/ 携 程 技 术 /

2022 年度合辑

大 前 端 / 研 发 效 能
架 构 / 数 据 库 / 云 计 算 / 运 维

Trip.com 携程技术出品

“携程技术”微信公众号

分享，交流，成长

作为携程集团的核心竞争力，携程技术由数千位来自海内外的精英工程师组成，为携程集团业务的运作和开拓提供全面技术支持，并以技术创新源源不断地为产品和服务创造价值。技术从来都不是闭门造车，携程技术团队会一直以开放和充满热情的心态，通过各种渠道和方式，和圈内小伙伴们探讨、交流、碰撞，共同收获和成长。

目录

序

2023 年，当世界重新与中国相连，我们有信心让与世界睽违三年再次踏上旅途的旅行者拥有完美旅程。这是携程人在新年伊始面对春天做出的承诺以及对更美好世界的渴望。

疫情三年，一直在改变着我们交流的方式，令生活被切割成无可拼凑的碎片；我们急切地讨论着远方的故事，却又置身不确定的当下。而如今春暖花开，作为携程技术人正在把握机遇，做到用实力过硬的技术，打造强适应性的产品，用“全球化视野 + 本地化运营” 升级用户体验和服务模式。技术趋势的变化离不开内外因的合力作用，面对创新发展和走出去的战略需要，携程技术人将从拥抱变化到引领变化，以己为尺，拓宽边界，从“Copy to Global”走向“Born Global”。

- 大前端领域在过去的一年降本增效、专注核心。加深对前端框架和底层库、性能稳定等相关建设，例如 Flutter、Taro、QUIC 等性能提升，通过拓展基建能力，快速稳定保证用户体验；在工程效能、安全兜底等方面，基于前端技术特点，优化对前端构建和诸多跨语言构建方案，突破核心问题，优化开发和产品运营体验；

- 数据的”扩展”和”融合”，业务的复杂多样性带来了技术需求的挑战，例如一些场景可能一个单机数据库就能支持，用分布式似乎有点“大材小用”，但是有一些场景又需要分布式数据库以支撑海量数据和海量交易。如何应对在数据库规模比较小的时候，不需要复杂的分布式事务机制，而随着数据规模增长，又能自动扩展分布式，而不需要大规模的数据库迁移等，在底层部署，支撑酒店、金融等业务方面，我们一直在打磨技术架构迭代精进；

- 随着微服务的云原生化，服务的角色也从单纯的提供资源，变成了构建应用的新平台，尽可能减小机器运维等低价值重复工作，从而聚焦于业务的创新，用更简单的方式提高资源利用率；我们一直在思考和实践如何借助云的优势，简化微服务的管理和运维问题，让技术人员更专注地赋能业务；

技术趋势推进行业变革，行业趋势反哺技术价值，两者相互促进，才能持续保持技术的生命力。但技术与应用的瞬息万变，又容易让人心生迷惘或疲于奔命，技术人唯有注重打好思考方法与技术理念的根基，将技术真正融入知识体系，从而一通百通，从容面对随时涌现的业务需求和技术变化。并与社区和技术同行们一起携手，持续开源革新，不断前进。

在 2023 年，携程技术人将本着”Born Global”的理念，利用自身技术能力，从一开始就着眼于打造出真正意义上的全球化产品和服务。越过障碍，直入本质，直至跃向更高级。

经验这个东西，有时并不能告诉我们什么一定对，但是可以告诉我们什么一定不对。希望今年这份作为来自“携程技术”微信公众号全年度的重要技术总结，合辑中的 30 篇文章，覆盖了大前端、架构、研发效能、数据库、云计算、运维等 6 个领域，就像一列满载经验分享的列车，带着我们的新年祝福，缓缓穿行在辞旧迎新的大地上，一路向家。

祝大家 2023 年宏“兔”大展，新年快乐，重逢将近！

携程副总裁/技术委员会主席 马超
2023 年 1 月 上海

大前端

开源 | 携程 Foxpage 前端低代码框架

【作者简介】Jason Wang，携程研发经理，目前主要负责低代码类产品的设计和研发，关注低代码行业的发展及相关解决方案在企业内部的落地。

一、背景

随着低代码开发方式被越来越多的人接受和认可，低代码得到了蓬勃发展，更被寄希望成为 IT 行业革命性的“新生产力”。据报道，全球低代码类产品市场规模在 2021 年超百亿美元，预计 2023 年将突破两百亿美元关口。

1.1 低代码行业现状

国外：Salesforece，Microsoft，OutSystems，Mendix 等，是处于领导者地位的资深玩家，产品功能强大，流程完善，生态健全。

国内：钉钉宜搭，明道云，织信，奥哲，简道云等，大都结合某个场景提供支持方案，还处于发展阶段。

还有些玩家在企业内部发展，并未公开。各类玩家各有优势，关注的方向也各有差异。有的是结合某个行业定制，有的是结合单个使用场景深耕，还有的执着于提供应用开发的完整低代码方案。大部分是 SaaS 产品，其中部分会有私有化部署方案。

1.2 暗藏的隐患

按理说，市场上低代码产品功能已经足够强大，企业内部却还在不停自研各类低代码产品，甚至同一个业务线的不同使用场景或不同的技术栈也都有不同的低码类实现方案。据说某大厂内部的低代码类系统或工具高达上百个。

其实这些系统或工具前端部分的核心功能和能力大体是相似的，就拿前端可视化页面搭建这部分来讲，从技术上看大家最终的实现方案和体验上都趋于相近或相同，自研的整个过程会有大量的重复性的工作，造成了人力资源的浪费。

为什么会造成这样的局面？大家的本意是想借助低代码方案解决业务问题，但却发现行业基建太差，借助成品的方案也难以落地。只能一遍遍重复着低代码的基础能力和设施的建设，乐此不疲也无奈地重复造轮子。

1.3 低码通用能力建设

基于此我们做了一些调研，整理了市面上成熟产品可能存在的问题：

- 功能臃肿，系统复杂，对接入和二次开发成本都较高；

- 不够开放，定制化程度高，部分的产品还有技术栈的约束；
- 场景支持单一，扩展较难，对一些碎片化的场景支持不友好。

既然大而全的低码产品不能用，那就给个小而美的低码框架吧。

为了能让前端项目快速且低成本的体验到低代码带来的便利，带着市面上成熟的产品的问题，我们决定从建设前端低代码开发平台的角度去研发一套前端低代码框架，提供前端低代码类产品的通用部分能力，帮助开发者解决重复开发，重复建设的问题。

同时也设立了一些需要完成的目标：

- 有多场景，多端，多技术栈的支持能力；
- 产出多个场景下前端组件化的最佳实践；
- 提供前端低代码开发所需要的基础能力和设施；
- 探索一套围绕着前端低代码开发的工程化体系。

二、Foxpage 是什么？

Foxpage 是一个轻量级的前端低代码框架，借助 Foxpage 可以让前端项目用低代码的方式进行迭代。Foxpage 重点在前端，关注前端页面的整个生命周期，希望成为一个易用，灵活，开放且百搭的开源框架。

2.1 特性

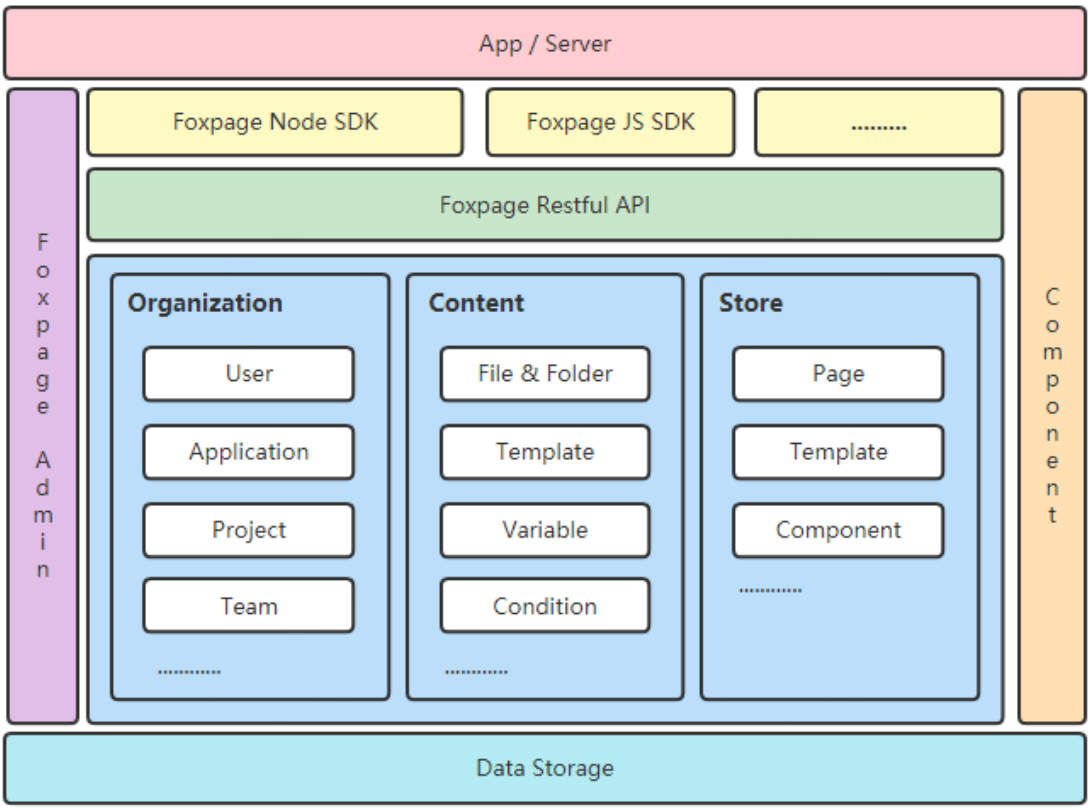
- 可视化，提供在线的可视化的前端页面搭建，所见即所得；
- 组件化，提供较为完善的组件制作流程和组件化方案，制作页面先从制作组件开始；
- 可扩展，提供多端，多技术栈及多种场景的支持；
- 国际化，提供一套国际化内容管理方案；
- 平台化，给开发者、编辑、运营等提供了一个在线合作的平台。

国际化和平台化为以后建设通用的前端低代码开发平台提供了基础。

三、架构

3.1 整体架构

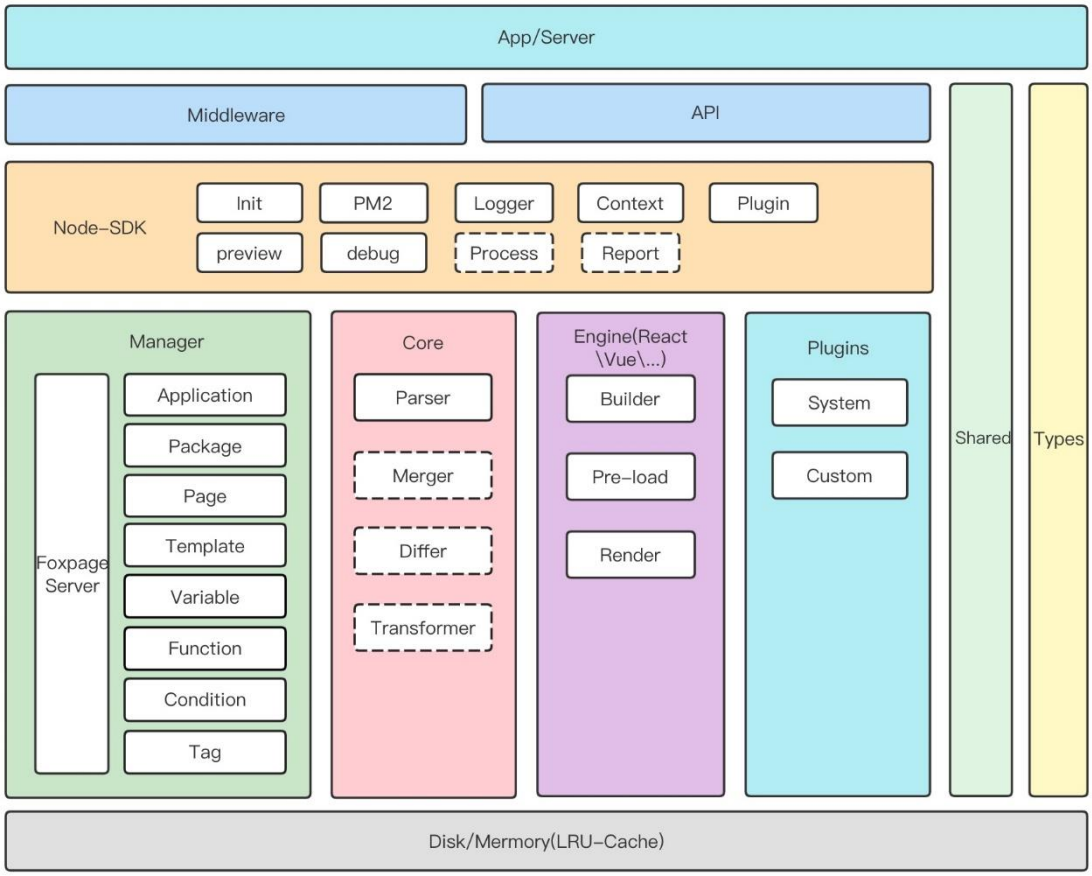
整个框架包含 Foxpage Admin，Foxpage API，Foxpage SDK，组件库等部分，下图描述了框架各模块及它们之间的关系。



- Foxpage Admin ： Foxpage 的管理后台，提供组织，应用，项目，页面&模板等管理功能；
- Foxpage API：Foxpage Restful API 主要用于为 Foxpage SDK 及 Foxpage Admin 提供的接口服务，开发者也可以使用其开发其他功能；
- Foxpage SDK ： 目前版本只提供了 Node SDK 及 JS SDK 用于 Node 端和浏览器端应用接入。

3.2 Node SDK 架构

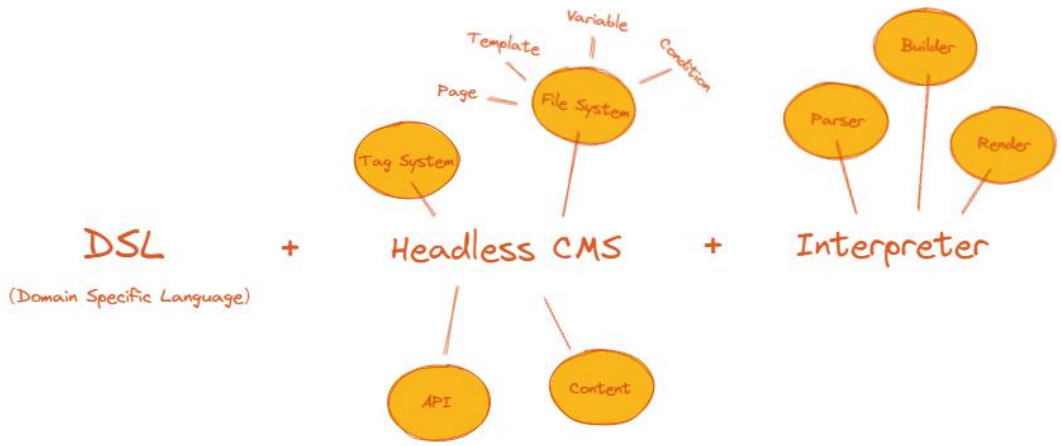
Node SDK 是提供给 node 端应用使用的开发工具包，通过 SDK 开发者能够快速接入和使用 Foxpage 框架。



除了上面的介绍，Foxpage 整个项目还有 Foxpage CLI、Foxpage Debugger 及组件部分相关的介绍，有兴趣的同学可以前往官方文档查看。

3.3 核“芯”设计

Foxpage 核心部分是围绕着 Foxpage 需要提供支持多场景，多端，多技术栈的能力来设计的，可以算是对低代码开发实践的沉淀。



（1）DSL（Domain Specific Language）

DSL 译为领域特定语言，作用是通过在表达能力上做的妥协来换取在某一特定领域的高效的。最常见的 DSL 有 Html，CSS，SQL，Regex 等。

为了各类内容编辑高效性和一致性，我们基于 JSON 设计了一套 Foxpage 的 DSL，主要用作描述页面和组件等内容。再结合自身需求做了一些扩展，如为了加强 DSL 的动态能力加入了“变量”，“条件”等描述。特定的语法和语义需要提供对应的解析器，结合不同的应用场景提供定制功能，再针对具体的端提供对应 SDK 实现供应用接入。

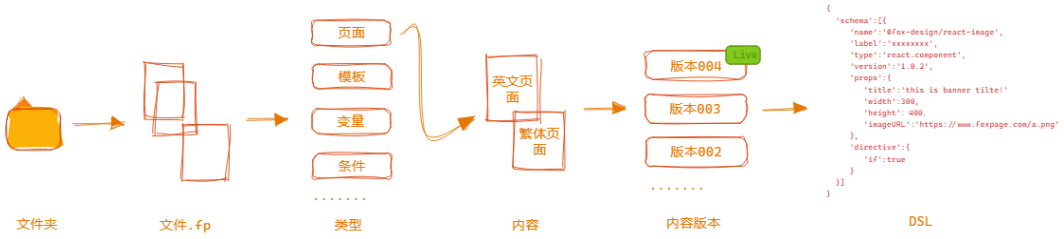
（2）页面 DSL 的片段

上图 DSL 中描述了一个横幅组件，包含资源，属性，事件，条件渲染等信息，当然完整的页面描述还包含模板，页面，资源详情等描述。为了减少 DSL 的冗余，我们对公共部分做了最大程度的提取和复用。同时为了防止单份 DSL 内容过大，还支持了 DSL 模块化设计。运行时在 DSL 解析时会做合并，补全等操作。

（3）无头 CMS（Headless CMS）

无头 CMS 与传统的 CMS 相比，不同之处在于其将内容和展示分离，达到“前后端分离”的目的，这样可以和前端的技术栈解耦，增强扩展性。

这里主要提供了各类内容的定义、管理、存储和分发等功能，其实就是管理着各类 DSL 数据，视为框架的基建部分。参照文件系统的设计，提供了文件夹和文件等基础的功能。并可自由的新增文件类型，让整个系统变得更自由，更易扩展。同时还提供了 Restful API 方便了各端可以自由的获取到托管的内容，也使得各接入端拥有了动态更新能力。

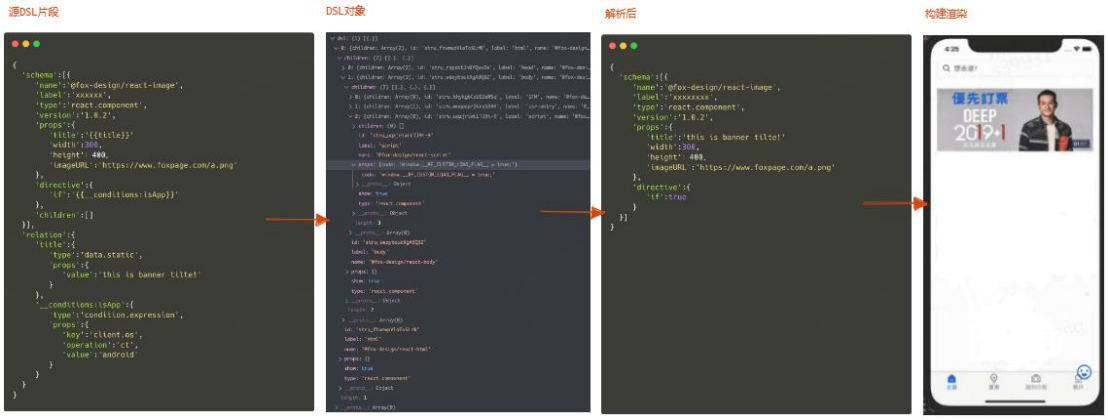


这样的设计也为前端项目想离线的使用 Foxpage 框架提供了基础，后续 Foxpage 也会为应用提供离线使用方案，使得整个框架变得更灵活。

（4）解释器（Interpreter）

这里解释器的作用，是在应用的运行过程中去读懂并理解 DSL 然后再做对应的执行。广义上来说解释器主要包含资源管理器，DSL 解析器，渲染引擎三个部分

可以简单看下解释器在 H5 页面渲染过程中所做的工作:

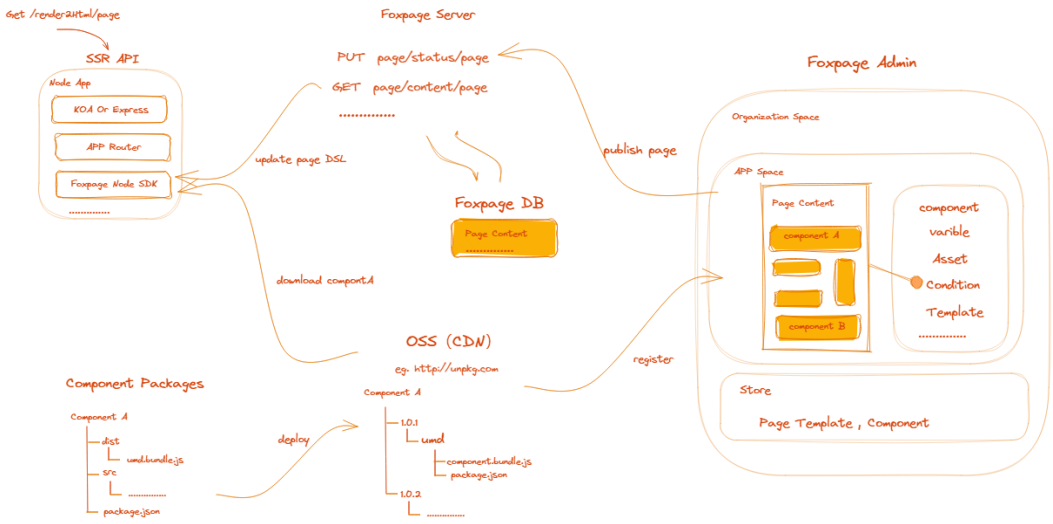


这种解释执行的方式会带来一定的性能损耗，但是和它带来的高扩展能力和动态能力来比还是可以接受的。整个低代码的开发方式，本身也是牺牲一定的灵活性来换取开发效率的提升的，这要看怎么平衡和取舍。

3.4 如何工作?

举个邮件页面渲染服务（SSR API 以下简称 API）的例子来说明下工作过程。

首先 API 项目需要接入 Foxpage Node SDK（以下简称 SDK）。当用户请求 API 获取邮件页面 HTML 文档时，SDK 会请求 Foxpage Restful API 获取邮件页面的内容信息（DSL），拿到页面 DSL 后会走解析流程，做一些预处理、数据绑定及资源文件加载（比如组件的 umd 文件）等工作。解析完成后 SDK 会根据解析后的对象做页面的构建和渲染（SDK 默认内置 Reactjs 框架，这里的邮件页面组件为 React 实现的），最终调用 Reactjs 框架的接口输出页面的 HTML 内容。



四、Why Foxpage?

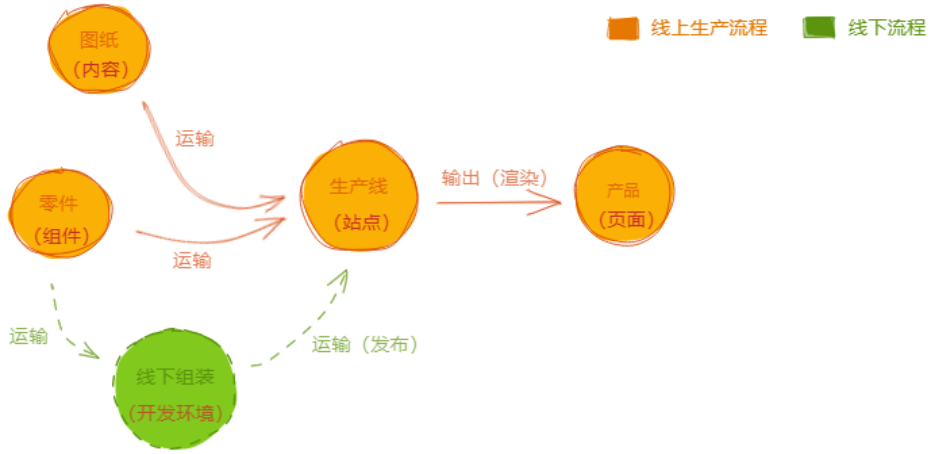
以下是 Foxpage 的技术目标，这也是框架核心的能力，降低成本同时提高生产力。

- 提高效率，效率包括组件开发效率，页面开发效率，发布效率等；
- 降低成本，成本包括人员成本，开发难度，迭代成本，维护成本等；
- 质量保障，质量包括稳定性，“良品率”等 （这里的质量的保障是可持续性的，不应因业务的变化而降低或缺失，是一套通用的质量保障体系）。

目前前端开发还是一个“劳动密集型行业”，需要大量的开发资源堆叠去解决业务需求。虽然有了一些工程化的加持，但在面对成倍的开发需求增长时，效率和成本问题会凸显并放大。Foxpage 的出现可以一定程度上缓解这一问题，让前端开发多一种选择。

4.1 低码开发模式

开发从“pro code”到“low code”的转变。拿前端页面开发举例（为方便理解，结合制造业流程来说明），传统的开发方式是在线下制作组装，经过一系列工序后发布到线上交付。而低码的方式是在线可视化搭建页面，然后产出一张张施工图纸，在交给生产线做线上的自动化组装，最后线上交付。



这种变化带来的益处：

- 降低开发门槛，有画图能力的人就能做页面开发；
- 提升交付效率；
- 页面质量更高。

4.2 工程化的支持

从传统前端的工程化转变为围绕着低代码开发建设的新的工程化体系。不管是新的还是传统的工程化，最终目的都是为了高效且低成本的开发并保质保量的交付。低码的开发模式中同样需要一些工程化的支持，只是会换成另外一种形式而已。

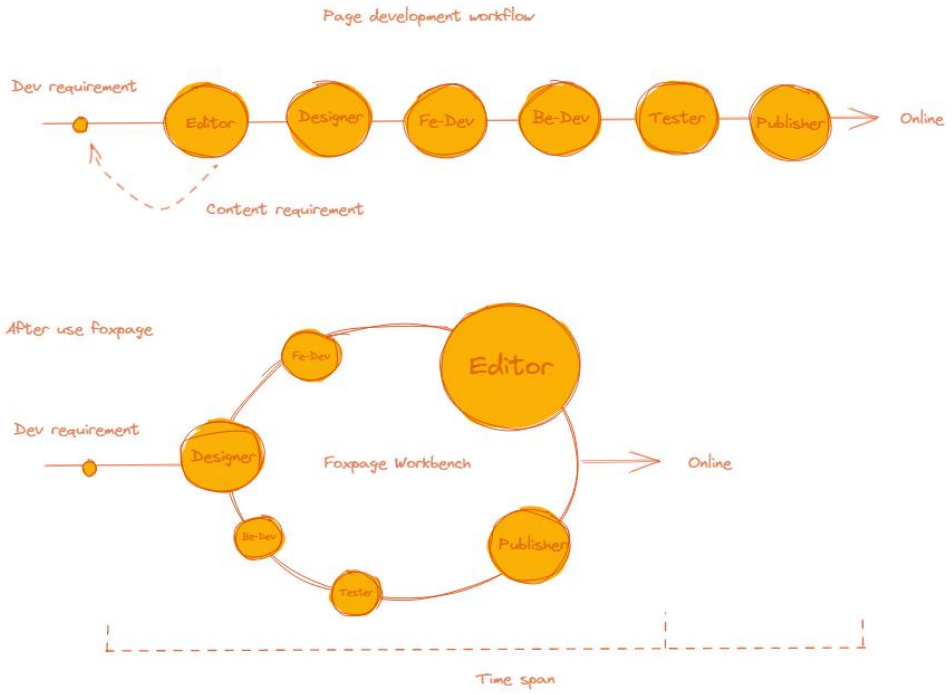
框架提供的基础能力：

- 零件（组件）：提供了一套零件生产工具及流程（脚手架、可视化调试工具，开发、调试及使用流程）；
- 图纸（页面结构内容）：提供了画图工具（可视化编辑器），模板；
- 生产线（站点）：提供了自动化生产设备（解析器，渲染引擎），自建物流（资源管理器）；
- 配套（平台）：提供建设各类仓库所需要的场地（按组织，团队，应用提供了相应的独立的空间），交易市场（商店）。

Foxpage 作为一款前端低代码的基建产品，会持续的探索围绕着低代码开发的工程化体系，给低代码开发带来更好的体验。

4.3 工作流的改变

让各个职能可以在线合作，改变了传统线性接力式的工作流程，减少了各职能之间不必要的依赖，缩短开发工期。有人用游戏中的“圆桌范式”来形容这种工作关系。我觉得有点像“手术台”的形式，这种形式更强调合作。



作为框架会有约束，但更多的是支撑。在享受低代码带来益处的同时也要适应它带来的一些变化及约束。

五、组件化

组件是低代码类产品非常重要的部分，可以说组件化的结果会直接影响项目低代码开发的体验。

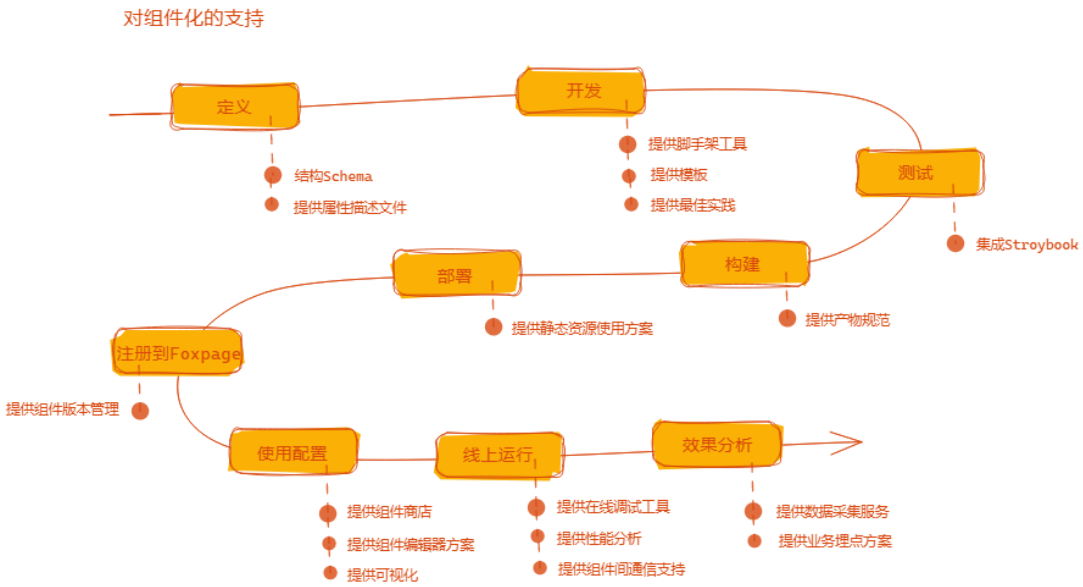
Foxpage 作为前端低代码框架，理论上是不涉及到业务部分的。开发者需要站在整个项目角度结合业务去考虑怎么组件化，如哪些模块需要组件化来降低重复开发的成本，哪些不适合组件化；组件的主体内容是静态数据，还是通过请求接口获取的动态数据；组件数据部分的复杂逻辑是否可以交给后端；哪些信息根据具体的业务是要做成可配置的会更灵活。

组件的粒度粗细怎么控制？粒度越细可能越灵活，但是组合起来可视化搭建过程中就会越复杂，增加使用成本。如果做的过粗，复用度就会降低，增加开发成本等。开发者需要结合自身业务综合地去考虑这些问题。

5.1 提供的支持

当开发者按照自己的设计去完成组件化落地时，Foxpage 会提供一些支持。包括组件开发的脚手架，可视化调试，本地构建等工具及经过大量实践后整理出的一些相对完善的流程。

我们关注组件化落地过程中整条链路上的开发和使用体验，从组件的定义，开发实现，调试，测试，部署，注册，可视化配置使用及效果分析。同时也会提供一些自己的实践供开发者参考。



当然在对组件化做支持的过程中，会有一些原则：

- 不去影响开发者现有的开发习惯
- 不去改变前端项目现有的工程化体系，只基于现有的体系做补充支持
- 不去破坏组件的通用性，组件不是为 Foxpage 特制的，不能影响组件在别的项目中使用

除了用户需要为自身业务实现的组件外，Foxpage 会提供一些基础且比较通用的组件，同时还会结合行业比较流行的 UI 库做一些组件化实践，提供给用户多样的选择。

5.2 不提供的支持

Foxpage 不提供组件项目静态资源存储、部署及 CDN 内容分发相关的功能。对于静态资源存储，开发者可以按照自身的情况来选择通过云服务自建或者使用第三方托管服务。Foxpage 也不提供组件项目的 CI/CD 相关的功能，因为 Foxpage 本身没有对应的构建和部署环境。开发者可以借助市面上成熟的工具或服务来完成 CI/CD，整个过程 Foxpage 不会介入。当然作为低代码配套基建的重要部分，我们未来也会去提供这部分的实践。

六、扩展性

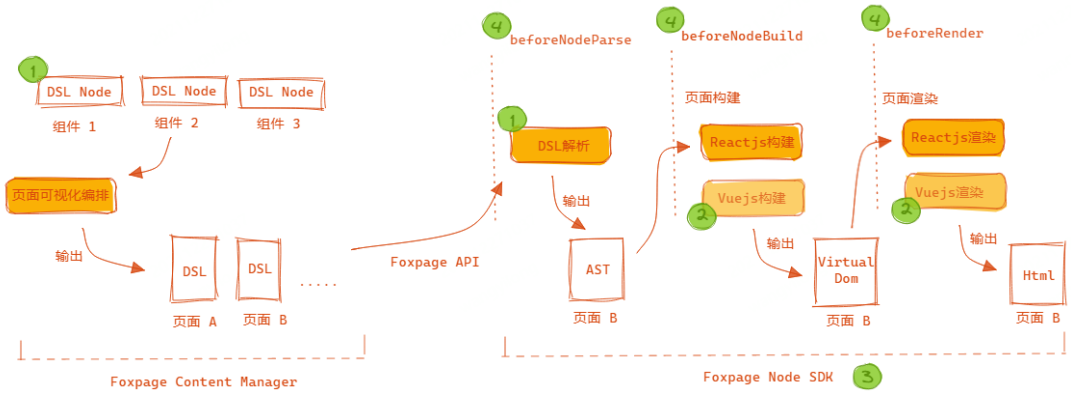
扩展能力是 Foxpage 框架最重要的能力之一，主要体现在对多场景，多技术栈，多端等的支持上。

在了解扩展能力之前，我们再回过头看看 Foxpage 的 DSL 部分。DSL 本身除了用作描述之外没有承担任何其他的职责，实体是一份 JSON 格式的数据。技术上不涉及到任何的编程语言，框架，运行环境等。它的能力会体现在对应的场景的具体实现上。

拿建筑行业举例来说 DSL 就好比施工图纸，它是一门工程语言，建筑工人可以根据施工图

纸在不同的环境和地点，选用不同的建筑材料搭建出建筑物。也正是因为如此才使得框架有了支持多场景，多技术栈，多端的可能。

那扩展能力具体是怎么体现出来的呢？我们可以先从 DSL 的视角，通过对 Web 页面的制作和渲染这个过程的简单剖析，看下 Foxpage 是怎么运作的，其中涉及到扩展的几个关键点已标注。



注 1：DSL 部分可以随意的扩展，只要提供对应 DSL 的解析器即可。

注 2：Foxpage Node SDK 内置 Reactjs 框架，主要负责页面的构建和渲染。如果想用其他的前端框架，可以提供 SDK 中对应的渲染引擎部分的实现即可。理论上可以选用任何你想用的前端框架，这是对多前端技术栈的支持能力的体现，当然要做到多技术栈支持还有其他工作需要做，如可视化，客户端渲染等环节也要有对应技术栈支持。框架后面也会提供对 Vuejs 的支持。

注 3: 对多端，多场景的支持，Foxpage 提供了对服务端的 Node SDK 和浏览器端的 JS SDK 的实现。不同端的实现会有差异，但大体都包含资源管理，DSL 解析器，渲染引擎。其核心架构中的无头 CMS 的设计，使得整个框架拥有了更强的扩展性。针对不同的端或场景可以提供对应的的实现。具体可以参考文档进阶之路频道中 Node SDK 实现部分。

注 4：我们在 DSL 的解析，页面的构建及渲染的过程中预置了钩子并暴露出了很多钩子函数（图中的只是一部分例子），结合框架提供的插件机制，开发者可以很方便的介入到整个运行过程中去，结合业务需求做一些定制，这样大大提高了可扩展性。

从上图的运行的过程中看，整个框架都是围绕着 DSL 来建设的，可以这么说 DSL 的描述能力有多强，框架扩展的能力就有多强。

七、可视化搭建

可视化积木式的搭建其实是网页低代码开发的一个很基础的能力，Foxpage 也提供了相应的功能，与其他同类的产品不同的地方是我们提供了好几种可视化编辑器，有针对富文本类的，Markdown 类的，有页面类的，有画图类的等等，根据不同的要编辑的内容的类型可以选择对应的编辑器。

这一部分我们也预留了常用的接口，甚至可以开放给用户定制内容编辑器，这也是 Foxpage 支持多场景能力的体现。当前 Foxpage 内置的只有针对网页内容的可视化编辑器，后续会考虑继续开发其他类型的内容编辑器。

可视化能力不仅体现在编辑的过程，还会在一些如历史版本快照，组件在线预览，页面点击曝光埋点等功能上有所体现。目前可视化核心部分已经组件化，这样可以给任何其他需要可视化功能的模块带来便利的，一致的可视化体验。

八、平台化

Foxpage 提供支持多应用的开发和管理，是 SaaS 产品的一种“多租户”的模式，在面对多个前端项目要使用 Foxpage 框架时，无需部署多套服务。同时为了各个应用之间的信息流通分享，还提供了商店模式，通过商店可以上架应用，页面模板，组件，变量等内容，从而大大的提高的各类内容的复用度。

开发者在完成一个前端项目低代码框架接入工作后，可以将自己项目的应用上架商店，通过这种方式开放自己应用的前端低代码开发权限，这个时候其他使用低代码开发的用户（可以是编辑，运营或开发者，以下简称用户）也可以在你的应用上做低代码开发。

这个过程中开发者变成了提供某一个项目的低代码开发的服务者，开发者接入低码框架后让项目拥有了低代码开发的能力，通过平台将这种能力赋予用户，这个过程中开发者相对于用户来说变成了服务方。

开发者还可以专门开发组件，提交到商店后供其他的开发者在自己的应用中使用。使用低码开发的用户可以将自己开发的作品，如搭建的 xxx 页面上架到商店供其他用户克隆使用等等。
Foxpage 为各种角色提供了一个在线合作的平台，同时提供在线协作能力，提高各个角色之间的合作效率。

九、项目实践

Foxpage 已经在 Trip.com 内部多个项目中使用，且已经稳定运行多年。这里结合几个有特点的项目来介绍下项目实践，这些项目本身的页面数非常多且结构各异，页面内容变更非常频繁。应用也都是 MPA（Multi-page applicaiton）多入口应用，业务需求决定需要支持多种渲染模式 SSR，CSR，SSG。

其实每一个项目的低代码实践，不仅仅只是解决单个项目的问题，而是解决了这一类项目的问题。下面三个实践都各有特点，在实践的过程中也遇到了各种问题，特别是在对组件化的支持上和不同系统之间的对接上，总之就是框架结合具体业务场景落地上还有各自的路要走。这里不做详细介绍只简单让大家对 Foxpage 的使用场景有个直观的感受。

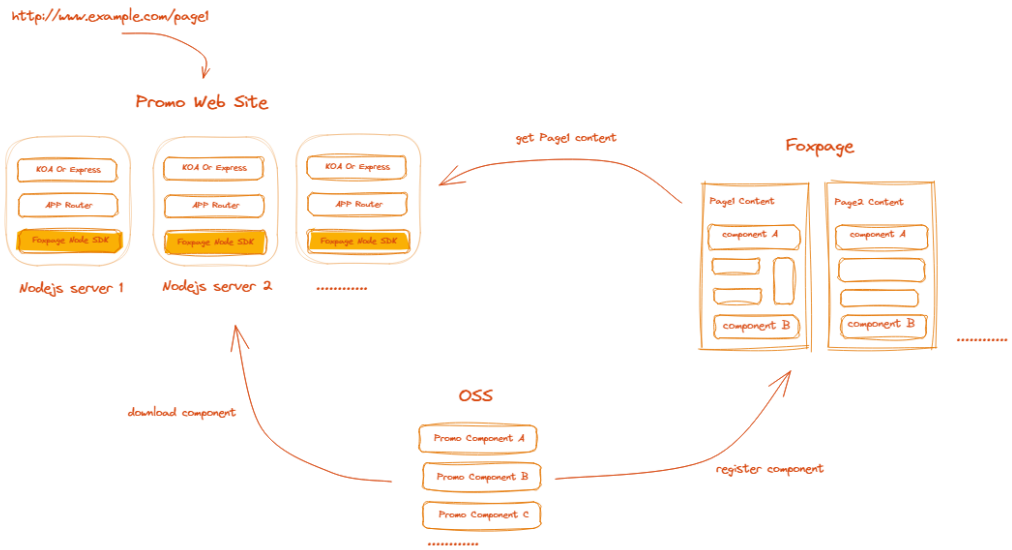
9.1 促销页面

为了营销需求而搭建的落地页面，有各种各样的玩法，主要有产品促销，好友拉新，秒杀，抽奖等活动。

需求特点：

- 短，活动有一定的时间范围，一般不会太长，由其性质决定；
- 频，活动的需求有确定性的，周期性的和临时性的，需求繁多，玩法各异；
- 快，一般活动上线时间都会有要求，可能配合一个热点事件，如果是突发类的，需要快速响应；
- 高，看活动需求，每个活动都可以看成单个大小不一的项目，支持成本比较高。

接入 Foxpage 后：



案例：Trip.com Eventsale 系统

简介：Eventsale 系统是 Trip.com 的活动配置系统，包含活动的基础信息，促销信息，玩法

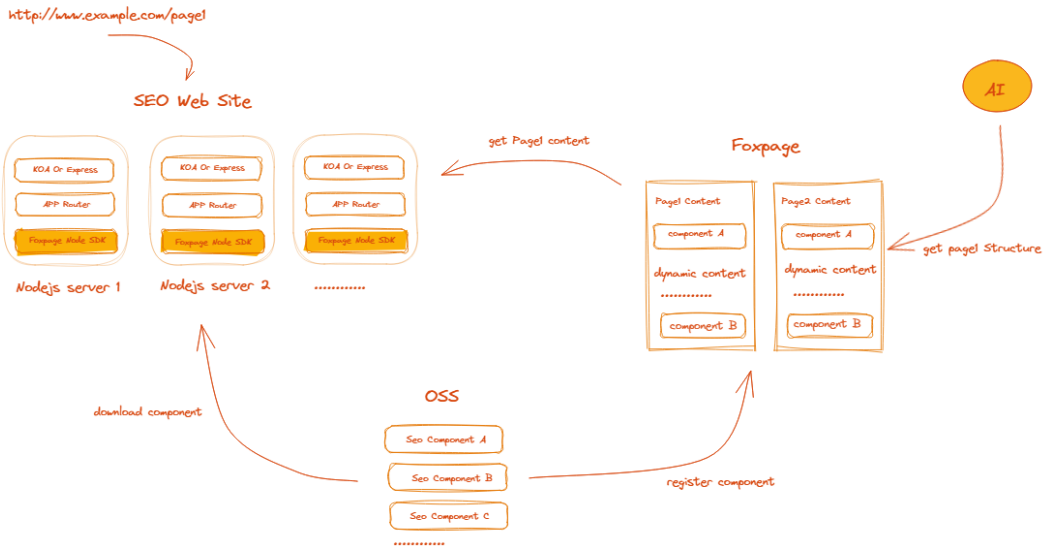
信息，活动页信息等配置。其中页面搭建能力由 Foxpage 提供。

现状：截至 2021 年中为止，大约有数千张的活动页面通过活动配置系统制作完成。

9.2 SEO 页面

为了搜索引擎优化提供的一些页面。大部分页面的主体内容都是动态生成的，不同的关键词页面的结构和内容都不一样。页面模块需要有一定的动态性，会根据模块点击和曝光动态排序以及控制是否展示。

接入 Foxpage 后：



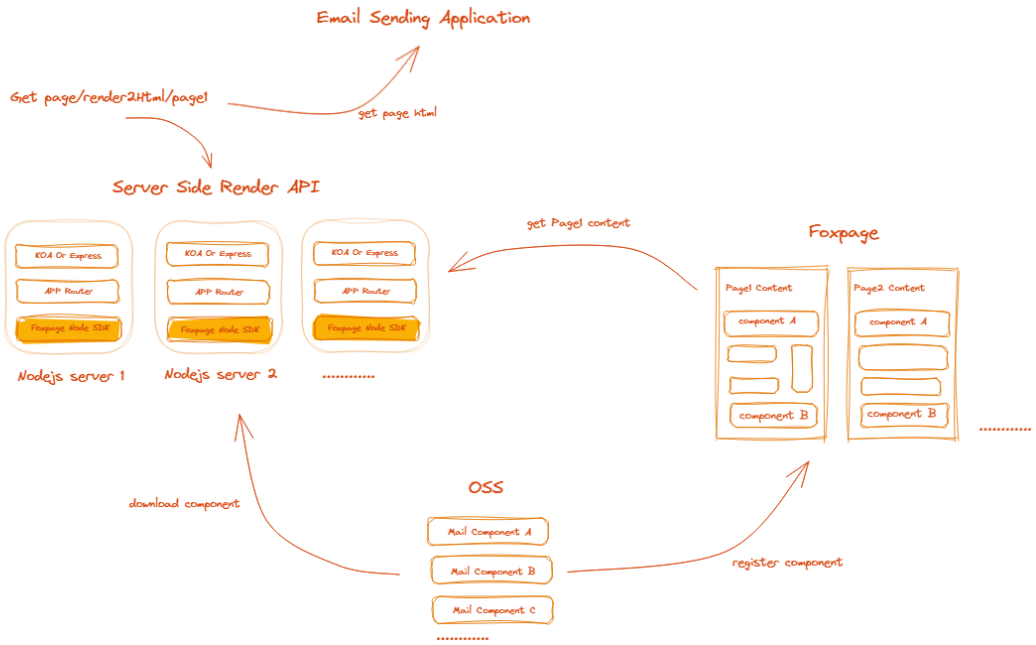
案例：Trip.com SEO 管理平台

简介：SEO 管理平台管理着 Trip.com Hot 频道的内容，包括关键词管理，页面 TDK 信息，结构信息，内容等。其中页面搭建能力由 Foxpage 提供。

现状：截至 2021 年中为止，SEO 平台目前由 Foxpage 框架生成的页面大概有 百万级的页面，页面部分的主要模块都是通过算法动态生成。

9.3 邮件页面

在邮件页面发送这个场景中，传统的方式是前端切图，将 HTML 交付给到后端，后端再结合模版引擎做数据绑定，然后调用发送渠道发送。这个过程中前后端没有分离，前后端合作低效。结合 Foxpage 方案后很好地解决了痛点。



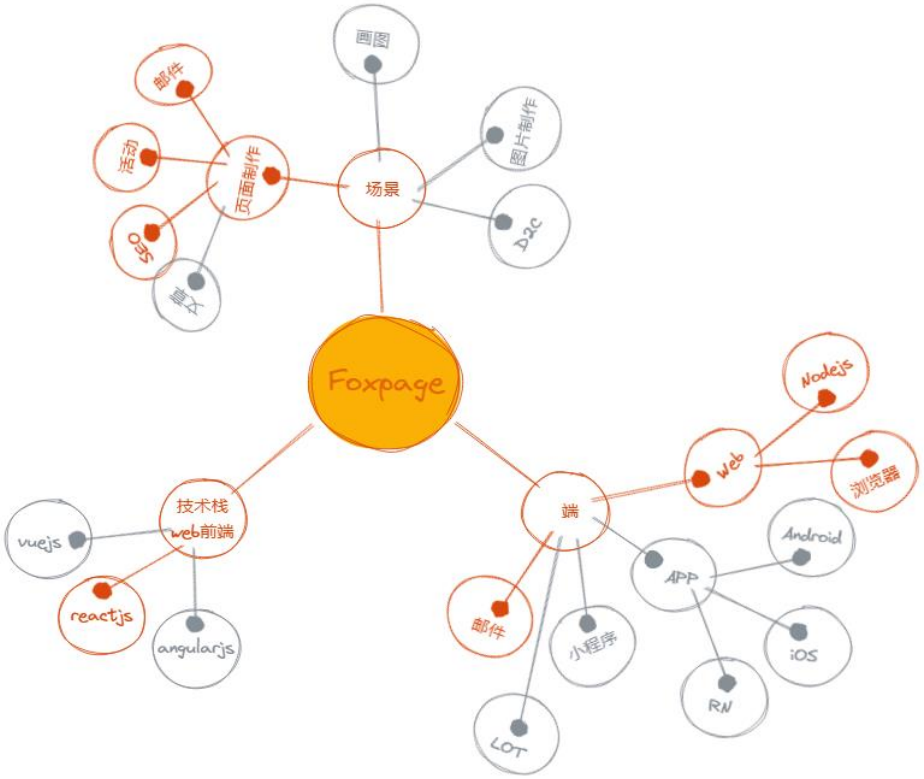
案例：Trip.com MessageHub 系统

简介：MessageHub 系统是 Trip.com 触达用户的消息系统，包含短信，邮件，站内信等内容的管理和发送。其中邮件页面搭建能力由 Foxpage 提供。

现状：截至 2021 年中为止，通过邮件配置系统制作的各类邮件页面大约数千张左右， SSR 服务平均每天调用量在几百万次，静态页面渲染耗时 99 线在 60ms 左右。

9.4 更多的前端使用场景

适用的场景除了我们已经实践过的其实还有很多，大家可以继续探索。



- 网页搭建：后台系统页面（表单，表格，菜单），数据可视化看板，贺卡，调研问卷，弹窗广告，富文本页面，Markdown；
- 图片制作：海报；
- D2C：各类 Pro code 生成，橱窗代码。

9.5 不适用的场景

主要是从体验和使用成本角度总结出以下不适用场景：

- 交互逻辑复杂的前端项目；
- 定制化程度高且不易组件化的前端项目；
- 页面内容或结构不会频繁变化的前端项目。

两个需要用户注意的事项：

- 在线搭建不等于在线写代码，不要把低代码类产品变成了在线的 IDE；
- 不要为了低代码而低代码，有些项目使用低码的方式不仅成本高，而且体验也差。

低代码虽好却并非适合所有的项目，根据实际情况谨慎使用。

十、未来规划

目前 Foxpage 低代码框架才完成基础部分的功能开发，离一个成熟完善的低码框架还有很

长的路要走，2022 年我们将从以下几个方面梳理规划：

10.1 框架迭代

1）版本 1.0（2022 H1）：提供低代码开发所需的基础功能，将于上半年发布 1.0 版本

2）版本 1.x（2022 Q2）：提供对国际化，系统权限，Debugger 工具，自定义组件等的支持

3）版本 2.0（2022 Q4）：对 Foxpage Admin UI 改造，重点会在交互部分，视觉部分也会升级，同时做功能上的简化和流程上的优化

4）版本 2.x：提供对系统集成，Serverless 方案，页面数据分析等的支持

10.2 场景拓展

除了网页制作部分已有的使用场景，还会探索图片生成，主要会在海报制作方向

10.3 配套建设

（1）组件：

- 继续丰富基础组件库；
- 提供邮件场景下的组件化最佳实践；
- 对市面成熟的 UI 库支持。

（2）静态资源服务：结合云服务的对象存储服务，提供静态资源服务建设的最佳实践

（3）托管服务：结合云服务提供 SaaS 产品

开源 | 携程度假零成本微前端框架-零界

【作者简介】工业聚，携程高级前端开发专家，react-lite, react-imvc, farrow 等开源项目作者。乐文，携程前端开发工程师，专注组件化开发、前端性能优化。

一、 前言

1.1 微前端的含义

在研发一个系统的初期，我们可以把所有代码放到一个项目中。随着企业的发展，业务逻辑越发复杂和专业化，又会细分出不同的研发团队，独立负责其中某一部分。

每个开发团队有他们各自的迭代节奏，很难在耦合的同一个项目中，满足所有团队的需求。我们很自然地会将整个系统拆解到多个子应用/子项目中，他们可以独立开发、独立部署，但共同协作支撑了系统的整体功能。

当上述系统拆解过程，发生在后端时，它被称之为——微服务；当它发生在前端时，则被称之为——微前端。

从某种意义上，微前端是默认值，不需要额外的努力。浏览器一开始就实现了通过超链接的方式，支持多个 HTML 页面之间跳转。

Tim Berners-Lee, a British scientist, invented the World Wide Web (WWW) in 1989, while working at CERN. The Web was originally conceived and developed to meet the demand for automated information-sharing between scientists in universities and institutes around the world.

1989 年，英国科学家蒂姆-伯纳斯-李在欧洲核子研究中心工作时发明了万维网（WWW）。万维网最初是为了满足世界各地大学和研究所的科学家之间自动分享信息的需求而构思和开发的。

Web 自它被发明开始，就已经是一种服务于跨团队（不同大学、不同科学组织）之间的沟通与协作的信息技术。

但是，朴素的页面跳转，往往会在页面过渡阶段产生白屏，在体验上不能满足我们的需求。

因此，当我们说“微前端”时，我们想要达到的目标是：

- 不同的前端团队，可以独立开发和部署他们的应用，满足自己的迭代需求；
- 多个前端子应用之间的协作与切换，不应该产生不可接受的用户体验下降。

1.2 微前端的类型

我们可以把微前端按照其拆解的颗粒度，分成：

- 页面级微前端（page-level）：每个子应用独享一个页面，子应用之间的切换就是页面之间的跳转/切换。
- 区域级微前端（section-level）：在同一个页面中，存在两类区域：
  a) 共享区域，如顶部菜单栏、侧边栏等，由所有子应用共享。
  b) 切换区域，通常作为主体内容呈现，子应用在该区域做局部切换。

页面级微前端（page-level）是浏览器的默认功能，但体验不佳；因此，当前大部分微前端框架，致力于区域级微前端（section-level），代表框架有 Qiankun，Single-Spa 等。

区域级微前端的主要实现思路，可以粗略概括如下：

- 代理或劫持 window 环境，让多个子应用及其依赖的前端框架，可以互不干涉地独立运行；
- 每个子应用注册了“创建”与“销毁”等生命周期，等待主应用根据 url 去驱动和调度它们。

区域级微前端（section-level）可以很好地解决某一类微前端场景（如复杂的后台系统），子应用恰好拥有相同的界面风格，甚至相同的 Layout，如顶部菜单栏、侧边栏等模块，只有内容主体部分有差异。

然而，在另一些场景中，我们可能仍然需要页面级微前端（page-level）。

- 子应用之间拥有不同的 UI 风格，甚至不同的 Layout，它们之间的切换，就是整页的切换，而不是局部的切换。
- 我们不希望子应用为了迎合区域级微前端（section-level）的接入要求，而做出巨大的调整，甚至改变开发方式。
- 子应用需要同时存在，并且可以在切换过程中，以滑入/滑出的动画方式转场，在回退过程中，可以自动保持滚动条位置等。
- etc。

今天我们要介绍的——零界微前端，就属于上述页面级微前端（page-level），它克服了子应用切换过程的体验问题。

二、 零界介绍

2.1 设计理念

- 成本可控。接入成本不应该随着应用的接入数量增加而指数级地上升，接入 2 个应用和接入 100 个应用考虑的问题应该是一致的。
- 真正的技术无关。无论应用使用的是什么技术栈、渲染方式是 SSR 还是 CSR、应用类型是 SPA 还是 MPA，都可以无缝接入。
- 零耦合。微应用和主应用之间、微应用和微应用之间，完全没有依赖关系。应用的接入和退出不会对应用本身和已经接入的应用带来任何副作用。

2.2 基本工作原理

零界作为页面级微前端（page-level）解决方案，在架构上和区域级微前端（section-level）大体一致，但在实现方式上有所不同。

零界采用经典的基座应用 + 配置的方式来管理子应用。

在零界中，基座又叫做 shell 。shell 只做两件事：存放微应用和调度微应用。

所有微应用都加载在 iframe 中，零界通过 shell 管理多个 iframe 的加载和切换。

然而，iframe 会带来路由不同步的问题。零界通过 history api 如 pushState 和 replaceState，将当前激活的页面的地址，同步到浏览器地址栏里的 location 中，保持了 URL 一致。

与市面上微前端框架最大的不同是，在零界中没有生命周期、Event Bus 等复杂的概念，而是监听微应用的跳转行为，通过将跳转记录存储在浏览器中， 把所有的微应用串联起来。每一次微应用的跳转，新的页面会以 iframe 的形式加载至零界微前端，并且不会立即释放之前微应用的内存，可以快速回退。为了避免过多的 iframe 导致页面卡顿，零界限制了 iframe 的最大数量。

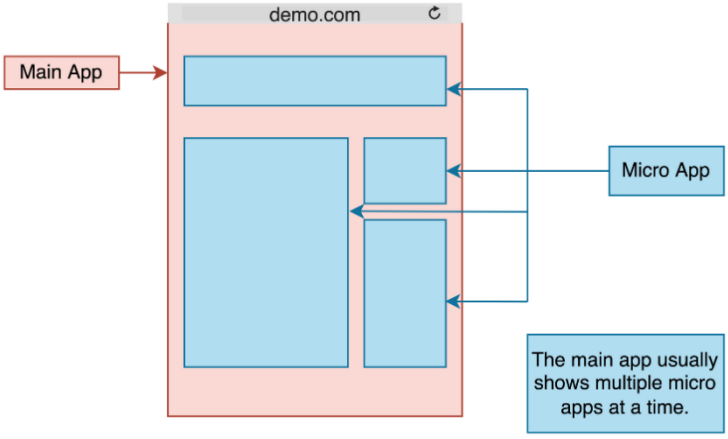
特点：

- 无需改造原有代码。技术栈无关，无需担心前端开发的难度。
- 几乎零接入成本。每个页面只需引入一个 script 文件，即可加入零界微前端机制。
- 无刷新切换页面。提供无刷新页面切换的 SPA 体验，给用户一致性的体验。
- 安全可靠。所有页面可随时退出零界微前端机制，回归原始状态。
- 状态同步。刷新页面不会丢失路由状态，页面回退更快展示，并保留前一页的滚动条以及页面状态。
- 完美隔离。完全隔离了每个页面的 css 和 js，避免了各个应用之间的变量污染。

2.3 为什么是 iframe

构建区域级微前端（section-level）时，由于 iframe 使用简单、自带进程级别隔离等特性，许多开发者都曾考虑使用 iframe 构建微前端，但最终都不约而同地放弃了这个方案。

让我们结合下图，再回顾下利用 iframe 构建区域级微前端（section-level）可能会带来的具体问题。



（1）DOM 割裂严重。蒙层只能覆盖其中一个微应用（一块蓝色区域），无法遮住整个应用（整个粉色区域）；

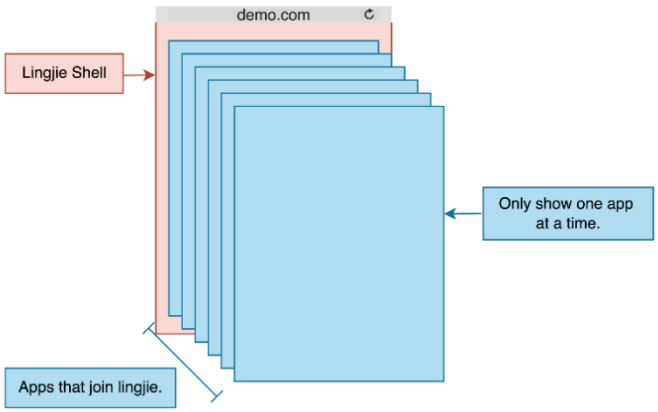
（2）通信困难。不同的微应用同时存在于一个页面，微应用之间需要额外的通讯，而 iframe 只能通过 postmessage 传递序列化的消息，无法满足需求；

（3）加载慢。一个页面中通常存在多个微应用，微应用会频繁挂载、卸载，iframe 每一次加载都是一次上下文的重新构建；

（4）路由状态丢失。刷新页面后 iframe 会回到首次加载的状态；

可以看出，这些痛点是由 iframe 自带的特性导致的，不只是针对区域级微前端（section-level），而是使用 iframe 时要考虑的通用性问题。

现在，我们再站在页面级微前端（page-level）的角度，逐一思考上面的问题：



（1）DOM 割裂严重；无需解决✅ 如上图所示，所有微应用的展示都是全屏的，不存在蒙层无法全局展开的问题。

（2）通信困难；无需解决✅ 各个微应用之间原本就属于完全不同的应用，所以并不用侧重于应用之间的通信。

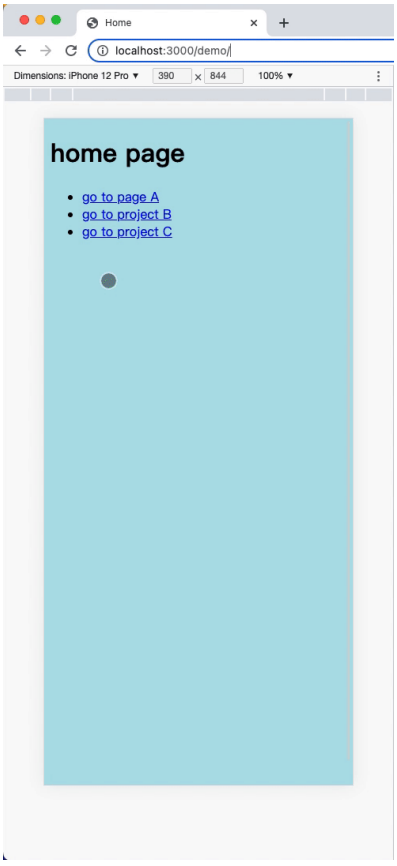
（3）加载慢；无需解决✅ 在页面级微前端（page-level）中，每次进入页面只会加载一个微应用（iframe）。

（4）路由状态丢失；问题同样存在于页面级微前端。

也就是说，我们只需要解决浏览器历史记录同步的问题，就可以最大化利用 iframe 的特性，这就是零界选择 iframe 管理微应用的原因。

三、 如何使用

3.1 基本使用



如上图所示，假设我们现在需要做到上面展示的 home Page，page A，page B 和 page C 这 4 个页面无刷新切换的效果，应该如何实现呢？

如果他们是同一个应用的不同组件，则可以通过 React 或 Vue 的 TransiitonGroup 等组件快速实现。

但是，如果他们是 4 个朴素的 HTML 页面/应用，可能很难通过传统前端框架实现，甚至，

大多数区域级微前端（section-level）也无法完成。

而在零界中，每个微应用都是全屏的，分别存放在 iframe 里，可以通过操作 iframe 的方式来操作微应用，就像把样式叠加在普通的 DOM 元素上一样。

零界针对 H5 页面模拟了 Native App 中 WebView 切换的机制，也就是上图的切换效果，接入零界即可开箱即用。

让我们来看下如何搭建零界微前端。

第一步，创建零界 shell。

假设 4 个页面的地址分别为：

- localhost:3000/demo/index.html
- localhost:3000/demo/pageA.html
- localhost:3000/demo/pageB.html
- localhost:3000/demo/pageC.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Lingjie Shell</title>
  </head>
  <body>
    <!-- 配置接入零界的应用 -->
    <script>
      window.__lingjie_shell_config__ = {
        rules: [
          {
            // 微应用的路径匹配规则
            test: "/demo",
            // 微应用的域名入口
            originList: ["http://localhost:3000"]
          }
        ]
      };
    </script>
    <!-- 引入零界shell脚本 -->
    <script src="https://unpkg.com/lingjie/dist/shell/lingjie-shell.umd.js"></script>
  </body>
</html>
```

如上图所示，无需通过 npm/yarn 安装，也无需调用任何函数，只需要对一个普通的 HTML 页面做两个改动就可以完成 shell 的搭建：

- 设置接入零界的微应用的匹配路径。
- 引入零界 shell 脚本，引入后就可获得零界的能力。

第二步，接入零界。

在 4 个应用的 HTML 中，分别在 head 标签里写入下面的代码

```
<!-- 配置开启、关闭接入零界微前端，非必填，默认为开启 -->
<script>
  window.__lingjie_config__ = {
    disable: false
  };
</script>
<!-- 引入零界shell脚本，目的是跳转至零界 shell 页面-->
<script src="https://unpkg.com/lingjie/dist/page/lingjie-page.umd.js"></script>
```

我们在接入零界的微应用上，也只做了两个改动：

- 配置开启/关闭零界。
- 引入零界 page 脚本。

以上就是构建零界微前端的所需的所有代码。

通过这样接入有以下 3 个好处：

- 没有学习成本，直接引入。
- 不影响应用本身的 SEO。
- 在零界中以子应用运行和应用独立运行没有区别，他们的路由路径一致。

另外，我们可以发现，当微应用不能匹配 shell 中配置的路径，或者微应用关闭了零界时，都无法接入零界。

所以，当一个应用接入零界后导致无法正常访问时，可以通过配置化的方式远程关闭零界，这个页面就会退化为普通页面，而不必等待 shell 去改变配置。并且，这样既不影响零界中已有的微应用跳转，也不影响零界中的微应用跳转至这个页面。

3.2 零界进阶

上文展示了朴素页面的切换，体验了零界在 H5 页面的滑入滑出的效果。然而朴素页面并不能满足我们实际的需求。

想象一下这样一个场景：有多个 CSR 应用，他们共享同一个 Sidebar，但拥有不同的 Content，直接展示它们都会有一段白屏，我们希望在切换时，消除白屏，直接看到更完整

内容的页面。

这是一个常见的 B 端项目优化需求，区域级微前端（section-level）和页面级微前端（page-level）都可以提供解决方案。

在现代 web 开发模式中，通常将页面中的内容按功能、区域划分为不同的组件，以提高代码复用性、扩展性。因此 Sidebar 和 Content 可以视为两个不同的组件。

区域级微前端（section-level）和页面级微前端（page-level）对应用中的组件有不同的处理方式，产生了不同的优化策略：

- 区域级微前端（section-level）以组件（区域）为单位，拆分原应用，并重构组件。之后，会从组件的角度，考虑如何在基座应用中主动挂载、卸载，达到想要的效果。
- 页面级微前端（page-level）以页面为单位，在不改动原有应用组件的情况下，聚合所有应用。所以聚合之后，会从应用的角度，考虑如何被动式地对内部组件进行优化。

通过区域级微前端解决，大概分为 4 步：

（1）将每个应用中的 Sidebar 和 Content 拆分出来。

（2）把每个 Content 作为一个微应用单独部署，并配置基础信息、添加生命周期。

（3）将 Sidebar 直接放入基座应用中，或者，作为一个微应用单独部署。

（4）创建基座应用，注册所有的微应用。

在切换应用时，只需卸载前一个应用的 Content，加载下一个应用的 Content 既可，共享的 Sidebar 部分并没有变动，完全模拟了在一个应用中的切换体验。

但是，受限于样式隔离、运行性能、子应用保活等因素，需要在市面上现有的区域级微前端（section-level）解决方案中权衡，很难找到完全满足诉求的方案。并且无论是哪种改造方案，都需要较高改造的成本，这个成本还会随着应用的数量指数级上升。

让我们来看下页面级微前端会如何解决。

零界提供了另一种思路，不侵入式地改变原有应用的前提下，优化应用之间的交互。

改造分为 2 步：

（1）创建零界 shell，配置接入微应用的路径。

（2）在所有接入的应用中，引入零界 page 脚本。

至此，和之前展示的朴素页面切换效果一致，但是页面的跳转还是产生割裂感。

```
<script>
  window.__lingjie_shell_config__ = {
    rules: [
      {
        test: "/demo",
        originList: ["http://localhost:3000"],
        // 等待元素加载
        waitFor: "#sidebar",
        // 超时时间
        timeout: 5000
      },
    ]
  };
</script>
```

为了提升用户体验，在零界微前端切换页面时，顶部会展现 Progressbar，表示页面切换的进度。

并且，受 Puppeteer 和 Playwright 这两个 e2e 测试工具的启发，零界模拟了里面的 waitFor 功能，如上图所示，表示等待 Sidebar 组件展示到页面后，再进行页面切换。这样当多个应用在拥有相同 Sidebar 的页面之间切换时，Sidebar 的部分在视觉上是固定的，只有 Content 发生变化，通过这种方式在多页应用中获得沉浸式的体验。

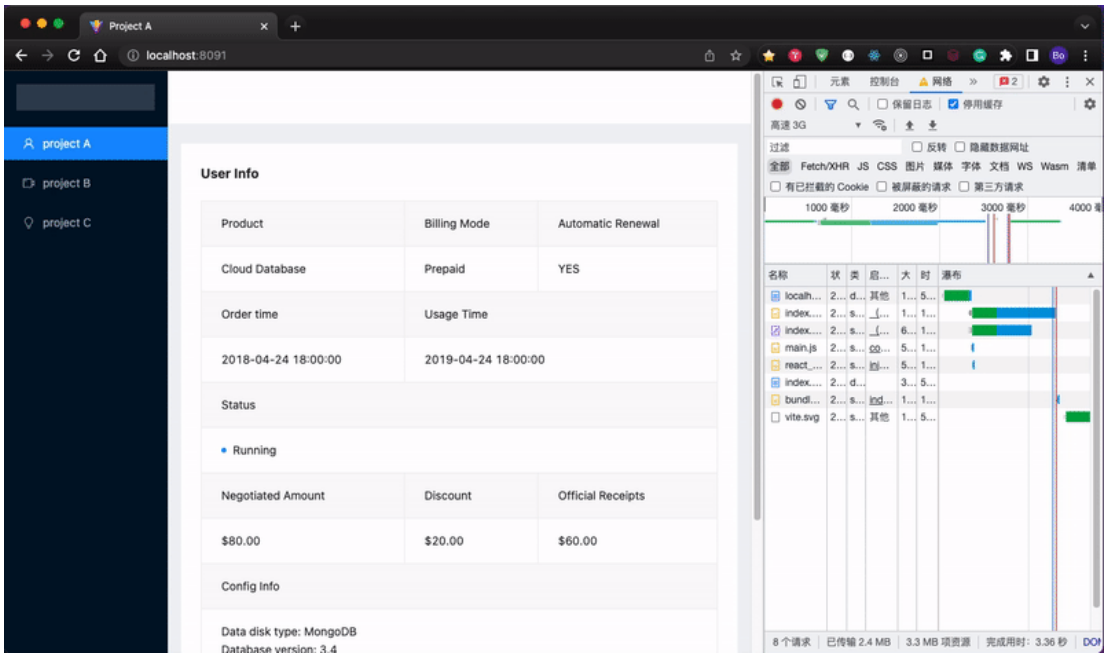
不仅如此，还可以通过 timeout 设置最长等待时间，一旦超过等待时间，页面则会强制切换。

这种优化方式带来以下几个好处：

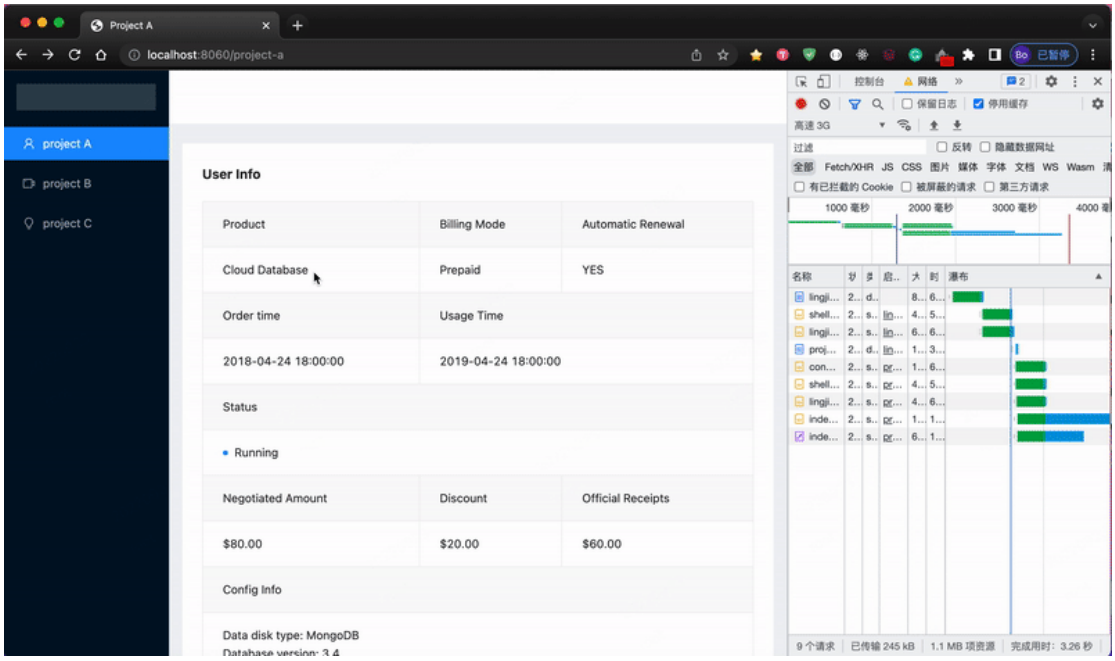
- 应用的 Content 和 Sidebar 的交互，并不需要额外的机制，因为它们本来就是同一个应用的不同组件。
- 应用不必须携带相同的 Sidebar，随着业务的发展需要可以更灵活地决定自己的 UI，零界不会是应用扩展的瓶颈。

让我们来对比下优化前后的效果，为了能更直观地感受其中的差异，我们把网速调整为高速 3G。

未经过任何优化，每个页面都是不同的应用（网速：高速 3G）。



经过零界优化后（网速：高速 3G）。



可以非常明显地看出，经过零界优化后，多页应用的跳转更为流畅，并且支持快速回退页面。

细心的读者可能发现，这两个动图的 URL 不一致。这里仅展示零界带来的优化效果，通过本地 Node 代理服务器完成零界跳转，所以和应用原有 URL 不同。在开发企业级项目时，通常不存在这个问题，可以通过 SLB 等方式快速解决。

另外，值得一提的是，零界文档也是基于零界微前端构建的，可以直接体验零界在 MPA 中切换的效果，有兴趣的话可以查看零界文档。

总结

至此，我们介绍了零界的基本原理和使用方式，其适用场景可以概括如下：

- 资源整合。组合多个已有的中大型应用，无需重构。
- MPA 优化。将 MPA 的跳转提升至 SPA 的体验。
- H5 端体验优化。低成本通过框架的能力达到 WebView 的切换效果。

虽然零界是页面级微前端（page-level）解决方案，但这并不表示它和区域级微前端（section-level）是冲突的。如果你想快速体验微前端，或者你的项目由于现有微前端框架可能带过高的成本而迟迟没有落地，那么可以先尝试使用零界，在零界中可以随时退出，不会带来任何副作用。若尝试后不能满足需求，再考虑接入颗粒度更细致的区域级微前端（section-level）。

如果你对这个项目感兴趣，欢迎一起来为零界做贡献。

携程基于 GraphQL 的前端 BFF 服务开发实践

【作者简介】工业聚，携程高级前端开发专家，react-lite, react-imvc, farrow 等开源项目作者。兰迪咚，携程高级前端开发专家，对开发框架及前端性能优化有浓厚兴趣。

一、前言

过去两三年，携程度假前端团队一直在实践基于 GraphQL/Node.js 的 BFF (Backend for Frontend) 方案，在度假 BU 多端产品线中广泛落地。最终该方案不仅有效支撑前端团队面向多端开发 BFF 服务的需要，而且逐步承担更多功能，特别在性能优化等方面带来显著优势。

我们观察到有些前端团队曾尝试过基于 GraphQL 开发 BFF 服务，最终宣告失败，退回到传统 RESTful BFF 模式，会认为是 GraphQL 技术自身的问题。

这种情况通常是由于 GraphQL 的落地适配难度导致的，GraphQL 的复杂度容易引起误用。因此，我们期望通过本文分享我们所理解的最佳实践，以及一些常见的反模式，希望能够给大家带来一些启发。

二、GraphQL 技术栈

以下是我们 GraphQL-BFF 项目中所采用的核心技术栈:

- Graphql

基于 JavaScript 的 GraphQL 实现

- koa v2

Node.js Web Framework 框架

- apollo-server-koa

适配 koa v2 的 Apollo Server

- data-loader

优化 GraphQL Resolver 内发出的请求

- graphql-scalars

提供业务中常用的 GraphQL Scalar 类型

- faker

提供基于类型的 Mock 数据
结合 GraphQL Schema 可自动生成 Mock 数据

- @graphql-codegen/typescript

基于 GraphQL Schema 生成 TypeScript 文件

- graphql-depth-limit

限制 GraphQL Query 的查询深度

- jest

单元测试框架

其他非核心或者公司特有的基础模块不再赘述。

三、GraphQL 最佳实践

携程度假 GraphQL 的主要应用场景是 IO 密集的 BFF 服务，开发面向多端所用的 BFF 服务。

所有面向外部用户的 GraphQL 服务，我们会限制只能调用其他后端 API，以避免出现密集计算或者架构复杂的情况。只有面向内部用户的服务，才允许 GraphQL 服务直接访问数据库或者缓存。

对 RESTful API 服务来说，每次接口调用的开销基本上是稳定的。而 GraphQL 服务提供了强大的查询能力，每次查询的开销，取决于 GraphQL Query 语句查询的复杂度。

因此，在 GraphQL 服务中，如果包含很多 CPU 密集的任务，其服务能力很容易受到 GraphQL Query 可变的查询复杂度的影响，而变得难以预测。

将 GraphQL 服务约束在 IO 密集的场景中，既可以发挥出 Node.js 本身的 IO 友好的优势，又能显著提高 GraphQL 服务的稳定性。

3.1 面向数据网络（Data Graph），而非面向数据接口

我们注意到有相当多 GraphQL 服务，其实是披着 GraphQL 的皮，实质还是 RESTful API 服务。并未发挥出 GraphQL 的优势，但却承担着 GraphQL 的成本。

```
query ($userId: ID!, $productIds: [ID!]!) {
  user: getUser(id: $userId) {
    id
    name
    email
  }
  products: getProductList(ids: $productIds) {
    id
    title
    description
    price
  }
  favorites: getFavoriteList(userId: $userId) {
    id
    product {
      id
      title
      description
      price
    }
  }
}
```

如上所示，原本 RESTful API 的接口，只是挂载到 GraphQL 的 Query 或 Mutation 的根节点下，未作其它改动。

这种实践模式，只能有限发挥 GraphQL 合并请求、裁剪数据集的作用。它仍然是面向数据接口，而非面向数据网络的。

```
type Query {
  getUser(id: ID!): User!
  getProductList(ids: [ID!]!): [Product]!
  getFavoriteList(userId: ID!): [Favorite]!
  getRecommendList(ids: [ID!]!): [Product]!
}
```

如此无限堆砌数据接口，最终仍然是一个发散的模型，每增加一个数据消费场景需求，就追加一个接口字段。并且，当某些接口字段的参数，依赖其它接口的返回值，常常得重新发起一次 GraphQL 请求。

而面向数据网络，呈现的是收敛的模型。

```
type User {
  id: ID!
  name: String!
  email: String!
  "收藏的产品列表"
  favorites: [Product!]!
}

type Product {
  id: ID!
  title: String!
  description: String!
  price: Float!
  "关联的推荐产品列表"
  recommends: [Product!]!
}

type Query {
  user(id: ID!): User!
  products(ids: [ID!]!): [Product]!
}
```

如上所示，我们将用户收藏的产品列表，放到了 User 的 favorites 字段中；将关联的推荐产品列表，放到了 Product 的 recommends 字段中；构成一种层级关联，而非并列在 Query 根节点下作为独立接口字段。

相比一维的接口列表，我们构建了高维度的数据关联网络。子字段总是可以访问到它所在得上下文里的数据，因此很多参数是可以省略的。我们在一次 GraphQL 查询中，通过这些关联字段，获取到所需的数据，而不必再次发起请求。

当逐渐打通多个数据节点之间的关联关系，GraphQL 服务所能提供的查询能力可以不断增加，最后会收敛在一个完备状态。所有可能的查询路径都已被支持，新的数据消费场景，也无须开发新的接口字段，可以通过数据关联网络查询出来。

3.2 用 union 类型做错误处理

在 GraphQL 里做错误处理，有相当多的陷阱。

第一个陷阱是，通过 throw error 将错误抛到最顶层。

假设我们实现了以下 GraphQL 接口：

```
  extend type Mutation {
    addTodo(text: String!): AddTodoResult
  }
`

TodoService.resolve('Mutation', {
  addTodo: ctx => {
    throw new Error('throw error in addTodo(..)')
```

当查询 addTodo 节点时，其 resolver 函数抛出的错误，将会出现在顶层的 errors 数组里，而 data.addTodo 则为 null。

不仅仅在 Query/Mutation 节点下的字段抛错会出现在顶层的 errors 数组里，而是所有节点的错误都会被收集起来。这种功能看似方便，实则会带来巨大的麻烦。

我们很难通过 errors 数组来查找错误的节点，尽管有 path 字段标记错误节点的位置，但由于以下原因，它带来的帮助有限：

- 总是需要过滤 errors 去找到自己关心的错误节点；
- 查询语句是易变的，错误节点的位置可能会发生变化；
- 任意节点都可能产生错误，要处理的潜在情形太多。

这个陷阱是导致 GraphQL 项目失败的重大诱因。

错误处理在 GraphQL 项目中，比 RESTful API 更重要。后者常常只需要处理一次，而 GraphQL 查询语句可以查询多个资源。每个资源的错误处理彼此独立，并非一个错误就意味着全盘的错误；每个资源所在的节点未必都是根节点，可以是任意层级的节点。

因此，GraphQL 项目里的错误处理发生的次数跟位置都变得多样。如果无法有效地管理异常，将会带来无尽的麻烦，甚至是生产事件。长此以往，项目宣告失败也在意料之内了。

第二个陷进是，用 Object 表达错误类型。

```
type Todo {
  id: ID!
  text: String!
  completed: Boolean!
}

type AddTodoResult {
  "code: 0, success; 1, error"
  code: Int!
  "error message"
  message: String!
  "success result"
  data: Todo
}

extend type Mutation {
  addTodo(text: String!): AddTodoResult
}
```

如上所示，AddTodoResult 类型是一个 Object：

- data 字段是一个 Object，它包含了查询结果；
- code 字段是一个 Int，它表示错误码；
- message 字段是一个 String，它表示错误信息。

这种模式，即便在 RESTful API 中也很常见。但是，在 GraphQL 这种错误节点可能在任意层级的场景中，该模式会显著增加节点的层级。每当一个节点需要错误处理，它就多了一层 { code, data, message }，增加了整体数据复杂性。

此外，code 和 message 字段的类型都带 !，表示非空。而 data 字段的类型不带 !，即可能为空。这就带来一个问题，code 为 1 表达存在错误时，data 也可能不为空。从类型上，并不能保证，code 为 1 时，data 一定为空。

也就是说，用 Object 表达错误类型是含混的。code 和 data 的关系全靠服务端的逻辑来决定。服务端需要保证 code 和 data 的出现关系，一定满足 code 为 1 时，data 为空，以及 code 为 0 时，data 不为空。

其实，在 GraphQL 中处理错误类型，有更好的方式——union type。

```
type Todo {
  id: ID!
  text: String!
  completed: Boolean!
}

type AddTodoError {
  message: String!
}

type AddTodoSuccess {
  newTodo: Todo
}

union AddTodoResult = AddTodoError | AddTodoSuccess

extend type Mutation {
  addTodo(text: String!): AddTodoResult!
}
```

如上所示，AddTodoResult 类型是一个 union，包含 AddTodoError 和 AddTodoSuccess 两个类型，表示或的关系。

要么是 AddTodoError，要么是 AddTodoSuccess，但不能是两者都是。

这正是错误处理的精确表达：要么出错，要么成功。

```
mutation {
  addTodoResult: addTodo(text: "Hello world!") {
    ... on AddTodoError {
      __typename
      message
    }
    ... on AddTodoSuccess {
      __typename
      newTodo {
        id
        text
        completed
      }
    }
  }
}
```

查询数据时，我们用 … on Type {} 的语法，同时查询两个类型下的字段。由于它们是或的关系，是互斥的，因此查询结果总是只有一组。

```
"data": {
  "addTodoResult": {
    "__typename": "AddTodoError",
    "message": "Something went wrong"
  }
},
```

失败节点的查询结果如上所示，命中了 AddTodoError 节点，伴随有 message 字段。

```
"data": {
  "addTodoResult": {
    "__typename": "AddTodoSuccess",
    "newTodo": {
      "id": "1",
      "text": "Hello world!",
      "completed": false
    }
  }
},
```

成功节点的查询结果如上所示，命中了 AddTodoSuccess 节点，伴随有 newTodo 字段。

当使用 graphql-to-typescript 后，我们可以看到，AddTodoResult 类型定义如下：

```
export type AddTodoResult =
  | {
      __typename: 'AddTodoError';
      message: string;
    }
  | {
      __typename: 'AddTodoSuccess';
      newTodo: Todo;
    };

declare const result: AddTodoResult;

if (result.__typename === 'AddTodoError') {
  console.log(result.message);
} else if (result.__typename === 'AddTodoSuccess') {
  console.log(result.newTodo);
}
```

我们可以很容易通过共同字段 __typename 区分两种类型，不必猜测 code 和 data 字段之间的可能搭配。

union type 不局限于组合两个类型，还可以组合更多类型，表达超过 2 种的互斥场景。

```
union GetUserResult = UserInfo | UserNotFound | InvalidPassword | Unauthorized
```

如上所示，我们把 getUser 节点的可能结果，都用 union 类型组织起来，表达更精细的查询结果，可以区分更多错误种类。

此外，union type 也不局限于做错误处理，而是任意互斥的类型场景。比如获取用户权限，我们可以把 Admin | Owner | Normal | Guest 等多种角色，作为互斥的类型，放到 UserRole 类型中。而非用 { isAdmin, isOwner, isNormal, isGuest, ... } 这类含混形式，难以处理它们同时为 false 或同时为 true 等无效场景。

3.3 用 ！表达非空类型

在开发 GraphQL 服务时，有个非常容易疏忽的地方，就是忘记给非空类型标记 !，导致客户端的查询结果在类型上处处可能为空。

客户端判空成本高，对查询结果的结构也更难预测。

这个问题在 TypeScript 项目中影响重大，当 graphql-to-typescript 后，客户端会得到一份来自 graphql 生成的类型。由于服务端没有标记 !，令所有节点都是 optional 的。TypeScript 将会强制开发者处理空值，前端代码因而变得异常复杂和冗赘。

如果前端工程师不愿意消费 GraphQL 服务，久而久之，GraphQL 项目的用户流失殆尽，项目也随之宣告失败了。

这是反常的现象，GraphQL 的核心优势就是用户友好的查询接口，可以更灵活地查询出所需的数据。因为服务端的疏忽而丢失了这份优势，非常可惜。

善用 ！标记，不仅有利于前端消费数据，同时也有利于服务端开发。

在 GraphQL 中，空值处理有个特性是，当一个非空字段却没有值时，GraphQL 会自动冒泡到最近一个可空的节点，令其为空。

Since Non-Null type fields cannot be null, field errors are propagated to be handled by the parent field. If the parent field may be null then it resolves to null, otherwise if it is a Non-Null type, the field error is further propagated to its parent field.

由于非空类型的字段不能为空，字段错误被传播到父字段中处理。如果父字段可能是 null，那么它就会解析为 null，否则，如果它是一个非 null 类型，字段错误会进一步传播到它的父

字段。

如上，在 GraphQL Specification 的 6.4.4Handling Field Errors 中，明确了如何置空的问题。

假设我们有如下 GraphQL 接口设计：

```
type Parent {
  child: Child!
}

type Child {
  child: Grandchild!
}

type Grandchild {
  value: Int!
}

extend type Query {
  parent: Parent
}
```

其中，只有根节点 Query.parent 是可空的，其他节点都是非空的。

我们可以为 Grandchild 类型编写如下 GraphQL Resolver：

```
GraphQLService.resolve('Grandchild', {
  value: (ctx) => {
    if (Math.random() > 0.5) {
      ctx.result = 1;
    } else {
      ctx.result = null;
    }
  },
});
```

我们概率性地分配 null 给 ctx.result(它表示该类型的结果)。尽管 Grandchild 是非空节点，但 resolver 里也能够给它置空。通过置空，告诉 GraphQL 去冒泡到父节点。否则我们就需要在 Grandchild 的层级去控制 parent 节点的值。

这是很难做到，且不那么合理的。因为 Grandchild 可以被挂到任意对象节点作为字段，不一定是当前 parent。所有 Grandchild 都可以共用一个 resolver 实现。这种情况下，

Grandchild 不假设自己的父节点，只处理自己负责的数据部分，更加内聚和简单。

我们用如下查询语句查询 GraphQL 服务：

```
query {
  parent {
    child {
      child {
        value
      }
    }
  }
}
```

当 Grandchild 的 value 结果为 1 时，查询结果如下：

```
"data": {
  "parent": {
    "child": {
      "child": {
        "value": 1
      }
    }
  }
},
```

我们得到了符合 GraphQL 类型的结果，所有数据都有值。

当 Grandchild 的 value 结果为 null 时，查询结果如下：

```
"data": {
  "parent": null
},
```

通过空值冒泡，Grandchild 的空值，被冒泡到 parent 节点，令 parent 的结果也为空。这也是符合我们编写的 GraphQL Schema 的类型约束的。如果只有 Grandchild 的 value 为 null，反而不符合类型，因为该节点是带 ！的非空类型。

3.4 最佳实践小结

在 GraphQL 中，还有很多实践和优化技巧可以展开，大部分可以在官方文档或社区技术文章里可以找到的。我们列举的是在实践中容易出错和误解的部分，分别是：

- 数据网络
- 错误处理
- 空值处理

深入理解上述三个方面，就能掌握住 GraphQL 的核心价值，提高 GraphQL 成功落地的概率。

在对 GraphQL(以下简称 GQL) 有一定了解的基础上，接下来分享一些我们具体的应用场景，以及项目工程化的实践。

四、GraphQL 落地

一个新的 BFF 层规划出来之后，前端团队第一个关注问题就是“我有多少代码需要重写？”，这是一个很现实的问题。新服务的接入应尽量减少对原有业务的冲击，这包括前端尽可能少的改代码以及尽可能减少测试的回归范围。由于主要工作和测试都是围绕服务返回的报文，因此首先应该让 response 契约尽可能稳定。对老功能进行改造时，接口契约可以按照以下步骤柔性进行：

- 保持原有服务 response 契约不变；
- 对原有契约提供剪裁能力；
- 在有必要的前提下设计新的字段，并且该字段也应能被剪裁。

假设之前有个前端直接调用的接口，得到 ProductData 这个 JSON 结构的数据。

```
const Query = gql`
    type ProductInfo {
        "产品全部信息"
        ProductData: JSON
    }
    extend type Query {
        productInfo(params: ProductArgs!): ProductInfo
    }
`
```

如上所示，一般情况我们可能会在一开始设计这样的 GQL 对象。即对服务端下发的字段不做额外的设计，而直接标注它的数据类型是 JSON。这样的好处是可以很快地对原客户端调用的 API 进行替换。

这里 ProductData 是一个“大”对象，属性非常多，未来如果希望利用 GQL 的特性对它进行动态裁剪则需要将结构进行重新设计，类似如下代码：

```
const Query = gql`
    type ProductStruct {
        "产品 id"
        ProductId: Int
        "产品名称"
        ProductName: String
        ......
    }
    type ProductInfo {
        "产品全部信息"
        ProductData: ProductStruct
    }
    extend type Query {
        productInfo(params: ProductArgs!): ProductInfo
    }
`
```

但这样做就会引入一个严重的问题：这个数据结构的修改是无法向前兼容的，老版本的 query 语句查询 ProductInfo 的时候会直接报错。为了解决这个问题，我们参考 SQL 的「Select *」扩展了一个结构通配符「json」。

4.1 json：查询通配符

```
const Query = gql`
    type ProductStruct {
        "原始数据"
        json: JSON
        "未来扩展"
        ProductId: Int
        ......
    }
    type ProductInfo {
        "产品全部信息"
        ProductData: ProductStruct
    }
    extend type Query {
        productInfo(params: ProductArgs!): ProductInfo
    }
`
```

如上，对一个节点提供一个 json 的查询字段，它将返回原节点全部内容，同时框架里对最终的 response 进行处理，如果碰到了 json 字段则对其解构，同时删除 json 属性。
利用这个特性，初始接入时只需要修改 BFF 请求的 request 报文，而 response 和原服务是一致的，因此无需特别回归。而未来即使需要做契约的剪切或者增加自定义字段，也只需

要将 query 内容从 {json} 改成 {ProductId, ProductName, etc....} 即可。

五、GraphQL 应用场景

作为 BFF 服务，在解决单一接口快速接入之后，通常会回到聚合多个服务端接口这个最初的目的，下面是常见的几种串、并调用等应用场景。

5.1 服务端并行

如上图顶部的产品详情和下面的 B 线产品，分别是两个独立的产品。如果需要一次性获取，我们一般要设计一个批量接口。但利用 GQL 合并多个查询请求的特性，我们可以用更好的方式一次获取。

首先 GQL 内只需要实现单一产品的查询即可，非常简洁：

```
ProductInfo.resolve('Query', {
    productInfo: async (ctx) => {
        ctx.result = await productSvc.fetch(ctx.args.productId)
    }
})

const ProductInfoHandle: ProductInfo = {
    BasicInfo: async ctx => {
        let {BasicInfo} = ctx.parent
        ctx.result = {
            json: BasicInfo,
            ...BasicInfo
        }
    },
    .....
}
ProductInfo.resolve('ProductInfo', ProductInfoHandle);
```

客户端在查询的时候，只需要重复添加查询语句，并且传入另外一个产品参数。GQL 内会

分别执行上述 resolve，如果是调用 API，则调用是并行的。

```
query getProductData(
    $mainParams: ProductArgs!
    $routeParams: ProductArgs!
) {
    mainProductInfo(params: $mainParams) {
        BasicInfo{json}
        .....
    }
    routeProductInfo(params: $routeParams) {
        BasicInfo{json}
        .....
    }
}

//主产品查询请求
[Node] [Inject Soa Mock]: 12345/productSvc 开始： 11ms 耗时: 237ms 结束: 248ms
//子产品查询请求
[Node] [Inject Soa Mock]: 12345/productSvc 开始: 12ms 耗时: 202ms 结束: 214ms
```

事实上这种方式不局限在同一接口，任何客户端希望并行的接口，都可以通过这样的方式实现。即在 GQL 内单独实现查询，然后由客户端发起一次“总查询”实现服务端聚合，这样的方式避免了 BFF 层因为前端需求变更不停跟随修改的困境。这种“拼积木”的方式可以用很小的成本实现服务的快速聚合，而且配合上面提到的“json”写法，未来也具备灵活的扩展性。

5.2 服务端串行

在应用中经常还会有事务型（增删改）的操作夹在这些“查”之中。比如：

```
mutation TicketInfo(
    $ticketParams: TicketArgs!
    $shoppingParams: ShoppingArgs!
) {
    //查询门票 并 添加到购物车
    ticketInfo(params: $ticketParams) {
        ticketData {json}
    }
    //根据“更新后”的购物车内的商品 获取价格明细
    shoppingInfo(params: $shoppingParams) {
        priceDetail {json}
    }
}
```

如上所示，获取价格明细的接口调用必须串行在「添加购物车」之后，这样才不会造成商品

遗漏。而此例中的「mutation」操作符可以使各查询之间串行执行,如下:

```
//查询门票
[Node] [Inject Soa Mock]: 12345/getTicketSvc  开始: 16ms  耗时: 111ms  结束: 127ms
//添加到购物车
[Node] [Inject Soa Mock]: 12345/updateShoppingSvc 128ms  耗时: 200ms  结束: 328ms

//根据「更新后」的购物车内的商品 获取价格明细
[Node] [Inject Soa Mock]: 12345/getShoppingSvc  开始: 330ms  耗时: 110ms  结束: 440ms
```

同时，在 GQL 代码里也应按照前端查询的操作符来决定是否执行“事务性”操作。

```
async function recommendExtraResource(ctx){
  //查询门票
  const extraResource = await getTicketSvc.fetch()
    const { operation } = ctx.info.operation;
    if (operation === 'mutation'){
        //添加到购物车内
        await updateShoppingSvc.fetch(extraResource)
    }
    ctx.result = extraResource
}

ExtraResource.resolve('Query', { recommendExtraResource });
ExtraResource.resolve('Mutation', { recommendExtraResource });
```

这样的设计使查询就变得非常灵活。如前端仅需要查询可用门票和价格明细并不需要默认添加到购物车内，仅需要将 mutation 换成 query 即可，服务端无需为此做任何调整。而且因为没有执行更新，且操作符变成了 query，两个获取数据的接口调用又会变成并行，提高了响应速度。

```
//查询门票
[Node] [Inject Soa Mock]: 12345/getTicketSvc  开始: 16ms  耗时: 111ms  结束: 127ms
//根据「当时」的购物车内的商品 获取价格明细
[Node] [Inject Soa Mock]: 12345/getShoppingSvc  开始: 18ms  耗时: 104ms  结束: 112ms
```

5.3 父子查询中的重复请求

我们经常会碰到一个接口的入参，依赖另外一个接口的 response。这种将串行调用从客户端移到服务端的做法可以有效的降低端到端的次数，是 BFF 层常见的优化手段。但是如果我们有多个节点一起查询时，可能会出现同一个接口被调用多次的问题。对应这种情况，我们可以使用 GQL 的 data-loader。

```
ProductInfo.resolve('Query', {
        productInfo: async (ctx) => {
            let productLoader = new DataLoader(async RequestType => {
                // RequestType 为数组，通过子节点的 load 方法，去重后得到。
                let response = await productSvc.fetch({ RequestType })
                return Array(RequestType.length).fill(response)
            })
            ctx.result = { productLoader }
    }
})

ExtendInfo.resolve('Product',{
    extendInfo: async (ctx) => {
            const BasicInfo = await ctx.parent.productLoader.load("BasicInfo")
            ctx.result = await extendSvc.fetch(BasicInfo)
    }
})
```

如上，在父节点的 resolve 里构造 loader，通过 ctx.result 传递给子节点。子节点调用 load(arg) 方法将参数添加到 loader 里，父节点的 loader 根据“积累”的参数，发起真正的请求，并将结果分别下发对应地子节点。在这个过程中可以实现相同的请求合并只发一次。

六、工程化实践

6.1 异常处理

在 GQL 关联查询中父节点失败导致子节点异常的情况很常见。而这个父子关系是由前端 query 报文决定的，因此需要我们在服务端处理异常的时候，清晰地通过日志等方式准确描述原因，上图可以看出 imEnterInfo 节点异常是由于依赖的 BasicInfo 节点为空，而根因是依赖的 API 返回错误。这样的异常处理设计对排查 GQL 的问题非常有帮助。

6.2 虚拟路径

由于 GQL 唯一入口的特性，服务捕获到的访问路径都是 /basename/graphql，导致定位错

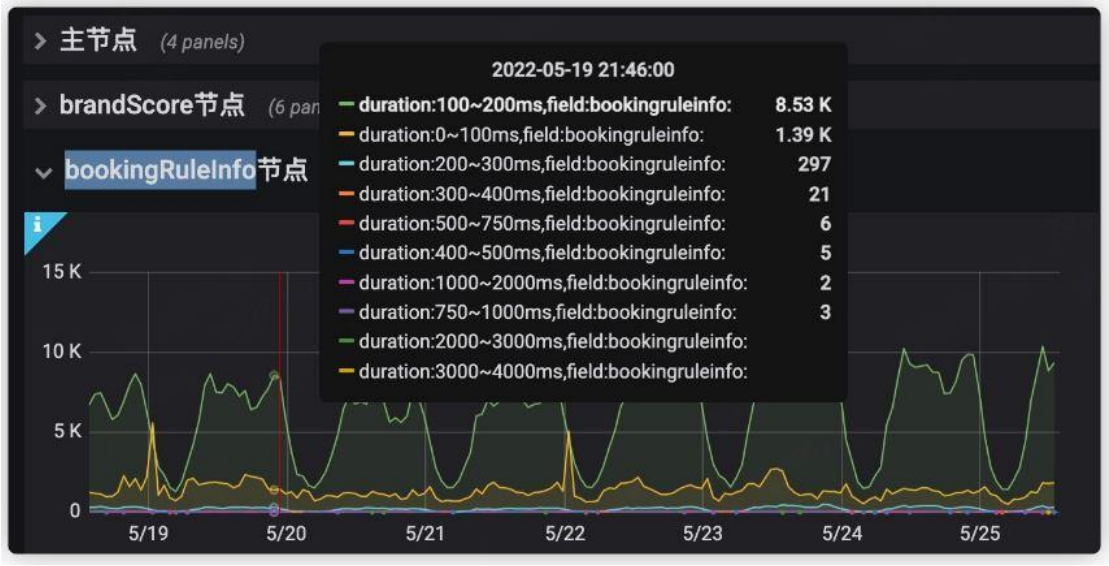
误很困难。因此我们扩展了虚拟路径，前端查询的时候使用类似「/basename/graphql/productInfo」。这样无论是日志、还是 metric 等平台等都可以区分于其他查询。

并且这个虚拟路径对 GQL 自身不会造成影响，前端甚至可以利用这个虚拟路径来测试 query 的节点和 BFF 响应时长的关系。如：H5 平台修改了首屏 query 的内容之后将请求路径改成 “/basename/graphql/productInfo_h5”，这样就可以通过性能监控 95 线等方式，对比看出这个“h5”版本对比其他版本性能是否有所下降。

在很多优化首屏的实践中，利用 GQL 动态查询，灵活剪切契约等是非常有效的手段。并且在过程中，服务端并不需要跟随前端调整代码。降低工作量的同时，也保证了其他平台的稳定性。

6.3 监控运维

GQL 的特性也确实造成了现有的运维工具很难分析出哪个节点可以安全废弃（删除代码）。因此需要我们在 resolve 里面对节点进行了埋点。



6.4 单元测试

我们利用 jest 搭建了一个测试框架来对 GQL BFF 进行单元测试。与一般单测不同的是，我们选择在当前运行环境内单独起一个服务进程，并且引入“@apollo/client”来模拟客户端对服务进行查询，并校验结果。

```
import { gql} from '@apollo/client';
import {GraphqlQuery} from "../../_util/graphqlServer"
describe('[Product Service Unit Test]', () => {
    let datas:any = null
    beforeAll(async () => {
        const GET_LAUNCH = gql`
            query getProductInfo($params: ProductArgs!) {
                productInfo(params: $params) {
                    BasicInfo{ json }
                    .....
                }
            }
        `
        const variables = {…
        }
        datas= await GraphqlQuery({variables:variables,query:GET_LAUNCH})
    })
    describe('[校验 ProductDetail]', () => {
        test('ProductDetail 节点完整',() => {
            let {BasicInfo
            } = datas
            expect(BasicInfo != null).toEqual(true)
        })
    }
```

其他诸如 CI/CD、接口数据 mock、甚至服务的心跳检测等更多的属于 node.js 的解决方案，就不在这里赘述了。

七、总结

鉴于篇幅原因，只能分享部分我们应用 GraphQL 开发 BFF 服务的思考与实践。由前端团队开发维护一套完整的服务层，在设计和运维方面还是有不小的挑战，但是能赋予前端团队更大的灵活自主性，对于研发迭代效率的提升也是显著的。

希望对大家有所帮助，欢迎更多关于 GraphQL 的实践和交流。

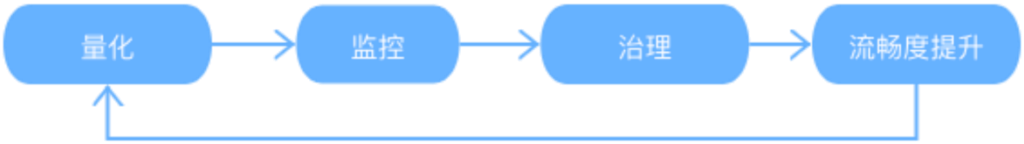
从 47%到 80%，携程酒店 APP 流畅度提升实践

【作者简介】Jin，携程高级研发经理，专注移动技术开发；Dan，携程测试开发经理，关注数据挖掘以及数据在系统质量提升中的应用；Lanbo，携程软件技术专家，专注移动技术开发。

一、背景

APP 性能提升一直是研发团队永恒的主题。在进行 APP 性能优化实践中，除了性能技术方案本身外，还会面临两方面问题：第一，APP 的性能优化，不具有持续性，往往经过一段时间优化实践，效果明显，但是随着后续需求迭代和代码变更，APP 性能很难维持在一个较好的水平上；第二，APP 性能改善提升，缺乏一套科学量化手段进行衡量。

引用管理学大师彼得•德鲁克的一句话：If you can't measure it, you can't improve it，如果你无法度量它，你就无法改进它。基于此，携程酒店前端 APP 团队进行了深入思考和探索，希望通过量化，治理，监控三方面手段，持续改善 APP 性能和用户体验。



二、流畅度指标定义

流畅度，简单说就是度量用户使用 APP 体验的一部分，它是用户快速、无阻碍使用 APP 的一项体验指标。主要包括三方面内容：稳、快、质。稳的含义是用户在打开具体一个页面时，没有出现白屏、崩溃、闪动等。快的含义是页面打开很快，用户在页面进行交互时，操作流畅自然。质的含义，是在浏览页面时，没有无故的弹窗拦截，打断用户的操作。如下图所示：

基于以上理论基础，APP 中白屏，崩溃闪退，加载慢，卡顿，闪动，报错，都是用户在感知层面形成不流畅的因素。于是我们提出了流畅率量化指标，把用户页面 PV 以及用户在页面触发的二次加载次数之和，定义为流畅率的分母，也就是样本总量，如下公式：

样本量 = 页面 pv+二次加载数

把页面慢加载/页面卡顿/图片/视频慢加载PV去重后数量，加上页面出现的崩溃，滑动卡顿，图片/视频加载失败，全局弹窗报错，输入失焦，按钮点击无效，二次加载失败，二次加载慢等异常情况之和定义为不流畅因子数。那么流畅率的公式定义为：

流畅率 = （样本量-不流畅因子数）/ 样本量

2.1 页面可交互加载时长

页面可交互加载时长，是页面渲染绘制时间叠加网络服务的请求响应时间，可以简单用下面公式表示：

页面可交互加载时长（TTI）= 页面本地渲染时长+服务网络加载时长

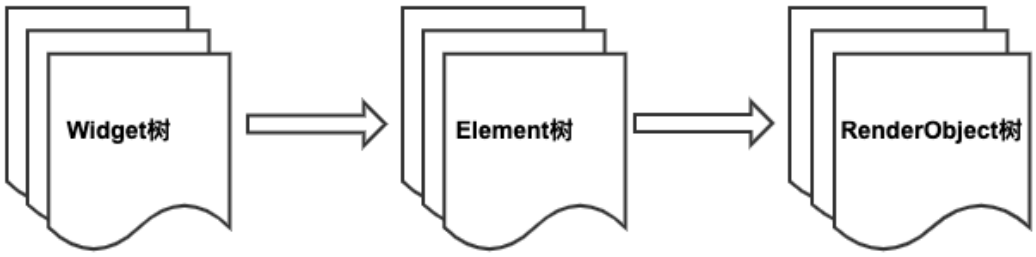
2.2 页面可交互加载时长采集原理

在我们的核心页面中，都包含了 Text 控件，可以通过扫描页面中特定区域内的文本来确定用户可交互时间。我们的技术栈大体上分为 Flutter 和 Ctrip React Native，以下分别介绍加载时长采集原理。

2.2.1 Flutter 页面可交互加载时长采集原理

在 Flutter 中，最终的 UI 树其实是由一个个独立的 Element 节点构成。UI 从创建到渲染的大体流程如下：

根据 Widget 生成 Element，然后创建相应的 RenderObject 并关联到 Element.renderObject 属性上，最后再通过 RenderObject 来完成布局排列和绘制。如下图如所示：



所以可以从根节点开始遍历 Element，直到找到扫描窗口内的 Text 组件且组件的内容不为空，即可判定页面 TTI 检测成功，Flutter 提供如下接口支持 Element 遍历：

voidvisitChildElements(ElementVisitor visitor)

2.2.2 Ctrip React Native 页面可交互加载时长采集原理

我们知道，ReactNative 最终是由 Native 组件来渲染的，在 iOS/Android 中可通过从根 View 从 View 树中递归查找 Text 文本控件，来获取页面内文内的内容，去掉页面顶部固定静态展示和底部静态展示区域之外，扫描到的文本数量大于 1 个，我们就认为页面 TTI 检测成功了。

2.3 渲染卡顿和帧率

Google 对卡顿定义：界面呈现是指从应用生成帧并将其显示在屏幕上的动作。要确保用户能够流畅地与应用互动，应用呈现每帧的时间不应超过 16ms，以达到每秒 60 帧的呈现速度。如果应用存在界面呈现缓慢的问题，系统会不得不跳过一些帧，这会导致用户感觉应用不流畅，我们将这种情况称为卡顿。

2.3.1 卡顿标准

判断 APP 是否出现卡顿，应该从 APP 类型是普通应用还是游戏应用出发，不同类型 APP，对应不同的卡顿标准。针对普通类型应用，可以参考借鉴 Google 的 Android Vitals 性能指标，针对游戏，可以参考借鉴腾讯的 PrefDog 性能指标。因为我们 APP 是普通应用，简单的介绍下 Google Vitals 的卡顿定义。

GoogleVitals 把卡顿分为了两类：

第一类是呈现速度缓慢：在呈现速度缓慢的帧数较多的页面，当超过 50% 的帧呈现时间超过 16ms 毫秒时，用户感官明显卡顿。

第二类是帧冻结：帧冻结的绘制耗时超过 700ms，为严重卡顿问题。

另外，要注意的是，FPS 的高低和卡顿没有必然关系，帧率 FPS 高并不能反映流畅或不卡顿。比如：FPS 为 50 帧，前 200ms 渲染一帧，后 800ms 渲染 49 帧，虽然帧率 50，但依然觉得非常卡顿。同时帧率 FPS 低，并不代表卡顿，比如无卡顿时均匀 FPS 为 15 帧。

2.3.2 卡顿量化

当了解卡顿的标准以及原理之后，可以得出结论，只有丢帧情况才能准确判断是否卡顿。

Flutter 官方提供一套基于 SchedulerBinding.addTimingsCallback 回调实现的实时帧数据的监控。当 flutter 页面有视图绘制刷新时，系统吐出一串 FrameTiming 数据 ，FrameTiming 的数据结构如下：

vsyncStart,
buildStart,

buildFinish,
rasterStart,
rasterFinish

vsyncStart 变量表示当前帧绘制的起始时间，buildStart/buildFinish 表示 WidgetTree 的 build 时间，rasterStart/rasterFinsih 表示上屏的光栅化时间，那么一帧的总渲染时间，可以利用下面公式得到：

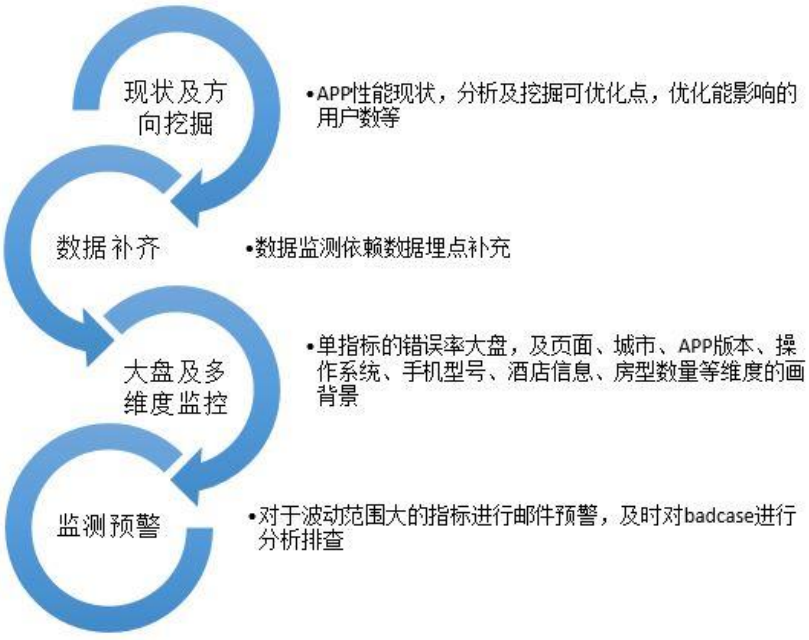
totalSpan=>rasterFinish - syncStart

对应 Google Android Vitals 卡顿的标准：如果一帧 totalSpan > 700ms，认为发生了帧冻结，产生了比较严重的卡顿；如果 1s 内，有超过 30 次的帧的绘制时间 totalSpan> 16ms，产生了呈现速度缓慢。

三、流畅度监控方案

在流畅度监控体系中，对于不流畅感知因子，进行单项分析及挖掘，旨在在迭代优化的同时，维持或提升已有的用户体验。

监控体系的搭建，分为现状及优化方向挖掘、监控指标依赖数据补齐、多维度的数据监控、指标监测预警。



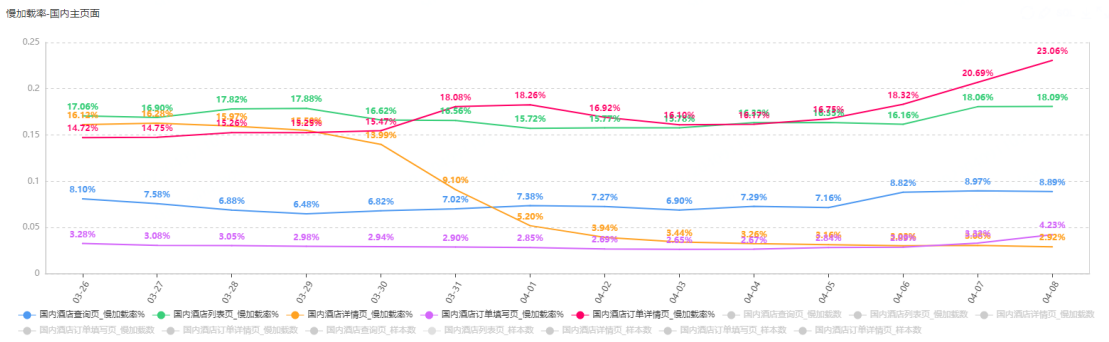
监控搭建前期会对于 APP 现有的性能现状进行分析，挖掘可优化的方向，初步获得优化所带来的预计收益、影响的用户数等信息。如：预计使用预加载的方式，来降低用户的慢加载率，通过各场景的不同用户操作分析，以及目前客户端及服务端技术实现的现状（酒店主服务返回报文大小统计、酒店详情纯前端渲染时间等），来确定慢加载的覆盖面、触发时机，以达到更优的效果。

接下来，针对流畅度优化的业务、技改上线的同时，补充对应的监测场景埋点，以支撑流畅度量化衡量的数据，为后续的监控及预警，奠定坚实的基石。

监控体系的核心是大盘及多维度监控的铺开，大盘数据（如下图所示），能快速宏观的了解用户预订体验，明确流畅度提升的进度；而通过各种维度的数据表，即可以找到提升目标，也能监控优化效果。

在实际监控中，会针对不同的指标，设计不同的监控标准，如：慢加载、白屏、奔溃、卡顿等系统因素，除了大盘指标外，还增加了各指标影响占比、酒店主页面的报错率趋势、版本对比趋势、报错机型 top 分布等。

对于业务场景比较重的因素，结合业务数据进行分桶等方式的监控，如：详情页房型数量关联 TTI 耗时分布、单酒店 crash 数据等。并与 AB 实验系统打通，业务、技改类需求都可以在 AB 系统中配置流畅度观测指标，比对业务或技改需求对流畅度的指标影响，作为实验是否通过的考量指标。



对于各项指标进行单项波动预警，做到有上升就预警、有新增就预警，做到不放过、不遗漏。如：填写页业务报错量（可订服务、提交订单、失焦错误数），除了对各类报错率趋势进行监控外，还会综合实际用户流量，区分单项业务报错的流量大小进行预警，且对拆分多维度（单用户、单房型等）触发次数，便于寻找到有特性的 badcase，快速定位用户遇到的问题，挖掘更多的业务优化点。

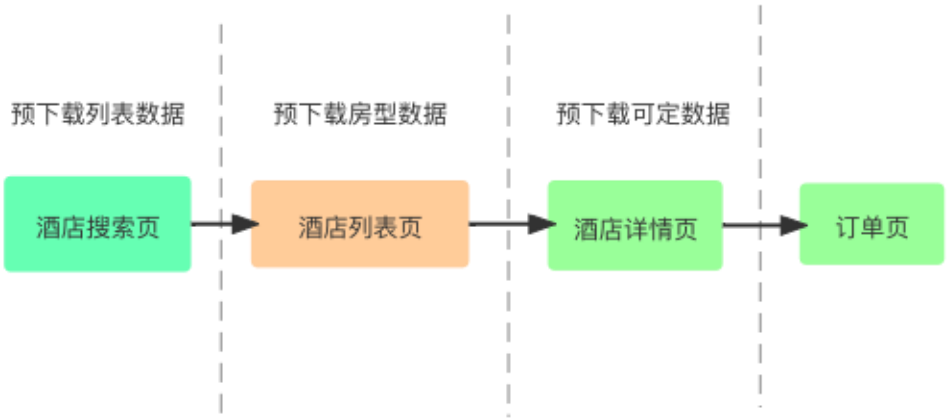
四、流畅度治理实践

在 APP 流畅度治理上，主要从页面启动加载速度，长列表卡顿治理，页面加载闪动三个方面进行了诸多优化实践，这些优化并没有涉及高大上的底层引擎优化技术，也没有复杂的数学理论基础，更没有重复造轮子。我们坚持以数据为导向，用数据驱动方案，用数据验证方案，发现问题，提出解决方案，解决问题。

4.1 页面加载速度优化

在页面加载速度优化上，我们从 2021 年 8 月份开始进行迭代优化至今，酒店预订流程页面的慢加载率从初始值的 42.90%降低至现阶段的 8.05%。

在页面启动加载速度优化上，一般都会采用数据预获取方案，原理是在上一个页面提前获取服务数据，在用户跳转到当前页面时，直接从缓存获取，节省了数据的网络传输时间，达到快速展示当前页面内容的效果。目前在酒店核心预订流程，都运用了数据预加载技术，如下图所示：



结合酒店业务特点，数据预加载需要考虑几个方面问题：第一，酒店预订流程页面 PV 量较高，酒店列表和详情页 PV 在千万级别。需要考虑数据预加载的时机，避免服务的资源浪费；第二，酒店列表、详情、订单填写页都有价格信息，价格信息对用户来说是动态信息，实时都有变价可能，所以需要考虑数据预加载的缓存策略，避免因为价格的前后不一致造成用户误解。

4.2 Flutter 服务通道优化

携程 APP 采用的私有服务协议，目前发服务的动作还是在 Native 代码上，而酒店的核心页面已经转到了 Flutter 上。通过 Flutter 框架提供的通道技术，Native 到 Flutter 的数据传输通道需要对数据做一次额外的序列化及反序列化的传输，同时传输的过程比较耗时，会阻塞 UI 的渲染主线程，对页面的加载会造成明显的影响。我们检测到这个环节之后，和公司的框架团队一起对 Flutter 的底层框架进行了改造，可以实现数据流直接的透传，同时不阻塞 UI 主线程，性能得到了极大的提升。

优化前，通过服务返回的数据流传递到 Flutter 使用，整个过程要经历以下 4 步：

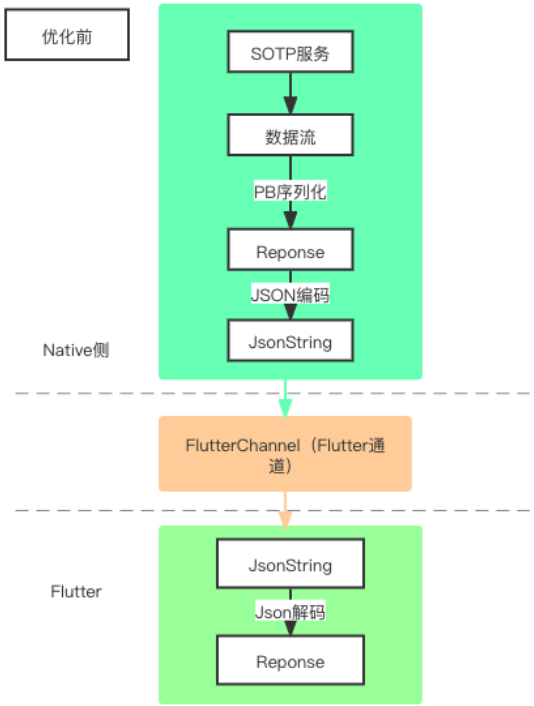
（1）PB 反序列化；

（2）Reponse 到 JsonString 的编码；

（3）JsonString 到 Flutter 通道传输；

（4）JsonString 到 Reponse 的解码；

整个过程链路长，数据传输量大，效率低，影响到页面加载性能，如下图所示：

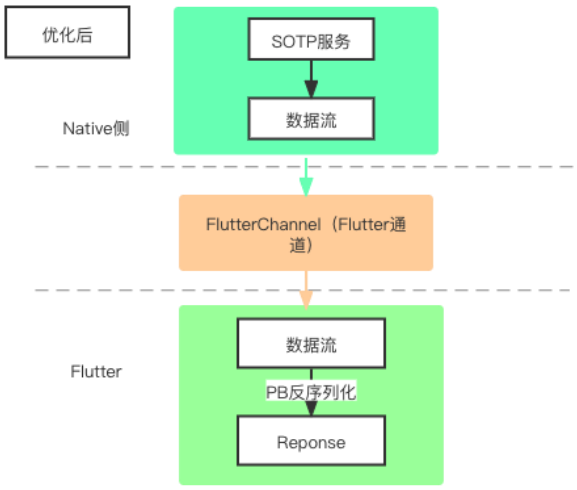


改造后，通过服务返回的数据流，直接传输到 Flutter 侧，在 Flutter 直接进行 PB 的反序列化，传输性能得到极大提升。

（1）Flutter 通道传输

（2）序列化到 Reponse

整个过程链路短，数据传输量小，效率高，如下图所示：



4.3 卡顿问题分析和定位

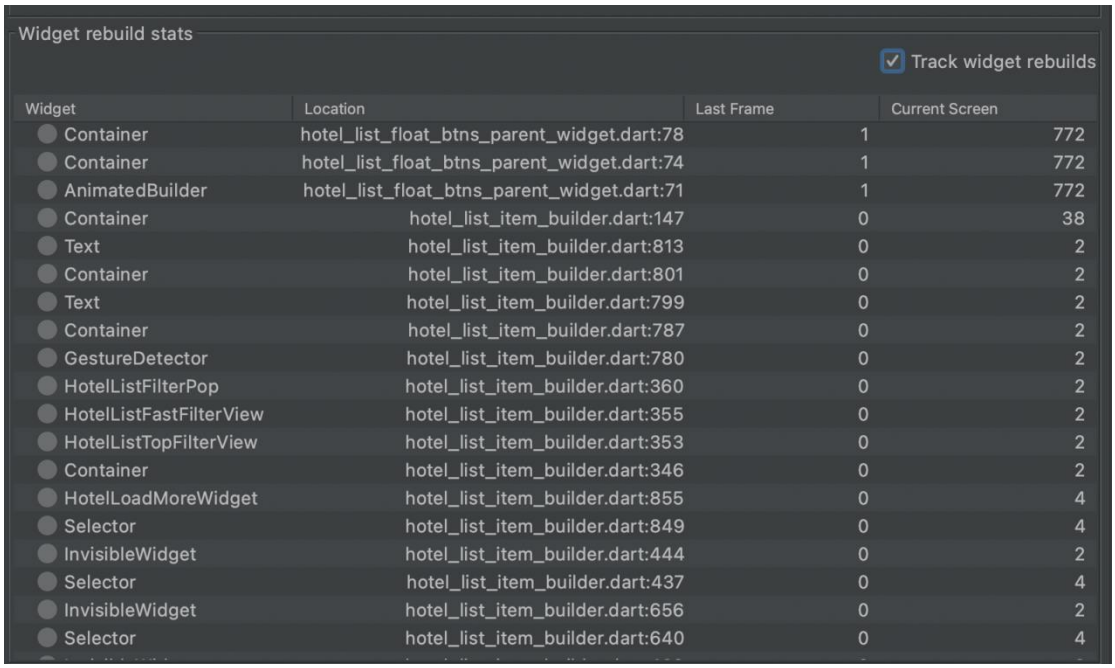
在 Flutter 中，可以利用性能图层（Performance Overlay），来分析渲染卡顿问题。如果 UI 产生了卡顿，它可以辅助我们分析并找到原因。如下图所示：

GPU 线程的绘制性能情况在图表的上方，CPU UI 线程的绘制情况显示在图表下方，蓝色垂线表示已渲染的帧，绿色色垂线代表的是当前帧。

为了保持 60Hz 刷新频率，每一帧耗时都应该小于 16ms（1/60 秒）。如果其中有一帧处理时间过长，就会导致界面卡顿，图表中就会展示出一个红色竖条。下图演示了应用出现渲染和绘制耗时的情况下，性能图层的展示样式：

如果红色竖条出现在 GPU 线程图表，意味着渲染的图形太复杂，导致无法快速渲染；而如果是出现在了 UI 线程图表，则表示 Dart 代码消耗了大量资源，需要优化代码执行时间。

另外我们可以借助于 AS 里面的 Flutter Performance 工具查看 Flutter 页面的 rendering 性能问题，里面有个很有用的功能 Widget rebuild stats，它统计在渲染 UI 的时候，各个 widget rebuild 数量情况，可以辅助我们很快的定位存在问题的 widget，如下图：

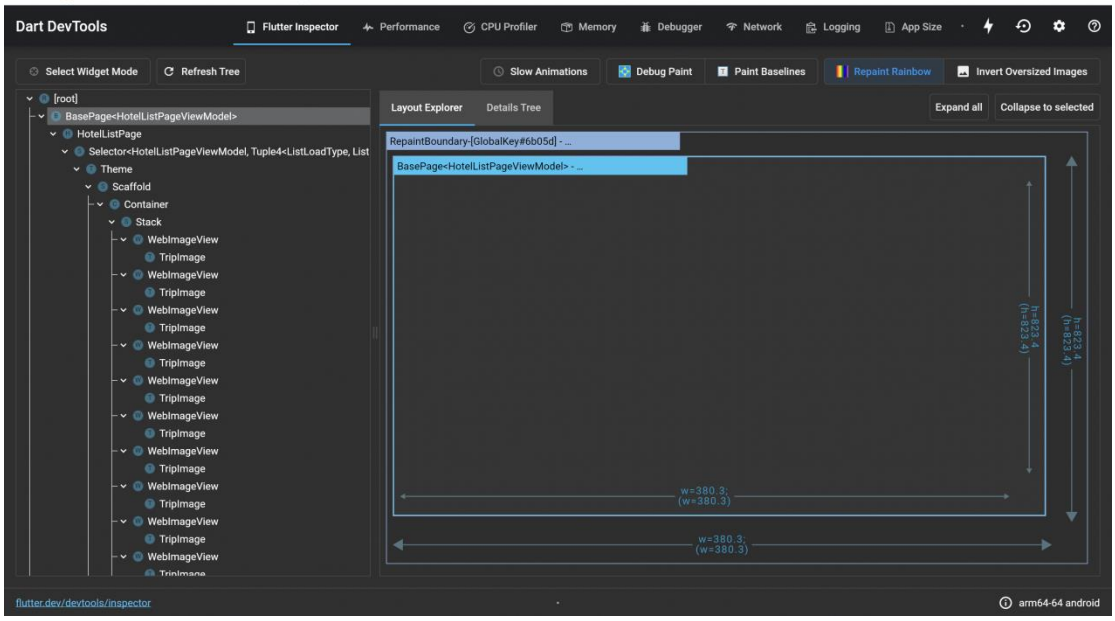


Widget	Location	Last Frame	Current Screen
Container	hotel_list_float_btns_parent_widget.dart:78	1	772
Container	hotel_list_float_btns_parent_widget.dart:74	1	772
AnimatedBuilder	hotel_list_float_btns_parent_widget.dart:71	1	772
Container	hotel_list_item_builder.dart:147	0	38
Text	hotel_list_item_builder.dart:813	0	2
Container	hotel_list_item_builder.dart:801	0	2
Text	hotel_list_item_builder.dart:799	0	2
Container	hotel_list_item_builder.dart:787	0	2
GestureDetector	hotel_list_item_builder.dart:780	0	2
HotelListFilterPop	hotel_list_item_builder.dart:360	0	2
HotelListFastFilterView	hotel_list_item_builder.dart:355	0	2
HotelListTopFilterView	hotel_list_item_builder.dart:353	0	2
Container	hotel_list_item_builder.dart:346	0	2
HotelLoadMoreWidget	hotel_list_item_builder.dart:855	0	4
Selector	hotel_list_item_builder.dart:849	0	4
InvisibleWidget	hotel_list_item_builder.dart:444	0	2
Selector	hotel_list_item_builder.dart:437	0	4
InvisibleWidget	hotel_list_item_builder.dart:656	0	2
Selector	hotel_list_item_builder.dart:640	0	4

（1）UI 线程问题定位

UI 线程问题实际就是应用性能瓶颈。比如在 Widget 构建时，在 build 方法中使用了一些复杂运算，或是在 Root Isolate 中进行了耗时的同步操作（比如 IO）。这些都会明显增加 CPU 处理时间，造成卡顿。

我们可以使用 Flutter 提供的 Performance 工具，来记录应用的执行轨迹。Performance 是一个强大的性能分析工具，能够以时间轴的方式展示 CPU 的调用栈和执行时间，去检查代码中可疑的方法调用。在点击了 Flutter Performance 工具栏中的“Open DevTools”按钮之后，系统会自动打开 Dart DevTools 的网页，我们就可以开始分析代码中的性能问题了。



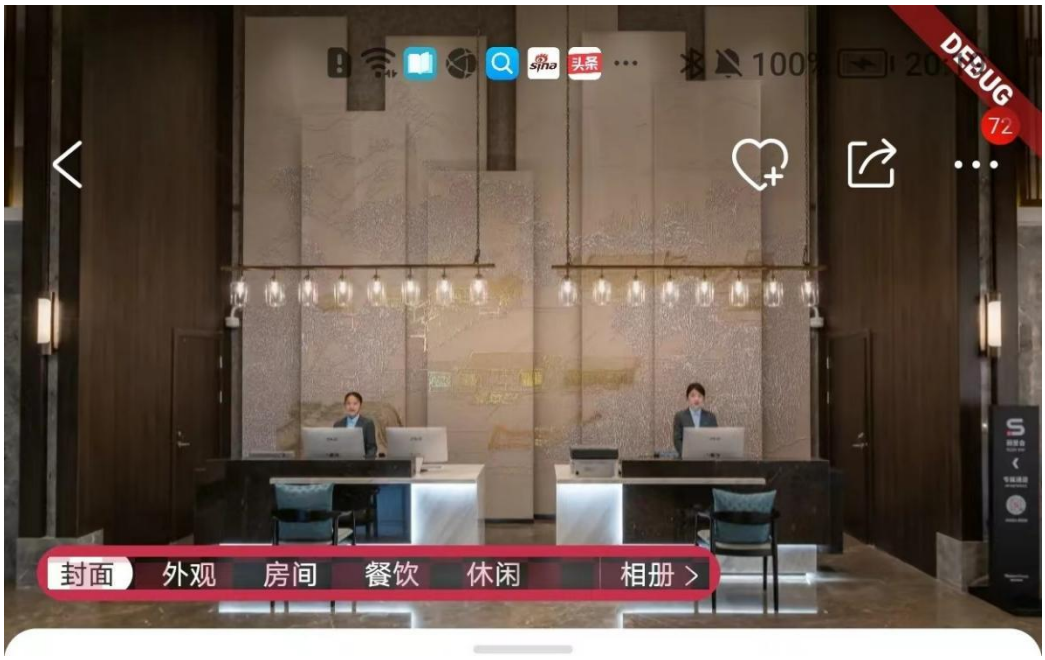
（2）GPU 问题定位

GPU 问题主要集中在底层渲染耗时上。有时候 Widget 树虽然构造起来容易，但在 GPU 线程下的渲染却很耗时。涉及 Widget 裁剪、蒙层这类多视图叠加渲染，或是由于缺少缓存导致静态图像的反复绘制，都会明显拖慢 GPU 的渲染速度可以使用性能图层提供的两项参数，负责检查多视图叠加的视图渲染开关 checkerboardOffscreenLayers 和负责检查缓存的图像开关 checkerboardRasterCacheImages。

（3）heckerboardOffscreenLayers

多视图叠加通常会用到 Canvas 里的 savaLayer 方法，这个方法在实现一些特定的效果（比如半透明）时非常有用，但由于其底层实现会在 GPU 渲染上涉及多图层的反复绘制，因此会带来较大的性能问题。对于 saveLayer 方法使用情况的检查，我们只要在 MaterialApp 的初始化方法中，将 checkerboardOffscreenLayers 开关设置为 true，分析工具就会自动帮我们检测多视图叠加的情况了，使用了 saveLayer 的 Widget 会自动显示为棋盘格式，并随着页面刷新而闪烁。

不过，saveLayer 是一个较为底层的绘制方法，因此我们一般不会直接使用它，而是会通过一些功能性 Widget，在涉及需要剪切或半透明蒙层的场景中间接地使用。所以一旦遇到这种情况，我们需要思考一下是否一定要这么做，能不能通过其他方式来实现。如下图所示，因为详情头部 bar 用到高斯模糊，同时使用 ClipRRect 裁切圆角，ClipRRect 会调到 savelayer 接口，所以该部分产生闪烁。



（4）checkerboardRasterCacheImages

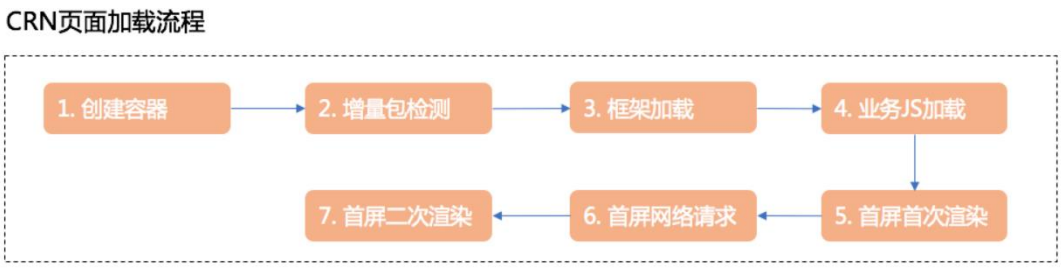
从资源的角度看，另一类非常消耗性能的操作是，渲染图像。这是因为图像的渲染涉及 I/O、GPU 存储，以及不同通道的数据格式转换，因此渲染过程的构建需要消耗大量资源。

为了缓解 GPU 的压力，Flutter 提供了多层次的缓存快照，这样 Widget 重建时就无需重新绘制静态图像了。与检查多视图叠加渲染的 checkerboardOffscreenLayers 参数类似，Flutter 也提供了检查缓存图像的开关 checkerboardRasterCacheImages，来检测在界面重绘时频繁闪烁的图像（即没有静态缓存）。

我们可以把需要静态缓存的图像加到 RepaintBoundary 中，RepaintBoundary 可以确定 Widget 树的重绘边界，如果图像足够复杂，Flutter 引擎会自动将其缓存，避免重复刷新。当然，因为缓存资源有限，如果引擎认为图像不够复杂，也可能会忽 RepaintBoundary。

4.4 Ctrip React Native(简称 CRN)页面的优化

下图是基本的 CRN 页面的加载流程，各个阶段的优化之前已有文章进行过描述，如容器预加载，Bundle 拆分，容器复用，框架预加载等等在容器层面做了优化。



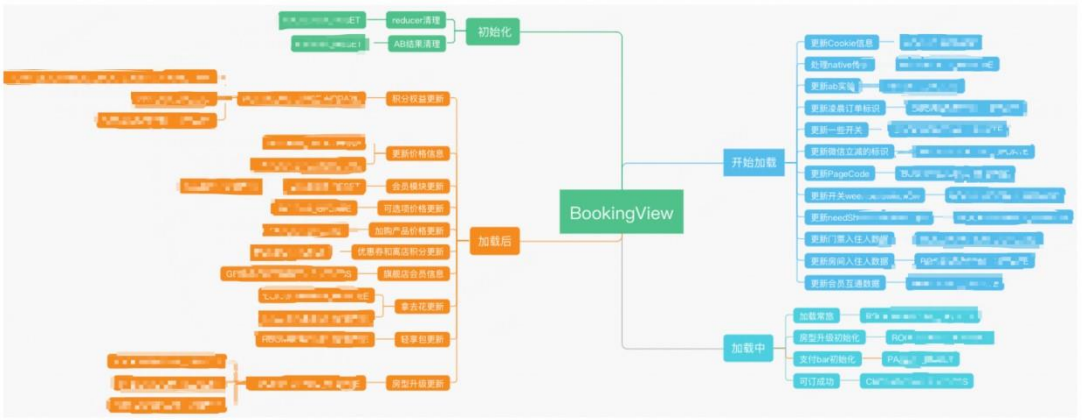
以酒店订单填写页为例，此页面采用了 CRN 的架构，在已有各类容器层面和框架层面的优化之后，我们重点对页面内重绘做了治理，并将重绘治理做到了极致，主要涉及到上图中的“5. 首屏首次渲染”和“7. 首屏二次渲染”。

4.4.1 页面内 Action 整合

此页面采用 Redux 架构，前期经历了几年的粗放式开发之后，页面内的 action 众多（Action 通过异步事件的方式触发状态管理的改变，从而达到页面重绘的目的，可以参考 Redux 的 Action-Reducer-Store 模式）。

优化前，如下图，页面初始化/开始加载/加载中/加载完成，均触发多个 action，由于 action 是异步的，每个数据处理模块都有一些耗时和异步，加载完成后页面可能已经刷新，此处有可能展示了未处理完成的数据，等后续 action 执行完成后，页面会再次刷新。

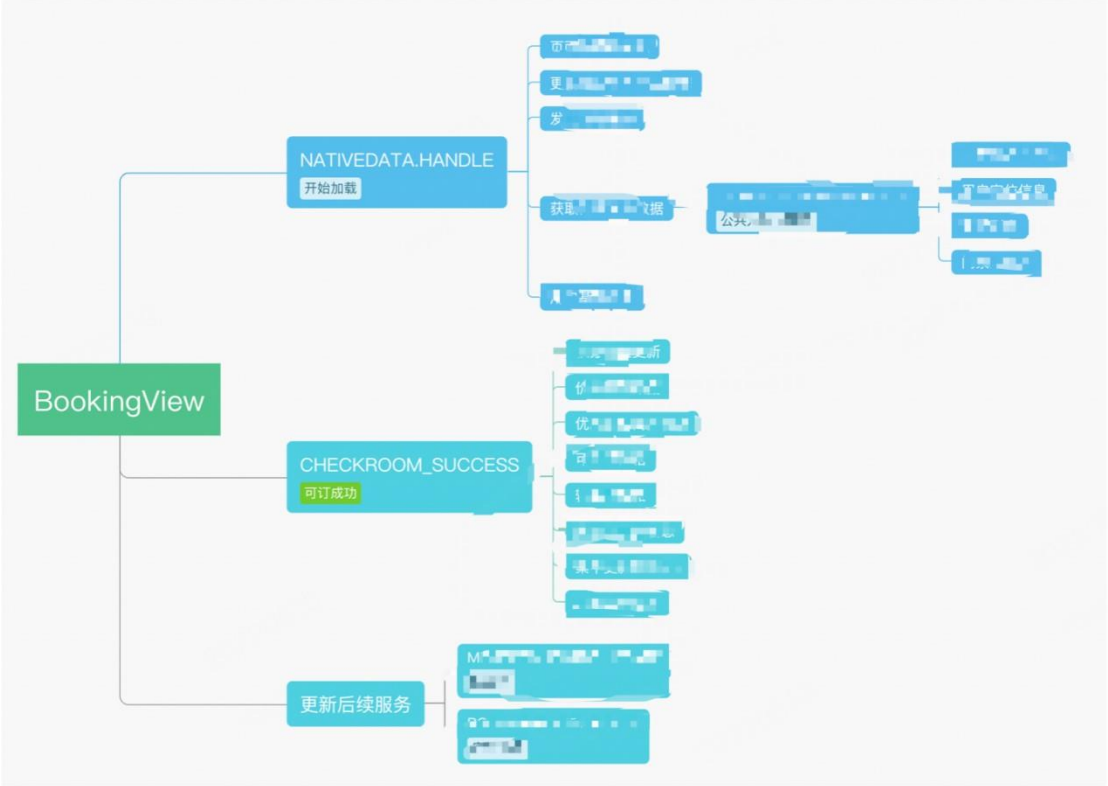
由于有数据变化，页面内元素可能会有变化，从而对用户而言，页面产生了抖动，同时也会加大 JS<=>Native 的通信量，页面内元素的不断变化，也会不断刷新 native 中的渲染树，消耗大量 CPU 时间，进而导致页面不流畅，耗时较长。



针对上述情况，我们对页面内的 Action 做了整合：

- 静态数据避免使用 action
- 触发时机相同的 action 尽量合并
- 非必要数据延迟加载
- 多层 action 的更新进行整合

整合后，页面内的 action 大致如下：基本只有页面初始化，主服务返回，以及后续子服务的 action 了。



在此过程中我们采用了 redux-logger 的方式来监控 action，同时采用 MessageQueue 的方式来监控 action 变化触发刷新的情况，如下图：

