携程技术

2021 年度合辑

无线大前端/大数据和人工智能

框架架构/质量保障/云计算/数据库/运维

携程 携程技术出品

“携程技术”微信公众号

(ID：ctriptech)

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

2021 年仍然是艰难的一年。反复的疫情和全球经济的不确定性，让几乎所有对旅游业不利的因素都在释放。但于变局中开新局，在危机中育新机，旅游业人带着穿越寒冬的信念，奋力前行。携程技术人则“深耕国内，心怀全球”，切身体会到反复疫情为用户出行带来的困难，我们重点提升服务效率，推动进一步化繁为简并提高灵活性，加强系统的自动化程度，加快对用户需求的响应。

- 移动端的跨端技术深入进一步赋能业务，React Native、Flutter、小程序等跨平台框架将动态化的能力最大化的支撑业务，带来业务迭代周期缩短、开发效率的提升；同时，也解决了我们面临着的多端复用、多 App 复用、单页面多业务团队、平台支撑等业务问题
- 后深度学习时代下，技术迭代全面进入深水区，数据治理更加迫切，把知识驱动与数据驱动结合起来，个性化推荐优化、疫情变化下的快速客服调度系统，让我们回到初心，通过科技链接用户、商户和内容
- 回顾行业的发展历程，基础设施从物理机到虚拟机，再从虚拟机到容器；服务架构从传统单体应用架构到 SOA 架构，再从 SOA 架构到微服务架构，演变的本质原则无非是解决资源成本或者研发效率的问题。Serverless 和 Service Mesh 的落地演进，从完成技术选型到优化核心链路，在节省资源成本的同时，保障“深耕”和“出海”两大业务稳定性

所谓“天下武功，唯快不破”，这一年我们聚焦研发效率的提升，着眼用户的极致体验。一个看起来很小的业务需求——例如卖票，为了达到极致，需要深入理解用户的痛点，用技术保障每个体验点都是顺滑而自然的。真正的“高手”不是拿着所掌握的技术去卡需求，而是倾听用户，给出更贴合的方案。

“知彼知己，百战不殆；不知彼知己，一胜一负；不知彼不知己，每战必殆。” 2022 年，携程技术人依然秉承着“合作共赢，共建开发生态”，从引入领先的技术设计能力，构建全方位、多维度技术架构体系，到大数据人工智能的贴合业务生态，落实收益，回馈社区，我们一直在路上。

年度总结既是过去一年的回顾，也是对新一年的期许，照例献上我们的技术年货。作为集合来自“携程技术”微信公众号全年度的重要技术总结，合辑中的 30 多篇文章，覆盖了前端、后端、大数据、自动化测试、运维等 7 个领域，本着“更好的技术，更好的服务”的使命，诚意满满！欢迎大家针对书中的技术问题深入探讨。

祝大家 2022 年一切顺利，如虎添翼！

携程副总裁/技术委员会主席 马超
2022 年 1 月 上海

无线大前端

携程火车票 Flutter 最佳实践

【作者简介】本文为联合撰稿，作者为携程火车票 Flutter 团队，致力于跨端快速、高性能开发。

背景

在竞争激烈的移动时代，各大互联网公司都在争相抢夺市场，如何提高研发效率，快速迭代产品成为非常重要的因素。

跨平台方案能够节约一定开发、测试、运维成本。Flutter 是由谷歌开源的跨平台框架，可以快速在 iOS 和 Android 上构建高质量的原生用户界面。

一、 为什么选择 Flutter

携程在已经引入了 React Native 的情况下，为什么还会选择 Flutter？更多是对性能的考虑。开发效率与性能体验就像天平两端，需要找到一个平衡点。RN 能够满足我们绝大部分的业务，并且热更、版本控制都很灵活。但是在复杂页面上，特别是在长列表的渲染上，还是存在一定的问题，促使我们去尝试一些新的解决方案。Flutter 官宣自绘 UI 引擎，采用原生方式做渲染，媲美原生体验。

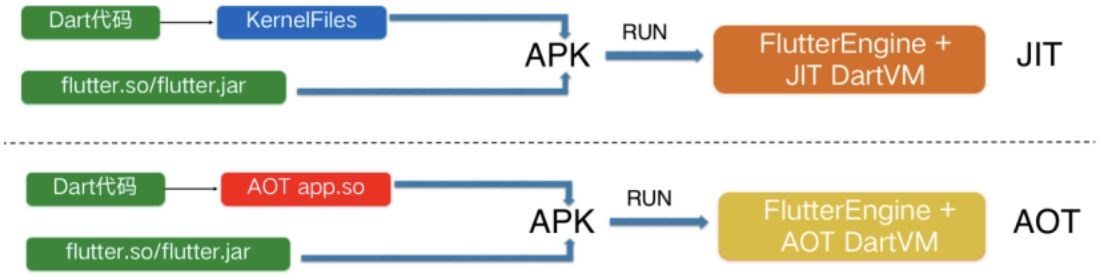
Native 、React Native、Flutter 对比如下：

对比维度			
跨平台	不支持	支持	支持
开发效率	一般	高	高
开发语言	Java/Kotlin/OC/Swift	JS	Dart
支持HOTFIX难度	一般	简单	难
用户体验	丝滑柔顺	略有卡顿	无卡顿
包体积	报体积一般	包体积很小	包体积大
内存问题	没有额外开销	有较小额外开销	每次创建FlutterEngine开销较大

1.1 研发效率

Flutter 具有跨平台性，可以在多端上运行。同时 Dart 语言作为开发语言，本身的优势就在

于它既支持 JIT，又支持 AOT，在 JIT（Just In Time）即时编译功能下，能提供 Hot Reload 功能。在开发过程中，实时地看到界面改动。生产包 AOT 编译，将代码编译成 ARM 二进制，从而既可以享受运行时又具有原生语言相近的运行效率。



1.2 扩展性好

Flutter 提供了多种不同的 Channel，用于 Dart 和平台之间相互通信。通过这些桥方法，使 Flutter 具有很好地与 Native 和 React Native 进行混合编程的能力。赋予 Flutter 一些 Native 的能力，同时也能很好地让我们在现有 Native 项目混合 Flutter 开发。

二、 Provider 对 MVVM 架构的实践

在 Flutter 的开发过程中，特别是一些业务复杂的页面，为了代码结构清晰，模块逻辑解耦，我们一般采用的是模块化的编程思想。随之而来的问题就是，组件之间怎么相互通讯，比如变更了登录态，如何通知其他模块刷新？

推荐使用 Provider 来管理各个组件的状态，我们实践下来 ，主体布局采用 MVVM 模式是比较方便做模块化编程的。

2.1 为什么需要使用 Provider

如果状态是该组件私有的，则应该由组件自己管理；但是如果状态要跨组件共享，则该状态应该由各个组件共同的父元素来管理。对于组件私有的状态很好理解，当需要刷新当前 widget 的时候，只需要通过 setState()的方法来实现组件重绘的效果；对于跨组件共享的状态，可以使用 EventBus 来实现。

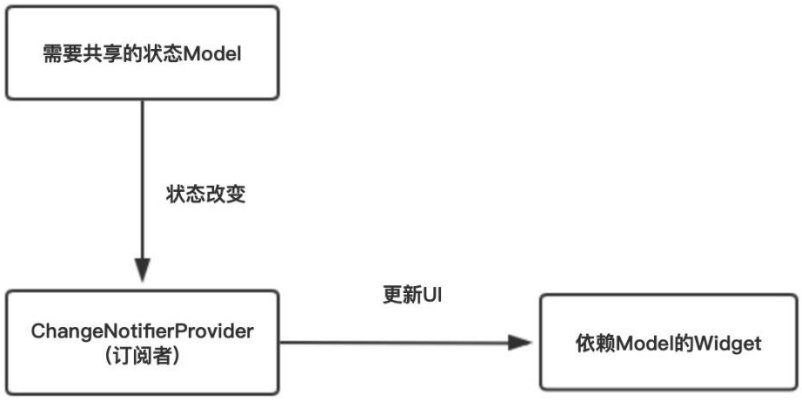
可是当事件多了的时候，难以正确管理，其次订阅者必须要显式注册状态改变回调，也必须在组件销毁的时候手动解绑以避免内存泄漏。而 Provider 就可以通过自身的原理，简单地去实现状态共享，不需要麻烦的操作。且 Provider 是官方推荐的状态管理方式，具有良好的生态环境及维护团队。

2.2 Provider 的实现原理

1) InheritedWidget 简单介绍

Provider 是基于 InheritedWidget 的再次封装，InheritedWidget 提供了一种数据在 Widget 树中自上而下传递，共享的方式。我们在根 Widget 继承了 InheritedWidget，然后在该组件中存放一个数据 data，那么可以在任意子 Widget 中来获取该组件的数据并使用。当在任一组件中改变了共享数据 data，InheritedWidget 组件会自上而下通知所有使用过共享数据的组件并刷新组件，同时会回调 didChangeDependencies() 方法。

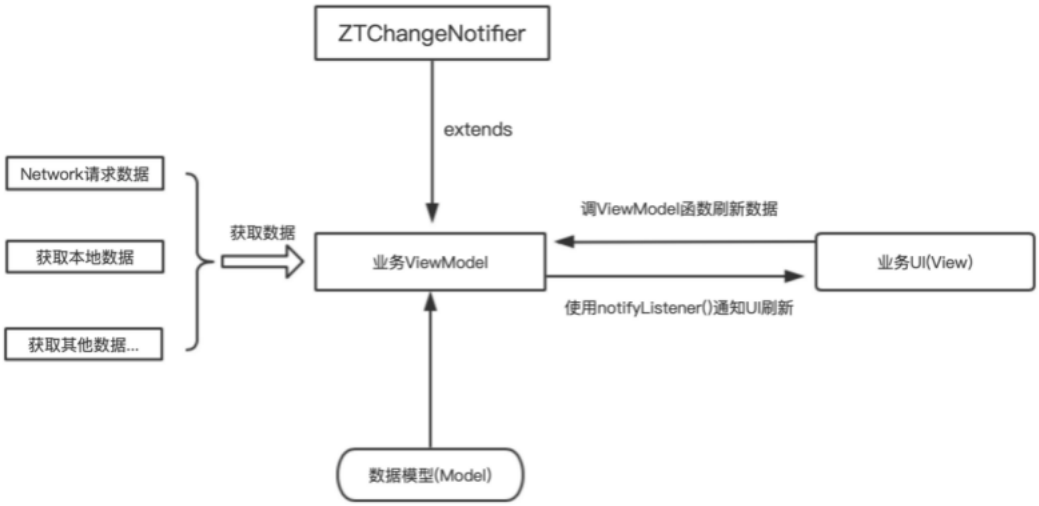
2) Provider 的原理和流程



共享数据的 Model 变化后，会自动通知 ChangeNotifierProvider，ChangeNotifierProvider 内部会重新构建 InheritedWidget，而依赖该 InheritedWidget 的子 Widget 就会更新。

2.3 Provider 的使用方式

架构模式图如下:



1）创建业务 ViewModel，在 ViewModel 内部存放需要共享的数据。ViewModel 继承 Flutter SDK 中提供的 ChangeNotifier 类，它继承 Listenable，也实现了一个 Flutter 风格的订阅者模

式，其内部实现了 addListener()，removeListener()等方法，实现对订阅者的处理。同时最好复写 dispose()和 notifyListeners()方法，防止用户在调用数据时销毁界面，而等到数据获取到以后通知界面刷新导致 Crash。

2）注册状态管理类，使用 ChangeNotifierProvider 或者 MutiProvider 将需要共享数据的 Widget 包起来，单个 NotifierProvider 时使用 ChangeNotifierProvider，多个 NotifierProvider 时使用 MutiProvider 包装，如下：

```
///多个 NotifierProvider 的时候
return MultiProvider(providers: [
      ChangeNotifierProvider(create: (context) =>
dataViewModel(mCommonAdvancedFilterRoot,query)),
      ChangeNotifierProvider(create: (context) => UserPreferentialViewModel(query)),
      ChangeNotifierProvider(create: (context) => UserPromotionViewModel())
///需要调用共享数据的子 Widget
], child: ListResearchPageful(query));
```

3)在被包起来的 Widget 中的任一子组件中获取共享数据 ViewModel, 可以在 StatefulWidget 中的 builder()方法中获取，也可以使用 Builder 组件进行获取，如下：

```
///在 StatefulWidget 中的 build()方法中获取 ViewModel
class ListResearchPageState extends TripState<ListResearchPageful> {
@override
  Widget build(BuildContext context) {
///使用 Provider 包装以后，可以在 widget 的任一一个子 widget 获取共享数据并操作数
据，在这里就是可以在 HotelListView 方法下的唯一位置获取 ViewModel
      var listViewModel = Provider.of<ListDataViewModel>(context);
      var userPromotionViewModel = Provider.of<UserPromotionViewModel>(context);
      return MediaQuery(
            child: QueryListPage(widget.query,
      ListDataViewModel, userPromotionViewModel));
  }
}

///借用 Builder 组件进行获取 ViewModel
@override
Widget build(BuildContext context) {
///使用 Provider 包装以后，可以在 widget 的任一一个子 widget 获取共享数据并操作数
据，在这里就是可以在 ListView 方法下的唯一位置获取 ListDataViewModel
  var userPromotionViewModel = Provider.of<UserPromotionViewModel>(context);
  return MediaQuery(
        child: Builder(builder: (context) {
var listDataViewModel = Provider.of<ListDataViewModel>(context);
            return queryListPage(
```

```
widget.query, listDataViewModel, userPromotionViewModel);
        },));
}
```

4）获取到 ViewModel 后，可以在子组件中直接使用 viewmodel 中的共享数据，如下：

```
//领券监听
///此处可以直接使用 viewModel 调用 viewmodel 中的方法
Event.addEventListener(
"UPDATE_QUERY_RESULT_LIST",(eventName, eventData) {
  if (isOnPause) {
    listViewModel.isNeedRefresh = true;
    listViewModel.refreshListData(listViewModel.query);
  } else {
    listViewModel.refreshListData(listViewModel.query);
  }
});
```

2.4 Provider 的优势

1）我们的业务代码更专注数据，只要更新 Model，UI 就会自动更新，不用在状态改变后再去手动调用 setState()来显示更新页面。

2）数据改变的消息传递被屏蔽时，我们无需手动去处理状态改变事件的发布和订阅，provider 自行处理。

3）在大型复杂应用中，尤其是需要全局共享的状态非常多时，使用 Provider 将会大大简化代码逻辑，降低出错的概率，提高开发效率。

三、Flutter 性能调优

一个新技术改造完成，我们最关注的当然是性能体验有没有达到预期。那 Flutter 页面性能评判标准是什么，如何去度量，有没有可视化工具，帮我们去做一些性能调优。

3.1 Flutter 渲染原理简介

在做性能优化之前，先让我们了解一下渲染的原理。Flutter 的一切皆为 Widget。为了性能又区分了 StatefulWidget，StatelessWidget。StatefulWidget 能通过 setState()来实现刷新。这样的设计方便我们去控制局部刷新，从而提高性能。

Flutter 中的控件会历 Widget -> Element -> RenderObject -> Layer 这样的变化过程，而其中 Layer 的组成由 RenderObject 中的 isRepaintBoundary 标志位决定。

当调用 setState() 时，RenderObject 就会往上的父节点去查找，根据 isRepaintBoundary 是

否为 true，会决定是否从这里开始往下去触发重绘，来确定要更新哪些区域。

3.2 构建运行 Profile 模式

Flutter 支持三种模式编译 app，Debug 模式、Release 模式和 Profile 模式。Debug 模式 采用 JIT 编译，支持 HotReload，所以在 Debug 模式下会放大性能问题。性能分析需要确保使用真机并在 profile 模式下运行，这样拿到的数据是最接近真实性能的。

1）Debug 模式对应 Dart 的 JIT 模式，可以在真机和模拟器上运行。该模式会打开所有的断言，以及所有的调试信息、服务扩展和调试辅助。此外，该模式支持有状态的 Hot reload。

2）Release 模式对应 Dart 的 AOT 模式，只能在真机上运行，不能在模拟器上运行，其编译目标为最终的线上发布。该模式会关闭所有的断言，以及尽可能多的调试信息、服务扩展和调试辅助。此外，该模式优化了应用快速启动、代码快速执行，以及二级制包大小。

3）Profile 模式，基本与 Release 模式一致，只是多了对 Profile 模式的服务扩展的支持，包括支持跟踪，以及一些为了最低限度支持所需要的依赖。该模式用于分析真实设备实际运行性能。

- 纯 Flutter 项目构建 Profile 模式

flutter run —profile 命令是使用 Profile 模式来编译的。IDE 也是支持这个模式的，例如 Android Studio 提供了 Run > Profile… 菜单选项。

- Flutter 与 Native 混合项目构建 Profile 模式

a. 打包 Flutter 工程 Profile 产物

```
// 进入 flutter 项目，执行 build-release，并指定输出目录 tripflutter
build-release -o /projects/ctrip_flutter/release -i info
```

b. 配置 Native 项目

打包好 flutter 产物之后，需要导入到 native 项目并打包。修改 Native 项目根目录的 gradle.properties 文件。

```
### 开启 Profile 模式
TRIP_FLUTTER_PROFILE=true
### 设置 profile 模式下 js 使用的产物目录（过程 1 构建的 ./profile 目录)
TRIP_FLUTTER_LOCAL_OUTPUTS_PATH=/projects/ctrip_flutter/profile
```

c. 构建 Native 工程

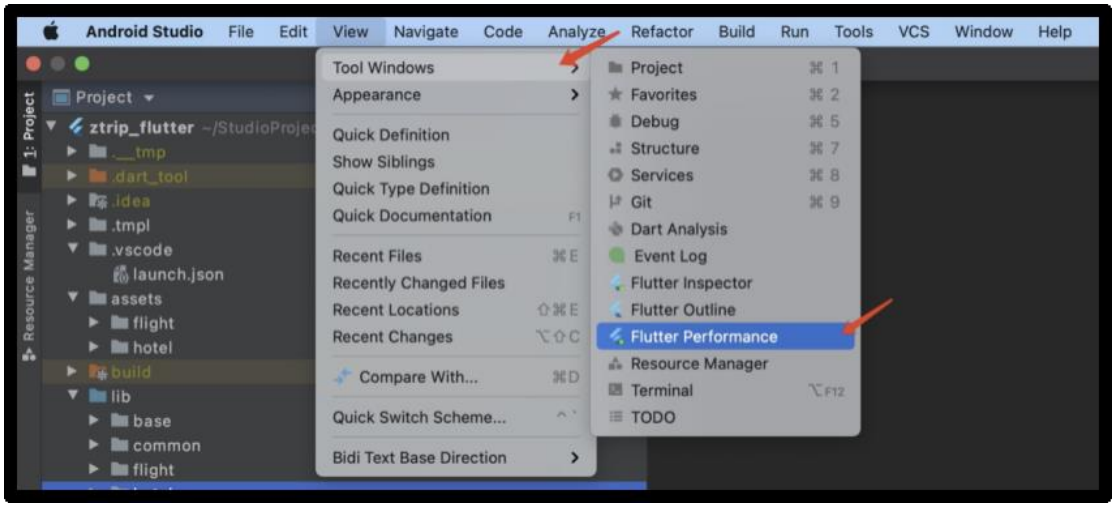
直接通过 IDE 运行到手机上。

3.3 性能分析工具及方法

1）performance overlay

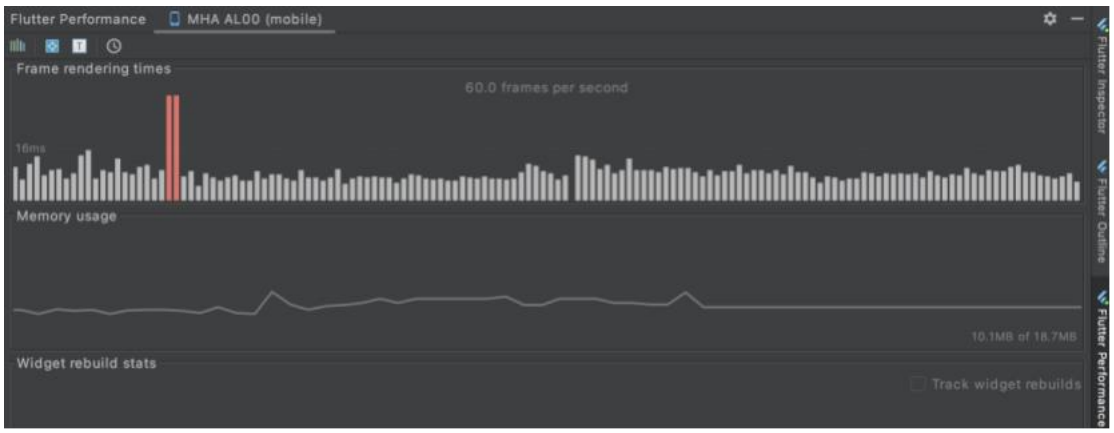
平时常用的性能分析工具有 performance overlay，通过它可以直观看到当前帧的耗时。在 Profile 模式下，通过 Android Studio 看页面的 FPS，注意需要在 HotReload 连接的情况下查看。

选中 View > Tool Windows > Flutter Performance。



点击上面图中的箭头所指的按钮，就会在手机或模拟器中打开（如下图所示）。FPS 是一个动态过程，页面滑动这个值是一直变化的，最右边的是当前帧。出现红色则表示耗时超过 16.6ms，也就是发生丢帧现象，也是我们常说的页面闪动问题。performance overlay 的主要功能如下：

- 获取 FPS 数值来衡量页面性能，方便对比 Flutter、Native 页面帧率；
- 直观统计页面在各个机型上面的表现；
- 定位页面的具体哪个模块有问题；



2）Dart DevTool

另一个工具是 Dart DevTool ，在 Android studio 右侧，还可以从 Flutter inspector 里面的 more action，以及 Flutter Performance 底部的入口进入。

目前 DevTools 支持的功能有如下一些：

- 检查和分析应用程序的 UI 布局和状态。
- 诊断应用的 UI 性能问题。
- 检测和分析应用程序的 CPU 使用情况。
- 分析应用程序的网络使用情况。
- Flutter 或 Dart 应用程序的源代码级调试。
- 调试 Flutter 或 Dart 应用程序的内存使用情况和分析内存问题。
- 查看运行的 Flutter 或 Dart 应用程序的一般日志和诊断信息。

3.4 实战性能技巧

1）懒加载 ListView

推荐使用 ListView.builder () 构建 List, 这样当 Item 滚入屏幕时才创建 Item, 而不是 ListView-children，这样会立刻创建所有的 Item。

```
///Bad code 不推荐使用 children 构建 List
ListView(children: getItems(mList))
List<Widget> getItems(List<FilterNode> mList){
   List<Widget> items=new List<Widget>();
   if(null!=mList){
for(Node node in mList){
items.add(Text("不推荐写法"));
}
   }
   return items;
}

///推荐写法
ListView.builder(
// physics: NeverScrollableScrollPhysics(),
//shrinkWrap: true,
itemCount:mList.length,
itemBuilder: (BuildContext context, int index) {
return Text("推荐使用 ListView.builder （） ");
})
)
```

注意，无论是 ListView 还是 GridView，只要是设置了 shrinkWrap: true 属性，都没有了懒加载的效果了。

2）控制刷新范围与次数

- 尽量避免在滑动监听中触发 setStat()刷新视图。

如上图所示，需要滑动的过程中，显示、隐藏标题栏，并且是一个渐变的过程，遇到这种情况，一定要尽量的控制刷新的范围和频次。控制在只在头图可见的情况下面触发 setStat()，避免不必要的页面滑动触发刷新。

```
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;
              });
  }
});
```

- 尽量将 setStat()放在放置于视图树的低层级，好处是 build 时影响范围极小，简称局部

刷新。

如上图所示在列表中 Item 中存在大量的倒计时。一定要控制刷新倒计时只影响控件本身，并且只有可视的区域视图是在刷新的，不可见的情况下及时销毁计时器。一直刷整个列表，性能开销是恐怖的。

```
Widget build(BuildContext context) {
return Text(timeRemaining,
        style: TextStyle(
            color: HotelColors.hotel_list_reduction_sale_color,
            fontSize: 10,
            fontWeight: FontUtil.mediumWeight));
}
```

3）避免组件重复创建

能复用的组件尽量复用，特别是在组件化编程，页面级的情况下面，每次刷新页面把所有的子组件都重新渲染一遍，性能开销也是很大的。尽量复用，避免不必要的视图创建。

```
///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;
  }
}
```

四、Flutter 布局技巧

4.1 Flutter 不可见组件预加载

Flutter 一些组件基本都是有懒加载的，不可见的组件是没有渲染视图的，这样滑动过去，有

用到网络图片的地方，经常会先白一下。针对这种情况我们对将要加载的图片进行预加载处理，比如列表页在分页请求数据回来的时候做图片预加载。还有，下一个页面的图片，需要一进去就有图片直接显示，就可以在当前页面做图片预加载。

预加载

未预加载

代码如下所示：

```
///对每一页加载的数据进行做图片预加载
(hotelListViewModel.currentPageHotels ?? []).forEach((element) {
var logo = element?.logo ?? "";
  if (StringUtil.isNotEmpty(logo)) {
    precacheImage(NetworkImage(logo), context);
  }
});
```

当数据出来后使用 PreChcheImage()预加载处理图片链接，以保证当用户滑动图片以后不会看到图片加载白屏这种问题。

4.2 Flutter 数据预加载

为了缩短用户的加载等待时长，我们经常需要一些预加载方法。比如在前一个页面预加载下一个页面的数据，或者在长列表的分页请求时候，可以做分页预加载。比如当你滑动到第五个可见的时候，就提前把下一页的数据加载好。

列表页通过桥方法获取上一个页面预加载的数据，这样就能有一个直出体验，这里要考虑数据已经加载好、加载中、加载失败的情况。同时还要考虑，缓存数据的时效性，什么情况下需要删除缓存。

```
///请求列表数据数据
void loadListData(HotelQuery query) {
///在首页提前获取列表页的数据并缓存到本地，当用户进入列表页时可以直接展示数据
  if (resultModel != null) {
    ///判断是否需要再次请求数据
    _dealWithResult(resultModel);
    return;
  } else if (isPreloading) {
    /// 通过桥方法获取首页已经缓存的数据
HotelBridge.getListCache<Map>({'queryModel':query.toJson()})
        .then((resp) {
    final newResultModel =
        QueryResultModel.fromJson(resp);
      ///有缓存数据直接处理使用
      _dealWithResult(newResultModel);
    }).catchError((error) {
      ///没有数据采取请求列表页的数据
      getHotelList();
    });
  }
}
```

4.3 布局自适应高度

如果需要根据内容填充的高度来自适应左边图片的高度，目前 Flutter 并不支持该功能，我们可以借助 IntrinsicHeight 组件来完美地解决该问题。InstrinsicHeight 可以让同一行的子 widget 都是相同的高度。

- 可以将需要自适应高度的 Widget 使用 ConstrainedBox 进行包裹，并设置最低高度；
- 将图片作为 Container 的背景图片，使用 DecorationImage 进行修饰当前的 Container；
- 将图片的填充方式设置为 BoxFit.Cover 或者 fillHeight 即可；

五、Flutter 中常见问题分析及解决方案

5.1 设置 State 引起的问题

1）错误展示信息：

NoSuchMethodError: The method markNeedsBuild was called on null。

2）错误分析

这个错误一般情况下出现在异步任务，比如一些界面请求网络数据，异步获取本地数据等，需要根据数据的状态来改变刷新 Widget State。异步任务结束在页面被销毁之后，没有检查 State 是否还是 mounted 状态，继续 setState()就会出现这个错误。错误代码如下所示：

```
///从服务器端获取当前活动终止时间，当服务器返回以后，会通知刷新这里
///如果用户在数据返回之前销毁该界面，等数据回来以后刷新界面就会报错
final endTime = roomDetailItemEntity?.tonightEndTime ?? '';
int endTimeOfNum = 0;
if (endTime.isNotEmpty) {
    try {
        endTimeOfNum = int.parse(endTime) ?? 0;
        if(endTimeOfNum - Util.currentTimeMillis() > 0) {
            this.setState(() {
```

```
                    _showCountDown = true;
                });
            }
    } catch (e) {}
}
```

3）处理办法

在调用 setState()方法之前检查是否 mounted，mounted 是一个标示当前 Widget 树是否已经被渲染的状态值。所以 mounted 检查很重要，只要涉及到异步还有各种回调的时候，都不能忘记检查该值。如下：

```
final endTime = roomDetailItemEntity?.tonightEndTime ?? '';
int endTimeOfNum = 0;
if (endTime.isNotEmpty) {
    try {
        endTimeOfNum = int.parse(endTime) ?? 0;
        if(endTimeOfNum - Util.currentTimeMillis() > 0) {
            if(mounted) {
                this.setState(() {
                _showCountDown = true;
                });}}} catch (e) {}
}
```

5.2 使用 MediaQuery.of()动态获取屏幕属性的问题

1）错误展示信息

```
BoxConstraints has a negative minimum width;
```

2）错误分析

这种情况一般出现在需要获取屏幕宽度，根据屏幕宽度减去另外一个组件的宽度，用来设置另外一个组件的宽度导致，在一些计算速度比较低的手机，可能获取到的屏幕宽度为 0，这样就会导致你的组件的宽度为负数，报出错误异常。如下所示：

```
Widget hotelListDesContent(BuildContext context) {
return Container(
///此处想实现左边是图片，右边是相关信息的布局，如果 MediaQuery.on(context).size.width 获取为 0 时，就会报出异常
    width: MediaQuery.of(context).size.width - Dimens.image_width80,
  ///右边内容
  child: Stack(children: [
    Container(child: Column(
```

```
                    mainAxisSize: MainAxisSize.min,
                    mainAxisAlignment: MainAxisAlignment.start,
                    children: <Widget>[
                      hotelListDesName(),
                      englishName(),
                      hotelListRemarkContent(),],),),
              ///左边图片
              Positioned(child: fullRoomItem()),
            ],
));
}
```

3）处理方式

尽量使用 Expand，Flexible，Flex，Wrap，Stack 等组件配合 Column，Row 进行动态布局设置组件的宽高等。如下所示：

```
Widget hotelListDesContent(BuildContext context) {
return Expanded(
        flex: 1,child: Stack(
        children: [Container(
              child: Column(
                  mainAxisSize: MainAxisSize.min,
                  mainAxisAlignment: MainAxisAlignment.start,
                  children: <Widget>[
                    hotelListDesName(),
                    englishName(),
                    hotelListRemarkContent(),],),),
              Positioned(child: fullRoomItem()),
            ],
        ));
}
```

5.3 使用 Provider 时，未判断界面状态通知界面刷新的问题

1）错误信息展示

Null check operator used on a null value;

2）错误分析

一般情况下出现这种问题是由于界面销毁后，继续调用 notifyListeners()方法通知界面刷新引起的 bug。当用户打开一个界面，我们发送了 API 请求，此时用户销毁了界面，我们并未监听，等到数据返回以后，强行通知界面刷新，导致 Crash。如下所示：

```
HotelServices.getTyHotelRoomPrice(params, ApiCallBack(onSuccess: (Object obj) {
this.roomPriceEntity = HotelRoomPriceEntity.fromJson(obj);
   this.resultCode = 1;
   ///如果在数据返回是，用户已经关闭当前界面，此处通知刷新界面会导致 crash
   notifyListeners();
}, onError: (int code, String message) {}
   notifyListeners()
}));
```

3）处理方式

正常情况下，我们会写一个基类继承 ChangeNotifier，在内部重新复写 dispose（）方法，同时重新封装方法通知刷新界面，在每次需要通知刷新界面的时候判断当前界面是否已经被销毁。如下所示：

```
import 'package:flutter/cupertino.dart';
/// ViewModel 基类
class HotelViewModel extends ChangeNotifier{
   bool _disposed = false;
   @override
   void dispose() {
      _disposed = true;
      super.dispose();
   }
   void hotelNotifyListeners() {
      if(!_disposed){
         notifyListeners();
      }
   }
}
```

5.4 使用 Text.rich 时导致的问题

1）错误信息展示：UnimplementedError

2）错误分析

出现这个问题的原因在于使用 Text.rich 来展示多个 Span 组件时，如果设置了最大行数，当组件超过最大行数，有别的组件未成功展示时，再次点击当前 widget，使它接受时间，就会导致 crash，用户的感知为操作无响应，其实已经 crash。如下所示：

```
///母房型名称，当前我们 Text 最大显示两行，当大于两行是，出现...,可是此时第二个组件无处显示，当用户点击就会 crash
Row(children: <Widget>[
```

```
Expanded(child: Text.rich(TextSpan(
      children: [TextSpan(
      text: itemRoomEntity.baseName ??""),
        WidgetSpan(
          child: Container(
            padding: EdgeInsets.only(bottom: Dimens.gap_dp3),
            child: Icon(HotelIcons.show_more),
          ),
        ),
      ]), maxLines: 2, overflow: TextOverflow.ellipsis),
  ),
], crossAxisAlignment: CrossAxisAlignment.center,),
```

3）解决办法

使用 Flexible 代替 Expanded，直接使用 Text 即可，区别在于 Flexible 不会自动填充整个剩余宽度，如下所示：

```
///母房型名称
Row( mainAxisAlignment: MainAxisAlignment.start,
        children: <Widget>[
Flexible(child: Text((childCount > 1)?itemRoomEntity.baseName ?? "":"",
          maxLines: 2,
          overflow: TextOverflow.ellipsis,),),
Container(child: Icon(childCount ==1?HotelIcons.show_more:null),
        margin: EdgeInsets.only(top: Dimens.gap_dp2),),
  ], crossAxisAlignment: CrossAxisAlignment.center,)
```

六、总结与展望

总结一下，本文我们介绍了选择 Flutter 的初衷，Provider 状态管理的实际使用，建议 Flutter 主体的构架采用 MVVM 模式，还介绍了一些 Flutter 性能检测、量化工具和一些性能优化点供大家参考。收集了 Flutter 开发过程中常见并且大量发生的问题，并提供了相应的解决方案。

在复杂业务和长列表上面体验，确实 Flutter 优于 React Native。但是 React Native 也有它的优势，比如灵活的版本迭代。没有最好的跨平台方案，只有最合适业务的。目前来说，Flutter 还处于早期阶段，随着 Flutter2.0 的重大升级，其跨平台能力、性能、生态系统将会蓬勃发展，还是很值得尝试的。后续我们也将有更多的业务接入 Flutter。

【参考文档】

[1] Flutter 开发文档
https://flutter.cn/docs/perf/metrics

[2] Tripflutter 开发文档
http://pages.release.ctripcorp.com/trip-flutter/docs/
[3] 咸鱼技术
https://developer.aliyun.com/group/idlefish?spm=a2c6h.12873639.0.0.2c9618dd4mdBAQ#/?_k=khoksz
[4] Flutter 实战
https://flutter.cn/docs/perf/metrics
[5] 美团技术
https://tech.meituan.com/

Trip.com APP 启动优化实践

【作者简介】Shanks，携程移动开发专家，关注移动端基础技术。

引言

启动是用户对 App 的第一印象，对于用户体验尤为重要，所以我们花了很多时间在启动时间的优化上。本文将分享 Trip.com App 的启动优化实践，从分析 App 启动的过程开始，在了解启动流程的基础上制定大的优化原则和小的具体方案，希望能对大家有所帮助。

一、App 启动的流程分析

想做启动优化，首先要了解清楚启动的各个流程，然后才能对各个环节去做针对性措施。

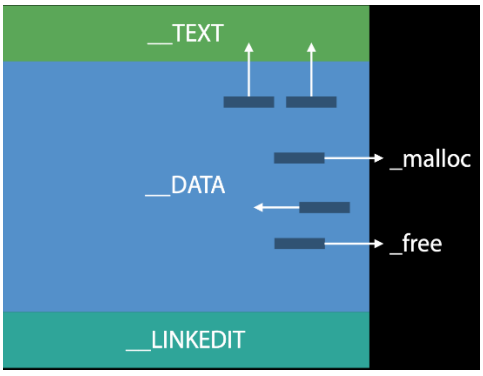
借用 WWDC 对启动阶段的定义图：

1.1 System Interface

- 加载 App 可执行文件
- Load dylibs

加载动态链接器 dyld，dyld 会递归加载 App 依赖的动态库，然后执行符号绑定 Rebase，Bind。一般应用会加载 100 到 400 个 dylib 文件，幸运是大部分是系统库，且系统会在操作系统启动时计算和缓存系统动态库。

Apple 为了解决安全问题，引入 ASLR 和 Code Sign，如果不作符号修正，程序将没法正常运行，所以会有 Rebase 和 Bind 过程。



- Rebase

在镜像内部调整指针的指向，其实就是将内部指针都加上偏移量（Slide=实际新地址-旧地址）

- Bind

修正指外部的指针，比如上图中 malloc，这个符号不存在于我们 App 的 Mach-O 中，需要从外部的镜像中获取，这时候就需要 Bind 操作把这个关联起来。

- libSystem init

调用系统的一些初始化方法，这部分一般时间比较固定，可以不用太关注。

1.2 Runtime Init

- Objc 和 Swift 的初始化

通过_dyld_objc_notify_register 注册回调，在 image 加载完时初始化语言相关。

- 加载 category

在上面语言初始化完之后，会加载所有 category，处理 category 的所有方法，协议和属性等。

- 调用所有+load

也是通过向 dyld 注册回调，在 image 加载完时，通过 load_images 触发，处理该 image 相关的所有+load 方法，按照继承层级依次调用：父类+load→子类+load→category +load，注意 category 的+load 不会覆盖原类。

- 调用 C++的构造函数属性函数 attribute((constructor))

1.3 UIKit Init

- 实例化 UIApplication 和 UIApplicationDelegate
- 开始事件处理和系统集成

1.4 Application Init

这部分是我们熟悉的 UIApplicationDelegate 的几个生命周期调用：

- application:willFinishLaunchingWithOptions:
- application:didFinishLaunchingWithOptions:
- applicationDidBecomeActive:

- scene:willConnectToSession:options:
- sceneWillEnterForeground:
- sceneDidBecomeActive:

1.5 Initial Frame Render

这里是 App 渲染第一帧，主要做了创建、布局和绘制视图的工作，并把准备好的第一帧提交给渲染层渲染。这里面布局计算，图片解码，图层树的递归 commit 到 Render Server 等都是可能影响耗时的点，所以要特别注意。

1.6 Extended

这里按照苹果的定义，是异步获取数据展示界面的逻辑。比如我们首页要从网络请求数据然后展示最新数据在页面上。

二、针对启动的各个流程我们能做什么

2.1 总体原则

不管哪个流程，我们都想尽量遵循下面两个原则：

- 删

删的原则是指，对 App 启动和运行不是必须的任务，或者跟首页渲染第一帧无关的任务，都从启动流程中删除。对于删除的任务，可以进行懒加载的形式，需要时再调用；也可以换到其他的时机去触发，比如首页渲染完之后。

- 压

压的原则是指，对 App 启动和运行必须的任务，或者直接影响首页渲染第一帧的任务，都尽可能压缩其运行时间。至于做法，可以是优化方法内的实现，使其运行更快；也可以将方法执行的线程切换到子线程，以并发的形式降低其对整个启动过程的影响。

2.2 具体方案

2.2.1 减少动态库

动态库的加载在启动阶段是必须的，所以我们要尽量减少非必要的动态库。对此我们做了以下几点：

1）梳理所有动态库，将用不到的或者可以简单替代的动态库删除

可以通过 otool -L xxx.app/xxx 或者打开打包后的产物，从 xxx.app/Frameworks 路径中找到所有动态库，逐个筛选，将其中可以废弃和替代的动态库删除。

2）通过推进社区（第三方 SDK）将现有动态库转成静态库

因为依赖了第三方 SDK，我们是不包含源码的，所以这部分需要推进社区提供静态库的版本，或者通过 cocoapods 等工具打包 SDK 的静态库版本。

3）将我们自己的 SDK 编译成静态库

对于我们自己的 SDK，因为有源码，所以直接修改 MACH_O_TYPE 为 Static Library 重新打包即可。

4）App 最低支持系统版本升级到 12.2

因为 iOS 在 12.2 版本及以上才内置了 Swift 的支持，所以在此之前 Swift 的动态库都是随着 App 下发的，也在 xxx.app/Frameworks 里。

当然，这个决策是会直接应用到用户和订单的，所以是要有数据支持的，我们是根据用户占比到达某个阈值才支持 12.2 的。如果允许，甚至可以升级到 iOS 13，因为 iOS13 以上 dlyd3 做了很多加载和缓存的优化。

2.2.2 删除无用代码

如果符号越多，很显然 Rebase 和 Bind 的处理时间就会越长，Objc 的初始化也受影响，所以我们需要尽可能减少代码：

1）通过逆向二进制或者生成 linkmap，解析所有方法（TEXT.text）和引用到的方法（__DATA_objcselrefs），找出无用方法删除

2）解析所有类（DATA.objcclasslist）和引用到的类（DATA.objcclassrefs），找出无用的类删除

3）使用第三方工具或者 clang 扫描重复代码，精简去重

4）使用 LLVM_LTO 和 GCC_OPTIMIZATION_LEVEL 等其他编译选项优化二进制大小

2.2.3 合并 category

合并 category，可以减少 category 加载时的耗时。不过这部分收益不大，并且也会影响编程习惯，所以我们并没有投入很多时间，不再赘述。

2.2.4 删除+load

以前会有很多代码为了省事，加到了+load 中，这部分很显然占用启动时间，所以尽量要把这其中的代码转移，可以放到 initialize 中懒加载，或者放到启动任务中并发执行，尽量减少这部分的影响。

Xcode 调试时，可以通过正则添加所有+load 方法的断点 br s -r "\+\[.+ load\]$" ，然后使用 br list 打印出所有+load 列表，这样方便我们定位所有+load。

2.2.5 UIApplication 子类优化

为了减少 UIKit Init 的时间，可以对 UIApplication 的子类初始化工作优化。我们这部分不存在，所以没有做什么工作。

2.2.6 启动任务并发

想象一下，如果 application:didFinishLaunchingWithOptions:里面执行的所有启动任务不作任何处理，那么代码框架将会很乱，你的优化也只能单点单点去做。

所以我们将 application:didFinishLaunchingWithOptions:阶段所有方法任务化，一个任务做一种类型的事。任务拆分好之后，就可以根据任务之间的相关性，选择哪些任务是可以并发执行，哪些任务是必须有依赖关系前后执行。

以前：

A	->	B	->	C	->	D	->	E

现在：

A	->	D
	C	
B	->	E

当然，任务的拆分颗粒度也很重要，拆分太粗的话，很难达到最优的组合，可能一个任务里的方法之间仍然有并行的空间。拆分太细的话，也有可能导致同一时间并发数太多，造成额外的线程切换开销。

2.2.7 I/O 处理

尤其要注意启动阶段的 I/O，一般出现于读取磁盘中的文件，比如配置文件等。

使用 Instrument→App Launch 去查看启动过程就会发现，如果主线程执行出现很多灰色的块，那就是 I/O，找到这些 I/O 产生的方法，尽量在子线程并发执行，避免阻塞主线程。

2.2.8 首页数据的预加载和懒加载

首页上有很多数据要加载，比如图片、上次缓存在本地的数据等等，这些数据的加载如果在写代码时不作特殊处理，那会在主线程执行，不知不觉就会有很多耗时。

1）预加载

对首页渲染必须的数据，比如一个 icon，或者一个翻译的数据，我们通过在启动任务（之前提到的拆分的并发任务）中新增加一个预加载启动任务，专门负责在 application:didFinishLaunchingWithOptions: 的过程中并发执行数据的获取。因为获取数据大多比较耗时，所以放在子线程充分利用启动阶段的空闲。同时这类任务大多数是 I/O 操作，并不会占用太多 CPU 资源。

更进一步，其实可以对首页用到的资源在运行时作个标记，记录到磁盘，下次启动的时候读取这个记录，对用到的资源进行提前预加载，这样避免 hard code 很多资源名在代码中。

2）懒加载

首页的数据往往很多，但并不是一开始要全部用到。可以对数据作区分，和第一屏展示无关的，使用懒加载，真正用到的时候再去加载。

2.2.9 二进制重排

1）page fault

由于虚拟内存的机制，应用启动时不会把所有数据加载到内存，而是以页为单位逐步从磁盘中加载，内存中的虚拟地址和磁盘中的物理地址有个映射关系。当程序执行时，如果发现要访问的东西不在内存里，就会触发一次 page fault ，去磁盘中加载新的一页。

启动阶段有很多方法要调用，而这些方法在 Mach-O 中的位置又是在编译时确认的。如果有 10 个方法刚好在不同页，可能就要产生 10 次 page fault 。

二进制重排要做的就是将启动阶段要用到的方法，在编译时提前确定，通过.order 文件告诉编译器，这样这些方法会排布在 Mach-O 的最前面，之前的 10 次 page fault 很可能就变成一两次 page fault。

通过在 Other C Flags 中添加 -fsanitize-coverage=func,trace-pc-guard 再通过 __sanitizer_cov_trace_pc_guard 记录启动阶段所有方法的调用，再将这些写入到.order 文件中，在 Xcode 的 ORDER_FILE 设置中配置即可生效。

通过测试，我们的二进制重排大概优化 100-200ms。

2.2.10 其他通用手段

针对启动任务和首页渲染阶段，通用的手段是通过 instrument，profile 出耗时长的任务，对任务针对性地做方法优化。如果有的方法是第三方库的，那就需要推进社区去更新。我们在

做的过程中给 Firebase 和 Google 的一些 SDK 提了很多 issue，对方开发人员配合很积极，对我们帮助很大。

三、成果如何

通过长期的优化，以上手段全部用完之后，我们的启动时间从原来的 2 秒，优化到 1 秒以内。

总结

在优化启动时间的过程中，我们的收获不仅是对启动时间的优化，也对系统的启动机制有了更深的了解，同时优化了我们自己的代码，使其变得更加更加健壮和高性能。

携程 Web CI/CD 实践

【作者简介】西杰，携程软件技术专家，关注前端技术及其生态，致力于提升前端开发效能及质量。

一、背景

在携程的日常 Web 开发生命周期中，本地代码开发阶段可通过 NFES 框架（携程内部一个支持 SSR 框架，其中还包含许多公共基础业务模块及 UI 组件）来快速完成项目需求。但开发完代码之后常常会遇到如下几点问题：

- 环境问题：Web/Node 资源本地构建/测试环境和生产环境差异化大，导致有些问题无法及时发现
- QA 流程：Web/Node 端提交代码流程没有规范的 QA 环节，代码质量不可控
- 构建流程：资源本地构建与镜像构建是独立的，版本易混淆
- 代码开发完后的各个步骤比较分散，难集中管理

二、目标

通过引入 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。

三、实现与实践

NFES 的 Web CI/CD 的实现，简单来说就是通过管道化（GitDev Pipeline）的执行过程来完成持续集成和持续交付，这篇文章先不涉及持续部署。

其管道（Pipeline）中集成 QA，资源构建，生成镜像等多个 Stage，而每个 Stage 中都包含详细的 Step 来完成其功能。接下来我们来详细从管道（Pipeline）中的 Stage/Step 的角度来介绍下 NFES 的 Web CI/CD。

管道在这里可以理解为实现目标的顶层组件，整个 NFES Web CI/CD 就是这样的组件组合而成。目前 Web/Node 相关的管道分为三个 Stage：

1）Install Stage

a. Install Step，安装用户项目下的依赖模块

2）Verify Stage

这里集成了三个 Step：

a. Lint Step，静态代码检测
b. Test Step，单元测试/UI 测试
c. Build Step，资源构建

3）SonarAndImage Stage

a. Sonar Step，Sonar 代码检测并上传，此步骤依赖于 Verify Stage 中的 Lint/Test Step
b. Image Step，构建 Docker 镜像，此步骤依赖于 Verify Stage 中的 Build Step

上面三个 Stage 是依次顺序执行，而在同个 Stage 中的多个 Step 则是并发执行的。这些执行顺序的控制可通过编写.gitlab-ci.yml 文件来完成。这里先简单介绍下.gitlab-ci.yml CI/CD 配置的编写。

.gitlab-ci.yml 是放在仓库根目录中的文件，默认仓库会去这个文件中读取 CI/CD 的相关配置。在此文件配置中你可以定义如下：

- 定义环境变量
- 需要顺序或者并行运行的脚本命令
- 前后 Step 依赖关系
- 此 Step 所需使用缓存和设置缓存
- 触发的条件分支

具体常用配置代码如下：

```
#配置所需的基础镜像地址
image: xxxxxxxxx

#配置相关变量
variables:
  PROXY: http://proxy
  HTTP_PROXY: $PROXY

#配置 Stages 的名称及顺序
stages:
  - Install
  - Verify

……
# Install Stage 的详细配置
Test:              #Step 的名称
  stage: Verify    #属于哪个 Stage
  artifacts:       #配置产物存档文件,可在 Pipeline 运行界面取到配置的文件,但此存档只能
```

```
保存默认一周
    paths:
      - reports/
    exclude:          #忽略某些文件不作为产物存档文件
      - .git
      - .git/**
    when: always
  cache:              #配置缓存
    key: keyName
    paths:
      - node_modules   #所需缓存的文件/文件夹
    policy: pull       #如需获取缓存的文件,这里定制 policy 属性为 pull
  allow_failure: true #此步骤是否允许失败,如果允许,即使步骤执行失败,仍旧可执行下个
Stage
  dependencies:        #配置此 Step 依赖哪个 Step
    - Install
  script:              #配置所需执行的命令
    - cd /testFolder
    - node index
  only:
    - master           #配置分支 只有配置的分支才会执行相关的 Pipeline

.....
```

在日常开发使用中，携程的 GitDev CI/CD 则提供公用的配置模版，如用户没有特殊 Step 的需求，可通过选择 Step 模版或者选择应用类型模版来自动生成上面的配置文件，无需关注 yml 的详细配置。

接下来我们详细看下 NFES Web CI/CD 的 Install，Verify 和 SonarAndImage 三个 Stage 做了哪些事情?

3.1 Install Stage

Install Stage 中只包含一个 Step，即执行安装用户项目下的模块依赖。此阶段安装结束后的 nodemodules 则会作为缓存给之后的 Step 使用，可节省很多不必要的重复安装模块的时间。当然如果在同一个 commitID 的情况下，多次执行这个 Install Stage，则后面几次安装的 nodemodules 其实就是取第一次安装的缓存。

3.2 Verify Stage

Verify Stage 默认会包含三个步骤 Lint，Test，Build。这个 Stage 其实是一个规范的 QA 环节，而 Build 的 Step 为什么要放在此处，就是想构建与测试并发执行，从而缩短整个 Pipeline 的运行时间。

详细的各个 Step 的实现如下：

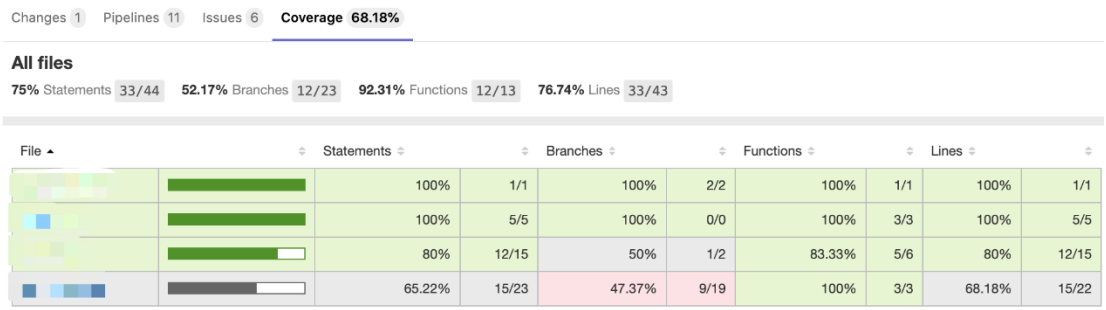
1）Lint Step 集成了 eslint 静态代码检测功能

静态代码检测功能通过封装的全局模块来完成代码检测，其默认使用 eslint:recommended 推荐规则。如用户需要自定义 eslint 规则可以直接把规则写在当前项目的 eslintrc.json 文件中，模块会自动整合其默认规则。如想要忽略检查某些文件，则把规则写在.eslintignore 文件中。执行完成后会生成 eslint-report.json，此文件会作为 artifacts 可在 pipeline 的 step 任务页面中直接下载查看，也会通过后面的 Sonar Step 上传到 Sonar。

2）Test Step 集成了单元测试以及 UI 测试

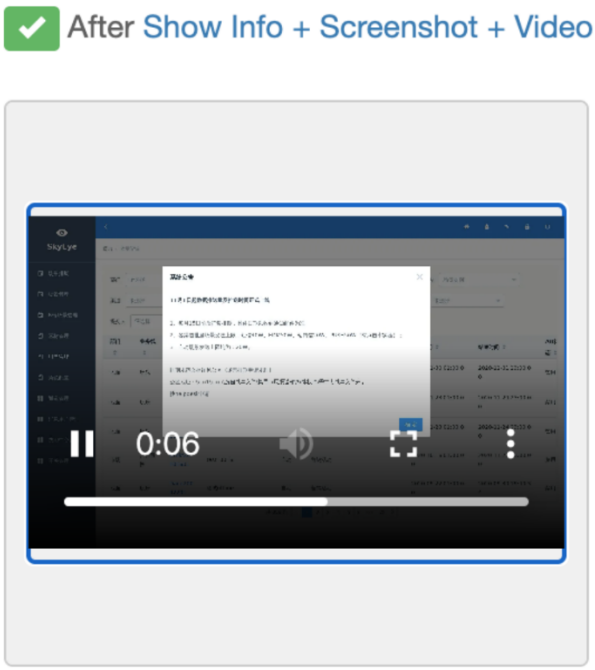
集成的单测框架又可分为 mocha 和 jest，Web 端统一使用 jest，NFES 测试镜像中默认已有 jest 相关模块，如无特殊需求则不需要在用户项目的依赖中安装测试相关依赖的模块。如需自定义 jest 相关配置可写在用户项目下的 jest.config.js 中。单元测试的运行命令统一为：npm run test，其执行结果会以 html/json/clover/lcov 输出，输出结果中 lcov 和 clover.xml 文件与 GitDev 做集成，使其结果与代码的 commitID 进行绑定，这样每次代码提交就可在界面上直接查看本次提交代码的具体单测运行结果。这里也可设置对每次代码提交的单元测试覆盖率的要求，如其覆盖率不低于 60%，否则不能进行下一步骤。

每次代码提交的 CommitID 的单元测试结果展示如下：



集成的 UI 测试是作为一个可选 Step，我们提供了集成 puppeteer/cucumber 的镜像，用户如有 UI 测试的需求可自行在 Test Stage 中添加该 UI Test Step。在 UI 测试中增加了视频录制的功能，每个 Case 对应一个视频，等用户的 UI Cases 执行完成后，则会自动生成报表并发布到资源站点上，方便用户查看及排查问题。

UI 测试报表结果中录制视频（部分截图）展示如下：



3）Build Step 集成页面的资源构建

这里的构建其实就是把在线构建搬到了 Pipeline 的 Build Step 中。首先是构建环境的搭建，分为两块：框架所需的依赖模块环境和用户项目依赖的模块环境。

关于 NFES 框架的依赖模块环境，Build Step 使用的构建镜像中已经集成了 NFES 项目所需的开发态模块（我们对开发态模块加载做了些优化，把如 Babel 插件，webpack，loader 等通用的模块全都集成到 cli 的全局模块中，然后预装到构建镜像)。执行构建时，更改构建时项目所需开发态模块路径指向预装路径，这样就可以不需要安装框架依赖模块。而对于用户项目依赖的模块环境则可以重用 Install Stage 中的 node_modules 中的缓存，这两点使用户项目安装模块的时间大幅度减少。

搭建完构建环境后，执行相关在线构建命令开始构建，构建的过程及日志都可通过 Pipeline 界面得到。构建完成后接下来是构建产物的处理。这里的 NFES 项目构建产物可分为 Web 端资源/node 服务端资源。Web 端的资源可以直接发布并获得相应的资源地址，此 Web 资源地址也会及时更新到 node 服务端资源中的资源路径。最后通过配置中 artifacts 属性来确定哪些 node 端的资源文件需要上传给下一步 Image Stage 来构建发布镜像。

3.3 SonarAndImage Stage

SonarAndImage 包含了 Sonar 和 Image 两个 Step， 这个 Stage 是目前管道中最后一个专门收集与处理前面依赖 Step 产物的 Stage。

1）Sonar Step

此步骤是依赖于 Test 和 Lint 这两个 Step， 用来收集依赖的这两个 Step 执行的结果并上传

至 Sonar 中。用户可以在 sonarqube 的网站查看历史的代码质量报告。

2）Image Step

此步骤是依赖于 Build Step，它是获取 Build 的构建产物与基础镜像一起构建出发布镜像并推送到 Hub 中，为接下来的应用发布做准备。到此步骤整个 NFES Web CI/CD 的流程就结束了。

四、小结

以上就是整个 NFES Web CI/CD 的实现与实践。目前几乎所有的 NFES 项目都已经切到 CI/CD 的流程上，它带来了集中式流程化管理，一站式对用户透明的资源构建与镜像构建更简单快捷，开发效率得到了很大的提高。

前端跨端业务整合的探索与实践

【作者简介】Jeff，携程前端开发经理，对前端自动化技术感兴趣，推动了团队使用 cucumber 进行 UI 自动化测试。Harry，携程前端开发工程师，秉持“Don't make me think“的理念向用户交付页面、向同事协作工程。

为应对携程国际化的需求，机票前端团队开始业务统一化的步伐，Ctrip 和 Trip 的业务整合和代码复用成为面临的困难和挑战。在实践过程中，团队积累了大量的经验，下文是机票实现业务统一化、技术中台化、迭代敏捷化的思路和方法。

一、背景

Trip 与 Ctrip 为独立运行的两个站点，虽存在各自品牌化的差异，其业务功能有着极高的一致性。两个站点相互独立开发与维护存在着以下的问题：

1.1 技术架构不统一

Trip 与 Ctrip 使用的开发技术栈存在较大差异。Trip 订后场景在 APP 端使用 Native iOS、Android 开发，H5/PC 端采用 React 技术；Ctrip 订后项目使用可在 iOS 及 Android 双端运行的基于 React Native 的 CRN①框架，在 H5 端采用 CRN-WEB②进行动态打包将 CRN 代码生成对应 H5 页面。

两个站点整体技术架构上多种技术方案并行，相同的业务逻辑需要在各端分别实现，在打包发布流程中，各端需要通过不同的方式进行相关操作（如 MCD③、Ares④、PAAS⑤等）。再加上订后场景业务维护复杂性比较高，开发周期冗长，两个站点分别开发的效率不容乐观。

1.2 功能迭代存在冗余

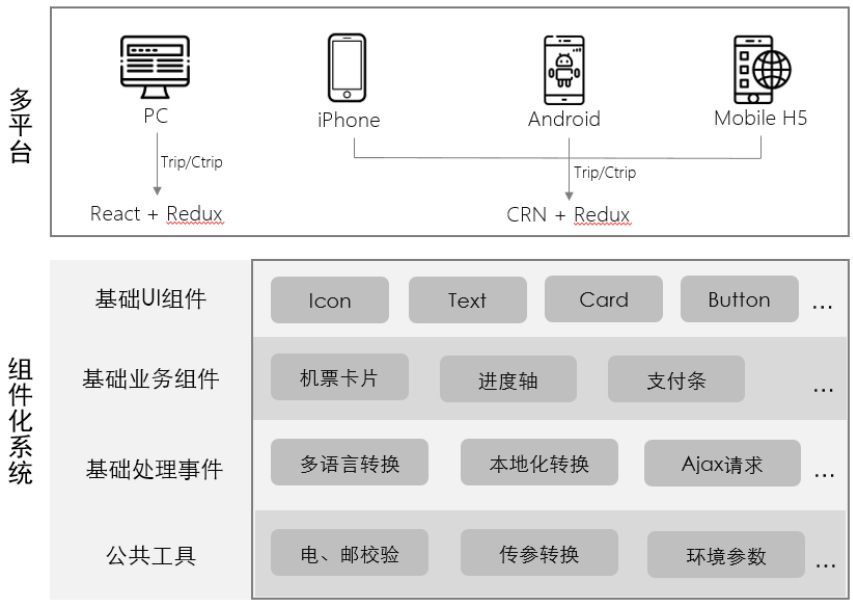
由于技术架构的不统一导致在业务维护上需要分别进行开发迭代，在开发效率上存在很大的冗余，同时开发团队需要面对多种技术栈，学习成本和开发成本都非常高。相同的业务在多团队重复开发时也难以实现功能与进度上的对齐。

1.3 发布与监控分散

正由于第一点提到的，Trip 与 Ctrip 使用了不同的技术框架分别进行开发，其打包、发布及日常埋点监控、数据维护存在极大差别且难以整合。Trip 的 iOS、Android 使用的是 MCD③平台使用双频道分别进行打包与发布，而 H5 页面需结合 Ares④平台的打包以及 PAAS⑤的发布管理。相对而言，Ctrip 对于 APP、H5 端使用 MCD 打包发布，PC 端使用 Ares 与 Captain。对于不同的开发代码，相关的数据埋点纬度与落地方式也不同，分散的埋点数据给之后的统一分析带来了困难。

因此 TRIP 和 CTRIP 技术架构的统一，共同维护一套流程，那将极大解放开发资源而着重于更有意义的业务探索，也缩小了其后维护流程的成本。在改造过程中，我们将技术栈统一，

将原先 iOS、Android、H5 替换为 CRN 架构，将 PC 替换为 React 架构，并在此基础上建造了模块化的基础组件，打造前端中台化产品。本文将针对前端中台化改造的探索做出阐述。



整体架构图

// 章节尾注

① CRN：Ctrip React Native，携程对于 React Native 的再封装，提供多种业务部门可以直接使用的基础工具；

② CRN-Web：携程提供的将 CRN/RN 转为 H5 页面的工具，使 APP 页面能在浏览器上展示。

③ MCD：携程无线持续交付平台。含多类型项目的集成、测试、发布、运营等多种服务。

④ Ares：携程提供一套前端研发整体解决方案。从编码、测试、编译、发布等多个环节整合前端资源的开发。

⑤ PAAS：携程研发服务平台。一站式提供多种服务产品。

二、面临的挑战

面对庞大繁复的业务逻辑、Ctrip 站点与 Trip 站点的表现差异，中台化开发两边的产品线并不是一个简单的改造。就前端而言，将现有的国内站点代码直接套用于国际站点是一个海量级的改造工作，但经过仔细拆分，难点分类列出并逐个击破其实是一个可量化好控制的迭代流程。

1）组件共用；
2）样式拆分；
3）暗黑模式的适配；
4）多语言的适配(i18n)
5）国际单位的本地化(l10n)
6）基于 Gitlab pipeline 的自动化测试
7）发布与监控

针对以上各个方面，机票前端开发做出一些探索供大家探讨。

三、解决方案

3.1 组件化开发

为了使一套代码能驱动仍存在差异的 Ctrip 与 Trip 流程，首先需要将公用的且因平台存在差异的模块或功能抽象化为组件。机票订后流程开发技术栈基于 React Native + Redux 的技术框架，控制流程逻辑的 action 和 reducer 一层可以高度重用。然而和视觉相关的 View 层需要做品牌化区别、不同平台的语言需要不同的翻译结果、响应同一操作的服务请求与底层处理逻辑也会有些许不同。由此搭建一套兼容两端的公共组件库是拼接一切业务的基石。

为了满足各个功能的兼容性，公共库包含了原子 UI 组件、业务基础组件、基础处理事件、公共工具等四象限纬度的组件：

公共库四象限组件

原子 UI 组件：包含了页面展示的基础 UI，包括 Icon、Card、Button 等。其组件与上下文无关，更多是在针对 Ctrip 及 Trip 不同平台进行品牌化差异的样式处理（详见第 2 小节）、基础事件的绑定和必要的曝光点击等埋点的处理。

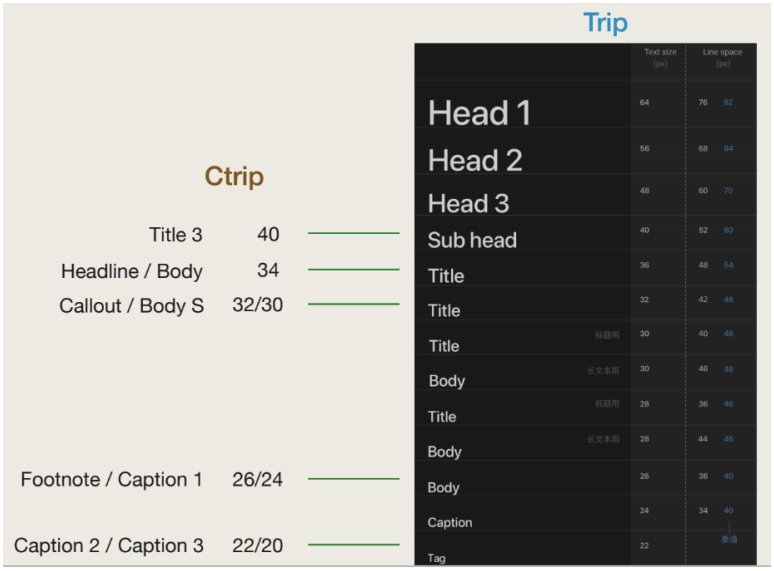
基础业务组件：是针对一个原子业务块的 UI 封装，例如机票卡片、进度轴、运价明细卡片等，通常需要依赖上下文数据的传入。一个业务组件虽然依赖的数据源往往是一致的，但其组装起的基础 UI 组件、页面的排版格式往往存在一定的差异。封装之后的业务组件可使业务开发无需考虑其中的展示、排版逻辑，使用统一的数据源及事件操作可达到相同效果。

基础处理事件：包含了相同性质的业务事件处理，例如 Ajax 请求的发送、业务埋点的发送、多语言的转换（详见第 4 小节）、国际单位的本地化转换（详见第 5 小节）等。

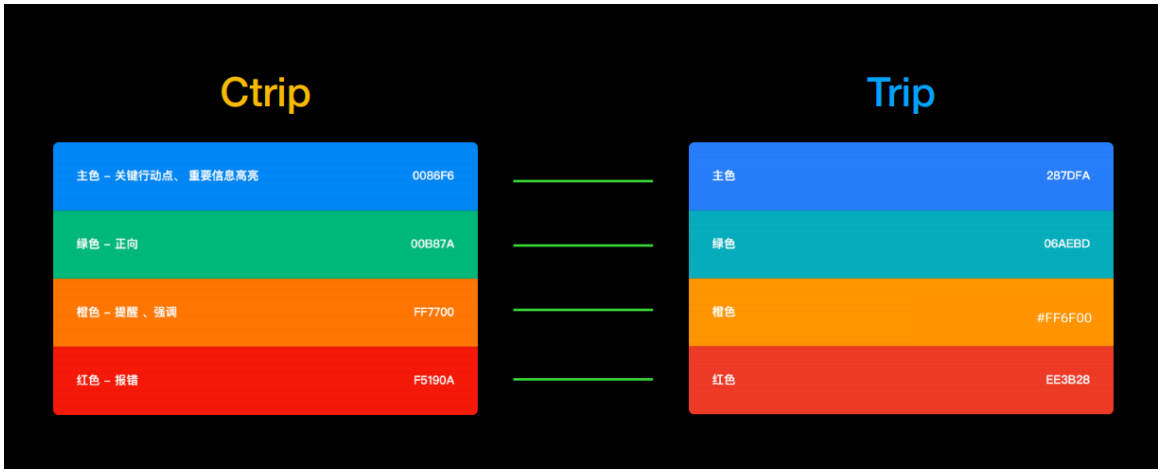
公共工具：则提供与上下文无关的纯函数处理工具，例如电话号码的正则校验、url 参数的转换等等。保证对其调用不会受方法以外的环境影响，也不会影响对外的环境。

3.2 样式拆分

在追求流程统一化的前提下，Ctrip 和 Trip 的品牌化视觉体验还是有很大的差异。两端针对字号、颜色、头部样式、弹窗样式、甚至圆角都有各自的标准。



Ctrip & Trip 字号大小映射表



Ctrip & Trip 颜色映射表

为了解决两端样式的适配，公共库封装了技术样式表组件。改造初期对于整个流程针对字号和颜色进行了一次整理，将流程所使用到的字号和颜色总结到了一张基准样式常量表，再将常量表再跟进国际站点的标准重填入对应的值，并写入样式表组件库。之前写到样式表里的字号和颜色全部改为引用样式表里的常量，而用哪张表则取决于当前是哪个站点的 APP。抽离常量的过程虽然繁琐，换来的是两端的代码可以尽可能得使用一张样式表。

```
const BasicIBU = {
  // Fonts
  subHead: 20, // sub head
  headline: 18, // title
  headlineS: 16, // title
```

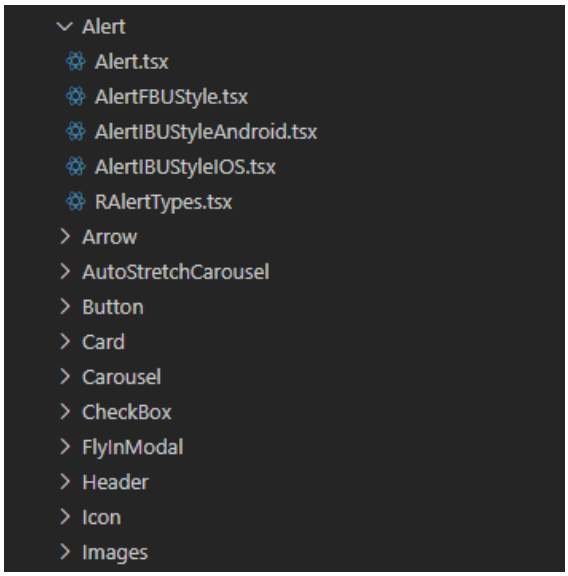
```
  bodyText: 15, // body
  callout: 14, // title
  textNormal: 14,
  bodyTextS: 13,
  footnote: 13, // void
  caption1: 12, //caption
  // ...

  // Colors
  black: '#0F294D', //  基础黑色字
  secondaryBlack: '#455873', //  基础黑色字  二级黑
  tertiaryBlack: '#8592A6', //  基础浅黑色字  三级黑
  titleBlack: '#042950', //  标题黑色字
  grey: '#ACB4BF', //  基础灰色
  switchGrey: '#AAA', //  切换按钮灰
  lightGrey: '#F5F7FA', //  基础浅灰
  lineGrey: '#DDDDDD', //  边框灰
  placeholderGray: '#F0F2F5', //  呼吸态灰
  // ...
}
```

IBU 基础样式常量表截取

```
modalTitle: {
  textAlign: 'center',
  fontSize: BasicStyles.headline,
  color: BasicStyles.black,
}
```

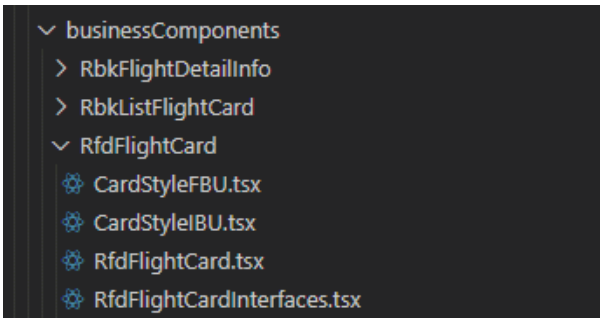
一个简单的样式表

有了字号和颜色的基础，可以在这基础上开发出两端共用的基础 UI 组件与基础业务组件。例如页面头部、选择弹框、抽屉浮层等。因为基础组件的交互逻辑一致，不同的只是两端（或者三端：国际站点针对 IOS 端和 Android 端有不同标准）的表现样式，所有的公共组件都是针对逻辑写了一份共用的 JS 逻辑以及针对渲染层级写一份共用的 JSX Dom 表。JSX Dom 表上绑定的 StyleSheet 则是针对性得读取三张表（FBU、IBU IOS、IBU Android）的样式内容。而样式表中的字体、颜色使用基础样式表的封装便可按图索骥渲染不同的品牌样式。



公共组件目录结构

同样的，在业务开发过程中，非基础组件的 View 层也需要区别开发。因为业务逻辑相同，各个业务场景往往只需制作一套 JSX Dom 表，而对于 FBU 和 IBU 往往需要两套样式表。



业务组件目录结构

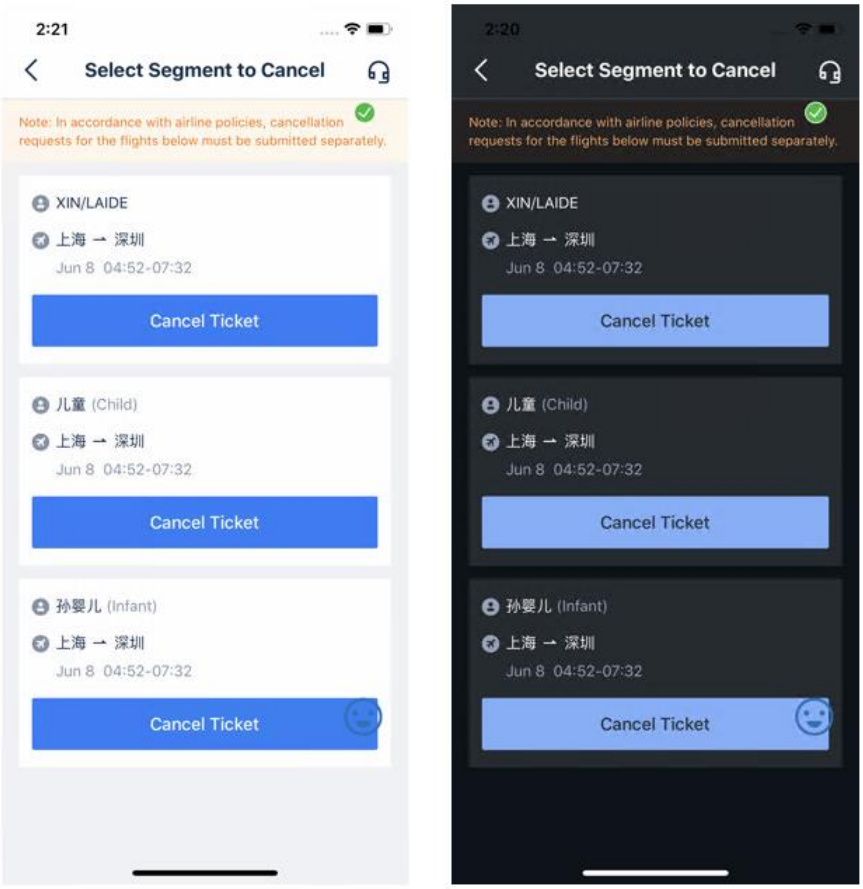
3.3 暗黑模式的适配

除了常规的两端样式差异以外，为了加强品牌效应，流程还需要支持暗黑模式（DarkMode）。DarkMode 在转换时，看似只是将颜色做一个简单的白转黑，黑转白映射转换，实在底层有很多让人头疼的逻辑。

首先并不是白色都转换为统一的白色，明亮模式下白色卡片相互叠加因为有黑色边框或者黑色阴影的隔离，层级区分很自然明细；然而在暗黑模式下，自然黑色的边框和阴影并不能将黑色的卡片有效的区分开来，所以需要将所有白色做语义化区分，不同层级不同语义的白色转换为不同深浅的黑色。如此，一个 #FFF 白色，可以根据场景语义化区分为背景纯白 ThemeWhite，主要内容白色 PrimaryContentWhite，次要内容白色 SecondaryContentWhite 等，分别对应的暗黑模式色值为`#0D131A`#0D131A，`#252B31`#252B31，`#333B46`#333B46。

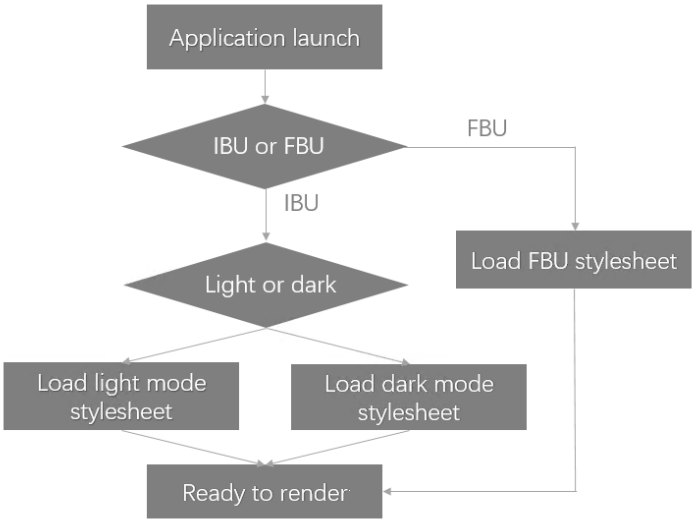
其次，如上面提到的阴影和边框等拟物色，在暗黑模式下不能转换（自然界中未有过白色的

阴影吧)。需要将这些拟物色剥离出来（如阴影的 ShadowBlack)，在暗黑模式下不做转换。最后，所有的彩色在亮度更低的暗黑模式下需要转换为饱和度更低的对应颜色。例如警戒红色从#EE3B28#EE3B28 映射为#F37668#F37668，品牌蓝色从#287DFA#287DFA 映射为#7EB0FC#7EB0FC。



明亮模式&暗黑模式对比图

颜色的映射规则弄清楚了，那怎么把暗黑模式应用到流程的呢。这就要回到在样式品牌化章节提到的基础样式表，FBU 站点有一张基础样式表，IBU 有一张基础样式表，只需要将原来的 IBU 基础样式表作为明亮模式的样式，再在此基础上映射出一张暗黑模式表。在 APP 启动阶段动态得判断当前所在的模式，并加载对应的样式表。



加载基础样式表流程

3.4 多语言的适配（i18n）

国际化的改造自然离不开多语言的适配（i18n 即 internationalization 的简写，由首尾的 i、n 及中间的 18 个字母组成）。此次机票订后流程的多语言翻译及加载机制依托于携程的 Shark 多语言整体解决方案⑥。Shark 翻译平台以及框架提供的 i18n 组件已经是相当成熟的一套系统，这里不再赘述。这次改造的难点还是在如何在已有的流程中抠出需要翻译的文本，以及管理各页面翻译文本的加载。

在流程改造初期，一个繁重但必不可少的工作就是在全流程代码抠出需要翻译的展示词条。为了方便管理以及优化资源分配，整个业务层将词条分页整理为多个数组：其中全流程都使用的基础词条（如“确定”、“取消”等）单独列为一个数组；而页面独有的词条根据页面纬度分别建组。数组里的每个词条实体包含一个键值对，键为提供给 Shark 平台翻译唯一标记的 key，值为其 key 对应的默认简体中文文案。

```
// 通用 KEY
export const sharkCommon = {
  goToDetailPage: {
    id: 'key.flight.postservice.common.godetail',
    defaultMessage: '去订单详情',
  },
  calendarTitle: {
    id: 'key.flight.postservice.calendar.title',
    defaultMessage: '选择日期',
  },
  // ...
}
// 退票选程页用的 KEY
export const refundSelectTrip = {
  changeTagTxt: {
    id: 'key.flight.postservice.refund.select.change.tag.txt',
    defaultMessage: '航班调整',
  },
  refundBtnTxt: {
    id: 'key.flight.postservice.refund.select.refund.btn.txt',
    defaultMessage: '申请退票',
  },
  // ...
}
```

shark keys

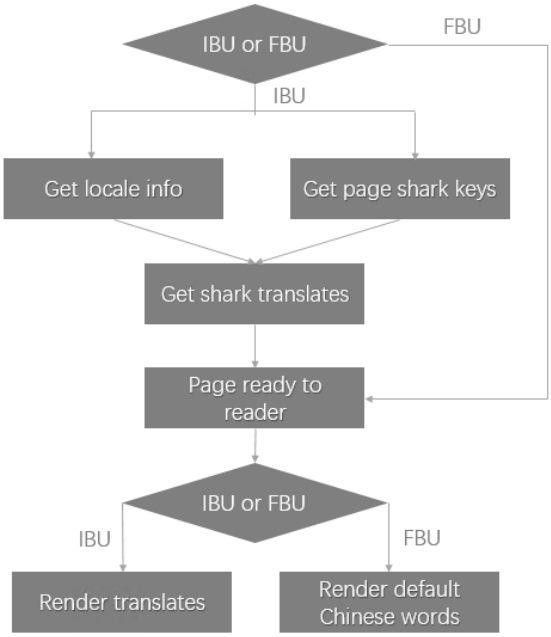
页面加载时，会根据各个页面使用情况动态加载去加载翻译文本。每个页面继承了一个基础的页面组件（CommonBasePage），组件加载后（于 RN 的生命周期 componentDidMount）首先需要锁住页面的渲染展示加载态，这时两条业务线的加载逻辑略有不同。

针对 Trip 站点，加载态时需要拿着当前页面渲染使用的翻译词条给到 Shark SDK 申请翻译结果，翻译词条一般包含之前划分好的公用词条组和当前页面特有词条组，Shark SDK 会拿

词条的 key 与当前手机配置的区域语言匹配到翻译后文案并返回给业务端，当翻译返回后放开页面的加载态继续进行页面后续的渲染工作。而针对 Ctrip 站点，不需要向 shark 平台请求翻译结果，所有内容都已包含翻译键值对的默认翻译中，则直接跳过获取翻译这一步，并取消加载态进入后续的页面渲染。

Shark 平台其实也提供了简体中文的翻译，那么为什么不将 Ctrip 业务线的展示词条也托管到 shark 平台统一管理呢？

这里有几方面的考虑：首先 shark 平台的简体中文需要开发和产品自行翻译且维护，和维护键值对的默认中文并无区别；其次，通过 Shark SDK 加载翻译需要额外的外部依赖调用，且在目前流程是阻塞式的，页面稳定性及页面加载效率来说不如本地读取键值对的方案；最后，Ctrip 团队针对业务线已写有大量的 UI 自动化测试及单元测试且已接入 CI/CD 持续化构建平台，如若每次测试都需要额外调用 Shark SDK，稳定性及自动化构建的效率也会受到挑战（关于自动化测试相关解决方案之后章节有更详细讨论）。



页面获取翻译流程

在流程上线之后，仍需要对翻译结果查漏补缺，监控可能出现的因漏提翻译或系统错误导致的展示中文的情况。好在前端的所有文字展示都使用 Text 基础拓展组件，组件在触发渲染时对子元素所包含的字符串做一次正则检测。在 Trip 环境中若正则检测到中文，则发送一次警告。开发可以方便得通过警告信息关联的订单号或流水号定位到系统展示中文是因为该字段是漏提交了翻译还是系统错误造成的。（除去中文检测，此正则式还可以很方便得检测出 null、NaN、undefined 等页面的错误展示。）

```
static checkChildren(children?: ReactNode) {
  const chineseReg = /[\u4e00-\u9fa5]/;
  if (StringChecker.needCheck && children && typeof children === 'string') {
```

```
      if (chineseReg.test(children)) {
        if (!StringChecker.data[children]) {
          StringChecker.data[children] = DataStatus.DEFAULT;
          Expose.sendError('Error:find chinese', { str: children }, true);
        }
      }
    }
}
```

查找中文并报警

另外，针对一些特殊场景如人名等，可以配置相应的可忽略检查的语言以及业务模块。

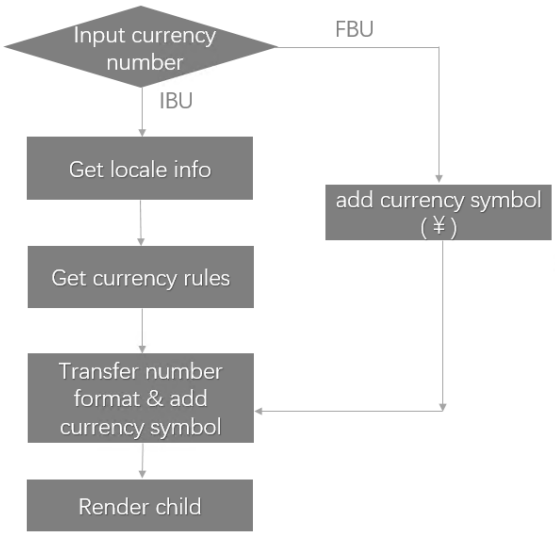
// 章节尾注
⑥ Shark：携程提供的多语言站点 UI 文案管理与翻译的一整套解决方案。实现提供原文后交于统一交于翻译团队，并通过其提供的 SDK 工具于业务代码中抓取下发对应翻译后的多语言结果。

3.5 国际单位的本地化（I10n）

和国际化（i18n）相对的，Trip 站点也许考虑各计量单位的本地化（I10n 即 localization 的简写，由首尾的 l、n 及中间的 10 个字母组成）。每个国家和地区对于货币、时间、重量、距离等的展示标准各有差异，因此需要根据 APP 所设置的地区与语言，动态得去转换所展示的计量数据格式。

例如时间的展示，不同的区域会展示如“01/01/2020 Monday”、“2020/01/01 月曜日”等格式。决定时间以何种格式展示，方法类似于上一章节的多语言翻译。基础页面组件（CommonBasePage）加载翻译语言词条时，也会拿手机当前语言及地区向 Shark SDK 请求对应的基础计量单位展示格式制式包，其中包含了诸如日期、重量、数字等计量单位展示时所使用的标准格式，之后的业务代码再将具体需要转换的数字向对应的格式进行转换。这样就使服务下发或计算出来的唯一格式的时间根据不同的 APP 设置转换为不同的格式。

货币，重量、距离、数字的千分位展示及小数默认位数等的个数都需要根据不同的地域语言做区分。转换使用方式类似，这里不再赘述。

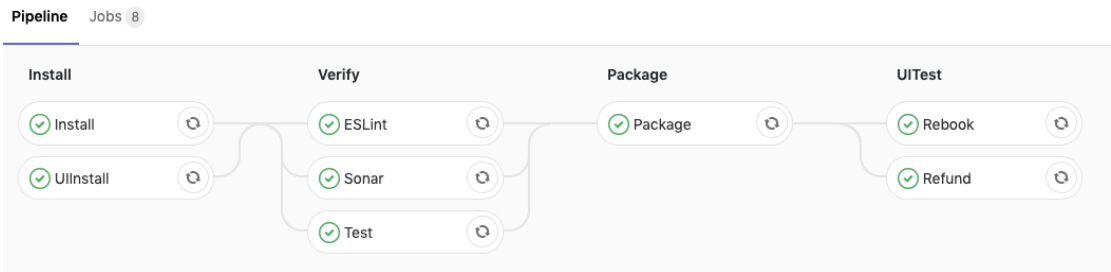


Currency 转换

3.6 基于 Gitlab Pipeline 的自动化测试流程

在质量这一块，除了常规的 UT 之外，机票前端团队做了大量的自动化测试，这些自动化的流程适配于中台化开发的流程中，保证了 Ctrip 和 Trip 代码的质量。

我们基于 Gitlab，在其 Pipeline 中加入相应的检测机制，把 Soanr、UT、UI 自动化等流程融合到流程中，确保每一次代码签入和 MR 都执行成功。



在 UI 自动化测试实现过程中，内核采用的是 Cucumber⑦和 Puppeteer⑧运行业务代码的 H5 版本来实现测试。我们将 CTRIP 和 Trip 的测试步骤拆分（增加 isIBU 字段作为标识），在 Cucumber 底层会根据此标识区分测试 Trip 站点与 Ctrip 站点对应的页面。

针对于多语言环境，我们执行在 Trip 站点用例时会分情况进行判断。如若此用例更多偏向测试业务逻辑而不太在意翻译的内容与质量，运行过程将屏蔽获取翻译的流程，直接使用默认的简体中文内容，这样我们的测试用例断言内容全部都不需要修改，并且屏蔽了因翻译服务不稳定、翻译内容时常变更带来的比确定性。而若此用例需要考虑进测试多语言翻译结果的正确性，则可以给予标识打开翻译流程，实时获取翻译内容进行校验。

```
Scenario Outline: 国内_改签_新版改签权益_（多人/单人）同种权益_改1人 B /A /C /D 部分单一：2，3，4
  Given 配置MOCK数据
    | getFlightCondition          | /getFlight.js                                |
    | QueryRebookCondition        | <QueryRebookCondition>                       |
    | CertificateValidity         | ../../../CertificateValidate/Certificate.js |
    | FlightTicketChangeSearchDom | /list.js                                     |
    | FlightDetailSearchDom       | /detail.js                                   |

  Given 打开'改签申请'页
    | orderId | 123456 |
    | isIBU | <isIBU> |
  #    Given 接口'合并订单选程'成功返回
  Given 接口'改签条件查询'成功返回
  Then 进入'新版改签申请'
  And wait 1s
  Then 检查'新版改签申请'页面包含以下信息
    | 改签权益入口文案 | <改签权益入口文案> |

  Then 点击'第一个可改乘机人'
  Then 点击'第一个日历'
  Then 点击'第一个可选时间'
  Then 点击'选择改签航班'
  Given 接口'国内退后重订列表查询'成功返回
  Then 进入'h5国内退后重订列表页'

  And wait 2s
  Then 检查'h5国内退后重订列表页'页面包含以下信息
    | 改签权益入口文案 | <改签权益入口文案2> |
  Then 点击'改签权益入口'
  Then 检查'h5国内退后重订列表页'页面包含以下信息
    | 改签权益浮层 | <改签权益浮层> |
```

// 章节尾注

⑦ Cucumber：一个基于行为驱动(BDD: Behavier Driven Development)的开发测试工具（https://cucumber.io/）；

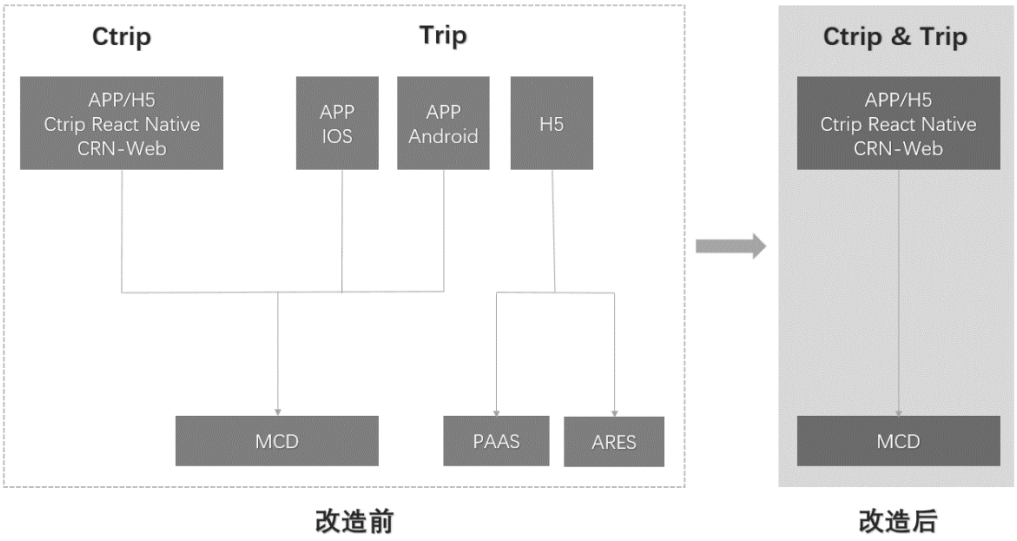
⑧ Puppeteer: 一个通过提供高阶 API 用于控制 Chrome 或 Chromium 的 Node 工具集（https://pptr.dev/）。

3.7 发布与监控

相较于多端分开开发，中台化开发还带来一个好处是发布更加便捷、易于管理、更新更加及时。

原先发布就一个需求而言，全平台上线需要先后于 MCD 平台分别发布 IOS、Android 版本，于 Ares 打包发布 H5 的静态资源，于 PAAS 平台将 H5 打包结果发布生产站点。特别的，IOS、Android 需要跟随 APP 主程序包的更新进行发布，这样便限制了 APP 端业务的发布节奏。也就是说进行单频道的热更新修复或者紧急需求上线比较困难，并且上线之后未更新的客户端仍无法使用最新的业务逻辑。

进行中台化开发后的订后产品，使用相同的技术栈，在 APP 端采用 CRN 框架开发，在 IOS、Android、H5 统一使用 MCD 发布系统进行打包发布，避免了多平台发布的差异性。并且使用中台化开发后，所有站点需求会在统一的发布时间节点进行统一发布，意味着 Trip 站点和 Ctrip 站点的需求可以统一管理发布上线。使用 CRN 还可以很方便得在 APP 内进行热更新，和 APP 版本发布相解耦，实现了需求的随发布随使用，解决了紧急修复难于上线的困难。



统一前后的发布逻辑

在监控上，Ctrip 和 Trip 站点接入统一的监控平台，数据采集统一汇总到 hickwall 中，制作相应的监控面板和告警规则，针对客户端 Error 统一采集并分发到 Bigeyes 管理平台，同时针对上面提到的异常空字符串也会做到及时监控、及时告警。

四、小结

以上几点便是机票前端后服团队针对 Ctrip 和 Trip 实现中台化开发的一些探索，虽然磕磕碰碰绕过弯路，现在仍有一些待优化的点。但阵痛过后的收获是满满的：现在 Ctrip 和 Trip 站点在 APP、H5、Online 三线业务逻辑统一，相同功能迭代的开发用时减少，多个站点只需要维护一套代码，开发成本得到很大的降低，开发效率也有很大的提高。

以模型为中心，携程契约系统的演进

【作者简介】章鱼，携程资深后端开发工程师，契约系统创始人与核心研发。Sylar，携程资深研发经理，专注 Java 技术栈相关研究。

一、前言

随着微服务化在携程的全面落地，业务被拆解得越来越细，接口数量和内外部调用方不断增多；另一方面，随着产品迭代的不断增速，对接口的修改也变得愈加频繁。

接口契约，作为各端的沟通桥梁，在微服务时代显得尤为重要。如何管理好不断变化的接口契约，是携程机票 BU 在微服务化过程中遇到的一大难题。本文结合一些契约管理的实践经验，介绍下携程契约管理的演进，以及携程机票 BU 自研发的契约系统（简称 MOM - model object management，以下都以此代替）。

二、携程契约管理演变史

2.1 线下管理契约

在携程机票 BU 以及其它的一些团队里，早期的契约管理，主要由程序员线下进行。当发生需求变更，需要增加新的接口，或改动原有契约时，由相应的开发线下编辑契约文件（如 XSD 文件），再生成指定格式的契约（如 Java 类，Protobuf 文件），提供给使用方。

在单体应用时代，或业务没有拆分很细的时候，接口契约并不会太多，调用方也相互熟悉，只要有专人负责线下契约维护，这种管理方式并不会有太大问题。但随着微服务化，接口和契约不断增多，此种管理方式的瓶颈逐渐凸显，问题主要集中在以下几点：

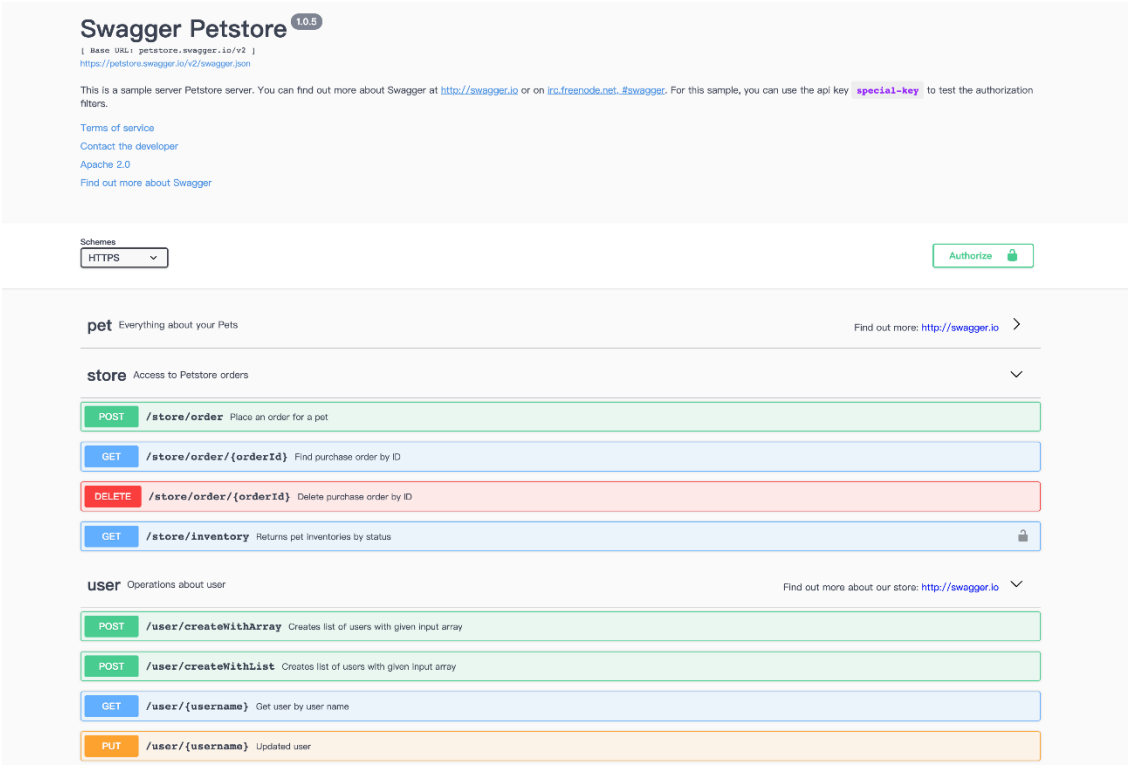
1）线下管理，契约容易丢失。
2）契约变更，只有编辑者才能主动感知，无法很好的周知到所有使用方。若契约变更不当，容易在上线时才发现问题。
3）当有多个需求，需要对同一份契约进行修改时，需手工解决各种冲突，这个过程极易出现问题。就好比不用 Git，手工去进行代码分支管理一样。
4）契约可读性差，接入沟通成本大，各团的原始契约也没有一个统一的标准。

2.2 第三方工具管理契约

由于线下契约管理，存在太多各式各样的问题。携程内部各团队都在积极探寻，尝试借助一些第三方工具，去更好地管理契约。常见的有 Swagger、Knife4j、YAPI 等工具。

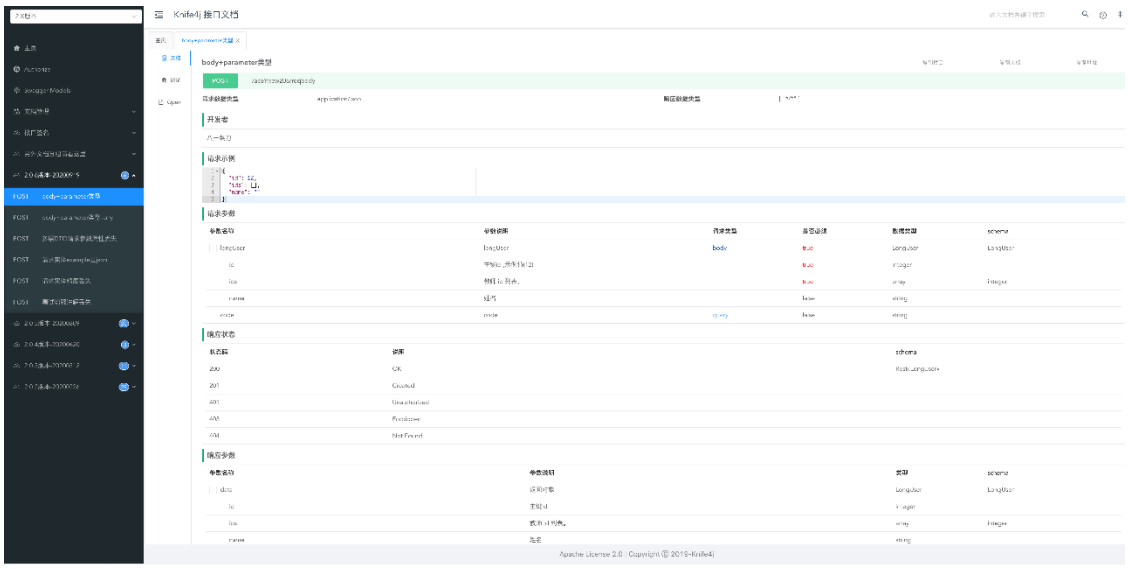
Swagger 是一款非常受欢迎的开源框架，它通过文档与代码的绑定，能很好地对 API 进行文档化描述，接入起来也比较方便。Swagger 一定程度解决了线下管理契约易丢失、可读性差等问题。但由于 Swagger 单机或集群的契约管理方式，会使得携程内部各项目（应用）之间，显得比较割裂，所以 Swagger 并未在携程内部大范围推广。同时，Swagger 的界面比较

粗糙，文档的编写需要遵守一系列的规范，上手有一定难度。



Swagger

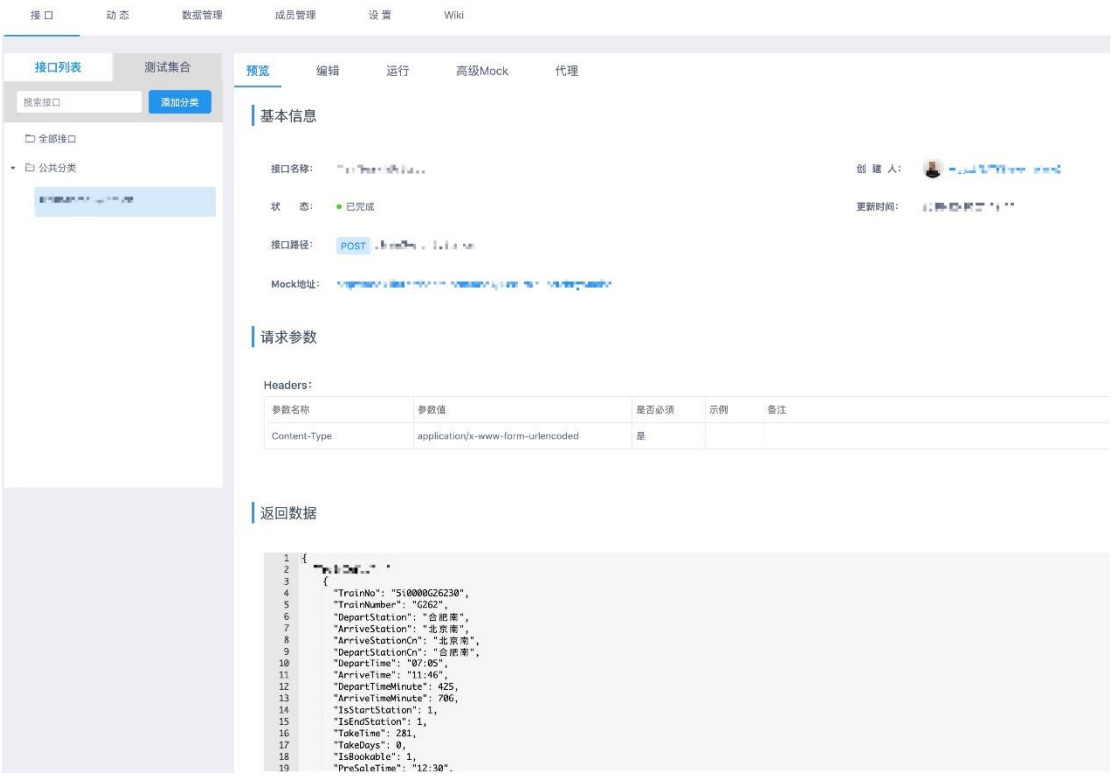
Knife4j 是一款国产开源项目，也是用于 API 文档的生成。与 Swagger 相比，它有着更加出色的界面，并且能够支持如离线文档、安全控制、在线调试等功能。但是与 Swagger 有相同问题，Knife4j 并不是一个中心化的契约管理方案。



Knife4j

YAPI 是去哪儿为推进 API 标准化，研发的一款 API 治理工具。本身也拥有丰富的功能，携程

内部也有一些团队在使用，但它的设计更注重于 API 的多样化管理，不支持模型共享，不支持单独模型聚合方式管理，必须要有接口名称和输入输出信息。这些限制给契约的管理带来不少麻烦。



YAPI

总的来说，Swagger、Knife4j 和 YAPI，将契约管理方式从线下转到了线上，一定程度解决了契约易丢失，可读性差的问题，同时还提供一些 API 管理的高级功能，如调试、Mock 等，那为什么还需要 MOM 呢？

2.3 为什么研发 MOM

携程机票 BU 结合自己的实践经验，并走访收集了其它团队在契约管理上的痛点，总结出一款好用的契约管理系统，所应当具备的功能有：

1）云端管理，项目、权限管理
2）友好的界面，支持在线编辑
3）版本管理，差异比对
4）契约变更消息通知
5）契约导入，代码生成
6）模型共享

由于 Swagger、Knife4j 以及 YAPI，对以上功能的支持或多或少有缺失，而且它们都是以面向 API 管理为主，并不能很好地解决上述问题。携程机票 BU 开始尝试研发 MOM，并在近一年内陆续在携程内部推广开来。

契约系统又称为 MOM（model object management），从名字就能看出，其管理的是模型对象。这个设计理念也是与 Swagger，YAPI 等工具面向 API 做管理的最大不同。

模型，无关于接口实现，无关于契约文件类型，也无关于具体生态环境，它仅由字段组合而成。在 MOM 中，模型作为被管理的最小单位，可以进行任意的组合与嵌套，通过这种结构化的管理方式，使得契约的描述性更加强大，突出表现在能够模型共享上。

模型管理这一理念，其实也是源于携程机票 BU 的实践经验。在 2017 年左右，携程服务端的技术栈从.Net 转向 Java，前端有 iOS、Android、React Native、Hybrid 等技术栈，一个接口对应了多份契约文件。同时，随着微服务化的深入，接口数量也暴发式增长，我们发现很多接口契约的部分内容是可以复用的，例如航班信息，主要就包含了航班号，出发到达地，机型等信息。

为了解决一份契约会有不同的文件格式，以及契约复用的问题，MOM 设计之初就抽象出模型这一概念，并将其作为底层设计的核心思想，这也使得 MOM 之后具有良好的扩展性，所有其它工具支持的功能，理论上 MOM 都能支持。

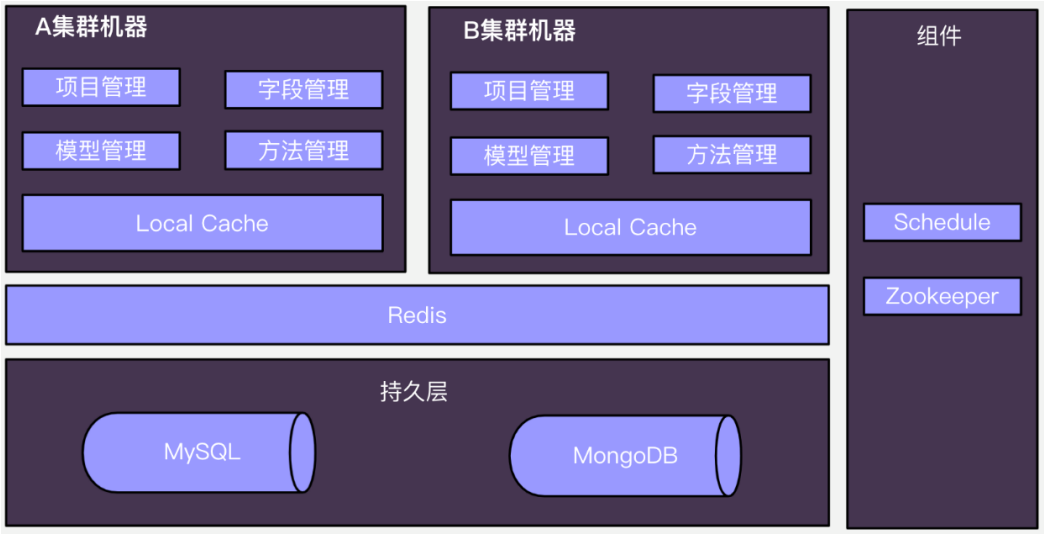
总的来说，Swagger，Knife4j 和 YAPI 主要关注的都是对 API 的管理，而 MOM 更加关注的是对契约的管理。

\		Swagger	Knife4j	YAPI	MOM
API管理	API 在线编辑	部分支持	部分支持	支持	不支持
	API 调试				
	API MOCK				
	API 自动化测试				
契约管理	契约编辑导入	不支持	不支持	部分支持	支持
	模型共享				
	多版本比对&合并				
	契约发布				
	变更通知				

三、MOM 架构

MOM 作为一个帮助大家管理契约的工具，在使用频率方面，并不会太高，并发压力不会太大。研发时更侧重于其功能的丰富性。所以，整体的物理架构不是很复杂，但支持的功能以及逻辑单元相对丰富，且留有很好的扩展性。

3.1 物理架构



MOM 采用双集群进行灾备，使用 MySQL 管理模型、项目、版本等结构化数据，Mongo 用于管理用户上传的原始文件及最终生成的契约文件。系统采用 Local Cache + Redis 的二级缓存结构进一步降低 DB 的访问压力，同时用 Zookeeper 保证各台机器的数据一致性，可能发生变更的静态数据由 Schedule 定时进行更新。整个系统使用传统的 MVC 架构进行设计，前端页面使用 Angular 搭建，服务使用 Java 进行开发。

3.2 逻辑单元

从以上的逻辑单元可以看出，MOM 主要包含的功能有：项目管理，契约编辑，多版本管理，契约生成，变更通知，以及底层的核心：模型管理。后面会对这些功能进行展开介绍，分享这些功能的相关经验。

四、MOM 功能介绍

4.1 契约编辑

契约编辑主要分为两块，契约导入和在线编辑。

4.1.1 契约导入

契约导入，顾名思义是将现有的契约，可以是各类契约文件，导入到系统中， 方便后续的查看和修改。这功能的必要性有一定历史原因，因为各团队维护契约文件的方式都不相同，契约文件类型存在差异，要想后续都在 MOM 上进行管理，就必须提供这么一个功能方便各方快速接入。

之前也提到过，MOM 的核心理念是管理模型，不是具体的接口或契约类型，这一层抽象使得契约导入功能能够很方便的实现，需要支持什么类型的契约导入，开发其相应的文件解析器，将契约文件转换成 MOM 底层维护的模型即可。

契约导入完成后， MOM 界面上统提供树形结构与扁平化的展示方式，方便用户直观看到契约模型的相关信息。目前，MOM 支持 BJSC（携程 SOA 契约），Protobuf，XSD，Json Schema 等契约文件的导入。后续若要支持更多契约类型，也能很快地进行扩展。

4.1.2 在线编辑

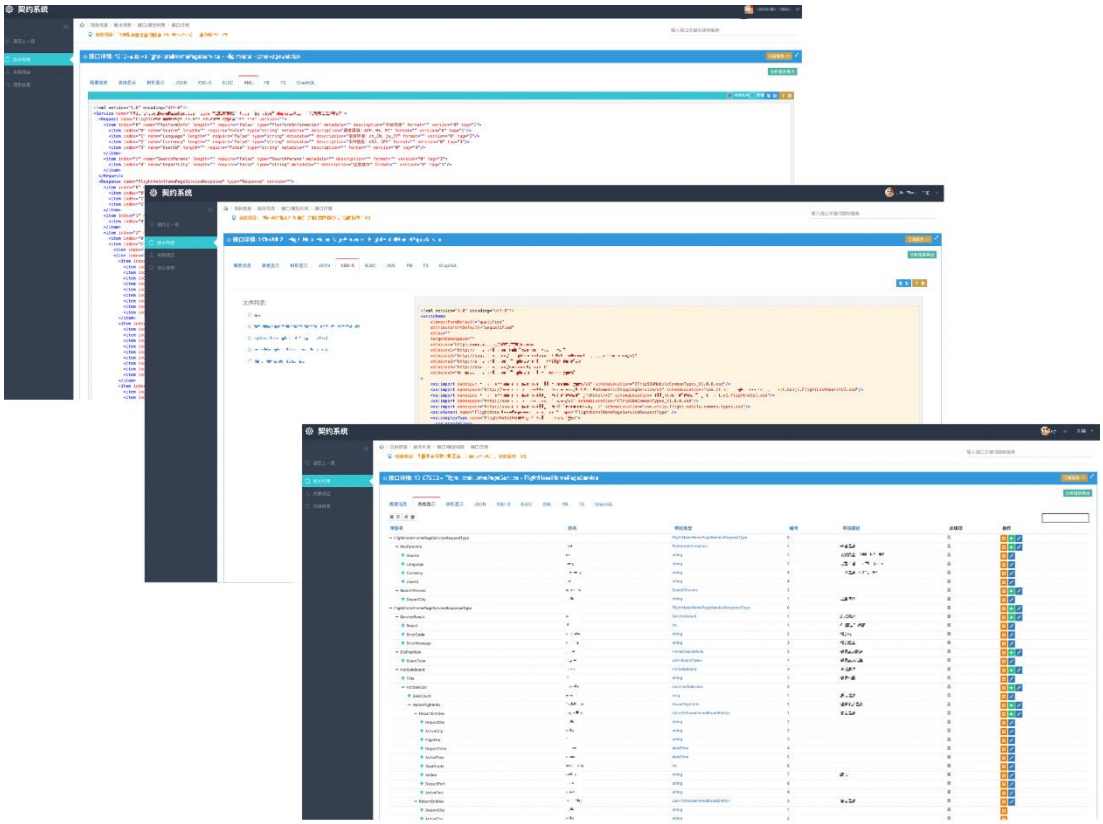
在线编辑是 MOM 的特色之一，对于历史契约，在用户完成契约导入后，MOM 会将用户的契约模型持久化下来。对于新接口新契约，MOM 也支持在界面上直接进行创建。后续，当接口需要扩展功能，修改契约时，MOM 支持在界面上直接修改并保存。界面上友好的契约修改操作流程，避免了人工编辑契约文件可能带来的问题。

值得一提的是，对契约的编辑修改有一项 Best Practice：对于已发布的契约，后续的修改只能是顺序增加字段，不能修改现有字段的名称和类型，也不能在任意位置插入新字段。因为，修改已有字段的名称和类型，必定会带来线上兼容性问题；要求顺序增加字段，主要是针对像 Protobuf 这类对顺序有强要求的契约格式。那字段越加越多，越来越冗余怎么办？可以考虑另起炉灶升级接口契约，或尝试 GraphQL。

4.1.3 契约编辑小结

契约导入和在线编辑是 MOM 的一大特色。这功能有别于其它工具，因为 Swagger、Knife4j、YAPI 都是面向 API 做管理，或与代码绑定不支持导入编辑，或只支持从 Postman 等工具导入接口。这与 MOM 专注契约的管理，有本质上的不同。契约可以看作是 API 的进一步抽象，MOM 导入和编辑的是契约，而不是 API。

当契约导入后，或完成编辑保存后，数据是以模型进行管理，在 MOM 上，模型到契约间的转换是双向的。举个例子，导入的是 XSD 格式契约，MOM 将其转成模型进行存储，最终在界面上，MOM 又可以将模型转换成各类契约格式，进行展示。



4.2 项目管理

MOM 中的契约模型，是以项目维度进行管理，项目分为应用与自定义项目。

应用是携程内部的定义，可以包含一组接口，通常也是生产发布的最小单位。通过这种直接与应用一一绑定的方式，可以方便用户快速查找自己应用的契约。

自定义项目与应用的差异，主要在于管理模型的差异上。自定义项目更多用于，存储共享模型、数据埋点模型等，它们可以脱离携程的生态环境而存在。值得一提的是，模型在不同的项目下是相互隔离的，原则上一个项目只能使用该项目下的模型，如果想要跨项目使用模型，必须先建立项目之间的绑定关系。

