Trip.com

2023年度合辑

携程技术

大前端/架构/大数据/人工智能/
数据库/质量保障

携程技术出品

“携程技术”微信公众号

分享，交流，成长

作为携程集团的核心竞争力，携程技术由数千位来自海内外的精英工程师组成，为携程集团业务的运作和开拓提供全面技术支持，并以技术创新源源不断地为产品和服务创造价值。

技术从来都不是闭门造车，携程技术团队会一直以开放和充满热情的心态，通过各种渠道和方式，和圈内小伙伴们探讨、交流、碰撞，共同收获和成长。

目录

序

“疾风知劲草，烈火见真金。”2023 年，是我们的挑战之年，也是我们的成长之年。全球旅游行业在经历了三年疫情的严重打击后，终于迎来了全面的复苏和爆发式的增长。我们多个业务线订单量均突破了历史最高值，随着五一、暑期、十一、春运多个业务高峰接踵而至，我们系统承受了前所未有之压力，面对压力，我们没有退缩，而是选择了挑战。我们进行了一系列的技术改进和创新，提升了系统的效率、可扩展性、稳定性和安全性，同时降低了成本。强有力地保障了业务的发展。

“读万卷书，行万里路。”我们知道，知识的力量是无穷的，分享的价值是无尽的。因此，我们把这一年的经验和心得，写成了文章，合并成了这本年度合辑，希望能够和所有的技术人分享。本次技术合辑将围绕“大前端技术”，“技术架构”，“大数据技术”，“数据库技术“，”人工智能“，“高质量”等几个领域，专注在如何利用技术提升用户体验、提升效率，保障系统稳定性和降低成本的实践分享，同时也有我们在业务出海过程中的技术挑战和应对方面的经验分享。

“未来可期，星辰大海。”展望未来，我们充满信心。围绕集团的两大战略“国际化”和“高质量”，我们将继续坚持技术创新，我们将全面拥抱 AI 技术，我们坚信，AI 技术将成为我们新的引擎，推动我们的业务飞跃新的高度。我们将以更开放的心态，更积极的态度，迎接新挑战和机遇。

最后，我要向每一个团队成员表示最深的敬意和最热烈的感谢，是你们的辛勤付出，为公司的发展做出了巨大的贡献。同时，也感谢每一位读者的关注和支持，希望我们的经验和心得，能给你们带来启发和帮助。

携程集团副总裁/技术委员会主席 马超

大前端

携程跨端解决方案的新选择：Taro-CRN

【作者简介】李羽，携程高级前端研发工程师，专注前端跨平台框架领域的开发与研究。Hyme，携程前端研发经理，专注前端小程序/H5 领域的开发与研究。Chao，携程前端研发经理，关注前端跨平台领域与前端研发效率提升。

一、项目背景

随着小程序用户的增长，APP 和小程序在需求迭代上呈齐头并进的趋势。与此同时，前端研发人员面对多套平台代码的维护与开发，研发投入上耗时耗力。目前携程内部急需一种跨平台的开发框架，来节约不必要的多套开发量。

二、框架介绍

Taro-CRN 是帮助携程开发者基于 Taro 开发 CRN 项目的框架，实现一套代码在小程序和 APP 上的跨端开发，也为后续携程的跨端开发生态打下基础。Taro-CRN 由携程机票团队与火车票团队共建而成。Taro 本身是业内比较成熟的跨平台解决方案，目前已经支持转换到多平台小程序、H5、RN 页面，并且有很好的社区支持。

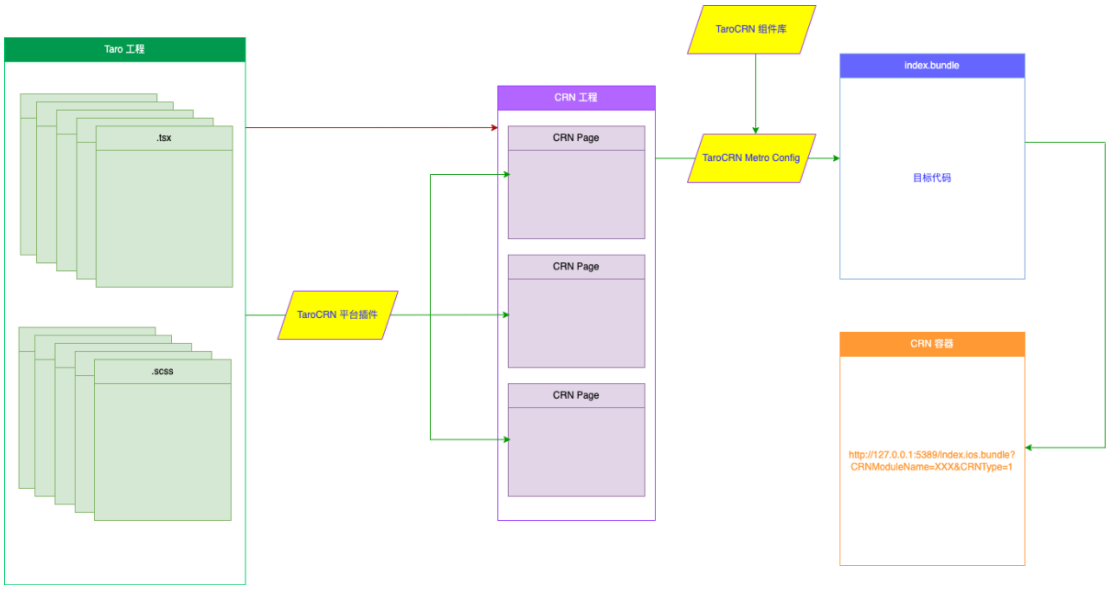
在携程内部，Taro 也拥有同样广泛的使用基础。多个业务线应用 Taro 来实现多平台小程序的开发，也有大量的 H5 业务页面是由 Taro-H5 转换而成。然而，Taro-RN 作为 Taro 跨端开发方案的最后一块拼图，在携程内部却很少有团队应用，其根本原因在于其难以与携程的 CRN 框架结合使用。CRN 是适用于携程 APP 业务开发的 React Native 框架，在携程系 APP 上有极为广泛的应用。CRN 会在构建过程中，进行一些针对携程业务的分包、混淆、引入公共包等优化策略。这一点与 Taro-RN 那种直接打出 bundle 包的方案难以融合。

由此我们确定了 Taro-CRN 框架的设计方向：

1）低成本接入，实现用 Taro 开发 CRN 页面。
2）贴近 Taro 官方方案，享受 Taro 强大社区支持。
3）接入 CRN 的构建与打包，适用携程业务开发。
4）低学习成本，提升开发者体验。

三、架构介绍

Taro-CRN 框架主要由 3 部分组成：平台插件、Metro Config 插件、基础组件及 API 库。



Taro-CRN 的平台插件提供了根据 Taro 来构建 CRN 工程的能力，这个构建出的 CRN 工程可以直接用来进行携程内部的 MCD 发布。同时平台插件也在 CRN 工程中引入了 metro config 的插件，通过对 metro 的配置做到引用的转向和 transformer 的支持，同时也在这里配置了 Taro-CRN 组件库的映射。在 CRN 的集中构建过程中，已经抹平差异的组件库与 API 库就被引入并做了替换。这样就做到了在 Taro 的业务工程上开发，低成本地打成 CRN 的 bundle 包并执行发布。

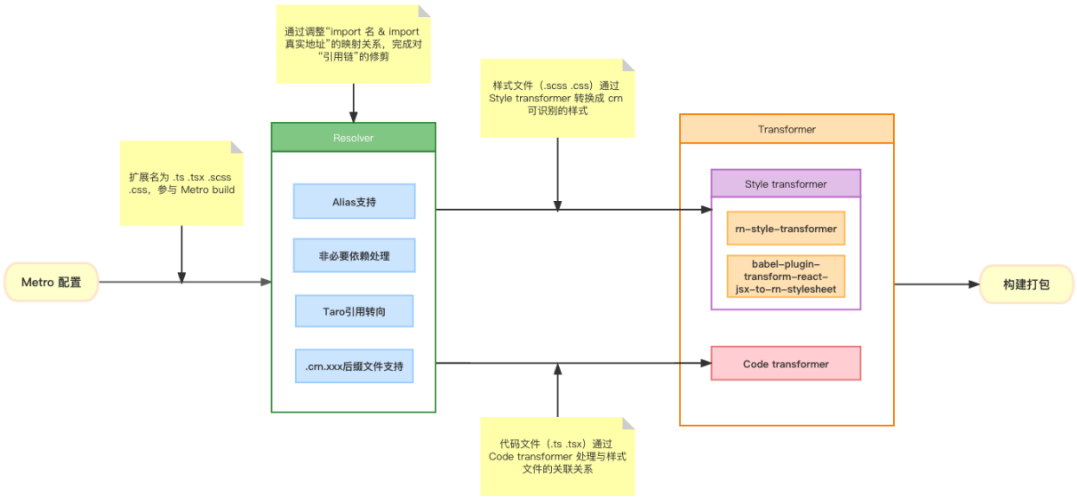
3.1 Taro-CRN 平台插件

Taro 为了让开发者拓展更多定制化功能，引入了插件化机制。Taro-CRN 的平台插件基于此，会按 Taro 工程的文件结构，基于内置的 CRN 模板文件，生成 CRN 工程所需的页面入口及目录结构，紧接着原本的 Taro 业务代码会被插件按目录塞到这个 CRN 的壳工程中。

同时平台插件也通过`ctx.registerPlatform`，将 CRN 与其他平台并作一起，可以像小程序或 H5 一样按 Taro 官方的命令进行开发与构建，提升了 Taro 开发者的开发体验。

3.2 Metro Config 插件

那么怎样将这样一个壳子是 CRN 结构、内嵌 Taro 业务代码的项目，打成 CRN 的最终产物呢？我们选择在 metro 构建过程中来处理。CRN 框架本身为业务方的 metro 配置提供了扩展的途径，我们由此通过 metro-config-plugin 插入对 Taro-CRN 项目的额外构建配置。



平台插件在生成壳工程的同时，就会向项目里植入 metro-config-plugin。在 metro 的解析阶段，插件会根据“引用链”分析 Taro 的组件及 API 的引用，并转向引用对应的 Taro-CRN 的组件及 API 库。得益于 Taro 支持完整的 React 开发体验，引用转向后就可以看做一个完整的 CRN 项目了。与此同时，Taro 的框架代码与依赖就这样被隔离到打包的 CRN 项目之外，规避了其他跨端方案普遍存在的包 size 增大的弊端，这保证了 Taro-CRN 框架生成项目的性能与直接用 RN 开发的项目无差异。

除此之外，metro-config-plugin 也做了很多方便业务开发的配置。比如 Taro 开发者熟悉的文件平台后缀，我们在这一层也实现了根据.crn.xxx 的后缀来支持不同端上的业务差异代码。还有一些携程内部平台的差异支持，Metro Config 插件中也保留了扩展能力。

对于 RN 与 Taro 在样式等写法风格的转换，我们选择在 transformer 的部分直接继承 Taro-RN 样式相关的 transformer，配合我们开发的'code-transformer'一起实现 babel 转换与样式转译。这种方案最大的好处就是在开发调试阶段，开发者可以直接调试原始 Taro 代码，而非转后的 CRN 代码，极大提升开发效率。

3.3 Taro-CRN 组件及 API 库

对于基础组件和 API，我们严格按照 Taro 官方文档一一对应提供，这样极大降低开发门槛，Taro 开发者甚至不需要学习 RN 即可使用。对于现有的 Taro 项目也可以不用做组件上的改动而直接转成 CRN 项目，拓展了框架的使用场景。

大多组件是直接沿用 Taro-RN 的实现，部分 CRN 有过优化的组件及 API，我们用 CRN 对应 Taro 的参数进行了实现。对于部分 RN 中存在，但在 Taro 中不存在的组件，我们也开发提供相应的组件为使用者提供便利。

携程体系下的多平台小程序，做了非常多基于业务上的 Taro 优化，并提供了相应的 API。为了方便携程主板的开发者，Taro-CRN 也对部分 API 做了抹平，进一步降低了使用门槛。

3.4 其他支持

`@ctrip/plugin-publish-crn`是 Taro-CRN 提供的发布插件。CRN 的发布是需要指定 CRN 的代码仓库，因此平台插件产出的 CRN 壳工程需要有独立的仓库或分支来存放。引入发布插件后，可以简单配置指定所需发布仓库或者是开发仓库下的发布分支，无需手动转移，提升发布效率。

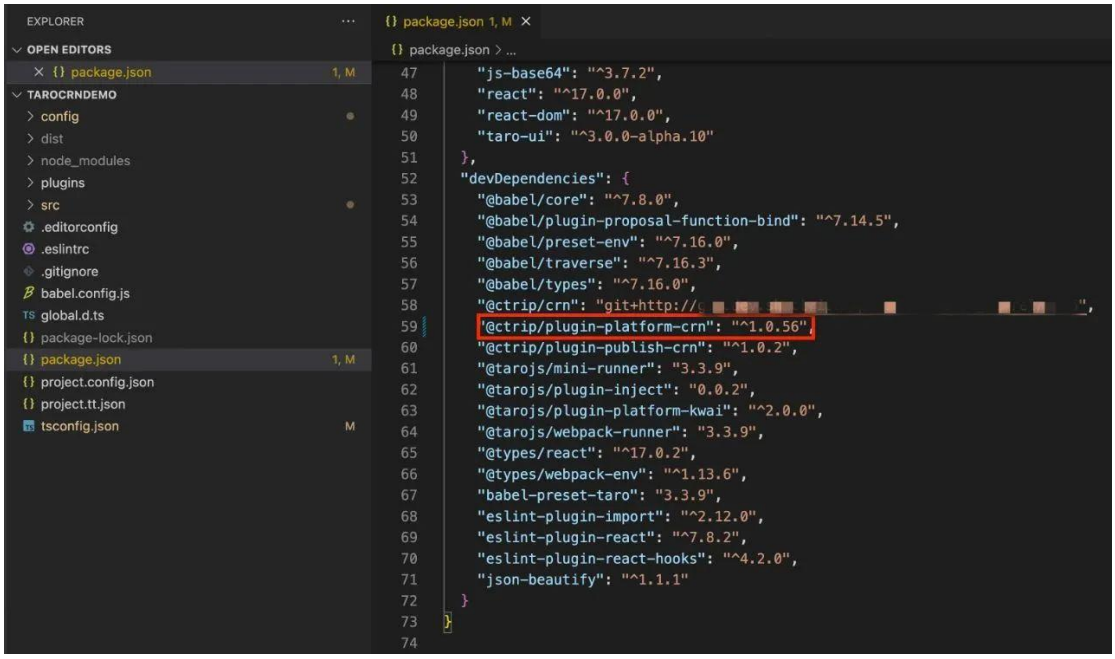
四、接入使用

4.1 Taro 项目中接入 Taro-CRN

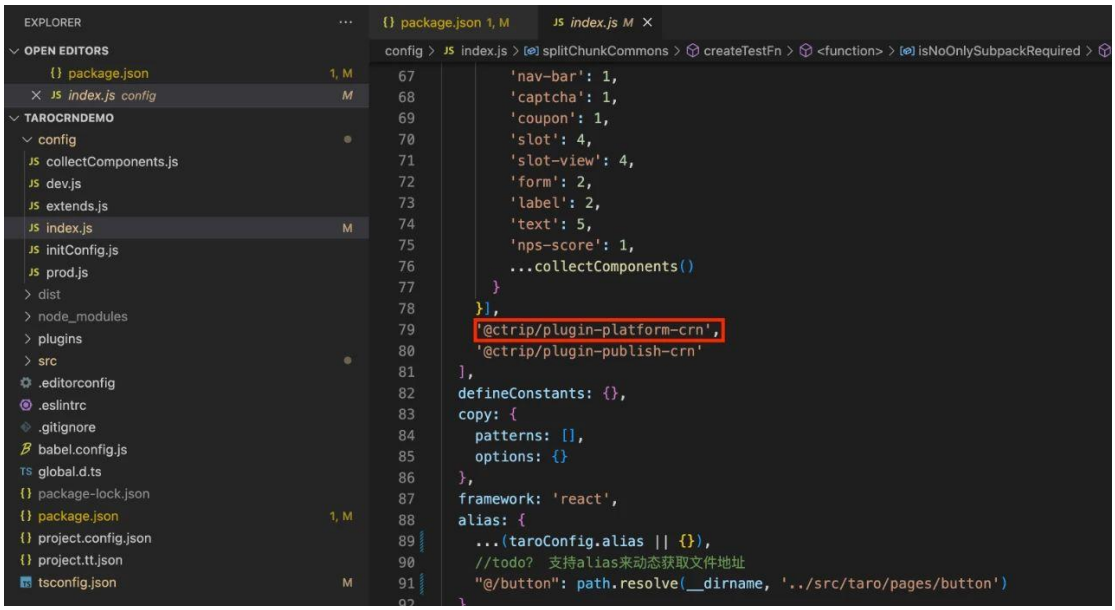
无论是接入新建 Taro 项目还是现有项目，都可以低成本转成 CRN 的项目来进行开发与调试。

a. 引入 Taro-CRN 平台插件

在 Taro 项目中引入前面提到的平台插件，`@ctrip/plugin-platform-crn`。另外如果也需要发布插件的话也可以接入`@ctrip/plugin-publish-crn`。



在 Taro 的目录结构中，config 是 Taro 官方提供扩展配置的目录，在 plugins 中配置相关的插件依赖。



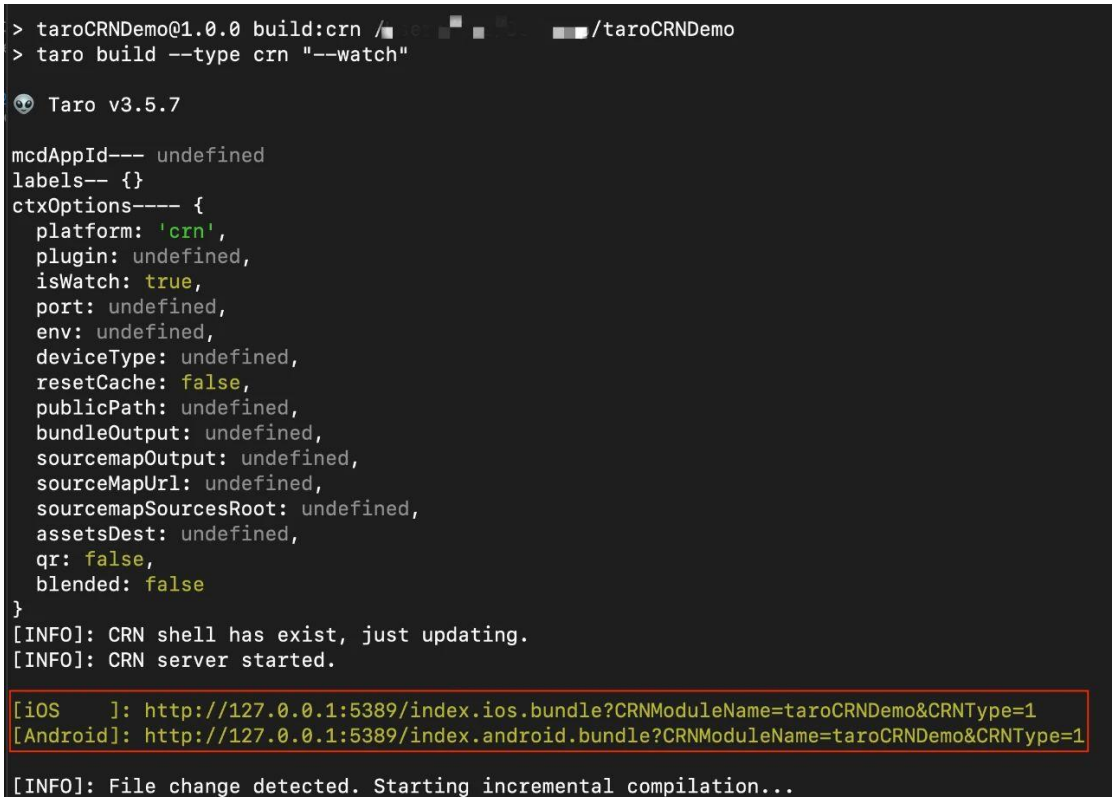
b. 配置 CRN 平台参数

在 config 目录下，除了插件依赖配置项之外，还有各平台的相关参数配置。CRN 作为待转的平台之一，可以像其他平台一样，支持在这里扩展配置。

```
crn: {
    projectName: 'YourProjectName', // 项目名
    partner: 'ct', // 内部多平台配置项
    publish: {
        target: 'git@Host:Group/Project.git',
        branch: 'specBranch',
        localDir: 'specDir'
    }, // 发布插件相关配置
    dependencies: {
      "react-native-svg": "~12.1.1",
      "babel-plugin-inline-import": "^3.0.0"
    }, // 项目所需额外依赖
    enableSvgTransform: true // 项目是否需要支持SVG
    // 更多其他扩展配置
  }
```

c. 调试与构建

接下来，开发者就可以按照 Taro 的开发习惯，直接用`taro build --type crn`来构建，或者通过 watch 来进行开发调试。



这样用户在模拟器中，打开红框对应的地址就可以开启调试了

d. 携程主板分包小程序的接入

携程当前的小程序生态为各业务线提供了针对 Taro 项目的扩展配置，这部分的接入也在 Taro-CRN 提供了额外的支持，只需要换成引入`@ctrip/plugin-platform-crn-tarox`版本的平台插件，并增加对应的分包配置，即可按前面的流程进入开发。

4.2 CRN 项目中接入 Taro-CRN 业务组件

在业务开发过程中，有很多场景并不需要完整页面全部支持跨端，组件上的跨端支持则更为灵活。比如对于一些业务模块卡片，Taro-CRN 支持 Taro 开发组件并输出，提供给 CRN 项目或者 Taro 项目直接引入，做到组件级的跨端开发。这样在被接入的 CRN 项目中只需要引入对应 Taro-CRN 的依赖，然后接入额外的 metro 配置即可。

```
const TaroCRNMetroConfig = require("@ctrip/metro-taro-crn");
const configBu = {
    resolver: TaroCRNMetroConfig.resolver,
    transformer: TaroCRNMetroConfig.transformer,
};

module.exports = configBu;

```

五、框架总结

对于跨端开发，业内其实有提供很多方案，开发者只有了解各自方案的使用场景，才能做出适合项目的选型。Taro-CRN 的适合的场景：

1）适用于已有 CRN 支持的 APP 接入 Taro 方案；
2）适用于需要提供业务模块同时支持 APP 与小程序的场景；
3）适用于缺少 RN 开发经验的团队实践跨端开发。

当然，Taro-CRN 框架仍是需要持续地打磨与优化。尤其为应对更加复杂的业务场景，在基础组件之上可以开发更多适配多平台的业务组件，来进一步的提升开发效率。这也是后续我们与携程内部多方合作优化的方向之一，以便于更好的促进 Taro 技术生态在携程的落地，给与开发者更多选择的空间。欢迎感兴趣的 Taro 或 RN 的开发者交流意见。

新时代的 SSR 框架破局者：qwik

【作者简介】19 组清风，携程资深前端开发工程师，负责商旅前端公共基础平台建设，关注 NodeJs、研发效能领域。

引言

今天这篇文章中和大家聊一聊号称世界上第一个 O(1) 的 JavaScript SSR 框架：qwik。

别担心，如果你不是特别了解 SSR 也没关系，文章大概会从以下几个方面作为切入点：

- 首先会围绕对比 SSR 与 SPA 各自的优劣势，从而展开 SSR 的运行机制以及 SSR 相较于 SPA 究竟为了解决什么问题。
- 之后，会根据 NextJs 的运行机制思考针对目前主流 SSR 框架设计思路上存在的不足从而引出 qwik 为何会在众多成熟框架中脱颖而出。
- 最后，会针对于 qwik 提出自己的看法以及聊聊目前 qwik 存在的“问题”。

诸如社区内部 SSR 框架其实已经产生了非常优秀的作品，比如大名鼎鼎的 NextJS 以及新兴势力代表的 Remix 和 isLands 架构的 Astro、Fresh 等等优秀框架。

为何 qwik 可以在众多老牌优秀框架中脱颖而出。接下来，让我们一起来一探究竟吧。

一、SSR & CSR

目前业内存在非常多基于 SSR 的优秀框架，比如 Next、Remix、Nuxt 等等。

针对于 Qwik 我们先来聊聊基于 Next 体系的传统 SSR 方案。

1.1 Client Side Rendering

在开始 SSR 之前我们先来聊聊它的对立面，所谓的 CSR(Client Side Rendering)。

服务器端渲染 (SSR) 是一种在服务器中进行渲染 HTML 而不是由浏览器中执行 JS 获得网页(SPA)的技术。

目前国内社区中主流框架比如 VueJs、React 等严格意义上来说都是基于 CSR(Client Side Rendering) 的产物。

所谓 CSR 的意味着当发出一个请求时，服务器会返回一个空的 HTML 页面以及对应的 JavaScript 脚本。

比如：

```
<html>
<head>
   <title>携程商旅</title>
</head>

<body>
     <div id="root"> </div>
     <script src="./index.js"> </script>
 </body>
 </html>
```

当浏览器下载完成对应的 JS 脚本后才会动态执行对应的 JS 脚本然后在返回的 HTML 页面上进行渲染页面内容。

你可以简单的理解为上述的 ./index.js 会在客户端下载完成后执行该脚本，从而执行 document.getElementByld('root').innerHTML = '...' 来进行页面渲染。

这种方式并不是从服务端下发的 HTML 文件来进行渲染页面，相反而是通过浏览器获取到服务端下发 HTML 中的所有的 JS 文件后执行 JS 代码从而在客户端通过脚本进行页面渲染。

以及通常在 CSR 中当我们点击任何页面中的导航链接并不会向服务端发起请求，而是通过下载的 JS 脚本中的路由模块(比如 ReactRouter、VueRouter 这样的模块)重新执行 JS 来处理页面跳转从而进行页面重新渲染。

上面的概念是非常典型的 CSR ， 浏览器仅仅接受一个用作网页容器的 HTML 页面，这样的方式通常也被称为单页面应用 (SPA)。

1）优势

那么上述我们提到的 CSR 广泛存在于目前大量页面中，必然存在它自己的优势。

在页面初始化访问后加载速度极快且响应非常迅速。在页面初始化后，网站所有的 HTML 内容都是在客户端通过执行 JS 生成，并不需要再次请求服务器即可重新渲染 HTML 。

此外，有关任何实时的数据获取都可以通过 AJAX 请求对于页面进行局部更新从而刷新页面。

2）劣势

可是，CSR 真的有那么完美吗。任何一件技术方案一定存在它的两面性，我们来看看 CSR

方式究竟存在哪些问题：

- 初始加载时间长。

首次请求完服务器获取到 HTML 页面后，初始化的页面仍然需要在一段时间内处于白屏状态。

在初始渲染之前，浏览器必须等待 HTML 页面中的所有 Javascript 脚本加载完成并且执行完毕，此时页面才会进行真正的渲染。

当然，使用代码拆分或延迟加载等多种方案可以有效的减少上述的问题。但是这些方式始终是治标不治本，因为它并没有从本质上解决 CSR 存在的问题。

- SEO（搜索引擎优化） 的负面影响。

上边我们提到过，所谓 CSR 本质上首先会返回一个空的 HTML 页面，所以这也就造成了在搜索引擎对于该页面的数据爬取中会认为它是一个空页面。从而影响对应的搜索结果排名。

虽然说在最新的 Google 中已经可以触发执行 JS 对于网站进行关键字排名，但是在 JS 体积足够大的时候针对于 SEO 仍然是存在一部分问题导致无法解析出正确的关键字匹配。

当然 CSR 还存在一些其他方面的缺点，比如网站强依赖于 JS 当用户禁用 JS 时网站只能是白屏展现给用户等等之类。

1.2 Server Side Render

简单聊完客户端渲染后，我们稍微来看看所谓的服务端渲染是什么含义。

基于旧时代的类似 Java 的 JSP 页面我在这里就不赘述了，显然 JSP 的方式每个 HTML 都需要单独请求服务器返回对应的 HTML 内容严格意义上来说这也是 SSR 的方式但是很明显这已经被时代淘汰了。

目前国内各家公司广泛应用的服务端渲染技术大概的思路是这样的（Next 的 SSR 模式也是同样的思路）：

当用户首次访问你的应用站点时：

- 首先服务器会根据对应的 URL 在服务端根据对应路径渲染对应的 HTML 模版。

  注意这里渲染的 HTML 模版是具有该页面真正的内容。同时它并不具备任何交互逻辑（比如 DOM 元素的点击事件），这是一份完全的静态站点。

- 服务器会下发这份仅具有静态内容的 HTML 模版，同时这份模版中也会包含对应的

JavaScript 执行脚本。

第一时间会展示给用户对应的 HTML 页面，此时对于访问站点的用户来说首屏渲染相较于 SPA 应用来说会非常快。因为它并不需要在客户端浏览器上再次下载和执行 JavaScript 脚本来进行页面渲染。

其次，针对于 SEO 的优化也会非常良好，因为服务器上下发的 HTML 页面是包含当前站点的真实 HTML 结构，对于搜索引擎的爬虫来说会非常容易的匹配到当前关键字。

- 之后，浏览器会下载当前这份 HTML 的 JS 脚本。

  因为首先呈现给用户的一份静态的 HTML 页面，并不具备任何交互效果。我们需要为页面上的元素增加对应交互，HTML 页面中的 JS 脚本中会包含网站的交互逻辑。

- 最后，当下载完 HTML 脚本中的 JS 脚本后，自然会执行这些 script 脚本。从而发生一种被称为 # hydrate(水合) 的方式，从而为页面上静态 HTML 元素再次添加对应的事件处理从而保证页面具有交互性。

当 hydration 过程完成后，会由我们的客户端框架接管网站的后续渲染。在后续的导航链接跳转和页面渲染中和服务器已经没有任何关系了，我们完全可以利用客户端的路由切换（History Api/Hash Api）利用 JS 进行页面渲染从而保证切换页面不用再次请求浏览器保证非常及时的页面交互。

1）hydration

上述过程中有一个非常重要的关键字 hydration（水合）。

首次访问页面时，页面的静态 HTML 是在服务端生成的。在服务端我们将生成的静态 HTML 以及 HTML 中携带的 JS 脚本发送到客户端。

此时静态 HTML 会立即显示在用户视野中，然后浏览器会利用网络进程下载当前 HTML 脚本中的 JS 脚本。

当 JS 脚本下载完成后，会立即执行同时发生一种被称为 hydration 的过程。

所谓的 hydration 简单来说，也就是客户端下载完成 JS 脚本后，浏览器会执行下载的 JS 脚本这些脚本中有部分内容会将已经存在的 HTML 内容通过执行下载的 JS 脚本添加上对应的事件监听器从而保证页面的交互。

注意，在 React、Vue 中 hydration 并不意味着重新渲染。因为在 Server 端已经渲染了和 Client 完全相同的 DOM 结构所以完全没有必要在此重新渲染。

所以 hydration 的过程是给当前页面中已经生成的 HTML 页面添加上对应的事件监听器。

这也是为什么在 Next 等框架中为什么必须要保证 Server 端和 Client 的渲染 HTML 结构必须一致的原因。

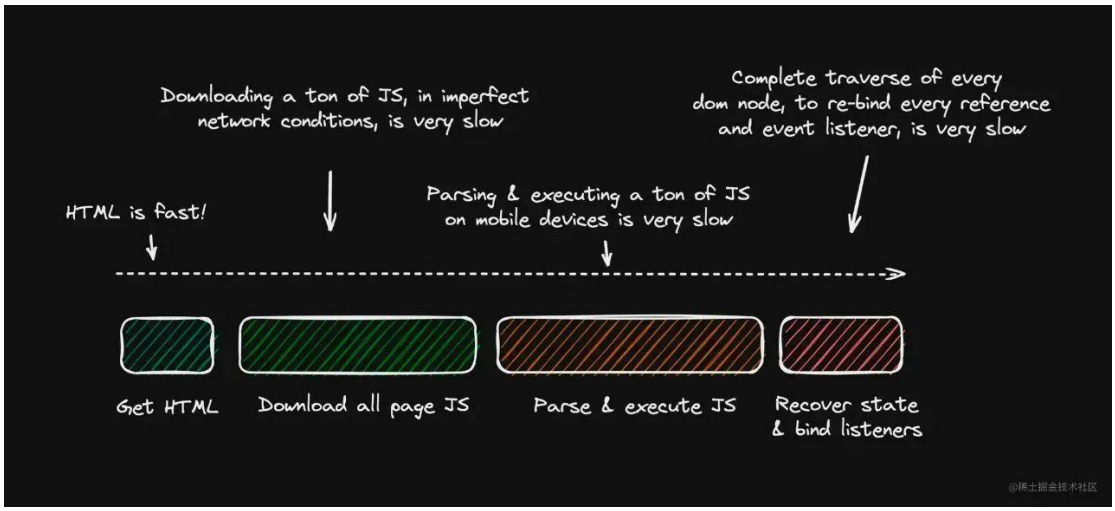
比如我们以 Next 举例来说（Vue 也是同样的道理）：

- 当用户访问 www.trip.biz 时，服务端接收到请求调用 ReactDOMServer.renderToString() 生成当前页面的 HTML 静态结构。
- 服务器会下发这个 HTML 页面给客户端，同时这个 HTML 页面上也会携带一部分 JS 脚本 script 标签。
- 用户的浏览器中会立即展现到该 HTML 页面，同时也会下载对应 JS 脚本并执行。
- 当 JS 脚本执行完毕后，客户端会调用 ReactDOM.hydrate() 发生水合为当前页面的 HTML 页面添加事件交互处理，同时后续由 JS 接管页面的跳转渲染。

针对于第一步 Next 中存在 Automatic Static Optimization 的优化，并不一定会在每次访问时调用 renderToString 方法，有可能在构建时也会直接生成对应的 HTML 模版。

当然，在最新的 Next 版本中已经支持了 Stream 以及 Server Components。

整个过程就像是这张图中的样子：



2）优势

简单聊过了所谓 SSR 的原理后，如果你有认真看上述的内容。其实我相信相较于 CSR，SSR 这种方式的好处不言而喻：

- 更好的搜索引擎优化 SEO 方式，HTML 模板是从服务端直接下发这也就导致搜索引擎爬虫中更多的关键字匹配。
- 更快的首屏渲染，因为相较于 SPA 它少了在 Client 中下载和执行 JS 脚本后渲染的过程。
- 页面不需要 JS 也可以正常渲染，虽然没有 JS 意味着页面失去了可交互性。但对于禁

用 JS 的用户来说，展示一些静态内容总比 SPA 应用的白屏来的更加友好一些对吧。

3）劣势

当然，任何技术方案在不同场景下也存在它自己的不足。

- 强依赖于服务。

针对于 CSR 的方式它是一种纯静态资源。我们可以直接将它放在 CDN 上就可以良好的用户访问到，而 SSR 的方式必须依赖于一个服务器进行服务端预渲染。（当然纯 SSG 应用我们不在这个讨论范围之内）

同时，有服务的地方就存在并发压力。当你需要为你的应用考虑服务端渲染的方式时，一定不要忘记为你的服务器进行压测。

- Time to Interactive 可交互时间（TTI）的增长，虽然说 SSR 的方式有效的缩短了首屏加载的方式，但是会增加所谓的 TTI（可交互时间）。

所谓的 TTI 指标测量页面从开始加载到主要子资源完成渲染，并能够快速、可靠地响应用户输入所需的时间。

因为 SSR 的方式在用户访问时会下发当前页面中静态的 HTML 内容，也就是所谓的 First Contentful Paint 首次内容绘制（FCP）会非常快速，但是页面需要用户交互效果又需要下载和执行完成 JS 脚本发生 hydatrion 后才具有交互性。

这也就造成页面的 TTI 相较于 CSR 方式会有所差劲，因为 CSR 在渲染完成后就会立即具有交互性（不需要其他任何多余步骤）。

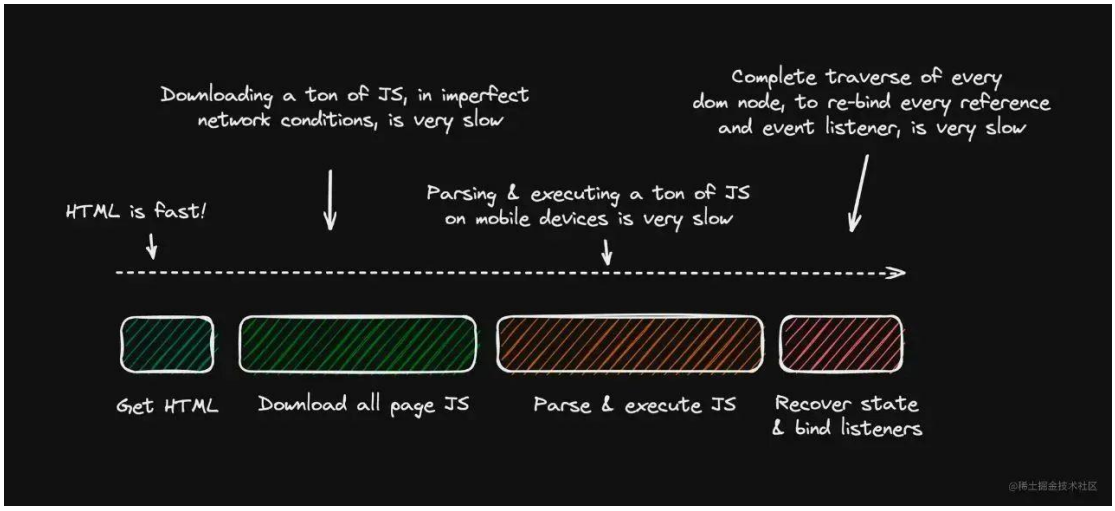
1.3 qwik

上述聊了那么多前置内置，终于要和大家切入正题了。

所谓磨刀不费砍柴功，上边和大家强调现阶段 SSR 的方案以及对应的优劣势就是为了引入下面的内容。

首先，这篇文章的目的是为了让大家在当前众多 SSR 框架中思考性能方面是否可以有所提升的，在服务器方面不会过多的深入。

我们可以稍微思考下上述服务器端渲染的过程：



第一步我们需要在服务端获取对应页面的 HTML 页面，大多数情况（非纯静态页面）就需要在服务端掉用对应渲染方法渲染出 HTML 页面。

那么，如果我们能在第一步渲染 HTML 页面时，就添加对应的事件处理。后续的 3 步是不是完全可以省略下来了对吧。

其实社区内部之前已经有非常多的方案来提升所谓 SSR 框架的性能方案。

比如 Remix 的 HTTP stale-while-revalidate 缓存指令

比如 astro 等新兴框架的 Islands 架构方案，关于 Islands 有兴趣的朋友可以参考神三元的这篇 Islands 架构原理和实践。

针对于上面的概念，我们直接来看看 qwik 中提到的 Hydration is Pure Overhead （完全多余的 Hydration）。

1）Hydration 造成的开销

首先针对于 Hydration 的过程，我们提过到首先会在服务器上进行一次静态 HTML 渲染，之后当 HTML 下发到客户端后又会再次进行 hydrate 的过程，在客户端进行重新执行脚本添加事件。

Hydration 过程的难点就在于我们需要知道需要什么事件处理程序，以及将该事件处理程序附加在哪个对应的 DOM 节点上。

这个过程中，我们需要处理：

- 每一个事件处理程序中的内容，绝大多数框架中的状态都作为闭包函数保存在内容中。所以需要 hydration 的过程来重新获取状态。
- 其次，在搞清楚了每个事件处理函数的内容后。我们也需要将对应的事件处理函数附加

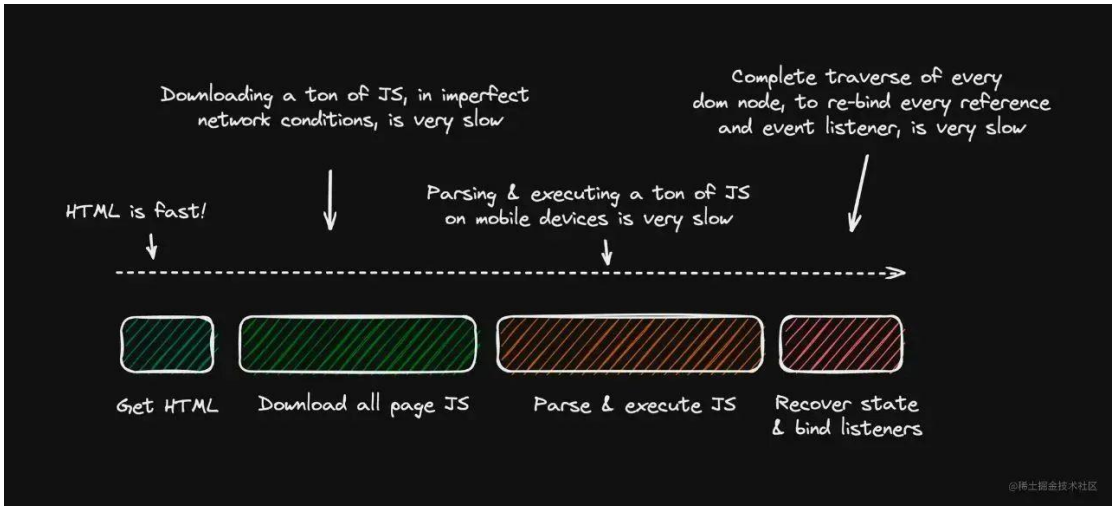
到对应的 DOM 节点上，同时还要确保该监听器的正确事件类型。

更加复杂每个事件处理函数中的内容是一个闭包函数，这个函数内部需要处理两种状态，APP_STATE 以及 FRAMEWORK_STATE。

- APP_STATE：应用程序的状态。简单来说应用程序的状态就是 HTML 事件中的各个状态事件，如果不存在这些事件状态那么所有的内容都是没有任何交互效果的。
- FRAMEWORK_STATE：框架内部状态。通常我们会利用诸如 React 或者 Vue 等框架进行接替渲染。如果没有 FRAMETER_STATE，框架内部就不知道应该更新哪些 DOM 节点，也不知道应该在什么时候更新它们。

通俗来说 Hydration 就是在客户端重新执行 JS 去修复应用程序内部的 APP_STATE 以及 FRAMEWORK_STATE。

同样还是这这张图：



在图中的前三个阶段可以被称为 RECOVERY 阶段，这三个阶段主要是在重建你的应用程序。

当从 Server 端下发的 HTML 静态页面后，我们希望它是具有交互效果的 HTML 正常应用程序。

那么此时 hydartion 的过程必须经历下载 HTML 、下载所有相关 JS 脚本、解析并且执行下载的 JS 脚本。

RECOVERY 阶段是和 hydartion 的页面的复杂性成正比，在移动设备上很容易花费 10 秒。

由于 RECOVERY 是昂贵的部分，大多数应用程序的启动性能都不是最佳的，尤其是在移动设备上。

前三个阶段被称为 RECOVERY 的阶段其实是完全没有必要的，因为在服务端我们已然渲染

过对应的 HTML ，但是为了应用程序的可交互性以及服务端仅保留了静态的 HTML 模版导致不得不在 Client 上继续执行一次 Server 端的逻辑。

总而言之，hydration 其实是通过下载并重新执行 SSR/SSG 呈现的 HTML 中的所有 JS 脚本并执行来恢复组建中的事件处理程序。

同一个应用程序，会被发送到客户端两次，一次作为 HTML，另一次作为 JavaScript。

此外，框架必须立即执行 JavaScript 以恢复在服务器上被丢掉的 APP_STATE 和 FRAMEWORK_STATE。所有这些工作只是为了检索服务器已经拥有但丢弃的东西!!
比如这样一个例子:

```
export const Main = () => <>
   <Greeter />
   <Counter value={10}/>
</>

export const Greeter = () => {
  return (
    <button onClick={() => alert('Hello World!'))}>
      Trip Biz
    </button>
  )
}

export const Counter = (props: { value: number }) => {
  const store = useStore({ count: props.number || 0 });
  return (
    <button onClick={() => store.count++)}>
      {store.count}
    </button>
  )
}
```

上边的例子中我们编写了一个 Counter 的计数器组件，在传统 SSR 过程中该组件会被渲染成为:

```
<button>Greet</button>
<button>10</button>
```

可以看到上边的两个按钮不拥有任何处理状态的能力。

要使网页具有交互性，必须要做的就是通过下载对应 HTML 页面中的 script 脚本并执行代码从而恢复按钮上的交互逻辑和状态。

为了具有交互性，客户端不得不执行代码实例化组件后重新创建状态。

当上述过程完成后，你的应用程序才会真正具有可交互性。无疑，同一个组件的渲染逻辑被执行了两遍，这是一个非常冗余且耗费性能的过程。

2）Resumability: 更加优雅的 hydartion 替代方案

所以为了消除额外的开销，我们需要思考如何避免重复的 RECOVERY 阶段。同时还要避免上面的第四步，第四步是执行脚本后给现有的 HTML 附加正确的事件处理程序。

qwik 中提出了一个全新的思路来规避 RECOVERY 带来的外开销:

- 将所有必需的信息序列化为 HTML 的一部分。

qwik 将需要的状态以及事件序列化保存在 Server 端下发的 HTML 模版中，需要序列化信息需要包括 WHAT (事件处理函数内容) , WHERE (哪些节点需要哪些类型的事件处理函数) , APP_STATE（应用状态），和 FRAMEWORK_STATE（框架状态)。

- 依赖于事件冒泡来拦截所有事件的全局事件处理程序。

qwik 中事件处理程序是在全局处理的，这样我们就不必在特定的 DOM 元素上单独注册所有事件。

- qwki 内部存在一个可以延迟恢复事件处理程序的工厂函数。

该工厂函数主要用于处理 WHAT 阶段，也就是用来识别某个事件处理函数中应该存在什么脚本逻辑。

我们可以看到所谓的 Resumable 对比 Hydration 明显可以省略不需要后三个阶段，直接获取 HTML 后页面其实就已经准备完毕，这无疑对于性能的提升是巨大的。

对比传统的 hydration 方案，在客户端获得服务端下发的 HTML 后会立即请求需要的 JS 脚本并执行从而为页面附加对应的交互效果。

而 qwik 提出的概念恰恰相反，获取完服务端下发的 HTML 页面后所有的交互效果实际上都是一种惰性创建的效果。

因为我们在 HTML 中的每个元素中都已经通过序列化从而在它的标签属性上记录了对应事件处理函数的位置以及脚本内容（自然内容中也包含对应的状态），所以当获得 HTML 页面后其实就可以说此时页面已经加载完毕了而不需要任何实时的 JS 执行。

这样做的好处是在 qwki 中完全可以省略 hydration 的多余步骤，甚至可以说完全抛弃了 hydration 的概念。

客户端完全不必和服务端的 HTML 进行水合，相同的渲染内容仅仅是在 Server 端进行一次渲染客户端即可拥有对应的事件处理内容。

简单来讲 Qwik 的工作原理就是在服务端序列化 HTML 模版，从而在客户端延迟创建事件处理程序，这也是它为什么非常快速的原因。

3）qwik 工作机制

上边我们讲到了 qwik 的原理部分，同样拿上边的计数器的例子我们来对比下：

```tsx
export const Main = () => <>
   <Greeter />
   <Counter value={10}/>
</>

export const Greeter = () => {
  return (
    <button onClick={() => alert('Hello World!'))}>
      Trip Biz
    </button>
  )
}

export const Counter = (props: { value: number }) => {
  const store = useStore({ count: props.number || 0 });
  return (
    <button onClick={() => store.count++)}>
      {store.count}
    </button>
  )
}
```

在 qwik 编译后，服务端会序列化对应组件的 HTML 结构从而下发如下的模板：

```html
<div q:host>
  <div q:host>
    <button on:click="./chunk-a.js#button">Trip Biz</button>
  </div>
  <div q:host>
    <button q:obj="1" on:click="./chunk-b.js#count[0]">10</button>
  </div>
</div>
<script id="qwikloader">/* qwik 中设置全局事件监听器的代码 */</script>
<script id="qwik/json">/* 用于反序列化的 JSON 相关信息 */</script>
```

我们可以看到经过 qwik 编译后的 html 结构并不单单只有 DOM 元素，同时会在对应需要状态 & 事件的 DOM 元素上通过 HTML 元素属性来记录当前元素的事件和状态信息，这既是 qwik 中的序列化。

比如上边 button 的 on:click 属性记录了该元素后续需要恢复的所有信息。

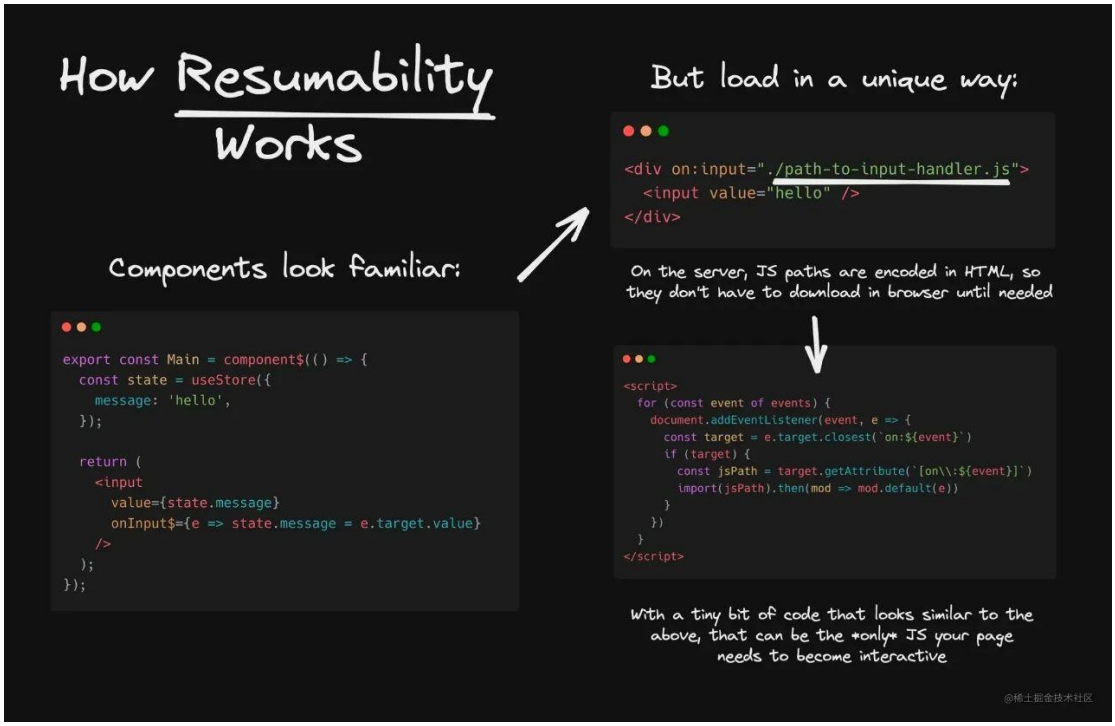
需要注意的是序列化这一步是在服务端渲染时完成的，这也就意味着后续客户端可以通过服务端序列化的属性信息进行反序列化从而达到所谓的可恢复性而不需要重复执行组件。

当然你可能会好奇 qwik 是如何进行这些事件 & 状态的恢复，qwik 正是通过在返回的 HTML 页面中内嵌的所谓 qwikloader 的 script 脚本（这段脚本的大小不超过 1kb）配合 qwikjson 映射表，从而在全局进行恢复事件和状态的逻辑。

正因为这个原因，使得 qwik 相较于传统 SSR 的 hydration 在 Client 中再次执行渲染从而水合页面状态和事件处理程序，这简直可以说是接近零 JS 的执行过程。

最终在用户触发事件时候达到惰性的创建事件并执行，这个过程中完全没有重复任何服务器已经完成的任何工作。

整个工作过程就像下面这张图描述的那样：



上边的这张图完美的描述了 qwik 的工作原理，相信经过上述的描述大家对于这张图中想表达的思想已经可以完美的理解了。

利用 qwik 的这个优势，在绝大多数应用中我们可以利用 qwik 保证你的 SSR 应用在保证快速的 FCP 的前提也同样拥有与之不相上下的 TTI 体验效果。

4）惰性加载脚本会影响用户交互体验吗

当然上文说过任何框架的优势和劣势都不是绝对的，在我们看来 qwik 的确会存在以下一些问题。

大多数同学看完上边的内容我相信也会存在“惰性加载脚本会影响用户交互体验吗”这样的疑问。

首先，qwik 中既然选择在触发用户行为时，再惰性加载并执行响应的 JS 脚本。那么难免需要在用户触发交互时动态生成对应的事件处理函数进行执行。

这样的方式相较于传统 hydration 的确会存在一些不足，需要额外生成事件会额外造成交互响应时间的损耗而传统 SSR 方式在页面首次加载时就已经绑定好（相当于生成了）相应的事件处理函数。

就惰性加载生成事件这点：

针对于动态加载 JS 脚本，其实已经存在诸如非常多的 prefetch 等等预加载技术。

无论是基于传统 Next 方案还是基于 qwik 这种惰性可恢复的方案，利用 prefetch 等预加载技术优先在网络空闲时加载响应重要的 JS 脚本都是非常有必要的，所以这点在我看来并不是特别重要的问题。

5）延迟加载会带来 bundle 数量的上升吗

qwik 推崇的延迟加载其实已经是一项非常成熟的构建技术了。无论是使用 webpack、rollup 又或是其他任何构建工具都存在延迟加载 & 代码分割的技术。

传统构建工具中关于代码分割会带来以下两点的困难：

- 需要开发人员自行去处理更加细粒度的代码分割，当然这并不是最主要的。因为目前我们可以利用 Magic Comments 配合 Dynaic Imports 来解决需要手动切入多个入口点的问题。

当然，这一系列事情比起魔法注释。因为 qwik 本身提倡的就是所谓的延迟加载，所以在框架内部已经帮我们足够智能的去处理这个过程。

但是需要注意的是这并不意味着开发者无法自主去控制这个过程。只是在使用框架的过程中，qwik 希望开发者更加专注于他们自身的业务逻辑。

- 当存在非常多的延迟加载时，传统构建工具会从一个大 bundle 分割成为无数个小的 bundle 。

延迟加载模块的确会存在多个 small bundle 的问题，可是当我们拥有的 bundle 越多，其实我们就拥有更多的自由度去以各种各样的方式去拼装成为单个大的 bundle。

qwik 中存在足够的方式提供给我们将多个小的 chunk 自由组合成为一个从而有效的减少细碎 chunk 的数量，当然这个点在传统构建工具中也是这样。

6）动态创建事件函数会造成内存泄漏吗

qwik 的设计思想在与每次事件触发时通过 qwikloader 来动态创建事件处理函数，相信有的同学存在疑问“那么多次触发事件会造成额外的开销吗”。

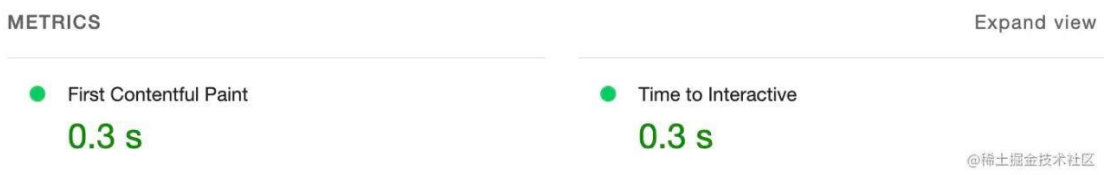
qwik 的作者 miško hevery 在 Hydration is Pure Overhead 中明确的表示过 qwik 会在每次事件执行完毕后释放函数，相当于每次事件执行完毕都会进行一次“去水合”的过程。

所以，当你触发一次事件和无数次事件函数在执行过程中对于内存占用来说是相差无几的。

当然相较于传统 hydration 的方式（在页面首次渲染时在内存中记录所有状态），无疑 qwik 这种并不在内存中记录任何状态的方式恰恰对于内存的占用比 dyration 更加轻量化。

7）qwik 真的有那么快吗

说了那么多，那么 qwik 真的有那么快吗。



上图是利用 qwik 搭建的 builder.io 官方网站，我相信 builder.io 的数据已经告诉我们答案了。

固然上述的数据并不仅仅只是单纯一个 qwik 框架带给网站的优化，一定会有代码层面或者构建层面等等方面的优化配合而来的数据。但是针对于 FCP 和 TTI 时间上的一致性这在一个中型 SSR 应用程序其实可以称得上是非常优秀了，我相信这足以说明了 qwik 的确名副其实。

二、结语

我们可以看出来，qwik 的核心思路还是通过更加细粒的代码控制配合惰性加载事件处理程序以及事件委托来缩短首屏 TTI。

文章中我们也讲到了 qwik 其实并不是因为使用了多么牛逼的算法导致它有多么快，而它的速度正是得益于它的设计思路，省略了传统 SSR 下首屏需要加载庞大的 JS 进行 hydration 的过程。

当然，我们对于 qwik 也仍是在学习阶段。后续会在公司里的更多项目尝试 qwik 之后也会

和大家分享关于它的更多心得。

总而言之，qwik 的“无水合”设计思路目前看来的确会在框架层面带来巨大的性能提升。大家如果有机会的话也可以在项目中尝试一下 qwik ， 相信会给你带来意想不到的收益效果。

提升前端工程化，携程 Design2Code 从零到一的实践

【作者简介】by，携程高级研发经理，专注低代码平台搭建和前端智能化技术。Jialu，携程研发总监，专注大前端技术和工程化的研究与发展。

一、 背景

在软件开发过程中，团队协作效率的提高是我们共同关注的问题。为了解决这一问题，许多团队都开始使用智能化工具。Design2Code（简称 D2C）工具是其中一种广受欢迎的选择。

在本文中，我们将分享 D2C 工具的核心算法方案设计和实现过程，以及一系列的解决方案。无论你是开发人员，还是设计师，本文都将为你提供有价值的信息。我们希望，通过阅读本文，能帮助你更好地了解 D2C 工具，并在实际工作中发挥出最大的价值。

二、 前置知识

2.1 基本概念

D2C 是一种通过使用智能化技术将设计稿转化为代码的工具，旨在提升开发效率，减少人力成本，并缩短设计到开发的流程。通过将设计稿源文件（如：PSD、Sketch、Figma）转化为 React、Vue、小程序等平台的前端代码，再使用领域特定语言（DSL）将其转化为各端代码，D2C 通常通过人工智能训练的模型或算法实现。

2.2 哪些特性

- 高效协作：使用智能化技术，提升前端工程师和设计师之间的协作效率。
- 高还原度：降低设计师检查页面还原度的人力成本。
- 高兼容性：减少多端适配的开发成本。
- 快速上线：自动生成视图层代码，减少手工工作量。

三、 业界现状

我们对业内比较出色的 D2C 产品进行调研，包括阿里 imgcook、京东 Deco、CodeFun、微软 AI Lab 和 Locofy。我们在调研过程中，对各家公司的产品进行了对比，但这里不再赘述。通过调研，我们得出初步结论：

- 预装 Sketch 插件或上传文件解析，导出 DSL 数据。
- 通过 AI 自动处理并生成前端代码，支持使用 Web 编辑器对代码进行人工干预和编辑。
- 需要将设计稿数据存储在对方服务器。
- 代码未开源，解决方案也只是简略描述。
- 部分公司进阶功能需付费使用。

四、 效果展示

D2C 系统转换流程介绍：

- 1. 上传 Sketch 源文件入口 → 2. 对数据进行处理并返回结果 → 3.展示转换结果。
- 通过“查看代码”功能获取多端代码。
- 我们对 APP、H5 、Online 和小程序设计稿进行了还原比对，还原精度达到 80%，且在不断提升中。

下图展示了系统转换效果：

五、 解决方案

5.1 方案分析

根据上述调研结果以及业务场景，我们的目标是将设计稿转换为代码。主要解决三个问题：

（1）提取图层信息：需要提取设计图中的图层信息，以获取所有图层元素的位置、大小、形状和颜色等信息。这些信息将为后续的页面布局提供基础。

（2）信息预处理：在将图层信息转换为代码之前，需要对信息进行一些预处理。例如，筛选和过滤无用信息和图层，处理图层 style 字段与 css 的映射关系，解决设计稿中文件夹 order 如何转换为布局 order 的问题，以及处理图片资源的导出等。预处理有助于提高后续代码生成过程中的代码质量和布局准确性。

（3）构建布局关系：利用所提取的图层信息，可以进行精确的页面布局。使页面尽可能地还原设计稿。

5.2 方案思考

为了更好的理解问题，将从简单到复杂依次讲解上述三个问题。

（1）问题 1："提取图层信息"，使用 sketch2json 开源工具可以从 Sketch 和 Figma 设计稿中提取图层信息，将矢量数据转换为 JSON 格式。数据结构中的 'layers' 对应着 HTML 的结构层，而 'style' 对应着 CSS 的表现层。下图展示了从 Sketch 中获取的数据结构。


```
▼layers: [{_class: "group", do_objectID: "C748DCF9-C925-446B-A1D5-AD9CB9824F81", booleanOperation: -1,…},…]
 ▼0: {_class: "group", do_objectID: "C748DCF9-C925-446B-A1D5-AD9CB9824F81", booleanOperation: -1,…}
    booleanOperation: -1
    clippingMaskMode: 0
    do_objectID: "C748DCF9-C925-446B-A1D5-AD9CB9824F81"
   ▶exportOptions: {_class: "exportOptions", includedLayerIds: [], layerOptions: 0, shouldTrim: false, exportFormats: []}
   ▶frame: {_class: "rect", constrainProportions: false, height: 134, width: 162, x: 16, y: 49}
   ▶groupLayout: {_class: "MSImmutableFreeformGroupLayout"}
    hasClickThrough: false
    hasClippingMask: false
    isFixedToViewport: false
    isFlippedHorizontal: false
    isFlippedVertical: false
    isLocked: false
    isTemplate: false
    isVisible: true
    layerListExpandedType: 2
   ▶layers: [{_class: "rectangle", do_objectID: "96865880-C0C0-4522-86AE-9E72E330321D", booleanOperation: -1,…},…]
    name: "78fee534-bcd8-4fd9-9bb2-7c0ff5e6cb20"
    nameIsFixed: false
    resizingConstraint: 63
    resizingType: 0
    rotation: 0
    shouldBreakMaskChain: false
   ▶style: {_class: "style", do_objectID: "6C4347CE-D5C8-4C57-8A97-85F546EA963E", endMarkerType: 0,…}
    _class: "group"
 ▶1: {_class: "symbolInstance", do_objectID: "D262A361-540F-4650-A633-3801550E24AF", booleanOperation: -1,…}
 ▶2: {_class: "group", do_objectID: "4CD06BB0-B217-48F1-8AC8-CD3773D6F951", booleanOperation: -1,…}
 ▶3: {_class: "symbolInstance", do_objectID: "34F1DB9D-4D90-4823-A2E3-776E5DE0A5C3", booleanOperation: -1,…}
  name: "iPhone 11"
  nameIsFixed: false
  overlayBackgroundInteraction: 0
  presentationStyle: 0
 ▶prototypeViewport: {_class: "MSImmutablePrototypeViewport", libraryID: "EB972BCC-0467-4E50-998E-0AC5A39517F0",…}
  resizesContent: false
```


（2）问题 2："信息预处理"，需要对基于"提取图层信息"的数据进行一系列处理。这一步的流程较为复杂，涉及到多个技术难点。我们曾经在这一步踩过许多坑，并对技术方案进行了多次调整。其中，有几个技术要点值得着重讲述，例如：

- 筛选过滤无用信息和图层：对于 sketch2json 获取的 JSON 数据中，有因 Sketch 的映射关系存在的大量对于图像还原无效的数据字段和因为设计师对于图像绘制不规范产生的无用图层信息，所以需要对其中无效数据字段和无用图层信息做一层数据过滤，减轻 JSON 的量级方便后续计算。

- 字段类型映射：瘦身处理后的数据还需要再精细化处理，其中 Sketch 的样式字段等与传统的 CSS 不一致，需要通过转换进行关系映射。对于 Sketch 的图层具体定义为文本还是图片等也需要做相关字段映射。

- 图层结构重组：sketch2json 转化后的图层嵌套结构严格依赖视觉稿规范程度，可用性较低，需要对其进行一次结构重组，将所有图层打平，后续利用图层位置信息计算出准确的包含关系。

- Order 层级深度推算：图层打平重组后，如何计算每个图层在 DOM Tree 中的层叠关系。我们可以根据初始 JSON 数据中的原始嵌套关系进行推算，为重组后的图层赋予新的层级深度值。

- Symbol 元素处理：在 Sketch 中，Symbol 作为一种特殊元素，创建后可以被重复引用，类似前端中的组件。Symbol 元素在 Sketch 转化后的 JSON 数据中，只包含了其引用 ID，它的真实图层数据被保存在了其它位置，需要通过引用 ID 检索到真实数据后替换 JSON 数据中的 Symbol 元素。

进行系列处理之后，我们得到了前端可用的图层信息，即最终的 DSL 树形数据。

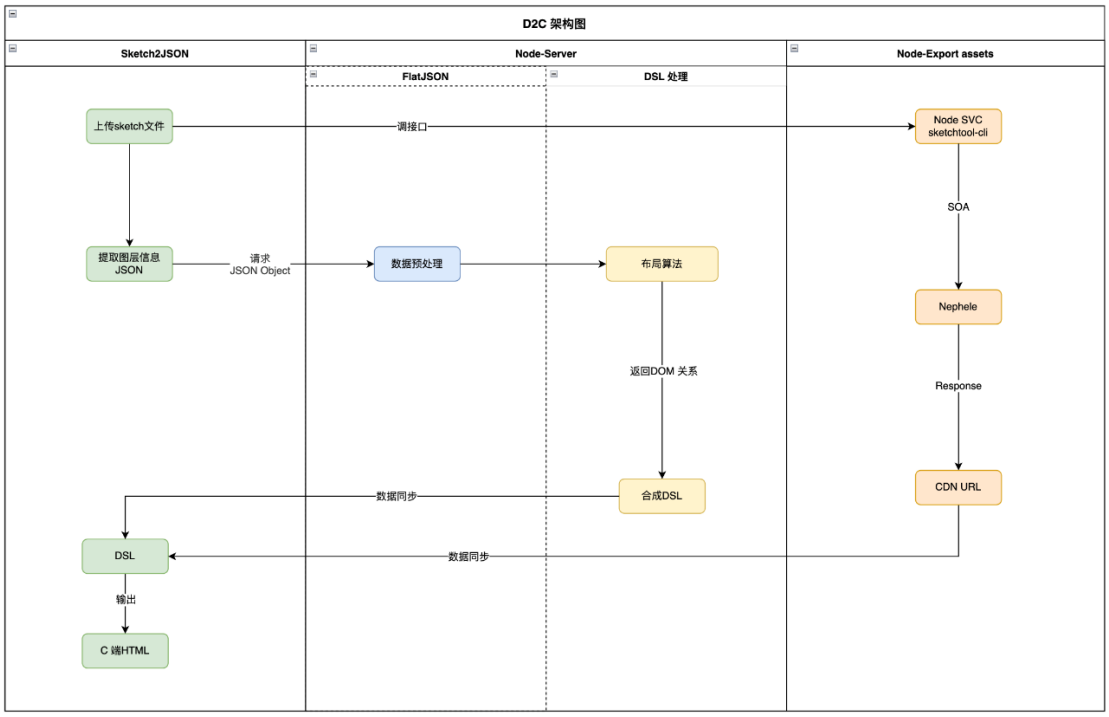
下图展示了预处理后的 DSL 数据结构：


```
{data: {type: "artboard", id: "BF71B8EB-4F48-4CAB-964D-20C00F41E57E", layoutType: 2,…}, retcode: 0}
  data: {type: "artboard", id: "BF71B8EB-4F48-4CAB-964D-20C00F41E57E", layoutType: 2,…}
    children: [{type: "box", id: "3A5CD710-1852-45DF-8A2B-464A527A778E", layoutType: 2,…},…]
      0: {type: "box", id: "3A5CD710-1852-45DF-8A2B-464A527A778E", layoutType: 2,…}
        children: [{type: "box", id: "2BA695F7-9DF4-4D7E-AB26-8483F50614E5", layoutType: 2,…}]
          0: {type: "box", id: "2BA695F7-9DF4-4D7E-AB26-8483F50614E5", layoutType: 2,…}
            children: [{type: "box", id: "FAA0DE82-D7E7-43F2-8DC7-598A312AC326", layoutType: 2,…},…]
            flated: false
            frame: {x: 0, y: 0, width: 750, height: 128}
            id: "2BA695F7-9DF4-4D7E-AB26-8483F50614E5"
            layoutType: 2
            order: 3
            position: 0
            style: {borderRadius: {topLeft: 0, topRight: 0, bottomLeft: 0, bottomRight: 0},…}
              alignItems: 0
              background: {type: "color", color: "rgba(248,248,248,0.8600000143051147)"}
                color: "rgba(248,248,248,0.8600000143051147)"
                type: "color"
              borderRadius: {topLeft: 0, topRight: 0, bottomLeft: 0, bottomRight: 0}
              boxShadow: {x: 0, y: 1, blurRadius: 0, spreadRadius: 0, color: "rgba(0,0,0,0.3)"}
              margin: {top: 0, left: 0, bottom: 0, right: 0}
              padding: {top: 1, left: 12, bottom: 21, right: 10}
            type: "box"
          flated: false
          frame: {x: 0, y: 0, width: 750, height: 128}
          id: "3A5CD710-1852-45DF-8A2B-464A527A778E"
          layoutType: 2
          order: 2
          position: 0
          style: {padding: {top: 0, left: 0, bottom: 0, right: 0}, alignItems: 1,…}
          type: "box"
        1: {type: "box", id: "8461CB08-144A-482E-BB9B-4B7E691BFC12", layoutType: 2,…}
        2: {type: "box", id: "4630C41A-4916-4F68-AA81-2112AFC15A11", layoutType: 2,…}
        3: {type: "box", id: "3B53D3E8-3C43-4708-B768-6D95B19500DA", layoutType: 2,…}
        4: {type: "box", id: "082D0F56-4232-45AA-A657-2BC83F8EE8C6", layoutType: 2,…}
      flated: true
```


（3）问题 3："构建布局关系"是所有步骤中最复杂的一步，因为它涉及到许多技术难点，且几乎没有可供借鉴的开源资料。我们可能需要通过不断的尝试来解决问题。实际上，构建布局关系是基于之前提到的 JSON 数据进行 absolute 和 flex 关系的规整，然后对 DSL 进行递归渲染。具体实现过程可以参阅下文的布局核心算法设计。

5.3 方案实现

5.3.1 架构设计



在进行系统架构设计时，我们综合了上述问题分析和思考后基本确定了技术的可行性和方向。为了满足系统的目标和功能，我们把系统拆分成了三个核心应用，分别负责执行核心任务。这张架构图展示了我们设计的架构，其中包括：视图渲染应用、布局算法服务和切图算法服务三个应用。在下文中，我们将对这些应用进行详细的阐述。

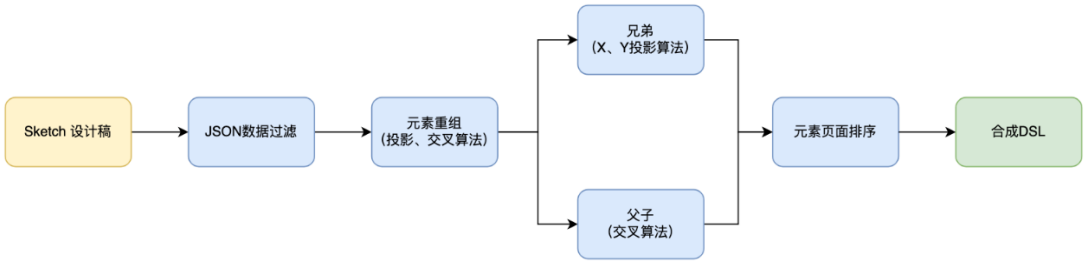
- 视图渲染应用：负责渲染转换结果界面。

通过上传的 Sketch 文件，我们可以提取出设计稿信息并将其转换为 JSON 格式。这些信息将被传递给布局算法服务进行处理。算法服务将返回包含图层信息和样式信息的 DSL，以树形结构形式呈现。我们将使用 DSL 来渲染各端页面代码。

- 布局算法服务：负责计算元素的位置和布局，以确保界面还原精准。

具体来说，它会将设计稿的矢量数据转化为前端可用的结构化数据。这个过程包括打平和再加工等主要步骤。打平的目的是将图层之间的关系（父子和兄弟关系）重新组织，再加工的目的是过滤一些干扰图层。最终，算法服务将返回中间层的 DSL 数据，该数据将用于视图渲染应用的前端代码转换。

完整流程图：

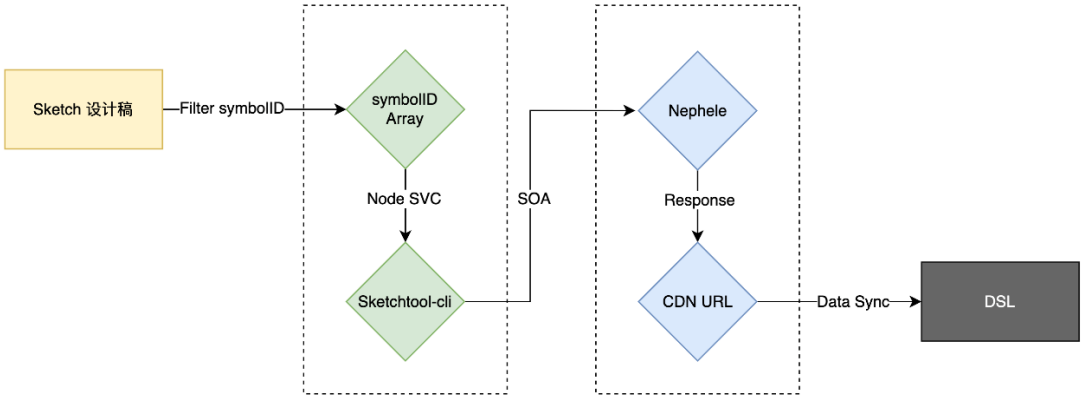


- 切图算法服务：负责将图像切割成合适的尺寸显示在界面上。

为此，我们使用一台预装了 Sketch 软件的 Mac 机器，并使用 Sketchtool-cli 处理设计稿中的图像、图标和路径。通过接收前端传递的图层 symbolID，然后根据 symbolID 将设计稿中的图片导出并存储在本地磁盘中。接下来，前端会读取本地磁盘中的对应文件夹的图片，并通过接口调用的方式将图片上传到静态资源服务器。最后，服务器会返回可生产访问的 CDN URL 给前端，前端会将所有 URL 同步到 DSL 中。

另外如何正确合理的对图片进行切割，我们做了如下分类处理：对于 sketch 里的样式属性利用 css 不易画出时将这个图层定义为图片背景、对于多种形状结合、蒙层合并的情况，首先判断其当前图层类型、再根据特定算法判断其子图层里的元素是否满足条件进行图层之间的合并，将多个单一图片合并为一整张合理的大图、对于已经采用了三角形、椭圆等特殊形状的图层确立为图片。

完整流程图：



5.3.2 核心算法设计

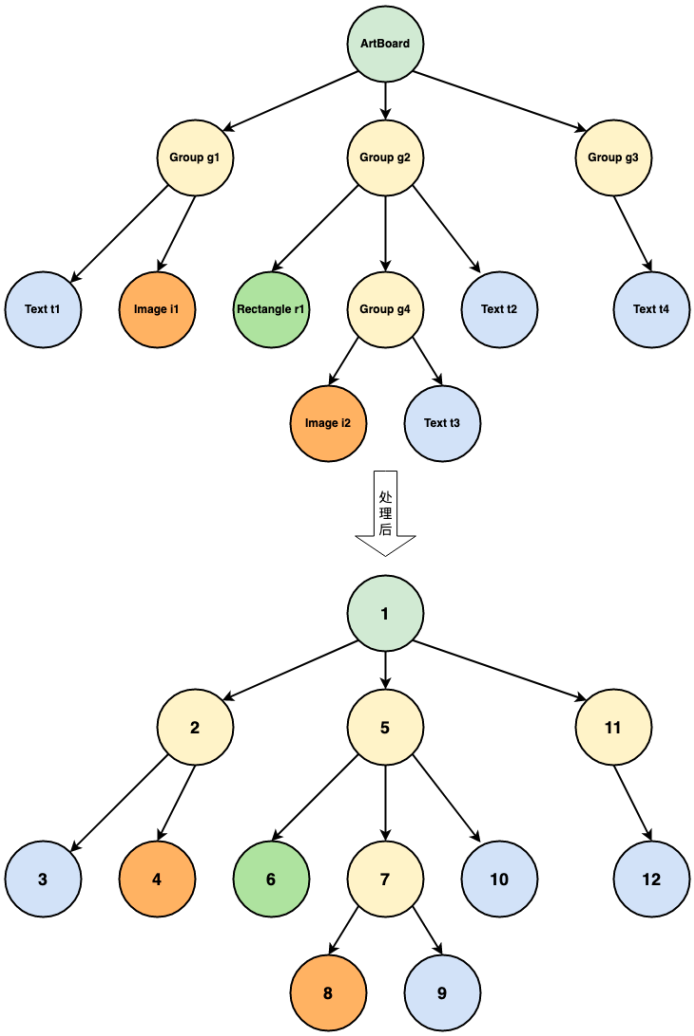
（1）Sketch 层级规律：

对于 Sketch 设计稿初始 JSON 数据中的树状图层结构，每两个图层的层级大小判断存在以下规则：

- 若图层节点 A 是图层节点 B 的子孙节点，那么，图层 A 层级 大于 图层 B 层级；

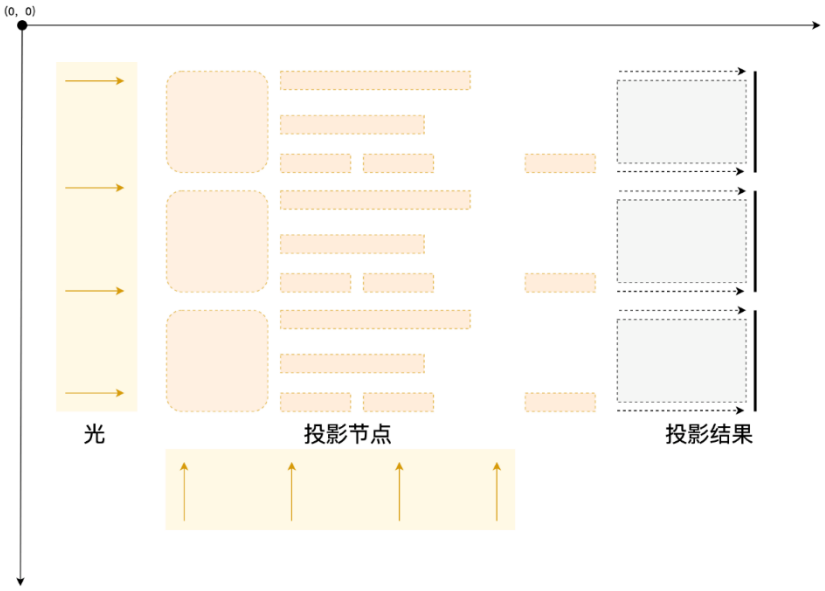
- 若两个图层节点非子孙/祖父节点，那么，这两个节点所在的最小公共子树中，若存在图层 A 的祖父节点为图层 B 的祖父节点的右兄弟节点，则表示图层 A 层级 大于 图层 B 层级。

经过分析，图层层级大小和树的先序遍历访问先后顺序一致，即根节点 < 左子树节点 < 右子树节点。



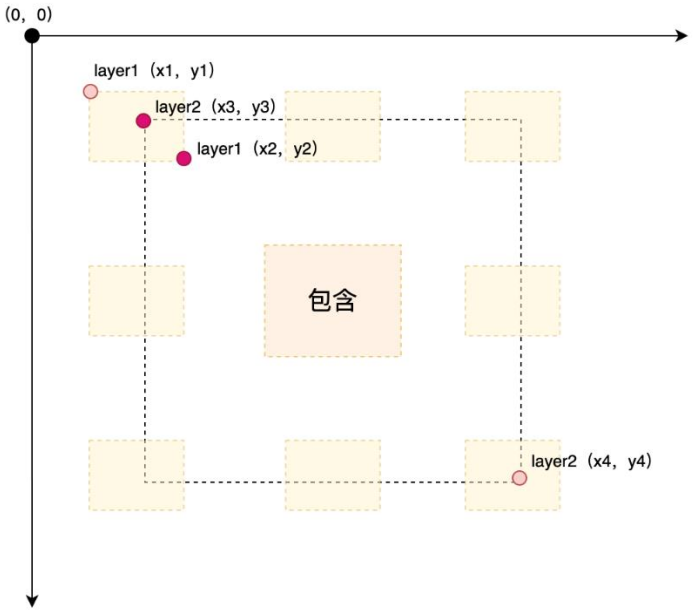
（2）投影算法：

布局关系之兄弟关系，利用光影投射的原理，通过从 X 轴和 Y 轴的方向对其中的元素进行映射，投影后所在同一个阴影块内的元素，则构成了兄弟关系。兄弟关系算法的示意图：



（3）交叉算法：

布局关系之父子关系：通过坐标计算，对于有相交关系的或包含关系的元素之间，则构成了父子关系。父子关系算法示意图：



另外在父子关系中，还需要考虑元素是否有定位。如果两个元素相交，则需要再次判断元素的大小，以决定父子关系中的父与子。为了方便后续处理布局关系，我们可以为子元素打上绝对定位的标签。

通过两个元素对角数学关系进行判断：

```
/**
 * 判断是否有交叉包含关系
 * @param layer1
 * @param layer2:
 * @returns
 */
export const isContain = (layer1: DSL, layer2: DSL): boolean => {
  if (!layer1 || !layer2 || !layer1.frame || !layer2.frame) return false;
  //layer1
  const x1 = layer1.frame.x, //左下角 x
    y2 = layer1.frame.y, //右上角 y
    y1 = y2 + layer1.frame.height, //左下角 y
    x2 = x1 + layer1.frame.width, //右上角 x
    //layer2
    x3 = layer2.frame.x, //左下角 x
    y4 = layer2.frame.y, //右上角 y
    y3 = y4 + layer2.frame.height, //左下角 y
    x4 = x3 + layer2.frame.width; //右上角 x
  return (y4 - y1) * (y3 - y2) < 0 && (x4 - x1) * (x3 - x2) < 0 ;
};
```

（4）flex 布局关系推断:

处理完父子关系和兄弟关系后，为了更好地满足布局的合理性，我们需要通过算法来推测元素中的 flex 布局方向。

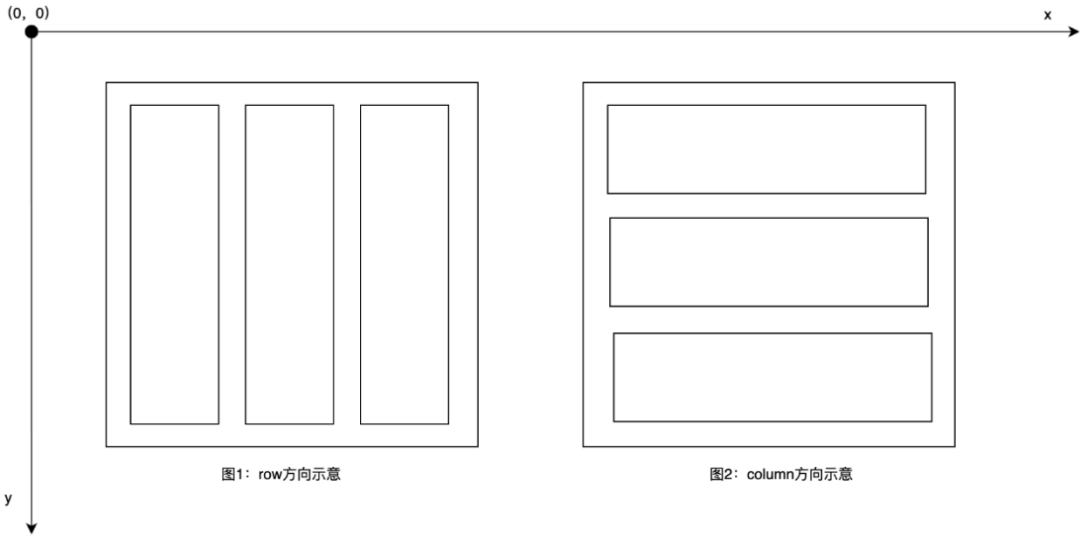


图1：row方向示意　　图2：column方向示意

- 通过判断子 layer，当他们的大小接近，并且 Y 值相差在固定阈值内且 X 值相差过大时，可以推断出该层 layer 的布局使用 flex，且 direction 为 row。

- 通过判断子 layer，当他们的大小接近，并且 X 值相差在固定阈值内且 Y 值相差过大时，可以推断出该层 layer 的布局使用 flex，且 direction 为 column。

六、视觉约束

为了更高效、准确地还原设计稿，建议输出的 Sketch 源文件做一些规范约束：

- 建议将组件保持在独立的画板中，这样可以避免转换无效的元素，方便工程师后续的代码逻辑开发。
- 保留有意义的图层，删除无效的空图层，并且尽量避免过深的图层嵌套。这样可以避免渲染树过深，影响页面加载性能。
- 图层分组（即 Sketch 中的文件夹）应该合理嵌套，避免交错放置。合理摆放元素文件夹可以有利于还原元素的 DOM 树关系。
- 除设计上有必要以外，应尽量避免将同级元素放在容器中交错排列，因为这会导致这些元素在渲染树中被视为父子关系。
- 对于相同类型的多行文本，建议使用一个文本图层，这样可以避免因为设计稿中的两个图层而将相同类型的文本转换为两个 HTML 段，从而提高代码复用率，减少代码重复。对于滤镜等特殊效果，建议使用图片来实现，以提高还原精度。

七、后续规划

（1）提高布局合理性：

- 提高 flex 弹性布局的推算准确度
- 提升 dom tree 分割准确度
- 减少 dom tree 深度

（2）提高 UI 还原度：

- 列表循环识别能力
- 提升长页面还原度能力

（3）其它：

- 减少视觉稿人工干预频次
- 适配不同团队设计规范

八、总结

D2C 技术是一种旨在提高企业产品研发效率，优化设计到开发流程，加速研发进度，以及辅

助前端开发人员探索前端智能化道路的技术。主要优势包括：

- 探索前端智能化：通过将设计稿转化为前端代码的方式，实现前端智能化。可以提高前端开发的效率，保证设计的精确实现。

- 优化产品研发流程：可以提高产品研发的效率，减少人力成本，提升团队的协作效率。更高效地进行产品的迭代和优化。

- 简化设计到开发流程：可以优化设计到开发的流程，减少沟通成本，提升工作效率。设计人员可以直接使用设计工具制作设计图，并将其转化为代码，减少设计到开发的翻译过程。

- 加速研发进度：可以帮助团队加速研发进度，更快地将产品推向市场，并可以更快速地完成设计和开发工作。同时，也可以更快地进行产品的测试和验证。

日均报错量降低 95%，携程小程序生态之自动化错误预警方案

【作者简介】携程前端框架团队，为携程集团各业务线在 PC、H5、小程序等各阶段提供优秀的 Web 解决方案。产品涉及各类前端/Node 端应用框架、研发工作台、前端中台化、静态资源发布系统等。当前主要专注方向包括：新一代研发模式探索，Rust 构建工具链路升级、Serverless 应用框架开发、在线文档系统开发、低代码平台搭建、适老化与无障碍探索等。

摘要

携程小程序自动化错误预警方案是一套完整且通用的小程序前端错误监控方案。此方案提供小程序错误自动采集 SDK，并对错误所在的页面路径进行偏移矫正，能够准确通知相应的开发负责人；将错误信息分为生产、测试、开发 3 种阶段，根据错误发生的阶段提供相应的错误预警及处理方案；开发负责人只需在小程序管理平台配置告警所需信息，即可快速接入、实时监控生产报错、生成告警通知；源码映射能力可以帮助开发者快速定位错误原因，提升修复效率。接入此方案可实时捕获开发环境错误，确保在小程序发布上线前发现并解决业务报错，可极大地减少线上小程序的错误量。

一、为什么要做错误监控

携程小程序产品是 to C 的，用户访问量大，使用的机型和场景复杂多样；当用户遇到问题时，我们无法保证所有用户都及时反馈，且提供明确、详细的信息。因此，为了保证系统稳定运行、提高用户体验，我们需要主动收集错误信息、实时监控错误变化趋势、及时解决。

为了提升小程序代码质量、守护线上页面质量，Web 前端团队提出了携程小程序的小程序错误预警方案，可分为 4 个阶段，如图 1 所示：

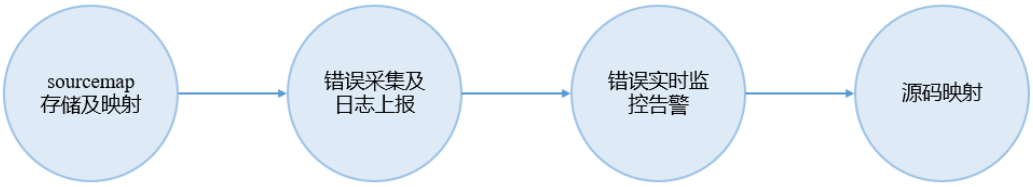


图 1 小程序错误预警分阶段图示

携程小程序自动化错误预警方案在自动化部署阶段将 buildId 注入代码，并通过固定 robotId 的方式获取对应的 sourcemap 文件；客户端运行时自动采集错误信息，同时进行埋点上报；服务端处理存储的错误数据，并按照页面路径、错误类型等维度进行分类；告警平台将实时监控报错情况，根据自定义的告警规则给开发负责人发送告警通知，详细的方案设计参见图 2：

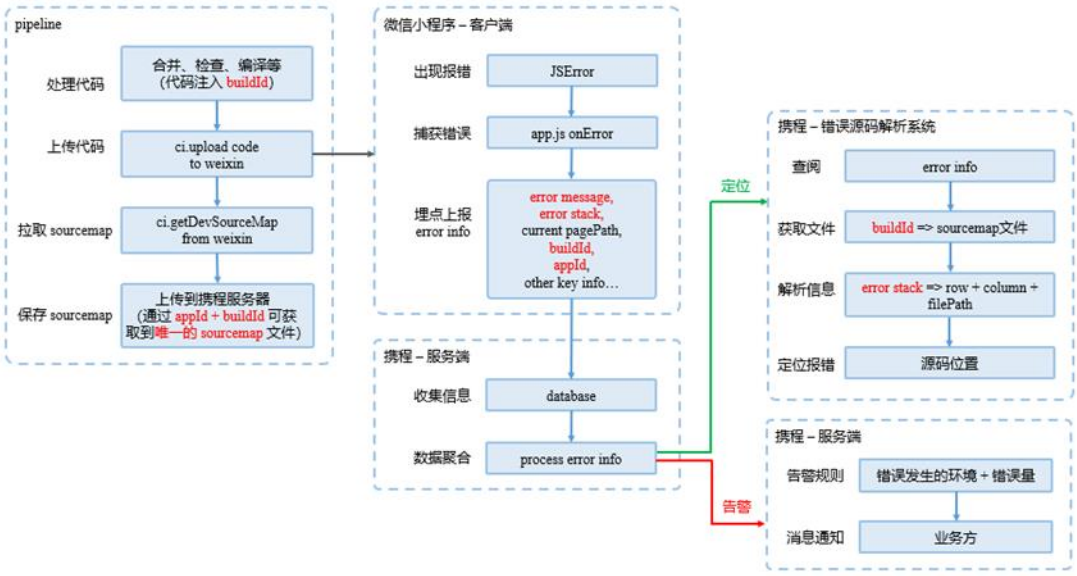


图 2 错误预警方案设计图

二、sourcemap 存储及映射

携程小程序发布是通过 ci 工具上传小程序代码实现的。在发布前，我们会将无线持续交付平台（Mobile Continuous Delivery，下文统称为 MCD 发布平台）创建的 buildId 注入到小程序代码中，以供运行时获取。由于微信不支持通过版本信息或其他的唯一信息获取对应的 sourcemap 文件，我们设计了一套获取相应 sourcemap 的方案：

我们通过固定 robotId 的方式上传源码，从而获取对应的 sourcemap 文件；在成功上传小程序代码后，将延迟 5 秒后使用 ci 工具拉取最近一次上传版本的 sourcemap 文件；此外，将会比对 sourcemap 的解压源文件中 buildId 与当前上传代码中的 buildId 是否一致，如果不一致将再次获取，从而确保上传的源码和获取的 sourcemap 文件是匹配的，保证源码映射的准确性。

随后，将 sourcemap 文件存储到携程的文件服务器，并将服务端返回的文件链接地址、当前的 buildId、时间点以及自定义备注等信息以对象的形式保存到数据库中，即可实现 buildId 与 sourcemap 文件的信息存储及映射。

三、错误采集及日志上报

在小程序客户端运行时，我们会在 wx.onError 和 wx.onUnhandledRejection 的回调函数中捕获到所有的错误信息，随后进行错误信息的筛选、处理和提取。其中，错误的关键信息有：pagePath, message, stack 等。此时，我们会对捕获到错误时所在页面的 pagePath 进行偏移纠正，避免页面偏移导致误报的情况。那么如何实现错误偏移纠正和错误采集呢？

首先，在上传小程序代码包前，我们会通过读取 app.json 的内容，获取所有注册的页面路径，并将其注入到小程序代码中。接着，在捕获到错误信息时，我们会对 error stack 进行解析，从上往下进行页面路径贪婪匹配，匹配到的第一个页面就是发生报错的页面。随后，将

错误信息、当前的 appId 以及发布时注入的 buildId 一同通过用户行为数据采集系统（User Behavior Tracking，下文统称为 UBT）的错误接口上报，上传到 UBT 的服务端。至此，便完成了一次错误采集与上报的流程。

四、数据聚合

采集到上报的错误数据后，服务端会对数据进行清洗、过滤、聚合等操作，随后将预处理后的数据存入 APM 的 MySQL 数据库中。通过内部的 APM 监控平台主动读取数据库中的数据，实现各种自定义维度的数据聚合，比如分别根据部门、客户端平台（Platform）、报错页面路径（PagePath）、小程序版本（version）、Top50 错误类型等维度进行排序并展示数据列表（如图 3）。

这些处理后的信息为下文的错误告警提供了数据支撑，告警主要依据的是报错页面路径（PagePath）及 Top50 错误类型这两个维度。

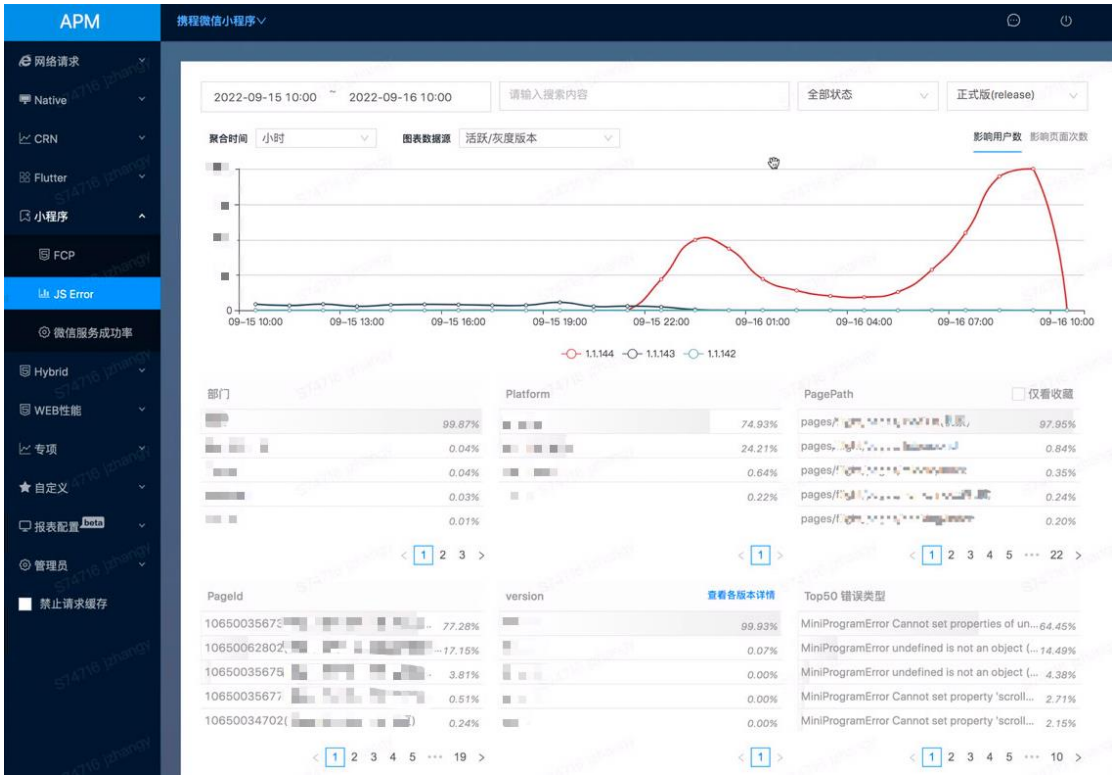


图 3 APM — 携程微信小程序 JS Error

五、错误实时监控告警

除了业务主动去 APM 上实时查看报错情况，我们还提供了错误告警机制，会在不同阶段对采集到的错误进行消息推送提醒，详细流程如图所示：

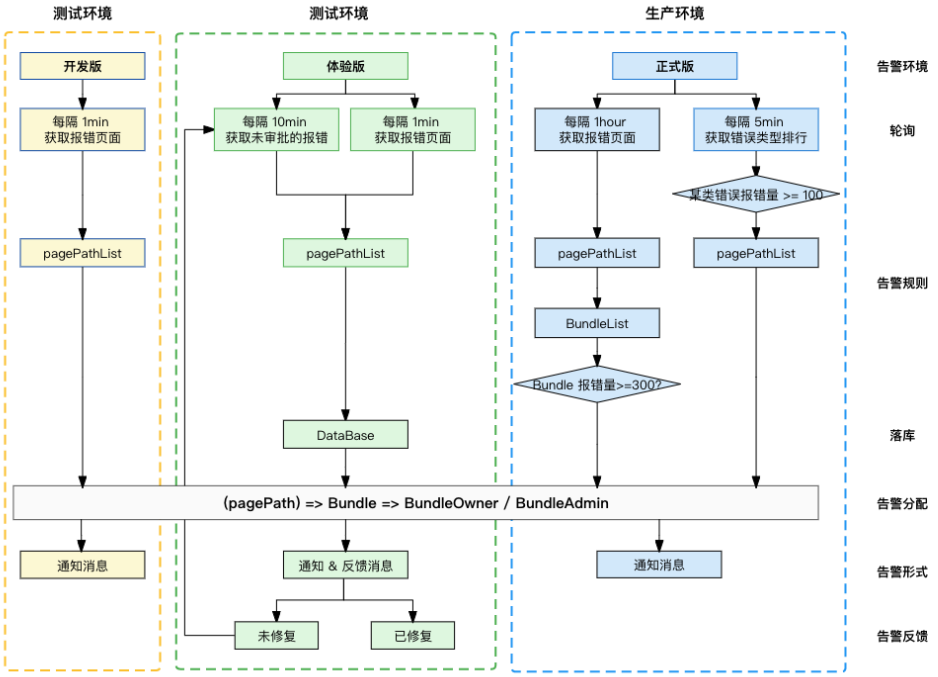


图 4 小程序错误告警流程

从上图可看出，错误告警贯穿整个小程序的开发流程，共分为三个版本，包括开发版、体验版以及生产版。错误告警服务每隔一段时间轮询不同环境的报错信息（调用 APM 提供的数据聚合接口），若报错量超过我们设置的告警阈值（详情可见表 1），则会根据错误页面找到对应的 Bundle(最小的业务线单位)，然后再找到对应的开发人员或管理员（对应关系已在小程序管理平台中注册），通过携程内部沟通工具（下文统称为 TripPal）将错误提醒推送给相关人员，在收到信息后可以通过点击信息直接跳转到具体的 APM 错误页面中，在 APM 中可以进行错误的反查与定位。下表展示了各阶段告警配置的详细情况：

