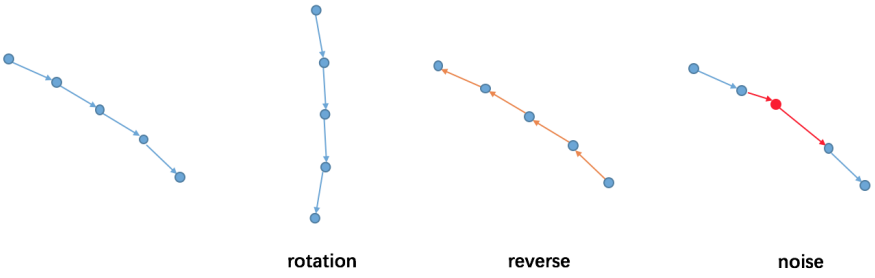
其次，分析朝向信息是否要使用，经统计发现真值标注的朝向信息非常不准确，从下图可以看到，大部分的标注方向信息都和轨迹方向有较大差距，因此决定不使用朝向信息进行预测。



然后，分析数据的完整性，在训练过程中每个障碍物需要 12 帧数据，才可以模拟测试过程中使用 6 帧数据来预测未来 6 帧的轨迹。但是在真实搜集数据的时候，没有办法保证数据的完整性，可能前后或中间都可能缺少数据，因此，我们根据前后帧的位置关系插值生成一些训练数据，以填补数据的缺失。



最后，对数据做了增强，由于我们的方法不考虑障碍物之间的交互，仅依赖每个障碍物自身的信息进行训练，因此障碍物轨迹进行了旋转、反向、噪声的处理。

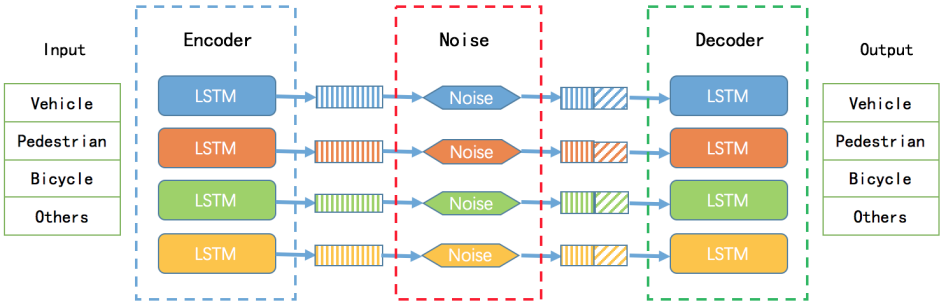


模型结构

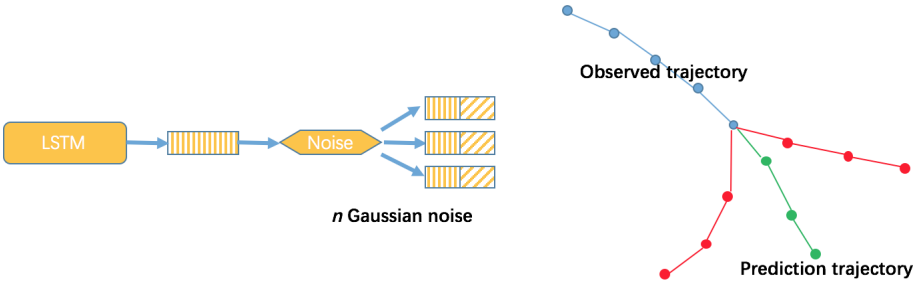
由于这次轨迹预测的问题是预测所有类别的轨迹，所以使用解决单一类别的轨迹预测模型不适用于该问题，而且如果把所有的物体放在单一的交互模型中来，不能正确提取出不同障碍物之间的交互特征。我们尝试了一些方法也证实了这一点。

因此在竞赛中，我们使用了多类别的独立预测方法，网络结构如下图，该方法针对每个类别构造一个 LSTM 的 Encoder-Decoder 模型，并且在 Encoder 和 Decoder 之间加入了 Noise 模块，Noise 模块生成固定维度的高斯噪声，将该噪声

和 Encoder 模块输出的 LSTM 状态量进行连结作为 Decoder 模块的 LSTM 初始状态量，Noise 模块主要作用是负责在多轮训练过程中增加数据的扰动，在推理过程中通过给不同的 Noise 输入，可以生成多个不同的轨迹。



最终，需要在不同的轨迹输出中选择一个最优的轨迹，这里采用了一个简单的规则，选择预测的轨迹方向和历史轨迹方向最接近的轨迹作为最终的轨迹输出。



实验结果

我们仅使用了官方提供的数据进行训练，按照前述数据增强方法先对数据进行增强，然后搭建网络结构进行训练，Loss 采用 Weighted Sum of ADE（WSADE），采用 Adam 优化方法，最终提交测试的 WSADE 结果为 1.3425。

方法	WSADE
我们的方法	1.3425
StarNet（基于交互的方法）	1.8626
TrafficPredict（ApolloScape Baseline 方法）	8.5881

总结

在这次竞赛中，我们尝试了使用多类别的独立预测方法，通过对数据增强和加入高斯噪声，以及最终人工设计规则选择最优轨迹的方法，在这次障碍物轨迹预测挑战赛（Trajectory Prediction Challenge）中获得了较好的成绩。但是，我们认为，基于交互的方法用的好的话应该会比这种独立预测方法还是要好，比如可以设计多类别内部交互和类别间的交互。另外，也关注到现在有一些基于图神经网络的方法也应用在轨迹预测上，今后会在实际的项目中尝试更多类似的方法，解决实际的预测问题。

参考文献

Yanliang Zhu, Deheng Qian, Dongchun Ren and Huaxia Xia. StarNet: Pedetrian Trajectory Prediction using Deep Neural Network in Star Topology[C]//Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). 2019.

Gupta A, Johnson J, Fei-Fei L, et al. Social gan: Socially acceptable trajectories with generative adversarial networks[C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR). 2018: 2255-2264.

Apolloscape. Trajectory dataset for urban traffic. 2018. http://apolloscape.auto/trajectory.html.

作者简介

李鑫，美团无人配送与视觉部 PNC 组轨迹预测组算法专家。

炎亮，美团无人配送与视觉部 PNC 组轨迹预测组算法工程师。

德恒，美团无人配送与视觉部 PNC 组轨迹预测组负责人。

冬淳，美团无人配送与视觉部 PNC 组负责人。

顶会论文：基于神经网络 StarNet 的行人轨迹交互预测算法

炎亮　德恒　冬淳　华夏

1. 背景

民以食为天，如何提升超大规模配送网络的整体配送效率，改善数亿消费者在”吃“方面的体验，是一项极具挑战的技术难题。面向未来，美团正在积极研发无人配送机器人，建立无人配送开放平台，与产学研各方共建无人配送创新生态，希望能在一个场景相对简单、操作高度重复的物流配送中，提高物流配送效率。在此过程中，美团无人配送团队也取得了一些技术层面的突破，比如基于神经网络 StarNet 的行人轨迹交互预测算法，论文已发表在 IROS 2019。IROS 的全称是 IEEE/RSJ International Conference on Intelligent Robots and Systems，IEEE 智能机器人与系统国际会议，它和 ICRA、RSS 并称为机器人领域三大国际顶会。

1.1　行人轨迹预测的意义

在无人车行驶过程中，它需要对周围的行人进行轨迹预测，这能帮助无人车更加安全平稳地行驶。我们可以用图 1 来说明预测周围行人的运动轨迹对于无人车行驶的重要性。

图 1 中蓝色方块代表无人车，白色代表行人。上半部分描述的是在不带行人轨迹预测功能情况下无人车的行为。这种情况下，无人车会把行人当做静态物体，但由于每个时刻行人都会运动，导致无人车规划出来的行驶轨迹会随着时间不停地变化，加大了控制的难度，同时还可能产生碰撞的风险，这样违背了安全平稳行驶的目标。下半部分是有了行人轨迹预测功能情况下的无人车行为。这种情况下，无人车会预测周围行人的行驶轨迹，因此在规划自身行驶时会考虑到未来时刻是否会与行人碰撞，最终规划出来的轨迹更具有“预见性”，所以避免了不必要的轨迹变化和碰撞风险。

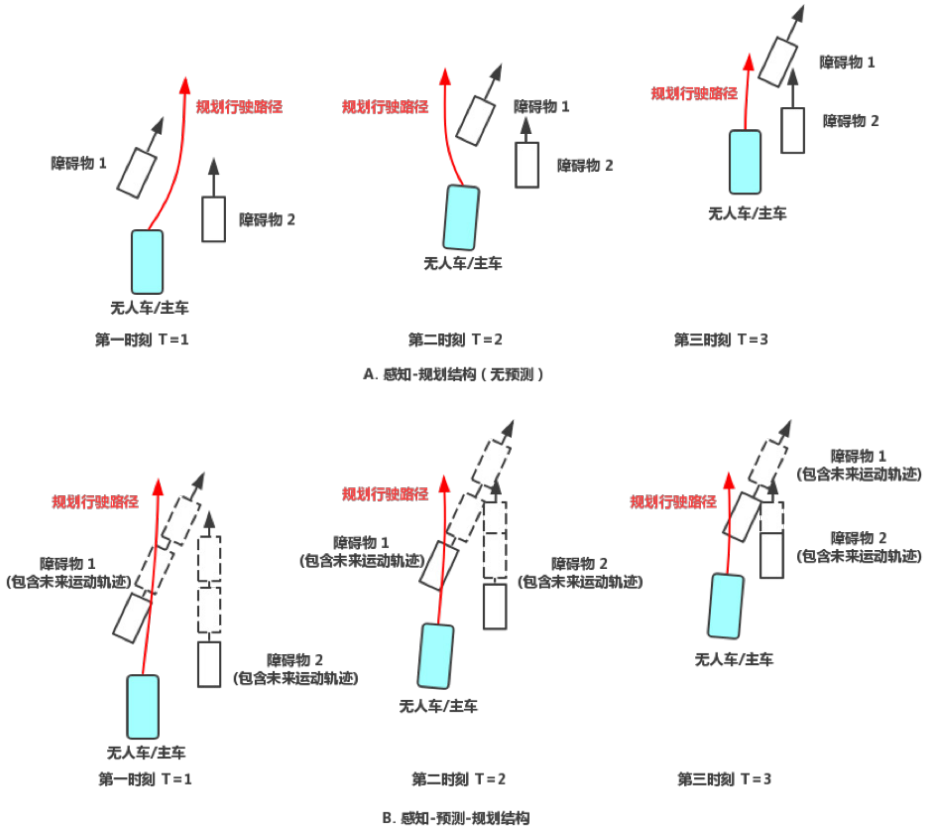


图 1　主车规划轨迹跳变问题

1.2　行人轨迹预测的难点

总体而言，行人轨迹预测的难点主要有两个：

第一，行人运动灵活，预测难度大。本身精确预测未来的运动轨迹是一个几乎不可能完成的任务，但是通过观察某个障碍物历史时刻的运动轨迹，可以根据一些算法来大致估计出未来的运动轨迹（最简单的是匀速直线运动）。在实际中，相比于自行车、汽车等模型，行人运动更加灵活，很难对行人建立合理的动力学模型（因为行人可以随时转弯、停止、运动等），这加剧了行人预测的难度。

第二，行人之间的交互，复杂又抽象。在实际场景中，某一行人未来的运动不仅受自己意图支配，同样也受周围行人的影响（例如避障）。这种交互非常抽象，在算法

中往往很难精确地建模出来。目前，大部分算法都是用相对空间关系来进行建模，例如相对位置、相对朝向、相对速度大小等。

1.3 相关工作介绍

传统算法在做预测工作时会使用一些跟踪的算法，最常见的是各类时序模型，例如卡尔曼滤波（Kalman Filter, KF）、隐马尔可夫（Hidden Markov Model, HMM）、高斯过程（Gaussian Process, GP）等。这类方法都有一个很明显的特点，就是根据历史时序数据，建立时序递推数学公式：$X^t = f\left(X^{t-1}\right)$或者$p\left(X^t \mid X^{t-1}\right)$。因为这类方法具有严格的数学证明和假设，也能处理一些常规的问题，但是对于一些复杂的问题就变得“束手无策”了。这是因为这些算法中都会引入一些先验假设，例如隐变量服从高斯分布，线性的状态转换方程以及观测方程等，而最终这些假设也限制了算法的整体性能。神经网络一般不需要假设固定的数学模型，凭借大规模的数据集促使网络学习更加合理的映射关系。本文我们主要介绍一些基于神经网络的行人预测算法。

基于神经网络的预测算法（主要以长短期记忆神经网络 Long Short Term Memory，LSTM 为主）在最近 5 年都比较流行，预测效果确实比传统算法好很多。在 CVPR（IEEE Conference on Computer Vision and Pattern Recognition）2019 上，仅行人预测算法的论文就有 10 篇左右。这里我们简单介绍 2 篇经典的行人预测算法思路，如果对这方面感兴趣的同学，可以通过文末的参考文献深入了解一下。第一篇是 CVPR 2016 斯坦福大学的工作 Social-LSTM，也是最经典的工作之一。Social-LSTM 为每个行人都配备一个 LSTM 网络预测其运动轨迹，同时提出了一个 Social Pooling Layer 的模块来计算周围其他行人对其的影响。具体的计算思路是将该行人周围的区域划分成 NxN 个网格，每个网络都是相同的大小，落入这些网格中的行人将会参与交互的计算。

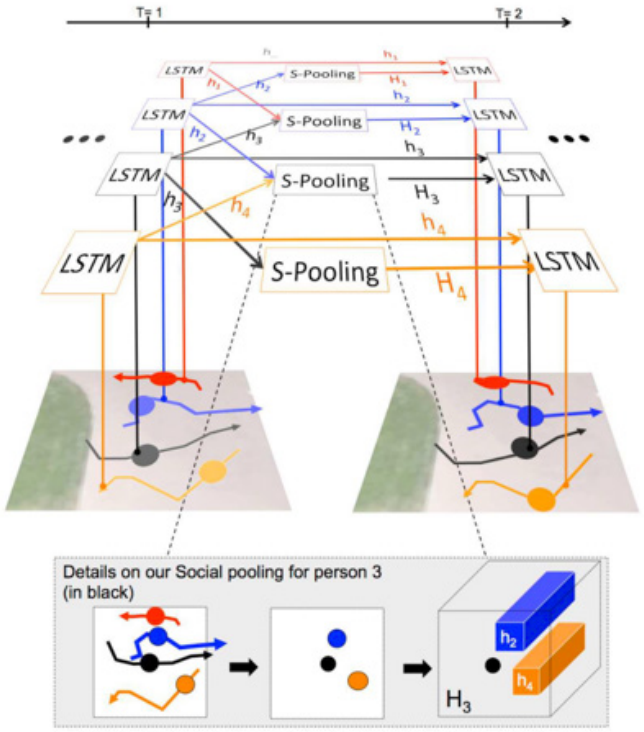


图 2　左：Social LSTM 原理 右：Social Pooling 计算过程

第二篇是 CVPR 2019 卡耐基梅隆大学 & 谷歌 & 斯坦福大学的工作，他们的工作同样使用 LSTM 来接收历史信息并预测行人的未来轨迹。不同于其他算法的地方在于，这个模型不仅接收待预测行人的历史位置信息，同时也提取行人外观、人体骨架、周围场景布局以及周围行人位置关系，通过增加输入信息提升预测性能。除了预测具体的轨迹，算法还会做粗粒度预测（决策预测），输出行人未来时刻可能所在的区域。

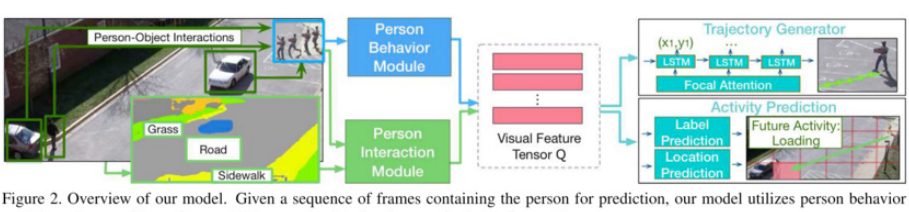


Figure 2. Overview of our model. Given a sequence of frames containing the person for prediction, our model utilizes person behavior module and person interaction module to encode rich visual semantics into a feature tensor.

图 3　算法整体结构

其他的相关工作，还包括基于语义图像 / 占有网格（Occupancy Grid Map, OGM）的预测算法，基于信息传递（Message Passing, MP）的预测算法，基于图网络（Graph Neural Network, GNN）的预测算法（GCN/GAT 等）等等。

2. StarNet 介绍

目前，现有的轨迹预测算法主要还是聚焦在对行人之间交互的建模，轨迹预测通常只使用 LSTM 预测即可。如下图 4 左，现有关于轨迹预测的相关工作基本都是考虑行人之间两两交互，很少有考虑所有行人之间的全局交互（即使是 GCN，也需要设计对应的相似矩阵来构造拉普拉斯矩阵，这也是一个难点）。我们可以举一个例子来说明现有其他算法预测的流程：

- 假设感知模块检测到当前 N 个行人的位置，如何计算第一个行人下一时刻的位置？ Step 1 计算其他人对于第一个行人的交互影响。将第 i 个行人在第 t 时刻的位置记为（一般是坐标 x 和 y）。可以通过以下公式计算第一个行人的交互向量：$Interaction_1^t = f\left(P_2^t - P_1^t, P_3^t - P_1^t, \cdots, P_N^t - P_1^t\right)$

  从上述公式可以大致看到，相对位置关系是最重要的计算指标，计算的函数 f 一般是一个神经网络。 Step 2 计算第一个行人下个时刻的位置。通常需要根据上一时刻的位置与交互向量：$P_1^{t+1} = g\left(P_1^t, Interaction_1^t\right)$ 上述公式中，计算的函数 g 同样是神经网络，即上面提到的长短期记忆神经网络 LSTM。

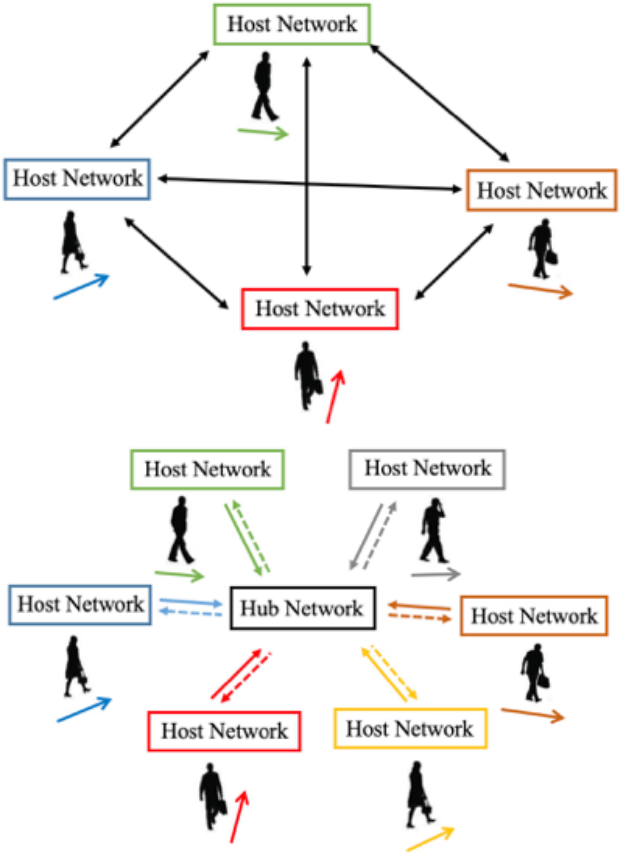


图 4　算法思路对比图 上：传统算法 下：StarNet

两两交互的方式存在两个问题：

1. 障碍物 2 和 3 确实会影响障碍物 1 的运动，但是障碍物 2 和 3 之间同样也存在相互影响，因此不能直接将其他障碍物对待预测障碍物的影响单独剥离出来考虑，这与实际情况不相符。
2. 两两计算消耗的资源大，如果有 N 个障碍物，那么两两交互就需要 N 的平方次计算，随着 N 的变大，计算量呈平方倍增长。我们希望障碍物之间的交互能否只计算 1 次而非 N 次，所有障碍物的轨迹预测都共享这个全局交互那就更好了。

基于上述两个问题，我们提出了一种新的模型，该模型旨在高效解决计算全局交互的问题。因为传统算法普遍存在计算两两交互的问题（即使是基于 Attention 注意力机制的 Message Passing 也很难考虑到全局的交互），本文想尝试通过一些更加简单直观的方式来考虑所有障碍物之间的全局交互，我们的算法大致思路如下：

每个时刻所有障碍物的位置可以构成一张静态的“地图”，随着时间的变化，这些静态地图就变成了一张带有时序信息的动态图。这张动态图中记录了每个区域内的障碍物运动信息，其中运动信息是由所有障碍物一起影响得到的，而非单独地两两交互形成。对于每个障碍物的预测阶段，只要根据该障碍物的位置，就可以在这张时序地图中查询该区域在历史时刻的障碍物运动信息（例如这个区域在历史时刻中，障碍物 1、2、4、5 都有其运动的轨迹）。通过“共享全局交互地图 + 个体查询”的方式，就可以做到计算全局交互以及压缩计算开销。

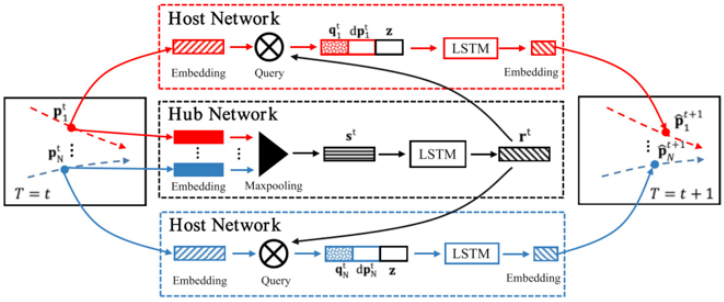


Fig. 2: The process of predicting the coordinates. At time step t, StarNet takes the newly observed (or predicted) coordinates $\{\mathbf{p}_i^t\}_{i=1}^N$ (or $\{\widehat{\mathbf{p}}_i^t\}_{i=1}^N$) and outputs the predicted coordinates $\{\widehat{\mathbf{p}}_i^{t+1}\}_{i=1}^N$.

图 5　StarNet 网络结构图

我们的算法结构如上图 5 所示，Host Network 是基于 LSTM 的轨迹预测网络；Hub Network 是基于 LSTM 的全局时序交互计算网络。在论文具体的实现中，首先 Hub Network 的静态地图模块是通过接受所有障碍物同一时刻的位置信息、全连接网络和最大池化操作得到一个定长的特征向量 s^t；然后动态地图模块使用 LSTM 网络对上述的特征向量 s^t 进行时序编码，最终得到一个全局交互向量 r^t。Host Network 首先根据行人（假设要预测第一个行人下时刻的位置）的位置 P_1^t 去动态地图 r^t 中查询自己当前位置区域内的交互 q_1^t，具体我们采用简单的点乘操作（类似于 Attention 机

制)。最终自己的位置 P_1^t 和交互 q_1^t 一起输入 LSTM 网络预测下时刻的的位置 P_1^{t+1} 。

实验阶段，我们与 4 种经典的算法作比较，使用的数据集为 UCY&ETH 数据集，这两个数据集包含 4 个子场景，分别为 ZARA-1/ZARA-2、UNIV、ETH、HOTEL。在预测过程中，所有算法根据每个行人过去 3.2 秒的运动轨迹，预测出它在未来 3.2 秒的轨迹。每 0.4 秒采样一个离散点，因此 3.2 秒的轨迹可以用 8 个轨迹离散点表示。对比的指标有：

(a) 平均距离差 ADE (Average Displacement Error)：用算法预测出的轨迹到真实轨迹所有 8 个点之间的平均距离差。(b) 终点距离差 FDE (Final Displacement Error)：用算法预测出的轨迹与真实轨迹最后一个终点之间的距离差。(c) 前向预测时间以及参数量。

最终的实验结果如下表：

TABLE II: Prediction Errors

Metric	Dataset	LSTM	Social LSTM	Social GAN	Social Attention	StarNet (Ours)
ADE	ZARA-1	0.25	0.27	**0.21**	1.66	0.25
	ZARA-2	0.31	0.33	0.27	2.30	**0.26**
	UNIV	0.36	0.41	0.36	2.92	**0.21**
	ETH	0.70	0.73	0.61	2.45	**0.31**
	HOTEL	0.55	0.49	0.48	2.19	**0.46**
Average ADE	-	0.43	0.45	0.39	2.30	**0.30**
Variance of ADE	-	0.028	0.026	0.021	0.166	**0.008**
FDE	ZARA-1	0.53	0.56	**0.42**	2.64	0.47
	ZARA-2	0.65	0.70	0.54	4.75	**0.53**
	UNIV	0.77	0.84	0.75	5.95	**0.40**
	ETH	1.45	1.48	1.22	5.78	**0.54**
	HOTEL	1.17	1.01	0.95	4.94	**0.91**
Average FDE	-	0.91	0.91	0.78	4.81	**0.57**
Variance of FDE	-	0.118	0.101	0.802	1.394	**0.031**

TABLE III: Computational Time

Metric	LSTM	Social LSTM	Social GAN	Social Attention	StarNet (Ours)
Inference Time (Seconds)	0.029	0.504	0.202	3.714	**0.073**
Number of Paramters (Kilo)	22.87	156.06	108.03	874.95	**31.90**

从实验结果可以看到，我们的算法在 80% 的场景下都优于其他算法，且实时性高(表中 LSTM 的推理时间为 0.029 秒，最快速是由于该算法不计算交互，因此速度最快参数也最少，但是性能较差)。

总结一下，我们提出算法 StarNet 的优势主要包括以下两点：

- 使用全局动态地图的形式来描述行人之间在时间和空间上的相互影响，更加合理，也更加准确。

- Hub Network 全局共享的特征提升了整个算法的计算效率。

3. 未来工作

首先，我们会进一步探索新的模型结构。虽然我们的算法在数据集上取得了不错的效果，但这是我们的第一次尝试，模型设计也比较简单，如果提升模型结构，相信可以取得更好的结果。

其次，我们会提升预测的可解释性。同现有算法一样，目前的模型对计算到的交互缺乏可解释性，仍然依赖于数据驱动。在今后的工作中，我们将通过对交互的可解释建模来提升预测的准确性。

最后，在构建时序的动态地图过程中，引入对于每个障碍物的跟踪信息。换句话说，我们知道每块区域在各个时间点障碍物的位置，但目前算法没有对障碍物在时序上做跟踪（例如时刻 1 有三个障碍物，时刻 2 三个障碍物运动了得到新的位置，网络输入为三个障碍物的位置信息，但是网络无法理解两个时刻中障碍物的对应关系，这降低了交互的性能），这点在以后的工作中还需要继续改进。

参考文献

[1] A. Alahi, K. Goel, V. Ramanathan, A. Robicquet, F. Li and S. Savarese, “Social lstm: Human trajectory prediction in crowded spaces,” in 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). IEEE 2016, pp. 961–971.

[2] H. Wu, Z. Chen, W. Sun, B. Zheng and W. Wang, “Modeling trajectories with recurrent neural networks,” in 28th International Joint Conference on Artificial Intelligence (IJCAI). 2017, pp. 3083–3090.

[3] A. Gupta, J. Johnson, F. Li, S. Savarese and A. Alahi, “Social GAN: Socially acceptable trajectories with generative adversarial networks,” in 2018 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). IEEE, 2018, pp. 2255–2264.

[4] A. Vemula, K. Muelling and J. Oh, “Social attention: Modeling attention in human crowds,” in 2018 IEEE International Conference on Robotics and Automation (ICRA). IEEE, 2018, pp. 1–7.

[5] Y. Xu, Z. Piao and S. Gao S, “Encoding crowd interaction with deep neural network for pPedestrian trajectory prediction,” in 2018 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). IEEE, 2018, pp. 5275–5284.

[6] D. Varshneya, G. Srinivasaraghavan, “Human trajectory prediction using spatially

aware deep attention models," arXiv preprint arXiv:1705.09436, 2017.
[7] T. Fernando, S. Denma, S. Sridharan and C. Fookes, "Soft+hardwired attention: An lstm framework for human trajectory prediction and abnormal event detection," arXiv preprint arXiv:1702.05552, 2017.
[8] J. Liang, L. Jiang, J. C. Niebles, A. Hauptmann and F. Li, "Peeking into the future: Predicting future person activities and locations in videos," in 2019 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). IEEE, 2019, pp. 5725-5734.
[9] A. Sadeghian, V. Kosaraju, Ali. Sadeghian, N. Hirose, S. H. Rezatofighi and S. Savarese, "SoPhie: An attentive GAN for predicting paths compliant to social and physical constraints," in 2019 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). IEEE, 2019, pp. 5725-5734.
[10] R. Chandra, U. Bhattacharya and A. Bera, "TraPHic: Trajectory prediction in dense and heterogeneous traffic using weighted interactions," in 2019 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). IEEE, 2019, pp. 8483-8492.
[11] J. Amirian, J. Hayet and J. Pettre, "Social Ways: Learning multi-modal distributions of pedestrian trajectories with GANs," arXiv preprint arXiv:1808.06601, 2018.

作者简介

朱炎亮，美团无人配送部
钱德恒，美团无人配送部
任冬淳，美团无人配送部
夏华夏，美团无人配送部

招聘信息

美团轨迹预测组招聘深度学习算法工程师，我们希望你：

- 具有扎实的编程能力，能够熟练使用 C++ 或 Python 作为编程语言。
- 具有深度学习相关知识，能熟练使用 TensorFlow 或 Pytorch 作为深度学习算法研发框架。
- 对预测无人车周围障碍物的未来轨迹感兴趣。

欢迎有兴趣的同学投送简历到 tech@meituan.com（邮件标题注明：美团轨迹预测组）。

大数据

基于大数据，可以更好的赋能业务、优化开发、反馈产品、助力运营。在美团点评，大数据应用于各业务线，以数据驱动的方式帮助公司实现业务目标，持续提高公司的运营效率和核心竞争力。同时，美团点评大数据不只做平台能力建设，更注重平台建设与平台能力在数据领域的实践，从而形成体系化的数据解决方案。

2019 年，美团技术博客发布的大数据相关文章涉及数据治理、资源优化、大数据平台能力建设、非固化的探索式数据分析领域。《Hadoop YARN：调度性能优化实践》和《将军令：数据安全平台建设实践》阐述了美团点评大数据平台能力的建设;《OneData 建设探索之路：SaaS 收银运营数仓建设》、《研发团队资源成本优化实践》、《Jupyter 在美团民宿的应用实践》三篇文章分别从数据治理、资源优化、非固化的探索式数据分析三个方面讲述了美团点评不同业务场景的应用实践。

抛砖引玉，希望美团点评的这些经验能够对数据工程师们有所启迪，未来一起交流、成长。

Hadoop YARN：调度性能优化实践

世龙　廷稳

背景

YARN 作为 Hadoop 的资源管理系统，负责 Hadoop 集群上计算资源的管理和作业调度。

美团的 YARN 以社区 2.7.1 版本为基础构建分支。目前在 YARN 上支撑离线业务、实时业务以及机器学习业务。

- 离线业务主要运行的是 Hive on MapReduce，Spark SQL 为主的数据仓库作业。
- 实时业务主要运行 Spark Streaming，Flink 为主的实时流计算作业。
- 机器学习业务主要运行 TensorFlow，MXNet，MLX（美团点评自研的大规模机器学习系统）等计算作业。

YARN 面临高可用、扩展性、稳定性的问题很多。其中扩展性上遇到的最严重的，是集群和业务规模增长带来的调度器性能问题。从业务角度来看，假设集群 1000 台节点，每个节点提供 100 个 CPU 的计算能力。每个任务使用 1 个 CPU，平均执行时间 1 分钟。集群在高峰期始终有超过 10 万 CPU 的资源需求。集群的调度器平均每分钟只能调度 5 万的任务。从分钟级别观察，集群资源使用率是 50000/(100*1000)=0.5，那么集群就有 50% 的计算资源因为调度能力的问题而无法使用。

随着集群规模扩大以及业务量的增长，集群调度能力会随着压力增加而逐渐下降。假设调度能力依然保持不变，每分钟调度 5 万个任务，按照 5000 台节点的规模计算，如果不做任何优化改进，那么集群资源使用率为：50000/(100*5000) = 10%，剩余的 90% 的机器资源无法被利用起来。

这个问题解决后，集群在有空余资源的情况下，作业资源需求可以快速得到满

足，集群的计算资源得到充分地利用。

下文会逐步将 Hadoop YARN 调度系统的核心模块展开说明，揭开上述性能问题的根本原因，提出系统化的解决方案，最终 Hadoop YARN 达到支撑单集群万级别节点，支持并发运行数万作业的调度能力。

整体架构

YARN 架构

YARN 负责作业资源调度，在集群中找到满足业务的资源，帮助作业启动任务，管理作业的生命周期。

YARN 详细的架构设计请参考 Hadoop 官方文档。

资源抽象

YARN 在 cpu，memory 这两个资源维度对集群资源做了抽象。

```
class Resource{
  int cpu;    //cpu 核心个数
  int memory-mb; // 内存的 MB 数
}
```

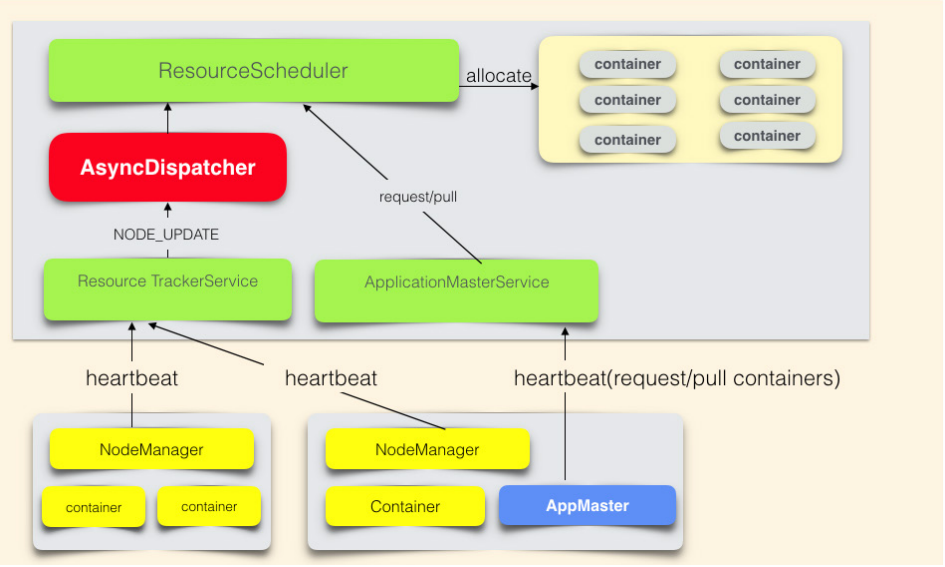
作业向 YARN 申请资源的请求是：List[ResourceRequest]

```
class ResourceRequest{
  int numContainers; // 需要的 container 个数
  Resource capability;// 每个 container 的资源
}
```

YARN 对作业响应是：List[Container]

```
class Container{
  ContainerId containerId; //YARN 全局唯一的 container 标示
  Resource capability;  // 该 container 的资源信息
  String nodeHttpAddress; // 该 container 可以启动的 NodeManager 的 hostname
}
```

YARN 调度架构



YARN 调度器

名词解释

- ResourceScheduler 是 YARN 的调度器，负责 Container 的分配。
- AsyncDispatcher 是单线程的事件分发器，负责向调度器发送调度事件。
- ResourceTrackerService 是资源跟踪服务，主要负责接收处理 NodeManager 的心跳信息。
- ApplicationMasterService 是作业的 RPC 服务，主要负责接收处理作业的心跳信息。
- AppMaster 是作业的程序控制器，负责跟 YARN 交互获取 / 释放资源。

调度流程

1. 作业资源申请过程：AppMaster 通过心跳告知 YARN 资源需求 (List[ResourceRequest])，并取回上次心跳之后，调度器已经分配好的资源 (List[Container])。

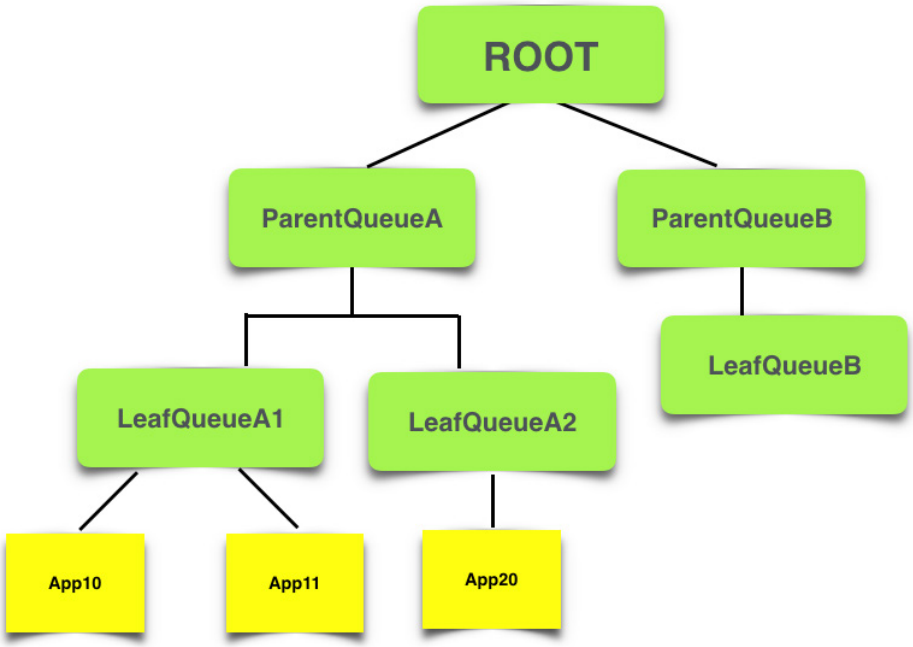
2. 调度器分配资源流程是：Nodemanager 心跳触发调度器为该 NodeManager 分配 Container。

资源申请和分配是异步进行的。ResourceScheduler 是抽象类，需要自行实现。社区实现了公平调度器 (FairScheduler) 和容量调度器 (CapacityScheduler)。美团点评根据自身的业务模式的特点，采用的是公平调度器。

公平调度器

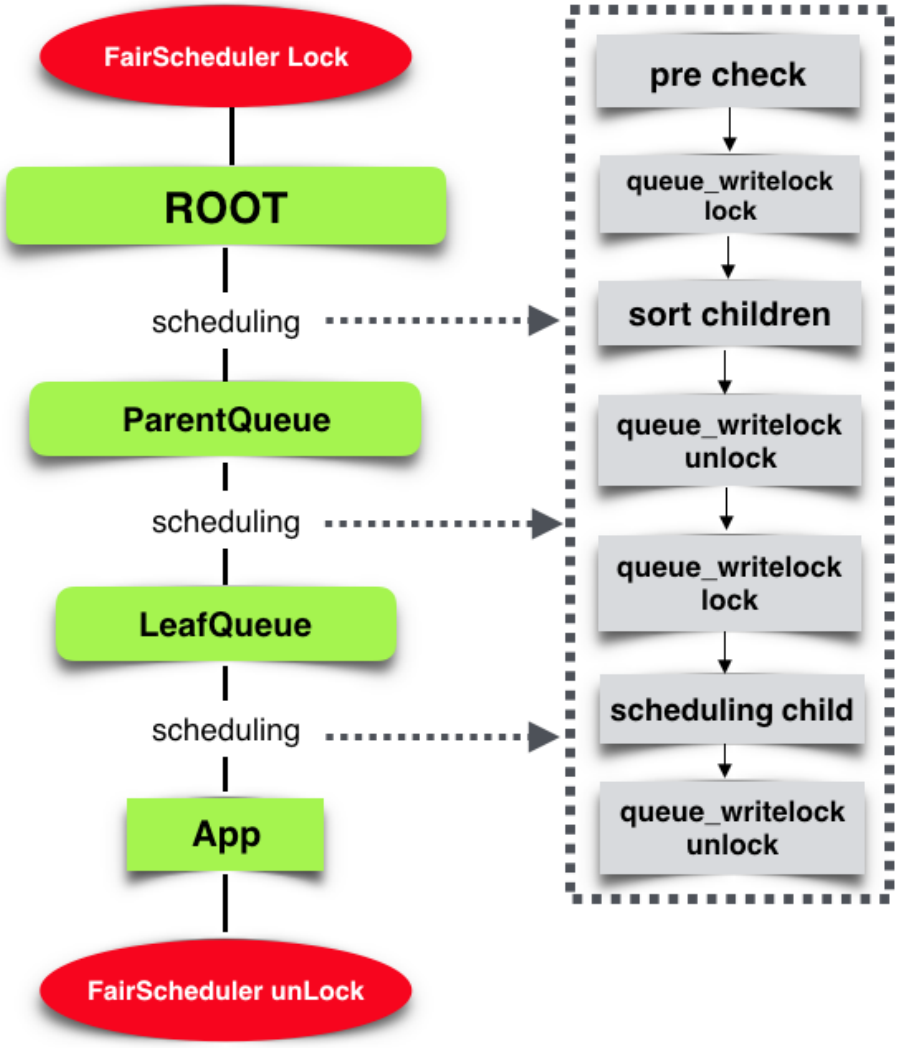
作业的组织方式

在公平调度器中，作业 (App) 是挂载如下图的树形队列的叶子。



队列结构

核心调度流程



核心调度流程

1. 调度器锁住 FairScheduler 对象，避免核心数据结构冲突。
2. 调度器选取集群的一个节点 (node)，从树形队列的根节点 ROOT 开始出发，每层队列都会按照公平策略选择一个子队列，最后在叶子队列按照公平策略选择一个 App，为这个 App 在 node 上找一块适配的资源。

对于每层队列进行如下流程：

1. 队列预先检查：检查队列的资源使用量是否已经超过了队列的 Quota
2. 排序子队列 /App：按照公平调度策略，对子队列 /App 进行排序
3. 递归调度子队列 /App

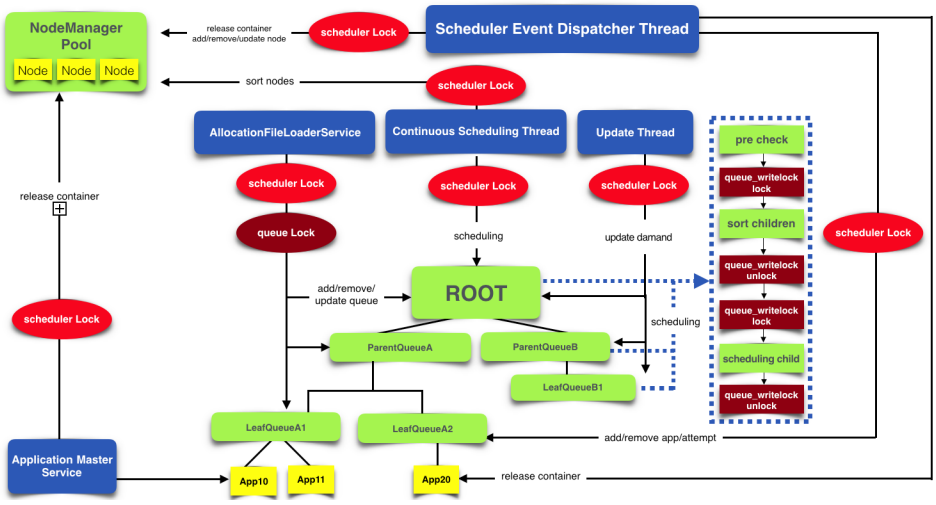
例如，某次调度的路径是 ROOT -> ParentQueueA -> LeafQueueA1 -> App11，这次调度会从 node 上给 App11 分配 Container。

伪代码

```
class FairScheduler{
  /* input：NodeId
   *  output：Resource 表示分配出来的某个 app 的一个 container 的资源量
   *  root 是树形队列 Queue 的根
   */
  synchronized Resource attemptScheduling(NodeId node){
    root.assignContainer(NodeId);
  }
}

class Queue{
  Resource assignContainer(NodeId node){
    if(! preCheck(node) ) return;  // 预先检查
      sort(this.children);  // 排序
    if(this.isParent){
      for(Queue q: this.children)
        q.assignContainer(node);  // 递归调用
    }else{
      for(App app: this.runnableApps)
        app.assignContainer(node);
    }
  }
}

class App{
  Resource assignContainer(NodeId node){
    ......
  }
}
```

公平调度器架构

公平调度器是一个多线程异步协作的架构，而为了保证调度过程中数据的一致性，在主要的流程中加入了 FairScheduler 对象锁。其中核心调度流程是单线程执行的。这意味着 Container 分配是串行的，这是调度器存在性能瓶颈的核心原因。



公平调度器架构

- scheduler Lock：FairScheduler 对象锁
- AllocationFileLoaderService：负责公平策略配置文件的热加载，更新队列数据结构
- Continuous Scheduling Thread：核心调度线程，不停地执行上节的核心调度流程
- Update Thread：更新队列资源需求，执行 Container 抢占流程等
- Scheduler Event Dispatcher Thread: 调度器事件的处理器，处理 App 新增，App 结束，node 新增，node 移除等事件

性能评估

上文介绍了公平调度器的架构，在大规模的业务压力下，这个系统存在性能问题。从应用层的表现看，作业资源需求得不到满足。从系统模块看，多个模块协同工

作，每个模块多多少少都存在性能问题。如何评估系统性能已经可以满足线上业务的需求？如何评估系统的业务承载能力？我们需要找到一个系统的性能目标。因此在谈性能优化方案之前，需要先说一说调度系统性能评估方法。

一般来说，在线业务系统的性能是用该系统能够承载的 QPS 和响应的 TP99 的延迟时间来评估，而调度系统与在线业务系统不同的是：调度系统的性能不能用 RPC（ResourceManager 接收 NodeManager 和 AppMaster 的 RPC 请求）的响应延迟来评估。原因是：这些 RPC 调用过程跟调度系统的调度过程是异步的，因此不论调度性能多么差，RPC 响应几乎不受影响。同理，不论 RPC 响应多么差，调度性能也几乎不受影响。

业务指标 – 有效调度

首先从满足业务需求角度分析调度系统的业务指标。调度系统的业务目标是满足业务资源需求。指标是：有效调度 (validSchedule)。在生产环境，只要 validSchedule 达标，我们就认为目前调度器是满足线上业务需求的。

定义 validSchedulePerMin 表示某一分钟的调度性能达标的情况。达标值为 1，不达标值为 0。

```
validPending = min(queuePending, QueueMaxQuota)
if  (usage / total  > 90% || validPending == 0):   validSchedulePerMin =
1 // 集群资源使用率高于
90%，或者集群有效资源需求为 0，这时调度器性能达标。
if (validPending > 0 &&  usage / total < 90%) : validSchedulePerMin =
0;// 集群资源使用率低于
90%，并且集群存在有效资源需求，这时调度器性能不达标。
```

- validPending 表示集群中作业有效的资源需求量
- queuePending 表示队列中所有作业的资源需求量
- QueueMaxQuota 表示该队列资源最大限额
- usage 表示集群已经使用的资源量
- total 表示集群总体资源

设置 90% 的原因是：资源池中的每个节点可能都有一小部分资源因为无法满足任何的资源需求，出现的资源碎片问题。这个问题类似 linux 内存的碎片问题。由于离线作业的任务执行时间非常短，资源很快可以得到回收。在离线计算场景，调度效率的重要性远远大于更精确地管理集群资源碎片，因此离线调度策略暂时没有考虑资源碎片的问题。

validSchedulePerDay 表示调度性能每天的达标率。validSchedulePerDay = Σ validSchedulePerMin /1440

目前线上业务规模下，业务指标如下：validSchedulePerMin > 0.9; validSchedulePerDay > 0.99

系统性能指标 – 每秒调度 Container 数

调度系统的本质是为作业分配 Container，因此提出调度系统性能指标 CPS – 每秒调度 Container 数。 在生产环境，只要 validSchedule 达标，表明目前调度器是满足线上业务需求的。而在测试环境，需要关注不同压力条件下的 CPS，找到当前系统承载能力的上限，并进一步指导性能优化工作。

CPS 是与测试压力相关的，测试压力越大，CPS 可能越低。从上文公平调度器的架构可以看到，CPS 跟如下信息相关：

- 集群总体资源数；集群资源越多，集群可以并发运行的的 Container 越多，对调度系统产生越大的调度压力。目前每台物理机的 cpu、memory 资源量差距不大，因此集群总体资源数主要看集群的物理机节点个数。
- 集群中正在运行的 App 数；作业数越多，需要调度的信息越多，调度压力越大。
- 集群中的队列个数；队列数越多，需要调度的信息越多，调度压力越大。
- 集群中每个任务的执行时间；任务执行时间越短会导致资源释放越快，那么动态产生的空闲资源越多，对调度系统产生的压力越大。

例如，集群 1000 个节点，同时运行 1000 个 App，这些 App 分布在 500 个 Queue 上，每个 App 的每个 Container 执行时间是 1 分钟。在这样的压力条件下，

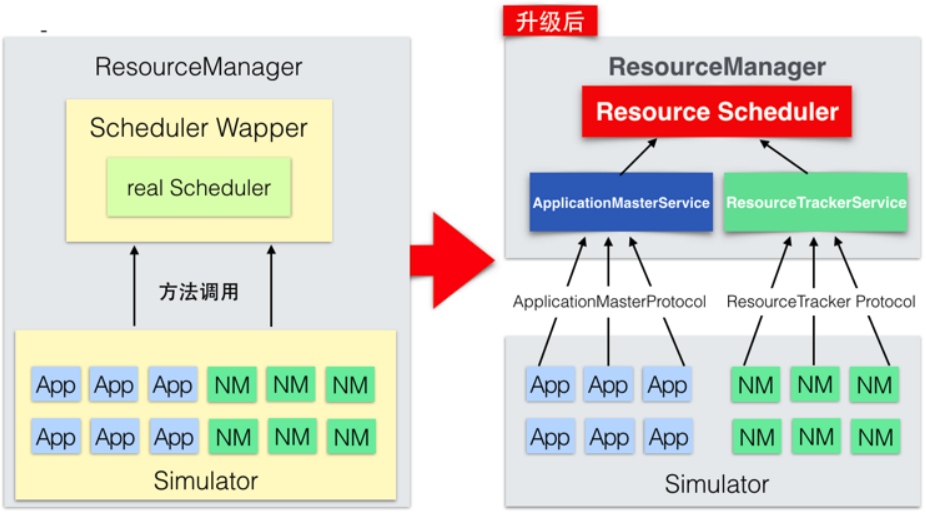
调度系统在有大量资源需求的情况下，每秒可以调度 1000 个 Container。那么在这个条件下，调度系统的 CPS 是 1000/s。

调度压力模拟器

在线上环境中，我们可以通过观察上文提到的调度系统的指标来看当前调度性能是否满足业务需求。但我们做了一个性能优化策略，不能直接到在线上环境去实验，因此我们必须有能力在线下环境验证调度器的性能是满足业务需求的，之后才能把实验有效的优化策略推广到线上环境。

那我们在线下也搭建一套跟线上规模一样的集群，是否就可以进行调度器性能优化的分析和研究呢？理论上是可以的，但这需要大量的物理机资源，对公司来说是个巨大的成本。因此我们需要一个调度器的压力模拟器，在不需要大量物理机资源的条件下，能够模拟 YARN 的调度过程。

社区提供了开源调度器的压力模拟工具 - Scheduler Load Simulater(SLS)。



调度压力模拟器

如上图，左侧是开源 SLS 的架构图，整体都在一个进程中，ResourceManager 模块里面有一个用线程模拟的 Scheduler。App 和 NM(NodeManager) 都是

由线程模拟。作业资源申请和 NM 节点心跳采用方法调用。

开源架构存在的问题有：

- 模拟大规模 APP 和 NM 需要开启大量的线程，导致调度器线程和 NM/App 的模拟线程争抢 cpu 资源，影响调度器的评估。
- SLS 的 Scheduler Wapper 中加入了不合理的逻辑，严重影响调度器的性能。
- SLS 为了通用性考虑，没有侵入 FairScheduler 的调度过程获取性能指标，仅仅从外围获取了 Queue 资源需求，Queue 资源使用量，App 资源需求，App 资源使用量等指标。这些指标都不是性能指标，无法利用这些指标分析系统性能瓶颈。

针对存在的问题，我们进行了架构改造。右侧是改造后的架构图，从 SLS 中剥离 Scheduler Wapper 的模拟逻辑，用真实的 ResourceManager 代替。SLS 仅仅负责模拟作业的资源申请和节点的心跳汇报。ResourceManager 是真实的，线上生产环境和线下压测环境暴露的指标是完全一样的，因此线上线下可以很直观地进行指标对比。详细代码参考：YARN-7672

细粒度监控指标

利用调度压力模拟器进行压测，观察到 validSchedule 不达标，但依然不清楚性能瓶颈到底在哪里。因此需要细粒度指标来确定性能的瓶颈点。由于调度过程是单线程的，因此细粒度指标获取的手段是侵入 FairScheduler，在调度流程中采集关键函数每分钟的时间消耗。目标是找到花费时间占比最多的函数，从而定位系统瓶颈。例如：在 preCheck 函数的前后加入时间统计，就可以收集到调度过程中 preCheck 消耗的时间。

基于以上的思路，我们定义了 10 多个细粒度指标，比较关键的指标有：

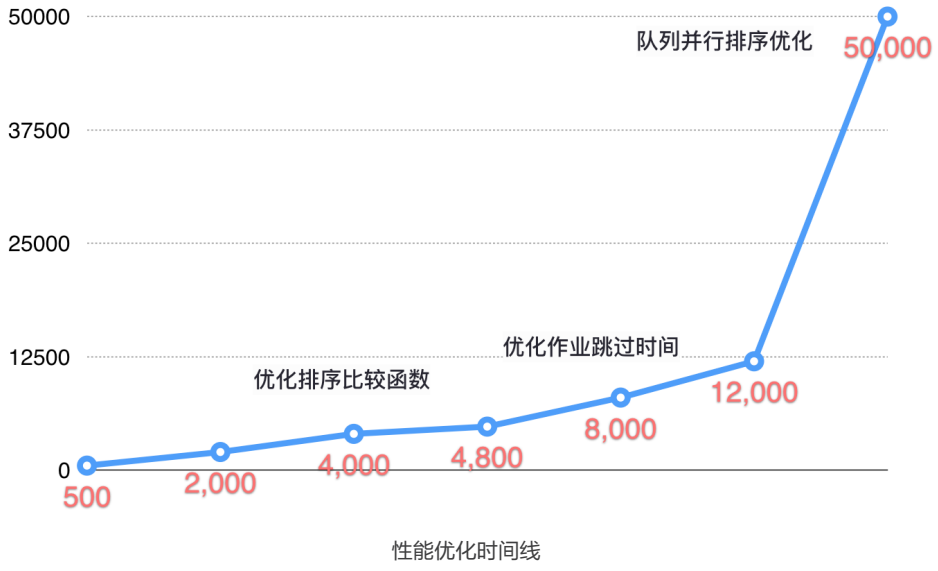
- 每分钟父队列 preCheck 时间
- 每分钟父队列排序时间
- 每分钟子队列 preCheck 时间

- 每分钟子队列排序时间
- 每分钟为作业分配资源的时间
- 每分钟因为作业无资源需求而花费的时间

关键优化点

第一次做压测，给定的压力就是当时线上生产环境峰值的压力情况（1000 节点、1000 作业并发、500 队列、单 Container 执行时间 40 秒）。经过优化后，调度器性能提升，满足业务需求，之后通过预估业务规模增长来调整测试压力，反复迭代地进行优化工作。

下图是性能优化时间线，纵轴为调度性能 CPS。



优化排序比较函数

在核心调度流程中，第 2 步是排序子队列。观察细粒度指标，可以很清楚地看到每分钟调度流程总共用时 50 秒，其中排序时间占用了 30 秒，占了最大比例，因此首先考虑优化排序时间。

排序本身用的快速排序算法，已经没有优化空间。进一步分析排序比较函数，发

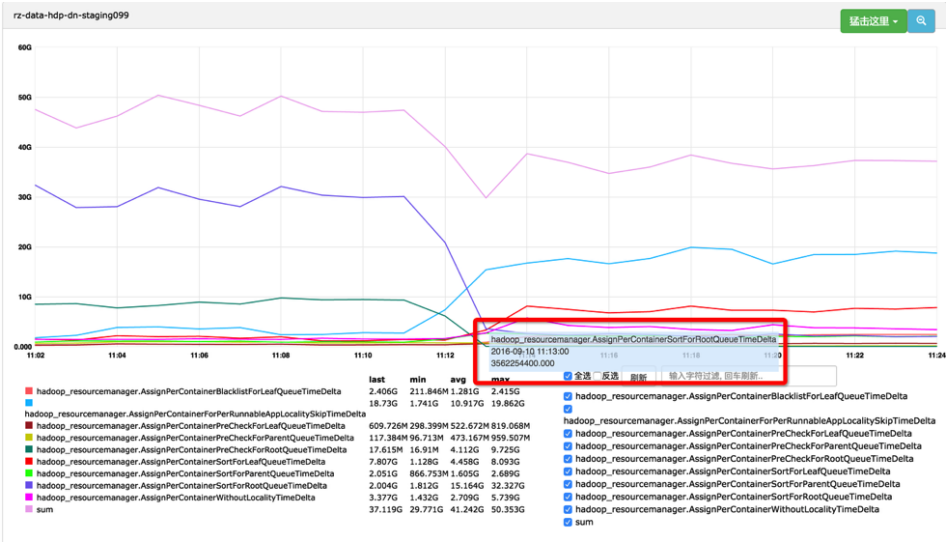
现排序比较函数的时间复杂度非常高。

计算复杂度最高的部分是：需要获取队列 / 作业的资源使用情况 (resource-Usage)。原算法中，每 2 个队列进行比较，需要获取 resourceUsage 的时候，程序都是现场计算。计算方式是递归累加该队列下所有作业的 resourceUsage。这造成了巨大的重复计算量。

优化策略：将现场计算优化为提前计算。

提前计算算法：当为某个 App 分配了一个 Container（资源量定义为 containerResource），那么递归调整父队列的 resourceUsage，让父队列的 resourceUsage += containerResource。当释放某个 App 的一个 Container，同样的道理，让父队列 resourceUsage -= containerResource。 利用提前计算算法，队列 resourceUsage 的统计时间复杂度降低到 O(1)。

优化效果：排序相关的细粒度指标耗时明显下降。



优化排序比较函数效果

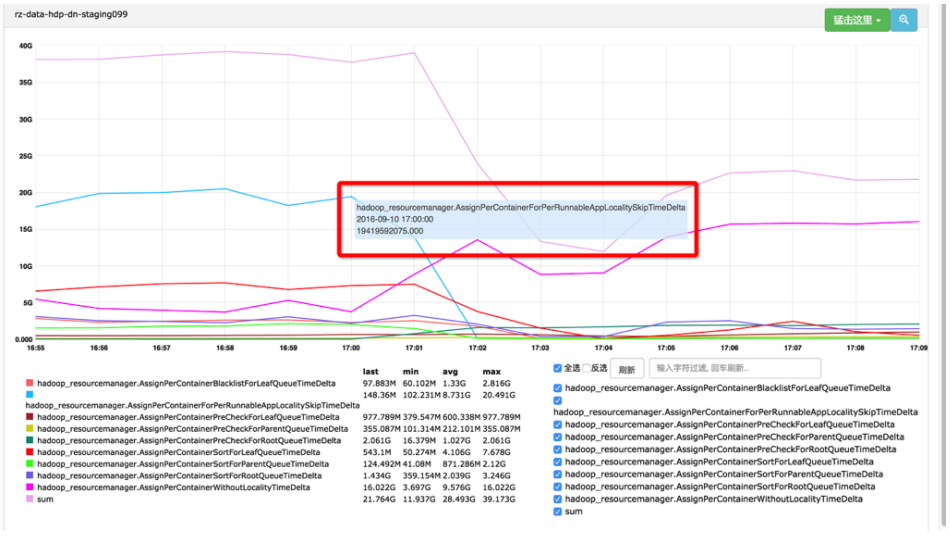
红框中的指标表示每分钟调度器用来做队列 / 作业排序的时间。从图中可以看出，经过优化，排序时间从每分钟 30G（30 秒）下降到 5G（5 秒）以内。 详细代码参考：YARN-5969

优化作业跳过时间

从上图看，优化排序比较函数后，蓝色的线有明显的增加，从 2 秒增加到了 20 秒。这条蓝线指标含义是每分钟调度器跳过没有资源需求的作业花费的时间。从时间占比角度来看，目前优化目标是减少这条蓝线的时间。

分析代码发现，所有队列 / 作业都会参与调度。但其实很多队列 / 作业根本没有资源需求，并不需要参与调度。因此优化策略是：在排序之前，从队列的 Children 中剔除掉没有资源需求的队列 / 作业。

优化效果：这个指标从 20 秒下降到几乎可以忽略不计。详细代码参考：YARN-3547



优化作业跳过时间

这时，从上图中可以明显看到有一条线呈现上升趋势，并且这个指标占了整个调度时间的最大比例。这条线对应的指标含义是确定要调度的作业后，调度器为这个作业分配出一个 Container 花费的时间。这部分逻辑平均执行一次的时间在 0.02ms 以内，并且不会随着集群规模、作业规模的增加而增加 , 因此暂时不做进一步优化。

队列并行排序优化

从核心调度流程可以看出，分配每一个 Container，都需要进行队列的排序。排序的时间会随着业务规模增加（作业数、队列数的增加）而线性增加。

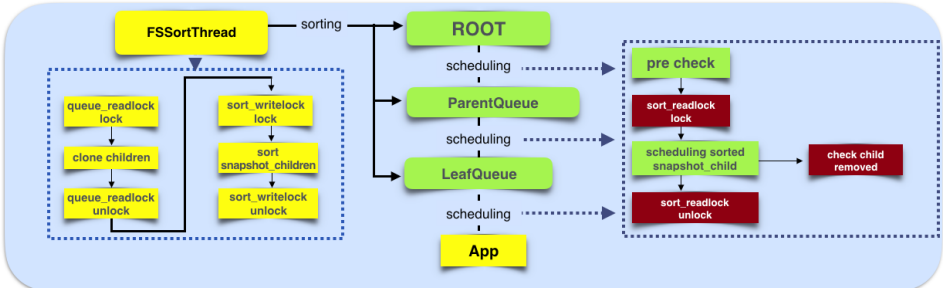
架构思考：对于公平调度器来说，排序是为了实现公平的调度策略，但资源需求是时时刻刻变化的，每次变化，都会引起作业资源使用的不公平。即使分配每一个 Container 时都进行排序，也无法在整个时间轴上达成公平策略。

例如，集群有 10 个 cpu，T1 时刻，集群只有一个作业 App1 在运行，申请了 10 个 cpu，那么集群会把这 10 个 cpu 都分配给 App1。T2 时刻 (T2 > T1)，集群中新来一个作业 App2，这时集群已经没有资源了，因此无法为 App2 分配资源。这时集群中 App1 和 App2 对资源的使用是不公平的。从这个例子看，仅仅通过调度的分配算法是无法在时间轴上实现公平调度。

目前公平调度器的公平策略是保证集群在某一时刻资源调度的公平。在整个时间轴上是需要抢占策略来补充达到公平的目标。 因此从时间轴的角度考虑，没有必要在分配每一个 Container 时都进行排序。

综上分析，优化策略是排序过程与调度过程并行化。要点如下：

1. 调度过程不再进行排序的步骤。
2. 独立的线程池处理所有队列的排序，其中每个线程处理一个队列的排序。
3. 排序之前，通过深度克隆队列 / 作业中用于排序部分的信息，保证排序过程中队列 / 作业的数据结构不变。

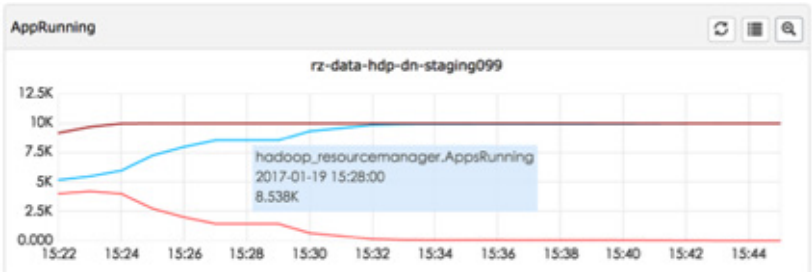


并行排序优化

优化效果如下：

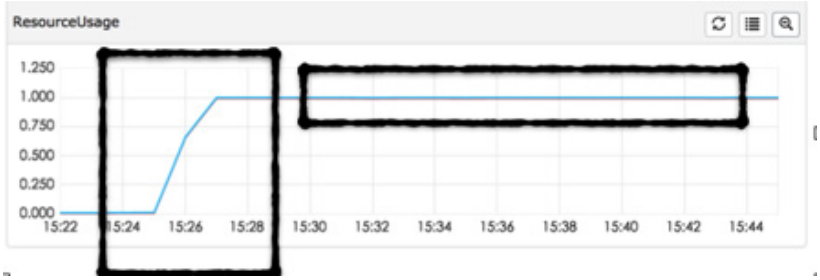
队列排序效率：利用线程池对 2000 个队列进行一次排序只需要 5 毫秒以内(2ms-5ms)，在一秒内至少可以完成 200 次排序，对业务完全没有影响。

在并行运行 1 万作业，集群 1.2 万的节点，队列个数 2000，单 Container 执行时间 40 秒的压力下，调度 CPS 达到 5 万，在一分钟内可以将整个集群资源打满，并持续打满。



并发作业数

作业资源需求量



集群资源使用率

上图中，15:26 分，pending 值是 0，表示这时集群目前所有的资源需求已经被调度完成。15:27 分，resourceUsage 达到 1.0，表示集群资源使用率为 100%，集群没有空闲资源。pending 值达到 4M（400 万 mb 的内存需求）是因为没有空闲资源导致的资源等待。

稳定上线的策略

线下压测的结果非常好，最终要上到线上才能达成业务目标。然而稳定上线是有难度的，原因：

- 线上环境和线下压测环境中的业务差别非常大。线下没问题，上线不一定没问题。
- 当时 YARN 集群只有一个，那么调度器也只有一个，如果调度器出现异常，是整个集群的灾难，导致整个集群不可用。

除了常规的单元测试、功能测试、压力测试、设置报警指标之外，我们根据业务场景提出了针对集群调度系统的上线策略。

在线回滚策略

离线生产的业务高峰在凌晨，因此凌晨服务出现故障的概率是最大的。而凌晨 RD 同学接到报警电话，执行通常的服务回滚流程（回滚代码，重启服务）的效率是很低的。并且重启期间，服务不可用，对业务产生了更长的不可用时间。因此我们针对调度器的每个优化策略都有参数配置。只需要修改参数配置，执行配置更新命令，那么在不重启服务的情况下，就可以改变调度器的执行逻辑，将执行逻辑切换回优化前的流程。

这里的关键问题是：系统通过配置加载线程更新了调度器某个参数的值，而调度线程也同时在按照这个参数值进行工作。在一次调度过程中可能多次查看这个参数的值，并且根据参数值来执行相应的逻辑。调度线程在一次调度过程中观察到的参数值发生变化，就会导致系统异常。

处理办法是通过复制资源的方式，避免多线程共享资源引起数据不一致的问题。调度线程在每次调度开始阶段，先将当前所有性能优化参数进行复制，确保在本次调

度过程中观察到的参数不会变更。

数据自动校验策略

优化算法是为了提升性能，但要注意不能影响算法的输出结果，确保算法正确性。对于复杂的算法优化，确保算法正确性是一个很有难度的工作。

在“优化排序比较时间”的研发中，变更了队列 resourceUsage 的计算方法，从现场计算变更为提前计算。那么如何保证优化后算法计算出来的 resourceUsage 是正确的呢?

即使做了单元策略，功能测试，压力测试，但面对一个复杂系统，依然不能有 100% 的把握。 另外，未来系统升级也可能引起这部分功能的 bug。

算法变更后，如果新的 resourceUsage 计算错误，那么就会导致调度策略一直错误执行下去。从而影响队列的资源分配。会对业务产生巨大的影响。例如，业务拿不到原本的资源量，导致业务延迟。

通过原先现场计算的方式得到的所有队列的 resourceUsage 一定是正确的，定义为 oldResourceUsage。 算法优化后，通过提前计算的方式得到所有队列的 resourceUsage，定义为 newResourceUsage。

在系统中，定期对 oldResourceUsage 和 newResourceUsage 进行比较，如果发现数据不一致，说明优化的算法有 bug，newResourceUsage 计算错误。这时系统会向 RD 发送报警通知，同时自动地将所有计算错误的数据用正确的数据替换，使得错误得到及时自动修正。

总结与未来展望

本文主要介绍了美团点评 Hadoop YARN 集群公平调度器的性能优化实践。

1. 做性能优化，首先要定义宏观的性能指标，从而能够评估系统的性能。
2. 定义压测需要观察的细粒度指标，才能清晰看到系统的瓶颈。
3. 工欲善其事，必先利其器。高效的压力测试工具是性能优化必备的利器。
4. 优化算法的思路主要有：降低算法时间复杂度；减少重复计算和不必要的计

算；并行化。

5. 性能优化是永无止境的，要根据真实业务来合理预估业务压力，逐步开展性能优化的工作。
6. 代码上线需谨慎，做好防御方案。

单个 YARN 集群调度器的性能优化总是有限的，目前我们可以支持 1 万节点的集群规模，那么未来 10 万，100 万的节点我们如何应对?