为了降低契约被项目成员之外的用户误操作，MOM 提供了权限管理，权限实际是建立在应用维度之上的。MOM 会定时同步应用权限（谁是 Owner、管理员），只有拥有这些角色的用户才允许编辑该项目下的契约模型。

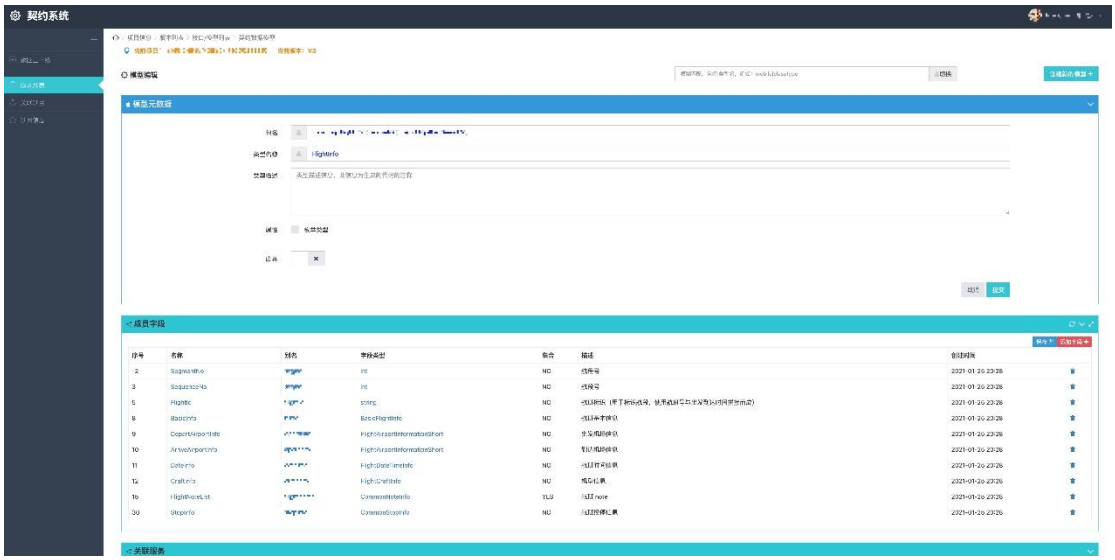
4.3 模型管理

模型管理，是 MOM 底层的核心。模型是最小的管理单元，契约的节点树是在此基础上进行嵌套搭建的。MOM 维护了基础数据类型，除此之外系统允许用户定义自定义模型和枚举类。

MOM 参考了 Java 的设计，项目以文件夹形式进行管理，模型所处的文件路径，决定了模型最终的生成位置。为了更好的进行模型描述，MOM 对模型抽象出了模型名称、字段、描述、

注解等相关属性，通过构建绑定关系使得共享策略得以生效。

契约系统的模型共享，主要分为项目间的共享与外部 jar 包内的模型共享。项目间的主要使用场景是，用户构建自定义项目维护共享模型，该模型可以被其他项目中的接口访问，来避免触发冲突规则。外部 jar 包引用，是通过同包下的同名模型进行相关的替换操作，系统解析 maven 仓库中 jar 包中的原始文件， 提取类型的节点树信息，替换项目中的模型。



4.4 多版本管理

多版本管理是契约管理的一个重要组成部分。

之前提到，由于需求的快速迭代，团队协作中，契约不可避免的产生多个可编辑的版本。MOM 为多个版本提供了相互隔离的环境，并且提供回滚，增量覆盖全量覆盖的相关功能。需要注意的是，版本在发布之后是不允许进行编辑操作的。但是多版本隔离的同时会带来冲突的合并问题，如何解决多版本冲突，提升契约的稳定性更是重中之重。为了解决稳定性问题系统先后提供两种解决方案，其一是版本比较，其二是版本冲突的自动合并。

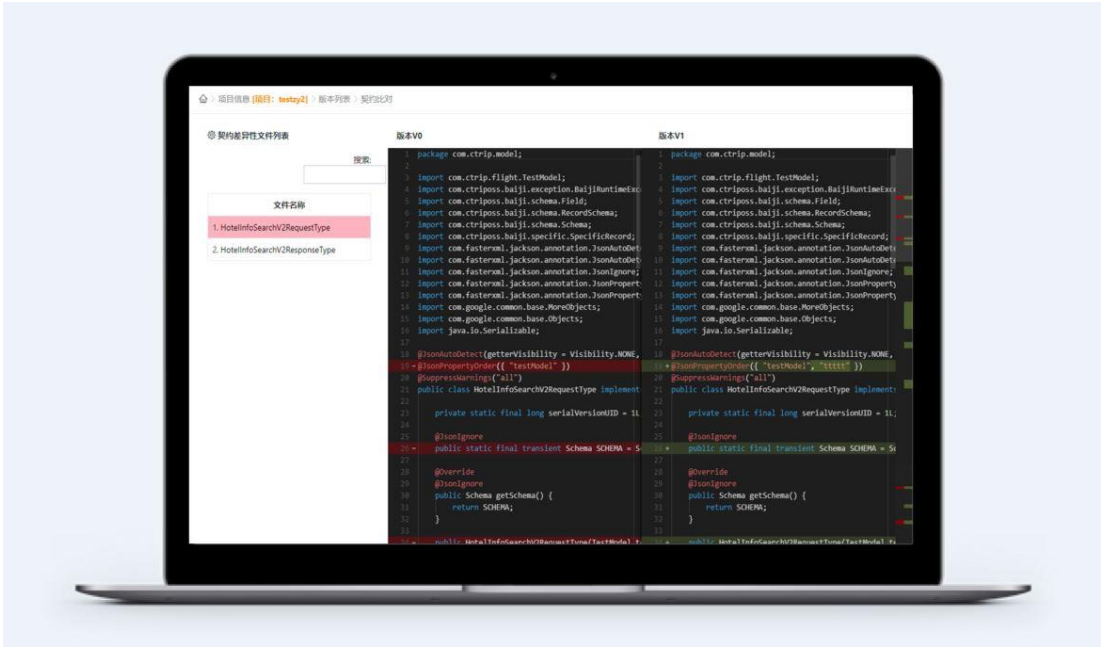
4.4.1 多版本比较

如何帮助用户更快的发现契约的变更，除了契约变更通知这一种手段外，系统自定义了相关的比较规则，降低用户比较的费力度。核心思路是减少比较内容，突出差异信息。

因为契约的生成格式可以是多种多样的，如果是对每一种生成格式都提供一套比对方案虽然能满足各方的需求，但是必然给系统的维护带来一定的挑战。前文提到系统使用结构化的模型管理，那么是否能提供一种可读性良好，同时能突出展示差异的解决方案？答案是有的。

结合目前线上在使用的契约，参考多方的契约处理格式，最终系统仿照 Protobuf 的文件定义出模型及接口的比较模板，针对用户关心的主要数据，提取出：字段名、字段别名、注释、模型等相关属性。使得用户比较时可以更直观的发现契约的改动。

当然系统也提供了生成文件的比较，如 Java 文件，方便特定的用户进行比较。如果有需要，后续也会对其他的格式逐步增加比对支持。

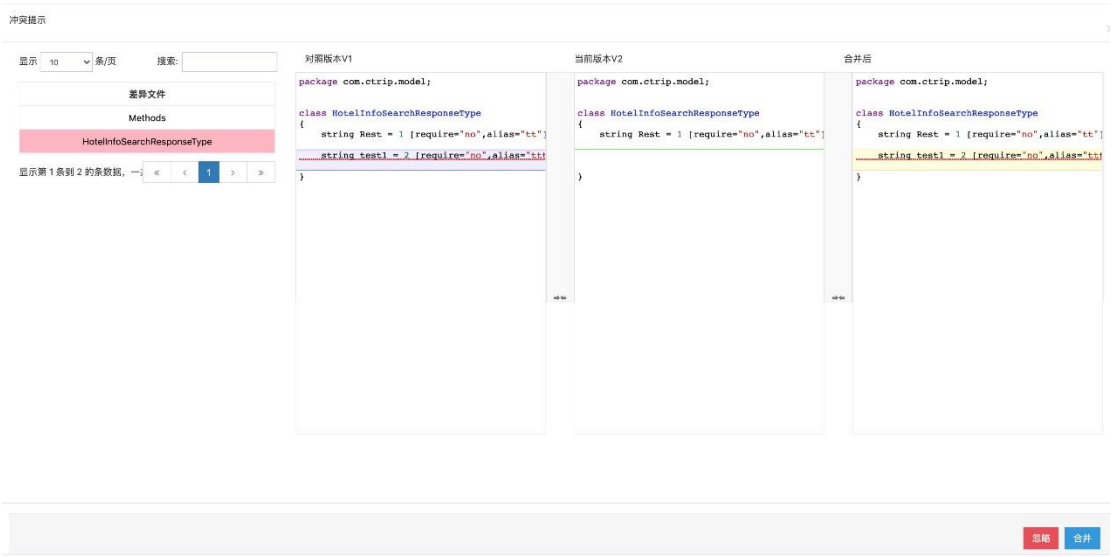


4.4.2 版本冲突自动合并

版本比较一定程度上帮助用户发现问题，但冲突合并才是解决问题的关键。能否支持冲突合并，成为考量系统易用性的重要标准。

契约的冲突合并，与代码的冲突合并类似，每当契约需要进行正式发布时，都应检查在其之前是否已有其它修改发布了。若有，则需解决可能引入的冲突。冲突的类型可能是新增类型，或者对某个模型的字段的修改，这些变更如果遗漏，显然是致命的。

针对该问题，系统参考 Git 的文件管理规则进行设计。契约合并的时机，发生在用户即将发布正式的契约版本时，系统自动拉取最新的发布版本与当前的发布版本进行比较，并做冲突展示，由用户选择解决冲突的方式，允许用户忽略冲突发布，当然相应的结果由用户承担。冲突的自动合并，降低了手工操作犯错的几率，极大的提升了团队协作效率。

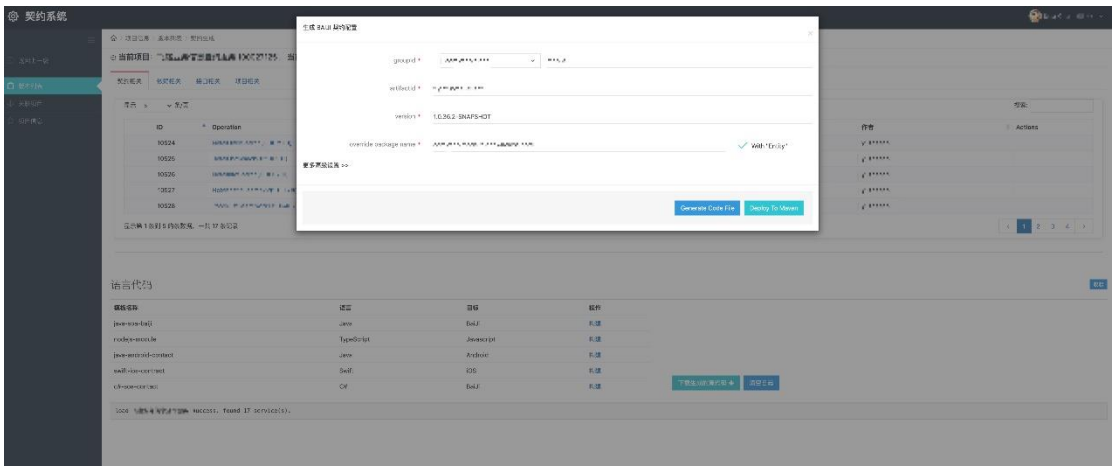


4.5 契约生成

契约生成是大多数用户比较关心的地方。能否满足各用户的使用场景，对系统而言也是一个重大挑战。

契约系统独立维护了一套描述性极强的模型结构，对于不同的生成契约类型，只需要根据不同契约的生成规则，定制不同的契约模板即可。随着系统的不断迭代，目前已经支持 Java、TS、GraphQL、C#、XSD、Protobuf 等文件的生成。

此外，光生成契约文件对于用户来说，易用性可能还不够。MOM 还支持将生成好的契约，直接发布到代码仓库。目前，MOM 已支持直接将契约部署到 maven 和 NPM（Node Package Manager）仓库，只需简单配置仓库地址即可。至此，契约的编辑到部署，可以全在 MOM 的操作界面上完成，这极大的提高了开发效率。



4.6 变更通知

MOM 不仅方便了契约维护方对契约进行管理， 也方便了接入方能够很好的查看接入。当契约发生变更时，将变动消息告知所有的关心此契约的人，是很有必要的一个功能。

MOM 的通知包含变更通知、发布通知。通知的渠道可以是各式各样的。通知的范围包含项目的管理员、owner、项目的关注人以及项目的调用方。契约系统会把用户的每一次编辑记录进行收集作为后续的通知内容，并把契约的改动可能影响的潜在用户列入通范围。此外，通知内容上提供 ReleaseNote 的编辑方式，用户编辑完成之后可以选择需要通知的对象及通知的方式进行通知操作。

目前，MOM 的消息通知渠道，主要包括 Trippal（携程内部通讯工具） 和 Email。系统预留了可扩展性，若需其它渠道的消息通知，可以快速进行开发或配置。

编辑操作							
模型名称	字段名称	字段短名	字段类型	Metadata	字段描述	是否必须	字段版本	范围描述
HotSaleList	SaleCount	scount	long	Int20		N	0	
	HomeFlightInfo	flightinfoh	HomeFlightInfo	Class		N	0	
	HomeHotelInfo	hotelinfoh	HomeHotelInfo	Class		N	0	

五、总结

总的来说，MOM 与其它工具最大的不同，是其更关心契约的管理。特别是在敏捷模式盛行的今天，业务变更快速而频繁，不可避免的经常对契约进行修改。如何管理并支持好这些修改，是 MOM 这个产品需要解决的核心问题。

目前，MOM 在携程内部正处于推广阶段，功能也在不断的增强与完善中。后续会考虑剥离携程相关生态的依赖， 在契约导入、契约生成、消息通知等功能上， 做到可插拔可扩展。最终开源到社区，帮助大家解决契约管理的问题。

最后：

- 如果你维护的项目多，接口契约经常发生变化
- 如果你的接口文档不清晰，甚至还是线下维护的方式
- 如果契约模型需要在各项目，各接口间共享

- 如果你需要生成各式类型的契约，提供到使用方法
- 如果你有多个团队共同维护一个项目， 契约修改经常冲突
- 如果你想把契约的变更，及时通知到各个关注方

那你可以参考 MOM 以模型为中心的契约管理方案，也可以持续关注 MOM 的后续消息。

减少 50%空间，携程机票 React Native Bundle 分析与优化

【作者简介】Sheila，携程资深前端开发工程师，关注前端性能优化；xqin，携程前端开发专家，CRN bundle 分析平台开发者。

一、前言

在业务迭代上线的过程中，往往会出现一些代码冗余，导致最终打包出来的 bundle size 不尽如人意。同时，业务包占用的尺寸过大，对应用的性能以及用户体验都会造成一定程度的影响。

本文将从 JavaScript 层面对 React Native 的业务包进行分析与优化，在这个过程中会运用 CRN （Ctrip React Native）bundle 分析平台等工具，在项目开发的中后期对业务包的尺寸进行裁剪优化。

二、现状

目前针对 React Native 的性能调优可以使用的工具少之又少，下面将介绍 React Native 中可以对 bundle 进行可视化的本地工具，以及我们为什么需要一个在线平台去构建 bundle 分析结果。

2.1 使用 bundle-analyzer 进行包模块内容的实时查看

在 react-native 中可以使用 react-native-bundle-visualizer 进行 bundle 的查看。它的原理是使用了 source-map-explorer 进行了 Metro bundler 的可视化输出。

Metro 是 React Native 官方的打包程序，会生成对应的 bundle 文件。在 react 中或者是使用 webpack 等工具打包出来的内容，都可以使用与 source-map-explorer 相关的一些打包分析工具进行可视化内容查看。

使用 bundle 分析工具，可以比较明显地辨识出哪些业务文件大小比较异常、需要进行优化，或者是引用了哪些 Javascript 库，导致 bundle 膨胀。

执行如下命令进行安装并启动：

cmd yarn add --dev react-native-bundle-visualizer && yarn run react-native-bundle-visualizer

如果没有跑出具体内容，则需要手动添加入口文件。例如，--entry-file ./index.android.js。执行结果会把 node_modules 和源文件中打包出来的代码尺寸都包含在内，可以清晰地看出哪些文件占用的空间比较大。

2.2 为什么要开发 CRN bundle 分析平台

Web 端针对 React 的分析优化工具很多，包括 webpack 官方也有提供打包分析，但这些针对 React Native 都不能使用。在上一小节中提到的工具，也只能在本地运行，每次改动后需要生成新的 treemap 进行图片之间的对比查看，不直观并且不方便对比。

React Native 开发的模块最后都会打包到 APP 中，如果能在平时的开发阶段，就注重保持 Bundle SIZE 的简洁，注意观察业务包 SIZE 的限制大小，那么不需要后期进行排查裁剪。

CRN bundle 分析平台不需要使用者手动运行，只需要使用者选择自己的业务包名称，即可进行在线的分析，并且可生成过去 7 天 bundle size 色阶图，可以让使用者对过去一段时间内的开发打包结果进行及时排查，也就是说可以对包内尺寸的膨胀进行告警。

现有的 React Native Bundle 分析工具，除了只能本地进行运行以外，还存在的缺点就是它是针对 React Native 官方的打包工具的运行结果进行的分析，对于 Ctrip React Native 或者是其他基于 React Native 优化的跨平台开发框架，是会有一定缺陷的，例如无法找到正确的入口文件、无法找到对应的依赖关系等等。

针对 React Native 进行 bundle 分析的在线平台，相较于现有的工具，具有以下优点：

- 便于 React Native 性能调优
- 便于减少 APP SIZE，提升应用整体性能
- 在线分析展示
- 包内 SIZE 膨胀告警
- Ctrip React Native Bundler 打包结果定制化分析

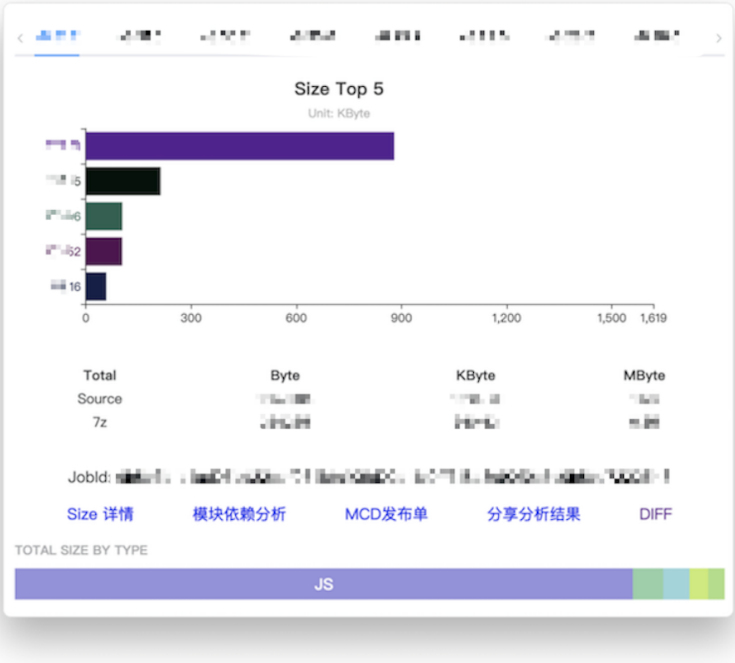
定制化的 RN bundle 分析平台，可以随时拉取当前业务包的历史打包结果，并且进行在线分析与告警，还可以让使用者得到一个关于本次优化内容的文件差异对比内容，在分析优化后，可以快速看到优化效果，简单高效。

三、CRN bundle 分析平台

3.1 功能介绍

CRN bundle 分析平台，可以对 React Native 打包后的内容进行在线二次分析。它具有项目内部模块依赖分析、文件尺寸树状结构矩形图等图表展示功能。

3.1.1 bundle 概要



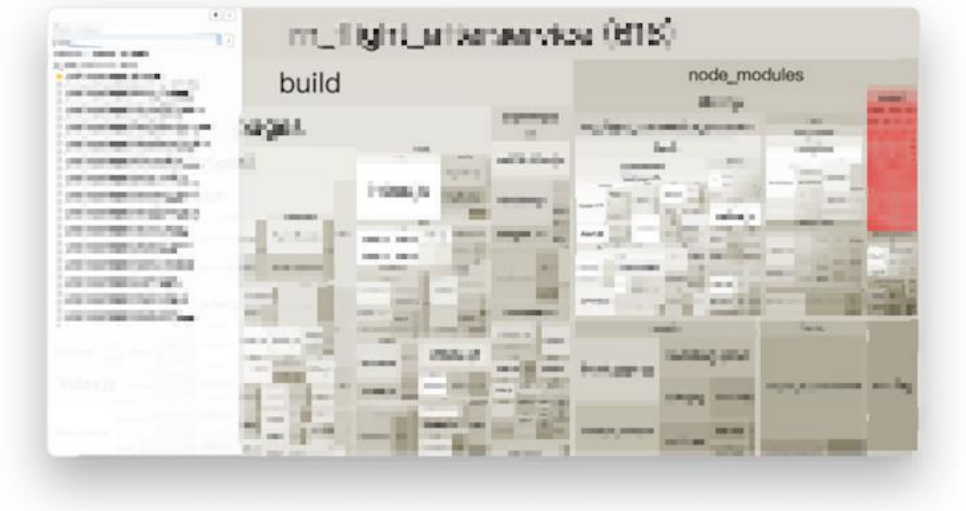
如上所示中，展示了业务模块的整体大小以及压缩后的尺寸，并且进行了图形化的占比展示。

在条形图中，从打包的模块内容角度，显示了当前业务包中占比最大的五项内容，包括 build 后生成的内容，以及 node_modules 中的模块大小占比。

底部的占比图中，从文件类型的角度，显示了当前业务包中的 JavaScript、Font、Image 等文件类型的大小占比。

在 bundle 概要的页面，显示了当前业务包的源代码大小以及打包后的压缩大小。

3.1.2 SIZE 详情

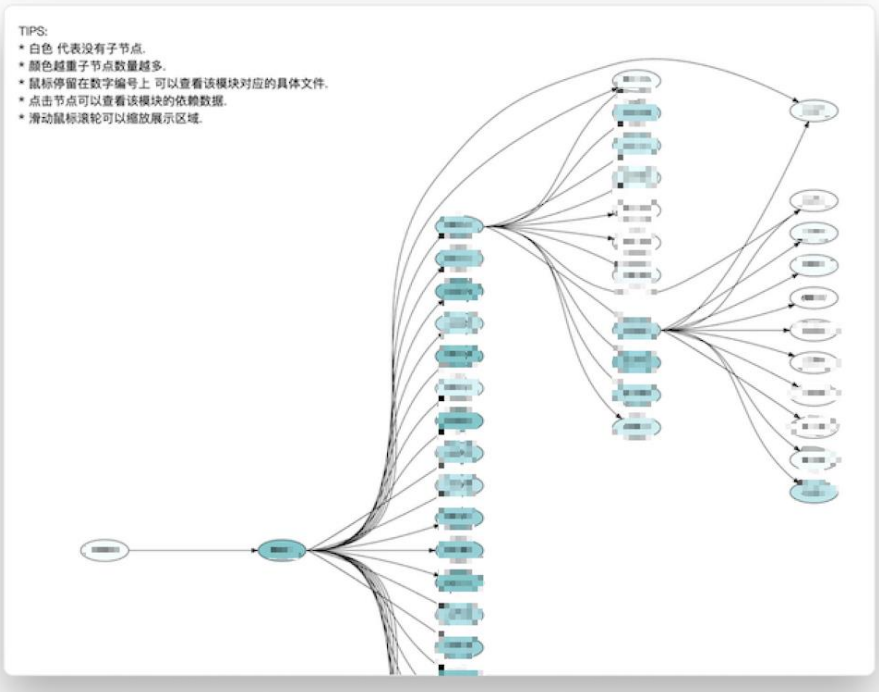


bundle Size 详情页面，使用树形结构图，直观地展示了当前业务包中各个模块的尺寸大小以及占比。

可对相应的模块文件进行搜索查看，同时会高亮展示在树形结构图区域，以便排查和优化打包结果。

使用详情页面，可以对优化前后的结果进行图形化的尺寸对比。

3.1.3 模块依赖分析



模块依赖分析页面，会根据模块的依赖关系生成 dependency graph，便于排查模块之间的深层依赖。

3.1.4 文件差异比较

对于任意两个发布单，可以根据 JobId 进行包内各个文件的大小 diff 对比，并且会链接到 gitlab 对应的 changes 内容，可以看到代码优化部分的相关 commmits 。

使用该功能，可以直观地进行某次裁剪前后的尺寸大小对比，快速验证优化效果。

3.1.5 CRN 模块可用包大小统计

对于每个业务包可以给出一个可用包尺寸大小，并且根据每日打包结果，生成对应的过去时间段中的打包尺寸大小色阶图，使用色阶可以预警过去的时间段中是否出现超限的业务包打包结果，及时对打包内容进行排查。

3.2 实现原理

CRN bundle 分析平台主要依赖三个部分进行实现，分别是处理 JOB 数据、使用后台 API 分析打包后的业务包文件，最后在前端进行各种图表化的展示。

打开平台页面后，使用者选择要分析的业务包名称，后台 API 根据参数调用相关接口，得到要分析的业务包的下载地址和对应的内容映射文件，并且将数据添加到队列中，等待后续分析处理。

循环调用后台 API 去获取要分析的 JOB 进行数据处理。在这个过程中，调用 Nodejs 对当前选择的业务包进行基础分析，并与 map 文件相结合，得到关键依赖数据与代码详情内容，生成最基础也是最重要的数据包，这个数据包使用 JobId 作为文件名称，得到一个 JSON 格式的数据内容，后续的处理都在这个 JSON 文件的基础上进行。

获得一个基础数据文件后，使用前端把数据处理为需要的格式进行展示。在前端交互逻辑上使用了 Vue.js 与 element-ui 进行基础页面构建，使用 d3.js 进行了数据的可视化展示，在这里用到了树形矩阵图、色阶图、条形图、依赖关系图等等图表进行内容展示。在 DIFF 页面中，同时分析了两个指定的 JobId 下的业务包内容，并且按照差异内容进行了详细的 SIZE 增减对比。

四、分析包模块

在进行包裁剪之前，我们需要先分析业务包内各模块的占比大小，以便对具体的模块进行修改。工欲善其事，必先利其器。在有了分析工具后，可以对业务包模块进行详细的分析与优化。在这里，使用本地安装的 bundle 分析工具进行普适的分析。

在这个截图中，可以很清楚地看到，除了公共引用库以外的内容中，有几个比较明显的膨胀模块，分别是 lodash、moment，以及一个工具类库下的业务逻辑文件。接下来我们针对这几处明显的问题进行优化。

五、解决方案

5.1 常用类库优化方案

Momentjs 和 Lodashjs 是前端常用类库，但这两个都有很明显的问题，所占据的文件空间略大，而且大多数时候我们只需要用到其中小部分的功能。在如下类库替换过程中用到的方法，可以运用到所有常用类库的优化使用中。

5.1.1 选择满足需求的最小类库

moment 是一个常用的 JavaScript 日期处理类库，它支持多语言的日期格式。moment 的核心代码只有 52kb，但是包含了全世界语言的本地化文件，也就是说当你使用其中的功能时，也包含了很多你用不到的特性。

对应的解决方案是你可以通过 npm 安装 moment-mini，该库非官方维护，但暴露了官方的 moment-min.js 作为 npm 模块开源使用。或者你可以直接使用一些更为简洁的 JavaScript 日期格式化类库。

作为 momentjs 的替代方案，可以使用 luxon、date-fns、dayjs，或者直接使用 JavaScript 的原生 API 来做日期国际化（JavaScript Internationalization API）。如果不需要引入日期国际化，dayjs 核心代码只有 7.1k，可以作为 momentjs 的替代。

NAME	SIZE ORIGINAL/GZIPPED	TREE-SHAKING	POPULARITY (STARS)	METHODS RICHNESS	PATTERN	TIMEZONE SUPPORT	LOCALE
Moment.js	329K/69.6K	No	43.4k	High	OO	Good (moment-timezone)	123
Luxon	59.9K/17.2K	No	9k	High	OO	Good (Intl)	-
date-fns	78.4k/13.4k without tree-shaking	Yes	21.3k	High	Functional	Good (date-fns-tz)	64
dayjs	6.5k/2.6k without plugins	No	25.8k	High	OO	Not yet	130

5.1.2 不必要时避免引入整个类库

lodash 是一个实用性非常高的 JavaScript 工具库，可以对 array、object、string 等值进行操作和检测等等，还具有一些非常实用的函数。但 lodash 类库所占用的空间达到了 71K，而且也存在很多你用不上的方法。实际上，我们在使用中或许只会用到非常少的几个函数。

官方虽然也提供了 lodash-cli 这样的工具，让使用方可以针对具体的某些函数进行打包，但官方是不推荐这种用法的，并且在新的版本中也取消了这样的部分模块打包方式。官方推荐的方式是，在引用时指定对应的函数，这样最终打包时只会打包对应的函数。

如下所示，如果直接引用 lodash，大小时 71K。

javascript import get from 'lodash' // 71K (gzipped: 24.7K)

如果引用对应的函数，那么所需要的空间会大大减少。

javascript import get from 'lodash/get' // 8.2K (gzipped: 2.5K)

- 通过 Babel 插件配置

babel-plugin-transform-imports 这个插件可以把全局 import 替换为具体模块的单独引入。

配置如下：

```
# .babelrc
"plugins": [
  ["transform-imports", {
    "lodash": {
      "transform": "lodash/${member}",
      "preventFullImport": true
    }
  }]
]
```

具有如下效果：

```
import { map, some } from 'lodash'
// 被替换为
import map from 'lodash/map'
import some from 'lodash/some'
```

注意这个选项 preventFullImport 在引入整个库的时候会让插件抛出异常。

- 通过 ESLint 规则配置

在 ESLint 中配置 no-restricted-imports 规则，也可以在全局引入时抛出异常。

```
# .eslintrc
"no-restricted-imports": [
  "error",
  {
    "paths": [
      "lodash"
    ]
  }
]
```

在如下引入方式时会抛出异常：

```
import { map } from 'lodash'
```

但按照这样编写则不会报错：

```
import map from 'lodash/map'
```

具体使用方法可查看该规则说明，可以对引入模块的代码风格进行控制。

5.1.3 删除可替代的类库，重写方法实现

使用功能齐全的工具性函数是非常诱人的，可以快速交付，或者是能够对未来的功能进行快速实现。但是过度实现增加了目前不需要的代码，其造成的复杂性，会对 bundle 的大小产生一定的影响。

在我们的项目中使用到的是 Lodash，官方虽然指出只引入对应模块就会便捷很多。但 Lodash 依然有很多存在依赖关系的内部函数需要一起打包进去。如果你仅仅是使用到这个实用库类的部分工具函数，那么可以用一些体积更小的工具包进行优化，或者直接使用对应的原生实现方式进行替换。

如果我们的对于项目代码中的依赖关系，只引入了一小部分相关内容，并且可以在合理的时间内对其进行重写。那么我们应该重写这部分代码，以达到优化冗余代码的目的。把项目中涉及到的工具库类函数直接用原生代码替换，不失为一个很好的解决方案。

以下是原生 JavaScript 实现 Lodash 的 debounce 函数：

```
function debounce(func, wait, immediate) {
  var timeout;
  return function() {
    var context = this, args = arguments;
    clearTimeout(timeout);
    timeout = setTimeout(function() {
      timeout = null;
      if (!immediate) func.apply(context, args);
    }, wait);
    if (immediate && !timeout) func.apply(context, args);
  };
}
// Avoid costly calculations while the window size is in flux.
jQuery(window).on('resize', debounce(calculateLayout, 150));
```

5.2 替换 package 中不必要的模块

这里的替换掉不必要的组件/模块，更多地是从业务逻辑方面来说的。如果已经引用的库里面存在某些业务逻辑功能，或者有公用的组件已经实现了对应的功能，那么我们应该进行替换，删除掉多余的业务内代码。

同样的，检查下 package.json 文件中也许会存在未使用的包，或者是重复功能。在开发阶段，也许会存在引用了某些库类，随着业务变化，又在具体逻辑中删除了引用，但未清除彻底，导致 package 中还有残余，却给 bundle size 带来了一定的负担。也或者是同上面 lodash 和 moment 库，可以通过用一些更简单的库，或者自己实现几个常用功能来进行整

个模块的替换。

在这里对于我们的业务包来说，包内存在以下这些问题：

1）把业务内不必要的组件替换为公用组件
2）删除不必要的 node_modules 模块，或者用其他模块替代

在这个层面上来说，是细粒度上面的代码冗余的清理，包括下线实验代码的处理工作等等。

5.3 代码拆分

除了页面展示需要的代码模块以外，不应该加载多余的代码逻辑。对于不同的业务固然有不同的方法，但核心的两个主要方法是：

- 基于路由的代码拆分
- 基于功能/组件的代码拆分

1) 使用 Ctrip React Native 的 lazyRequire 方案

React Native 官方提供的 require 目前并不支持动态加载，所以 CRN 框架提供了 lazyRequire 来支持懒加载方案。App 组件将在跳转页面的时候再加载该模块。

```
let page = lazyRequire('./src/Page.js');
const pages = [
    {
        component: page,
        name:'page',
        isInitialPage: true
    }
];
```

2) 使用 require 延迟加载

我们可以通过内联引用的方式，延迟模块或文件的加载，直到实际需要该文件。但如上所说，目前 React Native 并不支持动态加载，所以需要 state 属性去控制是否引入对应模块。

```
import React, { Component } from 'react';
import { TouchableOpacity, View, Text } from 'react-native';

let VeryExpensive = null;

export default class Optimized extends Component {
  state = { needsExpensive: false };
```

```
  didPress = () => {
    if (VeryExpensive == null) {
      VeryExpensive = require('./VeryExpensive').default;
    }

    this.setState(() => ({
      needsExpensive: true,
    }));
  };

  render() {
    return (
      <View style={{ marginTop: 20 }}>
        <TouchableOpacity onPress={this.didPress}>
          <Text>Load</Text>
        </TouchableOpacity>
        {this.state.needsExpensive ? <VeryExpensive /> : null}
      </View>
    );
  }
}
```

5.4 静态资源优化、静态代码检查

5.4.1 图片资源优化

在 bundle 分析截图中，到目前为止虽然已经解决了一些库类引用不合理的问题，但还存在逻辑文件过长的问题，分析后发现是里面包含了超大的 base64 图片。

base64 更适合出现在一些重复使用的背景图片，或者尺寸极小的 ICON 的情形，而一些较大的图片则适合使用 PNG 或者 JPEG 。

PNG 是无损的，JPEG 是有损的。如果不需要背景透明，那么把 PNG 转换为 JPEG 会更节省空间。

1）剪裁图片大小：设计师给出的图片一般会比较大，而实际应用中不需要这么大的图片，可以适当地进行图片大小的裁剪。
2）压缩图片质量：对图片进行无损压缩后，再进行 base64 使用。

经过以上两个步骤以后，base64 的图片字节数会明显减少很多。如果字节数还是很大，那么应该考虑是否不适合使用 base64 进行展示。

5.4.2 ESLint 检测 React Native 的 CSS 冗余

在 React Native 的 ESLint 规则中配置 react-native/no-unused-styles ，会检测在 React 组件中存在的未使用 CSS 。

在长期对组件进行开发的过程中，随着 UX/UI 的更改，会存在一些冗余的样式散落在文件中。这样的一个配置可以很好地显示出冗余的部分。

```
# .eslintrc
  "rules": {
        "react-native/no-unused-styles": 2
  }
```

但它存在很明显的缺陷，就是在 css-in-js 的写法中可以检测到当前文件中的 css 对象引用，也就是对于 styles.xxx 可以很好地检测到。

但是对于以下几种，目前这个规则都无法检测：

1）Import 进来的 CSS 文件无法识别；
2）使用 StyleSheet.flatten 等方法操作过的 style 无法识别；

```
const styles = StyleSheet.flatten([style1, style2])
// 无法检测到该对象中存在的样式
```

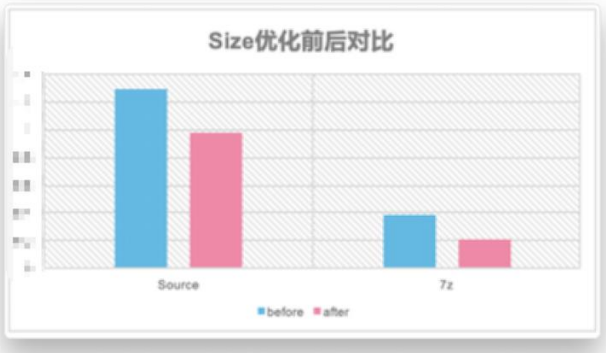
3）CSS 对象初始化与使用名称不同时，无法识别。

```
const A = StyleSheet.create({
 header: {
      color: "#61dafb",
      fontSize: 30,
      marginBottom: 36
  }
})

const styles = IS_BOOL ? A : B
// 无法检测到 A/B 中存在的样式
```

这些问题官方目前都未修复。这个规则只能检测最简单形式的 CSS 内容中的冗余，如果希望一直能使用该规则，则在代码规范上需要保持简洁的 CSS 引用形式。

六、结束语

在经过一系列的分析与细节的优化操作过后，成果是压缩后的 Bundle Size 减少了约 50% 的空间占比。



进行 bundle 分析后，可以明显地找出尺寸异常的文件或者模块，进行对应的优化，从大的层面上进行分析与尺寸优化：

1）根据 bundle 分析裁剪具体模块，分析模块引用代价；
2）替换不合理的库类引用

粗粒度的优化后，剩下的有关逻辑的代码优化，就跟平时的编写有关。从小的层面上进行优化需要：

1）从逻辑上分析不必要存在的库类/模块引用；
2）编写逻辑代码时，需要更加注重保持代码行数的简洁；
3）提取常用功能为公用组件进行使用；
4）静态资源使用优化

在代码编写阶段保持最佳实践是最好的，但在中后期我们也能通过一些分析工具进行代码包的裁剪。项目结构与模块的依赖关系更加复杂的时候，运用以上方案进行 bundle 裁剪会更加有效。

保持 Bundle 尺寸精简是一个长期的任务，就像保持代码简洁一样，甚至可以说这项任务与项目的生命周期是紧密相连的。有了良好的工具可以更加方便开发者进行分析，实时查看目前的代码简洁程度。

携程 APP Native/RN 内嵌 Flutter UI 混合开发实践和探索

【作者简介】Deway，携程资深工程师，iOS 客户端开发，热衷于大前端和动态化技术；Frank，携程高级工程师，关注移动端热门技术，安卓客户端开发。

前言

随着各种多端技术的蓬勃发展，如今的移动端和前端早已不再拘泥于自身的边界，而是不断延伸、扩展和融合，逐步向着真正的大前端技术迈进。跨端技术也从早期的 Cordova/PhoneGap、纯 H5 页面发展到如今的 ReactNative（以下简称 RN）、Weex、小程序、Flutter 群雄并存的局面。各种技术栈各有优劣和特点，技术选型需视团队自身情况而定，没有绝对好坏之分。然而在实际开发中，并不是只选用一种技术栈，那么研究多种技术栈融合和嵌套使用的就有了迫切的必要性。

本文我们从实际业务场景出发，初步实践了在 RN 里面嵌套 flutter view、在 native 里面嵌套 flutter view，探索其可行性，并回顾这个过程中遇到的一些问题和解决方案。

一、背景

1.1 现状

随着时间的推移，携程 app 中酒店列表和详情两大页面已经全部转为 flutter 技术栈，初期的使用场景也比较单一，只在主流程使用。然而业务不断迭代之后，flutter 页面在其他流程使用的频次也越来越高，比如列表页面，作为酒店一线 SKU 产品展示的主页面，复用的需求非常旺盛和迫切。那么此时需要思考更多的通用性和可移植性，以适用于在不同的场景不同的技术栈页面嵌入使用。

1.2 两大场景

- 场景一：上左图为携程大搜页面的酒店列表。在本次技术改造之前，大搜页面的酒店列表和酒店主流程的列表大相径庭，差异不光是在 UI 展示方面，酒店频道列表的信息和优惠更加完整，价格体系也更统一。大搜的技术栈是 RN，而酒店列表技术栈是 flutter，如果想要统一无非两种途径：1）查漏补缺追平 RN 侧业务；2）将 flutter 酒店列表打包嵌入 RN 页面。

- 场景二：上右图为查询页钟点房标签下的钟点房列表，查询页目前还是 native 技术栈，那么此时也必须考虑将 flutter 列表页嵌入 native 页面。

对于不同技术栈的业务场景，不断为多侧业务同步补齐功能，维护成本是相当巨大的。对于酒店列表业务来说，唯一可选的路径就是在大搜和酒店主频道等业务场景中共用一个列表，甩开历史包袱，实现真正意义上的业务对齐。所以，基于以上两个场景，我们初步探索了 flutter 页面在多种复杂结构的嵌套使用，即 RN 中嵌套 flutter、原生 ListView 中嵌套 flutter，并将解决方案记录在本文中，为之后可能遇到的多业务场景提供一个思路。

二、RN 中使用 Flutter

2.1 可行方案的探究

在接到这个嵌套需求的时候，考虑到成本最低的方式是直接在大搜页面层上盖列表，即在切换到酒店 tab 的时候将 flutter view 盖在上层。实际上在思考利弊之后，放弃了这个方案。

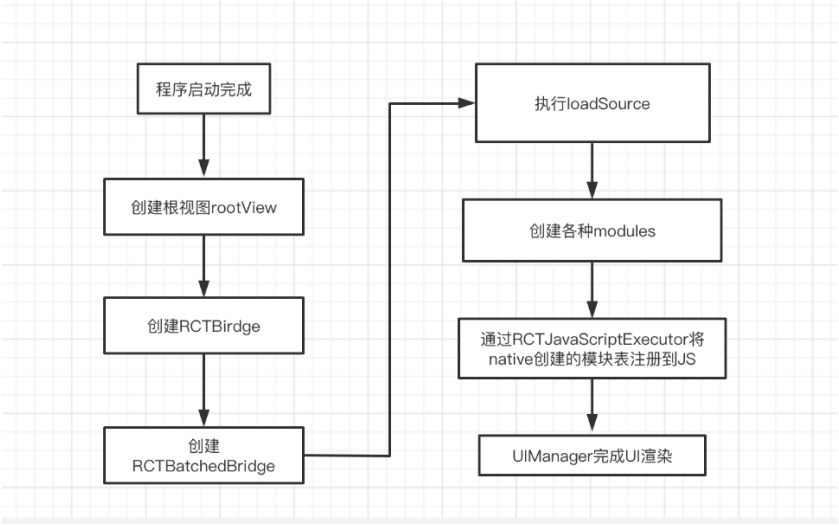
有如下几个弊端:

- RN 无法单独控制 flutter view 层的展示，需要通过层层事件通知，复杂且繁琐
- RN 需要计算出上盖 offset 的偏移值，在不同屏幕尺寸存在偏差
- 在不同 tab 切换的时候，flutter 控制器生命周期难以及时被同步

基于上述的几个问题,那么考虑的方向就偏向于直接把 flutter view 包装成 RN 的 Component 使用。

2.2 Native Components 的原理

我们先简单回顾下 RN 的启动流程（以 iOS 为例）。



RN 启动流程

- 程序启动完成的时候创建了根视图 RCTRootView，负责展示所见内容的根容器
- 创建管理 native 和 js 的交互的桥接对象 RCTBridge
- 创建 RCTBatchedBridge 批量桥对象，这个类是负责交互通信的核心
- 加载所有自定义的 native modules。这些 modules 最终会被转为 RCTModuleData 类型，包含方法列表、队列等信息，并缓存到全局的模块配置信息表中
- 通过 jsExecutor 将 native 创建的模块表注册到 js 端
- 开始异步加载 js 代码，执行完成后通知到 RCTUIManager 去调用对应的 native 组件进行渲染

这里省去了一些非关键步骤，可以看到 RN 本身是支持调用 native 原生组件的，调用 native UI components 这一步比较关键的是 RCT_EXPORT_MODULE。这是一个宏定义，重写了 load 方法，在其中调用 RCTRegisterModule 方法。再看看 RCTRegisterModule 的实现，其实就是将 moduleClass 注册到一个全局容器。

```
/// iOS
#define RCT_EXPORT_MODULE(js_name)                \
```

```
  RCT_EXTERN void RCTRegisterModule(Class); \
  +(NSString *)moduleName                   \
  {                                         \
    return @ #js_name;                      \
  }                                         \
  +(void)load                               \
  {                                         \
    RCTRegisterModule(self);                \
  }


/// iOS
void RCTRegisterModule(Class moduleClass)
{
  static dispatch_once_t onceToken;
  dispatch_once(&onceToken, ^{
    RCTModuleClasses = [NSMutableArray new];
    RCTModuleClassesSyncQueue =
        dispatch_queue_create("com.facebook.react.ModuleClassesSyncQueue",
DISPATCH_QUEUE_CONCURRENT);
  });

  RCTAssert(
      [moduleClass conformsToProtocol:@protocol(RCTBridgeModule)],
      @"%@ does not conform to the RCTBridgeModule protocol",
      moduleClass);

  // Register module
  dispatch_barrier_async(RCTModuleClassesSyncQueue, ^{
    [RCTModuleClasses addObject:moduleClass];
  });
}
```

moduleClass 注册完之后什么时候使用呢？接下来就到 RCTCxxBridge 的 start 方法，将所有注册的组件放入 moduleClasses，并将继承于 RCTViewManager 的 module 单独拿出来再处理。

```
/// iOS
(void)[self _initializeModules:RCTGetModuleClasses() withDispatchGroup:prepareBridge
lazilyDiscovered:NO];


/// iOS
  _componentDataByName = [NSMutableDictionary new];
```

```
  for (Class moduleClass in _bridge.moduleClasses) {
    if ([moduleClass isSubclassOfClass:[RCTViewManager class]]) {
      RCTComponentData *componentData = [[RCTComponentData alloc]
initWithManagerClass:moduleClass bridge:_bridge];
      _componentDataByName[componentData.name] = componentData;
    }
  }
```

从头到尾来理解下，在 main 函数开始执行之前，将申明为 RCT_EXPORT_MODULE 的组件注册到全局容器中，并在 bridge 中生成 RCTViewManager 对应的 RCTComponentData 对象，并配置 moduleConfig 的模块信息表（上述步骤 4 中完成）。然后在 RCTUIManager 中建立和 js 布局层的对应关系，最后在 js 层进行计算、排版之后通过 UIManager.js 通知到 native 层的 RCTUIManager 进行渲染绘制。这就是一个 RN 使用 Native 原生组件的原理和过程，由此可以见 RN 对于 modules 层的设计具备高度可扩展性和伸缩性。

2.3 前置条件

2.3.1 组件生命周期

携程主站是一个包含 native、RN、H5、flutter 技术栈的混合 app，基础框架由 native 代码实现，因此 flutter 业务需要依赖于兼容 native、flutter 的技术框架，业内比较成熟的解决方案是 FlutterBoost。

FlutterBoost 的理念是将 flutter 像 Webview 那样来使用，通过 native 容器来管理 flutter 页面。类似的，携程 app 中 RN 技术栈也是一个 RN-native 混合方案 CRN，用 native 容器封装了 RN 页面。这样的方案可以实现一个 native 容器中同时嵌套 native、RN、flutter 组件，并由 native 容器管理生命周期。

那 flutter-RN 组件嵌套时，如何实现不同组件生命周期相关联？由于目前列表 flutter view 是依附列表控制器存在的，在创建 RN 对应的列表控制器 view 时，将 flutter view 的控制器挂载到父控制器，这样实现了 flutter view 依赖 RN 的生命周期，伪代码如下。

```
// iOS
 UIViewController *rootVC = (UIViewController *)[self currentVisibleViewController];
 [rootVC addChildViewController:flutterViewControler];
```

从官方手册可知，

This method creates a parent-child relationship between the current view controller and the object in the childController parameter. This relationship is necessary when embedding the child view controller's view into the current view controller's content. If the new child view controller is already the child of a container view controller, it is removed from that container before being added.

Android 的实现类似，从 xml 文件可以看出，同样是将 flutter view 挂载到 RN 父 ViewGroup 中，即 RNLinearLayout。

```
<!-- Android -->
<?xml version="1.0" encoding="utf-8"?>
<com.android.list.RNLinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <com.android.list.RNFrameLayout
        android:id="@+id/flutter_view"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
</com.android.list.RNLinearLayout>
```

小结一下，flutter 的生命周期可以依赖于嵌入的父组件，如下表所示。

父组件	子组件	生命周期依赖对象
RN	Flutter	RN
Native	Flutter	Native

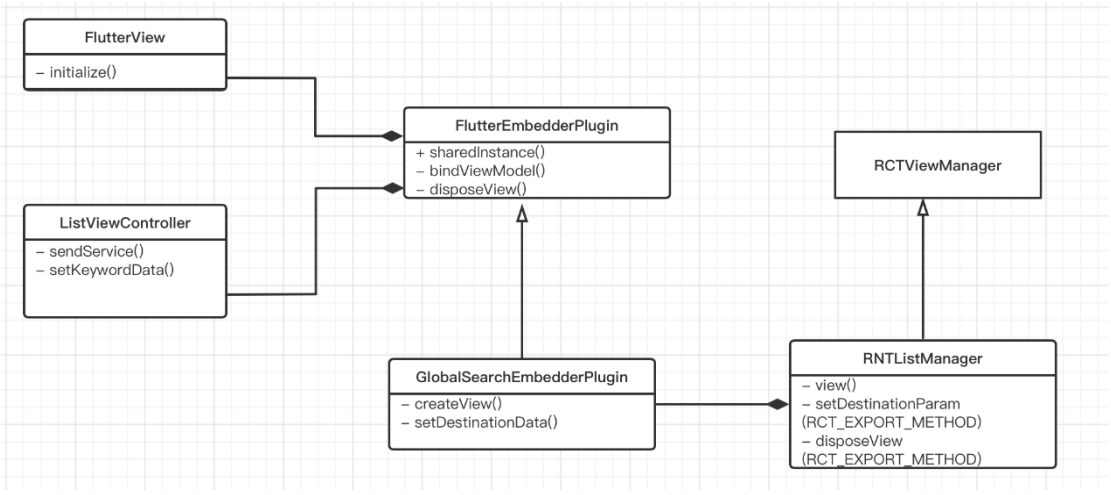
2.3.2 flutter 页面启动方式

由 FlutterBoost 官方文档可知，flutter 页面以路由的方式启动，携程 app 中实现（以 Android 为例）如下代码段所示。启动时需配置一个 flutter url，包含页面类型、业务参数、UI 相关参数等，用一个 fragment 来管理 view，并在 fragment 的生命周期不同阶段完成 flutter 初始化、绘制、销毁等操作，伪代码如下。

```
/// Android
val flutterFragment = new FlutterFragment
            .CachedEngineFragmentBuilder(FlutterFragment.class, FlutterBoost.ENGINE_ID)
            .flutterURL(mFlutterURL)
            .build();
fragmentTransaction.add(R.id.fragment_stub,
flutterFragment).commitNowAllowingStateLoss();
```

2.4 开始

从前述原理来看，native 的 UI 组件直接遵守 RCTViewManager 的模式提供 view 方法就可以被 RN 调用。那么是不是 flutter view 的嵌入也可以遵从这套范式呢？顺着这个思路设计结构图如下：



rnFlutter 混合结构图

- RNTListManager 继承于 RCTViewManager，负责提供 view 给 RN，设置和更新视图的属性，并接受 RN 传递过来的事件，比如目的地关键字、入离日期以及业务埋点数据。
- FlutterEmbedderPlugin，统一处理 flutter view 的创建、回收、销毁以及与之相关的事件回调。之后需要增加业务场景时，那么创建其子类处理具体业务就行。
- GlobalSearchEmbedderPlugin 作为 FlutterEmbedderPlugin 子类，负责处理具体场景下的业务，提供 RNTListManager 需要的视图。

js 层包装类如下：

```
/// RN 伪代码
class RNFlutterView extends PureComponent {
    componentDidMount() {
        const {keyword} = this.props;
            let data = { keyword: keyword};
            UIManager.dispatchViewManagerCommand(
                findNodeHandle(this),
                UIManager.FlutterListView.Commands.callFlutterEmbedderMethod,
                [data]
            );
    }

    /*..... 省略了无关的业务代码 */

    render() {
        return <FlutterListView style={{width: screenWidth}}
      />
     }

     componentWillUnmount() {
       /// 页面消失的时候回调 flutterEmbedder
```

```
            UIManager.dispatchViewManagerCommand(
                    findNodeHandle(this),
                    UIManager.FlutterListView.Commands.disposeView,
                    null
                );
        }
}
const FlutterListView = requireNativeComponent('FlutterListView', RNFlutterView);

export default RNFlutterView;
```

下面罗列主体结构的部分代码。iOS 的实现部分伪代码如下:

```
/// iOS
/// RNTListManager
RCT_EXPORT_MODULE(FlutterListView)

- (UIView *)view {
    if (self.viewPlugin == nil) {
        self.viewPlugin = [[GlobalSearchEmbedderPlugin alloc] init];
        [self.viewPlugin createView];
    }
    return self.viewPlugin.exportView;
}

RCT_EXPORT_METHOD(setDestinationParam:(nonnull NSNumber *)reactTag
param:(NSDictionary *)paramDict) {
    [self.bridge.uiManager addUIBlock:^(__unused RCTUIManager *uiManager,
NSDictionary<NSNumber *, UIView *> *viewRegistry) {
        UIView *view = (UIView*)viewRegistry[reactTag];
        if (view) {
            [self.viewPlugin setDestinationData:paramDict];
        }
    }];
}

RCT_EXPORT_METHOD(disposeView:(nonnull NSNumber *)reactTag) {
    [self.bridge.uiManager addUIBlock:^(__unused RCTUIManager *uiManager,
NSDictionary<NSNumber *, UIView *> *viewRegistry) {
        UIView *view = (UIView*)viewRegistry[reactTag];
        if (view) {
            [self.viewPlugin disposeView];
        }
    }];
```

```
}


/// iOS
@interface GlobalSearchEmbedderPlugin : FlutterEmbedderPlugin

- (void)createView;

- (void)setDestinationData:(NSDictionary *)param;

@end

@implementation GlobalSearchEmbedderPlugin

- (void)createView {
    if (self.listVC == nil) {

        [self initListViewController];

        [self bindCacheData];

        /* listVC 挂载到 UIViewController，生命周期保持一致*/
        UIViewController *rootVC = (UIViewController *)[self currentVisibleViewController];
        [rootVC addChildViewController:self.listVC];
    }
}

- (void)setDestinationData:(NSDictionary *)param {
    ///  省略业务代码部分

    [self.listVC sendSerivce];
}

@end
```

Android 平台上对于 flutter view 嵌入 RN 容器有相似的流程。首先需要在 RN 初始化时创建 ViewReactPackage，它会提供给 RN 一个 RNViewManager，伪代码如下：

```
/// Android
class GlobalSearchEmbedderPlugin : ReactPackage {
    override fun createNativeModules(reactContext: ReactApplicationContext): MutableList<NativeModule> {
        return ArrayList()
    }
```

```
    override fun createViewManagers(reactContext: ReactApplicationContext): MutableList<SimpleViewManager<*>> {
        val result = ArrayList<SimpleViewManager<*>>()
        val listManager = RNViewManager()
        result.add(listManager)
        return result
    }
}
```

RNViewManager 在 RN 里注册嵌入 native 模块的名字、layout、RN 和 native 通信接口实现。native 模块的名字需要与 RN 中的 RCT_EXPORT_MODULE 名字、iOS native 模块的名字对应。command 接口实现了相关业务逻辑, 比如 initFlutterFragment()方法中创建 flutter view, 其它 command 接口中实现了目的地关键字、入离日期以及业务埋点数据等等。

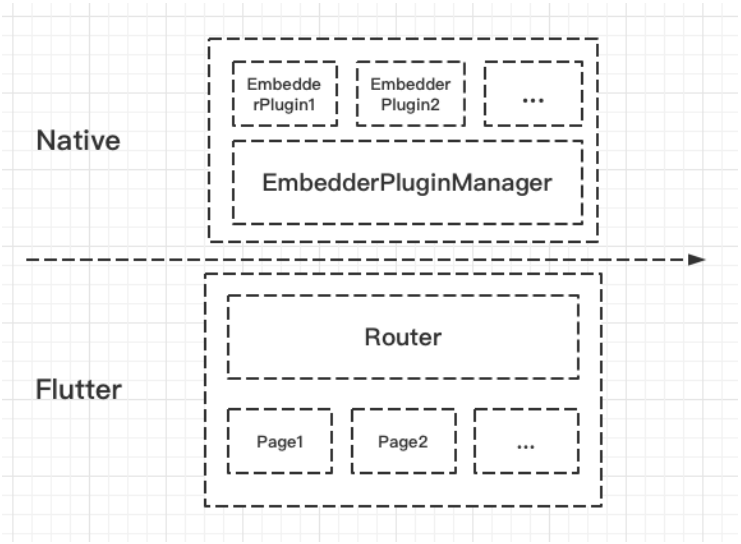
```
/// Android
class RNTListManager : SimpleViewManager<RNLinearLayout>() {
    companion object {
        const val COMMAND_SET = 1
    }

    override fun getName(): String {
        return "FlutterListView"
    }

    override fun createViewInstance(reactContext: ThemedReactContext): RNLinearLayout {
        return LayoutInflater.from(reactContext)
            .inflate(R.layout.flutter_container, null) as RNLinearLayout
    }

    override fun getCommandsMap(): MutableMap<String, Int> {
        return MapBuilder.of(
            "setDestinationParam", COMMAND_SET,
        )
    }

    override fun receiveCommand(root: RNLinearLayout, commandId: String?, args: ReadableArray?) {
        when (commandId?.toInt()) {
            COMMAND_SET -> initFlutterFragment(root.context, args)
        }
    }
}
```

RN native layout 有一些特殊，由于 flutter view 自身是一个 framelayout，RN native layout

定义为一个子 view 是 framelayout 的 linearlayout，这样可以实现动态地在 RN native viewGroup 中加入 flutter view。根据官方文档，RN native view 需要覆写 requestLayout()方法，并在方法中重新做测量和布局，伪代码如下：

```
/// Android
public class RNLinearLayout extends LinearLayout {
    public RNLinearLayout(Context context) {
        super(context);
    }

    public RNLinearLayout(Context context, AttributeSet attrs) {
        super(context, attrs);
    }

    public RNLinearLayout(Context context, AttributeSet attrs, int defStyleAttr) {
        super(context, attrs, defStyleAttr);
    }

    private final Runnable measureAndLayout = () -> {
        measure(
                MeasureSpec.makeMeasureSpec(getWidth(), MeasureSpec.EXACTLY),
                MeasureSpec.makeMeasureSpec(getHeight(), MeasureSpec.EXACTLY));
        layout(getLeft(), getTop(), getRight(), getBottom());
    };

    @Override
    public void requestLayout() {
        super.requestLayout();
        post(measureAndLayout);
    }
}
```

考虑到 flutter view 并不是单一场景使用，比如上述 1.2 节的场景二，需要在酒店查询页移植列表页。最终的结构设计如下图，Native 层对应具体的业务创建对应的 EmbedderPlugin，负责处理相关的事件，flutter 层在不同路由下对应具体的 page。



flutterEmbedder 结构

三、Native 嵌套 Flutter

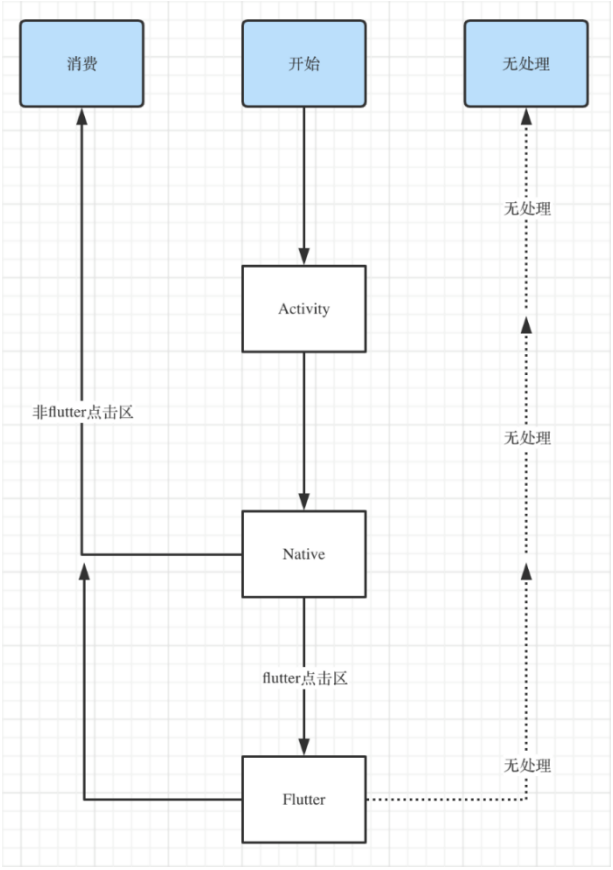
3.1 可行方案的探究

从 view 树的角度，RN 嵌套 flutter 的实现和 native 嵌套 flutter 的实现是一致的。RN 嵌套 flutter 时，flutter view 作为一个 view group 加入到 RN container 中，而 native 嵌套 flutter 时，flutter view 作为一个 view group 直接加入到 native view 树中。这样的实现需要考虑四个要点：点击事件传递、view 启动顺序、flutter 层与 native 层的业务交互、页面的生命周期。

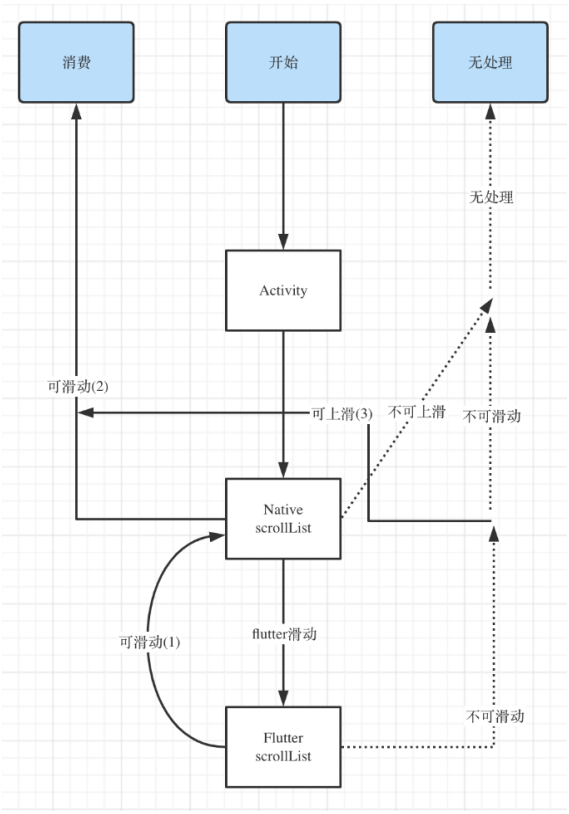
3.2 方案实现

3.2.1 点击事件传递

处理点击事件传递，flutter view 作为一颗 view 子树，能够直接接受到从上到下传递的点击事件。点击事件传递过程如下左图所示，在 flutter 点击区域由 flutter 处理事件，若 flutter 不处理则回到父 view 处理。



flutter 点击事件



flutter 滑动事件

list 滚动事件则需要在 flutter view 子树的祖先 view 中进行适当屏蔽，确保 flutter 列表能嵌套滚动。本次实现的业务场景是 1.2 节中的场景二，在一个 native 滚动列表最下方嵌入 flutter 滚动列表，flutter 滚动列表正好能占满一个屏幕。整个列表向下滚动过程中，先滚动外层列表，当滚动到底部时滚动 flutter 列表；反之，整个列表向上滚动过程中，先滚动 flutter 列表，当 flutter 列表滚动到头部时滚动，向上滚动外层列表。

如上图所示，滑动过程(1)是 flutter 列表可滑动场景，需要将事件返回外层列表；滑动过程(2)是列表可滑动场景；滑动过程(3)是 flutter 列表不可上滑，而外层列表可上滑场景，此时需要将事件传递到外层列表使其上滑，这个过程中会有顿挫感，我们在实现中给外层列表添加了滑动效果进行补偿。

3.2.2 view 启动顺序

通常是先创建 native view 树，在 view 树创建成功后，手动创建 flutter view 并加入 view 树中。手动创建 flutter view 可以根据业务需要，以懒加载的方式创建。在 app 启动之后，不管是否启动 flutter view，都需要先初始化 flutter 引擎。

3.2.3 flutter-native 业务交互

业务在 flutter 层与 native 层的交互，主要通过 flutter method channel，在 native 层预先注册 method channel 和各种事件，在 flutter view 启动之后由 flutter 层或 native 层双向发送消息。

3.2.4 页面的生命周期

生命周期已在 2.3.1 节中详细描述，可以由 native 层容器或者 flutter view 来控制，通常是根据业务所占页面比例决定，我们的实现中是将 flutter view 包在一个 native 容器中，这样可以用相同的方法在 native 控制生命周期。

四、写在最后

至此我们初步探索了不同技术栈 UI 的嵌套，为之后的组件的复用提供了一套切实可行的实践依据。后续会在此基础上做进一步的优化，比如 flutter view 的滚动事件如何很平滑地传输到 native，使得双列表嵌套滚动的时候没有顿挫感。在实践中，随着组件复杂度的和依赖度升高，混合的改造成本也是逐步增加的，那么是否需要混合、如何轻量化的移植也是需要进一步衡量和思考的。

参考文献

- Native UI Components
- flutter_boost
- ReactNative 流程源码分析

聊聊移动端安全加固

【作者简介】老松树，携程高级开发经理，专注于移动应用的信息安全。

随着移动互联网产业的高速发展，智能手机的全面普及，移动 App 已经无处不在。据统计，我国智能手机用户达到 12 亿，手机 App 总量达到 400 万款。手机 APP 在方便人们生活之余，也带来了巨大的安全隐患。

App 主要面临的风险：

1）静态攻击风险

APP 通常很容易被反编译逆向分析源码，出现被破解、篡改、广告植入、二次打包、仿冒/钓鱼应用等风险。

2）动态攻击风险

由于 APP 运行环境和用户操作行为不可控，导致 APP 在运行过程中面临各种动态攻击，如模拟器、多开器、加速器、注入攻击、动态调试、设备篡改、位置欺诈等攻击。

3）业务作弊风险

企业大量业务已经由线下及 Web 端转至手机 APP 端，目前地下黑灰产通常在 APP 注册、登录、营销活动场景进行批量化、机器化作业，威胁平台利益和用户账号安全。

移动安全已经成为互联网企业发展过程中面临的一个重要问题，需要更多的关注和投入。

一、手机 APP 安全加固介绍

解决移动安全问题，要从 APP 前端的加固，和业务后端的分析两方面进行。本文将介绍手机 APP 安全加固方面的知识。

下面从 iOS 平台着手，从以下几个方面，介绍移动端安全加固的方案。

一个标准的安全加固 SDK，要包含以下几个要素：

- 运行环境检测
- 环境数据的收集
- 符号，代码的混淆
- 算法的混淆
- 虚拟机技术

二、运行环境检测

确认 APP 的运行环境是否安全，是 APP 加固的最基础条件。所以我们的加固 SDK 中，要尽量靠前的进行 APP 的检测。

检测主要包括以下方面：

- 越狱检测
- Hook 检测
- Debug 检测
- 重签名检测
- 模拟器检测
- 代理检测

2.1 越狱检测

2.1.1 越狱概念

越狱是指利用 iOS 系统的某些漏洞，取得到 iOS 的 root 权限，然后改变一些程序使得设备的功能得到加强，突破封闭式环境。

越狱使得第三方管理工具可以完全访问 iOS 设备的所有目录，并可安装更改系统功能的插件和盗版的软件。

对于攻击者来说，越狱一般是 iOS 设备破解的第一步，只有越狱过才能实现后续的安装插件，砸壳获取二进制文件等一系列操作。

2.1.2 越狱检测

根据越狱手机和非越狱手机权限等方面的区别，我们可以检测当前环境是否是越狱设备。

例如，可以检测下面这些目录：

- 检测系统根目录文件
- 检测是否有一些越狱插件

```
/bin/bash
/bin/sh
/var/lib/cydia/
/Applications/Cydia.app
/Library/MobileSubstrate
...
```

如果这些目录存在，则表示当前是越狱设备。

2.2 Hook 检测

2.2.1 Hook 概念

Hook（钩子），实际上是一个处理消息的程序段，通过系统调用，把它挂入系统。每当特定的消息发出，在没有到达目的窗口前，钩子程序就先捕获该消息。在 Hook 程序执行完，再决定后续的操作。

攻击者在破解 APP 时，经常会通过 Hook 方式，对 APP 中的关键位置的函数进行 Hook。

通过 Hook 手段，攻击者可以方便破解分析，可以模拟发送业务请求，可以调用加密方法进行数据处理等。

2.2.2 Hook 检测

iOS 中的 Hook 主要有以下几种:

- 通过 iOS 的 runtime 消息转发的特性，实现对 iOS 函数的 Hook。
- 利用程序在调用外部函数时，通过符号表查询函数实际地址，实现的例如 C 系统函数的 Hook。
- 利用第三方插件库，实现汇编级别的 Hook。

在 Hook 检测时，可以通过各类 Hook 的原理和特性，进行反向的检测。

对于汇编级别的 Hook，通过比较函数内的汇编代码，来判断是否有汇编级别的 Hook。

2.3 Debug 检测

2.3.1 Debug

在破解 App 包时，需要通过 GDB 或 LLDB 来进行调试。所以对 APP 的调试检测，也是必须的。

2.3.2 Debug 检测

可以通过 ptrace 等方法，查看是否处于调试状态。对于已经发布的包，如果检测出调试状态，应该及时终止进程，阻止攻击者的攻击。并且让崩溃点，尽量远离检测点，并清空进程信息，防止攻击者确认检测关键点位置。

2.4 重签名检测

2.4.1 重签名

App Store 下载的应用，在砸壳之后，可以通过重签名的方式，实现 App 包的安装运行，实现逻辑的修改，APP 的多开等功能。

2.4.2 重签名检测

对于包的重签名检测，可以从以下两方面进行：

- 检测 embedded.mobileprovision 中的签名信息，是否正确。
- App Store 下载的包都是加密的，砸壳后的包是没有加密的，可以检测加密标志的状态。

2.5 模拟器检测

攻击者也有可能使用 ARM 模拟器运行，需要进行模拟器检测。

2.6 代理检测

攻击者可以通过设置网络代理，抓取 App 的业务请求，实现修改重发等操作，要进行必要的代理检测。

三、环境数据的收集

通过收集 App 运行手机的环境信息，尽可能的识别和标识运行设备，跟踪危险用户动作行为，进行持续分析。

通过设备信息的聚合分析，判断用户的环境是否真实，是否是虚拟的伪造的设备。例如：对于攻击者通过设备，进行自动化的注册登录等行为，通过设备 ID 等信息的聚合，可以发现其中的一些规律。

手机的环境信息，列举部分：

IDFA 广告标示符
IDFV Vendor(供应商)标识
Appleid appleid 的相关的一串 id
Wifi Wi-Fi 信息
fingerprint 设备指纹信息
os_model 系统型号信息
CUP 信息
手机型号
屏幕宽高
内存大小
手机容量
.....

通过种类繁多的设备信息的收集，上传到后端后，由后端通过 AI，大数据等手段进行设备打分，设备聚合等一系列操作，给设备定义出一个安全范围。为安全人员的后续分析，提供有力的数据支撑。

四、符号，代码的混淆

4.1 符号的混淆

对代码中，关键部分的代码，类名，函数名等部分，要进行混淆。让攻击者无法轻易通过名称，理解当前代码的功能含义，增加破解难度。

可以通过宏定义的方式，混淆函数名称，例如：

```
#define   MyClass          FX123ADHZA
#define   propertyName     XZAF891AFJ
#define   method_arg1      KAJ18XA91F
#define   method_arg2      JAC12DDASS
#define   method_arg3      WIAK198FJS

@interface MyClass : NSObject
@property (nonatomic,copy) NSString * propertyName;
- (void)method_arg1:(NSString *)arg1 method_arg2:(NSString *)arg2 method_arg3:(NSString *)arg3;
@end
```

编译出来的代码如下图所示，混乱的方法名，让攻击者理解方法含义的成本增加：

4.2 字符串混淆

攻击者在进行逆向分析时，通常会根据一些特定的字符串，查找定位到代码的关键位置。所以在安全加固的过程中，不让关键的字符串以明文的形式存在，要对字符串进行混淆，在运行使用时，再解析出来：

例如字符串，对这段字符串可以使用异或等方式进行混淆，混淆后的结果为：

0123456789ABCDEF

混淆后的效果为：

```
__const:000000010004ADF8 byte_10004ADF8  DCB 6
__const:000000010004ADF8
__const:000000010004ADF9                 DCB 0x87
__const:000000010004ADF9
__const:000000010004ADFA                 DCB 0xEF
__const:000000010004ADFA
__const:000000010004ADFB                 DCB 0x53 ; S
__const:000000010004ADFB
__const:000000010004ADFC                 DCB    9
__const:000000010004ADFC
__const:000000010004ADFD                 DCB 0xD1
__const:000000010004ADFD
__const:000000010004ADFE                 DCB    8
__const:000000010004ADFF                 DCB 0x53 ; S
__const:000000010004AE00                 DCB 0x4F ; O
__const:000000010004AE00
__const:000000010004AE01                 DCB 0x51 ; Q
__const:000000010004AE02                 DCB 0xF8
__const:000000010004AE03                 DCB 0x65 ; e
__const:000000010004AE04                 DCB 0x8F
__const:000000010004AE05                 DCB 0xB4
__const:000000010004AE06                 DCB 0xF0
__const:000000010004AE07                 DCB 0x80
```

4.3 数据混淆

常用的加密算法，算法通常都有一些运算的模数。对这些模数，也要进行混淆处理，在使用时再解析出来使用。

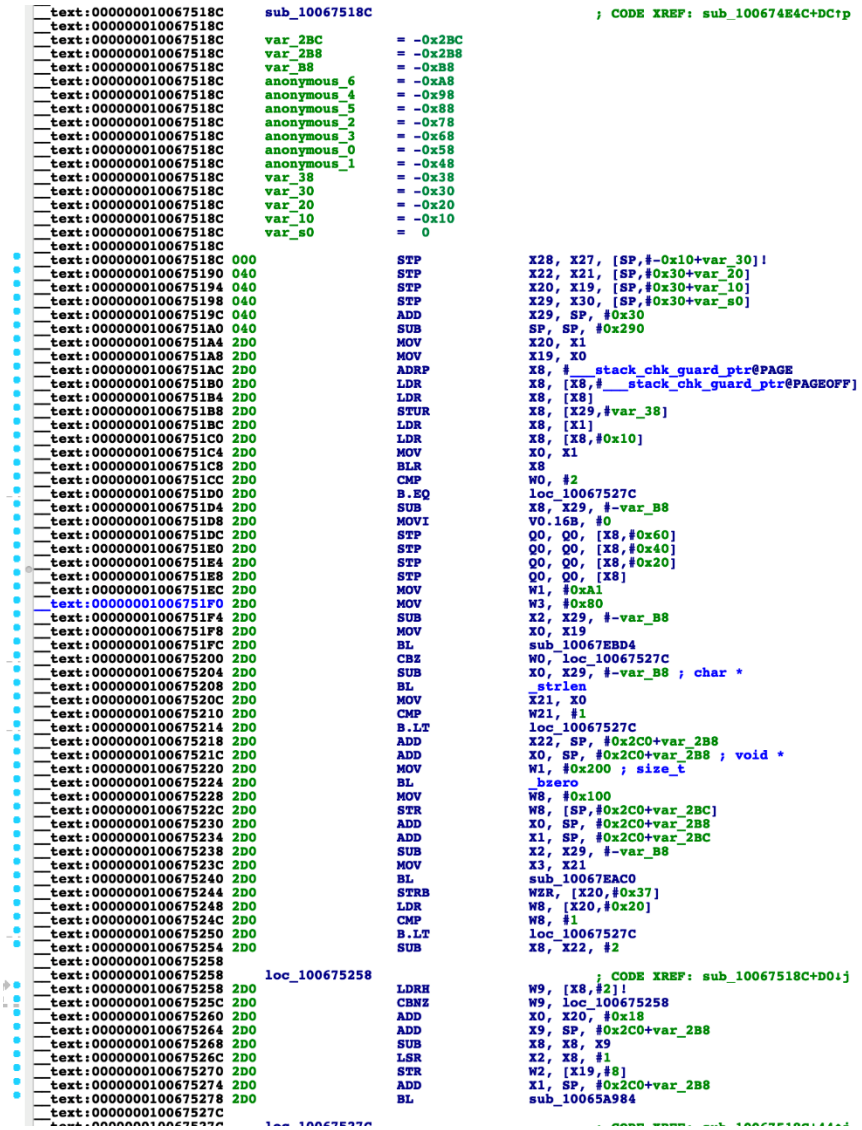
五、算法的混淆

攻击者逆向过程中，通过砸壳获取到 APP 的二进制文件。将二进制文件放入 IDA 软件，可以进行静态分析，还可以通过插件，反编译出伪代码。

所以，在安全加固的过程中，要对安全 SDK 中的关键算法，进行算法的编译混淆。

通过 Clang，在编译过程中，将算法逻辑打乱重排，嵌套一些 while 循环，或者 switch 分支，并且插入一些混淆代码，防止攻击者的反编译。

例如，正常的算法的汇编代码：



```
__text:000000010067518C     sub_10067518C                           ; CODE XREF: sub_100674E4C+DC↑p
__text:000000010067518C
__text:000000010067518C     var_2BC         = -0x2BC
__text:000000010067518C     var_2B8         = -0x2B8
__text:000000010067518C     var_B8          = -0xB8
__text:000000010067518C     anonymous_6     = -0xA8
__text:000000010067518C     anonymous_4     = -0x98
__text:000000010067518C     anonymous_5     = -0x88
__text:000000010067518C     anonymous_2     = -0x78
__text:000000010067518C     anonymous_3     = -0x68
__text:000000010067518C     anonymous_0     = -0x58
__text:000000010067518C     anonymous_1     = -0x48
__text:000000010067518C     var_38          = -0x38
__text:000000010067518C     var_30          = -0x30
__text:000000010067518C     var_20          = -0x20
__text:000000010067518C     var_10          = -0x10
__text:000000010067518C     var_s0          =  0
__text:000000010067518C
__text:000000010067518C 000                 STP             X28, X27, [SP,#-0x10+var_30]!
__text:0000000100675190 040                 STP             X22, X21, [SP,#0x30+var_20]
__text:0000000100675194 040                 STP             X20, X19, [SP,#0x30+var_10]
__text:0000000100675198 040                 STP             X29, X30, [SP,#0x30+var_s0]
__text:000000010067519C 040                 ADD             X29, SP, #0x30
__text:00000001006751A0 040                 SUB             SP, SP, #0x290
__text:00000001006751A4 2D0                 MOV             X20, X1
__text:00000001006751A8 2D0                 MOV             X19, X0
__text:00000001006751AC 2D0                 ADRP            X8, #___stack_chk_guard_ptr@PAGE
__text:00000001006751B0 2D0                 LDR             X8, [X8,#___stack_chk_guard_ptr@PAGEOFF]
__text:00000001006751B4 2D0                 LDR             X8, [X8]
__text:00000001006751B8 2D0                 STUR            X8, [X29,#var_38]
__text:00000001006751BC 2D0                 LDR             X8, [X1]
__text:00000001006751C0 2D0                 LDR             X8, [X8,#0x10]
__text:00000001006751C4 2D0                 MOV             X0, X1
__text:00000001006751C8 2D0                 BLR             X8
__text:00000001006751CC 2D0                 CMP             W0, #2
__text:00000001006751D0 2D0                 B.EQ            loc_10067527C
__text:00000001006751D4 2D0                 SUB             X8, X29, #-var_B8
__text:00000001006751D8 2D0                 MOVI            V0.16B, #0
__text:00000001006751DC 2D0                 STP             Q0, Q0, [X8,#0x60]
__text:00000001006751E0 2D0                 STP             Q0, Q0, [X8,#0x40]
__text:00000001006751E4 2D0                 STP             Q0, Q0, [X8,#0x20]
__text:00000001006751E8 2D0                 STP             Q0, Q0, [X8]
__text:00000001006751EC 2D0                 MOV             W1, #0xA1
__text:00000001006751F0 2D0                 MOV             W3, #0x80
__text:00000001006751F4 2D0                 SUB             X2, X29, #-var_B8
__text:00000001006751F8 2D0                 MOV             X0, X19
__text:00000001006751FC 2D0                 BL              sub_10067EBD4
__text:0000000100675200 2D0                 CBZ             W0, loc_10067527C
__text:0000000100675204 2D0                 SUB             X0, X29, #-var_B8 ; char *
__text:0000000100675208 2D0                 BL              _strlen
__text:000000010067520C 2D0                 MOV             X21, X0
__text:0000000100675210 2D0                 CMP             W21, #1
__text:0000000100675214 2D0                 B.LT            loc_10067527C
__text:0000000100675218 2D0                 ADD             X22, SP, #0x2C0+var_2B8
__text:000000010067521C 2D0                 ADD             X0, SP, #0x2C0+var_2B8 ; void *
__text:0000000100675220 2D0                 MOV             W1, #0x200 ; size_t
__text:0000000100675224 2D0                 BL              _bzero
__text:0000000100675228 2D0                 MOV             W8, #0x100
__text:000000010067522C 2D0                 STR             W8, [SP,#0x2C0+var_2BC]
__text:0000000100675230 2D0                 ADD             X0, SP, #0x2C0+var_2B8
__text:0000000100675234 2D0                 ADD             X1, SP, #0x2C0+var_2BC
__text:0000000100675238 2D0                 SUB             X2, X29, #-var_B8
__text:000000010067523C 2D0                 MOV             X3, X21
__text:0000000100675240 2D0                 BL              sub_10067EAC0
__text:0000000100675244 2D0                 STRB            WZR, [X20,#0x37]
__text:0000000100675248 2D0                 LDR             W8, [X20,#0x20]
__text:000000010067524C 2D0                 CMP             W8, #1
__text:0000000100675250 2D0                 B.LT            loc_10067527C
__text:0000000100675254 2D0                 SUB             X8, X22, #2
__text:0000000100675258
__text:0000000100675258     loc_100675258                           ; CODE XREF: sub_10067518C+D0↓j
__text:0000000100675258 2D0                 LDRH            W9, [X8,#2]!
__text:000000010067525C 2D0                 CBNZ            W9, loc_100675258
__text:0000000100675260 2D0                 ADD             X0, X20, #0x18
__text:0000000100675264 2D0                 ADD             X9, SP, #0x2C0+var_2B8
__text:0000000100675268 2D0                 SUB             X8, X8, X9
__text:000000010067526C 2D0                 LSR             X2, X8, #1
__text:0000000100675270 2D0                 STR             W2, [X19,#8]
__text:0000000100675274 2D0                 ADD             X1, SP, #0x2C0+var_2B8
__text:0000000100675278 2D0                 BL              sub_10065A984
__text:000000010067527C
```

这样的汇编代码，可以在 IDA 中，通过 IDA 等分析工具，可以反编译出伪代码，攻击者可以很方便的进行分析，通过调试还原出代码的原始逻辑：

```
size_t __fastcall sub_10067518C(__int64 a1, __int64 a2)
{
  __int64 v2; // x20
  __int64 v3; // x19
  size_t result; // x0
  size_t v5; // x21
  char *v6; // x8
  int v7; // t1
  int v8; // [xsp+4h] [xbp-2BCh]
  char v9; // [xsp+8h] [xbp-2B8h]
  __int128 v10; // [xsp+208h] [xbp-B8h]
  __int128 v11; // [xsp+218h] [xbp-A8h]
  __int128 v12; // [xsp+228h] [xbp-98h]
  __int128 v13; // [xsp+238h] [xbp-88h]
  __int128 v14; // [xsp+248h] [xbp-78h]
  __int128 v15; // [xsp+258h] [xbp-68h]
  __int128 v16; // [xsp+268h] [xbp-58h]
  __int128 v17; // [xsp+278h] [xbp-48h]

  v2 = a2;
  v3 = a1;
  result = (*(__int64 (__fastcall **)(__int64))(*(_QWORD *)a2 + 16LL))(a2);
  if ( (_DWORD)result != 2 )
  {
    v16 = 0u;
    v17 = 0u;
    v14 = 0u;
    v15 = 0u;
    v12 = 0u;
    v13 = 0u;
    v10 = 0u;
    v11 = 0u;
    result = sub_10067EBD4(v3, 161LL, &v10, 128LL);
    if ( (_DWORD)result )
    {
      result = strlen((const char *)&v10);
      v5 = result;
      if ( (signed int)result >= 1 )
      {
        bzero(&v9, 0x200uLL);
        v8 = 256;
        result = sub_10067EAC0(&v9, &v8, &v10, v5);
        *(_BYTE *)(v2 + 55) = 0;
        if ( *(_DWORD *)(v2 + 32) >= 1 )
        {
          v6 = (char *)&v8 + 2;
          do
          {
            v7 = *((unsigned __int16 *)v6 + 1);
            v6 += 2;
          }
          while ( v7 );
          *(_DWORD *)(v3 + 8) = (unsigned __int64)(v6 - &v9) >> 1;
          result = sub_10065A984(v2 + 24, &v9);
        }
      }
    }
  }
  return result;
}
```

而我们在编译过程中，将汇编代码进行打乱重排，插入虚假判断逻辑等，可以有效的进行防范。如下图所示，便是一种混淆模式，将算法混淆之后的效果:

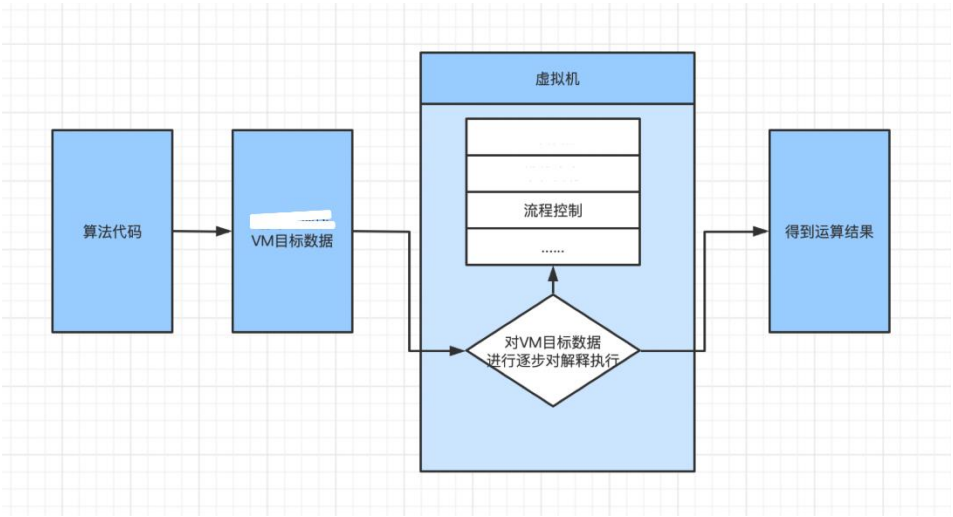
混淆后的代码，代码段被切割重排，中间还插入了无用代码，使攻击者无法阅读，更增加了攻击者调试的成本。

六、虚拟机技术

在安全加固方面，安全性最高的一种加固手段，就是虚拟机加固技术。它是将原始代码的算法代码，编译为动态的 VM 虚拟机指令，在虚拟机中解释执行，最终计算出计算结果。

加密后的代码，具有不可逆性，无法被反编译回源代码。算法代码动态执行，单步执行，难以调试分析。

而且，虚拟机处理的算法代码，可以动态下发，极大的提高了算法的安全性。



七、携程安全加固 SDK 的产品功能

7.1 种类齐全的设备信息

携程安全加固产品，经用户授权后，收集种类相对齐全的设备信息，通过设备信息，后端分析算法，可以高效辨别设备的真实性。

通过大数据技术，对设备信息进行多维度分析，给设备进行打分，建立用户安全等级。

7.2 多样/隐蔽/安全的设备环境监测

我们对设备的环境监测，越狱，调试，重签名等检测项，都提供了两种以上的检测方案，做到多方案互备。

每种检测方法，都使用了非常隐蔽的调用方法，减少被 Hook，被定位的可能性。让加固产品的环境监测更加安全可信。

7.3 自主实现的代码混淆技术

使用了自主实现的代码混淆技术(bnof)，对比市面上开源的混淆算法，更加安全和高效。

携程的代码混淆技术，可以抗优化，可以灵活调节代码切割粒度，执行效率更加高效。混淆后的汇编代码，无法进行反编译，修复成本极高，能对关键加密代码提供很好的保护。

7.4 自主实现的虚拟机技术

携程的安全加固产品，使用了自主实现的虚拟机技术(VM)，安全，稳定，可靠。虚拟机技术为核心的加密代码，提供了极强的保护。

八、总结

在严峻的安全形势下，携程加固产品上线以来，取得了良好的效果，为安全部门和业务部门的风控和反爬虫工作，提供了强力的支持。

大数据和人工智能

携程数据血缘构建及应用

【作者简介】cxzl25，携程软件技术专家，关注大数据领域生态建设，对分布式计算和存储、调度等方面有浓厚兴趣。

一、前言

Data lineage includes the data origin, what happens to it and where it moves over time. Data lineage gives visibility while greatly simplifying the ability to trace errors back to the root cause in a data analytics process. ——百科 Data lineage

大数据时代，数据的来源极其广泛，各种类型的数据在快速产生，数据也是爆发性增长。从数据的产生，通过加工融合流转产生新的数据，到最终消亡，数据之间的关联关系可以称之为数据血缘关系。

数据血缘是元数据管理、数据治理、数据质量的重要一环，追踪数据的来源、处理、出处，对数据价值评估提供依据，描述源数据流程、表、报表、即席查询之间的流向关系，表与表的依赖关系、表与离线 ETL 任务，调度平台，计算引擎之间的依赖关系。数据仓库是构建在 Hive 之上，而 Hive 的原始数据往往来自于生产 DB，也会把计算结果导出到外部存储，异构数据源的表之间是有血缘关系的。

数据血缘用途：

- 追踪数据溯源：当数据发生异常，帮助追踪到异常发生的原因；影响面分析，追踪数据的来源，追踪数据处理过程。
- 评估数据价值：从数据受众、更新量级、更新频次等几个方面给数据价值的评估提供依据。
- 生命周期：直观地得到数据整个生命周期，为数据治理提供依据。
- 安全管控：对源头打上敏感等级标签后，传递敏感等级标签到下游。

本文介绍携程数据血缘如何构建及应用场景。第一版 T+1 构建 Hive 引擎的表级别的血缘关系，第二版近实时构建 Hive，Spark，Presto 多个查询引擎和 DataX 传输工具的字段级别血缘关系。

二、构建血缘的方案

2.1 收集方式

- 方案一：只收集 SQL，事后分析。

当 SQL 执行结束，收集 SQL 到 DB 或者 Kafka。

优点：当计算引擎和工具不多的时候，语法相对兼容的时候，用 Hive 自带的 LineageLogger 重新解析 SQL 可以获得表和字段级别的关系。

缺点：重放 SQL 的时候可能元数据发生改变，比如临时表可能被 Drop，没有临时自定义函数 UDF，或者 SQL 解析失败。

- 方案二：运行时分析 SQL 并收集。

当 SQL 执行结束后立即分析 Lineage，异步发送到 Kafka。

优点：运行时的状态和信息是最准确的，不会有 SQL 解析语法错误。

缺点：需要针对各个引擎和工具开发解析模块，解析速度需要足够快。

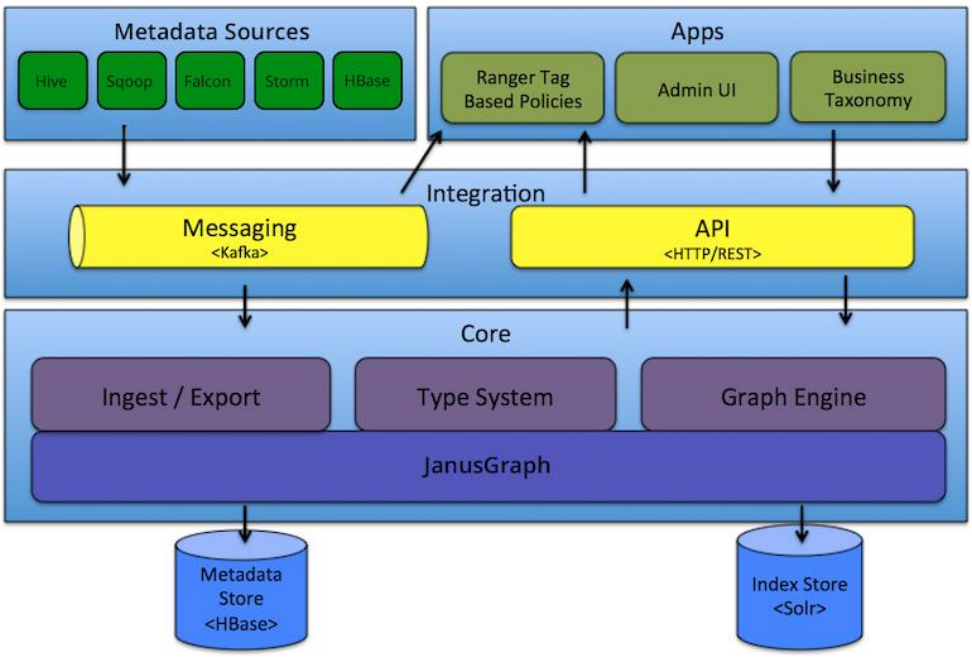
2.2 开源方案

- Apache Atlas

Apache Atlas 是 Hadoop 社区为解决 Hadoop 生态系统的元数据治理问题而产生的开源项目，它为 Hadoop 集群提供了包括数据分类、集中策略引擎、数据血缘、安全和生命周期管理在内的元数据治理核心能力。官方插件支持 HBase、Hive、Sqoop、Storm、Storm、Kafka、Falcon 组件。

Hook 在运行时采集血缘数据，发送到 Kafka。Atlas 消费 Kafka 数据，将关系写到图数据库 JanusGraph，并提供 REST API。

其中 Hive Hook 支持表和列级别血缘，Spark 需要使用 GitHub 的 hortonworks-spark/spark-atlas-connector，不支持列级别，Presto 则不支持。

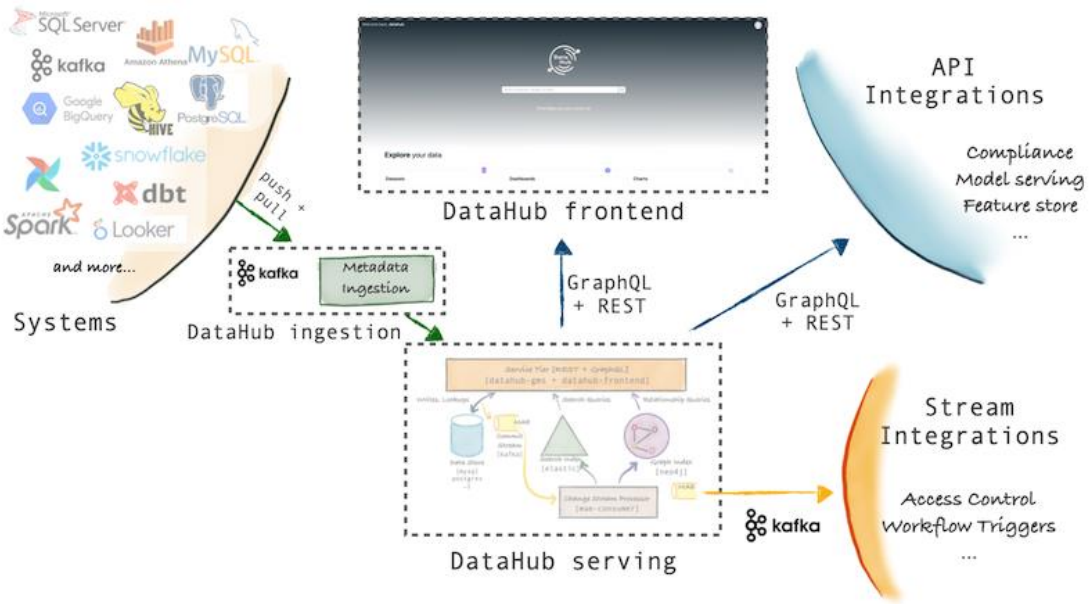


- Linkedin DataHub

WhereHows 项目已于 2018 年重新被 LinkedIn 公司设计为 DataHub 项目。它从不同的源系统中采集元数据，并进行标准化和建模，从而作为元数据仓库完成血缘分析。

社区提供了一个 Demo，演示地址：https://demo.datahubproject.io/

与 Airflow 集成较好，支持数据集级别血缘，字段级别在 2021Q3 的 Roadmap。



三、携程方案

携程采用了方案二，运行时分析 SQL 并收集分析结果到 Kafka。由于开源方案在现阶段不满足需求，则自行开发。

由于当时缺少血缘关系，对数据治理难度较大，表级别的血缘解析难度较低，表的数量远小于字段的数量，早期先快速实现了表级别版本。

在 16-17 年实现和上线了第一个版本，收集常用的工具和引擎的表级别的血缘关系，T+1 构建关系。

在 19 年迭代了第二个版本，支持解析 Hive，Spark，Presto 多个查询引擎和 DataX 传输工具的字段级别血缘关系，近实时构建关系。

四、第一个版本-表级别血缘关系

4.1 处理流程

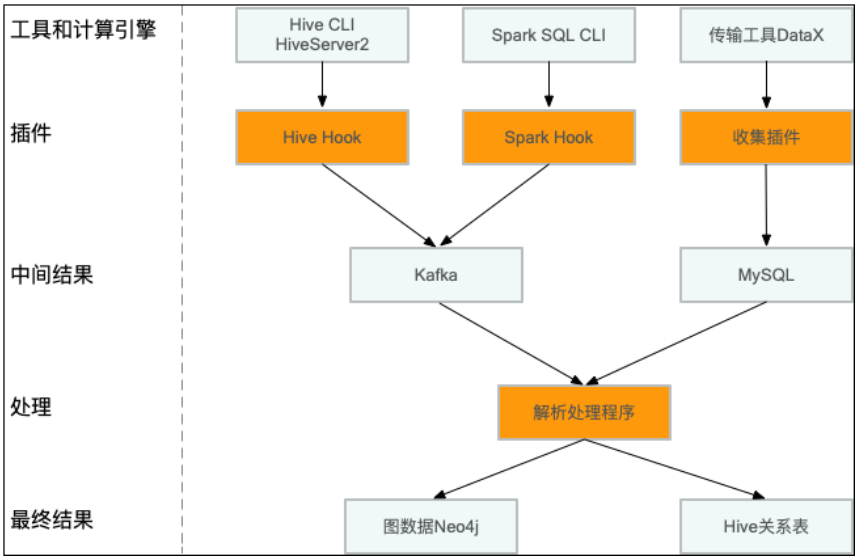
针对 Hive 引擎开发了一个 Hook，实现 ExecuteWithHookContext 接口，从 HookContext 可

以获得执行计划，输入表，输出表等丰富信息，异步发送到 Kafka，部署的时候在 hive.exec.post.hooks 添加插件即可。

在 17 年引入 Spark2 后，大部分 Hive 作业迁移到 Spark 引擎上，这时候针对 Spark SQL CLI 快速开发一个类似 Hive Hook 机制，收集表级别的血缘关系。

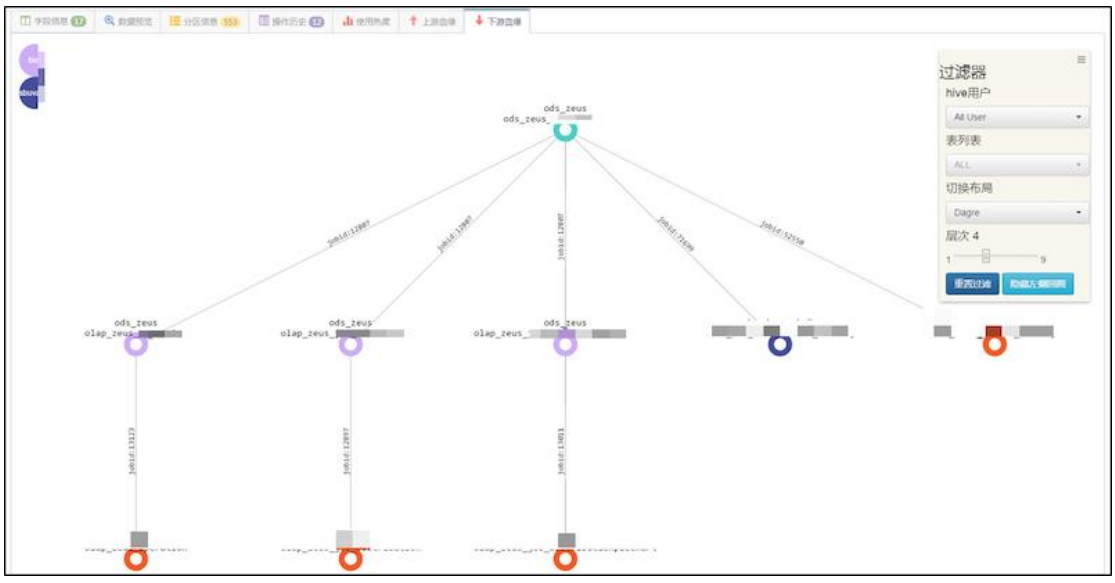
传输工具 DataX 作为一个异构数据源同步的工具，单独对其开发了收集插件。

在经过解析处理后，将数据写到图数据库 Neo4j，提供元数据系统展示和 REST API 服务，落地成 Hive 关系表，供用户查询和治理使用。



4.2 效果

在元数据系统上，可以查看一张表多层级的上下游血缘关系，在关系边上会有任务 ID 等一些属性。



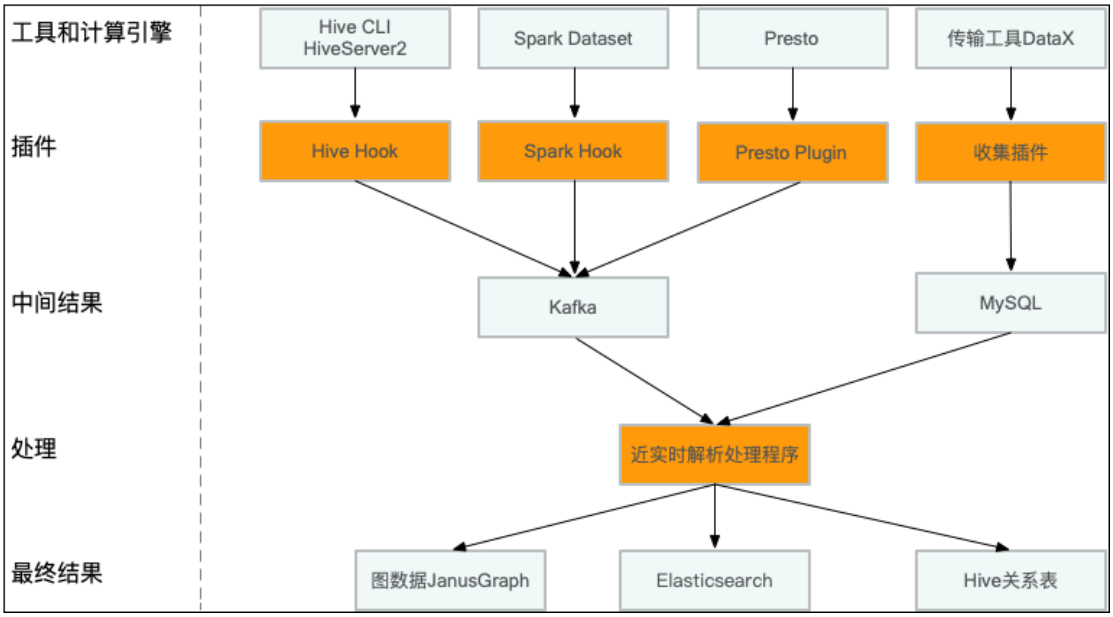
4.3 痛点

- 随着计算引擎的增加，业务的增长，表级别的血缘关系已经不满足需求。
- 覆盖面不足，缺少 Spark ThriftServer，Presto 引擎，缺少即席查询平台，报表平台等。
- 关系不够实时，期望写入表后可以快速查询到关系，用户可以直观查看输入和输出，数据质量系统，调度系统可以根据任务 ID 查询到输出表，对表执行质量校验任务。
- 图数据库 Neo4j 社区版为单机版本，存储数量有限，稳定性欠佳，当时使用的版本较低，对边不能使用索引(3.5 支持)，这使得想从关系搜索到关联的上下游较为麻烦。

五、第二版本-字段级别血缘关系

之前实现的第一个版本，对于细粒度的治理和追踪还不够，不仅缺少对字段级别的血缘关系，也不支持采集各个系统的埋点信息和自定义扩展属性，难以追踪完整链路来源，并且关系是 T+1，不够实时。

针对各个计算引擎和传输工具 DataX 开发不同的解析插件，将解析好的血缘数据发送到 Kafka，实时消费 Kafka，把关系数据写到分布式图数据 JanusGraph。

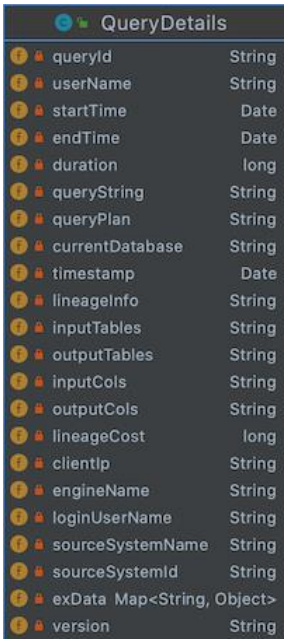


5.1 传输工具 DataX

阿里开源的 Druid 是一个 JDBC 组件库，包含数据库连接池、SQL Parser 等组件。通过重写 MySqlASTVisitor、SQLServerASTVisitor 来解析 MySQL / SQLServer 的查询 SQL，获得列级别的关系。

5.2 计算引擎

计算引擎统一格式，收集输入表、输出表，输入字段、输出字段，流转的表达式等一些信息。



Hive
参考 org.apache.hadoop.hive.ql.hooks.LineageLogger 实现，异步发送血缘数据到 Kafka。

Atlas 的 HiveHook 也是实现 ExecuteWithHookContext 接口，从 HookContext 获得 LineageInfo，也可以参考 HIVE-19288 引入的 org.apache.hadoop.hive.ql.hooks.HiveProtoLoggingHook，采集更多引擎相关的信息。

其中遇到几个问题：

- 通过 HiveServer2 执行获取的 start time 不正确

HIVE-10957 QueryPlan's start time is incorrect in certain cases

- 获取执行计划空指针，导致收集失败

HIVE-12709 further improve user level explain

- 获取执行计划有可能出现卡住，可以加个调用超时。

Spark

前置条件：引入 SPARK-19558 Add config key to register QueryExecutionListeners automatically，实现自动注册 QueryExecutionListener。

实现方式：通过实现 QueryExecutionListener 接口，在 onSuccess 回调函数拿到当前执行的 QueryExecution，通过 LogicalPlan 的 output 方法，获得所有 Attribute，利用 NamedExpression 的 exprId 映射关系，对其进行遍历和解析，构建列级别关系。

覆盖范围：Spark SQL CLI、Thrift Server、使用 Dataset/DataFrame API（如 spark-submit、spark-shell、pyspark）

遇到问题：

- 使用 analyzedPlan 而不是 optimizedPlan，optimizer 的执行计划可能会丢失一些信息，可以在 analyzedPlan 的基础上 apply 一些有助于分析的 Rule，如 CombineUnions。

- 传递的初始化用的 hiveconf/hivevar 变量被 Thrift Server 忽略，导致初始化 Connection 没有办法埋点。

打上 Patch SPARK-13983 ，可以实现第一步，传递变量，但是这个变量在每次执行新的 statement 都重新初始化，导致用户 set 的变量不可更新。后续给社区提交 PR SPARK-26598，修复变量不可更新的问题。

SPARK-13983 Fix HiveThriftServer2 can not get "--hiveconf" and "--hivevar" variables since 2.0
SPARK-26598 Fix HiveThriftServer2 cannot be modified hiveconf/hivevar variables

- Drop Table 的限制，DropTableCommand 执行成功的时候，该表不一定在之前存在过，如果在 Drop 之前存在过，元数据也已经被删除了，无从考证。

在 DropTableCommand 增加了一个标志位，真正在有执行 Drop 操作的话再置为 True，保证收集的血缘数据是对的。

- 使用 Transform 用户自定义脚本的限制

Transform 不像 java UDF，只输入需要用到的字段即可，而是需要将所有后续用到的字段都输入到自定义脚本，脚本再决定输出哪些字段，这其中列与列之间的映射关系无法通过执行计划获得，只能简单的记录输出列的表达式，如 transform(c1,c2,c3) script xxx.py to c4。

Presto

开发 Presto EventListener Plugin，实现 EventListener 接口，从 queryCompleted 回调函数的 QueryCompletedEvent 解析得到相应的信息。

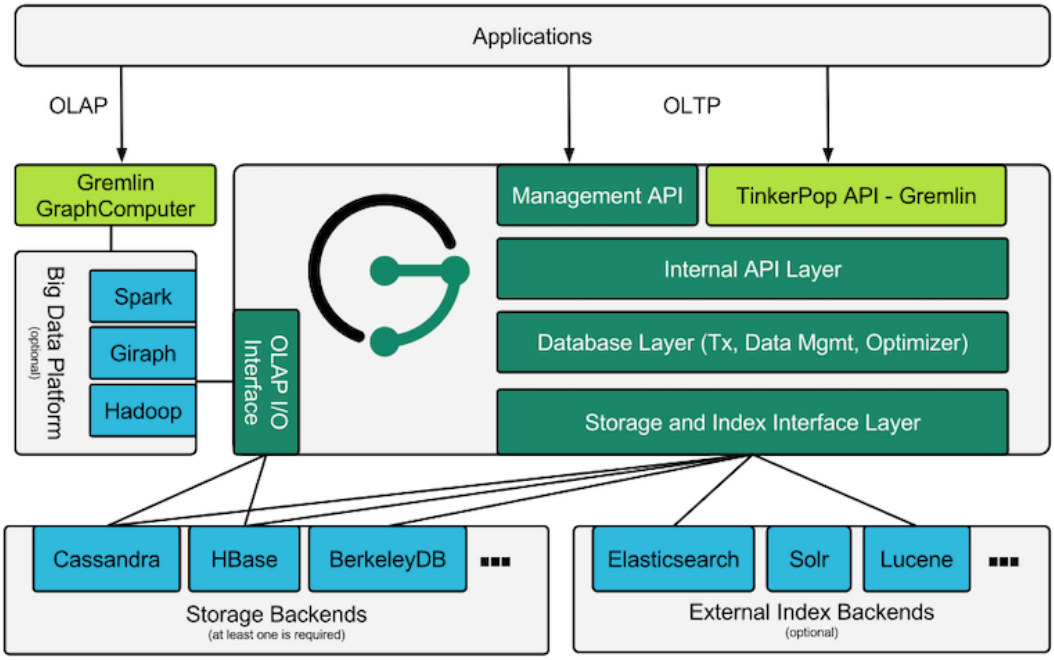
上线的时候遇到一个无法加载 Kafka 加载 StringSerializer 的问题（StringSerializer could not be found）。

Kafka 客户端使用 Class.forName(trimmed, true, Utils.getContextOrKafkaClassLoader()) 来加载 Class，优先从当前线程的 ContextClassLoader 加载，与 Presto 的 ThreadContextClassLoader 有冲突，需要初化始 KafkaProducer 的时候，将 ContextClassLoader 暂时置为 NULL。
https://stackoverflow.com/a/50981469/1673775

5.3 图数据库 JanusGraph

JanusGraph 是一个开源的分布式图数据库。具有很好的扩展性，通过多机集群可支持存储和查询数百亿的顶点和边的图数据。JanusGraph 是一个事务数据库，支持大量用户高并发地执行复杂的实时图遍历。

生产上，存储我们使用 Cassandra，索引使用 Elasticsearch，使用 Gremlin 查询/遍历语言来读写 JanusGraph，有上手难度，熟悉 Neo4j 的 Cypher 语法可以使用 cypher-for-gremlin plugin。



以下是数据血缘写入图数据库的模型，Hive 字段单独为一个 Lable，关系型 DB 字段为一个 Label，关系分两种，LABELWRITE，LABELWRITE_TTL。

只有输入没有输出（Query 查询操作），只有输出没有输入（建表等 DDL 操作）也会强制绑定一个来源系统的 ID 及扩展属性。

在生产上使用 JanusGraph，存储亿级的血缘关系，但是在开发过程中也遇到了一些性能问题。

- 写入速度优化

以 DB 名+表名+字段名作为唯一 key，实现 getOrCreateVertex，并对 vertex id 缓存，加速顶点的加载速度。

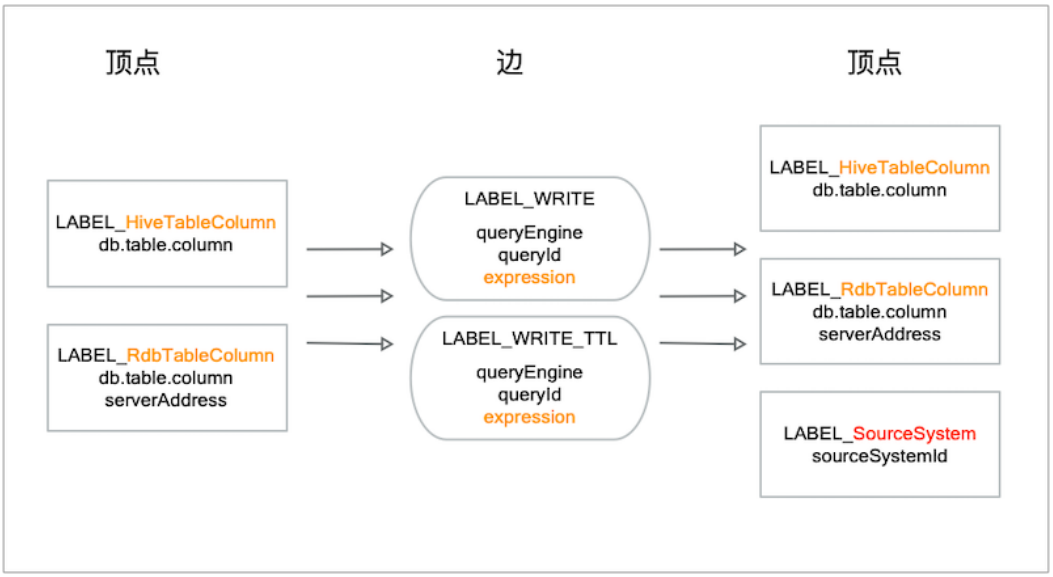
- 关系批量删除

关系 LABELWRITETTL 表示写入的关系有存活时间（TTL-Time to live），这是因为在批量删除关系的时候，JanusGraph 速度相当慢，而且很容易 OOM。比如要一次性删除，Label 为 WRITE，x=y，写入时间小于等于某个时间的边，这时候 Vertex 和 Edge load 到内存中，容易 OOM。

g.E().hasLabel("WRITE").has("x",eq("y")).has("publishedDate",P.lte(new Date(1610640000))).drop().iterate()