	轮询时间	告警形式	告警依据
开发版	1min	通知消息	页面路径报错量 ≥ 1
体验版	10min	通知消息 反馈消息	页面路径报错量 ≥ 1
生产版	5min	通知消息	错误类型报错量 ≥ 100 的页面路径
生产版	1hour	通知消息	Bundle 下所有页面路径报错数量之和 ≥ 300

表 1 各阶段告警配置

（1）开发版

在开发阶段，及时捕获错误有助于开发人员第一时间进行修改，因此每隔 1 分钟，服务端会获取每个页面路径的报错量，只要页面有报错，将推送消息至该 Bundle 的相关人员。

图 5 开发版告警

（2）体验版

体验版是一个完整稳定的版本，是可以作为发布候选的版本，因此对该环境的报错进行监控是十分必要的。体验版除了能接收到与开发版一致的错误信息外，还会发送反馈消息，由 Bundle 管理员进行审批，审批的意义在于告知管理员当前要发布到生产的代码中存在错误，需要管理员知晓并判断是否修改，若 10 分钟内未处理，将再次发消息进行提醒。

图 6 体验版告警

在 MCD 发布平台生成体验码时，若仍有 Bundle 未处理体验版报错，将在发布群中再次提醒：

图 7 MCD 构建结果通知

以上两个阶段都属于开发测试阶段，目的在于将所有的业务报错信息都在开发阶段暴露出来，然后让各个业务及时修改，保证发到生产的代码没有业务报错。

（3）生产版

总有意外的情况会导致生产运行出现错误，为了更快更全面的感知生产报错，我们会对生产错误进行两个维度的监控：

- 每隔 5 分钟，服务端获取某类错误的报错量超过 100 的页面路径，随即发出通知消息至相应的开发和管理人员，管理人员此时需要重视下频繁报错是否会对生产业务造成影响并给 PMO 反馈修复的时间以及是否需要紧急发布。

图 8 生产版告警（错误类型维度）

- 每隔 1 小时，服务端会获取每个页面路径的报错量，然后聚合到每个具体的 Bundle，如果发现 Bundle 一小时报错量超过了 300，也将发送告警信息。

图 9 生产版告警（Bundle 报错量维度）

由于通知消息本身不具备强制性、约束力，因此我们接入了公司的告警治理系统（下文统称为 BigEyes）。生产版报错将通过 BigEyes 进行统一通知，如果告警事件未在预设的响应时间内被响应，则进行通知升级，即通知原对象的上级，以此提高业务方的重视程度及故障恢复效率。

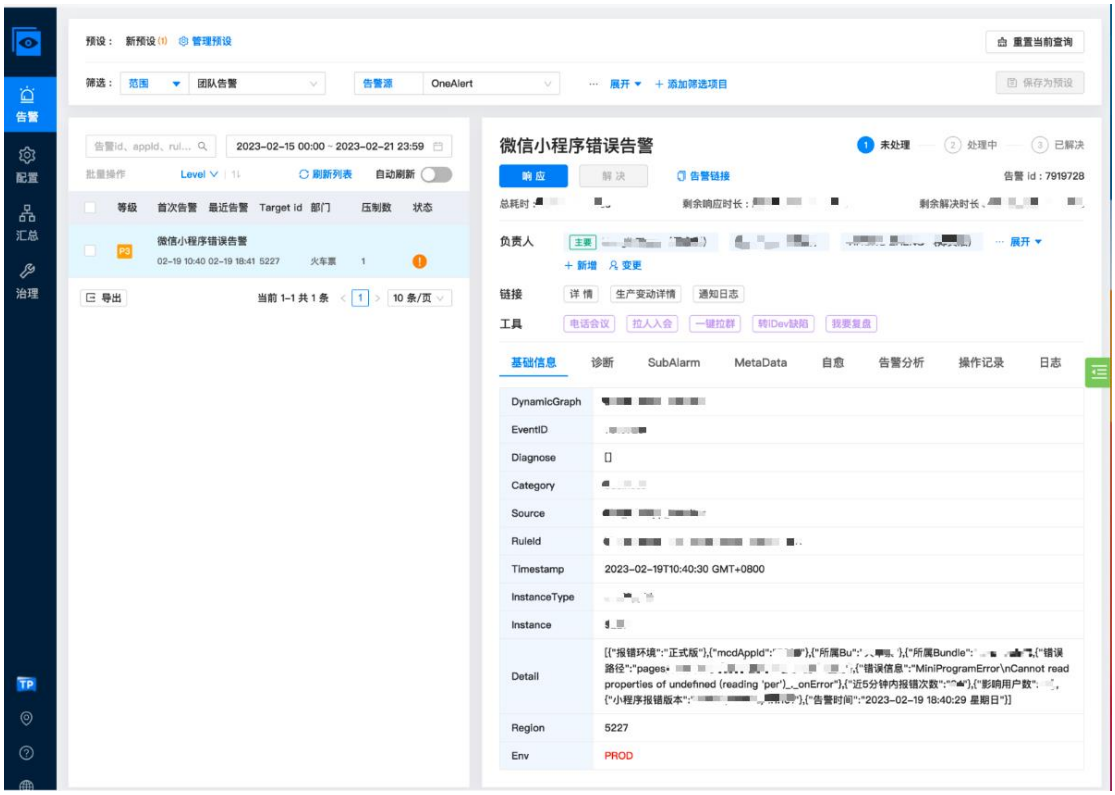


图 10 小程序告警接入 BigEyes

为了便于独立小程序的接入，在现有方案的基础上，我们设计并实现了一套通用的可定制化的错误告警配置服务。独立小程序在引用错误采集 SDK 的前提下，通过在小程序管理平台配置告警所需信息，包括用于告警的专属 Trippal 服务号、开发负责人、不同环境（正式、体验、开发）的告警阈值等，即可在触发告警条件时，收到服务号发来的错误通知。

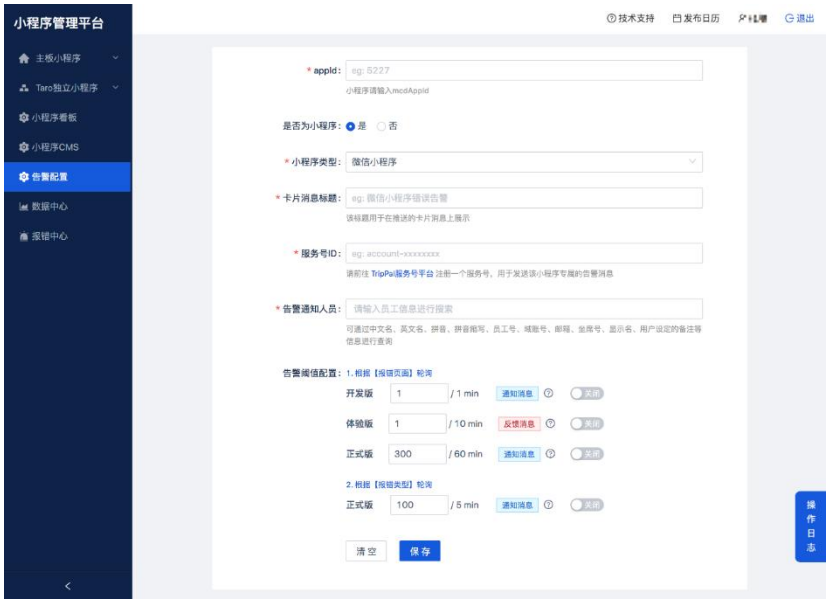


图 11 独立小程序错误告警配置页面

六、源码映射

从 APM 平台查看错误信息时，可以通过错误的详情页面进入小程序错误源码解析系统。由于我们在错误采集时会将 buildId 和错误相关信息一起上报至服务端，因此错误详细信息内包含报错代码包对应的 buildId（如图 14）；在点击错误堆栈上的链接跳转至错误源码解析系统时，系统将根据 buildId 获取到相应的 sourcemap 文件；随后，根据错误信息找到对应的 sourcemap 内容、依据错误位置进行反解析；最后，使用 Monaco Editor 将源码展示在页面中，开发者可以据此查看错误所在的源码位置，从而助力开发者快速、准确定位错误原因。

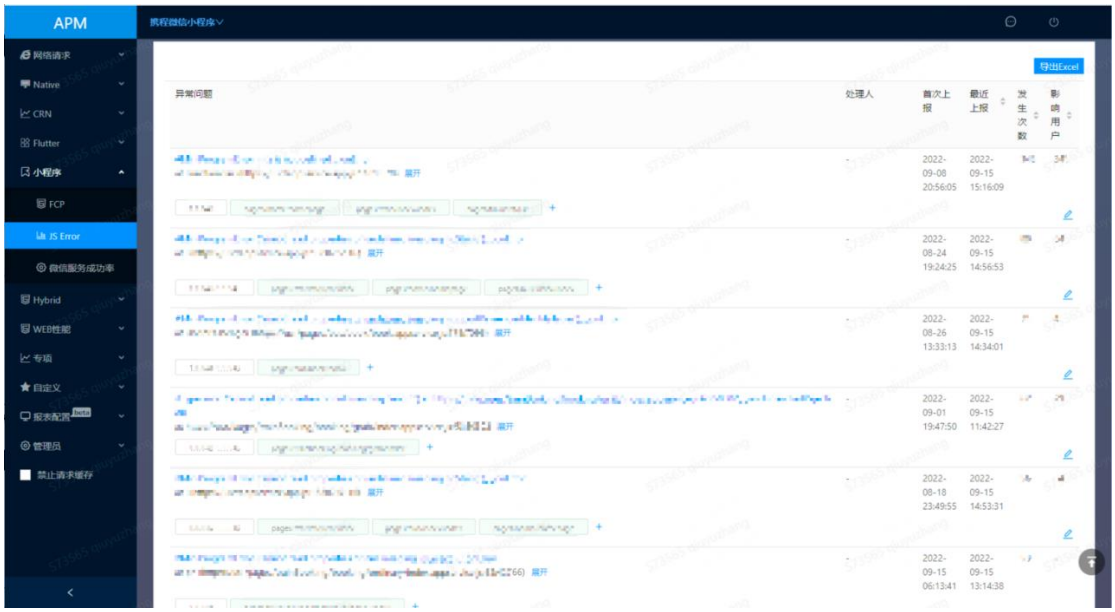


图 12 小程序 JS Error 概览页面

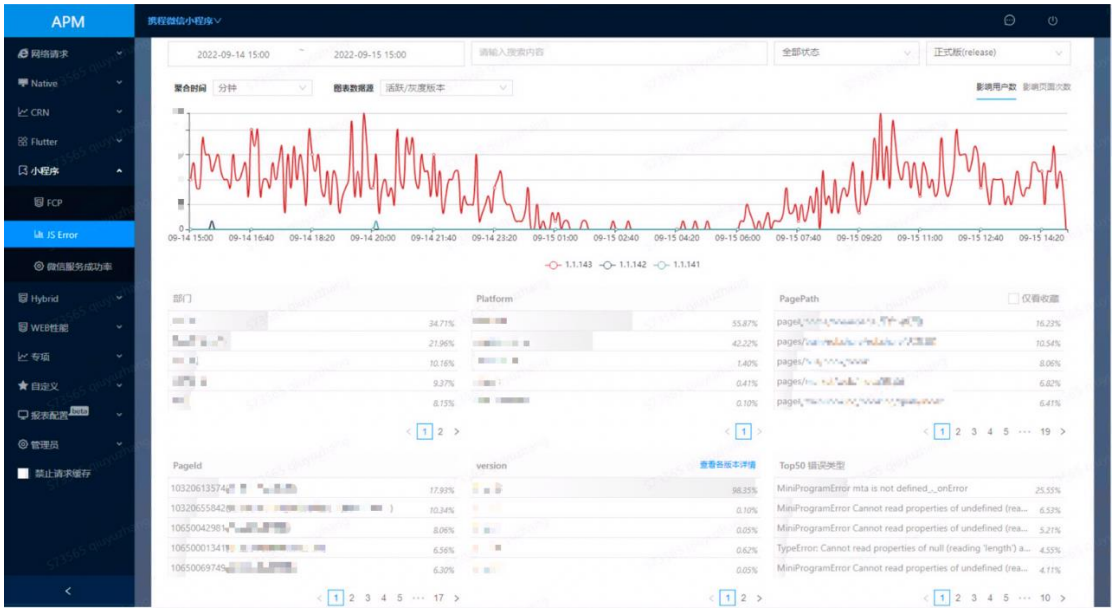


图 13 小程序 JS Error 聚合、分类页面

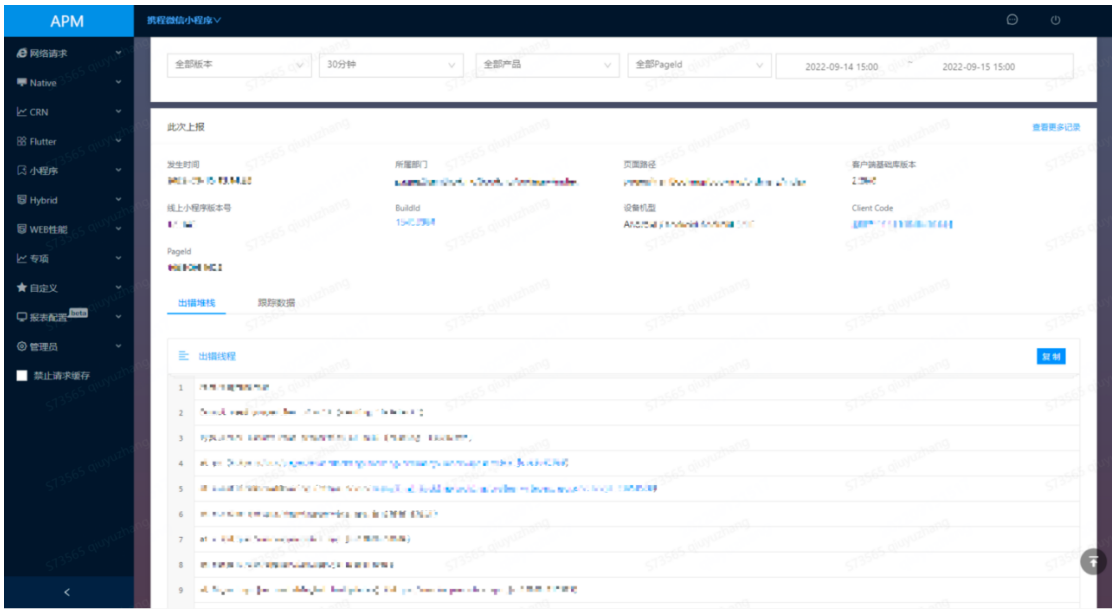


图 14 小程序 JS Error 详情页面

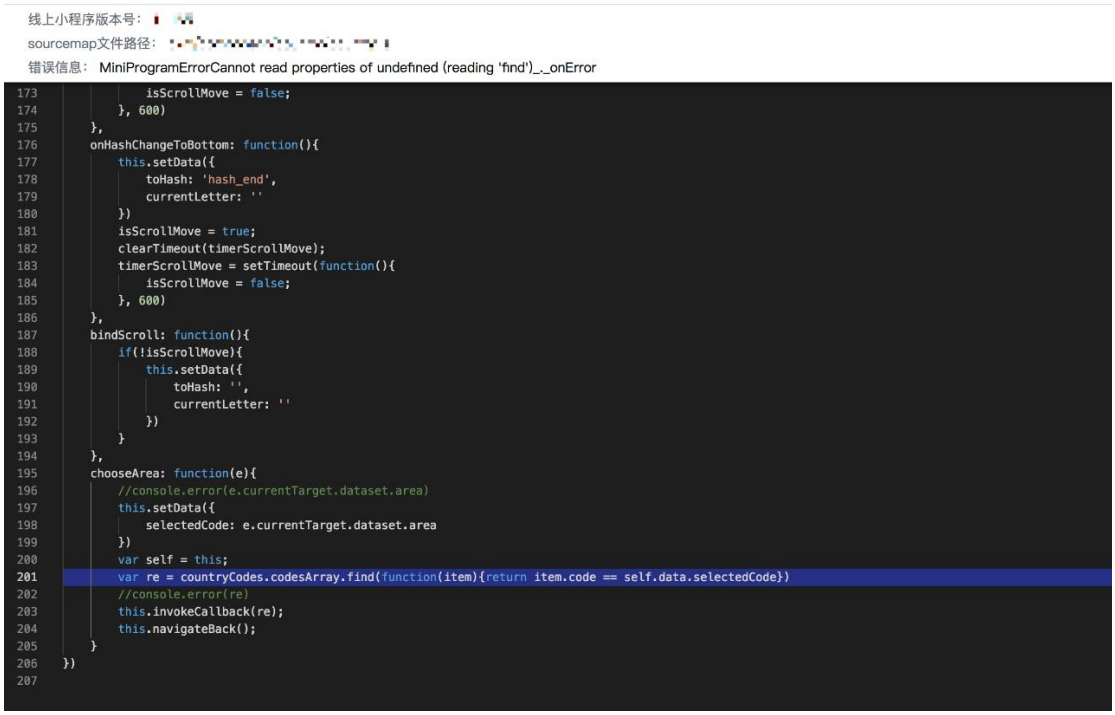


图 15 小程序 JS Error 源码解析页面

七、总结

自动化错误预警方案可以帮助各业务方在开发和测试阶段捕获运行时可能出现的业务报错，避免此类错误发布到生产环境，从而降低线上的错误量。在生产环境可以通过告警机制实时监控错误趋势，当出现告警时可通过源码映射系统快速定位错误原因、提高开发者的修复效率。

该方案极大地提升了小程序的运行稳定性，可以有效提升用户体验。自主板小程序接入携程小程序自动化错误预警方案以来，其日均线上报错量降低了 95%左右，受到各业务部门的好评，印证了本方案的可行性和有效性。

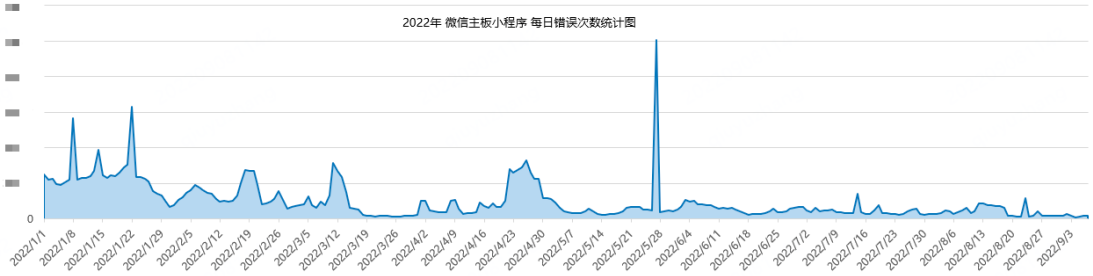


图 16 2022 年携程微信主板小程序每日错误次数统计图

携程小程序内嵌 webview 实践指南

【作者简介】思语，携程高级前端开发工程师，关注互动营销领域。Olivio，携程高级前端开发工程师，关注 React Node 组件化。Stone，携程高级研发经理，关注跨端解决方案，云原生落地等领域。

一、背景

这篇文章将向大家分享团队在小程序 webview 方面的开发心得，以微信小程序为主要环境，介绍在业务开发中处理**小程序 webview 内嵌 H5 所遇到的问题及解决方案。**具体将从小程序平台与 H5 差异、小程序内嵌 webview 通信、小程序 webview 常见问题展开叙述。

二、平台差异

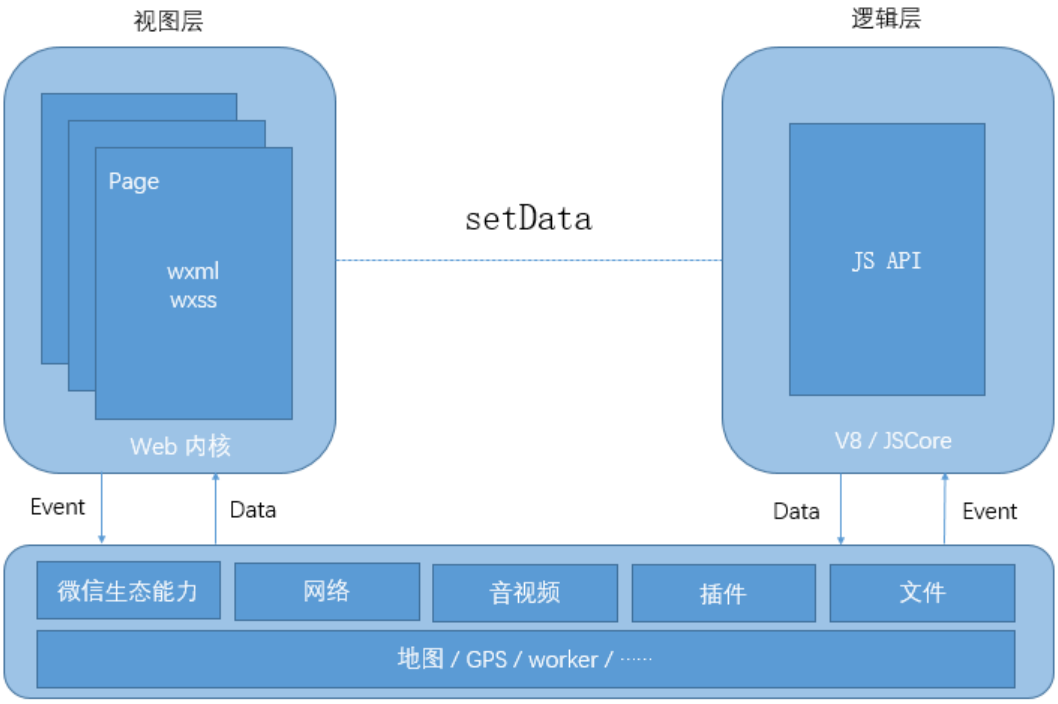
下面将浅析并回顾一下小程序和 H5 在渲染方面的几点差异。

	小程序	H5
离线能力	高	低
页面切换体验	高	中
首屏速度	离线能力是天然优势	SSR等技术不逊于小程序原生页面
二次渲染速度	低	高
列表性能	很低	高
操作响应	低	高
开发灵活性	低	高
api能力	高	中

2.1 小程序方面

以微信小程序为例，相信今天大部分的读者对微信小程序的系统架构都比较熟悉了，总体来讲分为两部分：

- View 视图端通过小程序的框架，将用户采用 WXML 和 WXSS 描述的 UI 信息处理成 H5 元素，最终交给 WebView 去渲染；
- 逻辑层运行 JS 逻辑，并且可以调用具有微信开放能力的 JSAPI。逻辑和视图分离，通过事件和数据彼此之间建立联系。



微信小程序使用 WebView 渲染，与原生客户端的是两套不同的视图渲染体系。一个小程序存在多个界面，所以渲染层存在多个 WebView。逻辑层采用 JSCore 线程运行 JavaScript 脚本。这两个线程间的通信经由小程序 Native 侧中转，逻辑层发送网络请求也经由 Native 侧转发。

如此设计的初衷是为了管控和安全，微信小程序阻止开发者使用一些浏览器提供的，诸如跳转页面、操作 DOM、动态执行脚本的开放性接口。将逻辑层与视图层进行分离，视图层和逻辑层之间只有数据的通信，可以防止开发者随意操作界面，更好地保证了用户数据安全。同时小程序设计一套组件框架—— Exparser ，基于这个框架内置了一套组件，以涵盖小程序的基础功能，便于开发者快速搭建出任何界面，同时也提供了自定义组件的能力，开发者可以自行扩展更多的组件，以实现代码复用。

值得一提的是，内置组件有一部分较复杂组件是用客户端原生渲染的，同时微信团队又通过结合 Flutter 和 LV-CPP，把实现代码收敛在 C++ 和 Dart 上，进一步简化了基于小程序技术栈实现跨平台业务开发的框架维护成本，以提供更好的性能。

2.2 小程序 webview 内嵌 H5

H5 页面投放在小程序 WebView，在配置完合法域名后，即可在小程序应用中展示。那么，针对不同厂商小程序，可能法务、厂商合规有所差异，需要 H5 判断所在的环境，去调用不同 api 方法，展示不同的业务页面。

在携程内部封装了小程序 CWX 的 SDK，小程序端主要采用原生+Taro 框架，H5 这块主要是 NFES(React)和 Vue，无论是哪一段端都通过一个 CWX 来连接，内部封装了各端通用的功能比如登录、发布、支付、个人中心等功能，这些功能都可以直接通过 CWX 这个中间件进行调用。

并且，H5 在检测到当前处于小程序 webview 环境下时，会根据环境异步加载 SDK 文件、及其厂商的 JS-SDK，初始化小程序版本 wx.config。这里的关键点是我们要做个 api 调用的队列，因为 sdk 加载异步的过程，如果期间页面内发生了 api 调用，那肯定得不到正确的响应。因此要做个调用队列，当 sdk 初始化完毕之后再处理这些调用。其实 CWX 原理很纯粹，如果你想实现多端适配，那么只需要根据所在的环境去加载不同的 sdk 就可以了。

> 下面简要列举一下工作中常用的几个小程序环境判断：

```
let ua = window.navigator.userAgent.toLowerCase();
const globalObj = {
    testDataArr: [],
    doJSReadyFuncExecuted: false,
    errorInfo: '',
    miniappSDK: null,
    miniappType: '',
    actionQueue: [],
    MINIAPP_TYPE: {
        WECHATMINIAPP:  'WECHATMINIAPP',
        WECHATAPP:      'WECHATAPP',
        OLDQUICKAPP:    'OLDQUICKAPP',
        NEWQUICKAPP:    'NEWQUICKAPP',
        ALIPAYAPP:      'ALIPAYAPP',
        BAIDUAPP:       'BAIDUAPP',
        TOUTIAOAPP:     'TOUTIAOAPP',
        QQAPP:          'QQAPP'
    },
    // 加载不同应用环境的SDK，比如微信、支付宝、快应用等
    JSSDK_URL_OBJ: {
        WECHATMINIAPP: 'https://res.wx.qq.com/open/js/jweixin-1.6.0.js',
        WECHATAPP: 'XXXXXX',
        OLDQUICKAPP: 'XXXXXX',
        NEWQUICKAPP: 'XXXXXX',
        ALIPAYAPP: 'XXXXXX',
        BAIDUAPP: 'XXXXXX',
        TOUTIAOAPP: 'XXXXXX',
        QQAPP: 'XXXXXX'
    },
    bversion: '1.0.0'
}

function isWechatMiniapp () {
    //  ... 这里包括一些设备、应用的判断，包括 isAndroid、isQQ 等等
}

export {
    isAndroid,       // 判断H5页面是否处于安卓系统
    isWechatMiniapp, // 判断H5页面是否处于微信小程序环境
    ......
}
  ...
}

window.native = new Native();
export default native;
```

使用时的注意事项：

使用前，最好查阅相应小程序的文档，因为各个小程序对 API 的支持程度不同。引用 bridge.js 的方式视情况而定，因为 bridge.js 引入 JSSDK 的方式是 为 head 标签添加 script 标签，若在 head 标签中引入 bridge.js，就会报错若打开 h5，显示“页面访问受限”之类的提示信息，

可尝试下方的操作：（这种情况，一般是打开测试环境的 h5 url 时出现）勾选 IDE 中的“忽略 webview 域名合法性检查”和“忽略 request 域名合法性检查”。

【快应用相关】
目前 Vivo、Oppo、华为三家厂商已支持新版快应用，Vivo、OPPO 已上线，华为正在测试中，小米不支持。对于新版快应用，若 H5 页面需要调用新版快应用 JS-SDK 中提供的 API，需要提前将该 H5 链接的域名配置到可信任的网址里(应写成正则表达式的形式进行配置)。

【头条相关】
头条小程序的 redirectTo、navigateTo 等页面跳转的 api 只支持 url 为 / 开始的绝对路径。

【支付宝相关】
目前的 1.0.73 版 bridge.js 判断是否处于支付宝小程序的方法，会将 h5 处于支付宝小程序、h5 处于支付宝内置浏览器都判断为处于支付宝小程序内。因此，在调 my.XXXX 之前，需要先调环境工具函数判断一下，确保确实是处于支付宝小程序内，而非支付宝内置浏览器内。

三、小程序内嵌 webview 通信

3.1 小程序中 h5 页面 onShow 和跨页面通信的实现

首先想到的是 onShow 方法的实现，之前有人提议用 visibilitychange 来实现 onShow 方法，但调研过后，发现这种方式在 ios 中表现符合预期，但是在安卓手机里，是不能按预期触发的。

于是就有了下面的方案，这个方案需要 h5 和小程序的 webview 都做处理。核心思想：利用 webview 的 hash 特性。

- 小程序通过 hash 传参，页面不会更新（这个和浏览器一样）
- h5 可以通过 hashchange 捕获最新参数，进行自定义逻辑处理
- 最后执行 window.history.go(-1)

为什么要执行 window.history.go(-1)？因为 hash 变更会导致 webview 历史栈长度+1，用户需要多一次返回操作。但这一步明显是多余的。同时 window.history.go(-1)后，会把 webview 在 hash 中添加的参数去掉，还能保证和之前的 url 一致。

3.2 注意点

出于平滑接入的考虑，不能上来搞一刀切，要保证现有页面不再做任何修改的情况下继续访问。新能力要通过额外参数区分，如：检测 url 中的 query 部分，带有 __isonshowpro=1 再进行通过 hash 方式传参。改造原有逻辑，让__isonshowpro=1 时，hash 处理逻辑优先级最高参数定义，在前面加入了两个下划线，目的是为了区分 url 中正常的参数。我们来看看 h5 端的 sdk 是怎么实现的。

```
import util from './util';
class WASDK {
  constructor(){
    if('onhashchange' in window && window.addEventListener && !WASDK.hashInfo.isInit){
      WASDK.hashInfo.isInit = true;
      // 绑定hashchange
      window.addEventListener('hashchange', ()=>{
        if (util.getHash(window.location.href, '__wachangehash') === '1') {
          const jsticket = window.native && window.native.adapter && window.native.adapter.jsticket || null;
          const ua = navigator.userAgent;
          // 非安卓系统要重新设置wx.config
          if (jsticket && !(ua.indexOf('Android') > -1 || ua.indexOf('Adr') > -1)) {
            window.wx.config({
              debug: false,
              appId: jsticket.appId,
              timestamp: jsticket.timestamp,
              nonceStr: jsticket.noncestr,
              signature: jsticket.signature,
              jsApiList: ['onMenuShareTimeline', 'onMenuShareAppMessage', 'onMenuShareQQ',
              'onMenuShareQZone', 'onMenuShareWeibo', 'scanQRCode', 'chooseImage',
              'uploadImage', 'previewImage', 'getLocation', 'openLocation']
            })
          }
          // 触发缓存数组的回调
          WASDK.hashInfo.callbackArr.forEach(callback=>{
            callback();
          })
          setTimeout(()=>{
            window.history.go(-1);
          }, 0);
        }
      }, false)
    }
  }

  static hashInfo = {
    isInit: false,
    callbackArr: []
  }

  onShow(callback){
    if (typeof callback === 'function') {
      WASDK.hashInfo.callbackArr.push(function(){
        if(util.getHash(window.location.href, '__isonshow') === '1'){
          callback();
        }
      })
    } else {
      util.console.error(`参数错误，调用onShow请传入正确callback回调`);
    }
  }

  serviceDone(obj, condition){
    if(obj && obj.key){
      const message = {
        // 消息名称
        key: obj.key,
        // 消息体
        content: obj.content || '',
        // 触发条件
        trigger: {
          // 类型 'immediately'在下一次onshow中立刻触发， 'url'，在找到指定h5链接时触发，'path'在打开指定小程序路径时触发
          type: 'immediately',
          // 条件内容，immediately是为空，url是为h5链接地址，path是为小程序路径
          content: ''
        }
      };
      // 解析触发条件
      condition = condition || 0;
      // 如果是路径
      if(typeof condition === 'string' && (condition.indexOf('http') > -1 || condition.indexOf('pages/') > -1)){
        // 设置消息触发条件
        message.trigger = {
          type: condition.indexOf('http') > -1 ? 'url' : 'path',
          content: condition
        }
      }
      // 发送消息
      wx.miniProgram.postMessage({
        data: {
          messageData: message
        }
      });
      // 如果不是url或者path触发，则对conditon是否需要返回进行判断
      if(message.trigger.type === 'immediately'){
        // 查看是否需要返回指定的层级，兼容传入'-1'字符串这种类型的场景
        try{
          condition = parseInt(condition, 10);
        }catch(e){}
        // 保证返回级数的正确性
        if(condition && typeof condition === 'number' && !isNaN(condition)){
          this.handler.navigateBack({delta: Math.abs(condition)});
        }
      }
    }else{
      util.console.error(`参数错误，调用serviceDone方法，传入的对象中不包含key值`);
    }
  }
  ...
}
window.native = new Native();
export default native;
  ...
}
window.native = new Native();
export default native;
```

总结下来是两点：

- onShow 方法的实现

绑定一个 hashchange 事件（这里做了防止重复绑定事件的处理），将传入的 onShow 自定义事件缓存在一个数组中，hashchange 触发时，根据特有的标志位__isonshow 和__wachangehash 确定是否触发。

- serviceDone 方法的实现

触发条件：immediately 表示最近的一次 onShow 触发，或者自己指定 url 通过 wx.miniProgram.postMessage 发送数据。

浏览器访问资源是通过 URL 地址，如果内嵌 H5 的地址不发生变化，那么 web-view 访问资源会从缓存里取，而缓存里并没有最新的数据，这就导致了服务端的最新资源根本无法到达浏览器，这也解释了为什么修改 Nginx 的 Cache-Control 配置也无法生效的原因。

所以，要想彻底解决及时刷新，必须让 web-view 去访问新的地址。我们假定小程序访问的 URL 地址为：https://www.yourdomain.com/101/#/index ，其中 101 就是构建的一个版本号，每次递增，保证次次不同即可。

四、webview 常见难题与解决方案

小程序和 h5 之间的通信基本上常用两种方式，一个是 postMessage，这个方法大家都知道，只有在三种情况才可以触发，后退、销毁和分享。但也有个问题，就是需要注意这个方法是基础库 1.7.1 才开始支持的，1.7.1 以下就只能通过第二种方法来传递数据，也就是设置和检测 webview 组件的 url 变化，类似 pc 时代的 iframe 的通信方式。

sdk 这块怎么做呢，定义一个 share 方法，首先需要检测下基础库版本，看是否支持 postMessage，如果支持直接调用，如果不支持，把分享参数拼接到 url 当中，然后进行一次重载。也就是说，通过 url 传递数据有个缺点，就是页面可能需要刷新一次才能设置成功。

目前在 webview 环境下支持支持的几种通用业务：

4.1 左上角返回

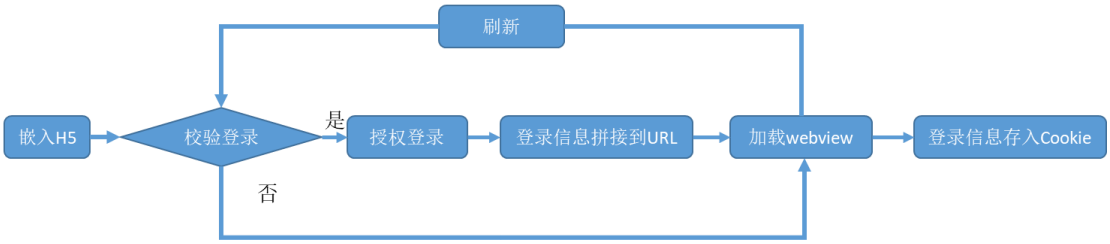
在访问小程序 webview 页面时，首先进入的是一个空白的中转页，然后进入 h5 页面，这样左上角就会出现返回按钮了，当用户按左上角的返回按钮时候，页面会被重载到小程序首页去，这个看似简单又微小的动作，对业务其实有很大的影响。

经过我们的数据统计发现，左上角返回按钮点击率高达 70%以上，因为这种落地页一般是被用户分享出来的，以前纯 h5 的时候只能通过左上角返回，所以在小程序里用户也习惯如此；第二个数字，重载到首页以后，后续页面访问率有 10%以上，这两个数字对业务提升其实蛮大的。其实现原理很简单，都是通过第二次触发 onShow 时进行处理。

4.2 H5 和小程序登录态同步问题

分两种情况，接入的 H5 可能一开始就需要登录，也可能开始不需要登录态中途需要登录，这两种情况我们约定了 h5 通过自己的 url 上一个参数进行控制。

webview内嵌H5登录



一开始就需要登录态的情况，具体来讲就是在加载 webview 之前，首先进行授权登录，然后把登录信息拼接到 url 里面，再去来加载 webview，在 h5 里面通过 adapter 来把登录信息提取出来并且存到 cookie 里，这样 h5 一进来就是有登录态的。

一开始不需要登录态的情况，一进入小程序就直接通过 webview 加载 h5，h5 调用 login 方法的时候，把 needLogin 这个参数拼接到 url 中，然后利用 api 进行重载，就走了第一种情况进行授权登录了。

Q：可能出现的登录同步问题

A: 跳到个人页登录完成，此时是新开的 webview 同步两端登录态，点返回，到上一个 webview，此时这个 webview 嵌套的首页，没有触发 react-imvc onshow 事件。这个页面是老的，退出登录也是一样，所以在首页会去跳 h5 的登录而不是小程序登录，导致登录不同步。

解决思路：需要返回首页刷一下 h5 页面。

误区：直接在个人登录之后，relaunch 到首页，会导致没有直接调用注销 webview 把 token 置换，无法退出。

解决方案：判断从个人页返回的时候，设置 webview 的 url 加个参数，重新刷一下。

4.3 Webviwe 分享

在没接入 websocket 之前，小程序主要通过 bind。首先通过 bindmessage 事件接收 h5 传回来的数据，然后在用户分享的时候 onShareAppMessage 判断有没有回传的数据，如果没有就到 webviewurl 当中取，否则就是用默认分享数据。

4.4 支付

1）webview 页面刷新问题

因为小程序 webview 里面不支持直接调起微信支付，所以基本上需要支付的时候，都需要来到小程序里面，支付完再回去。上面做好了以后，在 h5 这块调用一句话就可以了。

针对产品有大量内嵌 H5 页面的情况下，最好根据业务分两种支付页面，一是有的业务 h5 有自己完善的交易体系，下单动作在 h5 里面就可以完成，他们只需要小程序付款，因此我们有一个精简的支付页，进来直接就拉起微信支付。

还有一种情况是业务需要小程序提供完整的下单支付流程，通过直接进入小程序的收银台来，图上是 sdk 里面的基本逻辑，通过 payOnly 这个参数来决定进到哪个页面。再看下小程序里面精简支付怎么实现的，onload 之后直接调用 api 拉起微信支付，支付成功以后根据 h5 传回来的参数，如果是个小程序页面，直接跳转过去，否则就刷新上一个 webview 页面，然后返回回去。

新的问题与挑战：webview 返回上一页数据刷新问题

有客户反馈在 A 页面点击任务后跳转到 B 页面，待任务完成后，手机手势左滑返回或点击默认导航栏的左上角返回，上一个页面不会触发任务的更新。原因是上一个页面已经初始化并没有执行重渲染，在 APP 环境下 JSBridge 没有提供侦听手势左滑返回、左上角物理返回的回调事件，且在小程序 webview 页面也会遇到上述同样的情况。

由于微信并没有提供侦听手势左滑返回、左上角物理返回的，且 webview 页面也不支持自定义导航栏，这导致下一个页面触发的新事件，在返回上个页面时 无法做到针对性的更新。

前期可以简单粗暴地通过约定参数 doRefreshWhileBack=true 作为 options，来通过 webview 页面每次 onShow 刷新页面，但是刷新整个页面的成本太大，且用户体验不好。

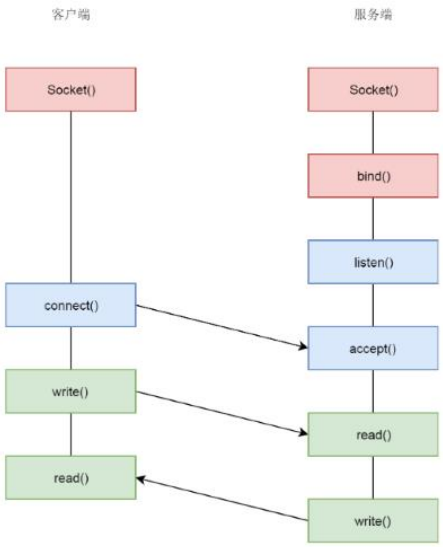
2）引入 websocket

带着这些疑问，我们进行一系列的尝试与试验，最终采用了 websocket 的方式，解决并封装出我们市场业务的轻量的 websocket 服务，主要用于解决 webview 跨页面通信和游戏方面的业务。

在这个过程中，我们总结出了一些经验，希望能给从事相关研究的同学带来一些帮助。上述做法是针对不同的应用环境，分别使用或约定不同的 api 派发给各自的事件系统，从而解决页面物理回退时页面不主动刷新的方案。

简要介绍一下 websocket，websocket 标准诞生于 2011 年，RFC 文档编号是 6455。TML 5 规范定义了 WebSocket 协议，它可以通过 HTTP 的端口（或者 HTTPS 的端口）来完成，从而最大程度上对 HTTP 协议通透的防火墙保持友好。但是，它是真正的双向、全双工协议，也就是说，客户端和服务端都可以主动发起请求，回复响应，而且两边的传输都互相独立。和上文的 Comet 不同，WebSocket 的服务端推送是完全可以由服务端独立、主动发起的，因此它是服务端的"真 Push"。

WebSocket 是一个可谓"科班出身"的二进制协议，也没有那么大的头部开销，这样就解决了接线员要反复解析 HTTP 协议，还要查看 identity info 的信息，因此它的传输效率更高。同时，和 HTTP 不一样的是，它是一个带有状态的协议，双方可以约定好一些状态，而不用在传输的过程中带来带去。而且，WebSocket 相比于 HTTP，它没有同源的限制，服务端的地址可以完全和源页面地址无关，即不会出现的浏览器"跨域问题"。



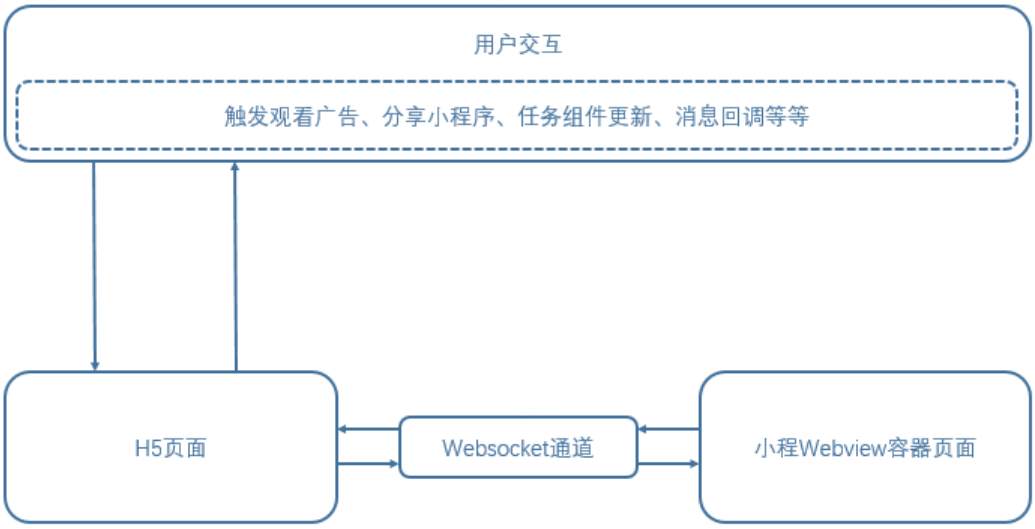
优势：

- 消息实时：真正的双向、全双工协议，完全的服务端推送保证了数据的时效性。
- 通信高效：可以由客户端和服务端主动发送请求，不会像轮询那样产生大量无效传输报文。
- 协议支持：标准诞生较早，浏览器支持度高，且没有同源策略的限制。

劣势：

- 开发与维护成本：服务器长期维护长连接需要一定的成本，且受网络限制比较大，需要处理好重连。

借助 websocket 的辅助，在小程序 webview 内嵌 H5 的业务场景中，可做的事情就更多了。在市场的 webview 容器加载流程中。



3）websocket 背景下的 webview 通信实践

小程序 webview 初始化并在 onLoad 阶段通过 options.useMktsocket 判断是否需要加载 socket，同时判断应用环境通过 wx.connectSocket api 连接不同的 socket 服务；

初始化 webview socket 服务，接受服务器消息-对服务器消息进行甄别，如果 H5 页面通过 socket 传递给 webview 容器的数据 data 格式符合预期，且 H5 环境下登录态中的 openId 与小程序环境一致，则认为此次通信合法；

webview 容器中绑定了 小程序分享 miniShare 、小程序订阅 openScribe、 健康检查 health 等常用业务 API，用于处理广告、订阅、任务更新等业务实时回调；H5 业务可通过此接口设置触发小程序原生页面的一些原生功能，为上层业务提供服务。

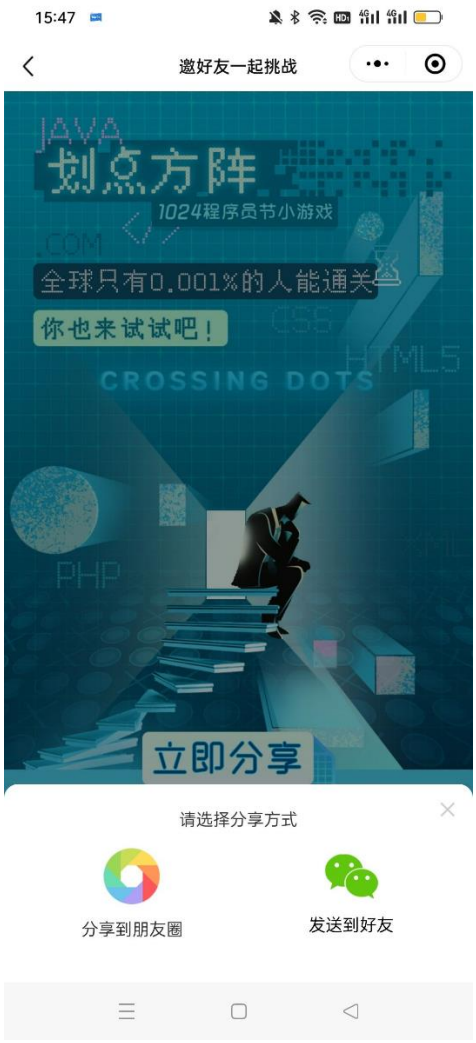
H5 页面就可以通过 socket 通信更改并调用小程序的胶囊栏分享、通知 webview 容器页面调用小程序广告、也可以调用唤起小程序页面中的分享组件面板、触发左上角物理返回时及时通知 H5 页面触发回调等诸多业务；同时小程序容器页面原生事件完成后（比如广告、分

享）再次通过 socket 返回给 H5 页面的回调，实现小程序 webview 跨页面的实时通信。

在 websocket 加持下，此时的小程序 webview 赋予了更多和 H5 通信的功能。

4.5 自定义分享面板

H5 页面可以通过 websocket 通信更改并调用小程序的分享参数，不再依赖于页面 options 参数，可以调用在 webview 页面封装的分享面板，提供更加灵活的分享方式。



4.6 H5 调用小程序原生的激励广告

H5 页面可以通过 websocket 通信调用小程序原生的激励广告。

4.7 任务体系中用户任务组件状态的更新

用户在访问加载了 webview-h5 的页面会与 websocket 的 server A 服务器连接、小程序原生页面与 server B 连接时，这两个页面因为在不同的容器下，所以无法通信和告知；但是只要这两个页面加载的是同一个市场的 websocket 服务，服务端可以设置共享一个 redis，通过 redis 的发布订阅功能，连通集群内部各个机器，那么在页面前进、回退时都可以绑定对应的回调事件，实现任务组件的灵活更新，给用户展示最新的任务状态。

五、总结

在处理小程序 webview 的业务方面，可以通过封装一个包含各端环境的 SDK，在 H5 初始化时加载，打通 H5 和小程序 webview 之间的通道，实现 H5 控制分享、登录态同步、支付信息同步等功能。

在遇到跨页面数据刷新问题时，借助了 websocket 这把利器，通过 redis 的发布订阅通知链接了 websocket 服务器的页面，实现小程序 webview 物理返回上一页而数据不刷新的问题，同时 websocket 使得 H5 与 webview 的通信更加便捷灵活，拓展了 H5 调用小程序原生激励广告、封装并调用小程序原生的分享面板等功能。

【参考文献】

《WebSockets 教程》，链接：https://www.tutorialspoint.com/websockets/

携程度假基于 RPC 和 TypeScript 的 BFF 设计与实践

【作者简介】工业聚，携程高级前端开发专家，react-lite, react-imvc, farrow, remesh 等开源项目作者，专注 GUI 开发、框架设计、工程化建设等领域。

一、前言

随着多终端的发展，前后端的数据交互的复杂性和多样性都在急剧增加。不同的终端，其屏幕尺寸和页面 UI 设计不一，对接口的数据需求也不尽相同。构建一套接口满足所有场景的传统方式，面对新的复杂性日益捉襟见肘。

在这个背景下，BFF 作为一种模式被提出。其全称是 Backend for frontend，即为前端服务的后端。它的特点是考虑了不同端的数据访问需求，并给予各端针对性的优化。

在这篇文章中，我们将介绍一种基于 RPC 和 TypeScript 的 BFF 设计与实践。我们称之为 RPC-BFF，它利用前后端都采用同一语言（TypeScript）的优势，实现了其它 BFF 技术方案所不具备的多项功能，可以显著提升前后端数据交互的性能、效率以及类型安全。

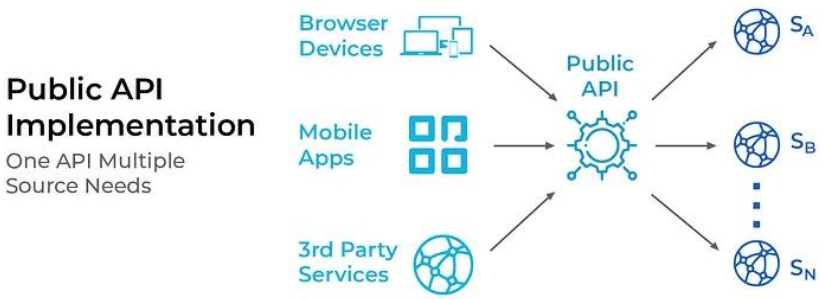
二、为什么会需要 BFF?

用发展的视角来看，业界存在的两大趋势催生了 BFF 模式：

- 硬件行业的多终端发展趋势
- 软件行业的微服务发展趋势

其中，微服务化以及中台化的趋势，改变了后端团队构建服务的方式。整个系统将按照领域模型分隔出多个微服务，这增强了各个服务的内聚性和可复用性的同时，也给下游的接口调用者增加了数据聚合的成本。

多终端的发展，又让数据聚合的需求进一步多样化。使得处于微服务和多终端之间的团队，不管是前端团队还是后端团队，他们面对的问题日趋复杂化。

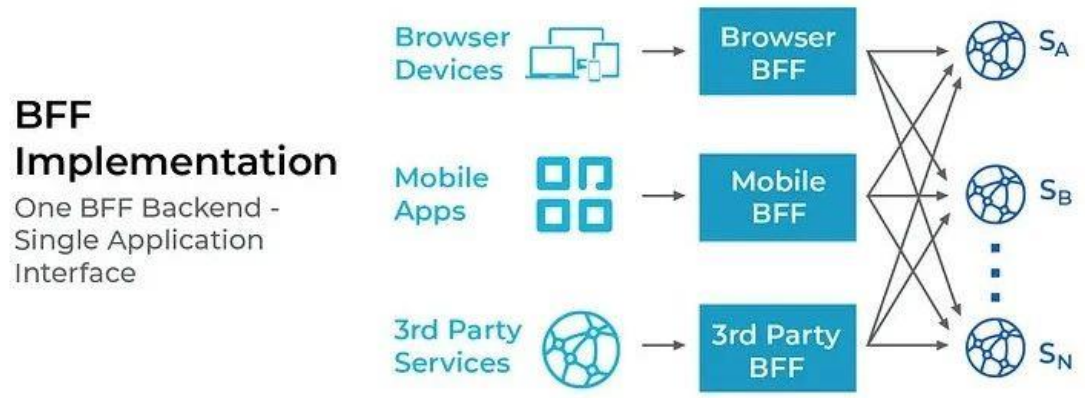


系统复杂度的增加，将体现在代码复杂度和团队协作复杂度上。这意味着，没有采用有效手

段应对复杂度的团队，自身将成为产研流程中的瓶颈。他们既要面对上游多个微服务的联调需求，又要应对下游多个端的数据消费需求；有更多的会议要参加，更多的需求文档要读，更多的代码、单元测试和技术文档要写，然而敏捷开发模式的交付周期却不随之增加。

此时一般有两个应对策略。一种是去掉数据聚合层，让各个下游前端应用自行对接微服务接口，将聚合数据的业务逻辑转移到前端，增加它们的代码体积，拖慢其加载速度，同时显著增加客户端向服务端发起的请求数，达到拉低页面渲染性能，破坏用户体验的效果。

另一种做法则是采用 BFF 模式，降低前后端数据交互的复杂度。微服务强调按领域模型分隔服务，BFF 则强调按终端类型分隔服务。将原本单个团队处理“多对多”的复杂关系，转变成多个团队处理的“一对多”关系。因此，本质上，BFF 模式的优化方式是通过调整开发团队在人力组织关系层面的职能分工而实现的。



也就是说，BFF 不是作为新技术用以提升生产力，而是通过改变生产关系去解放生产力。从单一团队成为产研流程中的单点瓶颈，转变成多个 BFF 研发团队各自应对某一端的数据聚合需求。

这种转变所争取到的是增加人手以提升效率的空间。一个工作任务相耦合的研发团队不能无限加人提效，但若能拆成多个研发团队，则可以扩大每个子团队的提效空间。当然，代价是团队之间可能存在重复工作，只是相比效率瓶颈而言，这些问题可能不属于现阶段主要矛盾，值得取舍。

尽管 BFF 不是新技术，也不要求新技术，但不意味着不能引入新技术，或者引入新技术无法提效。我们仍可以用新技术去实现 BFF，解决因团队拆分而产生的问题，收获研发效率（生产力）和产研流程（生产关系）两方面的提升。

接下来，我们先看一下实现 BFF 的几种不同方式，然后介绍作为新技术出现的 RPC-BFF。

三、BFF 的实现方式

如前所述，BFF 是作为模式（Pattern）被提出，而非一种新技术。在技术层面上，任何支持服务端开发的语言和运行时，不管是 Java、Python、Go 还是 Node.js，都可以开发 BFF 服

务。只要它们所实现的这个服务，是面向前端需求的。

BFF 的实现方式多种多样，不仅跟技术选型有关，跟研发团队的分工方式、职能边界、协作流程等因素也息息相关。

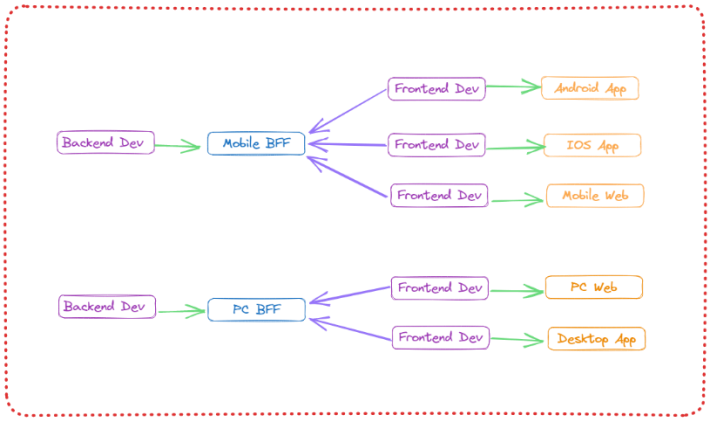
3.1 朴素模式

所谓的朴素模式，是指 BFF 的实现方式不改变前后端的分工方式、职能边界和协作流程。前端不介入 BFF 层的实现，BFF 层的需求由后端团队自行消化。

也就是说，BFF 在这里仅仅是后端团队的内部分工：

- 开发微服务应用（后端团队）
- 开发 BFF 服务（后端团队）
- 消费 BFF 接口（前端团队）

前端的工作方式跟之前一样，后端负责开发 BFF 为前端团队提供面向前端的数据聚合接口。开发 BFF 的编程语言由后端决定。



如上图所示，紫框为研发团队（前端或后端），蓝框为 BFF 服务，黄框为前端应用，绿色箭头表示“开发”，紫色箭头表示“调用”。

在图示中，BFF 按照终端尺寸分为 Mobile BFF 和 PC BFF 两类，它背后的假设是：相近终端尺寸的前端应用拥有相近的数据访问需求。移动端应用调用 Mobile BFF，PC 端应用调用 PC BFF。

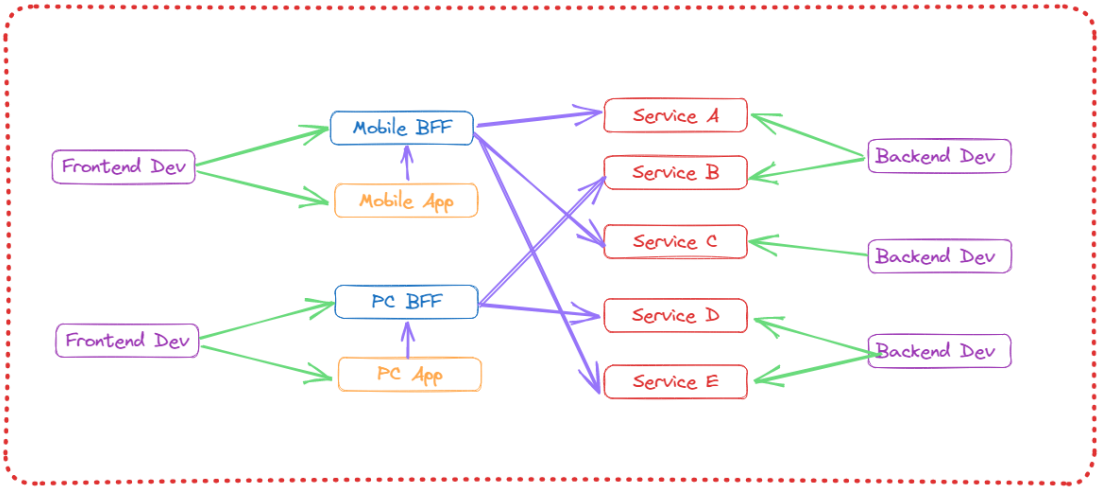
3.2 解耦模式

解耦模式，相比朴素模式而言，它改变了分工方式、职能边界和协作流程。后端不介入 BFF 层的实现，BFF 层的需求由前端团队自行消化。

也就是说，BFF 是前后端团队共同完成的一种新的分工：

- 开发微服务应用（后端团队）
- 开发 BFF 服务（前端团队）
- 消费 BFF 服务（前端团队）

前端的工作方式跟之前有所不同，前端团队自己开发 BFF 服务以供自己的前端应用消费聚合好的数据。开发 BFF 的编程语言由前端决定。



如上图所示，前端团队同时开发了 App 及配套的 BFF 服务。前端开发的 BFF 服务背后调用了后端团队提供的各个微服务接口，将它们聚合起来，转换为前端应用可直接消费的数据形态。

3.3 朴素模式 VS 解耦模式

两种模式都有其适用场景，具体要看不同研发团队的人力资源、应用类型和技术文化风格等多个因素。

从技术发展的角度，解耦模式更能代表“彻底的前后端分离”的趋势和目标。

前后端分离可以大体分为两个阶段：

- 渲染服务回归前端团队（SSR）
- 数据服务回归前端团队（BFF）

部分开发者可能认为完成了第一阶段，就达到了前后端分离的目标。此时前端团队可以自行构建 SSR 服务器，更早介入渲染流程，不必等到浏览器加载页面的 JavaScript 脚本后才开始发挥作用。不必再跟后端频繁沟通，交代他们在 html 文件里添加指定的 id 或 class 名；前端团队可以全权处理，后端团队只需要提供数据接口即可。

然而，“前后端分离”不只是把前端代码（如 html 文件）从后端代码仓库转移到前端代码仓库里，这只是形式和手段。前后端分离是指职能的分离，是为了让前端研发人员不必低效率

地遥控后端研发人员，让他们去机械地调整面向前端需求的代码。前端团队可以自主负责、自主修改、快速迭代，专业的人做专业的事儿。

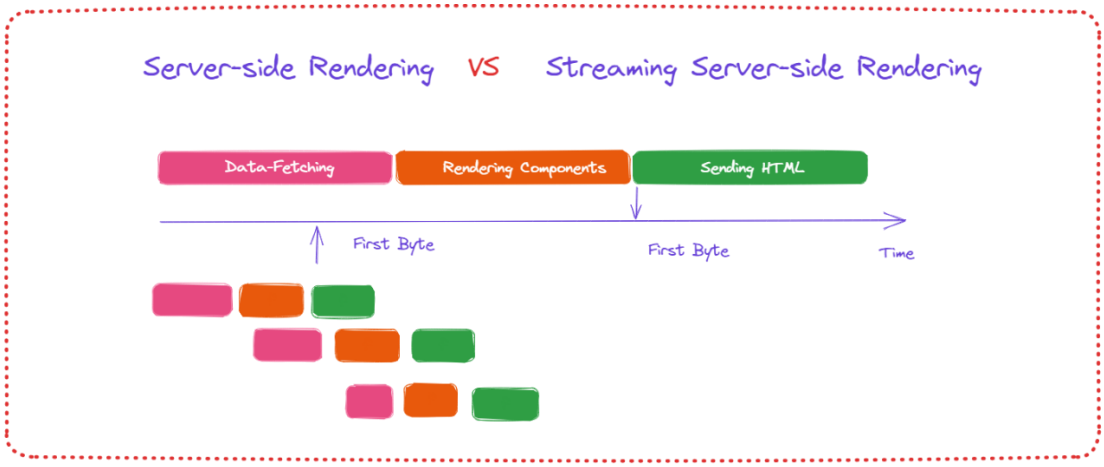
面向前端需求的代码，不仅仅是 html 文件，也包括了面向渲染优化的数据聚合代码。前端的职能是为用户开发出体验更好的 GUI 应用，有助于这个目标的所有合理的技术手段都在其职能范围之内。

让 SSR 服务从后端团队转到前端团队，是为了得到面向前端需求的界面渲染优化空间。让更加理解渲染优化原理的前端团队，可以在 SSR 服务上应用新的渲染优化措施，包括但不限于 Streaming SSR、Suspense、Selective Hydration、Server Component 等技术可以得以应用。它们可以显著提升页面的各项性能指标，令用户更早看到内容，更早看到有意义的内容，更早跟界面自由交互。

让 BFF 服务从后端团队转到前端团队，则是为了获得面向前端需求的数据聚合优化空间，可以提高代码的跨端复用率、减少前端应用的代码体积、减少前端应用的加载时间，提升用户体验。

当研发团队完成了 SSR 和 BFF 两个阶段的前后端分离后，前端团队同时掌握了 SSR 和 BFF 两重优化空间。他们既不需要让后端团队帮忙添加 id 和 class 到 html 文件中，也不需要让他们帮忙把某个文案添加到某个接口里。面向前端的渲染优化需求和数据优化需求，都能在前端团队职责范围内自主解决，显著减少前后端的沟通频次和成本，分离彼此的关注度，提升双方的专业聚焦水平。

更重要的是，SSR 和 BFF 在渲染优化上有着不可分割的关系。视图的渲染不是凭空的，它依赖数据的准备，有意义的内容才得以呈现；当视图需要流式渲染，数据请求也需要做相应配合。假设我们页面的首屏所依赖的数据，都被聚合到一个单一接口里，其后果便是 Streaming SSR 无法发挥充分价值；所有组件都在等待单一聚合接口的响应数据，才开始进行渲染。



如上图所示，紫色箭头表示“时间”，粉色框为数据获取，橙色框为组件渲染，绿色框为发送 HTML 到浏览器。我们可以看到，朴素的 SSR 是一个串行过程，组件渲染阶段需要等待数

据获取全部结束。而充分的 Streaming SSR 则有多次数据获取的并发任务（调用了多次后端接口），组件渲染并不需要等待所有数据获取任务结束，只需相关数据就位即可进入组件渲染及后续发送 HTML 的流程。用户可以更早看到内容。

在前端团队缺乏 BFF 掌控能力的情况下，他们无法自主控制 SSR 的数据获取过程，只能被动接受后端提供的接口。即便视图层框架（如 React）支持 Streaming rendering，也仅仅是把一大块 html 分多次发送给浏览器，它仍受制于 data-fetching 的阻塞时间，无法做到充分的 Streaming SSR。

因此，技术层面更合理的做法是，每个组件描述它自身所需的数据依赖，页面渲染时遇到没有数据依赖的静态组件，立即发送给用户，遇到有数据依赖的组件则发起请求（Render-As-You-Fetch），不同组件可以独立发起不同请求，每个组件数据请求完毕后即刻开始自身渲染，最终得到页面流式渲染，用户渐进式地看到一块块成形的界面渲染的效果。

我们可以看到，在这个渲染优化需求下，传统意义上的数据聚合思路（尽可能少的接口）反而是不利的；微服务式的多个接口，反而是有利的。当然，这不意味着前端直接对接微服务接口，不需要 BFF；这其实意味着我们需要 Streaming BFF 去解放 Streaming SSR 的潜力。

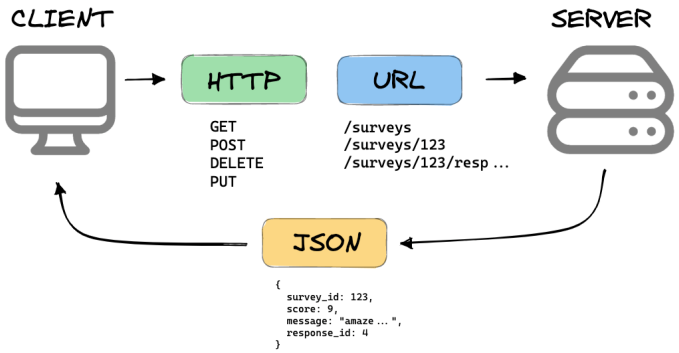
总的而言，彻底的前后端分离是指前端掌握了面向渲染优化的充分条件，包括 SSR 和 BFF 两个彼此紧密关联的优化空间，缺少任意一个，都难以获得充分的渲染优化能力。从这个角度来看，前端团队开发 BFF 是一个未来的技术方向。研发团队的分工模式随着技术发展，将从朴素模式逐渐转向解耦模式。

四、解耦模式的 BFF 技术选型

解耦模式的 BFF 服务由前端团队开发，所用的编程语言也由前端团队决定。大部分情况下前端团队会采用相同的编程语言（JavaScript/TypeScript），基于 Node.js 运行时开发相应的 BFF 服务。基于这个前提，我们讨论几种技术选型。

4.1 RESTful API

处于模仿阶段的前端团队，往往会采用 RESTful API 方式实现 BFF 服务。技术目标是把后端之前做但现在不做的功能，用同样的方式和思路让前端团队用 Node.js 实现一遍。



实现 BFF 服务的前端开发人员，其心智模型跟普通后端无异，涉及 URL,HttpRequest, HttpResponse, RequestHeader, Query, Body, Cookie, Authorization, CORS, Service, Controller 等等。

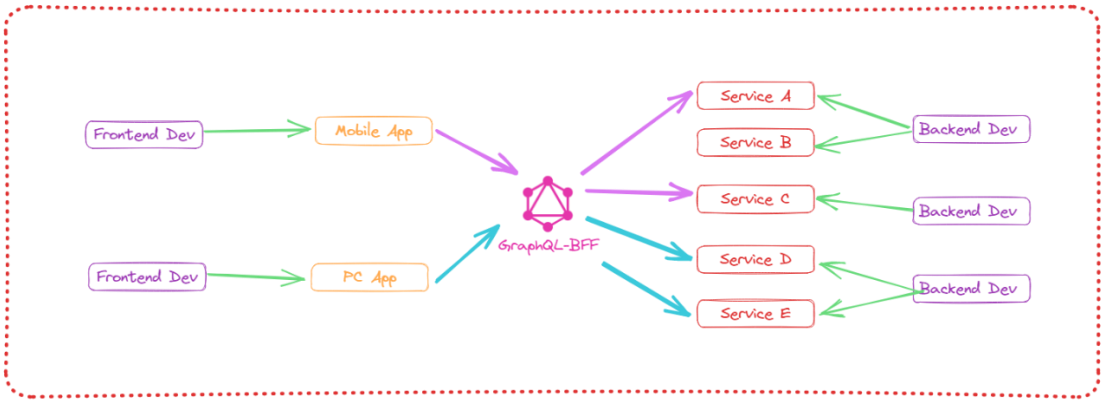
然而，即便是现在，掌握成熟后端接口开发能力的前端开发人员，依旧是稀缺的。所以，往往这种方式开发的 BFF 服务的质量，劣于专业后端开发的，并且几乎不考虑面向前端的极致优化，能满足需求已经达到了目标。

此外，从语义角度看，RESTful API 是面向资源的，跟面向前端需求的 BFF 场景并不契合。很多时候，前端需要的数据并不是后端资源的直接映射，而是经过聚合、转换、过滤、排序等处理后的结果。

因此，一开始基于 RESTful API 开发的 BFF 服务，最终将有意或无意、主动或被动地演变成只有一半功能的 RPC 服务。它的 url 参数设计是函数语义，而非资源语义，但调用这些远程函数时仍然要考虑 server-client 之间底层通讯细节。既没有被封装，也缺少优化。

4.2 GraphQL

GraphQL 是一个面向前端数据访问优化的数据抽象层，它相当适合作为 BFF 技术选型，并且也是我们之前包括现在仍在使用的 BFF 方案。



它的技术特点是，在数据访问层实现了一定的控制反转 (Inversion of control，IOC) 的能力。

GraphQL 服务的开发者负责构建一个数据网络结构（Graph），支持其消费者根据自身需求编写 GraphQL Query 语句查询所需 JSON 数据（Tree）。这种灵活的查询能力，实际上是将前后端数据交互相关的代码分成了两部分，一部分是关于通用性的，放在数据提供方（GraphQL 服务）里，一部分是关于特殊性的，放在数据消费方（前端应用的查询语句）里。

如此，GraphQL BFF 可以将按终端尺寸分类的多个 BFF 整合成一个 BFF。从之前 Mobile BFF 和 PC BFF，变成统一的 GraphQL BFF。减少了两个 BFF 之间重复的部分。

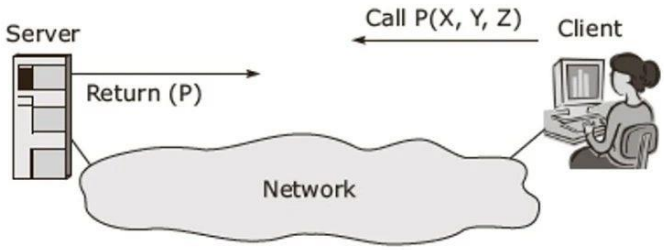
- Mobile BFF = GraphQL BFF + Mobile GraphQL Query
- PC BFF = GraphQL BFF + PC GraphQL Query

通过统一的 GraphQL-BFF 配合差异的 *-GraphQL-Query 实现了之前多个 BFF 服务提供的数据访问能力。

前端团队开发 GraphQL BFF 应用时，其心智模型不再是纯粹的后端概念，而是 GraphQL 相关的概念，如 Schema, Query, Mutation, Resolver，DataLoader, Directive, SelectionSet 等等，它们更加利于前后端数据访问的优化。更多详情可以阅读另一篇文章《GraphQL-BFF：微服务背景下的前后端数据交互方案》。

4.3 RPC

RPC 是指 Remote Procedure Call，即远程过程调用。这个模式也适用于 BFF 服务的实现。



它的技术特征是，将实现端和调用端之间的通讯过程封装，作为技术细节隐藏起来，并不暴露给调用者。对于调用者而言，仿佛像调用本地函数。

正因如此，它不要求调用一次 RPC 函数即发起一次独立的通讯过程。它可以将多次 RPC 函数批量化(Batching)并流式响应（Streaming），进一步减少反复重建通讯过程的开销。

前端团队基于 RPC 模式开发 BFF 时，其心智模型跟开发传统后端服务不同。所谓的接口设计，被转化为函数参数设计和返回值设计，沿袭了前端开发者熟悉的标准编程语言特性。相比 RESTful API 和 GraphQL 而言，RPC 引入更少概念和领域知识要求。

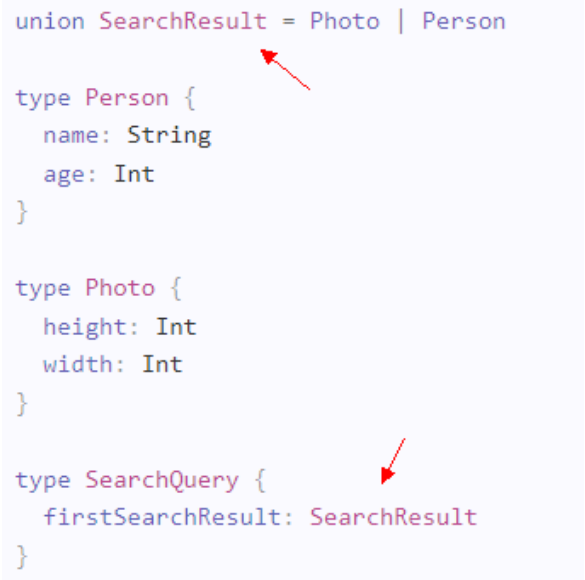
4.4 GraphQL VS RPC

我们团队从四年前开始使用 GraphQL-BFF，并成功落地到多个项目，取得了不错的效果。我们看到了它带来的好处和价值，同时也发现了它当前的一些局限性。为了突破桎梏，争取到更广阔的优化空间，我们开始探索 RPC-BFF 方案，并试图克服 GraphQL-BFF 方案未能解决的问题。

值得提醒的是，本文提到的 GraphQL-BFF 面临的难题，是在精益求精的层面上的探讨，并非否定和质疑 GraphQL 作为 BFF 方案的合理性。

GraphQL-BFF 的第一个难题是类型问题。GraphQL 是一个跨编程语言的数据抽象层，它自

带了一套 Type System。使用具体某个带类型的编程语言（如 TypeScript）开发 GraphQL 服务时，就存在两个类型系统，因此难免有一些语言特性无法对齐以及类型重复定义的地方。



比如，GraphQL 的类型系统支持代数数据类型（Algebraic data type），可以用 union 定义 A 或 B 的类型关系。这在 Rust 和 TypeScript 中有不错的支持，在 Java 或 Go 中还没有直接的支持。然而即便是 TypeScript，很多类型声明也得在 GraphQL 和 TypeScript 中分别定义一份。

此外，GraphQL 的 Client-side 类型问题也是一个挑战。由于 GraphQL 服务的返回值取决于发送过来的查询语句，因此其响应的数据类型不是固定的，而是随着查询语句代码的修改而变化的。

尽管 GraphQL 由于提供了内省机制（Introspection），可以构建出专门根据 GraphQL Schema + GraphQL Query 生成 TypeScript 类型定义的 Code Generator 工具。但其中包含很多手动处理或复杂的工程化配置环节。

开发者有可能需要手动从 TypeScript 代码里复制出 GraphQL Query 语句，在 GraphQL Code Generator 工具里生成 TypeScript 类型后，复制该类型定义到项目中，然后传入 GraphQL Client 调用函数标记其返回值类型。

或者像 Facebook/Meta 公司推出的 Relay 框架那样，实现一个 Compiler 去扫描代码里的 GraphQL Query 语句，自动生成类型到指定目录，让开发者可以直接使用。这块对研发团队的技术能力和工程化水平有较高要求。

GraphQL-BFF 的第二个难题是，缺乏 Streaming 优化支持。当前的 GraphQL 数据响应，是由查询语句中最慢的节点决定的，尚未支持已 resolved 的节点提前返回给调用端消费的能力。

虽然 GraphQL 有 @stream/@defer 相关的 RFC，但目前并未进入 GraphQL 规范中，也

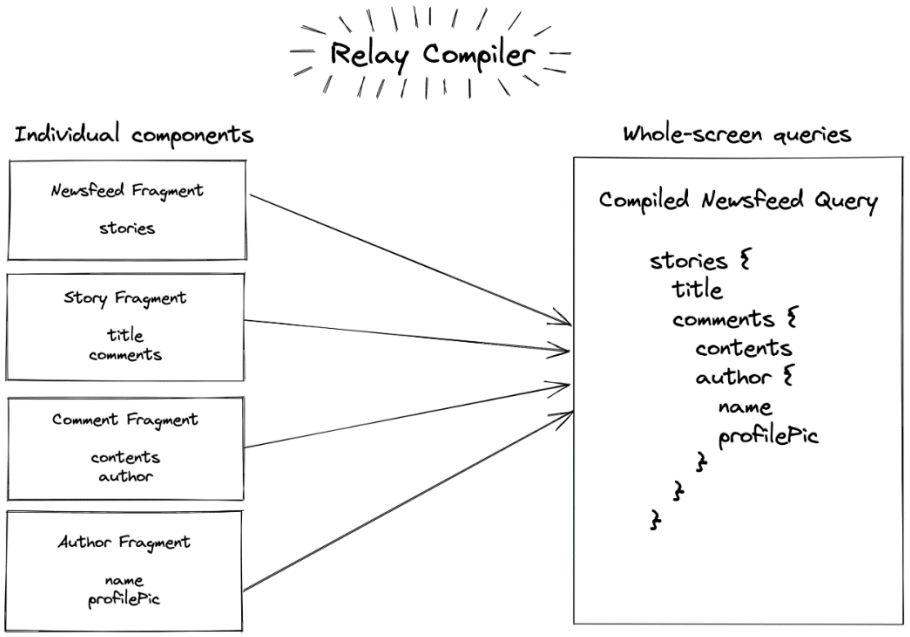
未在相关 GraphQL 库中得到实现或推广。

GraphQL-BFF 的第三个难题是缺少 Client-side Data-Loader 优化支持。

在 Server-side 的 GraphQL 有相关 Data-Loader，支持在一次查询请求处理过程中，相同资源的访问可以被去重。但是在调用端，一次 GraphQL Query 就对应着一次 Http 请求与响应。多次 GraphQL Query 很难被自动 merge 和 batching，遑论 streaming 优化。

如前所述，为了渲染上的进一步优化，前端组件实践的流行趋势是，每个组件可以将自己的数据需求定义在自己的组件代码中，而非托管给父级组件。如此，可以方便组件自身做 streaming 优化；当它自身的数据已经获取完毕，它可以先行渲染，不必等待。

相关 GraphQL Client（如 Relay）的处理办法是，让视图组件用 GraphQL Fragment 而非 GraphQL Query 去表达数据需求。通过编译器处理后，它们将可以提取到父级组件或者根组件里合并为 GraphQL Query。实现编写时在组件内，运行时托管在父级组件中获取数据。



这种策略在实践上是可行的，然而既有较高的工程化门槛，难以普遍推广，又不是 GraphQL 规范所定义的标准行为，甚至需要额外自定义很多指令以达到目标（如 Relay 框架的 @argument, @argumentDefinitions 等），这进一步损害了它的易用性。

我们需要的一种 BFF 技术方案是：

- 支持使用标准的语言特性解决问题
- BFF 实现端类型定义不必编写两份
- BFF 调用端可无缝复用 BFF 实现端的定义，不必重复定义
- 支持 Client-side Data-Loader 机制，可以将客户端的多个数据访问调用自动
  - merging

- batching
- streaming

RPC-BFF 技术方案，可以满足上述目标。

五、RPC-BFF 的技术选型

基于 RPC 实现 BFF 的思路和方案，也有很多种选择。

5.1 gRPC

gRPC 是一个非常优秀的 RPC 技术方案，但它跟 GraphQL 一样是跨编程语言的，需要额外使用一种 DSL 定义类型（Interface Definition Language，IDL），因此有类似的重复定义类型的问题，也未对前端常用语言（如 TypeScript)做充分的针对性优化，并且它主要服务于分布式系统这类 server-to-server 的调用，对 client/ui-to-server 的 BFF 场景没有特殊处理。

在很多基于 gRPC 的 BFF 实践中，BFF 跟背后的领域服务之间是 RPC 模式，BFF 跟前端之间则回到 RESTful API 模式。而我们所谓的 RPC-BFF，其实是指前端和 BFF 之间是 RPC 模式的调用关系。

5.2 tRPC

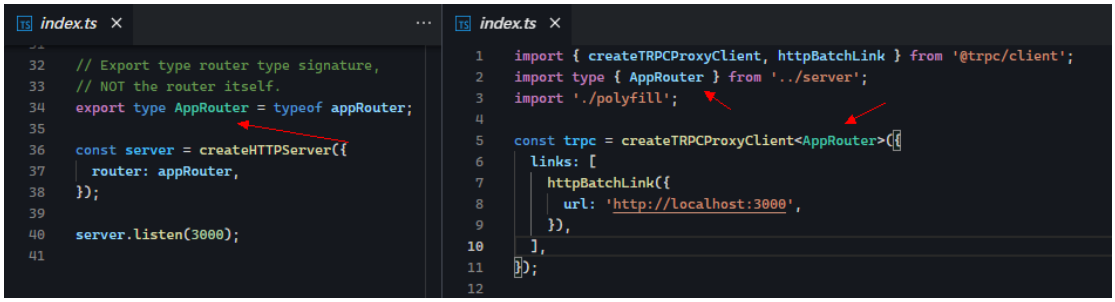
tRPC 的设计目的则跟我们的目标更加贴合，它是基于 TypeScript 实现的端到端类型安全方案（End-to-end type-safe APIs）。

然而，tRPC 的技术实现方式，跟我们的需求场景却不契合。

tRPC 跟前文的 Relay 框架某种意义上是两个极端。Relay 框架完全依赖它的 Relay Compiler 去扫描代码，充分分析和收集代码里的 GraphQL 配置，有很多编译、构建和代码生成的环节。而 tRPC 则相反，它目前没有构建、编译和代码生成的步骤，正如其官方文档里所言：tRPC has no build or compile steps, meaning no code generation, runtime bloat or build step

.

tRPC 假设了开发者的项目是全栈项目（full-stack application），或者前后端代码都在一个仓库。



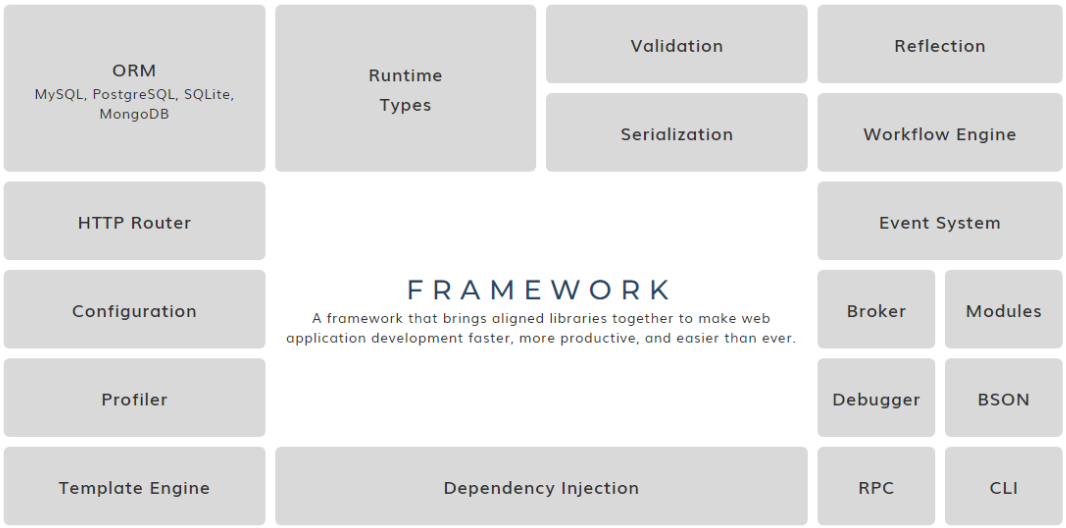
因此，前端项目可以直接 import 后端项目里的 TypeScript 类型。

如果我们构建的 BFF 项目，不只为了一个前端项目服务，而是多个前端项目共用的，涉及跨端、跨语言、跨团队等复杂组合。使用 tRPC 时，可能就得采取 Monorepo 模式，把 BFF 项目及其所有调用方的代码项目，放到一个 Git 仓库中管理。这将显著地增加多个团队之间的 Git 工作流、CI/CD、研发和部署流程等多方面的协调问题，要求处理新的工程化复杂度（Monorepo），能满足这个条件的团队不多。

我们的场景所期望的技术方案是，允许 BFF 项目及各个下游前端项目在不同仓库中管理，它们无法直接 import BFF 项目的类型，不满足 tRPC 的项目假设。

5.3 DeepKit

DeepKit 是一个富有野心的项目，它在 TypeScript 基础上增加了很多特性，力图打造一个更加完备的 TypeScript 后端基础设施。



它也有 RPC 模块的部分，但出于以下几个原因，最后未被我们选择。

第一个原因是，DeepKit 的 RPC 实践方式跟 tRPC 有类似的项目结构假设。

如上所示，rpc server 和 rpc client 之间需要有个共享的接口，一个负责实现该接口，一个负责消费该接口。这也要求前后端需要放到一个仓库中。或者采用 npm 包这类更低效的代码共享途径，对库和框架这种变动比较不频繁的场景来说是合适的，对业务迭代这种更新频率则难以接受。

另一个原因则是，选择 DeepKit 并不是选择一个库或者框架这种小决策，它从编程语言开始侵入，然后到运行时以及库和框架等方方面面。有较强的 Vendor lock-in 风险，一个项目要从 DeepKit 中迁移到另一个技术相当困难。

团队需要下很大的决心才敢押注 DeepKit 选型，以 DeepKit 目前的完成度和流行度，还无法支撑我们做出这个决定。

5.4 自研 RPC-BFF

如前所述，我们深入分析了 RPC BFF 的优势，以及考察了多个不同的技术选型。有的过于庞大、过分复杂，有的则过于简单、过于局限。

但这些项目也启发了我们的自研方向，帮助我们从简单到复杂的光谱中，根据自身实际需求找到一个平衡点，可以用尽可能低的研发成本、尽可能小的侵入性、尽可能少的项目结构要求，实现 RPC-BFF 模式。

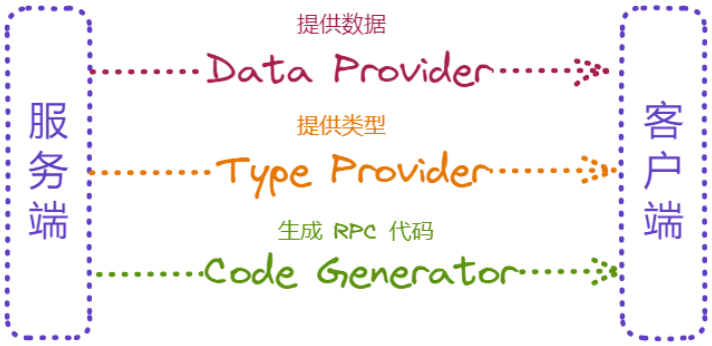
也就是说，我们既不要像 DeepKit 和 Relay 那样，从完整的编译器乃至编程语言层面切入到代码生成和运行时，它们或许有更大的目标和野心，配得上如此高昂的技术成本和实现难度，但对纯粹的 BFF 场景而言可能过犹不及。同时也不要像 tRPC 那样完全没有代码生成，而是选择一个最小化代码生成的路线，满足 RPC-BFF 这一聚焦场景的需求。

六、自研 RPC-BFF 的设计与实现

6.1 RPC-BFF 的设计思路

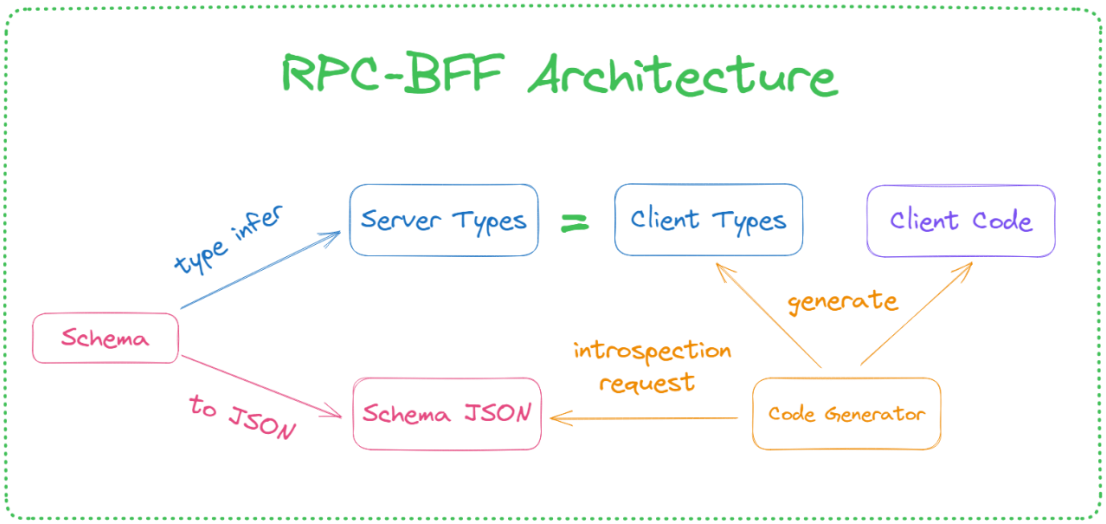
RPC-BFF 可以看作朴素 BFF 的拓展增强版。在朴素 BFF 中，后端在最简陋的情况下只为前端提供了数据。

而 RPC-BFF 则需要做更多：提供数据、提供类型以及提供代码。



如上图所示，RPC-BFF 既是数据提供者（Data Provider），也是类型提供者（Type Provider），还是代码提供者（Code Provider）。前端不必重复定义 BFF 响应的数据类型，也不必亲自构造 Http 请求处理通讯问题。

要做到这些功能，同时又不能损害易用性，需要极致地挖掘 TypeScript 的类型表达能力。



上图为 RPC-BFF 架构图示意，其中存在四种颜色，分别的含义如下：

- 服务端运行时（server-runtime）为粉色，服务器代码在此运行
- 类型编译时（compile-time）为蓝色，类型检查在此进行
- CLI 运行时（cli-runtime）为黄色，本地开发阶段使用的命令行工具
- 客户端运行时(client-runtime)为紫色，前端代码在此运行

该 RPC-BFF 架构设计的核心在于 Schema 部分，它是一切的基础。我们可以看到，Schema 有两条箭头，一条为 type infer，一条为 to JSON。也就是说，Schema 既作用于类型(type)所在的编译时(compile-time)，也作用于值(value)所在的运行时(runtime)。

当 BFF 端的代码经过编译，类型信息被编译器抹除后，我们仍可以在运行时访问到 JSON

数据结构表示的 Schema。

通过这种机制，我们的 RPC-BFF 像 GraphQL 那样支持内省特性（introspection）。在开发阶段，前端可以通过本地 CLI 工具向 RPC-BFF 发起内省请求（Introspection Request）拿到 JSON 形式的 Schema 结构，然后通过 Code Generator 生成前端所需的 Client 类型和 Client 代码。

如此，我们既不需要用编译器去扫描和分析服务端代码以提取类型，也不需要用编译器去扫描和分析前端代码去生成类型。对于 RPC-BFF 来说，不需要掌握到图灵完备的编程语言的所有信息才能工作，只需要掌握 RPC 函数列表及其输入和输入类型结构即可。

6.2 RPC-BFF Schema 设计与实现

Schema 是一段特殊的代码，它介于 Type 和 Program 之间，面向特定领域保留其所需的元数据性质的配置结构。Schema 往往比类型复杂，但比一般意义上的程序简单。

不管是 tRPC 还是 GraphQL 都包含 Schema 性质的要素。然而，对于 RPC-BFF 的场景来说，它们分别都有能力的缺失。

tRPC 中的 Schema 是只起到了 Validator 和 Type-infer 的作用，而没有 Introspection 机制。

```
// Our examples use Zod by default, but usage with other libraries is identical
import { z } from 'zod';

export const t = initTRPC.create();
const publicProcedure = t.procedure;

export const appRouter = t.router({
  hello: publicProcedure
    .input(
      z.object({
        name: z.string(),
      }),
    )
    .query((opts) => {
      const name = opts.input.name;
                 const name: string
      return {
        greeting: `Hello ${opts.input.name}`,
      };
    }),
});
```

如上所示，tRPC 自身没有实现 schema 部分，但可以从开源社区的多个 schema-based validator 库中选择一个它目前支持的（如 zod），从而得到在 server runtime 对客户端传递进来的参数验证，以及在开发阶段提供 type-infer 功能。

而 GraphQL 的情况则复杂一些，它分为 Schema-first 和 Code-first 两类实践方式。

```
var { graphql, buildSchema } = require("graphql")

// Construct a schema, using GraphQL schema language
var schema = buildSchema(`
  type Query {
    hello: String
  }
`)

// The rootValue provides a resolver function for each API endpoint
var rootValue = {
  hello: () => {
    return "Hello world!"
  },
}

// Run the GraphQL query '{ hello }' and print out the response
graphql({
  schema,
  source: "{ hello }",
  rootValue,
}).then(response => {
  console.log(response)
})
```

Schema-first GraphQL 实践如上图所示，GraphQL Schema 以字符串的形式出现在 TypeScript/JavaScript 代码中，它是 DSL 形态，要复用 host language（如 TypeScript）的类型系统相当困难。

```
// server.js using code-only graphql-js
const schema = new GraphQLSchema({
 query: new GraphQLObjectType({
   name: 'Query',
   fields: {
     hello: {
       type: GraphQLString,
       resolve: () => 'hello world'
     }
   }
 })
});
```

Code-first/Code-only GraphQL 则如上图所示，它是 eDSL 形态，即嵌入式领域特定语言(Embedded domain specific language)，它可以基于 host language 的 API 以程序的方式而非字符串的方式，创建出 GraphQL Schema。因此，这种模式下 GraphQL 相当于嵌入到 TypeScript 中，它有机会利用 TypeScript 的类型推导(type infer)能力，反推出 TypeScript 类型；也能够在运行时 stringify 为 DSL 形态的 GraphQL Schema。

可以说，这种实践方式的 GraphQL 跟 host language 的整合度更高，某种程度上是更灵活的，尽管牺牲了 DSL 那种直观性。

然而，目前几乎所有 Code-first/Code-only 的 GraphQL 库的 TypeScript 类型支持程度都有很大的不足。特别是对 GraphQL 这种类型之间递归结构特别频繁的技术来说，其类型推导的技术挑战远大于 zod 等朴素 schema-based validator 库。

即便是 zod 这类更简单的场景，对递归类型也没有充分支持。

You can define a recursive schema in Zod, but because of a limitation of TypeScript, their type can't be statically inferred. Instead you'll need to define the type definition manually, and provide it to Zod as a "type hint".