我们的解决思路是：基于社区的思路，设计适合美团点评的业务场景的技术方案。社区 Hadoop 3.0 研发了 Global Scheduling，完全颠覆了目前 YARN 调度器的架构，可以极大提高单集群调度性能。我们正在跟进这个 Feature。社区的 YARN Federation 已经逐步完善。该架构可以支撑多个 YARN 集群对外提供统一的集群计算服务，由于每个 YARN 集群都有自己的调度器，这相当于横向扩展了调度器的个数，从而提高集群整体的调度能力。我们基于社区的架构，结合美团点评的业务场景，正在不断地完善美团点评的 YARN Federation。

作者简介

世龙、廷稳，美团用户平台大数据与算法部研发工程师。

团队介绍

数据平台资源调度团队，目标是建设超大规模、高性能、支持异构计算资源和多场景的资源调度系统。目前管理的计算节点接近 3 万台，在单集群节点过万的规模下实现了单日数十万离线计算作业的高效调度，资源利用率超过 90%。资源调度系统同时实现了对实时计算作业、机器学习模型 Serving 服务等高可用场景的支持，可用性超过 99.9%。系统也提供了对 CPU/GPU 等异构资源的调度支持，实现了数千张 GPU 卡的高效调度，以及 CPU 资源的离线与训练混合调度，目前正在引入 NPU/FPGA 等更多异构资源，针对机器学习场景的特点实现更高效合理的调度策略。

团队有多个岗位正在招聘，如果你对超大规模系统的挑战感到兴奋，如果你对异构计算资源的调度策略感到好奇，欢迎加入我们，联系邮箱 tech@meituan.com ，注明“用户平台大数据”。

将军令：数据安全平台建设实践

夷山　中华

背景

在大数据时代，数据已经成为公司的核心竞争力。此前，我们介绍了美团酒旅起源数据治理平台的建设与实践，主要是通过各种数据分析挖掘手段，为公司发展决策和业务开展提供数据支持。

近期，业内数据安全事件频发，给相关企业造成了无可挽回的损失，更为数据安全防护意识薄弱的企业敲响了警钟。如何对公司内部数据最为集中的数据分析、数据服务、数据治理等各种数据类产品进行权限管控，已经成为数据安全建设中最为重要的任务。

如果从控制力的角度来进行划分的话，权限管控可以分为功能级权限管控和数据级权限管控。早期的数据安全产品大多使用传统的权限模型，只能实现功能级权限管控，无法进行数据级权限管控。基于数据类产品更高的安全要求，我们需要构建一个同时满足各类产品数据安全的我们需要构建一个同时满足各类产品数据安全的平台。

为此，美团用户平台应用研发组不仅设计了能表达和管控各种复杂关系的权限模型，还针对事前、事中、事后等三个场景，分别设计了审批、权限、审计三个子系统以保障数据安全的完整闭环，进而满足数据安全的各种要求。

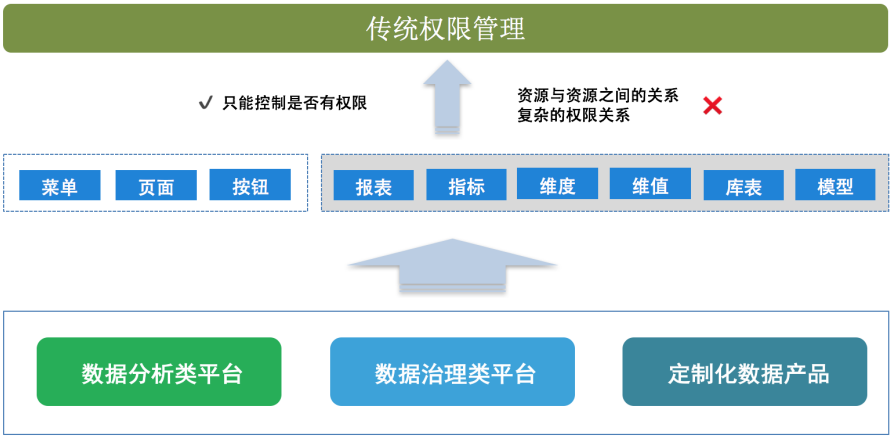


图 1　权限背景

功能应用类产品的权限表达，一般为“是否有权限”，而数据类产品权限表达的关系更加复杂。例如数据类产品的报表，不仅需要表达出用户能否访问这个报表，而且需要表达出用户能访问报表中的哪些维度、指标以及维值的范围。还需要告知这些维度指标来自于哪些库表模型，是否有权限访问以及创建报表。

权限模型

传统的权限模型有 ACL（Access Control List）访问控制列表，RBAC（Role-Based Access Control）基于角色的访问控制等。以上模型比较适用于应用类型产品的权限管控，而数据类型的产品对信息安全的要求更高，而且各类资源间的关系也更复杂，使用传统的模型难以将内部关系进行清晰的表达，所以我们在 RBAC 权限模型的基础上，扩展设计了新的权限模型。

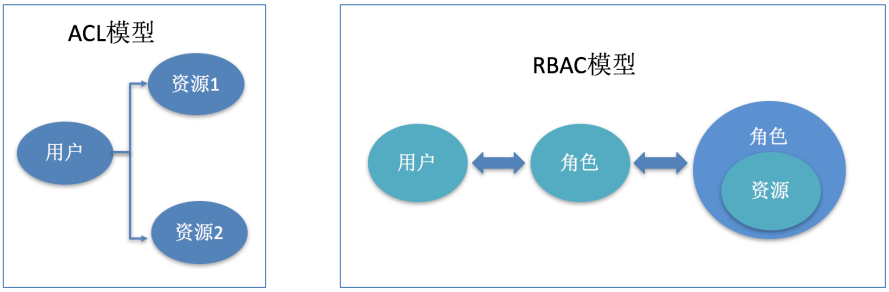


图 2　传统权限模型

如图 2 所示，传统的权限模型：

- ACL 访问控制列表，用户与权限直接关联，直接维护用户与列表中资源的关系从而达到权限管控的目的。
- RBAC 模型则是角色与权限进行关联，用户成为相应的角色而获得对应的权限。

为什么要设计新的权限模型?

1. ACL 模型是用户与资源直接建立关系，没有角色的概念。当某些用户需要一批同样资源的权限时，赋权操作就变得很复杂，此时这种模型就不太适应了。
2. RBAC 模型引入了角色的概念，角色与资源建立关系。当某些用户需要一批同样资源的权限时，只需要构建一个角色并赋予使用这批资源的权限。当用户加入这个角色时，则拥有该角色的所有权限。解决了赋权操作复杂的问题。

不过 ACL 模型和 RBAC 模型，都存在以下几个问题：

1. 数据类产品资源之间关系复杂，不能很好地表达这种复杂的关系。例如：一个报表下有多个标签页，一个标签页下有多个组件，一个组件下有多个维度、指标等。同时维度、指标又来自不同的数据模型、库表等。资源与资源之间存在关系，当管理员给一个用户赋予报表的全部或部分权限时，报表下的子资源需要同时获得对应的权限。
2. RBAC 模型中角色与角色之间没有对应的关系。例如：组织架构中，员工所

在的组织架构如下：华东区 / 销售一区 / 销售一组，员工拥有的角色是销售一组的角色。当角色之间没有关系时，员工如果需要华东区角色的权限时，需要添加到华东区的角色中。而如果角色与角色之间具有从属关系时，则能很好地解决这个问题。

新的权限模型是如何解决上面这些问题的：

1. 设计资源模型时，资源与资源之间具有从属关系，并且资源允许多层级，以树形结构展示。例如报表是一个父资源，标签、组件、维度指标都是报表下的子资源，这样赋权时能清晰地展示出报表资源与下面的子资源的关系，赋权和鉴权时才能满足各种权限控制的要求。
2. 角色与角色之间具有从属关系，例如员工在华东区 / 销售一区 / 销售一组的组织架构中，华东区 / 销售一区 / 销售一组这 3 个角色之间分别具有父子级的从属关系，当员工在销售一组部门下，则拥有华东区、销售一区、销售一组的所有权限。当权限不冲突时则直接合并所有权限，冲突时则以“就近原则”进行覆盖。

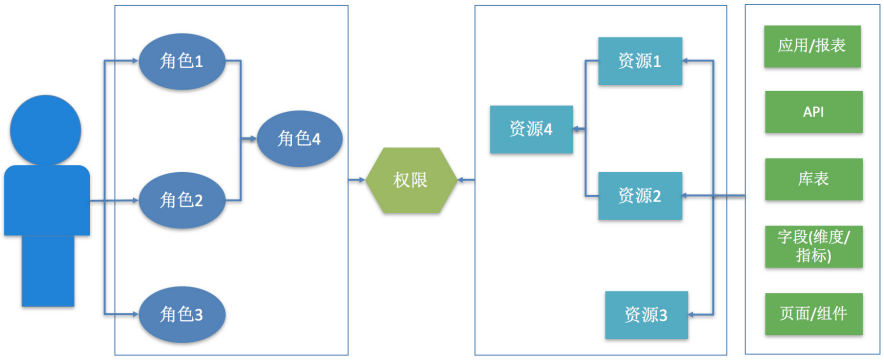


图 3　新的权限模型

如图 3 所示，新的权限模型包含 3 个部分，用户中心、资源中心、权限中心。

用户中心：用户管理、角色管理

- 角色分为个人、组织、自定义 3 种，一个用户可以同时拥有多个角色，比如用户默认对应一个个人角色，又可同时拥有在公司组织架构中组织角色、在自定义组织的自定义角色。
- 角色支持多层级，满足角色间权限继承的表达方式。
- 用户、部门信息 Mafka（美团基于 Kafka 开发的一个分布式消息中间件综合解决方案）实时更新，每天 ETL 定时同步，保证人员入职、转岗、调离权限实时同步。

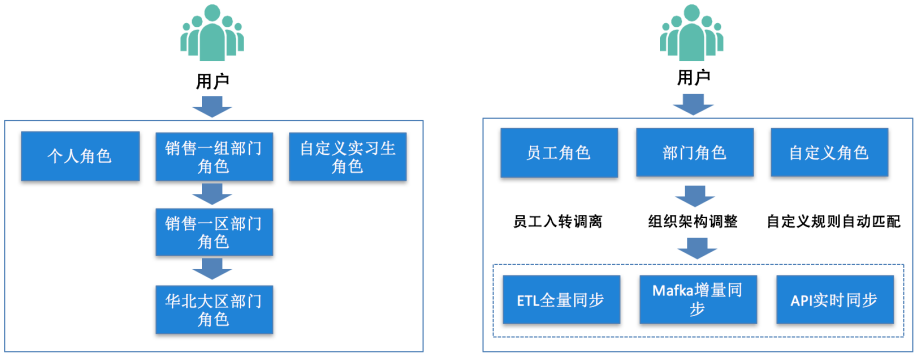


图 4　用户中心

资源中心：资源管理

- 资源类型支持自定义，在通用资源类型的基础上支持自定义的资源接入，满足各个系统不同资源的统一管控。
- 资源支持多层级，树形结构的资源展示方式便于资源的统一赋权鉴权；给一个报表资源赋权时，挂在报表下的维度、指标等资源能统一获得权限。
- 支持资源打包简化赋权流程。
- 资源安全密级、资源负责人，支持按照资源配置不同的审批模板进行权限自助申请。

图 5　资源中心

权限中心：角色与资源的关系的多种策略表达

- 范围策略：例如报表中的平台维度的维值包括美团和大众点评，赋权时，支持按要求给用户赋予部分或全部权限；鉴权时，按照规则解析为某人拥有某维度的部分或全部权限。
- 表达式策略：当把报表给用户赋权时，设置表达式为 limit 10，表示当前用户在该报表其他权限的基础上再进行限制，只能返回前 10 条记录。
- 权限自动合并：一个用户拥有多个角色，多角色的同一资源的权限鉴权时按照规则自动合并；规则解析时，权限数据不冲突时取合集，冲突时按照优先级取对应的值。
- 黑白名单：支持按照特定的规则，对某人针对某资源全面开发和封禁，黑白名单策略的优先级最高，其中黑名单高于白名单。

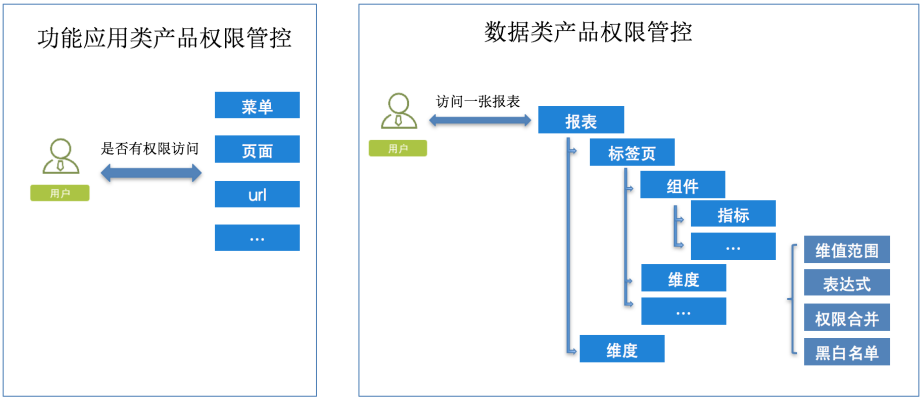


图 6　权限中心

挑战

在建设数据安全平台的过程中，主要面临以下几点挑战：

- 随着支持的业务线增加，通用平台的不能满足各个业务线的定制需求时，需要保证系统的灵活可扩展。
- 提供一个通用的数据安全平台，满足大部分的数据安全的要求，保证系统的通用性。
- 权限系统作为一个高 QPS 访问的系统，如何保证系统的高可用。

解决思路

1. 提供灵活可插拔的 Plugin 服务，在通用权限基础上，满足各个业务线灵活的权限管控要求。
2. 提供一个通用的数据安全平台，满足基本的权限、审批、审计的基础功能。
3. 微服务架构、核心与非核心服务分离、数据缓存降级满足系统高可用。

解决方案

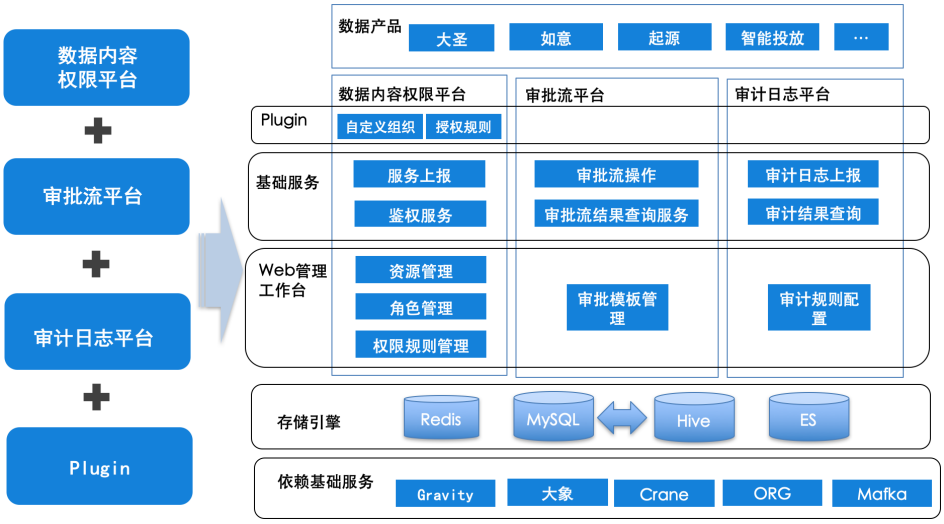


图 7　将军令整体架构

如图 7 所示，将军令分 3 块，数据内容权限平台、审批流平台、审计日志平台：

- 提供各种灵活可插拔的 Plugin 服务，支持在通用服务的基础基础上进行定制开发。
- 提供基础服务，满足各种通用的数据安全要求。
- 提供管理工作台，支持管理员对各种数据和规则进行页面管理和配置。

具体方案

Plugin 服务层，保证系统灵活可扩展

在满足通用权限的基础上，各个业务线难免会有定制的权限管控需求，于是设计了权限 Plugin 模块。

通用服务提供用户管理、资源管理、鉴权授权的服务，Plugin 调用基础服务实现特殊的权限管控。Plugin 模块的应用和数据各自单独管理，通过 RPC 方式调用通用服务实现灵活可插拔。后续 Plugin 模块的服务支持各个接入的应用单独定制开发。

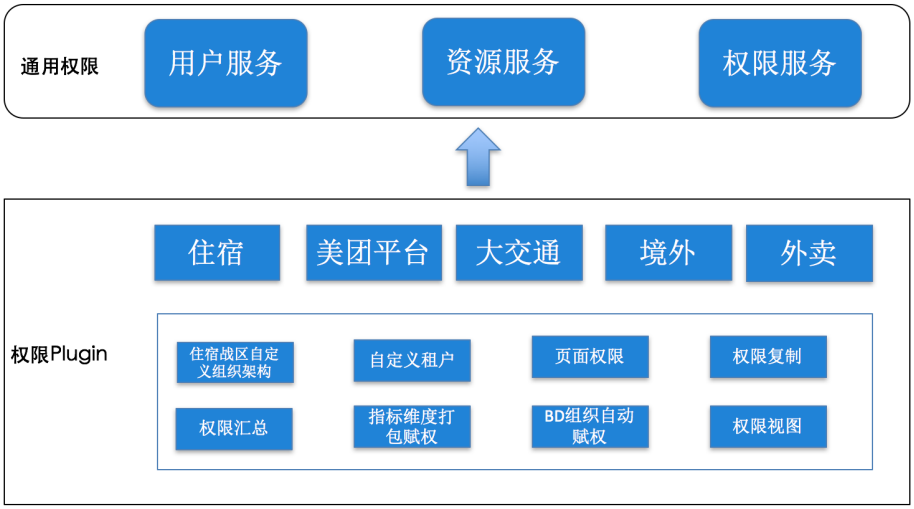


图 8　Plugin 服务

如图 8 所示，通用权限的服务与 Plugin 的服务是分离的，支持多个 Plugin 服务灵活可插拔：

- 通用服务提供用户、资源、鉴权授权等通用服务，大部分的系统基于通用服务即可实现权限管控要求。
- Plugin 服务基于通用服务对外提供的 SDK 进行拓展，各个 Plugin 服务单独部署，保证系统之间互相独立。

最终的权限实现分层管控，分为核心数据层（用户、资源、权限数据）和应用层。核心数据层的数据由通用服务进行管理，达到权限数据统一管控的要求。应用层以 Plugin 服务方式接入，Plugin 通过通用服务层对外的 SDK 进行权限数据读写，达到定制的管控要求。应用层的数据各自存储，可以自定义管控规则。接口之间的调用通过 BA 认证鉴权，保证服务之间调用的安全性。

基础服务层，保证系统通用性

通用权限系统架构

使用微服务架构设计，系统分为接入层、服务层、数据库层、以及外部服务层。主要包含以下几个核心服务：

- 用户服务：主要包含用户和部门信息同步、角色管理。
- 资源服务：包含资源注册、资源定时同步、资源密级及管理员管理、资源包管理。
- 赋权服务：权限自助申请、管理员赋权。
- 鉴权服务：提供各种鉴权的 SDK 供使用方调用。

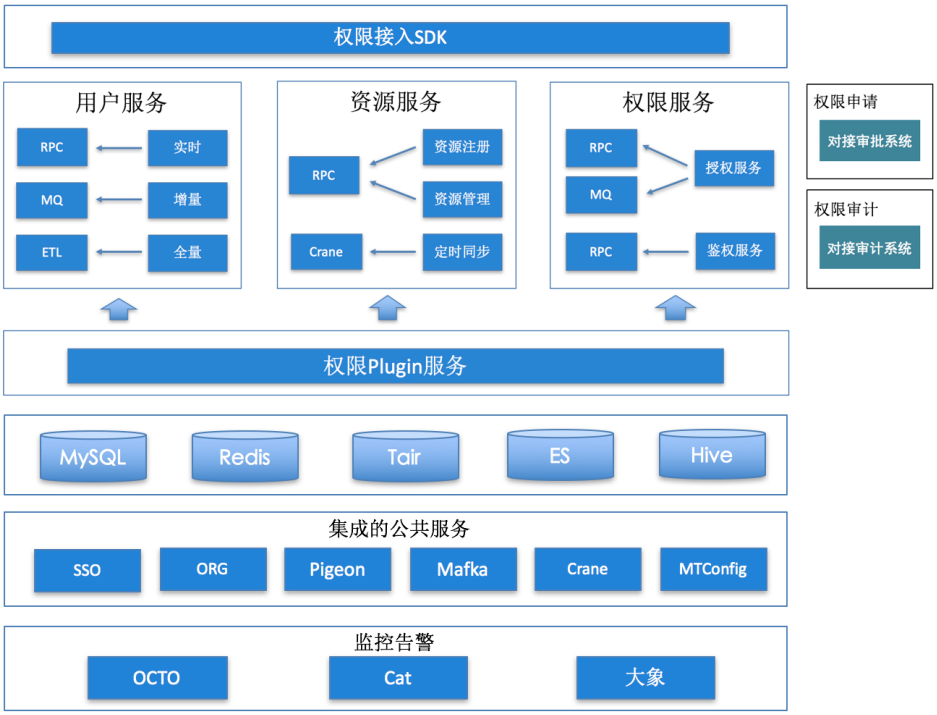


图 9　权限系统架构

如图 9 所示：

- 接入层：对外所有系统通过统一的 SDK 调用服务。
- 服务层：微服务架构，各个服务之间互相之间提供服务。
- 数据库层：合理利用缓存、数据降级，保证服务高可用。
- 集成公司公共服务，保证系统稳健运行。

审批系统架构

提供通用的审批服务，提供多级审批模板，使用时选择模板启动审批流，审批系统按照启动的参数进行规则解析，自动适配对应的审批流程。缩减接入流程支持一键接入。

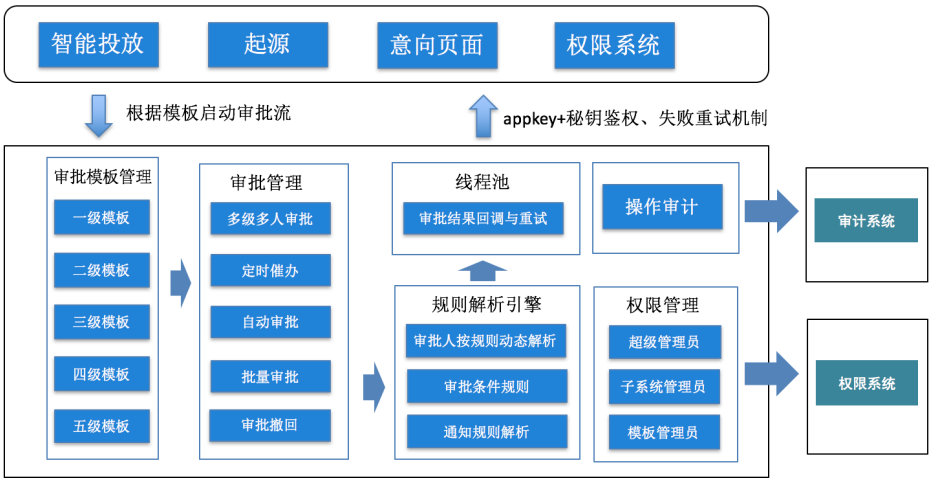


图 10　审批系统架构

如图 10 所示：优化审批接入流程，提供通用的审批服务，减少系统接入开发成本：

- 前期开发一个审批功能需要 6 个步骤，绘制流程图，配置审批的组和成员，配置通知的消息，配置事件映射，启动审批流，开发回调接口改状态。
- 而我们在平台的审批服务基础上进行封装，提供通用的审批模板，接入审批系统只需要选择模板启动审批流，并提供回调接口即可。能满足大部分的审批功能。

提供通用的规则解析引擎，支持审批人、审批条件、审批通知按照规则动态解析匹配。灵活实现自动审批、多人多级审批、定时催办等多种通用功能。

- 对接权限和审计系统，保证审批系统数据安全；
- 对接权限系统，提供管理员权限管控。

对接审计系统，操作数据落到审计系统便于后续的数据审计。

审计系统架构

提供通用的数据审计服务，客户端日志埋点上报，审计日志按类型落到 Elastic-search 中存储。对接如意可视化报表出审计报告，对接权限系统管控数据权限。

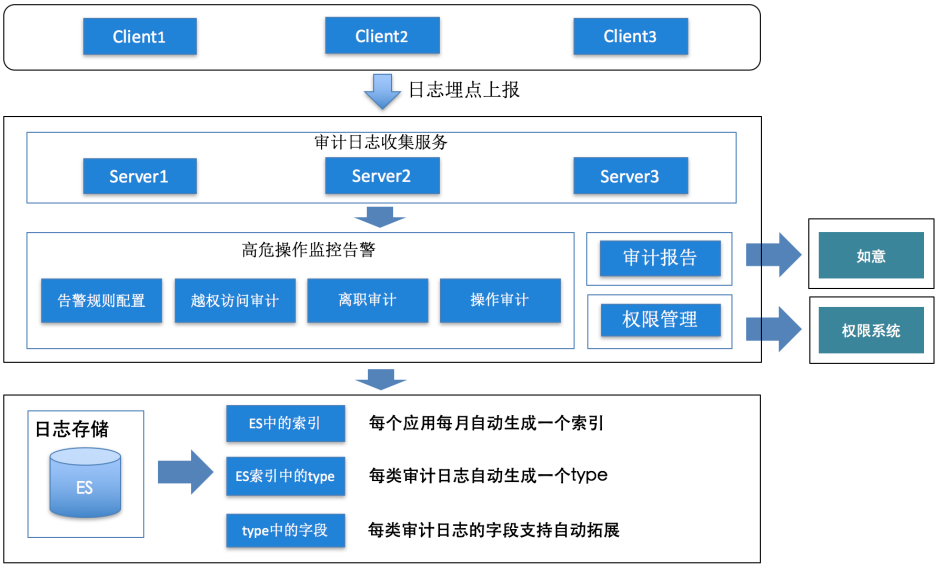


图 11　审计系统架构

如图 11 所示：审计数据模型层支持自动扩展：

- 每个应用对应一个 appkey，每个 appkey 按照模板分日期自动创建一个索引，支持自动扩展。
- 每种类型的审计日志对应 Elasticsearch 索引中的一个 type，新增一种操作日志时，type 自动创建。

- 审计日志中的字段对应 type 中的字段，新增字段时自动扩展。

保证系统高可用

微服务架构服务分离

随着系统的模块功能越来越多，单一架构模式已不再适合敏捷开发，模块越来越大系统启动则越慢，任一模块出错则整个系统的服务都不可用。

为了保证服务的高可用和扩展性，于是以微服务架构把模块进行拆分，并把核心与非核心服务进行分离。

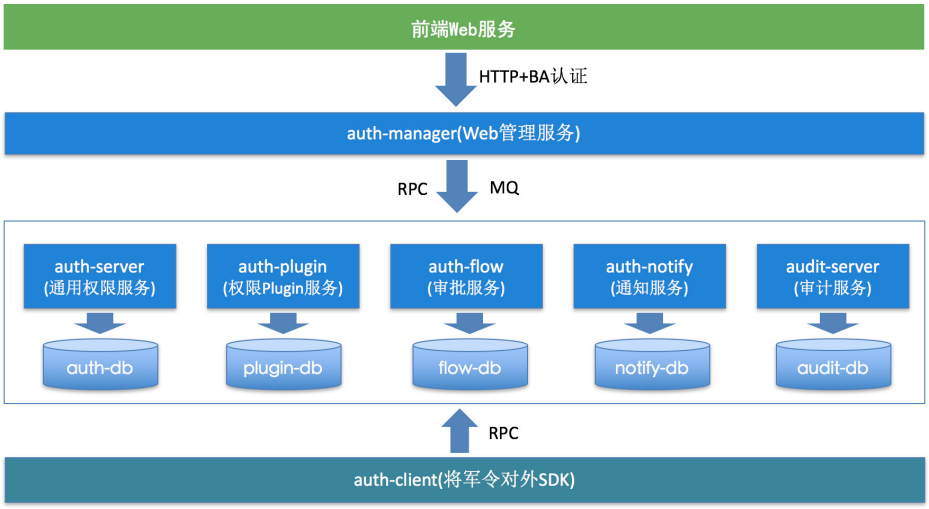


图 12　微服务架构

如图 12 所示：

- 前端接入层通过 HTTP 接入，BA 认证校验请求合法性，通过 Nginx 负载均衡。
- 管理控制台，通过调用服务层的各个服务实现统一管理。
- 服务层，抽象系统各个模块，每个模块都是一个微服务，每一个微服务都独立部署，可以根据每个服务的规模按需部署。
- Client 层，对外提供统一的 Pigeon（美团内部分布式服务 RPC 通信框架）接口，通过 POM 引入调用服务层各个服务。

权限继承

由于资源支持多层级，设计权限模型时支持权限继承，当赋权时开启继承，则用户默认拥有该资源以及下面所有资源的全部权限，数据存储时只需要存储祖先资源与用户之间的关系。大大减少了权限矩阵大小。

图 13　权限继承

权限数据存储

接入的系统越多，则资源和用户就越多。随着系统运行越久，对应的权限数据也会随之快速增长。如何在数据增长的同时保证接口的性能和高可用。

权限备份与恢复

参照 HBase 的版本号和 MySQL 的 Binlog 的设计思路，赋权时权限只存储当前用户最新权限数据，历史权限数据和操作记录用版本号的方式存储到 Elasticsearch 中。用户鉴权时只需要查询 MySQL 的权限数据即可，保证鉴权接

口的高效性。

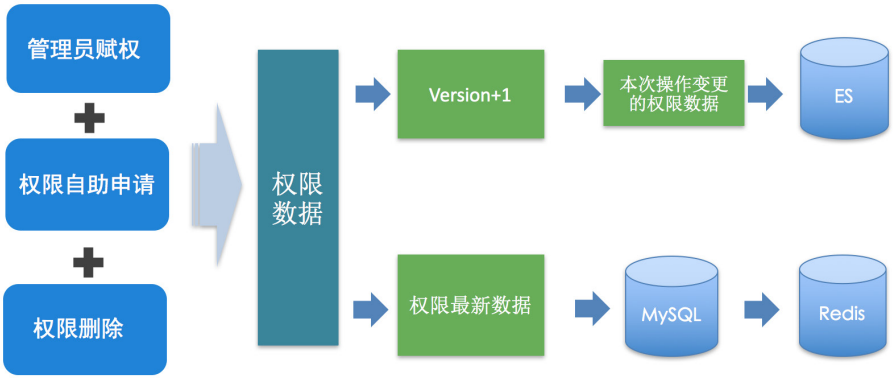


图 14　权限备份与恢复

如图 14 所示：

- 赋权操作时，通过版本号管理权限数据，每次操作后版本号加 1，MySQL 和 Redis 中只存储最新的权限数据。
- 历史权限数据通过版本号的方式存储到 Elasticsearch 中，每次查看历史操作记录或恢复权限数据时，根据版本号回溯即可。

权限过期清理

- 通过 Crane 定时调度，根据配置的通知规则，扫描即将过期的权限数据，发送消息通知用户进行权限续期。
- 扫描已过期的权限数据，清理 MySQL 和 Redis 中的过期权限数据，并转储到 Elasticsearch 中保存，已备后续的权限审计。

数据读写分离、缓存、备份以及服务熔断降级

各个服务使用 MySQL 分库存储，使用 Zebra（美团数据库访问层中间件）进行读写分离；合理使用数据缓存与备份，并支持服务的熔断降级，以保证服务的高可用。

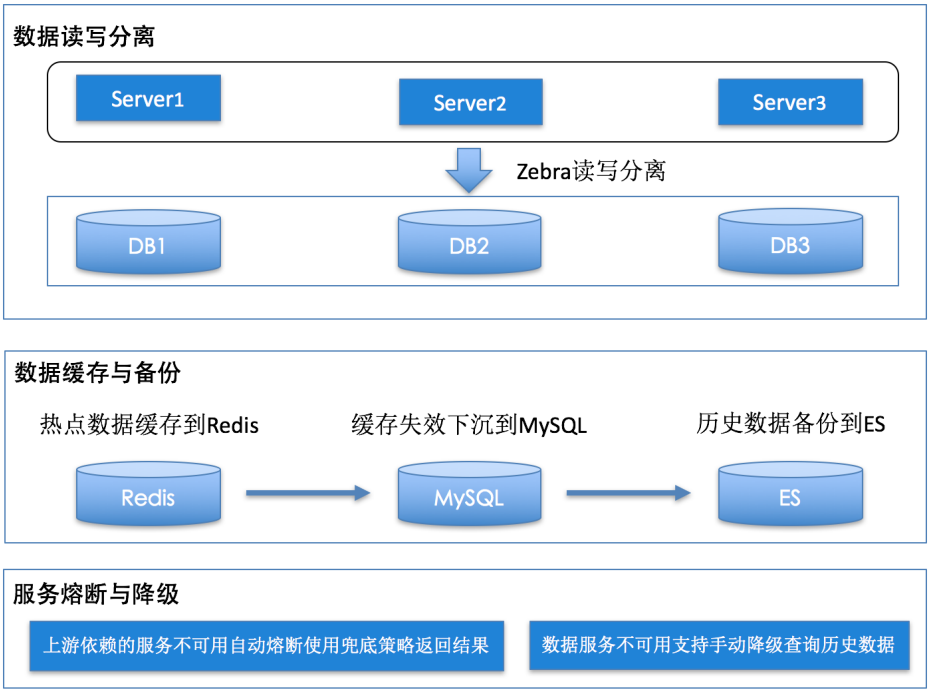


图 15　数据读写分离、缓存、备份以及服务熔断降级

如图 15 所示：

- 各个服务使用 MySQL 分库存储；核心服务与非核心服务分离，服务和数据库支持按需弹性拓展。
- 角色、资源等热点数据使用 Redis 做缓存，并在 Redis 缓存不可用时自动下沉到 MySQL 进行查询。
- 操作记录和历史数据等不活跃数据落地到 Elasticsearch，以便审计和数据恢复。
- 服务不可用时支持熔断降级，以保证核心服务的可用性。

合理使用消息队列、任务调度、线程池、分布式锁

使用消息队列、任务调度、线程池进行异步、削峰、解耦合，减少服务响应时间，提升用户体验。并使用分布式锁保证数据一致性。

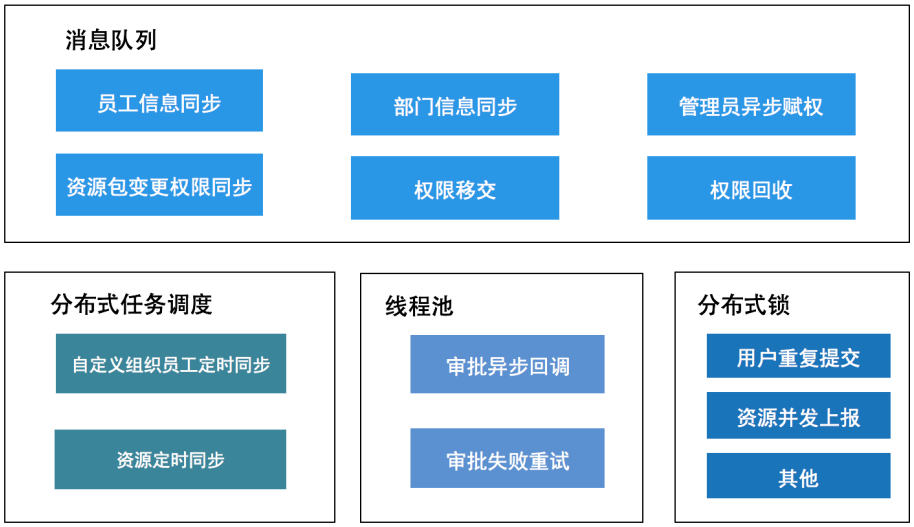


图 16　提高服务响应速度

如图 16 所示：

- 使用消息队列处理用户请求，实时返回操作成功，后台根据接受到的 MQ 消息异步进行处理并修改状态，页面轮询状态展示最终结果或发送大象（美团内部通讯工具）消息进行最终结果推送。
- 需要定时同步的任务通过 Crane 分布式任务调度平台进行定时调度执行。
- 审批回调时使用线程池处理审批结果回调与失败重试，较少创建销毁线程的开销。
- 分布式锁，保证同一个方法在同一操作上只能被一台机器上的一个线程执行，避免用户重复提交或者多机器重复处理导致的数据不一致。

展望

作为一个通用的数据安全平台，各个业务线的各种定制需求不可能都满足。目前在系统架构上已支持提供多个可插拔的 Plugin 服务，在通用服务的基础上实现定制的权限管控。后续将军令将针对权限、审批、审计提供 Plugin 开发规范，支持接入的系统在现有的基础上进行定制开发。

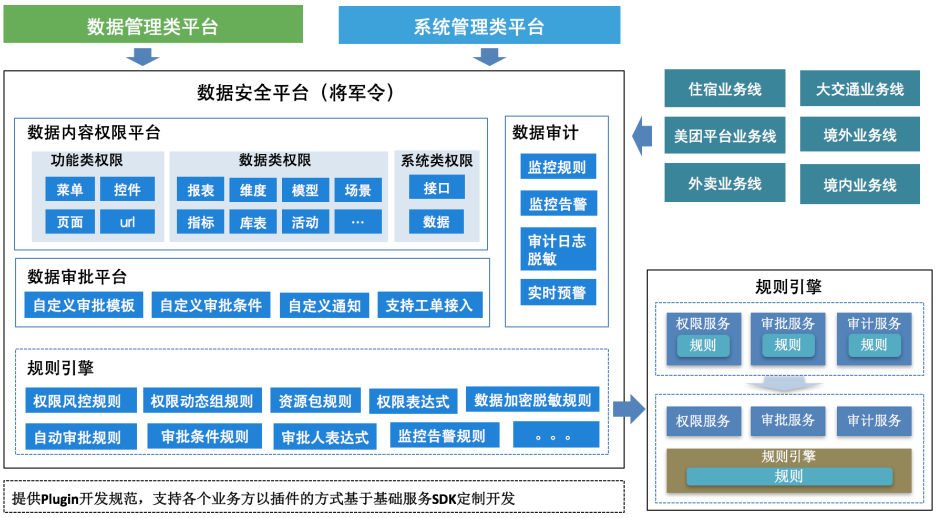


图 17　总体架构与展望

如图 17 所示：

- 后续将对外提供统一的 Plugin 开发规范，支持各个接入方系统以 Plugin 服务的形式在平台基础服务之上进行定制开发，以满足各自的特殊权限管控要求。从而实现数据产品权限集中管控确保数据安全。
- 把将军令中的规则从现有的服务中分离出来，抽象出一个通用的规则引擎服务，实现规则灵活可配置。

作者简介

夷山，美团点评技术专家，现任 TechClub-Java 俱乐部主席，2006 年毕业于武汉大学，先后就职于 IBM、用友、风行以及阿里。2014 年加入美团，长期致力于 BI 工具、数据安全与数据质量工作等方向。

中华，美团点评数据系统研发工程师，2017 年加入美团点评数据中心，长期从事于 BI 工具、数据安全相关工作。

招聘信息

最后插播一个招聘广告，有对数据产品工具开发感兴趣的可以发邮件给 fuyishan#meituan.com。

我们是一群擅长大数据领域数据工具，数据治理，智能数据应用架构设计及产品研发的工程师。

OneData 建设探索之路：SaaS 收银运营数仓建设

周成

背景

随着业务的发展，频繁迭代和跨部门的垂直业务单元变得越来越多。但由于缺乏前期规划，导致后期数仓出现了严重的数据质量问题，这给数据治理工作带来了很大的挑战。在数据仓库建设过程中，我们总结的问题包括如下几点：

- 缺乏统一的业务和技术标准，如：开发规范、指标口径和交付标准不统一。
- 缺乏有效统一的数据质量监控，如：列值信息不完整和不准确，SLA 时效无法保障等。
- 业务知识体系散乱不集中，导致不同研发人员对业务理解存在较大的偏差，造成产品的开发成本显著增加。
- 数据架构不合理，主要体现在数据层之间的分工不明显，缺乏一致的基础数据层，缺失统一维度和指标管理。

目标

在现有大数据平台的基础上，借鉴业界成熟 OneData 方法论，构建合理的数据体系架构、数据规范、模型标准和开发模式，以保障数据快速支撑不断变化的业务并驱动业务的发展，最终形成我们自己的 OneData 理论体系与实践体系。

OneData 探索

OneData：行业经验

在数据建设方面，阿里巴巴提出了一种 OneData 标准，如图 1 所示：

图 1　OneData 标准

OneData：我们的思考

他山之石，可以攻玉。我们结合实际情况和业界经验，进行了如下思考：

1. 对阿里巴巴 OneData 的思考

- 整个 OneData 体系覆盖范围广，包含数据规范定义体系、数据模型规范设计、ETL 规范研发以及支撑整个体系从方法到实施的工具体系。
- 实施周期较长，人力投入成本较高。
- 推广落地的工作比较依赖工具。

2. 对现有实际的思考

- 现阶段工具保障方面偏弱，人力比较缺乏。
- 现有开发流程不能全部推翻。

经过综合考量，我们发现直接全盘复用他人经验是不合理的。那我们如何定义自己的 OneData，即能在达到目标的前提下，又能避免上述的难题呢?

OneData：我们的想法

首先，结合行业经验，自身阶段的实践及以往的数仓经验，我们预先定义了 OneData 核心思想与 OneData 核心特点。

OneData 核心思想：从设计、开发、部署和使用层面，避免重复建设和指标冗余建设，从而保障数据口径的规范和统一，最终实现数据资产全链路关联、提供标准数据输出以及建立统一的数据公共层。

OneData 核心特点：三特性和三效果。

- 三特性：统一性、唯一性、规范性。
- 三效果：高扩展性、强复用性、低成本性。

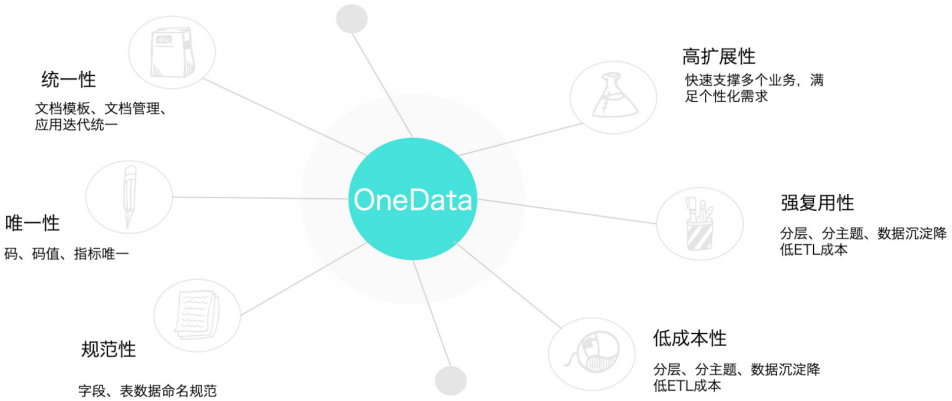


图 2　OneData 的六个特性

OneData：我们的策略

OneData 即有核心思想又有核心特点，到底怎么来实现核心思想又能满足其核心特点呢？通过以往经验的沉淀，我们提出两个统一方法策略：统一归口、统一出口。

根据两个统一的方法策略，我们开启了 OneData 的实践之路。

OneData 实践

统一业务归口

数据来源于业务并支撑着业务的发展。因此，数仓建设的基石就是对于业务的把控，数仓建设者即是技术专家，也应该是“大半个”业务专家。基于互联网行业的特点，我们基本上采用需求推动数据的建设，这也带来了一些问题，包括：数据对业务存在一定的滞后性；业务知识沉淀在各个需求里，导致业务知识体系分散。针对这些问题，我们提出统一业务归口，构建全局知识库，进而保障对业务认知的一致性。

图 3　支持业务的数据源知识库

设计统一归口

为了解决数据仓库建设过程中出现的各种痛点，我们从模型与规范两个方面进行建设，并提出设计统一归口。

1. 模型

规范化模型分层、数据流向和主题划分，从而降低研发成本，增强指标复用性，并提高业务的支撑能力。

（1）模型分层

优秀可靠的数仓体系，往往需要清晰的数据分层结构，即要保证数据层的稳定又要屏蔽对下游的影响，并且要避免链路过长。结合这些原则及以往的工作经验，我们将分层进行统一定义为四层：

图 4　数据分层架构

（2）模型数据流向

重构前，存在大量的烟囱式开发、分层应引用不规范性及数据链路混乱、血缘关系很难追溯和 SLA 时效难保障等问题。

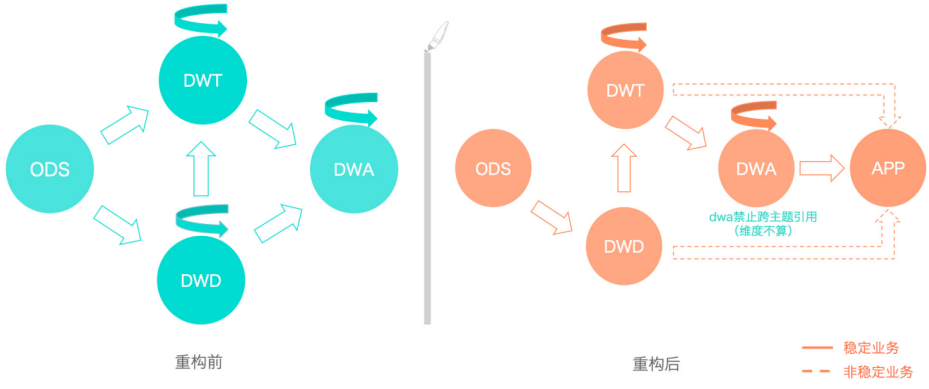


图 5　重构前和重构后的数据流向图

重构之后，稳定业务按照标准的数据流向进行开发，即 ODS->DWD->DWA->APP。非稳定业务或探索性需求，可以遵循 ODS->DWD->APP 或者 ODS->DWD->DWT->APP 两个模型数据流。在保障了数据链路的合理性之后，又在此基础上确认了模型分层引用原则：

- 正常流向：ODS>DWD->DWT->DWA->APP，当出现 ODS >DWD->DWA->APP 这种关系时，说明主题域未覆盖全。应将 DWD 数据落到 DWT 中，对于使用频度非常低的表允许 DWD->DWA。
- 尽量避免出现 DWA 宽表中使用 DWD 又使用（该 DWD 所归属主题域）DWT 的表。
- 同一主题域内对于 DWT 生成 DWT 的表，原则上要尽量避免，否则会影响 ETL 的效率。
- DWT、DWA 和 APP 中禁止直接使用 ODS 的表，ODS 的表只能被 DWD 引用。
- 禁止出现反向依赖，例如 DWT 的表依赖 DWA 的表。

2. 主题划分

传统行业如银行、制造业、电信、零售等行业中，都有比较成熟的主题划分，如 BDWM、FS-LDM、MLDM 等等。但从实际调研情况来看，主题划分太抽象会造成对业务理解和开发成本较高，不适用互联网行业。因此，结合各层的特性，我们提出了两类主题的划分：**面向业务**、**面向分析**。

- 面向业务：按照业务进行聚焦，降低对业务理解的难度，并能解耦复杂的业务。我们将实体关系模型进行变种处理为实体与业务过程模型。实体定义为业务过程的参与体；业务过程定义是由多个实体作用的结果，实体与业务过程都带有自己特有的属性。根据业务的聚合性，我们把业务进行拆分，形成了七大核心主题。
- 面向分析：按照分析聚焦，提升数据易用性，提高数据的共享与一致性。按照分析主体对象不同及分析特征，形成分析域主题在 DWA 进行应用，例如销售

分析域、组织分析域。

3. 规范

模型是整个数仓建设基石，规范是数仓建设的保障。为了避免出现指标重复建设和数据质量差的情况，我们统一按照最详细、可落地的方法进行规范建设。

（1）词根

词根是维度和指标管理的基础，划分为普通词根与专有词根，提高词根的易用性和关联性。

- 普通词根：描述事物的最小单元体，如：交易 -trade。
- 专有词根：具备约定成俗或行业专属的描述体，如：美元 -USD。

（2）表命名规范

通用规范

- 表名、字段名采用一个下划线分隔词根（示例：clienttype->client_type）。
- 每部分使用小写英文单词，属于通用字段的必须满足通用字段信息的定义。
- 表名、字段名需以字母为开头。
- 表名、字段名最长不超过 64 个英文字符。
- 优先使用词根中已有关键字（数仓标准配置中的词根管理），定期 Review 新增命名的不合理性。
- 在表名自定义部分禁止采用非标准的缩写。

表命名规则

表名称 = 类型 + 业务主题 + 子主题 + 表含义 + 存储格式 + 更新频率 + 结尾，如下图所示：

存储层-ods	业务库表名			全量-不加 快照-Snapshot-ss 增量-Increment-inc 拉链、缓慢变化维-SlowlyChangingDimensions-scd	按小时更新-hourly 按天更新-daily 按周更新-weekly 按月更新-monthly 每年-yearly	[_his]
明细层-dwd	业务主题-m	二级主题简称	自定义表名[_(detail|ext)]			
主题宽表层-dwt						
轻度汇总层-dwa						
应用层-app						
维度表-dim			维度名[_(detail|ext)]			
临时表-temp	业务表名					序号:01-100
视图	业务表名					view
外部表	原表名					extnl

例如：dwt_m_ord_order_ss_daily

[]：可选项，可以省略
()：多选项，以 | 分隔，必须选填一项

图 6　统一的表命名规范

（3）指标命名规范

结合指标的特性以及词根管理规范，将指标进行结构化处理。

A. 基础指标词根，即所有指标必须包含以下基础词根：

基础指标词根	英文全称	Hive数据类型	MySQL数据类型	长度	精度	词根	样例
数量	count	Bigint	Bigint	10	0	cnt	
金额类	amout	Decimal	Decimal	20	4	amt	
比率 / 占比	ratio	Decimal	Decimal	10	4	ratio	0.9818
……	……	……	……			……	

B. 业务修饰词，用于描述业务场景的词汇，例如 trade- 交易。

C. 日期修饰词，用于修饰业务发生的时间区间。

日期类型	全称	词根	备注
日	daily	d	
周	weekly	w	
……	……	……	

D. 聚合修饰词，对结果进行聚集操作。

聚合类型	全称	词根	备注
平均	average	avg	
周累计	wtd	wtd	本周一截止到当天累计
……	……	……	

E. 基础指标，单一的业务修饰词 + 基础指标词根构建基础指标 ，例如：交易金额 -trade_amt。

F. 派生指标，多修饰词 + 基础指标词根构建派生指标。派生指标继承基础指标的特性，例如：安装门店数量 -install_poi_cnt。

G. 普通指标命名规范，与字段命名规范一致，由词汇转换即可以。

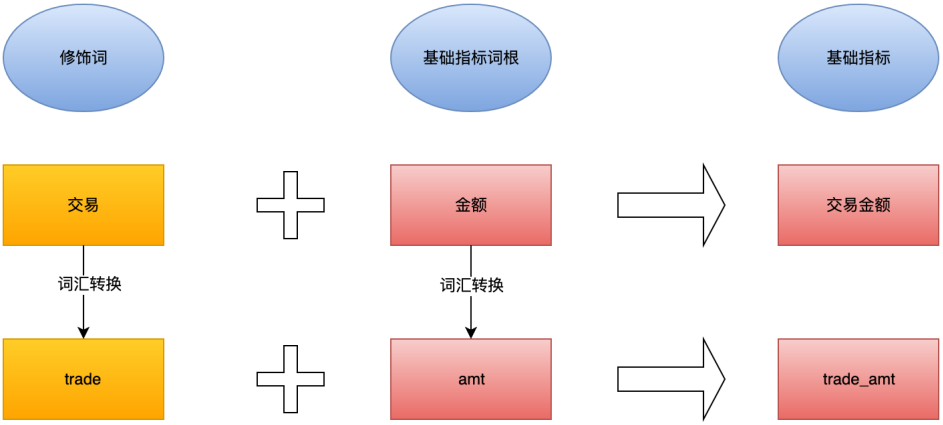


图 7　普通指标规范

H. 日期类型指标命名规范，命名时要遵循：业务修饰词 + 基础指标词根 + 日期修饰词 / 聚合修饰词。将日期后缀加到名称后面，如下图所示：

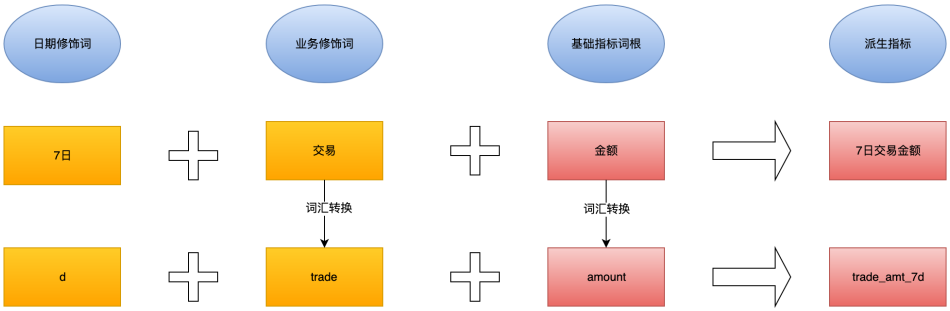


图 8　日期类型指标规范

I. 聚合类型指标，命名时要遵循：业务修饰词 + 基础指标词根 + 聚合类型 + 日期修饰词。将累积标记加到名称后面，如下图所示：

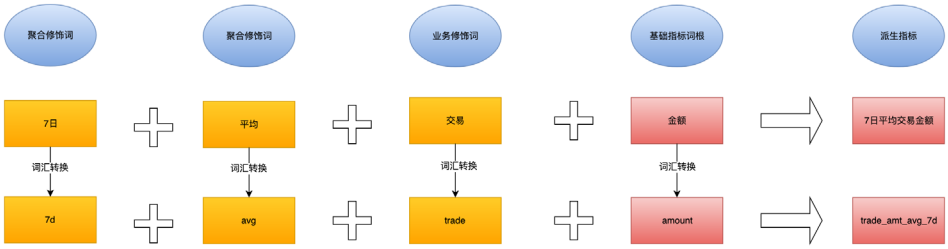


图 9　聚合类指标规范

（4）清洗规范

确认了字段命名和指标命名之后，根据指标与字段的部分特性，我们整理出了整个数仓可预知的 24 条清洗规范：

数据类型	数据类别	Hive 类型	MySQL 类型	长度	精度	词根	格式说明	备注
日期类型	字符日期类	string	varchar	10		date	YYYY-MM-DD	日期清洗为相应的格式
数据类型	数量类	bigint	bigint	10	0	cnt	活跃门店数量	
……	……	……	……	……	……	……	……	

结合模型与规范，形成模型设计及模型评审两者的工作职责，如下图所示：

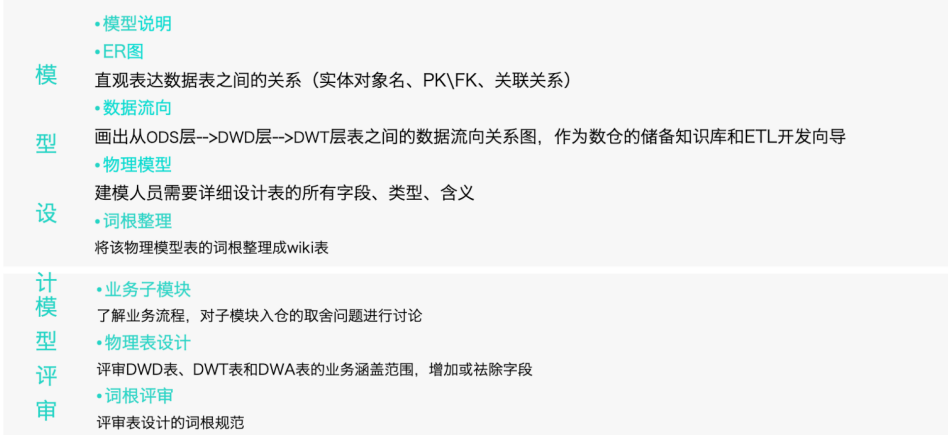


图 10　模型设计和审计职责

统一应用归口

在对原有的应用支持流程进行梳理的时候，我们发现整个研发过程是烟囱式。如果不进行改善就会导致前面的建设”毁于一旦“，所以需对原有应用支持流程进行改造，如下图所示：

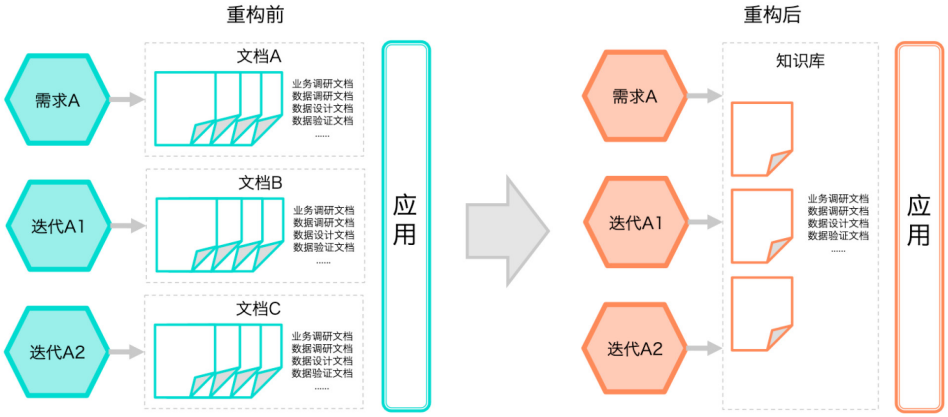


图 11　应用归口

从图中可以看出，重构前一个应用存在多次迭代，每次迭代都各自维护自己的文档。烟囱式开发会导致业务信息混乱、应用无法与文档对齐、知识传递成本、维护成本和迭代成本大大增加等问题。重构后，应用与知识库相对应，保证一个应用只对应一份文档，且应用统一要求在一份文档上进行迭代，从根源上改变应用支持流程。同时，针对核心业务细节和所支撑的数据信息，进行了全局调研并归纳到知识库。综合统一的知识库，降低了知识传递、理解、维护和迭代成本。

统一归口策略包含业务归口统一、设计归口统一和应用归口统一，从底层保证了数仓建设的**三特性**和**三效果**。

统一数据出口

数仓建设不仅仅是为了数据内容而建设，同时也为了提高交付的数据质量与数据使用的便利性。如何保证数据质量以及推广数据的使用，我们提出了统一数据出口策略。在进行数据资产管理和统一数据出口之前，必须高质量地保障输出的数据质量，从而树立 OneData 数据服务体系的权威性。

1. 交付标准化

如何保证数据质量，满足什么样的数据才是可交付的，是数据建设者一直探索的问题。为了保证交付的严谨性，在具体化测试方案之前，我们结合数仓的特点明确了数据交付标准的五个特性，如下图所示：

图 12　交付标准化

《交付标准化》完善了整个交付细节，从根本上保证了数据的质量，如：业务测试保障数据的合理性、一致性；技术测试保障数据的唯一性、准确性；数据平台的稳定性和后期人工维护保障数据的及时性。

2. 数据资产管理

针对如何解决数据质量中维度与指标一致性以及如何提高数据易用性的问题，我们提出数据资产的概念，借助公司内部平台工具“起源数据平台”实现了整个数据资产管理，它的功能如下图所示：

图 13　起源功能体系

借用起源数据平台，我们实现了：

- 统一指标管理，保证了指标定义、计算口径、数据来源的一致性。
- 统一维度管理，保证了维度定义、维度值的一致性。
- 统一数据出口，实现了维度和指标元数据信息的唯一出口，维值和指标数据的唯一出口。

通过交付标准化和数据资产管理，保证了数据质量与数据的易用性，同时我们也构建出 OneData 数据体系中数据指标管理的核心。

实践的成果

流程改善

我们对开发过程进行梳理，服务于整个 OneData 体系。对需求分析、指标管理、模型设计、数据验证进行了改善，并结合 OneData 模型策略，改善了数仓管理流程。

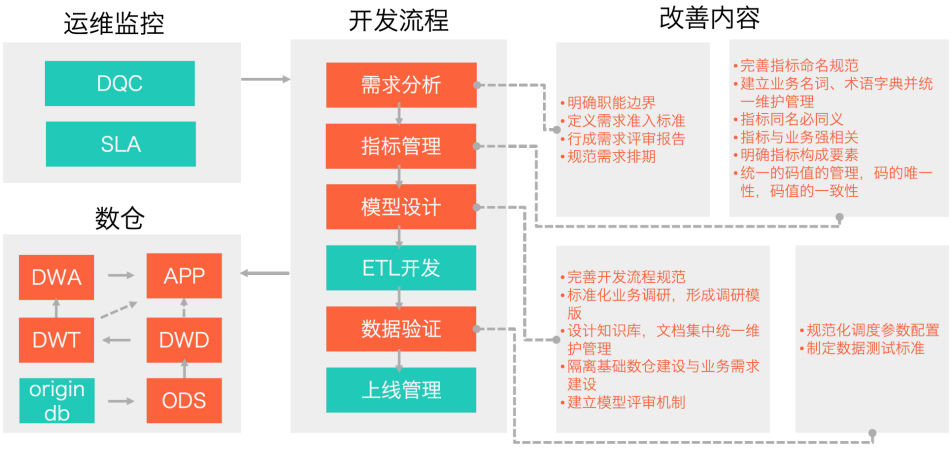


图 14　数仓管理流程

数仓全景图

基于 OneData 主题建设，我们采用**面向业务**、**面向分析**的建设策略，形成数仓全景图，如下图所示：

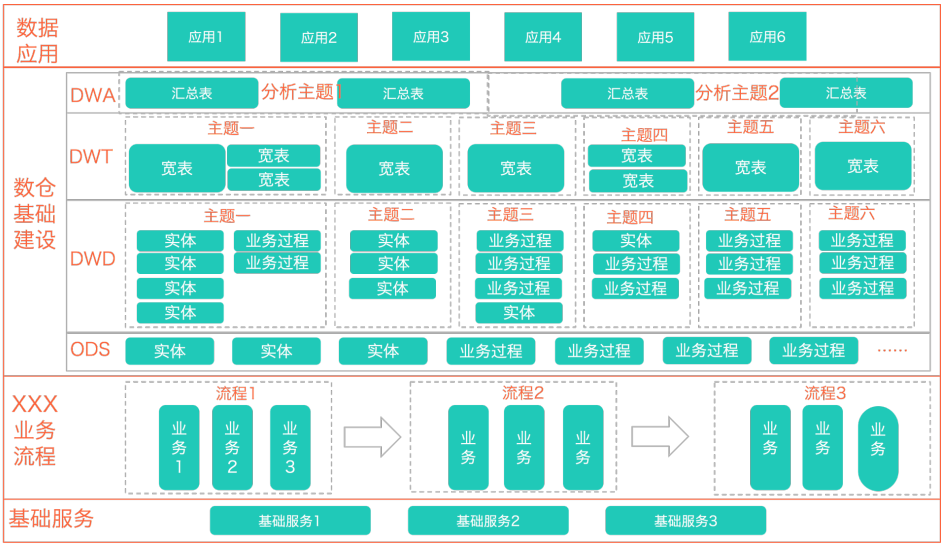


图 15　数仓全景图

资产管理列表

基于起源数据平台形成的资产管理体系，如下图所示：

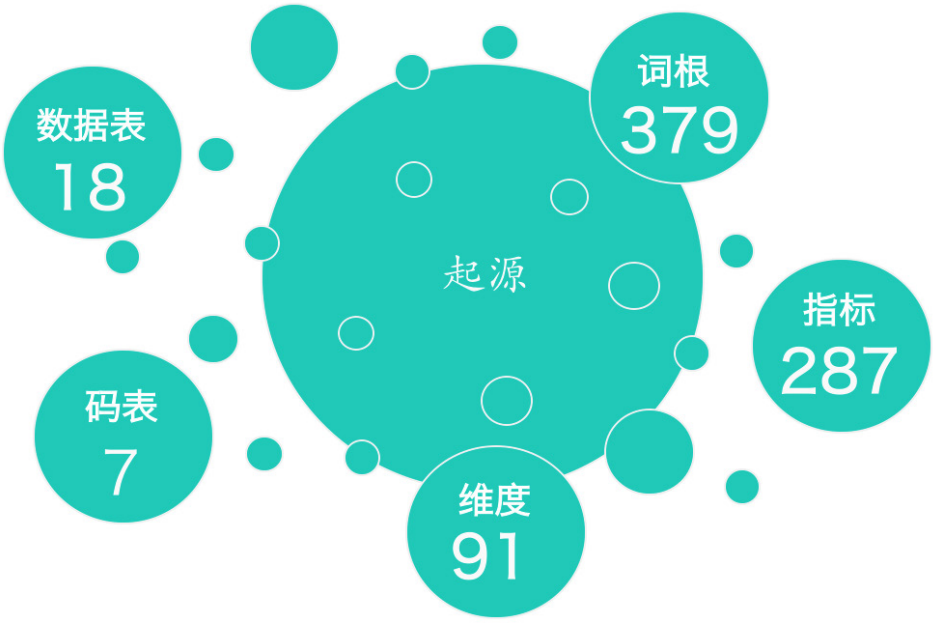


图 16　数据资产管理

项目收益

基于 OneData 建设成果，我们结合实际项目建设样例，对比以前未进行 OneData 建设时的收益。如下图所示：

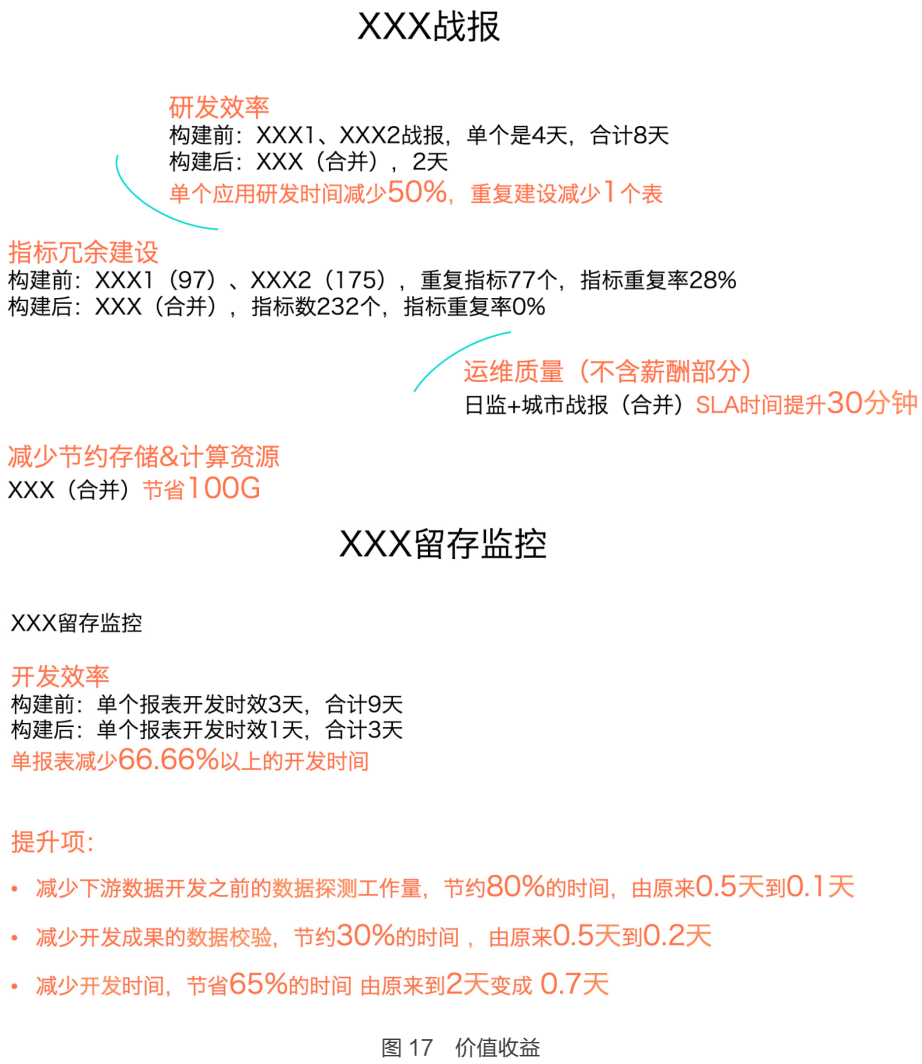


图 17　价值收益

总结和展望

我们结合了 OneData 核心思想与特点，构建一种稳定、可靠的基础数据仓库，从底层保障了数据质量，同时完成 OneData 实践，形成自有的 OneData 理论体系。未来，我们还将在技术上引入实时数据仓库，满足灵活多样、低延时的数据需求；在业务层面会横向拓展其他业务领域，不间断地支撑核心业务的决策与分析。下一步，我们将为企业级 One Entity 数据中台（以 Data As a Service 为理念），