```
const MessageQueue = require('react-native/Libraries/BatchedBridge/
MessageQueue');
MessageQueue.spy(true);
```

4.4.2 控件重绘治理

为了更好的控制控件重绘的频率，我们对控件做了以下拆分：

- 尽量的拆细组件
- 降低单文件的复杂度
- 组件复用更加方便
- 依赖数据变少，状态更好管理
- 局部更新数据不影响其他组件
- 使用 Fragments 避免多层嵌套

拆分之后组件颗粒度更小，弱业务相关的采用了 PureComponent，强业务组件采用

Component+shouldComponentUpdate+自行比较属性是否变化来避免组件的重绘。

通过上述方式的治理，进入填写页内已明显感觉页面比较轻，主服务返回后页面立等可刷新，页面的渲染速度大幅提升。

重绘治理我们采用了 https://github.com/welldone-software/why-did-you-render 的方案来检测组件由于什么原因重绘，如下图：

```
const whyDidYouRender = require('@welldone-software/why-did-you-render');
whyDidYouRender(React, {
  trackAllPureComponents: true,
    groupByComponent: true, collapseComponentGroups: true,
});
```

五、规划和总结

整个 APP 流畅度治理中，从流畅率从初始 47%提升到目前 80%，页面慢加载率从原来的 45%降低到现在的 8%，白屏率从 1.9%降至现在的 0.3%，主流程页面控件闪动基本消除，APP 性能及用户体验有了较明显的提升。

回顾近半年中文酒店 APP 流畅度实践，整个过程艰辛，也时刻伴随着焦虑。流畅度每一点的进步都不是一蹴而就，轻易达成的。但对整个团队，收获满满，整个实践过程中，我们对 flutter 工程架构做了整体升级，尤其是数据传输层改造，业务层逻辑收口等；数据的预加载方案，也从 1.0 版本升级到 2.0 版本。最重要的是，整个团队形成了数据量化的思想意识和用户视角出发去优化和解决问题。

目前流畅度 2.0 的版本也已经落地实践，2.0 将更多的不流畅感知因子加入流畅度统计，如主服务的二次加载，地图慢加载、图片及视频慢加载、图片及视频加载失败、弹窗及提示信息等，从更多系统及业务层面来提升用户的预订体验。

Taro 性能优化之复杂列表篇

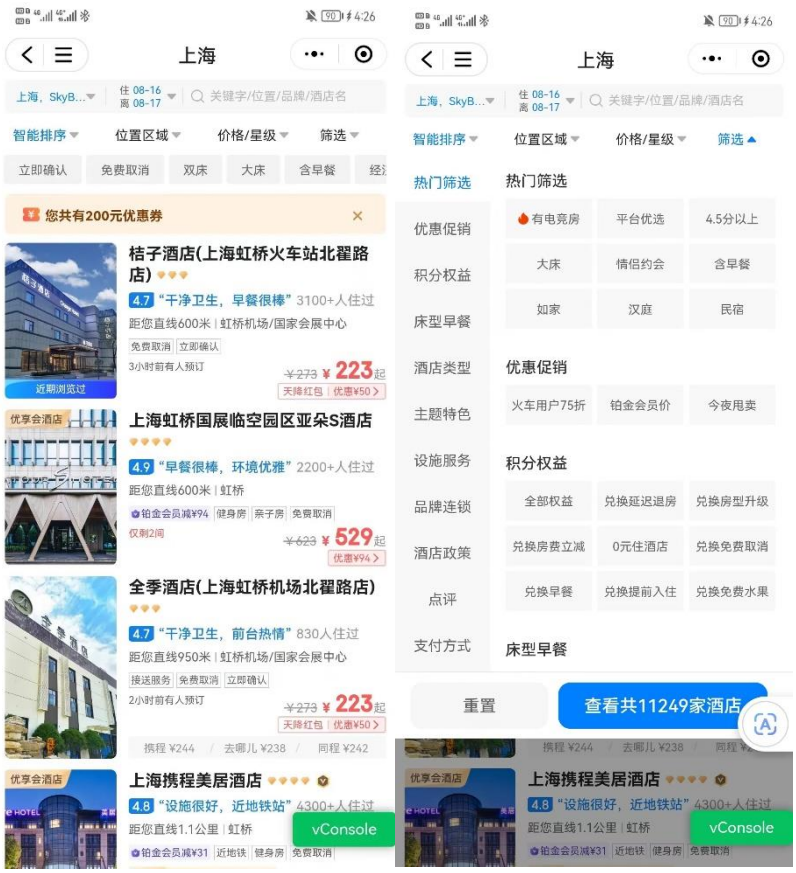
【作者简介】Kenny，携程高级前端开发工程师。2021 年加入携程，从事小程序/H5 相关研发工作。

一、背景

随着项目的不断迭代，规模日益增大，而基于 Taro3 的运行时弊端也日渐凸显，尤其在复杂列表页面上表现欠佳，极度影响用户体验。本文将以复杂列表的性能优化为主旨，尝试建立检测指标，了解性能瓶颈，通过预加载、缓存、优化组件层级、优化数据结构等多种方式，实验后提供一些技术方案的建议，希望可以给大家带来一些思路。

二、问题现状及分析

我们以酒店某一多功能列表为例(下图)，设定检测标准(setData 次数及该 setData 的响应时效作为指标)，检测情况如下:



指标	setData 次数	渲染耗时(ms)
第一次进入列表页	7	2404
下拉长列表更新	3	1903
多屏列表下 筛选项更新	2	1758

多屏列表下 列表项更新	2	748

由于历史原因，该页面的代码，由微信的原生转成的 taro1，后续迭代至 taro3。项目中存在小程序原生写法可能忽略的问题。根据上面多次测出的指标值，以及视觉体验上来看，存在以下问题:

2.1 首次进入列表页的加载时间过长，白屏时间久

- 列表页请求的接口时间过长;
- 初始化列表也是 setData 数据量过大，且次数过多;
- 页面节点数过多，导致渲染耗时较长;

2.2 页面筛选项的更新卡顿，下拉动画卡顿

- 筛选项中节点过多，更新时 setData 数据量大;
- 筛选项的组件更新会导致页面跟着一起更新;

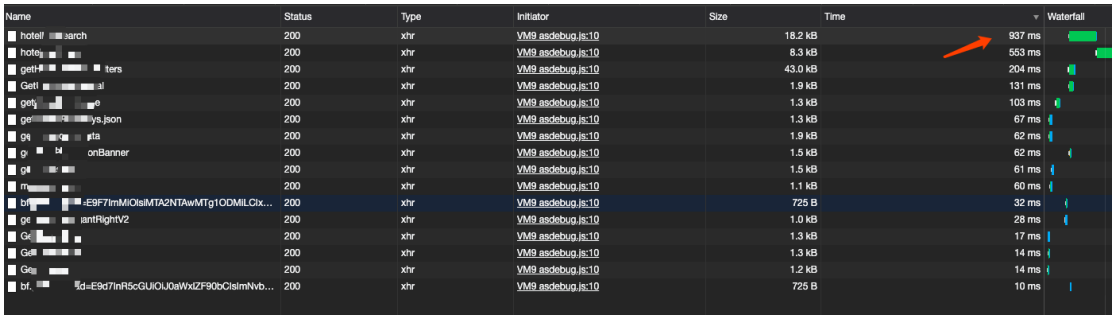
2.3 无限列表的更新卡顿，滑动过快会白屏

- 请求下一页的时机过晚;
- setData 时数据量大，响应慢;
- 滑动过快时，没有从白屏到渲染完成的过渡机制，体验欠佳;

三、尝试优化的方案

3.1 跳转预加载 API：

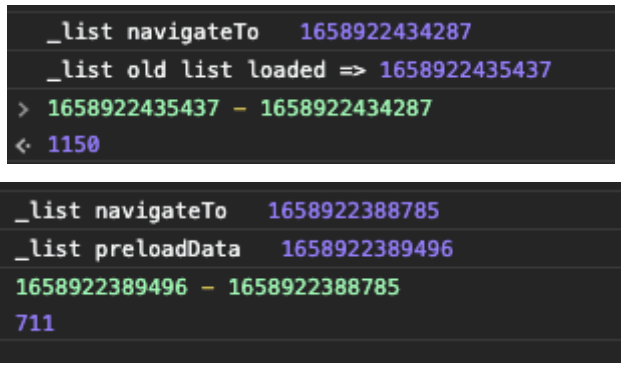
通过观察小程序的请求可以发现，列表页请求中，有两个请求耗时较为长。



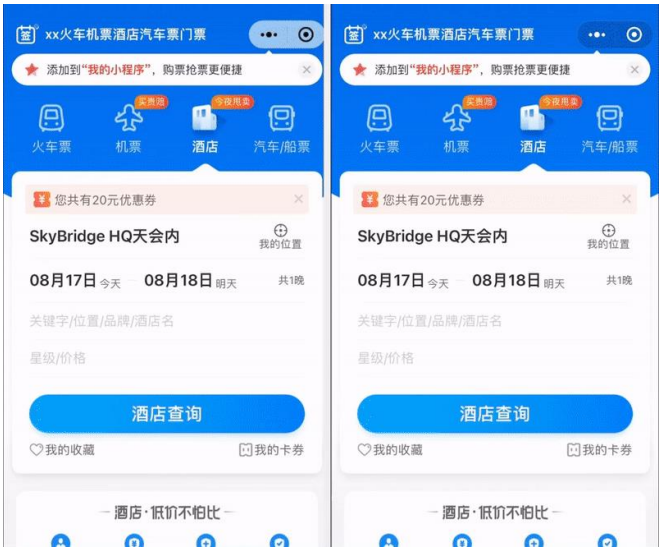
在 Taro3 的升级中，官方有提到预加载 Preload，在小程序中，从调用 Taro.navigateTo 等路由跳转 API 后，到小程序页面触发 onLoad 会有一定延时(约 300ms，如果是分包新下载则跳转时间更长)，因此一些网络请求可以提前到发起跳转时一起去请求。于是我们在在跳转前，使用 Taro.preload 预先加载复杂列表的请求:

```
// Page A
  const query = new Query({
    // ...
  })

  Taro.preload({
    RequestPromise: requestPromiseA({data: query }),
  })

// Page B
  componentDidMount() {
    // 在跳转的过程中，发出请求，因为返回的是一个 promise，所以需要在 B 页面承
接:
    Taro.getCurrentInstance().preloadData?.RequestPromise?.then(res => {
      this.setState(this.processResData(res.data))
    })
  }
```

用同样的检测方式反复测试后，使用 preload 的时，能提前 300~400ms 提前拿到酒店的列表数据。



左边是没使用 preload 的旧列表，右边是预加载的列表，能明显看出预加载后的列表会快一些。



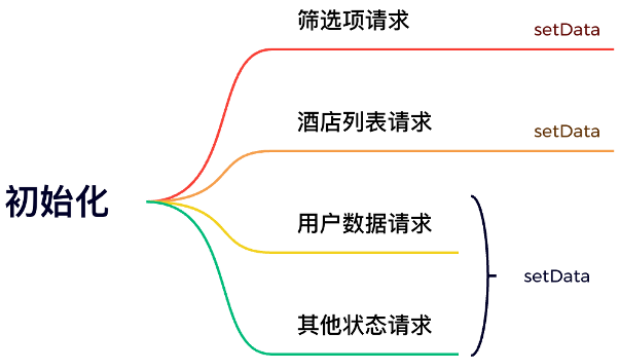
然而在实际的使用中我们发现 preload 存在部分缺陷，对于承接页面，如果接口较为复杂，会对业务流程的代码有一定的入侵。究其本质，是前置了网络请求，所以我们可以对网络请求部分加入缓存策略，即可达到该效果，且接入成本会大大降低。

3.2 合理运用 setData

setData 是小程序开发中使用最频繁、也是最容易引发性能问题的 API。setData 的过程，大致可以分成几个阶段：

- 逻辑层虚拟 DOM 树的遍历和更新，触发组件生命周期和 observer 等；
- 将 data 从逻辑层传输到视图层；
- 视图层虚拟 DOM 树的更新、真实 DOM 元素的更新并触发页面渲染更新。

数据传输的耗时与数据量的大小正相关，旧的列表页第一次加载的时候，一共请求了 4 个接口，setData 短时间里有 6 次，数据量偏大的有两次，我们尝试的优化方式为，将数据量大的两次分开，另外五次发现都是一些零散的状态和数据，可以作为一次。



指标	setData 次数	setData 耗时（ms）	减少耗时百分比
第一次进入列表页	3	2182	9.23%

进行完这一步的操作，平均能减少 200ms 左右，效果较小，因为页面的节点数没变，setData 主要的耗时还分布于渲染时间。

3.3 优化页面的节点数

根据微信官方文档的说明，一个太大的节点树会增加内存使用的同时，样式重排时间上也会更长。建议一个页面节点数量应少于 1000 个，节点树深度少于 30 层，子节点数不大于 60 个。

在微信开发者工具中分析该页面两个模块存在大量的节点数。一个是筛选项模块，一个是长列表的模块。因为这部分功能较多，且结构复杂，我们采用了选择性渲染。如在用户浏览列表式，筛选项不生成具体节点。点击展开筛选的时候再渲染出节点，对于页面列表的体验有一定程度的缓解。另一方面，对于整体布局的书写上，有意识的避免嵌套过深的写法，如 RichText 使用，部分选择图片代替等。

3.4 优化筛选项相关

3.4.1 改变动画方式

在重构筛选项的过程中，发现在一些机型上，小程序的动画效果不太理想，比如当打开筛选项 tab 的时候，需要实现一个向下拉出的效果，早期在实现的时候，会出现两个问题:

- 动画会闪一下 然后再出现;
- 筛选页面节点过多时，点击响应过慢，用户体验差。

旧的筛选项的动画是通过 keyframes 方式实现了一个 fadeIn 的动画，加在最外层，但是无论如何在动画出现的那一帧，都会闪一下。分析下来，因为 keyframes 执行动画造成的卡顿：

```
.filter-wrap {
    animation: .3s ease-in fadeIn;
}

@keyframes fadeIn {
    0% {
        transform: translateY(-100%)
    }
    100% {
        transform: translateY(0)
    }
}
```

于是，尝试换了一种实现方式，通过 transition 来实现 transfrom：

```
.filter-wrap {
        transform: translateY(-100%);
        transition: none;
        &.active {
            transform: translateY(0);
            transition: transform .3s ease-in;
        }
}
```

3.4.2 维护简洁的 state

操作筛选项的时候，每操作一次都需要根据唯一 id 从筛选项的数据结构中循环遍历，去找到对应的 item，改掉 item 的状态，然后将整个结构重新 setState。官方文档中提到关于 setState，应该尽量避免处理过大的数据,会影响页面的更新性能。

针对这一问题，采取的办法是：

（1）复杂的对象扁平化,示例如下：

```
{
    "a": {
        "subs": [{
```

```
            "a1": {
                "subs": [{
                    "id": 1
                }]
            }
        }]
    },
    "b": {
        "subs": [{
            "id": 2
        }]
    },

    // ...
}
```

扁平化后的筛选项数据结构：

```
{
    "1": {
        "id": 1,
        "name": "汉庭",
        "includes": [],
        "excludes": [],
        // ...
    },
    "2": {
        // ...
    },

    // ...
}
```

不改变原有的数据，利用扁平化后的数据结构维护一个动态的选中列表：

```
const flattenFilters = data => {
    // ...

    return {
        [id]: {
            id: 2,
            name: "全季",
            includes: [],
            excludes: []
```

```
        // ...
      },

      // ...
    }
}

const filters = [], filtersSelected = {}
const flatFilters = flattenFilters(filters)

const onClickFilterItem = item => {

  // 所有的操作需要先拿到扁平化的 item
  const flatItem = flatFilters[item.id]

  if (filtersSelected[flatItem.id]) {
    // 已选中，需要取消选中
    delete filtersSelected[flatItem.id]
  }
  else {
    // 未选中，需要选中
    filtersSelected[flatItem.id] = flatItem

    // 取消选中排斥项
    const idsSelected = Object.keys(filtersSelected)
    const idsIntersection = intersection(idsSelected, flatItem.selfExcludes) // 交集
    if (idsIntersection.length) {
      idsIntersection.forEach(id => {
        delete filtersSelected[id]
      })
    }

    // 其他逻辑 （快筛，关键词等）
  }

  this.setState({filtersSelected})
}
```

上面是一个简单的实现，前后对比，我们只需要维护一个很简单的对象，对其属性进行添加或者删除，性能有细微的提高，且代码更为简单整洁。在业务代码中，类似这种通过数据结构转换提升效率的地方有很多。

关于筛选项，可以对比下检测的平均数据，减少 200ms~300ms，也会得到一些提升:

指标	setData 耗时旧	setData 耗时新	减少耗时百分比
长列表下筛选项展开	1023	967	5.47%
长列表下点击筛选项	1758	1443	17.92%

3.5 长列表的优化

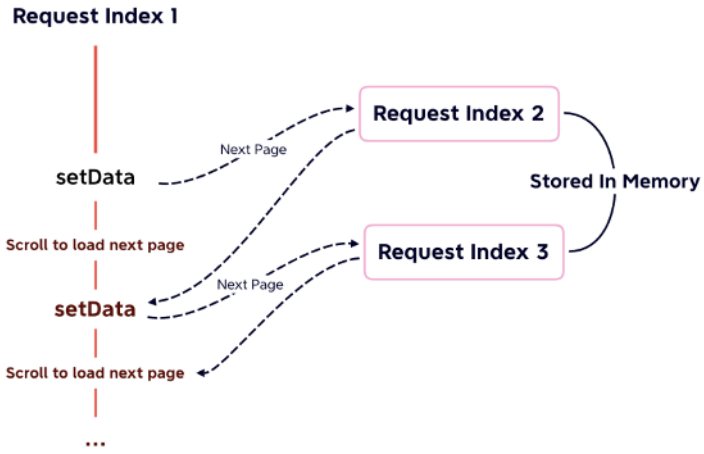
早期酒店列表页引入了虚拟列表，针对长列表渲染一定数目的酒店。核心的思路是只渲染显示在屏幕的数据，基本实现就是监听 scroll 事件，并且重新计算需要渲染的数据，不需要渲染的数据留一个空的 div 占位元素。

（1）一页有轻微的卡顿：

通过数据发现，下拉更新列表平均耗时 1900ms 左右：

指标	setData 次数	setData 耗时
下拉列表更新	3	1903

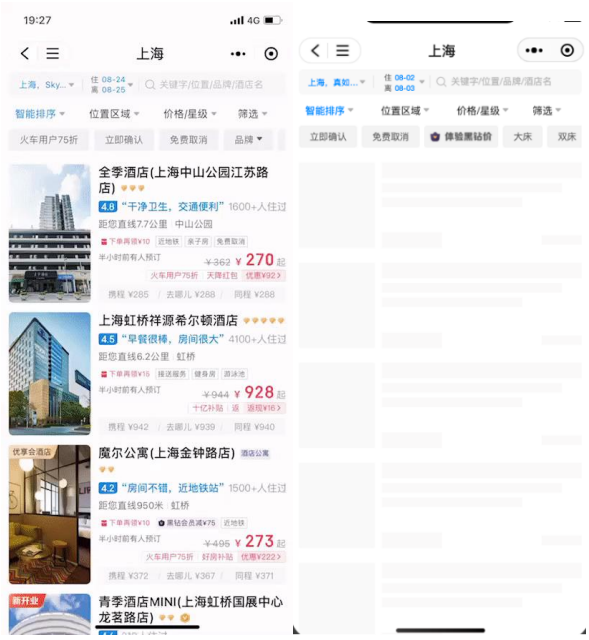
针对这个问题，解决方案是，提前加载下一页的数据，将下一页存入内存变量中。滚动加载的时候直接从内存变量中去取，然后 setData 更新到数据中。



（2）度过快会出现白屏（速度越快白屏时间越久，下方左图）：

虚拟列表的原理就是利用空的 View 去占位，当快速回滚的时候，渲染的时候当节点过于复杂，特别是酒店带有图片，渲染就会变慢，导致白屏，我们进行了三种方案的尝试:

① 动态的骨架图代替原有的 View 占位（下方图右）



② CustomWrapper

为了提升性能，官方推荐了 CusomWrapper，它可以将包裹的组件与页面隔离，组件渲染时不会更新整个页面，由 page.setData 变为 component.setData。

自定义组件是基于 Shadow DOM 实现的，对组件中的 DOM 和 CSS 进行了封装，使得组件内部与主页面的 DOM 保持了分离。图片中的#shadow-root 是根节点，成为影子根，和主

文档分开渲染。#shadow-root 可以嵌套形成节点树（Shadow Tree）

```
<custom-wrapper is="custom-wrapper">
    #shadow-root
      <view class="list"></view>
  </custom-wrapper>
```

包裹的组件被隔离，这样内部的数据的更新不会影响到整个页面，可以简单看下低性能客户端下的表现。效果还是明显的，同一时间点击，右侧弹窗出现的耗时平均会快 200ms ~ 300ms (同一机型同一环境下测出)，机型越低端越明显。

（右侧是 CustomWrapper 下的）

③ 使用小程序原生组件

用小程序的原生组件去实现这个列表 Item。原生组件绕过 Taro3 的运行时，也就是说，在用户对页面操作的时候，如果是 taro3 的组件，需要进行前后数据的 diff 计算，然后生产新的虚拟 dom 所需要的节点数据，进而调用小程序的 api 去对节点进行操作。原生组件绕过了这一些列的操作，直接是是底层小程序对数据的更新。所以，缩短了一些时间。可以看一下实现后的效果：

指标	setData 次数(旧)	setData 次数(新)
下拉列表更新	3	1

setData 次数(旧)	setData 次数(新)	减少耗时百分比
1903	836	56.07%

可以看出原生性能提升很大，平均更新列表缩短 1s 左右，但是使用原生也有缺点，主要表现为以下两个方面:

- 组件包含的所有样式 需要按照小程序的规范写一遍，且与 taro 的样式相互隔离;
- 在原生组件中无法使用 taro 的 API，比如 createSelectorQuery 这种。

对比三种方案，性能提升逐步加强。考虑到使用 Taro 原本的意义在于跨端，如果使用原生，就没办法达到这个目的，不过我们在尝试是否可以通过插件，在编译时生成对应原生小程序的组件代码，以此解决这一问题，最终达到最优效果。

3.6 React.memo

当复杂页面子组件过多时，父组件的渲染会导致子组件跟着渲染，React.memo 可以做浅层的比较防止不必要的渲染:

```
const MyComponent = React.memo(function MyComponent(props) {
  /* 使用 props 渲染 */
})
```

React.memo 为高阶组件。它与 React.PureComponent 非常相似，但它适用于函数组件，但不适用于 class 组件。

如果你的函数组件在给定相同 props 的情况下渲染相同的结果，那么你可以通过将其包装在 React.memo 中调用，以此通过记忆组件渲染结果的方式来提高组件的性能表现。这意味着在这种情况下，React 将跳过渲染组件的操作并直接复用最近一次渲染的结果。

默认情况下其只会对复杂对象做浅层对比，如果你想要控制对比过程，那么请将自定义的比较函数通过第二个参数传入来实现。

```
function MyComponent(props) {
  /* 使用 props 渲染 */
}

function areEqual(prevProps, nextProps) {
  /*
  如果把 nextProps 传入 render 方法的返回结果与
  将 prevProps 传入 render 方法的返回结果一致则返回 true,
  否则返回 false
  */
}

export default React.memo(MyComponent, areEqual);
```

四、总结

本次复杂列表的性能优化我们前后经历较久，尝试了各种可能的优化点。从列表页的预加载，筛选项数据结构和动画实现的改变，到长列表的体验优化和原生的结合，提升了页面的更新和渲染效率,目前仍密切关注，继续保持探索。

提升 50 分，Trip.com 机票基于 PageSpeed 的前端性能优化实践

【作者简介】Patrick，携程资深前端开发工程师，专注于前端工程化和性能优化。

前言

网站性能对于用户体验、转化率和流失率、SEO 排名等至关重要，Trip.com 主要用户来自海外，对网站访问性能有更高的要求。能够快速响应的网站通常有机会获取更多流量，并为用户带来更好的体验。

近期我们对 Trip.com 机票站点做了一版性能优化，通过对主要 landing 页面进行系统优化，将页面的 PageSpeed 评分从原本 30 左右提升到 80 分以上。

这里分享在优化过程中的一些经验，将从性能指标、性能测量与优化实践方案三个方面展开，期望可以给大家提供一些思路和参考。

一、性能指标

1.1 性能指标的发展与演进

针对线上项目做性能优化，首先需要有一个确定的可量化的评判标准，用来判断优化工作是否有效。

1.1.1 传统的性能指标以及它们存在的问题

传统的性能指标最典型的是 DOM Ready 时间和页面加载时间（load time）：前者指的是初始 HTML 文档完全加载和解析完成的时间，一般是通过 DOMContentLoaded 事件获得；后者指的是整个页面所需的资源（包括脚本、样式、图片等）加载完成的时间，通过全局的 load 事件获取。

普遍存在的问题是：在早先前后端耦合的时代，通过在服务端使用模板引擎渲染出 HTML，能比较好地反映网站性能。后来前端领域的迅猛发展，尤其是随着客户端渲染方案的盛行，以及各种动态技术的大量运用，这两个指标差不多已经失去其原有的意义，无法准确反映性能。

1.1.2 指标和用户实际感受之间的差异

再往后，采用浏览器提供的 Navigation Timing API ，通过 performance.timing 获取从页面开始加载到结束全过程中不同阶段的时间点。用这种方法，开发者可从多个维度去定义一些指标，通过简单的差值计算得到数据，并以此去监控站点性能。

比如在携程现有的 UBT（User Behavior Tracking）中基于 Timing API 主要定义了以下 7 个关键指标 DNS、Connect、Request、Response、Blank、Domready 和 Onload。

- DNS (domainLookupEnd - domainLookupStart)
- Connect (connectEnd - connectStart)
- Request (responseStart - requestStart)
- Response (responseEnd - responseStart)
- Blank (domInteractive - responseStart)
- Domready (domContentLoadedEventEnd - navigationStart)
- Onload (loadEventEnd - navigationStart)

1.1.3 以用户为中心的性能指标

上述这些指标更侧重于技术角度，跟用户在实际使用过程中的真实感受会有偏差。以此为标准去做性能优化的话，很可能面临的一种场景是，已经把某些特定指标如加载时间的数值大幅减少，但用户体验仍然不佳。基于此，Chrome 团队和 W3C 性能工作组推出了一组 以用户为中心的性能指标，从用户角度更好地去评判页面性能。

这些指标主要包含：

- FCP，First Contentful Paint 首次内容绘制
- LCP，Largest Contentful Paint 最大内容绘制
- TTI，Time to Interactive 可交互时间
- TBT，Total Blocking Time 总阻塞时间
- CLS，Cumulative Layout Shift 累积布局偏移

1.2 指标介绍

接下来简单介绍下主要性能指标的具体定义：

1.2.1 FCP

FCP 指标测量的是页面从开始加载到页面内容的任何部分在屏幕上完成渲染的时间。“内容”可以是文本、图像（包括背景图像）、<svg> 元素或非白色的 <canvas> 元素。

这个指标回答了一个用户问题，应用正在运行吗。

还有一个从名称上很接近的指标，FP （首次绘制），它们之间的区别如下：

- FP first-paint 大致可以认为是白屏时间
- FCP first-contentful-paint 大致可以认为是首屏时间

1.2.2 LCP

这个指标对应的关键用户问题是，内容是否有用，即页面是否已经呈现出对用户有用的内容。

早先有过一些类似的指标比如 FMP （首次有效绘制），但有效绘制的定义是什么通常很难解释，而且算法容易出错。

相反，最大内容绘制的定义简单明了，这里的“内容”和 FCP 中的定义基本一致，指的是在可视区域内的最大图片或文本块完成渲染的时间。

元素大小指的是内容占据的面积大小，即 size = width * height ，不包含边距边框。

大多数情况下，页面上最吸引用户的内容往往就是最大元素，可以视为页面中最重要的内容。

1.2.3 TTI

可交互时间，对应的用户关注点是可以使用吗。

早期，关于可交互时间一直并没有一个清晰明确的定义。刀耕火种的时代，开发者通过自定义时间节点，并在代码中埋点来获取相关数据。

比如通过在 setTimeout 中放一个任务获取执行时间点，再计算到页面开始加载的差值。

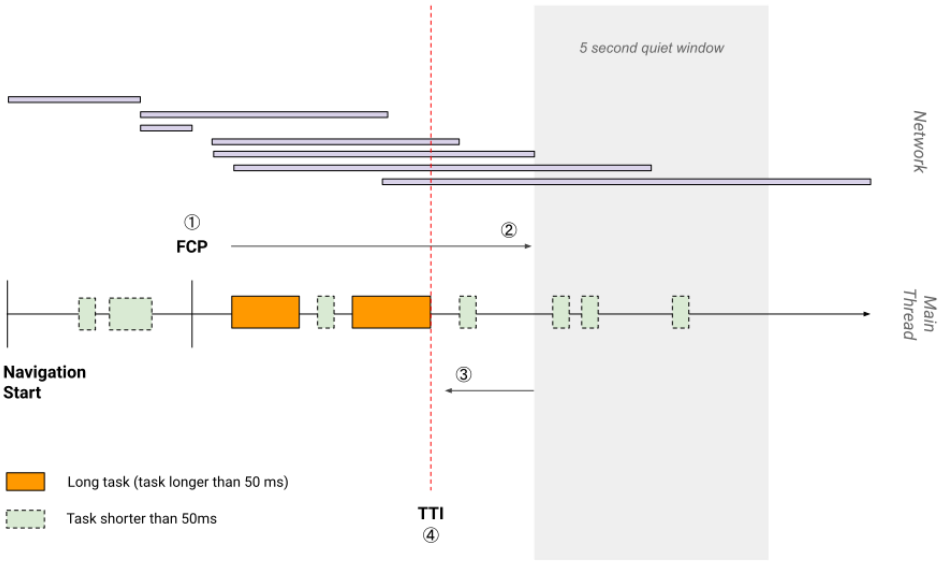
```
setTimeout(function() {
    tti = new Date() - navigationStartTime
}, 0)
```

或者，在使用 React 等特定框架时，通过向主要组件的生命周期函数 componentDidMount 埋点，并以此计算 TTI 时间。

而在 Lighthouse 中，可交互时间指标有了更通用、标准化的定义。TTI 应从 FCP 时间点开始沿时间轴查找，如果出现 5 秒的静默窗口（没有长任务并且不超过 2 个正在处理的 GET 请求），那么最后一个长任务执行结束的时间点即为可交互时间。

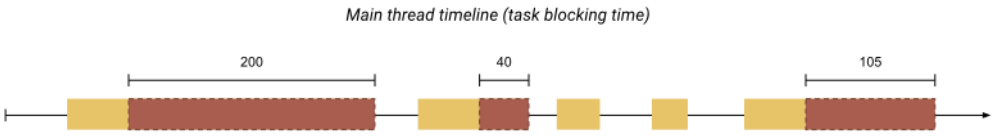
长任务指的是执行时间超过 50 ms 的任务。

定义的根据是，主线程上若不存在导致阻塞状态的长任务，则视为此时已可以响应用户交互。



1.2.4 TBT

TBT 和 TTI 是一对配套指标，用于衡量在页面可交互之前的阻塞程度。TBT 是指在 FCP 和 TTI 之间所有长任务超过 50ms 的部分的时间总和（注意不是长任务的时间总和）。



1.2.5 CLS

累积布局偏移指标用于衡量页面视觉稳定性。单次布局偏移分数是影响分数（不稳定区域占可视区域的百分比）与距离分数（不稳定元素最大位移距离占比）的乘积。CLS 指标本身一直在不断进化 ，便于更加准确地去衡量布局偏移对用户的影响。

1.2.6 其他

- SI，Speed Index 速度指数，属于 Lighthouse 六大性能指标之一，使用 speedline 模块来衡量视觉进度；
- TTFB，Time to First Byte 首字节时间，用于衡量服务器响应能力，所有请求包括页面、脚本 和 AJAX 等都可以统计；
- FID，First Input Delay 首次输入延迟，由于主线程繁忙导致用户首次输入的延迟时间；
- FCI，First CPU Idle 首次 CPU 空闲，与 TTI 指标相似，目前已不推荐使用。

二、性能测量

了解需要关注的性能指标之后，面临的问题是应该怎么样去有效测量呢?

性能测量分两种类型，实验室测量与现场测量（真实用户监控)。部分指标只能通过实验室测量，或是只能现场测量。

2.1 实验室测量

实验室测量指的是在一个受控环境下,使用预定义的硬件设备和网络配置等规则去运行网站页面,进行性能数据采集,提取性能指标。目前最流行的工具是 Google 提供的 Lighthouse ，最初作为一个独立浏览器扩展程序推出，需开发者自行安装（支持 Firefox)，目前已经集成到 Chrome DevTools 。Lighthouse 不仅仅是一个性能测量工具，除此之外还提供 PWA 、SEO 、可访问性、最佳实践等审计报告。

在做性能优化的时候，如何有效评估优化方案的效果是一个问题，由于还未发布到线上环境无法采集真实用户性能数据，这时使用工具进行实验室测量就显得尤为重要。

同时，Lighthouse 提供开源 CI 工具 Lighthouse CI ，开发者能自行部署服务，并集成到现有的 CI 体系中。

2.2 现场测量

现场测量，也称真实用户监控（RUM)，即实时采集真实用户性能数据。

实验室测量的是在一系列特定条件下的性能数据，不能完全反映现实世界中用户的真实情况。现场测量的优势在于样板量足够大,包罗各种不同设备不同网络环境下的用户性能数据,从统计上更能反映真实性能情况。缺点是，现场测量需基于浏览器提供的性能 Web API ，受限于当前设备采集到的数据不及实验室测量丰富。

2.3 定量评估的问题与方案

定量评估每一项优化方案的效果并不容易，原因包括环境差异问题，分数计算问题等。

解决方案是:

- 开发模式启动站点应用与生产模式差别较大,将应用发布到独立测试服务器再进行性能测量;
- 本地启动 Lighthouse 进行测量，在不同时间的系统状态差异较大，部署测量工具到特定服务器;
- 由于环境影响单次测量的差异可能很大，基于 lighthouse NPM 包一次性跑 10 次，去除最大值和最小值之后再取中位数和平均值作为参考;
- 性能分数由 6 大性能指标计算而来，单项指标的数值优化最终在分数上体现可能没有差异，分开对比具体指标数值。

三、性能优化方案

确定优化方向，并且有了可定量评估的方案之后，接下来要做的就是如何实施具体的优化方案。

性能优化是一个老生常谈，同时与时俱进的主题。早期大名鼎鼎的 雅虎 35 条性能军规 到现在大部分仍然适用，另一方面随着技术的发展，基于上述以用户为中心的性能指标，能更有针对性地实施方案。同时借助 Lighthouse 工具，能更有效地评估具体方案效果。

我们的 Web 应用基于 React 技术栈，以下部分内容基于 React 来进行阐述。

3.1 减小包体积

Web 应用与传统客户端应用很不同的一点在于，应用所需资源文件都是存放在远端服务器上的，每次访问都有相当大的性能开销是用于资源获取。

如何让资源高效加载成了一个非常重要的问题，其中最重要的一环是网络传输，专用 CDN 服务器包含就近访问，资源缓存和传输体积压缩等功能，能节省大量网络传输时间，这是基础设施的角度。

从应用开发者的角度，首先可以对应用包体积进行瘦身。

包体积的问题主要表现为：

- 不再使用的冗余代码
- 复制粘贴的重复代码
- 非必要的大体积类库
- 未经优化的图片文件

3.1.1 冗余代码的优化

冗余代码的产生有多种，比如是已经废弃不用但仍然被导入的功能模块，或者是在做 AB 实验完成后未完全移除的版本代码等。

借助相关工具，比如 Webpack 插件 webpack-bundle-analyzer 能用一种可视化的方式呈现每个包的具体模块信息，体积大小、依赖关系一目了然。而 Chrome DevTools Coverage 工具能分析出运行过程中文件（脚本和样式）的使用情况，可作为参考更好地针对性地瘦身优化。

3.1.2 重复代码的优化

重复代码很大一部分是实现相似功能的过程中，直接复制粘贴一方代码进行修改导致，借助 jsinspect 可以检测到相同和相似代码，然后进行合理抽象。还有一种情况是，依赖 NPM 包提供多种方式的代码，比如 dist 目录下的打包代码，lib 目录下的 CommonJS 代码，和 es 目录下的 ES Modules 代码。若是不小心在不同地方引入不同方式的包，就等同于是引入重复功能模块。更甚一步，在跨团队合作中依赖包只提供打包版本，也会出现 babel polyfill 代

码多次重复，并且无从分析。解决方案是制定统一的标准，推荐 NPM 包都提供仅 babel 编译不打包版本。

3.1.3 类库开销的优化

在类库的使用上同样需要注意，比如仅使用一两个方法就引入整个 lodash 库，推荐做法是按需引入，不用改变写法加入 babel-plugin-lodash 这类插件就能在代码构建时转换。另外一种情况是引入 moment 这类体积较大的库用作时间处理与格式化，可以视实际情况采用体积更小的替代品。对于更简单的需求，则完全可以基于原生 API 自行实现封装一些方法。

3.1.4 图片文件的优化

未经优化的图片可高达几百 KB ，应在保证图片清晰度的情况下合理压缩体积。

另一方面，为现代浏览器提供有更高效压缩算法的图片格式，相比传统的 PNG 和 JPG 格式，WebP 在同等质量下有更小的体积，注意做好降级方案。

3.1.5 更激进的做法

关于编译构建，传统做法为了浏览器兼容性引入很多非必要的 polyfill ，解决方案是提供一个动态 polyfill 脚本，根据当前客户端提供不同内容的 polyfill 。另一方面，现代浏览器已经支持越来越多的 ES6+ 语法特性，针对这部分用户可提供 ES6+ 版本的代码，并使用 ES Modules 格式，从而大幅减小包体积。

3.2 优化资源加载

做好包体积优化能节省网络传输时间，以及一部分代码执行时间，但更重要的是让资源有效加载，可从资源加载顺序和优先级方面着手。

3.2.1 Resource Hints

为了使页面可以快速加载，我们基于 PRPL 模式 进行优化。PRPL 是四个词的首字母缩写，分别代表：

- Preload 预加载最重要的资源
- Render 尽快渲染初始内容
- Pre-cache 预缓存其他资源
- Lazy load 懒加载其他路由和非关键资源

首先，我们需要优化关键路径资源，页面中要呈现的内容很多，但不是所有内容都需要第一时间呈现，应优先呈现最重要的内容。浏览器并不知道哪些资源是最重要的，基于 Resource Hints 可以告诉浏览器资源优先级。常用的有以下几类：

- preconnect 启动早期连接，包括 DNS 查找，TCP 握手等；

- preload 预加载资源并缓存，以便需要时立即使用；
- prefetch 预获取资源，优先级比 preload 低，浏览器自行判断合理时间执行操作。

在使用过程需要注意：

- 不要无限制的滥用，因为使用本身会消耗资源，尤其是添加了但却未使用；
- 资源设置 crossorigin ，对应预处理提示也要设置，否则两者不匹配导致重复加载。

3.2.2 Service Worker

使用 Service worker 缓存预载资源，对后续访问会有极大的性能提升，能节省大量网路传输开销。

在项目中推荐采用 Google 提供的 Workbox 库，可以通过配置的方式对不同类型资源应用不同缓存策略。

Service Worker 带来的优化效果不能从 PageSpeed Insights 网站上的分数直接体现，因为 PageSpeed 总是单次分数并且不使用缓存。

3.2.3 优化加载第三方脚本

应用依赖的第三方脚本通常会减慢页面加载速度，一般采用以下方式：按需加载和延迟加载。

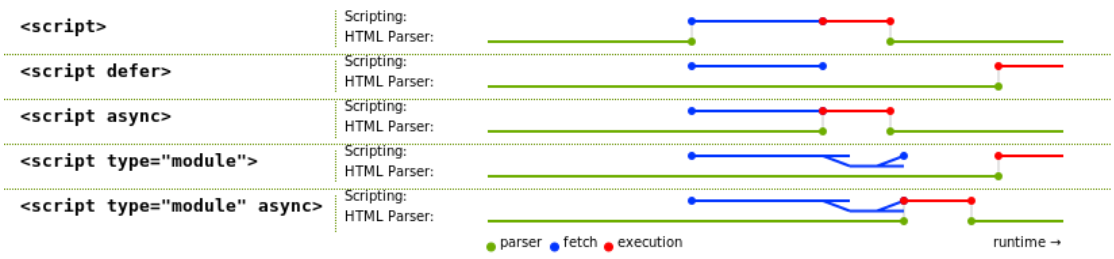
（1）按需加载

需用户交互才用到的功能模块应按需加载。举个例子，用户登录时要调用一个第三方验证模块，就没必要在页面一开始就引入该脚本，在用户执行登录操作时引入更合理。

（2）延迟加载

像是 Google analysis 和合作商营销等第三方日志埋点脚本，由于业务需要无法移除，加载后占用大量性能资源。

由于这类脚本和应用没有依赖关系，可使用 defer script 延迟脚本的解析执行。更进一步，延迟到在可交互时间之后加载就基本不会有任何影响。



3.3 组件懒加载

可视区域之外的内容和需要用户交互时才呈现的组件，都可采用懒加载，保证页面首要内容快速呈现。

要做懒加载，首先需要合理定义拆分点进行代码分割，然后基于动态导入和 React.lazy 即可实现。

对于大部分点击触发的组件来说，这样已经足够，但针对页面底部可视区域之外需常规滚动查看的内容，还要做一些额外的工作。可以自行封装实现一个组件，在内部进行判断内容是否可视，并监听 scroll 事件重新渲染。

实际中，我们结合 react-lazyload 和 @loadable/component 实现所需功能，如下：

```
import React from 'react';
import loadable from '@loadable/component';
import LazyLoad from 'react-lazyload';

const LazyComponent = loadable(() => import(/* webpackChunkName: "home_lazy" */
'./LazyComponent'));

export function HomePage() {
    return (
        <>
            <MainComponent />
            <LazyLoad>
                <LazyComponent />
            </LazyLoad>
        </>
    );)
}
```

懒加载可能导致懒加载组件自身体验下降，可对用户比较频繁使用的组件进行预加载。

过度拆分可能会产生很多体积很小的包，可适当地进行合并。借助 webpack magic comment ，配置相同 chunk name 即可合并打包。

```
import loadable from '@loadable/component';

export const SortLayer = loadable(() => import(/* webpackChunkName:
"depart_select_layer" */ './SortLayer'));
export const StopLayer = loadable(() => import(/* webpackChunkName:
"depart_select_layer" */ './StopLayer'));
export const TimeLayer = loadable(() => import(/* webpackChunkName:
```

"depart_select_layer" */ './TimeLayer'));

3.4 优化渲染方式

3.4.1 服务端渲染

CSR （客户端渲染）的最大问题在于受用户环境影响太大，一方面是网络层面脚本文件的加载，一方面是浏览器的执行效率，不同环境下差异很大。

SSR （服务端渲染）则能解决这个问题，直出 HTML 能快速呈现页面主要内容，能很好地改善 FCP 和 LCP 指标。

SSR 的优势相对 CSR 主要有两点：

将渲染（这里是指 JavaScript 执行层面的）工作转移到服务端，毕竟服务端相对更可控

在首屏之前避免减少资源网络传输，从而减少耗时，网络是更不可控的一个因素

实际上，大部分时候都是结合二者，针对首屏采用服务端渲染，让用户更快看到内容，其他仍使用客户端渲染的模式，减轻服务器压力，毕竟将大量用户的渲染任务转移到服务端会是一笔不小的开销。这时，结合缓存机制可以大幅节省渲染时间。

3.4.2 预渲染

基于构建时的预渲染，是使用 webpack 和 babel 等工具提前生成对应的 HTML 以及引用的脚步和样式文件。还有一种方式是基于运行时的，使用 headless 浏览器。但预渲染并不适用于有大量动态内容的页面。

实际应用场景是针对信息展示的内容页面。

3.5 优化长任务

Long Task （长任务）的定义是执行时间超过 50 ms 的任务。我们知道，JavaScript 是单进程单线程的模型，主线程上一旦有耗时长的任务存在时，就会造成阻塞，无法响应用户输入。

Long Task 跟 Lighthouse 中的两个重要性能指标 TTI 和 TBT 息息相关，而这两个指标占比为 40% ，可以说优化好 Long Task 能大幅提升页面性能。

Long Task 可借助对应的 Long Task Web API 进行监控，开发过程中则使用 Chrome DevTools Performance 面板进行查看和调试。需要注意的是，实际用户尤其是移动端的用户环境通常不够乐观，调试过程应适当调低硬件配置和网络速度，模拟大部分用户的实际使用情况，发现并优化更多的 Long Task 。

任务类型有多种，除了最常见的脚本执行之外，还包括脚本解析编译、HTML 解析、CSS 解

析、布局、渲染等。脚本执行是长任务的主要表现形式，因此着重说明在 JavaScript 执行上的一些优化方式：

- requestIdleCallback API
- Web Worker
- 记忆函数
- Debounce 和 Throttle

3.5.1 requestIdleCallback API

针对一些不重要的任务比如埋点日志可以直接丢到 requestIdleCallback 中，浏览器会在空闲时间执行。在不支持的环境可使用 shim ，基于 setTimeout 实现近似的功能。

库 idlize 中封装了一些非常实用的帮助函数，使用这些方法可把任务延迟到需要的时候再执行。

3.5.2 Web Worker

如果项目中确实存在比较复杂的计算，可启动 Web Worker 单独另开一个线程来计算，并使用 message 通信。

3.5.3 记忆函数

如果一个函数被大量调用，合理运用记忆函数一个很好的选择，有大量的库可供我们选择，也可以根据使用场景自行实现。

3.5.4 Debounce 和 Throttle

针对 input change 和 scroll 等可能频繁触发的事件，结合 Debounce 或 Throttle 避免无节制地调用。

3.6 React 性能优化

在 React 框架使用上有一些性能优化的实践，常用的有：

- 使用 PureComponent 和 Memo 避免不必要的重新渲染，复杂场景适当使用 shouldComponentUpdate 或是 areEqual 方法；
- 函数组件可使用 useMemo 用于记忆计算结果，其他场景可引入外部库如 reselect 简化处理；
- 在更新 state 深层嵌套数据时避免使用深拷贝，可借助 immer 这类库来处理不可变数据；
- 保持 DOM 结构简洁，避免层级过深。比如，最简单的一个点是使用 React.Fragment。

最后说明一点，仅在必要的时候进行性能优化，大部分情况下无需考虑，保持简洁和可维护

性更重要，而且滥用方法反而损害性能。

3.7 减少布局偏移

（1）如何调试监控

使用对应的 Layout Instability API 可收集用户的布局偏移数据。

在开发调试中，Layout Shift 同样可以使用 Chrome DevTools Performance 进行分析，能查看每一次布局偏移的分数，进行针对性优化。

（2）常用的优化方案

- 为动态元素预留静态空间
- 图片宽高尺寸固定

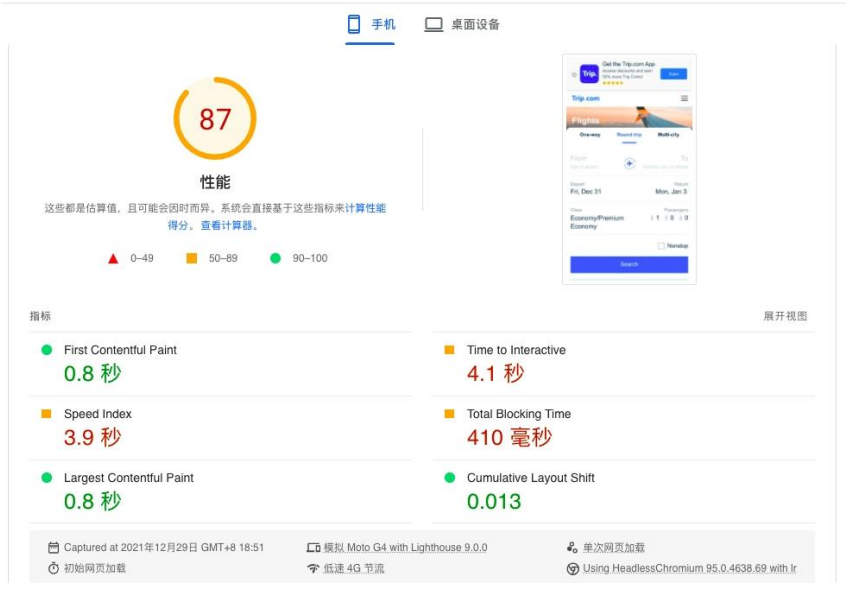
预留空间可减少其他页面元素的偏移，比如出现在最顶部的广告位，在数据还未获取到的时候预先设置好一个容器，避免后续大幅偏移。

针对整页动态的内容，使用骨架屏是一种很好的模式，业界已有不少成熟方案可自动生成。

设置图片宽高，可保证浏览器在加载图片过程中始终能分配正确的空间大小。

四、总结反思

借助上文中提到的性能测量方式，我们逐步实施优化方案并发布上线，经过近两个月断断续续的时间，最终让性能分数稳定在 80 分以上。



性能优化也适用于二八定律，优化方式很多，简单堆砌叠加使用很可能适得其反。不同场景下的优化方案千差万别，关键在于找准最核心的问题。

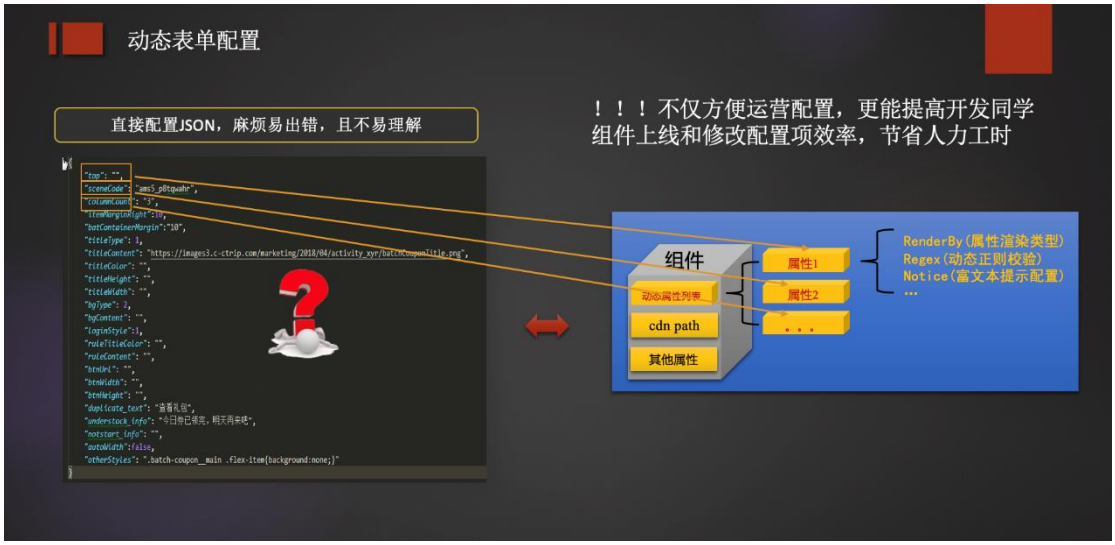
以上仅提供一些思路作为参考，部分方案对特定指标效果很好，部分方案不会反映到指标分数，但有助提升用户体验。再者，指标衡量的是单个页面速度，而作为开发者还应考虑到后续页面，从应用维度去平衡，真正从用户角度考虑。

携程动态表单 DynamicForm 的设计与实现

【作者简介】Daryl，携程高级研发经理，关注业界大前端及高并发应用解决方案。

一、简介

在很多软件系统中，表单开发都是很重要的一个部分。在表单开发中，往往会遇到重复开发的问题，例如在页面搭建系统中，除了组件本身的逻辑，配置组件数据的表单通常也需要开发人员重复手动开发。这就导致开发人员不仅要维护组件本身的逻辑，还要维护组件的配置表单，严重影响组件的开发和迭代效率。



为了让开发人员更加专注于组件本身的逻辑处理，我们开发了 DynamicForm 动态表单配置系统，可以通过拖拽的方式，快速创建一个表单。

DynamicForm 是由携程市场营销"活动平台"及"会员平台"共同设计的 React 表单组件，它包括表单可视化设计、校验、预览、渲染等功能。目前最重要的应用场景，是为乐高平台提供组件属性配置的动态化表单配置能力。

乐高平台是探索组件化构建页面的实际应用，期望通过归纳常规 web 组件和业务组件，归纳单一职责服务接口从而构建出通过配置自动生成 h5 页面的方案，以达到代码复用，逻辑复用，节省开发时间，经验积累，节省页面上线时间等目的。

二、乐高早期表单的实现

2.1 实现阶段 1：手动开发

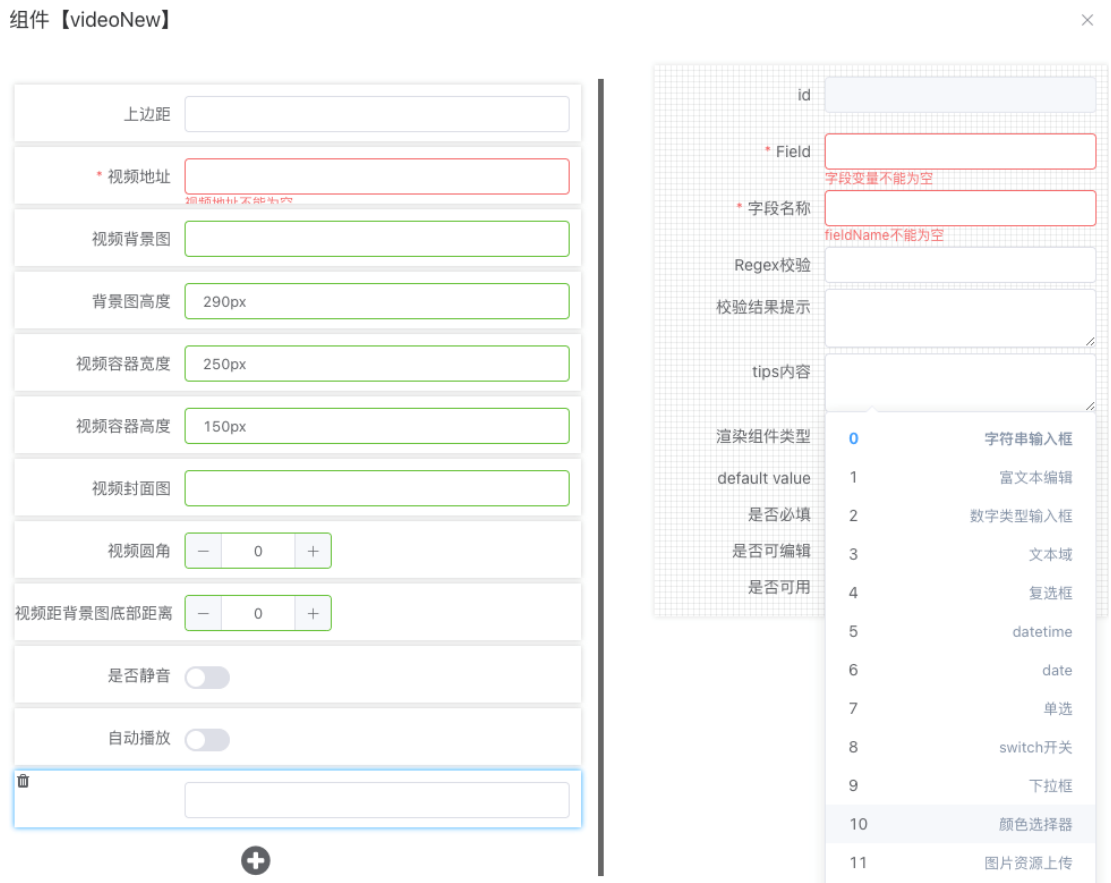
乐高最初完全使用定制化开发的形式来实现属性表单，这样做的好处是表单外观可以随心所欲，界面对于运营可以做的非常友好。当然，缺点也是显而易见的，定制化的开发无法满足快速迭代的活动运营要求，往往一个需求来了立马就要上线，这个时候定制化的开发和发布

流程就会严重制约生产力。

2.2 实现阶段 2：动态表单 1.

乐高表单第二版使用的是半自动化表单，支持动态配置表单控件类型和表单校验等。

配置面板如图：



此版本的表单不支持可视化拖拽，控件自上而下顺序排列。解决了需求快读迭代的问题，但因为无法自定义布局，随之带来了属性的臃肿以及运营人员操作复杂的问题。

因此我们开发了动态表单 2.0（DynamicForm）。

2.3 DynamicForm 在乐高平台的应用

为了应对乐高组件快速迭代的业务需求，必须研发出一种能够让组件属性快速得到应用的表单技术框架，这样以保证在组件新增属性时，无需进行新代码的开发，仅需通过简单的配置即可生成新的组件属性。

为了达到表单配置的灵活性，DynamicForm 必须满足以下几个条件：

（1）丰富的表单控件类型

DynamicForm 包含了以下内容，满足了多样化的配置要求。

- 通用控件：文本框、单选框、多选框、下拉选择、颜色选择、图片上传等
- 自定义控件：组件可视化选择，热区定义，JSON 可视化配置控件等。

（2）DIY 表单界面

运维人员可以通过对控件拖拽，实时编辑的形式，对表单进行自由设计，以达到理想的 UI 效果。

（3）表单控件配置

可以对控件的默认值、是否必填、提示信息、控件宽度、正则匹配等进行自由配置，以达到理想效果。

4）表单数据关联

为了达到表单的属性项的关联，需配置数据关联，控制分组关联等。

配置界面示例见下图：

二、亮点

已实现的 DynamicForm 具有如下亮点：

- 可视化：可视化搭建、修改和预览表单；
- 可拖拽布局：控件可在画布内拖拽至任意坐标，以搭建最佳布局；

- 可扩展：可二次开发，可扩展控件集；
- 可联动：某个控件可以控制别的控件的显示和隐藏；
- 支持复杂数据类型：支持对象结构以及对象数组结构等复杂数据类型(JSON)的配置。

三、架构

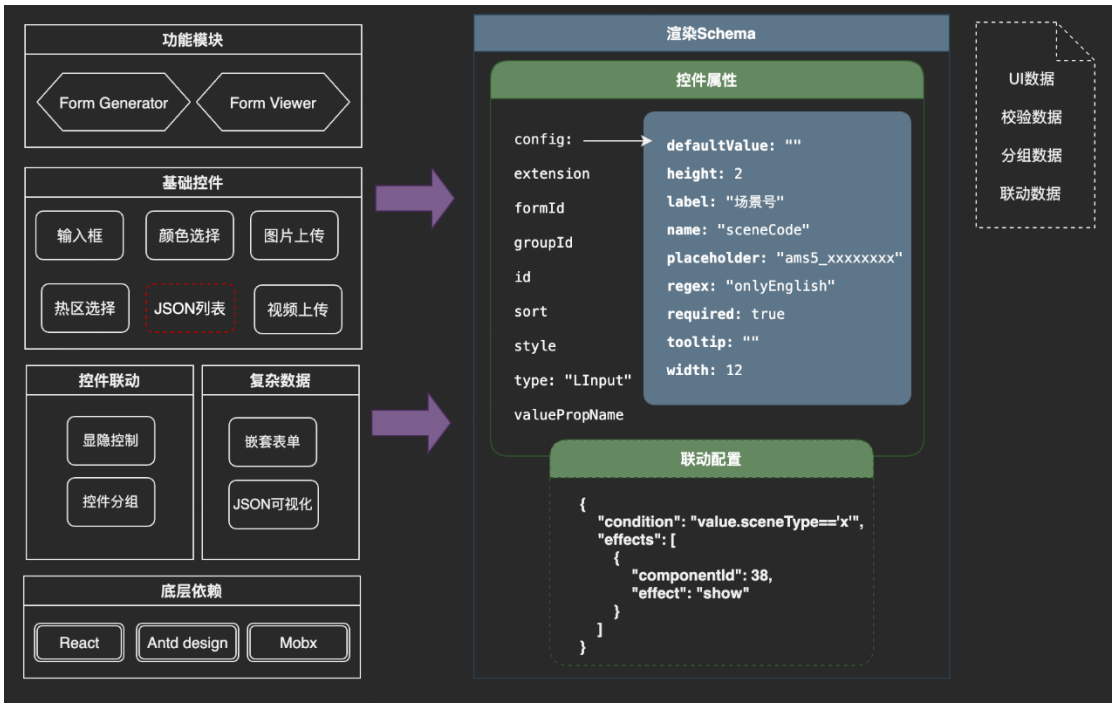
DynamicForm 架构设计的目的是降低表单的维护成本、降低开发人力，解决表单开发中常见的联动、校验、自定义展示等问题。

3.1 功能梳理

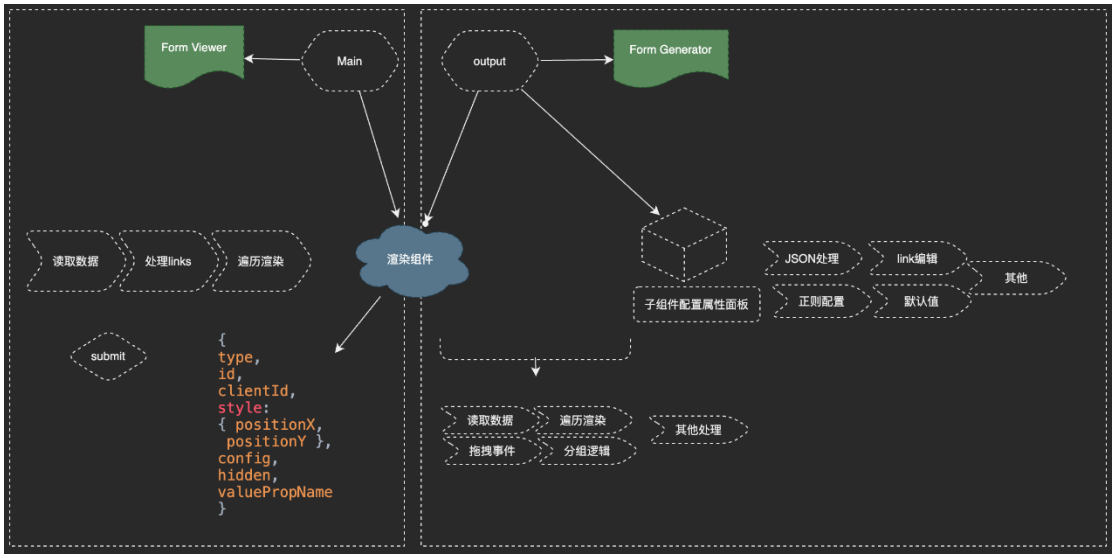
Dynamic 的核心功能包括联动、校验、扩展、可视化等。

- 联动：某个控件变化后，改变其他控件的显示和隐藏。
- 校验：通过正则表达式（预留或者自定义）进行表单校验。
- 扩展：支持自定义控件和校验。
- 可视化：支持可视化拖拽和可视化编辑，节省开发表单时间，并降低运营的配置成本。

3.2 架构模块



渲染流程：



系统有表单生成器编辑面板 Form Generator，表单渲染入口 Form Viewer 两个主要模块。

这两个模块共用常规的基础组件如输入框, 颜色选择等, 还有一些基于业务扩展的复杂组件, 例如热区选择，视频上传，数据聚合（JSON 列表）等。

动态表单一个比较重要的点是需要解决 JSON 可视化配置，为此表单也开发了 table 列表式的 JSON 列表组件，子项的配置就基于嵌套表单实现配置字段，并且能够增删改查条目，excel 导入，导出数据。

其他复杂数据类型配置，如["a","b"]和["a","c"]，表单提交数据结构即为{a:{b:"",c:""}}

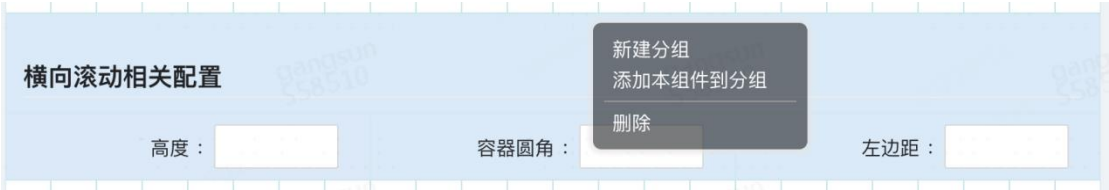
另外一个功能点是解决动态属性间的联动问题，为此表单通过配置联动表达式解决了控件联动问题，例如控制显隐等。


```
 * 表单联动处理
 */
const handleFormLink = () => {
    // @ts-ignore
    const value = form.getFieldsValue()
    console.log(Object.keys(value).length,value)
    // a(value) //使用value，防止编译时value被删除
    const {links = []} = localeStore.formConfig
    const conditions = {}
    links.map(item => {
        const {condition, effects = []} = item
        try {
            console.log("Main.tsx--condition---",condition);
            const result = !!eval(condition);
            // const result = !!new Function('', `${"return ("+ condition + ")"}`)();
            effects.map(({componentId, effect}) => {
                if (componentId !== undefined) {
                    switch (effect) {
                        case 'show': {
                            conditions[componentId] = conditions[componentId] ? {
                                ...conditions[componentId],
                                hidden: !result
                            } : {hidden: !result}
                            break
                        }
                    }
                }
            })
        } catch (e) {
            console.error(e)
        }
    })
    const tempComponentList = localeStore.currentComponentList.map(item => {
        if (conditions[item.clientId]) {
            item.hidden = conditions[item.clientId].hidden
        } else if (item.groupId && conditions[item.groupId]) {
            item.hidden = conditions[item.groupId].hidden
        }
        return item
    })

    localeStore.currentComponentList = autoFillBlankLine(tempComponentList)
}
```

表单底层则依赖 React（hooks），Ant Design 的主题 UI 库，Mobx。

3.3 灵活的布局

- 组件自由拖拽布局，自动对齐等;
- 组件可控制添加分组，从而批量操作布局。



3.4 校验

提供预留的常规校验，如中文，英文校验等，以及可自定义扩展的校验配置。

四、后续计划

DynamicForm 将作为独立的 npm 模块使用，为其他动态表单场景提供公共功能，打造泛应用动态表单。

DynamicForm 代码开源，与社区共同交流与进步。

携程机票前端 Svelte 生产实践

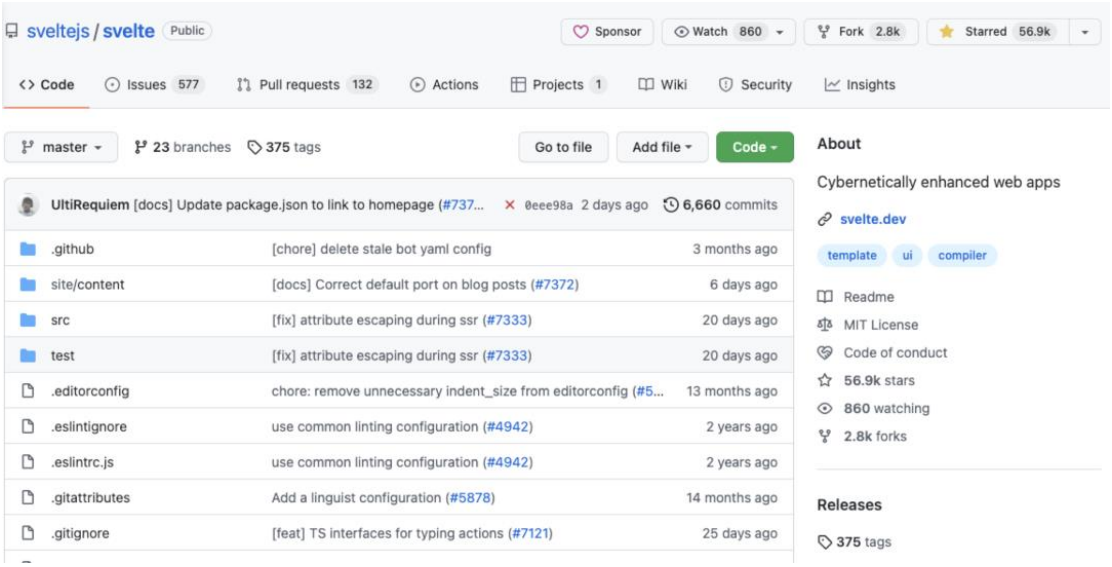
【作者简介】shuan feng，携程高级前端开发工程师，关注性能优化、低代码、svelte 等领域。

一、技术调研

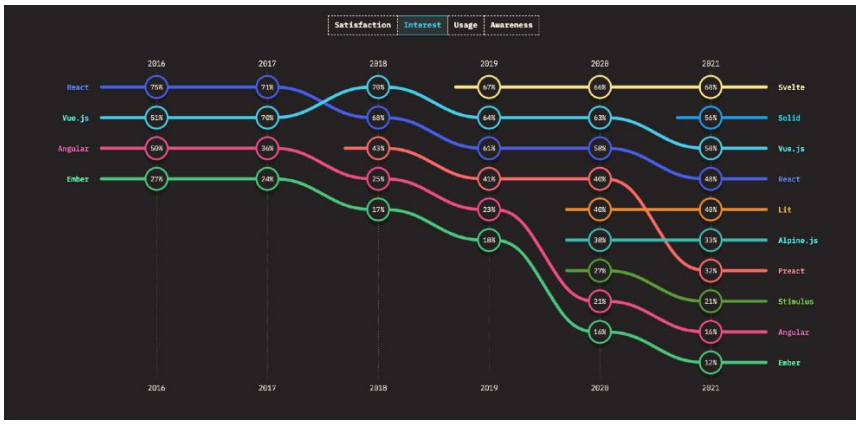
最近几年，前端框架层出不穷。近两年，前端圈又出了一个新宠：Svelte。作者是 Rich Harris，也就是 Ractive, Rollup 和 Buble 的作者，前端界的“轮子哥”。

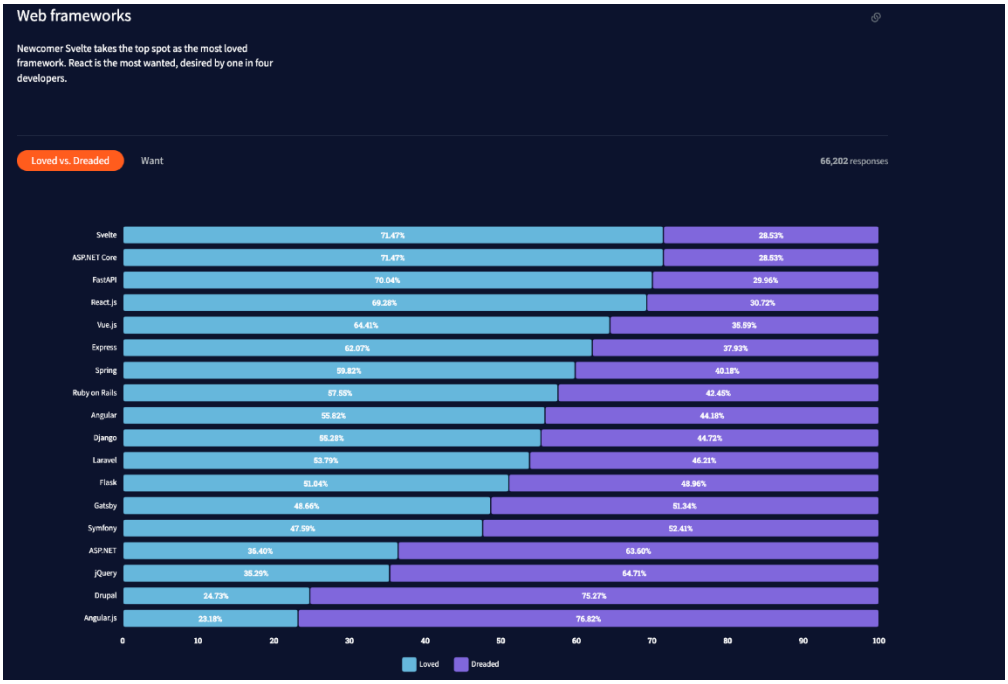
通过静态编译减少框架运行时的代码量。一个 Svelte 组件编译之后，所有需要的运行时代码都包含在里面了，除了引入这个组件本身，你不需要再额外引入一个所谓的框架运行时！

在 Github 上拥有 5w 多的 star！



在最新的 State of JS 2021 和 Stack Overflow Survey 2021 的排名情况中，也一定程度上反映了它的火热程度。





在早前知乎的如何看待 svelte 这个前端框架？问题下面，Vue 的作者尤雨溪也对其做出了极高的评价：

尤雨溪

前端开发话题下的优秀答主

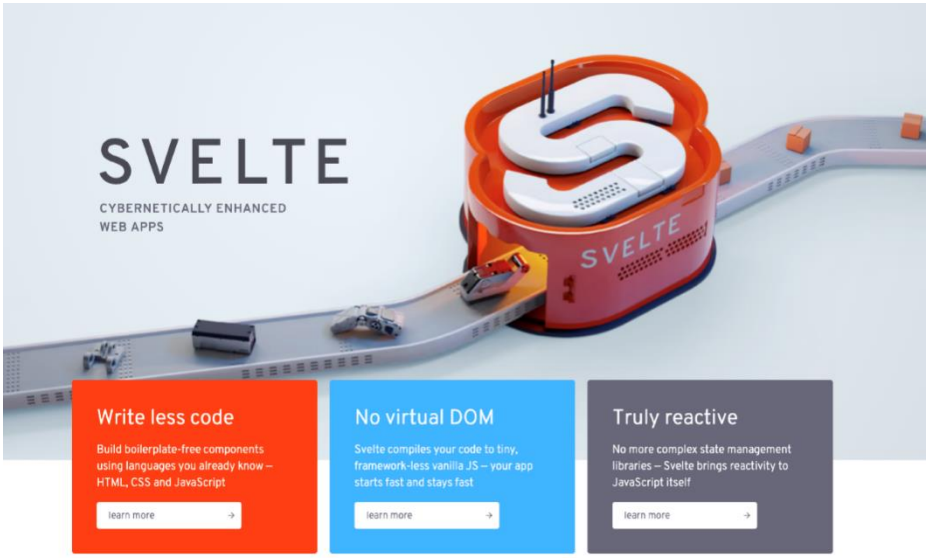
1,654 人赞同了该回答

作者是 Rich Harris，也就是 Ractive, Rollup 和 Buble 的作者，堪称前端界的轮子哥，现在又带来新轮子了！

这个框架的 API 设计是从 Ractive 那边传承过来的（自然跟 Vue 也非常像），但这不是重点。Svelte 的核心思想在于『通过静态编译减少框架运行时的代码量』。举例来说，当前的框架无论是 React Angular 还是 Vue，不管你怎么编译，使用的时候必然需要『引入』框架本身，也就是所谓的运行时 (runtime)。但是用 Svelte 就不一样，一个 Svelte 组件编译了以后，所有需要的运行时代码都包含在里面了，除了引入这个组件本身，你不需要再额外引入一个所谓的框架运行时！

当然，这不是说 Svelte 没有运行时，但是出于两个原因这个代价可以变得很小：

去它的官网看一下：



官网上清楚的表明了三大特性：

- Write less code
- No virtual DOM
- Truly reactive

1.1 Write less code

顾名思义，是指实现相同的功能，Svelte 的代码最少。这一点会在后面的示例中有所体现。

1.2 No virtual DOM

Svelte 的实现没有利用虚拟 DOM，要知道 Vue 和 React 的实现都是利用了虚拟 DOM 的，而且虚拟 DOM 不是一直都很高效的吗？

Virtual DOM 不是一直都很高效的吗？

其实 Virtual DOM 高效是一个误解。说 Virtual DOM 高效的一个理由就是它不会直接操作原生的 DOM 节点，因为这个很消耗性能。当组件状态变化时，它会通过某些 diff 算法去计算出本次数据更新真实的视图变化，然后只改变需要改变的 DOM 节点。

用过 React 的同学可能都会体会到 React 并没有想象中那么高效，框架有时候会做很多无用功，这体现在很多组件会被“无缘无故”进行重渲染（re-render）。所谓的 re-render 是你定义的 class Component 的 render 方法被重新执行，或者你的组件函数被重新执行。

组件被重渲染是因为 Vitual DOM 的高效是建立在 diff 算法上的，而要有 diff 一定要将组件重渲染才能知道组件的新状态和旧状态有没有发生改变，从而才能计算出哪些 DOM 需要被更新。

正是因为框架本身很难避免无用的渲染，React 才允许你使用一些诸如 shouldComponentUpdate，PureComponent 和 useMemo 的 API 去告诉框架哪些组件不需要被重渲染，可是这也就引入了很多模板代码。

那么如何解决 Vitual DOM 算法低效的问题呢？最有效的解决方案就是不用 Virtual DOM！

1.3 Truly reactive

第三点真正的响应式，上面也提到了前端框架要解决的首要问题就是：当数据发生改变的时候相应的 DOM 节点会被更新，这个就是 reactive。

我们先来看下 Vue 和 React 分别是如何实现响应式的。

（1）React reactive

```
const Countdown = () => {
  const { countdown, setCountdown } = useState(10)
  useEffect(() => {
    setInterval(() => {
      setCountdown(c => c-= 1)
    }, 1000)
  }, [])
  return <span>{countdown}</span>
}
```

通过 useState 定义 countdown 变量，在 useEffect 中通过 setInterval 使其每秒减一，然后在视图同步更新。这背后实现的原理是什么呢？

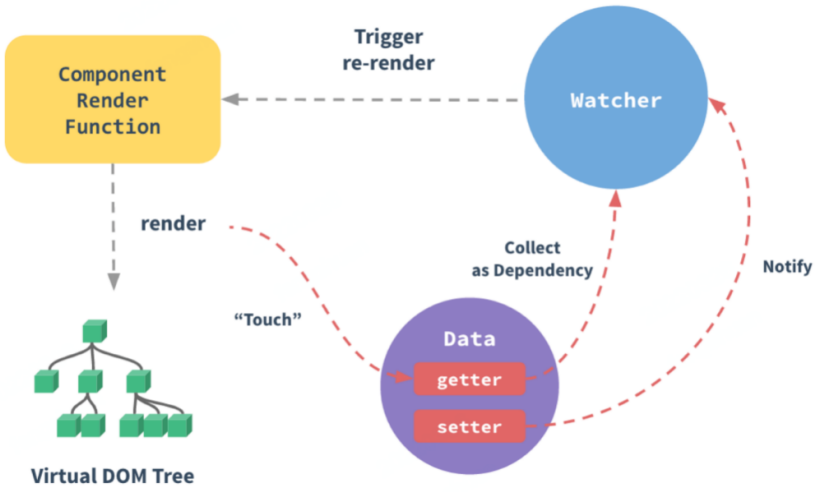
React 开发者使用 JSX 语法来编写代码，JSX 会被编译成 ReactElement，运行时生成抽象的 Virtual DOM。

然后在每次重新 render 时，React 会重新对比前后两次 Virtual DOM，如果不需要更新则不作任何处理；如果只是 HTML 属性变更，那反映到 DOM 节点上就是调用该节点的 setAttribute 方法；如果是 DOM 类型变更、key 变了或者是在新的 Virtual DOM 中找不到，则会执行相应的删除/新增 DOM 操作。

（2）Vue reactive

```vue
<script setup>
    import { ref, onMounted } from 'vue'
    const countdown = ref(10)
    onMounted(() => {
      setInterval(() => countdown.value -= 1, 1000)
    })
</script>
<template>
    <span>{{countdown}}</span>
</template>
```

用 Vue 实现同样的功能。Vue 背后又是如何实现响应式的呢？



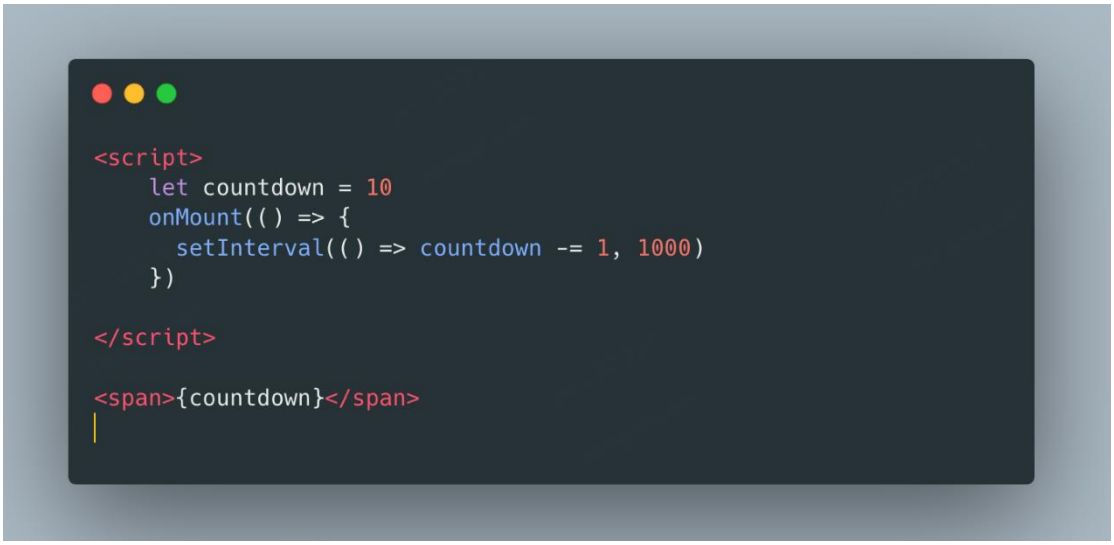
大致过程是编译过程中收集依赖，基于 Proxy（3.x），defineProperty(2.x) 的 getter，setter 实现在数据变更时通知 Watcher。

像 Vue 和 React 这种实现响应式的方式会带来什么问题呢？

- diff 机制为 runtime 带来负担
- 开发者需自行优化性能
- useMemo
- useCallback
- React.memo
- ...

那么 Svelte 又是如何实现响应式的呢？

（3）Svelte reactive

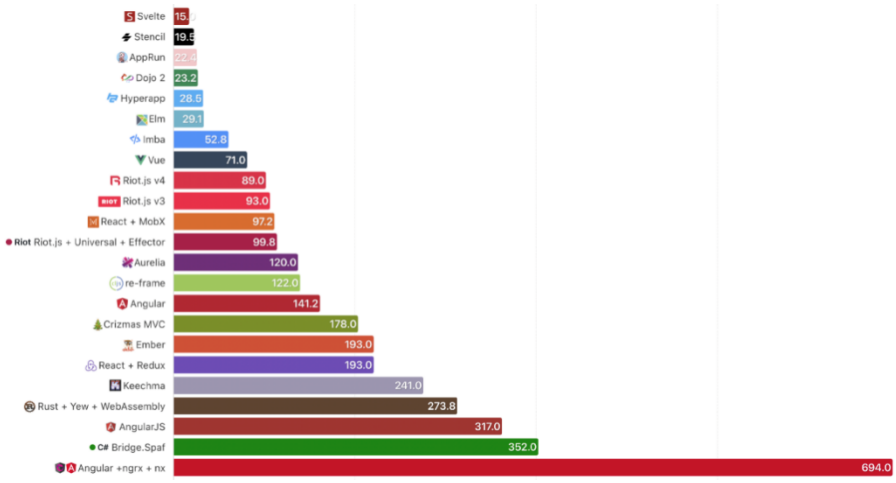


其实作为一个框架要解决的问题是当数据发生改变的时候相应的 DOM 节点会被更新（reactive），Virtual DOM 需要比较新老组件的状态才能达到这个目的，而更加高效的办法其实是数据变化的时候直接更新对应的 DOM 节点。

这就是 Svelte 采用的办法。Svelte 会在代码编译的时候将每一个状态的改变转换为对应 DOM 节点的操作，从而在组件状态变化的时候快速高效地对 DOM 节点进行更新。

深入了解后，发现它是采用了 Compiler-as-framework 的理念，将框架的概念放在编译时而不是运行时。你编写的应用代码在用诸如 Webpack 或 Rollup 等工具打包的时候会被直接转换为 JavaScript 对 DOM 节点的原生操作，从而让 bundle.js 不包含框架的 runtime。

那么 Svelte 到底可以将 bundle size 减少多少呢？以下是 RealWorld 这个项目的统计：

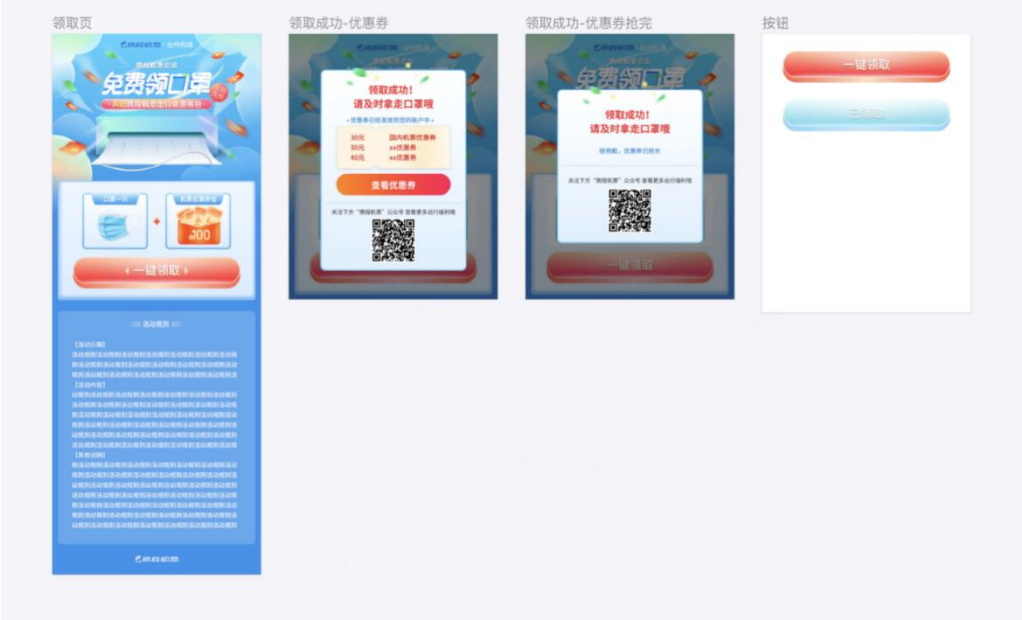


由上面的图表可以看出实现相同功能的应用，Svelte 的 bundle size 大小是 Vue 的 1/4，是 React 的 1/20！单纯从这个数据来看，Svelte 这个框架对 bundle size 的优化真的很大。

看到这么强有力的数据支撑，不得不说真的很动心了！

二、项目落地

为了验证 Svelte 在营销 h5 落地的可能，我们选择了口罩机项目：

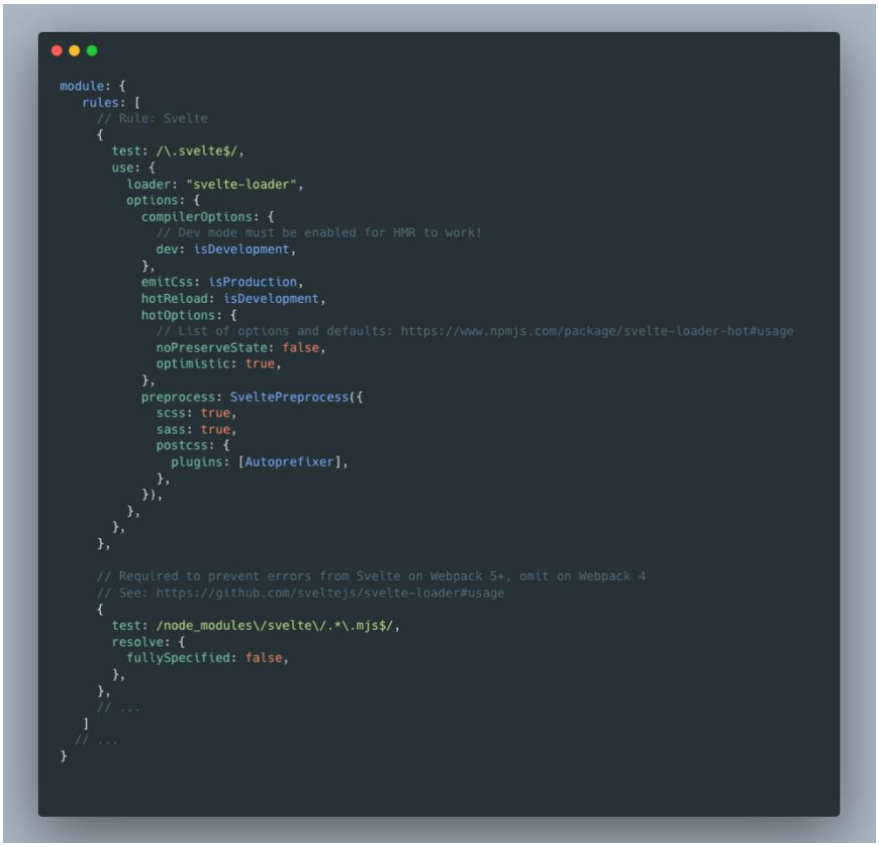


上图是口罩机项目的设计稿，不难看出，核心逻辑不是很复杂，这也是我们选用它作为 Svelte 尝试的原因。

首先项目的基础结构是基于 svelte-webpack-starter 创建的，集成了 TypeScript、SCSS、Babel 以及 Webpack5。但这个基础模板都只进行了简单的支持，像项目中用到的一些图片、字体等需要单独使用 loader 去处理。

启动项目，熟悉的 hello world：

这里看下核心的 webpack 配置：



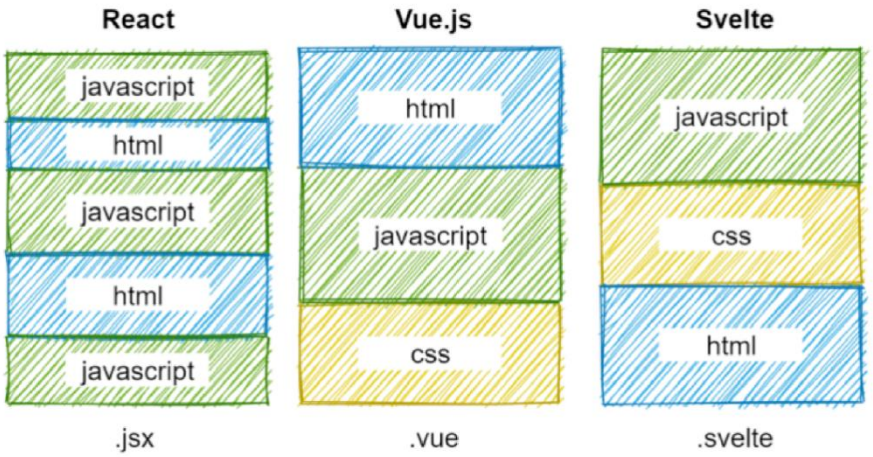
```
module: {
  rules: [
    // Rule: Svelte
    {
      test: /\.svelte$/,
      use: {
        loader: "svelte-loader",
        options: {
          compilerOptions: {
            // Dev mode must be enabled for HMR to work!
            dev: isDevelopment,
          },
          emitCss: isProduction,
          hotReload: isDevelopment,
          hotOptions: {
            // List of options and defaults: https://www.npmjs.com/package/svelte-loader-hot#usage
            noPreserveState: false,
            optimistic: true,
          },
          preprocess: SveltePreprocess({
            scss: true,
            sass: true,
            postcss: {
              plugins: [Autoprefixer],
            },
          }),
        },
      },
    },

    // Required to prevent errors from Svelte on Webpack 5+, omit on Webpack 4
    // See: https://github.com/sveltejs/svelte-loader#usage
    {
      test: /node_modules\/svelte\/.*\.mjs$/,
      resolve: {
        fullySpecified: false,
      },
    },
    // ...
  ]
  // ...
}
```

当然开发环境使用 webpack 有时不得不说体验不太好，每次都要好几秒，我们就用 Vite 来替代了，基本都是秒开，Vite 的配置也比较简单：

```
import { defineConfig } from "vite";
import { svelte } from "@sveltejs/vite-plugin-svelte";
import sveltePreprocess from "svelte-preprocess";

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [
    svelte({
      preprocess: sveltePreprocess(),
    }),
  ],
  resolve: {
    extensions: [".mjs", ".js", ".ts", ".jsx", ".tsx", ".json",
".svelte"],
  },
  server: {
    proxy: {
      "/api": {
        target: "http://                          ",
        secure: false,
        changeOrigin: true,
      },
    },
  },
});
```

2.1 组件结构差异

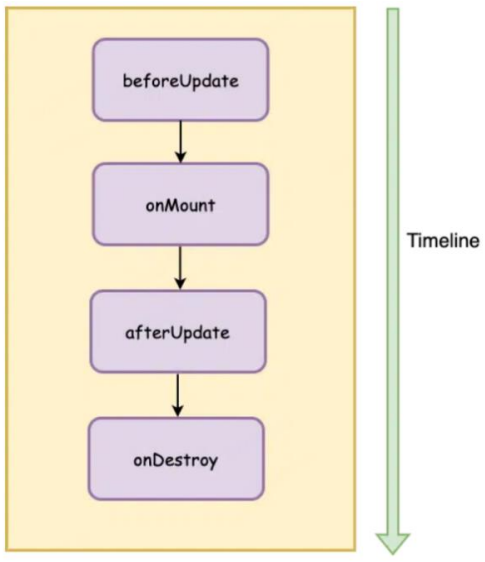
和 React 组件不同的是，Svelte 的代码更像是以前我们在写 HTML、CSS 和 JavaScript 时一样（这点和 Vue 很像）。



所有的 JavaScript 代码都位于 Svelte 文件顶部的 <script></script> 标签当中。然后是 HTML 代码，你还可以在 <style></style> 标签中编写样式代码。组件中的样式代码只对当前组件有效。这意味着在组件中为 标签编写的样式不会影响到其他组件中的 元素。

2.2 生命周期

Svelte 组件的生命周期有不少，主要用到的还是 onMount、 onDestoy、beforeUpdate、afterUpdate，onMount 的设计和 useEffect 的设计差不多，如果返回一个函数，返回的函数将会在组件销毁后执行，和 onDestoy 一样：



2.3 初始状态

接下来是对初始状态的定义：

我们发现代码在对变量更新的时候并没有使用类似 React 的 setState 方法，而是直接对变量进行了赋值操作。仅仅是对变量进行了赋值就可以引发视图的变化，很显然是数据响应的，这也正是 Svelte 的 truly reactive 的体现。

2.4 条件判断

项目中使用了很多的条件判断，React 由于使用了 JSX，所以可以直接使用 JS 中的条件控制语句，而模板是需要单独设计条件控制语法的。比如 Vue 中使用了 v-if。

Svelte 中则是采用了{#if conditions}、{:else if}、{/if}，属于 Svelte 对于 HTML 的增强。

上面代码中有这么一行：

$: buttonText = isTextShown ? 'Show less' : 'Show more'

buttonText 依赖了变量 isTextShown，依赖项变更时触发运算，类似 Vue 中的 computed，这里的 Svelte 使用了$:关键字来声明 computed 变量。

这又是什么黑科技呢？这里使用的是 Statements and declarations 语法，冒号:前可以是任意合法变量字符。

2.5 数据双向绑定

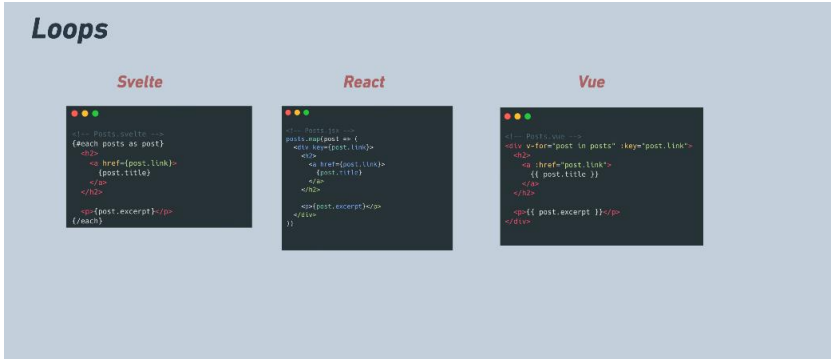
项目中有很多地方需要实现双向绑定。我们知道 React 是单向数据流，所以要手动去触发变量更新。而 Svelte 和 Vue 都是双向数据流。

Svelte 通过 bind 关键字来完成类似 v-model 的双向绑定。

2.6 列表循环

项目中同样使用了很多列表循环渲染。Svelte 使用 {#each items as item}{/each} 来实现列表循环渲染，这里的 item 可以通过解构赋值，拿到 item 里面的值。

不得不说有点像 ejs



2.7 父子属性传递

父子属性传递时，不同于 React 中的 props，Svelte 使用 export 关键字将变量声明标记为属性，export 并不是传统 ES6 的那个导出，而是一种语法糖写法。

注意只有 export let 才是声明属性

```
<script lang="ts">

export let status = 2;

$: ({
  status,
} = $$props);

</script>

<div class="coupon-result">
  <div class="tip">
    {#if status === 2}

    {/if}
   </div>
  {#if status === 3}

  {/if}
</div>
```

2.8 跨组件通讯（状态管理）

既然提到了父子组件通讯，那就不得不提跨组件通讯，或者是状态管理。这也一直是前端框架中比较关注的部分，Svelte 框架中自己实现了 store，无需安装单独的状态管理库。你可以定义一个 writable store，然后在不同的组件之间进行读取和更新：

```
// store.js
import { wrtiable } from 'svelte/store'
export const count = writable(0)

// App.svelte
<script>
import { count } from './store.js'

let count_value;

const unsubscribe = count.subscribe(value => {
  count_value = value;
});
</script>
<h1>The count is {count_value}</h1>

// Incrementer.svelte
<script>
    import { count } from './stores.js';

    function increment() {
        count.update(n => n + 1);
    }
</script>

<button on:click={increment}>
    +
</button>

// Resetter.svelte
<script>
    import { count } from './stores.js';

    function reset() {
        count.set(0);
    }
</script>

<button on:click={reset}>
    reset
</button>
```

每个 writable store 其实是一个 object，在需要用到这个值的组件里可以 subscribe 他的变化，然后更新到自己组件里的状态。在另一个组件里可以调用 set 和 update 更新这个状态的值。

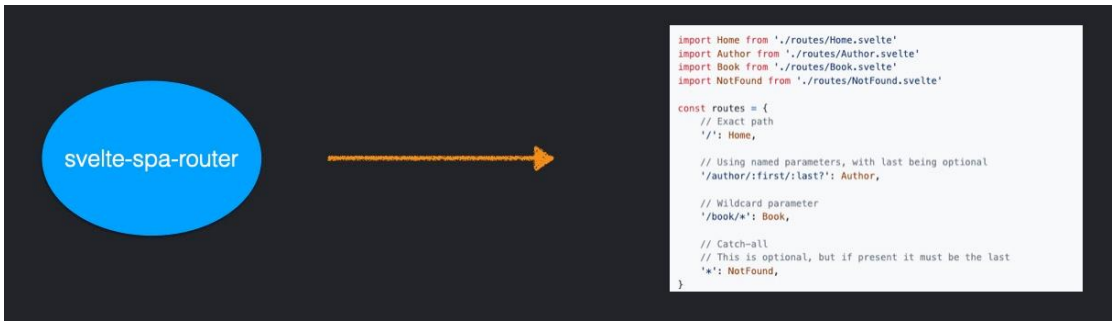
2.9 路由

Svelte 目前没有提供官方路由组件，不过可以在社区中找到：

- svelte-routing
- svelte-spa-router

svelte-routing 和 react-router-dom 的使用方式很像：

而 svelte-spa-router 更像 vue-router 一点：



2.10 UI

项目中也用到了组件库，通常 react 项目一般都会采用 NFES UI，但毕竟是 react component，在 Svelte 中并不适用。我们尝试在社区中寻找合适的 Svelte UI 库，查看了 Svelte Material UI、Carbon Components Svelte 等，但都不能完全满足我们的需求，只能自己去重写了（只用到了几个组件，重写成本不算很大）。

2.11 单元测试

单元测试用的是@testing-library/svelte：

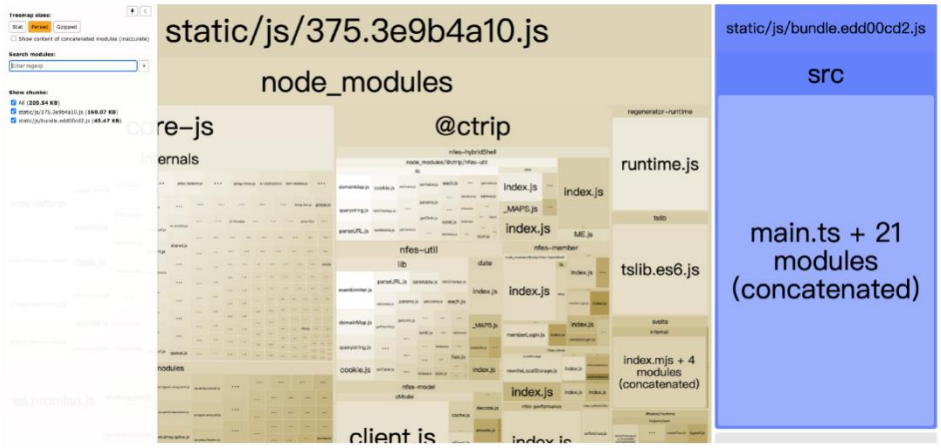
```
<script>
  export let name

  let buttonText = 'Button'

  function handleClick() {
    buttonText = 'Button Clicked'
  }
</script>

<h1>Hello {name}!</h1>

<button on:click="{handleClick}">{buttonText}</button>
```

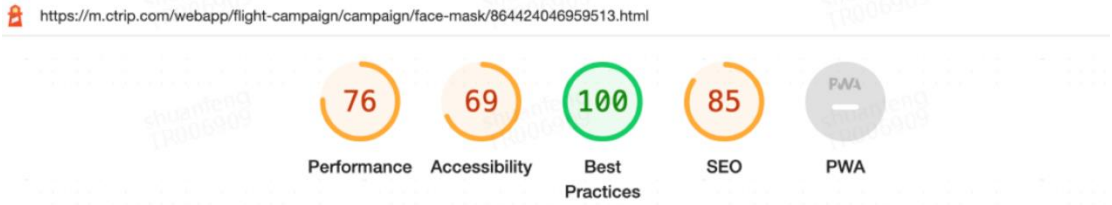
```
// NOTE: jest-dom adds handy assertions to Jest and it is recommended, but not required.
import '@testing-library/jest-dom'

import {render, fireEvent} from '@testing-library/svelte'

import Comp from '../Comp'

test('shows proper heading when rendered', () => {
  const {getByText} = render(Comp, {name: 'World'})

  expect(getByText('Hello World!')).toBeInTheDocument()
})

// Note: This is as an async test as we are using `fireEvent`
test('changes button text on click', async () => {
  const {getByText} = render(Comp, {name: 'World'})
  const button = getByText('Button')