如 zod 官方文档所述，当我们的 schema 中存在递归，type-infer 就受到了限制，需要更繁琐的方式去自行拼装出递归类型。

```
const baseCategorySchema = z.object({
  name: z.string(),
});

type Category = z.infer<typeof baseCategorySchema> & {
   subcategories: Category[];
};

const categorySchema: z.ZodType<Category> = baseCategorySchema.extend({
   subcategories: z.lazy(() =>categorySchema.array()),
})
```

如上所示，它需要先用 schema 方式定义递归类型（Category）中非递归的部分，然后用 type-infer 在 type-level 定义递归部分的类型，最后回到 schema-level 中显式类型标注，定义递归 schema。

它在 schema-level 和 type-level 中来回穿梭。每一个递归字段都要求拆成上述 3 个部分，其工程上的易用性缺乏保障，其代码也缺乏可读性。有多少读者能轻易看出上面的复杂 schema 是为了定义下面这几行代码？

```
type Category = {
   name: string
      subcategories: Category[]
}
```

回到 GraphQL，我们现在能够看到，它在 Validator 和 Introspection 特性上有良好的支持，可以在 server runtime 验证参数结构和返回值，也可以通过内省请求曝露出 schema 结构，但在 type-infer 上仍有一些难以攻克的挑战存在。

RPC-BFF 的 Schema 需要同时满足 Validator，Type-infer 和 Introspection 三个能力，现有方案并不满足，因此我们通过自研 Schema 方案实现了它们。

下面是一段定义 User 类型的 TypeScript 代码：

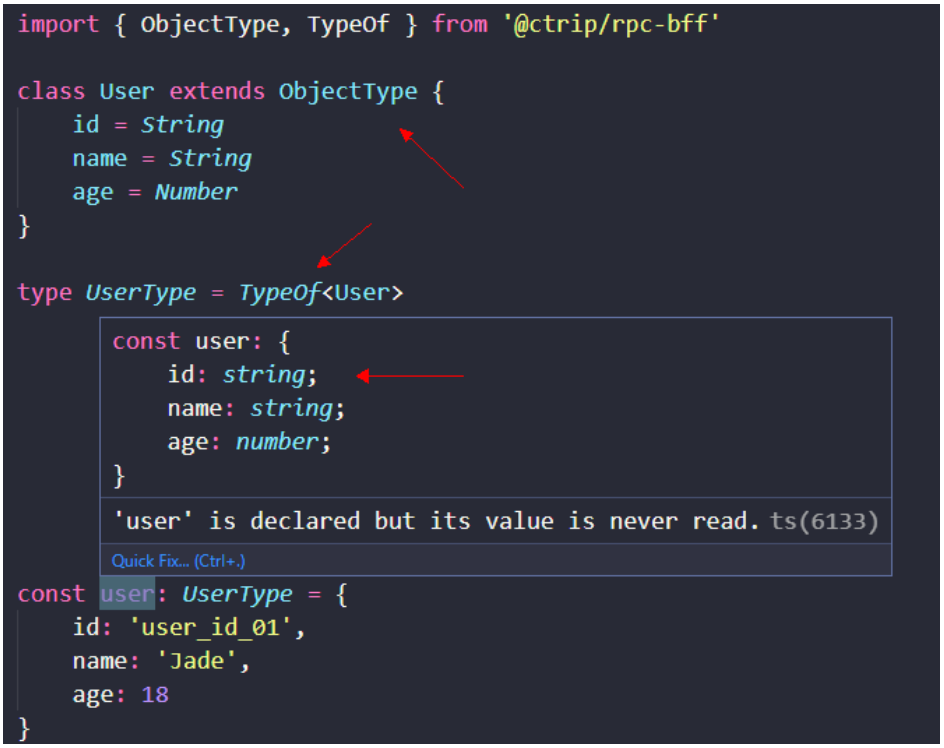
```
type UserType = {
        id: string;
```

```
    name: string;
    age: number;
}

const user: UserType = {
    id: 'user_id_01',
    name: 'Jade',
    age: 18
}
```

经过编译后，在运行时执行的 JavaScript 代码如下：

```
"use strict";
const user = {
  id: 'user_id_01',
    name: 'Jade',
    age: 18
};
```

类型信息被抹去，在运行时无法获取。通过 RPC-BFF Schema 重新定义 User 结构如下：



我们用 ObjectType 定义了 User Schema，用 TypeOf 推导出 UserType，经过 TypeScript 编译后，JavaScript 代码如下所示：

```
import { ObjectType } from '@ctrip/rpc-bff';
```

```
class User extends ObjectType {
  constructor() {
            super(...arguments);
            this.id = String;
            this.name = String;
            this.age = Number;
      }
}
const user = {
      id: 'user_id_01',
      name: 'Jade',
      age: 18
};
```

我们可以看到，由 ObjectType 定义的 User Schema 在运行时也得到了保留，因此我们可以基于这些信息，实现在运行时的 Validation 和 Introspection 功能。

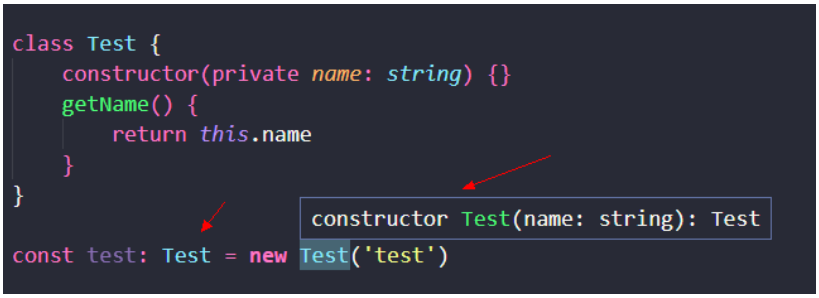
此外，我们的 RPC-BFF Schema 技术还克服了 zod 等库未能解决的递归类型定义问题：

我们无需为了支持递归而人为拆分类型，可以直观地定义出上述的 Category 结构，并支持静态类型推导。其秘诀在于利用 TypeScript 中 class 声明具备的独特性质——value & type

二象性。

当我们用 class 声明一个结构时，它同时也定义了：

- Constructor 函数值
- Instance Type 实例类型



如上图所示，左边箭头的 Test 是一个实例类型（Instance Type），右边箭头的则是类的构造器函数(Constructor)。

目前就我们所知，只在 class 声明场景下 TypeScript 对递归 Schema 有良好的类型推导支持。因此，如果 zod 或者 Code-first GraphQL 库想要支持递归 Schema 结构，它们的 Schema API 可能需要大改，变成我们上面演示的 RPC-BFF Schema 的 API 风格。

尽管我们掌握了在 Validator, Type-infer 和 Introspection 能力上更完备的 Schema 技术，满足了 Code-first GraphQL 的技术要求，但也仅限于 server 端的情况，在 client 端的类型，仍需要引入复杂的编译技术扫描前端代码库里的 GraphQL Query 片段以生成类型。这不是我们期望的。

因此，我们目前先将这种技术应用于更简单的 RPC-BFF 场景，未来也不排除支持 GraphQL。

6.3 定义 RPC-BFF 函数

有了 RPC-BFF Schema 之后，我们就可以用它来定义 RPC-BFF 函数了。

在纯 TypeScript 的代码里，定义函数的 input 和 output 是这样的：

```
// 定义 input
type HelloInput = {
  name: string
}

// 定义 output
type HelloOutput = {
  message: string
}
```

type HelloFunction = (input: HelloInput) => HelloOutput

然后实现满足该函数类型的代码：

```
const hello: HelloFunction = ({ name })=> {
  return {
    message: `Hello ${name}!`,
  }
}
```

在 RPC-BFF 里，我们只是换了一种方式去定义 input 和 output。

去掉注释，并且引入 ObjectType，前面的 hello 函数就变成了这样：

```
import { Api, ObjectType } from '@ctrip/rpc-bff'

class HelloInput extends ObjectType {
  name = String
}

class HelloOutput extends ObjectType {
  message = String
}

export const hello = Api(
  {
    input: HelloInput,
    output: HelloOutput,
  },
  async ({ name }) => {
    return {
      message: `Hello ${name}!`,
    }
  },
)
```

可以看到，它跟我们纯 TypeScript 的结构几乎是一样的。

6.4 RPC 函数和普通函数的区别

RPC 函数和普通函数的区别在于，RPC 函数的 input 和 output 都是 value，而普通函数的 input 和 output 都是 type。

type 会在编译时被擦除，而 value 会在运行时被保留。所以，RPC 函数的 input 和 output 需要是 value，它们在运行时是可以被访问到的。这样可以为 RPC-BFF client 生成类型代码和调用代码。

尽管我们使用 value 的方式去定义 RPC 函数的输入和输出，但通过 TypeScript 提供的 type infer 能力，我们不必为 RPC 函数的实现重新写一次类型定义，而是可以使用 TypeScript 的类型推导能力，让 TypeScript 自动推导出 RPC 函数的输入和输出类型。

```
export const hello = Api(
  {
    input: HelloInput,
    output: HelloOutput,
  },
                    function({ name }: {
                        name: string;
                    }): Promise<{
                        message: string;
                    }>
  async ({ name }) => {
    return {
      message: `Hello ${name}!`,
    }
  },
)
```

可以看到，我们的 hello 函数实现是有类型的。不仅如此，我们还可以通过 TypeOf 获取到 Schema API 定义出来的结构。

```
import { Api, ObjectType, TypeOf } from '@ctrip/rpc-bff'

class HelloInput extends ObjectType {
  name = String
}

class HelloOutput extends ObjectType {
  message = String
}

      type HelloInputType = {
          name: string;
      }
type HelloInputType = TypeOf<HelloInput>
```

通过这种方式，我们实现了 RPC 函数的输入和输出结构，具备以下能力：

- 可以在运行时保留，用以生成代码或者生成文档
- 可以在编译时被隐式地推导出来，用以做类型检查
- 可以通过 TypeOf 工具类型显式地获取到，用以做类型标记

现在，让我们把 RPC-BFF 函数放到 RPC-BFF App 中：

import { createApp } from '@ctrip/rpc-bff'

```
import { hello } from './api/hello'

export const app = createApp({
    entries: {
        hello,
    }
})
```

createApp 将创建一个 RPC-BFF App，其中 options.entries 字段就是我们想要曝露给前端调用的 RPC-BFF 函数列表。

启动后，一个 RPC-BFF Server 就运行起来了。

6.5 RPC-BFF 的 Client

在前端项目的开发阶段，它将会新增 rpc.config.js 配置脚本。

rpc.config.js

```
const { createRpcBffConfig } = require('@ctrip/rpc-bff-cli')

module.exports = createRpcBffConfig({
    client: {
        rootDir: './__generated__/',
        list: [
            {
                src: 'http://localhost:3001/rpc_bff',
                dist: 'my-bff-client.ts',
            }
        ]
    }
})
```

如上所示，当该脚本被 rpc-bff-cli 运行时，它会向 src 发起 introspection request 并生成代码到指定目录下的指定文件（如 my-bff-client.ts）。

就我们前面所演示的 hello 函数来说，其生成的代码大概如下所示：

./__generated__/my-bff-client.ts

```
/**
 * This file is auto generated by rpc-bff-client
 * Please do not modify this file manually
 */
```

```
import { createRpcBffLoader } from '@ctrip/rpc-bff-client'

export type JsonType =
  | number
  | string
  | boolean
  | null
  | undefined
  | JsonType[]
  | { toJSON(): string }
  | { [key: string]: JsonType }

/**
 * @label HelloInput
*/
export type HelloInput = {
  name: string
}

/**
 * @label HelloOutput
*/
export type HelloOutput = {
  message: string
}

export type ApiClientLoaderInput = {
  path: string[]
  input: JsonType
}

declare global {
  interface ApiClientLoaderOptions {}
}

export type ApiClientOptions = {
  loader: (input: ApiClientLoaderInput,
options?: ApiClientLoaderOptions) => Promise<JsonType>
}

export const createApiClient = (options: ApiClientOptions) => {
  return {
    hello: (input: HelloInput,
```

```
loaderOptions?: ApiClientLoaderOptions) => {
        return options.loader(
          {
            path: ['hello'],
            input: input as JsonType,
          },
          loaderOptions
        ) as Promise<HelloOutput>
      }
  }
}

export const loader = createRpcBffLoader("http://localhost:3001/rpc_bff")

export default createApiClient({ loader })
```

我们可以看到，RPC-BFF 的代码生成结果主要包含三个部分：

- RPC-BFF Server 里的 Schema 变成了前端里的 Type，将在编译后被擦除，不会增加前端代码体积
- RPC-BFF Server 里的 entries 变成了 createApiClient 函数，包含了跟 BFF 端对齐的函数调用列表及其类型信息
- RPC-BFF Client 被引入和实例化，它将在前端的运行时接管 RPC 函数的前后端通讯过程，对前端调用者无感

通过 Introspection + Code-generator 途径，一个 RPC-BFF 服务不必跟它的下游前端项目绑定，而是每个前端项目通过 rpc.config.js 各自同步它们所需的 RPC-BFF 服务。如此解耦了前后端的项目依赖，同时这个模式在 Monorepo 项目中也能很好地工作，是一种更加灵活的方式。

七、RPC-BFF 特性概览

至此，我们了解到了 RPC-BFF 的后端和前端分别的开发方式，可以看到对于 RPC-BFF 服务的开发者来说，并没有引入复杂的 API 或者概念，仅仅是在编写朴素函数的心智模型的基础上，将定义函数输入和输出结构的方式，从朴素的 Type 换成了 RPC-BFF Schema。

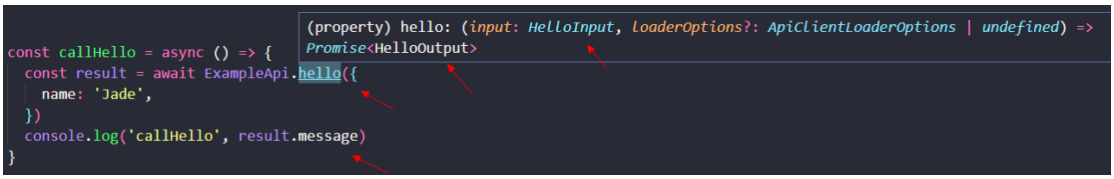
对于 RPC-BFF 服务的调用方而言，只是增加了 rpc.config.js 配置脚本，在开发阶段就能得到 RPC-BFF 的类型及其 Client 封装，用极小的成本获得极大便利。

但这仍不是 RPC-BFF 的优势的全部，接下来，我们来了解一下 RPC-BFF 的几大特性。

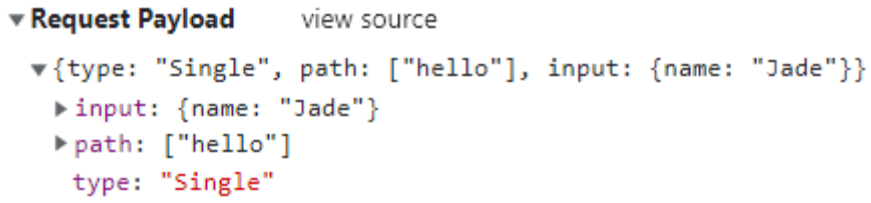
7.1 端到端类型安全的函数调用

端到端类型安全(End to end type-safety)的函数调用是 RPC-BFF 的基本功能，前端通过生

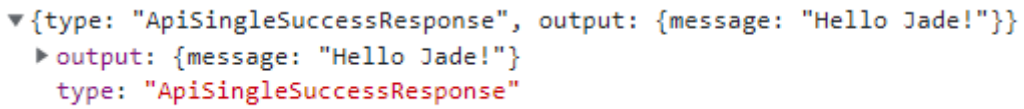
成的 RPC-BFF Client 模块访问 RPC-BFF Server 时，像调用本地异步函数一样。



如上所示，BFF 后端和前端的类型对齐。前端不必关心底层 HTTP 通讯细节，可以聚焦 RPC 函数的 input 和 output 结构。



当 RPC 函数执行时，它将发起 HTTP 请求，将所调用的 RPC 函数的路径(path)和输入(input)等信息打包发送给 RPC-BFF Server。



RPC-BFF Server 接受到 RPC 函数调用请求后，将匹配出指定函数并以 input 参数调用它得到其 output 结果后发送给前端，前端收到响应结果后，RPC-BFF Client 将其转换为前端 RPC 调用函数的返回值。整个 RPC 过程的前后端流程就完成了。

7.2 直观的错误处理

前面介绍的 RPC 调用只涉及请求正确处理和返回的情况，如果服务端报错了，前端如何处理呢？

对于这个问题，RPC-BFF 也有其优势。不同于朴素的接口请求错误处理，需要去判断 HTTP Status Code 检查请求状态码是否正确，甚至还得判断 result.code 检查业务状态码是否正确等等。

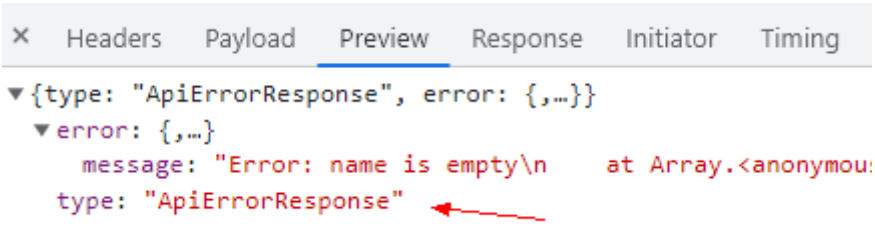
RPC-BFF 的错误处理支持最直观和自然的 throw 和 try-catch 特性。在 RPC-BFF Client 中可以 catch 到 RPC-BFF Server 里 throw 的错误。

```
export const hello = Api(
  {
    input: HelloInput,
    output: HelloOutput,
  },
  async ({ name }) => {

    if (name === '') {
      throw new Error('name is empty')
    }

    return {
      message: `Hello ${name}!`,
    }
  },
)
```

如上，改造我们的 hello 函数，当它遇到空的 name 参数时 throw 指定错误。

```
const callHello = async () => {
  try {
    const result = await ExampleApi.hello({
      name: '',
    })
    console.log('callHello', result.message)
  } catch (error) {
    console.log('callHello error', error)
  }
}
```

前端则构造一个空的 name 参数，并 try-catch 此次 RPC 调用，它将能捕获服务端抛出的错误。



再次执行后，从 Chrome Devtools 的 Network 面板中，我们看到了标记为错误的 RPC 响应结果。

```
callHello error Error: Error: name is empty                    App.tsx:15
    at Array.<anonymous>
app\dist\server\index.js:167:13)
```

在 Console 面板中，我们则看到了前端 catch 到的 RPC 调用错误日志输出。

7.3 代码即文档

RPC-BFF 从 GraphQL-BFF 中学习到了很多优秀之处。在 GraphQL 中，可以基于其 Schema 的 Introspection 能力，构建 GraphQL Playground 平台，可以在其中查看接口的参数类型、字段描述等信息，还能发起查询，相当方便。

RPC-BFF 的 Schema 也拥有 Introspection 能力，因此我们可以为前端提供更多内容。

```
export class Todo extends ObjectType {
  id = {
    description: `Todo id`,
    [Type]: Int,
  }

  content = {
    description: 'Todo content',
    [Type]: String,
  }

  completed = {
    description: 'Todo status',
    [Type]: Boolean,
  }
}

export class AddTodoInput extends ObjectType {
  content = {
    description: 'a content of todo for creating',
    [Type]: String,
  }
}

export class AddTodoOutput extends ObjectType {
  todos = {
    description: 'Todo list',
    [Type]: TodoList,
  }
}

export const addTodo = Api(
  {
    description: 'add todo',
    input: AddTodoInput,
    output: AddTodoOutput,
  },
  (input) => {
```

```
        state.todos.push({
            id: state.uid++,
            content: input.content,
            completed: false,
        })
        return {
            todos: state.todos,
        }
    },
)
```

如上所示，我们在定义 addTodo 接口的 input schema 和 output schema 时，不仅仅提供了对应的类型，还添加了相关的 description 描述。

经过前端的代码生成后，将得到如下代码：

```
/**
 * @label Todo
*/
export type Todo = {
    /**
    * @remarks Todo id
    */
    id: number,
    /**
    * @remarks Todo content
    */
    content: string,
    /**
    * @remarks Todo status
    */
    completed: boolean
}

/**
 * @label AddTodoInput
*/
export type AddTodoInput = {
    /**
    * @remarks a content of todo for creating
    */
    content: string
}
```

```
/**
 * @label AddTodoOutput
*/
export type AddTodoOutput = {
  /**
  * @remarks Todo list
  */
  todos: (Todo)[]
}

export const createApiClient = (options: ApiClientOptions) => {
  return {
    /**
    * @remarks add todo
    */
    addTodo: (input: AddTodoInput, loaderOptions?: ApiClientLoaderOptions) => {
      return options.loader(
        {
          path: ['addTodo'],
          input: input as JsonType,
        },
        loaderOptions
      ) as Promise<AddTodoOutput>
    },
  }
}
```

相比文本形式的接口契约文档，RPC-BFF 通过注释的方式将接口描述信息呈现出来。

如上所示，生成到注释的方式，比朴素的接口契约更贴近开发者，可以在代码编辑器里直观地看到接口描述和类型描述，并且基于开发阶段的同步机制，它总是实时反映当前 RPC-BFF 的最新状态，避免了接口文档过时的问题。

当我们想要废弃一个 RPC 函数时，这项机制尤为重要。

```ts
export const hello = Api(
  {
    deprecated: `use \`addTodo\` instead`,
    input: HelloInput,
    output: HelloOutput,
  },
  async ({ name }) => {
    if (name === '') {
      throw new Error('name is empty')
    }

    return {
      message: `Hello ${name}!`,
    }
  },
)
```

如上，我们通过添加 RPC 函数的 deprecated 描述，宣布废弃。

前端经过代码生成同步到 RPC-BFF 最新状态时，将能在代码编辑器里直观地看到废弃的提示信息。如此可以实现流畅的前后端接口废弃过程。

7.4 自由的函数组合

在 RPC-BFF 中，RPC 函数跟普通函数本质上是一样的，只是它通过 Schema 定义额外携带了描述自身的元数据信息。我们可以像组合普通函数一样，组合 RPC 函数。

假设我们有 updateTodo 和 removeTodo 两个 RPC 函数，然后我们希望添加一个功能：当 updateTodo 收到的 todo.content 为空时，则 remove 该 todo。

那么，我们不必把 removeTodo 功能分别在 updateTodo 和 removeTodo 中各自实现一遍，而是在 updateTodo 中根据条件调用 removeTodo。

如上所示，在 updateTodo 中调用 removeTodo 并没有特殊的要求，就像调用别的异步函数一样简单，并且对于前端发起的 updateTodo RPC 调用也没有额外开销，仍是一次对 updateTodo 的远程函数调用。

7.5 可靠的接口兼容性识别与版本跟踪

由于 RPC-BFF 的调用方在自己的前端项目中，通过 rpc.config.js 同步了 RPC-BFF 当前的接口的类型，因此它还解决了前后端之间常常遇到的接口兼容性争议。

在以往朴素的实践中，接口兼容往往只是后端对前端的口头承诺，缺乏自动化的、系统化的方式去发现和识别接口兼容性，甚至往往将问题暴露在生产环境。然后前后端开始争执判空职责的前后端边界划分问题。

而现在，RPC-BFF 提供了自动发现接口兼容性机制，并且是以系统性的、无争议的方式实现的。

```
export class AddTodoOutput extends ObjectType {
  todos = {
    description: 'Todo list',
    [Type]: Nullable(TodoList),
  }
}
```

如上，当后端的 addTodo 接口改变了返回值类型，从非空的 todos 变成可空的 todos，这是一种不兼容的变更。

前端同步 RPC-BFF 的接口契约后，在代码编辑器里立即可以看到类型系统的 type-check 结果。

```
const addTodo = async (content: string) => {
  const result = await ExampleApi.addTodo({
    content,
  })
  return result.todos.filter(todo => {
    return todo.content !== ''
  })
}
```

```
(property) todos?: Todo[] | null | undefined
@remarks — Todo list
Object is possibly 'null' or 'undefined'. ts(2533)
View Problem (Ctrl+K N)   No quick fixes available
```

在不解决这个类型问题的情况下，前端项目难以通过编译，从而避免将问题泄露到生产环境。

```
/**
 * @label AddTodoOutput
 */
export type AddTodoOutput = {
  /**
   * @remarks Todo list
   */
-  todos: (Todo)[]
+  todos?: (Todo)[] | null | undefined
}
```

此外，RPC-BFF 生成的代码也进入了前端项目的版本管理中，可以从 git diff 中，清晰地看到每一次迭代的接口契约变更记录。更完整无误地追溯和跟踪前后端的接口契约历史。

7.6 自动 merge & batch & stream 优化

如前文所描述的，RPC-BFF 需要支持自动的 merge & batch & stream 功能，以便达到更少的 HTTP 请求、更快的数据响应以及更优的 SSR 支持。

我们先来讲解一下它们分别的含义。

首先讲 batch，它是一种常见的优化技术，可以把一组数据请求合并为一次，往往用在相同资源的批量化请求上。这需要服务端接口提供支持，比如 getUser 接口是获取单个用户信息的，而 getUsers 则是获取多个的，后者可以被视为 batch 接口。

而 merge 在这里则偏向前端概念，接口支持 batch 只是说我们可以调用一个 batch 接口获取更多数据，但不意味着前端代码里多个地方调用 getUser 接口，会自动 merge 到一起去调用 getUsers 接口。

merge & batch 结合起来，就可以让前端里分散的各个调用自动合并到一次 batch 接口的请求中。

现在我们来看 stream，它跟 batch 一样需要服务端的支持。一次 batch 接口请求的响应时间，往往取决于最慢的数据，因为服务端需要准备好所有数据后才能返回 JSON 结果；所

谓的 stream 支持，则是服务端能够提前将已获得的数据一份份发送给前端。

某种意义上，stream 也需要前端的支持。假设接口支持 stream 可以一批批返回数据，前端却一个 await 等待所有数据就位后才开始下一步，那么等于没有发挥出接口流式响应的优势。

因此，merge & batch & stream 三者的结合才能发挥出更充分的优化效果。

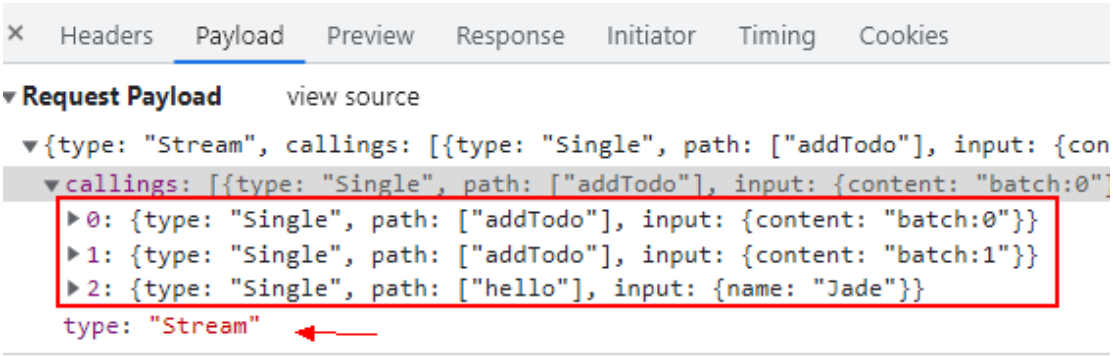
- 通过 merge，前端代码里各个地方的接口调用被 batch 起来
- 通过 batch & stream 接口，一份份数据从服务端发送给前端
- 通过 RPC-BFF Client 内部数据分发，指定的数据被一份份地发送到各个前端调用点

每个前端调用点收到数据响应后都将第一时间进入后续的渲染流程，不会受到其它调用点的阻塞影响。

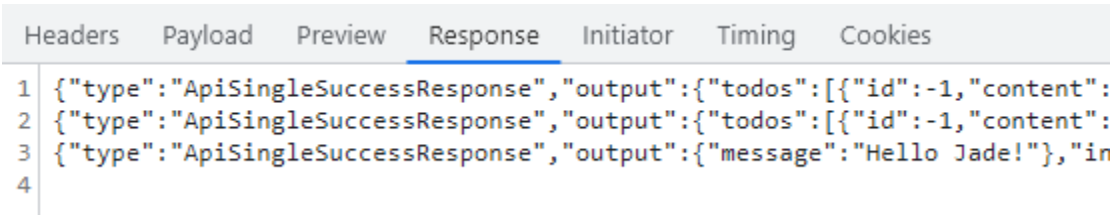
我们来看一个例子：

```
const tryBatch = async () => {
  const result = await Promise.all([
    ExampleApi.addTodo({
      content: 'batch:0',
    }),
    ExampleApi.addTodo({
      content: 'batch:1',
    }),
    ExampleApi.hello({
      name: 'Jade',
    })
  ])
                    const result: [AddTodoOutput, AddTodoOutput, HelloOutput]
  console.log('result', result)
}
```

如上，为了演示方便，我用 Promise.all 将 3 次 RPC 函数调用的结果打包到一起返回，但我们仍需知道，它们其实是三个独立的 RPC 调用，被写到一起还是分散在其它地方调用，不影响结果。Promise.all 只是在前端汇总 promises 结果，不包含接口相关的 merge & batch & stream 等作用。



从 Network Request 面板中，我们看到了一个被标记为 Stream 的请求，它里面包含了上面 3 个 RPC 调用的所有信息。



在 Network Response 面板中，我们则看到了一个 Newline delimited JSON 响应，即用换行符分隔的 streaming JSON 格式。每一个被 batch & stream 起来的 RPC 函数调用返回数据后，都将立即产生一条 JSON 结果发送给前端，每一行对应一次 RPC 调用。

7.7 自动缓存和去重

除了 merge & batch & stream 以外，自动的缓存和去重(cache & dedup)对于渲染优化也很重要。

在前端界面中，两个组件依赖同一份接口数据的情况很常见。传统方式是，手动去重，即两个组件都不包含接口调用，而是 lift up 到公共的祖先级组件统一处理后通过 props/context 等方式将同一份数据传送给这两个组件。

这种方式的缺点是:

- 两个组件都无法被独立使用和复用，因为它们的数据请求逻辑都被挪出去了，内聚性被打破
- 组件优化不足。只有枝叶组件各自发起请求时，Streaming 渲染得到了更优条件。越是顶层的组件里悬停，子组件渲染越是受到阻塞

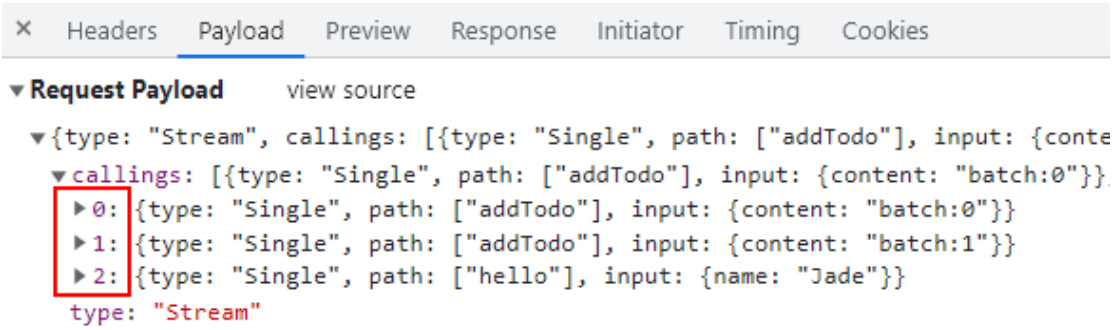
这就是为什么 React 目前致力于接管 data-fetching 层，让开发者手动去管理缺少将系统性优化。

对于 RPC-BFF 而言，支持 cache & dedup 变得重要。

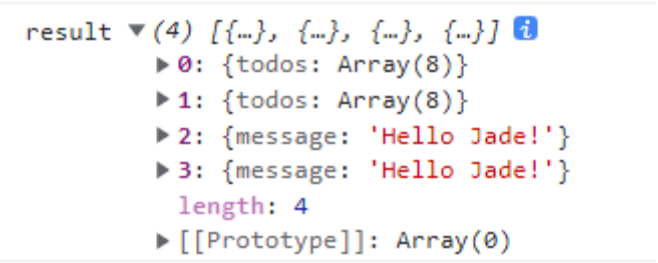
```
const tryBatch = async () => {
  const result = await Promise.all([
    ExampleApi.addTodo({
      content: 'batch:0',
    }),
    ExampleApi.addTodo({
      content: 'batch:1',
    }),
    ExampleApi.hello({
      name: 'Jade',
    }),
    ExampleApi.hello({
      name: 'Jade',
    })
  ])

  console.log('result', result)
}
```

```
const result: [AddTodoOutput, AddTodoOutput, HelloOutput, HelloOutput]
```

改造前面的 tryBatch 代码，使其包含 4 次 RPC 调用，有 4 个返回值，但有 2 次的参数和函数名是重复的。



但我们的请求却只包含了 3 次 RPC 调用及其响应，其中 2 次的重复调用被合并为一次。



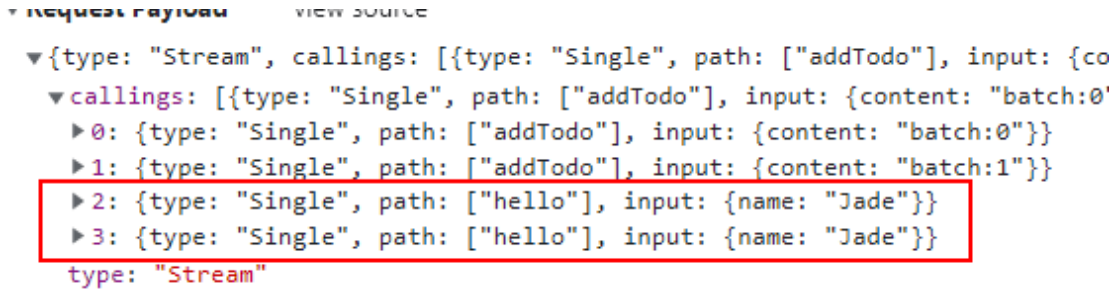
但并不影响每个 RPC 调用拿到它对应的 4 个调用结果。

通过这种 cache & dedup 机制，每个组件的数据请求都可以内聚于组件代码内部，而不必被迫 lift up 到父级组件做请求托管了。

值得提醒的是，不是所有相同参数的 RPC 调用都能被缓存和去重。特别是对于 mutation 性质的请求来说，连续调用两次相同参数的 createTodo，应当是创建两条而非一条。

```
const tryBatch = async () => {
  const result = await Promise.all([
    ExampleApi.addTodo({
      content: 'batch:0',
    }),
    ExampleApi.addTodo({
      content: 'batch:1',
    }),
    ExampleApi.hello({
      name: 'Jade',
    }, {
      cache: false
    }),
    ExampleApi.hello({
      name: 'Jade',
    })
  ])

  console.log('result', result)
}
```

因此，RPC-BFF 支持在前端传递第二个 options 参数给 RPC 函数，可以关闭 cache 特性。

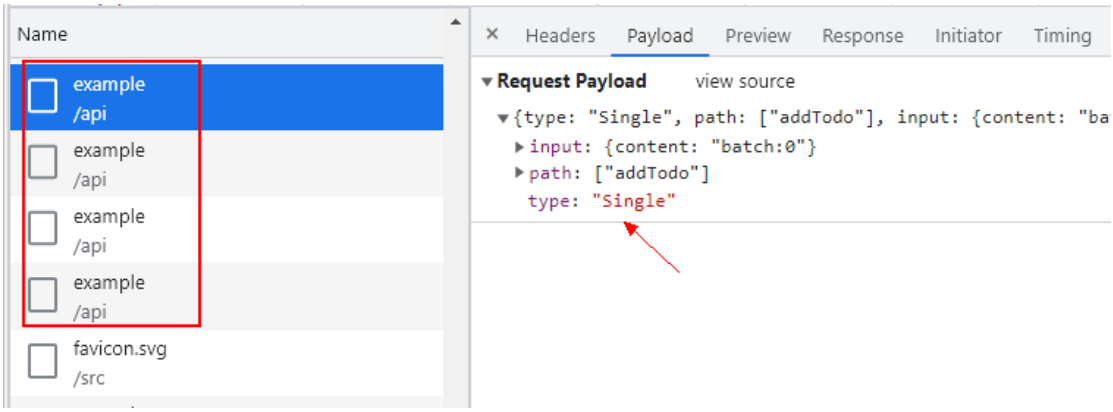


如上图所示，关闭 cache 后，即便是相同的 RPC 调用，也不会被缓存和去重。

options 选项，不仅可以关闭 cache，还可以关闭 batch, stream 等特性。

```
const tryBatch = async () => {
  const result = await Promise.all([
    ExampleApi.addTodo({
      content: 'batch:0',
    }, {
      batch: false
    }),
    ExampleApi.addTodo({
      content: 'batch:1',
    }, {
      batch: false
    }),
    ExampleApi.hello({
      name: 'Jade',
    }, {
      batch: false
    }),
    ExampleApi.hello({
      name: 'Jade',
    }, {
      batch: false
    })
  ])

  console.log('result', result)
```

如上所示，options.batch 是 cache 和 stream 的前提，我们将所有 RPC 调用的 batch 选项都关闭。



其结果是所有 RPC 函数调用退化为朴素形态，每一个 RPC 调用对应独立的一次 HTTP 请求。

通过灵活的选项配置，我们可以按需决定 RPC-BFF 里的函数调用的优化策略。

八、总结

在这篇文章中，我们介绍了 BFF 的起源、模式和技术选型，并根据界面渲染优化需求，了解到 Streaming BFF 的重要性，同时也给出了我们当前探索的技术方向——RPC-BFF。

我们对比了开源社区的一些流行方案，并根据我们自身的场景做了分析，尽管最终采用了自研的方式，但在调研和实验开源技术的过程中，也让我们学习到了很多知识，使得 RPC-BFF

的设计和实现能够吸收开源社区的技术成果。

我们最终达到了预期的技术目标，实现了：

- Validation: BFF 端可以在运行时验证参数结构和返回值结构的合法性
- Type-infer: 支持从 Schema 中静态类型推导出 TypeScript 类型，避免重复定义
- Introspection: RPC-BFF 服务可以通过内省请求曝露出其函数列表的契约结构
- Code-generation: 支持为调用方生成类型代码和调用代码，解耦项目依赖
- Merge: 前端多次 RPC 调用可以自动合并为一次 HTTP 请求
- Batch: 一次 HTTP 请求支持包含多个 RPC 调用
- Stream: 多个 RPC 调用可以流式响应，一份份发给前端，避免阻塞
- Dedup: 重复的 RPC 调用可以被去重合并
- Cache: 重复的 RPC 调用可复用缓存结果
- Type-safe: 前端可复用和对齐 BFF 端的类型
- Code as documentation: 代码即文档，RPC 接口的文档描述通过代码生成，以代码注释的形态直接作用于前端项目中

目前我们的 RPC-BFF 技术方案已经在内部试点项目中落地并上线平稳运行，接下来将会推广和迭代，并持续挖掘 RPC-BFF 技术方向上的优化潜力。

以上，希望能给大家带来帮助。

携程 Taro 多端化探索与实践

【作者简介】Frank，携程前端研发，专注前端性能优化、一码多端、工程化建设等领域。

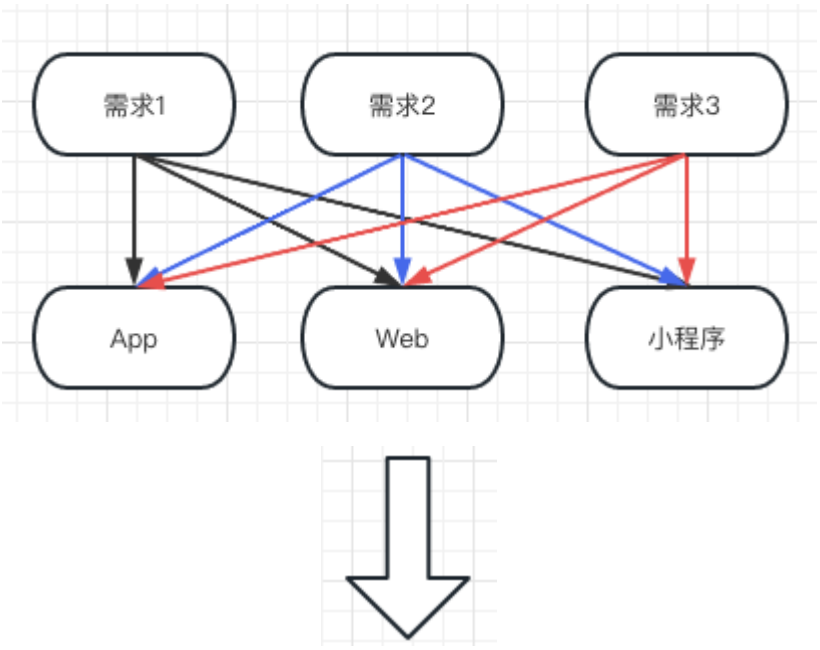
一、业务背景

随着移动互联网和智能设备的普及，前端开发人员需要采用多端同构技术来适配不同的终端（小程序、App 和 Web）。这些终端之间存在着明显的差异，包括浏览器引擎、操作系统、交互方式以及代码语言等方面。

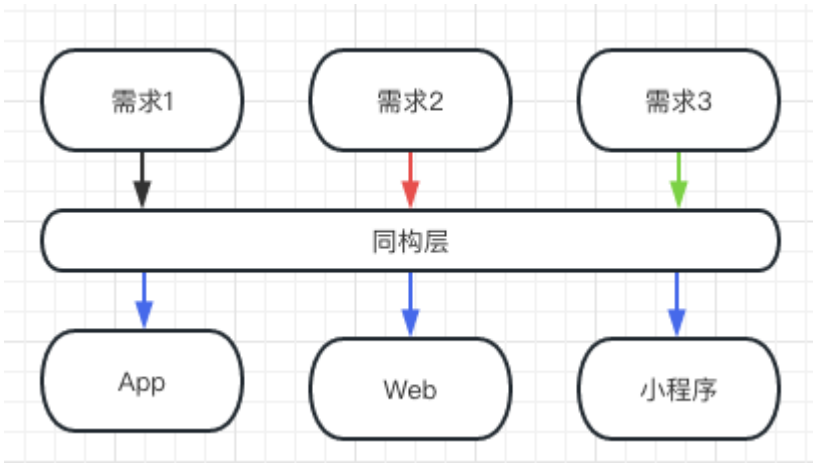
这些差异给前端开发人员带来了不少挑战。一方面，不同终端采用不同的浏览器引擎和操作系统，导致页面渲染和交互行为的表现各不相同。另一方面，不同终端所使用的代码语言和开发工具也存在差异，需要开发人员具备不同的技术背景和知识，才能编写多份代码来适配不同的终端。这样做不仅增加了研发人员的开发工作量和代码维护的难度，还可能导致用户在不同设备上遇到不一致的用户体验，影响产品的质量和用户满意度。

为了解决这些问题，多端同构技术应运而生。通过多端同构技术，旅游前端和公共团队合作多端探索与实践，根据不同终端的特性进行灵活的适配和定制。这样可以减少开发成本和维护难度，提高开发效率和代码的可复用性。同时，多端同构技术还能提供一致的用户体验，无论用户使用哪种设备访问应用程序，都能获得相似的界面和功能。

行业现状



三端同构



二、多端同构技术选型

在进行多端同构技术选型时，我们需要综合考虑跨端能力、成本、性能、代码语言通用性以及现有技术的支持度。这将有助于我们选择最适合的技术方案，以下是对当前前端主流跨端技术的分析：

	Hybrid	React Native	Flutter	Weex	Taro
跨端能力	★★★★	★★	★★★	★★★	★★★★
成本	★★★★	★★	★★	★★	★★★
性能	★	★★★	★★★★	★★★	★★★
代码语言通用性	★★★★	★★★★	★★	★★★	★★★★
携程支持度	★★★	★★★★	★★★★	★	★★★★

Hybrid：使用 JavaScript 语言，支持快速构建多端应用。由于依赖于 Webview 容器来运行，所以其用户体验和性能受到一定的限制的。这种限制会导致应用的响应速度变慢，页面加载时间变长等问题。适用于三端业务述求比较高，研发成本又比较低，性能要求不高场景，比如营销广告页。

React Native：使用 JavaScript 语言开发的 React 的组件，支持构建 App、Web，不支持原生小程序。App 上有接近原生应用的性能和用户体验。适用于对小程序性能要求不高的场景。

Flutter：使用 Dart 语言和自带的渲染引擎，支持范围同 ReactNative。在渲染速度和用户体验方面表现比 ReactNative 更加出色。由于 ios 平台规则限制，目前对于热更新支持并不友好。适用于对 App 性能要求较高，小程序性能要求不高的场景。

Weex：使用 JavaScript 语言开发的 Vue 的组件，支持范围与性能同 ReactNative，社区活跃度不如 ReactNative。

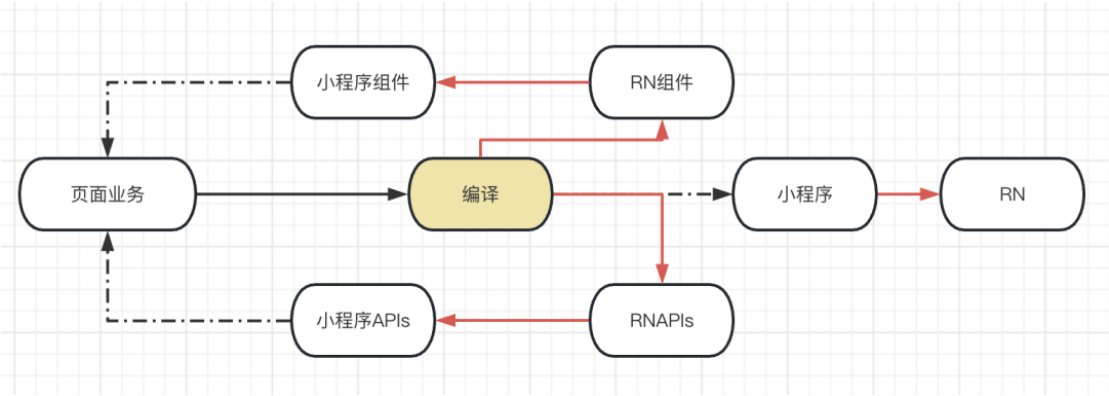
Taro：开放式跨端跨框架解决方案，它提供了一套统一的开发语法和组件规范，使开发人员能够使用一套代码来开发适配不同平台的原生应用程序。适用于对于三端述求高，性能要求也高的场景。由于设计之初是面向小程序的，所以规范上对 ReactNative 研发来说并不友好。

考虑到我们业务对于多端和性能的要求都很高，结合现有团队技术储备能力，所以选择 Taro 多端同构技术方案。在本文中，我们会重点讲述旅游事业部门票活动前端团队和公共无线技术团队合作将 Taro 技术栈与现有技术进行融合后，遇到的问题以及相应的解决方案。

三、Taro 如何与现有技术融合

Taro 提供的多端同构技术，在不需要考虑与现有技术栈的结合的前提下，是可以直接使用的。针对本身已有一套技术方案情况，就需要考虑如何将 Taro 与现有的 App 或 Web 技术进行融合。

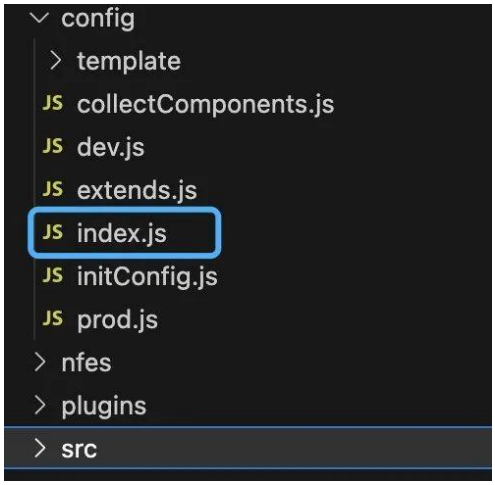
Taro 跨端方案是基于静态编译的解决方案，最终生成的是将源代码编译为目标代码并打包成可执行的文件。该文件既不能直接集成到业务方（携程）RN、Web 的框架中，也不能直接调用携程提供的业务组件，如城市、日历、支付等。因此，开发者需要对 Taro 进行适配后，才能解决与现有框架融合的问题。



如上图，Taro 的核心原理是在编译构建时通过注入自定义配置，将原本的小程序组件和 API 替换为适应不同平台的组件和 API，从而实现多端能力。这样一来，业务开发中可以使用相同的代码来适配不同的终端，消除多端开发中的差异。

3.1 融合 App（与携程 React Native 技术融合）

1）在 Taro 的配置文件中，注入自定义 plugins 插件




```
const config = {
    projectName: "xtaro-tnt",
    date: "2023-1-13",
    designWidth: 750,
    deviceRatio: {
        640: 2.34 / 2,
        750: 1,
        828: 1.81 / 2,
    },
    sourceRoot: "src",
    outputRoot: getOutputRoot(),
    plugins: [
        [
            path.join(process.cwd(), "/plugins/replaceMiniAppPath.js"),
            {
                taroConfig,
                miniappPath: "",
            },
        ],
        [
            path.join(process.cwd(), "/plugins/moveRes.js"),
            {
                taroConfig,
                miniappPath: "",
            },
        ],
        "@tarojs/plugin-platform-kwai",
        [...
        ],
        "@ctrip/xtaro-plugin-platform-crn",
        "@ctrip/xtaro-plugin-platform-nfes",
    ],
```

2）通过 Metro 打包配置，进行别名替换（原有的 taro 引用替换成新的 RN 路径）

```
function getAlias() {
  const config = getProjectConfig();
  const alias: Record<string, string> = config.alias || {};

  alias['@tarojs/components'] = '@ctrip/xtaro-components-crn';
  alias['@tarojs/taro'] = '@ctrip/xtaro-components-crn';
  alias['@tarojs/runtime'] = '@ctrip/xtaro-components-crn';
  alias['@ctrip/taro-platform-component'] = '@ctrip/taro-platform-component-crn';
  alias['@/miniapp/cwx'] = '@ctrip/xtaro-cwx-crn/dist';

  return alias;
}
```

3）抹平 Taro 的组件和 APIs 方法

Text 组件

```
import * as React from 'react';
import { CRNText } from '@ctrip/crn';
import { TextProps as TaroTextProps } from '@tarojs/components';

export default function TaroCRNText({
  children,
  selectable,
  decode,
  onClick,
  numberOfLines,
  ...otherProps
}: { children: any } & TaroTextProps) {
  const _numberOfLines: number | undefined = React.useMemo(() => {
    // fix: CRNText numberOfLines 不识别 字符串格式的问题
    if (Number(numberOfLines)) return Number(numberOfLines);
    return undefined;
  }, [numberOfLines]);

  return (
    <CRNText selectable={!!selectable} onPress={onClick as any} numberOfLines={_numberOfLines} {...(otherProps as any)}>
      {decode ? decodeURI(children) : children}
    </CRNText>
  );
}
```

页面跳转 API

```
export function redirectTo(option: IRouteOptionNavigateTo): Promise<CallbackResult> {
  const { pageName, params } = handleUrl(option.url);

  let errMsg: string | undefined;

  try {
    const context = getCurrentPageContext();
    context.replace(pageName, params);
  } catch (e: any) {
    errMsg = e;
  }

  return getNavigateResult(option, 'navigateTo', errMsg);
}

export function navigateTo(option: IRouteOptionNavigateTo): Promise<CallbackResult> {
  const { pageName, params } = handleUrl(option.url);

  let errMsg: string | undefined;

  try {
    const context = getCurrentPageContext();
    context.push(pageName, params);
  } catch (e: any) {
    errMsg = e;
  }

  return getNavigateResult(option, 'navigateTo', errMsg);
}
```

按照以上步骤，并且结合 ReactNative 的脚手架，就可以运行起来。

3.2 融合 Web（与携程 NFES 技术融合）

Taro 同构技术已在 Web 端的非 SSR 和 SSR 模式。SSR 模式是以 NextJS 框架未基础的，通过提供编译插件 tarojs-plugin-platform-nextjs 来支持。但由于这个编译插件并不支持基于 NextJS 技术扩展的 Web 框架或其它 Web 框架，所以需利用 Taro 脚手架中开放的编译能力，在构建时通过 babel 插件将 APIs 和组件库替换为支持服务端同构的版本，同时生成适配当前框架的目录及项目配置，使得 Taro 具备转换为对应 Web 框架的能力，具体参考如下步骤：

1）同 RN，注入自定义 H5 plugins 插件

```
const config = {
    projectName: "xtaro-tnt",
    date: "2023-1-13",
    designWidth: 750,
    deviceRatio: {
        640: 2.34 / 2,
        750: 1,
        828: 1.81 / 2,
    },
    sourceRoot: "src",
    outputRoot: getOutputRoot(),
    plugins: [
        [
            path.join(process.cwd(), "/plugins/replaceMiniAppPath.js"),
            {
                taroConfig,
                miniappPath: "",
            },
        ],
        [
            path.join(process.cwd(), "/plugins/moveRes.js"),
            {
                taroConfig,
                miniappPath: "",
            },
        ],
        "@tarojs/plugin-platform-kwai",
        [...
        ],
        "@ctrip/xtaro-plugin-platform-crn",
        "@ctrip/xtaro-plugin-platform-nfes",
    ],
```

2）通过 Webpack 打包配置，进行别名替换

```
return {
    name: 'alias-plugin',
    visitor: {
        ImportDeclaration(path) {
            const source = path.get('source')
            if (source.isStringLiteral()) {
                if (source.node.value === '@tarojs/components') {
                    source.replaceWith(
                        t.stringLiteral(`${nfesPlugin}/components/lib`)
                    )
                } else if (source.node.value === '@tarojs/taro') {
                    source.replaceWith(
                        t.stringLiteral(`${nfesPlugin}/taro`)
                    )
                }
            }
        }
    }
}
```

3）抹平 Taro 的组件和 APIs 方法

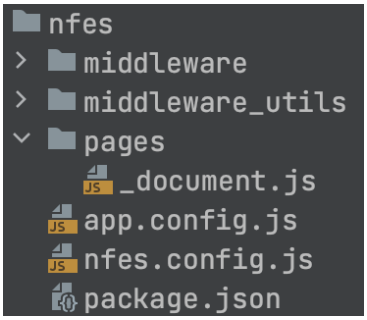
Text 组件

```
<div
    className={classNames('taro-input', className)}
    {...props}
>
    <div className='taro-input_content'>
        <div
            ref={placeholderEl}
            className={classNames('taro-input_placeholder', placeholderClass)}
            style={placeholderStyle}
        >
            {placeholder}
        </div>
        <input
            ref={inputEl}…
            }}
        />
    </div>
</div>
```

页面跳转 APIs

```
let navigateTo: navigateToType = ({url}) => {
    location.href = url
}

// eslint-disable-next-line prefer-const
let navigateBack: navigateBackType = () => {
    history.back()
}
```

4）根据自身框架的调整路由、中间件等项目配置，以下是携程 NFES 示例图



按照以上步骤，并且结合自身 Web 的脚手架，就可以运行起来。

四、技术实践

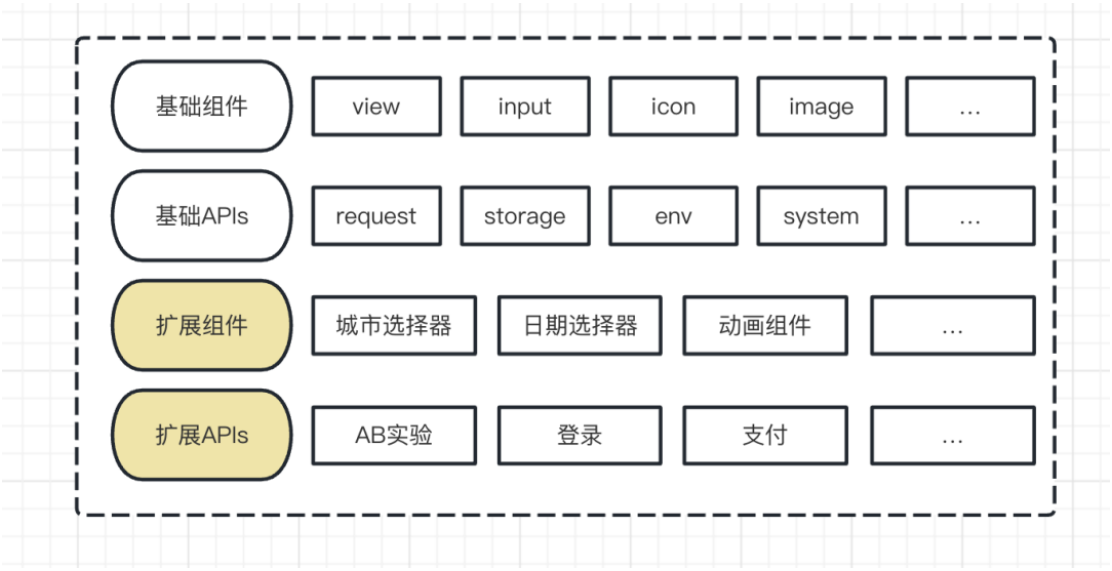
在解决好 Taro 多端框架与现有技术融合的问题之后，还需要进一步完善组件和 API 的丰富度，提升应用程序的性能，并解决 CSS 适配的问题，以实现降低开发成本和提升用户体验的目标。

4.1 组件库与 API

1） 组件和 API 丰富度

Taro 多端同构技术的核心方案是通过抹平组件库和 API 差异，实现跨端同构，从而使得性能和用户体验与独立开发单一端的应用程序相一致。然而，这种方法的不足之处在于需要开发各端的组件库和 API，以与 Taro 小程序相对齐，这需要较大的初始成本。

Taro 多端设计时已考虑到了降低研发人员首次投入的成本，所以提供对齐 Taro 小程序的组件库和 API，共计 60 多个。经过实践验证，已满足大部分常用的业务需求。



除了已提供的组件和 API 外，仍旧需要开发面向业务的扩展组件和 API，例如，弹层、折叠、日历和城市选择等组件以及支付、登录等（如上图）。大部分组件只需要在官方提供组件上做二次封装，研发成本不大。

2） 多端组件和 API 差异性

多端组件和 API 在不同平台上可能存在一些差异，无法完全抹平。每个平台有自己的特性和限制，因此在开发多端应用时，需要对这些差异进行适配和处理。

比如在动画实现方面就存在不同平台之间的差异。在 ReactNative 中，只能使用 Animation 组件来实现动画效果，在小程序和 Web 端是使用 CSS 样式来实现动画效果，为了尽量保持多端一致性，将动画实现封装成一个统一的组件，以便在不同平台上使用。封装后的动画组件，在 RN 端调用的是 Animation 组件，在小程序和 Web 端则使用组件内通过 Js 添加 Css 样式来实现动画。这种方式解决了动画实现的差异性，使得开发人员可以通过使用统一的接

口来调用动画效果，无需过多关注不同平台的具体实现细节。

把以上遇到抹平问题，可以归纳为以下 3 类情况：

情况说明	解决方案	例如
A,B 端都有此功能但差异不大	抹平差异	input、路由跳转等
A,B 端都有此功能但差异很大	抹平差异	动画组件封装成统一 API
A 端有此功能但 B 端没有	降级抹平差异或差异抹平	差异抹平：各端实现各端，如 RN 使用 Flatlist，其它端使用 scrollview 降级抹平：有的显示，没有的不显示，如头部导航栏不存在小程序中

4.2 CSS 适配

CSS 的跨端支持性是较弱的，受限于 ReactNative 的平台限制，所以支持并不友好。

ReactNative 不支持 CSS 样式的嵌套。只能将样式拆分成多个独立的对象，并通过 StyleSheet.flatten 方法将它们合并成一个对象，从而实现在一个层级节点上设置独立样式。目前只能通过差异抹平适配多端方法，牺牲其他端 CSS 灵活性。

ReactNative 不支持 CSS 中的伪元素选择器。如::before 和::after，因为它没有 DOM 元素并且不支持这些选择器。可以通过添加 HTML 节点来适应选择器写法。

上述的写法限制了多端开发的效率，但并不影响产品的功能实现。另外一些样式等问题，大部分可以使用 Babel 插件（如 rn-style-transformer）来抹平。

平台默认属性差异

属性	ios-rn	android-rn	web	小程序
fontSize	14	16	16	16
color	#000	#777	#000	#000
margin	0	0	8	0
padding	0	0	1	0

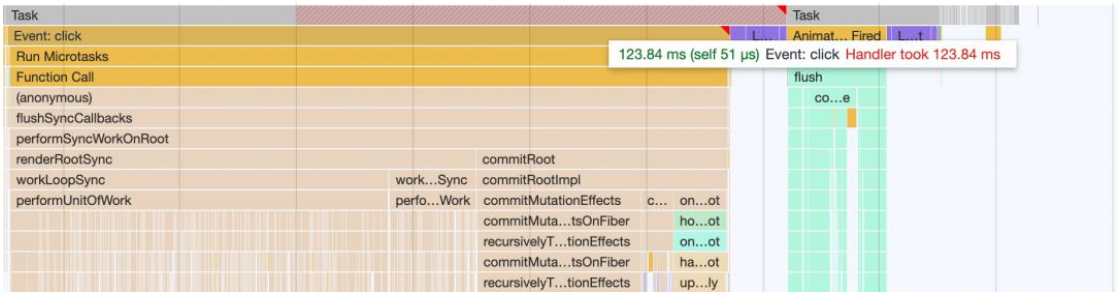
平台属性支持差异

属性	ios-rn	android-rn	web	小程序
background	不支持	不支持	支持	支持
position：fixed	不支持	不支持	支持	支持
textIndent:number	不支持	不支持	支持	支持
dashed	不支持	不支持	支持	支持

4.3 性能

Taro 由于采用的是静态编译时生成平台代码，所以性能优于动态编译时生成的方式。在 App 端性能和原生 RN 性能相当，但是在 Web 端会将 Dom 节点替换为 Web Component，而 Web Component 的渲染能力相对于原生组件较低。因此，如果在转换过程中，如果存在大量 Web Component，会导致页面渲染的变慢。在电脑型号为 MacBook Pro（14 英寸，2021 年），浏览器型号为 chrome，浏览器版本为 113.0.5672.63（正式版本）(arm64)的测试条件下，以 taro-view-core（View）组件为例，重复渲染 2000 次，总耗时大约在 123ms。如果换成 div 重复渲染 2000 次耗时大约在 17ms，大概相差 7 倍左右，实验截图如下：

Web Component 耗时：



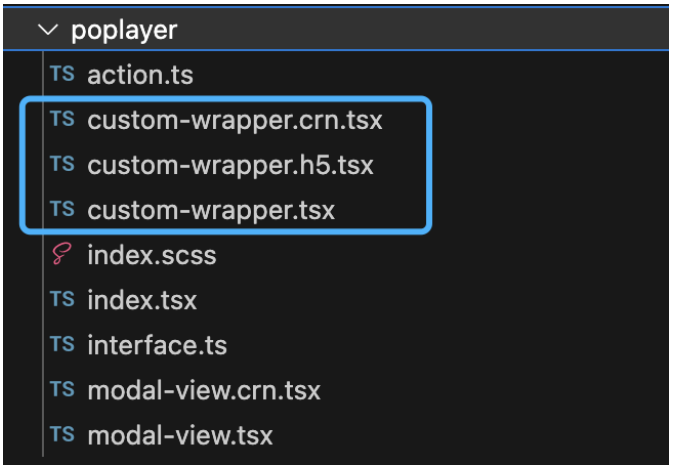
原生 div 耗时：

从以上实验可以得出，不要直接使用 Taro 提供的 View 和 Text 等组件，而是在 Web 原生组件上再包一层具备 Taro 功能的组件。

五、 适用场景和成本

5.1 View 层同构

根据交互和产品设计的需要，对于 App、H5、小程序交互方式相似度大于 70%建议可以采用一套 View，差异部分可以用 Taro 工程提供的文件扩展名方式实现各自的差异部分。



由于 PC 端的交互方式差异较大，因此通常需要编写两套 View 组件，这样做比较合适。

5.2 多端同构适用场景

多端同构适用于需要在多个平台上提供相同功能的应用程序, 达到提高开发效率和用户体验的目的。

不适用于对性能要求较高以及高度依赖平台的专属特性的应用程序，比如基于 canvas 制作的游戏，对于不适用场景且多个平台都需要支持的话，只能各自实现各自效果。

5.3 多端同构的成本

尽管多端同构技术可以减少开发的成本，但不同平台之间仍存在样式和 API 的差异，需要研发人员进行适配和补充。实际各端的研发成本对比可参考下表:

	研发成本	多端同构后	备注
App	1	0.2	
H5	1	0.2	
小程序	1	1.2	先开发的平台
PC	1	0.4	
总计	4	2.2	

随着开发经验的积累和组件的丰富化，研发和测试成本也会进一步降低。

学习成本：多端同构开发需要研发人员具备跨端开发的能力和经验，需要了解各个平台的特性和差异，同时还需要关注代码的性能、可维护性和可扩展性等方面。我们可以通过多种培训和分享来提高他们的能力和技能。

测试成本：在多端同构的开发模式下，如果不慎改错一端会影响到所有端，所以测试成本会增加。测试范围更广，测试时间也会更长，因此测试成本也会相应地增加。另外，由于不同平台之间的差异，测试人员需要具备跨平台测试的能力，这也会对测试人员的研发能力提出更高的要求。为了解决这些问题，可以普及 UT（单元测试）和 AT（自动化测试），这可以降低测试成本，提高测试效率。

生产稳定性：因为多端同构技术采用的是统一的代码逻辑和组件封装，一旦出现问题，多个平台都会受到影响。因此，在开发过程中需要进行严谨的测试和质量控制，以确保代码的稳定性和可靠性。

六、总结与展望

本文介绍的是通过使用 Taro 实现多端同构，在跨多平台业务场景中降低研发成本，提升用户体验。通过使用同一开发语言和代码框架，实现在不同端上复用代码，达到统一业务逻辑的目的。

可以预见在不久的将来，无论是基于业务需求还是技术实践与创新，都将出现更多的解决方案，使得多端开发之路变得更加平坦。同时，这套方案将成为公司主推的多端框架。

提升前端开发效率，携程机票定制代码生成器实践

【作者简介】Zilin Wang，资深前端开发工程师，擅长前端打杂，专注于 Remix、Radix UI、Haskell 等领域。Sheila Dai，资深前端开发工程师，关注前端性能优化、前端智能化。

在日常的研发工作中，编写前端界面结构占据了一部分工作量。很多 UI 组件都存在共性，如何减少编写 UI 界面的开发时间，以及提取公用的前端组件，从而达到提升研发效能的目的，是我们的重要课题。

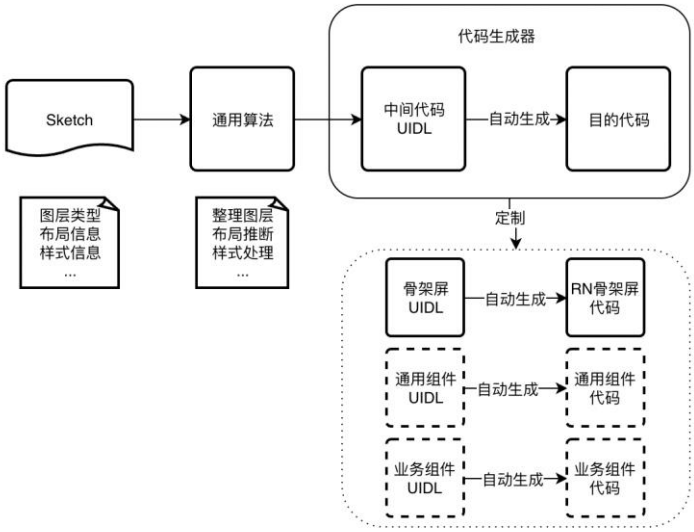
在《前端智能化探索，骨架屏低代码自动生成方案实践》中，我们曾经探索过一种自动生成骨架屏代码的方案，在此基础上，我们设计了一套代码生成器的定制流程，到达可以定制任意目的代码的效果。本文将围绕视觉稿生成任意代码，探讨代码生成器的原理与细节，最后是落地的效果展示。

一、背景

骨架屏代码自动生成，作为一个最小 MVP (Minimum Viable Product)，验证了视觉稿自动生成代码的可行性与实用性。

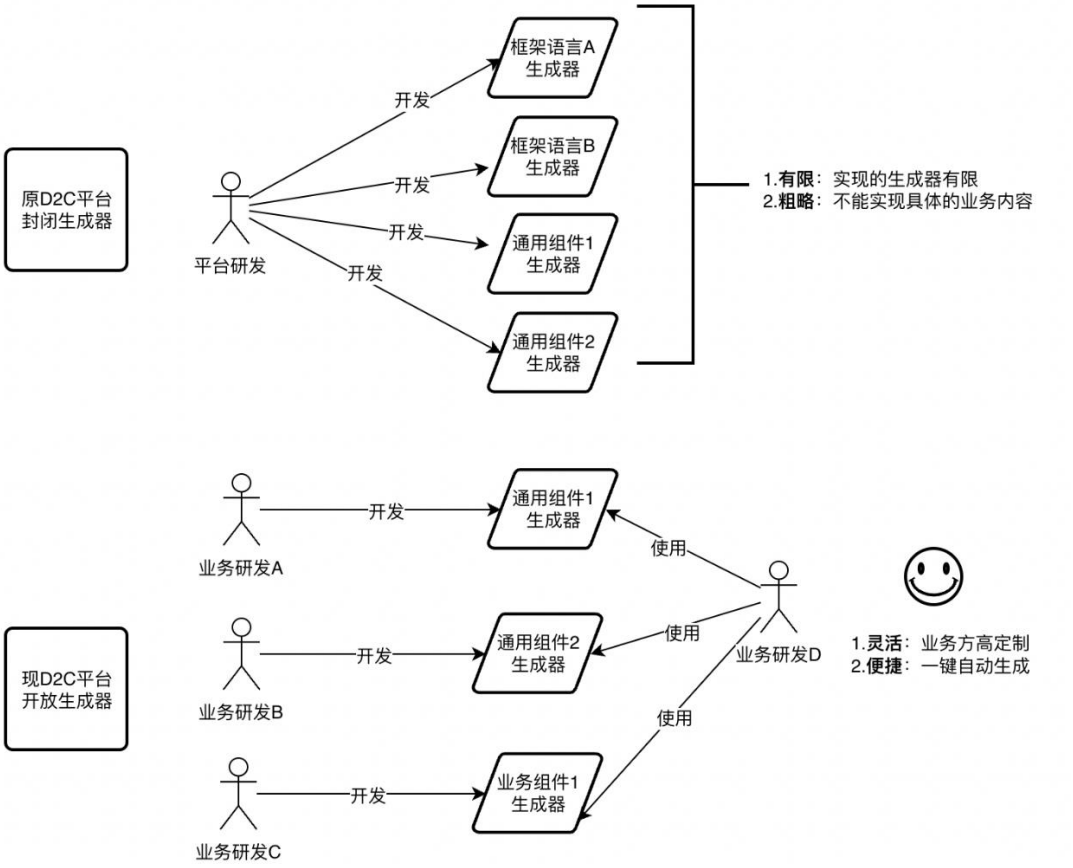
那么，除了高度重复性质的骨架屏以外，还能够实现哪些通用组件？甚至能不能实现应用的业务组件？

如图是视觉稿生成代码的粗略步骤，我们可以看到，在提取视觉稿信息后使用通用算法，得到中间代码后，可以生成我们的目的代码。在这个步骤中，通过编辑中间代码，我们可以自动生成不同的目的代码。

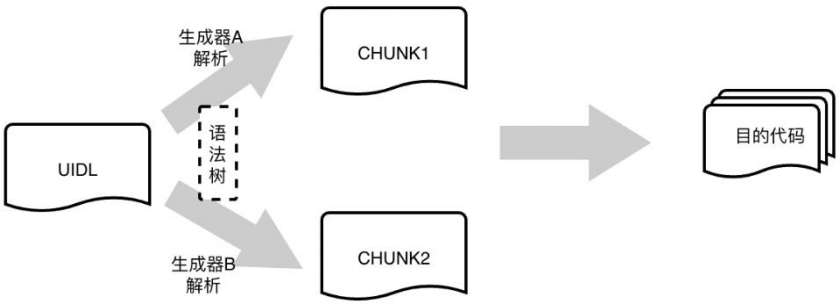


我们的预期则是把这一步骤，交付给业务研发自己来实现，通过在 D2C 平台上插入不同的

代码生成器，实时自动生成需要的目的代码。



在整个页面的转换结果上，页面上的每个组件都可以通过选择不同的代码生成器来得到相应的结果。



二、问题分析

本方案是在骨架屏探索成果上的拓展设计，我们面临的问题主要有以下三个：

1）平台角度：如何让生成器可高度定制？

我们首先需要解决的问题，是让业务研发可以自行定义代码生成器。那么我们需要把中间代码层从封闭的平台中剥离出来，变成一个开放的插件入口。

2）生成器作者角度：如何快速上手编写？

虽然针对一个指定目标代码结果的生成器，只需要一次中间代码的编写，即可作为一个公开插件在平台上提供给其他研发进行使用。但这个中间代码的编写过程依然存在一定的门槛，会让想要使用的人望而却步。我们需要降低这个门槛，让业务研发可以随时发布或者调整自己的代码生成器。

3）普通使用者角度：如何零成本使用已有生成器？

如果平台上提供的生成器，已经满足使用者的需求了，那么他可以在不学习相关知识的前提下进行一键生成代码使用。除此之外，我们需要在平台上新增一些功能，以便让使用者在这个过程中可以更加顺畅地进行预期的自动代码生成。

三、解决方案

为了解决上文提到的三个问题，我们从三个方向去解决这些问题：

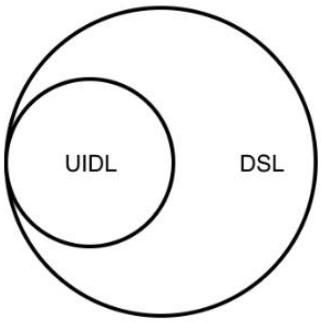
- 中间代码插件化
- 生成器编写模版化
- D2C 平台优化

3.1 中间代码插件化

在自动生成代码的流程中，我们需要把生成器这部分从封闭的平台中剥离出来，提供给业务研发进行自研。我们主要借助了 UIDL （Universal Interface Definition Language）结构，依据这个 UI 层面的数据结构解析得到我们需要的目的代码内容。

1) UIDL 简介

UIDL，即通用界面定义语言，一种用于描述 Web、移动和桌面应用程序用户界面的通用语言，是 teleport 对于 UI 描述的一种定义规范。UIDL 是 DSL（Domain Specific Language，解决特定领域问题的语言） 的子集。它与平台无关，可以应用于任何平台或者应用程序。



使用 UIDL 的主要目的就是将用户界面描述成一种机器可读的格式，以帮助开发人员更加高效地构建、测试和维护用户界面。UIDL 具体定义内容可以参考官方文档，这里给出一个简单的示例：

```
{

  "name": "Badge",
  "propDefinitions": {
    "count": {
      "type": "number"
    }
  },
  "node": {
    "type": "element",
    "content": {
      "elementType": "container",
      "attrs": {
        "class": "badge"
      },
      "children": [
        {
          "type": "element",
          "content": {
            "elementType": "text",
            "attrs": {
              "class": "badge-text"
            },
            "children": [
              {
                "type": "dynamic",
                "content": {
                  "referenceType": "prop",
                  "id": "count"
                }
```

```
            }
          ]
        }
      }
    ]
  }
}
```

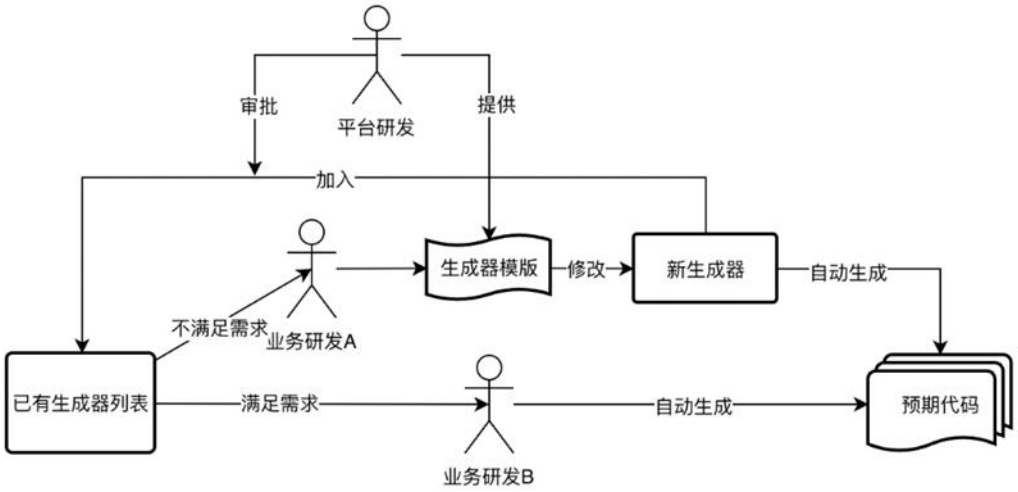
上面的 UIDL 描述了一个 Badge 组件，它接受一个 number 类型的属性 count，输出我们常见的 Badge 组件 UI 结构。通过使用 teleport 提供的 React 代码生成器，我们可以得到下面的结果：

```
import React from 'react'
import PropTypes from 'prop-types'
const Badge = (props) => {
  return (
    <div className="badge">
      <span className="badge-text">{props.count}</span>
    </div>
  )
}


Badge.propTypes = {
  count: PropTypes.number,
}
export default Badge
```

UIDL 会给出一个通用的 UI 界面节点描述，通过生成器可以渲染得到我们需要的指定框架代码。

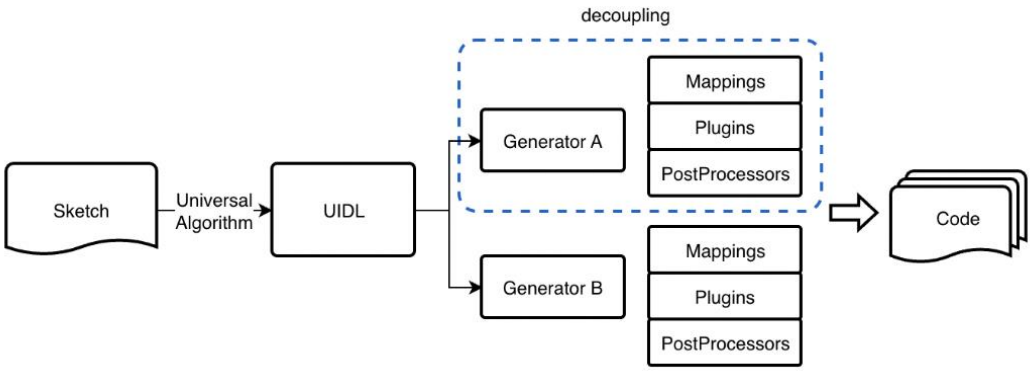
2）生成器解析

在已有生成器满足业务研发的需求前提下，可以直接进行使用；不满足则业务研发通过编辑模版可以得到新的生成器，再通过平台研发审核发布到已有生成器列表中，开放给所有研发进行代码的自动生成。



在把中间代码层插件化的过程中，我们主要借助了 UIDL 的定义结构，使用 teleport 定义的生成器结构来进行代码转换，内部包含了 Mappings（UI 组件映射）、Plugins（插件处理逻辑）、PostProcessors（后处理器）。

- Mappings：给出组件映射关系；
- Plugins：处理复杂对应逻辑；
- PostProcessors：美化代码格式等等。

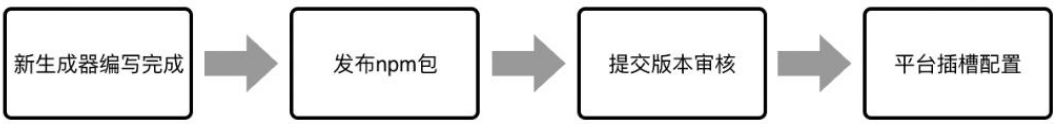


代码生成器实际上就是对中间代码进行解析，不同的解析得到不同的结果。

3）生成器插槽

我们在平台上提供了选择不同生成器的入口，根据即时选择的类型，实时自动生成对应的代码。

业务研发完成新生成器的编写后，发布成 npm 包，通知平台研发进行审核。审核成功后，相应的生成器版本会展示在平台的生成器入口处。



其他业务研发可以在这个公开的生成器列表中，选择适合自己业务场景的生成器，进行代码的一键自动生成。

3.2 生成器编写模版化

为了降低新生成器编写者的学习成本，我们提供了可本地预览的代码模版，并且封装了常用前端框架生成器及一些通用方法。

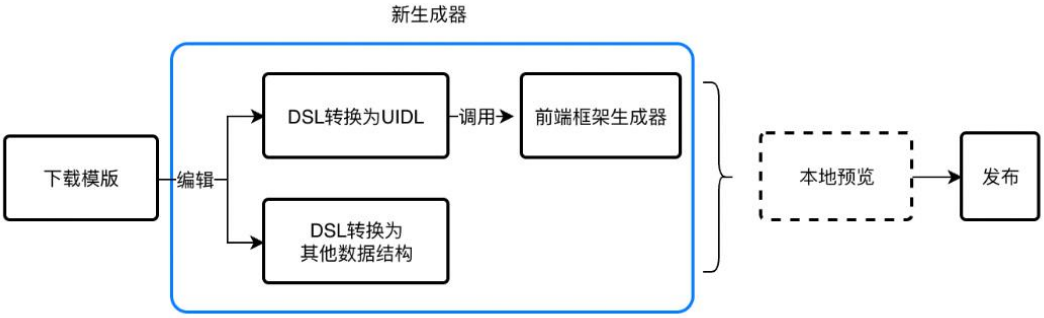
1）开发流程

业务研发 clone 模版仓库到本地，模版编写主要有两个方向：

a. 编写某种前端框架下的通用组件：视觉稿 DSL 转换为需要的 UIDL 结构（调整层级、组件名称等），再调用对应的框架生成器，生成代码；

b. 编写特定的数据结构：获取 DSL 中的节点数据，构建为新的数据结构。

再在本地进行效果预览，最后发布成为一个独立的 npm 包，通知平台研发审核后插入到插槽中。



2）本地开发与预览

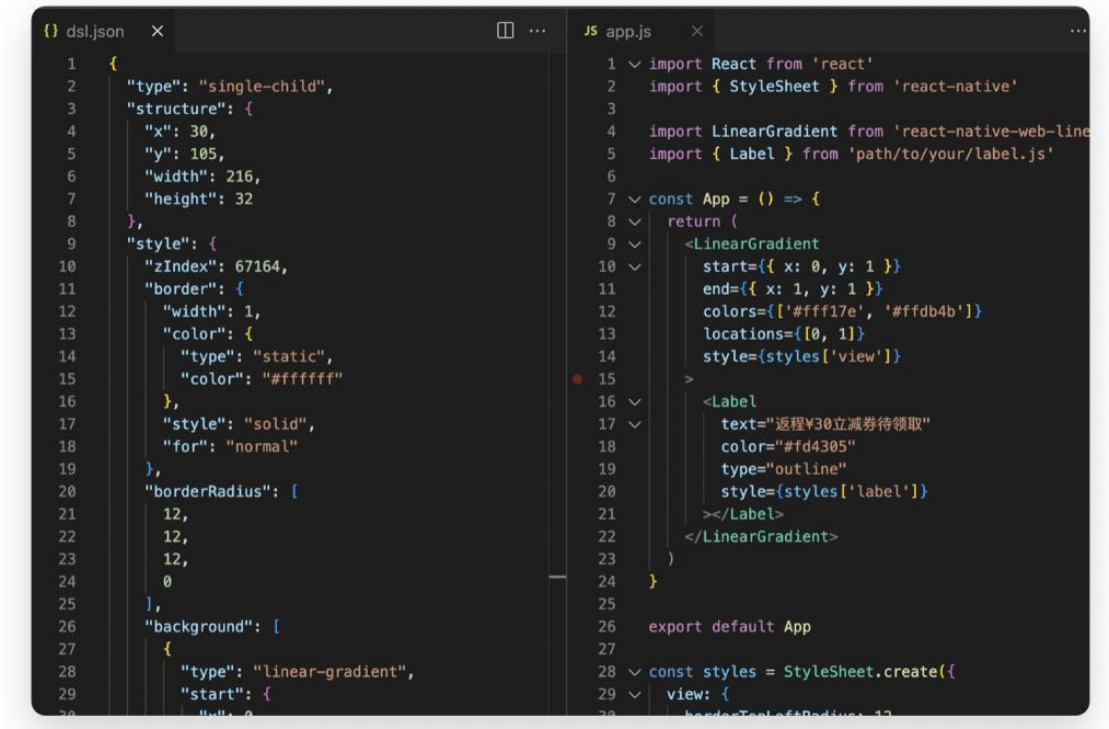
业务研发可以根据我们提供的模版来进行定制化输出，也可以参考其他公开的生成器进行编写。

以下为提供的模版结构：

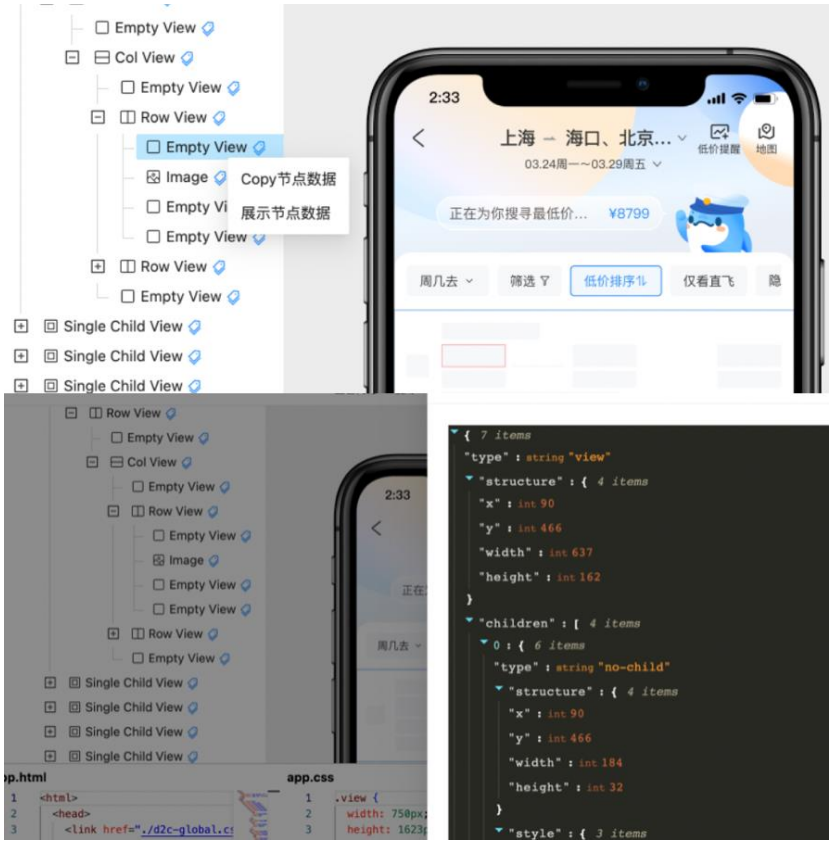
```
custom-generator-template
  |----package.json            // 发布的 npm 包信息
  |----README.md
  |----src
  |       |----index.ts        // 编写的自定义 DSL
  |----test                    // 本地测试输入与输出
  |       |----files           // 输出的文件结构
  |       |----dsl.json       // 输入的 dsl 结构
  |       |----index.ts
  |----tsconfig.json           // 默认配置源文件为 ts
  |---- dist                   // 输出的 dist 目录
```

模版中内置了输入与输出的本地便捷测试：在编辑 src/index.ts 文件后，执行 npm run test 即可在 test/目录下看到输出的内容。

下图为本地预览示例，左为 DSL 结构，右为经过生成器转换的代码：



本地测试使用的 DSL 结构可以在平台上快速复制得到。



3）常用 API 封装

为了降低生成器编写者的学习成本，我们会处理前序与后序步骤，在实际使用中都是可选的：

- 前序：原始视觉稿 DSL 转换为 UIDL 结构；
- 后序：调用 API 生成相应框架代码。

普通的 DSL 代码生成模型如下：

```
export type GenerateCustomFiles = (dsl: DSL) => Promise<CompiledComponent>
// CompiledComponent 是定义好的输出结构，包含代码依赖和代码文件
```

对于页面级别的 DSL，引进了 Design Tokens 和共享样式的概念，会在普通的 DSL 上添加一些属性：

```
export type GenerateCustomFiles = (rootDSL: RootDSL, options: GenerateCustomOptions) =>
Promise<CompiledComponent>;
// options 是传入的生成文件参数，当前有样式参数(CSS Module、Inline Style 等)、布局参数(Pixel、Rem)等
```

如果调用平台开发的 API，就可以很方便的生成业务定制化的代码：

```
const generateFiles: GenerateCustomFiles = async (rootDSL, options) => {

const rootUIDL = rootDSLToRootUIDL(rootDSL, {
  layoutUnit: options.layoutUnit,
  uidlAdjust,
})  // 前序：使用 uidlAdjust 进行 DSL 转成 UIDL 过程中属性的修改

    // 核心逻辑：UIDL 结构修改
    // ...

  const cc = await generateReactFiles(rootUIDL, { layoutUnit: options.layoutUnit, style:
options.style })
    // 后序：调用前端框架生成 API

  return cc
}
```

在后序步骤中，携程 Design To Code 支持的框架 API 如下：

- generateHtmlFiles
- generateVueFiles
- generateReactFiles
- generateReactNativeFiles

这样处理是为了让编写者无需关心复杂的前端框架自动生成过程，仅通过修改 DSL 结构层次或名称即可到达自己定制的目的。

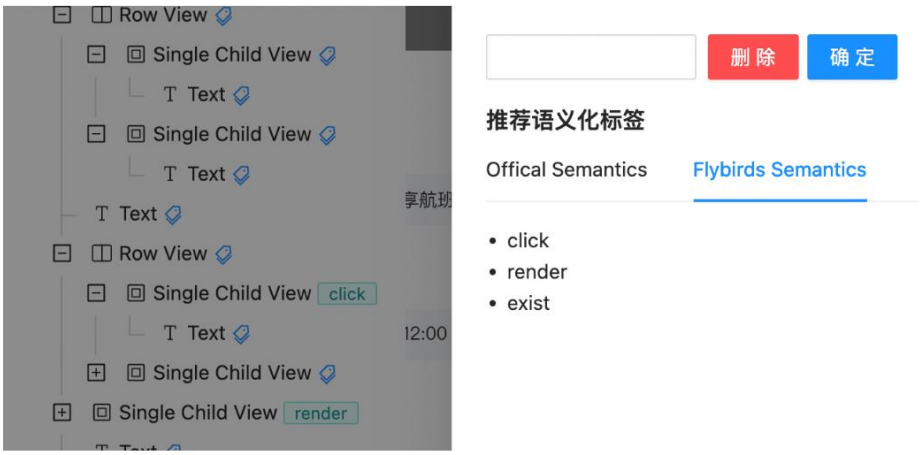
3.3 D2C 平台优化

为了让普通使用者零成本地在 D2C 平台上进行代码的一键自动生成，我们对平台进行了一些优化。接下来介绍其中三个重要功能：

- 节点标签
- 局部自动生成
- 沙盒预览模式

1) 节点标签

点击界面左侧的 DSL 节点的标签 icon，弹出浮层，可以手动给当前节点添加语义化标签名称。



手动打标签，是在 DSL 节点中新增了一个属性。在生成器中可以根据该属性实现不同的效果。

例如在落地方案 Flybirds 测试用例的自动生成中，通过给视觉稿中不同的文案部分做了三个特定标签识别：render、exist、click，分别对应了以下的执行语句：

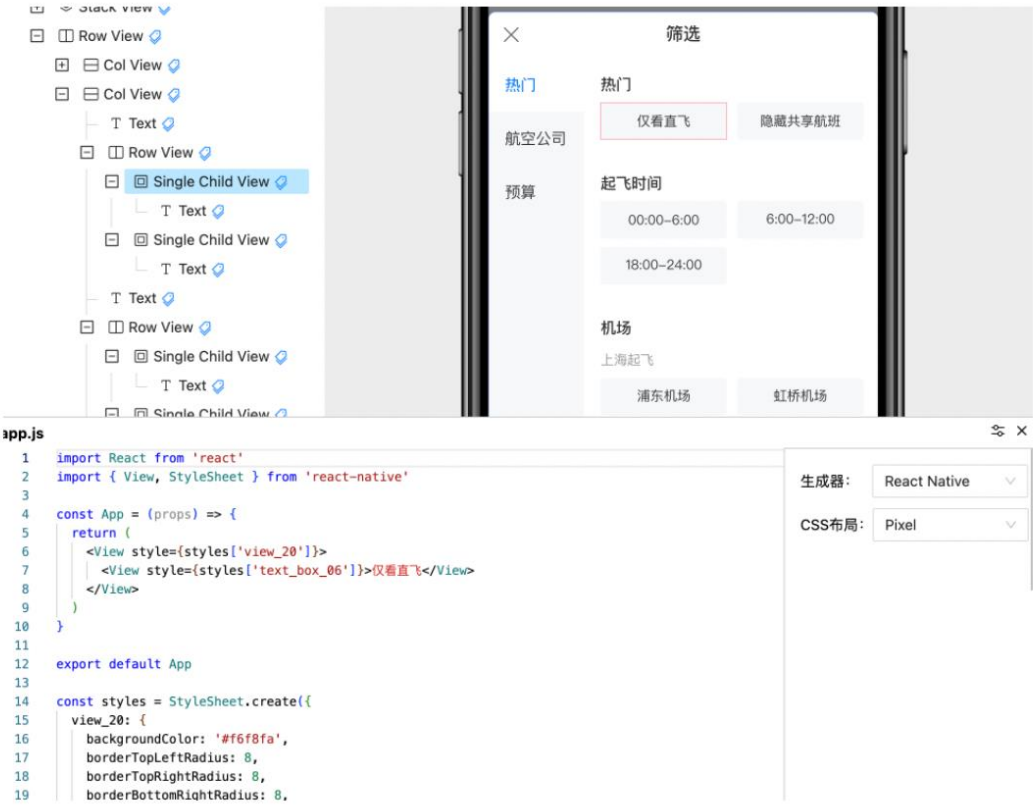
- render: 页面渲染完成出现元素 [机票]
- exist: 存在[机票]的文案
- click: 点击[text=机票]

从视觉稿信息中来说，这些结构都是文字节点，没有什么区别。通过手动打标可以在自动生成时补充额外的信息。

2）局部自动生成

下图中，左边是 DSL 节点，右边是视觉稿标注。点击左侧节点会突出显示视觉稿中内容，同样地，点击右侧视觉稿局部也会在左侧节点中同步突出对应节点。

同时，会在下端实时展示当前选择的局部画板通过生成器自动生成的代码内容。图示为点选“仅看直飞”按钮局部后，生成的 React Native 代码。

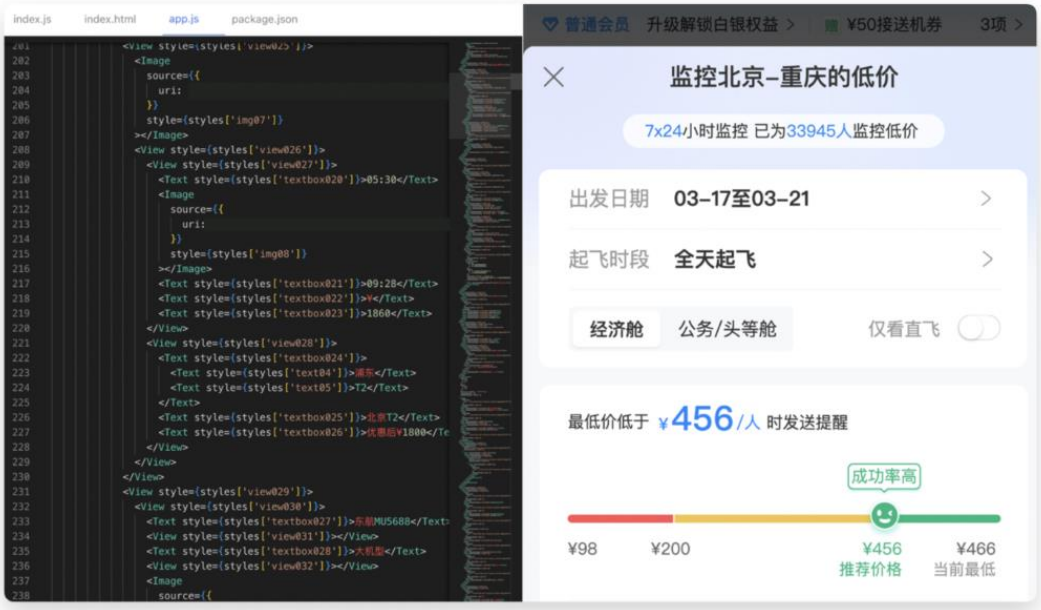


在生成器满足业务研发需求的前提下，可以点选局部后，直接复制下方的代码到仓库中进行应用。

在初始进入画板时，或者点选了 DSL 的根节点，下端会通过选择的生成器实时渲染整个页面的代码。

3）沙盒预览模式

为了更加真实地预览自动生成的代码，我们提供了沙盒模式。React / Vue 等代码可以直接在 web 端预览，React Native 我们也通过 react-native-web 转为 web 端代码，可以进行实时编辑并查看对应效果。