提供强有力的数据支撑。在后续数仓维护过程中，不断地发现问题、解决问题和总结问题，保障数据稳定性、一致性和有效性，为核心业务构建价值链，最终形成企业级的数据资产。

作者

禄平，周成，黄浪，健平，高谦，美团数据研发工程师。

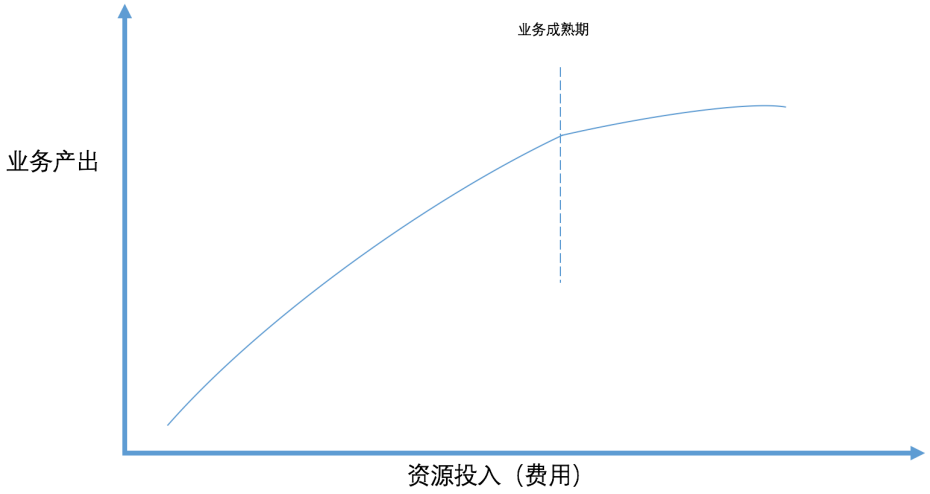
研发团队资源成本优化实践

刘强 建钟 小英 杨轩 云杰 方旭 鹏文

背景

工程师主要面对的是技术挑战，更关注技术层面的目标。研发团队的管理者则会把实现项目成果和业务需求作为核心目标。实际项目中，研发团队所需资源（比如物理机器、内存、硬盘、网络带宽等）的成本，很容易被忽略，或者在很晚才考虑。

在一般情况下，如果要满足更多的技术指标如并发量和复杂度等，或者满足峰值业务的压力，最直接有效的方法就是投入更多的资源。然而，从全局来看，如果资源成本缺乏优化，最终会出现如下图所示的“边际效用递减”现象——技术能力的提升和资源的增幅并不匹配，甚至资源的膨胀速度会超过技术能力的提升，从而使技术项目本身的 ROI 大打折扣。



从笔者十余年的工作经验来看，资源成本优化宜早不宜迟。很多管理者在业务发展的较前期阶段，可能并没有意识到资源成本膨胀所带来的压力。等到业务到了一定规模，拿到机器账单的时候，惊呼“机器怎么这么费钱”，再想立即降低成本，可能

已经错过了最佳时机，因为技术本身是一个相对长期的改造过程。

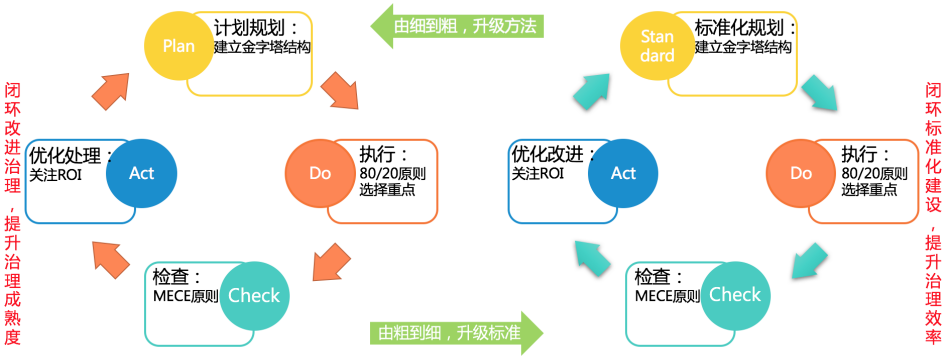
业务阶段	业务特点	成本管理意识
探索期	验证模式，从0到1	粗放式管理，只控制上限
进攻期	市场占有率是唯一目标	不需要控制成本，要多少给多少
发展期	稳居市场TOP2，业务成熟	看财报，发现设备这么花钱？
变革期	增速放缓，转型或变革	机器成本要控制了！

所以，正在阅读此文的读者，假如你已经感觉到了成本膨胀的压力，或者正在做成本控制相关的工作，恭喜，这是幸福的烦恼，贵公司的业务体量应该已经达到一定规模，同时也说明你的管理意识可能已经超前于业务发展了（握手）。

本文我们将分享美团到餐研发团队的资源成本优化实践。

实践

像美团这种体量的公司，资源的提供方包括多个团队，除了到店餐饮研发团队自用的资源外，还有多个兄弟团队也提供了资源支持或者联合共建，比如 SRE、大数据团队、风控团队、广告团队等等。在每个月拿到成本账单之后，我们都需要抽丝剥茧，对每一项进行拆解，才能制定对应的解决策略，具体流程如下图所示。



1. 确定方法论

“凡事预则立，不预则废”。在做一件事之前，要充分评估整个工作完整生命周期的要素，并进行整体工作框架的设计，一个科学的方法论是十分有必要的。成本优化

遵循的是一个行业内成熟的 P（Plan）D（Do）C（Check）A（Act）方法论，在每个阶段都又有对应的二次迭代和微循环。具体方法论如下：

- **在 Plan 或 Standard 阶段要做的事**：建立意识 -> 确定目标 -> 分析现状 -> 确定评价指标。
- **在 Do 执行阶段要做的事**：分解原子项目 -> 确定方案 -> 落实到人 -> 优化原子指标。
- **在 Check 检查阶段要做的事**：规定动作检查 -> 行动结果评估 -> 系统问题定位 -> 修正标准动作。
- **在 Act 优化处理阶段要做的事**：定期复盘 -> 形成报告 -> 迭代认知 -> 升级方法论 -> 下阶段目标。

2. 计划规划阶段（Plan&Standard）

在这个阶段的核心目标是：**用 2-3 个指标来衡量自己的工作**。很多工作之所以最后失败，很多时候是相关人员根本没有办法用具体可衡量的指标来衡量自己的工作，这样对于工作结果，只能有一个“定性”的认识（比如很好，很不错，不好，较差），而无法做到“定量”。

对于研发人员来讲，不能定量的结果是不够科学的，具体如何确定指标，或者确定哪些指标作为工作目标，其实也是一门学问（有机会另外发文章讨论）。这个阶段的几个建议步骤为：

- **建立意识**。这个是团队 Leader 的首要责任，要让团队成员明白自己在资源上花了多少钱，成本控制是不是一件真正有意义和价值的事，要做到大家认知一致。虽然见到过一些团队在提倡成本控制，但是落实到具体行动时，却流于形式或者无从下手，最后只能停留在口头上，并没有产生实际的效果。
- **确定目标**。这个过程相对宏观，也可以认为是“定性”的阶段。在这个阶段要明确的就是，在成本控制这件事上，后续动作要解决的问题是什么？比如有些团队是总体成本偏高，但有些团队总成本并不高，而是应该增加成本，有些团

队是非核心服务消耗的成本偏高，这些目标都需要经过团队成员讨论后得到一致的结果。在后续阶段的迭代中，也可以进行不断地修正。就像“客户永远不知道自己的需求”一样，很多人是不清楚自己的目标的，可以使用 SMART 原则来明确目标。

- **分析现状**。对成本这件事，罗列相关的数据，尽可能多地帮助自己做判断。自己团队在成本优化这件事上，处在哪一个阶段，哪些工作有可能被进一步优化，在此阶段要明确出来。
- **确定评价指标**。对于不同的专业序列，甚至对于同一专业序列的不同人员，大家对于成本的评价指标都不一样。这个阶段要做到最终的收敛，把团队未来成本优化的结果，用明确的数据表示出来。具体在到餐研发团队，我们确认了 2 个优化的核心指标：总成本、总订单成本。后续大家所有努力的目标，如果跟这两个指标没有关系或者弱相关，都可以忽略。

本阶段最大的经验是“知易行难”，虽然拍脑袋想出来一两个方向和目标很容易，但是最后用数据论证现状时，如何判断自己这个指标是“优秀”、“良好”还是“不及格”？对标的团队是谁？为什么对标的对象是 TA？都是需要从人员规模、业务阶段、业务量、行业特点等方面考虑仔细，也需要想清楚，其工作量甚至不比实际干活阶段小。

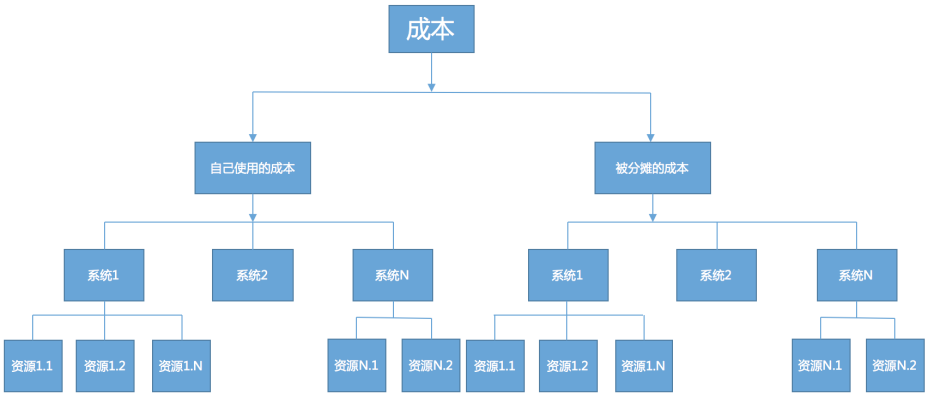
3. 执行阶段（Do）

3.1 建立思考流程

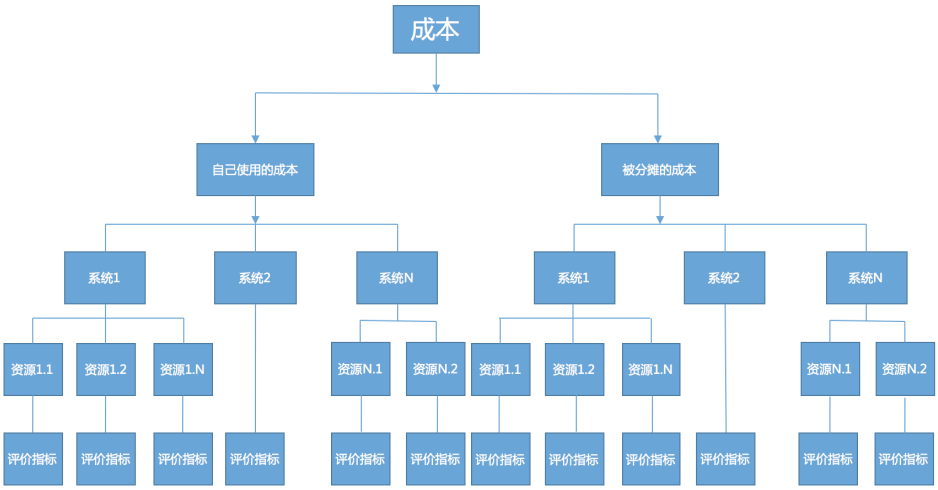
在执行阶段的流程是：分解原子项目 -> 确定方案 -> 落实到人 -> 优化原子指标。在这里包括两个核心要素：1）把核心指标相关的工作向下一层分解；2）在下一层，找到具体的人来执行，这个人要具备将自己负责的指标继续分解到更细的能力，类似于我们说的树状结构。这样层层地分解下去，每一层的叶子节点都可以找到对应的负责人。这种“总分”结构，在一本经典教程《金字塔原理》中也有详细的阐述。

- **分解原子项目**。在本阶段要建立一个完全细化的分级结构，用金字塔原理中的”MECE 不重不漏”原则，将工作内容分解到最细的可控粒度。至于按哪个

维度进行拆分，不同的团队或者业务可能会有不同的原则，比如有些团队直接按子团队进行拆分，有些团队按业务进行拆分，有些团队按流程进行拆分。从较多团队通用的角度，成本控制这件事，可以简单的将指标分解到二级指标，包括“自身使用的成本”和“被分摊的成本”。其中，“自身使用的成本”是指，为了满足自己业务的需要，由本技术团队申请或者使用资源产生的成本；“被分摊的成本”是指，由于根据某种计算逻辑，间接使用了其他团队的资源，为其他技术团队承担一部分成本费用，比如常见的资源包括公司其他团队开发的广告、投放、风控、安全等系统。如果可以分拆到具体的系统，则每个系统又可以继续向下拆分到更细粒度的构成项目，每个节点都是一个小的“总分”结构，按这个逻辑继续向下分解，可以分为“可落地的最细粒度的成本”和“可落地的最细粒度的分摊成本”。



再根据开篇描述的方法，确定每个原子的评价指标，无法量化的项目都是“耍流氓”。这样就形成了一个更完整的金字塔结构，如下图所示：

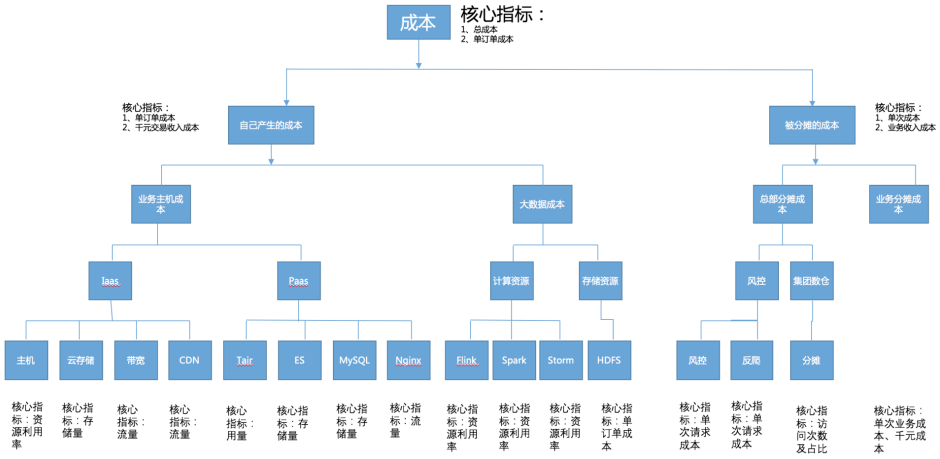


- **确定方案**。根据上面的金字塔结构，每个原子指标，都需要专业的同学来评价分析，确定如何进行优化。比如，系统主机的成本，主要集中在虚拟机 + 存储这样的资源上，衡量的指标可以确定为“资源利用率”和“单订单成本”，为了解决“资源利用率”这个原子指标，就需要考虑目前的空闲机器是否可以下线，在线的服务是否可以优化或者合并；为了解决“单订单成本”这个指标，可以考虑分析下系统架构，跟核心流程处理有关的服务是否可以更加高效或者抽象出来成为服务中台，这样就可以释放一些”烟囱式”的建设资源，使得核心处理能力更加集中、高效。类似这样将所有的解决方案整合起来，就形成了最后的解决方案。
- **落实到人**。有了方案之后，一定要确定唯一的 Owner（主 R），根据经验，主 R 只有一个会比较好，否则会造成“责”、“权”、“利”分割不清。在这个过程中，也是培养团队技术能力和架构能力的好机会。
- **优化指标**。不同的方案，实施的周期和代价不同，各个主 R 深入到不同专业后，会对目前的资源指标有分析和反馈，有可能理论上所有的指标都需要优化，也有可能一些指标已经很好了，这时候要甄别出来哪些资源指标的实施“杠杆率”比较高，建议应用 80/20 原则进行分析，即某些指标投入 20% 的资源和精力可以解决最后 80% 的核心问题，保证投入适合的工作量带来较高

的产出。对于没有解决方案的资源或者实施难度过大的资源，建议果断放弃或者搁置。

3.2 实践分析框架

在具体实践中，我们可以把以上的过程，再次用一个金字塔结构来表述，如下图所示：



建立了以上的结构，就可以根据各个专业的不同，对各自的指标进行优化了，如果最细一级的指标被成功优化之后，最上层的指标一定会有下降。因为上述指标都有其各自深层次的业务、技术，甚至是财务上的逻辑，故在此把一些需要关注的概念再赘述一下 .

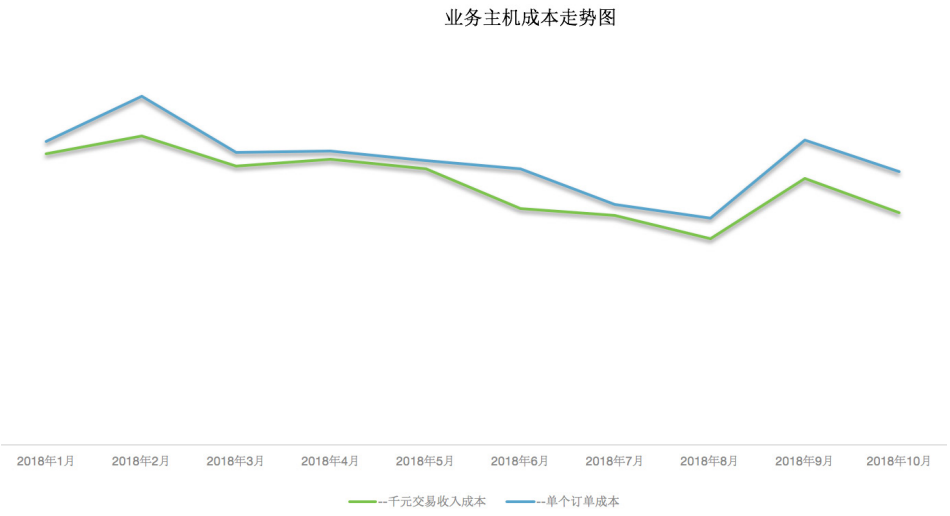
很多公司每个技术团队的机器成本，在财务上叫做“网站运维成本”（网站？听起来还像 PC 时代的概念对不对），从顶层可以分为两类构成因素，就是“自己产生的成本”（自己用的）和“被分摊的成本”（别人替你用的）两大类。跟自己有关的继续向下钻取，可以分为交易相关的资源成本（跟业务流程相关的）以及跟分析有关的大数据成本（分析、算法、决策相关）。

3.2.1 业务主机成本

大部分业务系统的团队，使用的资源成本都包含在这个部分，比如商户研发团队、订单系统研发团队、前端研发团队、供应链研发团队、营销系统研发团队、

CRM 研发团队等。这些资源典型的物理载体就是物理机、虚拟机、容器资源以及对应的机器连接的存储（DB、缓存、K-V 数据库等）资源，还会包含由于交换、存储以上资源之间的数据产生的带宽、云资源、CDN 等。

这部分资源，我们从控制成本的角度，最浅的层次，建议关注服务组（OWT）所消耗主机的资源利用率，如果资源利用率较低的主机数量较多，建议及时下线。同时，从技术方案本身来说，任何一个服务承载的业务能力和消耗资源之间，会有相对的一个“比例”或者权重。某些高利用率的服务从架构上是否可以重构、解耦或者改造，也非常有利于节省资源。这块内容到餐技术部在过去一年的工作中，对于核心、非核心的服务都进行了梳理，对于其中可以优化的服务也进行了部分重构。相比年初，很好的降低了资源的成本，业务主机成本的两个主要指标的变化情况如下（备注，后续由于新增其他业务导致成本略有上升）：



3.2.2 大数据成本

数据行业在互联网的应用目前已经较为成熟，行业主流的数据处理架构都是 Yarn 2.0 或者类似框架，核心的资源消耗主要基于 Container（Vcore+Mem）的计算资源 + 基于 HDFS 的存储资源消耗这两部分：

第一部分，是存储资源的消耗，行业通用的模型是基于物理 HDFS 或自研的类

似存储引擎，这部分主要是指离线 ETL 用来按分区（一般是按时间戳）进行存储的资源，由于数据仓库的核心理念之一是保存“所有”的数据，并在此基础上按照维度建模理论对数据进行预汇总、加和。但是，由于对于模型建设本身的理解深度不同，故在基础数据之上的数据冗余，在很多数据研发人员看来是理所应当的，进而导致存储资源的快速膨胀，这是每个数据团队在管理过程中面临的难题。

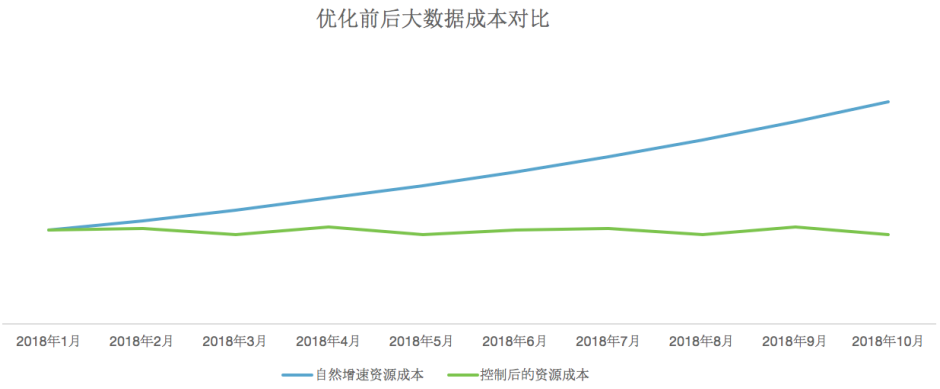
在此，到餐研发团队主要采取了两种手段：

1. 对于数据模型的热度进行了分级，把数据分为冷、温、热数据，对于需要保留的数据才保存在生产环境的 HDD、SSD 中，对于不重要的冷数据，通过异构的方式存入其他介质中。
2. 对于数据模型本身，需要重新思考数据的价值和存储，在数据的中间层（汇聚层），对数据模型进行重构，这也是很多数据团队忽略的基本功部分。

到餐数据团队对于数据仓库进行了二次迭代，每次都基于新的业务模式，重新构建中间层以及之上的集市、宽表层，有效节省了空间。还有一种技术手段是压缩，比如流量的数据往往是存储大户，但是流量数据相对的格式比较固定，所以很多流量数据可以进行压缩或者改变其存储格式（如 map 型），根据实测可以节省 20% 以上的流量数据空间。

另外需要补充的，还有一部分 OLAP 存储资源，也会消耗大量资源，比如 Kylin、Elasticsearch、Druid、MySQL 等，这些数据库主要用来将基于 HDFS 上的文件，同步到前端可以直接访问的介质上，供系统访问。这部分资源有些也是基于 HDFS 的（如 Kylin、HBase），有些需要单独的存储介质，也需要关注其膨胀速度以及存储周期。

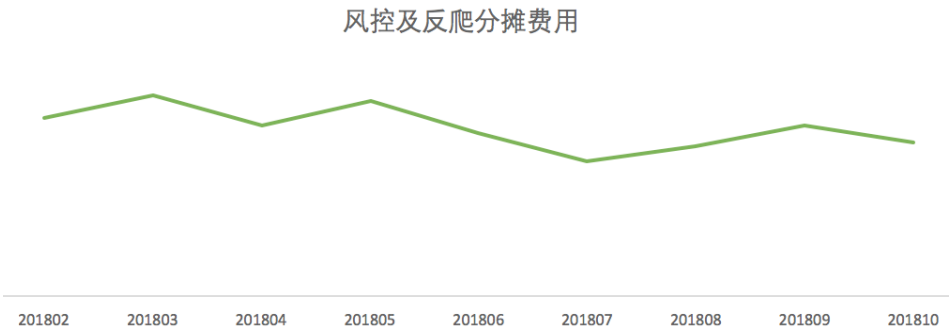
第二部分，是计算资源的消耗，主要满足基于复杂规则的分析或者机器学习算法中的计算，也就是实时 ETL 计算和离线 ETL 计算的场景（代表性的引擎如 Storm、Flink 的计算还有 MapReduce 的计算）。这部分计算消耗的资源类似于业务系统，可以参照业务系统的“资源利用率”确定几个指标，进行机器优化或者算法逻辑优化。



3.2.3 分摊成本（一）风控及反爬

在某些公司里，某个技术团队开发的内容，有可能为了服务其他团队业务，比如前文中提到的风控、反爬、广告等，会为各种业务提供基础的技术能力。这时候就涉及到一个重要的概念“分摊”。分摊有两种规则，一种是按“实际用量进行”，另外一种是按照“使用比例”进行，这两种模式之上，可能还有混合计费模式，即“按照实际发生的比例进行整体费用的分摊”，做成本控制时，就要清楚地知道这部分成本是按哪种逻辑来进行计算的。

在风控及反爬的实践中，美团的风控及反爬按照整体风控技术团队的总体成本，按比例分摊给业务团队。所以作为业务团队，如果试图降低这部分成本，也要关注两个组成项：一是自己使用的风控及反爬的原子业务数量的绝对值，对每天风控及反爬的总体请求次数是否合理需要进行判断，以保证自己的业务请求量不增加；二是自己业务使用的比例。需要跟相关技术团队一起进行分析，以防止某些场景下，自身业务使用的绝对值下降了，但是因为其他业务绝对值下降的更快，导致自己比例反而上升，进而导致成本上升。

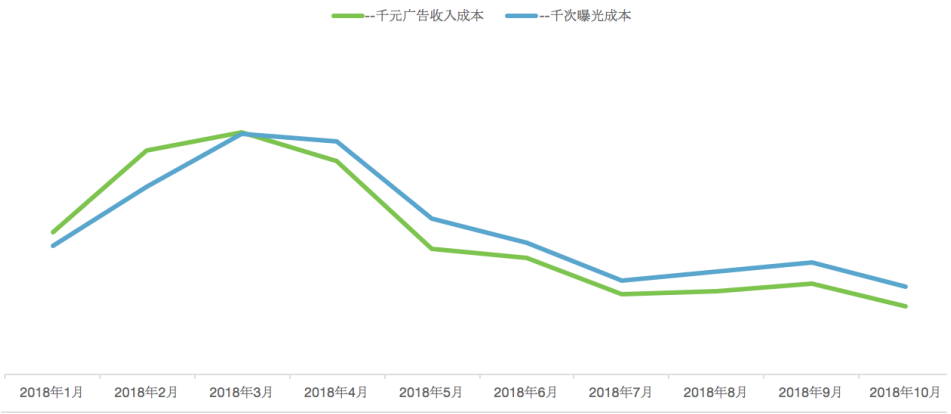


3.2.4 分摊成本（二）安全数仓成本

为了保证各个业务团队之间的离线数据交换，美团集团层面建设了安全数据仓库，用来满足跨团队之间的数据交换。这部分的费用也按照实际发生的资源占比进行统计，所以同理，为了降低成本，需要关注两个组成项目：一是自己使用的数量，从架构设计上能否将相关数据模型的效率提升、降低空间是关键因素；二是自己的使用资源在整体资源的占比，这时候也需要跟相关团队一起努力降低总成本。很多公司的技术团队，也有类似的数据共享仓库或者共建仓库的概念。

3.2.5 分摊成本（三）广告成本

很多互联网公司都有做广告业务的技术团队，广告的形式主要有按点击收费 CPC，按时长收费 CPT 等等，这部分分摊的逻辑同上述两者，也是按最终的总费用中的占比进行分摊。但是这块有一个需要关注的点是，由于广告的业务逻辑并不在到餐自己的业务方，也就是说归到餐研发团队可以控制的部分较小，故在这个过程中需要建立有效的评价体系，来衡量广告分摊的费用，在此采用的指标是“千次曝光成本”和“千元广告收入成本”，这里仅供大家参考。



3.2.6 其他成本

除了以上梳理的项目之外，每月还会有一些新增的成本项目加入进来，团队要保持足够的关注。在实践中会发现某项成本在个别月份突然升高，这时候就要找到是新增加了项目，还是某个指标在业务或者算法上有所调整。

4. 检查（Check）

在这个阶段，建议关注以下结果：

- **规定动作检查**。规定的方案是否执行？相关的同学是否按照规定的动作进行了相对应的行动？这个阶段只关注过程不关注结果，而且更多的是关注执行人、配合方、时间点，用项目管理的思路来运营。
- **结果评估**。之前梳理出来的指标是否得到了优化？这个过程是在验证结果，各项指标中得到优化和未优化的都要整理出详细的 List，有些指标如“资源利用率”是立即可以查看结果的，有些结果是需要周期性的时间才能获得。在这个基础上可以继续深入反向思考，按“指标定义是否有问题 -> 方案制定是否有问题 -> 执行人是否有问题 -> 配合方是否有问题”这个流程来进行评估。
- **系统问题定位**。在这个过程中，可以做到小范围闭环，建议针对某个指标的优化方案可以设计多套，方案 A 不行马上迭代成方案 B，快速试错，找到合理的方案。

- **修正标准动作**。在执行的过程中，很多方案和动作，都是在一线现场发现和修正的，不需要等待大规模复盘的时候再提出问题和总结，主 R 要具备这样的意识，在执行过程中多说多问，找到关键要素，相信每个同学都有过这样的经历。经历过某个完整项目生命周期的同学，往往也是团队内成长最快的骨干。

在到餐研发团队的实践中，业务系统的指标定义上也有类似的经验可以分享。开始进行优化工作的时候，定义了非常多的的项目和指标，比如业务主机分为云存储、带宽、CDN、Tair、Redis 等等，关注到每一项对于 RD 投入的时间和精力都是巨大的损耗，后来经过反复跟相关兄弟团队确认，向上抽象了一层“服务组的资源利用率”，这时候就不需要关注太多细碎的项目，而只关注与这些服务有关机器的使用情况，因为机器会自然的消耗 CPU、内存、带宽、CDN 等，这样可以有效节省运营的时间成本，把精力集中在优化机器和优化服务架构设计层面。

5. 复盘总结，继续迭代 (Act)

- **定期复盘**。复盘是一个非常重要的能力，个人以为，复盘总结的能力在某种程度上也代表了自己的“抽象能力 + 思考能力 + 管理能力”，关于复盘的方法论书籍很多，这里不再进行赘述。在这个阶段，个人建议关注的点在于两个“知道”：“知道自己不知道”，通过复盘掌握了成本优化的方法、框架、方案、团队素质、结果；“不知道自己原来知道”，通过一些结果，知道了自己原来一直是在正确的道路上还是在错误的道路上前进，把带有“运气”成分的成功，升华成为一种未来的“习惯性成功”。
- **形成报告**。让第一次看到这个报告的人，也能通过 1–2 次实践，学会成本优化这件事。
- **迭代认知**。将之前的过程开始深化和迭代，也是再次进行 PDCA 的过程，反复打磨自己的抽象能力、思考能力、管理能力，使自己工作深度、广度的 ROI 继续提升。在迭代过程中，总会有一些惊喜和收获。从个人来说，原来以为成本项目仅仅是个管理项目，在不断通过技术手段取得成本优化的过程中，收获了对架构、技术的理解，并且很多时候需要用创新的手段来解决前人未曾突破

的问题，另外还收获了 7 项跟架构升级、数据压缩、技术处理有关技术专利，也是技术能力提升的一个佐证。

总结

成本优化这件事，有可能被阶段性忽略，但是重要性一直存在。到餐研发团队通过将近一年时间的运营，帮助公司节省了几千万的成本。这个过程有时候枯燥，有时候让人兴奋，有时候又让人懊恼和沮丧，某些时候其实是在拷问自己一个问题：“保证业务不停的前提下，敢砍掉多余的机器吗?”在管理越来越精细化的今天，相信更多的有识之士也有一些需求或者进行了一些实践。期待跟行业同侪一起，在保证技术能力和满足业务的前提下，更加合理使用资源，节约公司成本，不断提升研发团队的效率，希望本文能给大家带来一些启发。

作者简介

刘强，美团到店餐饮研发中心数据方向负责人，美团数据技术通道委员，2017 年入职美团点评，就职于到店餐饮研发中心，负责到餐数据仓库、数据产品、数据系统的研发工作。之前曾任多家公司的数据方向负责人。

建钟、小英、杨轩、云杰、方旭、鹏文，均为美团到餐研发团队工程师，对本文均有贡献。

特别感谢

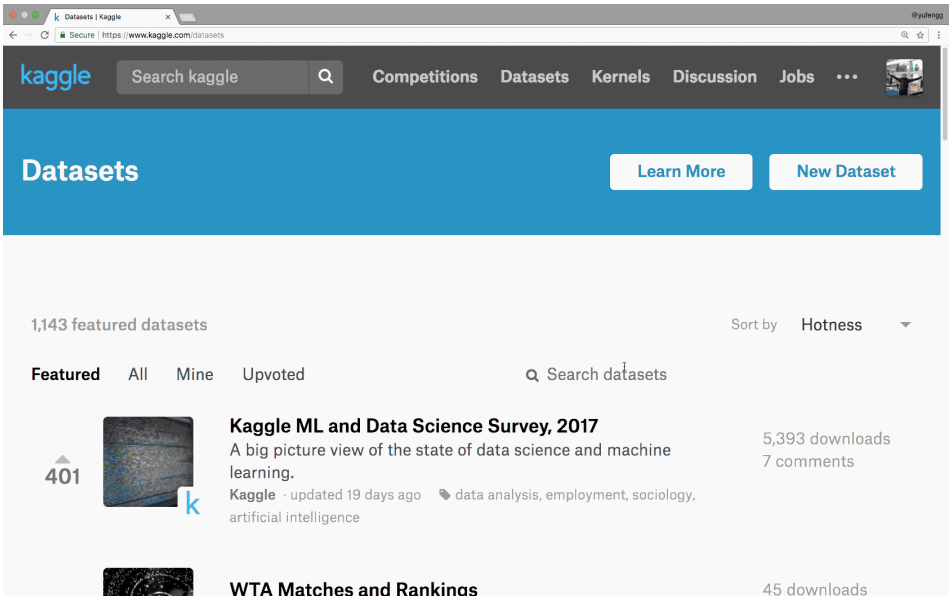
在本文及成本优化过程中，得到了美团技术团队李伟、任登君、李闻、谢语宸、洪丹、左普存、郭树熠、刁士涵等人的支持和帮助，在此表示感谢！

Jupyter 在美团民宿的应用实践

文龙　颖艺

前言

做算法的同学对于 Kaggle 应该都不陌生，除了举办算法挑战赛以外，它还提供了一个学习、练习数据分析和算法开发的平台。Kaggle 提供了 Kaggle Kernels，方便用户进行数据分析以及经验分享。在 Kaggle Kernels 中，你可以 Fork 别人分享的结果进行复现或者进一步分析，也可以新建一个 Kernel 进行数据分析和算法开发。Kaggle Kernels 还提供了一个配置好的环境，以及比赛的数据集，帮你从配置本地环境中解放出来。Kaggle Kernels 提供给你的是一个运行在浏览器中的 Jupyter，你可以在上面进行交互式的执行代码、探索数据、训练模型等等。更多关于 Kaggle Kernels 的使用方法可以参考 Introduction to Kaggle Kernels，这里不再多做阐述。



对于比赛类的任务，使用 Kaggle Kernels 非常方便，但我们平时的主要任务还

是集中在分析、处理业务数据的层面，这些数据通常比较机密并且数量巨大，所以就不能在 Kaggle Kernels 上进行此类分析。因此，大型的互联网公司非常有必要开发并维护集团内部的一套「Kaggle Kernels」服务，从而有效地提升算法同学的日常开发效率。

本文我们将分享美团民宿团队是如何搭建自己的「Kaggle Kernels」—— 一个平台化的 Jupyter，接入了大数据和分布式计算集群，用于业务数据分析和算法开发。希望能为有同样需求的读者带来一些启发。

美团内部数据系统现状

现有系统与问题

算法同学在离线阶段主要包含三类任务：数据分析、数据生产、模型训练。为满足这些任务的要求，美团内部也开发了相应的系统：

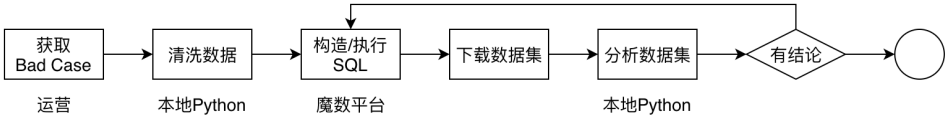
- 魔数平台：用于执行 SQL 查询，下载结果集的系统。通常在数据分析阶段使用。
- 协同平台：用于使用 SQL 开发 ETL 的平台。通常用于数据生产。
- 托管平台：用于管理和运行 Spark 任务，用户提供任务的代码仓库，系统管理和运行任务。通常用于逻辑较复杂的 ETL、基于 Spark 的离线模型训练 / 预测任务等。
- 调度平台：用于管理任务的依赖关系，周期性按依赖执行调度任务。

这些系统对于确定的任务完成的比较好。例如：当取数任务确定时，适合在魔数平台执行查询；当 Spark 任务开发就绪后，适合在托管平台托管该任务。但对于探索性、分析性的任务没有比较好的工具支持。探索性的任务有程序开发时的调试和对陌生数据的探查，分析性的任务有特征分析、Bad Case 分析等等。

以数据探索为例，我们经常需要对数据进行统计与可视化，现有的做法通常是：魔数执行 SQL -> 下载 Excel -> 可视化。这种方式存在的问题是：

- 分析和取数工具割裂。
- 大数据分析可视化困难。

以 Bad Case 分析为例，现有的做法通常是：



这种方式存在的问题是：

- 分析与取数割裂，整个过程需要较多的手工操作。
- 分析过程不容易复现，对于多人协作式的验证以及进一步分析不利。
- 本地 Python 环境可能与分析对象的依赖有冲突，需要付出额外精力管理 Python 环境。

离线数据相关任务的模式通常是取数（小数据 / 大数据）-> Python 处理（单机 / 分布式）-> 查看结果（表格 / 可视化）这样的循环。我们希望支持这一类任务的工具具有如下特质：

- 体验流畅：数据任务可以在统一的工具中完成，或者在可组合的工具链中完成。
- 体验一致：数据任务所用工具应该是一致的，不需要根据任务切换不同工具。
- 使用便捷：工具应是开箱即用，不需要繁琐的前置配置。
- 结果可复现：分析过程能够作为可执行代码保存下来，需要复现时执行即可，也应支持修改。

探索和分析类任务往往会带来可以沉淀的结果，如产生新的特征、模型、例行报告，希望可以建立起分析任务和调度任务的桥梁。

我们需要怎样的 Jupyter

参考 Kaggle Kernels 的体验和开源 Jupyter 的功能，Notebook 方式进行探索分析具有良好的体验。我们计划定制 Jupyter，使其成为完成数据任务的统一工具。

这个定制的 Jupyter 应具备以下功能：

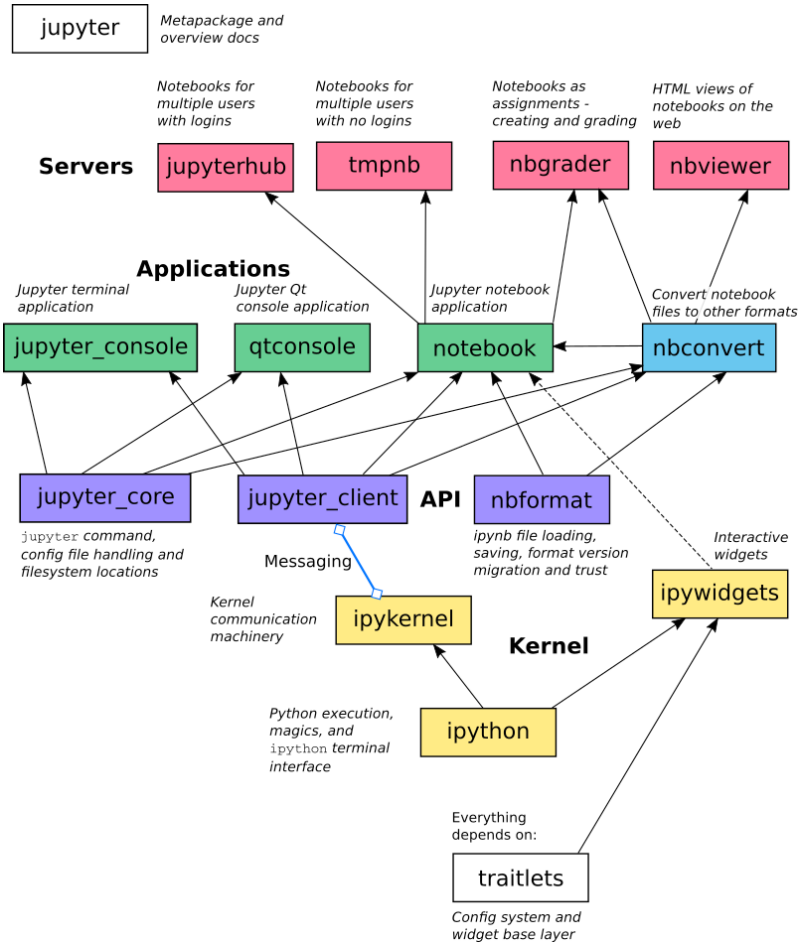
- 接入 Spark：取数与分析均在 Jupyter 中完成，达到流畅、一致的体验。

- 接入调度系统：方便沉淀分析结果。
- 接入学城系统（内部 WiKi）：方便分享和复现。
- 预配置环境：提供给用户开箱即用的环境。
- 用户隔离环境：避免用户间互相污染环境。

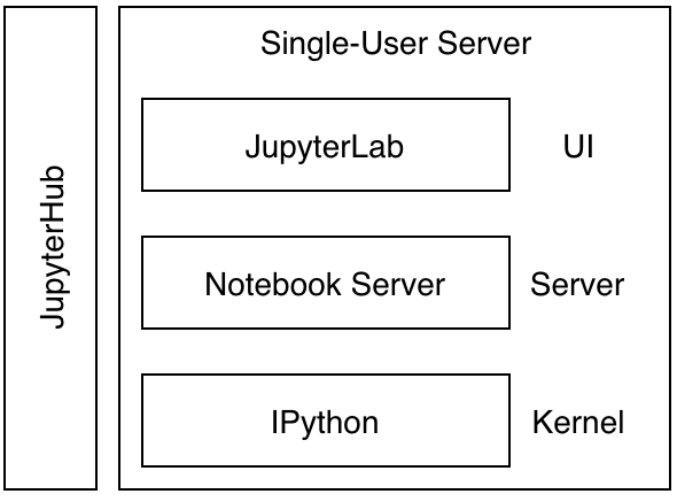
如何搭建 Jupyter 平台

Jupyter 项目架构

Project Jupyter 由多个子项目组成，通过这些子项目可以自由组合出不同的应用。子项目的依赖关系如下图所示：



这个案例中，Jupyter 应用是一个 Web 服务，我们可以从这个维度来看 Jupyter 架构：

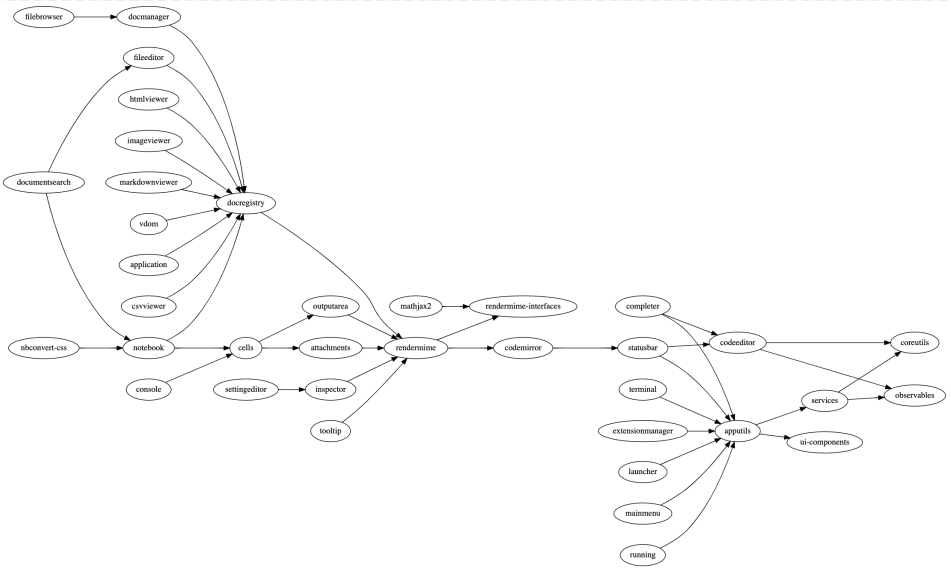


Jupyter 扩展方式

整个 Jupyter 项目的模块化和扩展性上都非常优秀。上图中的 JupyterLab、Notebook Server、IPython、JupyterHub 都是可扩展的。

JupyterLab 扩展（labextension）

JupyterLab 是 Jupyter 全新的前端项目，这个项目有非常明确的扩展规范以及丰富的扩展方式。通过开发 JupyterLab 扩展，可以为前端界面增加新功能，例如新的文件类型打开 / 编辑支持、Notebook 工具栏增加新的按钮、菜单栏增加新的菜单项等等。JupyterLab 上的前端模块具有非常清楚的定义和文档，每个模块都可以通过插件获取，进行方法调用，获取必要的信息以及执行必要的动作。我们在提供分享功能、调度功能时，均开发了 JupyterLab 扩展。JupyterLab 扩展通常采用 TypeScript 开发，开发文档可参考：https://jupyterlab.readthedocs.io/en/stable/developer/extension_dev.html。



JupyterLab 核心组件依赖图

Notebook Server 扩展（serverextension）

Notebook Server 是用 Python 写的一个基于 Tornado 的 Web 服务。通过 Notebook Server 扩展，可以为这个 Web 服务增加新的 Handler。增加新的 Handler 通常有两种用途：

1. 为 JupyterLab 扩展提供对应的后端接口，用于响应一些需要由服务端处理的事件。例如调度任务的注册需要通过 JupyterLab 扩展发起请求，由 Notebook Server 扩展执行。
2. 提供一个前端界面以及对应的后端处理服务。例如 jupyter-rsession-proxy，用于在 JupyterHub 中使用 RStudio。

Notebook Server 扩展开发文档可参考：https://jupyter-notebook.readthedocs.io/en/stable/extending/handlers.html。

##3# Jupyter Kernels

Jupyter 用于执行代码的模块叫 Kernel，除了默认的 ipykernel 以外，还可以有

其他的 Kernel 用于支持其他编程语言。例如支持 Scala 语言的 almond、支持 R 语言的 irkernel，更多详见语言支持列表。

IPython Magics

IPython Magics 就是那些 %、%% 开头的命令。常见的 Magics 有 %matplotlib inline，设置 Notebook 中调用 matplotlib 的绘图函数时，直接展示图表在 Notebook 中。执行 Magics 时，事实上是调用了该 Magics 定义的一个函数。对于 Line Magics（一个 %），传入函数的是当前行的代码；对于 Cell Magics（两个 %），传入的是整个 Cell 的内容。定义一个新的 IPython Magics 仅需定义一个函数，这个函数的入参有两个，一个是当前会话实例，可以用来遍历当前会话的所有变量，可以为当前会话增加新的变量；另一个是用户输入，对于 Line Magics 是当前行，对于 Cell Magcis 是当前 Cell。

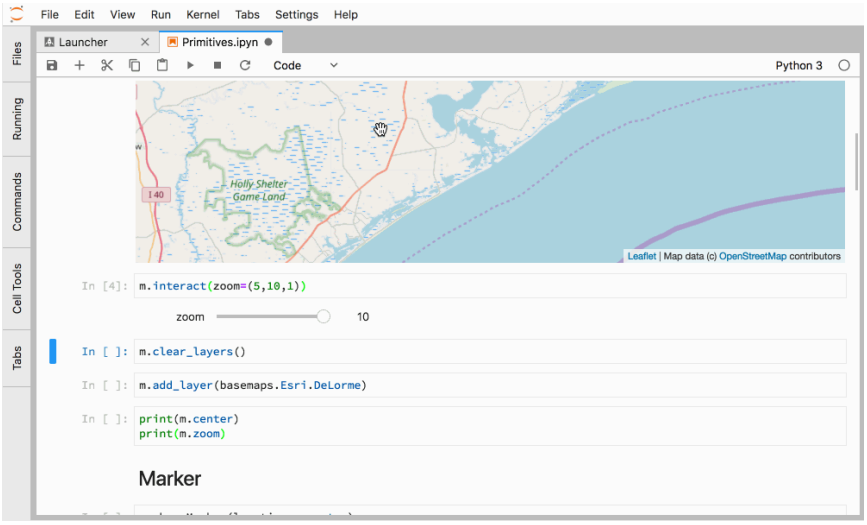
IPython Magics 在简化代码方面非常有效，我们开发了 %%spark、%%sql 用于创建 Spark 会话以及 SQL 查询。另外很多第三方的 Magics 可以用来提高我们的开发效率，例如在开发 Word2Vec 变种时，使用 %%cython 来进行 Cython 和 Python 混合编程，省去编译加载模块的工作。

IPython Magics 开发文档可参考：https://ipython.readthedocs.io/en/stable/config/custommagics.html。

IPython Widgets（ipywidgets）

IPython Widgets 是一种基于 Jupyter Notebook 和 IPython 的可交互控件。与普通可视化不同的是，在控件上的交互会触发和 Python 的通信并执行相应的代码，Python 上相应的动作也会触发界面实时变化。

IPython Widgets 在提供工具类型的功能增强上非常有用，基于它，我们实现了一个线上排序服务的调试和复现工具，用于展示排序结果以及指定房源在排序过程中的各种特征以及中间变量的值。IPython Widgets 的开发可以通过组合现有的 Widgets 实现，也可以完全自定义一个，IPython Widgets 开发文档可参考：https://ipywidgets.readthedocs.io/en/stable/examples/Widget%20Custom.html。



ipyleaflet

扩展 JupyterHub

Authenticators

JupyterHub 是一个多用户系统，登录模块可替换，通过实现新的 Authenticator 类并在配置文件中指定即可。通过这个扩展点，我们实现了使用内部 SSO 系统登录 JupyterHub。Authenticator 开发文档可参考：https://jupyterhub.readthedocs.io/en/stable/reference/authenticators.html。

Spawners

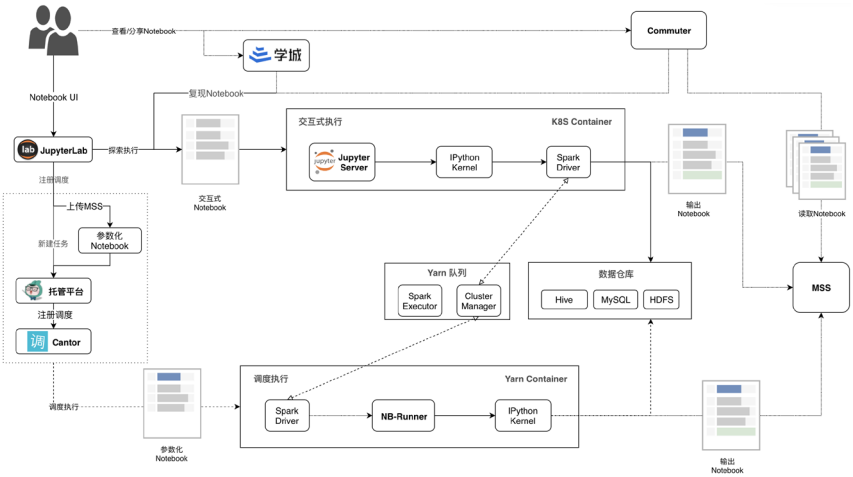
当用户登录时，JupyterHub 需要为用户启动一个用户专用 Notebook Server。启动这个 Notebook Server 有多种方式：本机新的 Notebook Server 进程、本机启动 Docker 实例、K8s 系统中启动新的 Pod、YARN 中启动新的实例等等。每一种启动方式都对应一个 Spawner，官方提供了多种 Spawner 的实现，这些实现本身是可配置的。如果不符合需求，也可以自己开发全新的 Spawner。由于我们需要实现 Spark 接入，对 K8s 的 Pod 有新的要求，所以基于 KubeSpawner 定制了一个 Spawner 来解决 Spark 连接集群的网络问题。Spawner 开发文档可参考：https://jupyterhub.readthedocs.io/en/stable/reference/spawners.html。

我们的定制

回顾我们的需求，这个定制的 Jupyter 应具备以下功能：

- 接入 Spark：可以通过配置容器环境以及 Spawner 完成。
- 接入调度系统：需要开发 JupyterLab 扩展以及 Notebook Server 扩展。
- 接入学城系统：需要开发 JupyterLab 扩展以及 Notebook Server 扩展。
- 预配置环境：镜像配置。
- 用户隔离环境：通过定制 Authenticators + K8s Spawner 实现容器级别环境隔离。

我们的方案是基于 JupyterHub on K8s。下图是平台化 Jupyter 的架构图，从上到下可以看到三条主线：1. 分享复现、2. 探索执行、3. 调度执行。



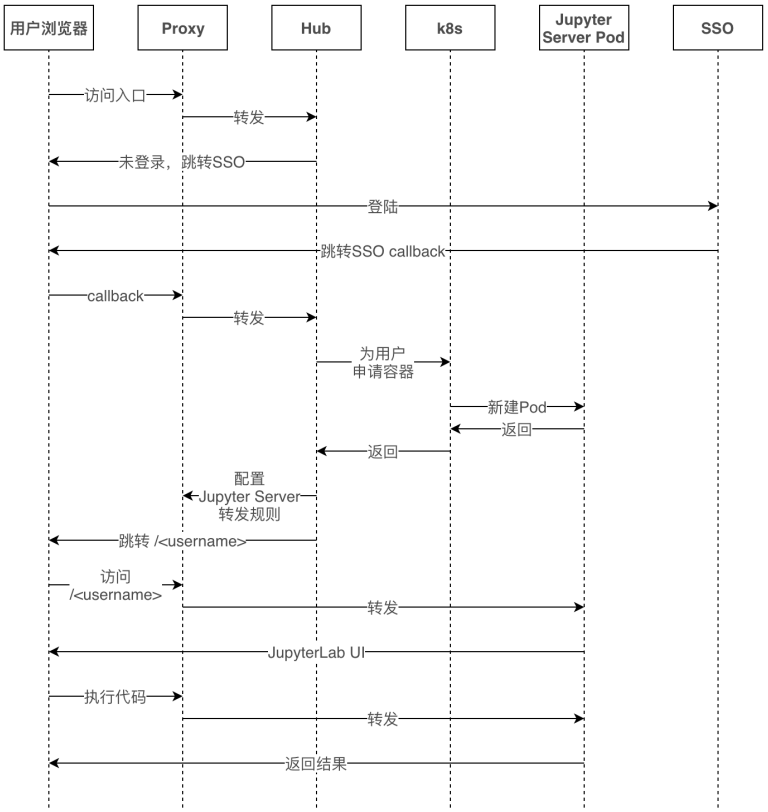
几个关键组件介绍：

- JupyterLab：交互式执行的前端，开源项目。
- Jupyter Server：交互式执行的后端，开源项目。
- Commuter：浏览 Notebook 的工具，开源项目。
- K8s：容器编排系统，开源项目。
- Cantor：美团调度系统，同类开源项目有 AirFlow。

- 托管平台：美团离线任务托管平台，给定代码仓库和任务参数，为我们执行 Spark-Submit 的平台。
- 学城：美团文档系统。
- MSS：美团对象存储。
- NB-Runner：Notebook Runner，在 nbconvert 的基础上增加了参数化和 Spark 支持。

在定制 Jupyter 中，最为关键的两个是接入 Spark 以及接入调度系统，下文中将详细介绍这两部分的原理。

JupyterHub on K8s 包括几个重要组成部分：Proxy、Hub、Kubernetes、用户容器（Jupyter Server Pod）、单点登录系统（SSO）。一个用户在登录后新建容器实例的过程中，这几个模块的交互如下图所示：



可以看到，新建容器实例后，用户的交互都是经过 Proxy 后与 Jupyter Server Pod 进行通信。因此，扩展功能的工作主要是定制 Jupyter Server Pod 对应的容器镜像。

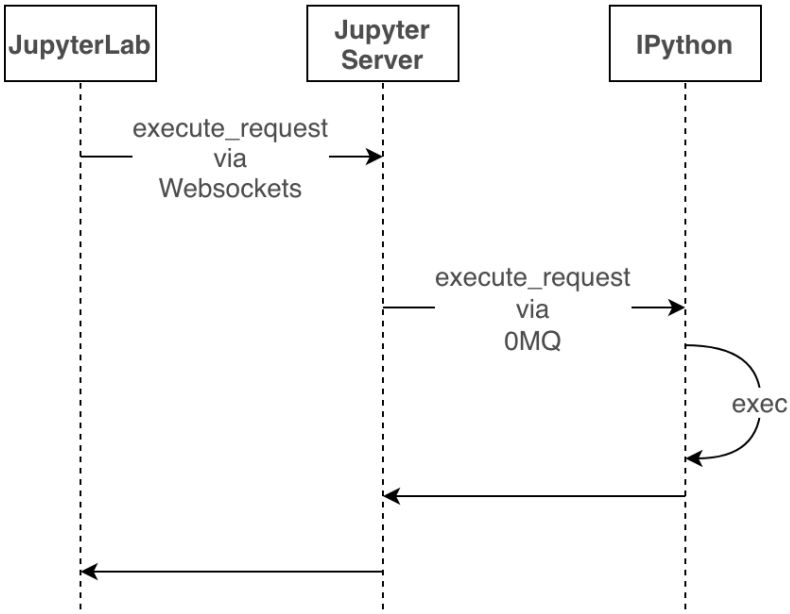
让 Jupyter 支持 Spark

Jupyter 平台化后，我们得到一个接近 Kaggle Kernel 的环境，但是还不能够使用大数据集群。接下来，就是让 Jupyter 支持 Spark，Jupyter 支持 Spark 的方案有 Toree，出于灵活性考虑，我们没有使用。我们希望让普通的 Python Kernel 能支持 PySpark。

为了能让 Jupyter 支持 Spark，我们需要了解两方面原理：Jupyter 代码执行原理和 PySpark 原理。

#3## Jupyter 代码执行原理

所用到的 Jupyter 分三部分：前端 JupyterLab、服务端 Jupyter Server、语言 Kernel IPython。这三个模块的通信如下图所示：



Jupyter 执行代码时序图

这里，需要在 IPython 的 exec 阶段支持 PySpark。

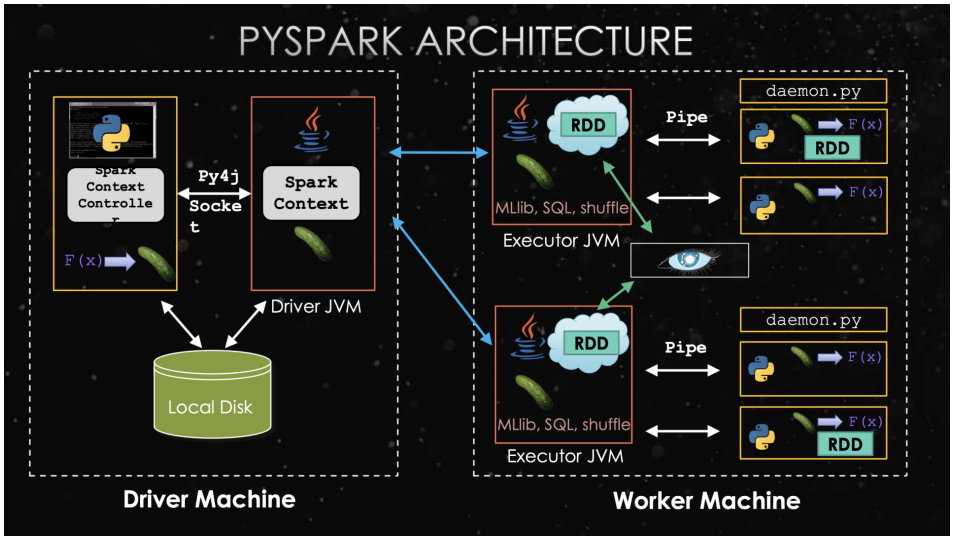
##3# PySpark 原理

启动 PySpark 有两种方式：

- 方案一：PySpark 命令启动，内部执行了 spark-submit 命令。
- 方案二：任意 Python shell（Python、IPython）中执行 Spark 会话创建语句。

这两种启动方式有什么区别呢?

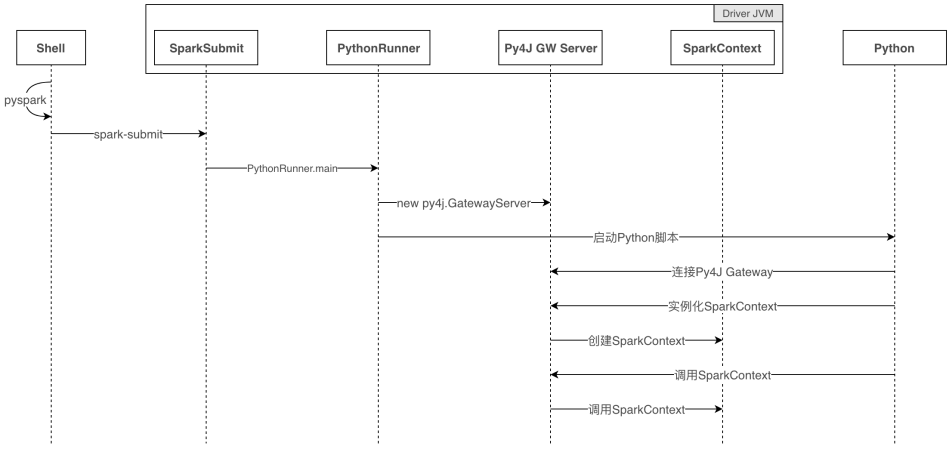
看一下 PySpark 架构图：



PySpark 架构图，来自 SlideShare

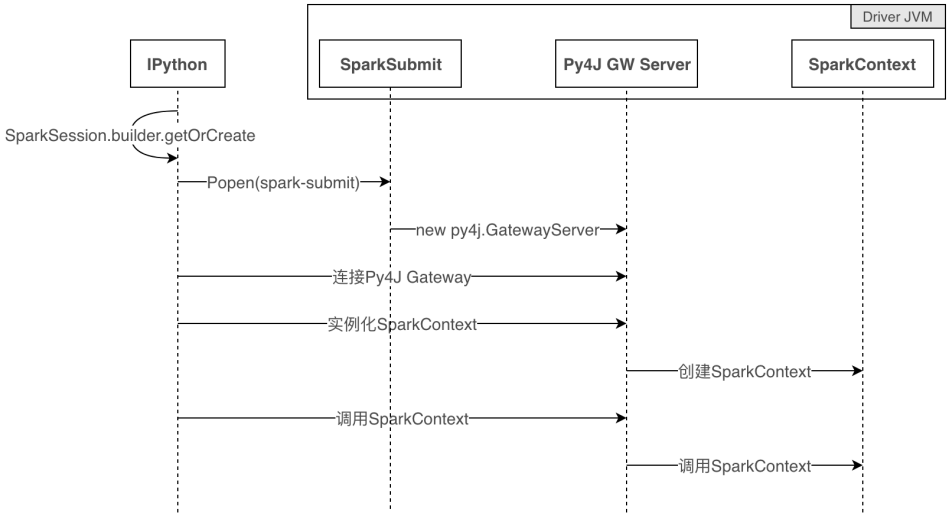
与 Spark 的区别是，多了一个 Python 进程，通过 Py4J 与 Driver JVM 进行通信。

PySpark 方案启动流程



PySpark 启动时序图

IPython 方案启动流程



实际的 IPython 中启动 Spark 时序图

Toree 采用的是类似方案一的方式，脚本中调用 spark-submit 执行特殊版本的 Shell，内置了 Spark 会话。我们不希望这么做，是因为如果这样做的话就会：

- 多了一个 PySpark 专供的 Kernel，我们希望 Kernel 应该是统一的 IPython。
- PySpark 启动参数是固定的，配置在 kernel.json 里。希望 PySpark 任务是可以按需启动，可以灵活配置所需的参数，如 Queue、Memory、Cores。

因此我们采用方案二，只需要一些环境配置，就能顺利启动 PySpark。另外为了简化 Spark 启动工作，我们还开发了 IPython 的 Magics，%spark 和 %sql。

环境配置

为了让 IPython 中能够顺利启动起 Spark 会话，需要正确配置如下环境变量：

- JAVA_HOME：Java 安装路径，如 /usr/local/jdk1.8.0_201。
- HADOOP_HOME：Hadoop 安装路径，如 /opt/hadoop。
- SPARK_HOME：Spark 安装路径，如 /opt/spark-2.2。
- PYTHONPATH：额外的 Python 库路径，如 $SPARK_HOME/python:$SPARK_HOME/python/lib/py4j-0.10.4-src.zip。
- PYSPARK_PYTHON：集群中使用的 Python 路径，如 ./ARCHIVE/notebook/bin/python。集群中使用 Python 通常需要虚拟环境，通过 spark.yarn.dist.archives 带上去。
- PYSPARK_DRIVER_PYTHON：Spark Driver 所用的 Python 路径，如果你用 Conda 管理 Python 环境，那这个变量应为类似 /opt/conda/envs/notebook/bin/python 的路径。

为了方便，建议设置各 bin 路径到 PATH 环境变量中：$SPARK_HOME/sbin:$SPARK_HOME/bin:$HADOOP_HOME/sbin:$HADOOP_HOME/bin:$JAVA_HOME/bin:$PATH。

完成这些之后，可以在 IPython 中执行创建 Spark 会话代码验证：

```
import pyspark
spark = pyspark.sql.SparkSession.builder.appName("MyApp").getOrCreate()
```

在 Spark 任务中执行 Notebook

执行 Notebook 的方案目前有 nbconvert，Python API 方式执行样例如下所示，暂时称这段代码为 NB-Runner.py：

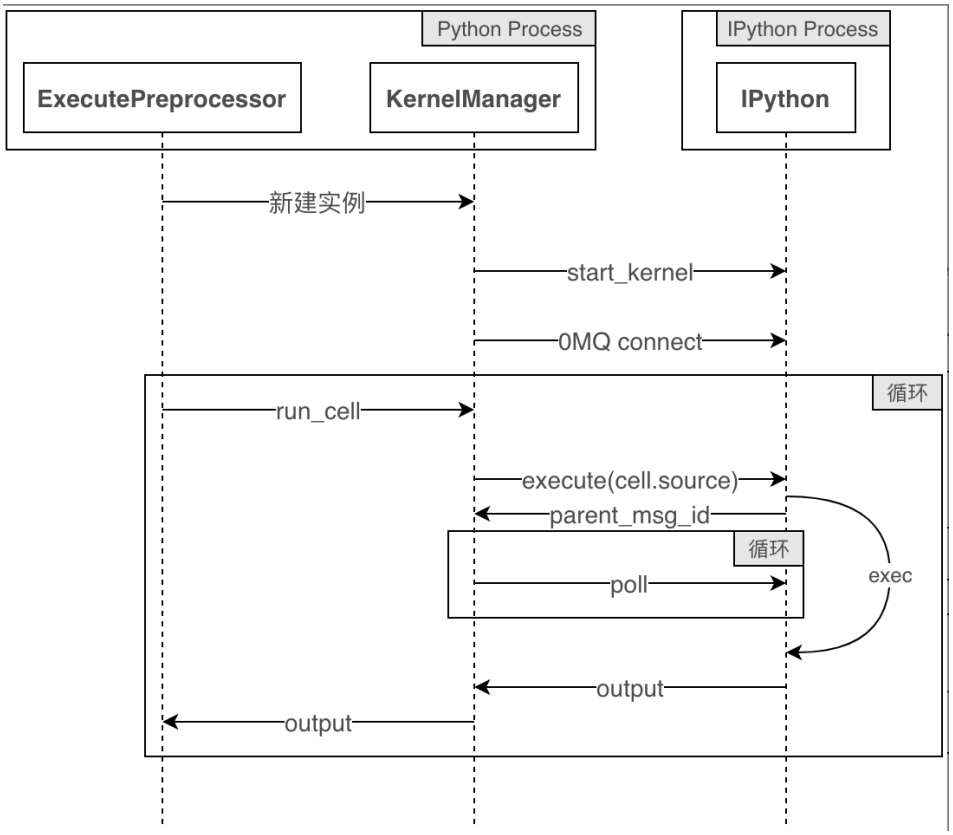
```
# Import：首先我们 import nbconvert 和 ExecutePreprocessor 类：
import nbformat
from nbconvert.preprocessors import ExecutePreprocessor

# 加载：假设 notebook_filename 是 notebook 的路径，我们可以这样加载：
with open(notebook_filename) as f:
    nb = nbformat.read(f, as_version=4)

# 配置：接下来，我们配置 notebook 执行模式：
ep = ExecutePreprocessor(timeout=600, kernel_name='python')

# 执行（preprocess）：真正执行 notebook 的地方是调用函数 preprocess：
ep.preprocess(nb, {'metadata': {'path': 'notebooks/'}})

# 保存：最后，我们保存 notebook 执行结果：
with open('executed_notebook.ipynb', 'w', encoding='utf-8') as f:
    nbformat.write(nb, f)
```

现在有两个问题需要确认：

- 当 Notebook 中存在 Spark 相关代码时，Python NB-Runner.py 能否正常执行?
- 当 Notebook 中存在 Spark 相关代码时，Spark-Submit NB-Runner.py 能否正常执行?