  // Using await when firing events is unique to the svelte testing library because
  // we have to wait for the next `tick` so that Svelte flushes all pending state changes.
  await fireEvent.click(button)

  expect(button).toHaveTextContent('Button Clicked')
})
```

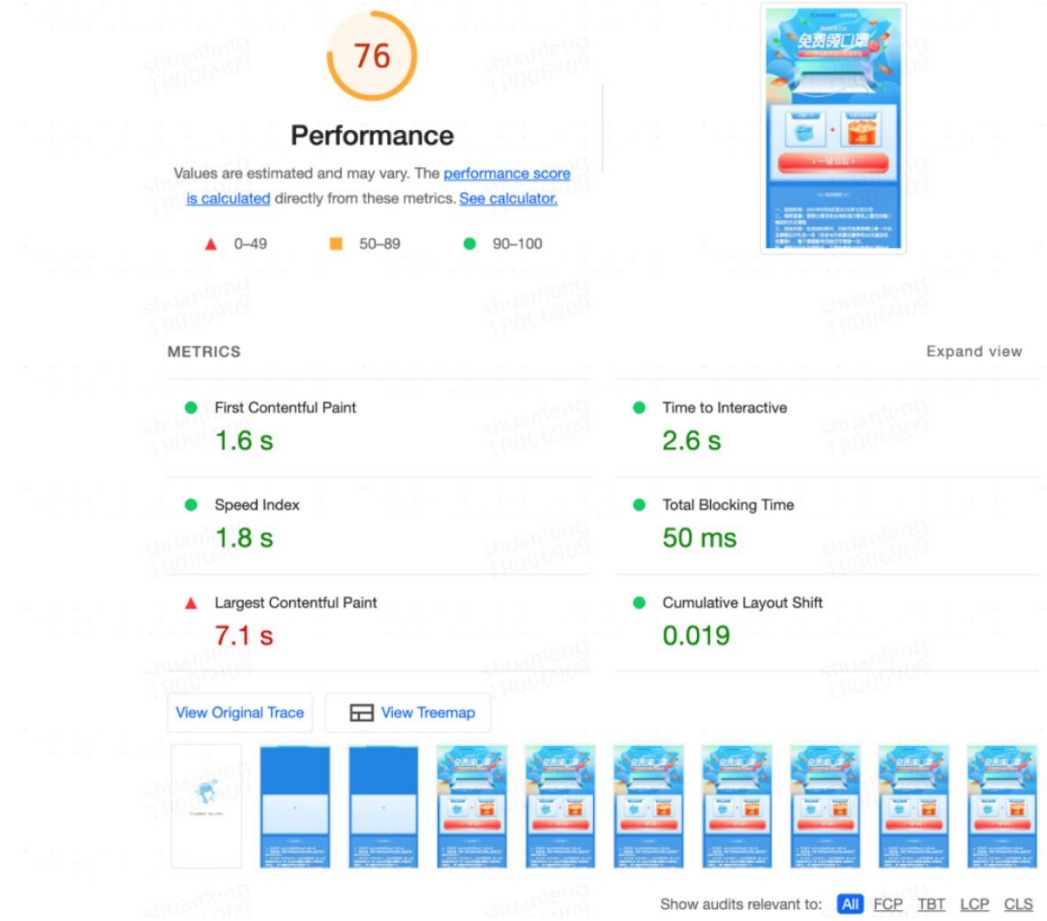
基本用法和 React 是很类似的。

业务代码迁移完毕，接着就是对原有功能 case 的逐一验证。

为了验证单单使用 Svelte 进行开发的效果，我们没有进行其他的优化，发布了一版只包含 Svelte 的代码到产线，来看下 bundle size（未做 gzip 前）和 lighthouse 评分情况：

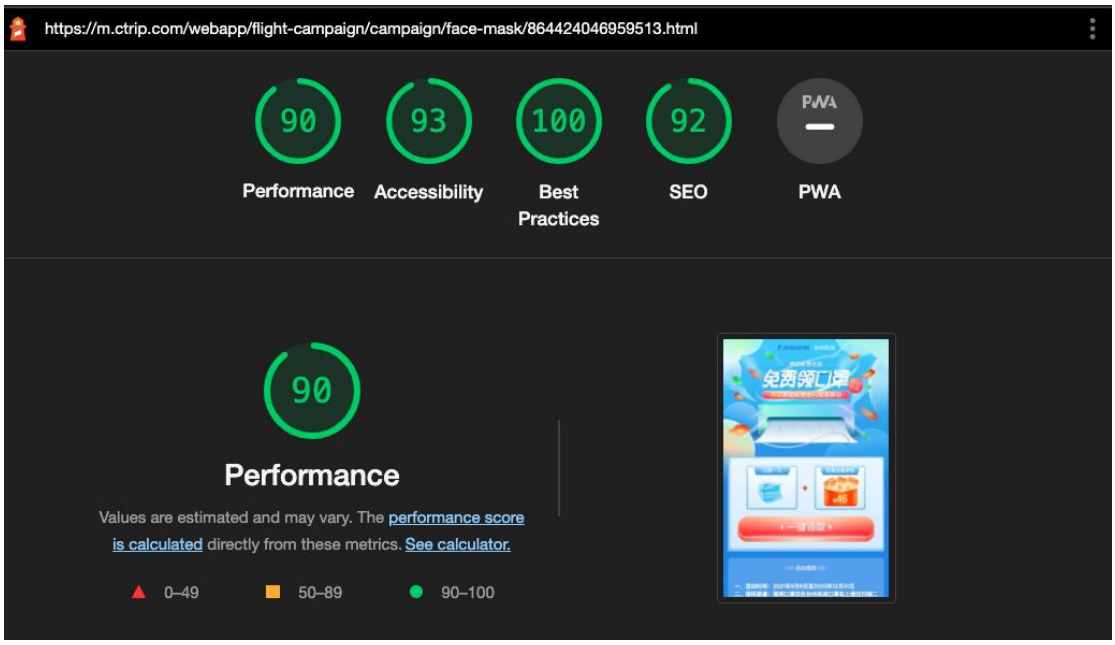






除此之外，我们遵循 lighthouse 给出的改进建议，对 Performance、Accessibility 和 SEO 做

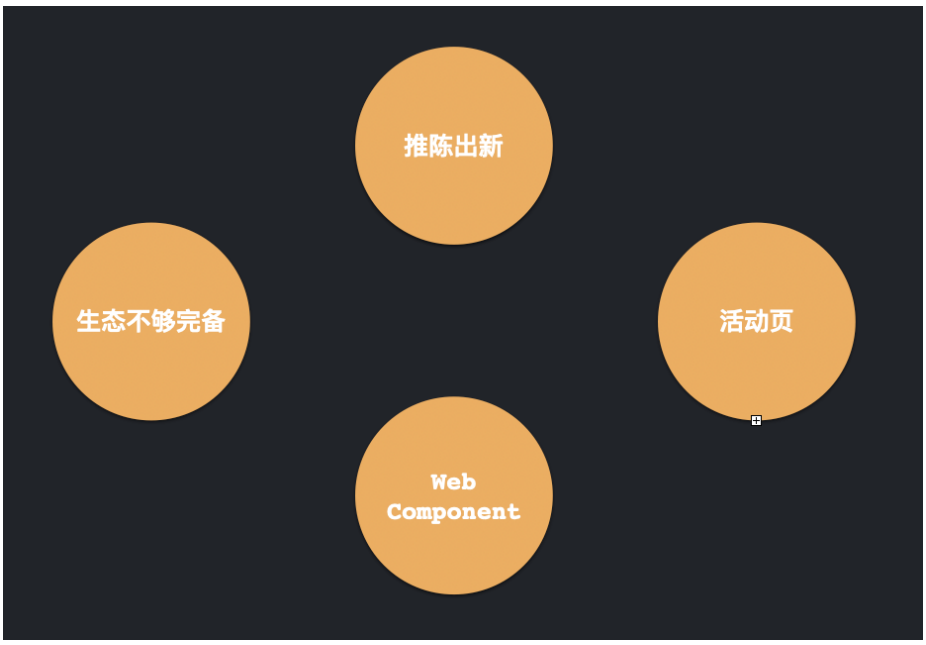
了更进一步的优化改进：



Performance 的提升主要得益于图片格式支持 webp 以及一些资源的延迟加载，Accessibility 和 SEO 的提升主要是对 meta 标签的调整。

三、实践总结

通过这次技改，我们对 Svelte 有了一些全新的认知。



整体来说，Svelte 继前端三大框架之后推陈出新，以一种新的思路实现了响应式。

因其起步时间不算很长，国内使用程度仍然偏少，目前来说其生态还不够完备。

但这不能掩盖其优势：足够“轻”。Svelte 非常适合用来做活动页，因为活动页一般没有很复杂的交互，以渲染和事件绑定为主。正如文章最开始说的，一个简单的活动页却要用 React 那么重的框架多少有点委屈自己。所以对于一些营销团队，想在 bundle size 上有较大的突破的话，Svelte 是绝对可以作为你的备选方案的。

另外现在社区对于 Svelte 还有一个很好的用法是使用它去做 Web Component，好处也很明显：

- 使用框架开发，更容易维护
- 无框架依赖，可实现跨框架使用
- 体积小

所以对于想实现跨框架组件复用的团队，用 Svelte 去做 Web Component 也是一个很好的选择。

四、参考链接

- svelte、react、vue 对比：https://joshcollinsworth.com/blog/introducing-svelte-comparing-with-react-vue
- 你还没有听过 svelte 吗？https://developpaper.com/you-havent-heard-of-sveltejs-yet/
- 如何看待 svelte 这个前端框架？https://www.zhihu.com/question/53150351

携程小程序生态之 Taro 跨端解决方案

【作者简介】携程前端框架团队，为携程集团各业务线提供优秀的 Web 解决方案，当前主要专注：新一代研发模式探索，Rust 构建工具链路升级、Serverless 应用框架开发、在线文档系统开发、低代码平台搭建、适老化与无障碍探索等。

一、摘要

随着携程接入小程序平台类型的增加，前端需要负责的端越来越多，研发成本也随之成倍增加。为了解决一套代码多端运行的诉求，携程小程序框架不断调整、升级，逐渐形成了携程 Taro 跨端解决方案。

二、背景

2.1 小程序现状

近几年业界推出了各种小程序平台，每个小程序平台都会提供一个专属的原生小程序 DSL，这些 DSL 之间或多或少存在一些差异，这意味着使用某一类型小程序 DSL 编写的代码，无法直接复用到其他小程序平台上，造成开发和维护成本成倍增加。

2.2 业务现状

目前携程支持的小程序业务涉及多个小程序平台，如果全部只使用平台自身的 DSL 开发，开发人员至少需要同时开发及维护 5 个活跃版本，开发任务繁重，代码维护困难。此外，每新增一类小程序入口，开发人员必须将原有业务逻辑重写一遍，不仅工作内容重复，而且严重影响业务落地速度。为此，一套代码多端运行的诉求迫在眉睫。

值得注意的是，携程已经接入了多个平台的小程序，使用多端统一开发框架从零开始开发小程序代码既浪费研发资源又不现实。我们需要考虑如何在携程当前已有的小程序代码的基础上使用跨端框架开发新业务、逐步切换原有代码，实现多端统一开发方案的平滑接入。

2.3 解决方案

为了解决上述问题，我们研发了携程 Taro 跨端解决方案，开发者只需使用 Taro 框架书写一套代码，便可获得在多个平台皆运行良好的小程序项目。此外，该方案还提供了仓库管理、辅助脚手架、编译功能扩展及过程优化等功能。

三、Taro 跨端解决方案设计

3.1 技术选型

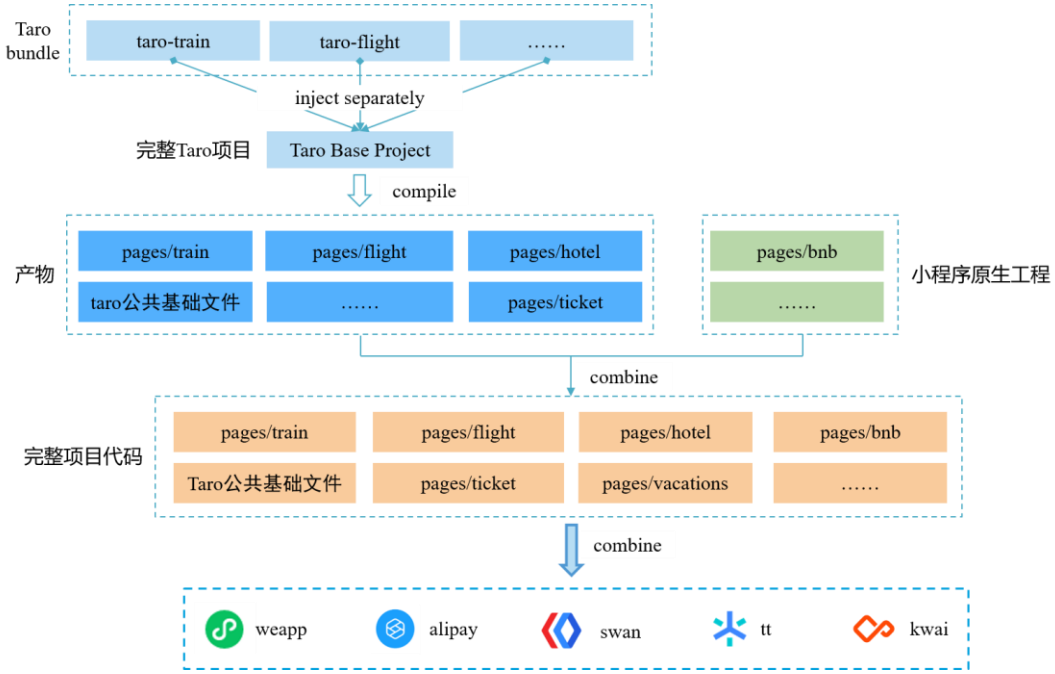
为了在保留原有代码的前提下实现一套代码多端运行，我们对市面上的跨端开发框架进行了调研，最终决定使用 Taro 3 作为携程小程序的跨端框架。主要是考虑到 Taro 3 具有以下 4

项优点：

- 框架稳定性高
- 支持的平台种类多
- 支持使用 React 语法规范进行开发
- 支持 Taro 和原生混合开发

3.2 整体架构设计

携程小程序随着业务的发展、多平台化趋势和跨端技术的不断演进，逐渐形成了一套多平台复用的 Taro 跨端解决方案。



携程小程序的项目工程架构图如图 1 所示。上半部分是跨端复用层，这一层的项目代码是基于 Taro 框架进行开发的，多个 Taro 模块可以灵活组合成一个完整的 Taro 项目；从下半张图可以看出，Taro 项目是完整小程序项目的其中一个模块，Taro 项目的运行需要依赖小程序原生壳工程。整个 Taro 项目是依据插件化的设计思想组织代码的，由多个独立的 Taro 模块和一个 Taro 基础壳工程构成。

3.3 Taro 模块的插件化设计

首先，携程小程序是由多个团队协同开发的项目，跨团队协作开发时常常会出现代码冲突、文件覆盖等问题。因此，需要思考如何通过合理的项目架构从根本上解决这些问题，保证多团队并行开发的效率。考虑到可以采用模块化的概念，根据业务线类型将项目代码拆解成多个子模块，并约定文件放置以及引用规则，从而确保各个子模块的源码文件能够完全隔离。

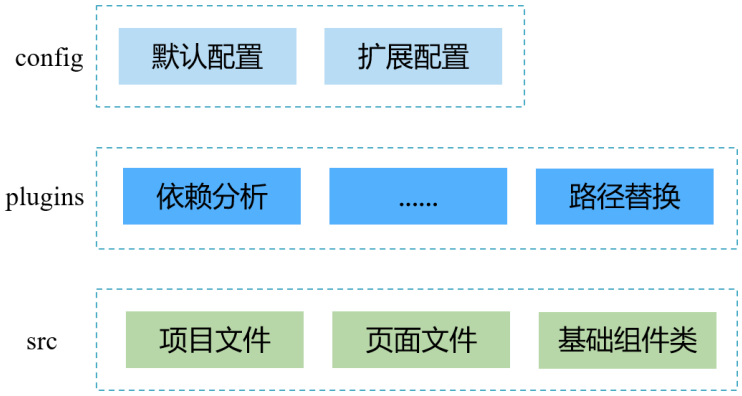
然而，使用模块化开发方案，得到的 Taro 项目几乎不具有扩展性，Taro 模块也无法快速便捷的被复用。怎样才能提高 Taro 模块的灵活性，使得任取一个或多个 Taro 模块进行组合都

能得到一个完整的 Taro 项目，且合并到小程序原生壳工程中能够正常运行？为解决以上问题，我们进一步将插件化的设计思想应用在 Taro 跨端解决方案上，提供了定制化的 Taro 基础壳工程以及一套开发规范。

开发者需使用 Taro 基础壳工程作为开发模版，并遵循规范进行业务开发，所有 Taro 模块在本地开发或发布时按照统一的标准进行编译、融合，从而达到在不破坏原有项目结构的前提下灵活组合使用的目标。下面我们将从项目构成与开发规范、仓库管理、开发架构，运行方案等方面详细讲解 Taro 跨端解决方案。

3.4 项目结构与开发规范

Taro 基础壳工程是由 Taro 官方的模版项目拓展而来，内部增加了定制化的编译配置、Plugins 和基础组件类。如图 2 所示，Taro 基础壳工程内仅包含与公共基础功能相关的文件，这些文件可抽象成 3 类内容：编译配置文件、用于扩展编译过程的 Plugins，以及页面基类。



开发 Taro 模块时，开发者需要关注 3 块内容（扩展配置、项目文件、页面文件，参见图 2），并遵守以下几项规范进行开发：

（1）主包的大小直接影响着小程序启动性能，为此我们提出“非必要不打入主包”原则：除小程序启动时需要用到的文件、tabBar 页面及公共基础文件外，其他文件应全部拆入分包中。Taro 模块也须遵循该原则，开发者应将业务代码全部放置在自己的分包目录下，项目文件 app.config.js 中只增加分包页面配置。

（2）为了避免合并项目时出现业务线之间文件相互覆盖或页面路径冲突，统一约定分包页面路径的前缀为“pages/业务线英文缩写”，结合“非必要不打入主包”原则使用，可以有效隔离各业务线的源码文件。

（3）为确保 Taro 模块的业务相关内容（包括依赖包）完全放置在分包路径下，不占用主包的大小，我们提供了 commonModule 方案：在引用第三方依赖包前，需要开发者本地进行预编译操作，将需要引用的内容打包成放置在分包中的一个或多个 commonModule 文件，随后从预编译产物（commonModule）中引用所需的模块。除此之外，还可以通过 commonOrigin 方案获取依赖包的源码，此时会将所需依赖包的原样复制到开发者指定的文件夹目录下。

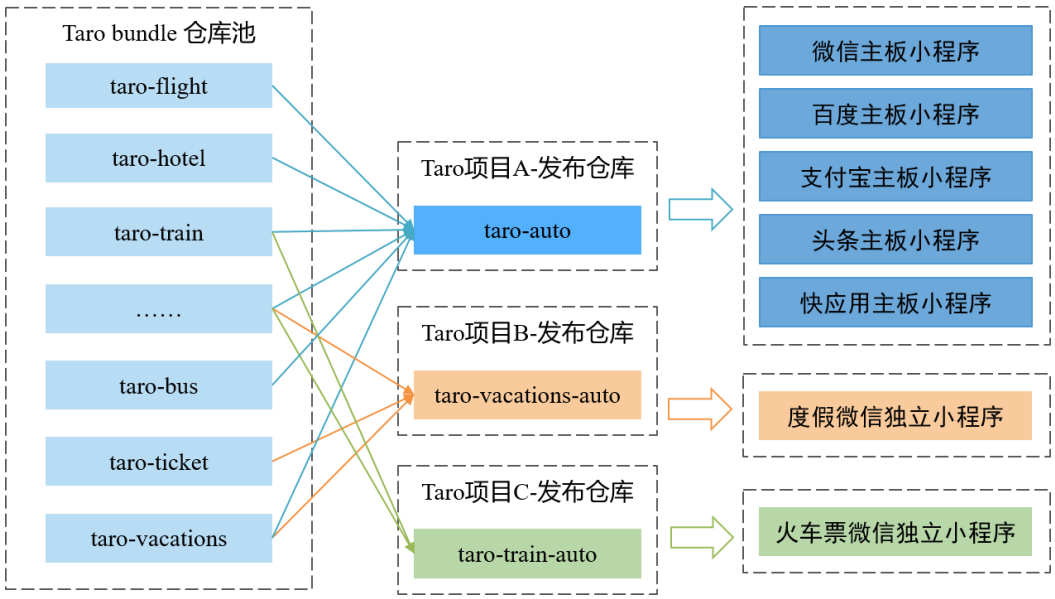
3.5 仓库管理

首先，Taro 项目采用分仓开发的模式，将每个业务线的 Taro 模块存放在一个单独的 git 仓库中。将 Taro 模块分别存放在不同的仓库，可以保持各个业务仓库提交代码操作的独立。

其次，我们借助 gitsubmodule 工具将各个 Taro 模块所在的仓库以及 Taro 基础壳工程仓库作为子目录包含到整个 Taro 项目的发布仓库中，为发布仓库和多业务仓建立起父子仓库的关联。建立仓库间的关联后，Taro 项目可以借助 git submodule 的获取子模块功能快速克隆自己所需的 Taro 模块源码，并且可以随时拉取各个业务仓库的最新代码。

再次，由于 gitsubmodule 允许一个仓库作为多个仓库的子目录，这意味着可以选取不同的 Taro 模块，将他们的仓库组合成新的发布仓库，结合携程小程序管理平台中各个小程序所需 Taro 模块的配置一起使用，可实现根据配置动态引入 Taro 模块的效果。

随后，通过对多个 Taro 模块进行组合，可以快速获得各种包含多个业务线的 Taro 项目，从而提高 Taro 模块在不同场景中的复用。



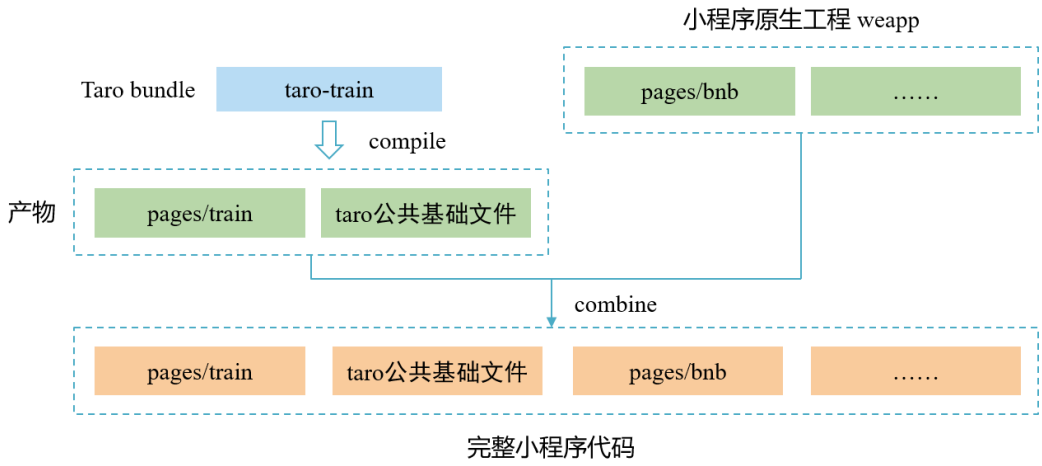
然后，将 Taro 项目作为完整小程序的一个 bundle，将 Taro 项目的编译产物与小程序原生壳项目进行合并，即可获取到 Taro 混合开发的完整小程序代码。

如图 3 所示，通过组合 Taro 模块可以获取到包含不同功能的 Taro 项目，接着将 Taro 项目与不同类型的小程序原生壳项目结合，便可以轻松获取多个 Taro 混合开发的小程序项目。

3.6 开发及运行架构

开发者只需安装 miniTools 脚手架并执行初始化命令行，即可快速获取 Taro 模块的开发模版和小程序原生壳工程，完成项目初始化。开发 Taro 模块时，开发者需要遵循开发规范，在分包目录下添加文件并编写业务代码。编写过程中，只需执行编译指令，便可将本地开发

的源码编译并融合到小程序原生壳工程中，得到包含 Taro 模块内容的完整小程序代码了。



结合上述本地开发过程，Taro 跨端解决方案具体提供的功能以及优化工作说明如下：

（1）Taro 模块直接引用小程序原生壳项目内的模块。提供@/miniapp 标识符，用于指代小程序完整项目根目录。同时，编译过程中会识别代码中的标识符，动态计算并修改引用路径。开发 Taro 模块时只需使用@/miniapp 拼接文件的相对路径，便可以直接引用小程序完整项目根目录内的文件。

（2）扩展页面配置项，提供设置自定义组件嵌套层级的功能。开发者可以在页面配置文件中增加自定义组件的嵌套层级配置，编译时将检索页面配置文件的内容，汇总并设置 Taro 项目用到的自定义组件的嵌套层级。

（3）根据分包路径，动态生成 splitChunks。为了防止公共文件被打到主包中（占用主包的大小），编译时会读取 Taro 模块配置的分包路径，动态生成 splitChunks。该方案可以将分包用到的公共文件单独抽离到分包中，随后为分包中的所有页面添加对分包公共文件的引用即可。

（4）提供扩展配置文件，允许自行添加 alias 和 externals 配置，便于开发者自定义目录别名以及不需要打包的依赖。

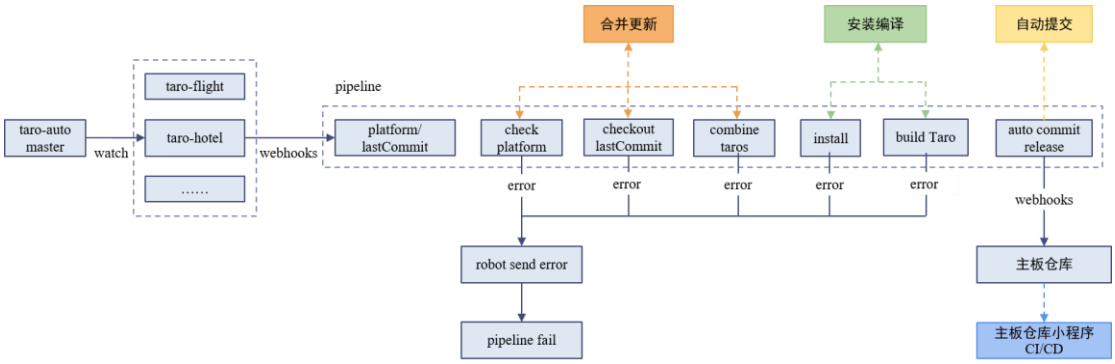
（5）提供模块分析功能，开发者可以更加便捷地查看每个 chunk 内包含哪些文件

（6）使用混合模式进行打包，随后自动将编译产物移动到小程序原生壳工程中，同时将分包配置添加到小程序项目配置内。这一步是为了将 Taro 项目编译产物与携程原有的小程序代码合并成一个完整的 Taro 混合小程序项目。在开发规范的作用下，Taro 模块的分包路径是根据各业务线隔离的，每个 Taro 模块的文件都严格放置在自己的分包路径内，因此只需将公共基础文件放置到项目根目录，分包内容迁移至各自的分包目录下，便可顺利完成代码合并。

3.7 项目发布

我们利用 webhooks 实现 Taro 项目的流水线式迭代开发，当 Taro 模块提交修改时，会触发 Taro 项目发布仓库的 pipeline。Taro 项目发布仓库 pipeline 的主要工作流程如下：

首先，Taro 项目会拉取所有子仓库的最新代码。接着，将从小程序管理平台获取当前 Taro 项目使用的 Taro 模块列表及相应的发布版本号，并据此按需将各个 Taro 模块的发布代码拉取到 docker 中。至此，Taro 项目所需发布的所有源码已经获取完毕。



接下来，将进行 Taro 项目的合并工作：将各个 Taro 模块的代码切换至指定版本，获取各个 Taro 模块中配置的分包路径，并将相关配置文件和分包目录下业务代码合并到 Taro 壳工程中。

Taro 项目合并完成后，便会编译成指定小程序类型的代码。值得一提的是，Taro 基础壳工程既是 Taro 项目壳又是开发模版，它提供了统一的 Taro 项目结构和编译方式，也是 Taro 模块能灵活组合的原因所在。

最后，将 Taro 项目的编译产物与相应类型的小程序原生代码进行合并，即可获得完整的 Taro 小程序项目。将项目代码上传到小程序后台，则标志着一整套项目发布流程的顺利完成。

四、总结

目前，Taro 跨端解决方案已支持一套代码运行在 5 类小程序（微信、支付宝、字节跳动、百度、快手）平台。使用此方案进行开发的 Taro 小程序项目灵活度和复用性很高，可以按需选用 Taro 模块组合成一个完整的 Taro 项目。

此外，我们还提供了配套的脚手架工具、仓库管理、版本管理以及 pipeline 自动化方案，极大提升了小程序的开发、测试和发布效率。今后我们将继续完善小程序生态系统，为解决业务痛点不断孵化出更多的解决方案。

用 DDD（领域驱动设计）和 ADT（代数数据类型）提升代码质量

【作者简介】工业聚，携程高级前端开发专家，react-lite, react-imvc, farrow 等开源项目作者。

很多开发者都有一个迷思，认为项目里的代码质量和可维护性的持续下降，主要根源在于时间紧迫、需求变动频繁。如果产品需求更加明确，并给予足够的开发时间，开发团队可以长期保证代码质量和可维护性。

今天介绍的 DDD（领域驱动设计）和 ADT（代数数据类型）模型，给出了另一部分的答案：代码质量持续下降，开发团队也要负主要责任。

如果没有采用更合理的开发模型，项目的代码质量将随着时间和复杂度的增加而急剧下降。再明确的产品需求，再多的开发时间，也很难阻止代码库的腐坏。

由于 DDD 和 ADT 都是很大的主题，很难在一篇技术文章中充分展开。因此，本文主要简要介绍它们之间的关系以及背后的思想，希望能够带给大家一些启发。感兴趣的同学，后续可以查阅更深入的材料。

主要内容分成以下几个部分：

- 代码质量的评估方式
- DDD 的概念与定义
- ADT 的概念与定义
- 案例：用 DDD + ADT 做数据建模
- 总结

一、代码质量的评估方式

'If you can’t measure it, you
can’t improve it.'

不管做哪种提升和优化，最关键的起始步骤总是寻找衡量的标准和方法。提升代码质量也不例外。

1.1 代码质量的评估标准

- 可拓展性(Extensibility)
- 可维护性(Maintainability)
- 可读性(Readability)

- 可测试性(Testability)
- etc.

上述指标是在一个相对宏观的层面评估代码，偏向定性指标。

给定两段代码，开发可以判断出哪一个可拓展性更好，但具体好多少，却不容易有一致的判断。特别是可读性这个指标，非常依赖开发者的经验和主观偏好。

有一些定量指标，比如一些 Lint 规则、循环引用检测等，对提升代码质量也有帮助。但这类定量指标的不足之处在于，它们在一个非常微观的层面做代码评估，并不关心代码要解决的问题是什么。

这类定量指标，容易失效甚至带来反效果。对处理具体问题的帮助也有限，主要服务于统一代码风格、约束写法、规避常见反模式。

我们希望有一种评估标准，既能在宏观定性层面帮助我们考虑问题，又能在微观定量层面帮助我们解决问题。

1.2 几种代码质量的提升方法

- SOLID Principles
- Clean Code Principles
- Design Patterns
- Low Coupling, High Cohesion
- etc.

有很多编程原则可以提升代码质量，其中最著名的两个或许是 SOLID 和设计模式 (Design Patterns)，它们已经出现至少二三十年，在今天依然被开发者们津津乐道。

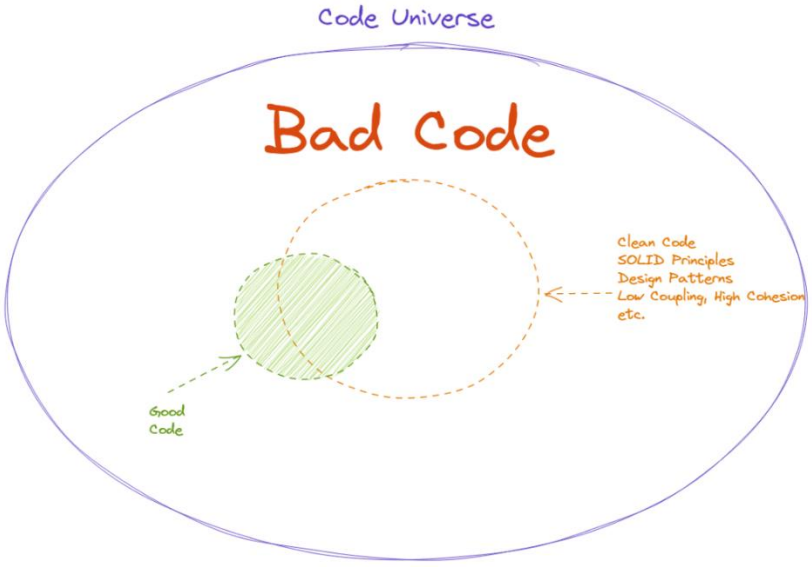
单一职责、开闭原则、接口分离原则、观察者模式、订阅模式、解释器模式、装饰器模式、低耦合、高内聚……这些词汇在程序员群体里耳熟能详，历久弥新。

不过：

- 有些开发者认为编程语言已经发展出了很多新特性，很多问题已经变了，原始的设计模式已经不适用于现在的情况。
- 很多开发者发现，严格地遵从和推广这些编程原则，反而让代码更加难以理解、难以维护。

大家发现了编程原则背后的认知偏差——相关性不等于因果。

从高质量的代码中提炼的特征，跟代码质量之间有一定的相关性，但不意味着是因果关系：高质量代码都有某些特征，不意味着有某些特征的代码就是高质量的代码。



如上图所示，假设最大的圈是整个代码宇宙，里面的小绿圈是优质代码，绿圈之外则是低质量代码，黄圈则是符合 SOLID 等编程原则的代码。

我们可以看到，在这种关系中：

- 优质代码大部分具备 SOLID 等编程原则的特征（所以我们能从中提炼出 SOLID 特征）
- 但还有更多符合 SOLID 等特征的代码是低质量代码
- 有一小部分代码不符合 SOLID 特征，但也是优质代码

因此，盲目地在代码库里推广某种代码特征，往往会带来反效果。

1.3 当前代码指南的不足之处

目前的代码质量评估模型和代码指南，仍有以下几个不足之处：

- Subjective，依赖开发者主观经验
- Unclear，表述模糊，不够清晰
- Hindsight，对已写就的代码做事后评估，对写代码本身缺乏建设性指导
- Imprecise，不够精确，不够准确
- External，围绕代码表面的形式，忽视问题的本质特征，或者假设问题已经被解决
- etc.

与上面的不足相对的，我们可能想要的是：

- Objective，更加客观的，所有理性的开发者都有一致的认知
- Clear，表述清晰明确
- Insight，在写代码之前或写代码之时就能帮助洞察问题
- Precise，精确的代码评估标准

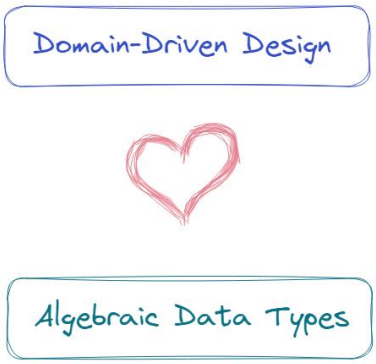
- Internal，围绕问题本质出发，不仅仅是代码的编写形式
- etc.

优雅的代码，不只是某种写法或者编程技巧，而是更深入地认识问题的本质后，所带来的副产品。

因此，代码怎么写，怎么写出高质量的代码，离不开提高对问题的理解水平。

编写高质量代码的指导模型，不能只关注代码怎么写的这一阶段，还需要有怎么认识问题的前置阶段。

二、DDD 的概念与定义

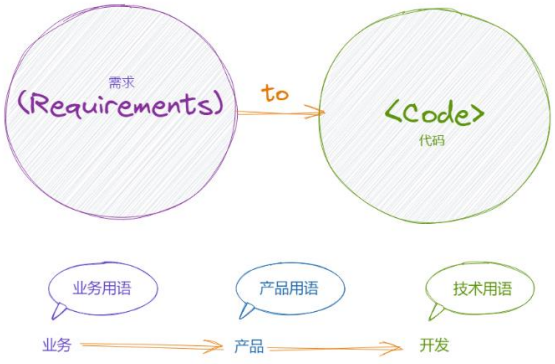


DDD（领域驱动设计）是 2003 年开始流行的一种开发模式，它的含义非常广。

领域驱动设计，不只是关注代码怎么写，更关注写代码之前的活动，比如产研流程和协作。

偏向写代码的 DDD，叫战术型 DDD；偏向产研流程和协作的 DDD，叫战略性 DDD。但它们的核心是一致的，后面我们会展开讨论。在此之前，让我们看看，朴素的产研模式。

2.1 朴素产研模型：需求驱动设计



我们可以把这种模式称之为——需求驱动设计。

它的一般过程是，产品经理跟业务同事们沟通以及做市场调研分析，挖掘出产品需求，并制作成产品需求文档（PRD），然后开发根据 PRD 中的需求描述，提供代码实现。

在这种模式中，业务们有自己的业务用语，产品们也有自己的产品用语，开发工程师有自己的技术用语。它们之间有一定的关联，但并不足够紧密和强烈。

不会有人要求开发写代码时一定要用业务用语里的名词和概念。

从领域驱动设计的角度看，这个模式的问题就在于，不像一个团队，而是三个团队。

开发之间主要通过技术用语交流，消费的是上游产品用语的材料。开发很难确定自己接到的需求，是不是业务真正的需求，自己理解的逻辑，跟业务同事理解的是不是一样的。

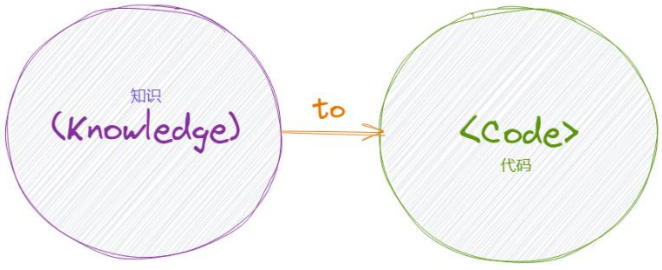
在这种情况下，开发会强烈希望需求是尽可能真实的、明确的、稳定的。开发会很反感上游交付的是不确定的需求，代码频繁修改。

当遇到跟产品难以通过沟通达成共识的时候，开发会想要直接跟业务沟通，了解业务的一手需求。

在需求驱动设计的模式里，需求的生命周期很短，基本上需求发布上线不久之后，它们就被遗弃了。很少有产品团队能够始终维护完整的产品文档，跟线上的功能一一对应。往往在产品经理换人之后，需要通过读代码来反推产品逻辑。

这里问题就来了，代码是在技术用语的上下文里编写的，它们其实不容易反映产品里的概念，更加不用说业务概念了。

2.2 领域驱动设计的核心思想与关键过程



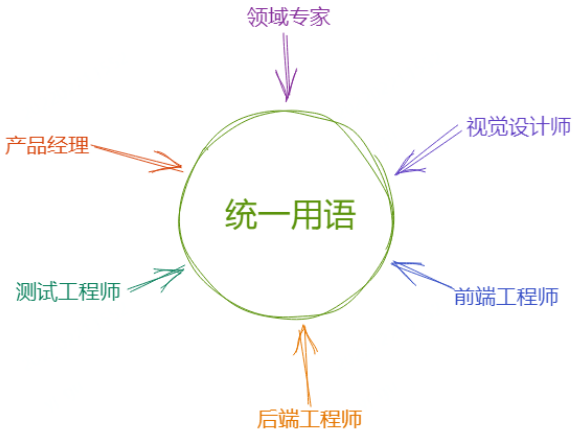
领域驱动设计的产研模型里，所凸显的重点是——知识。

如果说，需求是关于 How、要怎么做的，知识就是关于 What 和 Why 的，是什么，为什么。

知识和需求，也不冲突。需求就来自知识。

我们也可以把领域驱动设计，叫做——知识驱动开发。它强调的是，从领域知识到代码实现的过程。

不管是战术型 DDD，还是战略性 DDD，或者其它 DDD 的方面，都围绕知识驱动开发这个核心思想展开。把握住了核心，不容易被 DDD 里诸多复杂概念所迷惑。



领域驱动设计的关键过程，是解决团队中不同岗位和角色之间，交流语言不统一的问题。

上图中的领域专家，大部分情况下指业务人员。但也不尽然如此，DDD 里的领域专家，不是一个岗位，而是一个角色。在特定领域里更专业、懂得更多、更权威，他就成了领域专家的角色。

通过构建团队统一用语，开发可以更为确定产品需求是真需求，自己理解的逻辑，跟领域专家或业务同事们的理解是一致的。因为大家都用同一套词汇，相同的定义，彼此有共识基础。

领域驱动设计和需求驱动设计的差异，可以从它们的会议形式上管中窥豹。

需求驱动设计的会议，通常是产品经理将已完稿的 PRD 进行讲解，前置的知识提取、梳理和定义等阶段，大体已经成形。

而领域驱动设计的会议，有所不同。如上图是一种被称之为事件风暴（Event Storming）的会议模式。领域专家、开发工程师等多个角色共同在一块长条白板上，基于时间轴和事件进行领域知识的表达和建构。

在这个过程中，业务领域里的实体、事件、流程、关系等诸多概念被提出、定义和明确，在场所有人都对此有共识。每一次事件风暴会议形成的词汇和术语，都作为下一次会议或者工作交流中的统一用语和语料。

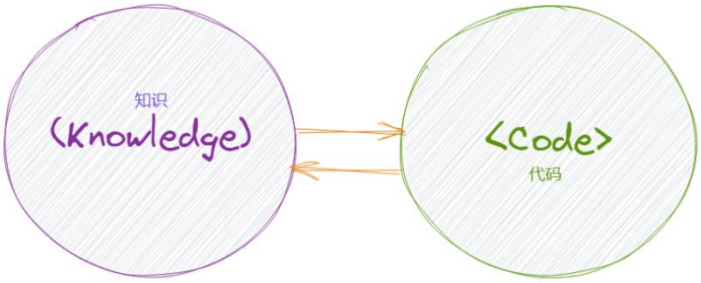
2.3 小结

- 高质量的代码来自对问题的正确认知，很难在不理解问题的基础上优雅地解决问题；
- 代码的写法、风格、模式等手段，建立在正确的认知基础上才能达到最佳的效果；
- 忽视提高对问题的认知水平，盲目地运用代码技巧、设计模式，往往让代码更糟糕；
- DDD 的核心思想是，以领域专家为核心，建立统一用语，确保知识和需求的可靠传递。

三、ADT 的概念与定义

采用领域驱动设计的战略部分，可以优化团队协作模式，确保从知识到代码的过程中，知识部分是基于良好定义的共识所得到的，需求因而更有概率是真实需求。

而领域驱动设计的战术部分，则是反过来：从代码到知识。



简单地说，DDD 要求代码中必须使用“团队统一用语”里的词汇和概念。

代码应当忠诚地反映领域知识。代码里的变量名、方法名以及核心逻辑，应当反映领域知识里的定义。代码编写不是随意的，而是每一个掌握了领域知识的开发者，都能写出大体一致的代码，而非五花八门的多样性实现。

如果说需求驱动设计，要求代码满足产品需求的外延定义。即在输入/输出的功能层面满足产品需求。

那么领域驱动设计，要求代码满足领域知识的内涵定义。不仅在输入/输出的黑盒层面满足功能要求，在代码细节的白盒层面也满足领域知识的定义。

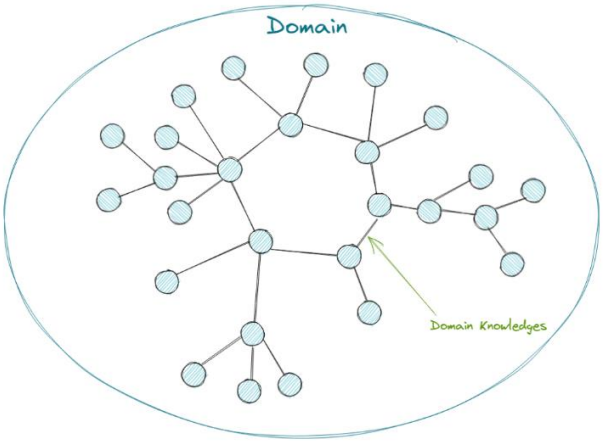
这要求我们必须知道，如何把领域知识翻译成代码实现。

代数数据类型(ADT, Algebraic Data Types)是支撑上述需求的关键编程特性。

3.1 领域和领域知识的定义

首先我们需要精确定义什么是领域（Domain），什么是领域知识(Domain Knowledge)。

- Domain（领域）是一系列关联问题构成的集合
- Domain Knowledge (领域知识) 是一系列关联问题涉及的所有真命题 (true proposition) 的集合



所有领域，都由一些关键概念节点构成，领域知识则是这些概念节点之间的关系的定义，表达为逻辑上的真命题。

作为命题的领域知识，在产品需求里被表述为一系列领域规则/业务规则。

产研团队常说的业务逻辑，可以被翻译为更严格的数学逻辑的表达形式。

3.2 柯里-霍华德同构

用代码表达领域知识的原理——柯里-霍华德同构(Curry–Howard isomorphism)。

Logic Name	Logic Notation	Type Notation	Type Name
True	$\top$	$\top$	Unit Type
False	$\bot$	$\bot$	Empty Type
Not	$\neg A$	$A \to \bot$	Function to Empty Type
Implication	$A \to B$	$A \to B$	Function
And	$A \wedge B$	$A \times B$	Product Type
Or	$A \vee B$	$A + B$	Sum Type
For All	$\forall a \in A, P(a)$	Π a : A . P(a)	Dependent Function
Exists	$\exists a \in A, P(a)$	Σ a : A . P(a)	Dependent Product Type

- 命题即类型，证明即程序 (Propositions as Types, Proofs as Programs)
- Type-Driven Development (类型驱动开发)，用类型去表达领域知识 (领域里的真命题)
- 符合类型的所有值，都是该类型所表征的命题的证明 (Witness)
- 真命题：至少有一个值的类型
- 假命题：没有任何值的类型

柯里-霍华德同构，给出了将命题翻译为类型的方式，它提供了将领域知识翻译成代码实现的通用方式。

特别是对应着 And 关系的 Product Type，和对应着 Or 关系的 Sum Type，它们两个构成了代数数据类型(ADT)。

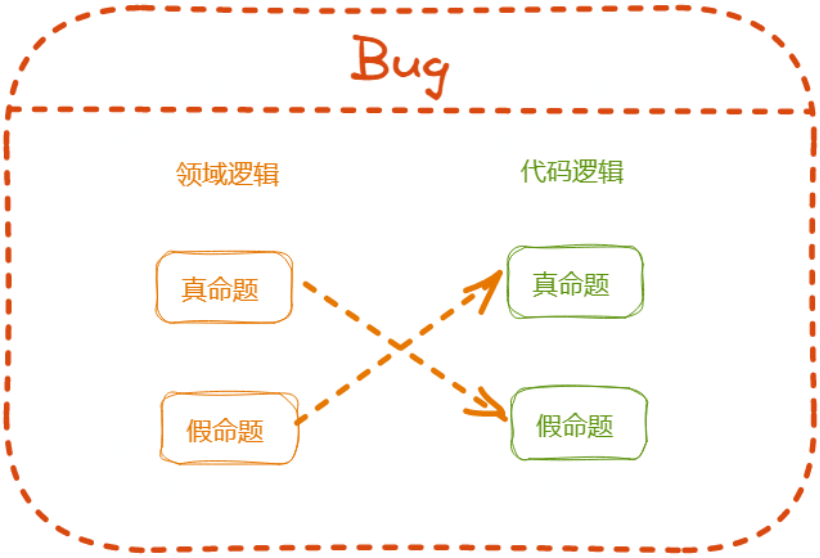
后续我们将演示，如何用 Product Type 和 Sum Type 做数据建模和流程建模，跟领域知识/业务规则一一对应起来。

值得一提的是，在柯里-霍华德同构中，真命题和假命题，并不是对应着 Boolean Type 的 true 和 false。而是类型所允许的值的数量(size)，0 个值为假命题，至少 1 个值为真命题。

也就是说，所有能运行的代码在这个意义上都是真命题，抛错等行为才意味着碰到程序的假命题。

3.3 Bug 的定义和高质量代码的判别标准

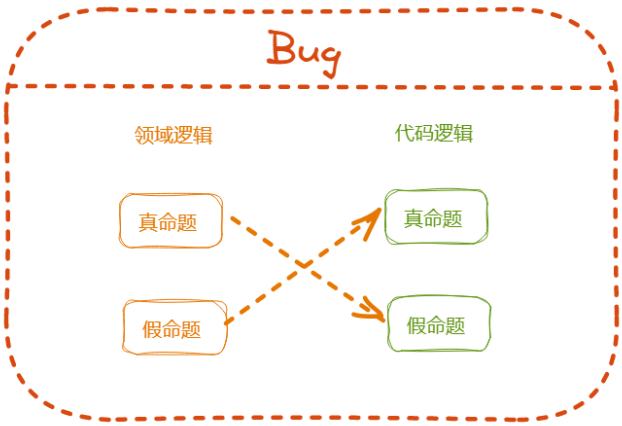
通过柯里-霍华德同构，Bug 的定义，以及高质量的代码的判别标准，也有了更清晰的陈述。



在将领域逻辑翻译成代码逻辑的过程中，如果发生匹配错乱，领域里的命题跟代码里的命题不一致，Bug 就发生了。

- 领域里的真命题（领域知识），代码里却是假命题（Error/Crash/Halt）
- 领域里的假命题，代码里却是真命题（Illegal-States/Unexpected-Behaviors）

所有能跑的程序都是程序意义上的真命题，但不意味着是业务意义上的真命题，这种不匹配被称之为非法状态（Illegal-States）、非法操作（Illegal-Operations）或不预期的行为（Unexpected Behaviors），是应当被避免的。



所谓的高质量代码，与 Bug 的定义相反，领域里的命题跟代码里的命题一致。

- 领域里的真命题（领域知识），代码里也是真命题
- 领域里的假命题，代码里也是假命题

这意味着，代码里运行着的程序和数据，需要都是有业务意义的。更少的非法状态、非法操作和不预期的行为。

3.4 类型论（Type Theory）的基础知识

In type theory, every term has a type. A term and its type are often written together as "term : type".

类型论和集合论在某些方面很类似，它们都是对合集（collection）的不同建模。在这里，合集（collection）是指一堆事物的聚合。

在类型论中，事物被称之为项（term），它有且只有一个类型（type）。而类型（type），可以有零到任意多个项（term）。其中一些典型的类型，列举如下：

- 空类型（Empty type），0 个项；
- 单元类型（Unit type），1 个项；
- 布尔类型（Boolean type），两个项，true 或者 false；
- 自然数类型（Natural Numbers type），无限多个项，从 0 开始。
- etc.

如上，我们例举了一些基础类型，它们的 term 的数量也标记了出来，跟集合论里的元素数量类似。

3.5 代数数据类型（Algebraic Data Types）

In type theory, an algebraic data type is a kind of composite type, i.e., a type formed by combining other types.

所谓的代数数据类型（ADT），指的是多个类型组合成新的类型的方式，这些组合方式拥有一些代数特征。

由前面的柯里-霍华德同构，我们知道，逻辑的 Or，对应的类型是 Sum type，表达的是互斥的、或的关系。而逻辑的 And，对应的类型是 Product type，表达的是并存的，与的关系。

其中，Sum 含义是相加，Product 是指相乘，描述的是 Sum type 和 Product type 的项的数量，跟它的组成部分的类型的项数量的关系。

Sum type: size(A | B) = size(A) + size(B)

Product type: size(A & B) = size(A) * size(B)

如上，size(..) 函数可以获取特定 type 所允许的 term 的数量。

那么，Sum type 对应的就是加法，因为 A 和 B 在这里是“或”的关系，是互斥的，不会一起出现，所以要么 A，要么 B。那所有可能性就是 A 的可能数量，加上 B 的可能数量。

```
// sum types
type Value = string | number;

const a: Value = 'John';
const b: Value = 70;
```

而 Product type 对应的是乘法，因为 A 和 B 在这里是“与”的关系，是并存的，会一起出现，所以既有 A，也有 B。那就进入 A 和 B 的组合可能性，每一个 A 都能搭配所有 B，有 A 个 B，即 A * B。

```
// product types
type Person = { name: string; age: number };
// type Person = { name: string } & { age: number }

const person: Person = { name: 'John', age: 70 };
```

如上，对象的字段之间是 product type 关系。product type 不是具体的某个类型，而是一系列类型，只要它们背后的 size 关系是乘法相关的。同理，sum type 指的是那些背后的 size 关系是加法相关的类型。

只需要了解到目前的程度，代数数据类型 (ADT) 已经可以优化我们的代码质量了。接下来，我们来看看它在数据建模和流程建模上的应用。

四、案例：用 DDD + ADT 做数据建模

假设有以下用户信息的领域规则（业务定义）：

- 用户要么是已登录用户，要么是未登录用户（游客）
- 游客拥有随机的昵称
- 已登录用户拥有昵称、Email 信息
- Email 信息要么是已验证的 Email，要么是未验证的 Email
- 已验证的 Email 有验证时间戳
- 用户信息通过 Http API 获取

它们被描述为一组自然语言描述的规则，其实也可以提取出一些逻辑语言描述的命题。比如：

- 登录用户拥有昵称——true，真命题

- 未登录用户拥有 Email 信息——false，假命题
- 未验证的 Email 有时间戳——false，假命题
- etc

因此，领域知识未必是直白的逻辑语言，但它们总是可以提炼出逻辑命题，这些命题就是我们翻译到类型的指引。

4.1 常见的数据建模

```
type UserInfo = {
  // 当用户未登录时，id 为空字符串
  id: string;
  // 当用户未登录时，name 为随机生成的昵称
  name: string;
  // 当用户未登录时，email 为空字符串
  email: string;
  // 用户是否登录
  isLogin: boolean;
  // 当邮箱未验证时，这个字段为 false
  isEmailVerified: boolean;
  // 当邮箱已验证时，这个字段为验证时间戳，否则为空字符串
  emailVerifiedAt: string;
};

// Http API 获取用户信息
type JsonResponse = {
  error?: string;
  // 当 error 为空时，这个字段为用户信息
  data?: UserInfo;
};
```

本案例中的用户信息的数据建模，其实非常简单，大部分开发者都习以为常，很容易写出像上面那种代码，并且不觉得有任何问题。代码简洁、清晰并且直观，注释完整。即便不能说是优质代码，起码是不坏的代码。

4.2 基于 DDD + ADT 的领域类型建模

采用 DDD + ADT 的模型，用户信息的类型大致如下：

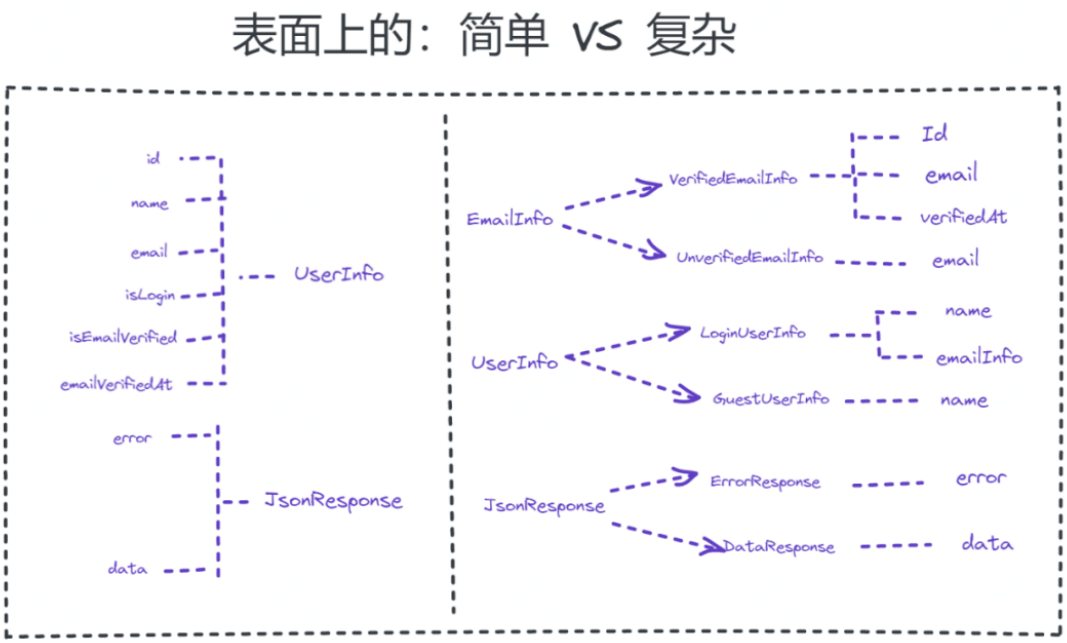
```
type VerifiedEmailInfo = {
  type: 'VerifiedEmailInfo';
  email: string;
  verifiedAt: string;
};
```

```
type UnverifiedEmailInfo = {
  type: 'UnverifiedEmailInfo';
  email: string;
};

type EmailInfo = VerifiedEmailInfo | UnverifiedEmailInfo;

type LoginUserInfo = {
  type: 'LoginUserInfo';
  id: string;
  name: string;
  emailInfo: EmailInfo;
};

type GuestUserInfo = {
  type: 'GuestUserInfo';
  name: string;
};

type UserInfo = LoginUserInfo | GuestUserInfo;

type ErrorResponse = {
  type: 'ErrorResponse';
  error: string;
};

type DataResponse = {
  type: 'DataResponse';
  data: UserInfo;
};

type JsonResponse = ErrorResponse | DataResponse;
```

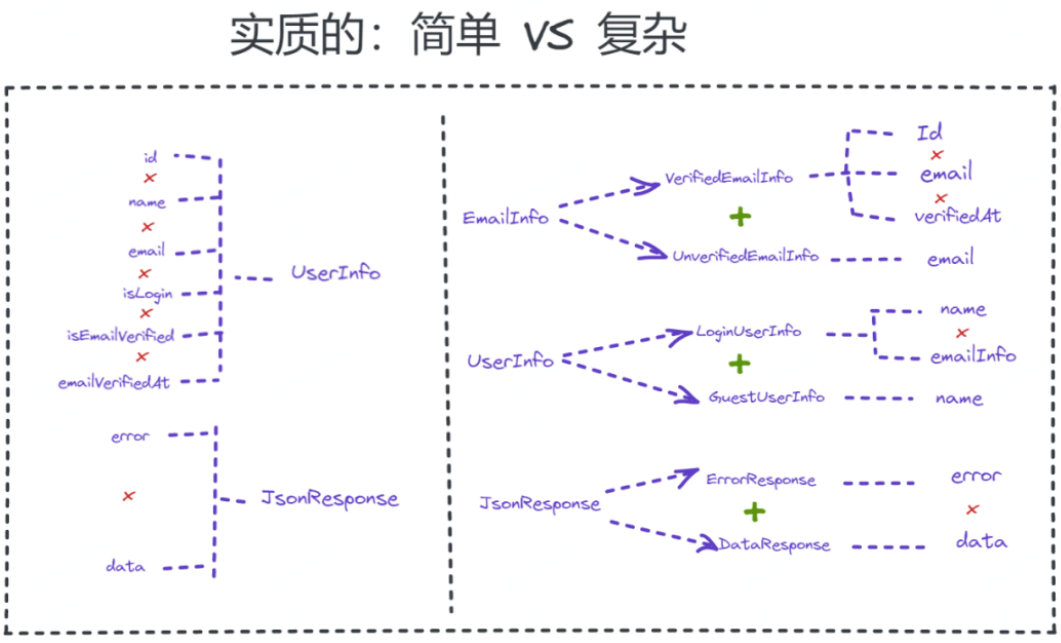
我们一口气写了 9 个类型，看起来复杂了许多，更多的嵌套，更多的重复字段(如 email, name 等)，一行注释都没有，代码行数也翻倍了。

然而，上面的代码，更加符合业务规则的描述，更加准确地匹配了领域知识里的 And 和 Or 的关系。如果说它看起来不如第一种简单，那这个复杂度也是领域知识里自身的复杂度，更简单的定义某种意义上是——过度简化。

此外，代码的复杂度，跟代码的长度，不是必然关系。表面上的简单，和实质的简单，也不是一回事儿。



表面上，左侧拥有更少的嵌套，更少的字段，更少的类型，更简短的代码，在形式上无疑是比右侧的更简单的。



实际上，当我们把类型之间的关系标记出来时，我们发现，左侧其实拥有更多的乘法，而右侧则包含几个加法。多个乘法复杂度的类型，允许的 term 的可能性更多，size 更大。因此，左侧的 term size 大于右侧的。

4.3 非法状态对代码库的腐坏

当类型的命题空间，大于领域规则的命题空间，所多出来的部分，就是非法状态的空间。

非法状态将持续腐坏我们的代码库，从以下几个方面可见一斑。

（1）非法状态鼓励错误的代码

```
const handleLoginUser = (userInfo: UserInfo) => {
  // 直接访问 email，而没有先判断是否 isLogin
  console.log('login user email', userInfo.email);
}
```

对象的字段之间是 product type 的并存关系，即便是未登录的用户，也有 email 字段，只是为空字符串。开发者需要主动的、自觉的记得判断是否登录，否则将产生错误隐患。

（2）非法状态导致过度防御性编程，增加代码复杂度和代码量

```
const handleLoginUser = (userInfo: UserInfo) => {
  // 防御性编程，判断是否已登录
  if (!userInfo.isLogin) {
    return;
  }
  console.log('login user email', userInfo.email);
}

const handleLoginUser1 = (userInfo: UserInfo) => {
  // 防御性编程，判断是否已登录
  if (!userInfo.isLogin) {
    return;
  }
  console.log('login user email', userInfo.email);
}
```

在所有消费 UserInfo 的函数或者方法中，必须添加防御性逻辑，手动验证和排除非法状态，然后再消费数据。

也就是说，定义类型时所节省的复杂度，在类型消费的所有地方，都额外增加了防御性代码。数据的类型定义只有一处，但数据的消费却可以有很多处。相较之下，整体代码量更多，代码复杂度更高。

（3）非法状态带来更多逻辑不同步

很多防御性判断，是在迭代过程中，一点点累积起来的。一开始往往没有收口到通用的函数里，它们重复地出现在多个消费数据的函数里。当需要更新某个防御性判断时，需要开发者记得在所有修改的地方，都改一遍。任何遗漏，都带来防御逻辑的不一致。

正因如此，Don't Repeat Yourself (DRP) 才作为一个最佳实践的指导原则被提出。在这个场

景中，DRP 属于治标的做法，治本的做法则是从源头解决，用更精确的类型定义，减少不必要的防御性判断。

（4）非法状态带来更差的性能

所有消费数据的函数和方法，都需要带上特定的防御性逻辑。尽管我们可以通过 DRP 原则，将重复的防御性逻辑，收口在一处。但它们仍需在各个消费函数中被调用。

这些消费函数彼此互相调用时，防御性逻辑将被重复执行。即便上一个函数已经调用过，下一个函数依然要进行防御。因为它无法判断是否会被独立的调用，它自身需要做好防御性逻辑的内聚。

因此，非法状态将带来更差的性能，数据验证工作在代码运行期间反反复复地被计算。

4.4 知识和代码同构的巨大收益

与非法状态相反，当代码里的类型更加忠实地反映领域知识，非法状态被减少或消除，它们难以被构造和传播。领域知识被编码到类型里，由 type-checker 进行约束。

从以下几个方面带来巨大收益：

（1）拒绝错误的代码

```
type UserInfo = LoginUserInfo | GuestUserInfo;

declare let user: UserInfo
```

```
any
Property 'emailInfo' does not exist on type 'UserInfo'.
  Property 'emailInfo' does not exist on type 'GuestUserInfo'. ts(2339)
View Problem (Ctrl+K N)    No quick fixes available
```

```
user.emailInfo
```

UserInfo 是一个 Sum type，有两种可能性 LoginUserInfo 和 GuestUserInfo。其中 GuestUserInfo 完全符合领域规则描述，只有一个 name 属性，而没有 emailInfo。

因此，user.email 无法通过类型检查。必须先证明 user 属于登录用户，才能访问 emailInfo 属性。

```
type UserInfo = LoginUserInfo | GuestUserInfo;

declare let user: UserInfo

if (user.type === 'LoginUserInfo') {
  user.emailInfo
} else if (user.type === 'GuestUserInfo') {
        any
        Property 'emailInfo' does not exist on type 'GuestUserInfo'. ts(2339)
        View Problem (Ctrl+K N)   No quick fixes available
  user.emailInfo
}
```

如上，当 user.type === LoginUserInfo 时，我们可以访问 emailInfo。而 user.type === GuestUserInfo 时，emailInfo 是不能访问的。

当我们正确地用 Sum type 表达领域知识里的 Or 的关系，我们更难写出错误的代码。

（2）减少不必要的防御性逻辑及其运算开销

```
const handleLoginUser = (userInfo: LoginUserInfo) => {
  console.log('login user name', userInfo.name);
}

const handleLoginUser1 = (userInfo: LoginUserInfo) => {
  handleLoginUser(userInfo);
}

const handleLoginUser2 = (userInfo: LoginUserInfo) => {
  handleLoginUser1(userInfo);
}
```

我们可以直接使用 Sum type 中，我们感兴趣的部分类型，构造我们的函数。如上所示，处理登录用户时，我们直接用 LoginUserInfo 类型，不必在函数内部防御是否登录。LoginUserInfo 存在，已经意味着已登录。

handleLoginUser1 调用了 handleLoginUser，handleLoginUser2 又调用了 handleLoginUser1。它们都没有额外的防御性逻辑代码。

```
if (user.type === 'LoginUserInfo') {
  handleLoginUser2(user)
} else {
        let user: GuestUserInfo
        Argument of type 'GuestUserInfo' is not assignable to parameter of type 'LoginUserInfo'.
          Type 'GuestUserInfo' is missing the following properties from type 'LoginUserInfo': id,
        emailInfo ts(2345)
        View Problem (Ctrl+K N)   No quick fixes available
  handleLoginUser2(user)
}
```

最外部调用 handleLoginUser2 时，才进行 sum type 分流判断。某种程度上，可以理解为，防御性代码被隔离到最外部的函数调用中。内部函数编写和组合时，代码更短、更安全、更少冗余开销。

（3）代码更加容易阅读和维护

相比传统模式，领域知识被放到注释里，描述字段之间的协同关系的业务含义。DDD + ADT 的领域知识，就在类型里。

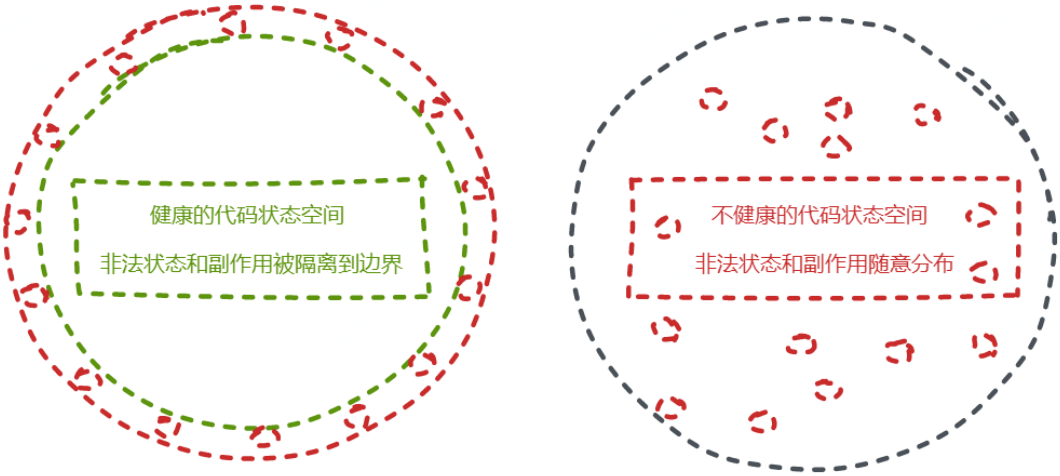
我们不必去猜测 user.isLogin, user.name, user.emailInfo 彼此之间的依赖关系，猜测有多少种可能的组合和场景。

```
type EmailInfo = VerifiedEmailInfo | UnverifiedEmailInfo;
type UserInfo = LoginUserInfo | GuestUserInfo;
type JsonResponse = ErrorResponse | DataResponse;
```

我们可以很直观地看到，两种就是两种，没有注释也能正确理解。不仅开发者理解，编译器也理解，并在每次消费 Sum type 时，约束开发者使其难以忘记和误解。

4.5 小结

- 领域知识里 Or 的关系，被曲解为 And，类型由加法复杂度，变成乘法复杂度；
- 代码上能写出来的值 (value) 的数量 (terms size)，大于领域知识里的真命题的需求；
- 代码里的真命题(多出来的值)，是领域里的假命题，它们成了非法状态 (Illegal-States)；
- 所有消费数据的地方，都需要做防御性判断，排除非法状态，否则就导致程序出现 BUG；
- 系统的可维护性，跟非法状态在代码库里的泄漏程度成反比，泄漏越多，越难以维护和预测。



不健康的代码状态空间，非法状态和副作用随机分布。需要靠开发者付出更多额外的努力、写更多的注释、更多的防御性代码……，才能缓解代码腐坏的进程。然而，没有从源头解决

问题，治标不治本，最终非法状态的蔓延，将很容易超出开发团队的掌控能力。特别是在人员流失和更替的过程中，领域知识在上一任开发者的脑海里，随着上一任的离去而丢失，在下一任开发者在脑海里重新建立领域知识的过程中，代码库可能已经加速腐坏。

通过 DDD + ADT, 我们可以构建更健康的代码状态空间, 用更精确和反映领域知识的类型, 将非法状态的防御性判断逐层隔离到边界。让我们的核心代码变得简单可靠，领域知识被编码到类型里，由编译器的 type-checker 进行保证。即便开发者产生更替，编译器依然可以做出正确的提示。

通过 ADT 让非法状态无法被表示出来，从根源上优化代码质量。

五、案例：用 DDD + ADT 做数据建模

假设有以下领域规则:

- 用户发帖有 3 个阶段：草稿、审核、发布
- 草稿不能跳过审核直接发布
- 草稿可以提交审核
- 审核通过后可以发布
- 审核中的帖子不能修改
- 审核不通过退回草稿阶段

5.1 常见的流程建模

```
class Post {
  constructor(
    private isDraft: boolean,
    private isReviewing: boolean,
    private isPublished: boolean,
    private content: string
) {}
  edit(content: string) {
    if (!this.isDraft) {
      throw new Error('Post is not in draft stage');
    }
    this.content = content;
  }
  review() {
```

```
    if (!this.isDraft) {
      throw new Error('Post is not in draft stage');
    }
    this.isDraft = false;
    this.isReviewing = true;
  }
  publish() {
    if (!this.isReviewing) {
      throw new Error('Post is not in reviewing stage');
    }
    this.isReviewing = false;
    this.isPublished = true;
  }
  reject() {
    if (!this.isReviewing) {
      throw new Error('Post is not in reviewing stage');
    }
    this.isReviewing = false;
    this.isDraft = true;
  }
}
```

很多开发者很自然地编写出了上述代码逻辑。当调用 edit 方法编辑内容时，会先判断是否处于草稿阶段。每个相关方法内，都有状态验证。

问题在于，这种做法跟业务规则不是同构的。在业务规则中，编辑、审核、发布、退回等操作，不完全是并存的，而是随着草稿阶段、审核阶段、发布阶段而变化。但在 Post Class 中，edit, review, publish 和 reject 等方法并存，是 product type 的关系，没有忠实地体现领域知识。

因此，Post 的实例，存在很多非法操作 (Illegal Operations)。每一次方法调用，都需要调用者提前判断当前阶段，否则调用 edit 等方法将抛出错误。有意无意地忘记提前防御判断，Bug 将蔓延在代码库里。

当我们把各个方法里的防御性逻辑，从 throw error 改成静默，即只在符合条件时执行操作，否则什么都不做。那么，非法操作 (Illegal Operations)将变成不预期行为 (Unexpected Behaviors)。也就是说，所有方法调用，我们都不确定是否产生了效用，常常仍需额外的判断逻辑去确认。

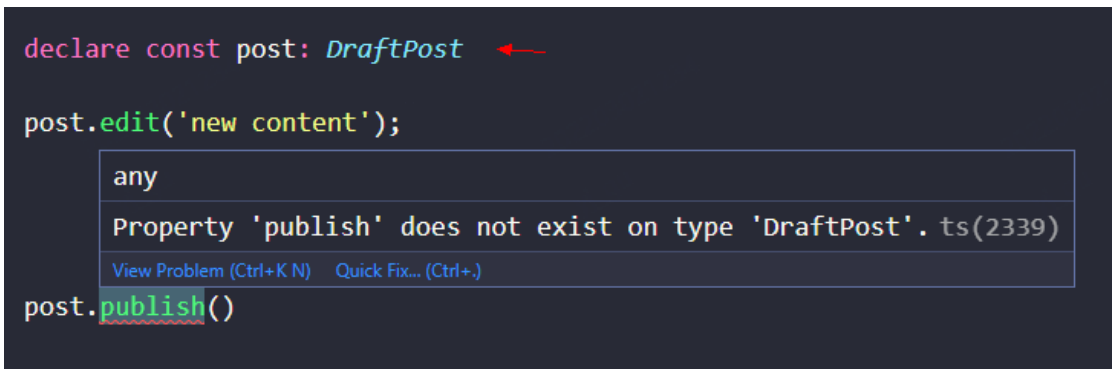
非法操作一旦存在，不管以何种方式隐瞒不报，都会持续腐坏代码库。

5.2 基于 DDD + Class 的忠实流程建模

```
export class DraftPost {
```

```
    constructor(private content: string) {}
    edit(content: string) {
        this.content = content;
    }
    review() {
        return new ReviewingPost(this.content);
    }
}

class ReviewingPost {
    constructor(private content: string) {}
    publish() {
        return new PublishedPost(this.content);
    }
    reject() {
        return new DraftPost(this.content);
    }
    approve() {
        return new PublishedPost(this.content);
    }
}

class PublishedPost {
    constructor(private content: string) {}
    getContent() {
        return this.content;
    }
}
```

如上所示，我们定义了三个 Class，分别表达三个阶段的 Post，对于每个阶段，所允许的方法都精确对应了领域规则。只有 DraftPost 拥有 edit 方法，是可编辑的；只有 ReviewingPost 拥有 publish 方法，是可发布的。



当我们获取到 DraftPost 实例，我们可以编辑它，但不能跳过审核直接 publish 发布它。

```
declare const post: ReviewingPost
      any
      Property 'edit' does not exist on type 'ReviewingPost'. ts(2339)
      View Problem (Ctrl+K N)    Quick Fix... (Ctrl+.)
post.edit('new content');
post.publish()
```

当我们获取到 ReviewingPost 的实例，我们可以发布它，但不能编辑它。

```
declare const post: PublishedPost

post.edit('new content');
post.publish()
```

当我们获取到 PublishPost 的实例，我们既不能编辑，也不能重复发布它。

```
const draftPost = new DraftPost('draft post content');
const reviewingPost = draftPost.review();
const publishedPost = reviewingPost.approve();
const publishedPostContent = publishedPost.getContent();

console.log(publishedPostContent);
```

业务规则描述的流程，被编码到各个阶段的 Post 的方法调用的传递中，由编译器的 type-checker 去约束。

5.3 基于 DDD + ADT 的忠实流程建模

```
type DraftPost = {
  type: 'DraftPost';
  content: string;
}

type ReviewingPost = {
  type: 'ReviewingPost';
```

```
    content: string;
}

type PublishedPost = {
    type: 'PublishedPost';
    content: string;
}

const edit = (post: DraftPost, newContent: string): DraftPost => {
    return {
        ...post,
        content: newContent
    }
}

const review = (post: DraftPost): ReviewingPost => {
    return {
        type: 'ReviewingPost',
        content: post.content
    }
}

const approve = (post: ReviewingPost): PublishedPost => {
    return {
        type: 'PublishedPost',
        content: post.content
    }
}

const reject = (post: ReviewingPost): DraftPost => {
    return {
        type: 'DraftPost',
        content: post.content
    }
}
```

除了用 Class 进行流程建模以外，面向数据的 ADT 也能做到，两者在流程建模的表达上是等价的。区别在于，数据和行为不再被放到一起，而是分开定义。但我们的 edit 只接受 DraftPost 数据，因而表达了只有草稿阶段才能编辑。review, approve, reject 等函数同理。

5.4 小结

- 将互斥的操作放到一起并存，关系从 Or 变成了 And，从加法复杂度变成乘法复杂度；
- 代码上能调用的函数/方法的数量 (terms size)，大于领域知识里的真命题的实际需求；

- 代码里的真命题（多出来的方法调用），是领域里的假命题，它们成了非法操作（Illegal-Operations）；
- 所有调用方法的地方，都需要做防御性判断，排除非法调用，否则可能导致程序抛错和出 Bug；
- 系统的可维护性，跟非法操作在代码库里的泄漏程度成反比，泄漏越多，越难以维护和预测。

'Making illegal operations unrepresentable'

采用 DDD + ADT 的模型，让非法操作不能被调用，从根源上优化代码质量。

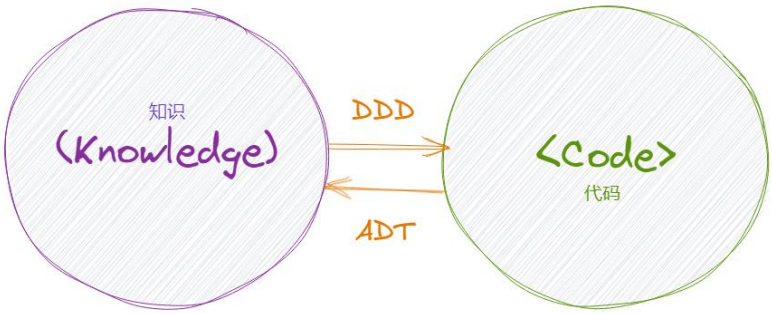
六、总结

不管是用户信息的数据建模，还是 Post 的流程建模，都是很常见的业务需求。它们的逻辑并不复杂，甚至在本文中还做了简化。即便如此，我们可以看到，大部分开发者下意识的代码实现，都包含着诸多隐患。存在很多难以消除的非法状态，以及难以管理的非法操作，不断地侵蚀着代码库。

可以想象，随着项目迭代，持续泄漏的非法状态和非法操作，将指数级地增加项目复杂度，让代码库难以理解、难以阅读和难以维护。犯错是被鼓励的，领域知识被写在不可运行和检查的注释中，代码的性能、逻辑一致性等诸多指标得不到保障。

而 DDD + ADT 模式，可以从根源上改变现状，优化代码库的整体质量。

- 运用领域驱动设计（DDD），建立团队统一用语，获得可靠的领域知识，挖掘真实需求；
- 运用代数数据类型（ADT），对领域知识进行一比一建模，获得可靠的代码设计；
- DDD+ADT：从知识中可以推导出代码，从代码中可以推导出知识，知识与代码的同构；
- 核心技巧：多用 Sum type，少用 Product type，减少非法状态和非法操作的泄漏。



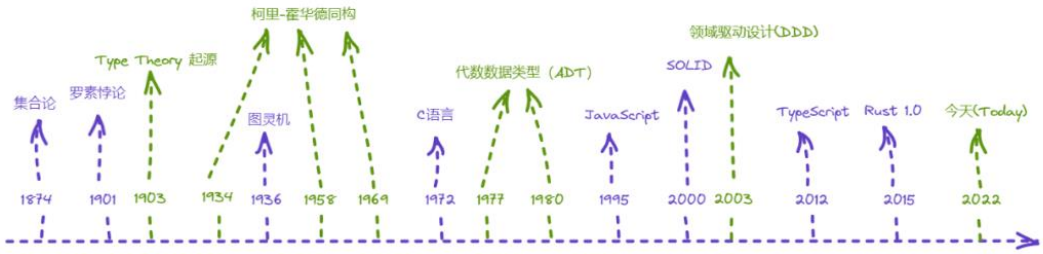
不仅从产品功能和行为的外延意义上，满足了业务需求；从代码的逻辑细节等内涵意义上，也满足了业务知识的要求。

让我们再回顾一下，我们想要的代码质量提升模型：

- Objective，更加客观的，所有理性的开发者都有一致的认知
- Clear，表述清晰明确
- Insight，在写代码之前或写代码之时就能帮助洞察问题
- Precise，精确的代码评估标准
- Internal，围绕问题本质出发，不仅仅是代码的编写形式

ADT+DDD 模式，更好地达到了上述目标。它们不仅仅是在代码工程领域的经验总结，背后其实有着一个世纪的学术积累与沉淀。

用 Phillip Wadle 的话来说，ADT 不是发明的，而是发现的。大部分开发者使用的编程语言及其特性，主要都是人类的发明。现在邀请大家来用人类所发现的语言特性。



- 1874 年，康托尔建立集合论；
- 1901 年，罗素发现集合论里的罗素悖论；
- 1903 年，罗素用类型的思想，试图克服悖论，被视为类型论的起源之一；
- 1934~1969 年，柯里和霍华德分别发现，类型论和逻辑演绎系统隐含的对应关系；
- 1936 年，图灵发表图灵机模型；
- 1972 年，C 语言诞生；
- 1977~1980 年，代数数据类型（ADT）出现在函数式编程语言中；
- 1995 年，前端的 JavaScript 语言诞生；
- 2000 年，SOLID Principles 被归纳发表；
- 2003 年，领域驱动设计（DDD）被提出；
- 2012 年，前端的 TypeScript 语言诞生（本文的示例语言）；；
- 2015 年，Rust 语言 1.0 版本发布
- 2022 年，今天，我们的文章发表时间点。

每一行 DDD + ADT 的建模代码，背后都承载着历史的厚重、闪耀着人类理性的光辉。

Flutter 在携程复杂业务的高性能之旅

【作者简介】本文为联合撰稿，作者为携程火车票 Flutter 团队，关注 Flutter 开发的效率、质量和新技术，致力于提升 Flutter 业务流畅度。

一、背景

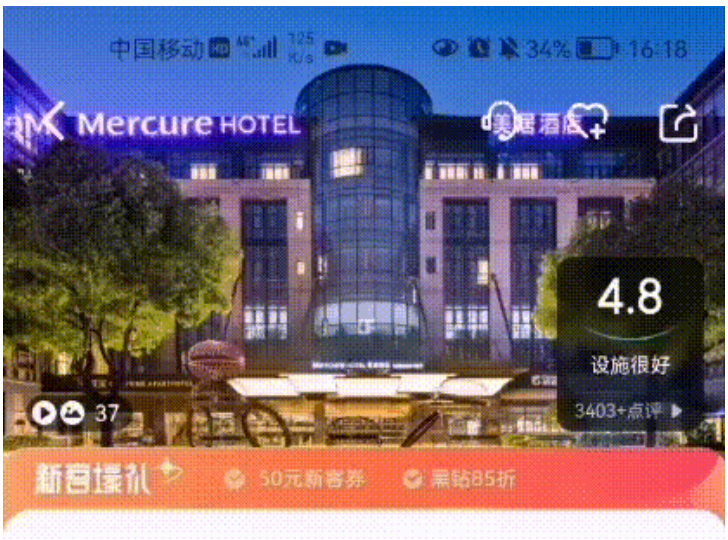
携程火车票在十余个核心业务的列表页及主流程大规模进行了 Flutter 实践。经过一年多的开发、维护 ，总结了一套行之有效的性能优化方案。本文主要介绍结合性能分析工具，来识别、区分、定位一些性能问题，并且能够找到具体的方法和代码位置，帮助更快地解决问题。此外，也会分享我们做的一些性能优化案例和体验上的优化，希望能够给你带来一些启发。

二、渲染优化

Flutter 渲染性能问题主要可以分为 GPU 线程问题和 UI 线程（CPU）问题两种。通过 Performance Overlay 工具就能很清晰的分辨出来。UI 线程图表报红或者两个图表都报红，则表示 Dart 代码消耗了大量资源，需要优化代码执行时间。再结合火焰图， 分析 CPU 的调用栈就能很轻松的找到哪个方法的耗时长，方法名是什么，渲染的层级有多深，而且还能做到性能优化前后的一个对比。 如果仅仅是 GPU 线程图表报红的话，意味着渲染的图形太复杂，导致无法快速渲染。有时候 Widget 树的构建很简单，但是 GPU 线程的渲染却很耗时，就要考虑是否过度渲染，缺少组件缓存，涉及到 Widget 的裁剪、蒙层这类多视图叠加的渲染。

2.1 Selector 控制刷新范围

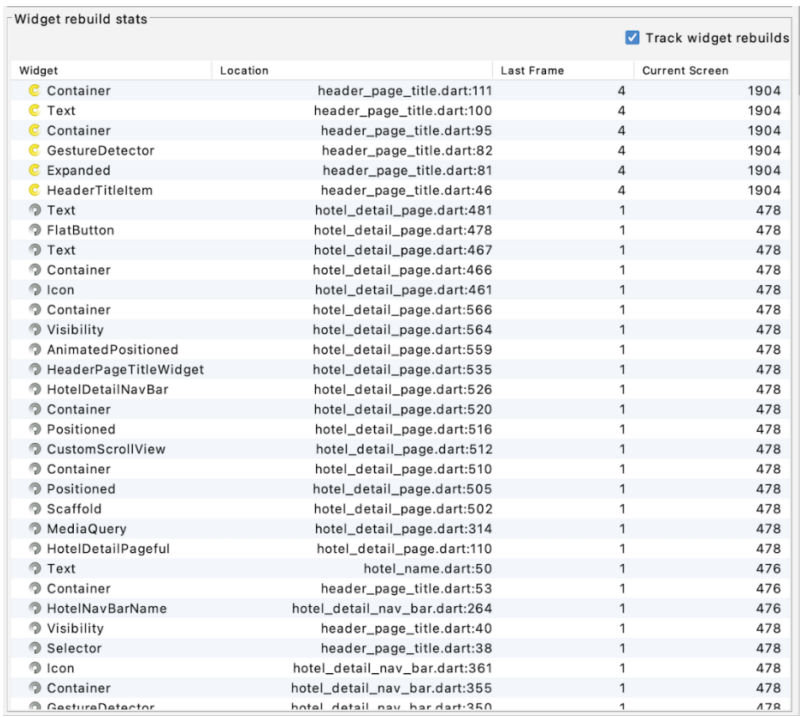
在 StatefulWidget 中，很容易通过 setState 来进行渲染刷新界面，要尽量的控制刷新范围，避免不必要的界面组件重新渲染，使得 GPU 消耗过大，造成界面卡顿。举个例子如下所示:



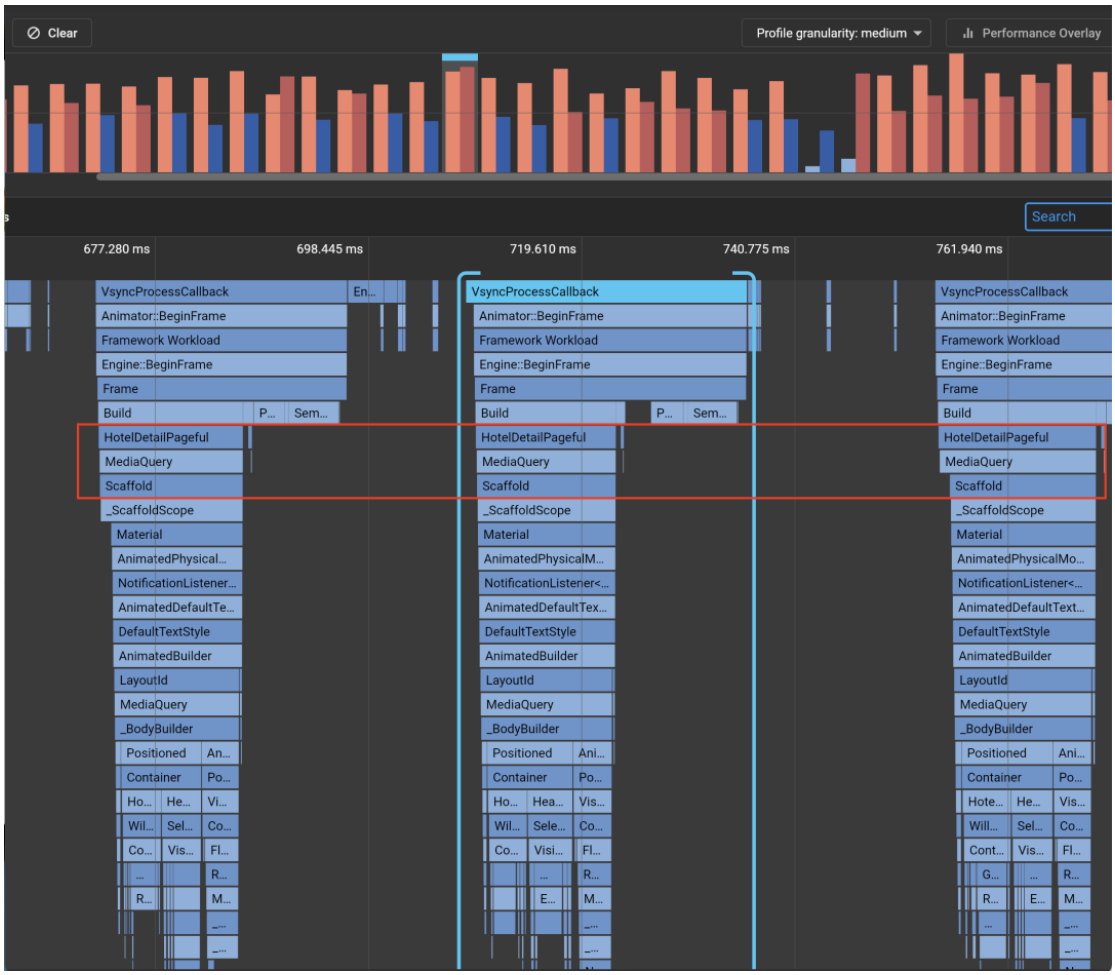
在界面滚动的时候，我们需要监听 CustomerScrollView，然后设置顶部悬浮组件的透明度去实现效果，代码如下：

```
/// 动画距离
int scrollHeight = 120;
_scrollController.addListener(() {
  if (_scrollController.offset > scrollHeight && _titleAlpha != 255) {
    setState(() {
      _titleAlpha = 255;
    });
  }
  if (_scrollController.offset <= 0 && _titleAlpha != 0) {
    setState(() {
      _titleAlpha = 0;
    });
  }
  if (_scrollController.offset > 0 && _scrollController.offset < scrollHeight) {
    setState(() {
      _titleAlpha = _scrollController.offset * 255 ~/ scrollHeight;
    });
  }
});
```

根据滚动距离，设置透明度；但是 setState 会去刷新整个界面，整个界面的组件都会被重新渲染。通过 Flutter Performance 查看组件渲染次数，发现整个界面都在刷新，当我们多次滑动页面后，发现很多组件都渲染了多次，如下图所示：



通过 DevTools，在滑动改变顶部的透明度时，发现 FPS 值很低，而且几乎每一帧都会超过 16ms，火焰图很深，说明渲染的层级很深，整个界面的组件自上而下都重新渲染了，如图所示：



现在就能理解为什么在用户滑动界面的时候会造成卡顿了，主要是由于渲染消耗过大，没有控制好界面的刷新范围。当改变顶部悬浮组件的时候，只需要改变顶部组件状态，而没有必要刷新整棵树。改造策略是通过 Provider 的 Selector 进行控制刷新范围的，将透明度值存放在 ChangeNotifier 的子类中，当透明度发生改变时，通过 notifyListeners()函数通知界面刷新。

监听代码如下：

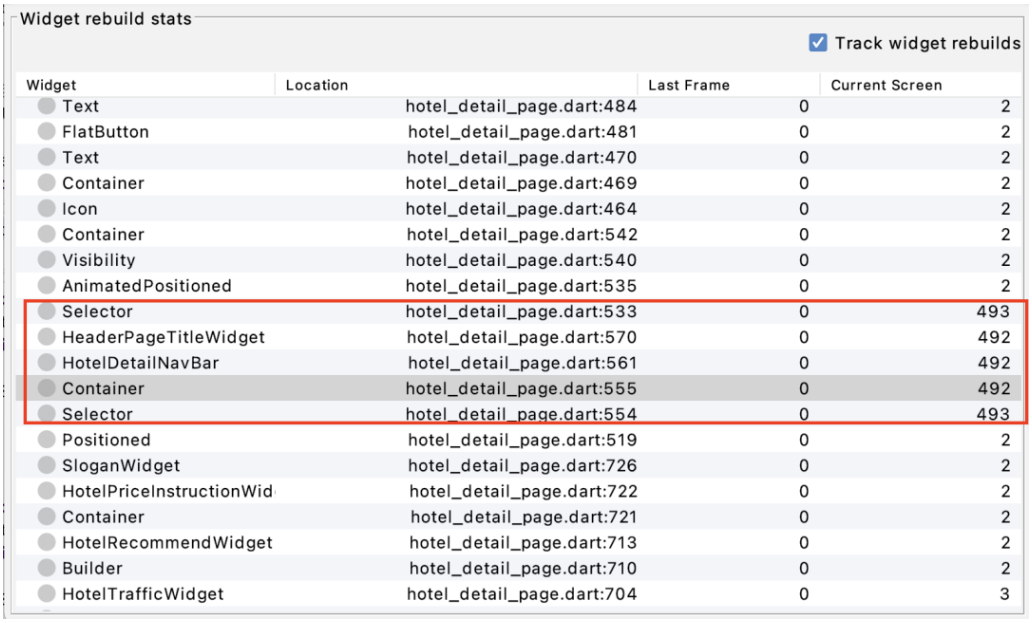
```
void addScrollListenerForTopTitle(BuildContext context) {
  var tabViewModel = Provider.of<TopTabStatusViewModel>(context, listen: false);
  /// 动画距离
  int scrollHeight = 120;
  _scrollController.addListener(() {
    ///根据滚动距离来设置顶部 titleBar 的透明度
    if (_scrollController.offset > scrollHeight && tabViewModel.titleAlpha != 255) {
```

```
            tabViewModel.titleAlpha = 255;
        }
        if (_scrollController.offset <= 2 && tabViewModel.titleAlpha != 0) {
            tabViewModel.titleAlpha = 0;
        }

        if (_scrollController.offset > 0 && _scrollController.offset < scrollHeight) {
            tabViewModel.titleAlpha = _scrollController.offset * 255 ~/ scrollHeight;
        }
    });
}
```

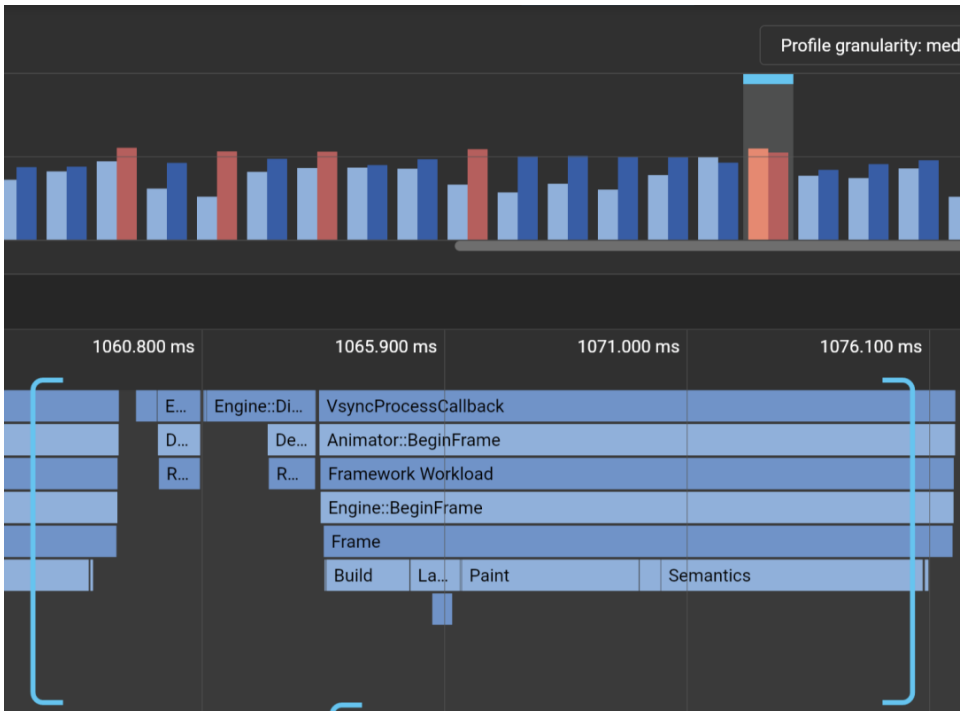
透明度渐变组件：

```
Selector<TopTabStatusViewModel, int>(builder: (context, alpha, child) {
    return Container(
        color: Colors.white.withAlpha(tabViewModel.titleAlpha),
        child: Column(
            children: [
                HotelDetailNavBar(tabViewModel.titleAlpha, widget.pageDeliverData, hotelDetail),
            ],
        ),
    );
}, selector: (context , viewModel) => viewModel.titleAlpha);
```

改造之后，可以看到，当界面滑动的时候，只重新渲染了需要改变透明度的组件，组件重建状态如下图所示：



火焰图如下所示：



这样很大程度的减小了组件的重建范围，每次都只是按需加载，build 层级明显减少，总耗时也明显降低。因此在界面渲染的时候，应尽量降低 Widget Tree 遍历的出发点，合理控制重建范围。

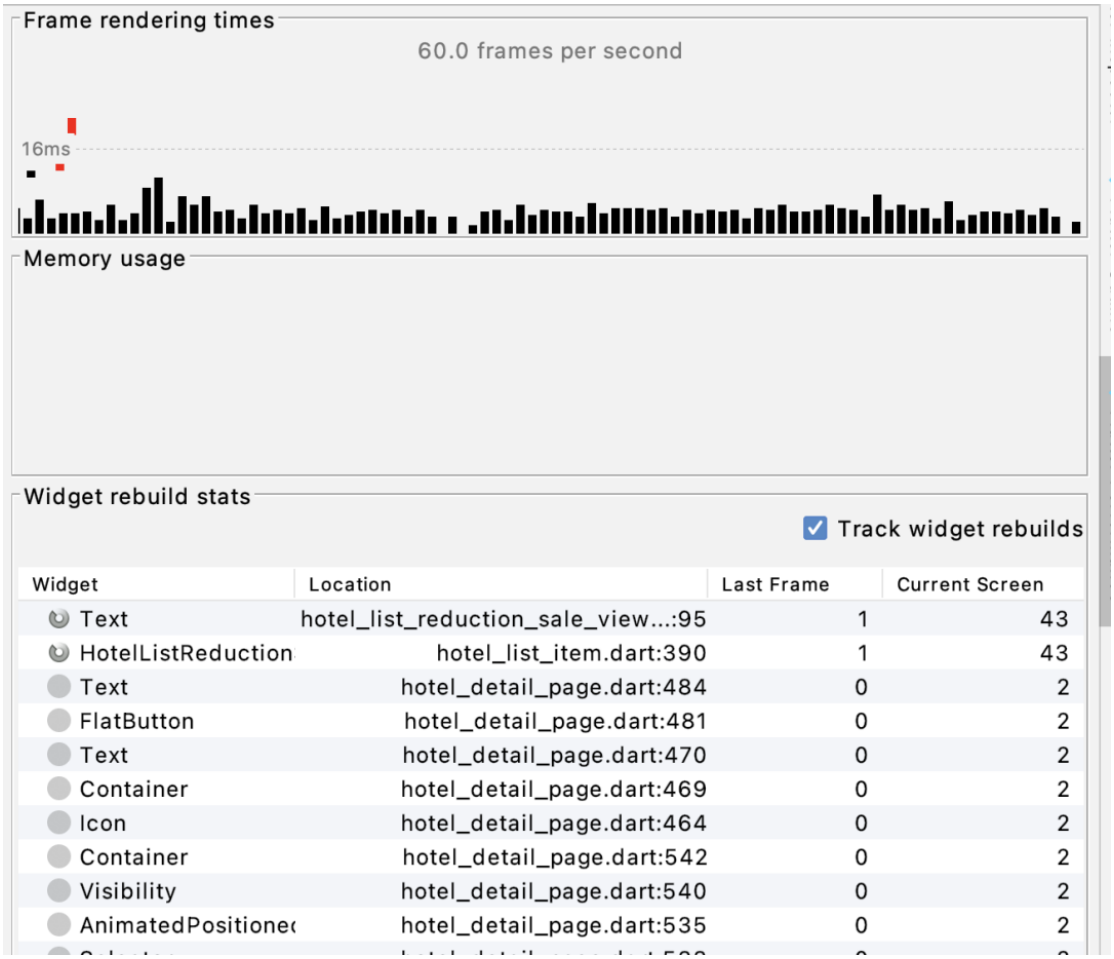
2.2 setState 降低刷新颗粒度

如图所示，有一个动态的轮播效果，需要每间隔 2s 进行轮播一次，实现的方式是使用一个 Timer，每间隔 2s 进行 setState 一下文字，以实现轮播的效果。

但是发现这个时候，这整个 View 都会被重绘，导致了巨大的开销，造成不必要的渲染，当前需求只是修改一个文字，没有必要整棵 Widget 树都去重新载入。这里需要考虑到没有合理控制刷新的范围。改进策略是将这个具有轮播效果的组件进行独立封装，以同样的方式去实现轮播效果；

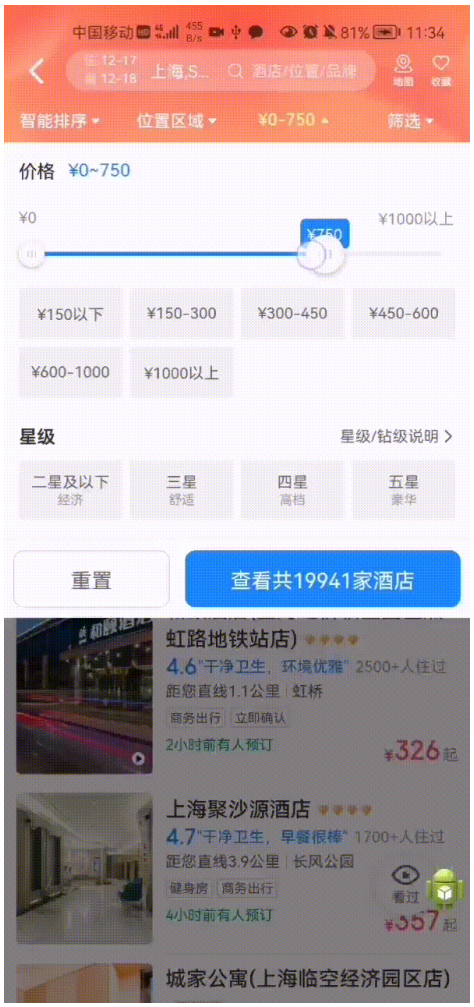
```
Widget build(BuildContext context) {
   ///使用 Timer 每间隔 2s 去修改 texts 的值
   return Container(
      alignment: Alignment.center,
      child: Text(this.texts),
   );
}
```

这样每次渲染的 Widget 就只有文本这个组件本身，如下图所示：



2.3 减少组件重绘的次数

开发过程中，很容易触发界面的重新渲染，大多数时候都是没有控制好组件的刷新次数，这样很容易导致内存消耗过大，或多次无效的网络加载，导致界面在滑动的时候出现卡顿，用户体验差等问题。如下图所示，借助 flutter_xlider 三方组件实现区间选择效果：



在 onDragCompleted 回调方法中处理界面及数据刷新，代码如下：

```
Widget rangeSliderView() {
  return FlutterSlider(
    values: [0, 1000],
    onDragCompleted: (handlerIndex, lowerValue, upperValue) {
      if (mounted) {
        setState(() {
          startSortPrice = lowerValue;
          endSortPrice = upperValue;
        });
      }
      /// 更新价格区间并刷新数据
      refreshPriceText(lowerValue, upperValue);
    },
  );
}
```

如上图，这里存在一个问题，再次选同样的价格区间，也会触发界面和数据刷新，是完全无效的刷线操作。这里改进策略是添加条件限制避免重复的无效刷新。优化代码如下：

```
Widget rangeSliderView() {
  return FlutterSlider(
    values: [0, 1000],
    onDragCompleted: (handlerIndex, lowerValue, upperValue) {
      if(lowerValue != startSortPrice || upperValue != endSortPrice) {
        if (mounted) {
          setState(() {
            startSortPrice = lowerValue;
            endSortPrice = upperValue;
          });
        }
        /// 更新价格区间并刷新数据
        refreshPriceText(lowerValue, upperValue);
      }
    },
  );
}
```

2.4 拆分 ViewModel 降低界面刷新几率

在开发 Flutter 的过程中，很多时候不会千篇一律的都使用 setState 去控制一个界面的状态，因为这样会使得界面过于零碎且难以控制。这时可以使用 Provider 进行管理界面的状态，使得界面的状态集中管理且界面渲染都在可控范围之内。

将存放状态的对象叫做 ViewModel，针对一个大的界面，数据可能有多个来源，如果将所有的数据及状态值都存放在一个 ViewModel 中，就会使得 ViewModel 过于冗余，当 ViewModel 中的数据发生变化时，可能会导致整个界面被触发重新渲染，这个显然是不合适的。因此可以将 ViewModel 进行拆分，尽量使得一个 ViewModel 只管理一个 View，将 ViewModel 与 View 进行绑定，然后使用 MultiProvider，将所有的 Provider 统一存放在界面的入口处：

```
MultiProvider(
  providers: [
    ChangeNotifierProvider(
      create: (context) => CalendarSelectorViewModel(),
    ),
    ChangeNotifierProvider(
      create: (context) => TopTabStatusViewModel(),
    ),
  ],
  child: HotelDetailPageful(scriptDataEntity),
);
```

一个 ViewModel 只对应界面中的一个 UI，也就是说当数据变化的时候，只会控制对应的 View 进行刷新，而不会刷新无关的 View，从而降低无关 View 的刷新频率。

2.5 缓存高层级组件

复杂页面，页面级的每个模块都是独立的组件，每次刷新页面把所有的子组件都重新渲染一遍，性能开销非常大。尽量复用，避免不必要的视图创建。List<Weight> 缓存高层级组件。

```
///存放界面所有的 widgets，用以缓存
List<Widget> widgets = new List<Widget>();
///因为头部布局是静态的不刷新，使用变量控制是否复用以前的 widgets
var refreshPage = true;
///获取界面布局所有的 widgets
List<Widget> getPageWidgets(ScriptDataEntity data) {
if(widgets.isNotEmpty && !refreshPage) {
    return widgets;
  }
}
```

2.6 const 标识

当调用 setState()，Flutter 会 Rebuild 当前 View 中的每一个子组件，避免全部重新构建的方法就是用 const; 特别是在一些有动画效果的组件上，更应该用 const 修饰避免频繁构造。同时使用 const 修饰还能减少垃圾回收。

2.7 RepaintBinary 隔离

对于一些经常需要变动渲染的组件，比如 Swiper、PageView、Lottie 等，可以使用 RepaintBoundary 进行隔离。RepaintBoundary 就是重绘的边界，用户重绘时独立于父布局。因为它会为经常发生显示变化的内容提供一个新的 layer，新的 layer paint 不会影响到其他的 layer。

```
RepaintBoundary(
  child: Container(
    child: Lottie.network(
      InlandPicture.otaLottieJson,
    ),
  ),
)
```

2.8 尽量避免使用 ClipPath 组件

在开发过程中应尽量避免使用 ClipPath，裁剪 path 是一个很昂贵的操作，在绘制小部件的时候，ClipPath 会影响每个绘图指令，做相交操作，之外的部分裁剪掉，因此这是一个耗时

操作。如果只是想裁剪圆角之类的组件，还是推荐使用 Container 的 raidus 进行去设置。

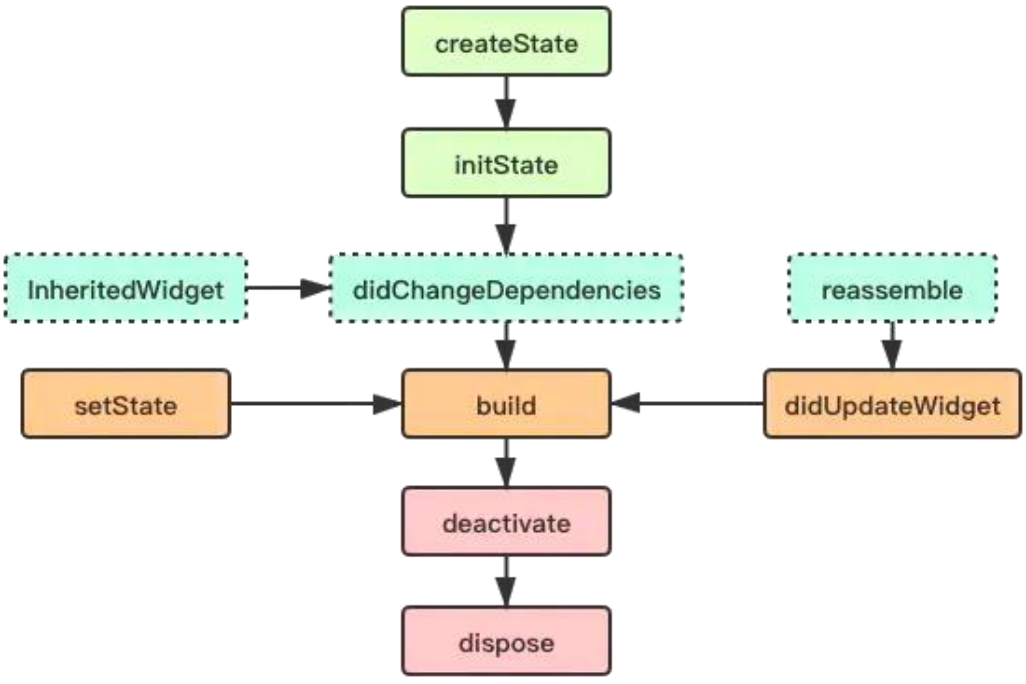
2.9 减少使用 Opacity 类型组件

减少 Opacity Widget 的使用，尤其是在动画中，因为它会导致 widget 的每一帧都会被重建，可以用 AnimatedOpacity 或者 FadeInImage 进行代替。

```
AnimatedOpacity(
      opacity: showHeader ? 1.0 : 0.0,
      duration: Duration(milliseconds: 200),
      child: Container(
            color: SmartColor.d_FFFFFF,
            padding: EdgeInsets.fromLTRB(6, 0, 6, 0),
            child:  SmartTrainHeader(showHoverHeader:  showHoverHeader,handlerCallBack:
widget.handler)),
  )
```

三、Root Isoate 优化

3.1 减少 build 中逻辑处理



尽量减少 build 中处理逻辑，因为 widget 在页面刷新的过程中会随时通过 build 重建，build 调用频繁，应该只处理跟 UI 相关的逻辑，因此将一些不涉及每次渲染都必须的操作，存放在 initState 中，或者使用变量进行状态判断，避免每次界面元素刷新触发 build 重绘时都需

要大量重复切不必要的计算，从而降低 CPU 的消耗。

3.2 耗时计算放到 Isolate 去执行（多线程）

针对 UI 线程存在的一些耗时操作，可以使用 Isolate 以”多线程“的方式去执行。

Isolate 本质更接近于操作系统中的”进程“概念，Dart 中不存在共享内存的并发机制，由于不用担心线程抢占的问题因此也不会造成死锁，Isolate 是没有共享内存的，这是跟常见的其它多线程语言区别较大的地方。

创建一个线程会增加 2MB 左右的内存，尽可能还是避免滥用导致内存开销。

酒店详情页的头部 header，跟随页面的滚动需要实时的计算当前的透明度，滑动到最顶部的时候全透明显示，滑动出头部图片显示区域的时候则完全显示出来，并且在界面滑动的过

程中需要监听每个对应模块滑动的偏移量，以修改顶部悬浮 Tab 的状态；因此使用 isolate 将滑动实时计算透明度及偏移量的逻辑进行隔离操作，计算成功后将结果返回。这样就不会影响到 UI 主线程滚动页面的操作，可以提升页面的流畅性。

四、长列表滑动性能优化

4.1 ListView Item 复用

通过 GlobalKey 可以得到 widget，包括获得组件的 renderBox 在内的各种 element 有关的信息，可以得到 state 里面的变量。在长列表分页加载时，数据变更会造成整个 ListView 重现构建，我们就可以利用 globalkey 获得 widget 的属性，来实现 Item 复用。从而解决分页加载成功后大量渲染引造成的页面卡顿问题。

```
Widget listItem(int index, dynamic model) {
  if (listViewModel!.listItemKeys[index] == null) {
    listViewModel!.listItemKeys[index] =RectGetter.createGlobalKey();
  } else {
      final rectGetter = listViewModel!.listItemKeys[index];
      if (rectGetter is GlobalKey) {
        final widget = rectGetter.currentWidget as RectGetter?;
        if (widget != null) {
          return widget;
        }
      }
  }
```

使用 GlobalKey 不应该在每次 build 的时候重建 GlobalKey，它应该是 State 拥有的长期存在的对象。

4.2 首页预加载

为了减少等待时间，能让用户进入列表页就能看到内容，在上个页面预加载列表的数据。预加载数据有几种情况，已加载成功直接带入加载数据结果，“在途请求”通过桥方法重新获取数据。代码如下：

```
_loadHotels() {
  if (isFirstLoad && page == 1) {
    // response 首页携带已请求完毕的数据
    if (response != null) {
      // 处理展示列表页数据
      return;
      // 数据还在请求当中
    } else if (isPreloading) {
      // 首页数据加载完毕后回调，处理展示列表页数据
```

```
            return;
        }
    }
    // 正常加载数据
}
```

4.3 分页预加载

通常情况下当用户滑动到底部的时候才会去加载下一页的数据，这样用户要花费等待加载的时间，影响用户体验。可以采用剩余法预加载数据，当用户滑动到剩余一定数量的酒店时，开始加载下一页的数据，在网络良好的情况下，滑动场列表界面，界面基本不会存在等待加载的时间。

```
// getRectFromKey 获取到 scrollView 的位置信息，遍历指定剩余数量的 item，如果在当前屏幕中去加载一下页数据
if (!(itemRect.top > rect.bottom || itemRect.bottom < rect.top)) {
        // 加载下一页数据
}
Rect? getRectFromKey(GlobalKey key) {
    final renderObject = key.currentContext?.findRenderObject();
    final translation = renderObject?.getTransformTo(null).getTranslation();
    final size = renderObject?.semanticBounds.size;
    if (translation != null && size != null) {
        return Rect.fromLTWH(translation.x, translation.y, size.width, size.height);
    }
    return null;
}
```

4.4 取消在途网络请求

频繁做一些筛选等操作会在短时间内多次请求网络，如果网络较差或者服务端返回时间过长，会导致数据展示错乱的问题，在刷新列表时要取消掉还未返回数据的请求。

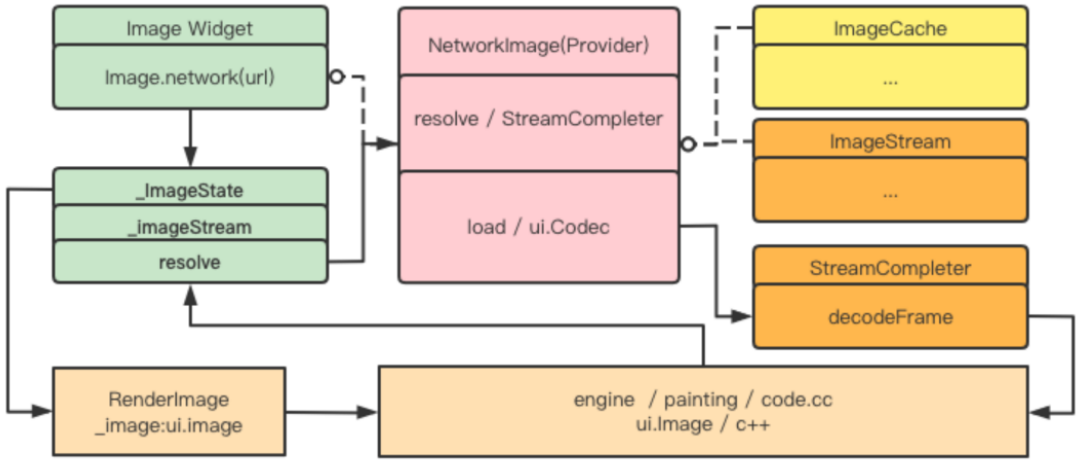
```
_loadHotels() {
        if (isRefresh) {
            // 通过标识符取消请求
            cancelRequest(identifier);
        }
        identifier = 'QUERY_IDENTIFIER' + '时间戳';
        // 列表数据请求
}
```

五、图片渲染性能和内存开销治理

图片加载是 APP 最常见也最基本的功能，也是影响用户体验的重要因素之一。在看似简单的图片加载背后却隐藏着很多技术细节，在接下来的章节，将主要介绍 Flutter 图片加载上做的一些优化尝试。

5.1 图片加载原理

以 NetworkImage 为例，我们看一下 Flutter 中图片的加载过程，首先通过 ImageProvider 的 resolve 获取相应的图片资源，得到 ImageStream，通过底层进行解码，并生成纹理。ImageState 接收到纹理对象绘制图片，上层获取图片纹理后会调用 ImageState 的 SetState 方法将纹理对象传给底层 Render object，排版完成后图片就会绘制到屏幕。当上层 Image Widget 被销毁，Image Cache 清空时，触发底层纹理的释放。



5.2 图片加载治理

在业务开发中，我们总希望页面内容可以尽可能快的展示给用户，给用户“直出”的用户体验。在酒店列表和详情页面中，都有较多的酒店和房型的图片，图片多，导致内存占用高，加载耗时，影响用户体验。

5.3 图片预加载

数据预加载：如果使用的图片资源是一些异步获取的数据，可以考虑是不是可以提前获取相关的数据，在要使用的时候，再拿过来使用。利用空闲资源，提前获取加载所需关键数据。

图片预加载机制：precacheImage，在合适的时机提前使用 precacheImage 对需要展示的图片数据进行预加载到内存中，这样在真正展示的时候，图片已经被加载到内存了，就可以在内容加载时达到“直出”的效果。

延时加载：在很多场景中，如酒店列表，酒店详情头部轮播图，第一次只需要加载首屏内的数据，就可以对非首屏的数据进行延迟加载，避免加载瞬时资源竞争，优先保证重要资源的加载，实现良好的加载体验。

5.4 图片资源优化

图片资源处理，图片压缩，图片格式建议优先使用 webp 格式，Flutter 中原生支持 webp 图片格式。

CDN 优化是另一个非常重要的方面，主要是在资源层面，最小化传输图片大小，最快响应图片请求，最优化图片选择，支持网络图片大小裁剪，根据实际的需要，加载对应的图片，比如大的头图和小的缩略图，根据具体的场景，加载裁剪之后的不同的图片资源。

5.5 图片内存优化

经过预加载和资源优化，已经可以比较流畅的加载相关业务了，但是过多的数据加载到内存，又会导致内存占用过高，怎么合理高效的利用内存就成为了接下来要解决的问题，一方面，Flutter 图片管理能力较弱，缺乏本地存储能力；另一方面，在混合 APP 开发时，因为前面说的缓存不同，图片的重复下载，很容易造成内存过高，从而发生 OOM（OutOfMemory）情况。在梳理 Flutter 原生图片方案之后，为了更稳定流畅的体验，是不是有机会在某个环节将 Flutter 图片和 Native 以原生的方式打通。

共享内存：打通 Native 内存数据，保证同样的数据在内存中只保留一份，避免重复加载造成的内存开销。使用磁盘缓存，这样既可以增大缓存的数据量，同时通过磁盘，Native 和 Flutter 又可以共享一份数据，极大的减少了内存占用，保证了内存平稳运行。

图片加载：Flutter 的图片加载有两种方式：一是默认方式不指定 cacheWidth/cacheHeight，最终图片的加载使用的是原图分辨率，这就可能导致内存使用过大出现内存泄漏的情况；二是指定 cacheWidth/cacheHeight，以此限制图片的加载分辨率，同时图片的 key 也会受此影响，即同一源的图片多次不同分辨率加载会多次占用内存，这既不方便也没有节约到内存。

因此针对以上情况，图片的内存缓存的命中和 width/height、cacheWidth/cacheHeight 等参数相关，这样从分根据图片的参数来设置缓存数据，更有效的保证缓存的真实有效性。在使用缓存时，发现一个问题，就是图片容易模糊，变形。比如在加载一个高清大图时，采样比例无法单纯的根据页面 widget 的宽高来计算，设置太小会模糊，设置大了，又不利于节省缓存。

```
Image.network("<image-source-uri>",
  width:100, height:100,
  completeCallback:(Object? key, bool success, {ImageUsageInfo? imageUsageInfo}) => {
  // success，代表图片是否加载成功，成功为true，失败为false。
  print("ImageUsageInfo数据详解：" +
    "key:$key" +                                    //图片原始信息，比如Image.network包含source、scale、headers等等
    "fileWidth:${imageUsageInfo?.fileWidth}" +      //图片原始尺寸宽度
    "fileHeight:${imageUsageInfo?.fileHeight}" +    //图片原始尺寸高度
    "fileSize:${imageUsageInfo?.fileSize}" +        //图片物理文件大小（bytes）
    "cacheWidth:${imageUsageInfo?.cacheWidth}" +    //图片使用尺寸宽度
    "cacheHeight:${imageUsageInfo?.cacheHeight}" +  //图片使用尺寸高度
    "memorySize:${imageUsageInfo?.cacheSize}"       //图片内存占用大小（bytes）
  )});
```

六、总结

本文介绍了遇到 Flutter 页面渲染问题，结合 Performance Overlay 性能分析工具来确定是 UI 线程的性能问题，还是 GPU 线程的性能问题。UI 线程的性能问题可以通过火焰图来具体分析是哪个方法造成的。GPU 的线程问题可以通过查看渲染的次数，渲染的范围来确定。下面是我们常用的一些性能优化的方法：

（1）UI 线程优化

- 拆分 VieModel 降低刷新几率
- Provider 监听数据推荐使用 Selector
- 减少在 build 中做耗时操作，放到 Isolate 去执行
- 缓存高层级组件
- 控制刷新范围、频次
- setState 刷新颗粒度在最低层
- const 修饰避免频繁构造

（2）GPU 线程优化

- 使用 RepaintBinary 隔离 提别是轮播广告、动画
- 减少 ClipPath 的使用,简单圆角采用 BoxDecoration 实现
- 避免 Opacity,可以通过切图实现。有动画效果的建议用 AnimatedOpacity
- 避免使用带换行符的长文本

同时也介绍了 Flutter 在长列表、图片加载上的一些体验优化措施，希望能在你做 Flutter 性能优化和用户体验时有一些帮助。

携程酒店 Flutter 性能优化实践

【作者简介】Qifan，携程高级工程师，专注移动端开发；Yinuo，携程高级工程师，专注移动端开发；popeye，携程软件技术专家，关注移动端跨端技术，致力于快速，高性能地支撑业务开发。

一 、前言

携程酒店业务使用 Flutter 技术开发的时间快接近两年，这期间有列表页、详情页、相册页等页面使用了 Flutter 技术栈进行了跨平台整合，大大提高了研发效率。在开发过程中，也遇到了一些性能相关问题和用户反馈，比如长列表滚动卡顿、页面打开时间较长、页面打开后部分数据加载时间较长等问题。为解决这些问题，我们选用了多个性能指标监控业务运行状态，借助性能检测工具定位问题，并查阅源码、文档等资源解决问题，形成了这篇文章。

同时在不断的需求迭代和代码更新过程中，APP 的性能稳定性持续受到挑战，为此我们建立了线上性能监控系统，通过量化，治理，监控三方面手段，持续改善 APP 性能和用户体验。目前页面的各种性能指标诸如 FPS、TTI、内存等都达到了不错的效果，本文将介绍我们在优化过程中所遇到的问题和采取的主要优化方案。

二、FPS&TTI 提升性能优化

2.1 常用性能指标和卡顿定义

对于客户端应用来说，流畅度是影响用户使用体验的关键因素。流畅度低主要有：低 FPS、高 TTI、卡顿。这些现象出现时，页面会出现不连续的动画，页面刷新会短暂停顿，打开新页面速度较慢，新页面出现白屏或者较长时间的加载动画，用户做点击滑动等交互时页面不响应。

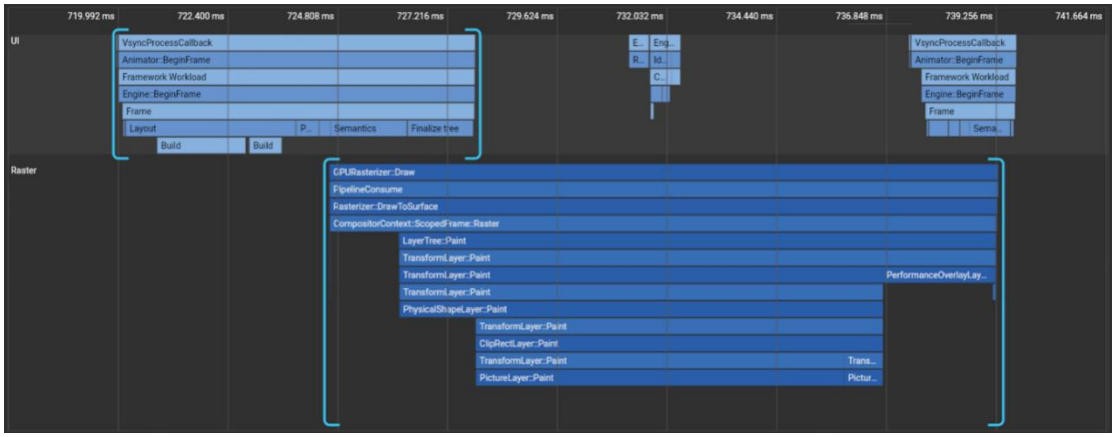
用户操作 FPS 的定义是每秒传输帧数 (Frames Per Second)，是图像领域的概念。对于手机客户端来说，主流显示屏的刷新率为 60Hz，高端手机显示屏刷新率可以达到 120Hz 及以上。理想情况下，页面绘制的 FPS 和屏幕刷新率一致。屏幕画面刷新次数越多，屏幕可以展示的动态细节越多，所以数值越高越好。TTI 的定义是从页面加载开始到页面处于完全可交互状态 (Time To Interactive)，完全可交互状态指的是页面有内容呈现并且用户可以进行操作。

2.2 FPS 优化的工具介绍

Flutter 官方提供了三种应用编译选项，debug 模式、release 模式和 profile 模式。当我们需要做性能分析的时候，需要打包 profile 模式的应用，这个模式的性能接近 release 模式，并且有性能相关的信息分析。我们使用的工具是官方提供开发者工具中的 Performance View，并选择了 Enhance tracing 模式。

上图是帧渲染时间，横坐标是帧号，纵坐标是绘制时间，蓝色代表该帧满足 60fps，橙色代表不满足 60fps。从这张图可以快速定位到绘制时间较长的帧，而下图是选中某帧之后，UI 绘制和光栅化时间，如果选择了 Enhance tracing 模式，可以看到耗时较长的方法、widget build。

前文已经介绍过 FPS 的定义，对于 flutter 绘制而言，每帧绘制耗时前三的是 UI 绘制时间、光栅化时间、vsync ahead。UI 绘制时间主要是 widget build、layout、paint，简单认为是 CPU 时间；光栅化时间可以简单认为是 GPU 时间；vsync ahead 是 vsync 信号与 widget build 之间的延时。



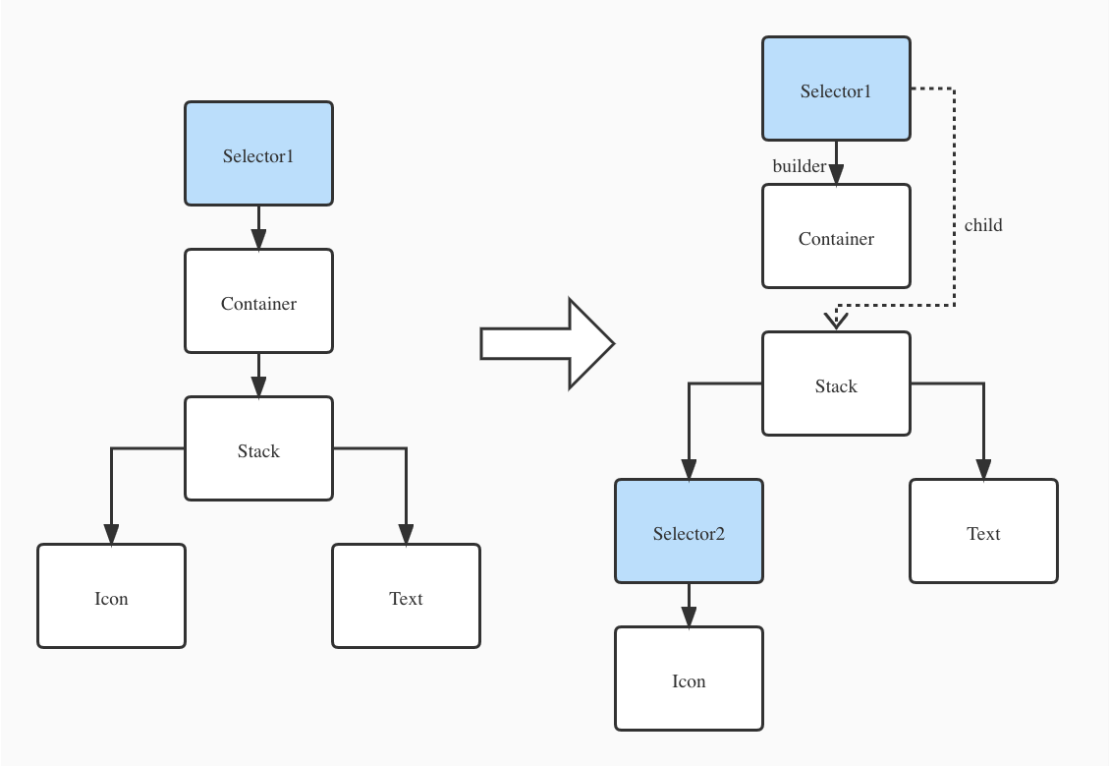
2.3 具体实践方案

（1） 控制 setState 次数，使用 Provider 机制减小刷新范围

我们的业务开发是 MVVM 结构的，数据驱动 UI 更新。UI 的绘制占了性能开销的很大部分，减少不必要的 UI 绘制、控制 UI 绘制的范围这两种方法能显著改善性能。

减少不必要的 UI 绘制是通过控制 build 次数实现的。widget build 是通过 setState 方法或者 builder 方法触发的，在业务中，尽量减少非必要的 setState，只有真正页面数据发生变化，页面状态变化时才调用 setState 方法。对于 builder 方法，可以实现 shouldRebuild 等接口，增加触发 builder 方法的限制。

控制 UI 绘制的范围是通过改变 widget 树层级实现的。MVVM 中数据触发 UI 更新的方式有很多，我们的业务主要用到了 Provider 机制，这是一种观察者模式设计。如下图所示，对于左边的 widget 树，如果只需要更新 Container 容器配置和 Icon 图标配置，那么可以将 selector 拆分到这两个 widget 的双亲 widget，实现了 Text widget 不刷新。对于 widget 树较大的业务，这样的改动能显著提升 FPS。



（2）预构建 widget (AnimatedBuilder)

上图是酒店详情页头部沉浸式动画的 UI，头部展开的过程中，图片和图片上的蒙层需要重

新绘制，图片上部 SHA logo 不需要重新绘制，图片下部 tab 栏不需要重新绘制，对于这个需求的做法是用 AnimatedBuilder。

AnimatedBuilder 提供了几个可选参数，animation 是对动画的监听，builder 是动画过程中需要重新绘制的部分，child 是动画过程中不需要重新绘制的部分，child 作为参数会传入 builder 中。下面的伪代码是一个例子，动画过程中 Text 并不会多次绘制。

```
@override
    Widget build(BuildContext context) {
      return AnimatedBuilder(
        animation: _controller,
        child: Container(
          width: 200.0,
          height: 200.0,
          color: Colors.green,
          child: const Center(
            child: Text('Text!'),
          ),
        ),
        builder: (BuildContext context, Widget child) {
          return Transform.rotate(
            angle: _controller.value * 2.0 * math.pi,
            child: child,
          );
        },
      );
    }
```

对于详情页头部沉浸式动画的例子，可以把 widget 树进行拆分，只有图片和图片蒙层放入 builder 方法中，其余的 widget 作为 child 传入 builder，同时用 Stack widget 实现两部分 UI 的组合，这样改进之后，FPS 在动画过程中有较大提升。

（3）const widget

对于 dart 语法，需要分清楚 final 和 const 关键字的区别。关键字 final 的意思是一次赋值，不能改变；而关键字 const 的意思是常量，确定的值。这两者的区别是 final 变量在第一次使用时被初始化，而 const 变量是一个编译时替换为常量值。同样的，对于 const widget，这个 widget 在编译阶段就已经确定，不会有状态的变化和成员变量更新。const widget 特别适合于标签、特殊 Icon 等可以复用的 UI，性能开销较小。

（4）减少耗时计算，放到 Isolate

Flutter 应用中的 Dart 代码执行在 UI Runner 中，而 Dart 是单线程的，我们平时使用的异步任务 Future 都是在这个单线程的 Event Queue 之中，通过 Event Loop 来按顺序执行。需要

避免将一些耗时计算放在 UI 线程，可以把耗时计算放到 Isolate 去执行。

（5）懒加载

能够实现懒加载的有 ListView.builder、PageView.builder 和 GridView.builder，这些 widget 可以用户长列表或重复容器结构的 UI，通过判断单个 item 是否在屏幕内或者将要进入屏幕位置而进行绘制。与之对应的是 Column、Row 等一次性绘制 widget，对于重复结构的数据，尽量避免使用这些组件。

如下图中，酒店周边景点美食购物列表和附近同类型酒店列表都实现了按需加载。酒店周边景点美食购物列表的卡片数量超过 20 个，最初使用 Row 组件构建时，第一次构建时间超过 25ms，达不到 60FPS 的 16ms 绘制时间要求。当然，按需加载也有性能开销，出现在列表的滑动过程中。如果一次性全部构建了列表，滑动过程中不会触发新的构建，滑动流畅度体验更好，但是第一次构建时的卡顿感明显。

（6）分帧渲染

错峰加载方案使用分帧渲染，分帧渲染的原理是将一棵 Widget 树中的部分绘制时间较长的节点在第一帧时只占位不绘制，等到下一帧开始时，节点替换占位 UI，单独使用一帧时间绘制。

在酒店详情头部信息绘制中运用了分帧渲染技术，下左图未使用分帧渲染，下右图对图片 tab 栏、酒店设施标签、点评模块、地址栏使用分帧渲染。从结果看，减少了 3 次卡顿和 1 次轻微卡顿，流畅帧占比超过 90%。

布局与绘制的基本单位是一棵 widget 树，分帧渲染的原理是将布局与绘制时间较长的子 widget 先用 Container 占位，再等下一帧开始时单独渲染。使用占位 widget 的伪代码如下，build 方法返回占位 widget，并在 widget 构建帧结束时替换占位 widget 并触发绘制。

```
@override
  void initState() {
     super.initState();
     result = widget.placeHolder;
     replaceWidget ();
  }

  @override
  Widget build(BuildContext context) {
     return result;
  }

  void replaceWidget() {
     SchedulerBinding.instance.addPostFrameCallback((t) {
        TaskQueue.instance.scheduleTask(() {
           if (mounted)
              setState(() {
```

```
                result = widget.child;
            });
        }, Priority.animation, id: widget.index);
    });
  }
```

帧的绘制状态可以从 SchedulerBinding 获得，同时建立队列保证一帧执行一个子 widget 绘制。

```
// 等待当前帧结束时替换占位 widget 并触发绘制
  await SchedulerBinding.instance.endOfFrame;

  // 执行任务队列中的绘制任务
  final TaskEntry<dynamic> entry = _taskQueue.first;
  entry.run();
```

2.4 UI GPU 问题定位与优化

GPU 问题主要集中在底层渲染耗时上。有时候 Widget 树虽然构造起来容易，但在 GPU 线程下的渲染却很耗时。涉及 Widget 裁剪、蒙层这类多视图叠加渲染，或是由于缺少缓存导致静态图像的反复绘制，都会明显拖慢 GPU 的渲染速度。可以使用性能图层提供的两项参数，负责检查多视图叠加的视图渲染开关 checkerboardOffscreenLayers 和负责检查缓存的图像开关 checkerboardRasterCacheImages 来检查这种模块的存在。

（1） checkerboardOffscreenLayers

多视图叠加通常会用到 Canvas 里的 savaLayer 方法，这个方法在实现一些特定的效果（比如半透明）时非常有用，但由于其底层实现会在 GPU 渲染上涉及多图层的反复绘制，因此会带来较大的性能问题。对于 saveLayer 方法使用情况的检查，我们只要在 MaterialApp 的初始化方法中，将 checkerboardOffscreenLayers 开关设置为 true，分析工具就会自动帮我们检测多视图叠加的情况了，使用了 saveLayer 的 Widget 会自动显示为棋盘格式，并随着页面刷新而闪烁。

不过，saveLayer 是一个较为底层的绘制方法，因此我们一般不会直接使用它，而是会通过一些功能性 Widget，在涉及需要剪切或半透明蒙层的场景中间接地使用。所以一旦遇到这种情况，我们需要思考一下是否一定要这么做，能不能通过其他方式来实现。如下图所示，因为详情头部 bar 用到高斯模糊，同时使用 ClipRRect 裁切圆角，ClipRRect 会调到 savelayer 接口，所以该部分产生闪烁。

（2）checkerboardRasterCacheImages

从资源的角度看，另一类非常消耗性能的操作是，渲染图像。这是因为图像的渲染涉及 I/O、GPU 存储，以及不同通道的数据格式转换，因此渲染过程的构建需要消耗大量资源。

为了缓解 GPU 的压力，Flutter 提供了多层次的缓存快照，这样 Widget 重建时就无需重新绘制静态图像了。与检查多视图叠加渲染的 checkerboardOffscreenLayers 参数类似，Flutter 也提供了检查缓存图像的开关 checkerboardRasterCacheImages，来检测在界面重绘时频繁闪烁的图像（即没有静态缓存）。

我们可以把需要静态缓存的图像加到 RepaintBoundary 中，RepaintBoundary 可以确定 Widget 树的重绘边界，如果图像足够复杂，Flutter 引擎会自动将其缓存，避免重复刷新。当然，因为缓存资源有限，如果引擎认为图像不够复杂，也可能会忽 RepaintBoundary。

2.5 页面预加载提升 TTI

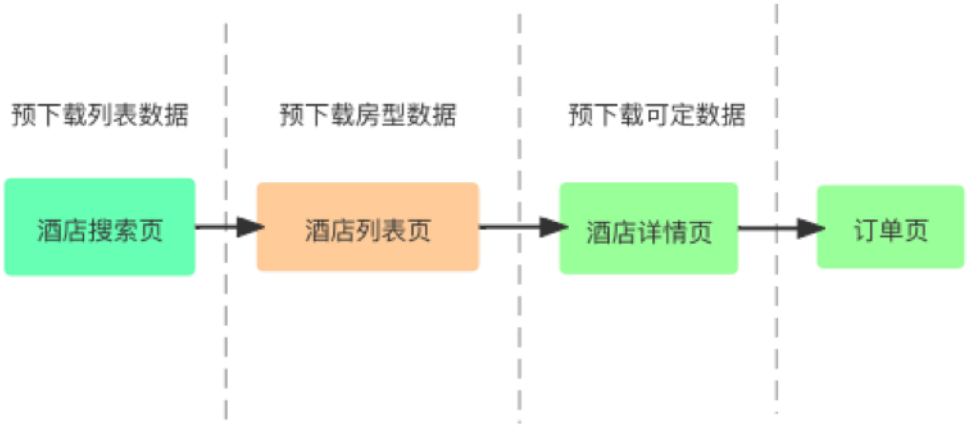
网页应用的主要流程有三步，通过链接打开页面，发送服务请求获得页面数据，将页面数据展示在页面上。对客户端应用来说，页面之间跳转是相对确定的，数据在页面之间存在共享的可能，预加载的工作是在打开页面之间预先获得页面的数据，从而减少打开页面到页面展示的时间。

预加载数据有三种常见方法，第二个页面的数据在第一个页面的服务结果中获得；第二个页面的数据在客户端其它页面中预先获得并缓存；第二个页面的服务请求在打开页面之前发送。

（1）预加载页面数据

页面数据预获取的方案，实现方法是在上一个页面提前获取服务数据，在用户跳转到当前页面时，直接从缓存获取，节省了数据的网络传输时间，达到快速展示当前页面内容的效果。

目前在酒店核心预订流程，都运用了数据预加载技术，如下图所示。



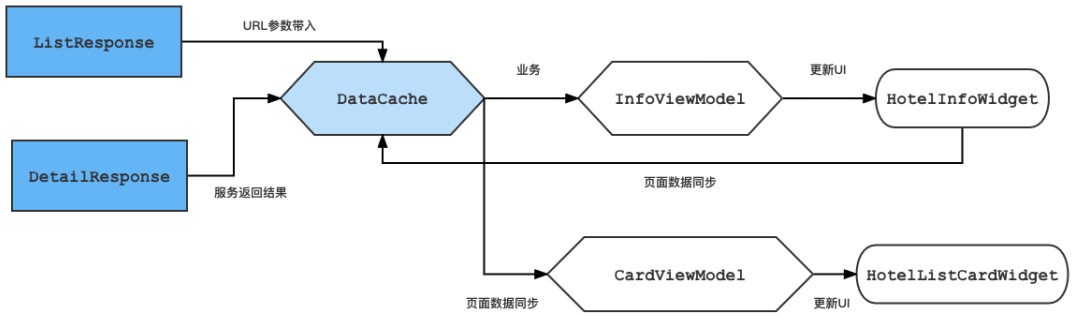
结合酒店业务特点，数据预加载需要考虑几个方面问题，第一，酒店预订流程页面 PV 量都很高，酒店列表和详情页 PV 都是千万级别，所以需要考虑数据预加载的时机，避免服务的资源浪费。第二，酒店列表，详情，填单页都有价格信息，价格信息对用户来说是动态信息，实时都有变价可能，所以需要考虑数据预加载的缓存策略，避免因为价格的前后不一致造成用户误解。

在实现全流程预加载方案之后，我们酒店预订流程页面的慢加载率从初始值的 42.90%降低至现阶段的 8.05%。

（2）预加载 ViewModel

与数据预获取的方案相比，预加载 ViewModel 更进一步，将预获取的数据处理成 ViewModel 形式，在打开页面时直接用 ViewModel 进行展示。这种方案减少了业务对数据处理的时间。

上图是杭州绿城尊蓝钱江豪华精选酒店在酒店列表页和酒店详情页头部的 UI 对比。可以看出，酒店详情页头部的信息主要是酒店名称、星级、榜单、特色设施、点评、开业装修时间等信息，这些信息和列表页酒店卡片信息存在重合。如果用户浏览的轨迹为从酒店列表页到酒店详情页，那么可以直接将列表页的数据带入酒店详情页作为头部展示。



上图为详情页头部预加载的主要流程。我们的 flutter 业务代码采用 MVVM 的结构，将服务请求的结果处理完的数据放入 ViewModel 中，ViewModel 的数据更新通过 Provider 机制触发页面 UI 更新。

图中可以开到，详情页头部 ViewModel 的数据有两个来源，分别是列表页服务请求的结果和详情页服务请求的结果。这两个服务请求结果到 ViewModel 的业务流程不一样，列表页的服务结果数据通过 URL 参数的方式传入详情页，而详情页服务结果可以直接生成详情页头部的 ViewModel。

图中还有一个重要模块是列表页服务结果和详情页服务结果之间的通用缓存 DataCache，它的功能是实现页面之间数据的一致性。页面上的数据可以由服务更新，也可以由用户交互更新。业务的 ViewModel 依赖这个通用缓存，数据更新会触发页面 UI 更新。

三、Flutter 服务通道优化

3.1 背景

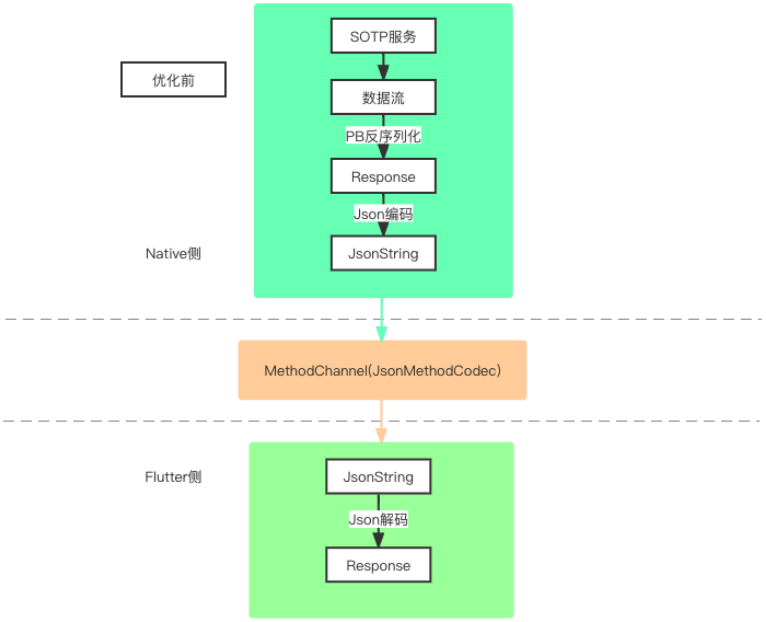
因为我们 APP 采用的私有服务协议，目前发服务的动作还是在 Native 代码上，而酒店的核心页面已经转到了 Flutter 上。通过 Flutter 框架提供的通道技术 Native 到 Flutter 的数据传输通道需要对数据做一次额外的序列化及反序列化的传输，同时传输的过程比较耗时，会阻塞 UI 的渲染主线程，对页面的加载会造成明显的影响。我们检测到这个环节之后和框架一起对 Flutter 的底层框架进行了改造，可以实现数据流直接的透传，同时不阻塞 UI 主线程，性能得到了极大的提升。

优化前，通过服务返回的数据流传递到 flutter 使用，整个过程要经历以下 4 步：

- PB 反序列化
- Response 到 JsonString 的编码
- JsonString 到 MethodChannel(使用 JsonMethodCodec 编解码)

- 传输 JsonString 到 Reponse 的解码

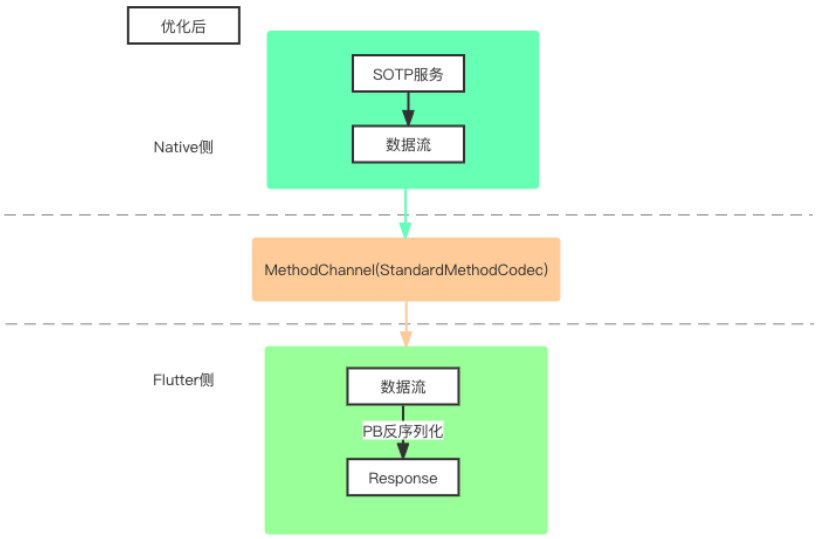
整个过程链路长，数据传输量大，效率低，影响到页面加载性能，如下图所示：



改造后，通过服务返回的数据流，直接传输到 Flutter 侧，在 Flutter 直接进行 PB 的反序列化，传输性能得到极大提升。

- PB 的数据流到 MethodChannel（使用 StandardMethodCodec 编解码）传输
- PB 反序列化到 Response

整个过程链路短，数据传输量小，效率高，如下图所示：



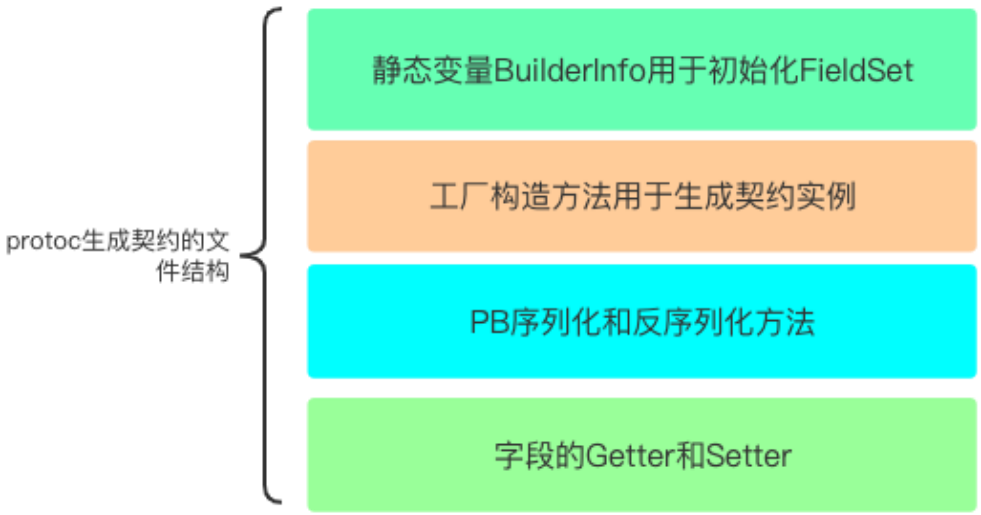
其中 MethodChannel 的编解码器由 JsonMethodCodec 换成了 StandardMethodCodec。因为 StandardMethodCodec 可以避免转换 JsonString 的操作，能节省传输时间。

3.2 Flutter 中使用 Protobuf

在 flutter 中使用 Protobuf，首先需要将 proto 契约文件转化成 dart 文件，可以借助官方编译工具 protoc 进行编译。

（1）获取 protoc 工具

安装 C++
sudo apt-get install autoconf automake libtool curl make g++ unzip
安装 Protobuf 发行版
https://github.com/protocolbuffers/protobuf/releases
下载完成之后，解压，进到目录中执行下面命令编译安装
./configure
make
make check
sudo make install
sudo ldconfig # refresh shared library cache.
安装 protoc-gen-dart 插件
dart pub global activate protoc_plugin
在 Terminal 中执行 protoc 命令生成 dart 文件
protoc --dart_out=. <文件名>.proto



（2）使用生成的 dart 契约文件

执行 flutter pub add protobuf 命令，修改项目的 pubspec.yaml，在 dependencies 中加上:

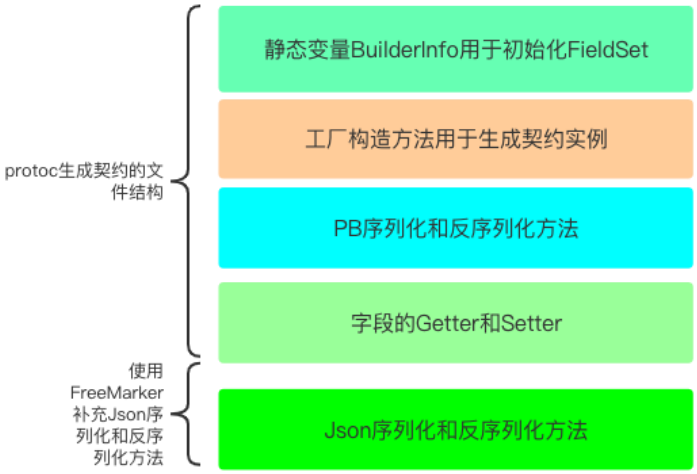
protobuf: ^2.0.1
编写如下测试代码:

```dart
import 'Person.pb.dart';
void main() {
  final person1 = Person();
  person1.name = 'Alice';
  person1.age = 21;
  person1.phone.add('16611113333');

  final buffer = person1.writeToBuffer();
  final person2 = Person.fromBuffer(buffer);
  print(person2);
}
```

执行后可以得到如下结果:

```
name: Alice
age: 21
phone: 16611113333
```

其中，生成 Person 的类继承了 Protobuf 包里的 GeneratedMessage 类，序列化和反序列化由基类实现。但是这种方式不能根据需要定制化生成契约文件。因此，为了更好的兼容 Json 格式的数据，可以使用 FreeMarker 模板引擎定制化生成契约文件。



3.3 使用 FreeMarker 定制化生成 dart 契约文件

FreeMarker 是一款模板引擎：即一种基于模板和要改变的数据，并用来生成输出文本（HTML 网页、电子邮件、配置文件、源代码等）的通用工具。它不是面向最终用户的，而是一个 Java 类库，是一款程序员可以嵌入他们所开发产品的组件。

下面介绍如何使用 FreeMarker 和 protoc 命令生成任意编程语言的契约文件

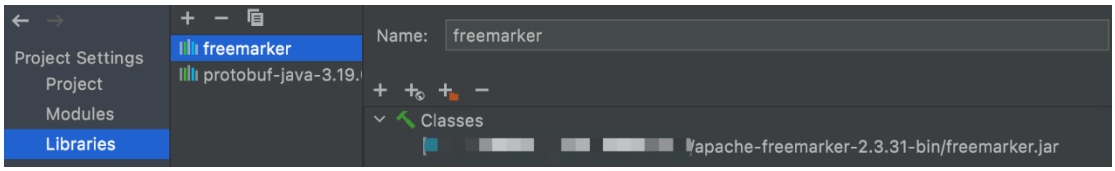
- 下载 FreeMarker 最新版 jar 包

https://freemarker.apache.org/freemarkerdownload.html

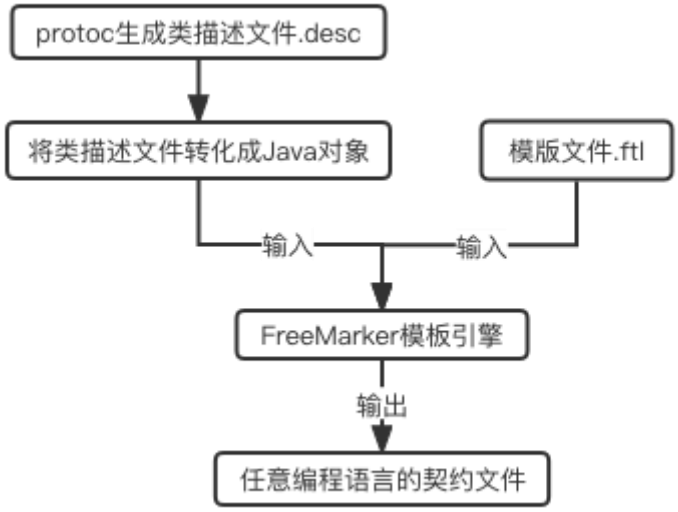
- 下载 Protobuf 对应版本的 jar 包

https://mvnrepository.com/artifact/com.google.protobuf/protobuf-java

- 在 Java 项目中导入对应 jar 包



- 编写 Java 程序



程序的流程如上图所示。首先使用 protoc 命令生成对应的描述文件，其次将描述文件转换成对应 java 对象，最后使用 FreeMarker 模板引擎生成任意语言的契约文件。

```
public class Main {
    private static final String path = '[redacted]';
    public static void main(String[] args) throws Exception{
        //生成类的描述文件desc
        Runtime.getRuntime().exec( command: "protoc -I="+path+" --descriptor_set_out=./Test.desc Test.proto");
        //将类描述文件转化成Java对象
        FileInputStream fin = new FileInputStream( name: path+"Test.desc");
        FileDescriptorSet fileDescriptorSet = FileDescriptorSet.parseFrom(fin);
        List<ClassModel> classModels = new ArrayList<>();
        for (FileDescriptorProto fdp: fileDescriptorSet.getFileList()) {
            for (DescriptorProto dp: fdp.getMessageTypeList()) {
                classModels.add(generateClassModel(dp));
            }
        }
        Map<String, Object> map = new HashMap<>();
        map.put("root", classModels.get(0));
        //生成任意语言的文件
        Configuration config = new Configuration(Configuration.VERSION_2_3_0);
        config.setClassForTemplateLoading(Main.class, basePackagePath: "");
        Template template = config.getTemplate( name: "test.ftl");
        template.process(map, new OutputStreamWriter(System.out));
    }
```

由上图可知，模板引擎的输入是一个 classModel 对象。如下图实现了将描述文件转化成 classModel 对象的功能。

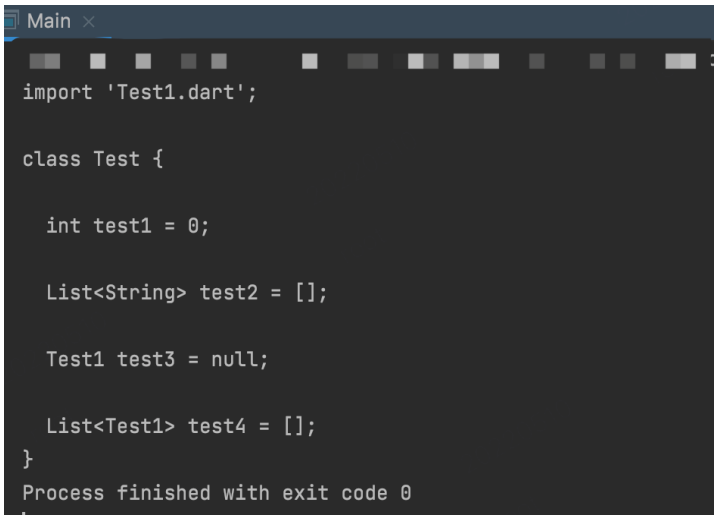
```
private static ClassModel generateClassModel(DescriptorProto dp){
    ClassModel classModel = new ClassModel();
    classModel.setClassName(dp.getName());
    List<FieldModel> fieldModels = new ArrayList<>();
    for (FieldDescriptorProto fdp : dp.getFieldList()) {
        fieldModels.add(generateFieldModel(fdp, classModel));
    }
    classModel.setFieldList(fieldModels);
    return classModel;
}
```

```
private static FieldModel generateFieldModel(FieldDescriptorProto fdp, ClassModel classModel) {
    FieldModel fieldModel = new FieldModel();
    fieldModel.setFieldName(fdp.getName());
    String type = fdp.getType().toString().toLowerCase();
    String label = fdp.getLabel().toString();
    boolean isList = false;
    if ("LABEL_REPEATED".equals(label)) {
        isList = true;
    }
    fieldModel.setIsArray(isList);
    if ("type_int32".equals(type)) {
        fieldModel.setFieldType("int");
        fieldModel.setDefaultValue("0");
    } else if ("type_string".equals(type)) {
        fieldModel.setFieldType("String");
        fieldModel.setDefaultValue("\"\"");
    } else if ("type_message".equals(type)) {
        String typeName = fdp.getTypeName();
        String fieldType = typeName.substring(1);
        fieldModel.setIsMessage(true);
        fieldModel.setFieldType(fieldType);
        fieldModel.setDefaultValue("null");
        classModel.addLowerCaseImportStr(fieldType);
```

FTL 模板文件如下图所示：

```
<#list root.lowerCaseImportStrList as importStr>
import '${importStr}.dart';
</#list>

class ${root.className} {
<#list root.fieldList as fieldModel>
  <#assign fieldName = fieldModel.fieldName>
  <#assign fieldType = fieldModel.fieldType>
  <#assign fieldNameUncapFirst = fieldName?uncap_first>
  <#assign defaultValue = fieldModel.defaultValue>
  <#if fieldModel.isArray>

  List<${fieldType}> ${fieldNameUncapFirst} = [];
  <#else>

  ${fieldType} ${fieldNameUncapFirst} = ${defaultValue};
  </#if>
</#list>
}
```

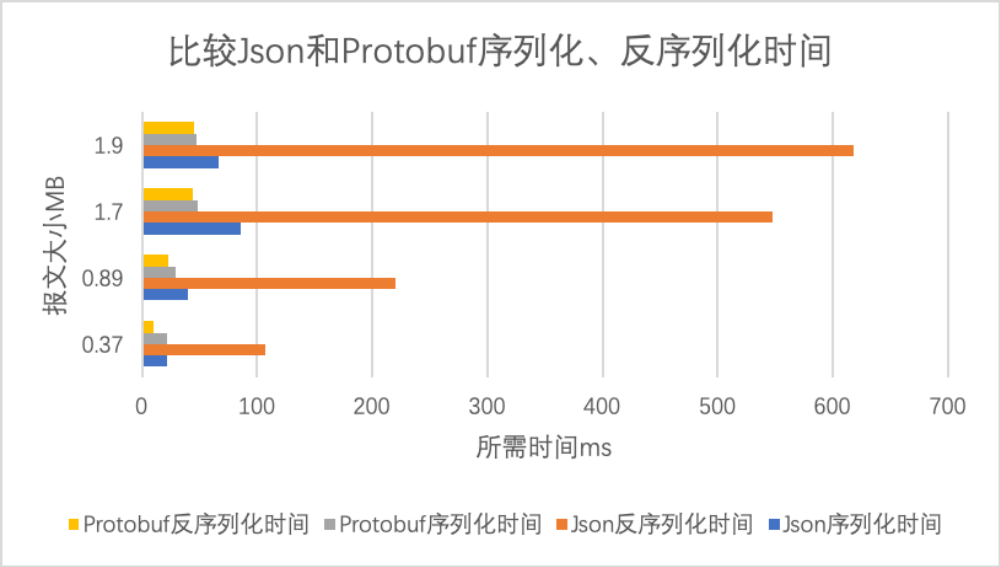
- 执行代码输出契约文件



这样就可以实现了根据 proto 文件自定义生成任意编程语言的契约文件。

3.4 Json 与 Protobuf 的性能对比

我们对比了相同报文情况下 Json 和 Protobuf 在序列化和反序列化上所花费的时间。从下图可知，Protobuf 在序列化和反序列化相同大小报文时比 Json 花费的时间大大减少了，也大大提高了我们获取数据的速度。



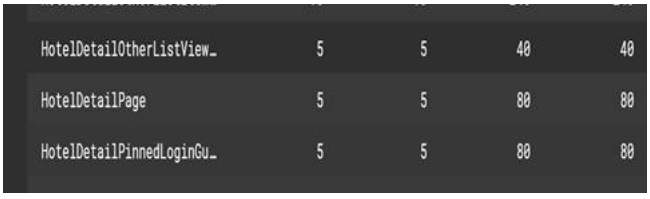
四、内存泄漏治理

4.1 内存泄漏的常用监控手段

内存泄漏是一个比较严重的问题，如果出现，对 App 的稳定性和用户体验都有非常大影响。因此对这块的监控和治理也是我们非常关注的一块。

在监控方面 Flutter 现在比较通用的方法就是利用 Expando 中的弱引用去监控我们要检查是否有泄漏的对象，如果出现则从 VM 中获取其引用链接，从而分析其泄漏原因。我们的框架也利用此方法监控了我们 app 中的每个页面是否在退出时还存在泄漏。

另外通过 Flutter 的 Dev tool 中的内存监控工具也能实现对泄漏对象的发现。比如对于酒店详情页面，反复进入和退出此页面，如果有泄漏会发现，在内存监控工具中出现此页面多个的对象存活，此时基本可以判断出此页面出现了泄漏了。下图的第一列是类名，第二、三列是实例数量，第四、五列是对应分配的字节数。



4.2 内存泄漏的治理

下面介绍一下，我们在我们页面的内存泄漏治理中发现的一些导致泄漏的原因和解决的办法。

（1）调用 Native 的 Plugin 时，对 Future 的 Then 设置的闭包没有关闭

在调用 Native 的 Plugin 接口时，有时会设置一个 Then 的闭包，期望在这个闭包里去处理这个 Plugin 的返回结果。这个闭包会注册到引擎的全局变量里面，如果 Native 调用了 result 的 listener，这个 Then 的闭包会走到，然后会被清除掉。如果某些 case，Native 没有调用，则这个闭包会泄露，如果这个闭包所属的 Model 能引用到页面对象的话，则会造成整个页面的泄露。

比如下面这个例子，我们进入 flutter 页面时会调这个 plugin，但是 native 对应的 result 则必须在某些 case 情况下才会回调。而大部分情况下，是不会回调的，从而造成整个页面的泄露。解决方法是把 future 转换成 stream，然后我们在页面退出时 cancel 掉，就能避免闭包的泄漏。

例子：调用 Native 的 Plugin 时出现泄漏的情况

Flutter 侧的调用：

```
void callNative() {
FlutterBridge.callNative("method", map).then((value) {
        do some thing;});
}
```

Native 的响应：

```
override fun flutterPluginAction ( result: MethodChannel.Result){
if (condition) {
            result.success(ret)
      } else {
            do something;
      }
}
```

可以看到 Native 在接受到这个 plugin 调用时，对于 result 的调用返回不是一直都会做的，它需要等到满足条件才会做这件事情，而如果它不做这件事情，对应的 flutter 那边的闭包就会一直被保存在引擎中，这个引用链也会一直存在，从而造成这个引用链上的对象都泄漏了。

解决的方法：

```
void callNative () {
Future future = FlutterBridge. callNative ("method");
      _streamSubscription?.cancel();
      _streamSubscription = future?.asStream()?.listen((value)
{
            do something;
});
}
```

我们的解决方式，就是对这种异步但不能确定回调是否一定完成的情况，换成用

StreamSubscription 去监听。然后当页面退出时做一下 cancel 的动作，这样就能避免泄漏的发生。

```
void onPageDestroy() {
    _ streamSubscription?.cancel();
}
```

这种等待对异步调用的回调监听其实都可能存在类似问题，只不过如果是单纯在 Dart 中的异步调用一般不会存在这种不回调的情况。但是对于 plugin 这种跟 native 的交互的地方，我们在初期接触 flutter 时没有关注到这块，有可能会造成遗漏。

（2）一些观察者模式中的订阅者在页面退出时没有取消订阅

这种是大家比较熟悉的一种情况。常见的例子有例如像 Timer，EventBusCenter.defaultBus 和 LifeCycleObserver 等。这些订阅者如果在页面退出时不需要了，需要记得取消掉。否则也会造成内存泄漏，这种情况我们也应该避免。

五、小结

性能优化是一件不断持续，不断深入的事情。我们通过本文中所介绍的改进措施对页面性能实现了很大的优化，达到了不错的效果。后续也会在此基础之上对还可提高的地方继续加深，同时也会对已经验证实行有效的方案去做一些抽象，封装工作，后续提供通用的解决方案。

Trip.com APP QUIC 应用和优化实践

【作者简介】Logan，携程移动开发专家，关注大前端技术领域，对 APP 网络、性能、稳定性有深入研究。

Trip.com APP（携程国际版）主要服务于海外用户，这些用户请求大多需要回源至国内，具有链路长、网络不稳定、丢包率高等特性。为了解决用户请求耗时长、成功率低的痛点，在 2021 年初我们尝试引入 QUIC 来提升网络质量。经过近一年的优化实践，取得了显著的成果：网络耗时降低 20%，成功率提升至 99.5%，极大地改善了用户体验。本文将从客户端的视角详细介绍 QUIC 的应用和优化经验。

一、背景

Trip.com APP 原网络框架是基于 TCP 的，经过一系列优化后，成功率和耗时均已到达瓶颈。主要的失败原因集中在请求超时和链接断开。这是 TCP 协议本身的限制导致：

- TCP 是基于链接的，用户网络发生切换，或者 NAT rebinding 都会导致链接断开请求失败，同时每次重新建立链接均需要握手耗时。
- TCP 内置了 CUBIC 拥塞控制算法，这种基于丢包的拥塞控制在 Trip.com 的长肥管道场景（请求大多是海外回源国内）及弱网环境下更容易超时失败。
- TCP 的头部阻塞场景进一步增加请求耗时。

而 QUIC（quick udp internet connection）是一种基于 UDP 的可靠传输协议，有 0 RTT，链接迁移，无队头阻塞，可插拔的拥塞控制算法等优秀特性，非常契合我们的用户请求场景。

二、配套

（1）Trip.com QUIC 客户端的实现采用了 Google 开源的 Cronet，并在此基础上做了进一步的 size 精简和订制性的优化。

（2）Trip.com QUIC 服务端使用了 Nginx 的 QUIC 分支，目前还没有 release 版本，使用中修复了很多 bug，并做了适配性的改造。

三、应用和优化实践

引入 QUIC 过程中最大的难点就是 Cronet 库体积过大，需要经过裁剪后才能在 APP 中使用。使用后经过对比实验发现 QUIC 并没有达到预期的效果，这是因为 QUIC 的 0 RTT，链接迁移等诸多优秀特性并不是开箱即用的，需要做定制化的改造才能享受到这些特性带来的性能提升。于是我们又进一步做了 IP 直连，支持单域名多 IP 链接，0 RTT 和链接迁移改造，QPACK 优化，拥塞控制算法选择，QUIC 使用方式优化等许多改造。经过改造后的整套网络方案极大的提升了网络质量，改善了用户体验，接下来详细介绍下我们优化过程中踩过的坑和相关经验成果。

3.1 Cronet 代码裁剪

业内有很多客户端 QUIC 的实现方案，Cronet 是最成熟的，但是 5M 的 size 让很多 APP 望而却步，所以我们做的第一件事就是对 Cronet 进行裁剪。

Cronet 是 chrome 的核心网络库, 内部集成了 HTTP1/HTTP2/SPDY/QUIC, QPCK, BoringSSL, LOG，缓存，DNS 解析等很多模块，我们仅保留了 QUIC/QPACK/BoringSSL 等必须的核心功能，将 Cronet 库 size 减少 60%以上。针对 Cronet 的环境搭建、ninja 编译、如何修改.gn 文件来剔除无用模块，具体的逻辑代码删减等细节后续会推出专门的文章来介绍，同时也会争取将裁剪后的代码开源供大家使用。

3.2 IP 直连

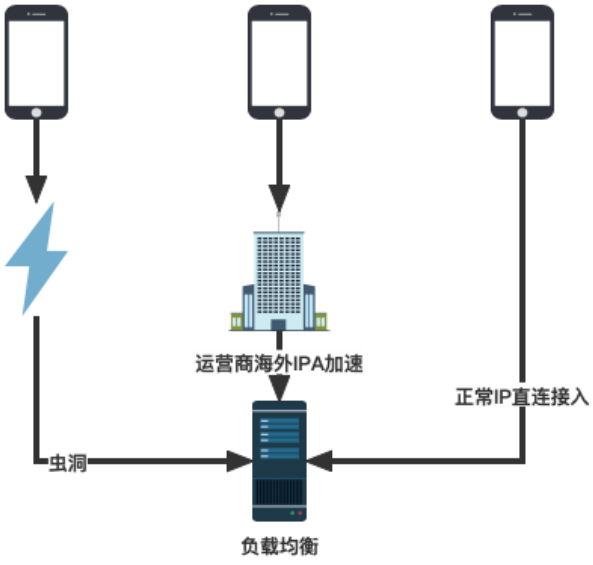
我们通过修改 Cronet 源码，直接指定最优 QUIC IP，实现了 IP 直连，减少 DNS 解析耗时。

DNS 解析是需要耗时的，并且可能出现解析失败，DNS 拦截等问题。有时还会受网络运营商的影响，DNS 解析出的不是最优 ServerIP。

Cronet 对 DNS 解析做了很多优化，UDP 请求，TCP 补偿，支持 Https 解析以防止 DNS 拦截等，但这是浏览器需要的通用方案。

对于企业自己的 APP 来说，域名固定，服务入口 IP 变化较少，所以 Trip.com APP 内置了 QUIC Server IP 列表（支持 IP 列表动态更新），根据用户位置和网络状况指定最优 IP。使得 DNS 解析的耗时和失败率均达到了 0 。

以下为目前的 QUIC 接入方案，海外用户可以灵活的选择通过虫洞方式接入或者海外运营商提供的 IPA 加速（UDP 转发）节点接入，大陆用户可以通过普通的 IP 直连方式接入。



3.3 支持单域名多链接

Cronet 对一个域名仅支持建立一个 QUIC 链接，但在大多数 APP 的使用场景中域名固定，需要建立基于该域名下多个 IP 的 QUIC 链接，择优使用，于是我们做了进一步的改造。

Cronet 建立链接是用域名作为 session_key 的，所以一个域名只能同时建立一个链接，如果存在同域名的 session 直接复用，代码如下：

```
int QuicStreamFactory::Create(const QuicSessionKey& session_key,
                              const HostPortPair& destination,
                              quic::ParsedQuicVersion quic_version,
                              RequestPriority priority,
                              int cert_verify_flags,
                              const GURL& url,
                              QuicStreamRequest* request,
                              std::string& specified_ip) {
  if (clock_skew_detector_.ClockSkewDetected(base::TimeTicks::Now(),
                                             base::Time::Now())) {
    MarkAllActiveSessionsGoingAway(kClockSkewDetected);
  }
  DCHECK(HostPortPair(session_key.server_id().host(),
                      session_key.server_id().port())
             .Equals(HostPortPair::FromURL(url)));


  // Use active session for |session_key| if such exists.
  // TODO(rtenneti): crbug.com/498823 - delete active_sessions_.empty() checks.
  if (!active_sessions_.empty()) {
    auto it = active_sessions_.find(session_key);
    if (it != active_sessions_.end()) {
      QuicChromiumClientSession* session = it->second;
      request->SetSession(session->CreateHandle(destination));
      return OK;
    }
  }

  // Associate with active job to |session_key| if such exists.
  auto it = active_jobs_.find(session_key);
  if (it != active_jobs_.end()) {
    it->second->AddRequest(request);
    return ERR_IO_PENDING;
  }
```

这显然不能满足我们的需求：

- 用户的网络变化会导致最优 IP 的变化，比如用户的网络从电信的 Wi-Fi 切换到联通的 4G，此时最优 IP 也会从电信切换到联通。
- 某个 IP 对应的机房发生故障需要立马切换到其他 IP。
- 某些情况下，需要同时建立不同 IP 的多条链接来发送请求，对比不同链接的表现（成功率，耗时等），动态调整 IP 权重。

所以我们对链接的管理进行了订制化的改造：

- 对 HTTP request 增加 IP 参数，支持对不同请求指定任意 IP。
- 修改 Cronet QuicSessionKey 的重载方法，将指定的 IP+Host 做为 Session Key 以支持单域名多 IP 链接。

改造核心代码如下：

```
bool QuicSessionKey::operator<(const QuicSessionKey& other) const {
  return std::tie(server_id_, socket_tag_, network_isolation_key_,
                  disable_secure_dns_,specified_ip_) <
         std::tie(other.server_id_, other.socket_tag_,
                  other.network_isolation_key_, other.disable_secure_dns_,other.specified_ip_);
}
bool QuicSessionKey::operator==(const QuicSessionKey& other) const {
  return server_id_ == other.server_id_ && socket_tag_ == other.socket_tag_ &&
         network_isolation_key_ == other.network_isolation_key_ &&
         disable_secure_dns_ == other.disable_secure_dns_ && specified_ip_ == other.specified_ip_;
}
```

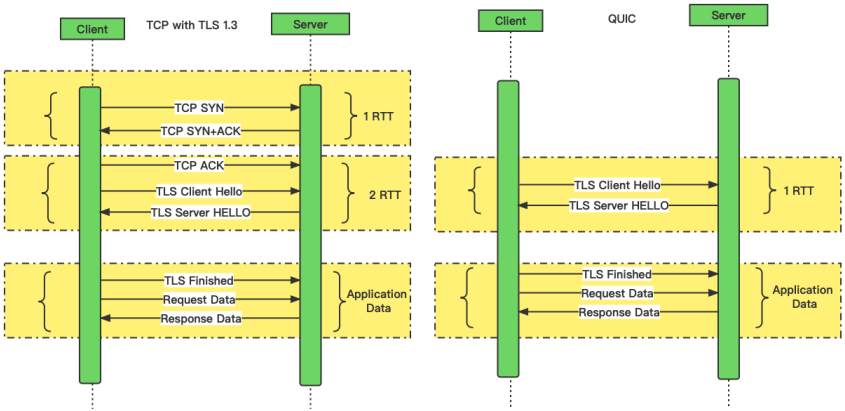
改造后的代码支持了单域名多链接，形成了 QUIC 链接池，能让开发者灵活的选取最优的链接进行使用，进一步降低了请求耗时，提升了请求成功率。

3.4 0 RTT 优化

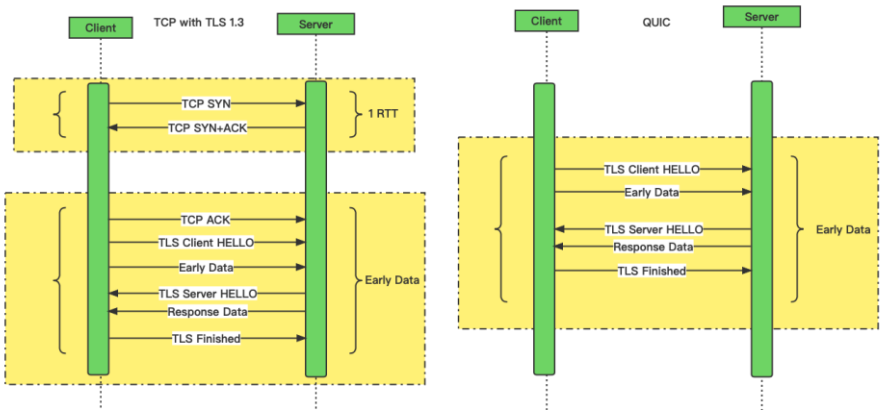
0 RTT 是 QUIC 最让人心动的一个特性，没有握手延迟，直接发送请求数据。但由于负载均衡的存在和重放攻击的威胁使得我们必须对 Cronet 和 Nginx 进行定制化的改造才能完整的体验 0 RTT 带来的巨大的性能提升。

目前 Trip.com 的多数请求是回源到国内的，以一个纽约用户访问为例，纽约到上海的直线距离是 14000km，假设两地直连光纤，光的传输速度为 300000km/s，考虑折射率，光纤中的传输速度为 200000km/s，那么 1 个 RTT 则需要 14000/200000 *2 = 140ms。而实际上的传输链路很复杂，要远大于这个数字，所以 RTT 的减少对我们来说是至关重要的。首先让我们了解一下 0 RTT 的工作原理。

使用 TLS1.3 的情况下，首次建立链接，在发送真正的请求数据前 TCP 需要经过两个完整的 RTT（TLS1.2 需要 3 个 RTT），一次用于 TCP 握手，一次用于 TLS 加密握手。而 QUIC 由于 UDP 不需要建立链接，仅需要一次 TLS 加密握手，如下图所示。



多数情况下，在整个 APP 的生命周期内首次建链只会发生一次，之后客户端再需要建立链接会节省一个 RTT，这时候 QUIC 能以 0 RTT 的方式直接发送请求（Early Data）如下图所示：



QUIC 使用了 DH 加密算法，DH 加密算法比较复杂，在这里不做详细解释，有兴趣的可以参考这篇 wiki：《迪菲-赫尔曼密钥交换》。大概的原理是客户端和服务端各自生成自己的非对称公私钥对，并将公钥发给对方，利用对方的公钥和自己的私钥可以运算出同一个对称密钥。同时客户端可以缓存服务的公钥（以 SCFG 的方式），用于下次建立链接时使用，这个就是达成 0-RTT 握手的关键。客户端可以选择内存缓存或者磁盘缓存 SCFG。内存缓存在 APP 本次生命周期内有效，磁盘缓存可以长期有效。

但是 SCFG 中的 ticket 有时效性（比如设置为 24 小时），过了有效期，Client 发起 0 RTT 请求会收到 Server 的 reject，然后重新握手，这反而增加了建立链接开销。Trip.com 是旅游类的低频 APP，所以使用了内存缓存，对于社交/视频/本地生活等高频类 APP 可以考虑使用磁盘缓存。

0 RTT 开启后我们实验观察请求耗时并没有明显降低。通过 Wireshark 抓包发现 GET 请求和 POST 请求的 0 RTT 方式并不一致。

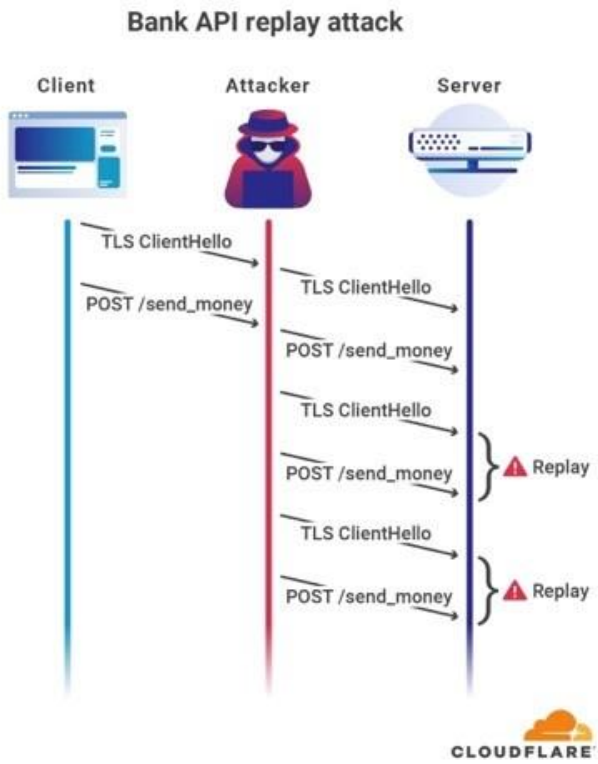
POST 请求的 0 RTT 如下图所示，客户端会同时向服务发送 Initial 和 0 RTT 包，但是并没有发送真正的应用请求数据（Early Data），而是等服务返回后再同时发送 Handshake 完成包及数据请求包。这说明 POST 是在 TLS 加密完成后才开始发送请求数据，依然经历了一次完整的 RTT 握手，虽然握手包大小和数量相对于首次建立链接有所减少，但是 RTT 并没减少。

Time	Source	SourcePort	Destination	DestPort	Protocol	Length	Info
58.156120					QUIC	1392	Initial, DCID=80e942b40eded445, PKN: 1, CRYPTO, PADDING
58.159853					QUIC	122	0-RTT, DCID=80e942b40eded445
58.219950					QUIC	1290	Protected Payload (KP0)
58.219951					QUIC	609	Protected Payload (KP0)
58.220401					QUIC	76	Protected Payload (KP0)
58.220682					QUIC	132	Handshake, DCID=80e942b40eded44511540a6e3d92aa8e403399a2
58.221836					QUIC	87	Protected Payload (KP0), DCID=80e942b40eded44511540a6e3d92aa8e403399a2
58.221837					QUIC	472	Protected Payload (KP0), DCID=80e942b40eded44511540a6e3d92aa8e403399a2
58.244379					QUIC	66	Protected Payload (KP0)
58.280441					QUIC	151	Protected Payload (KP0)
58.281363					QUIC	95	Protected Payload (KP0)
58.281581					QUIC	87	Protected Payload (KP0), DCID=80e942b40eded44511540a6e3d92aa8e403399a2
58.287454					QUIC	1013	Protected Payload (KP0)
58.287492					QUIC	84	Protected Payload (KP0)
58.287739					QUIC	87	Protected Payload (KP0), DCID=80e942b40eded44511540a6e3d92aa8e403399a2
58.307406					QUIC	68	Protected Payload (KP0)

GET 请求的 0 RTT 如下图所示，客户端同时向服务发送 Initial 和两个 0 RTT 包，其中第二个 0 RTT 包中携带了 early_data，即真正的请求数据。

Time	Source	SourcePort	Destination	DestPort	Protocol	Length	Info
58.644196					QUIC	1392	Initial, DCID=0e298571a413a834, PKN: 1, CRYPTO, PADDING
58.654191					QUIC	121	0-RTT, DCID=0e298571a413a834
58.655333					QUIC	460	0-RTT, DCID=0e298571a413a834
58.706518					QUIC	1290	Protected Payload (KP0)
58.706519					QUIC	609	Protected Payload (KP0)
58.707244					QUIC	132	Handshake, DCID=0e298571a413a83411560a6e3d91cdcb0953e177
58.708396					QUIC	87	Protected Payload (KP0), DCID=0e298571a413a83411560a6e3d91cdcb0953e177
58.713381					QUIC	76	Protected Payload (KP0)
58.715363					QUIC	100	Protected Payload (KP0)
58.715726					QUIC	87	Protected Payload (KP0), DCID=0e298571a413a83411560a6e3d91cdcb0953e177
58.766433					QUIC	151	Protected Payload (KP0)
58.792078					QUIC	87	Protected Payload (KP0), DCID=0e298571a413a83411560a6e3d91cdcb0953e177
59.798546					QUIC	1059	Protected Payload (KP0)
59.798547					QUIC	84	Protected Payload (KP0)
59.798982					QUIC	87	Protected Payload (KP0), DCID=0e298571a413a83411560a6e3d91cdcb0953e177

深究其原因会发现这是 0 RTT 不具备前向安全性和容易受到重放攻击导致的。这里重点说一下重放攻击，如下图所示，如果用户被诱导往某个账户里转账 0.1 元，该请求正好是发生在 0 RTT 阶段，即 early_data 里携带的正好是一个转账类的请求，并且该请求如果被攻击者监听到，攻击者不断的向服务发送同样的 0 RTT 包重放这个请求，会导致客户的银行卡余额被掏空！



对于 Cronet 来说无法细化哪个请求使用 Early Data 是安全的，只能按照类型划分，POST，PUT 请求均是等握手结束后再发请求数据，而 GET 请求则可以使用 Early Data。注意，握手结束后的数据是前向安全的，此时会再生成一个临时密钥用于后续会话。

但对 Trip.com APP 而言我们可以做更加细分的处理，能较好的区分出是否为幂等请求，对幂等类请求放开 Early Data，非幂等类则禁止。在 APP 中，大多数请求为信息获取类的幂等

请求，因此可以充分利用 0 RTT 来减少建立链接耗时，提升网络性能。

同时我们也对 Nginx 做了 0 RTT 改造。现实情况下服务是多机部署，通过负载均衡设备进行请求转发的。由于每台机器生成的 SCFG 并不一致（即生成的公私钥对不唯一），当客户端 IP 地址发生变化，重新建立链接时，请求会随机打到任意一台机器上，如果与首次建立链接的机器不一致则校验失败，nginx 会返回 reject，然后客户端会重新发起完整的握手请求建立链接。具体的改造方式参照我们在服务端的 QUIC 应用和优化实践一文。

简单的来说，通过改造，保证所有机器的 SCFG 一致。目前 Trip.com 0 RTT 成功率在 99.9% 以上。

上面的两条完成后，再对比一下：

- 正常的 Http2.0 请求在发送请求前，需要经过 DNS 解析+TCP 三次握手（1 个 RTT）+TLS 加密握手（TLS1.2 需要 2 个 RTT，TLS1.3 需要 1 个 RTT），共 2 个 RTT（TLS1.2 共 3 个 RTT）。
- 自研的 TCP 框架需要经历 TCP 三次握手共 1 个 RTT。
- 经过我们优化后的 QUIC 大多数情况下发送请求前只需要 0 RTT。

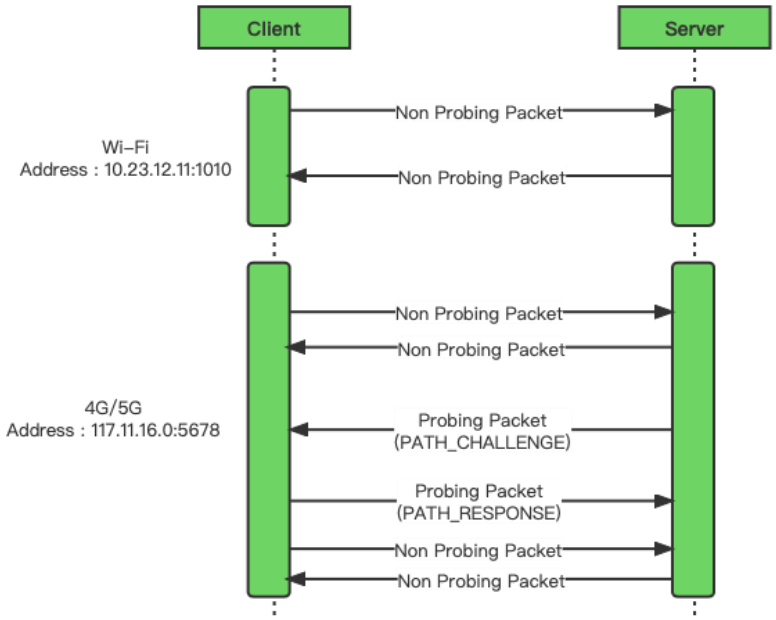
使用改造后的 QUIC，在 Trip.com APP 中，用户建立链接的耗时约等于 0，极大的降低了请求耗时。

3.5 链接迁移改造

QUIC 的链接迁移能让用户在网络变化（NAT rebinding 或者网络切换）时保持链接不断开，但因为负载均衡的存在会使用户网络变化时请求转发到不同的服务器上导致迁移失败，因此需要做一些定制化的改造才能体验这一特性带来的用户体验提升。

TCP 链接是基于五元组的，客户端 IP 或者端口号发生变化都会导致链接断开请求失败。大家生活中的网络情况日趋复杂，经常在不同的 Wi-Fi、蜂窝网之间来回切换，如果每次切换都出现失败必然是非常影响体验的。

而 QUIC 的链接标识是一个 20 字节的 connection id。用户网络发生变化时（无论是 IP 还是端口变化），由于链接标识的唯一性，无需创建新的链接，继续用原有链接发送请求。这种用户无感的网络切换就是链接迁移。下图是链接迁移的工作流程：



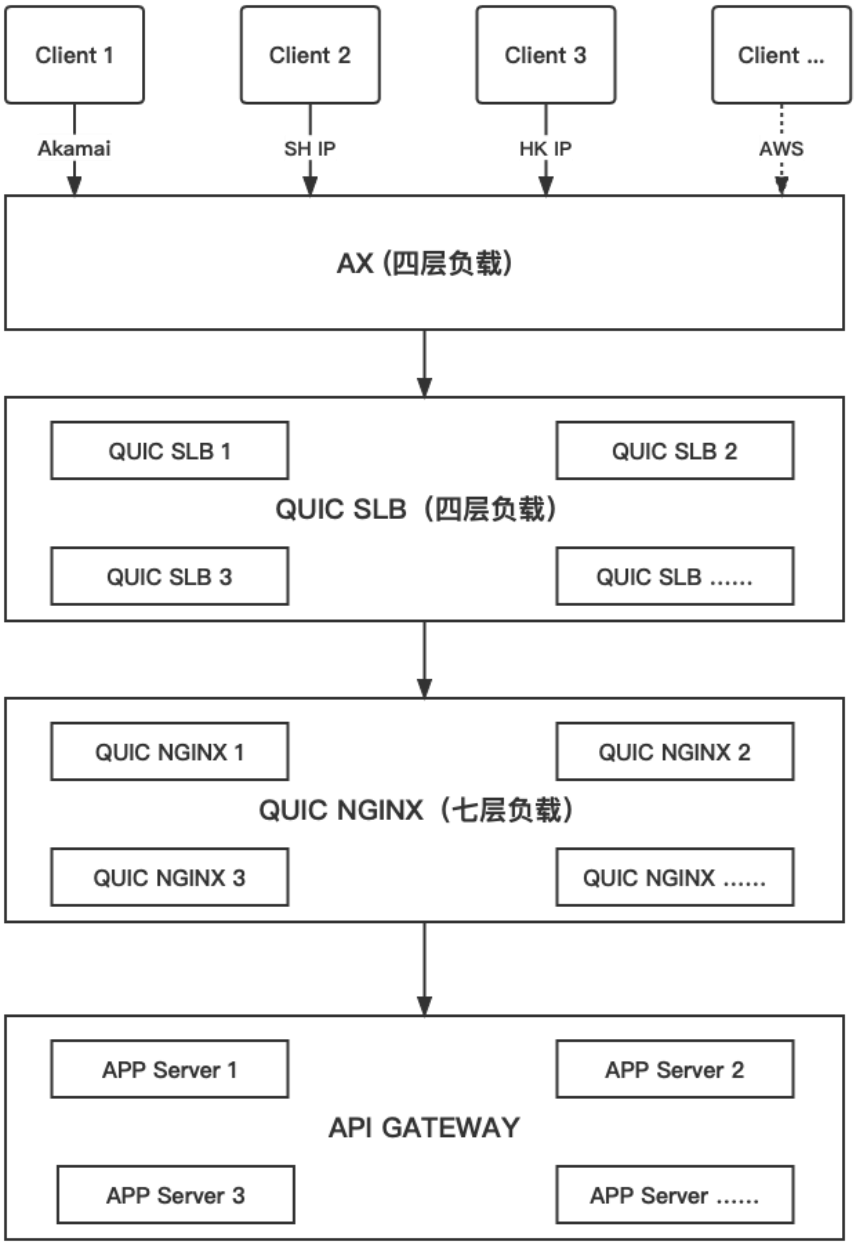
名词解释：

- Probing Frame 是指具有探测功能的 Frame，比如 PATH_CHALLENGE, PATH_RESPONSE, NEW_CONNECTION_ID, PADDING 均为 Probing Frame。
- 一个 Packet 中只包含 Probing Frame 则称为 Probing packet，其他 Packet 则称为 Non Probing Packet。

如上图所示，开始时用户和服务正常通信。某个时间点用户的网络从 WI-FI 切换到 4G，并继续正常向服务发送请求，服务检测到该链接上客户端地址发生变化，开始进行地址验证。即生成一个随机数并加密发给客户端（Path_Challenge）,如果客户端能解密并将该随机数发回给服务（Path_Response），则验证成功，通信恢复。

但由于负载均衡的存在会使用户网络变化时请求转发到不同机器上导致迁移失败，我们首先想到的是修改负载均衡的转发规则，利用 connection id 的 hash 进行转发似乎就可以解决这个问题，但是 QUIC 的标准规定链接迁移时 connection id 也必须进行更改，同时建立链接前初始化包中的 connection id 以及链接建立完成后的 connection id 也不一致，所以此方案也行不通。最终我们通过两个关键点的改造实现了链接迁移：

- 修改 connection id 的生成规则，将本机的特征信息加入到 connection id 中；
- 增加 QUIC SLB 层，该层仅针对 connection id 进行 UDP 转发，当链接迁移发生时如果本机缓存中不存在则直接从 connection id 中解析出具体的机器，找到对应的机器后进行转发，如下图所示：



改造细节也可参照服务端的 QUIC 应用和优化实践一文。

通过链接迁移的改造，Trip.com 用户不会再因为 NAT rebinding 或者网络切换导致请求失败，提升了请求成功率，改善了用户体验。

3.6 QPACK 优化

QPACK 即 QUIC 头部压缩，复用了 HTTP2/HPACK 的核心概念，但是经过重新设计，保证了 UDP 无序传输下的正确性。QPACK 有灵活的实现方式，目标是以更少的头部阻塞来接近 HPACK 的压缩率。而我们的改造能使得 Trip.com APP 的请求压缩率和头部阻塞均达到最优。

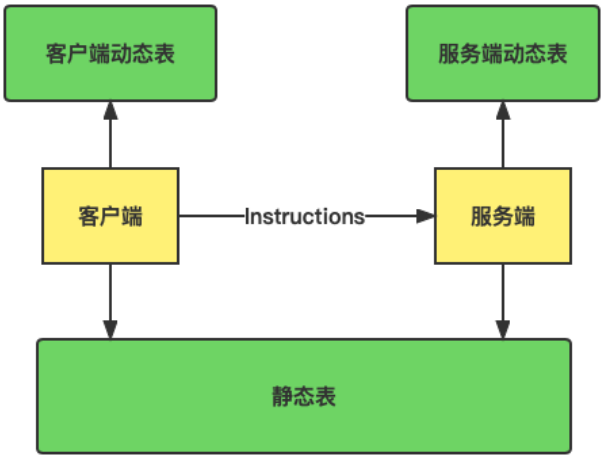
nginx 要开启 QPACK 动态表，需要指定两个参数：

- http3_max_table_capacity 动态表大小，Trip.com 指定为 16K；
- http3_max_blocked_streams 最大阻塞流数量，如果指定为 0，则禁用了 QPACK 动态表。

伴随着 QPACK 动态表的开启，HTTP3 是会出现头部阻塞的（目前发现的唯一一个 QUIC 中头部阻塞的场景），QPACK 是如何工作，在 Trip.com APP 中如何做才能让 QPACK 既能拥有高压缩率又能避免头部阻塞呢？

我们知道 HTTP header 是由许多 field 组成的，比如下图是一个典型的 HTTP header。:authority: www.trip.com 就是其中一个 field。:authority 为 field name，www.trip.com 是 field value。

当 QUIC 链接建立后，会初始化两个单向 stream，Encoder Stream 和 Decoder Stream。一旦建立，这两个 stream 是不能关闭的，之后 HTTP header 动态表的更新就在这两个 stream 上进行。我们以发送 request 为例，客户端维护了一张动态表，并通过指令通知服务端进行更新，以保持客户端和服务端的动态表完全一致，如图所示：



我们可以看到 encoder 和 decoder 共享一张静态表，这张表是由 ietf 标准规定的，服务和客户端写死在代码里永远不会变的，表的内容固定为 99 个字段，截取部分示例：

Index	Name	Value
0	:authority	
1	:path	/
2	age	0
3	content-disposition	
4	content-length	0
5	cookie	
6	date	
7	etag	
8	if-modified-since	
9	if-none-match	
10	last-modified	
11	link	
12	location	
13	referer	
14	set-cookie	
15	:method	CONNECT
16	:method	DELETE
17	:method	GET
18	:method	HEAD
19	:method	OPTIONS
20	:method	POST
21	:method	PUT
22	:scheme	http
23	:scheme	https
24	:status	103
25	:status	200
26	:status	304
27	:status	404

而动态表初始为空，有需要才会插入。

假如我们连续发送三个请求 request A,B,C，三个请求的 Http Header 均为：

```
{
  :authority: www.trip.com
  :method: POST
  cookie:this is a very large cookie maybe more than 2k
  x-trip-defined-header-field-name: trip defined headerfield value
}
```

发送 request A 之前客户端会将 header 做第一次压缩，主要是用静态表进行替换，并将某些数据插入动态表。规则为：

- 静态表存在完全匹配（name+value 完全一致），则直接替换为静态表的 index，比如:method: POST 直接替换为 20；
- 动态表存在完全匹配，则直接替换为动态表 index，首次请求动态表为空；
- 静态表存在 name 匹配，则 name 替换为 index，value 插入动态表，比如:authority:www.trip.com 会替换为 0: www.trip.com，其中 www.trip.com 会插入动态表，假设在动态表中的 index 为 1，我们用 d1 表示动态表中的 index 1；
- 动态表存在 name 匹配，则 name 替换为 index，value 插入动态表；
- 均不存在，name 和 value 均插入动态表，比如 x-trip-defined-header-field-name: trip defined header field value 会整条插入。

客户端本地插入动态表后必须要向服务端发送指令同步更新服务端动态表。经过首次压缩，header 变为：

```
{
  0: d1,
  20,
  5:d2,
  d3,
}
```

替换后的 header 已经非常小了，但是 QPACK 还会对替换后的 header 进行二次 encode。具体压缩代码如下：

```
std::string QpackEncoder::EncodeHeaderList(
    QuicStreamId stream_id,
    const spdy::SpdyHeaderBlock& header_list,
    QuicByteCount* encoder_stream_sent_byte_count) {
  // Keep track of all dynamic table indices that this header block refers to so
  // that it can be passed to QpackBlockingManager.
  QpackBlockingManager::IndexSet referred_indices;

  // First pass: encode into |instructions|.
  Instructions instructions =
      FirstPassEncode(stream_id, header_list, &referred_indices,
                      encoder_stream_sent_byte_count);

  const uint64_t required_insert_count =
      referred_indices.empty()
          ? 0
          : QpackBlockingManager::RequiredInsertCount(referred_indices);
  if (!referred_indices.empty()) {
    blocking_manager_.OnHeaderBlockSent(stream_id, std::move(referred_indices));
  }

  // Second pass.
  return SecondPassEncode(std::move(instructions), required_insert_count);
}
```

其中 SecondPassEncode 会对所有的 field 再次进行处理，不同类型字段处理方式不同，比如对 string 类型进行 huffman 压缩。

```
void QpackInstructionEncoder::Encode(
    const QpackInstructionWithValues& instruction_with_values,
    std::string* output) {
  DCHECK(instruction_with_values.instruction());

  state_ = State::kOpcode;
  instruction_ = instruction_with_values.instruction();
  field_ = instruction_->fields.begin();

  // Field list must not be empty.
  DCHECK(field_ != instruction_->fields.end());

  do {
    switch (state_) {
      case State::kOpcode:
        DoOpcode();
        break;
      case State::kStartField:
        DoStartField();
        break;
      case State::kSbit:
        DoSBit(instruction_with_values.s_bit());
        break;
      case State::kVarintEncode:
        DoVarintEncode(instruction_with_values.varint(),
                       instruction_with_values.varint2(), output);
        break;
      case State::kStartString:
        DoStartString(instruction_with_values.name(),
                      instruction_with_values.value());
        break;
      case State::kWriteString:
        DoWriteString(instruction_with_values.name(),
                      instruction_with_values.value(), output);
        break;
    }
  } while (field_ != instruction_->fields.end());

  DCHECK(state_ == State::kStartField);
}
```

```
void QpackInstructionEncoder::DoWriteString(quiche::QuicheStringPiece name,
                                            quiche::QuicheStringPiece value,
                                            std::string* output) {
  DCHECK(field_->type == QpackInstructionFieldType::kName ||
         field_->type == QpackInstructionFieldType::kValue);

  quiche::QuicheStringPiece string_to_write =
      (field_->type == QpackInstructionFieldType::kName) ? name : value;
  if (use_huffman_) {
    if (use_fast_huffman_encoder_) {
      http2::HuffmanEncodeFast(string_to_write, string_length_, output);
    } else {
      http2::HuffmanEncode(string_to_write, string_length_, output);
    }
  } else {
    QuicStrAppend(output, string_to_write);
  }

  ++field_;
  state_ = State::kStartField;
}
```

二次 encode 后就只有几个字节了。所以我们用 wireshark 抓包会发现 http header 非常小，小到只有一两个字节，这就是 QPACK 压缩的威力。

当压缩后的 header 传到服务端时，服务端找到解析 header 需要的最大的动态表 index，目前是 d3，如果比当前的动态表最大 index 还要大，说明动态表插入请求还没收到，这是 UDP 传输的无序性导致的，需要进行等待。

等待期间 Request B, C 的请求也到了，他们的 header 是一致的，但是都没法解析，因为 requestA 的动态表插入请求还没收到，于是出现了头部阻塞。nginx 有 http3_max_blocked_streams 字段配置允许阻塞的 stream 数量，如果超过了，后续的请求不会等待动态表的插入而是直接将完整的字段 x-trip-defined-header-field-name: trip defined header field value 压缩后发送给服务端。

正常使用 QPACK 只需要做好配置就 ok 了，但是 Trip.com APP 中有些特殊的 Http header 字段，比如 x-xxxx-id: GUID . 这类字段的 value 是每次变化的 GUID，用作请求的唯一标识或用来对请求进行链路追踪等。由于这类字段的 value 每次变化，导致动态表频繁插入很快就会超过阈值，动态表超过阈值后会对老的字段进行清理，删除后如果后续请求又用到了这些字段则还需要再次进行插入。动态表的频繁插入删除则会加重头部阻塞。

所以针对这些 value 一定会变化的字段，我们需要做特殊处理，这类字段的 value 不插入动态表，即不以动态表索引的方式进行替换，只做二次 encode 压缩处理如下（代码较长不做完整展示）：

```
QpackEncoder::Instructions QpackEncoder::FirstPassEncode(
    QuicStreamId stream_id,
    const spdy::SpdyHeaderBlock& header_list,
    QpackBlockingManager::IndexSet* referred_indices,
    QuicByteCount* encoder_stream_sent_byte_count) {
  // If previous instructions are buffered in |encoder_stream_sender_|,
  // do not count them towards the current header block.
  const QuicByteCount initial_encoder_stream_buffered_byte_count =
      encoder_stream_sender_.BufferedByteCount();

  Instructions instructions;
  instructions.reserve(header_list.size());

  // The index of the oldest entry that must not be evicted.
  uint64_t smallest_blocking_index =
      blocking_manager_.smallest_blocking_index();
  // Entries with index larger than or equal to |known_received_count| are
  // blocking.
  const uint64_t known_received_count =
      blocking_manager_.known_received_count();
  // Only entries with index greater than or equal to |draining_index| are
  // allowed to be referenced.
  const uint64_t draining_index =
      header_table_.draining_index(kDrainingFraction);
  // Blocking references are allowed if the number of blocked streams is less
  // than the limit.
  const bool blocking_allowed = blocking_manager_.blocking_allowed_on_stream(
      stream_id, maximum_blocked_streams_);

  // Track events for histograms.
  bool dynamic_table_insertion_blocked = false;
  bool blocked_stream_limit_exhausted = false;

  for (const auto& header : ValueSplittingHeaderList(&header_list)) {
    // These strings are owned by |header_list|.
    absl::string_view name = header.first;
    absl::string_view value = header.second;
    bool is_field_never_reuse = is_never_reuse_field_name(name);
    bool is_static;
    uint64_t index;

    auto match_type = is_field_never_reuse ? QpackHeaderTable::MatchType::kNoMatch :
        header_table_.FindHeaderField(name, value, &is_static, &index);

    switch (match_type) {
      case QpackHeaderTable::MatchType::kNameAndValue:
        if (is_static) {
          // Refer to entry directly.
          instructions.push_back(
              EncodeIndexedHeaderField(is_static, index, referred_indices));
```

改造后的 QPACK 在最佳压缩率和头部阻塞之间取得了平衡，减少了请求 size 的同时加快了请求速度，进一步提升了用户体验。

3.7 拥塞控制算法对比

Cronet 内置了 CUBIC，BBR，BBRV2 三种拥塞控制算法，我们可以根据需求灵活的选择，也可以插拔方式的使用其他拥塞控制算法。经过线上实验对比，在 Trip.com APP 场景中，BBR 性能优于 CUBIC、BBRV2。所以目前 Trip.com 默认使用 BBR。后续也会引入其他拥塞控制算法进行对比，并持续优化。

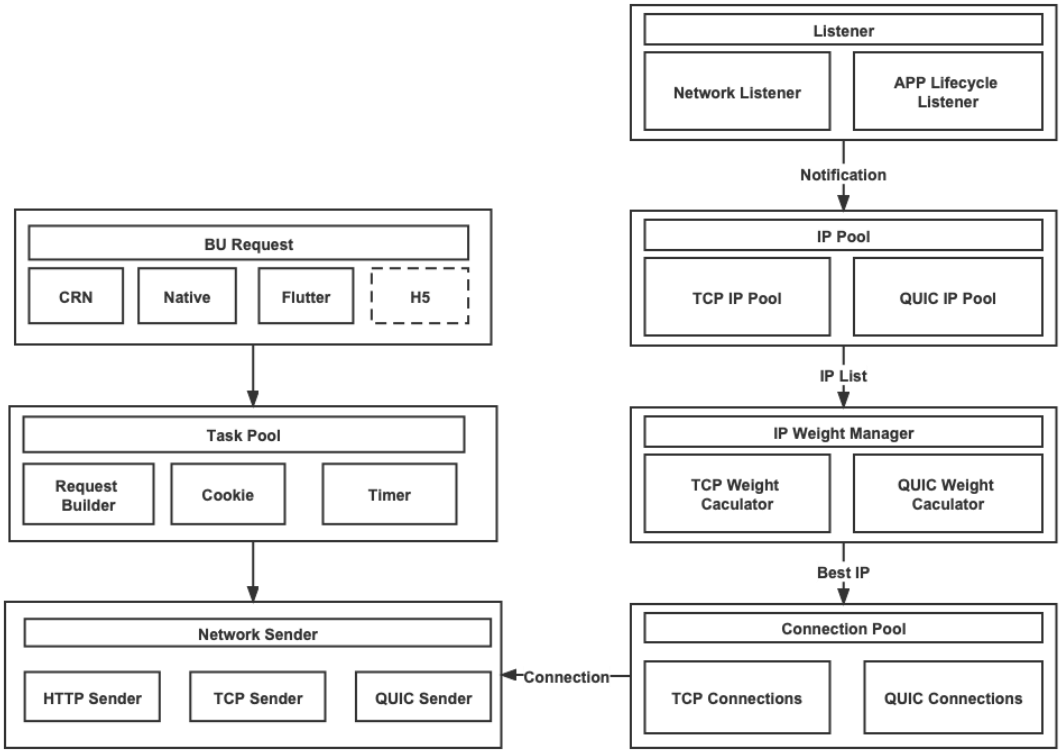
3.8 使用方式优化

在生产实验中我们发现，QUIC 并不合适所有的网络状况，所以我们不是用 QUIC 完全的替换原有 TCP 框架，而是做了有机融合，择优使用。

以下是两种比较常见的不支持 QUIC 场景：

- 某些办公网 443 端口会直接禁止 UDP 请求；
- 某些网络代理类型不支持 QUIC。

为了能适配各种网络环境，保证请求成功率，我们对 QUIC 的使用方式也进行了优化。



上图是目前 Trip.com 客户端的网络框架，APP 启动或网络变化时会通过一定的权重计算，选择最优的协议 (TCP 或 QUIC) 进行使用，并且进一步选择最优的 Server IP 预建立链接池，当有业务请求需要发送时，从链接池里选择最优链接进行发送。

改造后的使用方式充分利用了 TCP 和 QUIC 在不同网络环境下的优势，保证了用户请求的成功率，并能在各种复杂的网络环境下取得最佳的发送速度。目前 Trip.com APP 80%以上的网络请求通过 QUIC 进行发送，私有 TCP 协议和 Http2.0 作为补充，整体的成功率和性能得到了很大的提升。

四、总结和规划

由于 QUIC 具有精细的流量控制，可插拔式的拥塞算法，0 RTT，链接迁移，无队头阻塞的多路复用等诸多优点，已经被越来越多的厂商应用到生产环境，并取得了非常显著的成果。

我们也通过一年多的实践，深入了解了 QUIC 的优点和适用场景，并通过定制化的改造使得网络性能得到了极大的提升。但由于配套不完善，目前市面上所有的 QUIC 都无法达到开箱即用的效果。所以我们也希望贡献自己的力量，尽快开源改造后的整套网络方案，能让开发者可以不进行任何改动就能体验到 QUIC 带来的提升，实现真正的开箱即用。请大家持续关注我们。

架构

万字长文详解携程酒店订单缓存 & 存储系统升级实践

【作者简介】荣华，携程高级研发经理，专注于后端技术项目研发管理。军威，携程软件技术专家，负责分布式缓存系统开发 & 存储架构迁移项目。金永，携程资深软件工程师，专注于实时计算，数据分析工程。俊强，携程高级后端开发工程师，拥有丰富 SQLServer 使用经验。

前言

携程酒店订单系统的存储设计从 1999 年收录第一单以来，已经完成了从单一 SQLServer 数据库到多 IDC 容灾、完成分库分表等多个阶段，在见证了大量业务奇迹的同时，也开始逐渐暴露出老骥伏枥的心有余而力不足之态。基于更高稳定性与高效成本控制而设计的订单存储系统，已经是携程在疫情后恢复业务的必然诉求。

目前携程酒店订单系统正面临着在业务高增长的同时信息读写管理能力受制于数据库自身性能与稳定性的窘境。综合分析，一则为携程服役了十多年的 SQLServer 服务器集群的磁盘容量设计，已经跟不上时下新增订单量的空间诉求；二则在系统能力提升上造成了各大业务系统巨大的底层瓶颈与风险，同时又相比业界主流已基于 MySQL 架构设计存储系统而言，我们的订单存储系统仍基于 SQLServer 构建也整体推高了运营成本。

为了支撑未来每日千万级订单的业务增长目标，同时满足高可用、高性能、高可扩展的高效成本控制期望，我们为酒店部门的订单 DB 所有访问开发并落地了一套稳定且可靠的统一中间件封装方案，对现状收敛并提供了全局统一的热点缓存系统，彻底解决了当下订单上层应用与数据库间直连的方案缺陷。

新系统由中间件服务统一实现了对上层应用提供数据链服务，并达成了为现有依赖订单库的应用以及其他直接或间接的数据应用无感的实现存储底层由 SQLServer 向 MySQL 技术架构迁移的目标。

一、架构综述

通过对现有系统瓶颈的分析，我们发现核心缺陷集中在订单数据缓存分散导致数据各端不一致，各订单应用则与数据库直连又造成可扩展性差。通过实践我们编写中间件抽象并统一了数据访问层，以及基于数据库部署架构镜像构建了订单缓存统一管理热点数据，解决了各端差异。

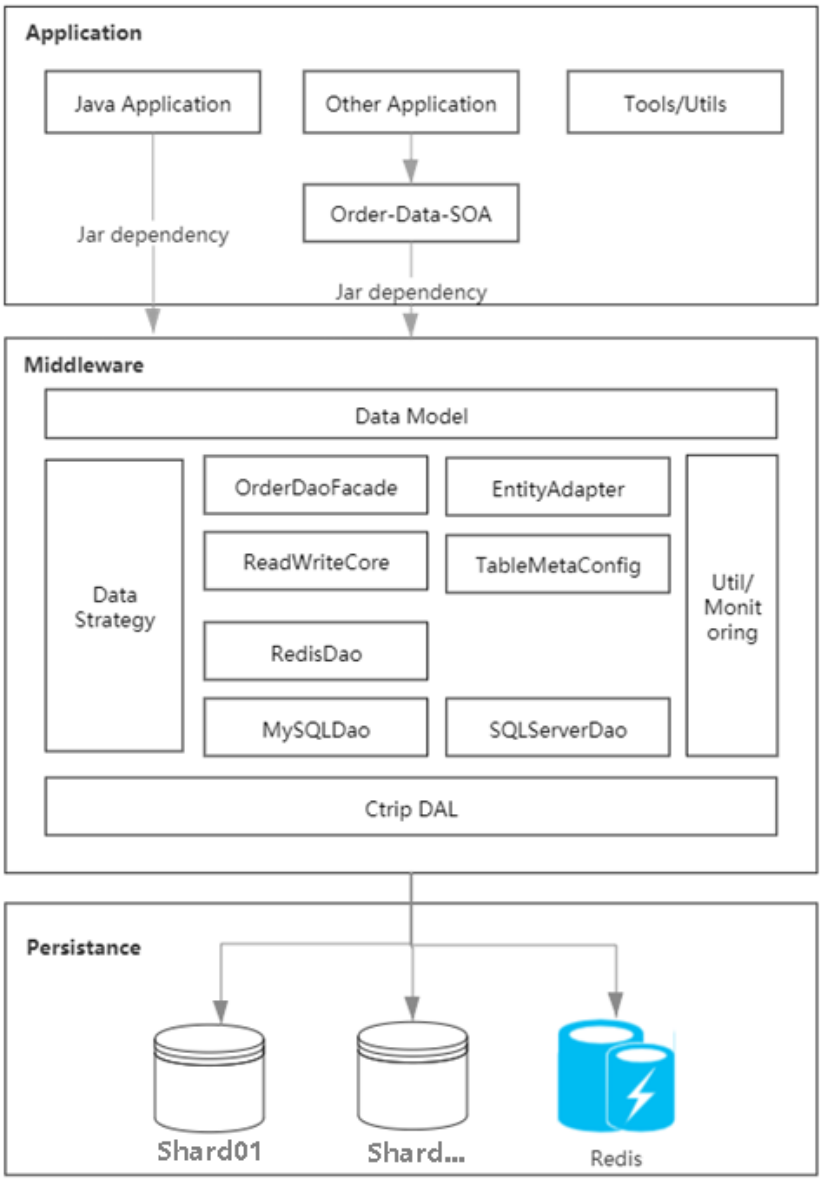


图 1.1 存储系统架构图

2.1 新单秒级各端同步

从订单的提交到各端可见的速度为存储服务的核心指标之一，我们对数据链的主要环节进行了优化，覆盖了新单同步、消息实时推送、查询索引构建以及数据平台离线归档等主要环节，使大系统内数据到达速度在 3 秒以内，即用户刚下完单即可跳转我携列表可见。

- 当新用户创单时，同步服务作为数据链入口将用户订单数据通过中间件写入订单库，此时中间件同时完成订单缓存的构建；
- 当订单完成入库行为和热点数据构建后抛订单消息，实时输出给各子系统；
- 当新单入库完毕即刻构建订单明细信息的 ES 索引，为第三方提供检索支持；
- 最后数据平台 T+1 实施当日数据的归档供 BI 等各类离线业务使用。

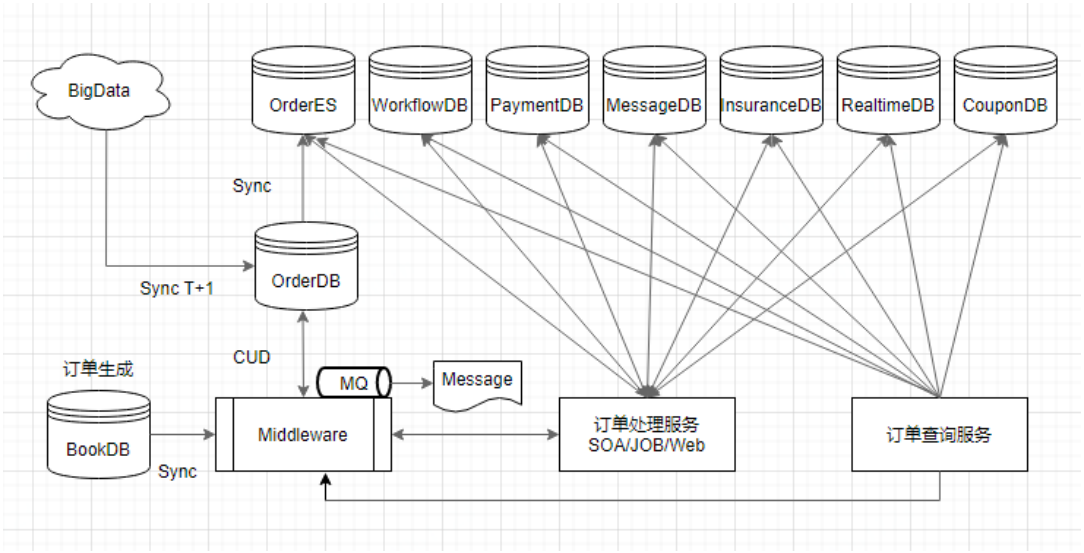


图 2.1 数据链

2.2 自动发单与工作台

对客、商、员工工作台三端的支持是订单存储系统的基本角色，图 2.1 数据链在新单提交后为自动发单与工作台起到的衔接作用功不可没。自动发单即在客人提交订单后，以最快的响应速度向商户发送订单明细信息进行核实货位、确认订单等流程。工作台则协助员工介入流程及时获取订单处理人工事件。

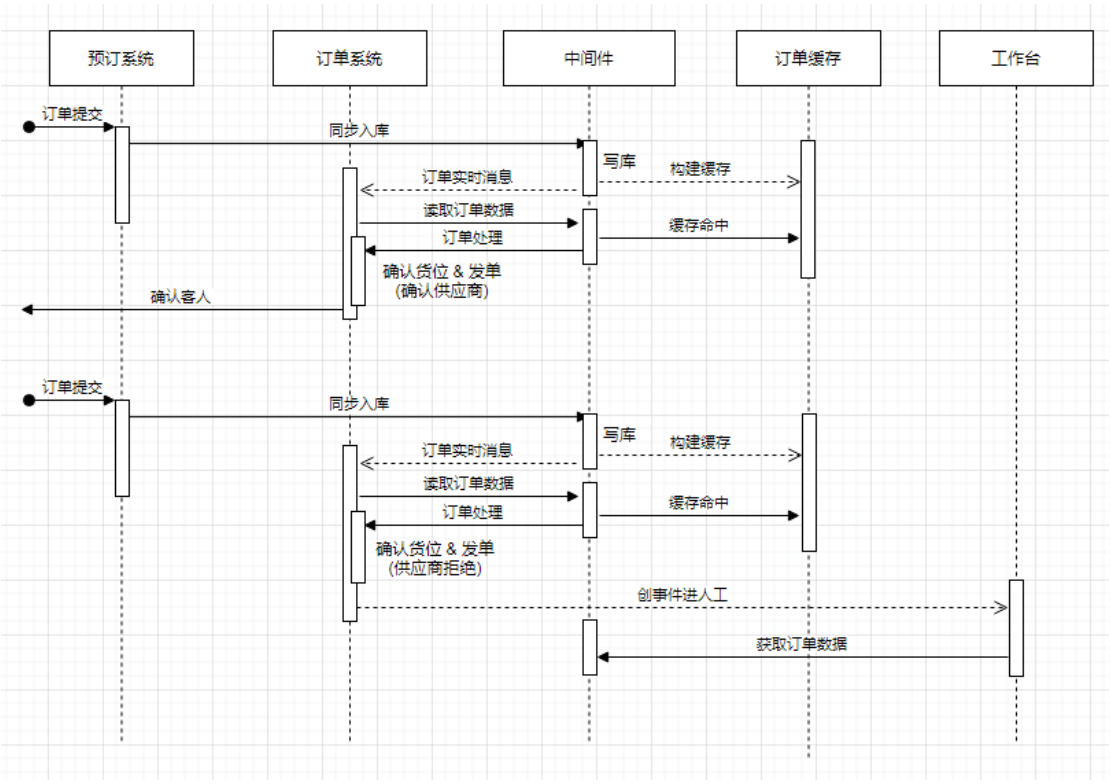


图 2.2 基于存储系统的发单与工作台关系（缩略细节）

2.3 查询与数据分析

基于订单数据为核心的主要分为在线查询和数据分析两条业务线，以对详情查询为例，访问 QPS 终年保持在高位，每逢假期高峰则容易造成查询瓶颈，根因复盘后在本次架构升级中我们做了调整来优化相关场景的高可用性。

在线查询以订单缓存为主，订单提交即构建热点缓存纾解查询压力，并可按配置时间参数长时段有效。

非在线查询场景，以实时消息推送并结合 Hive 数仓 T+1 方式交付，凡需要长周期订单数据的场合（例如实时报表）均接入订单消息实时计算。离线 BI 按年度等大批量数据分析时使用 Hive 表，并每日凌晨低峰时段以从库低频访问的方式实施数据同步。

如此以上，我们将订单主库的访问保护在订单缓存、实时消息、Hive 数仓三架马车之后，与业务尽最大可能的解耦。

三、系统升级实践

在对携程核心存储系统进行更新换代的过程中，贯穿全程需要做到的是热迁移，并达成所有操作对数据链路上的各应用透明无损的目标。我们的设计通盘分析了集团数据链路的特性，由订单缓存系统提供数据库镜像降低应用与数据库的直连耦合，继而再通过中间件对应用透明掉数据源于 SQLServer / MySQL 的物理关系，提供底层热迁移的操作空间。

结合无损迁移的工艺设计，注重对每一笔数据库流量的可见及可控，支持全库、Shard 级、表级、CRUD 操作级的流量分配策略，提供了底层数据迁移足够的实施手段。数仓衔接设计则侧重于解决数据平台百亿级离线数据与双库在线期间的同步问题，以及解决全量接入 MySQL 期间产生的数据问题。

以下将分三个部分分享我们在这一过程中学到的经验。

3.1 分布式订单缓存

随着业务发展，用户数和访问量越来越大，订单系统应用和服务器的压力也与日俱增。在没有引入订单缓存之前，每个应用独立连接数据库，造成查询出来的数据无法在应用间共享，并且 DB 每秒查询量和连接数都有上限，而酒店核心交易链路基于 DB 存储，存在单点故障风险。

经过埋点数据分析，订单系统是典型的读多写少，为了共享热点查询数据以及降低 DB 负载，一个有效的办法就是引入缓存，如图 3.1，用户的请求过来时，优先查询缓存，如果存在缓存数据，则直接返回结果；缓存没有命中，则去查询 DB，根据配置策略校验 DB 结果数据，校验通过则将 DB 数据写入缓存留作后续查询使用，否则不写入缓存，最后返回 DB 查询结果。

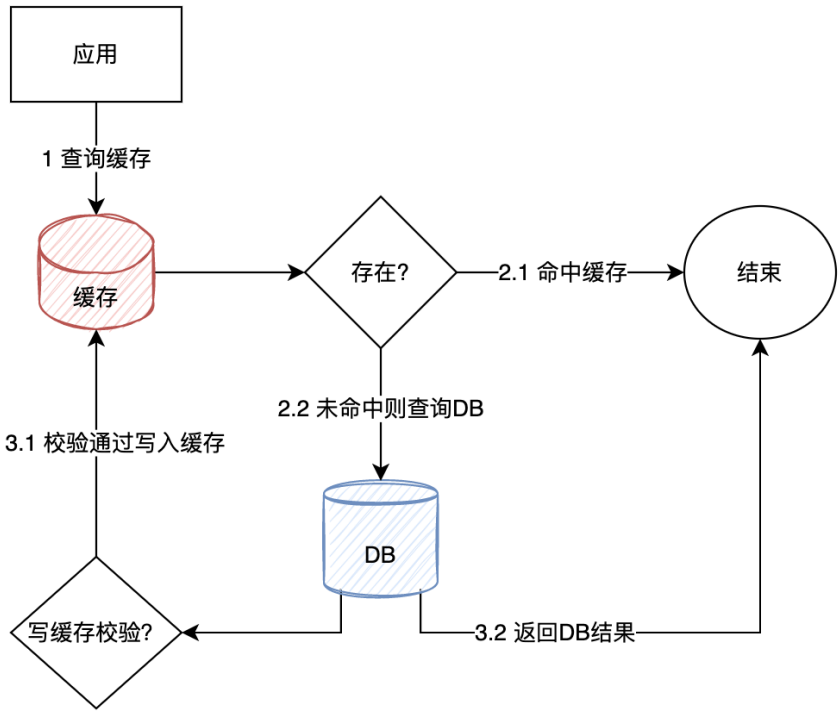


图 3.1.1 单缓存基本设计

关于引入新的缓存组件后的硬件开销，可通过收敛原来各应用分散的硬件资源来降低总成本，但还会因为中心化管理带来可用性挑战以及数据一致性等问题，故需要充分对现有系统进行容量评估、流量估算和缓存表价值分析。只缓存访问量高的热点数据表，通过恰当的缓存结构设计、数据压缩和缓存淘汰策略，最大程度提高缓存命中率，在缓存容量、硬件成本和可用性之间做好权衡。

传统的缓存设计，是一条数据库表记录对应一条缓存数据。而在订单系统中，一个订单查询多表的场景很常见，如果采用传统设计，在一次用户查询中，Redis 的访问次数是随着表数量增加的，这种设计网络 IO 较大并且耗时较长。在盘点表维度流量数据时，我们发现有些表经常一起查询，不到 30%的表其查询流量超过 90%，在业务上完全可以划分为同一个抽象领域模型，然后基于 hash 结构进行存储，如图 3.2，以订单号作为 key，领域名称作为 field，领域数据作为 value。

这样无论是单表还是多表查询，每个订单都只需要访问一次 Redis，即减少了 key，又减少了多表查询次数，提升了性能。同时 value 基于 protostuff 进行压缩，还减少了 Redis 的存储空间，以及随之而来的网络流量开销。

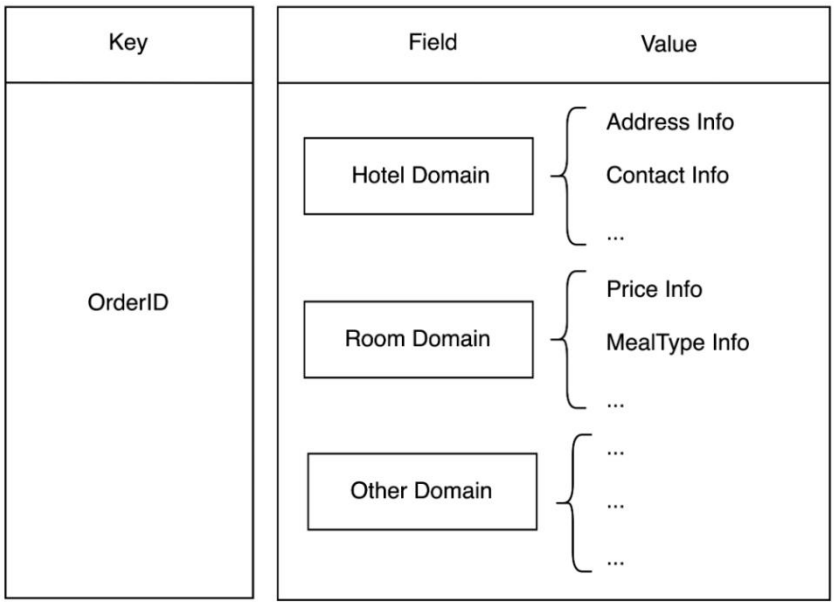


图 3.1. 2 基于 domain 的存储结构简述

3.2 无损迁移工艺

如何做到无损热迁移是整个项目最具挑战性的地方。在工艺设计之前我们的前置工作首先完成了中间件的开发，通过中间件将数据库与业务层应用一分为二。其次抽象 Dao 层实现领域化，并由数据领域层向应用提供数据服务，领域之下适配 SQLServer 和 MySQL 两种数据库并统一封装。以此为基础才能委以下述工艺设计实施无损热迁移。

SQLServer 和 MySQL 双库在线，实施双写，主写 SQLServer，同步副写 MySQL，如果 SQLServer 操作失败则整体失败，回滚双写事务。

SQLServer 和 MySQL 之间增加一路同步 Job，实时查询 SQLServer 最近时间窗口变更的数据进行一致性校验 MySQL 中的条目，差异点追齐，可以确保双写期间不可预期的两边不一致，特别是还残有直连写 SQLServer 应用的阶段特别有用。

中间件设计有配置系统，支持任一主要查询维度可按配置精准的将数据源定向到 SQLServer 或 MySQL，并可控制是否读取后加载到订单缓存。初期设定只加载 SQLServer 数据源，避免因双库间的数据不一致而造成缓存数据跳跃。并在初期可设置灰度，将小批量非核心表直连 MySQL 验证可靠性。后期数据一致性达成预期后，订单缓存也可自由按指定数据库加载缓存。

解决了查询场景下的数据一致性问题后，流量策略支持图 3.2 中任一可调控维度进行数据库单写。实际项目中以表维度实施单写为主，当指定表被配置单写 MySQL 后，所有涉及该表的 CRUD 行为全部定向 MySQL，包括缓存加载源。

最后通过中间件统一收口对外发送的订单消息，所有消息基于中间件的 CUD 操作发送与物理数据库无关，这样实现消息的数据源透明，且可联动以上所有工艺操作，数据链保持一致。

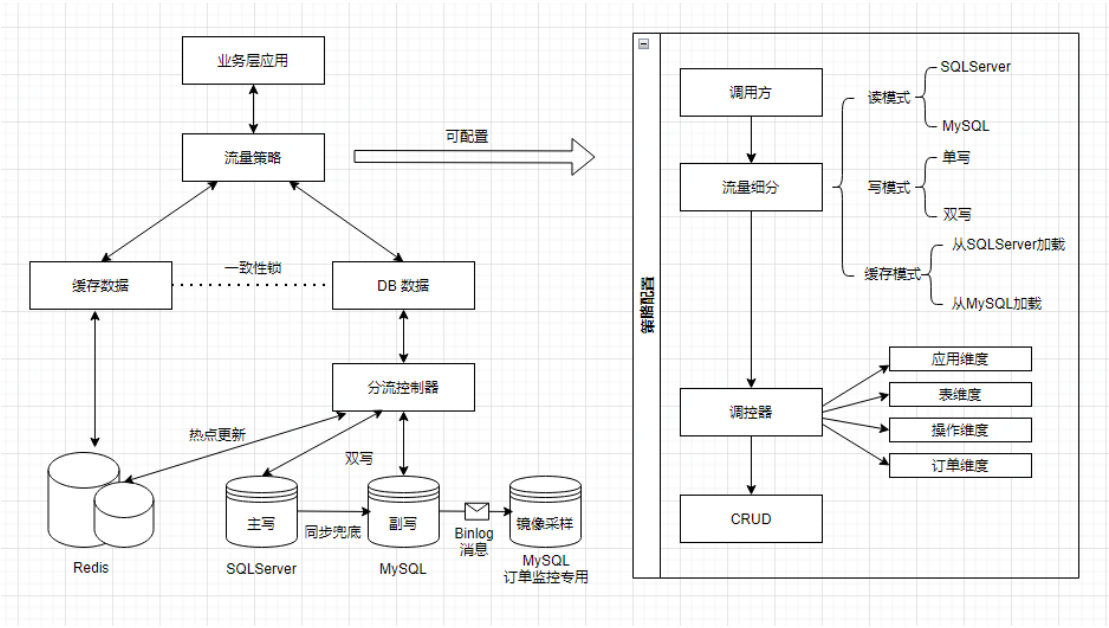


图 3.2　操作工艺简介

3.3　数仓衔接

为了方便理解生产数据到数据仓库 ODS 层数据的迁移，做到对下游透明，这里简单介绍一下常规数据仓库的分层体系。通常数据仓库主要分为五层：ODS(原始数据层)、DIM(维度)、EDW(企业数仓)、CDM(通用模型层)、ADM(应用模型层)，如下图所示：

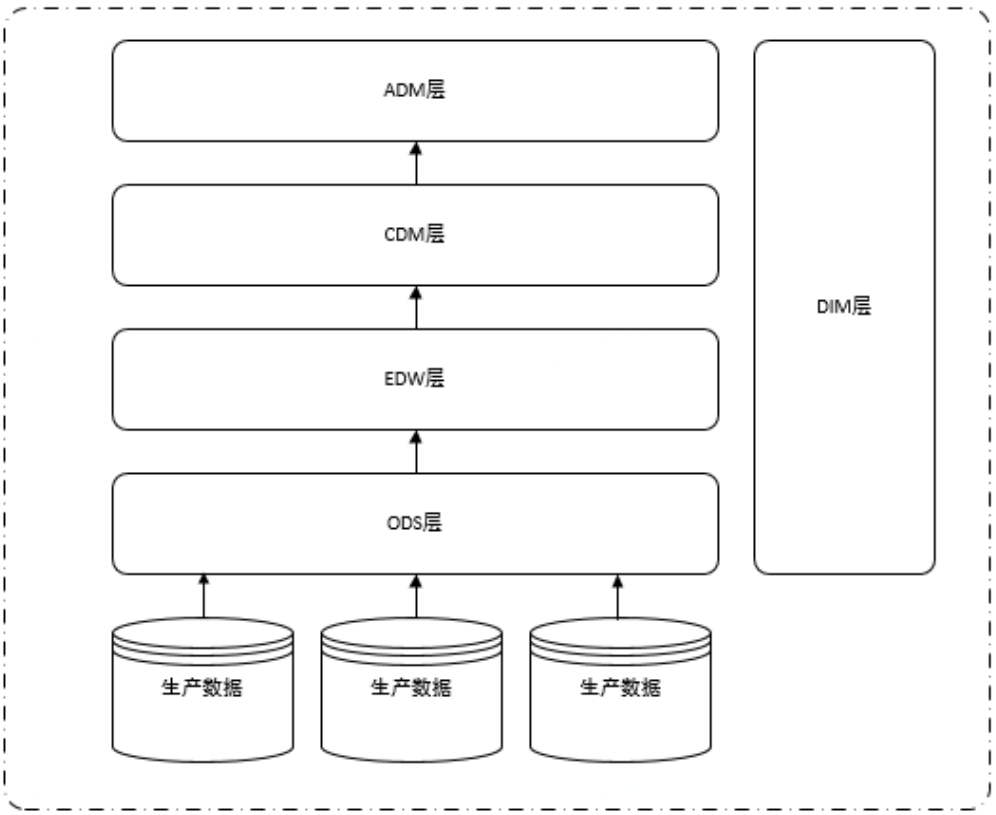


图 3.3.1　数据仓库分层结构

从图 3.3.1 上可以看出，数据仓库各层都依赖 ODS 层的数据，为了不影响数据平台所有应用，我们只需要将原来订单库 ODS 层数据源从 SQLServer 迁移到 MySQL 库即可。

从图上很直观的看出，迁移只需换个数据源不是很麻烦，但是为了保证数据质量，我们做了很多的前置工作，比如：DBA 预先将生产数据同步到生产 MySQL 库、MySQL 数据实时同步、生产两侧数据一致性校验、MySQL 侧数据同步到 ODS 层、ODS 层数据一致性校验及原有 ODS 层同步 Job 数据源切换等。

其中，生产两侧数据一致性校验和数据仓库 ODS 层数据一致性校验最为复杂，耗时也最长，要确保每张表、每个字段都要一致时才能切换数据源。但是，从实际操作过程中，却做不到完全一致。根据实际情况，适当处理时间类型、浮点值精度及小数位等。

下面介绍一下整体流程：

首先，对于线上数据一致校验，我们开发了在线同步 Job，将 SQLServer 的数据和 MySQL 数据进行比较，发现不一致时，就将 MySQL 的数据以 SQLServer 数据为基准更新掉，确保两边数据的一致性。

其次，对于离线数据一致性校验，我们和数据仓库同事合作把 MySQL 侧数据同步到 ODS 层(以库名区分是 SQLServer 还是 MySQL 的表)，并且将定时跑的任务和 SQLServer 侧任务在时间上尽量一致。两侧数据都准备好后，我们开发了离线数据校验脚本生成器，根据数据仓库元数据，为每张表生成一个同步 Job，将其部署到调度平台。

同步任务会依赖两侧 ODS 层同步数据，T+1 数据同步完成后，执行一致性校验，将不一致的订单号记录到不一致明细表中，并统计不一致的数据量，将结果保存到统计表中。然后在自助报表平台制作一个报表，将每天统计的不一致的表及不一致量发送到邮箱，我们每天对不一致的表进行排查找出问题，调整比较策略，更新比较 Job。大致流程如下：

图 3.3.2　一致性校验整体流程

最后，随着线上和离线数据逐步趋于一致后，我们将原先 SQLServer 同步到 ODS 层 Job 的数据源切换到 MySQL。这里可能有同学会有疑问：为什么不直接使用 MySQL 侧 ODS 层的表呢？原因是，经过统计，依赖原先 ODS 层表的 Job 有上千个之多，如果让依赖 Job 切换到 MySQL 侧 ODS 表，修改工作量非常大，所以我们直接将原来的 ODS 层同步数据源直接切换成 MySQL。

实际操作中，切数据源并不能一次全部切完，我们分三批进行，先找十几个不那么重要的表作为第一批，切完后运行两周，并收集下游数据问题的反馈。第一批表顺利切完两周后，我们没收到下游报数据问题，说明数据质量没问题。然后再将剩余的几百张表按重要程度分两批继续切，直到切完。

至此，我们完成了订单库从 SQLServer 迁移到 MySQL 在数据仓库层的迁移工作。

四、核心问题精编

实际上再周密的分析与设计，总是难免遇到执行过程中的各种挑战。我们总结了一些经典问题，虽然通过技术手段最终解决了这些大大小小问题并达成了目标，但是相信各位看官必定还有更好的解决方案，我们乐见共同学习与进步。

4.1 SQLServer & MySQL 流量迁移如何细粒度监控

订单系统涉及到的应用和表数量众多，一个应用对应 1 到 n 张表，一张表又对应 1 到 n 个应用，是典型的多对多关系。如图 4.1，对于上层应用来说，从一个 SQLServer 数据库，切换到另一个 MySQL 数据库，其基本流程参照操作工艺章节至少分为以下几步：

- 从单写 SQLServer 变成双写 SQLServer 和 MySQL
- 从单读 SQLServer 变成单读 MySQL
- 从双写 SQLServer 和 MySQL 变成单写 MySQL
- 下线 SQLServer

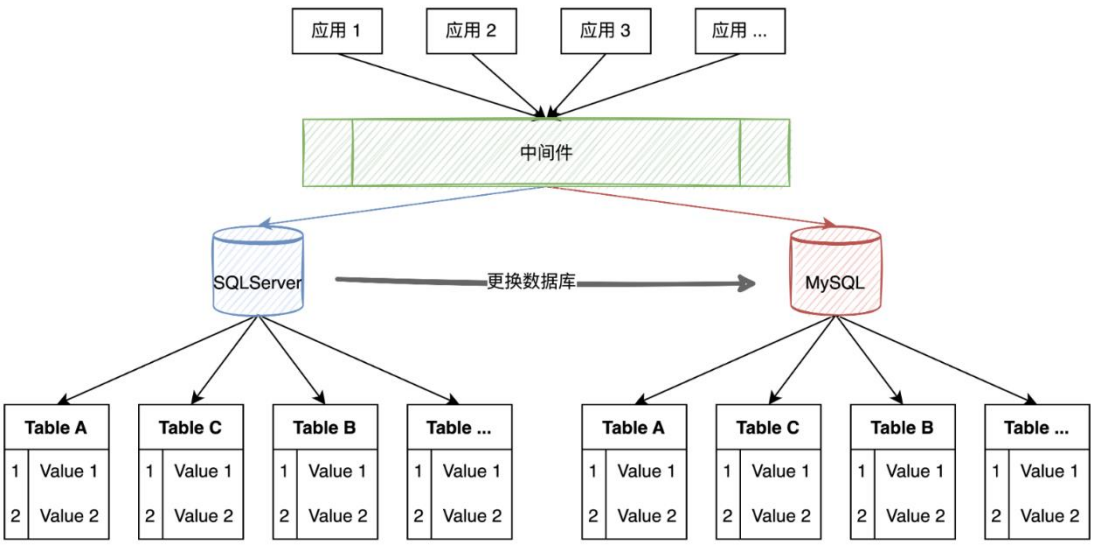


图 4.1 应用和数据库以及表的关系图

在生产环境更换数据库系统，就像在高速公路上不停车换轮胎，需要维持原有的车速不变，且对用户无感，否则后果不敢设想。

在切换工艺中双写、单读和单写流程，环环相扣，步步相依，作为配套设计监控手段必须确认上一个操作达到预期效果才能进行下一个。如果跳过或者没有切换干净就贸然进行下一步，比如还没有双写完全一致，就开始读 MySQL 数据，可能造成查无此数据或者查到脏数据！那么就需要对每一个 CRUD 操作的读写进行监控，在迁移过程中做到 360 度无死角可视化流量细分控制，所见即所得。具体的做法如下：

- 所有应用接入中间件，CRUD 由中间件根据配置控制读写哪个 DB 的哪张表；
- 每一个读写操作的详细信息均写入 ES，在 Kibana 和 Grafana 上可视化展示，并且通过 DBTrace，可以知道每条 SQL 是在哪个 DB 上执行；
- 按照应用级别逐步配置双写 DB，通过同步 Job 实时比对、修复和记录两侧 DB 差异，再通过离线 T+1 校验双写中出现的最终不一致，如此往复直到双写一致；
- 双写一致之后，就开始逐步将读 SQLServer 切换到读 MySQL，通过 ES 监控和 DBTrace 确认完全没有 SQLServer 读，则表明单读 MySQL 完成，考虑到自增主键情况，我们采取按照表维度，按批次断写 SQLServer，直至所有表都单写 MySQL。

综上所述，基本方案为通过中间件为管道为所有接入的应用统一埋点，通过实时展示应用层的行为观察流量分布，并结合公司数据库侧 Trace 的可视化工具核实应用的流量切换行为与数据库实际 QPS 及负载浮动保持一致来监督迁移任务。

4.2 如何解决双写期间 DB 一致性问题

酒店的订单库有着二十年左右历史，经年累积，跨部门和酒店内部多个团队直接或间接依赖订单库 SQLServer，要想切换到 MySQL，就得先解决双写 DB 一致性问题，不一致主要体现在以下两点：

- 双写时实际仅单写了 SQLServer，漏写 MySQL；
- 双写 SQLServer 和 MySQL 成功，并发、不可靠网络、GC 等发生时 MySQL 数据有几率和 SQLServer 不一致；

关于双写数据一致性的保证，我们基于同步 Job 将 SQLServer 数据为准线，根据最后更新时间，拉取两侧 DB 数据进行比对，如果不一致则修复 MySQL 的数据并将不一致信息写入 ES，供后续排查根因。

但也因为引入了额外的 Job 操作 MySQL 数据，带来了新的问题，那就是多表双写时，因为耗时翻倍，Job 发现 SQLServer 有数据而 MySQL 没有，就立即修复了 MySQL 数据，造成双写失败。所以双写部分失败又加上了 Failover 机制，通过抛送消息，触发新一轮的比对和修复工作，直到两侧 DB 数据完全一致。

同步 Job 和 Failover 消息机制虽然可以让数据最终一致，但毕竟有秒级的间隔，两侧数据是不一致的，并且对于众多应用的各种场景，难免会有遗漏时单写 SQLServer。对于这些漏写 MySQL 的地方，通过 DBTrace 是无法找到的，因为无法确定一个 CUD 操作只写入 SQLServer，而未写入 MySQL。那么有没有办法事前就能找出漏写 MySQL 的场景呢，确实被我们找出来一点，那就是更换数据库连接串，接入中间件的应用使用新连接串，然后找出所有使用旧连接串操作 SQLServer 的 SQL，就能准确定位出漏写 MySQL 的流量了。

最终，我们将双写 DB 不一致率从十万分之二逐步降低到了几乎为 0，为什么是几乎呢，因为 DB 的一些特性差异问题，会天然的导致数据无法完全一致，这个在后续内容会有详细的论述。

4.3 引入订单缓存后导致的数据不同步问题处理

引入缓存之后，就涉及到对缓存进行写入或者更新，业界常见的做法分为以下几种：

- 先写 DB 再写缓存
- 先写缓存再写 DB
- 先删缓存再写 DB
- 先写 DB 再删缓存

在具体实施上还会进行双删缓存或者延迟双删缓存，此处不再比较各种做法的优劣。我们采用的是先写 DB 再删缓存方案，对于数据敏感表，会进行延迟双删，后台的同步 Job 定时比对、修复和记录数据库数据与 Redis 数据的差异，虽然设计上已经能保证最终一致性，但是在前期还是出现过大量的数据不一致。主要体现在以下几个方面：

- 应用有场景未接入中间件，对 DB 进行 CUD 操作之后，漏删除缓存；
- 写 DB 后删除缓存延迟导致读取到缓存脏数据，比如不可靠网络、GC 等造成删缓存延迟；
- 写 DB 后删除缓存失败导致读取到缓存脏数据，比如 Redis 主从切换期间，只能读不可写；

而为了解决缓存一致性问题，如图 4.3，我们在原有的缓存和 DB 基础上，增加了乐观锁和 CUD 施工标记，来限制并发情况下同时存在加载数据到缓存相互覆盖的行为，以及对当前被查数据正在进行 CUD 操作的感知。在此两种场景未结束期间可以做到 Query 流量直连 DB，通过基于乐观锁的最后写入者获胜机制解决竞争问题。最终我们的缓存不一致率从百万分之二控制到了千万分之三。

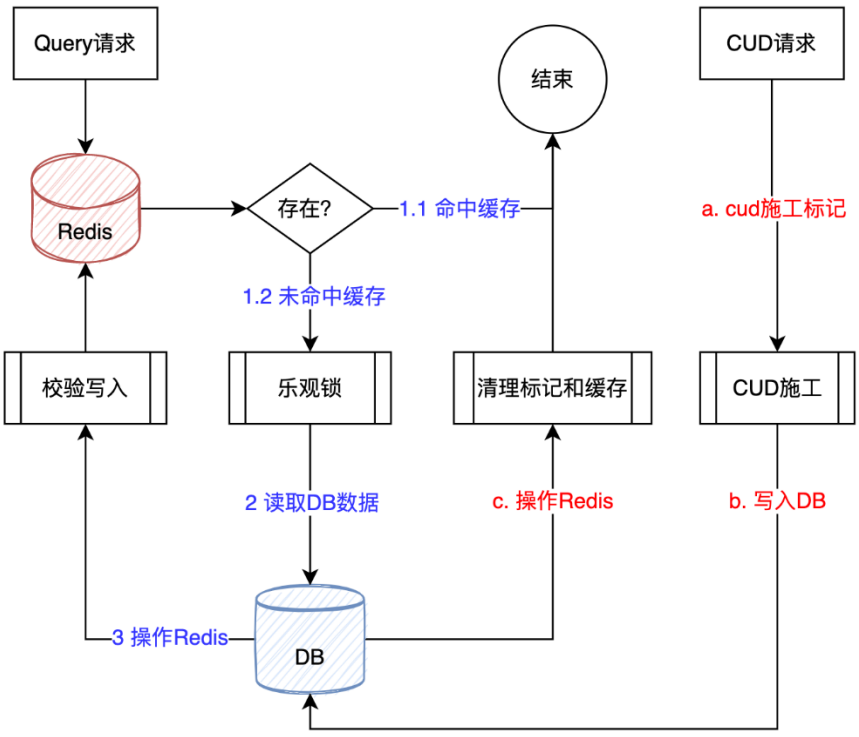


图 4.3 缓存一致性解决

注：图 4.3 当查询未命中缓存，或当前存在该数据的乐观锁或施工标记时，当次查询直连 DB，直至相关事务完成后放开缓存数据自动加载功能。

4.4 存量订单数据如何一次性校准

项目启动初期我们对 MySQL 进行了最近 N 年数据的一次性铺底，这就产生了在双写阶段无法校准的如下两个场景的数据：

- 因生产上订单库预置保留近 N 年的数据，负责清理备份的 Job 在接入中间件前，MySQL 已存在的 N 年外的这批数据无法被策略覆盖而清理掉。
- 所有应用接中间件花了很长时间，接中间件双写前数据有可能不一致的，需要全部应用接中间件和全部表双写后，对之前的数据进行一次性修复。

针对第一点，我们开发了 MySQL 数据专项清理 Job，由于订单数据库是多 Shard 的，Job 内部根据实际 Shard 数设置核心线程总量，每个线程分别负责对应 Shard 中的指定表进行清理，并行开多台服务器分发任务进行清理，通过速度控制既保证了效率又不影响生产上数据库的负载。

针对第二点，在所有应用接中间件和所有表实现双写后，通过调整线上同步 Job 扫描的开始时间戳，对存量订单数据进行修复。修复时特别注意的是，扫描数据要按时间段分片处理，防止加载数据太多导致订单库服务器 CPU 太高。

4.5 一些数据库特性差异问题

在如此庞大的系统下进行数据库热迁移，我们就必须了解不同数据库之间的差异与不同，做到知己知彼，对症下药。MySQL 与 SQLServer 虽同为时下流行的关系型数据库，均支持标准化 SQL 查询，但在细枝末节上还是有些许差异。下面我们通过迁移中所面临的问题来具体分析一下。

自增键问题，为保证数据自增序号一致，不能让两个数据库各自去进行自增，否则一旦不一致就要面临修数据甚至更大风险。因此，在数据双写时，我们将 SQLServer 写入后生成的自增 id，回写入 MySQL 自增列，在数据单写 MySQL 时直接使用 MySQL 生成自增 id 值。

日期精度问题，双写后为了保证数据一致性，要对两侧数据进行一致性校验，类型为 Date、DateTime、Timestamp 的字段，由于保存精度不一致，在对比时就需要做特殊处理，截取到秒进行比较。

XML 字段问题，SQLServer 中支持 XML 数据类型，而 MySQL 5.7 不支持 XML 类型。在使用 varchar(4000)代替后，遇到 MySQL 数据写入失败，但同步 Job 将 SQLServer 数据回写 MySQL 时又能正常写入的案例。经过分析，程序在写入时会将未压缩的 XML 字符串写入，SQLServer XML 类型会自动压缩并存储，但 MySQL 并不会，导致长度超过 4000 的写入操作失败，SQLServer 压缩后长度小于 4000，又能够正常回写 MySQL。为此我们提出应对措施，写入前压缩并校验长度，非重要字段截取后再存储，重要字段优化存储结构或更换字段类型。

下面列举一些迁移过程中常见的注意点。

SQLServer	对应MySQL
使用IDENTITY自增	使用AUTO_INCREMENT自增
MONEY、SMALL MONEY类型	DECIMAL(19,4)、DECIMAL(10,4)类型
UNIQUEIDENTIFIER类型	BINARY(16)类型
串联运算符+或||	CONCAT('string1', 'string2')函数
日期函数GETDATE()	NOW()、CURRENT_TIMESTAMP()
日期函数DATEADD()	ADDDATE()
Top子句	使用Limit
VARCHAR(n)可存储n/2个汉字	VARCHAR(n)可存储n个汉字

五、预警实践

我们的预警实践并不局限于项目推进期间的监控诉求，如何在百亿级数据中周期扫描数据写入的异常，完成项目期间双写数据一致率的复核，如何实时监控与预警订单库每个分片上订单写入量的正常趋势，如何定期验收/核验整套系统的高可用性将在以下篇幅中描述。

5.1 百亿级数据差异校验预警

要满足订单数据 SQLServer 迁移到 MySQL 库，数据质量是迁移的必要条件，数据一致性达不到要求就无法透明迁移，所以设计合理的校验方案，关乎迁移的进度。针对数据校验，我们分为线上和线下两种：

线上数据校验和预警：迁移期间我们通过同步 Job，在计算出不一致数据后，将不一致的表及字段写入 ElasticSearch，再用 Kibana 制作出不一致数据量及不一致表所占比例的监控看板，通过监控看板，我们就可以实时监控哪些表数据不一致量比较高，再根据表名称通过 DBA 工具排查出哪些应用对表进行了 CUD 操作，进一步定位漏接中间件的应用和代码。

在实际操作中，我们确实找出了大量未接中间的应用并对其改造，随着接入中间件的应用越来越多，数据一致性逐渐提高，从监控看板上看到不一致的量也慢慢降低。但是一致性始终没有降低到零，原因是应用和同步 Job 并发导致的，这个也是最令人头疼的问题。

或许有同学会疑问，既然双写了为什么不停止掉同步 Job 呢？原因是双写以 SQLServer 为主写，以受中间件覆盖的 CUD 范围为基准，除了不能保证写入 MySQL 的数据百分百成功外也不能保证两库的数据量相等，所以需要一致性 Job 兜底。由于并发的存在，虽然做不到数据百分百一致，但是可以进一步降低。

我们的做法是，一致性 Job 比较时设置一个 5 秒的稳定线（即距离当前时间 5 秒内的数据视为不稳定数据），订单数据时间戳在稳定线内的不进行比较，稳定线外的比较时，会再一次计算订单数据是否在稳定线内，如果确认全部数据在稳定线外，就进行比较操作，否则放弃本次比较，由下一次调度执行一致性校验。

离线数据校验和预警：订单库迁移涉及到几百张表，离线数据比较多，一年的订单相关数据就有上百亿了，对于离线数据校验比较有挑战。我们编写了数据一致性脚本生成器，为每张表生成一个比较脚本并部署到调度平台，比较脚本依赖上游 SQLServer 和 MySQL 两侧的同步 Job，上游 Job 执行完毕后自动执行数据比较，将不一致数据的订单号写到明细表中，并根据明细表统计出不一致量，以日报的形式发出，每天对数据不一致比较高的表排查并解决。

通常一是能修复对比脚本的瑕疵，二是发现离线数据问题，就这样反复摸排解决不一致问题。对于离线数据每张表每个字段的校验是非常复杂的，我们编写 UDF 函数进行比较，UDF 函数功能也很简单，就是将每张表的非主键字段进行拼接生成一个新字段，两侧表进行全外连接，主键或者逻辑主键相等的记录，生成新字段也应该一样，只要不一样就视为不一致数据。这里要注意日期字段截取、数据精度及末尾为零的小数处理问题。

经过三个多月的努力，我们排查出所有未接中间件的应用，并将其 CUD 操作全部接入中间件，开启双写后线上线下数据一致性逐步提高，达到了迁移数据的目标。

5.2 ALL Shard 实时订单总量监控

每个公司对于订单量的监控是不可或缺的，携程有一个统一预警平台 Sitemon，它主要监控各类订单告警，包括酒店，机票，无线，高铁，度假。并能按照 online/offline，国内/国际，或者支付方式单独搜索和展现，并对各类订单做了告警。

订单数据从 SQLServer 迁移到 MySQL 期间，我们梳理出来依赖订单库的预警策略近两百个，负责监控的相关同事对 SQL Server 数据源的预警策略原样复制一份连接 MySQL 数据源。以 MySQL 为数据源监控告警都添加完成后，开启报警策略，一旦订单量异常报警，NOC 会收到两条通知，一条来源于 SQLServer 数据告警，一条来源于 MySQL 告警，如果两边一致，说明灰度验证通过。否则，不通过，需排查 MySQL 监控问题。

经过一段时间的灰度验证，两边报警数据一致，随着 SQLServer 数据表下线（即单写 MySQL 数据），以 SQLServer 为数据源的预警策略也跟着及时下线。

5.3 “流浪地球”实操

为了做好系统安全保障工作，提高应对突发事件的能力，必要的演练压测等是少不了的。为此，我们制定了完备的应急预案并定期组织开展应急演练——流浪地球。演练项目包括核心/非核心应用熔断、DB 熔断、Redis 熔断、核心防火墙、交换机应急切换等。

以缓存为例，为了保证缓存服务的高可用，我们在演练时会下线部分节点或机器甚至切断整个 Redis 服务，模拟缓存雪崩、缓存击穿等场景。按照计划，在熔断前我们会先切断应用的 Redis 访问，一步步降低 Redis 负载，然后熔断 Redis，以此检验在无 Redis 的情况下各应用

系统是否能够正常运转。

但在首次演练中，熔断 Redis 后应用报错量就急剧上升，果断停止演练回退并查找原因。经过分析，部分应用 Redis 操作未统一收口，不受中间件统一控制，Redis 熔断后应用随即出现异常。针对这一情况，我们分析后一方面将报错应用的订单缓存访问收口接入中间件，另一方面强化了中间件与 Redis 的弱依赖关系，支持一键断开 Redis 操作，并完善了各项指标监控。最终在第二次演练中顺利完成 Redis 熔断，各业务系统在全流量打入 MySQL 的状态下的正常运行。在最近一次的流浪地球演练中，机房网络阻断、非核心应用阻断等一轮轮故障注入后，我们的系统更是取得了很好的预期效果。

就这样，在一次次的演练中，我们发现问题，总结经验，优化系统，完善应急预案，一步步提升系统应对突发故障的能力，保证业务的连续性以及数据的完整性。做好底层数据支撑，为整个酒店订单系统保驾护航。

六、未来规划

6.1 订单缓存手工调控台

虽然我们有完善的监控看板与预警系统，但对于像熔断演练、自动化故障演练、硬件故障和维护以及不可提前预知的问题，若刚好核心开发人员未能及时在现场响应操作，系统尚不能完全自主降级可能导致部分性能有所下降，比如响应耗时增加等。在将来计划增加手工调控看板，授权后可以让 NOC 或者 TS 进行针对性操作，比如 Redis 全部或者部分集群宕机，可以一键切割故障 Redis 分片，或者根据 Redis 已计划中的不可用时间段来提前设置切割时间，可以最大程度保证系统的可控性。

6.2 中间件自动降级

既然可以手工进行调控，那么我们也考虑后续可以通过一些核心指标的监控，比如 Redis 主从切换期间，正常情况是秒级，但是我们也出现过部分 Redis 10 秒以上不可写的情况，此时可以监控缓存与数据库不一致的脏数据量，也可以在 Redis 发生故障时通过监控响应耗时异常的阀值来应用一些策略，让中间件自动降级切割掉这些故障主机保证服务的基本稳定，然后在探测到集群指标稳定后再逐步尝试恢复。

6.3 中间件接入 Service Mesh

当前订单团队内部是以 JAR 的方式使用中间件，由中间件来屏蔽数据库底层差异和操作 Redis 以实现更复杂的功能，天然具备接入 Service Mesh 能力，接入后底层升级更加快速和无感、调用更加轻量化、更好与框架进行网格化集成以及上云更加方便，能够更好的支撑携程的国际化战略目标。

携程 SOA 的 Service Mesh 架构落地

【作者简介】本文作者 Dozer、Bender、vio-lin 来自携程 SOA 团队。目前主要负责 SOA 系统的研发工作和 Service Mesh 架构的演进、落地工作，同时也关注服务治理、JVM、云原生等技术领域。

一、背景

携程的 SOA 系统经历了 ESB、微服务等架构的演变，正处于一个较平稳的阶段。但当前的微服务架构却遇到了各种业内经常遇到的问题，例如：

- 无法支撑多语言战略，团队没有精力维护除了 Java 以外其他语言的 SDK；
- 客户端 SDK 版本升级推进困难，特别是遇到 Bug 的时候，彻底下线一个版本可能会花上几个月的时间，给业务带来了隐患；

在 Service Mesh 架构出现时，我们就注意到了它。一边探索一边实践，尝试着用 Service Mesh 来解决我们的痛点。

二、技术方案

携程主营业务在国内，并且在国际上也有着不小的业务量。基于这个特点，国内使用自建机房、国外使用公有云的模式是非常合适的。

正因为技术栈需要支持跨机房部署，所以将云原生架构作为演进的目标。

Istio 作为云原生架构中重要的一位成员，和云原生架构中的其他成员相辅相成。

除此以外，携程当前 SOA 以 HTTP 协议为主，Istio 对 HTTP、HTTPS、gRPC 这几个传输协议有着全面的支持。

自然地，基于 Istio 做二次开发成为了我们内部推行 Service Mesh 的技术实现方案。

虽然开源项目开箱即用，但在调研和方案设计的过程中，我们也遇到了很多问题，例如：

- 如何实现不改造业务代码即可接入 Service Mesh，做到无感知迁移
- Istio 无法覆盖所有携程需要的功能
- Istio 没有配置按需下发方案
- Istio 性能无法支撑携程规模
- Istio 对高可用方面的支持不够

接下来，本文会着重介绍 1-3 的解决思路，4-5 可以参考我们另外两篇文章：

- 携程 Service Mesh 性能优化实践

- 携程 Service Mesh 可用性实践

三、控制平面

控制平面想要实现无感知迁移，那么最重要的就是要实现两套系统的互通，其中主要包括：统一配置管理、服务注册与发现、功能对齐和 SDK 兼容。

3.1 统一配置管理

我们现有的 SOA 系统已经有了一套包括管理后台、实时推送等功能模块的系统，并不需要再造一套。

但从云原生架构的角度看，这样的设计就不够云原生。云原生的架构下更推崇声明式 API 和 GitOps 工作流。我们现有的系统显得有点落伍，对将来上公有云也不太友好。

所以这里就出现了两个方案：

- 方案一：用户依然在现有系统中操作，通过 Operator 转换成 Istio 的配置；
- 方案二：上线后一次性将现有配置全部迁移到 Istio 配置，编写适配器给原系统使用者调用，用户可以用 GitOps 工作流也可以用原有的管理后台操作。

团队内部对这两个方案也争论了很久，两者各有利弊。

最终我们还是采取了先用当前配置为权威的方案，将来再慢慢将 Istio 配置作为权威并引入 GitOps 工作流。

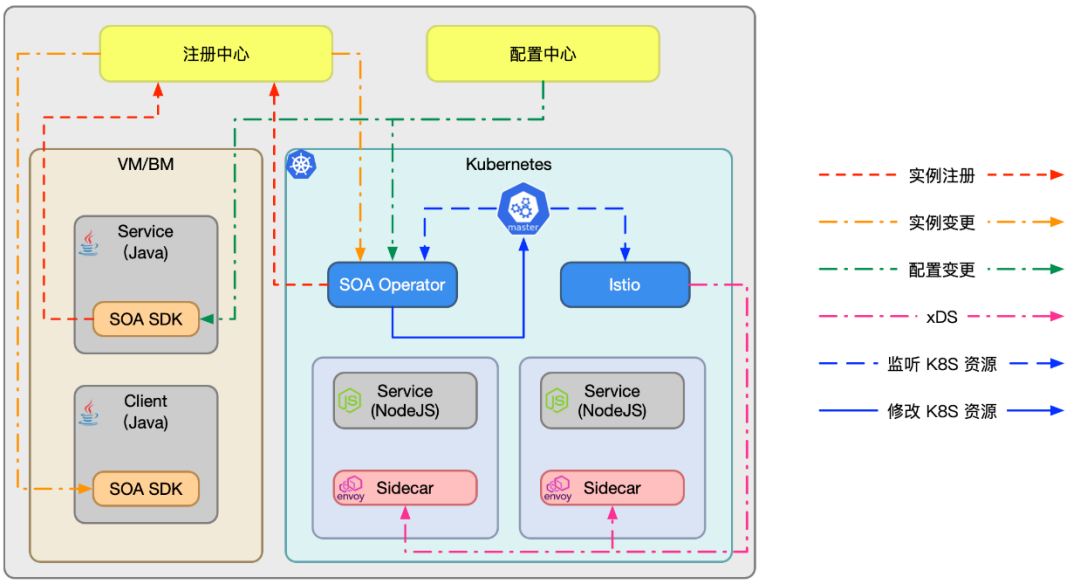
这样选择最主要的原因就是，做现有系统配置和 Istio 配置的映射并不是一个确定的转换规则，随着我们对 Istio 的改造和理解更深入，转换逻辑会得到优化，最终的结果也会改变。另外想要实现方案二，不仅要实现一套老配置到新配置的适配，还要实现一套新配置到老的管理后台的适配，工作量非常大。

3.2 服务注册与发现

想要让服务注册与发现互通，主要方案也会有两个：

- 方案一：按照 Istio 的标准用法，Service Mesh 应用部署在独立的集群中，所有进出集群的流量都走 Gateway；
- 方案二：无论是原有的 SOA 集群还是新的 Service Mesh 集群，网络打通，两套服务注册与发现系统做实例双向同步。

由于历史原因，当前携程内部的应用有部署在 BM、VM、Kubernetes 等环境中，之前并未给不同集群做网络隔离，相互之间都是互通的。因此在实现 Service Mesh 的时候也采用了相同的方案，我们要做的就是一套实例双向同步系统。



对于在 Service Mesh 环境中部署的应用，我们的 Operator 会读取系统中应用和服务之间的绑定关系，通过监听 Kubernetes API 感知到 Pods Ready 后帮助 Pods 注册到老的注册中心，应用的 SDK 无需做任何操作。也就是说在 Service Mesh 环境部署了一个其他语言编写的应用，只要它在系统中绑定了对应的服务并在标准端口暴露了服务，就可以被 Service Mesh 中的应用访问到，也可以被原 SOA 系统中的应用访问到。

对于在原 SOA 系统中部署的应用，它的 SDK 会先将自己上报到注册中心。Operator 会监听注册中心的实例变化并转换成 Istio 的配置。

在 Istio 的模型中，不仅可以将 Kubernetes 中的 Services 和 Pods 转换成 Envoy 的 Clusters 和 Instances，还可以定义 ServiceEntry 和 WorkloadEntry，将外部的地址转换成 Envoy 的 Clusters 和 Instances。我们正是利用了这个功能将原注册中心的服务和实例同步到了 Service Mesh 集群中。

3.3 功能对齐

接入 Service Mesh 之后，原来 SOA SDK 支持的一些功能，我们也需要进行支持，比如预热、熔断、限流等。

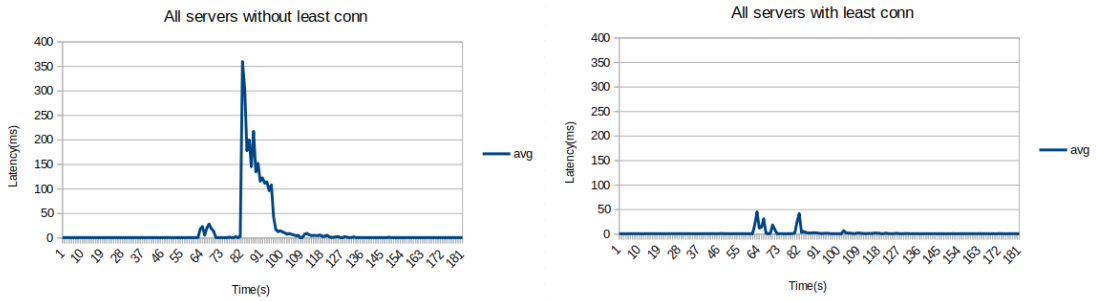
（1）预热

基于 JVM 的应用在刚启动时，由于热点代码还没有进行 JIT 编译等原因，如果这时就接入和平时一样的请求流量，会导致这部分请求的响应时间增加。预热的基本思想就是让刚启动的机器逐步接入流量，目前 SOA SDK 中已经实现了一些预热算法，客户端会根据服务端的配置来控制流量以达到预热的效果，可配置的参数有：预热算法（例如控制流量直线型增长或指数型增长等）、预热时间等。

而 Envoy 在最新版中才提供了类似的功能：Slow start mode。

那如何在现有的版本中实现这个功能？由于这里的核心目标就是让刚启动的服务实例逐步接入流量，最小连接数的负载均衡算法就可以达到这个目标，并且负载均衡方式通过 Istio 的 DestinationRule 直接就可以进行配置，这个算法可以使客户端每次都去访问当前活跃请求数最小那个服务端。于是我们考虑在服务发布期间，将服务的负载均衡方式设置为最小连接数。

对于利用最小连接数的负载均衡的实际效果我们也做了相关的测试，可以看到引入后服务端在启动过程中，客户端的平均响应时间大幅度下降。



在 SOA SDK 支持的预热中需要配置预热时长，而用户很难确定这个时长需要配置为多少。不同的机器配置、不同的 QPS、不同的业务逻辑等，都会影响服务从启动到预热完成所花的时间。如果配置的预热时长太短，实例没有真正的预热完成，就会导致部分请求响应时间增加了；而如果配置的太长，就会导致已经预热完成的实例没有充分的发挥作用。

而在服务接入 Service Mesh 后，在发布期间我们通过最小连接数的负载均衡算法，自适应地调整不同实例的负载。用户不需要关心需要配置什么预热算法，也不需要决定预热时长是多少，只要打开预热开关就可以了。

（2）熔断

Istio 中可以通过设置 DestinationRule 的 ConnectionPool 和 OutlierDetection，来实现熔断策略。当服务由于某些故障开始响应变慢时，ConnectionPool 中关于 pending 请求数、最大并发请求数的设置，会限制客户端继续向变慢的服务发送更多请求，以此来给服务一些时间从响应变慢中恢复。当服务的部分实例出现故障时，OutlierDetection 的配置使客户端停止对这些故障实例的访问，来减轻偶发的部分服务实例故障对客户端的影响，也让故障的那部分实例有时间恢复。

我们的 SOA SDK 目前基于 Hystrix 来实现线程隔离，这部分功能基本对应到 Istio 中的 ConnectionPool 配置，Istio 中不同的目标服务都有自己的一组连接池。两者实现方式不同，但基本可以达到同样的效果。

当服务的部分实例出现故障时，Hystrix 会将新的请求拒绝，这部分可以对应到 Istio 中的 OutlierDetection 配置。不过这里有个区别，这种情况下 Istio 会将故障实例摘除，而不是直接报错。Istio 中这个功能的本质是一种客户端健康检测，但可以达到类似的效果，应该来说

比直接报错更好。

（3）限流

SOA SDK 支持的限流为本地限流，具体包括对特定操作限流、对特定的请求方 AppID 限流等，我们需要在 Service Mesh 中也支持这些功能。由于目前 Istio 没有对限流抽象出模型定义，我们通过 EnvoyFilter 打 patch 的方式，来对开启了限流的服务生成对应的 Envoy 限流配置。

为了将 SOA SDK 已有的丰富的限流配置都在 Service Mesh 中得到对应的支持，我们使用了 Envoy 限流的 descriptors 特性。

参考文档：Local rate limit

比如我们可以通过 descriptor.entry，对来自不同客户端的请求、不同的请求 path 配置不同的限流值。如下的配置，对来自 client-a 的请求，设置的是每 60s 填充 10 个 token，对来自 client-b 且请求 path 是/foo 的请求，设置的是每 60s 填充 100 个 token。

```
descriptors:
  - entries:
      - key: client_appid
        value: client-a
    token_bucket:
      max_tokens: 10
      tokens_per_fill: 10
      fill_interval: 60s
  - entries:
      - key: client_appid
        value: client-b
      - key: path
        value: /foo
    token_bucket:
      max_tokens: 100
      tokens_per_fill: 100
      fill_interval: 60s
```

这里的 client_appid 和 path 的值，通过如下在 route 的 ratelimit 配置中添加 actions 来设置。

```
route:
  cluster: service_protected_by_rate_limit
  rate_limits:
    - actions:
        - request_headers:
```

```
            header_name: x-envoy-client-appid
            descriptor_key: client_appid
        - request_headers:
            header_name: ":path"
            descriptor_key: path
```

以上配置本质上就是通过从请求中提取一些特征，例如读取一些特定的 Header，然后再针对不同的请求分配不同的限流值。

对于更加复杂的场景，例如需要根据多个 Header 做逻辑判断时，我们通过 Envoy Filter 实现相关逻辑并设置到 metadata 中。然后在 ratelimit 的 actions 中从 metadata 中提取特征。例如，我们可以 patch 如下的配置到 envoy 中，生成限流使用的 metadata 数据。

```
- applyTo: HTTP_FILTER
  match:
    context: SIDECAR_INBOUND
    listener:
      filterChain:
        filter:
          name: envoy.filters.network.http_connection_manager
          subFilter:
            name: istio.metadata_exchange
  patch:
    operation: INSERT_FIRST
    value:
      name: service.metadata.ratelimit
      typed_config:
        '@type': type.googleapis.com/envoy.extensions.filters.http.lua.v3.Lua
        inline_code: |-
          function envoy_on_request(request_handle)
              custom_key_1 = build_custom_key_1()
              custom_key_2 = build_custom_key_2()

request_handle:streamInfo():dynamicMetadata():set("metadata.custom.ratelimit",
"request.info", {
                    customKey1 = custom_key_1,
                    customKey2 = custom_key_2
                })
          end
```

然后通过 metadata_key 的方式，拿到我们自己设置的限流相关的 metadata。