尝试使用多线程+分批次的方式，即 N 个线程，每个线程删除 1000 条，速度也不太可接受。

这时候采用了折中的方案，需要删除关系用另外一种 Label 来表示，并在创建 Label 指定了 TTL，由于 Cassandra 支持 cell level TTL，所以边的数据会自动被删除。但是 ES 不支持 TTL，实现一个定时删除 ES 过期数据即可。



5.4 覆盖范围

Zeus 调度平台 （ETL 操作 INSERT、CTAS，QUERY）
Ad-Hoc 即席查询平台 （CTAS，QUERY）
报表平台 （QUERY）
元数据平台 （DDL 操作）
GPU 平台 （PySpark）
通过 ETL 任务 ID，查询任务 ID，报表 ID，都可以获取到输入，输出的表和字段的关系。

5.5 局限

使用 MapReduce、Spark RDD 读写 HDFS 的血缘暂时没有实现。

思路可以在 JobClient.submitJob 的时候采集输入和输出路径，又或者通过 HDFS 的 AuditLog、CallerContext 来关联。

5.6 效果

在第一版使用图的方式展示血缘关系，在上下游关系较多的时候，显示较为混乱，第二版改成树状表格的方式展示。

字段 operator 在调度系统 Zeus 被转换成 hive_account，最后输出是 ArtNova 报表系统的一张报表。

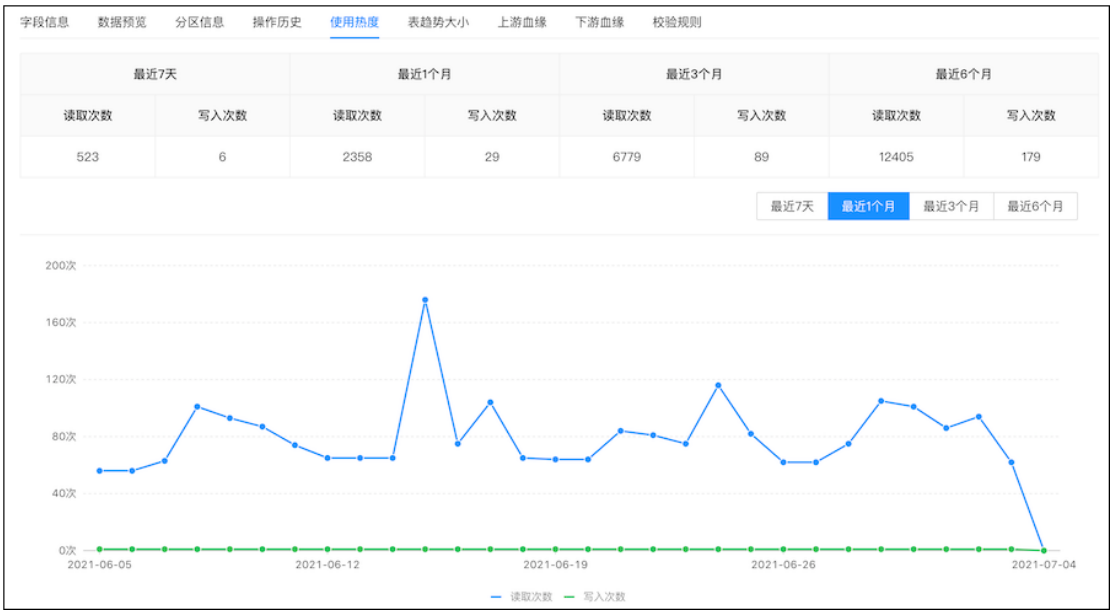
六、实际应用场景

6.1 数据治理

- 通过血缘关系筛选，每天清理数千张未使用的临时表，节约空间。
- 作为数据资产评估的依据，统计表、字段读写次数，生成的表无下游访问，包括有没有调度任务，报表任务，即席查询。

6.2 元数据管理

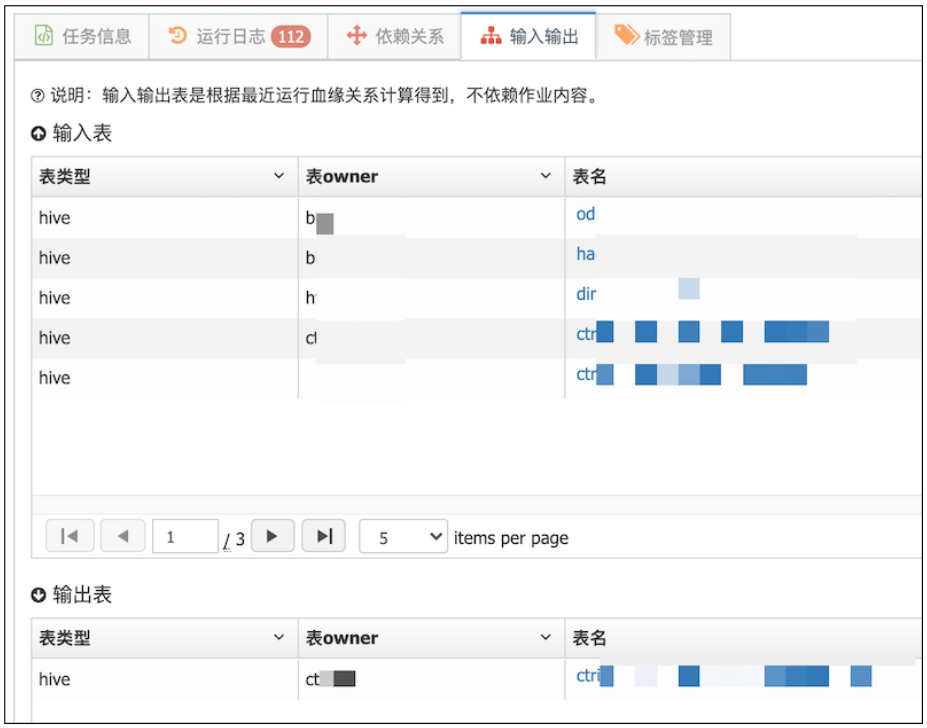
- 统计一张表的生成时间，而不是统计整个任务的完成时间。
- 数据异常，或者下线一张表、一个字段的时候，可以找到相关的 ETL 任务或者报表任务，及时通知下游。
- 统计表的使用热度，显示趋势。



6.3 调度系统

得益于在图数据库 JanusGraph 可以使用关系边的 key 作为索引，可以根据任务 ID 可以轻松获得该任务输入和输出表。

- 当配置一个任务 A 的依赖任务列表的时候，可以使用推荐依赖，检查依赖功能，获得任务 A 的所有输入表，再通过输入的表获得写入任务 ID 列表，即为任务 A 所需依赖的任务列表。
- 在任务结束后，获取该任务所有输出的表，进行预配的规则进行数据质量校验。



6.4 敏感等级标签

当源头的数据来自生产 DB 时，生产 DB 有些列的标签已打上了敏感等级，通过血缘关系，下游的表可以继承敏感等级，自动打上敏感标签。

七、总结

以上描述了携程如何构建表和字段级别的血缘关系，及在实际应用的场景。

随着业务需求和数据的增长，数据的加工流程越来越复杂，构建一套数据血缘，可以轻松查询到数据之间的关系，进行表和字段级的血缘追溯，在元数据管理，数据治理，数据质量上承担重要一环。

百万 QPS，秒级延迟，携程基于实时流的大数据基础层建设

【作者简介】纪成，携程数据开发总监，负责金融数据基础组件及平台开发、数仓建设与治理相关的工作。对大数据领域开源技术框架有浓厚兴趣。

一、背景

2017 年 9 月携程金融成立，本着践行金融助力旅行的使命，开始全面开展集团风控和金融业务，需要在携程 DC 构建统一的金融数据中心，实现多地多机房间的数据融合，满足离线和在线需求；涉及数千张 mysql 表到离线数仓 、实时数仓、在线缓存的同步工作。由于跨地域、实时性、准确性、完整性要求高，集团内二次开发的 DataX（业界常用的离线同步方案）无法支持。以 mysql-hive 同步为例，DataX 通过直连 MySQL 批量拉取数据，存在以下问题：

1）性能瓶颈：随着业务规模的增长，离线批量拉取的数据规模越来越大，影响 mysql-hive 镜像表的产出时间，进而影响数仓下游任务。对于一些需要 mysql-hive 小时级镜像的场景更加捉襟见肘。

2）影响线上业务：离线批量拉取数据，可能引起慢查询，影响业务库的线上服务。

3）无法保证幂等：由于线上库在实时更新，在批量拉取 SQL 不变的情况下，每次执行可能产生不一样的结果。比如指定了 create_time 范围，但一批记录的部分字段（比如支付状态）时刻在变化。也即无法产出一个明确的 mysql-hive 镜像 ， 对于一些对时点要求非常高的场景（比如离线对账） 无法接受。

4）缺乏对 DELETE 的支持：业务库做了 DELETE 操作后，只有整表全量拉取，才能在 Hive 镜像里体现。
—

二、方案概述

基于上述背景，我们设计了一套基于 binlog 实时流的数据基础层构建方案，并取得了预期效果。架构如图，各模块简介：

1）webUI 做 binlog 采集的配置，以及 mysql->hive，mysql→实时数仓，mysql→在线缓存的镜像配置工作。

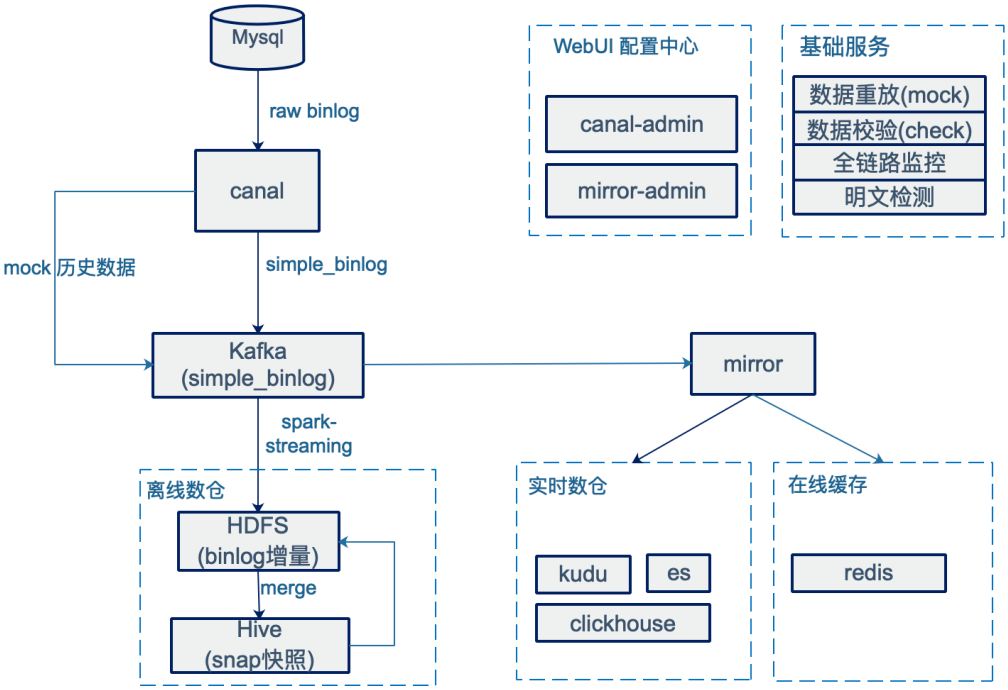
2）canal 负责 binlog 采集，写入 kafka；其中 kafka 在多地部署，并通过专线实现 topic 的实时同步。

3）spark-streaming 负责将 binlog 写入 HDFS。

4）merge 离线调度的 ETL 作业，负责将 HDFS 增量和 snap 合并成新的 snap。

5）mirror 负责将 binlog 事件更新到实时数仓、在线缓存。

6）基础服务：包括历史数据的重放，数据校验，全链路监控，明文检测等功能。



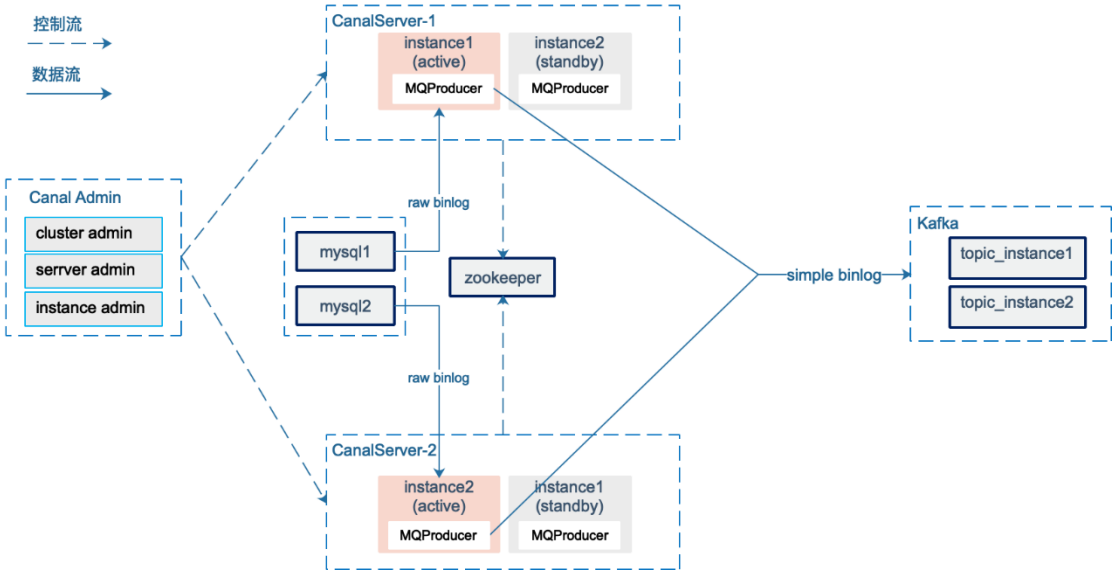
三、详细介绍

本章将以 mysql-hive 镜像为例，对技术方案做详细介绍。

3.1.binlog 采集

canal 是阿里巴巴开源的 Mysql binlog 增量订阅和消费组件，在业界有非常广泛的应用，通过实时增量采集 binlog，可以降低对 mysql 的压力，细粒度的还原数据的变更过程，我们选型 canal 作为 binlog 采集的基础组件，根据应用场景做了二次开发，其中 raw binlog → simple binlog 的消息格式转换是重点。

下面是 binlog 采集的架构图：



canal 在 1.1.4 版本引入了 canal-admin 工程，支持面向 WebUI 的管理能力；我们采用原生的 canal-admin 对 binlog 采集进行管理 ， 采集粒度是 mysql instance 级别。

Canal Server 会向 canalAdmin 拉取所属集群下的所有 mysql instance 列表，针对每个 mysql instance 采集任务，canal server 通过在 zookeeper 创建临时节点的方式实现 HA，并通过 zookeeper 实现 binlog position 的共享。

canal 1.1.1 版本引入 MQProducer 原生支持 kafka 消息投递 ，图中 instance active 从 mysql 获取实时的增量 raw binlog 数据，在 MQProducer 环节进行 raw binlog → simple binlog 的消息转换，发送至 kafka。我们按照 instance 创建了对应的 kafka topic，而非每个 database 一个 topic ，主要考虑到同一个 mysql instance 下有多个 database，过多的 topic (partition) 导致 kafka 随机 IO 增加，影响吞吐。发送 Kafka 时以 schemaName+tableName 作为 partitionKey，结合 producer 的参数控制，保证同一个表的 binlog 消息按顺序写入 kafka。

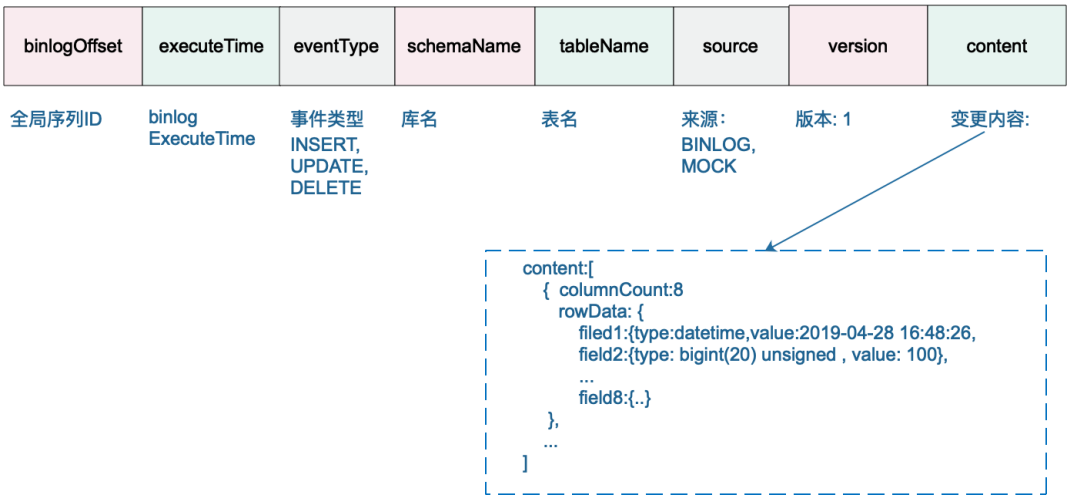
参考 producer 参数控制：

max.in.flight.requests.per.connection=1
retries=0
acks=all

topic level 的配置：

topic partition 3 副本，且
min.insync.replicas=2

从保证数据的顺序性、容灾等方面考虑，我们设计了一个轻量级的 SimpleBinlog 消息格式：



- binlogOffset：全局序列 ID，由${timestamp}${seq} 组成，该字段用于全局排序，方便 Hive 做 row_number 取出最新镜像，其中 seq 是同一个时间戳下自增的数字，长度为 6。
- executeTime：binlog 的执行时间。
- eventType：事件类型：INSERT，UPDATE，DELETE。
- schemaName：库名，在后续的 spark-streaming，mirror 处理时，可以根据分库的规则，只提取出前缀，比如(ordercenter_001 → ordercenter) 以屏蔽分库问题。
- tableName：表名，在后续的 spark-streaming，mirror 处理时，可以根据分表规则，只提取出前缀，比如(orderinfo_001 → orderinfo) 以屏蔽分表问题。
- source：用于区分 simple binlog 的来源，实时采集的 binlog 为 BINLOG， 重放的历史数据为 MOCK 。
- version：版本
- content：本次变更的内容，INSERT，UPDATE 取 afterColumnList，DELETE 取 beforeColumnList。

金融当前部署了 4 组 canal 集群，每组 2 个物理机节点，跨机房 DR 部署，承担了数百个 mysql instance binlog 采集工作。Canal server 自带的性能监控基于 Prometheus 实现，我们通过实现 PrometheusScraper 主动拉取核心指标，推送到集团内部的 Watcher 监控系统上，配置相关报警，其中各 mysql instance 的 binlog 采集延迟是全链路监控的重要指标。

系统上线初期遇到过 canal-server instance 脑裂的问题，具体场景是 active instance 所在的 canal-server ，因网络问题与 zookeeper 的链接超时，这时候 standby instance 会抢占创建临时节点，成为新的 active；也就出现了 2 个 active 同时采集并推送 binlog 的情况。解决办法是 active instance 与 zookeeper 链接超时后，立即自 kill，再次发起下一轮抢占。

3.2 历史数据重放

有两个场景需要我们采集历史数据：

1）首次做 mysql-hive 镜像 ，需要从 mysql 加载历史数据；
2）系统故障（丢数等极端情况），需要从 mysql 恢复数据。

有两种方案：

1）从 mysql 批量拉取历史数据，上传到 HDFS 。需要考虑批量拉取的数据与 binlog 采集产出的 mysql-hive 镜像的格式差异，比如去重主键的选择，排序字段的选择等问题。

2）流式方式， 批量从 mysql 拉取历史数据，转换为 simple binlog 消息流写入 kafka，同实时采集的 simple binlog 流复用后续的处理流程。在合并产生 mysql-hive 镜像表时，需要确保这部分数据不会覆盖实时采集的 simple binlog 数据。

我们选用了更简单易维护的方案 2，并开发了一个 binlog-mock 服务，可以根据用户给出的库、表（前缀）以及条件，按批次（比如每次 select 10000 行）从 mysql 查询数据，组装成 simple_binlog 消息发送 kafka。

对于 mock 的历史数据，需要注意：

1）保证不覆盖后续实时采集的 binlog：simple binlog 消息里 binlogOffset 字段用于全局排序，它由${timestamp}+${seq}组成，mock 的这部分数据 timestamp 为发起 SQL 查询的时间戳向前移 5 分钟，seq 为 000000;

2）落到哪个分区：我们根据 binlog 事件时间(executeTime) 判断数据所属哪个 dt 分区，mock 的这部分数据 executeTime 为用户指定的一个值，默认为${yesterday}。

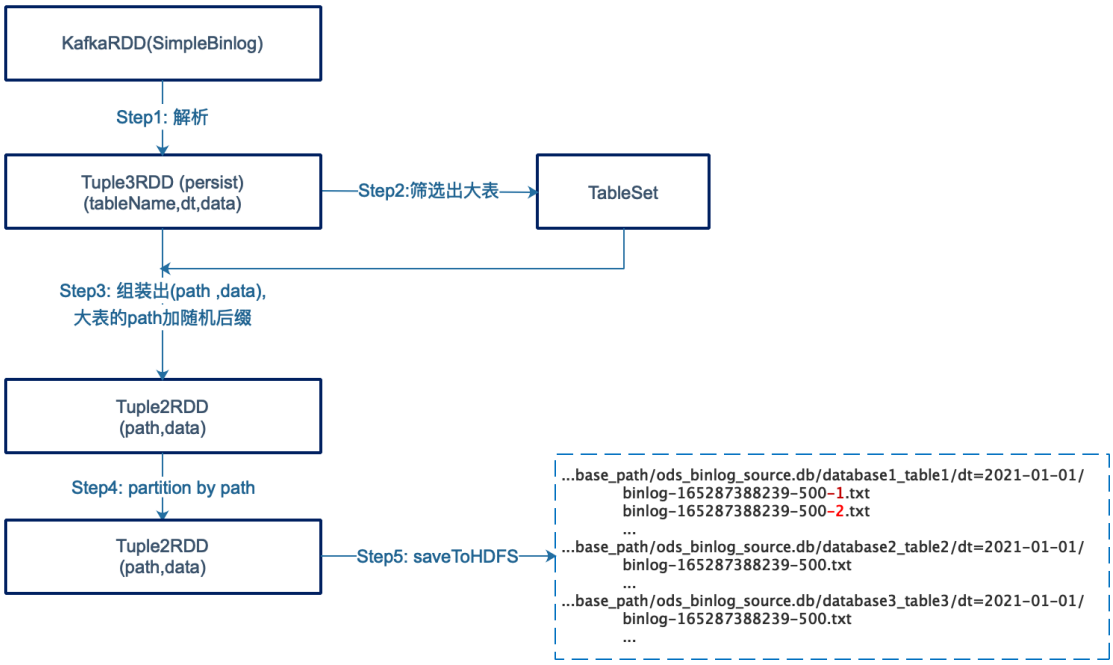
3.3 Write2HDFS

我们采用 spark-streaming 将 kafka 消息持久化到 HDFS，每 5 分钟一个批次，一个批次的数据处理完成（持久化到 HDFS）后再提交 consumer offset，保证消息被 at-least-once 处理；同时也考虑了分库分表问题、数据倾斜问题：

屏蔽分库分表：以订单表为例，mysql 数据存储在 ordercenter_00 … ordercenter_99 100 个库，每个库下面又有 orderinfo_00…orderinfo_99 100 张表，库前缀 schemaNamePrefix=ordercenter, 表前缀 tableNamePrefix=orderinfo，统一映射到 tableName=${schemaNamePrefix}_${tableNamePrefix}里; 根据 binlog executeTime 字段生成对应的分区 dt，确保同一个库表同一天的数据落到同一个分区目录里：base_path/ods_binlog_source.db/${database_prefix}_${table_prefix}/dt={binlogDt}/binlog-{timestamp}-{rdd.id}

防止数据倾斜: 系统上线初期经常出现数据倾斜问题，排查发现某些时间段个别表由于业务跑批等产生的 binlog 量特别大，一张表一个批次的数据需要写入同一个 HDFS 文件，单个 HDFS 文件的写入速度成为瓶颈。因此增加了一个环节（Step2），过滤出当前批次里的“大表''，将这些大表的数据分散写入多个 HDFS 文件里。

base_path/ods_binlog_source.db/${database_prefix}_${table_prefix}/dt={binlogDt}/binlog-{timestamp}-{rdd.id}-[${randomInt}]



3.4 生成镜像

3.4.1 数据就绪检查

spark-streaming 作业每 5 分钟一个批次将 kafka simple_binlog 消息持久化到 HDFS，merge 任务是每天执行一次。每天 0 点 15 分，开始进行数据就绪检查。我们对消息的全链路进行了监控，包括 binlog 采集延迟 t1、kafka 同步延迟 t2 、spark-streaming consumer 延迟 t3。假设当前时间为凌晨 0 点 30 分，设为 t4，若 t4>(t1+t2+t3) 说明 T-1 日数据已全部落入 HDFS，即可执行下游的 ETL 作业（merge）。

3.4.2 Merge

HDFS 上的 simple binlog 数据就绪后，下一步就是对相应 MySQL 业务表数据进行还原。以下是 Merge 的执行流程，步骤如下:

1）加载 T-1 分区的 simple binlog 数据

数据就绪检查通过后，通过 MSCK REPAIR PARTITION 加载 T-1 分区的 simple_binlog 数据，注意：这个表是原始的 simple binlog 数据，并未平铺具体 mysql 表的字段。如果是首次做 mysql-hive 镜像，历史数据重放的 simple binlog 也会落入 T-1 分区。

2）检查 Schema，并抽取 T-1 增量

请求 mirror 后台，获取最新的 mysql schema，如果发生了变更则更新 mysql-hive 镜像表(snap)，让下游无感知；同时根据 mysql schema 的 field 列表 、以及"hive 主键" 等配置信息，从上述 simple_binlog 分区抽取出 mysql 表的 T-1 日明细数据 (delta)。

3）判断业务库是否发生了归档操作，以决定后续合并时是否忽略 DELETE 事件。

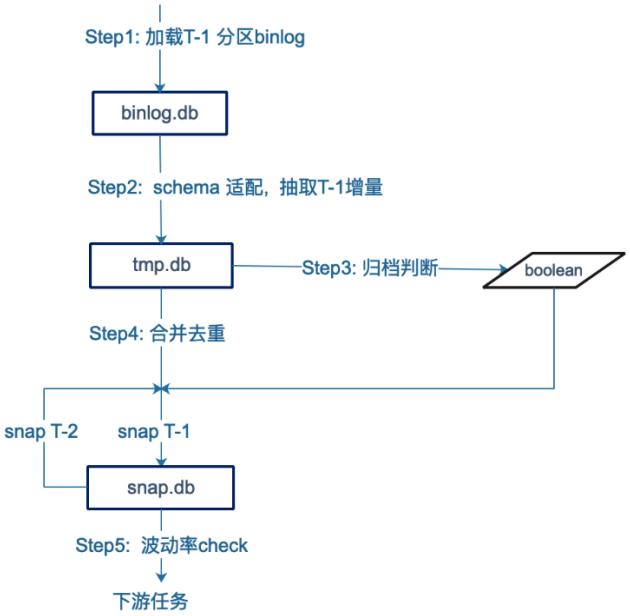
业务 DELETE 数据有 2 种情况：业务修单等引起的正常 DELETE，需要同步变更到 Hive；业务库归档历史数据产生的 DELETE，这类 DELETE 操作需要忽略掉。

系统上线初期，我们等待业务或 DBA 通知，然后手工处理，比较繁琐，很多时候会有通知不到位的情况，导致 Hive 数据缺失历史数据。为了解决这个问题，在 Merge 之前进行程自动判断，参考规则如下：

a）业务归档通常是大批量的 DELETE(百万+)，因此可以设置一个阈值，比如 500W 或日增量的 7 倍。

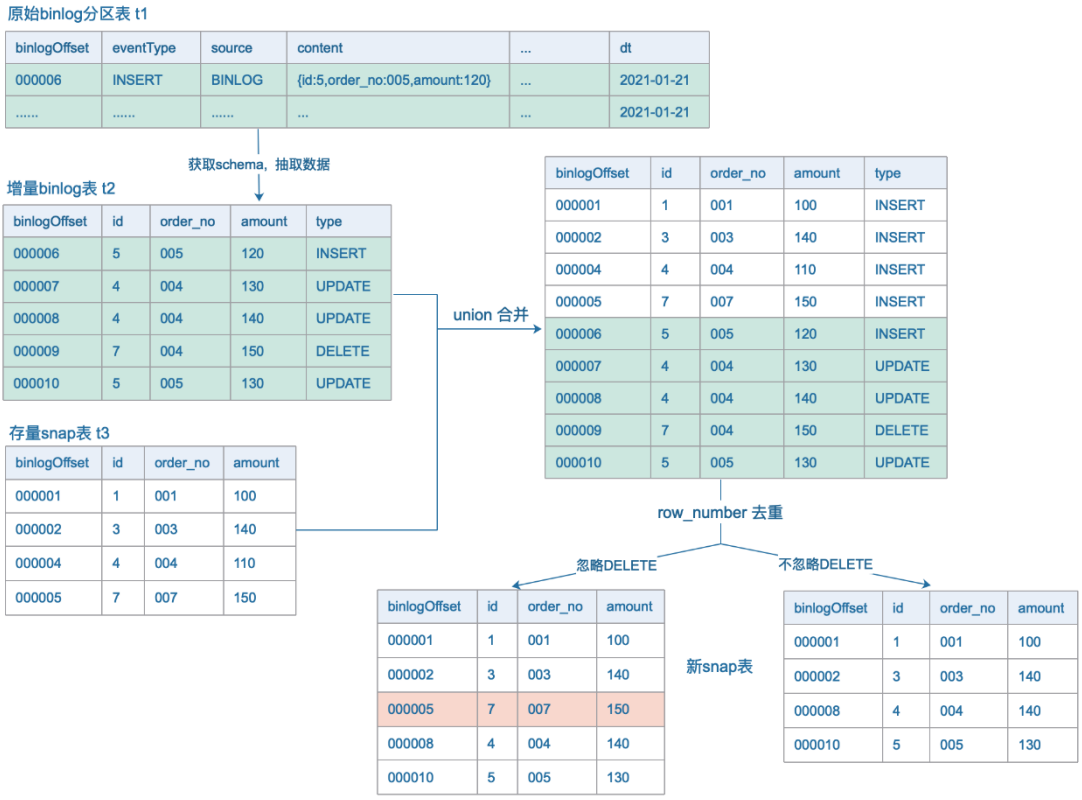
b）业务归档的时间段通常比较久，比如设置阈值为 30 天。如果满足了条件 1，且删除的这些数据在 30 天以前，则属于归档产生的 DELETE。

4）对增量数据（delta）和当前快照（snap T-2）进行合并去重，得到最新 snap T-1。



下面通过一个例子说明 merge 的过程，假设订单 order 表共有 id,order_no,amount 三个字段，id 是全局唯一建; snap 表 t3 是 mysql-hive 镜像，merge 过程如图展示。

1）加载目标（dt=T-1）分区里的 simple binlog 数据，表格式如 t1；
2）请求 mirror 后台获取 mysql 的最新 schema，从 t1 抽取数据到临时表 t2；
3）snap 表 t3 与 mysql schema 进行适配（本例无变更）；
4）对增量表 t2、存量 snap t3 进行 union（对 t3 自动增加 type 列，值为 INSERT），得到临时表 t4；
5）对 t4 表按唯一键 id 进行 row_number，分组按 binlogOffset 降序排序，序号为 1 的即为最新数据。



3.4.3 check

在数据 merge 完成后，为了保证 mysql-hive 镜像表中数据准确性，会对 hive 表和 mysql 表进行字段和数据量对比，做好最后一道防线。我们在配置 mysql-hive 镜像时，会指定一个检查条件，通常是按 createTime 字段对比 7 天的数据；mirror 后台每天凌晨会预先从 mysql 统计出过去 7 日增量，离线任务通过脚本（http）获取上述数据，和 snap 表进行校验。实践中遇到一些问题：

1）T-1 的 binlog 落在 T 分区的情况

check 服务根据 createTime 生成查询条件去 check mysql 和 Hive 数据，由于业务 sql 里的

createTime 和 binlog executeTime 不一致，分别为凌晨时刻的前后 1 秒，会导致 Hive 里漏掉这条数据，这种情况可以通过一起加载 T 日分区的 binlog 数据，重新 merge。

2）业务表迁移，原表停止更新，虽然 mysql 和 hive 数据量一致，但已经不符合要求了，这种情况可以通过波动率发现。

3.5 其他

在实践中，可根据需要在 binlog 采集以及后续的消息流里引入一些数据治理工作。比如：

1）明文检测：binlog 采集环节对核心库表数据做实时明文检测，可以避免敏感数据流入数仓；

2）标准化：一些字段的标准化操作，比如 id 映射、不同密文的映射；

3）元数据：mysql→hive 镜像是数仓 ODS 的核心，可以根据采集配置信息，实现二者映射关系的双向检索，便于数仓溯源。这块是金融元数据管理的重要组成部分。

通过消费 binlog 实现 mysql 到实时数仓（kudu、es）、在线缓存（redis）的镜像逻辑相对简单，限于篇幅，本文不再赘述。
—

四、总结与展望

金融基于 binlog 的数据基础层构建方案，顺利完成了预期目标：

1）金融数据中心建设（ODS 层）：数千张 mysql 表到携程 DC 的镜像，全部 T+1 1:30 产出；

2）金融实时数仓建设：金融核心 mysql 表到 kudu 的镜像，支持实时分析、分表合并查询等偏实时的运营场景；

3）金融在线缓存服务：异地多活，缓存近 1000G 业务数据；支撑整个消金入口、风控业务近 100W/min 的请求。

该方案已经成为金融在线和离线服务的基石，并在持续扩充使用场景。未来会在自动化配置（整合 mirror-admin 和 canal-admin，实现一键构造）、智能运维（数据 check 异常的识别与恢复）、元数据管理方面做更多的投入。

本文介绍了携程金融构建大数据基础层的技术方案，着重介绍了 binlog 采集和 mysql-hive 镜像的设计，以及实践中遇到的一些问题及解决办法。希望能给大家带来一些参考价值，也欢迎大家一起来交流。

实时数据聚合怎么破

【作者简介】数据猩猩，携程数据分析总监，关注分布式数据存储和实时数据分析。

实时数据分析一直是个热门话题，需要实时数据分析的场景也越来越多，如金融支付中的风控，基础运维中的监控告警，实时大盘之外，AI 模型也需要消费更为实时的聚合结果来达到很好的预测效果。

实时数据分析如果讲的更加具体些，基本上会牵涉到数据聚合分析。

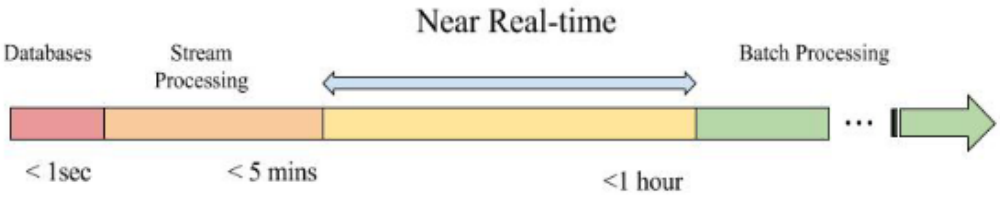
数据聚合分析在实时场景下，面临的新问题是什么，要解决的很好，大致有哪些方面的思路和框架可供使用，本文尝试做一下分析和厘清。

在实时数据分析场景下，最大的制约因素是时间，时间一变动，所要处理的源头数据会发生改变，处理的结果自然也会因此而不同。在此背景下，引申出来的三大子问题就是:

- 通过何种机制观察到变化的数据
- 通过何种方式能最有效的处理变化数据，将结果并入到原先的聚合分析结果中
- 分析后的数据如何让使用方及时感知并获取

可以说，数据新鲜性和处理及时性是实时数据处理中的一对基本矛盾。

另外实时是一个相对的概念，在不同场景下对应的时延也差异很大，借用 Uber 给出的定义，大体来区分一下实时处理所能接受的时延范围。



一、数据新鲜性

为简单起见，把数据分成两大类，一类是关键的交易性数据，以存储在关系型数据库为主，另一类是日志型数据，以存储在日志型消息队列（如 kafka）为主。

第二类数据，消费端到感知到最新的变化数据，采用内嵌的 pull 机制，比较容易实现，同时日志类数据，绝大部分是 append-only，不涉及到删改，无论是采用 ClickHouse 还是使用 TimeScaleDB 都可以达到很好的实时聚合效果，这里就不再赘述。

针对第一类存储在数据库中的数据，要想实时感知到变化的数据（这里的变化包含有增/删/改三种操作类型），有两种打法。

打法一：基于时间戳方式的数据同步，假设在表设计时，每张表中都有 datachange_lasttime 字段表示最近一次操作发生的时间，同步程序会定期扫描目标表，把 datachange_lasttime 不小于上次同步时间的数据拉出进行同步。

这种处理方式的主要缺点是无法感知到数据删除操作，为了规避这个不足，可以采用逻辑删除的表设计方式。数据删除并不是采取物理删除，只是修改表示数据已经删除的列中的值标记为删除或无效。使用这种方法虽然让同步程序可以感知到删除操作，但额外的成本是让应用程序在删除和查询时，操作语句和逻辑都变得复杂，降低了数据库的可维护性。

打法一的变种是基于触发器方式，把变化过的数据推送给同步程序。这种方式的成本，一方面是需要设计实现触发器，另一方面是了降低了 insert/update/delete 操作的性能， 提升了时延，降低了吞吐量。

打法二：基于 CDC（Change Data Capture）的方式进行增量数据同步，这种方式对数据库设计的侵入性最小，性能影响也最低，同时可以获得丰富的开源组件支持，如 Cannal 对 MySQL 有很好支持，Debezium 对 PostgreSQL 有支持。利用这些同步组件，把变化数据写入到 Kafka，然后供后续实时数据分析进一步处理。

二、数据关联

新鲜数据在获取到之后，第一步常见操作是进行数据补全（Data Enrichment），数据补全自然涉及到多表之间的关联。这里有一个痛点，要关联的数据并不一定也会在增量数据中，如机票订单数据状态发生变化，要找到变化过订单涉及到的航段信息。由于订单信息和航段信息是两张不同的表维护，如果只是拿增量数据进行关联，那么有可能找不到航段信息。这是一个典型的实时数据和历史数据关联的例子。

解决实时数据和历史数据关联一种非常容易想到的思路就是当实时数据到达的时候，去和数据库中的历史数据进行关联，这种做法一是加大了数据库的访问，导致数据库负担增加，另一方面是关联的时延会大大加长。为了让历史数据迅速可达，自然想到添加缓存，缓存的引入固然可以减少关联处理时延，但容易引起缓存数据和数据库中的数据不一致问题，另外缓存容量不易估算，成本增加。

有没有别的套路可以尝试？这个必须要有。

可以在数据库侧先把数据进行补全，利用行转列的方式，形成一张宽表，实现数据自完备，宽表的变化内容，利用 CDC 机制，让外界实时感知。

三、计算及时性

在解决好数据变化实时感知和数据完备两个问题之后，进入最关键一环，数据聚合分析。为了达到结果准确和处理及时之间的平衡，有两大解决方法：一为全量，一为增量。

3.1 全量计算（1m<时延<5m）

全量计算以时间代价，对变化过的数据进行全量分析，分析结果有最高的准确性和可靠性。成本是花费较长的计算时间和消耗较多的计算资源。可以使用的分析引擎或计算框架有 Apache Spark 和 Apache Flink。

全量数据容量一般会比较大，为了节约存储，同时为了方便数据过滤和减少不必要的网络传输，大多会使用列式存储， 列式存储使用较多的当属 Parquet 和 ORC。

列式存储最大的不足是无法进行删/改操作，为了支持删改，一般会把列式存储和行式存储相结合。最近时间内变化的数据采用行式存储如 avro 格式，然后定期合并成列式存储。非常成功和红火的 Apache Hudi 和 Delta IO 就是基于这种思路。

3.2 增量计算

假设当前处理的时间窗口中有 10 万条记录，因为其中不到 100 条的记录发生变化，而对所有记录的聚合指标进行计算重演，显然不是非常合理，那么有没有可能只对增量数据导致的变化聚合指标进行重算。答案是肯定的，或者说在部分场景下，是可以实现的。

让我们把增量计算分成几种不同情况：

1）增量数据会添加新的聚合记录，对原有计算结果无影响
2）增量数据会添加新的聚合记录，并导致原有计算结果部分失效
3）增量数据不添加新的聚合记录，但导致原有计算结果全部失效

第 1、2 两种情况下，增量计算会带来实时性上的收益，第三种不会，因为所有指标均被破坏，都需要重演，已经褪化成全量计算。

增量处理模型除了 Apache Flink 之外，非常著名的还有 Microsoft 提出的 Naiad 模型，后者更为高效。由于后者只提供了非常底层的调用 API，在生态建设方面远不如 Apache Flink，但其思想深刻影响了 TensorFlow 等框架的设计和实现，等有时间再详细介绍一下 Naiad。

上面讨论的全量也好，增量也罢，都是把数据从数据库拉出来再进行计算，那么有没有可能在数据库内部实现增量计算的可能？

Oracle 在 12.x 版本中提供物理视图（materialized view）的自动刷新机制，这意味着用户可以把实时聚合逻辑定义在物理视图中，然后每当有数据更新，视图会被自动更新。既然 Oracle 有，那么在开源的世界里一定会有对应的东西出现，最起码会有相应的影子在浮现，这个影子就是 PostgreSQL IVM。

PostgreSQL IVM 使用到 Transition Table 这个概念，在触发器中，用户可以看到变化前和变化后的数据，从而计算出变更的内容，利用这些 Delta 数据，进行刷新预先定义好的物理视图。

四、计算触发机制

- 定时触发
- trigger for every new element

计算成本比较

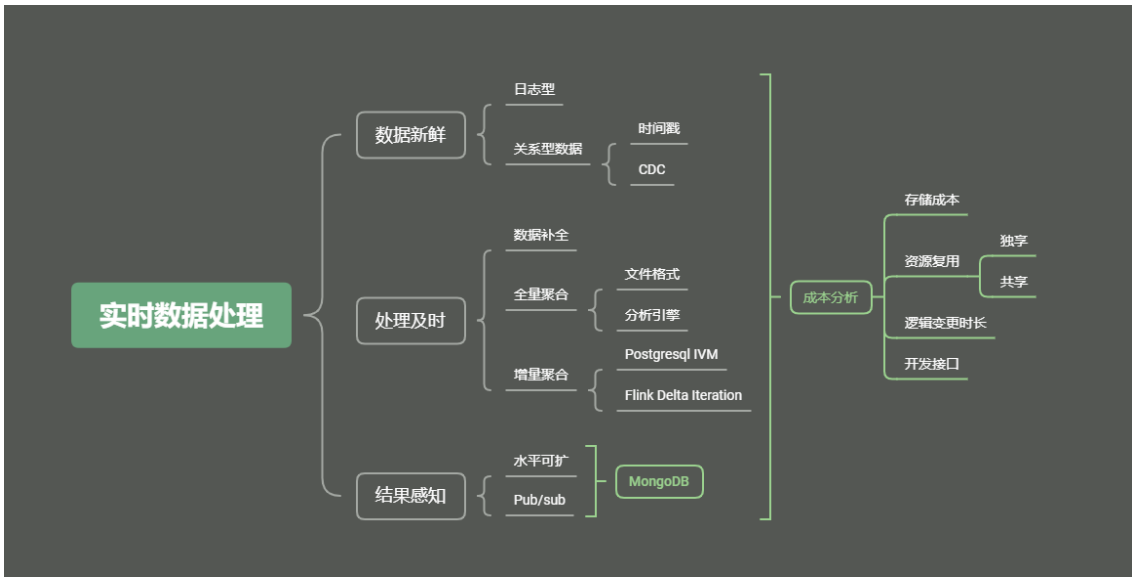
	时延	资源复用率	开发难度	逻辑变更实时型	SQL兼容度	普及程度
全量	高	低	编程实现(较高)	低	高	高
增量	低	高	SQL实现(低)	高	低	低

五、聚合结果实时可见

聚合结果的存储要支持 upsert 语义，聚合结果的消费者实时感知到，同时聚合结果的存储要有水平可扩性。结合这三个要求，比较推荐使用 NoSQL 来进行指标的存储，具体可以使用 MongoDB。

六、小结

本文尝试对实时数据聚合分析中涉及到的问题和常见思路进行梳理，文中定有不少疏漏，不足之处希望读者批评指正。



携程平台化常态化数据治理之路

【作者简介】瑞强，携程高级大数据开发工程师，负责集团客户数据平台、数据资产管理平台的开发和数据治理的推进。

一、背景

数据的重要性不言而喻。每个数据工程师每天会产生大量数据，但这些数据占用的成本、带来的价值、质量如何，以及在保证安全的前提下是否能够更高效地使用，是每个公司在大数据发展到一定阶段后都会遇到的问题。而携程由于涉及的业务线多，数仓团队多，数据安全高效地流通也是一个治理难点。

二、治理思路

何为数据治理？数据治理和众多新兴学科一样，也有很多种定义。IBM 认为，数据治理是根据企业的数据管控政策，利用组织人员、流程和技术的相互协作，使企业能够将“数据作为资产”来管理和应用。根据伯森和杜波夫的定义，数据治理是一个关注于管理信息的质量、一致性、可用性、安全性和可得性的过程。这个过程与数据的拥有和管理的职责紧密相关。

通常认为，数据治理是围绕数据资产展开的一系列工作，以服务组织各层决策为目标，是数据管理技术、过程、标准和政策的集合。

综上，数据治理离不开数据资产的沉淀，只有对数据有宏观地把控、明细地探究，才能贴合数据特性进行治理。所以要进行集团层面的数据治理，就需要集团层面的数据资产平台。携程数据资产管理平台（大禹）应运而生。携程数据治理体系的目标是可以让每一位数据生产者对各自拥有的数据进行常态化治理。而目前阶段数据治理的核心目标就是提升数据价值、提高数据质量、促进数据流通。

- 数据价值：首先要治理的就是低价值甚至无价值的数据，例如长期无访问、生命周期过长的数据。其次计算资源消耗较多的数据要进行归因分析，针对性优化。
- 数据质量：完善表的元数据信息包括责任人、数仓分层、主题、重要等级和敏感等级，配置数据质量监控，重点治理无人维护的数据。
- 数据流通：保障安全的前提下，提高权限审批效率，促进数据流转。

三、方案实施

3.1 元数据建设

数据治理的首要工作是搭建元数据数仓。元数据一般分为四类：技术元数据、操作元数据、管理元数据和业务元数据，分别描述了数据的物理化、处理过程、管理过程及数据定义。

- 技术元数据：存储相关数据，包括表的元数据、字段元数据等。
- 操作元数据：ETL 相关数据，包括调度元数据、执行元数据、调度之间的血缘元数据等。

- 管理元数据：包括管理者信息、监控日志、管理日志、管理成效等。
- 业务元数据：包括数据标准、数据质量、数据指标、数据字典、数据代码、数据安全等。

现阶段最为丰富的数据是技术元数据和操作元数据，有了这些元数据就可以对计算/存储成本、元数据完整度、数据质量监控的覆盖率/通过率、临时表、无人维护表等进行统计分析，进而推进相关专项治理。

3.2 专项治理

3.2.1 成本治理

大多数的数据工程师关注的是需求交付，对存储、计算成本认识不足。目前集团大数据集群计算成本和存储成本比例是 4：6，通过初步治理，可节约年成本数千万元。在大禹（数据资产管理平台）上可以直观地看到每个员工拥有的 Hive 表数、日均存储成本、日均计算成本和在完成数据治理后预计节省的年成本。

3.2.1.1 计算成本

计算成本主要来自于 CPU 资源的消耗，根据每个调度任务对 CPU 核数和时间的占用情况估算出成本。CPU 的运行成本根据集群的运营情况，计为 10 元/1M VCS（每个 CPU 核占用的秒数）。

计算资源主要消耗在 ETL 调度和 Adhoc 查询，由此我们对典型低效 SQL 进行了归因分析。选择部分 BU 作为试点，针对单次消耗大于 10 元的高消耗调度进行优化。虽然集团内这些高消耗调度占比 1%，但是占据了千万量级的年计算成本。

问题归因	解决方案	调度占比
不再使用的任务	下线	45%
全量查询或跨度过大	分区裁剪	33%
SQL撰写问题	技术参数优化	5%
关联表过多	业务逻辑优化	3%
重复计算	逻辑合并	3%
一次性作业	下线	3%
程序逻辑问题	代码优化	3%

表 1：高消耗问题归因及解决方案

对于 Adhoc 查询而言，1%超过 30 元/次，13%超过 0.3 元/次。仅这 14%的查询就占据了超一半的算力成本。除了逻辑、业务、分区层面的优化，技术参数优化也进行全面推广。例如常见的几类 MR 优化：

1）合并小文件：配置 Map 输入合并、Map/Reduce 输出合并。

2）合理控制 reducer 数量

参数 1：hive.exec.reducers.bytes.per.reducer（默认 1G）

参数 2：hive.exec.reducers.max（默认为 999）
reducer 的计算公式为：min（参数 2，总输入数据量/参数 1），也可以通过设置 mapred.reduce.tasks 直接控制 reducer 个数。

3）使用相同的连接键：当对 3 张或更多表进行 join 时，如果 on 条件使用相同字段，会合并为一个 MapReduce Job。

4）SMB（sort merge bucket join）：用于两张大表进行 join，但需要预先给每张大表基于 join 的字段建立桶。

```
set hive.enforce.bucketing = true; --启用桶表
set hive.optimize.bucketmapjoin = true;
set hive.optimize.bucketmapjoin.sortedmerge = true;
set hive.input.format=org.apache.hadoop.hive.ql.io.BucketizedHiveInputFormat;
```

还有将数据倾斜的异常值打散或单独处理、启用压缩、矢量化执行等。

3.2.1.2 存储成本

存储成本重点治理长期无访问数据和用户行为数据（UBT），其次统一表存储格式为 ORC，采用冷热存储、EC 存储，最后清理重复的大文件和业务不再需要的数据。通过这些治理手段，新增存储需求缩减 50%，占总存储的 20%。

1）近 30 天无访问表的成本占据总存储的 20%，其中 99%是临时表。这些无访问表由 BU 内部进行确认清理，一些日志表或者集团的用户行为数据等需要长期保存的会加入白名单，没有加入白名单的表会自动删除。

2）用户行为数据之前全链路保存了三年的历史，通过逐渐缩短整个流程数据的生命周期达到缩减成本的目的。为了做到治理过程中下游无感知，将原表改为备份表再创建一个原表表名的视图，逐渐缩短视图可读的时间范围，待下游使用无异常之后可将备份表的生命周期缩短。这个优化节省了大量存储成本。

3) 由于历史遗留问题, 之前表的数据格式未完全统一。RCFile 占比 13.46%, Avro 占比 1.99%, 压缩表占比 5.4%，非结构化数据占比 24.15%。所以将这些表转化为 ORC 格式，同时提升计算效率和存储能力。

4）将不常用但需要保存的数据进行冷存储。冷存储的成本为热存储的 40%，使用 EC 技术可进一步压缩到 20%。但是冷存储会影响查询的性能，需要根据数据的使用场景综合考虑。这个优化也节省了不小的存储成本。

3.2.2 质量规范

首先完善表的元数据信息，配置数据质量监控（DQC），其次重点治理无人维护的表和临时表。

1）完善元数据信息：

表的元数据信息

目前统计到的表和字段的元数据信息见上图，从中选取了 12 个重要指标作为完整性维度的统计，如下图。历史表的完整性也会按照设定的截止时间进行批量补充，同时新建正式表严格按照完整性的规范建立，否则无法创建。

表的更新频率是否设置	X	表的更新频率未设置
表的状态是否设置	✓	
表的数仓分层是否设置	X	未设置分层
表的生命周期是否设置	✓	
表时间区分的格式是否设置	1/1	设置了的字段数：1, 全部的分区字段：1
表的数据类型是否设置	X	表的数据类型未设置
表的主键是否设置	X	表的主键未设置
表的基线是否设置	✓	非高级别的表可以不设置基线
表的主题是否设置	X	未设置分层
表的敏感等级是否设置	✓	
表的责任人未离职	✓	
表的重要等级是否设置	✓	

2）配置 DQC：原则上每个正式使用的表都需要配置 DQC 校验，比如保证调度完成后的数据要大于一定数量，今天和昨天的数据波动要在一定的范围，某些情境下需要主键唯一，或者自定义校验规则。校验规则分为强规则、弱规则。强规则会熔断下游，防止错误数据影响到下游的使用，对生产造成不可逆的影响。弱规则会触发邮件警告。

3）无人维护表治理：因为离职转岗等原因，有些表的责任人缺失，给下游使用造成了一些困难。我们首先将无人维护表的明细开放给各 BU，推动 BU 补全责任人信息。后期开发了资源转移系统，离职或转岗前会将责任人名下的资源进行一键转移。

4）临时表治理：临时表数量占总表数量的比例较高，需要进行治理。我们明确了临时表的使用规范，只是作为临时使用，七天后自动删除。可以用来进行探索性分析、排障，但是不可用于报表依赖、调度依赖、数据传输。调度任务中产生的中间表需要在任务结束后删除。

3.2.3 数据流通

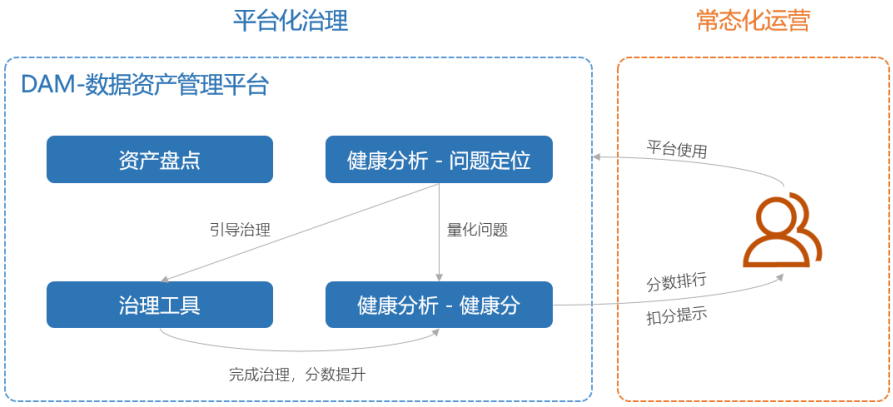
数据流通主要关注的是共享数据。有两个来源：跨 BU 合作的项目，中台提供的服务于全业务的数据比如：统一订单数据等。重点治理的是跨 BU 合作的项目中由于组织架构的改变、项目组变动、数据源变更等原因产生的权限外溢。

现阶段的治理考虑两个方面：既要增加 BU 之间的数据流通性、提高数据价值，又要及时治理权限外溢、敏感数据泄露。易用性与安全性之间的平衡存在一定挑战。为此我们上线级联审批功能。对于设置级联审批的表，其下游表的权限审批需要上游表 owner 共同参与，进一步加强了数据安全性。同时上线了基于密级的差异化审批流程。对于高密表从严把控，低密表则尽量简化审批流程，方便数据快速流通。

四、平台化与常态化

数据资产管理平台目前有三大功能模块，分别是资产盘点、治理工具、健康分析。三个模块

的关系如下图所示：

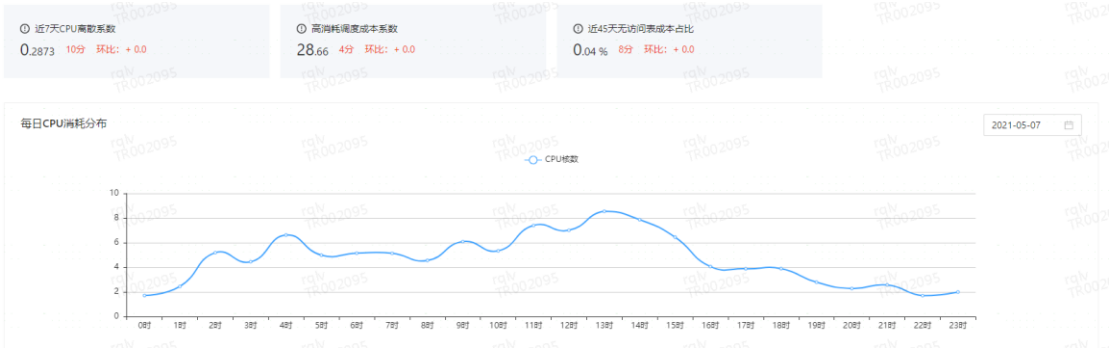


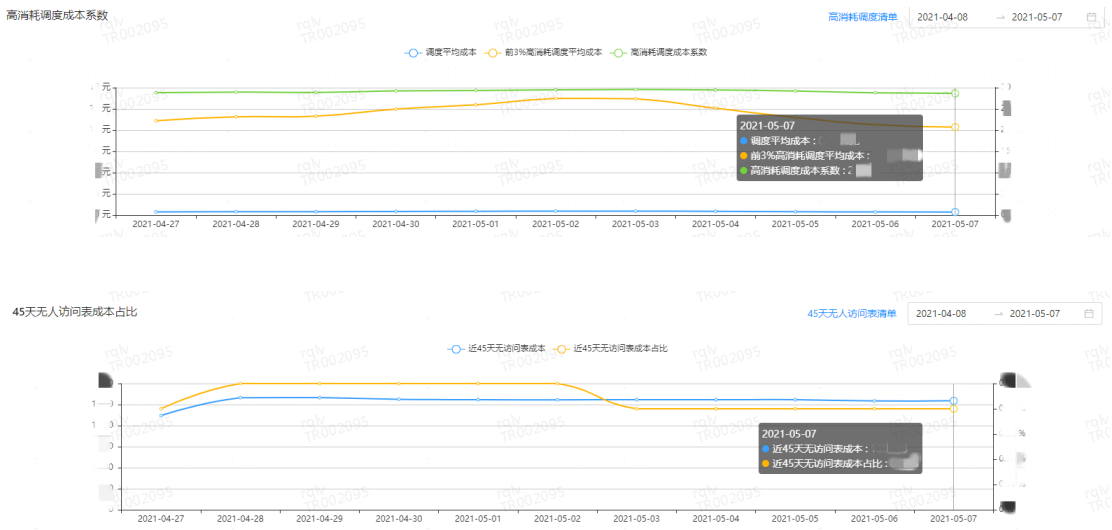
其中资产盘点主要是资产数据看板，包含集团、BU 组织和个人的资产概览，成本分析，质量和数据共享相关指标。

第二个模块是数据治理。数据属主可以在“我的工作台”对有问题的数据进行便捷地治理。需要治理的数据都会以问题标签的形式进行分类展示。

第三个模块是数据健康分析。分为资源利用、管理规范、成果交付、数据安全四个维度对数据的健康状态进行统计。BU 内部想要提质降本、提高开发效率，健康分会是一个最直观的指标。如果有 BU 疏于数据治理，那么相应的健康分和 BU 之间的排名就会下降，以此来促进常态化治理。下图为数据健康分总览。

资源利用：考察近 7 天 CPU 离散系数、高消耗调度成本系数及近 45 天无访问表成本占比。



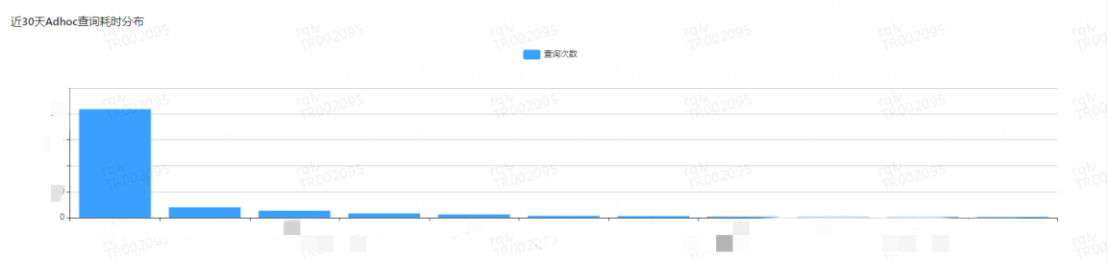


资源利用健康分

管理规范：考察表的元数据（数仓分层、责任人、重要等级、基线、敏感等级和主题等）完整性。

管理规范健康分

成果交付：考察失败调度占比和查询时长。



成果交付健康分

数据安全：重点考察对敏感数据的使用是否存在风险。

五、总结

数据治理是一个比较宽泛的概念，每个公司需要治理的数据不一样，并且同一公司不同的发展阶段治理的内容也不一样。需要决策层根据数据体系发展的阶段确定本阶段治理的核心目标，以此来展开治理。

现阶段我们针对数据的成本、质量、流通三个维度的重点问题进行了治理。下阶段将会有更高的治理要求。同时由于数据在不断产生，治理也不是一劳永逸的，所以借助平台让每个数据生产者可以便捷地进行常态化治理是必经之路。

携程酒店推荐模型优化

【作者简介】Yorkey，携程高级算法专家，主要从事大规模分布式推荐系统设计和算法研发工作。

一、背景

当用户在线上浏览酒店时，作为旅行平台，如何挑选更合适的酒店推荐给用户，降低其选择的费力度，是需要考虑的一个问题。在携程 APP 中，一般会触发多种场景。在 Figure 1 中，我们列举了几种典型的场景：欢迎度排序，智能排序和搜索补偿推荐。

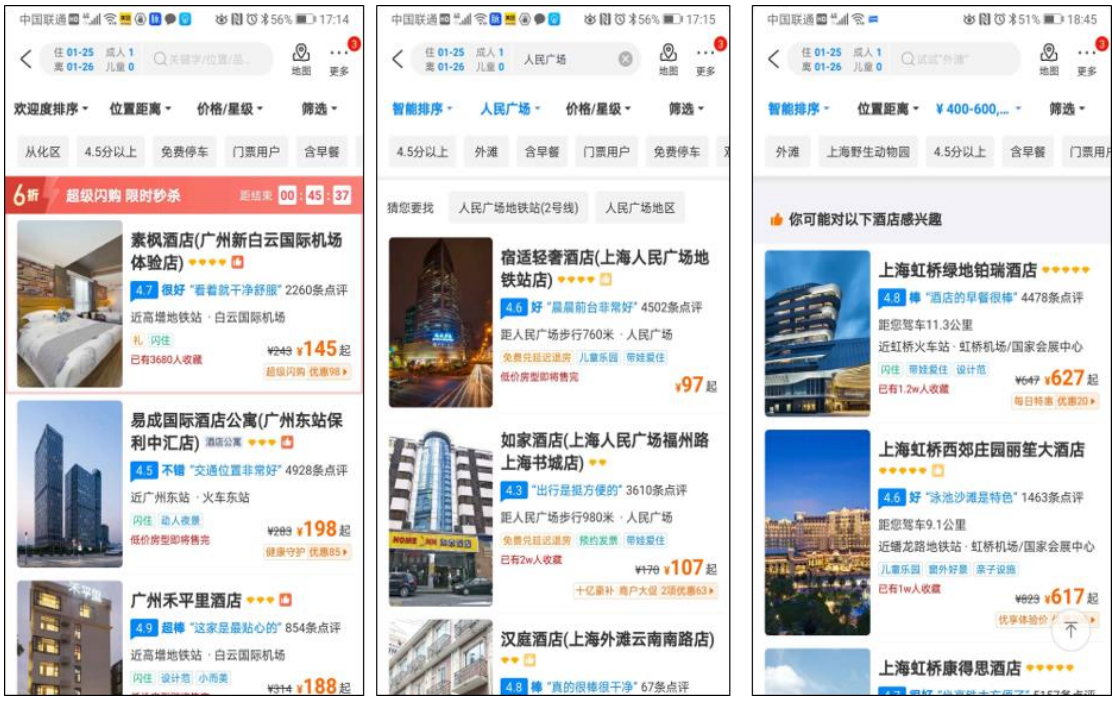


Figure1：用户触发的场景

欢迎度排序：在异地场景下，当用户没有明确表达自己需求的情况下，默认按照欢迎度排序展示酒店；

智能排序：在同城或者当用户在浏览上述排序结果的过程中，发现自己的兴趣并没有得到满足的时候，往往会添加筛选条件。比如添加地标选项，从而获得该地标附近的相关酒店。相比欢迎度排序，用户表达了更多的个性化需求。

搜索补偿推荐排序：如果用户在当前搜索的展示结果中，还是没有找到满足自己需要的酒店，就会不断往下翻页查看更多的搜索结果。而当满足搜索限制条件的酒店数目不足时，会触底。这个时候就会触发搜索补偿推荐。前面两个场景都没有满足用户需求，所以在搜索补偿场景中，我们需要深入挖掘用户历史行为，提供更加个性化的推荐内容。

本文将主要介绍我们在补偿推荐场景中所做的算法优化工作。包含模型迭代、模型迭代过程

中遇到的技术需求以及针对技术需求所做的一些基建等。

二、推荐模型的迭代

在酒店推荐的场景中，我们需要把满足用户需求的产品优先曝光给用户，减少其使用产品的费力度。我们以用户在酒店的 TOP 点击（编者注：可以简单理解为用户点击排在 TOP 位置酒店的概率，TOP 点击命中率越高，用户体验越好）和转化命中率（CR）作为费力度指标；CR 优化问题被建模成二分类问题，离线采取 AUC 和 NDCG 作为模型效果评估标准。

分类问题的本质就是要找到一条决策边界函数 f(x)，来把正样本（比如成单）和负样本（比如没有成单）数据分开。这里，x 就是模型特征，好的特征能更好地表征正负样本的差异，让函数学习事半功倍；学习函数 f(x)的过程就是建模过程。在本节中，我们分别介绍下我们在特征和建模方面所尝试的工作。

2.1 特征

推荐特征使用的特征可以分为：用户侧特征、物品侧特征以及用户和物品的交互特征。从特征的数值特性上，又可以分成：连续值特征和离散值特征。算法刚开始接入的时候，我们的模型只有连续值特征。后经历了从完全连续值特征，到离散特征比重越来越重的过程。

连续值特征既有静态的（比如用户的性别，酒店价格等），也有基于用户行为的动态的（比如用户点击某个酒店的次数）。连续值特征的优点是具备良好的泛化能力。一个用户对一个商圈的偏好可以泛化到另外一个对这个商圈有相同统计特性的用户身上。连续值特征的缺点是缺乏记忆能力导致区分度不高。比如：在同一个商圈酒店列表中，一个用户点击了(A,B,C,D)，而另外一个用户点击了(W,X,Y,Z)。虽然两个用户的行为序列不同，但是统计值特征忘记了用户具体点击了哪些酒店，认为两个用户都是点击了 4 家酒店。

离散值特征是细粒度的特征：设备 ID，用户 ID，用户点击的 item ID 都可以做特征。这样一来不同的人，有不同的行为就可以在特征上有很好的区分度，因此离散值特征是模型可以把千人千面做得更进一步的基础。

离散特征的优点就是记忆力强区分度高。我们还可以让离散特征通过特征组合的方式，挖掘用户对于酒店更深层次的兴趣偏好：比如点击 item A 的人也喜欢 item B，我们可以直接基于(A,B)生成一个组合离散特征，来学习 A 和 B 的协同信息。

离散特征的缺点是泛化能力相对较弱。这是因为特征粒度太细，预测的时候命中率会偏低。同时在模型训练的时候，细粒度的特征相对粗粒度的特征更容易获得权重，让泛化能力强的粗粒度的特征学到的信息更少，进一步的恶化了模型的泛化能力。

举一个例子来说：每个样本都有一个唯一的样本 id，如果我们以样本 id 为特征，那么我们模型训练的时候，样本 id 特征可以完美的拟合 label；但实际测试中，会发现因为模型严重过拟合而效果非常差。这是设计离散特征需要考虑的特征记忆性和泛化性的 tradeoff，也就是特征尽可能细和特征在测试数据中命中率尽可能高的 tradeoff。

我们结合业务场景，探索出技术方案：可以让离散值特征在线上也有良好的泛化能力，从而最终让模型兼具较强的记忆能力和泛化能力。在工程方面：因为组合特征的存在，我们特征空间会很大（e.g. 现在酒店推荐场景特征可以轻松到亿级别），对模型训练和在线 serving 工程提出新的挑战。这也是我们在推进大规模离散 DNN 训练框架要解决的关键问题。

2.2 模型

我们模型经历了从 Logistic Regression (LR)，Gradient BoostingDecision Tree(GBDT)到 Deep Neural Network（DNN）的迭代过程。在业务开始阶段，数据量和特征量都比较少，通常会采用 LR 模型。随着算法的迭代，数据量和特征规模越来越多的时候，基于 XGBOOST 或者 LightGBM 构建 GBDT 模型是业务成长期快速拿到收益的好的选择。当数据量越来越大的时候，需要基于 DNN 的框架来把个性化模型做的更细。下面对三种模型的特点做一些简单的介绍。

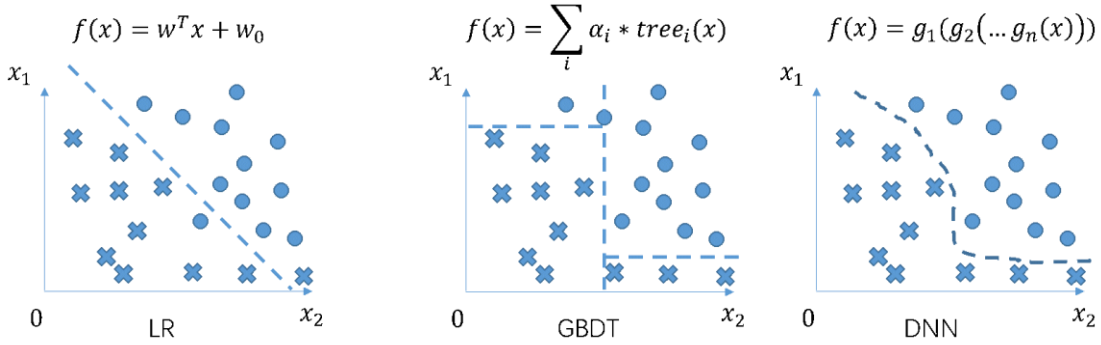


Figure2：模型决策边界

在 Figure 2 中，展示了三个模型的决策边界：

1）LR

在 LR 里，决策边界函数是线性的。

模型的优点：可以通过模型的权重大小，解释特征的重要性；同时 LR 支持增量更新；在引入大规模离散特征的情况下，业界在 LR 时代的经典做法是对 LR 加 L1 正则并通过 OWLQN 或者 Coordinate Descent 的方式进行优化，也可以通过 FTRL 算法让模型稀疏避免过拟合。

模型的缺点：线性决策边界这个假设太强，会让模型的精度受到限制；另外，模型的可扩展性程度低。

2）GBDT

在 GBDT 中，决策边界是非线性的；模型通过将样本空间分而治之的方式，来提高模型精度。

模型的优点：树模型可以计算每个特征的重要性程度，来获得一些可解释性；同时模型比 LR 有更高的精度。

模型的缺点：不支持大规模的离散特征，不支持增量更新；模型可扩展性程度低。

我们在酒店推荐场景中，尝试了 pointwise loss 和 pairwise loss，每次尝试都获得了不少的提升。

3）DNN

在 DNN 中，决策边界是高度非线性的。我们知道计算机通过与或非这种简单的逻辑，可以表达各种复杂的对象：音频，视频，网页等。而 DNN 每一层网络比与或非更加复杂，DNN 通过多层神经元叠加，成为一个万能函数逼近器。在理想情况下，只要有足够的数据量，不论我们实际的决策边界如何复杂，我们都可以通过 DNN 来表达。

同时 DNN，支持增量更新，支持根据业务场景进行灵活定制各种网络结构，支持大规模离散 DNN，在离散模型中学习出来的 Embedding 向量还可以用在向量相似召回里面。正因为有这么多的好处，DNN 正在成为业界推荐算法的标配。

这个模型的缺点是：特征经过不同层交叉，交互耦合关系过于复杂，而导致可解释性不好；工程复杂度在我们用不同结构的时候所有不同。

下面的表总结了三个模型的特点：

模型	决策边界	大规模离散特征	增量/在线学习	精度	可扩展性	可解释性	工程复杂度
LR	线性	支持	支持	中	低	高	低
GBDT	非线性	不支持	不支持	高	低	中	中
DNN	高度非线性	支持	支持	很高	高	低	Depends

酒店推荐算法

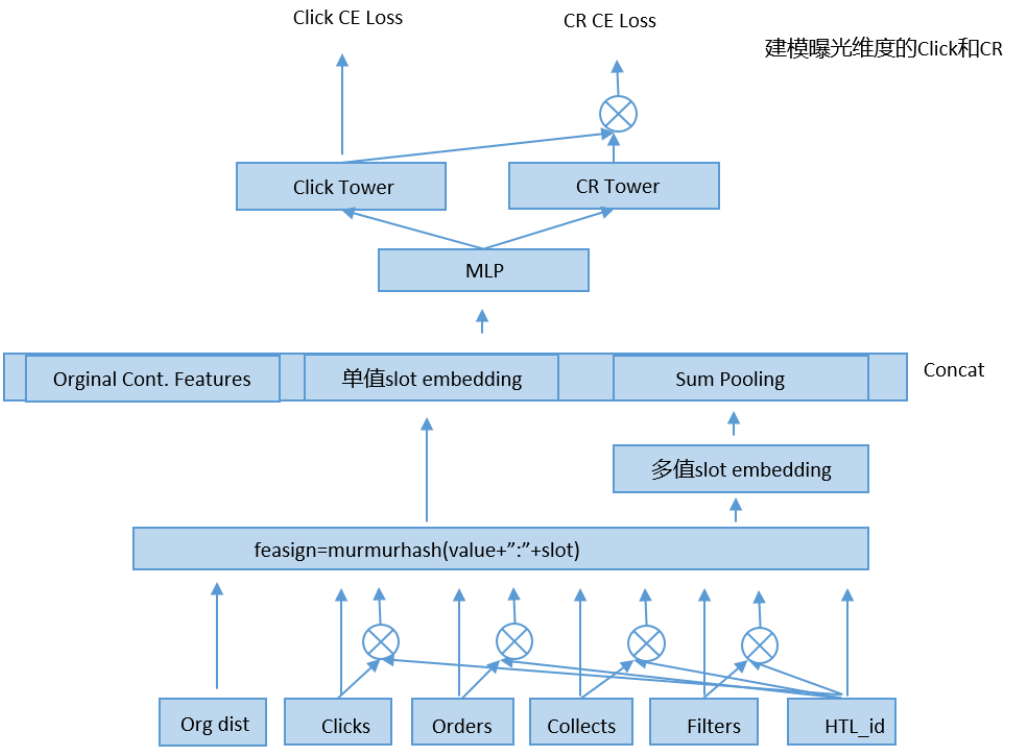


Figure 3：模型结构

用户在酒店的行为包括点击(click)和转化(CR)。在 LR 和 GBDT 的算法迭代阶段，我们优化目标只用到了 CR 信息。后来在 DNN 模型优化过程中，在模型建模中也引入了 click 信息，主要有两方面的考虑：

1）相对 CR，click 行为更加稠密；同时 click 也能反映一定的用户个性化偏好。所以合理的利用稠密的 click 信息，会对最终 CR 的学习有所帮助。

2）在酒店列表中，用户是先有点击，再有转化。对用户的行为漏斗建模，方便分析转化不好是哪个阶段导致的。对因为点击率偏低导致转化低的酒店，可以做定向优化和机制上的细节调整。

在引入了 click 信息后，需要解决的是如何进行多目标建模的问题。因为 DNN 结构灵活，我们尝试了各种多目标学习的方式，最后采取了 Figure 3 所示的 ESMM 的模型结构。整个模型从下到上包括如下三个步骤。

1）离散特征签名：签名是通过 hash 函数使“特征-slot”映射到 feasign，作为特征的唯一标示。每个 feasign 会通过模型学习得到一个 K 维 embedding 向量。

2）Pooling 和 Concat：离散特征由单值离散和多值离散组成：单值离散是指一个 slot 只有一个唯一值，比如用户 id；多值离散则是一个 slot 会有多个值，比如用户历史点击行为。相同的 slot 会放到一起做 sum pooling，这一操作把多值离散从 N*K 维向量映射为 1*K 维 embedding 向量；再和单值离散的 1*K 维 embedding concat 连接。

3）Multilayer Perceptron（MLP）和损失函数：通过 pooling 和 concat 将每条样本映射为固定长度的 1*M 维向量，通过全连接神经网络，以交叉熵为损失函数拟合是否点击、转化两个二分类目标。

基于以上的模型框架，我们相对 pairwise 的 GBDT 模型，有了显著的提升。后来在以上模型结构的基础上，我们针对离散特征，做了更多模型交叉的尝试，其中 Deep Cross Network(DCN)模型结构，在用户费力度指标和 AUC/NDCG 这些模型指标上都获得了非常大的提升。

在模型推进的过程中，我们也遇到了很多问题。包括如何做 normalization、如何处理异常值和缺省值、在从 LR 到 GBDT 再到 DNN 升级中如何处理数据分布变化，这些细节对最终模型线上效果都有直接的影响。当然还有一些很基础性的困难，是跟工程团队一起克服推进的。这个困难主要包括两方面：

1）DNN 模型结构灵活，在离散特征加成下，规模也很大。我们需要一个模型训练平台，支持我们定义 any model 并且支持 any scale 规模模型的训练。

2）在 DNN 这种复杂的非线性模型下，线上线下不一致问题会被放大。一个直接的影响是线下 AUC 提升很明显，但是线上效果是负向的。

针对这两个问题，我们开发了一套大规模离散 DNN 模型训练框架和统一特征处理框架，分别解决以上两个问题。

三、推荐工程的演进

针对策略需求，推荐中台做了很多的标准化服务。在本章节中，我们主要介绍推荐中台里面的大规模离散 DNN 训练框架和特征处理框架。

3.1 大规模离散 DNN 训练框架

大规模离散 DNN 模型因为具备很高精度和灵活性，成为目前业界做推荐算法的主流方法。但要推进大规模离散 DNN 在我们业务场景中落地，有一个前提：需要一个能训练大规模离散 DNN 模型的框架。我们结合长期实践经验，开发了一套分布式大规模离散 DNN 训练平台。这个平台具备如下特点：

- 支持任意形式的模型 Any Model：训练框架前端采取 tensorflow。Tensorflow 允许我们定义任意在推荐中常用的网络结构，并提供网络结构梯度计算功能。

- 支持任意规模的模型 Any Scale：训练框架后端是一个异步参数服务器；服务器节点可以任意横向扩展。服务器会做梯度融合和权重更新的工作。

- 开箱即用的默认参数：参数服务器中的参数（学习率，初始值范围）和权重更新函数是经过长期实践的结果；用这组默认的参数，不用调参，基本上在我们碰到的推荐业务场景都具备良好的效果。DNN 调参是比较关键的，参数调整不合适会直接影响模型精度；

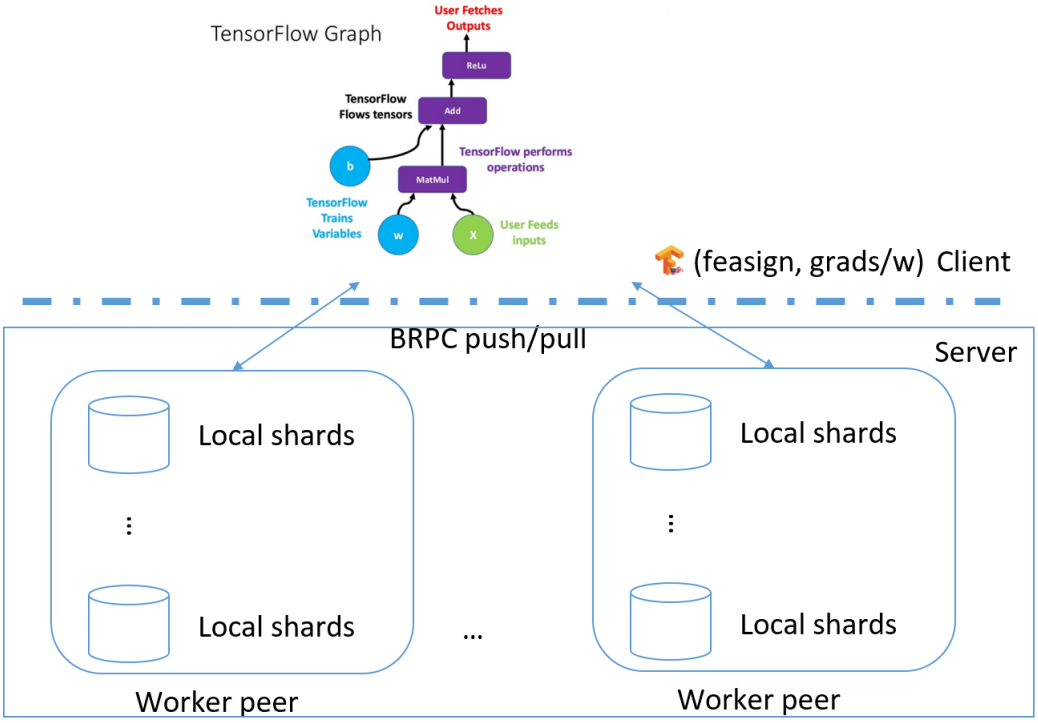
而因为 DNN 的复杂性，调参往往需要较长的时间摸索。我们调研的相对通用的默认参数，是目前上线快速迭代的前提。

模型框架有如下几个部分组成：

- Front 前端：采取 Tensorflow，策略同学通过 python 编写模型结构；tensorflow 本身提供灵活的网络结构定义和梯度计算能力。Tensorflow 是我们策略扩展性的保障。

- Client 端：Front 通过 Client 和 Server 进行通信；通信的内容就是（feasign, grads/weights）。Client 可以看作一个路由器：feasign 可以看作是 ip，我们根据 feasign 把相应的特征分发到相应的 server 机器上；而 grads/weights 可以看作通讯报文；而通信 push/pull 可以看作 http 的 post/request 请求。

- Server 端：基于 BRPC 协议，用 c++编写的后台。目前酒店推荐模型扫描一天只需要 6 分钟，训练效率高于 xgb。Server 端把每次请求里相同的梯度信息做 merge，并且根据自定义的优化器进行权重更新。Server 可以通过增加节点来横向扩展。Server 是我们工程能力扩展性的保障。



我们构建了分布式训练集群，开发了 python API 接口以支持用户模型任务的创建，任务队列的管理以及日志监控等。现在这套大规模离散 DNN 的训练框架以中台服务的形式，同时支持着许多其他场景推荐模型的迭代。

3.2 统一特征处理框架

线上线下不一致是目前推荐算法应用常见的问题。出现这个问题的本质原因有两个方面：

1）输入不一致：在业务的早期，我们并没有完善的落日志服务。离线训练数据的输入来自 hive 表的落盘数据；而线上的特征是来自近线数据服务。虽然工程同学建立了一定的数据同步机制，但仍然难以保证线上和线下数据严格一致。

2）OP 不一致：在业务早期，离线特征处理 OP 的逻辑是由策略的同学在 hive 调度框架上以 hive sql 的形式实现的；而线上代码则由策略同学跟工程的同学沟通后，由工程的同学开发的 java OP。这里面既存在策略工程两边同学沟通代价，也带来了因为不同人在实现 OP 细节的时候存在的不一致。

如果没有标准的框架来做特征处理的话，每次上线都会花很多时间在分析、排查线上线下问题上。

针对这个问题，推荐中台专门开发了一套统一的特征处理框架。在这个框架里面，我们做到了数据跟框架的解耦，以方便框架做不同业务场景的扩展。数据以标准化的 Protobuf（PB）协议定义，线上用户请求的时候框架会填充 PB 里面的上下文、用户侧和 Item 侧的信息。线上用这个 PB 作为特征抽取的输入，并同时异步落盘 PB 数据到 hive 表中。这样一来线上线下特征输入都是同一份 PB 数据，保证输入严格一致。同时我们线上线下用同一个特征抽取 JAR 包，特征配置也是同一份，这样保证 OP 严格一致。

通过这种方式，彻底地解决了线上线下不一致问题。特征抽取框架作为推荐中台模型服务的重要组成部分，支持很多业务场景做标准化的特征工程迭代。在这套框架下，工程同学专注优化框架性能，策略同学专注实现 OP 算子。这样一来，策略工程同学各司其职，减少不必要的沟通代价，增加整体迭代效率。

四、总结

本文主要介绍了我们在推进酒店排序模型迭代过程中，在策略和工程方面做的尝试，线上也取得了不错的效果。实际酒店业务中，排序模型仅仅是整个系统中的一环。线上推荐系统还要综合考虑用户端、平台方和酒店端的多方生态。但不论如何，准确的预估用户的个性化偏好，是任何精细化机制策略的基础。

在未来的工作中，我们还会面临很多挑战。比如酒店业务同学非常关注的黑盒模型的可解释性问题，我们在 DNN 模型可解释方面也做了一些探索性工作。随着基建的愈加完善，我们在算法上面还会有更深入的探索。

10 分钟给上万客服排好班，携程大规模客服排班算法实践

【作者简介】博天，携程高级研发经理，关注大规模约束优化问题的建模和算法设计。

一、背景

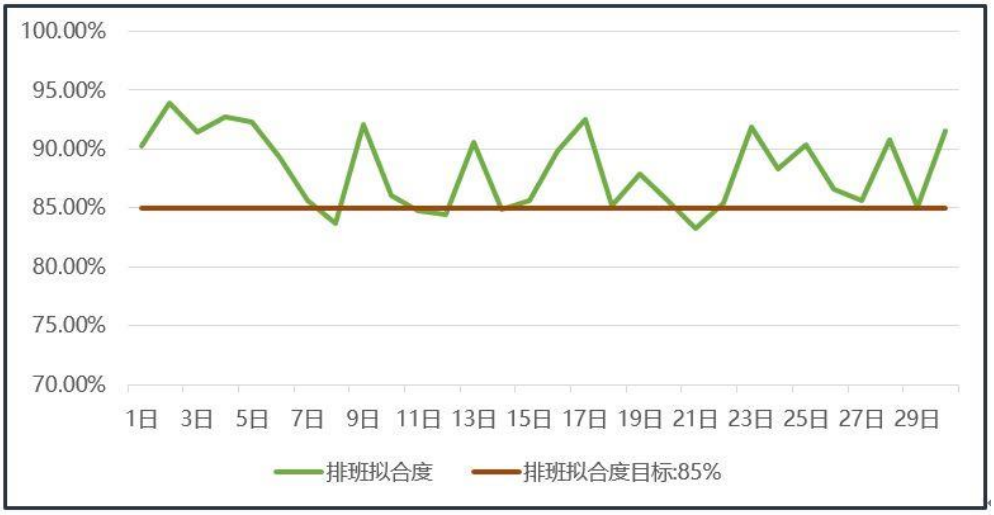
客户服务部门是携程以服务质量赢得客户信赖的基石，其拥有上万名一线的客服，每天进线量巨大；且伴随着业务量的起伏，每一周甚至每一天的不同时段都有需求量上的巨大变化。

如何在各个时段满足客户服务指标，同时尽量提高客服的工作体验，提升整体工作效率？这里自然而然就产生了对智能排班的需求。排班问题的核心是用更少的客服资源，既保证用户的服务质量，又尽可能保障客服班次体验。在这样的背景下，今天我们要探讨的排班算法就成了优化服务质量与成本的一个重要课题。

二、应用效果

携程的客服团队历史悠久，在很长一段时期里，排班使用人工调整的方式，已经形成了一整套完整的更新流程。但是各个部门的逻辑又有很大的区别，比如欧洲的客服需要核对时区的变化；有些部门以人为单位排班，而有些部门的管理架构要加一层小组的概念来排班。这些不同的逻辑和概念，是为了更好的管理和协调整个服务部门的资源，在携程几十年的历史中发挥了重要的作用。但也为算法的落地带来了大量不一的约束。

现在智能排班已经为携程的多个部门提供班表生成服务，在十分钟内提供优质的班表，并且符合各部门各个工种对不同工作场景的约束需求。



某段指定时期的验证数据

三、问题分析

排班问题，实际上就是一个带大量软硬约束的超大规模最优化问题。这是一个整数规划问题，这类问题一般都是 NP 难的。

解决这类问题，传统的方法有：

1）精确算法，主要有割平面法，分支定界法。这类算法解决的问题有一些特点：一般针对的问题规模都较小；优势是求解出来的必定是最优解。不过我们遇到的问题规模极其庞大，此方法的效率太低，无法满足业务需求。

2）近似算法，根据问题使用一些技巧自己设计出来。需要给出算法的近似比，复杂度分析，具有很强的推理能力。这类算法放弃获得最优解，提升了一些性能，不过同 1 一样，这类算法所能求解的问题规模依然比较受限制。在我们这个场景下，问题规模依然过于庞大，无法满足业务需求。

3）启发式算法，和前两种算法相比，启发式算法没有足够严格的理论分析，是算法设计者们根据经验或者观察性质设计出来的。没有严格的理论分析，通过启发式算法获得解，我们无法知道其是否是最优解，甚至无法精确得到其离最优解还有多远的距离，但是其性能上的优势十分明显，即使我们这种大规模的问题，也可以在数十分钟内获得非常令人满意的结果。

由于携程客服数以万计，再加上数十种基本班次，整体问题规模十分庞大。传统方法在可接受的时间范围内无法给出可接受的解。因此我们最终选择了启发式算法来解决问题。

针对这类启发式优化问题，问题的建模是个十分重要的步骤，这一步的好坏将直接决定问题最终的解决效果。

3.1 Nurse Rostering Problem

遇到问题，首先想要找类似的问题。我们找到了一个有一些相似的护士排班问题（Nurse Rostering Problem，后文简称 NRP），NRP 可以很好地帮助我们理解问题。

护士排班问题是说在给定的时间内为特定的一组护士安排班次，并使该排班方案满足各种硬性约束条件，同时尽量满足各种软性约束条件。但为了方便后续理解，这里以 the International Nurse Rostering Competition 2010 的规则为例，简要介绍一下这个问题，这部分规则约束十分核心。

NRP 问题的核心约束分软硬两类：

硬约束：

- 班次全部分配：每个班次都需要分配给一名员工。
- 班次不能冲突：员工每天只能轮班一次。

软约束：
这些约束实际情况中经常违反，因此这里决定将这些约束定义为软约束。

工作感受约束：

- 班次分配范围分配：每位员工需要工作超过 a 个班次且少于 b 个班次（取决于合同或约定）。
- 连续工作天数：每位员工需要连续工作 c 至 d 天（取决于合同或约定）。
- 连续休息天数：每位员工需能连续休息 e 到 f 天（取决于合同或约定）。
- 连续工作周末数：每位员工可以连续工作的周末数在 g 到 h 间（取决于合同或约定）。
- 尽量保持周末完整：员工如果需要在周末上班，那就周末两天都值班，放休就要尽量两天都放休。
- 周末上班班次一致：同一个员工在周末两天都上班的情况下，周末班次尽量保持一致。
- 不人性化的排班模式：尽量避免前后班次间隔时间太短，或连续上太辛苦的班次。例如：第一天上晚班，第二天接着上早班；或者连续上 3 天早晚班。

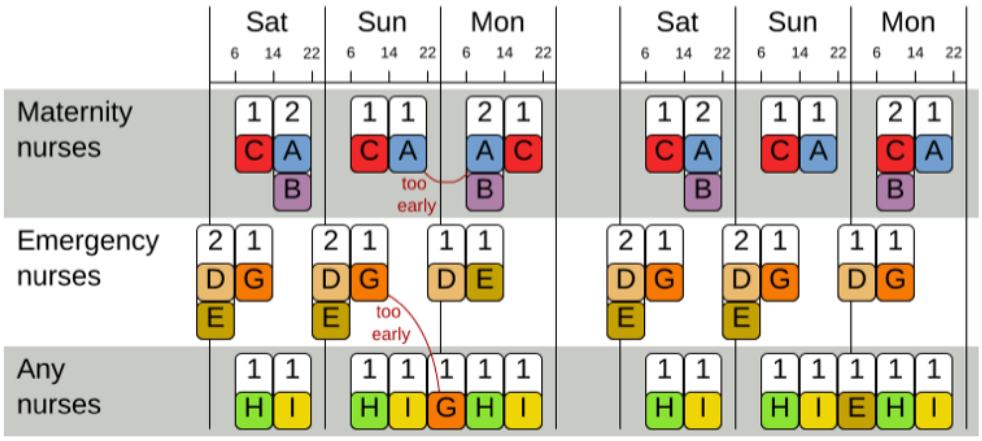
员工对班次的一些临时期望：

- 要求安排班次：某位员工想在指定的一天进行工作，尽量不要放休。
- 希望放休：某位员工指定某一天希望能给其放休，不要安排班次。
- 希望指定班次：员工希望能分配给其特定的班次。
- 希望避免班次：员工不希望被分配到特定的班次。

技能要求：尽量安排上班的护士已熟练掌握该班次所需的技能。

看完约束，NRP 问题的描述就很明了了。即在数值化定义好各个约束的重要性后，在尽量平衡所有约束的情况下，不停调整班表，获得最好的排班。

如下图，是一个最为简单的调整示例：



而最终的目标是得到一份最终班表，表示所有护士每天的班次安排。要注意在 NRP 问题中，调整的最小颗粒度是班次，这里将引出携程客服排班问题和 NRP 问题的最大不同。

3.2 携程集团客服排班

如开篇所述，携程的业务量每天都有着巨大的变化，加之我们的用户电话进线等待时间按秒计算，为了保证服务质量，对业务的控制需要细化到 15 分钟级别，才能保证服务质量，给用户带来最好的客服体验。

因此我们无法像 NRP 问题将排班简化为一天 3 个班次，而是细化到每天 96 个时段。由于整体安排的颗粒度细化到分钟级，就不能再仅仅安排班次，需要提高一个维度，额外计算员工的会议，休息，加班，放休等对每分钟业务量和应答量的影响。这导致整个问题复杂程度成指数级上升。

而且现实中的核心约束远比 NRP 复杂地多，总共近百条各种规则约束，对约束设计也提出了挑战。

约束设计需要数值化，但是数十上百的约束，两两之间的比例关系要恰当。这就要求数千的比例关系都要详细论证并且恰当，才能得到最终合适的班表，如若不然，很容易导致约束失衡，最终导致班表的不可用。比如，如果我们设计不合理，可能在业务量无法满足的基础上，安排部分员工超时加班，这将导致这部分员工的工作体验极差；或者在不需要的情况下，安排员工进行意义很小的加班，导致人员的浪费。

货币化约束设计

约束的数值化设计是问题的重点，但是问题的复杂性导致整个约束设计几乎非人力可为。

针对这一问题，我们提出了一种货币化的思路。所谓货币化，就是将单独的分数设计转换为一种标准分值（在这个场景下就类似于货币），统一货币化后，我们就可以直接将约束进行对比。

很多约束是天然可以货币化的，比如加班成本、津贴等等。余下无法天然货币化的部分，积极沟通，从业务角度给出一个初始的设计，再在算法使用中做微调，即可得到可用的约束设计与最终结果。

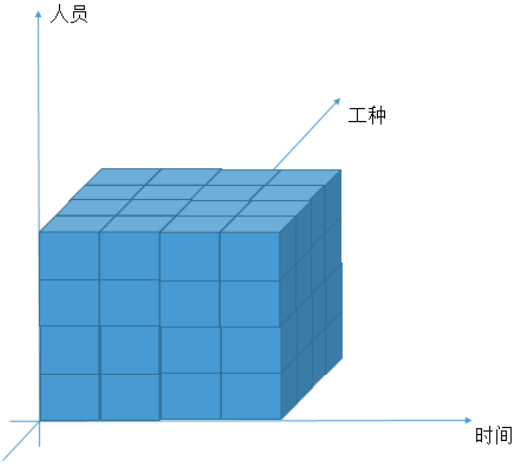
四、建模与设计

4.1 基础结构

如前所述，排班问题是一个超大规模的优化问题，解决这种问题我们选用启发式算法。根据以往经验，用启发式算法解决现实复杂问题的时候，最核心的是对问题进行建模，模型建立的好坏将直接决定后续系统的整体效果。

客服排班实际上就是要安排员工在每个时间段的工作状态（可能包括工作，休息，开会，加班等等），以及每个时段员工所做的技能工种。

因此，我们先抽象出一个最基础的模型，包含三个维度：员工，工种以及时间，这样就形成了一个三维度的空间。



三维空间中的每一块就是最小的工作状态安排单元

如上图所示，在这个模型下我们的目标就是填满整个空间，在尽量满足约束条件的情况下，给每个最小单元安排上工作状态（工作，下班，休息等），这样就获得了班表。

这样的结果逻辑上可行，但实际呢？

4.2 结构优化

我们先不考虑吃饭，开会等等复杂的设定条件，简单设定每个最小单元的工作状态为三种：工作，下班，休息。

按照基本结构直接进行排班是否可行呢？答案是肯定的。只需要通过算法调整每个最小单元的状态，在这三种状态中选择一个，然后设定合理的约束条件就能开始跑起来了，理论上也可以得到我们想要的结果。

不过设定好一切你会发现，根本得不到一个优秀的解，甚至一个合法的解都很难得到。这是因为按照这个模型去启发式搜索，搜索空间太广。我们可以简单算一下，设排班人数为 100 人，有 2 种不同的工种可以安排，安排一周的班次，时间颗粒度为 30 分钟，那整个任务的搜索范围高达：$3^{100\times2\times7\times48}=3^{67200}$。

作为对比，宇宙中所有粒子总数大约在10^{80}，围棋一共个 19×19=361 点，每个点由 3 种状态（黑，白，空），围棋所有的可能性也不过3^{361}。

而真实情况远比这假设要复杂的多，整个问题的复杂度也是指数上升，无法算出合适的结果。

因此需要在基础结构上做一些优化，加一层结构。首先，变量由每个最小时间单元改为工作区间，设上班时间都是整点，那一天最多 24 个班次，加上本休不上班，一共 25 种可能。休

息开会等可以抽象为上班时间内的整点“中断”，这样整体复杂度可以下降为大约：$25^{100\times7}\times8^{100\times7}$，这样总体看起来就是一个可解决的问题了。

五、算法流程

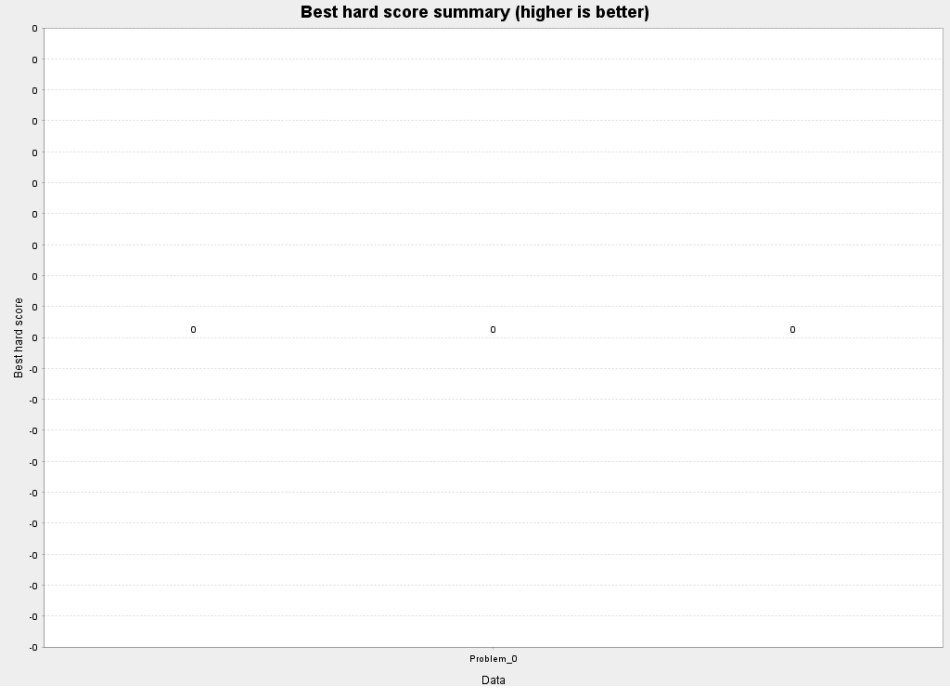
在有了模型的前提下，需要做一些算法选择与流程设计。我们首先做了一些基准测试，然后对整个算法流程做一个统一的规划与设计。

5.1 基准测试

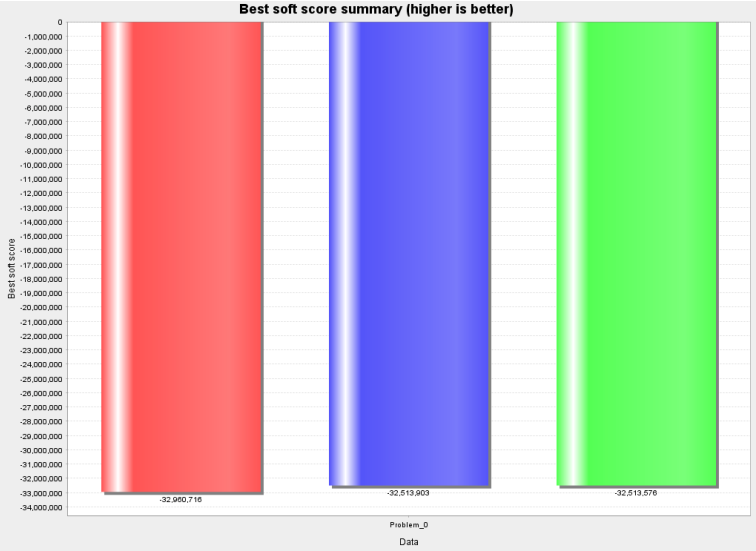
我们获取了一个简化的业务场景，保存下来后，以此为基准搭建了一个基准测试。

首先测试了常见的算法，包括：FirstFit，Hill Climbing，Tabu Search，Late Acceptance，Great Deluge，Simulated Annealing。

以下是几个算法的部分测试结果：



硬约束结果对比

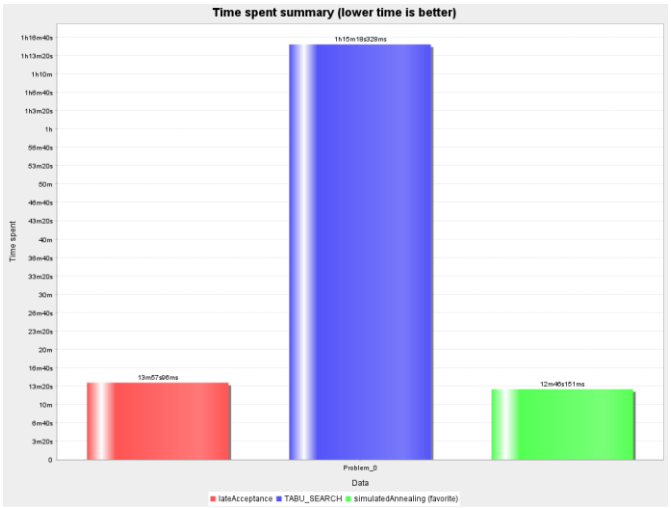


软约束结果对比

可以看到几种主要算法总体能得到的效果差异不大。从理论上来说，即使采用暴力搜索（Brute Force）的方式，只要时间够长，也总能得到一个不错的解。

而在实际的业务场景中，对于班表的计算速度是有很严苛的要求的。有时我们发现预测的业务量产生偏差，这时候就需要立刻重新生成当天之后的班表，并及时下发。对时间的要求是分钟级的，因此也要测试一下算法的运行速度。

几个重要算法的部分测试结果：



算法时间测试结果

而几个算法的提升速度，测试下来短期 Tabu Search 的结果提升较快，不过后续乏力，整体略逊于另外两个算法。

5.2 应用

这里的算法基准测试只是初步筛选一下可能有用的算法，整个计算流程中，从班次的确定，再到休息，加班，放休等的安排，实际是分多个层次的。从逻辑上讲，顺序应该是：

- 确定班次
- 确定加班放休安排
- 确定休息，吃饭等当天安排
- 确定周会安排

我们算法流程设计上也必须遵循一定的顺序，毕竟在班次没确定的情况下，我们也没办法安排当天的餐休。

因此流程上：

- 先安排了 First Fit，快速得到一个不太差的初始解。
- 接下来选择能够短时快速提升的 Tabu Search，仅仅针对班次，快速搜索找到一片高分区域。
- 最后利用 LAHC 或 Simulated Annealing 等算法针对全部变量搜索一个较优解。

5.3 性能优化

在我们的业务场景中，问题规模很大，正常计算需要数小时甚至数天才能得到最终的结果，这一场景下是不可接受的。

一般的算法系统框架都是单机甚至单核的，这里首先要对算法系统进行多核支持，启发式算法的流程分为：

- 选择待选步骤
- 各步骤尝试
- 确定下一步（或者找不到下一步）。

对整个流程进行梳理后，我们判断整个算法性能消耗最多的地方是第二步，也就是各个步骤的尝试部分，在这一步加入多线程的支持，在选择了待选步骤后，对各个待选项进行多线程的并行尝试。

事实证明，这一步将性能提升 10 倍以上。

5.4 分布式多机并行

但在有些场景下依然不能满足目标性能的需求，无法在 30 分钟内得到班表。因此我们不得不利用分布式计算的思想，将问题进行切分，尝试在多机上进行并行计算，最后将结果汇总，在主机上进行汇总计算。

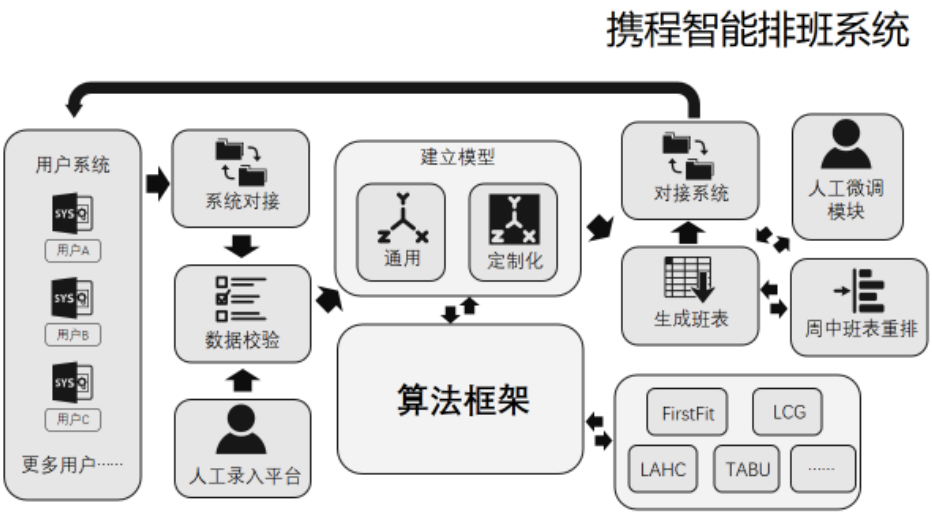
了解启发式算法的同学应该清楚，这样每台单机都获得的是问题的一个子集，缺点是无法全局考虑问题，最终结果和单机运算相比会略有下降。但是性能确实成倍的提升，在多工种的

特殊场景中，我们通过这种思路，实现了复杂工种技能下的混合排班。并且最终效果也依然能够达到我们对拟合度的要求。在启发式搜索问题中，这是一种损失少量效果而大幅提升速度的有效技巧。

六、平台结构

在算法的基础上，我们搭建了一套智能排班中台作为项目目标，使用户可以轻松访问服务，得到灵活的班表。

整体架构示意图：



整个系统包含多个模块，互相协调保证携程各个部门的灵活调用。

总体调用流程如下：

1）各个 BU 通过系统对接排班系统，将数据做清洗和标准化
2）进行基本的数据校验，在实际环境中，各种各样隐藏的矛盾错误数据，都需要在这一步识别出来。
3）合并人工录入的部分数据，考虑到部分数据的时效性以及一些偶发需求，我们需要开放人工信息的录入平台，让用户可以录入部分数据
4）建立模型
5）算法求解得到优质解
6）生成班表
7）（可选）人工微调
8）生成班表，并对接回各个业务系统

七、结语

随着越来越多的 BU 接入到智能化排班的系统中，算法平台发挥着更大的作用。在不断接触

不同部门的过程中，我们也发现了模型建立中和算法设计中的各种问题。在我们长期的开发过程中，也走过不少弯路和死胡同，最终发现，需要全局的去看待一个业务问题，并提出解决方案，抓住真实痛点，才能最终落地。

CrateDb 在携程机票 BI 的实践

【作者简介】Loredp，携程数据分析经理，关注大数据存储、大数据处理以及 linux 等领域。

一、前言

随着整个互联网流量红利进入末期，各大厂在着力吸引新客的同时，在既有客户群体的运营上也是煞费苦心，各种提高客户体验、个性化服务的场景层出不穷。

携程机票大数据部门在实践过程中需要同步数据、选型引擎来存储处理数据，利用接口将模型结果开放给生产环境调用，因此我们的数据存储修炼之旅会涉及到接口现状、接口大道之旅、安装部署、同步数据、生产应用以及未来的趋势-如何实现容器化。这当中，我们遇到了很多问题，也解决了很多问题，本文将分享机票大数据平台在数据存储这一块的实践经验。

二、机票大数据接口现状

携程机票大数据平台接口组碰到的问题：

- 如何存储
- 如何查询
- 如何维护

2.1 如何存储

机票大数据基础架构团队接口组在 2018 年之前，数据的存储方案基本是：hive、mysql、redis。以下是我们现有的存储选型：

接口需求		Hive	Mysql	Redis
性能要求	请求 QPS			
>1s	<1	√	√	√
<1s	<10		√	√
<500ms	>100			√
<100ms	>100			√

这就造成了机票大数据部门的 redis 集群内存需求暴涨，目前我们统计 redis 使用的数据：挂在机票大数据部门的 redis 集群数量有几十个，内存达到了十几个 T。当然接口的性能也达到了前所未有的快速和高效，基本都是 10ms 左右。

2.2 如何查询

Redis 的查询方式比较单一：通过唯一 key 去查询 value。这种查询方式在简单的唯一值查询中比较有效，但是当遇到，同一个数据源多关键字查询的时候，就得维护多份数据源。举例：在价格趋势的接口中，我们提供了多种价格趋势组合：国内、国际、单程、往返、航线、航

班。如果使用 redis 存储，需要维护同一份数据多种 key 的存储方式，极大地浪费了存储空间。

Redis 还有一个问题是时间范围的筛选，还是在上面的价格趋势接口中，需要按照查询时间返回历史同期在一定起飞时间范围的价格数据，所以我们需要存储多个时间日期的数据（当然也可以用 set 等结构，但是会面临如何删除过期数据的问题），同时在查询的时候需要循环取一定时间范围的价格。

2.3 如何维护

1）接口维护

大数据基础平台团队一共维护了几百个接口，其中 1/3 的接口是提供数据给调用方的，这当中又有一些接口只是提供简单的查询操作，但就是这些简单的查询，需要我们提供海量的数据存储、快速精准的查询。每个接口的上线需要经过项目资源申请（包括机器资源、人员资源）、数据同步、开发、测试流程，最后才能上线。一整套流程走下来，耗费 2-3 天/人，而且基本上都是是重复性的工作。如何解放这些人力和机器资源，就变得很迫切了。

2）数据同步

提供给外部使用的数据大部分都是存储在 hive 中，在不使用 presto api 的方式访问时，我们需要将 hive 数据导入到 redis 或者 mysql 中，供接口访问。在 zeus 平台上，我们建立了各种导数据的流程，如何将这些简单、重复度高的流程自动化呢?