此外，沙盒模式还提供了一键打包下载的功能，使项目可以进行快速部署发布。

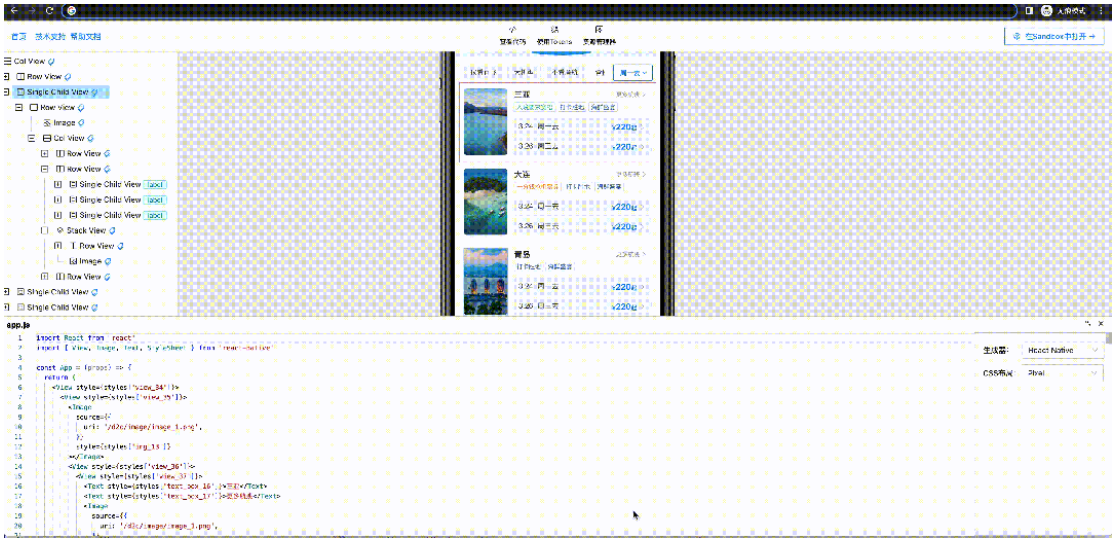
四、落地效果

落地效果均为机票实现的生成器，可在平台中进行一键自动生成。我们从三个不同的维度来进行自定义生成器效果展示。

4.1 自动生成指定框架代码

1）效果演示

图例为 React Native 代码自动生成。



2）内部实现

语言框架的应用，是作为自动化转码的一个基础底层代码内容。在此基础上，我们可以得到组件化的自动代码生成。但语言框架转码比组件化更为复杂，可以说语言框架转码是组件化转码的一个超集。

在这个过程中，借助了前文提到的 teleport 定义的 generator 生成器结构，其中需要调整 mappings、plugins、postprocessors，来得到我们预期的框架代码结构。需要处理的具体内容如下：

- **识别抽象 DSL 节点**：建立我们自定义的语言框架的首要步骤，需要把抽象的 DSL 映射为基础的 HTML 节点，识别 elementType 为正确的对应组件名；
- **映射组件**：支持 React Native 组件映射及对应引用 package，读取资源文件生成 React Native 组件。在这里最终映射预览的文件建立在 react-native-web 的基础上；

```
export const ReactNativeMapping = {
  elements: {
    container: {
      elementType: 'View',
      dependency: {
        type: 'library',
        path: 'react-native',
        version: '^0.64.1',
        meta: {
          namedImport: true,
        },
      },
    },
    lineargradient: {
      elementType: 'LinearGradient',
      dependency: {
        type: 'package',
        path: 'react-native-web-linear-gradient',
        version: '^1.1.2',
        meta: {
          namedImport: false
        }
      }
    }
  }
}
```

- 处理依赖：处理文件之间的依赖关系，加载组件，以便输出正确文件；
- 样式表风格化：第一步，将 CSS 风格的样式表转换为 React Native StyleSheet；第二步，处理屏幕适配；
- 调整 DSL 结构：处理中间 DSL，减少冗余以及修正转化错误；在有了大体的转化前后结构内容，依然需要进一步修复转化过程中的一些冗余，以减少代码 size，或者降低生

成的样式在不同机型上出错的可能性。

以 React Native 为例，我们主要需要做到：

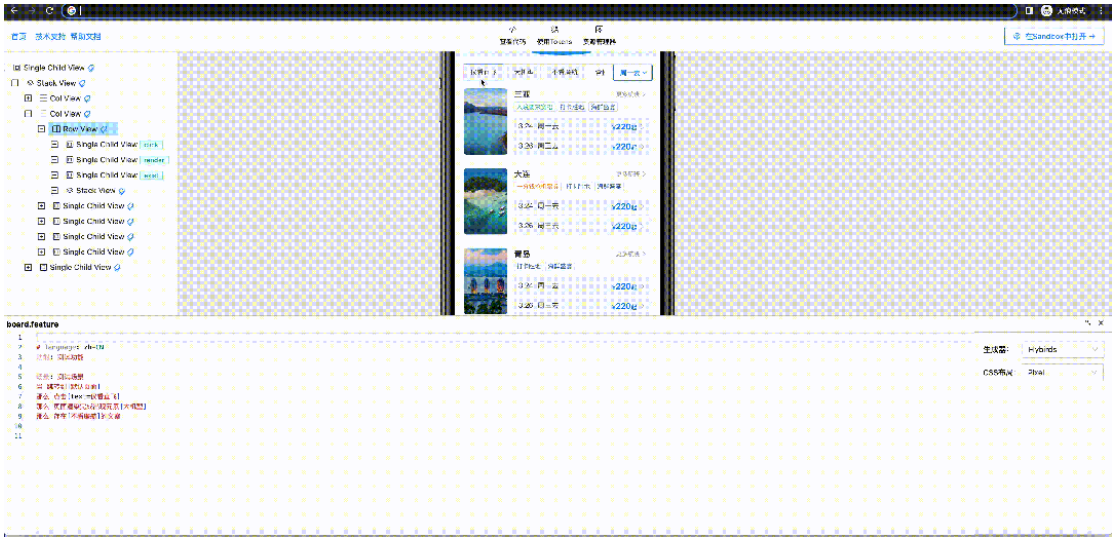
(i). 删除冗余：删除不需要的样式节点，例如 fontFamily、letterSpacing、text 节点的 width / height；
(ii). 合并代码：合并节点，例如 paddingTop 与 paddingBottom 需要合并为 paddingVertical；
(iii). 兼容样式：删除或者修改会引起不同机型样式兼容问题的节点，例如 lineHeight、fontWeight。

- 美化代码：需要格式化生成的 typescript 代码；
- 支持在线预览自动生成的 React Native 代码：我们需要在 web 页面进行实时编辑预览，因此引入了 react-native-web，以便使用者能实时调试。

4.2 自动生成指定组件代码

1) 效果演示

图例为 React Native Label 组件代码自动生成。



2) 内部实现

我们可以通过编辑中间代码，来得到预期的业务组件功能代码，包含动画效果、交互逻辑等。目标是生成一套在生产环境高可用性、复用性的组件。以标签组件为例，示范如何生成预期的组件代码。

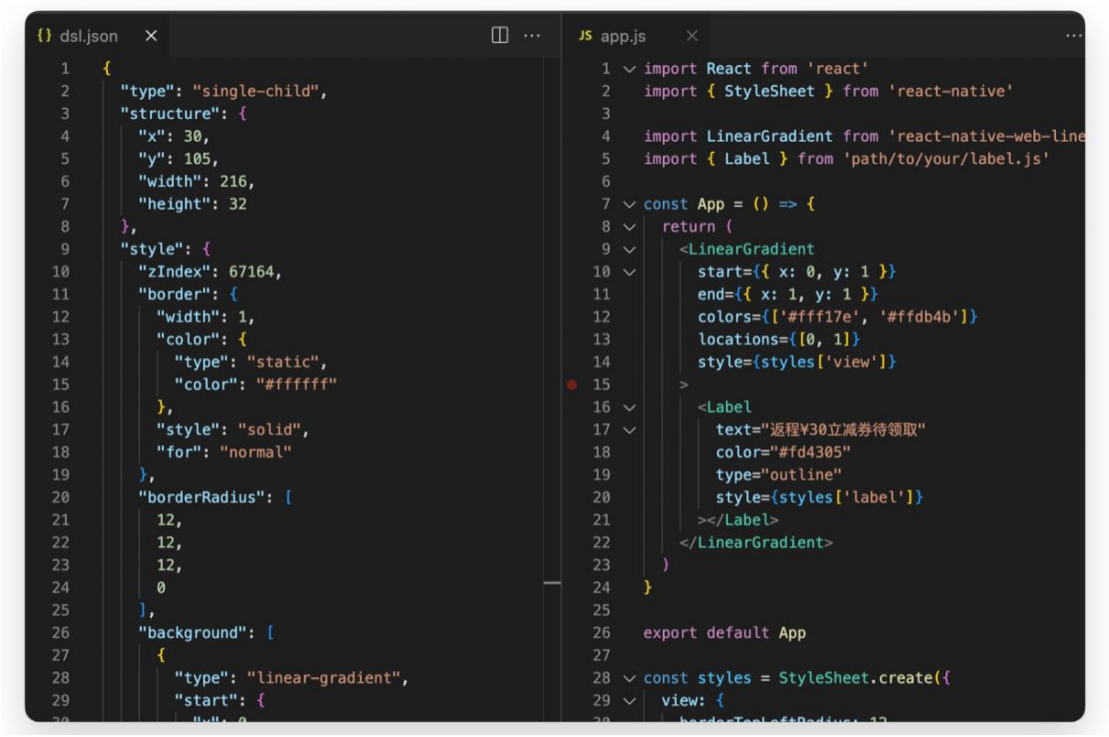
在这个过程中，需要使用多个真实场景视觉稿进行代码渲染，在线预览效果，进行代码调试与可用性测试。经过该场景的多次测试，我们需要覆盖两个结构不同的转换场景：

- 带有渐变背景的标签处理
- 常规文案处理。

转换核心代码如下（抓取对应节点信息，构建为我们预期的代码结构：

```
const parseLabelUIDL = (node, nodeType) => {
    node = Object.assign({}, node)

    const children_0_content = path(['content', 'children', 0, 'content'], node)
    const color = path(['style', 'color'], children_0_content)
    const text = path(['children', 0, 'content'], children_0_content)
    const style = nodeType === 1 ? path(['content', 'style'], node) :
      path(['content', 'children', 0, 'content', 'style'], node)

    if (nodeType === 2) {
      style.borderColor = 'transparent'
    }

    node = {
      "type": "element",
      "content": {
        "elementType": "Label",
        "name": "label",
        "attrs": {
          text,
          color,
          "type": "outline",
        },
        "style": style
      }
    }

    return node
  }
```

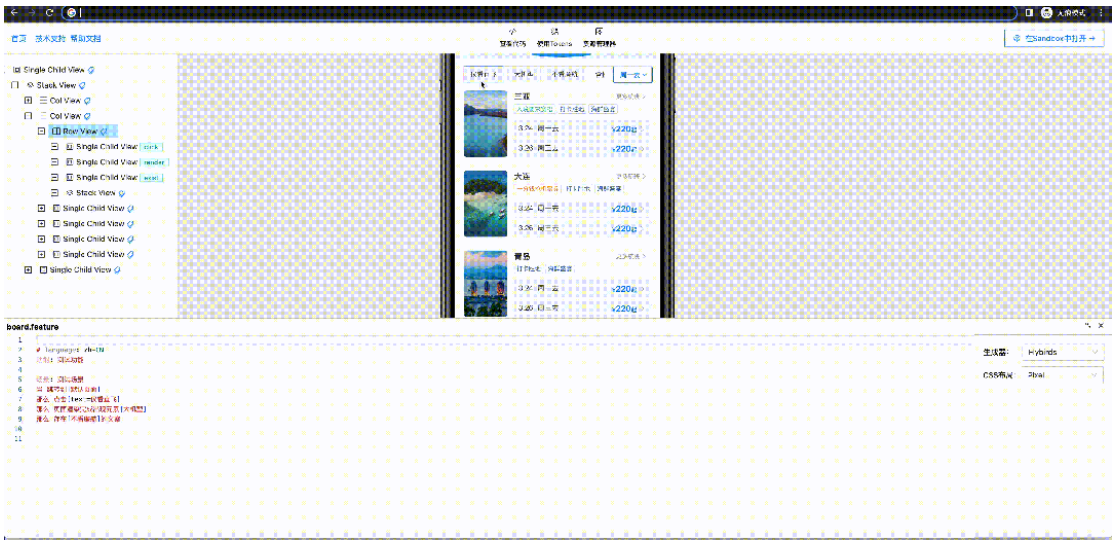
根据当前节点的类型，做不同的层级处理。原始 DSL 与目的代码前后预览如下：



4.3 自动生成测试用例代码

1）演示效果

图例为 Flybirds 测试用例代码自动生成。



2) 内部实现

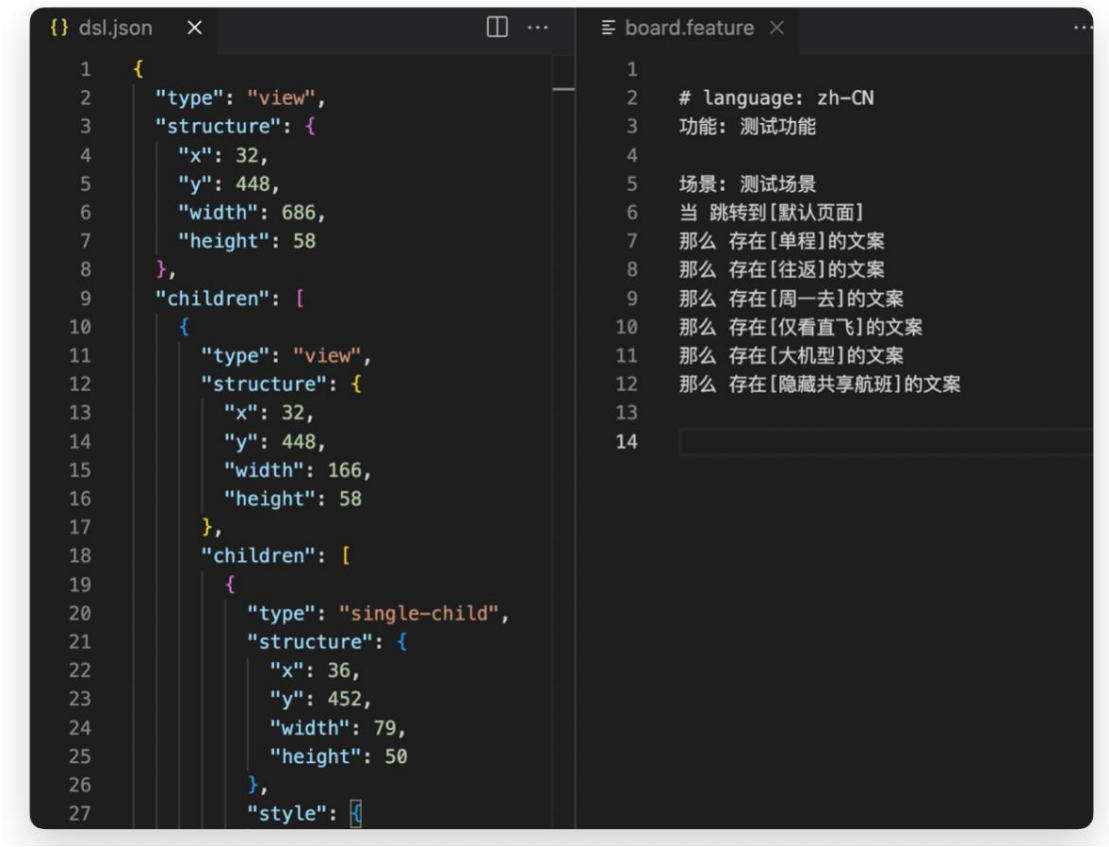
我们使用[携程机票开源的跨端跨框架的 BDD UI 自动化测试方案——Flybirds]的用例结构来进行构建该生成器的目的内容。

因为文本节点的结构是一致的，我们需要手动给不同的文本节点赋上不同的语句含义，在此处我们通过在平台上给视觉稿图层打标签进行实现。

在实现上，通过递归 DSL 结构中的文本节点进行文案内容的查找与输出对比。转换代码如下：

```
let generateText: string[] = []
const parseTestcase = (item: DSL, semantic?) => {
if (!item) return

// 识别标签赋值
const type = path(['ai', 'semantic'], item) || semantic

if (item.type === 'text') {
  let txtContent: string | undefined = path(['textAttributes', 'contents', '0', 'content'],
item)
  txtContent = txtContent!.trim()
  let txtMap = {
    'exist': `那么 存在[${txtContent}]的文案\n`,
    'render': `那么 页面渲染完成出现元素[${txtContent}]\n`,
    'click': `那么 点击[text=${txtContent}]\n`,
  }

  // 此处放开则可针对所有文本内容输出 限制则仅针对有标签的节点
  if (!type) return
  generateText.push(txtMap[type || 'exist'])
} else {
  if(item.type === 'single-child') parseTestcase(item.child, type)
  if(item.type === 'view') item.children.map((item: any) => parseTestcase(item, type))
  }
}
parseTestcase(rootDSL.dsl)
```

执行这段代码，本地运行 npm run test，前后输出内容对比：



测试用例生成作为一项实验性实践，充分说明了自动转码的自由性——有了视觉稿的原始信息结构，你可以从中解析出你需要的数据节点，并且修改为需要的目的内容。

五、未来规划

在后续的开发规划上，我们侧重于完善流程中的细节，以及通过我们官方的示例，增强 D2C 自动代码内容的实用性，继续推出更多语言框架，以及适配更多应用平台，例如小程序等等。

除了平台细节的完善，在这里主要提出未来的两个方向上的规划：

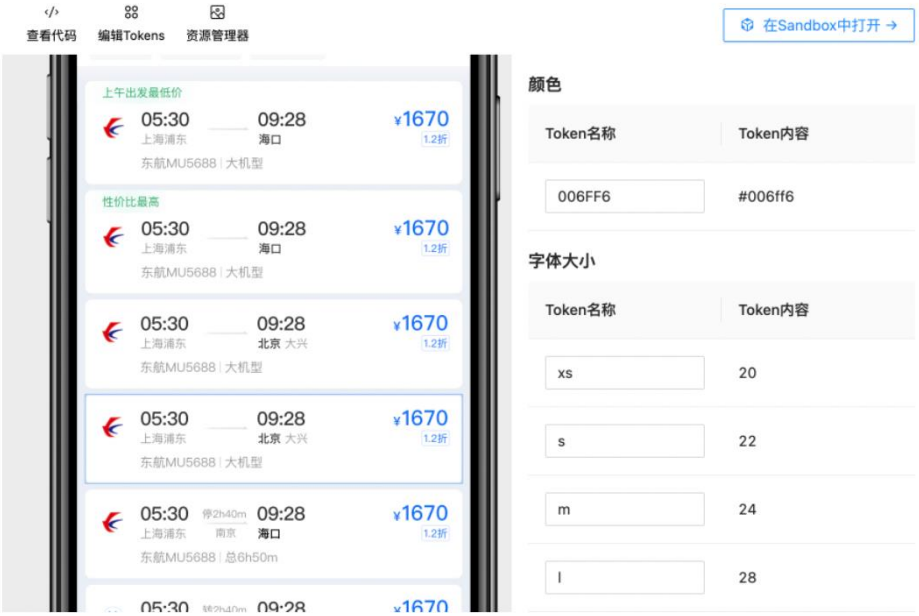
5.1 人工智能：与 copilot 结合的未来

目前的 D2C 方案，存在一个很明显的劣势，那就是无法处理复杂的逻辑反馈。我们提供的自定义生成器功能虽然可以高度定制化生成的组件，但是开发使用的内容依然没有相关的逻辑代码内容。

人工智能的出现可以弥补这一缺憾。目前人工智能还未推出完善的编写代码功能，但能够根据输入的自然语言描述或者是环境来推断开发需要使用的代码片段。copilot 可以辅助开发人员在开发过程中自动生成一些复杂的代码，减少开发人员的负担，提高效率与质量。

5.2 深度定制化：一键换肤，Design Tokens + custom DSL

Design Tokens 是一种用于描述设计系统中的基本视觉和品牌属性的集合。这些属性包括颜色、字体、间距、边角半径等。通过设计标记，我们可以将视觉属性抽象出来，从而实现统一的设计语言。



实际上目前平台已经支持了 tokens 的抽象抽取，与对应的主题映射表。除了一些设计元素上的调整，我们也可以在不同环境下使用不同的组件来进行兼容展示，例如在 React Native 中，通过修改 mapping 得到需要的交互组件。

这种深度定制化的方案，不仅可以为用户提供更好的体验，同时可以大大提高开发效率。

六、结语

携程机票开放了视觉稿生成代码流程中的生成器入口，通过让业务研发参与生成器的发布与更新，抽象出更多适合业务场景的组件/数据结构。

同时，机票在三个维度上进行了生成器落地示例，多次验证了该方案的可行性与实用性。在提高项目生产效率与设计稿还原质量的同时，确保了代码的一致性与可维护性。

视觉稿自动生成代码，可以极大地提高团队的效率，减少人为错误和重复劳动，从而更加专注于创意和创新。我们相信，在未来的发展中将会为我们带来更加高效和便捷的工作体验。

架构

提升内存管理效率，携程酒店查询服务轻量化探索和实践

【作者简介】NekoMatryoshka，携程酒店资深后端开发工程师，主要工作是缓存类组件的开发维护，并对业务应用的排障和优化有所关注。

一、背景和目标

在容器化部署成为主流的现在，降低集群中单个容器的资源需求的意义已经不只限于更少的硬件成本，同时也意味着整个集群更加轻量化，这通常会带来一系列其他优势：例如更短的恢复时间，更精确的资源控制和调度，和更快速的伸缩和部署等。但在另一方面，一味的追求压缩容器配置必然会严重影响应用在稳定性、响应耗时和吞吐量等方面的表现，所以轻量化的措施需要在多个性能维度上进行仔细的权衡取舍，以达到一个总体更优的结果。

作为携程计算量最大的接口之一，酒店查询服务一直承担着沉重的硬件成本压力，仅仅详情页集群就包含了千余台服务器实例和数十 TB 的 Redis 资源，因此对应用进行全面的轻量化有着很高的必要性和预期收益。在内存方向上，我们的主要目标是将单个容器的内存从 32GB 压缩到 16GB，并在以下两个基本方向上进行了探索：

- 减少内存增长速度：压缩本地缓存，减少浮动内存的产生，并对线程，类库，参数和代码逻辑进行针对性的优化和调整。
- 提升内存管理效率：加强 JVM 等服务依赖的基础组件其本身的性能。

由于第一个方向需要根据应用的具体代码实现来分析和排查，普适性相对较差，所以本文将主要分享查询服务在轻量化中对于内存管理方向上的探索过程和实践经验。

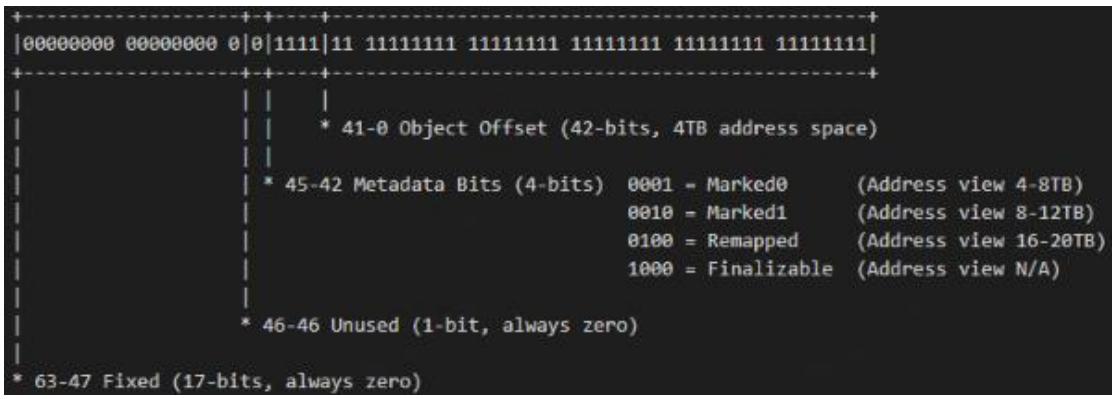
二、堆内内存管理

我们的应用原本运行在 JDK8 的 CMS 收集器之上，但是在 JDK11 以后，CMS 已经被完全淘汰。于是，要提高堆内的内存管理效率，我们首先尝试的便是对 GC 进行升级和调优。因此我们对 G1、ZGC 和 ShenandoahGC 等更现代的收集器进行了性能上的测试和对比，来尝试找出最合适的技术选型。

2.1 垃圾收集器的选型

首先，在 JDK17 上，ZGC 第一次以生产环境可用的状态登陆了 LTS 版本，所以我们这次选型起初的目标也是尝试将应用迁移到 ZGC 之上。相对于大家熟悉的 G1，ZGC 最主要的优势在于其通过着色指针和读屏障两个特性，使得用户线程几乎可以全程与标记-复制算法并行，基本解决了 YGC 的 STW 问题。简单来说，ZGC 在标记过程中会向 64 位指针的高位 4bit 中记录三色标记、重分配标记和可达性标记；当应用线程访问对象时，读屏障机制会依据指针状态和复制表信息去更新对象的地址和状态。

这样，即使 GC 线程正在后台转移、复制或清理对象，也可以保证前台线程能始终访问到正确的地址，这使得 ZGC 几乎可以做到无停顿回收。除此之外，ZGC 还向用户承诺了可扩展性：由于 ZGC 的停顿时间基本只和初始扫描中 GC Roots 的数量相关，堆的大小和活跃对象的数量并不会导致停顿时间的增长。



其次，ShenandoahGC 与 ZGC 同为新一代的零停顿收集器，总体来看，其内存布局非常类似于 G1，而并发设计则与 ZGC 如出一辙，所以我们也将其作为一个可能的备选方案。

ShenandoahGC 与 ZGC 的主要区别在于其使用的是 Brook 指针而非染色指针：即在对象头中额外记录一个指向复制后正确地址的指针。但是由于额外信息记录在对象头中，Brook 指针的读屏障无法在第一次访问后直接更新正确地址来自我恢复。另一方面，ShenandoahGC 的区块布局和回收阶段则与 G1 非常相似，甚至部分代码都是直接复用的。其不同主要在于 ShenandoahGC 利用了一个被称为连接矩阵的二维数组来取代 G1 中开销巨大的记忆集，来解决跨区引用问题：例如区块 N 引用了区块 M，则在数组的`[N][M]`坐标打上标记。

最后，作为现在最主流的收集器，同时也是 CMS 的取代者，G1 理所应当的也被我们作为最成熟和稳妥的一个选择。G1 本身的内存布局使得其对可控的停顿耗时和吞吐量的平衡上有较好的兼顾，在理论上使它更适合查询接口这种会短时间内突然生成大量临时对象的计算密集型应用。

综上所述，我们以原本在轻量化前的生产配置（16C32G+JDK1.8+CMS）作为基准，选取了以下几个组合作为测试方案：

方案	核心数	堆内存	JRE	GC
生产配置	16C	26GB	openjdk1.8.0_201	CMS
轻量化配置1	16C	12.6GB	openjdk11.0.4	G1
轻量化配置2	16C	12.6GB	openjdk17.0.2	G1
轻量化配置3	16C	12.6GB	openjdk17.0.2	ZGC
轻量化配置4	16C	18GB	openjdk17.0.2	ZGC
轻量化配置5	16C	12.6GB	openjdk11.0.4	ShenandoahGC

其他各相关参数都为默认配置。

2.2 G1 调优实践

在横向比较不同收集器的性能之前，我们首先需要按应用的需求对每个收集器做一些简单的适配和调整，以发挥这些收集器的全部性能。由于 G1 是为开箱即用而准备的默认收集器，使用起来相对简单，基本上只需要简单设置下堆大小和线程数等参数即可。然而在实际使用中，我们仍然遇到了一些小问题，需要对关键某些参数进行控制。

（1）现象 1：某个高压力场景的计算集群的 YGC 频率相对较高，GC 吞吐量不足，最终甚至会引发 FGC。

由于轻量化配置的资源本身就比较紧张，很难通过增加`ConcGCThreads`线程数来提升吞吐量，于是我们尝试放宽了`MaxGCPauseMillis`来减少 G1 的回收压力。作为 G1 最核心的参数，当`MaxGCPauseMillis`过小时，G1 会自动调整其他 GC 参数来尽可能满足该目标，进而导致 YGC 非常频繁并影响吞吐量。由于各个应用的情况不同，需要开发人员手动进行基准测试来找到最合适的数值：当我们把数值从 200 调整到 300 后，平均响应有非常明显的下降，而继续上调的边际效应则很不明显。

（2）现象 2：应用在长时间稳定运行后，老年代突然大量上涨后却不及时回收，在数小时内老年代都维持高位，挤占年轻代的占比并影响到了 YGC 的停顿耗时。

由于查询服务主要是无状态的计算逻辑，除了一部分本地缓存外大部分的对象都是相对短命的，但是当 GC 压力上升的情况下仍然会有大量对象进入老年代。这里我们通过缩小`InitiatingHeapOccupancyPercent`来降低 MGC 的阈值，让 G1 可以及时回收掉进入老年代的短命对象。

除此之外，我们也对 JDK8、11 和 17 上的 G1 进行过纵向对比，发现实际反映到机器性能和响应耗时维度上的区别非常小，在排除掉宿主机本身的硬件区别后几乎可以忽略不计。最终，由于依赖类库和监控平台等各种原因，最终选择了相对成熟的 JDK11 作为 G1 的平台。

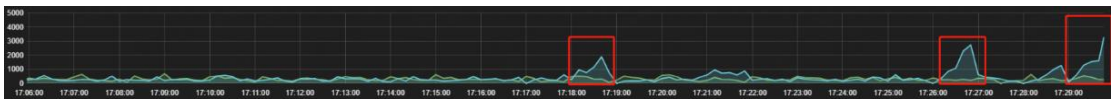
2.3 ZGC 调优实践

与 G1 的普适明显不同，对 ZGC 上的适配工作明显困难的多，作为最现代的收集器之一，ZGC 并不是万用的银弹，因此也并没有成为 JDK17 的默认收集器。相比 G1，ZGC 并不能简单的适配于所有场景，我们在试运行过程中遇到了一系列难以解决的问题，经过大量的参数调整和性能测试后，才能发挥其全部的实力。

（1）现象 1：在访问量突然上涨时，会观察到非常显著的分钟级响应尖刺。

由于 ZGC 会使用之前数次的 GC 指标来预测下一次 GC 的回收策略，使得其相比 CMS 等传统收集器更容易受到流量波动的冲击，在高流量压力下的鲁棒性很差。解决响应尖刺的办法主要有两个：首先是通过提高`ZAllocationSpikeTolerance`的数值来减少触发 GC 的阈值，其次是打开周期性的主动回收参数`ZProactive`，并通过减少`ZCollectionInterval`来缩短两次主

动 GC 的间隔。两个参数的具体数值需要开发人员手动调试确定，但是在高压力下无论怎么调整参数，提升都相对有限。



（2）现象 2: GC 日志出现大量 Allocation Stall/Rellocation Stall，同时监控上发现秒级的 STW 出现，某些时候甚至伴有 OOM 报错。

虽然 ZGC 的停顿时间很短，但整个回收阶段很长，期间用户线程一直处于并发运行的状态。这使得回收过程中会产生大量的浮动垃圾，只能等到下次 GC 时再回收。此时如果浮动垃圾占满了整个堆使得回收无法继续，ZGC 就会直接暂停对应的用户线程，来优先执行回收任务，同时在日志上记录对应线程的 Allocation/Relocation Stall。简而言之，Allocation Stall 是一种当 GC 吞吐量不够时触发的用户线程暂停，大量秒级的 Allocation Stall 甚至比 FGC 的影响更大。

这种情况一般都是 GC 回收速度跟不上内存申请速度导致的，如果 GC 资源相对充足的话，可以通过上面两个主动 GC 参数来增加 GC 频率，而如果 GC 资源本身就很匮乏，则只能通过增加 GC 线程数`ConcGCThreads`和`ParallelGCThreads`来根本性的解决问题。

```
355s][1556][gc          ] Allocation Stall (http-nio-8080-exec-44) 3246.563ms
355s][1556][gc          ] Allocation Stall (Custom-Parallels-Worker-thread-20) 3238.933ms
355s][1556][gc          ] Allocation Stall (http-nio-8080-exec-26) 865.476ms
355s][1556][gc          ] Allocation Stall (streaming-parallels-pool-thread-12) 3240.697ms
355s][1556][gc          ] Allocation Stall (streaming-parallels-pool-thread-5) 3240.657ms
355s][1556][gc          ] Allocation Stall (Custom-Parallels-Worker-thread-6) 3226.051ms
```

（3）现象 3：ZGC 堆使用的 RSS 持续上升，其大小不会随着内存使用情况智能伸缩，最终导致了堆外溢出。

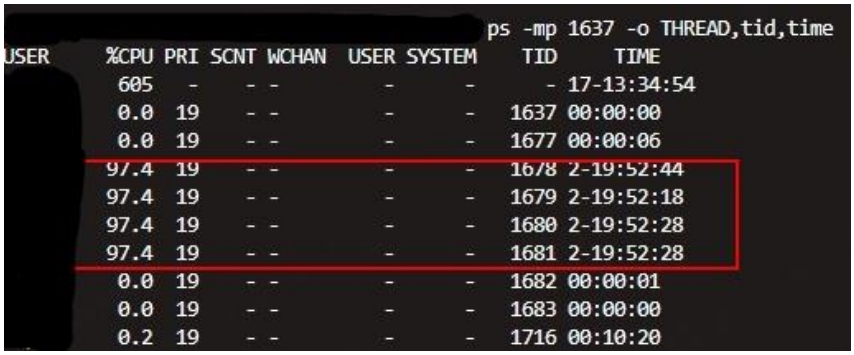
低版本的 ZGC 并不会主动将长期未使用的堆内存返还给系统，JDK13 后 ZGC 提供了`ZUncommitDelay`参数来设置将空闲内存返还给 OS 的期限，可以通过缩短这个值来使得 RSS 空间被更加灵活的使用。为了保证生产环境服务的稳定性，我们直接通过让堆大小的上下限相同来防止堆的伸缩。

2.4 基准测试的结果

在实际试验中，我们首先观察到 ZGC 确实无愧于其零停顿收集器的名号，可以做到在全程任何情况下都达到百微秒级每次的停顿时间，每分钟累计不超过 1ms，同时 CMS 中令人困扰的 FGC 现象也不再成为问题。而与之原理相近的 ShenandoahGC 的性能表现也非常好，平均每分钟的停顿时间也不超过 10ms。G1 与这些新一代的收集器相比虽然逊色许多，但是仍然能在仅仅使用生产配置一半的内存下，达到比 CMS 更好的 GC 性能：其 YGC 停顿约下降了 50%，每分钟停顿约为 200ms 左右，并且 MGC 也基本可以满足老年代的回收需要，数天时间内没有观察到 FGC。

但是随着流量压力的上升，我们很快发现作为首选的 ZGC 和 ShenandoahGC 等零停顿收集器实际上并不适合查询接口这样的运算密集型应用：他们的运行显著地依赖于资源开销，最终严重挤占了业务逻辑的计算资源，使得响应耗时飙升，服务趋于崩溃。

为了稳定服务，我们不得不重新分配了更多计算资源并降低了流量，并得出了结论：即使使用了两倍的线程资源，在 CPU 利用率三倍于 CMS 的情况下，ZGC 和 ShenandoahGC 仍然只能达到相当于生产环境约 50%-60%的极限吞吐量。与之相对，G1 在这方面的表现则好得多，在轻量化配置下比起生产配置的整体吞吐量仅稍有下降，在相同 QPS 下的 CPU 利用率变高了约 5-7%，几乎可以忽略不计。经过后续排查，我们发现主要的原因是 ZGC 的四条 ZWORKER 线程几乎每个都会 100%的占用一个核心（如下图），大量的吞吃了 CPU 资源并影响到了核心处理流程。



在内存方面，ZGC 的实际表现也并不尽如人意。ZGC 不仅比传统收集器记录了更多的额外信息，且很多优化和特性（例如字符串去重、分代、指针压缩等）也暂时无法使用，这使得 ZGC 无论是堆内还是堆外的内存开销都要明显高于 G1 和 CMS。测试中我们使用 NMT 和 JMX 简单对比了各收集器在未接入流量一段时间内的平均内存开销，发现 ZGC 的堆内开销大约比 CMS 高三分之一，堆外则高达 CMS 的 10 倍左右。

收集器	已提交的堆内存	已使用的堆内存	堆外GC内存
CMS	12.6G	4.1G	72M
G1	12.6G	4.8G	298M
ZGC	12.6G	6.2G	816M

综合来看，ZGC 等零停顿收集器虽然可以达到 10ms 以下的每分钟停顿时间，但是其占用的大量 GC 资源会严重影响核心计算逻辑。在资源紧张的轻量化场景下，切换至 ZGC 导致了服务在极限压测中损失了相当于 100%生产流量的吞吐量，同时接口的响应耗时上升了 70%。与之相对，G1 在响应耗时上的表现则几乎与原来相同，各细节指标上也仅仅只有 CPU 利用率略差。

收集器	配置	JRE	CPU利用率	极限吞吐量（与生产的比例）	响应耗时 ms	YGC停顿 ms/min	FGC频率 /h
CMS	16C32G	8	12%	2倍以上	30ms	500-2000	2-4
G1	16C16G	11	16%	2倍以上	30ms	200-500	0
ZGC	16C16G	17	65%	90%	49ms	1-10	0

2.5 迁移到 JDK11

由于 ZGC 和 ShenandoahGC 在测试中表现不佳，且各种问题在资源紧张的限制下几乎无法解决，我们最终将目光转向了更为成熟稳定的 G1+JDK11 的组合。从 JDK8 迁移到 11 是两个连续的 LTS 版本之间的迁移，比起直接迁移到 17 来说简单很多。我们在实际迁移中主要遇到了三个类型的问题：官方类库缺失，权限控制，以及第三方类库报错。

官方类库缺失：JDK11 中移除了一系列官方类库，其中部分类库只是从 rt.jar 中被拆分，可以简单的通过 maven 补回，例如 javax.*，但是其他一些包含危险操作的类库则被直接删除，例如 jdk.nashorn，sun.misc 等，一般也可以通过重写来绕过这些代码。

权限控制：JDK11 中对各种权限做了更精细的控制。例如自代理需要使用参数`-Djdk.attach.allowAttachSelf`控制，而跨类库的反射权限则需要用`--add-exports=`和`--add-exports`来打开。如果未能注意到这些参数则会造成大量权限控制报错。

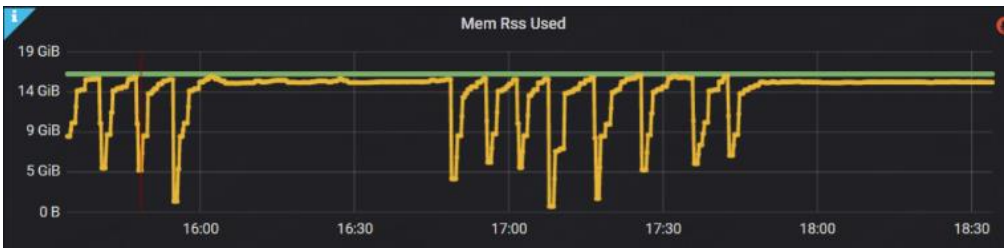
第三方类库报错：此类报错一般都是兼容性问题导致，Lombok 和 AspectJ 等类库需要根据 JDK 版本来选择对应的类库版本。主要排查难点在于报错的形式五花八门，报错信息对定位几乎没有帮助，有时候很难确定是哪个类库导致的。

三、堆外内存管理

一般来说，对于运行在容器中的单个 Java 应用，大部分堆外相关的细节都会被虚拟机给屏蔽掉，导致开发人员往往很少会深入到相关问题。然而，由于这次轻量化后剩余的内存资源非常紧张，我们被迫给堆外留下了非常有限的空间，导致了后续测试过程中出现的一系列问题。

3.1 问题表象与排查

具体来说，在生产测试中，我们经常观察到应用经常在半夜多次无故宕机后被拉起，最终反复点火失败导致应用崩溃无法继续服务。经过大量试验后，我们发现这一现象与 JDK 版本和 GC 无关，可以在多个轻量化配置上复现，现象为 RSS 在较长的一段时间内持续上升，最终导致了应用多次崩溃重启。



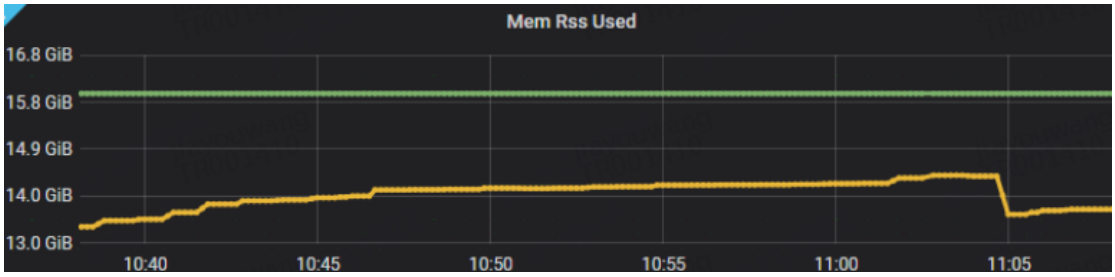
由于是 RSS 持续上升，我们排查时首先怀疑是堆内溢出。但是却并没有在应用日志上发现 OOM 报错，且使用 JFR 检查堆内存增长情况后也并没有找到明显的溢出迹象。所以我们进一步检查了机器的 dmesg 日志，发现反复崩溃的原因是 malloc 申请不到内存，导致内核的 oom_killer 线程直接 kill 掉了 DMESGTomcact 进程，然后又被重启脚本重新拉起。

```
[Sun Feb 13 04:06:24 2022] Memory cgroup stats for [redacted]
[Sun Feb 13 04:06:24 2022] Tasks state (memory values in pages):
[Sun Feb 13 04:06:24 2022] [  pid  ]   uid  tgid total_vm      rss pgtables_bytes swapents oom_score_adj name
[Sun Feb 13 04:06:24 2022] [ 184958]     0 184958      256        1    32768        0          -998 [redacted]
[Sun Feb 13 04:06:24 2022] [ 532438]     0 532438   115843     1869   159744        0           924 [redacted]
[Sun Feb 13 04:06:24 2022] [ 532890]     0 532890    30379     4652   286720        0           924 [redacted]
[Sun Feb 13 04:06:24 2022] [ 532919]     0 532919    26967     3756   258048        0           924 [redacted]
[Sun Feb 13 04:06:24 2022] [ 532920]    28 532920   123255     1068   208896        0           924 [redacted]
[Sun Feb 13 04:06:24 2022] [ 532921]     0 532921    28765     2000   258048        0           924 [redacted]
[Sun Feb 13 04:06:24 2022] [ 314596]  1004 314596 250264157 15735681 132706304        0           924 java
[Sun Feb 13 04:06:24 2022] Memory cgroup out of memory: Kill process 314596 (java) score 3429 or sacrifice child
[Sun Feb 13 04:06:24 2022] Killed process 314596 (java) total-vm:1001056628kB, anon-rss:4931624kB, file-rss:53592kB, shmem-rss:57957756kB
[Sun Feb 13 04:06:25 2022] oom_reaper: reaped process 314596 (java), now anon-rss:0kB, file-rss:8kB, shmem-rss:58038112kB
```

由于堆内存是基本稳定的，我们使用 NMT baseline 对 JVM 的堆外内存使用情况进行了检查。虽然现代 GC 本身的 native 占用相对较高，但增长并不显著，总体内存使用很稳定，也没有泄露的倾向。

```
GC (reserved=8942MB +297MB, committed=782MB +297MB)
   (malloc=718MB +297MB #122496 +54172)
   (mmap: reserved=8224MB, committed=64MB)
```

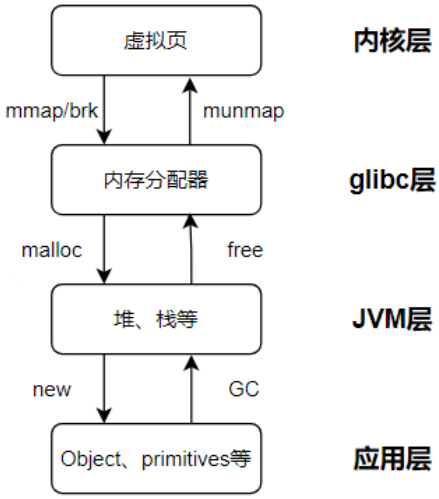
进一步向下排查，我们用 `gdb --batch --pid 36563 --ex 'call malloc_trim()'` 强行回收内存后，发现 RSS 有明显下降，至此基本判断是 glibc 这一块造成了泄露问题。



3.2 内存分配器

要解释溢出的原因，首先需要了解一下内存管理的机制。对于 Java 应用来说，内存管理一般分为四层：内核负责管理和映射虚拟页，glibc 进行通用的内存算法管理，JVM 负责屏蔽

内存申请和回收的细节，而最上层才是 Java 应用。



由于 mmap 和 brk 是系统调用，如果应用每次申请内存时都直接访问内核函数的话，性能会非常差，代码实现也更加困难。所以，linux 会使用通用的内存分配算法来缓冲和规划内存的使用，其主要关注点在三个指标上：

- 减少申请和释放操作的时间和开销
- 减少小对象分配带来的内存碎片
- 减少分配器本身数据结构的额外内存开销

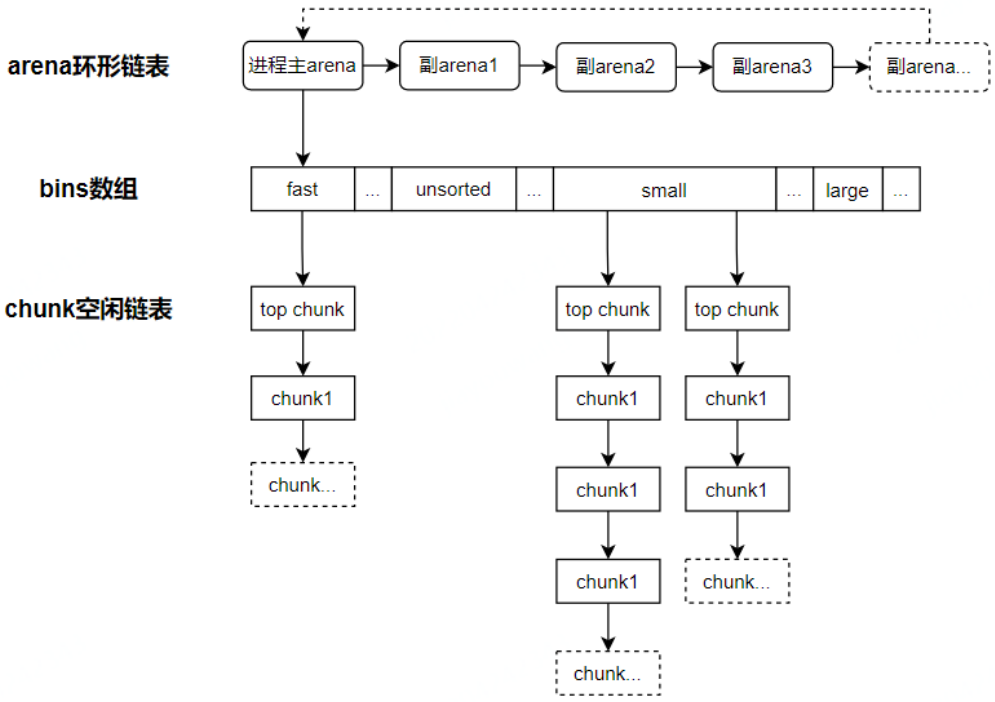
3.3 默认分配器 PTMALLOC 的优缺点

在默认情况下，glibc 使用的是其原生的 ptmalloc2 内存分配器，由以下三层数据结构组成：

- arena（分配区）是 ptmalloc 中的内存缓冲区，在一个环形链表上被管理。也是 ptmalloc 中最小的锁颗粒度。

- bin（空闲链表）是 arena 中用于管理可用内存块的链表。不同的 bins 根据其管理的内存块大小而被分为 fast、unsorted、small 和 large 四种。

- chunk（内存块）则是用户申请和释放内存的最小单位。bins 的头部永远是一个被称为 top chunk 的空闲块，当没有合适的 chunk 时，会扩容并返回 top chunk 来处理请求。



ptmalloc 作为标准实现，主要使用了以下几个方法来优化以上三个重要指标:

- 内存池（减少频繁的系统调用和内存碎片）：用户 free 掉的内存，不会被直接被归还给系统，而是暂存到 bins 中，供下次申请时直接分配。

- 多分配区（减少锁竞争）：所有内存操作都需要加锁，如果没有找到未上锁的 arena，则会新增一个副 arena 并上锁，直到 arena 的数量上限。

ptmalloc 虽然满足了内存分配器的基本需求，但是本身实现有很多缺陷，导致了内在的 OOM 倾向：

- 额外内存开销大：每个 chunk 都需要额外消耗 8b 的内存，而 chunk 是内存操作的最小单位，这会导致整体上浪费了非常多内存。

- 内存利用率不稳定：由于多分配区的机制，激烈的锁竞争会导致副 arena 数量快速增多。并且，新增的副 arena 永远不会被销毁，且保留会其初始的 chunk。这意味着在一台 16 核的标准机器上最多会有 128 个 arena，并占用高达 8G 的堆外缓冲区。

- 多线程性能差：所有的内存操作都需要进行悲观锁的加锁解锁操作，导致其性能较差。同时，即使有多分配区机制，在动辄 500 个以上线程的生产环境中这个并发量完全不够。

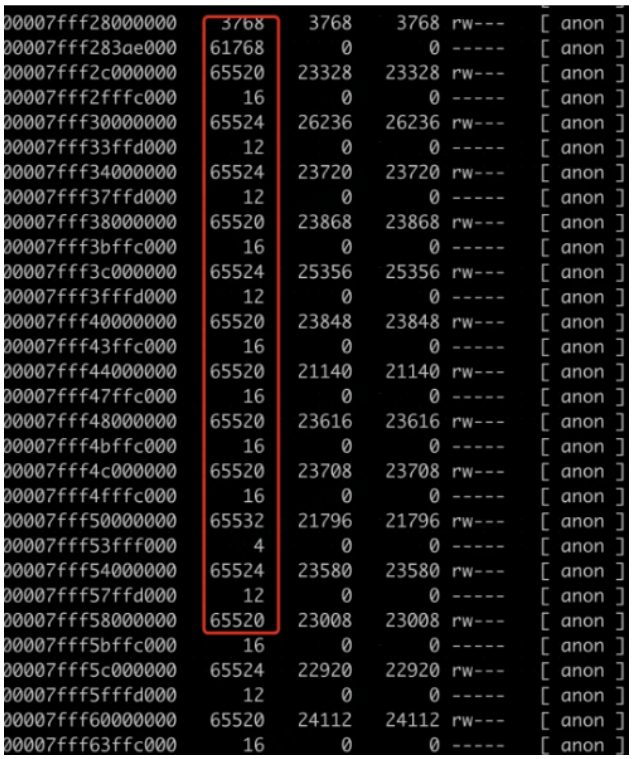
- 回收机制简陋：由于 bins 是链表结构，ptmalloc 的内存收缩必须从上向下收缩，这意味着只要后申请的内存没有被释放，之前申请的所有 chunk 都无法被收缩，这导致了在管理长周期内存时，有内存泄漏的可能性。

具体到这次详情页的溢出，则主要有三个原因：

- 前置条件：由于轻量化的需要，应用目前仅仅能给堆外大约 2.5G 的空间。并且 G1 本身使用的堆外空间是 CMS 的 4-6 倍之多。而原来的 32G 的堆外空间充足，所以之前没有发现类似问题。

- 由于大量的缓存、快照和报文的处理，应用本身有非常频繁和重量级的 NIO 和序列化/压缩操作（尤其是点火的时候，线程数非常多），这导致了应用会高频的申请和释放堆外内存作为 IO 缓冲区。因此 ptmalloc 在这种情况下新增了大量的 arena 来避免频繁的锁竞争（下图中有大量 64M 大小的内存块）。

- ptmalloc 本身的释放机制就导致申请的内存被归还的特别慢，甚至有内存溢出的倾向，这些因素综合在一起引起了 OOM 的发生。



3.4 解决方案 JEMALLOC

考虑到 ptmalloc 的性能相对较差，我们将目光转向了第三方的内存管理器。无论是谷歌的 tcmalloc 还是脸书的 jemalloc 都完全是默认分配器的上位替代，各项性能远超 ptmalloc，并且迁移起来非常方便。虽然 tcmalloc 和 jemalloc 两者之间优劣差别不大，但是由于 jemalloc 相对优秀的工具链，我们最终优先对它进行了测试。

jemalloc 是专精于多核多线程场景的内存分配器，可以说在并发量越大的情况下，jemalloc 的优势越明显。对比 pemalloc，jemalloc 有以下的优势：

（1）内存碎片率：jemalloc 承诺至多 20%的内存碎片。

通过将内存块根据大小进一步细分为 232 个小类，同一个 bins 中的内存块大小一致，向上取整申请，来提高每个内存块在分配时的利用率和性能。(同样处理 10kb 内存的申请，返回 10kb 的 chunk 和返回 20kb 的 chunk 之间肯定有区别)

采用了低地址优先的分配策略，进一步降低了内存碎片率。(使用红黑树记录了地址排序，总是从低地址开始分配，使高地址的内存更整块)

（2）锁的颗粒度：jemalloc 在大部分场景下几乎是无锁的。

每个线程都拥有动态伸缩的缓存 tcache，在小内存操作时是无锁的。

大部分的线程都会被绑定到专属的 arena 上，使其操作无锁化（类似于 JVM 的偏向锁）。即使多个 线程共享一个 arena，也会在 arena 内部细化为局部锁，而不是直接使用全局锁。

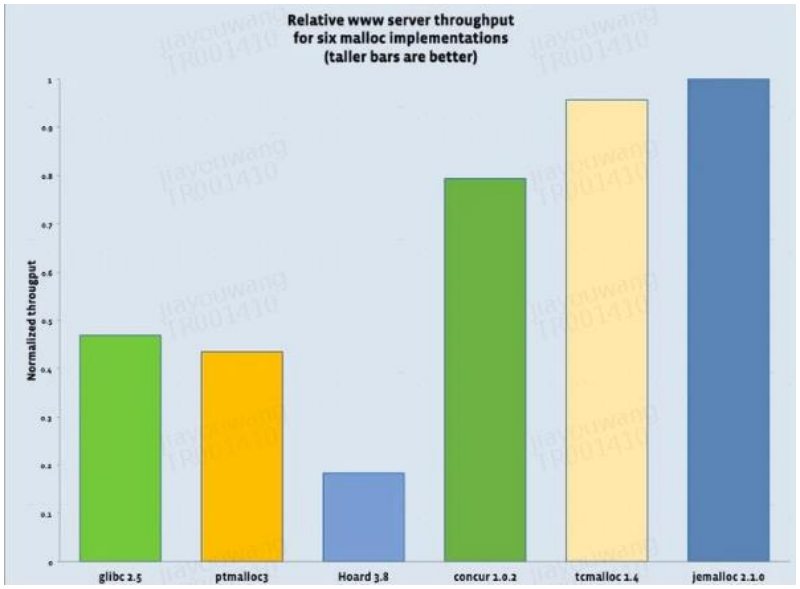
（3）内存回收：除了类似于 ptmalloc 的回收机制外，jemalloc 还有两种机制。

当发现某个 chunk 全部都是脏页后，会直接释放整个 chunk。

当脏页数量超过某个阈值的时候，进行主动的 purge 操作。

（4）额外开销：仅仅占用约 2%的额外内存，用于存储一些 meta 信息。

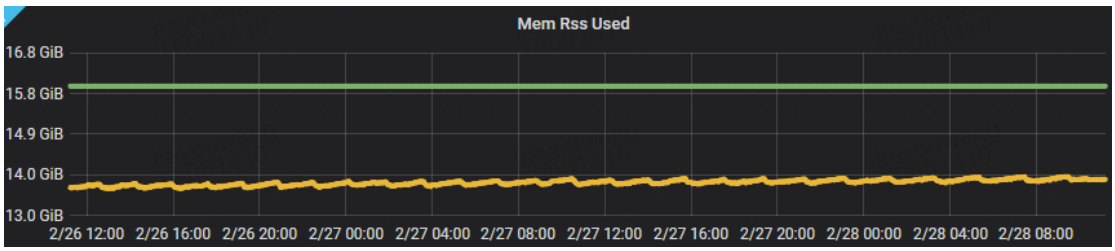
（5）工具链：jemalloc 有完善的内存分析工具，可以更好的定位溢出和泄露问题。



3.5 迁移和收益

对于简单的性能测试，手动安装 jemalloc 非常容易，甚至不需要重新编译代码，直接在一台正常运行的机器上安装好 jemalloc 后，修改 tomcat 的 sh 文件中将 LD_PRELOAD 变量指定为对应的 so 文件覆盖 glibc 动态库并重启 tomcat 即可。而后续容器部署也只需要在 dockerfile 中自定义数行代码模拟上述操作，然后构建并上传自定义镜像便能完成。

目前查询服务已经在 jemalloc 上生产运行了数个月，至今还没有观察到再次出现堆外溢出的问题；同时 RSS 的波动非常稳定，即使遇到流量高峰也不会出现内存尖刺，可以保持良好的响应时间和稳定。



从实际的情况来看，jemalloc 与 ptmalloc 相比主要有以下收益：

- 从运维方面来看，集群为了方便调度，一般会限制几个预设的容器配置以供选择。在资源相对紧张的情况下，jemalloc 可以使得应用整体的部署更加灵活，而使用默认的 ptmalloc 则会被迫将容器配置向上升级，否则就需要额外对特殊配置进行审批和调度，这样不但会造成不必要的资源浪费，同时在流量尖峰时也难以对集群进行调度和扩容。

- 在成本方面，从测试结果出发，仅仅使用 jemalloc 本身就能比 ptmalloc 在每台机器上节省 1-1.5G 的堆外内存，虽然在单机上可能不够显著，但是推广到整个云的范围时收益应该是非常可观的。

- 性能上，jemalloc 的内存回收和多线程机制更加高效和智能化，对低配置机器更加友好，能大大加强内存资源紧张的机器上服务的鲁班性，同时对 IO、GC、类加载等多线程 native 操作有较大的优化。

- 从迁移角度看，迁移到 jemalloc 几乎是无成本的操作，仅仅需要简单的镜像自定义和一定的灰度测试，就可以完成优化。

故综合来看，jemalloc 的收益相比于成本大得多的，有一定的分享和推广的意义。

四、结语

本文相对完整的记述了酒店查询服务在轻量化中的一次优化过程，希望其中的经验和过程能对读者有所帮助。然而，对于应用的优化过程是一个从猜想到验证的循环。在有了可能的猜测和方向之后，比起反复的调研，更重要的则是不断向着落地验证去推进。虽然这些经验有一些普适性，但是由于应用之间各有不同，仍然需要读者根据实际情况亲手试验后，才能最终确定是否有借鉴意义。

降低复杂度提升效率，DDD 在携程用车/租车订单系统重构中的实践

【作者简介】小白龙，携程资深后端开发工程师，关注架构落地、研发效能领域。

随着历史业务不断迭代和业务场景越来越复杂，携程用车、租车（简称两车）面临历史技术债和系统复杂度越来越高带来的理解、维护、迭代困难等问题，我们开始寻求如何更有效的降低复杂度和提升效率的方法。

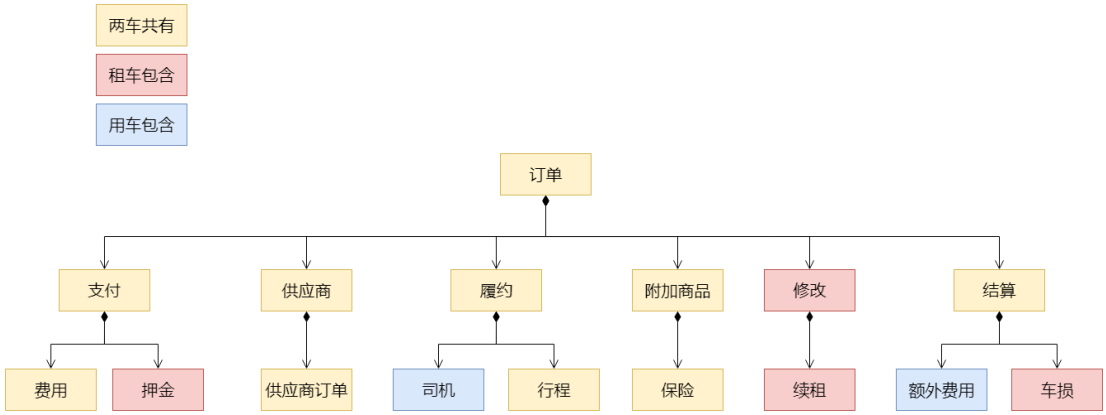
本文描述了两车如何利用 DDD（Domain-driven Design，领域驱动设计）方法论降低系统复杂度以及在重构历史系统中的取舍和思考。对于复杂业务场景下的领域驱动设计具有借鉴意义。

一、案例介绍

携程用车订单相关业务包括接送机、包车、打车这些产线，订单相关的功能包括订单状态管理、支付状态管理、供应商订单状态管理、履约状态管理，其中履约状态中包含司机相关状态，完成订单需要将额外费用结清。

携程租车订单相关功能包括订单状态管理、支付状态管理、押金扣款记录、供应商订单状态管理、履约状态管理，其中履约状态主要是取车和还车相关状态。

订单和相关实体如下图所示：



二、问题分析

由于两车业务存在一些差异，为了读者更容易理解，因此将抽取共性问题来说明。

2.1 沟通困难

关于沟通困难，我们发现整个开发过程中，沟通实际上是一个非常消耗时间的事情，需求方

需要和产品沟通，产品要和研发人员沟通，研发开发过程中发现一些忽略的细节需要产品确认，来回之间耗费了大量时间。如果是跨团队沟通，这样的问题会更加复杂，以下总结了一些常见的场景：

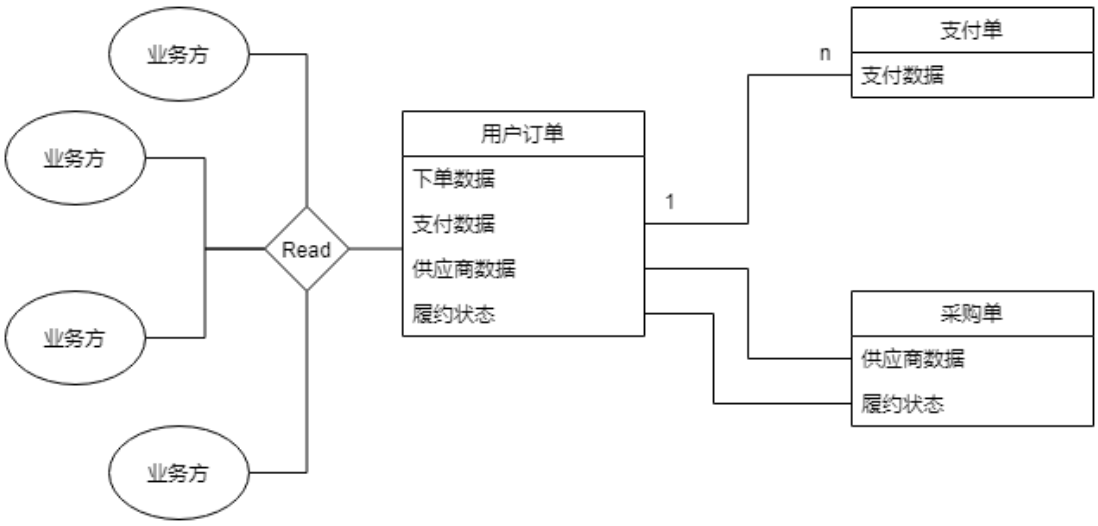
- 产品不关心研发的实现，但是觉得需求很简单或者很复杂。
- 研发开发过程中发现一些忽略的细节需要产品确认，产品要找需求方确认。
- 历史逻辑没人知道，需求评审的时候无法发现问题，做到最后发现有问题。
- 跨团队之间不了解对方的业务，需要反复沟通确认。
- 遇到同一个名词不同的理解导致无效沟通。
- 一个需求到底该哪个域来实现是我们在实践中经常反复探讨的问题。
- ...

例如订单和供应商订单在不同的团队内都叫订单，在沟通中针对“订单”的讨论就会产生歧义。

2.2 业务边界不清晰

设计之初，订单被各调用方当作了对外输出的数据源头，数据需求方只要调订单详情即可获取全量数据，这为以后订单的迭代带来了相当大的隐患。订单在自己的业务模型中加入大量不涉及自身业务的冗余字段，在系统的演进过程中，由于无脑插入他方业务字段使得订单自己也要维护相关的逻辑（解释和修改），导致各方对订单的耦合日益加深，导致订单服务的发布变成高风险行为，甚至一个无关订单业务的相关字段修改也可能导致系统故障。

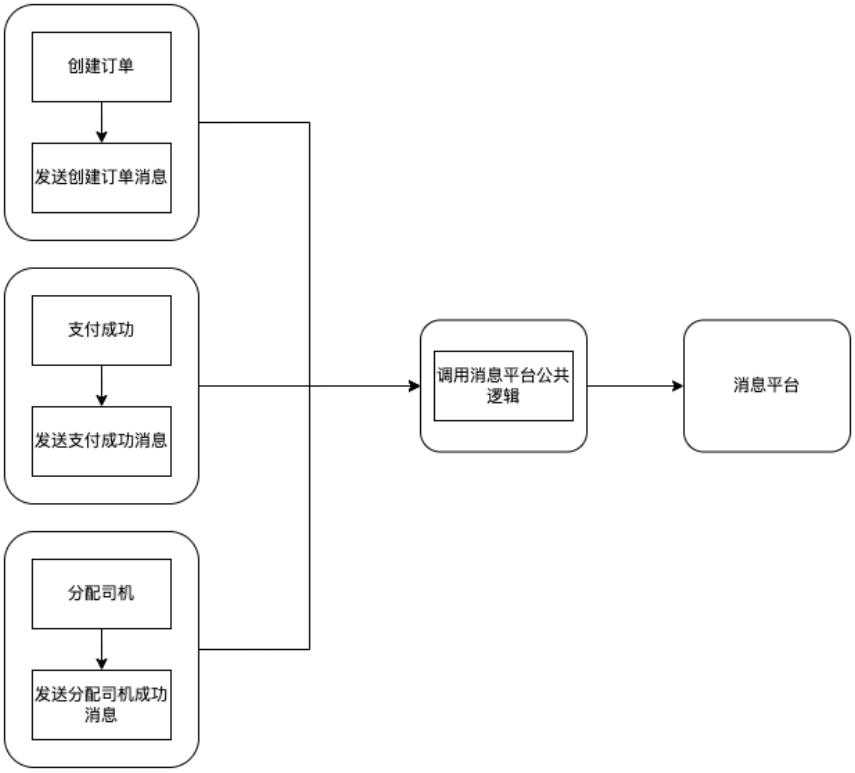
例如订单上关于供应商的相关数据，用户订单有一份，采购订单也有一份，当采购要修改供应商的相关逻辑时要用户订单也一起修改，而用户订单必须排查和推动相关使用到这个字段的业务方切换替代方案。



2.3 面对业务变化修改困难

随着历史业务迭代，订单中耦合了许多非订单关注的业务逻辑。例如历史上给用户发消息通知是根据用户订单状态变化触发的，由于和通知平台交互，因此订单要提供通知相关的所有

参数，等于订单依赖通知相关的模版，明显存在核心依赖非核心的问题。而此时如果我们提出需求，要对于某些通知平台发送失败的消息进行重发，逻辑似乎也只能做到订单上，不论怎么看都很不优雅。



三、解决方案

3.1 回归业务本质——挖掘愿景

为了解决业务归属问题和明确系统发展方向，避免将资源投入那些非核心的功能，我们需要明确当前项目它是什么，目标是什么。因此我们需要为系统准备一份愿景，它将指导我们在未来的迭代中不迷失方向。这个愿景相当于我们的产品定位，是我们的系统和其它系统不同之处，也是当前系统的边界。

愿景就像手电筒中发出的光，在光暗之间是我们系统的边界，系统的未来也在光的方向中。

输出一个愿景说明有很多方式，为了简化落地的门槛，我们采取麦肯锡“电梯演讲”的方式，围绕机会、挑战、优势、劣势给出一组结果，由领域专家和开发团队一起进行头脑风暴，实际上这也是 DDD 统一语言的开始，我们必须从愿景开始就达成一致。

产品名称	用户订单管理系统
产品品类	一个查看和管理订单平台
描述目标客户或利益相关人的需求或机会	1、定后履约的查看和管理 2、流畅：每个用户有操作需求的节点高度自动化，无需人工处理 3、贴心：想用户所想，在合适的时间告诉用户合适的事情 4、无忧：减少用户的焦虑，让用户放心的使用我们的服务
阐释产品能够带来的关键价值（或者说购买的理由）	为用户提供订单流程管理。
与竞争产品的不同之处	

友情提醒

Eric Even 在他的书中曾提到过一种模式：领域愿景描述（Domain Vision Statement）

“由于一开始项目的模型通常不存在，但是需求是早已定下的重点，为了我们在后续阶段清楚了解系统的价值，以价值作为我们的导向。”

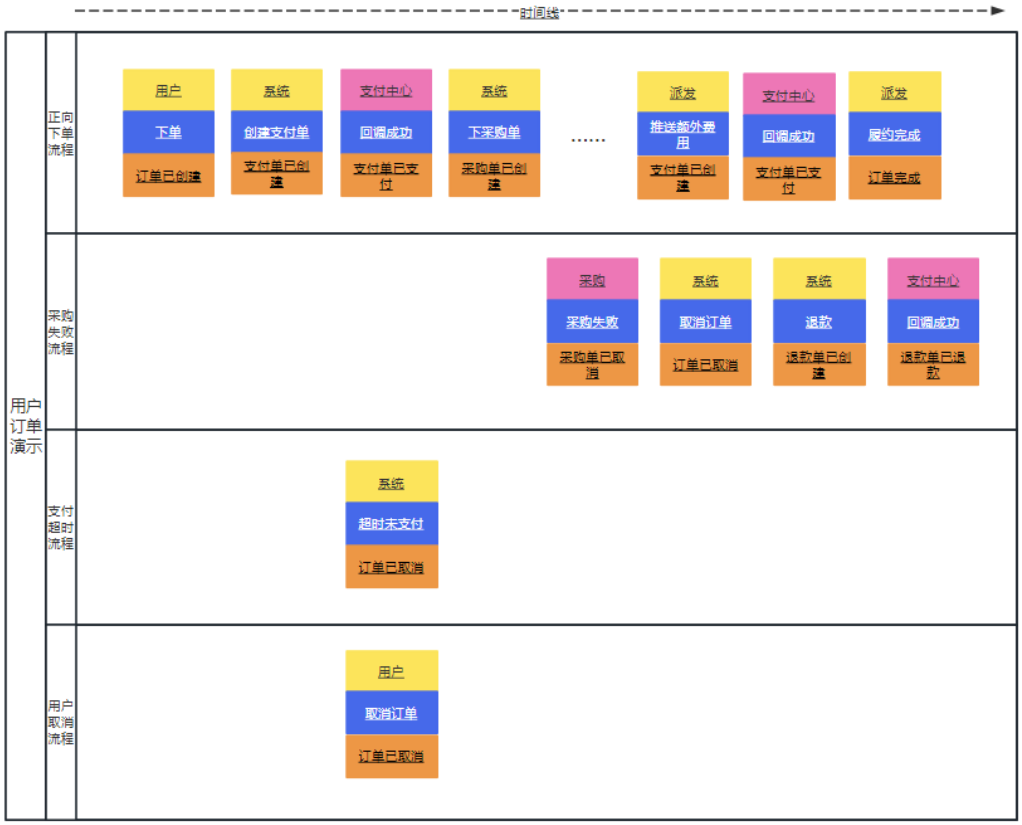
我们在研究领域愿景描述时发现要写出一份合格的文档并不容易，因为它缺乏明确的规范和套路，Eric 也只是给了我们几个案例体会，不得不说虽然写出来容易，但是要做到合格还是有门槛的。因此我们退一步，回到 Eric 说的愿景说明来：“很多项目团队都会编写‘愿景说明’以便管理。最好的愿景说明会展示出应用程序为组织带来的具体价值。”

3.2 高效沟通——利用事件风暴统一语言

说到统一语言，最经典的例子应该是传话游戏，一句话从最初的人口中说出，经历中间多人转述，最后可能完全变成另一种意思。

为了快速实现统一语言，我们在订单重构中花了比较多的时间进行事件风暴。事件风暴有以下几点优势：

- 事件风暴围绕业务流程进行讨论，使在场的每一个人都通过多条流程深入了解业务实体的变化。
- 事件风暴聚集了“领域专家”，产品、开发、测试等，本质也是一场集合集体智慧的头脑风暴，所有人在事件风暴中达成业务共识。
- 事件风暴集合了所有人的领域知识，同样是一场领域知识的分享会。



原本事件风暴是以工作坊的方式在线下组织，这样大家的参与感更强烈。但是由于成本和线上办公的兴起，我们在在线工作坊的实践会更多一些。这里推荐两个工具，一个是行知蜂（BeeArt），另一个是可画（canva），都支持多人在线协作。

事件风暴其实非常简单，就是业务流程+业务用例，将业务流程横向展开，通过用例将业务中的名词状态变化一一列举。其中色块的大小和颜色可以参考 www.eventstorming.com，但是我认为只要能够统一大家的认知，颜色是次要的。

经过我们的尝试，先列举业务中单据的状态变化，后补全触发状态变化的动作和角色效率会更高一些。关键是将大家认知中的不同事物相同名词、不同名词相同事物识别出来，利于后续建模。

通过事件风暴，我们主要关注以下几种情况：

- 沟通中那些脱离当前领域就难以理解的词汇；
- 相同名词，含义不同的；
- 名词不同，含义相同的。

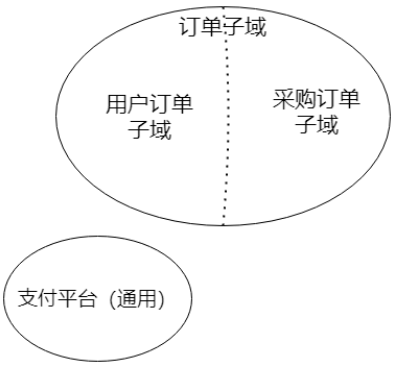
将以上三种情况涉及的名词动词总结成统一语言表，特别是第三种情况恰恰是我们划分限界上下文的关键依据。例如我们在聊支付单时发现存在两种支付单，一个是包含我们业务的支付单，它需要记录当前支付的场景并包含一定的业务规则；另一个是支付平台的支付单，每次支付都会生成一个支付单，它可以认为是和更抽象的订单相关（例如会员订单、优惠券订单）。

于是我们提取了费项记录这个概念，表达一笔订单可以有多个费项记录，用于区分我们的支付单和支付中台的支付单之间的差别。

3.3 自上而下细化边界——子域划分

传统面向过程的开发方法面对复杂系统通常会采用 DFD 数据流图的方式进行拆分，在 DDD 中则是提出了子域的概念。我们总是会听到领域（Domain）和子域（Sub Domain），不论是 Eric 的 DDD 还是 IDDD 中都大量使用这些概念，但是我们会发现他们并未向我们解释清楚子域是如何划分而来的。

对于一个已有的系统而言，我们可以根据康威定律得出：团队边界=系统边界，因此可以认为每个团队负责的部分就是天然的子域。由于目前订单团队本就分为用户订单组和采购派发组，因此我们可以初步得出一个领域划分：

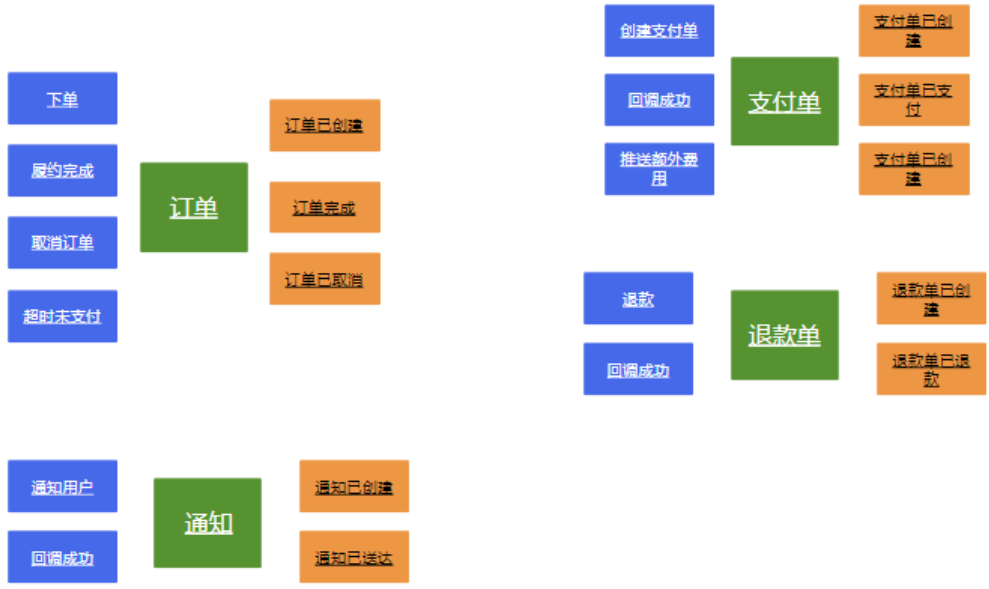


此时我们根据愿景，可以明确两个子域各自的职责：用户订单子域负责提供用户订单流程的查看和管理，并且负责在需要的环节主动通知用户；采购订单子域则负责真正定后履约流程的流转，包括供应商和行前行中行后的状态更新。

最后支付使用的是携程金融的能力，由于支付平台的能力在携程内部是统一的，因此我们认为支付平台属于通用域。

3.4 自下而上抽象概念——限界上下文

领域的概念相对而言还是模糊的，因此 Eric 提出了 DDD 中最重要的概念：限界上下文。而限界上下文并非凭空而来，而是需要对我们在事件风暴中得到的名词进行归纳而来。

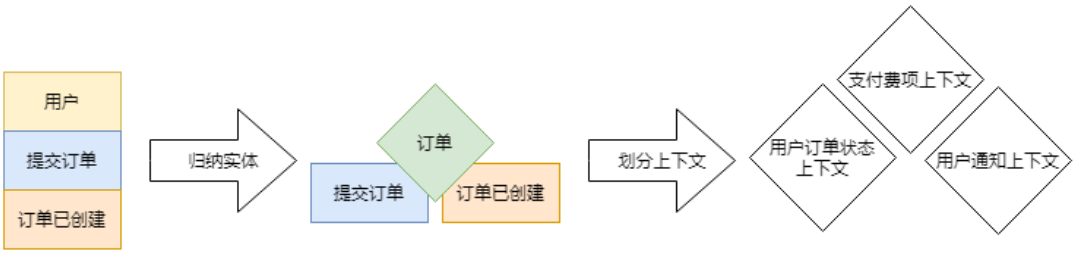


首先我们列举了用户订单域的各种用例，包括下用户单、支付订单、修改订单、取消订单等。

通过建模法归纳模型，例如在订单流程中我们存在多场景的支付，同时又依赖支付平台的支付单，因此我们得到了维护支付单状态的支付费项记录，它既维护了支付单相关的信息，也维护了当前订单系统内关于支付的业务逻辑。

最后我们根据业务相关性对得出的实体进行归纳，结合我们的愿景得出三个上下文，分别是：

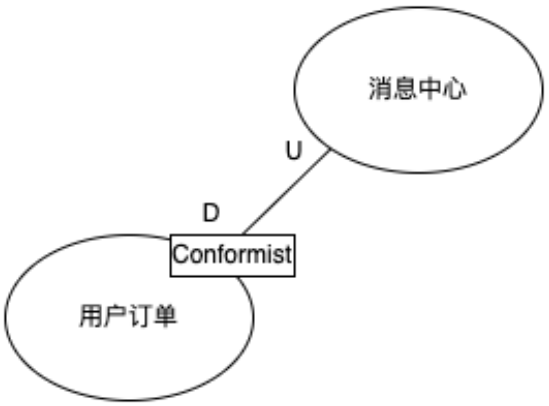
- 用户订单状态上下文：负责管理用户订单状态管理；
- 支付费项上下文：负责订单支付相关状态管理；
- 用户通知上下文：负责对用户进行多种方式的通知。



3.5 挖掘业务变化的瓶颈——上下文依赖关系

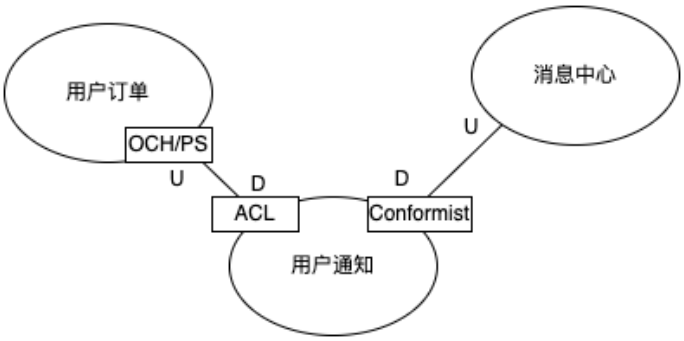
实际上限界上下文可以拆到很细的粒度，但是我们应该遵循“奥康姆剃刀”的规则，尽量设置

合理的数量，拆分的要有理有据。我们可以先看看原来的系统上下文依赖关系：



根据 Eric 对上下文关系的总结，我们可以得出消息中心作为携程的消息中台，不会为了某个业务线做特殊逻辑，因此是很明显的遵奉者（Conformist）。此时消息相关的处理耦合在订单内部，如果发送消息没有业务逻辑那么采取防腐层（ACL）的方式是比较常见的。

由于我们已经识别到用户通知存在业务逻辑，因此订单直接和消息中心交互显得奇怪，而且订单作为核心域，本来就应该尽量不依赖其它域，对此我们进行了如下设计：



这样用户订单上下文更加内聚，而用户通知也更加易于迭代。

四、收益总结

4.1 业务逻辑耦合降低

通过上下文拆分和职责的明确，由各领域维护自己的数据和领域知识，使得订单不再维护这些字段，而由数据写入的业务方去维护，后续有和订单无关的业务逻辑变更时订单无需改动。

4.2 团队效率提高

随着上下文拆分和康威定律的应用，各团队职责和各自的领域形成映射，过去由于团队职责划分不清，经常为某功能谁做来争论不休的问题也得到了解决。

4.3 性能和稳定性提高

通过上下文拆分后，订单实体从之前的 780 多个字段简化到 200 多个字段，大大降低了订单的维护成本，存储数据量减少，原来的业务逻辑也由每个写入方自己进行维护，接口性能 p95 写由 68ms 优化到 12ms ， 读从 63ms 降低到 5ms。

4.4 数据一致性

由于过去业务方写入数据到订单可能由于网络抖动等原因写入失败或业务方写错数据导致修复数据需要两边一起改，现在业务方将数据存放在自己的领域内，不再存入订单，避免了数据不一致和字段写错等问题的产生。

4.5 人力成本大幅下降

产研沟通涉及的相关方大量减少，链路缩短。刨去原本因为业务逻辑耦合导致订单跟着修改的人力成本，整体人力成本小项目下降 70%，大项目更是下降 80%。

五、遇到的问题和方案探索

落地 DDD 实际上是一个非常困难的过程，我们必须面对缺乏领域专家，业务需求多且急，团队对 DDD 理解不深等诸多问题。对此我们总结出以下几点经验：

5.1 领域专家难寻

领域专家是 DDD 中最重要的角色之一，没有领域专家我们就无法获取知识，就没有后续的建模等等。但是实际工作中要寻找一个严格意义上的领域专家是困难且成本高昂的，因此我们需要寻求一些其它方式曲线救国，例如该领域的资深研发，资深 QA 等，同时我们两车还采取互相借鉴的方式，虽然业务不完全相同，但是领域上也有互通之处。

5.2 业务需求多且急

实际工作中我们经常忙于各种业务项目，关键是还很急。这就很容易导致我们没办法专注于 DDD 改造，怎么办呢？我们的方案是在有时间的时候把方向定下来，提前进行设计，然后在做业务项目时将这些设计逐步进行实现。

5.3 团队对 DDD 理解不深

为了提高团队对 DDD 的理解，我们专门成立了 DDD 培训小组，将我们的一些落地经验整理成规范和最佳实践。同时在落地时由培训小组的同学进行把关，避免大家走弯路。

以上就是此次分享的全部内容，如果后续大家有什么疑问可以在下方留言，如果后续有机会我们会在疑惑较多的点进行再次分享。希望大家通过这篇文章得到一些想要的收获!