```
route:
  cluster: inbound|8080|http|svc-a
```

```
rate_limits:
  - actions:
      - metadata:
          descriptor_key: customKey1
          metadata_key:
            key: metadata.custom.ratelimit
            path:
              - key: request.info
              - key: customKey1
      - metadata:
          descriptor_key: customKey2
          metadata_key:
            key: metadata.custom.ratelimit
            path:
              - key: request.info
              - key: customKey2
```

（4）其他功能

除了一些服务治理常见的功能，携程内部也有不少定制化的功能需要在 Service Mesh 中实现。

这部分需求部分是通过 Lua Filter 实现，部分是扩展 Envoy 编写了 C++ Filter 来实现的。

因为这 C++ Filter 这部分需求比较稳定，所以静态编译在 Sidecar 中也并不是个大问题。

如果可以通过 WebAssembly 实现当然是可以做得更灵活。但在我们项目启动的时候，Istio 的 WebAssembly 还未正式发布，后期我们会考虑引入 WebAssembly。

3.4 SDK 兼容

接入 Service Mesh 后的一大优势就是可以为 SOA SDK 做轻量化，仅保留基本的功能即可。

而 Service Mesh 的接入是一个长期的过程，应用是一批批接入的，同一个应用在不同的机房也有可能存在接入和不接入两种状态。让业务方写两个版本的代码肯定是不合适的。

因此我们在现有的 SOA SDK 中实现了无缝接入功能，原理也非常简单。

凡是接入 Service Mesh 的应用在发布时就会被注入一个环境变量，当 SOA SDK 探测到这个环境变量后，便会启动轻量化模式。

其中轻量化模式中被移出的功能包括：

- 禁用服务注册与发现

- 禁用路由功能，每个服务都有一个固定的域名，所有请求直接按服务请求固定的域名即可
- 禁用各种服务治理功能，例如：熔断、限流、黑白名单、重试、负载均衡、路由等

这样不仅有利于业务方快速回滚，也可以方便业务方对两种 SOA 架构进行性能对比。

四、数据平面

4.1 HTTP 协议

携程当前主流的 SOA 传输协议还是 HTTP，这块的确慢了半拍，但这也更利于我们接入 Service Mesh。

Istio 本身对 HTTP 协议有着很好的支持，因此这部分并不需要我们做什么调整。

4.2 Dubbo 协议

携程在 2018 年的时候引入了 dubbo 协议作为 HTTP 协议的补充，在两年时间的发展中也积累了较多的用户群体。

我们的 dubbo 的调用方式依赖 Dubbo 框架中的 dubbo 协议。

部分应用扩展使用了压缩率更高的 protobuf 来做序列化并进一步使用 gzip 压缩提升性能。

dubbo 协议本身是四层的私有协议，在 Istio 中的支持力度远不如 HTTP。另外 Dubbo 3.0 中也将 gRPC 协议作为了新的传输协议。

如何让 dubbo 协议升级到 gRPC 成为 Service Mesh 落地必须解决的问题。

（1）现状

当前携程内部通过 Dubbo 框架调开发服务端应用有近千个，对应的调用方就更多了。

并且考虑到携程内部使用 Node.js、Python 等语言的应用越来越多，新版升级必须满足如下的要求：

- 使用 gRPC 协议以支持 Service Mesh
- 使用 Dubbo 框架的业务方尽量不改或者少改代码
- 新协议注册的服务端符合 gRPC 的规范并使得其他语言客户端可以很方便地调用
- 不强制依赖 protobuf，能让用户保持 Code First 的编码习惯

（2）技术选型

我们调研了当时主流的做法并通尝试得出三条可行的升级道路。

这里最大的难点是当前使用 dubbo 的旧服务不是基于 protobuf 编写契约的，所以不能直接通过依赖 protobuf 结构的 gRPC 发起调用。

【方案一】

依然用原来的序列化器将数据处理成二进制，在 gRPC 调用时 wrap 一个 protobuf 对象，用一个字段传递原来数据的二进制数据流，再用另外一些字段描述它的序列化方式。这也是 Dubbo 3.0 中透明升级 gRPC 协议时所使用的方案。

- 优点：

该方案不需要对业务代码改动，并且支持 Code First

- 缺点：

这种方式对于标准的 gRPC 客户端不友好；
两次序列化和反序列化影响性能。

【方案二】

gRPC 标准中，没有规定 gRPC 强依赖 protobuf 做序列化器，gRPC 官方的 FAQ 中这样写道：

gRPC is designed to be extensible to support multiple content types. The initial release contains support for Protobuf and with external support for other content types such as FlatBuffers and Thrift, at varying levels of maturity.

那具体如何使用其他序列化器呢？

从协议角度，只需要改变 content-type 即可。例如想表达用 json 作为序列化格式，那具体内容为：application/grpc+json。

在 Golang 中只需要额外加一行配置一下即可：

```
grpc.WithDefaultCallOptions(grpc.CallContentSubtype(codec.JSON{}.Name()))
```

- 优点：

对业务开发友好，符合 gRPC 协议的标准
不会影响性能

- 缺点：

Java gRPC 客户端并没有实现这个功能

【方案三】

通过代码规范来约束字段的前后顺序，然后使用类似 protostuff 这样的框架把其他契约转换成 protobuf。

- 优点：

符合 gRPC 标准，对标准 gRPC 客户端友好
不会影响性能

- 缺点：

对于业务开发不友好，不能删字段，不能调整字段顺序，非常容易出错

【最终方案】

经过对比后，我们选择方案二，基于如下理由：

- 查看 gRPC 插件生成的 Java 类，其中存在可以定制序列化器的部分的代码，使用 json 改写序列化器是可能的。
- 对于 Golang 来说，天然支持这种使用方式。
- 对于 Node.js 和 Python 等动态语言，替换序列化器非常简单。
- Dubbo 支持 POJO 并且基于 Java POJO 的服务定义方式在携程大多数应用的开发形式。

（3）技术实现细节

【去契约化】

原生的 gRPC 基于 proto 文件，依赖 gRPC 的插件完成代码生成。

Dubbo 中的 gRPC 依赖 proto 文件生成 gRPC 代码的同时，需要还要配合 Dubbo 扩展的 proto 插件生成 DubboGprcWrapper。

Dubbo 本身为 gRPC 做了包装，可以让 gRPC 协议的入口复用 dubbo 本身的注册发现，服务注册与发现以及负载均衡等扩展。

proto 文件生成 gRPC 代码本质上有两部分，一部分是对服务的定义，另一部分是对 DTO 的定义和 DTO 的序列化反序列化代码。

如果只有代码而没有 proto，我们可以依靠反射服务接口类来实现获取服务和 DTO 的定义。

而 DTO 的序列化反序列化，如果替换成 json 或 hessian，那就不依赖 gRPC 生成的代码了，只要有 POJO 就可以处理。

XXXMethodDefinition.setResponseMarshaller(io.grpc.protobuf.ProtoUtils.marshaller(InputTy

pe.getDefaultInstance()))

上面的代码是 gRPC 插件生成的，只需要将这部分替换成其他序列化器，就可以实现序列化器的替换。

【Java 字节码编译技术动态编译】

有了上述去契约化的实现，后续我们基于 Java 字节码编译技术，直接在内存中生成对应的 gRPC 协议依赖的对象。

使得用户可以在代码优先的情况下，依旧可以继续开启 gRPC 协议传输。

同时因为 gRPC 相关类是动态生成的，我们可以在动态编译的时候为接口定制序列化器。

只要将序列化格式换成其他等不依赖 protobuf 结构的类，旧的服务也可以直接升级为 gRPC。

而 Dubbo 依赖的 DubboGprcWrapper 的类也改由动态编译生成，至此用户将脱离契约的限制。

这里我们使用了基于 JDKCompiler 实现编译器，使用 JDK8 的版本编译生成代码。

我们在代码中用 mustache 模板定义了代码生成的主要框架，然后根据 Java 服务类反射获取元数据渲染 Java 类。

部分模板代码如下：

```
{#methods}}
    {{^isMultiParameter}}
    {{#isManyInput}}
        {{#isManyOutput}}
        {{/isManyOutput}}
        {{^isManyOutput}}
    public io.grpc.stub.StreamObserver<{{inputType}}> {{methodName}}(
        io.grpc.stub.StreamObserver<{{outputType}}> responseObserver) {
        return asyncUnimplementedStreamingCall(getGetStreamMethod(),
responseObserver);
     }
        {{/isManyOutput}}
    {{/isManyInput}}
     {{^isManyInput}}
        {{#isManyOutput}}
        {{/isManyOutput}}
        {{^isManyOutput}}
```

```
            {{/isManyOutput}}
        {{/isManyInput}}
{{/methods}}
```

【用户无缝升级】

解决代码热生成之后，支持旧服务无缝升级还需要解决一些问题。

首先业务原生的代码实现逻辑与 gRPC 接口存在差异。

比如 Dubbo 的 gRPC 要求实现类需要实现 XXXBase 接口，而实际 Code First 的用户实现类是实现了对应的 XXXInterface 服务接口。我们也无法让用户在编码阶段使用我们动态生成类。

我们在动态生成时代码内部额外生成一个 Wrapped 类，该类代理业务的实际逻辑并且实现 gRPC 服务需要实现的 XXXBase 基类。

并且在业务实际代码和 gRPC 代码之间实现各种兼容的逻辑。而 gRPC 接口在默认方法中调用子类的对应方法。

```
@javax.annotation.Generated(
value = "generate by CDubbo",
comments = "Source: {{protoName}}")
public class {{className}}Wrapper extends
Dubbo{{serviceName}}Grpc.{{serviceName}}ImplBase {
    private static final TransformUtils transformUtils =
CDubboInjector.getInstance(TransformUtils.class);
    private static final CDubboUtil cDubboUtil =
CDubboInjector.getInstance(CDubboUtil.class);
    //实际的业务实现类
    private {{serviceName}} instance;

    public {{className}}Wrapper({{serviceName}} instance) {
      this.instance = instance;
    }

{{#unaryMethods}}
  // 被代理的对象
  @Override
  public {{outputType}} {{methodName}}({{inputType}} request) {
    try {
        return instance.{{methodName}}(request);
    }catch (Throwable e) {
        throw cDubboUtil.toGrpcException(e);
```

```
    }
  }
{{/unaryMethods}}
```

服务接口类：

```
public interface I{{serviceName}} extends {{serviceName}}{
   default public void {{methodName}}({{inputType}} request,
io.grpc.stub.StreamObserver<{{outputType}}> responseObserver){
            responseObserver.onNext(this.{{methodName}}(request));
            responseObserver.onCompleted();
    }
}
```

我们在 Spring 注入的时候更新 BeanDefinition 来实现替换，将包装类设置为服务实现类的 Wrapper。

客户端这边也做类似的兼容逻辑。

上述过程完成后，用户只要在 ProtocolConfig 配置中明确申明使用 gRPC，就会自动为服务和客户端开启 gRPC 协议。

【新老方法兼容之 CompletableFuture】

改动生成服务上下文的时候标注该方法是 CompletableFuture 方法，在服务端的默认实现的位置获取到 Future 之后更改对应的方法转换为 ResponseObserver。直接更新 Dubbo 部分代码生成的结果。

对应的更改的模板文件：

```
default public void {{methodName}}({{inputType}} request,
io.grpc.stub.StreamObserver<{{outputType}}> responseObserver){
                CompletableFuture<{{outputType}}> response =
this.{{methodName}}(request);
              response.whenComplete((res,thr)->{
                if(thr == null){
                   responseObserver.onNext(res);
                   responseObserver.onCompleted();
                }else {
                   responseObserver.onError(thr);
                }
              });
        }
```

客户端使用返回 ListenableFuture 的方法返回 Future 之后再使用 Guava 的工具转换成 CompletableFuture：

```
{{#futureRequestMethods}}
    public CompletableFuture<{{outputType}}> {{methodName}}({{inputType}} request) {
        com.google.common.util.concurrent.ListenableFuture<{{outputType}}>
responseFuture = futureStub
                .withDeadlineAfter(url.getParameter(TIMEOUT_KEY,
DEFAULT_TIMEOUT), TimeUnit.MILLISECONDS)
                .{{methodName}}(request);
        return FutureConverter.toCompletableFuture(responseFuture);
    }
{{/futureRequestMethods}}
```

【Stream 方法原地升级】

Dubbo 中并没有类似 gRPC Stream 的支持，所以我们基于 Dubbo Callback 进行了扩展。

CDubbo 中扩展后的接口所示：

```
public interface StreamContext<T> {
    Result write(T v);
    void close();
}
```

将 dubbo 协议升级到 gRPC 协议后如何对接到 gRPC Stream 呢？

我们在服务 Wrapper 里面 针对 gRPC 的 ResponseObserver 转换成对应的 StreamContext 传递给服务的实现类：

```
StreamContext<{{outputType}}> streamContext = new StreamContext<{{outputType}}>() {
        @Override
        public Result write({{outputType}} v) {
          try{
            responseObserver.onNext(v);
            return Result.SUCCESSFUL;
          }catch ( Exception e){
            throw new RuntimeException(e);
          }
        }

        @Override
        public void close() {
          responseObserver.onCompleted();
```

```
            }
        };

        try{
            // 现在都是一个请求然后拿 stream 的方式 暂时也只支持这样
            instance.callStreamMethod(request,streamContext);
        }catch (Throwable e){
            responseObserver.onError(e);
        }
    }
```

客户端需要在调用 Stream 的时候将 gRPC 的响应返回回调。Stream 有统一处理响应消息的位置，只要拿到请求的 StreamId 就能把 gRPC 的响应回调给具体的某次响应。

```
StreamContextImpl responseObserver1 = (StreamContextImpl) responseObserver;
    // stream 请求有个专门处理响应的地方 streamId 表示是第几个 Stream 请求的响应
    String streamId = responseObserver1.getStreamId();

    io.grpc.stub.StreamObserver<{{outputType}}> streamObserver = new
io.grpc.stub.StreamObserver<{{outputType}}>() {
        @Override
        public void onNext({{outputType}} value) {
            // streamManager 是统一处理所有服务端响应的位置
            streamManager.dispatch(streamId,value);
        }

        @Override
        public void onError(Throwable t) {
            // 服务端报错就直接抛异常
            streamManager.killWithException(streamId);
            throw new RuntimeException(t);
        }

        @Override
        public void onCompleted() {
            // 服务端结束 则客户端把对应的 entry 结束掉
            streamManager.kill(streamId);
        }
    };
    stub.withDeadlineAfter(url.getParameter(TIMEOUT_KEY, DEFAULT_TIMEOUT),
TimeUnit.MILLISECONDS)
                .getStreamResponse(request, streamObserver);
```

【服务端自动选择序列化器】

Java 中既然可以替换默认的序列化器，那么也可以实现根据 Content Type 自动选择序列化器。

我们在代码生成的时候为 gRPC 注入一个用于桥接的序列化器 BridgeMarshaller，而不是一个特定的序列化器。

因为序列化器中没有办法拿到 Header，所以需要找一个扩展点从 Header 里面获取 Content Type，并通过上下文传递到序列化线程。

我们发现了 StreamTracer 这个扩展，虽然它的原始目的是为了处理 Trace Context，但和我们要做的事情是一样的，都是提取 Header 里的数据放到上下文中。

Trace Context：实现分布式追踪时需要透传相关上下文，在 HTTP 调用中一般将它放在 Header 中。

StreamTracer 扩展：

```
public class CDubboGrpcConfigurator implements GrpcConfigurator {
  private static final CDubboUtil cDubboUtils =
CDubboInjector.getInstance(CDubboUtil.class);
  // 获取 header 传递上下文
  @Override
  public NettyServerBuilder configureServerBuilder(NettyServerBuilder builder, URL url) {
    builder.addStreamTracerFactory(new ServerStreamTracer.Factory() {
      @Override
      public ServerStreamTracer newServerStreamTracer(String fullMethodName,
Metadata headers) {
        String contentType = cDubboUtils.extractContextType(headers);
        return new ServerStreamTracer() {
          @Override
          public Context filterContext(Context context) {
            return context.withValue(GrpcConstants.Serialization.SERIALIZATION_KEY,
contentType);
          }
        };
      }
    });
    return builder;
  }
}
```

序列化选择：

```
public class GenericMarshaller<T> implements MethodDescriptor.Marshaller<T> {
```

```
    //.....
@Override
    public T parse(InputStream stream) {
        String serialization =
cDubboUtil.getSerialization(GrpcConstants.Serialization.SERIALIZATION_KEY.get());
        GrpcSerialization grpcSerialization = SERIALIZATION_MAP.get(serialization);
        if(grpcSerialization == null){
            throw new IllegalArgumentException("can not find any serialization [" + serialization
+ "] by content-type in header.");
        }
        return grpcSerialization.getMarshaller(clazz).parse(stream);
    }
}
```

CDubbo 升级 gRPC 后默认使用 hessian2 作为序列化器与 Dubbo 原生保持一致，尽量避免引入兼容性问题。

而其他语言例如 Golang 等就可以选择使用 json，而且并不需要 protobuf 定义契约，直接调用 gRPC 底层方法即可。

```
func main() {
    conn, err := grpc.Dial("127.0.0.1:9080",
        grpc.WithDefaultCallOptions(grpc.CallContentSubtype(codec.JSON{}.Name())), //
codec.JSON 只要实现 gRPC 的`encoding.Codec`接口即可
        grpc.WithTransportCredentials(insecure.NewCredentials()),
    )
    if err != nil {
        log.Fatalf("%v\n", err)
    }

    resp := new(CheckHealthResponse) // Response 可以用 Code First 自己构建
    err = conn.Invoke(context.Background(),
        "/com.test.TestService/CheckHealth", // 这部分是 Java 中的服务接口完整名称 +
方法名
        CheckHealthRequest{Message: "Hello World"}, // Request 可以用 Code First 自己
构建
        resp, grpc.EmptyCallOption{})
    if err != nil {
        log.Fatalf("%v\n", err)
    }
    fmt.Printf("%+v\n", resp)
}
```

另外如果这个 Dubbo 服务本身就是用 protobuf 做契约的，那么 prodobuf 也是可以使用

的。

【gRPC 中异常传递】

而在 gRPC 的原生 API 中响应是缺少这部分数据的。服务处理的异常不 catch 直接会在客户端转化为 StatusRuntimeException: UNKNOWN

```
@Override
public void getLatest(User request, StreamObserver<CallHistory> responseObserver) {
    String name = request.getName();
    try {
        Integer.parseInt(name);
    }catch (Throwable e){
         // grpc 这边默认是可以带有异常 但是显示抛出
        responseObserver.onError(INVALID_ARGUMENT.withCause(e).withDescription("服务内部错误").asRuntimeException());
        return;
    }
}
```

我们上述代码中已经对服务实现类做了一层 wrapper，实现了 gRPC API 之间的兼容。

由于我们是基于 Dubbo 开发的，Dubbo 遇到未知异常会转换成 RpcException。如果我们不处理，这个异常在 gRPC 中就是 UNKNOWN。因此需要在底层把 RpcException 和 GrpcException 的转换。将一般错误变为 gRPC 的异常抛出。

五、配置按需下发

在集群规模较小时，Service Mesh 中的每个服务默认可以访问任何其他的服务是没有问题的。但是当集群规模变大之后，就会出现以下问题：

- 推送频率高：任何一个服务的变动都要通知到所有的 Sidecar
- 推送数据量大：Sidecar 中包含所有服务的 Cluster、Route 等信息。

在实际的服务调用关系中，一个服务并不会真的要访问所有其他的服务，相对于服务的总量而言，每个服务只会访问到很小一部分的服务。

我们需要找到一种方式来管理服务间的调用依赖关系，并且能让 Istio 根据这份调用关系，减少推送的频率和数据量。也就是说，当 A 服务发生变化时，只有依赖 A 服务的其他服务会收到推送。

5.1 解决方案

在 Istio 中，可以通过 Sidecar 资源对 Sidecar 进行配置。例如如下的 Sidecar 配置，svc-a 需要访问 svc-b 和 svc-c，当 svc-b 和 svc-c 的配置变化时，svc-a 的代理会收到推送，

而其他 svc 的配置变更，并不需要推送给 svc-a 的代理：

```
apiVersion: networking.istio.io/v1beta1
kind: Sidecar
metadata:
  name: svc-a
  namespace: prod
spec:
  egress:
    - hosts:
        - "prod/svc-b.soa.mesh"
        - "prod/svc-c.soa.mesh"
        - "istio-system/*"
  workloadSelector:
    labels:
      app: svc-a
```

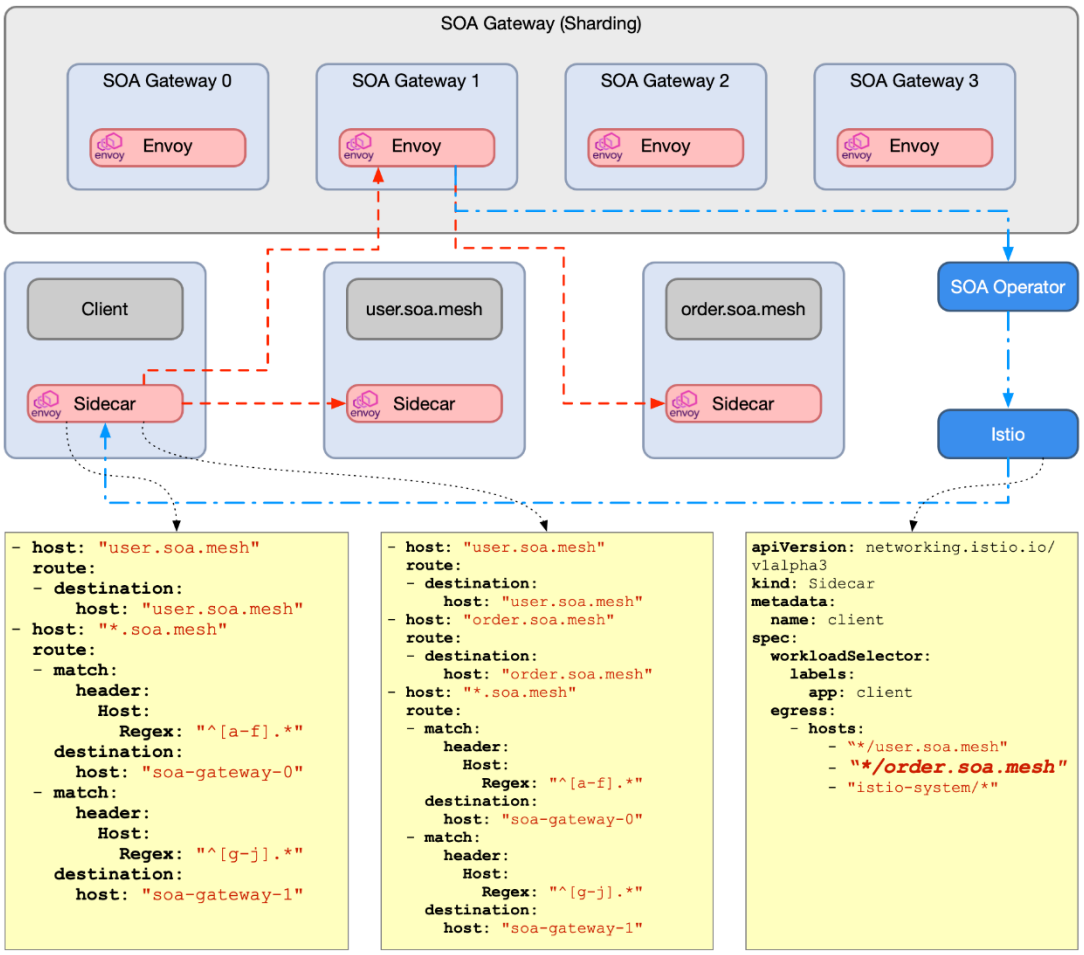
这样可以解决推送量大和推送频率高的问题。

但是随之而来的新问题是，怎么知道 svc-a 需要访问 svc-b 和 svc-c 呢？一种直接的方式是，接入 Service Mesh 之前分析好服务调用依赖，配置到 Sidecar。

携程目前并没有为所有服务之间梳理过调用关系，这种方式会给接入 Service Mesh 的用户带来负担，我们需要一种更加自动化的方式来透明地解决服务依赖关系的问题。

我们内部将所有的服务都映射到了.soa.mesh 这个域名上。例如一个用户服务的域名是：user.soa.mesh。

我们采取的方案是为每个 Sidecar 加一条 host 为*.soa.mesh 的路由，其中的路由目标设置为一个默认的 Gateway。当访问一个不在 Sidecar 资源的 egress.hosts 里列出的服务时，都会匹配到这条兜底路由，转发到 Gateway 进行处理。Gateway 收到这条转发过来的请求后，也就知道了服务间的调用依赖关系。转发这条请求的时候通过一定的机制去更新 Sidecar 资源。



图中的兜底路由通过 Host 做了分片，因为单个 Gateway 也无法承载所有的服务。

最初我们考虑创建一个 hosts 为*.soa.mesh 的 VirtualService 来让 Istio 生成对应的路由，但是实际测试发现这种方式无法达到预期的效果，Istio 在处理包含*前缀域名的时候有点问题。另外如果想把这条路由下发到 Sidecar，那我们也必须在 Sidecar 资源里加上*.soa.mesh。但如果这么一加，岂不是就把所有服务下发下去了？

最终我们通过 EnvoyFilter 的方式，为每个 Sidecar 来 patch 这条兜底路由。因为 Istio 中的 EnvoyFilter 作用在更底层，并不受 Sidecar 资源的控制。

```
apiVersion: networking.istio.io/v1alpha3
kind: EnvoyFilter
metadata:
  name: soa-default-route
spec:
  configPatches:
    - applyTo: VIRTUAL_HOST
      match:
        context: SIDECAR_OUTBOUND
```

```
    routeConfiguration:
      name: "80"
  patch:
    operation: ADD
    value:
      name: mesh-soa-gateway
      domains:
        - "*.soa.mesh"
        - "*.soa.mesh:80"
      routes:
        - match:
            prefix: /
          route:
            cluster: outbound|80||mesh-soa-gateway
            timeout: 0s
            max_grpc_timeout: 0s
```

六、未来展望

6.1 WebAssembly

在 Service Mesh 中引入 WebAssembly 应该来说是一个大趋势，这也是 Service Mesh 的一大优势，不把这个优势利用好就太浪费了。

一方面，Istio 的内置功能不可能覆盖所有公司的业务需求。另一方面，各个事业部或部门也常常会有一些团队公共需求。

如果能将这些功能通过 WebAssembly 实现热加载、热更新，那 Service Mesh 的价值就被充分地发挥了。

抱着这个美好的愿景，我们在前期就调研了 Istio 中 WebAssembly 模块的功能、性能和易用性。

只可惜在一年多前这块还是有不少问题的，包括 Sidecar 占用内存大增、内存泄露、WebAssembly SDK 扩展性不够等。

目前我们内部的 Istio 版本经过了不少定制化，所以还停留在 1.10 版本，并不会紧跟官方最新版。

但后续随着官方对 WebAssembly 功能的完善，将静态编译的 Envoy C++ Filter 迁移到 WebAssembly 上必定是一个趋势。

6.2 Sidecar 模式

Service Mesh 背后的模式一般被称为 Sidecar 模式。

这种模式是否可以扩展到其他领域？Service Mesh 只是解决了 SOA 这个领域的问题，能否把这种模式扩展到其它领域？充分发挥 Sidecar 模式的优势？

这是很多人会问的问题。

如果把所有模块都做成了 Sidecar 模式，我觉得要想清楚 2 个问题：

- 相关模块能接受额外 1ms 的响应延迟吗？Sidecar 带来的优势能否弥补这个问题？
- Sidecar 模式和 Proxy 模式的本质区别是什么？

引入 Sidecar 势必会引入额外的性能损耗，不经特殊优化的情况下 Istio 最好响应延迟也要增加 1ms 以上，不能忽略了这部分的影响。

例如 Redis 中，额外引入 1ms 的响应延迟一般是无法接受的。

在关系型数据库中，通过 Proxy 来实现读写分离和分库分表是一种很常见的做法，Sidecar 模式本质上也是一种 Proxy。

那它和传统 Proxy 模式有什么区别呢？

它们最大的区别就在于这个代理部署在哪里。如果在客户端一侧，那就是 Sidecar 模式；如果在服务端一侧，那就是 Proxy 模式。

对于这两种模式如何取舍呢？

如果服务端并不是分布式的，或者目标服务器在同一个机房离得很近，那在服务器同一个机房部署一套 Proxy 集群维护起来会更加方便。

而 SOA 最大的特点是一个应用往往既是客户端也是服务端，调用关系是网状的。

对于 SOA 来说，在 Sidecar 和 Proxy 之间也就只有 Sidecar 可选了。

所以，想在其他模领域引入 Sidecar 模式前要把这两个问题想清楚了，而不是无脑地引入 Sidecar 模式。

七、总结

携程的 Service Mesh 从启动到现在经历了将近 2 年的时间，正式上线也有 1 年多了。当前已经接入了大约 2000 个应用、4000 个服务和 6000 个 Pods，并且一直保持着稳定运行，也经受住了各种故障演练的考验。

接入 Service Mesh 后，很多 Bug 仅需修改一下控制面代码或配置就可以轻松地实现热更

新；原来的 SOA SDK 也进入了冻结维护状态不再开发新功能；公司内部大量非 Java 应用也可以轻松地接入 SOA 体系。这让 SOA 团队可以为公司产出更大的价值。

当然，还有许多挑战摆在我们的面前：WebAssembly 还未发挥应有的作用；dubbo 升级 gRPC 也还未在生产部署；控制面的性能指标依然不能达到我们的要求等。这些是我们接下来需要去解决的。

最后，希望本文中提到的各种问题和解决方案能给大家带来帮助和启发。

支持 10X 增长，携程机票订单库 Sharding 实践

【作者简介】初八，携程资深研发经理，专注于订单后台系统架构优化工作；JefferyXin，携程高级后端开发专家，专注系统性能、业务架构等领域。

一、背景

随着机票订单业务的不断增长，当前订单处理系统的架构已经不能满足日益增长的业务需求，系统性能捉襟见肘，主要体现在以下方面：

- 数据库 CPU 资源在业务高峰期经常达到 50%以上，运行状况亮起了黄灯；
- 磁盘存储空间严重不足，需要经常清理磁盘数据腾挪可用空间；
- 系统扩容能力不足，如果需要提升处理能力只能更换配置更好的硬件资源。

因此我们迫切需要调整和优化机票订单数据库的架构，从而提升订单系统的处理性能。通过建立良好的水平扩展能力，来满足日益增长的业务需求，为后续系统优化和支撑 10x 订单量的增长打下良好基础。

1.1 存储架构的演进

我们选择一个新的系统架构，应该基于当下面临的问题，综合成本、风险、收益等多方面因素，选择出的最合适的方案。机票订单库的架构演进也不例外。

我们最开始接触机票订单数据库时，它是一个非常庞大的数据集合，所有的订单业务全部都集中一个数据库上，因此整体 BR 非常高。同时，我们的 SQL 语句也非常复杂，混杂着很多历史遗留下来的存储过程。可想而之，整个数据库当时的压力巨大，维护成本居高不下。DBA 每天的工作也非常忙碌，想方设法降高频，解决慢 SQL 等线上问题。生产偶尔也会因为某些没有 review 的 SQL 导致数据库短暂的停止服务。

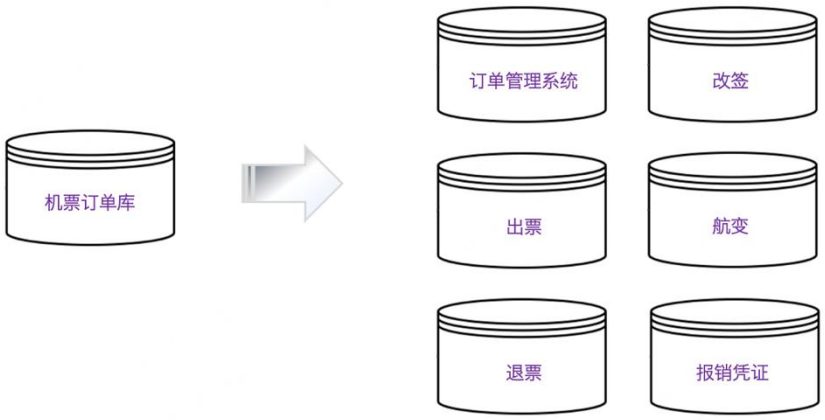
初期，我们采用了最常见的几种手段进行优化，包括：

- 索引优化
- 读写分离
- 降高频

虽然手段比较常规，通过一段时间的治理，订单库的稳定性也得到了一定的增强。总体实施成本较低，效果也是立竿见影的。

随着时间的推移和数据的积累，新的性能瓶颈逐渐显露。我们再次对系统进行了升级，对数据库架构做了改进。主要包括以下几个方面：

- 垂直拆分

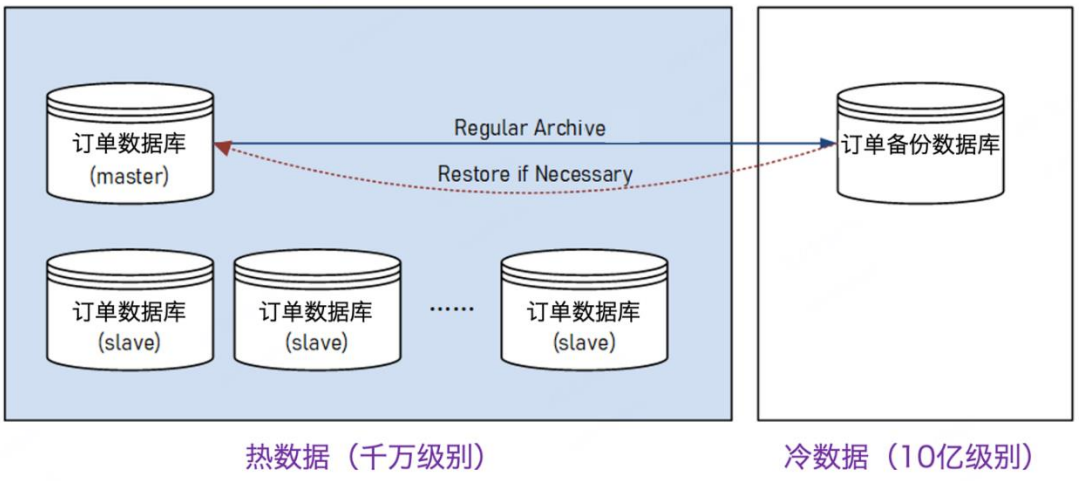


订单库根据业务属性拆分成了多个数据库

基于业务对数据库进行垂直拆分在很大程度上提高了系统的可靠性和可维护性。一个上百人的团队，同时对一套数据库进行维护，对于发布变更来说是一种煎熬，同时也存在很大的风险。当一个非核心链路上的发布出现了问题，例如某些操作导致了锁表或者占用过多的系统资源，其他关键链路的数据库访问都会因此受到影响。

我们根据不同的业务场景，例如：订单管理系统、出票、退票、改签等业务，将数据库进行垂直拆分。使各自业务系统数据隔离，减少相互的影响。这些拆分的数据库，可以根据不同性能要求，灵活调整数据库的部署方式，来降低总体成本。

- 水平拆分（冷热数据分离）



通常来说，当航班过了起飞时间并且用户已经使用了当前机票，那么我们认为该订单服务已经完成，后续订单数据发生改变的可能性很小，于是会将该数据迁移到一个具有相同结构的冷数据库中。该数据库仅提供查询功能，不提供修改功能。但是我们发现少数场景仍然需要对这些数据进行修改。于是我们开发了一套数据还原功能，将处于冷数据库中的数据，还原到热数据库中，然后再进行操作。

注: 我们当时采用的数据库和数据结构是完全一致的, 这样做备份和还原、查询会比较方便。其实也可以采用其他类型的数据库，例如 Mongo 等。在读取性能和使用成本等方面可能会更具优势。

这次升级同样解决了不少问题，使数据库的稳定性得到了很大的增强。

1.2 基于冷热数据分离的适用性

虽然基于冷热数据的分库方案，在目前来看遇到了瓶颈。但是我认为它是一个非常值得借鉴的方案。我们现在仍然有大量的业务系统数据库采用这种方案对数据进行拆分。它不仅实施简单，同时运维成本也相对较低。

优势:

- 功能简单
- 实施成本低

局限性:

- 数据冷处理的规则应该相对简单，不应该经常发生变化
- 热数据的膨胀需要受到限制，否则热数据的量一旦累积过多，性能瓶颈仍然会出现
- 需要额外的查询来找到订单所处的位置（冷/热数据库）
- 因为冷数据量庞大，冷数据的查询能力、表结构调整能力都收到了限制，不能进行复杂的业务查询操作

根据我们的系统规划，在当下或者可预见的未来满足以上提到原因中的多个，那么就得谨慎选择采用此方案。或者在改方案的基础上进行优化。

正是由于我们目前的业务场景恰好命中了上面列举的所有问题, 我们才需要对这个架构进行进一步调整，选择一个更好的水平扩展的方式，解决当前系统面临的问题。

1.3 当时面临的主要问题

从 2019 年开始，我们就开始着手研究和规划订单数据库 sharding 项目。当时主要面临如下问题:

1.3.1 订单的存储要求

受制于当前订单数据库架构的限制以及机票业务的特殊性（通常不超过 2 年的处理生命周期), 改造前的订单数据库仅能够支持 2 年的订单存储。超过 2 年, 我们会将数据进行归档。用户和员工都无法通过在线查询的方式获取订单信息。

但是基于以下几个方面原因，原本 2 年的存储和处理周期已经不能满足客户和业务的需要:

- 从客户的角度出发，仍然有查询历史订单的需求；
- 业务场景的拓展导致机票订单整个服务周期变长。

原先机票使用完成（出行）一段时间后就可以视为服务结束，大部分订单 3 个月后就不会发生变化，但是由于新业务的推出，热点数据查询和处理周期明显变长。

1.3.2 系统架构瓶颈

- 热数据膨胀

热数据原本仅千万级别，由于业务的变化热数据数量不断膨胀。

- 冷数据量庞大

由于订单存储周期拉长和订单量的增长，冷数据的数量也不断攀升。冷数据库查询性能不断下降；索引调整也变得非常困难，经常出现修改失败的场景。

- 数据库高峰期 BR 达到了 10w+；
- 系统存储了 20TB 的数据，磁盘使用率达到 80%以上，经常触发使用容量告警；
- 主库的 CPU 使用率高峰期接近 50%；
- 由于采用了读写分离的架构，当主库的服务器的性能受到影响的时候，AG 延迟变得非常高，偶尔达到分钟级，有的时候甚至更长。

主从同步的延迟，导致了数据新鲜度的降低。我们之前的 ORM 层封装了一个新鲜容忍度的参数。当不能满足新鲜度要求的时候，读取会切换到主库，从而进一步加重了主库的负担。

因此，订单库的整体性能压力非常大，如果想快速解决性能问题，只能对机器进行扩容。但是由于数据库本身就是消耗资源大户，CPU 和内存消耗非常高，只能通过进一步提高数据库的硬件配置来解决问题，因此整体升级的成本居高不下。另外硬件升级完成后，SQLServer 的授权成本可能也会进一步提升。

为了彻底解决以上问题，我们计划通过优化架构来提升系统的水平扩展能力，从而进一步提升我们系统的性能和服务水平。

二、项目目标和实施方案

2.1 目标

基于上文提到的这些问题，为了确保系统能够长久持续的稳定运行并且提升订单系统的处理能力，我们计划对数据库的架构进行升级，总体实现以下目标：

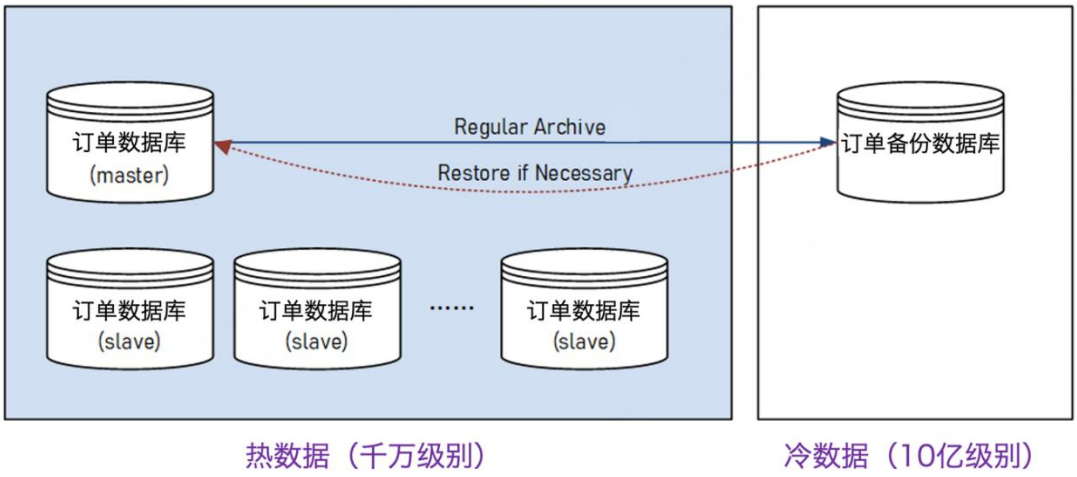
- 订单的存储和处理周期至少达到 5 年
- 提升订单系统的处理能力，支撑订单 QPS10 倍的规模增长
- 在提升系统性能的前提下，降低总体成本
- 提高系统的水平扩展能力，通过简便的操作可以快速扩容以应对长期的业务增长

我们希望通过 1-2 年的时间，实现对数据库架构升级改造以及完成 SQLServer 迁移到 MySQL 的目标。

2.2 架构改造

2.2.1 新旧架构的对比

（1）旧系统架构



架构说明：

- 订单数据库

为热数据的主库，提供读写功能

- 订单数据库 Slave

为热数据的从库，在保障新鲜度的前提下提供只读功能，采用 SQLServer 的 AlwaysOn 技术

- 订单备份数据库

为冷数据库（没有主从之分），仅提供只读功能

- Archive

备份机制，根据业务的需要，为了缓解主数据库的压力，对于符合条件的订单（通常时起飞后+已出行）定时迁移到冷数据库的操作

- Restore

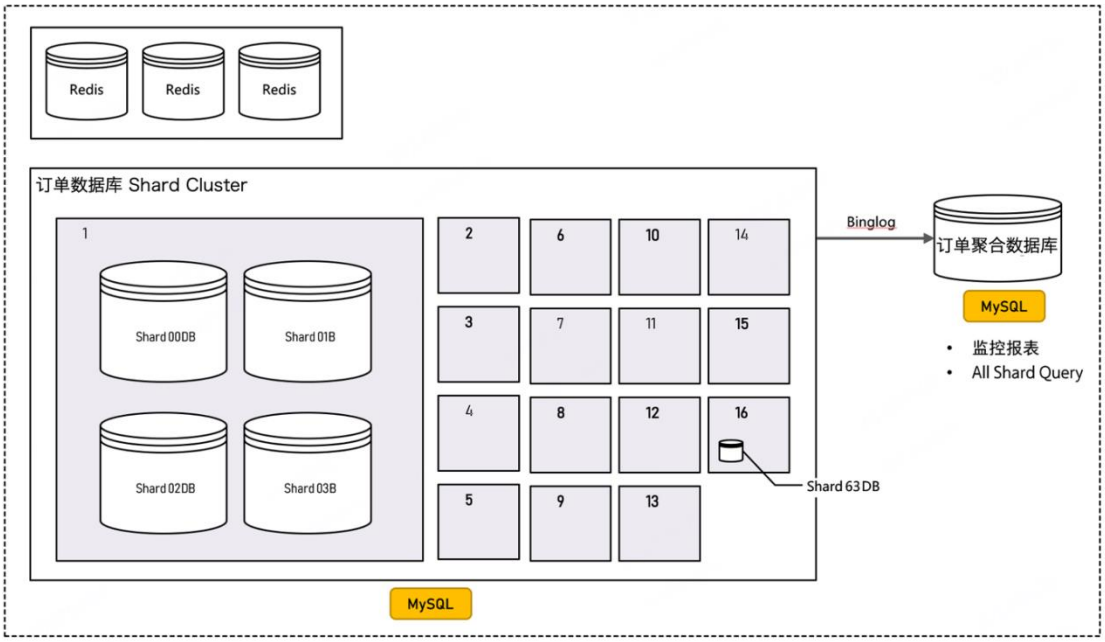
还原机制。在一些特殊情况下，需要对已经备份的数据进行修改，我们需要将数据从冷数据库中恢复到热数据库然后才能进行操作。这个操作叫做还原。

当前存在的问题：

- 数据量变多，业务场景变得复杂后，主库的数据量从千万级增长到亿级别，对数据库的性能产生明显影响

- 冷数据库的历史累计数据量也在不断膨胀，受到本地 SSD 磁盘容量的限制，磁盘空间使用率达到了 80%以上
- 冷数据库的订单数量达到了十亿级，数据库索引调整，结构调整变得较为困难；查询性能也受到了很大的影响
- 备份和还原逻辑需要根据业务要求不断调整

（2）新系统架构：



架构说明：

- 订单数据库 Shard Cluster

新数据库基于订单号将数据水平拆分成 64 个分片，目前部署在 16 台物理机上

- 订单聚合数据库

针对热点数据，通过 Binglog 和有序消息队列同步到订单聚合数据库，方便数据监控，并且用于提高数据聚合查询的性能

2.2.2 新旧架构的差异

新旧系统的主要差别包括：

- 新数据库的拆分维度从冷热数据变更成了根据订单号进行水平拆分；
- 数据库从 1 拆 2，变成了 1 拆 64 解决了磁盘存储空间不足的问题；
- 新数据库的部署方式更加灵活，如果 16 台物理机器资源不足时，可以通过增加服务器的数量快速提高数据库的处理性能；
- 如果 64 个分片的数量不足时，可以通过调整分片计算的组件功能，扩展分片数量；
- 原先的 SQLServer 采用的一主多从一 DR 的模式进行配置。当前系统每个分片物理服务

器采用一主一从一 DR 的模式进行配置；
- 通过增加订单聚合数据库将部分跨分片的数据通过 Binglog+有序消息的方式聚合到新的数据库上，降低跨分片查询带来的性能损失。

2.3 技术方案

在项目执行过程中有非常多的技术细节问题需要分析和解决。我们列举一些在项目过程中可能会遇到的问题：

- 如何选择分片键
- 如何解决跨分片查询性能的损失
- 如何提高开发效率，降低项目风险
- 全链路的灰度切换方案
- 分片故障的处理方案

下面我们就选择几个典型的例子，来说明我们在项目过程中遇到的问题，以及解决这些问题的方案。

2.3.1 分片键选择

分库的第一步也是最重要的一步，就是选择分片键。选择的原则是：

- 分片键必须是不会被更新的字段；
- 各个分库的数据量和读写压力要均匀，避免热点分库；
- 要尽量减少单次查询涉及的分库数量，降低 DB 压力。

分片键的选择，是需要根据具体的业务场景来确定。对于订单数据的拆分，常见的选择是订单 ID 和用户 ID 两个维度，这也是业内最常用的两个分片键。我们最终采用的是主订单 ID，主要是基于四个因素：

- 90%的请求都是基于订单 ID 进行查询；
- 主订单 ID 是对应于用户的一个订单，包含多个行程和贵宾休息室等附加产品，后台会可能将这些拆分为多个子订单，而子订单之间会做 Join 等关联处理，所以不能选择子订单维度；
- 一个主订单可能关联多个用户 ID，比如用户 A 为用户 B 购买机票，用户 B 又可以自己为这个订单添加值机的功能。一个订单 ID 关联了两个用户 ID，从而使用用户 ID 用作分片键会导致订单分布在不同的分片；
- 分销商的订单量非常大，按用户 ID 分库会导致数据不均衡。

我们决定采用主订单号作为分片键后，进行了下列改造，用于实现并且加速分片选择的过程。

（1）订单 ID 索引表

【问题】：如何获取主子订单对应的分片 ID？

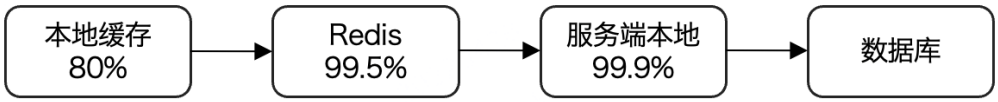
按主订单 ID 分库，首先产生的问题是子订单 ID 如何计算分库，需要查询所有分库么？我们是采用异构索引表的方式，即创建一个订单 ID 到主订单 ID 的索引表，并且索引表是按订单 ID 进行分库。每次查询订单 ID 查询时，从索引表中获取对应的主订单 ID，计算出分库，再进行业务查询，避免查询所有分库。

（2）索引表多级缓存

【问题】：通过索引表查询分片 ID 会增加了查询的二次开销，使查询性能损失严重，如何减少数据库二次查询的开销来提高查询性能呢？

订单 ID 的二次查询，仍然会带来数据库的压力明显上升，实际上订单 ID 是不会更新的，订单 ID 和主订单 ID 的映射关系也是不会发生变化的，完全可以把订单 ID 索引表的信息缓存起来，每次查询时从缓存中就可以获取主订单 ID。

我们设计了多级缓存来实现查询加速，所有的缓存和分库逻辑都封装在组件中，提供给各个客户端使用。三级缓存结构如下：



注：图下方的数字代表在当前缓存和它的所有上级缓存命中率的总和。例如 Redis 的 99.5% 代表 1000 个订单有 995 个在本地缓存或者是 Redis 缓存中命中了。

客户端本地缓存：将最热门的订单 ID 索引存放在应用的本地内存中，只需要一次内存操作就能获取主订单号，不需要进行额外的网络 IO

Redis 分布式缓存：将大量的索引信息存放在 Redis 中，并且所有客户端可以共用 Redis 缓存，命中率超过 99%，并且由于订单的映射关系是不会发生变化的，因此可以在生成订单号的阶段对缓存进行预填充

服务端本地缓存：对 DB 索引表的读取，都是在特定的应用中实现，未命中缓存时客户端是通过服务端获取索引信息。服务端也有本地缓存，使用 Guava 实现用于减缓热点 key 的流量尖刺避免缓存击穿

（3）本地缓存的内存优化

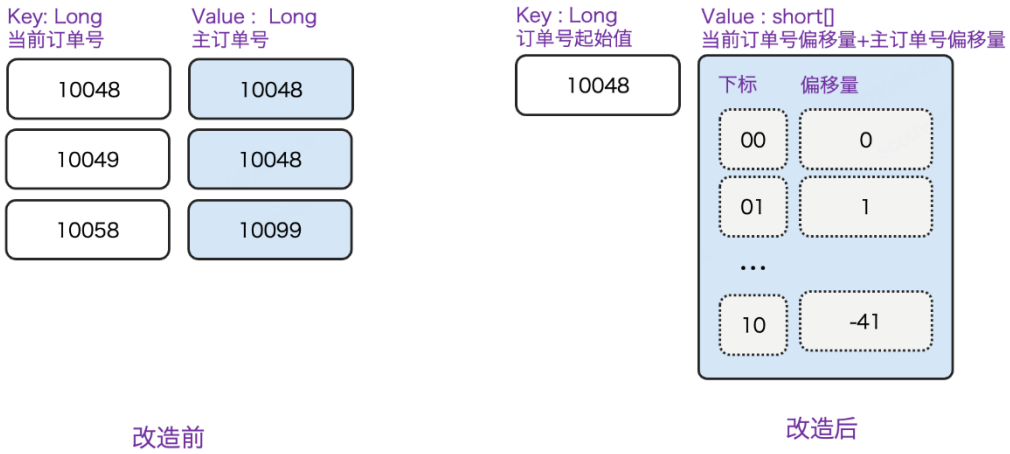
【问题】：使用本地缓存可以减少索引表查询开销，如果需要提高缓存命中率，就需要消耗更多的内存使用，那么如何减少内存占用的问题呢？

本地缓存的效率是最高的，存储在本地的索引信息自然是越多越好。但本地内存是宝贵而有限的，我们需要尽量减少单个索引占用的内存。订单 ID 都是 Long 类型，每个 Long 类型占用 24 个字节，通常情况下，单个索引中包含两个 Long 类型， 还需要缓存内部的多层 Node

节点，最终单个索引大约需要 100 个字节。

我们主要是结合业务场景来改进内存的使用。订单 ID 是有序的，而且主子订单 ID 的生成时间是非常接近的，大部分情况下，主订单 ID 和子订单 ID 的数值差异是很小的。对于连续的数字，数组的方式是非常节省空间的，100 个 Long 类型占用 2400 个字节，而一个长度为 100 的 long 数组，则只占用 824 个字节。同时不直接存储主订单 ID，而是只存储主子订单 ID 的差值，从 long 类型缩减为 short 类型，可以进一步减少内存占用。

最终的缓存结构为：Map<Long, short[]>。从而使整体的内存占用减少了大约 93%的存储空间。也就意味着我们可以适当增加本地缓存的容量，同时减少内存的消耗。



改造后：

【Key】表示订单 ID 所在的桶，计算方式为订单 ID 对 64（数组长度）取模

【下标】表示订单 ID 的具体位置，计算方式为订单 ID 对 64（数组长度）取余数，即【KEY】和【下标】合计起来表示订单 ID

【偏移量】表示主订单 ID 的信息，计算方式是主订单 ID 减去订单 ID

最优情况下，存储 64 个索引只需要一个 Long 类型、一个长度 64 的 short 数组和约 50 个字节的辅助空间，总计 200 个字节，平均每个索引 3 个字节，占用的内存缩减到原来的 100 个字节的 3%。

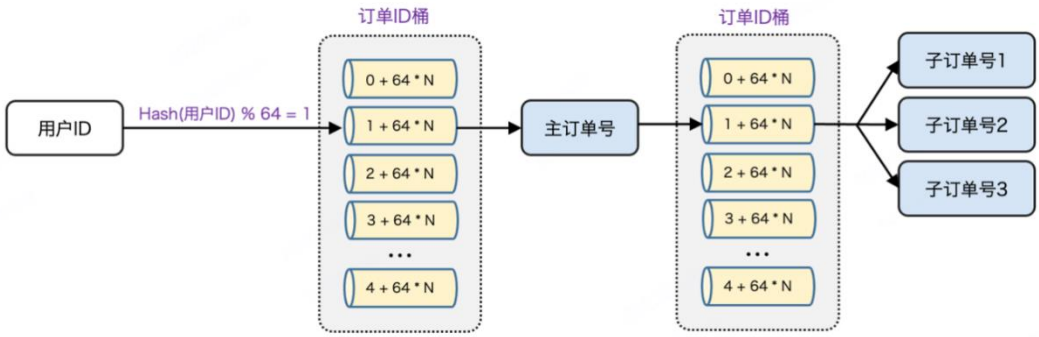
值得注意的是对于偏移量的设计仍然有一定的讲究。我们需要分析主子订单的差异区间范围。Short 的取值范围是-32768 ~ 32767，首先将-32768 定义为非法值。我们还发现大部分的订单分布区间其实并没有和这个取值范围重叠，因此需要额外再给偏移量增加二次偏移量来优化这个问题，实际的取值范围是：-10000 ~ 55534，进一步提高了 short 偏移量的覆盖面。

（4）主子订单 ID 同余

【问题】我们对订单 ID 索引做了各种改进，使它运行的越来越顺畅，但三级缓存的引入，也使得我们的系统结构变复杂，是否有办法跳过索引表呢？

我们将未使用的订单 ID 按余数分成多个桶，新增订单在拆分订单时，子订单 ID 不再是随机生成，而是按照主订单 ID 的余数确定对应的桶，然后只允许使用这个桶内的订单 ID，即保证主订单 ID 和子订单 ID 的余数是相同的。在查询时，子订单 ID 直接取余数就能确定对应的分库，不需要读取订单索引。

再进一步，生成主订单 ID 时也不再是随机选择，而是基于用户 ID 来分桶和选择，做到一个 UID 下的订单会尽量集中到单一分库中。



用户 ID/主订单 ID/子订单 ID 三者同余

2.3.2 跨分片查询优化

数据分库后，当查询条件不是分片键时，例如使用用户 ID、更新时间等作为查询条件，都需要对所有分片进行查询，在 DB 上的执行次数会变为原来的 64 倍，消耗的 CPU 资源也会急剧放大。这是所有分库分表都会遇到的问题，也是一个分库项目最具有技术挑战的环节。我们针对各种场景，采取多种方式来进行优化。

（1）UID 索引表

【问题】UID 是除了订单号以外消耗资源最多的查询之一，大约占用大约 8%的数据库使用资源。使用 allShards 查询会消耗非常多的资源，严重降低查询性能。那么我们如何对 UID 查询进行优化，从而提升查询效率呢？

索引表是一种常见的解决方案，需要满足三个条件：

- 索引字段不允许更新
- 订单库中用户 ID 是不会被更新的
- 单个字段值关联的数据要少，或者关联的分库数量少

关联的数据过多，最终还是到所有分库中获取数据，也就失去了索引表的意义。对于我们的

业务场景来说，用户购买机票是一种较为低频的行为。因此，大部分用户的订单数量相对有限，平均每个用户的订单涉及的分库数量远小于所有分库数量。

- 查询频率要足够高

索引表本质上是一个“空间换时间”的思路，只有足够高的查询频率，有足够的收益，才值的实现索引表。

以用户 ID 作为条件的查询，是业务中非常重要的一类查询，也是排除订单 ID 查询后，最多的一类查询。基于业务和现有数据来分析，由于单个用户购买机票的总数并不是很多，用户 ID 分布在了有限的分库上。我们增加一个用户 ID 索引表，存储用户 ID 与订单 ID 的映射信息，并按照用户 ID 进行分库存储。如下图，每次用户 ID 的查询，会先查询索引，获取包含此用户订单的所有分库列表，通过一次额外的查询，能够快速排除大量无关的分库。再结合前面提及的用户 ID 与订单 ID 同余的策略，单个用户 ID 的新增订单会集中存储在单一分库中，随着历史数据的逐步归档，单个用户查询的分库数量会越来越少。

UserID	PrimaryOrderId	ShardId
M10001	10048	0
M10001	10049	1
M10002	10099	51

UserIDIndex 表结构

（2）镜像库

【问题】：并不是所有的查询都可以像用户 ID 一样，通过建立一个二级索引表来优化查询问题，而且建立二级索引表的代价比较大，我们需要一个更通用的方案解决这些查询问题。

AllShards 查询中的另一类查询就是时间戳的查询，尤其是大量的监控查询，大部分请求是可以接受一定的延迟，同时这些请求只是关注热点数据，比如尚未被使用的订单。

我们新建了一套 MySQL 数据库，作为镜像库，将 64 个分库中的热点数据，集中存储到单一数据库中，相关的查询直接在镜像库中执行，避免分库的问题。

镜像库的数据同步，则是通过 Canal+QMQ 的方式来实现，并定时对比数据，业务应用上则是只读不写，严格保证双边数据一致性。

（3）ES/MySQL 对比

【问题】：镜像库存在多种实现方案，很多系统采用了 ES 作为查询引擎，我们该如何选择？ES 也是解决复杂查询场景的一种常见方案，我们曾经考虑采用 ES 来提升查询性能，并且进行了详细的评估和测试，但最终放弃了 ES 方案，主要考虑到以下几点原因：

项目前期对所有的查询进行了充分简化和规整，目前所有的查询使用 MySQL 都可以很好的运行。

在已经正确建立索引和优化 SQL 语句的情况下，MySQL 消耗的 CPU 可能远小于 ES，尤其是订单 ID、时间戳等数字类型的查询，MySQL 消耗的 CPU 只是 ES 消耗的 20%甚至更低。

ES 并不擅长数字查询，而是更合适索引字段多变的场景。

因此具体采用 ES 还是 MySQL，或者是其他数据库来建立镜像数据库，最重要的一点还是要基于现有的业务场景和实际生产上的需求进行综合分析和验证后，找出一个最适合自己当前情况的方案。

2.3.3 双写组件设计

因为技术栈的问题，目前我们的 ORM 采用的是公司的 DAL 组件。这个组件本身对公司的环境支持较好，而且该组件对于 Sharding 数据库也提供了良好的支持。因此我们在该项目上仍然使用 DAL 作为我们数据库的访问组件。

但是原生的 DAL 并不支持双写的功能，不支持读写的切换。针对项目的特性，我们需要尽可能的让开发少感知或者不感知底层数据库的双写和读写切换的操作。一切对于用户来说变得更简单、更透明。另一方面，我们打算优化组件本身的使用接口，让用户使用起来更傻瓜化。

组件的升级改造需要符合以下原则：

- 对业务代码侵入少
- 改造少，降低工作量
- 使用简单
- 符合直觉

这些改造的意义是非常重大的，它是我们能够高质量上线的关键。于是我们对组件进行了一些封装和优化。

（1）业务层对象和数据库层对象进行隔离

为了统一维护方便我们将团队内所有的数据库对象（Pojo）都维护在了公共组件中。因此，在公共 jar 包中生成的对象通常是一个大而全的数据库实体。这种大而全的实体信息存在以下几个问题：

- 单表查询时直接只用 pojo 返回了全量信息，影响查询性能；
- 直接在代码中使用 pojo 带来了大量无用的字段，不符合按需使用的原则；
- 很难统计应用对于数据库字段的依赖的问题；
- 数据库字段和代码直接耦合，在代码编写期间不能对字段的命名等问题进行优化。

为了解决以上问题，我们中间新增了 Model 层，实现数据库 pojo 和业务代码的隔离。例如我们的航班信息表（Flight）有 200 多个字段，但是实际在代码中仅需要使用航班号和起飞时间。我们可以在业务代码中定义一个新的 FlightModel，如下图所示：

```
@Builder
public class Flight implements DalDto {
    /**
     * 订单号
     */
    private Long orderId;

    /**
     * 航班号
     */
    [DalField="flight"]
    private String flightNo;
}
```

扩展组件将该对象映射到数据库的 Pojo 上，并且可以改变字段的命名甚至类型从而优化代码的可读性。在数据库查询时也进行了优化，仅仅查询必要字段，减少了开销。

（2）双写功能

我们实现的双写方案是先写 SQLServer 再写 MySQL，同时也实现了失败处理相关的策略。

双写模式包括：

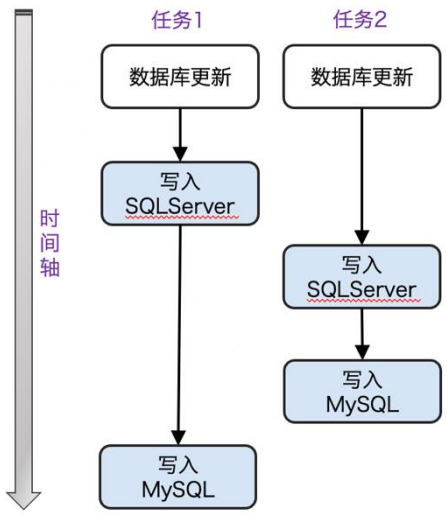
- 异步双写

这个主要是在双写功能实现的初期，我们会使用队列+异步线程的方式将数据写入到 MySQL。采取这种方式的数据一致性是比较差的，之所以采用这种方式也是在初期我们对数据库处于探索阶段，避免 MySQL 数据库故障对当前系统产生影响。

- 同步双写

当 SQLServer 写入成功后，在相同的线程中对 MySQL 进行写入。这种模式相对来说数据一致性会比较好，但是在极端情况下仍然可能存在数据不一致的情况。

如下图所示。当任务 1 更新 MySQL 数据库之前，如果有别的任务抢先更新了相同的数据字段就有可能产生脏写的问题。



我们可以通过以下手段减少数据不一致的问题:

- 数据表的读写尽可能收口;
- 访问收口以后，通过对业务系统增加分布式锁等手段缓解此类问题的产生;
- 可以增加数据比对的工具，主动发现数据的不一致并进行修复，通过一个异步的扫描时间戳的工具来主动进行数据对比注和修复;
- 写入失败需要根据当前的模式触发自动补偿的策略，这个可以参考下文提到的数据双写异常的补偿方案。

注：数据对比和补偿需要注意热点数据频繁更新和由于读取时间差导致的不一致的问题

刚才提到我们抽象了 Model 层的数据，在此基础上，我们的双写改造对用户来说非常的容易。

```
@DalEntity(primaryTypeName = "com.ctrip.xxx.dal.sqlserver.entity.FlightPojo",
           secondaryTypeName = "com.ctrip.xxx.dal.mysql.entity.FlightPojo")
@Builder
public class Flight implements DalDto {
```

我们仅需在 Model 对象上增加 DalEntity 注解实现数据库 Pojo 的双边映射。除此之外，开发人员不需要对业务代码做其他调整，即可以通过配置实现双写、数据源切换等操作。

（3）双写异常处理模式

双写时，我们需要尽可能保证数据的一致性，对于 MySQL 数据写入异常时，我们提供了多种异常处理模式。

- AC

异步双写时，如果从库发生异常进行数据捕获，不抛出异常，仅输出告警信息

- SC

同步双写时，如果从库发生异常进行数据捕获，不抛出异常，仅输出告警信息

- ST

同步双写时，如果从库发生异常，抛出异常，中断处理流程

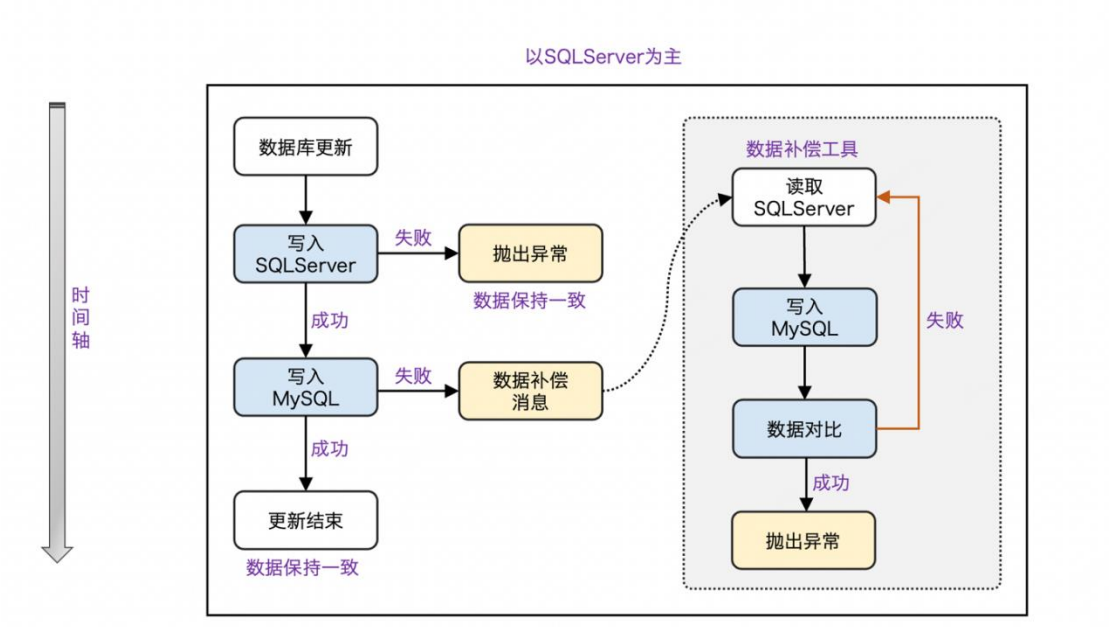
（4）双读功能

双写功能相对比较好理解。在灰度切换过程中，假如存在灰度控制的订单 A 以 SQLServer 为主，订单 B 以 MySQL 为主。但是我们查询到结果中同时包含了订单 A 和订单 B 的场景。这个时候我们希望的是，同时查询 SQLServer 和 MySQL 的数据源，并且从不同数据源中获取相应的订单数据，然后进行组合、排序、拼接。这些筛选逻辑由我们的组件来自动完成，从而实现了更加精细的灰度控制。

值得注意的是 allShard 查询的结果在部分情况下（例如分页查询）和单库查询的结果存在较大的差异，也需要组件的支持。

（5）数据写入异常的补偿方案

我们需要在不同阶段设计不同的补偿方案。初期 MySQL 的数据并不会对外提供服务，即使数据写入失败，也不能影响系统流程的正常运行，同时也要保证数据写入的准确性。因此，我们采用了 SC 的异常处理模式，并且增加了主动和被动的数据补偿。



但是我们的目标是使用 MySQL 的数据。因此，当主数据源需要 SQLServer 切换到 MySQL 后，虽然数据库写入的顺序仍然保持先写 SQLServer 再写 MySQL，但是数据写入失败的处理模式需要发生变化。

这里先插播一个问题，就是为什么不能先写 MySQL 然后同步更新 SQLServer。主要考虑到以下两个因素：

- 数据库主键生成的历史遗留原因

由于 MySQL 是 Sharding 数据库，如果先插入该数据库，默认情况下会通过雪花算法生成主键。写入完成后，我们将该主键同步给 SQLServer。

但是受到公司 ORM 框架和历史遗留的技术限制，SQLServer 不会使用该数据，仍然采用自增的方式生成主键。导致数据严重不一致。

- 数据双向同步的复杂度问题

当我们以 SQLServer 作为主数据库时，如果数据不一致需要同步给 MySQL（异步存在延时）；当以 MySQL 作为主数据库时，如果发生数据不一致，需要进行反向同步。

一来，数据补偿程序复杂度很高。二来，如果我们如果在 MySQL 和 SQLServer 数据库谁作为主库之间切换频繁，数据同步程序就会变得非常迷茫，到底谁该同步给谁？

那么如何提高在以 MySQL 为主的情况下，双边数据库的一致性呢？

首先，我们得关闭自动补偿功能，异常处理模式需要从 SC 切换到 ST，遇到 MySQL 失败直接抛出数据库异常，然后基于系统的业务场景进行如下操作：

- 依赖业务系统的自动补偿

对于订单处理系统，大部分的流程其实具备了自动补偿的能力，因此哪怕 SQLServer 更新成功，MySQL 未成功。下次补偿程序仍然读取 MySQL，SQLServer 会被二次更新，从而达到最终一致性。

这个时候，需要考虑的 SQLServer 的可重入性。

- 无法自动补偿的场景，提供手工数据补偿的功能

因为此时 MySQL 已经作为主要数据源，如果 SQLServer 存在不一致的场景可以提供手工的方式将数据补偿回 SQLServer。这边没有实现自动补偿，因为理论上只有在数据不一致的场景，并且发生了回切才会产生影响。

- 数据的比对功能仍然正常开启，及时发现数据的不一致

（6）组件设计的功能和策略分离

我们整体的功能都整合在名为 Dal-Extension 的系统组件里，主要分为功能实现和策略两大部分。

功能就是前面提到的例如双写，读切换，异常处理模式切换等。策略就是引擎，它实现了功能和功能间的联动。例如上文提到的，如果以 SQLServer 作为主数据源，那么系统自动采用 SC 的异常处理模式，并且主动调用数据补偿功能。如果是以 MySQL 作为主数据源，那么系统自动切换到 ST 的异常处理模式。

相较于基于应用、表维度的切换策略。我们提供了维度更丰富的切换组合策略。

- 表
- 应用/IP 地址
- 读/写
- 订单区间

通过对以上维度的配置进行灵活调整，我们即可以实现单表，单机器的试验性切换控制，也可以进行全链路的灰度切换，确保一个订单在整个订单处理生命周期使用相同的数据源，从而避免因为数据双写或者同步导致的数据读取结果不一致的问题。整体的数据切换操作由配置中心统一托管。

2.3.4 分片故障处理

原先的数据库如果发生了故障，会导致整个系统不可用。但是新的数据库扩展成 64 个分片后，其实相对来说故障概率提高了 64 倍。因此，我们需要避免部分分片故障导致整个系统失效的情况。另外增加故障转移和隔离功能，避免故障扩散，减少损失也是我们重点关注的功能。

（当然，如果发生分片故障，首选的故障恢复方案是数据库的主从切换）。

（1）返回仅包含查询成功分片的部分数据

【问题】针对跨分片查询的场景，如果一个分片故障默认情况下会导致整个查询失败，那么如何提高查询成功率呢？

我们调研了数据使用端，发现有很多场景，例如人工订单处理的环节，是可以接受部分数据的返回。也就是说有查询出尽可能多符合条件的订单，放入人工待处理列表中。我们增加了 continueOnError 参数来表示当前查询可以接受部分分片失效的场景。并且，系统返回了查询结果后，如果存在分片查询失败的场景，系统会提供了错误分片的信息。这样业务上不仅能够确保了很多业务环节处理不中断，同时针对它提供的错误分片信息可以让我们快速感知失效的分片，以便系统自动或者人工对这些分片进行干预。

（2）故障分片隔离

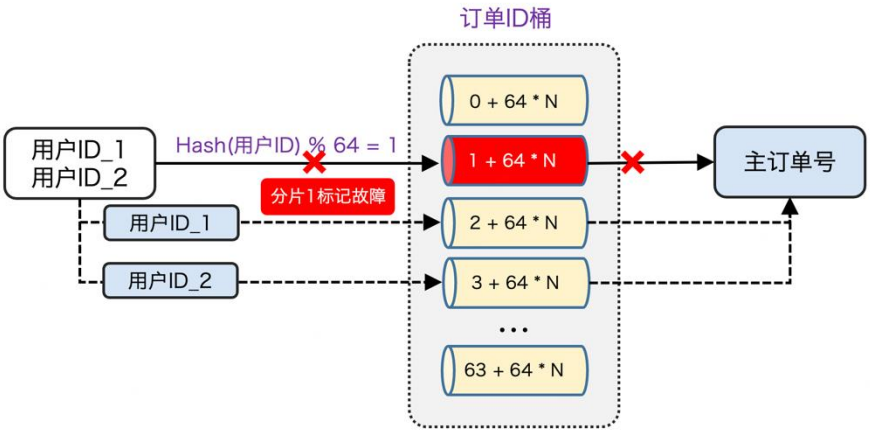
【问题】当故障分片出现大规模错误后，如果是因为响应时间长会导致大量线程 block，从而拖累整个应用服务器。那么如何解决此类问题呢？

当分片发生故障时，有可能我们的数据库请求被 hang 住。我们 allShards 查询的底层实现是基于共享线程池。当部分分片的响应慢时，会拖累整个线程池。另外单表查询时，也可能会因为数据库响应时间的问题导致工作线程数量上涨的问题。

我们为此增加了分片屏蔽的参数。当我们启用分片临时屏蔽功能后，底层数据库查询时，发现该分片被屏蔽直接抛出异常，让应用程序能够得到快速响应。从而避免了网络和数据库访问时间消耗，提高了异常执行的效率，避免问题扩散到正常的分片的数据处理。

（3）故障订单转移

【问题】根据之前的介绍，用户订单号是根据 UID 的哈希值进行分配的。也就是说同一个用户分配的分片是固定的。如果该分片故障时，用户就无法提交订单。那么如何避免或者减少此类问题呢？



如上图所示，用户 ID_1 和用户 ID_2 根据哈希算法，原先会在分片 1 上生成订单。但是如果发生了分片 1 故障时，我们的 UID 分片计算组件会将分片 1 标记为不可用，然后通过新的 Hash 算法计算出新的分片。

这里需要注意的是，新 hash 算法的选择。

- 方法 1:

使用同样的哈希算法，但是生成结果后取模的值为 63（64-1），但是这个存在的问题是用户 ID_1 和用户 ID_2 计算出来的分片结果是一致的。假如新的分片号为 2 的话，如果发生分片 1、分片 2 同时失效的情况下。那么仍然有 1/64 的订单出现问题。

- 方法 2:

采用新的哈希算法，尽量使订单分布在出了分片 1 以外的其他分片上。那么这种方法，即使

分片 1、分片 2 同时失效。那么仅仅会影响到 1/64 * 1/63 的订单。受影响的订单量大幅降低。

三、项目规划

除了以上提到的技术问题以外，我们再谈谈项目的管理和规划问题。首先，圈定合理的项目范围，划清项目边界是项目顺利实施的重要前提。这个项目的范围包括两个重要的属性：数据和团队。

数据范围

（1）划定数据表范围，先进行表结构优化的工作

我们需要在项目初期明确数据表的范围，针对一些可以下线的表或者字段，先完成合并和下线的工作，来缩小项目范围。避免表结构的变化和该项目耦合在一起，造成不必要的困扰。

（2）相关数据表中哪些数据需要被迁移

我们在处理这个问题上，有一些反复。

- 方案 1：仅迁移热数据

因为订单数据分为冷热数据，所以我们最开始考虑是不是只要迁移热数据就好了，冷数据仅保留查询功能。

但是，这个方案有两个很大的问题：一是存在冷数据需要被还原到热数据的场景，增加了系统实现的复杂度。二是冷数据保留时限的问题，无法在短时间内下线这个数据库。

- 方案 2：部分数据自然消亡的表和字段不进行迁移

针对有一些表由于业务或者系统改造的原因，可能后续数据不会更新了，或者在新的订单上这些字段已经废弃了。大家在设计新表的时候其实往往很不喜欢把这些已经废弃了的信息加到新设计的表中。但是，我们需要面临的问题是，旧数据如何兼容是一个非常现实的问题。

因此，当我们开发到中间的过程中，还是将部分表和字段重新加了回来。来确保旧数据库尽快下线以及历史逻辑保持兼容。

- 方案 3：保留当前所有的表结构和信息

我们最后采用了这个方案，哪怕这个数据表或者字段未来不会做任何修改。

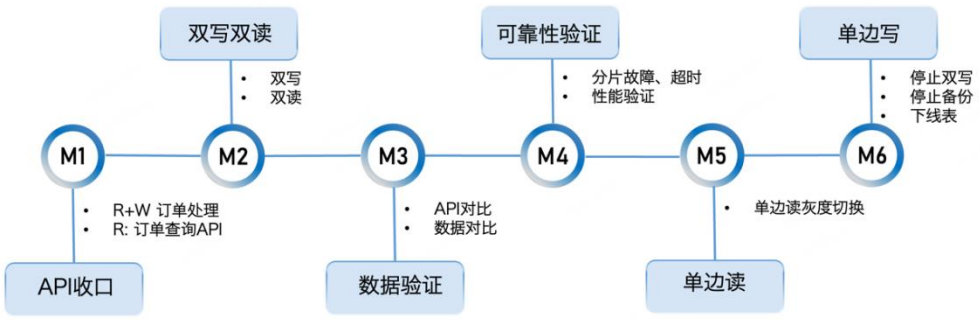
团队范围

确定好数据范围后，我们需要根据这些数据，确定我们需要做的工作以及找到完成这些工作的相关团队并提前安排好资源。整个项目的资源分为核心成员和相关配合改造的团队。

核心成员需要做到组织分工明确，并且需要经常一起头脑风暴，提出问题，解决问题，消除隐患。核心成员的另一个职责是帮助配合改造的团队，协调并且解决技术问题、资源问题等等。特别是涉及到的改动点较多、改造难度较大的团队，需要提前介入，在适当的时候提供更多的帮助。

3.1 规划

确定了项目的目标和范围后，我们为项目设计了 6 个里程碑，来帮助我们更好的完成这个项目。



- 阶段 1：通过 API 对读取进行收口

这个阶段虽然难度并不大，但是周期很长，沟通成本较高。在这个阶段重点在于任务的协调和跟进。DBA 帮助我们研发了生产 Trace 查询的工具，能够准实时的知道数据表的访问情况，帮我们快速验收并且圈定改造范围。

我们建立了任务的看板，为每一个任务设定了负责人以及预期解决的时间，定期对任务进行进行跟踪。项目的负责人也作为验收人，确认每个任务的完成情况。

通过一段时间的努力，数据库的访问收口在极少数内部应用当中。实现了数据访问的收口。

- 阶段 2：开发双读/双写功能来实现平滑的数据切换

这个阶段需要将整个项目的技术点、难点都逐一的找到，并且给出解决方案。如何提高效率和质量也是这个阶段重点关注的话题，我们尽量把这些双写、切换的功能进行封装，让业务逻辑层尽可能少感知，或者不感知这些底层逻辑。降低代码开发量，不仅能提高效率，还能提升质量。

总的来说，这个阶段需要提升开发效率，提高开发质量并且降低项目风险。

- 阶段 3：验证数据一致性

这个是对阶段 2 的验证工作，需要注意的是在验证中减少噪音，提高验证的自动化率，能有效的提升项目的开发质量。

- 阶段 4：通过压测，故障模拟等手段验证系统性能。

在数据库故障时，提供可靠的系统的灾备和故障隔离能力。

- 阶段 5：数据读取从 SQLServer 切换到 MySQL

这个阶段可能不需要有太多的资源投入，但是风险却是最大的。这个阶段是对前面所有阶段成果的验收。做好数据监控、制定良好的切换方案、出现问题时能够回退是这个阶段顺利实施的重点。

- 阶段 6：停止 SQLServer 写入并且下线相关数据表

相比起阶段 5，阶段 6 没有后悔药。一旦停止了 SQLServer 的写入，就非常难进行回切的操作。所以得仔细做好白名单的验证，并且及时响应和解决相关问题。

3.2 原则

整个项目周期较长，我们需要制定好每个阶段的目标，每个任务的目标。由于数据库承载了非常核心的业务，因此整个阶段、所有任务以及技术方案其实围绕着一个原则展开，就是降低风险。

所以我们在设计每个技术方案的时候，尽可能考虑这点。例如在数据源切换的开关虽然涉及较多的服务实例，但是我们通过一个集中控制的平台，来实现全链路的切换和灰度控制。

四、经验分享

该项目整体的周期较长，每个阶段的挑战不尽相同。为了确保项目的上线质量，后续在读切换、写切换两个流程的灰度时间比较久。项目大约在 2021 年下半年顺利完成。

实现了以下主要目标和功能：

- 系统的水平扩展能力得到大幅提升

系统分片数量为 64，部署在 16 台物理机上。后续根据业务需要机器的部署方式和分片数量可以进行灵活调整。

- 数据库资源利用率大幅下降，可靠性提升

数据库服务器的 CPU 利用率从高峰期 40%下降到目前的 3%-5%之间。

- 订单处理能力提升和存储能力提升

原先区分冷热数据，热数据大约仅能支持 3 个月的订单，按照现在硬件资源推算，系统可以处理至少 5 年以上的订单。

- 数据访问收口

原先近 200 个应用直接访问数据库，给我们的改造带来很大的不便。目前仅有限的内部应用允许直接访问订单库。

- 整体成本下降

原先主从服务器的 CPU 为 128 核，内存 256G；现在服务器缩减为 40 核心的标准配置。

在项目过程中也积累了不少的经验，例如：

- 项目的规划要清晰，任务要明确，跟踪要及时

整个项目中大约建立了数百个子任务，每一个任务需要落实负责人以及上线时间，并对上线结果进行验收。才能确保整个项目的周期不至于拉的非常长，减少后续的项目返工和风险。

- 减少例外情况的发生

当一个大型的项目存在非常多的例外情况，这些特殊情况就得特殊处理，那么到最后总会有一些没有处理干净的尾巴。这些问题都是项目的潜在隐患。

- 减少项目的依赖

这个和我们日常开发关系也非常密切，当一项任务有多个依赖方的时候，往往项目的进展会大幅超出我们的预期。因此减少一些前置依赖，在不是非常确定的情况下。我们得先做好最坏的打算。

- 一次干好一件事

很多时候我们往往会高估自己的能力，例如在这次的改造中，我们会顺便优化一些表的结构。于是造成了 MySQL 和 SQLServer 的数据表差异过大的问题。那么这些差异其实为后面的开发造成了不小的困扰。所有的方案，包括数据补偿、迁移、数据源的切换等等场景都得为这些特殊差异的表单独考虑方案，单独实现逻辑。一不留神或者没有考虑的很周全的情况下，往往会漏掉这部分的差异。导致项目返工，甚至出现生产故障。

项目的成功上线离不开每一个成员的努力。在实施过程中，遇到的问题比这篇文章列举的问题多得多，很多都是一些非常琐碎的事情。特别是项目初期，我们往往是解决了一个，冒出了更多的问题。但是每次遇到问题后，团队的成员都积极思考，集思广益，攻破了一个又一个的技术问题和业务问题。通过一年多时间的锻炼，团队成员的项目能力、技术能力进步显著；发现问题的角度更敏锐，思考的角度更全面；团队的凝聚力也得到了明显提升。

携程海外 MySQL 数据复制实践

【作者简介】Joy，携程软件技术专家，负责 MySQL 双向同步 DRC 和数据库访问中间件 DAL 的开发演进，对分布式系统高可用设计、分布式存储，数据一致性领域感兴趣。

一、前言

在携程国际化战略背景下，海外业务将成为新的发力点，为了保证用户高品质的服务体验，底层数据势必需要就近服务业务应用。一套标准且普适的数据复制解决方案能够提升业务决策效率，助力业务更快地触达目标用户。

DRC (Data Replicate Center) 作为携程内部数据库上云标准解决方案，支撑了包括但不限于即时通讯、用户账号、IBU 在内的核心基础服务和国际业务顺利上云。

二、业务上云场景

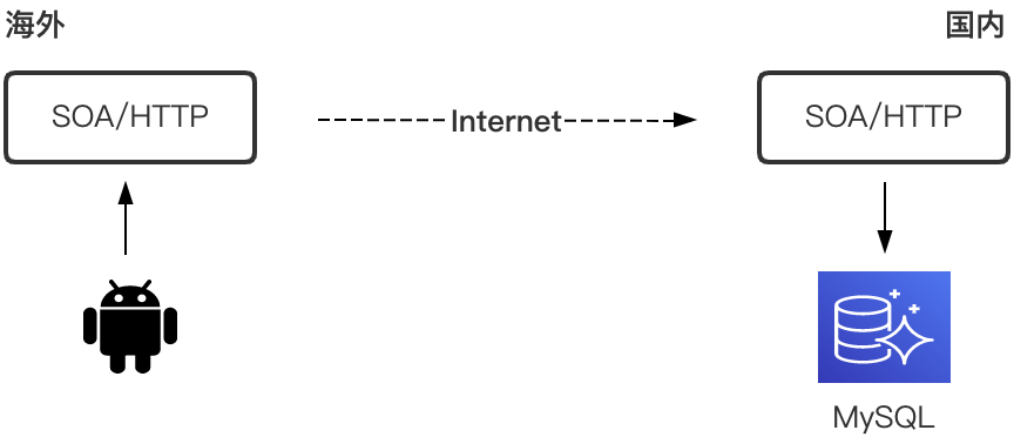
业务上云前，要先要思考 2 个问题：

- 数据库是否需要上云？
- 在数据库上云情况下，海外数据库提供只读还是读写操作？

2.1 应用上云

针对用户延迟不敏感或者离线业务，可以采用只应用上云数据库不上云，请求回源国内。该方案下业务需要改造应用中读写数据库操作，根据应用部署地，决定流量是否需要转发。

不建议海外应用直连国内数据库，网络层面专线距离远，成本太高，不现实；安全层面应禁止跨海访问，否则可能导致预期就近访问流量由于非预期错误，将海外流量写入国内数据库，从而引起国内数据错误。

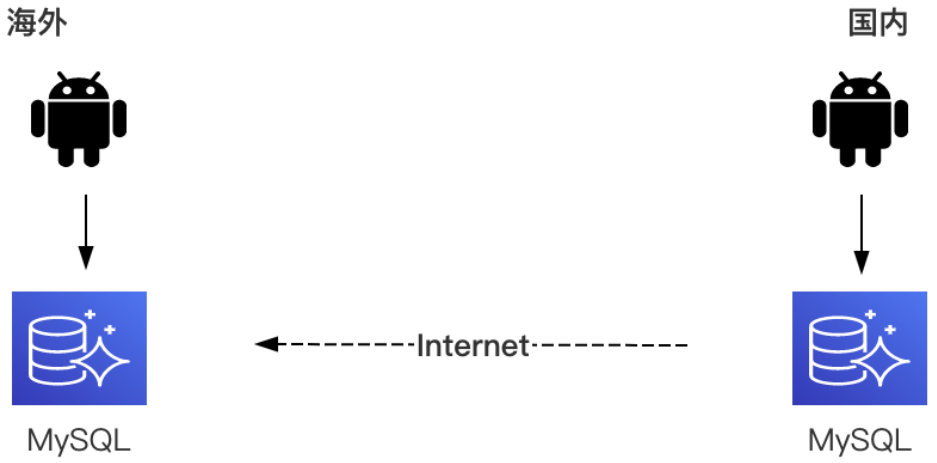


2.2 数据库上云

对于在线用户延迟敏感应用，数据库必须跟随应用一同上云，将请求闭环在海外，从而就近提供服务响应。在确定数据库上云的前提下，根据不同业务特点，可再细分为海外只读和读写两种场景。

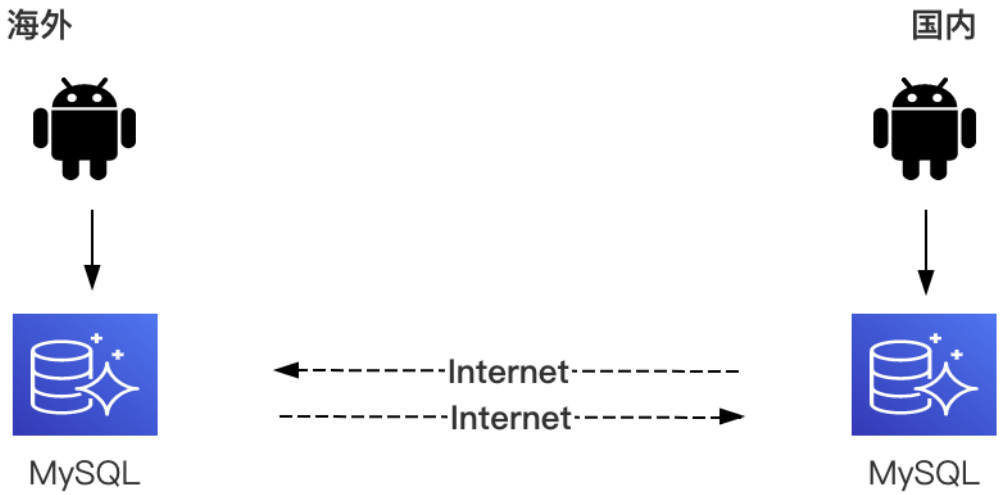
- 只读场景

对于海外只读场景，国内数据只需要单向复制，该方案下业务海外账号默认无写权限或者业务改造写操作，避免出现由于误写导致国内海外数据不一致。



- 读写场景

对于海外读写场景，国内海外数据需要双向复制，业务代码无需改造。该方案下由于有 2 个 Master 可以写入，业务需要在应用层对流量进行切分，比如用户归属地维度，从而避免在两侧同时修改同一条数据，进而导致复制过程出现数据冲突。



2.3 上云成本

数据距离用户越近，应用直接提供的服务功能越丰富，对应业务改造量越小，机器资源消耗量越大。携程海外应用部署在 AWS 公有云上，AWS 入口流量不计费，只针对出口流量计费。应用上云数据库不上云场景，请求回源国内产生出口流量费用；只读业务单方向数据复制流入，不收费；读写业务数据复制回国内产生出口流量费用。

上云场景	AWS出口流量	数据库成本	机器成本	业务改造
应用上云	业务请求流量	无	无	改造读写请求
数据库上云/只读	无	RDS费用	单向复制	改造写请求
数据库上云/读写	海外→国内复制流量	RDS费用	双向复制	无

上云成本主要集中在流量和数据库费用。AWS 出口 Internet 流量 0.09$/GB，当流量大时，可通过数据压缩，损耗复制延迟降低出口流量；RDS 根据核数计费，1004 元/核/月，业务流量少时采用普通 4C16G 机型即可，流量增加后动态提升配置。核心业务 RDS 配置一主一从，非核心业务单主即可，并且多个 DB 可共用一个集群，进而降低成本。

2.4 小结

为了提供高品质的用户体验，数据势必需要上云。在解决了是否上云的问题后，如何上云就成为新的疑问点。下面就详细分析携程内部上云过程中依赖的数据库复制组件 DRC 实现细节。

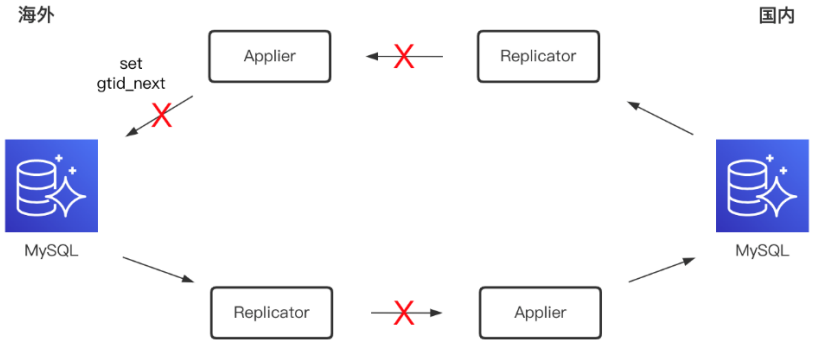
三、数据库上云方案

DRC 基于开源模式开发，公司内部生产版本和开源保持一致，开源地址 https://github.com/ctripcorp/drc，欢迎关注。

DRC 孵化于异地多活项目，参见《携程异地多活-MySQL 实时双向（多向）复制实践》，解决国内异地机房间数据库同步问题。当其中一个或多个机房位置转变为公有云时，伴随着物理距离的扩大，新的问题应运而生。

就 DRC 自身架构实现而言：

- 公有云和国内机房间互不联通，同步链路被物理阻断；
- 公网传输不如国内跨机房之间专线质量，丢包频发；
- 公有云数据库自主运维灵活性下降，如无法获取 root 权限，直接导致 set gtid_next 无法正常工作.



就业务接入而言：

- 国内海外数据隔离，按需复制成为刚需；
- 公有云数据库成本压力导致混部，一对一复制不再满足业务灵活多变的真实部署场景.

基于以上限制，DRC 调整架构，引入代理模块解决网络联通性问题，借用事务表降低复制链路对权限的要求；为了适应业务的多样性，分别从库、表和行维度支持按需复制。

3.1 架构改造挑战

3.1.1 架构升级

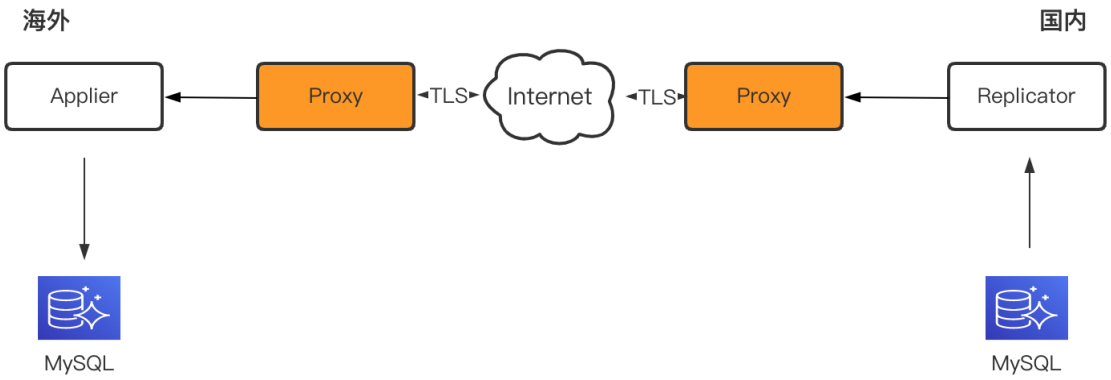
DRC 中有 2 个核心功能需要跨公网传输数据：

- 业务 Binlog 数据复制
- DRC 内部延迟监控探针

（1）数据复制

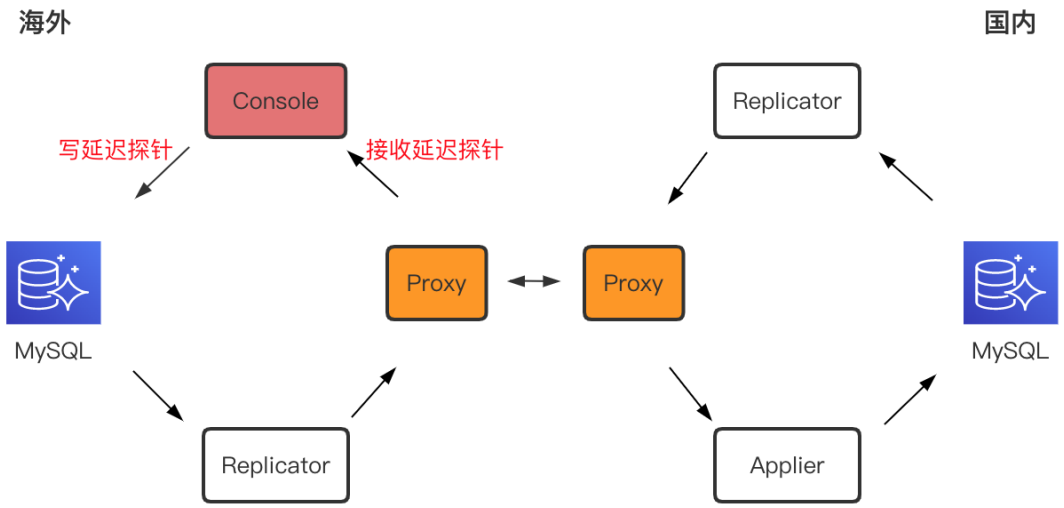
以单向复制为例，在 Binlog 拉取模块 Replicator 和解析应用模块 Applier 之间引入 Proxy，负责在 TCP 层将内网/公网流量转发到公网/内网。Proxy 绑定公网 IP，采用 TLS 协议加密传输内网流量。鉴于公网质量不稳定特性，Proxy 使用 BBR 拥塞控制算法，优化丢包引起的卡顿。

Proxy 作为公网数据传输携程内部统一的解决方案，参见《携程 Redis 海外机房数据同步实践》，开源地址：https://github.com/ctripcorp/x-pipe，欢迎关注。



（2）延迟监控

延迟监控探针从业务流量同侧机房的 Console 写入到业务数据库延迟监控表(初始化时新建)，经过双向复制链路，从异侧机房接收延迟探针，从而计算差值得到复制延迟。为了提升 Proxy 间隔离性，数据复制和延迟监控可以分别配置不同的 Proxy 实例实现数据传输。



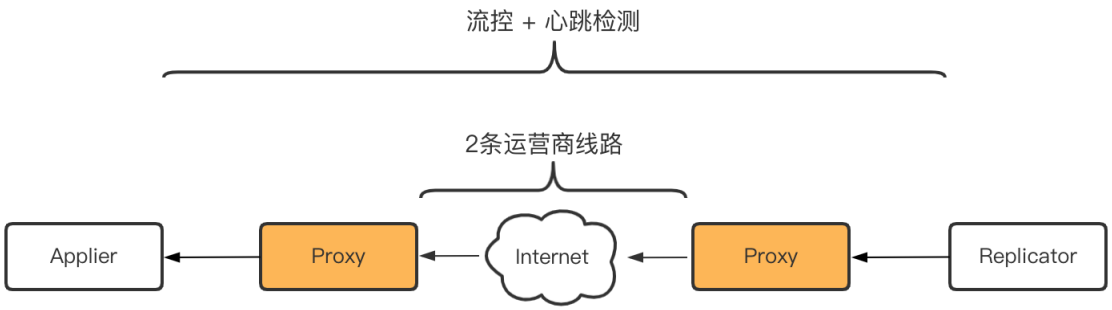
（3）Proxy Client

由于 Applier 和 Console 都需要对接 Proxy，如何降低 Proxy 对 DRC 系统的侵入性就成为一个需要解决的问题。为此我们借助 Java Agent 技术，动态修改字节码，实现了可插拔的接入方式。接入方只需要引入 proxy-client 独立 Jar 包，业务层按需实现 Proxy 的注册和注销。

3.1.2 网络优化

公网网络丢包和拥塞频发，为了在弱网环境下实现平稳复制，就需要快速地异常检测恢复机制。除了在系统层将 Proxy 拥塞控制算法优化为 BBR 外，DRC 在应用层额外增加：

- 心跳检测，实现连接自动切换
- 流量控制，避免突增流量引起资源耗尽进而影响数据复制
- 2 条互备海外出口运营商线路，随机切换



（1）心跳检测

Binlog 生产方 Replicator 定时对下游消费方进行心跳检测，消费方接收到心跳检测需回复响应，Replicator 根据最后一次接收时间检测并自动关闭长期没有响应的连接。

这里有一种场景需要特别处理，当下游消费方比较忙，主动关闭连接 auto_read 属性时，由

于应用层无法读取暂存在缓冲区的心跳包，从而造成无法响应。这就需要消费方在 auto_read 改变时，主动上报生产方自身的 auto_read 状态。

（2）流量控制

公网网络质量下降导致复制延迟变大，数据堆积在发送端 Proxy，进而引起 Replicator 和 Proxy 触发流控；MySQL 性能抖动，应用 Binlog 速度减缓，数据堆积在 Applier，进而引起 Applier 触发流控并逐层反馈到 Replicator。

（3）运营商线路

针对 Proxy 出口 IP，分别配置移动和联通两条运营商线路，当 Binlog 消费方由于触发空闲检测出现超时重连时，Proxy 会随机选择一个运营商出口 IP，从而实现运营商线路的互备。

3.1.3 事务表复制

国内机房间数据复制时，DBA 可以给予 DRC 拥有 root 权限的账号，以实现 Applier 模拟原生 Slave 节点 set gtid_next 工作方式应用 Binlog，从而将一个事务变更从源机房复制到目标机房，并且在两端分配到同一个 gtid 下。但是公有云上 RDS 出于安全原因是无法开放 root 权限，直接从原理上否定了原有的复制方案。

为了找到合理的替换方案，我们首先从 MySQL 服务端视角分析下 set gtid_next 的效果：

事务在提交后会被分配指定的 gtid 值，否则 MySQL 服务端会自动分配一个 gtid 值

gtid 值加入 MySQL 服务端全局变量 gtid_executed 中

其根本性作用在于将 DRC 指定的 gtid 值保存到 MySQL 系统变量。既然无法利用 MySQL 系统变量，那么从业务层增加一个复制变量保存 gtid 信息即可实现同等效果。

其次，转换到 DRC 复制视角，set gtid_next 起到如下作用：

- 记录 Applier 复制消费位点，并以此向 Replicator 请求 Binlog
- 解决循环复制，Replicator 根据 gtid_event 中的 uuid 判断是否是 DRC 复制产生的事件

综上分析，新的替代方案需要引入持久化变量，记录复制位点并且能够提供循环阻断信息功效，为此 DRC 引入基于事务表的同步方案解决了海外复制难题。

（1）位点记录

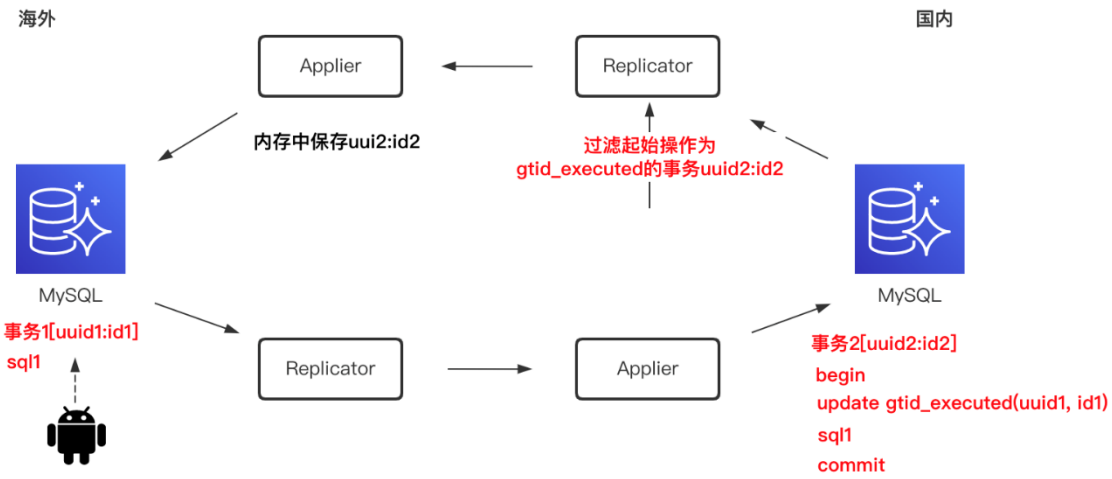
海外复制业务集群需要新增复制库 drcmonitordb，其中新建事务表 gtid_executed。

```
CREATE TABLE `drcmonitordb`.`gtid_executed` (
  `id` int(11) NOT NULL,
```

```
  `server_uuid` char(36) NOT NULL,
  `gno` bigint(20) NOT NULL,
  `gtidset` longtext,
  PRIMARY KEY (`id`,`server_uuid`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

- server_uuid：源端数据库 UUID 号
- gno：事务 id。该列值为 0 的行为汇总行
- gtidset：对于 gno=0 的汇总行，该列批量存储 gno 编号，例如 server_uuid:1-10:20:30

当 Applier 应用 SQL 到目标数据库前，需要先更新事务表，记录 gtid，然后再执行事务中变更语句，完整的复制流程如下图所示。事务表中 gno=0 行中 gtidset 等效 MySQL 系统变量 gtid_executed，Applier 执行过程中定时汇总非 0 行事务 gno，从而达到记录位点功能。



（2）循环阻断

针对 Binlog 中第一个写事件是事务表 gtid_executed 操作的事务，Replicator 将其判断为 DRC 复制数据，从而阻断循环复制，否则一条数据会在双向复制环内无限死循环。

3.2 业务落地挑战

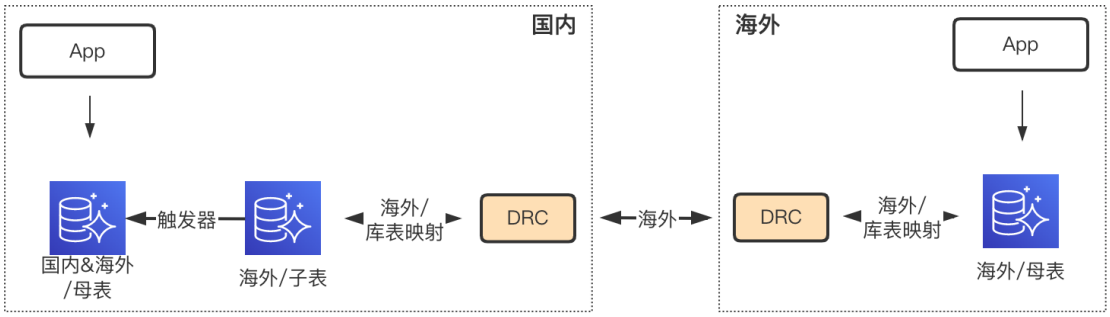
至此 DRC 解决了理论上阻碍复制的已知技术问题，在实际业务落地过程中，出于数据安全、费用和改造成本的考虑，业务对数据复制提出了更精细化控制的需求。

3.2.1 数据隔离

出于合规的要求，业务上云后，需要完成国内和海外用户数据的隔离。业务上云前，国内和海外用户数据全部在国内数据库；上云时就需要将海外用户数据单独复制到公有云而过滤掉国内用户数据。

（1）库表映射

上云前国内和海外数据在同一张母表。为了上云，业务通过在国内数据库新增子表，实现国内数据的分离。海外由于只存在海外数据，所以物理上只需要一张母表即可，即国内子表与海外母表相对应，搭建 DRC 实现双向复制即可。由于母表和子表表名不同，复制时需要做库表映射，从而屏蔽应用层对不同表名的感知，降低业务改造量。

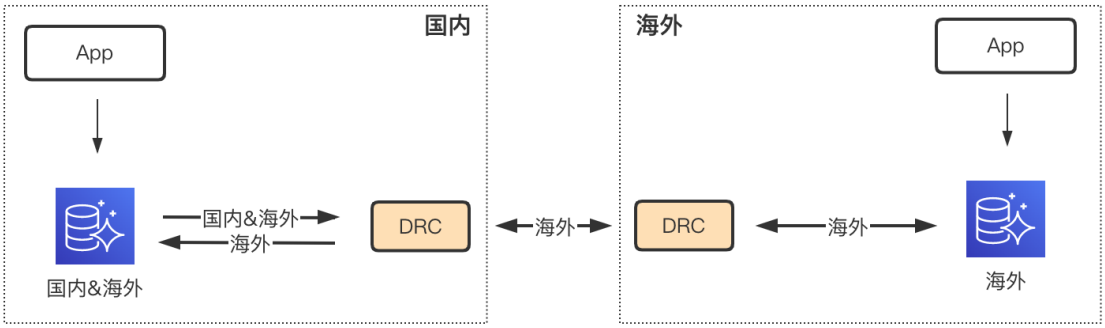


（2）行过滤

库表映射不涉及数据过滤，经过 DRC 的流量都会进行复制，因此映射在 Applier 端处理，直接根据映射规则替换表名即可。为此业务需要进行 2 处改造：

- 人工分离国内机房国内和海外数据；
- 为了使国内母表保存全量数据，海外用户数据经过 DRC 复制回国内时，需要通过触发器自动同步到母表。

为了进一步降低业务改造量，DRC 提供行过滤功能，用户无需进行业务改造，只需保证表中包含 Uid 字段即可，DRC 根据 Uid 自动判断数据归属地，进行数据过滤。



单向复制链路级别添加行过滤配置，其中包括：

- 过滤类型

Uid 过滤，业务层面一般通过 Uid 维度进行拆分，通过 SPI 动态加载 Uid 过滤实现，携程内部由于 Uid 无特殊标记，无法通过 Uid 名称判断出归属地，只能通过 SOA 远程调用实时判断 Uid 归属地获得过滤结果；如果 Uid 有规则可循，则可以通过正则表达式匹配即可

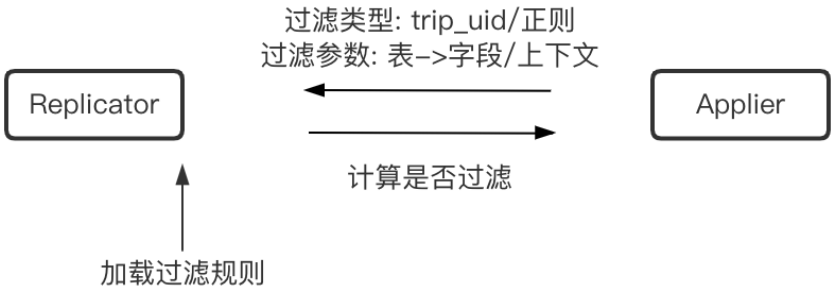
Java 正则表达式，支持针对单字段的 Java 正则表达式简单匹配计算，适合单一维度数值有

规则的业务场景

Aviator 表达式，支持针对多字段的 Aviator 表达式复杂匹配计算，适合多维度数值相关联的业务场景

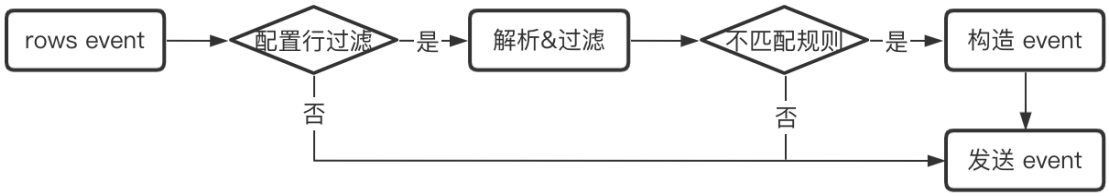
- 过滤参数

包含表到过滤字段的映射关系，以及与过滤类型对应的上下文，比如正则表达式。



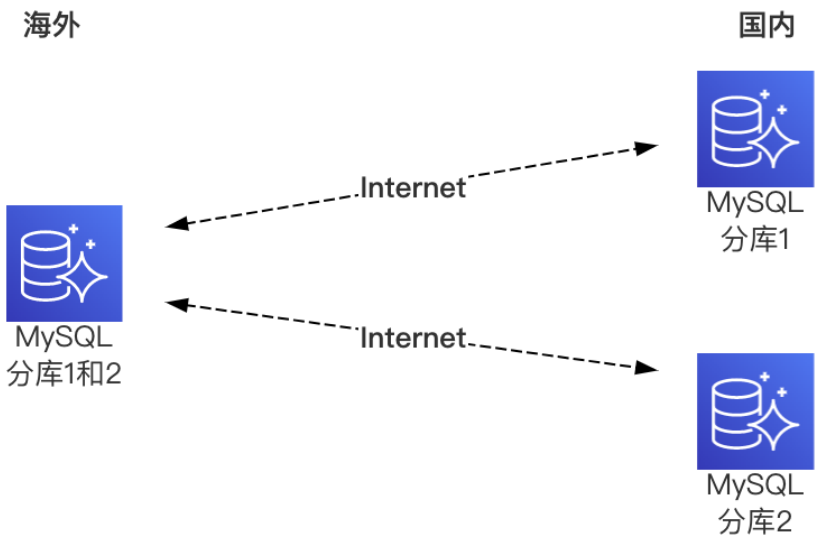
Applier Binlog 请求中携带行过滤配置，Replicator 根据过滤类型加载对应的过滤规则，从而计算出过滤结果。

行过滤在发送端 Replicator 实现，这样实现的好处是跨海发送数据量大大降低，但同时也带来了解析和重构 Rows Event 的复杂性和性能损耗，即先解析 Rows Event 并根据过滤后的行数据生成新的 Rows Event。Rows Event 的解析需要表结构信息，而表结构信息是保存在 Binlog 的头中，势必在 Rows Event 前保证能够获得对应的表结构；解析后就可以将每行过滤字段值应用到过滤规则上，若匹配出需要过滤的行，则需要根据过滤后的行构造新的 Rows Event 并发送，否则直接发送即可。



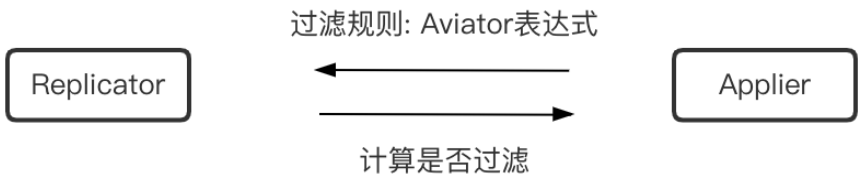
3.2.2 数据库混部

核心业务随着数据量的膨胀，会采用分库来降低数据库压力，在公有云部署时，鉴于云上初始流量不多，并且可动态提升机器配置，DBA 部署时会将所有分库部署在同一个 RDS 集群，此时复制从一对一变成一对多。



（3）表过滤

单向复制链路级别添加库表过滤配置，支持 Aviator 表达式。Replicator 发送前，通过将从 Binlog 中解析的库表名作用于 Aviator 表达式从而得到过滤结果。



3.3 数据库上云流程

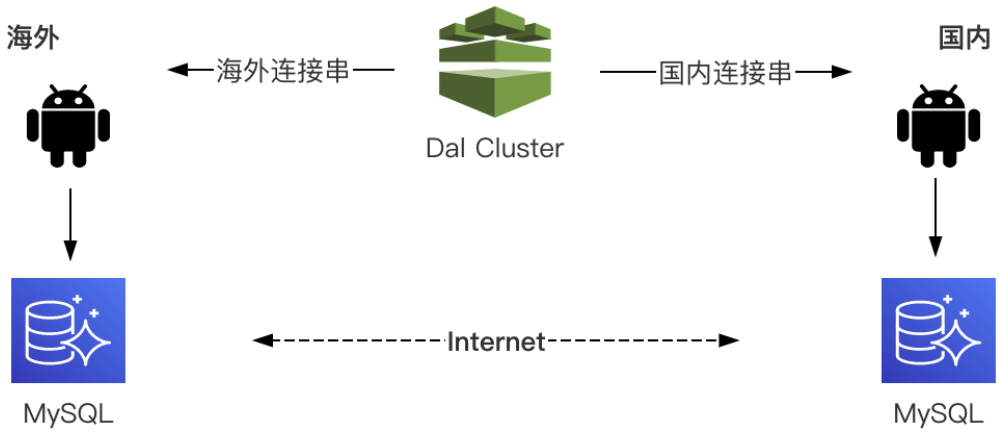
完整的业务上云流程一般分为四步：

- 数据库先上云，搭建国内海外数据库复制，验证海外数据可用性和完整性；
- 在海外数据可用的前提下，应用上云，就近访问海外数据库，验证部署海外应用可行性；
- 流量路由层灰度业务流量，可根据 Uid 白名单、流量百分比在流量接入层进行灰度，验证业务逻辑正确性；
- 灰度完成，国内和海外流量完成切分，验证国内和海外业务隔离性，为此后下线底层数据复制做准备。

数据库上云在每一步都有所涉及，第一步通过 DRC 解决了数据的可用性问题，第二步通过数据库访问中间件解决了数据可达性问题，第三步业务通过流量准确切分保证数据一致性问题，第四步国内海外实现数据隔离后，即可下线 DRC 数据复制。在分析完 DRC 原理后，下面再分析下其他几步数据库相关问题。

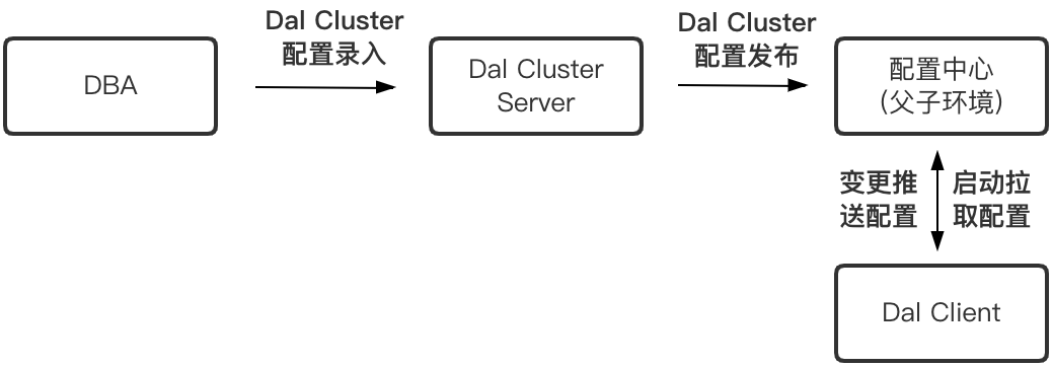
3.3.1 数据访问层

Dal 包含中心化配置管理服务端 Dal Cluster 和 Dal 客户端两部分。上云前同一个数据库物理上只有一个集群，上云后海外增加相同集群，服务端 Dal Cluster 就需要根据客户端环境下发正确的 MySQL 配置文件。



（1）Dal Cluster 原理

Dal Cluster 变更推送功能借由分布式配置中心完成，配置中心提供子环境功能，国内数据库配置默认放在父环境，海外数据库则会在上线流程中生成对应的子环境数据库配置。这样在 Dal Client 启动时，带有不同环境配置的客户端会拉取到不同的配置，从而实现数据库的就近访问，整个过程对业务透明，代码无需改造。



3.3.2 流量切分

业务上云一般采用 Uid 归属地进行流量切分，当流量开始灰度后，两端数据库都开始接收写流量。如果流量灰度不干净，针对同一个 Uid 数据在两端同时被修改，则会导致底层 DRC 数据复制时出现数据冲突。

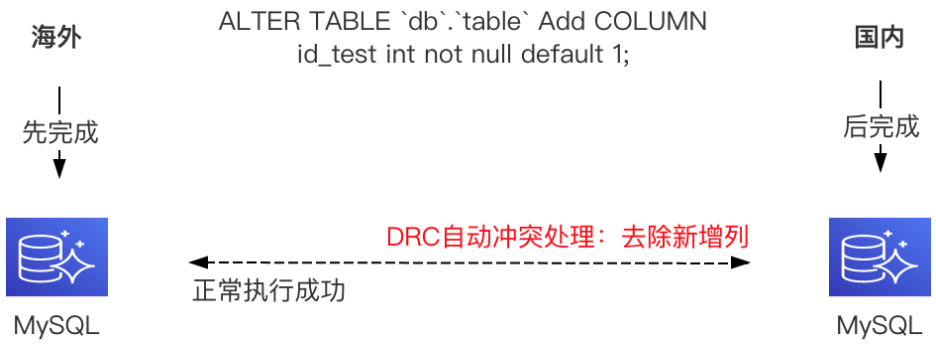
当冲突发生时，Applier 默认根据时间戳进行冲突策处理，接入 DRC 的表都有一个精确到毫秒自动更新的时间戳，时间戳最新的数据会被采用，从而实现数据的一致。

3.3.3 表结构变更

通过 DRC 复制的集群，在表结构变更流程中，会自动关联到公有云集群，在两端同时进行变更操作。

由于变更完成时间有先后，假设一个增加字段的变更海外先完成，在国内完成变更前的时间范围内，针对该表海外到国内的复制将出现复制冲突，默认 DRC 会捕获该异常，并从异常信息中提取出列名，将多出的列从 SQL 中移除后再执行，从而自动处理掉冲突。

当国内集群完成表结构变更后，新增列的值在两端都为默认值，数据仍然一致。



3.4 业务落地成果

海外数据库复制从 2021 年 11 月上线至今，接入公司 90+复制集群;
上海↔新加坡 AWS 复制平均延迟 90ms，上海↔法兰克福 AWS 复制平均延迟 260ms;
账号集群通过库表映射，常旅、收藏等通过行过滤实现用户数据隔离;
通过一对多部署，公有云/国内机房 MySQL 集群比维持在 1/5，DRC 复制成本/MySQL 集群成本维持在 2/5;

四、未来规划

为了支持更多 Binlog 消费方，支持消息投递;
DRC 当前只支持增量数据的实时复制，后续会支持存量数据的复制以及敏感数据的初始化过滤，覆盖业务上云过程中更多数据复制场景;
Replicator 作为有状态实例，使用本地磁盘保存 Binlog，公有云使用的块存储本身即是分布式存储系统，Replicator 可探究存储架构改造，实现主备共用同一份存储，从而降低使用成本。

五、DRC 开源地址：

https://github.com/ctripcorp/drc

每分钟写入 6 亿条数据，携程监控系统 Dashboard 存储升级实践

【作者简介】大伟，携程软件技术专家，关注企业级监控、日志、可观测性领域。

一、 背景概述

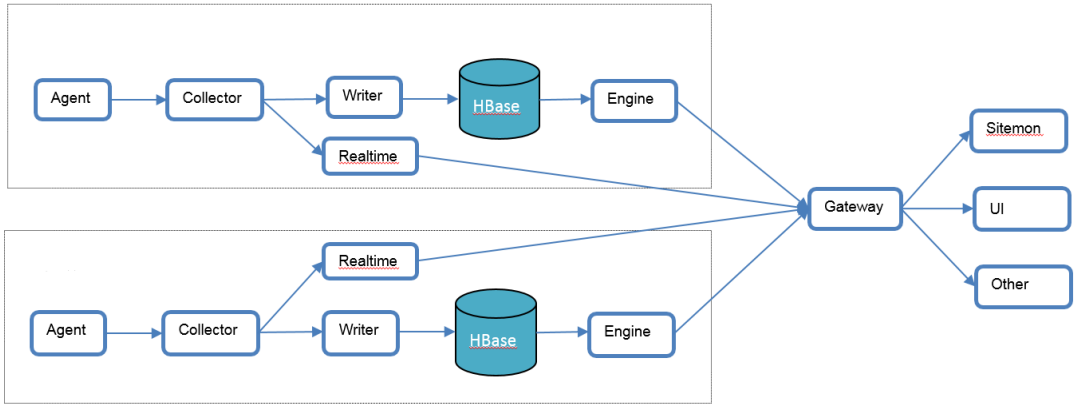
框架 Dashboard 是一款携程内部历史悠久的自研监控产品，其定位是企业级 Metrics 监控场景，主要提供用户自定义 Metrics 接入，并基于此提供实时数据分析和视图展现的面板服务，提供可定制的基于时间序列的各类系统级性能数据和业务指标数据的看板。还可以提供灵活的数据收集接口、分布式的大容量存储和灵活的展现方式。

由于时间较早，那时候业界还没有像样的 TSDB 产品，类似 Prometheus，InfluxDB 都是后起之秀，所以 Dashboard 选型主要使用了 HBase 来存储 Metrics 数据。并且基于 HBase 来实现了 TSDB，解决了一些 HBase 热点问题，同时将部分查询聚合下放到 HBase，目的是优化其查询性能，目前看来总体方案依赖 HBase/HDFS 还是有点重。

近些年，随着携程监控 All-in-One 产品的提出。对于内部的 Metrics 存储统一也提出了新的要求。由于 Dashboard 查询目前存在的诸多问题以及 Metrics 统一的目标，我们决定替换升级 Dashboard 现有的 HBase 存储方案，并且在 Metrics 场景提供统一的查询层 API。

二、 整体架构

Dashboard 产品主要分了 6 个组件，包括 dashboard-engine，dashboard-gateway，dashboard-writer，dashboard-HBase 存储，dashboard-collector，dashboard-agent。目前实时写入数据行数 6 亿条/分钟，架构图如下：



- dashboard-engine 是查询引擎。
- dashboard-gateway 是提供给用户的查询界面。

- dashboard-writer 是数据写入 HBase 的组件。
- dashboard-collector 是基于 Netty 实现的 Metrics 数据收集的服务端。
- dashboard-agent 是用户打点的客户端，支持 sum，avg，max，min 这几种聚合方式。
- dashboard-HBase 是基于 HBase 实现的 Metrics 存储组件。

产品主要特性如下：

- 支持存储精确到分钟级的基于时间序列的数据
- 单个指标数据可支持多个 tag。
- 展现提供任意形式的视图同时可灵活基于 tag 进行分组。

三、 目前的存在问题

基于 HBase 的 Metrics 存储方案虽然具有良好的扩展性，比较高的吞吐，但是随着时间发展，已经不是最优的 TSDB 方案了，可以归纳总结为如下几个痛点。

- 在 TSDB 场景查询慢，整体表现不如专业的 TSDB。
- HBase 热点问题，容易影响数据写入。
- HBase 技术栈运维操作很重。
- 采用自研协议，不支持业界标准的 Prometheus 协议，无法和内部 All-in-one 监控产品较好的融合。

四、 替换难点

- 系统写入数据量大，6 亿条/分钟。
- Dashboard 数据缺乏治理，很多不合理高维的 metrics 数据，日志型数据，经过统计，整体基数达上千亿，这对 TSDB 不友好，这部分需要写入程序做治理。如图 2 所示是 top20 基数统计，有很多 Metric 基数已经上亿。
- Dashboard 系统存在时间久，内部有很多程序调用，替换需要做到对用户透明。

metrics cardinality top20

	min	max	avg	current
_soa.service.request.count_value{db="CIS-SYSTEM",source="SHAOY",stype="soa",topic="clogging.metrics.d...	755 Mil	755 Mil	755 Mil	755 Mil
_hotel.alliance.common.staticinfolib.client.accesscount_value{db="CIS-SYSTEM",source="SHAOY",stype="h...	569 Mil	569 Mil	569 Mil	569 Mil
_ar.auth.illegal.app_value{db="CIS-SYSTEM",source="SHAOY",stype="other",topic="clogging.metrics.default",...	326 Mil	326 Mil	326 Mil	326 Mil
_com.ctrip.flight.dom.engine.coresearch.business.metrics.metricfavresult2_value{db="CIS-SYSTEM",source...	294 Mil	294 Mil	294 Mil	294 Mil
_datacenter.forbiddenqueryairline_value{db="CIS-SYSTEM",source="SHAOY",stype="other",topic="clogging....	192 Mil	192 Mil	192 Mil	192 Mil
_hotel.product.searchservice.metrics.java.v3.roompriceinterface_value{db="CIS-SYSTEM",source="SHAOY",...	188 Mil	188 Mil	188 Mil	188 Mil
_hotel.alliance.common.staticinfolib.client.runtime_value{db="CIS-SYSTEM",source="SHAOY",stype="hotel",t...	186 Mil	186 Mil	186 Mil	186 Mil
_com.ctrip.flight.dom.trade.ndc.repository.metric.airshoppingfromcacheresponsetime_value{db="CIS-SYST...	137 Mil	137 Mil	137 Mil	137 Mil
_hotel.product.searchservice.metrics.java.v3.requestsize_value{db="CIS-SYSTEM",source="SHAOY",stype="...	129 Mil	129 Mil	129 Mil	129 Mil
_trainorderdetailv3info_value{db="CIS-SYSTEM",source="SHAOY",stype="other",topic="clogging.metrics.defa...	99 Mil	99 Mil	99 Mil	99 Mil
_betanew.encryption.simple_value{db="CIS-SYSTEM",source="SHAOY",stype="other",topic="clogging.metric...	94 Mil	94 Mil	94 Mil	94 Mil

五、 替换升级方案

从上面的架构来看，目前我们替换的主要是 dashboard-writer 和 dashboard-HBase 这两个

最核心的组件。为了对用户的平滑迁移，其他组件稍作改动，在 dashboard-engine 组件上对接新的查询 API 即可替换升级成功。对于用户侧，查询的界面 dashboard-gateway 和打点的客户端 dashboard-agent 还是原有的模式不变，因此整个的替换方案对用户透明。具体如下：

5.1 dashboard-HBase 升级为 dashboard-vm

存储从 HBase 方案替换成 VictoriaMetrics+ClickHouse 混合存储方案：

- VictoriaMetrics 是兼容主流 Prometheus 协议的 TSDB，在 TSDB 场景下查询效果好，所以会接入绝大多数 TSDB 数据。

- 基于 ClickHouse 提供元数据服务，主要为界面的 adhoc 查询服务，原来这部分元数据是存储在 HBase 里面，新的方案采用 ClickHouse 来存储。元数据主要存储了 measurement 列表，measurement-tagKey 列表，measurement-tagKey-tagValue 列表这三种结构，目前在 ClickHouse 创建了一张表来存这些元数据。