整个接口的架构图如下:

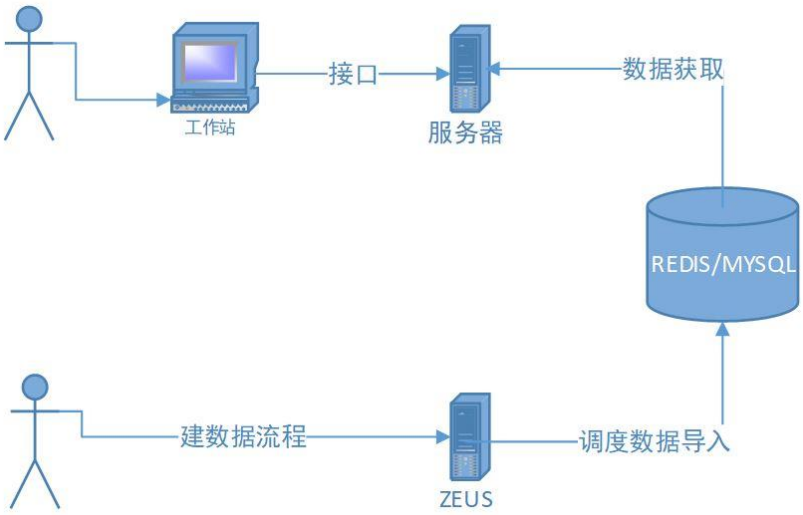


图 1 redis/mysql 作为主要存储的架构图

三、机票大数据接口的大道之旅

认真研究了接口调用方本身的性能，我们发现调用方在调用第三方提供的接口时，基本都是异步进行的。如果把调用方调用的所有第三方接口当成一个木桶，机票大数据基础架构团队的接口就是其中的一块木板，只要不是最短的木板，就可以在保证性能的情况下降低整个接口的响应时间（当然这不是技术上的退步，而是选择合适的方案）。通过上面的存储选型对比之后，发现在 100ms-500ms 这个性能段里面没有一个合适的存储方案能够提供。

我们调研了几种 NOSQL 数据库方案，综合存储、查询等指标发现 CrateDB 比较符合现实需求。将几种存储做了一个对比，如下：

对比	Redis	Mongo	CrateDB
查询速度	<10ms	100ms~500ms	100ms~500ms
SQL	不支持	不支持	支持
数据结构化	不支持	支持	支持
存储机制	hash	Sharding+partition	Sharding+partition
资源利用	内存资源	硬盘+内存	硬盘+内存
数据可重复使用	不支持，单一固定 key	支持	支持

3.1 CrateDB 介绍

CrateDB 是构建在 NoSQL（ElasticSearch）基础之上的分布式 SQL 数据库，它结合了 SQL 的通晓程度和 NoSQL 的可扩展性和数据灵活性：

a、使用 SQL 处理结构化或非结构化的任何类型的数据
b、以实时速度执行 SQL 查询，甚至 JOIN 和聚合
c、简单缩放

3.2 CrateDB 与接口存储

CrateDB 很好地解决了 100ms-500ms 性能段的短板，并且使用磁盘+内存的方式存储数据，减少了内存的使用。目前在我们生产时间中，通过 12 台 8 核 24G 虚拟机 30%的磁盘空间覆盖了 10 亿数据（如果是 redis 至少需要 300G 的内存，如果做 slave，容量 double）。

3.3 CrateDB 与接口查询

CrateDB 提供了如 MYSQL 的表、字段等概念（底层使用 ES 存储引擎），我们可以将同一份数据源进行多维度的操作，比如上述讲到的价格趋势里面基于航线和航班的价格趋势，这两个接口可以使用同一套数据源，因为航线的价格可以基于航班数据进行聚合操作，这样就大大减少了冗余的数据。同时类 MYSQL 表的特性使得时间范围的查询变的 so easy 了。

3.4 CrateDB 与接口维护

1）与接口结合使用

因为 CrateDB 支持标准的 SQL，我们开发了机票大数据基础平台的通用性 api 系统，通过将取数逻辑 SQL 化的方式，同时利用 qconfig api 将新增的数据需求进行模板化、配置化，统一了接口代码开发的流程。配置页面如下：

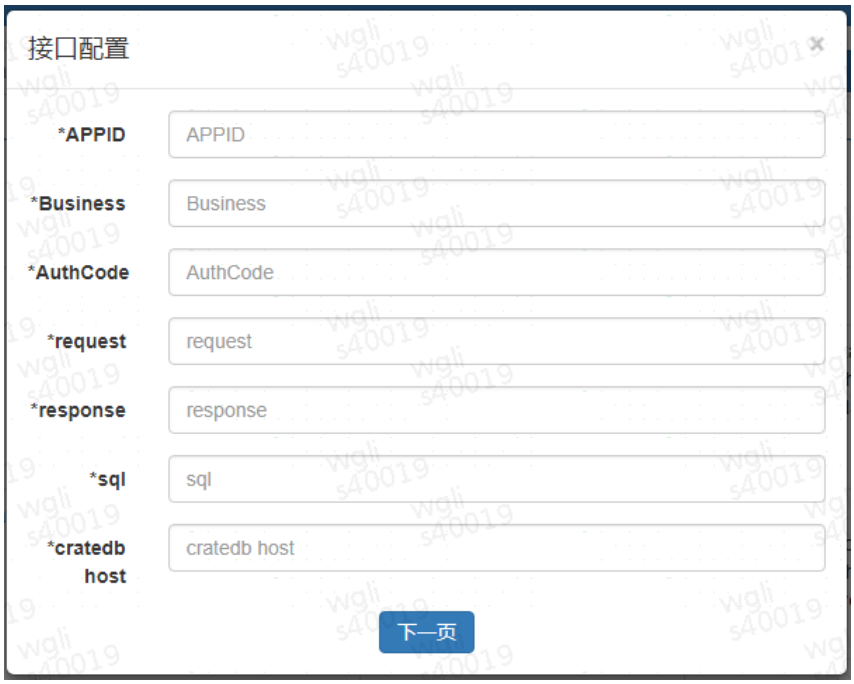


图 3 接口配置页面

2）数据同步

通过 zeus api 将同步数据流程模板化，配置页面如下图。并且在 zeus 平台上，使用 spark shell 方式将 hive 数据导入到 CrateDB 中，抛弃了以前 jar 包的方式。这种方案可以在几分钟内导入千万级的数据（取决于 CrateDB 表的数据结构，减少索引、doc_values 以及刷新间隔会都有利于导入的速度）。

图 4 zeus 流程配置页面

3）容器化

如何更加有效地管理、维护 CrateDB 集群？为此我们上了 k8s，将 CrateDB 容器化。为了更好地管理这些 k8s 集群，引入了 rancher，rancher 是开源的企业级容器管理平台，通过 rancher，我们再也不必自己去从头搭建容器服务平台。同时 rancher 提供了在生产环境中使用的管理 docker 和 kubernetes 的全栈化容器部署与管理平台。将网络、磁盘虚拟化之后，资源的利用率大大提高，减少了虚拟机的使用。自动水平扩展，以及 pod 的监控等特性，都极大地提高了维护 CrateDB 的能力，我们管理的 CrateDB 集群如下：

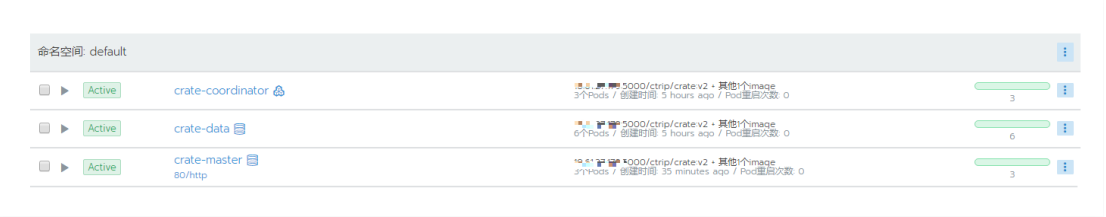


图 5 rancher 管理 CrateDB 集群图

3.5 与接口结合的其他优势

1）存储机制多样化，底层的存储机制支持多样化的数据类型，同时支持 partition、sharding；

2）数据结构化，CrateDB 提供结构化的展示，有利于数据的可视化以及降低非技术人员的理解难度，解决了 redis 可读性差的问题；

3）存储可靠性，数据持久化存储在磁盘上，支持 replica，相比于 redis 的内存存储更加可靠（当然 redis 也可以落盘，但这就会限制 redis 的速度）；

4）成熟的优化机制，针对 es 的优化我们有丰富经验的技术人员支持。举个例子：我们有 9000 万+的用户行程数据，因为数据比较详细，字段的内容比较庞大。通过去掉部分字段的索引，去掉 doc_values 等操作将数据存储大小从 90G 降到了 30G，同时也提升了搜索速度。

目前在生产上我们部署了 2 个 CrateDB 集群，其中一个集群由 12 台 8 核 24G 内存虚拟机组成。在集群中建立了 12 个数据表，存储了 20+亿条数据，经受了生产的实际考验，接口性能指标如下：

数据量	99line	95line	avg	查询特点	描述
10 亿+	200ms	80ms	10ms	多关键字、时间范围查询	整个集群请求量 1500qps
500w+	150ms	50ms	10ms	多关键字查询、排序	单个表请求量 400 qps
9000w+	200ms	100ms	60ms	多关键字查询	单个表请求量 10qps

性能满足了大部分调用方的使用需求，同时系统数据上线的流程由以前的申请资源、开发代

码、测试、上线，到现在的系统配置、测试、上线，释放了部分的开发资源，并且保证了数据的质量。接口上线时间由以前平均 2-3 天，缩短为 2-3 小时。新的接口架构图如下：

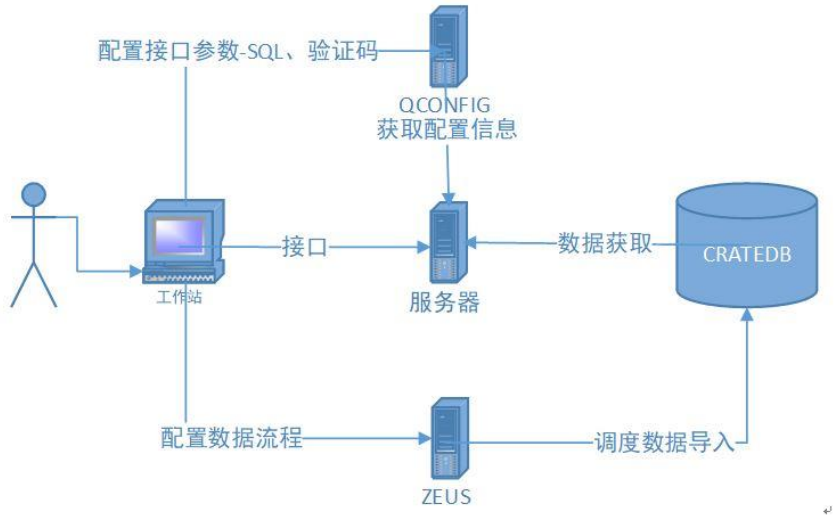


图 6 CrateDB 作为主要存储的架构图

四、安装部署

CrateDB 有官方版以及社区版，为了更好地进行自维护，我们选择了社区版（通过源码编译）。CrateDB 的部署与 ES 的部署基本一致。需要注意的是，在分配内存的时候尽量多留一些内存给系统，这将有利于数据查询速度。部署后的 webui 如下：

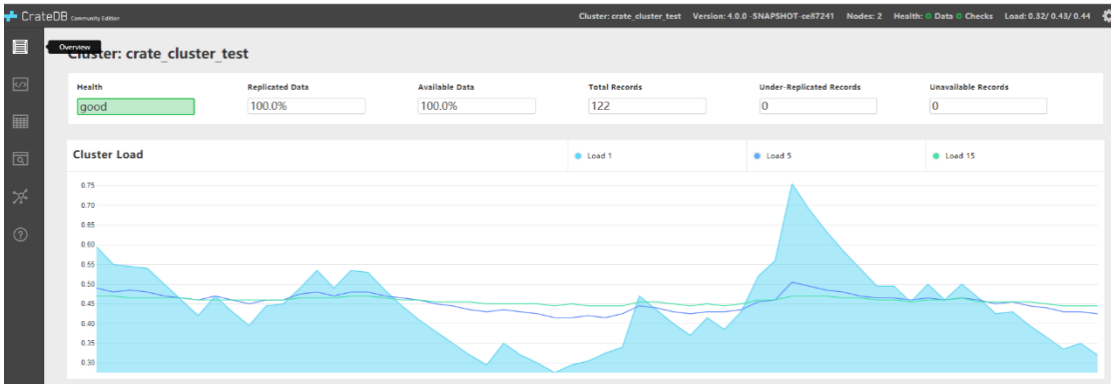


图 7 CrateDB webUI

五、数仓中的实现

目前在数仓中的应用主要体现在各种指标 dashboard、metrics 的展示，比如 fltinsight。与以往通过 presto 接口获取数据的方式相比，更加直接、高效。而且 CrateDB 支持各种字段的聚合、统计，是各种指标存储、展示的不二之选。当然后续数仓组也会在数据展示这一块全面推广 CrateDB 的使用。

六、小结

没有完美的存储方案，只有最适合的存储方案。通过上述机票大数据平台在数据存储这一块的实践经验，相信每个团队在面对选择存储方案的时候，结合自身需求去选择适合自己的存储技术方案，达到“大道”。

携程度假数据治理之数据标准管理实践

【作者简介】Leon Gu，携程数据仓库专家，专注于大数据、数据仓库、数据治理等领域。

本文分享的是数据标准管理在携程度假数据治理中的实践，希望对想要了解和学习数据治理实践的读者有所帮助，也希望能收获宝贵的建议。

一、为什么要做数据标准管理

数据治理的问题并不仅仅只是治理数据本身，其最终目标是提升数据价值，它是一个包括组织、制度、流程、工具的管理体系。去年我曾写过一篇关于数据治理的文章《数据治理落地难？携程度假数据治理需求设计实践》，从团队提效、需求梳理、模型设计、指标管理四个方面分享了携程度假在数据治理中的经验。

数据治理不是简单的一次性的行为，它是一个长期持续性的项目集，要想通过数据治理将企业的数据管理、数据应用水平提升到新高度，而不是沦为理论实践两张皮，需要跨组织职能的协调以及在数据治理的各环节中将标准管理落实到位。

数据治理是涵盖数据的采集、处理、分析、使用的全流程管理体系，数据标准则是数据治理各项活动的基础，是企业数据治理理论和方法与实际信息系统和数据的桥梁。携程度假的实践经验总结发现，数据标准管理需要包含以下三要素：

- 范围：成功的数据治理应当是清楚地了解需要治理什么
- 工具：对规范数据治理活动标准提供系统的支持
- 制度：对在人员和流程方面的行为方式及有效地使用工具提供指导

二、数据标准管理在携程度假的实践

下文将从数据管理的两个核心领域中选取部分案例来分享一下携程度假在数据治理方面的探索与实践：

- 数据集成
- 元数据管理

2.1 数据集成

携程度假覆盖的数据源有业务系统类的结构化数据、埋点日志类的半结构化数据及其他内容类的非结构化数据。数据集成不仅只是解决技术上的从源端抽取到数据中台，其数据内容的变更通常会对现有流程及下游的数据应用产生影响，因此基础数据的管理重点在于变更管理和统一标准管控。这里会介绍针对结构化数据生产变更的标准管理。

2.1.1 生产变更的标准管理

生产变更的标准管理主要解决了以下问题：

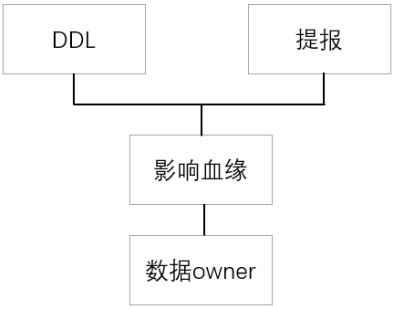
第一、哪些变更是需要通知的？

1）结构类变更，这部分的操作都会经过数据库的 DDL 转换，所以方案上只需要监听变更消息，自动进行解析出核心的变更信息，比如库名、表名、表 OWNER、变更类型（新增/修改/删除）、变更对象（表/字段）等，并且定义一个标准的数据结构用于通知。目前工具可以对以下类型的变更做自动化感知及通知：

- 新增字段
- 修改字段名
- 修改字段类型
- 删除字段
- 修改表名

2）内容类变更，这部分往往通过系统化的方式无法感知到，需要对应表的研发评估出影响面，并线下周知下游。这种方式常常会有两种问题产生，其一是影响面评估不准——漏报；其二是依赖人的责任心——忘报。目前工具可以对以下类型的变更做线上人工提报及通知：

- 字段枚举值变更
- 字段废弃
- 字段逻辑变更
- 表废弃
- 表迁移

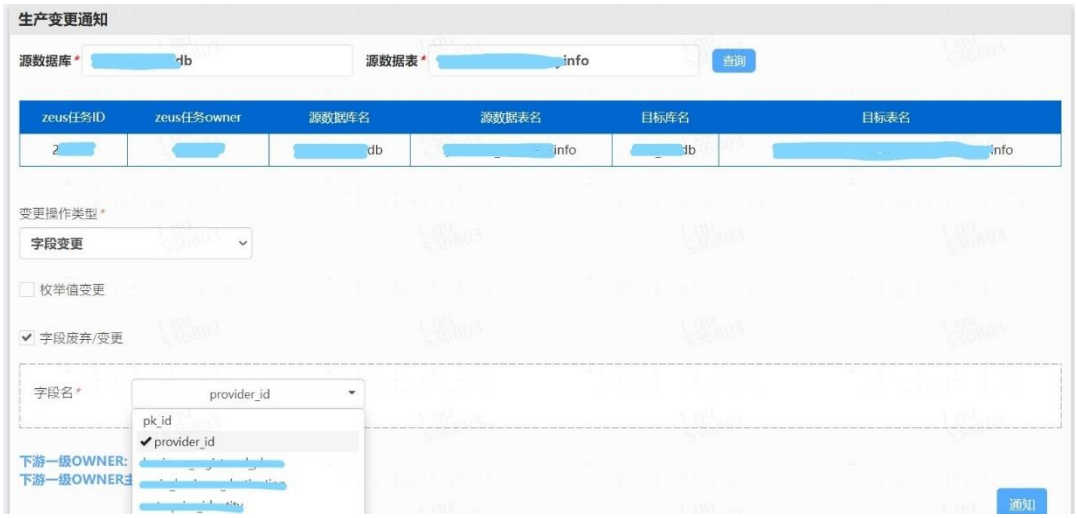


第二、变更通知谁（影响范围）？

能评估全、评估准生产变更对于数据侧的影响面及通知人对于研发侧的 TO 来说是一件有挑战的事情。但从数据中台能力的角度而言，数据血缘是一个中台基础元数据组件，其中已经包含了完整的关系信息，表与人、表与表、表与字段、字段与字段等，我们可以基于这些关系信息建立起影响分析的能力。

目前携程度假的感知做到了 ODS 层，如图，TO 在界面上选择相应的库和表，影响分析就能识别展示出相关的影响面及通知人。下一阶段我们还将打通应用端的影响分析，将会通过整

合应用端数据血缘信息覆盖到数据应用终端的影响面识别。



第三、有工具如何能保证强执行?

生产变更制度与流程:

1)当生产发生变更时,通过自动化感知或 TO 在线提报的方式通知到下游相关数据 OWNER,正常境况下都会有一定的时间余量。

2）数据 OWNER 接收到生产变更通知后，需要及时确认影响面，必要的沟通确认还是不可缺少的重要环节，并评估改动成本及计划时间。

3）无论是自动感知还是在线提报，都有可能由于人为因素导致执行不到位，所以必须对于所有的变更感知方式有一个事后的 DQC 告警，同时告警的能力最好能够做到 T+0，因为可以及时感知告警在第一时间处理解决，保障任务的基线和数据应用不受影响。

4）最后需要把质量闭环加入到流程中，保证在流程中发现的问题和由于人为因素导致的执行不到位能定期反馈到 QA 或者在质量会上曝光，以此来提升大家的质量意识，形成良性循环。

2.2 元数据管理

元数据对于数据管理和数据使用来说都是必不可少的。所有大型组织都会产生和使用大量的数据，在整个组织中，不同的人拥有不同层面的数据知识，但没有人知道关于数据的一切。因此，必须将这些信息记录下来，否则组织可能会丢失关于自身的宝贵知识。

元数据管理提供了获取和管理组织数据的主要方法，建立业务术语表，用于定义和定位组织中的数据，确保组织中数量繁多的元数据被管理和应用。假如没有可靠的元数据，组织就不知道它拥有什么数据，为保证其高质量，应把元数据当作产品来进行管理。好的元数据不是偶然产生，而是认真计划的结果。这里介绍携程度假对于业务元数据的标准管理。

2.2.1 业务元数据的标准管理

业务元数据的标准管理主要解决了以下问题：

第一、数据地图中的业务元数据需要覆盖哪些？

携程度假的数据地图工具集成了模型、指标、看板、数据集四种业务元数据，除了元数据信息的搜索与展示，也打通了权限申请流程及在线管理的功能。

- 模型：数据中台中规范化的主题域模型。
- 指标：数据中台中标准化的业务分析指标。
- 看板：支持有固化场景的标准化的指标分析看板。
- 数据集：支持无固化场景下的明细查询与数据探索分析。

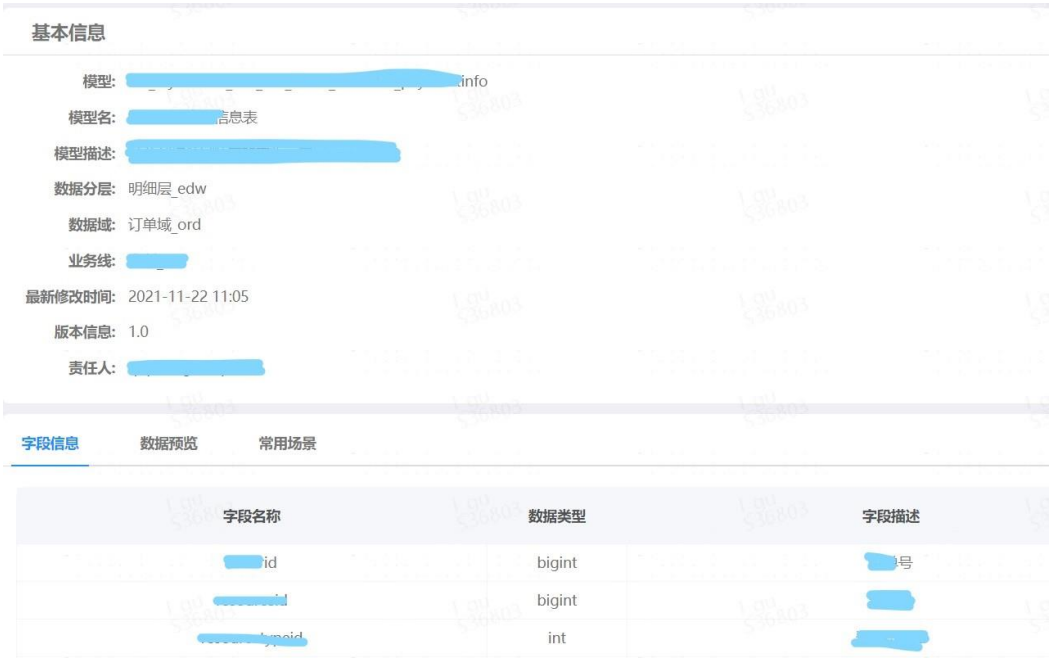
第二、如何管理并维护准确一致的元数据？

- 建表元数据规范

数仓模型建表的流程需要严格遵守建表工具规范，主要的元数据信息有：

- 分层：按照经典建模分层思路，分为 ODS 层（操作数据层），EDW 层（明细数据层），CDM 层（汇总数据层），ADM 层（数据应用层），MID 层（中间层），DIM 层（维度层）
- 一级主题：按照数据域进行划分，例如常规的订单域、日志域、商品域、服务域等
- 二级主题：按照业务线进行划分，度假包含的业务线较多，例如团队游、门票、用车、租车等
- 分区类型：全量分区或增量分区
- 重要等级：标识表的重要程度优先级，分为 P0-P3
- 敏感级别：标识商密与个密敏感程度，分为 L1-L4

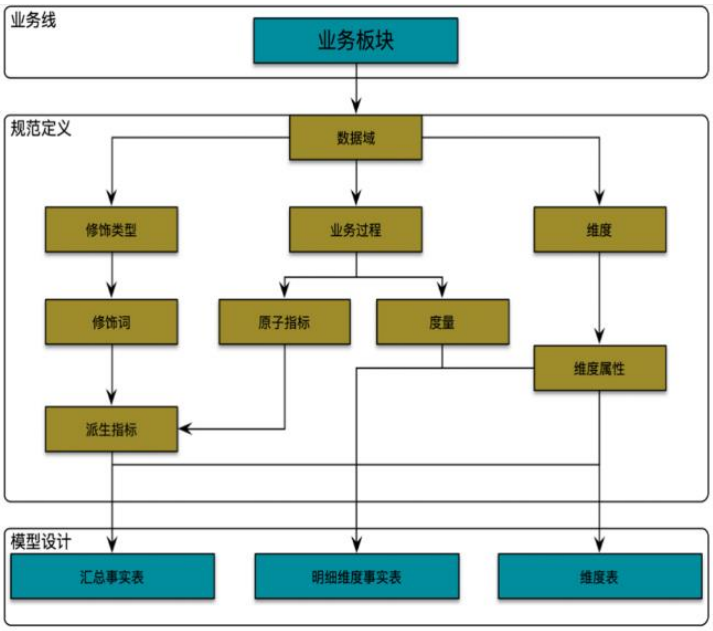
建表工具会根据元数据的选择自动生成标准的建表语句模板，其中包含了表名规范、字段名及注释规范等，表 owner 根据工具的引导完成表名及表的逻辑结构的设计，并将这些信息和完整的注释通过工具提交建立正式表。



- 指标元数据规范

指标的定义是由组成指标的业务术语构建而成，主要的业务术语有：

- 数据域：指面向业务分析，将业务过程或者维度进行抽象的集合。例如常规的订单域、日志域、商品域、服务域等
- 业务过程：指企业的业务活动事件。例如订单域中的下单、支付、退款等
- 时间周期：指用来明确数据统计的事件范围或者时间点。例如最近 30 天、最近半年、截至当日等
- 修饰词类型：指对修饰词的一种抽象划分。例如商品维度-商品类型、时间维度-预定日期、渠道维度-分销渠道等
- 修饰词：指除了统计维度外指标的业务场景限定抽象。
- 原子指标：指基于某一业务行为下的度量，是业务定义中不可再拆分的指标。例如成交-订单数
- 维度：指用来反应业务的一类属性，这类属性的集合构成一个维度。例如商品维度、时间维度、渠道维度等
- 维度属性：指隶属于一个维度下的属性值。例如地理维度下的城市 ID、城市名称、所属国家等
- 派生指标：派生指标=业务线+一个原子指标+多个修饰词（可选）+时间周期。例如团队游_成交-订单数



指标的设计与注册必须严格遵守指标的定义规范，且在指标管理系统中进行操作，所有上述的业务属于都在系统后台事先进行标准化，标准化的内容包括术语的命名、分类以及准确的定义。原子指标和派生指标的生成过程都是基于标准化的组装，所有的相关信息也是结构化的自动生成。

指标实践中在最终生成一个派生指标完整元数据时，有两个设计上必须考虑到两点：

第一，指标的口径必须有一个明确的业务维护人而不应该只有一个数据开发 owner，关于指标的生命周期管理（变更或者下线）都是需要指标业务 owner 收口，这样才能保证指标的定义和业务的一致性；

第二，在业务术语定义标准结构化的同时，最好加上一个业务话术的定义描述，便于业务更好的理解指标的业务含义。

第三、数据地图如何更好的帮助业务使用数据?

- 门户看板的指标应用标准化

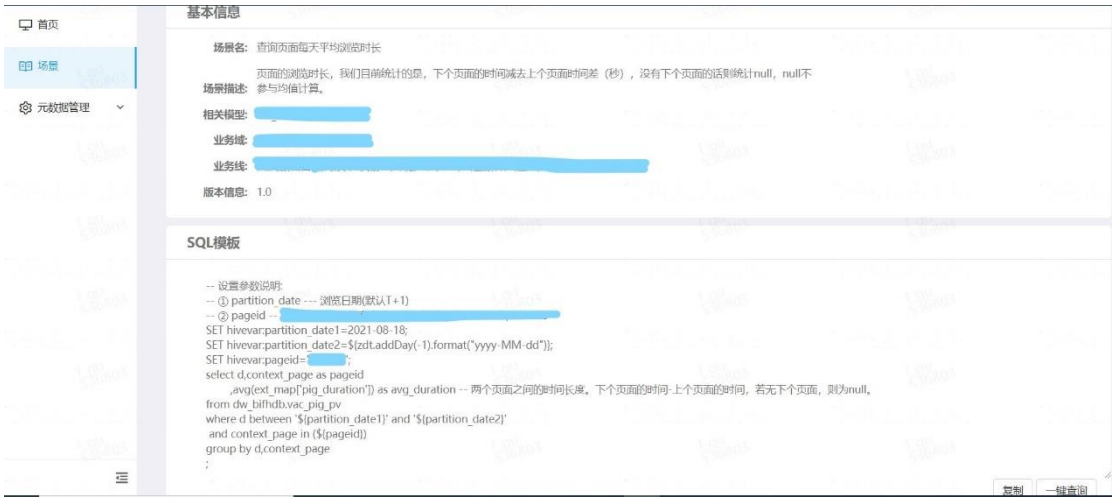
业务门户看板是固化场景的数据查询分析入口，其中包含了标准化的指标、筛选条件及可视化图表。由于前端展示的个性化需求，指标的展示名称往往不能直接反应指标的口径，往往存在同名不同意的情况，导致业务汇报及使用数据的混乱或需要频繁的线下沟通及确认。

门户看板的指标应用标准化就是通过整合数据地图的指标元数据，在看板工具中强制需要绑定标准化过的指标 ID，即已在指标管理系统中维护的指标，在前端的交互上，会清晰的显示出相关指标的指标定义，如果业务还需要进一步查看更多元数据信息，也可以跳转到具体指标信息详情页，会有更多的相关信息可供业务查看。

- 自助分析的取数场景标准化

除了固化场景的看板与数据集，自助取数是另一个让业务能快速利用数据解决业务分析的通道。但对于业务而言，自身对于数据的理解程及取数能力往往远低于门槛线，自助取数中的效率和质量都难以达到相对可用的标准。

自主分析的取数场景标准化就是通过固化相对标准且注释清晰的取数场景模板，简化业务方编写 SQL 代码的能力，通过简单参数的修改，一键查询即可跳转至自助取数平台进行业务分析。



三、总结与思考

数据管理是一个复杂的过程，在这个过程中绝不仅仅只是数据团队的努力，要将制定制度和实施细则，在组织内多个层次上实践数据管理，并参与组织变革管理工作，积极向组织传达改进数据治理的好处以及成功地将数据作为资产管理所必须的行为。

企业的数据战略必须和业务战略目标保持高度一致，即使拥有最佳的数据战略，数据治理和数据管理计划也可能不会成功，除非企业愿意接受并进行管理变革。数据治理越显著地帮助解决组织问题，才会有越来越多的人去接受改变、去接受数据治理实践。

携程度假的数据治理之路还很漫长，其中也参考了不少领域中优秀成熟的治理思路与方案，希望本文的一点点实践经验能给读者带来一点点的帮助。

框架架构

分布式缓存与 DB 秒级一致设计实践

【作者简介】大卫，携程服务端开发经理，对应用架构设计、云原生、代码整洁之道有浓厚兴趣。

一、前言

爆款项目是 2020 年携程的一个新项目，目标是将全品类、高性价比的旅行商品统一集合在一个频道供用户选购。出于这样的业务定位，项目有三个特点：

1）高流量
2）部分商品会成为热卖商品
3）承担下单职能

那么在系统设计之初，就必须考虑下面两个点：

1）如何应对高 QPS（包括整体高 QPS 和个别商品的高 QPS），高流量，保障 C 端用户体验？
2）在满足第一点的情况下，如何保障信息的时效性，让用户尽可能看到最新的信息，避免下单时的信息和看到的信息不一致？

很显然，要想较好的应对高 QPS，高流量的前端请求，需要借助缓存（我们使用了公司推荐的 Redis，后文不再做特别说明）。但是怎么使用好缓存解决上面两个问题，这是需要考虑的。我们对比了本文讨论的方案，和另外几个传统方案的优缺点，见下：

考量点/方案	本方案	应用层按需查数据没有时进行缓存	定期全量同步DB数据进入缓存	定期全量同步DB数据至Redis以及通过canal等中间件增量同步DB数据进入缓存
是否可以应对高QPS	可以	可以	可以	可以
是否可以解决热点key的问题	可以	不可以	不可以	不可以
缓存是否可以感知数据库中数据变化而快速更新	可以	不可以	不可以	可以
缓存数据更新的延迟	秒级	取决于缓存过期时间	取决于同步周期，通常是小时/天级别	秒级
缓存过期后能否及时重新写入缓存	可以	可以	不可以	不可以
新增/修改缓存时旧数据是否可能会覆盖新数据	不会	会	不会	会
是否支持有选择性的写入缓存，节省缓存集群资源	可以	可以	不可以	不可以
是否会周期性的遍历DB中需要缓存的数据表从而给DB带来额外压力	不会	不会	会	会
是否可以与特定业务解耦，从而被其他业务复用	可以	不可以	不能	不能
实现复杂度	较为复杂	简单	简单	中等

综上，本方案除了第一次实现有较高的复杂度外，带来的其他优势都是很可观的。目前线上运行下来，数据访问层应用相关的数据如下：

QPS：近万 QPS
性能：平均耗时个位数毫秒
缓存数据更新延时：秒级
Redis 集群的请求量：进行热点 key 处理后，减少原本 1/4 的请求量
Redis 集群内存占用：按需进行缓存，内存占用为传统方案的 1/10
Redis 集群 CPU 使用情况：整体平稳，未出现局部机器 CPU 异常

本文将从应用架构、缓存访问组件、缓存更新平台几方面介绍本方案。

二、应用架构

本方案的应用架构大致如下：

其中，浅绿色部分由业务实现，深绿色部分是本方案实现的。后文会详细介绍缓存访问组件和缓存更新平台的设计思路。

三、缓存访问组件

该组件的存在主要是为了封装缓存的访问。主要做了两件事：

1）按需异步将缓存中需要增、删、改的键值对通过消息传递给缓存更新平台，让其进行实际的缓存更新操作。
2）对热点 key 进行本地缓存与更新，避免对某个 key 的大量请求直接打到缓存导致缓存雪崩。

3.1 为什么要异步操作缓存？

这里，可能大家会有一个疑问，为什么要将简单的缓存操作由传统方案中的同步操作变为基于消息机制的异步操作呢？

这是由于我们的业务场景要求 DB 数据与缓存数据能够快速最终一致而决定的。如果采取传统的同步操作，那么极端情况下，可能会出现下面这样的多线程执行时序：

时刻（由近及远）	线程1	线程2	线程3
T1	发现key没有缓存数据，进而读取DB，得到数据v1		
T2		更新DB中key对应的数据，由v1变更为v2	
T3			发现key没有缓存数据，进而读取DB，得到数据v2
T4			将key<->v2写入缓存
T5	将key<->v1写入缓存		

可以发现，这样的时序执行完后，缓存中 key 对应的 value 是过期的 v1 而不是数据库中最新的 v2，这就会导致严重的用户体验问题，并且这个问题很难被发现。

那么我们是不是可以采用 Redis 的 SETNX 命令来解决这个问题呢？其实也是不行的。比如，上面的时序变为线程 1 先执行完，线程 3 再执行完，那么实际上缓存中的数据依然会是过期的 v1。因为线程 3 在采用 SETNX 命令设置缓存时，发现 key 已经有对应的值了，所以线程 3 最终的 SETNX 命令不会执行成功，也就导致了该更新的缓存反而没有更新。

不难看出，这类问题就是由于我们会有大量的并行操作同一个 key 导致的。所以，这里引入消息机制来异步执行缓存操作就是为了使同一个 key 的并行操作变为串行操作。

异步操作带来的问题

由于缓存操作由传统方案中的同步操作变为异步操作，那么引入了两个新问题：

1）如果投递消息失败了怎么办？
2）业务希望数据更新成功后缓存务必更新成功，也就是说希望 DB 数据更新和缓存更新近乎在一个事务里面，这该怎么办？

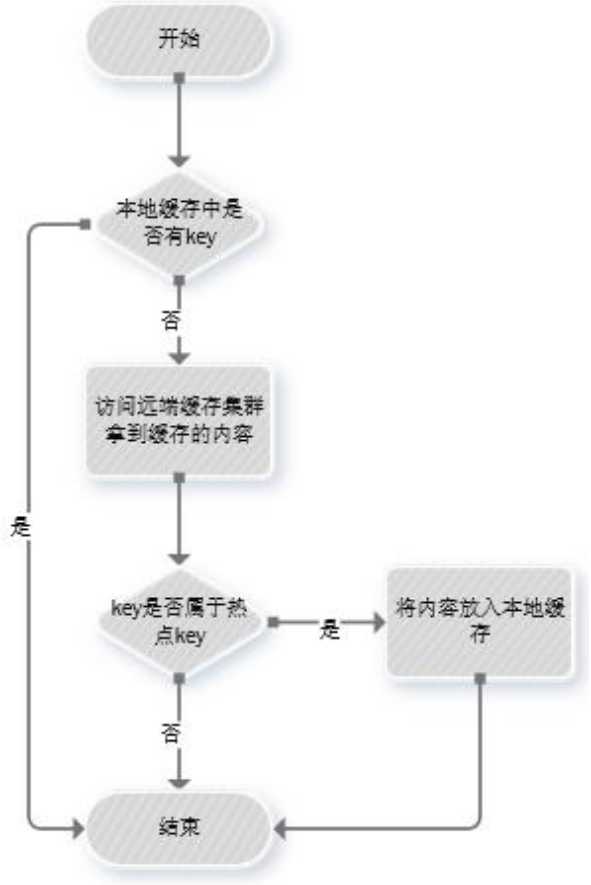
在这个组件中，我们通过引入一张存放于业务的 DB 的消息记录表来解决上述两个问题。它相当于是一个容灾方案，只要消息进入这张表，缓存更新平台就保证这条消息必然会被消费。

3.2 关于热点 key 的处理

该组件还有一大功能就是对热点 key 的处理。众所周知，缓存热点 key 在很多业务中都存在。例如若页面中存在长列表/瀑布流，那么第一屏的产品的访问量肯定比第二屏的产品访问量要高很多；又例如某些商品做活动，那么这类商品肯定要比没做活动的商品访问量高很多。

而爆款业务在可预见的未来，肯定也会出现热点 key 的问题。若热点 key 的问题不及时解决，当对单一 key 的请求量足够大时，可能导致缓存集群中存储该 key 的机器性能严重下降，从而导致缓存雪崩。所以在系统设计上，我们需要为解决热点 key 预留可扩展性。

目前组件内部热点 key 的处理流程如下：



通过上述流程可以看到，组件内部解决热点 key 主要是要解决下面三个问题：

1）如何判断是热点 key？
2）热点 key 如何存储？
3）热点 key 的内容如何更新？

3.2.1 如何判断是热点 key？

首先我们需要知道哪些 key 是热点 key 才能解决热点 key 的问题，识别热点 key 采取下面两个方案互补：

1）动态识别热点 key：主要针对部分 key 的访问流量增长相对平稳没那么陡的场景，使应用有能力应对线上一些无法预知的突发情况。

2）预设热点 key：主要针对定点开始的活动（比如电商的秒杀），这类流量增长通常会非常

陡且高峰很短暂。如果这种场景也采取方案 1 来主动识别通常就会导致滞后性，其实最终不会起到任何作用。所以我们就需要预设热点 key。

由于爆款业务处于起步阶段，场景 1 的问题尚不紧急，所以目前方案 1 我们计划在未来的迭代实现，这里不做过多讨论。

对于方案 2，业务目前可以主动将可以判定为热点的 key 灌给缓存访问组件。组件收到这类 key 后，当它在从缓存拿到这类 key 的内容后会主动将内容存入本地内存。后续所有的访问，都会从本地内存读取，从而大幅降低对远端缓存服务器的访问。

3.2.2 热点 key 如何存储?

在前文已经提到，针对热点 key，我们选择将其内容存放于应用服务器的内存中，这样做基于下面两个原因：

1）应用服务器本身一般都是以集群来部署，可以弹性缩扩容；
2）应用服务器的内存基本上可用空间都在 50%以上；

这样做带来的好处是：应用服务器在基于流量变化进行横向缩扩容时，热点 key 的内存与并发量的支持也跟着一起调整了，避免了多余的维护成本。

缓存访问组件在进行本地缓存时，考虑到热点 key 的访问流量通常是增长快下降也快，而且极端情况下可能出现本地缓存内容和数据库中的内容不一致，所以我们选择在本地进行一个很短时间的缓存，便于其能够应对突发的流量增长的同时也能在极端情况下快速与数据保持一致。

3.2.3 热点 key 的内容如何更新?

前文有提到，我们希望尽可能快的将数据库中最新的数据反映到缓存，热点 key 的本地缓存也不例外。所以，我们需要建立一个广播机制，让本地缓存能够知晓远端缓存的内容变化了。

这里，我们借助了缓存更新平台。由于所有的缓存更新都是发生在缓存更新平台（见后文），所以其可以将发生变化的缓存 key 通过消息队列广播给所有缓存访问组件，组件消费到这条消息后，若 key 是热点 key，则进行本地缓存的更新。极端情况下，可能会出现组件消费消息失败从而未更新的问题。针对这种情况，前文有提到，我们采取了很短时间的本地缓存，所以即便出现这个问题，也只会在较短时间有问题，最大程度保障了用户体验。

四、缓存更新平台

缓存更新平台主要有下面两大功能：

1）执行实际的缓存增、删、改命令；
2）缓存内容发生了变更后通知业务方；

由于缓存更新平台汇总了所有的缓存更新操作，所以它能够在缓存发生变更后，通过广播消息及时通知业务方，业务方拿到该消息后可以判断是否要做处理。目前这个功能主要用于解决热点 key 的内容更新问题，这在前文热点 key 处理的相关章节已做了详细说明，后文不再赘述。

后文主要介绍该平台的第一点功能。

前文有讲到，我们为了规避并行操作同一个 key 导致缓存中存储旧值而非最新值的问题，从而引入消息机制将缓存操作串行化。该缓存更新平台就用于串行的从消息队列消费缓存操作消息。

所以我们的核心需求是：单线程处理同一个 key 的缓存操作消息且不让旧的缓存覆盖新的缓存。

基于上面的需求，产生了四个问题：

1）怎么判断多个消息属于同一个 key 的缓存消息？
2）缓存操作的消息量级非常大（峰值情况下几十万条/分钟），怎么快速消费完？
3）怎么知道缓存内容是新还是旧，是否该对该消息进行处理？
4）由于基于消息，如何保障消息一定会被处理？

4.1 怎么判断多个消息是属于同一个 key 的缓存消息？

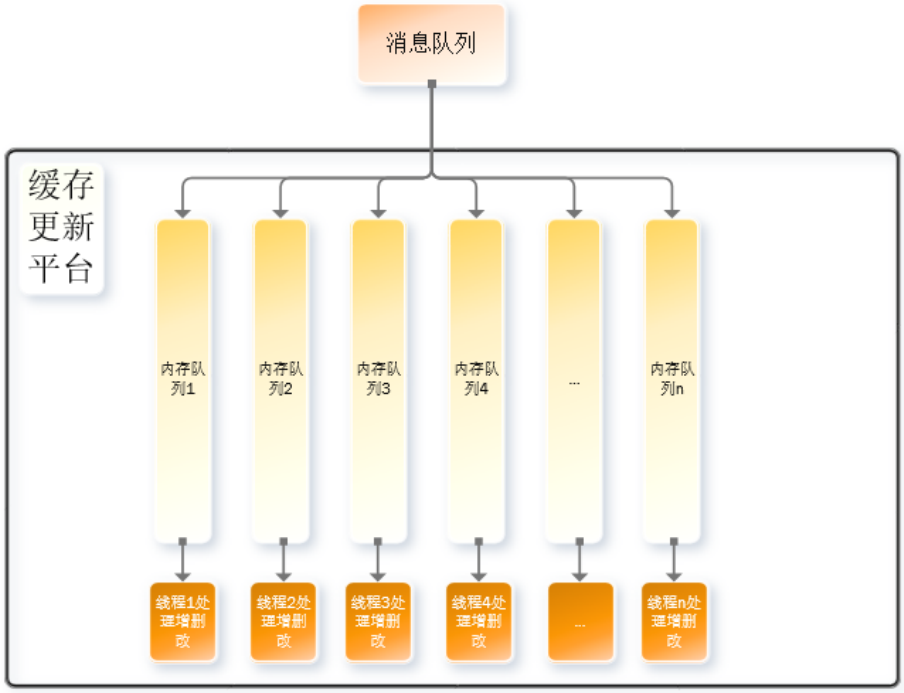
针对这个问题，我们通过在消息中携带缓存的 key 来解决这个问题，这样做带来了几个好处：

1）将业务和缓存更新平台解耦，key 的内容由业务全权决定；
2）通过首先计算 key 的 hash 值，然后对其取模，可以将相同的 key 分配到相同的线程处理（见后文）；
3）可扩展性强，针对后续热点 key 的分析和自动化加载热点 key 也起到了关键作用（后续迭代计划的功能）；

4.2 怎么快速消费消息？

由于核心需求是单线程的处理同一类 key 的消息，所以不同 key 的消息由不同的线程处理既能很好的解决性能问题，又不会产生逻辑问题。

我们采取了如下架构去消费产生的消息：



同一类 key 通过计算其 hash 值，然后再对结果进行取模，可以保证它们进到同一个内存队列和线程，从而规避并行操作同一个 key 的问题。通过这个架构，如果某个 key 的消息消费过慢，也不会影响其他 key 的消费进度，从而既保障了消费速度也满足了需求。

实践下来，目前我们仅用了两台机器就能做到每分钟消费几十万条消息，且远未遇到瓶颈。

4.3 怎么知道缓存内容是新还是旧，是否该对该消息进行处理?

虽然我们做到了同一类 key 的单线程处理，并且，我们使用的公司的消息队列能保障消息的有序性。但依然有个问题没解决，那就是旧的缓存可能会覆盖新的缓存，因为我们没法保障新的缓存消息一定在旧的缓存消息产生之后再产生。考虑下面这个场景:

时刻（由近及远）	线程1	线程2	线程3
T1	基于key，读到DB中的值为v1		
T2		基于key，更新DB中的值为v2	
T3			基于key，读到DB中的值为v2
T4			投递缓存内容为key<->v2的消息
T5	投递缓存内容为key<->v1的消息		

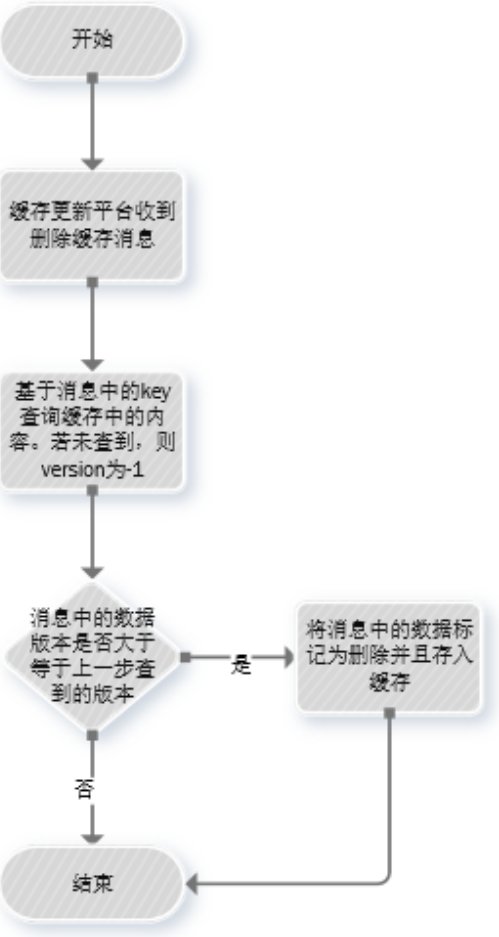
从上面可以看到，由于线程 3 的 key<->v2 消息先产生，所以它会被先消费，此时缓存的数据会变为 v2，然后缓存更新平台再处理 key<->v1 这条消息，从而导致 v1 覆盖缓存中的 v2，出现旧值覆盖新值的问题。

在这里，我们引入了缓存版本的概念来解决这个问题，我们认为每条缓存的数据都应该有一个版本号（业务提供，例如可以是修改数据的时间戳，只要满足单调递增即可）。基于此，缓存的增、删、改操作全部基于这个版本号来进行判断是否执行操作。具体的判断逻辑，在后文介绍。

缓存的增、删、改流程

删除缓存流程

先看下面流程图：



我们整个流程上是基于消息通知，这个消息生产的时机是只要业务删除了数据库中的数据就可以向缓存更新平台发送一条删除缓存消息。

从流程上可以看到，针对该消息的处理，流程里面并不是简单的删除一个 key，而是将删除的内容标记一下存入缓存。这样做带来了如下的好处：

1）能够避免缓存穿透；
2）能够避免缓存“复活”已经删除的数据；

如果我们简单的删除缓存中的内容而不是将被删除的内容标记起来存入缓存，那么当出现下面这个场景时，缓存中就会长期存在已经删除的数据，从而导致数据使用方误认为该数据仍然有效。

首先假设现在某个 key 在缓存中不存在。线程先消费了删除该 key 的消息且删除的数据版本是 v1，然后消费了存储缓存 key<->v1 的消息，这个时候就会将 key<->v1 写入缓存，但其实这个数据已经被删除了。

但即便将删除的内容放入缓存，考虑极端情况，仍然可能会有问题，考虑下面这个场景：

有两条邻近产生的消息：

消息 1：删除 key<->v1 的缓存消息
消息 2：新增 key<->v1 的缓存消息

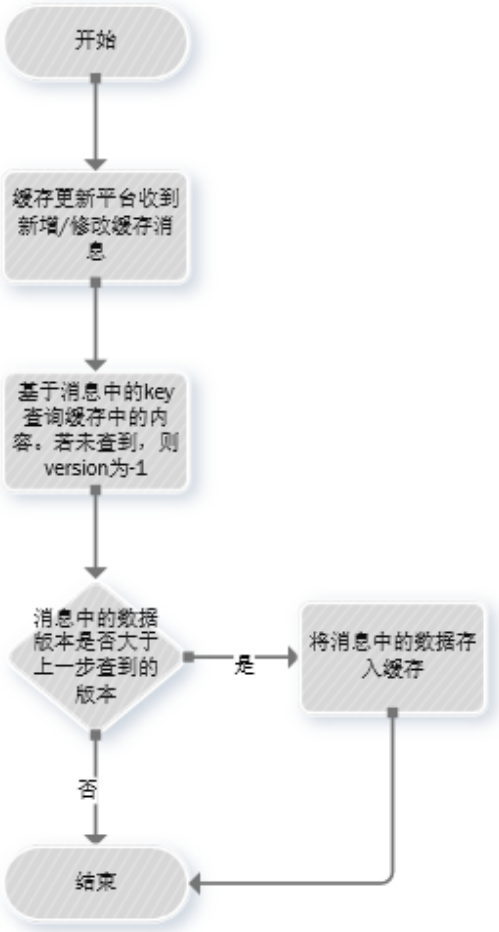
假设消费完消息 1 后，因为某种原因（如平台宕机或者消息队列出问题等等），消息 2 过了很久（缓存 key 已经过期）才被消费到，这时在缓存中存入该消息也会导致被删除的数据“复活”。所以针对这类情况，有两种措施：

1）缓存永久有效
2）超过一定时间未处理的消息就不处理了（我们采取的方案）

关于删除缓存消息中的版本，前文有提到，我们认为每条缓存数据都是有版本的。所以即便业务要删一条数据，那么被删的数据肯定也是有版本号的，而这个版本就是该条消息的版本。我们借助这个版本，就知道缓存中的数据是否是更新的版本，是否可以被覆盖并且被标记为删除了。

新增&修改缓存流程

新增缓存消息的处理流程和修改缓存消息的处理流程一致，见下：



首先，消息的生产时机是：

新增缓存消息：

- 业务往数据库中插入数据
- 业务流程因为缓存缺失导致直接访问到数据库的数据

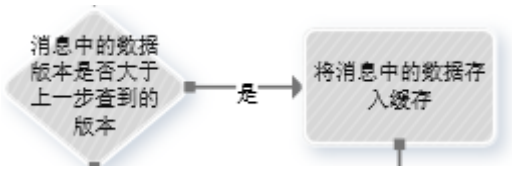
修改缓存消息：

业务修改了数据库中的数据

流程上可以看到，当缓存更新平台收到新增/修改缓存消息时，拿着消息中的 key 去查缓存，如果没有，则直接存入缓存；如果缓存中存在，则拿着缓存中的数据版本与消息的数据版本进行对比，如果消息中的数据版本更高（即更新），那么就可以安全覆盖缓存中的数据；反之，则不应该覆盖。通过这个流程，就可以很好的避免传统缓存更新里面经常出现的低版本数据覆盖缓存中高版本的数据。

新增&修改缓存流程与删除缓存流程大体一致，仅有一个区别点，如下：

新增&修改缓存：



删除缓存:

删除流程中关心的是消息中的版本是否大于等于缓存中的版本，而新增&修改缓存流程只关心消息中的版本是否大于缓存中的版本，为什么删除流程要关心版本相同的情况而新增&修改流程不关心呢?

针对删除,假设删除的数据对应的版本是 3,而缓存中正好也有这个数据且数据版本也是 3,这说明删除操作其实针对的是最新的数据，所以可以将缓存标记为删除态。

针对新增&修改，假设某条数据修改后，数据版本为 3。此时缓存里面正好也有版本为 3 的数据，那么缓存中的这条数据会有下面两种情况:

1）该数据在缓存中被标记为删除态了（即被业务删除了该数据）

若此时写入缓存，会导致删除态数据重新“复活”

2）该数据处于正常状态

若此时写入缓存，没有任何意义。

综上，无论针对上述哪种情况，只要不对这条消息进行处理，就不会有任何问题。所以，修改流程只有当消息中的版本高于缓存中的版本时才设置缓存。

4.4 如何保障消息一定会被处理?

由于整个平台依赖于消息队列中间件，那么如果消息队列中间件出了问题（如宕机/网络问题/消息投递失败等等）导致消费变得很慢或漏掉消息，怎么办?

前文提到，我们提供的缓存访问组件内部会将每条消息记录到业务 DB。缓存更新平台通过业务提供的接口增量轮询该表，确保所有消息都被及时消费掉。通过这样的容错措施，确保不会因为单点故障导致缓存来不及更新。

五、小结

可以看到，通过上述的缓存访问组件和缓存更新平台，可以做到缓存与数据库数据的快速一致，从而既保障了性能同时又最大程度的降低了用户看到过期数据的可能性。

接下来，我们将继续迭代，解决 1）减少缓存访问组件在业务代码上的侵入性；2）在缓存更新平台引入缓存 key 的分析机制，可以自动判定是否是热点 key 等问题。

携程最终一致和强一致性缓存实践

【作者简介】GSF，携程高级技术专家，关注高可用、高并发系统建设，致力于用技术驱动业务。

一、前言

携程金融从成立至今，整体架构经历了从 0 到 1 再到 10 的变化，其中有多个场景使用了缓存来提升服务质量。从系统层面看，使用缓存的目的无外乎缓解 DB 压力（主要是读压力），提升服务响应速度。引入缓存，就不可避免地引入了缓存与业务 DB 数据的一致性问题，而不同的业务场景，对数据一致性的要求也不同。本文将从以下两个场景介绍我们的一些缓存实践方案：

- 最终一致性分布式缓存场景
- 强一致性分布式缓存场景

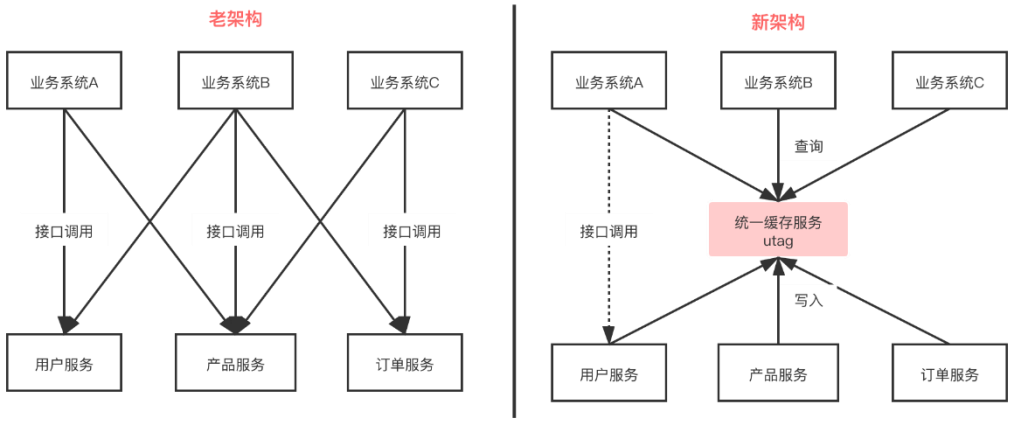
注：我们 DB 用的是 MySQL，缓存介质用的是携程高可用 Redis 服务，存储介质的选型及存储服务的高可用不是本文重点，后文也不再做特别说明。

二、最终一致性分布式缓存场景

2.1 场景描述

经过几年演进，携程金融形成了自顶向下的多层次系统架构，如业务层、平台层、基础服务层等，其中用户信息、产品信息、订单信息等基础数据由基础平台等底层系统产生，服务于所有的金融系统，对这部分基础数据我们引入了统一的缓存服务（系统名 utag），缓存数据有三大特点：全量、准实时、永久有效，在数据实时性要求不高的场景下，业务系统可直接调用统一的缓存查询接口。

我们的典型使用场景有：风控流程、APP 入口信息提示等，而对数据一致性要求高的场景依然需要走实时的业务接口查询。引入缓存前后系统架构对比如下：



统一缓存服务的构建给部门的整体系统架构带来了一些优势：

对业务系统：

- 响应速度提升：相比直接调用底层高流量的基础服务，调用缓存服务接口的系统响应时间大大减少（缓存查询接口 P98 为 10 毫秒）。
- 统一接口，降低接入成本：一部分业务场景下可以直接调用统一缓存服务查询接口，而不用再对接底层的多个子系统，极大地降低了各个业务线的接入成本。
- 统一缓存，省去各个服务单独维护缓存的成本。

对基础服务：

- 服务压力降低：基础平台的系统本身就属于高流量系统，可减少一大部分的查询流量，降低服务压力。

整体而言，缓存服务处于中间层，数据的写入方和数据查询方解耦，数据甚至可以在底层系统不感知的情况下写入（见下文），而数据使用方的查询也可在底层服务不可用或“堵塞”时候仍然保持可用（前提是缓存服务是可用的，而缓存服务的处理逻辑简单、统一且有多种手段保证，其可用性比单个子系统都高），整体上服务的稳定性得到了提升。

在构建此统一缓存服务时候，有三个关键目标：

- 数据准确性：DB 中单条数据的更新一定要准确同步到缓存服务。
- 数据完整性：将对应 DB 表的全量数据进行缓存且永久有效，从而可以替代对应的 DB 查询。
- 系统可用性：我们多个产品线的多个核心服务都已经接入，utag 的高可用性显的尤为关键。

接下来先说明统一缓存服务的整体方案，再逐一介绍此三个关键特性的设计实现方案。

2.2 整体方案

我们的系统在多地都有部署，故缓存服务也做了相应的异地多机房部署，一来可以让不同地区的服务调用本地区服务，无需跨越网络专线，二来也可以作为一种灾备方案，增加可用性。

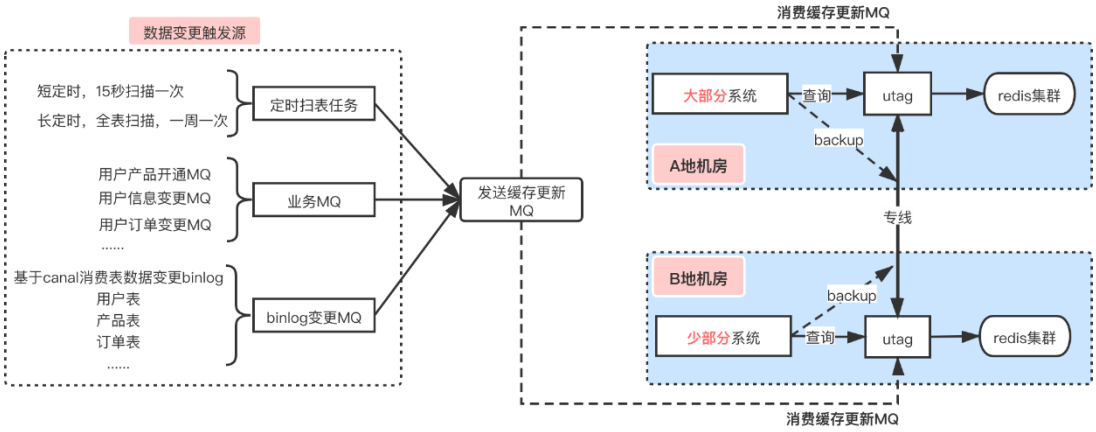
对于缓存的写入，由于缓存服务是独立部署的，因此需要感知业务 DB 数据变更然后触发缓存的更新，本着“可以多次更新，但不能漏更新”的原则，我们设计了多种数据更新触发源：定时任务扫描，业务系统 MQ、binlog 变更 MQ，相互之间作为互补来保证数据不会漏更新。

此外为了缓存更新流程的统一和与触发源的解耦，我们使用 MQ 来驱动多地多机房的缓存更新，在不同的触发源触发后，会查询最新的 DB 数据，然后发出一个缓存更新的 MQ 消息，不同地区机房的缓存系统同时监听该主题并各自进行缓存的更新。对于 MQ 我们使用携程开源消息中间件 QMQ 和 Kafka，在公司内部 QMQ 和 Kafka 也做了异地机房的互通。

对于缓存的读取，utag 系统提供 dubbo 协议的缓存查询接口，业务系统可调用本地区的接口，省去了网络专线的耗时（50ms 延迟）。在 utag 内部查询 redis 数据，并反序列化为对应的业务 model，再通过接口返回给业务方。

为了描述方便，以下异地多机房部署统一使用 AB 两地部署的概念进行说明。

整体框架如下图所示：



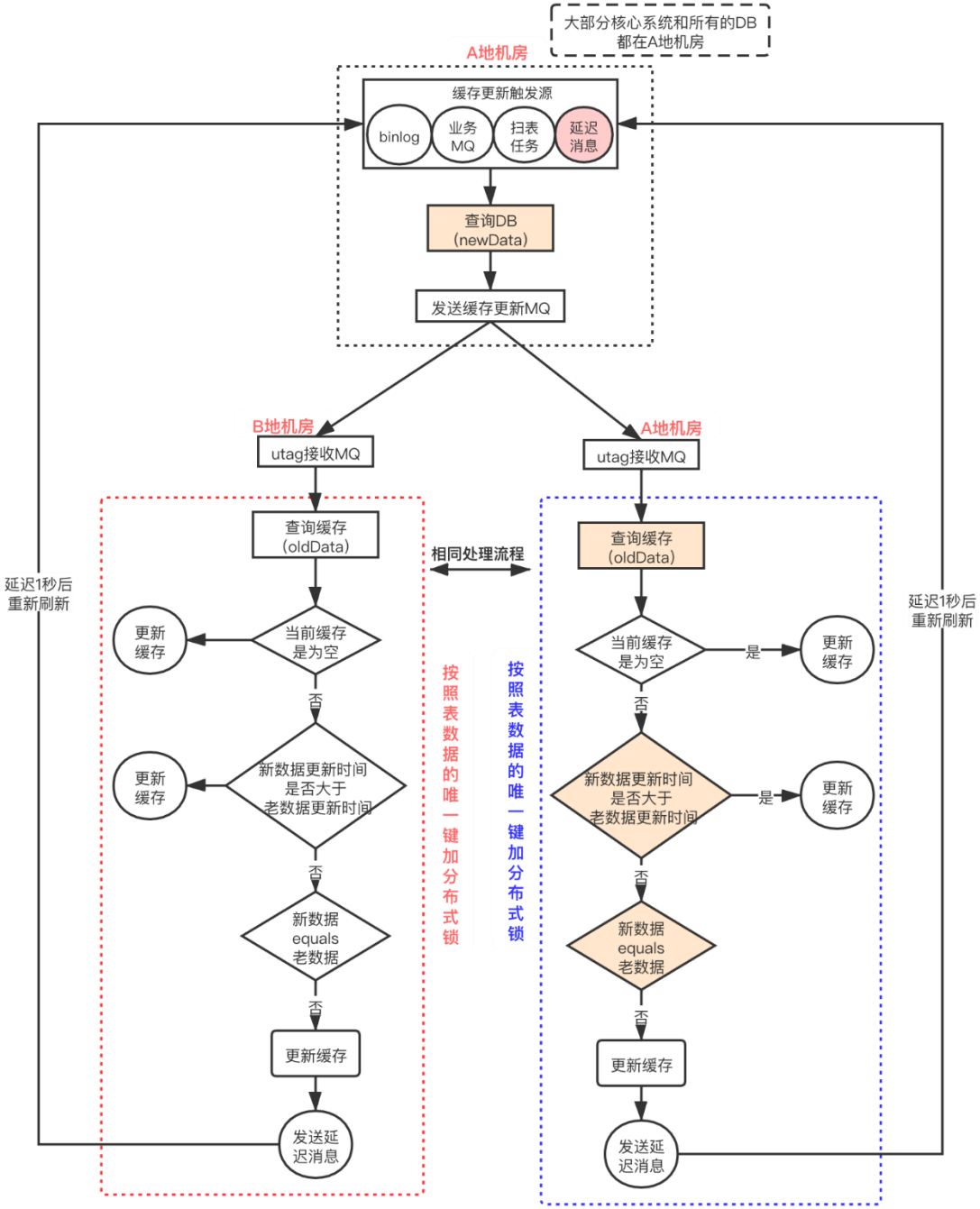
接下来介绍一下几个关键点的设计 。

2.3 数据准确性设计

不同的触发源，对缓存更新过程是一样的，整个更新步骤可抽象为 4 步：

- step1：触发更新，查询 DB 中的新数据，并发送统一的 MQ
- step2：接收 MQ，查询缓存中的老数据
- step3：新老数据对比，判断是否需要更新
- step4：若需要，则更新缓存

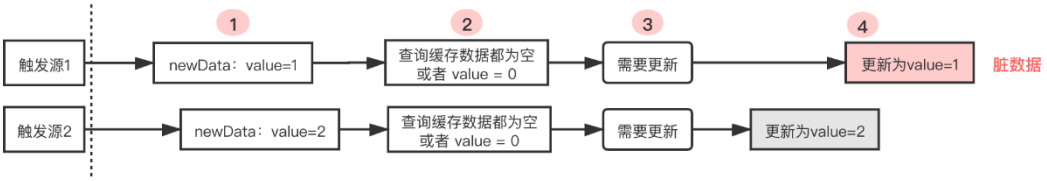
由于我们业务的大部分核心系统和所有的 DB 都在 A 地机房，所以触发源（如 binlog 的消费、业务 MQ 的接收、扫表任务的执行）都在 A 侧，触发更新后，第一步查询 DB 数据也只能在 A 侧查询（避免跨网络专线的数据库连接，影响性能）。查询到新数据后，发送更新缓存的 MQ，两地机房的 utag 服务进行消费，之后进行统一的缓存更新流程。总体的缓存更新方案如下图所示：



由于有多个触发源，不同的触发源之间可能会对同一条数据的缓存更新请求出现并发，此外可能出现同一条数据在极短时间内（如 1 秒内）更新多次，无法区分数据更新顺序，因此需要做两方面的操作来确保数据更新的准确性。

（1）并发控制

若一条 DB 数据出现了多次更新，且刚好被不同的触发源触发，更新缓存时候若未加控制，可能出现数据更新错乱，如下图所示：



故需要将第 2、3、4 步加锁，使得缓存刷新操作全部串行化。由于 utag 本身就依赖了 redis，此处我们的分布式锁就基于 redis 实现。

（2）基于 updateTime 的更新顺序控制

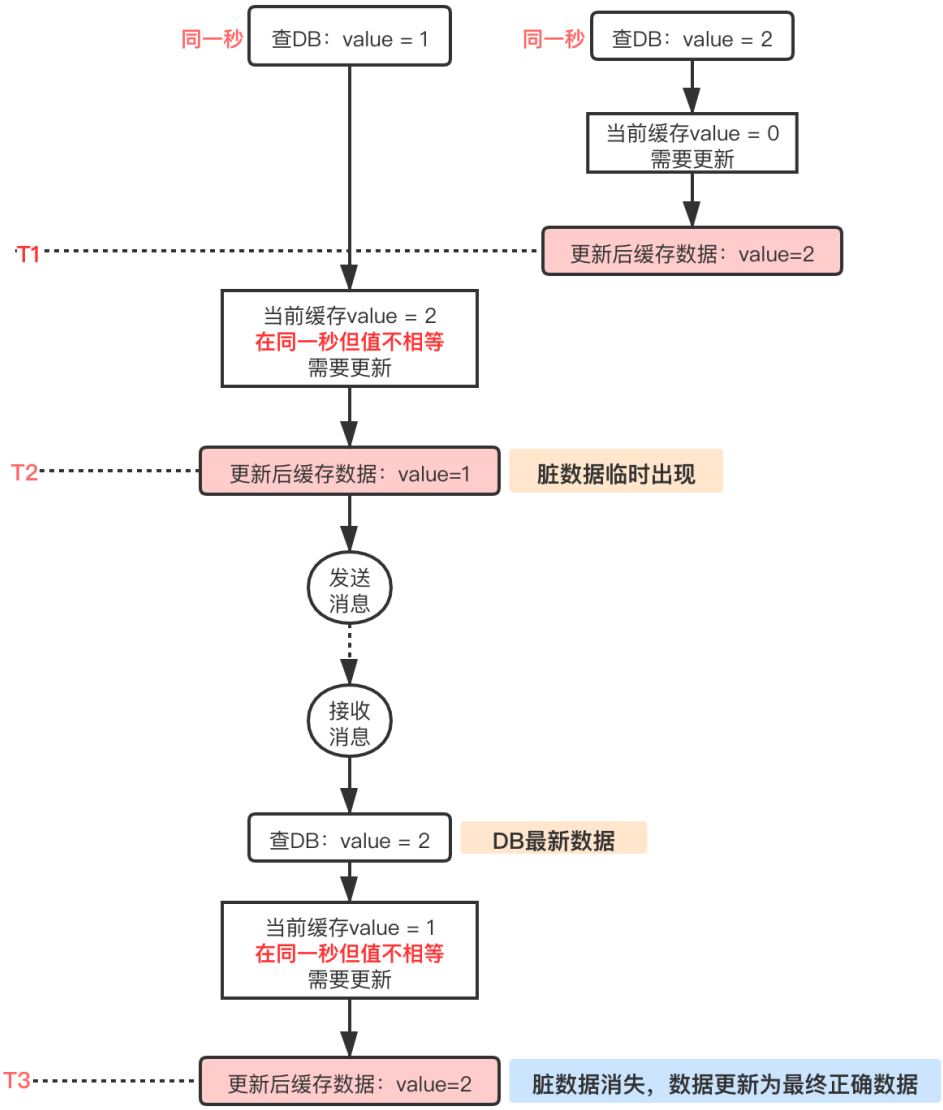
即使加了锁，也需要进一步判断当前 db 数据与缓存数据的新老，因为到达缓存更新流程的顺序并不代表数据的真正更新顺序。我们通过对比新老数据的更新时间来实现数据更新顺序的控制。若新数据的更新时间大于老数据的更新时间，则认为当前数据可以直接写入缓存。

我们系统从建立之初就有自己的 MySQL 规范，每张表都必须有 update_time 字段，且设置为 ON UPDATE CURRENT_TIMESTAMP，但是并没有约束时间字段的精度，大部分都是秒级别的，因此在同一秒内的多次更新操作就无法识别出数据的新老。

针对同一秒数据的更新策略我们采用的方案是：先进行数据对比，若当前数据与缓存数据不相等，则直接更新，并且发送一条延迟消息，延迟 1 秒后再次触发更新流程。

举个例子：假设同一秒内同一条数据出现了两次更新，value=1 和 value=2，期望最终缓存中的数据是 value=2。若这两次更新后的数据被先后触发，分两种情况：

- case1：若 value=1 先更新，value=2 后更新，（两者都可更新到缓存中，因为虽然是同一秒，但是值不相等）则缓存中最终数据为 value=2。
- case2：若 value=2 先更新，value=1 后更新，则第一轮更新后缓存数据为 value=1，不是期望数据，之后对比发现是同一秒数据后会通过消息触发二次更新，重新查询 DB 数据为 value=2，可以更新到缓存中。如下图所示：



通过以上方案我们可以确保缓存数据的准确性。有几个点需要额外说明：

- 为什么要用延迟消息？

其实不用延迟消息也是可以的，毕竟 DB 数据的更新时间是不变的，但是考虑到出现同一秒更新的可能是高频更新场景，若直接发消息，然后立即消费并触发二次更新，可能依然查到同一秒内更新的其他数据，为减少此种情况下的多次循环更新，延迟几秒再刷新可作为一种优化策略。

- 不支持 db 的删除操作

因为删除操作和 update 操作无法进行数据对比，无法确定操作的先后顺序，进而可能导致更新错乱。而在数据异常宝贵的时代，一般的业务系统中也没有物理删除的逻辑。

- 若当前 db 没有设置更新时间该如何处理？

可以将查 DB、查缓存、数据对比、更新缓存这四个步骤全部放到锁的范围内，这样就不需要处理同一秒的顺序问题。因为在这个串行化操作中每次都从 DB 中查询到了最新的数据，可以直接更新，而时间的判断、值的判断可以作为优化操作，减少缓存的更新次数，也可以减少锁定的时间。

而我们为何不采用该方案？因为查询 DB 的操作我们只能在一侧机房处理，无法让 AB 两地系统的更新流程统一，也就降低了二者互备的可能性。

- 其他方案
- DB 数据模型上增加版本字段，可严格控制数据的更新时序。
- 将 update_time 字段精度设置为精确到毫秒或微秒，提升数据对比的准确度，但是相比增加版本字段的方案，依然存在同一时间有多次更新的可能。
- 这些额外的方案，都需要业务数据模型做对应的支持。

2.4 数据完整性设计

上述数据准确性是从单条数据更新角度的设计，而我们构建缓存服务的目的是替代对应 DB 表的查询，因此需要缓存对应 DB 表的全量数据，而数据的完整性从以下三个方面得到保证：

（1）“把鸡蛋放到多个篮子里”，使用多种触发源（定时任务，业务 MQ，binglog MQ）来最大限度降低单条数据更新缺失的可能性。

单一触发源有可能出现问题，比如消息类的触发依赖业务系统、中间件 canel、中间件 QMQ 和 Kafka，扫表任务依赖分布式调度平台、MySQL 等。中间任何一环都可能出现问题，而这些中间服务同时出概率的可能相对来说就极小了，相互之间可以作为互补。

（2）全量数据刷新任务：全表扫描定时任务，每周执行一次来进行兜底，确保缓存数据的全量准确同步。

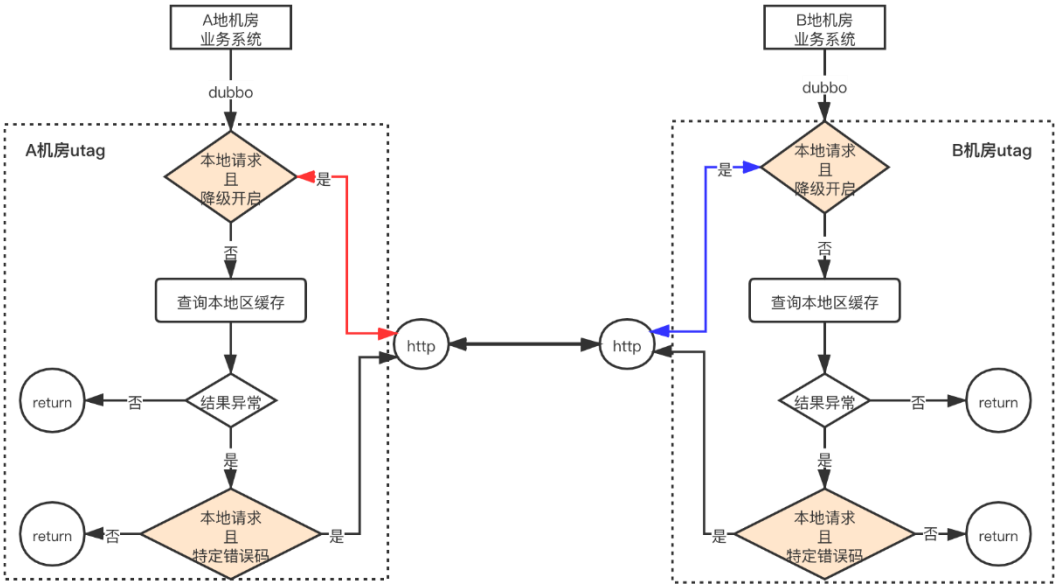
（3）数据校验任务：监控 Redis 和 DB 数据是否同步并进行补偿。

2.5 系统可用性设计

统一缓存服务被多个业务线的核心系统所依赖，所以缓存服务的高可用是至关重要的。而对高可用的建设，除了集群部署、容量规划、熔断降级等常用手段外，针对我们自己的场景也做了一些方案。主要有以下三点：

（1）异地机房互备

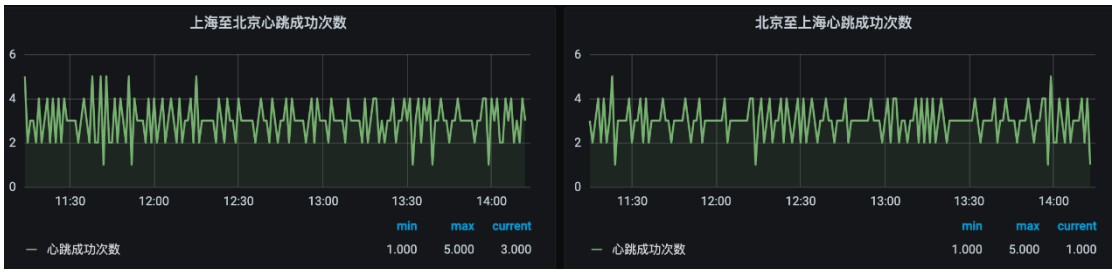
如上所述，我们的服务在 AB 两地部署，两机房的缓存通过两地互通的 MQ 同时写入。在这套机制下，本地区的业务系统可以直接读取本地区的缓存，如果出现了本地区 utag 应用异常或 redis 服务异常，则可以快速降级到调用另外机房的服务接口。具体方案如下图所示：



本地业务系统通过 dubbo 调用本地的 utag 服务，在 utag 的本地处理流程中，查询本地缓存前后分别可根据一定的条件进行服务降级，即查询另一机房。

- 查询本地缓存前降级：若本地 redis 集群出现故障，可以在配置平台人工快速切换到查询另一侧的服务。
- 查询本地缓存后降级：本地处理结束，若出现特定错误码（"OPERATE_REDIS_ERROR"）则可降级到查询另一侧服务。该功能也需要手工配置开关来启用。

为了避免循环调用，在降级调用前，需要判断当前请求是否来自本地，而此功能通过 Dubbo 的 RpcContext 透传特定标识来实现。除此之外，还建立了两机房的应用心跳，来辅助切换。



（2）QMQ 和 Kafka 互备

缓存更新流程通过 MQ 来驱动，虽然公司的 MQ 中间件服务由专人维护，但是万一出现问题长时间不能恢复，对我们来说将是致命的。所以我们决定同时采用 Kafka 和 QMQ 两种中间件来作为互备方案。默认情况下对于全表扫描任务和 binlog 消费这类大批量消息场景使用 Kafka 来驱动，而其他场景通过 QMQ 来驱动。所有的场景都可以通过开关来控制走 Kafka 或者 QMQ。目前该功能可通过配置管理平台来实现快速切换。

（3）快速恢复

在极端情况下，可能出现 Redis 数据丢失的情况，如主机房（A 机房）突然断网，redis 集群切换过程出现数据丢失或同步错乱，此时很可能无法通过自动触发来补齐数据，因此设计了全表快速扫描的补偿机制，通过多任务并行调度，可在 30 分钟内将全量数据完成刷新。此功能需要人工判断并触发。

2.6 总结

以上介绍了我们最终一致性分布式缓存服务的设计思路和要点，其中的关键点为数据准确性、数据完整性、系统可用性的设计。除此之外，还有一些优化点如降级方案的自动触发、异地机房缓存之间、缓存与 DB 之间做旁路数据 diff，可进一步确保缓存服务整体的健壮性，在后续的版本中进行迭代。

三、强一致性分布式缓存场景

3.1 场景描述

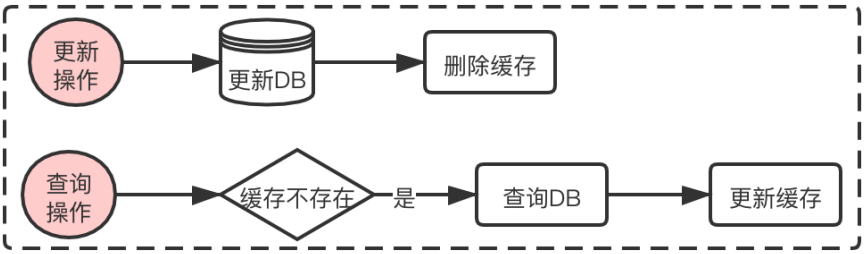
强一致性分布式缓存目前主要应用在我们携程金融的消金贷前服务中。随着我们用户量和业务量的增涨，贷前服务的查询量激增，给数据库带来了很大的压力，解决此问题有几种可选方案：

（1）分库分表：成本和复杂度相对较高，我们场景下只是数据查询流量较大。
（2）读写分离：出于数据库性能考虑，我们的 MySQL 大部分采用异步复制的方式，而由于我们的场景对数据实时性要求较高，因此无法直接利用读写分离的优势来分担主库压力。

综合来看，增加缓存是更加合适的方案，我们决定设计一套高可用的满足强一致性要求的分布式缓存。接下来介绍我们的具体设计实现方案。

3.2 整体方案

缓存的处理我们采用了较为常见的处理思路：在更新操作中，先更新数据库，再删除缓存，查询操作中，触发缓存更新。

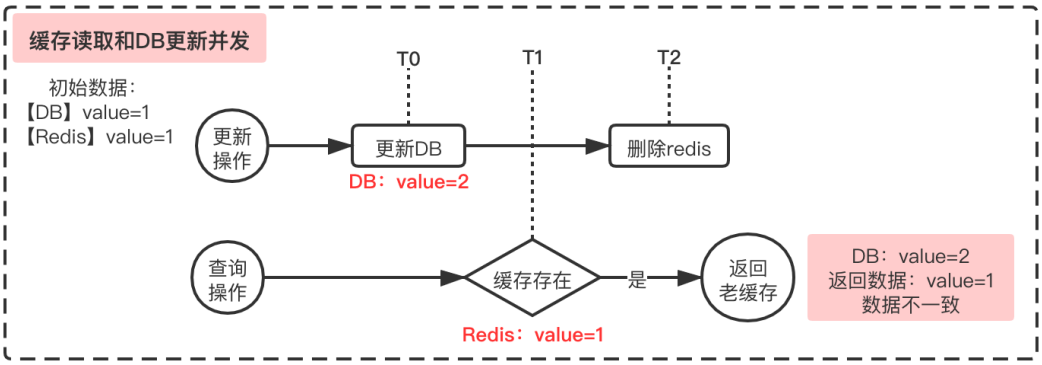


在此过程中，若不加控制，则会存在数据不一致性问题，主要是由于缓存操作和 DB 更新之

间的并发导致的。具体分析如下：

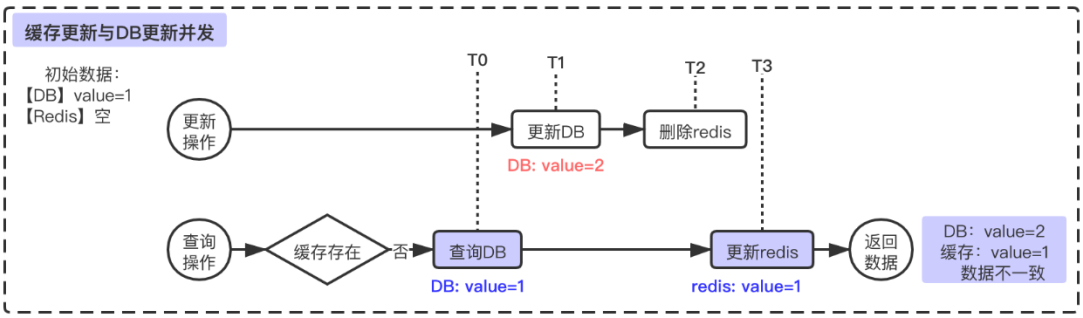
（1）缓存读取和 DB 更新并发

如下图所示，查询时候若缓存已经存在，则会直接返回缓存数据。若查询缓存的操作，发生在“更新 DB 数据”和“删除缓存”之间，则本次查询到数据为缓存中的老数据，导致不一致。当然下次查询可能就会查询到最新的数据。这种并发在我们服务中是存在的，比如某个产品开通后，会在更新 DB（产品开通状态）后立即发送 MQ（事务型消息）告知业务，业务侧处理流程中会立即发起查询操作。此场景中数据库的更新和数据的查询间隔极短，很容易出现此种并发问题。



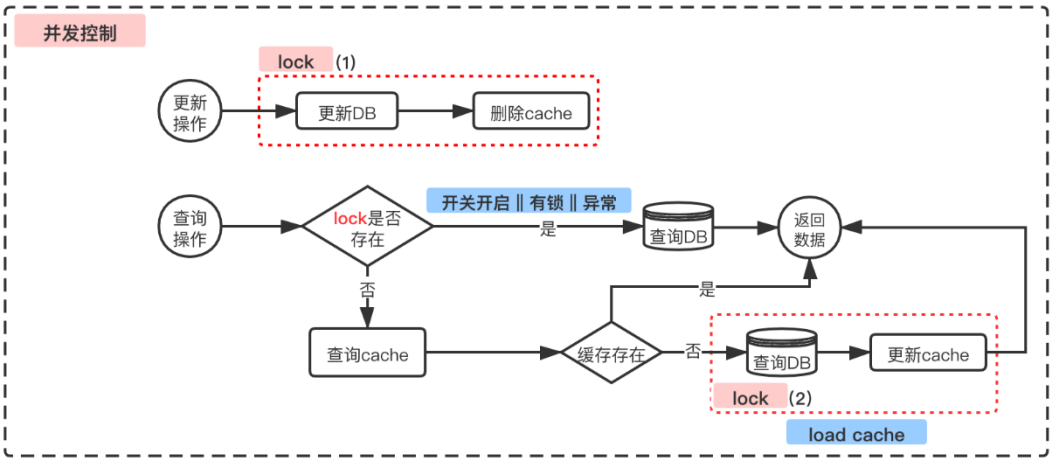
（2）缓存更新与 DB 更新并发

如下图所示，查询的时候，若缓存不存在，则更新缓存，流程是先查询 DB 再更新 Redis。若更新缓存时候，出现以下时序：查询 DB 老数据（T0 时刻，DB 中 value=1）→ 更新 DB（T1 时刻，更新 DB 为 value=2）→ 删除 Redis（T2）→ 更新 Redis（T3），则会导致本次查询返回数据及缓存中的数据与 DB 数据不一致，即接口返回和更新后的缓存都为脏数据。若 T2 和 T3 互换，即更新 DB 后，先更新 Redis，再删除 Redis ，由于缓存被删除在下次更新可能会被正确更新，但本次返回数据依然与 DB 更新后的数据不一致。



基于以上分析，为了避免并发带来的缓存不一致问题，需要将"更新 DB"+"删除缓存"、"查询 DB"+"更新缓存"两个流程都进行加锁。此处需要加的是分布式锁，我们使用的是 redis 分布

式锁实现。加锁后的读写整体流程如下：



如上图所示，有两处加锁，更新 DB 时加锁，锁范围为"更新 DB"+"删除 cache"（图中 lock1），更新缓存时加锁，锁范围为"查询 DB" + "更新 cache"（图中 lock2），两处对应的锁 key 是相同的。基于此方案，对于上面所说的两种并发场景，做针对性分析如下：

（1）缓存查询和 DB 更新的并发控制

查询操作流程中，先判断 lock 是否存在，若存在，则表示当前 DB 或缓存正在更新，不能直接查询缓存，在查询 DB 后返回数据。之所以这么做，还是由场景决定的，如前文所述，我们场景下的基本处理思路是，缓存仅作为“DB 降压”的辅助手段，在不确定缓存数据是否最新的情况下，宁可多查询几次 DB，也不要查询到缓存中的不一致数据。此外，更新操作相对于查询操作是很少的，在我们贷前服务中，读写比例约为 8:1。

此处另外的一个可行方案是可在检测到有锁后可进行短暂的等待和重试，好处是可进一步增加缓存的命中率，但是多一次锁等待，可能会影响到查询接口的性能。可根据自身场景进行抉择。

此外，为了进行降级，在锁判断前也增加了降级开关判断，若降级开关开启，也会直接查询 DB。而降级主要是由于 redis 故障引起的，下文详述。若检测是否有锁时发生了异常同样也会直接查询 DB。

（2）缓存更新和 DB 更新的并发控制

查询操作流程中，若缓存不存在，则进行缓存的更新，在更新时候先尝试进行加锁，若当前有锁说明当前有 DB 或缓存正在更新，则进行等待和重试，从而可避免查询到 DB 中的老数据更新到缓存中。

其中 lock2 的流程（load cache），我们是同步进行的。另外一个可行的方案是，异步发起缓存的加载，可减少锁等待时间，但是若出现瞬时的高并发查询，可能缓存无法及时加载产生

从而频繁产生瞬时压力。可根据自身场景进行抉择。

以上为我们的整体设计思路，接下来从实现的角度分别描述一下基于本地消息表的缓存删除策略，缓存的降级和恢复这两个方面的具体方案。

3.3 缓存删除策略

在更新操作中，在锁的范围内，先更新 DB，再删除缓存。

其中锁的选型，我们采用与缓存同介质的 redis 分布式锁，这样做的好处是若因为 redis 服务不可用导致的锁处理失败，对于缓存本身也就不可用，可以自动走降级方案。

此外，更新流程还要考虑两点：锁的范围和删除缓存失败后如何补偿。

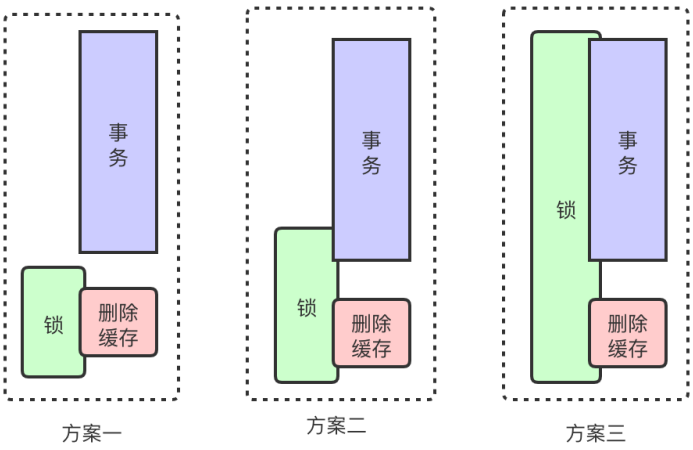
（1）锁粒度

更新操作中加锁粒度有以下三种方案：

方案一：事务提交后加锁，只锁定删除缓存操作。对原事务无任何额外影响，但是在事务提交后到删除缓存之间存在与查询的并发可能性。

方案二：在事务提交前加锁，删除缓存后解锁。在满足一致性要求的前提下，锁的粒度可以做到最小，但是增加了 DB 事务的范围，若 redis 出现超时则可能导致事务时间拉长，进而影响 DB 操作性能。

方案三：在事务开始前加锁，删除缓存后解锁。锁的范围较大，但是能满足我们一致性要求，对单个 DB 事务也基本无影响。且对同一个用户来说，贷前数据的更新并不频繁，锁范围稍大一些是我们可以接受的。



立足自身场景，权衡一致性要求和服务性能要求，我们剔除了方案二，默认情况下使用方案

三，但是若在事务开始前加锁失败，为了不影响原业务流程（缓存只是辅助方案，redis 故障不影响原应用功能）会自动降级到方案一，即在事务提交后删除缓存前再加锁。而这种降级，若出现并发的查询操作，依然可能出现上述不一致的问题，但是是可以容忍的，原因如下：

通常情况下加锁失败是由于操作 redis 异常或者锁竞争引起的。

- 若出现 redis 异常，同时出现了并发的查询，而并发的两个操作时间间隔是极短的，因此查询时候，锁检测操作通常也是异常的，此时查询会自动降级为查询 DB。
- 若极短时间内的 redis 集群抖动，事务执行前 redis 不可用，事务执行后 redis 恢复，而此时在加锁操作还没有完成前恰巧又进行了并发的查询操作，检测锁成功且锁不存在，才可能会出现查询出老数据的情况。这种是极其严苛的并发条件。
- 而加锁过程会进行重试（可动态调整配置），多次重试后可解决大部分的锁竞争情况。

综上，在上述锁降级的方案下，数据不一致出现的情况虽然无法完全避免，但是产生条件极其苛刻，而应对这种极其极端的情况，在系统层面做更加强的方案带来的复杂度提升与收益是不成正比的，一般情况下做好日志记录、监控、报警，通过人工介入来弥补即可。从该方案上线后至今两年多的时间内，没有出现过该情况。

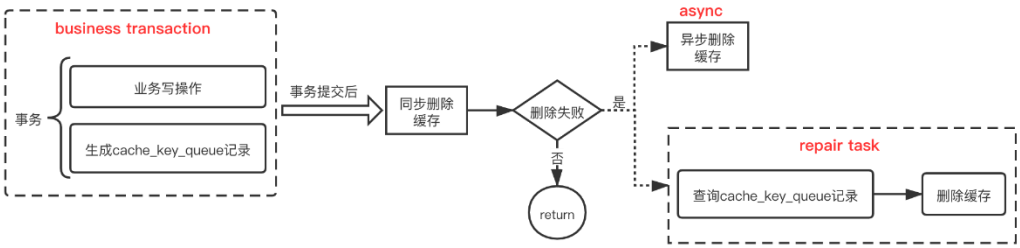
（2）删除缓存失败的补偿

另外要考虑的问题是，如果更新 DB 成功但删除缓存失败要后如何处理，而此种情况往往因应用服务器故障、网络故障、redis 故障等原因导致。

若应用服务器突然故障，则服务整体不可用，跟缓存就没多大关系了。若是由于网络、redis 故障等原因导致的删除缓存失败，此时查询缓存也不可用，查询走 DB，但需要可靠地记录下哪些数据做了变更，待 redis 可用后需要进行恢复，需要将中间变更的记录对应的缓存全部删除。

此处的一个关键点在于数据变更的可靠性记录，受到 QMQ 事务消息实现方案的启发，我们的方案是构建一张简易的记录表（代表发生变更的 DB 数据），每次 DB 变更后，将该变更记录表的插入和业务 DB 操作放在一个事务中处理。事务提交后，对应的变更记录持久化，之后进行删除缓存，若缓存删除成功，则将对应的记录表数据也删除掉。若缓存删除失败，则可根据记录表的数据进行补偿删除，而在 redis 的恢复流程中，需要校验记录表中是否存在数据，若存在则表示有变更后的数据对应的缓存未清除，不可进行缓存读取的恢复。

此外删除操作还要进行异步重试，来避免偶尔超时引起的缓存删除失败。此方案整体流程如下：



其中 cache_key_queue 表即为我们的变更记录表，放在业务的同 DB 内。其表结构非常简单，只有插入和删除操作，对业务 DB 的额外影响可以忽略。

```
CREATE TABLE `cache_key_queue` (
    `id` bigint(20) UNSIGNED NOT NULL AUTO_INCREMENT COMMENT '主键',
    `cache_key` varchar(1024) NOT NULL DEFAULT '' COMMENT '待删除的缓存 key',
    `create_time` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP COMMENT '创建时间',
    PRIMARY KEY (`id`)
) ENGINE = InnoDB AUTO_INCREMENT = 0 CHARSET = utf8 COMMENT '缓存删除队列表'
```

基于以上分析，为了锁范围尽可能小，且为了尽可能降低极端的 redis 抖动情况下产生的影响，我们期望可以在事务提交后立即触发缓存的删除操作。为了能够对 redis 不可用期间发生变更的数据进行清除，我们需要可靠地记录数据变更记录。幸运的是，基于 Spring 的事务同步机制 TransactionSynchronization，可以很容易实现该方案。简单来说，该机制提供了 Spring 环境中事务执行前后的 AOP 功能，可以在 spring 事务的执行前后添加自己的操作，如下所示（代码和注释经过了简化）：

```
public interface TransactionSynchronization extends Flushable {
    /**
     * 事务同步器挂起处理
     */
    void suspend();

    /**
     * 事务同步器恢复处理
     */
    void resume();

    /**
     * 事务提交前处理
     */
    void beforeCommit(boolean readOnly);

    /**
```

```
     * 事务完成(提交或回滚)前处理
     */
    void beforeCompletion();

    /**
     * 事务提交后处理
     */
    void afterCommit();

    /**
     * 事务完成(提交或回滚)后处理
     */
    void afterCompletion(int status);
}
```

基于此机制，我们可以很方便且相对优雅地实现我们的设计思路，即在 beforeCommit 方法中，插入 cache_key_queue 记录；在 afterCommit 方法中同步删除缓存，若删除缓存成功则同步删除 cache_key_queue 表记录；在 afterCompletion 方法中进行锁的释放处理。若同步删除缓存失败，则 cache_key_queue 表记录也会保留下来，之后触发异步删除，并启动定时任务不断扫描 cache_key_queue 表进行缓存删除的补偿。需要注意的是可能存在嵌套事务，一个完整事务中，可能存在多次数据更新，可借助 ThreadLocal 进行多条更新记录的汇总。

3.4 缓存的熔断和恢复

除了上述锁处理流程中讨论的 redis 抖动问题外，还需要考虑缓存服务 redis 集群不可用（网络问题、redis 集群问题等）。按照我们的基本原则，引入的缓存服务仅做辅助，并不能强依赖。如果缓存不可用，主业务依然要保持可用，这就是我们接下来要讨论的缓存的熔断和恢复。

（1）缓存熔断

熔断的目的是在 redis 不可用时避免每次调用（查询或更新）都进行额外的缓存操作，这些缓存操作会进行多次尝试，比如加锁操作我们设置的自动重试 3 次，每次间隔 50ms，总耗时会增加 150ms。若 redis 不可用则每次调用的耗时都会有额外增加，这对主业务功能可能会产生影响，降低底层服务的质量和性能。因此我们需要识别出 redis 不可用的情况，并进行熔断。

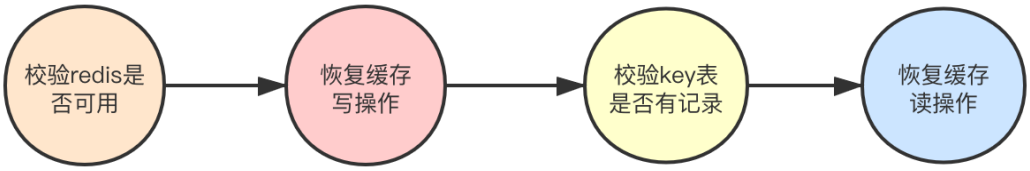
我们的熔断判断逻辑为：每个 redis 操作都 try-catch 异常，并做计数统计（不区分读写操作），若在 M 秒内出现 N 次异常则开启熔断。我们的场景下设置为 10 秒内出现 50 次异常就熔断，可根据自身场景设置，需要注意的是如果 redis 请求次数比较少，则需要在配置上保证在 M 秒内至少出现 N 次请求。

此外熔断开关的配置是放在应用服务器的内存中，即单机熔断，而非集群熔断，这样做的原因是，redis 服务不可用有可能是单机与 redis 服务的连通性问题导致，而在其他机器上依然

可以访问缓存。

（2）缓存恢复

熔断之后的恢复策略相对复杂一些，需要区分缓存的读操作恢复和写操作恢复。具体如下流程如下：



step1：校验 redis 是否可用

判断逻辑为，连续发起特定的 set 操作 N 次，每次间隔一定时间，若都成功，则认为 redis 恢复。

此处需要注意的是，我们的 redis 集群是 Cluster 模式，不同的 key 会散落在不同的 redis 主节点上，因此最保险的做法是判断当前集群中所有的主节点都恢复才认为操作恢复，而简单的做法是每次探测恢复的 set 操作都设置不同的 key 以求能尽可能散列到不同的节点去。可按照自身场景进行方案抉择。

step2：恢复缓存写操作

若 redis 恢复，缓存的写操作就可以恢复了。即可在更新操作中进行加锁、更新 DB、删除缓存。但是此时读操作还不能立即恢复，因为 redis 不可用期间发生了 DB 变更但是缓存并没有变更，依然为老数据，因此需要将这部分老数据剔除后才能恢复读操作。

step3：校验挤压的 cache_key_queue 记录

轮训查看 cache_key_queue 表中是否有记录存在，若存在记录则认为当前有不一致的缓存数据，需要等待定时任务将暂存的 key 表记录对应的缓存全部删除（同时也会删除 cache_key_queue 表记录）。

step4：恢复缓存读操作

若当前不存在 cache_key_queue 记录则可恢复读操作。

以上阐述了 redis 缓存的自动熔断和恢复方案。需要明确的是，能够进行熔断是有前提条件的，即应用完全去掉缓存，DB 还是可以抗住一段时间压力的，否则一旦出现缓存服务故障，流量全部走到应用，超过了应用和 DB 的承受能力，将服务压垮，后果更加严重。所以不能强依赖熔断机制，不能强依赖缓存，而这就需要接口限流等其他手段来从整体上保证服务的

高可用。此外可进行定期压测，来锚定服务性能上限，进而不断优化对各种策略和资源的配置。

3.5 总结

以上描述了我们强一致性缓存方案的设计思路及一些实现细节。基于该方案，我们核心数据库的 QPS 降低了 80%，缓存的命中率达到 92%。而该方案的关键是通过加锁来控制读写，从表面上看会牺牲一些性能，但是实际上高缓存命中率同样弥补了此缺陷，缓存的建立使得我们服务查询接口 AVG 响应时间降低了 10%左右。

四、结语

以上分别描述了我们的最终一致性和强一致性缓存设计和实现思路。两套缓存方案侧重点各有不同：

- 最终一致性场景的基本思路是：读缓存优先，数据可以容忍暂时不一致，因此重点在及时补偿。
- 强一致性场景的基本思路是：读 DB 优先，缓存仅作为“DB 降压”的辅助手段，在不确定缓存数据是否最新的情况下，宁可多查询几次 DB，也不要查询到缓存中的不一致数据。

此外，我们的最终一致性缓存方案是独立的缓存服务，而该强一致缓存方案，是需要嵌入到应用系统中去使用的。方案的选择需要立足于自身场景，希望我们的分享能够给大家带来一些启发。

日均流量 200 亿，携程高性能全异步网关实践

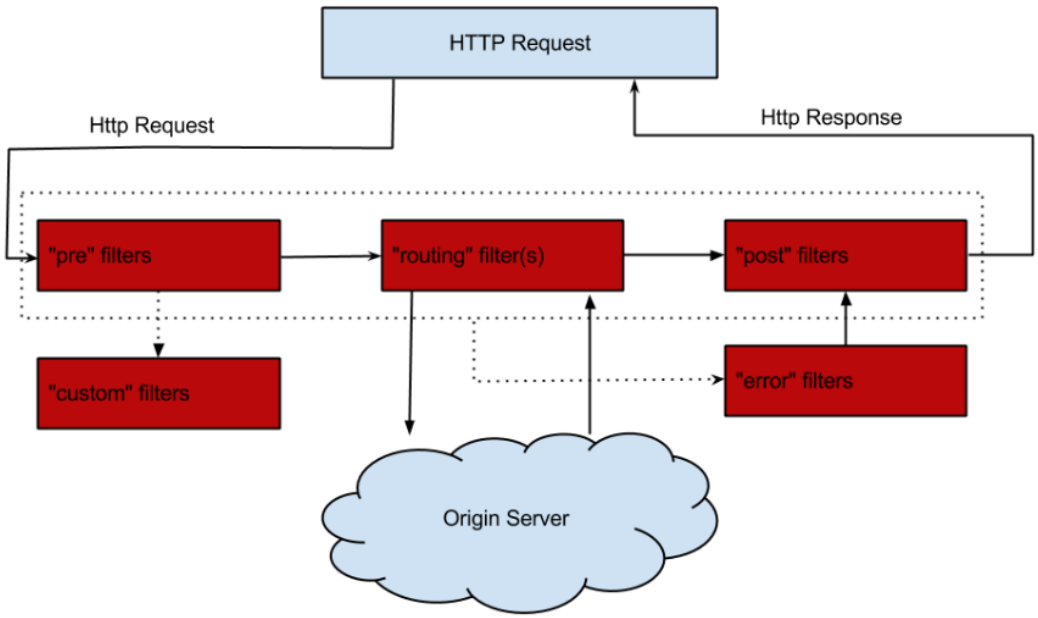
【作者简介】Butters，携程软件技术专家，专注于网络架构、API 网关、负载均衡、Service Mesh 等领域。

一、概述

与许多公司一样，携程 API 网关也是同微服务架构一起引入的基础设施，最早版本发布于 2014 年。随着服务化在公司的快速推进，网关逐渐成为应用暴露到外网的标准方案。后来的“ALL IN 无线”、国际化、异地多活等，网关跟随着公司公共业务与基础架构共同演进。截止 2021 年 7 月，整体接入服务数 3000 以上，日均处理流量 200 亿。

技术方案上，公司微服务早期发展受 NetflixOSS 影响较深，网关方面最早也是参考了 Zuul 1.0 进行的二次开发，核心可概括为四点：

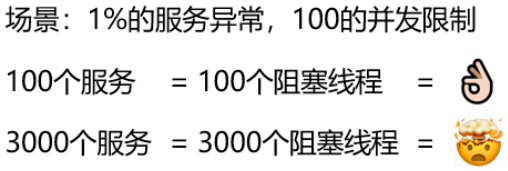
- server 端：Tomcat NIO + AsyncServlet
- 业务流程：独立线程池，分阶段的责任链模式
- client 端：Apache HttpClient，同步调用
- 核心组件：Archaius（动态配置客户端），Hystrix（熔断限流），Groovy（热更新支持）



众所周知，同步调用阻塞线程，系统吞吐受 IO 影响大。作为行业先驱，Zuul 在设计上也考虑到了这点：通过引入 Hystrix，资源隔离配合限流，将故障（慢 IO）框在一定范围内；配合熔断策略，可提前释放部分线程资源；最终达到局部异常不影响全局的目的。

但随着公司业务的发展，上述策略效果逐渐减弱，主要原因在于两方面的变动：

- 业务出海：网关作为海外接入层，部分流量需转回国内，慢 IO 成为常态
- 服务规模增长：局部异常常态化，加上微服务异常扩散的特性，线程池可能长期处于亚健康状态



全异步改造是携程 API 网关近年的一项核心工作点，本文也将由此展开，聊一聊我们在网关方面的工作与实践。重点包括：性能优化、业务形态、技术架构、治理经验等。

二、高性能网关核心设计

2.1. 异步流程设计

全异步 = server 端异步 + 业务流程异步 + client 端异步

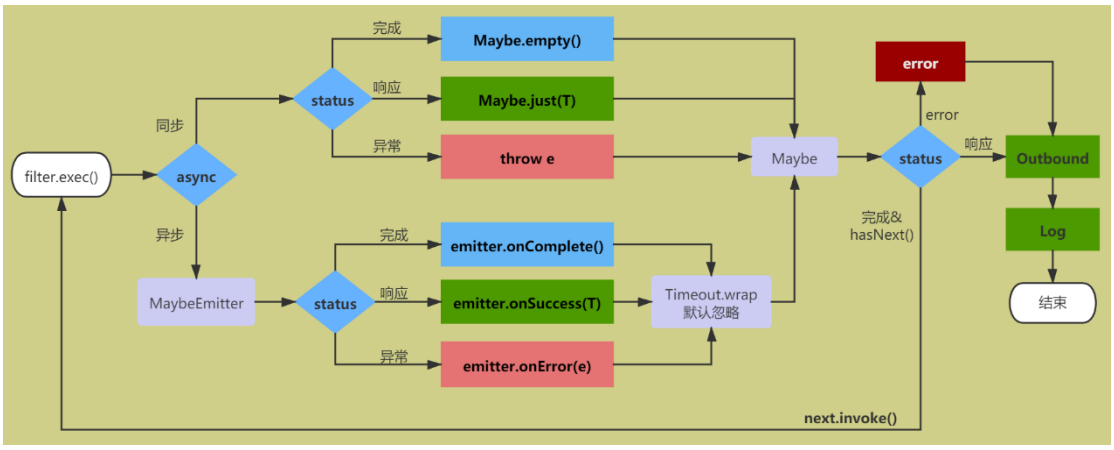
对于 server 与 client 端，我们选择了 Netty 框架，NIO/Epoll + Eventloop 本身就是事件驱动的设计。改造核心在于业务流程的异步化，常见异步场景包括：

- 业务 IO 事件：如请求校验、身份认证，涉及远程调用
- 自身 IO 事件：如读取到了报文的前 xx 字节
- 请求转发：包括 TCP 连接，HTTP 请求

经验上，异步编程相比同步在设计、读写上都会困难一些，一般包括：

- 流程设计&状态转换
- 异常处理，包括常规异常与超时
- 上下文传递，包括业务上下文与 trace log
- 线程调度
- 流量控制

尤其在 Netty 上下文内，对 ByteBuf 生命周期设计的不完善，很容易造成内存泄漏。围绕这些问题，我们设计了对应外围框架，最大努力对业务代码抹平同步/异步差异，方便开发；同时默认兜底与容错，保证程序整体安全。工具上借助了 RxJava，主要流程如下图所示。



- Maybe

- RxJava 内置容器类，标识正常结束、有且仅有一个对象返回、异常三种状态
- 响应式，方便整体状态机设计，自带异常处理、超时、线程调度等封装
- Maybe.empty()/Maybe.just(T)，适用同步场景
- 工具类 RxJavaPlugins，方便切面逻辑封装

- Filter

- 代表一块独立的业务逻辑，同步&异步业务统一接口，返回 Maybe
- 异步场景（如远程调用）统一封装，如涉及线程切换，通过 maybe.obesrveOn(eventloop) 切回
- 异步 filter 默认增加超时，并按弱依赖处理，忽略错误

```
public interface Processor<T> {
    ProcessorType getType();

    int getOrder();

    boolean shouldProcess(RequestContext context);

    //对外统一封装为 Maybe
    Maybe<T> process(RequestContext context) throws Exception;
}


public abstract class AbstractProcessor implements Processor {
    //同步&无响应，继承此方法
    //场景：常规业务处理
    protected void processSync(RequestContext context) throws Exception {}

    //同步&有响应，继承此方法，健康检测
```

```
    //场景：健康检测、未通过校验时的静态响应
    protected T processSyncAndGetReponse(RequestContext context) throws Exception {
        process(context);
        return null;
    };

    //异步，继承此方法
    //场景：认证、鉴权等涉及远程调用的模块
    protected Maybe<T> processAsync(RequestContext context) throws Exception
{
        T response = processSyncAndGetReponse(context);
        if (response == null) {
            return Maybe.empty();
        } else {
            return Maybe.just(response);
        }
    };

    @Override
    public Maybe<T> process(RequestContext context) throws Exception {
        Maybe<T> maybe = processAsync(context);
        if (maybe instanceof ScalarCallable) {
            //标识同步方法，无需额外封装
            return maybe;
        } else {
            //统一加超时，默认忽略错误
            return maybe.timeout(getAsyncTimeout(context), TimeUnit.MILLISECONDS,
                    Schedulers.from(context.getEventloop()), timeoutFallback(context));
        }
    }

    protected long getAsyncTimeout(RequestContext context) {
        return 2000;
    }

    protected Maybe<T> timeoutFallback(RequestContext context) {
        return Maybe.empty();
    }
```

- 整体流程

- 沿用责任链的设计，分为 inbound、outbound、error、log 四阶段
- 各阶段由一或多个 filter 组成
- filter 顺序执行，遇到异常则中断，inbound 期间任意 filter 返回 response 也触发中断

```
public class RxUtil{
        //组合某阶段（如 Inbound）内的多个 filter（即 Callable<Maybe<T>>）
        public static <T> Maybe<T> concat(Iterable<? extends Callable<Maybe<T>>>
iterable) {
            Iterator<? extends Callable<Maybe<T>>> sources = iterable.iterator();
            while (sources.hasNext()) {
                Maybe<T> maybe;
                try {
                    maybe = sources.next().call();
                } catch (Exception e) {
                    return Maybe.error(e);
                }
                if (maybe != null) {
                    if (maybe instanceof ScalarCallable) {
                        //同步方法
                        T response = ((ScalarCallable<T>)maybe).call();
                        if (response != null) {
                            //有 response，中断
                            return maybe;
                        }
                    } else {
                        //异步方法
                        if (sources.hasNext()) {
                            //将 sources 传入回调，后续 filter 重复此逻辑
                            return new ConcattedMaybe(maybe, sources);
                        } else {
                            return maybe;
                        }
                    }
                }
            }
            return Maybe.empty();
        }
    }

public class ProcessEngine{
        //各个阶段，增加默认超时与错误处理
      private void process(RequestContext context) {
            List<Callable<Maybe<Response>>>          inboundTask          =
get(ProcessorType.INBOUND, context);
            List<Callable<Maybe<Void>>>          outboundTask          =
get(ProcessorType.OUTBOUND, context);
            List<Callable<Maybe<Response>>>  errorTask  =  get(ProcessorType.ERROR,
context);
```

```
            List<Callable<Maybe<Void>>> logTask = get(ProcessorType.LOG, context);

            RxUtil.concat(inboundTask)       //inbound 阶段
                    .toSingle()              //获取 response
                    .flatMapMaybe(response -> {
                        context.setOriginResponse(response);
                        return RxUtil.concat(outboundTask);
                    })                       //进入 outbound
                    .onErrorResumeNext(e -> {
                        context.setThrowable(e);
                        return RxUtil.concat(errorTask).flatMap(response -> {
                            context.resetResponse(response);
                            return RxUtil.concat(outboundTask);
                        });
                    })                       //异常则进入 error，并重新进入 outbound
                    .flatMap(response -> RxUtil.concat(logTask))   //日志阶段
                    .timeout(asyncTimeout.get(), TimeUnit.MILLISECONDS,
Schedulers.from(context.getEventloop()),
                            Maybe.error(new ServerException(500, "Async-Timeout-
Processing"))
                    )                        //全局兜底超时
                    .subscribe(              //释放资源
                            unused -> {
                                logger.error("this should not happen, " + context);
                                context.release();
                            },
                            e -> {
                                logger.error("this should not happen, " + context, e);
                                context.release();
                            },
                            () -> context.release()
                    );
        }
    }
```

2.2. 流式转发&单线程

以 HTTP 为例，报文可划分为 initial line/header/body 三个组成部分。

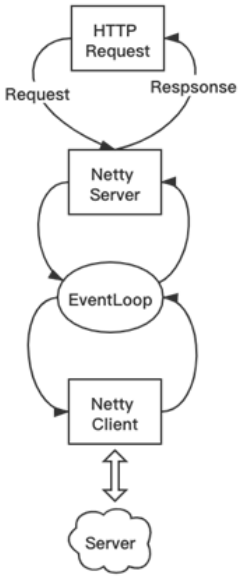
```
POST https://www.ctrip.com/example HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
Content-Type: text/xml; charset=utf-8
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Host: www.ctrip.com
Content-Length: 95

<?xml version="1.0" encoding="utf-8"?>
<string xmlns="http://clearforest.com/">string</string>
```

在携程，网关层业务不涉及 body。因为无需全量存，所以解析完 header 后可直接进入业务流程。于此同时，如果接收到 body 部分：①若已向 upstream 转发请求，则直接转发；②否则需要将其暂存，待业务流程处理完毕，同 initial line/header 一并发送；③对 upstream 端响应的处理方式亦然。

对比完整解析 HTTP 报文的方式，这样处理：

- 更早进入业务流程，意味着 upstream 更早接收到请求，能有效降低网关这层引入的延迟
- body 生命周期被压缩，可降低网关自身的内存开销

虽说提升了性能，但流式的方式也极大提升了整个流程的复杂度。



非流式场景下，Netty Server 端编解码、入向业务逻辑、Netty Cerver 端编解码、出向业务逻辑，各子流程相互独立，各自处理完整的 HTTP 对象。采取流式后，请求则可能同时处于多流程内，引入的困难可归纳为以下三点：

- 线程安全问题：不同流程若采用不同线程，会涉及上下文的并发修改；
- 多阶段联动：比如 Netty Server 请求接收一半遇到了连接中断，此时已经连上了 upstream，那么 upstream 侧的协议栈是走不完的，也必须随之关闭连接；
- 边缘场景处理：比如 upstream 在请求未完整发送情况下返回了 404/413，是选择继续发送、走完协议栈、让连接能够复用，还是选择提前终止流程，节约资源，但同时放弃连接？再比如，upstream 已收到请求但未响应，此时 Netty Server 突然断开，Netty Client 是否也要随之断开？等等。