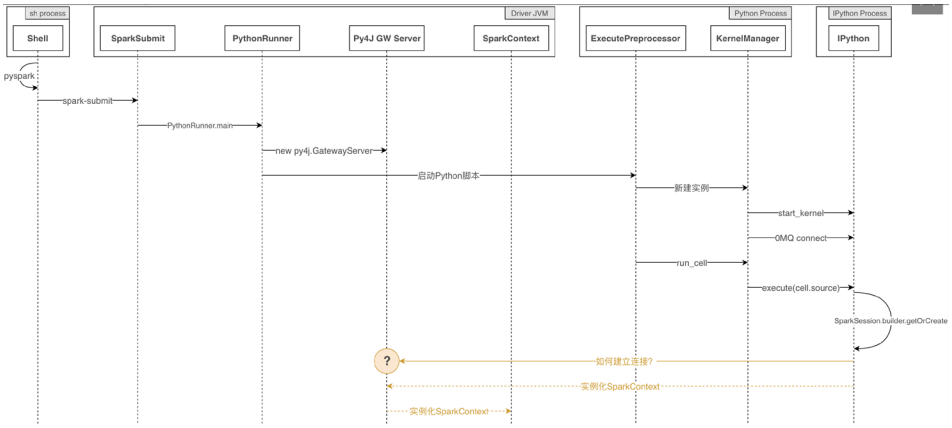
之所以会出现问题 2，是因为我们的调度系统只能调度 Spark 任务，所以必须使用 Spark-Submit 的方式来启动 NB-Runner.py。

为了回答这两个问题，需要了解 nbconvert 是如何执行 Notebook 的。



nbconvert 执行时序图

问题 1 从原理上看，是可以正常执行的。实际测试也是如此。对于问题 2，答案似乎并不明显。结合“PySpark 启动时序图”、“实际的 IPython 中启动 Spark 时序图”与“nbconvert 执行时序图”：



Spark-Submit NB-Runner.py 的方式存在问题的点可能在于，IPython 中执行 Spark.builder.getOrCreate 时，Driver JVM 已经启动并且 Py4J Gateway Server 已经实例化完成。如何让 Spark.builder.getOrCreate 执行时跳过上图“实际的 IPython 中启动 Spark 时序图”的 Popen(spark-submit) 以及后续的启动 Py4J Gateway Server 部分，直接与 Py4J Gateway Server 建立连接?

在 PySpark 代码中，看到如下这段代码：

```
def launch_gateway(conf=None):
    """
    launch jvm gateway
    :param conf: spark configuration passed to spark-submit
    :return:
    """
    if "PYSPARK_GATEWAY_PORT" in os.environ:
        gateway_port = int(os.environ["PYSPARK_GATEWAY_PORT"])
    else:
        SPARK_HOME = _find_spark_home()
        # Launch the Py4j gateway using Spark's run command so that we
pick up the
        # proper classpath and settings from spark-env.sh
        on_windows = platform.system() == "Windows"
        script = "./bin/spark-submit.cmd" if on_windows else "./bin/
spark-submit"
...
```

如果我们能在 IPython 进程中设置环境变量 PYSPARK_GATEWAY_PORT 为真实的 Py4J Gateway Server 监听的端口，就会跳过 Spark-Submit 以及启动

Py4J Gateway Server 部分。那么 PYSPARK_GATEWAY_PORT 从哪来呢？我们发现在 Python 进程中存在这个环境变量，只需要通过 ExecutorPreprocessor 将它传递给 IPython 进程即可。

使用案例

数据分析与可视化

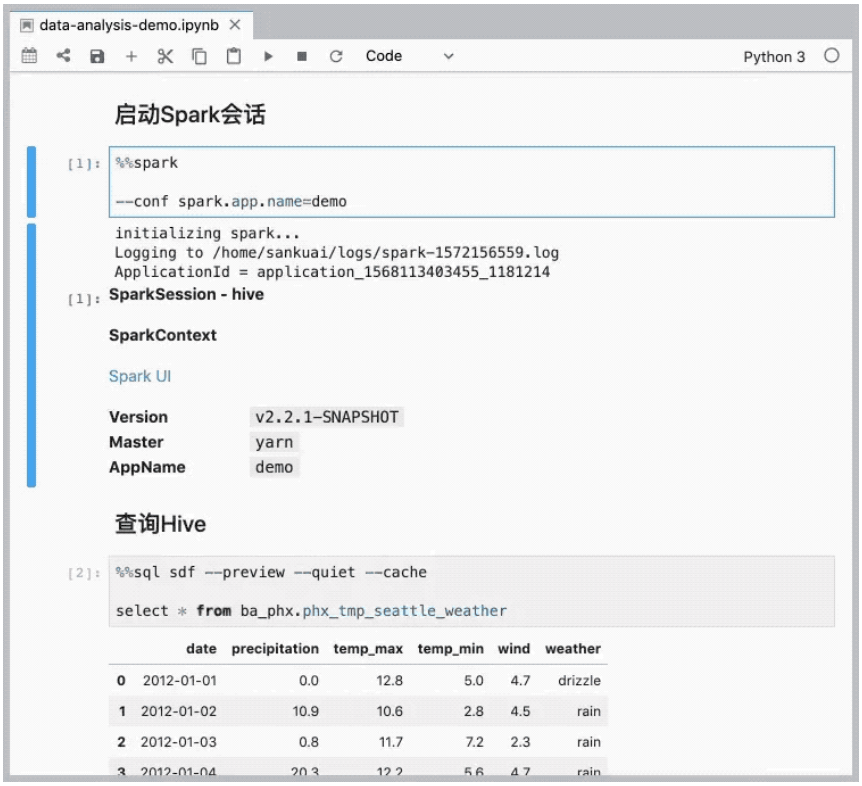
数据探查和数据分析在这里都是同样的流程。用户要分析的数据通常存储在 MySQL 和 Hive 中。为了方便用户在 Notebook 中交互式的执行 SQL，我们开发了 IPython Magics %%sql 用来执行 SQL。

SQL Magics 的用法如下：

```
%%sql <var> [--preview] [--cache] [--quiet]
SELECT field1, field2
  FROM table1
 WHERE field3 == field4
```

SQL 查询的结果暂存在指定的变量名中，对于 MySQL 数据源的类型是 Pandas DataFrame，对于 Hive 数据源的类型是 Spark DataFrame。可用于需要对结果集进行操作的场合，如多维分析、数据可视化。目前，我们支持几乎所有的 Python 数据可视化库。

下图是一个数据分析和可视化的例子：



数据分析与可视化

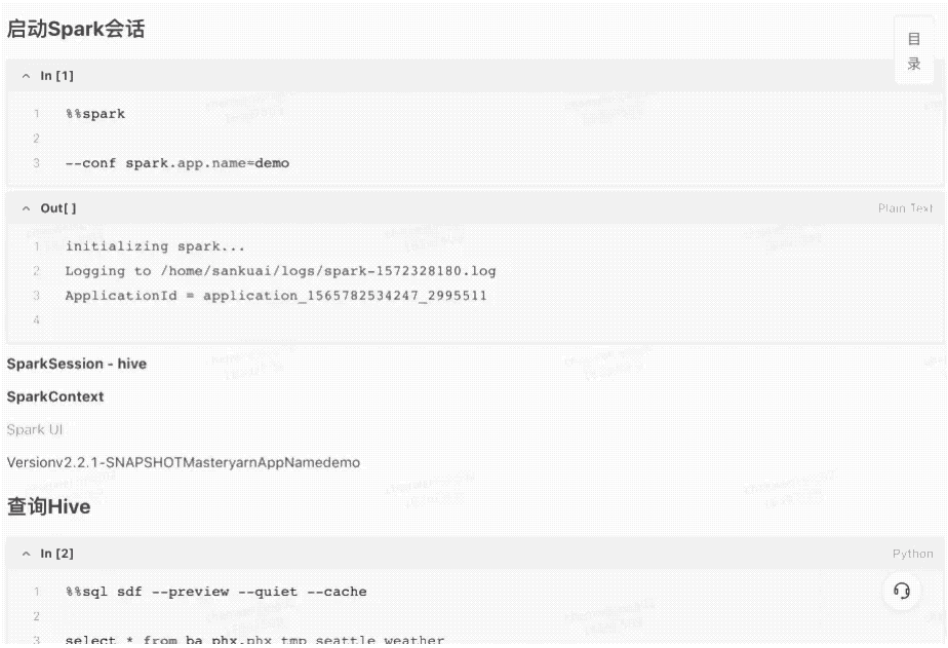
Notebook 分享

Notebook 不仅支持交互式的执行代码，对于文档编辑也有不错的支持。数据分析过程中的数据、表格、图表加上文字描述就是一个很好的报告。Jupyter 服务还支持用户一键将 Notebook 分享到美团内部的学城中。

一键分享：

一键分享

上述数据分析分享到内部学城的效果如下图所示：



Notebook 分享效果

模型训练

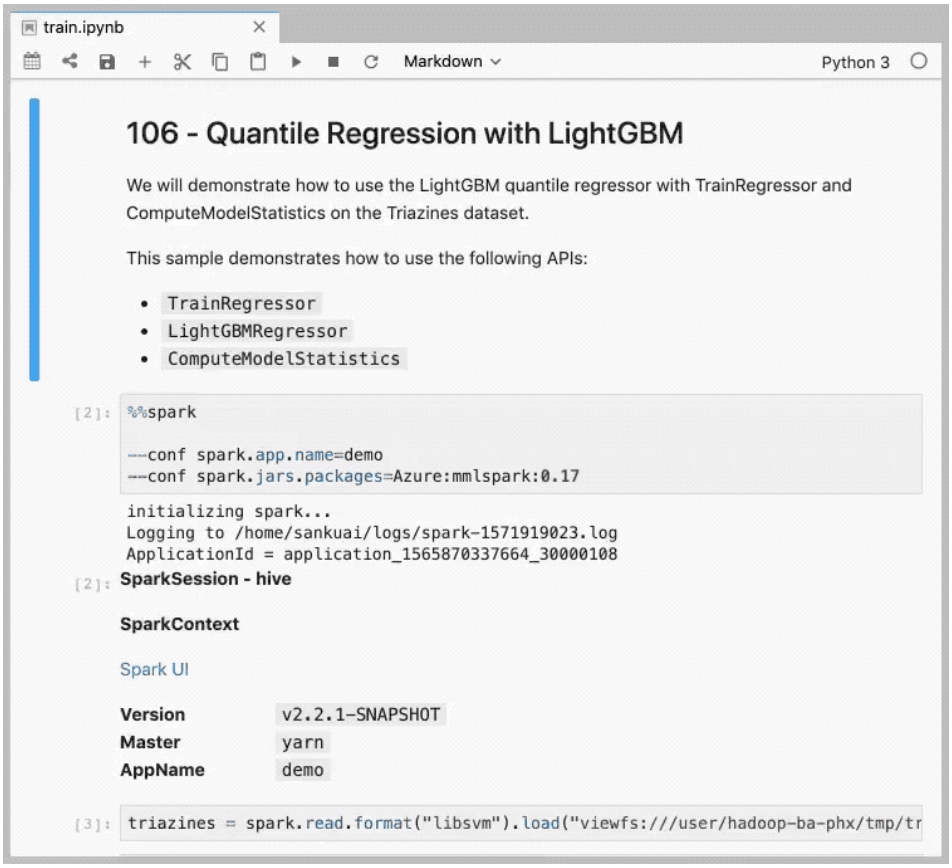
基于大数据的模型训练通常使用 PySpark 来完成。除了 Spark 内置的 Spark ML 可以使用以外，Jupyter 服务上还支持使用第三方 X-on-Spark 的算法，如 XGBoost-on-Spark、LightGBM-on-Spark。我们开发了 IPython Magics %%spark 来简化这个过程。

Spark Magics 的用法如下：

```
%%spark
[--conf <property-name>=<property-value>]
[--conf <property-name>=<property-value>]
...
```

执行 %%spark 后，会启动 Spark 会话，启动后 Notebook 会话中会新建两个变量 spark 和 sc，分别对应当前 Spark 会话的 SparkSession 和 SparkContext。

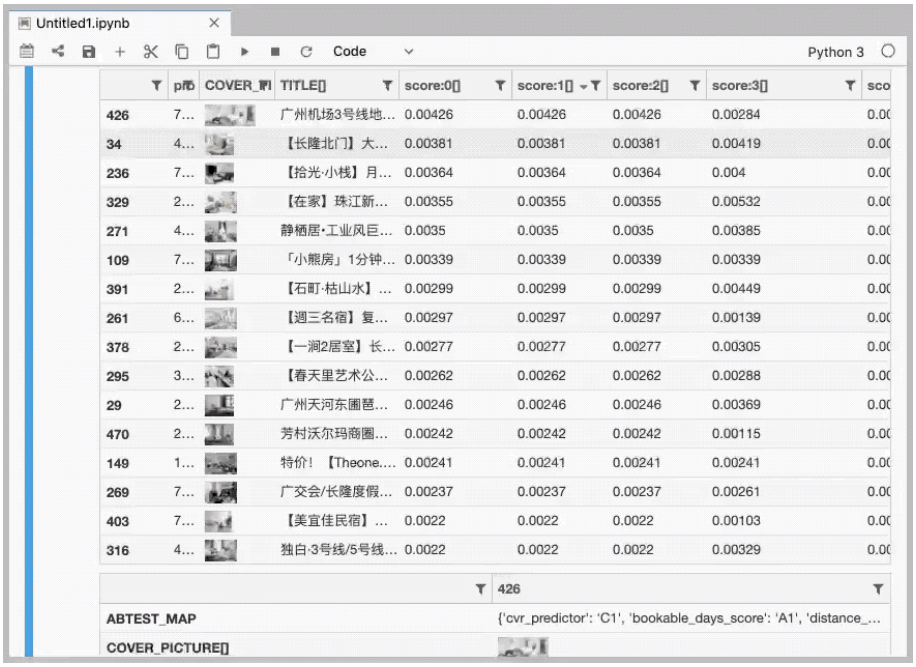
下图是一个使用 LightGBM-on-Yarn 训练模型的例子，基于 Azure/mmlspark 官方 Notebook 例子，仅需添加启动 Spark 语句以及修改数据集路径。



LightGBM on Spark Demo

排序策略调试

通过开发 ipywidgets 实现了一个线上排序策略的调试工具，可以用于查看排序结果以及排序原因（通过查看变量值）。



总结与展望

通过平台化 Jupyter 的定制与部署，我们实现了数据分析、数据生产、模型训练的统一开发环境。在此基础上，还集成了内部公共服务和业务服务，从而实现了从数据分析到策略上线到结果分析的全链路支持。

我们对这个项目未来的定位是数据科学的云端集成开发环境，而 Jupyter 项目所具有的极强扩展性，也能够支持我们朝着这个方向不断进行演进。

作者

文龙，美团民宿研发团队工程师。
颖艺，美团民宿研发团队工程师。

招聘

美团民宿研发团队诚招数据系统研发工程师，Base 厦门，欢迎有兴趣的同学投递简历到 tech@meituan.com 。

人物志

MIT 科技创新“远见者”：美团 NLP 负责人王仲远

王鹏

2019 年 1 月 21 日，《麻省理工科技评论》发布了 2018 年“35 岁以下科技创新 35 人”（35 Innovators Under 35）中国榜单，美团点评 AI 平台部 NLP 中心负责人、点评搜索智能中心负责人王仲远获评为“远见者”。

Innovators Under 35 是《麻省理工科技评论》杂志从 1999 年开始的年度评选，针对新兴科技产业的青年从业者，肯定他们的创新工作。历史上的获奖者包括 Google 创始人 Larry Page 和 Sergey Brin，Linux 创始人 Linus Torvalds，Facebook 创始人马克·扎克伯格（Mark Zuckberg），网景浏览器创始人 Marc

Andreessen，Apple 设计负责人 Jonathan Ive 等杰出人士。

趁此机会，美团技术学院采访了王仲远博士，本文内容根据采访内容整理而成。

大学

王仲远没有想到，他高考时填报的第一志愿是中国人民大学国际经济与贸易专业，最后却被调剂到了中国人民大学的计算机系。更没想到的是，他从此爱上了这个专业，并且在计算机这个行业越走越深。

幸运的是，王仲远就读的人民大学虽然以文科著称，却是中国数据库的学术重镇，萨师煊、王珊、杜小勇和孟小峰等几代人都是数据库领域的知名学者。

刚开始的时候，王仲远也很困惑学校为什么总是教数据结构、操作系统、编译原理、计算机组成这种基础性的知识，好像都不是找工作所需要的技能。孟小峰教授却告诉他，在学校里把基本功打扎实最重要，如果有了扎实的基本功，社会上培训班教的那些“短平快”的技术，其实是很容易学习的。

大二结束的暑假，王仲远找到孟小峰教授主动请缨，开始进入孟老师的 WAMDM（网络与移动数据管理）实验室学习，做了很多数据相关的项目，包括国内首款 Native XML 数据库 OrientX 的系统测试，以及 Deep Web 数据集成项目的研究。他发现孟老师的教导完全正确，有了基础课的底子，对很多工作都有着完全不一样的理解，做实际项目时上手很快。

本科期间，王仲远与人合作的论文《Deep Web 数据集成中的实体识别方法》被中国数据库学术会议 NDBC 2006 录用，并发表在《计算机研究与发展（增刊）》上。这小小的一个进展，却意外开启了王仲远的研究之路。2007 年他获得了国际顶级学术会议 SIGMOD 2007 Undergraduate Scholarship 奖（当年全球只有 7 人获奖）。这更坚定了他踏上学术研究道路的决心。

微软亚洲研究院

硕士毕业时，王仲远很冒险地拒绝了百度、腾讯和 IBM 等众多知名公司研究机构的邀请，选择坚持等待微软亚洲研究院的 offer。

“其实当时我也很迷茫，在宿舍里望着窗外的枝头，希望能看到喜鹊经过。”而王

仲远不知道的是，他在当时微软亚洲研究院所有候选人中面试结果排名第一。他未来的老板王海勋和田江森主动找到研究院院长说，“这个小孩真不错，如果我们再不发 offer，就对不起人家了。”当年 12 月份的寒冬，王仲远终于成为研究院那批候选人中第一个拿到 offer 的人。

王仲远说自己很幸运，微软亚洲研究院具备很多先天优势，在刚刚工作时就可以接触到很多世界级的学者，了解各领域前沿研究已经做到什么程度，无需摸着石头过河。

2010 年 10 月，微软创始人、当时的全球首富比尔 · 盖茨来研究院访问，正式入职两个多月的王仲远获得做现场演示的机会。“可以想象，我当时是多么的激动。”王仲远所在的团队那时已经开始做一些知识图谱领域的探索和研究，虽然业界还没有这个概念。演示很成功，盖茨的反馈非常正面，给王仲远很大的激励，更坚定了他在知识图谱探索的道路上苦心孤诣地前行。这张与比尔 · 盖茨合影的照片，至今他还保留着。

与盖茨的合影，后排右五是王仲远。照片中的很多人如今都已成为 AI 领域的杰出人物，你能认出多少位?

王仲远在微软亚洲研究院 6 年多，从校招生一直做到主管研究员，负责了微软研究院知识图谱项目和对话机器人项目。他一直专注于自然语言处理、知识图谱及其在文本理解方面的研究，取得了不少成绩，在国际顶级学术会议如 VLDB、ICDE、IJCAI、CIKM 等发表论文 30 余篇，并获得 ICDE 2015 最佳论文奖。

谈到做研究的经验，王仲远总结说，很多时候我们觉得某件事情比较高深，只不过是不了解而已。做学术研究跟创业一样，只有自己真正扎进去才会发现其中的奥秘。

同时，做研究需要长期有耐心，这刚好和现在美团所倡导的价值观相符。因为在这个过程中，你会受到非常多的质疑，也会面临多方面的挑战，包括来自你的同事、你的老板、还有学术界中其他流派的挑战。同时，还有短期的压力和长期的压力，也有项目无法落地的压力。但只要对这件事本身真的感兴趣，不管是做技术还是做研究，都可以做的很好。

在王仲远看来，做研究需要有一颗强大的内心，按现在美团的话说，是炼心志。一方面他经常虚心地向前辈们请教，另一方面就是当别人质疑的时候，坚持自己的理想和信念。在他看来，**做研究，过程往往比结果更重要，做正确的事情，好结果自然会来**。

“我从来没有给自己定一个目标，比如说要发几十篇顶级会议的学术论文。而是告诉自己，要踏踏实实地把这个研究项目做好，实实在在解决这个技术所面临的一些挑战性问题，当这些突破和研究的成果不断出现的时候，发表论文就是一个自然而然的事情。”

现在，王仲远也经常跟美团 NLP 中心的同学讲，**做事情要首先关注问题本身，要进行深度的思考，注重解决问题的逻辑和体系**，而不是一上来就简单粗暴地冲着结果去。因为往往是人们越想得到结果，就越得不到结果。反而是专注解决问题本身，好结果就自然获得了。

Facebook

2016 年，王仲远在考虑自己下一步职业规划的时候，更多的是思考如何将自己

的一身所学付诸实践，而不仅仅只停留在研究层面。“如何将技术转化为更为实际的生产力，更加直接地影响几十亿人的生活”，这是他再次出发的初心。

这一次，王仲远依然拿到了很多顶级机构发出的 offer。他最终选择了 Facebook。因为相比于微软，Facebook 是一家纯粹的互联网企业，能够更加直接地面向消费者和用户。而且，在王仲远的眼中，扎克伯格是一个优秀的创始人。“我选择公司非常看重创始人的素质，他是不是一个有理想、有抱负、有信念的人。因为只有这样的创始人才能有战略性的思考，才能不被短期资本市场所影响，才能顶住财报的压力、舆论的压力，才能帮助企业走的更远。”

在 Facebook，王仲远主要负责公司的产品级 NLP Service，要在用户每天发布的几十亿条帖子（Post）中，完成语义分析、查询以及搜索等相关的工作，从非常庞大且复杂的信息流中，找到一个用户想要看到的结果。比如在 Facebook 用户搜索 Trump 时，系统就知道用户想要找美国总统特朗普相关的信息。不过，在特朗普没有当上美国总统时，更多返回的结果其实是纽约的地标建筑特朗普大厦。

王仲远说：“这是非常有意思的一件事，用户的各种查询意图，其实会随着时间而变化，我们做的事情就是在有限的关键词中解读出非常丰富的信号，然后用于各种搜索的召回、排序以及展示。今天，美团大脑以及点评搜索的深度查询理解服务也在做类似的事情，只不过我们处理的信息变成了几十亿条餐饮娱乐的评论以及数百万的活跃商户信息。”

在 Facebook 工作期间，他所负责的一个项目是做实体链接，就是要把查询（Query）和知识图谱进行打通，这也是 NLP 领域一个非常重要的方向。短短半年的时间，效果就提升了 80% 左右，成为 Facebook 内部最重要同时也是世界上最先进的产品级实体链接服务。Facebook 的搜索、推荐、广告、智能助理等许多系统中，都在使用他负责的这些技术。

美团

“我们中国的移动互联网现在真的是非常发达，一天到晚不带钱包、信用卡，生活毫无问题，方便快捷。相比之下，美国就是一个发达的大农村。在国内叫个美团外

卖，半小时就能送到家门口，在美国这简直是无法想象的。”2018 年，因为家庭方面的考虑，王仲远选择回国发展。

他收到了多家知名公司的橄榄枝，百度、腾讯和阿里巴巴都给出非常丰厚的待遇。但是他的考虑是，此前已经在微软亚洲研究院、Facebook 工作过，再去选择一家非常成熟的大公司，并不会发生太大的改变。他想接受新的挑战，承担更重要的角色，更希望选择一家能够发挥出自己更大优势的公司。

滴滴、快手、今日头条等很多互联网新贵，当时也给王仲远发出了邀请，但他最终选择了美团。

为什么？王仲远说，他最看好美团，相信美团是一家能够持续几十年乃至更久生命力的公司。

阿里巴巴从解决“衣食住行”中的“衣”开始起家，电子商务也成就了阿里巴巴。那么在“食、住、行”等这些生活服务领域呢？王仲远相信，一样会成就新的互联网巨头。而且民以食为天，美团外卖已经占据绝对领先的市场份额，美团酒店单日入住间夜不断在刷新行业的新纪录，美团出行也完成了战略布局。

美团最新的战略是“Food + Platform”，王仲远对此非常认可。他相信，十年后，一定会有新的纯线上 App 出现，大家那时很可能早就不玩“抖音”了。还会有更多新鲜好玩的游戏，也不会再玩“王者荣耀”了。这种纯线上的信息和娱乐服务，变化是非常快的，所谓“江山代有才人出”。

“但是无论怎么变，大家总是还要吃饭的。”即使将来有一天，技术真的可以让我们吃饭的方式完全不一样，但是人类也绝对不会放弃对美食的追求，因为这本身也是一种乐趣。这也意味着，美团很有可能成长为一家长期有耐心、不断积累、不断发展的公司，“Food + Platform”也会是一项非常长期的事业。

王仲远的另外一个思考就是，AI 技术想真正能够落地，需要算力，需要数据，需要算法模型，更需要丰富的应用场景。美团的应用场景丰富程度，显然远超滴滴、快手、今日头条等互联网同行。“在这样的情况下，美团对我来说，可能就是不二的选择了。”

还有一个重要因素是，王仲远非常认可美团创始人王兴，“我还是很崇拜兴哥的，

他是非常有理想、有信念、有战略思考的一个人。”

王仲远坦言，最终选择加入美团，也是非常认同美团倡导的“以客户为中心”、“追求卓越”、“长期有耐心”这些价值观，他相信可以在美团中发挥自己的才华，而美团也提供了一个广阔的舞台，可以让他尽情地施展。

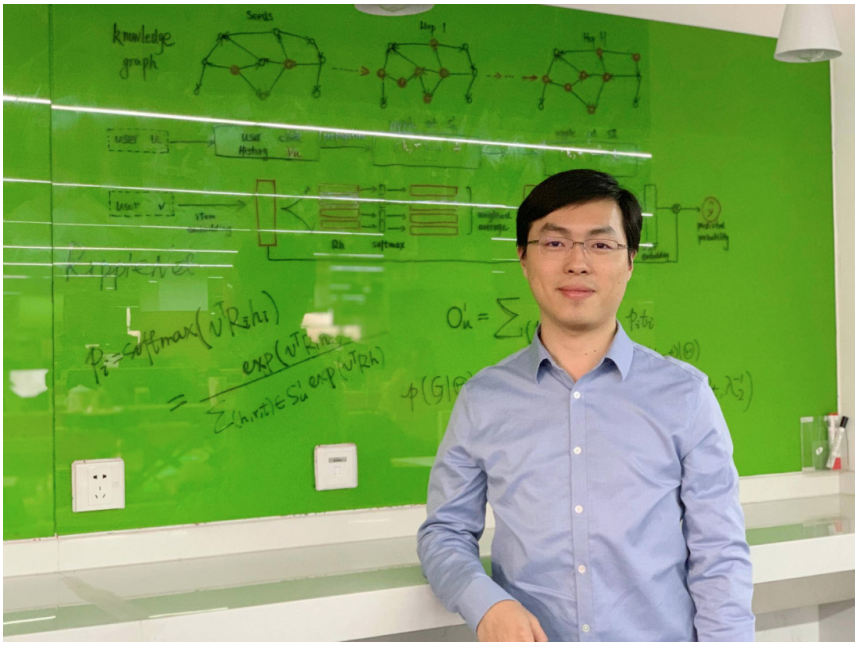
从 0 到 1 组建新团队，最看重成员的价值观

刚到美团，王仲远受命组建美团 AI 平台部的 NLP 中心。他对团队成员的要求是：知行合一，希望大家都是带着自己梦想加入这个团队，踏踏实实地把事情做出来。

王仲远说：“从 0 到 1 组建一个新的团队，挑战还是非常大的。我们的标准非常高，即使招聘速度再慢，都要保证团队成员的质量。”

因为招聘要求很高，所以王仲远需要花很多时间去吸引那些认可美团价值观的候选人。他是怀抱将 AI 技术在各种场景中落地的梦想加入美团，他相信肯定有很多跟自己一样有梦想、有信念的候选人。他也希望找到这样一群志同道合的人，共同前行。

今天，美团 NLP 中心团队已经初步成型，汇集了很多优秀的 AI 专家和工程师。



从“美团大脑”到搜索系统的智能化升级

很快王仲远又开始负责大众点评搜索智能中心。王仲远说：“我非常感谢公司管理层对我的信任，能让我同时负责两个团队，一个 AI 平台团队，一个业务平台团队。这使得我能够更好地规划和掌控 AI 技术的落地，让 AI 技术更好更快地发挥价值，帮助业务平台进行智能化升级。”

2018 年 5 月，他开始领导团队构建美团大规模餐饮娱乐知识图谱的平台——美团大脑。这个“大脑”充分挖掘、关联各个场景数据，使用 AI 算法让机器“阅读”用户针对商户的公开评论，理解用户在菜品、价格、服务、环境等方面的喜好，构建人、店、商品、场景之间的知识关联，从而形成一个“知识大脑”。

在王仲远眼中，美团大脑更像一个 AI 的基础设施，目前这个 AI 平台已经开始逐步服务于美团的搜索、SaaS 收银、金融、外卖、智能客服等众多应用场景。在这些场景中，既有 ToC 的业务，也有 ToB 的业务。

同时管理两个团队，经常往返于北京上海之间，对王仲远而言，虽然辛苦，但效果也很显著。在 NLP 中心以及大众点评搜索智能中心两个团队的紧密合作下，短短半年时间，点评搜索核心 KPI 在高位基础上仍然大幅提升，是过去一年半涨幅的六倍之多，提前半年完成全年目标。

王仲远眼中的微软、Facebook 和美团

一个在学术圈深耕了六、七年的研究型人才，进入企业做项目落地。拥有两家全球顶尖科技企业的从业背景，但回国后却放弃优厚的待遇，选择一家互联网公司再次出发。王仲远的每一次选择，都显得有些与众不同。但是在他身上，我们没有看到任何”莽撞“的成分。每一次选择，他都经过深思熟虑，而且是慎之又慎。

在王仲远的眼中，微软亚洲研究院是中国互联网行业学术研究方向的领头羊，对他的培养和成长，都有很大的帮助。时至今日，微软亚洲研究院对他的影响，仍是不可磨灭的。不过，微软也面临着时代的挑战，虽然这么多年一直在尝试突破，但是它仍然更像一家传统的软件公司。

而 Facebook，是一家非常顶尖的互联网公司，它也是很多国内的互联网公司学习的榜样。Facebook 有一个口号是“快速行动，打破传统（Move Fast and Break Things)”，可以看出，他们对“快”的追求。Facebook 还有很多像“Go Big or Go Home”这一类的内部口号，这跟美团的技术团队“要么牛 X，要么滚蛋”的说法异曲同工。王仲远认为，Facebook 的进化速度要比微软快很多。

而美团是一家比 Facebook 节奏更快的互联网公司，当然这代表了中国速度，也代表了中国互联网的发展速度。王仲远说：“很多的事情，可能在微软亚洲研究院我们需要要用一年的时间来做，在 Facebook 可能会用半年。但是在美团，我们可能只有两个月到三个月的时间。”

美团的高速成长，给王仲远的团队带来很大的挑战。他们每天都要面临各种的持续迭代，要做很多快速的技术演化和突破。美团技术团队是在为生活服务各行业构建信息基础设施，实现需求侧和供给侧的数字化，任重而道远。

未来的路还很长，王仲远和他的 AI 团队，还在路上。

对话王仲远：关于职业发展、知识图谱以及 AI 的未来

秉持信念，不忘初心，不断拥抱变化，才能真正把工作做好

Q：你觉得偏研究型的人才，怎么在企业中发挥出自己的价值?

王仲远：任何科技的发展，都需要研究型人才的推动。但是研究型的人才分几

种，有的是做纯理论研究的人才，可能高校是他们最好的选择。这些人愿意几十年如一日的深耕一个领域。比如深度学习先驱 Geoffrey Hinton 教授，长期从事神经网络领域的研究，不管是神经网络发展的高潮还是低谷，他都能坚持做下去。

还有一些人才，他们更希望把自己学到的各种研究模型、研究成果，能够实实在在地应用在真正的科技产品里面，然后去影响几亿人甚至几十亿人的生活。那么这一类研究型人才，他们更看重应用型的研究，公司可能会更需要他们。

我们整个 NLP 中心也在做一个平衡，80% 的同学会偏应用型研究，也会有 20% 同学会做偏理论的研究，也会鼓励同学们根据兴趣做一些前沿的技术研究。我们希望能够保持较强的科技创新能力，并具备长期的核心竞争力。

Q：外企和海归背景的人，怎么在本土企业中证明自己的价值?

王仲远：不建议给自己贴“标签”。文化上肯定会有一些冲突和差异。但是能不能适应，其实很大程度上取决于个人，取决于这些人能不能拥抱变化。我必须承认，国内互联网的变化或者演化速度，远远超出原来我待过的两家外企，我也必须要适应和拥抱这种变化。

如果我们能够**秉持自己的信念，不忘自己的初心，同时能不断地提炼自己，升级自己，愿意拥抱这些变化**，我相信在新团队也可以做的更好。而且我也观察过，**真正的最顶尖的人、最聪明的那些人，他们不管做什么行业，不管做什么事情，不管身处什么样的环境，都能够把事情做的非常好，都能够取得非常好的成就**。

在我看来，“人”本身的因素要比“外企背景”这种因素重要很多。我深信真正聪明的人的适应力也都是非常强的。而且加入美团后，我们发现很多很厉害的人，即使把他们放在一个完全不懂，或者不熟悉的项目中，他们一样可以做的很优秀。比如美团内部某个业务部门的负责人，曾经做了 15 年的互联网音乐。我相信拥抱变化的人，都不会做的很差。

至于如何证明自己? 其实刚刚我也提到过，不要刻意。越刻意地想去证明自己，往往越证明不了自己。越想刻意拿到结果，往往越拿不到结果。我会给身边的同学提这样的建议。**关注事情本身，关注怎么解决用户的痛点，关注怎么解决技术的难点，关注怎么解决业务的需求**。如果把这些事情做好了，既能拿到结果，最终也能证明自

己的价值。

知识图谱技术的春天来临，是因为大数据在推动

Q：像知识图谱相关的技术已经存在了很多年，为什么迟迟没有进入大众的视野呢?

王仲远：技术的发展，永远都是起起伏伏，处于一个螺旋上升的阶段。

知识图谱并不是新技术。早在上个世纪 80 年代，就有很多知识库系统的研究了。包括 1984 年开始的 Cyc 就是一个知识图谱项目。其实比 Cyc 更早之前，还有很多医疗诊断的专家系统。

但是受限于当时的计算能力、人们的认知、数据量，很多都是通过人工编写规则，或者去找专家建设行业知识库，这种方式不仅效率低下，而且人的思考也具备局限性。通过领域专家去构建的知识库，通常就是几十万的量级，显然不能够满足需求。

2000 年以后，随着互联网的高速发展，数据也越发的丰富，促进了知识图谱技术的蓬勃发展。我们这些做知识库的研究者开始有了新的思路，不再以专家系统为驱动，而是变成依靠数据来驱动产生知识。从观念层面发生的根本性改变后，知识图谱马上展示出自身巨大的价值。特别是 2012 年，谷歌发布了 Google Knowledge Graph，人们又重新认识了知识图谱技术。

此外，知识图谱之所以影响力有限，还有一个很重要的原因就是，拥有这种规模数据的只有少数几家大公司。它们暂时也没有办法去开放，因为涉及到用户隐私问题、数据安全问题，同时还涉及到核心竞争力。但我相信，所有从事 AI 行业相关的人都清楚地知道知识图谱的价值和意义。其实在这些互联网巨头内部，不管是微软、Facebook，包括谷歌和百度，它们对知识图谱技术都非常重视。

Q：你觉得像知识图谱这些技术的落地，目前面临的最大挑战是什么?

王仲远：可能很多人不太清楚，十年前或则更久之前，在学术研究界就已经存在非常多的且成熟的知识图谱相关的技术，包括知识的提取、知识的发现、实体识别、实体链接等等。

但是，学校实验室或者研究所缺少海量的研究数据，科研人员基于的数据量基本处在几十万或者百万的量级，当这些技术遇到亿级甚至百亿量级的数据时，很多技术

走不通。

我们目前面临的最大挑战就是，即使是一个非常简单的，或者已经被学术界认定非常成熟的技术，在去解决百亿、千亿量级知识图谱应用问题时，基本都会失效。因此，**应该如何重新设计算法，应该如何重新设计新的工程架构，是我们需要解决的核心问题。**此外，深度学习跟知识图谱进行结合，也是一个非常重要的研究方向。

Q：你觉得对企业而言，如何才能做好 NLP？

王仲远：NLP 是人工智能所有方向中一个还需要突破的领域。像语音识别和图像识别，通过深度学习已经可以得到非常好的结果。但人类和动物一个非常重要的区别就是，人类拥有非常完备的理解、推理、思考能力，这些能力是 NLP 需要解决的，但是挑战性也很高。

目前在 NLP 领域，我们已经做了很多的突破，也有一些很好的落地场景，比如智能客服，语音助手，还有搜索、推荐、广告这些应用场景。但是在特定的领域，比如餐饮、酒店预订、出行等领域，还有很大的提升空间。

在我看来，企业想要做好 NLP，首先需要打通基础数据。像美团大脑目前包含了 23 类概念、18 亿实体、600 亿三元组，这个知识关联数量级已经达到了世界级的规模。只有数据被打通，才能发挥出更大的价值。其次，需要构建扎实的技术底层，打造一个平台，使得不同的业务线都能更方便、快捷地去使用 NLP 的这些技术，当新技术出现时，也可以快速进行升级和迭代。

总的来说，一个是打通数据，一个构建技术底层。当然还有人才，人才也是实现以上两者的根本因素。当然，既能够安心做研究，又能够落地的核心人才也是最稀缺的，我们美团 NLP 中心对于优秀人才的渴求，也是永无止境的，我们随时都欢迎有理想、有信念的同学加入，一起创造未来。

相信 AI 的未来：道路是曲折的，前景是光明的

Q：怎么看待 AI 未来的发展？

王仲远：我应该属于理性的乐观派。在我看来，不管是深度学习还是知识图谱，**技术发展历程永远会有高潮，也会有低谷**。这两年因为资本的涌入，因为媒体的宣传，有时甚至过度炒作，使得 AI 进入了大众的视野。

这会带来一个好处，就是人们对 AI 有了更加广泛的关注。但是也带来很多的风险，比如很多不是特别成熟的 AI 技术也被“催熟”了，如果这些 AI 产品不能够很好地解决人们实际需求的时候，就会被大家所质疑，比如像前几年很火爆的 VR、AR 等等。还有无人驾驶技术，距离真正落地，还存在很多亟待解决的问题。

我预计未来一两年，随着资本的收缩，AI 可能会陷入冷静期甚至是低谷期。但这个时间应该不会持续太久，因为**核心技术一直都在研究和突破**。随着 5G 时代的到来，随着 AR、VR 技术更加成熟，随着 NLP 技术更加成熟，随着无人驾驶技术更加成熟，未来 AI 也会带来产业新的热潮。

20 年 ToC，20 年 ToB。在互联网高速发展的前 20 年，我们主要解决了消费者的需求。但在很多传统产业，供给侧的数字化，AI 对它们的影响还很有限，这也是我们未来的机会。

Q：对从事 AI 相关工作的同学，有什么建议吗？

王仲远：我觉得大家还是要多关注核心技术，以及核心技术跟业务场景的结合。一个属于“基本功”，另一个是思考技术对业务的价值。

因为技术永远是日新月异，也会不断地更新迭代。特别是做算法研究的同学，相对比较辛苦，要持续的进行学习和提升，如果有一段时间不学习，很可能会被淘汰。

在我们的面试过程中也能够感觉出来。如果面试算法工程师的同学，还是用传统模型而不是深度学习的话，就会面临很大的挑战。此外，如果只关注模型本身，而不

去思考如何跟业务进行结合，在工作中也很难证明自己。

如果是在校的大学生，建议大家不要上来就学习 Java、Python 这些编程语言。应该先对计算机的各种原理、基础知识（概率论、数学分析）等掌握清楚，再去学习很多技术时，就会有完全不一样的理解。我们未来的一切，都应该建立在一个扎实的基本功之上。

影响王仲远的书、人、事

Q：你觉得对你影响最大的一件事是什么？

王仲远：很难说具体的哪件事影响最大。刚刚也提到了，这么多年的职业生涯，我做了很多次、可能看起来比较重大的选择，但是我很难讲，哪一件事会彻底的改变自己的职业生涯。

相对来说，影响比较大的事情有很多，像大学时获得“SIGMOD 2007 Undergraduate Award”奖项，让我坚定地选择了研究这条路。后来在这条道路上，也受到很多老板对我的鼓励。还有刚工作两个月，就得到跟比尔盖茨做汇报的机会，得到他的正向反馈。还有在顶级学术会议发表论文，以及现在做的美团大脑项目，包括刚刚获得的《麻省理工科技评论》这个荣誉等等，我想这些事对都产生了非常大的影响。

但对于具体的一件事而言，我觉得，**更多的是在每一次做人生选择时所做的坚持，以及坚持之后通过努力所带来的一些认可**。这也是鼓励我在各种质疑中、在各种压力中，持续前进的一个非常重要的动力，使我能够长期地坚持下去。

Q：你觉得，这么多年对你影响最大的一个人是谁？

王仲远：对我影响很大的人其实很多。比如我的第一个导师孟小峰教授，他带领我进入了研究的大门。我的第一任老板王海勋博士，他也是我在 Facebook 的老板，给了我很多次机会。还有后来研究院的常务副院长马维英（现今日头条人工智能实验室主任），从他身上，我学到了很多管理的能力。加入美团后，兴哥和老王（美团联合创始人王慧文）对我的影响也非常大，每一次听他们的分享，都让我受益良多，比如兴哥对战略的思考能力和大局观，老王对市场敏锐的把握、对产品的认知等等，都让我学到非常多新的知识。我再次感觉到自己在快速的成长之中。他们对我的帮助也都

非常大。

Q：如果让你推荐一本书，你会推荐哪本？

王仲远：推荐微软 CEO 萨提亚·纳德拉首部作品《刷新：重新发现商业与未来》。我觉得萨提亚能够带领微软这样的巨型科技公司成功转型，再次成为市值最高的公司，非常令人钦佩。

尤其是当年我也曾在微软工作过，感受过微软所面临的那种困境与挣扎，更能体会到萨提亚的管理智慧。在书中，萨提亚反复提到“同理心”，这是我所非常认同的，它也深刻地影响我的思考、沟通和行为方式。

相关阅读

王仲远博士获评《麻省理工科技评论》“35 岁以下科技创新 35 人”中的“远见者”，麻省理工科技评论给出的获奖理由为：

“在知识图谱和自然语言处理领域解决多项挑战性问题，其工作涉及搜索引擎、广告推荐、知识挖掘、关系推理、智能助理等多个领域。

获奖人（王仲远）在微软负责包括微软概念知识图谱、企业知识图谱等多个知识

图谱和 NLP 相关项目，提出的概念化模型能让计算机像人类一样对文本进行理解；在 Facebook 领导团队构建了世界上最大的产品级社交网络知识图谱实体链接服务。在美团仅用半年就领导团队构建出世界上最大的餐饮娱乐知识图谱“美团大脑”。

获奖人的开发应用于餐饮、出行、休闲娱乐、旅游、金融等各个场景，为数亿人提供了更便捷、更智能的服务。”

受访者简介

王仲远博士，美团点评高级研究员、高级总监，美团 AI 平台部 NLP 中心负责人、大众点评搜索智能中心负责人。加入美团点评前，担任美国 Facebook 公司 Research Scientist，负责 Facebook 产品级 NLP Service。在 Facebook 之前，担任微软亚洲研究院的主管研究员，负责微软研究院知识图谱项目和对话机器人项目。多年来专注于自然语言处理、知识图谱及其在文本理解方面的研究，在国际顶级学术会议如 VLDB、ICDE、IJCAI、CIKM 等发表论文 30 余篇，获得 ICDE 2015 最佳论文奖，并是 ACL 2016 Tutorial“Understanding Short Texts”的主讲人，出版学术专著 3 部，获得美国专利 5 项。在 NLP 和 KG 研究领域及实际产品系统中均有丰富经验，研究领域包括自然语言处理、知识图谱、深度学习、数据挖掘等。

参考文献

- 《麻省理工科技评论》年度中国科技青年英雄榜发布！ 35 位中国入选者涵盖全球最前沿科学与技术
- 计算机系本科生王仲远荣获 SIGMOD07 Undergraduate Scholarship

美团技术委员会前端通道主席洪磊：爱折腾的斜杠青年

技术学院

洪磊，2013 年加入美团，目前是美团外卖事业部终端组的负责人，也是美团技术委员会前端通道主席。在加入美团之前，洪磊的职业生涯可以用“跌宕起伏”来形容。他就读于中南财经政法大学，曾任职于雅虎中国，先后担任产品经理、前端开发工程师等职位。拥有 4 年创业经历，对前端和硬件技术有着很高的热情。

高中时，洪磊就参加各种计算机竞赛，很喜欢技术，但是大学并没有选择计算机这个方向。2002 年，洪磊考入中南财经政法大学，读国际贸易专业。不过大四那年，他瞒着家里人跟几个小伙伴一起休学创业了。

当时，他们做了一款基于位置的本地服务（跟美团颇有渊源），受限于团队经验和当时的市场环境，最后以失败而告终。随后加入了雅虎中国，起初做音乐搜索产品

经理，一心想做一个好用的音乐搜索产品，但却被技术同学各种“刁难”，“这个不好做，要开发 1 个月”。于是，他从产品转到了后台，然后又从后台转到了前端。

自从 2005 年，雅虎中国被阿里巴巴收购，就在不断拥抱变化，洪磊并不是很喜欢那种工作氛围。2009 年，创业之火再次燃起，于是受邀加入魔位娱乐，在此期间，洪磊做过网页游戏开发，做过塞班 App，从业务、运维、后台再到前端，几乎涉足整个流程并都有较深的参与，练就了“十八般武艺”。2013 年，他觉得自己在团队管理和个人发展上遇到了瓶颈，于是正式加入美团，负责移动前端组，再次踏上新的征程。从美团触屏版开始做起，后面还搞了公司的运营系统，最后，“阴差阳错”地来到了美团外卖技术团队，一直到现在。

近日，我们美团技术学院采访了洪磊，一个“爱折腾”的创业者，一个做过产品、运营、后台和前端的工程师，他说家里的所有的智能家居，都是自己亲手“焊板子”捯饬出来的。我们在洪磊身上，真正看到了一种极客精神。今天，让我们一起听洪磊讲讲，这么多年他“折腾”出来的那些故事。

Q：当时怎么接触到的计算机？为什么大学选择了中南财经政法大学，而且选择了国际贸易专业？有什么特别的故事可以分享吗？

洪磊：其实，我在高中时候就经常参加计算机的竞赛。当时的高中在浙江淳安县，那是一个小县城，学校对计算机的理解或者说关注度其实是很低的。我参加一些省里和市里计算机竞赛，还给学校拿过一些奖项，这也是淳安中学最早的几个计算机奖项。

那个时候，对计算机还是非常感兴趣的，而且通过这几次获奖的经历，在自己心里也打下了这个“烙印”。但因为我们江浙一带的人，更偏向于从商，所以当时选择了中南财经政法大学的国际贸易专业。不过整个大学期间，我也非常有收获，对经济学、会计学、保险、物流、运输等等很多方面，都有了一定了解。国际贸易是一个非常广的学科，它涵盖的东西很多，甚至还要学习谈判技巧、各地文化和礼仪。虽然我觉得这个专业非常赞，但是骨子里，我更偏爱技术一点。

最终我还是选择了休学创业，但我知道，家里肯定会反对。所有我没有敢跟家里讲，我只是跟辅导员聊了一下，她当时很支持我的选择，虽然我对经济类专业也

比较感兴趣。但是她觉得在计算机领域，我可能会有更好的发展。辅导员了解到我顾虑，她说，“没关系，如果你不敢讲，我帮你去讲”。现在，我还是很感谢辅导员给我的支持。

Q：辅导员为什么对你那么有自信?

洪磊：因为我在大学期间，参加了学校里一些计算机相关的社团，还给班里同学辅导技术机等级考试的相关知识，有时还会接一些网站外包的活。最主要的是，我和几个小伙伴给微软旗下的即时通讯工具 MSN Messenger 开发了一个插件，可以加密聊天记录，并且添加了很多好用的小功能，我当时主要负责 Tab 系统的开发和维护。这应该也是 MSN Messenger 历史上最著名的一个插件了。

那是一个虚拟的小组，我当时在武汉，另两个开发者一个在上海，一个在广州，我们是三地办公。有个同学是专门去破解 MSN Messenger 的相关接口，获取相关权限。另一个同学是做通讯内容的加密，我负责 Tab 功能相关的系统，三个人配合的井井有条。

后来发现 Yahoo! Messenger 也有自己的标签系统，我顺便也破解了，这也是后面我能够加入雅虎中国的重要原因。所以我辅导员觉得我在计算机方面有一定的天赋，而且也做出了一些成绩，因此她坚定的支持我去做创业。

Q：现在回看第一次创业，失败的原因是什么？对自己的影响是什么？有没有后悔过创业这件事?

洪磊：首先是经验不足，我们想的太过理想，觉得把产品做出来就能有用户。但在当时，时机并不成熟，没有智能手机，上网主要靠 PC，几乎无法实现精准的定位。做送餐这样的服务，对商家、用户的体验都很差。现在看来，在技术上，我们做的还行，但是在业务上，我们的思考就太少了。

这次创业后，我觉得事，都要有条理地去做，当时团队应该算是“散兵游勇”，没做市场调查，没有 MRD，连 PRD 也是很简单的一个框架图而已，我们就开始开发了，甚至一些技术的实现也因为总是变动功能而做得比较糙。

对休学这件事，还是有一些遗憾的，但是没有后悔过。比如因为没有毕业证，导致我现在办不了北京的工作居住证，部分城市的落户也会很困难。但是不后悔，如果

没有当时的爱折腾，可能也不会有现在这样一个“特别”的我。也许就是每天简单的上班、吃饭、下班、陪家人，当然平淡也是一种幸福。但是我知道，这不是我向往的生活。每次参加校园招聘，我会劝小同学们要来北京这样的大城市，因为视野不同，眼界也大有不同。

我很清楚的记得，有个美团校招时被我的思想“安利”的小同学，她入职一年后告诉我，“真是完全不一样的感觉，如果当时留在成都，大概率就是一生默默无闻的做一份工作，可能永远都不会触达到世界各地。但是，现在我一有空，就全球各地跑、去玩，去了解更多更新的知识，原来世界上有这么多有意思的东西，值得我们去体会。”

Q：为什么会选择雅虎?

洪磊：雅虎当时还是全球第一大网站。而且我刚刚也讲到了，在大学期间，我把 Yahoo! Messenger 的 Tab 系统破解了，还总是给他们提一些问题和想法，所以就认识了当时雅虎中国的产品总监 Sam。大二的时候，他就邀请我去雅虎实习。后来创业失败，也没有机会参加校招，干脆就过去了。

雅虎中国也没有看我的学历。我当时还有点沾沾自喜。因为很多同学参加校招，各种面试，最后等发 Offer。而我，只写了一个 PPT，下面坐了好几个部门的负责人，等我去选择。其实，我当时也不知道自己想做什么，Sam 问我这个问题，我记得是这么回答的：“我想做一些把雅虎美国的产品引入中国以后，让中国用户用的更爽的事，我想做一些本地化，或者说更炫酷的功能。”

Q：给现在的实习生或者在校生提点建议吧?

洪磊：这个问题我需要好好回答一下，我参加过很多次校招，也结合我亲身的一些经历，我给现在还在实习的同学们三点小建议：

- 第一点，要想清楚自己喜欢的工作，或者自己喜欢的方向是什么，这点很重要。很多同学都觉得找个朝阳行业，一份收入高的工作就好。工作以后，我们会逐渐发现，只有我们真正喜欢的，才可能去深入。
- 第二点，建议大家第一份工作要去一个大点的公司，如果是互联网公司，建议去 BAT 或者美团这样的大型互联网企业。因为大公司能够提供一个体系化的

成长平台，让同学们少走弯路。

- 第三点，不要眼高手低，应该从最基础的岗位做起，从底层去了解一个技术。我经常跟美团的小伙伴讲，“要多去看那些基础知识，少去看那些高大上的东西。”如果真想自己去写一个东西，那我希望是从底层原理去了解，而不是通过一些现成的类库来实现，甚至到网上找一个类似的直接改改。

Q：刚开始做的是产品相关的工作，什么原因让你转做前端了呢?

洪磊：这段经历其实挺有意思的，最早我去雅虎中国是做音乐搜索产品助理的，但是对整个产品体系了解不够，就被我老板发配去做运营。其中有个重要的工作就是每天要用几个小时去统计业务的各种数据，非常枯燥。

不过那个过程，对我的帮助非常之大，通过持续了两个月的数据统计工作，我对整个音乐产品有个非常清晰的了解，知道业务上哪些地方有改进的空间。所以后面做产品设计的时候，就可以游刃有余地通过数据来推动业务的发展。

说到转型，还有段精彩的故事。几乎所有的工程师都比较抵触产品，每次去排期的时候，工程师都说这个事情比较难，我们做不了，或者这个事情我们排不上期，如果逼急了，他们可能就来句“你行，你上啊”。当然我也不示弱，就接了一句：“好吧，我自己来搞，你把权限开放给我。”因为我大学的时候就写过 PHP 代码，雅虎用的也是 PHP，所以做一些简单开发，不会有太大的压力。

写完后台以后，发现前端又是个瓶颈，前端人员少，排不上期，那行吧，我又开始写起了前端。这时突然发现前端是我喜欢的一个方向，我就直接转入 UED（雅虎的前端在 UED 团队）。这里有个让我记忆犹新的小故事，可以分享给大家，我作为前端工程师写的第一个页面，因为没有做到到像素级的呈现，结果被投诉到我老板那，被训了一顿。

这件事对我触动很大，做每一件事，都应该符合需求方的诉求，尊重上游同学的劳动成果，而不是自己埋头做一些自己觉得很轻松的事。这也促使我跟设计师同学们的配合都非常默契，包括现在小米（米 UI）的设计部门负责人跟我关系就很好，之前他的个人网站就是他设计我开发的，其中各种渐变、圆角加阴影（当时要考虑低版本 IE），他非常赞赏我完美的呈现，我经常说他“就剩下对像素的这点追求了”。

Q：你会鼓励技术同学转型吗？

洪磊：我还是会鼓励同学们去做一些有挑战的事。而选择转型，去干一个全新的事情，主要还是看个人的兴趣和追求。但是非常推荐同学们去合作团队轮岗。美团外卖就一直在推动轮岗的事，我最近也在产品线轮岗。

我觉得各种技术在思想上其实互通的，我们可以通过轮岗学到交叉领域的很多知识。2013 年在平台做 LocalStorage 缓存优化，就借鉴了 MemCache 的一些想法去实现的。所以很多情况，我们可以去借鉴其他技术的思想，只是代码的实现层面，略有差异。

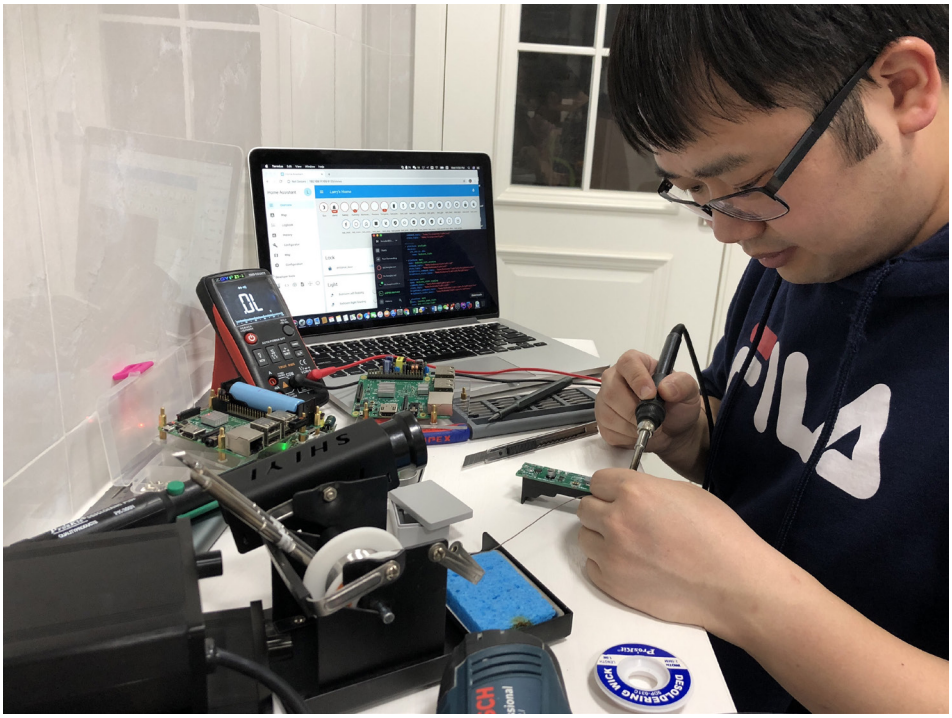
Q：当时为什么会离开雅虎？再次选择创业，有哪些新的收获？

洪磊：很重要的一个原因，就是雅虎中国乃至全球的业务在慢慢萎缩，心里有一定的落差。而且收购后，阿里巴巴的企业文化、价值观我也不是非常适应。刚巧有一个好哥们邀请我和他一起创业，我就决定和他一起“干一票大的”。当然，创业依然是一个很艰辛的过程。

创业要有极强大的内心。我当时从雅虎中国出来，家人朋友并不支持我，因为阿里巴巴在浙江非常有名，大家都挤破了头往里走。父母会觉得，从那么优秀的一家公司出来跑去创业，还是一家“生死未卜”的公司，是脑袋有“坑”的节奏吗？

这次创业中间有很多波折的，做过很多业务，比如网页游戏，比如塞班客户端应用等等。我觉得创业要有耐心，要能够坚持在一个业务上不断深入。也需要灵活，在发现某个业务方向完全行不通的时候快速变化。这次创业，我最大的收获是，从业务到技术，甚至采购和市场都要自己去管理，这让我能更全面的去思考一个业务的发展过程。

其实我还是一个技术流，更喜欢去研究技术问题，家里的智能家居系统都是我自己搞的，其中还包括大部分的硬件，自己购买芯片和元件，然后做 PCB 板、焊接。我希望对这些设备有“掌控权”，或者说我希望这些设备对我来说，是一个完全“透明”的东西。不希望设备“不听话”，更不希望数据莫名奇妙到了别人的服务器上，对于一个做技术同学来说，这是很难容忍的一件事情。



Q：这么多年的创业，对技术的发展有哪些体会？

洪磊：对技术而言，第一点感触是，在商业社会中，能服务好业务的技术才有价值，不推崇去搞一些不容易落地的技术。

第二点感触是，技术和产品应该和业务紧密相连，不能单独割裂开来，我们每个技术同学也都应该了解产品还有业务的思路，做出最合适的技术实现。

Q：后来为什么选择美团？

洪磊：创业的前两年比较辛苦，后面基本上全球跑，每天都可以有“说走就走的旅行”。公司的业务趋于稳定，没有太大的发展，感觉自己的发展也遇到了一些瓶颈。

还有一点，我们的公司属于“哥们型”的公司，在团队管理层面，存在很大的问题，我当时管了 20 人左右，很难再扩大规模了。恰巧雅虎的一个同事推荐我来美团，当时跟亮哥（陈亮，美团高级副总裁）也比较谈得来，所以就加入了。

Q：到美团后经历了哪些事？

洪磊：刚来美团的时候，我们团队只有 3 个人。移动前端在当时也是一个刚兴起

不久的方向，几乎相当于从零开始做，我很享受这个过程。

团队主要工作就是开发美团系相关触屏版页面，期间还管理了相关的后端团队和产品团队，最多的时候有 30 多人，也创造了不错的业绩。同时也为公司很多部门输送了大批人才，包括猫眼（已经独立）、大象、美团酒旅最早期的前端的同学，都是从我们移动前端组过去的。

2016 年初，公司希望我去管理外卖前端团队，最初还是有点犹豫的，后来和外卖同学做了简单沟通后就欣然接受了挑战，当时并没有想到会在外卖团队获得如此快速的成长。

到外卖后，第一个挑战就是 App 的质量欠佳，我并不是客户端出身，所以对客户端技术了解甚少，只能通过管理的手段来做事情，还好当时有非常称心的“左膀右臂”，加上自己精通前端技术，很快跟小伙伴打成一片。我觉得美团的工程师文化还是很好的，“只要你技术好，我就是服你的，可以跟你干”。

现在我们团队有 100 多名小伙伴，管理团队在成员的引入、留存和培养上下足了功夫，资深工程师占比超过 30%，是美团比较优秀的团队之一，我也为此骄傲。在此，也欢迎更多的同学加入我们。

Q：你觉得美团的技术氛围如何？

洪磊：我觉得我们美团的技术氛围挺好的，至少在我待过的几家公司里是最好的。因为我们美团的工程师比较有冲劲，自驱力很强。

第二点，再就是大家都愿意去做一些事情，提高整个团队的影响力。在美团这个大家庭里，永远不是一个人在向前跑，而是共同追求进步。

Q：平时有哪些爱好？

洪磊：一是爱折腾，折腾技术，折腾硬件这些。家里的智能家居系统，都是我自己一个人焊板子，捯饬出来的，我希望家里的东西都是“透明”的，代码也要“透明”。

然后，有时间就去旅行，满世界跑。有时一个人，有时约上三两个好友一起自驾。我不太喜欢“走马观花”，更喜欢在一个地方待上一段时间，去体会那里的风土人情，感受当地的文化。旅行时，我不喜欢有太多的规划。很多时候，不知道自己的

下一个目的地是哪里，只要一直在路上就好。

Q：写代码多少年了？

洪磊：要说第一行代码，应该要追溯到 2000 年左右高二时候了，那时学校还用的是 DOS 系统，我会用 Pascal 语言“暴力”获取一些简单游戏的通关步骤。当时“文曲星”上面有一个叫“汉诺塔”的游戏，我就用穷举法找到了最少的移动步骤。

第一行 PHP 代码是 2003 年写的，直到现在还会偶尔写写。我对写代码这件事兴趣盎然，应该会永远充满激情。

Q：对技术同学有哪些建议？

洪磊：第一点，我觉得应该深入了解底层的原理，而不要只停留在应用层面。这样才能够真真切切地明白程序运行的机理。

第二点，要懂得举一反三，把自己学到的知识，通过加工，创造出更好的实现，使知识得到升华。我感觉中国的工程师比较守规矩，发散性思维较弱，这方面还可以提升很多。

第三点，就是要长期有耐心，这也是我们美团倡导的价值观和做事方式。

论文精选

美团技术团队不仅在新技术落地上追求卓越，还积极探索前沿技术。2019 年，美团技术团队联合国内外多所高校和学者，在数个国际会议上发表重要论文，包括人工智能领域顶会 KDD、CVPR、IJCAI、WWW，GIS 顶会 SIGSPATIAL，机器人顶会 IROS，运筹学年会 INFORMS。

美团点评还在文字识别国际顶会 ICDAR 2019 发布了真实场景的中文门脸招牌图像数据集，并举办了中文门脸招牌文字识别比赛。

依托美团点评生活服务领域多样化真实场景及丰富数据，我们愿与学术界携手，协同创新，探索前沿技术方向，推动理论研究在产业实践中的落地。

The 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems

Effective Recycling Planning for Dockless Sharing Bikes

Cong Zhang
Beijing Uni. of Posts and Tele.
cong1126@bupt.edu.cn

Yanhua Li
Worcester Polytechnic Institute, USA
yli15@wpi.edu

Jie Bao
JD Finance
baojie@jd.com

Sijie Ruan
Xidian University
ruansijie@jd.com

Tianfu He
Harbin Institute of Technology
Tianfu.D.He@outlook.com

Hui Lu, Zhihong Tian
Guangzhou University
{luhui,tianzhihong}@gzhu.edu.cn

Cong Liu
The University of Texas at Dallas
cong@utdallas.edu

Chao Tian
Tencent
astortian@tencent.com

Jianfeng Lin, Xianen Li
Mobike
{linjianfeng,lixianen}@mobike.com

ABSTRACT

Bike-sharing systems become more and more popular in the urban transportation system, because of their convenience in recent years. However, due to the high daily usage and lack of effective maintenance, the number of bikes in good condition decreases significantly, and vast piles of broken bikes appear in many big cities. As a result, it is more difficult for regular users to get a working bike, which causes problems both economically and environmentally. Therefore, building an effective broken bike prediction and recycling model becomes a crucial task to promote cycling behavior. In this paper, we propose a predictive model to detect the broken bikes and recommend an optimal recycling program based on the large scale real-world sharing bike data. We incorporate the realistic constraints to formulate our problem and introduce a flexible objective function to tune the trade-off between the broken probability and recycled numbers of the bikes. Finally, we provide extensive experimental results and case studies to demonstrate the effectiveness of our approach.

CCS CONCEPTS

• **Applied computing** → *Transportation*; *Forecasting*; *Transportation*; • **Information systems** → *Spatial-temporal systems.*

KEYWORDS

bike-sharing systems, predictive model, optimal recycling program

ACM Reference Format:
Cong Zhang, Yanhua Li, Jie Bao, Sijie Ruan, Tianfu He, Hui Lu, Zhihong Tian, Cong Liu, Chao Tian, and Jianfeng Lin, Xianen Li. 2019. Effective Recycling Planning for Dockless Sharing Bikes. In *SIGSPATIAL '19: 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, November 5–8, 2019, Chicago, Illinois, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3347146.3359340

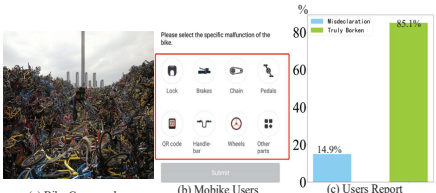
SIGSPATIAL '19, November 5–8, 2019, Chicago, Illinois, USA

ACM ISBN 978-1-4503-6909-1/19/11…$15.00
https://doi.org/10.1145/3347146.3359340

1 INTRODUCTION

Bike-sharing system is a popular transportation system in modern cities, as it not only provides an environment friendly choice for short-distance travelling, but also eases the traffic congestion. Currently, there are over 1,000 deployed bike-sharing systems world wide, and more than 300 systems are in the progress of deployment [29]. In recent years, station-less bike-sharing services, like Mobike[1], which allow users to pick up and drop off bikes at any locations they want, become more popular.

Due to the sharing nature of the bike-sharing systems, the sharing bikes have much higher broken possibilities compared with private bikes due to the high ridden frequency and open-air parking problem. For example, the bike sharing system in New York saw 3.6 daily rides per bike [2]. As a result, as shown in Figure 1(a), thousands of broken station-less sharing bikes are being kept in a bike graveyard.



(a) Bike Graveyards (b) Mobike Users Malfunction Report (c) Users Report Fault Bike Review

Figure 1: Issues with broken Sharing Bike.

Since the number of bikes put in the market is limited, without the proper maintenance, the number of bikes in good condition is continuously decreasing. The broken bikes not only cause economic losses to the companies but also lead to environmental pollution. Therefore, an effective bike recycling plan should be conducted. Currently, Mobike develops a broken bike report function in the app, so that the broken bikes can be discovered in a crowdsourcing way. As shown in Figure 1(b), users can report different types of bike problems in the mobile app, so that the company can arrange workers to collect and recycle them.

[1]https://en.wikipedia.org/wiki/Mobike
[2]https://bit.ly/2T6q5SE

However, there are *three* challenges to conduct such a broken bike recycling task:

Inaccurate and Inadequate Labels. Though the broken bike report function can help the company to quickly locate the broken bike, the report cannot be fully trusted. As shown in Figure 1(c), we manually exam the status of the reported broken bikes. Only 85.1% bikes are truly broken. Furthermore, not all of the users are willing to report the broken status of the bikes, as the broken report function is not a required step.

Arbitrary Spatial Distribution. Different from the station-based systems, the parking location of each individual station-less bike is totally arbitrary, which makes the recycling routes vary from day to day.

Limited Recycle Capacity. Given a set of bikes to be recycled, the worker can only collect the limited number of broken bikes within the working hour. Besides, the capacity of the collecting vehicle is limited, and the worker has to drive back to the recycling site as soon as the vehicle is full of broken bikes.