本地表结构为：

```
CREATE TABLE hickwall.downsample_mtv
(`timestamp` DateTime,
`metricName` String,
`tagKey` String,
`tagValue` String,
`datasourceId` UInt8 DEFAULT 40)
ENGINE = ReplicatedMergeTree('/clickhouse/tables/hickwall_cluster-
{shard}/downsample_mtv', '{replica}')
PARTITION BY toYYYYMMDD(timestamp)
ORDER BY (timestamp, metricName, tagKey)
TTL timestamp + toIntervalDay(7)
SETTINGS index_granularity = 8192
```

分布式表结构为：

```
CREATE TABLE hickwall.downsample_mtv__dt
(`timestamp` DateTime,
 `metricName` String,
 `tagKey` String,
 `tagValue` String,
 `datasourceId` UInt8 DEFAULT 40)
ENGINE = Distributed(hickwall_cluster, hickwall, downsample_mtv, rand())
```

- ClickHouse 存储少量日志型的数据

由于长期缺乏一些治理，Dashboard 还存储了一些日志型数据，这类数据是一些基数很大但

数据量少的数据，不适合存储在 VictoriaMetrics。为了实现所有数据透明迁移，这部分数据经过评估，通过白名单配置的方式接入 ClickHouse 来存储，需要针对每一个接入的日志型指标来创建表和字段。目前的做法是按照 BU 维度来建表，并且针对指标 tag 来创建字段，考虑到接入的日志型指标数量少，所以表的字段数量会相对可控。用机票 FLT 的表结构举例如下图。

5.2 Dashboard-writer 升级为 Dashboard-vmwriter

Dashboard-collector 会分流全量的数据到 Kafka，Dashboard-vmwriter 的工作流程大致是消费 Kafka->数据处理->数据写入存储。Dashboard-vmwriter 主要实现了以下几个核心的功能：

- Metrics 元数据抽取功能，负责抽取出 measurement，tagKey，tagValue 写入 ClickHouse 的 mtv 本地表。这块元数据存储主要依赖了 Redis（用于实时写入）和 ClickHouse（用于查询）。

- 指标预聚合功能，用于加速查询。对接公司内部的配置中心来下发预聚合的配置，配置格式如下。

下面的配置会生成 ClusterName 和 appid 这两个维度组合的 credis 预聚合指标。

```
{
        "metricName": "credis.java.latency",
        "tagNames": [
            "ClusterName",
            "appid"
        ]
```

}

配置下发后，Dashboard-vmwriter 会自动聚合一份预聚合指标存入 VictoriaMetrics，指标命名规则为 hi_agg.{measurement}_{tag1}_{tag2}_{聚合 field}。同样的，查询层 API 会读取同样的预聚合配置来决定查询预聚合的指标还是原始的指标，默认为所有的 measurement 维度都开启了一份预聚合的配置，因为在 TSDB 实现中，查一个 measurement 的数据会扫描所有的 timeseries，查询开销很大，所以这部分直接去查预聚合好的 measurement 比较合理。

- 数据治理：异常数据自动检测及封禁，目前主要涉及以下两方面：

  - HyperLogLog 的算法来统计 measurement 级别的基数，如果 measurement 的基数超级大，比如超过 500 万，那么就会丢弃一些 tag 维度。

  - 基于 Redis 和内存 cache 来统计 measurement-tagKey-tagValue 的基数，如果某个 tagValue 增长过快，那么就丢弃这个 tag 的维度，并且记录下丢弃这种埋点。Redis 主要使用了 set 集合，key 的命名是{measurement}_{tagKey}，成员是[tagValue1，tagValue2，…，tagValueN]，主要是通过 sismember 来判断成员是否存在，sadd 来添加成员，scard 判断 key 的成员数量。

写入程序会先在本地内存 Cache 查找 Key 的成员是否存在，没有的话会去 Redis 查找，对 Redis 的 qps 是可控的，本地 Cache 是基于 LRU 的淘汰策略，本地内存可控。整个过程是在写入的时候实时进行的，也能保证数据的及时性和高性能，写入 Redis 的元数据也会实时增量同步到 ClickHouse 的 mtv 表，这样用户界面也能实时查询到元数据。

数据高性能写入，整个消费的线程模型大概是一个进程一个 kafka 消费线程 n 个数据处理线程 m 个数据写入线程。线程之间通过队列来通信，为了在同一个进程内方便数据做预聚合操作。假设配置了 4 个数据处理线程，那么就会按照 measurement 做 hash，分到 4 个 bucket 里面处理，这样同一个 measurement 的数据会在一个 bucket 里面处理，也方便后续的指标预聚合处理。

```
private int computeMetricNameHash(byte[] metricName) {
int hash = Arrays.hashCode(metricName);
hash = (hash == Integer.MIN_VALUE ? 0 : hash);
return hash;
}
byte[] metricName = metricEvent.getName();
hash = computeMetricNameHash(metricName);
buckets[Math.abs(hash) % bucketCount].add(metricEvent);
```

经过程序埋点测算，正常情况下整体链路的数据写入延迟控制在 1s 内，大约在百毫秒级。

5.3 Metrics 统一查询层

契约上，兼容了 Dashboard 原来的查询协议，也支持标准的 prometheus 协议。

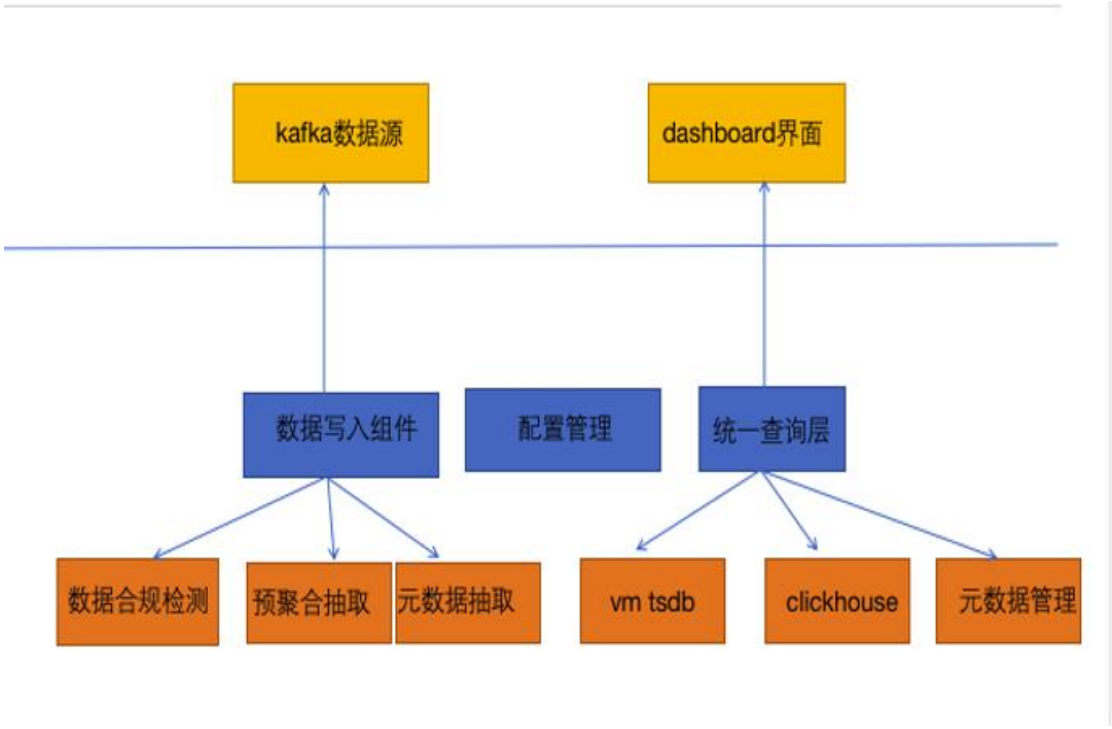
实现上，封装了 VictoriaMetics+ClickHouse 的统一查询，支持元数据管理，预聚合管理，限流，rollup 策略等。

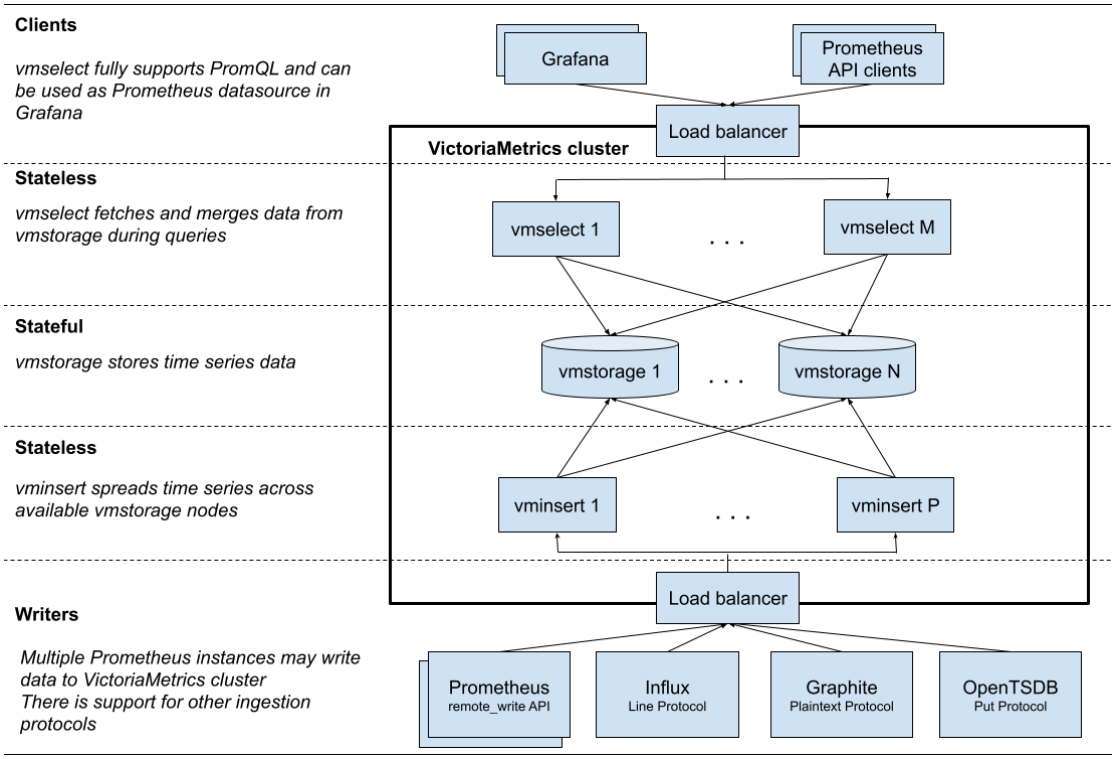
查询层主要提供了以下四个核心接口。

- Data 接口：根据 measurement，tagKey，tagValue 返回时序数据，数据源是 VictoriaMetrics。
- Measurement 接口：返回 limit 数量的 measurement 列表，数据源是 ClickHouse。
- Measurement-tagKey 接口：返回指定 measurement 的 tagKey 列表，数据源是 ClickHouse。
- Measurement-tagKey-tagValue 接口：返回指定 measurement 和 tagkey 的 tagValue 的列表，数据源是 ClickHouse。

如下图第一张所示是新的存储架构，第二张是 VictoriaMetrics 自身的架构。

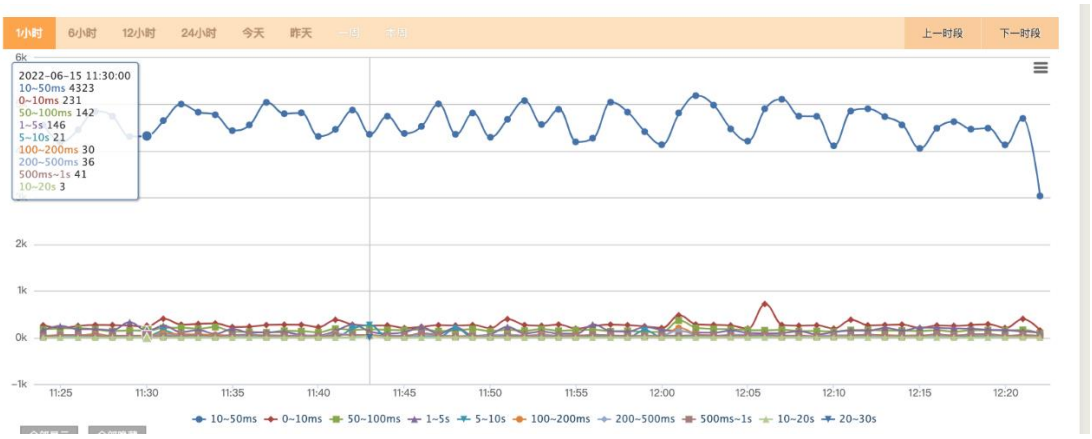
需要注意到，整个数据写入层是单机房写单机房的存储集群，是完全的单元化结构。最上层通过统一的数据查询层汇总多个机房的数据进行聚合输出。在可用性方面，任何单一机房的故障仅会影响单机房的数据。





六、 替换前后效果对比

- 替换后的查询耗时从 MAX, AVG, STD 提升近 4 倍。查询耗时大多落在 10-50ms 之间。相比之前 HBase 经常查询超时，整体查询的稳定些也好了很多，见图 6，7。
- 写入稳定性提升，彻底解决了因为 HBase 热点引发的数据积压。
- 替换后支持了更多的优秀的特性，可以基于 promQL 实现指标的逻辑计算，同比环比，模糊匹配等。



七、 未来规划

（1）统一查询层接入所有 Metrics 数据，除了 Dashboard，目前内部还有 HickWall，Cat 有大量 Metrics 数据没有接入统一查询层，目前采用的是直连 openrestry+VictoriaMetrics 的方

式，openrestry 上面做了一些简单的查询逻辑，这块计划后续接入统一查询层，这样内部可以提供统一的元信息管理，预聚合策略等，达到 Metrics 架构统一。

（2）提供统一写入层，总体 Metrics 目前是近亿级/秒，这块写入目前主要是基于 Kafka 消费进存储的方式，内部这块写入是有多个应用在处理，如果有统一的写入层那么就能做到写入逻辑统一，和查询层的查询策略也能做到联动，减少重复建设。

（3）Metrics 的存储统一层提供了较好的典范，内部的日志存储层统一也在如火如荼的进行中，也会往这样的一个方向发展。

Islands Architecture（孤岛架构）在携程新版首页的实践

【作者简介】携程前端框架团队，为携程集团各业务线在 PC、H5、小程序等各阶段提供优秀的 Web 解决方案。当前主要专注方向包括：新一代研发模式探索，Rust 构建工具链路升级、Serverless 应用框架开发、在线文档系统开发、低代码平台搭建、适老化与无障碍探索等。

一、项目背景

2022，携程 PC 版首页终于迎来了首次改版，完成了用户体验与技术栈的全面升级。

作为与用户连接的重要入口，旧版 PC 首页已经陪伴携程走过了 22 年，承担着重要使命的同时，也遇到了很多问题：

（1）维护/更新困难

祖传代码黑盒逻辑过多，产品也难以推动新需求的上线，旧版首页已经不能满足高速发展的业务需求。

（2）技术栈陈旧且不统一

互联网技术日新月异，旧版首页的整体架构设计和技术栈都相对落后，且大首页中各个组件的研发涉及多事业部合作，存在技术选型差异的问题，增加了维护成本。

（3）用户体验有待改善

旧版携程首页的设计风格沿用至今，在视觉和交互层面上，都已经难以满足用户不断提升的互联网体验和审美需求。

综合上述情况，为了给用户提供更好的服务，携程首页的整体改造迫在眉睫。

二、需求分析

携程首页改造需要考虑的核心问题包括以下几个方面：

（1）技术选型

为了优化首屏性能，提升用户体验，携程新版首页采用服务端渲染模式。在技术选型上，考虑到我们希望应用层是轻量的，只做页面 HTML 拼接和响应两件事情，最终决定基于 Node.js 构建应用载体，客户端则统一使用公司主流的 React 技术栈。

（2）跨团队合作

首页作为携程的重要门户，涉及多业务线的流量入口。如图 1 所示，我们可以将整个页面进行切割，按业务线划分成多个组件模块。

图 1 携程首页业务模块切分图

可以看到，整个页面的研发是需要框架部门和各个事业部业务团队紧密合作才能完成的，这就需要一整套完善的跨团队合作模式。其中，我们希望业务团队只需要关注业务逻辑的实现，完成组件模块的开发。框架团队则负责提供：

- 组件模块服务端/客户端构建方案
- 组件模块服务端渲染方案
- 应用层面，实现页面组装及响应
- 组件模块开发环境
- 监控及维护

上线后，我们需要时刻关注应用状态，及时响应异常情况。因此，需要对应用及组件进行埋点监控。除此之外，由于需要跨团队合作，对于业务组件，我们希望各个业务团队不仅可以实现开发/构建自由，彼此独立互不影响，在监控及版本管理上也能实现自控。因此，我们将各个业务组件包装成 Node.js 应用，开发人员可以直接在发布系统查看组件版本，完成发布/回退，也可以通过应用 ID 在埋点管理平台查看组件的相关埋点。

三、整体架构设计

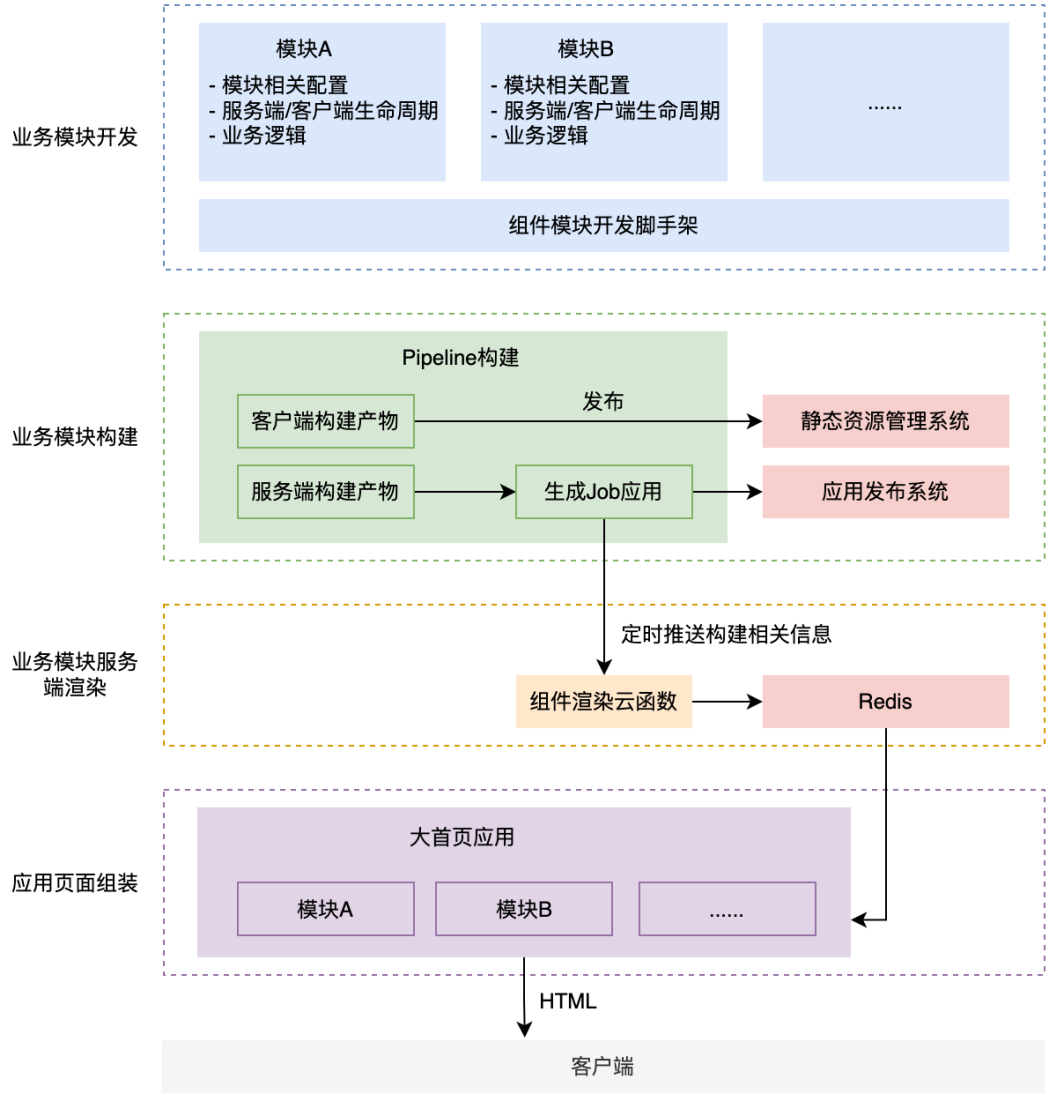


图 2 携程首页架构设计图

基于上述需求分析，携程新版首页的整体架构设计如图 2 所示，可以分为四个部分：

（1）业务模块开发

我们将携程首页拆分为多个业务模块，由各业务团队负责完成相应组件的开发。与常规 React 组件开发不同的是，首先，开发人员需要在配置文件中设置好模块相关配置，如组件唯一 ID；其次，组件开发需遵循一些规则，如为防止出现样式污染，我们强制使用 CSS Modules；最后，我们支持服务端渲染组件，可以在服务端生命周期中拉取数据，然后在服务端/客户端使用。为了更好的辅助业务团队完成组件开发，框架团队会提供脚手架帮助创建组件模版，搭建开发环境，模拟完整首页场景。

（2）业务模块构建

业务模块开发完成后，就需要构建/发布至生产环境。整个构建过程会在 Pipeline 中完成，开发人员 git push 代码后会自动触发。基于不同的 entry 及配置，我们会使用 webpack 分别

完成客户端及服务端代码的生产态构建，并将客户端构建产物（js+css）上传至静态资源管理系统。

之后，我们会将服务端构建产物（js）连同组件及静态资源版本相关信息包装成一个 Job 应用，该应用中会有一个定时任务负责推送当前版本信息，触发组件完成服务端渲染，这里我们是使用定时器来实现定时任务的管理。最后，需要由开发人员在发布系统中将构建好的应用镜像部署到生产环境，完成组件的发布。

（3）业务模块服务端渲染

业务模块的服务端渲染主要包括两部分：

在沙盒中完成服务端渲染
将组件相关信息及渲染生成的 html 存到 Redis 中

我们将相关功能实现封装成云函数，作为服务提供出去。由于部分组件对服务端渲染具有数据更新的需求。因此，上文我们提到过，Job 应用中会有一个定时任务，负责触发组件进行服务端渲染，这里也就是会触发云函数的调用。

（4）应用页面组装

最后，我们就需要在应用中将所有的业务模块拼装起来，定时从 Redis 中获取组件相关信息，组装成首页 html 返回到客户端。

四、整体架构的核心功能实现

对应上述首页架构设计，我们简要介绍下部分核心功能的实现：

4.1 搭建组件开发环境，模拟首页场景

我们会在开发阶段提供脚手架辅助业务团队开发组件，其中一项重要功能就是搭建组件开发环境。常规的 webpack 搭建 React 开发环境我们这里就不赘述了，为了实现开发环境的统一标准化，我们还做了以下事情：

- 将 webpack，babel 的相关配置封装到 cli 中，有选择的提供可配置项，规范化组件开发环境；
- 对 entry 进行收口。这里需要注意的是，服务端和客户端的 entry 是不同的，对于客户端 entry，需要获取服务端传过来的数据，并通过调用 ReactDOM.render()完成渲染。

```
import React from 'react'
import ReactDOM from 'react-dom'
import Comp from '__COMP_PATH__'

const render = async() => {
```

```
    let data

    // 获取服务端传递到客户端数据
    const container = document.getElementById('__MFE___MODULE___DATA__')
    if (container && container.textContent) {
        try {
            data = JSON.parse(container.textContent)
        } catch(e) {
            console.log(e)
        }
    }

    const root = document.getElementById('__MODULE__')

    // 客户端渲染组件
    if (module.hot) {
        ReactDOM.render(<Comp serverData={data} />, root)
    } else {
        ReactDOM.hydrate(<ErrorBoundary><Comp serverData={data}
/></ErrorBoundary>, root)
    }
}

render()
```

- 对于服务端 entry，则需要调用服务端生命周期拉取数据，并调用 renderToString()完成渲染：

```
import React from 'react'
import { renderToString } from 'react-dom/server'
import Comp from '__COMP_PATH__'

const render = async() => {
    let data
    // 执行服务端生命周期
    if (Comp.getInitialProps) {
        data = await Comp.getInitialProps(_ctx)
    }

    // 沙盒中传入 setMfeData 方法，见下文中服务端渲染组件实现
    setMfeData(data)

    // 服务端渲染组件，返回 html
    return renderToString(<Comp serverData={data} />)
```

```
}

export default render()
```

搭建首页场景。我们希望开发人员在组件开发时，就可以看到其嵌入在整个首页中的效果，而不是只能看到自己的组件。因此，我们在服务端处理页面请求时，通过以下方式搭建了首页场景：

- 读取首页 html 文件（首次从线上拉取）；
- 解析/处理首页 html，移除当前组件相关的线上 script/link 标签，添加开发态构建产物；
- 在沙盒中服务端渲染组件，替换首页 html 中的组件 html。

4.2 SSR-Service 服务端渲染组件

我们会在沙盒中运行服务端构建生成的代码（可结合上文中服务端 entry 看），完成组件渲染，得到服务端生命周期中返回的数据及组件 html。

```
const vm = require('vm')

const render = async ({content, request}) => {
    // content 即为服务端构建生成的代码
    const script = new vm.Script(content)

    let moduleObj = {
        exports: {}
    }
    let mfeEnv = 'prod'
    let mfeData

    // 基于云函数中的 request 模拟 req
    const _req = {
        url: request.rawPath,
        query: request.queryStringParameters,
        headers: request.headers
    }

    let sandBox = {
        ...global,
        process,
        require,
        module: moduleObj,
        console,
        _ctx: {
            req: _req,
```

```
            env: mfeEnv,
        },
        setMfeData: (data) => {
            mfeData = data
        }
    }

    // 沙盒中运行，执行服务端渲染
    const ctx = vm.createContext(sandBox)
    script.runInContext(ctx)
    const comp = await sandBox.module.exports.default

    return {
        comp,
        mfeData
    }
}
```

4.3 整体页面组装

在首页应用中，我们会定时从 redis 中获取组件相关信息，拼装首页 html，在有客户端请求进入时，直接返回缓存中的最新 html。

```
let indexCache = ''

const renderPage = async (content) => {
    // 加载首页 html
    const $ = cheerio.load(content)

    // 更新组件
    for (let module of modules) {
        try {
            // moduleData 为从 redis 取到的数据
            let data = moduleData[module] || ''
            if (!data) {
                continue
            }

            data = typeof data == 'string' ? JSON.parse(data) : data
            const {comp, version, mfeData, style} = data

            // 更新组件相关的 html，link，script 标签
            parse(module, comp, $, version, mfeData, style)
        } catch(e) {
```

```
                console.log(e)
            }
        }

        // 生成 html
        const payload = $.html()
        if (!payload) {
            throw Error('renderPage error - html is null')
        }

        // 更新缓存
        indexCache = payload
}
```

五、公共组件的渲染原理及技术细节

前面说的是岛屿式架构之首页的整体架构和独立组件渲染的核心实现，其中有些独立组件（左侧菜单栏，头部等）除了在大首页中使用，还会在其他的页面中使用，这里就称为公共组件。

5.1 公共组件需求点和痛点分析

在开始开发公共组件前，需要整理一下目前各个事业部的接入需求、成本及痛点。所以总结了以下问题点：

（1）各个事业部的站点技术架构不同

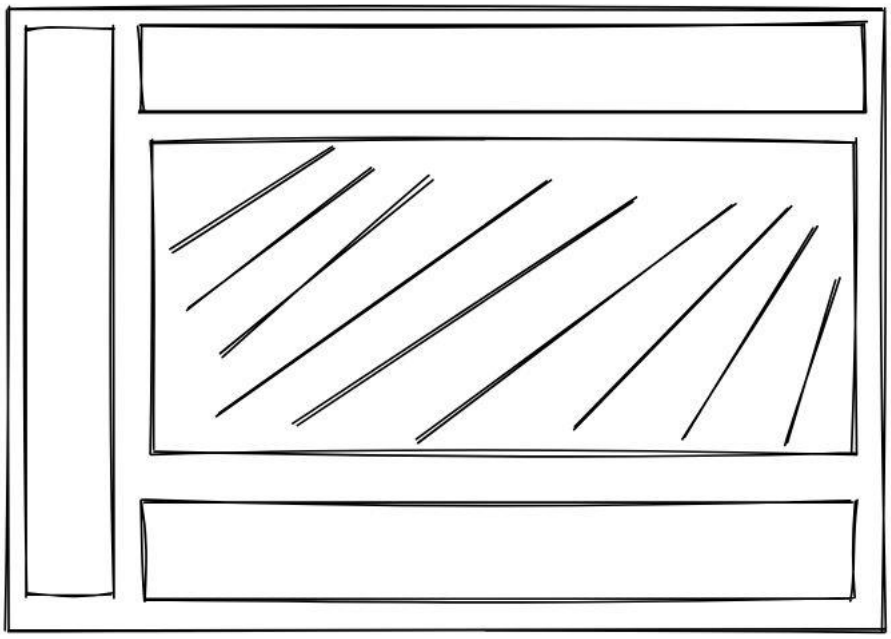
由于各个事业部的站点技术架构不同。有的事业部可能是服务端渲染，有的可能为客户端渲染 。在服务端渲染中，技术栈又可能出现 JAVA 和 NODE 。而在客户端渲染中，各个事业部技术栈也不统一，有 React、JQuery 或者 Vue 等等前端框架。这里的问题是各个事业部的技术栈的错综复杂，如果分开维护会带来不同的版本及很高的维护成本。

（2）所有页面中的公共组件有变更时能否统一热更新

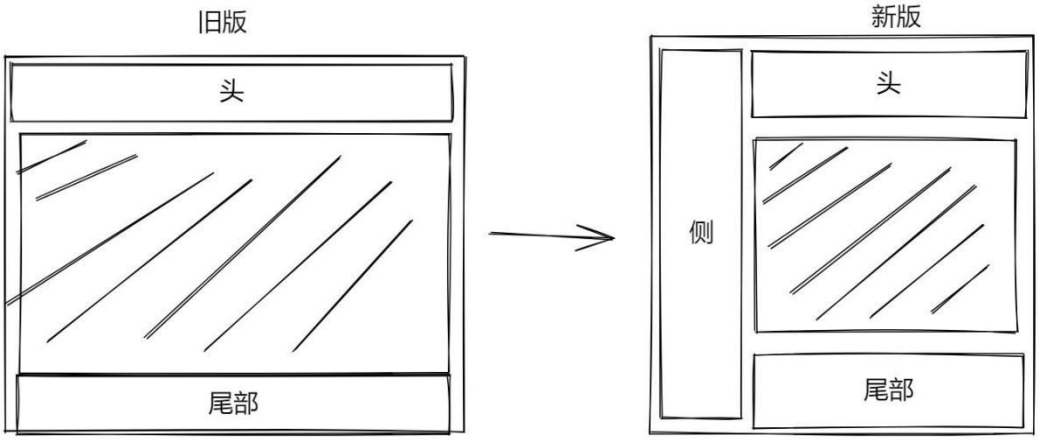
当公共组件的改动或有问题需要修复时，不能让所有的页面都去变更公共组件，而是应该我们变更后，所有页面上的公共组件会静默生效，各个事业部无需关心公共组件的变更。

（3）公共组件的样式如何不对页面造成巨大影响

由于各个业务方的样式风格不同并且还存在一些全局的公共样式，如何才能保证每个接入方为下图的页面布局方式，其页面组成的方式为阴影部分是事业部所维护的组件，其他是公共组件。



由于历史原因，旧版的公共组件已经使用了很多年了，新版头尾和旧版的头尾布局构造不同，要如何设计，才能使其改动最小，而不是去做很大的改动去适配公共组件。新旧版大首页页面布局变化如下图：



（4）公共组件的渲染性能问题

在背景中提到的不同形态的公共组件（比如有些不需要左侧菜单或者头部样式的不同），如何在客户端能第一时间展示给用户相应组件形态并且支持搜索引擎优化(SEO)。当多个公共组件在页面中如何能快速进行加载及渲染。

5.2 解决公共组件问题和痛点

（1）问题一：各个事业部的站点技术架构不同

前面提到了各个业务支线的技术栈不统一的问题，并且还存在服务端和客户端渲染的情况，如果为了多个技术栈去维护多个公共组件维护成本极高，且没有办法做到一套代码多端使用。这里就从服务端和客户端渲染分析，提供的相应解决的方案

- CSR(客户端渲染)

在 CSR 中，技术栈也不同。由于有 React、Vue、jQuery，所以我们需要提供的应该是一个原生 JS 的公共组件，这样能保证维护成本。但是大首页的首屏技术栈已经为 React，再去开发及维护一套原生 JS 组件显得冗余。所以需要一个方案来支持多技术栈运行，并且能够兼容我们大首页首屏的技术栈。

最终的方案是使用 Preact，它很轻量，重点是它可以帮助我们解决多技术栈运行并且能够兼容 React。可万一有页面同样在使用 Preact 和我们冲突怎么办？这里将 Preact 单独打包出来 common 包并且重名了全局的变量。这样即使页面使用了 Preact 也不会和我们有冲突，在 webpack 的 externals 的选项中可以配置组件需要的包名。

```
{
    //...
    externals: {
        preact: 'xxxxxx'
    }
    // ...
}
```

- SSR(服务端渲染)

在 SSR 中，在技术栈上选择了 Preact，Preact 它同样支持 SSR，可以构建一个服务端的 JS 来支持 SSR。因此我们的问题就迎刃而解了，我们在组件构建的时候多生成一份 Preact 的 SSR 的 JS，用沙盒执行服务端渲染输出 HTML 并存储。我们调研了以前的老的公共组件，全部携程的业务线存在的技术栈只有两种：JAVA、NODE，这样就提供两个接入方式的外壳即可——两套不同语言的 SDK 及接入方式，其内部都是获取统一的公共组件 HTML 字符串供页面使用。

```
{
        // ...
        resolve: {
            extensions: ['.ts', '.tsx', '.js'],
            alias: isPreact ? {
                "react": "preact/compat",
                "react-dom": "preact/compat",        // Must be below test-utils
                "react/jsx-runtime": "preact/jsx-runtime"
            } : {},
        }
        // ...
```

}
（React 轻松转换 Preact）

（2）问题二：所有页面中的公共组件有变更时能否统一热更新

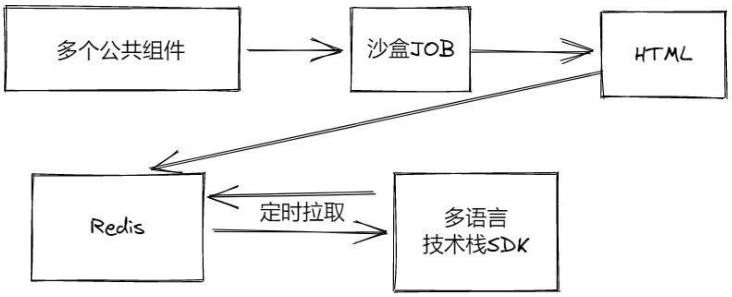
当公共组件更新或者修复紧急的某些问题，不应该影响业务方页面，应该是自动进行更新，当用户访问页面时，看到就是最新的公共组件，因此我们没有做类似 npm 包多版本的方式进行管理。

基于问题一的基础上：

- SSR(服务端渲染)的情况

SSR 的服务端的 HTML 从哪里来？HTML 怎么样才能是最新的？

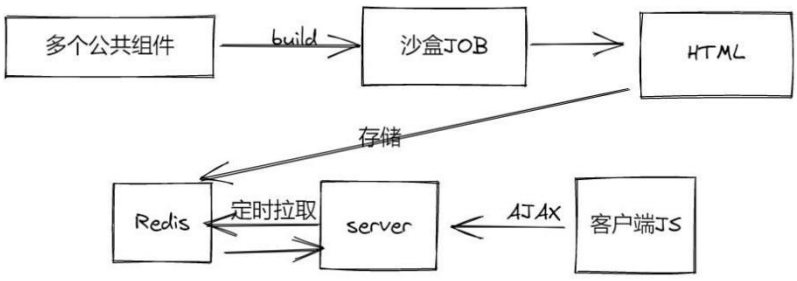
我们需要构建出来一份服务端的 JS 在沙盒中输出 HTML，存储在了 Redis 中，将多个公共组件统一构建出了多个 HTML，分别存放在 Redis 里。业务方接入 JAVA、NODE 的 SDK 其实要做的只有一件事：守护进程定时的去 Redis 里拿到最新的 HTML 结果。



- CSR(客户端渲染)的情况

CSR 如何保持为最新公共组件的？

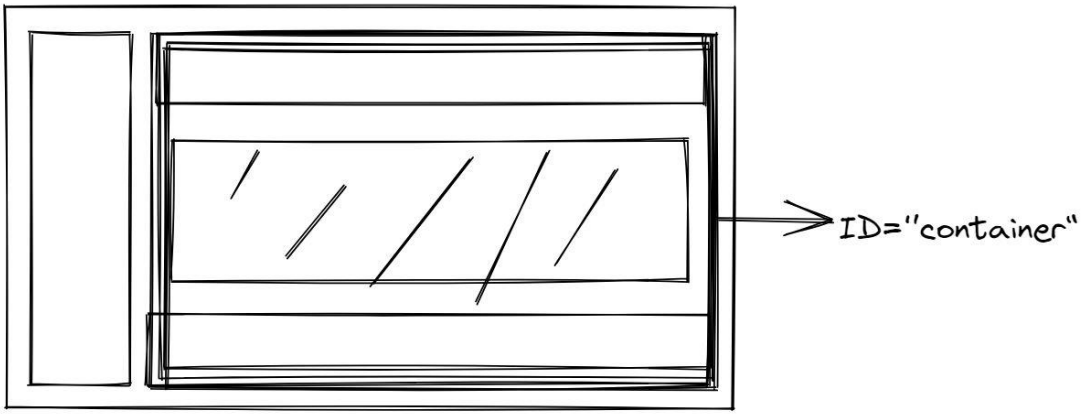
需要一台机器同多语言技术栈 SDK 一样，定时从 Redis 里读取数据，对外暴露一个接口，供客户端的 JS 调用。这样，每次用户访问页面的时候，客户端 JS 会发起请求，保证用户所看到的的内容永远是最新的。



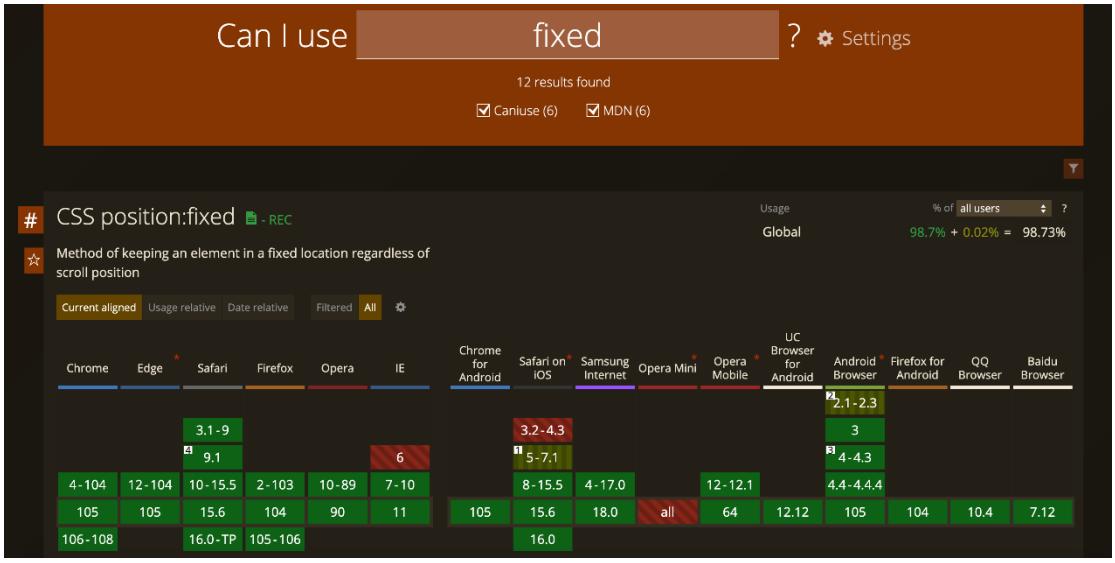
（3）问题三：样式问题

目前新版的相比之前旧版的公共组件在样式和交互上更加复杂。由于左侧菜单的存在，使得布局构造不同，而且各个事业部的页面样式可能五花八门，很难保证不会影响自身样式和事件等问题。

比如：如果使用 flex 的布局，需要在最外层套用一个 div，如果不套用的话则需要在 body 元素上添加 flex 样式，但是不能保证其他的事业部的页面的 body 是否有其他的样式，甚至 body 内是否存在其他的 div 元素等。还有很多事业部的页面的类似滚动等事件监听都是在 body 上进行监听的，所以如果外层套取 div，这种形式会让原来页面的事件监听滚动非常麻烦，各个事业部原来监听 body 的事件，需要一一进行改动。

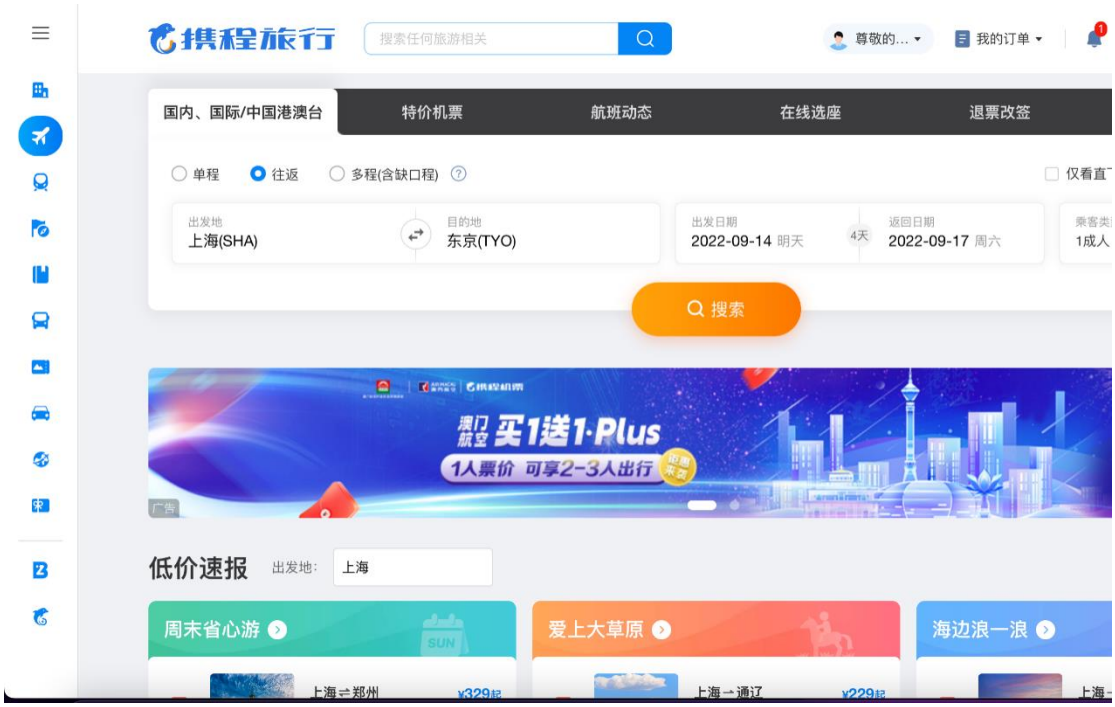


观察老项目发现，之前的公共组件骨架有个最外层的 div 元素，并且有一个名为"container"的 id，我们要做的就是将左侧的菜单 fixed 在左侧就好了.关于 css 的 fixed 的兼容性：



但是此时有个问题是，我们的左菜单是可以展开或收起的。所以在展开和收起的时候需要一个全局的通信机制，当左侧的组件变化时，在组件的内部应该触发全局的通信钩子，通知 id

为 container 的 div 元素跟随左菜单变化，达到 flex 布局的效果。



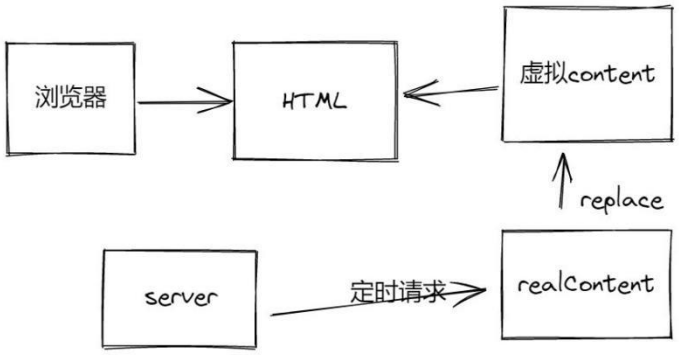
（4）问题四：性能问题

基于问题 1/2/3 大概已经拟定了技术的方向，并且已经能在各个事业部行的通了，证明思路是没有问题的，但是还有些个琐碎的问题需要考虑：

- 因为是定时从服务端里进行拉取，那么第一次没有拉取时或者在客户端渲染的情况下请

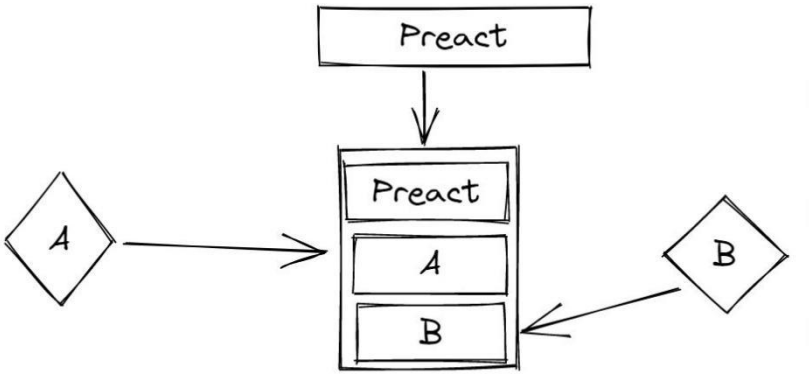
求 server 是需要时间的，这样请求回来 HTML 再进行异步渲染，是否时间过长？

为了解决上面的问题，我们考虑了先准备一个预渲染的 HTML 占位，类似骨架屏的意思，此时就可以先进行骨架屏的渲染，之后再异步拉取渲染，来解决异步渲染白屏等待时间的问题。



- 多个公共组件的客户端 JS 资源是否能够合并，将 Preact 公共包也一起合并打包。

为了解决这个问题，我们的那台跑沙盒 JOB 机器就可以继续做这件事情。因为每个组件构建后有资源的版本，我们需要将版本存储一份，一旦新的组件构建后，拉取其他公共组件的资源版本，将多个 JS 组装在一起。同时因为我们用了 Preact ，抽取了 Preact 为一个共同依赖，将它放在最前面，保证它的最先执行。



六、公共组件的数据动态配置系统

介绍完携程新版大首页的公共组件的渲染原理及技术细节，接下来就是公共组件中的数据如何支持动态配置。

6.1 为什么需要组件数据动态配置系统

携程 PC 版首页进行改版的过程中，按业务线将整个页面划分成了多个组件模块，每个组件模块内都有需要展示的业务数据。而页面上线之后，随着产品需求的变更，业务数据会被频繁的更新，如果每次更新数据都发布一次模块代码的话，这个成本和风险很大。

因此，将代码和数据分开发布是很有必要的，当组件数据有改动时无需发布组件，搭建一个专门用于发布大首页数据配置的管理系统势在必行。

6.2 组件数据动态配置系统的需求分析

携程大首页数据配置管理系统的核心功能是完成数据配置的发布，并保证发布的可靠性和安全性，为了实现这个目标，此管理系统应制定一套完整的数据检验规范和发布流程，其中主要功能包括：

- 规范数据配置上传格式，本地配置数据与线上配置数据差异对比；
- 制定不同组件模块的数据校验规则，并以此校验数据合法性；
- 数据配置发布前效果预览，确保与线上其他组件模块之间相互不影响；
- 更新线上页面。

6.3 组件数据动态配置系统架构设计

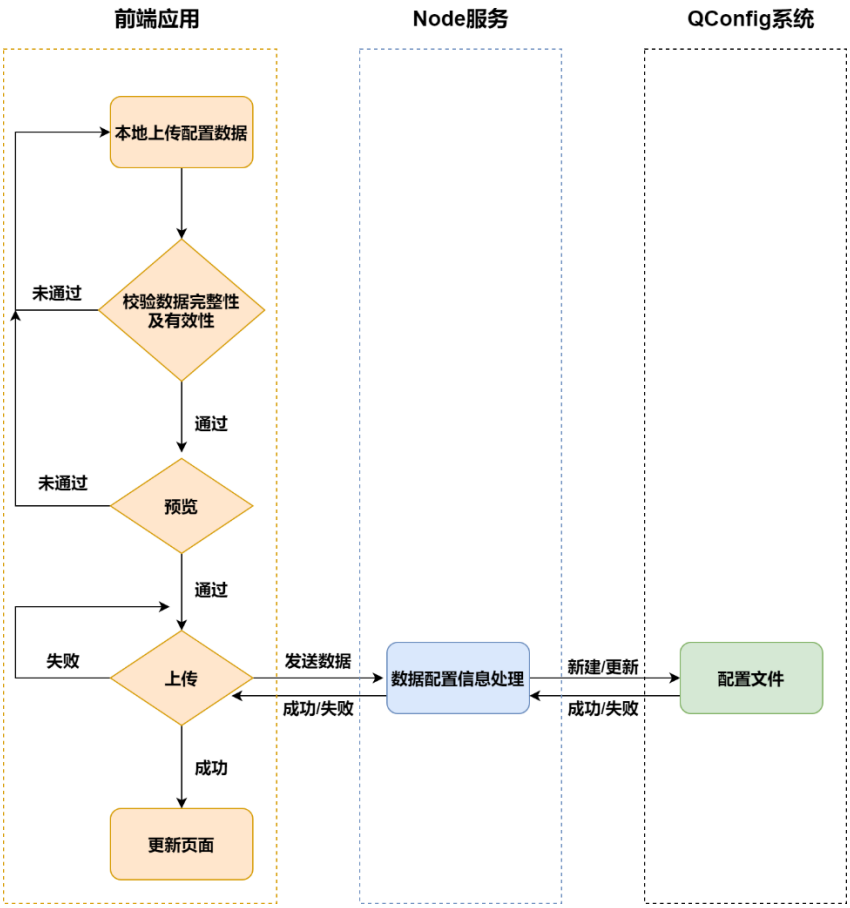


图 1 大首页数据配置管理系统架构设计

数据配置管理系统的架构设计 （如图 1 所示），为了实现需求分析中的四块主要功能，整个管理系统主要搭建了两个应用：

- 前端应用：以可视化界面的形式提供本地上传配置文件，预览数据效果以及更新页面等功能，同时完成了数据校验和预览检测。
- Node 服务：主要负责数据配置的处理及发布，将前端应用上传的数据配置保存到 QConfig 系统中。

其中，前端应用提供的预览功能的架构设计如下图 2 所示：

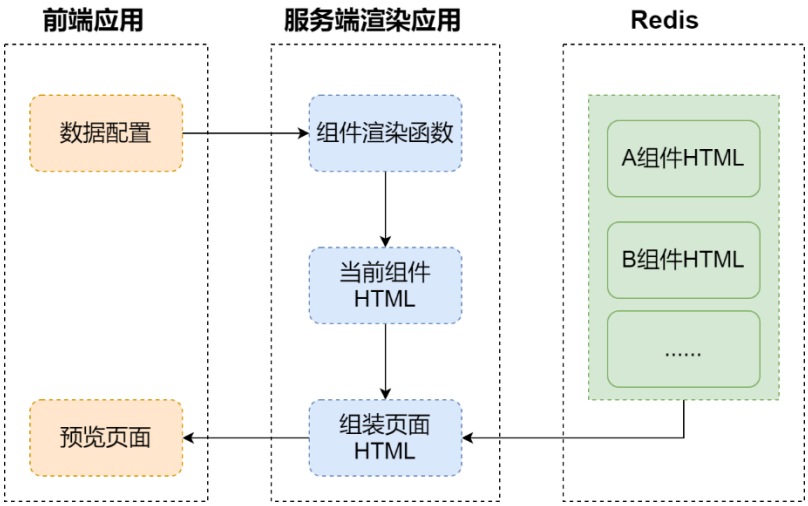


图 2 预览功能架构设计

预览功能的实现主要依赖三部分 （如图 2 所示）：

- 前端应用：负责提供数据配置和展示页面效果。
- 服务端渲染应用：调用组件渲染函数，根据数据配置渲染出当前组件 HTML，并从 Redis 拉取其他组件的 HTML，而后组装成一个完整页面的 HTML 吐给前端应用。
- Redis：存储所有组件模块的 HTML。

6.4 数据配置管理系统的核心功能实现

前面部分介绍了数据配置管理系统的架构设计，这里就架构中核心功能部分的实现进行详细介绍，主要包括：

- 数据配置规范及校验
- 组件及页面预览

（1）数据配置规范及数据校验

本地上传的数据配置最终要传给组件渲染出来，而数据配置的上传者不一定是组件的开发者，上传者并不一定清楚组件所需数据的类型和结构，那么如何保证上传的数据与组件要求

的数据结构保持一致呢？

这就需要管理系统制定一套数据配置规范来约束上传的数据，然而不同的组件，其数据结构是不同的，那么每个组件都应有一套自己的规范。管理系统提供了两种制定数据规范的方式：

录入组件的基本信息，其中包括详细的数据结构：数据名称，数据类型，必传或可选等。
组件使用 TypeScript（推荐的组件开发语言）语言开发时，可以上传.d.ts 声明文件，系统会根据此文件解析出具体的组件信息及数据结构。

规范制定完成之后管理系统会将其存储起来，每次有上传者上传某一组件的数据配置后（为方便上传者修改数据，管理系统规定数据配置以 JSON 文件的形式提供），系统会根据组件的数据规范校验上传的数据配置，如果校验通过则会展示上传数据与线上数据的差别，上传者可进行预览操作；如果校验未通过，则提示未通过原因及具体的不规范数据，上传者不可进行后续的预览操作，需重新上传数据配置，直到校验通过。

（2）组件及页面预览

此部分功能的核心实现在 SSR Service 服务端渲染组件中（上文中有详细介绍，这里不赘述），主要分为以下几个步骤完成：

应用的组件渲染函数在接收到符合组件数据规范的配置数据后，将数据通过 Props 传给组件，进而 render 出当前组件的 HTML。
从 Redis 中取出其他模块的 HTML 与当前组件 HTML 拼接在一起，为了保证预览的可靠性（减少其他模块出错时对当前组件的影响），其他模块均使用生产态的 HTML 进行拼接。为什么一定要将其他模块的 HTML 拼接在一起预览呢？为了测试配置数据发布之后对其他组件模块的影响，若有影响则不能发布，从而保证线上页面的安全性。

七、总结

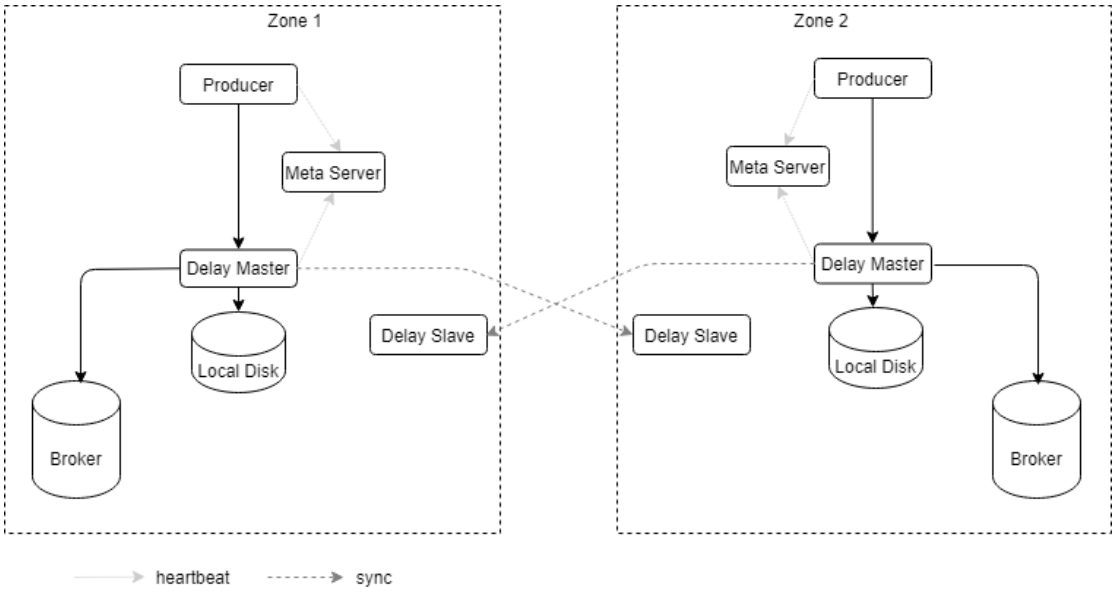
本文通过携程新版首页项目系统的介绍了其整体架构设计，组件开发，数据配置的整个流程及实现原理，是对岛屿式架构的一次实践。希望能对大家今后跨团队组件式开发的项目有所收获。

携程基于 BookKeeper 的延迟消息架构落地实践

【作者简介】本文作者 magiccao、littleorca，来自携程消息队列团队。目前主要从事消息中间件的开发与弹性架构演进工作，同时对网络/性能优化、应用监控与云原生等领域保持关注。

一、背景

QMQ 延迟消息是以服务形式独立存在的一套不局限于消息厂商实现的解决方案，其架构如下图所示。



QMQ 延迟消息服务架构

延迟消息从生产者投递至延迟服务后，堆积在服务器本地磁盘中。当延迟消息调度时间过期后，延迟服务转发至实时 Broker 供消费方消费。延迟服务采用主从架构，其中，Zone 表示一个可用区（一般可以理解成一个 IDC），为了保证单可用区故障后，历史投递的待调度消息正常调度，master 和 slave 会跨可用区部署。

1.1 痛点

此架构主要存在如下几点问题：

- 服务具有状态，无法弹性扩缩容；
- 主节点故障后，需要主从切换（自动或手动）；
- 缺少一致性协调器保障数据的一致性。

如果将消息的业务层和存储层分离出来，各自演进协同发展，各自专注在擅长的领域。这样，消息业务层可以做到无状态化，轻松完成容器化改造，具备弹性扩缩容能力；存储层引入分

布式文件存储服务，由存储服务来保证高可用与数据一致性。

1.2 分布式文件存储选型

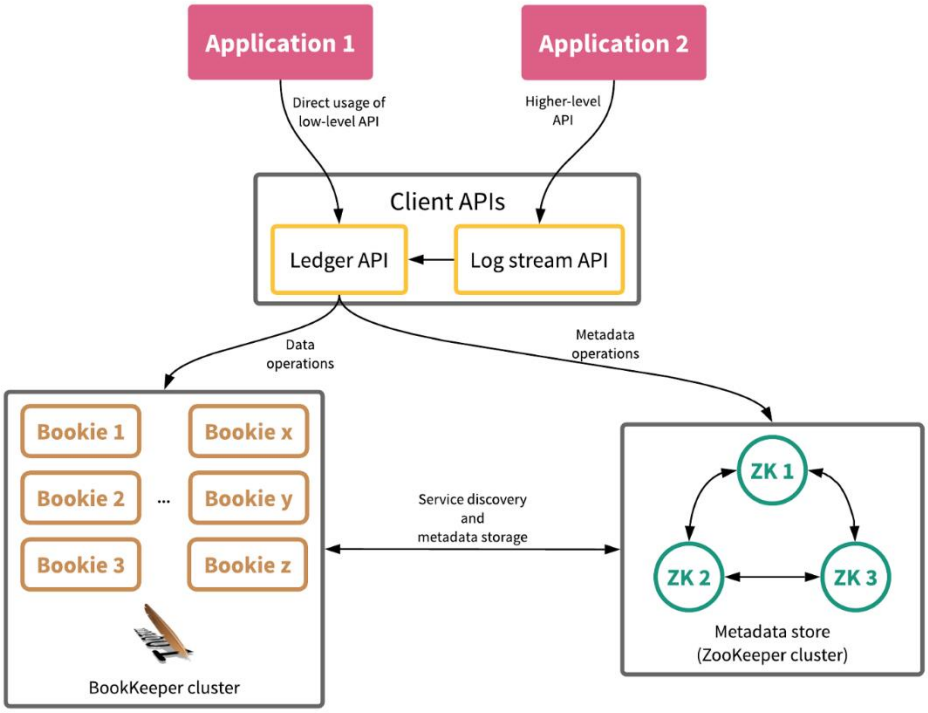
对于存储服务的选型，除了基本的高可用于数据一致性特点外，还有至关重要的一点：高容错与低运维成本特性。分布式系统最大的特点自然是对部分节点故障的容忍能力，毕竟任何硬件或软件故障是不可百分百避免的。因此，高容错与低运维成本将成为我们选型中最为看重的。

2016 年由雅虎开源贡献给 Apache 的 Pulsar，因其云原生、低延迟分布式消息队列与流式处理平台的标签，在开源社区引发轰动与追捧。在对其进行相关调研后，发现恰好 Pulsar 也是消息业务与存储分离的架构，而存储层则是另一个 Apache 开源基金会的 BookKeeper。

二、BookKeeper

BookKeeper 作为一款可伸缩、高容错、低延迟的分布式强一致存储服务已被部分公司应用于生产环境部署使用，最佳实践案例包括替代 HDFS 的 namenode、Pulsar 的消息存储与消费进度持久化以及对象存储。

2.1 基本架构



BookKeeper 基本架构

Zookeeper 集群用于存储节点发现与元信息存储，提供强一致性保证；

Bookie 存储节点，提供数据的存储服务。写入和读取过程中，Bookie 节点间彼此无须通信。

Bookie 启动时将自身注册到 Zookeeper 集群，暴露服务；

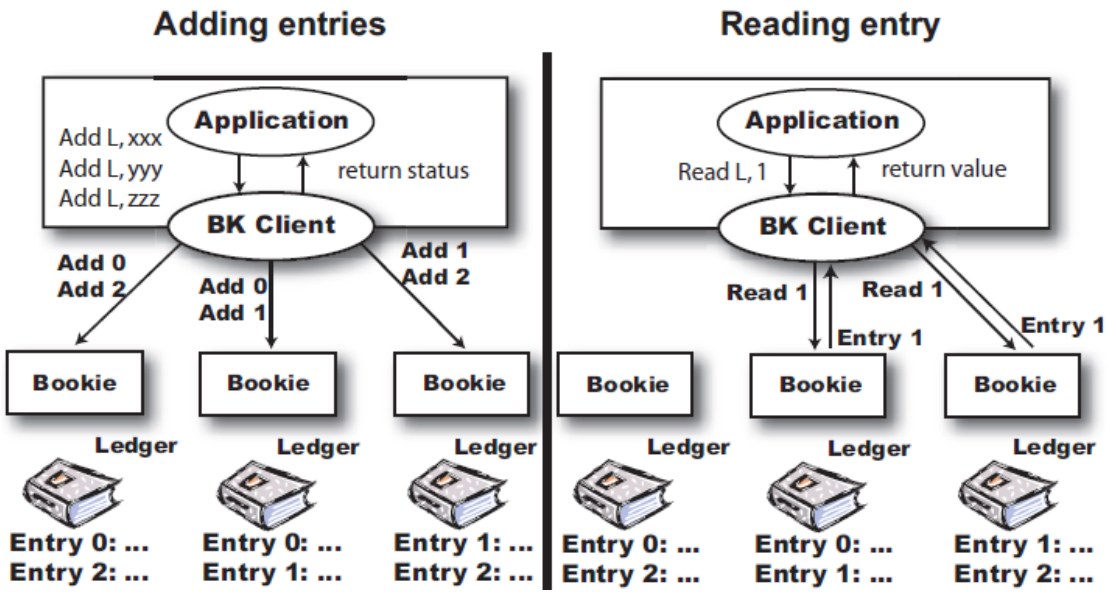
Client 属于胖客户端类型，负责与 Zookeeper 集群和 BookKeeper 集群直接通信，且根据元信息完成多副本的写入，保证数据可重复读。

2.2 基本特性

（1）基本概念

- Entry：数据载体的基本单元
- Ledger：entry 集合的抽象，类似文件
- Bookie：ledger 集合的抽象，物理存储节点
- Ensemble：ledger 的 bookie 集合

（2）数据读写



BookKeeper 数据读写

bookie 客户端通过创建而持有一个 ledger 后便可以进行 entry 写入操作，entry 以带状方式分布在 enemble 的 bookie 中。entry 在客户端进行编号，每条 entry 会根据设置的副本数（Qw）要求判定写入成功与否；

bookie 客户端通过打开一个已创建的 ledger 进行 entry 读取操作，entry 的读取顺序与写入保持一致，默认从第一个副本中读取，读取失败后顺序从下一个副本重试。

（3）数据一致性

持有可写 ledger 的 bookie 客户端称为 Writer，通过分布式锁机制确保一个 ledger 全局只有一个 Writer，Writer 的唯一性保证了数据写入一致性。Writer 内存中维护一个 LAC（Last Add Confirmed），当满足 Qw 要求后，更新 LAC。LAC 随下一次请求或定时持久化在 bookie 副

本中，当 ledger 关闭时，持久化在 Metadata Store（zookeeper 或 etcd）中；

持有可读 ledger 的 bookie 客户端称为 Reader，一个 ledger 可以有任意多个 Reader。LAC 的强一致性保证了不同 Reader 看到统一的数据视图，亦可重复读，从而保证了数据读取一致性。

（4）容错性

典型故障场景：Writer crash 或 restart、Bookie crash。

Writer 故障，ledger 可能未关闭，导致 LAC 未知。通过 ledger recover 机制，关闭 ledger，修复 LAC；

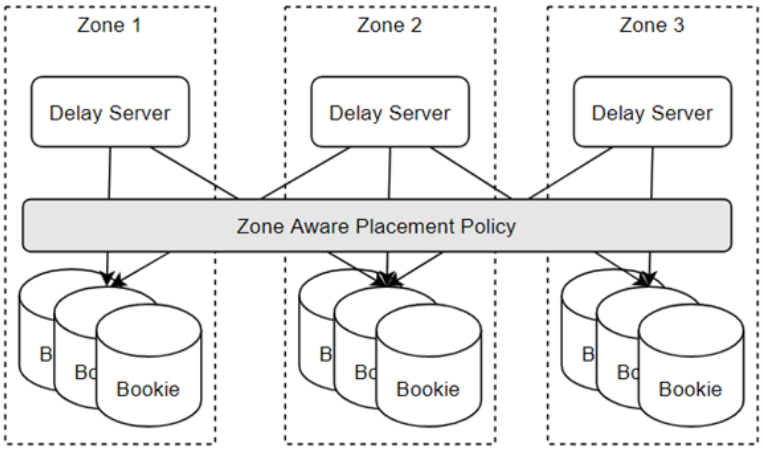
Bookie 故障，entry 写入失败。通过 ensemble replace 机制，更新一条新的 entry 路由信息到 Metadata Store 中，保障了新数据能及时成功写入。历史数据，通过 bookie recover 机制，满足 Qw 副本要求，夯实了历史数据读取的可靠性。至于副本所在的所有 bookie 节点全部故障场景，只能等待修复。

（5）负载均衡

新扩容进集群的 bookie，当创建新的 ledger 时，便自动均衡流量。

2.3 同城多中心容灾

上海区域 (region) 存在多个可用区 (az, available zone)，各可用两两间网络延迟低于 2ms，此种网络架构下，多副本分散在不同的 az 间是一个可接受的高可用方案。BookKeeper 基于 Zone 感知的 ensemble 替换策略便是应对此种场景的解决方案。



基于 Zone 感知策略的同城多中心容灾

开启 Zone 感知策略有两个限制条件：a）E % Qw == 0；b）Qw > minNumOfZones。其中 E 表示 ensemble 大小，Qw 表示副本数，minNumOfZones 表示 ensemble 中的最小 zone 数目。

譬如下面的例子：

minNumOfZones = 2
desiredNumZones = 3
E = 6
Qw = 3
[z1, z2, z3, z1, z2, z3]

故障前，每条数据具有三副本，且分布在三个可用区中；当 z1 故障后，将以满足 minNumOfZones 限制生成新的 ensemble：[z1, z2, z3, z1, z2, z3] -> [z3, z2, z3, z3, z2, z3]。显然对于三副本的每条数据仍将分布在两个可用区中，仍能容忍一个可用区故障。

（1）DNSResolver

客户端在挑选 bookie 组成 ensemble 时，需要通过 ip 反解出对应的 zone 信息，需要用户实现解析器。考虑到 zone 与 zone 间网段是认为规划且不重合的，因此，我们落地时，简单的实现了一个可动态配置生效的子网解析器。示例给出的是 ip 精确匹配的实现方式。

```
public class ConfigurableDNSToSwitchMapping extends AbstractDNSToSwitchMapping {

    private final Map<String, String> mappings = Maps.newHashMap();

    public ConfigurableDNSToSwitchMapping() {
        super();
        mappings.put("192.168.0.1", "/z1/192.168.0.1"); // /zone/upgrade domain
        mappings.put("192.168.1.1", "/z2/192.168.1.1");
        mappings.put("192.168.2.1", "/z3/192.168.2.1");
    }

    @Override
    public boolean useHostName() {
        return false;
    }

    @Override
    public List<String> resolve(List<String> names) {
        List<String> rNames = Lists.newArrayList();
        names.forEach(name -> {
            String rName = mappings.getOrDefault(name, "/default-zone/default-
upgradedomain");
            rNames.add(rName);
        });
        return rNames;
    }
```

}

可配置化 DNS 解析器示例

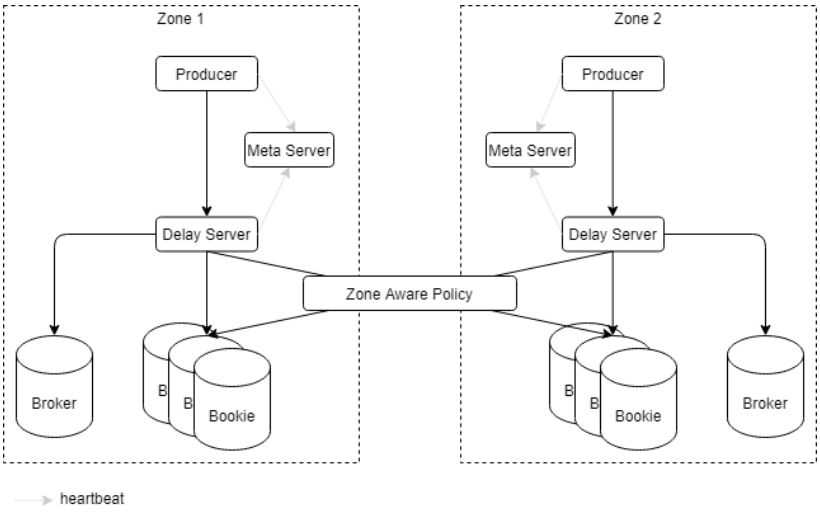
（2）数据副本分布在单 zone

当某些原因（譬如可用区故障演练）导致只有一个可用区可用时，新写入的数据的全部副本都将落在单可用区，当故障可用区恢复后，仍然有部分历史数据只存在于单可用区，不满足多可用区容灾的高可用需求。

AutoRecovery 机制中有一个 PlacementPolicy 检测机制，但缺少恢复机制。于是我们打了个 patch，支持动态机制开启和关闭此功能。这样，当可用区故障恢复后可以自动发现和修复数据全部副本分布在单可用区从而影响数据可用性的问题。

三、弹性架构落地

引入 BookKeeper 后，延迟消息服务的架构相对漂亮不少。消息业务层面和存储层面完全分离，延迟消息服务本身无状态化，可以轻易伸缩。当可用区故障后，不再需要主从切换。



延迟消息服务新架构

3.1 无状态化改造

存储层分离出去后，业务层实现无状态化成为可能。要达成这一目标，还需解决一些问题。我们先看看 BookKeeper 使用上的一些约束：

- BookKeeper 不支持共享写入的，也即业务层多个节点如果都写数据，则各自写的必然是不同的 ledger；
- 虽然 BookKeeper 允许多读，但多个应用节点各自读取的话，进度是相互独立的，应用必须自行解决进度协调问题。

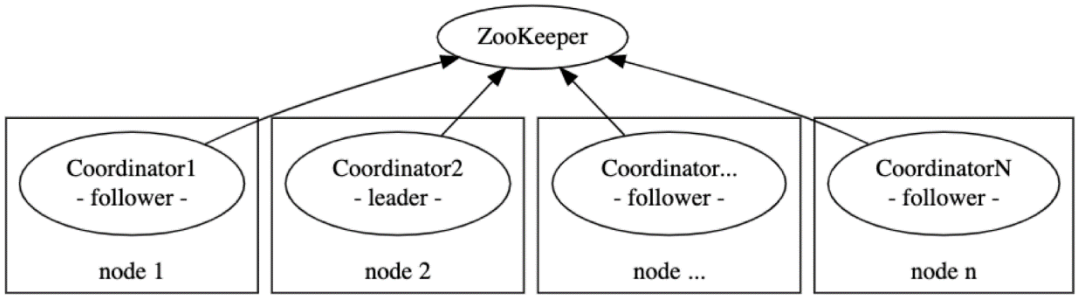
上述两个主要问题，决定我们实现无状态和弹性扩缩容时，必需自行解决读写资源分配的问

题。为此，我们引入了任务协调器。

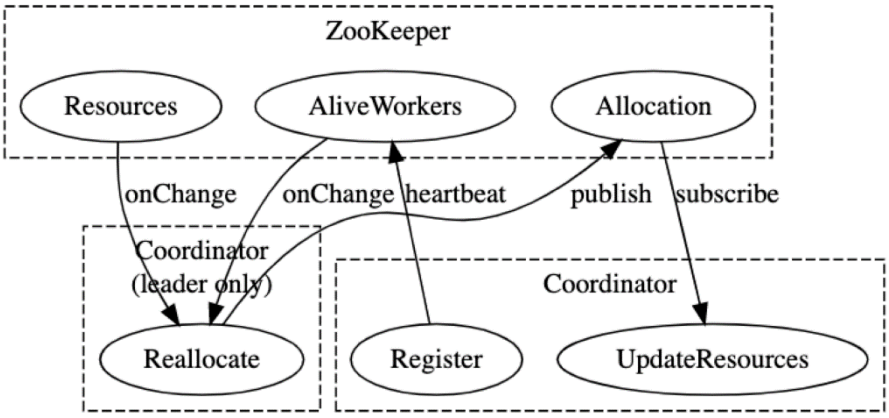
我们首先将存储资源进行分片管理，每个分片上都支持读写操作，但同一时刻只能有一个业务层节点来读写。如果我们把分片看作资源，把业务层节点看作工作者，那么任务协调器的主要职责为：

- 在尽可能平均的前提下以粘滞优先的方式把资源分配给工作者；
- 监视资源和工作者的变化，如有增减，重新执行职责 1；
- 在资源不够用时，根据具体策略配置，添加初始化新的资源。

由于是分布式环境，协调器自身完成上述职责时需要保证分布式一致性，当然还要满足可用性要求。我们选择了基于 ZooKeeper 进行选主的一主多从式架构。



如图所示，协调器对等部署在业务层应用节点中。运行时，协调器通过基于 ZooKeeper 的 leader 竞选机制决出 leader 节点，并由 leader 节点负责前述任务分配工作。



协调器选举的实现参考 ZooKeeper 官方文档，这里不再赘述。

3.2 持久化数据

原有架构将延迟消息根据调度时间按每 10 分钟桶存储在本地，时间临近的桶加载到内存中，使用 HashedWheelTimer 来调度。该设计存在两个弊端：

- 分桶较多（我们支持 2 年范围的延迟，理论分桶数量达 10 万多）；

- 单个桶的数据（10 分钟）如不能全部加载到内存，则由于桶内未按调度时间排序，可能出现未加载的部分包含了调度时间较早的数据，等它被加载时已经滞后了。

弊端 1 的话，单机本地 10 万+文件还不算多大问题，但改造后这些桶信息以元信息的方式存储在 ZooKeeper 上，我们的实现方案决定了每个桶至少占用 3 个 ZooKeeper 节点。假设我们要部署 5 个集群，平均每个集群有 10 个分片，每个分片有 10 万个桶，那使用的 ZooKeeper 节点数量就是 1500 万起，这个量级是 ZooKeeper 难以承受的。

弊端 2 则无论新老架构，都是个潜在问题。一旦某个 10 分钟消息量多一些，就可能导致消息延迟。往内存加载时，应该有更细的颗粒度才好。

基于以上问题分析，我们参考多级时间轮调度的思路，略加变化，设计了一套基于滑动时间分桶的多级调度方案。

调度器名称	屏障（最小调度时间）	单桶大小	典型桶数量
L3w	T + 14 days	1 week	104
L2d	T + 2 days	1 day	14
L1h	T + 2 hours	1 hour	48
L0m	T + 0	1 minute	120

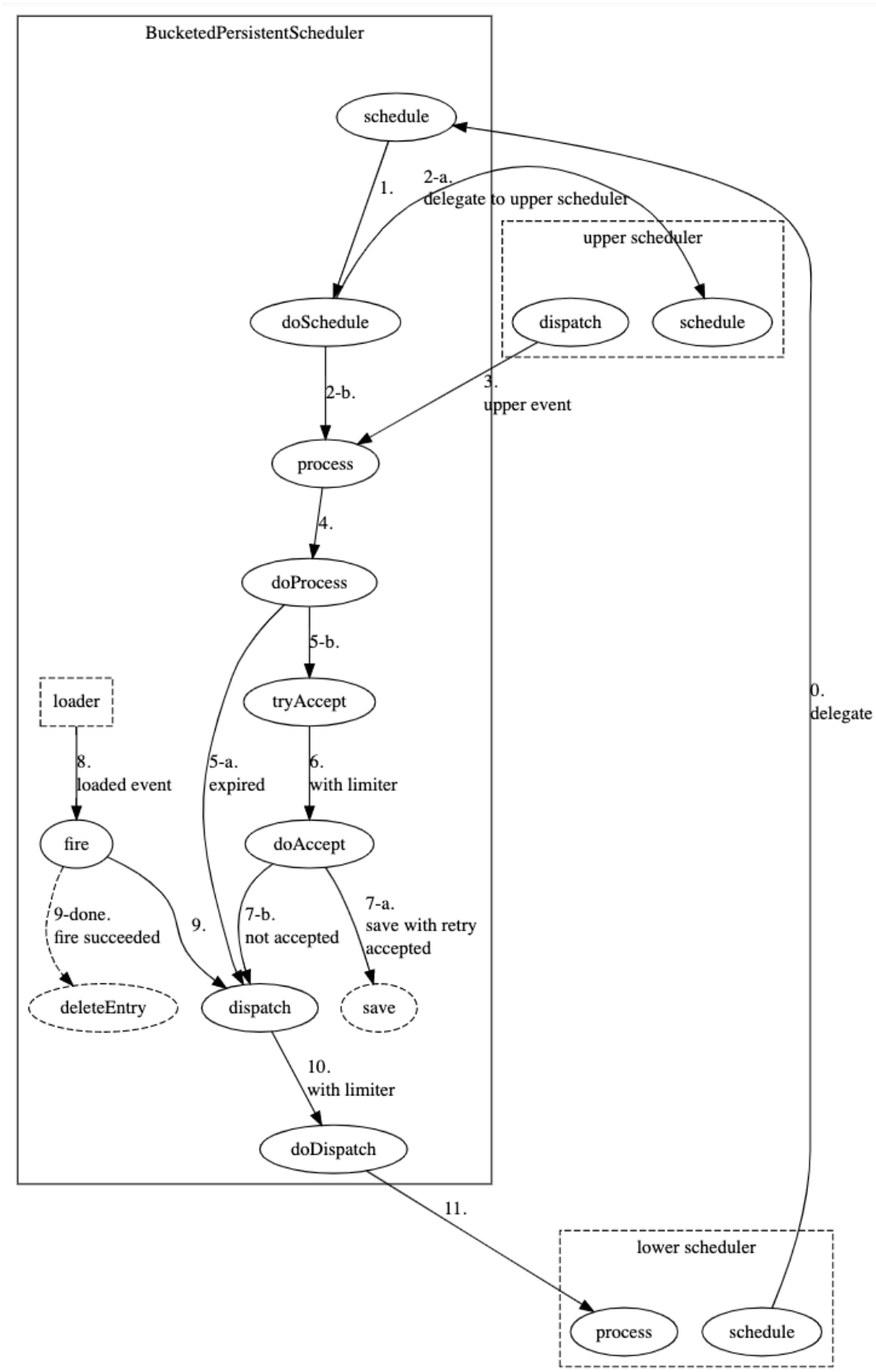
如上表所示，最大的桶是 1 周，其次是 1 天，1 小时，1 分钟。每个级别覆盖不同的时间范围，组合起来覆盖 2 年的时间范围理论上只需 286 个桶，相比原来的 10 万多个桶有了质的缩减。

同时，只有 L0m 这一级调度器需要加载数据到 HashedWheelTimer，故而加载粒度细化到了 1 分钟，大大减少了因不能完整加载一个桶而导致的调度延迟。

多级调度器以类似串联的方式协同工作。

每一级调度器收到写入请求时，首先尝试委托给其上级（颗粒度更大）调度器处理。如果上级接受，则只需将上级的处理结果向下返回；如果上级不接受，再判断是否归属本级，是的话写入桶中，否则打回给下级。

每一级调度器都会将时间临近的桶打开并发送其中的数据到下一级调度器。比如 L1h 发现最小的桶到了预加载时间，则把该桶的数据读出并发送给 L0m 调度器，最终该小时的数据被转移到 L0m 并展开为（最多）60 个分钟级的桶。


BucketedPersistentScheduler
schedule
1.
2-a.
delegate to upper scheduler
upper scheduler
doSchedule
dispatch
schedule
2-b.
3.
upper event
process
4.
doProcess
5-b.
loader
tryAccept
0.
delegate
8.
loaded event
5-a.
expired
6.
with limiter
fire
doAccept
9-done.
fire succeeded
9.
7-b.
not accepted
7-a.
save with retry
accepted
deleteEntry
dispatch
save
10.
with limiter
doDispatch
11.
lower scheduler
process
schedule

四、未来规划

目前 bookie 集群部署在物理机上，集群新建、扩缩容相对比较麻烦，未来将考虑融入 k8s 体系；bookie 的治理与平台化也是需要考虑的；我们目前只具备同城多中心容灾能力，跨 region 容灾以及公/私混合云容灾等高可用架构也需要进一步补强。

五、参考文档

- QMQ 开源地址：
https://github.com/qunarcorp/qmq

- Pulsar 官网：
https://pulsar.apache.org/

- BookKeeper 官网：
https://bookkeeper.apache.org/

- 为什么选择 BookKeeper：
https://medium.com/streamnative/why-apache-bookkeeper-part-1-consistency-durability-availability-ac697a3cf7a1

携程百亿级缓存系统探索之路——本地缓存结构选型与内存压缩

【作者简介】一十，携程资深后端开发工程师；振青，携程高级后端开发专家。

一、前言

携程酒店查询服务是酒店 BU 后端的核心服务，主要负责提供所有酒店动态数据计算的统一接口。在处理请求的过程中，需要使用到酒店基础属性信息、价格信息等多维度的数据信息。为了保证服务的响应性能，酒店查询服务对所有在请求过程中需要使用到的相关数据进行了缓存。随着携程酒店业务的发展，查询服务目前在保证数据最终一致性以及增量秒级更新延迟的情况下，在包括服务器本地内存以及 Redis 等多种介质上缓存了百亿级的数据。

本文将主要讨论酒店查询服务技术团队是如何在保证读取效率的前提下，针对存储在服务器本地的缓存数据进行存储结构选型以及优化的过程。

二、内存结构选型

为了寻找合适的内存结构以达到理想的效果，本节将详细探讨在通用数据查询场景下，不同类型的数据结构的可行性与优劣势。

2.1 缓存结构的基础需求

2.1.1 支持高性能读取

在大部分应用场景下，之所以需要在服务器内存中存储缓存数据，是因为请求处理过程中需要高频次读取各类数据。为了保证服务的响应性能，我们有的时候会选择将数据提前存储在本地内存，利用空间换取时间。酒店查询服务就有这样的需求：服务平均每次请求需要读取数据上千次，同时对响应时间也有着严格的要求。因此，在这种高频次访问缓存的场景下，对数据的查找性能便有着极高的要求。

在常见的数据结构中，数组和散列表都能提供 O(1)的查询速度，是不考虑其他因素下最高性能的选择。查找复杂度为 O(log2N)的树则其次，其查找速度和数据规模有关，一般只能在数据规模很小的场景下选用。

2.1.2 支持高更新频率

在实际应用场景下，生产环境的缓存数据必然有新鲜度要求。面对海量数据，高频度的数据更新几乎无可避免。特别是像价格这类数据，一方面更改频次极高，另一方面又必须保证新的增量数据可以在秒级内快速同步至缓存中。这就要求所使用的缓存数据结构必须支持高性能并发读写的场景。如果随意的使用锁机制或是线程不安全的存储结构都会可能导致一些预期外的问题与风险：

（1）并发更改风险

众所周知，Java 提供实现的最常用的散列表 HashMap 是非线程安全的数据结构。若直接使用该类作为缓存结构，则在并发读写时就可能会因为重新 Hash 而读到错误的数据，甚至在极端情况下产生死循环的问题。

（2）滥用读写锁

在频繁并发更新与读取的场景下，错误的锁机制很有可能导致在高频次写入时直接卡死应用处理请求时的高频次读取，进而产生大量请求排队以及其他问题。

因此，高更新频率需求所带来的线程安全问题，导致大部分的基础数据结构都无法适用于存储生产缓存数据。在绝大部分情况下，都需要牺牲一部分性能选择线程安全的数据结构。当然，对于某些特殊场景，也可根据需要来设计定制化的结构与锁机制来达到更优的性能。

经过上面的简单分析后，我们可以暂时认为线程安全的数组和散列表是一个较优的用以承载缓存数据的结构。

2.2 低空间开销的结构选型

由于实际应用的内存都是有限的，因此在保障读写性能的同时，我们也需要思考如何降低缓存所消耗的内存资源。为了保证服务正常的响应请求，酒店查询服务需要在本地存储千万量级的数据，而缓存能够在虚拟机上使用的内存空间却非常有限。因此，除了对数据本身进行过滤等预处理之外，用以存储数据的通用结构的内存开销也要尽可能的小。

在对不同数据结构进行分析前，我们需要从最基础的问题开始：Java 中的一个对象是以何种结构存储在内存里的？

2.2.1 Java 对象内存结构模型

一个 Java 对象在内存中的存储结构通常包括对象头、实例数据与对齐填充。

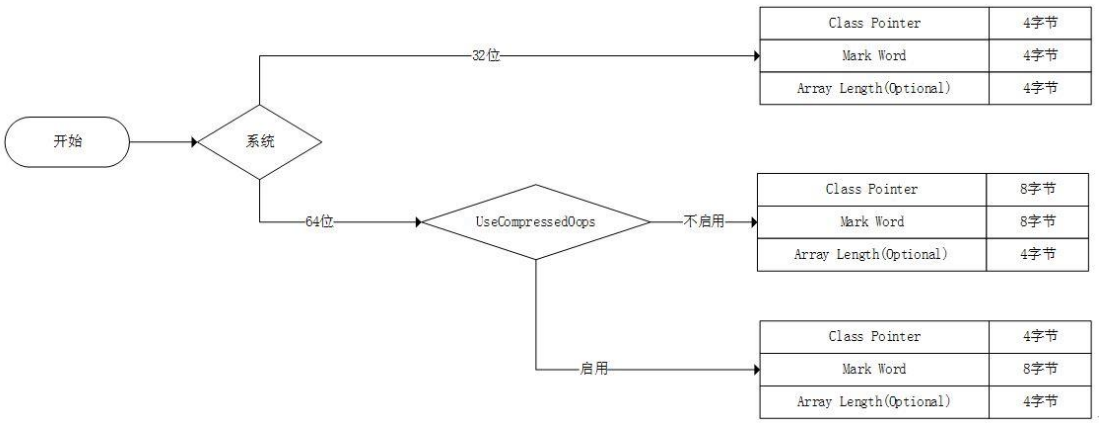
（1）对象头

对象头用于存储对象的标记位（Mark Word）与类型指针（Class Pointer）。

标记位里存储了包含锁状态与 GC 标记位等信息，其在 32 位系统上占存 4 字节而在 64 位系统上占存 8 字节。

类型指针是一个对象指向它元数据的指针，因此，其在 32 位系统上占存 4 字节，在 64 位系统上占存 8 字节。同时，若在 64 位机器上开启指针压缩参数-XX:UseCompressedOops，则此时类型指针在对象头中仅占存 4 字节。

另外，若实体为数组，则会额外有 4 个字节用以存储该数组的长度。



（2）实例数据

实例数据内部存储了对象所定义的所有成员变量。这些成员变量会紧密排列，若对象是由子类创建的，则其父类的成员变量也会包含在其中。若成员变量为 NULL 值，则不会给该成员变量申请指针空间。

（3）对齐填充

若对象所申请的内存空间不是 8 的倍数，则 JVM 会添加合适的对齐填充使得整个对象所申请的空间为 8 的倍数。

综上，若一个简单实例对象内部存储一个 int 字段以及一个 byte 字段，那么在开启指针压缩的 64 位机器上其占存应为 24 字节。其中包含对象头的 8 字节标识位与 4 字节类型指针、内部字段 int 的 4 字节与 byte 的 1 字节以及对齐填充 7 字节。

对象头		实例数据		对齐
标识位	类型指针	int	byte	
8	4	4	1	7

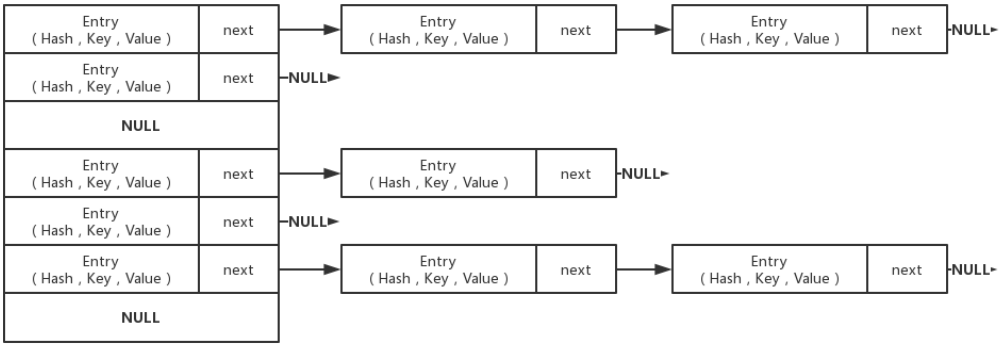
2.2.2 原生 HashMap 结构内存开销

基于上述的 Java 实例内存结构的基础理论，我们将以 HashMap<Integer,Integer>为主要示例，详细探讨 JDK 原生 HashMap 内部结构以及其内存开销。

下表是一个简单的实验结果。我们统计了在开启指针压缩的 64 位机器上，不同数据条数的键值类型均为 Integer 的 HashMap 的内存占存。作为参考，我们将其所存储的所有 Integer 实例的占存视作数据占存，将剩余的占存视作结构占存。从下表的实验结果来看，无论在何种数据规模下，HashMap 内部结构的内存开销占比都很高，占到了整体的 55%以上。

数据量	HashMap 占存	数据占存	结构占存	结构开销占比
32	2352	1024	1328	56.46%
512	36912	16384	20528	55.61%
4096	294960	131072	163888	55.56%
65536	4718640	2097152	2621488	55.56%
1048576	75497520	33554432	41943088	55.56%

需要知道此类现象的成因还是需要从 HashMap 内部存储结构为入手点进行分析。如下图所示，HashMap 主要由一个哈希桶数组及多个存储在哈希桶中的节点 Node<Key,Value>所构成。



下面我们来分别具体解析一下哈希桶数组 table 和数据节点 Node 的内存开销。

（1）哈希桶数组 table

哈希桶数组实际是用于存储数据节点 Node 的数组。程序将根据数据 Key 的 HashCode 运算得到数据节点 Node 实际应存储在哈希桶数组的哪一个下标位置。通过哈希桶打散数据后，程序可以通过 Key 快速的查找到实际数据节点。其在源码中实际定义如下：

```
transient Node<K,V>[] table;
```

那么，在内存结构上，哈希桶就是由一个附带数组长度的对象头和数组元素集合组成。因此，一个长度为 N 的哈希桶数组的占存大小就会是：

8（对象头标识位）+ 4（类型指针）+ 4（数组长度 + 4 （实体引用）*N （实体数量）字节 + 对齐字节。即，长度为 32 的哈希桶数组则实际占存即为 16 + 4 *32 = 144 字节。

对象头			数据开销	对齐
标识位	类型指针	数组长度	数据实体引用	
8	4	4	4 * N	4 或 0

为了提升读写性能，HashMap 中哈希桶数组的实际长度并不会总是等于实际存储的数据量。哈希桶数组的实际长度在两个时候会产生变化：

- 初始化

在 HashMap 进行实例创建的时候，程序会按照所指定的容量（默认为 16）向上取最接近的 2 的整数幂作为实际初始化容量。该容量也是哈希桶数组的长度。比如外部创建一个指定容量为 100 的 HashMap，则其内部哈希桶数组的实际初始长度为 128。

- 扩容

HashMap 为了确保其读写效率，当内部数据量到达一定规模时，会进行扩容操作。而其负载因子和当前哈希桶数组的长度二者相乘所得出的扩容阈值决定了扩容前在哈希表内部最大元素数量。例如，一个容量为 32、负载因子为默认的 0.75 的 HashMap 的扩容阈值即为 32*0.75=24。那么，当此 HashMap 被插入 24 条数据后，其内部的原先 32 长的哈希桶数组就会被扩容至原长度的 2 倍 64。

综合上述的哈希桶长度策略，其实可以很明显的看到 HashMap 所存储的哈希桶实体数组在绝大部分情况下总是会将冗余出比实际数据量多一些的空间，以减少哈希碰撞、提升读取效率。

下表是在不同数据规模下哈希桶数组相对于普通实体数组，冗余的数组长度及其额外的开销。

数据量	实体数组长度	哈希桶数组长度	哈希桶数组冗余长度	冗余内存开销(字节)
50	50	128	78	312
200	200	512	312	1248
1000	1000	2048	1048	4192
10000	10000	16384	6384	25536
100000	100000	262144	162144	648576

那么依据上面的理论，我们就可以推算出不同数据量的 HashMap 中哈希桶数组的内存开销及其占比。

数据量	数据占存	哈希桶数组占存	总大小	哈希桶耗存占比
32	1024	272	2352	11.56%
512	16384	4112	36912	11.14%
4096	131072	32784	294960	11.11%
65536	2097152	262160	4718640	5.56%
1048576	33554432	4194320	75497520	5.56%

（2）数据节点 Node

Node 类继承于 Map.Entry，是 HashMap 中存储数据的基本单元。其内部除了存储了键值对数据外，同时存储了节点的哈希值以及是当其在链表或红黑树中时，其下个 Node 节点的引用。

```
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    V value;
    Node<K,V> next;
```

那么，我们可以依据其内部结构如计算出一个 Node 实例的字节数为 32 个字节。若要使用 Node 存储 32 个 Integer 键值对，那么所有 32 个节点实体一共要占用 1024 个字节。

对象头		数据引用		其他结构		对齐
标识位	类型指针	Key 引用	Value 引用	Next	Hash	
8	4	4	4	4	4	4

那么我们可以在前面的实验数据中再添加上 Node 的数据，得到完整的 HashMap 内存开销各部分的占比：

数据量	数据占存	哈希桶数组占存	Node 占存	总大小
32	1024	272	1024	2352
512	16384	4112	16384	36912
4096	131072	32784	131072	294960
65536	2097152	262160	2097152	4718640
1048576	33554432	4194320	33554432	75497520

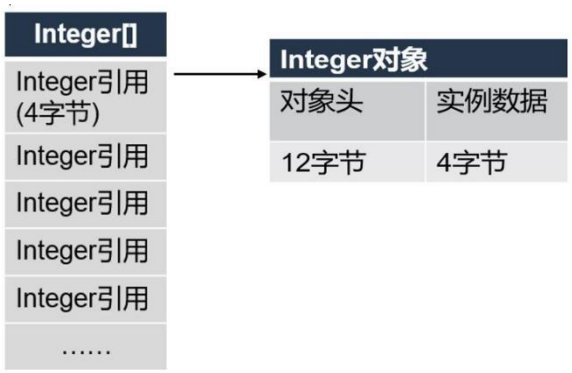
所以，在原生 HashMap 消耗了大量的额外空间结构换取其读写性能。这使得原生 HashMap 在大数据量且内存有限的应用上，并不是一个最优的缓存结构选型。

2.2.3 包装类型损耗

由于 Java 的泛型机制，绝大部分的数据结构的存储的类型只能声明为包装类。因此，即使需要存储是整型等基础类型，也将其不得不转换为对应的包装类型来存储在内存中。这不仅会有存取时产生额外的装拆箱的性能损耗，存储包装类相较基础类型也会产生更大的内存开销。以实际应用场景中最为常见的整型为例，我们将简单比较一下 Integer[] 和 int[] 这两种数组的内存大小差异。

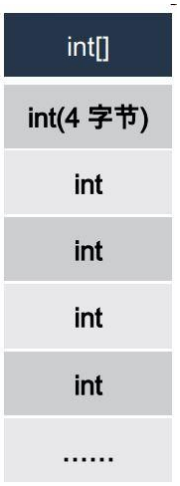
（1）Integer[ ]

由于 Integer 是包装类，因此数组中存储的实际是 4 字节长的 Integer 的引用。而对于一个 Integer 实例本身来说，参考前文所述，除了 4 字节的实际数据外，其还需要 12 字节来保存其对象头。所以在集合中要保存一个 Integer 的实际开销就会是 4 + 12 + 4 = 20 字节。



（2）int[ ]

基础类型的 int[]则简单的多：在创建数组时，仅需为每个元素开辟 4 字节来保存整型即可。



所以，理论上每个 Integer 都会比 int 额外产生 16 字节的内存开销 。从实验结果可以看出，若我们可以直接使用基础类型来代替包装类存储时，可以大幅减少内存占存。此结论对其他如 HashMap 等数据结构也同样有效。

数据量	int[]占存	Integer[]占存	额外开销占比
32	144	656	78.05%
512	2064	10256	79.88%
4096	16400	81936	79.98%
65536	262160	1310736	80.00%
1048576	4194320	20971536	80.00%

2.2.4 其他结构选型

为了在保证读写性能的情况下尽可能压缩内存开销，我们调研了一些第三方的开源集合框架来尝试在内存和性能上尽可能取得平衡。

（1）ConcurrentHashMap

ConcurrentHashMap 是 HashMap 的线程安全版本，内部也是使用数组、链表与红黑树的结构来存储元素。相较于同样 JDK 中线程安全的 HashTable 来说，其锁竞争更少、读写效率更高。

（2）SparseArray

SparseArray 即稀疏数组，是 Android 提供的建议替换 HashMap<Integer, E>的用来存储整型类型对象键值对的类。其内部主要使用了数组作为存储方式，比 HashMap<Integer, E>要高效轻量。

（3）Guava Cache

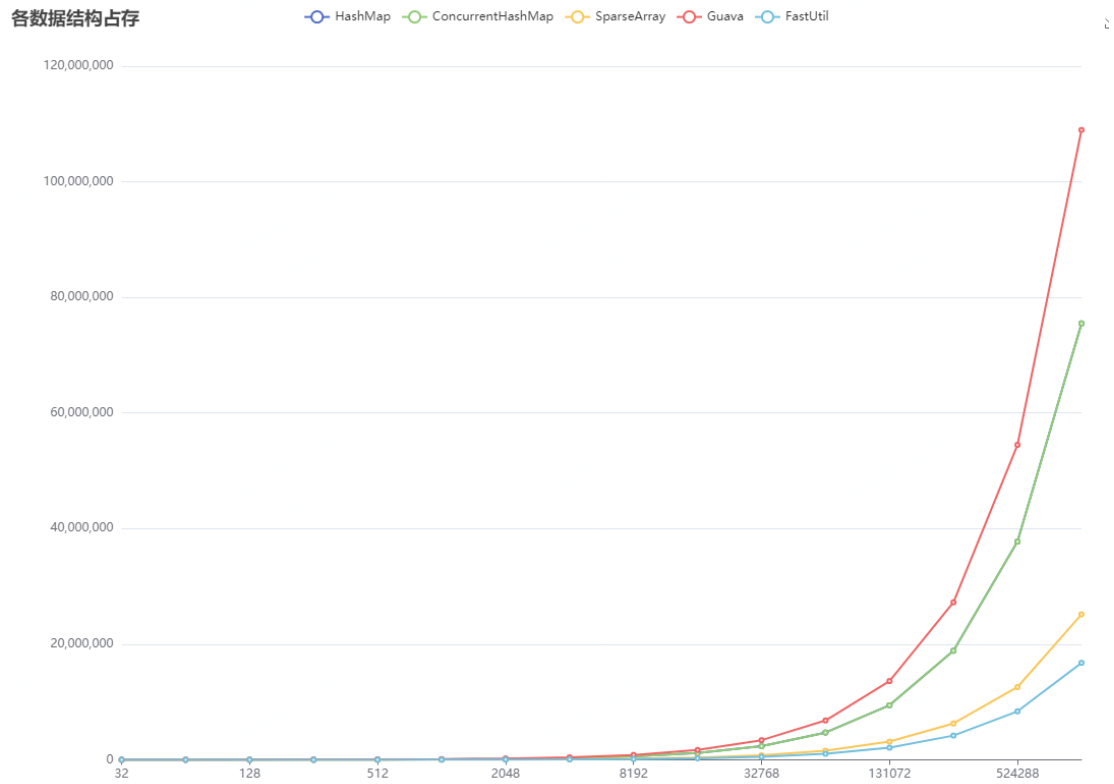
Guava Cache 是 google 开源的一款本地缓存工具库。其使用多个 segments 方式的细粒度锁，提供了支持高并发场景的线程安全的存储结构。

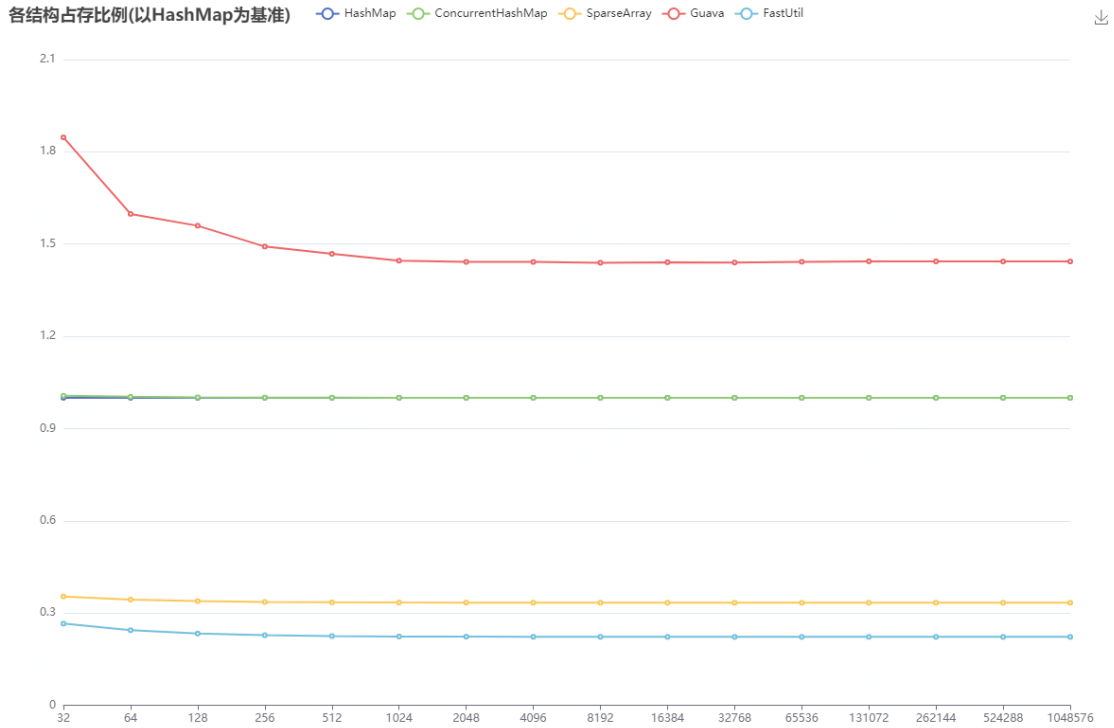
（4）Fastutil

FastUtil 是一个高性能的集合框架，提供了以基础类型为元素的集合来代替 JDK 原生的集合类型。基础类型为元素的集合避免了大量的基础类型的装箱拆箱。因此，在程序进行集合的遍历、根据索引获取元素的值和设置元素的值的时候，fastutil 可以提供更快的存取速度以及更低的内存消耗。

我们实验了整型键值对不同数据规模下各个集合的内存占比，并且用 HashMap 的数据作为基准进行横向比较。实验结果具体数据如下所示。

数据量	HashMap	ConcurrentHashMap	SparseArray	Guava	fastutil
32	2352	2368	832	4344	624
256	18480	18496	6208	27568	4208
1024	73776	73792	24640	106672	16496
32768	2359344	2359360	786496	3397104	524400
1048576	75497520	75497536	25165888	108970384	16777328





三、数据编码压缩

在实际应用场景下，几乎所有需要缓存的数据都有着比较高的重复率或是其他分布规律。在内存结构选型的基础上，针对于不同的数据特征，我们可以采用不同的数据编码压缩方式对

数据进行压缩处理，进一步降低缓存的内存开销。

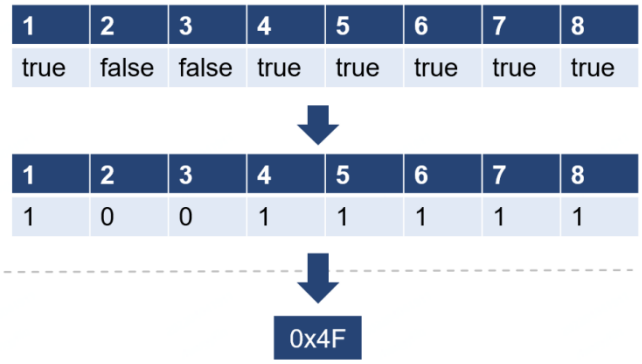
下面，我们将介绍几种常用有效的数据编码压缩方式。

3.1 常用编码技术

3.1.1 位图编码

位图（BitMap）是一种常见的编码格式，JDK 中提供的默认实现为 BitSet 类。它是用 Bit 位来存储数据的某种状态，通常指示是非有无。在最常见的情况下，当需要存储大量连续 ID 是否为 True 时，用到此类结构就可以大量减少内存的开销。

在下例中，需要存储的数据的 Key 为整型， Value 为该 Key 是否有效的状态数据。若是直接存储，则一条数据至少需要 4 个字节用于存储整型 Key 以及 4 个字节用于存储布尔型的状态值。那么，当需要存储 Key 从 1 至 8 的 8 条数据，则至少需要 64 字节。若使用位图编码技术对数据进行处理，那么我们仅需要 1 个 bit 即可存储一个 True or False 的状态信息。因此，就可以使用 1 个字节存储下所有 8 条数据的状态信息。此时，该字节的第 1 位的 bit 用以表示 Key = 1 的状态信息，第 2 位的 bit 用以 Key = 2 的状态信息，以此类推。

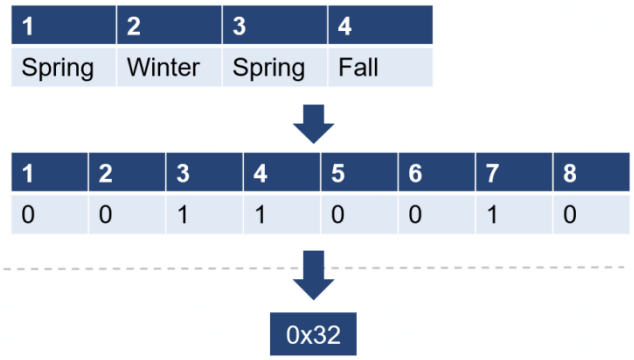


下表是在 64 位机器上开启指针压缩后， Java 原生 HashMap 与 BitSet 的耗存对照表。可以明显地看出，整体压缩效率是非常高的。在实际业务场景下，只要整型 Key 分布相对密集，就可以利用位图编码达到不错的压缩效果。

最大 ID	HashMap 占存	BitSet 占存
1w	532.84KB	2.04KB
10w	5.57MB	16.04KB
100w	53.77MB	128.04KB
1000w	521.76MB	2.00MB

另外，位图编码还可以额外扩展至一些类型较少的枚举类型上。例如，枚举类 Season 只有 4 种元素，则可以使用 2 个 bit 来代表一个属性，那么则只需 8bit 即可存储 id 从 1-4 的 4 个 Season 枚举。

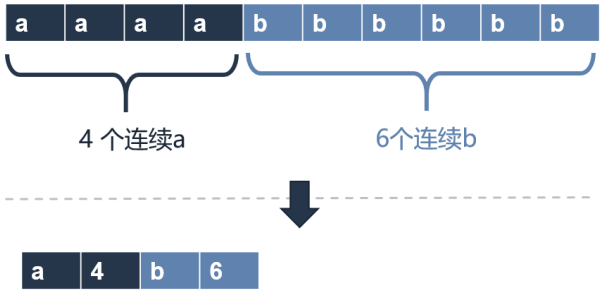
```
enumSeason{
        Spring,Summer,Fall,Winter;
}
```



3.1.2 游程编码

游程编码（Run-length encoding，RLE）是一种无损压缩数据的编码方式，主要方法是使用当前数据元素以及该元素连续出现的次数来取代数据中连续出现的部分。若数据存在大量的数据连续且重复的情况，就可以考虑使用 RLE 以降低内存。

比如，一个内部存储了 4 个连续的 a 与 6 个连续的 b 的字符串经过游程编码后为 a4b6。那么，该字符串长度就从 10 字节减少至 4 字节。

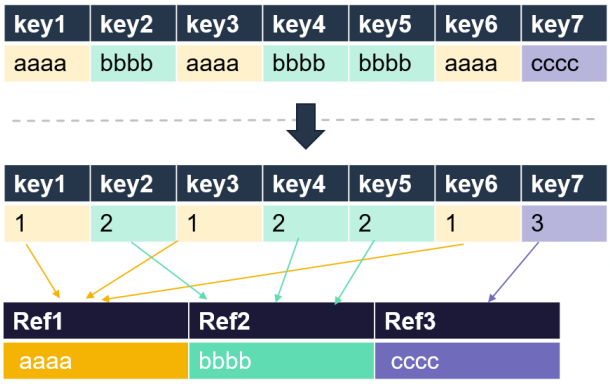


3.1.3 字典编码

字典编码是把整体重复性高的数据进行去重后建立字典，把原来存放数据的地方变为指向实体字典引用的编码方式。因为引用指针依然占存，因此适合单个的实例数据字段较多的数据缓存。

下例为原始数据为整型 Key 查询长字符串 Value 的场景。首先，将重复的字符串实体数据提取出来，将其单独作为一个实体字典进行存储。该字典 Key 为一个指针，Value 则为提取出的不重复的字符串数据。然后，原始字典的 Value 就可以变为一个指针，指向新实体字典的

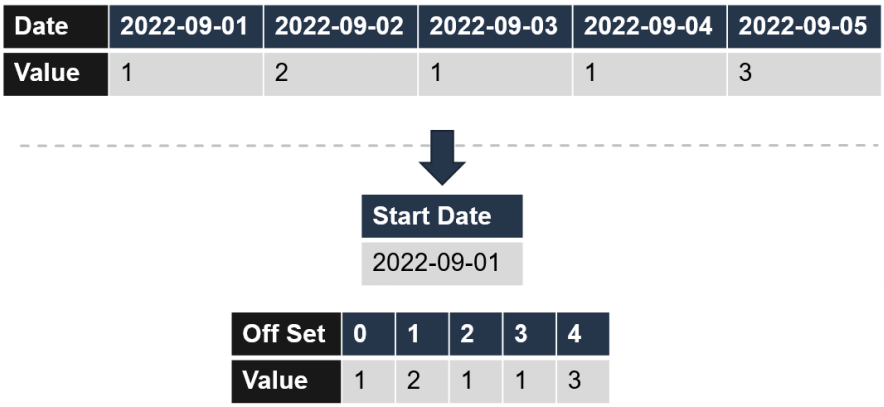
Key。当需要查询 Key1 内实际数据的时候，先在原始字典中查询到引用 Ref1，再在实体字典查询 Ref1 即可获得其 Value 值 aaaa。



3.1.4 差值编码

差值编码是对于非连续的数据 Key 通过差值计算的方式转化为连续的 Key，让字典可以转化为数组的编码方式。

下例中的数据 Key 为日期，Value 为一个整型。在日期相对连续的情况下，取所有日期的最小值为开始日期，以数据生效日期到开始日期的差值为新字典的 Key。那么编码前旧数据字典的 Key 为 Date 类型，而编码后的新数据字典的类型则可以转化为更小更泛用的 int 型。



下表是在 N 天连续的日期查整型的场景下，原生 HashMap 与编码后整型数组的耗存对照表。即使连续的日期数量较小，也可以看出整体存在的巨大差距。在实际的应用场景下，此类编码方式一般用于连续日期的缓存，也可以扩展到其他有着类似数据特征的缓存上。

日期数量	HashMap 占存	编码后整型数组占存
100	8.09KB	440B
1w	767.19KB	39.10KB
1000w	750.65MB	38.15MB

以上的编码技术可以在不同的数据场景下进行组合使用，接下来我们将简要的展示两个实际酒店查询服务缓存内存压缩优化的案例。

3.2 应用案例

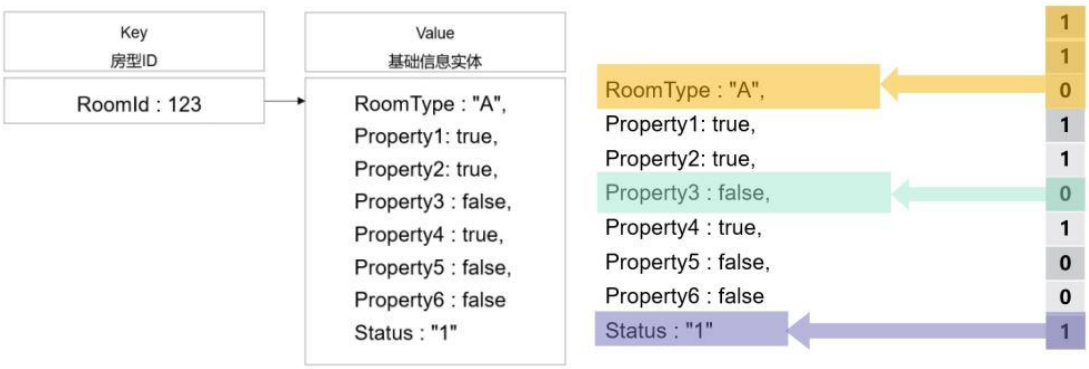
3.2.1 房型基础信息

查询服务缓存了上亿条房型信息数据。在请求处理过程中，服务可以在缓存中通过房型 ID 查询到该房型的信息。因为数据条数上亿且实体内部字段很多，因此未优化的缓存在内存中占存高达上百 GB，是一个较大的内存性能瓶颈。

因此，针对该缓存，我们使用了位图编码以及字典编码，大幅降低了其内存开销。

（1）使用位图编码对可枚举字段进行数据压缩

我们将房型数据实体上包括布尔型、枚举以及部分字符串等所有可以枚举的字段进行了位图编码，大幅降低了单个实体的占存大小。比如在下方作为例子的字段中，RoomType 虽然存储为一个 String，但是在实际业务场景中它一共只有 5 种取值可能性，因此也可以作为枚举类进行处理为 3 个 bit。



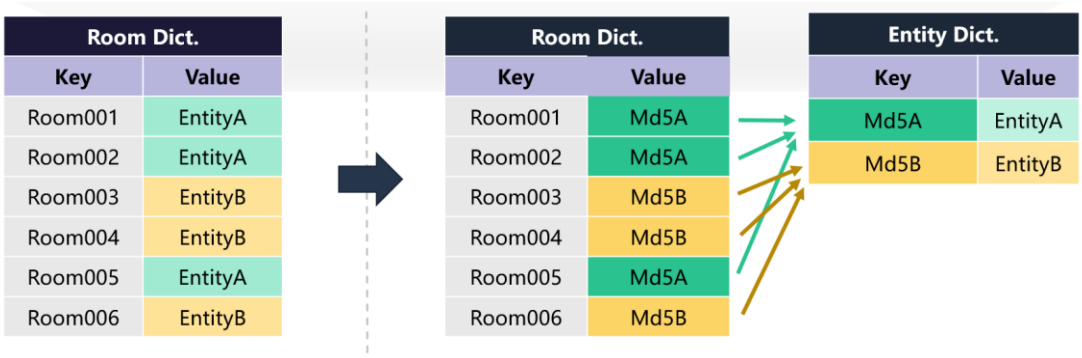
在原先存储方式的情况下，示例的一个房型实体字段就至少需要 16 字节，通过位图编码后一个房型实体字段实际仅需要 10 个 bit 即可无损的存储下所有有效信息。

	原存储格式		Bitmap化	
字段名	字段类型	占用	字段类型	占用
RoomType	string	5 + N	enum(5 type)	3
Property1	bool	1	bit	1
Property2	bool	1	bit	1
Property3	bool	1	bit	1
Property4	bool	1	bit	1
Property5	bool	1	bit	1
Property6	bool	1	bit	1
Status	string	5 + N	enum(FG,PP)	1
	总计	16(128bit) +	总计	2(10bit)

（2）使用字典编码对重复实体进行压缩

经过线上数据统计与分析，我们发现部分房型属性数据的重复率达到 99%。也就是说，虽然存在上亿个房型，但是实际去重后的房型的部分基础信息实体只有百万级。因此，在对房型基础信息实体本身进行位图编码的同时，我们采用了字典编码的方式对房型 ID 不同但内部字段信息完全重复的数据实体进行字典编码，以压缩这部分的消耗。

在实际处理过程中，我们会先将房型数据实体进行序列化后转换为 MD5，在房型字典中只存储 MD5 编码，而实体字典中存储 MD5 到实际房型信息实体的关系。在进行数据查询时，则是先通过房型 ID 在房型字典中查找到对应的 MD5 值，然后在实体字典中通过 MD5 值查找到对应的房型基础信息实体。



经过上述两个编码压缩优化后，房型实体缓存占存整体压缩率达到 2%以下，节省了数十 GB 的内存空间。

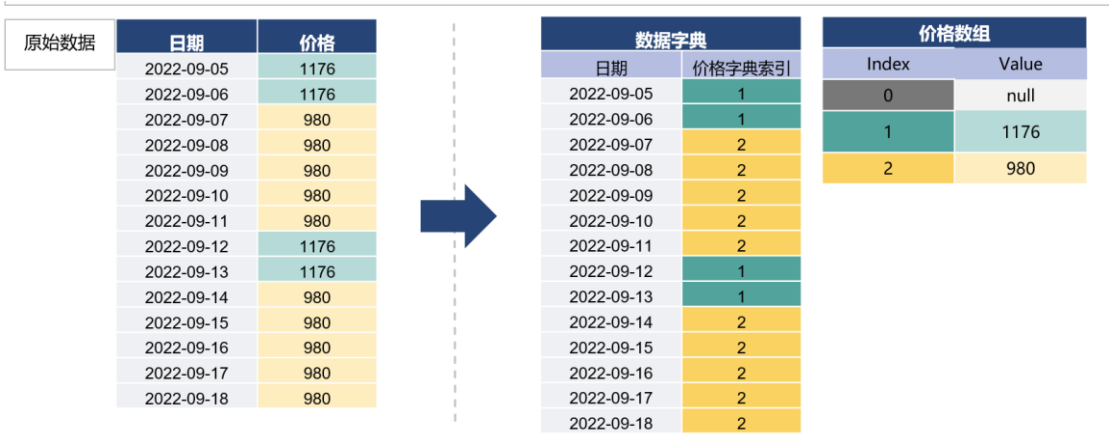
3.2.2 单天房价信息

单天房价信息缓存是存储每个房型每日价格的缓存，是查询服务数据量最大同时也是最核心的数据缓存。在应用请求处理过程中，会使用房型 ID 以及日期从该缓存中获取房型某一天的价格数据。

下面将以单房型下存储的日期信息为例，逐步展示数据压缩优化的全部过程。

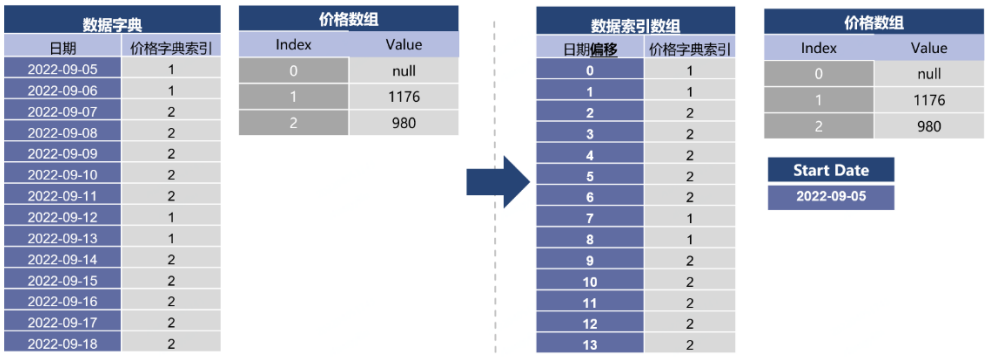
（1）使用字典编码对每日重复的价格信息进行编码

首先，将所有该房型上出现的价格提取并存储到一个价格数组上，在数据字典里则存储实际价格数据在价格字典的索引。同时，因为存在可能没有数据的日期，因此 Null 值被存储在所有价格数组上的第一个偏移 index 上，作为默认值。



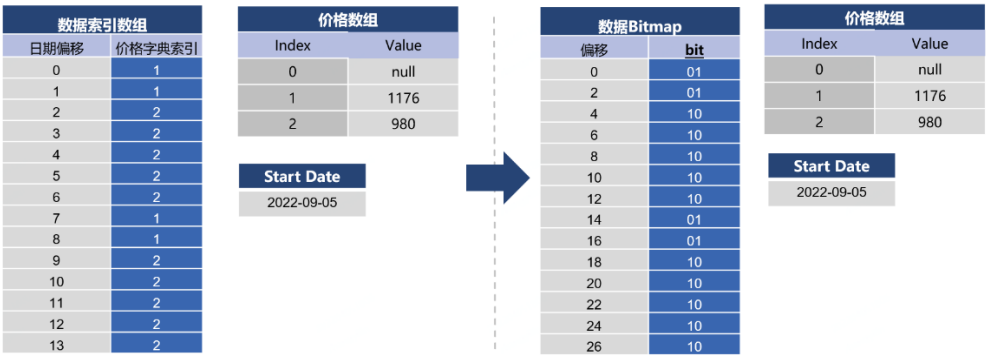
（2）使用差值编码处理日期

因为在绝大部分情况下，数据字典中的日期均为连续的，且从业务场景上来说最大的日期也不会过大，因此我们采用差值编码处理日期，将数据字典中的日期替换为与服务器启动日期之间相差天数的偏移量。此时，数据字典的 Key 则会变为一个从 0 开始的 int，那么就可以使用占存更小的数组来表示这个数据字典。该数据索引数组为一个 int[]，其下标表示日期偏移，值表示到价格字典的索引。



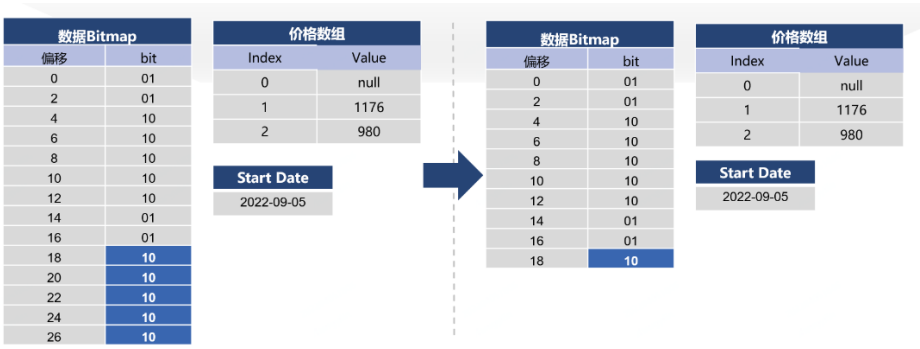
（3）使用位图编码处理可枚举的价格索引

因为单个房型下的价格数量是有限的，因此同样可以视作是枚举值的一种。对枚举值，就可以使用位图编码对数据索引数组进行压缩。在下图的数据样例中，因为价格数组长度为 3，即存在 3 种可能的价格，因此将 int 转换为 2 个 bit 进行存储，则位图的一天的偏移步长为 2。

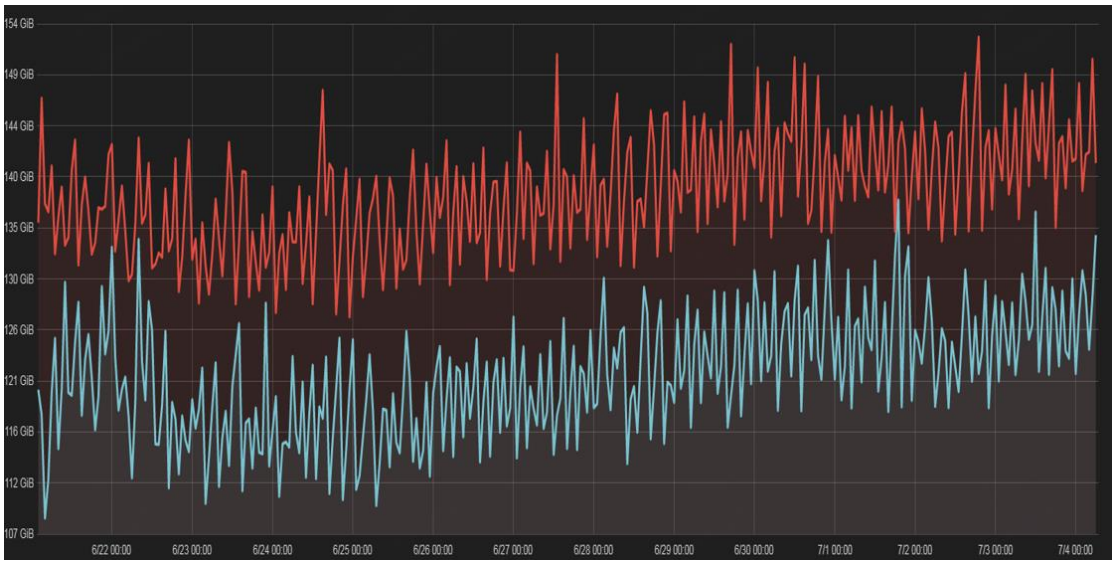


4）使用 RLE 编码处理末尾

在很多房型下的到天价格数据，在距离现在最远的日期带的价格通常都是重复的，因此，我们可以使用 RLE 的方式对末尾重复的数据进行截尾，来进一步压缩数据位图大小。



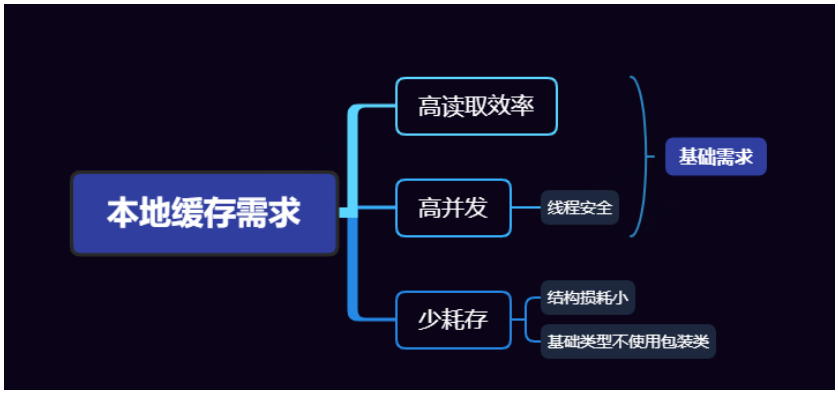
在所举的例子中，其在内存中单对象实例数据部分的内存可以从最初的数百字节降低至最终的 31 字节。而在实际业务场景中，该单天房价数据经过压缩处理后实际压缩率为 60%左右。



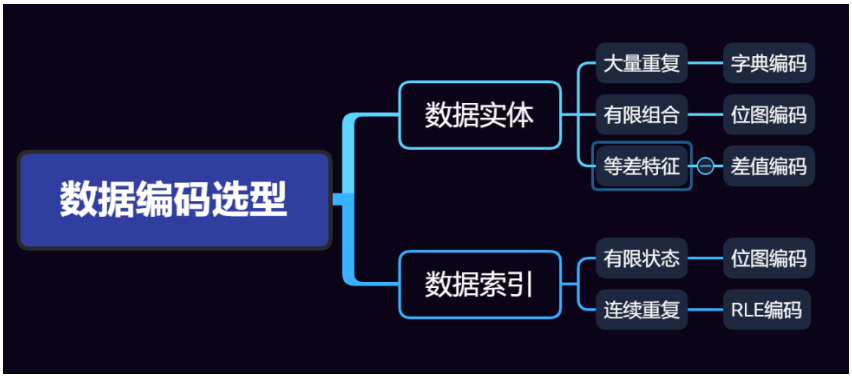
四、总结

本文主要介绍了携程酒店查询服务在本地缓存数据结构选型以及优化方面的探索与实际应用案例。

在常规缓存数据的存储结构选型上，我们先根据缓存场景的需求，分析比较了不同数据结构后，选择线程安全的 Map 结构作为基础研究方向。然后基于 Java 对象占存的原理，以原生 HashMap 为实际案例，详细分析了其内存实际占用的分布，并比对了多种常见的用于缓存的内存结构的优劣势。



在进一步优化的时候，针对不同类型的数据可以进行选择不同的编码方式，并以两个实际的缓存压缩方案为例，介绍了如何组合的使用此类编码来有效压缩本地缓存的内存大小。



综上所述，虽然存储在服务器内存中的缓存成本较高，但是若合理的进行数据选型并采用合适的优化方法，则可以达到使用少量的内存缓存大规模的数据，以较低的成本大幅提高应用的响应性能的目的。

五、参考资料

- Java 对象布局：https://www.jianshu.com/p/91e398d5d17c
- fastutil 官网：https://fastutil.di.unimi.it/
- Guava github：https://github.com/google/guava
- SparseArray 文档：https://developer.android.com/reference/android/util/SparseArray

研发效能

浅谈携程大住宿研发效能提升实践

【作者简介】Mia ，携程高级项目经理，负责酒店 Devops 实践，关注 Devops/敏捷等领域。YY，携程敏捷教练，负责团队敏捷转型，研发效能提升实践，关注 Agile、Devops、研发效能等领域。

一、前言

管理大师彼得·德鲁克在《卓有成效的管理者》一书中简明扼要地指出：“效率是‘以正确的方式做事’，效能则是‘做正确的事’。效率和效能不应偏废，我们希望同时提高效率和效能，但若效率与效能无法兼得时，我们首先应着眼于效能的提升”。携程大住宿研发效能提升的指导思想就是基于做正确的事展开，并以“持续快速，高质量的交付有效价值”作为研发效能改进的核心目标。通过持续不断的改进探索，让团队思考更加有效，工作更加高效。

在落地研发效能提升的过程中，我们遇到了很多的挑战，总结下来核心的现象有以下四种：

1. 目标不一致，导致协作低效：大住宿拥有 36 个规模大小不一的敏捷团队。有小型的 10 以下的特性团队，也有 50 人以上的全功能敏捷团队。各团队相对独立又存在无法规避的协作关系。当 A 团队的目标依赖 B 团队的支持，就会存在取舍和协同。当 AB 团队目标不对齐时，先完成自身目标还是支持对方完成目标的过程会增加非常多额外的协作沟通成本。

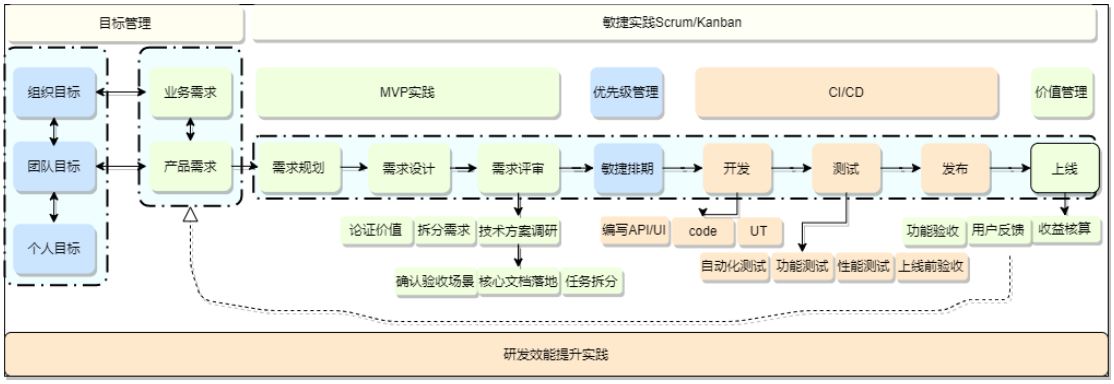
2. 视角割裂，产生无效价值：产品只负责产出需求，开发只管任务完成，最终交付验收发现不是想要的功能。这是大住宿在敏捷转型前遇到最频繁的问题。团队成员视角割裂导致各角色只关注于自己熟悉的领域，而忽略目标价值的交付，最终会产生非必要的浪费。

3. 基建薄弱，导致额外成本增加：有一种误会是只要转型敏捷研发效率就能 10 倍数提升。实践发现，基建的薄弱在一定程度上反而增加团队的负担。比如为了持续频繁的发布，自动化测试的缺失带来额外的人工回归成本；比如代码质量不可靠导致测试频繁的返工等，在一定程度上不仅影响了团队交付效率，还导致了用户满意度的下降。

4. 度量困难，缺少客观衡量数据：大住宿的敏捷转型试点，从一块物理白板，一堆便签，几只油性笔开始。缺少电子信息的沉淀，需要完成度量的费力度和成本非常的高。当时为了收集度量的数据，需要人工记录过程信息，然后通过 Excel 梳理整合，再进行分析处理。人为的记录和分析让数据缺失一定的客观性，无法很好的衡量团队的改进效果，也无法有效引导团队改进方向。

为了改善以上问题，我们从想好、做好、做快这几个维度齐头并进，持续优化，深度耕耘：

- 使用 OKR 工作法拉通产研，深度协作；
- 使用 MVP 实践，围绕价值交付；
- 通过深度敏捷实践，打造敏捷企业文化；
- 通过 DevOps 实践，支撑团队快速交付。



二、OKR 工作法-上下同欲、对齐目标

明确一致的目标是组织内各个部门和全体成员的合作基础，共同的目标是组织建立和存在的客观基础，是完善和发展组织的客观依据，也是为组织创造更大价值的必备因素。OKR 工作法（Objectives&KeyResult，目标与关键结果）是一种企业、团队、员工个人目标设定与沟通的最佳实践与工具，是通过结果去衡量过程的方法与实践。同时，OKR 还是一种能够促进员工与团队协同工作的思维模式。大住宿 OKR 工作法的落地推进，有效的促进了团队成员间的紧密协作，同时也迎来了更多的挑战：

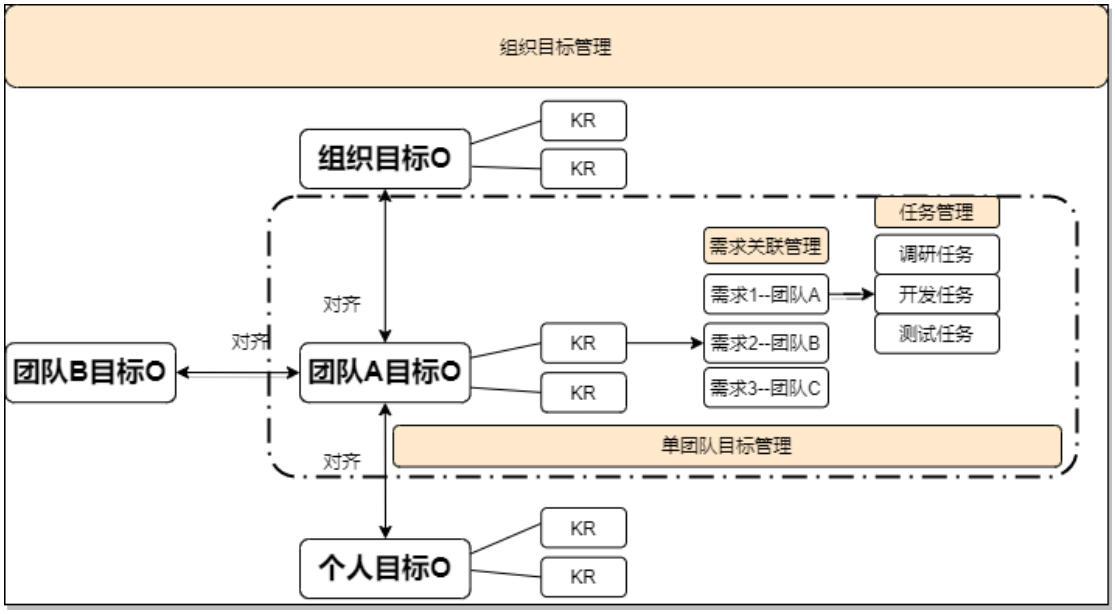
- 如何让整个组织的力量都聚焦在重要事项上，助力战略落地；
- 如何管理组织内的目标横向对齐，消除“部门墙”的障碍，协作更高效；
- 如何透明化组织、团队的目标，暴露重复、多余、无价的任务，节省成本。

面对挑战，大住宿正在持续不断的探索改进中：

1. 推行产研一体，聚焦整体价值交付。以敏捷团队为单位，团队的 PO/TO 与团队共创价值，将每个人的工作与团队目标联系起来。以季度为周期进行规划复盘，月度 review 进度和风险的节奏实施落地。无论是技术还是业务的需求，都聚焦到价值的交付上，团队内部形成良性平衡。

2. 试点部门级别产研一体的季度 OKR 复盘活动。为了更好的达到上下和左右对齐目标，提高协作效率，大住宿从今年 Q1 开始试行部门级产研一体的季度规划和复盘活动。各团队会前准备好复盘材料；会上回顾复盘材料并进行讨论、反馈和建议；会后根据会议内容形成下一季度的 OKR 调整内容和建议。通过活动让大家看到各部门、岗位等相关方的相互依赖关系，明确自己的价值定位、实现团队间的紧密高效协作。从而打破筒仓效应，最大程度整合组织资源。

3. 借助 IDEV 目标管理工具更有效的透明 OKR。IDEV 是公司提供的统一产品研发管理平台，大住宿在去年接入 IDEV 后，不仅提高了产品研发过程的透明性，也率先实现了需求数字化管理。结合实践管理发现需求目标的明确，可以更好的支撑需求的交付。经过沟通和设计，IDEV 平台开发目标管理功能来支持团队的数字化目标管理。通过每个需求关联专属的 KR 对齐目标，并使用关联功能管理依赖团队间的需求。工具支撑的信息透明让团队更高效的彼此对齐，相互支撑，保证了团队步调一致，从而完成最终目标的实现。



三、MVP 实践-共识价值，杜绝浪费

O 代表一种追求和方向，KR 是衡量目标达成的关键结果。为了更好地支持 KR 的达成，团队统一使用 MVP 思维。在规定的时间盒内选取最合适的需求，并用最低的成本，最快的速度，向用户交付产品的主要功能及特色信息，并通过及早的接触用户，获取客户反馈和市场验证来改进产品，迭代升级，以避免做无效需求。

为了更好的落地 MVP 实践，大住宿主要采取了以下 2 个措施:

1. 合理拆分需求，降低试错成本。需求拆分越小，需求越容易理解，改动成本越低，缺陷暴露越早，价值流动越快，也能更早的交付给用户，提前得到反馈。但如果需求拆分的过小，分批开发也会带来测试和发布的成本增加。如何通过合理的拆分需求，降低试错成本?