针对这些场景，我们采用了单线程的方式，核心设计：

- 上线文绑定 Eventloop，Netty Server/业务流程/Netty Client 在同个 eventloop 执行；

- 异步 filter 如因 IO 库的关系，必须使用独立线程池，那在后置处理上必须切回；
- 流程内资源做必要的线程隔离（如连接池）；

单线程方式杜绝了并发问题，在多阶段联动、边缘场景问题处理时，整个系统也处于确定的状态下，有效降低了开发难度与风险；此外减少线程切换，一定程度上也能够提升性能。与之相对的，因为 worker 线程数较少（一般等于 CPU 核数），eventloop 内必须完全杜绝 IO 操作，否则将对系统吞吐造成毁灭性打击。

2.3 其他优化

- 内部变量懒加载

针对请求的 cookie/query 等字段，如无必要，不提前进行字符串解析

- 堆外内存&零拷贝

结合前文流式转发的设计，进一步降低系统内存开销

- ZGC

项目因 TLSv1.3 而引入了 JDK11（JDK8 支持相对较晚，8u261 版本，2020.7.14），自然也对新一代的 GC 算法进行了尝试，实际表现也确实不负盛名。除 CPU 占用有少量提升，整体 GC 耗时下降非常明显。

- 定制的 HTTP 编解码

HTTP 的悠久历史，加之协议自身的开放性，催生了许多“坏实践”，轻则影响成功率，重则威胁网站安全，举两个例子：

- 流量治理

诸如请求体过大（413）、uri 过长（414）、非 ASCII 字符（400）等问题，一般 WebServer 会选择直接拒绝并返回对应状态码。由于直接跳过了业务流程，这类问题在统计、服务定为、排障上都会比较麻烦。扩展编解码，让问题请求也能够走完路由流程，可以帮助解决非标流量的治理问题。

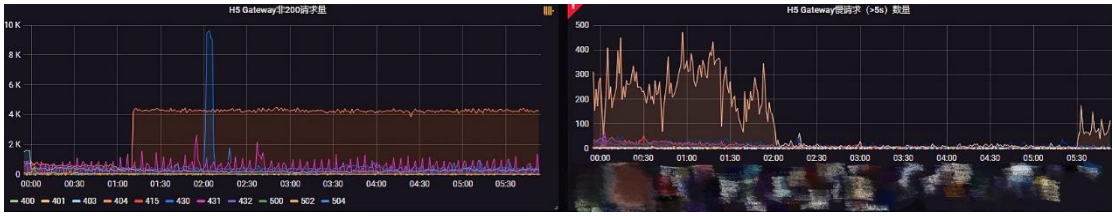
- 请求过滤

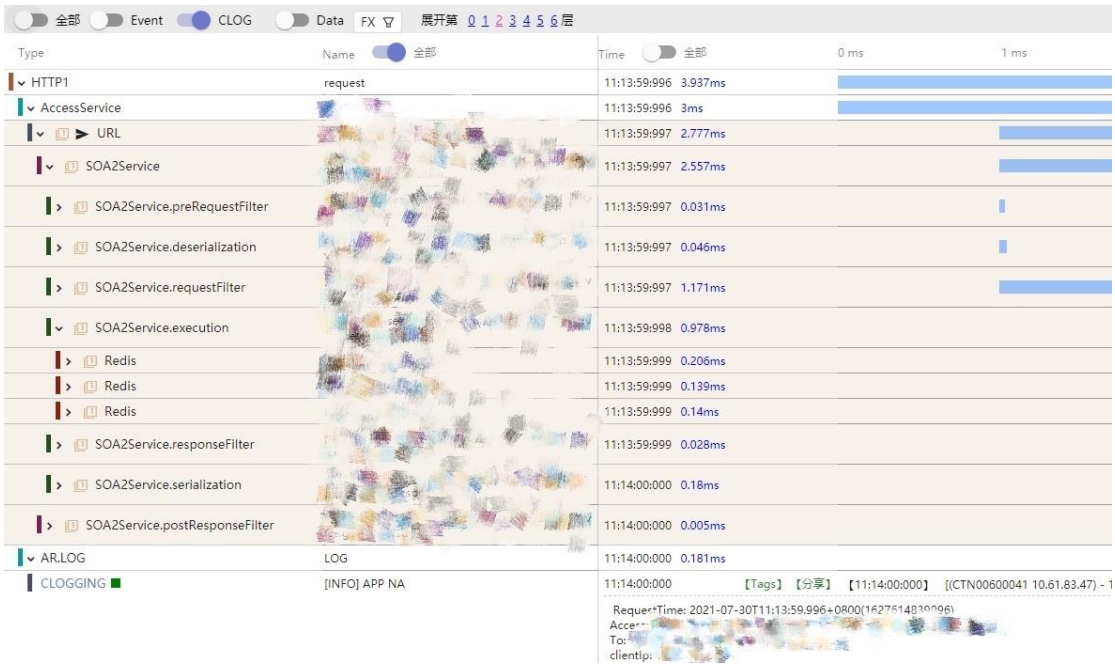
如 request smuggling（Netty 4.1.61.Final 修复，2021.3.30 发布）。扩展编解码，增加自定义的校验逻辑，让安全补丁能够更快落地。

三、网关业务形态

作为独立、统一的入向流量收口点，网关对公司的价值主要体现在三方面：

- 解耦不同网络环境：典型场景包括内网&外网、生产环境&办公区、IDC 内部不同安全域、专线等；
- 天然的公共业务切面：包括安全&认证&反爬、路由&灰度、限流&熔断&降级、监控&告警&排障等；





- 高效、灵活的流量控制

这里展开讲几个细分场景：

- 私有协议

在收口的客户端（APP），由框架层拦截用户发起的 HTTP 请求，通过私有协议（SOTP）的方式发往服务端。

选址方面：①通过服务端下发 IP，杜绝 DNS 劫持；②连接预热；③自定义的选址策略，可依据网络质量、环境等自行切换。

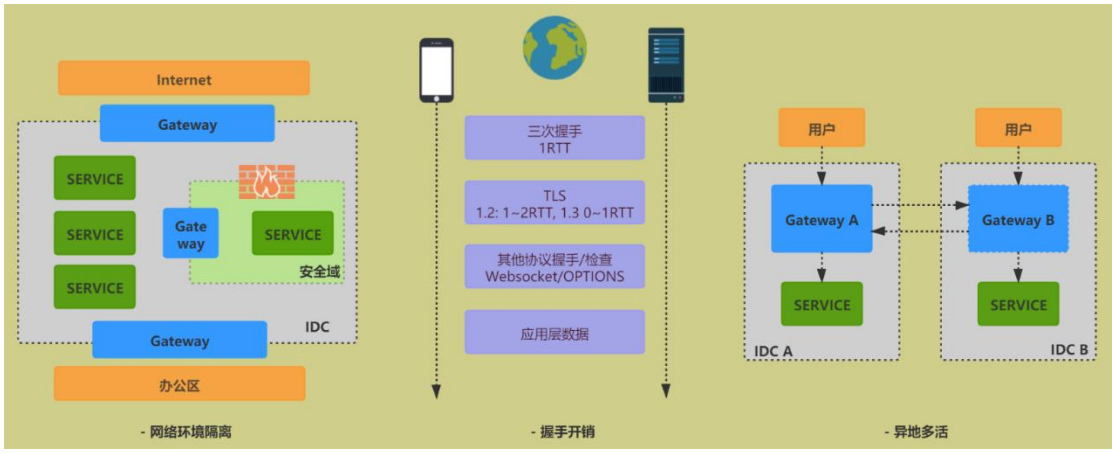
交互方式上：①更加轻量的协议体；②统一加密&压缩&多路复用；③协议在入口处由网关统一转换，对业务透明。

- 链路优化

核心是引入接入层，让远距离用户就近访问，缓解握手开销过大的问题。同时，因为接入层与 IDC 是可控的两端，网络链路选择、协议交互模式上都有更大的优化空间。

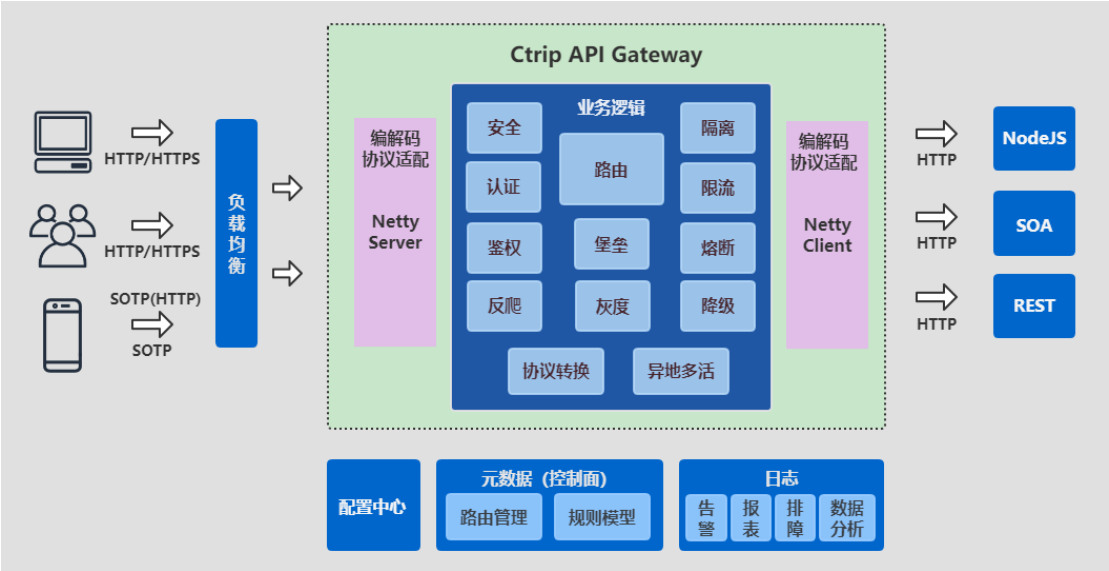
- 异地多活

区别于按比例分配、就近访问策略等，异地多活模式下，网关（接入层）需按照业务维度的 shardingKey 进行分流（如 userId），防止底层数据冲突。



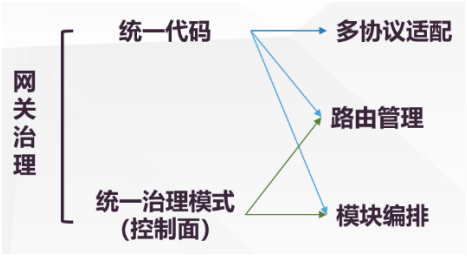
四、网关治理

下图总结了线上网关的工作状态。横向对应我们的业务流程：不同渠道（APP、H5、小程序、供应商）、不同协议（HTTP、SOTP）的流量经由负载均衡打到网关，经过系列业务逻辑的处理，最终转发至后端服务。经历了第二章的改造后，横向业务在性能、稳定性上都得到了较好的提升。



另一方面，由于多渠道/协议的存在，线上网关按业务划分，进行了独立集群的部署。业务差异（路由数据、功能模块）早期通过独立代码分支管理，随着分支数的增加，整体的运维复杂度越来越高。系统设计中，复杂度往往也意味着风险。如何对多协议、多角色的网关实施统一治理，如何以较低的成本，快速为新业务搭建定制化网关，成为了我们后一阶段的工作重心。

解决方案也比较直观地在图中画了出来，一是协议上兼容处理，让线上代码跑在一套框架下；二是引入控制面，对线上网关的差异特性进行统一管理。



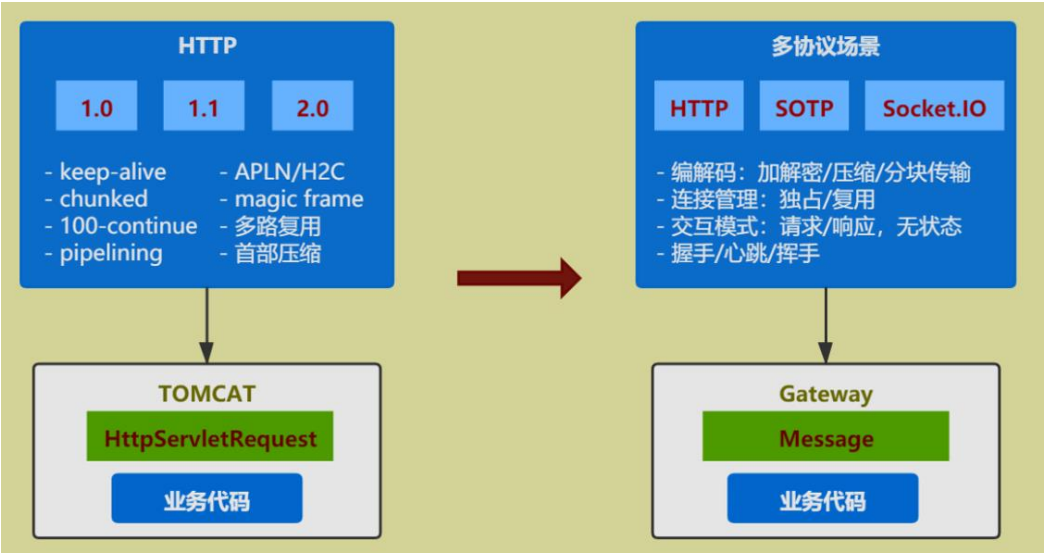
4.1 多协议兼容

协议兼容的做法并不新鲜，整体可以参考 Tomcat 对 HTTP/1.0、HTTP/1.1、HTTP/2.0 的抽象。HTTP 自身虽然在各个版本内新增了大量 feature，但我们在做业务开发时通常感知不到这些，核心在于 HttpServletRequest 接口的抽象。

在携程，网关面对的都是请求—响应模式的无状态协议，报文组成上也可以划分为元数据、扩展头、业务报文三部分，因此可以比较方便地进行类似的尝试。对应工作可以用以下两点概括：

- 协议适配层：用于屏蔽不同协议的编解码、交互模式、对 TCP 连接的处理等
- 定义通用中间模型与接口：业务面向中间模型与接口编程，更好地聚焦到协议对应的业

务属性上去



4.2 路由模块

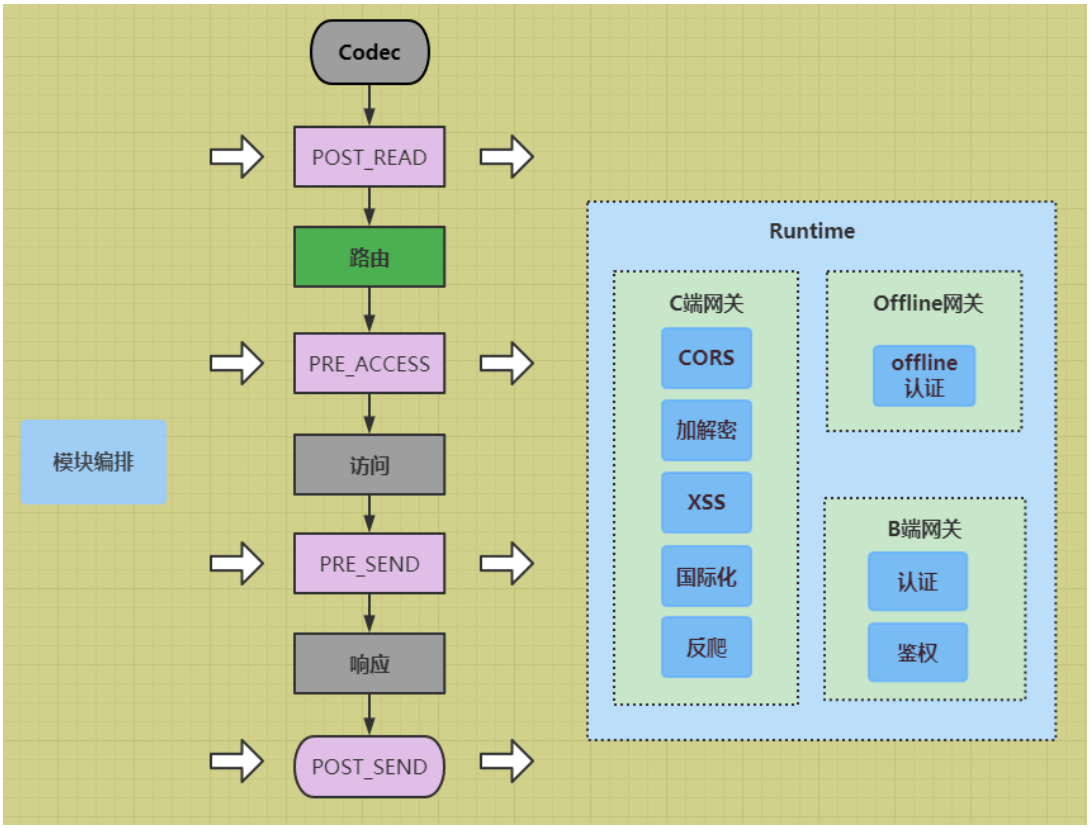
路由模块是控制面的两个主要组成部分之一，除了管理网关—服务间的映射关系，服务本身可以用以下模型概括：

```
{
        //匹配方式
        "type": "uri",

        //HTTP 默认采用 uri 前缀匹配，内部通过树结构寻址；私有协议（SOTP）通过服务唯一标识定位。
        "value": "/hotel/order",
        "matcherType": "prefix",

        //标签与属性
        //用于 portal 端权限管理、切面逻辑运行（如按核心/非核心）等
        "tags": [
            "owner_admin",
            "org_framework",
            "appId_123456"
        ],
        "properties": {
            "core": "true"
        },

    //endpoint 信息
        "routes": [{
```

```
            //condition 用于二级路由，如按 app 版本划分、按 query 重分配等
            "condition": "true",
            "conditionParam": {},
            "zone": "PRO",

            //具体服务地址，权重用于灰度场景
            "targets": [{
                "url": "http://test.ctrip.com/hotel",
                "weight": 100
            }
        ]
    }]
}
```

4.3 模块编排

模块编排是控制面的另一项核心部分。我们在网关处理流程内预留了多个阶段（图中用粉色标记）。除开熔断、限流、日志等通用功能，运行时不同网关所需执行的业务功能由控制面统一下发。功能本身在网关内部有独立的代码模块，控制面额外定义了功能对应的执行条件、参数、灰度比例、错误处理方式等。这种编排方式也在侧面保证了模块间的解耦。




```
{
    //模块名称，对应网关内部某个具体模块
```

```
    "name": "addResponseHeader",

    //执行阶段
    "stage": "PRE_RESPONSE",

    //执行顺序
    "ruleOrder": 0,

    //灰度比例
    "grayRatio": 100,

    //执行条件
    "condition": "true",
    "conditionParam": {},

    //执行参数
    //大量${}形式的内置模板，用于获取运行时数据
    "actionParam": {
      "connection": "keep-alive",
      "x-service-call": "${request.func.remoteCost}",
      "Access-Control-Expose-Headers": "x-service-call",
      "x-gate-root-id": "${func.catRootMessageId}"
    },

    //异常处理方式，可以抛出或忽略
    "exceptionHandle": "return"
}
```

五、总结

网关长期以来都是各类技术交流平台上的热点，方案也非常丰富：发展早、易上手的 Zuul1.0、高性能的 Nginx、集成度高的 SpringCloud Gateway、如日中天的 Istio 等等。最终决定选型的还是各公司自身的业务背景与技术生态。也正因此，在携程我们选择了自研的道路。

技术不断发展，我们也在持续探索，公共网关同业务网关的关系、新协议的落地（HTTP3）、与 ServiceMesh 的关系等等，真诚欢迎有兴趣的同学一起参与讨论。

多业务线亿级体量，携程是怎么做账务中台的

【作者简介】本文为联合撰文，作者团队负责携程集团支付账务系统、消费金融账务系统、清结算和对账等工作的的开发、设计和运维工作。

一、前言

原先携程内部的各账务系统都是随着自身的业务发展而建立起来的，其中有些共同的东西，但也存在着不少差异。但其对底层业务的抽象是统一的，都可以抽象为：账户开立、记账、稽核。为了系统开发、运维的简便性，也为了更好的为前台业务提供支持，我们计划实施账务中台系统，从而做到账务系统的：

1）敏捷：快速的适应业务需求的变化，满足外部快速变化的需求，实现业务的敏捷。
2）解耦：建立功能独立的系统，避免因为一个功能的修改而影响很多方面，降低功能的耦合度。
3）复用：对一些公共组件的复用，提高开发效率，避免重复建设，使得数据和流程都可以得到管控。

二、账务中台之路-基础篇

2.1 系统概述

2.1.1 背景

账务系统从 2014 年开始一期建设，先后经历了两次大的技术架构升级。

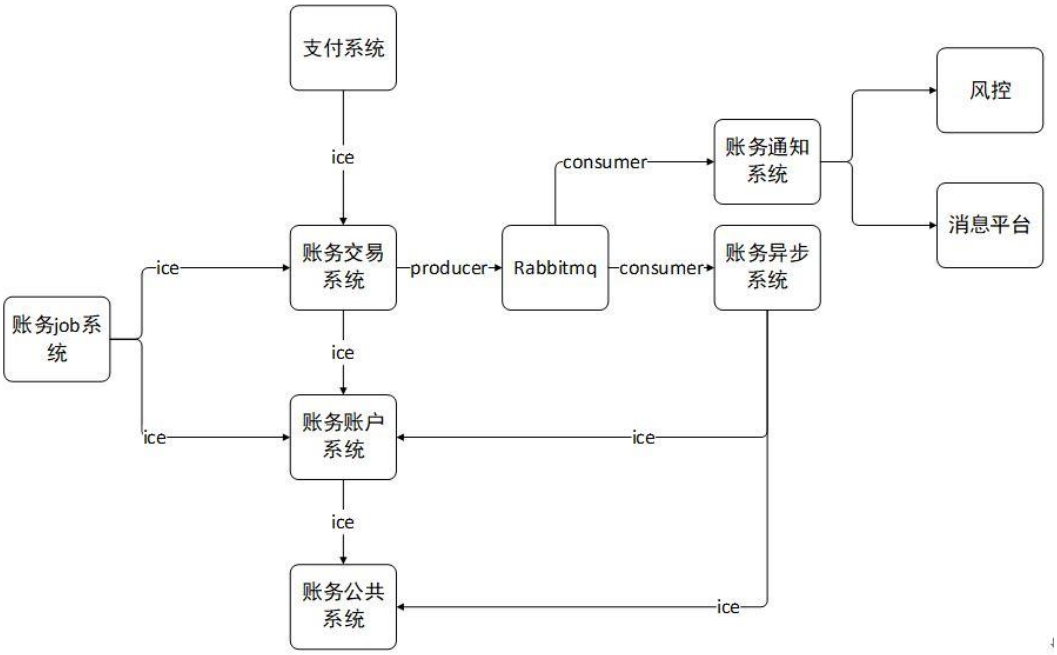
一期：

账务组刚成立的时候，携程的 JAVA 技术栈尚未完善。

技术上我们是首批试点使用 JAVA 的小组，使用的分布式 RPC 框架是 Zeroc ICE，它可以支持多语言，通过 slice 文件生成代码，性能也比较好，所以当初选型用了这个。消息队列用的 RABBITMQ，缓存用的 REDIS。这些集群都是自运维的。

业务上，一期的业务只支持单用户单账户的模式，交易支持充值、支付、退款、预授权类（预授权冻结，预授权撤销，预授权完成，预授权完成撤销）、提现、转账，接口都是基于业务接口独自开发的。

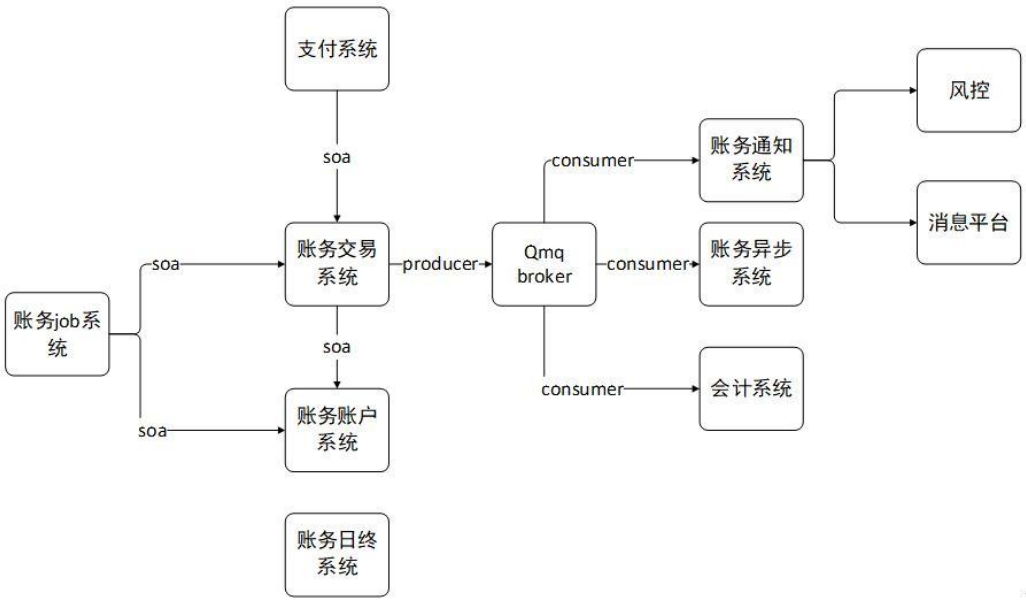
系统业务架构图如下：



二期:

随着携程 JAVA 技术栈的完善，二期主要针对 JAVA 技术栈进行了升级，放弃了自运维的集群，使用了携程 JAVA 体系，包括 SOA，qconfig，qmq，qshecdule 等技术。业务上增加了会计系统和日终的实现。

系统业务架构图如下:



原有的账务核心系统存在以下问题:

1）抽象不足：对业务规则抽象不够，如果新增业务需要编码实现
2）隔离不够：系统不同业务的系统流量、数据没有隔离，存在某一业务有问题，影响全局的风险
3）降级策略：不支持
4）扩容困难

基于以上问题，我们设计并实现了新的统一账务平台。

2.1.2 目标

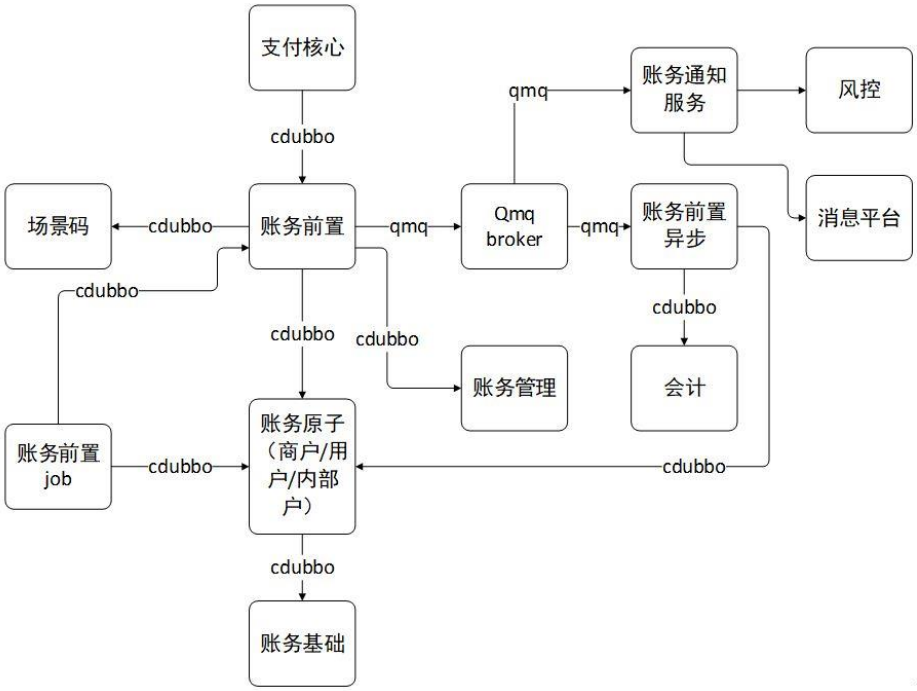
针对旧系统的不足，我们确定统一账务平台的目标:

1）抽象
2）隔离
3）易扩容
4）配置化
5）支持多机构多币种

2.2 系统架构与简介

统一账务系统旨在建立一套立足于携程集团之下的高可用, 易扩展, 业务可定制的账务系统。系统包括场景码系统, 账务前置系统, 账务核心系统, 账户管理系统, 会计系统, 异步系统, job 系统，日志系统。各个系统之间通过 dubbo 进行服务拆分解耦。

系统业务架构图如下:



- 前置系统：账务的业务处理系统，主要负责对上游业务系统的对接，完整账户的拆分等工作。
- 账务核心系统（原子系统）：主要负责账户记账，记录对商户、用户、内部户等客户账的动账及明细。
- 管理系统：对外提供商户、用户、内部户的管理服务，包括创建、查询、状态冻结、状态解冻等服务。
- 会计系统：采用复式记账法根据分录规则对发生的交易进行记录，来表示资金的流转。
- 基础服务系统：对外提供科目、分录、交易码等基础配置的查询服务。
- 日终系统：对记账原子和会计系统数据进行稽核，完成数据校验工作。

2.3 系统设计

2.3.1 基础组件设计

2.3.1.1 日志组件

我们日志进行 logger 输出，会碰到以下的痛点：

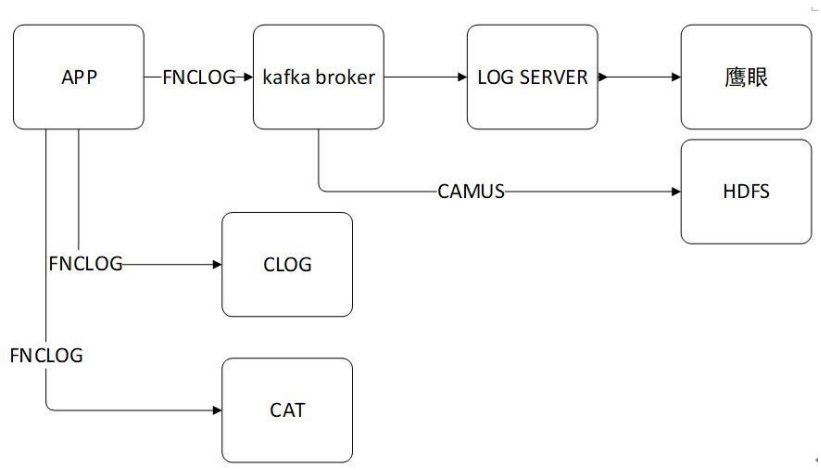
1）我们经常会对方法的入参，出参及异常进行日志打印，还要把 tag 写入 clog 和 ES，手写的话工作量巨大。
2）有些日志需要脱敏处理，比如手机号、身份证号、卡号，不能把明文输出到日志。
3）logger 目前只支持抛公司的 logger 日志平台，部门想自定义日志查询分析工具比较困难。

所以在设计统一账务中台化的工程中，进行了日志组件的设计：
1）统一使用高性能的 log4j2 替代 logback；
2）通过 spring aop 和 annotation，支持方法入参、出参、异常日志的自动打印；
3）支持 clog 和 es 的 tag 的配置，可以从参数中获取，并通过 log4j2 的 ThreadContext 打入本地线程，线程使用过程中 tag 共享，代码如下所示：

```
@Override
@FncLog(onMdcs = {
        @FncLog.Mdcs(mdcKey = TAG_KEY_MAIN_TYPE, mdcValue = "FI.frontAdd"),
        @FncLog.Mdcs(mdcKey = TAG_KEY_DATA_TYPE, mdcValue = "frontAdd"),
        @FncLog.Mdcs(mdcKey = TAG_KEY_TRAN_NO, mdcTemplate = "${args[0].requestNo}")
})
public FrontAddRsp frontAdd(FrontAddReq request) {
```

4）支持配置脱敏规则，进行敏感信息的脱敏处理；
5）同步抛公司的 clog 和 cat，异步抛 kafka，异步接受程序进行 ETL 处理，抛部门自己的日志系统（比如鹰眼系统，hive 日志分析系统）；
6）抛 kafka 时，对于原始报文进行 apache arvo 压缩处理，减少传输带宽；
7）支持原生 API，可以手工打 tag 和脱敏处理。

流程图如下：



2.3.1.2 分库分表组件

分库组件，我们调研过公司现有的和开源的组件，最终选用了开源的 sharding-jdbc。

1）携程的 dal 组件

dal 使用 dal cluster 通过服务端配置分库分表信息，但是全套使用 dal 太过笨重，它是一个完全的 ORM 框架，生成 sql 的工具不支持生成特殊自定义的 sql。

2）去哪儿的 qdb 组件
通过配置表达式或算法的方式进行分库分表配置，缺点是文档较少，容易踩坑。

3）MYCAT

Mycat 是一个中间件，它拦截了用户发送过来的 SQL 语句，首先对 SQL 语句做了一些特定的分析：如分片分析、路由分析、读写分离分析、缓存分析等，然后将此 SQL 发往后端的真实数据库，并将返回的结果做适当的处理，最终再返回给用户。Mycat 的缺点就是需要搭建一套中间件做拦截者，而且需要自运维，成本比较高。

4）sharding-jdbc

当当网首先推出的开源分库组件，现在变成 apache 项目，分为 sharding-jdbc、sharding-proxy。开源社区活跃。我们使用轻量级的 sharding-jdbc，可以编写算法，支持精确分片、范围分片、复合分片和自定义 hint 分片，配置方式支持 xml、yml 和 java api 方式。基本能解决我们所有的分库分表需求。我们把分库算法包成 jar 包，方便使用。配置我们使用 yml。在使用过程中，需要结合 dal cluster 的 key，代码示例如下：

```
@Primary
@Bean(name = "frontDs")
public DataSource frontDs(DalDataSourceFactory dalDataSourceFactory, DsQconfig dsQconfig) {
    DataSource dataSource = null;
    try {
        File ymlFile = ResourceUtils.getFile( resourceLocation: "classpath:dbshard/front-dbshard.yml");
        Map<String, DataSource> frontDs = Maps.newHashMap();
        Map<String, String> frontDsMap = dsQconfig.getSpecificDsMap( keyPrefix: "frontds");
        for (Map.Entry<String, String> entry : frontDsMap.entrySet()) {
            frontDs.put(entry.getKey(), dalDataSourceFactory.createDataSource(entry.getValue()));
        }
        dataSource = YamlShardingDataSourceFactory.createDataSource(frontDs, ymlFile);

    } catch (Exception e) {
        LOGGER.error(e.getMessage(),e);
    }
    return dataSource;
}
```

2.3.2 前置系统设计

账务前置位于整个账务体系的最上游，提供标准的交易接口，包括入款、入款返还、出款、出款返还、预授权类、转账以及批量接口。

2.3.2.1 标准的交易接口

整合之前的老系统，都是业务导向的接口，随着业务的不断迭代，接口越来越多，职责不清晰，代码重复，给维护带来很大的工作量。比如：光提现接口，就分为个人提现、返现提现、商户提现和定向提现。另外，原先的子账户的交易顺序是硬编码的，如果发生子账户的增加或交易顺序的变化，带来的复杂度就成倍增加。

我们经过研究，发现账务处理是有共性的，对于交易顺序、原子交易类型都是可以提取出属性的。所以我们建立了场景码模型。
首先，我们定义子账户 id，按账户类型+币种+业务类型唯一定义一个子账户。

其次，按产品代码+交易类型来定义一个交易顺序，交易顺序关联子账户 id，该顺序设置为默认的场景码。接口只要传入产品代码和交易类型就能能走默认的场景码。

第三，支持商户自定义场景码，我们维护了一个后台管理系统，允许商户自定义场景码，审核通过后，接口传入该场景码编号就可以走自己定义的场景码。

2.3.2.2 异步化

我们接口都是同步接口，为了减少同步响应时间，把次要的工作通过 mq 进行异步化处理。比如：转账的转入方入款，抛送会计等。

2.3.2.3 数据库策略

除了支持自己的支付业务，也支持把账务系统输出到其他 BU。为了数据权限及互不影响，我们做了数据隔离。需要特别说明的是，只做数据隔离，系统还用同一套，不做隔离，方便发布和运维。数据隔离分两层，第一层是 domain，区分自有/BU。第二层是具体分库。

sharding 库也分两套，Mapping 库和交易库。Mapping 库存放请求流水号和前置流水号的关系。交易库存放所有的交易信息。特别地，我们把逆向交易和原交易落在同一 DB 中，这样有利于控制逆交易和原交易在一个事物内。

首先，我们使用请求流水号做 hash 算法，分散到 mapping db。Mapping db 只保存请求流水号和前置流水号的关系，Mapping db 也是分库的，分库数量初始是固定的，以后扩展可以用一致性 hash 算法进行扩容。

我们的交易表是通过权重配置来分库，通过权重可以进行数据的自由分配。DB 支持友好的扩容，下线和故障切换。我们有一套故障切换的方案，如果某个分片出现 dbconnect 异常，我们会抛送支付的监控系统。监控系统会有套智能算法会实时监控，达到阀值后自动触发该分片的 markdown/markup 机制。

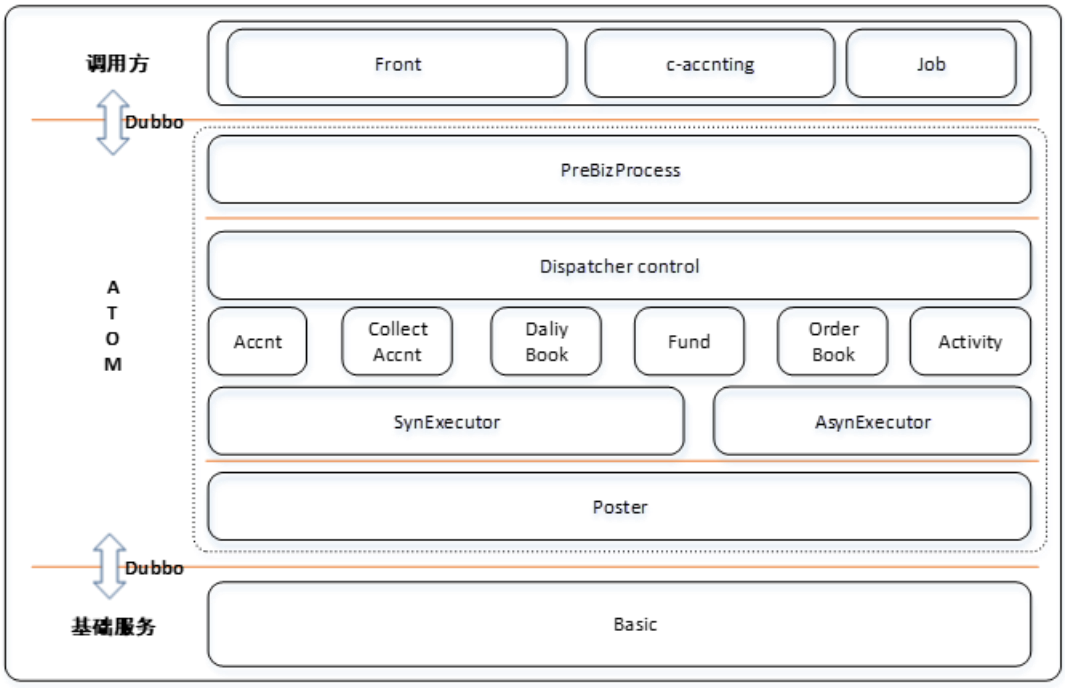
2.3.2.4 异常处理

高并发场景下的异常处理一直都是各方研究的课题。为了适应高并发场景，前置做了如下的异常处理：

1）幂等机制：所有接口都支持幂等操作。
2）重试机制：接口内部调用的重试，job 补偿重试，上游也可以发起相同请求报文的失败重试。
3）查询机制：所有接口都写了一套查询接口，上游可通过查询接口查该交易的最终状态。
4）通知机制：支持成功/失败的结果主动通知上游的机制。

2.3.3 原子系统设计
原子系统流程处理中，主要有以下几步：订单参数预处理，分单，同步执行器，异步执行器，后处理，最后封装参数返回。为方便业务扩展，系统维护，在分单和执行部分，系统架构采用责任链模式的分单器；代理进行分单，产生 drivers，再由系统自动注册的同步执行器和异步执行器进行执行。目前，只有订单登记簿采用异步方式完成，后续 job 系统中会对任务进行相应的补偿。

下图为原子系统的系统架构图：

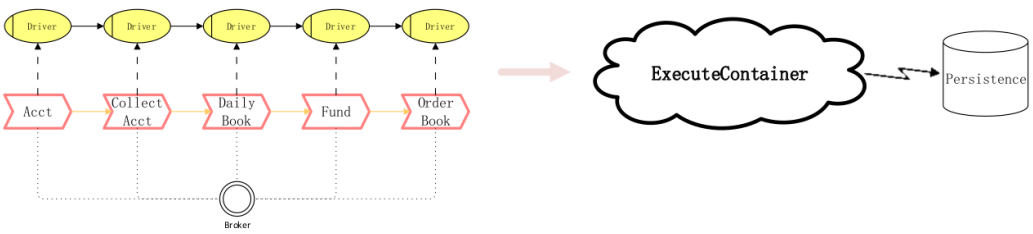


PreBizProcess 是参数预处理部分根据调用域不同进行了定制化校验与信息构造。

DispatcerControl 是分单部分，因为原子系统负责账户余额的管理，不存在任何业务逻辑，所以可以将记账模型进行抽象以适应不同的业务需求。记账模型包括账户入账与出账（Accnt）、活动入账与出账（Activity）、资金冻结与解冻（Fund）、日登记簿入账与出账（DaliyBook）、订单登记入账与出账（OrderBook）等记账模型。针对有效期概念的账户，增加了登记簿管理，日登记簿对账户同一有效期的资金进行汇总，订单登记簿是有效期的资金的订单维度的记录。

SynExecutor 是同步执行器主要负责出账入账，资金冻结解冻，日登记簿处理；AsynExecutor 是异步执行器负责订单登记簿的记账操作。日登记簿采用同步执行的方式，而日登记簿记账成功可保证订单登记簿记账成功，故订单登记簿采用异步记账既可以保证记账成功又能减少系统同步处理的时间。

后处理部分会发送动账消息，给关心账户余额变动的系统，比如风控。



2.3.4 会计系统设计

会计系统采用复式记账方式完成，可以清晰的表述资金是从哪里来并流向了哪里，针对不同的业务涉及不同的清分规则。在清分规则中可以配置记账的不同策略，比如单条、汇总记账等不同策略。

针对同一业务多科目的场景，添加扩展配置，实现清分规则的科目动态化。

2.3.5 日终系统设计

2.3.5.1 为什么需要日终系统

1）提供账务系统支撑

要保证账务系统能正常运转，账务的余额要 100%准确。影响账务系统稳定性的主要因素有以下几个方面：

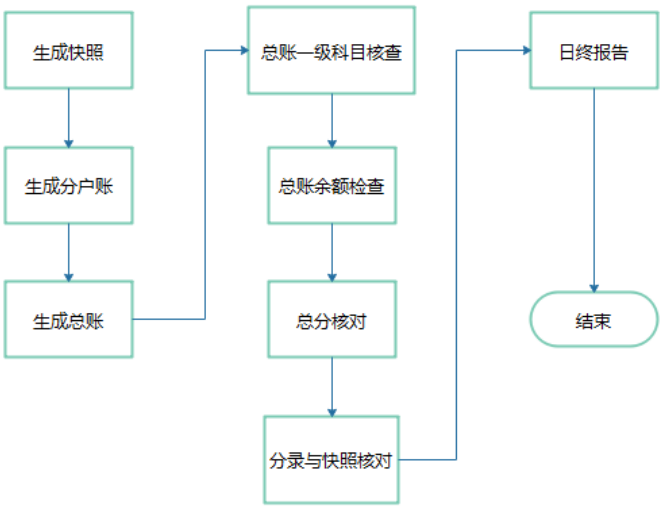
- 缓冲记账，问题表现：分录有数据，明细无数据
- 会计异步记账，问题表现：明细有数据，分录无数据
- 分录规则配置，问题表现：同一笔分录中，借贷方金额不一致

2）为企业大财务提供汇总记账凭证

账务系统记录的是业务账，这些数据是整个企业财务数据的一部分，需要合并到公司的大财务系统中去。

日终系统对会计分录进行加工映射为大财务的分录，然后汇总，直接对接企业 ERP 总账。

2.3.5.2 日终系统都做了什么



1）生成快照

每日凌晨统计截至上一日的所有账户的快照。

2）生成分户账

根据快照生成分户账。

3）生成总账

根据分录流水生成科目总账，科目发生额和余额从末级科目逐级汇总到一级科目。

4）总账平衡检查

余额平衡检查：一级借方科目余额=一级贷方科目余额；
发生额平衡检查：一级科目借方发生额=一级科目贷方发生额

5）总分核对

总账科目余额等于分户账科目余额。业务 24 小时不间断运行，账户中余额在不断变化，无法准确取到期末的账户余额进行核对，采用余额快照与总账科目余额进行核对。

6）稽核明细

检查明细账与分录流水是否一致。对于当日发生过余额变动的账户，昨日余额与分录流水中的发生额进行轧差，检查计算出的余额与快照余额是否一致。

2.3.5.3 日终系统遇到的挑战

1）24 小时记账

在银行账务系统中，对于 24 小时运行，有很多种方案，例如切换余额、记不同分户账、日切后补流水等，但无论哪种方案，都不能实现完全 24 小时运行。

其问题，主要是因为日终要进行总分核对，而分户账余额是在不断变化的，所以要想办法把期末的分户账余额取出来进行核对。

本日终系统解决方案，采用余额快照与总账进行核对，这样即使分户账余额进行变化，也不影响总分核对。

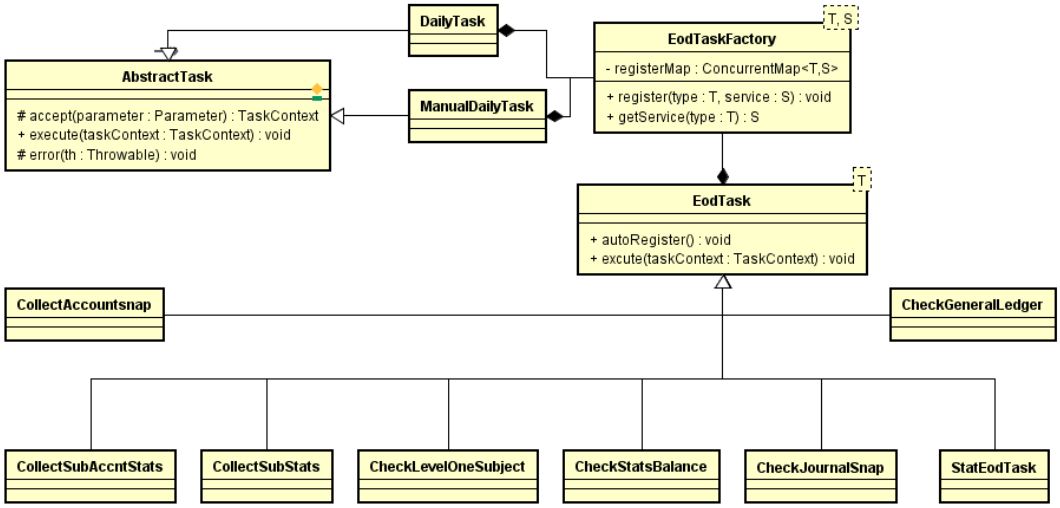
2）生成账户快照

生成快照的方式有两种：

- 从账户余额中获取
- 交易明细按账户汇总发生额更新快照

相较于数亿账户而言，每日发生交易的则要少得多。采用动账汇总的方式，对于数据库的操作更少，处理时间更快。

3）流程复杂、测试困难



对日终任务模型进行抽象，按照业务边界划分为：快照生成、分户账生成、总账生成等多个子任务，自动注册到任务工厂中，以便编排调用。

2.4 总结

此系统业务接入规则、会计清分规则都是基于配置的，在业务发展的新增账户类型、业务、币种、机构等日常变化都可以基于配置进行。
—

三、后记

账务中台建设到现在，已经完成了携程体系内账务中台的基本建设，这只是中台建设的第一步，后续规划还包括分布式事务、热点账户的处理；新机构业务接入如何更简洁等等。

数据为王，携程国际火车票的 Sharding-Sphere 之路

【作者简介】瑞华，携程高级后端开发工程师，关注系统架构、分库分表、微服务、高可用等。

一、前言

随着国际火车票业务的高速发展，订单量快速增长，单数据库瓶颈层面的问题逐渐显露，常规的数据库优化已无法达到期望的效果。同时，原先的底层数据库设计，也存在一些历史遗留问题，比如存在部分无用字段、表通过自增主键关联和各个应用直连数据库等问题。

为此，经过讨论后，我们决定对订单库进行分库分表，同时对订单表进行重构，进而从根本上解决这些问题。

二、问题挑战

目标确定后，实践起来可不轻松，出现了很多的问题和挑战。这里列举一些典型问题，大致可以分为两大类：分库分表通用问题、具体业务关联问题。

分库分表通用问题

- 如何切分，垂直分还是水平分？分片的键，如何选取？
- 如何根据键值路由到对应库、对应表？
- 采用什么中间件，代理方式还是中间件的方式？
- 跨库操作等问题，如跨库事务和跨库关联？
- 数据扩容问题，后续如何进行扩容？

具体业务关联问题

- 各个应用直连数据如何解决？
- 如何进行平滑过渡？
- 历史数据如何恰当迁移？

三、方案选型

3.1 如何切分

切分方式，一般分为垂直分库、垂直分表、水平分库和水平分表四种，如何选择，一般是根据自己的业务需求决定。

我们的目标是要从根本上解决数据量大、单机性能问题等问题，垂直方式并不能满足需求，所以我们选取了水平分库+水平分表的切分方式。

3.2 分片键选取

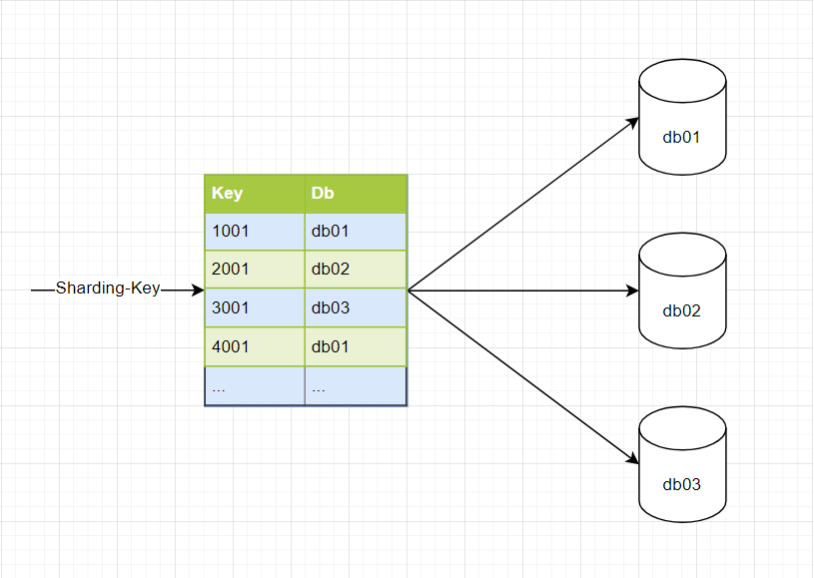
一般是根据自己的实际业务，来选择字段来作为分片的键，同时可以结合考虑数据的热点问题 、分布问题。比如订单系统，不能根据国家字段进行分片，否则可能会出现某些国家很多的订单记录，某些国家几乎没有订单记录，进而数据分布不均。相对正确的方式，比如订单类系统，可以选择订单 ID；会员系统，可以选择会员 ID。

3.3 如何路由

选定了分片的键之后，接下来需要探讨的问题，就是如何路由到具体的数据库和具体的表。以分片键路由到具体某一个数据库为例，常见的路由方式如下：

映射路由

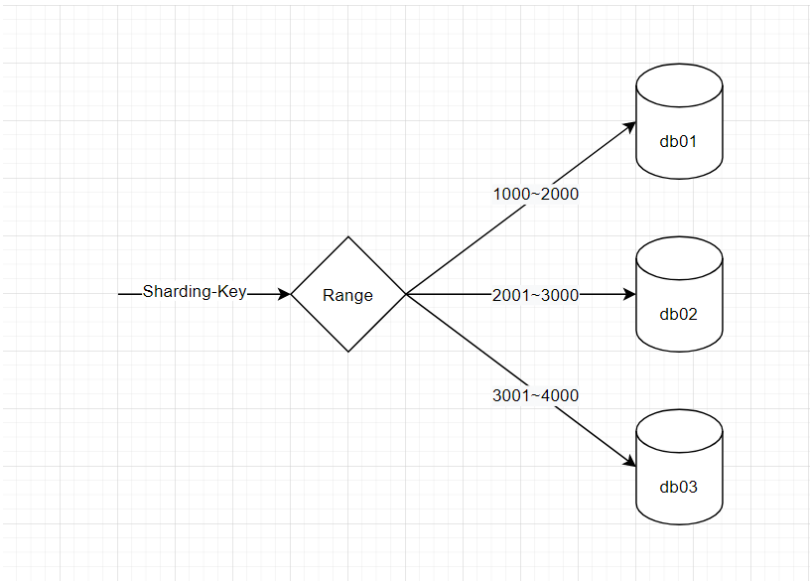
映射路由，即新增一个库，新建一个路由映射表，存储分片键值和对应的库之间的映射关系。比如，键值为 1001，映射到 db01 这个数据库，如下图所示：



映射方式，优点是映射方式可任意调整，扩容简单，但是存在一个比较严重的不足，就是映射库中的映射表的数据量异常巨大。我们本来的目标是要实现分库分表的功能，可是现在，映射库映射表相当于回到了分库分表之前的状态。所以，我们在实践中，没有采取这种方式。

分组路由

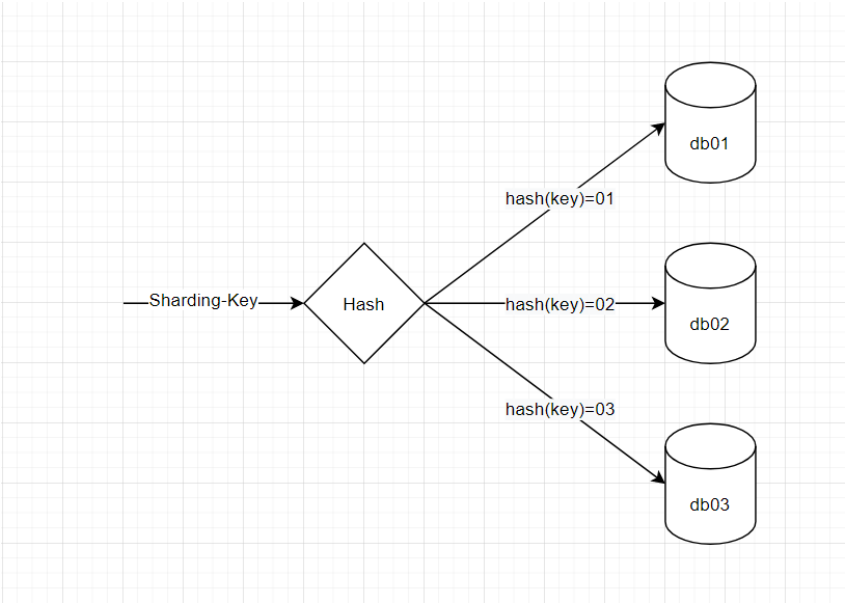
分组路由，即对分片的键值，进行分组，每组对应到一个具体的数据库。比如，键值为 1000 到 2000，则存储到 db01 这个数据库，如下图所示：



分组方式，优点是扩容简单，实现简单，但是也存在一个比较严重的不足，是数据分布热点问题，比如在某一个时间内，分片键值为 2001，则在将来一段时间内，所有的数据流量，全部打到某一个库（db02）。这个问题，在互联网环境下，也比较严重，比如在一些促销活动中，订单量会有一个明显的飙升，这时候各个数据库不能达到分摊流量的效果，只有一个库在接收流量，会回到分库分表之前的状态。所以，我们也没有采取这种方式。

哈希路由

哈希路由，即对分片的键值，进行哈希，然后根据哈希结果，对应到一个具体的数据库。比如，键值为 1000，对其取哈希的结果为 01，则存储到 db01 这个数据库，如下图所示：

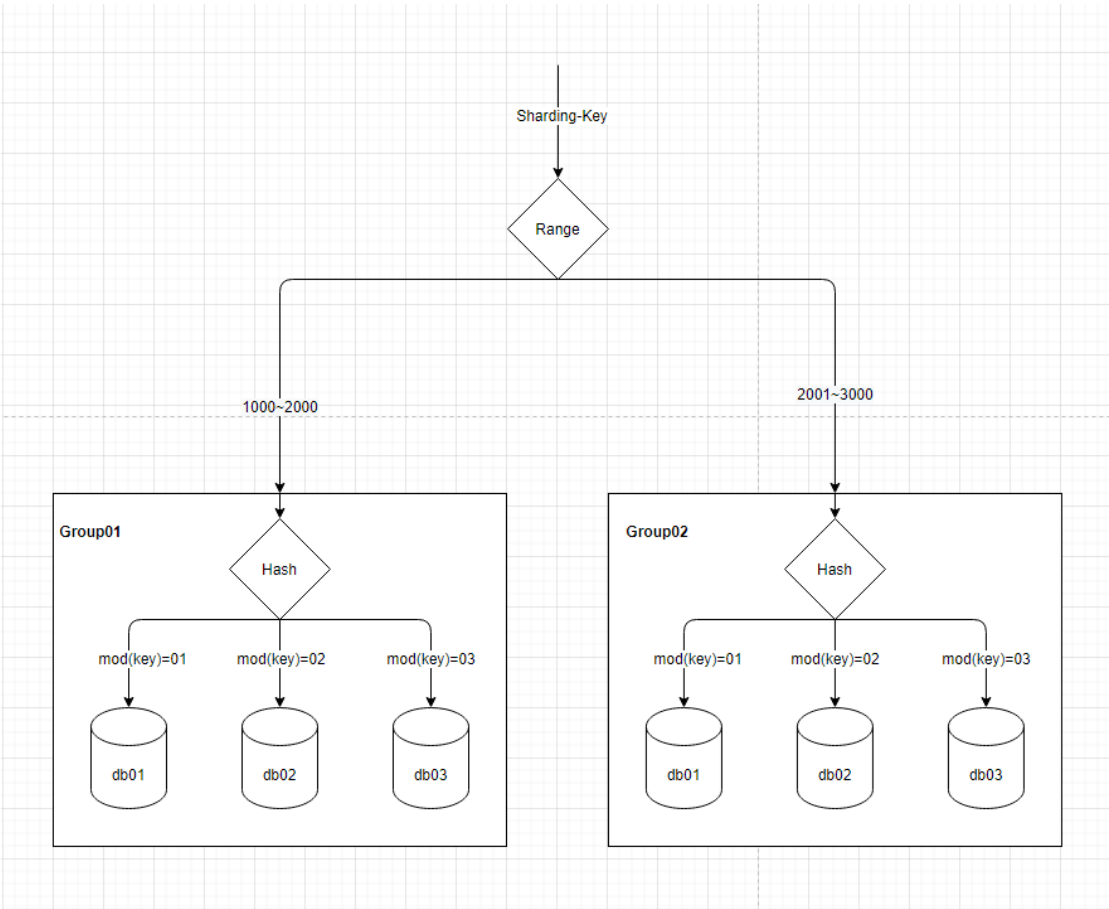


哈希方式，优点是分布均匀，无热点问题，但是反过来，数据扩容比较麻烦。因为在扩容过程中，需要调整哈希函数，随之带出一个数据迁移问题。互联网环境下，迁移过程中往往不

能进行停服，所以就需要类似多库双写等方式进行过渡，比较麻烦。所以，在实践中也没有采取这种方式。

分组哈希路由

分组哈希路由，即对分片的键值，先进行分组，后再进行哈希。如下图所示：



在实践中，我们结合了前面的几种方式，借鉴了他们的优点不足，而采用了此种方式。因为分组方式，能很方便的进行扩容，解决了数据扩容问题；哈希方式，能解决分布相对均匀，无单点数据库热点问题。

3.4 技术中间件

分库分表的中间件选取，在行业内的方案还是比较多的，公司也有自己的实现。根据实现方式的不同，可以分为代理和非代理方式，下面列举了一些业界常见的中间件，如下表（截至于 2021-04-08）：

名称	团队	关注度	最近发布	说明描述
MyCAT	Alibaba	8.8k	2 Nov 2020	代理服务、开源、多数据库、多租户等等
Atlas	360	4.5k	5 May 2015	基于MySQL-Proxy、稳定性待验证
Vitess	Youtube	11.6k	26 Jan 2021	代理服务、始于 Youtube，国内使用不多
Sharding-Sphere	Apache	13.6k	12 Jun 2020	轻量级组件、路由透明、MySQL
TDDL	Alibaba	1.3k	Archived	依赖diamond配置管理，使用不广泛
DAL	Trip.com	1.1k	14 Jul 2020	公司组件、Java和 C#语言、契合公司框架

我们为什么最终选择了 Sharding-Sphere 呢？主要从这几个因素考虑：

技术环境

- 我们团队是 Java 体系下的，对 Java 中间件有一些偏爱
- 更偏向于轻量级组件，可以深入研究的组件
- 可能会需要一些个性定制化

专业程度

- 取决于中间件由哪个团队进行维护，是否是名师打造，是否是行业标杆
- 更新迭代频率，最好是更新相对频繁，维护较积极的
- 流行度问题，偏向于流行度广、社区活跃的中间件
- 性能问题，性能能满足我们的要求

使用成本

- 学习成本、入门成本和定制改造成本
- 弱浸入性，对业务能较少浸入
- 现有技术栈下的迁移成本，我们当前技术栈是 SSM 体系下

运维成本

- 高可用、高稳定性
- 减少硬件资源，不希望再单独引入一个代理中间件，还要考虑运维成本
- 丰富的埋点、完善的监控

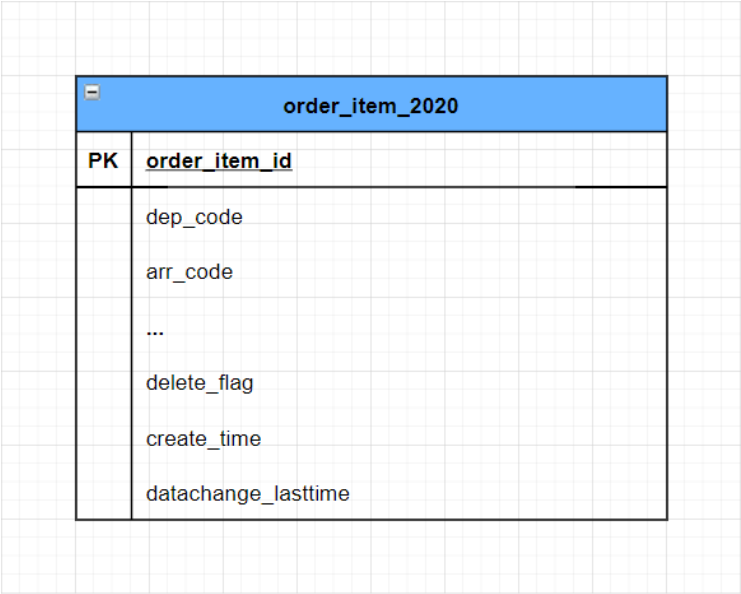
四、业务实践

在业务实践中，我们经历了从新库新表的设计，分库分表自建代理、服务收口、上游订单应用迁移，历史数据迁移等过程。

4.1 新表模型

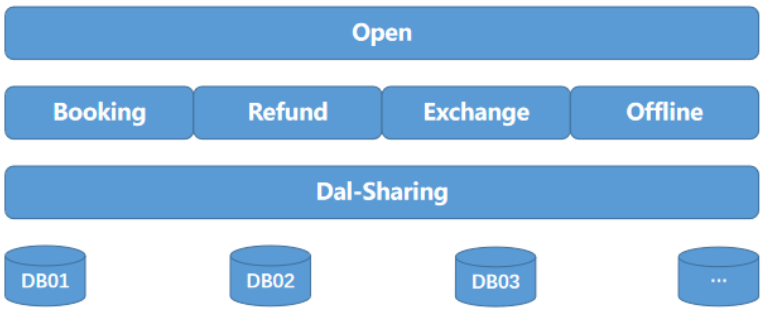
为了建立分库分表下的关联关系，和更加合理有效的结构，我们新申请了订单分库分表的几个库，设计了一套全新的表结构。表名以年份结尾、规范化表字段、适当增删了部分字段、不使用自增主键关联，采用业务唯一键进行关联等。

表结构示例如下图：



4.2 服务收口

自建了一个分库分表数据库的服务代理 Dal-Sharding。每一个需要操作订单库的服务，都要通过代理服务进行操作数据库，达到服务的一个收口效果。同时，屏蔽了分库分表的复杂性，规范数据库的基本增删改查方法。



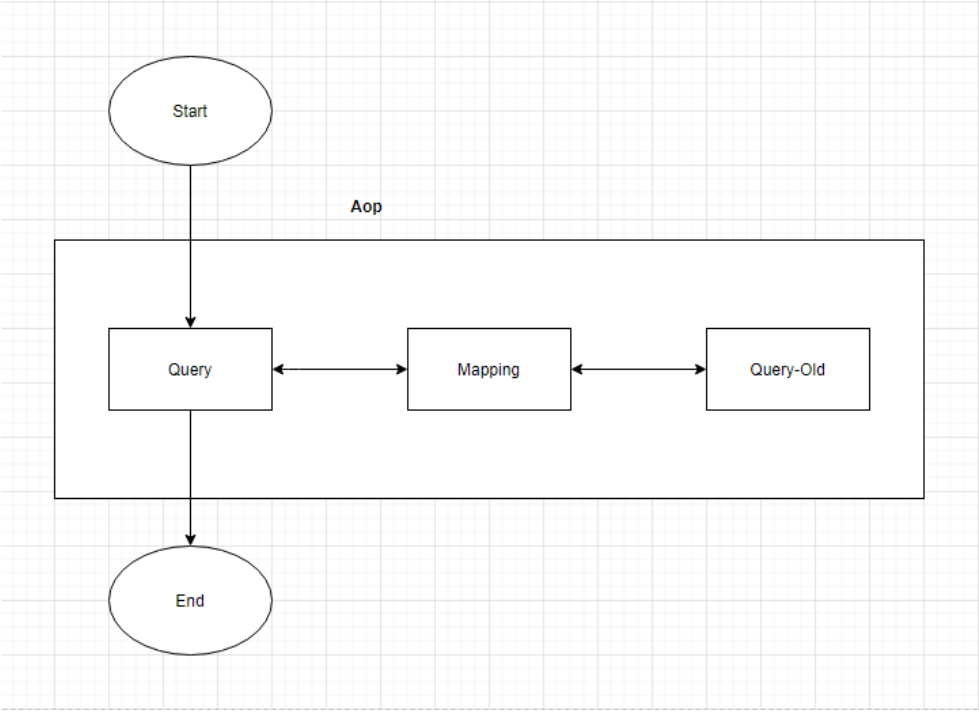
4.3 平滑过渡

应用迁移过程中，为了保证应用的平滑过渡，我们新增了一些同步逻辑，来保证应用的顺利迁移，在应用迁移前后，对应用没有任何影响。未迁移的应用，可以读取到迁移后应用写入的订单数据；迁移后的应用，能读取到未迁移应用写入的订单数据。同时，统一实现了此逻辑，减少各个应用的迁移成本。

新老库双读

顾名思义，就是在读取的时候，两个库可能都要进行读取，即优先读取新库，如果能读到记录，直接返回；否则，再次读取老库记录，并返回结果。

双读的基本过程如下：



新老库双读，保证了应用迁移过程中读取的低成本，上游应用不需要关心数据来源于新的库还是老的库，只要关心数据的读取即可，减少了切换新库和分库分表的逻辑，极大的减少了迁移的工作量。

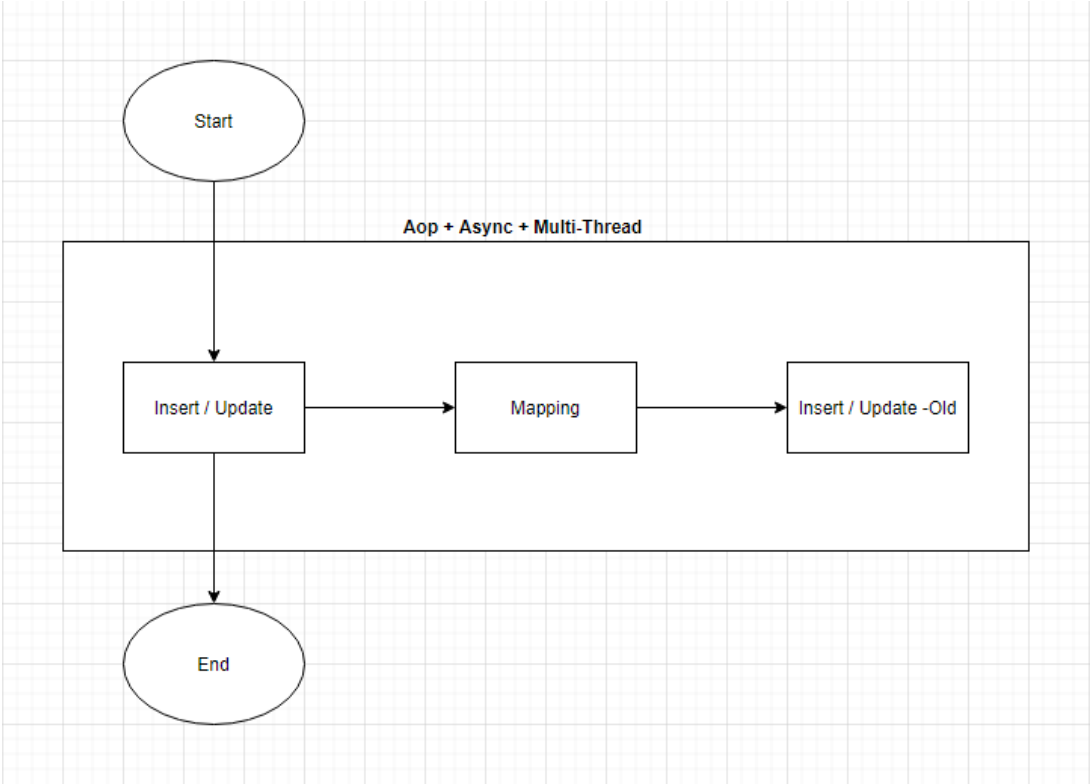
实践过程中，我们通过切面实现双读逻辑，将双读逻辑放入到切面中进行，减小新库的读取逻辑的侵入，方便后面实现对双读逻辑的移除调整。

同时，新增一些配置，比如可以控制到哪些表需要进行双读，那些表不需要双读等。

新老库双写

新老库双写，就是在写入新库成功后，异步写入到老库中。双写使得新老库都同时存在这些订单数据，尚未迁移通过代理服务操作数据库的应用得以正常的运作。

双写的基本过程如下：



双写其实有较多的方案，比如基于数据库的日志，通过监听解析数据库日志实现同步；也可以通过切面，实现双写；还可以通过定时任务进行同步；另外，结合到我们自己的订单业务，我们还可以通过订单事件（比如创单成功、出票成功、退票成功等），进行双写，同步数据到老库中。

目前，我们经过考虑，没有通过数据库日志来实现，因为这样相当于把逻辑下沉到了数据库层面，从实现上不够灵活，同时，可能还会涉及到一些权限、排期等问题。实践中，我们采取其他三种方式，互补形式，进行双写。异步切面双写，保证了最大的时效性；订单事件，保证了核心节点的一致性；定时任务，保证了最终的一致性。

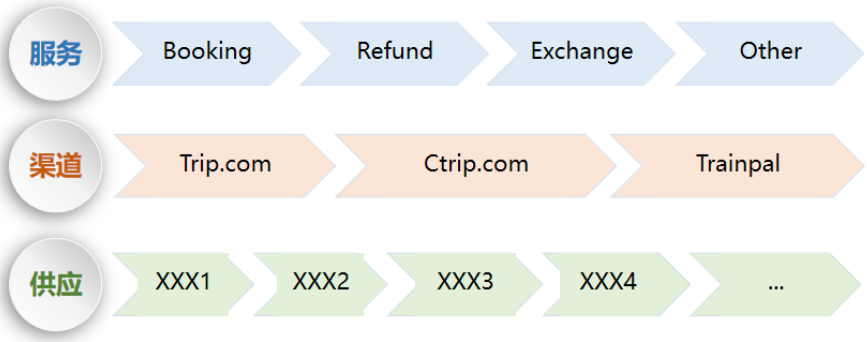
跟双读一样，我们也支持配置控制到哪些表需要进行双写，那些表不需要双写等。

过渡迁移

有了前面的双读双写作为基础，迁移相对容易实行，我们采取逐个迁移的方式，比如，按照服务、按照渠道和按照供应进行迁移，将迁移工作进行拆解，减少影响面，追求稳健。一般分为三步走方式：

1）第一阶段，先在新对接的供应商中进行迁移新库，因为新上线的供应商，订单量最少，同时哪怕出现了问题，不至于影响到之前的业务。
2）再次迁移量比较少的线上业务，此类订单，有一些量，但是追求稳定，不能因为切换新

库而产生影响。所以，将此类业务放到了第二阶段中进行。
3）最后一步是，将量较大的业务，逐渐迁移到新库中，此类业务，需要在在有前面的保证后，方能进行迁移，保证订单的正常进行。



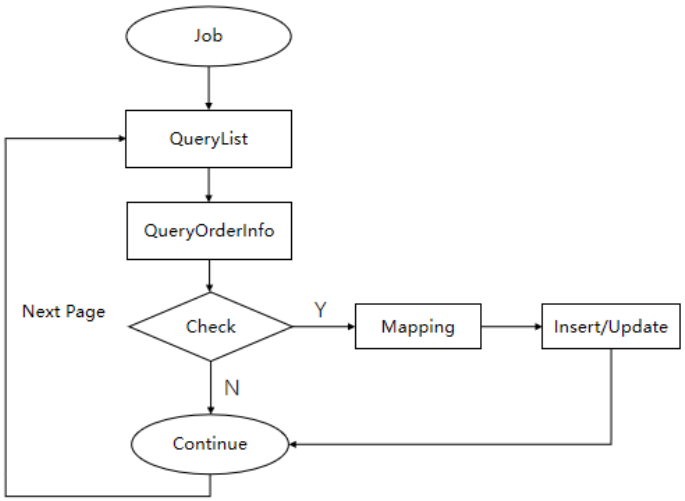
4.4 数据迁移

数据迁移，即将数据，从老库迁移到新库，是新老库切换的一个必经过程。迁移的常规思路，一般是每个表一个个进行迁移，结合业务，我们没有采取此做法，而是从订单维度进行迁移。

举个例子：假如订单库有 Order 表、OrderStation 表、OrderFare 表三个表，我们没有采取一个一个表分别进行迁移，而是根据订单号，以每一个订单的信息，进行同步。

大致过程如下：

1）开启一个定时任务，查询订单列表，取得订单号等基本订单信息。
2）根据这个订单号，去分别查询订单的其他信息，取得一个完整的订单信息。
3）校验订单是否已经完成同步，之前完成同步了则直接跳过，否则继续执行下一个订单号。
4）将老库的完整的订单信息，映射成新库的对应的模型。
5）将新的订单信息，同步写入到新库各个表中。
6）继续执行下一个订单号，直到所有的订单号都完全同步结束。



4.5 完成效果

订单库经过一个全新的重构，目前已经在线上稳定运行，效果显著，达到了我们想要的效果。

- 服务收口，将分库分表逻辑，收口到了一个服务中；
- 接口统一管理，统一对敏感字段进行加密；
- 功能灵活，提供丰富的功能，支持定制化；
- 分库分表路由透明，且基于主流技术，易于上手；
- 完善的监控，支持到表维度的监控；

五、常见问题总结

5.1 分库分表典型问题

问题 1：如何进行跨库操作，关联查询，跨库事务？

回答：对于跨库操作，在订单主流程应用中，我们目前是禁止了比如跨库查询、跨库事务等操作的。对于跨库事务，因为根据订单号、创建年份路由，都是会路由到同一个数据库中，也不会存在跨库事务。同样对于跨库关联查询，也不会存在，往往都是根据订单来进行查询。同时，也可以适当进行冗余，比如存储车站编码的同时，多存储一个车站名称字段。

问题 2：如何进行分页查询？

回答：目前在订单主流程应用中的分页查询，我们直接采用了 Sharding-JDBC 提供的最原始的分页方式，直接按照正常的分页 SQL，来进行查询分页即可。理由：主流程订单服务，比如出票系统，往往都是查询前面几页的订单，直接查询即可，不会存在很深的翻页。当然，对于要求较高的分页查询，可以去实现二次查询，来实现更加高效的分页查询。
问题 3：如何支持很复杂的统计查询？

回答：专门增加了一个宽表，来满足那些很复杂查询的需求，将常用的查询信息，全部落到此表中，进而可以快速得到这些复杂查询的结果。

5.2 API 方法问题

问题：服务收口后，如何满足业务各种不同的查询条件？

回答：我们的 API 方法，相对固定，一般查询类只有两个方法，根据订单号查询，和根据 Condition 查询条件进行查询。对于各种不同的查询条件，则通过新增 Condition 的字段属性来实现，而不会新增各种查询方法。

5.3 均匀问题

问题：在不同 group 中，数据会存在分布不均匀，存在热点问题？

回答：是的，比如运行 5 年后，我们拓展成了 3 个 group，每一个 group 中存在 3 个库，那么此时，读写最多的应该是第三个 group。不过这种分布不均匀问题和热点问题，是可接受的，相当于前面的两个 group，可以作为历史归档 group，目前主要使用的 group 为第三个 group。

随着业务的发展，你可以进行调配，比如业务发展迅速，那么相对合理的分配，往往不会是每个 group 是 3 个库，更可能是应该是，越往后 group 内的库越多。同时，因为每个 group 内是存在多个库，与之前的某一个库的热点问题是存在本质差别，而不用担心将单数据库瓶颈问题，可以通过加库来实现扩展。

5.4 Group 内路由问题

问题：对于仅根据订单号查询，在 group 内的路由过程是读取 group 内所有的表吗？

回答：根据目前的设计，是的。目前是按年份分组，订单号不会存储其他信息，采用携程统一方式生成，也就是如果根据订单号查询，我们并不知道是存在于哪个表，则需要查询 group 内所有的表。对于此类问题，通常推荐做法是，可以适当增加因子，在订单号中，存储创建年份信息，这样就可以知道对应那个表了；也可以年份适当进行延伸，比如每 5 年一次分表，那么这样调整后，一个 group 内的表应该相对很少，可以极大加快查询效能。

5.5 异步双写问题

问题：为什么双写过程，采用了多种方式结合的方式？

回答：首先，切面方式，能最大限度满足订单同步的时效性。但是，在实践过程中，我们发现，异步切面双写，会存在多线程并发问题。因为在老库中，表的关联关系依赖于数据库的自增 ID，依赖于表的插入顺序，会存在关联失败的情况。所以，单纯依靠切面同步还不够，还需要更加稳健的方式，即定时任务（订单事件是不可靠消息事件，即可能会存在丢失情况）的方式，来保证数据库的一致性。

参考连接

[1] Sharding-Sphere 概述
https://shardingsphere.apache.org/document/current/cn/overview/

[2] 大众点评订单系统分库分表实践
https://tech.meituan.com/2016/11/18/dianping-order-db-sharding.html

[3] Mycat 与 ShardingSphere 如何选择
https://blog.nxhz1688.com/2021/01/19/mycat-shardingsphere/

[4] 分库分表：如何做到永不迁移数据和避免热点？
https://mp.weixin.qq.com/s/-YNU6wDZ3_IEh7vlsslDfQ

秒级上下线，携程服务注册中心架构演进

【作者简介】Alex，携程资深软件工程师，关注微服务架构及分布式缓存技术。

一、前言

携程的微服务框架产品从 2013 年发展至今，已经历了 7 年多的打造。其中所使用的服务注册中心也从最开始人工数据维护架构演进到了现在全自动、百万容量级的架构。本文将逐一回顾携程服务注册中心所经历的三轮迭代过程，并重点介绍最新的第三版架构的设计与实现。

二、服务注册中心是什么?

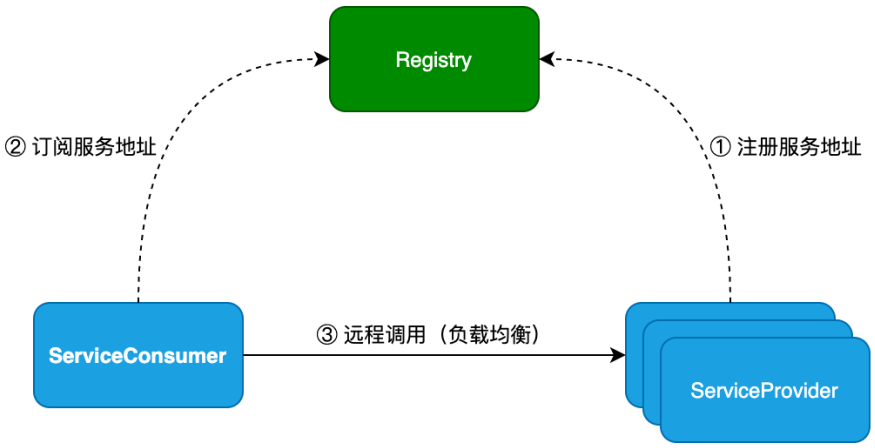


图 2-1 微服务架构

微服务架构中所要解决的最核心的技术问题有两点，一个是服务发现，另一个是负载均衡。而服务注册中心就是用来解决服务发现问题的。

如图 2-1 所示，在微服务架构中，服务提供方(ServiceProvider)，需要手动或自动地将服务地址注册到服务注册中心（Registry)。注册的信息包括但不限于 ServiceID 和 URL。服务消费方（ServiceConsumer）在首次调用服务前，需要先从服务注册中心查询对应服务的注册信息，然后依据返回的服务地址信息来发起调用。

三、携程服务注册中心演变史

3.1 人工数据维护阶段

在携程微服务架构推广初期，为了快速搭建微服务体系，服务的调用过程继续使用传统的域名方式来进行。在服务治理层面，服务提供方首先需要将服务的一个完整 URL 提交到注册中心。服务消费方在运行时会定期从注册中心同步最新的 URL，并发起服务调用。而这个 URL 与应用服务器的关联关系则由运维人员人工在负载均衡设备上配置。

这种模式下的服务注册中心的优点是结构简单、容易实现且运维工作量小，有利于微服务架构快速推广。而缺点则主要集中在以下几个方面：

- 配置复杂：负载均衡和服务发现的数据依赖人工维护，影响开发效率和体验。
- 单点问题：服务调用强依赖于负载均衡设备，该设备的可用性会直接影响到微服务体系的可用性。
- 性能问题：服务调用需要经过一层负载均衡设备，存在额外的网络开销，会直接影响到性能。

3.2 基于 etcd 的服务注册中心

在携程微服务体系扩展到 Java 平台时，我们希望能够解决前面由于使用外部负载均衡设备所带来的各种缺陷，所以计划将负载均衡设备的功能集成到微服务的 SDK 中，同时由注册中心下发的服务注册信息从之前的固定使用域名的 URL，改为服务集群各台服务器 IP 所对应的 URL。

改进后的工作流程是这样的：服务提供方启动后，SDK 会把包含本机 IP 的服务实例地址上报给注册中心；而服务消费方启动后，SDK 会定期从注册中心获取最新的服务地址列表，并使用内置的负载均衡算法选出一个地址来发起请求。同时，为了保证服务注册数据的有效性，其中设置有“存活时间”（TTL，Time to live）。所以需要服务注册中心支持清理过期的注册数据。在设计新的架构时，综合以上这些考虑，我们选择了 etcd 来存储服务注册数据。

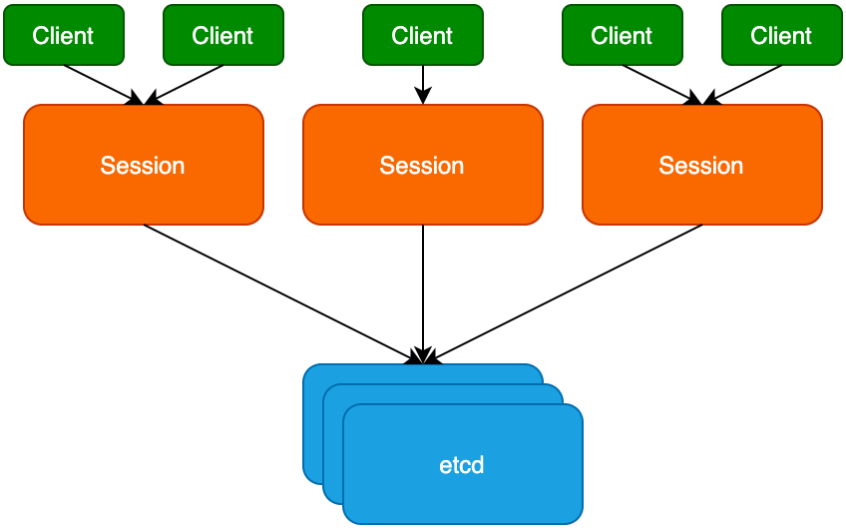


图 3-1 基于 etcd 的服务注册中心架构

基于 etcd 的服务注册中心整体架构，如图 3-1 所示，包含三个角色。

- Client

提供应用接入服务注册中心的基本 API。应用通过嵌入到应用程序内的 SDK，实现服务的自注册和自发现。

- Session

负责处理 Client 提交的服务注册和发现请求。Client 的请求经 Session 协议转换后，直接转发给 etcd。etcd 的响应数据经 Session 的服务治理逻辑处理后，再返回给 Client。

- etcd

负责存储服务注册数据。集群内各节点间使用 Raft 协议来进行数据同步。在没有网络分区的情况下，节点上数据可以做到完全一致。

etcd 满足 CAP（Consistency：数据一致性、Availability：服务可用性、Partition-tolerance：分区容错性）中的 CP，即优先考虑分布式缓存的数据一致性。从其设计的出发点来看，etcd 不适合对读写性能要求特别高的场景，而是适合量小且需要高可靠和一致性数据存储服务，比如配置数据、K8s 中的集群元数据等等。

在经过一段时间的线上部署和运维后，我们发现 etcd 中存在潜在的可用性和性能问题。

先说下可用性问题。假设 etcd 集群存在 A、B、C、D 和 E 五个节点，A 是当前集群的 Leader 节点。如果此时发生网络分区故障，其中 A、B 在一个分区，而 C、D 和 E 在另一个分区。Leader A 向所有的 Follower 发送心跳，但无法获取到大多数节点响应（计算公式为(N+2)/2，即在拥有五个节点的集群中需要至少获得三个节点的响应）。心跳超时后，集群进入选举阶段。但受到网络分区的影响，A 和 B 都无法获得大多数节点投票。所以由于缺少 Leader，A 和 B 所在的分区会处于不可用的状态，无法写入数据。

再说下性能问题。etcd 所有的写操作都由 Leader 节点负责执行。而自注册服务实例的健康检测，是依赖注册中心数据中的过期机制实现的。所以各个服务实例需要不断的发送心跳，来保持数据的活跃和有效。但这样就会产生大量的写操作，对 Leader 节点的性能和网络带宽都是一个极大的挑战。

在服务发现的场景下，服务注册中心的可用性比数据一致性更加重要。数据不一致可以通过客户端容错（比如熔断或踢出不可用服务器等），来把影响降到最低，甚至可以忽略不计。而可用性的下降将直接会导致服务的注册和发现异常，甚至会引发大规模的生产故障。

综合以上问题，并考虑到 etcd 无法很好的接入携程当时的运维和监控体系，我们走上了自研服务注册中心的道路。

3.3 携程自研的服务注册中心

在设计这套自研的服务注册中心时，我们参考了当时业界使用比较广泛的由 Netflix 开源的 Eureka。新版注册中心同样没有使用外部存储，而是将服务的注册数据保存在内存中。节点间采用对等的架构设计。所有节点都可以接受客户端的读写请求。节点间会进行数据同步，实现数据最终一致。

在基本的服务注册和发现功能外，为了提升效率，我们还在其中增加了服务变更通知推送功能。这样客户端可以以最快的速度获取到更新的服务注册信息，目前已经实现了服务实例的秒级上下线。

我们将这套全新的服务注册中心的开发代号起名为 Artemis。为了简单，后文中均以该开发代号来进行指代。

四、Artemis 架构说明

4.1 总体架构

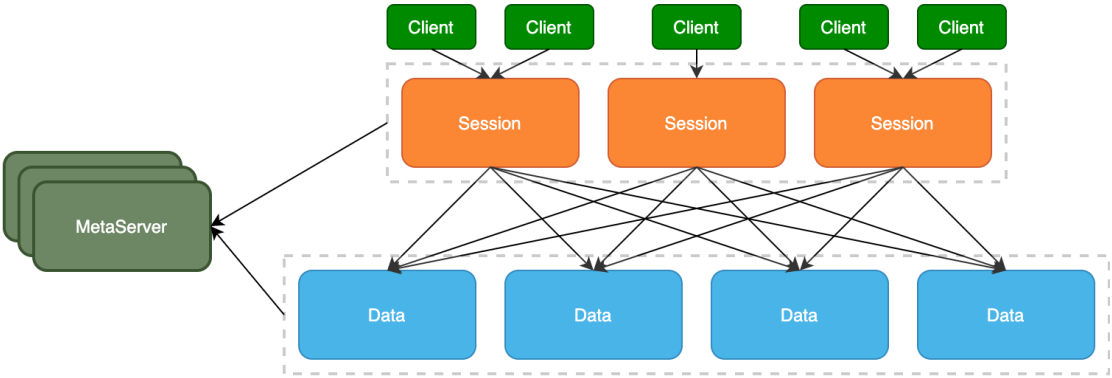


图 4-1 Artemis 架构

Artemis 的整体架构与基于 etcd 的服务注册中心类似。如图 4-1 所示，一共包含四个角色：

- Client

提供应用接入注册中心基本 API。应用通过引用 Artemis 对外提供 SDK，以编程方式实现服务注册和发现。

- Session

负责接受 Client 的服务注册和发现请求。Session 作为中间层将服务提供方的注册请求复制分发给 Data，并从 Data 上查询服务注册数据或推送数据变化给服务消费方。Session 节点自身是无状态的，集群规模可随着 Client 的规模增长而扩容，支持 Artemis 服务能力的水平扩展。

- Data

负责存储服务注册数据，数据按 ServiceId 进行一致性哈希分片存储，通过多副本备份保证数据的高可用。Data 集群规模可随着注册数据量增长而持续扩容，从而支持 Artemis 数据存储容量的水平扩展。

- MetaServer

负责从 K8s 同步 Artemis 集群服务器地址列表。在 Artemis 集群发生变化时，MetaServer 会实时通知到 Session。Session 在程序启动或者收到 Artemis 集群变化通知时，将主动从 MetaServer 拉取最新的 Artemis 地址列表并缓存到本地。

4.2 如何支持海量数据

分布式系统在处理海量数据时，首先是考虑如何拆分数据，其次是在数据拆后的如何保障系统的可用性。

Artemis 使用一致性哈希环来拆分数据。一致性哈希环的基本使用方式是通过一个哈希函数来计算数据或节点的哈希值，令该哈希函数的数据值域为一个环，即哈希函数输出的最大值是最小值的前序，节点依据其哈希函数计算结果分布在环上，每个节点负责处理从自己开始逆时针至下一个节点全部哈希值域上的数据。Artemis 使用服务注册数据的 ServiceId 来计算哈希值，这样可以保证同一个服务的注册数据可以被存储在相同的节点上，减少网络调用的操作量。

例：假设一致性哈希函数值域是[0, 8)，系统中有三个节点 A、B、C，分别处于一致性哈希环的 2、5、6 位置。由此可知，节点 A 的负责范围为 [7,8）和 [0,3)，节点 B 的负责范围为 [3, 6)，节点 C 的负责范围为[6, 7)，如图 4-2 所示。

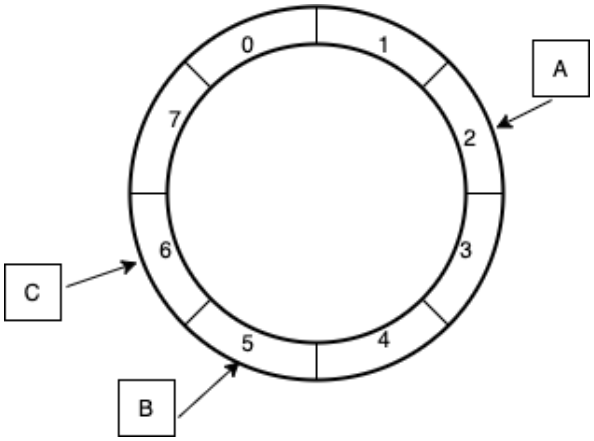


图 4-2 一致性哈希环

使用一致性哈希环拆分数据的优点在于可以任意动态扩容或缩容节点。对集群进行扩容或缩容的操作，仅会影响与被操作节点相邻的节点上的数据分布。

但最基本的一致性哈希环用法存在一个很明显的缺陷，那就是环上的节点分布不均匀。由此所带来另一个较为严重的问题是，当一个节点出现异常时，该节点的压力会全部转移到相邻的一个节点；而当一个新节点加入时，只能为一个相邻节点分摊压力。

一种常见的改进算法是引入虚拟节点（virtual node）的概念。系统在初始化时，每个真实节点都会对应的创建多个虚拟节点。虚拟节点的个数一般远大于集群中服务器的个数。依据虚拟节点的哈希值，系统将它们分布到环上。在操作数据时，首先需要通过数据的哈希值在环上找到对应的虚拟节点，然后从元数据中查找到对应的真实节点，再进行数据读写操作。使用虚拟节点有多个好处。首先，一旦系统中某个节点出现不可用，其对应的所有虚拟节点也会同时变为不可用，从而它的服务压力会被均衡的分配到多个相邻的真实节点上。同理，一旦系统中加入一个新节点，也将在环上引入多个虚拟节点，从而使得新节点可以均衡的分担多个真实节点的压力。从全局看，这种实现方式更加容易实现集群扩容时的负载均衡。

Artemis 使用一致性哈希环加虚拟节点的方法，实现了海量数据的分片存储和集群扩缩容时的负载均衡。而在数据拆分后的集群可用性方面，Artemis 则是通过数据副本策略来保障的。每一条服务注册数据同时被存储在多个节点上，其中一个是主副本节点，其余的是从副本节点。假如我们需要在集群中选择 N 个节点来存储同一条数据，那么在根据哈希函数计算出数据中 ServiceID 的哈希值后，系统会从哈希值落到环上的位置开始顺时针依次选择连续 N 个不同的真实节点来存储这一份数据的各个副本。

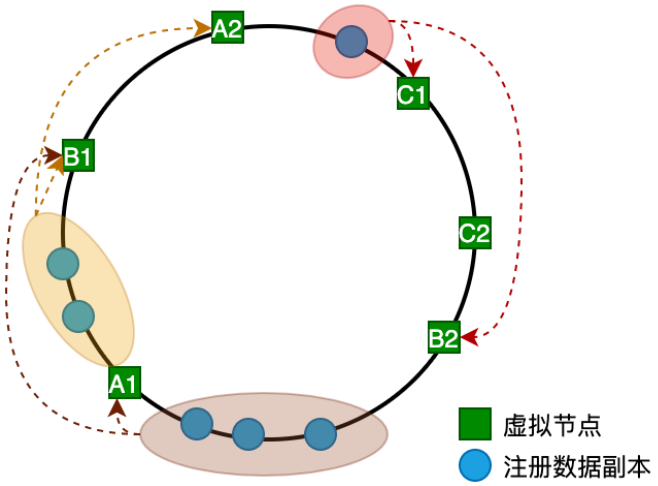


图 4-3 一致性哈希环（6 个虚拟节点，2 个注册数据副本）

4.3 服务实例秒级上下线

在设计服务注册中心时，一个重要的考量指标就是服务实例的上下线延迟。这个延迟是指从服务注册中心确定服务实例发生了上下线变更起，到服务消费方收到更新后的注册数据的时间间隔。延迟越高，则服务流量切换所需时间就越长，涉及到流量切换的场景（故障转移、服务发布）给用户带来的体验就越差。

为了降低服务实例的上下线延迟，Artemis 基于 WebSocket 实现了服务实例的上下线通知功能。通知可以秒级送达到服务消费方。这一功能的具体实现过程如下：

- 服务消费方在初始化过程中，会先经 Session 域名查询 Session 的 IP 地址列表并缓存到本地，然后再从列表中选择一台 Session 服务器与之建立 WebSocket 长连接，并发送服务订阅请求。
- Session 在收到服务订阅请求后，先会将服务订阅信息和 WebSocket 连接的映射关系存储到本地。后续当 Session 收到 Data 推送的服务变更消息时，它会先从上述映射关系中查询该服务对应的变更订阅方（即对应的 WebSocket 连接列表），然后将消息通过这些连接推送出去。
- 在收到服务变更消息后，服务消费方会根据消息的内容更新本地缓存中的服务地址列表。

4.3.1 服务实例上线过程

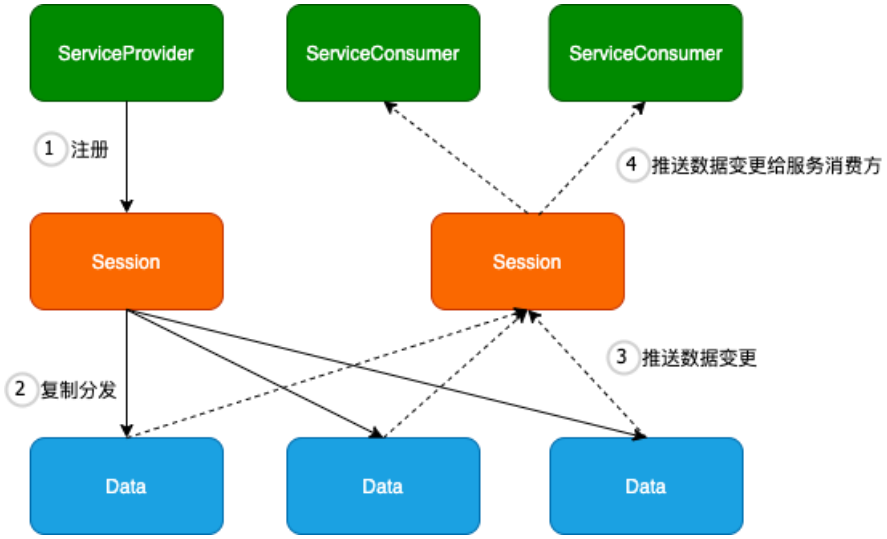


图 4-4 服务实例上线过程

如图 4-4 所示，这是一次服务实例正常上线过程，其中包含了服务注册数据在 Artemis 内部的流转过程。

- 服务提供方发送注册数据给 Session。
- Session 收到服务实例的注册数据，依据 ServiceID 在环上查找到相应的 Data 节点列表，再将数据写到对应的数个 Data 节点上。
- Data 在收到数据后，先将数据写入本地缓存，然后推送服务实例上线消息给所有的 Session 节点。
- Session 在收到服务实例上线消息后，将消息推送给对应的服务消费方。
- 服务消费方在收到服务实例上线消息后，将消息中所包含的服务地址加入到本地缓存中的服务地址列表，后续客户端 SDK 中的负载均衡模块将分配部分流量给新上线的服务实例。

4.3.2 服务实例下线过程

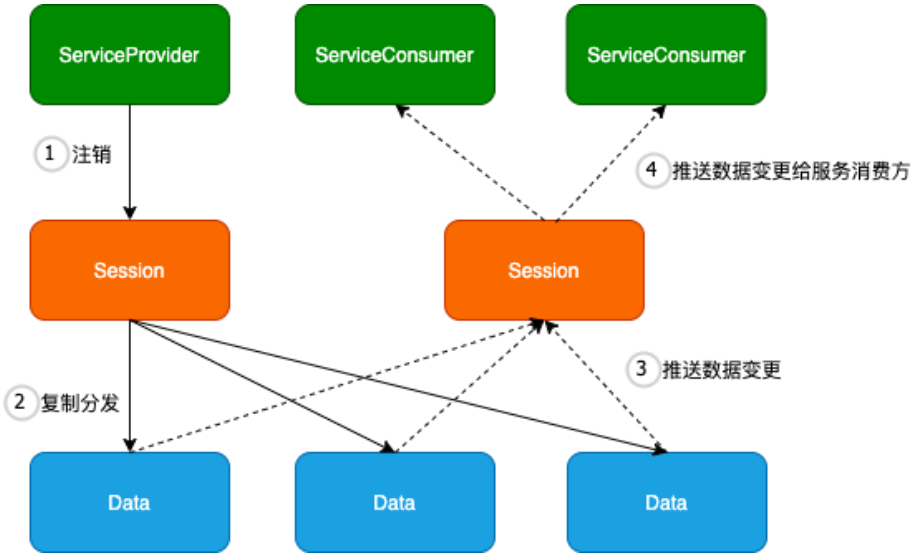


图 4-5 服务实例正常下线过程

服务实例下线分为正常下线和异常下线两种情况。正常下线并不会在服务消费方引起调用异常，而异常下线则可能会导致服务消费方出现短时间的调用异常。

服务实例正常下线，一般是通过监听应用程序关闭事件（如 JVM 的 Shutdown Hook），主动触发服务实例注销操作，将服务实例从 Artemis 中删除。服务下线大致过程与服务上线过程类似，这里就不再赘述了。

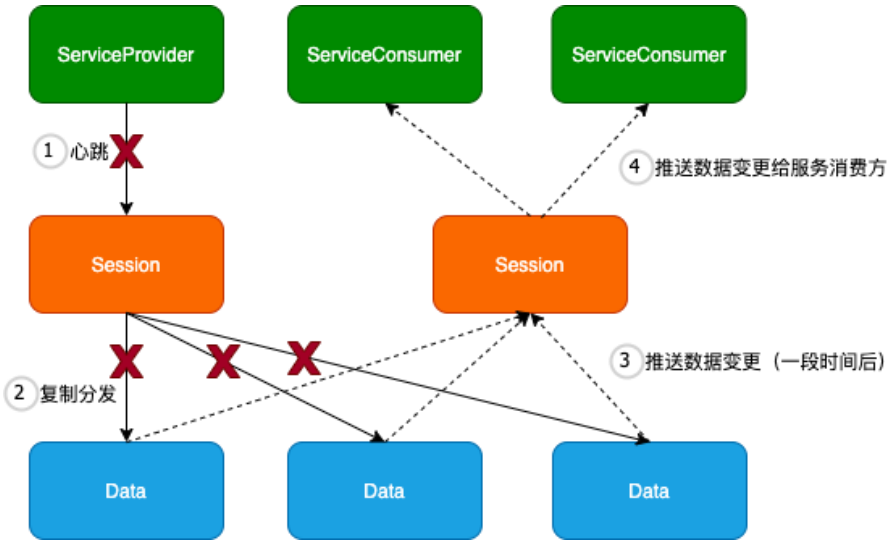


图 4-6 服务异常下线过程

服务实例异常下线，是指服务因意外情况（如宕机、网络中断或断电等）而不可用，但没有将注册数据从 Artemis 中删除。这些异常的注册数据，依赖 Artemis 的健康检测机制进行处理。类似于 Eureka 和 etcd 等系统中的数据过期机制，Artemis 中的服务实例注册数据以 Lease（租约）的形式存在，需要服务提供方不断发送心跳来续约。同时 Artemis 内部会运行一个异步线程来自动踢出到期的 Lease。异常下线的服务实例由于不会再继续上报心跳，它的注册数据在一段时间后（TTL）将自动被 Artemis 清理掉。

Artemis 也支持用户在客户端自定义健康检测逻辑，当应用程序不健康时，应用程序可以主动更新服务提供方的状态或停止上报心跳。那么服务提供方状态又是如何被服务消费方感知到的呢？当服务提供方注册数据修改后，服务注册数据会生成一个新的版本号（单调递增），并在下一次上报心跳时，发送给 Artemis。Artemis 在收到服务提供方的心跳后，会先检查心跳中服务注册数据的版本号。如果该版本号大于本地 Lease 中的服务注册数据版本号，Artemis 就会更新 Lease 中的服务注册数据，并生成一条服务变化消息，逐级经 Data、Session 推送给服务消费方。

五、小结

本文介绍了携程的微服务注册中心架构演进迭代过程，并重点介绍了当前版本服务注册中心（Artemis）的架构，以及海量数据存储和服务秒级上下线机制的实现。在携程微服务架构演进的过程中，服务发现的主要方式从手工维护，逐步过渡到自注册和自发现，解决了注册地址的维护复杂性问题。负载均衡的主要实现方式也从外部设备的负载均衡，逐步过渡到在应

用程序内嵌代理（SDK）的软件负载均衡，解决了单点问题和性能问题，但同时也带来了多语言、多版本 SDK 维护的复杂性问题。

现在，携程正在构建全新的 ServiceMesh 平台，计划以 K8s 替换 Artemis 来作为服务的注册中心，并通过 sidecar 模式将服务发现、负载均衡以及一些切面功能（例如熔断、限流、监控等）从 SDK 中剥离出来，使得这些功能可以独立于用户应用外进行更新升级。ServiceMesh 与云原生技术的推广，将极大的提升服务的治理效率，为携程的微服务开发者和使用者带来更上一层楼的使用体验。

参考文档

1、https://github.com/netflix/eureka

2、https://github.com/sofastack/sofa-registry

携程商旅订单系统架构设计和优化实践

【作者简介】Gavin，携程技术专家，致力于系统优化、重构和开发效率提升。

概述

携程商旅是一站式互联网差旅服务平台，为客户提供差旅管控、预订、出行、结算、成本、风控等服务的在线 TMC（Travel Management Companies）。主要产线有：机票、酒店、火车票、打车、接送机、包车、租车、汽车票等。携程商旅订单系统针对 B 端客户定制化需求多样性，商旅产线丰富性，TMC 业务与产品业务结合的复杂性等，对订单系统架构进行了优化。

一、背景介绍

订单系统作为入口，承上启下，涵盖了订单生命周期中所有的流程管理、TMC 管控、创单中心、费用中心、履约中心、服务中心、聚合中心以及投诉理赔管理等。订单打通用户、商家、产品、库存、订后等关键业务，是驱动交易全流程运转的核心。

订单系统具备的能力，可以按照下面两个维度和多个角度进行切入拆解：

- 客户维度

- 员工视角：预定、公账垫付、个人支付、行程管理、退改服务、物流跟踪等；
- 财务视角：成本中心、交易流水、费用明细、账单报表、报销凭证等；
- 审计视角：审批授权、差旅管控、数据合规、政策合规、风控、反作弊等。

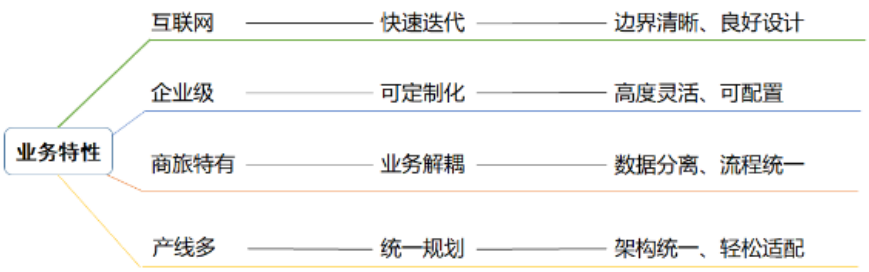
- 平台维度

- 客户实施视角：账户配置快照、订单聚合服务、API 对接等；
- 供应商视角：商家拆单、订单状态管理、费用结算、售后管理等。

携程商旅订单系统在早期为了快速满足业务需求，产线为纯纵向独立模式，在业务的快速上线、风险隔离，有效地降低产品之间的耦合性作用明显。随着业务的快速发展、产品丰富度增加、产品之间业务关联程度深化、客户的定制化需求增多和创新型产品快速上线，当前系统架构出现了单一臃肿、耦合严重、功能重复、流程不统一、监控和运维弱等各种问题。

- 无统一的流程引擎，不能以灵活、可配置的方式支持 B 端客户快速增长的定制化、碎片化需求；
- 无统一的编排服务，导致子系统之间高度耦合、相互调用、边界不清晰；
- 多产线纵向模式、流程不统一、相似功能重复开发，不能快速的支持业务迭代上线；
- 多产线的订单数据无统一抽象、设计、聚合；接入端学习成本高、接入工作量大；
- 实时场景（online）和非实时场景（offline）耦合、不能有效隔离和保护核心资源，降低了系统可靠性；

- 缺乏有效的自动化补偿机制、监控体系、数据埋点，不能满足 B 端客户越来越严苛的数据合规要求。



业务特性

二、基于业务特性的设计原则

2.1 分层设计 - B 端系统 C 端化

从预定视角能够感知到的系统功能：快速下单、支付、出票、确认供应商等。这一点无论 C 端还是 B 端都是相通的，使用对象都是“人”，核心目标就是确保用户的行程可以顺利起飞、出行、入住等。C 端系统的功能是 B 端系统的基础，从时间维度划分以下流程，在此基础上 B 端系统会有其特有的业务属性被嵌入。

这里需要避免按照传统的思维梳理流程图，将 B 端的功能与 C 端的功能作为一个整体的流程开发。B 端特有的逻辑和服务应该独立开发并且与产品、产线解耦，通过服务编排、流程引擎等实现功能可配置、可扩展、可插拔的方式对订单系统进行统一的调度和驱动，实现 B 端业务特性与 C 端逻辑分离和解耦，达到产品需求 C 端化。

- 下单 - 购买

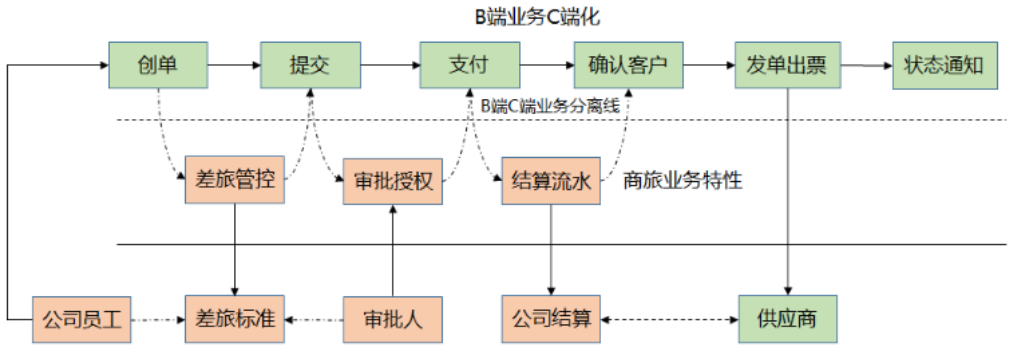
B 端嵌入功能：选择审批人、成本中心；差旅申请、管控校验；支付账号（公账）、费用政策、报销凭证规则等。

- 履约 - 支付、出票、发单等

B 端嵌入功能：发起审批（审批人通过、拒绝、终止）；公账扣款、公司信用支付、费用流水；公司对接推送等；

- 服务 - 出票、出行等

B 端嵌入功能：退改审批流程、管控校验、打卡校验；订单金额、订单状态、票号状态终态的合规验证；未出行订单自动处理流程；费用流水处理、报销凭证处理等；



B端系统C端化

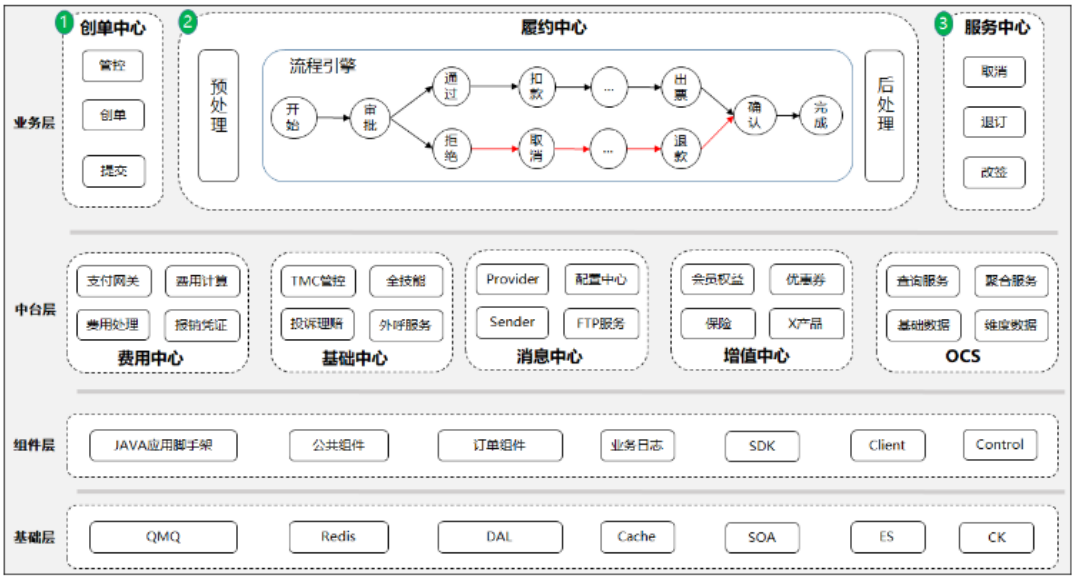
2.2 流程设计 - 产线多、共性小

从多产线中抽取出共性（去重复、标准化），针对 B 端系统所具有的定制化、碎片化、特定性等需求，提供通用、统一的标准化流程方案。在标准化流程的基础上再结合订单系统特有的订单状态、订单事件、票号状态等，抽象出流程引擎，实现可配置、可扩展的标准化流程，从而达到化繁为简的效果。

2.3 快照设计 - 管控粒度细、维度多

差旅管控信息（账户配置、支付账号、审批授权、差旅标准、政策执行人等）作为差旅业务的基础信息，对其任何的改动都至关重要，一方面把每次的更改记录都保存下来。另一方面在每次交易订单的各个场景，需要将当时差旅管控信息持久化。以此作为各个子系统交易和运行的基础服务实现统一收口，同时以便问题的排查、处理投诉、维权、溯源等。

三、架构演进



系统架构图

3.1 Order Consolidation - 数据抽象

纵向模式向分层模式演进最大的挑战是不同产线之间的数据库的横向，任何数据层面的改动和迁移犹如行驶中的汽车更换轮胎。系统在初期通过纵向的订单详情接口满足前端、各子系统、IM 等不同场景，此模式简单清晰，数据全量输出，但是带来的问题也是显而易见的：DB 压力过大、接入复杂、学习成本高、数据耦合等。

通过借鉴数据仓库建模的方式，以 offline 系统思维来降低 online 系统的复杂性。以子系统的数据详情服务作为事实表 Fact，根据不同的业务场景和粒度输出维度表 Dimensions。较少的系统是从头开始、从轮子开始建造，都是在已有基础上优化和迭代，通过通用数据冗余和增加维度数据聚合输出服务，实现数据层面的横向。

1）数据建模、数据冗余，降低系统复杂性

订单数据进行统一的抽象建模。以订单基础数据、常用数据、各子系统数据行程服务， 报销凭证，退改服务，管控服务等详情数据作为事实表 Fact， 建立各种维度表 Dimension：订单基础数据、产品基础数据、账户配置快照、行程信息、支付费用信息、TMC 管控详情等。