In this paper, we design a broken bike recycling route planning system for the worker. This system consists of two main modules: 1) broken bike inference, which infers the broken probability of each sharing bike using its inherent characteristics and the user trajectories associated with it; and 2) recycling route planning, which plans multiple closed recycling routes for the worker to conduct in each day.

The contributions of the paper are summarized as follows:

(1) We propose a novel broken sharing bike recycling problem, which takes the broken probability, working time constraint, and vehicle capacity into consideration.

(2) We build a broken bike inference model using inherent features and trajectory features extracted from the sharing bike so that the status of every single bike can be accurately inferred.

(3) We propose a scatter search-based heuristic algorithm for the broken sharing bike recycling problem.

(4) Experiments show the recycling efficiency of the broken bikes recommended by scatter search algorithm is 2.5 times that of the regional random search method and 1.5 times that of the Nearest neighbor routing search method. At the same time, the result of the algorithm is twice the efficiency of Mobike employees' broken bikes recycling.

The rest of the paper is organized as follows: Section 2 describes the problem and the system overview. Broken sharing bike inference model is discussed in Section 3. Section 4 gives the solution of broken sharing bike recycling routing problem. Experiments and case studies are given in Section 5. Related works are summarized in Section 6. Section 7 concludes the paper.

2 OVERVIEW

In this section, we define the broken prediction and recycling routing problem for Sharing Bike, and outline our solution framework.

2.1 Preliminaries

We define p_i as the inferred broken probability of sharing bike b_i. In the recycling task, we only consider bikes which are inferred as broken, i.e., $p_i > 0.5$. The bike with high broken probability is preferred to collect in the priority given limited working time. However, the bikes with high broken probability can distribute unevenly in the given region, which introduces large traveling time, and finally leads to less number of broken bike collected. As a result, we define a beneficial score $score_i$ below to characterize the worthiness of collecting a particular bike b_i. In the broken bike recycling mission, the dockless sharing bike can be at any location in the city, e.g., hiding in the residential area or close to the road network, where the parking location of collecting vehicles is usually along with the road network. As a result, the distance between them varies significantly, which we define v_w (walking speed) and rt (registration time) below to better characterize the individual bike collecting events.

Definition 1. ***(Beneficial score)*** *$score_i$ captures the overall benefit to recycle bike b_i, which characterizes the trade-off between the broken likelihood and the recycling cost of b_i.*

$$score_i = \alpha^{\frac{p_i}{\min p}} \qquad \alpha \geq 1 \tag{1}$$

where the parameter α represents the trade-off preference on the broken probability p_i vs recycling cost. $\min p$ is the minimum broken probability over all the bike in the region, which serves as a normalization term.

Each bike b_i has a broken probability p_i, i.e., the likelihood of being a broken bike. In practice, the trade off when choosing a bike is: If we seek for only bikes of high broken probability p_i, we may end up with a small number of bikes collected (less efficient); on the other hand, if we seek for a large number of collected bikes, many bikes collected may not be broken (false positive). The beneficial score defined in definition 1 captures such a trade-off by the parameter α. The reason for designing a score function using the exponential function is that the bike with higher broken probability will have a higher score ($\alpha > 1$). When α is close to 1, the efficiency is highly considered, leading to a large number of collected bikes; on the other hand, when $\alpha \gg 1$ is large, the broken probability p_i is highly considered, thus only bikes with high p_i will be collected. Especially, $\alpha = 1$ means that we do not care about the broken probability of the bike, and every broken bike has the same beneficial score. The α is a tunable parameter (chosen by the service operators), which provides them the flexibility between the efficiency (i.e., the number of collected bikes) and the likelihood of the collected bike being broken. From the operator's perspective, there are different objectives under various circumstances, for example, in regions hard to access, the efficiency should be highly considered (i.e., choosing α close to 1), while in areas with bikes densely populated, e.g., downtown, accurately collecting each broken bike is preferred, thus the likelihood of broken bikes needs to be considered more (i.e., choosing a large α). As a result, the beneficial score measures the practical "benefit" of collecting each bike.

Definition 2. ***(Sub-route)*** *Each closed route, which starts and ends at the collection site s, is considered as a sub-route.*

Definition 3. ***(Time Cost)*** *The time cost of sub-route R_j is composed of the vehicle travelling time between consecutive locations and the visiting time at each broken bike. Given a sub-route $R_j = s \rightarrow b_{r_1} \rightarrow \cdots \rightarrow b_{r_n} \rightarrow s$, the time cost T_j is calculated as follows:*

$$T_j = T_{travel}(R_j) + \sum_{i=1}^{n} T_{visit}(b_{r_i}). \tag{2}$$

Let us denote the shortest road network distance between broken bike b_i and b_j as $dist(b_i, b_j)$, and the vehicle driving speed as v_d, then the travelling time cost is calculated as follows:

$$T_{travel}(R_j) = \frac{dist(s, b_{r_1}) + \sum_{i=1}^{n-1} dist(b_{r_i}, b_{r_{i+1}}) + dist(b_{r_n}, s)}{v_d}. \tag{3}$$

The broken bike visiting time includes the walking time between the vehicle on the main road and the location of the broken bike, and the broken bike registration time rt. We denote the walking speed as v_w, and the perpendicular distance of the broken bike b_i to the nearest road segment as $shift_i$, then the visiting time cost can be represented as

$$T_{visit}(b_{r_i}) = \frac{2shift_{r_i}}{v_w} + rt. \tag{4}$$

Problem Definition. Given the road network RN, driving speed v_d, walking speed v_w, collection site s, broken bike registration time rt, working hour T, vehicle capacity M, and a broken sharing bike distribution graph $G = (V, E)$. The vertex set $V = \{b_1, b_2, \cdots, b_n\}$ represents all the broken bikes in the given service region of s, each of which is associated with a spatial location and a collection score $score_i$, and the edge set E denote the road network connectivity of broken bike pairs.

The objective of the broken bike recycling route planning problem aims to plan multiple traveling routes for the worker, so that the total score collected is maximized. The recycling route planning problem fulfills three constraints: (1) each broken bike is collected at most once; (2) the working time of the personnel is no more than T; and (3) the broken bikes collected in each sub-route are no more than the vehicle capacity M. If we use δ_{ij} to denote whether the broken bike b_i is collected during sub-route R_j, the problem can be formulated as follows:

$$\max_{\mathcal{R}} \quad \sum_{b_i \in V} \sum_{R_j \in \mathcal{R}} \delta_{ij} score_i \tag{5}$$

$$str. \quad \sum_{R_j \in \mathcal{R}} \delta_{ij} \leq 1, \quad \forall b_i \in V \tag{6}$$

$$\sum_{R_j \in \mathcal{R}} T_j \leq T \tag{7}$$

$$\sum_{b_i \in V} \delta_{ij} \leq M, \quad \forall R_j \in \mathcal{R} \tag{8}$$

Such a problem of finding k budget constrained connected components with a maximum beneficial score is NP-hard as proven in Lemma 1 below.

Lemma 1 (NP-difficulty). *When time and capacity constrained, collecting broken-sharing-bikes with a maximal beneficial score is NP-hard.*

Proof. The broken sharing bikes collection problem is a combination of broken sharing bike vertex selection and determining the shortest path between the selected vertices. As a consequence, We can reduce our problem of collecting broken-sharing-bikes with maximal beneficial score from the Knapsack Problem (KP) and the Travelling Salesperson Problem (TSP), when time and capacity constrained. We can view each broken sharing bike $b_i \in V$ as an item, with an item size (i.e., Collecting time cost), and an item profit (e.g., a beneficial score contribution). The set V of selected broken sharing bikes is viewed as a knapsack, with a fixed size T (i.e., total working time constraint). Furthermore, not all broken sharing bike $b_i \in V$ have to be visited in the problem. Determining the shortest path between the selected vertices $b_i \in V'$ will be helpful to visit as many vertices as possible in the available time. our goal is to maximize the total score collected. If a recycling worker with not enough time and capacity to collect all possible broken sharing bikes. He knows the number of beneficial scores to expect in each broken bike and wants to maximize the total beneficial score, while keeping the total travel time limited to T. Our problem boils down to an Orienteering Problem problem (OP), which is known to be NP-complete [41]. □

Given it is an NP-hard problem, we develop a heuristic-algorithm to tackle the issue.

2.2 System Overview

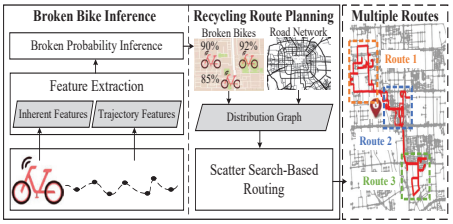


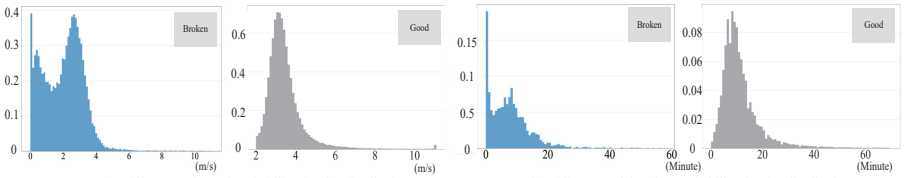
Figure 2: System Overview.

Figure 2 gives an overview of our system, which consists of two main components: (1) *Broken Bicycle Inference*, which calculates broken probability for each sharing bike, which takes the sharing bike's parameters, e.g., the bike inherent feature, and trajectory features, and outputs the bike broken probability and current status (detailed in Section 3) and (2) *Recycling Route Planning* component takes the results of the prediction model, the road network data and the recycling of historical data as input. It establishes the distribution graph of the broken bikes (detailed in Problem Definition) and recommends the optimal route for recycling the broken bike (detailed in Section 4).

3 BROKEN BIKE INFERENCE

Due to the fact that there is only a small proportion of sharing bikes reported as broken by the users, and not all of the reported bikes are truly broken, a broken bicycle inference model is required to detect the real broken bikes for the worker to collect. An inference model under the supervised-learning paradigm is used to assign a broken probability to each bicycle. In the later routing algorithm, the bike with high broken possibility is preferred to collect.

The training bike samples are selected as follows: 1) If a bike is reported as broken by Mobike user and the broken status is confirmed by the worker, we regard it as a broken bike sample; 2) If



(a) Difference of speed probability density distributions (b) Difference of duration probability density distributions

Figure 3: Mobike Trip Characteristics.

a bike is rode repeatedly in a time period (i.e., one month), and the user does not report the status of the bike as a broken, we regard it as a good bike sample.

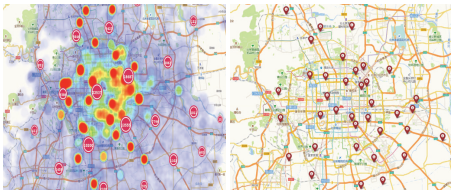
Feature Extraction. Whether a bike is broken can be inferred mainly from two aspects: *1) inherent features*, such as the life time of the bicycle, the number of ridden times, the total duration of cycling, and the number of maintenance; and *2) trajectory features*, which include the average travel speed and trip duration distributions. The selected trajectory features are derived from the analytics of Mobike trajectories. As shown in Figure 3(a), the probability of the average riding speed less than $1m/s$ for the broken bike is much higher than the good bike. This may be because some broken bikes are more cumbersome, the cycling speed will be slower. And from Figure 3(b), the trip duration of the broken bike is much shorter compared with the good one. This may be because the user finds that there is some problem with the bike after scanning the bicycle to ride, thereby terminating the cycling behavior. This phenomenon of user riding helps to determine the state of the sharing bike.

Broken Probability Inference. Since the sharing-bike status takes two values: good or broken (not good), we use a 0-1 valued binary variable y to denote the status outcome, where 1 stands for broken and 0 stands for good. We use p_i to denote the broken probability of the bike b_i. The probability depends on many factors, such as the trip duration and speed of a bike, etc. Such information can be encoded into a feature vector X_i, which is associated with the inherent features and the trajectory features of sharing bikes. Given extracted feature vector X_i, we can estimate the acceptance probability as: $p_i = p(y = broken|X_i)$. Since then, the broken inference task can be formulated as a typical binary classification problem, and the traditional classification model, such as Logistic Regression [11], can be employed.

4 RECYCLING ROUTE PLANNING

After the broken probability of each bike in the service region of a collection site is obtained, the distribution graph is constructed using bike locations with broken probabilities and the road network data. In this section, we describe the *scatter search-based routing algorithm* for the broken bike recycling problem using the constructed distribution graph.

In broken bike recycling problem, the instance size is surely beyond the solvability of standard solver, for example, as shown in Figure 4, there are typically hundreds of broken bikes in some regions, and 39 broken sharing bike collection site in Beijing. The collection



(a) Distribution of Broken Sharing Bikes (b) Distribution of Collection Site

Figure 4: Boken Mobike sharing Bike and collection site Distribution in Beijing

site of broken sharing bike need to occupy certain resources, so each collection site has its own service range. The departure and return locations of the workers are the same collection site in the area. If there is no limit to the capacity of the recycling vehicle and there are no restrictions on the working hours of the recycling workers. Our problem of recycling broken bikes with maximal beneficial score can be converted into a problem of recycling all broken sharing bikes and minimizing the overall recycling path, which can be converted into a tsp problem. However, in the case of working hours and the limited capacity of the recovered vehicle. The problem can be described as workers with not enough time and vehicle capacity to collect all possible broken bikes. He knows the beneficial score which is uniquely defined by the practical broken-bike collection problem (detailed in 2) of each broken bikes, and wants to maximize beneficial scores, while with the working hours and vehicle capacity limited.

Main Idea. Due to the capacity limitation M of the recycling vehicle, the worker can only collect the limited number of bikes during one sub-route. The main idea is that during each sub-route, we first try to collect at most M bikes with high broken probabilities (i.e., high beneficial scores), which are spatially close to each other, and then carefully plan the visiting order, so that the traveling time in each sub-route is minimized. We continuously find such sub-route until the working time is used up. The discovery of each sub-route is explained in following three stages: 1) *broken bike clustering*; 2) *sub-route selection*; 3) *status update*.

Stage 1: Broken Bike Clustering. In this stage, the bikes inferred to be broken are clustered using spatial clustering algorithm, e.g., kMeans [18], so that the broken bikes in each cluster are spatially close to each other. The number of clusters k is computed

according to both recycling vehicle capacity M and The total number of broken bikes in the area n. We initialize k as $k = round(n/M)$.

Stage 2: Sub-route Selection. In this stage, the algorithm finds the best sub-route in each cluster, and the best sub-route over all the clusters is selected. The goodness of a sub-route is defined as the beneficial score per time cost. The sub-route selection in each cluster is conducted in an iteratively way following the scatter search idea. We first select bikes with top M high scores as the initial bike set to recycle, and then design recycling route for it using TSP algorithm. Then we randomly replace a bike in the sub-route with a bike outside the sub-route but inside the cluster, to check whether there is any improvement. This process is repeated N times to obtain a stable sub-route in each cluster.

Stage 3: Status Update. In each iteration, the algorithm puts the best sub-route R_j into the final recycling route set $\mathcal{R}$, and updates the working time by subtracting the time spent recycling broken bikes in R_j and broken bike vertex set V by subtracting the recycled broken bikes in sub-route R_j. The algorithm terminates when working time T is used up, and then returns the recycling route set $\mathcal{R}$ as the recommended broken bikes recycling plan.

Algorithm Design. Algorithm 1 gives the pseudo-code of our scatter search-based heuristic algorithm. In each iteration of the *Scatter Search stage*(Line 2), the algorithm first partition the vertex set of broken bike nodes V into k clusters. The value of K is determined by the number of broken sharing bikes and the capacity of the recycling vehicle. Then, optimal broken sharing bikes collection scheme in the cluster is then selected separately in each independent cluster. When initializing the sub-route set in each cluster, two initialization strategies are employed depending on the value of a α. If the number of broken bikes in the candidate set S_i is greater than recycling vehicle capacity M, the initial recovery of the bicycle is selected using two methods. If tuning parameter α is equal to 1, the algorithm random select M broken bikes point in set S_i. otherwise, the algorithm select M broken bike in set S_i by the probability value of each broken bike as the candidate set C_i (Line 4-10). After selecting the initial result set C_i in cluster i, we use Function RecyRoute to solve the optimal recycling order of the broken bike in the result set and calculate the gain of recycling benefit score g_i. In the set S_i in which the number of each broken bicycle is larger than the recycling vehicle capacity, Take the broken bicycle not included in the set C_i which are randomly selected from the set S_i to replace random replace a broken bike in the set C_i. During the process, we keep track of the set C_i' and C_i, which has the maximum score gain in the iteration. If the number of broken bikes in the candidate set S_i is less than recycling vehicle capacity M, we just calculate the corresponding beneficial score gain. Select the best set C_i which has the maximum score gain from all clusters, and puts the best set R_j in recycling route set $\mathcal{R}$ base on Function RecyRoute. Then, R_i is removed from broken sharing bikes set V, the remaining working time is updated by subtracting the time cost $R_i.time$.At the same time, due to the reduction of the number of broken bikes, the number of clusters is also reduced (Line 11- 19).

Finally, when all the working time budget is used up, the algorithm terminates, and broken sharing-bikes recycling route set $\mathcal{R}$ is returned as the recommended broken bike recycling plan.

Algorithm 1 Scatter Search-based Routing Algorithm

Input: Broken sharing-bikes distribution graph $G = (V, E)$, working time T, parameter α, capacity M, initial number of clusters k and the maximum number of iterations N.
Output: Recycling route set $\mathcal{R}$.

1: **while** $T > 0$ **do**
*//**Stage 1: Broken Bike Clustering***
2: $(\mathcal{S}_1, \mathcal{S}_2, \cdots, \mathcal{S}_k) \leftarrow$ KMEANS(V, k)
*//**Stage 2: Sub-route Selection***
3: **for** $i \leftarrow 1$ to k **do**
4: **if** $\mathcal{S}_i > M$ **then**
5: **if** $\alpha = 1$ **then**
6: Random select M points in $\mathcal{S}_i$ as C_i
7: **else**
8: Select the top M of broken probability in $\mathcal{S}_i$ as C_i
9: **else**
10: Select all point in $\mathcal{S}_i$ as C_i
11: $R_i, g_i \leftarrow$ RECYROUTE(C_i)
12: **for** $l \leftarrow 1$ to N **do**
13: Randomly swap $b_m \in C_i$ by $b' \in \mathcal{S}_i - C_i$ as C_i'
14: $R_i', g_i' \leftarrow$ RECYROUTE(C_i')
15: **if** $g_i' > g_i$ **then**
16: $C_i \leftarrow C_i'$; $R_i \leftarrow R_i'$; $g_i \leftarrow g_i'$
17: $j \leftarrow_i g_i$
*//**Stage 3: Status Update***
18: $\mathcal{R} \leftarrow \mathcal{R} \cup \{R_j\}$; $T \leftarrow T - R_j.time$; $V \leftarrow V - R_j$
19: $k \leftarrow k - 1$
20: return $\mathcal{R}$

Function RECYROUTE(C_i)
$R_i \leftarrow$ TSP(C_i); $g_i \leftarrow \frac{R_i.score}{R_i.time}$
return R_i, g_i

5 EXPERIMENTS

In this section, we conduct extensive experiments to evaluate the effectiveness of our system. We first describe the real dataset used in the paper. Then, we present comparison results with other baseline methods over different values of α and working time constraints. Finally, we present real-world case studies to evaluate our broken bike detection and recycling route planning algorithm.

5.1 Datasets

Road Networks. The road network data in Beijing and Guangzhou, China is collected from Open Street Map [3].

Mobike Order Data. Each Mobike order contains a bike ID, a user ID. The dataset used in the paper includes the entire Mobike orders in the City of Beijing and Guangzhou from 01/08/2018 to 12/31/2018.

Mobike Recycling Data. Each Mobike recycling record contains a bike ID, a worker ID, the start time and the end time to recycle the bike. The dataset is collected in the City of Beijing, with the time span of 01/06/2017 - 12/31/2018.

Mobike Trajectories. Each Mobike trajectory contains a bike ID, a user ID, the time interval of the trajectory, the start/end locations, and a sequence of intermediate GPS points. The dataset includes the

[3] https://www.openstreetmap.org/

entire Mobike trajectory data in the City of Beijing and Guangzhou from 01/08/2018 to 12/31/2018.

5.2 Data Pre-Processing

Data Pre-processing takes the road network, the Mobike order data, Mobike recycling data, and the Mobike trajectories as input, and performs the following three tasks to prepare the data for further processing:

Data Cleaning. *Data Cleaning* cleans the raw order data, trajectories, and recycling data from Mobike. Essentially as a type of crowdsensing data, Mobike trajectories are generated by the GPS modules from mobile phones. As a result, a noticeable portion of trajectories has different data errors, which significantly affect the accuracy of the broken bike inference model. This step cleans the raw trajectories from Mobike users by filtering the noisy GPS points with a heuristic-based outlier detection method [43].

Map-Matching. In this module, we map the GPS points onto the corresponding segments in road networks, which is crucial for the broken sharing bike collection. The Mobike sharing bike can be at any location in the city, e.g., hiding in the residential area or close to the road network, where the parking location of collecting vehicles is usually along with the road network. As a result, we should employ v_w (walking speed) to better characterize the individual bike collecting events. This step evaluates the distance of each broken sharing bikes to the nearest corresponding segments in road networks with a global map matching method [28].

Map Griding. For the ease of assessing the regional rt (registration time), we adopt the griding based method, which simply partitions the map into equal side-length grids [23, 24]. our approach divides the urban area into equal-size grids with a pre-defined side-lengths in 100 meters.

5.3 Effectiveness Evaluation

In this subsection, we study the effectiveness of both broken bike prediction and recycling. Unless mentioned otherwise, the default parameters used in the experiments are: recycling vehicle capacity $M = 20$, the average speed of the worker's walking is $v_w = 1m/s$, and the average speed of the worker's driving is $v_d = 25km/h$.

5.3.1 Broken Bike Prediction.

In the broken bike prediction model, we tried two popular models: logistic regression (LR) [11] and random forest (RF) [2] algorithms. We train the models for different cities and evaluate both methods in terms of Accuracy (ACC) and Area under the Curve of ROC (AUC). Experimental results for Beijing and Guangzhou are shown in Table 1, where we observe that 1) LR outperforms RF slightly and 2) both models get good results, which validates the effectiveness of our feature extraction scheme.

5.3.2 Performance of Different TSP Methods in Recycling Route Planning.

We study the effect of different TSP methods in our recycling route planning. The test data select from Haidian District, Beijing, which the inference model give 537 broken bikes in this area as shown in Figure 7. In this work, we tried five popular models: Simulated Annealing Algorithms (SA) [13], Genetic Algorithms (GA) [31], Ant Colony Optimizations Algorithms (ACO)[9], Lin-Kernighan(LK)[14] and Self-organizing Feature Maps (SOFM)[3].

Table 1: Results of LR and RF

Beijing				
Model	ACC	AUC	Recall	F-score
LR	0.9768	0.9965	0.9763	0.97796
RF	0.9750	0.9934	0.9746	0.97608
Guangzhou				
Model	ACC	AUC	Recall	F-score
LR	0.9757	0.9948	0.9759	0.9756
RF	0.9746	0.9933	0.9745	0.9750

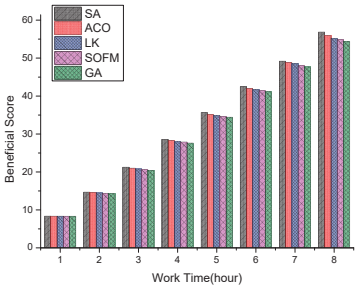


Figure 5: The Evaluation of Different TSP Methods.

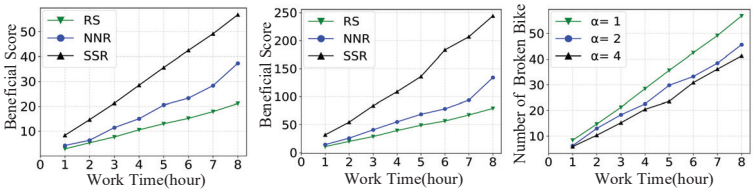
We evaluate five methods in terms of the total beneficial scores in our recycling broken sharing bike problem. Experimental results for Beijing are shown in Figure 5. Our experiments show that for our test data, these TSP algorithms do not make a significant difference as well. SA is slightly more accurate. Therefore, we choose SA as the TSP method in our recycling route planning model.

5.3.3 Recycling Route Planning.

We study the effect of different parameter settings of α and working time, and we compare our method, i.e. Scatter Search-based Routing (SSR), with two other baselines.

• **Baseline 1: Random selection (RS).** If there is no inference model in the collection problem, we assume that workers collect broken bikes according to user reports which occur randomly. We directly take the next car after each collection of a broken bike for collection. When the number of broken bikes collected reaches the vehicle capacity, return to the broken bike station in the area and repeat the random collection process for the next round. The collection process terminates when the total collection time exceeds the working time.

• **Baseline 2: Nearest neighbor routing (NNR).** The location where the broken bike is relatively densely distributed is selected as the starting area for collecting the first broken bike. In NNR, the recycling vehicle starts at the recycling parking spot, repeatedly visits the nearest broken bikes node until the capacity of the vehicle and the working hours of the workers exceed the constraint and returns back to the parking spot.



(a) Results with α =1 (b) Results with α =2 (c) Result of Recycling of Different α

Figure 6: Effectiveness Evaluation.

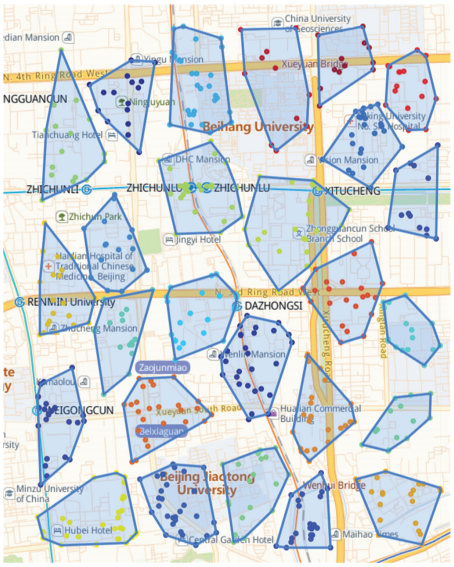


Figure 7: Broken sharing bike distribution cluster in Haidian District.

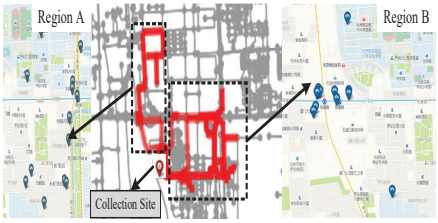


Figure 8: A Real Case Study in Haidian District, Beijing.

Effects on Total Working Time Budget. Figure 6 illustrates the total beneficial scores with different total working time budgets for a worker with Mobike, from 1 Hour to 8 Hour. The experimental results of finding a broken bike based on a random walk of RS is the average value of the income score after the algorithm solves the problem 1000 times independently. From the figure, we make the following observations: 1) the scatter search-based heuristic SSR method performs better than other baseline models. 2) When working hours are between 5 hours and 6 hours, the NNR method will have a useful period of slow growth. This is because, during this time, the NNR method took a long time in a broken bicycle. It is interesting that, because of the slight damage to the bike, the user is not quite sensitive to it and the bike moves within a certain range. This has resulted in a higher recovery cost for workers. The SSR method can adjust the recovery of the beneficial score by controlling the parameter α, which can solve the phenomenon and make the overall recovery more effective as shown in Figure 6(b). Figure 6(c) provides the results with different α settings, It is interesting that, when α is large, the number of recycling broken sharing bikes will be reduced to some extent. Moreover, with a higher α, the number of recycling broken bikes is smaller, but the degree of reduction will gradually decrease. The reason behind these phenomena is that a bicycle with a higher probability of failure prediction has a higher score. Where the value of α is larger, and it is more preferable to collect a bicycle which has higher broken probability when collecting broken sharing bikes. However, the distance between bikes also affects the time cost of recycling, so when the value of a is larger, the number of broken bikes collected will become smaller.

5.4 Case Studies

To better understand the effectiveness of our bike prediction and recycling model, we conduct a field case study. We choose to visit the area near Zhichun Road, Dazhongsi, and Beitucheng subway station in Haidian District, Beijing.

Figure 8 gives the path that Mobike operators use to recycle broken bikes in this area. The workers recovered a total of 32 broken bikes in the vicinity in 8 hours, and mainly concentrated near the temporary parking spots. The traditional recycling methods of worker are similar to the NNS method. The worker first finds the area where the broken bikes are densely distributed near the temporary parking point through the location reported by bikes. Then,

a broken bike in the dense area is selected and recycled according to the scheme of the shortest travel distance of the map navigation. When the target bicycle is found, another broken bike which is closest to the current location is selected for recycling. However, this will make recyclers tend to pay more attention to broken bikes that are closer to temporary parking spots. The distribution of broken bikes in the area is not fully considered, resulting in low efficiency and high cost of recycling. Figure 9 shows the results of the broken recovery path recommended by the SSR method. Mobike's worker recycles the broken bikes in a given area three times and collect 59 broken bikes in an 8 hour working time limit. It is interesting that, the recommended recovery path of the algorithm is not only the broken bikes concentrated near the temporary parking point, but also the broken bikes far from the temporary parking point are also included in the recommended collection of recycling. This is because the algorithm parameters take into account of the overall distribution of the bikes and the optimal recovery sequence in the actual recycling process, and the parameters are tuned according to the efficiency gain of each bike recovery. This makes the recycling of bikes more efficient. At the same time, the two broken bikes at the bottom left of Figure 9 are the broken bikes that are seriously broken in the recovery. One is the chain is broken, and the other is the seat is lost. The confidence of the two cars in the model is 0.99988294 and 0.99961954 respectively. These two bikes belong to the broken bike that is preferentially collected when the value of α is greater than one. This shows that the model is sensitive to bike with a relatively high degree of failure.

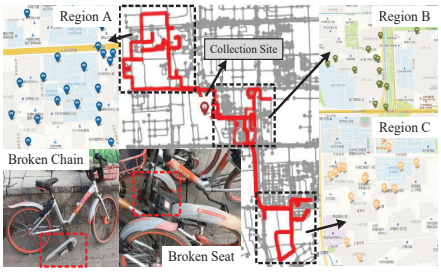


Figure 9: A Real Case Study Base on Scatter Search Model Result in Haidian District, Beijing.

6 RELATED WORK

The research of fault sharing bikes recycling can be summarized in two main areas: 1) Urban Crowd Sourcing, and 2) Route Planning.

Urban Crowd Sourcing. Essentially, we take advantage of the massive Mobike users in a city to perform the fault bike detection task. Similar problems are addressed with the crowdsourcing techniques[19, 26]. For example, The literature [42] quantifies the fragility of cities through detecting the delay in commuting activities using GPS data collected from smartphones. The literatures [34, 36] infer noise levels for locations by smartphone users. The literature [27] proposes a bike sharing network optimization approach by extracting fine-grained discriminative features from human mobility data, point of interests (POI), as well as station network structures. The literatures [7, 15] identify potholes or classify road quality from vehicle's accelerometer data. Differing from the above works, we focus on the problem of broken sharing bikes detection and collecting path planning.

Route Planning. The fault sharing bike recycling problem is related to the multiple Traveling Salesman Problem (mTSP) [1, 38] and orienteering Problem (OP) [4, 39, 41]. mTSP and OP can be considered as a relaxation of our problem, with the capacity or working time restrictions removed. The solutions for these two problems are primarily in two fold: 1) optimal algorithms and 2) heuristic algorithms. In literatures [21, 35], the authors use branch-and-bound to solve instances with less than 20 and 150 vertices, respectively. The authors in [22] use a cutting plane method to obtain better upper bounds. In literatures [10, 12], the authors propose branch-and-cut algorithms. However, the branch-and-cut procedure with instances up to 500 vertices cannot be performed. GAs are relatively stochastic search algorithms based on evolutionary biology and computer science principles [16]. Using GAs to the mTSP problem have several representations, like one chromosome technique [33], the two chromosome technique [32] and the latest two-part chromosome technique. The authors in [25] propose an ant colony optimization approach and a tabu search algorithm. In literature [37], the authors develop a Pareto ant colony optimization algorithm and a multi-objective variable neighborhood search algorithm. In [40], the authors propose a Variable Neighbourhood Search (VNS) algorithm and embed an exact algorithm to deal with a path feasibility subproblem. In [20], the authors present two polynomial size formulations for OP. The authors in [30] discuss several vehicle routing algorithms, and present a heuristic method which searches over a solution space formed by the large number of feasible solutions to an mTSP. The authors in [17] study the adaptive stochastic knapsack problem with deterministic size and stochastic rewards. Their problem objective is to find a sequential inserting policy to maximize the probability of the reward exceeding a threshold value without violating the capacity constraint. In [5], the authors study the adaptive stochastic knapsack problem with items of deterministic reward and stochastic size. Their goal is to maximize expected value while fitting all the items in the knapsack. The authors demonstrate the benefit of an adaptive policy and provide an approximation approach. In [6], the authors study an orienteering problem with stochastic travel times and present adaptive path planning methods to take advantage of dynamically updating data; combine the orienteering problem and optimal path finding into a single model. The authors in [8] discuss the vehicle routing problem with hard time windows and stochastic service times (VRPTW-ST). They adopt the dynamic programming algorithm to account for the probabilistic resource consumption by extending the label dimension and by providing new dominance rules. In this paper two recourse strategies are proposed and the resulting problems are solved by branch-price-and-cut algorithms. However, all of these works cannot be directly used for broken sharing bikes recycling, because these works simply test on benchmark instances and fail to consider the realistic constraints and road network distance.

7 CONCLUSION

In this paper, we introduce a novel approach to detect broken sharing bikes and recommend the appropriate bicycle recycling path to the worker based on the real sharing bikes data collected from Mobike (a major station-less bike sharing system). Our system can address the problem of recycling efficiency of broken sharing bikes in a more realistic fashion, considering the constraints and requirements from sharing bike worker's perspective: 1) working time limitations, 2) vehicle capacity constraints, and 3) broken sharing bike recovery benefit. We also propose a flexible beneficial score function to adjust preferences between the number of bikes recovered and the predicted probability of damage to bikes. The formulated problem is proven to be NP-hard, thus we propose a *scatter search-based heuristic* algorithm. We perform extensive experiments on a large scale Mobike data and demonstrate the effectiveness of our proposed broken sharing bike predict model and bike recycling routing model, where our model can predict the broken sharing bikes with above 97% accuracy and recommends that the number of real broken bikes recovered by the recycling path of the broken bikes is two to three times that of the Mobike traditionally recycling broken bikes.

REFERENCES

[1] Tolga Bektas. 2006. The multiple traveling salesman problem: an overview of formulations and solution procedures. *Omega* 34, 3 (2006), 209–219.

[2] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[3] Łukasz Brocki and Danijel Koržinek. 2007. Kohonen self-organizing map for the traveling salesperson problem. In *Recent Advances in Mechatronics*. Springer, 116–119.

[4] I-Ming Chao, Bruce L Golden, and Edward A Wasil. 1996. The team orienteering problem. *European journal of operational research* 88, 3 (1996), 464–474.

[5] Brian C Dean, Michel X Goemans, and Jan Vondrdk. 2004. Approximating the stochastic knapsack problem: The benefit of adaptivity. In *45th Annual IEEE Symposium on Foundations of Computer Science*. IEEE, 208–217.

[6] Irina Dolinskaya, Zhenyu Edwin Shi, and Karen Smilowitz. 2018. Adaptive orienteering problem with stochastic travel times. *Transportation Research Part E: Logistics and Transportation Review* 109 (2018), 1–19.

[7] Jakob Eriksson, Lewis Girod, Bret Hull, Ryan Newton, Samuel Madden, and Hari Balakrishnan. 2008. The pothole patrol: using a mobile sensor network for road surface monitoring. In *Proceedings of the 6th international conference on Mobile systems, applications, and services*. ACM, 29–39.

[8] Fausto Errico, Guy Desaulniers, Michel Gendreau, Walter Rei, and L-M Rousseau. 2018. The vehicle routing problem with hard time windows and stochastic service times. *EURO Journal on Transportation and Logistics* 7, 3 (2018), 223–251.

[9] Jose B Escario, Juan F Jimenez, and Jose M Giron-Sierra. 2015. Ant colony extended: experiments on the travelling salesman problem. *Expert Systems with Applications* 42, 1 (2015), 390–410.

[10] Matteo Fischetti, Juan Jose Salazar Gonzalez, and Paolo Toth. 1998. Solving the orienteering problem through branch-and-cut. *INFORMS Journal on Computing* 10, 2 (1998), 133–148.

[11] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. 2001. *The elements of statistical learning*. Vol. 1. Springer series in statistics New York.

[12] Michel Gendreau, Gilbert Laporte, and Frederic Semet. 1998. A branch-and-cut algorithm for the undirected selective traveling salesman problem. *Networks: An International Journal* 32, 4 (1998), 263–273.

[13] Xiutang Geng, Zhihua Chen, Wei Yang, Deqian Shi, and Kai Zhao. 2011. Solving the traveling salesman problem based on an adaptive simulated annealing algorithm with greedy search. *Applied Soft Computing* 11, 4 (2011), 3680–3689.

[14] Keld Helsgaun. 2009. General k-opt submoves for the Lin–Kernighan TSP heuristic. *Mathematical Programming Computation* 1, 2-3 (2009), 119–163.

[15] Marius Hoffmann, Michael Mock, and Michael May. 2013. Road-quality classification and bump detection with bicycle-mounted smartphones. In *Proceedings of the 3rd International Conference on Ubiquitous Data Mining-Volume 1088*. CEUR-WS. org, 39–43.

[16] John Holland. 1975. Adaptation in natural and artificial systems: an introductory analysis with application to biology. *Control and artificial intelligence* (1975).

[17] Taylan Ilhan, Seyed MR Iravani, and Mark S Daskin. 2011. The adaptive knapsack problem with stochastic rewards. *Operations research* 59, 1 (2011), 242–248.

[18] Anil K Jain. 2010. Data clustering: 50 years beyond K-means. *Pattern recognition letters* 31, 8 (2010), 651–666.

[19] Shenggong Ji, Yu Zheng, and Tianrui Li. 2016. Urban Sensing Based on Human Mobility. UbiComp 2016. https://www.microsoft.com/en-us/research/publication/urban-sensing-based-human-mobility/

[20] Imdat Kara, Papatya Sevgin Bicakci, and Tusan Derya. 2016. New formulations for the orienteering problem. *Procedia Economics and Finance* 39 (2016), 849–854.

[21] Gilbert Laporte and Silvano Martello. 1990. The selective travelling salesman problem. *Discrete applied mathematics* 26, 2-3 (1990), 193–207.

[22] Adrienne C Leifer and Moshe B Rosenwein. 1994. Strong linear programming relaxations for the orienteering problem. *European Journal of Operational Research* 73, 3 (1994), 517–523.

[23] Yanhua Li, Jun Luo, Chi-Yin Chow, Kam-Lam Chan, Ye Ding, and Fan Zhang. 2015. Growing the charging station network for electric vehicles with trajectory data analytics. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 1376–1387.

[24] Yanhua Li, Moritz Steiner, Jie Bao, Limin Wang, and Ting Zhu. 2014. Region sampling and estimation of geosocial data with dynamic range calibration. In *2014 IEEE 30th International Conference on Data Engineering*. IEEE, 1096–1107.

[25] Yun-Chia Liang, Sadan Kulturel-Konak, and Alice E Smith. 2002. Meta heuristics for the orienteering problem. In *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No. 02TH8600)*, Vol. 1. IEEE, 384–389.

[26] Dongyu Liu, Di Weng, Yuhong Li, Jie Bao, Yu Zheng, Huamin Qu, and Yingcai Wu. 2016. Smartadp: Visual analytics of large-scale taxi trajectories for selecting billboard locations. *IEEE transactions on visualization and computer graphics* 23, 1 (2016), 1–10.

[27] J. Liu, Q. Li, M. Qu, W. Chen, J. Yang, H. Xiong, H. Zhong, and Y. Fu. 2015. Station Site Optimization in Bike Sharing Systems. In *2015 IEEE International Conference on Data Mining*. 883–888. https://doi.org/10.1109/ICDM.2015.99

[28] Yin Lou, Chengyang Zhang, Yu Zheng, Xing Xie, Wei Wang, and Yan Huang. 2009. Map-matching for low-sampling-rate GPS trajectories. In *Proceedings of the 17th ACM SIGSPATIAL international conference on advances in geographic information systems*. ACM, 352–361.

[29] Russell Meddin and Paul DeMaio. 2015. The bike-sharing world map. (2015). URLhttp://www.bikesharingworld.com

[30] RH Mole, DG Johnson, and K Wells. 1983. Combinatorial analysis for route first-cluster second vehicle routing. *Omega* 11, 5 (1983), 507–512.

[31] Yuichi Nagata and Shigenobu Kobayashi. 2013. A powerful genetic algorithm using edge assembly crossover for the traveling salesman problem. *INFORMS Journal on Computing* 25, 2 (2013), 346–363.

[32] Yang-Byung Park. 2001. A hybrid genetic algorithm for the vehicle scheduling problem with due times and time deadlines. *International Journal of Production Economics* 73, 2 (2001), 175–188.

[33] Jean-Yves Potvin, Guy Lapalme, and Jean-Marc Rousseau. 1989. A generalized k-opt exchange procedure for the MTSP. *INFOR: Information Systems and Operational Research* 27, 4 (1989), 474–481.

[34] Zhaokun Qin and Yanmin Zhu. 2016. NoiseSense: A crowd sensing system for urban noise mapping service. In *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 80–87.

[35] R Ramesh, Yong-Seok Yoon, and Mark H Karwan. 1992. An optimal algorithm for the orienteering tour problem. *ORSA Journal on Computing* 4, 2 (1992), 155–165.

[36] Rajib Kumar Rana, Chun Tung Chou, Salil S Kanhere, Nirupama Bulusu, and Wen Hu. 2010. Ear-phone: an end-to-end participatory urban noise mapping system. In *Proceedings of the 9th ACM/IEEE international conference on information processing in sensor networks*. ACM, 105–116.

[37] Michael Schilde, Karl F Doerner, Richard F Hartl, and Guenter Kiechle. 2009. Metaheuristics for the bi-objective orienteering problem. *Swarm Intelligence* 3, 3 (2009), 179–201.

[38] Joseph A Svestka and Vaughn E Huckfeldt. 1973. Computational experience with an m-salesman traveling salesman algorithm. *Management Science* 19, 7 (1973), 790–799.

[39] Tommy Thomadsen and Thomas K Stidsen. 2003. The quadratic selective travelling salesman problem. (2003).

[40] Fabien Tricoire, Martin Romauch, Karl F Doerner, and Richard F Hartl. 2010. Heuristics for the multi-period orienteering problem with multiple time windows. *Computers & Operations Research* 37, 2 (2010), 351–367.

[41] Pieter Vansteenwegen, Wouter Souffriau, and Dirk Van Oudheusden. 2011. The orienteering problem: A survey. *European Journal of Operational Research* 209, 1 (2011), 1–10.

[42] Takahiro Yabe, Kota Tsubouchi, and Yoshihide Sekimoto. 2017. CityFlowFragility: Measuring the Fragility of People Flow in Cities to Disasters using GPS Data Collected from Smartphones. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 1, 3 (2017), 117.

[43] Yu Zheng. 2015. Trajectory data mining: an overview. *ACM Transactions on Intelligent Systems and Technology (TIST)* 6, 3 (2015), 29.

2019 IEEE/RSJ International Conference on Intelligent Robots and Systems

StarNet: Pedestrian Trajectory Prediction using Deep Neural Network in Star Topology

Yanliang Zhu, Deheng Qian, Dongchun Ren, Huaxia Xia

***Abstract*— Pedestrian trajectory prediction is crucial for many important applications. This problem is a great challenge because of complicated interactions among pedestrians. Previous methods model only the pairwise interactions between pedestrians, which not only oversimplifies the interactions among pedestrians but also is computationally inefficient. In this paper, we propose a novel model StarNet to deal with these issues. StarNet has a star topology which includes a unique hub network and multiple host networks. The hub network takes observed trajectories of all pedestrians to produce a comprehensive description of the interpersonal interactions. Then the host networks, each of which corresponds to one pedestrian, consult the description and predict future trajectories. The star topology gives StarNet two advantages over conventional models. First, StarNet is able to consider the collective influence among all pedestrians in the hub network, making more accurate predictions. Second, StarNet is computationally efficient since the number of host network is linear to the number of pedestrians. Experiments on multiple public datasets demonstrate that StarNet outperforms multiple state-of-the-arts by a large margin in terms of both accuracy and efficiency.**

I. INTRODUCTION

Pedestrian trajectory prediction is an important task in autonomous driving [1], [2], [3] and mobile robot applications [4], [5], [6]. This task allows an intelligent agent, e.g., a self-driving car or a mobile robot, to foresee the future positions of pedestrians. Depending on such predictions, the agent can move in a safe and smooth route.

However, pedestrian trajectory prediction is a great challenge due to the intrinsic uncertainty of pedestrians' future positions. In a crowded scene, each pedestrian dynamically changes his/her walking speed and direction, partly attributed to his/her interactions with surrounding pedestrians.

To make an accurate prediction, existing algorithms focus on making full use of the interactions between pedestrians. Early works model the interactions [7], [8], [9], [10] by hand-crafted features. Social Force [7] models several force terms to predict human behaviors. The approach in [8] constructs an energy grid map to describe the interactions in crowded scenes. However, their performances are limited by the quality of manually designed features. Recently, data-driven methods have demonstrated their powerful performance [11], [12], [13], [14]. For instance, Social LSTM [11] considers interactions among pedestrians close to each other. Social GAN [13] models interactions among all pedestrians. Social Attention [14] captures spatio-temporal interactions.

*This work was supported by the Meituan-Dianping Group.

Yanliang Zhu, Deheng Qian, Dongchun Ren and Huaxia Xia are with the Meituan-Dianping Group, Beijing, China. zhuyanliang@meituan.com

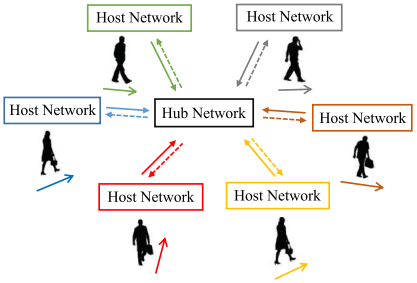


Fig. 1: The structure of StarNet. StarNet mainly consists a centralized hub network and several host networks. The hub network collects movement information and generates a feature which describes joint interactions among pedestrians. Each host network, corresponding to a certain pedestrian, queries the hub network and predicts the pedestrian's trajectory.

Previous methods have achieved great success in trajectory prediction. However, all these methods assume that the complicated interactions among pedestrians can be decomposed into pairwise interactions. This assumption neglects the collective influence among pedestrians in the real world. Thus previous methods tend to fail in complicated scenes. In the meanwhile, the number of pairwise interactions increases quadratically as the number of pedestrians increases. Hence, existing methods are computationally inefficient.

In this paper, we propose a new deep neural network, StarNet, to model complicated interactions among all pedestrians together. As shown in Figure 1, StarNet has a star topology, and hence the name. The central part of StarNet is the hub network, which produces a representation $\mathbf{r}$ of the interactions among pedestrians. To be specific, the hub network takes the observed trajectories of all pedestrians and produces a comprehensive spatio-temporal representation $\mathbf{r}$ of all interactions in the crowd. Then, $\mathbf{r}$ is sent to each host network. Each host network predicts one pedestrian's trajectory. Specifically, depending on $\mathbf{r}$, each host network exploits an efficient method to calculate the pedestrian's

interactions with others. Then, the host network predicts one pedestrian's trajectory depending on his/her interactions with others, as well as his/her observed trajectory.

StarNet has two advantages over previous methods. First, the representation $\mathbf{r}$ is able to describe not only pairwise interactions but also collective ones. Such a comprehensive representation enables StarNet to make accurate predictions. Second, the interactions between one pedestrian and others are efficiently computed. When predicting all pedestrians' trajectories, the computational time increases linearly, rather than quadratically, as the number of pedestrians increases. Consequently, StarNet outperforms multiple state-of-the-arts in terms of both accuracy and computational efficiency.

Our contributions are two-folded. First, we propose to describe collective interactions among pedestrians, which results in more accurate predictions. Second, we devise an interesting topology of the network to take advantage of the representation $\mathbf{r}$, leading to computational efficiency.

The rest of this paper is organized as follows: Section II briefly reviews related work on pedestrian trajectory prediction. Section III formalizes the problem and elaborates our method. Section IV compares StarNet with state-of-the-arts on multiple public datasets. Section V draws our conclusion.

II. RELATED WORK

Our work mainly focuses on human path prediction. In this section, we give a brief review of recent researches on this domain.

Pedestrian path prediction is a great challenge due to the uncertainty of future movements [7], [8], [10], [11], [13], [14], [15]. Conventional methods tackle this problem with manually crafted features. Social Force [7] extracts force terms, including self-properties and attractive effects, to model human behaviors. Another approach [8] constructs an energy map to indicate the traffic capacity of each area in the scene, and uses a fast matching algorithm to generate a walking path. Mixture model of Dynamic pedestrian-Agents (MDA) [10] learns the behavioral patterns by modeling dynamic interactions and pedestrian beliefs. However, all these methods can hardly capture complicated interactions in crowded scenes, due to the limitation of hand-crafted features.

Data-driven methods remove the requirement of hand-crafted features, and greatly improve the ability to predict pedestrian trajectories. Some attempts [11], [13], [14], [26], [27] receive pedestrian positions and predict determined trajectories. Social LSTM [11] devises social pooling to deal with interpersonal interactions. Social LSTM divides pedestrian's surrounding area into grids, and computes pairwise interactions between pedestrians in a grid. Compared with Social LSTM, other approaches [13], [15] eliminate the limitation on a fixed area. Social GAN [13] combines Generative Adversarial Networks (GANs) [16] with LSTM-based encoder-decoder architecture, and sample plausible trajectories from a distribution. Social Attention [14] estimates multiple Gaussian distributions of future positions, then generates candidate trajectories through Mixture Density Network (MDN) [17].

However, existing methods compute pairwise features, and thus oversimplified the interactions in the real word environment. Meanwhile, they suffer from a huge computational burden in crowded scenes. In contrast, our proposed StarNet with novel architecture is capable of capturing joint interactions over all pedestrians, which is more accurate and efficient.

III. APPROACH

In this section, we first describe the formulation of the pedestrian prediction problem. Then we provide the details of our proposed method.

A. Problem Formulation

We assume the number of pedestrians is N. The number of observed time steps is T_{obs}. And the number of time steps to be predicted is T_{pred}. For the i-th pedestrian, his/her observed trajectory is denoted as $O_i = \{\mathbf{p}_i^t \mid t = 1, 2, \cdots, T_{obs}\}$, where $\mathbf{p}_i^t$ represents his/her coordinates at time step t. Similarly, the future trajectory of ground truth is denoted as $F_i = \{\mathbf{p}_i^t \mid t = T_{obs}+1, T_{obs}+2, \cdots, T_{obs}+T_{pred}\}$.

Given such notations, our goal is to build a fast and accurate model to predict the future trajectories $\{F_i\}_{i=1}^N$ of all pedestrians, based on their observed trajectories $\{O_i\}_{i=1}^N$. In other words, we try to find a function mapping from $\{O_i\}_{i=1}^N$ to $\{F_i\}_{i=1}^N$. We employ a deep neural network, which is called StarNet, to embody this function. Specifically, StarNet consists of two novel parts, i.e., a hub network and N host networks. The hub network computes a representation $\mathbf{r}$ of the crowd. Then, each host network predicts the future trajectory of one pedestrian depending on the pedestrian's observed trajectory and $\mathbf{r}$. We first describe the hub network and then present host networks.

B. The hub network

The hub network takes all of the observed trajectories simultaneously and produces a comprehensive representation $\mathbf{r}$ of the crowd of pedestrians. The representation $\mathbf{r}$ includes both spatial and temporal information of the crowd, which is the key to describe the interactions among pedestrians.

Note that our algorithm should be invariant against isometric transformation (translation and rotation) of the pedestrians' coordinates. The invariance against rotation is achieved by randomly rotate our training data during the training process. While the invariance against translation is guaranteed by calculating a translation invariant representation $\mathbf{r}$.

As shown in Figure 2, the hub network produces $\mathbf{r}$ by two steps. First, the hub network produces a spatial representation of the crowd for each time step. The spatial representation is invariant against the translation of the coordinates. Then, the spatial representation is fed into a LSTM to produce the spatio-temporal representation $\mathbf{r}$.

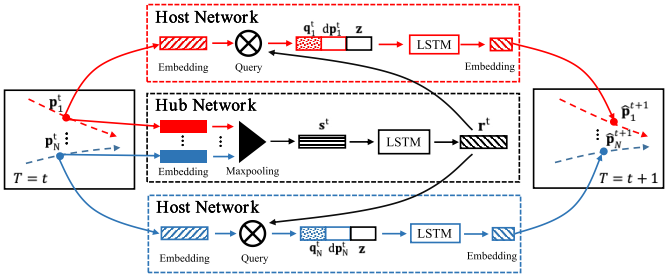


Fig. 2: The process of predicting the coordinates. At time step t, StarNet takes the newly observed (or predicted) coordinates $\{\mathbf{p}_i^t\}_{i=1}^N$ (or $\{\widehat{\mathbf{p}}_i^t\}_{i=1}^N$) and outputs the predicted coordinates $\{\widehat{\mathbf{p}}_i^{t+1}\}_{i=1}^N$.

1) Spatial representation: In the first step, in order to make the representation invariant against translation, the hub network preprocesses the coordinates of pedestrians by subtracting the central coordinates of all pedestrians at time step T_{obs} from every coordinate.

$$\mathbf{p}_i^t \leftarrow \mathbf{p}_i^t - \frac{1}{N}\sum_{n=1}^{N}\mathbf{p}_n^{T_{obs}}. \tag{1}$$

Thus, the centralized coordinates are invariant against translation. Such coordinates of each pedestrian are mapped into a new space using an embedding function $\phi(\cdot)$ with parameters W_1,

$$\mathbf{e}_i^t = \begin{cases} \phi\left(\mathbf{p}_i^t; W_1\right), & if\ t \in [1, T_{obs}], \\ \phi\left(\widehat{\mathbf{p}}_i^t; W_1\right), & if\ t \in \left[T_{obs}+1, T_{obs}+T_{pred}\right], \end{cases} \tag{2}$$

where $\widehat{\mathbf{p}}_i^t$ is the predicted position of the i-th pedestrian at time step t. $\mathbf{e}_i^t$ is the spatial representation of the i-th pedestrian's trajectory at time step t. The embedding function is defined as:

$$\phi\left(\mathbf{x}; W\right) \triangleq W\mathbf{x}. \tag{3}$$

Then, we use a maxpooling operation to combine the spatial representation of all pedestrians, obtaining the spatial representation of the crowd at time step t,

$$\mathbf{s}^t = MaxPooling\left(\mathbf{e}_1^t, \mathbf{e}_2^t, \cdots, \mathbf{e}_N^t\right), \tag{4}$$

Spatial representation $\mathbf{s}^t$ contains information of the crowd at a single time step. However, pedestrians interact with each other dynamically. To improve the accuracy of predictions, a spatio-temporal representation is required.

2) Spatio-temporal representation: In the second step, the hub network feeds a set of spatial representations $\left\{\mathbf{s}^1, \mathbf{s}^2, \cdots, \mathbf{s}^{T_{obs}}\right\}$ of sequential time steps into a LSTM. Then, the LSTM combines all the spatial representations in its hidden state. Thus, the hidden state of the LSTM is a spatio-temporal representation $\mathbf{r}^t$ of all pedestrians. Specifically, we can calculate $\mathbf{r}^t$ as follows:

$$\begin{cases} \mathbf{h}_c^0 = \mathbf{0}, \\ \mathbf{e}^t = \phi\left(\mathbf{s}^t; W_2\right), \\ \left[\mathbf{o}_c^t, \mathbf{h}_c^t\right] = LSTM\left(\mathbf{h}_c^{t-1}, \mathbf{e}^t; W_3\right), \\ \mathbf{r}^t = \phi\left(\mathbf{o}_c^t; W_4\right), \end{cases} \tag{5}$$

where W_2 and W_4 are the embedding weights, W_3 is the weight of LSTM. $\mathbf{o}_c^t$ and $\mathbf{h}_c^t$ are the output and hidden state of the LSTM respectively.

Note that, $\mathbf{r}^t$ depends on the observed trajectories of all pedestrians. Hence, our algorithm is able to consider complicated interactions among multiple pedestrians. This property allows our algorithm to produce accurate predictions. Meanwhile, $\mathbf{r}^t$ is able to be obtained in a single forward propagation of the hub network at each time step. In other words, the time complexity of computing interactions among pedestrians is linear to the number of pedestrians N. This property allows our algorithm to be computationally efficient. By contrast, conventional algorithms compute pairwise interactions, leading to oversimplification of the interactions among pedestrians. Also, the number of pairwise interactions increases quadratically as N increases.

C. The host networks

The spatio-temporal representation $\mathbf{r}^t$ is then employed by host networks. For the i-th pedestrian, the host network first embeds the observed trajectory O_i, and then combines the embedded trajectory with the spatio-temporal representation $\mathbf{r}^t$, predicting the future trajectory. Specifically, the host network predicts the future trajectory by two steps.

First, the host network takes the observed trajectory O_i and the spatio-temporal representation $\mathbf{r}^t$ as input and generates an integrated representation $\mathbf{q}_i^t$,

$$\mathbf{q}_i^t = \begin{cases} \mathbf{r}^t \odot \phi\left(\mathbf{p}_i^t; W_5\right), & if\ t \in [1, T_{obs}], \\ \mathbf{r}^t \odot \phi\left(\widehat{\mathbf{p}}_i^t; W_5\right), & if\ t \in \left[T_{obs}+1, T_{obs}+T_{pred}\right], \end{cases} \tag{6}$$

where W_5 is the embedding weight, and $\odot$ denotes the point-wise multiplication. $\mathbf{q}_i^t$ depends on both the trajectory

of the i-th pedestrian and the interactions between the i-th pedestrian and others in the crowd.

Second, the host network predicts the future trajectory of the i-th pedestrian depending on the observed trajectory O_i and the integrated representation $\mathbf{q}_i^t$. To encourage the host network to produce non-deterministic predictions, a random noise $\mathbf{z}$, which is sampled from a Gaussian distribution with mean 0 and variance 1, is concatenated to the input of the host network. Specifically, the host network encodes the observed trajectory O_i with the hidden state $\mathbf{h}_{ei}^{T_{obs}}$, i.e.,

$$\begin{cases} \mathrm{d}\mathbf{p}_i^0 = \mathbf{0}, \\ \mathrm{d}\mathbf{p}_i^{t-1} = \mathbf{p}_i^t - \mathbf{p}_i^{t-1}, \\ \left[\mathbf{o}_{ei}^t, \mathbf{h}_{ei}^t\right] = LSTM_E\left(\mathbf{h}_{ei}^{t-1}, \left[\mathbf{q}_i^t, \mathrm{d}\mathbf{p}_i^{t-1}\right]; W_6\right), \\ t \in [1, T_{obs}], \end{cases} \tag{7}$$

where $LSTM_E(\cdot)$ with weight W_6 denotes the encoding procedure. Then, the host network proceeds with

$$\begin{cases} \left[\mathbf{o}_{di}^t, \mathbf{h}_{di}^t\right] = LSTM_D\left(\mathbf{h}_{di}^{t-1}, \left[\mathbf{q}_i^t, \mathrm{d}\widehat{\mathbf{p}}_i^{t-1}, \mathbf{z}\right]; W_7\right), \\ \mathrm{d}\widehat{\mathbf{p}}_i^t = \phi\left(\mathbf{o}_{di}^t; W_8\right), \\ \widehat{\mathbf{p}}_i^t = \widehat{\mathbf{p}}_i^{t-1} + \mathrm{d}\widehat{\mathbf{p}}_i^t, \\ t \in [T_{obs}+1, T_{obs}+T_{pred}], \end{cases} \tag{8}$$

where $LSTM_D(\cdot)$ with weight W_7 is the decoding function. W_8 is the embedding weight of the output layer. And the initial states are set according to,

$$\begin{cases} \mathbf{h}_{di}^{T_{obs}} = \mathbf{h}_{ei}^{T_{obs}}, \\ \widehat{\mathbf{p}}_i^{T_{obs}} = \mathbf{p}_i^{T_{obs}}, \\ \mathrm{d}\widehat{\mathbf{p}}_i^{T_{obs}} = \mathbf{p}_i^{T_{obs}} - \mathbf{p}_i^{T_{obs}-1}. \end{cases} \tag{9}$$

D. Implementation Details

The network configuration of StarNet is detailed in TABLE I.

TABLE I: Network Configuration of AstoridNet

Weight	Weight Dimension
W_1	64x2
W_2	64x64
W_3	64x32, 32x1(bias)
W_4	32x64
W_5	64x2
W_6	64x66, 64x1(bias)
W_7	64x74, 64x1(bias)
W_8	2x64

We train the proposed StarNet with the loss function applied in [13]. Specifically, at the training stage, StarNet produces multiple predicted trajectories for each pedestrian. Each predicted trajectory $\{\widehat{F}_{ik}\}_{k=1}^K$ has a distance to the ground truth trajectory F_i. Only the smallest distance is minimized. Mathematically, the loss function is,

$$L = \frac{1}{NT_{pred}} min_{k=1}^K \sum_{j=1}^{N} \sum_{t=T_{obs}+1}^{T_{obs}+T_{pred}} \left(\widehat{\mathbf{p}}_{jk}^t - \mathbf{p}_j^t\right)^2, \tag{10}$$

where K is the number of sampled trajectories. This loss function improves the training speed and stability. Moreover, we employ an Adam optimizer and set the learning rate to 0.0001.

In practice, all host networks share the same weights, since pedestrians in a scenario have the same behavioral patterns, such as variable-speed movement, sharp turning and so on. In our approach, we use shared weights to learn the aforementioned behavioral patterns. Each host network contains specific LSTM state which captures certain pedestrian's behavior, and predicts the pedestrian's future trajectory. The observed trajectories of all pedestrians form a batch, which is fed into one single implementation of the host network. In this way, the prediction for all pedestrians is able to be obtained in a single forward propagation.

IV. EXPERIMENTS

We evaluate our model on two human crowded trajectory datasets: ETH [24] and UCY [25]. These datasets have 5 sets with 4 different scenes. In these scenes, there exist challenging interactions, such as walking side by side, collision avoidance and changing directions. Following the settings in [11], [13], [14], we train our model on 4 sets and test on the remaining one.

We compare our StarNet with three state-of-the-arts including Social LSTM, Social GAN and Social Attention. Besides, we test the basic LSTM-based encoder-decoder model, which does not consider the interactions among pedestrians, as a baseline.

Following [11], [13], [14], we compare these methods in terms of the Average Displacement Error (ADE) and Final Displacement Error (FDE). The ADE is defined as the mean Euclidean distance between predicted coordinates and the ground truth. Specifically, all methods output 8 coordinates uniformly sampled from the predicted trajectory. Then the distance between such 8 points with the ground truth is accumulated as the ADE. The FDE is the distance between the final point of the predicted trajectory and the final point of the ground truth. All these methods are trained with the loss Eq. (10) to deal with multimodal distribution during evaluation. Besides, we compare the computational time of all these methods. All experiments are conducted on the same computational platform with an NVIDIA Tesla V100 GPU.

A. Experimental Results

1) Accuracy: As shown in TABLE II, StarNet outperforms the others in most cases. A possible explanation is that StarNet considers the collective influence among pedestrians all together to make more accurate predictions. In comparison, other state-of-the-arts only model the pairwise interactions between pedestrians.

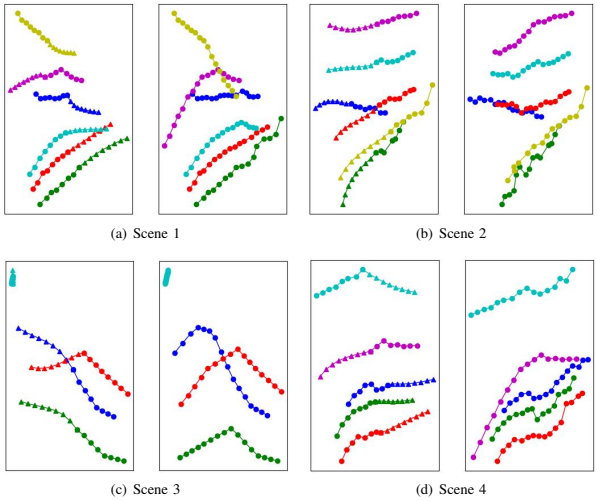
Interestingly, we notice that the test datasets include multiple senses. In these scenes, StarNet has the smallest variances of ADE and FDE, which means that StarNet is robust against the changes of scenes.

TABLE II: Comparison of Prediction Errors

Metric	Dataset	LSTM	Social LSTM	Social GAN	Social Attention	StarNet (Ours)
ADE	ZARA-1	0.25	0.27	**0.21**	1.66	0.25
	ZARA-2	0.31	0.33	0.27	2.30	**0.26**
	UNIV	0.36	0.41	0.36	2.92	**0.21**
	ETH	0.70	0.73	0.61	2.45	**0.31**
	HOTEL	0.55	0.49	0.48	2.19	**0.46**
Average ADE	-	0.43	0.45	0.39	2.30	**0.30**
Variance of ADE	-	0.028	0.026	0.021	0.166	**0.008**
FDE	ZARA-1	0.53	0.56	**0.42**	2.64	0.47
	ZARA-2	0.65	0.70	0.54	4.75	**0.53**
	UNIV	0.77	0.84	0.75	5.95	**0.40**
	ETH	1.45	1.48	1.22	5.78	**0.54**
	HOTEL	1.17	1.01	0.95	4.94	**0.91**
Average FDE	-	0.91	0.91	0.78	4.81	**0.57**
Variance of FDE	-	0.118	0.101	0.802	1.394	**0.031**

TABLE III: Comparison of Computational Time

Metric	LSTM	Social LSTM	Social GAN	Social Attention	StarNet (Ours)
Inference Time (Seconds)	**0.029**	0.504	0.202	3.714	**0.073**
Number of Paramters (Kilo)	**22.87**	156.06	108.03	874.95	**31.90**



(a) Scene 1 (b) Scene 2

(c) Scene 3 (d) Scene 4

Fig. 3: Predicted trajectories and the corresponding ground truths. Different colors indicate different trajectories. The trajectories of ground truth are labeled with dots. The predicted trajectories are labeled with triangles.

To assess StarNet qualitatively, we illustrate the prediction results in 4 scenes, as shown in Figure 3. In each scene, the left sub-figure presents the observed trajectories and the predicted trajectories of all pedestrians. The right sub-figure shows the trajectories of ground truth.

We can observe that StarNet could handle complicated

interactions among pedestrians. Most predicted trajectories accurately reflect the pedestrians' movements and have no collisions with other trajectories. However, there are some failure cases due to the multimodal distribution of future trajectories. For example, in 3(c), the predictions for the blue and green trajectories fail to match the ground truth. We argue that although these predicted trajectories do not match the ground truth, these trajectories are still plausible in crowded scenes.

2) Computational time cost: When deployed in mobile robots and autonomous vehicles, the prediction algorithm needs to be invoked with a high frequency. Hence the computational time of the prediction algorithm is a crucial property.

As shown in TABLE III, the basic LSTM model is the fastest model since the model takes no interactions among pedestrians into consideration. StarNet is the second fastest model. Specifically, StarNet is 51 times faster than Social Attention, 7 times faster than Social LSTM, and 3 times faster than Social GAN. Meanwhile, the number of parameters employed by StarNet is less than state-of-the-arts by a large margin. StarNet is computationally efficient since the interpersonal interactions among pedestrians are computed in a single forward propagation, as discussed in Section II.

V. CONCLUSION

In this paper, we propose StarNet, which has a star topology, for pedestrian trajectory prediction. StarNet learns complicated interpersonal interactions and predicts future trajectories with low time complexity. We apply a centralized hub network to model the spatio-temporal interactions among pedestrians. Then the host network takes full advantage of the spatio-temporal representation and predicts pedestrians' trajectories. We demonstrate that StarNet outperforms state-of-the-arts in multiple experiments.

REFERENCES

[1] D. Ferguson, D. Michael, U. Chris and K. Sascha, "Detection, prediction, and avoidance of dynamic obstacles in urban environments," *in 2008 IEEE International Conference on Intelligent Vehicles Symposium (IVS)*. IEEE, 2008, pp. 1149-1154.

[2] Y. Luo, P. Cai, A. Bera, D. Hsu, W. S. Lee and D. Manocha, "Porca: Modeling and planning for autonomous driving among many pedestrians," *IEEE Robotics and Automation Letters*, vol. 3, no. 4, pp. 3418-3425, 2018.