携程中转交通方案拼接性能优化

【作者简介】简言，携程后端开发经理，关注技术架构、性能优化、交通规划等领域。

一、背景介绍

由于交通规划和运力资源的限制，用户查询的两地之间可能没有直达交通，或者在重大节假日时，直达交通都已售罄。不过，通过火车、飞机、汽车、船舶等两程或多程中转的方式，用户仍然可以到达目的地。此外，中转交通有时在价格和耗时方面更具有优势。例如，对于从上海到运城，通过火车中转可能比直达火车更加快捷和便宜。

图 1 携程火车中转交通列表

在提供中转交通方案时，很重要的一个环节是将两程或多程的火车、飞机、汽车、船舶等拼接起来组成可行的中转方案。而中转交通拼接的第一个难点是拼接空间极大，仅考虑上海做中转城市，就可以产生近亿种组合；另一个难点在于对实时性有要求，因为产线数据随时变化，需要不断地查询火车、飞机、汽车、船舶的数据。中转交通拼接需要大量的计算资源和 IO 开销，因此，对其性能进行优化显得尤为重要。

本文将结合实例，介绍在中转交通拼接性能优化过程中所遵循的原则、分析和优化方法，旨在为读者提供有价值的参考和启示。

二、优化原则

性能优化需要在满足业务需求的前提下，在各种资源和约束条件下去平衡和取舍，遵循一些大的原则有助于消除不确定性，去逼近解决问题的最优解。具体来说，中转交通拼接优化过程中主要遵循以下三个原则：

2.1 性能优化是手段而不是目的

虽然本文是关于性能优化的，但仍需要在最开始强调：不要为了优化而优化。满足业务需求的方式有很多，性能优化只是其中一种。有时候问题非常复杂，限制也很多，在不显著影响用户体验的前提下，通过放宽限制或采用其他流程来减少对用户的影响，这也是解决性能问题的好方法。在软件开发中，存在许多通过牺牲少量性能来实现大幅降低成本的事例。例如，在 Redis 中用于基数统计（去重）的 HyperLogLog 算法，它在标准误差为 0.81%的前提下，只需要 12K 空间就能够统计 264 的数据。

回到问题本身，由于需要频繁地查询产线数据，并且进行海量的拼接操作，那么如果要求每个用户查询时都立刻返回最新鲜的中转方案，成本将会非常高。为了降低成本，需要在响应时间和数据新鲜度之间进行平衡。经过仔细考虑选择可以接受分钟级的数据不一致，对于一些冷门线路和日期，可能在首次查询时没有好的中转方案，此时引导用户重新刷新页面即可。

2.2 不正确的优化是万恶之源

Donald Knuth 在《Structured Programming With Go To Statements》中提到：“程序员们浪费大量的时间去思考、担忧非关键路径的性能，而尝试优化这部分性能，对整体代码的调试和维护都有非常严重的负面影响，因此 97%的情况，我们应该忘记小的优化点”。简而言之，在没有发现真正的性能问题之前，在代码层面过度炫技式的优化，不仅不会提高性能，反而可能会导致更多的错误。然而作者同样也强调：“对于剩下关键的 3%，我们也不要错过优化的机会”。因此，需要时刻关注性能问题，不做会影响性能的决策，并在必要的时候做正确的优化。

2.3 量化分析性能，明确优化方向

正如前一节所述，在进行优化之前，首先要量化性能并找出瓶颈，这样优化的才更有针对性。量化分析性能可以借助耗时监控、Profiler 性能分析工具、Benchmark 基准测试工具等，重点关注耗时特别长或者执行频率特别高的地方。正如阿姆达尔定律所述:“系统中对某一部件采用更快执行方式所能获得的系统性能改进程度，取决于这种执行方式被使用的频率，或所占总执行时间的比例”。

此外，还需要注意到性能优化是一场持久战。随着业务的不断发展，架构和代码也不停地变化，因此更需要持续量化性能，不断分析瓶颈和评估优化效果。

三、性能分析之路

3.1 梳理业务流程

在性能分析之前，首先要梳理业务流程。中转交通方案拼接主要包含以下四个步骤：

a. 加载线路图，如北京经南京中转到上海，只考虑线路本身的信息，与具体的班次无关；
b. 查火车、飞机、汽车、船舶的产线数据，包括出发时间、到达时间、出发站、到达站、价格和余票信息等；
c. 拼接出所有可行的中转交通方案，主要是考虑换乘时间不能过短，以免无法完成换乘；同时也不宜过长，以免等待太久。拼接出可行的方案后，还需要完善业务字段，例如总价格、总耗时和换乘信息等；
d. 根据一定的规则，从拼接出的所有可行中转方案中筛选出一些用户可能感兴趣的方案。

3.2 量化分析性能

（1）增加耗时监控

耗时监控是一种最直观的从宏观角度观察各个阶段耗时情况的手段。它不仅可以查看业务流程各阶段的耗时值与耗时占比，还可以长期观察耗时变化趋势。

耗时监控可以借助公司内部的指标监控告警系统，在中转交通方案拼接的主要流程中增加耗时打点。这些流程包括加载线路图、查询班次数据并进行拼接、筛选和保存拼接方案等。各个阶段的耗时情况如图 2 所示，可以看到，拼接（含查产线数据）的耗时占比最高，因此成为未来重点优化的目标。

图 2 中转交通拼接耗时监控

（2）Profiler 性能分析

耗时打点可能会侵入业务代码，并对性能产生影响，因此不宜过多，更适合监控主要流程。与之对应的 Profiler 性能分析工具（例如 Async-profiler），可以生成更具体的调用树以及各函数的 CPU 占用比例，从而帮助关键路径和性能瓶颈的分析与定位。

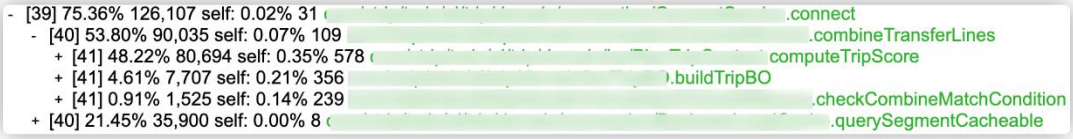


图 3 拼接调用树与 CPU 占比

如图 3 所示，拼接方案（combineTransferLines）占 53.80%，查产线数据（querySegmentCacheable，已使用缓存）占 21.45%。在拼接方案中，计算方案评分(computeTripScore，占 48.22%)、创建方案实体（buildTripBO，占 4.61%）和检查拼接可行性（checkCombineMatchCondition，占 0.91%）是占比最大的三个环节。

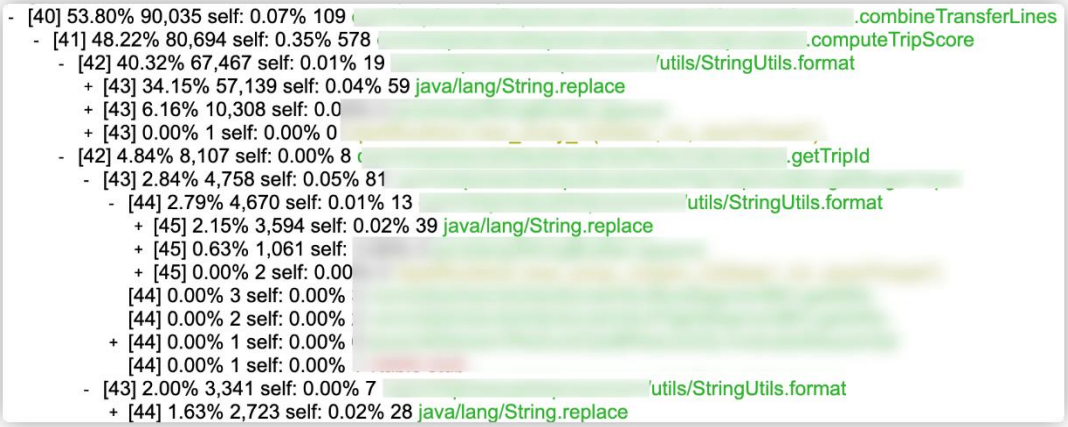


图 4 方案打分调用树和 CPU 占比

继续分析占比最高的计算方案评分（computeTripScore）时，发现主要与自定义的字符串格式化函数（StringUtils.format）有关，包括直接调用（用于展示方案评分细节），以及通过 getTripId 间接调用（用于生成方案的 ID）。自定义的 StringUtils.format 中占比最高的是 java/lang/String.replace，Java 8 原生的字符串替换是通过正则实现的，效率比较低（这一问题在 Java9 之后已经改进了）。

```
// 计算方案评分(computeTripScore) 中调用的 StringUtils.format 代码示例
StringUtils.format("AAAA-{0},BBBB-{1},CCCC-{2},DDDD-{3},EEEE-{4},FFFF-{5},GGGG-
{6},HHHH-{7},IIII-{8},JJJJ-{9}",
                aaaa, bbbb, cccc, dddd, eeee, ffff, gggg, hhhh, iiii, jjjj)

// getTripId 中调用 StringUtils.format 代码示例
```

```
StringUtils.format("{0}_{1}_{2}_{3}_{4}_{5}_{6}", aaaa, bbbb, cccc, dddd, eeee, ffff)

// 通过 Java replace 实现的自定义 format 函数
public static String format(String template, Object... parameters) {
    for (int i = 0; i < parameters.length; i++) {
        template = template.replace("{" + i + "}", parameters[i] + "");
    }
    return template;
}
```

（3）Benchmark 基准测试

借助 Benchmark 基准测试工具可以更准确地测量代码的执行时间。在表 1 中，我们通过 JMH（Java Microbenchmark Harness）对三种字符串格式化方法和一种字符串拼接方法进行耗时测试。测试结果表明，使用 Java8 的 replace 方法实现的字符串格式化性能最差，而使用 Apache 的字符串拼接函数性能最佳。

表 1 字符串格式化与拼接性能对比

实现	执行 1000 次平均耗时(us)
使用 Java8 的 replace 实现的 StringUtils.format	1988.982
使用 Apache replace 实现的 StringUtils.format	656.537
Java8 自带 String.format	1417.474
Apache 的 StringUtils.join	116.812

四、性能优化之路

通过以上的性能分析，我们发现拼接和查询产线数据是性能瓶颈，字符串格式化影响尤其大。因此，我们将致力于优化这些部分，以提高性能表现。

4.1 优化代码逻辑

优化代码逻辑是最简单且性价比最高的方法，可以是修正有问题的代码或替换为更好的实现。不同的实现，哪怕减上几纳秒，累加起来也是很可观的。借助一些经典算法或数据结构（如快速排序、红黑树等）可以在时间和空间复杂度方面带来显著优势。回到中转交通方案拼接性能优化本身，优化的代码逻辑主要包括：

（1）优化字符串拼接性能

如前面的 JMH 的结果所示，自定义的字符串格式化函数性能最差，因此作为重点优化目标。优化前后的对比如下所示：

```
// 优化前，通过 Java replace 实现的 format 函数
public static String format(String template, Object... parameters) {
    for (int i = 0; i < parameters.length; i++) {
        template = template.replace("{" + i + "}", parameters[i] + "");
    }
    return template;
}
// 优化后，通过 Apache replace 实现的 format 函数
public static String format(String template, Object... parameters) {
    for (int i = 0; i < parameters.length; i++) {
        String temp = new StringBuilder().append('{').append(i).append('}').toString();
        template = org.apache.commons.lang3.StringUtils.replace(template, temp,
String.valueOf(parameters[i]));
    }
    return template;
}
```

根据 JMH 的测试结果，即使是优化后的格式化函数，其性能也不是最优的。在不显著影响可读性的前提下，应尽量使用性能更优的 StringUtils.join 函数。

```
// 优化前
StringUtils.format("{0}_{1}_{2}_{3}_{4}_{5}_{6}", aaaa, bbbb, cccc, dddd, eeee, ffff)

// 优化后
StringUtils.join("_", aaaa, bbbb, cccc, dddd, eeee, ffff)
```

为进一步提升性能，可以在 computeTripScore 函数中添加一个开关，仅在调试模式下才展示评分细节，这将确保该字符串格式化函数仅在需要时才被调用。

```
if (Config.getBoolean("enable.score.detail", false)) {
    scoreDetail = StringUtils.format("AAAA-{0},BBBB-{1},CCCC-{2},DDDD-{3},EEEE-{4},FFFF-
{5},GGGG-{6},HHHH-{7},IIII-{8},JJJJ-{9}",
            aaaa, bbbb, cccc, dddd, eeee, ffff, gggg, hhhh, iiii, jjjj);
}
```

优化后的 CPU 占比如图 5 所示，此时字符串格式化已经不再是性能瓶颈。

图 5 优化后的拼接调用树和 CPU 占比

（2）增加索引降低拼接时间复杂度

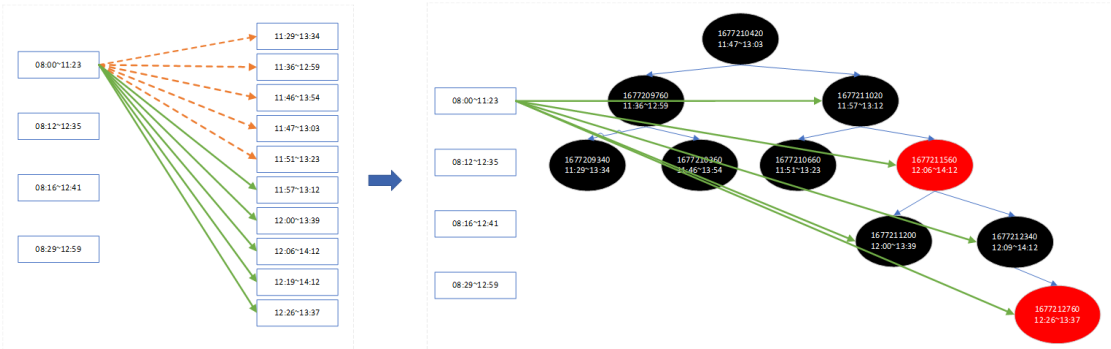


图 6 增加索引降低拼接时间复杂度

在中转拼接过程中，我们需要将第一程每个班次的到达时间与第二程每个班次的出发时间进行比较，以判断中转时间是否过短或过长。为简化说明，假设换乘时间间隔需要满足大于 30 分钟且小于 6 小时。以北京到上海经南京中转的两程火车为例，3 月 9 日北京到南京有 66 个班次，南京到上海有 275 个班次，考虑到隔夜车，还需要算上 3 月 10 日南京到上海的 275 个班次，那么最多需要比较 36300（66*275*2）次。

为避免频繁比较，参考了 MySQL B+树索引的思想，将第二程南京到上海的所有火车班次数据构建成红黑树。其中，树的键为秒级时间戳，例如 2023-03-09 11:29 出发的 G367 键为 1677247680，值为 G367 的班次数据。有了索引树，最多只需要 10 次比较，就可以找到最近的满足最小换乘时间要求的班次。同理，最多需要 10 次比较，就能找到满足最大换乘时间要求的最晚班次。两者之间的所有班次都满足耗时要求，直接进行拼接即可。改进后最多需要比较 1320(66*(10+10))次，约为原来的 1/27.5。

（3）使用多路归并求 Top-K 算法

在筛选方案时，会存在以下场景：有多个中转点，每个中转点都有数百个得分较高的方案（内部已按得分由高到低排序，通过小根堆实现）。最终需要将这些方案合并，并从中筛选出得分最高的 K 个方案。

最简单的方法是使用快速排序将所有的方案排序，然后选取前 K 个，时间复杂度约为 O(nlog2n)。然而，这并没有利用到每个队列自身有序的特点。通过多路归并算法时间复杂度可降为 O(nlog2k)，具体步骤为：

a. 从每个队列中拿出第一个元素（得分最高的方案），放入大根堆中；
b. 从大根堆堆顶拿出最大的元素，放到结果集中；
c. 如果该元素所在的队列还有剩余元素，则将下一个元素加入堆中；
d. 重复步骤 2 和 3，直到结果集中包含 K 个元素或所有的队列都为空。

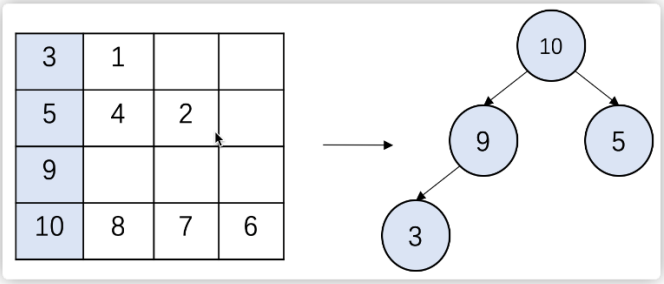


图 7 多路归并求 Top-K 算法

4.2 构建多级缓存

缓存是一种典型的以空间换时间策略，可以缓存数据和计算结果，缓存数据可以提高访问效率，缓存结果避免了重复计算。缓存在带来性能提升的同时，又会引入新的问题：

- 缓存容量有限，需要仔细斟酌数据的加载、更新、失效和替换策略；
- 缓存架构的设计：通常来说内存缓存（如 HashMap、Caffeine 等）性能最高，而 Redis 等分布式缓存次之，RocksDB 相对较慢，容量上限则正好相反，需要仔细选型并搭配使用；
- 缓存不一致问题如何解决，能接受多久的不一致。

在中转交通方案拼接过程中，需要使用大量的基础数据（如车站、行政区域等），以及海量的动态数据（例如班次数据）。综合以上因素并结合中转交通拼接的业务特点，缓存架构做如下设计：

- 基础数据（如车站、行政区域等），因数据量小，变化频率低，全量保存到 HashMap 中，周期全量更新；
- 部分火车、飞机、汽车、船舶的班次数据缓存到 Redis 中，以提高访问效率和稳定性。不同产线采取的缓存策略稍有不同，但总的来说是定时更新与搜索触发更新相结合的方式；
- 一次拼接过程中可能查询数百次产线数据，Redis 毫秒级的延迟累加起来也是非常大的。因此，希望在 Redis 之上再构建一层内存缓存以提高性能。通过分析发现拼接过程中存在非常明显的热点数据，热门日期和线路的查询占比非常高且数量相对有限。因此可以将这部分热点数据保存到内存缓存中，使用 LFU(Least Frequently Used)替换，最终产线数据内存缓存命中率达到 45%以上，相当于降低近一半的 IO 开销。
- 因为可以接受分钟级的数据不一致，所以将拼接结果缓存起来，在有效期内，如果下一个用户查询同一出发日期的相同线路，直接使用缓存数据即可。因为拼接的中转方案数据相对较大，所以将拼接结果保存到 RocksDB 中，虽然性能不如 Redis，但是对于单次查询影响还可以接受。

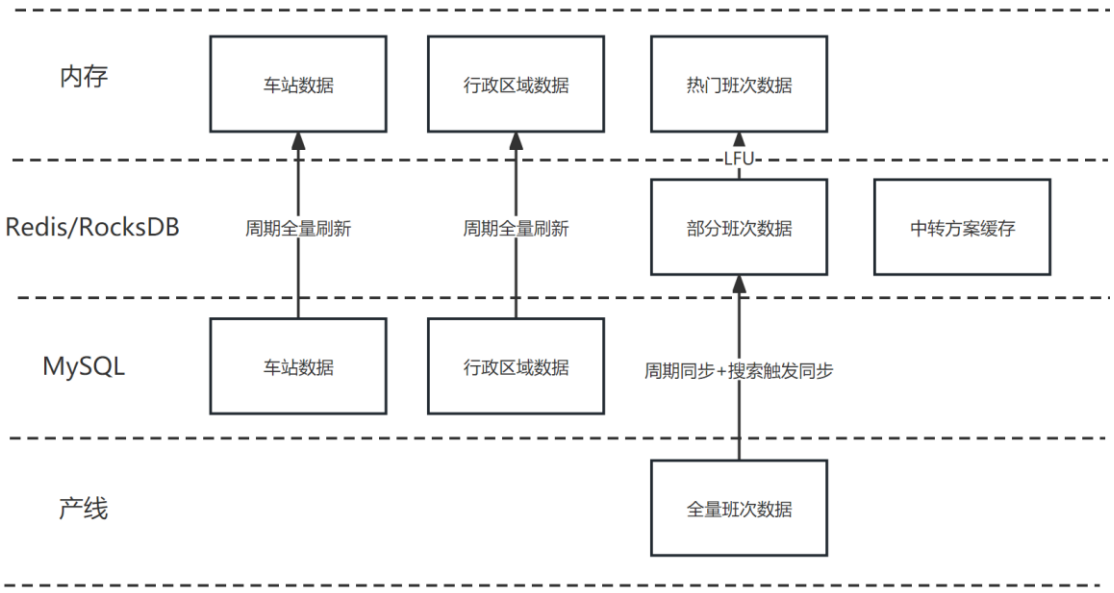


图 8 多级缓存结构

4.3 预处理

尽管理论上可以选择任意城市作为两地的中转点，但实际上大部分中转城市都无法拼接出优质的方案。因此，先通过离线预处理筛选出部分高质量的中转点，从而将求解空间从几千降至数十。相对于动态变化的班次，线路数据是相对固定的，每天计算一次即可。此外离线预处理可以借助大数据技术，处理海量数据，相对对耗时不敏感。

4.4 多线程处理

在一次拼接过程中，需要处理数十条不同中转点的线路。每个线路的拼接是相互独立的，因此可以采用多线程处理，这样可以最大程度地降低处理时间。但受线路班次数量和缓存命中率的影响，不同线路的拼接耗时很难一致。很多时候，分配相同任务数量的两个线程，即使一个线程很快执行完，也要等待另外一个线程执行完才能进行下一步操作。为避免这种情况，这里借助 ForkJoinPool 的 work-stealing 机制。这个机制可以确保每个线程在完自己的任务后，还会分担其他线程未完成的工作，提高并发效率，减少空闲时间。

但是多线程也不是万能的，使用时需要注意：

- 子任务的执行需要相互独立、互不影响。如果存在依赖关系，则需要等待前一个任务执行完才能开始下一个任务，这样会使多线程失去意义；
- CPU 核数决定了并发能力的上限，过多的线程会因频繁切换上下文而降低性能，需要特别关注线程数、CPU 使用率、CPU Throttled time 等指标。

4.5 延迟计算

通过将计算推迟到必要的时刻，可能避免很多多余的开销。例如，在拼接完中转方案后，需要构建方案实体并完善业务字段，这部分也比较消耗资源。而且并非所有拼接的方案都会被

筛选出来，这意味着这部分未被筛选的方案仍然需要耗费计算资源。因此延迟完整方案实体对象的构建，先将拼接过程中的数以万计的方案保存为轻量的中间对象，只对筛选之后的数百个中间对象构建完整的方案实体。

4.6 JVM 优化

中转交通拼接项目是基于 Java 8 的，并使用 G1(Garbage-First)垃圾收集器，部署在 8C8G 机器上。G1 在实现高吞吐量的同时尽可能满足停顿时间的要求，系统架构部门设置的默认参数已经能够适用于大多数场景，通常不需要专门的优化。

但有些线路中转方案过多，导致报文太大，超过 Region 大小的一半(8G 默认 Region 大小是 2M)，导致很多应该进入年轻代的大对象直接进入了老年代，为了避免这种情况，将 Region 大小改为 16M。

五、总结

通过以上的分析和优化，拼接耗时变化如图 9 所示：

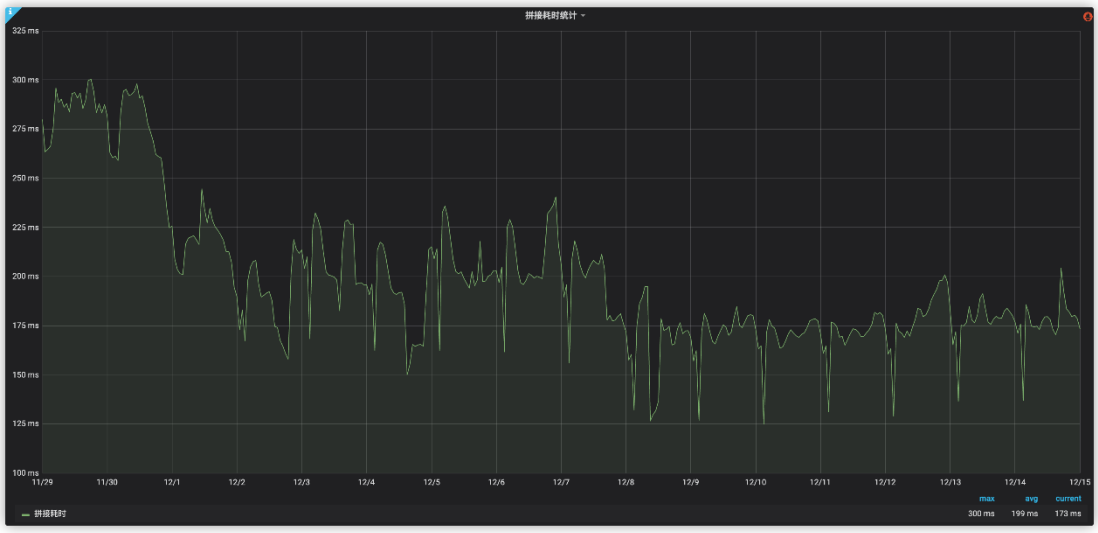


图 9 中转交通方案拼接性能优化效果

虽然每个业务和场景都有各自的特点，性能优化时也需要具体分析。但原理是相通的，依然可以参考本文所述的分析和优化方法。本文所有的分析和优化方法总结如图 10 所示。

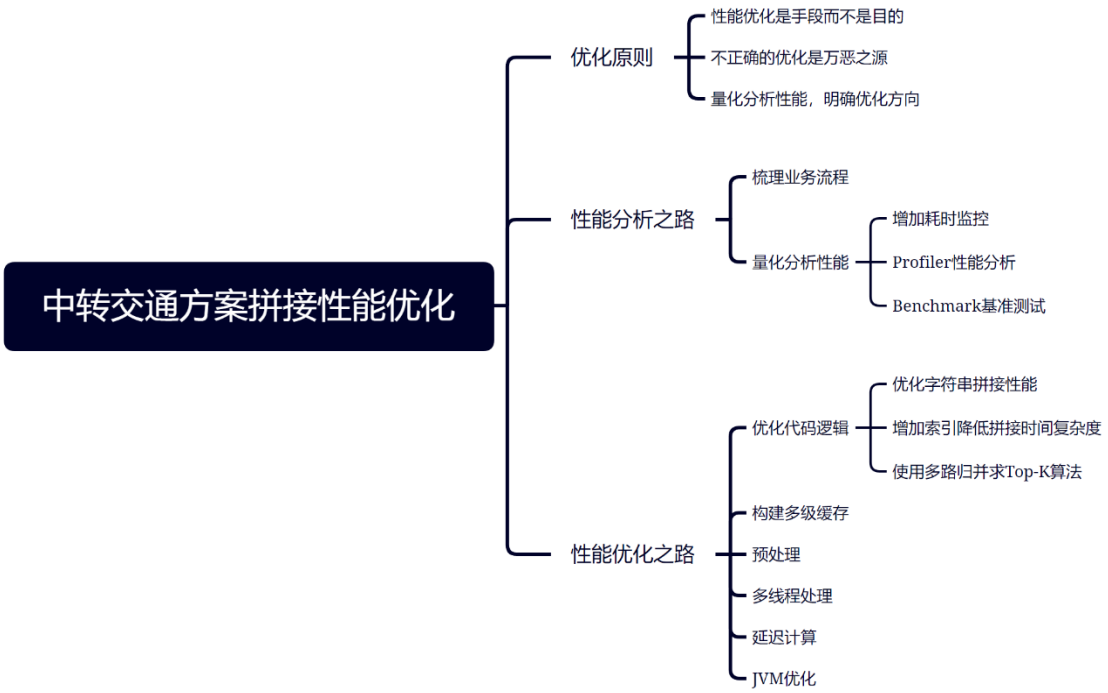


图 10 中转交通方案拼接优化总结

携程 10 个有效降低客户端超时的方法

【作者简介】Wen，携程资深后端开发工程师，专注系统性能、稳定性、交易系统等领域。

一、背景

在现今的信息时代，微服务技术已成为一种重要的解决方案，微服务技术可以使系统的规模和功能变的更加灵活，从而获得更高的可扩展性和可用性。然而，微服务调用中出现的超时问题，却也成为系统可用性的一大隐患。超时会导致客户端的性能下降，甚至可能无法正常工作。本文针对超时问题，提出相关的优化手段，降低微服务调用超时的风险。

1.1 误区

当我们遇到超时或执行慢的问题时，我们往往会认为是依赖方出现了问题。

例如：访问 Redis、DB、 RPC 接口变慢、超时，第一时间找依赖方排查问题，对方反馈的结论是，我这边（服务端）没有问题，请检查一下你那边（客户端）是否有问题。

实际上，性能下降是一个非常复杂的问题，它可能涉及多个方面，包括服务端和客户端。例如：代码质量、硬件资源、网络状况等问题都会导致性能下降，从而引发响应慢、超时等问题。因此，我们需要全面地分析问题，找出影响性能的各种因素。

1.2 分享的目的

本文将详细介绍我们在生产环境中遇到的慢执行和超时等问题，并提出相关的优化手段，通过优化长尾性能，降低变慢或超时的风险，提升系统的稳定性。

二、超时的分类

常见的超时一般有两类：

a. 连接超时（ConnectTimeout）：指建立网络连接所需要的时间超出了设定的等待时间。

b. Socket 超时（SocketTimeout）：指在数据传输过程中，客户端等待服务端响应的时间超出了设定的等待时间。

如下图，①就是连接超时关注的时间，②就是 Socket 超时关注的时间，本文讲解的超时为 Socket 超时。

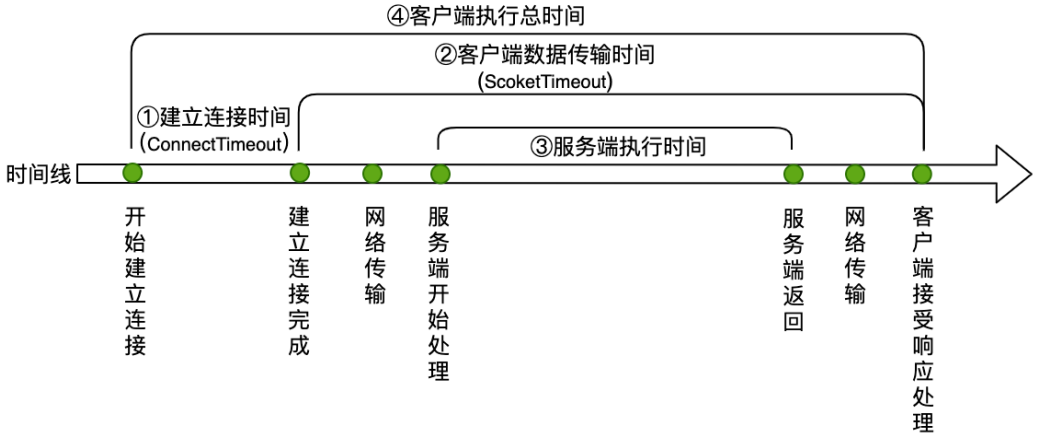


图 1 客户端请求过程

三、超时问题分析与优化

3.1 设置合理的超时时间

根据实际情况设置合理的超时时间，避免因为超时时间设置不合理导致的接口超时。

1）分析

看下客户端设置的超时时间是否合理。比如调用服务端 P99.9 是 100ms ，客户端设置的超时时间是 100ms ，就会有 0.1% 的请求会超时。

2）优化方案

我们在设置超时时间需要综合考虑网络延迟、服务响应时间、GC 等情况。

以门票活动查询引擎为例:

- 核心接口：最小值（ P99.9*3 ，用户可接受的等待时间），核心会影响到订单，在用户可接受范围内尽可能出结果。

- 非核心接口：最小值（ P99.9*1.5，用户可接受的等待时间），非核心不影响订单，不展示也没关系。

3.2 限流

当系统遇到突发流量时，通过限流的方式，控制流量的访问速度，避免系统崩溃或超时。

1）分析

看下超时时间点的请求量是否有突增，比如有某些突然的活动，这个时候应用没有提前扩容，面对突增流量会导致应用负载比较高，从而导致超时问题。

2）优化方案

评估当前应用最大可承载的流量，配置限流，维度可以是单机+集群。

单机限流：在面对突增流量时避免单机崩溃。

集群限流：在有限的资源下提供最大化的服务能力，保证系统稳定性，不会出现崩溃或故障。

3.3 提升缓存命中率

提升缓存命中率，可以提高接口的响应速度，降低接口的响应时间，从而减少超时的发生。

1）分析

分析调用链路，找到慢的地方对其进行优化，提升服务端的响应速度。

如下图所示，很明显可以看到服务端执行时间超过了客户端配置的超时时间 200ms 导致超时。

图 2 客户端调用服务端超时链路

继续分析服务端执行链路，发现是因为缓存没有命中导致的。

图 3 缓存未命中链路

2）优化方案

对于高并发系统来说，常见的是使用缓存来提升性能。

如下图是之前的缓存架构，这种缓存架构有两个风险。

a. 缓存是固定过期的，会导致某个时间大量 key 失效直接击穿到数据库。

b. 主动刷新机制是删除缓存，监听数据库 binlog 消息删除缓存，如果大批量刷数据会导致大量 key 失效。

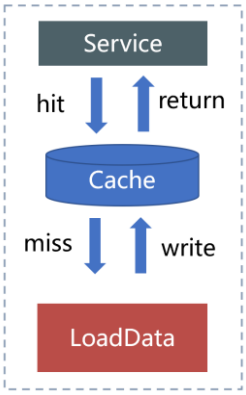


图 4 固定过期+懒加载模式

针对上面的风险我们优化了缓存架构，固定过期改为主动续期缓存，主动监听消息刷新缓存的方案，如下图所示。

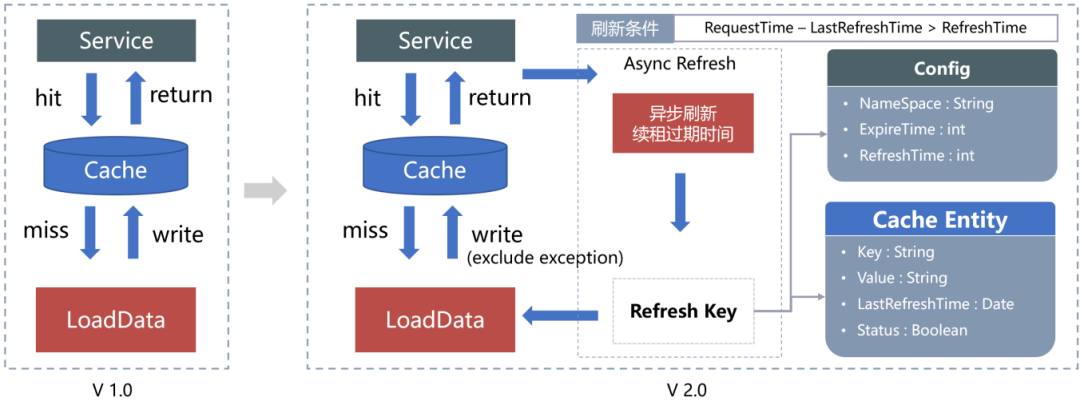


图 5 缓存前后架构对比

3）效果

缓存命中率提升到 98% 以上，接口性能（RT）提升 50% 以上。

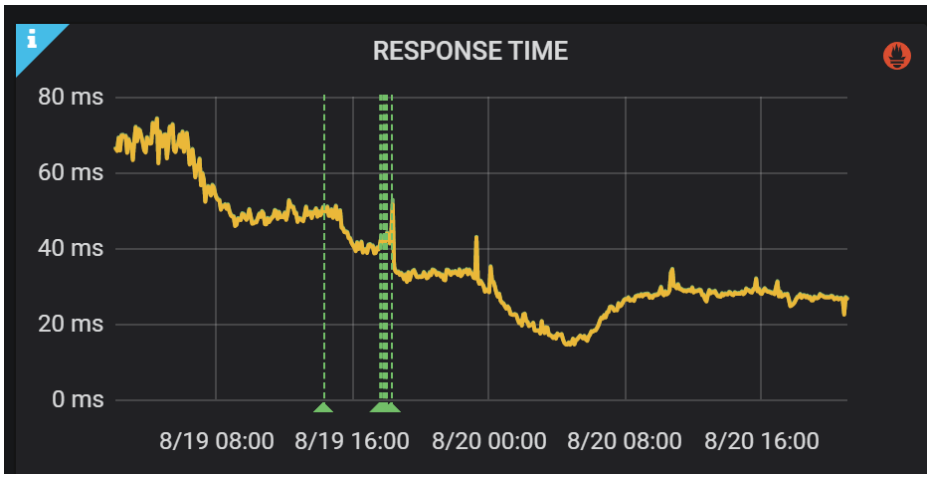


图 6 处理性能提升 50%

这个缓存优化方案在我们团队之前写的一篇文章《1 分钟售票 8 万张！门票抢票背后的技术思考》中有详细的介绍，具体细节这个地方不再展开，有兴趣的同学可以自行阅读。

3.4 优化线程池

减少不合理的线程，降低线程切换带来的超时。

1）分析

a. HTTP 线程数

先看下服务端 HTTP 线程数是否有明显增加，且流量没有增长，要确认 HTTP 线程数增加不是因为流量增长导致的。如下面两张图，流量正常的情况下 HTTP 线程数增加，说明是服务端响应变慢导致，可以确认超时是服务端原因。

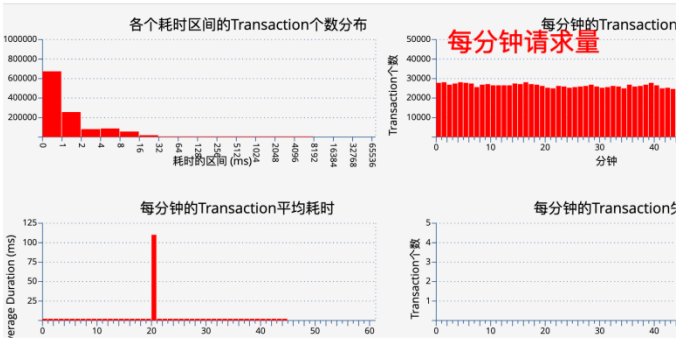


图 7 服务流量平稳

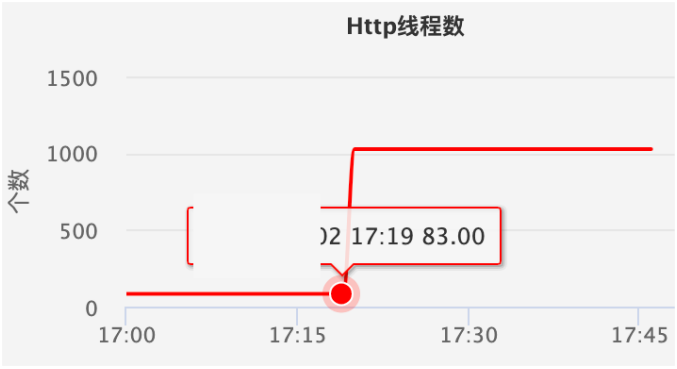


图 8 HTTP 线程数突增

b. 总线程数

再看下总线程数是否有增加（排除 HTTP 线程数），如果有，说明有使用多线程导致线程数量增加。这个时候需要 Dump 下线程，看下哪些线程使用的比较多。

2）解决方案

a. 统一管理线程池：动态配置参数+监控能力

通过工具类封装统一的线程池，提供动态配置参数和线程池监控能力。

- 效果

线程池具备监控能力，如下图是最小值（核心线程数）、最大值（最大线程数）和当前线程池中线程数量的监控，可以参考这个来调整线程池参数。

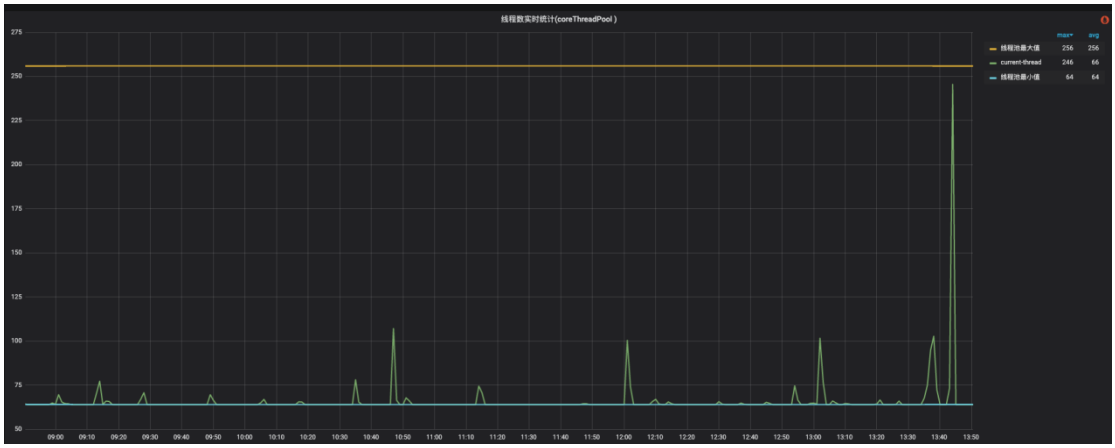


图 9 线程池水位线监控

b. 异步改同步：小于 10ms 的不使用多线程

高并发的场景下线程太多，线程调度时间得不到保障，一次任务需要多个 CPU 时间片，下一次调度的时间无法得到保障。

如下图是一个线程池执行耗时埋点，通过埋点发现 A 在线程池中执行的比较快，平均线和 P95 都在 10ms 以下，没有必要使用线程池，改成同步执行。

接口	Max(ms)	Avg(ms)	P95	P99.9
A	795	2.7	4.7	23.7

图 10 优化前执行耗时

- 效果

接口性能提升明显，平均线从 2.7ms 降低到 1.6ms，P99.9 从 23.7ms 降低到 1.7ms。

之前使用多线程，请求量有波动的时候线程增加比较多，导致线程调度时间得不到保障，体现到 P99.9 就很高。

接口	Max(ms)	Avg(ms)	P95	P99.9
A（优化前-异步）	795	2.7	4.7	23.7
A（优化后-同步）	345	1.6	1.2	1.7

图 11 优化前后耗时对比

另外可以明显看到总线程数也相应减少了很多。

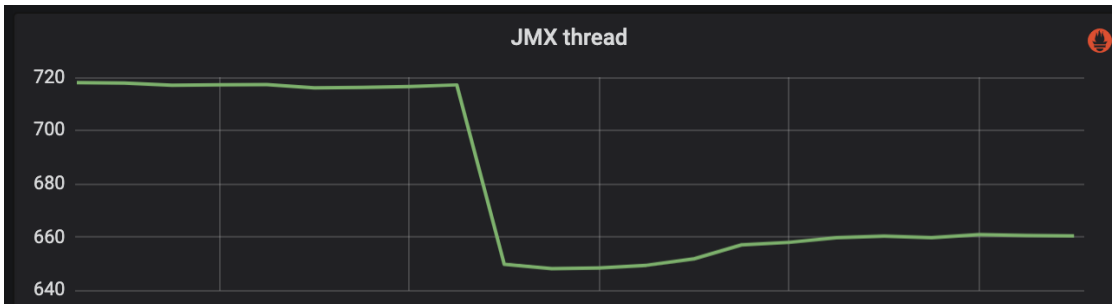


图 12 异步改同步后总线程数减少

3.5 优化 GC

优化 GC，减少 GC 的停顿时间，提高接口的性能。

1 ）分析

首先看超时时间点是否有 Full GC，没有再看下 Yong GC 是否有明显的毛刺，如下图可以看到 3 个明显的毛刺。

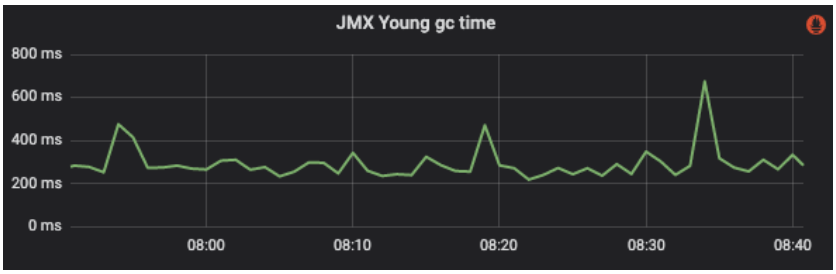


图 13 Yong GC 时间

如果超时的时间点（如下图）可以对应上 GC 毛刺时间点，那可以确认问题是由于 Yong GC 导致。

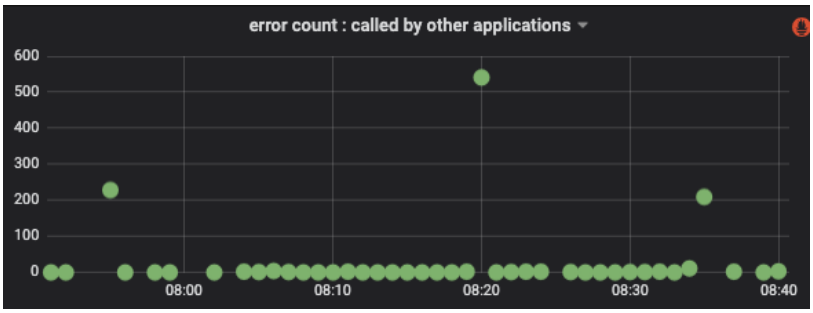


图 14 客户端调用服务端超时次数

2）解决方案

a. 通用性 JVM 参数调优

检查 -Xmx -Xms 这两个值设置的是否一样，如果不一样 JVM 在运行时会根据实际情况来动态调整堆大小，这个调整频繁会有性能开销，并且初始化堆较小的话，GC 次数会比较频繁。

- 效果

-Xmx3296m -Xms1977m 改成 -Xmx3296m -Xms3296m 后效果如下图所示，频率和时间都有明显的下降。

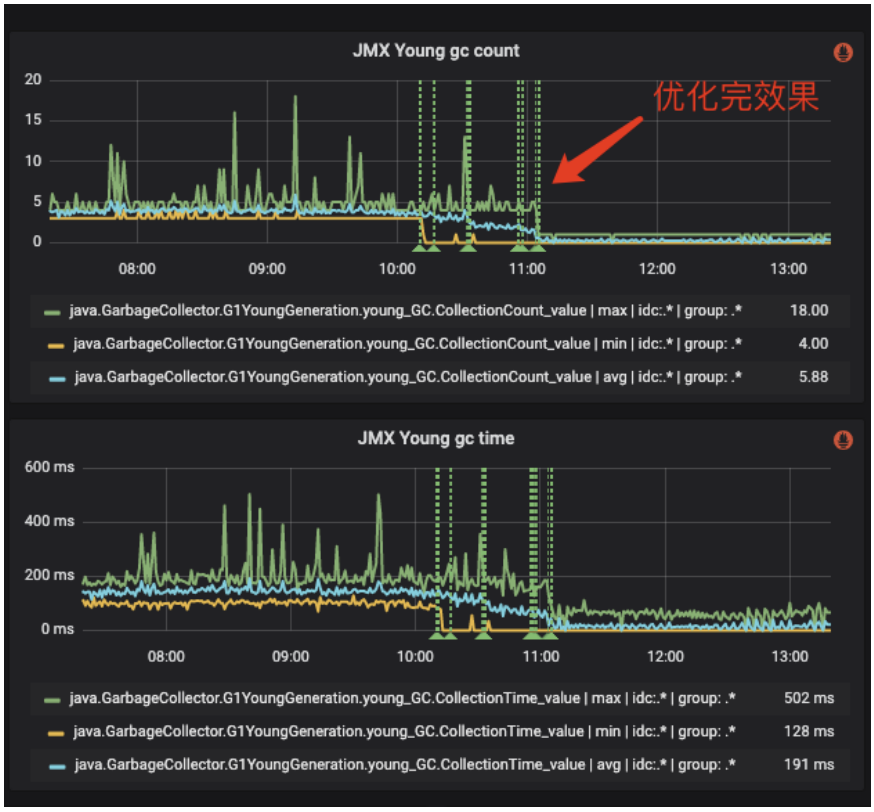


图 15 通用性 JVM 参数调优后效果

b. G1 垃圾回收器参数调优

背景：如果没有设置新生代最大值和最小值或者只设置了最大值和最小值中的一个，那么 G1 将根据参数 G1MaxNewSizePercent（默认值为 60）和 G1NewSizePercent（默认值为 5）占整个堆空间的比例来计算最大值和最小值，会动态平衡来分配新生代空间。

JVM 刚启动默认分配新生代空间是总堆的 5%，随着流量的增加，新生代很容易就满了，从而发生 Yong GC，下一次重新分配更多新生代空间，直到从默认的 5% 动态扩容和合适的初始值。这种配置在发布接入流量或者大流量涌入时容易发生频繁的 Yong GC。

针对这类问题，优化方案是调大 G1NewSizePercent，调大初始值，让 GC 更加平稳。这个值需要根据业务场景参考 GC 日志中 Eden 初始大小分布来设置，太大可能会导致 Full GC 问题。

以查询引擎为例，根据 GC 日志分析，新生代大小占堆比例在 35% 后相对平稳，设置的参数为：

-XX:+UnlockExperimentalVMOptions -XX:G1NewSizePercent=35

- 效果

优化后效果如下图，可以看到优化之后 GC 次数从 27 次/min 降低到 11 次/min，GC 时间从 560ms 降低到 250ms。

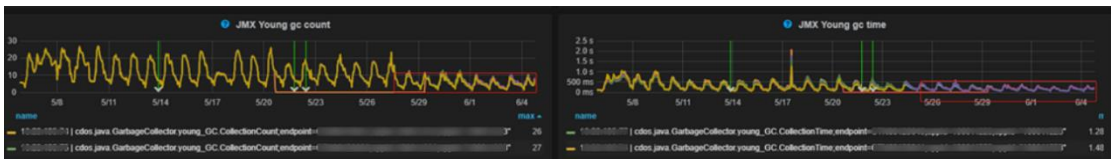


图 16 G1 参数调优后效果

3.6 线程异步改成 NIO 异步编程

NIO（非阻塞 IO）可以减少线程数量，提高线程的利用率，从而降低线程切换带来的超时。

1）分析：CPU 指标

分析 CPU 相关指标，如果出现 CPU 利用率正常，CPU Load 高需要重点关注（如果是 CPU 利用率高的情况，说明 CPU 本身就很繁忙，那 CPU Load 高也比较正常）。

在分析之前，先介绍几个概念：

a. CPU 时间片

CPU 将时间分成若干个时间片，每个时间片分配给一个线程使用。当一个时间片用完后，CPU 会停止当前线程的执行，进行上下文切换到下一个任务，以此类推。

这样可以让多个任务在同一时间内并发执行，提高系统的效率和响应速度。

下图模拟了单核 CPU 执行的过程，需要注意的进行上下文切换是需要开销的，但实际一次上下文切换需要的时间很短（一般是微秒级别）。

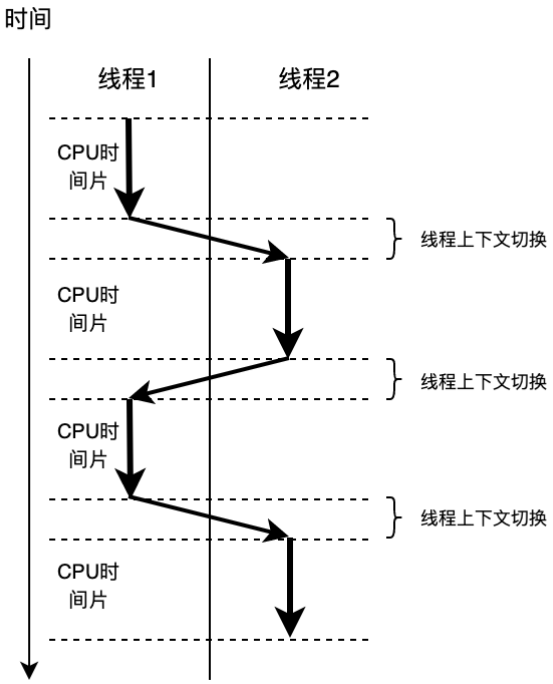


图 17 CPU 执行线程流程

b. CPU 利用率

按时间片维度来理解，假设每次时间片都正好被使用完。

$$PU利用率 = \frac{\text{CPU 执行线程时间片数}}{\text{CPU 执行线程时间片数} + \text{CPU 空闲时间片数}}$$

c. CPU Load

从上面概念分析，如果出现 CPU 利用率正常，但是 CPU Load 高，那说明 CPU 空闲时间片、等待线程数很多，正在使用的时间片很少，这种情况要减少 CPU Load 需要减少等待线程数。

$$CPU\ Load = \frac{\text{CPU 进程队列中线程的数量(正在使用时间片线程} + \text{等待线程数)}}{\text{同一时间单核 CPU 可处理的线程数}}$$

2）分析：实际案例

我们之前生产遇到过多次 CPU Load 高 CPU 利用率正常的情况。问题出现前后代码没有变动，比较明显的变化是流量有上涨。排查代码发现有使用线程池并发调用接口的地方，调用方式如下图。

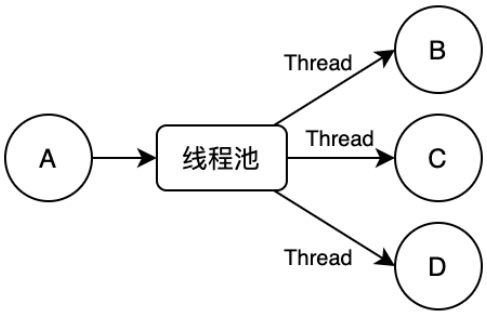


图 18 线程池执行模型

这种方式在流量较低的情况下看不出什么问题，流量变高会导致需要的线程数量成倍增加。

例如：一次请求 A 需要调用 BCD 3 个接口，那 100 个并发需要的线程数就是 100 + 3*100=400（第一个 100 是 A 对应的主线程，后面的 3*100 是 BCD 需要的 100 个线程）。

3）解决方案

线程池并发调用改成 NIO 异步调用，如下图所示。

和之前对比，100 个并发，需要的线程数也是 100（这个地方不考虑 NIO 本身的线程，这个是全局的，并且是相对固定很少的线程数）。

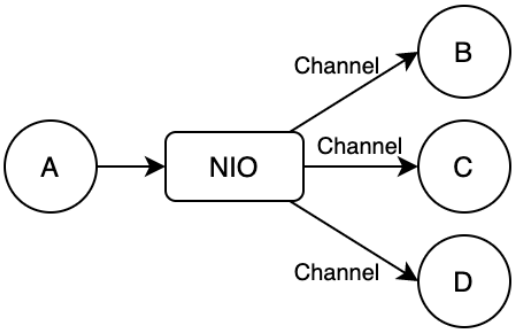


图 19 NIO 异步调用执行模型

4）效果

超时问题没再出现，CPU Load 平均下降 50%，之前 Load 经常超过 2（CPU 核数为 2），改造之后 Load 降到 0.5 左右。

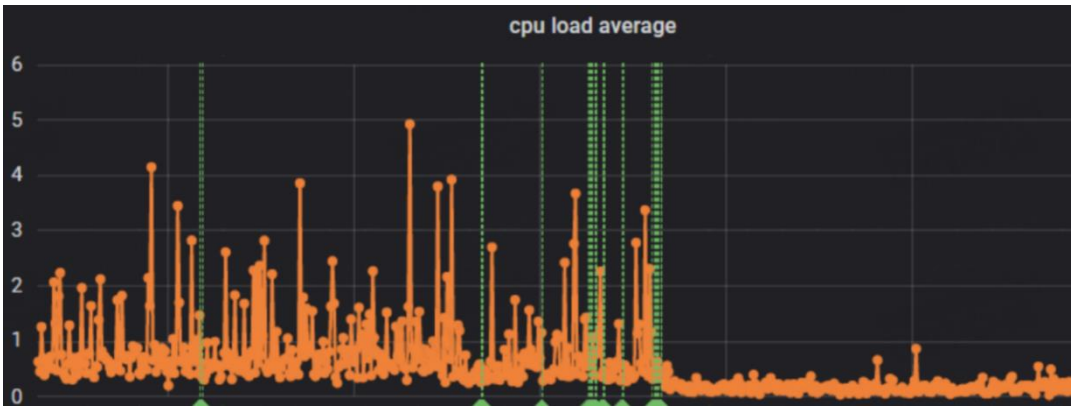


图 20 CPU Load 优化后效果

3.7 启动阶段预热

启动阶段预热可以提前建立链接，减少流量接入时的链接建立，从而降低超时的发生。

1）分析

应用拉入后出现大量超时，并且 CPU Load 高 CPU 利用率正常，说明有很多等待线程，这种是拉入后有大量请求在等待被处理。

之前我们生产遇到过是在等待 Redis 建立链接，建链的过程是同步的，应用刚拉入请求量瞬间涌入就会导致大量请求在等待 Redis 建链完成。

2）解决方案

启动阶段预热提前建立链接，或者是配置 Redis 的最小空闲连接数。其他资源准备也可以通过启动阶段预热完成，比如 DB 链接、本地缓存加载等。

3.8 优化 JIT

JIT（Just-In-Time）编译可以提高程序的运行效率，灰度接入流量将字节码编译成本地机器码，避免对接口性能的影响。

1）JIT 介绍

JIT 是 Just-In-Time 的缩写，意为即时编译。JIT 是一种在程序运行时将字节码编译为本地机器码的技术，可以提高程序的执行效率。

在 Java 中，程序首先被编译为字节码，然后由 JVM 解释执行。但是，解释执行的效率较低，因为每次执行都需要解释一遍字节码。为了提高程序的执行效率，JIT 技术被引入到 Java 中。JIT 会在程序运行时，将频繁执行的代码块编译为本地机器码，然后再执行机器码，这样可以大大提高程序的执行效率。

2）分析

JIT 技术可以根据程序的实际运行情况，动态地优化代码，使得程序的性能更好。但是 JIT 编译过程需要一定的时间，因此在程序刚开始运行时，可能会出现一些性能瓶颈。

如下图应用拉入后 JIT 时间很久，那可以确认是 JIT 导致超时。

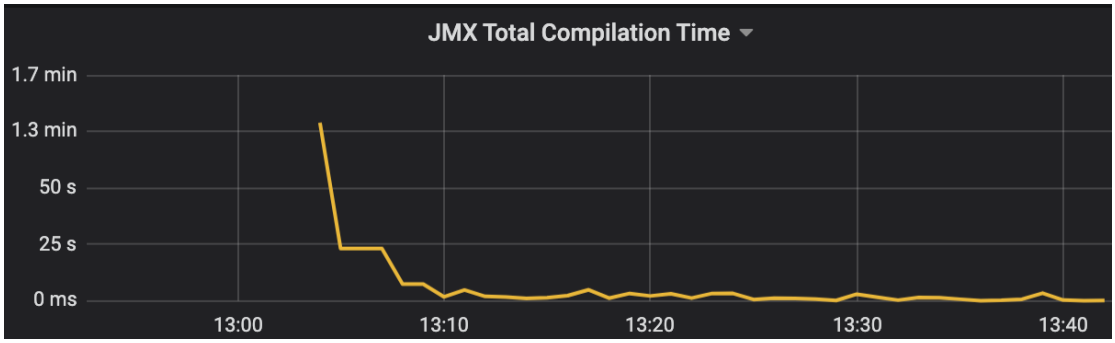


图 21 JIT 执行时间

3）解决方案

优化 JIT 一个比较好的方案是开启服务预热（预热功能携程 RPC 框架是支持的）。

原理是让应用拉入后不是立马接入 100% 流量，而是随着时间移动来逐渐增加流量，最终接入 100% 流量，这种会让小部分流量将热点代码提前编译好。

4）效果

开启服务预热后，如下图所示，应用流量是逐渐增加的，可以看到响应时间随着时间越来越低，这就达到了预热的效果。

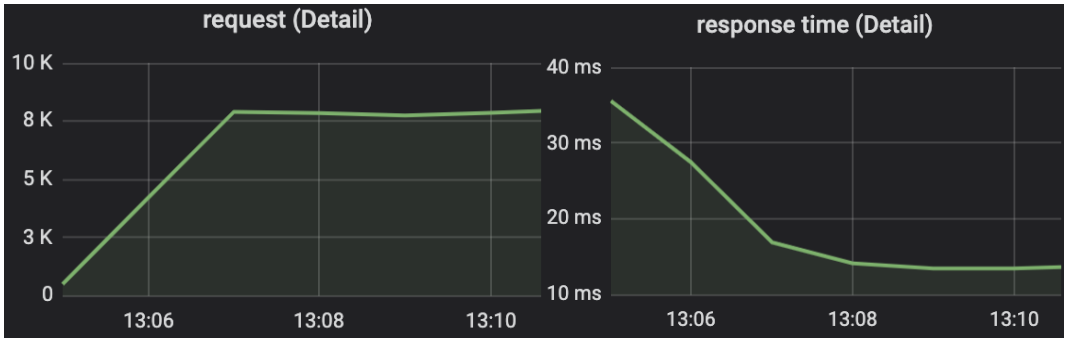


图 22 服务拉入后请求量和响应时间

3.9 换宿主机

当宿主机负载过高时，可以考虑更换宿主机，避免宿主机负载过高影响容器负载。

1）分析

a. CPU Throttled 指标

看下应用 CPU 节流指标，CPU 节流会导致 CPU 休眠引起服务停顿。如果 CPU 利用率正常还是出现了 CPU 节流，这种大多数都是宿主机问题导致。

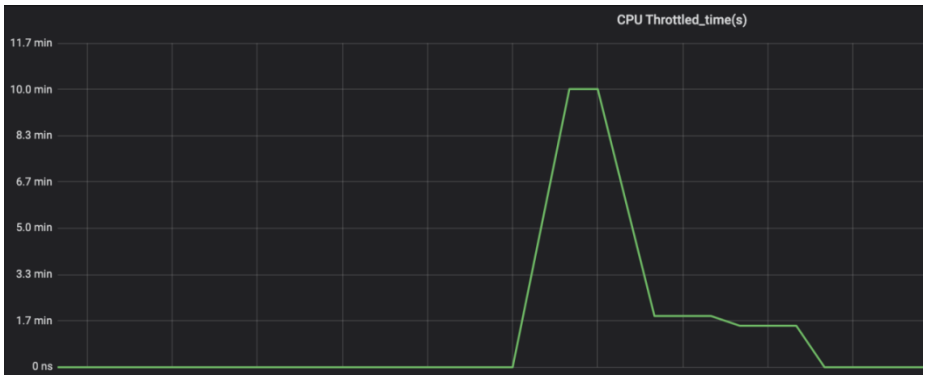


图 23 CPU 节流情况

b. 宿主机指标

排查宿主机 CPU 利用率、CPU Load、磁盘、IO 等指标是否正常，如下图 CPU Load 在某个时刻后大于 1 说明宿主机负载较大。

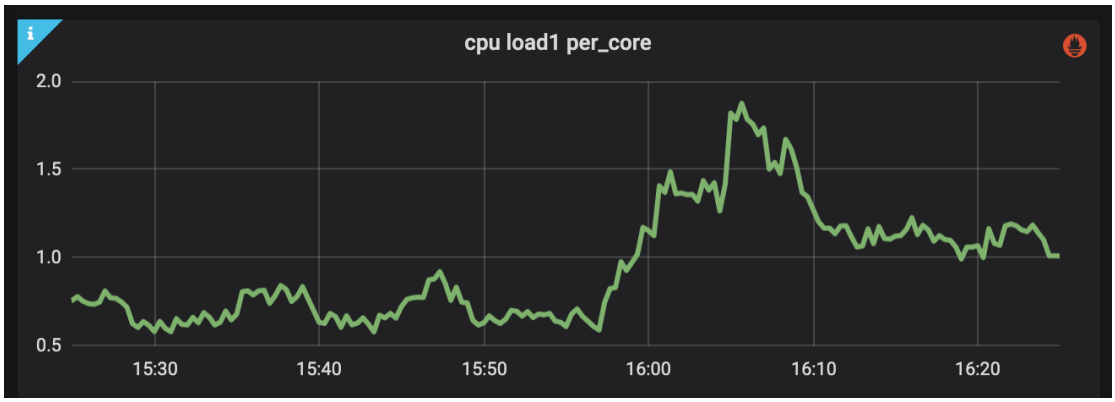


图 24 宿主机 CPU Load 监控

2）解决方案

重启机器换宿主机。

3.10 优化网络

排查不稳定的网络线路，保证网络的稳定性。

1）分析

网络的重点看下 TCPLostRetransmit（丢失的重传包）指标。比如下图，某个点指标异常，而这个点其他指标都正常，那可以初步怀疑是网络问题导致，最终确认需要找网络相关团队确认。

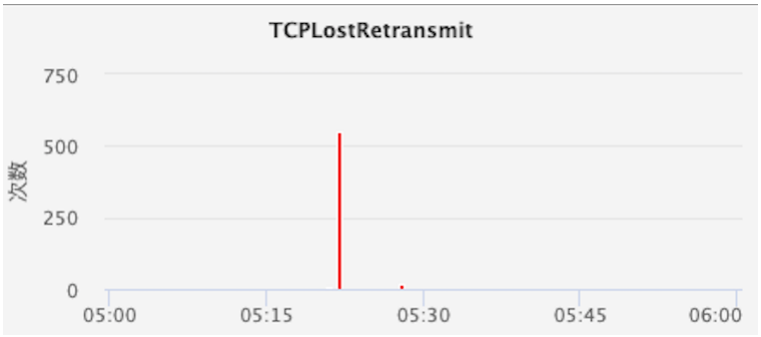


图 25 TCPLostRetransmit 指标

2）解决方案

找网络相关团队排查优化。

四、总结

回顾全文，主要讲解遇到超时问题怎么分析、怎么定位、怎么优化，从简单到复杂总结了 10 种常见的优化方法。这些方法不一定能解决其他不同业务场景的超时问题，具体需要结合自己的实际业务场景来验证。

本文总结的方法都是我们在生产中遇到的真实情况，通过不断实践总结出来的，希望这些内容能够给阅读本文的同学带来一定的收获。

4.1 优化注意事项

超时时间设置和 GC 调优需要结合自己业务场景来优化。

NIO 异步编程改造成本、复杂度较高，我们也在探索更简单的方式，例如 JDK19 引入的虚拟线程（类似 Go 协程），可以用同步编程方式来实现异步的效果。

携程国际机票架构重构实践

【作者简介】Mega，携程国际机票工程师，关注 Java、devops 领域。

一、前言

大多数的技术研发都对重构有所了解，而每个研发又都有自己的理解。从代码重构到架构重构，我参与了携程大型全链路重构项目，积累了一点经验心得，在此抛砖引玉和大家分享。

二、重构的定义

重构是指在不改变外部行为的情况下，改进其内部结构的软件系统更改过程。

三、重构的原因

3.1 组织架构调整

目前携程大部分业务场景都使用了微服务架构，要求服务应该封装单一的责任或单一的能力，以形成松散耦合的服务架构。

根据著名的康威定律，保证一个团队可以独立工作、快速交付变更、尽可能消除团队之间协作和协调的费力度。

所以当组织架构因为业务发展需要做相应调整时，决定了服务的架构也会需要相应的重构。

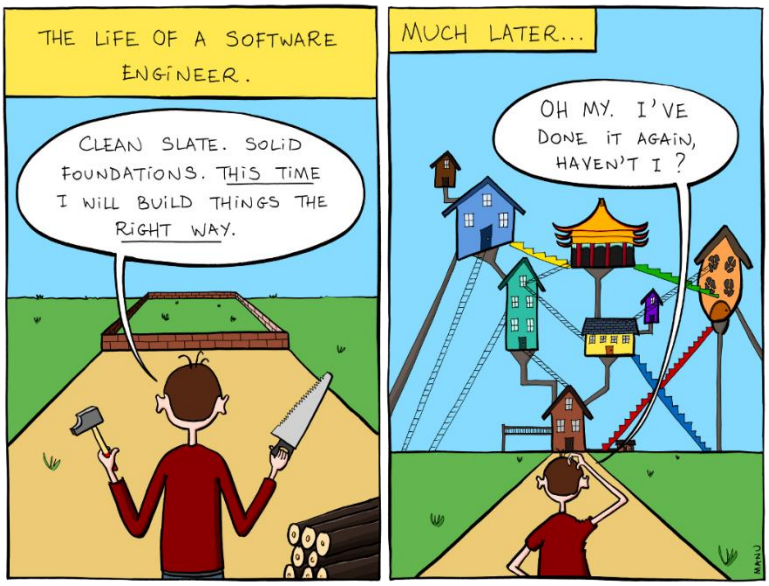
3.2 提升研发效能

重构遗留代码的目的，通常是使系统其更易于维护，减少维护系统所需的工作量，释放出研发资源处理更高价值的任务。

即使是在项目成功落地之后，仍然需要持续的系统迭代，支持新的业务需求。迭代过程引入新的缺陷也是难以避免的。

即使是最好的团队也可能会提交不成熟的代码，这往往会导致代码复杂性让系统变得难以维护，长期以往系统会越来越难以维护，影响研发效能。越来越长的开发周期，导致业务需求发布被推迟。

重构可以减少构建周期时间，改善需求交付时间。



3.3 偿还技术债

技术债是很多研发团队为了追求交互的速度，倾向于选择更简单但不太可靠的方案所付出的代价。短期内为了更快地交付做出的妥协都会在未来带来更多的工作量。

重构有助于消除开发人员积累的技术债务的数量。

3.4 应用现代化

遗留代码可能无法利用现代 DevOps 集成来加速 CI/CD。

遗留代码需要改造成无状态服务，这样可以适配云上的弹性伸缩，优化成本。

四、重构的影响面

从重构的影响面可以分为最高层次的架构重构和最低层次的代码重构。

代码重构是在修改代码后不会改变对外的行为，主要目的是提高整个软件的质量。一个重构通常是一个小任务，但是多个重构应用于代码可以显著提高其质量。

架构重构主要是将现有代码重新组织成新的层级，改变代码在逻辑层次中的组织方式。

4.1 架构重构和代码重构的比较

代码重构通常在一个开发团队内部达成一致，影响的范围容易控制和验证，风险较低。

架构重构必须要多个团队甚至是技术委员会达成一致，影响范围大，风险和成本较高。通常在做架构重构前尝试用可量化的指标来确定投入产出比，决策是否值得重构。

此外架构重构相比代码重构难度更高，对研发的专业知识要求也会更高，项目组里至少需要有一个熟悉链路上所有服务的资深研发，负责重构决策（特别是服务和服务边界处）。

五、重构计划

在开始重构编码之前，对现有系统深入分析，评估哪些部分应该被重构以及重构的优先级，有助于列出重构计划。

全链路上的大规模重构往往都是长期项目，初期制定的计划也肯定会有变化调整，特别是重构过程中会有新的业务需求进入，需要调整优先级，所以计划要分成不同的阶段性里程**碑目标。**

六、业务研发团队沟通

重构期间需要和负责业务需求的研发团队同步协调计划，优先对新需求部分重构，避免新功能完成后立即又成为重构任务。

例如，重构后的版本在下周就会发布运行，有些业务需求计划可以直接在新版本上实现，这样既避免了业务研发被技术债务困扰，也减少了重构项目组的工作量。

七、分层设计

分层设计即定义服务之间、模块之间的边界，是完成重构的关键，让代码之间的依赖关系变得更清晰，减少耦合性。

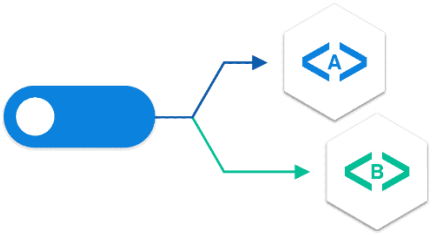
根据关注点分离原则，把现有的功能分类，将对应的代码移动到合适的层级和该层级对应的模块。

前提是需要理解每个类的功能是什么，当面对实现多个功能的类，必须要拆分成多个只服务于一个目的的类。

单一职责的类更具备可测试性，依赖更少的外部功能，减少测试用例里 Mock 的工作量。单一职责的类也更容易被复用，更容易维护。

大型重构项目会有许多工作要做，分层后的设计就像地图，避免研发迷失方向。

八、灰度



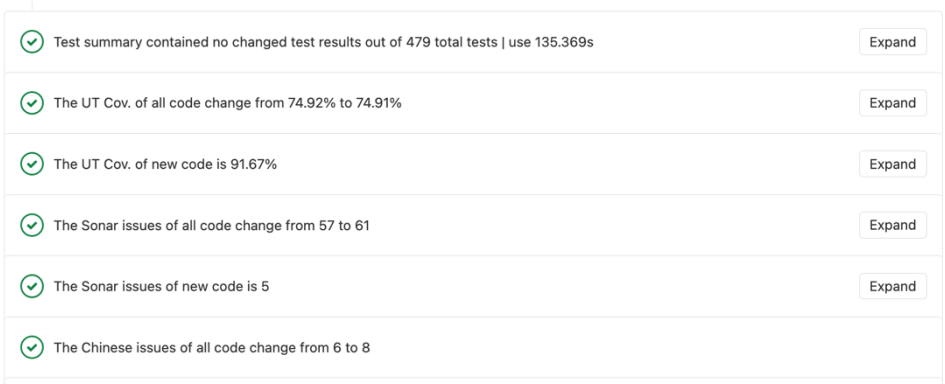
在分阶段重构的过程中，引入抽象层（例如面向接口编程），同一个接口有新老两套实现。抽象层的引入允许显示系统里新老版本多个实现的共存。

这种看似额外的临时兼容工作，可以带来以下好处：

- 我们在编写测试用例代码的时候，也尽量面向接口验证，这样在旧代码验证通过的案例能够运行在重构后的新代码上。
- 使用特性开关控制新老版本实现的切换，可以在不破坏原有的功能且对调用方透明的前提下逐步灰度在线上版本进行重大更改，最后新版本实现稳定后，删除老代码的实现。这种方式也能减少研发在重构时候的心智负担。
- 特性开关的引入也能够实现快速回退，控制因为新实现带来的破坏范围。但验证不能完全依赖线上业务流量（对业务有损），下文会提到重构版本在发布前的改善质量的方法。

九、改善质量

9.1 静态检查



静态检查工具 Sonar

- 解决循环依赖
- 解决 Cyclomatic Complexity
- 控制每个类的方法数
- 控制每个方法的行数
- 解决 warning（删除不使用的代码，也意味着减少重构的工作量）

9.2 自动化测试

自动化测试是重构最重要的基础，它能确保系统的行为不会因为重构而改变。

在重构开始优先保证测试的覆盖率，所以我们会在重构前优先使用集成测试来验证重构结果:

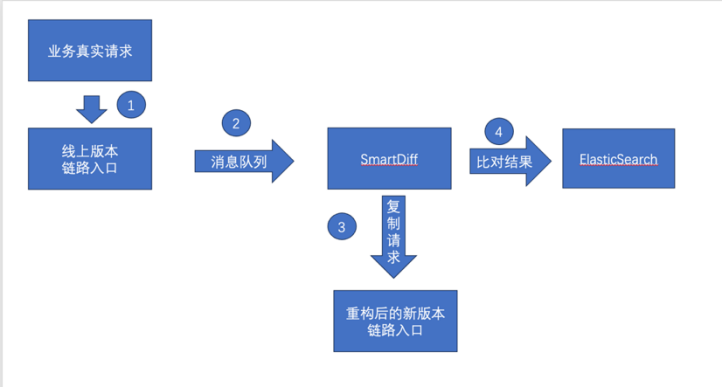
- 集成测试相比单元测试覆盖率会更大，随着覆盖率越高，覆盖剩余的测试成本也会越大，例如一些异常边界场景，用单元测试覆盖的成本会更低。
- 有些单元测试往往和老系统代码耦合，没有面向接口测试，或者是一些静态类的测试。重构代码意味着单元测试需要重写，在早期重构阶段优先用集成测试覆盖验证。

设计集成测试需要减少对其他系统（尤其是和外部第三方交互）的依赖，通过一些 Mock 工具提供稳定的测试数据，让测试结果更稳定可靠，比如使用缓存保证相同的请求能拿到相同的结果，这也能帮助后续新老版本的比对回归验证。

借助 CI 持续迭代，每一次变更提交会触发自动化测试回归验证，在早期开发阶段快速暴露问题。同时也鼓励研发多次提交变更，避免一次提交包含太多的功能点，减少测试不通过排查问题的复杂度。

9.3 链路比对

在整个链路上的服务完成一次里程碑后，会做链路上的端到端的自动化测试，比对新老系统结果，验证整条链路。

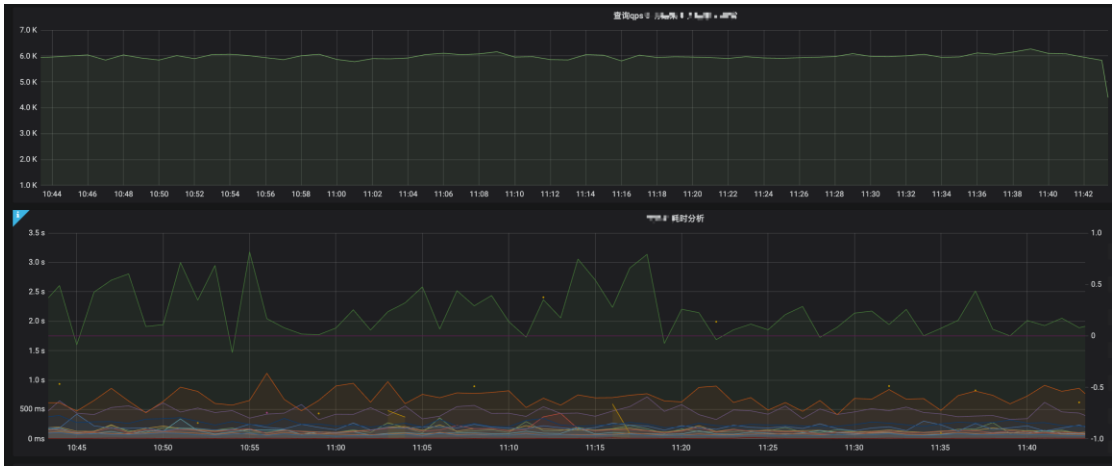


a. 线上版本链路入口在收到业务真实请求后，会把请求和响应推送到消息队列里。

b. 比对工具 SmartDiff 订阅消息队列，复制请求重构后的版本链路入口，拿到结果后和线上版本结果比对

c. SmartDiff 把比对结果写入 ES 日志，研发分析日志排查比对差异原因，修复问题。

十、可观察性



很多时候，软件架构容易被作为一个学术概念。即使是在一些成熟的研发团队，团队成员能够描述系统架构应该实现哪些重要的非业务功能需求（吞吐量、耗时、稳定性等等），但很难证明系统确实做到了这些。

尤其是系统经过长期多次迭代后，当前的架构实现是否还遵循原先的架构设计呢？如果无法验证，研发对架构也会失去信心。

所以我们需要对系统定义多个非功能性指标并将其可视化，线上持续地观察和确认是否和我们当初设计期望的指标一致。

比如读写缓存接口响应 P90 保持在 100ms 以内，假设后续为了提升缓存命中率，架构重构引入了多个缓存，P90 线 100ms 就可能会不达标。

指标可视化帮助我们快速发现新的设计带来的问题并做相应调整。这些非功能性指标就如同测试用例一样，快速发现问题、增强研发对架构重构的信心。

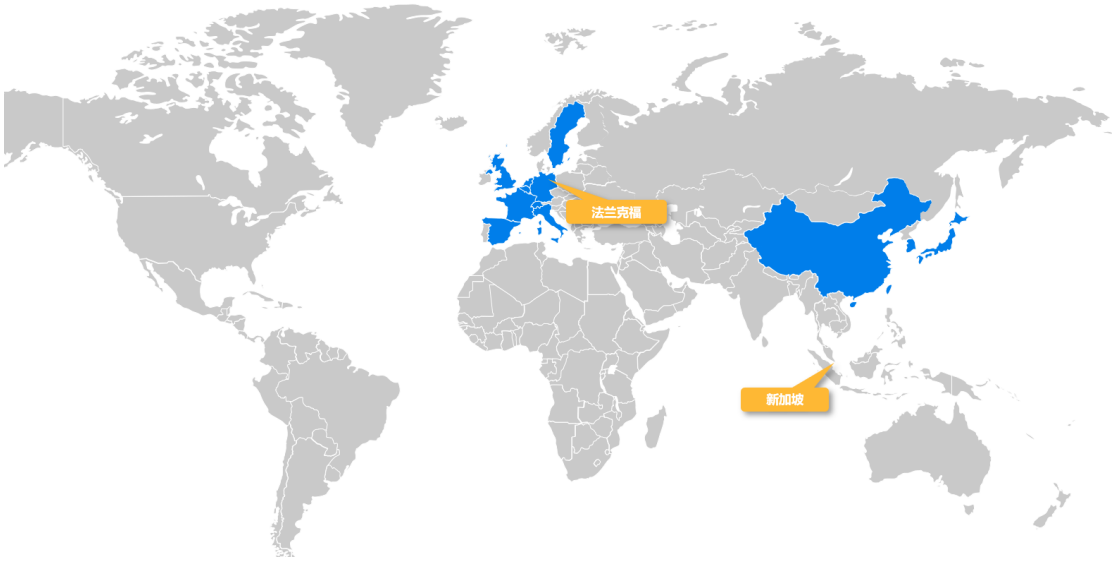
携程火车票出海架构演进之路

【作者简介】py.an，携程后端研发经理，关注性能优化、技术架构等领域；venson，携程后端高级研发经理，关注性能优化、技术架构等领域

一、引言

在全球化战略的背景下，Trip.com 作为一个面向国际市场的全球 OTA 平台，正努力推进国际化战略部署。Trip.com 火车票正在积极投入资源和技术力量来拓展海外业务，通过将应用、数据部署新加坡、法兰克福等中心，从而给全球用户带来更好的购票体验和减少数据合规带来的风险。

二、业务背景



如图所示，目前 Trip.com 火车票全球铁路业务主要集中在英国、亚洲和欧洲各国，其中欧洲作为世界上经济、交通非常发达的大洲，也成为更加关注的一站，未来还有更多更大的舞台。

随着全球疫情危机消退，旅游和出行需求得到释放，在多语言，多币种的场景支持下 Trip.com 火车票的全球化业务局面已逐步形成。

三、面临的挑战

全球化背景下，除了要考虑全球的平滑部署来满足应用可用性和用户访问性能要求外，还需要考虑数据出海的安全性、法律合规和数据隔离等严格要求。通过以下几个角度举例:

3.1 全球部署

改造前，Trip 火车票业务应用和数据都部署在原机房的同城：存在 IDC A+B 两中心的（同一个逻辑机房）同城双活。

与改造前架构特点相对比，如表格所示：

	容灾级别	同一逻辑机房	用户分区	就近访问	数据多活	公共组件
改造前(同城双活)	跨机房级别	是	否	否	否	支持完善，成熟
全球多中心	region 级别	否	是，单元化分区	是	需严格遵守数据跨境政策	需支持多 IDC 场景

由此得知，多 IDC 场景下不可避免地需要去面临数据分片、单元化、数据冲突和业务幂等问题。相比传统分布式架构，不止是业务应用项目，还有 PaaS 平台基础设施在应对全球化技术体系都遇到了全新的挑战，需要有巨大的调整。

3.2 性能问题

面对全球范围内的用户的业务请求响应，难免会有用户因为网络跨洋传输、链路传输距离过长等问题造成的业务访问质量差。如何保证用户的请求访问链路最优，减少网络延迟，提供更快服务响应。

3.3 数据合规和监管

如何严格遵守不同地区针对数据跨境流动、数据泄露等数据安全问题颁布的相关法律法规。

3.4 数据出海问题

数据一致性：多 IDC 读写场景下，全球范围内用户在多个数据中心创建和操作订单，多个数据中心之间相互同步和操作订单业务时，应该如何保证数据一致性的问题。

同步合规：因数据跨境政策影响，一般不进行异地多活，需要如何避免数据跨境流动所带来的违规。

3.5 全球扩展性

以轻松地扩大业务覆盖范围为目标，新业务扩展时，如何通过对业务和数据进行改造操作，达到便捷动态调整数据存储策略，来应对动态多变的的数据合规政策。

下面将结合全球化面临的挑战和问题，从海外部署、数据合规、架构改造实践等角度来详细说明 Trip 火车票全球化出海的架构演进实践。

四、出海架构演进实践

4.1 Region(可用区)选择

选择适合的 Region 需要考虑用户需求、法律和隐私、基础设施和网络、数据跨境风险评估以及成本和效益等多个因素。

Trip 火车票根据以上因素和自身业务需求发展方向综合考虑，并进行详细的市场调研和分析，做出可用区选择：把新加坡（SIN）和法兰克福（FRA）作为火车业务出海部署的数据中心。

4.2 网络接入层

Trip 火车票如何设置网络路由以实现可靠、高效的路由访问和数据传输，总共分三种场景。

- 外网：多路径、就近访问。

考虑到不同地域之间的网络延迟和带宽限制，Trip 火车票采用就近访问路由策略。即选择距离最近或带宽最大的路径进行数据传输，以减少延迟和提高速度。优势：保证同一用户就近访问网路链路最优的 IDC。

配置 FRA、SIN 多条路径进行数据传输，多路径路由。这样即使某一条路径出现故障，数据仍然可以通过其他路径传输。

- 内网：尽量访问同 Region 内的资源，实现同 Region 业务闭环。

- 跨 Region 访问场景：如果同 Region 内不存在需要获取的业务资源，必须跨 Region 访问时，则进行链路优化。比如，欧洲用户访问 FRA 通过专线链路请求 SIN 资源。这样避免直接跨洋访问其他 Region，因网络跨洋传输、链路质量不稳定等问题导致网络耗时过长。

4.3 数据层

1）数据出海合规改造

数据出海合规改造是一项复杂而重要的任务，需要综合考虑各种法律、法规和业务需求。通过以下改造措施，可以确保跨境数据传输和处理过程的合规性，并为用户提供更可靠的数据保护：

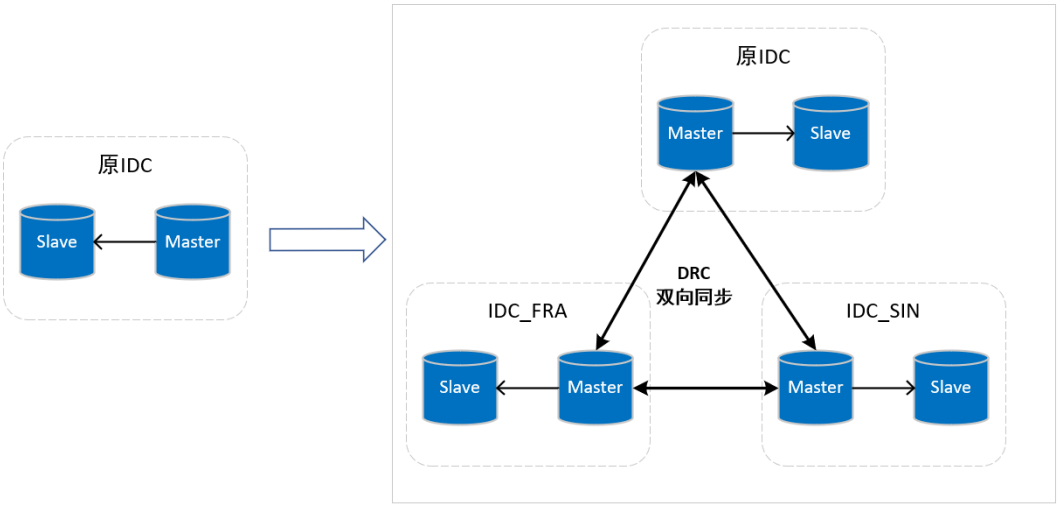
数据分类和标记：对业务数据进行分类和标记，明确标识出敏感数据、个人身份信息等受保护的数据。这有助于在数据传输和处理过程中更好地掌握敏感数据的位置和处理方式。

数据加密和匿名化：采用适当的加密技术和数据匿名化方法，对敏感数据进行保护。加密可以有效防止数据在传输和储存过程中被未经授权的访问者获取，而数据匿名化则可以保护个人身份信息的隐私。

出海数据业务剥离改造：数据跨境流动许多国家实施数据本地化策略，数据出海时需同时考虑数据输出地和输入地的数据跨境规则。跨境数据传输时需要进行风险识别和相关的数据控制措施，对业务数据进行剥离改造。

2）DB 多 IDC 部署

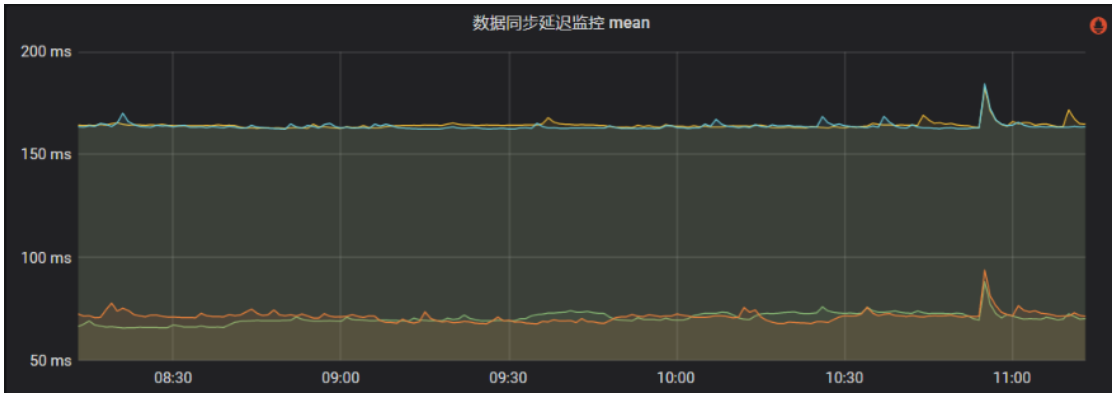
为确保能够满足业务需求，并提高数据库的可用性和容错能力，将出海 DB 进行多 IDC 部署方案。如下图所示：



需要注意的是，各 IDC 之间同步时，应考虑各国家和地区的法律法规要求，确保同步数据的链路符合当地的数据存储和隐私保护规定。

此外，多个 DB 数据相互同步时，架构会变得非常复杂。为确保各个 IDC 之间的网络延迟低、数据同步稳定，要关注每条同步链路的延迟、网络链路抖动和数据一致性问题，并且要定期进行监控、测试和演练，以验证整个部署方案的可靠性和有效性。

3）同步延迟监控



如上图所示，例如同步链路 DRC 同步延迟时间：

SIN<—>FRA:160ms+

4）数据库多 IDC 扩展性

引入 RegionCode：插入用户数据时增加记录机房标识 RegionCode。

根据 RegionCode 确定数据所在 Region，使得常用的数据查询或业务处理操作可以在单个节点上执行，以达到数据单元化处理和数据合规策略动态调整的效果，从而避免跨节点带来额外性能消耗和数据跨境合规问题。

4.4 基础组件层

1）PaaS 基础组件多 IDC 接入

a. 分布式配置中心：

应用多 IDC 部署的场景下，就出现了不同 IDC 环境下配置文件不同的情况，此时也需要对配置中心的配置文件进行调整：接入子环境，引入多 IDC 配置文件，支持不同 IDC 不同的配置场景。

b. 分布式调度中心：

因为业务中大部分 JOB 都是通过扫表来对数据进行批量处理，所以多 IDC 场景下则基于存储的 RegionCode 将任务分散到多个 IDC，数据经过单元化过滤后，进行分片处理。

c. Redis:

不做双向同步，多数据源。

业务中用到 Redis 的场景比较多，但 Redis 不同于业务数据库场景所以不做双向同步，每个 IDC 对应同单元内的 Redis 集群，每个 Redis 集群只服务于当前单元内的业务，所以不

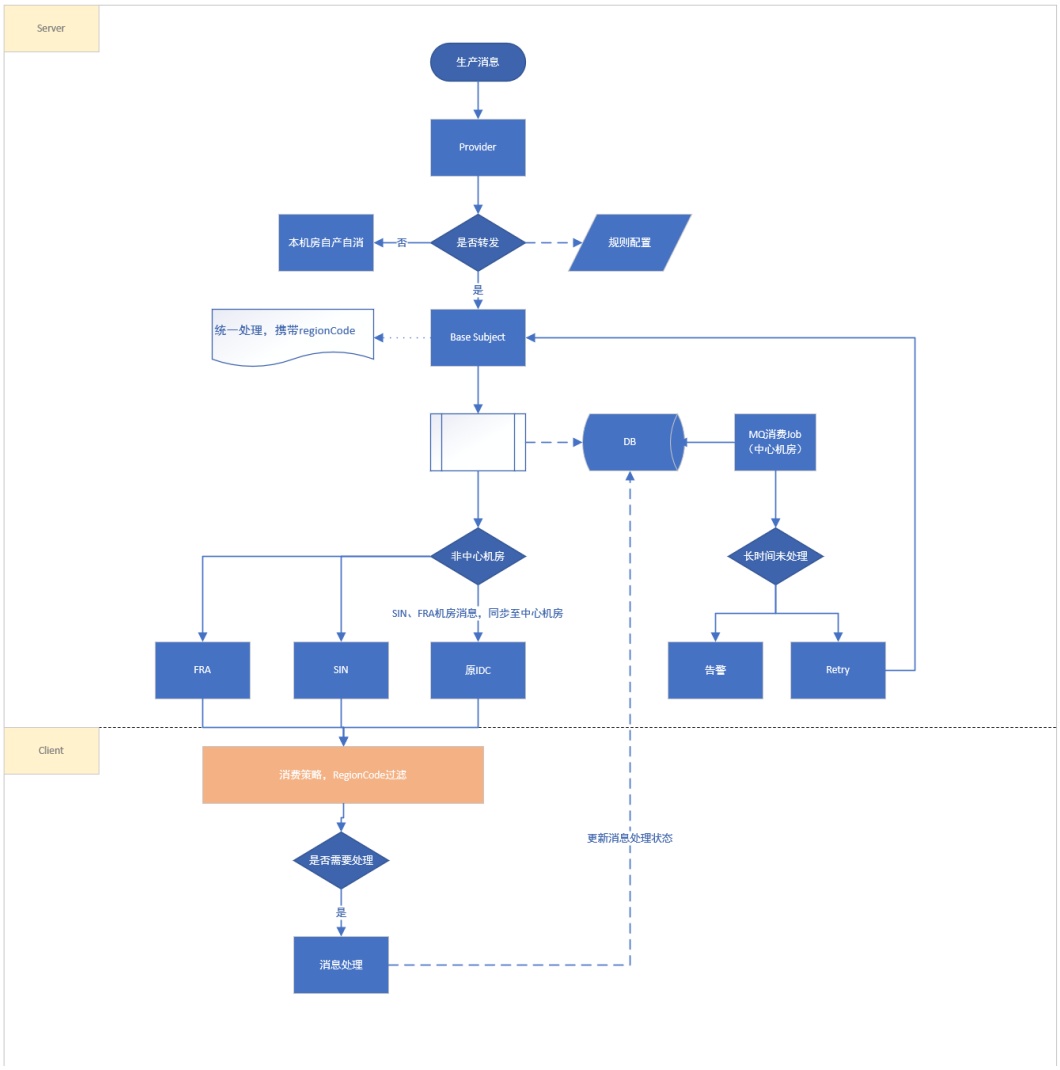
是全量的。所以在多 IDC 的场景下就有很多业务场景需要调整，基于 Redis 覆盖业务要保证单元内闭环。

2）消息中心多 IDC 改造

MQ 每个集群都是相互独立相互隔离的，多 IDC 场景下就必然面临了消息处理幂等的问题，所以对 MQ 进行了逻辑分组改造：

- 同 Region 内处理：同机房内的生产消费的 MQ 同 Region 内闭环处理
- 跨 Region 场景：需要跨 Region 的 MQ 通过 BaseSubject 同步到中心机房 Region，来保证正常业务流程
- 消费端幂等处理：消费端根据 RegionCode 逻辑分组，进行单元化消费

消息处理的改造流程图如下图所示：



4.5 项目业务层

1）业务单元化闭环改造

按照不同区域进行用户分区和每个单元内可以独立运作的原则。对项目业务进行改造，业务上尽可能保证所有业务在单元内可以独立完成，每个 IDC 可以独立承担部分用户的业务处理的能力。

2）请求链路改造

尽可能保证在同 Region 执行，减少跨洋请求造成的网络耗时过长等问题。

3）跨 Region 场景改造

跨 Region 耗时请求下，由原来的串行调用外部接口的业务处理逻辑调整为异步并发处理和数据预加载优化。比如获取用户优惠券场景下，需要跨 Region 获取，则采取提前请求优惠券的方式，去除掉跨 Region 的影响。

多次跨 Region 的场景通过接口改造减少跨 Region 的次数从而达到减少跨洋的效果。

当核心业务中的非核心跨 Region 业务时：采用非即时性处理原则，通过业务拆分对非核心业务进行异步 MQ 改造处理。

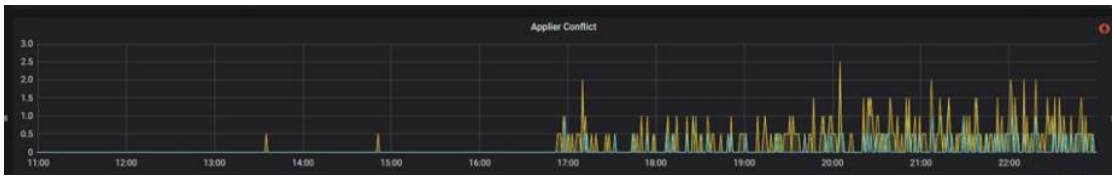
4.6 改造中的问题，演进中的思考点

在实际项目改造过程中，困难也属于改造过程中的一部分。关键是要拥有一个积极应对和解决问题的心态，通过分析问题、制定解决方案、执行和学习经验，从而克服困难并推动项目改造的顺利进行。

以下是改造过程中遇到的问题点以及解决方案

1）DB 同步冲突问题

在生产环境数据同步开启后，突发了网络不稳定造成 DRC 同步链路阻塞情况。



如图所示，在监控到 DRC 同步链路不稳定时，触发了 DRC 同步冲突告警。

原因：通过对 DB 数据的排查发现 SIN 和 FRA 对同一订单进行的更新操作，因为网络延迟导致同步时发生了 DRC 冲突，导致其中一个更新操作被丢弃，从而影响到了后续订单流程。

解决方案：修改订单更新逻辑在同 IDC 内执行。双写发生同步延迟问题必然会遇到一致性冲突问题，长期方案还是单元化，避免出现跨 Region 操作同一条数据的情况。

2）分布式锁问题

当前项目中的分布式锁是基于 Redis 实现的，因为不同 IDC 的 Redis 集群是相互隔离的，所以目前分布式锁的粒度只支持到了 Region 级别。目前业务都是围绕用户场景加的分布式锁，所以也可以满足目前的实际业务场景。如果后续有全局获取分布式锁的业务，则需要进一步设计，即保证同一时间所有 Region 有且只有一个地方能够获得该资源，并且其他 Region 必须等待，这有可能牺牲掉相当大的性能来实现此功能。

3）多机房库存问题

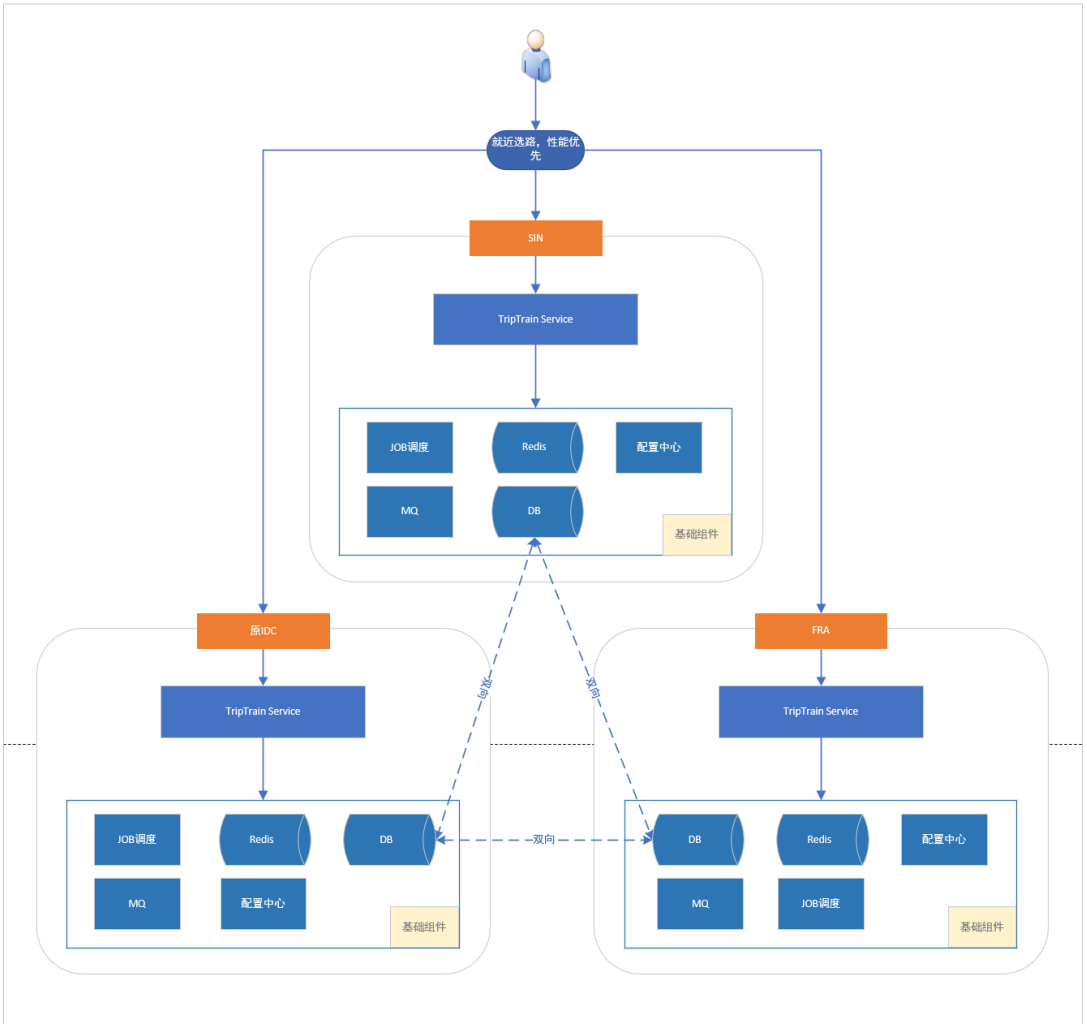
用户的请求保证在同一机房内完成闭环，但部分场景并不适合划分单元化，比如多机房库存扣减问题。面对多机房库存扣减问题目前的策略如下：

- 业务扣库存逻辑不调整，还是同步扣库存，但事先根据流量分配好每个机房库存
- 增加库存调配机制，当库存不足时触发库存调配，从有多余库存的机房进行调配，
- 增加监控和库存不足告警通知，除了自动资源调配，对活动上线后进行机房间的库存情况实时观测和实时手动调配。

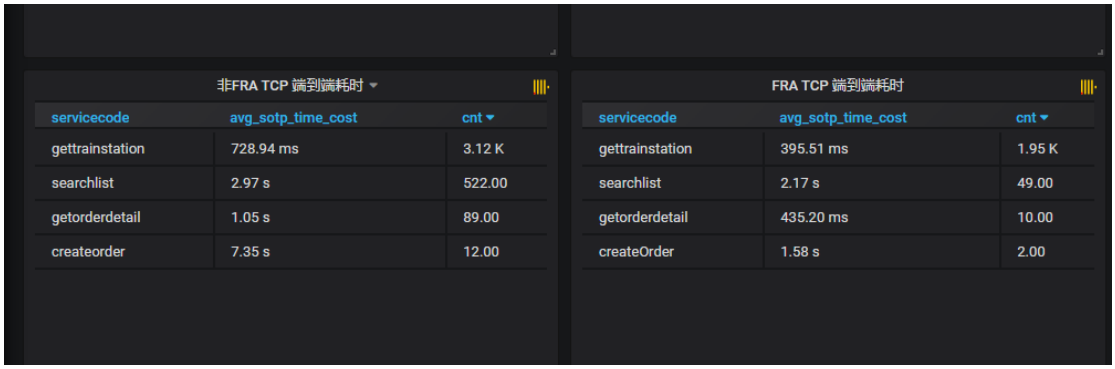
4.7 演进结果

通过以上的改造和优化，Trip.com 火车票的系统架构演进和性能优化如下面所示：

1）架构演进图



2）性能优化



非FRA TCP 端到端耗时

servicecode	avg_sotp_time_cost	cnt
gettrainstation	728.94 ms	3.12 K
searchlist	2.97 s	522.00
getorderdetail	1.05 s	89.00
createorder	7.35 s	12.00

FRA TCP 端到端耗时

servicecode	avg_sotp_time_cost	cnt
gettrainstation	395.51 ms	1.95 K
searchlist	2.17 s	49.00
getorderdetail	435.20 ms	10.00
createOrder	1.58 s	2.00

通过对用户网络链路优化，减少用户跨洋访问。FRA 接口耗时优化整体减少 300-800ms。

五、新起点，新征程

当前背景下还有很多不完善的地方和非常多的技术挑战，架构体系还需要持续演进迭代，接下来 Trip.com 火车票对于未来的全球化战略方向还需进一步进行优化和改造：

5.1 单元化路由

接入集团 UCS(unit control service)路由策略：根据用户的区域信息作为 ShardingKey 映射指定 IDC，以达到流量和组件多 IDC 场景下的完美落地。

5.2 数据单元化改造

当前第一指标是优先保证业务，各个 Region 的 DB 数据都会双向同步，每个 Region 的数据都是全量，也增加容错性，减少了数据出海异常情况时带来的业务中断的风险。但还需达到数据和业务单元内可以完全闭环的程度，可以随时切断同步链路避免数据跨境带来的违规问题，以实现数据单元化。

5.3 业务中心机房调整

为了适应多变的数据合规政策和迎合业务发展趋势，未来的中心机房设置为 SIN 数据中心，并且有能力移除原业务中心机房。

目前需要达到所有业务可以在海外闭环的能力后设置业务中心为 SIN，以达到海外合规建站的能力。

5.4 结语

伴随着 Trip.com 全球化的发展，火车票的技术发展也逐渐从原有的技术领域，延伸到要去应对更复杂的场景。想要建立起完善的全球化体系还有很长的路要走。在这种背景下，还需继续突破自身技术边界，实现单维能力向多维能力的转变，提前布局，并面向业务持续交付技术价值。

携程后台低代码平台的探究与实践

【作者简介】ck，携程后端开发专家，关注技术架构、高并发、性能调优等领域；Geralt，携程前端开发专家，关注前端框架及性能优化；Kaoru，携程资深前端开发工程师，关注前端性能及开发工具。

概述

PGClowcode 平台是携程市场内容 PGC 团队搭建的主要用于后台页面开发的低代码平台，第一版于 23 年 3 月上线，截至 10 月平台已经拥有 100+用户，在平台上开发了 130+个应用和 180+个页面。本文将主要介绍团队采用低代码平台的背景、方案调研、落地过程中遇到的问题以及解决方案，同时也大致介绍了该低代码平台提供的能力。

一、研究背景

1.1 为什么需要低代码平台

软件产品通常由客户端（App、小程序、网页）和运营后台组成，其整个生命周期需要不断地更新迭代，而在实际的迭代开发中经常会出现以下问题：

- 后台需求优先级低，排期经常被延期，通常使用配置中心等简易数据操作平台替代，不仅对运营人员不友好，而且产生了极大的生产风险
- 迭代需求涉及的前后端开发工作量不均衡，经常面临后台需求较多，但前端资源不足，服务端资源即便有空闲也没法帮忙支持后台的需求
- 不同业务间后台技术和组件大同小异，各业务反复造轮子，浪费时间和资源
- 开发人员开发工具类站点时，页面部分的工作需要付出较大的代价

针对这些问题我们也尝试了很多的解决方案，如要求开发人员全栈、剥离公共组件等，但结果都不太理想，经过充分的了解和分析，搭建低代码或零代码平台能在很大程度上解决这些问题。

1.2 需要什么样的低代码平台

低代码平台分为很多种，我们究竟需要哪种呢？经过将以上问题拆分细化和充分调研后，我们的低代码平台应该满足以下要求：

页面搭建方面：

- 界面友好、可视化，可以让研发和相关人员能通过拖拽的简单方式快速搭建 UI
- 页面逻辑仅需写少量简单的代码或无需代码

- 支持满足日常需求的组件库

部署运维方面：

- 具有开发、部署、运维等完整的持续交付功能，最好能一键或自动化完成

安全合规方面：

- 支持权限管理机制，保证系统安全
- 具备完善的发布审批机制，能严格保证开发质量和生产安全

兼容性方面：

- 能嵌入已有后台页面，已有后台尽量少改动的嵌入其开发的页面

可扩展性方面：

- 支持用户自助化组件开发，并且能在平台上推广

二、现状分析

2.1 行业现状

目前国内外低代码或零代码产品不下百种，既有商业平台，也有开源项目，它们在企业内部用于各种运营后台、数据面板、办公系统等内部系统的开发，在 B 端提供商品管理、广告投放、商铺搭建等企业服务，在 C 端广泛用于活动页面、促销频道、广告频道等类型产品的搭建，不仅为企业节省了大量的开发成本、产生巨大的商业价值，同时也为用户提供了丰富多彩的软件产品。

虽然低代码种类繁多，但是每类低代码项目往往具有特定的业务属性，在不同的行业，不同的公司可能需要定制不同的组件，不同的流程，因此市场上很少有能适用于所有场景的平台，也很少有企业愿意去做通用平台。下面分析了目前比较流行的产品：

2.2 产品分析

阿里低代码引擎

LowCodeEngine（阿里低代码引擎）是国内最知名的低代码类产品之一，其完整的实现了《低代码引擎搭建协议规范》和《低代码引擎物料协议规范》，它通过完整的协议栈约定了各种物料的开发、使用、扩展、流通，并且提炼了企业级低代码平台的特性，遵循面向扩展设计的理念，奉行最小内核、最强生态的设计原则而设计的内核引擎。