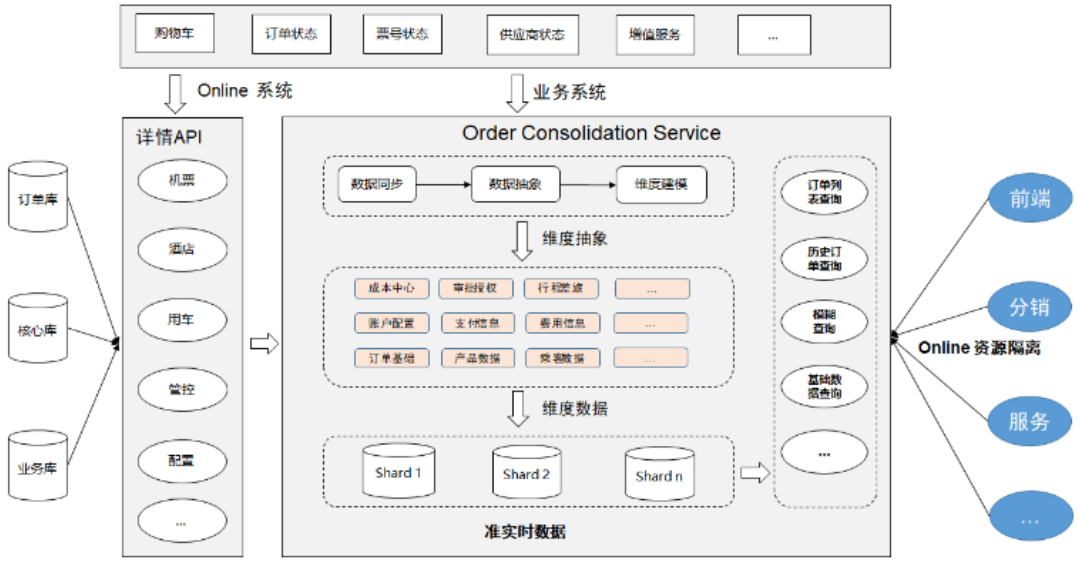
2）核心资源隔离和保护、确保关键流程高可靠性，支持单量的快速增长

随着运维能力和分布式系统的发展成熟，业务系统的 API 和微服务较容易水平扩展。性能瓶颈不容易解决主要是在 DB 层面，并不能轻易的扩展分库分表或者引入新的存储服务。订单库和 online DB 是整个订单系统交易的基础，降低数据库 IO 压力、保护关键路径和确保订单系统全流程的驱动顺畅需要尽可能的减少 DB 执行负荷重的任务：通用数据查询、列表查询、复杂查询、聚合查询、模糊查询、高频查询等，而这些场景基本上做到 NRT（near real-time）准实时输出即可。OCS 通过冗余落地的准实时数据和提供一系列的 API 服务达到隔离核心资源的目的。

3）大幅降低客户端学习成本、接入成本，支持业务的快速迭代

一个订单从预定到最终的结算经过较多的子系统和数据处理，整个订单生命周期产生的数据大而全、零碎、分散在各个 DB 和各个子系统，对于 client 端接入需要感知较多的逻辑和熟悉众多的交互 API，不仅学习成本高，同样一个场景在不同的产线需要多次实现。

对于较多的场景如列表查询、基本数据查询、交叉推荐查询等通过数据的统一抽象、融合、落地实现流程统一、API 统一、交互统一大幅简化接入端的成本，包括产品、BI、业务的取数规则等。



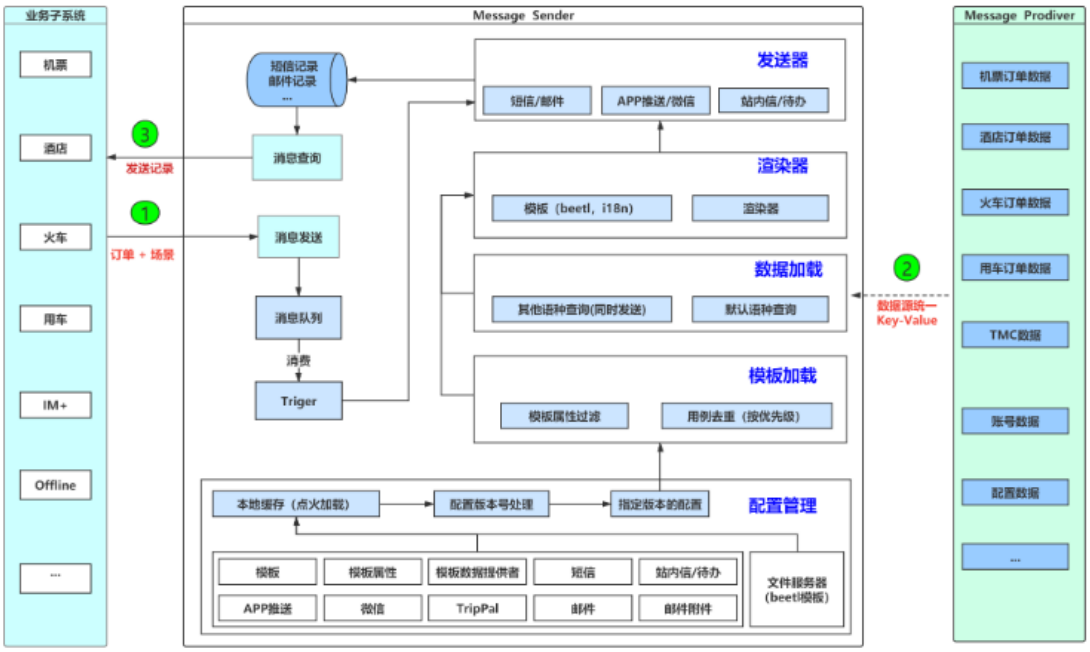
Order Consolidation Service

3.2 Order Message - 客户端减负

设计 API 和服务时通常考虑较多的是 API 如何设计、交互、传参、client 需要遵循的各种规则，较少的考虑 client 可以不做什么，client 在享受服务的同时应该可以不做一切可以不做的事情。传统上消息发送服务设计为提供不同消息的发送接口，由各客户端根据各自的业务流程处理指定模板、类型、动态内容等调用对应的接口， 这个看起来是符合高内聚、低耦合的通用设计原则的。

原流程发送一个消息需要 client 端调用多个数据源查询数据、各种逻辑判断、消息类型、发送场景、模板处理等。新的设计将各个消息抽象为产线，场景模式对应一套配置和一套数据源 API（统一为 key-value 输出），注册在消息中心，类似网卡插入主板后，需要安装驱动程序实现可插拔的方式大幅减少 client 端的工作，client 只需要传一个场景即可实现一系列消息的发送。根据实际的业务场景，需要根据下面 6 种场景的排列组合，触发一个或多个消息的发送。

1）发送场景：创单、支付、确认客户、出行、成交等。
2）发送模板：支付类型、订单处理阶段、订单状态等。
3）发送语言：中文简体、中文繁体、英文、日文、韩文等。
4）发送类型：短信、邮件、微信、企业消息、站内信、手工操作等
5）收 件 人： 联系人、预订人、出行人、授权人。
6）数据来源：订单数据、成本中心子系统、审批子系统、账户配置子系统，消息附件等



通用消息中心 - 配置服务

3.3 Order Conductor - 服务编排

1）服务拆分和边界确定之后，系统之间的交互同样需要统一的规则和整体上的设计。尤其需要避免单一事件触发后，需要调用一系列的 API 和较为复杂的交互流程，导致调用链过长，服务之间相互调用、耦合严重等问题。

- 每个服务都需要感知下游服务；
- 业务规则治理和调用的灵活性、扩展性较差；
- 不能做到真正意义上的事件驱动；

2）服务编排是根据业务逻辑，以串行、并行和分支等结构编排多个 API 及函数服务为工作流，达到统一调度和指挥各个业务子系统，同时更加有利于事件驱动机制的落地。极大简化了多个服务之间组合调用的开发和运维成本，更加专注于业务本身。要实现服务编排、自由组合、灵活扩展满足下面的条件是基本前提。

任务原子化：每个接口或 API 拆分合理、职责单一，实现任务的原子性。

数据私有化：每个任务都有一套对应的数据表，并且仅自己本身可以操作和访问。如果没有实现数据库和资源的隔离，那任务原子化也就不存在，需要避免多个任务读写同一数据表和资源。

契约相通：任务编排，契约相通才能实现服务的统一调度。契约的设计必须是高度精简、覆盖全场景的。订单系统全流程驱动中一定会有的是：订单号、场景。如事件：退票完成，各被调用的服务通过数据反查获取数据，实现服务之间通信的契约大幅简化， 而不是依赖调用端的各类传参。

事件驱动：定义全流程统一的事件列表枚举，非查询类的操作完成后触发其对应的事件通知编排服务中心，根据流程配置统一调度实现服务编排。尤其需要避免的是各个服务之间相互直接调用，相互订阅广播通知导致系统的耦合和边界不清晰，以及服务扩展较差。

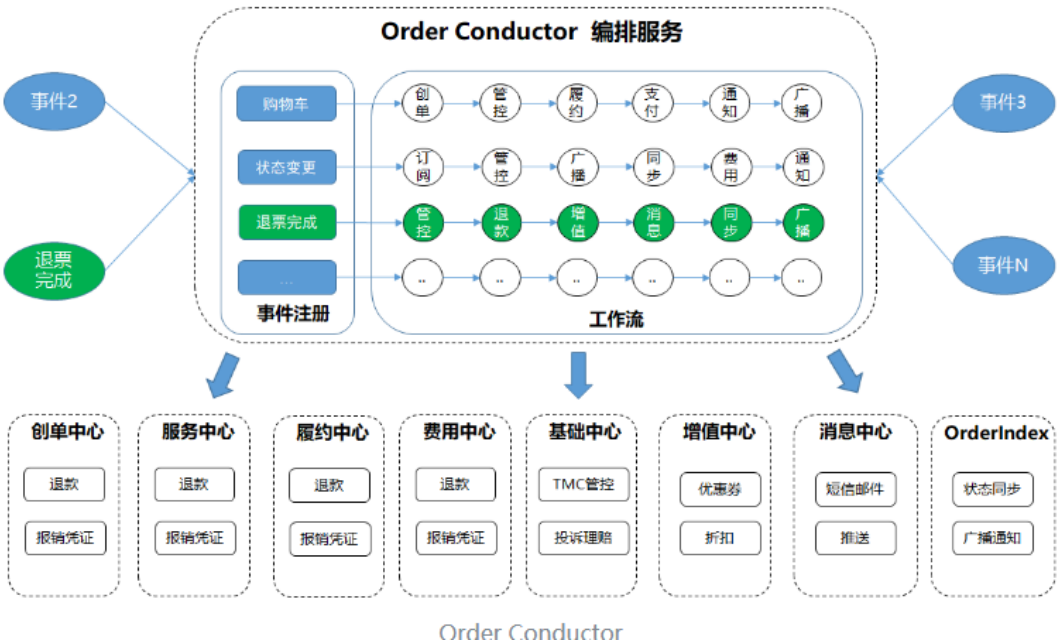
服务幂等性：服务的统一调度，不可避免的会出现多次被调用和触发的可能，每个原子性任务都需要根据自己的业务支持幂等性。

重试机制：与服务幂等性相辅相成、缺一不可，编排服务中心需要统一调度，就需要在配置和设计上支持任务的可重试、可降级和最终到达。

DB 访问降频：由于任务的原子性拆分，每个任务执行依赖的数据都需要调用详情查询服务获取需要的数据，即使是相同的数据也是需要各自获取导致数据库 IO 压力的大增。针对这个场景有很多种方案。

- 通过建立全局的 thread context 查询一次，各子线程或分支线程统一依赖降低 DB 的频繁访问。优缺点很明显的方案，维护和数据隔离较差，尤其是跨线程、子线程存在数据被篡改的风险和其他问题；

- 数据冗余、准实时同步、隔离核心资源以 offline 思维降低订单库和 online DB 的访问。通过维度建模，满足基本查询、通用查询、高频查询等场景实现流量的分流。由于统一的抽象，对于数据的输出更加有利于服务端的缓存建立。比如我们采用上一段落介绍的 OCS 方案。



Order Conductor

3.4 Order Fulfillment - B 端场景

B 端用户需求与传统的企业级软件有非常多的相似点，定制化和多样性是最为显著的共同点。商旅 B 端客户即使是基本的功能：支付流程、发单方式、管控流程、通知内容等都有显著的

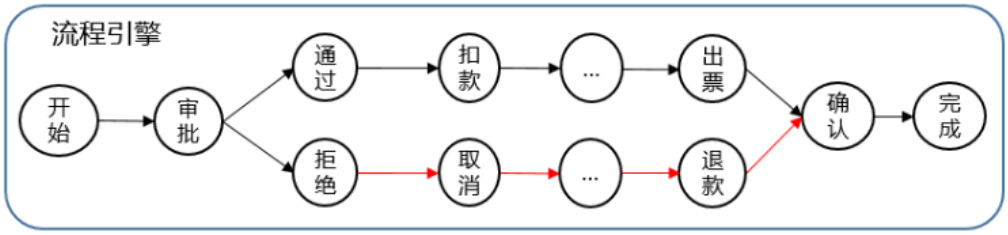
差异。以订单系统酒店产线举例，下面的订单属性和订单状态各个值排列组合出 50 种以上需要处理的流程，这仅是酒店一条产线。

订单属性：订单类型、授权顺序、支付方式、垫资方式、配置流程、国际国内、三方协议等，

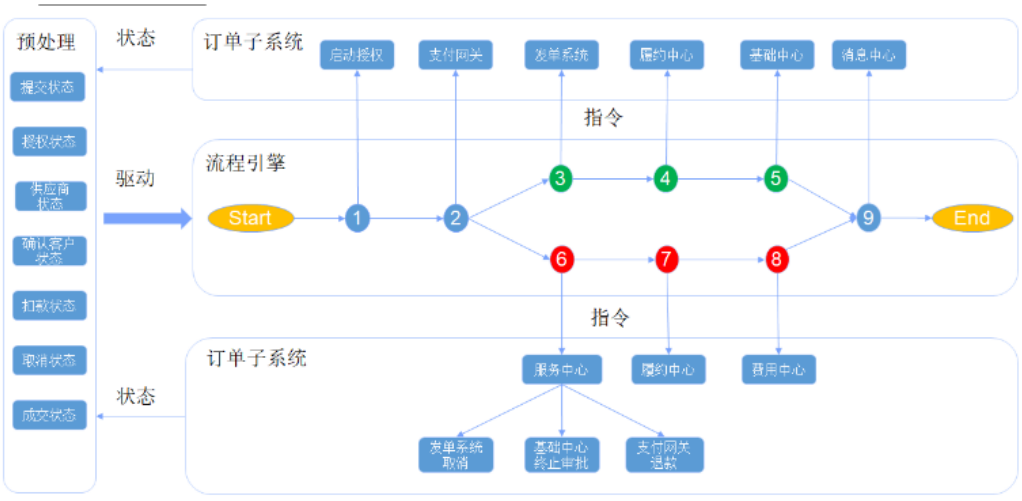
订单状态：已提交、未提交、授权通过、授权拒绝、已支付、已出票、已取消、确认客户等；

不仅微服务之间需要统一调度和流程配置，而对于每个应用或每个服务同样如此。订单系统仅通过工作流驱动并不能满足要求，其最为重要的属性就是订单状态、订单事件等触发和驱动订单系统的流转。通过开发基于订单状态、订单事件结合工作流开发流程引擎达到可配置、可扩展、易维护的目的，而又避免引入较笨重的开源工作流引擎达到“够用即可”：

- 有状态、可重试、支持幂等，
- 任务持久化、可配置、可扩展、易维护，
- 基于状态、事件的统一处理流程；



流程引擎



状态驱动 + 流程引擎

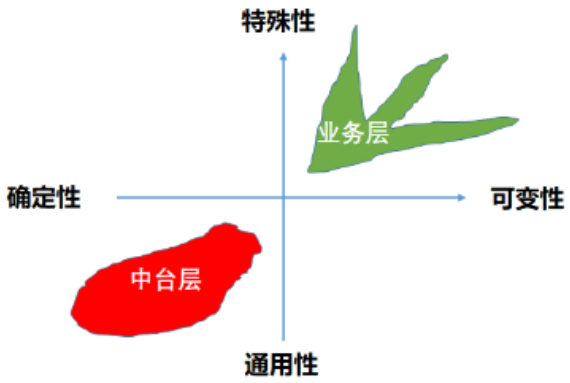
四、论述与总结

什么是好的设计？

- 一个好的设计不是还能增加什么，而是不能再减少什么。

- 一个好的设计不是客户端需要做什么，而是可以不做什么。
- 一个好的设计是自动化一切可以自动化的东西。
- 一个好的设计是标准化一切可以标准化的流程。

什么应该横向？什么应该纵向？



纵横结合

什么应该纵向？业务上共性少，差异大，对于客户需求需要快速响应，业务上需要快速试点和产品创新，正如图第一象限业务层展示的特点：特殊性，可变性。所以业务层在系统上应该纵向，隔离，但是在流程上标准化，可配置化，确保其灵活快速响应，尽可能少的涉及具体实现。

什么应该横向？高确定性、高通用性功能应该横过来供不同的团队和子系统共享，作为各个产线子系统的基础，同时也是支持业务在已有系统上快速上线，创新，试点的平台。

当业务需要快速的上线和需要速度运行，快速推向市场的时候，我们应该让业务、产线分开快跑，应该把业务变成纵向的，独立成建制的往前快速推进，摧枯拉朽，策马狂奔。随着单量的增长和业务的延伸，尤其各产线趋同功能变多，定制化需求和创新性需求增多，合规性性要求越来越严格，系统就暴露出大量问题：大量低水平重复建设，重复开发，效率低下，运维困难，任何新功能，新产品上线都需要从轮子开始建造。当我们需要效率、要积累、要沉淀的时候，尤其需要在已有系统上快速创新、试点新的产品时候，就非常明显要把有些东西横过来，从整体上规划和设计，让整个系统架构统一，流程统一成支撑体系包括技术沉淀，子系统中台化演进让其能够有办法共享给其他子系统和团队。

什么叫中台，为什么需要它？中台被过度神化过，也被极端的污名化。中台并无对错，主要取决于设计与边界。其被诟病主要原因：

1）圈地运动：什么都能做，什么都可以做，基于不能重复建设的原则各 client 被强制要求接入。基于原则而实际上是无原则的开发模式，导致系统臃肿、边界不清晰、耦合严重，不能通用而被通用只能通过给 client 制定各种条条框框和特殊定义来约束系统之间的交互、接入成本和沟通成本高，反映不够灵活，对于创新型的业务不能快速的支持，需要反复的沟通和规则制定，不仅没有享受到便利，反而成为了负担。是一种明显的山头主义，势力范围的

划分。

2）代理式中台：高举高内聚、低耦合的伟大旗帜，这也不能做，那也不能做，将 client 视为负担而非流量。client 需学习各种接入规则和传参契约，接入成本高和使用复杂，不同渠道稍有逻辑不一致需要 client 自行处理，某种意义上说仅起到转发的作用。中台实际上是一个横向策略。如果你要速度，要快速，要机动灵活，一定是这根杆子从上到下，都是一个人或一个团队负责，这是最快的。而如果 client 需要感知各种流程，等于是逻辑分散，没有做到真正的抽象和下沉。业务中台需从整体上设计基于配置化编程满足不同渠道的需求，而又不打破低耦合的原则，而非仅仅是个代理，是个类似基础架构的网关。

3）资源垄断：垄断了资源输出和对外交互，client 除了接入无其他选择。同时又受制于团队体量，团队话语权，ROI 等各种因素得不到排期或列为低优先级任务。

后微服务时代，领域驱动设计在携程国际火车票的实践

【作者简介】Ma Ning，携程国际火车票后端开发工程师，关注系统架构、微服务、高可用等技术领域。

一、前言

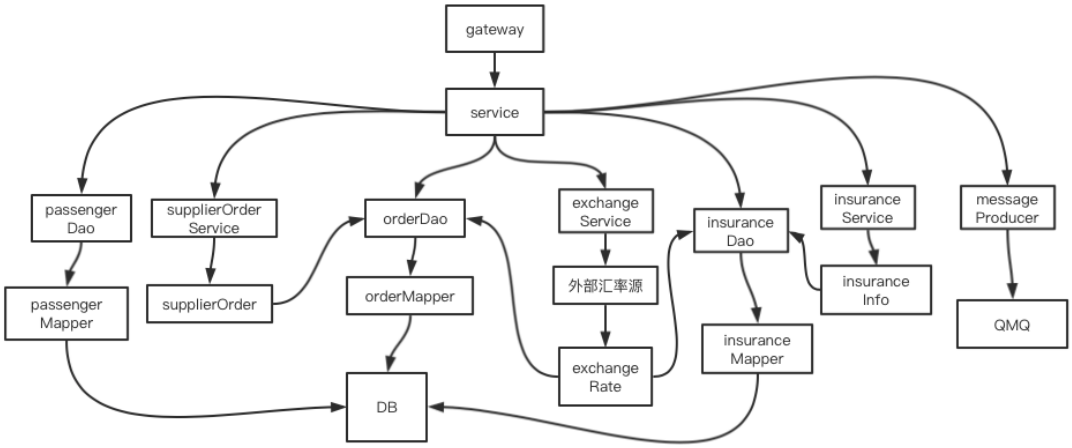
领域驱动设计（Domain-Driven Design，简称 DDD）是一种软件开发设计思想，其旨在以领域为核心，让软件系统在实现时准确地基于对真实业务过程的建模，专注于业务问题域的需要。

DDD 将软件系统设计分为了 2 个部分：战略设计和战术设计，战略设计用于提炼问题域并塑造应用程序的架构，战术设计用于帮助创建用于复杂有界上下文的有效模型。基于此，DDD 强调专注于核心领域，通过协作对公共语言和知识进行提炼，并且持续致力于领域的知识提炼，让模型持续发展。

本文基于 DDD 思想，在携程国际火车票中台预订系统项目进行实践。

二、实践背景

本文以国际火车票中台预订系统项目的创单流程为例，其服务结构下图所示：



伪代码如下所示：

```
@Override
protected CreateOrderResponse execute(CreateOrderRequest request) {
    // 1、参数校验
    if (!validate(request)) {
```

```
            throw new BusinessException(P2pBookingResultCode.PARAM);
        }
        if (orderMapper.select(request.getOrderId()) != null) {
            throw new BusinessException(P2pBookingResultCode.ORDER_EXISTS);
        }

        // 2、初始化订单
        OrderDao orderDao = new OrderDao();
        orderDao.setOrderId(request.getOrderId());
        orderDao.setOrderStatus(100);
        orderMapper.insert(orderDao);
        // 初始化乘客信息
        PassengerDao passengerDao = new PassengerDao();
        ...
        passengerMapper.insert(passengerDao);

        // 3、转换汇率
        ExchangeRate exchangeRate = exchangeService.getExchangeRate(originCurrency,
targetCurrency);

        // 4、购买保险
        if (isBuyInsurance(request)) {
            // 调用保险服务
            InsuranceInfo insuranceInfo = insuranceService.buyInsurance(request);
            // 保存保险信息
            InsuranceDao insuranceDao = new InsuranceDao();
            ...
            insuranceMapper.insert(insuranceDao);
        }

        // 5、供应商创单
        SupplierOrder supplierOrder = supplierService.createOrder(request, exchangeRate);
        // 保存供应商订单信息
        SupplierOrderDao supplierOrderDao = new SupplierOrderDao();
        ...
        supplierOrderMapper.insert(SupplierOrderDao);

        // 6、保存订单信息
        orderDao = new orderDao();
        orderDao.setOrderId(request.getOrderId);
        orderDao.setOrderStatus(OrderStatusEnum.WAIT_FOR_PAY.getCode());
        ...
        orderMapper.update(orderDao);
```

```
        // 7、发送超时支付取消消息
        messageProducer.push(MessageQueueConstants.TOPIC_TIMEOUT_CANCEL, "orderId",
String.valueOf(orderDao.getOrderId()), appSettingProp.getTimeoutMinutes(),
TimeUnit.MINUTES);

        // 8、返回结果
        return mappingResponse(orderDao, orderInsuranceEntity, exchangeRateResponse);
}
```

2.1 控制层臃肿

在传统的互联网软件架构中，通常都会采用 MVC 三层架构，其是一种古老且经典的软件设计模式，基于分层架构的思想，将整个程序分为了 Model、View 和 Controller 三层：

- Model（模型层）：最底下一层，是核心的数据，也就是程序需要操作的数据或信息；
- View（视图层）：最上面一层，直接面向最终用户的视图，它是提供给用户的操作界面，是程序的外壳；
- Controller（控制层）：中间的一层，就是整个程序的逻辑控制核心，它负责根据视图层输入的指令选取数据层的数据，然后对其进行相应操作产生最终结果；

MVC 三层架构模式，将软件架构分为了三层，就可以让软件实现模块化，使三层相互独立，修改外观或者变更数据都不需要修改其他层，方便了维护和升级。但是这种软件架构中模型层只关注数据，控制层只关注行为，随着迭代的不断演化，业务逻辑越来越复杂，便会导致整个控制层的代码量越来越多，而模型层和视图层的变更却很少，最终导致整个控制层变得十分臃肿，从而失去了分层的意义。

2.2 过度耦合

在业务初期，程序的功能都非常简单，此时系统结构逻辑是清晰的，但是随着程序的不断迭代，一方面会导致业务逻辑越来越复杂，系统逐渐冗余，模块之间彼此关联，软件架构设计模式逐渐向“大泥球”模式（BBoM，Big Ball of Mud）发展；另一方面系统会调用越来越多的第三方服务，从而导致数据格式不兼容，业务逻辑无法复用。

在出票系统中，除了订单相关的功能外，还包括了保险、汇率、供应商订单等多个服务接口，同时包括保险、供应商订单、乘客等多个模块的功能及存储均耦合在出票流程的控制层中，使得我们在维护代码时，修改一个模块的功能可能会影响到其他功能模块。

另一方面，如汇率服务这种第三方接口也会存在结构不稳定的情况，当其 API 签名发生变化或者服务不可靠需要寻找其他可替代的服务时，整个核心逻辑都会随之更改，迁移成本也是巨大的。

2.3 失血模型

失血模型是指领域对象里只有 get 和 set 方法的 POJO，所有业务逻辑都不包含在内而是放

在控制层中，该模型对象的缺点就是不够面向对象，对象只是数据的载体，几乎只做传输介质之用，它是没有生命、没有行为的。

与失血模型相对应的就是充血模型，充血模型就是会包含此领域相关的业务逻辑等，同时也可以包含持久化操作，它的优点对象自洽程度很高，表达能力很强，可复用性很高，更加符合面向对象的思想。

对于创单流程中的对象几乎都是使用的失血模型，虽然可以完成功能的实现，但是在系统逐渐迭代，业务逻辑逐渐复杂后，采用失血模型会导致业务逻辑。状态散落在大量的方法中，使得代码的意图渐渐不够明确，代码的复用性下降。

三、DDD 设计

通过上文的背景介绍，我们基于 DDD 思想对携程国际火车票中台预订系统做出了一定的重构，使系统实现高内聚、低耦合。

3.1 系统设计

Evic Evans 将软件系统的设计分为 2 个部分：战略设计和战术设计。战略设计提出了域、子域、限界上下文等概念，主要用于指导我们如何拆分一个复杂的系统，战术设计提出了实体、值对象、聚合、工厂、仓储。领域事件等概念，主要用于指导我们对于拆分出来的单个域如何进行落地，以及落地过程中需要遵循的原则。

3.1.1 战略设计

通用语言

对于国际火车票中台预定系统，我们定义了预定的通用语言：

- 通过用户搜索条件调用供应商下单；
- 记录供应商相关数据用于财务统计；
- 根据用户选定币种做汇率转换；
- 根据用户选择购买保险；

领域

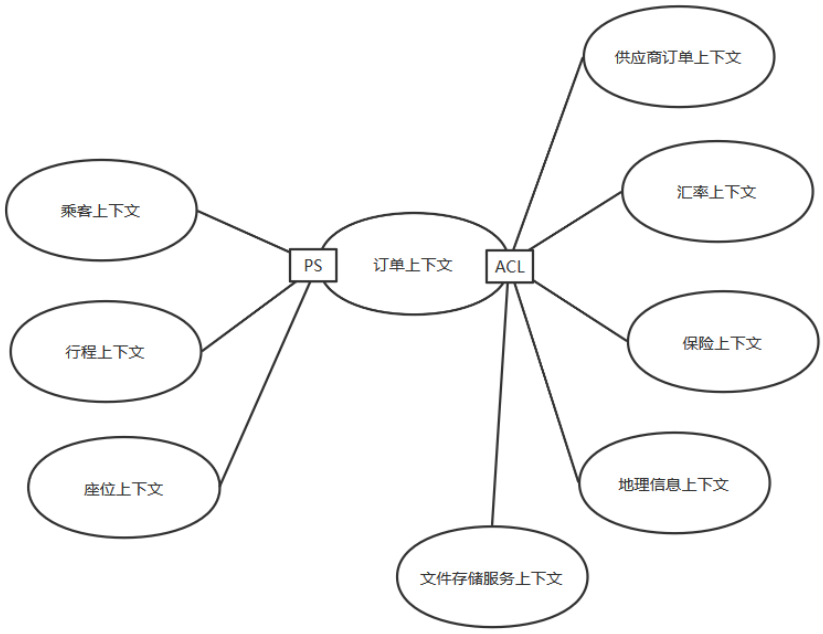
在通过软件实现一个业务系统时，建立一个领域模型是非常重要和必要的。因为领域模型是整个软件的核心，其是对某个边界的领域的一个抽象，反映了领域内用户业务需求的本质，开发者便可以仅关注领域边界内所需关注的部分。同时领域对象与技术实现无关，仅反映业务，领域模型贯穿软件分析、设计，以及开发的整个过程。领域专家、设计人员、开发人员通过领域模型进行交流，彼此共享知识与信息。因为大家面向的都是同一个模型，所以可以防止需求走样，可以让软件设计开发人员做出来的软件真正满足需求。

基于此，我们将预定系统划分为了对客订单和对供应商订单两个子域，对客订单负责处理客

户需要，对供应商订单负责记录供应商侧的相关数据用于财务统计。

限界上下文

划分限界上下文主要是想传达一种领域设计的思考方式，通过建模来划分清楚业务领域的边界，划分关系如下所示：



在上图左侧的 PS 代表合作关系（Partner Ship），右侧的 ACL 表示防腐层（Anticorruption Layer），即右侧几个上下文均是外部领域，需要通过防腐层来转换交互，以隔离业务。

3.1.2 战术设计

上文提到的失血模型，绝大多数来自于数据库的 Dao 对象，因为 Dao 对象仅仅是数据库结构的映射，没有包含业务逻辑，这样就会导致业务逻辑、校验逻辑散落在各个 service 层，不易维护。为了解决这个问题，DDD 将领域模型与数据模型做了区分，前者用于内聚自身行为，后者用于业务数据的持久化，仓储就是用来链接这两层的对象，数据模型又可以分为实体和值对象。

实体

实体（Entity）是指领域中可以由唯一标识进行区分的，且具有生命周期的对象，例如上文中的订单就是一个实体，其可以通过订单号进行唯一标识，且订单在整个预定系统中状态会发生改变。

值对象

值对象（Value Object）是指没有唯一标识的对象，也就是我们不需要关心对象是哪个，只需要关心对象是什么，例如上文中的行程上下文，故我们不能提供其 set 方法，行程如果需

要改变应该整个对象更新掉。

聚合根

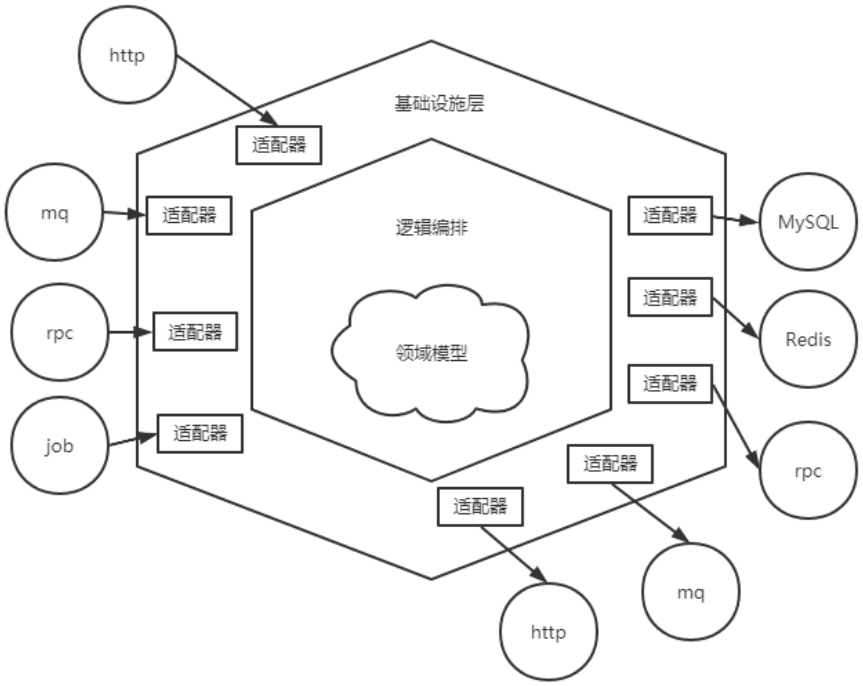
聚合（Aggregate）是指通过定义对象之间清晰的所属关系和边界来实现领域模型的内聚，并避免了错综复杂的难以维护的对象关系网的形成。聚合是一组相关对象的集合，每个聚合有一个根和边界，聚合根（Aggregate Root）是这个聚合的根节点，其必须是一个实体，边界定义了聚合内部有哪些实体或值对象。聚合内部的对象可以相互引用，对外通过聚合根进行交互。

仓储

仓储（repository）就是对领域的存储和访问进行统一管理的对象，聚合根被创建出来后进行持久化都需要跟数据库打交道，这样我们就需要一个类似数据库访问层的东西来管理领域对象。

3.2 架构设计

DDD 有多种分层架构模式，包括四层架构模式、五层架构模式、六层架构模式等，其核心均是定义一层领域层对领域对象及其关系进行建模，从传统的 MVC 三层架构中将领域抽出，但是依然是高层组件依赖低层组件，不同层次之间的耦合无法消除，故本文采用的是一种改进的分层架构模型：六边形架构，其结构如下所示：



六边形架构采用依赖倒置原则优化了传统的分层架构，低层组件应该依赖于高层组件提供的接口，即无论高层还是低层都依赖于抽象，这样使得整个架构变平。六边形中每条不同的边

代表了不同类型的端口，端口要么处理输入，要么处理输出，这样就将外界与系统内部进行了隔离，对于每种外部类型，都需要一个适配器与之对应。

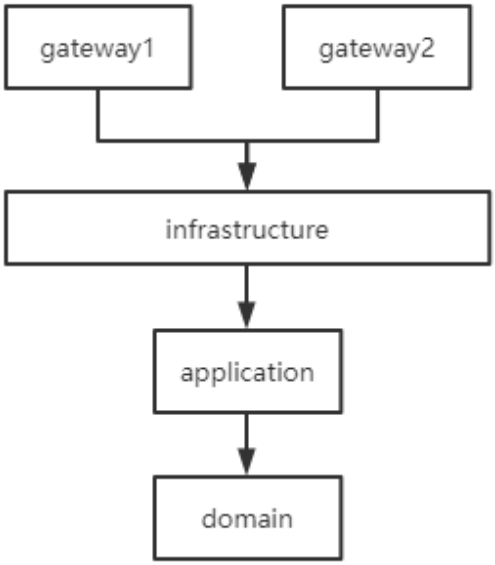
六边形架构的最大特点就是将技术与业务进行分离，六边形内部核心就是领域模型及不同领域的逻辑编排，领域模型外部的基础设施层就是为领域模型提供技术实现以及外部系统的适配，因为技术选型在项目之初就已经选定完成并且随着项目迭代也很少会发生更改，所以业务人员可以将更多的精力放在领域模型的更新上面。

如上文介绍的三方接口结构不稳定情况，也可以通过适配器转化为内部模型，防止修改成本过高。同时，对于外部请求，无论是通过 rpc，REST、HTTP 还是通过 MQ 等方式，均可通过适配器对输入进行转化，控制权由此交给内部区域进行处理。同时，上文战术设计中的仓储（repository）的实现也可以看作是持久化适配器，该适配器用于访问先前存储的聚合实例或者保存新的聚合实例，我们可以通过不同方式实现仓储适配器，如 MySQL、Redis 等。

四、DDD 实现

通过上文分析，本文以国际火车票中台预订系统项目作为 DDD 实践落地。

4.1 项目架构



根据 DDD 六边形架构原理，系统架构如上图所示，总共分为了 4 层：

- gateway：项目入口，其中包括 rpc、mq 等不同入口；
- infrastructure：基础设施层，一方面用作防腐，提供不同入口、出口的适配，另一方面实现领域层的接口提供技术实现；
- application：应用层，用于逻辑编排、管理、调度，突出核心逻辑，尽可能轻薄；
- domain：领域层，定义领域模型，对领域模型进行建模；

4.2 领域对象

前文提到 DDD 要解决的一个重要问题就是对象的失血问题，即对象不能仅作为数据的载体而没有行为，如上文代码中的参数校验应该是其自身的行为而非外部进行校验，通过适配器转换为内部对象就可以完成自身参数校验的行为，代码如下所示：

```
public class CreateOrderRequest extends CommonRequest {

    private List<SolutionOfferPair> outSolutionOfferPairList;
    private List<SolutionOfferPair> returnSolutionOfferPairList;
    private String transactionNo;
    ...
    private Contact contact;
    private List<Passenger> passengerInfoList;
    private boolean isSplitOrder;
    private boolean randomAssigned;
    private List<ExtraInfo> extraInfos;

    @Override
    public void requestCheck() {
        if (StringUtils.isEmpty(splitPlanId) &&
CollectionUtil.isEmpty(outSolutionOfferPairList)) {
            throw new BusinessException(ResponseCodeEnum.PARAM_ERROR);
        }
        ...
    }
}
```

4.3 战术设计实现

本文以订单聚合根为例具体说明战术设计的实现。

聚合根

聚合根中包含了实体和值对象，同时聚合根与仅有 getter、setter 的业务对象不同，其将业务逻辑也封装在内，提高了内聚性，同时将仓储封装在内，为聚合根提供持久化操作。

```
public class P2pOrder {

    private P2pOrderRepository repository;

    @Getter
    private long orderId;
    @Getter
```

```
    private OrderMasterModel orderMasterModel;
    @Getter
    private List<OrderItemModel> orderItemModels;

    public P2pOrder(P2pOrderRepository repository, long orderId) {
        this.repository = repository;
        this.orderId = orderId;
        orderInfoModel = new OrderInfoModel();
        orderItemModels = new ArrayList<>();
    }

    public boolean find() {
        return repository.find(this);
    }

    public void createOrder(CreateOrderRequest request) {
        if (find()) {
            throw new BusinessException(ResponseCodeEnum.ORDER_EXISTED);
        }
        this.orderMasterModel.createOrderMaster(request);
        repository.createP2pOrder(this);
        // 发送超时支付取消消息
        pushDelayMessage(this);
    }
}
```

实体

实体是指会存在状态变更的类，比如 order，其可以提供订单的变更状态等。

```
@Getter
public class OrderMasterModel {

    private OrderStatusEnum orderStatus;
    private LocalDateTime ticketTime;
    private LocalDateTime expirationTime;
    private String lang;
    ...

    public void init(CreateOrderRequest request) {
        this.channelName = request.getChannelMetaInfo().getChannel();
        this.orderStatus = OrderStatusEnum.SEAT_BOOKING;
        ...
    }
```

```
    public void ticketing() {
        if (this.orderStatus != OrderStatusEnum.WAIT_FOR_PAY) {
            throw new BusinessException(ResponseCodeEnum.ORDER_STATUS_ERROR);
        }
        this.orderStatus = OrderStatusEnum.TICKETING;
    }
}
```

值对象

而值对象是指仅作为描述没有唯一标识的类，比如行程信息，行程信息变更应该是整个行程信息进行变更而不是提供方法进行修改，故本文针对值对象的构造方法进行私有化处理，并仅提供静态方法用于重新创建对象。

```
@Getter
public class OrderSegmentModel {
    private long orderSegmentId;
    private short sequence;
    private TravelTypeEnum direction;
    private String   segmentType;
    private String departureLocationCode;
    private String departureLocationName;
    private String arriveLocationCode;
    private String arriveLocationName;
    ...

    private OrderSegmentModel() {}

    public static OrderSegmentModel init(OrderSegment orderSegment, short sequence) {
        OrderSegmentModel model = new OrderSegmentModel();
        model.orderSegmentId = Long.valueOf(orderFareId + "0" +
orderSegment.getSegmentId());

        model.sequence = sequence;
        if (Objects.nonNull(orderSegment.getDepartureLocation())) {
            model.departureLocationCode =
orderSegment.getDepartureLocation().getLocationCode();
            model.departureLocationName =
ConvertUtil.getLocationName(orderSegment.getDepartureLocation());
        }
        if (Objects.nonNull(orderSegment.getArrivalLocation())) {
            model.arriveLocationCode =
orderSegment.getArrivalLocation().getLocationCode();
```

```
                model.arriveLocationName                                    =
ConvertUtil.getLocationName(orderSegment.getArrivalLocation());
            }

            model.departureTime                                             =
DateUtil.parseStringToDateTime(orderSegment.getDepartureDateTime(),
DateUtil.YYYY_MM_DDHHmm);
            model.arriveTime                                                =
DateUtil.parseStringToDateTime(orderSegment.getArrivalDateTime(),
DateUtil.YYYY_MM_DDHHmm);
            ...
            return model;
        }
```

仓储

仓储封装于聚合根内部，不用于外部调用，故通过工厂方法将仓储注入聚合根中。

```
@Slf4j
@Component
public class OrderFactory {

    @Autowired
    private OrderIdGenerator orderIdGenerator;
    @Autowired
    private P2pOrderRepository repository;

    public P2pOrder create(CreateOrderRequest request) {
        long orderId = orderIdGenerator.generateOrderId();
        if (orderId < 1) {
            log.error("fail to gen order id");
            throw new BusinessException(ResponseCodeEnum.FAIL_GEN_ORDER_ID);
        }
        return new P2pOrder(repository, orderId);
    }
}
```

仓储用于链接领域层与数据层，使领域对象与 DAO 隔离，使我们软件更加健壮。

```
@Slf4j
@Component
public class P2pOrderRepositoryImpl implements P2pOrderRepository {

    @Autowired
```

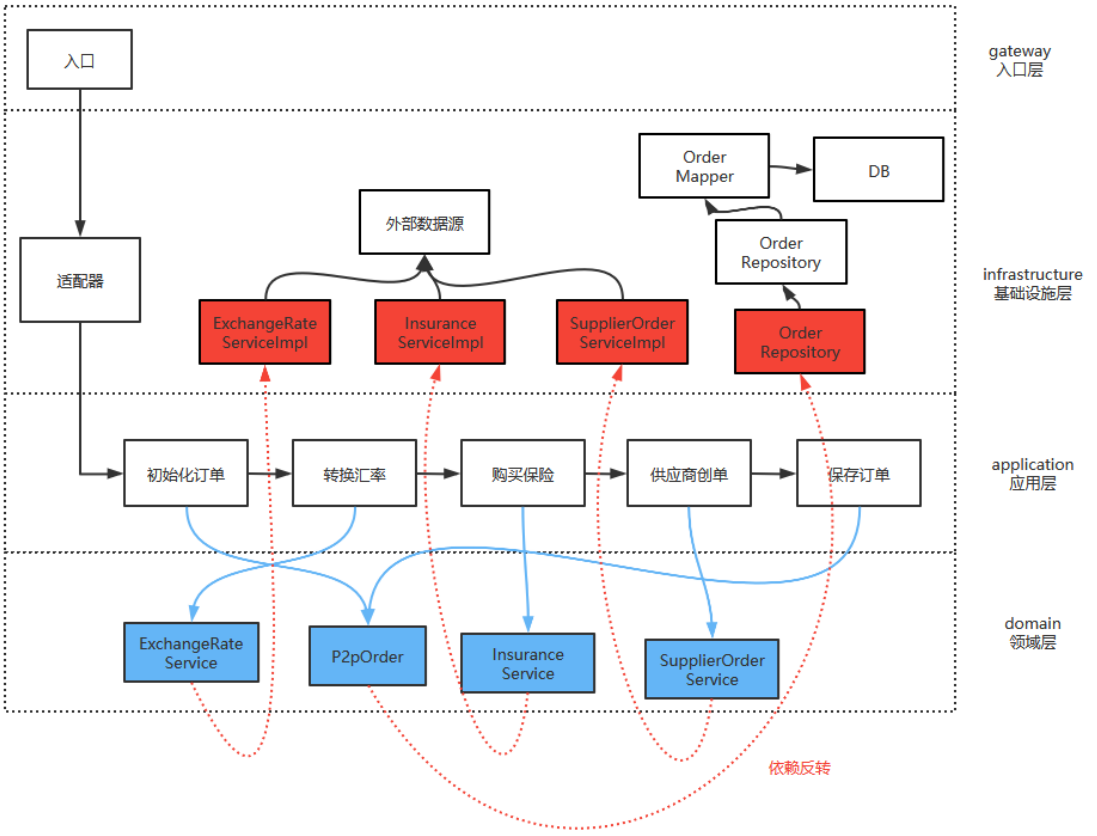
```
    private OrderMapper orderMapper;

    @Override
    public boolean createP2pOrder(P2pOrder p2pOrder) {
        OrderMasterEntity orderMasterEntity = new OrderMasterEntity;
        orderMasterEntity.setOrderId(p2pOrder.getOrderId());

orderMasterEntity.setOrderStatus(  p2pOrder.getOrderMasterModel().getOrderStatus().getC
ode());
        ...
        return orderMapper.insert(orderMasterEntity) > 0;
    }
}
```

防腐层

防腐层，又称为适配层，在对外部上下文的访问中，就需要引入防腐层对外部上下文进行一次转义，这样就可以将外部上下文转化为内部模型，防止因为外部更改导致改动影响过大。仓储也是防腐层的一种，因为其隔离了数据库的 DAO 对象，转化为了内部的实体和值对象。在本系统中，也需要对外部的汇率服务、保险服务等引入防腐层的概念。

4.4 服务结构

通过 DDD 思想进行建模，并采用 DDD 的六边形架构，重构后的服务结构如下：



五、总结

本文基于携程国际火车票出票系统对领域驱动设计进行实践，通过对出票系统中多个领域的划分使业务逻辑更加清晰，使得代码易于维护和迭代；并通过领域驱动设计的六边形架构将业务与技术进行了隔离，突出业务重点，使代码易于阅读；加入防腐层使外部上下文与内部模型进行隔离，防止外部对象侵蚀；将迭代需求转化为各个领域模型的更新，以领域来驱动后续功能开发，使其变得可控，避免了软件架构设计模式变成“大泥球”模式。

鉴于作者经验有限，对领域驱动的理解难免会有不足之处，欢迎大家共同探讨，共同提高。

参考文献

[1] Scott Millett 等著，蒲成 译；领域驱动设计模式、原理与实践(Patterns, Principles, and Practices of Domain-Driven Design)；清华大学出版社, 2016

[2] Evic Evans 著，赵俐 等译；领域驱动设计:软件核心复杂性应对之道；人民邮电出版社, 2010

[3] 领域驱动设计在互联网业务开发中的实践

[4] 阿里技术专家详解 DDD 系列 第二讲 - 应用架构

[5] 基于 DDD 思想的酒店报价重构实践

[6] DDD（领域驱动设计）总结

[7] 谈谈 MVC 模式

[8] 阿里技术专家详解 DDD 系列 第三讲 - Repository 模式

[9] 领域驱动设计详解：是什么、为什么、怎么做？

[10] 领域建模在有赞客户领域的实践

[11] DDD 分层架构的三种模式

Reactive 模式在 Trip.com 消息推送平台上的实践

【作者简介】KevinTen，携程后端开发工程师，关注 Reactive 和 RPC 领域，深度参与开源社区，对 Reactive 技术有浓厚兴趣；Pin，携程技术专家，Apache Dubbo 贡献者，关注 RPC、Service Mesh 和云原生领域。

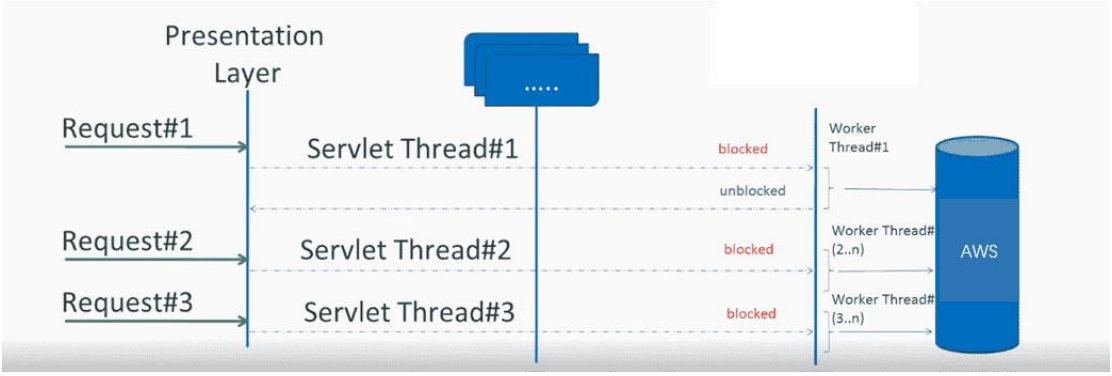
一、背景

1.1 业务需求

Trip.com 消息推送平台主要负责 Trip.com 在海外的邮件等渠道的营销消息推送，系统整体设计为面向上游消息的流式架构，当接收到上游的请求之后，经过一系列的计算逻辑，最后将会调用下游第三方发送接口，将邮件等消息通过网络发送出去。Trip.com 消息推送平台是典型的 IO 密集型应用。

1.2 当前解决方案

Trip.com 的业务层以 Java 技术栈为主，其中主要 web 服务基于同步+阻塞 IO 的 servlet 模型，也有少部分 web 服务基于异步 servlet。servlet 是经典的 JavaEE 解决方案，旨在用多线程模型解决 IO 高并发问题。这种同步编程模型的优点是开发简单，易于进行问题追踪，并且对开发人员的要求比较低，有利于业务的快速开发。



Tirp.com 消息推送平台也是基于同步+阻塞 IO 的 servlet 模型架构。当客户端发起网络请求的时候，请求首先会由 Tomcat 容器的 Acceptor 线程进行处理，将 Channel 放到待处理请求队列然后通过 Poller 线程进行 IO 多路复用的监听，当 Poller 监听到 Channel 的可读事件后，请求体将会从缓冲区被读入内存，然后交由 Tomcat 容器的 Worker 线程池进行消费。由于需要使用阻塞 IO 调用下游的第三方发送接口，所以 Worker 线程池需要启动大量的线程进行并发操作，根据 Tomcat 配置文件，最多可能启动 1024 个 worker 线程。

```
<!-- 最大线程数 1024，在极端情况下，最高可能启动 1024 个线程 -->
<Connector port="${port.http.server}"
protocol="org.apache.coyote.http11.Http11NioProtocol" minSpareThreads="20"
maxThreads="1024"/>
```

代码示例

Trip.com 消息推送平台使用 AWS 的 SES 服务进行邮件发送，在发送 Email 时将会调用 AWS 的同步 SDK:

```
SendEmailResult sendEmail(SendEmailRequest sendEmailRequest);
```

而 AWS 的同步 SDK 使用的是 apache 的 HttpClient，底层采用的 BIO 模式将会阻塞当前线程，直到会话缓冲区有数据到达或到达超时时间。

二、存在的问题

而随着业务量上涨带来上游消息负载增加，原有的阻塞 IO 模型在高并发下，会有大量线程处于阻塞状态，导致应用需要囤积大量的线程以应对峰值压力。过多的线程将会造成大量的内存占用和频繁线程上下文切换的开销，所以原有的 servlet 线程模型具有 CPU 利用率低、内存占用大、对异常请求不具备弹性等缺点。该平台在压力峰值时需要部署大量机器，它主要具有以下性能上的问题:

2.1 线程上下文切换开销

一次请求的 IO 时间平均在 1200ms，最高能达到 50000ms，而计算时间只有 1~2ms，根据最佳线程公式理论上 1C 需要 600~2500 个线程。囤积如此多的线程将会造成大量的上下文切换开销和上 GB 的内存占用。但若是使用少量的线程，将可能由于线程数量的限制，导致请求量过高时拿不到处理线程，最终请求超时，不具备低延迟等特性。

2.2 部署成本高

若是采用主流的 2C4G 容器配置，理论上将需要 1000+的线程用于处理请求，这将占用大概 1GB 的内存空间。同时若并发请求数>线程数时，需要采用水平扩容增加服务的吞吐量，所以服务需要按照峰值并发进行预估部署，造成空闲时间大量资源的浪费。

2.3 超时风险

一次 IO 最高能达到 50s，当有异常请求导致响应时间突增时，因为会阻塞线程，导致线程池中的线程大部分都被阻塞，从而无法响应新的请求。在这种情况下，少量的异常请求将会导致上游大量的超时报错，因此服务不具有弹性。

三、解决方案

面对传统 BIO 模式在 IO 密集型场景下的缺点，借助 NIO 和多路复用技术可解决上述 BIO 问题，目前 Java 项目对接 NIO 的方式主要依靠回调，代码复杂度高，降低了代码可读性与可维护性。随着 Reactive 反应式架构的流行，业界有一些公司开始推动服务的全异步升级，开始采用 Reactive 架构来解决此类问题。而 Trip.com 也开始逐渐重视服务网络 IO 的性能问题，已有部分团队开始进行 Reactive 实践。

Reactive 构建的程序代表的是异步非阻塞、函数式编程、事件驱动的思想。随着近年来 Reactive 编程模式的发展，能达到高性能与可读性的兼顾。Trip.com 消息推送平台利用 Reactive 相关技术对系统进行异步非阻塞 IO 改造，主要希望达到以下两个目标:

1）提升单机的吞吐量，提高有效 CPU 使用率、降低内存占用、保证业务请求突增时系统的可伸缩性，最终降低硬件成本。

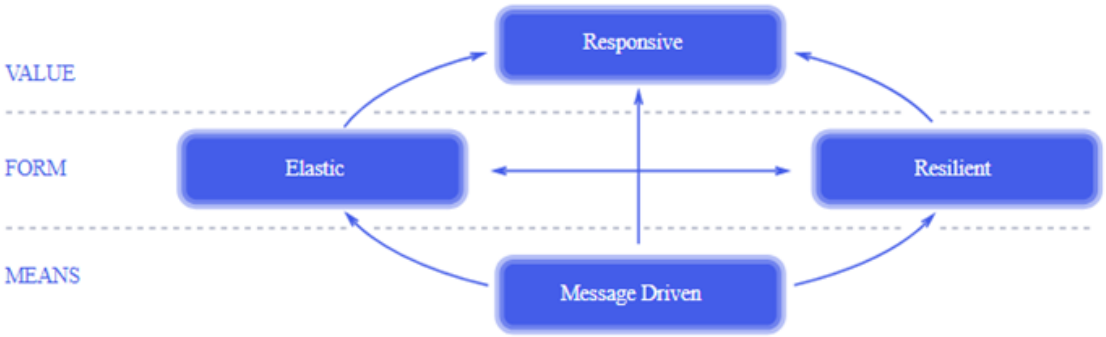
2）使用 Reactive 编程模型，替代处理 NIO 常用的异步回调模式，积累对同步阻塞应用进行异步非阻塞化升级的重构经验。

3.1 什么是 Reactive?

反应式宣言：来自不同领域的组织正在不约而同地发现一些看起来如出一辙的软件构建模式。它们的系统更加稳健，更加有可回复性，更加灵活，并且以更好的定位来满足现代的需求。这些变化之所以会发生，是因为近几年的应用需求出现了戏剧性的变化。仅仅在几年之前，大型应用意味着数十台服务器，数秒的响应时间，数小时的离线维护时间以及若干 GB 的数据。而在今天，应用被部署在一切场合，从移动设备到基于云的集群，这些集群运行在数以千计的多核心处理器的之上。用户期望毫秒级的响应时间以及 100%的正常运行时间。数据则以 PB 为单位来衡量。

昨天的软件架构已经完全无法地满足今天的需求。我们相信，一种条理分明的系统架构方法是必要的，而且我们相信关于这种方法的所有必要方面已经逐一地被人们认识到：我们需要的系统是反应式的，具有可回复性的，可伸缩的，以及以消息驱动的。我们将这样的系统称之为反应式系统。以反应式系统方式构建的系统更加灵活，松耦合和可扩展。这使得它们更容易被开发，而且经得起变化的考验。它们对于系统失败表现出显著的包容性，并且当失败真的发生时，它们能用优雅的方式去应对，而不是放任灾难的发生。反应式系统是高度灵敏的，能够给用户以有效的交互式的反馈。

Reactive 宣言



2013 年 6 月，Roland Kuhn 等人发布了《反应式宣言》，该宣言定义了反应式系统应该具备的一些架构设计原则。符合反应式设计原则的系统称为反应式系统。根据反应式宣言，反应式系统需要具备即时响应性（Responsive）、回弹性（Resilient）、弹性（Elastic）和消息驱动

（Message Driven）四个特质。

VALUE-即时响应性（Responsive)

只要有可能，系统就会及时地做出响应。即时响应是可用性和实用性的基石，而更加重要的是，即时响应意味着可以快速地检测到问题并且有效地对其进行处理。即时响应的系统专注于提供快速而一致的响应时间，确立可靠的反馈上限，以提供一致的服务质量。这种一致的行为转而将简化错误处理、建立最终用户的信任并促使用户与系统做进一步的互动。

反应式系统具备及时响应性，可以提供快速的响应时间，在错误发生时也会保持响应性。

FORM-回弹性（Resilient）

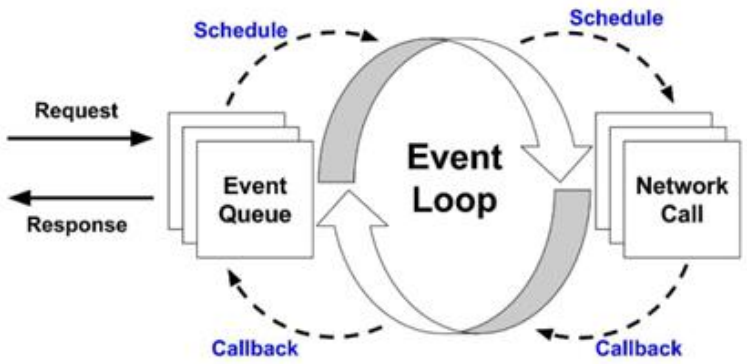
系统在出现失败时依然能保持即时响应性，每个组件的恢复都被委托给了另一个外部的组件，此外，在必要时可以通过复制来保证高可用性。因此组件的客户端不再承担组件失败的处理。

反应式系统通过背压等特性避免错误在系统中的传播，所以在失败发生的时候，反应式系统将会对错误具有更强的承受处理能力。背压是 reactive stream 定义的规范，可以使用 rsocket、grpc 这类网络协议实现背压的机制。

FORM-弹性（Elastic）

系统在不断变化的工作负载之下依然保持即时响应性。反应式系统可以对输入负载的速率变化做出反应，比如通过横向地伸缩底层计算资源。这意味着设计上不能有中央瓶颈，使得各个组件可以进行分片或者复制，并在它们之间进行负载均衡。

反应式系统的瓶颈不在于线程模型，在不同的工作负载下，使用 EventLoop 线程模型将始终提供 CPU 资源允许的计算能力，当达到计算能力瓶颈时可以横向拓展 CPU 计算资源。反应式系统通过 EventLoop+NIO 模型，避免线程的上下文开销，同时也避免线程池资源的大小成为系统的瓶颈。



3.2 使用 Reactive 技术进行重构

3.1 章节我们谈论了 Reactive 理论模型，以及它的部分技术原理。现在，我们要使用 Reactive 技术重构 Trip.com 消息发送平台。根据 reactive 思想的指导，对于 IO 密集型应用，我们可以采用 EventLoop+NIO 的方式对传统的同步阻塞 IO 模型进行优化。

在整个系统中，首先介绍三个主要的中间件：

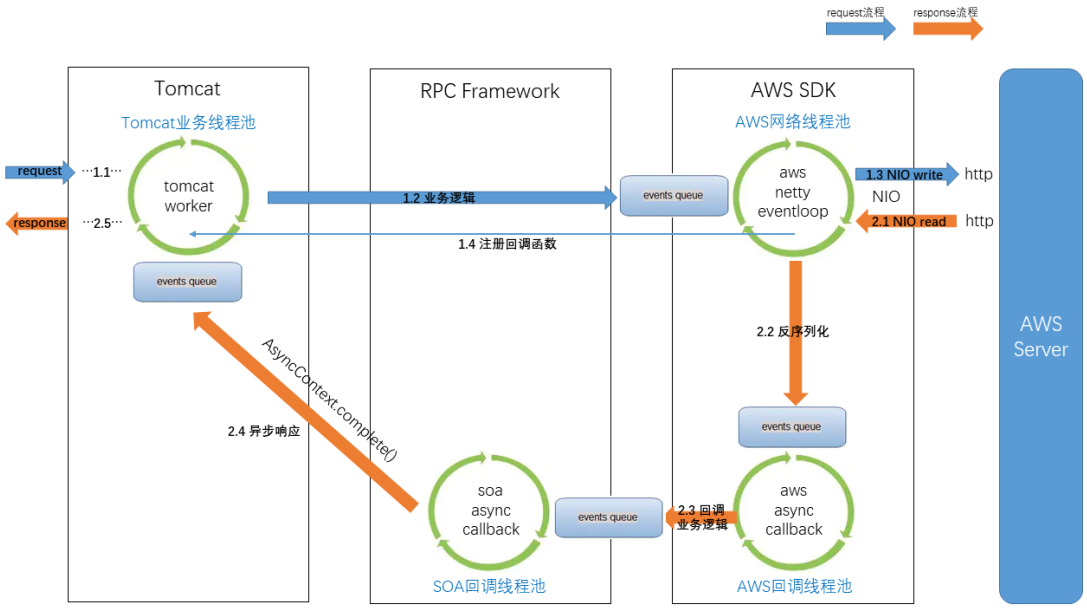
1）Tomcat：网络中间件，负责接收和响应网络请求
2）RPC Framework（soa）：Trip.com 集团的 RPC 框架，提供了同步和异步两种服务模式
3）AWS SDK：使用 AWS 的异步 SDK，通过 NIO 调用 AWS 服务

3.2.1 线程模型设计

在原同步版本中，首先使用 Tomcat 的 Worker 线程接收和处理 request 并执行同步逻辑，而后通过 AWS 的同步 SDK 进行 BIO 调用，此时 worker 线程将会 block 在 IO 调用上。当网络 IO 响应时，该 worker 线程将被唤醒，拿到 response 并执行响应逻辑。同步阻塞的线程模型是比较简单的，worker 线程基本负责了整个流程。

而采用 EventLoop+NIO 的异步非阻塞模式，将会无可避免的引入回调函数，为了回调流程的逻辑清晰和故障隔离等功能考虑，将会引入几组不同的回调线程池，来负责不同模块的回调逻辑。

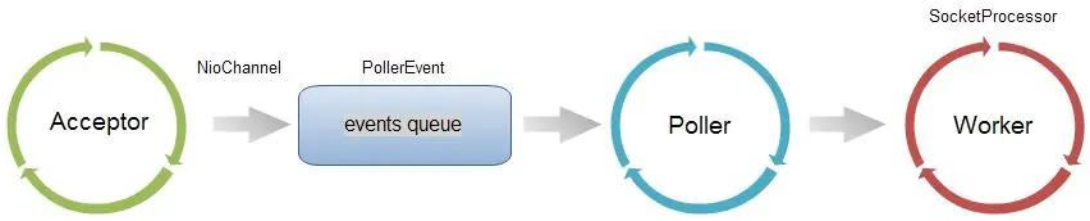
整个异步流程的线程模型设计如图所示：



request 流程：

1）使用 Tomcat 接收和处理网络请求

使用 Tomcat 的 Acceptor 线程接收 socket 连接事件，然后封装成 NioChannel 压入 events queue 中。然后 Poller 线程通过 Selector 获取到可读的 socket，并传递给 Worker 线程进行处理。该部分与原版本的同步模型相同，Tomcat 线程模型如下图所示：



2）业务逻辑处理部分

Tomcat 的 Worker 线程将负责执行同步逻辑，worker 线程将会依次同步执行 Tomcat 逻辑、RPC Framework 逻辑、业务逻辑、AWS SDK 逻辑。

Worker 线程执行完同步逻辑之后，将会把封装好的 request 放入 EventLoop 的 events queue 中，等待 EventLoop 的处理。

3）AWS SDK NIO 异步处理

AWS 的异步 SDK，使用 Netty 进行网络 IO 的传输，其内部会内置一个 Netty 的 EventLoop 线程池，负责网络 IO 的序列化反序列化。AWS 的 EventLoop 线程池定义如下，使用的是 Netty 的 NioEventLoopGroup：

```
int numThreads = Optional.ofNullable(builder.numberOfThreads).orElse(0);
ThreadFactory threadFactory = Optional.ofNullable(builder.threadFactory)
                                      .orElse(new ThreadFactoryBuilder()
                                      .threadNamePrefix("aws-java-sdk-
NettyEventLoop")
                                      .build());
return new NioEventLoopGroup(numThreads, threadFactory);
```

4）注册回调函数

① channelFuture 注册回调

Netty 使用 NIO 进行网络传输，并将对应回调函数注册到对应的 channelFuture 上。

② AWS SDK 注册回调

将 channelFuture 对应的 Promise 转换成 CompletableFuture，AWS SDK 通过

CompletableFuture.whenCompleteAsync 方法将回调函数提交给 futureCompletionExecutor 线程池。

```
responseHandlerFuture.whenCompleteAsync((r, t) -> {
    if (t == null) {
        responseFuture.complete(r);
    } else {
        responseFuture.completeExceptionally(t);
    }
}, futureCompletionExecutor);
```

futureCompletionExecutor 线程池的设置如下，为上图中的 AWS SDK 内置的回调线程池。

```
int processors = Runtime.getRuntime().availableProcessors();
int corePoolSize = Math.max(8, processors);
int maxPoolSize = Math.max(64, processors * 2);
ThreadPoolExecutor executor = new ThreadPoolExecutor(corePoolSize, maxPoolSize,
                                                     10, TimeUnit.SECONDS,
                                                     new
LinkedBlockingQueue<>(1_000),
                                                     new ThreadFactoryBuilder()
                                                         .threadNamePrefix("sdk-
async-response")
                                                         .build());
```

③ 业务逻辑注册回调

AWS 使用 futureCompletionExecutor 线程池执行回调逻辑，业务逻辑使用 Reactor 的 Mono 异步编程模型（3.2.3 章节介绍），所以需要将 AWS 的 CompletableFuture 响应转换为 Mono：

```
Mono<SendEmailResponse> responseMono = Mono.fromCompletionStage(() -> {
    CompletableFuture<SendEmailResponse>          responseFuture          =
sesAsyncClient.sendEmail(sendEmailRequest);
    return responseFuture;
});
```

在业务逻辑代码中，使用 Mono 进行 Reactive 风格的异步编程。最后，由于 Trip.com 的 RPC Framework 在异步编程模型中仅支持 ListenableFuture，所以我们需要将业务代码中的 MonoFuture 类型转换为 ListenableFuture 类型，并返回给 RPC Framework，在这里我们使用 Mono.subscribe()方法：

```
SettableFuture listenableFuture = SettableFuture.create();
responseMono.subscribe(
        response -> listenableFuture.set(response),
```

```
        throwable -> listenableFuture.setException(throwable));
return listenableFuture;
```

④ RPC Framework 注册回调

当 RPC Framework 接收到一个异步调用结果 ListenableFuture 后，将会通过 addListener()方法注册 RPC Framework 层级的回调函数：

```
responseFuture.addListener(() -> {...}, rpcExecutorService);
```

在这里 RPC Framework 使用了自己定义的回调线程池 rpcExecutorService，即上图中的 SOA 回调线程池：

```
ThreadPoolConfig config = threadPoolConfigProvider.getThreadPoolConfig(key);
rpcExecutorService = new RpcExecutorService(config.taskWrapper(), new
ThreadPoolExecutor(
        config.corePoolSize(),
        config.maximumPoolSize(),
        config.keepAliveTimeInMills(),
        TimeUnit.MILLISECONDS,
        config.workQueue(),
        config.threadFactory(),
        config.rejectedExecutionHandler()
));
```

至此，异步回调的链路组装完成。等待 NIO 收到响应的时候，将会依次触发上面的回调函数，进行响应流程的处理。

response 流程：

1）AWS SDK Netty 响应

当 netty 收到 IO 响应数据之后，对应的 EventLoop 线程将会处理可读事件并执行回调函数。EventLoop 首先会读取缓冲区中的数据并进行反序列化，而后执行 channel 的 pipeline，将反序列化后的 response 传递给下一流程。

2）AWS SDK 异步回调

AWS SDK 使用 1.4 中提到的 AWS 回调线程池，进行回调逻辑的处理。AWS SDK 的回调函数主要负责 AWS 内置的 response 处理，例如 AWS 的监控、埋点、日志等。

3）业务逻辑的异步回调

当 AWS 的异步回调流程完成之后，回调线程将会进入我们的业务代码注册的回调函数中，

此时线程是 1.4 中定义的 sdk-async-response 线程。在业务逻辑的回调响应中，我们可以定义自己的业务回调线程池进行处理，也可以直接使用 AWS 的回调线程进行处理。由于操作非常轻量，所以在这里我们没有再额外定义一个业务回调线程池，而是直接使用了 1.4 中的线程池，减少了一次线程切换的开销。

4）RPC Framework 的异步回调

如 1.4 所述，当业务回调逻辑全部执行完毕之后，将会触发 ListenableFuture 的回调流程，此时进入 RPC Framework 这一层的回调逻辑处理。首先由 aws-async-callback 线程继续进行同步处理，而后将会把(ListenableFuture)responseFuture 中的回调函数提交给 rpcExecutorService 线程池处理。在 RPC Framework 的回调函数中，将会执行 RPC 的监控、埋点等功能（可参考 dubbo），最终将会把异步响应传递给 Tomcat。

Servlet3.0 提供了 AsyncContext 用来支持异步处理请求。RPC Framework 在异步请求处理开始的时候，将会通过 servletRequest.startAsync()获取对应的 AsyncContext 对象，此时既不关闭响应流也不进行响应的返回。当 RPC Framework 执行完所有的异步回调逻辑之后，此时 rpcExecutorService 线程将会调用 asyncContext.complete()将上下文传递给 Tomcat 容器：

```
finally {
    try {
        ……
        asyncContext.complete();
    } catch (Throwable e) {
        ……
    }
}
```

5）Tomcat 的异步响应

asyncContext.complete() 使 Tomcat 容器接收到 ASYNC_COMPLETE 事件，在 NioEndpoint.processSocket()方法中，将会通过 Executor executor = getExecutor(); 操作获取到 Worker 线程池（注①），而后 rpcExecutorService 线程把响应操作写入到 Worker 线程池的 events queue 中，之后 worker 线程将响应流写回客户端（注②）。

```
public boolean processSocket(SocketWrapperBase<S> socketWrapper,
        SocketEvent event, boolean dispatch) {
    try {
        if (socketWrapper == null) {
            return false;
        }
        SocketProcessorBase<S> sc = null;
        if (processorCache != null) {
            sc = processorCache.pop();
        }
```

```
            if (sc == null) {
                sc = createSocketProcessor(socketWrapper, event);
            } else {
                sc.reset(socketWrapper, event);
            }
            // 注①
            Executor executor = getExecutor();
            if (dispatch && executor != null) {
                // 注②
                executor.execute(sc);
            } else {
                sc.run();
            }
        } catch (RejectedExecutionException ree) {
            getLog().warn(sm.getString("endpoint.executor.fail", socketWrapper) , ree);
            return false;
        } catch (Throwable t) {
            ExceptionUtils.handleThrowable(t);
            // This means we got an OOM or similar creating a thread, or that
            // the pool and its queue are full
            getLog().error(sm.getString("endpoint.process.fail"), t);
            return false;
        }
        return true;
    }
```

至此，响应流写回客户端，整个请求-响应过程完成。

3.2.2 异步线程模型总结

如 3.2.1 所述，为了实现异步非阻塞的流程，不仅需要 Tomcat 的 Worker 线程池，还需要引入两个回调线程池和一个 Netty 的 EventLoop 线程池。

其中一个是 AWS 异步 SDK 的回调线程池，主要负责 AWS 功能的处理，使用的异步编程模型是 CompletableFuture；另外一个是 RPC Framework 的回调线程池，主要是封装了 Servlet3.0 的 AsyncContext 并提供异步服务的能力，使用的异步编程模型是 ListenableFuture。

而我们的业务代码使用了 Reactor 的 Mono 异步编程模型，所以需要涉及不同 Future 类型的转换，通过 Reactor 丰富的操作符，我们可以很容易的做到这一点。

预期达到的效果

使用 NIO 方式发起 AWS 的调用，避免线程阻塞，从而最大限度的消除上述 BIO 缺点，提高系统性能。最终使得应用符合 Reactive 架构理念，从而具备弹性、容错性，以降低部署成本，提高资源利用率。

3.2.3 NIO 异步编程模型选择

NIO 消除了线程的同步阻塞，意味着只能异步处理 IO 的结果，这与业务开发者顺序化的思维模式有一定差异。当业务逻辑复杂以及出现多次远程调用的情况下，多级回调难以实现和维护。

AWS 原生异步 SDK 的调用模式如下，使用了 Java8 的组合式异步编程 CompletableFuture：

```
default CompletableFuture<SendEmailResponse> sendEmail(SendEmailRequest sendEmailRequest)
```

设计 NIO 编码，业界主流的编码方式主要有以上几种，通过 CompletableFuture 和 Lambda 表达式，可以快速实现轻量业务异步封装与编排，与 Callback 相比可以避免方法多层嵌套问题，但面对相对复杂业务逻辑时仍存在以下局限：

- 难以简单优雅实现多异步任务编排；
- 难以处理实时流式场景；
- 难以支持高级异常处理；
- 不支持任务延迟执行。

使用 Reactive 模型能够解决上述 Future 的局限。例如，使用 Reactor 封装 AWS 的异步调用：

```
// sending with async non-blocking io
return Mono
        .fromCompletionStage(() -> {
            // 注①
            CompletableFuture<SendEmailResponse> responseFuture = awsAsyncClient.sendEmail(sendEmailRequest);
            return responseFuture;
        })
        // 注② thread switching: callback executor
```

```
        // .publishOn(asyncResponseScheduler)
        // 注③ callback: success
        .map(response -> this.onSuccess(context, response))
        // 注④ callback: failure
        .onErrorResume(throwable -> {
            return this.onFailure(context, throwable);
        });
```

①调用 AWS 的异步 SDK，将返回的 CompletableFuture 转成的 Mono。
②如 2.3 所述，可以使用 Mono.publishOn()将业务逻辑的回调函数放入自定义的线程池执行，也可以继续使用 AWS 的回调线程继续执行，在这里没有使用自定义的线程池。
③如果执行成功，则执行 map()中的回调方法
④如果执行抛出异常，则执行 onErrorResume()中的回调方法

从上面简单对比可以看出，相比 Future，基于 Reactive 模型丰富的操作符组合（filter/map/flatMap/zip/onErrorResume 等高阶函数）代码清晰易读，搭配 Lamda 可以轻松实现复杂业务场景任务编排。

Reactor 异步原理

reactor-core 是一层编程框架，它提供的是 reactive 风格的编程模式，以及异步调用的编排能力。而本身并没有真正网络 IO 异步回调的功能，真正的异步回调功能是底层网络 IO 框架的 Future 提供，比如上面 AWS 返回的 CompletableFuture 才是真实绑定到网络 IO 上的 Future，而 Reactor 仅仅是将其包装，方便进行 reactive 编程。

从 fromCompletionStage 方法中可以找到，这里将实际的 CompletionStage 包装成了 MonoCompletionStage（注①），但在实际订阅的时候，其实是将 Mono 的回调函数放入了 future.whenComplete 中（注②），所以说 Mono 在这里是 CompletableFuture 的外层包装。

```
final class MonoCompletionStage<T> extends Mono<T>
        implements Fuseable, Scannable {

    static final Logger LOGGER = Loggers.getLogger(MonoCompletionStage.class);

    // 注①
    final CompletionStage<? extends T> future;

    MonoCompletionStage(CompletionStage<? extends T> future) {
        this.future = Objects.requireNonNull(future, "future");
    }

    @Override
    public void subscribe(CoreSubscriber<? super T> actual) {
        Operators.MonoSubscriber<T, T>
```

```
                sds = new Operators.MonoSubscriber<>(actual);

        actual.onSubscribe(sds);

        if (sds.isCancelled()) {
            return;
        }

        // 注②
        future.whenComplete((v, e) -> {
            if (sds.isCancelled()) {
                //nobody is interested in the Mono anymore, don't risk dropping errors
                Context ctx = sds.currentContext();
                if (e == null || e instanceof CancellationException) {
                    //we discard any potential value and ignore Future cancellations
                    Operators.onDiscard(v, ctx);
                }
                else {
                    //we make sure we keep _some_ track of a Future failure AFTER the
Mono cancellation
                    Operators.onErrorDropped(e, ctx);
                    //and we discard any potential value just in case both e and v are not
null
                    Operators.onDiscard(v, ctx);
                }

                return;
            }
            try {
                if (e instanceof CompletionException) {
                    actual.onError(e.getCause());
                }
                else if (e != null) {
                    actual.onError(e);
                }
                else if (v != null) {
                    sds.complete(v);
                }
                else {
                    actual.onComplete();
                }
            }
            catch (Throwable e1) {
                Operators.onErrorDropped(e1, actual.currentContext());
```

```
                    throw Exceptions.bubble(e1);
                }
            });
    }

    @Override
    public Object scanUnsafe(Attr key) {
        return null; //no particular key to be represented, still useful in hooks
    }
}
```

使用 Reactor 还有另外一个好处，那就是统一异步编程模型。比如有的异步编程框架提供 ListenableFuture，有的是 CompletableFuture，还有 gRPC、dubbo、webflux 等中间件框架，都提供了自己的异步编程模型实现。如果直接针对各个框架自己的原生实现进行异步编程，将会存在不同风格的代码。而 Reactor 是反应式库的当前标准，使用 Reactor 库可以封装不同异步编程框架的异构实现，使用统一的 API 执行异步编程。

四、压测对比

通过一系列的 Reactive 技术改造，我们现在已经拥有了一个基于 EventLoop+NIO 的 IO 密集型应用，那么它的性能是否如同我们的理论推导一样将会得到提升呢？接下来我们将会通过一系列的性能压测，得到最终的结论。

压测目标：

1）是否能够达到稳定状态，以及达到稳定状态后，系统表现和指标；
2）对两个应用在不同压力下的指标，进行全面的对比，得出压测结论；

以下数据均为"稳态"时数据，稳态定义如下：

指标	稳态数值	说明
CPU	60%	
RT	无明显上升	采用P95线
GC	无FullGC，无明显上升	

压测结果

当原应用和新应用都达到上述定义的稳态条件时，我们得到了一组对比数据。通过与原应用的压力测试结果对比，我们发现使用 EventLoop+NIO 的新应用，在相同硬件资源下，QPS 能够提升 2~3 倍，RT 缩短近 50%，同时在内存占用上也得到了一定的优化。证明该应用在

经过 Reactive 技术改造后，性能较之前同步阻塞的 Servlet 架构得到了明显提升。

五、总结和展望

在本文中我们首先介绍了 Trip.com 消息推送平台服务的现状，以及现有同步+阻塞模式在 IO 密集型场景下的缺点。接下来我们通过分析如何解决这些缺点，引入了业界流行的 reactive 反应式架构。接下来在 reactive 宣言的弹性和伸缩性两种手段中，总结出了 EventLoop、NIO、背压等技术手段，最后通过这些具体的技术手段来实现我们应用的升级重构。最终根据压测结果，可以看到服务性能较之前 servlet 架构得到了明显提升。

随着云原生浪潮的到来以及物联网、边缘计算的推进，未来应用间网络通讯的量级将会变得越来越庞大，网络 IO 会是系统架构中的一等公民。如何使我们的应用能够具有更高的性能和更健壮的特性，以及如何降低硬件资源的成本，这些挑战将促使应用开发人员不断的学习实践类似 reactive 相关的技术。而在学习实践的过程中，对经典的 servlet 架构的优化重构一定是具有代表性意义的。

在适合的业务场景下，反应式技术架构能够有效提升服务吞吐能力，降低业务编排复杂度，帮助构建云原生时代整体系统快速即时反应能力。但同时构建 Reactive 模式的程序也为开发者带来更高的要求，面临比同步更为复杂的编程模型，需要更好的处理好阻塞和写出更优秀的异步代码。希望与对反应式技术感兴趣的同学和团队多多交流。

【参考文档】

[1]高德云图异步反应式技术架构探索和实践

[2] Reactive 架构才是未来

[3]全面异步化：淘宝反应式架构升级探索

[4]淘宝应用架构升级——反应式架构的探索与实践

[5]高性能 Java 应用层网关设计实践

1 分钟售票 8 万张！门票抢票背后的技术思考

【作者简介】HongLiang，携程高级技术专家，专注系统性能、稳定性、承载能力和交易质量，在技术架构演进、高并发等领域有丰富的实践经验。

一、背景

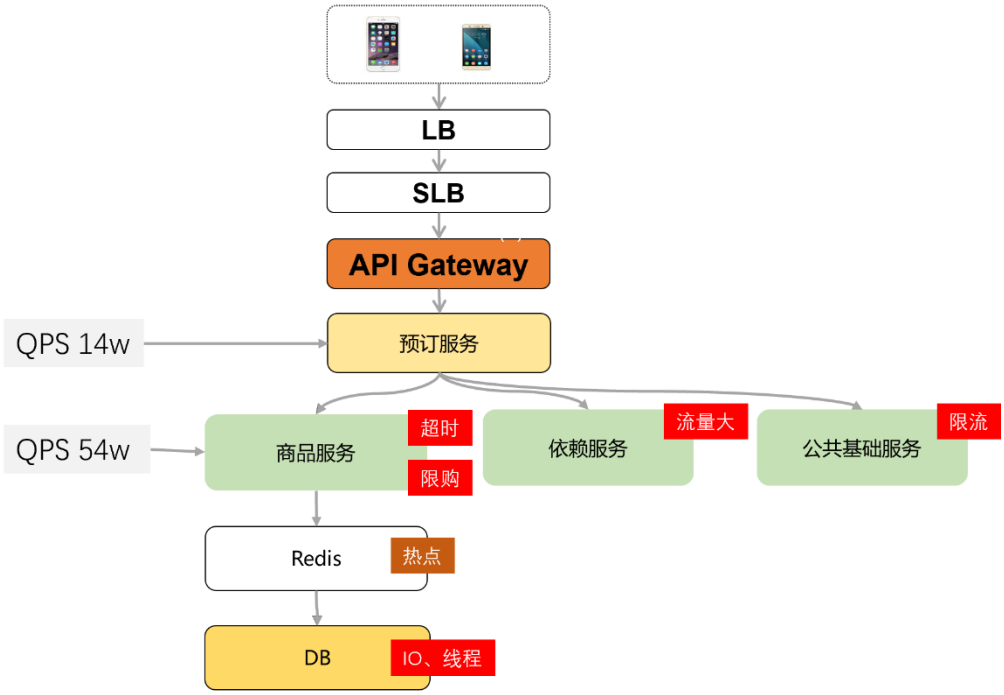
去年疫情后，为了加速启动旅游市场，湖北在全域范围内开展“与爱同行 惠游湖北”活动——全省所有 A 级旅游景区向全国游客免门票，敞开怀抱欢迎全国人民。本文将介绍在这一活动期间，线上预约抢票系统遇到的核心问题，系统的改造过程以及实施的一些经验。这是高并发、高可用场景下，提升系统稳定性的一次实战优化，希望能给面对同样问题的同学提供一些借鉴思路。

活动页面

二、风险与挑战

在活动初期，系统面临以下四类风险：

- 流量大，入口流量瞬间增长 100 倍，远超系统承载能力；
- 高并发下，服务稳定性降低；
- 限购错误；
- 热门门票、热门出行日期扣库存热点；



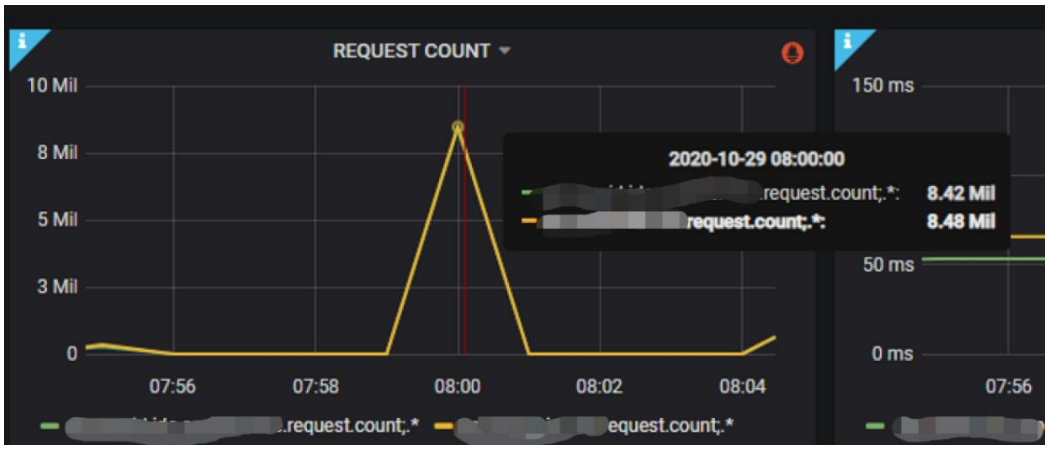
高并发下系统的挑战

下面我们一起来看下每个问题的影响和解决策略。

2.1 入口流量增长 100 倍

问题

活动开始时入口流量增长 100 倍，当前系统无法通过水平扩展解决问题。



请求量监控

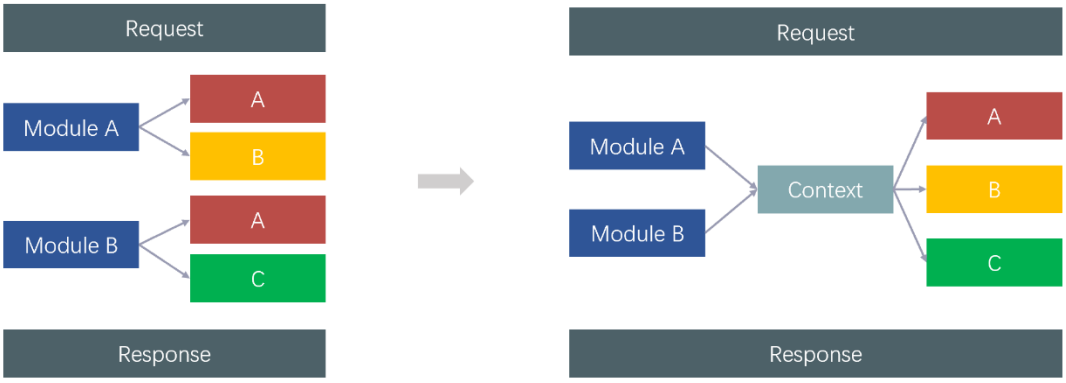
目标

提升入口应用吞吐能力，降低下游调用量。

策略

减少依赖

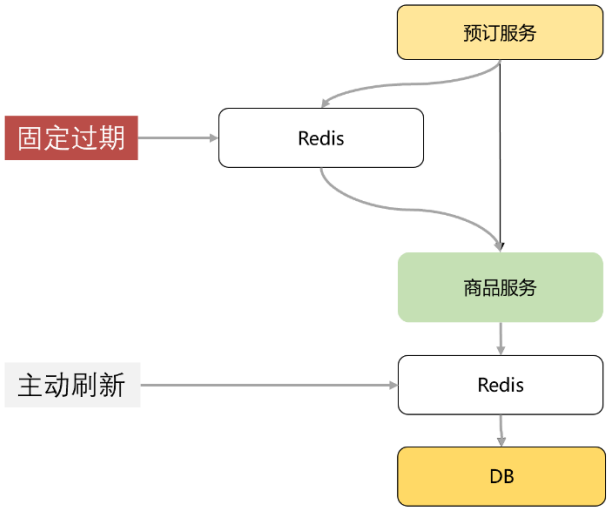
1）去除 0 元票场景不需要的依赖。例如：优惠、立减；
2）合并重复的 IO(SOA/ Redis/DB)，减少一次请求中相同数据的重复访问。



上下文传递对象减少重复 IO

提升缓存命中率

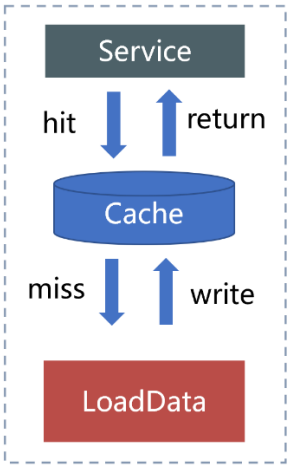
这里说的是接口级缓存，数据源依赖的是下游接口，如下图所示：



服务层-接口级缓存-固定过期

接口级缓存一般使用固定过期+懒加载方式来缓存下游接口返回对象或者自定义的 DO 对象。
当一个请求进来，先从缓存中取数据，若命中缓存则返回数据，若没命中则从下游获取数据

重新构建缓存，由于是接口级的缓存，一般过期时间设置都比较短，流程如下图：



固定过期+懒加载缓存

这种缓存方案存在击穿和穿透的风险，在高并发场景下缓存击穿和缓存穿透问题会被放大，下面会分别介绍一下这几类常见问题在系统中是如何解决的。

1）缓存击穿

描述：缓存击穿是指数据库中有，缓存中没有。例如：某个 key 访问量非常高，属于集中式高并发访问，当这个 key 在失效的瞬间，大量的请求就击穿了缓存，直接请求到下游（接口/数据库），造成下游压力过大。

解决方案：对缓存增加被动刷新机制，在缓存实体对象中增加上一次刷新时间，请求进来后从缓存获取数据返回，后续判断缓存是否满刷新条件，若满足则异步获取数据重新构建缓存，若不满足，本次不更新缓存。通过用户请求异步刷新的方式，续租过期时间，避免缓存固定过期。

例如：商品描述信息，以前缓存过期时间为 5min，现在缓存过期时间为 24H，被动刷新时间为 1min，用户每次请求都返回上一次的缓存，但每 1min 都会异步构建一次缓存。

2）缓存穿透

描述：缓存穿透是指数据库和缓存中都没有的数据，当用户不断发起请求，比如获取 id 不存在的数据，导致缓存无法命中，造成下游压力过大。

解决方案：当缓存未命中，在下游也没有取到数据时，缓存实体内容为空对象，缓存实体增加穿透状态标识，这类缓存过期时间设置比较短，默认 30s 过期，10s 刷新，防止不存在的 id 反复访问下游，大部分场景穿透是少量的，但是有些场景刚好相反。例如：某一类规则配置，只有少量商品有，这种情况下我们对穿透类型的缓存过期时间和刷新时间设置同正常的过期和刷新时间一样，防止下游无数据一直频繁请求。

3）异常降级

当下游出现异常的时候，缓存更新策略如下：

缓存更新：

- 下游是非核心：超时异常写一个短暂的空缓存（例如：30s 过期，10s 刷新），防止下游超时，影响上游服务的稳定性。
- 下游是核心：异常时不更新缓存，下次请求再更新，防止写入空缓存，阻断了核心流程。

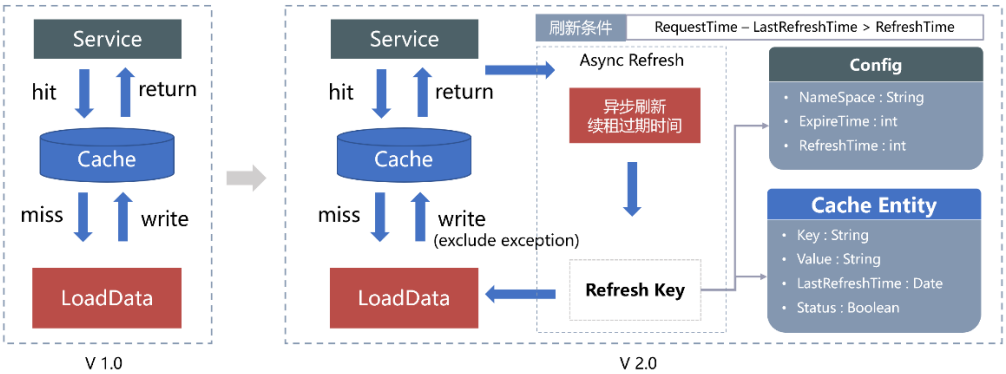
4）缓存模块化管理

将缓存 key 按照数据源做分类，每一类 key 对应一个缓存模块名， 每个缓存模块可以动态设置版本号、过期时间和刷新时间，并统一埋点与监控。模块化管理后，缓存过期时间粒度更为细致，通过分析缓存模块命中率监控，可以反推过期和刷新时间是否合理，最终通过动态调整缓存过期时间与刷新时间，让命中率达到最佳。

缓存模块命中率可视化埋点

我们将以上功能封装为了缓存组件，在使用的时只需要关心数据访问实现，既解决了使用缓存本身的一些共性问题，也降低了业务代码与缓存读写的耦合度。

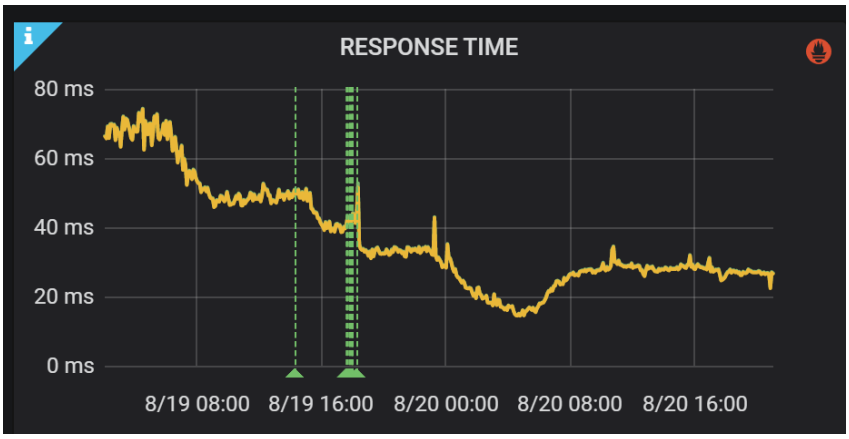
下图为优化前后缓存使用流程对比：



缓存使用对比

效果

通过解决缓存穿透与击穿、异常降级、缓存模块化管理，最终缓存命中率提升到 98%以上，接口性能（RT）提升 50% 以上，上下游调用量比例从 1 : 3.9 降低为 1 : 1.3，下游接口调用量降低 70%。

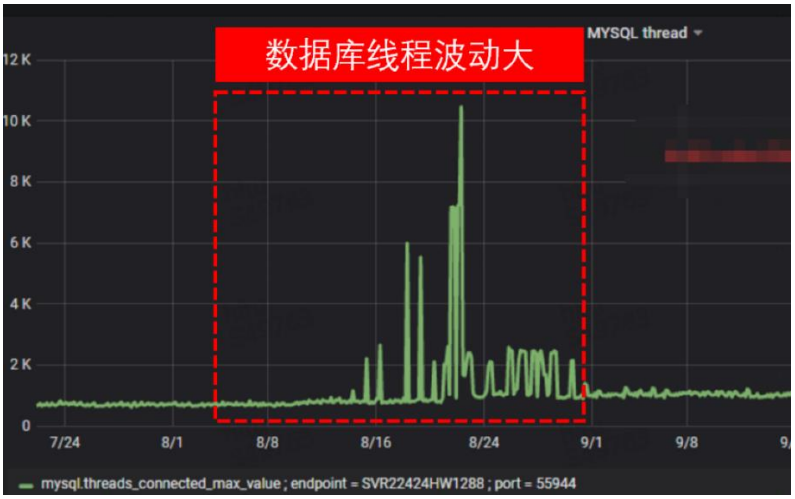


处理性能提升 50%

2.2 高并发下服务稳定性低

问题

在每天上午 8:00 抢票活动开始时，DB 连接池被打满，线程波动大，商品服务超时。



数据库线程波动

思考

- DB 连接池为什么会被打满?

- API 为什么会超时?
- 是 DB 不稳定影响了 API，还是 API 流量过大影响了 DB?

问题分析

1）DB 连接池为什么会被打满？分析三类 SQL 日志。

- Insert 语句过多 – 场景：限购记录提交，将限购表单独拆库隔离后，商品 API 依然超时（排除）
- Update 语句耗时过长 – 场景：扣减库存热点引起（重点排查）
- Select 高频查询 – 场景：商品信息查询

2）API 为什么会超时?

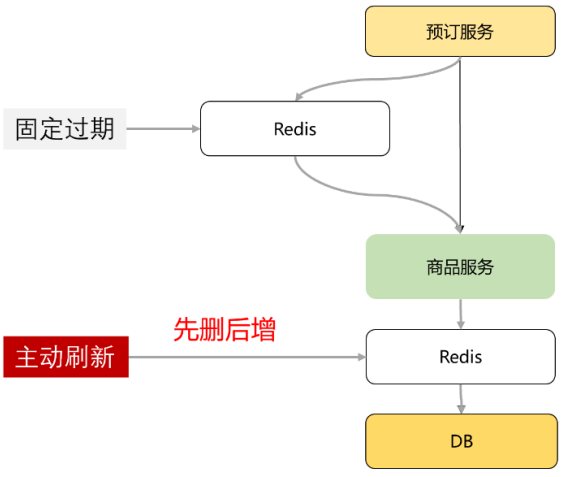
排查日志可以看到，8:00 活动开始后，大量热门商品信息查询到 DB 与 Select 高频查询一致。

3）是 DB 不稳定影响了 API，还是 API 流量过大影响了 DB?

根据#2 初步判断是由于缓存击穿，导致大量流量穿透到 DB。

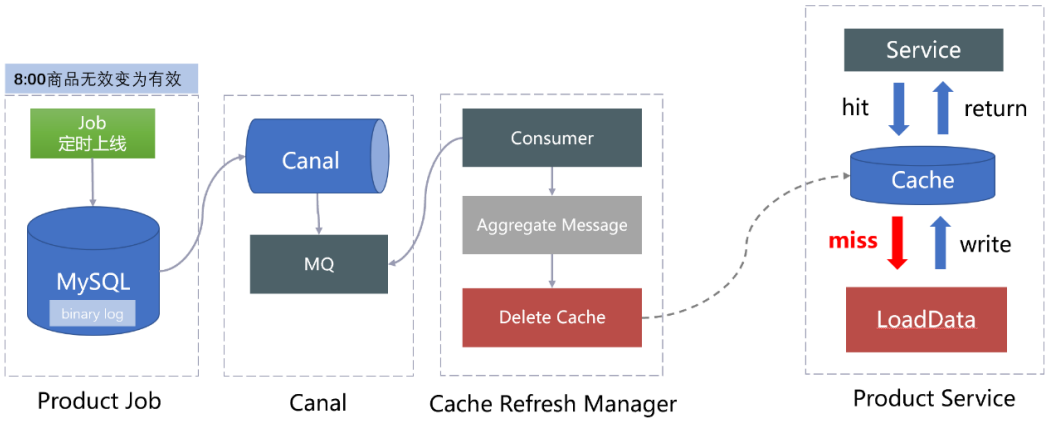
为什么缓存会被击穿?

梳理系统架构后发现，由于 8:00 定时可售通过离线 Job 控制，8:00 商品上线引发数据变更，数据变更导致缓存被刷新（先删后增），在缓存失效瞬间，服务端流量击穿到 DB，导致服务端数据库连接池被打满，也就是上文所说的缓存击穿的现象。



数据访问层-表级缓存-主动刷新

如下图所示，商品信息变更后主动让缓存过期，用户访问时重新加载缓存:



数据访问层缓存刷新架构（旧） - 消息变更删除缓存 Key

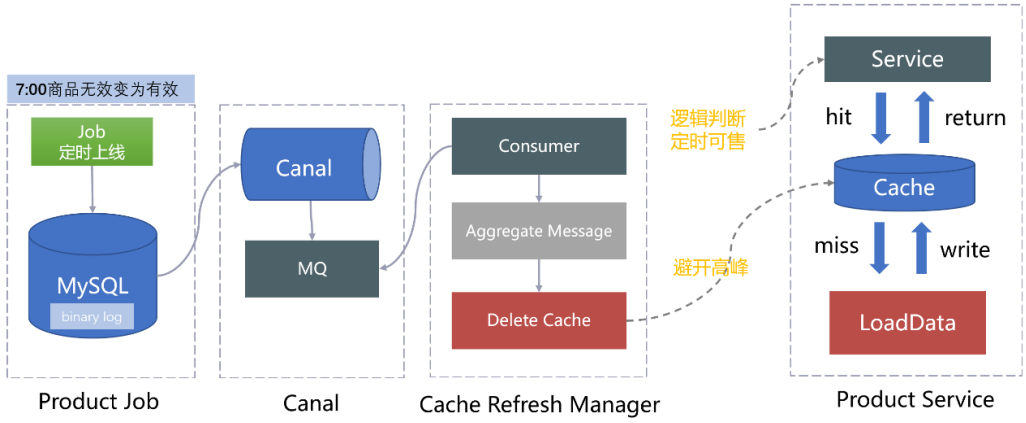
目标

为了防止活动时缓存被删除导致缓存击穿，流量穿透到 DB，采用了以下 2 种策略：

1）避开活动时数据更新导致缓存失效

我们将商品可售状态拆分商品可见、可售状态。

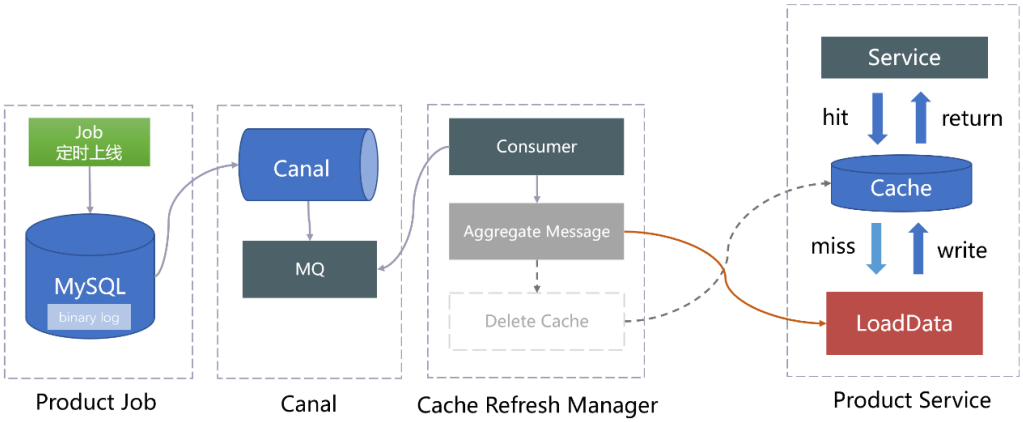
- 可见状态：7:00 提前上线对外可见，避开高峰；
- 可售状态：逻辑判断定时售卖，既解决定时上线修改数据后，导致缓存被刷新的问题，也解决了 Job 上线后，商品可售状态延迟的问题。



逻辑判断定时可售避开高峰缓存击穿

2）调整缓存刷新策略

原缓存刷新方案（先删后增）存在缓存击穿的风险，所以后面缓存刷新策略调整为覆盖更新，避免缓存失效导致缓存击穿。新缓存刷新架构，通过 Canal 监听 MySQL binlog 发送的 MQ 消息，在消费端聚合后，重新构建缓存。



数据访问层缓存刷新架构（新）- 消息变更重新构建缓存

效果

服务(RT)正常，QPS 提升至 21w。

上面两类问题与具体业务无关，下面我们介绍一下两个业务痛点：

- 如何防止恶意购买（限购）
- 如何防止库存少买/超买（扣库存）

2.3 限购

什么是限购?