大住宿研发效能改进计划实施中首先对产研需求进行了规范化的治理，共同约定 IDEV 上创建的每一个需求都是最小维度的可独立交付，可独立验收且可独立衡量价值维度。由于产研视角上的差异会产生不合理的拆分需求，研发团队如果无脑的接受产品拆分，会缺失对需求整体性的认知，也会面临技术实现相互冲突，还可能会对代码架构造成影响。在规范化需求后大住宿又进一步培训加强产研团队共同拆分需求机制落地。

2. MVP 思维贯穿需求整个生命周期。MVP 在实际实践中容易陷入一个误区，做完一个 MVP 就没有后续。大住宿在 MVP 实践中提倡将思想贯彻到产品的整个生命周期当中。上线的 MVP 及时的验证并基于反馈快速的调整寻找下一个方向，迭代循环，最终达成目标。敏捷团队在需求评审会上共识第一次价值，然后在需求上线后及时的验收，进行第二次价值同步。针对没有达到目标需求，快速调研分析后会尽快在最近的迭代周期内安排再次上线验证。整个团队均始终围绕价值持续交付。

四、敏捷实践-敏捷升级，助力效能

敏捷是研发效能提升的又一助力工具。敏捷开发是一种应对快速变化需求的软件开发模式，核心是小步快跑，快速迭代。

大住宿从 2014 年开始推行敏捷转型，敏捷让团队实现价值驱动管理。传统开发模式除了瀑布接力开发外，还有一个是任务驱动管理。任务驱动管理模式下，客户第一次看到实现的功能可能是在验收阶段，这时候发生需求变化或功能新增都会让开发团队的返工成本变得无法预估。还可能为了赶进度，牺牲掉质量。而敏捷开发模式帮助团队重心放在实现对客户有价值的需求上，让团队关注真正有价值的东西。

大住宿的敏捷转型是从 Scrum 开始试点，研发团队从只关注怎么实现需求到共同关注优先要实现哪些需求，如何更快的实现。但一支高效能的敏捷团队，不仅需要高有效的执行落地能力还需要持续不断的改进能力。缺失任何一种能力，都只会让敏捷停留在“伪敏捷”上。

酒店研发在转型路上，也常会因为执行落地不到位而遭遇一些低效的情况：

- 站会变成汇报会议，只有进度同步没有阻塞反馈。
- 回顾会无人说话，事不关己或变成批斗大会。
- 计划会上需求方案还未确认清楚就开始迭代开发，迭代过程中反复确认，沟通成本增加，工作效率低下。

针对以上问题我们做了如下的改进措施来帮助团队提高执行，持续改进。

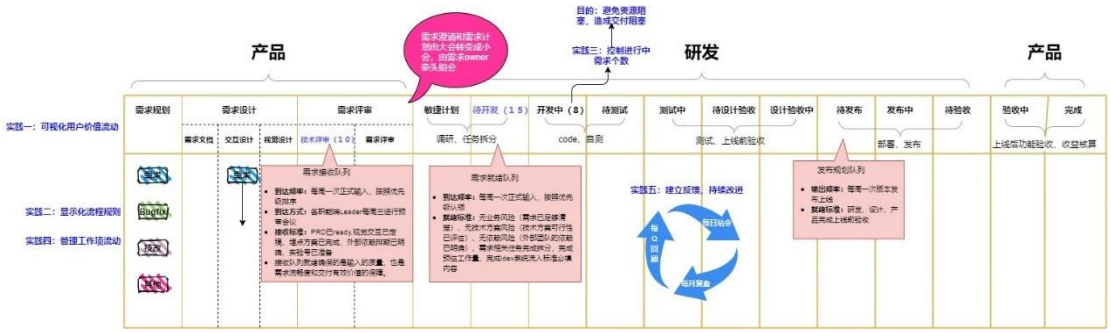
- 增加敏捷培训，邀请团队成员参与到敏捷管理活动中，从实操活动中加强团队成员对于敏捷中每个角色，每个会议的深层理解。
- 明确团队各阶段的完成定义并督促落地执行到位。
- 针对性的开展主题回顾会议，邀请相关干系人共同参与，保持频繁的反馈，持续改进。

会议名称	会议目的	会议前置任务	会议产出	会议时间	会议参与人
需求说明会	知道要做什么	1、带有优先级的需求list 2、PRD上传SVN 3、团队成员提前阅读PRD 4、视觉稿已ready	1、本次Sprint团队可用资源 2、需求工作量 3、需再次确认的内容	1h—2h	PO、SM、Team
计划会	知道要怎么做	1、任务已分配 2、需求已调研 3、外部依赖信息已明确	SprintBacklog—本次Sprint计划	0.5h-1h	PO(可选)、SM、Team
每日站会	制定24小时计划	1、我昨天做了什么 2、我今天要做什么 3、我需要什么协助	1、当日计划以及需要支持的内容 2、计划replan	5m-15m	PO(可选)、SM、Team
演示会	检查增量并调整	1、需要演示的测试数据准备	1、需要改善的产品建议清单	30m-1h	PO、SM、Team
回顾会	为下一个Sprint做改进	1、问题收集(好的持续发扬，不好的准备现象说明)	1、改进意见的落地方案	0.5h-1h	PO、SM、Team

早期的 Scrum 团队更多的关注在软件过程中的活动，而忽略了开发过程中的各种等待时长。Kanban 方法的加入帮助酒店团队看清各种等待不增值的环节。通过 Kanban 方法拉通产品、设计、交互、开发、测试、BI 等各职能各环节的价值流动，并通过 IDEV 需求管理平台实现上下游价值的流动可视化。

改进前团队的关注重心从“敏捷排期”阶段到“待验收”上线阶段。

改进后团队的关注重心从“需求规划”阶段开始到“完成”阶段的整个产品生命周期。



Scrum 和 Kanban 都是帮助团队尽早交付和持续改进的过程方法，方法各有千秋，合适的才是最好的。只有不断的实践，不断的总结，不断的调整，才能真正意义上帮助团队提升。

酒店研发在方法的选择上，也是基于团队自身情况进行决策，比如：

- 有版本限制的团队，采用 Scrum，节奏感可以帮助团队提高协作效率。
- 创新型业务，关注快速交付的团队，采用 Kanban，重点聚焦需求价值流动和及时反馈。
- 单周交付的团队，采用了 Scrum+Kanban 混合方式，有效平衡速度和节奏要求。

Scrum	Kanban
实践核心：化繁为简	实践核心：可视化价值流
定义团队角色：Scrum Master、PO、Team	无特殊规则
定义迭代，固定时间盒概念（两周迭代）	限制WIP（work in progress）
Sprint开始后建议不允许新增需求	只要生产力允许，即可新增需求
尽早交付价值	
持续改进	

八年的敏捷文化熏陶，大住宿大部分的敏捷团队已从“守”的阶段进入“破”和“离”的阶段。

1. 守， 团队能按照 scrum 的流程去实施敏捷，如团队中有三个角色(PO\SM\Team)，团队按照四会（站会，计划会，评审会，回顾会）开展工作等等。

2. 破， 团队能根据自身的状况，去突破敏捷原有的部分规则，去到更高的层次，比如根据敏捷的价值观去增加其它的一些东西，例如增加 TO 的角色、增加 code-review 会议等。

3. 离， 团队的成员已经非常熟悉敏捷的流程和规范，对敏捷的价值观驾轻就熟。团队根据自身状况制定相关的实践，比如 PO/TO 共创团队 OKR 等。

敏捷实践的升级让端到端的产品、开发、利益相关人更顺滑的聚合在一起，采用合作共赢的协作方式帮助团队价值最大化。

五、DevOps 实践-提升质量，加速交付

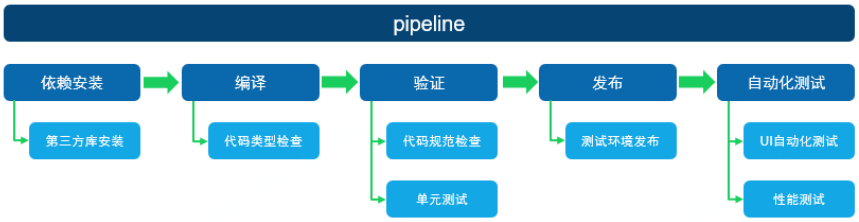
除了采用目标对齐，共识价值，高效的敏捷实践等改进措施，想要达到持续频繁的交付还需要持续集成持续发布能力的支撑。DevOps 强调通过一系列手段来实现既快又稳的工作流程，使每个想法（比如一个新的软件功能，一个功能增强请求或者一个 bug 修复）在从开发到生产环境部署的整个流程中，都能不断地为用户带来价值。CI/CD 作为 DevOps 的重要组成部分，核心价值便是效能与质量，一方面将整个软件研发流程自动化，降低人力成本，另一方面提供了相应的质量检查与测试工具，以期建立一个完整的质量度量体系。

酒店研发引入公司 CI/CD 解决方案，建立完善的准备环境/测试/资源构建/镜像构建一整个流程的链路，使它可帮助项目以更快的速度和更高的质量来交付。

以大住宿某前端研发团队的流水线为例，团队从以下三个目标出发:

- 代码效能
- 产品功能
- 产品性能

通过设置代码规范检查，单元测试、UI 测试、性能测试等任务来提升自动化覆盖率，提升集成效率，强化整体代码质量，提前发现问题，最终实现加快交付频率的目标。并通过采集流水线数据，可视化项目流水线执行概况、近期质量趋势，帮助团队用数据思考，利用数据，持续提升效率。



检查目标	检查任务	依赖任务	检查工具	检查卡口	步骤
/	安装第三方依赖	/	npm、yarn等	/	依赖安装
/	代码编译	安装第三方依赖	Typescript等	/	编译
代码效能	代码类型检查	代码编译	Typescript等	错误<=0	验证
代码效能	代码规范	代码编译	Eslint等	错误<=0	验证
产品功能	单元测试	代码编译	Jest等	错误<=0 覆盖率>=90	验证
/	发布测试环境	/	Mcd等	/	发布
产品功能	UI自动化测试	发布测试环境	Airtest、puppeteer等	错误<=0	自动化测试
产品性能	性能测试	发布测试环境	Lighthouse等	TTI>=上次检查结果	自动化测试

小结：OKR 工作法保障团队方向正确；MVP 实践帮助团队聚焦目标价值；敏捷实践专注快速交付价值，拥抱变化；DevOps 助力快速交付，强化自动化能力。四大措施持续改进，最终达到研发效能提升的目的：持续快速，高质量地向用户交付产品。

六、如何衡量研发效能得到了提升？

管理大师彼得·德鲁克还说没有度量就没有管理。度量最重要的目的是洞察出问题，进行指导改进，并衡量改进的效果。数字化时代的到来，很多企业已具备自动采集效能数据以实现度量所需的各种实时数据报表。大住宿在去年接入公司统一产品研发管理平台 IDEV 后，不仅提高了产品研发过程的透明性，也率先实现了需求数字化管理。

大住宿借助大量的客观数据从目标、价值、质量、效率这 4 个维度的进行分析找到团队的痛点，并引导团队做真正能解决问题的行为来持续改善。

（1）核心目标占比

核心目标价值的占比帮助团队对齐目标和资源整合。我们通过目标管理工具，规范需求与目

标的关联，再通过度量单位时间内围绕目标的交付需求占比来反映团队的目标对齐度。试点实践中遇到最大的问题是数据的失真。数据的准确与团队关联目标的规范息息相关，需要通过对团队进行不断的培训和宣导来帮助团队养成习惯，以此保障数据的准确性。

（2）需求价值指数

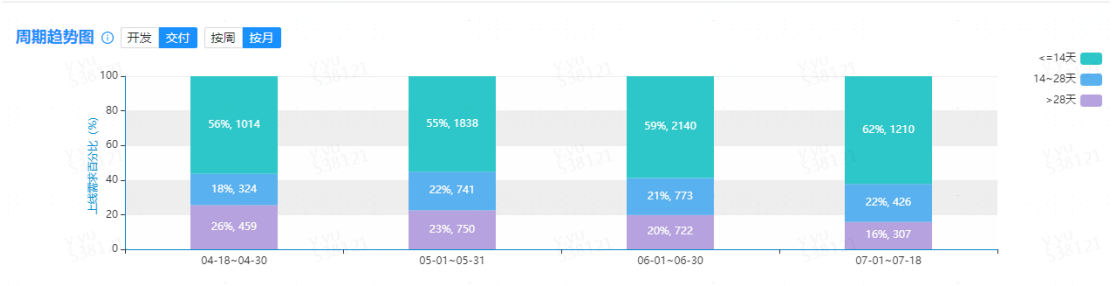
需求是价值的承载体现，假设交付需求均具有价值，那么交付需求的数量越多，代表交付的价值越多。但单以需求个数无法很好的反映团队的交付价值。每个需求的规模和价值大小不一。比如单位时间内可能只交付了一个收益很高的需求，并不能说明团队的产出变少。团队需求价值指数从更客观的维度衡量团队在单位时间内是否产出高价值的内容，以此杜绝高成本低收益的投入。需求价值指数由团队负责的需求个数、人员数、预估价值、实际价值、需求价值正态分布情况等综合评估得出 。

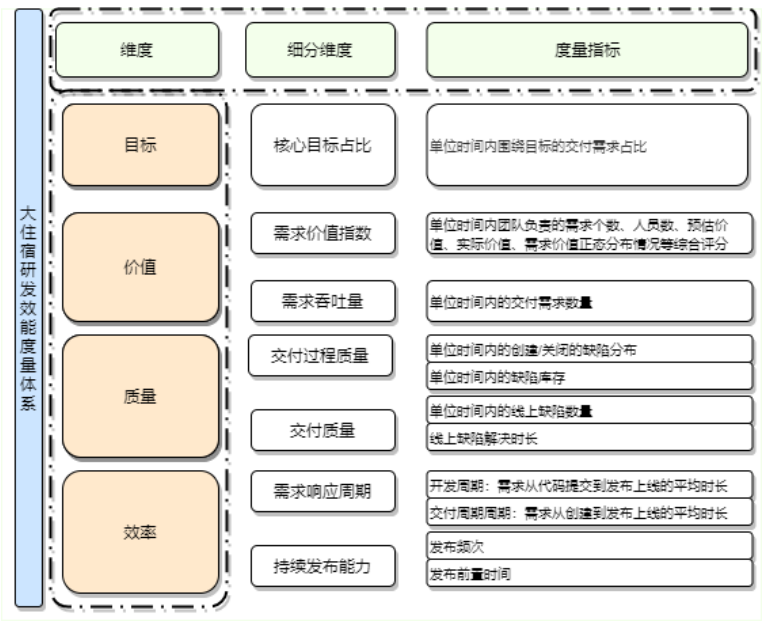
（3）交付质量

研发交付质量是指用户感受到的质量，可以理解为线上用户保障的缺陷。影响交付质量的一个重要因素就是交付过程质量。大住宿主要以单位时间内的缺陷数量趋势来衡量团队交付质量。为了降低缺陷数量，研发团队通过质量内建、提前验收等各种方法来前置保障交付过程质量。并通过分析线上以及过程缺陷，进行归因改进。从自动化，Mock 工具、开发自测等各个方面着手落实改进措施，持续提升交付质量。

（4）响应能力

需求的响应周期和团队持续发布的能力体现团队的持续和快速。交付周期指对用户需求、业务机会的响应速度。酒店研发采用从创建需求开始，到需求上线所经历的平均时长来度量交付周期；通过开始 code 到发布上线所经历的平均时长来度量开发周期；通过单位时间内的有效发布次数来衡量团队对外响应和价值的流动速度。经过一段时间的优化改进，大住宿 2 周内交付的需求占比呈稳定提升趋势。





七、总结

我们可以通过各种措施来提升改进，但研发效能的提升没有“银弹”，研发效能的提升没有最好，只有更好。需要我们从目标、价值、质量、效率每一个领域都进行深入地挖掘和思考，共同努力把持续改进的焦点从局部资源效率转向价值流动效率，以此保证全局和系统的持续优化。

- OKR 工作法：上下同欲、对齐目标
- MVP 实践：共识价值，消灭浪费
- 敏捷实践：敏捷升级，助力效能
- DevOps 实践：提升质量，加速交付

大住宿依然在探寻更好的效能提升方法的路上，就像敏捷宣言中提到的“我们一直在实践中探寻更好的软件开发方法，身体力行的同时也帮助他人。”也希望本篇浅浅的实践总结可以帮助到对研发效能有期待有困惑的你。

开源 | 携程机票 BDD UI Testing 框架 – Flybirds

【作者简介】Liang，携程资深测开专家，开源项目作者，专注于质量能效，自动化框架、工具平台等。

一、背景

携程机票从 2018 年年中正式引入 BDD，至今已 3 年多，成为内部首选的敏捷开发技术。

Flybirds 是一套基于 BDD 模式的前端 UI 自动化测试框架，提供了一系列开箱即用的工具和完善的文档，现在逐步稳定，成为机票内部首选的 BDD-UI-Testing 测试框架。

二、为什么开源

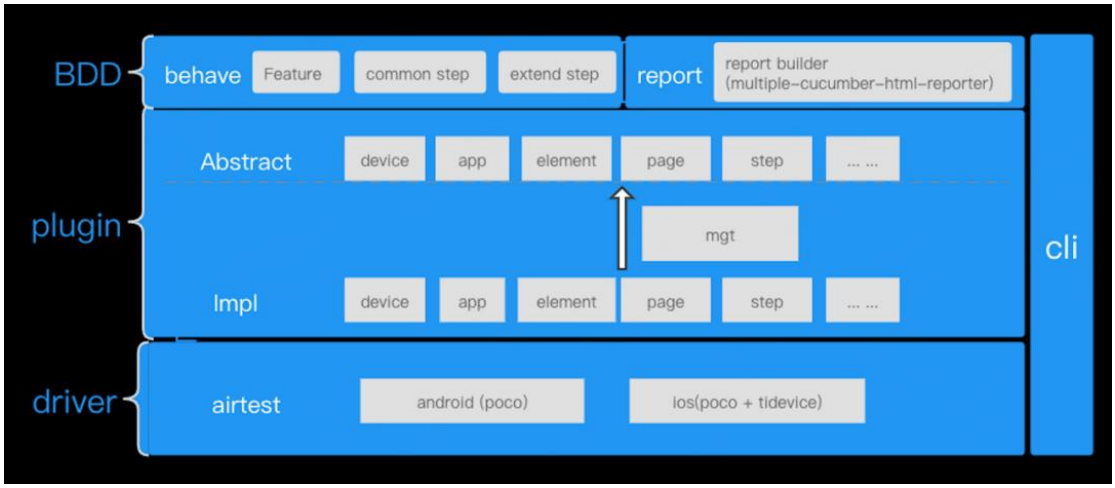
分享我们的 BDD 技术方案

期待业内使用 BDD 技术的同行通过开源社区与我们进行更深入的交流

本文将从特性介绍、环境搭建、使用帮助、自定义扩展、持续集成、发版计划这几个方面对框架进行介绍。

三、Flybirds

- 基于 Behave，实现 BDD 中“自然语言测试用例文档”和“自动化测试代码”关联需要用到支持 BDD 工具。
- 基于 Airtest，实现 BDD 中“测试用例能在自动化测试平台上执行”需要用到 UI 自动化测试框架。
- 基于 Multiple-cucumber-html-reporter，实现可视化的测试报告。



四、特性

使用 Flybirds 你能够完成大部分的手机端自动化操作，以下是一些帮助入门的特性描述：

- 基于 BDD 模式，类自然语言语法
- 支持自动化 APP 操作、表单提交、UI 元素校验、键盘输入、Deeplink 跳转等
- 默认支持英文、中文两种语言，支持更多语言扩展
- 插件式设计，支持用户自定义自动化扩展
- 提供 cli 脚手架，快速搭建项目
- 提供 html 报告

五、环境搭建

5.1 使用 pip 安装 flybirds 框架，过程中会自动安装所需的依赖包

pip3 install flybirds

在 Mac/Linux 系统下，需要手动赋予 adb 可执行权限

- for mac

cd {your_python_path}/site-packages/airtest/core/android/static/adb/mac
chmod +x adb

- for linux

cd {your_python_path}/site-packages/airtest/core/android/static/adb/linux
chmod +x adb

5.2 使用脚手架创建项目

flybirds create

```
Welcome to flybirds cli. Please enter any information to continue.
Please input your project name>>: my_first_project
Please input your test platform?(Android/IOS): Android
Do you want to configure your deviceId now?(this step can be skipped) [y/N]: n
You can configure your deviceId later in the project's flybirds_config.json file.
Do you want to configure your packageName now?(this step can be skipped) [y/N]: n
You can configure your packageName later in the project's flybirds_config.json file.
Cloning into 'my_first_project'...
Processing  [##################################]  100%
Done it! Create Project my_first_project has success!
You can find it at: /Users/test/my_first_project
```

六、快速上手

6.1 运行演示

为了帮助使用，项目创建时，会生成中英文的 Android、iOS 演示 feature，方便用户参考。

features/test/
features/test/android
features/test/android/cn/everything.feature
features/test/android/en/everything.feature
features/test/ios
features/test/ios/cn/everything.feature
features/test/ios/en/everything.feature

以“Android”为例：

- 执行命令 adb devices，检查设备列表中是否包含测试设备
- 开始运行

cd {PATH_TO_PROJECT_FOLDER}
flybirds run -P features/test/android

框架会通过 flybirds_config 中配置的 packagePath 自动下载测试包并安装（请确保手机已经打开”允许安装未知来源“ ）。

运行结果如下：

11 features passed, 0 failed, 0 skipped, 0 untested
23 scenarios passed, 0 failed, 0 skipped, 0 untested
117 steps passed, 0 failed, 0 skipped, 0 undefined, 0 untested
Took 5m21.300s
==========
Multiple Cucumber HTML report generated in:

/Users/test/my_first_project/report/7eb9162a-9d42-4fde-a5d7-d8d4bca7a8d8/index.html
==========

接下来，了解下更多项目细节。

6.2 项目结构

- config：配置文件
- features：测试用例 feature 文件
- pscript：自定义扩展
- report：测试报告

6.3 features 目录

基础目录结构如下：

- test：存放 feature 文件，这些文件使用自然语言编写，最好由软件项目中的非技术业务、产品人员参与者编写。
- steps：存放场景中使用的 step 语句实现，“steps.py”中加载了所有的 step 语句模版。

features/
features/test/
features/test/everything.feature
features/steps/
features/steps/steps.py

复杂些的目录结构参考如下：

features/
features/test/
features/test/list.feature
features/test/buy.feature
features/test/detail.feature
features/steps/
features/steps/steps.py

6.4 feature 文件

feature 文件包含用户动作，行为特征描述及预期结果的文本，行为特征部分使用 Gherkin 语言编写。

feature 文件，也称为功能文件，有两个目的：文档和自动化测试。

以关键字开头（“功能”、“场景”、“场景大纲”、“当”、“而且”、“那么”……），文件中的任何位置都允许使用注释行。

- 功能 (Feature) 是被测试功能的一些合理的描述性标题，由场景组成。他们可以选择有一个描述、一个背景和一组标签。
- 背景 (Background) 由一系列类似于场景的步骤组成。它允许你向功能的场景添加一些上下文。在此功能的每个场景之前执行。
- 场景 (Senario) 标题应该是被测试场景的合理描述性标题，由一系列给定条件的步骤组成。
- 场景大纲 (Senario Outline) 包含功能的详细描述，可以有一组预期条件和结果来配合你的场景步骤。

以下是中文 feature 例子：

```
# language: zh-CN
 功能: 首页机票模块、搜索框
       1. 进入首页后能够展示机票按钮和搜索框
       2. 点击机票按钮后显示特价机票模块
       3. 在搜索框中输入上海，点击搜索后，能够展示上海机票信息

 背景:
     当  启动APP[ctrip.android.view]
     而且 页面渲染完成出现元素[text=首页]

  场景: 验证首页机票按钮功能
     当  存在元素[text=机票]
     那么 点击[text=机票]
     那么 页面渲染完成出现元素[text=特价机票]
     那么 全屏截图
     那么 关闭APP


   场景: 验证首页搜索框功能
     当  开始录屏
     那么 在[text=搜索]中输入[上海]
     而且 点击[label=sbtn]
     那么 页面渲染完成出现元素[text=上海机票]
     那么 结束录屏
     那么 关闭APP
```

以下是英文 feature 例子:

```
# language: en
 Feature: Homepage ticket module, search box
         1. After entering the homepage, the ticket button and search box can be displayed
         2. Click the ticket button to display the special ticket module
         3. Input "Shanghai" in the search box and click search to display the “Shanghai ticket”
information

 Background:
     When start app [ctrip.android.view]
     And  page rendering complete appears element [text=Homepage]

  Scenario: verify the function of the homepage ticket button
     When exist element [text=ticket]
     And  click [text=ticket]
     Then page rendering complete appears element [text=SpecialTicket]
     Then screenshot
     Then close app

   Scenario: verify the function of the homepage search box
     When start record
     Then in [text=search] input [shanghai]
     And  click [label=sbtn]
     Then page rendering complete appears element [text=ShangHaiTicket]
     Then stop record
     Then close app
```

6.5 step 语句模板

以下列出了部分模版 | 更多模版查阅

语句模板	语义
跳转到 []	跳转到指定的url地址
等待 [] 秒	等待一段时间
页面渲染完成出现元素 []	进入新的页面时检查指定元素是否渲染完成
点击 []	点击指定属性的元素
在 [] 中输入 []	在指定选择器中输入字符串
向 [] 查找 [] 的元素	向指定方向查找指定属性的元素
全屏向 [] 滑动 []	全屏向指定方向滑动指定距离
存在 [] 的文案	检查页面中存在指定的字符串
存在 [] 的元素	检查页面中存在指定属性的元素
元素 [] 消失	检查页面中指定属性的元素在指定时间内消失
全屏截图	保存当前屏幕图像
开始录屏	开始录制视频
结束录屏	结束录制视频
安装APP []	安装APP
启动APP []	启动APP
-----	-----

6.6 Hooks

用户可在以下文件中定义 hooks:

pscript/dsl/step/hook.py

- before_step(context,step), after_step(context, step)

  在每个步骤(step)之前和之后运行

- before_scenario(context,scenario), after_scenario(context, scenario)

在每个场景(senario)之前和之后运行

- before_feature(context,feature), after_feature(context, feature)

在每个功能文件(feature)之前和之后运行

- before_tag(context,tag), after_tag(context, tag)

在用给定名称标记(tag)的部分之前和之后运行

- before_all(context), after_all(context)

在所有执行之前和之后运行

6.7 标签(Tags)

可以使用 tag 标记不同的场景，方便有选择性的运行。

下面是一个例子：

运行有特定 tag 的场景，多个用逗号隔开：

flybirds run -T tag1,tag2

‘-’开头表示运行不包含某 tag 的场景：

flybirds run -T -tag

七、运行前检查

7.1 请确保配置的测试设备能够正常连接

- Android: 执行命令 adb devices ， 检查设备列表中是否包含测试设备

- iOS：以 tidevice 库举例，执行命令 tidevice list，检查设备列表中是否包含测试设备

（1）Android 设备连接 Q&A

- 请先安装手机对应品牌的官方驱动，确保能使用电脑对手机进行 USB 调试；
- 确保已经打开了手机中的"开发者选项"，并且打开"开发者选项"内的"允许 USB 调试"；
- 部分手机需要打开"允许模拟位置"、"允许通过 USB 安装应用"；
- 关闭电脑上已经安装的手机助手软件，能避免绝大多数问题，请务必在任务管理器中手工结束手机助手进程。

（2）iOS 设备连接 Q&A

- 请先准备一台 macOS ，使用 xcode 部署 iOS-Tagent 成功后，能够在 mac 或 windows 器上连接到 iOS 手机。请点击链接下载项目代码到本地进行部署；
- mac 环境通过 Homebrew 安装 iproxy： brew install libimobiledevice；
- windows 环境安装 itunes。

7.2 下载安装测试包

- Android：

框架会通过 config 中配置的 packagePath 自动下载测试包并安装（请确保手机已经打开"允许安装未知来源“ ）。也可手动下载安装：下载地址

- iOS：

a. 请手动下载演示 APP 进行安装：下载地址
b. 开启 wdaproxy：shell tidevice --udid
$udid wdaproxy -B $web_driver_angnt_bundle_id -p $port

八、运行参数

在终端输入以下内容来查看 flybirds 运行项目时支持的操作

flybirds run --help

- run

执行 features 目录下所有的 feature 文件

- --path, -P

指定需要执行的 feature 集合，可以是目录，也可以指定到具体 feature 文件，默认是 ‘features’ 目录.

示例:

flybirds run -P ./features/test/demo.feature

- --tag, -T

运行有特定 tag 的场景，多个用逗号隔开，‘-’开头表示不运行包含此 tag 的场景

flybirds run -T tag1,tag2,-tag3,tag4

- --format, -F

指定生成测试结果的格式，默认是 json.

示例:

#默认
flybirds run --format=json

九、配置参数

提供了丰富的配置项 | 帮助文档

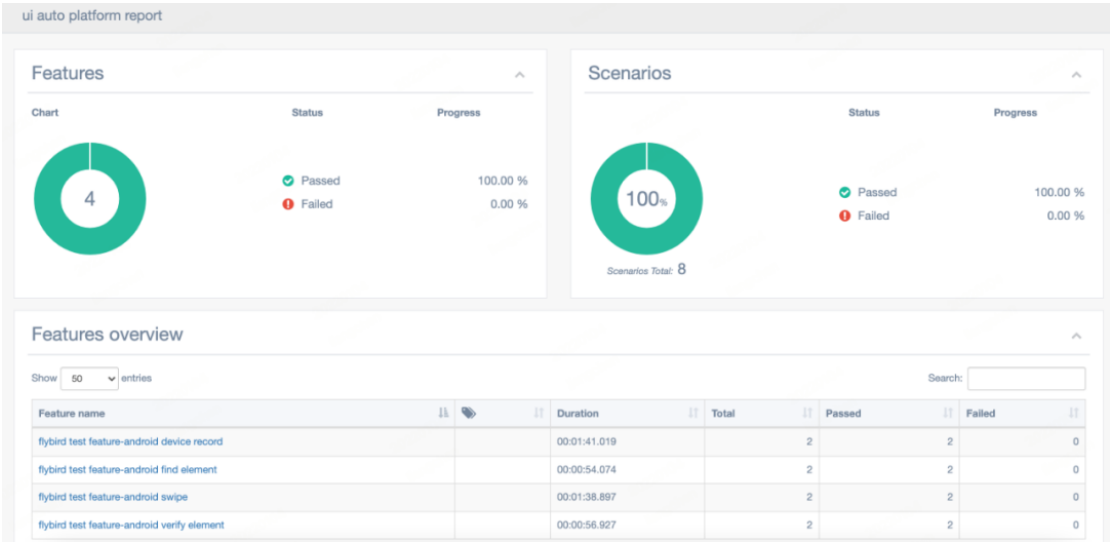
必须配置项：deviceId packageName。

连接 iOS 设备时，需要额外配置 webDriverAgent、platform。

```
{
  "app_info": {
    "packageName": "ctrip.android.view",
    "packagePath": "",
    "overwriteInstallation": false,
  },
  "device_info": {
    "deviceId": "127.0.0.1:8200",
    "platform": "Android",
    "webDriverAgent": "com.fd.test.WebDriverAgentLib.xctrunner"
  },
  "flow_behave": {
    "failScreenRecord": true,
    "scenarioScreenRecordTime": 120,
    "failRerun": false,
    "maxFailRerunCount": 1,
  },
  "frame_info": {
    "waitEleTimeout": 35,
    "waitEleDisappear": 10,
    "clickVerifyTimeout": 15,
    "useSwipeDuration": false,
    "swipeDuration": 6,
    "usePocoInput": true,
    "afterInputWait": 1,
    "useSearchSwipeDuration": false,
    "searchSwipeDuration": 1,
    "swipeSearchCount": 5,
    "swipeSearchDistance": 0.3,
    "pageRenderTimeout": 65,
    "appStartTime": 6,
    "swipeReadyTime": 3,
    "verifyPosNotChangeCount": 5,
    "screenRecordTime": 90,
    "useSnap": true
  },
}
```

十、报告(report)

报告包含汇总 Summary 和功能（feature）、场景（senario）的执行结果，对于失败的场景(senario)，报告中会展示当时的屏幕图像和视频，下面是一个例子。



十一、自定义 step 语句模板

在编写 Feature 的过程中，可能会遇到提供的公共语句不能满足自身项目的需求，需要自定

义语句。比如：需要对接某个内部工具 API，此时需要用到自定义语句功能。

自定义语句功能会用到 python，如果你不了解这门编程语言，也不必要太担心，因为只会使用到最基础的 python 语法，这并不会太难。

使用方法

- 进入项目目录"psscript/dsl/steps"
- 新建.py 文件来编写自定义语句
- 在 feature/steps/steps.py 中 import 该.py 文件

示例代码如下：

```python
# -*- coding: utf-8 -*-

from behave import step


@step("Create new milestone[{param1}]and[{param2}]")
def create_milestone_with_title(context, param1, param2):
    param1 = param1.strip()
    param2 = param2.strip()
    print(f'I create milestone with title {param1} and {param2}!')
```

对于团队内部通用的自定义功能，可以考虑创建一个 extend package，flybirds 支持动态加载，package 命名包含“-flybirds-plugin”即可。

十二、自定义框架扩展

理论上 BDD-UI-Testing 可以适用在所有端，比如：APP、Web、小程序。

框架的插件式设计模式，保留了良好的扩展，当前版本只开放了 APP 端支持，未来会逐步开放更多，下面是两个例子供大家参考。

（1）增加 web 端扩展

- 在 plugins.default 下添加自己的 web 包。
- 添加 web 对应的实现。比如 page.py,element.py，如果需要增加其他的插件实现类，只需要在 GlobalContext 类中添加对应的名称。
- 在 plugin.event 下添加自己的 web 包。
- 在 event.web 包下重写或者新增类，比如重写 run 事件可以在 plugin.event.web 下面添加"run.py"，具体实现逻辑可参照已有的 run.py。
- 在项目配置文件"flybirdes_config.json"配置 device_info.platform 值为"web"。

（2）修改当前 APP 端扩展

- 可通过配置"plugin_info.json"对已有的 plugins 进行修改（只支持修改不支持新增），比如你希望对 plugins 下面 ios.app 进行修改：

a. 可以在本地创建一个自己 app.py
b. 在 plugin_info.json 对应平台中添加如下配置：

```
"app": {
    "path": "{local_path}/app.py",
    "ns": "app.plugin"
}
```

{local_path} 为本地路径，"ns"为包名，注意包名的唯一性。

十三、其他语种支持

flybirds 可以支持 40 几种语言，在以下文件中增加公共方法的语言配置即可。

flybirds/core/dsl/globalization/i18n.py

示例代码如下：

```
step_language = {
    "zh-CN": {
        "install app[{param}]": ["安装APP[{param}]"],
        "delete app[{param}]": ["删除APP[{param}]"],
        "start app[{param}]": ["启动APP[{param}]"],
        "restart app": ["重启App"],
        "close app": ["关闭App"],
        "init device[{param}]": ["设备初始化[{param}]"],
        "connect device[{param}]": ["连接设备[{param}]"],
        "start recording timeout[{param}]": ["开始录屏超时[{param}]"],
        "start record": ["开始录屏"],
        "stop record": ["结束录屏"],
        "go to url[{param}]": ["跳转到[{param}]", "跳转页面到[{param}]"],
        "return to previous page": ["返回上一页"],
        "go to home page": ["回到首页"],
        "logout": ["退出登陆", "退出登录"],
        "wait[{param}]seconds": ["等待[{param}]秒"],
        "screenshot": ["全屏截图"],
        "click[{param}]": ["点击[{param}]"],
        "click text[{param}]": ["点击文案[{param}]"],
        "click position[{x},{y}]": ["点击屏幕位置[{x},{y}]"],
        "in[{param1}]input[{param2}]": ["在[{param1}]中输入[{param2}]"],
        "slide to {param1} distance[{param2}]": ["全屏向{param1}滑动[{param2}]"],
        "exist text[{param}]": ["存在[{param}]的文案"],
        "not exist text[{param}]": ["不存在[{param}]的文案"],
        "text[{param}]disappear": ["文案[{param}]消失"],
        "exist[{param}]element": ["存在[{param}]的元素"],
        "not exist element[{param}]": ["不存在[{param}]的元素"],
        "element[{param}]disappear": ["元素[{param}]消失"],
        "page rendering complete appears element[{param}]": [
            "页面渲染完成出现元素[{param}]"],
        "existing element[{param}]": ["存在元素[{param}]"],
    },
}
```

十四、持续集成

cli 提供的命令行执行模式，可以非常方便加入各种持续集成工具。

以 Jenkins 为例：

```
# Inside the jenkins shell command
cd {PATH_TO_PROJECT_FOLDER}
# Run
flybirds run -P ./features/test/everything.feature
cp -R reports $WORKSPACE
```

十五、发版计划

我们将按照 SemVer 版本控制规范进行发版。逐步新增功能和代码优化，非常欢迎加入到我们的共建计划中，在 Github 上提出宝贵建议，以及在使用时遇到的一切问题，我们也会对此每周进行一次小版本的迭代。你也可以在这里给我们精神支持，点上一颗 Star。

- Github 地址：

https://github.com/ctripcorp/flybirds

- PyPI 地址：

https://pypi.org/project/flybirds

- 贡献

1）Fork 仓库
2）创建分支 (git checkout -b my-new-feature)
3）提交修改 (git commit -am 'Add some feature')
4）推送 (git push origin my-new-feature)
5）创建 PR

- 欢迎在 github issues 区提问

- 支持邮箱：flybirds_support@trip.com

提前在开发阶段暴露代码问题，携程 Alchemy 代码质量平台

【作者简介】Lyan，携程资深后端开发工程师，负责自动化测试框架及平台类工具开发，关注 Devops、研发效能领域。

一、背景

随着敏捷开发，DevOps 开发模式的流行，代码质量分析作为研发质量保证体系的重要组成部分，不仅能有效的降低因频繁迭代带来的故障风险，而且对整个工程团队的效能提升有着巨大的价值。

携程很久以前就已经开始进行 DevOps 的建设，通过 Gitlab CI/CD 在开发提交代码触发的流水线 pipeline 中引入静态扫描、单元测试、集成测试等流程，在开发过程中打造了一套闭环的代码质量保障体系。其中，在静态代码分析阶段引入了 SonarQube，并且通过对原有 SonarQube 代码规范库中的规范进行筛选和扩展，形成了自己的代码规范库。但是在实际应用过程中，我们发现仍然有一些问题需要优化解决：

- 在开发过程中，代码规范只能通过开发人员自我约束，缺少统一的平台对各应用代码的潜在风险问题统一进行分析，且问题难以定位到开发人员进行治理。

- 代码单元测试通过率和代码覆盖率都很高，但仍然存在一些在单元测试阶段应被发现的问题未暴露出来，导致上线后出现 bug，单元测试用例的质量缺乏有效性及可靠性保证。

- 随着项目的发展，开发人员为了避免影响已有功能，在开发过程中大量复制粘贴，导致出现很多难以维护的重复代码，且程序逻辑结构过于复杂，修改逻辑牵一发而动全身，可维护性差。

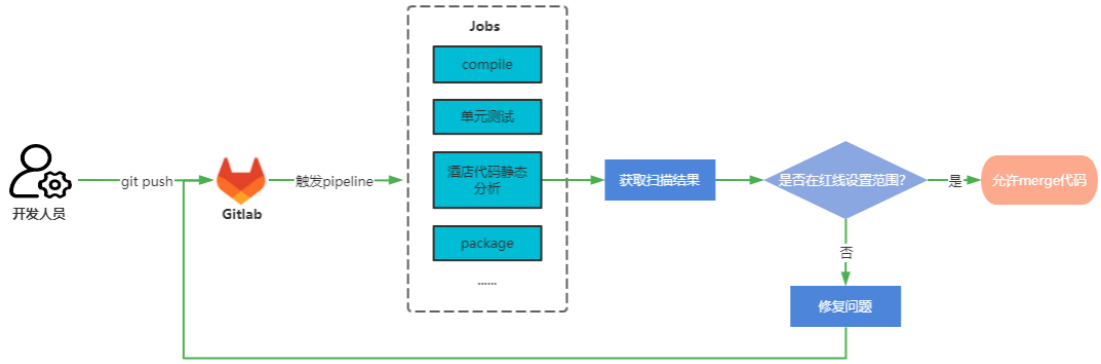
- 代码中充斥着大量的 sql 拼接，以及一些不规范的写法导致潜在的问题，需要对此类代码进行治理。

二、平台介绍

Alchemy 平台是一个代码质量分析平台，提供 Infer 分析、代码分析、自定义扫描、代码搜索等功能，其中代码质量分析内容包含代码行，sonar 问题， infer 问题，UT 规则，重复代码以及圈复杂度等。用户可以根据自己的需求在平台上进行扫描项配置，并查看应用的代码质量分析结果。

为了及时获得对提交代码变更的质量反馈，作为 DevOps 中重要的一环，Alchemy 平台与 Gitlab CI/CD 相结合，将静态代码分析提前至开发提交或合并代码阶段。开发人员提交代码至 Gitlab，触发流水线相关任务执行，任务执行完成之后可以对某些指标（如增量代码引入的空指针）设置红线进行卡点，如果指标在指定范围内，允许合并代码并发布，如果指标超

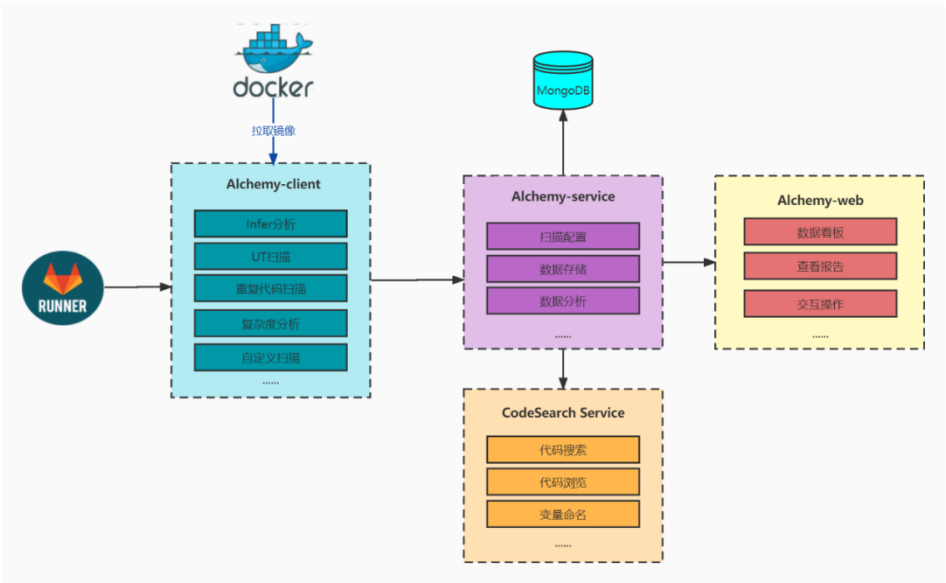
过了红线设置范围，则不允许合并代码，开发人员修复问题后再次提交代码进行流水线的集成发布。扫描分析结果可以在 Gitlab 或者 Alchemy 平台上展示，帮助开发人员在快速迭代的同时保证代码质量。



三、系统架构

Alchemy 平台包含 Alchemy-client、Alchemy-service 和 Alchemy-web。其中，Alchemy-client 为扫描脚本，包含 Infer 分析，UT 扫描，重复代码扫描、自定义扫描等功能，集成到 Docker 镜像中，Alchemy-service 提供数据存储、分析等后台服务，且依赖代码搜索服务 CodeSearch-Service 实现代码搜索功能，Alchemy-web 负责页面交互。

开发人员提交代码，触发 Gitlab CI/CD 中静态代码分析 job 在 GitRunner 中执行，执行时先从 Docker 仓库下载镜像，启动容器后执行 Alchemy-client 脚本，脚本会根据平台配置来执行相应的扫描任务，扫描完成后，将结果上传至 Alchemy-service，存储到 mongodb 数据库，最终在前端页面展示分析结果。

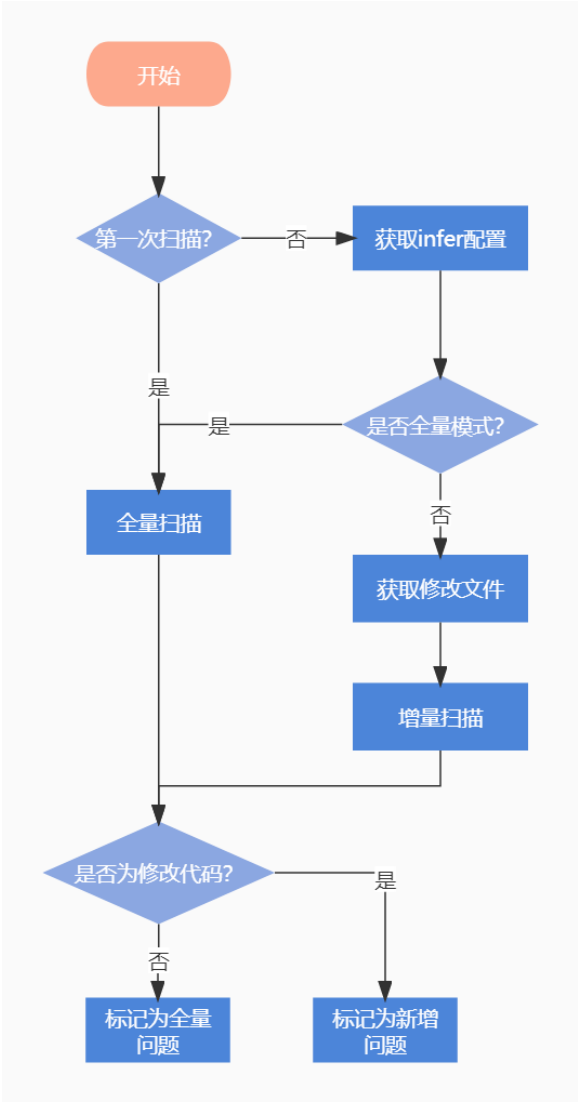


四、功能

4.1 Infer 分析

Infer 是 Facebook 出品的一个静态分析工具，可以分析 Java、Objective-c 或者 C 代码，报告潜在的问题，包括空指针，资源泄漏等。Alchemy 平台将 Infer 引入代码静态分析阶段，目前已支持全量和增量分析两种模式。

全量模式需分析应用仓库中的所有代码，能分析出所有代码引入的潜在问题，对于代码量较大的应用，由于需要分析所有代码文件，扫描时间比较长，在一定程度上影响开发发布进度，且对未修改的代码进行了非必要的重复分析，在代码修改量较少的情况下造成资源浪费。因此，我们尝试加入缓存机制，并引入了增量分析模式，增量模式需要获取本次提交修改的文件，在分析阶段只针对这些改动文件进行分析，能大大节省分析时间。Infer 分析流程如下：



在分析过程中，首先判断是否为第一次分析，如果没有分析历史记录，则系统默认采用全量模式，否则需判断 Infer 扫描配置，若配置为全量模式，则分析此代码工程的全部文件，若配置为增量模式，需获取此次提交修改的文件列表，编译过程完成之后，在分析阶段指定文件列表进行分析。获取到分析出的问题列表后，判断问题所在的行是否为修改行，如果是，

则记录为本次修改导致的新增问题，否则为历史遗留的全量问题。

在实际应用中，针对封装的判空方法，通过添加@TrueOnNull 或@FalseOnNull 注解，可识别对象的判空操作。但对于第三方包的判空方法，如 CollectionUtils.isEmpty()， 由于未添加注解，即使添加判空方法，仍会被误识别为空引用。因此，Alchemy 平台加入了忽略操作，针对此类问题进行二次确认，避免重复误判。

```
object `basicRoomTypeInfoList` last assigned on line 499 could be null and is dereferenced at line 518.
516.
517.                List<ChannelManagerBasicRoomTypeInfoDto> basicRoomTypeInfoDtoList = new ArrayList<>();
518. >              for (BasicRoomTypeInfoDto c : basicRoomTypeInfoList) {
519.                    if (!CollectionUtils.isEmpty(c.getRoomInfos())) {
520.                        ChannelManagerBasicRoomTypeInfoDto channelManagerBasicRoomTypeInfoDto = channelManagerConvert.convertCMBasicRoomTypeInfoDto(c);
```

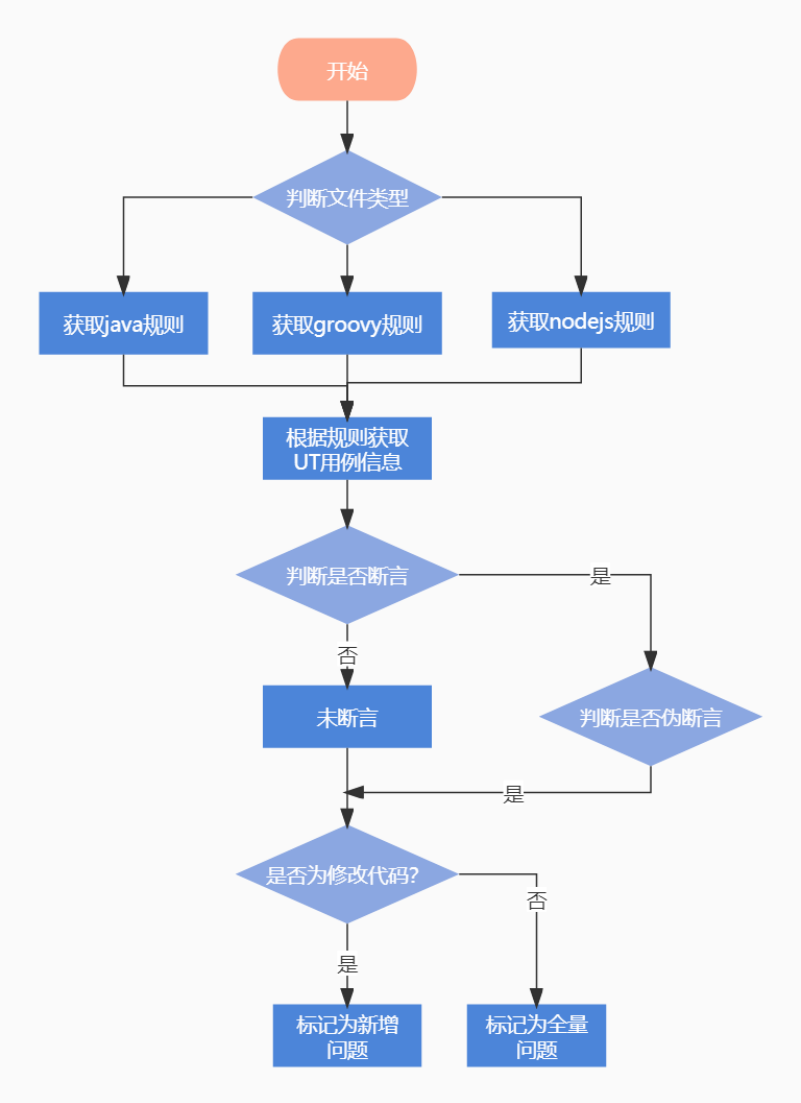
4.2 UT 规则扫描

单元测试是 DevOps 流程中一个非常重要的环节，我们可以利用通过率和代码覆盖率等指标来衡量单元测试用例的完整程度，却很难保证用例的有效性。阿里巴巴 java 开发手册规定，单元测试不允许使用 System.out 来进行人肉验证，必须使用断言 assert 来验证。

在实际的开发过程中，开发人员把主要的时间用在写业务逻辑代码上，在编写单元测试用例时，往往容易忽略对结果的验证，虽然通过率和代码覆盖率很高，但上线后仍然出现未对接口结果进行验证而导致严重问题的情况。无效的单元测试用例包含以下几种：

- 空函数：函数体为空；
- 空断言：用例中实现了对被测接口的调用逻辑，但未对接口返回结果进行验证；
- 伪断言：用例中使用类似 assertTrue(True)的假断言。

通过扫描空断言、空函数、伪断言等问题，能判断该用例是否对代码逻辑进行必要的验证。Alchemy 平台支持单元测试用例的有效性验证，目前，平台支持 Java、Kotlin、Groovy 和 Nodejs，同时也支持全量和增量 2 种扫描结果，全量结果即为所有测试用例中不满足规则的用例，增量结果为本次提交修改的测试用例中不满足规则的用例。

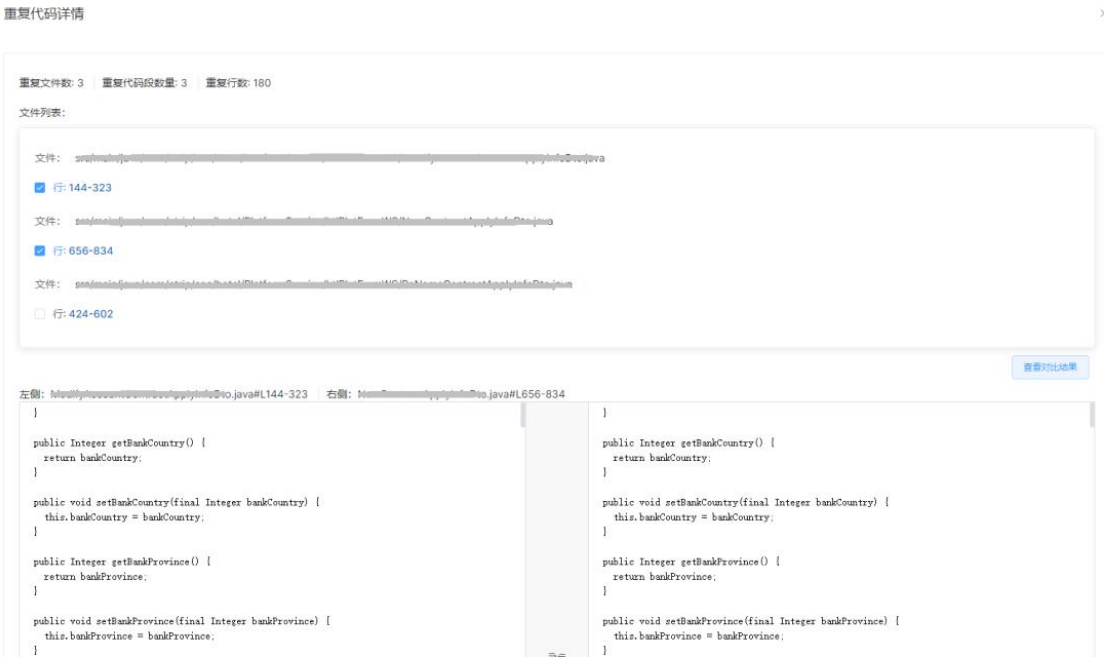


对单个单元测试文件的扫描流程如图。首先根据文件后缀判断语言类型，然后根据不同语言类型规则获取该文件中的用例信息，包含 case 名称、起止行、作者、最近修改时间、函数内容等，针对函数内容，先判断用例是否有断言，如果有，则判断是否为伪断言，如果未断言或者被判为伪断言，还需进一步根据用例的起止行，结合本次提交的改动信息，分析该用例是否为增量改动的用例，如果是则标记为增量问题，最后将结果推送给 gitlab，在合并代码或发布时根据红线配置进行卡点。扫描结果如图所示。

4.3 重复代码扫描

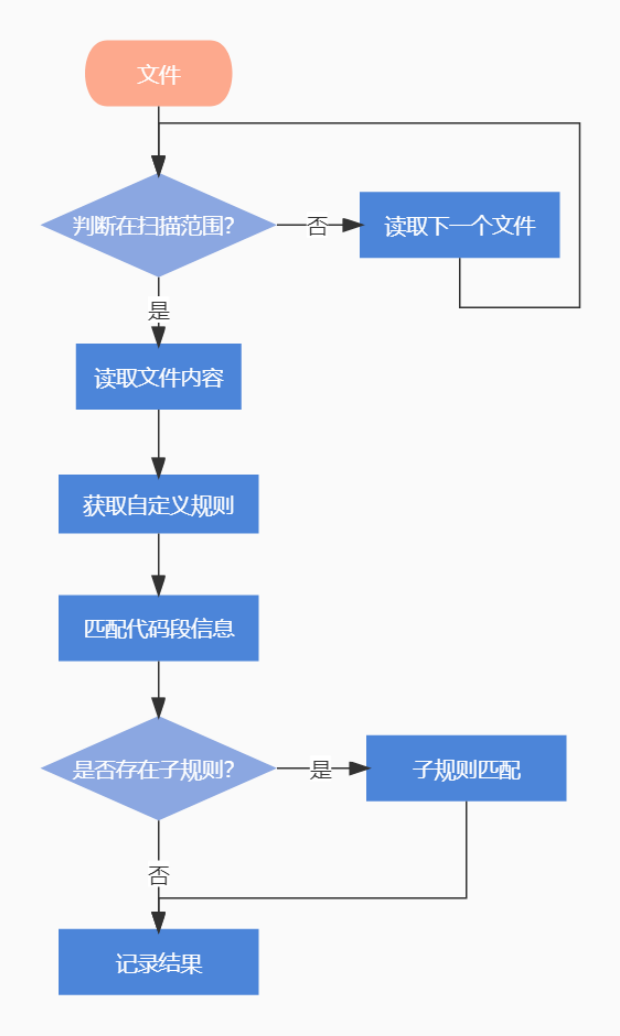
重复代码即为重复或近似的代码，在开发过程中，开发人员为了避免影响现有功能，使用复制粘贴快速完成开发任务，导致出现大量的重复代码。重复代码不仅让代码量大增，造成编译速度慢，而且占用大量存储空间，如果想要修改其中一段代码逻辑，则需要同时修改多个地方，容易遗漏，可维护性差。

当前市面上有很多代码检测工具，如 Simian，PMD-CPD，CloneDR 等，由于在实现算法上有所不同，不同工具所能检测的重复代码类型也不尽相同。我们利用 PMD-CPD 扫描代码仓库，可以检测出单文件或多个文件中除了空格、注释、换行以及变量名以外内容完全一致的代码段信息，这些信息包含文件路径、代码段内容、起止行以及作者信息，详情结果如图所示。



4.4 自定义规则扫描

Alchemy 支持对自定义规则的扫描，通过配置自定义正则表达式和扫描范围，识别代码文件中满足配置规则的代码段，可用于扫描代码中的拼接 SQL，敏感词等，并且可将不合规的代码定位到相关开发人员。

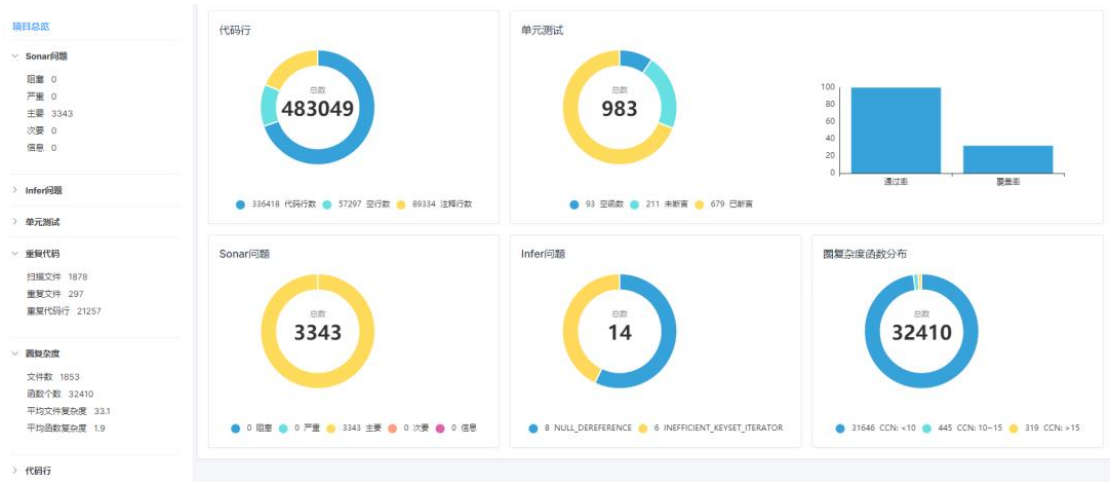


单个文件扫描流程如图，首先判断文件是否在扫描范围内，若不在则直接跳转扫描下一个文件，否则读取文件内容，同时根据文件类型获取对应的自定义规则，匹配满足规则的代码段信息，包含代码段内容、严重程度、起止行、作者等。在某些场景下，需要设置子规则进行二次匹配，比如扫描 update 未指定 where 条件的 sql 语句，可先根据规则找到 update 语句，然后根据子规则判断是否带 where 条件，最终记录二次匹配的结果。

4.5 代码分析

使用不同工具统计的代码质量指标可能分散在不同的平台，对这些指标进行全面分析的过程中难免会有所遗漏，特别是对于未设置发布卡点的指标，开发人员可能并不会关注它们，导致代码存在大量的潜在问题未被分析治理。

Alchemy 代码分析模块可以对代码不同维度的指标进行统计分析，包括代码行、单元测试、infer 问题、Sonar 问题、重复代码、圈复杂度等。用户可以在代码分析页面查看各维度问题分布情况，从而对项目的整体风险指标进行更全面的分析，可以根据问题的严重程度设置优先级进行针对性的治理。



4.6 代码搜索

在开发过程中，对于一些公共操作如中间件的使用方式，开发人员可能需要四处寻找接入文档。Alchemy 提供代码搜索功能，可以帮助开发人员根据关键词来查找收录项目中的代码使用示例，用户可以根据项目仓库、代码语言以及作者等条件进行细分查询。在编码过程中，命名规范是一个容易被忽视的问题，使用 Alchemy 的变量命名功能，用户可以根据不同语言，搜索中英文关键词来获取推荐的规范命名参考，能极大地提高开发效率。

五、结束语

在本文中，我们介绍了 Alchemy 平台提供的代码静态分析，代码探索以及通过与 Gitlab CI/CD 集成带来的持续集成能力，可以在开发阶段暴露出更多的代码潜在问题和风险，并及时反馈给相关人员。目前携程酒店已接入项目 800+，且在开发提交代码和发布阶段将分析的潜在问题接入了卡点流程。在后续的工作中，我们将从以下几个方向进行进一步的优化：

- 在代码分析层面支持更多语言；
- 开发 IDE 插件，在编码阶段实时扫描代码；
- 继续深挖代码风险指标，并引入评估机制。

数据库

携程酒店慢查询治理之路

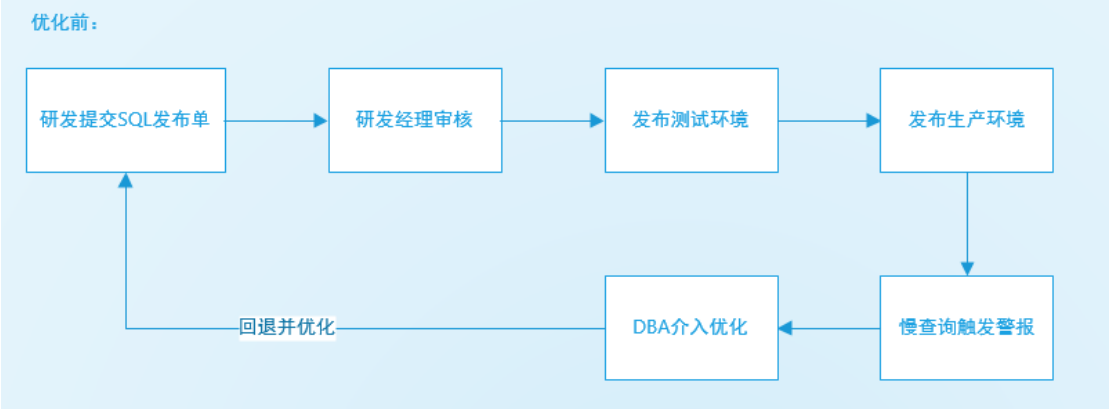
【作者简介】xuqi，携程资深数据库工程师，关注 MySQL、分布式数据库的优化、运维；潘达鸣，携程资深数据库工程师，关注数据库性能优化、高可用性领域；康男，携程数据库专家，关注数据库性能调优领域。

一、背景

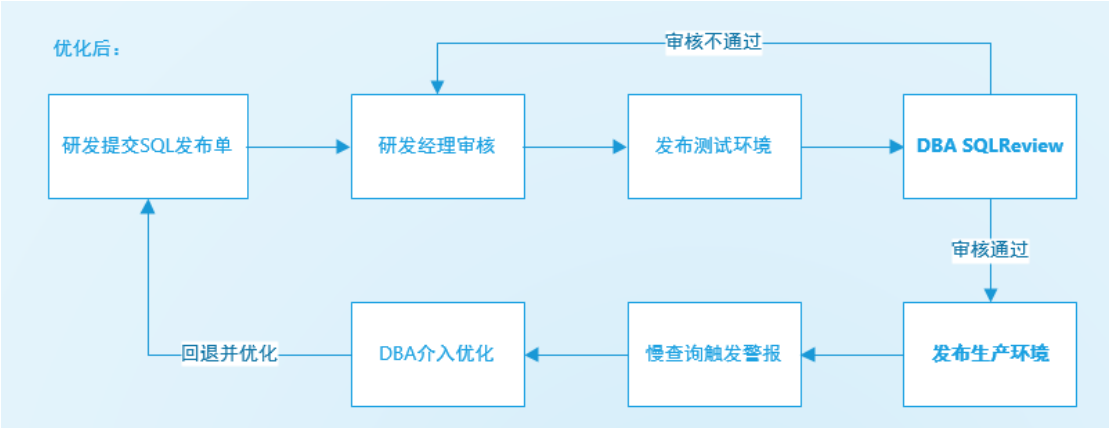
慢查询指的是数据库中查询时间超过了指定的阈值的 SQL，这类 SQL 通常伴随着执行时间长、服务器资源占用高、业务响应慢等负面影响。随着携程酒店业务的不断扩张，再加上大量的 SQLServer 转 MySQL 项目的推进，慢查询的数量正在飞速增长，每日的报警量也居高不下，因此慢查询的治理优化已经是刻不容缓，此文主要针对 MySQL。

二、慢查询治理实践

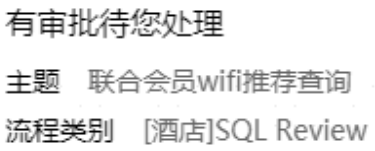
2.1 SQL 上线流程优化



之前的流程发布比较快捷，但是随着质量差的 SQL 发布\迁移得越来越多，告警和回退数量也随之变多，综合下来，数据库风险方面不容乐观，该流程需要优化。



和旧流程相比，新增了一个 SQLReview 的环节，将潜在的慢查询提前筛选出来优化，确保上线的 SQL 质量，在此流程保障下，所有上线到生产的 SQL 性能都能在 DBA 评估后的可控范围内，在研发提交审核后，会收到审批的事件单。



携程目前是存在自动化 review 审核的平台，但是由于酒店业务场景比较复杂，研发对于 SQL 的理解水平层次不齐，平台给出的建议并不能做到面面俱到，因此还没有被广泛使用于流程中，仅作为一个参考。

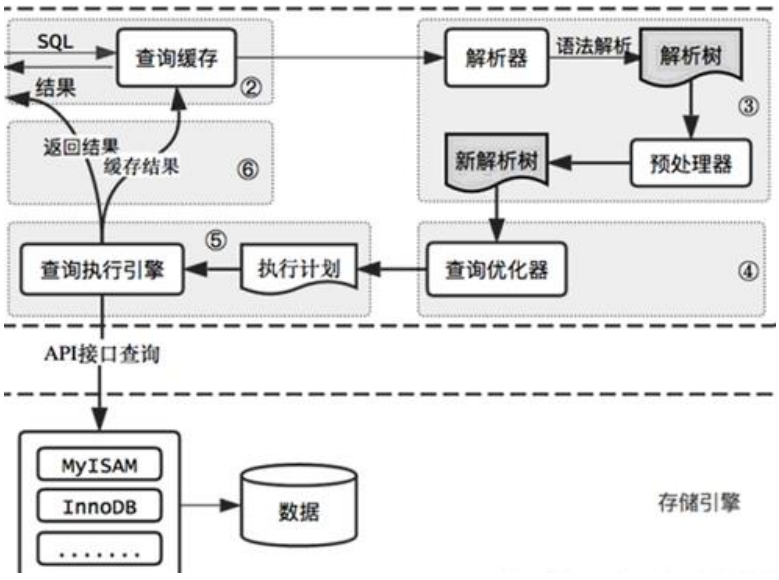
2.2 理解查询语句

要优化慢查询，首先要知道慢查询是如何产生的，执行计划是怎么样的，最后考虑如何去优化查询。

（1）流程及查询优化器

一条 sql 的执行主要分成如图几个步骤：

- SQL 语法的缓存查询（QC）
- 语法解析（SQL 的编写、关键字的语法之类）
- 生成执行计划
- 执行查询
- 输出结果



通常慢查询都发生在“执行查询”这步，读懂查询计划，可以有效地帮助我们分析 SQL 性能差的原因。

（2）执行计划

在 SQL 前面加上 EXPLAIN，就可以查看执行计划，计划以“表”的形式展示：

```
+----+-------------+-------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
| id | select_type | table | partitions | type  | possible_keys | key     | key_len | ref  | rows | filtered | Extra       |
+----+-------------+-------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | hotel | NULL       | range | PRIMARY       | PRIMARY | 4       | NULL |   41 |   100.00 | Using where |
+----+-------------+-------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

具体字段含义可以参考 MySQL 官方的解释，这里不多赘述。

Table 8.1 EXPLAIN Output Columns

Column	JSON Name	Meaning
id	select_id	The SELECT identifier
select_type	None	The SELECT type
table	table_name	The table for the output row
partitions	partitions	The matching partitions
type	access_type	The join type
possible_keys	possible_keys	The possible indexes to choose
key	key	The index actually chosen
key_len	key_length	The length of the chosen key
ref	ref	The columns compared to the index
rows	rows	Estimate of rows to be examined
filtered	filtered	Percentage of rows filtered by table condition
Extra	None	Additional information

2.3 优化慢查询

通过执行计划就可以定位到问题点，通常可以分为这几种常见的原因。

(1) 索引层面

- 索引缺失

这个查询由于缺少 name 字段索引，产生了全表扫描：

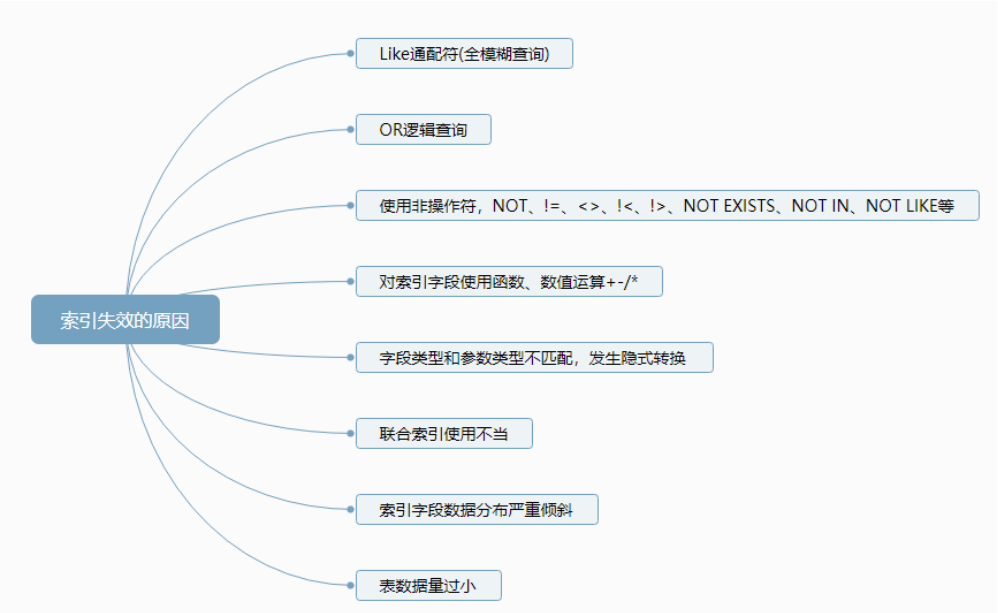
select * from hotel where name='xc';

```
mysql> explain select * from hotel where name='xc';
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | hotel | NULL       | ALL  | NULL          | NULL | NULL    | NULL |    2 |    50.00 | Using where |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

补上索引之后，提示使用到了索引。

```
mysql> alter table hotel add index idx_name(name);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> explain select * from hotel where name='xc';
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref   | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------------+
|  1 | SIMPLE      | hotel | NULL       | ref  | idx_name      | idx_name | 802     | const |    1 |   100.00 | Using index |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

- 索引失效



如图所示，索引失效的大致原因可以分为八类，这些场景通过查看执行计划都会发现产生 type=ALL 或者 type=index 的全表扫描。

- Like、or、非操作符、函数

explain select * from hotel where name like '%酒店%';
explain select * from hotel where name like '%酒店%'or Bookable='T';
explain select * from hotel where name <>'酒店';
explain select * from hotel where substring(name,1,2)='酒店';

```
mysql> explain select * from hotel where name like '%酒店%';
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | hotel | NULL       | ALL  | NULL          | NULL | NULL    | NULL |   41 |    11.11 | Using where |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

- 参数类型不匹配

create table t1 (
col1 varchar(3) primary key
)engine=innodb default charset=utf8mb4;

```
mysql> mysql> explain select * from t1 where col1 =111;
+----+-------------+-------+------------+-------+---------------+---------+---------+------+------+----------+--------------------------+
| id | select_type | table | partitions | type  | possible_keys | key     | key_len | ref  | rows | filtered | Extra                    |
+----+-------------+-------+------------+-------+---------------+---------+---------+------+------+----------+--------------------------+
|  1 | SIMPLE      | t1    | NULL       | index | PRIMARY       | PRIMARY | 11      | NULL |    6 |    16.67 | Using where; Using index |
+----+-------------+-------+------------+-------+---------------+---------+---------+------+------+----------+--------------------------+
1 row in set, 3 warnings (0.00 sec)

mysql> explain select * from t1 where col1 ='111';
+----+-------------+-------+------------+-------+---------------+---------+---------+-------+------+----------+-------------+
| id | select_type | table | partitions | type  | possible_keys | key     | key_len | ref   | rows | filtered | Extra       |
+----+-------------+-------+------------+-------+---------------+---------+---------+-------+------+----------+-------------+
|  1 | SIMPLE      | t1    | NULL       | const | PRIMARY       | PRIMARY | 11      | const |    1 |   100.00 | Using index |
+----+-------------+-------+------------+-------+---------------+---------+---------+-------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)

mysql>
```

t1 表的 col1 为 varchar 类型，但是参数传入的是数值类型，结果产生了隐形转换，索引失效导致 type=index 的全表扫描。

- 联合索引

Where 条件不符合“最左匹配原则”，则索引会失效。

alter table hotel add index idx_hotelid_name_isdel(hotelid,name,status);

以下条件均可以命中联合索引：

explain select * from hotel where hotelid=10000 and name='ctrip' and status='T';
explain select * from hotel where hotelid=10000 and name='ctrip';
explain select * from hotel where hotelid=10000;

```
mysql> alter table hotel add index idx_hotelid_name_isdel(hotelid,name,status);
Query OK, 0 rows affected (0.09 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> explain  select * from hotel where hotelid=10000 and name='ctrip' and status='T';
+----+-------------+-------+------------+------+-----------------------+-----------------------+---------+-------------------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys         | key                   | key_len | ref               | rows | filtered | Extra |
+----+-------------+-------+------------+------+-----------------------+-----------------------+---------+-------------------+------+----------+-------+
|  1 | SIMPLE      | hotel | NULL       | ref  | idx_hotelid_name_isdel | idx_hotelid_name_isdel | 818     | const,const,const |    1 |   100.00 | NULL  |
+----+-------------+-------+------------+------+-----------------------+-----------------------+---------+-------------------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)

mysql> explain  select * from hotel where hotelid=10000 and name='ctrip';
+----+-------------+-------+------------+------+-----------------------+-----------------------+---------+-------------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys         | key                   | key_len | ref         | rows | filtered | Extra |
+----+-------------+-------+------------+------+-----------------------+-----------------------+---------+-------------+------+----------+-------+
|  1 | SIMPLE      | hotel | NULL       | ref  | idx_hotelid_name_isdel | idx_hotelid_name_isdel | 807     | const,const |    1 |   100.00 | NULL  |
+----+-------------+-------+------------+------+-----------------------+-----------------------+---------+-------------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)

mysql> explain  select * from hotel where hotelid=10000;
+----+-------------+-------+------------+------+-----------------------+-----------------------+---------+-------+-------+----------+-------+
| id | select_type | table | partitions | type | possible_keys         | key                   | key_len | ref   | rows  | filtered | Extra |
+----+-------------+-------+------------+------+-----------------------+-----------------------+---------+-------+-------+----------+-------+
|  1 | SIMPLE      | hotel | NULL       | ref  | idx_hotelid_name_isdel | idx_hotelid_name_isdel | 5       | const | 15177 |   100.00 | NULL  |
+----+-------------+-------+------------+------+-----------------------+-----------------------+---------+-------+-------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

但是以下条件无法使用到联合索引：

explain select * from hotel where name='ctrip' and status='T';
explain select * from hotel where name='ctrip';
explain select * from hotel where status='T';

```
mysql> explain  select * from hotel where name='ctrip' and status='T';
+----+-------------+-------+------------+------+---------------+------+---------+------+-------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows  | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+------+---------+------+-------+----------+-------------+
|  1 | SIMPLE      | hotel | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 30355 |     1.00 | Using where |
+----+-------------+-------+------------+------+---------------+------+---------+------+-------+----------+-------------+
1 row in set, 1 warning (0.00 sec)

mysql> explain  select * from hotel where name='ctrip'
    -> ;
+----+-------------+-------+------------+------+---------------+------+---------+------+-------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows  | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+------+---------+------+-------+----------+-------------+
|  1 | SIMPLE      | hotel | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 30355 |    10.00 | Using where |
+----+-------------+-------+------------+------+---------------+------+---------+------+-------+----------+-------------+
1 row in set, 1 warning (0.00 sec)

mysql> explain  select * from hotel where status='T';
+----+-------------+-------+------------+------+---------------+------+---------+------+-------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows  | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+------+---------+------+-------+----------+-------------+
|  1 | SIMPLE      | hotel | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 30355 |    10.00 | Using where |
+----+-------------+-------+------------+------+---------------+------+---------+------+-------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

- 数据分布和数据量

索引字段的数据分布不均匀，表数据量过小的情况下，MYSQL 查询优化器可能认为返回的数据量本身就很多，通过索引扫描并不能减少多少开销，此时选择全表扫描的权重会提高很多。

- 查询不带 where 条件

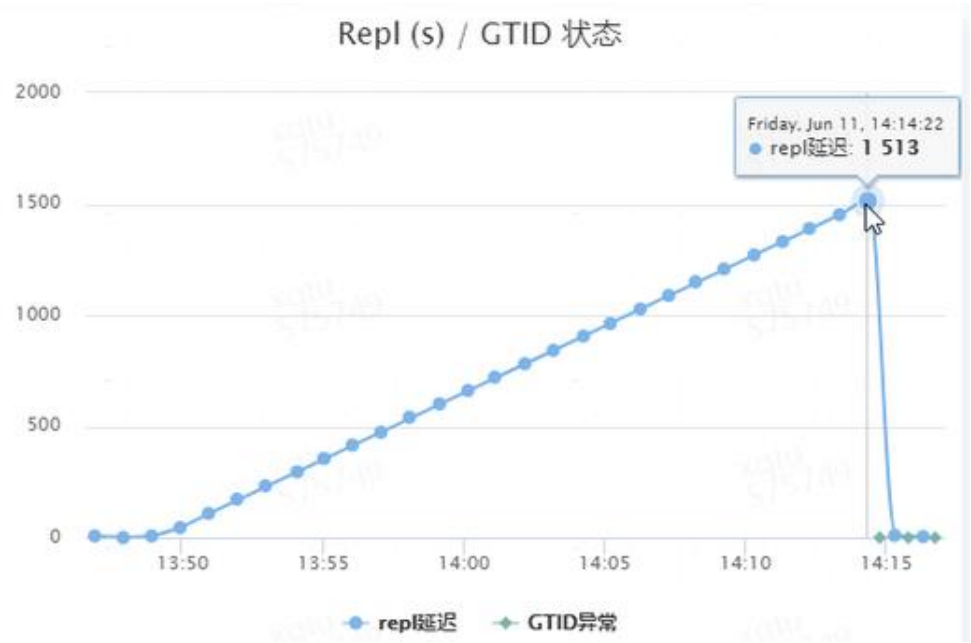
不带 where 条件直接查询\修改全表是很危险的操作，表数据量够大的话，尽量拆分成多批次操作。

```
mysql> explain select * from hotel;
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
|  1 | SIMPLE      | hotel | NULL       | ALL  | NULL          | NULL | NULL    | NULL |   41 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

优化中遇到的案例:

某天发现有一台 DB 服务器 IO 异常，服务器链接开始堆积，引发了大量应用报错.

监控显示此时 repl 延迟已经有 25 分钟，集群几乎处于无高可用状态，非常的危险。



登陆服务器排查后发现有一条全表删除的 SQL 在通过 JOB 系统跑，该表的数据量很大:

-tarpresqls "delete from XXXXXX"

最后紧急 Kill 这条 SQL 后恢复正常，直接在生产删除全表是很危险的操作。

- 强制使用索引

MySQL 中存在 force index()、ignore index（）方式来强制使用/忽略特定的索引。

这种方式可能会导致执行计划选择不到最优的索引，从而导致计划走偏。

```
mysql> explain select * from hotel force index(`idx_bookable`) where name = 'xc';
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | hotel | NULL       | ALL  | NULL          | NULL | NULL    | NULL |   41 |     2.44 | Using where |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

- 性能差索引的 Index Merge

Index merge 方法可以对同一个表使用多个索引分别进行条件扫描，检索多个范围扫描并将结果合并为一个。

```
mysql> explain select status,bookable from hotel where status='F' and bookable='T';
+----+-------------+-------+------------+-------------+-----------------------+-----------------------+---------+------+------+----------+-----------------------------------------------------------------------------------------------+
| id | select_type | table | partitions | type        | possible_keys         | key                   | key_len | ref  | rows | filtered | Extra                                                                                         |
+----+-------------+-------+------------+-------------+-----------------------+-----------------------+---------+------+------+----------+-----------------------------------------------------------------------------------------------+
|  1 | SIMPLE      | hotel | NULL       | index_merge | idx_bookable,idx_status | idx_bookable,idx_status | 11,11   | NULL |    9 |    94.62 | Using intersect(idx_bookable,idx_status); Using where; Using index |
+----+-------------+-------+------------+-------------+-----------------------+-----------------------+---------+------+------+----------+-----------------------------------------------------------------------------------------------+
1 row in set, 1 warning (0.00 sec)
```

但是，当遇到如图 2 个索引字段分布都很差的情况时（status 与 bookable 的区分度都很低），2 个索引的结果集存在大量数据需要 merge，性能就会变得很糟糕。

(2) SQL 频率

- 业务代码 while、for 循环的结束条件不正确，导致模块内产生死循环;
- 业务逻辑本身存在高并发场景，例如秒杀、短期促销活动、直播带货等;
- 通过定时 JOB 循环拉取全量数据，但是循环的并发节奏控制不到位;
- 缓存被击穿、业务代码发布后缓存失效等原因，导致大量请求直接打到了 db。

(3) 写法不规范

- 分页写法

最常见的分页写法就是使用 limit，在分页查询时，我们会在 LIMIT 后面传两个参数，一个是偏移量(offset)，一个是获取的条数(limit)。当偏移量很小时，查询速度很快，但是随着 offset 变大时，查询速度会越来越慢。

MySQL Limit 语法格式：

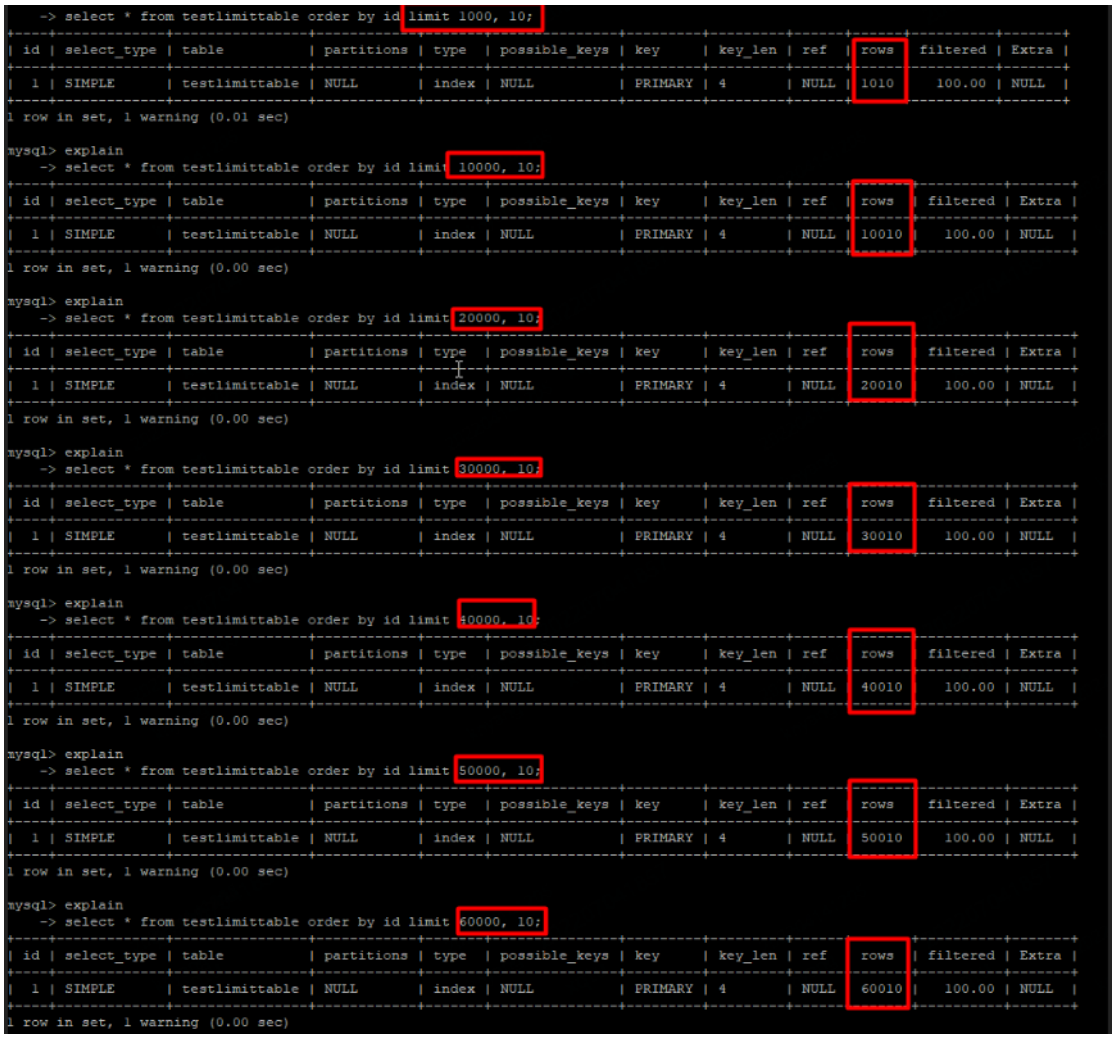
SELECT * FROM table LIMIT [offset,] rows | rows OFFSET offset

例如下列分页查询：

```
mysql> select * from testlimittable order by id limit 0,10;
+----+---------------------+
| id | insertdt            |
+----+---------------------+
|  1 | 2022-04-13 16:44:53 |
|  2 | 2022-04-13 16:44:55 |
|  3 | 2022-04-13 16:44:55 |
|  4 | 2022-04-13 16:44:56 |
|  5 | 2022-04-13 16:44:56 |
|  6 | 2022-04-13 16:44:57 |
|  7 | 2022-04-13 16:44:57 |
|  8 | 2022-04-13 16:44:58 |
|  9 | 2022-04-13 16:44:58 |
| 10 | 2022-04-13 16:44:59 |
+----+---------------------+
10 rows in set (0.00 sec)

mysql> explain select * from testlimittable order by id limit 0,10;
+----+-------------+---------------+------------+-------+---------------+---------+---------+------+------+----------+-------+
| id | select_type | table         | partitions | type  | possible_keys | key     | key_len | ref  | rows | filtered | Extra |
+----+-------------+---------------+------------+-------+---------------+---------+---------+------+------+----------+-------+
|  1 | SIMPLE      | testlimittable | NULL       | index | NULL          | PRIMARY | 4       | NULL |   10 |   100.00 | NULL  |
+----+-------------+---------------+------------+-------+---------------+---------+---------+------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

当 limit 只有 0，10 时，执行还是很快，但是随着 offset 增加，可以看到深度分页的情况下，分页越深，扫描的行数就越多，性能也就越来越差了。

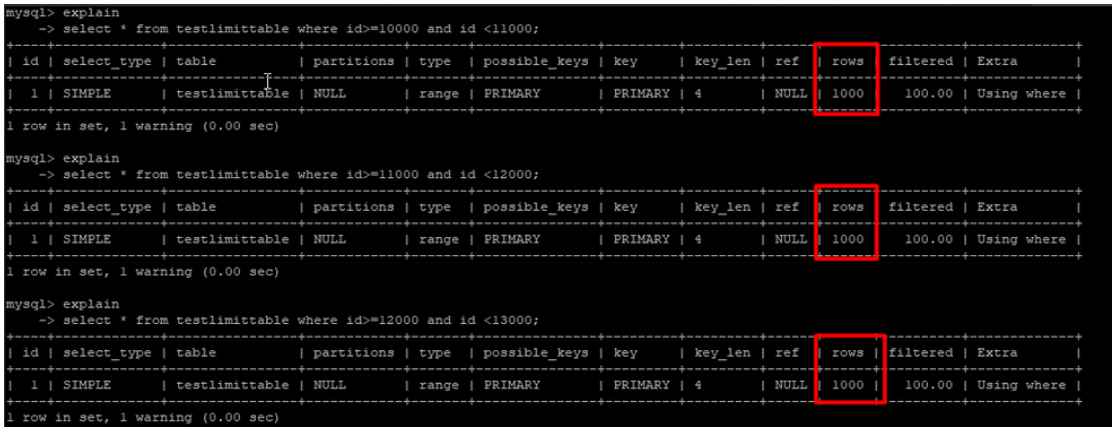
explain select * from testlimittable order by id limit 1000, 10;
explain select * from testlimittable order by id limit 10000, 10;
explain select * from testlimittable order by id limit 20000, 10;
explain select * from testlimittable order by id limit 30000, 10;
explain select * from testlimittable order by id limit 40000, 10;
explain select * from testlimittable order by id limit 50000, 10;
explain select * from testlimittable order by id limit 60000, 10;



```
    -> select * from testlimittable order by id limit 1000, 10;
+----+-------------+---------------+------------+-------+---------------+---------+---------+------+------+----------+-------+
| id | select_type | table         | partitions | type  | possible_keys | key     | key_len | ref  | rows | filtered | Extra |
+----+-------------+---------------+------------+-------+---------------+---------+---------+------+------+----------+-------+
|  1 | SIMPLE      | testlimittable | NULL       | index | NULL          | PRIMARY | 4       | NULL | 1010 |   100.00 | NULL  |
+----+-------------+---------------+------------+-------+---------------+---------+---------+------+------+----------+-------+
1 row in set, 1 warning (0.01 sec)

mysql> explain
    -> select * from testlimittable order by id limit 10000, 10;
+----+-------------+---------------+------------+-------+---------------+---------+---------+------+-------+----------+-------+
| id | select_type | table         | partitions | type  | possible_keys | key     | key_len | ref  | rows  | filtered | Extra |
+----+-------------+---------------+------------+-------+---------------+---------+---------+------+-------+----------+-------+
|  1 | SIMPLE      | testlimittable | NULL       | index | NULL          | PRIMARY | 4       | NULL | 10010 |   100.00 | NULL  |
+----+-------------+---------------+------------+-------+---------------+---------+---------+------+-------+----------+-------+
1 row in set, 1 warning (0.00 sec)

mysql> explain
    -> select * from testlimittable order by id limit 20000, 10;
+----+-------------+---------------+------------+-------+---------------+---------+---------+------+-------+----------+-------+
| id | select_type | table         | partitions | type  | possible_keys | key     | key_len | ref  | rows  | filtered | Extra |
+----+-------------+---------------+------------+-------+---------------+---------+---------+------+-------+----------+-------+
|  1 | SIMPLE      | testlimittable | NULL       | index | NULL          | PRIMARY | 4       | NULL | 20010 |   100.00 | NULL  |
+----+-------------+---------------+------------+-------+---------------+---------+---------+------+-------+----------+-------+
1 row in set, 1 warning (0.00 sec)

mysql> explain
    -> select * from testlimittable order by id limit 30000, 10;
+----+-------------+---------------+------------+-------+---------------+---------+---------+------+-------+----------+-------+
| id | select_type | table         | partitions | type  | possible_keys | key     | key_len | ref  | rows  | filtered | Extra |
+----+-------------+---------------+------------+-------+---------------+---------+---------+------+-------+----------+-------+
|  1 | SIMPLE      | testlimittable | NULL       | index | NULL          | PRIMARY | 4       | NULL | 30010 |   100.00 | NULL  |
+----+-------------+---------------+------------+-------+---------------+---------+---------+------+-------+----------+-------+
1 row in set, 1 warning (0.00 sec)

mysql> explain
    -> select * from testlimittable order by id limit 40000, 10;
+----+-------------+---------------+------------+-------+---------------+---------+---------+------+-------+----------+-------+
| id | select_type | table         | partitions | type  | possible_keys | key     | key_len | ref  | rows  | filtered | Extra |
+----+-------------+---------------+------------+-------+---------------+---------+---------+------+-------+----------+-------+
|  1 | SIMPLE      | testlimittable | NULL       | index | NULL          | PRIMARY | 4       | NULL | 40010 |   100.00 | NULL  |
+----+-------------+---------------+------------+-------+---------------+---------+---------+------+-------+----------+-------+
1 row in set, 1 warning (0.00 sec)

mysql> explain
    -> select * from testlimittable order by id limit 50000, 10;
+----+-------------+---------------+------------+-------+---------------+---------+---------+------+-------+----------+-------+
| id | select_type | table         | partitions | type  | possible_keys | key     | key_len | ref  | rows  | filtered | Extra |
+----+-------------+---------------+------------+-------+---------------+---------+---------+------+-------+----------+-------+
|  1 | SIMPLE      | testlimittable | NULL       | index | NULL          | PRIMARY | 4       | NULL | 50010 |   100.00 | NULL  |
+----+-------------+---------------+------------+-------+---------------+---------+---------+------+-------+----------+-------+
1 row in set, 1 warning (0.00 sec)

mysql> explain
    -> select * from testlimittable order by id limit 60000, 10;
+----+-------------+---------------+------------+-------+---------------+---------+---------+------+-------+----------+-------+
| id | select_type | table         | partitions | type  | possible_keys | key     | key_len | ref  | rows  | filtered | Extra |
+----+-------------+---------------+------------+-------+---------------+---------+---------+------+-------+----------+-------+
|  1 | SIMPLE      | testlimittable | NULL       | index | NULL          | PRIMARY | 4       | NULL | 60010 |   100.00 | NULL  |
+----+-------------+---------------+------------+-------+---------------+---------+---------+------+-------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

*：警惕通过分页写法来实现循环分批的逻辑，limit 深分页实现不了将大量数据拆分成若干小份的效果

分批可以采用分段拉取减少扫描的行数，如果分段拉取不连续的话可以传入上一次拉取最大的值作为下一次的起始值：



```
mysql> explain
    -> select * from testlimittable where id>=10000 and id <11000;
+----+-------------+---------------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
| id | select_type | table         | partitions | type  | possible_keys | key     | key_len | ref  | rows | filtered | Extra       |
+----+-------------+---------------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | testlimittable | NULL       | range | PRIMARY       | PRIMARY | 4       | NULL | 1000 |   100.00 | Using where |
+----+-------------+---------------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)

mysql> explain
    -> select * from testlimittable where id>=11000 and id <12000;
+----+-------------+---------------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
| id | select_type | table         | partitions | type  | possible_keys | key     | key_len | ref  | rows | filtered | Extra       |
+----+-------------+---------------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | testlimittable | NULL       | range | PRIMARY       | PRIMARY | 4       | NULL | 1000 |   100.00 | Using where |
+----+-------------+---------------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)

mysql> explain
    -> select * from testlimittable where id>=12000 and id <13000;
+----+-------------+---------------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
| id | select_type | table         | partitions | type  | possible_keys | key     | key_len | ref  | rows | filtered | Extra       |
+----+-------------+---------------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | testlimittable | NULL       | range | PRIMARY       | PRIMARY | 4       | NULL | 1000 |   100.00 | Using where |
+----+-------------+---------------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

- 最大最小值写法

由于 where 条件的字段数据分布问题，会导致 max 和 min 的查询非常慢：

explain select max(id) from hotel where hotelid=10000 and status='T';

```
mysql> explain  select max(id) from hotel where hotelid=10000 and status='T';
+----+-------------+-------+------------+------+---------------+-------------+---------+-------+-------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key         | key_len | ref   | rows  | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+-------------+---------+-------+-------+----------+-------------+
|  1 | SIMPLE      | hotel | NULL       | ref  | idx_hotelid   | idx_hotelid | 5       | const | 15175 |    10.00 | Using where |
+----+-------------+-------+------------+------+---------------+-------------+---------+-------+-------+----------+-------------+
1 row in set, 1 warning (0.01 sec)
```

由于 hotelid=10000 的数据分布比较多，可以看到扫描数很高：

a. 添加联合索引

alter table hotel add index idx_hotelid_status(hotelid,status);

```
mysql> alter table hotel add index idx_hotelid_status(hotelid,status);
Query OK, 0 rows affected (0.09 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> explain  select max(id) from hotel where hotelid=10000 and status='T';
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+--------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra                    |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+--------------------------+
|  1 | SIMPLE      | NULL  | NULL       | NULL | NULL          | NULL | NULL    | NULL | NULL |     NULL | Select tables optimized away |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+--------------------------+
1 row in set, 1 warning (0.00 sec)
```

在索引覆盖下，extra 提示 Select tables optimized away，这意味着在查询执行期间不需要读取表，可以通过索引直接返回结果。

b. 写为 order by 的方式

explain select id from hotel where hotelid=10000 and status='T' order by id desc limit 1;

```
mysql> explain select id from hotel where hotelid=10000 and status='T' order by id desc limit 1;
+----+-------------+-------+------------+-------+---------------+---------+---------+------+------+----------+--------------------------------+
| id | select_type | table | partitions | type  | possible_keys | key     | key_len | ref  | rows | filtered | Extra                          |
+----+-------------+-------+------------+-------+---------------+---------+---------+------+------+----------+--------------------------------+
|  1 | SIMPLE      | hotel | NULL       | index | NULL          | PRIMARY | 4       | NULL |    1 |     5.00 | Using where; Backward index scan |
+----+-------------+-------+------------+-------+---------------+---------+---------+------+------+----------+--------------------------------+
1 row in set, 1 warning (0.00 sec)
```

扫描数很少，虽然是 type=index 的索引扫描，但是由于 MYSQL 对 limit 的优化，实际上并不会全表扫描。

- 排序聚合写法

通常 SQL 在使用 Group by 及 Order by 后，会产生临时表和文件排序操作。若查询条件的数据量非常大，temporary 和 filesort 都会产生额外的巨大开销。

```
mysql> explain select name,hotelid from hotel group by name,hotelid order by name;
+----+-------------+-------+------------+------+---------------+------+---------+------+-------+----------+----------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows  | filtered | Extra                                        |
+----+-------------+-------+------------+------+---------------+------+---------+------+-------+----------+----------------------------------------------+
|  1 | SIMPLE      | hotel | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 30355 |   100.00 | Using temporary; Using filesort              |
+----+-------------+-------+------------+------+---------------+------+---------+------+-------+----------+----------------------------------------------+
1 row in set, 1 warning (0.00 sec)
```

a. 使用索引来满足排序聚合

alter table hotel add index idx_name_hotelid(name,hotelid);

```
mysql> alter table hotel add index idx_name_hotelid(name,hotelid);
Query OK, 0 rows affected (0.07 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> explain select name,hotelid from hotel group by name,hotelid order by name;
+----+-------------+-------+------------+-------+-----------------+-----------------+---------+------+------+----------+--------------------------+
| id | select_type | table | partitions | type  | possible_keys   | key             | key_len | ref  | rows | filtered | Extra                    |
+----+-------------+-------+------------+-------+-----------------+-----------------+---------+------+------+----------+--------------------------+
|  1 | SIMPLE      | hotel | NULL       | range | idx_name_hotelid | idx_name_hotelid | 807     | NULL |    6 |   100.00 | Using index for group-by |
+----+-------------+-------+------------+-------+-----------------+-----------------+---------+------+------+----------+--------------------------+
1 row in set, 1 warning (0.00 sec)
```

此时 MYSQL 可以通过访问索引来避免执行 filesort 及 temporary 操作

b. 取消隐形排序

在某些情况下，Group by 会默认实现隐形排序，通过添加 ORDER BY NULL 可以取消这种隐形排序。

*注意从 MySQL 8.0 开始，不会再有这种情况了，因此不需要 ORDER BY NULL 写法了

(4) 资源

- 锁资源等待

在读写很热的表上，通常会发生锁资源争夺，从而导致慢查询的情况。

a. 谨慎使用 for update 查询
b. 增删改尽量保证使用到索引
c. 降低并发，避免对同一条数据进行反复的修改

- 网络波动

a. 往客户端发送数据时发生网络波动导致的慢查询
b. 硬件配置
c. CPU 利用率高，磁盘 IO 经常满载，导致慢查询

三、总结

慢查询治理是一个长期且漫长的过程，不应等 SQL 超时报错后才开始考虑优化，从一开始

就要建立完善的日常化流程体系，才能有效的控制慢查询的增长。

但是经过长期优化后发现，仅仅从数据库层面优化，并不能实现慢查询完全“清零”，还有很多的痛点来自于业务逻辑和应用层面本身。这也需要研发工程师着重优化业务逻辑、应用策略，并加强数据库培训，在编写 SQL 时切勿过于随意，贪图省事，否则事后再优化会变得相当困难。

百亿节点，毫秒级延迟，携程金融基于 nebula 的大规模图应用实践

【作者简介】霖雾，携程数据开发工程师，关注图数据库等领域。

背景

2017 年 9 月携程金融成立，在金融和风控业务中，有多种场景需要对图关系网络进行分析和实时查询，传统关系型数据库难以保证此类场景下的关联性能，且实现复杂性高，离线关联耗时过长，因此对图数据库的需求日益增加。携程金融从 2020 年开始引入大规模图存储和图计算技术，基于 nebula 构建了千亿级节点的图存储和分析平台，并取得了一些实际应用成果。本文主要分享 nebula 在携程金融的实践，希望能带给大家一些实践启发。

本文主要从以下几个部分进行分析:

- 图基础介绍
- 图平台建设
- 内部应用案例分析
- 痛点与优化
- 总结规划

一、图基础

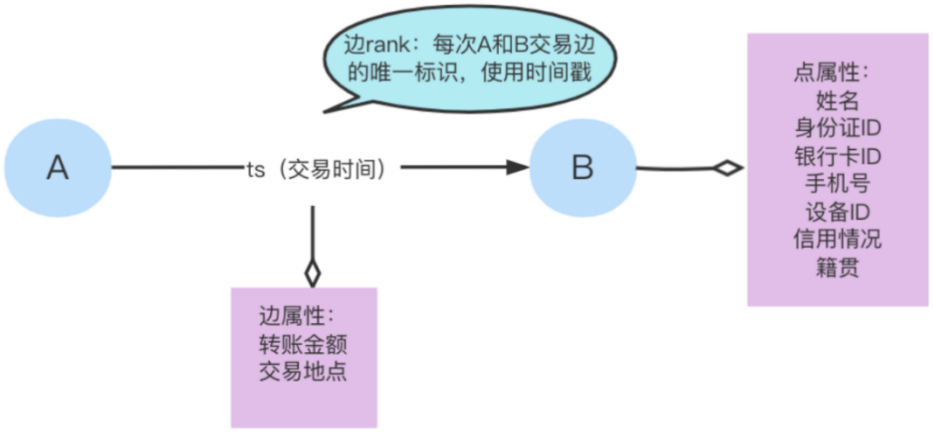
首先我们来简单介绍下图相关的概念:

1.1 什么是图

在计算机科学中，图就是一些顶点的集合，这些顶点通过一系列边结对（连接）。比如我们用一个图表示社交网络，每一个人就是一个顶点，互相认识的人之间通过边联系。

在图数据库中，我们使用 （起点,边类型,rank,终点）表示一条边。起点和终点比较好理解，表示一条边两个顶点的出入方向。边类型则是用于区分异构图的不同边，如我关注了你，我向你转账，关注和转账就是两种不同种类的边。而 rank 是用来区分同起始点同终点的不同边，如 A 对 B 的多次转账记录，起点、终点、边类型是完全相同的 ，因此就需要如时间戳作为 rank 来区分不同的边。

同时，点边均可具有属性，如：A 的手机号、银行卡、身份证号、籍贯等信息均可作为 A 的点属性存在，A 对 B 转账这条边，也可以具有属性，如转账金额，转账地点等边属性。



转账边构成示意图

1.2 什么时候用图

（信息收集于开源社区、公开技术博客、文章、视频）

（1）金融风控

- 诈骗电话的特征提取，如不在三步社交邻居圈内，被大量拒接等特征。实时识别拦截。（银行/网警等）；
- 转账实时拦截 （银行/支付宝等）；
- 实时欺诈检测，羊毛党的识别（电商）；
- 黑产群体识别，借贷记录良好用户关联，为用户提供更高额贷款、增加营收。

（2）股权穿透

影子集团、集团客户多层交叉持股、股权层层嵌套复杂关系的识别（天眼查/企查查）

（3）数据血缘

在数据仓库开发过程中， 会因为数据跨表关联产生大量的中间表，使用图可直接根据关系模型表示出数据加工过程和数据流向，以及在依赖任务问题时快速定位上下游。

（4）知识图谱

构建行业知识图谱

（5）泛安全

ip 关系等黑客攻击场景，计算机进程与线程等安全管理

（6）社交推荐

- 好友推荐，行为相似性，咨询传播路径，可能认识的人，大 v 粉丝共同关注，共同阅读文章等，商品相似性，实现好友商品或者咨询的精准推荐；
- 通过对用户画像、好友关系等，进行用户分群、实现用户群体精准管理。

（7）代码依赖分析

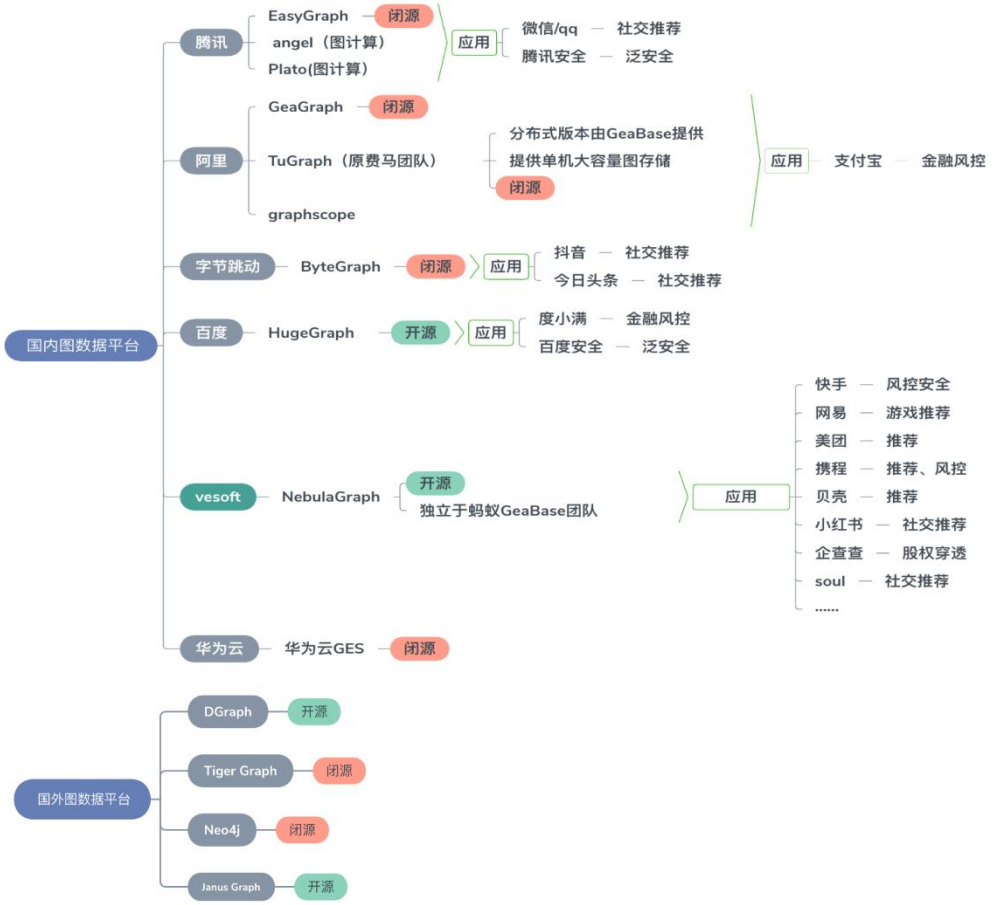
（8）供应链上下游分析

如汽车供应链上下游可涉及上万零件及供应商，分析某些零件成本上涨/供应商单一/库存少等多维度的影响（捷豹）

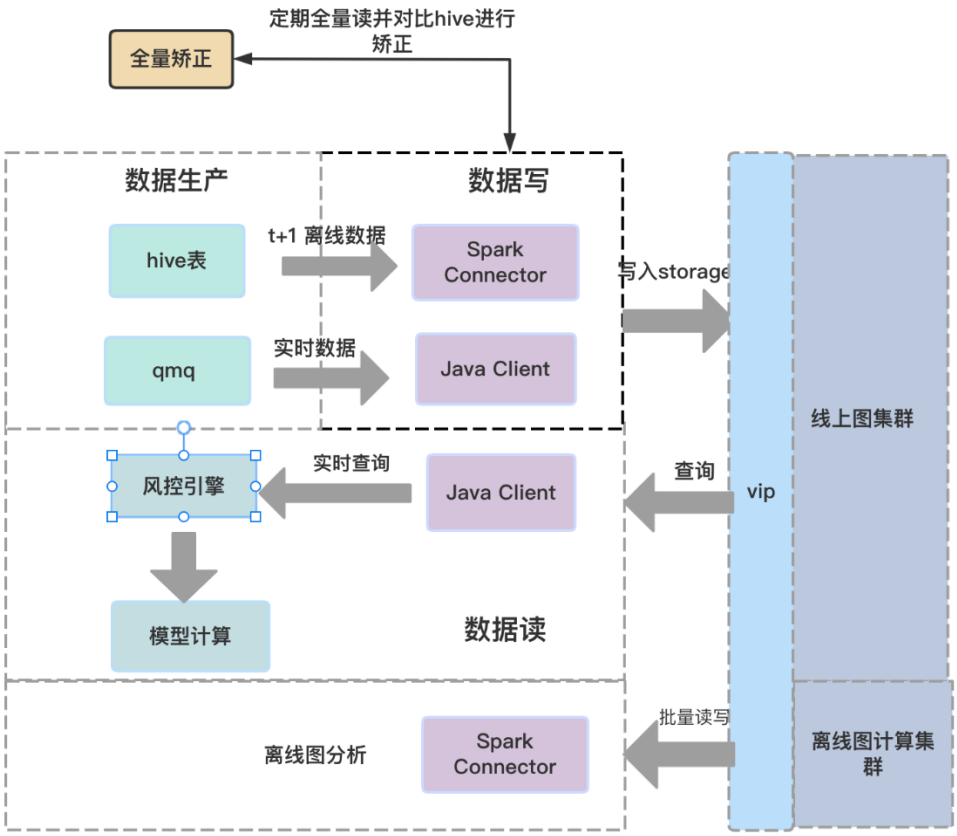
1.3 谁在研发图，谁在使用图

（信息收集于开源社区、公开技术博客、文章、视频）

目前国内几家大公司都有各自研发的图数据库，主要满足内部应用的需求，大多数都是闭源的，开源的仅有百度的 hugegraph。其他比较优秀的开源产品有 Google Dgraph, vesoft 的 nebula 等，其中 nebula 在国内互联网公司应用非常广泛。结合我们的应用场景，以及外部公开的测试和内部压测，我们最终选择 nebula 构建金融图平台。


国内图数据平台
腾讯
EasyGraph
闭源
angel（图计算）
Plato(图计算)
应用
微信/qq
社交推荐
腾讯安全
泛安全
阿里
GeaGraph
闭源
TuGraph（原费马团队）
分布式版本由GeaBase提供
提供单机大容量图存储
闭源
graphscope
应用
支付宝
金融风控
字节跳动
ByteGraph
闭源
应用
抖音
社交推荐
今日头条
社交推荐
百度
HugeGraph
开源
应用
度小满
金融风控
百度安全
泛安全
vesoft
NebulaGraph
开源
独立于蚂蚁GeaBase团队
应用
快手
风控安全
网易
游戏推荐
美团
推荐
携程
推荐、风控
贝壳
推荐
小红书
社交推荐
企查查
股权穿透
soul
社交推荐
.....
华为云
华为云GES
闭源
国外图数据平台
DGraph
开源
Tiger Graph
闭源
Neo4j
闭源
Janus Graph
开源

二、图平台建设



2.1. 图平台建设

我们的图平台早期只有 1 个 3 节点的 nebula 集群，随着图应用场景的不断扩充，需要满足实时检索、离线分析、数据同步与校验等功能，最终演化成上述架构图。

（1）离线图：主要用于图构建阶段（建模、图算法分析），通过 spark-connector 同集团的大数据平台打通，此外我们还将 Nebula 提供的数 10 种常用图算法进行工具化包装，方便图分析人员在 spark 集群提交图算法作业。

（2）线上图：经过离线图分析确定最终建模后，会通过 spark-connector 将数据导入线上图。通过对接 qmq 消息（集团内部的消息框架）实时更新，对外提供实时检索服务。 同时也会有 T+1 的 hive 增量数据通过 spark-connector 按天写入。

（3）全量校验：虽然 Nebula Graph 通过 TOSS 保证了正反边的插入一致性，但仍不支持事务，随着数据持续更新，实时图和离线(hive 数据)可能会存在不一致的情况，因此我们需要定期进行全量数据的校验(把图读取到 Hive，和 Hive 表存储的图数据进行比对，找出差异、修复），保证数据的最终一致性。

（4）集群规模：为了满足千亿节点的图业务需求，实时集群采用三台独立部署的高性能机器，每台机器 64core / 320GB / 12TB SSD ，版本为 nebulav2.5，跨机房部署。离线集群 64core /320GB /3.6TB SSD * 12 ， 测试集群 48core/ 188GB/5T HDD * 4.

2.2. 遇到的问题

在 nebula 应用过程中，也发现一些问题，期待逐步完善：

（1）资源隔离问题，目前 nebula 没有资源分组隔离功能 ，不同业务会相互影响；如业务图 A 在导数据，业务图 B 线上延迟就非常高。

（2）版本升级问题：

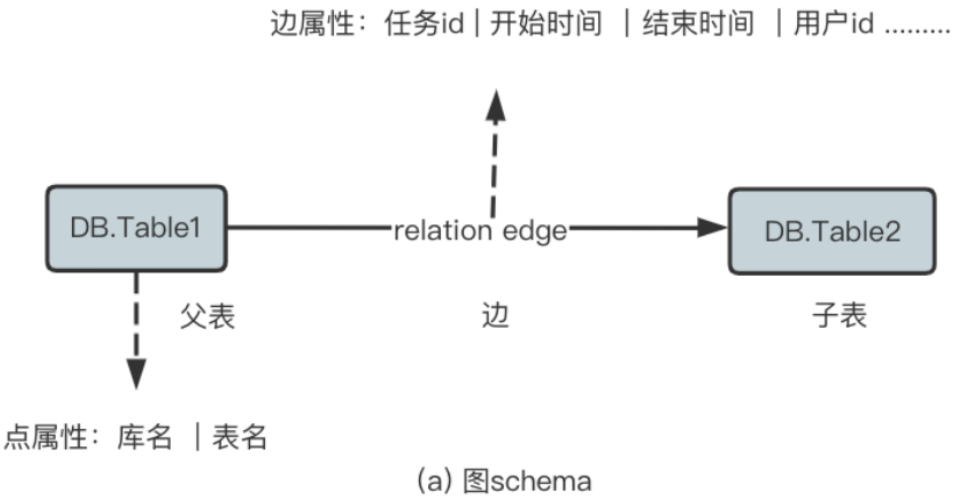
nebula 在版本升级过程中需要停止服务，无法实现热更新；对于类似实时风控等对可靠性要求非常高的场景非常不友好。此种情况下如需保证在线升级，就需要配备主备集群，每个集群切量后挨个升级，增加服务复杂性和运维成本。

客户端不兼容，客户端需要跟着服务端一起升级版本。对于已有多个应用使用的 nebula 集群，想要协调各应用方同时升级客户端是比较困难的。

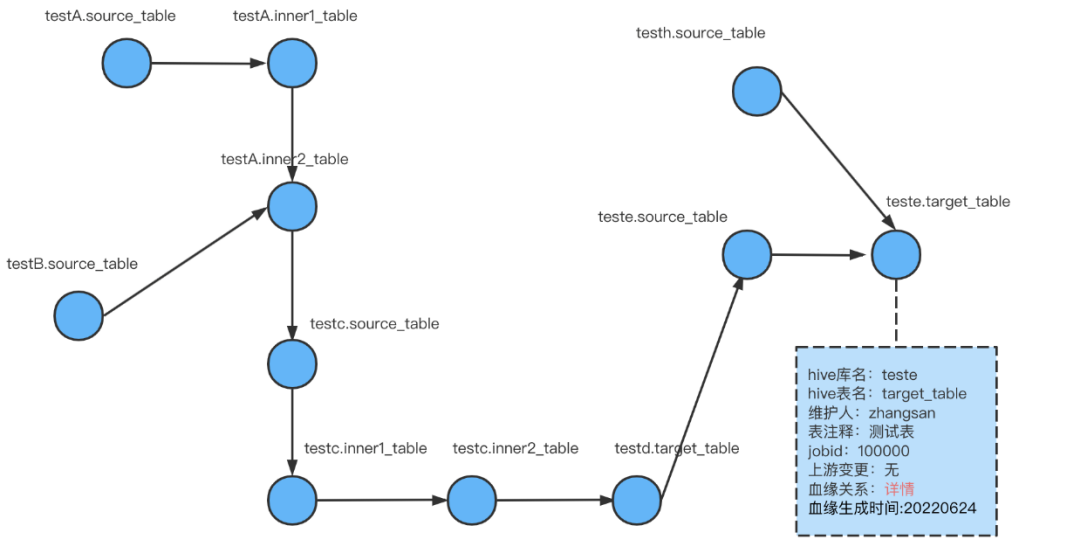
三、内部应用案例分析

3.1 数据血缘图

数据治理是近年来比较热的一个话题，他是解决数仓无序膨胀的有效手段，其中数据血缘是数据有效治理的重要依据，金融借助 nebula 构建了数据血缘图，以支撑数据治理的系统建设。



(a) 图schema



(b) 表依赖关系图　　数据规模：百万点边

数据血缘就是数据产生的链路，记录数据加工的流向，经过了哪些过程和阶段；主要解决 ETL 过程中可能产出几十甚至几百个中间表导致的复杂表关系，借用数据血缘可以清晰地记录数据源头到最终数据的生成过程。

图 a 是数据血缘的关系图，采用库名 + 表名作为图的顶点来保证点的唯一性，点属性则是分开的库名和表名，以便通过库名或者表名进行属性查询。在两张表之间会建立一条边，边的属性主要存放任务的产生运行情况，比如说：任务开始时间，结束时间、用户 ID 等等同任务相关的信息。

图 b 是实际查询中的一张关系图，箭头的方向表示了表的加工方向，通过上游或者下游表我们可以快速地找到它的依赖， 清晰明了地显示从上游到下游的每一个链路。

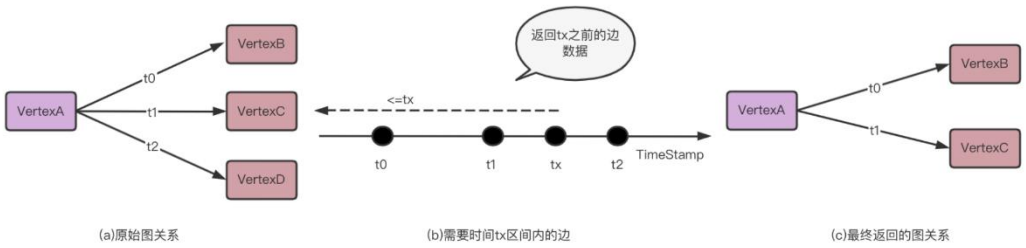
如果要表达复杂的血缘依赖关系图，通过传统的关系型数据库需要复杂的 SQL 实现（循环嵌套），性能也比较差，而通过图数据库实现，则可直接按数据依赖关系存储，读取也快于传统 DB，非常简洁。目前，数据血缘也是金融 BU 在图数据库上的一个经典应用。

3.2 风控关系人图

关系人图常用于欺诈识别等场景，它是通过 ID、设备、手机标识以及其他介质信息关联不同用户的关系网络。比如说，用户 A 和用户 B 共享一个 WiFi，他们便是局域网下的关系人；用户 C 和用户 D 相互下过单，他们便是下单关系人。简言之，系统通过多种维度的数据关联不同的用户，这便是关系人图。

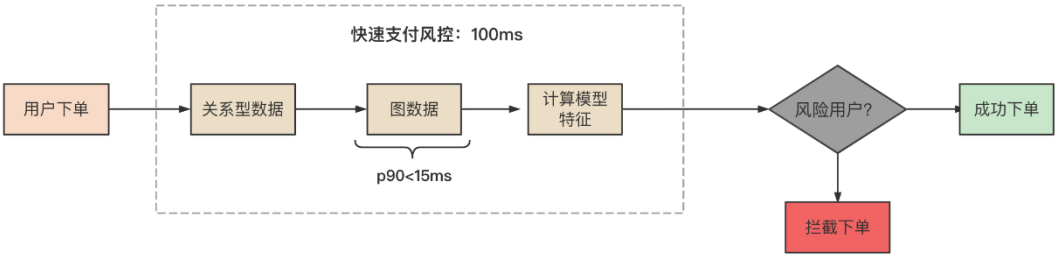
构建模型时，通常要查询某个时点（比如欺诈事件发生前）的关系图，对当时的图进行模型抽取和特征构建，我们称这个过程为图回溯。随着回溯时间点的不同，返回的图数据也是动态变化的；比如某人上午，下午各自打了一通电话， 需要回溯此人中午时间点时的图关系，只会出现上午的电话记录，具体到图，则每类边都具有此类时间特性，每一次查询都需要对时间进行限制。

对于图回溯场景，最初我们尝试通过 HIVE SQL 实现，发现对于二阶及以上的图回溯，SQL 表达会非常复杂，而且性能不可接受（比如二阶回溯 Hive 需要跑数小时，三阶回溯 Hive 几乎不能实现）；因此尝试借助图数据库来实现，把时间作为边 rank 进行建模，再根据边关系进行筛选来实现回溯。这种回溯方式更直观、简洁，使用简单的 API 即可完成，在性能上相比 Hive 也有 1 个数量级以上的提升（二阶回溯，图节点：百亿级，待回溯节点：10 万级）。



下面用一个例子说明：如图(a)，点 A 分别在 t0 、t1、 t2 时刻建立了一条边 ，t0、t1、t2 为边 rank 值，需要返回 tx 时的的图关系数据，只能返回 t0、 t1 对应的 点 B、C ，因为当回溯到 tx 时间点时候，t2 还没有发生；最终返回的图关系为 t0 和 t1 时候，VertexA ->VertexB 、 VertexA -> VertexC （见图 (c) ）。这个例子是用一种边进行回溯，实际查询中可能会涉及到 2~3 跳，且存在异构边（打电话是一种边，点外卖又是一种边，下单酒店机票是一种边，都是不同类型的边），而这种异构图的数据都具有回溯特征，因此实际的关系人图回溯查询也会变得复杂。

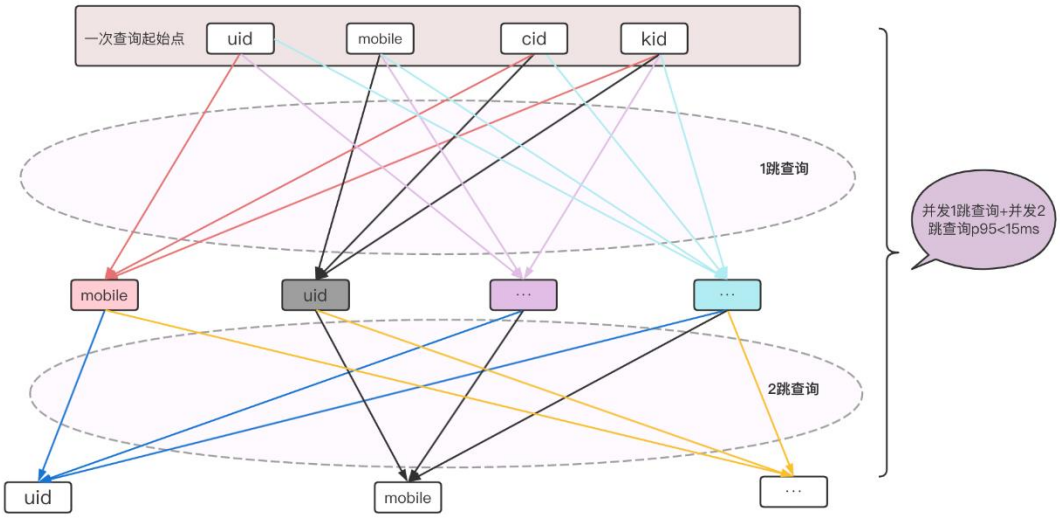
3.3 实时反欺诈图



下单流程

用户下单时，会进入一个快速风控的阶段：通过基于关系型数据库和图数据库的规则进行模型特征计算，来判断这个用户是不是风险用户，要不要对该用户进行下单拦截（实时反欺诈）。

我们可以根据图关系配合模型规则，用来挖掘欺诈团伙。比如说，已知某个 uid 是犯欺团伙的一员，根据图关联来判断跟他关系紧密的用户是不是存在欺诈行为。为了避免影响正常用户的下单流程，风控阶段需要快速响应，因此对图查询的性能要求非常高（P95 <15ms）。我们基于 nebula 构建了百亿级的反欺诈图，在查询性能的优化方面进行了较多思考。



此图 Schema 为脱敏过后的部分图模型，当中隐藏很多建模信息。这里简单讲解下部分的查询流程和关联信息。

如上图为一次图查询流程，每一次图查询由多个起始点如用户 uid、用户 mobile 等用户信息同时开始，每条线为一次关联查询，因此一次图查询由几十次点边查询组成，由起始点经过一跳查询和 2 跳查询，最终将结果集返回给风控引擎。

系统会将用户的信息，转化为该用户的标签。在图查询的时候，根据这些标签，如 uid、mobile 进行独立查询。举个例子，根据某个 uid 进行一跳查询，查询出它关联的 5 个手机号。再根据这 5 个手机号进行独立的 2 跳查询，可能会出来 25 个 uid，查询会存在数据膨胀的情况。因此，系统会做一个查询限制。去查看这 5 个手机号关联的 uid 是不是超过了系统设定的热点值。如果说通过 mobile 查询出来关联的手机号、uid 过多的话，系统就会判断其为热点数据，不进行边结果返回。（二阶/三阶回溯，图点边：百亿级）。

四、痛点及优化

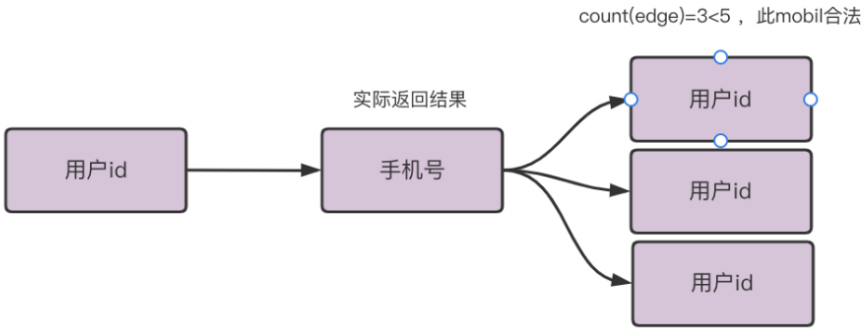
在上述应用场景中，对于风控关系人图和反欺诈图，由于图规模比较大（百亿点边），查询较多，且对时延要求较高，遇到了一些典型问题，接下来简单介绍一下。

4.1 查询性能问题

为了满足实时场景 2 跳查询 p95 15ms 需求，我们针对图 schema 和连接池以及查询端做了一些优化：

4.1.1 牺牲写性能换取读性能

如：查询id关联的手机号（要求:手机号关联id不超过5？）

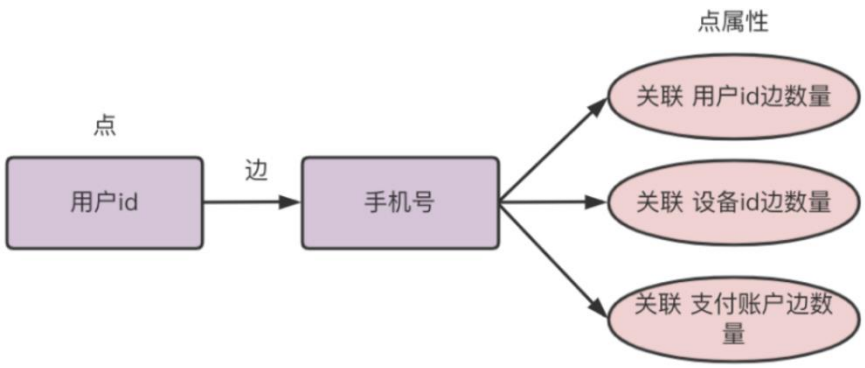