目前其生态已经相当完备，并且提供了非常详细的文档和使用案例，也有大量的 demo 开源，其使用起来相对比较便捷。但它并没有提供完整的平台代码，使用者需要在其内核的基

础上搭建面向用户的平台和服务系统，具有相当的开发成本。

腾讯低代码平台

腾讯的低代码产品包括搭建生成移动端 H5 页面的 tmagic-editor 开源项目和搭建管理端页面的无极低代码商业平台，其中 tmagic-editor 提供了完整的平台代码，使用者可以在开源社区将整个项目 pull 到本地部署使用，它具备自定义组件、插件、编辑器等扩展能力，内置了丰富的组件，提供了友好的可视化界面，通过拖拽+配置的方式开发页面。

但其限定于移动端 H5 页面的搭建，无法适用 pc 端页面开发的需求，大大限制了使用范围。无极低代码平台是商业化的面向企业付费使用的产品，具备管理端强大的开发能力，甚至结合了 AI 能力，能供面向企业丰富的解决方案，但它目前并没有开源。

开源低代码平台

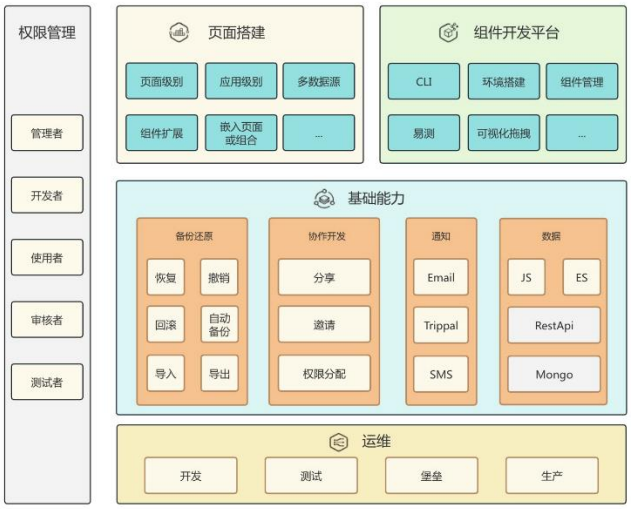
在 github 上搜索 lowcode 关键词可以看到非常多的项目，他们所使用的技术、项目的形态、star 数量、活跃的层度、使用场景各不相同，有些提供了完整的可视化界面，有些则需要通过配置文件或调用接口的方式生成页面或项目，总之需要花费较多的时间对比分析才能找到符合自身需求的项目。

总结

综上所述，我们最终决定选择一款开源程度比较高的项目（appsmith）作为蓝本，然后经过改造、开发搭建了 PGClowcode 平台。

三、平台功能介绍

PGClowcode 平台功能主要包含页面搭建、组件开发平台两大部分。



页面搭建包含开发、预览、测试、发布、回滚、备份、恢复等常用功能。在这些功能的基础

上，增加了"可视化拖拽"、"多用户协同开发"、"导入导出"、"多数据源"、"通知"等功能，形成了一个相当完善的开发体系。搭建的产物可以通过将平台上的应用或页面嵌入到已有的后台，或者反过来将已有的后台页面嵌入到平台，实现组合使用。

组件开发平台是对页面搭建能力的扩展，支持通过 CLI 构建本地项目，自定义开发新的组件以满足更复杂的业务需求。（此功能正在开发中，将在不久之后开放）

由于篇幅有限，下面将介绍几个主要功能。

3.1 用户和权限管理

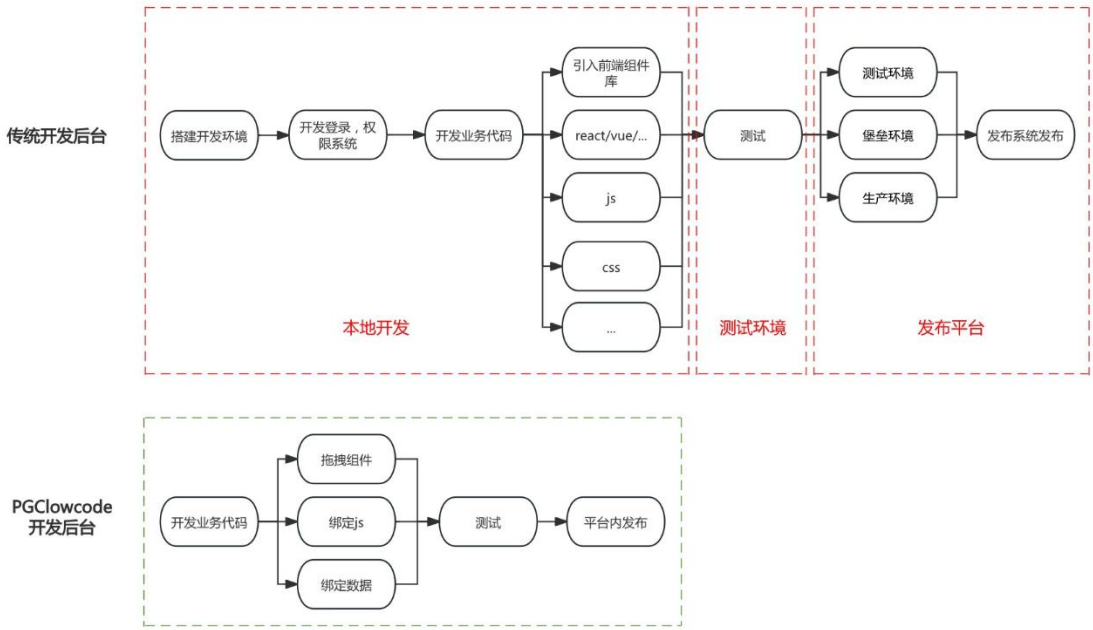
平台拥有自己的用户管理体系，为了与携程的账号体系打通，接入了 OIDC 域账号认证，使用者无需额外注册账号，只需要使用携程域账号登录即可。

用户的权限做了充分的拆分，平台的所有功能对每个用户开放，只是对于用户私有的数据做了权限控制，权限定义的最小单位是工作空间（workspace），用户可以拥有多个 workspace，每个 workspace 定义了管理员（admin）、开发者（developer）、使用者（viewer）、测试者（tester）、审核者（reviewer）五个权限组。其中每个权限组对 workspace 下面的资源拥有不同的管理权限，这些资源包括数据源、应用、页面等，admin 可以对 workspace 内的用户分配不同的权限组，从而对应用的开发、发布等流程上进行管理。具体各权限组的权限分配参考下表：

平台角色权限						
	编辑开发权限	查看应用	测试	审核发布	邀请其他开发	权限修改
管理员 Administrator	✓	✓	✓	✓	✓	✓
开发者 Developer	✓	✓			✓	
使用者 App Viewer		✓			✓	
审核者 Reviewer		✓		✓		
测试者 Tester		✓	✓			

3.2 可视化应用开发

传统后台开发与采用低代码平台进行后台开发的流程区别如下图所示：

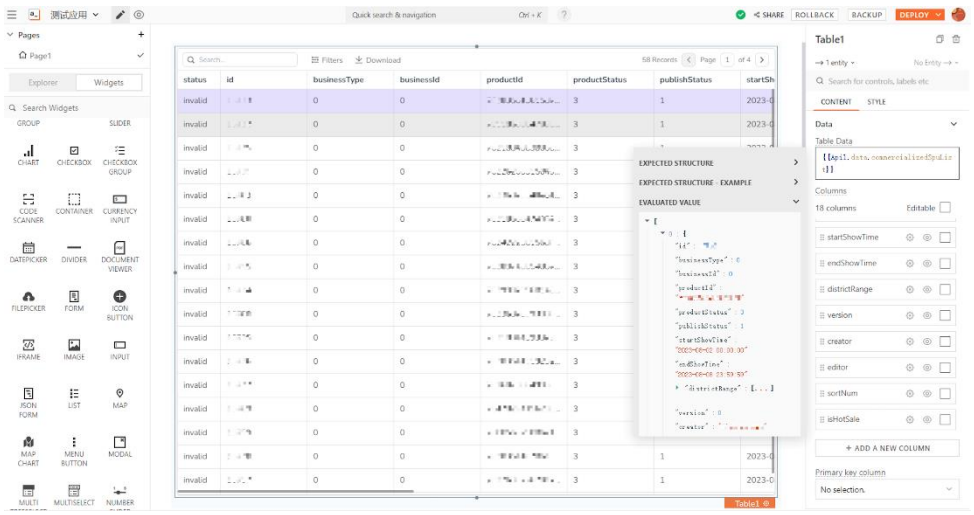


传统后台开发过程中需要开发者自身搭建开发环境，引入前端组件库如 Ant Design，相同的功能需要自己提取组件，开发效率低效。

PGClowcode 低代码平台提供了可视化拖拽的面板，支持页面复杂布局。组件栏支持 40+种通用组件，并可以组合使用。

在页面绘制方面，通过将其拖入画板，调整位置布局，简单几步完成界面的设计，做到了所见即所得。相同功能可以在画布中复制粘贴，应用本身也支持导入导出功能，方便项目复制。开发变得灵活高效，避免了一些基本构建所产生的 bug，达到了降本增效的效果。

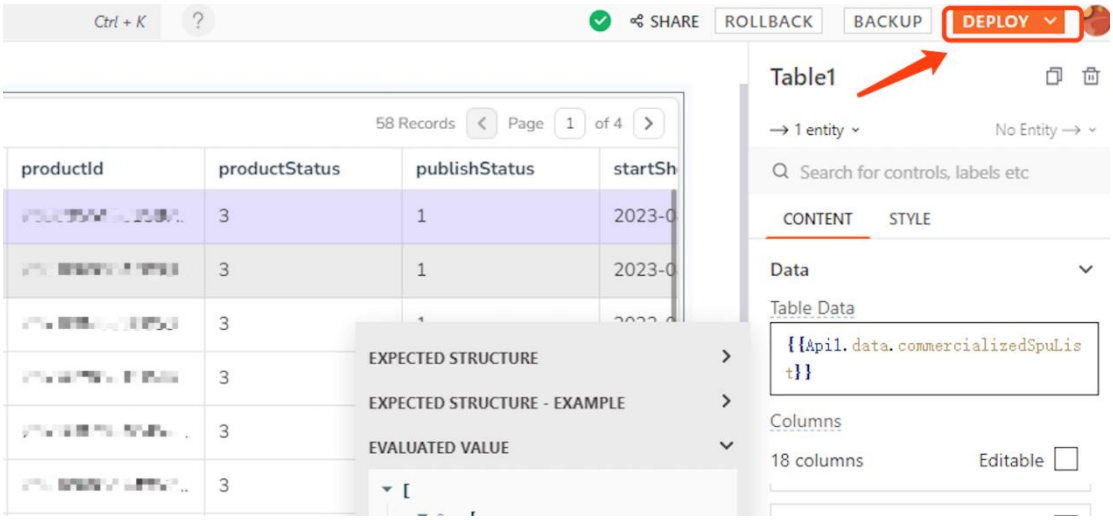
在组件的属性值设定方面，可以通过可视化的输入或者通过自定义 JS 代码的方式进行复杂的逻辑绑定，并且也支持编写 js 代码完成复杂的交互逻辑。平台内置了多种 js 库，可以将数据绑定到组件上，在开发状态下能立即看到数据渲染的效果，使得在预览状态下可以边开发边自测。



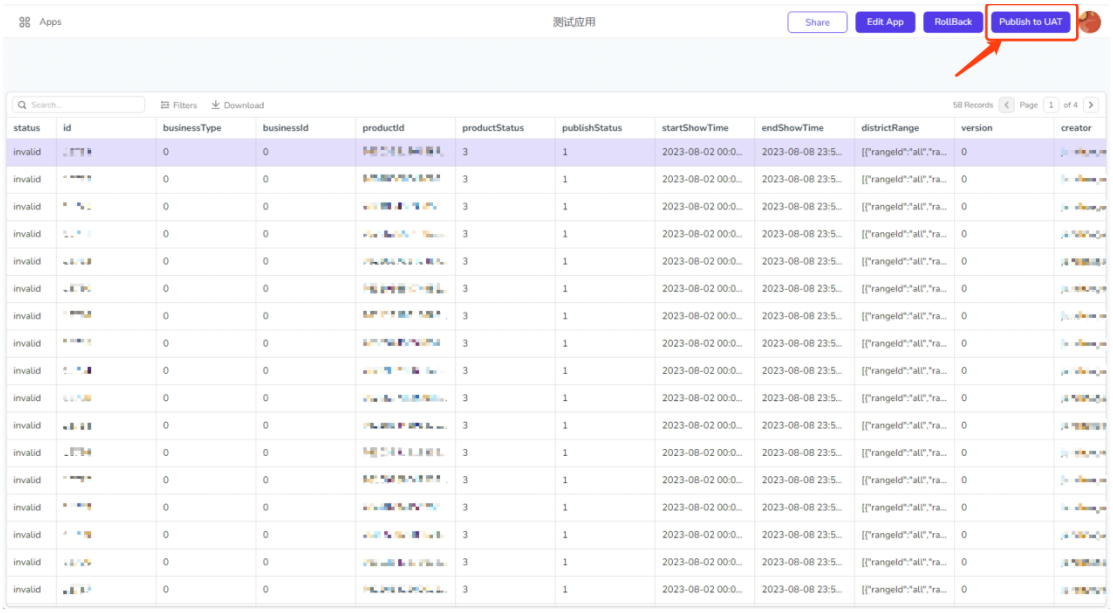
3.3 流程管理

应用从开发态->测试态->发布态的流转十分便利。平台设计了不同角色，将测试、审核的流程精简化，各角色登录后可以看到应用的不同状态，应用在开发和审批后自动流转至下一状态，只需要简单几个流程即可完成。

1）开发人员（开发态）：根据需求搭建、开发页面，然后发布到测试环境；



2）测试人员（测试态）：在页面测试，保证其能满足需求，且不存在质量问题后点击 Publish 提交发布申请；



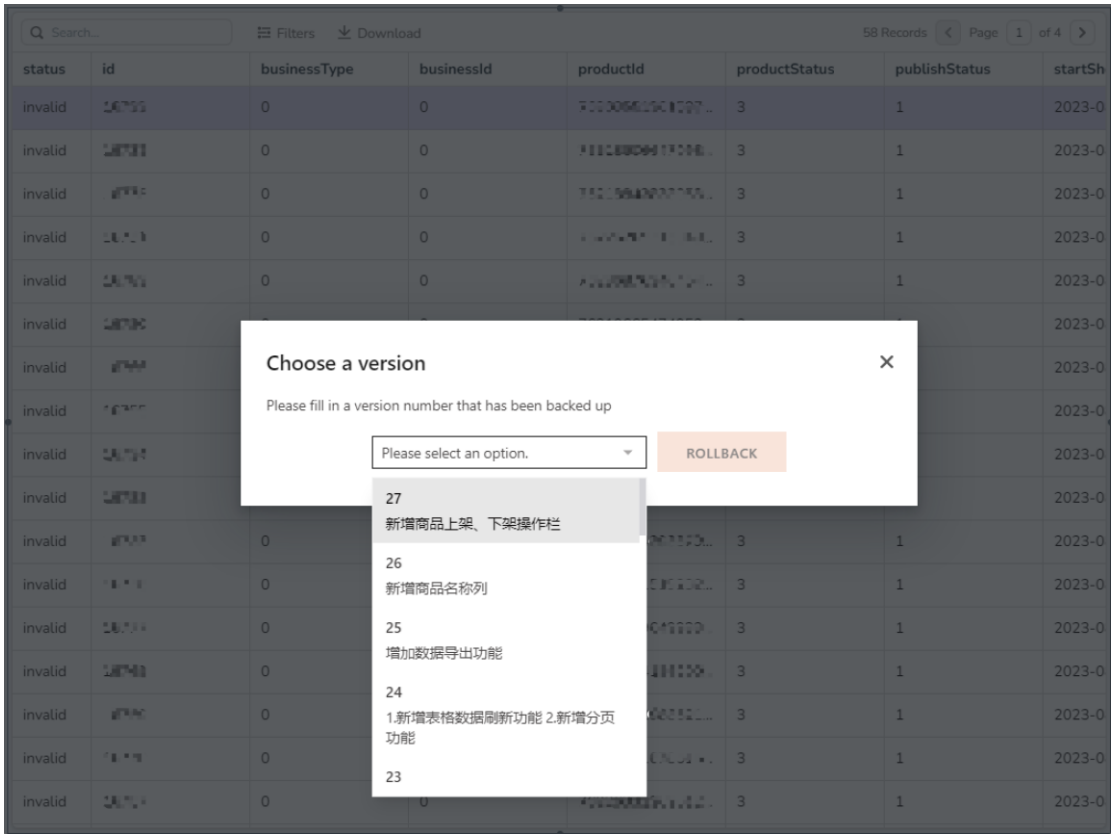
3）审核人员（发布态）：在“待我审核”列表中找到审核申请。审批通过，应用自动完成发布。

3.4 备份与还原

开发平台支持了应用数据的备份和历史版本数据的还原。在开发状态下平台采用了自动保存的设计方案，由于多人同时开发时容易出现相互覆盖或操作冲突的情况，为了减少这种问题导致的返工成本，我们设计了备份和还原的功能。

和正常普通的应用一样，用户可以将每个稳定版本的开发态备份到系统，在之后的操作中需要回到某个稳定版本则直接选中目标版本还原即可。

下图展示了备份还原的操作界面。

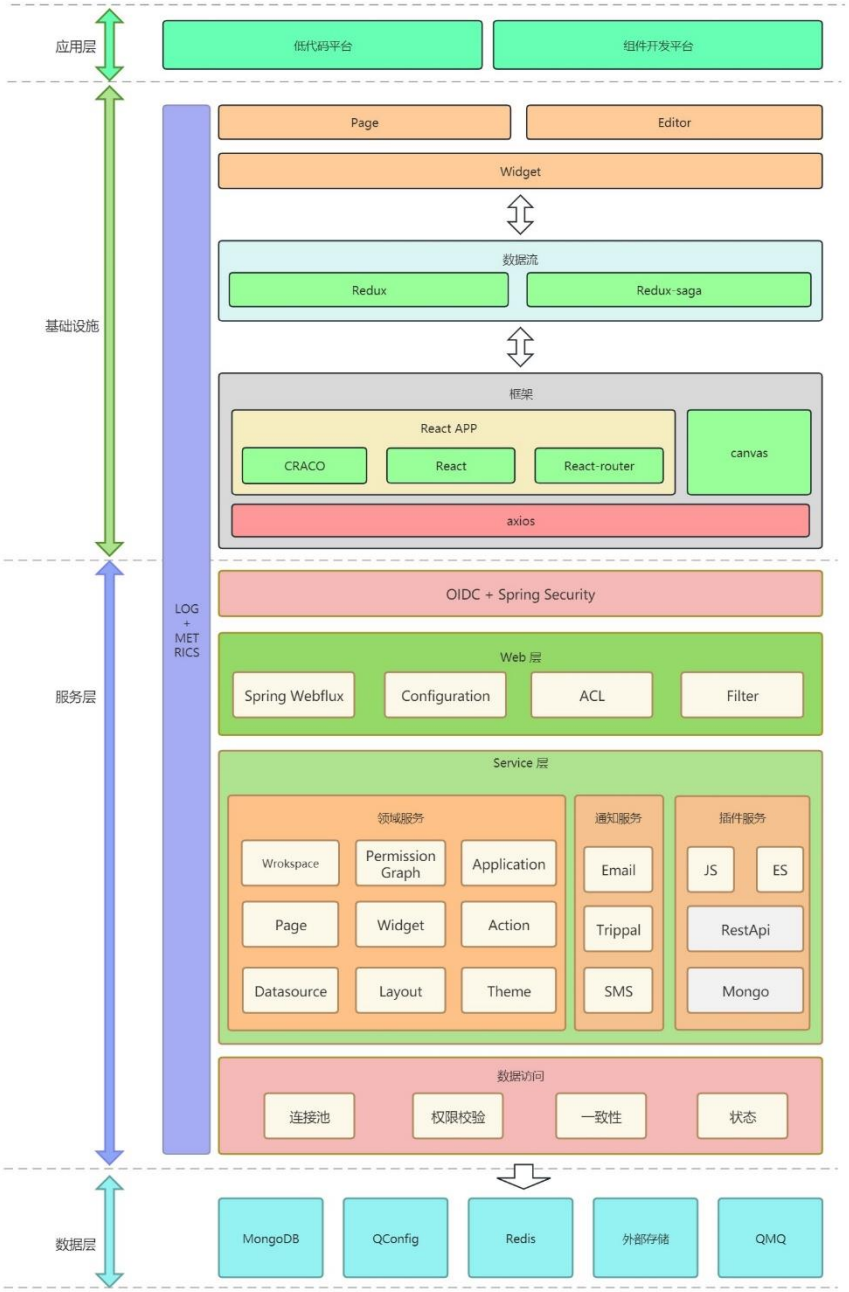


四、架构和技术

PGClowcode 平台采用了前后端分离架构。前端使用了 react 技术栈，并且集成了携程的多种公共框架和组件，依托于携程的 CI/CD 平台，实现了持续开发和交付的能力。服务端采用 spring webflux 框架，集成了 cat、clog（携程日志系统）、mongo、credis（携程 redis client）、qconfig(携程配置中心中间件，下文简称 QC)、qmq（携程 MQ 中间件）等技术框架，完全融入了携程的服务技术栈，可以通过 gitlab 自动化编译打包，在 captain（携程发布平台）

上发布。

4.1 架构



如上图所示，PGClowcode 平台的整体架构分为应用层、基础设施、服务层、数据层。

应用层是终端的两套平台系统，主要包括面向用户的低代码开发平台和面向开发人员的组件开发平台。

基础设施主要包含前端的基础框架、数据流控制以及抽象出来的前端可视的组件、页面和编

辑器等概念。基础框架主要依托于 React App 和 canvas 技术通过 axios 库和服务端进行数据交互，通过 Redux 及相关插件来控制整个平台的数据流，最终展示成用户可见的组件、页面、编辑器等 UI 模块。

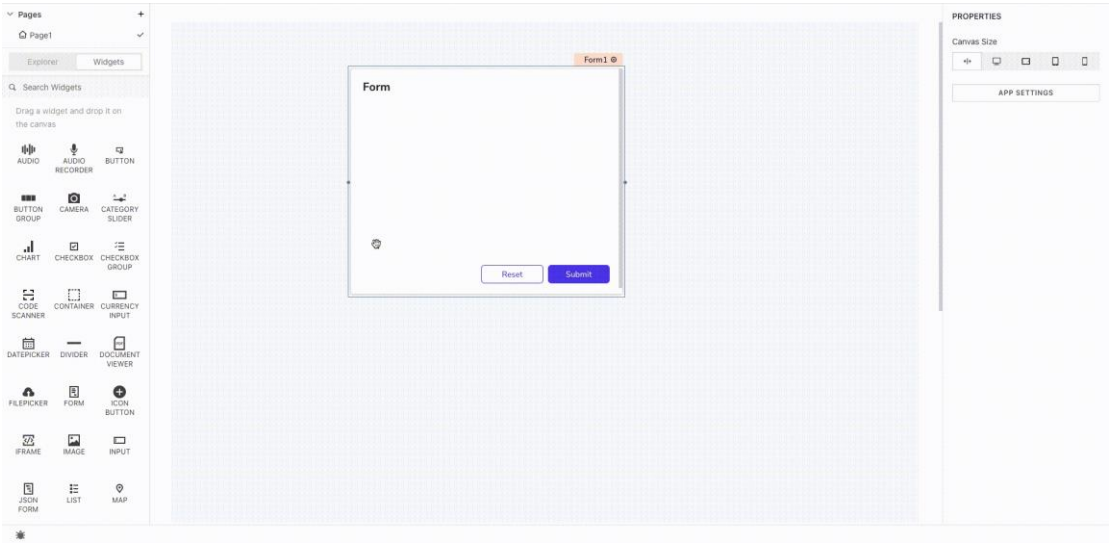
服务层主要由 web 层、service 层和数据访问层组成，主要提供权限验证、流程管控、插件管理、消息通知、数据访问等服务。服务采用了微服务架构的设计，按照不同的领域和功能拆分成领域服务、通知服务和插件服务。

领域服务又根据不同的 model 细分为不同的模块，各自完成独立单一的功能；通知服务主要用于 email、trippal、sms 等消息通知；插件服务主要提供插件的加载、初始化、调用、卸载等相关功能的服务。数据访问在核心服务的下层，实现了针对多种数据源的访问和数据处理、校验的功能。

数据层主要用于存储元数据、应用数据、插件数据等，应用的备份数据存储于 QC，并且通过 QC 实现跨环境发布。

下面两节主要对平台“组件的可视化拖拽实现”和“应用的开发-部署实现”两项比较核心的技术详细阐述。

4.2 组件的可视化拖拽实现



可视化拖拽组件是低代码平台的基本功能，在本平台中，用户可以在左侧组件库里选择任意组件拖拽到中间的编辑区域，并更改其大小。

在实现上述的拖拽功能时，我们会面临几个问题：

1）拖拽组件的过程中，组件的位置会实时变更，大量嵌套在一起的 dom 元素同时变更位置，会频繁触发浏览器的重绘制，导致性能的消耗。

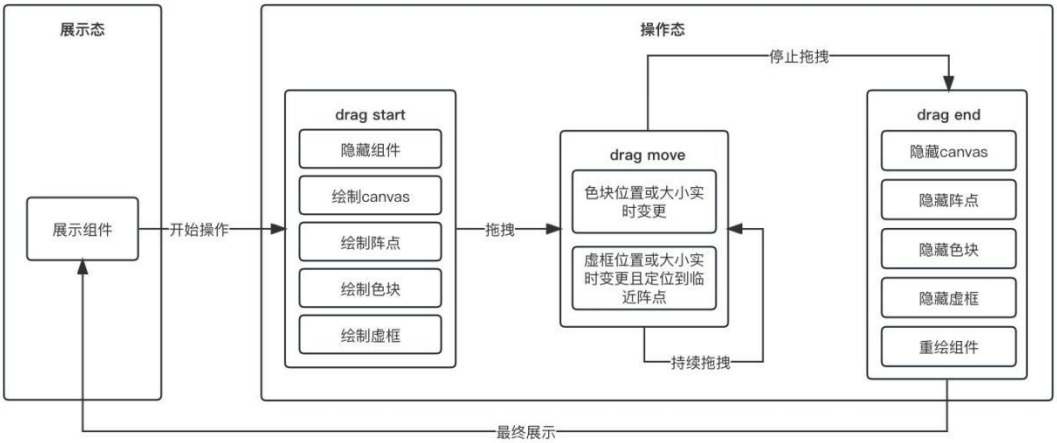
2）人为的拖拽排布，很难保证组件之间的对齐和排版，页面最终效果难以达到代码实现页面的规整程度。

对于上述的问题 1，平台的解决方案是依托 canvas 技术，在组件变更位置或者大小时，隐藏实际渲染在页面上的组件，并在编辑区域最上层渲染一层 canvas，在原组件位置画出一个同等大小的色块来代替原组件，并用以示意用户，利用 canvas 画布的特性来处理组件位置及大小的变更，在用户结束拖拽动作后，根据色块在 canvas 中的最终位置及大小，重绘制一次组件，并隐藏 canvas，展示出组件的最终效果。

在上述描述中可以看到，利用 canvas 可以极大程度的削减浏览器重绘制的次数，达到减少性能消耗的目的；选择色块来作为绘制对象而非原组件，既降低了技术实现的难度，又进一步降低了 canvas 绘制图形的性能消耗。

对于问题 2，平台的解决方案是阵点系统，当用户拖拽组件到编辑区域触发渲染 canvas 的同时，页面上会绘制一层阵点并均匀的平铺在 canvas 上，当用户在 canvas 上拖拽色块变更其位置或者大小时，在色块的周边会绘制出同等大小的虚线边框，边框会根据色块当前的位置及大小动态的定位到合适的临近阵点上。阵点系统中，横向及纵向两个相邻阵点的间隔即为组件长宽的最小单位，组件的四角位置只能定位在阵点上。由此可见，在拖拽过程中，组件的位置及大小都在一定的限制之内，这可以保证最终绘制出来的页面的规整性。

以下为一次完整的组件拖拽流程示意图：



1）页面无拖拽操作，主编辑区域仅展示组件的静态状态，此时为展示态。

2）当用户开始拖拽组件以期改变其位置及大小的动态状态，为操作态。操作态又可以细化为拖拽开始、拖拽中、拖拽结束，三个状态。

当用户开始拖拽组件时，页面会从展示态变更为操作态。拖拽开始时，编辑区域内绘制 canvas 并铺设阵点，原组件变为不可见，在 canvas 内原组件的相同位置绘制同等大小的色块以及虚线边框；在拖拽过程中，色块会随着鼠标移动变更位置或者大小，其外部虚线边框也会随色块移动或变更大小，并实时定位在色块当前位置的最相邻阵点上；拖拽结束时，根据当前

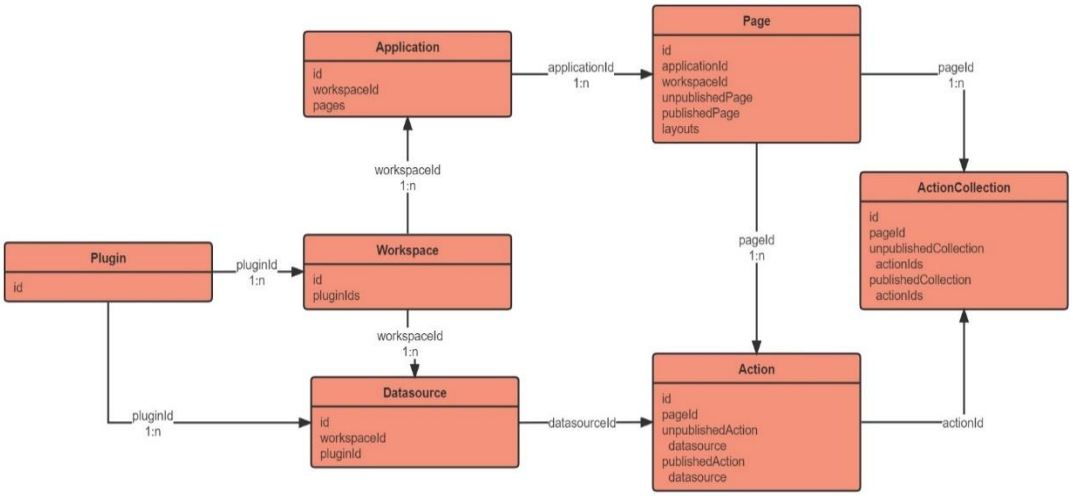
边框的最终大小及位置，重新绘制原组件，并隐藏 canvas、阵点系统、色块以及虚线边框；页面由操作态变更为展示态，展示出组件的最新状态。

4.3 应用的开发-部署实现

应用的开发和部署的技术实现主要分为应用的数据结构、数据流转和多种角色协同三个部分，最后针对应用的数据备份与还原也做了简单的介绍。

4.3.1 数据结构

当前大多数主流的低代码平台最终的产物可能是代码，但 PGClowcode 平台最终的产物是数据，包括应用信息、页面信息、组件、组件之间的关系、action、数据源等都是以数据的形式存储在 db 上，由于页面的布局、组件嵌套、函数的绑定等各种实体间关系非常复杂，使用传统的关系型数据库无法保证数据结构稳定，因此采用了 mongodb 作为数据库。应用相关的 collection 主要包括了 plugin、workspace、datasource、application、page、action、actioncollection，它们通过下图的关系构成了整个应用体系。



plugin 主要用于定义平台支持插件的元数据信息，包括类型、插件包路径、参数、状态等属性。

workspace 是应用开发的工作空间，它定义了空间内的用户组成、用户权限、数据源以及支持的插件集合。

application 定义了应用的名称、主题、icon、状态等基本信息，另外为了查询便捷，也冗余了部分页面的信息。

page 定义了页面的名称、布局、组件、组件的关系、组件与函数的绑定关系等。

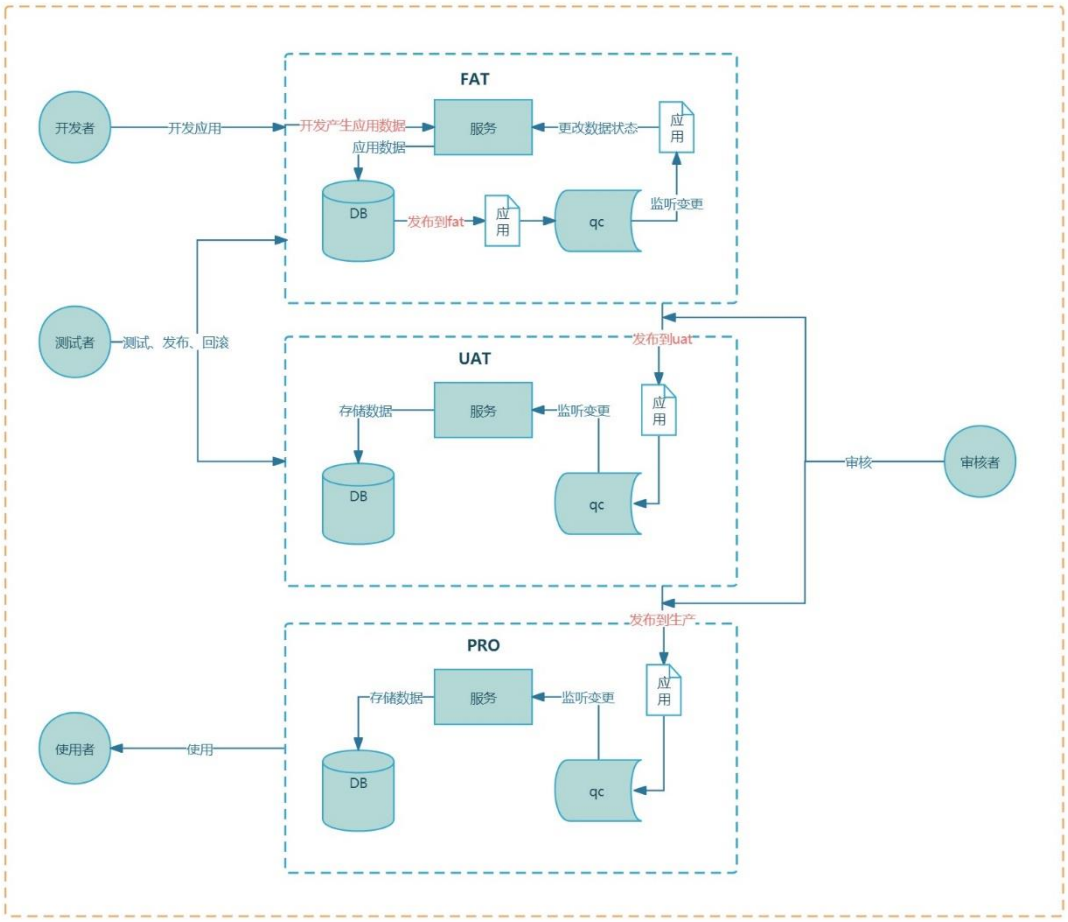
datasource 定义了外部数据访问的基础信息，如 restapi、es、mongodb 等外部数据访问的必要属性，为了避免重复配置，它的作用范围是 workspace 级别。

action 定义了外部数据访问的具体配置数据，它必须依托于 datasource，如 restapi 接口调用，datasource 配置了服务的域名和是否需要实时鉴权，而 action 则配置路径、参数、运行方式，与 datasouce 不同的是它的作用范围是页面。

actionCollection 是 action 或 js 函数的集合。

4.3.2 数据流转

应用数据主要是在不同的存储介质和不同的环境间流转，平台目前设计了三套环境 FAT、UAT、PRO，平台通过 QC 的跨环境机制实现数据流转。



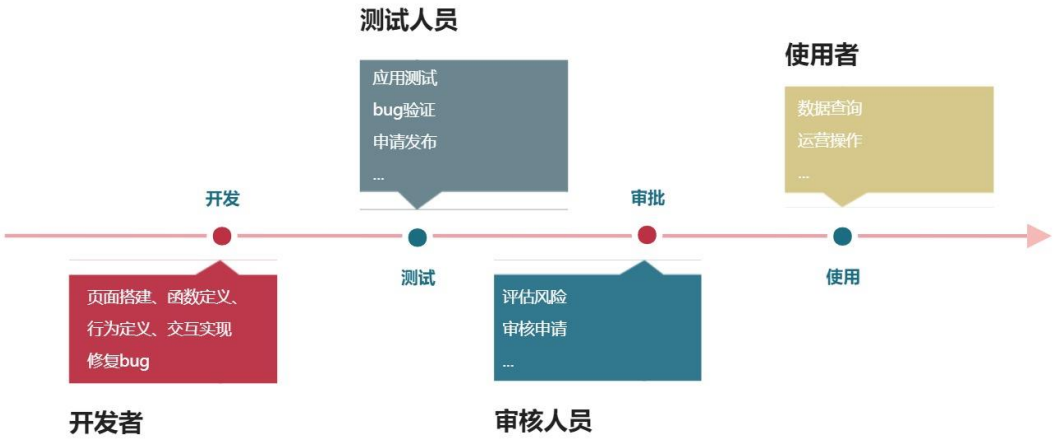
如上图所示，开发者在 FAT 上开发应用，应用数据以草稿态存储于 DB，开发完成后将草稿态数据导出到 QC 的 FAT 环境，服务监听到 QC 的数据变更将草稿态 copy 到发布态，测试则可以在测试页面上看到开发提交的应用，测试完成后提交 UAT 发布申请，服务将发布态的数据导出到 QC 的 UAT 环境，此时审核者将收到待审核通知。

进入平台将待发布申请审核通过后，UAT 环境的服务监听到 QC 数据变更将数据导入到 DB，并且将应用数据状态置为发布态，然后可以在 UAT 的测试页面看到发布态的应用，当 UAT

测试完成后申请发布到生产（PRO），UAT 服务将发布态应用数据导出到 QC PRO 环境，当审核者通过申请后则 QC PRO 的应用文件被发布，PRO 服务监听到数据变更就将应用数据导入到 DB，并置为发布态，此时应用的开发-部署流程结束，用户可以在生产环境的用户平台正常使用了。

4.3.3 多角色协同

对于应用的开发、测试、交付平台设计了多个角色，在整个需求开发的过程中每个角色能各司其责，保证应用能持续、稳定、高效迭代交付，如下图所示。



平台通过定义多个权限组来区分每个角色，当 workspace 被创建时这些权限组就会被创建出来，每个权限组定义了各自的权限，在每个受权限管理的资源中都有 policies 字段，它存储了能被操作的类型和权限组 id 的集合，当用户查询和操作时都会经过权限校验。

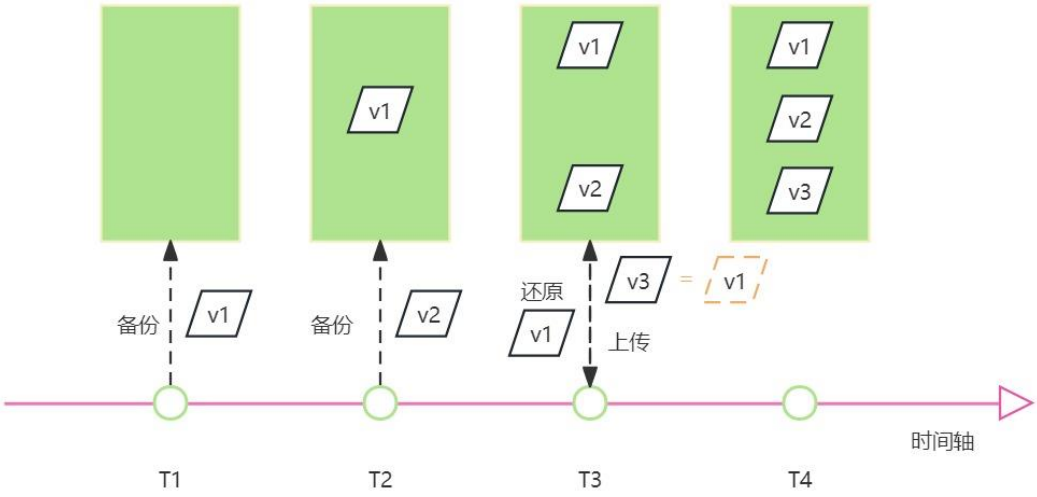
```
权限校验
Flux<PermissionGroup> findByWorkspaceId(String workspaceId, AclPermission permission); // 查询workspace时需要传权限类型AclPermission

权限数据
"policies": [
        {
            "permission": "query:workspace", // 查询workspace权限
            "permissionGroups": [ // 拥有这个权限的权限组id
                "63ad5afd2709b133ea69ee66",
                "63ad5afd2709b133ea69ee67"
            ]
        }
    ]
```

有了角色的定义，应用开发人员的协同就变得简单多了，如当应用发布测试时可以自动通知测试人员介入，提交发布生产申请时可以自动通知审核人员参与审核。

4.3.4 备份与还原

为了方便开发过程中多人同时开发，平台设计了备份还原功能，当用户点击备份时，服务将

当前草稿态数据导出到 QC，点击还原则将 QC 的数据导入到服务，覆盖当前 DB 中草稿态数据，得益于 QC 的版本管理功能，每次备份的数据都将存储起来，因此用户可以还原到历史的任意版本。



如上图所示，T1 时刻新增一个备份 v1，T2 时刻 QC 中存在 v1 版本的备份数据，如果 T2 时刻再增加一个备份 v2，到 T3 时刻 QC 存在 v1 和 v2 两个版本，如果在 T3 时刻将 DB 中的版本还原成 v1，则 DB 中的数据会被还原成 v1，与此同时会上传一个 v3 版本到 QC，实际上 v3 和 v1 数据是一样的，相当于将当前数据的基准切换到了最新版本，之后的操作都是在这个版本的基础上做的，如果再需要还原到这个基准就可以快速完成。

五、规划与展望

目前 PGC 低代码平台已经具备了非常完整的功能，基本上完成了预期的目标，也产生较大的价值，但我们对于它的期望绝非只限于此，并且组建了稳定的支持团队，制定了明确规划，在之后的迭代开发中会不断地完善已有的功能和流程，而且会根据实际的需求和业内平台的调研继续增加更强大、便捷的功能。

5.1 搭建组件扩展平台

平台原有的组件是比较基本的组件库，未来会难以满足日渐复杂的业务需求，因此自定义组件的需求就会逐渐凸显出来，本平台基于 Appsmith 框架是支持自定义组件的，但是原有的自定义功能有如下几个问题：

1) 原有的自定义组件功能，需要依托于完整的平台项目，在其 widget 文件夹下开发新组件，项目代码体量大，启动慢，且以开发组件为目的频繁更改提交平台项目并不利于平台项目的发布及管理。

2）原有的自定义组件功能并不适合多部门自定义组件的场景，没有相关的权限管理系统，

所有自定义组件都会展示在页面上，随着时间的推移会造成组件库的臃肿以及增加编辑页面时查找使用组件的费力度。

为了解决以上问题，我们会从主项目中抽离出相关代码，搭建一套独立的组件开发项目，以达到和平台主项目分离、代码纯净、项目快速启动的目的。同时也会构建一套自定义组件的权限管理系统以便更好的管理各部门开发的自定义组件。

5.2 建立模板库

Ctrl+C 和 Ctrl+V 可能是开发人员使用频率最高的按键组合，致使一些键盘不是“面目全非”就是“半身不遂”，试想一下，如果我们拿出来一键生成部署页面的功能，是否能让“久经磨难”的键盘依然“历久弥新”呢？没错，这是我们接下来的目标。

低代码平台的模板是指根据长期积累的经验，总结一些具有共性的通用方案，并且提炼成所有用户都能直接使用的页面或应用，收录到模板库中，当平台上的其他用户需要使用类似应用或页面时，只需要找到合适的模板，一键即可生成页面或应用，甚至能拿来即可用，无需做任何修改。后期可以允许建立团队自己的模板库，各成员可以搭建自己的模板专门供团队使用。

数据库

携程 MySQL 迁移 OceanBase 最佳实践

【作者简介】提挈，携程资深数据库工程师，专注于数据库自动化运维和分布式数据库的研究。Cong，携程数据库专家，主要负责 MySQL 和分布式数据库运维及研究。Typhoon，携程高级数据库工程师，负责分布式数据库的运维和工具设计。

一、前言

MySQL 在业界流行多年，很好地支撑了携程的业务发展。但随着技术多元化及业务的不断发展，MySQL 也遇到了新的挑战，主要体现在：业务数据模型呈现多元化，OLTP 和 OLAP 出现融合的趋势；在 MySQL 数据库上慢查询治理成本高；使用传统的分库分表方案对开发不友好，核心数据库改造成分库分表方案，时间一般以年为单位。

分布式数据库能比较好地解决上述问题，同时也带来了新的挑战。2021 年，OceanBase（简称 OB）开源，携程开始逐步探索 OceanBase 的基本特性和应用场景。OceanBase 兼容大部分 MySQL 的功能和语法，同时提供水平扩展性、强一致性和高可用性，能满足业务需求并降低运维成本。因此，我们开始推进部分 MySQL 实例迁移到 OB。为保证迁移顺畅，我们设计了迁移评估工具、OB 迁移流程、OB 监控大盘和 OB 故障诊断工具等。并将迁移过程中遇到的问题和大家进行分享。

二、评估工具

平滑迁移异构数据库，我们需要进行兼容性、性能和分区适应性等各项检查。提前把不兼容或有可能引起迁移异常的场景找出来并解决。官方提供了 OceanBase Migration Assessment(OMA)工具，用于异构数据库迁移到 OB 的可行性评估。迁移评估工具 OMA 有语法兼容性检查和性能评估，但还不能完全满足我们的需求。主要体现在下面几点:

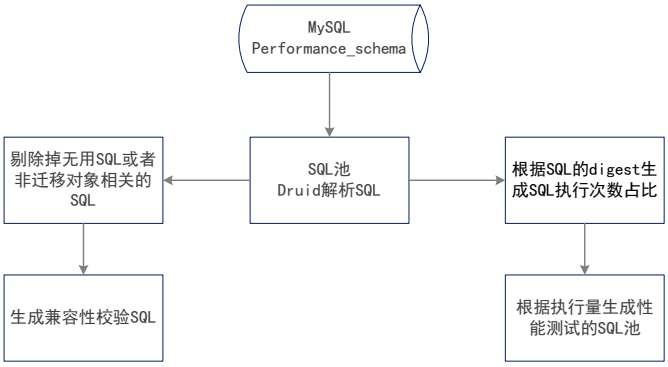
- 中间件版本检查，一个 DB 有多个应用在访问，只有某个版本后的中间件才开始支持 OceanBase，需要检查访问该 DB 的所有应用的中间件版本，并督促开发进行升级，以确保都在支持 OB 版本之上。

- 性能采集和回放提供的 MySQL General Log 采集模式有一定风险，尤其是对于业务繁重的数据库，我们需要更平滑的性能采集和回放方案。另外对于单实例多 DB 场景，存在迁移和不迁移的 DB 共存的情况，需要进行过滤。

- 线上存在非通过中间件访问的数据库账号，如 ETL 取数账号、数据查询工具账号、应用直连账号等，对其兼容性需要进行检查。因为迁移到 OB 之后，数据库登录账号需要进行改变，包含租户信息。

- OceanBase 是分布式数据库，数据如何进行分区就显得非常重要，以避免形成热点数据。一张表可能有多个字段都适合作为分区键，在迁移工具中，根据数据分布以及访问情况，需要提供表分区推荐，以减少迁移成本。

因此我们对 OMA 评估工具进行了拓展和改造。在不影响现有的数据库运行下，省去中间环节，做到一键评估。其中 MySQL 数据采集与分析大致流程示意图如下，全量数据导入 OceanBase 后，目标端我们用开源 Locust 工具，进行 SQL 回放和压测，并最终形成评估报告。



三、迁移流程

在评估流程完成并且评估结果符合迁移要求的前提下，可以发起 MySQL 到 OceanBase 自动迁移流程。为减少迁移成本，我们把迁移流程进行了封装，做到一键自动迁移，自动切换包含以下流程:

1）迁移前配置校验。迁移前，会集中对所有的切换注意事项和相关配置再进行一次全面的检查，提前排除配置问题可能导致的切换风险。

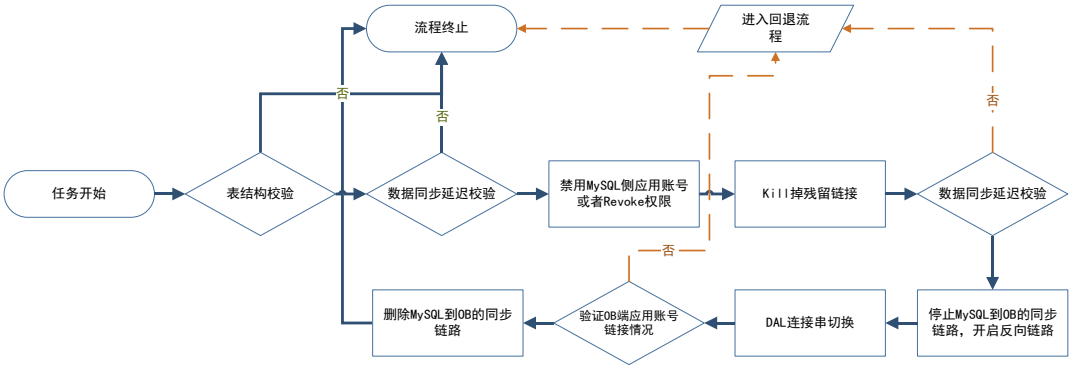
2）MySQL 账号兼容 OceanBase 带租户账号创建。由于 OceanBase 是多租户管理模式，应用的连接串必须指定租户名，因此相应账号需要在目标 OB 集群预先创建，中间件或工具切换账号时，只需重置连接并切换到新账号即可。

3）数据一致性校验。数据通过 Canal 从 MySQL 同步到 OB 后，我们需要对一致性做校验。校验的方法是根据表主键进行切分，进行结果集比较是否一致。当遇到热点表时，数据校验过程会发起多次尝试来反复验证。

4）DDL 表结构修改暂停。由于 MySQL 和 OceanBase 表结构变更方式差异较大，当 DB 迁移从 MySQL 到 OceanBase 触发流程后，我们会在源 MySQL 禁止 DDL 操作。当然，如果开发有紧急发布需求，我们可以废弃流程，等 DDL 发布完成后，再重启迁移流程。

5）反向同步链路搭建。无论前面的迁移评估或者流程多么完善，反向同步链路对于异构数据库的迁移是必备的。一旦迁移出现异常，可以快速回退。反向同步链路是基于 OceanBase 的 CDC 服务，订阅增量日志在 MySQL 端回放，保证迁移后 OceanBase 侧和 MySQL 侧数据始终一致。

当数据同步完成，并且没有增量延迟后，迁移流程将生成具体的切换任务，切换流程如下:



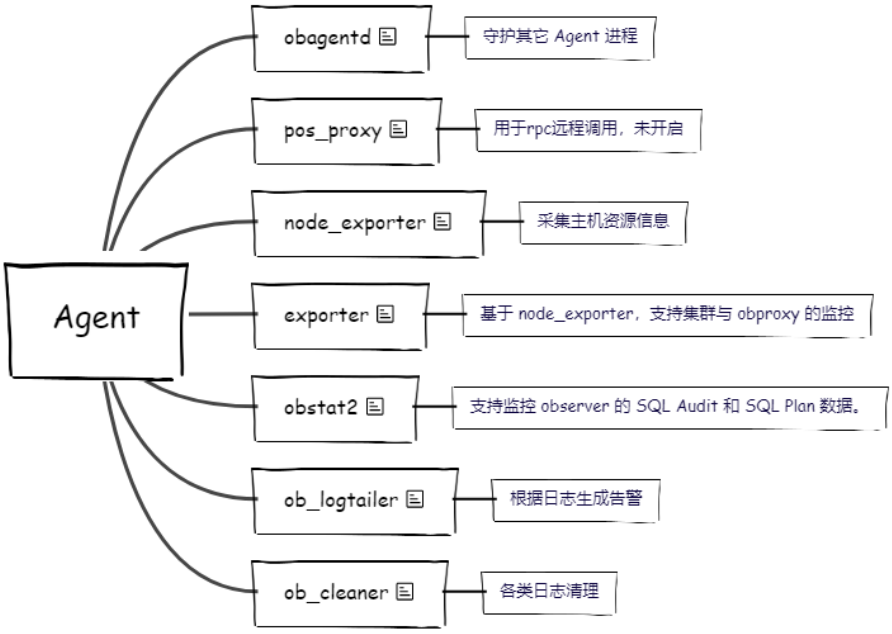
我们只需要在预定的时间窗口内，点击触发切换流程，就可以完成从 MySQL 到 OceanBase 的切换。整个切换流程可在一分钟之内完成，而且业务端无需进行改造。我们拥有反向链路，如碰到有异常情况，可以随时安排回退。反向链路在正常情况下将保留两周以上。

四、OceanBase 监控

分布式数据库和单机数据库一个比较大的区别在于分布式监控比单机版数据库更为复杂。一是因为组件众多，需要有一个全局视点；二是因为需要对告警点进行聚合。业务新迁移到 OceanBase 时，观察集群监控、关注告警信息是判断迁移成功与否的关键。日常的冒烟现象或者不规范现象，需要及时发现、及时处理，避免问题恶化。准确监控和及时告警可以帮助运维人员快速定位问题，快速解决故障。

4.1 监控大盘

OceanBase 的监控数据主要通过在每台 Server 上部署的 Agent 程序从本地直接采集。Agent 中包含众多组件，内容如下：



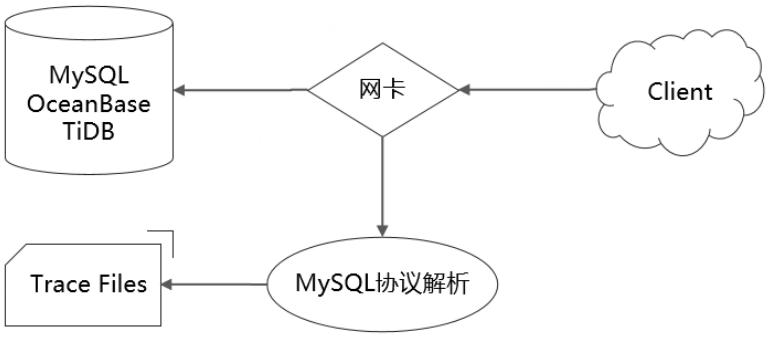
Agent 程序会向 hickwall 上报采集到的数据，以模板化的形式展示出来，以此形成监控大盘。如下图所示：

4.2 告警邮件

OceanBase 的告警，主要通过订阅 hickwall 上的监控数据以及定时的服务巡检来完成。基于采集的监控数据设立告警阈值，一旦指标超过阈值便会进行告警通知。另外，我们还会对配置进行定期检查，来解决规范性问题等。

4.3 OceanBase SQL 审计

OceanBase 接入了携程的 SQL 审计流程。与以往传统的审计插件模式不同，现在以抓取网络包的方式，通过对 MySQL 协议解析得到全量的 SQL 审计信息。接入审计流程后，可以快速定位到 SQL 信息，包括应用编号、访问 IP、执行参数、有无报错信息等。



4.4 OceanBase 审计运用案例

在使用 MySQL command-line tool 连接 OceanBase 过程中出现连接不上的错误时，我们使用 SQL 审计日志进行定位，发现客户端在连接 OB 的过程中会执行一些元数据查询工作，在

进行 show tables 这一步骤后会报错断连，后续定位到一个特殊的表，该表表名的最后一个字符是分号（t_sample;）导致了这次报错，随即我们在开源社区反馈了这例问题。

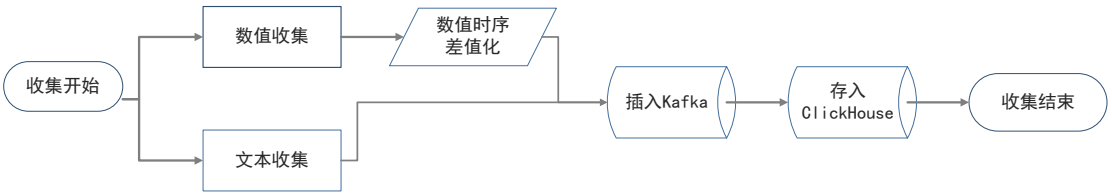
```
time                    client          thread_id       user            database    exec_time   aff_rows    SQL
2022-12-12 15:18:57.806880  192.108.114.158:56690   1190118     yc@myproxy#mycluster    NULL    0.00    0   [@sq1@ [LOGIN] @sq1@]
2022-12-12 15:18:57.812751  192.108.114.158:56690   1190118     yc@myproxy#mycluster    NULL    0.00    0   [@sq1@ select @@version_comment limit 1 @sq1@]
2022-12-12 15:19:00.758949  192.108.114.158:56690   1190118     yc@myproxy#mycluster    NULL    0.00    0   [@sq1@ SELECT DATABASE() @sq1@]
2022-12-12 15:19:00.759052  192.108.114.158:56690   1190118     yc@myproxy#mycluster    NULL    0.00    0   [@sq1@ use obxdb @sq1@]
2022-12-12 15:19:00.766526  192.108.114.158:56690   1190118     yc@myproxy#mycluster    obxdb   0.01    0   [@sq1@ show databases @sq1@]
2022-12-12 15:19:00.777371  192.108.114.158:56690   1190118     yc@myproxy#mycluster    obxdb   0.01    0   [@sq1@ show tables @sq1@]
2022-12-12 15:19:00.799260  192.108.114.158:56690   1190118     yc@myproxy#mycluster    obxdb   0.02    0   [@sq1@ desc table1 @sq1@]
2022-12-12 15:19:00.832549  192.108.114.158:56690   1190118     yc@myproxy#mycluster    obxdb   0.03    0   [@sq1@ desc table2 @sq1@]
                                                                                                    ERROR 2006 (HY000): MySQL server has gone away
```

五、OceanBase 自动故障诊断

随着越来越多的 MySQL 迁移到 OceanBase，数据库性能、故障定位的实时性和准确性的要求变得越来越高。自动故障诊断系统可以全方位、及时、精准地定位线上问题，为运维和排障提供依据。

5.1 构建实时性能数仓

OceanBase 性能数仓构建的流程图如下：



- 收集性能指标相关数据，以下是常用的性能指标对应的数据源：

服务器维度	CPU
	I/O
	网络流量
操作系统维度	linux相关计数器
	linux错误日志
数据库服务维度	OceanBase错误日志
数据库系统视图/内部表	gv$plan_cache_plan_stat
	gv$partition_audit
	gv$sysstat
	gv$sql_audit
	__all_rootservice_event_history
	global_variables
	global_status
	Processlist
	.....

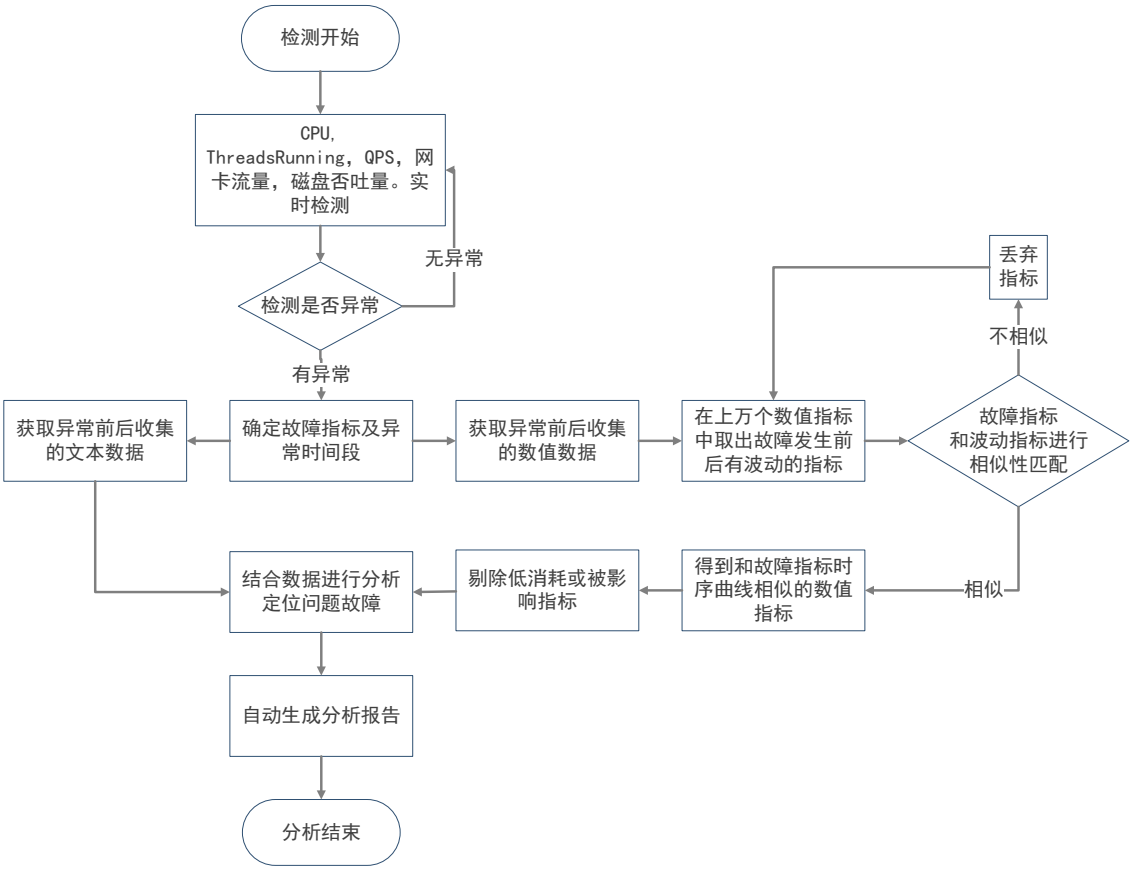
- 开发数据收集程序，在服务器本地每 10 秒采集一次上述性能指标的数据。并在采集之后对数据进行结构化处理，包括对数值型数据进行标准化处理，对文本型数据进行时序

化处理。

- 将结构化处理之后的数据落地存储到 ClickHouse 中。

5.2 自动化分析

自动化分析的流程图如下:



5.3 实时检测性能指标

通常判断性能异常的指标包括 CPU 占用率、磁盘 IO 占用率、Threads Running、QPS、网卡流量等。基于运维经验，可以针对每个指标设定相应的阈值，当突破阈值时，则认为当前实例存在性能问题。比如 CPU 占用率高于 65%或磁盘 IO 占用率高于 80%则代表服务器出现异常。

5.4 异常数据匹配数仓

首先，对于数值型数据，分析工具会自动选取故障指标和故障时间段，通过相似性匹配数仓中数据所有数值型数据包含 SQL、Table、Perf 三种类型，它们相关的性能指标说明如下:

• SQL 对应的性能指标：执行次数、总耗时、CPU 耗时、逻辑读次数、物理读次数等

• Table 对应的性能指标：增删改行数、增删改的 SQL 数、相关事务数等

• Perf 对应的性能指标：CPU、I/O、RPC 时长、索引缓存大小、缓存命中率等

其次，对于文本型数据，分析工具会通过故障时间区间获取所有时序化的文本数据，通常包含：数据库服务日志、系统内部任务记录、数据库进程信息等。

最后，基于前面两种类型的数据进行综合性分析，分析要点主要有：

• SQL 层面：SQL 性能消耗占比、有无正在执行的慢 SQL、是否缺失索引、是否存在远程执行或分布式执行等

• OceanBase 内部：OceanBase 是否在做合并、是否正在均衡副本、是否存在其他异常日志等

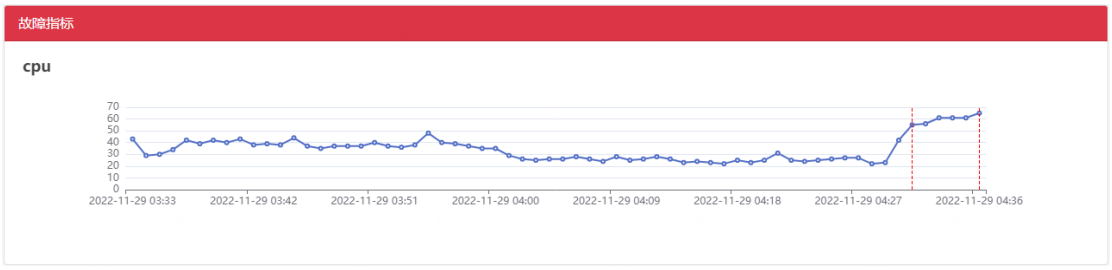
• 应用层面：客户端是否进行发布

最终基于以上自动化分析，实现服务器性能波动真实原因的精准定位，自动生成故障定位分析报告，并通过邮件及时推送给 DBA 和相关开发人员。

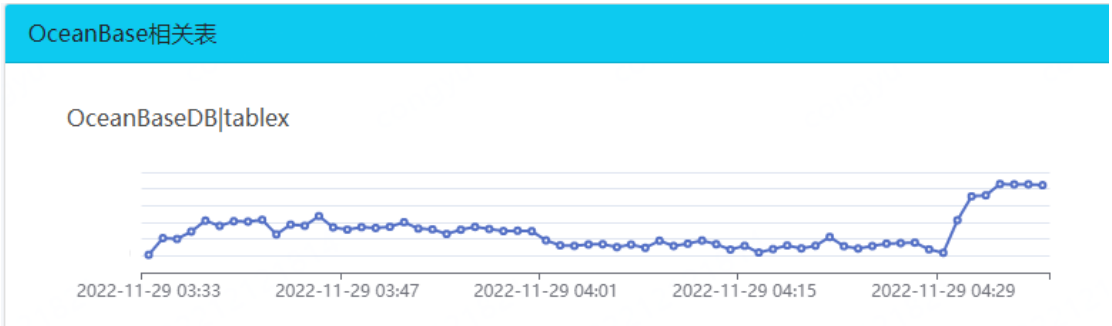
5.5 运用案例

下面基于该工具自动生成的一例分析报告来介绍该工具的实际运用：

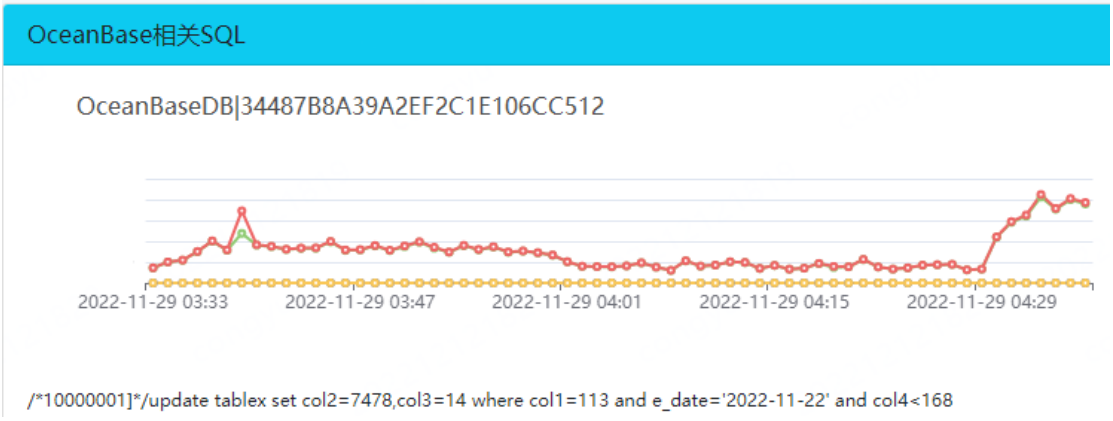
1）报告的故障指标板块显示 4:30 后服务器的 CPU 上升；



2）报告的 OceanBase 相关表板块显示 CPU 上升趋势和下面这张表的访问趋势一致；



3）报告的 OceanBase 相关 SQL 板块显示这张表的访问趋势和下面的 SQL 语句访问趋势一致；



4）报告的分析结果板块定位到 CPU 上升和 tablex 表的访问上升有关，而这张表的访问上升又和这 1 条 SQL 语句访问耗时增长有关，最终定位由于该 SQL 导致 CPU 上升。后续我们联系开发确认是正常业务上升，并添加服务器节点缓解 CPU 负载。

六、迁移遇到的问题和实践

6.1 .Net 应用访问 OceanBase 失败

在使用和测试 OceanBase 的过程中，我们发现.Net 应用的官方 MySQL 连接器连接 OceanBase 执行 SQL 失败。

经排查，我们发现.Net 应用依赖连接中的 ConnectionCharSetIndex，而 OceanBase 不存在 ConnectionCharSetIndex=83 即 utf8_bin，只有 utf8mb4_bin。因此我们对 OceanBase 的源码进行了修复来满足这类应用对 OceaBase 的适配性。

总结：OceanBase 不够完美，但是随着时间推移，通过反复的测试和迭代，正在逐步完善它的各方各面。我们也参与其中，以运维和产品使用者的视角对它进行优化和完善。

6.2 Druid 应用不兼容部分 OB 语法解析

我们在开发 Oceanbase 表结构设计工具的时候，发现 OceanBase 的 SQL 通过 Druid 解析时存在报错。这个错误会导致在表结构设计的时候导入 SQL DDL 语句报错。遇到问题后，我们先调整到 Druid 最新版本，发现问题仍然存在。

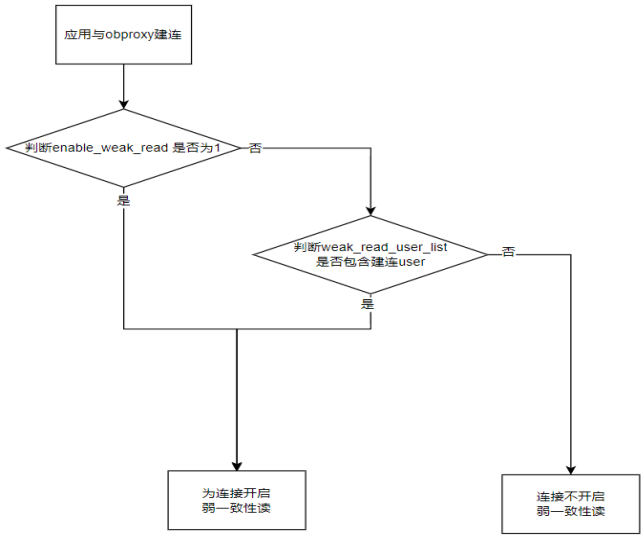
我们将问题先从复杂的表结构设计中抽离出最简单的 SQL DDL，并结合分析 Druid 的源代码，发现原来 Druid 代码对 OceanBase 的兼容在 SQLIndexDefinition 中实现，但没有在 SQLIndexOptions 实现。根据 OceanBase 的语法树，实际应该在 SQLIndexOptions 实现才合理，找到问题所在后，我们提交了 Pull Request，然后被合并到 Druild 主线。问题得以解决。

总结：开源工具的一个好处在于碰到问题后我们可以进行代码分析。并快速定位问题，最后反馈社区。

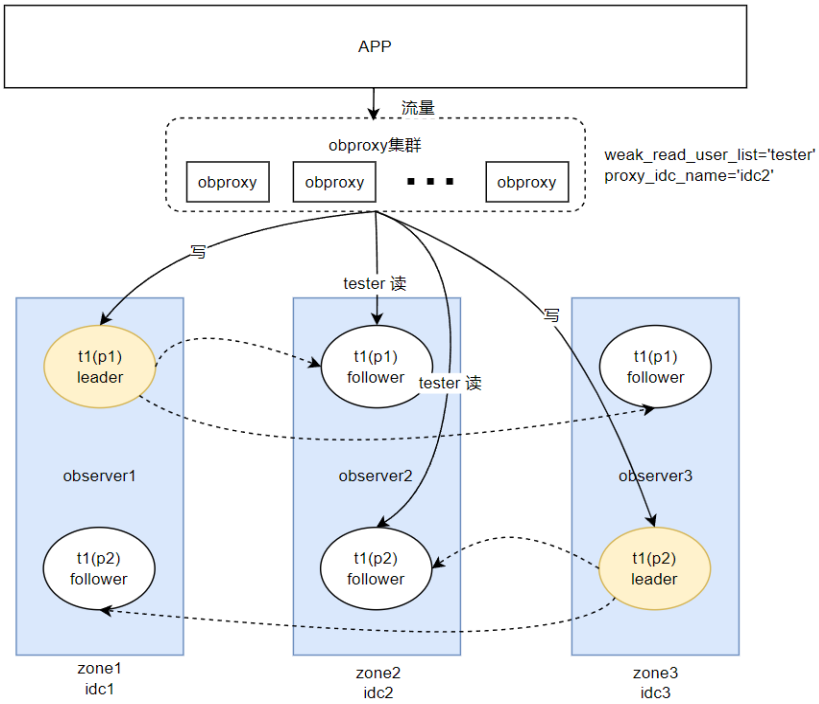
6.3 OceanBase 读写分离支持

读写分离是数据库非常重要的能力，在业务层面上，它覆盖到了 ETL 取数，BI 报表生成，缓存刷新等多个场景。Oceanbase 虽然支持读写分离功能，但需要在代码层显性设置弱一致性读参数，存在对业务高度侵入的缺陷。我们对 OceanBase 访问代理 OBProxy 做了代码改造，新增 enable_weak_read 以及 weak_read_user_list 两个参数，通过代理层控制开启读写分离策略，对应用透明度高。

读写分离场景下，应用与 OBProxy 建联的流程示意图如下：



基于以上的代码修改，我们设计了一套优化版本的读写分离方案，即通过以账号维度来控制是否使用读写分离。流量调度示意图如下：



总结：OceanBase 源生虽然提供了强大的功能，但是它并不一定 100%满足业务的场景和需求，因此对其组件进行二次开发是有必要的。我们不单单对于 OBProxy 进行了相关的功能适配，对于其他组件如 cdc、Deploy 组件等我们也根据实际场景需求，进行相应调整。

6.4 query range 过大导致内存溢出

在初期使用 OceanBase 时，我们有碰到过 Server Crash 的经历。当一个查询的条件中 IN 运算符中包含过多元素（一万级别以上）时，会爆出 stack overflow 的异常。

经过分析和社区交流，我们定位到优化器在抽取 query range 会耗费大量的内存。而 OceanBase 在算法迭代过程中没有检查查询超时，导致该查询一直消耗内存，直到用尽了 SQL ARENA 的内存。这种模式没有做好防御机制，从而导致内存溢出造成系统崩溃。这个问题在新版本中已经得到修复。当确认到问题后，我们第一时间通知开发减少 IN 内的元素数量，并安排了版本升级。

总结：OceanBase 作为新鲜产品，社区论坛和 Git issue 是获取日常运维和快速排障方案的利器，根据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。

6.5 修正执行计划

在迁移前后，数据库的 SQL 性能是最值得关注的地方。作为分布式数据库，OceanBase 的优化器相较于 MySQL 来说更复杂并且由于其特殊的存储结构导致表的统计直方图刷新频率很低，因此当可用索引和查询条件的适配度不高时，优化器在选择执行计划时可能存在偏差。

OceanBase 自带修正执行计划的能力，即通过在数据库层面直接指定同类型 SQL 以 outline 注释的方法强制绑定执行计划。

总结：OceanBase 相较于传统数据库，其分布式的架构和特殊的存储结构也会带来运维门槛的提高，不过它同时也给予运维人员更高的自由度。运维人员需要熟悉并掌握这些强大的功能和运维技巧，使线上业务具备更好的稳定性。

七、未来展望

OceanBase 开源已经一年有余，我们的运维工具也逐渐趋于成熟，运维能力也在逐步提高。越来越多的 MySQL 正在逐步往 OceanBase 上迁移。随着 OceanBase 4.0 版本的推出，许多新特性也已经在逐步测试中。我们对 4.0 版本的新功能也非常期待。

7.1 单机分布式一体化架构

OceanBase 4.0 版本推出单机分布式一体化架构，支持类似 MySQL 的轻量化单机模式部署，同时也可以在必要时迅速地扩容成分布式模式来提高性能上限。单机与分布式的灵活切换可以大大降低成本，并且基于源生主备库的能力可以快速的完成主备的 DR 切换，有更强的高可用性保证。

7.2 兼容性增强

OceanBase 对 MySQL 的高兼容性一直是我们考量的重点，高度兼容为开发同事节省了大量学习成本和代码成本。在 4.0 版本中，在字符集、约束、函数、存储过程等多方面与 MySQL 的匹配度更高，在使用上与 MySQL 更加接近。

当然，兼容性还包括对 MySQL 生态的兼容，包括 binlog 兼容、canal 兼容、闪回工具兼容等等。

7.3 运维能力提升

OceanBase 作为分布式数据库，组件多、运维环境复杂是痛点。我们后续将基于现有的日志收集工具和分析工具，完成链路式的问题诊断，更精准地定位性能问题、集群内部任务问题等。

携程 Redis On Rocks 实践，节省 2/3 Redis 成本

【作者简介】patpatbear，携程软件技术专家，负责携程缓存内核的维护，热爱开源，专注于高性能、分布式 NoSQL 系统的建设和应用。

一、背景

redis 使用内存作为存储介质，具有良好的性能和低延迟，但其内存容量通常成为瓶颈，且内存价格较高，导致 redis 使用成本较高。

随着 SSD 磁盘性能的不断提高，NVMe SSD 的随机读写延迟也仅有几十微秒，与 redis 的固有延迟（100~200us）相当，用 SSD 作为存储介质也可以达到较低的延迟，同时节省成本。

因此我们研发了 ROR(Redis-On-Rocks)产品，通过对 redis 内核增强以支持数据冷热交换，使用磁盘扩展缓存容量，可节省约 2/3 成本，而性能也能满足大多数业务需求。

二、ROR 简介

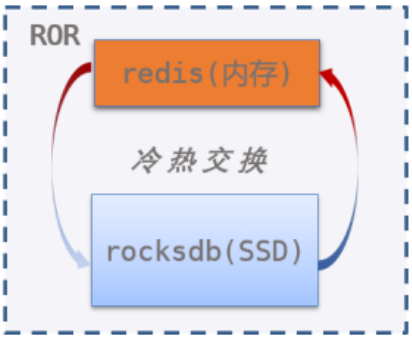
ROR 核心思路很简单：在 redis codebase 基础上扩展冷热交换功能，实现 redis 数据冷热多级存储，降低缓存的综合使用成本。

ROR 将数据分为冷热两部分：

- 热数据沿用 redis 引擎，使用内存存储，数据结构和原生 redis 完全一致
- 冷数据选用 RocksDB 引擎，使用磁盘存储，以 subkey 为粒度存储在 RocksDB 中

ROR 负责冷热数据的交换：

- 换入（从 RocksDB 到 redis）：当客户端访问冷数据，则将 RocksDB 中的数据换入到 redis 中，ROR 把命令依赖的数据换入到 redis，后续命令执行与原生 redis 一致。
- 换出（从 redis 到 RocksDB）：当内存用量超过 maxmemory 之后，则将热数据换出到 RocksDB 中，ROR 冷热交换算法采用了 redis 原生的 LFU 算法，原本被 redis evict 的数据将被交换到内存中。



由于 ROR 继承了 redis 的数据结构和命令实现，只负责冷热数据交换，因此可以兼容几乎所有的 redis 命令，可快速跟进 redis 官方新特性。

三、与 RoF 对比

从长远发展考虑，redis 是事实上的缓存标准，缓存内核基于社区开源 redis 更便于跟进社区 redis 演进，因此 ROR 选择了基于 redis 基础上扩展冷热交换能力。

RedisLabs 的商业产品 Redis-on-Flash(RoF)与 ROR 设计思路类似，但是调研之后，我们发现 RoF 在成本、通用性、性能等方面并不能满足我们的需求。

3.1 成本

RoF 把 value 保存在磁盘、key 保留在内存主表，可以方便地兼容 dbsize、scan、randomkey 等命令，但占用的内存量会随着 dbsize 线性上升。

冷数据 key 保存在内存主表（hashtable），每个 key 的辅助指针、robj 等平均占用约 50B，生产 String 类型平均 value 大小为 512B。从成本的角度看，按照 key 占内存 10%，value 占 90%计算

- 换出 80%value，可减少 72% (80%*90%) 内存
- 换出 80%冷 key，可继续减少 29% (10%*80%/(100%-72%))内存

因此 ROR 并不把冷数据的 key 保存在内存，而是保存到 RocksDB 中单独的 meta column family。

考虑到 meta column family 访问比较频繁，且只存储 type、expire 之类的少量元数据，因此用少量内存(block cache)可以缓存多数冷 key。

经验证，分配 256MB block cache 后，把冷数据的 key 存储到 RocksDB 并不会降低整体 QPS，但会增加 IO 线程的 CPU 消耗，由于 redis 宿主机 cpu 利用率只有 10%，用 cpu 换内存是可以接受的。

3.2 通用性

为了避免重复缓存，RoF 禁用了 RocksDB 层的 table cache 和文件系统层的 page cache。这意味着访问冷 key 时必须进行 IO 操作，因此冷 key 和热 key 的访问延迟会有较大区别。

为了提高通用性，ROR 合理利用 RocksDB 层的 table cache 和操作系统层的 page cache，尽可能利用未被占用的内存，减少访问冷 key 和热 key 之间的延迟差距。

实际上，无论是 DBA 还是业务方，都很难准确预测缓存集群是否存在明显的冷热特征。ROR 适用于通用场景，能够大大减少沟通成本和业务方关于延迟的担忧。

在 redis 迁移至 ROR 时，我们并不评估应用程序是否具有冷热特征，只要业务 QPS 在 redis 的一半以下，对 P99 延迟不是非常敏感，就可以将其迁移到 ROR。

3.3 性能

RoF 按 key 粒度存储，key 与 RocksDB key 一一对应；而 ROR 按 subkey 粒度存储，subkey 都与 RocksDB key 一一对应。

对于 HSET、HGET 等聚合类型命令，RoF 需要换入换出整个 key，而 ROR 只读写必要的 subkey，因此读写放大远低于 RoF，QPS 和延迟也优于 RoF。

以下为 ROR、RoF 在大压力（100 线程不限 QPS）和普通压力（1000 线程 10000QPS），读写纯冷数据的 QPS 和延迟。可以看出：

- 大压力情况下，ROR HGET、HSET 命令 QPS 约为 RoF 的 2~3 倍
- 普通压力情况下，ROR 延迟约 300~500us，远低于 RoF 14~120ms 延迟

场景\方案		ROR	RoF
HGET	100thd	QPS=22134 LAT(mean,p99)=4762 27495(us)	QPS=10195 Latency(mean,p99):9730 16521(us)
	1wqps	QPS=9968 LAT(mean,p99)=343 969(us)	QPS=9956 Latency(mean,p99):14270 100746(us)
HSET	100thd	QPS=26182 Latency(mean,p99): 4034 21802(us)	QPS=7994 Latency(mean,p99):12250 27492(us)

	1wqps	QPS=9969 Latency(mean,p99):511 4437(us)	QPS=7806 Latency(mean,p99):119396 242467(us)

测试说明：

- 数据：hash：5,000,000 (key count) * 2KB (per key，5 个 field，每个 filed 400B)
- 配置：ROR 的 maxmemory 设置为 200MB；RoF 有最小内存限制，设置为 2G
- 场景：a）100thd：压力测试，100 客户端并发，不限速测试；b）1wqps：模拟常规访问，1000 客户端，限速 1W QPS 测试

对于超大的聚合 key，RoF 将整个 key 加载到内存中，会有明显的延迟尖刺（可达秒级）；而 ROR 只将必要的 subkey 换入内存，则不会有明显的延迟尖刺。

多数使用 redis 的业务对延迟比较敏感，不能接受过大延迟尖刺。

场景\方案	**ROR**	**RoF**
HGET hugehash Field	<1ms	1.48s
LPOP hugelist	<1ms	0.704s

测试说明：

- hash：共有 1,000,000 个元素，每个元素 128B
- list：共有 1,000,000 个元素，每个元素 128B

四、实现方案

4.1 冷热交换

以下是客户端访问到冷 key 时 ROR 的处理过程。其中蓝色模块与原生 redis 相同，橙色模块为 ROR 新增的冷热交换功能。

总体上 ROR 先冷热交换（swap），再执行命令处理流程。

冷热交换（swap）过程主要分为以下步骤：

1）语法分析：分析客户端命令涉及哪些 key 和 subkey。比如，可以分析出 MGET k1 k2 k3 依赖于 k1,k2,k3；而 HMGET h1 f1 f2 f3，依赖于 h1.{ f1, f2, f3 }。

2）加锁：根据语法分析出的结果，对命令所依赖的 key 加锁。值得注意的是，这里的锁并

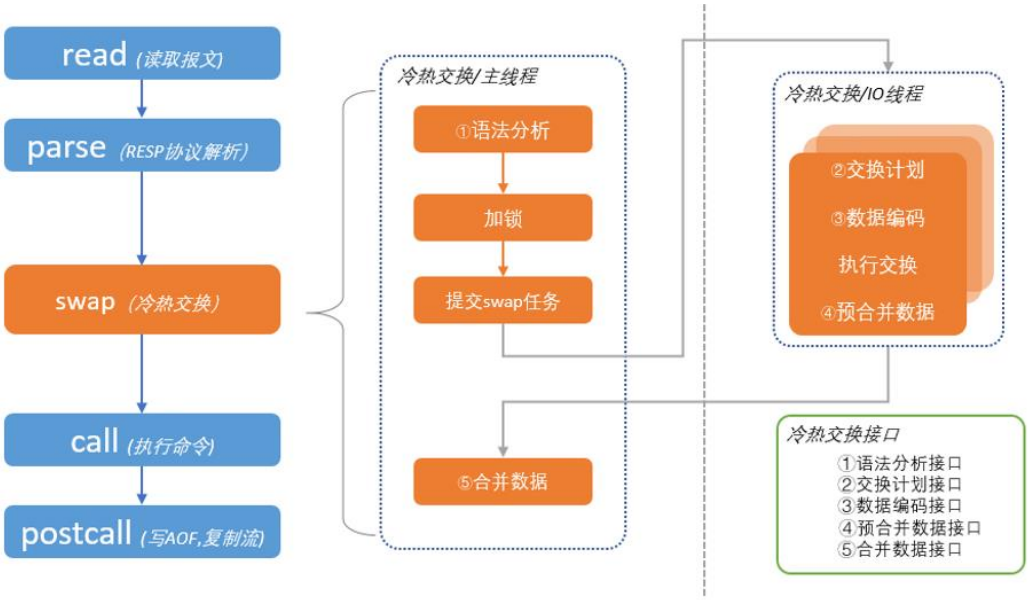
不是 pthread_mutex 之类的线程锁，而是 ROR 项目实现的一种单线程锁，本质上是一个等待队列，详细介绍参考后续并发控制章节。

3）提交 SWAP 任务：拿到锁之后，提交 swap 任务到 IO 线程组执行 RocksDB 读写。

4）执行 RocksDB 读写：IO 线程组执行 RocksDB 读操作。

5）合并数据：将 RocksDB 读取的数据合并到 redis 中。

经过 swap 过程之后，冷数据已经换入到 redis，后续执行命令与原生 redis 一致。



4.2 并发控制

redis 架构上为单主线程，而 RocksDB 提供的是阻塞模式的 API，直接使用 redis 主线程调用 RocksDB 将极大降低 redis 的性能。为了提高 IO 吞吐，ROR 使用了额外的 IO 线程组执行 RocksDB 读写。由于增加了 IO 线程组，对于同一 key 的读写不再是单线程，如果不加控制，那么数据将变得错乱。

为了控制并发，ROR 设计实现了一种单线程可重入锁来保证同一时间对同一 key 只有一个客户端在进行 IO 交换。这里的锁并不是 pthread_mutex 这种系统线程锁，其本质是一个等待队列：当 key 被锁定后，尝试获取该锁的客户端必须等待前序客户端释放锁之后才能获取到 key 的锁。

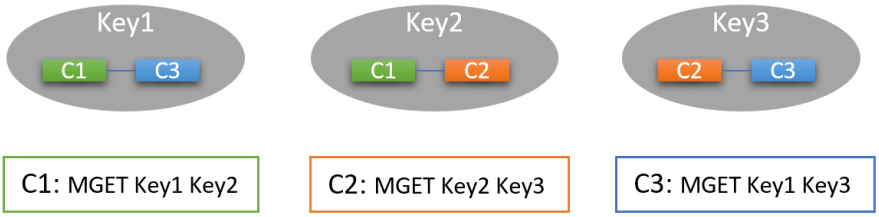
如下图所示，C1、C2、C3 三个客户端先后执行了 MGET 命令，其中 Key1、Key2、Key3 均为冷数据。

C1 依赖 Key1、Key2，由于这 2 个 key 未被锁定且为冷，因此 C1 获取到 Key1、Key2 的锁，并触发了 Key1、Key2 换入；

C2 依赖 Key2、Key3，由于 Key2 被 C1 锁定，因此 C2 等待 C1 执行结束才能获取 key2 锁；Key3 未被锁定且为冷，因此 C2 获取到了 Key2 的锁，并触发了 Key3 换入；

C3 依赖 Key1、key3，由于 Key1、Key3 分别被 C1、C2 锁定，因此 C3 等待 C1、C2 执行结束后才能获取 Key1、Key3 锁。

因此最终换入 Key1、Key2、Key3 换入后，客户端执行顺序为 C1=>C2=>C3。



以上是一个简单的示例，ROR 为了实现 FLUSHDB/BGSAVE 之类涉及整个 keyspace 的命令并发控制需求，等待队列包含 KEY、DB、SVR 三种粒度的锁，大粒度的锁需等待细粒度锁释放后才能获得。此外为了确保 MULTI/EXEC 事务不产生死锁，允许同一个事务重复锁定同一 key（亦即可重入）。

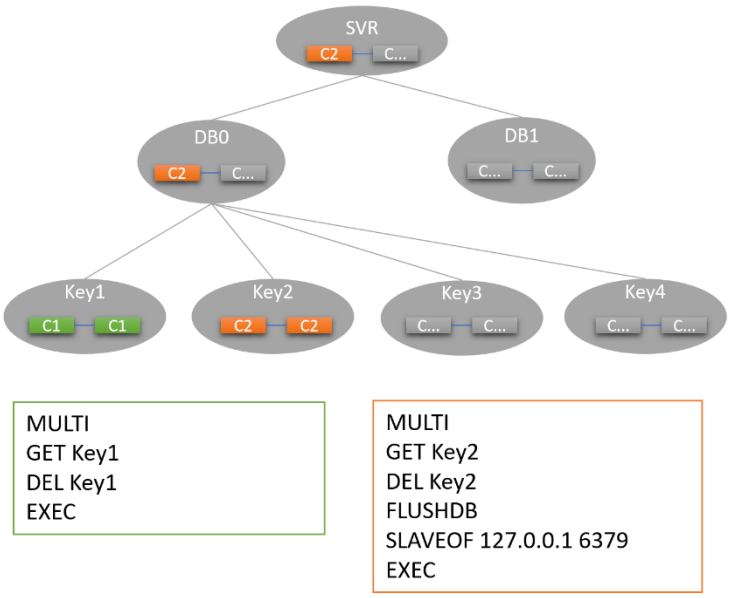
如下图所示，C1、C2 两个客户端先后发起 2 个事务。

C1 依赖 Key1（2 次），由于 C1 在同一事务中依赖 Key1（2 次）且为冷，因此 C1 获得 Key1 锁并触发换入；

C2 依赖 Key2（2 次）、DB0、SVR 锁，由于 C2 在同一事务中依赖 Key2（2 次）且为冷，因此 C2 获得 Key2 锁并触发换入；注意由于 C2 依赖 DB0 锁，DB0 锁范围大约 Key1、Key2，因此只有 C1 释放 Key1 之后才能获得 DB0 锁。

假设 Key1 先于 Key2 被换入，Key1 换入后，C1 事务得到执行并释放 Key1 锁。

当 Key2 换入后，C2 获得 DB0 锁以及 SVR 锁（获得所有锁），C2 事务得到执行。



4.3 冷数据存储

与业界多数方案一样，ROR 的冷数据存储采用了 RocksDB 引擎，设计上参考了 kvrocks、pika 等项目，主要有 3 个要点：

- key 存储到 RocksDB
- subkey 与 RocksDB KV 对应（i.e. 按 subkey 存储）
- lazy 删除聚合类型 key

key 存储到 RocksDB

ROR 为了做到内存消耗与 dbsize 无关，内存中并不会存储冷 key，key 类型、expire、version 等信息会存储到 RocksDB 的 metaCF 中。这样设计主要是考虑每个 key 需要额外消耗约 50B，如果 dbsize 为 1 亿则需要额外消耗约 5GB 内存。对 dbsize 大、value 小的集群来讲，额外消耗的内存过多，冷热分离的性价比则不高。

因此 ROR 和 RoF 不同，不会把冷 key 存储在内存中，少量与 key 相关（scan、randomkey、dbsize）命令，则进行适配性改造。

subkey 与 RocksDB KV 对应

RocksDB 的数据类型只有 KV，与 redis 支持 hash、set、zset 等聚合类型 key 不能一一对应，因此需要构造 redis 聚合类型 key 与 RocksDB KV 类型之间的对应关系。

最直接的方案是将 redis 的聚合类型 key 直接序列化单个为 RocksDB KV，但这种方案的缺点非常明显，即 HGET hash subkey 只依赖单个 subkey 的命令，也需要将整个聚合类型 key 换入到内存，这会造成严重的读写放大。

因此 ROR 将聚合类型的 subkey 存储为 RocksDB KV，换入聚合类型数据冷 key 只需要换入必要的 subkey。

lazy 删除聚合类型 key

对于聚合类型 key 而言，每个 subkey 对应 RocksDB KV，ROR 删除聚合 key 需要删除掉所有的 subkey，直接从 RocksDB 中迭代删除复杂度为 O(N)，会造成延迟尖刺。

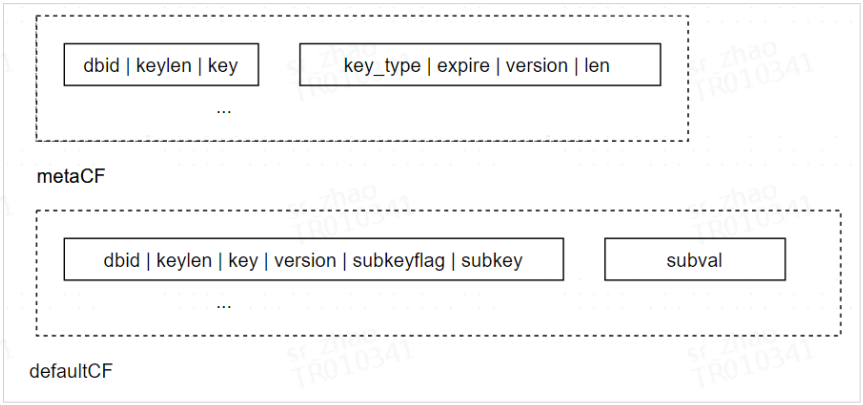
参考 pika、kvrocks 的设计，聚合类型 key 都有版本号，ROR 删除聚合 key 时，只删掉 metaCF 的元数据，而其他 subkey 则在 RocksDB compaction 中通过 compaction filter 逐渐过滤删除。

hash/set/zset 编码

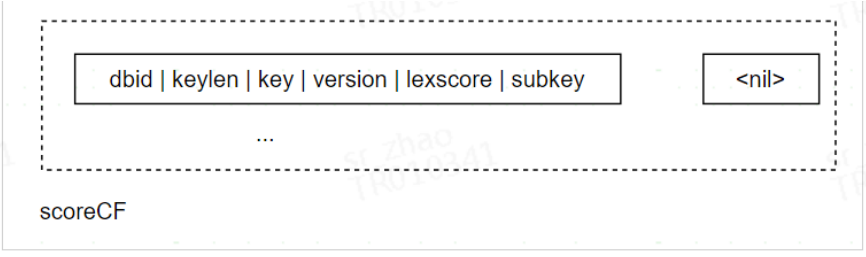
以下是 hash/set 类型的编码格式：

每个 hash/set 在 metaCF 有 1 个 RocksDB KV，记录了类型、超时时间、版本以及 subkey 数量。

每个 hash/set 在 defaultCF 有 N 个 RocksDB KV，每个 subkey 对应一个。由于每个 subkey 都记录了对应的 version，因此删除聚合 key 只需要把 metaCF 的 KV 删掉即完成 lazy 删除。



zset 类型的编码格式类似，只多了 scoreCF 记录 zset 的 score 排序。

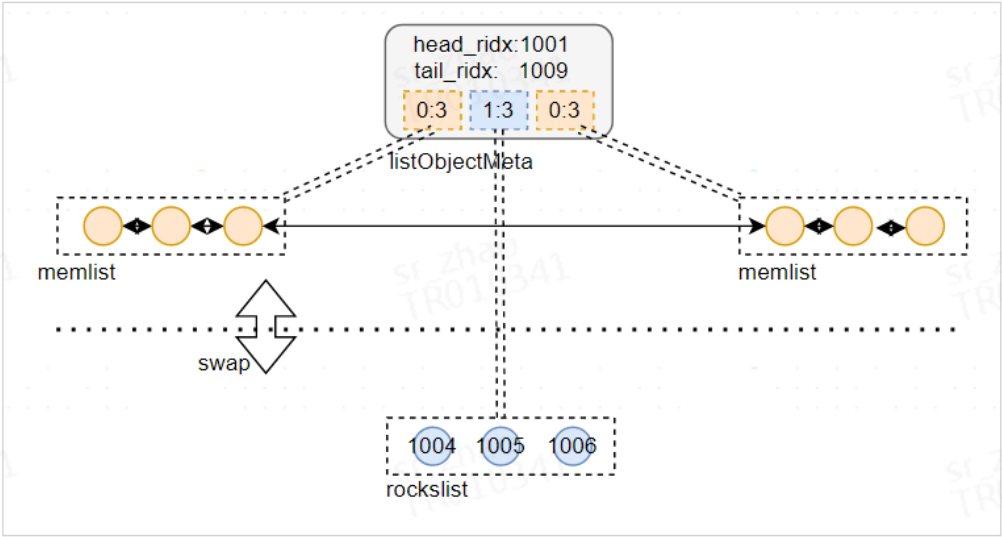


list 编码

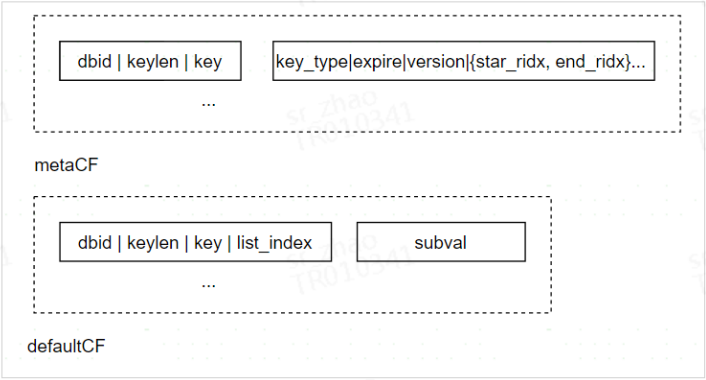
由于与 hash/set/zset 的操作差别较大，list 数据模型设计上也有所差别。设计上，ROR 内存中的 list 仍复用 redis 数据结构，且 list 可能只有部分 subkey 在内存中。