限购就是限制购买，规定购买的数量，往往是一些特价和降价的产品，为了防止恶意抢购所采取的一种商业手段。

限购规则（多达几十种组合）例如：

1）同一出行日期同一景区每张身份证只能预订 1 张；
2）7 天内（预订日期）某地区只能预约 3 个景区且最多限购 20 份；
3）活动期间，预约超过 5 次，没有去游玩 noshow 限购；
问题

扣库存失败，限购取消成功（实际数据不一致），再次预订被限购了。

原因

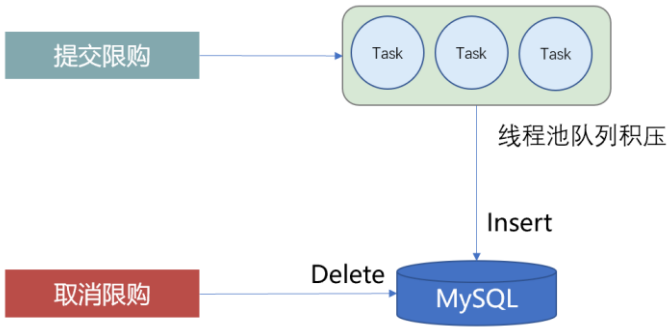
限购提交是 Redis 和 DB 双写操作，Redis 是同步写，DB 是线程池异步写，当请求量过大时，线程队列会出现积压，最终导致 Redis 写成功，DB 延时写入。在提交限购记录成功，扣库存失败后，需要执行取消限购记录。

如下图所示：

限购检查-提交限购-取消限购

在高并发的场景下，提交限购记录在线程池队列中出现积压，Redis 写入成功后，DB 并未写入完成，此时取消限购 Redis 删除成功，DB 删除未查到记录，最终提交限购记录后被写入，再次预订时，又被限购。

如下图：



线程队列积压，先提交的“提交限购”请求晚于“取消限购”

目标

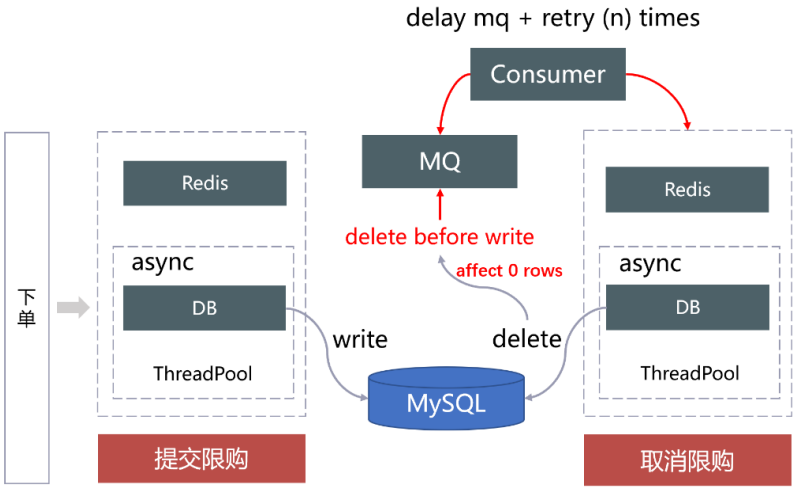
服务稳定，限购准确。

策略

确保取消限购操作 Redis/DB 最终一致。

由于提交限购记录可能会出现积压，取消限购时提交限购记录还未写入，导致取消限购时未能删除对应的提交记录。我们通过延迟消息补偿重试，确保取消限购操作(Redis/DB)最终一致。在取消限购的时候，删除限购记录影响行数为 0 时，发送 MQ 延迟消息，在 Consumer 端消费消息，重试取消限购，并通过埋点与监控检测核心指标是否有异常。

如下图所示：



下单-提交限购与取消限购

效果

限购准确，没有误拦截投诉。

2.4 扣减库存

问题

- 商品后台显示 1w 已售完，实际卖出 5000，导致库存未售完。
- MySQL 出现热点行级别锁，影响扣减性能。

原因

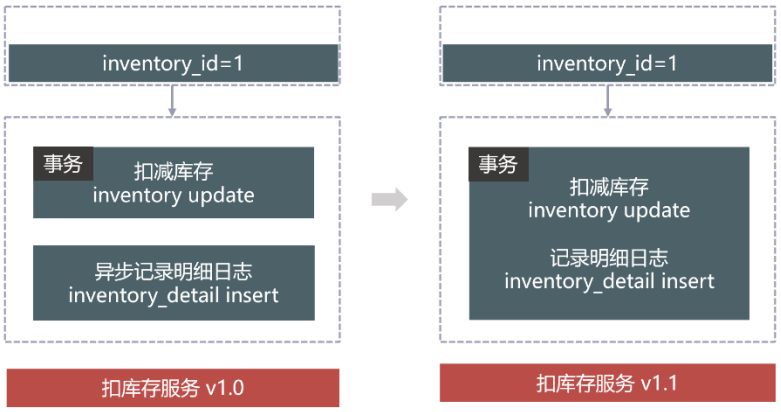
- 扣库存与库存明细 SQL 不在一个事务里面，大量扣减时容易出现部分失败的情况，导致库存记录和明细不一致的情况。
- 热门景点热门出行日期被集中预订，导致 MySQL 出现扣减库存热点。

目标

库存扣减准确，提升处理能力。

策略

1）将扣减库存记录和扣减明细放在一个事务里面，保证数据一致性。



DB 事务扣减库存

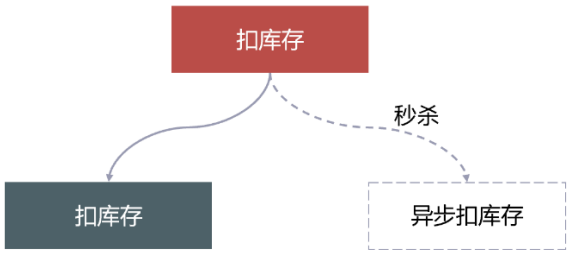
效果

优点：数据一致。

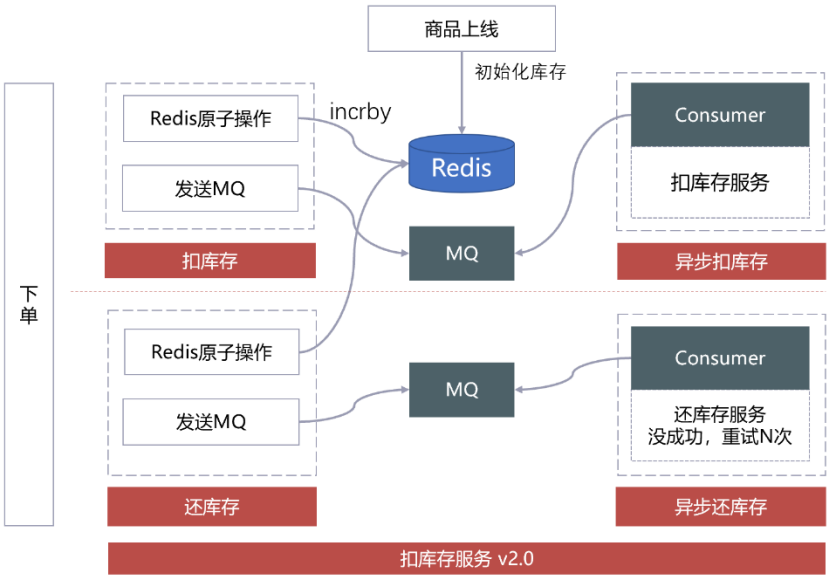
缺点：热点资源，热门日期，扣减库存行级锁时间变长，接口 RT 变长，处理能力下降。

2）使用分布式缓存，在分布式缓存中预减库存，减少数据库访问。

秒杀商品异步扣减，消除 DB 峰值，非秒杀走正常流程。



商品上线的时候将库存写入 Redis，在活动扣减库存时，使用 incrby 原子扣减成功后将扣减消息 MQ 发出，在 Consumer 端消费消息执行 DB 扣减库存，若下单失败，执行还库存操作，也是先操作 Redis，再发 MQ，在 Consumer 端，执行 DB 还库存，如果未查询到扣减记录（可能扣库存 MQ 有延迟），则延时重试，并通过埋点与监控检测核心指标是否有异常。



异步扣减库存

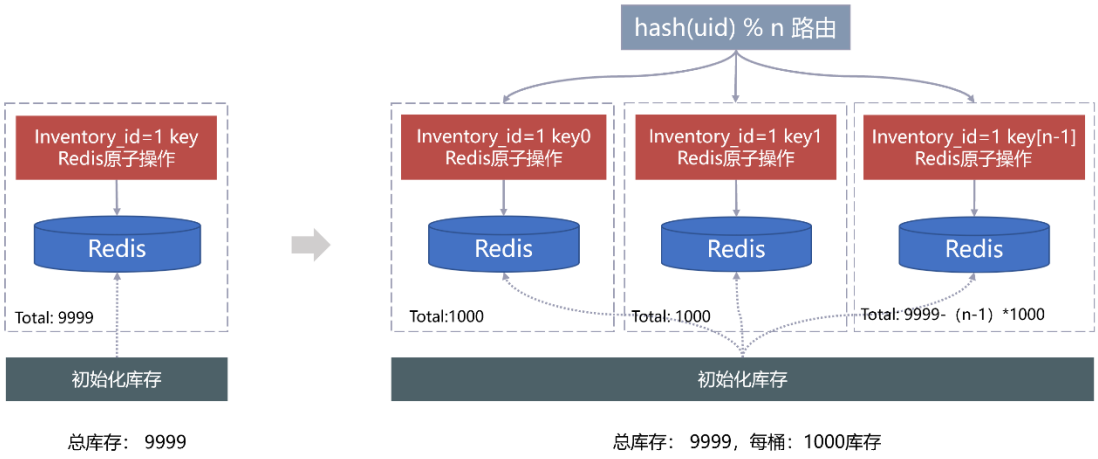
效果

- 服务 RT 平稳，数据库 IO 平稳
- Redis 扣减有热点迹象

3）缓存热点分桶扣减库存

当单个 Key 流量达到 Redis 单实例承载能力时，需要对单 key 做拆分，解决单实例热点问题。由于热点门票热门日期产生热点 Key 问题，观察监控后发现并不是特别严重，临时采用拆分 Redis 集群，减少单实例流量，缓解热点问题，所以缓存热点分桶扣减库存本次暂未实现，这里简单描述一下当时讨论的思路。

如下图所示：



缓存热点分桶扣减

分桶分库存：

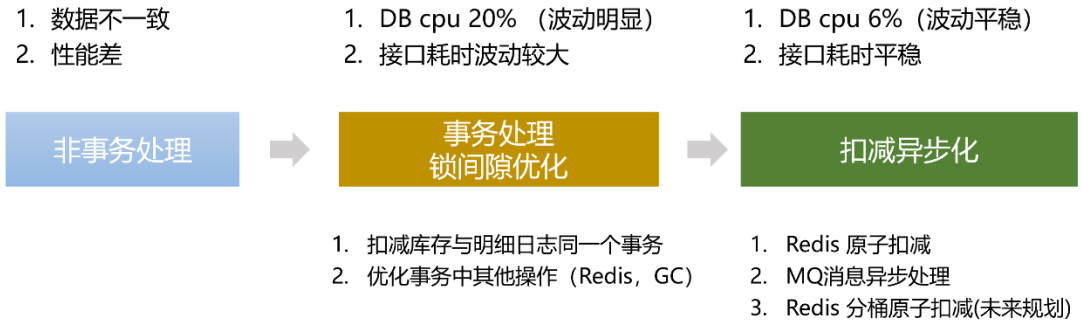
秒杀开始前提前锁定库存修改，并执行分桶策略，按照库存 Id 取模分为 N 个桶，每个分桶对应缓存的 Key 为 Key [0~ N-1]，每个分桶保存 m 个库存初始化到 Redis，秒杀时根据 Hash(Uid)%N 路由到不同的桶进行扣减，解决所有流量访问单个 Key 对单个 Redis 实例造成压力。

桶缩容：

正常情况下，热门活动每个桶中的库存经过几轮扣减都会扣减为 0。

特殊场景下，可能存在每个桶只剩下个位数库存，预订时候份数大于剩余库存，导致扣减不成功。例如：分桶数量为 100 个，每个桶有 1~2 个库存，用户预订 3 份时扣减失败。当库存小于十位数时，缩容桶的数量，防止用户看到有库存，扣减一直失败。

优化前后对比



扣减库存方案对比

三、回顾总结

回顾“与爱同行 惠游湖北”整个活动，我们整体是这样备战的：

- 梳理风险点：包括系统架构、核心流程，识别出来后制定应对策略；
- 流量预估：根据票量、历史 PV、节假日峰值预估活动峰值 QPS；
- 全链路压测：对系统进行全链路压测，对峰值 QPS 进行压测，找出问题点，优化改进；
- 限流配置：为系统配置安全的、符合业务需求的限流阀值；
- 应急预案：收集各个域的可能风险点，制作应急处理方案；
- 监控：活动时观察各项监控指标，如有异常，按预案处理；
- 复盘：活动后分析日志，监控指标，故障分析，持续改进；

本文阐述了在抢票活动中遇到的四个具有代表性的问题，在优化过程中，不断地思考和落地技术细节，沉淀核心技术，以最终达到让用户预订及入园顺畅，体验良好的目标。

质量保障

质量保障新手段，携程回归测试平台实践

【作者简介】Sedro，携程资深测试工程师，专注于测试技术探索及测试工具研发。

一、系统回归问题

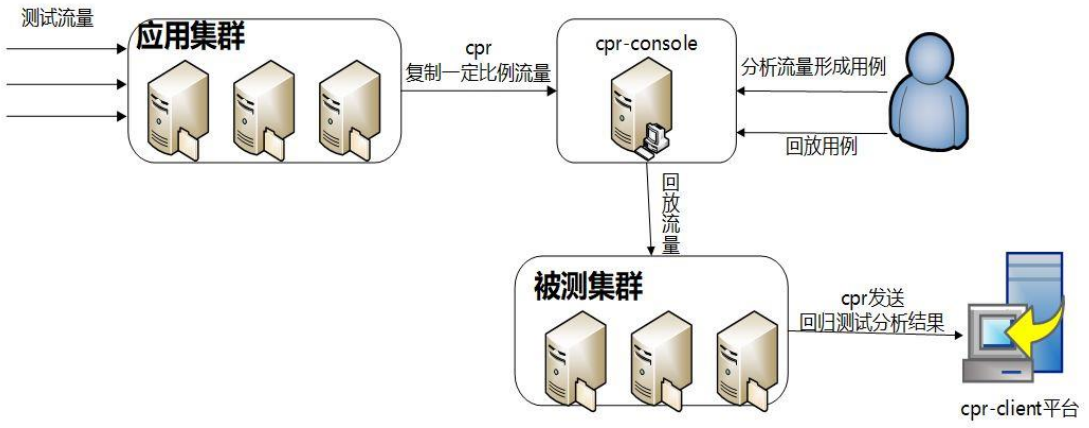
回归测试是软件生命周期一个十分重要的环节，但项目在随着版本的逐步迭代，功能日益增多，系统愈加复杂，在测试过程中测试人员常常需要回归稳定版本的功能以保证不被待发布版本需求所影响。若要对系统进行全方位回归，这个测试的工作量将会非常庞大，而且可能几百上千用例中才会发现一个甚至是 0 个问题，测试投入产出不成比例。

而 CPR 则为上述问题提供了较好的解决方案。它通过基于稳定版本的输出，对待发布版本的输出进行比较，同时还将校验两个版本对下游请求的差异。根据比对结果评判待发布版本是否正确，可以大大降低回归的工作量。

二、CPR 目标

- 大量真实流量确保覆盖率
- 将录制的流量作为用例管理起来进行自动化回归
- 流量回放支持子调用自动化 mock，避免回放产生脏数据
- 流量回放支持子调用结果的验证
- 减少人力资源

三、目标实现基本过程



1）首先将部署了稳定代码的服务器作为流量采集源。测试人员在进行功能、接口测试时，实现测试执行过程中主调用以及子调用的入参和返回值的录制。通过功能和接口测试实现对应用功能的全量覆盖，使得应用中请求流量都会被录制到。

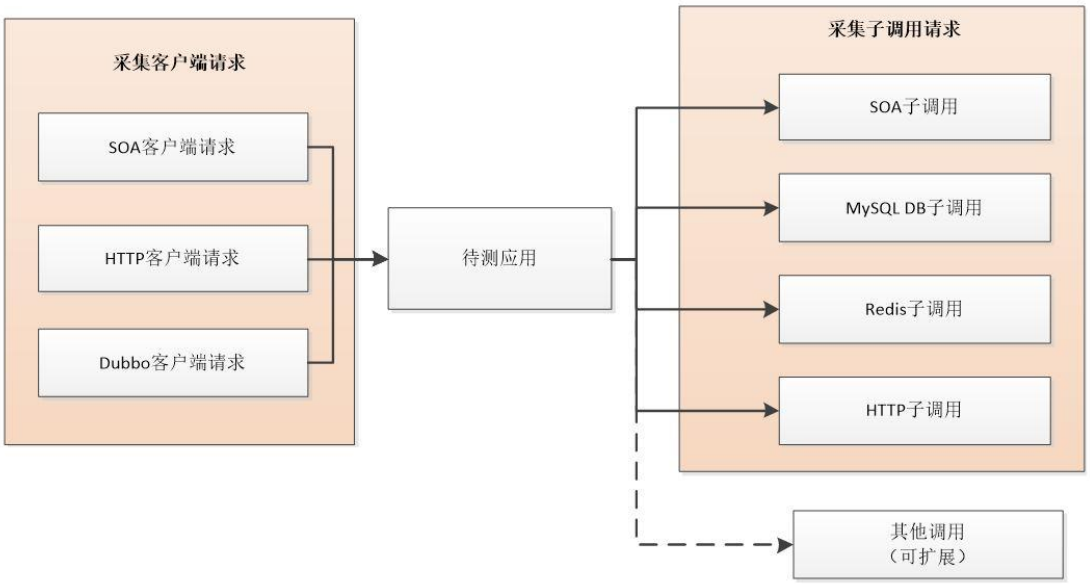
2）将录制到的请求流量复制到 console 平台，由测试人员分析有效的流量归纳为用例。后

续即可采用这些有效用例来对待发布版本进行回放和差异比对。

3）回放完成告知到测试人员，由测试人员对回放的差异结果进行确认，快速发现被测系统 bug 并解决。

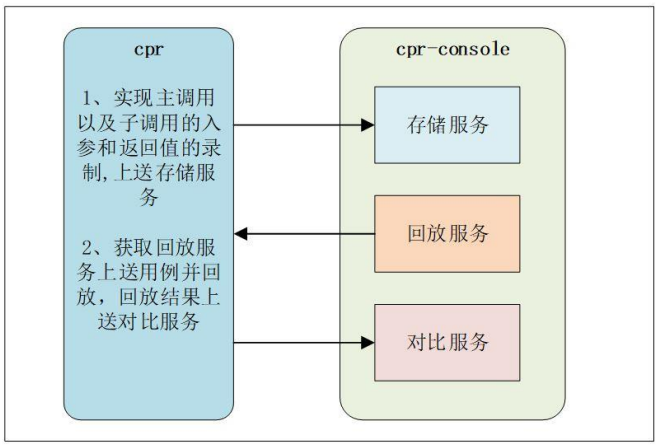
四、CPR 原理及实现

4.1. CPR 录制的数据抓取点



如上图所示，CPR 采集的数据主要为两方面：待测应用接收到的客户端请求及响应；待测应用接收请求后向外部服务的子调用请求及响应。CPR 目前支持的请求类型如上图橙色框中所示；但 CPR 本身设计时对各类请求采用插件式设计，使其具有较好的扩展性，对于后续其他需要捕获的类型能很好的进行支持。当前 CPR 系统支持的报文协议为 JSON 和 PB(ProtoBuf)。

4.2. CPR 结构简介



CPR 分为两大组件：

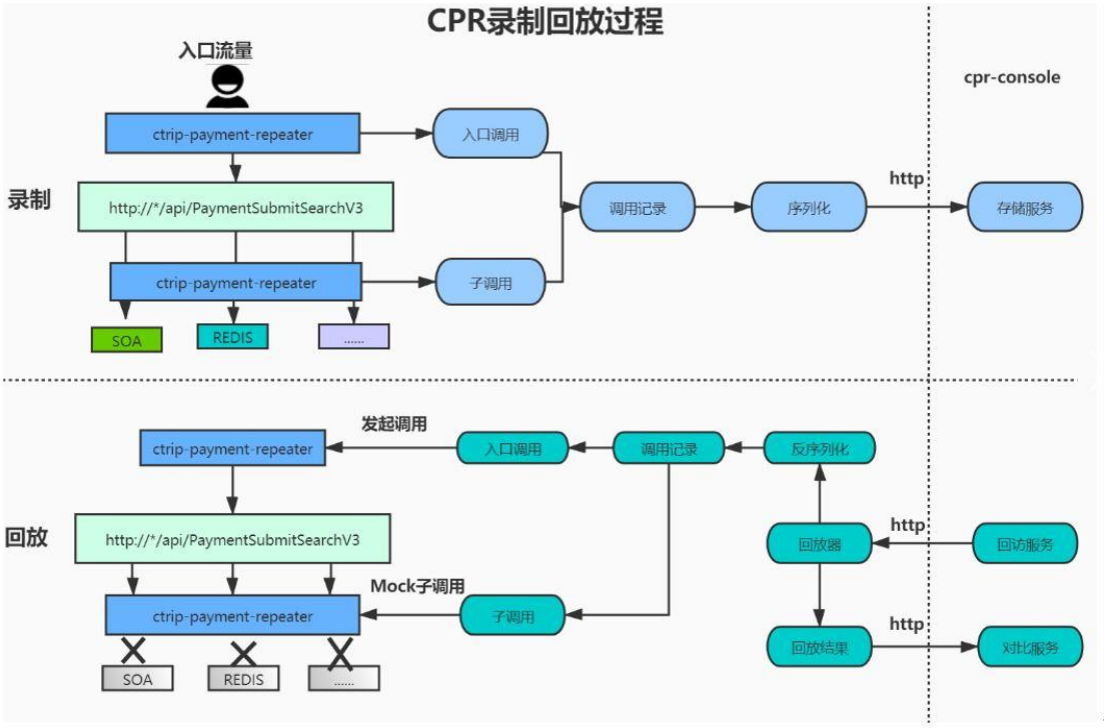
1）CPR (CtripPaymentRepeater) 组件，该组件基于开源的 jvm-sandbox 开发，用于录制和回放流量。此组件核心为两部分：

- CPRRecord：目标是在稳定代码环境中录制请求调用的入参和返回值，并上送到存储服务。使得 CPR Replay 具备回放流量的数据。
- CPRReplay：功能为接收回放服务提供的回放流量，在待测代码环境中进行回放，并将回放结果上送至对比服务，让其实现正常系统和待测系统返回结果比对差异的能力。

2）CPRConsole 组件，该组件主要录制/回放的配置管理；数据存储/数据对比等具备多种能力。主要包含三部分：

- 存储服务：对接收到的录制流量数据，将其持久化保存，待后续用户筛选有效流量。
- 回放服务：功能为将录制流量进行还原，然后对入口调用做一次流量的发起,使得 CPR Replay。
- 对比服务：对接收到的回放数据与录制数据进行差异比对。

4.3 CPR 处理流程



如上图所示，通过这两大组件的协同工作将稳定代码的流量自动捕获并持久化存储，然后由测试人员分析流量收藏为有效用例，对于后续待正式发布的代码，可以用有效用例来进行回放和差异比对，根据差异比对结果发现待测系统存在的问题。

该系统在处理比对时有两个与其他方案较大不同的特性：

1）对子调用 MOCK 处理：本方案采用的是记录下原始子调用内容及子调用返回内容；在系统内部完成对子调用返回的处理，不再需要实际的子系统或 mock 系统进行测试支撑。以此减小了环境、数据测试处理的难度和准确性。

2）子调用比对数据的处理：本方案采用的是比对子调用请求。例如一个 DB 操作，若请求内容是无误的，则认定请求正常；不再去比对具体的数据库存储值。这样的处理方式同样简化了测试环境搭建、测试数据准备，同时还解决了测试环境中数据共享变更可能带来的数据差异问题。

4.4 流量复制实现

在整个录制过程中，CPR 进行录制的处理流程主要是在 DefaultEventListener 类中实现。

1）DefaultEventListener 类

DefaultEventListener 是用来处理触发事件的类。不同的插件 (soa,mysql,http,redis 等) 可以自定义自己的 EventListener 来进行各自特殊的事件处理。目前插件默认使用 DefaultEventListener。

在 DefaultEventListener 中，承接的是所有事件的处理，也就是说录制/回放操作都是集中在这个类中实现的，只是根据不同的条件来区分是录制流量还是回放流量，从而判断该执行录制还是该执行回放。

当事件第一次过滤时，会进行一次初始化跟踪器。

```
/**
 * 初始化上下文;
 * 只有entrance插件负责初始化和清理上下文
 * 子调用无需关心traceContext信息（多线程情况下由ttl负责copy和restore，单线程由entrance负责管理）
 *
 * @param event 事件
 */
protected void initContext(Event event) {
    if (entrance && isEntranceBegin(event)) {
        Tracer.start();
    }
}
```

其中 traceId 实际上是这个跟踪器的独立标识。所以无论是录制和回放都会有其独立的 traceId。

2）Before 事件处理

对应的方法是 DefaultEventListener.doBefore。

```java
/**
 * 处理before事件
 *
 * @param event before事件
 */
protected void doBefore(BeforeEvent event) throws ProcessControlException {
    // 回放流量；如果是入口则放弃；子调用则进行mock
    if (RepeatCache.isRepeatFlow(Tracer.getTraceId())) {
        processor.doMock(event, entrance, invokeType);
        return;
    }

    Invocation invocation = initInvocation(event);
    invocation.setStart(System.currentTimeMillis());
    invocation.setTraceId(Tracer.getTraceId());
    invocation.setIndex(entrance ? 0 : SequenceGenerator.generate(Tracer.getTraceId()));
    invocation.setIdentity(processor.assembleIdentity(event));
```

根据 traceID 判断当前流量是否为回放流量，若为回放流量则调用 processor.doMock 方法执行 Mock，执行完成后直接返回，不再执行后续操作。

若当前流量为录制流量，则基于当前获取到的信息初始化 Invocation 实例，其中会调用插件中 processor#assembleIdentity、processor#assembleRequest、processor#assembleResponse、processor#assembleThrowable 四个方法，分别处理调用标识、请求参数、返回结果、抛出异常。最后将当前事件的 Invocation 信息存放到录制缓存中。

3）Return 事件处理/ Throw 事件处理

Return/ Throw 事件的处理逻辑基本一致。

```java
/**
 * 处理return事件
 *
 * @param event return事件
 */
protected void doReturn(ReturnEvent event) throws ProcessControlException {...}

/**
 * 处理throw事件
 *
 * @param event throw事件
 */
protected void doThrow(ThrowsEvent event) {
    if (RepeatCache.isRepeatFlow(Tracer.getTraceId())) {
        return;
    }
    Invocation invocation = RecordCache.getInvocation(event.invokeId);
    if (invocation == null) {
        log.debug("no valid invocation found in throw,type={},traceId={}", invokeType, Tracer.getTraceId());
        return;
    }
    invocation.setThrowable(processor.assembleThrowable(event));
    invocation.setEnd(System.currentTimeMillis());
    listener.onInvocation(invocation);
}
```

判断当前流量是否为回放流量，若为回放流量则直接返回，不执行后续操作。

若录制流量则从录制缓存中获取对应的 Invocation 实例，调用插件调用处理器的 assembleResponse/assembleThrowable 方法，并将 reponst/throwable 结果设置到 Invocation 实例中。

回调调用监听器 InvocationListener#onInvocation 方法，判断 Invocation 是入口调用还是子调用，如果是子调用则保存到录制缓存中，如果不是则调用消息投递器的 broadcastRecord 方法将录制记录序列化后上传给 cpr-console。

4.5 流量回放实现

回放流程，是由测试人员选择有效用例，在待测系统中执行回放。

回放服务接受到回放记录，对记录进行反序列化，还原到原来录制的对象，保证 classloader 与录制时候一致。CPR 在执行回放任务的过程中，首先会根据录制记录的信息，构造相同的请求，对被挂载的任务进行请求，并跟踪回放请求的处理流程，以便记录回放结果以及执行 mock 动作。整个回放过程比较复杂，主要包括以下几个个流程：

1）回放请求的处理

这里以 http 回放入口为例，RepeaterModule#repeat 方法是回放请求最初处理入口。添加 @Command 注解，使得外部可以使用 http 协议方式调用入口。

```java
/**
 * 回放http接口
 *
 * @param req    请求参数
 * @param writer printWriter
 */
@Command("repeat")
public void repeat(final Map<String, String> req, final PrintWriter writer) {
    try {
        String data = req.get(Constants.DATA_TRANSPORT_IDENTIFY);
        if (StringUtils.isEmpty(data)) {
            writer.write("invalid request, cause parameter {" + Constants.DATA_TRANSPORT_IDENTIFY + "} is required");
            return;
        }
        RepeatEvent event = new RepeatEvent();
        Map<String, String> requestParams = new HashMap<String, String>(16);
        for (Map.Entry<String, String> entry : req.entrySet()) {
            requestParams.put(entry.getKey(), entry.getValue());
        }
        event.setRequestParams(requestParams);
        EventBusInner.post(event);
        writer.write("submit success");
    } catch (Throwable e) {
        writer.write(e.getMessage());
    }
```

该方法在接收外部的回放请求后，对请求参数进行校验，校验通过后会将参数实例化，形成一个回放事件 RepeatEvent，发布到 EventBusInner 中被 RepeatSubscribeSupporter#onSubscribe 方法进行处理。

```java
    @AllowConcurrentEvents
    @Subscribe
    @Override
    public void onSubscribe(RepeatEvent repeatEvent) {
        Map<String, String> req = repeatEvent.getRequestParams();
        try {
            final String data = req.get(Constants.DATA_TRANSPORT_IDENTIFY);
            if (StringUtils.isEmpty(data)) {
                log.info("invalid request cause meta is null, params={}", req);
                return;
            }
            log.info("subscribe success params={}", req);
            final RepeatMeta meta = SerializerWrapper.hessianDeserialize(data, RepeatMeta.class);
            RepeaterResult<RecordModel> pr = StandaloneSwitch.instance().getBroadcaster().pullRecord(meta);
            if (pr.isSuccess()){
                DefaultFlowDispatcher.instance().dispatch(meta, pr.getData());
            } else {
                log.error("subscribe replay event failed, cause ={}", pr.getMessage());
            }
        } catch (SerializeException e) {
            log.error("serialize failed, req={}", req, e);
        } catch (Exception e) {
            log.error("[Error-0000]-uncaught exception occurred when register repeat event, req={}", req, e);
        }
    }
```

处理过程为首先会将 RepeatEvent 的参数进行反序列化，获取回放相关的录制记录的信息，

然后通过这些信息从 prepeater-console 拉取对应的录制记录详情（RecordModel），最后用默认流量分发器 DefaultFlowDispatcher 进行分发。

2）回放任务的分发

回放任务的分发由 DefaultFlowDispatcher#dispatch 方法实现。

```
@Override
public void dispatch(RepeatMeta meta, RecordModel recordModel) throws RepeatException {
    if (recordModel == null || recordModel.getEntranceInvocation() == null || recordModel.getEntranceInvocation().getType() == null) {
        throw new RepeatException("invalid request, record or root invocation is null");
    }
    Repeater repeater = RepeaterBridge.instance().select(recordModel.getEntranceInvocation().getType());
    if (repeater == null) {
        throw new RepeatException("no valid repeat found for invoke type:" + recordModel.getEntranceInvocation().getType());
    }
    RepeatContext context = new RepeatContext(meta, recordModel, TraceGenerator.generate());
    // 放置到回放缓存中
    RepeatCache.putRepeatContext(context);
    repeater.repeat(context);
}
```

主要是针对回放任务的信息进行校验，校验失败会抛出错误到上层。通过校验的回放任务，则会初始化回放上下文信息，并存放到回放缓存中。根据回放任务的入口插件类型，获取对应插件的 repeater 进行回放处理。

3）回放结果执行与记录

回放任务执行、回放结果记录的实现逻辑在 AbstractRepeater#repeat 中实现。

```
@Override
public void repeat(RepeatContext context) {
    Stopwatch stopwatch = Stopwatch.createStarted();
    RepeatModel record = new RepeatModel();
    record.setCaseId(context.getMeta().getCaseId());
    record.setBatchNo(context.getMeta().getBatchNo());
    record.setRepeatId(context.getMeta().getRepeatId());
    record.setTraceId(context.getTraceId());
    try {   // 根据之前生成的traceId开启追踪
        Tracer.start(context.getTraceId());
        // before invoke advice
        RepeatInterceptorFacade.instance().beforeInvoke(context.getRecordModel());
        Object response = executeRepeat(context);
        // after invoke advice
        RepeatInterceptorFacade.instance().beforeReturn(context.getRecordModel(), response);
        stopwatch.stop();

        List<MockInvocation> mockInvocations = RepeatCache.getMockInvocation(context.getTraceId());
                                                                                                    {
            if(mockInvocations.size()>0 && mockInvocations.get(mockInvocations.size() - 1).getCurrentOff
                MockInvocation mock = mockInvocations.get(mockInvocations.size() - 1);
                record.setResponse(mock.getCurrentArgs()[0]);
                mockInvocations.remove(mockInvocations.size() - 1);
                record.setMockInvocations(mockInvocations);
            }
        }else{
            record.setResponse(response);
            record.setMockInvocations(mockInvocations);
        }

        record.setCost(stopwatch.elapsed(TimeUnit.MILLISECONDS));
        record.setFinish(true);
    } catch (Exception e) {
        log.error("repeat error:"+e);
```

通过 executeRepeat 方法交由各个插件的回放器实例来调用。当插件开始执行回放任务时，会根据回放上下文以及当前 repeater 的应用信息初始化回放结果记录 RepeaterModel 的实例。

初始化回放结果记录后，初始化回放线程跟踪并触发回放请求并获取回放请求返回的结果。

获取回放请求返回的结果后，停止线程跟踪，将回放结果以及当前的一些状态信息保存到回放结果记录的实例中。

最后通过调用消息投递器的 AbstractBroadcaster#broadcastRepeat 方法将回放结果记录序列化后上传到 crp-console 保存。

4）回放过程中的 Mock 处理

在回放过程中，直接在 DefaultEventListner#Before 事件处理中执行。

```
/**
 * 处理before事件
 *
 * @param event before事件
 */
protected void doBefore(BeforeEvent event) throws ProcessControlException {
    // 回放流量：如果是入口则放弃；子调用则进行mock
    if (RepeatCache.isRepeatFlow(Tracer.getTraceId())) {
        processor.doMock(event, entrance, invokeType);
        return;
    }

    Invocation invocation = initInvocation(event);
    invocation.setStart(System.currentTimeMillis());
    invocation.setTraceId(Tracer.getTraceId());
    invocation.setIndex(entrance ? 0 : SequenceGenerator.generate(Tracer.getTraceId()));
    invocation.setIdentity(processor.assembleIdentity(event));
```

processor#doMock 方法，交给插件调用处理器执行 mock，并且只对子调用事件执行 mock。

```
public void doMock(BeforeEvent event, Boolean entrance, InvokeType type) throws ProcessControlException {
    /*
     * 获取回放上下文
     */
    RepeatContext context = RepeatCache.getRepeatContext(Tracer.getTraceId());
    /*
     * mock 执行条件
     */
    if (!skipMock(event, entrance, context) && context != null && context.getMeta().isMock())
{
        try {
            /*
             * 构建 mock 请求
             */
            final MockRequest request = MockRequest.builder()
                    .argumentArray(this.assembleRequest(event))
                    .event(event)
                    .identity(this.assembleIdentity(event))
                    .meta(context.getMeta())
                    .recordModel(context.getRecordModel())
                    .traceId(context.getTraceId())
                    .repeatId(context.getMeta().getRepeatId())
                    .index(SequenceGenerator.generate(context.getTraceId()))
                    .type(type)
```

```
                    .build();
            /*
             * 执行 mock 动作
             */

                final MockResponse mr =
StrategyProvider.instance().provide(context.getMeta().getStrategyType()).execute(request);
                 /*
                 * 处理策略推荐结果
                 */
                switch (mr.action) {
                    case SKIP_IMMEDIATELY:
                        break;
                    case THROWS_IMMEDIATELY:

ProcessControlException.throwThrowsImmediately(mr.throwable);
                        break;
                    case RETURN_IMMEDIATELY:
                     //                if(!type.equals(InvokeType.MYSQL)          &&!
type.equals(InvokeType.MYBATIS)){

ProcessControlException.throwReturnImmediately(assembleMockResponse(event,
mr.invocation));
                    //      }
                        break;
                    default:
                        ProcessControlException.throwThrowsImmediately(new
RepeatException("invalid action"));
                        break;
                }

        } catch (ProcessControlException pce) {
            throw pce;
        } catch (Throwable throwable) {
            ProcessControlException.throwThrowsImmediately(new
RepeatException("unexpected code snippet here.", throwable));
        }
    }
}
```

AbstractInvocationProcessor#doMock 方法的实现，与 AbstractRepeater#repeat 相似，实际调用时是通过插件中各自的 Processor 的实例进行调用的。

当开始执行回放 mock 时，会先获取回放上下文的信息。根据回放上下文的信息，判断是否

跳过 mock 的执行。当需要执行 mock 时，会根据回放上下文中的信息初始化 MockRequest 实例，通过 mock 策略计算获取 MockResponse。

根据 MockResponse 返回状态进行不同的操作:

1)当 MockResponse 执行结果是返回正常内容时，就会抛出一个终止当前调用操作并返回当前的返回结果，用 ProcessControlException 异常来阻挡需要 mock 的方法的实际调用。

2)当 MockResponse 执行结果是返回一个异常时，就会抛出一个终止当前调用操作并抛出 ProcessControlException 异常，此时需要 mock 的方法不会被真实调用。

3)只有当 MockResponse 返回状态是 skip 时，则不对这个调用事件做任何处理，调用会真实发生，其他任何状态都会通过抛出 ProcessControlException 异常来阻挡这个事件的实际调用。

4.6. 缺陷判定简析

1）对于读操作，我们主要关注在相同请求下正常系统和待测系统的返回结果的差异，读接口也提倡对所有对外请求进行 mock，这样回放时能保持当时的一个现场环境，保证验证的准确性。

2）对于写操作，只验证接口返回结果是不够的，需要验证它具体写入的数据是否正确。例如创建支付订单会调用 PPI 的写订单 PPI.Bill.CreatePaymentBill 接口，那么我们需要验证回放时调用 PPI 的参数和录制时调用 PPI 的参数是否一致。

3）对比默认是全对象字段逐一对比。由于录制环境和回放环境所处环境不同，有一些必然不一致的信息，例如随机数、时间，以及系统 ip 等等，这些内容系统做了默认不比对处理。系统对于子调用是默认对比所有子调用，也可以通过平台配置排除一些不关心的子调用的对比。

4.7 数据安全保护

系统录制与回放过程中基于对通讯、数据的安全保护要求，系统实现了对公司神盾安全系统的接入，使得传输过程、落地数据及日志等涉及安全要求的数据内容都符合了信息安全要求。也确保了使用者不会接触到敏感数据等限制性内容。

五、小结

本文系统的介绍了 CtripPaymentRepeater 的目标以及实现原理以及流量复制、流量回放的技术方案，希望给因频繁进行回归测试而困扰的朋友一些帮助和借鉴。

在系统成熟的前提下，可以扩展本平台进行生产数据的采集比对。当这一步实现后，可以想见回归将真正不再是一件难事，尤其是生产回归需要面对的数据污染等问题都将不再是问题。同时该平台还可演化为 MOCK 平台，这样的 MOCK 将不再依赖任何系统，只需在待测

应用的服务器上增加一个 Agent 即可完成。

携程机票前端安卓虚拟机测试集群建设实践

【作者简介】Liang，携程研发总监，关注 DevOps，前端&服务端质量保障、能效提升；Tony，携程资深测试经理，关注自动化测试框架及平台类工具开发。

一、背景

在携程内部业务高频率敏捷迭代发布节奏下，线上生产服务质量需要同步快速提升。这就依赖自动化测试的覆盖率提升，测试任务执行频次提升，测试任务执行速率提升。

通过调研机票前台内部，以及携程其他事业部前台研发和测试团队，发现存在普遍共性痛点。需要清晰化解耦自动化测试体系的三层架构，解决当前核心瓶颈。

- SaaS 上层服务，例如 UI 测试，API 测试，兼容性测试，稳定性测试。
- PaaS 中层服务，例如测试用例管理，测试任务调度，测试框架驱动。
- IaaS 底层服务，测试用例代码最终执行和测试报告输出功能。

Iaas 底层服务，公司各事业部的前端团队普遍通过采购手机设备，组建自动化测试设备小集群方案，或者基于开发测试人员的个人测试手机进行本地测试执行工作。这两种方式都存在一些问题。

1.1 IaaS 底层服务现状问题

1）测试手机采购成本较高。需要覆盖众多系统版本，众多厂商型号，以及持续更新换代。

2）测试手机的厂商型号差异化，导致自建小型测试设备集群的技术方案和运维管理困难。并且长期运行导致手机电池老化加速，电池膨胀损坏主板芯片，甚至存在潜在火灾隐患。

3）测试手机的用例执行稳定性和规模化受限，导致 PaaS 及 SaaS 上层的测试任务集整体执行效率降低。测试用例执行需要排队等待，测试任务容易运行失败，测试任务执行总耗时长。在集中发布日或大版本发布期间，大批量集成和回归测试用例集的任务堵塞拥挤。

1.2 IaaS 底层基建不稳对业务研发的影响

1）自动化测试工作的推广普及率和覆盖率受限。开发和测试人员在自动化测试工作中，很多耗时用于分析 Failed Job 日志，重启重试测试任务，修复运行兼容性问题。

2）海量测试用例集的总运行耗时较长，导致 DevOps 闭环反馈流程迟缓。于是大部分团队将自动化测试任务的频率延长到每天一次，每周一次，甚至每个版本一次。

3）自动化测试落地受阻，导致研发团队的生产交付质量被迫继续依赖手工测试团队的人工密集重复执行。手工测试容易人为疏忽遗漏，以及带来研发成本的固定支出。

二、项目目标

重构 IaaS 底层基建系统，降低采购和运维成本，提升测试任务可靠性和性能速度。提供通用测试设备服务，无缝支持多种上层测试框架，方便全公司各种前端团队的测试框架系统低成本接入。

三、系统选型

行业方案主要有三种：公有云真机集群、私有云真机集群和私有云虚拟机集群。

3.1 公有云真机集群

公有云真机集群是指使用行业内一些公司提供的云真机服务，例如 Testin，WeTest，以及华为，三星等厂商提供的真机云测实验室等，通常按照使用时间和使用设备数量收费。

优点：无需自建，公有云真机集群的设备型号较为完备。

缺点：费用成本中等，但是通常仅支持少量几种主流成熟测试框架，公司内各团队历史积累的测试用例集迁移成本较高。并且无法支持测试任务运行时依赖的众多内网系统，例如 Mock 服务、SOA 服务等等。导致真正可测试覆盖的场景受限，对线上生产交付的质量保障有限。

3.2 私有云真机集群

自建方式，需要采购真机设备和专用机柜，如需满足大规模测试用例集的高频率高并发执行，就要相应采购大量手机设备。

优点：真机设备的性能较好，并且可以针对性覆盖一些特殊测试场景，例如挖孔屏、折叠屏，特定厂商 API 等等。

缺点：设备数量决定了测试用例集合的并发执行速度，因此前期成本投入较高，只有当大量研发团队和测试用例任务接入后，才能逐渐平摊降低成本，并且始终存在设备运维更新换代的成本支出。

现状：携程机票前端自动化测试开发团队自 2017 年至今使用该方案实现小型测试设备集群，通过若干台 Mac mini 驱动管理大约 20 台手机真机设备。目前该方案仍然持续运维，作为私有云虚拟机集群的补充。真机设备来源于常规采购的日常开发调试工作的淘汰换置。随着使用时间增加，安卓系统更新换代，部分设备性能逐渐下降，给研发人员日常开发调试使用带来不便。于是我们就将其换置托管到自动化测试集群，发挥余热。

3.3 私有云虚拟机集群

自建方式，使用安卓虚拟机镜像（Android Virtual Device，以下简称 AVD）执行测试，以此组成测试设备集群，搭配一套管理系统对其进行统一调度。

优点：投入成本低，无需采购真机，便于根据使用量进行快速扩缩容，统一标准化管理，7x24小时可用，并且可以无缝衔接各种内部测试框架和内网依赖服务。

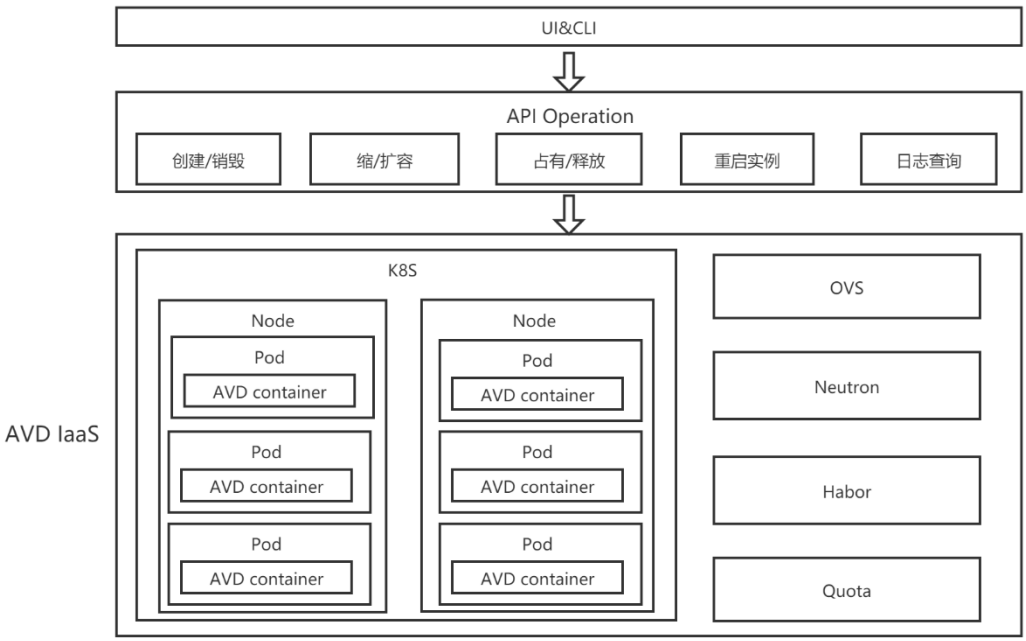
缺点：性能较真机略低，无法满足少量特殊测试场景，无法覆盖特定厂商机型定制功能。

随着 Google Android 官方团队对 AVD 相关组件的逐步优化，其在 X86 桌面环境模拟运行 ARM 指令的速度大大提升，并逐步剥离其与 Android Studio / SDK 的耦合，更加易于独立部署。随着 K8S，Docker 技术的成熟与普及，Google 开源了 android-emulator-container-scripts 实验性项目，使得 AVD + Docker + K8S 技术方案具备高可行性。因此我们启动了该项目的研发工作。

四、系统架构

主要由三部分组成：

- 容器实例层：基于 K8S 进行 AVD 容器的调度、编排，AVD 容器内包含 Android 模拟器及完整运行环境、驱动程序和预安装的系统依赖项，采用 OVS、Neuron、Quota、Harbor 组件实现网络通信、虚拟化管理、镜像存储和分发。
- 调度管理层：Android 模拟器的创建、销毁、缩扩容、占用、释放、重启、日志排查等，以 API 方式提供服务。
- 操作使用层：Android 模拟器的 Web GUI 可视化和 CLI 命令行操作使用。



4.1 容器实例层

AVD Container 的调度编排，使用了内部现有的 K8S 管理服务，为适应 AVD 容器化的使用要求，K8S 管理服务做了以下适配处理：

- Node 配置修改：开启 KVM 嵌套虚拟化支持
- K8S 容器化参数修改：设置 containers securityContext privileged = true

K8S 管理服务为每台 AVD 容器设备分配固定 IP，保证在部署和启动 AVD Container 后，调用者可通过固定 IP 访问。

Node 资源配置

携程 App 大部分是信息数据检索页面型应用场景，对于图像和音视频处理的要求不高，Node 采用 Intel(R) Xeon(R) CPU E5-2680 V3 @2.5GHZ 24Core 48Thread 192G Memory 配置，未配置 GPU 显卡。

当前 Node 配置的特点是内存足够，CPU 不足，所以考虑对 CPU 进行超分。虽然 CPU 超配可能会导致应用高并发下运行变慢，考虑到服务于测试环境，一定的性能损耗可以容忍。

超配策略的原则是在保证服务质量的同时，尽量提高资源的利用率。通过进行节点压测，分析历史真实使用数据，确定了 20%超分，高并发下 AVD container 仍然可以正常工作。

Pod 资源配置

CPU：4C
内存：8G（初始启动消耗 2.9G，运行时消耗 4G）
显卡：使用 Google SwiftShader 软加速库，基于 CPU 进行图形渲染

SwiftShader 介绍

SwiftShader 是 Vulkan 图形 API12 的基于 CPU 的高性能实现。其目标是为高级 3D 图形提供硬件独立性，可支持 Android 和 Chrome（OS）构建环境。为了向用户提供最佳性能，SwiftShader 使用多种方法高效地在 CPU 上执行图形计算。动态代码生成使在运行时针对现有任务自定义代码成为可能，与更常见的编译时优化完全不同。通过使用 Reactor 简化这种复杂的方法，Reactor 是一种自定义 C++ 嵌入式语言，具有直观的命令式语法。SwiftShader 还可以单指令多线程 (SIMT) 方式使用向量运算，并结合使用多线程技术来提高 CPU 可用内核和矢量单元之间的并行性。这样可实现实时渲染，其用途包括在 Android 上进行应用串流等。

使用 Airtest 测试框架，选取机票流程中的常用页面，从页面渲染、元素获取，模拟点击元素这几个维度，对 AVD 设备和普通 Android 设备进行性能对比测试。

性能数据	AVD 设备	普通 Android 真机
页面渲染（等待关键元素完成加载）	5-8 秒	3-5 秒
模拟点击元素	2 秒	2.1 秒
元素获取	600ms	500ms

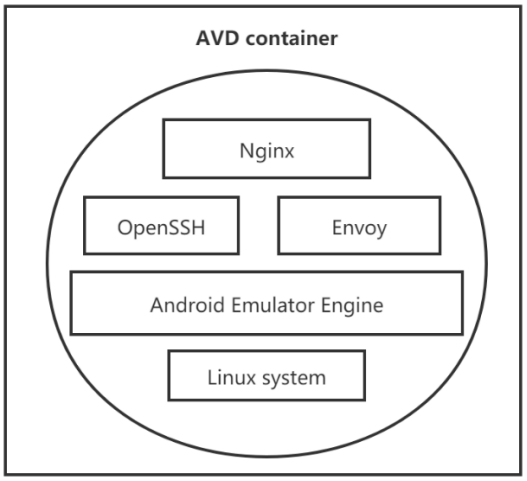
综上所述，AVD 设备除了页面渲染性能比真机略慢，其他维度和普通 Android 真机性能基本持平，能够满足日常集成测试需求。

AVD Container 内运行 AVD Docker 镜像，镜像构建采用了 Google 开源的 android-emulator-container-scripts 技术方案，基于公司内部统一的 Linux 系统基础镜像，自定义 Dockfile 生成 AVD Image，并上传至内部 Docker Hub 系统，镜像文件主要包含：

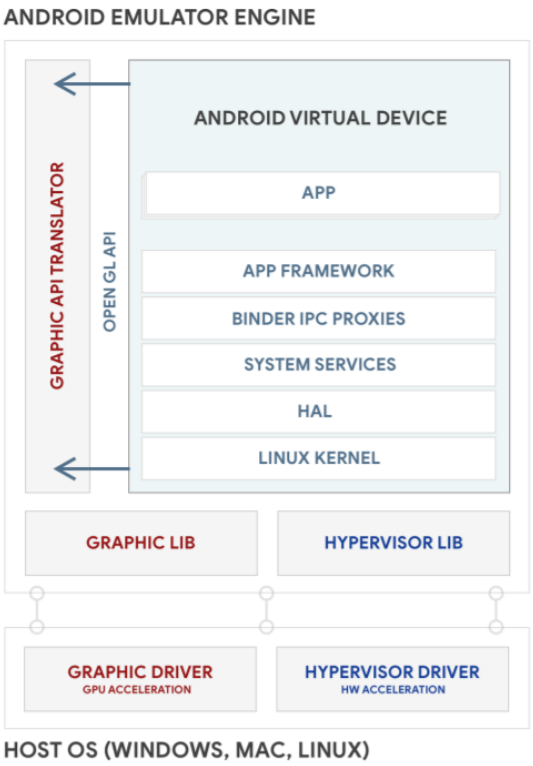
- Linux 操作系统
- Android 模拟器引擎
- 驱动程序和一些预安装的系统工具、网络服务代理

支持的 Android 系统版本

- Android 11 (API 级别 30)
- Android 10（API 级别 29）
- Android 9（API 级别 28）
- Android 8.1（API 级别 27）
- Android 8.0（API 级别 26）
- Android 7.1（API 级别 25）
- Android 7.0（API 级别 24）
- Android 6.0（API 级别 23）
- Android 5.1（API 级别 22）
- Android 5.0（API 级别 21）



安卓模拟器架构图（源自 Google Android 官网）



由于 Google 官方提供的 android-emulator-container-scripts 只在 Debian 和 Ubuntu 下进行过测试，我们在 Centos 系统下发现该脚本有诸多问题，因此沿用 Ubuntu 作为基础镜像。

按该脚本帮助文档中的方案激活虚拟环境，通过运行 emu-docker interactive --start 命令，以交互方式选择要使用的 android 和模拟器版本，之后将创建一个 docker 文件。

编辑 Dockfile 文件，并做以下修改：

- 修改 from 基础镜像
- 安装并配置 ssh 服务，用于远程管理容器
- 配置容器代理服务器，使 APP 可访问外网地址(可选)
- 修复容器启动过程中发现的 Dockfile 脚本问题等

将以上内容组合在一起生成 Docker 镜像，这样可以创建一个完整的运行环境，在其中运行 Android 模拟器，使得查找系统映像，管理系统依赖以及运行安卓模拟器变得非常容易。

4.2 调度管理层

实现 AVD 设备的创建、销毁、扩缩容与使用管理、设备状态监控等。restAPI 采用 java spring boot 技术实现，主要包括以下几个接口：

- 获取设备列表接口：获取设备状态、IP
- 占用/释放设备接口：占用时分配可用时长，时间到达后自动释放

- 扩容设备接口：耗时小于 2 分钟，单批最大 50 台
- 缩容设备接口：耗时小于 10 秒，单批最大 50 台
- 重启设备接口：耗时小于 10 秒

4.3 操作使用层

为方便用户使用，系统提供了 UI 交互界面和 CLI 命令行模式，以下是命令行操作示例：

获取设备列表、IP
$ avd devices
设备访问连接
$ adb connect device_ip_address：5555
注：device_ip_address 如上文所述由 paas 平台在创建 AVD Docker 主机时分配，该 IP 在测试环境和办公环境均可访问。
确认主机已连接到目标设备：
$ adb devices
List of devices attached
device_ip_address:5555 device

五、AVD Iaas 高可用保障

AVD 测试设备作为 Iaas 基础设施，稳定性是非常重要的指标，要能够为用户提供 7x24 的稳定保障，我们基于以下几个维度系统设计了保障策略。

1）API 服务架构层面

- 避免单点服务，多机部署
- 防止服务之间相互干扰，重要服务单独部署
- 防止数据库异常导致服务不可用，增加缓存处理

2）运维层面

- 通用指标监控：CPU、内存、API 请求量、响应时间、错误数
- 业务指标监控：自动化使用设备量、可用设备池库存、设备申请失败率
- 接入携程内部的报警系统，故障分钟级别响应

3）代码层面

- 保证代码异常不会导致服务挂掉
- 保证服务是无状态的，可以支持水平扩展

4）设备弹性调度

- 防止设备不足导致任务积压排队，系统会监控任务队列情况自动扩容

- 防止任务低谷下的设备大量闲置，系统会监控任务队列情况自动缩容

5）设备自动维护

- 防止设备被长期空闲占用，系统针对设备的使用情况进行定期检测，一段时间内未使用的设备，会自动收回到可用设备池，并通过 IM 消息通知到用户
- 根据设备监控报警，自动拉出异常设备重启维护，确认正常后加入设备池

综合以上几点，稳定性较原先真机设备大幅提升，满足 7x24 小时使用的需求，SLA 指标达到 99%，我们也在持续努力，继续提升。

5.1 遇到的问题

由于 ARM 编译 APP 在 X86 架构 Node 运行时，会将 ARM 指令都转换成 x86 指令，造成较高的性能负荷，因此与基于 x86 编译的 APP 相比，ARM 编译 APP 在 x86 宿主机上的运行速度会慢很多，而且它还无法使用 x86 处理器提供的硬件加速和 CPU 虚拟化技术。为了保障应用的执行性能，我们的最初方案是将测试应用 APP 编译为 X86 模式，这样可以减少 Android 系统指令转换的性能开销。

随着规模的逐渐发展和更多用户场景的提出，这套方案也逐渐暴露出了一些问题：

- 一些 APP 不支持 x86 编译
- 编译为 x86 后，少量场景运行时，底层 so 文件会出现异常，而同样的场景下，使用 ARM 编译的 APP 却没有问题

综合以上两个问题，我们开始寻找优化方案，可以更好的支持 ARM 架构编译的 APP 应用。

5.2 问题解决方案

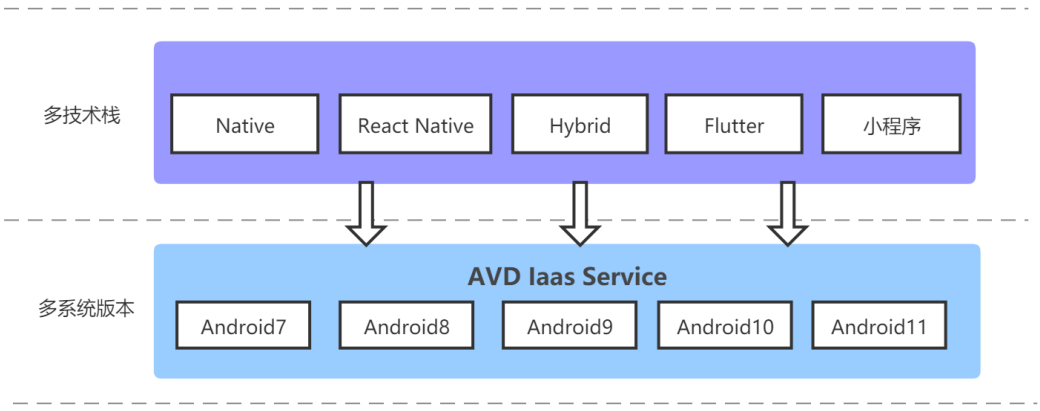
Google 官方在 2020 年开始推出 Android11，新版本带来了新特性。引入 Android11，可以解决 ARM 编译 APP 的性能问题。

全新的 Android 11 系统映像与 ARM 兼容，它不仅允许整个系统在本机运行 x86 指令，而且还可以照常使用虚拟化技术。当应用的某个进程需要使用 ARM 二进制代码时，代码仅会在该进程内被转换成 x86 指令，其余进程将继续在 x86 环境内执行，包括 Android 运行时 (ART) 以及其它性能关键库，例如 libGLES 和 libvulkan。除此以外，指令转换器也不会执行底层的硬件特定库，从而避免高成本的内存访问检测和相应的性能影响。经过测试，在 X86 服务器上基于 Android11 运行 ARM 架构 APP，性能确实比之前版本提升很多，因此我们引入 Android11，用户可根据 APP 编译类型选择合适的 AVD 容器。

六、AVD Iaas 服务中台化

2020 年携程无线公共团队提出建设无线 CTest 测试中台的目标，AVD Iaas 作为底层基础设施方案，也加入 CTest 中台提供给携程各事业部使用。目前已经有 15 个事业部接入，总使

用次数超过 10000+，满足公司内多版本、多技术栈的测试任务执行和兼容性验证需求。



6.1 大规模无线 UI 自动化集成测试应用

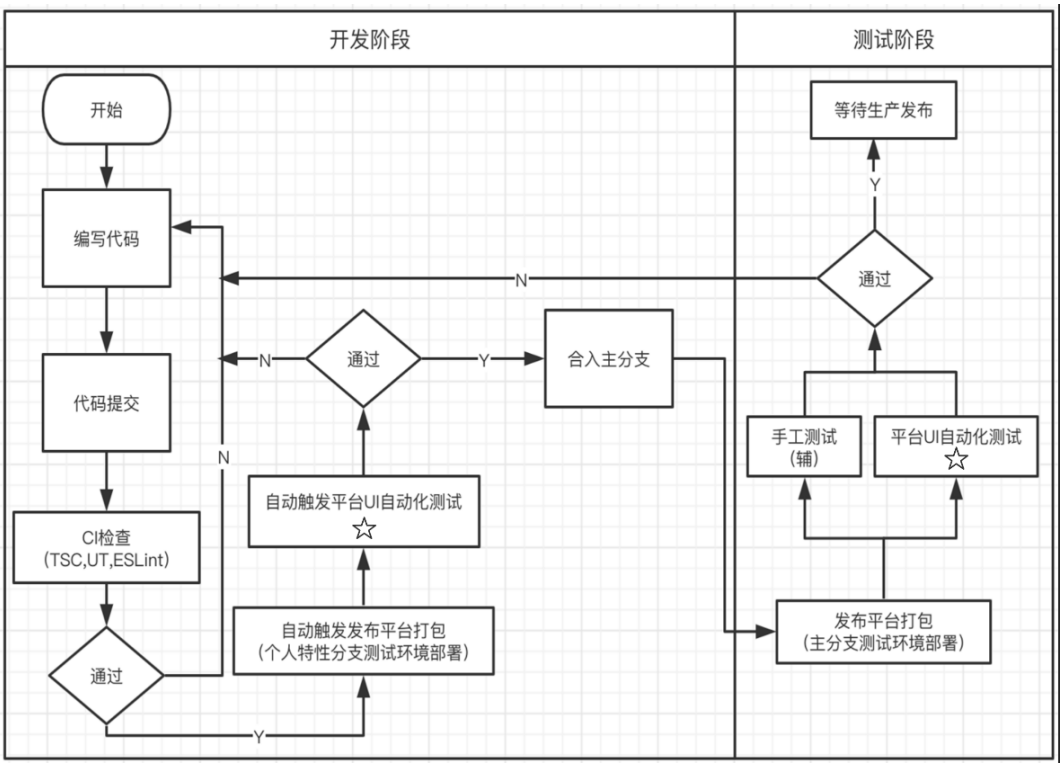
无线 UI 自动化集成测试，是 APP 应用持续交付过程中绕不开的关键环节。过去在每个月交互 1 个版本的模式下，还能够通过增加手工测试人力勉强满足版本的验证需求，随着交付节奏的不断加快，携程内部的交付频率不断提速，如机票前台研发团队，期望提速为每周交付 2 次。基于此大背景下，自动化集成测试的效率是必须要解决的问题。

下图为携程机票前台研发团队的 CICD 流程图。

通过在特性分支和发布分支高频触发自动化测试，并设置对应的质量卡口，大部分的测试工作交由自动化完成，帮助快速发现问题，减少开发修复成本，这对自动化集成测试提出了很高的性能要求。

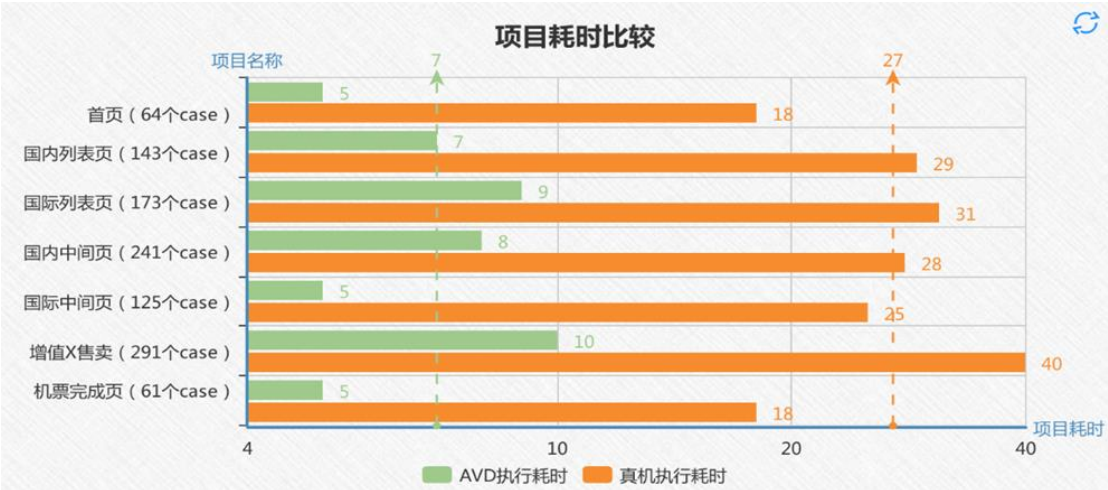
通过应用 AVD Iaas，基于 AVD 容器设备的快速扩缩容能力，在项目测试开始时，系统会根据项目的 case 数量，动态创建、分配测试设备，保证单个项目的 UI 自动化集成可以在几分钟内完成。使用结束后，自动缩容回收。

最终较好的支撑了机票前台研发团队每周发布 2 个版本，业务快速上线的要求。



下图为使用 AVD Iaas 设备后的测试执行耗时对比。

通过按需动态扩缩容，缓解任务排队瓶颈，提高并发能力，测试用例执行耗时平均降低 74%。



6.2 接入 AVD Iaas 时遇到的典型问题

在接入 AVD Iaas 的过程中，部分团队出现了 UI 自动化 case 执行通过率降低的现象。通过分析，发现主要是 case 代码不规范导致，比如：操作等待时长、滑动距离等操作存在硬编码，导致 case 代码只能在固定环境中执行通过。

为此团队总结了自动化代码规范，帮助接入方优化 case 代码，更好的支持不同设备环境下

的稳定执行，这里就不再详细展开了。

七、总结

目前 AVD laas 系统已经支撑了携程绝大部分业务线在不同场景下的移动端自动化测试设备需求。我们一直在努力丰富 AVD 容器设备的功能场景，不断提升系统稳定性和性能，此外我们也在积极构建 BDD 测试执行框架、用户流量回放等自研的研发工具，通过和 AVD laas 形成组合拳，解锁研发活动中更多的适用场景，帮助业务团队更好的提升能效。

云计算

携程酒店 AWS 实践

【作者简介】微末，携程软件技术专家，关注系统架构，致力于高可用高性能的支撑业务系统开发。

一、背景

随着携程海外酒店业务的发展，遍布全球的海外供应商与携程总部 IDC 之间的数据传输量快速增长。技术上，这种日益增长的数据量对跨境网络专线的带宽、延迟等提出了更高的要求；业务上，由于当前有限的跨境网络专线资源对业务处理效率及用户体验也造成了一定的影响；成本上，跨境网络专线作为一种昂贵的资源，通过单纯的专线扩容又会给 IT 成本造成巨大压力。所以我们开始思考是否可以通过公有云结合酒店直连的业务特性来解决日益增长的带宽压力和供应商接口延迟的问题。

酒店直连系统主要是使用自动化接口实现供应商或集团与携程之间的系统对接，实现静态信息、动态信息、订单功能等都通过系统的方式流转交互。目前携程大量海外酒店业务是通过酒店直连系统对接。

本文将主要从携程酒店直连服务迁移部署至 AWS 过程中所进行的应用架构调整及云原生改造，使用 AWS 后取得的技术和业务收益，在部署过程中对 EKS（Amazon Elastic Kubernates Service）、DNS 查询延时和跨 AZ 流量降低所做的成本优化等几方面进行详细介绍。

二、痛点

携程酒店海外直连对接了上千家海外供应商，所有的接口访问都通过代理出去（见图 1），由于酒店直连的业务特性，当一个用户请求过来时会根据人数、国籍、会员非会员等裂变成多个请求，最多的时候可能一个请求会裂变成数十个请求，而且请求报文十分巨大（通常为几十 Kb 到上百 Kb 不等），虽然我们可能只需要返回报文中的一小部分信息，但是因为目前架构的限制只能将所有报文全部请求回来再处理，这无疑浪费了大量的带宽。

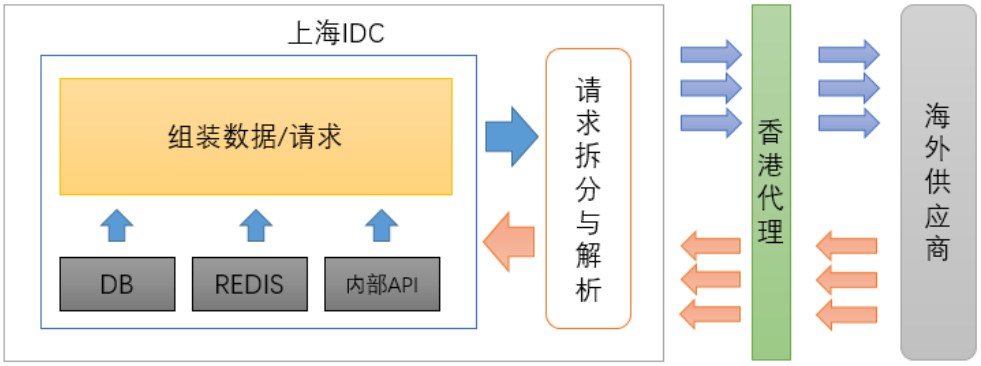


图 1

同时因为供应商遍布全球，所有的请求/响应都需要经过集团的代理出口，导致了部分供应商接口响应受到物理距离的影响延迟变高了，会降低用户的体验。

三、云服务选择及初步方案

本次核心目标之一是为了提高对接全球供应商的网络传输能力和延时改进，提升用户体验，必须选择一个在全球有广泛资源分布的云厂商帮助携程尽量靠近供应商访问数据。经过与多个公有云厂商的多轮交流，综合考虑各厂商技术水平、服务能力、成本价格等多方面因素，我们认为 AWS 无论是在全球覆盖及网络能力（见图 2）（AWS 在全球分布的 25 个区域和 80 个可用区提供广泛的服务能力，同时数据中心通过其骨干网互联，提升了未来不同数据中心的数据互访能力），云服务的先进性和成熟度、现场团队的服务能力、响应时间、专业水平都具有明显的优势，最终我们选择 AWS 作为资源部署的云厂商合作伙伴。

图 2

为了更好地与云上资源使用集成，我们采用 IDC 的容器化部署方案，最终考虑到托管容器平台的高可用性设计及 SLA 保证，及对社区的兼容性，使用 AWS 托管容器平台 EKS 作为部署的平台。

资源方面我们对服务进行改造后，大量使用竞价实例作为 EKS 工作节点，大幅降低成本并提高效率。

同时利用公有云的网络和平台优势，将原本部署在携程总部 IDC 的相应业务服务部署到离供应商距离更近的海外公有云站点，实现携程与海外供应商之间高可靠、低延迟的网络直连，并将部分数据预处理逻辑剥离出来前置部署到海外公有云上，实现仅将经过处理的有价值的数据（而非原始、全量的裸数据）压缩后再传输到携程总部数据中心，进而达到降低对跨境网络专线的压力、提升业务数据处理效率、降低成本、优化用户体验等目标。

四、酒店直连上云经验

4.1 云业务应用的云原生改造

为了充分的使用云服务带来的便利和成本优化，经过调研分析，我们如果直接将应用迁移至公有云上，虽然业务上会产生相应的价值，但成本会相对较高，因此我们对酒店直连服务进行了相应的云原生架构优化，相关的主要调整如下：

1 ）访问供应商模块上云

要节省带宽需要减少通过从代理出去的请求同时减少每个请求的报文大小。我们的做法是将请求拆分的逻辑搬到 AWS 上，这样每次一个用户请求过来通过代理出去只有一次请求/响应。同时我们在 AWS 上将供应商返回的报文中无用属性剔除，然后再根据业务属性合并相关节点最后再压缩返回，这样就达到了缩减报文大小的目的(见图 3)。从目前运行的数据上看，整个代理的带宽流量只用到了之前的 30%~40%。

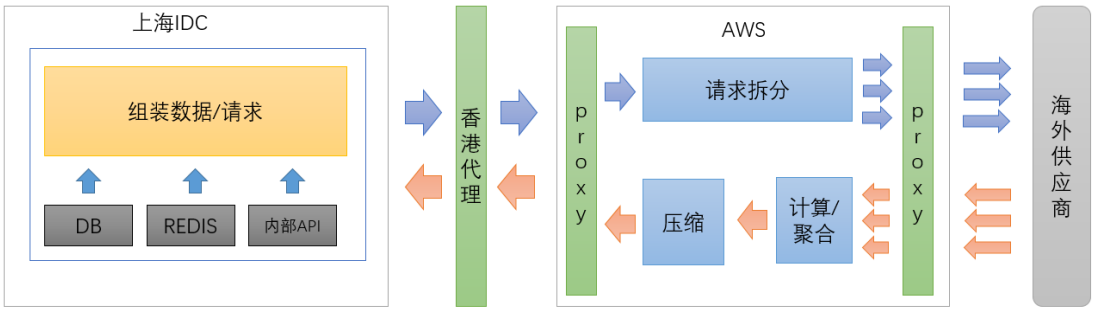


图 3

公有云厂商普遍采用按流量收费的价格策略，在设计网络出入站网络访问的技术方案过程中，默认情况下会使用 AWS NAT 网关，这样网络流量费用相对较高。考虑到酒店直连请求有个特性，通常情况下请求报文不到 1K，而响应报文平均有 10k 到 100K，利用这个特点，我们在 AWS 上采用了基于 EKS 自建 Squid 代理方案（见图 4），这样只有出站的请求报文会产生流量费用，而大量入站的响应报文不收费，从而大大降低 AWS 上产生的网络流量费用。

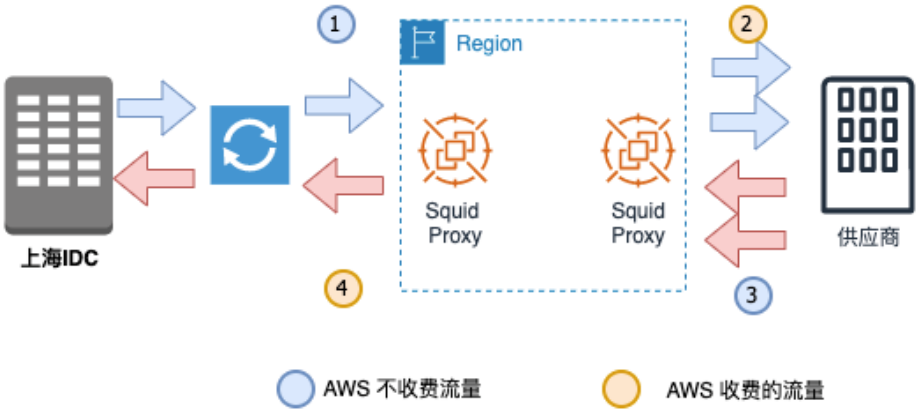


图 4

2）降低网络延迟，利用 AWS 全球数据中心对供应商就近访问

很多海外的供应商服务部署在全球各地，而我们所有的海外访问都统一从代理出去，这样一些服务器部署较远的供应商因为物理距离上的原因导致网络延迟很高。通过 AWS 的在全球各地的数据中心，我们可以将服务就近部署在供应商机房附近，同时利用 AWS 的骨干网络降低各数据中心到代理所在地附近的 AWS 数据中心的延迟，最后通过专线连接该 AWS 数据中心与携程 IDC（见图 5），整个过程对那些因物理距离对网络延迟影响较大的供应商性能提升较明显，最多可降低 50%的响应时间。

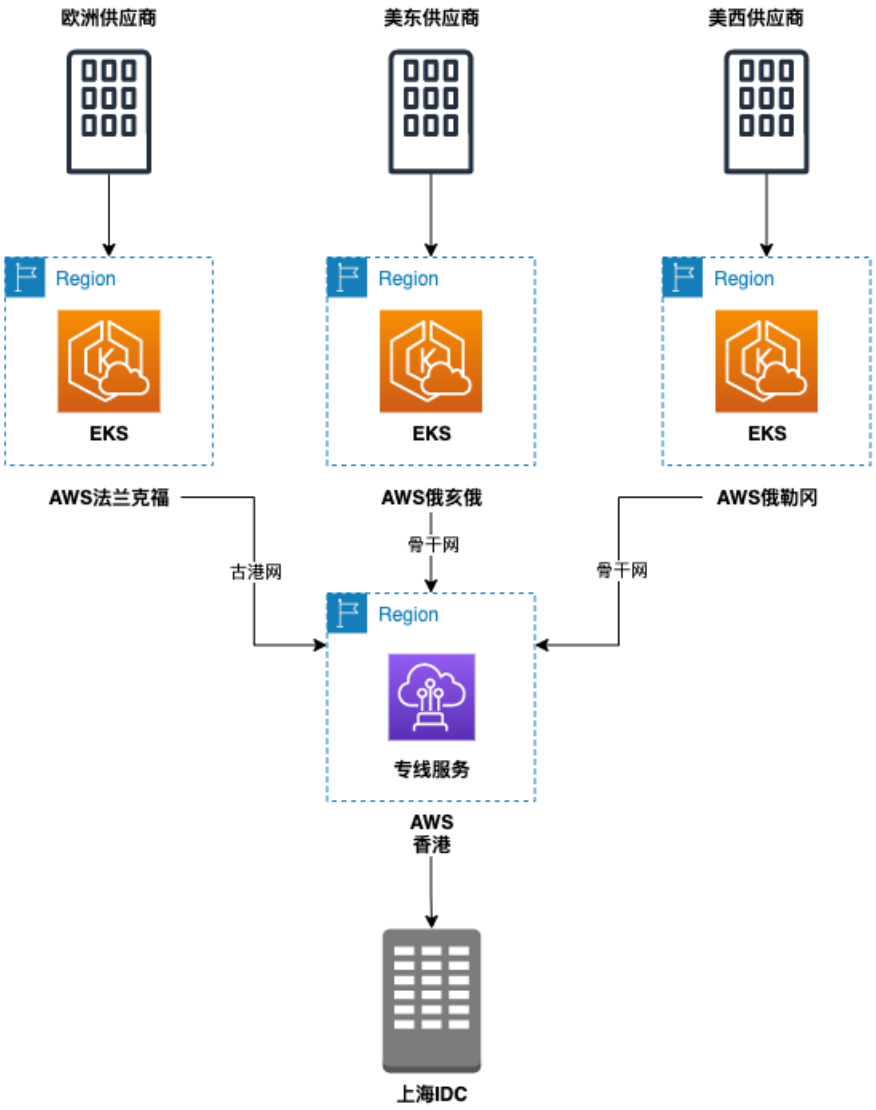


图 5

4.2 持续的架构改造和性能及成本优化

在目前的方案中，我们为了上云单独开发了一套全新的应用，这样带来的问题就是，当有业务变更时我们同时需要调整携程 IDC 和 AWS 上部署的两个应用，提高了系统维护成本。主要原因是原应用中大量依赖携程的基础组件，本次上云尝试使用的是完全独立的账号和 VPC 网络，如果在云上同样部署一套不太现实，一是成本太大，二是一些敏感数据不能放在在云端存储，所以后续我们会对适配器架构再进行优化，在不依赖携程基础组件的情况下复用一

套应用以适应不同的云环境。

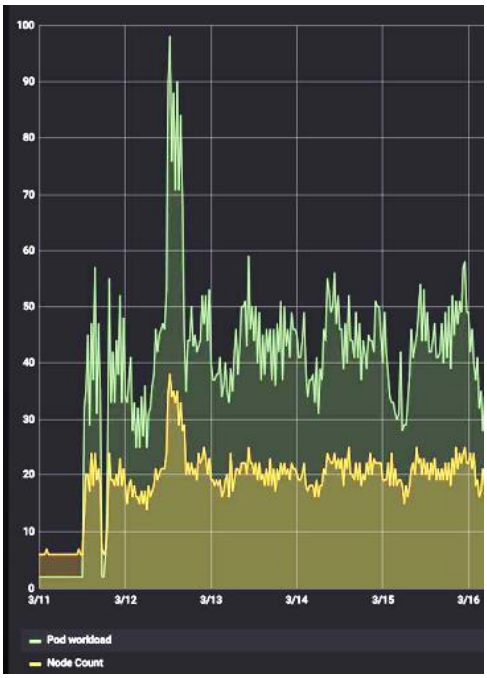
业务上线后为了验证未来更大规模的负载上云的可能性，我们同时也在对性能，成本，高可用性方面做持续不断的优化

4.2.1 利用云弹性伸缩能力

以计算资源成本为例：计算实例成本 = 实例运行时长 * 实例价格。如果只是简单粗暴把本地机房的运行模式套用到云上计算，云服务计算资源的费用是高于本地机房的。所以我们需要充分利用云上按需收费的特性，减少闲置资源成本。实例的运行时长和 Kubernetes 集群内的服务数量，以及分配给这些服务的计算资源成正比，同时服务的数量又是和流量成正比。

酒店直连业务场景存在不可预测的业务流量，比如临近节假日颁布的旅游政策，或者营销直播活动。云原生的弹性特性很好地利用合理的资源应对突发的流量。

Kubernetes 的 HPA 弹性架构会实时采集集群整体的负载指标，判断是否满足弹性伸缩条件和执行 pod 的伸缩。仅仅是 pod 的伸缩还不够，我们还需要在集群中使用 Cluster Autoscaler 组件，监控集群中由于资源分配不足无法被正常调度的 pod，自动从云平台的实例池中申请增加节点，同时在流量下降的时候，Cluster Autoscaler 组件也会检测集群中资源利用率较低的节点，将其中的 pod 调度到其他可用节点上，回收这部分闲置节点。



弹性伸缩案例

云原生的弹性特性不仅帮助减少资源使用成本，也提高服务对基础架构故障的容错率，在基础设施部分可用区中断不可用期间，其他可用区域会增加相应数量的节点继续保持整个集群的可用。

Kubernetes 支持对 pod 容器所需的 CPU 和内存调整，找到一个合理的配额以合理的成本达到最佳的性能。所以我们在服务上云前会做一些接近真实环境的负载测试，观察业务流量的变化对集群性能的影响（业务周期性高峰和低峰的资源使用率，服务的资源瓶颈，合适的余量资源 buffer 应对尖刺流量等等）。既不会因为实际利用率过高导致稳定性问题，比如 OOM 或者频繁的 CPU throttling，也不会因为过低浪费资源（毕竟，即使你的应用只使用了实例的 1%，也要支付该实例 100%的费用）。

4.2.2 采用公有云竞价实例

某些云平台会把一些闲置计算资源作为竞价实例，以比按需实例更低的定价出租，顾名思义竞价实例的最终费用是按市场供需出价决定的。按照我们实际使用的体验，如果不是特别热门的机型定价基本在按需实例费用的 10-30%左右。低价的竞价实例自然有它的限制，云平台会可能会调整竞价实例池的资源比例回收部分实例，一般回收的概率根据统计通常<3%，同时在回收前会提前 2 分钟通知到这些实例。我们通过 AWS 提供的 Terminal handler 组件在收到回收通知后提前把容器调度到其他可用的实例上，减少了资源回收对服务的影响。下图是某云对竞价实例的资源池划分，我们可以看到，即使相同的实例资源，在不同的可用区也是独立的资源池。

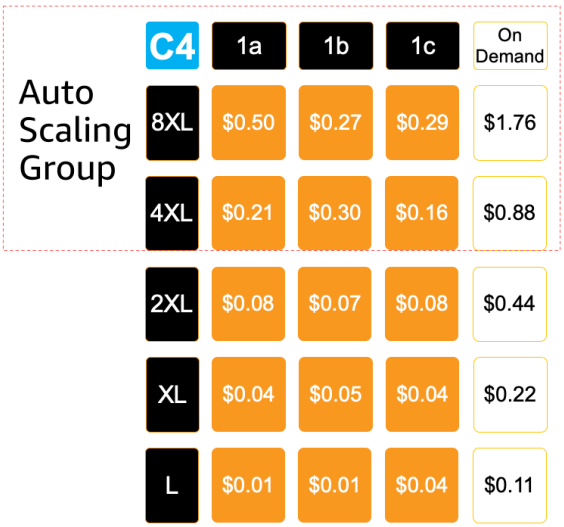


图 6

为了能最大限度减少竞价实例的中断影响，包括实例在多可用区的再平衡影响，我们在通过 ASG（AWS auto scaling Group 弹性扩展组）选择不同实例类型的情况下还将不同的实例资源池独立使用 ASG 进行管理，这样保证了资源的最大利用效率。

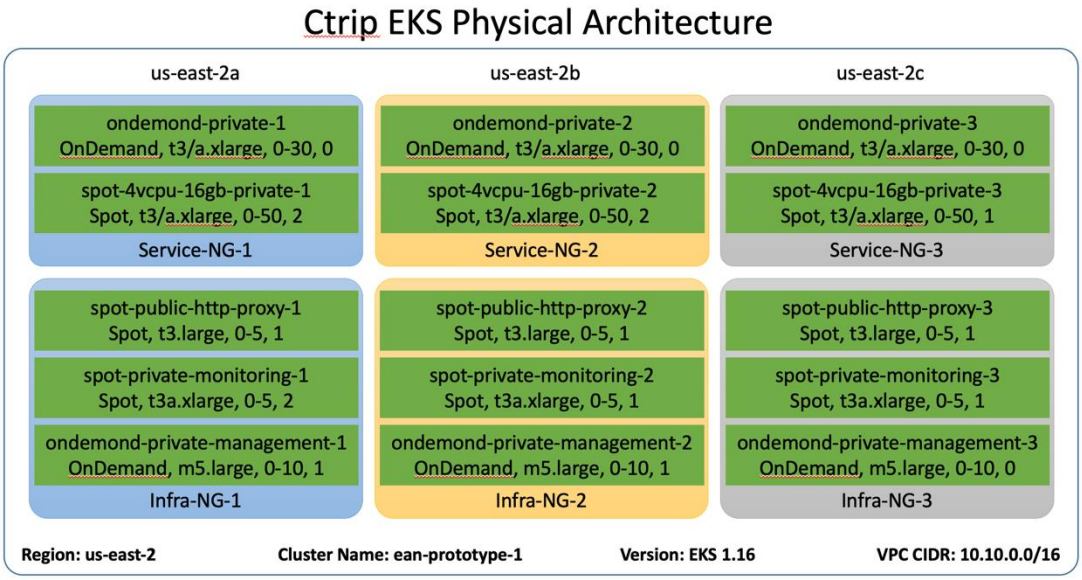


图 7

携程酒店直连使用按需实例和竞价实例的混合部署，保证低成本和高可用。一些系统关键组件（比如 Cluster Autoscaler），中断就会丢失数据的有状态服务（比如 Prometheus）运行在按需实例。而对错误容忍度高，使用灵活无状态的业务应用运行在竞价实例上。通过 kubernetes 的节点亲和性控制不同类型的服务调度到对应类型标签的实例上。（见图 8）

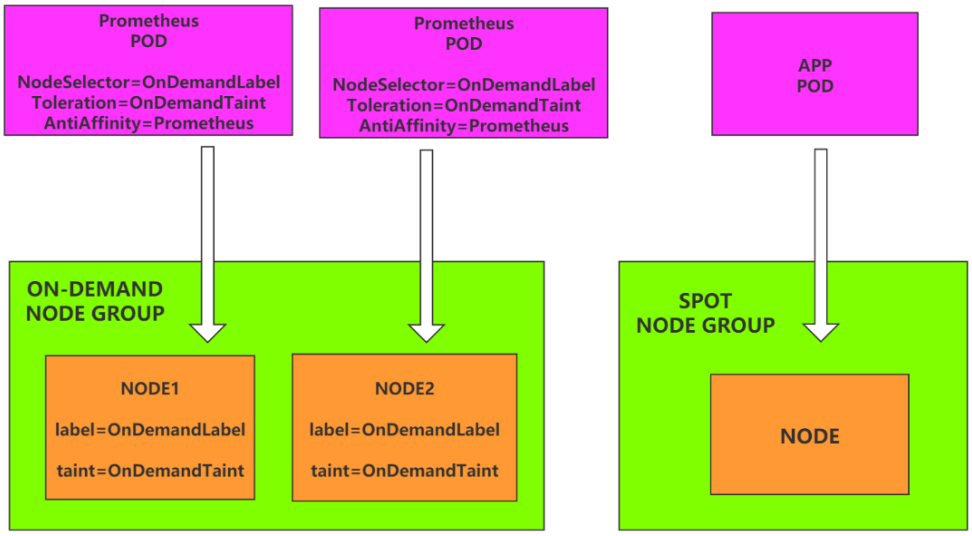


图 8

通过 kubernates 原生的 HPA 和 ClusterAutoscaler 组件结合 AWS ASG 及竞价资源的充分利用，可以将成本降低 50%-80%。

4.2.3 DNS 解析性能优化

当服务规模逐渐增大的时候，我们发现服务间的调用延时明显上升，平均达到 1.5S，高峰达

到 2.5 秒，经过分析发现，主要是因为 DNS 解析负载过高造成的性能解析瓶颈，最终我们采用社区比较主流的 localdns 方式，对热点解析域名做本地缓存，来降低对核心 DNS 频繁的解析请求从而提高性能：

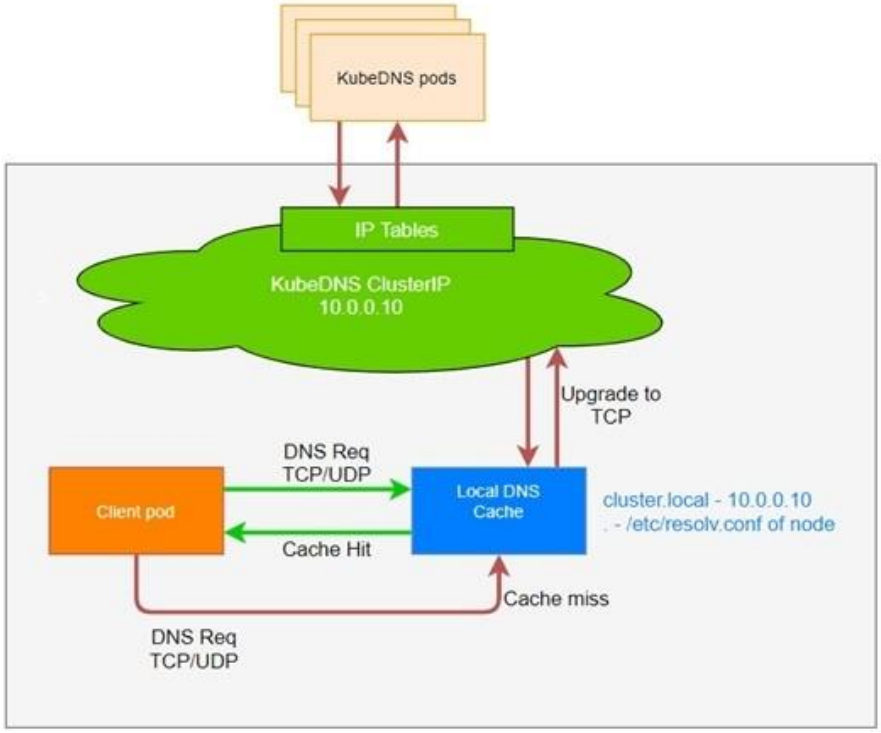
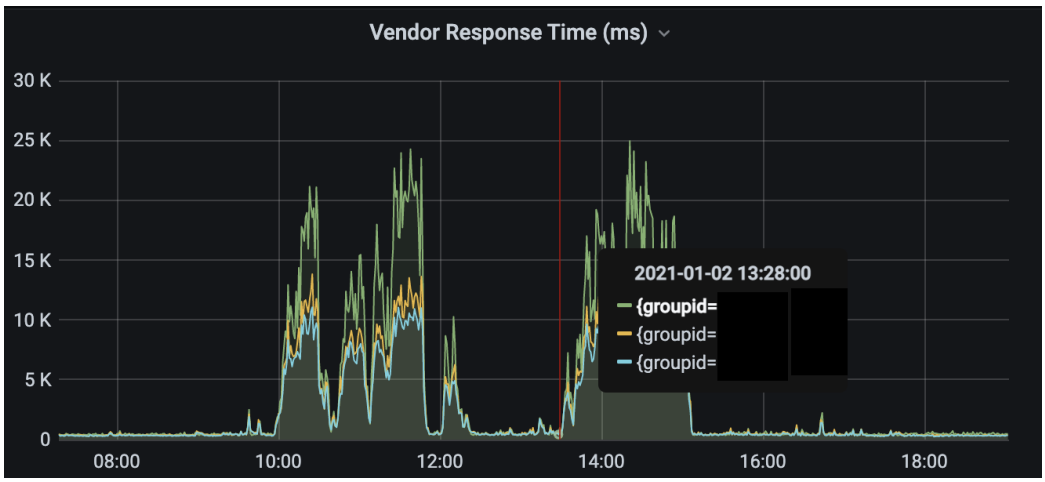


图 9

如图 9 所示，在每个 Node 部署基于 DaemonSet 的 NodeLocal DNSCache，通过 Node LocalDNS 缓解 CoreDNS 服务的 DNS 查询压力，LocalDNS Cache 会监听所在的 node 上每个 Client Pod 的 DNS 解析请求，通过本地的解析行为配置，Local DNS Cache 会尝试先通过缓存解析请求，如果未命中则去 CoreDNS 查询解析结果并缓存为下一次本地解析请求使用。

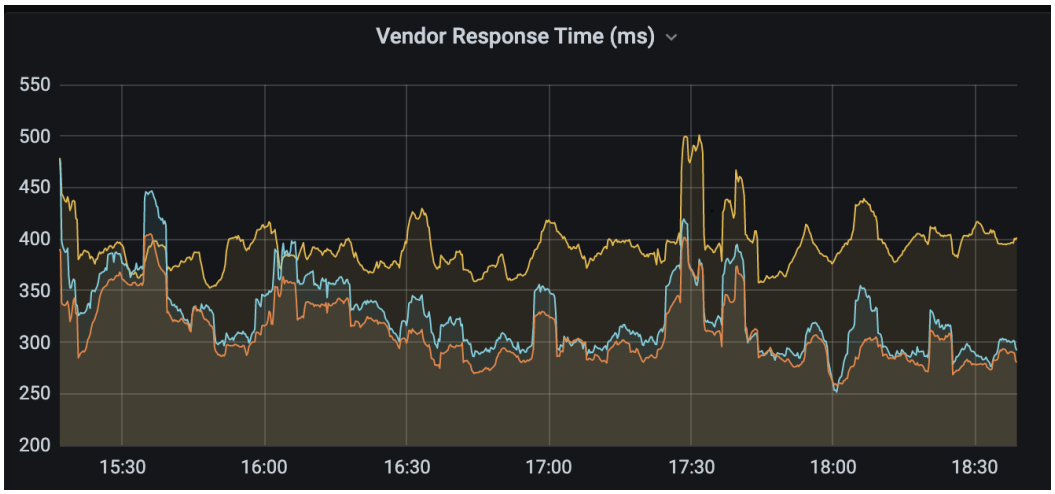
如下图，通过使用 LocalDNS 方案我们将高峰的延时从 2.5S 降低到 300ms，缩短了 80%的响应时间：

未使用 LocalDNS 前，平均响应在 1.5-2.5S。



未优化前

使用 LocalDNS 方案后，响应请求降低到 300-400ms，延时优化了 80%。



优化后

4.2.4 公有云跨可用区流量优化

在使用竞价实例对资源进行大幅优化后，我们注意到跨可用区的流量在服务大幅扩展后占比非常高（60%），这是因为在服务之间调用时，我们将服务单元部署到不同可用区，最大限度提高服务的可用性，同时带来的问题是服务间大量的流量交互带来了跨可用区的流量费用（见图 10）。

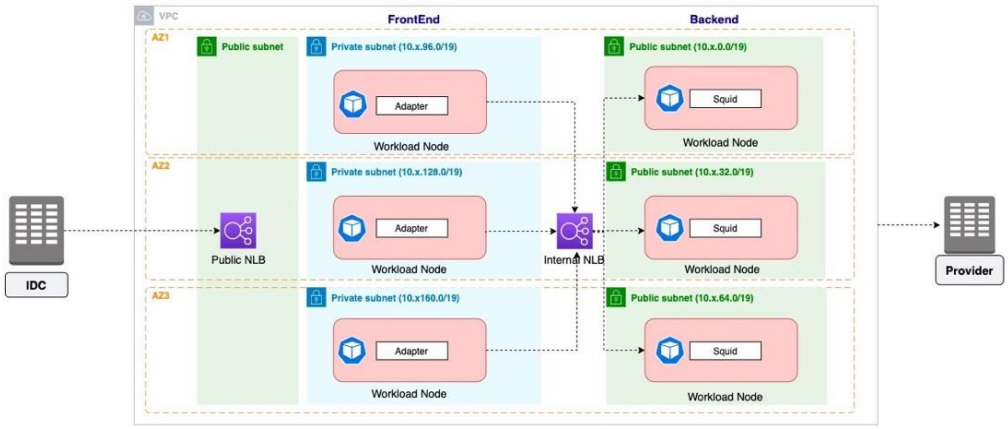


图 10

但是为了整个系统的高可用性，我们并不想将服务部署在单可用区，降低服务 SLA。我们需要降低跨可用区流量的同时保证服务的高可用性。

经过不同的方案调研最终我们使用 AWS NLB 来暴露服务，通过 NLB 的 disable cross-az 功能，对同可用区的上下游服务进行流量可用区管控。同时使用之前提到的 local dns 组件，将上游服务访问 NLB 不同可用区的域名解析进行固化，保证了上下游的服务流量只能在可用区内部进行互通。改造后如下图：

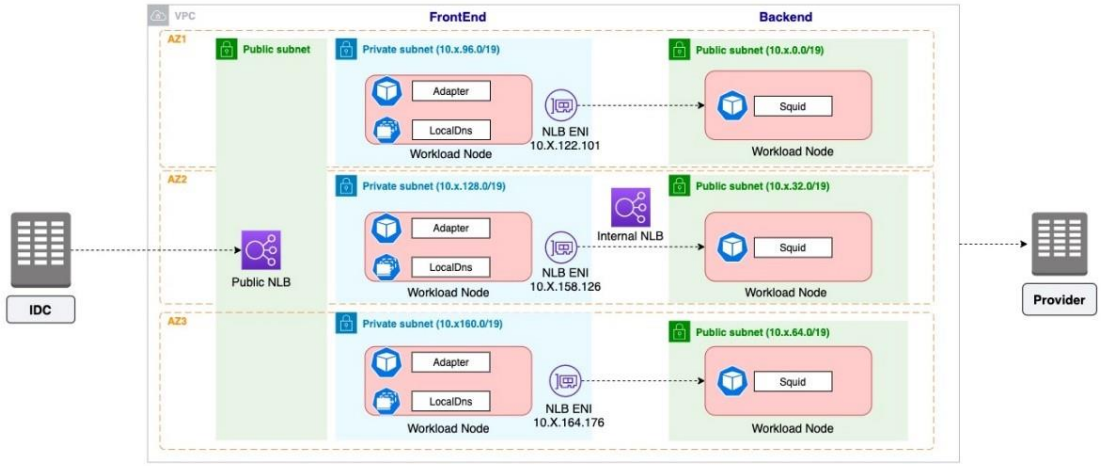


图 11

后段服务因为会通过 K8s 的 Kube-proxy 进行转发造成跨可用区跨节点，我们选择使用 externalTrafficPolicy 本地策略，将转发流量固化在本地节点的服务上，但是同时本地转发策略也带来了一些问题（见图 12）：

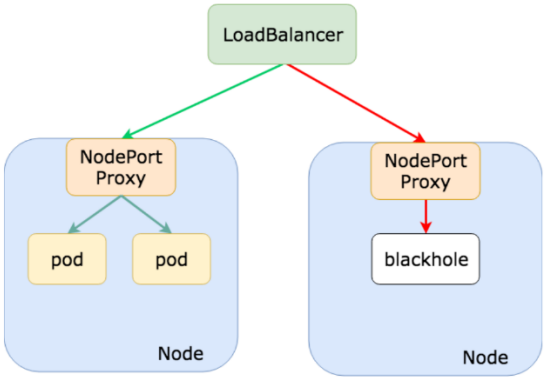


图 12

如上图所示，本地转发策略可能因为后端服务分布不均衡导致了流量黑洞和服务负载的不均衡，所以在这个基础上，我们利用 EKS 弹性扩展组策略对底层节点资源均衡分布到不同的可用区，同时利用 K8s 反亲和性策略，将服务尽量分布到不同可用区的节点上，最大程度的保证了流量的均衡性，同时保证了服务的跨可用区部署的高可用性。

优化后跨可用区流量降低了 95.4%。（见图 13）

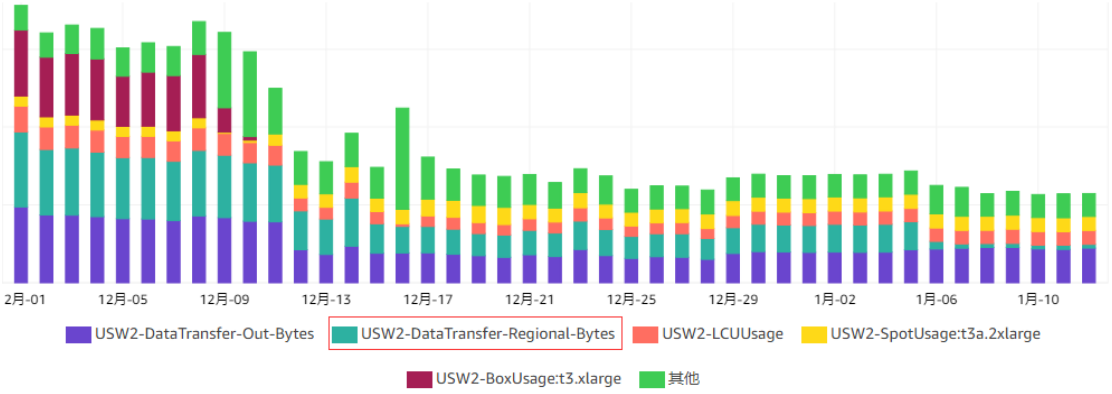


图 13

五、后续的优化改进方向

目前的架构虽然解决了我们业务上的一些问题，但还是有一些不足之处可以改进。为了可以就近访问供应商，我们使用了一个独立的 VPC 网络来部署和测试我们的集群，所以需要单独在云端部署相关的存储依赖以及日志监控组件，这样无疑增加了运维的难度以及服务在不同云上的迁移难度。

在最新的架构设计中针对这个问题我们计划做如下改造，首先将需要在云端计算并且依赖持久化存储数据的功能迁移回携程 IDC，这样这部分数据就不用再传到云端。其次因为公司在 AWS 的其他数据中心已经有一套成熟的环境，所以我们只需要配合 OPS 打通两个 AWS 数据中心之间的 VPC 网络，便可使用公司的日志和监控框架，减少运维成本。

六、总结

本文通过携程酒店直连在云原生的实践，分享了如何快速在云上搭建一套稳定高效的生产环境实现快速交付、智能弹性，以及在云上的一些成本优化经验。借助云原生体系实现了基础设施自动化，释放一部分的运维工作，可以更多地投入到业务迭代，更敏捷地响应业务需求迭代，通过监控和日志实现快速试错和反馈。希望借此能帮助到更多想上云的团队，少走弯路，拥抱云原生带来的好处。

如何构建系统优化成本，携程出海云原生实践

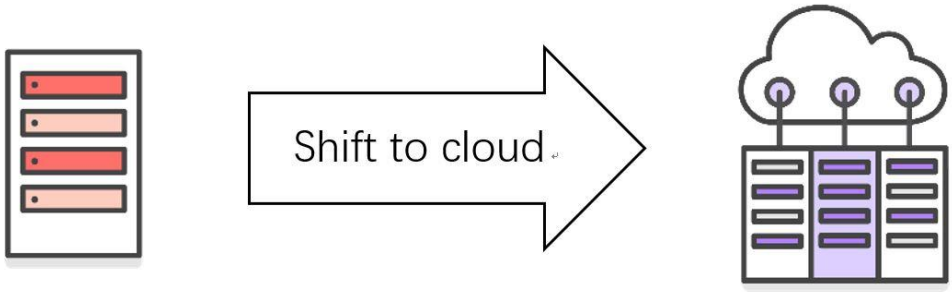
【作者简介】马孟起，携程后端开发专家，对操作系统和网络技术有浓厚兴趣。

一、背景

随着携程国际化战略的开展，为了给海外用户提供更好的服务，携程国际机票有很大一部分数据来源于世界各地的海外供应商和平台，在美国、德国、新加坡等全球众多的海外站点部署业务服务。相比自建私有云，购买设备和组建运维团队，公有云是企业出海的一个更好选择。

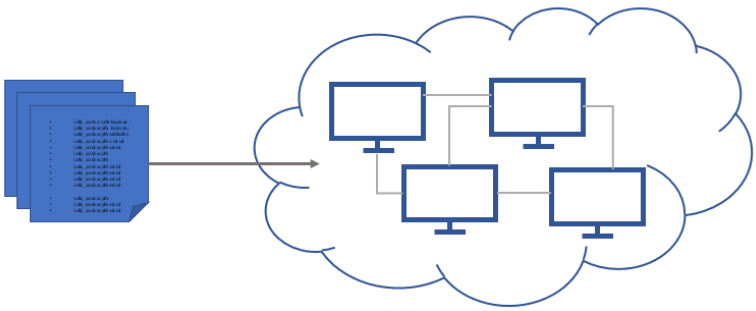
但该怎么上云，如何充分利用云原生的能力，是每个企业服务上云前面临的问题，其实我们可以遵循在业界已有的成熟云原生标准。所谓的云原生是一组最佳实践和方法论，指导我们在云环境下构建可伸缩、高可用、松耦合的应用，更快速和低成本运行服务，享受它带来的红利。

云原生本身是一个非常宽泛的概念，本文主要分享携程国际机票在上云实践中关于构建系统和成本优化的一些经验。



二、上云经验

2.1 基础设施即代码



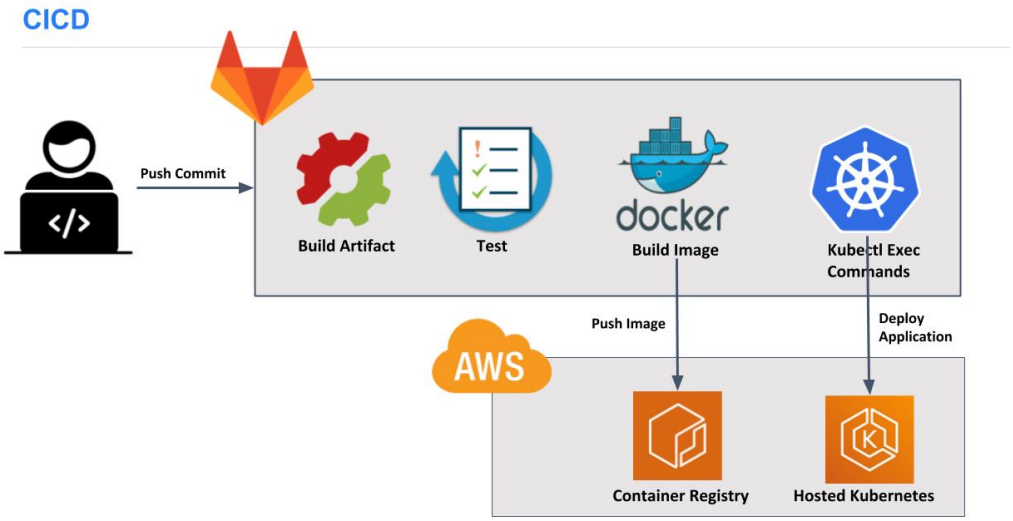
要让我们的应用能够在云上部署，首先需要构建基于云原生的基础设施。传统的手工构建费时费力，虽然可以通过一些自动化脚本取代手工操作实现构建，但是如果整个基础设施的构建过程逻辑复杂，会带来脚本的维护工作量和排查调试等新的问题。

在云原生背景下，我们遵循 Infrastructure as Code（基础设施即代码），应用除了自有的代码仓库外，还有对应的基础设施 IAC 仓库，包含了应用依赖的基础设施和周边系统的配置文件。引入版本仓库控制后，团队可以像管理自己的代码一样管理基础设施，包括追溯到任意版本分支复现问题，一致性的环境配置，所有对基础设施的变更也更容易追踪和 review。

另外我们还需要类似 Terraform 的自动化编排构建基础设施的工具帮助管理基础设施的生命周期。Terraform 对基础设施编码的声明式配置思想和 Kubernetes 如出一辙，我们只需要在配置文件里描述期望的基础设施配置，避免了复杂的过程命令式脚本开发维护，剩下的编排构建工作就交给 Terraform 实现。此外声明式的配置文件有更好的可读性，简单的最终结果一目了然。

对于基础设施系统，秉承让工程师专注于自己最擅长的任务 —— 为客户提供业务创新和可行的解决方案，摆脱繁重的运维工作，我们会倾向于尽可能选择托管版的 Kubernetes 服务。根据我们的实际经验，搭建一个生产级别可用的 Kubernetes 集群整个耗时在分钟级别。

基础设施作为代码后，就可以集成在 CI/CD 流水线里，实现自动化持续部署。无论是基础设施还是业务应用的 CI/CD 流水线，出于公司安全合规的考虑，从工程师提交变更，编译，自动化测试，打包镜像这些和 CI 相关的步骤在公司内网执行；之后的镜像会通过代理推送到相应云平台的私有仓库，通过类似 kubectl、Terraform 等一些编排工具的客户端发送请求到云端，远程执行 CD 部分的工作。



业务代码的 CI/CD 流水线

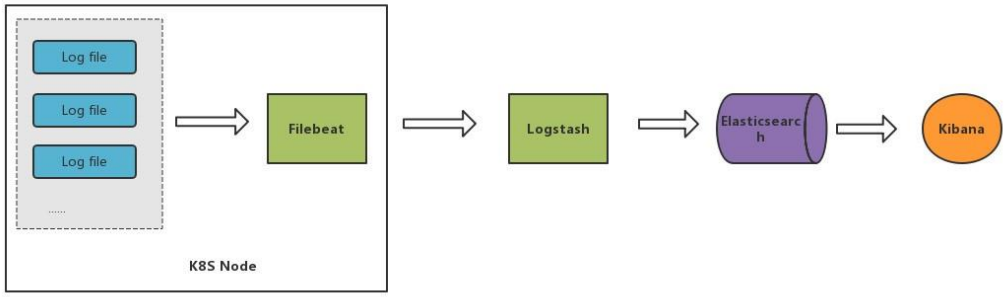
2.2 日志

基础设施之上除了运行业务应用外，还有支撑业务服务的周边系统，例如日志和监控。

日志系统同样使用了托管版的 Elastic Search，简化诸如硬件预置、软件安装和修补、故障恢复、备份和监控等管理任务，同时提供了多个可用区的高可用性。

上云后，应用可能会由于弹性伸缩经常会调度到不同的计算节点，伴随着调度的切换，本地的日志也会被销毁，因此需要实时把日志采集到统一的存储服务中，同时要求日志采集功能具备扩展性和适配性。

业务应用内部直接推送日志的方案虽然简单，但应用和日志收集功能互相耦合，对日志功能的维护带来不便，同时日志的采集开销也会影响到应用自身的运行。我们采用以 Daemon Set 方式在每个计算节点上部署日志采集代理，收集该节点上所有应用日志，随后按自定义的规则加工处理日志数据，输出到 Elastic Search，在前端 Kibana 展现。应用只需要简单地将日志发送到 stdout 和 stderr，完全解耦日志功能和业务应用。



2.3 监控

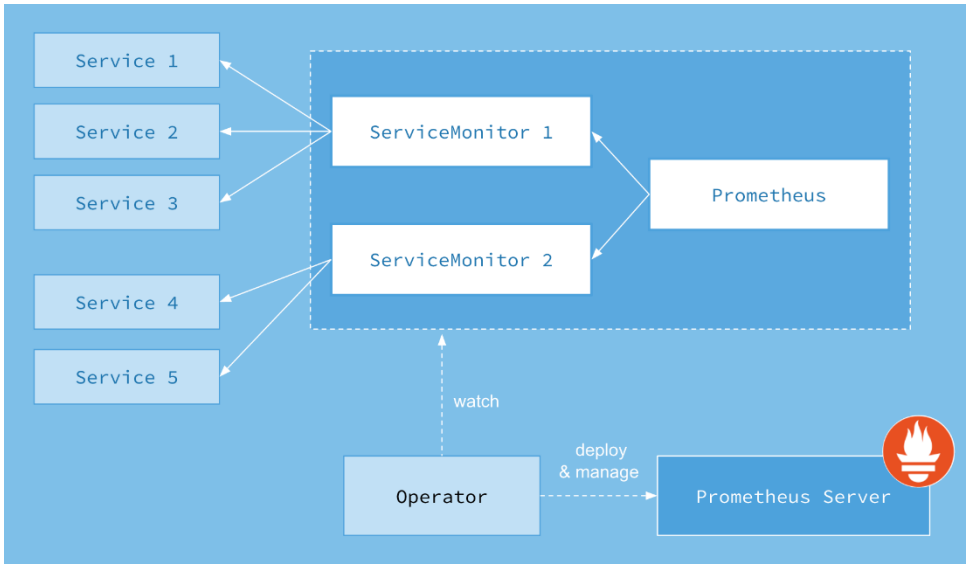
由于之前公有云平台并没有提供托管的监控系统，所以我们只能自己来搭建。监控系统选型几乎没有什么纠结，使用了几乎是云原生领域监控事实标准的 Prometheus + Grafana。但是原生的 Prometheus 本身只支持单机部署，存在单点故障，同时存储空间受限于本地磁盘，随着时间的推移，监控数据的增加，单机 Prometheus 的性能终究会达到瓶颈。

Prometheus 官方建议根据服务维度拆分，携程本身在上云前已经实现了微服务架构，单个 Prometheus 实例配置成仅采集指定的几个微服务指标，在一定程度上实现了水平扩容。虽然这个方案具备可行性，但是需要配置多个不同的 yaml 文件搭建多个 Prometheus 去发现多个不同服务的监控数据，增加了工程师的运维负担。

我们使用 Kubernetes Operator 来简化这部分工作。在 Kubernetes 的支持下，管理无状态的微服务已经变得比较简单，内置组件 Deployment 可以在无需附加操作的情况下，就可以管理应用的生命周期。

而对于数据库、监控系统等有状态的服务，就需要做一些额外的管理工作，这部分的工作又牵涉到具体应用相关的运维知识，具有资深经验的大牛可以把这些运维技术知识编写成自动化的 Operator，借助 Kubernetes 的能力，像操作 Kubernetes 内置资源一样简单地运行和管理这些复杂的有状态应用，减少工程师的运维成本。

Prometheus operator 通过自定义资源类型 CRD 来简化 Prometheus 部署，使用了 namespace selector 简化了监控目标服务的发现，每个 Prometheus 负责收集特定 namespace 里服务的监控数据。



Prometheus operator

虽然 Prometheus 可以多实例部署，但视图层 Grafana 查询数据需要配置多个数据源，这些数据源相互独立，不能聚合统一到全局视图。另外，我们希望能长期存放至少 3 个月的历史数据，数据全部放在本地磁盘存在高昂的存储成本和灾备迁移成本问题。我们引入了 Thanos 组件，解决以下几个核心需求：

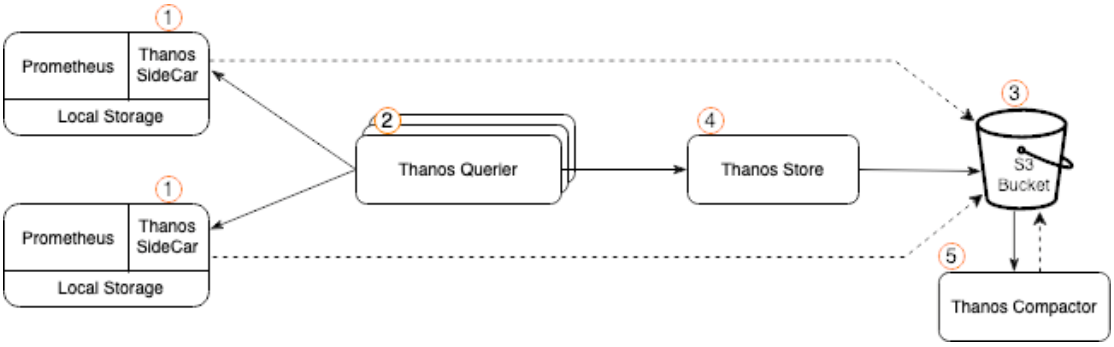
1）Thanos Sidecar 定期 2 小时从 Prometheus 服务收集数据上传到远程对象存储（AWS 的 S3），降低丢失数据的风险和历史数据存储成本。而且像 S3 这样的对象存储本身有多副本高可用特性，降低了容灾成本。

2）Grafana 只需要查询 Thanos 的全局接口，Thanos 从多个 Prometheus 实例和对象存储拉取数据去重聚合。

3）此外 Thanos Compact 组件可以对特别久的历史数据 downsample 和压缩，进一步减少存储成本。

同时，Prometheus operator 也封装了 Thanos 相关的集成运维工作，简单修改几个配置就能把 Thanos 作为 sidecar 组件接入。

通过 Grafana + Prometheus Operator + Thanos，一个高可用和高扩展的监控系统就搭建好了。



Thanos 架构

三、成本优化

上云的成本也是重要考量的标准。很多人会问，从 IDC 迁移到公有云后，成本会减低吗？如果只是直接把 IDC 使用方式照搬到公有云上，对资源依赖的总量没有变化，那自然成本也不会有多大的改善。

每个上云团队都会面临成本管理的挑战，从技术的角度看，我们可以优化应用提高资源利用率避免浪费，同时利用公有云的按需付费、不用补付费的弹性特性。

3.1 优化计算资源成本

3.1.1 弹性伸缩

以计算资源成本为例：计算实例成本 = 实例运行时长 * 实例价格。如果只是简单粗暴把本地机房的运行模式套用到云上计算，云服务计算资源的费用是高于本地机房的。所以我们需要充分利用云上按需收费的特性，减少闲置资源成本。实例的运行时长和 Kubernetes 集群内的服务数量，以及分配给这些服务的计算资源成正比。同时服务的数量又是和流量成正比。

机票业务场景存在突发的业务流量，比如临近节假日颁布的旅游政策，营销直播活动甚至是最近的疫情。云原生的弹性特性很好地利用合理的资源应对突发的流量。Kubernetes 的 HPA 弹性架构会实时采集集群整体的负载指标，判断是否满足弹性伸缩条件和执行 pod 的伸缩。仅仅是 pod 的伸缩还不够，我们还需要在集群中使用 Cluster Autoscaler 组件，监控集群中由于资源分配不足无法被正常调度的 pod，自动从云平台的实例池中申请增加节点。同时在业务流量下降后，Cluster Autoscaler 组件也会检测集群中资源利用率较低的节点，将其中的 pod 调度到其他可用节点上，回收这部分闲置节点。

下图是我们对资源节点数（黄色曲线）的监控，黄色曲线因为流量增加出现过一段尖刺，如果不使用弹性伸缩，我们要保证集群能在峰值流量时稳定可用，至少要把集群的总节点数维持在红色虚线这个位置；使用弹性伸缩后，集群节点的平均数量就可以保持在更低的绿色曲线位置。

弹性伸缩案例

云原生的弹性特性不仅帮助减少资源使用成本，也提高服务对基础架构故障的容错率，在基础设施部分可用区中断不可用期间，其他可用区域会增加相应数量的节点继续保持整个集群的可用。

Kubernetes 支持对 pod 容器所需的 CPU 和内存调整，找到一个合理的配额以合理的成本达到最佳的性能。所以我们在服务上云前会做一些接近真实环境的负载测试，观察业务流量的变化对集群性能的影响（业务周期性高峰和低峰的资源使用率，服务的资源瓶颈，合适的余量资源 buffer 应对尖刺流量等等）。既不会因为实际利用率过高导致稳定性问题，比如 OOM 或者频繁的 CPU throttling，也不会因为过低浪费资源（毕竟，即使你的应用只使用了实例的 1%，也要支付该实例 100%的费用）。

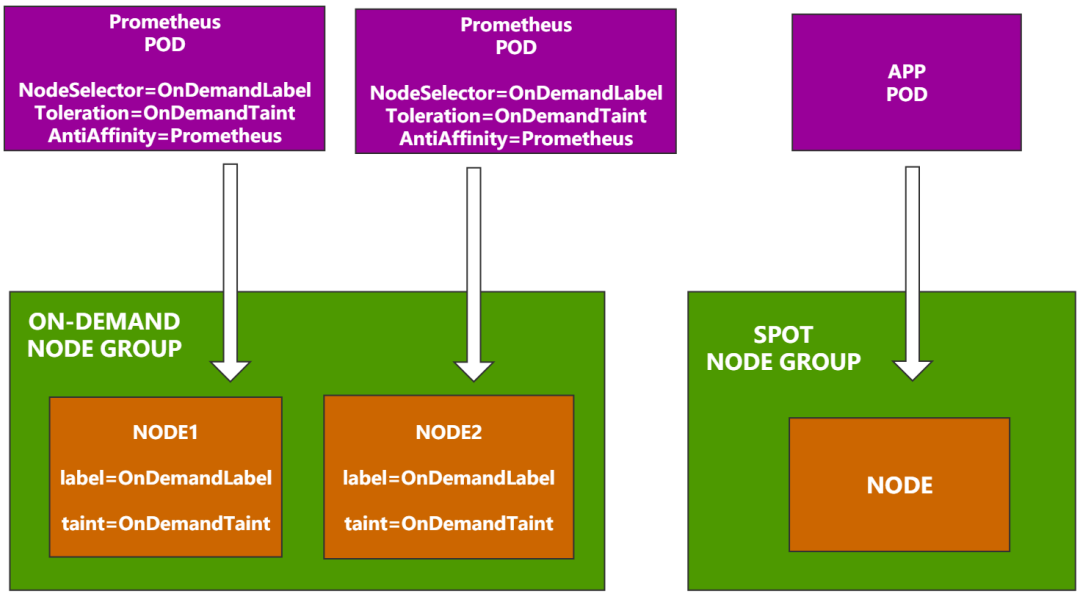
3.1.2 竞价实例

某些云平台会把一些闲置计算资源作为竞价实例，比按需实例更低的定价出租，顾名思义，竞价实例的最终费用是按市场供需出价决定的。按照我们实际使用的体验，如果不是特别热门的机型，定价基本在按需实例费用的 30%左右。低价的竞价实例自然有它的限制，云平台会因为按需实例池的资源不足会回收部分竞价实例，当然在回收前会提前通知到这些实例。例如 AWS 会提供 Terminal handler 组件在收到回收通知后提前把容器调度到其他可用的实例上。

更低的价格　　大量的可用实例池　　同样的性能

此外配置了多种机型和不同可用区域的竞价实例池，进一步降低了竞价资源被回收概率，最后会有保底的按需实例池避免竞价实例完全不可用导致服务中断。

携程国际机票使用按需实例和竞价实例的混合部署，保证低成本和高可用。一些系统关键组件（比如 Cluster Autoscaler），中断就会丢失数据的有状态服务（比如 Prometheus）运行在按需实例。而对错误容忍度高，使用灵活无状态的业务应用运行在竞价实例上。通过 Kubernetes 的节点亲和性控制不同类型的服务调度到对应类型标签的实例上。



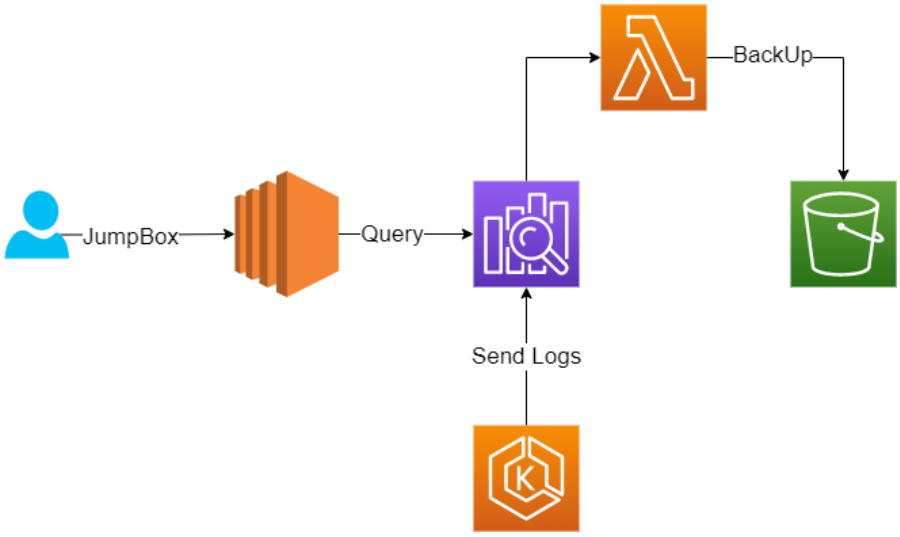
节点亲和性

3.2 数据存储成本优化

日志和监控的历史数据，我们会定期备份转移到更低成本的对象存储。

Thanos 的 Sidecar 会定期 2 小时备份监控数据到对象存储。
我们创建周期性的定时脚本任务，把 Elastic Search 的日志数据打成快照文件，迁移到低成

本的对象存储。通过 Serverless 方式运行定时任务。以 AWS 的 Lambda 为例，按照任务实际运行时长和请求次数计费，缺点是只能运行生命周期短的函数，但运行低频周期性且简单的定时脚本是非常合适的。