[3] F. Large, D. Vasquez, T. Fraichard and C. Laugier, "Avoiding cars and pedestrians using velocity obstacles and motion prediction," *in 2004 IEEE International Conference on Intelligent Vehicles Symposium (IVS)*. IEEE, 2004, pp. 375-379.

[4] B. D. Ziebart, N. Ratliff, G. Gallagher, C. Mertz, K. Peterson, J. A. Bagnell, M. Hebert, A. K. Dey and S. Srinivasa, "Planning-based prediction for pedestrians," *in 2009 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2009, pp. 3931-3936.

[5] P. Trautman and A. Krause, "Unfreezing the robot: Navigation in dense, interacting crowds," *in 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2010, pp. 797-803.

[6] N. E. D. Toit and J. W. Burdick, "Robot motion planning in dynamic, uncertain environments," *IEEE Transactions on Robotics*, vol. 28, no. 1, pp. 101-115, 2012.

[7] D. Helbing and P. Molnar, "Social force model for pedestrian dynamics," *Physical review E*, vol. 51, no. 5, pp. 4282, 1995.

[8] S. Yi, H. Li and X. Wang, "Understanding pedestrian behaviors from stationary crowd groups," *in 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2015, pp. 3488-3496.

[9] S. Yi, H. Li and X. Wang, "Pedestrian behavior modeling from stationary crowds with applications to intelligent surveillance," *IEEE transactions on image processing*, vol. 25, no. 9, pp. 4354-4368, 2016.

[10] B. Zhou, X. Wang and X. Tang, "Understanding collective crowd behaviors: Learning a mixture model of dynamic pedestrian-agents," *in 2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2012, pp. 2871-2878.

[11] A. Alahi, K. Goel, V. Ramanathan, A. Robicquet, F. Li and S. Savarese, "Social lstm: Human trajectory prediction in crowded spaces," *in 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE 2016, pp. 961-971.

[12] H. Wu, Z. Chen, W. Sun, B. Zheng and W. Wang, "Modeling trajectories with recurrent neural networks," *in 28th International Joint Conference on Artificial Intelligence (IJCAI)*. 2017, pp. 3083-3090.

[13] A. Gupta, J. Johnson, F. Li, S. Savarese and A. Alahi, "Social GAN: Socially acceptable trajectories with generative adversarial networks," *in 2018 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2018, pp. 2255-2264.

[14] A. Vemula, K. Muelling and J. Oh, "Social attention: Modeling attention in human crowds," *in 2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2018, pp. 1-7.

[15] Y. Xu, Z. Piao and S. Gao S, "Encoding crowd interaction with deep neural network for pPedestrian trajectory prediction," *in 2018 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2018, pp. 5275-5284.

[16] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville and Y. Bengio, "Generative adversarial nets," *in 28th Conference on Neural Information Processing Systems (NIPS)*. 2014, pp. 2672-2680.

[17] C. M. Bishop, "Mixture density networks," *Technical Report NCRG/4288*, Aston University, Birmingham, UK, 1994.

[18] D. Ha and D. Eck, "A neural representation of sketch drawings," *arXiv preprint* arXiv:1704.03477, 2017.

[19] E. Schmerling, K. Leung, W. Vollprecht and M. Pavone, "Multimodal probabilistic model-based planning for human-robot interaction," *in 2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2018, pp. 1-9.

[20] K. Cho, B. V. Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," *arXiv preprint* arXiv:1406.1078, 2014.

[21] K. Cho, B. V. Merrienboer, D. Bahdanau and Y. Bengoi, "On the properties of neural machine translation: Encoder-decoder approaches," *arXiv preprint* arXiv:1409.1259, 2014.

[22] D. Bahdanau, J. Chorowski, D. Serdyuk, P. Brakel and Y. Bengio, "End-to-end attention-based large vocabulary speech recognition," *in 2015 International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2016, pp. 4945-4949.

[23] C. R. Qi, H. Su, K and J. G. Leonidas, "Pointnet: Deep learning on point sets for 3d classification and segmentation," *in 2017 Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, 2017, pp. 652-660.

[24] S. Pellegrini, A. Ess, K. Schindler and L. V. Gool, "You'll never walk alone: Modeling social behavior for multi-target tracking," *in 2009 IEEE International Conference on Computer Vision (ICCV)*. IEEE, 2009, pp. 261-268.

[25] A. Lerner, Y. Chrysanthou and D. Lischinski, "Crowds by example," *Computer Graphics Forum*, vol. 26, no. 3, pp. 655-664, 2007.

[26] D. Varshneya, G. Srinivasaraghavan, "Human trajectory prediction using spatially aware deep attention models," *arXiv preprint* arXiv:1705.09436, 2017.

[27] T. Fernando, S. Denma, S. Sridharan and C. Fookes, "Soft+hardwired attention: An lstm framework for human trajectory prediction and abnormal event detection," *arXiv preprint* arXiv:1702.05552, 2017.

2019 INFORMS Annual Meeting

informs
2019 INFORMS Annual Meeting
Seattle, WA, October 20-23, 2019
| ISBN 978-0-9906153-1-6
https://doi.org/10.1287/infp.2019.XXXX
pp. 000–000

A Two-Stage Fast Heuristic for Food Delivery Route Planning Problem

Huanyu Zheng, Shengyao Wang, Ying Cha, Feng Guo, Jinghua Hao, Renqing He, Zhizhao Sun
Meituan-Dianping Group, No. 4 Wangjing East Street, Chaoyang District, Beijing, China,
{zhenghuanyu, wangshengyao, chaying, guofeng13, haojinghua, herenqing,
sunzhizhao}@meituan.com

Abstract Online food-delivery platforms are expanding choice, allowing customers to order from a wide variety of restaurants. As an industrial level technology, route planning algorithm is required to be fast enough. This paper proposes a two-stage fast heuristic for route planning, which solves the problem at millisecond level. To speed up the algorithm, we further utilize geographic information so that invalid search attempts are prevented. Finally, we compare our algorithm with brute-force algorithm and several state-of-the-art algorithms to show its effectiveness and efficiency.

Keywords food delivery, pickup and delivery, time windows, heuristics

1. Introduction

"Tap, order, and eat, all at home." Food ordering and delivery is a fast growing market all over the world. Worldwide, food ordering and delivery companies, such as Grubhub, Uber Eats, and Just Eat, are developing a $94 billion online food ordering and delivery business [4]. In China, over 300 billion customers order food from Meituan-Dianping food delivery platform with more than 3.6 million restaurants to choose, exceeding 24 million daily orders and deliveries in 2018 [9].

As shown in FIGURE 1, after a customer orders a meal, the food delivery platform pushes the order to a restaurant. At the same time, the platform dispatches this order to a driver, and plans a route for him. The dispatching system is shown in FIGURE 2. A real route is shown in FIGURE 3, in which the driver is planned to pick up four meals before delivering two of the meals to corresponding customers. Then, the driver pick up the last meal and deliver the rest meals to customers. At rush hours like lunch time, a driver may run with 10 orders for example, which means that he has to scurry across 10 restaurants and 10 customers. Under this setting, there are 2.38×10^{15} ways of route planning, which is hard to solve in limited time.

In this paper, we introduce an algorithm to find a satisfactory route, aiming to minimize delays and the route length. The algorithm is required to be highly efficient because it is an essential part of dispatching algorithm. Different from traditional logistics, food delivery dispatching algorithm is an online algorithm, which matches thousands of orders with thousands of drivers each time. Thus, route planning algorithm has to plan a route within milliseconds.

FIGURE 1. Food ordering and delivery process.

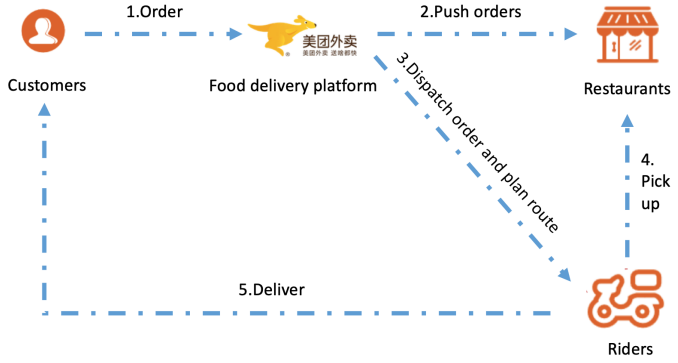


FIGURE 2. Dispatching algorithm.

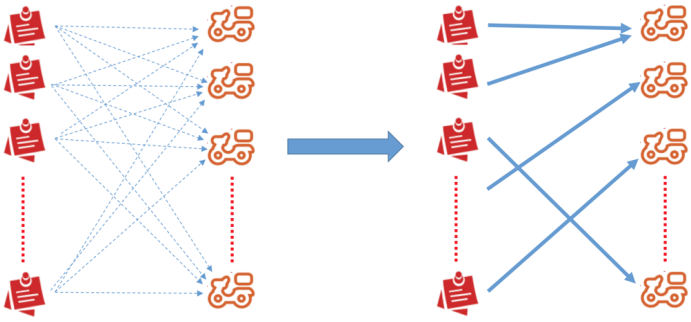
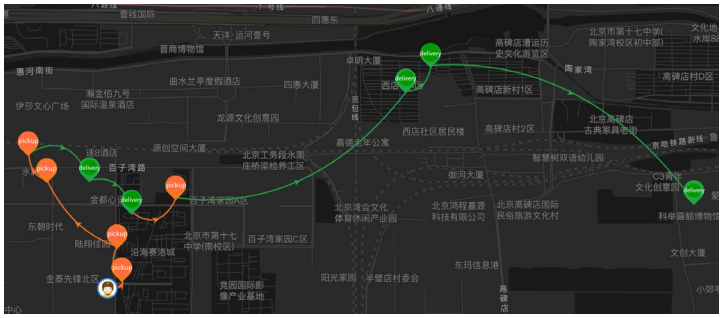


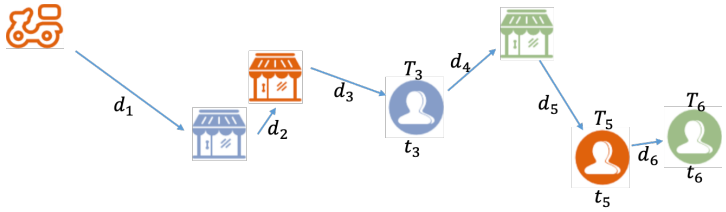
FIGURE 3. A real route.



2. Literature Review

Food delivery route planning problem is rarely studied in the literature. The most relevant literature is the Pickup and Delivery Problem with Time Windows (PDPTW), which models

FIGURE 4. A typical route.



a fleet of vehicles serving a collection of transportation requests. Each request specifies a pickup location and a delivery location. Vehicles are routed to serve all requests, optimizing a certain objective function such as total distance traveled, with precedence constraint and capacity constraint.

Problem in this paper can be modeled as the single vehicle PDPTW, which considers only one vehicle and several customers. [14] propose a variable-depth search algorithm, which produces near optimal solutions most of the time, but may end up with an infeasible solution. Then, they have to spend a large computation time with simulated annealing. [5] present a genetic algorithm, a simulated annealing algorithm and a hill climbing approach. They find simulated annealing algorithm is superior to other algorithms but requires longer running time.

A mathematical formulation of the PDPTW involving a single depot is given in [2]. [13] further formulate a general version of the PDPTW, and present a survey of the problem. In the past two decades, several exact algorithms and heuristics are designed to solve the PDPTW. Exact algorithms include column generation [2], branch-and-cut [7], and branch-and-cut-and-price [11, 1]. Heuristics include tabu search [10], insertion-based heuristic [8], adaptive large neighborhood search [12] and simulated annealing [15].

The PDPTW is applied to problems arising in logistics and public transit, such as transporting goods [2] and home health care [6]. The problems are usually in large scale and are acceptable to be solved in hours. However, food delivery route planning problem demands very low computational complexity. As a core algorithm supporting the food delivery platform, the time complexity of route planning algorithm is limited in only several milliseconds. In other words, algorithms proposed by papers above are not applicable to our problem. In this paper, we propose a new heuristic approach for PDPTW problem to meet this restrict time complexity requirement.

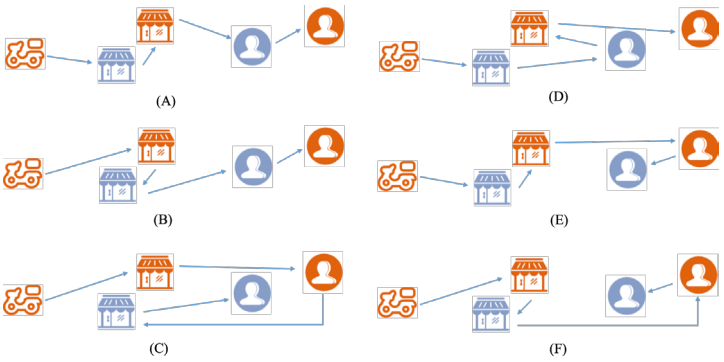
3. Algorithm

In this section, we present our Two-Stage Fast Heuristic (TSFH), including initialization and local search strategies. FIGURE 4 shows a typical route, where t_i is the Estimated Time of Arrival (ETA) [3] of a delivery point, which is shown to customer and restaurant as soon as the order generates. T_i is estimated time of driver's action, calculated by our route planning algorithm, and d_i is traveling distance from previous point to i-th point. Then we have Formula 1 presenting our objective, which minimizes delays and route length.

$$\min \sum_{i=1}^{n} [\max(T_i - t_i, 0) + d_i] \tag{1}$$

The problem has two constraints. (1) precedence constraint: A driver has to pick up meals before delivery. (2) capacity constraint: The total capacity of a driver is limited during the route.

FIGURE 5. Possible ways of insertion.



The TSFH has two stages. Stage I initializes one feasible solution with greedy search mechanism. To speed up the insertion, we utilize geographic knowledge to avoid invalid search. Stage II adjusts the solution with two local search strategies.

3.1. Stage I: Initialization

3.1.1. Initialization with Greedy Insertion The initialization stage can be described as follows. First, we sort orders according to their ETA. Then, we plan the first order. Due to precedence constraint, the only plan is to plan pickup point before delivery point. For the following orders, their pickup and delivery points are inserted into the route according to the objective. For instance, as shown in FIGURE 5, we have 6 ways to insert the second order to the route. We greedily insert pickup point and delivery point to minimize delays and route length. According to this criterion, FIGURE 5(A) is the optimal way to insert. After all points are inserted, we formulate a feasible route.

3.1.2. Speeding up with Geographic Information In China, restaurants are geographically close to each other. For example, restaurants at a central business district building may serve 60% customers within 5 kilometers. Also, customers are geographically close, most customers are gathered in certain communities. Thus, we can cluster pickup and delivery points by hierarchical clustering. With these clusters, we speed up the initialization stage by reducing "bad" insertion attempts.

Clustering algorithm is described as follows, where D is a given range, say 100 meters. For each point i, if it is not classified, it generates a new group and makes itself a center point. For each point j, if it not classified, it is classified into group i if $d_{ij} < D$; if it is classified to group k and is not a center point, then if $d_{ij} < d_{kj}$, we reclassify it into group i.

Lemma 1 (Insertion Before Own Group). *If point j is classified to group i, then inserting point j to groups before group i is worse than inserting point j to group i.*

Proof. Point j is classified to group i. Inserting point j to group k before group i must be worse than directly inserting it to group i, because the route length is longer, but no delivery point benefits from shorter delays.

FIGURE 6 gives an example illustrating the above lemma. Pickup and delivery points are classified into three groups. Consider insertion of 'green' pickup and delivery points, given 'blue' points and 'orange' points inserted beforehand. We can see from FIGURE 6, inserting a point before its own group (FIGURE 6 (B)) is always worse than inserting it to its own group (FIGURE 6 (A)), as the driver travels more and customers may suffer from more delays.

FIGURE 6. Insertion before own group.

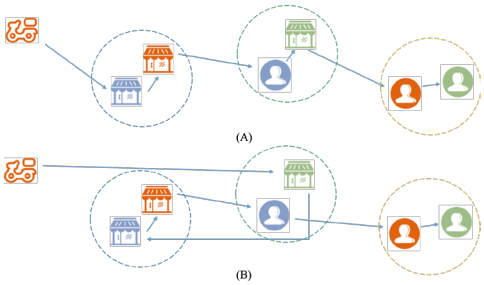
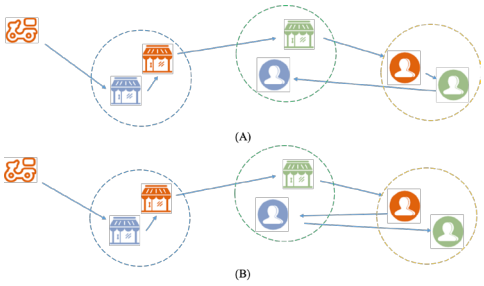


FIGURE 7. Insertion after own group.



Lemma 2 (Insertion After Own Group). *If point j is classified to group i, then we can always find an insertion better than the insertion of point j to group k after group i.*

Proof. Point j is classified to group i. Inserting point j to group k after group i must be worse than inserting it between group k and group $k+1$ (if group k is the last group, then inserting it to the end), because the route length is longer, but no delivery point benefits from shorter delays.

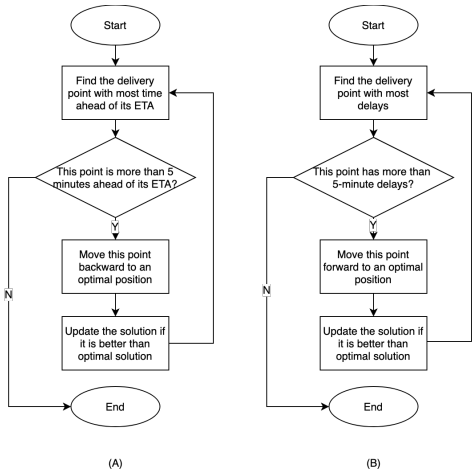
FIGURE 7 gives an example illustrating the above lemma. Pickup and delivery points are classified into three groups. Consider insertion of 'blue' pickup and delivery points, given 'green' points and 'orange' points inserted beforehand. We can see from FIGURE 7, inserting a point between a group after its own group (FIGURE 7 (B)) is always worse than inserting it to the last position (FIGURE 7 (A)), as the driver travels more and customers may suffer from more delays.

From Lemma 1 and Lemma 2, we conclude that point j is only worth to insert when it is inserted into its group, say group i, or between groups after group i. This conclusion reduces invalid insertion attempts and speeds up the algorithm.

3.2. Stage II: Local Search

FIGURE 8 shows the local search stage. After initialization stage, the local search improves the route by looking for better solutions at solution neighborhood. Our algorithm considers two kinds of neighborhood. First, we find delivery points with most delays and move them backward to an optimal position, which is shown in FIGURE 8 (A). Second, we find delivery points with most sufficient time and move them forward to an optimal position, which is shown in FIGURE 8 (B). Each time we find a better solution, the best solution is replaced.

FIGURE 8. Local search.



4. Numerical Results and Experiments

In this section, we provide numerical experiments to compare different algorithms. First, we compare initialization without speeding up technique and fast initialization with speeding up technique to show the effectiveness of initialization stage. Then, we use brute-force search to find the optimal solution to each problem as a baseline. Moreover, we compare our TSFH algorithm with the optimal solution to show our TSFH can generate near optimal solutions in limited time. Moreover, the TSFH is compared with several state-of-the-art algorithms to show its effectiveness and efficiency.

Instances are randomly sampled from real route planning problems. We partition the instance set into three sets, '< 10', '10 to 20', and '> 20', according to the number of pickup and delivery points.

In the following tables, we show performance of algorithms by total score and average time. Total score is the sum of delays (in minutes) and route length (in kilometers). Average time is the computation time of one instance (in milliseconds).

All experiments are run on a MacBook Pro with 2.2 GHz processors / 16 GB RAM in Mac-OS. The algorithms are coded in Java using Eclipse.

4.1. Comparison of Initialization and Fast Initialization

To show the effectiveness of initialization and local search, and efficiency of speeding up technique, we compare initialization and fast initialization using geographic information.

From TABLE 1, we can see that initialization can be sped up considering geographic information. For example, the average running time of the '10 to 20' instance set is reduced from 0.37ms to 0.21ms, which is 43.2% faster. Moreover, effectiveness of initialization is not harmed by speeding up as total score is nearly the same. Therefore, we have shown that our algorithm can generate near optimal solutions within several milliseconds.

4.2. Optimal Solutions by Brute-force Algorithm

The optimal solution can be solved using a brute-force search algorithm. Consider a driver with n pickup tasks and n delivery tasks, the complexity of brute-force search algorithm can be derived as follows. A solution from $2n$ points' full permutation can only be feasible when

TABLE 1. Numerical results of initialization and fast initialization.

Instance	Algorithm	Total Score	Delays	Route Length	Average Time
> 20	Initialization	240.88	232.17	8.71	2.60
> 20	Fast Initialization	242.48	233.74	8.73	1.08
10 to 20	Initialization	46.29	41.35	4.94	0.37
10 to 20	Fast Initialization	46.11	41.19	4.93	0.21
< 10	Initialization	28.59	24.09	4.50	0.14
< 10	Fast Initialization	28.67	24.18	4.50	0.10

TABLE 2. Numerical results of our TSFH and brute-force algorithm.

Instance	Algorithm	Total Score	Delays	Route Length	Average Time
> 20	TSFH	223.30	214.79	8.51	7.08
10 to 20	TSFH	42.48	37.65	4.83	1.01
< 10	TSFH	27.35	22.94	4.40	0.41
< 10	Brute-force	27.17	22.79	4.38	199.31

TABLE 3. Numerical results of our TSFH and the state-of-the-art algorithms.

Instance	Algorithm	Total Score	Delays	Route Length	Average Time
> 20	TSFH	**223.30**	214.79	8.51	**7.08**
> 20	VDS [14]	877.80	869.63	8.18	25.96
> 20	SA [5]	1195.50	1185.74	9.75	64.71
10 to 20	TSFH	**42.48**	37.65	4.83	**1.01**
10 to 20	VDS [14]	77.34	72.66	4.68	6.23
10 to 20	SA [5]	101.02	96.18	4.84	35.59
< 10	TSFH	**27.35**	22.94	4.40	**0.41**
< 10	VDS [14]	35.17	30.95	4.23	3.51
< 10	SA [5]	36.87	32.62	4.24	25.64
< 10	Brute-force	27.17	22.79	4.38	199.31

each pair of pickup and delivery points follows precedence constraint. Thus, the complexity is $O\left(\frac{(2n)!}{2^n}\right)$, if capacity constraint is not considered. As the complexity grows dramatically with the number of pickup or delivery points, we only provide results of '< 10' instance sets.

4.3. Comparison of Our TSFH and Brute-force Algorithm

In this subsection, we compare the TSFH with brute-force algorithm, which generates optimal solutions to each instance. TABLE 2 shows the results. From '< 10' instance set, we can find that our full algorithm uses only 0.21% of average time but generates nearly the same performance than brute-force algorithm. We can also see that the average computation time of the TSFH is limited in milliseconds in '10 to 20' and '< 10' instances. As most instances generated in real life have less than 20 pickup and delivery points, we show the TSFH is efficient enough to solve food delivery route planning problem within several milliseconds.

4.4. Comparison of Our TSFH and the State-of-the-art Algorithms

In this subsection, we compare the best results of our TSFH to those of variable-depth search (VDS) [14] and simulated annealing (SA) [5]. The comparative results are listed in Table 3. We can see that the TSFH outperforms VDS and SA in terms of total score and average running time. Particularly, in '< 10' instances, the TSFH reaches near optimal solutions, which is 0.7% above optimal, while VDS and SA are 29.4% and 35.7% above optimal respectively. Moreover, the TSFH also solves '< 10' instances in 0.41ms per instance, while VDS and SA solve those instances in 3.51ms and 25.64ms, respectively. Real life food delivery route planning demands high running speed. The TSFH can solve all '< 10' instances in 1 millisecond, while the computation time of VDS and SA is not acceptable. Therefore, we conclude that the TSFH is more effective and efficient than the state-of-art algorithms, and more suitable to be applied to real life problem solving.

5. Conclusions

In this paper, food delivery route planning problem is modeled as the single vehicle pickup and delivery problem with time windows (PDPTW). Unlike traditional approaches solving the single vehicle PDPTW, our paper presents a two-stage fast heuristic (TSFH). In stage I, a feasible and near optimal solution is generated in a greedy heuristic way. With restaurants and customers location cluster information, our algorithm is accelerated by avoiding bad searching attempts. In stage II, we improve the solution in stage I by exploiting two neighborhoods. From our numerical results, we represent the effectiveness and efficiency by comparing our results with those from brute-force algorithm and some best algorithms in the literature. Our TSFH algorithm generates near optimal solutions within milliseconds.

References

[1] Baldacci R, Bartolini E, Mingozzi A (2011) An exact algorithm for the pickup and delivery problem with time windows. *Operations Research* 59(2):414–426.
[2] Dumas Y, Desrosiers J, Soumis F (1991) The pickup and delivery problem with time windows. *European Journal of Operational Research* 54(1):7–22.
[3] Fang Z, Huang L, Wierman A (2017) Prices and subsidies in the sharing economy. *Proceedings of the 26th International Conference on World Wide Web*, 53–62 (International World Wide Web Conferences Steering Committee).
[4] Hirschberg C, Rajko A, Schumacher T, Wrulich M (2016) The changing market for food delivery. https://www.mckinsey.com/industries/high-tech/our-insights/the-changing-market-for-food-delivery.
[5] Hosny MI, Mumford CL (2010) The single vehicle pickup and delivery problem with time windows: intelligent operators for heuristic and metaheuristic algorithms. *Journal of Heuristics* 16(3):417–439.
[6] Liu R, Xie X, Augusto V, Rodriguez C (2013) Heuristic algorithms for a vehicle routing problem with simultaneous delivery and pickup and time windows in home health care. *European Journal of Operational Research* 230(3):475–486.
[7] Lu Q, Dessouky MM (2004) An exact algorithm for the multiple vehicle pickup and delivery problem. *Transportation Science* 38(4):503–514.
[8] Lu Q, Dessouky MM (2006) A new insertion-based construction heuristic for solving the pickup and delivery problem with time windows. *European Journal of Operational Research* 175(2):672–687.
[9] Meituan-Dianping Group (2019) Meituan to invest rmb 11 billion to support merchant development. https://www.prnewswire.com/news-releases/meituan-to-invest-rmb11-billion-to-support-merchant-development-300782802.html.

[10] Nanry WP, Barnes JW (2000) Solving the pickup and delivery problem with time windows using reactive tabu search. *Transportation Research Part B: Methodological* 34(2):107–121.

[11] Ropke S, Cordeau JF (2009) Branch and cut and price for the pickup and delivery problem with time windows. *Transportation Science* 43(3):267–286.

[12] Ropke S, Pisinger D (2006) An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation science* 40(4):455–472.

[13] Savelsbergh MW, Sol M (1995) The general pickup and delivery problem. *Transportation Science* 29(1):17–29.

[14] Van der Bruggen L, Lenstra JK, Schuur P (1993) Variable-depth search for the single-vehicle pickup and delivery problem with time windows. *Transportation Science* 27(3):298–311.

[15] Wang C, Mu D, Zhao F, Sutherland JW (2015) A parallel simulated annealing method for the vehicle routing problem with simultaneous pickup–delivery and time windows. *Computers & Industrial Engineering* 83:111–122.

2019 International Joint Conference on Artificial Intelligence

Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence (IJCAI-19)

Earlier Attention? Aspect-Aware LSTM for Aspect-Based Sentiment Analysis

Bowen Xing[1*], **Lejian Liao**[1], **Dandan Song**[1 †], **Jingang Wang**[2], **Fuzhen Zhang**[2], **Zhongyuan Wang**[2] and **Heyan Huang**[1]
[1]Lab of High Volume language Information Processing & Cloud Computing
Beijing Lab of Intelligent Information Technology
School of Computer Science & Technology, Beijing Institute of Technology
[2]Meituan-Dianping Group
{xingbowen,liaolj,sdd,hhy63}@bit.edu.cn,
{wangjingang02,zhangfuzheng,wangzhongyuan02}@meituan.com

Abstract

Aspect-based sentiment analysis (ABSA) aims to predict fine-grained sentiments of comments with respect to given aspect terms or categories. In previous ABSA methods, the importance of aspect has been realized and verified. Most existing LSTM-based models take aspect into account via the attention mechanism, where the attention weights are calculated after the context is modeled in the form of contextual vectors. However, aspect-related information may be already discarded and aspect-irrelevant information may be retained in classic LSTM cells in the context modeling process, which can be improved to generate more effective context representations. This paper proposes a novel variant of LSTM, termed as aspect-aware LSTM (AA-LSTM), which incorporates aspect information into LSTM cells in the context modeling stage before the attention mechanism. Therefore, our AA-LSTM can dynamically produce aspect-aware contextual representations. We experiment with several representative LSTM-based models by replacing the classic LSTM cells with the AA-LSTM cells. Experimental results on SemEval-2014 Datasets demonstrate the effectiveness of AA-LSTM.

1 Introduction

With increasing numbers of comments on the Internet, sentiment analysis is attracting interests from both research and industry. Aspect-based sentiment analysis is a fundamental and challenging task in sentiment analysis, which aims to infer the sentiment polarity of sentences with respect to given aspects. For example, *"Great salad but the soup tastes bad"*. It's obvious that the opinion over the *'salad'* is positive while the opinion over the *'soup'* is negative. In this case, aspects are included in the comments, and predicting aspect sentiment polarities of this kind of comments is termed aspect term sentiment analysis (ATSA) or target sentiment analysis (TSA). There is another case where the aspect is not explicitly included in the comment. For example, *"Although the dinner is expensive, waiters are so warm-hearted!"*. We can observe that there are two aspect categories mentioned in this comment: price and service with completely opposite sentiment polarities. Predicting aspect sentiment polarities of this kind of comments is termed aspect category sentiment analysis (ACSA), and the aspect categories usually belong to a predefined set. In this paper, we collectively refer aspect category, aspect term/target as aspect. And our goal is aspect-based sentiment analysis (ABSA) including ATSA and ACSA.

As deep learning have been successfully exploited in various NLP tasks [Zhen *et al.*, 2017; Yang and Mitchell, 2017; Xu *et al.*, 2017; Devamanyu *et al.*, 2018], many neural networks have been applied to ABSA. With the ability of handling long-term dependencies, Long Short-Term Memory neural network (LSTM) [Hochreiter and Schmidhuber, 1997] is widely used for context modeling in ABSA, and many recent best performing ABSA methods are based on LSTM because of its significant performance [Tang *et al.*, 2016a; Wang *et al.*, 2016; Chen *et al.*, 2017; Ma *et al.*, 2017; Devamanyu *et al.*, 2018; Xin *et al.*, 2018]. Current mainstream LSTM-based ABSA models adopt LSTM to model the context, obtaining hidden state vectors for each token in the input sequence. After obtaining contextual vector representations, they utilize attention mechanism to produce the attention weight vector.

Recent well performing LSTM-based ABSA models can be divided into three categories according to their way of modeling context: (1) "Attention-based LSTM with aspect embedding (ATAE-LSTM)" [Wang *et al.*, 2016] and "modeling inter-aspect dependencies by LSTM (IAD-LSTM)" [Devamanyu *et al.*, 2018] model the context and aspect together via concatenating the aspect vector to the word embeddings of context words in the embedding layer. (2) "Interactive attention networks (IAN)" [Ma *et al.*, 2017] and "aspect fusion LSTM (AF-LSTM)" [Tay *et al.*, 2018] model the context alone and utilize the aspect to compute context's attention vector in the attention mechanism. (3) "Recurrent attention network on memory (RAM)" [Chen *et al.*, 2017] introduces relative position information of context words and the given target into their hidden state vectors.

[*]This work was partially done during Bowen's internship at Meituan-Dianping Group.
[†]Corresponding author.

Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence (IJCAI-19)

The first category conducts simple joint modeling of contexts and aspects. In the secondary category, on behalf of most of the existing LSTM-based methods, the models only use context words as input when modeling the context, so they get the same context hidden states vectors when analysing comments containing multiple aspects. The second category models the context separately while utilizing the aspect information in context's attention calculation. And the method in the third category additionally multiplies a relative position weight. However, no aspect information is considered in the LSTM cells of all the above methods.Therefore, after context modeling, their hidden state vectors contain the information that is important to the "overall" comment semantics. This is determined by the functionality of classic LSTM. It retains important information and filters out useless information at the sentence-level semantic, in the hidden states corresponding to every context word.

In contrast, for the aspect-based sentiment analysis task, we think the context modeling should be aspect-aware. For a specific aspect, on one hand, some of the semantic information of the whole sentence is useless. These aspect irrelevant information would adversely harm the final sentiment representation, especially in the situation where multiple aspects exist in one comment. This is because when LSTM encounters an important token for the overall sentence semantics, this token's information is retained in every follow-up hidden state. Consequently, even if a good attention vector is produced via the attention mechanism, these hidden state vectors also contain useless information which is magnified to some extent. On the other hand, information that is important to the aspect may be not sufficiently kept in hidden states because of their small contribution to the overall semantic information of the sentence.

We take two typical examples to illustrate the two issues. First, *"The salad is so delicious but the soup is unsatisfied."*. There are two aspects (*salad* and *soup*) of opposite sentiment polarity. When judging the sentiment polarity of the *'soup'*, the word 'delicious' which modifies *'salad'* is also important to the sentence-level semantics of the whole comment, and its information is preserved in the hidden states vectors of subsequent context words, including 'unsatisfied'. So even if 'unsatisfied' is assigned a large weight in the attention vector, the information of 'delicious' will still be integrated into the final context representation and enlarged. Second, *"Pizza is wonderful compared to the last time we enjoyed at another place, and the beef is not bad, by the way."* Obviously, this sentence is mainly about pizza so classic LSTM will retain a lot of information that modifies *'pizza'* when modeling context. But when judging the polarity of beef, because traditional LSTM does not know the aspect is *'beef'*, much-retained *'pizza'* information causes that the information of *'beef'* is not valued enough in hidden state vectors. We define the above issues as the aspect-unaware problem in the context modeling process of current methods. To the best of our knowledge, this is the first time to propose this problem.

In this paper, we propose a novel LSTM variant termed aspect-aware LSTM (AA-LSTM) to introduce the aspect into context modeling process. In every time step, on one hand, the aspect vector can select key information in the context according to the aspect and keep the important information in context words' hidden states. On the other hand, the vector formed aspect information can influent the process of context modeling and filter useless information for the given aspect. So AA-LSTM can generate more effective context hidden states based on the given aspect. This can be seen as an earlier attention operation on context words. It is worth mentioning that though our AA-LSTM model takes the aspect as input, it does not actually fuse the aspect vector into the representation of the context, but only utilize the aspect to influence the process of modeling context via controlling information flow.

The main contributions of our work can be summarized as follows:

- We propose a novel LSTM variant termed as aspect-aware LSTM (AA-LSTM) to introduce the aspect into the process of modeling context.
- Considering that the aspect is the core information in this task, we fully exploit its potential by introducing it into the LSTM cells. We design three aspect gates to introduce the aspect into the input gate, forget gate and output gate in classic LSTM. AA-LSTM can utilize aspect to improve the information flow and then generate more effective aspect-specific context representation.
- We apply our proposed AA-LSTM to several representative LSTM-based models, and the experimental results on the benchmark datasets demonstrate the validity and generalization of our proposed AA-LSTM.

2 Related Work

In this section, we survey some representative studies in the aspect-based sentiment analysis (ABSA). ABSA is the task of predicting the sentiment polarity of a comment with respect to a set of aspects terms or categories included in the context. The biggest challenge faced by ABSA is how to effectively represent the aspect-specific sentiment information of the comment [Ma *et al.*, 2018]. Although some traditional methods for target sentiment analysis also achieve promising results, they are labor intensive because they have mostly focused on feature engineering or massive extra linguistic resources [Kiritchenko *et al.*, 2014; Wagner *et al.*, 2014].

As deep learning achieved breakthrough success in representation learning, many recent works utilized deep neural networks to automatically extract features and generate the context embedding which is a dense vector formed representation of the comment.

Since the attention mechanism was first introduced to the NLP field [Bahdanau *et al.*, 2014], many sequence-based approaches utilize it to generate more aspect-specific final representations. Attention mechanism in ABSA takes aspect information (usually aspect embedding) and the hidden states of every context word (generated by context modeling) as input and produces a probability distribution in which important parts of the context will be assigned bigger weights according to the aspect information.

There are some CNN-based [Xue and Li, 2018] and memory networks (MNs)-based models for context modeling [Tang *et al.*, 2016b; Tay *et al.*, 2017; Wang *et al.*, 2018]. [Tay *et*

al., 2017] model dyadic interactions between aspect and sentence using neural tensor layers and associative layers with rich compositional operators. [Wang *et al.*, 2018] argue that for the case where several sentences are the same except for different targets, relying attention mechanism alone is insufficient. It designed several memory networks having their own characters to solve the problem.

In particular, LSTM networks are widely used in context modeling because of its advantages for sequence modeling [Tang *et al.*, 2016a; Ma *et al.*, 2017; Devamanyu *et al.*, 2018; Wang *et al.*, 2016; Tay *et al.*, 2018; Ma *et al.*, 2018; Liu and Zhang, 2017; Yang *et al.*, 2017]. We divide recent well-performing methods into three categories according to the process of modeling context: First, modeling the context and aspect via concatenating the aspect vector to the word embeddings of context words in the embedding layer. [Wang *et al.*, 2016] firstly propose aspect embedding, and their ATAE-LSTM learns to attend to different parts of the context according to the aspect embedding. Although IAD-LSTM [Devamanyu *et al.*, 2018] model inter-dependencies between multiple aspects of one comment through LSTM after getting the final representation of the context, it is consistent with ATAE-LSTM [Wang *et al.*, 2016] in the way of context modeling.

Second, modeling the context alone and utilizing the aspect to compute context's attention vector in the attention mechanism. The main difference among this category of models is the calculation method of the attention mechanism. [Ma *et al.*, 2017] propose an interactive attention network (IAN) which models targets and contexts separately. Then it learns the interactions between the context and target in attention mechanism utilizing the averages of context's hidden states and target's hidden states. [Tay *et al.*, 2018] propose Aspect Fusion LSTM (AF-LSTM) model with a novel association layer after LSTM to model word-aspect relation utilizing circular convolution and circular correlation.

Third, introducing relative position information of the given target and context words to the hidden state vectors of context words. RAM [Chen *et al.*, 2017] realizes that the hidden state vector of a word will be assigned a larger weight if it is closer to the target through a relative location vector. This operation is conducted before their recurrent attention layer consisting of GRU cells.

Unlike all the above methods, we propose to introduce he aspect information into the process of context modeling. Our proposed AA-LSTM introduces the aspect into the LSTM cells to control information flow. AA-LSTM can not only select key information in the context according to the aspect and keep the important information in context words' hidden state vectors, but also filter useless information for the given aspect. Then AA-LSTM can generate more effective aspect-specific context hidden state vectors.

3 Aspect-Aware LSTM

In this section we describe our proposed aspect-aware LSTM (AA-LSTM) in detail. Classic LSTM contains three gates (input gate, forget gate and output gate) to control the information flow. We argue that aspect information should be considered into LSTM cells to improve the information flow. It is intuitive that in every time step the degree that aspect is integrated into the three gates of classic LSTM should be different. Therefore, we incorporate aspect vector into classic LSTM cells and design three aspect gates to control how much the aspect vector is imported into the input gate, forget gate and output gate respectively. In this way, we can utilize the previous hidden state and the aspect itself to control how much the aspect is imported in the three gates of classic LSTM. Figure 1 illustrates the architecture of the AA-LSTM network and it can be formalized as follows:

$$a_i = \sigma(W_{ai}\,[A, h_{t-1}] + b_{ai}) \tag{1}$$
$$I_t = \sigma(W_I\,[x_t, h_{t-1}] + a_i \odot A + b_I) \tag{2}$$
$$a_f = \sigma(W_{af}\,[A, h_{t-1}] + b_{af}) \tag{3}$$
$$f_t = \sigma(W_f\,[x_t, h_{t-1}] + a_f \odot A + b_f) \tag{4}$$
$$\widetilde{C_t} = tanh(W_C\,[x_i, h_{t-1}] + b_C) \tag{5}$$
$$C_t = f_t \odot C_{t-1} + I_t \odot \widetilde{C_t} \tag{6}$$
$$a_o = \sigma(W_{ao}\,[A, h_{t-1}] + b_{ao}) \tag{7}$$
$$o_t = \sigma(W_o\,[x_t, h_{t-1}] + a_o \odot A + b_o) \tag{8}$$
$$h_t = o_t * tanh(C_t) \tag{9}$$

where x_t represents the input embedding vector of the context word corresponding to time step t, A stands for the aspect vector, h_{t-1} is previous hidden state, h_t is the hidden state of this time step, σ and $tanh$ are sigmoid and hyperbolic tangent functions, $\odot$ stands for element-wise multiplication, W_{ai}, W_{af}, $W_{ao} \in R^{da \times (dc+da)}$ and W_I, W_f, W_C, $W_o \in R^{dc \times 2dc}$ are the weighted matrices, $b_{ai}, b_{af}, b_{ao} \in R^{da}, b_I, b_f, b_C, b_o \in R^{dc}$ are biases and da, dc stand for the aspect vector's dimension and the number of hidden cells at AA-LSTM respectively. $I_t, f_t, o_t \in R^{dc}$ stand for the input gate, forget gate and output gate respectively. The input gate controls the extent of updating the information from the current input. The forget gate is responsible for selecting some information from last cell state. The output gate controls how much the information in current cell state is output to be the hidden state vector of this time step. Similarly, $a_i, a_f, a_o \in R^{da}$ stand for the aspect-input gate, aspect-forget gate and aspect-output gate respectively. The three aspect-based gates determine the extent of integrating the aspect information into the input gate, forget gate and output gate.

Our proposed AA-LSTM takes two strands of inputs: context word embeddings and the aspect vector. At each time step, the context word entering the AA-LSTM dynamically varies according to the sequence of words in the sentence, while the aspect vector is identical. Specifically, aspect vector is the representation of the target in TSA, and it is the aspect embedding in ACSA. Next, we describe the different components of our proposed AA-LSTM in detail.

3.1 Input Gates

The input gate I_t controls how much new information can be transferred to the cell state. While the aspect-input gate a_i controls how much the aspect is transferred to the input gate I_t. The difference between the AA-LSTM and the classical LSTM lies in the weighted aspect vector input of I_t. The

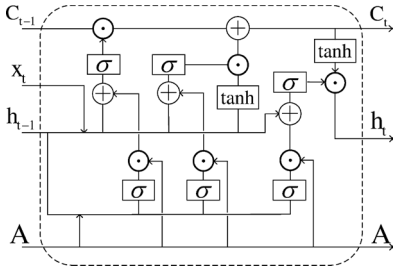


Figure 1: The overall architecture of AA-LSTM network

weight of aspect vector a_i is computed by h_{t-1} and A. h_{t-1} can be regarded as the previous semantic representation of the partial sentences which has been processed in the past time steps. Hence, the extent of the aspect's integration into I_t is decided by the previous semantic representation and the aspect vector A.

3.2 Forget Gates

The forget gate f_t abandons trivial information and retains key information from last cell state C_{t-1}. Similarly, the aspect-input gate a_f controls how much the aspect vector is transferred to the forget gate f_t. The difference between the AA-LSTM and the classical LSTM in f_t is the introduction of weighted aspect vector. And the weight of aspect vector a_f is computed by h_{t-1} and A. Therefore, the extent of the aspect's integration into I_t is decided by the previous semantic representation and the aspect vector A.

3.3 Candidate Cell and Current Cell

The candidate cell $\widetilde{C_t}$ represents the alternative input content. The current cell C_t updates its cell state by selecting important information from last cell state C_{t-1} and $\widetilde{C_t}$.

From Equation 6 we can observe that the alternative input content $\widetilde{C_t}$ aspects two strands of inputs: the last hidden state h_{t-1} and the input embedding x_t of this time step. So the hidden states in each time step contain the semantic information of the previous sentence segment. In the classical LSTM, information that is more important to the overall sentence semantics is more likely to be preserved in the hidden states vectors of the subsequent time steps. However, for some information which is retained because of its contribution to the semantics of the whole sentence, it may be noisy when judging the sentiment polarity of a given aspect. And the information that is crucial to analyze the given aspect's sentiment may be neglected due to its less contribution to the overall sentence. As demonstrated in the Introduction section, we define this phenomenon as an aspect-unaware problem in the process of context modeling.

Our proposed AA-LSTM can solve this problem by introducing aspect to the process of modeling context to control the flow of information. Information that is important for predicting the given aspect's sentiment polarity can be preserved in the hidden states vectors. In addition, as shown in Equation 6, the alternative input content does not include the aspect information. So the AA-LSTM only utilizes the given aspect to influent the information flow instead of integrating the aspect information into the hidden state vectors.

3.4 Output Gates

The output gate o_t controls the extent of the information flow from the current cell state to the hidden state vector of this time step. Similarly, the aspect-output gate a_o controls the extent of the aspect's influence on the output gate I_t. The difference between our proposed AA-LSTM and the classical LSTM in o_t is the introduction of weighted aspect vector into o_t. And the weight of aspect vector a_o is computed by h_{t-1} and A. Therefore, the degree of how much the aspect information is integrated into o_t is decided by the previous semantic representation and the aspect vector A.

4 Experiment

In this section, we introduce the tasks, the datasets, the evaluation metric, the models for comparison and the implementation details.

4.1 Tasks Definition

We conduct experiments on two subtasks of aspect sentiment analysis: aspect term sentiment analysis (ATSA) or target sentiment analysis (TSA) and aspect category sentiment analysis (ACSA). The former infers sentiment polarities of given target entities contained in the context. The latter infers sentiment polarities of generic aspects such as *'service'* or *'food'* which may or may not be found in the context, and the aspects belong to a predefined set. In this paper, these two kinds of tasks are both considered and they are collectively named as aspect-based sentiment analysis (ABSA).

4.2 Datasets

We experiment on SemEval 2014 [Pontiki *et al.*, 2014] task 4 datasets which consist of *laptop* and *restaurant* reviews and are widely used benchmarks in many previous works [Tang *et al.*, 2016a; Wang *et al.*, 2016; Ma *et al.*, 2017; Chen *et al.*, 2017; Devamanyu *et al.*, 2018; Tay *et al.*, 2018; Wang *et al.*, 2018]. We remove the reviews having no aspect or the aspects with sentiment polarity of "conflict". The dataset we used consists of reviews with at least one aspect labeled with sentiment polarities of *positive, neutral* and *negative*. For ATSA, we adopt Laptop and Restaurant datasets; And for ACSA, we adopt the Restaurant dataset. 20% of the training data is used as the development set. Full statistics of SemEval 2014 task 4 datasets are given in Table 1.

4.3 Evaluation Metric

Since the two tasks are both multi-class classification tasks, we adopt F1-Macro as our evaluation measure. And there are some other methods that use strict accuracy (Acc) [Wang *et al.*, 2016; Ma *et al.*, 2017; Chen *et al.*, 2017; Devamanyu *et al.*, 2018] for evaluation, which measures the percentage of correctly predicted samples in all samples. Therefore, we

Task	Dataset	Pos	Neg	Neu
ATSA	Restaurant Train	2164	807	637
	Restaurant Test	728	196	196
	Laptop Train	994	870	464
	Laptop Test	341	128	169
ACSA	Restaurant Train	2179	839	500
	Restaurant Test	657	222	94

Table 1: Statistic of all datasets

use these two metrics (F1-Macro and Acc) to evaluate the models' performances.

Generally, higher Acc can verify the effectiveness of the system though it biases towards the majority class, and F1-Macro provides more indicative information because the task is a multi-class problem.

4.4 Models for Comparison

In order to verify the advantages of our proposed AA-LSTM compared to classic LSTM, we choose some representative LSTM-based models to replace their original LSTM with our proposed AA-LSTM.

In Introduction and Related Work sections, we have divide recent well-performing methods into three categories according to their processes of modeling context. In order to prove the generalization ability of our model, we select a representative model from each of these categories for experiments. We choose ATAE-LSTM, IAN, and RAM as the representatives of the three categories of models because their architectures are novel and they are taken as comparative methods in many works. We also compare our model with the baseline LSTM model. We introduce them in detail as follows:

LSTM. This is the baseline that ignores targets and only models contexts using one LSTM network. The last hidden state is regarded as the final sentiment representation.

ATAE-LSTM. It concatenates the aspect embedding to the word embeddings of context words and uses aspect embedding to produce the attention vector. For ATSA, we take the average of the embeddings of the target words as the aspect embedding which is concatenated to the word embeddings of the context words.

IAN. It models context and target separately and selects important information from them via two interactive attention mechanisms. The target and context can have impacts on the generation of their representations interactively and their representations are concatenated as the final aspect-specific sentiment representations. For the ACSA task, we omit the modeling of the target and use the aspect embedding to produce the attention vector of context words.

RAM. It utilizes relative location to assign weights to original context hidden state vectors and then learns the attention vector in a recurrent attention mechanism consisting of GRU cell. It can only be applied to ATSA. For the consistent of comparison, we replaced the deep bidirectional LSTM in the original RAM with a unidirectional single-layer LSTM.

We also choose two state-of-the-art methods that are Memory Networks-based and LSTM-based respectively:

Target-sensitive Memory Network. [Wang *et al.*, 2018] construct six target-sensitive memory networks (TMNs) which have their own characteristics to resolve target sensitivity and got some improvement. We choose the NP (hops) and JCI (hops) that perform best on Laptop and Restaurant, respectively.

Inter-Aspect Dependencies LSTM. [Devamanyu *et al.*, 2018] model aspect-based sentential representations as a sequence to capture the inter-aspect dependencies.

We don't reimplement the above two models and the results are retrieved from their original papers.

4.5 Implementation Details

We implement the models in Tensorflow. We initialize all word embeddings by Glove [Jeffrey *et al.*, 2014] and out-of-vocabulary words by sampling from the uniform distribution $U(-0.1, 0.1)$. Initial values of all weight matrices are sampled from uniform distribution $U(-0.1, 0.1)$ and initial values of all biases are zeros. All embedding dimensions are set to 300 and the batch size is set as 16. We minimize the loss function to train our models using Adam optimizer [Diederik and Jimmy, 2014] with the learning rate set as 0.001. To avoid over fitting, we adopt the dropout strategy with $p = 0.5$ and the coefficient of *L2* normalization in the loss function is set to 0.01. All models use softmax classifier.

For ACSA, we initialize all aspect embeddings by sampling from the uniform distribution $U(-0.1, 0.1)$. As for the input aspect vector (A) of our proposed AA-LSTM which is replaced with the classic LSTM in the above models, we set it as follows:

Aspect Term Sentiment Analysis. We use the average of word embeddings of the target words as A except for IAN. For IAN, we use the average of the hidden states vectors of target words as A.

Aspect Category Sentiment Analysis. For all models, we use the aspect embedding as A.

We implement all models under the same experiment settings to make sure the improvements based on the original models come from the replacement of classic LSTM with our proposed AA-LSTM.

5 Results and Analysis

Our experimental results are illustrated in Table 2. We can observe that our proposed AA-LSTM and its substitution in other models has an overall advantage over classic LSTMs on their corresponding original models. It especially achieves higher F1-Macro which can better illustrate the overall performances of the models in multiple classes as the classes are unbalanced. On the ATSA task, except for the F1-Macro score on Restaurant, the performances of our variants overpass the performances of the representative state-of-the-art models. In the implementation of the experiment, the only difference between original models and their variants is the substitution of classic LSTM. As we replace the original LSTM with our AA-LSTM, the performance improvement can demonstrate the pure effectiveness of our AA-LSTM.

Task and Dataset	ATSA				ACSA	
	Laptop		Restaurant		Restaurant	
Model	F1-Macro	Acc	F1-Macro	Acc	F1-Macro	Acc
LSTM	59.77	65.99	61.04	75.00	70.07	81.71
ATAE-LSTM	61.28	66.93	64.47	77.41	70.15	82.12
IAN	64.54	70.53	65.67	78.48	70.81	83.25
RAM	67.05	71.32	65.84	78.57	-	-
IAD-LSTM	-	72.5	-	79.0	-	-
JCI (hops)	67.2	71.8	**68.8**	78.8	-	-
NP (hops)	67.8	72.4	66.0	75.7	-	-
AA-LSTM	61.45	66.93	66.24	78.21	**75.00**	83.45
ATAE-LSTM (AA)	62.10	69.28	66.46	78.21	74.51	83.97
IAN (AA)	65.62	71.94	68.71	**79.29**	74.43	**84.69**
RAM (AA)	**68.47**	**73.20**	68.15	78.13	-	-

Table 2: Comparisons of all models on three datasets. Last four models are our proposed AA-LSTM models, and the last three models with suffix "(AA)" is the variants in which the original classic LSTM is replaced with our proposed AA-LSTM. The results of IAD-LSTM, JCI (hops) and NP (hops) are retrieved from the original papers. Best scores are marked in **bold**.

Compared with LSTM, AA-LSTM's improvements on macro are up to 7% and 6% on Restaurant for ATSA and ACSA respectively. Like LSTM, AA-LSTM also directly uses the last hidden state vector as the final sentiment representation sent to the classifier. But because the aspect is introduced into the process of modeling context, the semantics of the last hidden state vector of AA-LSTM is aspect-specific. In fact, not only in the last hidden layer, but also in all hidden states vectors, the information which is important for determining the emotional polarity of the aspect is kept, and other useless information is filtered, which makes the context modeling result much better than LSTM.As the classifiers are the same, the reason AA-LSTM performs better than LSTM is that the final sentiment representation of AA-LSTM is more effective.

AA-LSTM's performance even surpassed ATAE-LSTM and exceed all original models on F1-macro for ACSA. It is worth mentioning that all baselines utilizes the attention mechanism and ATAE-LSTM also models the context and aspect together via concatenating aspect to every word embeddings of context words. In contrast, AA-LSTM only models the context without any other processing. This proves that AA-LSTM's result of modeling context is aspect-specific and effective. This is because the aspect information is used in the modeling process to control the flow of information, retain and filter information, who performs as earlier attention.

ATAE-LSTM (AA)'s performance exceeds ATAE-LSTM and AA-LSTM. This shows that AA-LSTM can be compatible with other components of ATAE-LSTM, improving the whole model's performance. ATAE-LSTM represents a category of models that combine the context and aspect together via concatenating the aspect vector to context word embeddings. So the experimental results verify that although the input embeddings contain aspect information, it doesn't conflict with the aspect information introduced in AA-LSTM.

IAN represents a category of models which encode the context alone and utilize the aspect to compute contexts' attention vector in the attention mechanism. IAN-LSTM (AA)'s overall performance exceeds IAN and AA-LSTM. This proves that the hidden states vectors generated by AA-LSTM can collaborate with the attention mechanism to achieve better performance.

RAM utilizes the relative location vector to assign weights to original context word hidden state vectors, and calculates the attention vector via a recurrent attention mechanism which is more complex than other baseline models. It is worth noting that compared with RAM, RAM (AA) has more improvement than other original models and their variants. This is because the advantage of AA-LSTM is amplified in RAM. In RAM (AA), while the tokens closer to the target are assigned larger weights, AA-LSTM keeps more important information about the target in the tokens closer to the target: adjectives, modifying phrases, clauses, etc. In addition, the context hidden states vectors generated by AA-LSTM and the recurrent mechanism work together to produce more effective final sentiment representation.

6 Conclusion

In this paper, we argue that aspect-related information may be discarded and aspect-irrelevant information may be retained in classic LSTM cells. To address this problem, we propose a novel LSTM variant termed as Aspect-Aware LSTM. Due to the introduction of the aspect into the process of modeling context, our proposed Aspect-aware LSTM can select important information about the given target and filter out the useless information via information flow control. Aspect-Aware LSTM can not only generate more effective contextual vectors than classic LSTM, but also be compatible with other modules.

Acknowledgements

The authors would like to thank the anonymous reviewers for their valuable comments. This work is supported by National Key Research and Development Program of China (Grant No. 2016YFB1000902), National Natural Science Foundation of China (NSFC, Grant Nos. 61866038 and 61751201), and Research Foundation of Beijing Municipal Science and Technology Commissions (No. Z181100008918002).

References

[Bahdanau *et al.*, 2014] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *Computer Science*, 2014.

[Chen *et al.*, 2017] Peng Chen, Zhongqian Sun, Lidong Bing, and Wei Yang. Recurrent attention network on memory for aspect sentiment analysis. In *Empirical Methods in Natural Language Processing*, pages 452–461, 2017.

[Devamanyu *et al.*, 2018] Hazarika Devamanyu, Poria Soujanya, Vij Prateek, Krishnamurthy Gangeshwar, Cambria Erik, and Zimmermann Roger. Modeling inter-aspect dependencies for aspect-based sentiment analysis. In *Conference of the North American Chapter of the Association for Computational Linguistics*, pages 266–270, 2018.

[Diederik and Jimmy, 2014] Kingma Diederik and Ba Jimmy. Adam: A method for stochastic optimization. *CoRR abs/1412.6980*, 2014.

[Hochreiter and Schmidhuber, 1997] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[Jeffrey *et al.*, 2014] Pennington Jeffrey, Socher Richard, and Manning Christopher. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing*, pages 1532–1543, 2014.

[Kiritchenko *et al.*, 2014] Svetlana Kiritchenko, Xiaodan Zhu, Colin Cherrt, and Saif Mohammad and. Nrc-canada-2014: Detecting aspects and sentiment in customer reviews. In *International Workshop on Semantic Evaluation*, pages 437–442, 2014.

[Liu and Zhang, 2017] Jiangming Liu and Yue Zhang. Attention modeling for targeted sentiment. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics*, page 572577, 2017.

[Ma *et al.*, 2017] Dehong Ma, Sujian Li, Xiaodong Zhang, and Houfeng Wang. Interactive attention networks for aspect-level sentiment classification. In *International Joint Conference on Artificial Intelligence*, pages 4068–4074, 2017.

[Ma *et al.*, 2018] Yukun Ma, Haiyun Peng, and Erik Cambria. Targeted aspect-based sentiment analysis via embedding commonsense knowledge into an attentive lstm. In *AAAI Conference on Artificial Intelligence*, pages 5876–5883, 2018.

[Pontiki *et al.*, 2014] Maria Pontiki, Dimitris Galanis, John Pavlopoulos, Haris Papageorgiou, Ion Androutsopoulos, and Suresh Manandhar. Semeval-2014 task 4: Aspect based sentiment analysis. In *International Workshop on Semantic Evaluation*, pages 27–35, 2014.

[Tang *et al.*, 2016a] Duyu Tang, Bing Qin, Xiaocheng Feng, and Ting Liu. Effective lstms for target-dependent sentiment classification. In *International Conference on Computational Linguistics*, pages 3298–3307, 2016.

[Tang *et al.*, 2016b] Duyu Tang, Bing Qin, and Ting Liu. Aspect level sentiment classification with deep memory network. In *Empirical Methods in Natural Language Processing*, pages 214–224, 2016.

[Tay *et al.*, 2017] Yi Tay, Tuan Luu Anh, and Hui Siu Cheung. Dyadic memory networks for aspect-based sentiment analysis. In *ACM on Conference on Information and Knowledge Management, CIKM*, pages 107–116, 2017.

[Tay *et al.*, 2018] Yi Tay, Luu Anh Tuan, and Siu Cheung Hui. Learning to attend via word-aspect associative fusion for aspect-based sentiment analysis. In *AAAI Conference on Artificial Intelligence*, pages 5956–5963, 2018.

[Wagner *et al.*, 2014] Joachim Wagner, Piyush Arora, Santiago Cortes, Utsab Barman, Dasha Bogdanova, Jennifer Foster, and Lamia Tounsi. Dcu: Aspect-based polarity classification for semeval task 4. In *International Workshop on Semantic Evaluation*, pages 223–229, 2014.

[Wang *et al.*, 2016] Yequan Wang, Minlie Huang, and Li Zhao. Attention-based lstm for aspect-level sentiment classification. In *Empirical Methods in Natural Language Processing*, pages 606–615, 2016.

[Wang *et al.*, 2018] Shuai Wang, Sahisnu Mazumder, Bing Liu, Mianwei Zhou, and Yi Chang. Target-sensitive memory networks for aspect sentiment classification. In *Annual Meeting of the Association for Computational Linguistics,*, pages 957–967, 2018.

[Xin *et al.*, 2018] Li Xin, Bing Lidong, Lam Wai, and Shi Bei. Transformation networks for target-oriented sentiment classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 946–956, 2018.

[Xu *et al.*, 2017] Jiacheng Xu, Xipeng Qiu, Kan Chen, and Xuanjing Huang. Knowledge graph representation with jointly structural and textual encoding. In *International Joint Conference on Artificial Intelligence*, pages 1318–1324, 2017.

[Xue and Li, 2018] Wei Xue and Tao Li. Aspect based sentiment analysis with gated convolutional networks. In *Annual Meeting of the Association for Computational Linguistics,*, pages 2514–2523, 2018.

[Yang and Mitchell, 2017] Bishan Yang and Tom M. Mitchell. Leveraging knowledge bases in lstms for improving machine reading. In *Annual Meeting of the Association for Computational Linguistics*, pages 1436–1446, 2017.

[Yang *et al.*, 2017] Min Yang, Wenting Tu, Jingxuan Wang, Fei Xu, and Xiaojun Chen. Attention based lstm for target dependent sentiment classification. In *AAAI Conference on Artificial Intelligence*, page 50135014, 2017.

[Zhen *et al.*, 2017] Xu Zhen, Liu Bingquan, Wang Baoxun, Sun Chengjie, and Wang Xiaolong. Incorporating loose-structured knowledge into conversation modeling via recall-gate lstm. In *International Joint Conference on Neural Networks*, pages 3506–3513, 2017.

The 25th ACM SIGKDD Conference on Knowledge Discovery and Data Mining

Research Track Paper KDD '19, August 4–8, 2019, Anchorage, AK, USA

Knowledge-aware Graph Neural Networks with Label Smoothness Regularization for Recommender Systems

Hongwei Wang
Stanford University
hongweiw@cs.stanford.edu

Fuzheng Zhang
Meituan-Dianping Group
zhangfuzheng@meituan.com

Mengdi Zhang
Meituan-Dianping Group
zhangmengdi02@meituan.com

Jure Leskovec
Stanford University
jure@cs.stanford.edu

Miao Zhao, Wenjie Li
Hong Kong Polytechnic University
{csmiaozhao,cswjli}@comp.polyu.edu.hk

Zhongyuan Wang
Meituan-Dianping Group
wangzhongyuan02@meituan.com

ABSTRACT

Knowledge graphs capture structured information and relations between a set of entities or items. As such knowledge graphs represent an attractive source of information that could help improve recommender systems. However, existing approaches in this domain rely on manual feature engineering and do not allow for an end-to-end training. Here we propose *Knowledge-aware Graph Neural Networks with Label Smoothness regularization* (KGNN-LS) to provide better recommendations. Conceptually, our approach computes user-specific item embeddings by first applying a trainable function that identifies important knowledge graph relationships for a given user. This way we transform the knowledge graph into a user-specific weighted graph and then apply a graph neural network to compute personalized item embeddings. To provide better inductive bias, we rely on *label smoothness* assumption, which posits that adjacent items in the knowledge graph are likely to have similar user relevance labels/scores. Label smoothness provides regularization over the edge weights and we prove that it is equivalent to a label propagation scheme on a graph. We also develop an efficient implementation that shows strong scalability with respect to the knowledge graph size. Experiments on four datasets show that our method outperforms state of the art baselines. KGNN-LS also achieves strong performance in cold-start scenarios where user-item interactions are sparse.

KEYWORDS

Knowledge-aware recommendation; graph neural networks; label propagation

ACM Reference Format:
Hongwei Wang, Fuzheng Zhang, Mengdi Zhang, Jure Leskovec, Miao Zhao, Wenjie Li, and Zhongyuan Wang. 2019. Knowledge-aware Graph Neural Networks with Label Smoothness Regularization for Recommender Systems. In *The 25th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '19), August 4–8, 2019, Anchorage, AK, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3292500.3330836

KDD '19, August 4–8, 2019, Anchorage, AK, USA

ACM ISBN 978-1-4503-6201-6/19/08…$15.00
https://doi.org/10.1145/3292500.3330836

1 INTRODUCTION

Recommender systems are widely used in Internet applications to meet user's personalized interests and alleviate the information overload [4, 29, 32]. Traditional recommender systems that are based on collaborative filtering [13, 22] usually suffer from the cold-start problem and have trouble recommending brand new items that have not yet been heavily explored by the users. The sparsity issue can be addressed by introducing additional sources of information such as user/item profiles [23] or social networks [22].

Knowledge graphs (KGs) capture structured information and relations between a set of entities [8, 9, 18, 24–28, 33, 34, 36]. KGs are heterogeneous graphs in which nodes correspond to *entities* (e.g., items or products, as well as their properties and characteristics) and edges correspond to *relations*. KGs provide connectivity information between items via different types of relations and thus capture semantic relatedness between the items.

The core challenge in utilizing KGs in recommender systems is to learn how to capture *user-specific* item-item relatedness captured by the KG. Existing KG-aware recommender systems can be classified into path-based methods [8, 33, 36], embedding-based methods [9, 26, 27, 34], and hybrid methods [18, 24, 28]. However, these approaches rely on manual feature engineering, are unable to perform end-to-end training, and have poor scalability. Graph Neural Networks (GNNs), which aggregate node feature information from node's local network neighborhood using neural networks, represent a promising advancement in graph-based representation learning [3, 5–7, 11, 15]. Recently, several works developed GNNs architecture for recommender systems [14, 19, 28, 31, 32], but these approaches are mostly designed for *homogeneous* bipartite user-item interaction graphs or user-/item-similarity graphs. It remains an open question how to extend GNNs architecture to *heterogeneous* knowledge graphs.

In this paper, we develop *Knowledge-aware Graph Neural Networks with Label Smoothness regularization* (KGNN-LS) that extends GNNs architecture to knowledge graphs to simultaneously capture semantic relationships between the items as well as personalized user preferences and interests. To account for the relational heterogeneity in KGs, similar to [28], we use a trainable and personalized relation scoring function that transforms the KG into a user-specific weighted graph, which characterizes both the semantic information of the KG as well as user's personalized interests. For example, in the movie recommendation setting the relation scoring function could learn that a given user really cares about "director" relation between movies and persons, while somebody else may care more

about the "lead actor" relation. Using this personalized weighted graph, we then apply a graph neural network that for every item node computes its embedding by aggregating node feature information over the local network neighborhood of the item node. This way the embedding of each item captures it's local KG structure in a user-personalized way.

A significant difference between our approach and traditional GNNs is that the edge weights in the graph are not given as input. We set them using user-specific relation scoring function that is trained in a supervised fashion. However, the added flexibility of edge weights makes the learning process prone to overfitting, since the only source of supervised signal for the relation scoring function is coming from user-item interactions (which are sparse in general). To remedy this problem, we develop a technique for regularization of edge weights during the learning process, which leads to better generalization. We develop an approach based on *label smoothness* [35, 38], which assumes that adjacent entities in the KG are likely to have similar user relevancy labels/scores. In our context this assumption means that users tend to have similar preferences to items that are nearby in the KG. We prove that label smoothness regularization is equivalent to *label propagation* and we design a *leave-one-out* loss function for label propagation to provide extra supervised signal for learning the edge scoring function. We show that the knowledge-aware graph neural networks and label smoothness regularization can be unified under the same framework, where label smoothness can be seen as a natural choice of regularization on knowledge-aware graph neural networks.

We apply the proposed method to four real-world datasets of movie, book, music, and restaurant recommendations, in which the first three datasets are public datasets and the last is from Meituan-Dianping Group. Experiments show that our method achieves significant gains over state-of-the-art methods in recommendation accuracy. We also show that our method maintains strong recommendation performance in the cold-start scenarios where user-item interactions are sparse.

2 RELATED WORK

2.1 Graph Neural Networks

Graph Neural Networks (or Graph Convolutional Neural Networks, GCNs) aim to generalize convolutional neural networks to non-Euclidean domains (such as graphs) for robust feature learning. Bruna et al. [3] define the convolution in Fourier domain and calculate the eigendecomposition of the graph Laplacian, Defferrard et al. [5] approximate the convolutional filters by Chebyshev expansion of the graph Laplacian, and Kipf et al. [11] propose a convolutional architecture via a first-order approximation. In contrast to these *spectral* GCNs, *non-spectral* GCNs operate on the graph directly and apply "convolution" (i.e., weighted average) to local neighbors of a node [6, 7, 15].

Recently, researchers also deployed GCNs in recommender systems: PinSage [32] applies GCNs to the pin-board bipartite graph in Pinterest. Monti et al. [14] and Berg et al. [19] model recommender systems as matrix completion and design GCNs for representation learning on user-item bipartite graphs. Wu et al. [31] use GCNs on user/item structure graphs to learn user/item representations. The difference between these works and ours is that they are all designed for homogeneous bipartite graphs or user/item-similarity graphs where GCNs can be used directly, while here we investigate GCNs for heterogeneous KGs. Wang et al. [28] use GCNs in KGs for recommendation, but simply applying GCNs to KGs without proper regularization is prone to overfitting and leads to performance degradation as we will show later. Schlichtkrull et al. also propose using GCNs to model KGs [17], but not for the purpose of recommendations.