模型上 list 的设计如下：

- list 为任意段 rockslist（冷）和 memlist（热）的组合
- list 元素要么在 memlist、要么在 rockslist，memlist 没有交集
- 分段信息存储在 listObjectMeta.segments 中，segments 的每个元素表示一段，记录了每段的类型以及长度。



rockslist 也按 subkey 粒度存储在 RocksDB 中。



4.4 cuckoo filter 减少 IO

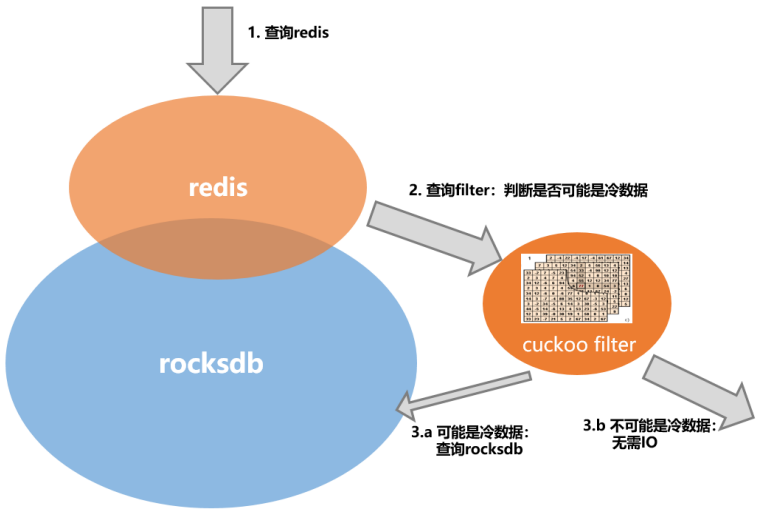
前面提到 ROR 为了做到内存用量与 dbsize 无关，key 元信息不存储在内存中，因此如果客

户端访问的 key 不是热数据，则必须查询 RocksDB 才能确认 key 是否存在：对于 key 存在的情况，读 RocksDB 并换入冷数据是必要的；但如果 key 不存在，则读 RocksDB 是非必要的。特别是当业务 keyspace miss 率高的情况（比如重复读不存在的 key），存在大量的非必要 IO 情况，降低了整体性能。

对于过滤不存在 key 问题，用 bloom filter 能以 8~10 bit per key 的内存取得很好的过滤效果，但由于 bloom filter 不支持删除，而 ROR 的 keyspace 始终处于动态变化中，因此 bloom filter 功能上无法满足需求。

经过调研之后，我们发现 cuckoo filter 可以很好地满足我们的需求，支持删除并且内存消耗量仅需 8 bit per key 即可满足 ROR 过滤准确度需求。

由于无法预测准确到 key 数量，ROR 实现 cuckoo filter 时采用了多个容量指数增长的 cuckoo filter 组成的 cascading cuckoo filter。



经过测试我们发现，对于 keyspace miss 场景，cuckoo filter 可以将 ROR 的 QPS 从 5W 提升到 6W 左右，吞吐提升约 20%；对于 keyspace hit 场景则无明显影响。

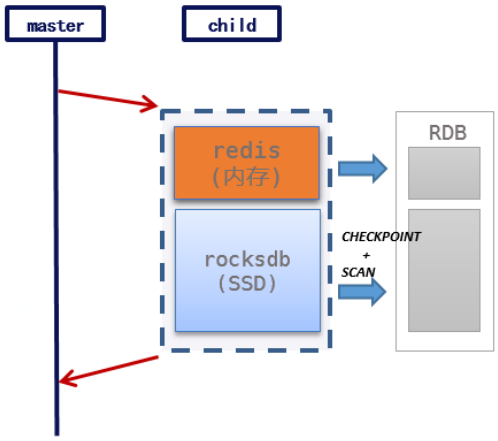
4.5 兼容 redis 复制

ROR 的复制协议完全兼容 redis 原生复制，全量复制采用 RDB 格式，增量复制使用 RESP 协议。由于完全兼容 redis 原生复制协议，ROR 可以直接对接 xpipe，具备 DR 能力。

流式全量复制

ROR 与 Redis 全量复制主要流程相同：master fork 出 child 进程，由 child 进程打 RDB。ROR 由于有冷热两类数据，因此生成 RDB 的与原生 Redis 有区别：

- 热数据生成 RDB 方案不变
- 冷数据先获取 RocksDB CHECKPOINT，然后 SCAN 冷数据转换为 RDB 格式



冷数据（RocksDB 部分）生成 RDB 的一种方案是将冷 key 临时加载内存，复用 redis 的序列化方法构造 RDB，但这种方案加载全部冷 key 会消耗大量 CPU，当遇到 redis 宿主机宕机重启，大量 redis 全量同步争用 CPU 将导致全量同步时间过长。

出于性能考虑，ROR 构造 RDB 并不加载冷 key，而是采用了流式构造 RDB 的方案：使用一个 IO 线程迭代 RocksDB 全量数据，并将迭代的数据流式 append 到 RDB 中。需要注意的是，流式构造 RDB 依赖于 ROR 在存储设计上将同一个聚合类型 key 的 subkey 存储在 RocksDB 相邻位置。

实现层面，流式构造 RDB 方案避免了把 key 加载到内存并跳过 redis 层重新编码，直接将 RockDB 数据流式填充到 rdb，全量复制速度 315MB/s，可以达到 redis 复制性能（390MB/s）的 80%左右。

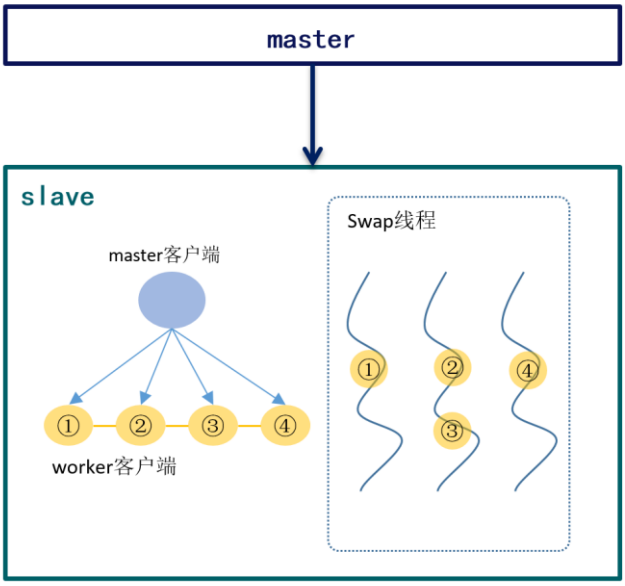
并发增量复制

redis 增量复制过程中，master 通过单个复制客户端推送复制流到 slave。由于复制客户端只有 1 个（冷热交换最大并发为 1）如果 ROR slave 直接用复制客户端交换数据，会出现 slave 复制无法跟上 master 写入。

为了提高复制交换性能，ROR 将从复制客户端将收到的命令分发到多个 worker 客户端，并发执行交换。

如果 worker 客户端在交换结束后直接调用命令，那么 slave 上命令执行的顺序可能与 master 不同，造成主从数据不一致。

ROR 采用的方案下，worker 客户端交换结束后并不立即执行命令，而是等到前序命令全部执行完之后在执行。由于 slave 执行增量复制命令与 master 向下传播的复制流的命令顺序一致，可以确保主从数据一致。



如上图所示，①、②、④在并发执行 IO 操作，虽然②、④可能在①之前完成数据交换，但一定会等到①完成 IO 后再执行命令。

ROR 增量复制并发改造后，slave 处理复制命令速度从几千 QPS 提升到大于 master 的最大写入速度（5~10W QPS 左右，与冷热数据占比相关）。

五、生产实践

从经验来看，多数 redis 集群 QPS 较低但内存用量较大，redis 宿主机通常因为达到内存上限触发扩容，但 CPU 资源则比较空闲，比如携程内 redis 宿主机平均 CPU 使用率约 15%，但平均内存使用率达到 50%。

ROR 采用磁盘增加了缓存容量，能容纳更多的数据量，但 RocksDB 引擎的 compaction 和压缩会消耗更多的 CPU 资源，因此 ROR 可以认为是用空闲的 CPU 换内存的成本解决方案。

成本方面，经验数据显示 1 个 ROR 实例可容纳 3 个 redis 实例的数据，因此 redis 迁 ROR 能节省 2/3 的成本。

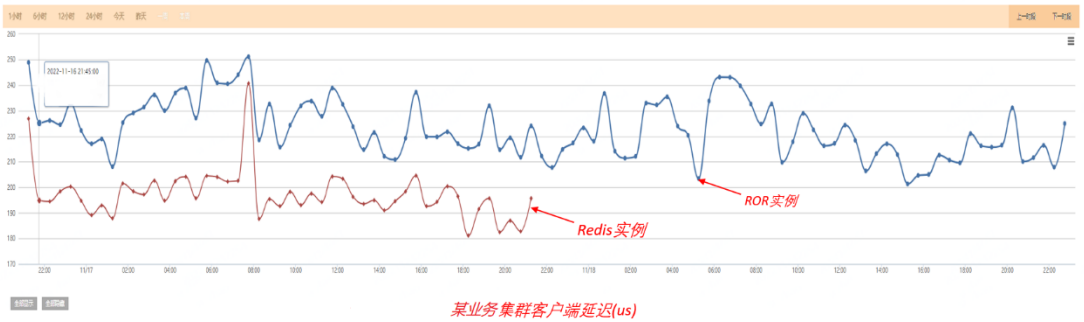
目前在 ROR 在生产部署了几万个实例。由于海外公有云内存单价高，已基本全部部署为 ROR，每年可以节省成本上千万元。

性能方面，从吞吐量考虑，携程内部 redis 集群高 QPS 占比较低，远低于 ROR 的 QPS 上限（参考上文性能数据）。

从延迟考虑，ROR 设计上合理利用缓存，按 subkey 粒度存储，且硬件上 nvme SSD 延迟只有几十微秒，因此与 Redis 相比延迟并没有特别明显的上升。

以下为一个典型 redis 集群迁移 ROR 后延迟对比，其中 80%为冷数据、20%为热数据，迁移

前后客户端访问延迟从 200us 变为 220us 左右。



某业务集群客户端延迟(us)

六、项目开源与未来计划

6.1 项目开源

目前 ROR(Redis-On-Rocks)已开源，采用与 Redis 一致的 BSD 协议。

6.2 未来计划

1）提升单实例 QPS

部分业务场景（比如大数据相关业务）不但数据量大，而且 QPS 也比较高，这些集群可能出现 ROR 主线程 100%情况。针对这些场景，我们考虑从软硬件 2 个层面优化，软件层面考虑减少冷热交换损耗、自动化 pipeline 减少网络 CPU 消耗；硬件层面使用更高主频的 CPU 提升上限。

2）完善数据结构支持

部分使用频次较少的数据结构待优化，比如：bitmap 目前按照 string 处理，读写放大比较大，待优化性能；stream 目前尚未支持，使用内存存储，待支持。

3）减少全量同步

国内与海外的带宽比较小，如果出现全量同步则海外业务受影响时间会比较久。随着随着海外部署量上升，这个问题的影响性逐步增大，后续 ROR 考虑提供可用性与一致性的选项，允许少量数据不一致的情况下增量同步。

大数据

携程日志系统治理演进之路

【作者简介】Dongyu，资深云原生研发工程师，专注于日志与 OLAP 领域，主要负责携程日志平台和 CHPaas 平台的研发及其运维管理工作。

本文将从以下五部分切入，讲述日志系统的演进之路：携程日志的背景和现状、如何搭建一套日志系统、从 ElasticSearch 到 Clickhouse 存储演进、日志 3.0 重构及未来计划。

一、日志背景及现状

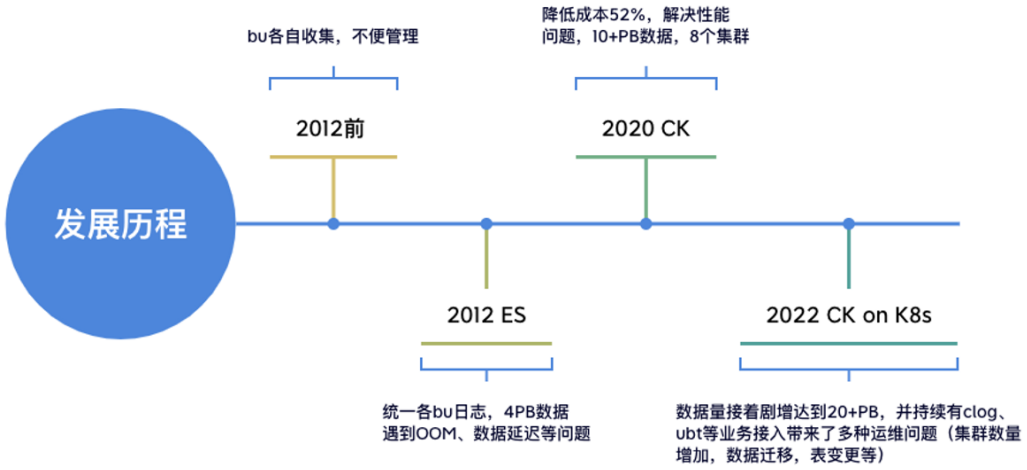


图 1

2012 年以前，携程的各个部门日志自行收集治理（如图 1）。这样的方式缺乏统一标准，不便治理管控，也更加消耗人力和物力。

从 2012 年开始，携程技术中心推出基于 ElasticSearch 的日志系统，统一了日志的接入、ETL、存储和查询标准。随着业务量的增长，数据量膨胀到 4PB 级别，给原来的 ElasticSearch 存储方案带来不少挑战，如 OOM、数据延迟及负载不均等。此外，随着集群规模的扩大，成本问题日趋敏感，如何节省成本也成为一个新的难题。

2020 年初，我们提出用 Clickhouse 作为主要存储引擎来替换 ElasticSearch 的方案，该方案极大地解决了 ElasticSearch 集群遇到的性能问题，并且将成本节省为原来的 48%。2021 年底，日志平台已经累积了 20+PB 的数据量，集群也达到了数十个规模（如图 2）。

2022 年开始，我们提出日志统一战略，将公司的 CLOG 及 UBT 业务统一到这套日志系统，预期数据规模将达到 30+PB。同时，经过两年多的大规模应用，日志集群累积了各种各样的运维难题，如集群数量激增、数据迁移不便及表变更异常等。因此，日志 3.0 应运而生。该方案落地了类分库分表设计、Clickhouse on Kubernetes、统一查询治理层等，聚焦解决了架构和运维上的难题，并实现了携程 CLOG 与 ESLOG 日志平台统一。

图 2

二、如何搭建日志系统

2.1 架构图

从架构图来看（如图 3），整个日志系统可以分为：数据接入、数据 ETL、数据存储、数据查询展示、元数据管理系统和集群管理系统。

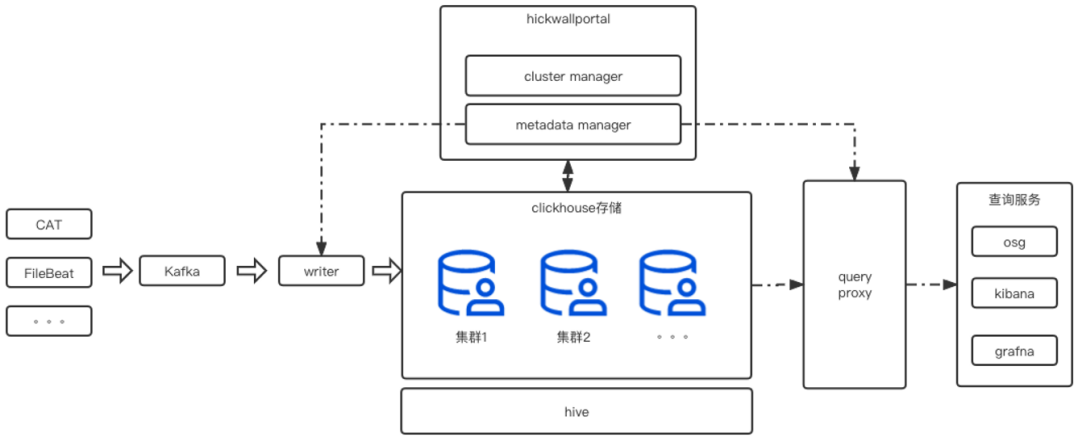


图 3

2.2 数据接入

数据接入主要有两种方式：

第一种是使用公司框架 TripLog 接入到消息中间件 Kafka（Hermes 协议）（如图 4）。

Scenario

```
private static final Logger log = LoggerFactory.getLogger(Demo.class);

public void demo() {
  TagMarker marker = TagMarkerBuilder.newBuilder().scenario("demo").add("tagA", "valueA").add("tagB", "valueB").build();
  log.info(marker, "Hello World");
}
```

图 4

第二种是用户使用 Filebeat/Logagent/Logstash 或者写程序自行上报数据到 Kafka（如图 5），再通过 GoHangout 写入到存储引擎中。

```
filebeat.config.inputs:
  enabled: true
  path:
filebeat.inputs:
- type: log
  enabled: true
  paths:
    - /var/log/history.log
    - /var/log/auth.log
    - /var/log/secure
    - /var/log/messages
  harvester_buffer_size: 102400
  max_bytes: 100000
  tail_files: true
  fields:
    type: os
  ignore_older: 30m
  close_inactive: 2m
  close_timeout: 40m
  close_removed: True
  clean_removed: True
output.kafka:
  hosts: ["
  topic: '
  required_acks: 0
  compression: none
  max_message_bytes: 1000000
processors:
  - rename:
      when:
```

图 5

2.3 数据传输 ETL（GoHangout）

GoHangout 是仿照 Logstash 做的一个开源应用（Github 链接），用于把数据从 Kafka 消费并进行 ETL，最终输出到不同的存储介质（Clickhouse、ElasticSearch）。其中数据处理 Filter 模块包含了常见的 Json 处理、Grok 正则匹配和时间转换等一系列的数据清理功能（如图 6）。GoHangout 会将数据 Message 字段中的 num 数据用正则匹配的方式提取成单独字段。

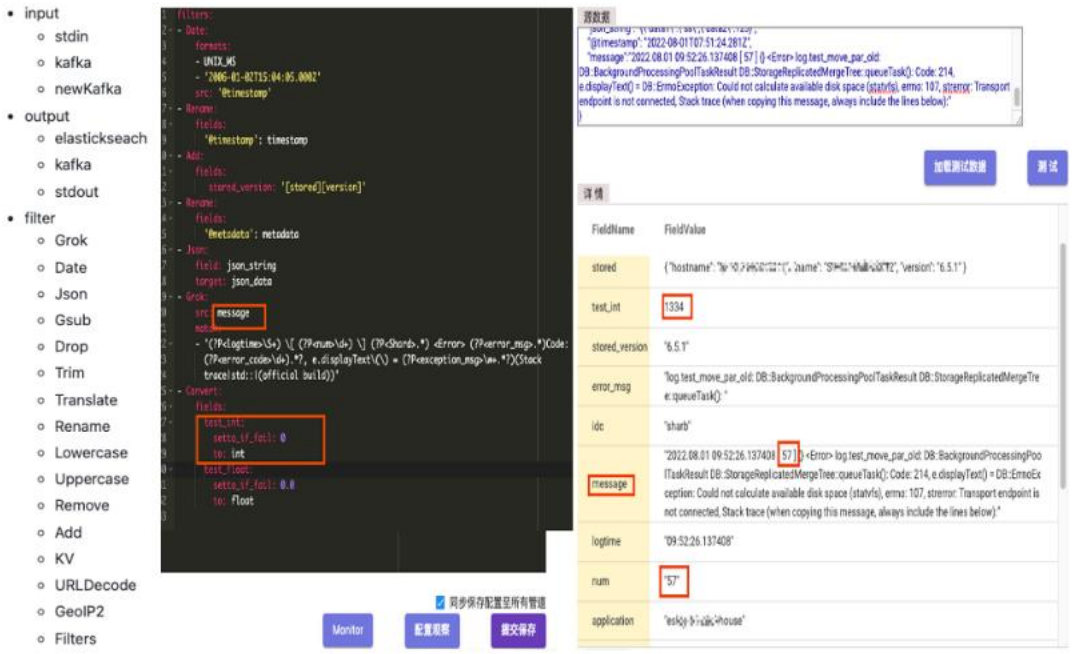


图 6

2.4 ElasticSearch 数据存储

早期 2012 年第一版，我们使用 ElasticSearch 作为存储引擎。ElasticSearch 存储主要由 Master Node、Coordinator Node、Data Node 组成（如图 7）。Master 节点主要负责创建或删除索引，跟踪哪些节点是集群的一部分，并决定哪些分片分配给相关的节点；Coordinator 节点主要用于处理请求，负责路由请求到正确的节点，如创建索引的请求需要路由到 Master 节点；Data 节点主要用于存储大量的索引数据，并进行增删改查，一般对机器的配置要求比较高。

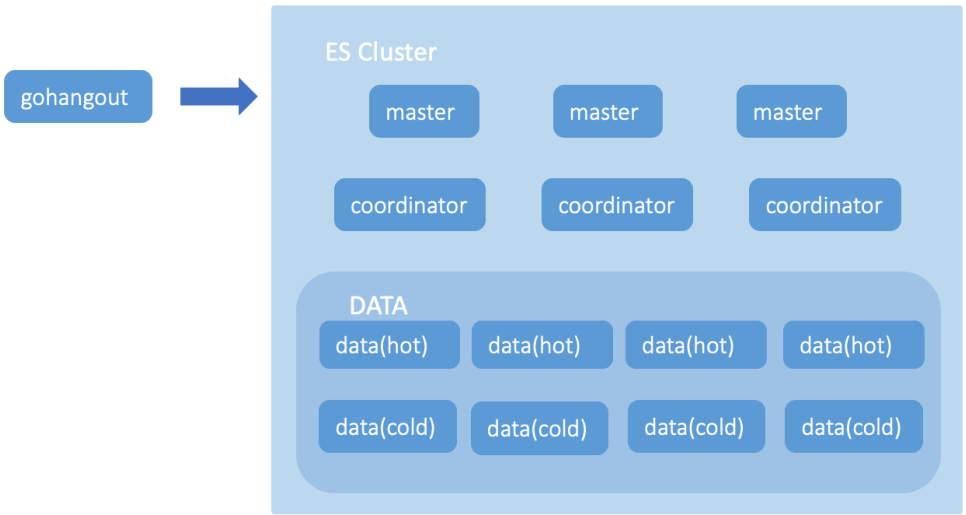


图 7

2.5 数据展示

数据展示方面我们使用了 Elastic Stack 家族的 Kibana（如图 8）。Kibana 是一款适合于 ElasticSearch 的数据可视化和管理工具，提供实时的直方图、线形图、饼状图和表格等，极大地方便日志数据的展示。

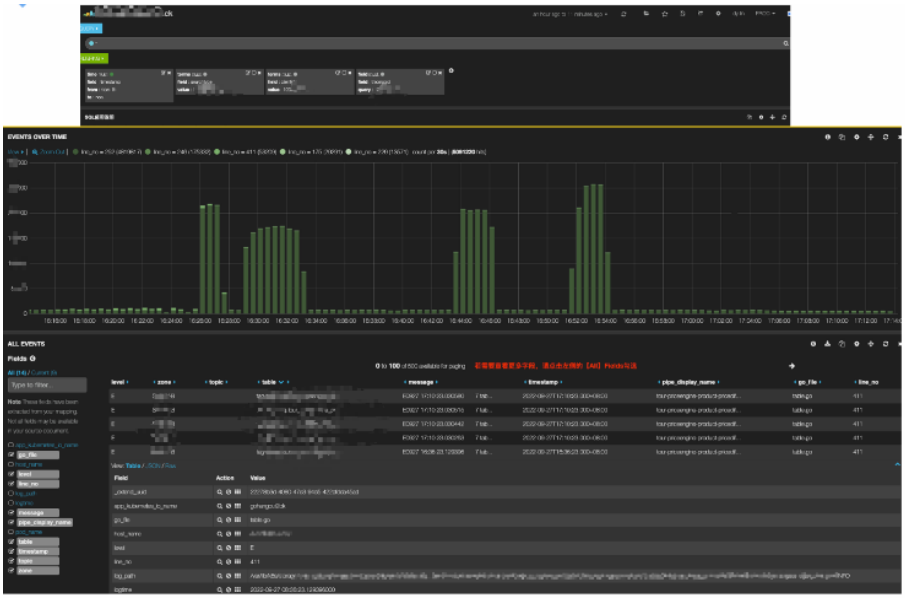


图 8

2.6 表元数据管理平台

表元数据管理平台是用户接入日志系统的入口，我们将每个 Index/ Table 都定义为一个 Scenario（如图 9）。我们通过平台配置并管理 Scenario 的一些基础信息，如：TTL、归属、权限、ETL 规则和监控日志等。

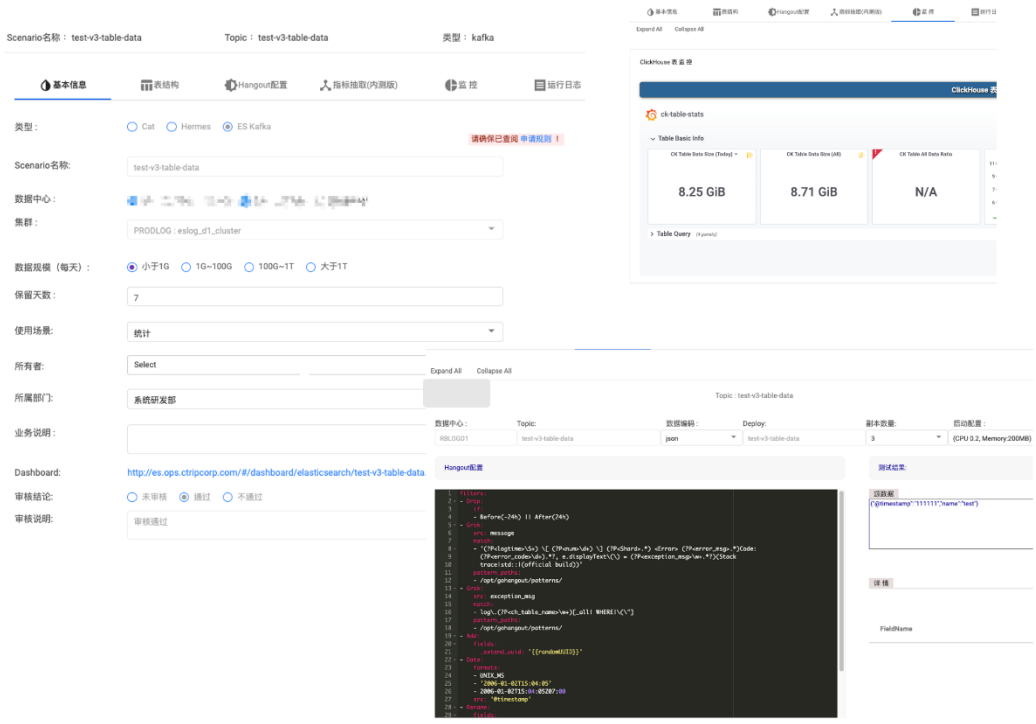


图 9

三、 从 Elasticsearch 到 Clickhouse

我们将会从背景、Clickhouse 简介、ElasticSearch 对比和解决方案四方面介绍日志从 ElasticSearch 到 Clickhouse 的演进过程。2020 年初，随着业务量的增长，给 ElasticSearch 集群带来了不少难题，主要体现在稳定性、性能和成本三个方面。

（1）稳定性上：

- ElasticSearch 集群负载高，导致较多的请求 Reject、写入延迟和慢查询。
- 每天 200TB 的数据从热节点搬迁到冷节点，也有不少的性能损耗。
- 节点间负载不均衡，部分节点单负载过高，影响集群稳定性。
- 大查询导致 ElasticSearch 节点 OOM。

（2）性能上：

- ElasticSearch 的吞吐量也达到瓶颈。
- 查询速度受到整体集群的负载影响。

（3）成本上：

- 倒排索引导致数据压缩率不高。
- 大文本场景性价比低，无法保存长时间数据。

3.1 Clickhouse 简介与 Elasticsearch 对比

Clickhouse 是一个用于联机分析 (OLAP) 的列式数据库管理系统 (DBMS)。Yandex 在 2016 年开源，使用 C++ 语法开发，是一款 PB 级别的交互式分析数据库。包含了以下主要特效：列式存储、Vector、Code Generation、分布式、DBMS、实时 OLAP、高压缩率、高吞吐、丰富的分析函数和 Shared Nothin g 架构等。

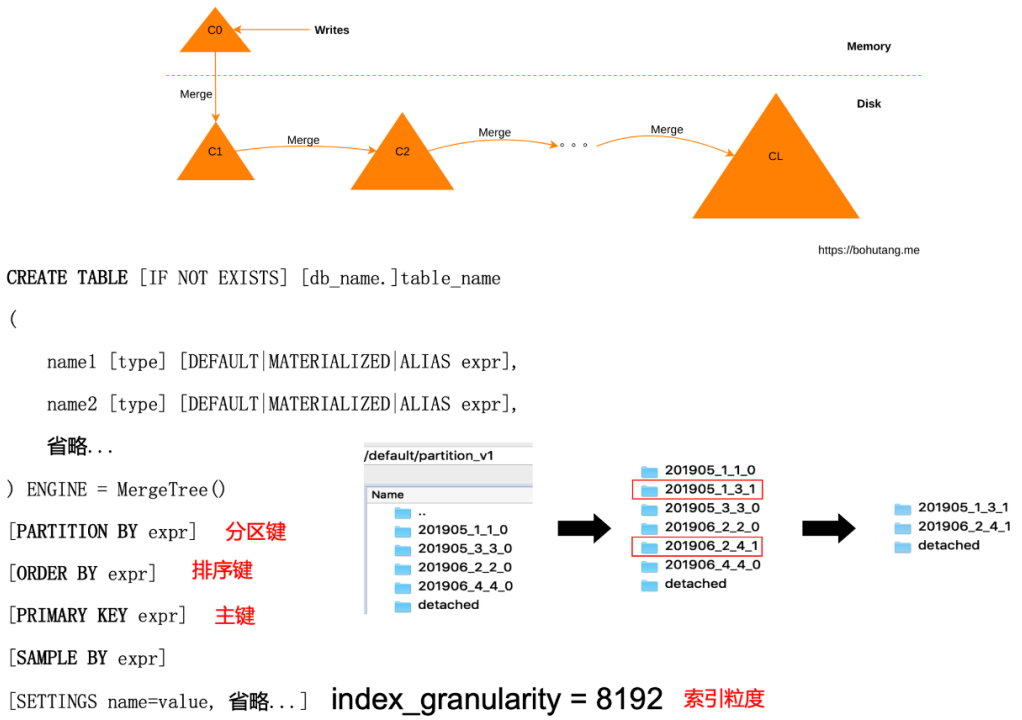


图 10

Clickhouse 采用的是 SQL 的交互方式，非常方便上手。接下来，我们将简单介绍一下 Clickhouse 的类 LSM、排序键、分区键特效，了解 Clickhouse 的主要原理。

首先，用户每批写入的数据会根据其排序键进行排序，并写入一个新的文件夹（如 201905_1_1_0），我们称为 Part C0（如图 10）。随后，Clickhouse 会定期在后台将这些 Part 通过归并排序的方式进行合并排序，使得最终数据生成一个个数据顺序且空间占用较大的 Part。这样的方式从磁盘读写层面上看，能充分地把原先磁盘的随机读写巧妙地转化为顺序读写，大大提升系统的吞吐量和查询效率，同时列式存储+顺序数据的存储方式也为数据压缩率提供了便利。201905_1_1_0 与 201905_3_3_0 合并为 201905_1_3_1 就是一个经典的例子。

另外，Clickhouse 会根据分区键（如按月分区）对数据进行按月分区。05、06 月的数据被分为了不同的文件夹，方便快速索引和管理数据。

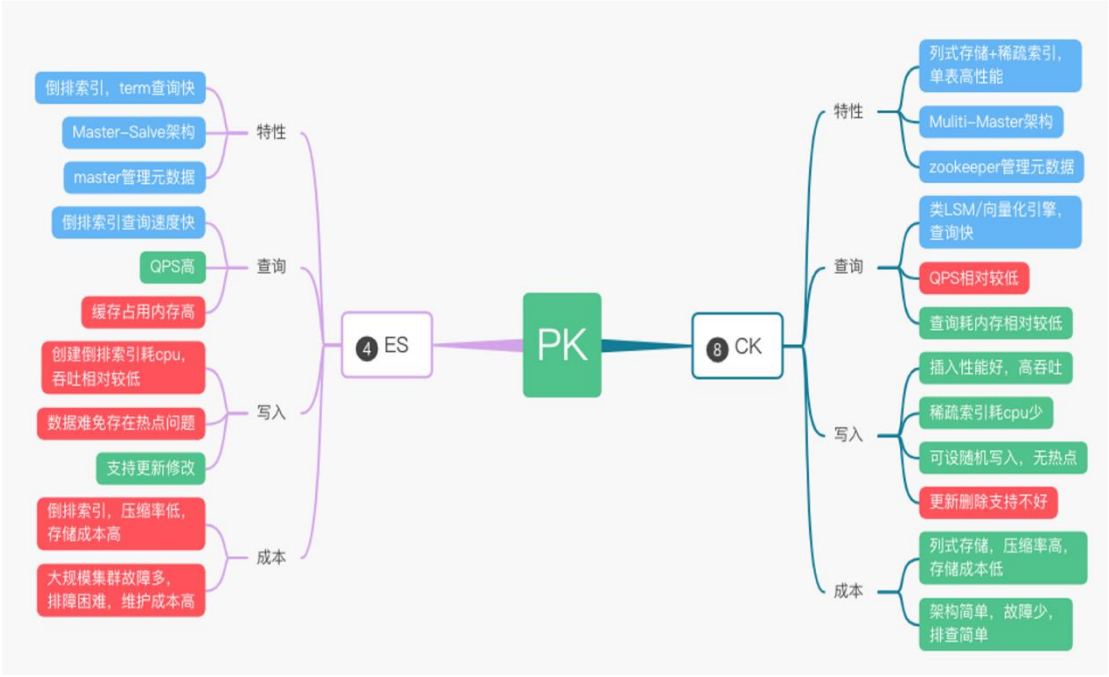


图 11

我们看中了 Clickhouse 的列式存储、向量化、高压缩率和高吞吐等特效（如图 11），很好地满足了我们当下日志集群对性能稳定性和成本的诉求。于是，我们决定用 Clickhouse 来替代原本 ElasticSearch 存储引擎的位置。

3.2 解决方案

有了存储引擎后，我们需要实现对用户无感知的存储迁移。这主要涉及了以下的工作内容（如图 12）：自动化建表、GoHangout 修改、Clickhouse 架构设计部署和 Kibana 改造

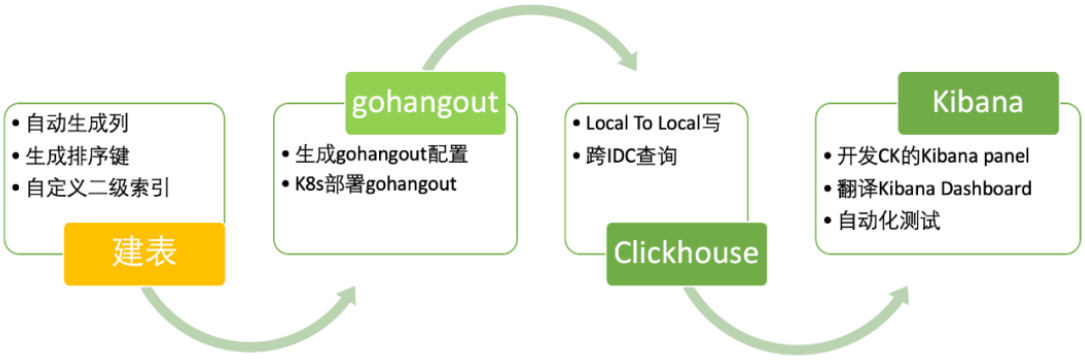


图 12

（1）库表设计

```
CREATE TABLE log.xxxx (
    `timestamp` DateTime,
    `_log_increment_id` Int64,
    `host_name` LowCardinality(String),
    `log_level` LowCardinality(String),
    `message` String,
    `message_prefix` String MATERIALIZED substring(message, 1, 256),
    `_tag_keys_str` Array(String),
    `_tag_vals_str` Array(String),
    `_tag_keys_str` Array(String),
    `_tag_vals_float` Array(Float64),
    ......
    INDEX idx_message_prefix message_prefix TYPE tokenbf_v1(8192,2,0) GRANULARITY 16,
    ......
)
ENGINE = ReplicatedMergeTree('/clickhouse/tables/{shard}/xxxx','{replica}')
PARTITION BY toYYYYMMDD(timestamp)
ORDER BY(timestamp,_log_increment_id,host_name,log_level)
TTL timestamp + toIntervalHour(168)
SETTINGS index_granularity = 8192
```

图 13

我们对 ck 在日志场景落地做了很多细节的优化（如图 13），主要体现在库表设计：

- 我们采用双 list 的方式来存储动态变化的 tags（当然最新的版本 22.8，也可以用 map 和新特性的 json 方式）。
- 按天分区和时间排序，用于快速定位日志数据。
- Tokenbf_v1 布隆过滤用于优化 term 查询、模糊查询。
- _log_increment_id 全局唯一递增 id，用于滚动翻页和明细数据定位。
- ZSTD 的数据压缩方式，节省了 40%以上的存储成本。

（2）Clickhouse 存储设计

Clickhouse 集群主要由查询集群、多个数据集群和 Zookeeper 集群组成（如图 14）。查询集群由相互独立的节点组成，节点不存储数据是无状态的。数据集群则由 Shard 组成，每个 Shard 又涵盖了多个副本 Replica。副本之间是主主的关系（不同于常见的主从关系），两个副本都可以用于数据写入，互相同步数据。而副本之间的元数据一致性则有 Zookeeper 集群负责管理。

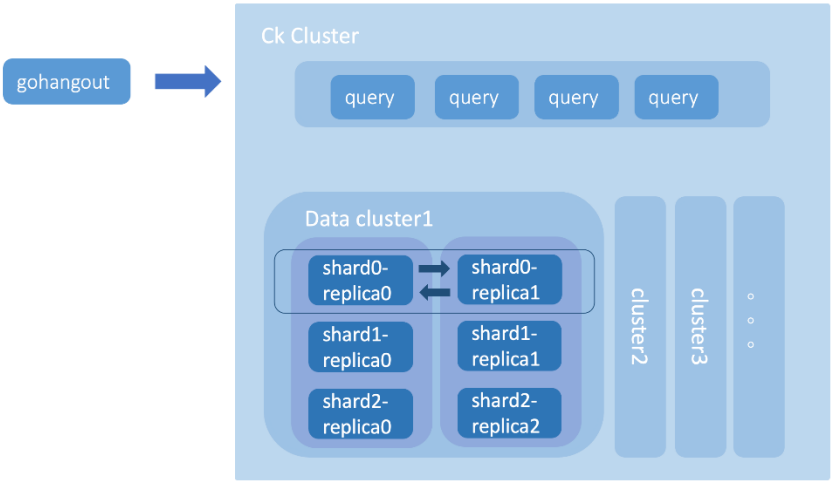
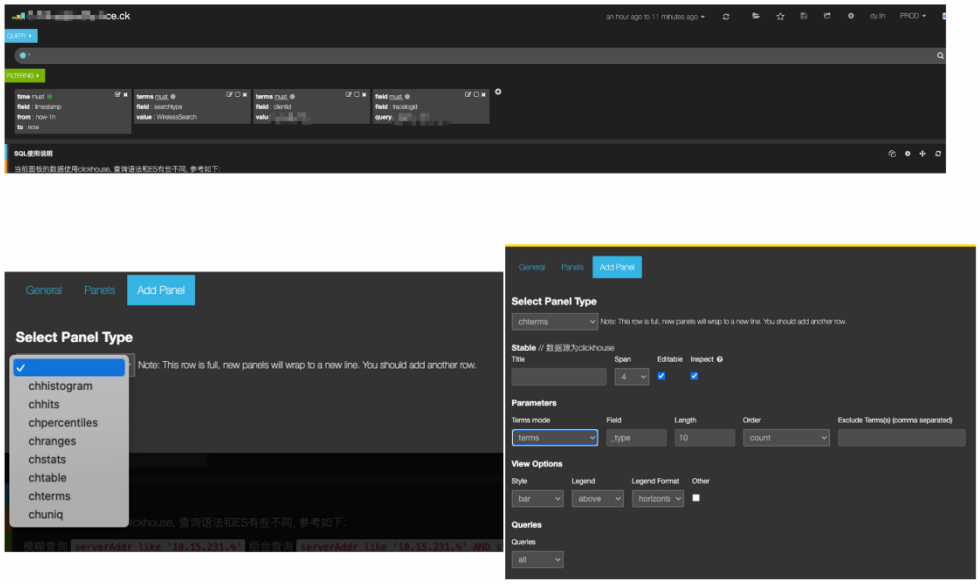


图 14

（3）数据展示

为了实现用户无感知的存储切换，我们专门实现了 Kibana 对 Clickhouse 数据源的适配并开发了不同的数据 panel（如图 15），包括：chhistogram、chhits、chpercentiles、chranges、chstats、chtable、chterms 和 chuniq。通过 Dashboard 脚本批量生产替代的方式，我们快速地实现了原先 ElasticSearch 的 Dashboard 的迁移，其自动化程度达到 95%。同时，我们也支持了使用 Grafana 的方式直接配置 SQL 来生成日志看板。



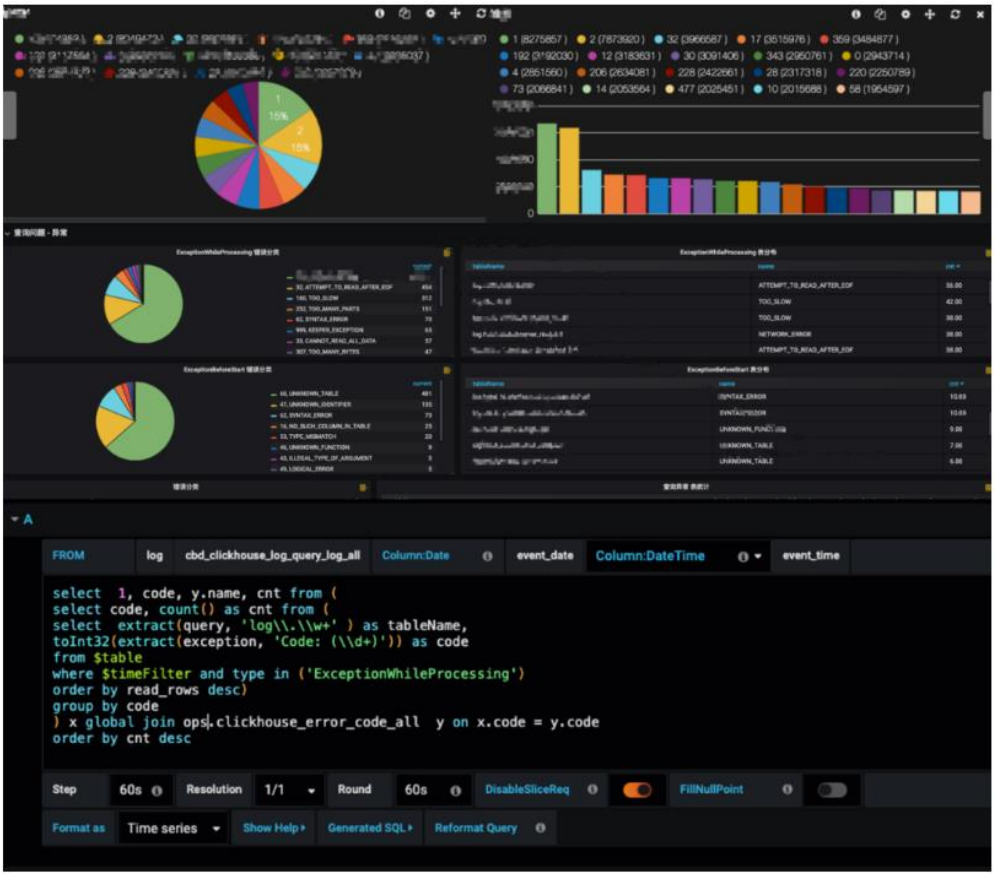


图 15

（4）集群管理平台

为了更好地管理 Clickhouse 集群，我们也做了一整套界面化的 Clickhouse 运维管理平台。该平台覆盖了日常的 shard 管理、节点生成、绑定/解绑、权重修改、DDL 管理和监控告警等治理工具（如图 16）。

图 16

3.3 成果

- 迁移过程自动化程度超过 95%，基本实现对用户透明。
- 存储空间节约 50+%（如图 17），用原有 ElasticSearch 的服务器支撑了 4 倍业务量的增长。
- 查询速度比 ElasticSearch 快 4~30 倍，查询 P90 小于 300ms，P99 小于 1.5s。

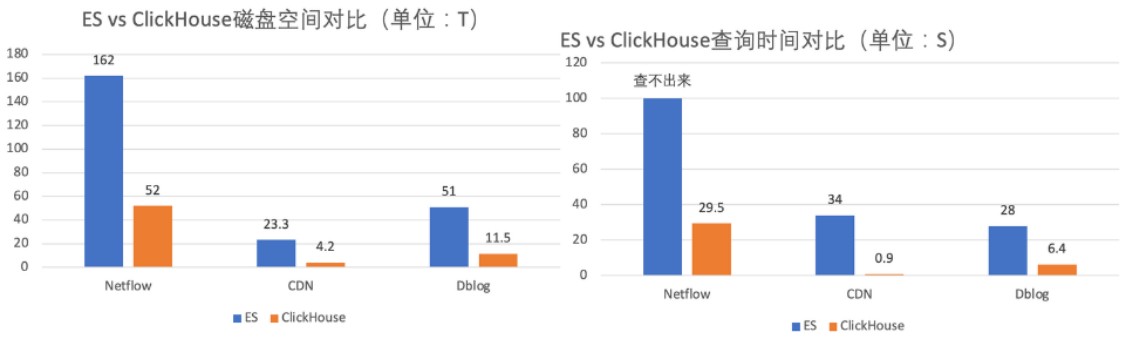


图 17

四、日志 3.0 构建

时间来到 2022 年，公司日志规模再进一步增加到 20+PB。同时，我们提出日志统一战略，将公司的 CLOG 及 UBT 业务统一到这套日志系统，预期数据规模将达到 30+PB。另外，经过两年多的大规模应用，日志系统也面临了各种各样的运维难题。

(1) 性能与功能痛点

- 单集群规模太大，Zookeeper 性能达到瓶颈，导致 DDL 超时异常。
- 当表数据规模较大时，删除字段，容易超时导致元数据不一致。
- 用户索引设置不佳导致查询慢时，重建排序键需要删除历史数据，重新建表。
- 查询层缺少限流、防呆和自动优化等功能，导致查询不稳定。

(2) 运维痛点

- 表与集群严格绑定，集群磁盘满后，只能通过双写迁移。
- 集群搭建依赖 Ansible，部署周期长（数小时）。
- Clickhouse 版本与社区版本脱节，目前集群的部署模式不便版本更新。

面对这样的难题，我们在 2022 年推出了日志 3.0 改造，落地了集群 Clickhouse on Kubernetes、类分库分表设计和统一查询治理层等方案，聚焦解决了架构和运维上的难题。最终，实现了统一携程 CLOG 与 ESLOG 两套日志系统。

4.1 ck on k8s

我们使用 Statefulset、反亲和、Configmap 等技术实现了 Clickhouse 和 Zookeeper 集群的 Kubernetes 化部署，使得单集群交付时间从 2 天优化到 5 分钟。同时，我们统一了部署架构，将海内外多环境部署流程标准化。这种方式显著地降低了运维成本并释放人力。更便利的部署方式有益于单个大集群的切割，我们将大集群划分为多个小集群，解决了单集群规模过大导致 Zookeeper 性能瓶颈的问题。

4.2 类分库分表设计

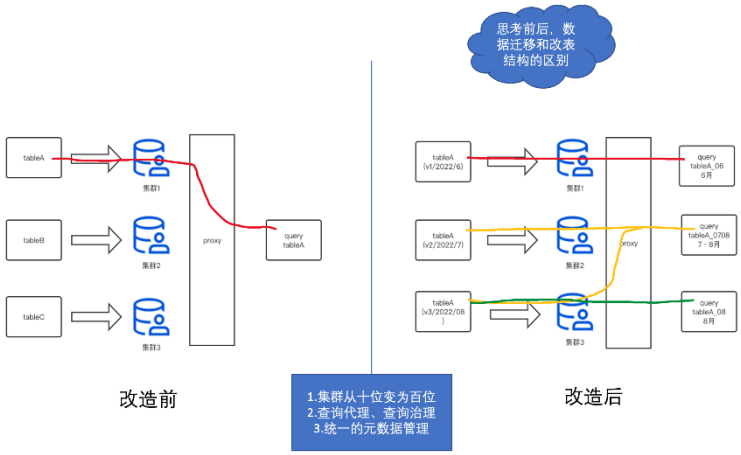


图 18

（1）数据跨如何跨集群

假设我们有三个数据集群 1、2、3 和三个表 A、B、C（如图 18）。在改造之前，我们单张表（如 A）只能坐落在一个数据集群 1 中。这样的设计方式，导致了当集群 1 磁盘满了之后，我们没有办法快速地将表 A 数据搬迁到磁盘相对空闲的集群 2 中。我们只能用双写的方式

将表 A 同时写入到集群 1 和集群 2 中，等到集群 2 的数据经过了 TTL 时间（如 7 天）后，才能将表 A 从数据集群 1 中删除。这样，对我们的集群运维管理带来了极大的不方便和慢响应，非常耗费人力。

于是，我们设计一套类分库分表的架构，来实现表 A 在多个集群 1、2、3 之间来回穿梭。我们可以看到右边改造后，表 A 以时间节点作为分库分表的切换点（这个时间可以是精确到秒，为了好理解，我们这里以月来举例）。我们将 6 月份的数据写入到集群 1、7 月写到集群 2、8 月写到集群 3。当查询语句命中 6 月份数据时，我们只查询集群 1 的数据；当查询语句命中 7 月和 8 月的数据，我们就同时查询集群 2 和集群 3 的数据。

我们通过建立不同分布式表的方式实现了这个能力（如：分布式表 tableA_06/tableA_07/tableA_08/tableA_0708，分布式表上的逻辑集群则是集群 1、2、3 的组合）。这样，我们便解决了表跨集群的问题，不同集群间的磁盘使用率也会趋于平衡。

（2）如何修改排序键不删除历史数据

非常巧妙的是，这种方式不仅能解决磁盘问题。Clickhouse 分布式表的设计只关心列的名称，并不关心本地数据表的排序键设置。基于这种特性，我们设计表 A 在集群 2 和集群 3 使用不一样的排序键。这样的方式也能够有效解决初期表 A 在集群 2 排序键设计不合理的问题。我们通过在集群 3 上重新建立正确的排序键，让其对新数据生效。同时，表 A 也保留了旧的 7 月份数据。旧数据会在时间的推移一下被 TTL 清除，最终数据都使用了正确的排序键。

（3）如何解决删除大表字段导致元数据不一致

更美妙的是，Clickhouse 的分布式表设计并不要求表 A 在 7 月和 8 月的元数据字段完全一致，只需要有公共部分就可以满足要求。比如表 A 有在 7 月有 11 个字段，8 月份想要删除一个弃用的字段，那么只需在集群 3 上建 10 个字段的本地表 A，而分布式表 tableA_0708 配置两个表共同拥有的 10 个字段即可(这样查分布式表只要不查被删除的字段就不会报错)。通过这种方式，我们也巧妙地解决了在数据规模特别大的情况下（单表百 TB），删除字段导致常见的元数据不一致问题。

（4）集群升级

同时，这种多版本集群的方式，也能方便地实现集群升级迭代，如直接新建一个集群 4 来存储所有的 09 月的表数据。集群 4 可以是社区最新版本，通过这种迭代的方式逐步实现全部集群的升级。

4.3 元数据管理

为了实现上述的功能，我们需要维护好一套完整的元数据信息，来管理表的创建、写入和 DDL（如图 19）。该元数据包含每个表的版本定义、每个版本数据的数据归属集群和时间范围等。

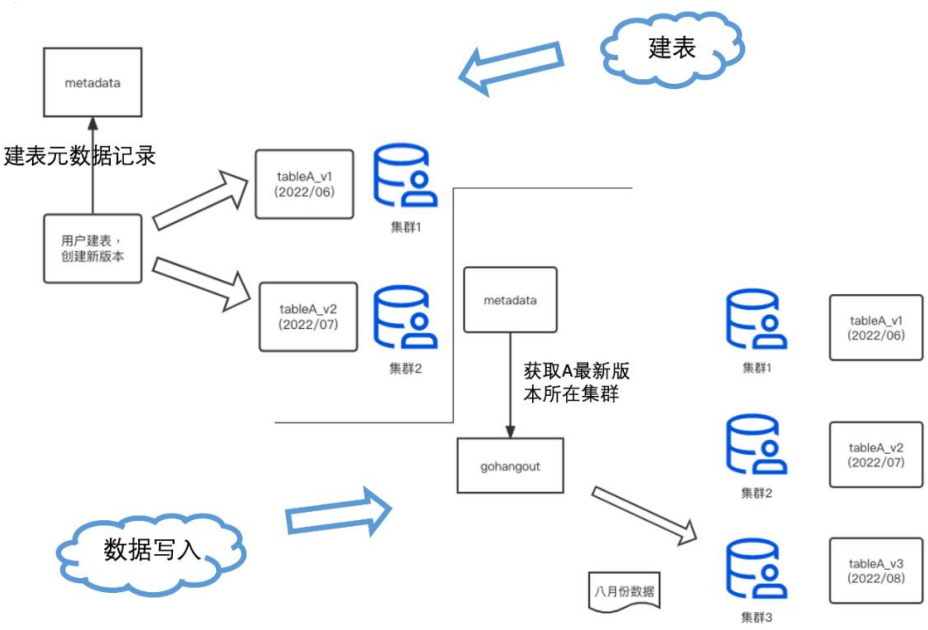


图 19

4.4 统一查询治理层

（1）Antlr4 的 SQL 解析

在查询层，我们基于 Antlr4 技术，将用户的查询 SQL 解析成 AST 树。通过 AST 树，我们能够快速地获得 SQL 的表名、过滤条件、聚合维度等（如图 20）。我们拿到这些信息后，能够非常方便地对 SQL 实时针对性的策略，如：数据统计、优化改写和治理限流等。

```
CREATE TABLE test_local_1 on
cluster ck100032722 ( `db_id` Int64 COMMENT 'db的id',
`desc` String COMMENT '描述',
`db_location_uri` String COMMENT '路径',
`name` String COMMENT '名称',
`dfs_quotas` String COMMENT 'quotas',
`dfs_usage` String COMMENT '使用量',
`owner_name` String COMMENT 'owner名称',
`owner_type` String COMMENT '类型',
`d` String COMMENT '日期分区键',
`test_string_1` String,
`test_string_2` String,
`test_string_3` String,
`test_1` String,
`test_2` String,
`test_3` String ) ENGINE = ReplicatedMergeTree('/clic
'{replica}')
ORDER BY
d PARTITION BY d SETTINGS index_granularity = 8192
```

```
SELECT
    orderid,
    result,
    toUInt32(timestamp) * 1000 as `timestampvalue`,
    stored_store_msg as `message`
from
    log.booking_schedule_all
where
    timestamp >= toDateTime('?')
    and timestamp <= toDateTime('?')
    and project in ('transfersvrAsync')
    and orderid = '?'
ORDER BY
    timestamp desc
LIMIT 100
```

图 20

（2）查询代理层

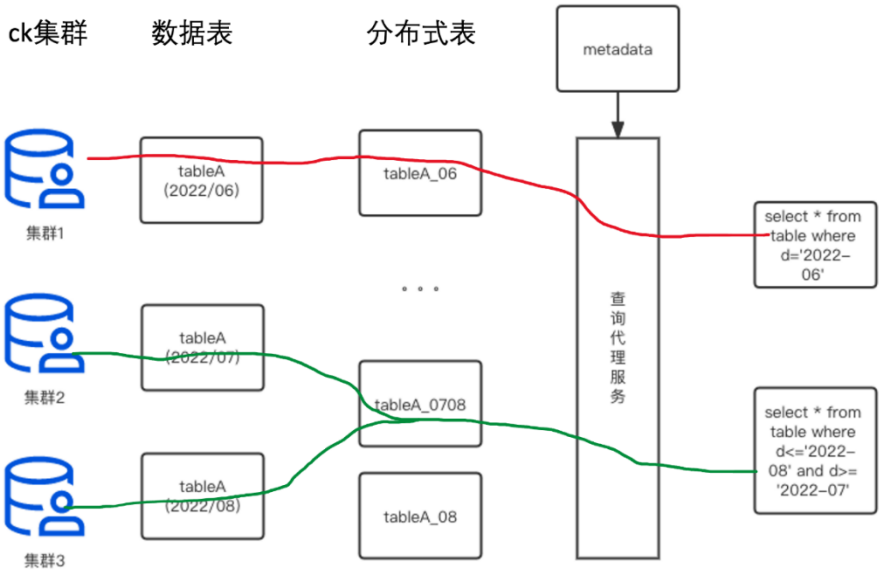


图 21

我们对所有用户的 SQL 查询做了一层统一的查询网关代理（如图 21）。该程序会根据元数据信息和策略对用户的 SQL 进行改写，实现了精准路由和性能优化等功能。同时，该程序会记录每次查询的明细上下文，用于对集群的查询做统一化治理，如：QPS 限制、大表扫描限制和时间限制等拒绝策略，来提高系统的稳定性。

五、未来计划

通过日志 3.0 的构建，我们重构了日志系统的整体架构，实现集群 Kubernetes 化管理，并成功地解决了历史遗留的 DDL 异常、数据跨集群读写、索引重构优、磁盘治理和集群升级等运维难题。2022 年，日志系统成果地支撑了公司 CLOG 与 UBT 业务的数据接入，集群数据规模达到了 30+PB。

当然，携程的日志系统演进也不会到此为止，我们的系统在功能、性能和治理多方面还有不少改善的空间。在未来，我们将进一步完善日志统一查询治理层，精细化地管理集群查询与负载；推出日志预聚合功能，对大数据量的查询场景做加速，并支持 AI 智能告警；充分地运用云上能力，实现弹性混合云，低成本支撑节假日高峰；推到日志产品在携程系各个公司的使用覆盖等。让我们一起期待下一次的日志升级。

提速 10 倍+，StarRocks 指标平台在携程火车票的实践

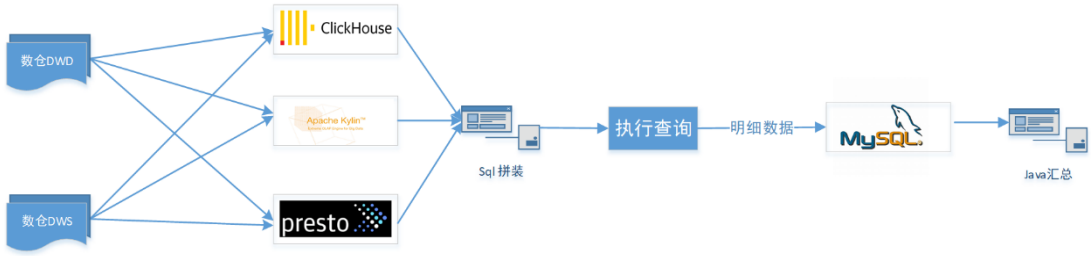
【作者简介】Kane，携程高级数仓经理，专注数仓建设、数据应用和分析；Wn，大数据平台开发专家，专注大数据领域。

携程火车票事业群运营着铁友、携程火车票和去哪儿火车票等重要的业务和品牌，目前正在积极地拓展海外市场。火车票的指标平台旨在为业务人员提供便捷的指标查询服务，让业务人员能够快速灵活地获得这些业务和品牌相关的指标数据。

一、早期 OLAP 架构与痛点

火车票事业群的业务涵盖了火车票、国际火车票、汽车票（含船票）等产品，错综复杂的业务也产生了多种多样订单和行为数据，通过对这些数据的分析可以揭示当前业务的发展现状，也可以为未来的发展提供方向指引。

早些时候事业群开发过一套指标平台，根据不同的指标类型使用了 3 套数据库引擎，分别是 ClickHouse，Apache Kylin （以下简称 Kylin）和 Presto，如下图所示。



在旧版的指标平台中，为了提升查询性能使用了 ClickHouse、 Kylin 和 Presto 等多种存储和查询引擎，数据层混合使用了明细层和轻度汇总层，由此带来的问题有：

- 指标数据源混乱，容易造成口径不一致，维护成本大。
- 学习成本高，BI 同学录入指标不仅需要了解不同存储的区别，还需要掌握不同引擎的数据同步方法。
- 架构不合理，指标平台将查询的中间结果通过 jdbc 写入 mysql 后再到服务端用 java 做汇总计算，处理链路过长，整体性能非常差，导致部分指标查询需要半小时以上的等待时间。

鉴于这些原因，无论是用户（运营人员）还是指标开发人员，都面临着使用极差的问题。在这种情况下，我们决定使用基于一种查询速度快和使用简单的分布式数据库来重构指标平台。

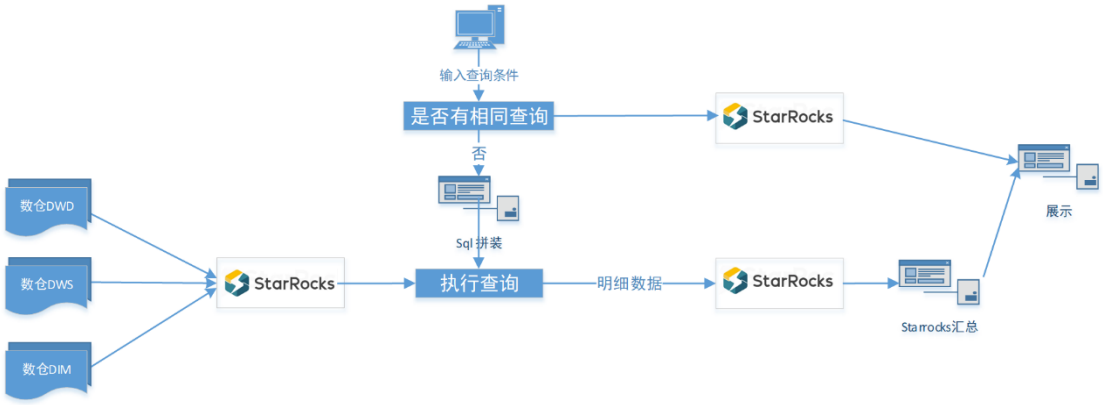
二、指标平台重构整体设计

首先，重构指标平台我们首先考虑的是将多套存储合并成一套，虽说 ClickHouse 和 Kylin 已经足够强大，但是不足也很明显。比如 ClickHouse 的 join 性能不尽如人意，并发性能差，SQL 语法是非标准的，使用起来不方便，大量的查询很容易将 CPU 打满；Kylin 是一个分析引擎，不支持增删改操作，修改数据需要重新导入，修改 schema 需要重建 Cube（ETL 成本很高），其次 Kylin 需要预先创建模型加载数据到 Cube 后才可进行查询，使用上需要具备一定的数仓知识。

于是我们将目光投向 StarRocks，StarRocks 是一款全场景的 MPP 数据库，相比 ClickHouse 等具有以下优点：

- 性能强悍：查询速度快，多张亿级表 join 也能秒级响应；
- 使用简单：兼容 MySQL 协议，用户使用门槛低；
- 支持高并发：满足大量用户同时查询；
- 支持多种数据模型：明细、聚合、更新和主键模型，可灵活配置 ETL 任务；
- 支持物化视图：可以自动路由到命中的物化视图，用户无感知；
- 支持多种导入方式：StreamLoad、SparkLoad、RoutineLoad，便于实时离线快速导入 StarRocks，流批一体。

因此，重构后的结构如下：



重构后的指标平台只有一个数据库，查询时利用 StarRocks 内部 ETL 将明细数据转存到临时表，后续的汇总从临时表查询，避免了反复扫描大表。

2.1 指标查询过程

当一个指标查询请求发起时，由于指标属性和用户想查看的信息不同，我们根据查询参数将查询拆解成若干子查询，子查询分为明细和汇总两类。

1）明细类子查询

a. 可累加的指标查询时间范围内的明细数据，以及去年和 2019 年同期的明细数据，这部分的明细会存储到临时表，后续查询都从这张表扫描，以避免对大表的频繁扫描；该表每天生成 T+1 分区，防止增加分区失败导致当天的指标查询无法进行。

```
-tarpresqls "

ALTER TABLE ${table} ADD PARTITION if not exists p${partition}

VALUES [('${zdt.addDay(1).format("yyyy-MM-dd")}'),('${zdt.addDay(2).format("yyyy-MM-dd")}'));

" \
```

b. 如果指标不可累加或 count（distinct）类，仅存储查询时间范围内的明细,不存储用户计算同环比的明细；

c. 当多个指标同时对相同维度进行查询时，将多个指标的数据 join 后以宽表模式存储。

2）汇总类子查询

这一类 sql 主要在明细的基础上根据用户的需要做相应的计算，相比旧版本在服务内部用 java 做汇总计算，这里全部借助了 StarRocks，主要的汇总功能有：

a. 指标卡汇总和同环比；

b. 折线图和维度下钻。

3） “缓存”

多维度特别是包含出发/到达城市组合的查询数据量非常大，耗时较长，同时避免相同的查询反复访问大表，我们增加了“缓存”功能，实现原理如下：

a. 记录初次查询的指标信息，主要包括维度和维度值，时间范围，指标原始计算 sql 的 MD5 值，以及是否查询成功；

b. 新的查询进入后，我们会在当天的记录中查找是否存在相同的查询。如果存在相同的查询，我们使用唯一的查询标识（groupkey）将当前查询指向上次已经执行过的查询。这样，我们可以直接读取上次查询的详细数据和汇总结果，从而提高查询效率。

因此这里的缓存非真实意义上的缓存，而是直接调用相同查询的结果。

2.2 数据同步

首先我们梳理了旧平台的数据源，从 300+ 指标的逻辑 sql 中提取了公共的 dwd 和 dim 表 51 张，并将这些数据统一同步至 StarRocks，但是对于一些指标使用的 dwd 表只出现一次的，依然将 dws 同步过来。

对于不同的 hive 表，我们使用了不同的 StarRocks 建表模型和同步方式，有以下几种：

a. 全量同步：主要针对一些数据量小的表，例如 shareout_trn.dim_ibu_alliance，大小为 608k；

b. 增量分区同步：每天同步 hive 表中 T-1 的分区，各分区之间独立；

c. 更新同步：火车票 BU 的一些订单数据由于涉及到预售和订单状态的变更，变更的数据时间跨度比较大，将跨度范围内的数据全部更新代价比较高，因此使用更新模型。

数据导入更新模型直接需要计算 T-1 和 T-2 分区有差异的数据，这里将所有字段使用 concat_ws（'|',***）拼接后取 hash 值，之后 join 找到 hash 值不一致的数据。

模型 KEY 设置：

UNIQUE KEY(`order_id`)

取两天有差异的数据：

select

t1.*

from

(select … where d='${cur_day}') as t1

left join

(select … where d='${pre_day}') as t2

on t1.business_pk_id=t2.business_pk_id

where t1.hash_code!=t2.hash_code or t2.order_id is null

d. 每天同步当月数据：如国际火车的访问数据量较小，每天一个分区会导致 StarRocks 集群有很多小的 bucket，分桶数太多会导致元数据压力比较大，数据导入导出时也会受到一些影响，因此我们按月设置分区，每天同步当月的数据。

时间范围：

startdate='${zdt.format("yyyy-MM-01")}'

endDate='${zdt.add(2,1).format("yyyy-MM-01")}'

表设计：

PARTITION BY RANGE(dt)(Start("2019-01-01") End("2023-03-01") Every(Interval 1 month))

DISTRIBUTED BY HASH(分桶字段) BUCKETS 桶的数量

PROPERTIES (

"dynamic_partition.enable" = "true",

"dynamic_partition.prefix" = "p",

"dynamic_partition.time_unit" = "month",

"dynamic_partition.end" = "1");

datax 配置：

-temporary_partitions "tp${partition}" \

-tarpresqls "

ALTER TABLE ${table} DROP TEMPORARY PARTITION if exists tp${partition};

ALTER TABLE ${table} ADD PARTITION if not exists p${partition} VALUES [('${startdate}'),('${endDate}'));

ALTER TABLE ${table} ADD TEMPORARY PARTITION tp${partition} VALUES [('${startdate}'),('${endDate}'));

" \

-tarpostsqls "

ALTER TABLE ${table} REPLACE PARTITION (p${partition}) WITH TEMPORARY PARTITION (tp${partition});"

此外，对于 UBT 类数据，数据量级非常大，并且常见用于查询 PV，UV 和停留时长等比较固定的场景，于是我们从中抽取出三张表：

ubt_for_pv: 每天按维度汇总 count（uid），每天数据大小只有几十 K；

ubt_for_duration: 每天按维度汇总 sum（duration），如需要计算平均停留时长除以对应的 pv 即可；

ubt_for_uv: 每天按维度去重，尽最大可能减少数据量。

最后，鉴于上游表的迭代可能带来的数据的不稳定，我们对需要同步的表的数据量做了监控，若发现当天的数据量波动超过 3sigma，监控任务自动发出邮件告警，这些 job 的同步都在 15 分钟内完成。

三、Starrocks 使用经验分享

在指标平台重构的过程中我们也遇到了一些问题，与数据和查询相关的有以下几个：

3.1 建表经验

首先是 buckets 设置不合理，多数是设置过多，通常一个桶的数据量在 500MB~1GB 为好，个别表设置的桶数量太少，导致查询时间长；其次是分区不合理，有些表没有设置分区，有些设置的分区后每个分区数据量很小，优化建议是将不常访问的数据按月分区，经常访问的数据按日分区。

3.2 数据查询

由于指标的查询 sql 之前是针对不同引擎编写，很多引擎是没有索引的，比如 Presto。StarRocks 有丰富的索引功能，统一至 StarRocks 希望利用索引加速查询，因此过滤条件中最好不要加函数，比如 select c1 from t1 where upper(employeeid) = upper(' s1')修改成 select c1 from t1 where employeeid in(upper(' s1'), lower(' s1'))。

另外很多 sql 没有使用分区，在 StarRocks 中将会全表扫描造成资源浪费。

3.3 函数问题

StarRocks 的 split 函数结果的下标从 1 开始，而 sparksql 等引擎对应的是从 0 开始，导致 sql 在 StarRocks 执行查询的时候不报错但是结果错误。

select split('a,b,c',',')[0] StarRocks 查询结果为空，其他引擎查询结果为'a'

select split('a,b,c',',')[1] StarRocks 查询结果为'a'，其他引擎查询结果为'b'

四、查询性能大幅提升

指标平台的重构主要是为了解决查询性能的问题，并且重构后也基本达到了预期。重构之前，复杂查询需要数分钟的时间才能完成。特别对于火车票相关指标，诸如出票票量指标，如果带上出发和到达城市查询，可能需要等待 30 分钟以上，并且查询失败率较高。而在重构后，查询时间大大缩短，复杂查询在 10s 左右，并且 P99 在 2 秒之内，因此整体体验得到显著提升，用户查询次数相比改造前也有了翻倍的增长。

此外，现在新指标系统还丰富了更多功能，比如同环比和维度下钻计算。得益于 StarRocks 的并发能力，我们可以在生成子查询 SQL 后并发提交，从而大幅度减少响应时间，使得用户在进行维度下钻时几乎无需等待即可快速获取所需数据。

五、 后续优化方向

目前，UV 类的 Count Distinct 查询是基于存储了大量明细数据的方式进行的。然而，对于部分指标，我们可以尝试使用 Bitmap 来减少不必要的明细数据存储空间，并且更重要的是可以提高查询速度。在接下来的工作中，我们计划尝试这种方案，以进一步优化 UV 类指标的查询性能。

对于全量或增量更新的表使用聚合模型，聚合模型会对导入后具有相同维度的数据做预聚合，查询的时候减少扫描数据的行数达到提升查询速度的目的。

当前的指标平台计算过程将所需的数据写入临时表，后续改成使用物化视图，在达到同样效果的情况下减少了复杂度。

节约 60%开发工时，离在线一体化数仓系统在携程旅游的落地实践

【作者简介】Chengrui，携程后端开发专家，关注实时数据处理、AI 基础平台建设以及数据产品等领域。

本文主要介绍离在线数据仓库建设在携程旅游团队的落地与实践，将从业务痛点、业务目标、项目架构、项目建设等维度展开。

一、业务痛点

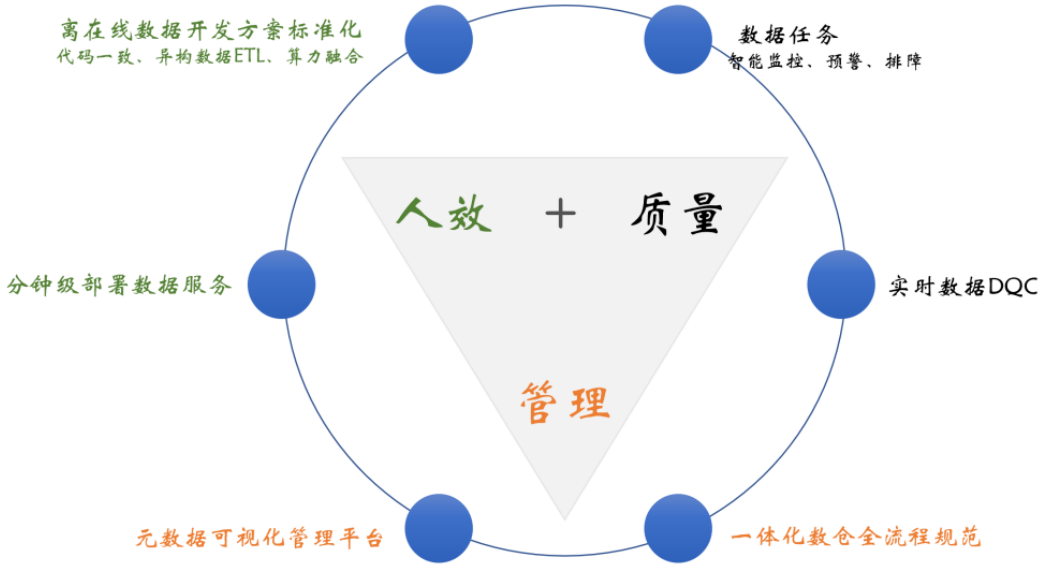
随着数据实时化需求增多，离线数仓暴露出来的业务痛点也越来越多，例如：

- 实时需求烟囱开发模式
- 中间数据可复用性差
- 离在线数据开发割裂
- 数据生产->服务周期长
- 实时表/任务杂乱、无法管理
- 实时血缘/基本信息/监控等缺失
- 实时数据 质量监控无工具
- 实时任务 运维门槛高 质量体系弱

这类典型的问题，会对我们的人效、质量、管理等方面带来较大考验，亟待一个体系化的平台来解决。

二、业务目标

围绕已知业务痛点，依托于公司现有的计算资源、存储资源、离线数仓标准规范等，我们的目标是在人效、质量、管理这几个层面进行系统建设。如下图：



2.1 人效层面

实现离在线数据开发方案标准化，如标准化数据处理、离在线代码兼容、算力融合等。

分钟级数据部署，实现 BI 同学层面的数据接口注册、发布、调试等可视化操作。

2.2 质量层面

数据内容 DQC，如内容对不对、全不全、是否及时、是否离在线一致等。

数据任务预警，如有无延迟、有无反压、吞吐怎么样、系统资源够不够等。

2.3 管理层面

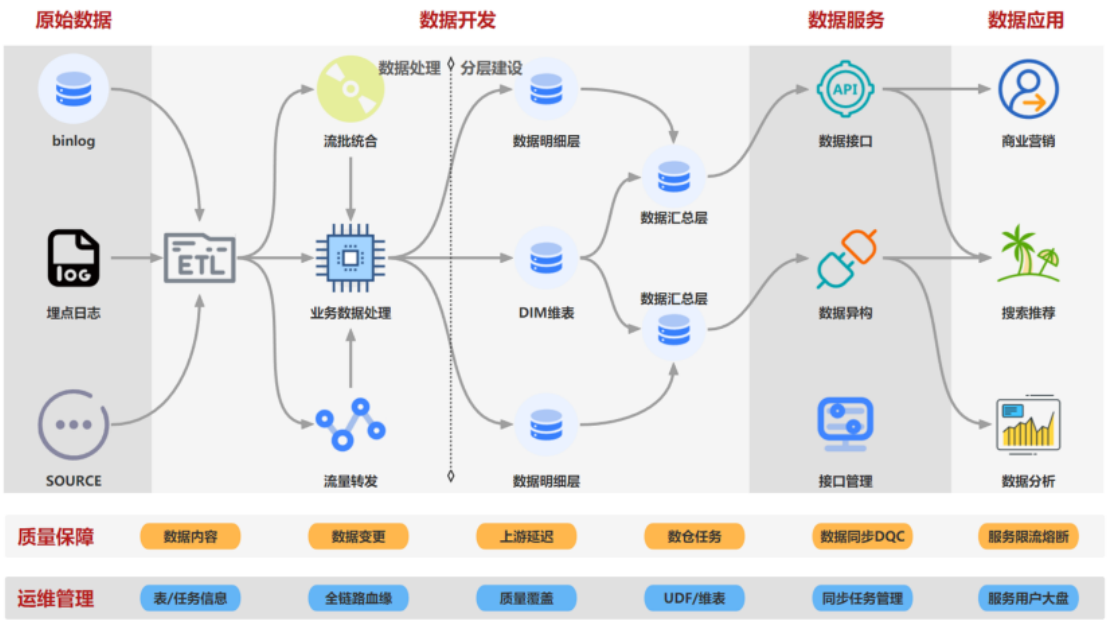
可视化管理平台，如全链路血缘、数据表/任务、质量覆盖率等基本信息。

一体化数仓全流程规范，如数据建模规范、数据质量规范、数据治理规范、存储选型规范等。

三、项目架构

项目架构如下图，该系统主要包括：原始数据 -> 数据开发 -> 数据服务 -> 数据质量 -> 数据管理等模块，提供实时数据秒级处理、数据服务分钟级部署的能力，供实时数据开发同学、后端数据服务开发使用。

不同数据来源的数据首先经过标准化 ETL 组件进行数据标准化，并经过流量转发工具进行数据预处理，使用流批融合工具以及业务数据处理模块进行分层分域建设，生产好的数据使用数据服务模块直接将数据进行数据 api 部署，最终供业务应用使用，整个链路会有对应的质量和运维保障体系。



四、项目建设

4.1 数据开发

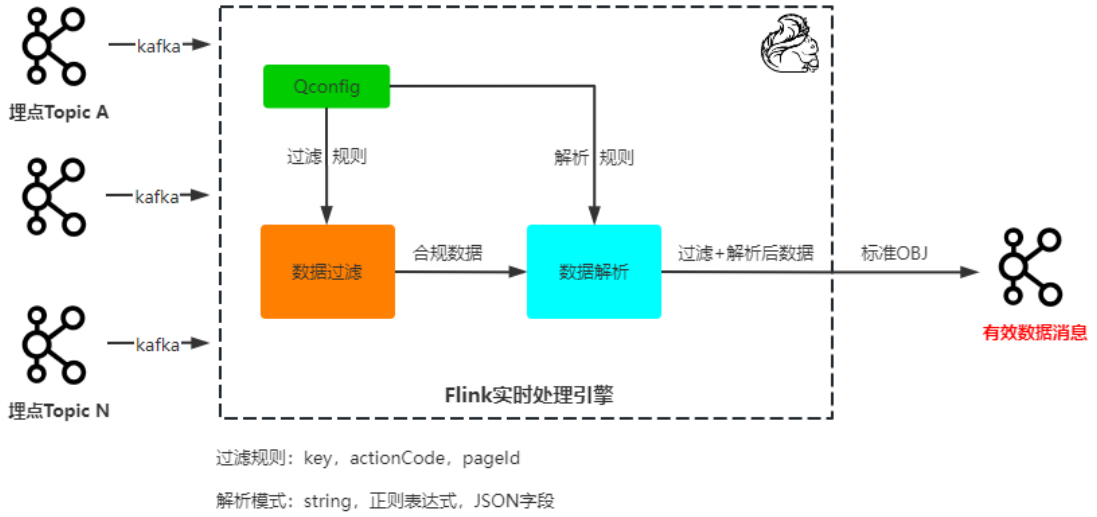
该模块主要包含数据预处理工具、数据开发方案选型。

4.1.1 流量转发工具

由于入口多、流量大，主要存在如下问题：

- 同维度的数据来源、解析方式可能有多种
- 使用到的埋点数据占总量的比例大约 20%，全量消费资源浪费严重，且每个下游都会重复操作
- 新增埋点后，数据处理需要开发介入（极端情况下涉及到全部使用方）

如下图，流量转发工具，具备动态接入多个数据源，并且做简单的数据处理，并且将有效数据进行标准化后写入下游，可解决上述问题。



4.1.2 业务数据处理方案演进

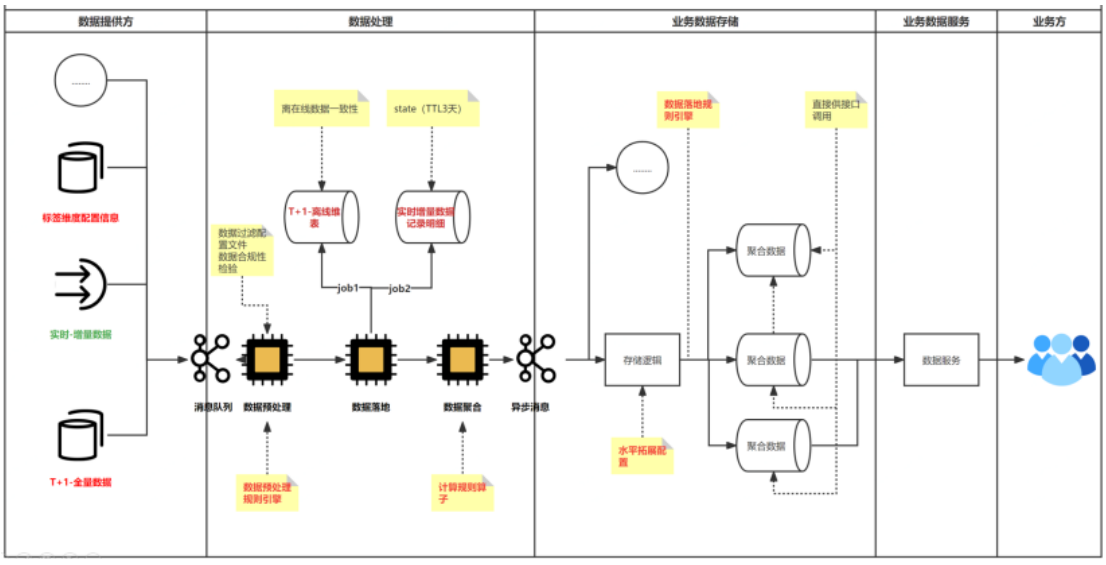
方案 1-离在线数据简单融合

背景

由于最开始的时候业务需求比较单一，如计算用户历史的实时订单量、聚合用户历史购买过的景点信息等。这类简单需求可以抽象成离线数据和实时数据简单聚合，如数值型的加减乘除、字符型的 append、去重汇总等。

解决方案

如下图，其中数据提供方：提供标准化的 T+1 和实时数据接入；数据处理：T+1 与实时数据融合；一致性校验；动态规则引擎处理等；数据存储：支持聚合数据水平扩展；标签映射等。



方案 2 - 支持 SQL

背景

虽然说方案 1 有如下优势：

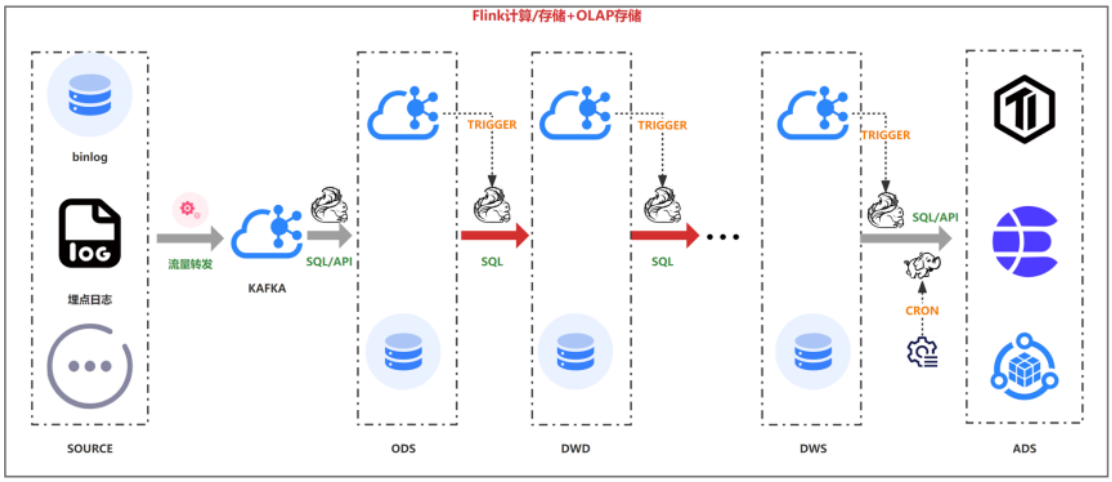
- 分层简单，时效性强
- 规则配置响应迅速，可承接大量的复杂 UDF
- 规则引擎等处理
- 兼容整个 java 生态

但是也存在明显劣势：

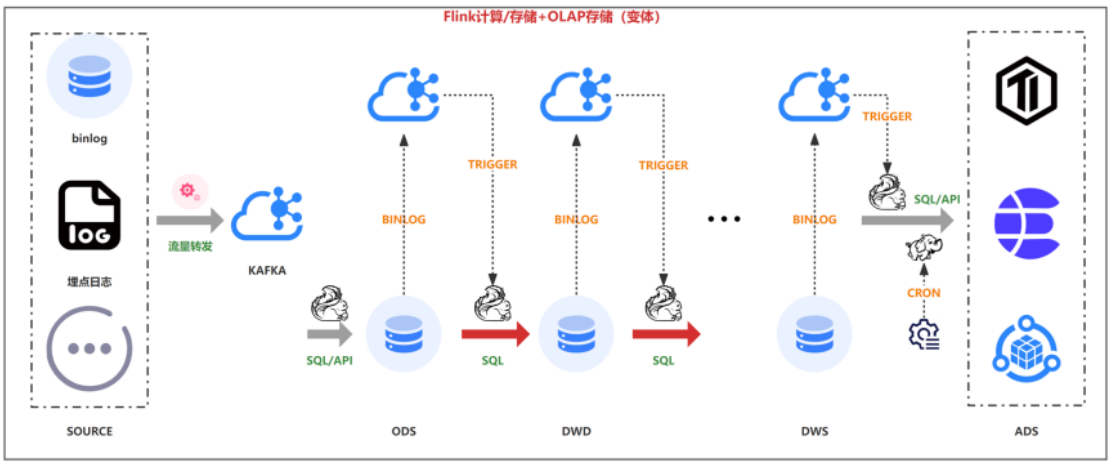
- BI SQL 开发人员基本无法介入、强依赖开发
- SQL 很多场景，使用 java 开发成本高，稳定性差
- 没有有效的数据分层
- 过程数据基本不可用，如果要保存过程数据，需要重复计算，浪费计算资源

解决方案

如下图，kafka 承载数据分层功能，Flink SQL 的计算引擎，OLAP 承载数据存储、分层查询，完成典型的数仓系统分层建设。



但是由于 kafka 和 olap 存储引擎是两个个体，可能会存在数据不一致的情况，比如 kafka 正常，数据库异常，会导致中间分层的数据异常，但是最终结果正常。为了解决上述问题，如下图，采用了传统数据库使用的 binlog 模式开发，kafka 数据强依赖 DB 的数据变更，这样最终结果强依赖中间分层结果，还是不能避免组件 big 导致的数据不一致问题，但大部分场景已经基本可用。



方案 3

背景

虽然说方案 2 有如下优势：

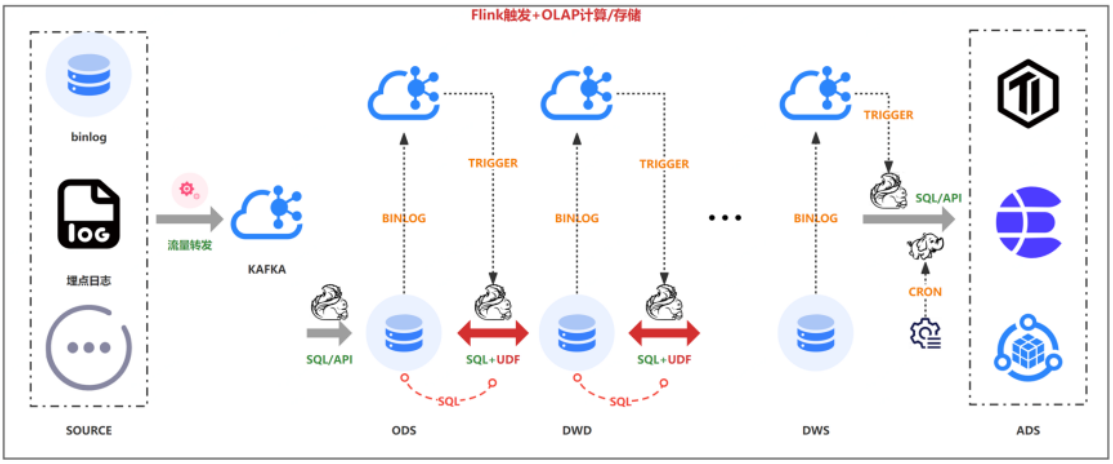
- SQL 化
- 天然分层查询

但是也存在明显劣势：

- 数据不一致的问题
- binlog 在 insert 的时候没啥问题，但是更新和删除不好搞，而且更新的时候要做大量的去重操作，sql 很不友好
- 长时间数据聚合，部分算子如 max、min 等 flink 状态大，容易不稳定
- 还要考虑 kafka 数据乱序，导致的数据覆盖问题

解决方案

如下图借用存储引擎的计算能力，kafka 的 binlog 只是作为数据计算的触发逻辑，直接使用 Flink UDF 进行直连 DB 查询。



优势：

- SQL 化
- 天然分层查询
- 数据一致
- FLink 状态小
- 可支持长时间的持久化数据聚合
- 无需关心 binlog 乱序、update 等带来的问题

劣势：

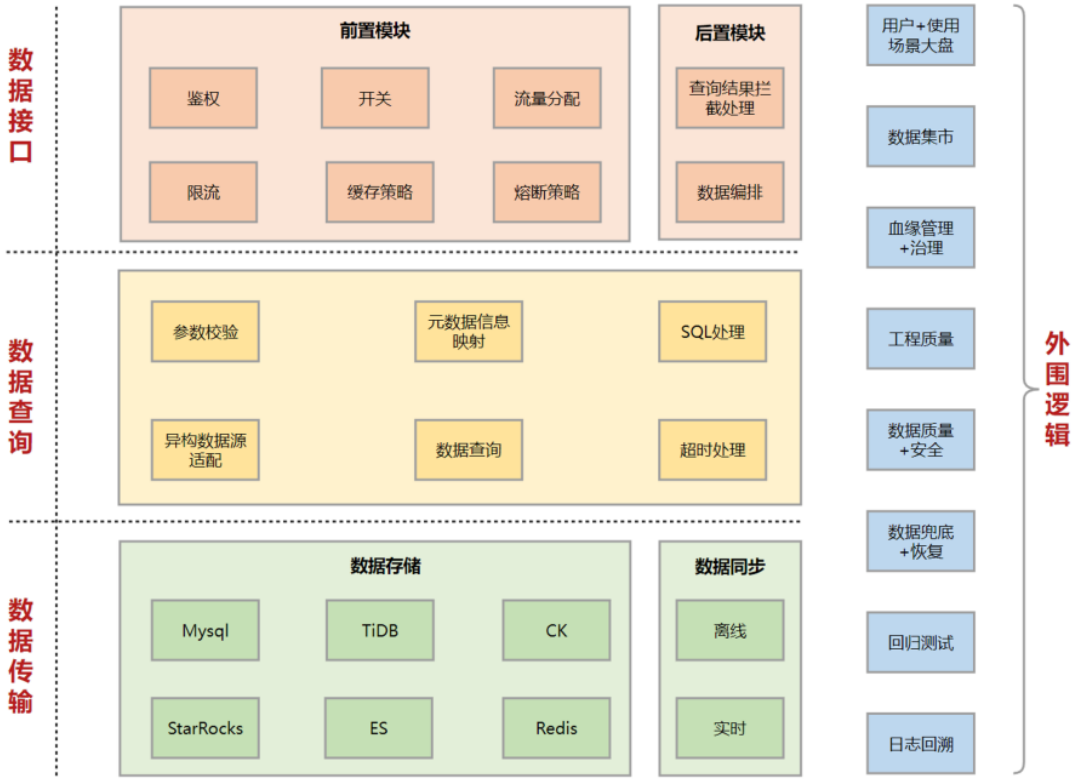
- 并发扛不起来，强依赖 olap 引擎性能，我们在数据源的时候会 window 限流，或者水平扩容 db
- sink 时与回撤流结合被打断，比如：group by，其实就是无脑的 upsert，udf 的聚合没法替代 flink 原生的聚合

各个方案都有适用场景，需要根据不同的业务场景和延迟需求，进行方案选型。目前我们 86%的场景都可以使用方案 3 进行承接，并且由于 Flink 1.16 各类离在线一体的特性加持，后期基本可覆盖全部场景。

4.2 数据服务

该模块提供了数据同步 -> 数据存储 -> 数据查询 -> 数据服务等能力，简单场景可实现分钟级的数据服务部署能力，可节约 90%的开发工时。实现了离线数据 DQC 强依赖、工程侧 DQC 异常兜底、客户端->接口级别的资源隔离/限流/熔断、全链路血缘（客户端->服务端->表->hive 表->hive 血缘）管理等，提供了按需进行各类性能要求接口部署和运维保障能力。