3.3 网络传输成本优化

网络部分的优化依赖实际业务场景。以携程国际机票业务为例，主要的数据传输量集中在查询场景（根据用户实际的搜索条件从海外各个供应商平台拉取机票相关的业务数据），和外部下游服务的流量比例是出少进多。一般服务和外网接口交互走的是云平台提供托管的 NAT 网关，对出网和入网流量收取相同的费用。但国际机票查询业务的场景更适用于只收取出网流量费用的收费模型，所以我们自建部署透明代理 squid 在能访问外网的子网段，运行在私有子网段的业务应用通过代理 squid 转发请求到外网供应商平台服务。

四、总结

本文通过携程国际机票在云原生的实践，分享了如何快速在云上搭建一套稳定高效的生产环境实现快速交付、智能弹性，以及在云上的一些成本优化经验。借助云原生体系实现了基础设施自动化，释放一部分的运维工作就可以更多地投入到业务迭代，更敏捷地响应业务需求迭代，通过监控和日志实现快速试错和反馈。

整体而言，上云涉及的方方面面实在太多了，限于篇幅很难全部讲清，希望本文的一些小分享能帮助到更多想上云的团队，少走弯路，拥抱云原生。

容器成本降低 50%，携程在 AWS Spot 上的实践

【作者简介】单双，携程 Cloud Container&Service 团队公有云专家，主要从事携程混合云建设，同时服务于携程集团内业务的上云工作

一、背景

AWS Spot 实例，即竞价实例，是 AWS 把用户未购买的空闲计算资源以低于按需价格的方式出售给用户，以期带来收益。通常，AWS Spot 实例的价格是按需实例价格的 30%，对于 AWS 使用者来说，如果合理使用，可以大大节省云上费用的支出，是节省成本的一大利器。

在企业的实践中，由于 Spot 实例会随时被回收，不合理的使用会对系统的稳定性造成冲击。如何在节省成本的同时，保证系统的稳定性和可靠性，是一个值得投入的课题。携程集团各业务(机票、酒店等)有大量应用长期运行在 AWS 上，我们通过 Spot 实例的大规模使用，成功将业务的容器使用成本降低了 50%，以下将分享我们的经验。

二、携程使用 Spot 实例的实践

2.1 Spot 实例特性分析

携程内部使用 Spot 实例的应用场景，是引发我们思考在引入 Spot 实例之后如何采取措施，更好地保证系统稳定可靠的出发点。

Spot 实例的优点

- 成本优势显著：通常价格仅是按需实例的 30%
- 计费方式灵活友好：相比预留实例（RI）或 SavingPlan 等其他成本节省方式，仅按使用时长付费，无需容量预留或预付，没有因过量购买而造成浪费的负担

Spot 实例的缺点

- 回收终止对程序造成的影响：实例随时会被 AWS 回收终止，导致程序中断
- 不确定性：回收不受用户自己控制，无法预估，无法确定下一刻会发生什么、目前的 Spot 实例是否很快会被回收，处于被动状态
- 容量在可用区间不均衡：各可用区的容量容易出现不均衡现象，即使打开 AutoScaling Group 的容量自动均衡功能，也无法避免。对于需强制多可用区部署的应用需要特别注意

2.2 Spot 实例的应用场景

我们看到 Spot 实例具有随时会被 AWS 回收终止的特点，所以 Spot 实例比较适合灵活性较高或具有容错性的应用程序。同时，实例被回收后又如何自动保证应用的容量，K8s 天然地解决了这一问题，所以，我们在 K8s 的无状态业务负载节点大量使用了 Spot 实例，容器的

单价成本节省了 50%。当然，一些对稳定性要求非常高或者有状态程序，如 K8s 核心组件 Scheduler webhook、HPA、Cluster Autoscaler、Metrics Server 等，应避免部署在 Spot 节点上。否则，Spot 实例回收过程中需进行容器迁移，这些组件会因重启造成抖动进而影响其他 Pod 正常启动，或者造成状态丢失，影响系统的可用性。

2.3 Spot 实例中断事件的处理

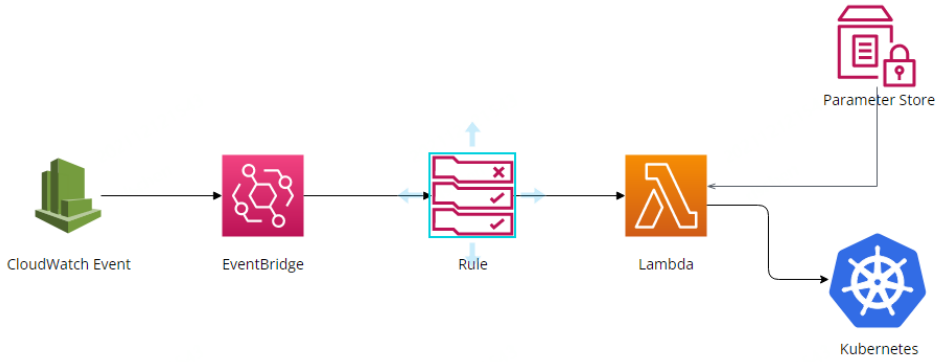
AWS 回收 Spot 实例时，将在执行回收动作前两分钟发出一个事件，这需要高弹性架构的支持以及处理实例突发中断的技术措施来应对。有两种方式可以检测到该事件：

1）CloudWatch Events：CloudWatch Events 会发出类型为“EC2 Spot Instance Interruption Warning“的事件，通过配置事件规则对事件进行匹配，触发对应的动作。

2）实例元数据（Metadata）服务：通过在实例内 curl http://169.254.169.254/latest/meta-data/spot/instance-action 可以查看该事件相关信息：

```
{"action": "terminate", "time": "2021-10-10T10:10:00Z"}
```

在 K8s 中，Spot 实例被回收，其本质来说，是个 node 驱逐的过程，执行的操作图如下：

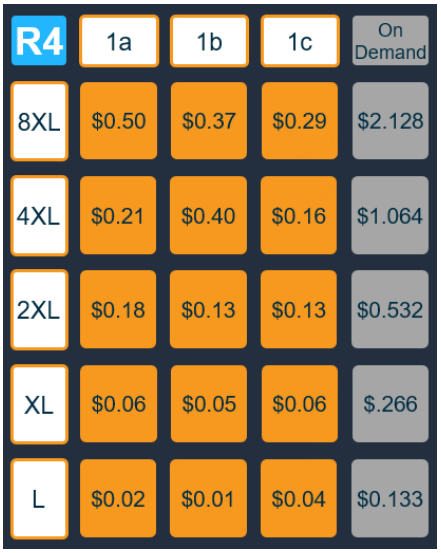


通过监测 EC2 Spot Instance Interruption Warning 的 CloudWatch 事件，配置 CloudWatch Events 规则，触发 Lambda。出于安全的考虑，调用 K8s apiserver 所需要的如认证等信息存储在 Parameter Store 上，给 Lambda 赋具有 AmazonSSMReadOnlyAccess 权限的 IAM 角色，调用 K8s Apiserver 对 Node 进行驱逐，对 Pod 进行迁移。

我们采用 CloudWatch Events 而非检测实例元数据服务的方式，一方面原因在于开销少，无需在机器上部署，包括对应日志收集的程序；更重要的原因在于考虑到对实例回收事件引发的故障的排障需求。由于实例会在两分钟内被释放，没有机器现场，后续排障只能依赖推送到日志系统的日志。若使用实例元数据服务，极有可能丢失事件现场的日志：实例元数据未能准确送入到元数据服务上、实例上程序异常退出、实例网络问题、日志链路不可用等。而 Lambda 运行的日志都保存在 CloudWatch Logs 中，CloudWatch Event 的方式是与 EC2 实例不交叉的链路，不存在上述的问题。

2.4 Spot 实例的应用高可用思考和设计

首先我们需要了解 AWS Spot 计算容量池的后台设计理念。Spot 容量池是一组未使用的 EC2 实例，它们具有相同的实例类型、操作系统、可用区和网络类型（EC2-Classic 或 EC2-VPC）。每个 Spot 容量池的价格都不同，具体取决于供需情况。我们都是 VPC 场景，Linux 平台，那么重点考虑下实例类型和可用区，如 eu-central-1a+r4.4xlarge 是作为一个 Spot 容量池进行提供的。当某一个容量池资源紧张时，那我们账号内该容量池的 Spot 实例很有可能将会被回收，而且用户不可控制。

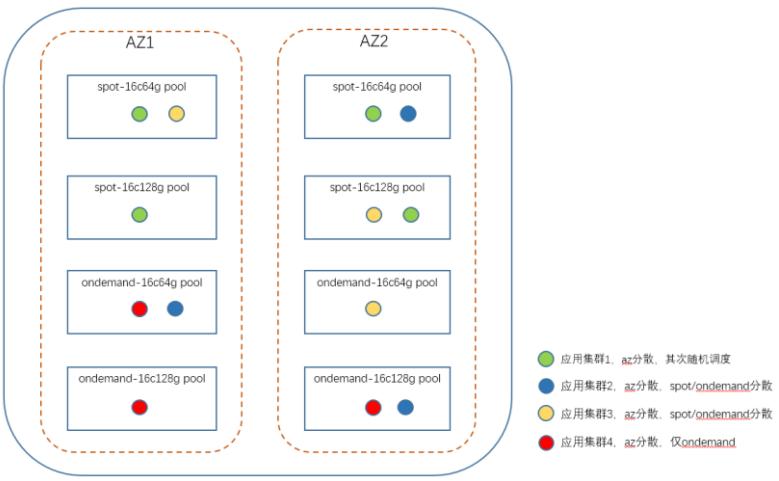


图引用自 AWS Container Day 2019 Barcelona – Amazon EC2 Spot Instances[1]

为了保证高可用，降低同时段 Spot 实例回收对应用的影响，我们对应用的部署和调度需要考虑容量池和可用区的因素，也就是我们需要重新设计 Spot 实例下应用的部署架构。

2.4.1 高可用部署架构

设计跨可用区的高可用架构，而且，需要把同应用分散到不同的 Spot 容量池。



1）从高可用的角度出发，应用必须跨可用区进行部署。首先，多可用区的设计避免在单可用区故障时，应用实例同时遭受影响导致服务不可用。同时，在平时 Spot 实例频繁回收的场景下，往往某个可用区的实例会相对其他可用区的实例的紧张度更突出，保证同个应用至少会使用两个 Spot 容量池，降低了某个 Spot 容量池紧张该应用下所有 Pod 实例同时受影响的可能性。

2）对于实例数不多（少于一定数量），但核心的应用，或者更高稳定性的要求，除了满足多可用区分散的调度策略外，还可以考虑混合使用 Spot/OnDemand 实例，降低同应用所在的宿主机实例同时被回收的风险。在我们的实践中，我们还添加了按应用开启该策略的能力。

3）对于 K8s 核心组件及有状态应用，仅部署到 OnDemand 实例上。

2.4.2 Pod 调度策略

我们使用调度器的 TopologySpreadConstraints 功能，来达到同个 K8s 集群内容器应用跨越多个故障域的高可用部署架构。

1）可用区分散的调度策略

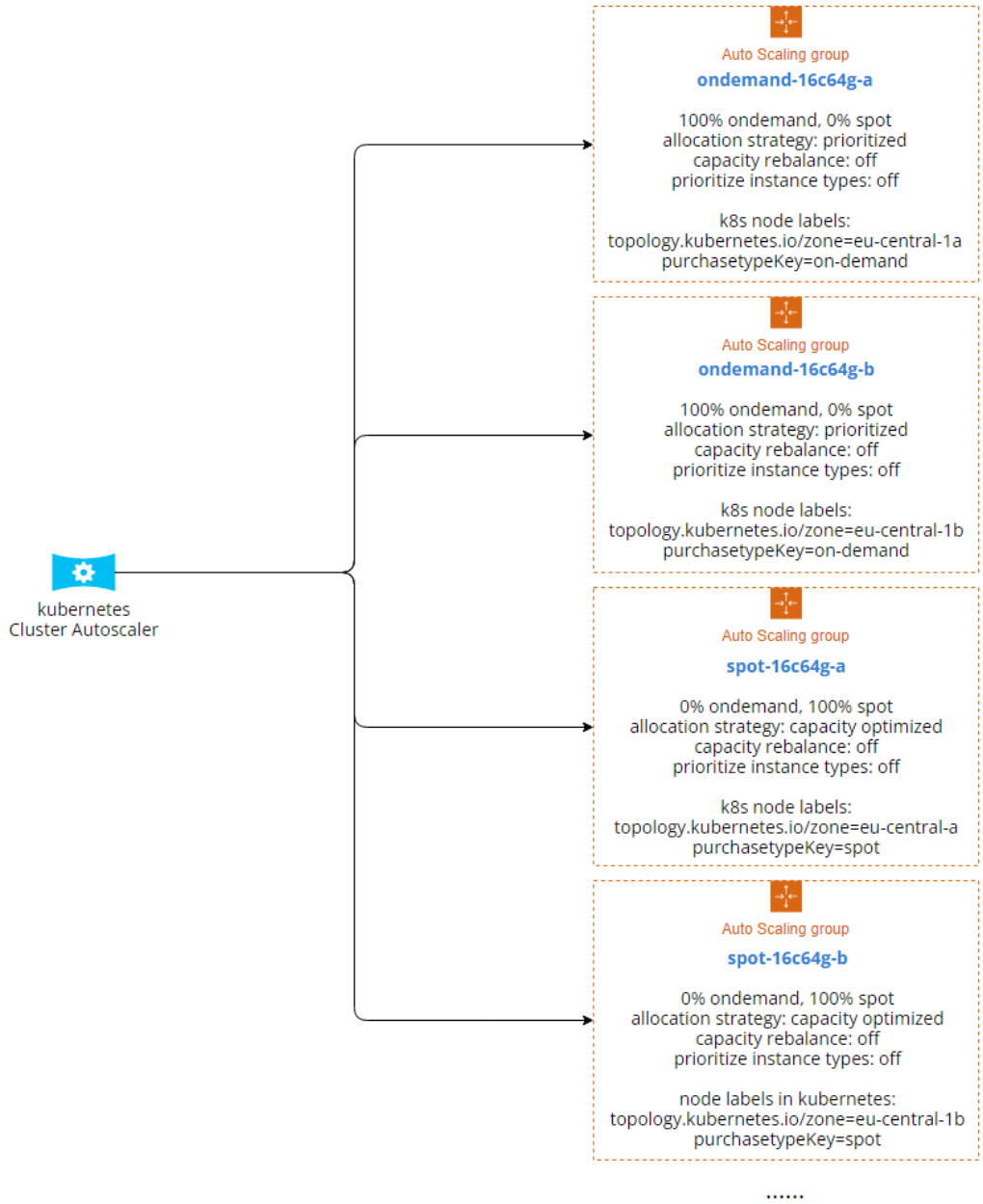
```
topologySpreadConstraints:
- maxSkew: 1
  topologyKey: topology.kubernetes.io/zone
  whenUnsatisfiable: DoNotSchedule
  labelSelector:
    matchLabels:
      GroupidKey: "%s"
```

2）可用区分散+Spot/OnDemand 混部的调度策略

```
topologySpreadConstraints:
- maxSkew: 1
  topologyKey: topology.kubernetes.io/zone
  whenUnsatisfiable: DoNotSchedule
  labelSelector:
    matchLabels:
      GroupidKey: "%s"
- maxSkew: 1
  topologyKey: purchasetypeKey
  whenUnsatisfiable: DoNotSchedule
  labelSelector:
    matchLabels:
      GroupidKey: "%s"
```

2.4.3 NodeGroup 与 Autoscaling Group 的设计

基于上述的应用部署架构和调度需求分析，我们在调度上需要感知可用区和Spot/OnDemand。基于 Cluster Autoscaler 的原理，系统中作为 Pod 调度策略的 Labels/Taints 键值对集合都需要作为一个单独资源池，才能精确触发对应的 Autoscaling Group 的扩缩容。所以在特定用途内，从实例配置（目前是核数和内存）、可用区、Spot/OnDemand 这几个维度完成资源池的结构设计：



2.4.4 异常处理策略

在可用区故障时，多可用区的部署架构首先使得服务不会整体挂，但是进入故障可用区的流量会受到影响，服务可用性降低。而且，在故障未完全解除前，故障可用区的部分云服务可能会出现在健康与不健康间抖动的状态，大大影响服务的正常工作。这种异常情况下，我们的策略是使用一切手段使得流量往正常可用区转移，实现故障恢复。这种策略基于原则：恢

复优先、容量优先及固化最佳实践为可重复的 SOP。具体的：

1）临时解除 Pod 可用区分散的调度策略：当可用区故障时，该可用区的 K8s Node 很可能会遭遇实例故障，如 Pod 的网络不通、Kubelet 自动重建 Pod；或故障处理过程执行迁移 Pod 时，当前的 K8s Node 资源很可能是不充足的状态，此时 Pending 事件触发 Cluster Autoscaler 进行扩容。如果还是保持可用区分散的调度策略，仍会扩容出故障可用区的实例。所以我们需要关闭可用区分散的调度策略。

2）临时关闭 Cluster Autoscaler 对故障可用区的 NodeGroup 纳管：去除可用区分散的调度策略后，Cluster Autoscaler 会随机挑选匹配的各可用区 NodeGroup，此时需要剔除故障可用区的 NodeGroup。具体方式是把故障可用区的 Autoscaling Group 的 k8s.io/cluster-autoscaler/enabled=true 标签都去除。

3）临时关闭 Cluster Autoscaler 对 Spot 的 NodeGroup 的扩容：在平时监控到自己账号内的 Spot 回收频率异常高时，需要及时联动进行 Spot 比例的调整，避免新扩容 Spot 实例。可用区故障时，非故障可用区资源也很有可能出现紧张、踩踏的情况，此时应避免开启 Spot。因为在容量池紧张的情况下，即使 Spot 竞价成功生命周期也不长，会很快被回收以优先满足按需实例购买者的用量，此时应避免引入更多的不稳定和波动因素，造成应用实例频繁地被迁移而导致故障影响时间被延长。

4）临时关闭自动弹性缩容：避免上游链路因故障致流量下降后资源使用降低而自动缩容，流量恢复后还得重新进行扩容。同时也给系统减轻负担，避免上下波动，往快速恢复方向用力。

上述就是在使用 Spot 实例时需要考虑的可用区故障、平时 Spot 回收状态异常高等场景需要采取的处理措施。这需要系统具备如下的能力：

1）调度策略动态调整的能力：受益于携程 K8s 容器平台的统一调度体系[2]，我们把默认可用区分散调度的策略和针对部分应用 Spot/OnDemand 分散调度的策略作为两份 PolicyTemplate，由 sched-webhook 根据绑定的 Policy 实现动态地配置 Pod 的调度行为。

2）SOP 处理流程规范：经反复验证实践出的 SOP 流程文档手册。

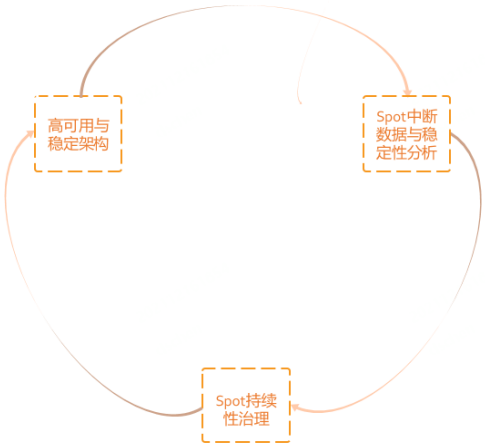
2.4.5 K8s Cluster Autoscaler 对 TopologySpreadConstraints 的支持

目前社区的 K8s Cluster Autoscaler 尚未支持 TopologySpreadConstraints，在我们的场景下会造成非预期 NodeGroup 的扩容或缩容。所以我们进行了扩展，新增对 Spec.TopologySpreadConstraints Metadata 的支持，目前已稳定运行在生产环境中。

2.5 Spot 实例集群的长期治理

虽然 AWS 提供了 Spot Advisor 工具帮助用户根据折扣及中断概率进行实例类型的选型，但是该工具的数据粒度较粗，也无直接可观测的工具可以查看自己账号内的 Spot 实例中断次数、频率等统计情况。从长期维护治理的角度出发，我们自己收集记录每次 Spot 实例回收

的事件及对应实例的信息，特别关注可用区、实例类型、存活时长等属性，把 Spot 中断历史数据保留下来，便于后续数据分析，进行持续性的治理。



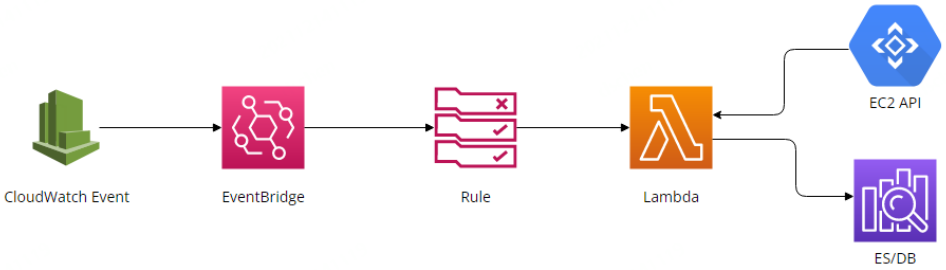
可观测性是大规模长期运营的前提，可以在以下场景发挥作用：

1）观测及排障：实际查询实例由于何原因被终止，是否因为 Spot 实例回收导致实例中断。

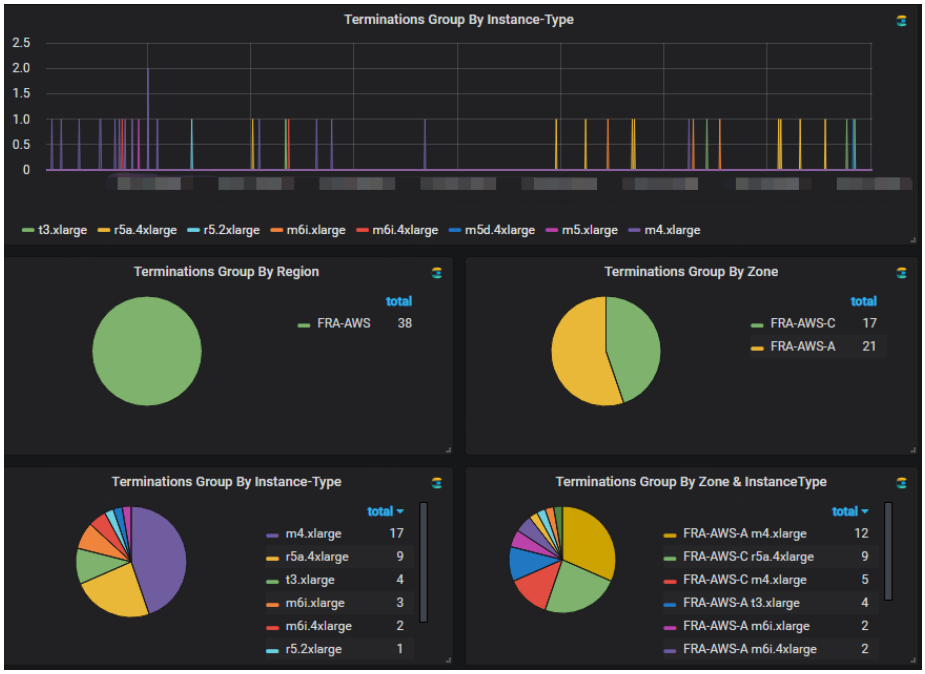
2）策略优化：回收经数据汇总分析后，可以查看当前各可用区各实例类型的中断频率，对中断频率过高的 Spot 容量池进行替换；以及直观了解现阶段 Spot 实例整体的波动状况，为当前 Spot 比例的控制提供依据；把 Spot 数据作为影响系统稳定性输入的一个因子，结合稳定性数据，计算出稳定性保证范围内的 Spot 比例阈值指标。

3）容量规划：比例阈值指标经数据分析后，得出如季节性或节日性规律，便于做好下一周期的 Spot 实例容量规划和控制管理。

可观测性



通过监测 Amazon EventBridge 中的 EC2 Spot Instance Interruption Warning 的 CloudWatch 事件，配置 CloudWatch 规则，触发 Lambda。Lambda 通过 EC2 API 获取该实例的详细信息，存入到存储系统。基于数据，可以制作监控的告警看板，以及进行后续的治理分析。



三、总结

本文介绍了携程在 AWS 上使用 Spot 实例的实践，重点介绍了使用的场景，引入 Spot 实例后面向恢复的高可用服务的思考和设计，不断优化保证稳定性与可靠性。Spot 实例的使用使得 AWS 上的容器成本降低了 50%。未来，会持续对 Spot 实例集群系统进行治理，在管理上更自动化智能化，在成本和稳定性的矛盾中继续探索，保证系统稳定性输出的同时，最大程度地降低成本。

四、引用

【1】AWS Container Day - Amazon EC2 Spot Instances
https://www.youtube.com/watch?v=tPdv1wAG6o4
【2】10W+ K8s 容器数量下，携程如何打造统一弹性调度体系
https://mp.weixin.qq.com/s/PZIPI6OJz26XnkAho3dY9g

数据库

分布式数据库 TiDB 在携程的实践

【作者简介】Army，携程数据库专家，主要负责分布式数据库运维及研究。Keira，资深数据库工程师，主要负责 MySQL 和 TiDB 运维。Rongjun，携程大数据架构开发，专注离线和实时大数据产品和技术。

前言

携程自 2014 年左右开始全面使用 MySQL 数据库，随着业务增长、数据量激增，单机实例逐渐出现瓶颈，如单表行数过大导致历史数据查询耗时升高，单库容量过大导致磁盘空间不足等。为应对这些问题，我们采取了诸多措施如分库分表的水平拆分、一主多从读写分离、硬件 SSD 升级、增加前端 Redis 缓存等，但同时也使得整个业务层架构更加复杂，且无法做到透明的弹性，因此开始将目光转移到分布式数据库以解决这些痛点。

近年来受到 Spanner&F1 的启发，基于 CAP 理论和 Paxos、Raft 协议作为工程实现的分布式数据库得到了蓬勃发展，从硅谷的 CockroachDB 到国产的 TiDB 都在社区产生了很强的影响力。携程也对这些产品从社区活跃度、用户规模、易用性等多个方面做了调研，最终选择了国产的 TiDB。

TiDB 是一个开源的 NewSQL 数据库，支持混合事务和分析处理（HTAP）工作负载，兼容大部分 MySQL 语法，并且提供水平可扩展性、强一致性和高可用性。主要由 PingCAP 公司开发和支持，并在 Apache 2.0 下授权。2018 年 11 月我们开始 TiDB 的 POC 以及与携程现有运维平台的整合，2019 年 1 月第一个线上应用正式接入，最初的目标只是保证数据库的可用性以及可以存储足够多的关系型数据。随着 TiDB 快速迭代，越来越多的功能进入社区，如 HATP 特性，让我们不局限于最初的目标，开始了新的探索。本文将介绍 TiDB 在携程业务场景中的运维实践，希望对读者有所帮助和参考。

一、架构

携程内部历时 1 年，代号为“流浪地球”的机房级故障演练，验证了 IDC 级别故障容灾能力。我们将 TiDB 的三个副本分布在三个数据中心，保证在单中心故障时不影响对外服务，同时数据一致性也不受影响，并在 tidb-server 层实现了自动探活以及自动故障切换，让 RPO 等于 0，RTO 小于 30S。

我们先来了解一下 TiDB 的整体架构（如图 1-1），再结合携程的场景来部署。

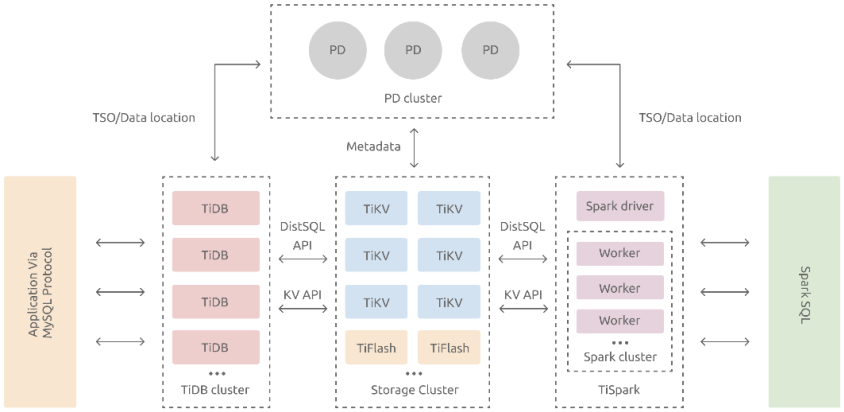


图 1-1 TiDB 的整体架构图

从 TiDB 的架构图我们可以看到，得益于组件 PD 和 TiKV 都通过 Raft 实现了数据的容灾，原生就提供了多 IDC 的部署能力，和 Google Spanner 采用原子钟方案不同的是，TiDB 采用了 PD 进行单点全局统一授时的 Timestamp 方案。TiDB 中的每个事务都需要向 PD leader 获取 TSO，当 TiDB 与 PD leader 不在同一个数据中心时，它上面运行的事务也会因此受网络延迟影响。目前携程的跨 IDC 延迟在 0.5-2ms 之间 ，属于可接受的延迟范围。配置三数据中心时，需要对相应的 TiKV 和 PD 的 label 配置规则，这样 PD 在调度 region 副本时会根据标签属性在每一个机房都拥有一份全量数据。具体的一个配置示例，如图 1-2:

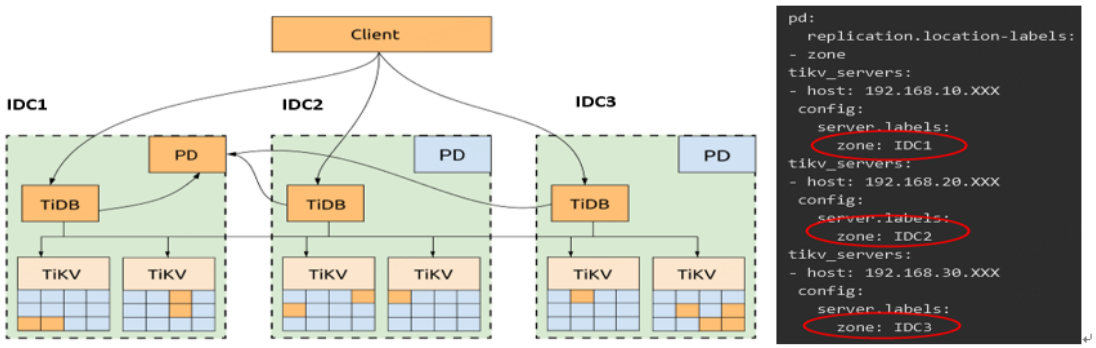


图 1-2 TiDB 在携程的部署架构和配置

这种部署架构的优点:

- 所有数据的副本分布在三个数据中心，具备 IDC 级别的高可用和容灾能力
- 任何一个数据中心失效后，不会产生任何数据丢失 (RPO = 0)
- 任何一个数据中心失效后，其他两个数据中心会自动发起 leader election，并在合理长的时间内（通常情况 20s 以内）自动恢复服务

二、应用场景

TiDB 目前已经应用到携程的多个业务场景，包括风控、社区、营销、搜索、酒店等。这里选

取两个比较典型的使用案例——国际业务 CDP 平台和酒店结算业务。

2.1 国际业务 CDP 平台

因为 Trip 数据来源比较广泛，既有自身数据也有外部数据；数据形式也非常多样化，既有结构化数据，也有半结构化和非结构化数据；数据加工形式既有离线数据处理，也有在线数据处理，因此国际业务构建了 CDP 平台以解决加工这些数据，形成业务系统、运营、市场需要并且可以理解的数据和标签，具体可以阅读往期文章：《携程国际业务动态实时标签处理平台实践》。

TiDB 在其中主要承担存储业务持久化的标签以及内部 SOA 调用的查询服务。查询分为 UID 等维度的基础信息查询、订单订阅基础信息查询的 OLTP，以及 EDM\Marketing 等人群的 OLAP 查询。整个 CDP 平台的架构如图 2-1:

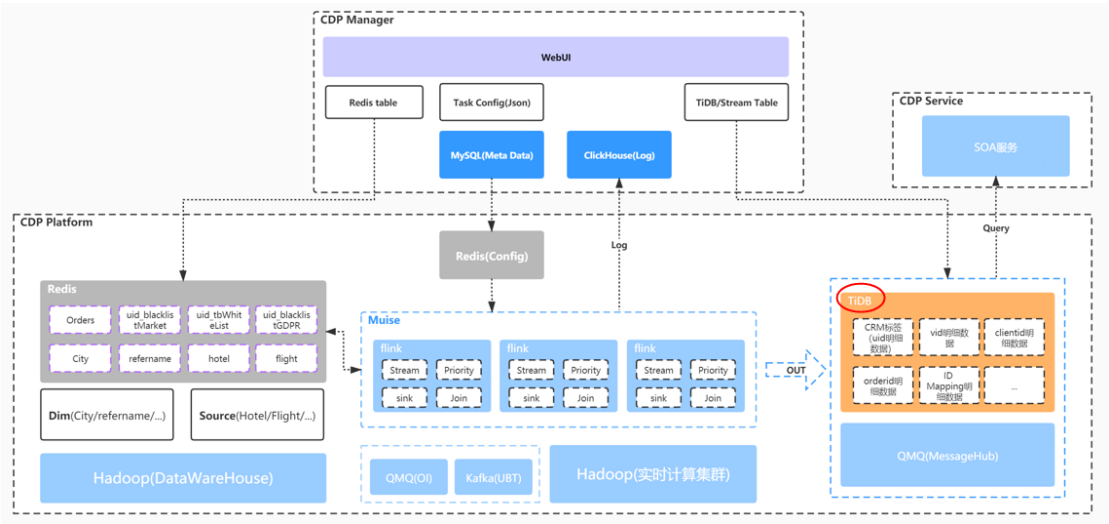


图 2-1 CDP 平台架构图

具体的数据处理，历史全量数据通过数据批处理引擎（如 Spark）转换完成以后批量写入到数据持久化存储引擎（TiDB），增量数据业务应用以消息的形式发送到 Kafka 或者 QMQ 消息队列，通过实时 DAG 处理完后持久化到存储引擎（TiDB）。

持久标签访问的主要场景有两个，一种是跟现有 CRM 系统对接，在线根据业务的特征圈选符合条件的业务数据，这种场景的查询条件不固定，返回结果集因筛选条件而定，对于数据存储引擎的数据计算和处理能力要求比较高，即我们在数据处理领域经常提到的 OLAP 的场景。另一种场景是线上业务根据前端传入的业务标签相关的唯一标识来查询是否满足特定业务要求，或者返回指定特征值，满足业务处理的需要，需要毫秒级响应，对应的是 OLTP 场景。

由于标签的多样性，有查询记录的字段多达 60 个，查询条件是 60 个字段的随机组合，无法通过传统数据库层的 Index 来提高查询效率，经典的方案是 OLTP 和 OLAP 分离，但数据会存储多份，多数据源的数据一致性是一个很大的挑战。

对于这种场景，我们开启了 TiDB 的 TiFlash，TiFlash 是 TiDB HTAP 形态的关键组件，它是 TiKV 的列存扩展，在提供了良好的隔离性的同时，也兼顾了强一致性。列存副本通过 Raft Learner 协议异步复制，但是在读取的时候通过 Raft 校对索引配合 MVCC 的方式获得 Snapshot Isolation 的一致性隔离级别。TiFlash MPP 模式如图 2-2。

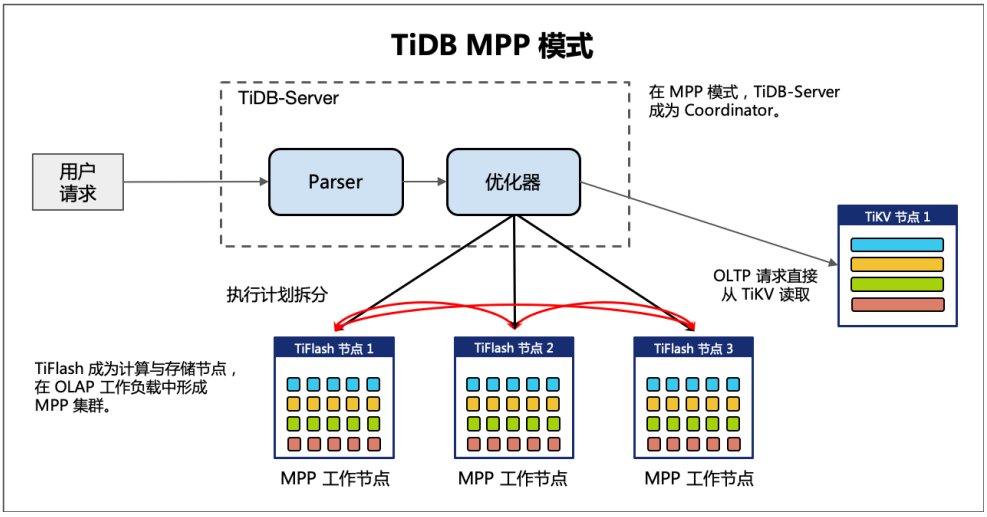


图 2-2 TiDB MPP 模式

这种架构很好地解决了 HTAP 场景的隔离性以及列存同步的问题，开启之后几个典型查询性能提升：

TiFlash MPP 提升，20s -> 1s
Set @@session.tidb_allow_mpp=1;
Set @@session.tidb_enforce_mpp=0;

id	estRows	actRows	task	access object	execution info
HashAgg_35	1.00	1	root		time:1.33s, loops:2, partial_worker:{wall_time:1.32919874s, concurrency:5, task_..
└─TableReader_37	1.00	1	root		time:1.33s, loops:2, cop_task: {num: 1, max: 0s, proc_keys: 0, copr_cache_hit_ra..
└─ExchangeSender_36	1.00	1	batchCop[tiflash]		tiflash_task:{time:1.33s, loops:1, threads:20}
└─HashAgg_9	1.00	1	batchCop[tiflash]		tiflash_task:{time:1.33s, loops:1, threads:1}
└─Selection_34	2315556.13	2096665	batchCop[tiflash]		tiflash_task:{time:1.32s, loops:3035, threads:20}
└─TableFullScan_33	193656482.00	194282298	batchCop[tiflash]	table:cdp_crm_uid_detail_basic	tiflash_task:{time:969.1ms, loops:3046, threads:20}

TiFlash 列裁剪，16.9s -> 2.8s
Set @@session.tidb_allow_mpp=1;
Set @@session.tidb_enforce_mpp=0;
Set session tidb_isolation_read_engines ='tidb,tiFlash';

id	estRows	actRows	task	access object	execution info	operator info
TableReader_7	36975350.65	36804646	root		time:16.9s, loops:36125, cop_task: {num: 380, max: 1.87s, min: 134.9m...	data:Selection_6
└─Selection_6	36975350.65	36804646	cop[tiflash]		tiflash_task:{proc max:866.7ms, min:115.2ms, p80:540.2ms, p95:646.9...	eq(ibucdptidb.cdp_crm_uid_detail_basic.message_locale, "zh-hk")
└─TableFullScan_5	193656482.00	194285534	cop[tiflash]	table:cdp_crm_uid_detail_basic	tiflash_task:{proc max:723.7ms, min:89.2ms, p80:394.9ms, p95:473.2ms...	keep order:false

2.2 酒店结算业务

携程酒店结算业务全库 6T，单服务器存储 6T 全量数据有很大挑战。常规的方法是用分库分

表的方式来减少实例数据量及压力，但分库分表的维度很难确定，无论从酒店维度还是供应商维度都无法避免跨片的查询，给程序的开发带来了很大的困难，并且大部分查询都是聚合运算，因此我们尝试迁移到 TiDB。

目前最大的表存储了 28 亿条数据，读写已完全切换到 TiDB。具体所使用的部署模式和上节提到的国际业务 CDP 平台类似，同样是开启了 TiDB 的 TiFlash 来加速查询的性能，具体的性能如图 2-3:

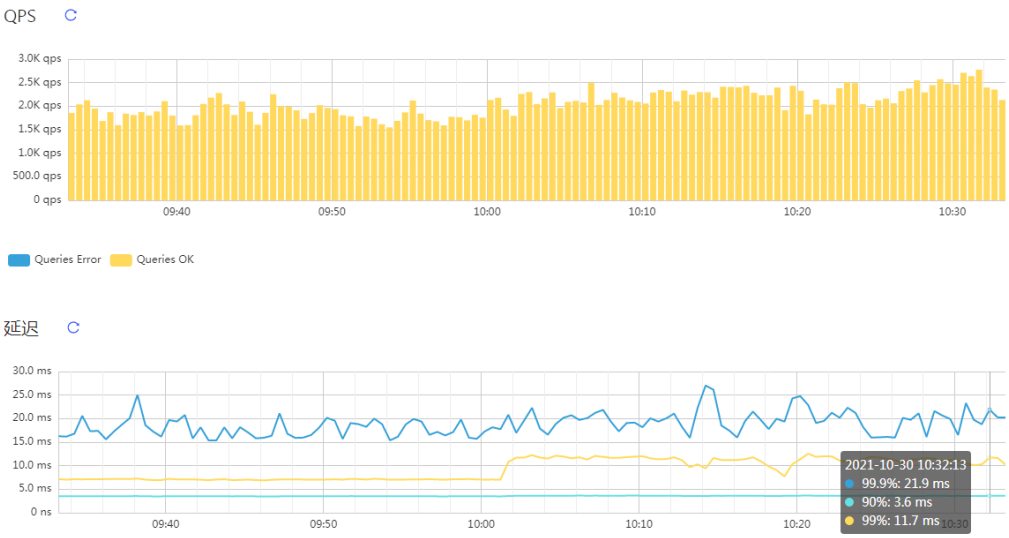


图 2-3 酒店结算性能监控

三、一些问题的实践

3.1 参数不合理导致的性能问题

分布式数据库有别于传统单机，通常 MySQL 遇到性能问题时可以快速定位是由于网络抖动、SQL 缺失索引还是请求次数激增等原因导致的，但分布式的 TiDB 组件众多，各个组件之间的网络通信、某个组件资源不足、SQL 复杂等都可能是导致出现性能问题的原因。目前官方提供了问题导图，方便根据不同的场景尽快定位原因。这里给出一个具体的案例，总结了一个典型问题的排查思路。

国际业务集群使用官方默认配置的集群上线测试时，发现写入耗时高达秒级，且耗时波动较大。来自应用端的监控（纵坐标单位为毫秒）,如图 3-1:

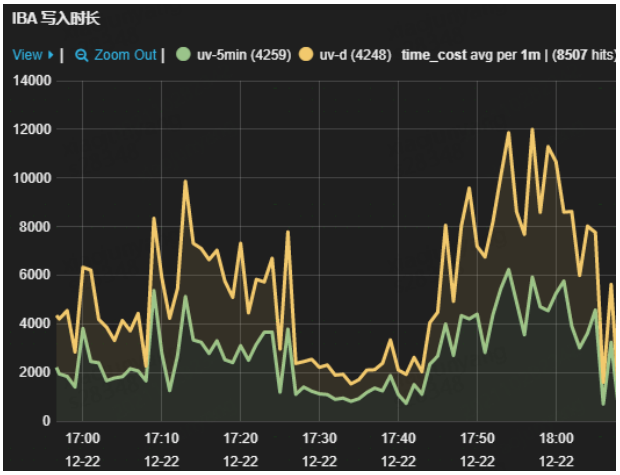


图 3-1 IBA 写入响应监控

根据 Pingcap 的导图发现 scheduler command duration 的时间约等于事务的 prewrite 时间（纵坐标单位为秒），可以看出 scheduler-worker 不足。如图 3-2:

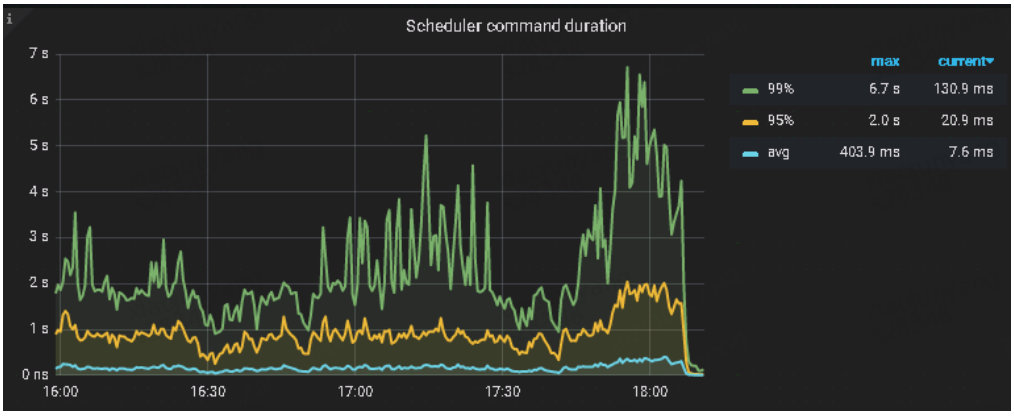


图 3-2 scheduler command duration 的时间

所以我们做了如下的调整:

- scheduler-worker-pool-size: 16 --> 40 （默认值为 4，最小值为 1，最大不能超过 TiKV 节点的 CPU 核数）
- scheduler-pending-write-threshold: "100MB" --> 1024MB （写入数据队列的最大值，超过该值之后对于新的写入 TiKV 会返回 Server Is Busy 错误）

调整完成后来自应用端的监控（纵坐标单位为毫秒），如图 3-3，红色箭头处是参数调整的时间点:

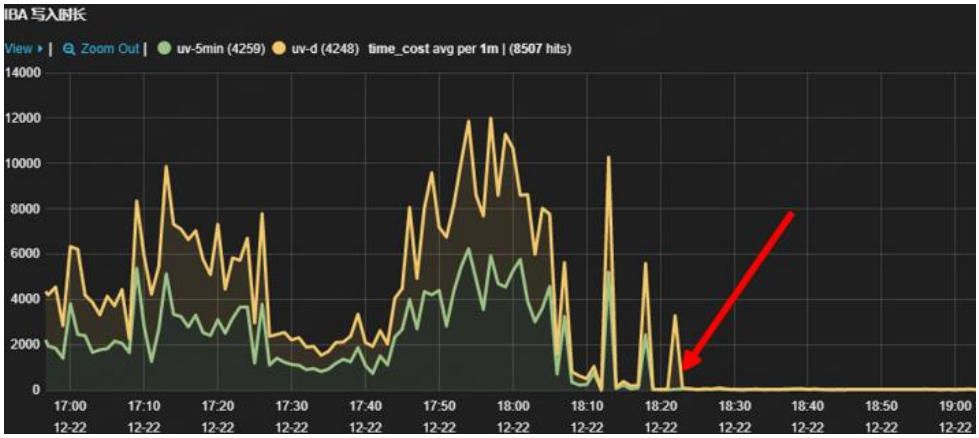


图 3-3 IBA 写入响应监控

总结：默认配置并非最佳配置，需要根据服务器硬件、使用场景不断调试并最终固化为每个集群甚至所有集群的最佳实践配置；根据 PingCAP 提供的问题导图，逐步定位具体哪个组件哪个方面存在瓶颈，我们同时也在进一步开发一键定位工具，能更快速的定位性能瓶颈。

3.2 分布式带来的自增列问题

含自增列的表，在自增列不强制赋值的情况下，insert 语句报主键冲突：

报错 SQL：INSERT INTO `xxx_table ` (`id, `name`, `tag`, `comment`, `creator`) VALUES (?, ?, ?, ?, ?)

报错内容： com.mysql.jdbc.exceptions.jdbc4.MySQLIntegrityConstraintViolationException: Duplicate entry 175190 for key 'PRIMARY'.

在 PingCAP 官方文档上，有以下介绍：

TiDB 中，自增列只保证自增且唯一，并不保证连续分配。TiDB 目前采用批量分配 ID 的方式，所以如果在多台 TiDB 上同时插入数据，分配的自增 ID 会不连续。TiDB 实现自增 ID 的原理是每个 tidb-server 实例缓存一段 ID 值用于分配（目前会缓存 30000 个 ID），用完这段值再去取下一段。假设集群中有两个 tidb-server 实例 A 和 B（A 缓存 [1,30000] 的自增 ID，B 缓存 [30001,60000] 的自增 ID），依次执行如下操作：客户端向 B 插入一条将 id 设置为 1 的语句 insert into t values (1, 1)，并执行成功。客户端向 A 发送 Insert 语句 insert into t (c) (1)，这条语句中没有指定 id 的值，所以会由 A 分配，当前 A 缓存了 [1, 30000] 这段 ID，所以会分配 1 为自增 ID 的值，并把本地计数器加 1。而此时数据库中已经存在 id 为 1 的数据，最终返回 Duplicated Error 错误。

通过这段介绍，我们了解到造成自增主键冲突，是因为存在自增主键显式插入导致。

结论：分布式数据库对于表自增列是预分配的，自增主键显式插入会导致 tidb-server 上的计数器混乱，引起数据写入报错。在开发规范中，我们明确要求 TiDB 不允许自增主键显式插入。

3.3 修改字段是否为空导致默认值异常

如下的表，我们字段从 int 升级到 bigint

```
CREATE TABLE `test` (`id` int);
alter table `test` add `col1` int(11) null default '0';
alter table `test` add `col2` int(11) null default '0';
alter table `test` change `col1` `col1` bigint(20) null default '0';
alter table `test` change `col2` `col2` bigint(20) null default '0';
```

我们发现默认值 0 不合适，因此，执行下面的语句，把默认值调整为 null

```
alter table `test` change `col1` `col1` bigint(20) null ;
alter table `test` change `col2` `col2` bigint(20) null';
```

此时，我们插入一条数据：insert into test(id) values(1);

神奇的发现，col1 和 col2 的值依旧是 0。这和我们的预期不符。经过一系列重现测试，以及社区论坛的查找， 我们发现这是一个已知问题：https://github.com/pingcap/tidb/pull/20491，该 Bug 在 TiDB 4.0.9 及以后版本已修复。

结论：成熟的社区论坛是 TiDB 日常运维和快速排障的利器，借助社区论坛上各种技术探索和交流分享，可以汲取优质内容，收获前沿知识，快速定位和解决问题。

四、监控与告警

对于分布式数据库运维，监控和告警是非常核心的一环，冒烟现象或者不规范现象，需要及时发现，及时解决，避免问题恶化。监控准确、告警及时可以帮助运维人员准确定位问题，快速解决故障。TiDB 使用开源时序数据库 Prometheus 作为监控和性能指标信息存储方案，使用 Grafana 作为可视化组件进行展示，我们在此基础上进一步做了整合。

4.1 TiDB 监控大盘

TiDB 原生提供 prometheus+Grafana 的性能大盘，数据非常丰富，但数据分散在单独的集群，无法提供全局视角，我们通过 prometheus 源生 remote write 到 9201 端口，自研了一个 adaptor 监听 9201 端口，转发性能数据到携程统一监控平台，搭建了我们自己的监控大盘。如图 4-1：

图 4-1 整合后的 TiDB 监控大盘

4.2 三副本监控

TiDB 使用三个以上的副本，通过 raft 协议来保证数据的一致性。当出现多数副本丢失或者宕机时，这部分数据处于不可用状态，是否存在副本缺失或者副本状态异常是需要特别注意的。因此我们会针对副本的数目及状态进行巡检，确保不会出现长时间内副本不足的情况，一旦发现有副本丢失，可以增加副本的调度线程，务必及时解决副本缺失问题。Region Peer 的监控如图 4-2:

time	instance	type	value
2021-10-16 10:03:50.260000		miss-peer-region-count	3
2021-10-16 10:03:50.260000		down-peer-region-count	4
2021-10-16 10:02:50.260000		miss-peer-region-count	0
2021-10-16 10:02:50.260000		down-peer-region-count	0
2021-10-16 10:01:50.260000		down-peer-region-count	0
2021-10-16 10:01:50.260000		miss-peer-region-count	0
2021-10-16 10:00:50.260000		down-peer-region-count	0
2021-10-16 10:00:50.260000		miss-peer-region-count	0
2021-10-16 09:59:50.260000		down-peer-region-count	0
2021-10-16 09:59:50.260000		miss-peer-region-count	0
2021-10-16 09:58:50.260000		down-peer-region-count	0
2021-10-16 09:58:50.260000		miss-peer-region-count	0
2021-10-16 09:57:50.260000		miss-peer-region-count	0
2021-10-16 09:57:50.260000		down-peer-region-count	0
2021-10-16 09:56:50.260000		miss-peer-region-count	0

图 4-2 三副本监控

4.3 磁盘容量监控

TiDB 存储数据量庞大，需要特别关注机器磁盘剩余可使用空间的情况，以免写满磁盘造成不必要的故障。对于磁盘的监控，我们的阈值是物理磁盘的 80%，一旦磁盘使用容量超过阈值，我们需要安排加机器扩容。对比相同情况下 MySQL 复杂的拆分方法，TiDB 的处理方法更简单高效。磁盘的监控告警如图 4-3:

群集名	异常点
test	store 的可用空间小于 500G：460.8
test	store 的可用空间小于 500G：459.8
test	store 的可用空间小于 500G：472.6
test	store 的可用空间小于 500G：460.2
test	store 的可用空间小于 500G：490.2
test	store 的可用空间小于 500G：469.3
test	store 的可用空间小于 500G：475.5

图 4-3 TiDB 磁盘监控

4.4 配置标准化检查

TiDB 集群的配置文件参数、系统参数众多，不同实例的配置项各不相同，且经常会对集群扩容缩容，因此我们要求变更前后集群的配置必须严格按照标准配置进行调整。只要做到配置标准，很大程度上就会保证集群标准化运行。配置标准化的监控告警如图 4-4：

主题：【巡检-生产-TIDB】检查TIDB群集配置cluster_config表

群集名	VIP	VIP_PORT	非标配置类型	非标配置实例	非标配置位置	非标配置
Testclu	192.168.1.1	1111	tidb	192.168.1.1:1111	南通	binlog.ignore-error
Testclu	192.168.1.2	1111	tidb	192.168.1.2:1111	南通	binlog.ignore-error
Testclu	192.168.1.3	1111	tidb	192.168.1.3:1111	南通	binlog.ignore-error
Testclu	192.168.1.4	1111	tidb	192.168.1.4:1111	南通	binlog.ignore-error

图 4-4 配置标准化的监控告警

4.5 性能告警

有时候会存在突发的流量上升，或者瞬间的性能尖峰的情况，这时候就需要关注性能告警。METRICS_SCHEMA 是基于 Prometheus 中 TiDB 监控指标的一组视图，有了基础的性能数据，我们只需要根据性能阈值，及时告警，及时分析处理。

五、周边工具

除了监控与告警，我们也开发了一系列周边工具，对于 TiDB 的运维，带来了更大的便利。这些周边工具主要包括：

5.1 和现有的数据周边工具打通

现有的数据周边工具主要包括：数据库的发布（DDL），数据在线查询，数据在线修改，以及和现有的大数据流程打通等，这些支持 MySQL 的工具也一样可以支持 TiDB，为 MySQL 迁移 TiDB 打解决了后顾之忧，让之前使用 MySQL 的开发测试人员可以方便流畅地切换到 TiDB。

5.2 TiDB 部署工具

TiDB 集群实例角色较多，集群部署有别于传统单机，需要单独开发一套部署工具，包括集群上线流程、集群下线流程、扩容缩容实例、集群版本升级等。

5.3 TiDB 闪回工具

有时候会遇到开发测试人员误操作数据的情况，可以使用数据闪回工具进行回退，我们借助 TiDB binlog 开发了闪回工具，对 binlog 的内容做反转，生成数据恢复 SQL，供 TiDB 数据恢复使用。

六、未来规划

6.1 故障的一键分析

分布式数据库与单机不同， TiDB 组件比较多，可供调整的参数有数百个，各个组件之间的网络通信、某个组件资源不足、SQL 复杂等都可能导致出现性能问题，后续计划将 TiDB 诊断做成自动化和智能化，目前已经通过改造 TiDB server 源码，完成了 TiDB 的全链路 SQL 收集和分析，这将作为未来故障一键分析的基础。

6.2 基于 HDD 硬盘测试

TiDB 所有的优化都是基于 SSD 来做的，高性能意味着高成本。我们还是会面临数据量比较大，但写入和查询都比较少，响应要求不高的场景。我们目前已经完成基于 HDD 硬盘的测试，选择的机器配置为 12 块 10T HDD 硬盘，单机部署 12 个 TiKV 实例，这种架构已经在小范围应用。

6.3 同城双中心自适应同步方案 DR Auto-Sync

DR Auto-Sync 正处在高速迭代的周期中，后续版本将会有一系列高可用和容灾能力的加强。从 5.3.0 开始将支持双中心对等部署，藉此获得快速恢复多副本的能力，我们也在保持关注中。

运维

携程持久化 KV 存储实践

【作者简介】布莱德，携程软件技术专家；蔚浩，携程资深软件工程师；小董，携程技术培训生。

一、背景

过去几年，携程技术保障部门在 Redis 治理方面做了很多工作，解决了运营上的问题，在私有云上也积累了丰富的经验。后又通过引入 Kvrocks，在公有云上实现降本增效的目的，从而支撑了公司的国际化战略。

与此同时，国内业务部门存在降低基础建设成本的客观需要，有些业务方期望提供一种非传统关系数据库来解决某些高性能海量空间的业务需求，并在此基础上支持特殊定制化以面对后疫情时代的挑战。这些变化使我们开始思考，是不是可以参考公有云上的思路，在私有云上构建一种持久化数据库，来满足业务方对高性能、低成本、海量、持久化的需求。

二、面对的问题

回顾之前在公有云上的方案，目的明确。因为公有云的内存较贵，我们将 Redis 的数据存在 SSD 上来降低成本，选型了 Kvrocks，并自研实现支持 Redis 的复制协议，将公有云上的成本降低了 60%（图 1）。

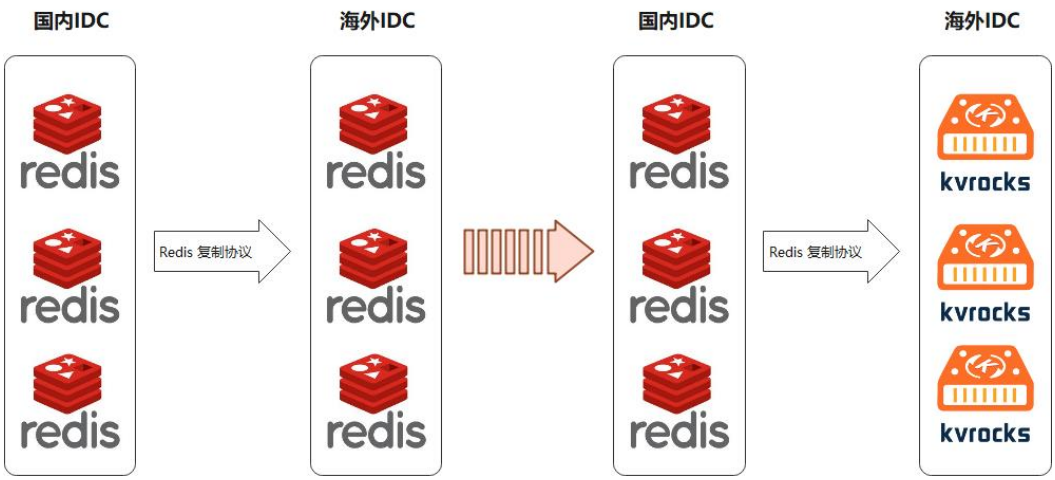


图 1

随着业务发展和 Redis 集群的日益增长，需求更加多样化，需要在私有云上同样能有一种持久化的 KV 存储系统来提供服务，包括：

1）KV 存储和读写的场景，Redis 能提供的存储上限过低，需要有大容量的 KV 存储系统；
2）数据持久化，而不是像 Redis 那样重启数据即丢失；

3）节约 Redis 的使用成本，毕竟私有云上的 Redis 集群非常庞大；
4）提供类似 selectforudpate 的语义来实现库存之类字段的扣减，而不是依赖外部的一些组件，比如分布式锁；
5）数据能提供相比 Redis 更高的一致性，比如支持同步复制。

我们仔细分析业务需求和业界可选的方案，以期望找到一种持久化的 KV 数据库，能兼容 Redis 满足大容量和成本降低的需求，而又不局限于 Redis，能提供更多样化的能力来支撑业务的诉求。

三、调研和选择

我们调研了业界大部分的 NoSQL/NewSQL 数据库，主要考虑以下几个方面。

是否为业界主流

主流有两层含义：第一，是否流行，比如 github 上的 star 数，是否是顶级开源基金的项目，或是否有大厂背书；第二，其理念是否主流，如现在使用最广的关系型数据库 mysql，以及 newsql TIDB，其相关概念如半同步复制，GTID，raft，计算存储分离等概念都比较深入人心。

是否有成熟的中间件

中间件成熟是非常重要的一种能力，一旦选择了一种不合适的数据库，中间件相关的路由，打点，监控，降级，熔断，DR 切换等每一项都需要投入大量的人力物力来做，此外稳定的中间件也是需要长时间打磨才能被业务方信赖，如果能复用现有中间件的大部分能力，能节约大量人力物力。

集群运维治理配套是否完善

选择一种 KV 数据库，除了中间件外，治理相关的如集群扩容，缩容，实例的迁移，资源利用率等一样要考虑进来。无论哪种数据库，部署后的运维治理相关，能复用现有的能力最好，如果不能复用，需要考虑：扩容到 10 倍需要多久时间，是否可以缩容？是否好迁移，对业务透明？大规模部署后，资源利用率是否可以提升？

性能是否满足要求，是否支持 10X 的扩展

上面说的这几点，如果都满足，但性能不满足或者不支持 10X 扩展，那也将一票否决。性能也是重要考量的一块，希望找到一种性能优异的 KV 数据库。

是否可以二次开发，独立演进

对于携程这样体量或相似体量的公司来说，持久化 KV 的数据库大多有自研的或基于开源二次开发的数据库，比如美团的 Cellar，饿了么的 Tidis，360 的 pika 等，我们同样需要选择一种易于二次开发或方便扩展的数据库，来开发自定义的特性支撑业务。

调研的过程受制于篇幅限制，不再一一展开，最终我们继续选择了 Kvrocks 来作为治理演进的对象，其他的 NoSql/NewSql 有各种不足，而 Kvrocks 受益于 Redis 运维治理的成熟，可以复用现有的大部分 Redis 中间件和运维治理的能力，在携程与 Redis 几乎无差异的部署/使用方式，当下无疑是最适合的一种持久化 KV 数据库。

四、从 Kvrocks 到 TRocks

经过不断的开发迭代和使用，最终我们将新系统命名为 TRocks（Trip+Kvrocks），作为携程自己的持久化 KV 数据库。相比于原来的 Kvrocks，除了与 Redis 可以互通协议互为主从外，主要是基于以下几个方面的改进。

4.1 功能增强

独占锁

一些业务方存在着流程协调，执行顺序的限制，往往会需要使用分布式锁，比如扣减库存的逻辑。常见的方式是引入一个第三方的分布式系统，将锁标识存储在那里用于共享访问，以达到锁的目的。

这样做虽然常见，但也有一些问题，首先需要引入额外的系统，并单独考虑各种异常情况的处理，增加了整个应用的复杂度。其次标识位往往有一定含义或者能与当前业务数据做关联，这就相当于额外存储了一份业务数据，存在一定的安全隐患。同时多个应用可能共用一套外部分布式系统来处理锁，这就无形中增加了系统的访问压力，一旦出现问题将影响多个依赖方，缺乏隔离性。

为了解决此类问题，TRocks 在内部实现了基于 Key 力度的锁功能，将其分布式部署并作为应用的业务数据库时，其本身就拥有了分布式锁的能力（图 2）。对锁的处理和业务数据在一起，无需引入多余的系统，降低复杂度，帮助业务方专注于业务代码的开发。

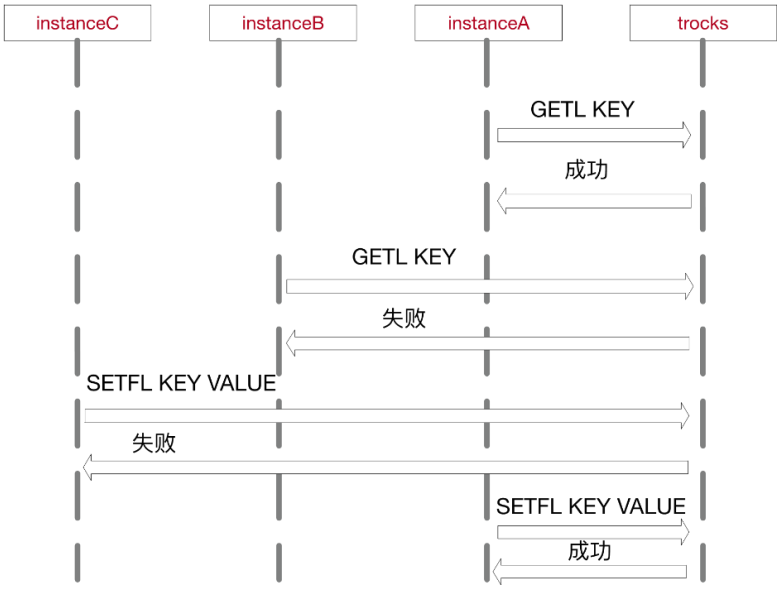


图 2

为了保证请求的唯一性和类似 raft 那样支持幂等重试的功能，每个请求需要带上标识唯一性的 clientid 和自增 seq，这些 metadata 和本身的 data 会被当成一个 writebatch 写入到 rocksdb 中，后续还会同步到 slave 上，从而保证整条链路上请求的原子性。

复合命令

由于 Redis 命令本身的限制，有些业务方反馈实现一个功能，比如对 hash key 进行超时处理需要进行 2 次操作，一次设置值，一次设置超时。虽然中间件将这层逻辑封装之后对外只提供一个 api，但内部执行仍然是 2 个命令，可能存在原子性问题。TRocks 针对这种情况增加了一些复合功能的命令，调用这些命令可以实现相同的效果并保证原子性，同时这些功能对用户是透明的，直接调用客户端相应 api 即可使用。

4.2 可用性增强

可调一致性

Kvrocks 本身的主从复制逻辑与 Redis 相似，都是通过异步方式进行的。在这种方式下，如果出现网络断开或者 master 宕机，数据还未来得及同步，就会出现数据丢失的情况。为了避免此类问题，TRocks 加入了类似 Mysql 的半同步复制来提高数据的一致性。我们可以通过打开半同步方式并指定至少需要参与的半同步 slave 的数量来启用该功能，提高灾备能力。

例如一个 1 主 4 从的集群，设定需要等待任意 2 台 Slave 响应。

如图 3 所示，当满足响应的 slave 为 2 的时候，半同步即可认为完成，即使此时另外两台 slave 可能还未完成同步工作。

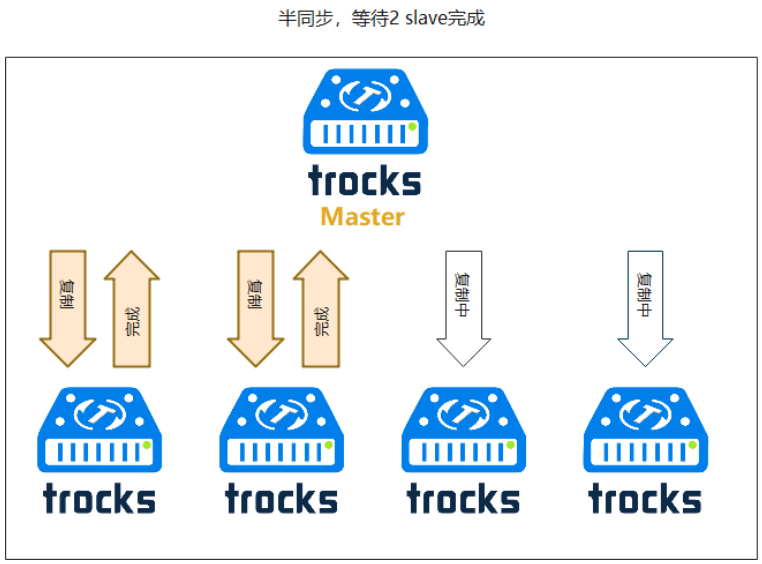


图 3

但这种方式在多机房部署的情况仍然可能存在问题。因为距离的关系，相同机房的数据传输速率会更高，所以 master 复制到和其在同一个机房的 slave 通常情况会更快（图 4）。

半同步，等待2 slave完成

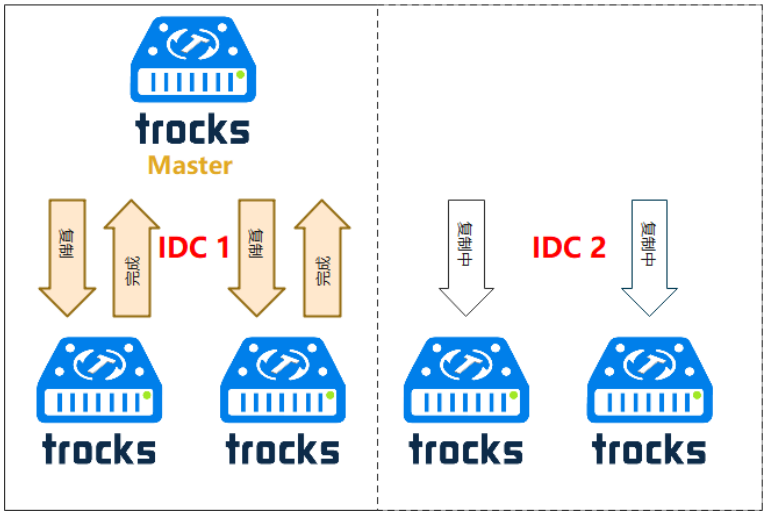


图 4

这样就很容易发生同机房的 slave 数据复制的进度要快于异地机房的 slave。如果发生机房级的故障，导致 master 所在集群的服务全都无法正常工作，这个时候就可能发生数据丢失。

为此我们在半同步复制的基础上增加了 IDC 模式，使得即使初始条件已经满足，也需要至少存在相关 IDC 的 slave 反馈才能完成整个复制流程。IDC 模式有两种，本地复制和异地复制。

以异地模式为例，如果返回 slave 的数量满足条件，且包含至少一台来自于 master 所在机房不同的 slave，则半同步复制完成。如果当前响应中未包含非 master 集群的 slave，则继续等待，直到 master 接收到一台来自异地的 slave 的反馈，半同步才能完成（图 5）。

半同步，等待2 slave完成（异地模式）

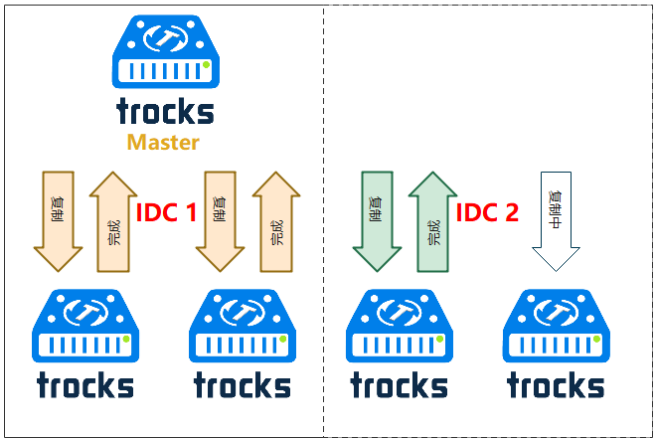


图 5

尽管异地模式数据的安全性更高，但也会影响整个系统的性能，这个性能差正常情况下取决于不同机房之间的网络延迟。基于对性能和数据可用性的不同要求，使用方可以酌情选择全异步复制（即关闭半同步），半同步 & 半同步（本地）复制或者半同步（异地）复制。

全量同步复制抑制

上面说到异步复制在异常情况下可能存在数据缺失的情况，如果再加上运维系统对主从关系的调整，就会发生数据冲突。而我们目前 TRocks 的版本还在快速迭代中，希望每次升级版本能够对用户透明，然而事实并非如此。

假设存在 master A 和 slave B，正常情况下 A 和 B 的数据是保持一致的（绿色部分），但当 A 发生宕机的时候，B 可能还未同步到 A 的最新数据，这时 B 的数据不再增加。但随后哨兵发现 master 无法访问，就把 B 提升为 master 并开始处理写入数据（蓝色部分）。当一段时间后，A 系统恢复，重新加入进集群，此时 A 会变为 masterB 的 slave，并尝试从 B 中同步数据，这里就可能存在冲突区（图 6）。

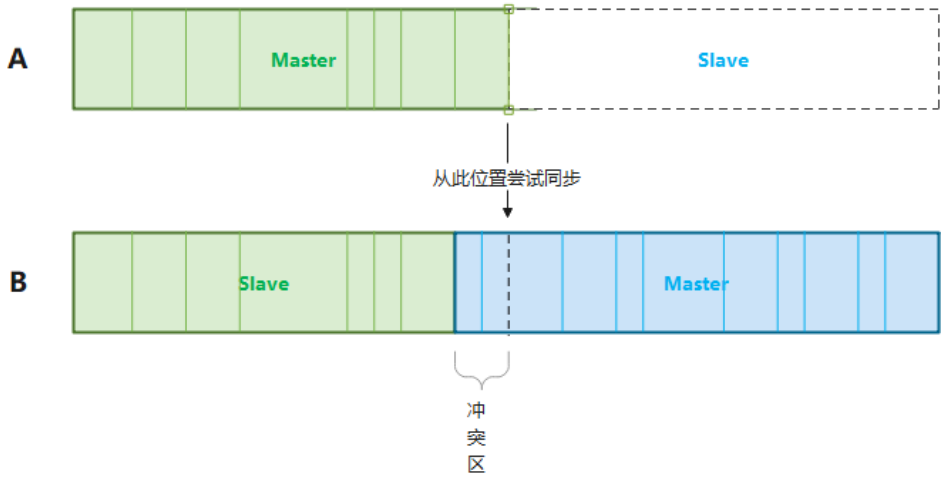


图 6

按照 Kvrocks 初始的复制逻辑，A 会认为自身数据存在问题，并放弃全部数据然后从头开始进行全量同步 B 的数据。这个行为本身没有问题。然而实际生产环境下，如果数据量很大的话，全量同步的耗时会比较长，而硬盘相比内存的带宽至少小两个数量级，因为我们的实例都是容器化部署，这有可能导致灾难性的后果，A 在同步数据的时候会产生大量的 IO，从而可能会影响 A/B 所在的宿主机上的所有的实例。

在数据一致性要求没有那么高的场景中，仅仅因为可能的几条数据不一致就重新同全量同步，代价非常昂贵。所以我们希望在非强一致性条件下，系统可以容忍极少量的数据差异，尽可能避免全量同步以便充分利用资源。

我们的方案是当检测到数据不一致的时候，主从之间会进行交互协调，计算出冲突区的范围，并从冲突区之后第一条数据开始进行同步。为什么不是直接从冲突区后面开始同步？这里需要有个概念，TRocks/Kvrocks 的数据都是追加形式的，增删改都会在 log 文件中追加一条记录，并提供起始位置（Sequence），对应不同的 Redis 类型的记录会有不同的长度（Count），比如一条 SET 指令对应的 Sequence 会累加 1，而 HSET 指令会累加 2。从 Sequence 到 Sequence+Count 就是一条记录的数据范围。当重新同步的时候，冲突区的结束位置如果处于正常数据的中间，这样是没有办法取得完整数据的，所以需要从冲突区后第一条数据开始。

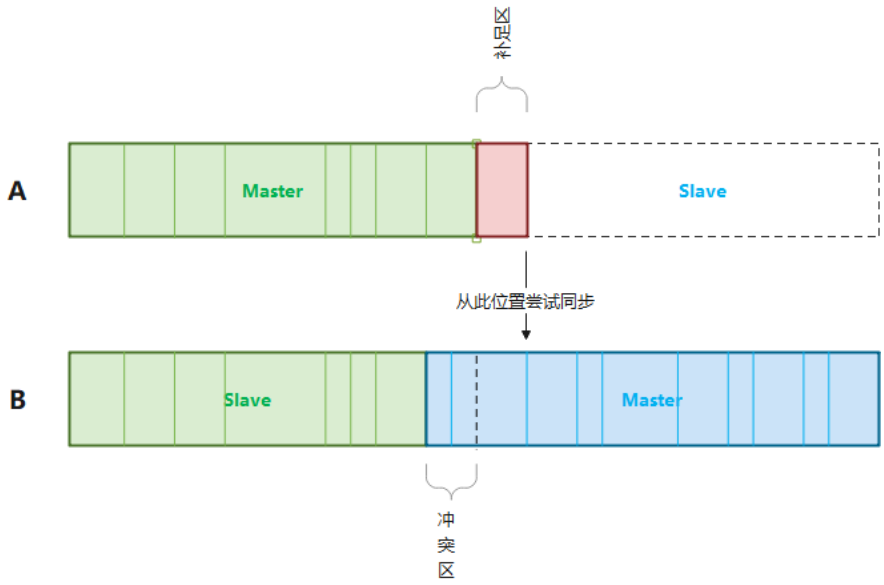


图 7

而冲突区与同步开始之间的区域是补足区（图 7），我们通过插入空白数据来进行填补，所以对于 A 和 B 来说，他们之间不一致区域是冲突区和补足区的总和。而对于冲突的部分，我们会记录下两边的差异，真有差异发生时，参考 git 解决冲突的思路，将数据的选择权交给用户。

上线该 feature 后，版本的升级就变得比较轻松，大部分情况下版本升级只是一次实例的拉出重启拉入，实例也是秒级 up，升级过程也基本上对业务做到了透明。

在解决此问题的同时，我们也注意到 master/slave 数据是对齐的某些情况下也会发生全量同步，检查下来发现是 pub/sub 命令的问题。这个命令是哨兵用于订阅服务消息的，但 Kvrocks 的 pub/sub 是一个写操作，这样就会造成持续性的数据写入从而累加 rocksdb 的 Sequence，这样如果一个 slave 宕机后恢复，还没来得及与 master 同步却被哨兵写入了一条无关紧要的 pub 消息，累加了 Seq 从而触发了不必要的全量同步，但实际上该功能并非必须，所以我们修改 Kvrocks 处理哨兵 pubsub 消息的规则，不去写之后这个命令只工作在内存中，自然不会累加 rocksdb 的 Sequence，杜绝这种情况全量同步的可能性。

4.3 运维治理能力增强

水平扩缩容

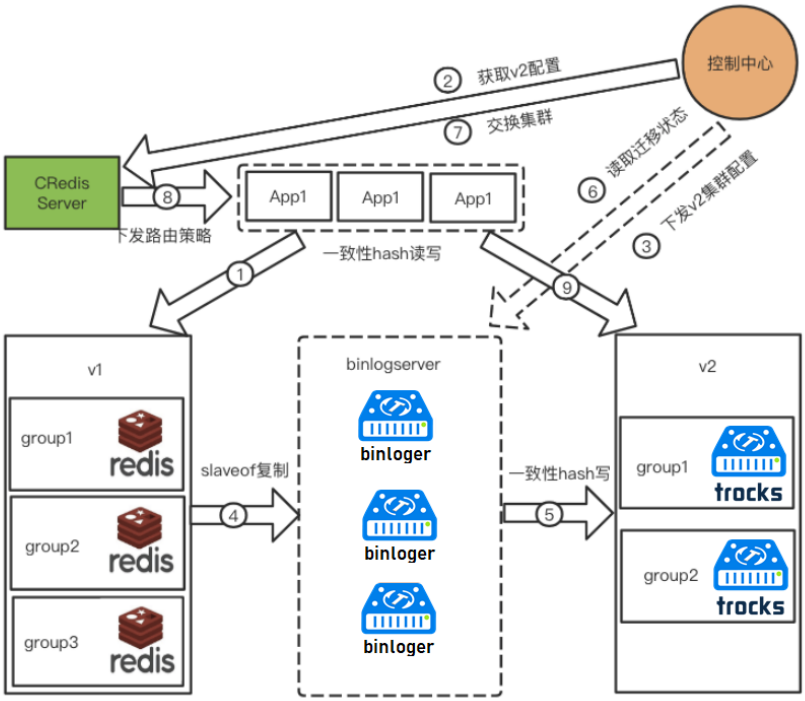


图 8

在之前的 Redis 治理演进之路文章中，我们介绍了一种新的扩缩容方案来解决 Redis 集群版本升级和扩缩容的问题（图 8），参考同样的思路，我们继续改造 BinlogServer 来实现 TRocks 的集群的水平扩缩容，这套方案实际上不仅解决了扩缩容的问题，同时也解决了 Redis 到 Redis 的数据迁移，TRocks 到 TRocks 的数据迁移，Redis 与 TRocks 之间的互相迁移，也可以帮助用户平滑的从 Redis 的访问过渡到 TRocks 的访问。

然而相比 Redis 扩缩容基本不需要考虑内存带宽，硬盘带宽太窄，而数据迁移的时候流量太大。由于所有数据最终都需要在新集群上刷盘，导致迁移过程中目标集群的磁盘读写会非常大，又由于我们都是容器化部署，大量的磁盘读写也可能会影响到统一宿主机上的其他无关的应用，所以我们调整了 TRocks 的写入限流设置，以避免大量写入影响磁盘性能，同时修改了 BinlogServer 加入了限流功能，平缓数据传输的速率。

哨兵多机房部署

为了保证 TRocks 集群可以跨机房容灾，哨兵需要部署在多个机房中，目前我们是三机房部署。如下图（图 9）：

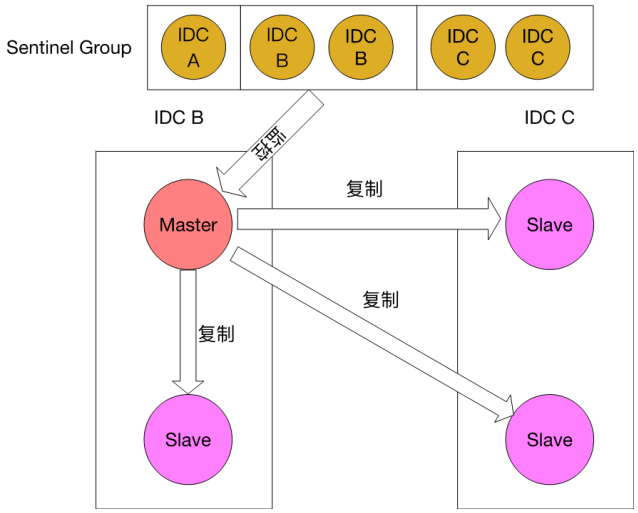


图 9

在部署的时候，遇到了一个问题，我们发现哨兵之间经常无法选出 leader，需要等下一个选举周期(6 分钟)才能重新选出，导致长时间无法确定 TRocks master。这个问题本身跟 TRocks 没有太大关系，只是实际使用中对我们故障处理带来了不小麻烦。

出现无法选出 leader 的原因是多个哨兵同时发起选举希望成为 leader，导致最终每个哨兵都选择了自己，无法达成共识。查看源码发现官方已经为发起选举前设置了随机的间隔时间（50~100ms），但实际操作中发现这个随机间隔反而增加了发生选角失败的可能，考虑应该是随机时间太短导致，所以我们将随件间隔修改为 100~200ms，同时在哨兵发现 master 宕机之后就立即发起选举来尽可能规避无法选主的问题。

五、一些数据

5.1 性能数据

TRocks 在内网上线后，在各个业务线都得到了广泛的使用，排除公有云的部分，私有云上已经有将近 2K 的实例，10T+的数据量，下图（图 10，图 11）可以看到同样的数据写 TRocks 和 Redis 的性能对比。平均响应时间，99.9%在同一个水平，并且我们还可以看到，得益于自定义的命令，同样的功能相比 Redis 更加简洁。

Name	添加告警	Count	Sum	Self	Failure	Failure%	Sample Link	Min(ms)	Max(ms)	Avg(ms)	SAvg(ms)	P95(ms)	P99.9(ms)	QPS
[:: show ::] Type:Redis	⚠	129,589,606	610.5m	607.6m	0	0.0000%	[:: show ::]	0	1,142	0.3	0.3	0.6	3.8	159,819.3
[:: show ::] [redacted]hmsetflex	⚠	5,136,375	51.2m	51.2m	0	0.0000%	[:: show ::]	0	270	0.6	0.6	1.2	3.6	6,334.6
[:: show ::] [redacted]hmget	⚠	119,316,844	508.5m	505.7m	0	0.0000%	[:: show ::]	0	1,142	0.3	0.3	0.6	3.8	147,150.2
[:: show ::] [redacted]hgetl	⚠	5,136,387	50.8m	50.7m	0	0.0000%	[:: show ::]	0	276	0.6	0.6	1.2	3.5	6,334.6

图 10

[:: show ::] Type:Redis	⚠	160,405,414	776.3m	772.5m	0	0.0000%	[:: show ::]	0	1,118	0.3	0.3	0.4	3.5	178,106.1
[:: show ::] [redacted]as:Transaction	⚠	5,692,724	58.6m	58.5m	0	0.0000%	[:: show ::]	0	345	0.6	0.6	0.7	4.1	6,320.9
[:: show ::] [redacted]as:native.watch	⚠	5,692,703	54.6m	54.6m	0	0.0000%	[:: show ::]	0	277	0.6	0.6	0.7	2.8	6,320.9
[:: show ::] [redacted]:hmget	⚠	131,941,820	551.3m	547.7m	0	0.0000%	[:: show ::]	0	1,118	0.3	0.2	0.4	3.7	146,501.5
[:: show ::] [redacted]:native.hmget	⚠	5,692,710	55.8m	55.8m	0	0.0000%	[:: show ::]	0	273	0.6	0.6	0.7	2.6	6,320.9
[:: show ::] [redacted]:pexpire	⚠	5,692,732	55.9m	55.9m	0	0.0000%	[:: show ::]	0	255	0.6	0.6	0.7	2.6	6,320.9
[:: show ::] [redacted]:tx.hmset	⚠	5,692,725	1.0s	1.0s	0	0.0000%	[:: show ::]	0	105	0.0	0.0	0.0	0.0	6,320.9

图 11

根据我们跟业务方压测，一台 40C 和 2 块 RAID0 的 SATA SSD 在保证良好响应的前提下（99.9%<10ms）约能提供读写的 QPS 为 8-10W，其中 value<1k。而如果换成 NVME SSD 这个 QPS 可以提升 3-5 倍。

5.2 成本数据

假设 TRocks 都是容器化部署，并且一台 40C 的宿主机上可以部署 20 个实例，每个实例大小为 40G，因为 TRocks 相比 Redis 有不小的压缩功能（约 3-7 倍的压缩率），如果将 Redis 的数据导过来可以平稳运行，那么 TRocks 相比 Redis 约可以节约 90%的成本。

既然能省这么多成本，是否所有的 Redis 都可以用 TRocks 来代替，我们是否需要将私有云上所有的 Redis 都替换成为 TRocks?

答案都是否定的，也不是我们推广 TRocks 的初衷，原因有以下两点：

1)如上文所提到的,我们希望 TRocks 能拥有 Redis 的大部分能力,而又不仅仅局限于 Redis,希望它更是一个通用的 KV 数据库，能提供更多样化的能力来支撑业务的诉求。

2）硬盘的带宽与内存有 2 个数量级的差距，而这些先天不足也无法满足某些 Redis 场景的需求。比如大 Key（>100K）响应和 Redis 还是有一定的差距，此外某些数据量小并且单个实例访问 QPS 较高的实例，用 TRocks 来替换也并不合适，因为规模化运维治理，我们需要考虑整个宿主机和每个实例是否能平稳运行，一般来说单个实例>10G，QPS<5K 是比较适合的。当然 NVME SSD 可以极大缩短大 Key 的响应时间和提升单个实例 QPS 的上限。

六、未来规划

6.1 复合命令增强

我们调研发现，业务经常为了获取一条数据，需要多次查询 TRocks，类似二度人脉的取数据逻辑，多次的网络 IO 会导致耗时增加，而设计通用的命令来支持业务需求，减少网络 IO 变得非常重要,此外还有些用户询问 TRocks 的 hash 类型中的 subkey 是否也可以实现过期。由于 hash 功能目前仍然是遵照 Redis 的规则，所以现在是按照整个 hash key 一起过期而不能实现内部数据项的过期。这个需求是有一定价值的,未来我们会通过提供一个特殊的 hash 结构来实现此类功能。

6.2 引入 checkpoint

Kvrocks1.X 在进行全量复制时，master 会生成硬的 backup，会拷贝文件产生大量的 IO，而官方 2.0 版本已经用 Rocksdbcheckpoint 解决了这个问题，我们也已经将 2.0 版本 merge 过来测试，准备适时升级上线。

6.3 使用 NVME SSD

目前携程的大量 TRocks 还是跑在 SATA 接口的 SSD 上,而据我们的测试下来两块 SATA raid0

的带宽大约为 800MB/S，导致硬盘非常容易跑满，相比之下，NVME SSD 的带宽基本都是几 G 起步，并且我们测试下来 NVME SSD 在小的压力下，对于 SATA SSD 性能有 3-5 倍的提升，而对于大 Key 的情况（超过 100K）和大的压力下，NVME SSD 的性能提升可以高达 10-100 倍。因此我们已经计划将 SATA SSD 全换成 NVME SSD，进一步提升 TRocks 的性能。

6.4 回馈社区

在 TRocks 开发过程中，我们一直受益于 Kvrocks 社区开发者的帮助，并跟社区保持着紧密沟通，也提交过比较多的 PR/issues 给社区。希望后续能更好回馈社区，将一些独立的比较大的 feature 分享出来，目前半同步复制的 feature 已经提交给社区 review，希望可以早日 merge 进主分支。

高效线上问题排查——套路化和工具化

【作者简介】丰富，携程高级技术专家，在性能优化、问题排查方面有较多的研究。

一、前言

线上问题排查相比于 coding，是一个低频的工作，很多人不会经常遇到。一旦需要进行问题排查的时候，往往是重要且紧急的，因此问题排查的效率，就显得尤为重要。有些线上问题，比较直观，比如磁盘使用率高、网络流量高这种，借助合适的工具很快能定位到原因；但对于一些复杂的问题，如系统 Load 高、RSS 占用高、内存溢出等，需要结合多方面的数据才能定位到原因。这时候，需要有正确的解题思路，并辅以合适的工具，才能高效地解决问题。

目前业界排查问题的优秀工具还是挺多的，比如国内阿里开源的 Arthas、PerfMa 开源的为终结性能问题而生的 xPocket，Java 官方的 JMC（JDK Mission Control）、Eclipse 的 MAT（Memory Analyzer Tooling），以及我一直很推崇的神器 Async-Profiler。上述只是列举了一些比较流行的开源工具，商业工具如 jProfiler、YourKit 等也都建立了稳定的用户群体，这些工具功能各有差异。当然这不是本文描述的重点，就不详细展开了。

在对这些工具进行横向对比时我们发现，他们的目标都是为了解决一些特定的问题，如果我们有清晰的问题排查思路，结合这些工具，可以很快解决问题。而对于一些复杂场景，尤其是一些陌生的复杂问题，在没有头绪的情况下，纵然有各种神兵利器，也无计可施。线上问题排查犹如开车，老司机驾轻就熟，新手则手忙脚乱。当然如果新手有老司机加以指点，也可能很快地解决问题。

但问题是，这种老司机并不常见，也不可能时刻都能帮你。我们可以去网上查阅其他人总结的问题排查套路，再结合我们自己的场景，去尝试解决问题，我也是经常这么干的。但这种方式效率依然不高，原因有三个：

1）信息检索的成本：我们需要花时间去翻阅资料，去跟自己的场景匹配以判断是否适合自己；
2）试错的成本：有些资料不适合我们的场景，我们按照资料去尝试，有可能被带沟里去，浪费时间；
3）问题排查需要借助于一些第三方工具，而这些工具在生产环境需要安装、配置和使用，也需要花较多的时间成本。

针对线上问题排查的特点和现状，我们是否可以构建一个系统，这个系统会针对各种线上问题的排查形成一个知识（套路）库，针对每一种问题，都有对应的套路和自动化工具帮助我们去定位问题。本文将结合一个比较有代表性的线上问题的排查过程，来探讨这种方式的可行性。

二、问题排查的套路化

本章将以 RSS 占用高为例来对问题排查的套路化进行说明。RSS 占用高是很多人遇到过的问

题，这个问题涉及的因素比较多，比较有代表性。当然在开启了 Swap 的运行环境中，Swap 高也是 RSS 高的一种表象，殊途同归。

RSS 是 Resident Set Size（常驻内存大小）的缩写，用于表示进程使用了多少内存（RAM 中的物理内存）。如果我们遇到进程 RSS 接近服务器的物理内存，那就意味着你需要关注应用的健康程度了，这意味着应用后面很有可能出现 OOM 的问题，比如进程被 OOM killer 杀死，或者容器重启，或者因使用 Swap 而速度变慢。

针对 RSS 高的问题，首先我们需要知道的是，Java 进程消耗的内存绝不仅仅是你设置的 Xmx 或堆内存的用量这么简单，Java 进程占用的内存主要分为 2 大部分：on-heap（堆内内存）和 off-heap（堆外内存）。而堆外内存又包含 JVM 自身消耗的内存、JVM 外的内存。所以，后续的排查思路我们也是按照堆内内存、JVM 内存、JVM 外内存 3 个方向来顺序展开。

2.1 堆内存是否太大

首先要确认一下 Java 应用的堆内存是否太大，因为 JVM 自身也会消耗一些内存，所以你至少需要预留出部分内存存给 JVM 使用。如果应用涉及较多的网络通信，那还需要预留一些内存给堆外使用，所以一般来说你的堆内存最多为服务器物理内存的 75%（经验值，需依据应用自身特点调整），如 4G 内存服务器，那么堆内存最大为 3G。

堆内存用量的查看手段非常多，相信各个公司的基础架构团队都提供了可视化的监控手段，当然也可以通过原生命令 jcmd <pid> GC.heap_info 查看，如图 1 所示：

```
[redacted]$ sudo -u deploy /usr/java/jdk11/bin/jcmd 38 GC.heap_info
38:
 garbage-first heap   total 28311552K, used 19594395K [0x0000000140000000, 0x0000000800000000)
  region size 32768K, 44 young (1441792K), 3 survivors (98304K)
 Metaspace       used 151147K, capacity 153284K, committed 154368K, reserved 391168K
  class space    used 17075K, capacity 17850K, committed 18176K, reserved 253952K
```

图 1

如果 Java 进程的堆内存用量已接近或超过物理内存的 75%，那么基本可以确定堆内存用量过大。这时可以调小 Xmx 来控制堆内存用量。如果 Xmx 不能减小，可以通过 dump 堆内存+MAT 或 JFR（Java Flight Recorder）+ JMC（JDK Mission Control）来分析内存占用/分配情况，通过程序调优来减少堆内存用量。

如果到此 RSS 占用呈稳定趋势，我们就可以告一段落了，否则要继续后面的步骤。

2.2 是否存在大量 ARENA 区

如果堆内存不大，那么继续排查非堆内存。首先去看一下 ARENA 区，在高并发的应用中，往往 ARENA 区占用的内存会比较多。为什么先看 ARENA 区的内存占用呢？是因为这个步骤是不需要重启 JVM 进程就可以完成的。至于 ARENA 是什么以及作用是什么，可以读一下这篇文章：https://blog.csdn.net/maokelong95/article/details/51989081。

这里我们直接进入排查问题环节。执行如下命令：sudo -u <user> pmap -x <pid>|sort -gr

-k2 |less，如果存在大量大小为 65536 或 60000 左右的内存区域，则很大可能是 ARENA 区域占用了太多的内存，如图 2 所示：

```
]$ sudo -u deploy pmap -x 40|sort -gr  -k2 |more
0000000180000000 27281408 27281408 27281408 rw---   [ anon ]
00007f7884400000  479232  479232  479232 rw---   [ anon ]
00007f786803f000  462596  426520  426520 rw---   [ anon ]
0000000801200000  235520       0       0 -----   [ anon ]
00007f78bb957000  137892   15564       0 r--s- modules
00007f78a1800000  106496  106496  106496 rw---   [ anon ]
00007f78b68cd000   82472       0       0 -----   [ anon ]
00007f78afff4000   70948       0       0 -----   [ anon ]
00007f785c000000   65536   65536   65536 rw---   [ anon ]
00007f71e4000000   65536   65536   65536 rw---   [ anon ]
00007f70b0000000   65536   65536   65536 rw---   [ anon ]
00007f70ac000000   65536   65536   65536 rw---   [ anon ]
00007f70a8000000   65536   65532   65532 rw---   [ anon ]
00007f70a4000000   65536   65536   65536 rw---   [ anon ]
00007f7050000000   65536   65536   65536 rw---   [ anon ]
00007f704c000000   65536   65536   65536 rw---   [ anon ]
00007f7048000000   65536   65536   65536 rw---   [ anon ]
00007f7044000000   65536   65536   65536 rw---   [ anon ]
00007f7040000000   65536   65536   65536 rw---   [ anon ]
00007f703c000000   65536   65536   65536 rw---   [ anon ]
00007f7038000000   65536   65536   65536 rw---   [ anon ]
00007f7034000000   65536   65536   65536 rw---   [ anon ]
00007f7030000000   65536   65536   65536 rw---   [ anon ]
00007f702c000000   65536   65536   65536 rw---   [ anon ]
00007f7028000000   65536   65536   65536 rw---   [ anon ]
00007f7024000000   65536   65536   65536 rw---   [ anon ]
00007f7020000000   65536   65536   65536 rw---   [ anon ]
00007f7828000000   65508   65508   65508 rw---   [ anon ]
00007f76d4021000   65404       0       0 -----   [ anon ]
00007f76c8021000   65404       0       0 -----   [ anon ]
00007f76c4021000   65404       0       0 -----   [ anon ]
00007f76b8021000   65404       0       0 -----   [ anon ]
00007f76b0021000   65404       0       0 -----   [ anon ]
00007f7690021000   65404       0       0 -----   [ anon ]
00007f7684021000   65404       0       0 -----   [ anon ]
00007f7674021000   65404       0       0 -----   [ anon ]
00007f75d8021000   65404       0       0 -----   [ anon ]
00007f7588021000   65404       0       0 -----   [ anon ]
00007f7584021000   65404       0       0 -----   [ anon ]
00007f7580021000   65404       0       0 -----   [ anon ]
00007f757c021000   65404       0       0 -----   [ anon ]
00007f753c021000   65404       0       0 -----   [ anon ]
00007f7530021000   65404       0       0 -----   [ anon ]
00007f752c021000   65404       0       0 -----   [ anon ]
00007f751c021000   65404       0       0 -----   [ anon ]
00007f750c021000   65404       0       0 -----   [ anon ]
00007f74dc021000   65404       0       0 -----   [ anon ]
00007f74d4021000   65404       0       0 -----   [ anon ]
00007f74b8021000   65404       0       0 -----   [ anon ]
00007f7470021000   65404       0       0 -----   [ anon ]
00007f744c021000   65404       0       0 -----   [ anon ]
00007f742c021000   65404       0       0 -----   [ anon ]
00007f7414021000   65404       0       0 -----   [ anon ]
00007f7400021000   65404       0       0 -----   [ anon ]
00007f73f0021000   65404       0       0 -----   [ anon ]
```

图 2

这种情况下，最简单粗暴的办法是在 JVM 启动参数中增加配置：export MALLOC_ARENA_MAX=1。需要注意的是，上述的数值只能是 1，其他大于 1 的数值经实践证明是无法控制 ARENA 数量的。

2.3 非堆内存是否开销过大

如果前面 2 个步骤过后都没有发现问题，还有很多内存你不知道消耗在哪里了，那么我们开始第 3 步：开启 Native Memory Tracking。前面说过，Java 应用的执行，JVM 自身也需要消耗一些内存的，通过开启 Native Memory Tracking，我们就能知道 JVM 自身消耗了多少内存。

书归正传，通过修改 JVM 参数并重启 Java 进程开启 NativeMemory Tracking：

-XX:NativeMemoryTracking=detail。

进程重启后，可以通过 NMT 的一些子命令（summary/detail/baseline/diff）查看 Native Memory 的占用情况：sudo -u <user>jcmd <pid> VM.native_memory detail。

图 3 是在使用 baseline 建立了基线的情况下用 detail.diff 看到的各内存区的变化情况：

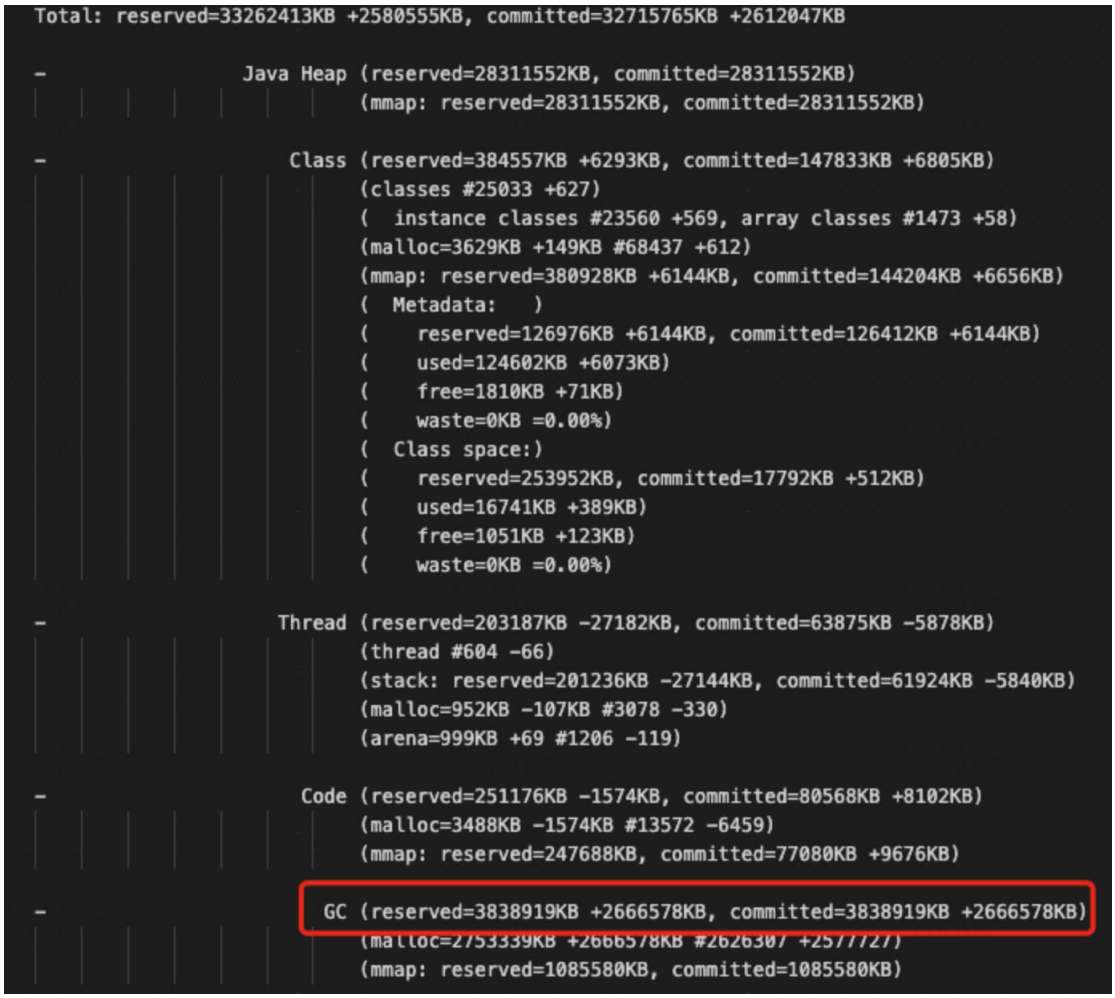


图 3

通过上图，可以看到 JVM 各个区域所使用的内存大小，主要包含了 Java Heap、Class、Thread、Code、GC、Compiler、Internal、Other、Symbol 等，各部分作用如下：

1）Class：加载的类与方法信息，其实就是 metaspace，包含两部分：一是 metadata，被-

XX:MaxMetaspaceSize 限制最大大小，另外是 class space，被-XX:CompressedClassSpaceSize 限制最大大小；
2）Thread：线程与线程栈占用内存，每个线程栈占用大小受-Xss 限制，但是总大小没有限制。在 x64 的 JVM 中，Xss 默认为 1024K，所以如果你的应用开启了 1000 个线程，那么这个 Thread 区占用将是 1024M，所以一般我们会把 Xss 设置为 256K 即满足要求；
3）Code：JIT 即时编译后（C1C2 编译器优化）的代码占用内存，受-XX:ReservedCodeCacheSize 限制；
4）GC：垃圾回收占用内存，例如垃圾回收需要的 CardTable，标记数，区域划分记录，还有标记 GC Root 等等，都需要内存。这个不受限制，一般不会很大，但也有例外，图 3 是 27G 的堆内存，使用 G1 垃圾回收器，你能看到 GC 区居然占用了 3.8G 的内存；
5）Compiler：C1 C2 编译器本身的代码和标记占用的内存，这个不受限制，一般不会很大；
6）Internal：命令行解析，JVMTI 使用的内存，这个不受限制，一般不会很大；
7）Symbol：常量池占用的大小，字符串常量池受-XX:StringTableSize 个数限制，总内存大小不受限制；

我们需要需要注意的是 Class、Thread、GC 几个区域的大小。图 4 是各种 JVM 垃圾回收器消耗内存的比例，注意这部分内存是堆内存之外的：

GC	Overhead, MB	%
Serial	7	0,3%
Shenandoah	38	1,9%
CMS	76	3,7%
Parallel	90	4,4%
G1	166	8,1%
Z	238	11,6%

OpenJDK 13 | Heap size = 2 GB

图 4

实践证明，G1 的内存开销甚至能占到堆内存的 20%，当然这不是本文要讨论的内容，感兴趣的读者可以去阅读《JVM G1 源码分析和调优》一书查看相关内容。

针对上面的各个区域大小做加法，看一下是否接近于 RSS 的大小，如果是，恭喜你可以到此结束了。后续你需要做的就是针对内存占用比较大的 JVM 区去做优化，这里就不详细介绍了。

如果不是，很遗憾，你进入到了最难啃的环节，继续往下看吧。

注：开启 NativeMemoryTracking 会造成 5%的性能下降，用完记得修改 JVM 参数并重启永久关闭，或者可以通过以下命令临时关闭：jcmd vm.native_memory stop <pid>。

2.4 堆外内存是否用量太多

堆外内存也是比较容易被忽略的一个区域，尤其是网络通信非常频繁的应用，这种应用往往

大量使用 Java NIO，而 NIO 为了提高效率，往往会申请很多的堆外内存。确认这个区域用量是否过大，最直接的方法是先查看是否是 DirectByteBuffer 或者 MappedByteBuffer 使用了较多的堆外内存。

如果你的服务器已经开启了远程 JMX，你可以通过 ops 提供的 jmx 查询工具去查询，也可以通过 jdk 自带的工具（比如 jconsole、jvisualvm）查询，如图 5 所示：

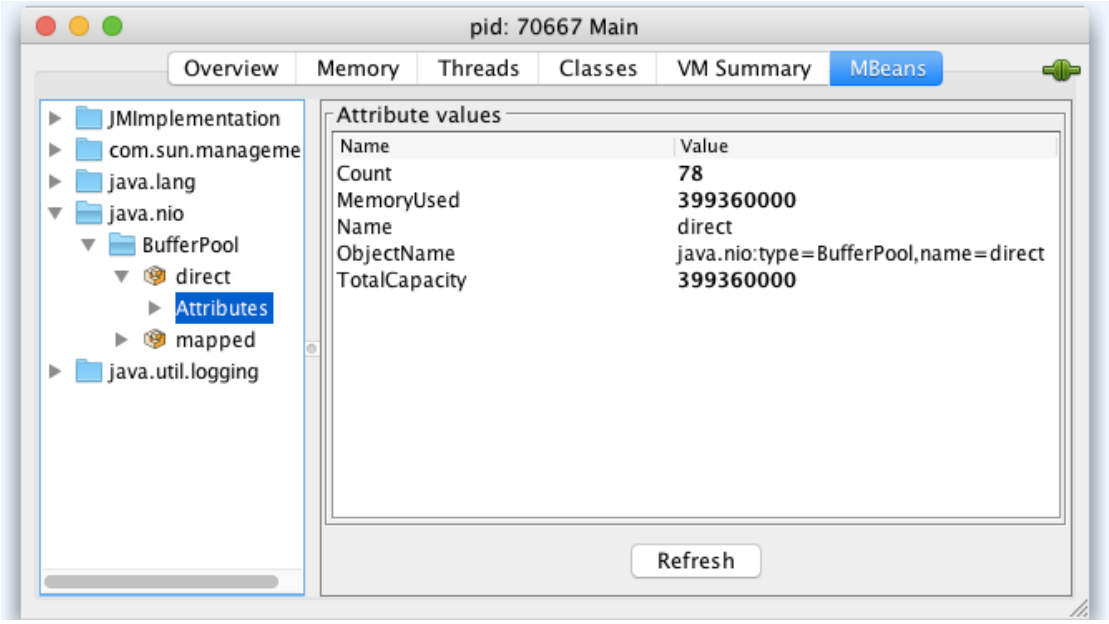


图 5

如果未开启远程 JMX，可以通过 jmxterm（https://docs.cyclopsgroup.org/jmxterm）工具在本地模式下查询“java.nio:name=direct,type=BufferPool”和“java.nio:name=mapped,type=BufferPool”两项内容确认用量。

如果确认上述堆外内存使用过多，那么可以通过在 jvm 参数中设置 -XX:MaxDirectMemorySize 这个参数控制一下，因为通过 DirectByteBuffer 分配的堆外内存，默认是不会控制这个区域的内存用量的。

如果上述内存用量不大，那我们就需要祭出终极杀器 jemalloc 来做进一步分析了。这里涉及的内容比较多，受限于文章篇幅限制，就不展开描述了。

通过 jemalloc 的收集到的数据，我们基本能够定位到堆外内存问题的原因。

2.5 总结

上述的 4 个步骤，基本能够解决大多数的 RSS 占用高的问题了。当然事无绝对，没有一种药是包治百病的。我们追求的基本目标是，通过问题排查的套路化，帮助工程师理清思路，少走弯路，以提高问题排查的效率。

三、问题排查的工具化

复盘一下上述的问题排查过程，我们用到了很多的命令和第三方工具，整个过程还是工程师驱动命令行和工具。如果对命令参数不熟悉，或本地没有安装相应的工具，那这种套路化的教程也只能在一定程度上提高效率。在这个套路的基础上，我们是否可以转变思路，以工具为主，工程师辅助，来提高排查问题的效率呢？让我们来尝试一下。

3.1 流程梳理

先来梳理工具的执行流程以及每个步骤需要做的事情，与第二节保持一致性，本节也划分成 4 个子步骤。

3.1.1 确认堆内存是否太大

第一步要做的事情比较多，梳理如下：

- Java 进程 pid：使用 jps -v 列出 java 进程列表，由用户选择具体的进程；
- 获取运行环境的物理内存和剩余内存（free -m）
- Java 进程的堆内存用量（jcmd GC.heap_info）
- Java 进程 GC 情况（jstat-gcutil）；

获取到上述信息后，判断堆内存是否太大。

3.1.2 是否存在大量 ARENA 区

通过 pmap 命令获取内存分配列表，辅以 awk 命令提取内存信息，据此判断是否存在大量 ARENA 区。

3.1.3 非堆内存是否开销过大

此步骤需要在 JVM 启动脚本中增加启动参数并重新启动进程，对于标准化的运行环境来说，在知道了启动脚本位置和启动命令的情况下，是可以通过工具来完成参数的修改和进程启动。如果不能知道启动脚本所在位置，我们可以复制当前进程的 JVM 参数来完成进程的启动。

当 JVM 进程启动完成，再次进入工具，我们就能借助于 Native Memory Tracking 的结果来判断当前环节是否存在问题。

3.1.4 堆外内存是否用量太多

此步骤的自动执行，需要安装第三方工具如 jxmterm、jemalloc。在生产环境访问外网受限的情况下，可以通过搭建内网资源服务器的方式来解决这个问题。通过一键安装脚本，我们能快速完成所依赖工具的安装和配置，剩下的就是让工具来收集和分析数据定位问题了。

3.2 工具实现

目前公司内部很多运维工具，都是采用 C+B/S 的方式实现，这种方式工程师不需要申请目标服务器权限就能够实现很多运维操作。但这种实现方式比较复杂，而我们的工具是带有实验性质，所以暂时使用 shell+工具包的方式实现，即使用 shell 脚本将主流程串起来，各节点使用的工具如果有缺失，yum 能安装使用 yum 安装，yum 不能安装的则提前下载内置到工具包中。

所有的准备工作完成后，编写脚本的工作就相对简单了，当然这需要用到很多的 shell 和 linux、java 命令，此处就不赘述了。脚本的最终运行效果如下：

```
>>>>>>>>>>RSS占用量大快速分析工具<<<<<<<<<<

数字:作为PID进行分析
字符:作为进程关键字查找到对应的PID进行分析
0:退出

请按上述提示输入:70755
进程pid:70755, 堆内存用量:18698(mb), 物理内存:32768(mb), 剩余内存:7256(mb)
Step1: 堆内存用量正常, 开始检查ARENA区内存占用...

Step2: ARENA区内存占用正常, 开始通过Native Memory Tracking检查堆外内存占用...

Class区用量(151 mb), 正常。
Thread区用量(81 mb), 正常。
Code区用量(80 mb), 正常。
GC区用量(3188 mb)偏大, 请关注。
Compiler区用量(6 mb), 正常。
Internal区用量(9 mb), 正常。
Other区用量(146 mb), 正常。
Symbol区用量(30 mb), 正常。
Tracking区用量(16 mb), 正常。
Step3: 堆外内存总用量:3712mb

Step4: BufferPool Direct用量: 141mb

是否继续使用jemalloc分析其他内存? [yes/no]
```

四、总结

通过针对 RSS 占用高问题的排查套路和排查工具的梳理，我们实现了一个简单的问题快速排查脚本。当然在这个过程中可以发现，很多问题的排查，都可以使用类似的思路来工具化，日积月累，就形成了一个问题排查的工具包。

以内存问题排查举例，我们积累了以下的快速工具，如图：

```
>>>>>>>>>>内存工具箱<<<<<<<<<<
1.堆内存使用情况
2.对象实例统计Top20
3.大对象Top20
4.dump堆内存
5.Jemalloc分析内存分配
6.内存分配火焰图(Async-profiler)
0.返回上级菜单
q.退出
```

这只是工具包的一部分，针对 CPU、磁盘、网络、GC 等问题，借助于 Arthas、Async-Profiler 等优秀的开源工具，我们都积累了很多快速工具，期望能帮工程师提高问题排查的效率。

如前面所讲，这种完全基于 shell 的方式，由于需要登录到目标服务器上操作，多数功能还需要有 sudo 权限，这有些许的不方便。另外，某些公司生产环境严格受限，那 shell 的方式就无法使用了。所以在此基础上，可以扩展成 Client+Server+Browser 的模式，让工程师在不登录到服务器的情况下，就能完成问题的排查。

到此，本文的内容就结束了，但我们的工具还在不断地积累中，在此也欢迎感兴趣的同学帮我们提供场景，我们不断丰富这个工具库。同时，受限于作者水平，文中内容难免有不当之处，也欢迎提出意见和建议。