2.2 Semi-supervised Learning on Graphs

The goal of graph-based semi-supervised learning is to correctly label all nodes in a graph given that only a few nodes are labeled. Prior work often makes assumptions on the distribution of labels over the graph, and one common assumption is smooth variation of labels of nodes across the graph. Based on different settings of edge weights in the input graph, these methods are classified as: (1) Edge weights are assumed to be given as input and therefore fixed [1, 37, 38]; (2) Edge weights are parameterized and therefore learnable [10, 21, 35]. Inspired by these methods, we design a module of label smoothness regularization in our proposed model. The major distinction of our work is that the label smoothness constraint is not used for semi-supervised learning on graphs, but serves as regularization to assist the learning of edge weights and achieves better generalization for recommender systems.

2.3 Recommendations with Knowledge Graphs

In general, existing KG-aware recommender systems can be classified into three categories: (1) *Embedding-based methods* [9, 26, 27, 34] pre-process a KG with *knowledge graph embedding* (KGE) [30] algorithms, then incorporate learned entity embeddings into recommendation. Embedding-based methods are highly flexible in utilizing KGs to assist recommender systems, but the KGE algorithms focus more on modeling rigorous semantic relatedness (e.g., TransE [2] assumes $head + relation = tail$), which are more suitable for graph applications such as link prediction rather than recommendations. In addition, embedding-based methods usually lack an end-to-end way of training. (2) *Path-based methods* [8, 33, 36] explore various patterns of connections among items in a KG (a.k.a meta-path or meta-graph) to provide additional guidance for recommendations. Path-based methods make use of KGs in a more intuitive way, but they rely heavily on manually designed meta-paths/meta-graphs, which are hard to tune in practice. (3) *Hybrid methods* [18, 24, 28] combine the above two categories and learn user/item embeddings by exploiting the structure of KGs. Our proposed model can be seen as an instance of hybrid methods.

3 PROBLEM FORMULATION

We begin by describing the KG-aware recommendations problem and introducing notation. In a typical recommendation scenario, we have a set of users $\mathcal{U}$ and a set of items $\mathcal{V}$. The user-item interaction matrix $\mathbf{Y}$ is defined according to users' implicit feedback, where $y_{uv} = 1$ indicates that user u has engaged with item v, such as clicking, watching, or purchasing. We also have a knowledge graph $\mathcal{G} = \{(h, r, t)\}$ available, in which $h \in \mathcal{E}$, $r \in \mathcal{R}$, and $t \in \mathcal{E}$ denote the head, relation, and tail of a knowledge triple, $\mathcal{E}$ and $\mathcal{R}$ are the set of entities and relations in the knowledge graph, respectively. For

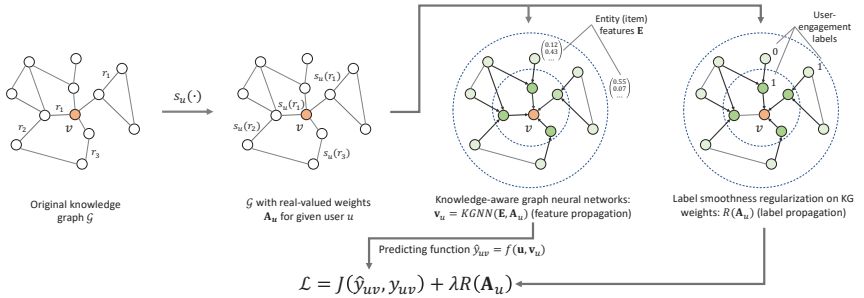


Figure 1: Overview of our proposed KGNN-LS model. The original KG is first transformed into a user-specific weighted graph, on which we then perform feature propagation using a graph neural network with the label smoothness regularization. The two modules constitute the complete loss function $\mathcal{L}$.

example, the triple (*The Silence of the Lambs*, *film.film.star*, *Anthony Hopkins*) states the fact that Anthony Hopkins is the leading actor in film "The Silence of the Lambs". In many recommendation scenarios, an item $v \in \mathcal{V}$ corresponds to an entity $e \in \mathcal{E}$ (e.g., item "The Silence of the Lambs" in MovieLens also appears in the knowledge graph as an entity). The set of entities $\mathcal{E}$ is composed from items $\mathcal{V}$ ($\mathcal{V} \subseteq \mathcal{E}$) as well as non-items $\mathcal{E} \backslash \mathcal{V}$ (e.g. nodes corresponding to item/product properties). Given user-item interaction matrix $\mathbf{Y}$ and knowledge graph $\mathcal{G}$, our task is to predict whether user u has potential interest in item v with which he/she has not engaged before. Specifically, we aim to learn a prediction function $\hat{y}_{uv} = \mathcal{F}(u, v|\Theta, \mathbf{Y}, \mathcal{G})$, where $\hat{y}_{uv}$ denotes the probability that user u will engage with item v, and Θ are model parameters of function $\mathcal{F}$.

We list the key symbols used in this paper in Table 1.

Symbol	Meaning
$\mathcal{U} = \{u_1, \cdots\}$	Set of users
$\mathcal{V} = \{v_1, \cdots\}$	Set of items
$\mathbf{Y}$	User-item interaction matrix
$\mathcal{G} = (\mathcal{E}, \mathcal{R})$	Knowledge graph
$\mathcal{E} = \{e_1, \cdots\}$	Set of entities
$\mathcal{R} = \{r_1, \cdots\}$	Set of relations
$\mathcal{E} \backslash \mathcal{V}$	Set of non-item entities
$s_u(r)$	User-specific relation scoring function
$\mathbf{A}_u$	Adjacency matrix of $\mathcal{G}$ w.r.t. user u
$\mathbf{D}_u$	Diagonal degree matrix of $\mathbf{A}_u$
$\mathbf{E}$	Raw entity feature
$\mathbf{H}^{(l)},\ l = 0, ..., L-1$	Entity representation in the l-th layer
$\mathbf{W}^{(l)},\ l = 0, ..., L-1$	Transformation matrix in the l-th layer
$l_u(e),\ e \in \mathcal{E}$	Item relevancy labeling function
$l_u^*(e),\ e \in \mathcal{E}$	Minimum-energy labeling function
$\hat{l}_u(v),\ v \in \mathcal{V}$	Predicted relevancy label for item v
$R(\mathbf{A}_u)$	Label smoothness regularization on $\mathbf{A}_u$

Table 1: List of key symbols.

4 OUR APPROACH

In this section, we first introduce knowledge-aware graph neural networks and label smoothness regularization, respectively, then we present the unified model.

4.1 Preliminaries: Knowledge-aware Graph Neural Networks

The first step of our approach is to transform a heterogeneous KG into a user-personalized weighted graph that characterizes user's preferences. To this end, similar to [28], we use a user-specific *relation scoring function* $s_u(r)$ that provides the importance of relation r for user u: $s_u(r) = g(\mathbf{u}, \mathbf{r})$, where $\mathbf{u}$ and $\mathbf{r}$ are feature vectors of user u and relation type r, respectively, and g is a differentiable function such as inner product. Intuitively, $s_u(r)$ characterizes the importance of relation r to user u. For example, a user may be more interested in *directors* of movies, but another user may care more about the *lead actors* of movies.

Given user-specific relation scoring function $s_u(\cdot)$ of user u, knowledge graph $\mathcal{G}$ can therefore be transformed into a user-specific adjacency matrix $\mathbf{A}_u \in \mathbb{R}^{|\mathcal{E}| \times |\mathcal{E}|}$, in which the (i, j)-entry $A_u^{ij} = s_u(r_{e_i,e_j})$, and r_{e_i,e_j} is the relation between entities e_i and e_j in $\mathcal{G}$.[1] $A_u^{ij} = 0$ if there is no relation between e_i and e_j. See the left two subfigures in Figure 1 for illustration. We also denote the raw feature matrix of entities as $\mathbf{E} \in \mathbb{R}^{|\mathcal{E}| \times d_0}$, where d_0 is the dimension of raw entity features. Then we use multiple feed forward layers[2] to update the entity representation matrix by aggregating representations of neighboring entities. Specifically, the layer-wise forward propagation can be expressed as

$$\mathbf{H}_{l+1} = \sigma\left(\mathbf{D}_u^{-1/2}\mathbf{A}_u\mathbf{D}_u^{-1/2}\mathbf{H}_l\mathbf{W}_l\right),\ l = 0, 1, \cdots, L-1. \quad (1)$$

[1] In this work we treat $\mathcal{G}$ an undirected graph, so $\mathbf{A}_u$ is a symmetric matrix. If both triples (h, r_1, t) and (t, r_2, h) exist, we only consider one of r_1 and r_2. This is due to the fact that: (1) r_1 and r_2 are the inverse of each other and semantically related; (2) Treating $\mathbf{A}_u$ symmetric will greatly increase the matrix density.

[2] There are several candidate designs for the architecture of our model, e.g., GCN [11] or GraphSAGE [7]. Here we use GCN [11] as our base model.

In Eq. (1), $\mathbf{H}_l$ is the matrix of hidden representations of entities in layer l, and $\mathbf{H}_0 = \mathbf{E}$. $\mathbf{A}_u$ is to aggregate representation vectors of neighboring entities. In this paper, we set $\mathbf{A}_u \leftarrow \mathbf{A}_u + \mathbf{I}$, i.e., adding self-connection to each entity, to ensure that old representation vector of the entity itself is taken into consideration when updating entity representations. $\mathbf{D}_u$ is a diagonal degree matrix with entries $D_u^{ii} = \sum_j A_u^{ij}$, therefore, $\mathbf{D}_u^{-1/2}$ is used to normalize $\mathbf{A}_u$ and keep the entity representation matrix $\mathbf{H}_l$ stable. $\mathbf{W}_l \in \mathbb{R}^{d_l \times d_{l+1}}$ is the layer-specific trainable weight matrix, σ is a non-linear activation function, and L is the number of layers.

A single GNN layer computes the representation of an entity via a transformed mixture of itself and its immediate neighbors in the KG. We can therefore naturally extend the model to multiple layers to explore users' potential interests in a broader and deeper way. The final output is $\mathbf{H}_L \in \mathbb{R}^{|\mathcal{E}| \times d_L}$, which is the entity representations that mix the initial features of themselves and their neighbors up to L hops away. Finally, the predicted engagement probability of user u with item v is calculated by $\hat{y}_{uv} = f(\mathbf{u}, \mathbf{v}_u)$, where $\mathbf{v}_u$ (i.e., the v-th row of $\mathbf{H}_L$) is the final representation vector of item v, and f is a differentiable prediction function, for example, inner product or a multilayer perceptron. Note that $\mathbf{v}_u$ is user-specific since the adjacency matrix $\mathbf{A}_u$ is user-specific. Furthermore, note that the system is end-to-end trainable where the gradients flow from $f(\cdot)$ via GNN (parameter matrix $\mathbf{W}$) to $g(\cdot)$ and eventually to representations of users u and items v.

4.2 Label Smoothness Regularization

It is worth noticing a significant difference between our model and GNNs: In traditional GNNs, edge weights of the input graph are fixed; but in our model, edge weights $\mathbf{D}_u^{-1/2}\mathbf{A}_u\mathbf{D}_u^{-1/2}$ in Eq. (1) are learnable (including possible parameters of function g and feature vectors of users and relations) and also requires supervised training like $\mathbf{W}$. Though enhancing the fitting ability of the model, this will inevitably make the optimization process prone to overfitting, since the only source of supervised signal is from user-item interactions outside GNN layers. Moreover, edge weights do play an essential role in representation learning on graphs, as highlighted by a large amount of prior works [10, 20, 21, 35, 38]. Therefore, more regularization on edge weights is needed to assist the learning of entity representations and to help generalize to unobserved interactions more efficiently.

Let's see how an ideal set of edge weights should be like. Consider a real-valued label function $l_u : \mathcal{E} \to \mathbb{R}$ on $\mathcal{G}$, which is constrained to take a specific value $l_u(v) = y_{uv}$ at node $v \in \mathcal{V} \subseteq \mathcal{E}$. In our context, $l_u(v) = 1$ if user u finds the item v relevant and has engaged with it, otherwise $l_u(v) = 0$. Intuitively, we hope that adjacent entities in the KG are likely to have similar relevancy labels, which is known as *label smoothness assumption*. This motivates our choice of energy function E:

$$E(l_u, \mathbf{A}_u) = \frac{1}{2} \sum_{e_i \in \mathcal{E}, e_j \in \mathcal{E}} A_u^{ij} \left(l_u(e_i) - l_u(e_j)\right)^2. \tag{2}$$

We show that the minimum-energy label function is *harmonic* by the following theorem:

Theorem 1. *The minimum-energy label function*

$$l_u^* = \operatorname*{arg\,min}_{l_u : l_u(v) = y_{uv}, \forall v \in \mathcal{V}} E(l_u, \mathbf{A}_u) \tag{3}$$

w.r.t. Eq. (2) is harmonic, i.e., l_u^* satisfies

$$l_u^*(e_i) = \frac{1}{D_u^{ii}} \sum_{e_j \in \mathcal{E}} A_u^{ij} \, l_u^*(e_j), \ \forall e_i \in \mathcal{E} \backslash \mathcal{V}. \tag{4}$$

Proof. Taking the derivative of the following equation

$$E(l_u, \mathbf{A}_u) = \frac{1}{2} \sum_{i,j} A_u^{ij} \left(l_u(e_i) - l_u(e_j)\right)^2$$

with respect to $l_u(e_i)$ where $e_i \in \mathcal{E} \backslash \mathcal{V}$, we have

$$\frac{\partial E(l_u, \mathbf{A}_u)}{\partial l_u(e_i)} = \sum_j A_u^{ij} \left(l_u(e_i) - l_u(e_j)\right).$$

The minimum-energy label function l_u^* should satisfy that

$$\left.\frac{\partial E(l_u, \mathbf{A}_u)}{\partial l_u(e_i)}\right|_{l_u = l_u^*} = 0.$$

Therefore, we have

$$l_u^*(e_i) = \frac{1}{\sum_j A_u^{ij}} \sum_j A_u^{ij} \, l_u^*(e_j) = \frac{1}{D_u^{ii}} \sum_j A_u^{ij} \, l_u^*(e_j), \ \forall e_i \in \mathcal{E} \backslash \mathcal{V}.$$

□

The harmonic property indicates that the value of l_u^* at each non-item entity $e_i \in \mathcal{E} \backslash \mathcal{V}$ is the average of its neighboring entities, which leads to the following label propagation scheme [39]:

Theorem 2. *Repeating the following two steps:*

(1) Propagate labels for all entities: $l_u(\mathcal{E}) \leftarrow \mathbf{D}_u^{-1}\mathbf{A}_u l_u(\mathcal{E})$, where $l_u(\mathcal{E})$ is the vector of labels for all entities;

(2) Reset labels of all items to initial labels: $l_u(\mathcal{V}) \leftarrow \mathbf{Y}[u, \mathcal{V}]^\top$, where $l_u(\mathcal{V})$ is the vector of labels for all items and $\mathbf{Y}[u, \mathcal{V}] = [y_{uv_1}, y_{uv_2}, \cdots]$ are initial labels;

will lead to $l_u \to l_u^*$.

Proof. Let $l_u(\mathcal{E}) = \begin{bmatrix} l_u(\mathcal{V}) \\ l_u(\mathcal{E} \backslash \mathcal{V}) \end{bmatrix}$. Since $l_u(\mathcal{V})$ is fixed on $\mathbf{Y}[u, \mathcal{V}]$, we are only interested in $l_u(\mathcal{E} \backslash \mathcal{V})$. We denote $\mathbf{P} = \mathbf{D}_u^{-1}\mathbf{A}_u$ (the subscript u is omitted from $\mathbf{P}$ for ease of notation), and partition matrix $\mathbf{P}$ into sub-matrices according to the partition of l_u:

$$\mathbf{P} = \begin{bmatrix} \mathbf{P}_{VV} & \mathbf{P}_{VE} \\ \mathbf{P}_{EV} & \mathbf{P}_{EE} \end{bmatrix}.$$

Then the label propagation scheme is equivalent to

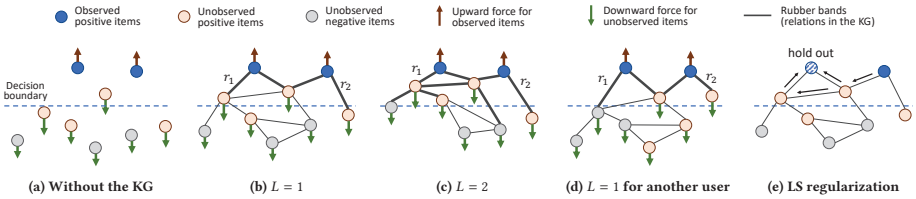
$$l_u(\mathcal{E} \backslash \mathcal{V}) \leftarrow \mathbf{P}_{EV} \mathbf{Y}[u, \mathcal{V}]^\top + \mathbf{P}_{EE} l_u(\mathcal{E} \backslash \mathcal{V}). \tag{5}$$

Repeat the above procedure, we have

$$l_u(\mathcal{E} \backslash \mathcal{V}) = \lim_{n \to \infty} (\mathbf{P}_{EE})^n l_u^{(0)}(\mathcal{E} \backslash \mathcal{V}) + \left(\sum_{i=1}^{n} (\mathbf{P}_{EE})^{i-1} \right) \mathbf{P}_{EV} \mathbf{Y}[u, \mathcal{V}]^\top, \tag{6}$$

where $l_u^{(0)}(\mathcal{E} \backslash \mathcal{V})$ is the initial value for $l_u(\mathcal{E} \backslash \mathcal{V})$. Now we show that $lim_{n \to \infty} (\mathbf{P}_{EE})^n l_u^{(0)}(\mathcal{E} \backslash \mathcal{V}) = \mathbf{0}$. Since $\mathbf{P}$ is row-normalized and $\mathbf{P}_{EE}$ is a sub-matrix of $\mathbf{P}$, we have

$$\exists \epsilon < 1, \ \sum_j \mathbf{P}_{EE}[i, j] \le \epsilon,$$



(a) Without the KG **(b)** $L = 1$ **(c)** $L = 2$ **(d)** $L = 1$ **for another user** **(e) LS regularization**

Figure 2: (a) Analogy of a physical equilibrium model for recommender systems; (b)-(d) Illustration of the effect of the KG; (e) Illustration of the effect of label smoothness regularization.

for all possible row index i. Therefore,

$$\begin{aligned}\sum_j (\mathbf{P}_{EE})^n[i,j] &= \sum_j \left((\mathbf{P}_{EE})^{(n-1)}\mathbf{P}_{EE}\right)[i,j] \\ &= \sum_j \sum_k (\mathbf{P}_{EE})^{(n-1)}[i,k]\,\mathbf{P}_{EE}[k,j] \\ &= \sum_k (\mathbf{P}_{EE})^{(n-1)}[i,k] \sum_j \mathbf{P}_{EE}[k,j] \\ &\le \sum_k (\mathbf{P}_{EE})^{(n-1)}[i,k]\,\epsilon \\ &\le \cdots \le \epsilon^n.\end{aligned}$$

As n goes infinity, the row sum of $(\mathbf{P}_{EE})^n$ converges to zero, which implies that $(\mathbf{P}_{EE})^n l_u^{(0)}(\mathcal{E}\backslash\mathcal{V}) \to \mathbf{0}$. It's clear that the choice of initial value $l_u^{(0)}(\mathcal{E}\backslash\mathcal{V})$ does not affect the convergence.

Since $lim_{n\to\infty}(\mathbf{P}_{EE})^n l_u^{(0)}(\mathcal{E}\backslash\mathcal{V}) = \mathbf{0}$, Eq. (6) becomes

$$l_u(\mathcal{E}\backslash\mathcal{V}) = \lim_{n\to\infty}\left(\sum_{i=1}^{n}(\mathbf{P}_{EE})^{i-1}\right)\mathbf{P}_{EV}\mathbf{Y}[u,\mathcal{V}]^\top.$$

Denote

$$\mathbf{T} = \lim_{n\to\infty}\sum_{i=1}^{n}(\mathbf{P}_{EE})^{i-1} = \sum_{i=1}^{\infty}(\mathbf{P}_{EE})^{i-1},$$

and we have

$$\mathbf{T} - \mathbf{T}\mathbf{P}_{EE} = \sum_{i=1}^{\infty}(\mathbf{P}_{EE})^{i-1} - \sum_{i=1}^{\infty}(\mathbf{P}_{EE})^{i} = \mathbf{I}.$$

Therefore, we derive that

$$\mathbf{T} = (I - \mathbf{P}_{EE})^{-1},$$

and

$$l_u(\mathcal{E}\backslash\mathcal{V}) = (I - \mathbf{P}_{EE})^{-1}\mathbf{P}_{EV}\mathbf{Y}[u,\mathcal{V}]^\top.$$

This is the unique fixed point and therefore the unique solution to Eq. (5). Repeating the steps in Theorem 2 leads to

$$l_u(\mathcal{E}) \to l_u^*(\mathcal{E}) = \begin{bmatrix}\mathbf{Y}[u,\mathcal{V}]^\top \\ (I - \mathbf{P}_{EE})^{-1}\mathbf{P}_{EV}\mathbf{Y}[u,\mathcal{V}]^\top\end{bmatrix}.$$

□

Theorem 2 provides a way for reaching the minimum-energy of relevancy label function E. However, l_u^* does not provide any signal for updating the edge weights matrix $\mathbf{A}_u$, since the labeled part of l_u^*, i.e., $l_u^*(\mathcal{V})$, equals their true relevancy labels $\mathbf{Y}[u,\mathcal{V}]$; Moreover, we do not know true relevancy labels for the unlabeled nodes $l_u^*(\mathcal{E}\backslash\mathcal{V})$.

To solve the issue, we propose minimizing the *leave-one-out* loss [35]. Suppose we hold out a single item v and treat it unlabeled. Then we predict its label by using the rest of (labeled) items and (unlabeled) non-item entities. The prediction process is identical to label propagation in Theorem 2, except that the label of item v is hidden and needs to be calculated. This way, the difference between the true relevancy label of v (i.e., y_{uv}) and the predicted label $\hat{l}_u(v)$ serves as a supervised signal for regularizing edge weights:

$$R(\mathbf{A}) = \sum_u R(\mathbf{A}_u) = \sum_u \sum_v J\big(y_{uv}, \hat{l}_u(v)\big), \tag{7}$$

where J is the cross-entropy loss function. Given the regularization in Eq. (7), an ideal edge weight matrix $\mathbf{A}$ should reproduce the true relevancy label of each held-out item while also satisfying the smoothness of relevancy labels.

4.3 The Unified Loss Function

Combining knowledge-aware graph neural networks and LS regularization, we reach the following complete loss function:

$$\min_{\mathbf{W},\mathbf{A}} \mathcal{L} = \min_{\mathbf{W},\mathbf{A}} \sum_{u,v} J(y_{uv}, \hat{y}_{uv}) + \lambda R(\mathbf{A}) + \gamma\|\mathcal{F}\|_2^2, \tag{8}$$

where $\|\mathcal{F}\|_2^2$ is the L2-regularizer, λ and γ are balancing hyper-parameters. In Eq. (8), the first term corresponds to the part of GNN that learns the transformation matrix $\mathbf{W}$ and edge weights $\mathbf{A}$ simultaneously, while the second term $R(\cdot)$ corresponds to the part of label smoothness that can be seen as adding constraint on edge weights $\mathbf{A}$. Therefore, $R(\cdot)$ serves as regularization on $\mathbf{A}$ to assist GNN in learning edge weights.

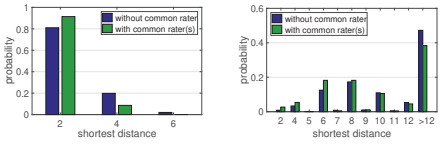
It is also worth noticing that the first term can be seen as *feature propagation* on the KG while the second term $R(\cdot)$ can be seen as *label propagation* on the KG. A recommender for a specific user u is actually a mapping from item features to user-item interaction labels, i.e., $\mathcal{F}_u : \mathbf{E}_v \to y_{uv}$ where $\mathbf{E}_v$ is the feature vector of item v. Therefore, Eq. (8) utilizes the structural information of the KG on both the feature side and the label side of $\mathcal{F}_u$ to capture users' higher-order preferences.

4.4 Discussion

How can the knowledge graph help find users' interests? To intuitively understand the role of the KG, we make an analogy with a

	Movie	Book	Music	Restaurant
# users	138,159	19,676	1,872	2,298,698
# items	16,954	20,003	3,846	1,362
# interactions	13,501,622	172,576	42,346	23,416,418
# entities	102,569	25,787	9,366	28,115
# relations	32	18	60	7
# KG triples	499,474	60,787	15,518	160,519

Table 2: Statistics of the four datasets: MovieLens-20M (movie), Book-Crossing (book), Last.FM (music), and Dianping-Food (restaurant).



(a) MovieLens-20M **(b) Last.FM**

Figure 3: Probability distribution of the shortest path distance between two randomly sampled items in the KG under the circumstance that (1) they have no common user in the dataset; (2) they have common user(s) in the dataset.

physical equilibrium model as shown in Figure 2. Each entity/item is seen as a particle, while the supervised positive user-relevancy signal acts as the force pulling the observed positive items up from the decision boundary and the negative items signal acts as the force pushing the unobserved items down. Without the KG (Figure 2a), these items are only loosely connected with each other through the collaborative filtering effect (which is not drawn here for clarity). In contrast, edges in the KG serve as the rubber bands that impose explicit constraints on connected entities. When number of layers is $L = 1$ (Figure 2b), representation of each entity is a mixture of itself and its immediate neighbors, therefore, optimizing on the positive items will simultaneously pull their immediate neighbors up together. The upward force goes deeper in the KG with the increase of L (Figure 2c), which helps explore users' long-distance interests and pull up more positive items. It is also interesting to note that the proximity constraint exerted by the KG is *personalized* since the strength of the rubber band (i.e., $s_u(r)$) is *user-specific* and *relation-specific*: One user may prefer relation r_1 (Figure 2b) while another user (with same observed items but different unobserved items) may prefer relation r_2 (Figure 2d).

Despite the force exerted by edges in the KG, edge weights may be set inappropriately, for example, too small to pull up the unobserved items (i.e., rubber bands are too weak). Next, we show by Figure 2e that how the label smoothness assumption helps regularizing the learning of edge weights. Suppose we hold out the positive sample in the upper left and we intend to reproduce its label by the rest of items. Since the true relevancy label of the held-out sample is 1 and the upper right sample has the largest label value, the LS regularization term $R(\mathbf{A})$ would enforce the edges with arrows to be large so that the label can "flow" from the blue one to the striped one as much as possible. As a result, this will tighten the rubber bands (denoted by arrows) and encourage the model to pull up the two upper pink items to a greater extent.

5 EXPERIMENTS

In this section, we evaluate the proposed KGNN-LS model, and present its performance on four real-world scenarios: movie, book, music, and restaurant recommendations.

5.1 Datasets

We utilize the following four datasets in our experiments for movie, book, music, and restaurant recommendations, respectively, in which the first three are public datasets and the last one is from Meituan-Dianping Group. We use Satori[3], a commercial KG built by Microsoft, to construct sub-KGs for MovieLens-20M, Book-Crossing, and Last.FM datasets. The KG for Dianping-Food dataset is constructed by the internal toolkit of Meituan-Dianping Group. Further details of datasets are provided in Appendix A.

- **MovieLens-20M**[4] is a widely used benchmark dataset in movie recommendations, which consists of approximately 20 million explicit ratings (ranging from 1 to 5) on the MovieLens website. The corresponding KG contains 102,569 entities, 499,474 edges and 32 relation-types.
- **Book-Crossing**[5] contains 1 million ratings (ranging from 0 to 10) of books in the Book-Crossing community. The corresponding KG contains 25,787 entities, 60,787 edges and 18 relation-types.
- **Last.FM**[6] contains musician listening information from a set of 2 thousand users from Last.fm online music system. The corresponding KG contains 9,366 entities, 15,518 edges and 60 relation-types.
- **Dianping-Food** is provided by Dianping.com[7], which contains over 10 million interactions (including clicking, buying, and adding to favorites) between approximately 2 million users and 1 thousand restaurants. The corresponding KG contains 28,115 entities, 160,519 edges and 7 relation-types.

The statistics of the four datasets are shown in Table 2.

5.2 Baselines

We compare the proposed KGNN-LS model with the following baselines for recommender systems, in which the first two baselines are KG-free while the rest are all KG-aware methods. The hyper-parameter setting of KGNN-LS is provided in Appendix B.

- **SVD** [12] is a classic CF-based model using inner product to model user-item interactions. We use the unbiased version (i.e., the predicted engaging probability is modeled as $y_{uv} = \mathbf{u}^\top \mathbf{v}$). The dimension and learning rate for the four datasets are set as: $d = 8$, $\eta = 0.5$ for MovieLens-20M, Book-Crossing; $d = 8$, $\eta = 0.1$ for Last.FM; $d = 32$, $\eta = 0.1$ for Dianping-Food.

[3] https://searchengineland.com/library/bing/bing-satori
[4] https://grouplens.org/datasets/movielens/
[5] http://www2.informatik.uni-freiburg.de/~cziegler/BX/
[6] https://grouplens.org/datasets/hetrec-2011/
[7] https://www.dianping.com/

Model	MovieLens-20M				Book-Crossing				Last.FM				Dianping-Food			
	R@2	*R@10*	*R@50*	*R@100*	*R@2*	*R@10*	*R@50*	*R@100*	*R@2*	*R@10*	*R@50*	*R@100*	*R@2*	*R@10*	*R@50*	*R@100*
SVD	0.036	0.124	0.277	0.401	0.027	0.046	0.077	0.109	0.029	0.098	0.240	0.332	0.039	0.152	0.329	0.451
LibFM	0.039	0.121	0.271	0.388	0.033	0.062	0.092	0.124	0.030	0.103	0.263	0.330	0.043	0.156	0.332	0.448
LibFM + TransE	0.041	0.125	0.280	0.396	0.037	0.064	0.097	0.130	0.032	0.102	0.259	0.326	0.044	0.161	**0.343**	0.455
PER	0.022	0.077	0.160	0.243	0.022	0.041	0.064	0.070	0.014	0.052	0.116	0.176	0.023	0.102	0.256	0.354
CKE	0.034	0.107	0.244	0.322	0.028	0.051	0.079	0.112	0.023	0.070	0.180	0.296	0.034	0.138	0.305	0.437
RippleNet	**0.045**	0.130	0.278	0.447	0.036	0.074	0.107	0.127	0.032	0.101	0.242	0.336	0.040	0.155	0.328	0.440
KGNN-LS	0.043	**0.155**	**0.321**	**0.458**	**0.045**	**0.082**	**0.117**	**0.149**	**0.044**	**0.122**	**0.277**	**0.370**	**0.047**	**0.170**	0.340	**0.487**

Table 3: The results of *Recall@K* in top-K recommendation.

Model	Movie	Book	Music	Restaurant
SVD	0.963	0.672	0.769	0.838
LibFM	0.959	0.691	0.778	0.837
LibFM + TransE	0.966	0.698	0.777	0.839
PER	0.832	0.617	0.633	0.746
CKE	0.924	0.677	0.744	0.802
RippleNet	0.960	0.727	0.770	0.833
KGNN-LS	**0.979**	**0.744**	**0.803**	**0.850**

Table 4: The results of *AUC* in CTR prediction.

- **LibFM** [16] is a widely used feature-based factorization model for CTR prediction. We concatenate user ID and item ID as input for LibFM. The dimension is set as $\{1, 1, 8\}$ and the number of training epochs is 50 for all datasets.
- **LibFM + TransE** extends LibFM by attaching an entity representation learned by TransE [2] to each user-item pair. The dimension of TransE is 32 for all datasets.
- **PER** [33] is a representative of path-based methods, which treats the KG as heterogeneous information networks and extracts meta-path based features to represent the connectivity between users and items. We use manually designed "user-item-attribute-item" as meta-paths, i.e., "user-movie-director-movie", "user-movie-genre-movie", and "user-movie-star-movie" for MovieLens-20M; "user-book-author-book" and "user-book-genre-book" for Book-Crossing, "user-musician-date_of_birth-musician" (date of birth is discretized), "user-musician-country-musician", and "user-musician-genre-musician" for Last.FM; "user-restaurant-dish-restaurant", "user-restaurant-business_area-restaurant", "user-restaurant-tag-restaurant" for Dianping-Food. The settings of dimension and learning rate are the same as SVD.
- **CKE** [34] is a representative of embedding-based methods, which combines CF with structural, textual, and visual knowledge in a unified framework. We implement CKE as CF plus a structural knowledge module in this paper. The dimension of embedding for the four datasets are 64, 128, 64, 64. The training weight for KG part is 0.1 for all datasets. The learning rate are the same as in SVD.
- **RippleNet** [24] is a representative of hybrid methods, which is a memory-network-like approach that propagates users' preferences on the KG for recommendation. For RippleNet, $d = 8$, $H = 2$, $\lambda_1 = 10^{-6}$, $\lambda_2 = 0.01$, $\eta = 0.01$ for MovieLens-20M; $d = 16$, $H = 3$, $\lambda_1 = 10^{-5}$, $\lambda_2 = 0.02$, $\eta = 0.005$ for Last.FM; $d = 32$, $H = 2$, $\lambda_1 = 10^{-7}$, $\lambda_2 = 0.02$, $\eta = 0.01$ for Dianping-Food.

5.3 Validating the Connection between $\mathcal{G}$ and Y

To validate the connection between the knowledge graph $\mathcal{G}$ and user-item interaction Y, we conduct an empirical study where we investigate the correlation between the shortest path distance of two randomly sampled items in the KG and whether they have common user(s) in the dataset, that is there exist user(s) that interacted with both items. For MovieLens-20M and Last.FM, we randomly sample ten thousand item pairs that have no common users and have at least one common user, respectively, then count the distribution of their shortest path distances in the KG. The results are presented in Figure 3, which clearly show that *if two items have common user(s) in the dataset, they are likely to be more close in the KG*. For example, if two movies have common user(s) in MovieLens-20M, there is a probability of 0.92 that they will be within 2 hops in the KG, while the probability is 0.80 if they have no common user. This finding empirically demonstrates that exploiting the proximity structure of the KG can assist making recommendations. This also justifies our motivation to use label smoothness regularization to help learn entity representations.

5.4 Results

5.4.1 Comparison with Baselines. We evaluate our method in two experiment scenarios: (1) In top-K recommendation, we use the trained model to select K items with highest predicted click probability for each user in the test set, and choose *Recall@K* to evaluate the recommended sets. (2) In click-through rate (CTR) prediction, we apply the trained model to predict each piece of user-item pair in the test set (including positive items and randomly selected negative items). We use *AUC* as the evaluation metric in CTR prediction.

The results of top-K recommendation and CTR prediction are presented in Tables 3 and 4, respectively, which show that KGNN-LS outperforms baselines by a significant margin. For example, the *AUC* of KGNN-LS surpasses baselines by 5.1%, 6.9%, 8.3%, and 4.3% on average in MovieLens-20M, Book-Crossing, Last.FM, and Dianping-Food datasets, respectively.

We also show daily performance of KGNN-LS and baselines on Dianping-Food to investigate performance stability. Figure 4 shows their *AUC* score from September 1, 2018 to September 30, 2018. We notice that the curve of KGNN-LS is consistently above baselines over the test period; Moreover, the performance of KGNN-LS is also

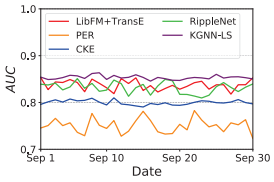


Figure 4: Daily *AUC* of all methods on Dianping-Food in September 2018.

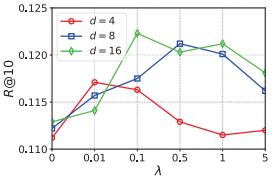


Figure 5: Effectiveness of LS regularization on Last.FM.

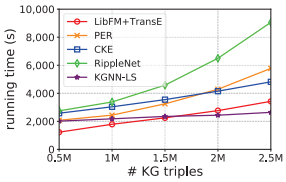


Figure 6: Running time of all methods w.r.t. KG size on MovieLens-20M.

r	20%	40%	60%	80%	100%
SVD	0.882	0.913	0.938	0.955	0.963
LibFM	0.902	0.923	0.938	0.950	0.959
LibFM+TransE	0.914	0.935	0.949	0.960	0.966
PER	0.802	0.814	0.821	0.828	0.832
CKE	0.898	0.910	0.916	0.921	0.924
RippleNet	0.921	0.937	0.947	0.955	0.960
KGNN-LS	**0.961**	**0.970**	**0.974**	**0.977**	**0.979**

Table 5: *AUC* of all methods w.r.t. the ratio of training set *r*.

L	1	2	3	4
MovieLens-20M	**0.155**	0.146	0.122	0.011
Book-Crossing	0.077	**0.082**	0.043	0.008
Last.FM	**0.122**	0.106	0.105	0.057
Dianping-Food	0.165	**0.170**	0.061	0.036

Table 6: *R*@10 w.r.t. the number of layers *L*.

d	4	8	16	32	64	128
MovieLens-20M	0.134	0.141	0.143	**0.155**	**0.155**	0.151
Book-Crossing	0.065	0.073	0.077	0.081	**0.082**	0.080
Last.FM	0.111	0.116	**0.122**	0.109	0.102	0.107
Dianping-Food	0.155	**0.170**	0.167	0.166	0.163	0.161

Table 7: *R*@10 w.r.t. the dimension of hidden layers *d*.

with low variance, which suggests that KGNN-LS is also robust and stable in practice.

5.4.2 Effectiveness of LS Regularization. Is the proposed LS regularization helpful in improving the performance of GNN? To study the effectiveness of LS regularization, we fix the dimension of hidden layers as 4, 8, and 16, then vary λ from 0 to 5 to see how performance changes. The results of *R*@10 in Last.FM dataset are plotted in Figure 5. It is clear that the performance of KGNN-LS with a non-zero λ is better than $\lambda = 0$ (the case of Wang et al. [28]), which justifies our claim that LS regularization can assist learning the edge weights in a KG and achieve better generalization in recommender systems. But note that a too large λ is less favorable, since it overwhelms the overall loss and misleads the direction of gradients. According to the experiment results, we find that a λ between 0.1 and 1.0 is preferable in most cases.

5.4.3 Results in cold-start scenarios. One major goal of using KGs in recommender systems is to alleviate the sparsity issue. To investigate the performance of KGNN-LS in cold-start scenarios, we vary the size of training set of MovieLens-20M from $r = 100\%$ to $r = 20\%$ (while the validation and test set are kept fixed), and report the results of *AUC* in Table 5. When $r = 20\%$, *AUC* decreases by 8.4%, 5.9%, 5.4%, 3.6%, 2.8%, and 4.1% for the six baselines compared to the model trained on full training data ($r = 100\%$), but the performance decrease of KGNN-LS is only 1.8%. This demonstrates that KGNN-LS still maintains predictive performance even when user-item interactions are sparse.

5.4.4 Hyper-parameters Sensitivity. We first analyze the sensitivity of KGNN-LS to the number of GNN layers *L*. We vary *L* from 1 to 4 while keeping other hyper-parameters fixed. The results are shown in Table 6. We find that the model performs poorly when $L = 4$, which is because a larger *L* will mix too many entity embeddings in a given entity, which *over-smoothes* the representation learning on KGs. KGNN-LS achieves the best performance when $L = 1$ or 2 in the four datasets.

We also examine the impact of the dimension of hidden layers *d* on the performance of KGNN-LS. The result in shown in Table 7. We observe that the performance is boosted with the increase of *d* at the beginning, because more bits in hidden layers can improve the model capacity. However, the performance drops when *d* further increases, since a too large dimension may overfit datasets. The best performance is achieved when $d = 8 \sim 64$.

5.5 Running Time Analysis

We also investigate the running time of our method with respect to the size of KG. We run experiments on a Microsoft Azure virtual machine with 1 NVIDIA Tesla M60 GPU, 12 Intel Xeon CPUs (E5-2690 v3 @2.60GHz), and 128GB of RAM. The size of the KG is increased by up to five times the original one by extracting more triples from Satori, and the running times of all methods on MovieLens-20M are reported in Figure 6. Note that the trend of a curve matters more than the real values, since the values are largely dependent on the minibatch size and the number of epochs (yet we did try to align the configurations of all methods). The result show that KGNN-LS exhibits strong scalability even when the KG is large.

6 CONCLUSION AND FUTURE WORK

In this paper, we propose knowledge-aware graph neural networks with label smoothness regularization for recommendation. KGNN-LS applies GNN architecture to KGs by using user-specific relation

scoring functions and aggregating neighborhood information with different weights. In addition, the proposed label smoothness constraint and leave-one-out loss provide strong regularization for learning the edge weights in KGs. We also discuss how KGs benefit recommender systems and how label smoothness can assist learning the edge weights. Experiment results show that KGNN-LS outperforms state-of-the-art baselines in four recommendation scenarios and achieves desirable scalability with respect to KG size.

In this paper, LS regularization is proposed for recommendation task with KGs. It is interesting to examine the LS assumption on other graph tasks such as link prediction and node classification. Investigating the theoretical relationship between feature propagation and label propagation is also a promising direction.

Acknowledgements. This research has been supported in part by NSF OAC-1835598, DARPA MCS, ARO MURI, Boeing, Docomo, Hitachi, Huawei, JD, Siemens, and Stanford Data Science Initiative.

REFERENCES

[1] Shumeet Baluja, Rohan Seth, D Sivakumar, Yushi Jing, Jay Yagnik, Shankar Kumar, Deepak Ravichandran, and Mohamed Aly. 2008. Video suggestion and discovery for youtube: taking random walks through the view graph. In *Proceedings of the 17th international conference on World Wide Web*. ACM, 895–904.

[2] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. 2013. Translating embeddings for modeling multi-relational data. In *Advances in Neural Information Processing Systems*. 2787–2795.

[3] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann Lecun. 2014. Spectral networks and locally connected networks on graphs. In *the 2nd International Conference on Learning Representations*.

[4] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*. ACM, 191–198.

[5] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. 2016. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems*. 3844–3852.

[6] David K Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P Adams. 2015. Convolutional networks on graphs for learning molecular fingerprints. In *Advances in Neural Information Processing Systems*. 2224–2232.

[7] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*. 1024–1034.

[8] Binbin Hu, Chuan Shi, Wayne Xin Zhao, and Philip S Yu. 2018. Leveraging meta-path based context for top-n recommendation with a neural co-attention model. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 1531–1540.

[9] Jin Huang, Wayne Xin Zhao, Hongjian Dou, Ji-Rong Wen, and Edward Y Chang. 2018. Improving Sequential Recommendation with Knowledge-Enhanced Memory Networks. In *the 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*. ACM, 505–514.

[10] Masayuki Karasuyama and Hiroshi Mamitsuka. 2013. Manifold-based similarity adaptation for label propagation. In *Advances in neural information processing systems*. 1547–1555.

[11] Thomas N Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. In *the 5th International Conference on Learning Representations*.

[12] Yehuda Koren. 2008. Factorization meets the neighborhood: a multifaceted collaborative filtering model. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 426–434.

[13] Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix factorization techniques for recommender systems. *Computer* 8 (2009), 30–37.

[14] Federico Monti, Michael Bronstein, and Xavier Bresson. 2017. Geometric matrix completion with recurrent multi-graph neural networks. In *Advances in Neural Information Processing Systems*. 3697–3707.

[15] Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. 2016. Learning convolutional neural networks for graphs. In *International Conference on Machine Learning*. 2014–2023.

[16] Steffen Rendle. 2012. Factorization machines with libfm. *ACM Transactions on Intelligent Systems and Technology (TIST)* 3, 3 (2012), 57.

[17] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. 2018. Modeling relational data with graph convolutional networks. In *European Semantic Web Conference*. Springer, 593–607.

[18] Zhu Sun, Jie Yang, Jie Zhang, Alessandro Bozzon, Long-Kai Huang, and Chi Xu. 2018. Recurrent knowledge graph embedding for effective recommendation. In *Proceedings of the 12th ACM Conference on Recommender Systems*. ACM, 297–305.

[19] Rianne van den Berg, Thomas N Kipf, and Max Welling. 2017. Graph Convolutional Matrix Completion. *stat* 1050 (2017), 7.

[20] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2018. Graph attention networks. In *Proceedings of the 6th International Conferences on Learning Representations*.

[21] Fei Wang and Changshui Zhang. 2008. Label propagation through linear neighborhoods. *IEEE Transactions on Knowledge and Data Engineering* 20, 1 (2008), 55–67.

[22] Hongwei Wang, Jia Wang, Miao Zhao, Jiannong Cao, and Minyi Guo. 2017. Joint topic-semantic-aware social recommendation for online voting. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. ACM, 347–356.

[23] Hongwei Wang, Fuzheng Zhang, Min Hou, Xing Xie, Minyi Guo, and Qi Liu. 2018. Shine: Signed heterogeneous information network embedding for sentiment link prediction. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*. ACM, 592–600.

[24] Hongwei Wang, Fuzheng Zhang, Jialin Wang, Miao Zhao, Wenjie Li, Xing Xie, and Minyi Guo. 2018. RippleNet: Propagating User Preferences on the Knowledge Graph for Recommender Systems. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*. ACM, 417–426.

[25] Hongwei Wang, Fuzheng Zhang, Jialin Wang, Miao Zhao, Wenjie Li, Xing Xie, and Minyi Guo. 2019. Exploring High-Order User Preference on the Knowledge Graph for Recommender Systems. *ACM Transactions on Information Systems (TOIS)* 37, 3 (2019), 32.

[26] Hongwei Wang, Fuzheng Zhang, Xing Xie, and Minyi Guo. 2018. DKN: Deep Knowledge-Aware Network for News Recommendation. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*. 1835–1844.

[27] Hongwei Wang, Fuzheng Zhang, Miao Zhao, Wenjie Li, Xing Xie, and Minyi Guo. 2019. Multi-Task Feature Learning for Knowledge Graph Enhanced Recommendation. In *Proceedings of the 2019 World Wide Web Conference on World Wide Web*.

[28] Hongwei Wang, Miao Zhao, Xing Xie, Wenjie Li, and Minyi Guo. 2019. Knowledge graph convolutional networks for recommender systems. In *Proceedings of the 2019 World Wide Web Conference on World Wide Web*.

[29] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. 2018. Billion-scale commodity embedding for e-commerce recommendation in alibaba. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 839–848.

[30] Quan Wang, Zhendong Mao, Bin Wang, and Li Guo. 2017. Knowledge graph embedding: A survey of approaches and applications. *IEEE Transactions on Knowledge and Data Engineering* 29, 12 (2017), 2724–2743.

[31] Yuexin Wu, Hanxiao Liu, and Yiming Yang. 2018. Graph Convolutional Matrix Completion for Bipartite Edge Prediction. In *the 10th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management*.

[32] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. 2018. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 974–983.

[33] Xiao Yu, Xiang Ren, Yizhou Sun, Quanquan Gu, Bradley Sturt, Urvashi Khandelwal, Brandon Norick, and Jiawei Han. 2014. Personalized entity recommendation: A heterogeneous information network approach. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*. ACM, 283–292.

[34] Fuzheng Zhang, Nicholas Jing Yuan, Defu Lian, Xing Xie, and Wei-Ying Ma. 2016. Collaborative knowledge base embedding for recommender systems. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 353–362.

[35] Xinhua Zhang and Wee S Lee. 2007. Hyperparameter learning for graph based semi-supervised learning algorithms. In *Advances in neural information processing systems*. 1585–1592.

[36] Huan Zhao, Quanming Yao, Jianda Li, Yangqiu Song, and Dik Lun Lee. 2017. Meta-graph based recommendation fusion over heterogeneous information networks. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 635–644.

[37] Dengyong Zhou, Olivier Bousquet, Thomas N Lal, Jason Weston, and Bernhard Schölkopf. 2004. Learning with local and global consistency. In *Advances in neural information processing systems*. 321–328.

[38] Xiaojin Zhu, Zoubin Ghahramani, and John D Lafferty. 2003. Semi-supervised learning using gaussian fields and harmonic functions. In *Proceedings of the 20th International conference on Machine learning*. 912–919.

[39] Xiaojin Zhu, John Lafferty, and Ronald Rosenfeld. 2005. *Semi-supervised learning with graphs*. Ph.D. Dissertation. Carnegie Mellon University, language technologies institute, school of computer science.

APPENDIX

A Additional Details on Datasets

MovieLens-20M, Book-Crossing, and Last.FM datasets contain explicit feedbacks data (Last.FM provides the listening count as weight for each user-item interaction). Therefore, we transform them into implicit feedback, where each entry is marked with 1 indicating that the user has rated the item positively. The threshold of positive rating is 4 for MovieLens-20M, while no threshold is set for Book-Crossing and Last.FM due to their data sparsity. Additionally, we randomly sample an unwatched set of items and mark them as 0 for each user, the number of which equals his/her positively-rated ones.

We use Microsoft Satori to construct the KGs for MovieLens-20M, Book-Crossing, and Last.FM datasets. In each triple of Satori KG, the head and the tail are either IDs or textual content, and the relation follows the format "domain.head_category.tail_category" (e.g., "book.book.author"). We first select a subset of triples from the whole Satori KG with a confidence level greater than 0.9. Then we collect Satori IDs of all valid movies/books/musicians by matching their names with the tail of triples *(head, film.film.name, tail)*, *(head, book.book.title, tail)*, or *(head, type.object.name, tail)*, respectively, for the three datasets. Items with multiple matched or no matched entities are excluded for simplicity. After obtaining the set of item IDs, we match these item IDs with the head of all triples in Satori sub-KG, and select all well-matched triples as the final KG for each dataset.

Dianping-Food dataset is collected from Dianping.com, a Chinese group buying website hosting consumer reviews of restaurants similar to Yelp. We select approximately 10 million interactions between users and restaurants in Dianping.com from May 1, 2015 to December 12, 2018. The types of positive interactions include clicking, buying, and adding to favorites, and we sample negative interactions for each user. The KG for Dianping-Food is collected from Meituan Brain, an internal knowledge graph built for dining and entertainment by Meituan-Dianping Group. The types of entities include POI (restaurant), city, first-level and second-level category, star, business area, dish, and tag; The types of relations correspond to the types of entities (e.g., "organization.POI.has_dish").

	Movie	Book	Music	Restaurant
S	16	8	8	4
d	32	64	16	8
L	1	2	1	2
λ	1.0	0.5	0.1	0.5
γ	10^{-7}	2×10^{-5}	10^{-4}	10^{-7}
η	2×10^{-2}	2×10^{-4}	5×10^{-4}	2×10^{-2}

Table 8: Hyper-parameter settings for the four datasets (S: number of sampled neighbors for each entity; d: dimension of hidden layers, L: number of layers, λ: label smoothness regularizer weight, γ: L2 regularizer weight, η: learning rate).

B Additional Details on Hyper-parameter Searching

In KGNN-LS, we set functions g and f as inner product, σ as *ReLU* for non-last-layers and *tanh* for the last-layer. Note that the number of neighbors of an entity in a KG may be significantly different from each other. Therefore, we uniformly sample a fixed-size set of neighbors for each entity instead of using all of its neighbors to keep the computation more efficient. The number of sampled neighbors for each entity is denoted by S. Hyper-parameter settings for the four datasets are given in Table 8, which are determined by optimizing $R@10$ on a validation set. The search spaces for hyper-parameters are as follows:

- $S = \{2, 4, 8, 16, 32\}$;
- $d = \{4, 8, 16, 32, 64, 128\}$;
- $L = \{1, 2, 3, 4\}$;
- $\lambda = \{0, 0.01, 0.1, 0.5, 1, 5\}$;
- $\gamma = \{10^{-9}, 10^{-8}, 10^{-7}, 2\times10^{-7}, 5\times10^{-7}, 10^{-6}, 2\times10^{-6}, 5\times10^{-6}, 10^{-5}, 2\times10^{-5}, 5\times10^{-5}, 10^{-4}, 2\times10^{-4}, 5\times10^{-4}, 10^{-3}\}$;
- $\eta = \{10^{-5}, 2\times10^{-5}, 5\times10^{-5}, 10^{-4}, 2\times10^{-4}, 5\times10^{-4}, 10^{-3}, 2\times10^{-3}, 5\times10^{-3}, 10^{-2}, 2\times10^{-2}, 5\times10^{-2}, 10^{-1}\}$.

For each dataset, the ratio of training, validation, and test set is $6:2:2$. Each experiment is repeated 5 times, and the average performance is reported. All trainable parameters are optimized by Adam algorithm.

The Web Conference 2019

Multi-Task Feature Learning for Knowledge Graph Enhanced Recommendation

Hongwei Wang[1,2], Fuzheng Zhang[3], Miao Zhao[4], Wenjie Li[4], Xing Xie[2], Minyi Guo[1*]
[1]Shanghai Jiao Tong University, Shanghai, China
[2]Microsoft Research Asia, Beijing, China
[3]Meituan-Dianping Group, Beijing, China
[4]The Hong Kong Polytechnic University, Hong Kong, China
wanghongwei55@gmail.com,zhangfuzheng@meituan.com,{csmiaozhao,cswjli}@comp.polyu.edu.hk
xingx@microsoft.com,guo-my@cs.sjtu.edu.cn

ABSTRACT

Collaborative filtering often suffers from sparsity and cold start problems in real recommendation scenarios, therefore, researchers and engineers usually use side information to address the issues and improve the performance of recommender systems. In this paper, we consider knowledge graphs as the source of side information. We propose **MKR**, a **M**ulti-task feature learning approach for **K**nowledge graph enhanced **R**ecommendation. MKR is a deep end-to-end framework that utilizes knowledge graph embedding task to assist recommendation task. The two tasks are associated by cross&compress units, which automatically share latent features and learn high-order interactions between items in recommender systems and entities in the knowledge graph. We prove that cross&compress units have sufficient capability of polynomial approximation, and show that MKR is a generalized framework over several representative methods of recommender systems and multi-task learning. Through extensive experiments on real-world datasets, we demonstrate that MKR achieves substantial gains in movie, book, music, and news recommendation, over state-of-the-art baselines. MKR is also shown to be able to maintain a decent performance even if user-item interactions are sparse.

KEYWORDS

Recommender systems; knowledge graph; multi-task learning

ACM Reference Format:
Hongwei Wang, Fuzheng Zhang, Miao Zhao, Wenjie Li, Xing Xie, and Minyi Guo. 2019. Multi-Task Feature Learning for Knowledge Graph Enhanced Recommendation In *Proceedings of The 2019 Web Conference (WWW 2019).* ACM, New York, NY, USA, 11 pages. https://doi.org/xxxxx

1 INTRODUCTION

Recommender systems (RS) aims to address the information explosion and meet users personalized interests. One of the most popular recommendation techniques is collaborative filtering (CF) [11], which utilizes users' historical interactions and makes recommendations based on their common preferences. However, CF-based methods usually suffer from the sparsity of user-item interactions and the cold start problem. Therefore, researchers propose using *side information* in recommender systems, including social networks [10], attributes [30], and multimedia (e.g., texts [29], images [40]). *Knowledge graphs* (KGs) are one type of side information for RS, which usually contain fruitful facts and connections about items. Recently, researchers have proposed several academic and commercial KGs, such as NELL[1], DBpedia[2], Google Knowledge Graph[3] and Microsoft Satori[4]. Due to its high dimensionality and heterogeneity, a KG is usually pre-processed by *knowledge graph embedding* (KGE) methods [27], which embeds entities and relations into low-dimensional vector spaces while preserving its inherent structure.

Existing KG-aware methods

Inspired by the success of applying KG in a wide variety of tasks, researchers have recently tried to utilize KG to improve the performance of recommender systems [31, 32, 39, 40, 45]. Personalized Entity Recommendation (PER) [39] and Factorization Machine with Group lasso (FMG) [45] treat KG as a heterogeneous information network, and extract meta-path/meta-graph based latent features to represent the connectivity between users and items along different types of relation paths/graphs. It should be noted that PER and FMG rely heavily on manually designed meta-paths/meta-graphs, which limits its application in generic recommendation scenarios. Deep Knowledge-aware Network (DKN) [32] designs a CNN framework to combine entity embeddings with word embeddings for news recommendation. However, the entity embeddings are required in advance of using DKN, causing DKN to lack an end-to-end way of training. Another concern about DKN is that it can hardly incorporate side information other than texts. RippleNet [31] is a memory-network-like model that propagates users' potential preferences in the KG and explores their hierarchical interests. But the importance of relations is weakly characterized in RippleNet, because the embedding matrix of a relation $\mathbf{R}$ can hardly be trained to capture the sense of importance in the quadratic form $\mathbf{v}^\top \mathbf{R}\mathbf{h}$ ($\mathbf{v}$ and $\mathbf{h}$ are embedding vectors of two entities). Collaborative Knowledge base Embedding (CKE) [40] combines CF with structural knowledge, textual knowledge, and visual knowledge in a unified framework. However, the KGE module in CKE (i.e., TransR [13]) is more suitable for in-graph applications (such as KG completion and link prediction) rather than recommendation. In addition, the CF module and the KGE module are loosely coupled

*M. Guo is the corresponding author. This work was partially sponsored by the National Basic Research 973 Program of China under Grant 2015CB352403.

WWW 2019, May 13–17, 2019, San Francisco, USA
2019. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1]http://rtw.ml.cmu.edu/rtw/
[2]http://wiki.dbpedia.org/
[3]https://developers.google.com/knowledge-graph/
[4]https://searchengineland.com/library/bing/bing-satori

in CKE under a Bayesian framework, making the supervision from KG less obvious for recommender systems.

The proposed approach

To address the limitations of previous work, we propose MKR, a multi-task learning (MTL) approach for knowledge graph enhanced recommendation. MKR is a generic, end-to-end deep recommendation framework, which aims to utilize KGE task to assist recommendation task[5]. Note that the two tasks are not mutually independent, but are highly correlated since an item in RS may associate with one or more entities in KG. Therefore, an item and its corresponding entity are likely to have a similar proximity structure in RS and KG, and share similar features in low-level and non-task-specific latent feature spaces [15]. We will further validate the similarity in the experiments section. To model the shared features between items and entities, we design a *cross&compress unit* in MKR. The cross&compress unit explicitly models high-order interactions between item and entity features, and automatically control the cross knowledge transfer for both tasks. Through cross&compress units, representations of items and entities can complement each other, assisting both tasks in avoiding fitting noises and improving generalization. The whole framework can be trained by alternately optimizing the two tasks with different frequencies, which endows MKR with high flexibility and adaptability in real recommendation scenarios.

We probe the expressive capability of MKR and show, through theoretical analysis, that the cross&compress unit is capable of approximating sufficiently high order feature interactions between items and entities. We also show that MKR is a generalized framework over several representative methods of recommender systems and multi-task learning, including factorization machines [22, 23], deep&cross network [34], and cross-stitch network [18]. Empirically, we evaluate our method in four recommendation scenarios, i.e., movie, book, music, and news recommendations. The results demonstrate that MKR achieves substantial gains over state-of-the-art baselines in both click-through rate (CTR) prediction (e.g., 11.6% *AUC* improvements on average for movies) and top-K recommendation (e.g., 66.4% *Recall*@10 improvements on average for books). MKR can also maintain a decent performance in sparse scenarios.

Contribution

It is worth noticing that the problem studied in this paper can also be modelled as *cross-domain recommendation* [26] or *transfer learning* [21], since we care more about the performance of recommendation task. However, the key observation is that though cross-domain recommendation and transfer learning have single objective for the target domain, their loss functions still contain constraint terms for measuring data distribution in the source domain or similarity between two domains. In our proposed MKR, the KGE task serves as the constraint term *explicitly* to provide regularization for recommender systems. We would like to emphasize that the major contribution of this paper is exactly modeling the problem as multi-task learning: We go a step further than cross-domain recommendation and transfer learning by finding that the inter-task similarity is helpful to not only recommender systems but also knowledge graph embedding, as shown in theoretical analysis and experiment results.

[5]KGE task can also benefit from recommendation task empirically as shown in the experiments section.

2 OUR APPROACH

In this section, we first formulate the knowledge graph enhanced recommendation problem, then introduce the framework of MKR and present the design of the cross&compress unit, recommendation module and KGE module in detail. We lastly discuss the learning algorithm for MKR.

2.1 Problem Formulation

We formulate the knowledge graph enhanced recommendation problem in this paper as follows. In a typical recommendation scenario, we have a set of M users $\mathcal{U} = \{u_1, u_2, ..., u_M\}$ and a set of N items $\mathcal{V} = \{v_1, v_2, ..., v_N\}$. The user-item interaction matrix $\mathbf{Y} \in \mathbb{R}^{M \times N}$ is defined according to users' implicit feedback, where $y_{uv} = 1$ indicates that user u engaged with item v, such as behaviors of clicking, watching, browsing, or purchasing; otherwise $y_{uv} = 0$. Additionally, we also have access to a knowledge graph $\mathcal{G}$, which is comprised of entity-relation-entity triples (h, r, t). Here h, r, and t denote the head, relation, and tail of a knowledge triple, respectively. For example, the triple (*Quentin Tarantino, film.director.film, Pulp Fiction*) states the fact that Quentin Tarantino directs the film Pulp Fiction. In many recommendation scenarios, an item $v \in \mathcal{V}$ may associate with one or more entities in $\mathcal{G}$. For example, in movie recommendation, the item "Pulp Fiction" is linked with its namesake in a KG, while in news recommendation, news with the title "Trump pledges aid to Silicon Valley during tech meeting" is linked with entities "Donald Trump" and "Silicon Valley" in a KG.

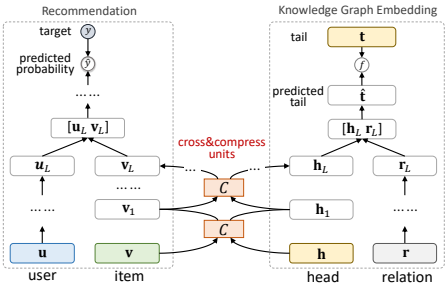
Given the user-item interaction matrix $\mathbf{Y}$ as well as the knowledge graph $\mathcal{G}$, we aim to predict whether user u has potential interest in item v with which he has had no interaction before. Our goal is to learn a prediction function $\hat{y}_{uv} = \mathcal{F}(u, v|\Theta, \mathbf{Y}, \mathcal{G})$, where $\hat{y}_{uv}$ denotes the probability that user u will engage with item v, and Θ is the model parameters of function $\mathcal{F}$.

2.2 Framework

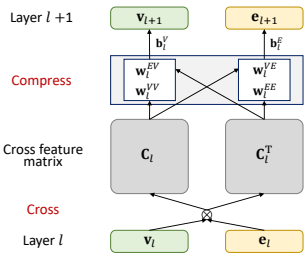
The framework of MKR is illustrated in Figure 1a. MKR consists of three main components: recommendation module, KGE module, and cross&compress units. (1) The recommendation module on the left takes a user and an item as input, and uses a multi-layer perceptron (MLP) and cross&compress units to extract short and dense features for the user and the item, respectively. The extracted features are then fed into another MLP together to output the predicted probability. (2) Similar to the left part, the KGE module in the right part also uses multiple layers to extract features from the head and relation of a knowledge triple, and outputs the representation of the predicted tail under the supervision of a score function f and the real tail. (3) The recommendation module and the KGE module are bridged by specially designed cross&compress units. The proposed unit can automatically learn high-order feature interactions of items in recommender systems and entities in the KG.

2.3 Cross&compress Unit

To model feature interactions between items and entities, we design a cross&compress unit in MKR framework. As shown in Figure 1b,



(a) Framework of MKR



(b) Cross&compress unit

Figure 1: (a) The framework of MKR. The left and right part illustrate the recommendation module and the KGE module, respectively, which are bridged by the cross&compress units. (b) Illustration of a cross&compress unit. The cross&compress unit generates a cross feature matrix from item and entity vectors by *cross* operation, and outputs their vectors for the next layer by *compress* operation.

for item v and one of its associated entities e, we first construct $d \times d$ pairwise interactions of their latent feature $\mathbf{v}_l \in \mathbb{R}^d$ and $\mathbf{e}_l \in \mathbb{R}^d$ from layer l:

$$\mathbf{C}_l = \mathbf{v}_l \mathbf{e}_l^\top = \begin{bmatrix} v_l^{(1)} e_l^{(1)} & \cdots & v_l^{(1)} e_l^{(d)} \\ \cdots & & \cdots \\ v_l^{(d)} e_l^{(1)} & \cdots & v_l^{(d)} e_l^{(d)} \end{bmatrix}, \tag{1}$$

where $\mathbf{C}_l \in \mathbb{R}^{d \times d}$ is the cross feature matrix of layer l, and d is the dimension of hidden layers. This is called the *cross* operation, since each possible feature interaction $v_l^{(i)} e_l^{(j)}, \forall (i,j) \in \{1, ..., d\}^2$ between item v and its associated entity e is modeled explicitly in the cross feature matrix. We then output the feature vectors of items and entities for the next layer by projecting the cross feature matrix into their latent representation spaces:

$$\begin{aligned} \mathbf{v}_{l+1} &= \mathbf{C}_l \mathbf{w}_l^{VV} + \mathbf{C}_l^\top \mathbf{w}_l^{EV} + \mathbf{b}_l^V = \mathbf{v}_l \mathbf{e}_l^\top \mathbf{w}_l^{VV} + \mathbf{e}_l \mathbf{v}_l^\top \mathbf{w}_l^{EV} + \mathbf{b}_l^V, \\ \mathbf{e}_{l+1} &= \mathbf{C}_l \mathbf{w}_l^{VE} + \mathbf{C}_l^\top \mathbf{w}_l^{EE} + \mathbf{b}_l^E = \mathbf{v}_l \mathbf{e}_l^\top \mathbf{w}_l^{VE} + \mathbf{e}_l \mathbf{v}_l^\top \mathbf{w}_l^{EE} + \mathbf{b}_l^E, \end{aligned} \tag{2}$$

where $\mathbf{w}_l^{\cdot\cdot} \in \mathbb{R}^d$ and $\mathbf{b}_l^{\cdot} \in \mathbb{R}^d$ are trainable weight and bias vectors. This is called the *compress* operation, since the weight vectors project the cross feature matrix from $\mathbb{R}^{d \times d}$ space back to the feature spaces $\mathbb{R}^d$. Note that in Eq. (2), the cross feature matrix is compressed along both horizontal and vertical directions (by operating on $\mathbf{C}_l$ and $\mathbf{C}_l^\top$) for the sake of symmetry, but we will provide more insights of the design in Section 3.2. For simplicity, the cross&compress unit is denoted as:

$$[\mathbf{v}_{l+1}, \mathbf{e}_{l+1}] = C(\mathbf{v}_l, \mathbf{e}_l), \tag{3}$$

and we use a suffix $[\mathbf{v}]$ or $[\mathbf{e}]$ to distinguish its two outputs in the following of this paper. Through cross&compress units, MKR can adaptively adjust the weights of knowledge transfer and learn the relevance between the two tasks.

It should be noted that cross&compress units should only exist in low-level layers of MKR, as shown in Figure 1a. This is because: (1) In deep architectures, features usually transform from general to specific along the network, and feature transferability drops significantly in higher layers with increasing task dissimilarity [38]. Therefore, sharing high-level layers risks to possible negative transfer, especially for the heterogeneous tasks in MKR. (2) In high-level layers of MKR, item features are mixed with user features, and entity features are mixed with relation features. The mixed features are not suitable for sharing since they have no explicit association.

2.4 Recommendation Module

The input of the recommendation module in MKR consists of two raw feature vectors $\mathbf{u}$ and $\mathbf{v}$ that describe user u and item v, respectively. $\mathbf{u}$ and $\mathbf{v}$ can be customized as one-hot ID [8], attributes [30], bag-of-words [29], or their combinations, based on the application scenario. Given user u's raw feature vector $\mathbf{u}$, we use an L-layer MLP to extract his latent condensed feature[6]:

$$\mathbf{u}_L = \mathcal{M}(\mathcal{M}(\cdots \mathcal{M}(\mathbf{u}))) = \mathcal{M}^L(\mathbf{u}), \tag{4}$$

where $\mathcal{M}(\mathbf{x}) = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$ is a fully-connected neural network layer[7] with weight $\mathbf{W}$, bias $\mathbf{b}$, and nonlinear activation function $\sigma(\cdot)$. For item v, we use L cross&compress units to extract its feature:

$$\mathbf{v}_L = \mathbb{E}_{e \sim \mathcal{S}(v)} \left[C^L(\mathbf{v}, \mathbf{e})[\mathbf{v}] \right], \tag{5}$$

where $\mathcal{S}(v)$ is the set of associated entities of item v.

After having user u's latent feature $\mathbf{u}_L$ and item v's latent feature $\mathbf{v}_L$, we combine the two pathways by a predicting function f_{RS}, for example, inner product or an H-layer MLP. The final predicted probability of user u engaging item v is:

$$\hat{y}_{uv} = \sigma\left(f_{RS}(\mathbf{u}_L, \mathbf{v}_L)\right). \tag{6}$$

[6] We use the exponent notation L in Eq. (4) and following equations in the rest of this paper for simplicity, but note that the parameters of L layers are actually different.
[7] Exploring a more elaborate design of layers in the recommendation module is an important direction of future work.