```
(1) var x = go from  $id over $edgeName |limit 5 ;
(2)  go from item over  $edgeName reversely  where count < 5;
```

(a)两跳查询，并返回一跳结果

首先，我们来看看这样的一个需求: 查询 id 关联的手机号 ，需要满足对于这个手机号关联边不超过 3 个。这里解释下为什么要限制关联边数量， 因为我们正常个体关联边数量是有限的，会有一个对于大多数人的 p95 这样的阈值边数量，超过这个阈值就是脏数据。为了这个阈值校验， 就需要对每次查询的结果再多查询一跳。

如图(a)所示，我们需要进行 2 次查询，第一跳查询是为了查询用户 id 关联的手机号， 第二跳查询是为了保证我们的结果值是合法的（阈值内），这样每跳查询最终需要进行 2 跳查询来满足。如图给出了图查询的 gsql 2 步伪码，这种情况下无法满足我们的高时效性。如何优化呢？看下图(b) ：




```
(1) go from  $id over $edgeName where $手机号._用户id边数量<5 |limit 5
```

(b)修改schema后一跳查询

我们可以将热点查询固定在点属性上，这样一跳查询时就可以知道该点有多少关联边， 避免进行图 a 中（2）语句验证。还是以图 (a)为例，从一个用户 ID 开始查询，查询他的手机号关联，此时因为手机号关联的边已经变成了点属性（修改了 schema），图(a) 2 条查询语句实现的功能就可以变成一条查询 go from $id over $edgeName where $手机号.用户 id 边数据 <5 | limit 5。

这种设计的好处就是，在读的时候可以加速验证过程， 节约了一跳查询。带来的成本是：每写一条边，同时需要更新 2 个点属性来记录点的关联边情况，而且需要保证幂等（保证重复提交不会叠加属性+1），当插入一条边的时，先去图里面查询边是否存在，不存在才会进行写边以及点属性 +1 的操作。也就是我们牺牲了写性能，来换取读性能，并通过定期 check 保证数据一致。

4.1.2 池化连接降低时延

第二个优化手段是通过池化连接降低时延。Nebula 官方连接池每次进行查询均需要进行建立初始化连接-执行查询任务-关闭连接。而在高频（QPS 会达到几千）的查询场景中，频繁的创建、关闭连接非常影响系统的性能和稳定性。且建立连接过程耗时平均需要 6ms， 比实际查询时长 1.5ms 左右高出几倍，这是不可接受的。因此我们对官方客户端进行了二次封装，实现连接的复用和共享。最后将查询 p95 从 20ms 降低到了 4ms。通过合理控制并发，我们最终将 2 跳查询性能控制在 p95 15ms 。

这里贴下代码供参考：

```
public class SessionPool {

    /**
     * 创建连接池
     *
     * @param maxCountSession  默认创建连接数
     * @param minCountSession  最大创建连接数
     * @param hostAndPort      机器端口列表
     * @param userName          用户名
     * @param passWord          密码
     * @throws UnknownHostException
     * @throws NotValidConnectionException
     * @throws IOErrorException
     * @throws AuthFailedException
     */
    public SessionPool(int maxCountSession, int minCountSession, String hostAndPort,
String userName, String passWord) throws UnknownHostException,
NotValidConnectionException, IOErrorException, AuthFailedException {
        this.minCountSession = minCountSession;
        this.maxCountSession = maxCountSession;
        this.userName = userName;
```

```
        this.passWord = passWord;
        this.queue = new LinkedBlockingQueue<>(minCountSession);
        this.pool = this.initGraphClient(hostAndPort, maxCountSession, minCountSession);
        initSession();
    }

    public Session borrow() {
        Session se = queue.poll();
        if (se != null) {
            return se;
        }
        try {
            return this.pool.getSession(userName, passWord, true);
        } catch (Exception e) {
            log.error("execute borrow session fail, detail: ", e);
            throw new RuntimeException(e);
        }
    }

    public void release(Session se) {
        if (se != null) {
            boolean success = queue.offer(se);
            if (!success) {
                se.release();
            }
        }
    }

    public void close() {
        this.pool.close();
    }

    private void initSession() throws NotValidConnectionException, IOErrorException,
AuthFailedException {
        for (int i = 0; i < minCountSession; i++) {
            queue.offer(this.pool.getSession(userName, passWord, true));
        }
    }

    private NebulaPool initGraphClient(String hostAndPort, int maxConnSize, int minCount)
throws UnknownHostException {
        List<HostAddress> hostAndPorts = getGraphHostPort(hostAndPort);
        NebulaPool pool = new NebulaPool();
        NebulaPoolConfig nebulaPoolConfig = new NebulaPoolConfig();
```

```
            nebulaPoolConfig = nebulaPoolConfig.setMaxConnSize(maxConnSize);
            nebulaPoolConfig = nebulaPoolConfig.setMinConnSize(minCount);
            nebulaPoolConfig = nebulaPoolConfig.setIdleTime(1000 * 600);
            pool.init(hostAndPorts, nebulaPoolConfig);
            return pool;
        }

        private List<HostAddress> getGraphHostPort(String hostAndPort) {
            String[] split = hostAndPort.split(",");
            return Arrays.stream(split).map(item -> {
                String[] splitList = item.split(":");
                return new HostAddress(splitList[0], Integer.parseInt(splitList[1]));
            }).collect(Collectors.toList());
        }

        private Queue<Session> queue;

        private String userName;

        private String passWord;

        private int minCountSession;

        private int maxCountSession;

        private NebulaPool pool;

}
```

4.1.3 查询端优化

对于查询端，像 3.3 中的例图，每一次图查询由多个起始点开始，可拆解为几十次点边查询，需要让每一层的查询尽可能地并发进行，降低最终时延。我们可以先对 1 跳查询并发（约十几次查询），再对结果进行分类合并，进行第二轮的迭代并发查询（十几到几十次查询），通过合理地控制并发，可将一次组合图查询的 P95 控制在 15 ms 以内。

4.2 边热点问题

在图查询过程中，存在部分用户 id 关联过多信息，如黄牛用户关联过多信息，这部分异常用户会在每一次查询时被过滤掉，不会继续参与下一次查询，避免结果膨胀。而判断是否为异常用户，则依赖于数据本身设定的阈值，异常数据不会流入下一阶段对模型计算造成干扰。

4.3 一致性问题

Nebula Graph 本身是没有事务的，对于上文写边以及点属性 +1 的操作，如何保证这些操作的一致性，上文提到过，我们会定期对全量 HIVE 表数据和图数据库进行 check，以 HIVE 数据为准对线上图进行修正，来实现最终一致性。目前来说，图数据库和 HIVE 表不一致的情况还是比较少的。

五、总结与展望

基于 nebula 的图业务应用，完成了对数据血缘、对关系人网络、反欺诈等场景的支持，并将持续应用在金融更多场景下，助力金融业务。我们将持续跟进社区，结合自身应用场景推进图平台建设 ； 同时也期待社区版能提供热升级、资源隔离、更丰富易用的算法包、更强大的 studio 等功能。

云计算

携程 Service Mesh 可用性实践

【作者简介】本文作者烧鱼、Shirley 博，来自携程 Cloud Container 团队，目前主要从事 Service Mesh 在携程的落地，负责控制面的性能优化及可用性建设，以及推进各类基础设施服务的云原生化。

一、背景

近几年，国内各大公司大规模生产落地 Kubernetes 和 Service Mesh，拉开了云原生革命的序幕。从 2019 年开始，团队开始在部分场景中落地 Istio Gateway，积累 Service Mesh 经验；2020 年中，我们开始与公司的框架部门合作着手 Service Mesh 在携程的落地，目前生产环境已有数百个应用接入，覆盖率还在持续的推进过程中。

Service Mesh 作为一项新技术，相比传统的微服务框架有很多优势。但在生产环境落地的过程中，若无法保证可用性，出现大的故障，将会大大打击对新技术采用的信心，也会影响最终用户，造成对品牌的负面影响。显而易见，可用性是一切的基石。我们在落地 Service Mesh 的过程投入了大量的精力进行可用性的建设，避免出现单点故障，保证服务的高可用。Service Mesh 在携程的落地并不是平地起高楼，公司内关于可用性已经有一套方法与模式，Service Mesh 的可用性建设也必须考虑现有的高可用体系。

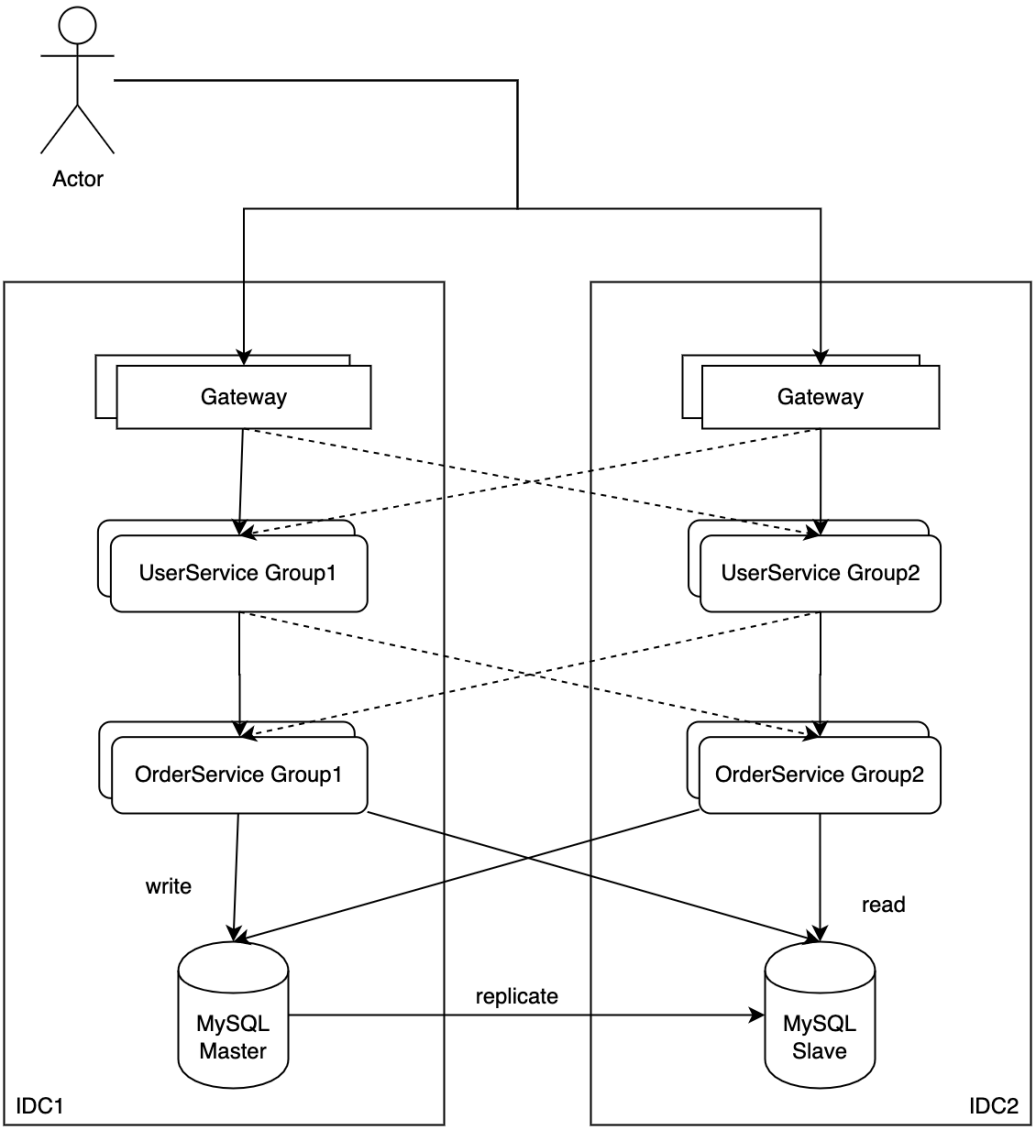
二、Service Mesh 高可用

2.1 携程高可用架构简介

携程现有的高可用设计自上而下可以简化为：IDC 级高可用、IDC 内应用部署的高可用、IDC 内基础设施的高可用。针对这三个层次的故障，分别有不同的设计，以下主要围绕 IDC 级及应用部署级可用性设计进行介绍。

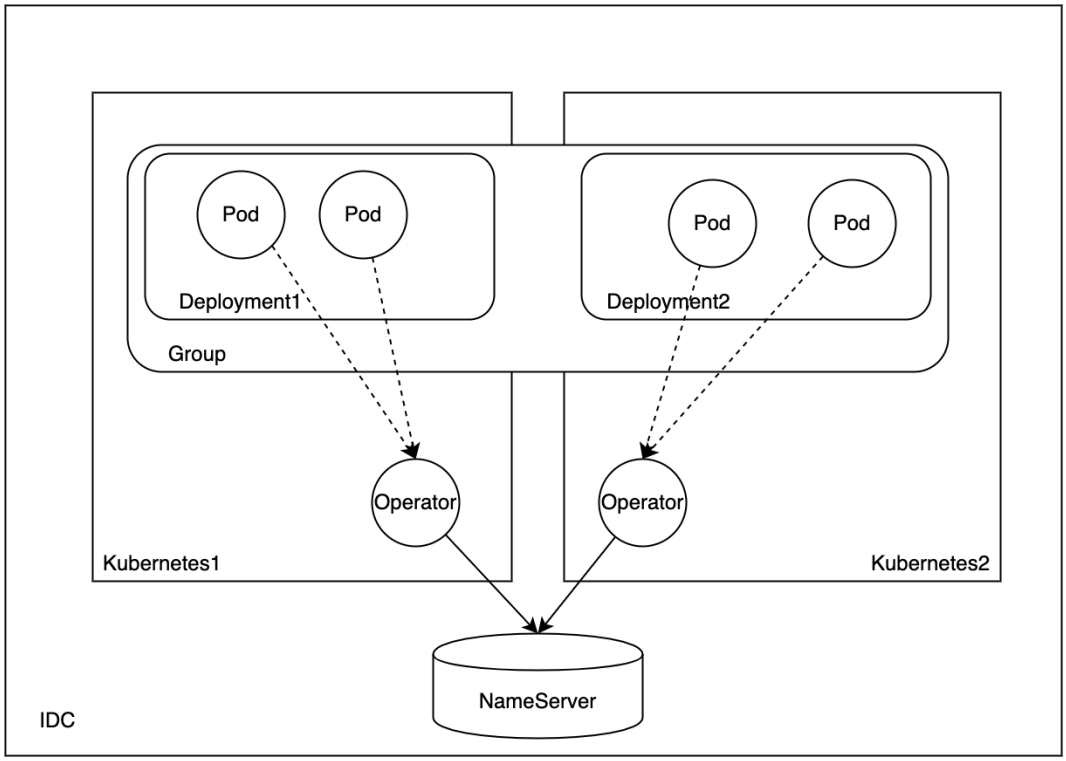
2.1.1 IDC 级灾备-同城双活

携程内部应用一般包含多个 Group，Group 是应用发布的最小单位，同时也是流量调度的单位。一个 DR(容灾)组一般包含两个 IDC，一个应用的多个 Group 会部署在 DR 组内的多个 IDC，当某一个 IDC 出现故障时，迅速将流量切换到另一个 IDC。其中核心应用多 IDC 部署，数据库跨 IDC 主备，流量调度层面也要支持跨 IDC 的管控。



2.1.2 IDC 内高可用-应用多集群部署

随着容器化在携程内部大规模推进，绝大部分应用都跑在 Kubernetes 集群上。应用部署时，将同一个 Group 的实例，打散到多个 Kubernetes 集群上。然后，各个集群中的 Operator 与外部的注册中心交互，将本集群中的实例信息注册到对应的 Group 中。单个 Kubernetes 的控制面不可用，只会影响到当前集群实例的变更。



可见网站的高可用设计就是通过划分故障域，进行故障隔离，切断故障的传导，保证故障时能够切换或控制故障范围。数据中心之间故障隔离，保证数据中心级的高可用；数据中心内部 Kubernetes 集群之间故障隔离，进而保证 Kubernetes 之上的可用性。在 Service Mesh 的高可用设计上也要遵守这些故障隔离的原则。

2.2 Service Mesh 高可用设计

思路上，我们先梳理故障场景，制定目标，然后设计方案，再结合故障场景进行可用性分析，上线之后再通过故障演练进行验证。

2.2.1 故障场景

（1）Service Mesh 数据面故障，可能会导致应用请求异常，从而影响服务质量，大规模的数据面故障可能会导致数据中心级别的故障。

（2）Service Mesh 控制面故障，无法给数据面下发最新的配置，数据面可以根据现在有配置提供服务。故障期间，实例 IP 的变化无法下发，路由信息也无法更新，部分服务可能会受影响。

（3）目前 Service Mesh 控制面还是以 Kubernetes 为基础的，所以 Kubernetes 控制面的故障也会导致 Service Mesh 控制面的故障。

2.2.2 目标制定

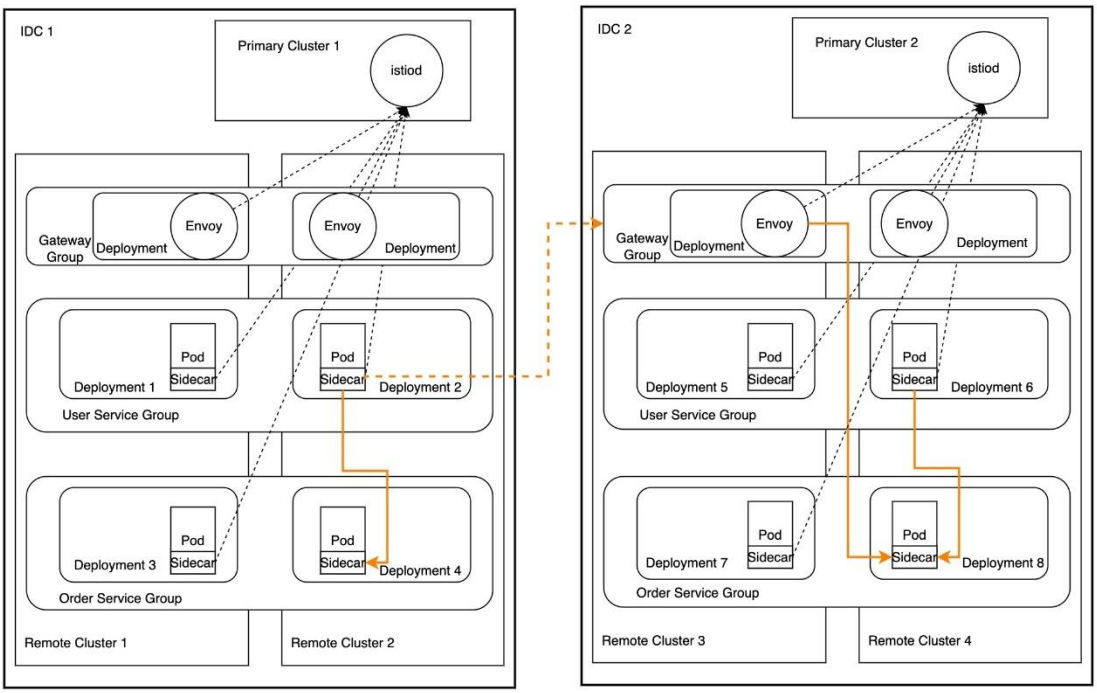
（1）数据中心级故障隔离。考虑极端的故障场景，数据面大规模故障，控制面长时间无法恢复等，设计上需要将故障控制在单个数据中心内部，防止故障传导到其他数据中心，从而保证有其他数据中心可以提供服务，可以进行数据中心切换。

（2）支持应用多集群部署架构。从设计上考虑，Service Mesh 的高可用不应该仅仅局限于现在的应用多集群部署的架构，而是与应用部署架构解耦，支持多种部署架构。

2.2.3 方案设计

多个数据中心之间隔离，每个数据中心有独立的控制面，将管理的资源收敛到每个数据中心内部。应用跨数据中心访问时需要走 Gateway，通过 ServiceEntry 的方式导入其他集群的 Gateway，由各个数据中心的 Gateway 收敛对外的服务入口，以数据中心为单位隔离故障。

在数据中心内部，控制面部署到独立的 Kubernetes 集群部署作为 Primary Cluster，应用所在的 Kubernetes 集群作为 Remote Cluster，Remote Cluster 上的 Sidecar 共用同一个 Service Mesh 控制面。通过将 Service Mesh 控制面与应用部署的 Kubernetes 的集群拆分开的方式，可以灵活应对其他的应用多集群部署架构。



2.2.4 可用性分析

（1）当某一个数据中心出现故障时，需要将流量切换到另一个数据中心，与现有架构保持一致。

（2）当 Service Mesh 数据面出现故障时，应用可服务的实例数会减少。但是基于 Envoy FailOver 的能力，随着健康的实例数的减少，也会逐渐将流量转到其他数据中心的 Gateway 上，服务可用性不会有太大影响。

（3）Primary Cluster 出现故障时（包括 Primary Kubernetes 集群故障和 Service Mesh 控制面故障），无法通过控制面下发新的配置。如果此时该数据中心有服务异常不可用，数据面也会自动 FailOver 其他数据中心，短时间内没有太大影响。而数据面访问其他数据中心的服务时，因为其他数据中心的服务都是收敛到了 Gateway 上，所以并不感知具体的实例信息，即使其他数据中心服务实例信息变化，也不会因为配置无法下发，导致数据面的访问出现异常的问题。

（4）在数据中心内部，Remote Cluster 出现问题时，只会影响应用的 HPA 和发布，在 Kubernetes 的高可用层面解决，Service Mesh 层面没有影响。

2.2.5 故障演练

（1）确认是否按照设计预期的方式应对故障。将某一个数据中心内的服务，置成不可用状态，观察数据面是否按照预期执行了 FailOver，以及数据面请求的成功率和延迟，是否在预期范围内。

（2）分析演练的效果，挖掘是否存在隐藏的问题点。当某一个数据中心内，服务不可用时，数据面会 FailOver 到其他数据中心的 Gateway 上，此时需要重点检查是否存在环路，防止请求在多个数据中心的 Gateway 上循环。

三、Service Mesh 自身的可用性提升

宏观层面的高可用的架构，通常是作为应对服务故障的兜底措施，可以应对不同级别的灾难，故障，但是数据中心级别的切换影响面较大，不能作为常规武器，服务自身也要加强可用性建设。Service Mesh 可用性建设还要围绕使用场景，深度完善可观察性指标，从而迭代优化提升可用性。

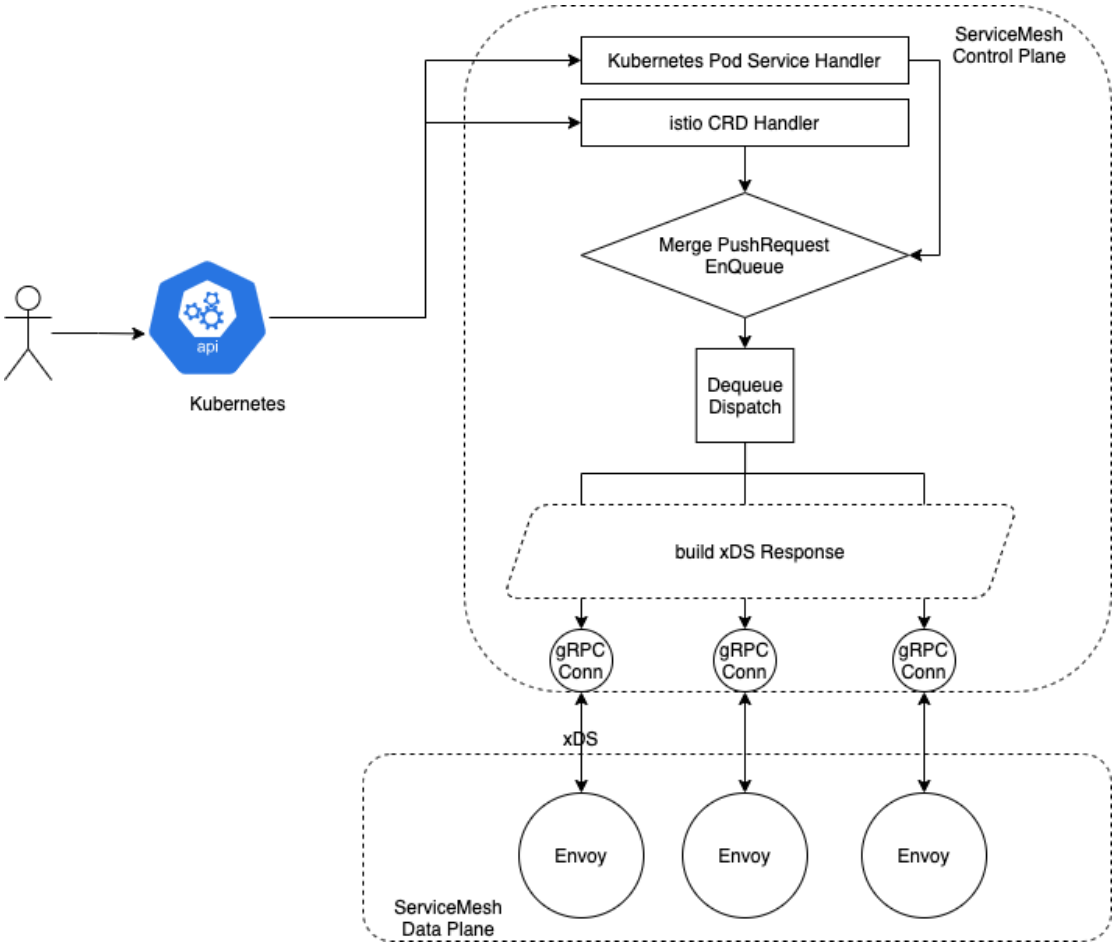
3.1 场景/目标

（1）控制面运行时，需要支撑大量客户端的数据面连接，同时需要快速的将配置推送到数据面。

（2）故障恢复，由于 Node 故障或者服务自身异常，控制面需要快速启动提供服务。

（3）发布场景，控制面和数据面的新版本发布，需要快速灰度和快速回滚的能力。

3.2 确定 xDS 推送指标



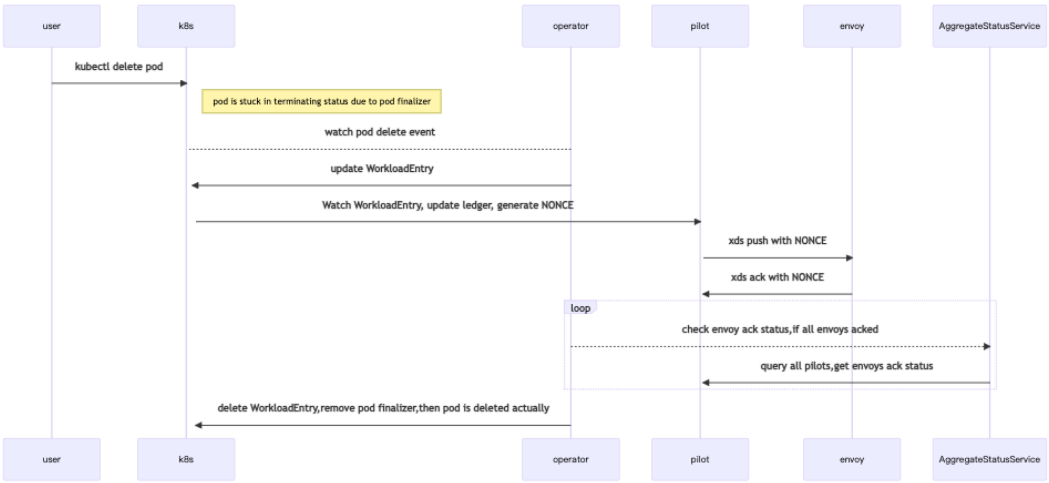
梳理配置下发的流程，apply 到 Kubernetes，到控制面处理该事件，并触发 xDS 的推送，最后数据面 ack 回来。主要是控制面内部的流程比较复杂，社区的版本中针对内部的各个阶段都有一些指标衡量，但是缺少一个描述整个下发流程耗时的指标。针对这个问题，内部已有相应的计划，初步验证了可行性。

以控制面接收到 Event 的时间做为起始时间，中间多个事件合并时取最小值，数据面的 ack 为结束时间。最终就可以得到控制面推送的耗时分布，从而进行针对性优化。

3.3 xDS 推送的可靠性保证

举例说明，在应用的滚动发布过程中实例的 IP 都发生了变化，老的实例已经被删除，但最新的实例信息可能没有及时推送到数据面，可能导致数据面出现访问异常。根本问题在于实例删除的动作和配置下发是异步的，无法保证在实例删除之前，所有的数据面已经获取到了该实例的下线信息将流量摘除。在大规模动态扩缩，或者临界场景时，这种异步和非确定性的方式极易导致大面积的服务不可用。

因此我们需要一个保证实例变更流程可靠的机制。



（1）在我们内部使用方式上，Pod 会有一个对应的 WorkloadEntry，然后通过 ServiceEntry 注册成为服务。内部 kubelet 扩展了 finalizer 机制，使得 finalizer 不权阻塞对象的删除，还会阻塞 kubelet 对 Pod 的 Kill 操作。Pod 创建时，会被对应的 Operator 打上 finalizer，所以删除 Pod 时，Pod 会因为 Finalizer 的存在而继续存活，此时还可以提供服务。

（2）由内部的 Operator 感知 Pod 变化，更新 WorkloadEntry 的 label，使之被更新为不可用状态，不会被任何 ServiceEntry 选中。此时 WorkloadEntry 的 generation 会+1 为 X。

（3）然后控制面 watch 到这个事件，会更新内部的 ledger 写入 WorkloadEntry 的 NamespacedName 以及 generation，并将此时 RootHash 作为 xDS 的 Nonce 推送给数据面，当数据面 ack 时，控制面会记录该数据面最新的 ack 的 Nonce。此时可以根据 Nonce 以及 NamespacedName 去 ledger 中查询已经推送下去的资源的 generation。

（4）再由 Operator 调用我们的聚合服务，去查询所有的控制面中数据面的 WorkloadEntry 的 generation，与 X 对比，如果都大于等于 X，则认为这个 WorkloadEntry 都下发成功了。如果是推送未完成，Operator 也会进行重试，等待推送成功。

（5）确认推送成功之后，Operator 会去删掉 WorkloadEntry，去除 Pod finalizer，真正的删除 Pod。

3.4 ServiceEntry/WorkloadEntry 事件处理耗时优化

实践上，我们通过 ServiceEntry 和 WorkloadEntry 将部署到虚拟机和物理机上的服务导入到 Mesh 中，但是发现一个 namespace 下 5 千 ServiceEntry 和 1 万 WorkloadEntry 的场景下，事件处理的耗时都在分钟级，多个应用发布时，延迟可能会到小时级别，根本无法满足上线的需求。

深入研究发现 istio 1.7，当一个 ServiceEntry/WorkloadEntry 发生变化时，会触发 maybeRefreshIndexes 这个方法，方法中会进行遍历所有的 ServiceEntry 和 WorkloadEntry 进行匹配，重新生成内存的 Map，因此会执行 5000*10000=5 千万次 workloadLabels.IsSupersetOf()。

```
func (s *ServiceEntryStore) maybeRefreshIndexes() {
    ...
    wles, err := s.store.List(gvk.WorkloadEntry, model.NamespaceAll)// 10000
    for _, wcfg := range wles {
        ...
        entries := seWithSelectorByNamespace[wcfg.Namespace]
        for _, se := range entries { // 5000
            workloadLabels := labels.Collection{wle.Labels}
            if !workloadLabels.IsSupersetOf(se.entry.WorkloadSelector.Labels) { // 5000 *
10000
                continue
            }
        }
    }
}
```

解决问题的思路肯定是全量变增量，当 WorkloadEntry 发生变化时，只要遍历所在 namespace 的 ServiceEntry，那么循环的次数就下降到了原来的万分之一，事件处理耗时也得到极大优化，下降到毫秒级。

具体实现上，处理事件的过程中需要更新多个 Map，为了保证数据的准确性，还是沿用了原有的全局锁。所以目前也只是完成从全量到增量的优化，但是事件还是串行处理方式，在大规模变更的场景下延迟会被逐步放大，最后的变更推送下去需要很长的时间。目前也在尝试拓展一个新的 CRD，实现上使用 Sync.Map 分段式锁，替换原有的全局锁，通过并发处理的方式来提升事件处理的效率。

熟悉 Kubernetes 和 istio 的同学也会发现，istio 在处理 event 时是并没有用 Kubernetes Controller Runtime 的方式编程，以 WorkloadEntry 和 ServiceEntry 为例，当 WorkLoadEntry 变化时，应该去查找关联的 ServiceEntry，然后触发 ServiceEntry 的 Reconcile。目前内部也在尝试通过引入 Controller Runtime 来处理 ServiceEntry，并且调大 MaxConcurrentReconciles，让控制面支持并发的处理事件，进一步提升推送的时效。

3.5 控制面冷启动

在我们实践过程中也发现控制面 ready 之后，没有及时给连接上来的数据面推送最新的配置，导致数据面无法获取到最新的实例信息，从而影响了请求的转发。

深入分析发现，控制面 ready 时只等待 Kubernetes 的 informer 完成 sync，但此时还有很多事件阻塞在内部队列中，导致控制面在 ready 之后的一段时间内，无法处理最新的事件，影响数据面下发最新的配置。在集群规模较大或者是 istio 资源较多的时候，尤为明显。对此我们增强了控制面启动流程，阻塞控制面 ready，等待内部的队列被清空，从而保证可以尽快处理后续的事件。虽然增加了控制面的启动耗时，但相比于服务的可靠性的提升，这个代价还是值得的。

我们也在 1.10 中引入了 DiscoveryNamespacesFilter，在控制面上忽略一些不需要关心的 namespace，加快事件的处理。控制面冷启动的过程中，也发现使用 DiscoveryNamespacesFilter 存在潜在的风险，也向 istio 社区提交了 PR https://github.com/istio/istio/pull/36628，目前已经合入。

```
func (c *Controller) SyncAll() error {
	c.beginSync.Store(true)
	var err *multierror.Error
+	err = multierror.Append(err, c.syncDiscoveryNamespaces())
	err = multierror.Append(err, c.syncSystemNamespace())
	err = multierror.Append(err, c.syncNodes())
	err = multierror.Append(err, c.syncServices())
	return err
}

+func (c *Controller) syncDiscoveryNamespaces() error {
+	var err error
+	if c.nsLister != nil {
+		err = c.opts.DiscoveryNamespacesFilter.SyncNamespaces()
+	}
+	return err
+}
```

3.6 灰度发布

内部在不停的优化控制面和数据面，所以也需要经常发布。随着应用接入的规模变大，更多的核心的应用接入的，传统的滚动更新也无法满足需求，更细力度的灰度发布也变得尤为重要。虽然每次发布都会经过内部多个环境的验证，最终才上线生产，但是也存在不可控的因素，变更就会带来风险，所以需要做到可灰度可快速回滚。

（1）Canary 发布

- 在同一个集群中部署一组 Canary 的控制面。
- 然后调整 Sidecar 注入策略，将部分 Sidecar 接入 Canary 的控制面。

- 通过自动化的方式进行验证，需要提前梳理测试的场景。

（2）Canary 之后会再灰度发布

- 创建一组控制面实例，逐步扩容，逐步接入流量，开始观察。
- 此时回滚就是直接缩容新实例，数据面会快速重新连接到老版本控制面上，从而快速回滚。
- 观察稳定之后，逐步缩容老的服务。

四、未来展望

Service Mesh 在流量管控领域具有重大的意义，不仅可拓展性强，还可以统一流量管理模型，统一多语言的异构系统。

未来我们还会持续投入，围绕可观察性，进一步提升服务可靠性，优化 xDS 的推送性能，支持内部大规模落地。同时增加与社区的沟通，期待与大家共同成长。

携程 Service Mesh 性能优化实践

【作者简介】本文作者佐思、烧鱼、Shirley 博，来自于携程 Cloud Container 团队，主要从事 Service Mesh 在携程的落地，负责控制面的可用性及优化建设，以及推进各类基础设施服务的云原生化。该团队负责 K8s 容器平台的研发和优化工作，专注于推动基础设施云原生架构升级，以及创新产品的研发和落地。

一、背景

为了支撑业务的高速发展，从 17 年开始，携程内部逐步推进应用容器化改造与业务上云工作，同期携程技术架构经历了从集中式单体应用到分布式微服务化的演进过程。

随着 Kubernetes 的不断发展和推广，服务网格（Service Mesh）在近几年也变得很流行。而 Servive Mesh 之所以越来越受欢迎，在提供更丰富的服务治理、安全性、可观测性等核心能力外，其从架构设计层面解决了以上几个痛点，服务治理能力以 Sidecar 的模式下沉到数据面，解决了 SDK 升级及多语言的问题，对于像负载均衡、熔断、限流等策略配置，由控制面统一管理和配置，并下发到数据面生效。在整体架构上云技术方案选型上，权衡各类方案的功能完备性、架构扩展性、改造维护成本及社区发展等，最终选择基于 Istio 构建 Service Mesh 平台治理方案。

1.1 携程 Service Mesh 发展

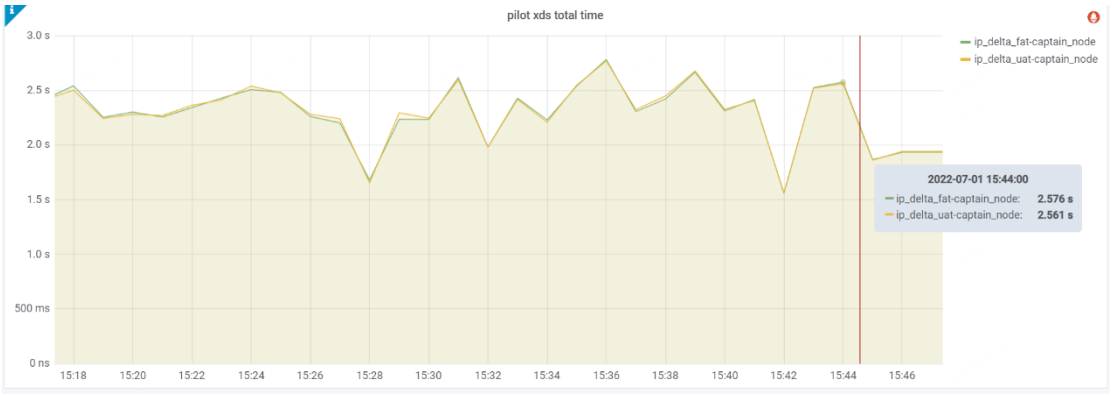
从 2020 年中，我们依托 K8S 底座能力，进行 Service Mesh 技术预研，深度定制 Istio，并与携程框架部门合作进行了小规模的落地试点。2021 年底，接入非核心应用 600+，为 Service Mesh 在携程的最终落地奠定基础。到目前为止，生产环境已有 2000 个应用（业务 POD 数近 1W）接入，预期下半年推进核心应用的接入。

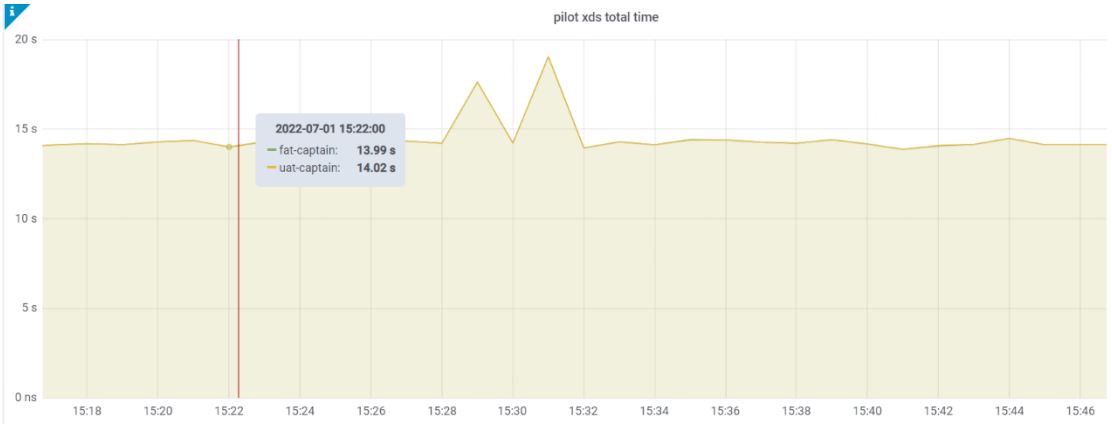
1.2 携程 Service Mesh 数据表现

在前期应用接入过程中，针对 Istio 稳定性（主要在性能）方面，梳理了以下几个问题：

- 控制面并发性能：pilot 对象处理的并发性能是否满足平台需求？
- 控制面配置下发时效性：配置下发的延迟及准确性是否能够满足业务需求？

本文主要分享在当前的体量下，回答上述问题，使控制面平稳支撑大规模 Sidecar 的落地，通过下述优化之后，测试域（Istio CR 量级在 1W+）如下图所示：





- 从实际生产来看，ServiceEntry 的处理效率提升了 15 倍左右；
- 从测试域来看，整体 initContext 时延从原先 P99 30s 左右到目前 P99 5-10 秒左右；
- 从测试域来看，整体优化水平从原先 P99 30s+到目前的 P99 15s 左右（该处为全量推送水平，其中 15s 的结果是平衡资源使用与推送效率调参的目标值，这里 PILOT_DEBOUNCE_AFTER 设置为 10s）；
- 开启增量推送后，实例推送在测试域的推送效果从 10-30s 缩减至 2.5s 左右。

二、Service Mesh 优化的思路与挑战

2.1 现状

针对 Service Mesh 在携程落地的服务目标，可以用一句话进行总结：能够通过横向扩展，支撑万级业务服务。为了完成上述目标，团队面临以下挑战：

- 当前 Istio 的对象处理性能等方面无法满足平台需求；
- 配置下发的延迟及准确性无法满足业务需求。

经过前中期，针对 Istio 架构进行深入研究以及上线前期测试的性能预研，核心问题聚焦在以下几点：

- istio 对象处理性能较低：在处理 ServiceEntry 时的并发性缺失及 WorkloadEntry Selector 模式的计算高耗时等；
- istio 配置推送性能较低：配置推送时对象的全量处理拉长下发时延，并会随着 Istio 对象增长而近线性增长。

2.2 优化实践

接下来主要分享携程所经历过的性能问题，和对应的优化的方向：

（1）对象处理性能

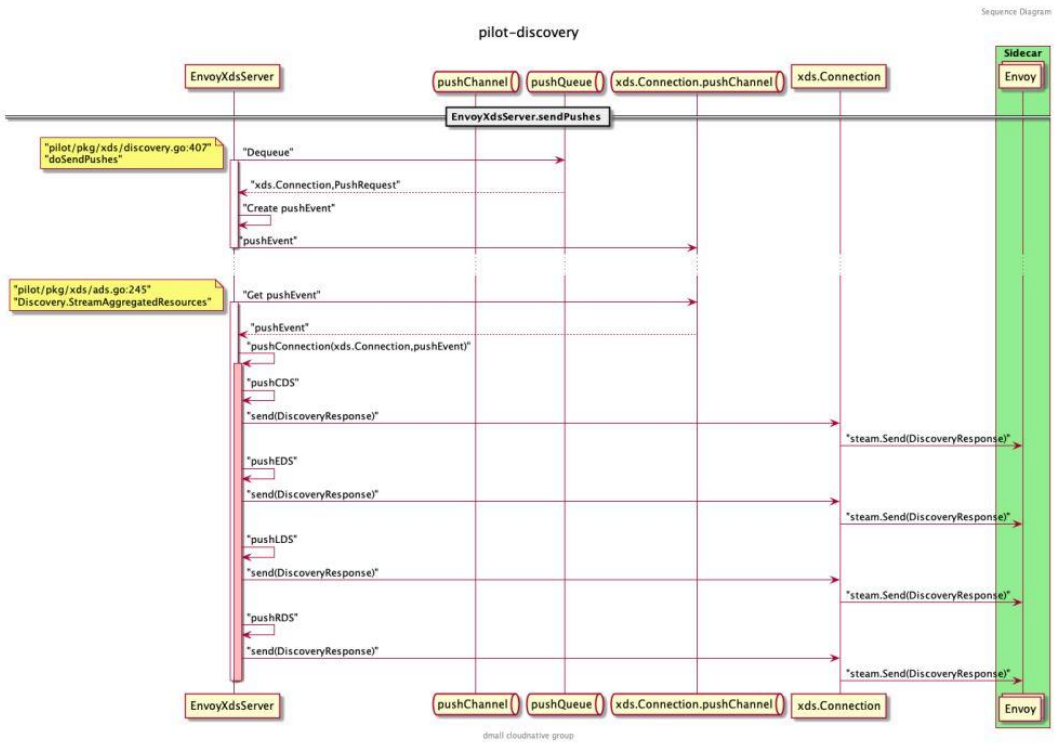
当前 istio 使用内部 queue 处理各类 Object 事件，其为线性处理流程，效率比较低下，为此社区提供 namespace filter 方式进行处理，以减缓性能问题，但针对 Istio 相关对象，未实现基于 namespace 的隔离，因此效率提升不太符合预期。

（2）推送性能

xDS 是 istio 控制面和数据面 envoy 之间的通信协议，其中 x 表示多种协议的集合，可以简单的把 xDS 理解为，网格内的服务发现数据和治理规则的集合。xDS 数据量的大小和网格规模是正相关的。

当前 istio 下发 xDS 使用的是全量下发策略，也就是网格里的所有 sidecar，内存里都会有整个网格内所有的服务发现数据。在大量服务实例的情况下，全量下发会影响 Pilot 和 Sidecar 的性能和稳定性，虽然 Istio 在不断的演进过程中引入了一些 scoping 的机制，就是 Sidecar 这个 CRD，这个配置可以显式的定义服务之间的依赖关系，但该 scoping 方案还是无法达到业务侧的推送延迟预期。

首先，简要介绍一下 Istio 推送的过程：



根据上图结合源码可知：

StreamAggregatedResources 会和当前的 Proxy 创建一个连接，并创建一个接受请求的 reqChannel 。同时开启一个新的协程 receiveThread 处理客户端主动发起的请求，期间调用 s.globalPushContext().InitContext(s.Env, nil, nil)进行数据初始化，其中 InitContext 需要处理 Istio 所有 CR 的全量数据（如 VirtualServices、DestinationRules、EnvoyFilters 和 SidecarScopes 等），该操作耗时较长，因测试环境上述对象量级在 2w 左右，导致执行耗时 P99 在 28s 左右。

从 con.pushConnection 中获取到 pushEv 事件后，调用 s.pushConnection() 进行处理，判断是否为全量推送：

```
if pushRequest.Full {
        // Update Proxy with current information.
        s.updateProxy(con.proxy, pushRequest.Push)
}
```

其中 updateProxy 更新 proxy 当前信息，主要逻辑如下所示：

```
func (s *DiscoveryServer) updateProxy(proxy *model.Proxy, push *model.PushContext) {
    s.setProxyState(proxy, push)
    if util.IsLocalityEmpty(proxy.Locality) {
        ...
        if len(proxy.ServiceInstances) > 0 {
```

```
                proxy.Locality =
util.ConvertLocality(proxy.ServiceInstances[0].Endpoint.Locality.Label)
            }
        }
}

func (s *DiscoveryServer) setProxyState(proxy *model.Proxy, push *model.PushContext) {
        proxy.SetWorkloadLabels(s.Env)
        proxy.SetServiceInstances(push.ServiceDiscovery)
        ...
        proxy.SetSidecarScope(push)
        proxy.SetGatewaysForProxy(push)
}
```

在 setProxyState 方法中的环节获取 SidecarScope 等相关信息。针对上述介绍可以明确下面几个优化方向：

虽然 Istio 针对 K8S 对象实现了基于 namespace 级别的隔离，但未对 Istio CR 对象实现 namespace 级别隔离

在 InitContext 方法中，Push() 这么慢，主要是在 req.Full 做全量推送的时候，需要初始化 PushContext ，初始化 PushContext 的过程中需要调用 initServiceRegistry、initEnvoyFilters 和 initSidecarScopes 等，耗时巨大

2.2.1 Pilot 性能优化

（1）资源基于 Namespace 隔离

虽然 Istio 针对 K8S 对象实现了基于 namespace 级别的隔离，但未对 Istio CR 对象实现 namespace 级别隔离，基于此，内部团队针对 Istio 1.10.3 版本针对 Istio CR 对象实现 namespace 隔离，使其影响范围控制在指定 namespace 下，其他用户操作 Istio CR 而彼此互不干扰，且能极大缩减 Istio Event 事件的处理，加速 Pilot 启动速度，提升事件处理效率，促进配置下发效率，在 CR Client 结构体中，新增 namespaceFilter 等相关字段，定义如下：

```
// Client is a client for Istio CRDs, implementing config store cache
// This is used for CRUD operators on Istio configuration, as well as handling of events on
config changes
type Client struct {
        ...

        namespaceInformer v1.NamespaceInformer
        namespaceFilter     filter.DiscoveryNamespacesFilter
        ...
}
```

截至目前为止，携程 Mesh 平台主要分为 SLB、SOA 两大 namespace，基于 namespace 隔离之后，效率提升预估在 50%左右。

（2）ServiceEntryStore 改造

ServiceEntryStore 的数据处理性能问题，主要有以下几点：

- 它里面有一个步骤，会全量更新实例的索引，这意味着 service 有一个发生变化了，它会更新全部 service 的索引，这是一个量级写放大。
- WorkloadEntry 与 ServiceEntry 的关联查询的耗时，随着彼此的数量逐步放大。
- configController 的 Queue 队列为线性处理，效率低下。

因此，携程通过针对 ServiceEntryStore 进行 Controller-Runtime 改造，将 ServiceEntry 对象由线性处理改为并发处理，同时将 WorkloadEntry 结构体废弃，选择直接使用 ServiceEntry，业务方 Operator 管理 ServiceEntry 对应 Endpoint 方式，优化处理性能，从实际生产效率来看，ServiceEntry 的处理效率提升了 4 倍左右。

（3）EnvoyFilter 增量化改造

通过上述简介可知，Push() 这么慢，主要是在 req.Full 做全量推送的时候，需要初始化 PushContext ，初始化 PushContext 的过程，都是全量且嵌套循环处理，因此当多个对象量级巨大，则计算耗时成倍增长，针对 EnvoyFilter 的全量处理，不涉及其他对象，可以通过定义 EnvoyFilterController 结构体以 Controller 方式运行，从而实现全量改增量，结构体定义如下：

```
type Controller struct {
    xdsUpdater model.XDSUpdater
    client        kube.Client
    queue         controllers.Queue

    // processed ingresses
    envoyFilter map[types.NamespacedName]*wrapEnvoyFilterWrapper

    envoyFilterInformer cache.SharedInformer
    envoyFilterLister     v1alpha3.EnvoyFilterLister

    mutex sync.Mutex

    envoyFiltersByNamespace map[string][]*wrapEnvoyFilterWrapper
}
```

（4）Sidecar 延迟及按需计算

在 InitContext 方法中，除了 EnvoyFilter 耗时较多外，initSidecarScopes 同样耗时巨大，通过

代码可知,Sidecar 有两种，一种是带 WorkloadSelector 的，一种是不带的。不带 Selector 的话就是对这个命名空间所有服务生效。如果没有手动创建默认的 Sidecar，Pilot 会通过 DefaultSidecarScopeForNamespace 为当前命名空间创建一个默认的 Sidecar，会将网格中所有的服务都遍历一遍，写入 SidecarScope 中。initSidecarScopes 循环计算如下：

sidecar 数量 x(egressConfigs 数量 x(selectVirtualServices 耗时 +selectServices 耗时)+out.EgressListeners 数量 x(listener.services 数量+listener.virtualServices 数量...))
因 SidecarScope 涉及其他 CR 对象结果，因此无法简单的由全量改增量，但可以通过延迟计算和按需计算方式，进行效率提升，延迟计算主要通过将 initSidecarScopes 计算逻辑后移至 push 阶段，按需计算即没必要计算所有 Sidecar，只需要根据链接的 proxy 进行计算即可，通过上述的优化，可以做以下针对性调整：

如果集群内服务较多，为每一个应用创建一个 sidecar，防止所有服务信息推送给 envoy，导致 envoy OOM。

在上述优化之后，InitContext 的处理耗时可以从 P99 30s 下降到 P99 5s 左右，此时，配置推送效率得到 5 倍左右的提升，那么 setProxyState 处的耗时，将会被放大，CPU 的使用率将会成倍增长，可以通过下述配置进行优化。

2.2.2 Pilot 配置优化

（1）启用 XDS 增量推送

通过给 istiod 配置 PILOT_ENABLE_EDS_DEBOUNCE 环境变量，我们启用 istiod 的增量推送而无需等待 full push。

（2）减少推送量

将 istiod 的 PILOT_FILTER_GATEWAY_CLUSTER_CONFIG 环境变量配置为 “true”，这样 Istio 将仅推送 Gateway 所需的服务信息，这个配置将极大的减少每次推送的量。开启这个特性之后，集群内的 istiod 每次向 Gateway 推送的服务信息降低 90%。

（3）关闭 Headless

将 istiod 的 PILOT_ENABLE_HEADLESS_SERVICE_POD_LISTENERS 环境变量配置为 “false”，因为 headless svc 对应的 endpoints 发生了变化，会触发 full push 的行为。

（4）提高吞吐

默认情况下，单个 istiod 的推送并发数只有 100，在较大的集群内，可能会导致配置生效的延迟。istiod 环境变量 PILOT_PUSH_THROTTLE 可以配置这个并发数。建议根据集群规模进行配置。

（5）避免频发推送

PILOT_DEBOUNCE_AFTER 与 PILOT_DEBOUNCE_MAX 是配置 istiod 去抖动的两个参数。

默认配置是 100ms 与 10s ，这也就意味着，当集群中有任何事件发生时，Istio 会等待 100ms，如果开启 EDS，则增量推送不会等待。

若 100ms 内无任何事件进入，Istio 会立即触发推送。否则 Istio 将会等待另一个 100ms，重复这一操作，直到总共等待的时间达到 10s 时，会强制触发推送。实践中可以适当调整这两个值以匹配集群规模和实际应用。携程内部调高 PILOT_DEBOUNCE_AFTER 到 10s，以避免频繁推送对性能产生影响，也能够避免极端情况下推送不及时导致的 503 问题。

三、Service Mesh 未来展望

控制面的重心在于解决规模化问题，后续控制面将会在下述领域深入探索：

- 控制面去除对 k8s 的资源的依赖，推送耗时下降到秒级别，满足更大规模的接入；
- NDS 实现 DNS 解析功能，避免 search 域多次查询，提升 Mesh 的可用性。

团队将与社区深度合作，针对控制面，密切关注增量推送等特性，后续将优先实现控制面稳定性增强，如下述功能：

- 连接限流：通过限流功能，降低大量 Sidecar 同时连接同一个 Pilot 实例的风险，减少服务风暴发生的机率。
- 熔断：基于生产场景的压测数据，测算出单实例 Pilot 可服务的 Sidecar 上限，超过上限值后，新连接会被 Pilot 拒绝。

Service Mesh 作为云原生领域下一代微服务技术，经过 2 年多摸索与演进，携程完成了多语言、多场景的业务落地，实际论证了 Service Mesh 在流量管控、系统扩展性的优势，具有下沉服务治理能力到基础设施层，高度解耦中间件与业务系统的可行性。

后续，携程将在总结前期非核心应用 Service Mesh 化改造的基础上，逐步推进核心应用的落地，同步打磨完善平台能力，全面提升稳定性，为行业落地 Service Mesh 提供最佳实践和相关借鉴。

四、参考资料

- 百度大规模 Service Mesh 落地实践
- 不是所有的应用都需要 Service Mesh
- 控制面核心组件
- istio 在知乎大规模集群的落地实践
- Istio Pilot 源码分析（三）

运维

携程基于 DPDK 的高性能四层负载均衡实践

【作者简介】Yellowsea，携程资深技术支持工程师，负责四层负载均衡研发及私有云 k8s cloud provider 开发，关注 Kubernetes、Linux Kernel、分布式系统等技术领域。

前言

在携程的服务流量接入架构中，一般是采用四层负载均衡与七层负载均衡相结合的方式，其中四层负载均衡支撑着业务运行的关键部分。在业务流量不断增长的过程中，不断考验着四层负载均衡的性能及可靠性。由于原硬件四层负载均衡存在成本高、采购周期长、HA 工作模式等问题，原有的体系难以满足快速增长的业务需求，迫切需要在开源社区中寻找高性能四层负载均衡软件化的解决方案。

本文主要讲述基于开源的 DPVS 打造携程的高性能四层软件负载均衡 TDLB (Trip.com Dpdk LoadBalancer)，其极大的提升了设备的转发性能，具有高可靠、可扩展、易使用等特性。

一、TDLB 高性能实现

传统 LVS 负载均衡的功能与硬件设备类似，但由于其性能存在瓶颈，难以满足携程四层负载均衡服务的实际业务场景需求。DPVS (https://github.com/iqiyi/dpvs) 结合 DPDK 的特性，解决了 LVS 的性能瓶颈，同时又能满足原有的负载均衡服务需求。

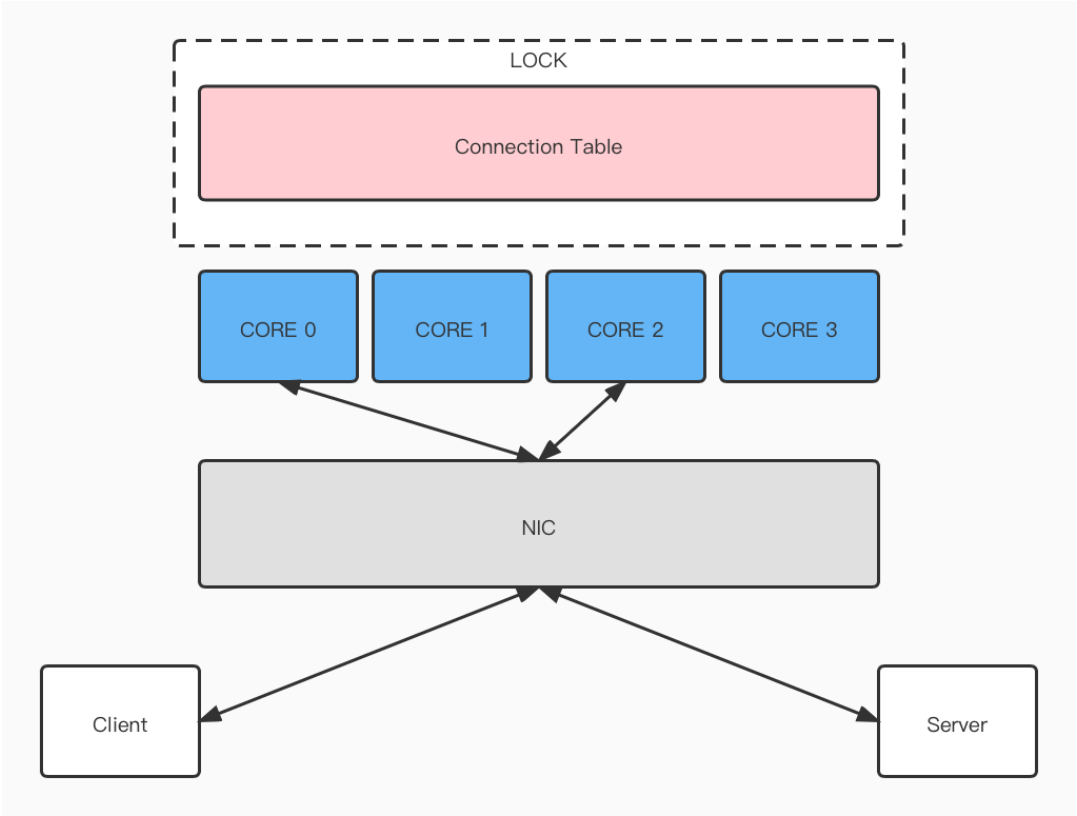
1.1 DPDK

在内核中，从网卡获取数据包是通过硬件中断模式完成，内核态与用户态的切换耗时，而且切换导致 cache 命中率下降，影响处理数据包的性能。

在 DPDK 中采用 kernel bypass 的设计，通过应用程序主动轮询的方式从网卡获取数据包，使应用程序维持在用户态运行，避免内核态与用户态切换的耗时问题，提升处理数据包时 cache 的命中率。

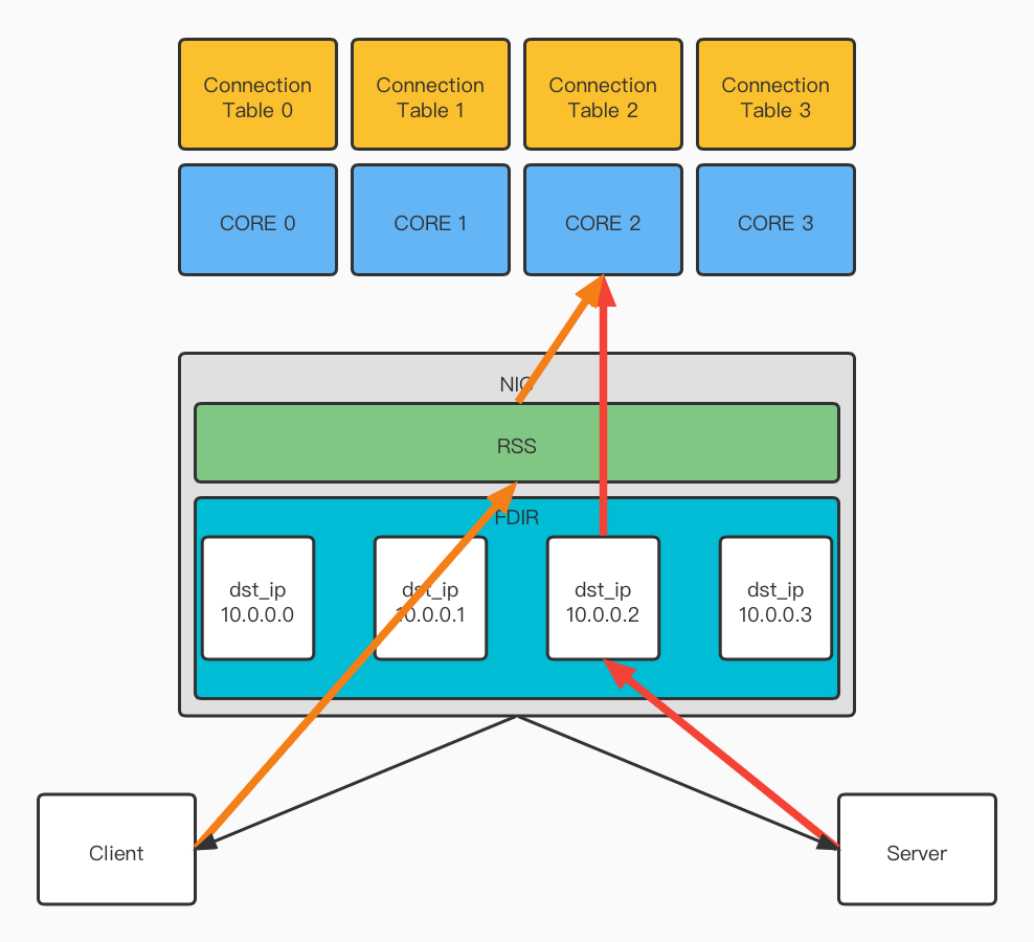
1.2 会话无锁

初期的 LVS 采用全局的会话资源供多个 core 同时使用，多个 core 之间会产生资源的竞争，带来了锁的问题。



TDLB 主要使用 fullnat 模式，为了避免 core 与 core 之间的资源竞争，我们设计了 percore 的会话，作为有状态的四层负载均衡必须保证入向流量与出向流量分配至同一个 core，出向流量才能匹配上原先建立的会话。

入向流量可以利用 RSS 将数据包散列至各个队列，而每个 core 绑定对应的队列，对于相同的数据包 (sip,sport,dip,dport) RSS 会被分配至同一 core。为了保证出向回程流量还经过原先的 core 可以为每个 core 分配不同的 SNAT IP，在 fullnat 模式中，client IP 会被转化成 SNAT IP，到达 server 后，server 回应报文的目的 ip 就是原先的 SNAT IP，此时可以借助网卡的 FDIR (Flow Director) 技术来匹配 SNAT IP，将回程报文分配至对应的 core，确保数据流的出向流量与入向流量分配至同一 core。



1.3 用户源 IP 透传

在 FNAT 模式中，后端服务器需要获取真实客户端 IP，目前主要有两种方式：

TOA：在 TCP 建连完成后传递的第一个数据包中，加入带有客户端信息的 TCP Option 字段，以此传递用户源 IP，后端服务可以在无感知的情况下获取用户源 IP，但是需要在服务器上挂载 TOA 相应的 kernel module。

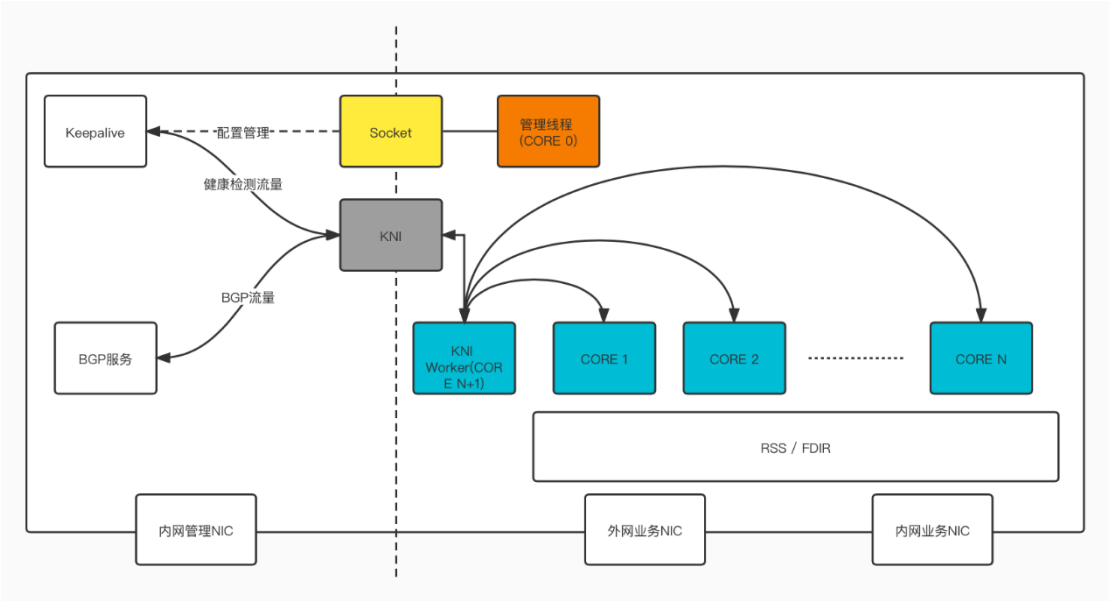
ProxyProtocol：在 TCP 建连完成后传递的第一个数据包的数据前，加入 ProxyProtocol 对应的报文，其中包含客户端信息，这部分报文通过单独的一个数据包发送，避免分片的产生，ProxyProtocol 可以同时支持 TCP 及 UDP，但是需要在服务的应用层提供 ProxyProtocol 的支持

DPVS 沿用 LVS 中的 TOA（TCP Option Address）的方式传递用户源 IP，TDLB 在 DPVS 的基础上增加了对 ProxyProtocol 的支持，以此满足不同服务场景的需求。

1.4 日志异步写入

在 DPDK 原日志存储机制中，当有大量日志需要记录时，单个文件 I/O 锁带来的耗时将影响各个 CPU 的数据包处理，严重时将影响控制平面流量并导致 BGP 连接断开。DPVS 在此基

础上加入消息处理机制，各个核产生的日志将进入消息队列，由日志处理的核单独处理，保证 I/O 不会受到锁的影响。

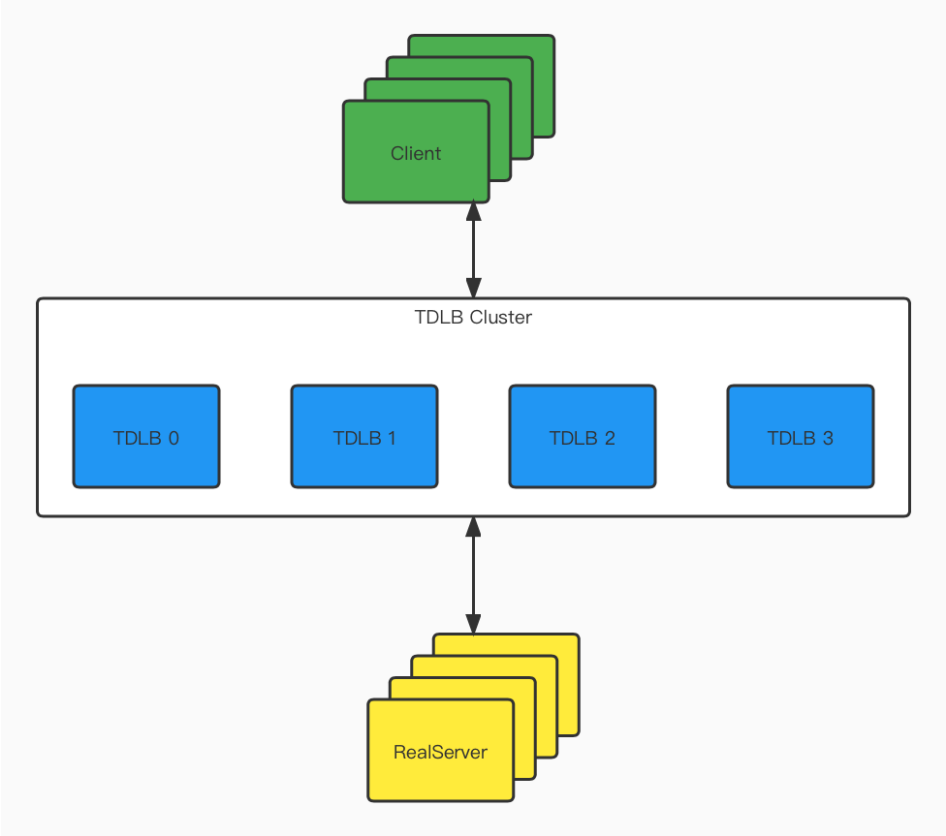


在 DPVS 框架基础上，技术经过初步的消化吸收，建立了 TDLB 的雏形。为了适配携程的网络环境、业务场景及需求，进一步提升单机及集群的可靠性，TDLB 主要从以下五个方面进行进一步的改造：

- 集群多活模式
- 资源隔离
- 集群配置管理
- 健康检测策略
- 多维度监控

二、集群会话同步

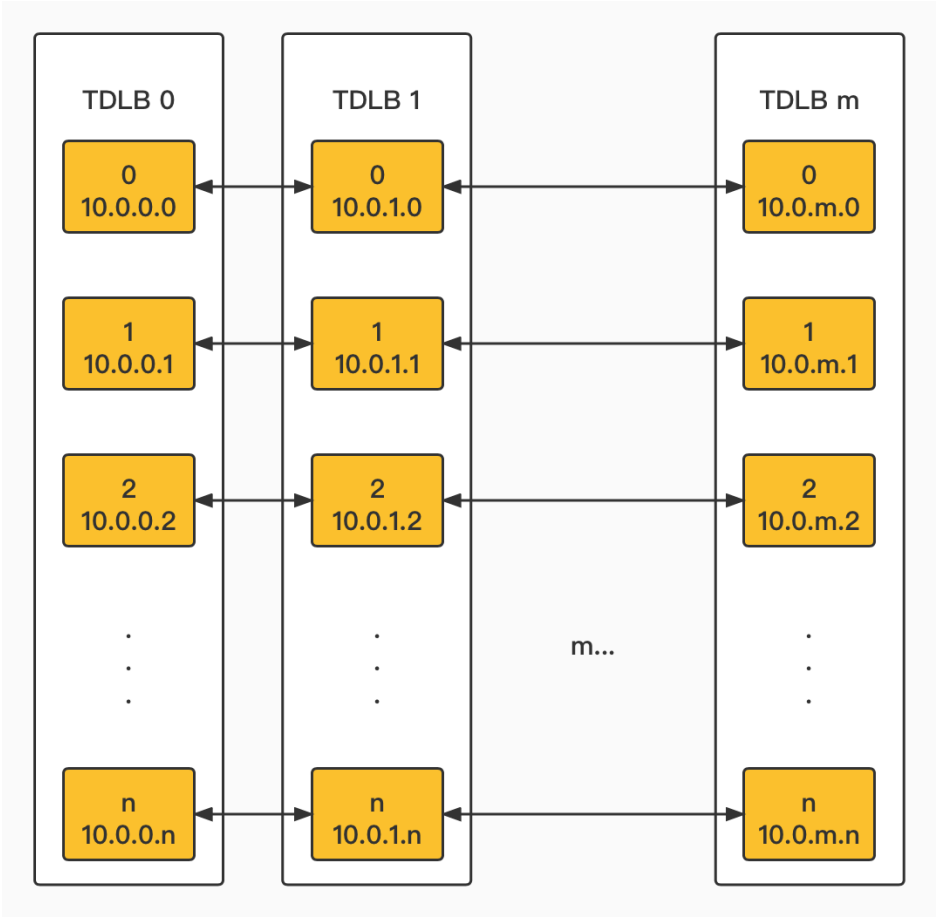
在集群多活模式下，服务器的扩缩容会导致路由路径的重新分配，没有会话同步功能支持的情况下，已有的连接会失效并导致应用层超时。这对长连接的应用影响更加明显，影响集群的可用性，无法灵活的扩缩容应对业务高峰。



2.1 同步策略

核与核间的会话信息资源已被隔离，同时入向业务流量是通过 RSS 进行分配的，所以在集群中服务器网卡 RSS 配置一致的情况下，同一网卡队列编号对应核的会话信息可以共用。

为了做到多台服务器核与核间的信息交互，为每个核单独分配一个内网 IP 地址（FDIR），用于转发数据包至后端服务器的同时（SNAT IP），用于会话信息同步的 Source Address 及 Dest Address。



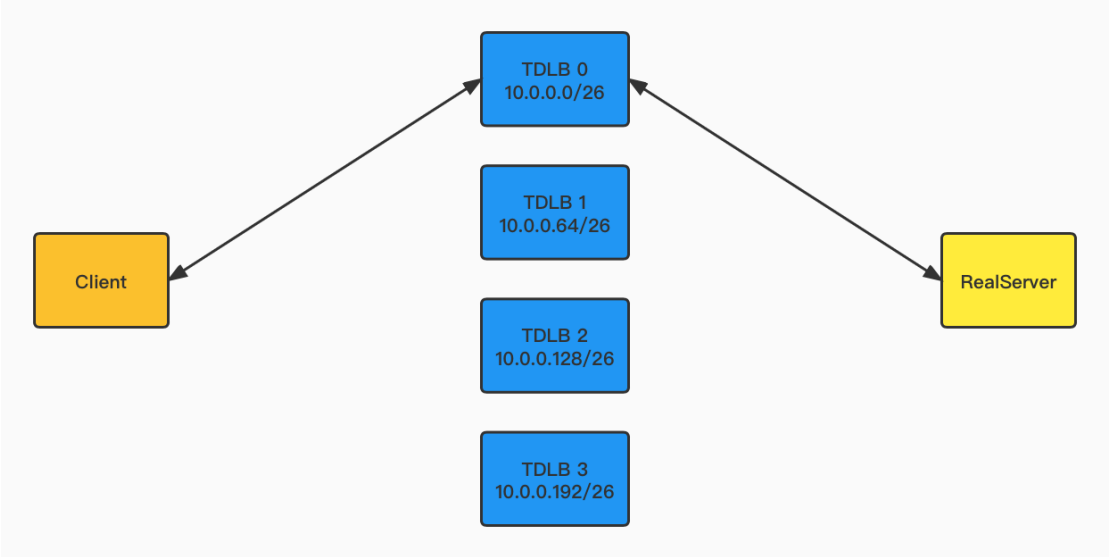
在此基础上，TDLB 集群使用的 SNAT Pool 需要一个连续的网段，每台 TDLB 服务器分配一个同等大小的子网，同时会利用 BGP 同时宣告三个子网掩码不同的 SNAT Pool 网段路由：

- 网卡对应 CPU 分配的子网段（32-log(n/2)）
- 服务器分配的子网段（32-log(n)）
- TDLB 集群的 SNAT Pool 网段（32-log(n*m)）

以此保证一台 TDLB 服务器拉出集群后，其他 TDLB 服务器可以接收到已有连接中 RealServer 的数据包，保证已有连接不会断开。

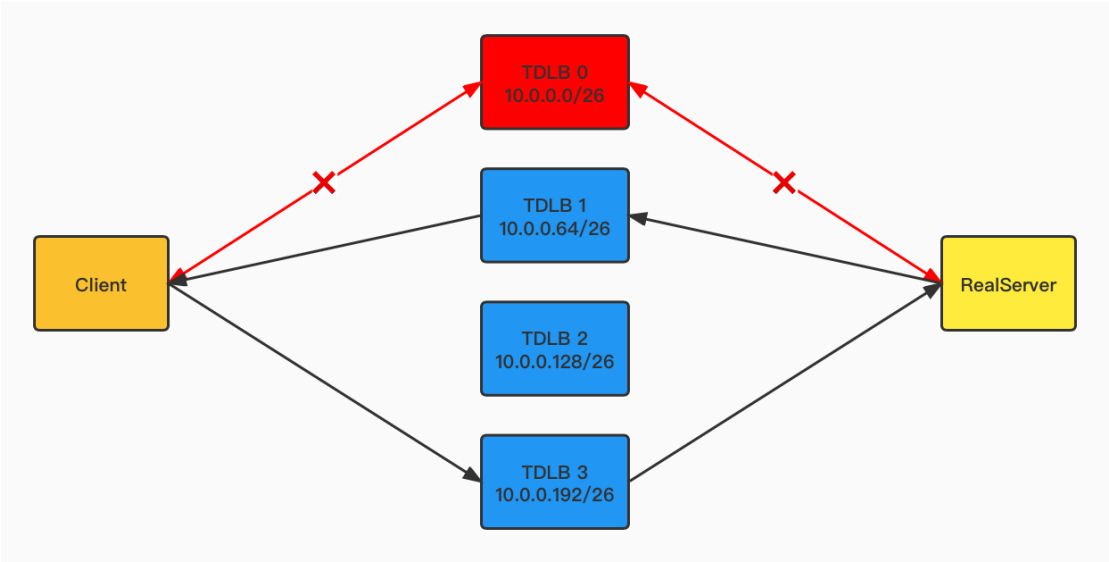
（1）服务器正常情况

当 Client（CIP: 1.1.1.0, CPort: 5000）请求服务（VIP: 1.1.1.1, VPort: 80）时，路由到 TDLB 0 并通过 SNAT IP（LIP: 10.0.0.0, LPort: 6000）转发给 RealServer（RIP: 10.0.1.0, RPort: 8080），其中这个 TDLB 集群对应的 SNAT Pool 是 10.0.0.0/24。



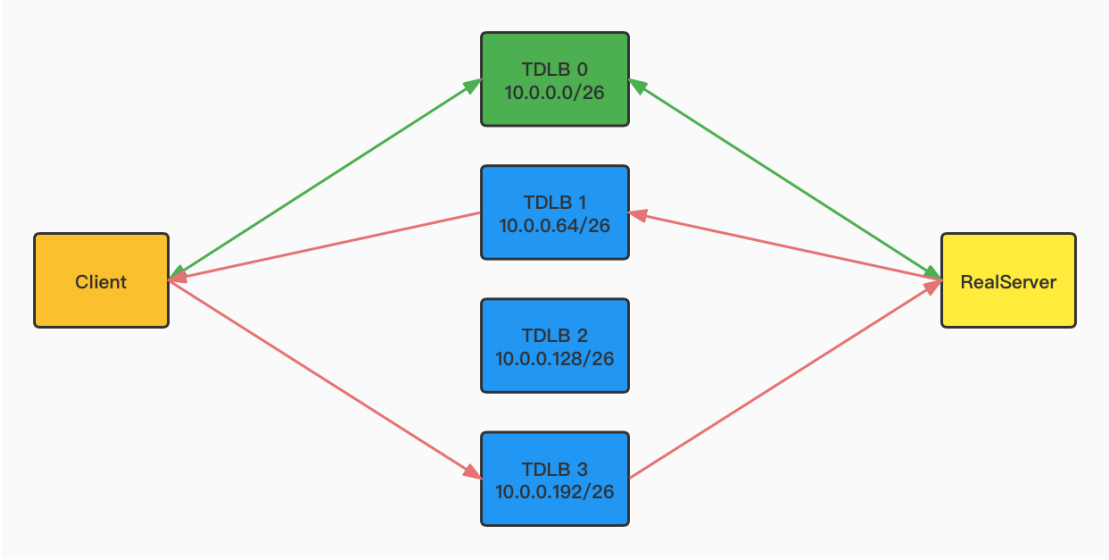
（2）服务器异常情况

当 TDLB 0 拉出集群（停止 BGP 宣告路由）时，Client 的请求被重新路由到了 TDLB 3，由于查询到了同步的会话信息，因此使用相同的 SNAT IP（10.0.0.0）转发给相同的 RealServer（10.0.1.0），RealServer 回复请求时被重新路由到了 TDLB 1 再转发给 Client。



（3）服务器恢复工作

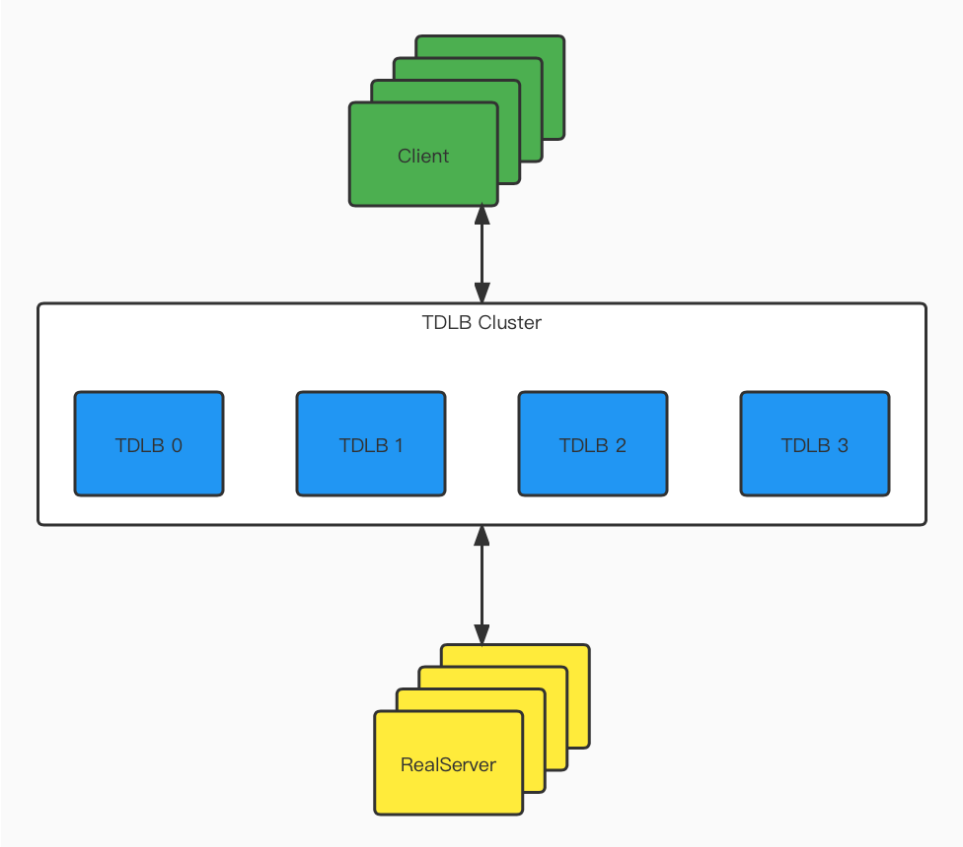
当 TDLB 0 拉入集群恢复工作后，原本属于 TDLB 0 的会话会被重新分配给 TDLB 0。



2.2 同步类型

为满足服务器处于不同状态下的会话同步，同步类型分为两种：

- 增量同步：在新的连接建立时进行，并随着数据的传输保持连接的状态同步；
- 全量同步：在新的服务器加入集群时，需要全量同步会话信息，保证接入流量时不对已有连接产生影响。



在支持会话同步的基础上，TDLB 集群可以采用多活模式进行灵活的水平扩缩容，一方面可以根据业务的需求调整集群服务器数量降低成本，另一方面可以在用户无感知的情况下完成 TDLB 集群的维护。

三、资源隔离

3.1 CORE 与 CORE 之间的数据隔离

利用网卡的 RSS，FDIR 等流控技术，将数据流分配至同一 core，保证了 core 处理数据流时不需要用到全局资源，避免了资源竞争带来锁的问题。处理数据流需要使用的相关资源可以在初始化时，为每个 core 单独分配资源，利用消息处理机制保证 core 与 core 之间的信息同步。

3.2 NUMA 架构下 CPU 数据隔离

目前服务器主要采用 NUMA 架构，CPU 与 CPU 之间跨 NUMA 访问数据在一定程度上限制了应用的性能，在 TDLB 设计的目标中尽可能避免跨 NUMA 访问数据的情况。

在上述设计中，为了达到会话无锁的目标，会话流量已经被限定在同一个队列、core 中，这已经为资源隔离打下了基础。在负载均衡服务处理中，高频访问的资源有网卡配置、VS 配置、地址、路由、会话表等，四层的会话处理相关资源已被隔离，网络协议栈中的相关资源与硬件资源相关，因此根据 NUMA 架构中 CPU 的数量各分配独立的硬件网卡资源即可，在做到资源隔离的同时，充分利用 CPU 的工作能力，提高单台服务器的负载能力。

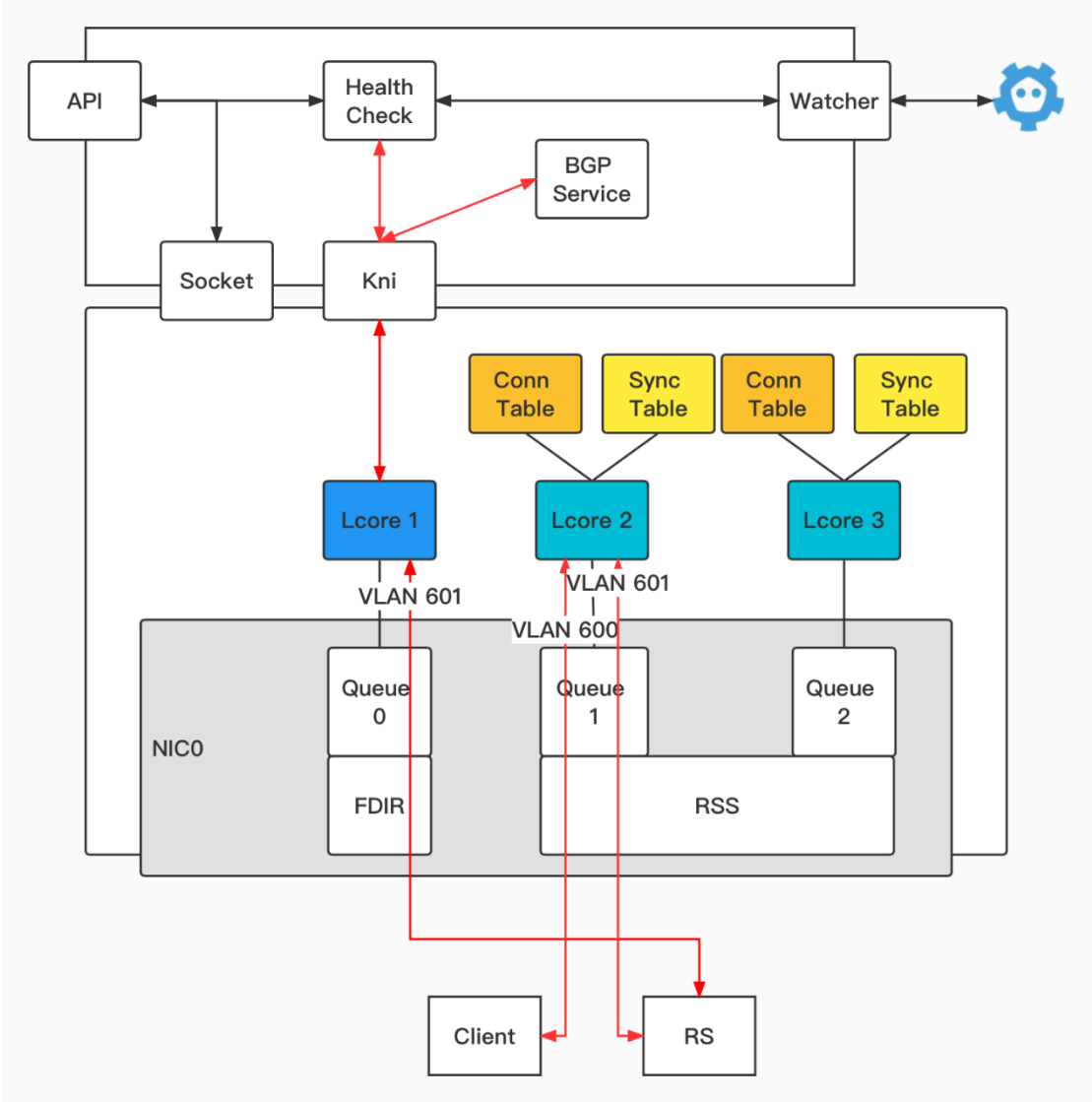
3.3 控制平面与数据平面流量隔离

TDLB 的流量主要分成两部分：

- 控制平面流量：BGP、健康检测及二层信息交互
- 数据平面流量：负载均衡服务

在初期设计中，控制平面流量需要通过数据平面的相关工作核处理后进入 KNI 队列中，当业务负载超出阈值时将影响到 BGP 及健康检测服务，且混杂的流量增加了系统的复杂度，使控制平面的流量难以定位。

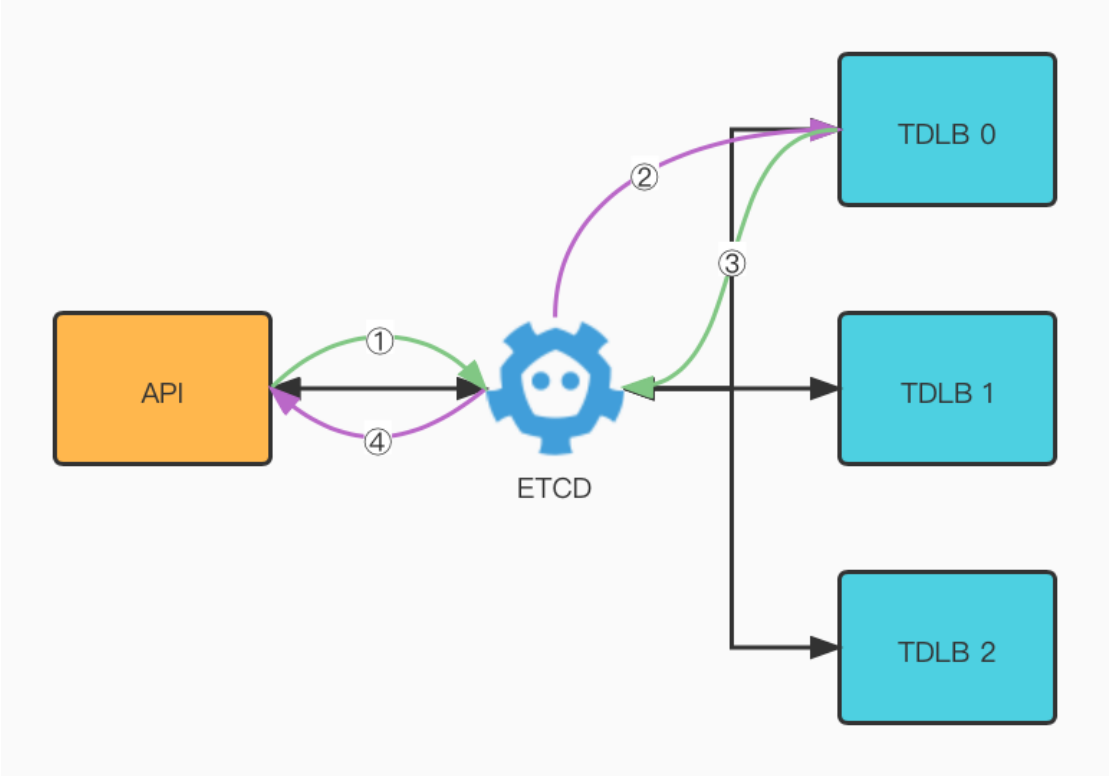
对原先的 RSS 配置进行修改，隔离出一个单独的队列，同时结合 FDIR 将控制平面流量导入隔离的队列中，实现控制平面与数据平面流量的隔离。



四、集群配置管理

在硬件设备以及 LVS 的集群管理中，都是通过 API 的方式与设备交互，进行配置的管理。当集群从主备模式转变为多活模式时，通过多线程的方式一定程度上可以保证配置的下发效率，但被动的配置更新无法保证服务器进入集群提供服务时配置的一致性，且每台服务器独立的 API 鉴权增加了控制的复杂度。在 k8s 中的 controller 及 reconcile 机制提供了解决方案。

使用 etcd 存储集群配置，每台 TDLB 服务器都会启动 operator 监听相关配置更新产生的事件，并通过 etcd 的证书鉴权保证配置的有效性。



- 在更新配置前，API 开始监听配置对应 key prefix 的事件，开启监听后更新 etcd 中的集群配置；
- 配置更新产生事件，TDLB 服务器上的 operator 捕获事件后进行相应的配置更新操作；
- 更新操作成功完成后，将配置版本回写至 etcd 相应的 key 中；
- 配置版本回写产生事件，API 监听到事件后进行统计，当集群所有服务器都更新完成，则操作完成。

每当 TDLB 服务启动后，都会与 etcd 进行配置同步，保证进入集群提供服务时配置的一致性。集群水平扩容服务器时，新进入集群的服务器通过这种同步机制即可完成配置的下发，与服务部署的自动化保证了水平扩缩容的效率，应对业务流量的增长。

五、健康检测策略

当一台负载均衡设备上存在多块网卡时，如果仅从一块网卡发起健康检测，当该网卡线路出现故障时，将影响到整台设备的服务，即网卡线路层面的故障升级到服务器层面。

为了控制网卡单点故障的影响范围，在同一时间，从每块网卡发起健康检测，并且将服务的部分配置按网卡进行分隔，当一条网卡的线路出现故障，仅影响线路对应的服务配置，其他网卡线路依旧正常工作，这样更好地保证了服务器剩余的工作能力。

六、多维度监控

在多活模式的集群中，需要从不同维度进行监控，提高故障的响应效率及定位效率：

- 集群维度：监控集群整体服务状态
- 服务器维度：监控单台服务器的服务状态以及集群中服务器间的差异
- 服务维度：监控每个服务的状态，便于用户对于服务状态的感知
- CORE 维度：监控每个 CORE 的工作状态，便于集群容量的评估

基于 DPDK latency stats 设计了 CORE 与 CORE 独立的 metric 计算与存储，从而高效的获取处理数据包耗时以及工作周期耗时的数据，监控单台机器以及整个集群的服务工作状态。

监控数据接入 prometheus 及 grafana，设置了各个维度的监控告警，帮助快速定位故障点。

总结

TDLB 作为以 DPVS 基础框架完成的高性能四层负载均衡软件，稳定运行近两年时间，支撑着携程各项业务。各项性能指标均符合预期，在满足业务需求的同时，大大降低成本，推进了四层负载均衡服务与私有云的接入，这也是拥抱开源社区的回馈。

携程监控系统 Hickwall 演进之路

【作者简介】大伟，携程软件技术专家，关注企业级监控、日志，可观测性领域。

一、背景

监控领域有三大块，分别是 Metrics，Tracing，Logging。这三者作为 IT 可观测性数据的三剑客，基本可以满足各类监控、告警、分析、问题排查等需求。

Logs：我们对于 Logs 是更加宽泛的定义，即记录事物变化的载体，包括常见的访问日志、交易日志、内核日志等文本型以及 GPS、音视频等泛型数据。日志在调用链场景结构化后其实可以转变为 Trace，在进行聚合、降采样操作后会变成 Metrics。

Metrics：是聚合后的数值，相对比较离散，一般有 name、labels、time、values 组成，Metrics 数据量一般很小，相对成本更低，查询的速度比较快。

Traces：是最标准的调用日志，除了定义调用的父子关系外（一般通过 TraceID、SpanID、ParentSpanID），一般还会定义操作的服务、方法、属性、状态、耗时等详细信息，通过 Trace 能够代替一部分 Logs 的功能，通过 Trace 的聚合也能得到每个服务、方法的 Metrics 指标。

近年来，可观测性这个概念如火如荼，可以看作是对监控的一次大升级。CNCF 也发布了 OpenTelemetry 标准，旨在提供可观测性领域的标准化方案。那么相比传统的监控告警，监控和可观测性有啥区别和联系呢？个人理解，可观测性能够以更加白盒的方式看透整个复杂的系统，帮助我们更好的观察系统的运行状况，快速定位和解决问题。

简单理解，监控和可观测性的关系。监控告诉我们系统的哪些部分是正常工作的，可观测性告诉我们那里为什么不工作了。监控侧重宏观，可观测性包括微观能力。监控是可观测性的子集。

近些年，随着携程集团对线上故障 1-5-10 目标的提出（即第 1 分钟发现故障，第 5 分钟定位故障，第 10 分钟解决故障），对监控系统提出了更高的要求。监控系统最重要的三个特点可以定义为，系统稳定性，数据及时性，数据精准性，三者缺一不可。

携程监控系统 Hickwall 是一个企业级的指标监控告警系统，兼容了业界的 Prometheus 监控标准，覆盖携程所有的指标监控数据，包括系统层和应用层。主要目标是实现指标数据的采集、接入、存储、展现，并在此基础上配置告警和通知，告警治理等，同时为第三方平台提供第一手的监控数据和告警事件。

二、遇到的问题

随着业务不断膨胀，系统规模的持续扩大，Hickwall 遇到了一些问题：

- 高基数查询，指标维度过多，导致整体查询慢，用户体验不佳。
- 云原生的监控方案缺乏，需要支持开源 PromQL 业界标准，Prometheus SDK 指标接入。
- 监控粒度粗，一些毛刺无法洞察，需要提高数据采样粒度。
- 告警系统多，技术方案杂，难维护，产品使用上用户到处找入口，规则和阈值定义不一样，很困惑。
- 监控数据延迟，导致误告警。
- 告警多，重复告警，缺乏治理。
- 容器大规模 HPA 带来的指标基数膨胀问题。

三、主要的演进

针对上述问题和痛点，Hickwall 过去两年进行了一些针对性的优化和演进。

3.1 云原生监控

（1）TSDB 升级，经过三次演进，现在是基于 VictoriaMetrics 实现的第四代的 TSDB 解决方案。完全兼容 Prometheus 查询语法。

（2）提供了自研 Beacon 容器监控组件，和 k8s 体系高度集成，不仅支持容器系统指标，JVM 指标，也支持自定义的 PrometheusSDK 埋点接入。

3.2 解决高基数问题

（1）产品升级，新增日志/指标预聚合能力，产品开放配置能力，根据一定配置策略，通过将多维原始数据降维，收敛指标维度，聚合输出预聚合数据，通过这种方式可以缩减指标量级，对后续链路的处理都有性能提升。目前系统配置了 166 条聚合规则，生成了 209 个指标。

（2）指标治理：监控统计指标维度，应用维度的高基数检测，对非法写入进行封禁。非法写入包括了 tag 的 value 使用了随机数，字符内容超过 256 个字符，指标名称使用了中文命名等。

（3）容量规划：做好集群的自监控，进行妥善的容量规划，主要是监控 ts 增长数量和 datapoint 数据量，以应对日益增长的指标数据。

（4）忽略有问题的 tag：治理平台能够按需配置 ignore tag，例如针对 HPA 场景下的应用埋点，忽略 value 容易发生变化的 hostname 和 ip 这两种 tag（一般不会关心这种维度），可以大大减少基数。

3.3 数据粒度提升

为了响应集团 1-5-10 目标，核心指标采集上秒级，目前主要涉及的是核心的系统指标，业务订单指标和部分应用指标，其他非关键可以按需自行配置。

3.4 告警中台接入

自研新一代统一的 pull 告警系统，统一各类老的告警技术方案，目前接入告警规则 10 万+，同时对接了告警中心，对用户提供一站式的告警治理能力。

3.5 解决数据延迟问题

数据延迟问题主要是数据链路还依赖了消费 Kafka 来写入 TSDB。因此我们将核心链路改造成最短路径，从数据网关分发数据直接写到 TSDB，从根本上解决了延迟问题。

3.6 时序存储的演进

Hickwall 存储这块主要经历了下面四个阶段。

第一阶段：ES 存储，Graphite 查询语法

第一个版本的架构主要以数据写 Kafka，消费 Kafka 进 ES 的套路来设计。这个方案的好处：

- 可以容忍比较大的系统 downtime
- 数据可以多次多种方式加以利用
- ES 存储的写入性能最大化
- ES 的聚合能力比较强，所以不少聚合都可以实时来做
- ES 非常稳定可靠，运维工作较少

第一个版本已经初步实现了监控系统的功能，但是在使用过程中同样暴露了一些问题：

（1）ES 存储导致数据容易堆积

ES 是一个非常稳定的全文索引工具，比较适合日志，搜索的场景。但是却不是最好的监控数据的存储方式，主要是写入性能不是很好，必须大批量，高等待的方式写才能达到比较大的量，但是这个比较大的量相对监控数据的场景也略显不够。

而且为了提高写的性能，还需要牺牲数据的实时性（提高 refresh time 来减少磁盘操作，提高写入量）。实时性又是一个高质量的监控系统所需要努力提高的。这就是一个矛盾点，虽然当时能够做到勉强接受，但肯定不是最理想的，当时的数据 latency 需要 30s 以上。

（2）数据链路过长

监控数据主要是从 Proxy 进来到 Trigger 告警需要依次经过 6 个组件，任何一个组件出现问题，都可能导致告警漏告或误告。

第二阶段：基于 InfluxDB 存储，打造自研的 Incluster 集群方案，Graphite 查询语法

ES 用于时间序列存储存在不少问题，例如磁盘空间使用大，磁盘 IO 使用多，索引维护复杂，写入和查询速度慢等。当时 InfluxDB 是排名第一的时序数据库，到 2017 年的时候已经比较稳定，所以我们萌生了用 InfluxDB 替换 ES 作为存储的方案。但是 InfluxDB 并没有开源的集群方案，因此我们自研了 Incluster 集群方案。

在元数据管理这块使用了 Raft 来保证一致性和分区容错性。集群大致的实现思路是，客户端通过 Incluster 节点写入数据，Incluster 按照数据分布策略将写入请求转发到相关的 InfluxDB 节点上，查询的时候按照数据分布策略进行数据读取和合并。在用户查询方面，实现了类 Graphite 语法用于配图，兼容上一代语法，从而可以减少用户迁移配图的成本。

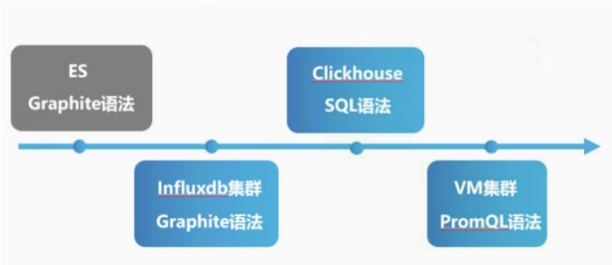
第三阶段：ClickHouse 列式存储，SQL 查询语法

2019 年，我们逐步开始推进应用埋点存储的统一接入。在这个阶段，InfluxDB 在高基数场景下，查询表现并不是很好，集群稳定性也受到了较大的挑战。因此我们调研了当时大火的 ClickHouse，开始接入应用埋点，并且提供 SQL 语法查询。携程的机票部门率先接入，在自定义应用埋点场景取得了比较好的效果。

第四阶段：基于开源的 VictoriaMetrics TSDB，PromQL 查询语法

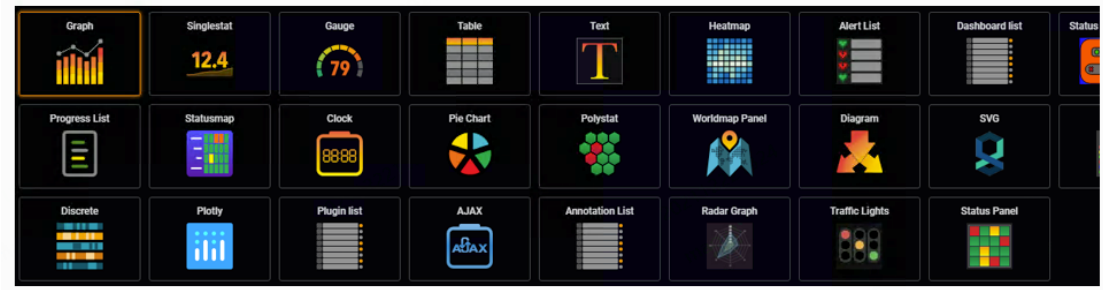
2020 年，随着云原生技术的发展，内部对云监控的需求越来越强烈。因此我们在 2020 年调研并测试了业界开源的 VictoriaMetrics TSDB，这款 TSDB 作为 Prometheus 的远端持久存储解决方案，提供了相较于传统 TSDB 较好的性能和天然兼容 Prometheus 协议的查询语法和接口。

这款 TSDB 经过测试在综合写入性能和查询方面表现较好。目前我们内部主要分了三个大集群，集群规模已经达百余台物理机，成为携程统一的 Metrics 存储方案。



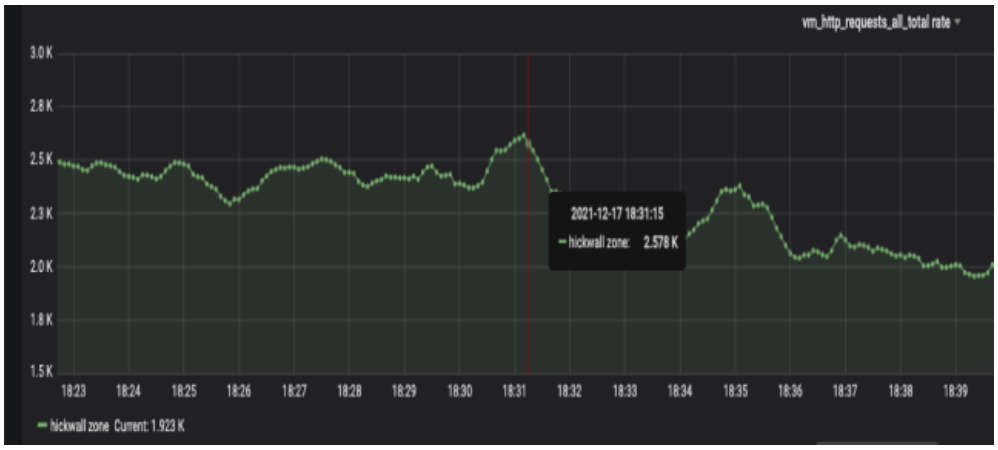
3.7 监控可视化的演进

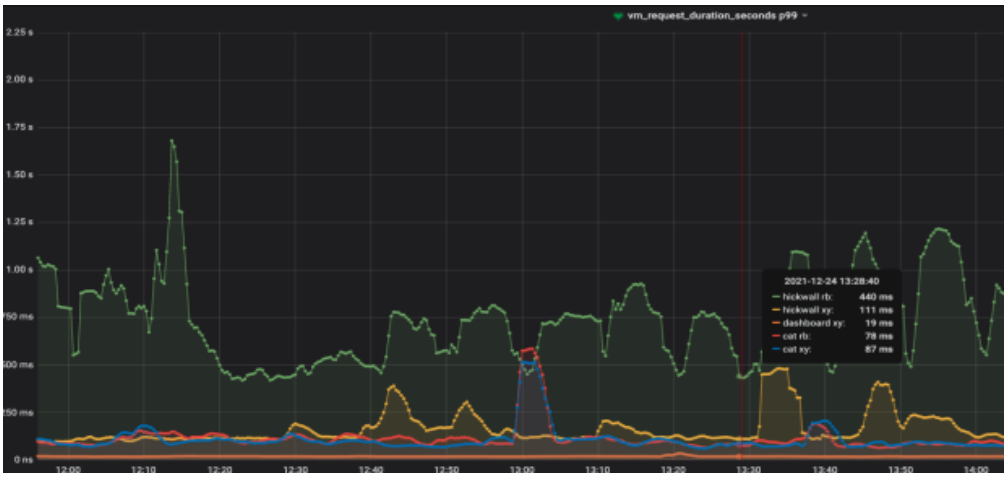
由于内部使用的可视化工具是基于 Grafana 二次开发，伴随着存储技术的升级，2020 年我们还进行了一次 Grafana2 版本到 6 版本的全面升级，新版本增加了多种新的数据源，所见即所得的告警能力，更多的图表类型展现，可视化方面大大提升了用户体验。



四、平台现状

随着多年的发展，目前平台指标数据量写入量峰值在千万级/秒，查询量数千 qps，接入各类告警规则 10 万+，查询 P99 控制在 1s 内。数据粒度最小支持到 10s 级，时序数据默认保存一年。计算+存储集群规模达百余台物理机，并且主要组件都上了 k8s 平台。数据统计如图 4 所示。



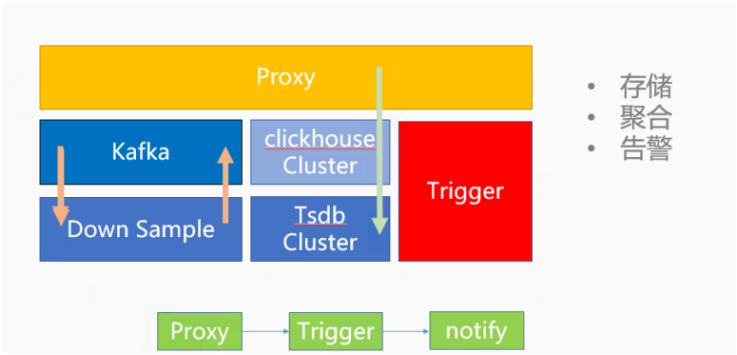


五、目前架构

从数据流向看，目前总体大致架构如图 5 所示，可见数据流和告警是走的最短路径。

- 数据：data->Proxy->TSDB
- 告警：data->Proxy->Trigger

这从根本上规避数据延迟和告警延迟问题。下面会主要介绍 Hickwall 所依赖的几个核心组件。



（1）Collector 组件：

数据采集，提供多种客户端，包括了 Hickwall SDK（应用埋点），Hickwall agent（机器数据采集），Prometheus SDK，Beacon（容器数据采集）。

（2）Proxy 组件：

提供给 Hickwall SDK，Hickwall agent，Prometheus SDK 的统一支持多协议的数据收集服务，主要是 thrift protocol 和 line protocol。作为数据接收的统一入口，承担了流量接入，分发，流量保护，数据统计等功能。Proxy 默认为每个应用 ID 提供了固定的流量配额，具备了基于指标，应用 ID 的限流能力，目前是基于固定时间窗口进行数据量流控。

（3）告警组件：

提供了 Trigger 流式告警和基于 Bosun 的统一 pull 告警。通过推拉结合的告警引擎解决了大规模阈值告警和复杂同环比告警场景。

（4）DownSample 组件：

数据降采样，支持可以配置的聚合采样粒度，节省存储成本。同时提供数据保存更长的时间。

（5）Piper 组件：

统一的告警通知服务，支持告警通知升级。

（6）Transfer 组件：

负责监控数据分发给第三方系统，供数据分析，容量规则，AI 智能告警等用途。

（7）Grafana 看板服务：

所见即所得的查询，提供丰富的图表展现以及监控大盘。

（8）TSDB Cluster：

是最核心的时序存储集群，时序类的查询一般 QPS 都比较高（主要有很多告警规则），通常都是 range 查询，每次查询某一个单一的指标/时间线，或者一组时间线进行聚合。所以对于这类数据都会有专门的时序引擎来支撑，目前主流的时序引擎基本上都是用类似于 LSM Tree 的思想来实现，以适应高吞吐的写入和查询。

（9）ClickHouse Cluster：

ClickHouse 作为优秀的 OLAP 列式数据库，早期是我们采用的第三代时序存储引擎，现在慢慢退居二线，目前现在用来导入一些时序数据和高基数指标数据，提供一些额外的数据分析能力。

（10）Hickwall Portal：

一站式的监控日志告警治理平台，目前提供了指标接入，指标查询，告警配置，通知配置，日志接入，日志管理，机器 agent 治理等模块。

从存储集群来看，TSDB Cluster 的架构如下：

- 总体架构分为三层结构，vminsert 写入层，vmstorage 存储层，vmselect 查询层。这三个组件都可以单独进行扩展，并运行在大多数合适软件上。

- 写入层无状态，支持多协议的写入，写入层支持多协议，包括 InfluxDB，OpenTSDB，Prometheus，Graphite 等。接受程序写入的数据，通过对 metric+tag 组合进行一致性 hash 写入到对应的存储节点，当有存储节点失联，会进行数据重路由分发到好的节点上面。重路由的过程中，由于数据分发策略的变化，可能会导致写入变慢，等待存储节点倒排索引重建完成，就会恢复写入速度正常。

- 存储层有状态，采用 shared nothing 的结构，每个节点数据不共享，独立存储，增加了集群的可用性，简化集群的运维和集群的扩展。支持多租户，采用了 ZSTD 压缩，列式存储，支持副本配置。

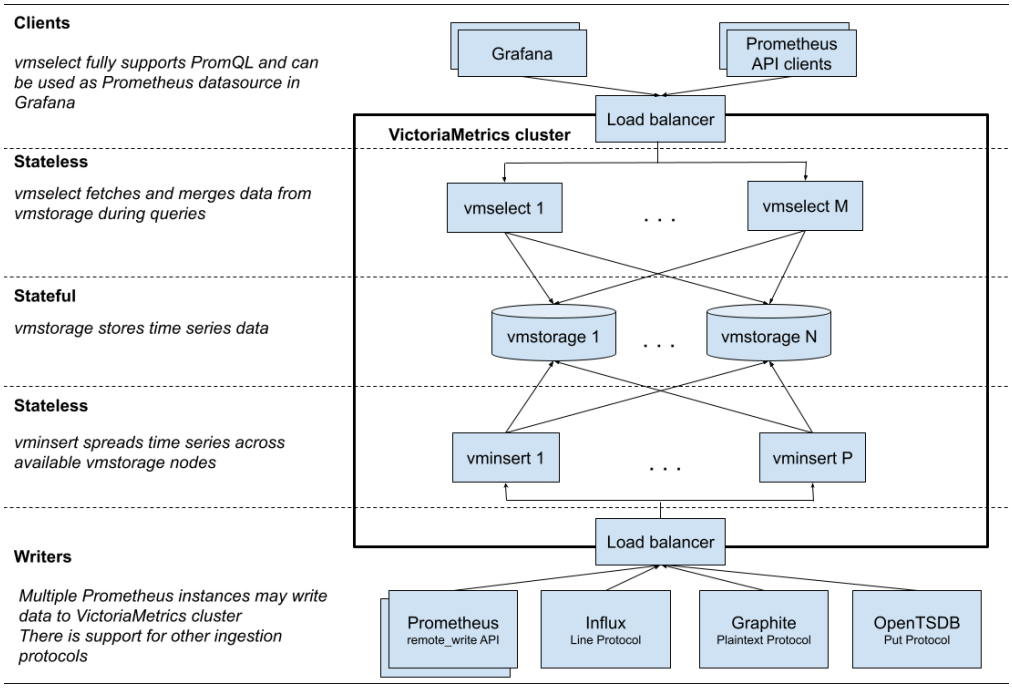
存储层的基本原理可以理解为存储了原始的数据，并且会依据查询层发来的 time range 和 label filter 进行数据查找并且返回。在存储层，针对时序数据做了很多存储优化。存储层要求配置一个数据保存的时间，俗称 Retention Period。Retention Period 到期后，会进行倒排索引的清理和重建，cpu 和 io 通常会大幅提升，会影响写入效率。

从压缩来看，压缩能够很好节省内存和磁盘空间，时序数据的压缩特征比较明显，TSDB Cluster 采用先做时序压缩，再做通用压缩的方法。比如，先做 delta-of-delta 计算或者异或计算，然后根据情况做 zig-zag，最后再根据情况做一次 ZSTD 压缩。据测试统计，在生产环境中，每个数据点（8 字节时间戳 + 8 字节 value 共计 16 字节）压缩后小于 1 个字节，最高可达 0.4 字节，能提供比 Gorilla 算法更好的压缩率。

- 查询层无状态，支持 PromQL 查询。

基本原理可以理解为进行查询语法解析，从存储层获取时序数据并且返回标准的格式，查询层往往会进行一些查询 QPS，查询耗时的限制，以保证后端服务不被拖垮。

TSDB 的部署架构图如下：



六、未来规划

随着可观测性技术的不断发展，仅仅局限于 Metrics 监控是不行的，我们对未来的展望如下。

（1）指标分级

指标管理没有优先级，希望提供分级管理的模式。

（2）云原生可观测性的探索

eBPF 指标采集的引入，提升主机端的可观测性能力。

（3）Logging，Metrics，Tracing 的结合。

多套方案交织：可能要使用至少 Metrics、Logging、Tracing3 种方案，维护代价巨大。在这种多套方案组合的场景下，问题排查需要和多套系统打交道，若这些系统归属不同的团队，还需要和多个团队进行交互才能解决问题，整体的维护和使用代价非常巨大。因此我们希望能够使用一套系统去解决所有类型可观测性数据的采集、存储、分析的功能。

（4）兼容业界主流协议，OpenTelemetry 的标准。

OpenTelemetry 旨在提供统一的可观测性数据收集，未来服务端可以提供兼容 OpenTelemetry 协议的接入，拥抱开源社区，我们在保持关注中。

（5）agent 边缘计算，前置数据聚合。

现在是服务端基于 Flink 做预聚合，希望可以在 agent 端提供一些预聚合能力，比如采集日志的 agent 能够聚合 Metrics 输出。