2.5 Knowledge Graph Embedding Module

Knowledge graph embedding is to embed entities and relations into continuous vector spaces while preserving their structure. Recently, researchers have proposed a great many KGE methods, including translational distance models [2, 13] and semantic matching models [14, 19]. In MKR, we propose a deep semantic matching architecture for KGE module. Similar to the recommendation module, for a given knowledge triple (h, r, t), we first utilize multiple cross&compress units and nonlinear layers to process the raw feature vectors of head h and relation r (including ID [13], types [36], textual description [35], etc.), respectively. Their latent features are then concatenated together, followed by a K-layer MLP for predicting tail t:

$$\begin{aligned} \mathbf{h}_L &= \mathbb{E}_{v\sim\mathcal{S}(h)}\left[C^L(\mathbf{v},\mathbf{h})[\mathbf{e}]\right], \\ \mathbf{r}_L &= \mathcal{M}^L(\mathbf{r}), \\ \hat{\mathbf{t}} &= \mathcal{M}^K\left(\begin{bmatrix}\mathbf{h}_L\\ \mathbf{r}_L\end{bmatrix}\right), \end{aligned} \tag{7}$$

where $\mathcal{S}(h)$ is the set of associated items of entity h, and $\hat{\mathbf{t}}$ is the predicted vector of tail t. Finally, the score of the triple (h, r, t) is calculated using a score (similarity) function f_{KG}:

$$score(h, r, t) = f_{KG}(\mathbf{t}, \hat{\mathbf{t}}), \tag{8}$$

where $\mathbf{t}$ is the real feature vector of t. In this paper, we use the normalized inner product $f_{KG}(\mathbf{t}, \hat{\mathbf{t}}) = \sigma(\mathbf{t}^\top \hat{\mathbf{t}})$ as the choice of score function [18], but other forms of (dis)similarity metrics can also be applied here such as KullbackâĂŞLeibler divergence.

2.6 Learning Algorithm

The complete loss function of MKR is as follows:

$$\begin{aligned} \mathcal{L} =& \mathcal{L}_{RS} + \mathcal{L}_{KG} + \mathcal{L}_{REG} \\ =& \sum_{u\in\mathcal{U}, v\in\mathcal{V}} \mathcal{J}(\hat{y}_{uv}, y_{uv}) \\ & - \lambda_1\Big(\sum_{(h,r,t)\in\mathcal{G}} score(h, r, t) - \sum_{(h',r,t')\notin\mathcal{G}} score(h', r, t')\Big) \\ & + \lambda_2\|\mathbf{W}\|_2^2. \end{aligned} \tag{9}$$

In Eq. (9), the first term measures loss in the recommendation module, where u and v traverse the set of users and the items, respectively, and $\mathcal{J}$ is the cross-entropy function. The second term calculates the loss in the KGE module, in which we aim to increase the score for all true triples while reducing the score for all false triples. The last item is the regularization term for preventing overfitting, λ_1 and λ_2 are the balancing parameters.[8]

Note that the loss function in Eq. (9) traverses all possible user-item pairs and knowledge triples. To make computation more efficient, following [17], we use a negative sampling strategy during training. The learning algorithm of MKR is presented in Algorithm 1, in which a training epoch consists of two stages: recommendation task (line 3-7) and KGE task (line 8-10). In each iteration, we repeat training on recommendation task for t times (t is a hyper-parameter and normally $t > 1$) before training on KGE task once in each epoch, since we are more focused on improving recommendation performance. We will discuss the choice of t in the experiments section.

[8] λ_1 can be seen as the ratio of two learning rates for the two tasks.

Algorithm 1 Multi-Task Training for MKR

Input: Interaction matrix $\mathbf{Y}$, knowledge graph $\mathcal{G}$
Output: Prediction function $\mathcal{F}(u, v|\Theta, \mathbf{Y}, \mathcal{G})$

1: Initialize all parameters
2: **for** number of training iteration **do**
   // *recommendation task*
3: **for** t steps **do**
4: Sample minibatch of positive and negative interactions from $\mathbf{Y}$;
5: Sample $e \sim \mathcal{S}(v)$ for each item v in the minibatch;
6: Update parameters of $\mathcal{F}$ by gradient descent on Eq. (1)-(6), (9);
7: **end for**
   // *knowledge graph embedding task*
8: Sample minibatch of true and false triples from $\mathcal{G}$;
9: Sample $v \sim \mathcal{S}(h)$ for each head h in the minibatch;
10: Update parameters of $\mathcal{F}$ by gradient descent on Eq. (1)-(3), (7)-(9);
11: **end for**

3 THEORETICAL ANALYSIS

In this section, we prove that cross&compress units have sufficient capability of polynomial approximation. We also show that MKR is a generalized framework over several representative methods of recommender systems and multi-task learning.

3.1 Polynomial Approximation

According to the Weierstrass approximation theorem [25], any function under certain smoothness assumption can be approximated by a polynomial to an arbitrary accuracy. Therefore, we examine the ability of high-order interaction approximation of the cross&compress unit. We show that cross&compress units can model the order of item-entity feature interaction up to exponential degree:

Theorem 1. *Denote the input of item and entity in MKR network as* $\mathbf{v} = [v_1 \cdots v_d]^\top$ *and* $\mathbf{e} = [e_1 \cdots e_d]^\top$*, respectively. Then the cross terms about* $\mathbf{v}$ *and* $\mathbf{e}$ *in* $\|\mathbf{v}_L\|_1$ *and* $\|\mathbf{e}_L\|_1$ *(the L1-norm of* $\mathbf{v}_L$ *and* $\mathbf{e}_L$*) with maximal degree is* $k_{\boldsymbol{\alpha},\boldsymbol{\beta}} v_1^{\alpha_1}\cdots v_d^{\alpha_d} e_1^{\beta_1}\cdots e_d^{\beta_d}$*, where* $k_{\boldsymbol{\alpha},\boldsymbol{\beta}} \in \mathbb{R}$*,* $\alpha_i, \beta_i \in \mathbb{N}$ *for* $i \in \{1, \cdots, d\}$*,* $\alpha_1 + \cdots + \alpha_d = 2^{L-1}$*, and* $\beta_1 + \cdots + \beta_d = 2^{L-1}$ *(*$L \geq 1, \mathbf{v}_0 = \mathbf{v}, \mathbf{e}_0 = \mathbf{e}$*).*

In recommender systems, $\prod_{i=1}^{d} v_i^{\alpha_i} e_i^{\beta_i}$ is also called *combinatorial* feature, as it measures the interactions of multiple original features. Theorem 1 states that cross&compress units can automatically model the combinatorial features of items and entities for sufficiently high order, which demonstrates the superior approximation capacity of MKR as compared with existing work such as Wide&Deep [3], factorization machines [22, 23] and DCN [34]. The proof of Theorem 1 is provided in the Appendix. Note that Theorem 1 gives a theoretical view of the polynomial approximation

ability of the cross&compress unit rather than providing guarantees on its actual performance. We will empirically evaluate the cross&compress unit in the experiments section.

3.2 Unified View of Representative Methods

In the following we provide a unified view of several representative models in recommender systems and multi-task learning, by showing that they are restricted versions of or theoretically related to MKR. This justifies the design of cross&compress unit and conceptually explains its strong empirical performance as compared to baselines.

3.2.1 Factorization machines. Factorization machines [22, 23] are a generic method for recommender systems. Given an input feature vector, FMs model all interactions between variables in the input vector using factorized parameters, thus being able to estimate interactions in problems with huge sparsity such as recommender systems. The model equation for a 2-degree factorization machine is defined as

$$\hat{y}(\mathbf{x}) = w_0 + \sum_{i=1}^{d} w_i x_i + \sum_{i=1}^{d} \sum_{j=i+1}^{d} \langle \mathbf{v}_i, \mathbf{v}_j \rangle x_i x_j, \quad (10)$$

where x_i is the i-th unit of input vector $\mathbf{x}$, $w_\cdot$ is weight scalar, $\mathbf{v}_\cdot$ is weight vector, and $\langle \cdot, \cdot \rangle$ is dot product of two vectors. We show that the essence of FM is conceptually similar to an 1-layer cross&compress unit:

Proposition 1. *The L1-norm of $\mathbf{v}_1$ and $\mathbf{e}_1$ can be written as the following form:*

$$\|\mathbf{v}_1\|_1 \,(or\ \|\mathbf{e}_1\|_1) = \left| b + \sum_{i=1}^{d} \sum_{j=1}^{d} \langle w_i, w_j \rangle v_i e_j \right|, \quad (11)$$

where $\langle w_i, w_j \rangle = w_i + w_j$ is the sum of two scalars.

It is interesting to notice that, instead of factorizing the weight parameter of $x_i x_j$ into the dot product of two vectors as in FM, the weight of term $v_i e_j$ is factorized into the sum of two scalars in cross&compress unit to reduce the number of parameters and increase robustness of the model.

3.2.2 Deep&Cross Network. DCN [34] learns explicit and high-order cross features by introducing the layers:

$$\mathbf{x}_{l+1} = \mathbf{x}_0 \mathbf{x}_l^\top \mathbf{w}_l + \mathbf{x}_l + \mathbf{b}_l, \quad (12)$$

where $\mathbf{x}_l$, $\mathbf{w}_l$, and $\mathbf{b}_l$ are representation, weight, and bias of the l-th layer. We demonstrate the link between DCN and MKR by the following proposition:

Proposition 2. *In the formula of $\mathbf{v}_{l+1}$ in Eq. (2), if we restrict $\mathbf{w}_l^{VV}$ in the first term to satisfy $\mathbf{e}_l^\top \mathbf{w}_l^{VV} = 1$ and restrict $\mathbf{e}_l$ in the second term to be $\mathbf{e}_0$ (and impose similar restrictions on $\mathbf{e}_{l+1}$), the cross&compress unit is then conceptually equivalent to DCN layer in the sense of multi-task learning:*

$$\begin{aligned} \mathbf{v}_{l+1} &= \mathbf{e}_0 \mathbf{v}_l^\top \mathbf{w}_l^{EV} + \mathbf{v}_l + \mathbf{b}_l^V, \\ \mathbf{e}_{l+1} &= \mathbf{v}_0 \mathbf{e}_l^\top \mathbf{w}_l^{VE} + \mathbf{e}_l + \mathbf{b}_l^E. \end{aligned} \quad (13)$$

It can be proven that the polynomial approximation ability of the above DCN-equivalent version (i.e., the maximal degree of cross terms in $\mathbf{v}_l$ and $\mathbf{e}_l$) is $O(l)$, which is weaker than original cross&compress units with $O(2^l)$ approximation ability.

3.2.3 Cross-stitch Networks. Cross-stitch networks [18] is a multi-task learning model in convolutional networks, in which the designed cross-stitch unit can learn a combination of shared and task-specific representations between two tasks. Specifically, given two activation maps x_A and x_B from layer l for both the tasks, cross-stitch networks learn linear combinations $\tilde{x}_A$ and $\tilde{x}_B$ of both the input activations and feed these combinations as input to the next layers' filters. The formula at location (i, j) in the activation map is

$$\begin{bmatrix} \tilde{x}_A^{ij} \\ \tilde{x}_B^{ij} \end{bmatrix} = \begin{bmatrix} \alpha_{AA} & \alpha_{AB} \\ \alpha_{BA} & \alpha_{BB} \end{bmatrix} \begin{bmatrix} x_A^{ij} \\ x_B^{ij} \end{bmatrix}, \quad (14)$$

where α's are trainable transfer weights of representations between task A and task B. We show that the cross-stitch unit in Eq. (14) is a simplified version of our cross&compress unit by the following proposition:

Proposition 3. *If we omit all biases in Eq. (2), the cross&compress unit can be written as*

$$\begin{bmatrix} \mathbf{v}_{l+1} \\ \mathbf{e}_{l+1} \end{bmatrix} = \begin{bmatrix} \mathbf{e}_l^\top \mathbf{w}_l^{VV} & \mathbf{v}_l^\top \mathbf{w}_l^{EV} \\ \mathbf{e}_l^\top \mathbf{w}_l^{VE} & \mathbf{v}_l^\top \mathbf{w}_l^{EE} \end{bmatrix} \begin{bmatrix} \mathbf{v}_l \\ \mathbf{e}_l \end{bmatrix}. \quad (15)$$

The transfer matrix in Eq. (15) serves as the cross-stitch unit $[\alpha_{AA}\ \alpha_{AB};\ \alpha_{BA}\ \alpha_{BB}]$ in Eq. (14). Like cross-stitch networks, MKR network can decide to make certain layers task specific by setting $\mathbf{v}_l^\top \mathbf{w}_l^{EV}$ (α_{AB}) or $\mathbf{e}_l^\top \mathbf{w}_l^{VE}$ (α_{BA}) to zero, or choose a more shared representation by assigning a higher value to them. But the transfer matrix is more fine-grained in cross&compress unit, because the transfer weights are replaced from scalars to dot products of two vectors. It is rather interesting to notice that Eq. (15) can also be regarded as an *attention mechanism* [1], as the computation of transfer weights involves the feature vectors $\mathbf{v}_l$ and $\mathbf{e}_l$ themselves.

4 EXPERIMENTS

In this section, we evaluate the performance of MKR in four real-world recommendation scenarios: movie, book, music, and news[9].

4.1 Datasets

We utilize the following four datasets in our experiments:

- **MovieLens-1M**[10] is a widely used benchmark dataset in movie recommendations, which consists of approximately 1 million explicit ratings (ranging from 1 to 5) on the MovieLens website.
- **Book-Crossing**[11] dataset contains 1,149,780 explicit ratings (ranging from 0 to 10) of books in the Book-Crossing community.
- **Last.FM**[12] dataset contains musician listening information from a set of 2 thousand users from Last.fm online music system.
- **Bing-News** dataset contains 1,025,192 pieces of implicit feedback collected from the server logs of Bing News[13] from

[9] The source code is available at https://github.com/hwwang55/MKR.
[10] https://grouplens.org/datasets/movielens/1m/
[11] http://www2.informatik.uni-freiburg.de/~cziegler/BX/
[12] https://grouplens.org/datasets/hetrec-2011/
[13] https://www.bing.com/news

Table 1: Basic statistics and hyper-parameter settings for the four datasets.

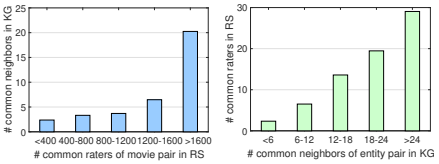
Dataset	# users	# items	# interactions	# KG triples	Hyper-parameters
MovieLens-1M	6,036	2,347	753,772	20,195	$L = 1, d = 8, t = 3, \lambda_1 = 0.5$
Book-Crossing	17,860	14,910	139,746	19,793	$L = 1, d = 8, t = 2, \lambda_1 = 0.1$
Last.FM	1,872	3,846	42,346	15,518	$L = 2, d = 4, t = 2, \lambda_1 = 0.1$
Bing-News	141,487	535,145	1,025,192	1,545,217	$L = 3, d = 16, t = 5, \lambda_1 = 0.2$

October 16, 2016 to August 11, 2017. Each piece of news has a title and a snippet.

Since MovieLens-1M, Book-Crossing, and Last.FM are explicit feedback data (Last.FM provides the listening count as weight for each user-item interaction), we transform them into implicit feedback where each entry is marked with 1 indicating that the user has rated the item positively, and sample an unwatched set marked as 0 for each user. The threshold of positive rating is 4 for MovieLens-1M, while no threshold is set for Book-Crossing and Last.FM due to their sparsity.

We use Microsoft Satori to construct the KG for each dataset. We first select a subset of triples from the whole KG with a confidence level greater than 0.9. For MovieLens-1M and Book-Crossing, we additionally select a subset of triples from the sub-KG whose relation name contains "film" or "book" respectively to further reduce KG size.

Given the sub-KGs, for MovieLens-1M, Book-Crossing, and Last.FM, we collect IDs of all valid movies, books, or musicians by matching their names with tail of triples (*head, film.film.name, tail*), *(head, book.book.title, tail)*, or *(head, type.object.name, tail)*, respectively. For simplicity, items with no matched or multiple matched entities are excluded. We then match the IDs with the head and tail of all KG triples and select all well-matched triples from the sub-KG. The constructing process is similar for Bing-News except that: (1) we use entity linking tools to extract entities in news titles; (2) we do not impose restrictions on the names of relations since the entities in news titles are not within one particular domain. The basic statistics of the four datasets are presented in Table 1. Note that the number of users, items, and interactions are smaller than original datasets since we filtered out items with no corresponding entity in the KG.



(a) RS to KG **(b) KG to RS**

Figure 2: The correlation between the number of common neighbors of an item pair in KG and their number of common raters in RS.

4.2 Baselines

We compare our proposed MKR with the following baselines. Unless otherwise specified, the hyper-parameter settings of baselines are the same as reported in their original papers or as default in their codes.

- **PER** [39] treats the KG as heterogeneous information networks and extracts meta-path based features to represent the connectivity between users and items. In this paper, we use manually designed user-item-attribute-item paths as features, i.e., "user-movie-director-movie", "user-movie-genre-movie", and "user-movie-star-movie" for MovieLens-20M; "user-book-author-book" and "user-book-genre-book" for Book-Crossing; "user-musician-genre-musician", "user-musician-country-musician", and "user-musician-age-musician" (age is discretized) for Last.FM. Note that PER cannot be applied to news recommendation because it's hard to pre-define meta-paths for entities in news.
- **CKE** [40] combines CF with structural, textual, and visual knowledge in a unified framework for recommendation. We implement CKE as CF plus structural knowledge module in this paper. The dimension of user and item embeddings for the four datasets are set as 64, 128, 32, 64, respectively. The dimension of entity embeddings is 32.
- **DKN** [32] treats entity embedding and word embedding as multiple channels and combines them together in CNN for CTR prediction. In this paper, we use movie/book names and news titles as textual input for DKN. The dimension of word embedding and entity embedding is 64, and the number of filters is 128 for each window size 1, 2, 3.
- **RippleNet** [31] is a memory-network-like approach that propagates usersâĂŹ preferences on the knowledge graph for recommendation. The hyper-parameter settings for Last.FM are $d = 8$, $H = 2$, $\lambda_1 = 10^{-6}$, $\lambda_2 = 0.01$, $\eta = 0.02$.
- **LibFM** [23] is a widely used feature-based factorization model. We concatenate the raw features of users and items as well as the corresponding averaged entity embeddings learned from TransR [13] as input for LibFM. The dimension is {1, 1, 8} and the number of training epochs is 50. The dimension of TransR is 32.
- **Wide&Deep** [3] is a deep recommendation model combining a (wide) linear channel with a (deep) nonlinear channel. The input for Wide&Deep is the same as in LibFM. The dimension of user, item, and entity is 64, and we use a two-layer deep channel with dimension of 100 and 50 as well as a wide channel.

Table 2: The results of *AUC* and *Accuracy* in CTR prediction.

Model	MovieLens-1M		Book-Crossing		Last.FM		Bing-News	
	AUC	*ACC*	*AUC*	*ACC*	*AUC*	*ACC*	*AUC*	*ACC*
PER	0.710 (-22.6%)	0.664 (-21.2%)	0.623 (-15.1%)	0.588 (-16.7%)	0.633 (-20.6%)	0.596 (-20.7%)	-	-
CKE	0.801 (-12.6%)	0.742 (-12.0%)	0.671 (-8.6%)	0.633 (-10.3%)	0.744 (-6.6%)	0.673 (-10.5%)	0.553 (-19.7%)	0.516 (-20.0%)
DKN	0.655 (-28.6%)	0.589 (-30.1%)	0.622 (-15.3%)	0.598 (-15.3%)	0.602 (-24.5%)	0.581 (-22.7%)	0.667 (-3.2%)	0.610 (-5.4%)
RippleNet	**0.920** (+0.3%)	0.842 (-0.1%)	0.729 (-0.7%)	0.662 (-6.2%)	0.768 (-3.6%)	0.691 (-8.1%)	0.678 (-1.6%)	0.630 (-2.3%)
LibFM	0.892 (-2.7%)	0.812 (-3.7%)	0.685 (-6.7%)	0.640 (-9.3%)	0.777 (-2.5%)	0.709 (-5.7%)	0.640 (-7.1%)	0.591 (-8.4%)
Wide&Deep	0.898 (-2.1%)	0.820 (-2.7%)	0.712 (-3.0%)	0.624 (-11.6%)	0.756 (-5.1%)	0.688 (-8.5%)	0.651 (-5.5%)	0.597 (-7.4%)
MKR	0.917	**0.843**	**0.734**	**0.704**	**0.797**	**0.752**	**0.689**	**0.645**
MKR-1L	-	-	-	-	0.795 (-0.3%)	0.749 (-0.4%)	0.680 (-1.3%)	0.631 (-2.2%)
MKR-DCN	0.883 (-3.7%)	0.802 (-4.9%)	0.705 (-4.3%)	0.676 (-4.2%)	0.778 (-2.4%)	0.730 (-2.9%)	0.671 (-2.6%)	0.614 (-4.8%)
MKR-stitch	0.905 (-1.3%)	0.830 (-1.5%)	0.721 (-2.2%)	0.682 (-3.4%)	0.772 (-3.1%)	0.725 (-3.6%)	0.674 (-2.2%)	0.621 (-3.7%)

4.3 Experiments setup

In MKR, we set the number of high-level layers $K = 1$, f_{RS} as inner product, and $\lambda_2 = 10^{-6}$ for all three datasets, and other hyper-parameter are given in Table 1. The settings of hyper-parameters are determined by optimizing *AUC* on a validation set. For each dataset, the ratio of training, validation, and test set is $6:2:2$. Each experiment is repeated 3 times, and the average performance is reported. We evaluate our method in two experiment scenarios: (1) In click-through rate (CTR) prediction, we apply the trained model to each piece of interactions in the test set and output the predicted click probability. We use *AUC* and *Accuracy* to evaluate the performance of CTR prediction. (2) In top-K recommendation, we use the trained model to select K items with highest predicted click probability for each user in the test set, and choose *Precision@K* and *Recall@K* to evaluate the recommended sets.

4.4 Empirical study

We conduct an empirical study to investigate the correlation of items in RS and their corresponding entities in KG. Specifically, we aim to reveal how the number of common neighbors of an item pair in KG changes with their number of common raters in RS. To this end, we first randomly sample 1 million item pairs from MovieLens-1M. We then classify each pair into 5 categories based on the number of their common raters in RS, and count their average number of common neighbors in KG for each category. The result is presented in Figure 2a, which clearly shows that *if two items have more common raters in RS, they are likely to share more common neighbors in KG*. Figure 2b shows the positive correlation from an opposite direction. The above findings empirically demonstrate that *items share the similar structure of proximity in KG and RS*, thus the cross knowledge transfer of items benefits both recommendation and KGE tasks in MKR.

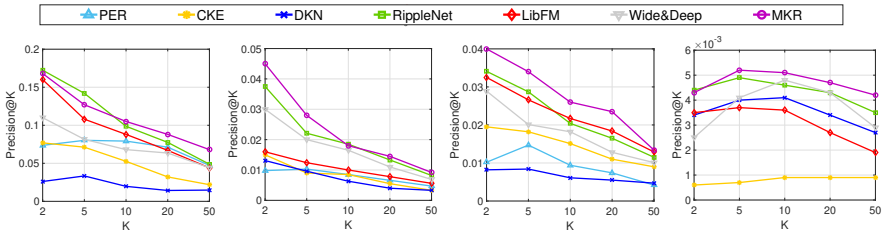
4.5 Results

4.5.1 Comparison with baselines. The results of all methods in CTR prediction and top-K recommendation are presented in Table 2 and Figure 3, 4, respectively. We have the following observations:

- PER performs poor on movie, book, and music recommendation because the user-defined meta-paths can hardly be optimal in reality. Moreover, PER cannot be applied to news recommendation.
- CKE performs better in movie, book, and music recommendation than news. This may be because MovieLens-1M, Book-Crossing, and Last.FM are much denser than Bing-News, which is more favorable for the collaborative filtering part in CKE.
- DKN performs best in news recommendation compared with other baselines, but performs worst in other scenarios. This is because movie, book, and musician names are too short and ambiguous to provide useful information.
- RippleNet performs best among all baselines, and even outperforms MKR on MovieLens-1M. This demonstrates that RippleNet can precisely capture user interests, especially in the case where user-item interactions are dense. However, RippleNet is more sensitive to the density of datasets, as it performs worse than MKR in Book-Crossing, Last.FM, and Bing-News. We will further study their performance in sparse scenarios in Section 4.5.3.
- In general, our MKR performs best among all methods on the four datasets. Specifically, MKR achieves average *Accuracy* gains of 11.6%, 11.5%, 12.7%, and 8.7% in movie, book, music, and news recommendation, respectively, which demonstrates the efficacy of the multi-task learning framework in MKR. Note that the top-K metrics are much lower for Bing-News because the number of news is significantly larger than movies, books, and musicians.

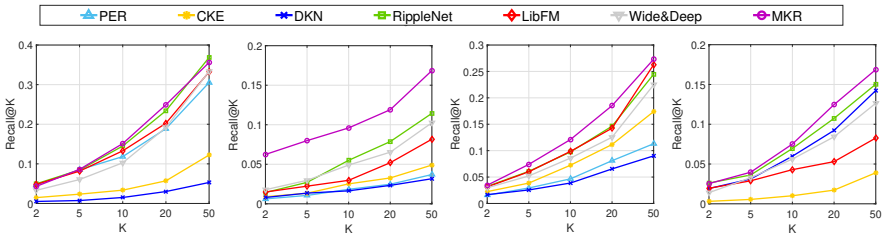
4.5.2 Comparison with MKR variants. We further compare MKR with its three variants to demonstrate the efficacy of cross&compress unit:

- MKR-1L is MKR with one layer of cross&compress unit, which corresponds to FM model according to Proposition 1. Note that MKR-1L is actually MKR in the experiments for MovieLens-1M.
- MKR-DCN is a variant of MKR based on Eq. (13), which corresponds to DCN model.
- MKR-stitch is another variant of MKR corresponding to the cross-stitch network, in which the transfer weights in Eq. (15) are replaced by four trainable scalars.



(a) MovieLens-1M (b) Book-Crossing (c) Last.FM (d) Bing-News

Figure 3: The results of *Precision@K* in top-*K* recommendation.



(a) MovieLens-1M (b) Book-Crossing (c) Last.FM (d) Bing-News

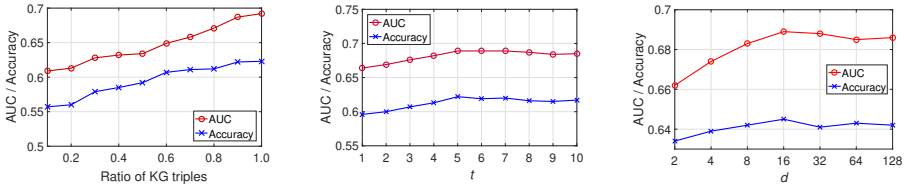
Figure 4: The results of *Recall@K* in top-*K* recommendation.

Table 3: Results of *AUC* on MovieLens-1M in CTR prediction with different ratios of training set *r*.

Model	*r*									
	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
PER	0.598	0.607	0.621	0.638	0.647	0.662	0.675	0.688	0.697	0.710
CKE	0.674	0.692	0.705	0.716	0.739	0.754	0.768	0.775	0.797	0.801
DKN	0.579	0.582	0.589	0.601	0.612	0.620	0.631	0.638	0.646	0.655
RippleNet	0.843	0.851	0.859	0.862	0.870	0.878	0.890	0.901	0.912	**0.920**
LibFM	0.801	0.810	0.816	0.829	0.837	0.850	0.864	0.875	0.886	0.892
Wide&Deep	0.788	0.802	0.809	0.815	0.821	0.840	0.858	0.876	0.884	0.898
MKR	**0.868**	**0.874**	**0.881**	**0.882**	**0.889**	**0.897**	**0.903**	**0.908**	**0.913**	0.917

From Table 2 we observe that MKR outperforms MKR-1L and MKR-DCN, which shows that modeling high-order interactions between item and entity features is helpful for maintaining decent performance. MKR also achieves better scores than MKR-stitch. This validates the efficacy of fine-grained control on knowledge transfer in MKR compared with the simple cross-stitch units.

4.5.3 Results in sparse scenarios. One major goal of using knowledge graph in MKR is to alleviate the sparsity and the cold start problem of recommender systems. To investigate the efficacy of the KGE module in sparse scenarios, we vary the ratio of training set of MovieLens-1M from 100% to 10% (while the validation and test set are kept fixed), and report the results of *AUC* in CTR prediction for all methods. The results are shown in Table 3. We observe that the performance of all methods deteriorates with the reduce of the training set. When $r = 10\%$, the *AUC* score decreases by 15.8%, 15.9%, 11.6%, 8.4%, 10.2%, 12.2% for PER, CKE, DKN, RippleNet, LibFM, and Wide&Deep, respectively, compared with the case when full training set is used ($r = 100\%$). In contrast, the



(a) KG size (b) Training frequency (c) Dimension of embeddings

Figure 5: Parameter sensitivity of MKR on Bing-News w.r.t. (a) the size of the knowledge graph; (b) training frequency of the RS module t; and (c) dimension of embeddings d.

Table 4: The results of *RMSE* on the KGE module for the three datasets. "KGE" means only KGE module is trained, while "KGE + RS" means KGE module and RS module are trained together.

dataset	KGE	KGE + RS
MovieLens-1M	0.319	**0.302**
Book-Crossing	0.596	**0.558**
Last.FM	0.480	**0.471**
Bing-News	0.488	**0.459**

AUC score of MKR only decreases by 5.3%, which demonstrates that MKR can still maintain a decent performance even when the user-item interaction is sparse. We also notice that MKR performs better than RippleNet in sparse scenarios, which is accordance with our observation in Section 4.5.1 that RippleNet is more sensitive to the density of user-item interactions.

4.5.4 Results on KGE side. Although the goal of MKR is to utilize KG to assist with recommendation, it is still interesting to investigate whether the RS task benefits the KGE task, since the principle of multi-task learning is to leverage shared information to help improve the performance of all tasks [42]. We present the result of *RMSE* (rooted mean square error) between predicted and real vectors of tails in the KGE task in Table 4. Fortunately, we find that the existence of RS module can indeed reduce the prediction error by 1.9% ∼ 6.4%. The results show that the cross&compress units are able to learn general and shared features that mutually benefit both sides of MKR.

4.6 Parameter Sensitivity

4.6.1 Impact of KG size. We vary the size of KG to further investigate the efficacy of usage of KG. The results of *AUC* on Bing-News are plotted in Figure 5a. Specifically, the *AUC* and *Accuracy* is enhanced by 13.6% and 11.8% with the KG ratio increasing from 0.1 to 1.0 in three scenarios, respectively. This is because the Bing-News dataset is extremely sparse, making the effect of KG usage rather obvious.

4.6.2 Impact of RS training frequency. We investigate the influence of parameters t in MKR by varying t from 1 to 10, while keeping other parameters fixed. The results are presented in Figure 5b. We observe that MKR achieves the best performance when $t = 5$. This is because a high training frequency of the KGE module will mislead the objective function of MKR, while too small of a training frequency of KGE cannot make full use of the transferred knowledge from the KG.

4.6.3 Impact of embedding dimension. We also show how the dimension of users, items, and entities affects the performance of MKR in Figure 5c. We find that the performance is initially improved with the increase of dimension, because more bits in embedding layer can encode more useful information. However, the performance drops when the dimension further increases, as too large number of dimensions may introduce noises which mislead the subsequent prediction.

5 RELATED WORK

5.1 Knowledge Graph Embedding

The KGE module in MKR connects to a large body of work in KGE methods. KGE is used to embed entities and relations in a knowledge into low-dimensional vector spaces while still preserving the structural information [33]. KGE methods can be classified into the following two categories: (1) Translational distance models exploit distance-based scoring functions when learning representations of entities and relations, such as TransE [2], TransH [35], and TransR [13]; (2) Semantic matching models measure plausibility of knowledge triples by matching latent semantics of entities and relations, such as RESCAL [20], ANALOGY [19], and HolE [14]. Recently, researchers also propose incorporating auxiliary information, such as entity types [36], logic rules [24], and textual descriptions [46] to assist KGE. The above KGE methods can also be incorporated into MKR as the implementation of the KGE module, but note that the cross&compress unit in MKR needs to be redesigned accordingly. Exploring other designs of KGE module as well as the corresponding bridging unit is also an important direction of future work.

5.2 Multi-Task Learning

Multi-task learning is a learning paradigm in machine learning and its aim is to leverage useful information contained in multiple related tasks to help improve the generalization performance of all the tasks [42]. All of the learning tasks are assumed to be related to

each other, and it is found that learning these tasks jointly can lead to performance improvement compared with learning them individually. In general, MTL algorithms can be classified into several categories, including feature learning approach [34, 41], low-rank approach [7, 16], task clustering approach [47], task relation learning approach [12], and decomposition approach [6]. For example, the cross-stitch network [41] determines the inputs of hidden layers in different tasks by a knowledge transfer matrix; Zhou et. al [47] aims to cluster tasks by identifying representative tasks which are a subset of the given m tasks, i.e., if task T_i is selected by task T_j as a representative task, then it is expected that model parameters for T_j are similar to those of T_i. MTL can also be combined with other learning paradigms to improve the performance of learning tasks further, including semi-supervised learning, active learning, unsupervised learning,and reinforcement learning.

Our work can be seen as an asymmetric multi-task learning framework [37, 43, 44], in which we aim to utilize the connection between RS and KG to help improve their performance, and the two tasks are trained with different frequencies.

5.3 Deep Recommender Systems

Recently, deep learning has been revolutionizing recommender systems and achieves better performance in many recommendation scenarios. Roughly speaking, deep recommender systems can be classified into two categories: (1) Using deep neural networks to process the raw features of users or items [5, 28–30, 40]; For example, Collaborative Deep Learning [29] designs autoencoders to extract short and dense features from textual input and feeds the features into a collaborative filtering module; DeepFM [5] combines factorization machines for recommendation and deep learning for feature learning in a neural network architecture. (2) Using deep neural networks to model the interaction among users and items [3, 4, 8, 9]. For example, Neural Collaborative Filtering [8] replaces the inner product with a neural architecture to model the user-item interaction. The major difference between these methods and ours is that MKR deploys a multi-task learning framework that utilizes the knowledge from a KG to assist recommendation.

6 CONCLUSIONS AND FUTURE WORK

This paper proposes MKR, a multi-task learning approach for knowledge graph enhanced recommendation. MKR is a deep and end-to-end framework that consists of two parts: the recommendation module and the KGE module. Both modules adopt multiple nonlinear layers to extract latent features from inputs and fit the complicated interactions of user-item and head-relation pairs. Since the two tasks are not independent but connected by items and entities, we design a cross&compress unit in MKR to associate the two tasks, which can automatically learn high-order interactions of item and entity features and transfer knowledge between the two tasks. We conduct extensive experiments in four recommendation scenarios. The results demonstrate the significant superiority of MKR over strong baselines and the efficacy of the usage of KG.

For future work, we plan to investigate other types of neural networks (such as CNN) in MKR framework. We will also incorporate other KGE methods as the implementation of KGE module in MKR by redesigning the cross&compress unit.

APPENDIX

A Proof of Theorem 1

Proof. We prove the theorem by induction:

Base case: When $l = 1$,

$$\begin{aligned}\mathbf{v}_1 =& \mathbf{v}\mathbf{e}^\top \mathbf{w}_0^{VV} + \mathbf{e}\mathbf{v}^\top \mathbf{w}_0^{EV} + \mathbf{b}_0^V \\ =& \left[v_1 \sum_{i=1}^{d} e_i w_0^{VV(i)} \ \cdots \ v_d \sum_{i=1}^{d} e_i w_0^{VV(i)} \right]^\top \\ &+ \left[e_1 \sum_{i=1}^{d} v_i w_0^{EV(i)} \ \cdots \ e_d \sum_{i=1}^{d} v_i w_0^{EV(i)} \right]^\top \\ &+ \left[b_0^{V(0)} \ \cdots \ b_0^{V(d)} \right]^\top .\end{aligned}$$

Therefore, we have

$$\begin{aligned}\|\mathbf{v}_1\|_1 &= \left| \sum_{j=1}^{d} v_j \sum_{i=1}^{d} e_i w_0^{VV(i)} + \sum_{j=1}^{d} e_j \sum_{i=1}^{d} v_i w_0^{EV(i)} + \sum_{i=1}^{d} b_0^{V(d)} \right| \\ &= \left| \sum_{i=1}^{d} \sum_{j=1}^{d} (w_0^{EV(i)} + w_0^{VV(j)}) v_i e_j + \sum_{i=1}^{d} b_0^{V(d)} \right| .\end{aligned}$$

It is clear that the cross terms about $\mathbf{v}$ and $\mathbf{e}$ with maximal degree is $k_{\boldsymbol{\alpha},\boldsymbol{\beta}} v_i e_j$, so we have $\alpha_1 + \cdots + \alpha_d = 1 = 2^{1-1}$, and $\beta_1 + \cdots + \beta_d = 1 = 2^{1-1}$ for $\mathbf{v}_1$. The proof for $\mathbf{e}_1$ is similar.

Induction step: Suppose $\alpha_1 + \cdots + \alpha_d = 2^{l-1}$ and $\beta_1 + \cdots + \beta_d = 2^{l-1}$ hold for the maximal-degree term x and y in $\|\mathbf{v}_l\|_1$ and $\|\mathbf{e}_l\|_1$. Since $\|\mathbf{v}_l\|_1 = \left| \sum_{i=1}^{d} v_l^{(i)} \right|$ and $\|\mathbf{e}_l\|_1 = \left| \sum_{i=1}^{d} e_l^{(i)} \right|$, without loss of generosity, we assume that x and y exist in $v_l^{(a)}$ and $e_l^{(b)}$, respectively. Then for $l + 1$, we have

$$\|\mathbf{v}_{l+1}\|_1 = \sum_{i=1}^{d} \sum_{j=1}^{d} (w_l^{EV(i)} + w_l^{VV(j)}) v_l^{(i)} e_l^{(j)} + \sum_{i=1}^{d} b_l^{V(d)} .$$

Obviously, the maximal-degree term in $\|\mathbf{v}_{l+1}\|_1$ is the cross term xy in $v_l^{(a)} e_l^{(b)}$. Since we have $\alpha_1 + \cdots + \alpha_d = 2^{l-1}$ and $\beta_1 + \cdots + \beta_d = 2^{l-1}$ for both x and y, the degree of cross term xy therefore satisfies $\alpha_1 + \cdots + \alpha_d = 2^{(l+1)-1}$ and $\beta_1 + \cdots + \beta_d = 2^{(l+1)-1}$. The proof for $\|\mathbf{e}_{l+1}\|_1$ is similar. □

B Proof of Proposition 1

Proof. In the proof of Theorem 1 in Appendix A, we have shown that

$$\|\mathbf{v}_1\|_1 = \left| \sum_{i=1}^{d} \sum_{j=1}^{d} (w_0^{EV(i)} + w_0^{VV(j)}) v_i e_j + \sum_{i=1}^{d} b_0^{V(d)} \right| .$$

It is easy to see that $w_i = w_0^{EV(i)}$, $w_j = w_0^{VV(j)}$, and $b = \sum_{i=1}^{d} b_0^{V(d)}$. The proof is similar for $\|\mathbf{e}_1\|_1$. □

We omit the proofs for Proposition 2 and Proposition 3 as they are straightforward.

REFERENCES

[1] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *Proceedings of the 3rd International Conference on Learning Representations*.

[2] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. 2013. Translating embeddings for modeling multi-relational data. In *Advances in Neural Information Processing Systems*. 2787–2795.

[3] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. 2016. Wide & deep learning for recommender systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*. ACM, 7–10.

[4] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems*. ACM, 191–198.

[5] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. 2017. DeepFM: A Factorization-Machine based Neural Network for CTR Prediction. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*.

[6] Lei Han and Yu Zhang. 2015. Learning tree structure in multi-task learning. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 397–406.

[7] Lei Han and Yu Zhang. 2016. Multi-Stage Multi-Task Learning with Reduced Rank.. In *AAAI*. 1638–1644.

[8] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. 2017. Neural collaborative filtering. In *Proceedings of the 26th International Conference on World Wide Web*. 173–182.

[9] Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry Heck. 2013. Learning deep structured semantic models for web search using clickthrough data. In *CIKM*. ACM, 2333–2338.

[10] Mohsen Jamali and Martin Ester. 2010. A matrix factorization technique with trust propagation for recommendation in social networks. In *Proceedings of the 4th ACM conference on Recommender systems*. ACM, 135–142.

[11] Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix factorization techniques for recommender systems. *Computer* 42, 8 (2009).

[12] Giwoong Lee, Eunho Yang, and Sung Hwang. 2016. Asymmetric multi-task learning based on task relatedness and loss. In *International Conference on Machine Learning*. 230–238.

[13] Yankai Lin, Zhiyuan Liu, Maosong Sun, Yang Liu, and Xuan Zhu. 2015. Learning Entity and Relation Embeddings for Knowledge Graph Completion.. In *The 29th AAAI Conference on Artificial Intelligence*. 2181–2187.

[14] Hanxiao Liu, Yuexin Wu, and Yiming Yang. 2017. Analogical Inference for Multi-Relational Embeddings. In *Proceedings of the 34th International Conference on Machine Learning*. 2168–2178.

[15] Mingsheng Long, Zhangjie Cao, Jianmin Wang, and S Yu Philip. 2017. Learning Multiple Tasks with Multilinear Relationship Networks. In *Advances in Neural Information Processing Systems*. 1593–1602.

[16] Andrew M McDonald, Massimiliano Pontil, and Dimitris Stamos. 2014. Spectral k-support norm regularization. In *Advances in Neural Information Processing Systems*. 3644–3652.

[17] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems*. 3111–3119.

[18] Ishan Misra, Abhinav Shrivastava, Abhinav Gupta, and Martial Hebert. 2016. Cross-stitch networks for multi-task learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 3994–4003.

[19] Maximilian Nickel, Lorenzo Rosasco, Tomaso A Poggio, et al. 2016. Holographic Embeddings of Knowledge Graphs.. In *The 30th AAAI Conference on Artificial Intelligence*. 1955–1961.

[20] Maximilian Nickel, Volker Tresp, and Hans-Peter Kriegel. 2011. A Three-Way Model for Collective Learning on Multi-Relational Data. In *Proceedings of the 28th International Conference on Machine Learning*. 809–816.

[21] Sinno Jialin Pan, Qiang Yang, et al. 2010. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering* 22, 10 (2010), 1345–1359.

[22] Steffen Rendle. 2010. Factorization machines. In *Proceedings of the 10th IEEE International Conference on Data Mining*. IEEE, 995–1000.

[23] Steffen Rendle. 2012. Factorization machines with libfm. *ACM Transactions on Intelligent Systems and Technology (TIST)* 3, 3 (2012), 57.

[24] Tim Rocktäschel, Sameer Singh, and Sebastian Riedel. 2015. Injecting logical background knowledge into embeddings for relation extraction. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 1119–1129.

[25] Walter Rudin et al. 1964. *Principles of mathematical analysis*. Vol. 3. McGraw-hill New York.

[26] Jie Tang, Sen Wu, Jimeng Sun, and Hang Su. 2012. Cross-domain collaboration recommendation. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 1285–1293.

[27] Hongwei Wang, Jia Wang, Jialin Wang, Miao Zhao, Weinan Zhang, Fuzheng Zhang, Xing Xie, and Minyi Guo. 2018. Graphgan: Graph representation learning with generative adversarial nets. In *AAAI*. 2508–2515.

[28] Hongwei Wang, Jia Wang, Miao Zhao, Jiannong Cao, and Minyi Guo. 2017. Joint Topic-Semantic-aware Social Recommendation for Online Voting. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. ACM, 347–356.

[29] Hao Wang, Naiyan Wang, and Dit-Yan Yeung. 2015. Collaborative deep learning for recommender systems. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 1235–1244.

[30] Hongwei Wang, Fuzheng Zhang, Min Hou, Xing Xie, Minyi Guo, and Qi Liu. 2018. Shine: Signed heterogeneous information network embedding for sentiment link prediction. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*. ACM, 592–600.

[31] Hongwei Wang, Fuzheng Zhang, Jialin Wang, Miao Zhao, Wenjie Li, Xing Xie, and Minyi Guo. 2018. RippleNet: Propagating User Preferences on the Knowledge Graph for Recommender Systems. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*. ACM.

[32] Hongwei Wang, Fuzheng Zhang, Xing Xie, and Minyi Guo. 2018. DKN: Deep Knowledge-Aware Network for News Recommendation. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 1835–1844.

[33] Quan Wang, Zhendong Mao, Bin Wang, and Li Guo. 2017. Knowledge graph embedding: A survey of approaches and applications. *IEEE Transactions on Knowledge and Data Engineering* 29, 12 (2017), 2724–2743.

[34] Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. 2017. Deep & Cross Network for Ad Click Predictions. In *Proceedings of the ADKDD'17*. ACM, 12.

[35] Zhen Wang, Jianwen Zhang, Jianlin Feng, and Zheng Chen. 2014. Knowledge graph and text jointly embedding. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 1591–1601.

[36] Ruobing Xie, Zhiyuan Liu, and Maosong Sun. 2016. Representation Learning of Knowledge Graphs with Hierarchical Types.. In *IJCAI*. 2965–2971.

[37] Ya Xue, Xuejun Liao, Lawrence Carin, and Balaji Krishnapuram. 2007. Multi-task learning for classification with dirichlet process priors. *Journal of Machine Learning Research* 8, Jan (2007), 35–63.

[38] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. 2014. How transferable are features in deep neural networks?. In *Advances in Neural Information Processing Systems*. 3320–3328.

[39] Xiao Yu, Xiang Ren, Yizhou Sun, Quanquan Gu, Bradley Sturt, Urvashi Khandelwal, Brandon Norick, and Jiawei Han. 2014. Personalized entity recommendation: A heterogeneous information network approach. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*. 283–292.

[40] Fuzheng Zhang, Nicholas Jing Yuan, Defu Lian, Xing Xie, and Wei-Ying Ma. 2016. Collaborative knowledge base embedding for recommender systems. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 353–362.

[41] Wenlu Zhang, Rongjian Li, Tao Zeng, Qian Sun, Sudhir Kumar, Jieping Ye, and Shuiwang Ji. 2015. Deep model based transfer and multi-task learning for biological image analysis. In *21st ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2015*. Association for Computing Machinery.

[42] Yu Zhang and Qiang Yang. 2017. A survey on multi-task learning. *arXiv preprint arXiv:1707.08114* (2017).

[43] Yu Zhang and Dit-Yan Yeung. 2012. A convex formulation for learning task relationships in multi-task learning. *arXiv preprint arXiv:1203.3536* (2012).

[44] Yu Zhang and Dit-Yan Yeung. 2014. A regularization approach to learning task relationships in multitask learning. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 8, 3 (2014), 12.

[45] Huan Zhao, Quanming Yao, Jianda Li, Yangqiu Song, and Dik Lun Lee. 2017. Meta-graph based recommendation fusion over heterogeneous information networks. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 635–644.

[46] Huaping Zhong, Jianwen Zhang, Zhen Wang, Hai Wan, and Zheng Chen. 2015. Aligning knowledge and text embeddings by entity descriptions. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. 267–272.

[47] Qiang Zhou and Qi Zhao. 2016. Flexible Clustered Multi-Task Learning by Learning Representative Tasks. *IEEE Trans. Pattern Anal. Mach. Intell.* 38, 2 (2016), 266–278.

The 15th International Conference on Document Analysis and Recognition

ICDAR 2019 Robust Reading Challenge on Reading Chinese Text on Signboard

Xi Liu[1], Rui Zhang[1], Yongsheng Zhou[1], Qianyi Jiang[1], Qi Song[1], Nan Li[1], Kai Zhou[1], Lei Wang[1], Dong Wang[1], Minghui Liao[2], Mingkun Yang[2], Xiang Bai[2], Baoguang Shi[3], Dimosthenis Karatzas[4], Shijian Lu[5]

[1]Meituan-Dianping Group, China, [2]School of EIC, Huazhong University of Science and Technology, China,

[3]Microsoft Redmond, USA, [4]Computer Vision Centre, UAB, Spain, [5]Nanyang Technological University, Singapore

***Abstract*—Chinese scene text reading is one of the most challenging problems in computer vision and has attracted great interest. Different from English text, Chinese has more than 6000 commonly used characters and Chinese characters can be arranged in various layouts with numerous fonts. The Chinese signboards in street view are a good choice for Chinese scene text images since they have different backgrounds, fonts and layouts. We organized a competition called ICDAR2019-ReCTS, which mainly focuses on reading Chinese text on signboard. This report presents the final results of the competition. A large-scale dataset of 25,000 annotated signboard images, in which all the text lines and characters are annotated with locations and transcriptions, were released. Four tasks, namely character recognition, text line recognition, text line detection and end-to-end recognition were set up. Besides, considering the Chinese text ambiguity issue, we proposed a multi ground truth (multi-GT) evaluation method to make evaluation fairer. The competition started on March 1, 2019 and ended on April 30, 2019. 262 submissions from 46 teams are received. Most of the participants come from universities, research institutes, and tech companies in China. There are also some participants from the United States, Australia, Singapore, and Korea. 21 teams submit results for Task 1, 23 teams submit results for Task 2, 24 teams submit results for Task 3, and 13 teams submit results for Task 4. The official website for the competition is http://rrc.cvc.uab.es/?ch=12.**

I. Introduction

Texts in natural images carry much important semantic information. Reading text in natural scene images has been widely studied recently since it is an important prerequisite for many content-based image analysis tasks such as photo translation, fine-grained image classification and autonomous driving.

It is widely recognized that large-scale, well-annotated datasets are crucial to today's deep learning based techniques. In scene text reading field, many scene text datasets have been collected. Especially for Chinese text reading, more and more Chinese scene text datasets are proposed, such as MSRA-500 [1], RCTW [2], SCUT-CTW1500 [3], CTW [4].

Chinese text reading is a huge challenge task. Different from English text reading, Chinese has more than 6000 commonly used characters. Besides, owing to the Chinese culture, the layouts, arrangements and fonts of Chinese characters are always in a great variety, as shown in Figure 1.

The Chinese signboards in street view may be the best source for Chinese scene text images since they have different backgrounds, fonts and layouts. In Meituan-Dianping Group, a Chinese leading company for food delivery services, consumer products and retail services, there are many signboard images collected by Meituan business merchants. Based on this, we propose a competition for Chinese text reading on signboard and construct a large-scale challenging natural scene text dataset of 25,000 signboard images. About 200,000 text lines and 600,000 characters are labeled with locations and transcriptions. We set up four tasks for this competition, namely character recognition, text line recognition, text line detection and end-to-end recognition. Besides, we propose a multi ground truth (multi-GT) evaluation method considering the Chinese text ambiguity. As illustrated in Figure 2, it is difficult to determine whether some words should be merged to a text instance or not. We thus provide one or more ground truths for each test image and compare the predicted result with all the ground truths when evaluating. The best matched GT will be used to calculate the evaluation metrics.

Figure 1. Characters with various layouts and fonts.

Figure 2. Chinese text ambiguity in signboard image.

Figure 3. Chinese character test images.

Figure 4. Text line test images.

The competition lasts from March 1st to April 30, 2019. It receives lots of attention from the community. For all the four tasks, there are all together 46 valid teams participating in the competition and hundreds of valid submissions are received. In this report, we will present their evaluation results.

II. DATASET AND ANNOTATIONS

The dataset, named ReCTS-25k, comprises 25,000 signboard images. All the images are from Meituan-Dianping Group, collected by Meituan business merchants, using phone cameras under uncontrolled conditions. Different from other datasets, this dataset mainly focuses on Chinese text reading on the signboards. The layout and arrangement of Chinese characters in signboards are much more complex for the sake of aesthetics appearance or highlighting certain elements. Figure 1 shows some example images.

We manually annotate the locations and transcriptions for all the text lines and characters in the signboard images. Note that the utterly obscure and small text lines and characters are marked with a difficult flag. Locations are annotated in terms of polygons with four vertices, which are in clockwise order starting from the upper left vertice. Transcriptions are UTF-8 encoded strings.

The dataset is split into two subsets. The training set consists of 20,000 images, and the test set consists of 5,000 images. Moreover, 29335 character images and 10789 text lines images, cropped from the 5000 test images, are used for task 1 and task 2 evaluation respectively.

III. CHALLENGE TASKS

Robust reading challenge on Chinese signboard consists of four tasks: 1) Character recognition, 2) Text line recognition, 3) Text line detection, 4) End-to-end recognition. Given that Chinese signboards have various layouts, fonts and orientations, character and text line reading are concerned. Therefore, in our competition, character based and text line based tasks are both evaluated.

Note that the half-width character and its corresponding full-width character are regarded as one character in the evaluation of task 2 and task 4. Moreover, the English letters are not case sensitive.

A. Task 1 – Character Recognition

The aim of this task is to recognize characters of the cropped character images from Chinese signboards. As illustrated in Figure 3, the Chinese characters take the largest portion and are in diverse fonts. Participant is asked to submit a text file containing character results for all test images. The recognition accuracy is given as the metric:

$$\text{accuracy} = \frac{\mathrm{N}_{\text{right}}}{N_{total}}, \tag{1}$$

where $\mathrm{N}_{\text{right}}$ is the number of characters predicted correctly and $\mathrm{N}_{\text{total}}$ is the total number of the test characters.

B. Task 2 – Text Line Recognition

The target of text line recognition is to recognize the cropped word images of scene text. The cropped text line images as well as the coordinates of the polygon bounding boxes in the images are given. The given points are arranged in the clockwise order, starting from the top-left point. Figure 4 shows some examples of the test set. The text line images may contain perspective and arbitrary arranged text lines.

The results are evaluated by the Normalized Edit Distance between the recognition result and the ground truth. The edit distances are summarized and divided by the number of test images. The resulting average edit distance is taken as the metric and is formulated as follows:

$$\text{accuracy} = 1 - \frac{1}{N}\sum_{i=1}^{N}\frac{D(s_i,\hat{s_i})}{\max(s_i,\hat{s_i})}, \tag{2}$$

where D stands for the Levenshtein Distance, s_i denotes the predicted text line and $\hat{s_i}$ denotes the corresponding ground truth, N is the total number of text lines.

C. Task 3 – Text Line Detection

The aim of this task is to localize text lines in the signboard. The input image is the full signboard images. The detection results submitted by the participants are required to give four vertices of the polygon in clockwise order.

In some signboard, there always exist the following case, as shown in Figure 2. It is difficult to determine whether the boxes "砂锅" , “炒面” , “拌面” , “烩肉” , “泡馍” should be merged to a large text box or not. Therefore, we regard the two cases (Figure 2(a) and Figure 2(b)) as correct ground truth. We provide one or more ground truths for each test image. When

evaluating, we compare the predicted result with all the ground truths and use the best matched one to calculate the evaluation metrics.

Following the evaluation protocols of ICDAR 2017-RCTW [2] dataset, the detection task is evaluated in terms of Precision, Recall and F-score with intersection-over-union (IoU) threshold of 0.5 and 0.7. The F-score at IoU=0.5 will be used as the only metric for the final ranking. All detected or missed ignored ground truths will not contribute to the evaluation result.

D. Task 4 – End-to-End Recognition

The aim of this task is to localize and recognize every text instance in the signboard. The input image is the full signboard images. Participants are required to submit the text file containing all the recognized text lines locations and transcriptions for each test image. Similar to Task 3, the locations are four vertices in clock-wise order and the transcripts are UTF-8 encoded strings.

The evaluation process consists of two steps. First, each detection is matched to a ground truth polygon that has the maximum IOU, or it is matched to 'None' if none IOU is larger than 0.5. If multiple detections are matched to the same ground-truth, only the one with the maximum IOU will be kept and the others are recorded as 'None'. Then, we calculate the edit distances between all matching pairs by Formula (2). Since one test image may have multiple ground truths, as stated in Task 3, we also compare the predicted result with all the ground truths and use the best matched one to calculate the evaluation metrics.

IV. ORGANIZATION

The competition starts on March 1, 2019, when the RRC website is ready and open for registration. The training set is released on March 18, the first part of test set is released on April 12 and the second part of test set released on April 20. We revise the test set more than once to fixed some errors before releasing the test set. The RRC website opens for result submission on April 20 and closes at 11:59 PM PST, April 30.

There are all together 46 valid teams participated in the competition. Most of the participants come from universities, research institutes, and tech companies in China. There are also some participants from the United States, Australia, Singapore, and Korea.

All the teams submit their results through the RRC website. Each team is allowed to submit 5 results at most and we choose the best result among the 5 results as the final result. 21 teams submit results for Task 1, 23 teams submit results for Task 2, 24 teams submit results for Task 3, and 13 teams submit results for Task 4.

V. SUBMISSIONS AND RESULTS

The evaluation script is implemented in Python. We run the script to evaluate all the submissions. Table I summarizes the top 5 results of Task 1. Methods are ranked by their accuracy. Table II summarizes the top 5 results of Task 2. Methods are ranked by their normalized edit distance. Table III summarizes the top 5 results of Task 3. Methods are ranked by their F-score. Table IV summarizes the top 5 results of Task 4. Methods are ranked by their normalized edit distance. You can view the complete ranking in the home page of the competition https://rrc.cvc.uab.es/?ch=12.

A. Top 3 submissions for Task 1

1. "BASELINE v1" (USTC-iFLYTEK) The method uses image classification methods and its ensemble.

2. "Amap_CVLab" (Alibaba AMAP) The method adds res-block [5] (for the lower dimension feature collapse avoiding) and se-block [6]. Their training dataset contains both the ReCTS-25k and other data.

3. "TPS-ResNet v1" (Clova AI OCR Team, NAVER/LINE Corp) The method uses Thin-plate-spline(TPS) [7] based Spatial transformer network (STN) [8], which normalizes the input text images. They use ResNet [5], BiLSTM [9] and attention mechanism. Their training dataset contains the Chinese synthetic datasets (MJSynth and SynthText [10]) and real dataset (ArT [11], LSVT [12], RCTW [2], ReCTS-25k).

B. Top 3 submissions for Task 2

1. "SANHL" (South China University of Technology, Northwestern Polytechnical University, The University of Adelaide, Lenovo and Huawei) The method uses an ensemble framework, which consists of attention-based network, transformer network and CTC-based [13] network. Apart from the official training dataset, about 2 million synthesized samples are used for training.

2. "Tencent-DPPR Team" (Tencent-DPPR Team) The method uses five types of deep models, which mainly include CTC-based nets and multi-head attention based nets. All samples are resized to the same height before feeding into the network. Furthermore, besides ReCTS, they use a synthetic dataset containing more than fifty million images, as well as open-source datasets including LSVT [12], COCO-Text [14], RCTW [2] and ICPR-2018-MTWI. In terms of data augmentation, they mainly use Gaussian blur, Gaussian noise and so on.

3. "HUST_VLRGROUP" (Huazhong University of Science and Technology) A CRNN based method.

C. Top 3 submissions for Task 3

1. "SANHL_v4" (South China University of Technology, The University of Adelaide, Northwestern Polytechnical University, Lenovo, HUAWEI) The method uses a sequential-free box discretization method to localize the text instances. Multi-scale testing and model ensemble are used to generate the final result. Their training dataset contains LSVT [12], ArT [11], MLT [15] and ReCTS-25k.

2. "Tencent-DPPR Team" (Tencent Data Platform Precision Recommendation) Their text detector is based on two-stage method with multi-scale training policy, and ResNet101 [5] is used as the backbone network. They use feature pyramid layers [16] to extract features instead of choosing one layer according to box sizes. They use LSVT [12] pre-trained model.

3. "Amap-CVLab" (Alibaba AMAP, Alibaba DAMO Academy for Discovery, Adventure, Momentum and Outlook) The method is based on Mask R-CNN [17]. Their training dataset contains RCTW[2], ICDAR2017-MLT[15], LSVT[12], ReCTS-25k.

D. Top 3 submissions for Task 4

1. "Tencent-DPPR Team" (Tencent-DPPR Team) In the detection part, they use a text detector based on two-stage method. This method uses ResNet101 [5] as feature extractor, and they design a policy to help proposals select feature pyramid layers [16] to extract features instead of choosing one layer according to box sizes. In detection ensemble stage, they apply a multi-scale test method with different backbones. When ensembling all the results, they develop an approach to vote boxes after scoring each box. In the recognition part, they use an ensemble model, which includes CTC-based nets and multi-head attention based nets. For this task, they use the predicted confidence scores of cropped words and the ensemble results to select the reliable one among results predicted by all models.

2. "SANHL" (South China University of Technology, Northwestern Polytechnical University, The University of Adelaide, Lenovo and Huawei) The method firstly detect possible text lines, and then predict strings by an ensembled recognition model.

3. "HUST_VLRGROUP" (Huazhong University of Science and Technology) The method uses Mask R-CNN as text detector and a CRNN based approach to predict strings.

E. Baseline submissions

For reference, we submit a baseline method to Task 1, Task 2, Task 3 and Task 4 respectively. The methods are implemented by ourselves. Their results are shown in Table I, II, III and IV.

For Task 1, the character Recognition method is based on the densely connected convolutional network (DenseNet) [18]. Our network inherits from the DenseNet-169 network model with dense blocks, but we reduce the number of last dense block to 24 and all the growth rates in the networks are 32. The training dataset consists of ReCTS and synthetic data.

For Task 2, We took the Chinese text line recognition as a sequence recognition task. We utilized a modified version of Inception-V4 [19], integrated with attention module to extract feature maps. The CTC layer for transcription is adopted. The baseline result is obtained by a single recognition model, the training dataset consists of ReCTS, RCTW [2], and LSVT [12], no synthetic data is utilized.

For Task 3, the text detection method is based on SEG-FPN [20] and Pixel-link [21]. We build a unified framework, which combines pixel link and segment link in feature pyramid network to detect scene text. The training dataset only consists of ReCTS.

For Task 4, we first detect the text line in the image. If the text line is horizontal, recognize it by the line recognition model; if the text line is vertical, character detection and character recognition model will be used. The text line detection part is the same as that for Task 3, the character recognition part is the same as that for Task 1, and the text line recognition part is the same as that for Task 2. A Faster-RCNN [22] based detection approach is adopted to detect Chinese character regions.

VI. CONCLUSIONS

We organize the competition on reading Chinese text on signboard (ReCTS). A large-scale challenging natural scene text dataset of 25,000 signboard images are released and four tasks are set up. We also propose a multi-GT evaluation strategy intended for Chinese text ambiguity. During the challenge, we receive hundreds of submissions from 46 teams, which shows the broad interest in the community. In the future, we plan to make the evaluation scripts available on the website https://rrc.cvc.uab.es/ and users can get the evaluation results shortly after they submit the results to the website.

REFERENCES

1. C. Yao, X. Bai, W. Liu, Y. Ma, Z. Tu, "Detecting Texts of Arbitrary Orientations in Natural Images," CVPR, 2012.
2. B. Shi, C. Yao, M. Liao, M. Yang, P. Xu, L. Cui, S. J. Belongie, S. Lu, X. Bai, "ICDAR2017 Competition on Reading Chinese Text in the Wild (RCTW-17)," ICDAR, 2017.
3. Y. L. Liu, L. W. Jin, S. T. Zhang, S. Zhang, "Detecting Curve Text in the Wild: New Dataset and New Solution," arXiv, 2017.
4. T. L. Yuan, Z. Zhu, K. Xu, "Chinese Text in the Wild," arXiv, 2018.
5. K. He, X. Zhang, S. Ren, "Deep Residual Learning for Image Recognition," CVPR, 2016.
6. J. Hu, L. Shen, G. Sun, "Squeeze-and-Excitation Networks," TPAMI, 2017.
7. F. L. Bookstein, "Principal warps: Thin-plate splines and the decomposition of deformations," TPAMI, 1989.
8. M. Jaderberg, K. Simonyan, A. Zisserman, K. Kavukcuoglu, "Spatial Transformer Networks," CVPR, 2016.
9. A. Graves, J. Schmidhuber, "Framewise phoneme classification with bidirectional LSTM and other neural network architectures," Neural Networks, 2005.
10. A. Gupta, A. Vedaldi, and A. Zisserman, "Synthetic data for text localisation in natural images," In CVPR, pages 2315–2324, 2016.
11. https://rrc.cvc.uab.es/?ch=14, 2019.
12. https://rrc.cvc.uab.es/?ch=16, 2019.
13. A. Graves, S. Fernandez, F. Gomez, J. Schmidhuber, "Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks," ICML, 2006.
14. V. Andreas, M. Tomas, N. Lukas, M. Jiri, B. Serge, "COCO-Text: Dataset and Benchmark for Text Detection and Recognition in Natural Images," arXiv, 2016.
15. N. Nayef, et al, "ICDAR2017 Robust Reading Challenge on Multi-lingual Scene Text Detection and Script Identification," ICDAR 2017.
16. Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, Serge Belongie, "Feature Pyramid Networks for Object Detection," arXiv preprint. arXiv: 1612.03144, 2017.
17. K. He, G. Gkioxari, P. Dollar, R. Girshick, "Mask r-cnn," ICCV, 2017.
18. G. Hunag, Z. Liu, L. V. DerMaaten, K. Q. Weinberger, "Densely connected convolutional networks." CVPR, 2017.
19. C. Szegedy, et al, "Inception-V4, inception-resnet and the impact of residual connections on learning," AAAI, 2017.
20. X. Liu, R. Zhang, Y. S. Zhou, D. Wang, "Scene Text Detection with Feature Pyramid Network and Linking Segments," ICDAR, 2019.
21. D. Deng, H. Liu, X. Li, and D. Cai, "Pixellink: Detecting scene text via instance segmentation," In AAAI, pages 6773– 6780, 2018.
22. S. Q. Ren, K. He, R. Girshick, J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," NIPS, 2015.

TABLE I: RESULTS SUMMARY FOR THE TOP-5 SUBMISSIONS OF TASK 1.

Ranking	Team Name	Affiliation	Accuracy
1	BASELINE-v1	iFLYTEK, University of Science and Technology of China	0.9737
2	Amap_CVLab	Alibaba AMAP	0.9728
3	TPS-ResNet-v1	Clova AI OCR Team, NAVER/LINE Corp	0.9612
4	SANHL_v4	South China University of Technology, The University of Adelaide, Northwestern Polytechnical University, Lenovo, HUAWEI	0.9594
5	Tencent-DPPR	Tencent (Data Platform Precision Recommendation)	0.9512
Baseline		**Meituan Dianping**	**0.9140**

TABLE II: RESULTS SUMMARY FOR THE TOP-5 SUBMISSIONS OF TASK 2.

Ranking	Team Name	Affiliation	N.E.D
1	SANHL_v1	South China University of Technology, The University of Adelaide, Northwestern Polytechnical University, Lenovo, HUAWEI	0.9555
2	Tencent-DPPR	Tencent (Data Platform Precision Recommendation)	0.9486
3	HH-Lab-v4 *	Huazhong University of Science and Technology (Visual and Learning Representation Group)	0.9483
4	TPS-ResNet-v1	Clova AI OCR Team, NAVER/LINE Corp	0.9477
5	Baseline-Beihang*	Beihang University	0.9437
Baseline		**Meituan Dianping**	**0.9089**

TABLE III: RESULTS SUMMARY FOR THE TOP-5 SUBMISSIONS OF TASK 3.

Ranking	Team Name	Affiliation	F-score
1	SANHL_v4	South China University of Technology, The University of Adelaide, Northwestern Polytechnical University, Lenovo, HUAWEI	0.9336
2	Tencent-DPPR	Tencent (Data Platform Precision Recommendation)	0.9303
3	Amap-CVLab	Alibaba AMAP, Alibaba DAMO Academy for Discovery, Adventure, Momentum and Outlook	0.9250
4	HH-Lab *	Huazhong University of Science and Technology (Visual and Learning Representation Group)	0.9127
5	maskrcnn_text *	Huazhong University of Science and Technology (Media and Communication Laboratory, Text detection)	0.9102
Baseline		**Meituan Dianping**	**0.9001**

TABLE IV: RESULTS SUMMARY FOR THE TOP-5 SUBMISSIONS OF TASK 4.

Ranking	Team Name	Affiliation	N.E.D
1	Tencent-DPPR	Tencent (Data Platform Precision Recommendation)	0.8150
2	SANHL_v1	South China University of Technology, The University of Adelaide, Northwestern Polytechnical University, Lenovo, HUAWEI	0.8144
3	HH-Lab *	Huazhong University of Science and Technology (Visual and Learning Representation Group)	0.7943
4	baseline_Beihang *	Beihang University	0.7661
5	SECAI *	Institute of Information Engineering, Chinese Academy of Sciences, University of Science & Technology Beijing	0.7437
Baseline		**Meituan Dianping**	**0.7298**

* means student contestant

微信扫码关注技术团队公众号

tech.meituan.com
美团技术博客

新年
快乐