架构如下：



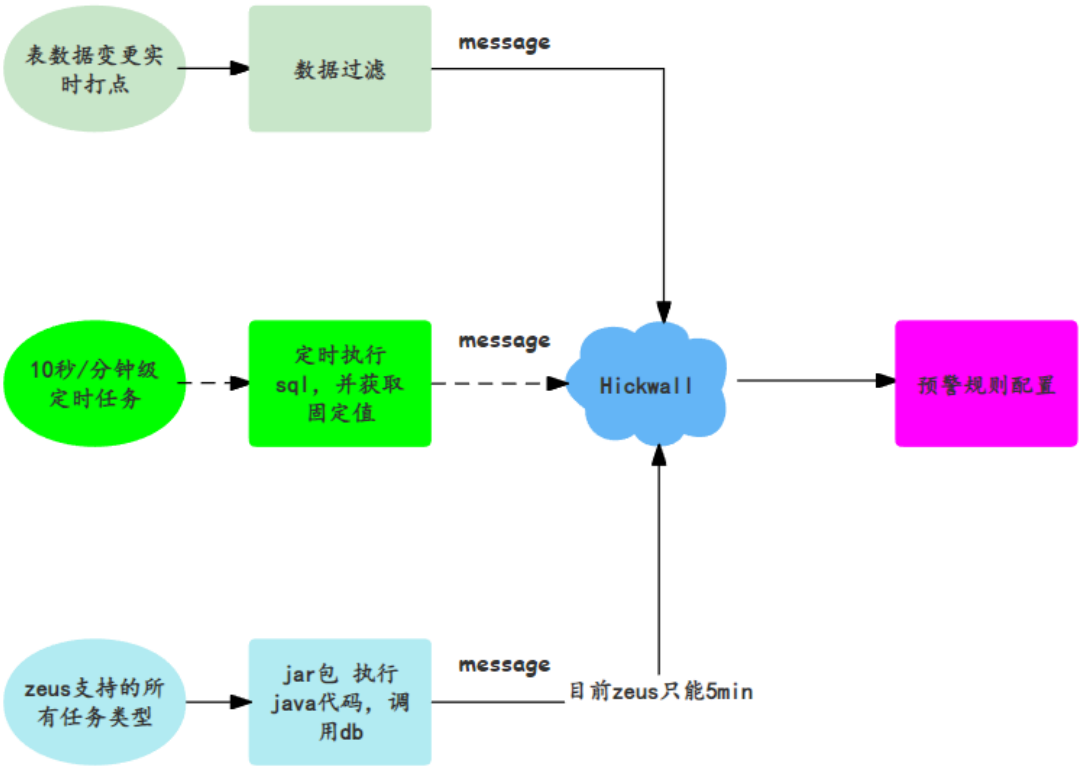
4.3 数据质量

该模块主要分为数据内容质量和数据任务质量。

4.3.1 数据内容

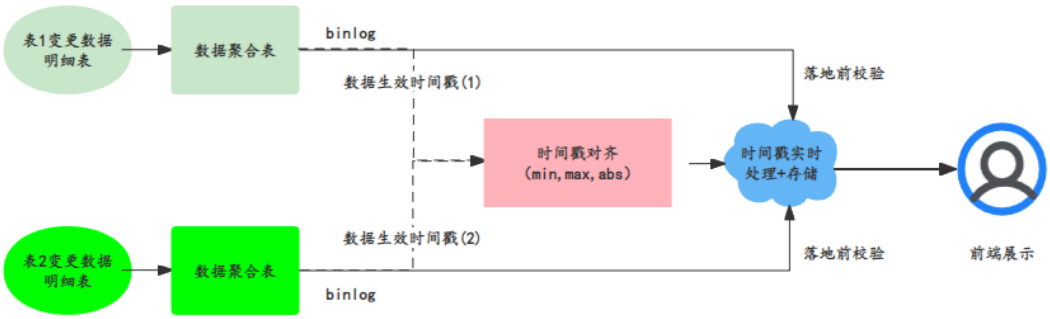
正确性/及时性/稳定性

该部分又分为数据操作变化、数据内容一致性、数据读取一致性、数据正确性/及时性等。如下图所示，数据变更：如果异常，可将数据打入公司的 hickwall 告警中台，并根据预警规则告警。数据内容：会有定时任务，执行用户自定义的 sql 语句，将数据写入告警中台，可实现秒级和分钟级预警。



读取一致性

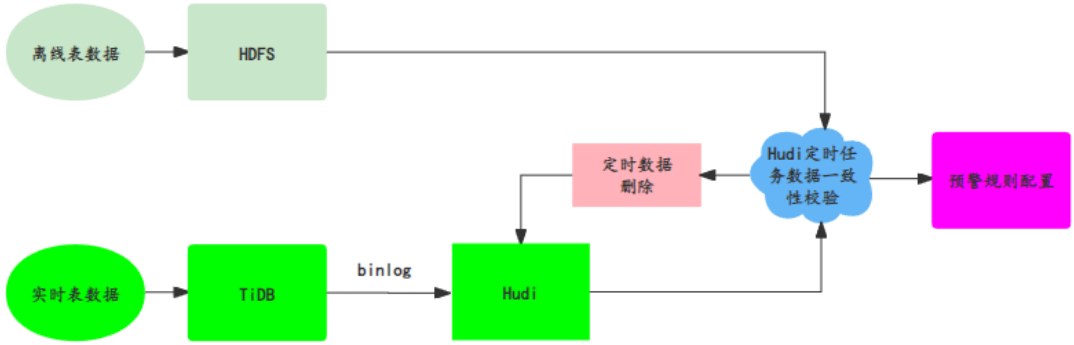
如下图，数据读取时，如果存在跨表的联合查询，如果其中某张表出现问题，大多数情况下不会展示错误数据，只会展示历史上的正确数据，待该表恢复后才会全部展示。



如：外露需要将表 1 和表 2 的数据做除法（表 1/表 2），如果表 2 数据生产异常，最近 2 小时没数据，在外露给用户时，业务需要只是展示 2 小时之前的数据，异常数据给出前端异常提醒 参照 flink watermark 的概念，将正确数据对其进行外显。

离在线一致性

关于离线和实时的数据一致性。如下图，我们采用较为简单的方法，直接将实时数据同步至 hudi，并且使用 hudi 进行离线和实时数据对比，打入告警中台。



4.3.2 数据任务

上游任务

依托公司自定义预警埋点、告警中台、计算平台等工具，可将上游的消息队列是否延迟、量是否异常等关键指标进行监控预警。

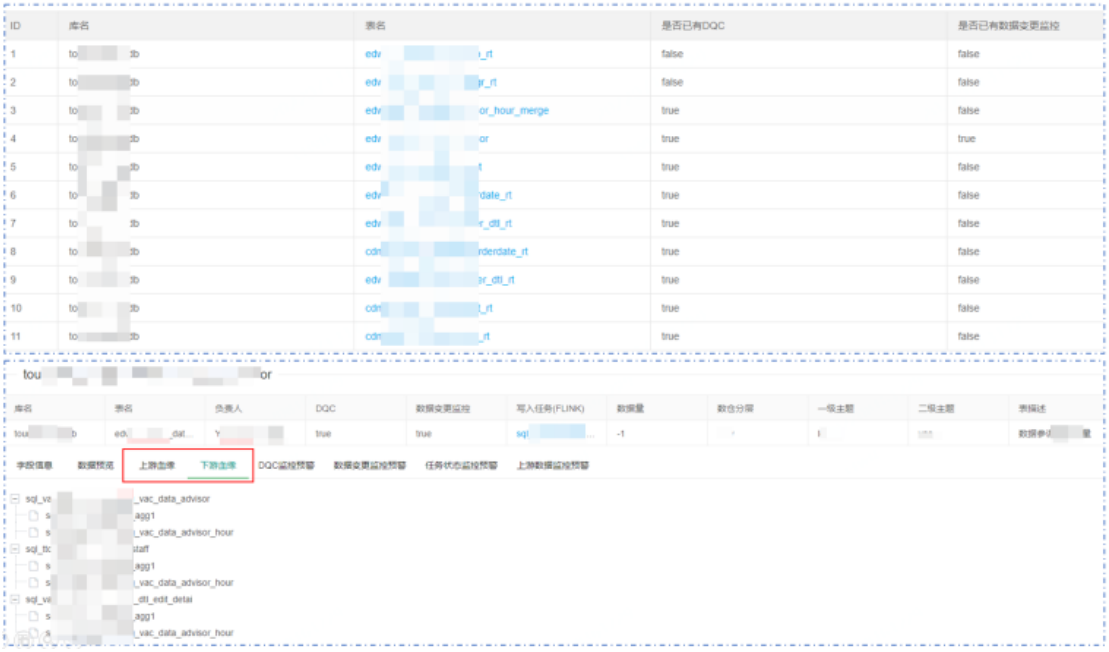
当前任务

可将数据处理任务的吞吐、延迟、反压、资源等关键指标进行监控预警，避免数据任务长时间异常。

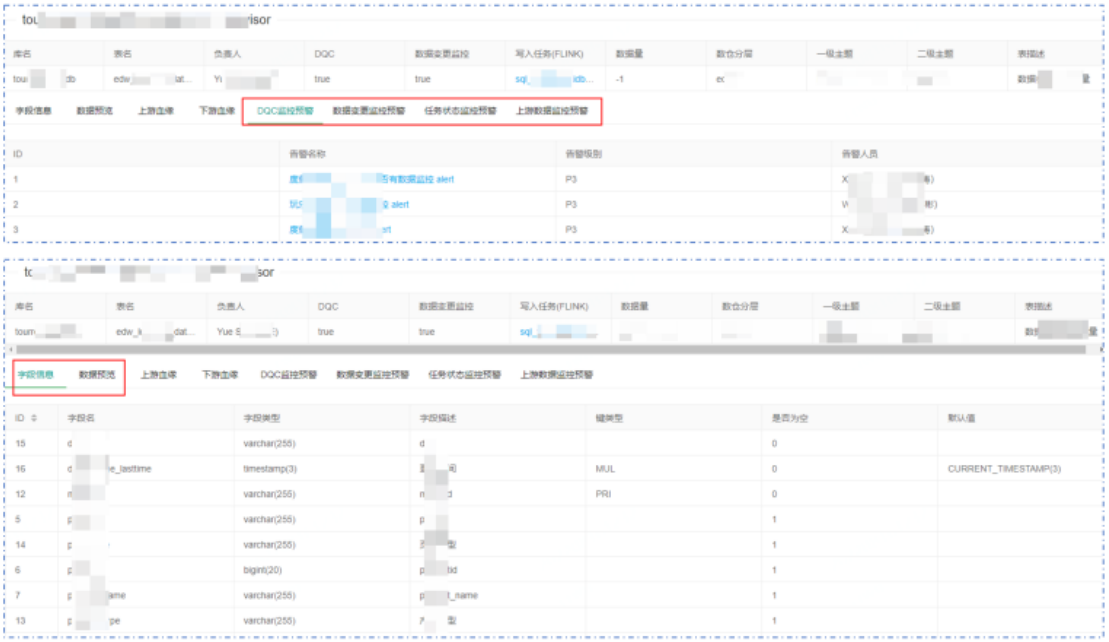
4.4 数据管理

该模块可将数据处理、质量等各模块进行串联，提供可视化的管理平台，如：表血缘/基本信息、DQC 配置、任务状态、监控等。

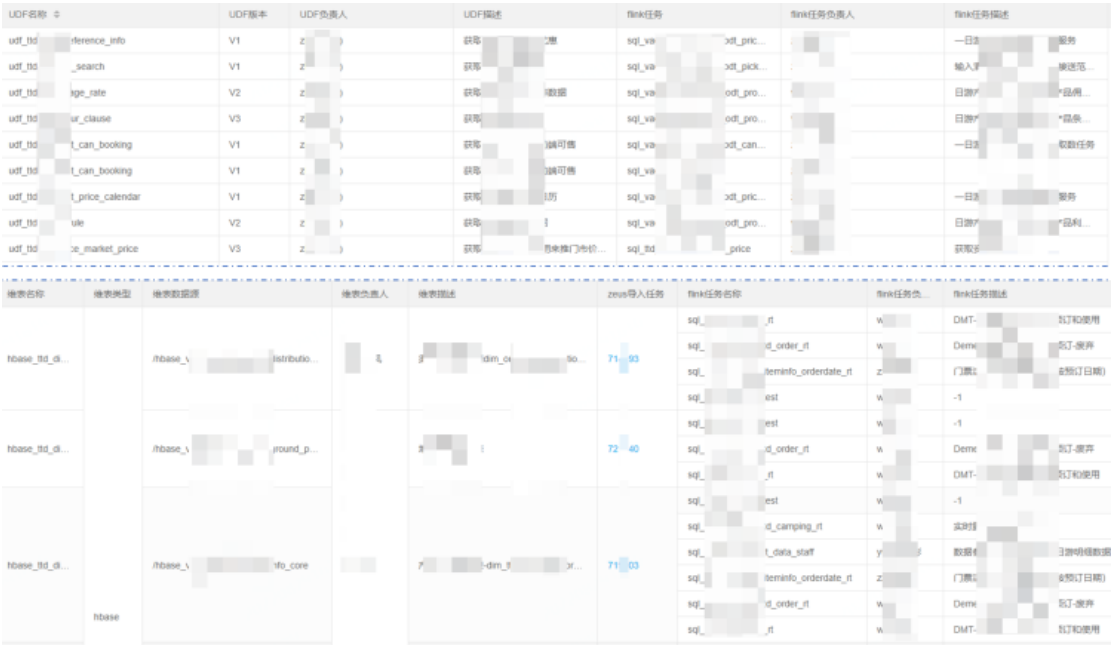
下图为各数据表上下游数据生产任务血缘关系。



下图为数据表质量信息详情



下图为各类 UDF 表的基本信息汇总



五、展望

目前该系统基本上已经能承接团队绝大多数数据开发需求，后期我们会在可靠性、稳定性、易用性等层面继续探索，如完善整个数据治理体系、建设自动数据恢复工具、排障运维智能组件、服务分析一体化探索等。

贝叶斯结构模型在全量营销效果评估的应用

【作者简介】Yiwen，携程数据分析师，专注用户增长、因果推断、数据科学等领域。

一、背景

如何科学地推断某个产品策略对观测指标产生的效应非常重要，这能够帮助产品和运营更精准地得到该策略的价值，从而进行后续方向的迭代及调整。

在因果推断框架下，效果评估的黄金准则一定是“AB 实验”，因为实验的分流被认为是完全随机且均匀的，在此基础上对比实验组与对照组的指标差异就可以体现某个干预带来的增量值。但是很多场景下，我们较难进行严格的 AB 实验，例如对于酒店的定价；现金奖励的发放等等，不适宜向不同人群展现不同的内容。对于这些问题，我们会采取因果推断的方法来进行策略的效果评估。

本文主要介绍 BSTS 模型原理以及 CausalImpact 对模型的代码实现，旨在面对一些具有特定周期性特点的数据时，更精准科学地进行因果效应值的估计。下文将首先对模型原理进行简要阐释；随后利用模拟数据展示代码逻辑，最后在具体的业务场景中进行实践。

二、现有方法及潜在问题

大部分运营和产品在评估一些全量上线的策略效果时，最常用的方式就是看上线前后的效果差异。但这种方法最大的问题在于其假设前提：假设上线的功能是唯一影响效果的变量（即没有任何其他干预和混淆变量），但这个假设现实中往往很难实现。

于是我们尝试使用更多因果推断的方法，例如 PSM（倾向分匹配法），在所有非实验组的用户群中，找到与实验组用户的特征非常相似的一群人，将他们的指标数据（例如下单率，订单收益等等）与实验组的用户进行对比，从而体现出干预带来的影响。但这个方法较为依赖选取的用户特征与最后的匹配效果。

再例如 SCM（合成控制方法），利用一些未受干预的地区合成一个“类似的虚拟地区”来与“上线策略的地区”进行整体的对比。但这也需要一个关键假设：可以找到长期变化趋势高度同步的地区来进行对照，而这个条件往往也很难实现。

进而在传统 SCM 的基础上，我们企图通过类似集成学习的方法，将多个未干预的对照组作为输入值，再结合实验组自身长期的时间序列波动情况，拟合出一个未受干预的虚拟对照组，从而将“对照组与实验组高度同步”的强假设降为弱假设。本文介绍的 BSTS 模型就是用来刻画某种“长期的时间序列波动”的数据模型，CausalImpact 是用来针对这样的数据进行因果效应值的估计。下文中我们将详细介绍这两个工具。

三、模型介绍

BSTS 模型 （Bayesian Structured Time Series）称为“贝叶斯结构化时间序列”，正如其名，它的主要特点体现在：

- 适用于有结构特征的时间序列数据
- 利用贝叶斯的思想来进行参数估计

结构化的时间序列数据在日常生活中不少见，尤其像携程这样的 OTA 行业，平台的订单情况其实是有一定时间规律的，例如周末和节假日是订单高峰期；周中是订单平峰期等。另一方面，贝叶斯的思想是指在得到样本数据之前，即对要估计的参数有一些“先天认知”），随后基于这样的认知，结合样本数据再得到后验分布（如下方公式展示）

$$P(\theta|\text{data}) = \frac{P(\text{data}|\theta) * P(\theta)}{\text{P(data)}}$$

故 BSTS 模型主要即对结构化时序数据进行模型拟合及预测，在拟合的过程中使用到了贝叶斯的先验思想。其好处是能够给出预测值的置信区间，使得预测结果更科学可信。下文将对这几种思想逐一进行介绍。

3.1 状态空间模型

结构化的时间序列数据是指某一观测数据的背后其实隐藏着随时间变化而变化的不同状态，其中观测值与状态值之间有对应关系；不同时刻的状态之间也有转换关系。我们一般用以下状态空间模型来刻画这两种映射逻辑：

$$y_t = Z_t^T \alpha_t + \epsilon_t \quad (1)$$
$$\alpha_{t+1} = T_t \alpha_t + R_t \eta_t \quad (2)$$

(1) 称为观测方程，反映观测值与其背后隐藏状态的关系；(2) 称为状态方程，反映随时间推移各个状态之间的转换。Z_t^T；T_t；R_t都是不同变量之间的“关系映射矩阵”；ϵ_t,η_t是独立于其他变量且服从正态分布的噪声。所谓数据的“结构化”，主要包括：

- Linear Local Trend（局部趋势）：一定时间内的单调性（单调上升或下降）
- Seasonality（季节性因子）：固定长度的变化，类似于一年四季的温度变化
- Cyclical（周期性）：类似季节性但波动时间不固定，波动频率也不固定的变化

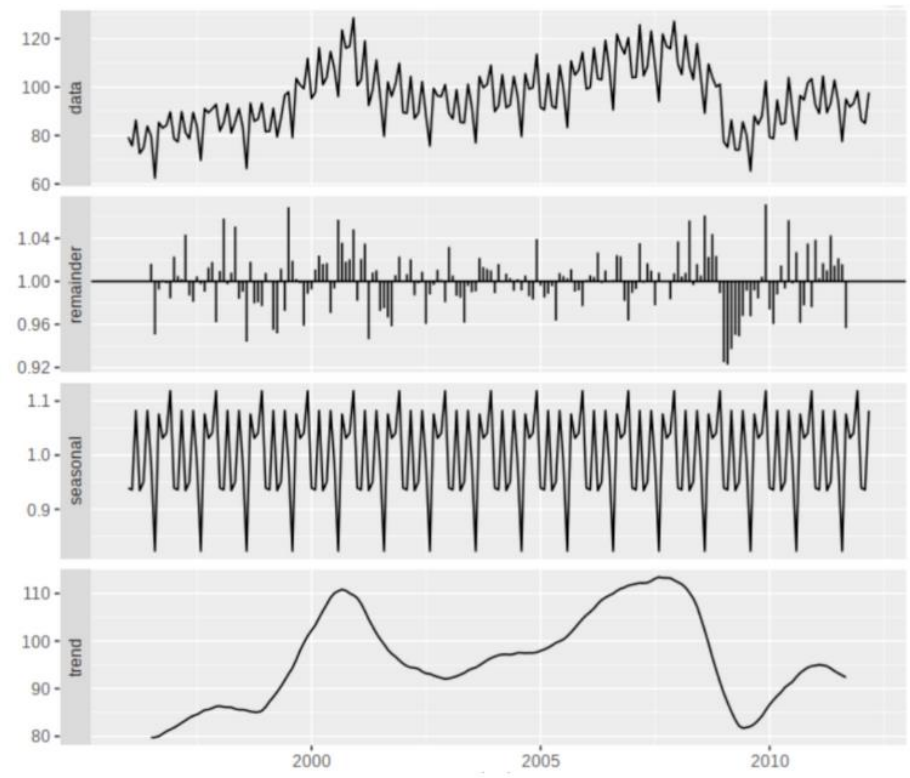


图 3-1：观测数据及其结构化元素。第一张图体现原数据的波动情况；第二张体现季节性因子的情况；第三张图体现局部趋势的情况。

如果希望在映射关系中加入协变量 X，可以将(1)拓展为：

$$y_t = Z_t^T \alpha_t + \beta X_t + \epsilon_t \quad (3)$$

其中βXt 表示协变量 X 与观测数据之间的关系，如果协变量项表现很好（如有显著影响）的话，那对应的 local trend 就会相对较弱。上述三个方程中的参数将在后文中展示估计方式。

3.2 贝叶斯及 MCMC（马尔可夫蒙特卡洛方法）

假设状态方程(2)中各个时刻的状态序列为 $\alpha=\{\alpha_1,...,\alpha_n\}$，θ表示模型中所有的参数。我们现在希望对 θ 进行估计，核心步骤如下：

- 对θ设置先验分布ρ(θ)以及初始状态的分布$\rho(\alpha_0|\theta)$
- 构造马尔科夫链，用 MCMC 方法得到ρ(y|α,θ)

- 通过贝叶斯公式计算得到参数的后验分布ρ(α,θ|y)

下面对于各个步骤中用到的方法进行简要说明：

1）贝叶斯估计：BSTS 模型的一大特点就是在参数估计上使用了贝叶斯估计，即在估计之前先给出参数设置先验分布，随后再结合样本数据给出参数的后验分布。不同类型的参数一般有一些常用的先验分布，例如均值一般使用正态分布 $\beta \sim N(\beta_0, \sigma_0^2)$，方差使用 inverse-Gamma 分布，$\frac{1}{\sigma^2} \sim \text{Gamma}(\frac{v}{2}, \frac{s}{2});$ 协方差矩阵可以使用 IW 分布等等。值得注意的是，先验分布的设置一定程度上会影响后续 MCMC 收敛的情况以及后验分布的准确性，因此并不能太过随意地设置先验分布，应尽可能多地根据实际数据推导出最合适的先验分布，或是比较各先验分布下后验分布和似然函数的值来进行选择。

2）MCMC 方法：我们尝试构造一条马尔可夫链（一种特殊的序列，当前时刻的状态值仅与前一时刻的状态值有关，最终序列会收敛到某个稳定的分布），使得其最终收敛的稳态分布就是参数的后验分布。这一过程我们可以通过 Gibbs 采样实现：设置先验分布之后，从初始状态 α_0 出发，每次固定 α 采样 θ；再固定 θ 采样 α，逐渐一次次更新两组参数，最终形成一条服从马尔可夫性质的链路{(α, θ)}，可以证明其稳态收敛的分布就是$p(\alpha, \theta | y_{1:n})$，其中 $y_{1:n}$ 代表所有的观测数据。

3）预测值估计：得到$p(\alpha, \theta | y_{1:n})$之后，我们从该分布中对（α，θ）进行采样，再代入状态空间方程(1)中对 y 进行预测，得到$|p(\tilde{y}_{n+1:m} | y_{1:n}, x_{1:m}))$，其中$\tilde{\boldsymbol{y}}_{n+1:m}$表示时间点 n 之后 y 的预测值。

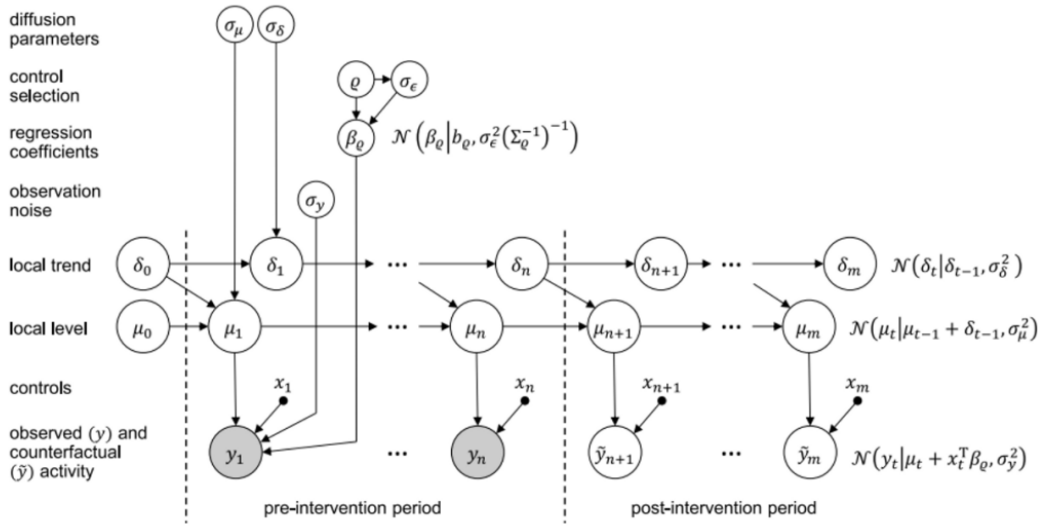


图 3-2：展示某结构化时序数据及其背后各个状态转换的过程。状态 α 包含 Local trend: δ_t（局部趋势); local level:μ_t (局部趋势的均值）以及协变量$x, y_{1:n}$表示所有的观测数据; $\tilde{y}_{n+1:m}$表示根据状态模型得到的预测数据。σ_μ;σ_δ分别表示μ_t, δ_t的标准差这些参数均通过 MCMC 的方式得到估计。

四、模型应用与代码实现

以上我们给出了 BSTS 模型及 MCMC 方法的简要理论推导及结果输出，核心目的就是对观测值 y 做出预测。接下来我们将介绍如何在因果推断场景中应用 BSTS 模型。

在对政策的效果评估上，我们核心想要的是观测对象“反事实值”，例如“如果没有这个广告投放，用户的浏览情况会怎样？”相较于传统的 PSM 或 SCM 方法，BSTS 胜在其能够对于时间序列数据进行效果评估；同时利用贝叶斯估计输出反事实值 y 的预测，并给出预测值的置信区间，能一定程度上降低反事实值预测的波动性，提升效应评估的准确性与稳定性。

在实践应用上，可以通过谷歌开源的 CausalImpact 包来实现 BSTS 模型，在 Python 和 R 中均可调用。

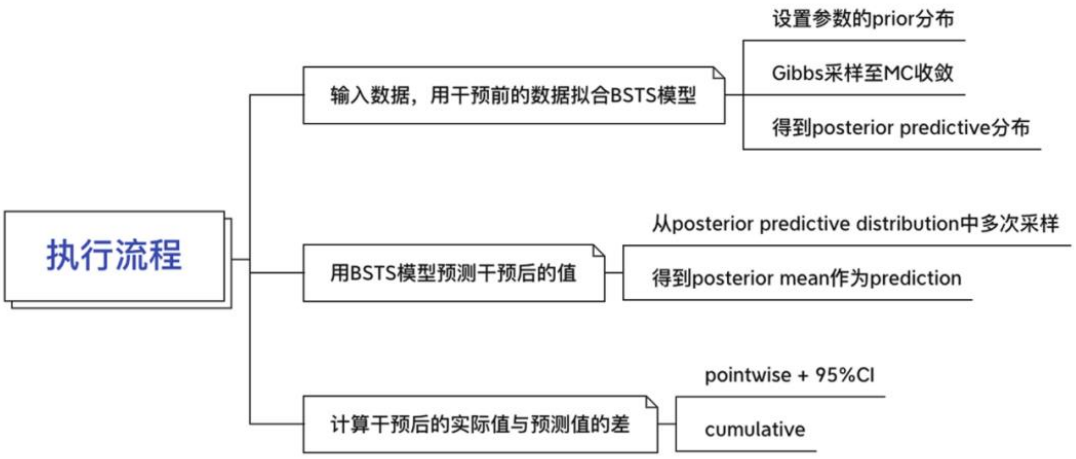


图 4-1：展示执行代码时的三个步骤：训练 BSTS 模型；反事实值预测；计算因果效应值，包括效应值的点估计及置信区间。

4.1 代码实现

下面通过模拟数据展示代码的具体命令

```
import tensorflow as tf

import tensorflow_probability as tfp
```

```
from causalimpact import CausalImpact

# 模型初始化 - 自定义时间序列数据:

def plot_time_series_components(ci):

        component_dists = tfp.sts.decompose_by_component(ci.model,
ci.observed_time_series, ci.model_samples)

            num_components = len(component_dists)

mu, sig = ci.mu_sig if ci.mu_sig is not None else 0.0, 1.0

for i, (component, component_dist) in enumerate(component_dists.items()):

                component_mean = component_dist.mean().numpy()

                component_stddev = component_dist.stddev().numpy()

# 自定义观测方程以及真实值 y:

def plot_forecast_components(ci):

                component_forecasts = tfp.sts.decompose_forecast_by_component(ci.model,
ci.posterior_dist, ci.model_samples)

            num_components = len(component_forecasts)

            mu, sig = ci.mu_sig if ci.mu_sig is not None else 0.0, 1.0

            for i, (component, component_dist) in enumerate(component_forecasts.items()):

                component_mean = component_dist.mean().numpy()

                component_stddev = component_dist.stddev().numpy()

# 生成模拟数据，包括一个实验组数据（有干预）以及两条对照组数据（无干预）

observed_stddev, observed_initial = (tf.convert_to_tensor(value=1,
dtype=tf.float32),tf.convert_to_tensor(value=0., dtype=tf.float32))

level_scale_prior = tfd.LogNormal(loc=tf.math.log(0.05 * observed_stddev), scale=1,
name='level_scale_prior')  # 设置先验分布
```

```
initial_state_prior = tfd.MultivariateNormalDiag(loc=observed_initial[..., tf.newaxis],
scale_diag=(tf.abs(observed_initial) + observed_stddev)[..., tf.newaxis],
name='initial_level_prior')  # 设置先验分布

ll_ssm = tfp.sts.LocalLevelStateSpaceModel(100, initial_state_prior=initial_state_prior,
level_scale=level_scale_prior.sample())  #训练时序模型

ll_ssm_sample = np.squeeze(ll_ssm.sample().numpy())

# 整合数据

x0 = 100 * np.random.rand(100)      # 对照组 1

x1 = 90 * np.random.rand(100)       # 对照组 2

y = 1.2 * x0 + 0.9 * x1 + ll_ssm_sample      #生成真实值 y

y[70:] += 10       #设置干预点

data = pd.DataFrame({'x0': x0, 'x1': x1, 'y': y}, columns=['y', 'x0', 'x1'])
```

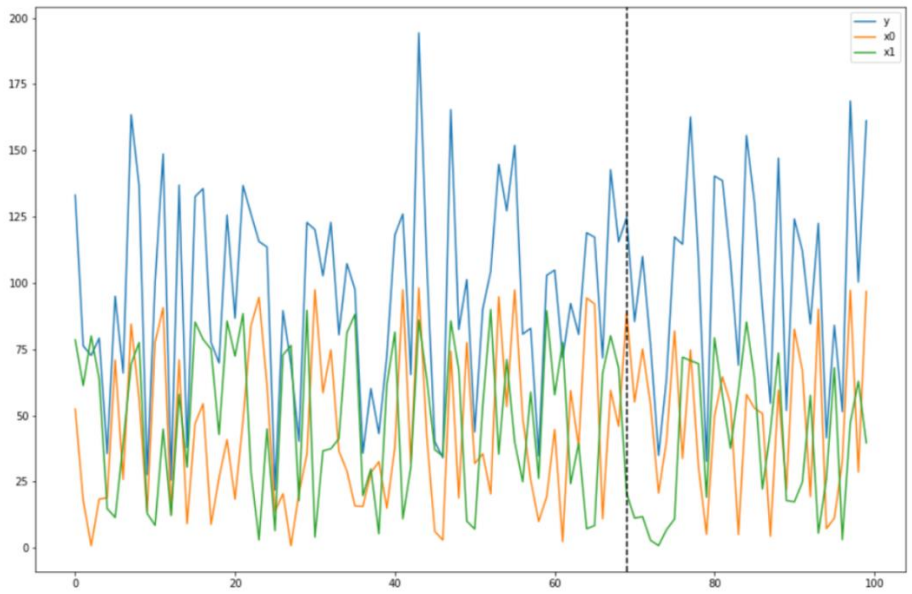


图 4-2：展示模拟数据。虚线表示干预发生的时间点，蓝线表示受到干预的观测数据；黄线与绿线表示没有受到干预的两组对照数据。

```
# 调用模型:

pre_period = [0, 69]      #设置干预前的时间窗口
```

post_period = [70, 99]　#干预后的窗口

ci = CausalImpact(data, pre_period, post_period)　#调用 CausalImpact

对于 causalImpact 的使用我们核心需要填写三个参数：观测数据 data、干预前的时间窗口、干预后的时间窗口。

输出结果：

ci.plot()

ci.summary()

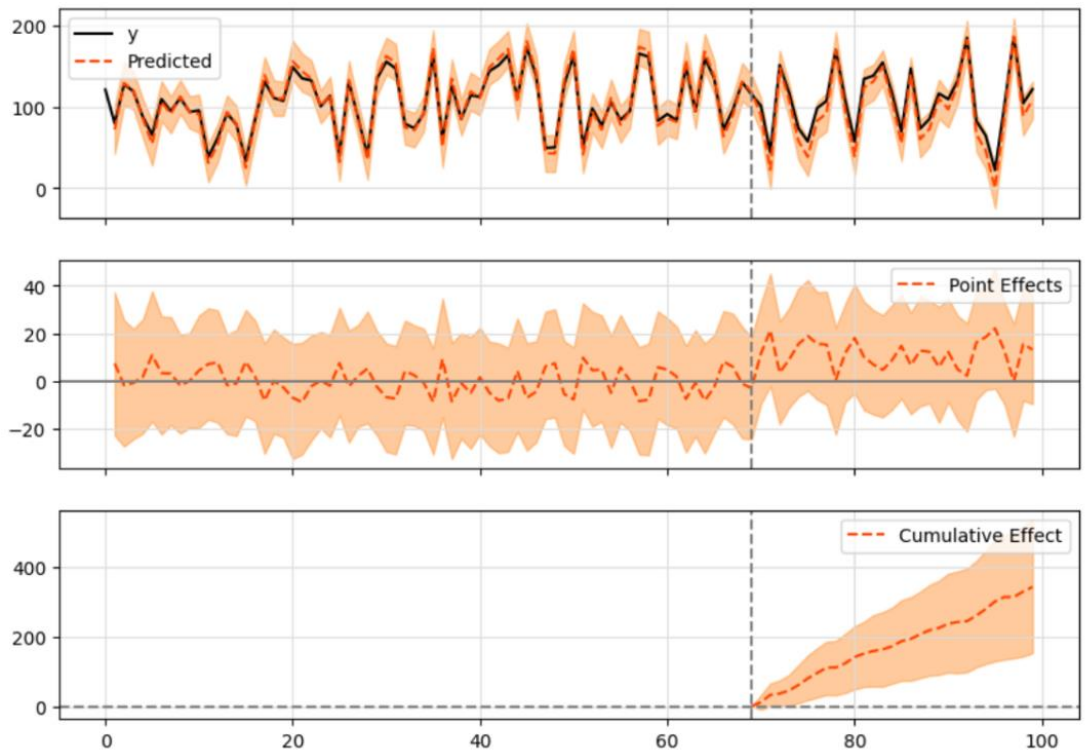


图 4-3：展示 CausalImpact 输出的结果图，图 1 表示真实值与模型拟合值的曲线；图 2 表示每个时刻真实值与预测值的差异，橙色阴影部分表示置信区间；图 3 表示真实值与预测值的累计差值。

```
Posterior Inference {Causal Impact}
                          Average            Cumulative
Actual                    107.71             3231.27
Prediction (s.d.)         96.25 (3.28)       2887.47 (98.45)
95% CI                    [89.77, 102.64]    [2693.2, 3079.13]

Absolute effect (s.d.)    11.46 (3.28)       343.8 (98.45)
95% CI                    [5.07, 17.94]      [152.14, 538.07]

Relative effect (s.d.)    11.91% (3.41%)     11.91% (3.41%)
95% CI                    [5.27%, 18.63%]    [5.27%, 18.63%]

Posterior tail-area probability p: 0.0
Posterior prob. of a causal effect: 100.0%

For more details run the command: print(impact.summary('report'))
```

表 3-1：展示 CausalImpact 输出的结果表格，量化效应值 effect 的估计及其置信区间，反映效应值是否具有显著性。107.71 表示干预之后实际值的平均；96.25 表示干预之后预测值的平均，3.28 表示估计的标准差，[89.77,102.64]表示反事实估计的置信区间。11.46 表示实际值与预测值的差距，[5.07,17.94]表示差值的置信区间，由于差距的置信区间在 0 的右侧，表示干预有显著的提升作用。

4.2 模型校验

对于模型拟合的结果，我们需要进行类似 AB 实验的“AA 校验”。一般可以通过图示的结果中的第二张图，观察干预之前真实值与预测值差值的置信区间是否包含 0，如果包含 0 则说明通过检验，模型拟合效果不错。上图中，置信区间均含 0，说明模型可用。

4.3 模型调整

- 过程参数：我们可以使用 Tensorflow 中的 Decomposition 来查看时序模型中各个结构元素，包括周期性/季节性等等。

  seasonal_decompose(data)

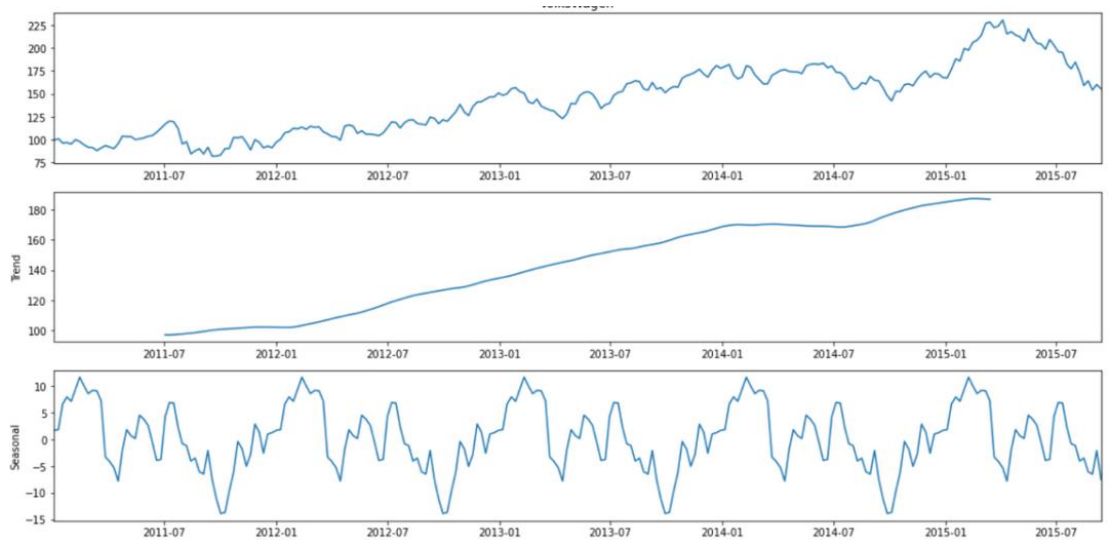


图 4-4 展示了生成数据背后的状态元素。第一张图反映原数据的走势；第二张图反映局部趋势因子；第三张图反映季节性因子。可以看出数据存在季节性结构且呈单调上升趋势。

- 自定义参数：我们可以自定义参数的先验分布；迭代次数；周期性的时间窗口长度等等。往往参数调整会对结果输出有影响，例如正确的选取先验分布会让结果更准确；迭代次数更多能保证 MCMC 收敛更稳定（但也可能导致模型运行时间较长）等等。其中最重要的是对时间窗口长度的设置，需要正确地反映观测数据的周期性。如果是年维度数据以星期为周期则设置 neasons=52；如果是天维度数据以小时为周期则设置 neasons=24 等等。

CausalImpact(..., model.args = list(niter = 20000, nseasons = 24))

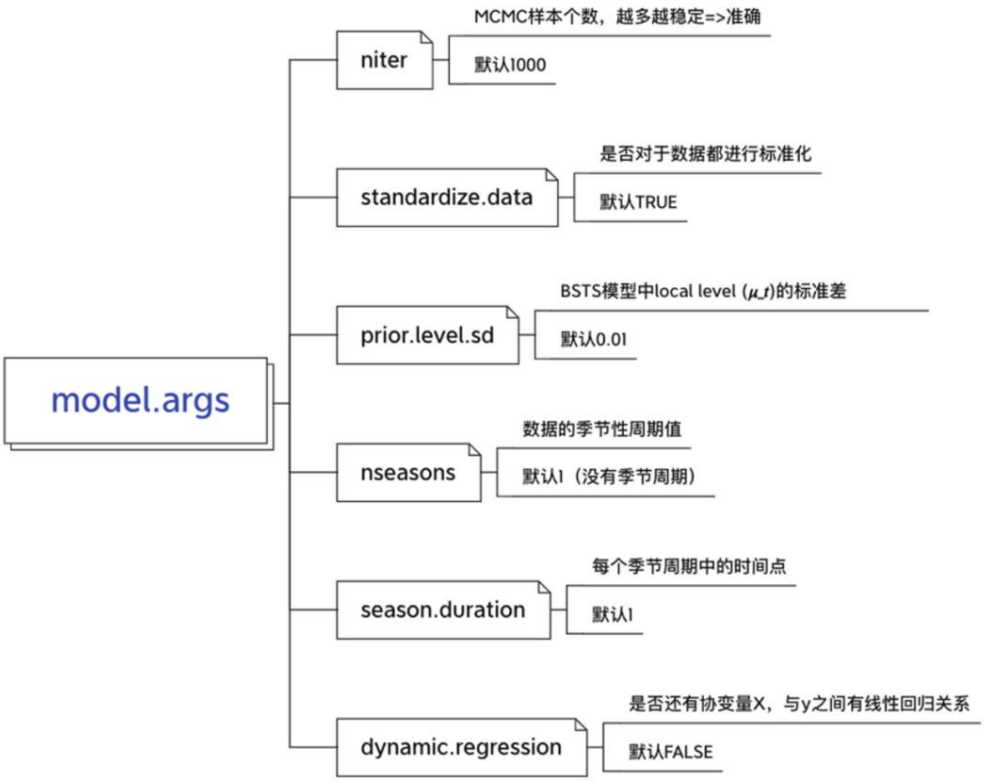


图 4-5 展示 CausalImpact 包中各个参数含义及其默认值。

- 自定义时序模型：causalImpact 的包中默认使用 BSTS 模型进行训练，我们也可以改为其他的时序模型，但前提是需要对数据进行标准化。（如果使用默认的 BSTS 则不一定需要标准化）

```
from causalimpact.misc import standardize

normed_data, _ = standardize(data.astype(np.float32)) #标准化数据

obs_data = normed_data.iloc[:70, 0]

# 使用 tfp 中的其他模型来训练时序数据

linear_level = tfp.sts.LocalLinearTrend(observed_time_series=obs_data)

linear_reg = tfp.sts.LinearRegression(design_matrix=normed_data.iloc[:, 1:].values.reshape(-1, normed_data.shape[1] -1))

model = tfp.sts.Sum([linear_level, linear_reg], observed_time_series=obs_data)

# 将自义定时序模型代入 CausalImpact 包中
```

```
ci = CausalImpact(data, pre_period, post_period, model=model)
```

五、业务场景实践

用户营销是促进留存及转化的重要方式，其中对用户进行消息触达是一大核心手段，尤其是在节假日的购票高峰期对用户进行推送，方式包括站内 push；微信生态中的小程序订阅消息；公众号或是企微环境等等，目的包括但不限于提醒用户购票；宣传品牌功能；发放优惠券吸引用户转化等等。在节假日之后，我们希望对这次的营销触达进行效果评估。

这是一个较为典型的不适合进行 AB 实验的场景，首先因为节假日是流量高峰时期，如果严格预留 50%用户不触达，可能会损失一批潜在的转化用户；如果改将对照组预留很少的人数，例如对照组：实验组=1:9，那对于后续的转化对比的科学性会产生影响。其次，节假日的推送策略往往非常精细化，总量达几十条，我们较难保证对照组用户的“纯粹性”，用户可能会被交叉触达。

基于种种问题，我们较难通过传统 AB 的手段来评估推送带来的转化效果。因此我们考虑使用因果推断的方式来解决。常规可选的方法和潜在问题如下：

- 如果使用 PSM，需要在大盘中寻找与推送人群相似但是没有被推送的用户作为对照组。但一般节假日推送时都会有兜底策略，几乎覆盖了 95%以上的平台用户，较难从中找到符合条件但未被推送的人群来进行对照。

- 如果使用 SCM，我们较难找到合适的对照组来合成。如评估度假 BU 的推送效果时，我们不太可能用火车、机票、酒店等各个产线合成一个“虚拟度假 BU”，因为本身各个产线的用户需求就不同，使用这样合成的虚拟对照组来对比度假订单的转化率是不够科学的。

- DID 的方式也同理，我们很难找到一个满足平行趋势假设且业务场景相似的对照组来进行推送前后的对比。

综上所述，一些传统的因果推断方法纵使在技术上可行，在业务的解释性上也有所欠缺。而且，以上三种方式都没有考虑到用户购票行为的“时间周期性”。因此即使合成了对照组也不一定能够匹配到实验组真正的结构特点，进而导致效应值计算有偏。于是我们考虑首先验证用户购票的数据周期性；在定位到周期规律之后尝试使用 BSTS 模型结合 CausalImpact 来进行反事实值的预测。下文我们选择 2022 年端午的火车票营销推送场景进行实践。

优先级	策略	具体策略	文案（待修改）	人群ID	预估人群最大量级
0	开售-搜索行为	30天内搜索/浏览了端午火车票的用户	标题：端午火车票明日开售！ 文案：点我立即预约，抢不到必赔￥600礼包>>		2000万
	开售-出行意向	短期内有强亲子出行意向			30万
	开售-出行意向	短期内有美食出行意向			250万
	开售-新客	有火车票新客权益的用户	标题：端午火车票明日开售！ 文案：您的新客礼包即将过期，点我立即免费试用VIP抢票权益>>		1000万

图 5-1 展示端午期间对于用户进行不同策略的推送触达

5.1 数据选取

我们以小时为周期窗口，通过简单的图像能够看出大盘的火车票下单人数确实随着时间推移呈现某种固定趋势。

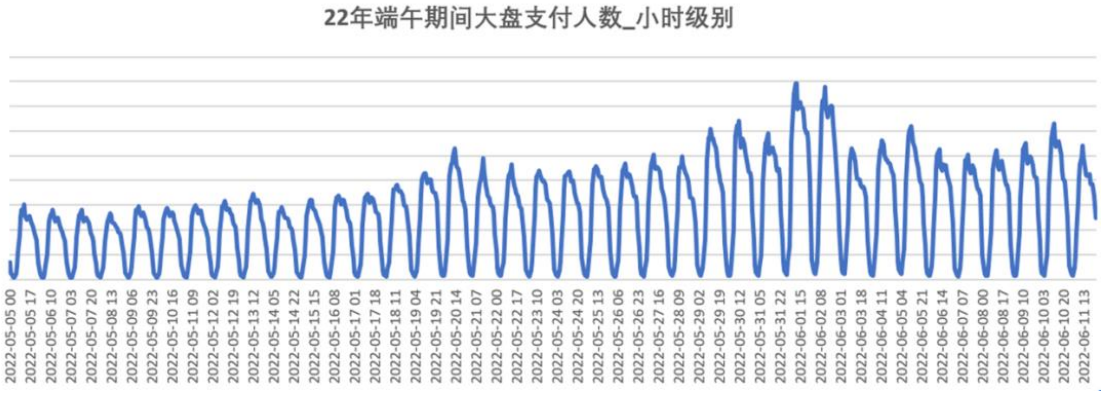


图 5-2 展示选取端午周期内（正端午前后 10 天）每小时的火车票大盘支付人数

考虑到端午作为节假日本身就有的自然流量增长，支付人数的提升不能完全归因于推送带来的，因此训练时序模型的时候，选取了 19 年-21 年所有的端午数据（正端午前后 10 天）输入 BSTS 模型进行训练，得到端午这个窗口内的特有的结构状态，随后用这个结构化的模型来代入 22 年的端午数据，对 2022 年端午推送之后的转化人数做出预测。

最后使用真实的转化人数与预测人数作差体现本次营销推送的效果。

5.2 R-代码实现

```
# 选取 19-22 年每年的端午窗口，按照小时划分，共 960 个数据点
```

```
y_hour=c(x1,x2,x3,x4)

x_time_hour=c(1:960)

data_hour <- cbind(y_hour, x_time_hour)

pre.period <- c(1, 808)     # 2022 年的推送发生在第 808 个时间点，故以此为干预节点。

post.period <- c(809, 960)

# nseasons=24，迭代次数 2000，fit the model

impact_hour <- CausalImpact(data_hour, pre.period, post.period, model.args = list(niter =
20000, nseasons = 24))

summary(impact_hour)

plot(impact_hour)
```

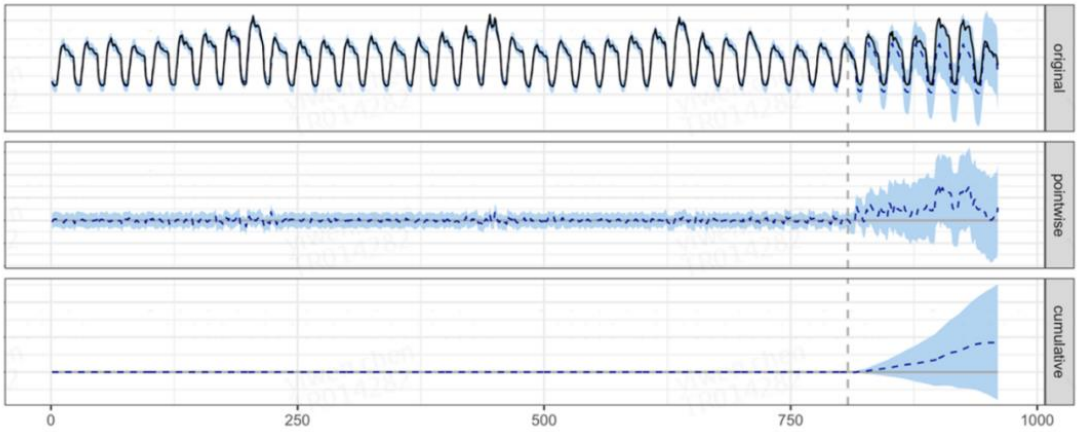


图 5-3 展示使用 CausalImpact 返回的结果图。第一张图表示真实支付人数与预测支付人数；第二张图表示真实值与预测值的差值及置信区间；第三张图是累计差值和置信区间。

图像显示模型能够通过 AA 校验，模型有效。在干预点之后，实际值较预测值有所提升，但提升的置信区间含 0，因此未达到显著程度。体现出 2022 年端午营销策略对于转化人数有所促进作用，但是效果未达显著。

六、方法优缺点

相较于传统因果推断方法，BSTS 模型有 2 个主要优点：

- 能够识别出数据背后的结构化特征，更好的做出预测；

- 利用了贝叶斯估计的思想，得到参数的后验分布情况，计算效应值时能够给出置信区间。但第（2）点对于 BSTS 模型是一把“双刃剑”，如果先验分布设置得不好，会影响 MCMC 的收敛速度和方向甚至最终的后验分布情况。因此对于先验分布的选取需谨慎。

七、方法拓展

本文介绍的结构化时序模型将数据的周期特点拆分成了趋势项、季节项、周期项等等，每种元素挨个探究。更进一步，我们可以将时间序列按照周期性的长短来进行拆分，分为长周期项（使用大滑动窗口）、短周期项（使用小滑动窗口）、季节项等等。这样的好处是防止一些小窗口内的周期情况被长周期的信息平滑掉，能够更好的体现出数据在不同程度上的周期特点。具体的方程可以拆分成如下形式：

$$
\begin{aligned}
Z_{n,t}^{(3)} &= \sigma_{n,t}^{ce} Z_{n,t}^{(4)} + \mu_{n,t}^{ce} \\
Z_{n,t}^{(2)} &= \sigma_{n,t}^{st} Z_{n,t}^{(3)} + \mu_{n,t}^{st} \\
Z_{n,t}^{(1)} &= \sigma_{n,t}^{se} Z_{n,t}^{(2)} + \mu_{n,t}^{se} \\
Z_{n,t}^{(0)} &= \sigma_{n,t}^{lt} Z_{n,t}^{(1)} + \mu_{n,t}^{lt}
\end{aligned}
$$

其中$Z_{n,t}$表示不同时间点的状态值；4 个模块分别代表长周期项/短周期项/季节项/序列相关性项（带有协变量 X）；每个结构模块都有一个均值和一个标准差。

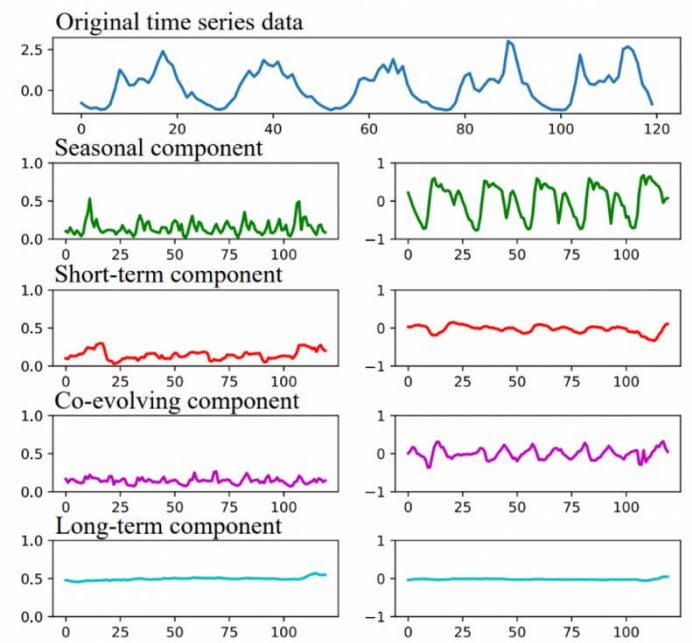


图 7-1 展示了某时间序列背后的 4 个模块：从上至下以此表示：原数据情况；季节性因子；短周期项；相关性项；长周期项。短周期来看数据的波动比较明显；长周期来看数据波动较不明显，因此这里需要考虑到短周期内的数据结构，避免被长周期的数据平滑。

接下来对以上 4 种模块分别进行预测。针对长周期和季节性，由于他们在短时间内的变化不大，因此可以直接使用对应方程中的 μ 和 σ 来进行预测；针对短周期项和相关性项可以通过其他机器学习方式进行预测。得到各个模块的预测结果之后，结合各模块特征进行融合，得到整体的预测结果。参考文献[4]中给出了更具体的预测方式和与传统方式的对比结果。

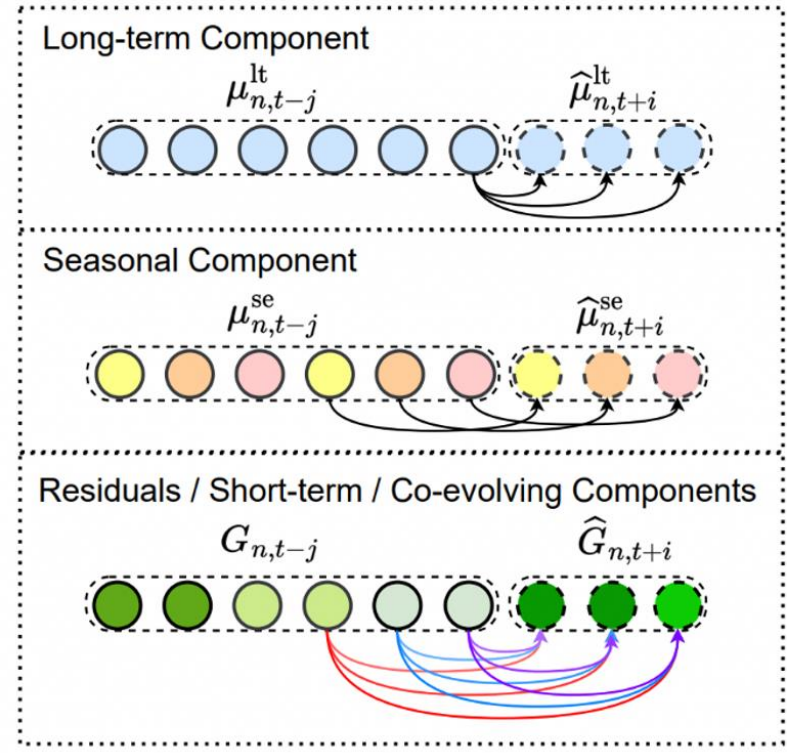


图 7-2 展示针对长短周期不同的预测方式：长周期项与季节项可以直接用 μ 表示预测；短周期及协变量相关项使用自定义的机器学习模型进行预测。

依照上述方法得到的时序预测模型后，我们再将其代入 CausalImpact 的代码中，调整参数 model 为自定义的时序模型即可。

八、总结

本文介绍了用因果推断的方式评估某一政策作用于时间序列数据带来的效应，使用了 BSTS 的状态空间模型来进行反事实值的预测，并通过 CausalImpact 的代码进行实现。

不同于其他因果推断方法的框架，本文中的方法对所有超参数进行贝叶斯估计，再对后验分布进行 MC 采样得到反事实的预测值，主要优势是能够根据最大程度考虑到所有对照组

以及实验组自身的结构特点，给出反事实预测值的估计以及置信区间，衡量效应值的显著性。

同时，本文介绍的方法主要聚焦于结构化时序数据，利用 BSTS 模型识别观测数据背后的状态值以及各个状态之间的转化情况，进而在进行反事实预测时，尽可能消除由隐藏状态带来的影响。

需要注意的是，使用 BSTS 模型之前，需要验证数据是否真的有周期性特点以及结构元素是怎样的（是否是长短周期等等），再挑选合适的时序模型来训练；同时对于参数的先验分布设置也需谨慎，尽可能使得最终的效应估计值科学稳定。

人工智能

携程商旅基于图网络的注册风控实践

【作者简介】Jue，携程数据分析师，专注风控、用户流量分析等领域。

一、背景

注册风控是企业风控场景中重要的一环，黑产用户想要进行各式各样的欺诈行为，首先需要注册账户。通常，这些黑产用户会采用技术手段批量注册虚假账号，而后进行一系列恶意行为。那么，如何准确快速地识别注册用户是否是异常聚集的黑产用户，这是注册风控场景中需要解决的一个问题。

因此，本文介绍了一种当前在携程商旅应用的基于注册图网络的恶意账户聚集检测模型。

二、前置知识[1]

2.1 图的定义

图是由顶点的集合和顶点之间的边的集合组成，通常表示为：G(V,E)，其中，G 表示一个图，V 是图 G 中顶点的集合，E 是图 G 中边的集合。

2.2 图的类型

1）有向图/无向图 （Undirected graphs / Directed graphs）

若顶点v_i到v_j之间的边没有方向，则称这条边为无向边，用无序对(v_i, v_j)来表示。如果图中任意两个顶点之间的边都是无向边，则称该图为无向图；

若顶点v_i到v_j之间的边有方向，则称这条边为有向边，用有序对$< v_i, v_j >$来表示。如果图中任意两个顶点之间的边都是有向边，则称该图为有向图。

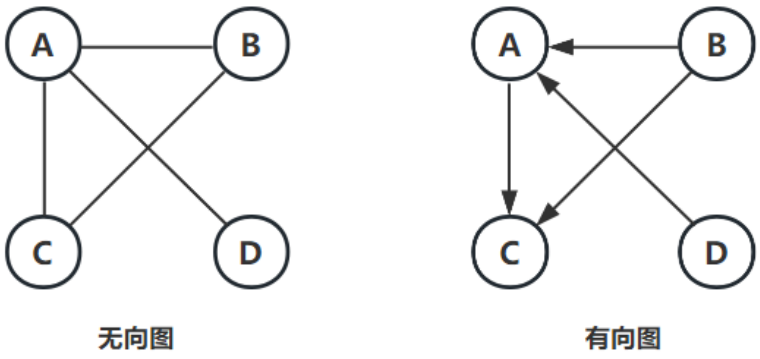


图 1：无向图和有向图

2）同构图/异构图（Homogeneous Graph / Heterogeneous Graph）

同构图：节点的类型和边的类型只有一种

如社交网络中可以考虑节点只有“人”这一个类型，边只有“认识”这一种连接关系，就是一种同构图；

异构图：节点的类型+边的类型 > 2

如论文引用网络中，存在着作者节点和论文节点，边的关系有作者-作者之间的共同创作关系连边、作者-论文之间的从属关系、论文-论文之间的引用关系。

三、数据挖掘

3.1 用户设备维度

1）设备聚集性

我们对用户所使用的设备进行了分析，这里的设备指的是一种广义的设备，包括 IP 地址、手机号、设备号等。黑产用户处于金钱和时间成本，会希望在尽量低的成本下完成获利，那么在设备维度就会出现聚集的现象。一般来说，普通用户的一台设备上通常不会登录多个账户，因此某些设备频繁切换不同账号是相当可疑的。

如下图所示，当 IP 地址数从 0-5 增加到 5-10 时，用户数占比骤降，60%的用户只使用过 0-5 个 IP 地址；超过 90%的用户只使用过一个手机号；超过 70%的用户只使用 1 个或 2 个设备。

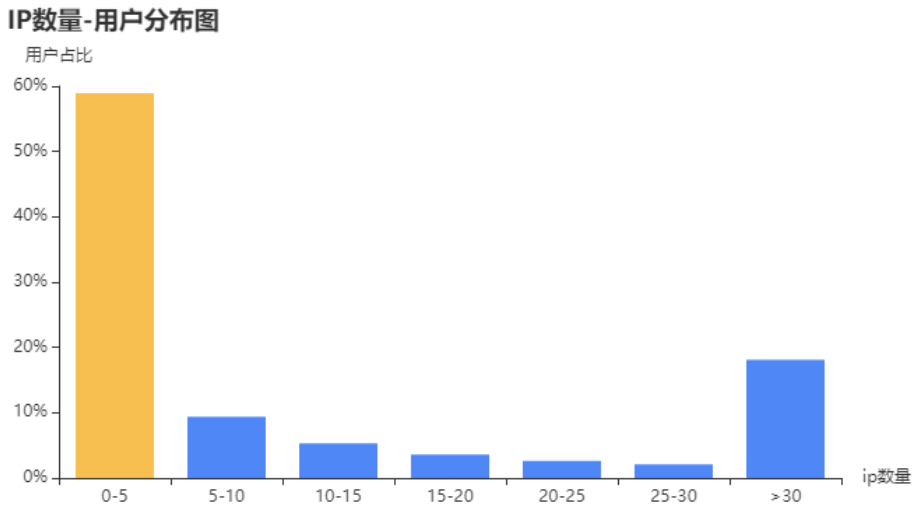


图 2：IP 数量 – 用户分布图

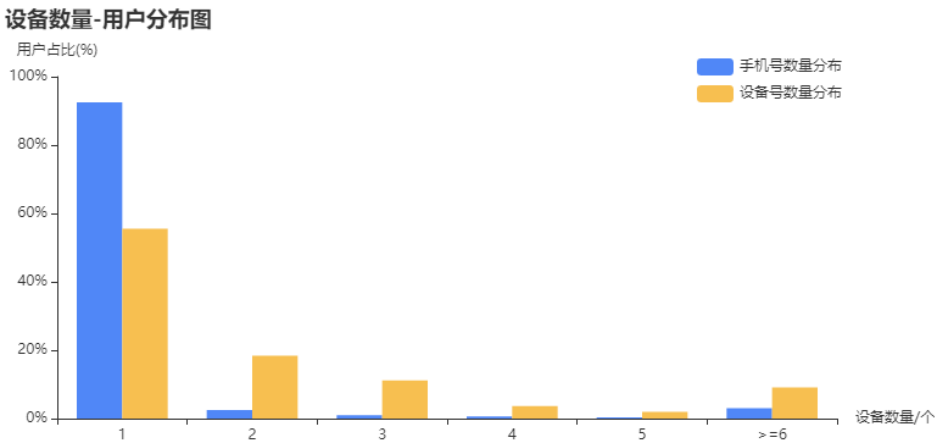


图 3：手机号/设备号数量－用户分布图

2）地理位置和设备版本

IP 地址和电话号码都可以映射到一个大概的地理位置。我们可以判断每个注册用户的两个地理位置是否相同，不同的时候为黑产用户的概率相对更大。

此外，我们也对设备版本进行了分析，发现大多数从过时操作系统版本注册的账户都是假账户，Android 和 iOS 都存在这种现象，原因可能是攻击者使用脚本自动注册假账户，而操作系统版本并未更新。

3.2 注册特征维度

1）注册时间

我们对正常用户和黑产用户的注册时间进行了对比，发现黑产用户更倾向于傍晚和夜间进行注册（17 点-次日 6 点）。

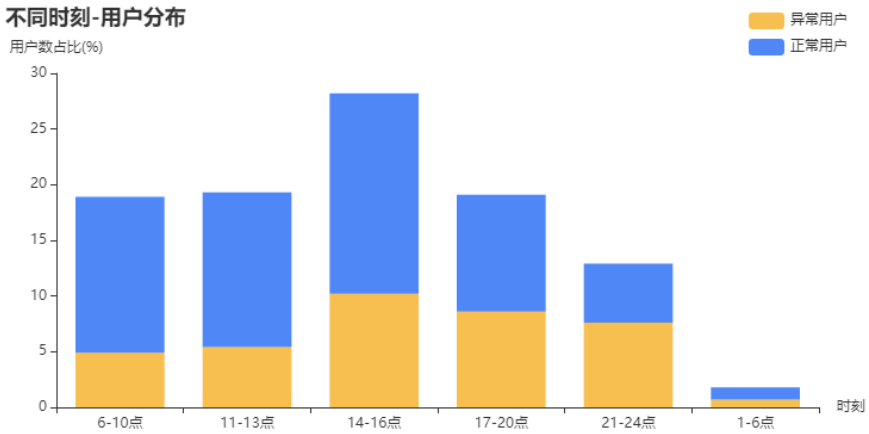


图 4：不同注册时间的用户分布

2）注册填写信息

基于商旅的背景下，用户在进行注册时需要填写公司名称、邮箱等信息，我们会判断用户所

填写的信息是否是真实的企业信息（如填写的公司名是否是正常的公司名、邮箱是否正常、是否是随机填写等），这些对注册填写信息相关的解析也会作为模型的特征。

四、模型设计

模型主要包括了四个步骤：特征提取、无监督权重学习、构建用户注册图和恶意账户监测[2]。

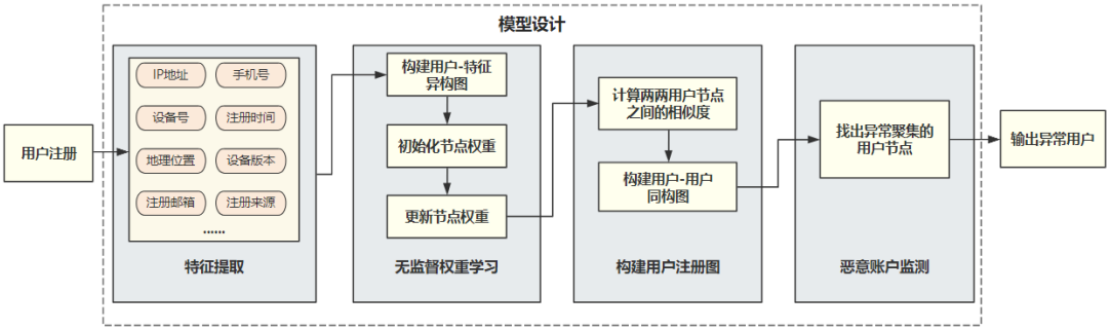


图 5：模型流程图

4.1 特征提取

根据上述数据挖掘部分提取用户的相关特征，对于连续的特征进行分箱处理使其离散化。

4.2 无监督权重学习

1）构建用户-特征异构图

构建用户-特征的无向图来捕获注册账户和提取特征之间的关系，用 G(V,E) 表示，其中 V 是所有的节点集合，包括用户节点 R 和特征节点 F，即每个 F 节点都表示某个特征；E 表示边。用户节点 R 与特征节点 F 存在边等价于该用户节点 R 拥有此 F 特征。

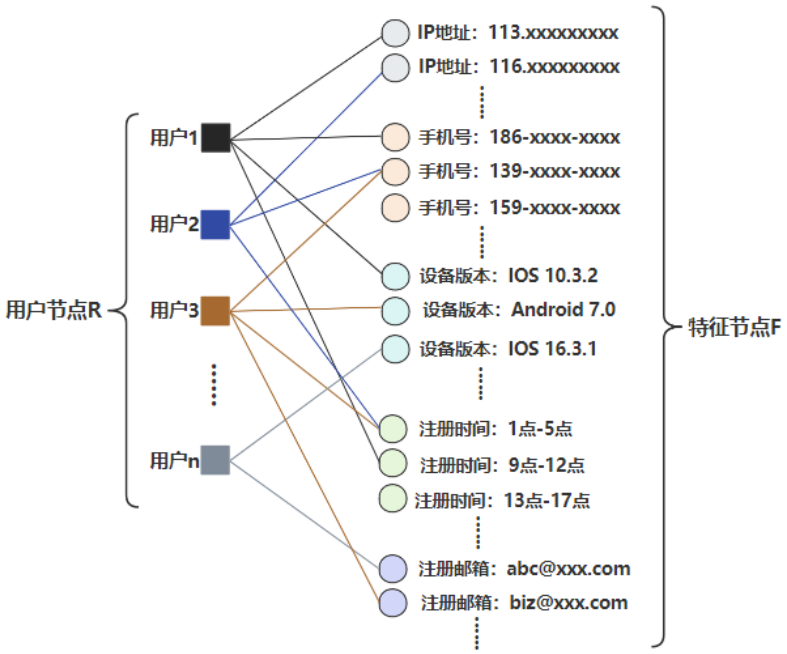


图 6：用户－特征异构图

2）用户节点和特征节点初始权重的计算

- 把特征分成 A、B 两类

A 类特征：除去 B 类的其他特征。

B 类特征：该特征如果被大量账户共享，则认为是异常的，如 ip 地址、手机号、设备号等。

- 计算特征节点初始权重

对于任意的特征节点 x，初始权重 Wx 计算如下：

$$w_x = \frac{1}{2}\left(dev(x) + base(x)\right)$$

其中：

$$dev(x) = \begin{cases} 1 - \dfrac{ratio(x)}{ratio\left(mode\left(pre(x)\right)\right)} & if\ pre(x) \in Pre\_A \\ \dfrac{ratio(x)}{ratio\left(mode\left(pre(x)\right)\right)} & if\ pre(x) \in Pre\_B \end{cases}$$

$$base(x) = \begin{cases} 1 - ratio\left(mode\left(pre(x)\right)\right) & if\ pre(x) \in Pre_A \\ ratio\left(mode\left(pre(x)\right)\right) & if\ pre(x) \in Pre_B \end{cases} \quad (4.1)$$

其中 Pre_A 和 Pre_B 分别表示第一步中的 A 类特征和 B 类特征，ratio(x)表示该特征节点值的在其所属特征中的频率，pre(x)表示该特征节点所属的特征，如特征节点 IP-113.xxxxxxxx 属于 IP 特征；mode(pre(x))表示该特征节点所属特征中频率最高的特征节点。

- 计算用户节点初始权重

按照上一步计算好每一个特征节点的初始特征后，每个用户节点的初始特征即为其所有邻居特征节点的均值。

3）更新节点权重

对所有节点的权重通过线性信念传播进行迭代更新，权重的迭代公式参考文献[2]和[3]的思想并做了改进优化。优化思路是基于每一次权重的迭代是对上一次节点权重的结果加上各邻居节点权重的平均影响，并且每次的权重在仍保持在 0-1 之间，由此得到如下的迭代公式，对比验证了优化后的公式在后续构造用户-用户同构图的过程中边关系更具稳定性：

$$p_u^{(0)} = w_u,$$
$$p_u^{(t)} = \frac{1}{2}\left(p_u^{(t-1)} + \frac{1}{|\Gamma(u)|}\sum_{v\in\Gamma(u)}\left(p_v^{(t-1)} - \frac{1}{2}\right) + \frac{1}{2}\right) \qquad (4.2)$$

其中$p_u^{(0)}$为节点 u 的初始权重，$p_u^{(t)}$表示节点 u 第 t 次迭代的权重，Γ(u)表示 u 的所有邻居节点的集合，|Γ(u)|表示 u 的邻居节点数量。

4.3 构建用户注册同构图

1）算两两用户节点之间的相似度

我们的分析主体在注册用户上，而上述构建出来的图是一个异构的二部图，转化为同构图的步骤为计算节点之间的相似度，并且将此相似度作为注册用户之间的边的权重，即

$$sim(u,v) = \sum_{s\epsilon\Gamma(u)\cap\Gamma(v)} p_s \qquad (4.3)$$

2）构建用户-用户同构图

两个用户节点相似度为公共的特征节点的权重之和，即若上述相似度大于设定的阈值（相似度阈值），则两个用户节点连上边。这时若只考虑用户节点和用户节点之间的边，即成为了一个同构的用户图。

相似度阈值的设置方式

当阈值设置为 0 时，所有的用户都互相连接，随着阈值的不断增大，用户和用户的边逐渐减少，当阈值足够大时，不存在任何边。通过对历史数据的分析，我们发现能够使得异常用户相连接的相似度阈值会存在一个小范围的稳定区间，因此阈值设置的逻辑为使得用户-用户同构图的边关系稳定（小区间内保持边关系不改变）的最小阈值。

我们的实际设置方式为：先将阈值设置成一个较小的值α，此时几乎所有的用户都互相连接，再将阈值每次增加δ，观察最大连通子图的节点数量，若该数量在连续 n 次的阈值增加的过程中几乎保持不变，则第 n 次的阈值 α+nδ 设置为当前模型的相似度阈值。

4.4 恶意账户监测—找出异常聚集的连通子图

找出所有的连通子图，当连通子图用户节点大于某个数值（聚集用户数阈值）时，则该连通子图的所有用户为异常聚集。图 8 展示了异常聚集用户和正常用户的边关系状态。

聚集用户数阈值的设置方式

聚集用户数的阈值是根据模型的效果来确定的，可以通过不同值的模型效果来确定最佳值，具体可以根据准确率、召回率、F1 值等的收敛情况来进行衡量确定。通常情况下注册风控选择的聚集用户数阈值建议在 10-30 区间为佳。

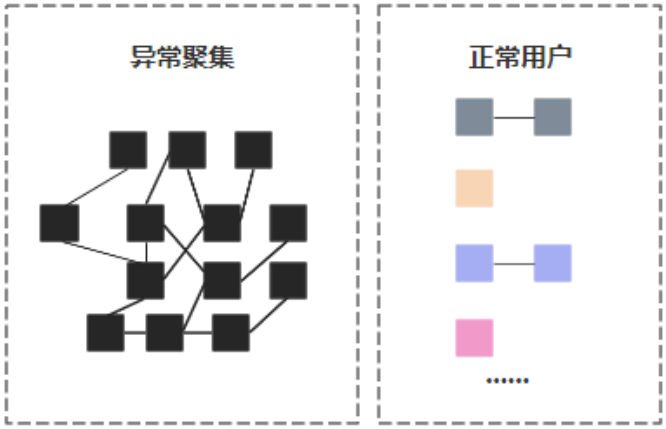


图 7：用户－用户同构图

五、实时策略

对于每天注册进来的新用户，如何高效迅速地识别注册用户是否是异常聚集的黑产用户呢？这里介绍商旅注册风控场景下的实时策略。

首先，每注册一个新用户会触发模型的更新（模型的数据取近 24~72 小时的注册用户），同时每 n 分钟更新一次模型参数作为兜底。模型需要保存的参数一般有：坏用户节点、坏用户的所有特征节点及其权重、相似度阈值。

如图 9 所示，对于一个新注册用户，获取到其特征后，计算该用户与当前最新模型中的坏用户节点的相似度（即与坏用户特征节点的公共节点权重之和），若与任意一个坏用户节点的

相似度大于模型的相似度阈值，则输出为异常聚集，否则认为是正常用户。

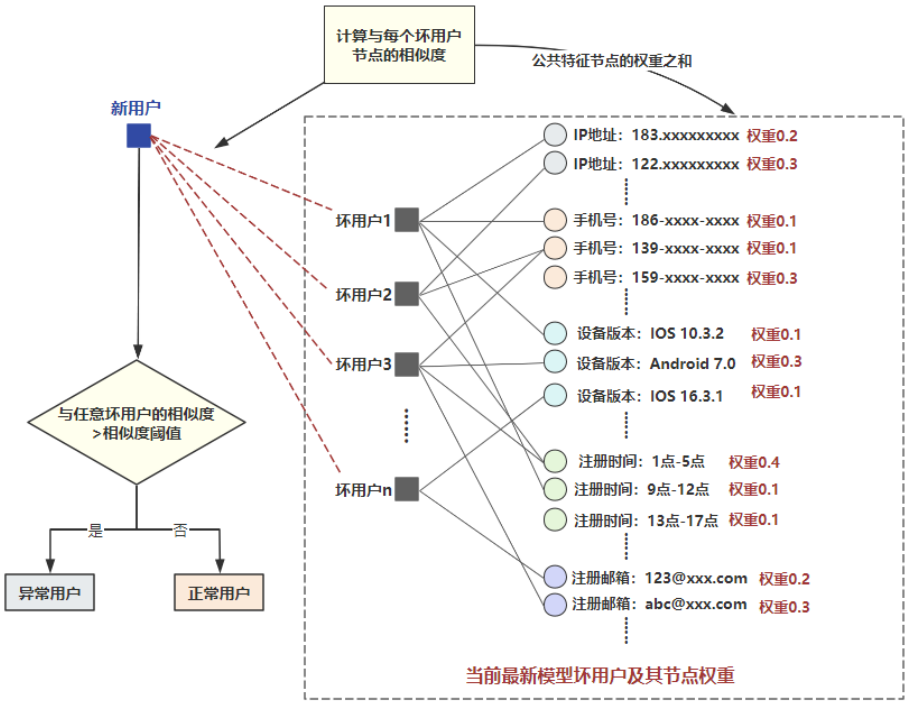


图 8：用户注册实时策略图

六、实际应用效果

6.1 数据模型指标

历史模型以天为单位，下图展示部分日期模型的准确率和召回率。

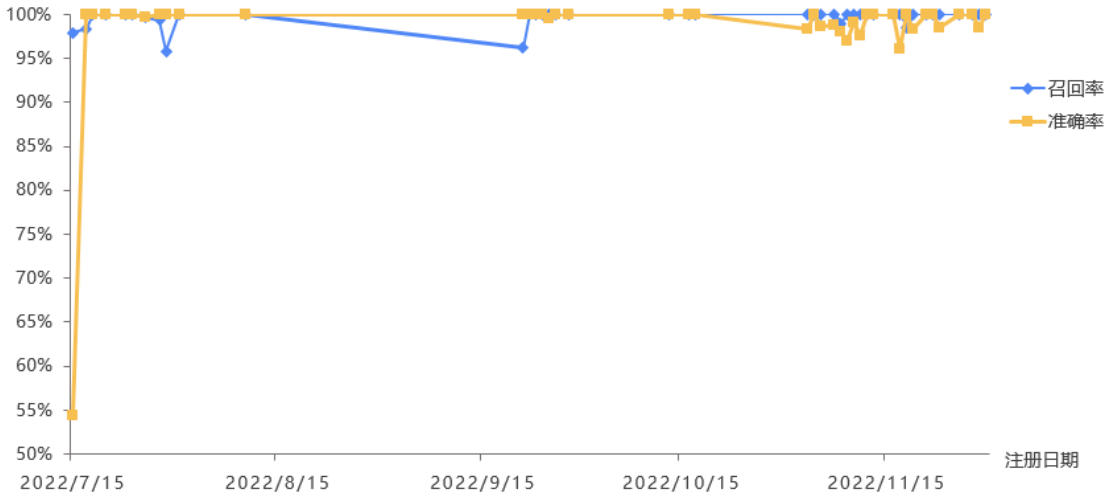


图 9：历史数据模型效果

6.2 线上业务指标

1）注册拦截率

用户在注册时若模型识别为异常，则需要用户进行身份验证（姓名和证件号），若用户验证失败或放弃注册则认为注册拦截，下表展示了 2023 年以来的月注册拦截率（未完成注册的用户数/注册时识别有风险需要验证的用户）的数据。

注册月份	月注册拦截率
1 月	100.0%
2 月	83.7%
3 月	78.1%
4 月	93.2%
5 月	99.9%

2）黑产用户注册时的风险识别率

商旅对于各个产线都有不同的风控场景识别，这些识别出来的黑产用户有些在注册时已被本文的模型识别为异常，但后续通过了身份验证也会成功注册。因此考虑黑产用户在注册时的风险识别率也是衡量模型效果的一个指标。下表展示了部分风控场景中的注册时风险识别率（注册时识别异常的黑产用户数/黑产用户数）的数据。

风控场景	注册时风险识别率
跟团游场景	99.8%
机票场景	91.7%
酒店场景 1	74.3%
酒店场景 2	63.3%

七、不足和思考

7.1 节点初始权重的计算

每更新一次模型，初始权重都会基于公式(4.1)重新进行计算，即当前模型节点的初始权重只和当前模型有关（每次模型的时间选取为近 24 小时），没有去学习历史的权重结果。实际上，对于历史模型认为异常的用户，其特征节点在当前模型中出现时被赋予更高的初始权重可能是更优的方案。

7.2 邻居影响力

节点的最终权重是根据公式(4.2)进行迭代更新的，从公式可以看出，对于每个节点，每一次权重的迭代是对上一次节点权重的结果加上各邻居节点权重的平均影响力得到的。若能根据

节点连接情况或历史数据，对不同邻居的影响力进行加权平均可能是更优的方案。

八、参考文献

- Zhou J, Cui G, Hu S, et al. Graph neural networks: A review of methods and applications[J]. AI Open, 2020, 1:57-81.

- Liang X, Yang Z, Wang B, et al. Unveiling Fake Accounts at the Time of Registration: An Unsupervised Approach[C]// KDD21: The 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining. ACM, 2021.

- Wang B, Jia J, Zhang L, et al. Structure-based Sybil Detection in Social Networks via Local Rule-based Propagation[J]. IEEE Transactions on Network Science & Engineering, 2018, PP(99):1-1.

携程火车票异常检测和根因定位实践

【作者简介】龙川泾渭，携程算法工程师，专注异常检测、根因分析、时间序列预测等领域。

摘要

携程火车票包含 1000+的业务指标，人工监测指标的异常情况耗时费力，而由于业务差异，基于规则和简单统计学的检测方案只能覆盖到单个指标或者单类指标，并且不能随着新业务上线或者功能变动灵活动态的调整相应的规则，并不适用于大量不同业务线的指标。我们希望使用 AI 算法来代替人工，对指标进行全自动的监控，旨在发现指标的异常和导致异常的潜在原因。

具体来说，对于异常检测，使用六种无监督检测算法计算异常得分，根据时间序列特性和指标的业务特点计算异常阈值，集成多种算法的异常结果进行硬投票，得到异常结果。对于根因定位，集成了 Adtributor、Hotspot 等四个算法做硬投票系统，按投票次数降序输出根因结果。此外根据指标的重要程度，设置不同的投票规则，来权衡精召率。

一、背景

最近几年，火车票业务持续高速成长，业务指标种类繁多，如何快速准确的发现指标的异常以及导致异常的原因，并及时排除问题显得尤为重要。基于 AI 的异常检出能力，可以根据指标的历史数据分布规律，实现实时监测，帮助开发人员尽早的发现问题和挖掘产生问题的原因。

1.1 业务指标特点

我们从已有的历史监控指标中，发掘时序数据的变化规律，从而根据数据的分布特点选取合适的算法。从图 1 我们可以看出，火车票业务的核心业务数据，主要呈现两种规律：

1）周期型时间序列，受人们出行规律的影响，大部分核心业务指标都会呈现较强的周期型规律。由于多数用户在工作日买票，周末出行，就形成工作日的支付票量和出票票量比周末多的规律，而出行票量则表现出刚好相反的趋势。在不同业务线下，相同的业务指标之间的周期规律也不尽相同。

2）平稳型时间序列，对于任意时刻，如果它的性质不随观测时间的变化而变化，那就认为是平稳型时间序列。由于指标本身的性质和偶发性的因素影响，有些指标没有很强的规律性，而展现出相对稳定的趋势。

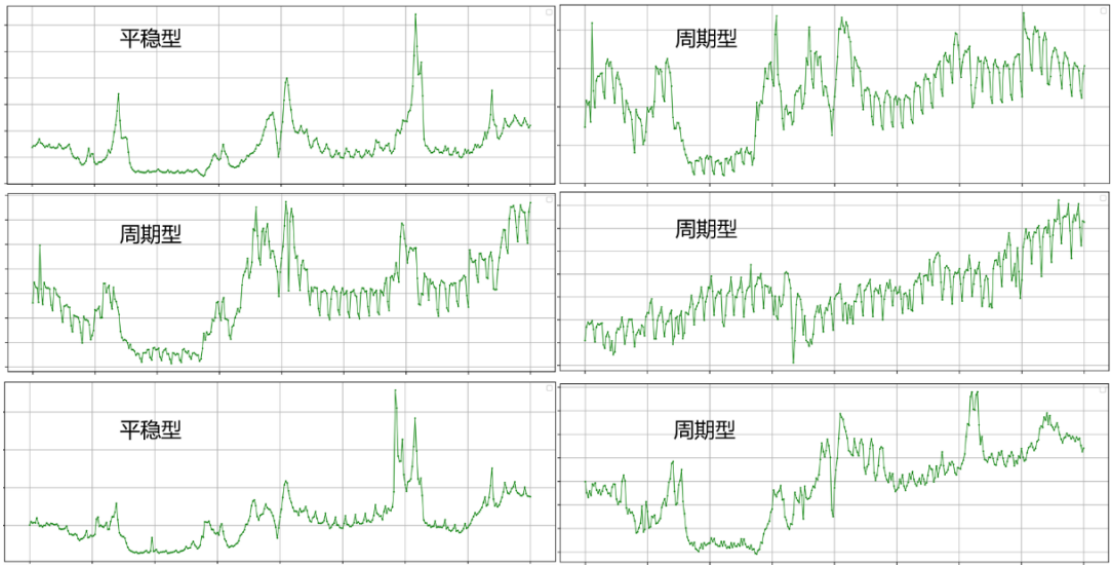


图 1 核心业务指标的时间序列类型

不论是平稳型还是周期型指标，根据当前的业务情况上线新的营销策略，发放优惠券等活动，尤其是对出行的影响巨大的疫情及其相关的政策，均会引起指标一定程度的剧烈波动，如陡升，陡降，整体抬升或降低。

1.2 主要痛点

1）基于规则的异常检测识别适用性差。火车票业务指标目前有 1000+，人工对所有指标进行检测耗时费力。而基于业务规则的检测方案只能覆盖到单个指标或者单类指标，并且业务规则的配置需要对业务特点熟悉，对于新业务上线或者功能变动则不能灵活动态的调整相应的规则，并不适用于大量不同业务线的指标。

2）大部分核心指标都是和出行相关的，而近三年由于疫情的影响，在特定的时间窗口内，用户的出行规律变化复杂，导致精准捕捉指标的周期变得困难，从而增加了异常点的检出难度。

3）异常故障需要数据分析师排查。当某个业务指标发生异常，需要快速准确的定位到是哪个交叉维度的细粒度指标的异常导致的，以便尽快做进一步的修复止损操作。由于指标维度多，每个维度取值范围大，导致人工介入解决问题的难度和时间成本高。

二、异动归因系统介绍

针对业务需求、指标的特点和当前存在的痛点等问题，我们开发了一套异动归因系统，它分为异常检测和根因定位两个子系统。异常检测系统负责将各业务指标的突然上涨和下跌等异常情况检测出来，根因定位系统是在指标出现异常的情况下，找出引起异常的原因。

2.1 异常检测系统

由于不同业务线指标数量多，规律差异大，无监督算法更符合火车票业务需求。常见的无监

督异常检测算法如表 1 所示【1】，对于局部异常点， LOF（局部异常因子）在统计上优于其他无监督方法，而使用 K 次（全局）最近邻距离作为异常评分的 KNN 是统计上最好的全局异常检测算法。

综合考虑局部异常和全局异常点的检出能力，以及训练时长，最终确定了 LOF，KNN，CBLOF，COF，IForest，PCA 作为火车票的异常检测算法。

算法	局部异常检测能力	全局异常检测能力	训练时长
LOF	☆☆☆☆☆	☆☆	☆
KNN	☆☆☆☆	☆☆☆☆☆	☆
CBLOF	☆☆☆☆	☆☆☆☆	☆
COF	☆☆☆☆	☆☆	☆
IForest	☆☆☆	☆☆☆☆	☆
OCSVM	☆☆	☆☆☆	☆☆☆☆
PCA	☆☆	☆☆☆☆	☆
HBOS	☆☆	☆☆☆☆	☆☆☆☆
ECOD	☆☆	☆☆☆	☆
COPOD	☆☆	☆☆☆	☆
SOD	☆☆	☆☆☆☆	☆☆☆☆
LODA	☆☆	☆☆☆	☆
DAGMM	☆	☆☆	☆☆☆☆

表 1 常见异常检测算法性能分析

无监督算法的性能在很大程度上取决于其假设和潜在异常类型的一致性【1】，所以对不同指标经常出现的异常类型的分析至关重要，以便于选择合适的检测算法。我们对典型的一些周期型指标序列和平稳型指标序列的异常情况进行分析统计，如图 2 所示：

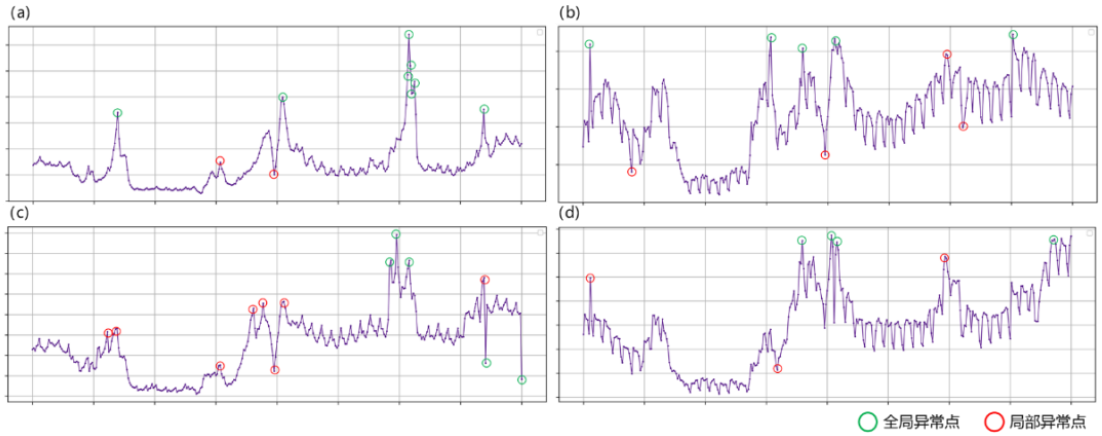


图 2 核心业务指标的异常点类型

分析发现，平稳型指标序列（a）出现的主要是全局异常点，而局部异常点主要分布在具有趋势性和周期性的序列（b）、（c）和（d）中。由于相同业务线的指标趋势规律大致相同，出现的异常类型也大致一样，我们可以根据不同业务线的指标序列类型，应用不同的算法组合。

在分析指标的异常类型和算法选取之后，我们需要根据指标的不同分布情况，针对每种算法的异常得分选取合适的阈值，得到最终异常结果。我们分析了部分核心指标的概率密度分布，对不同数据的分布采用了不同的阈值算法，最终确定在低偏态高对称分布下，Z-score+肘部法则的方法计算阈值，在高偏态分布下，使用箱型图（Boxplot）计算阈值。

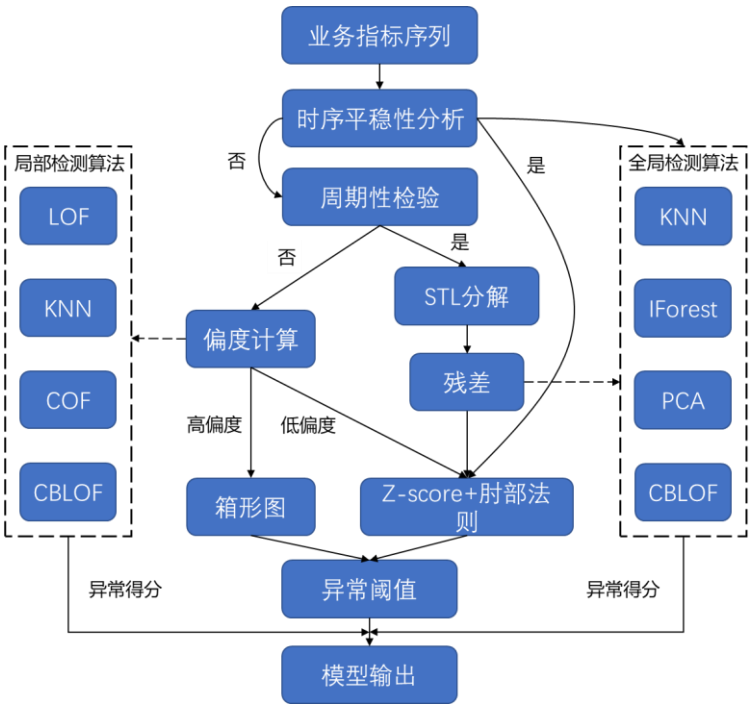


图 3 异常检测系统流程图

经过如上分析之后，我们的异常检测分析流程如图 3 所示，主要分为时间序列分析、异常得分计算和异常阈值计算三个模块，下面我们分别介绍：

1）时间序列分析。当时序数据满足平稳型检验，直接使用全局检测算法进行序列的异常得分计算，同时使用 Z-score+肘部法则的方法，计算异常得分阈值。当时序数据满足周期性检验时，使用 STL 算法提取时序数据的周期性分量和趋势性分量，将残差分量进行异常得分和异常得分阈值计算。当时序不满足周期性检验时，将时序数据转化为概率的分布图，如果偏度低于阈值，使用 Z-score+肘部法则的方法计算异常得分阈值，否则使用箱型图计算。

2）异常得分计算。根据对异常类型的检出能力，分为全局异常检测算法组：{KNN, IForest, PCA, CBLOF}和局部异常检测算法组：{LOF, KNN, COF, CBLOF}。

3）异常阈值计算。箱形图是通过计算一组值的最大值 Q4，上四分位数 Q3，中位数 Q2，下四分位数 Q1，最小值 Q0，来描述样本分布的离散程度以及对称性。一般情况下，箱形图不需要对样本进行任何假定，在对样本具有较高的异常容忍度的同时，能够描述样本的离群程度。

Z-score 是描述一个值和一组值的平均值的偏离程度的统计测量分数，定义如下：$z=((x-\mu))/\delta$。肘部法则定义每个类的畸变程度等于每个变量点到其类别中心的位置距离平方和，随着类别数量的增加，平均畸变程度的改善效果会降低，而改善效果降幅最大的位置对应的

值就是肘部，该方法一般用于聚类数量的选择。

这里我们结合 Z-score 和肘部法则，计算待检测时间序列的|z|序列,然后对其进行降序排列得到|z|desc 序列，最后利用肘部法则得到肘部值作为异常阈值。

最后根据每种算法的异常得分和异常阈值进行分析，以投票最多的作为最后的异常结果。针对指标的重要程度设置不同的投票规则权衡精召率，比如更重要的 P0 指标，降低投票阈值保证异常不漏报，而对于不太重要的 P2 指标，适当提高阈值保证精确率。

2.2 根因定位系统

在我们的各类火车票核心业务指标中，都是多维度的可加性指标集合，当总指标被检测出异常之后，需要尽快定位是哪些交叉维度的细粒度指标导致了总指标的异常。

例如对于某个业务线的订单量指标 A，它涉及到出发城市、APP 渠道、订单类型等维度，各维度又包含一系列的维度值（元素），出发城市：北京，上海，广州等，APP 渠道：支付宝小程序，微信小程序等，订单类型：国内，国外。当总指标 A 发生异常时，最可能的异常原因可以表示为不同维度的元素集合，如{订单类型=国内}，或者{出发城市=北京&广州，APP 渠道=微信小程序}等。这个查找定位交叉维度的细粒度元素集合的过程就是根因定位。

多维指标数据具有以下两个特征:

1）数据的总量大。指标包含多个维度，某些业务指标包含 20+的维度，并且每个维度包含不同的属性值，比如城市维度包含全国 600+属性值;

2）随着数据指标维度的分层，维度集合总数成指数级增长，并且维度之间具有复杂的相互影响关系。

我们的问题可以归结为，在所有维度和维度值组成的集合里，找出导致总指标发生突变的维度和维度值子集。这样需要设计一个合理的得分函数，来评估每个子集是否是根因的得分，然后选出得分最高的子集即可，由于搜索空间很大，所以需要各种启发式搜索方法或者剪枝策略。

我们调研常见的几种启发式搜索算法。

Adtributor【2】算法假定导致总指标异常的根因只会出现在某一个维度上，它提出 EP 值（解释力）和 S 值（惊奇力）两个得分指标来评估子集，EP 值表示该子集的指标波动和总指标的波动的比例，S 值是指维度下各个维度值的取值分布是否有变化。它的核心思想是将多维度根因分析问题分解为多个单维度根因分析问题,采用解释力和惊奇力定位每个维度下的异常元素集合，最后根据每个维度总的惊奇力值大小汇总输出根因集合。

HotSpot【3】设计了一种叫 potential score 的得分函数，并显式的考虑了多个根因同时作用的情况，采用蒙特卡洛树搜索（MCTS）算法来解决巨大搜索空间的问题，并使用了分层剪枝策略降低搜索复杂度。

Squeeze【4】算法主要是针对 HotSpot 进行了三个改进，首先提出泛化的 RE（ripple effect）原则，可以直接处理由可加性指标复合得到的率值指标；其次改进的 PS（potential score）得分函数能捕捉到变化幅度较小的异常；最后是先对细粒度属性组合进行聚类，然后在每一类组合中去搜索根因，显著降低定位时长。

Psqueeze【5】是对 Squeeze 的扩展，它提出一种新的基于 GRE（general ripple effect）的概率聚类方法，将属性组合分组到不同聚类中，然后根据 GPS（general potential score）来评估属性组合是根因的可能性。

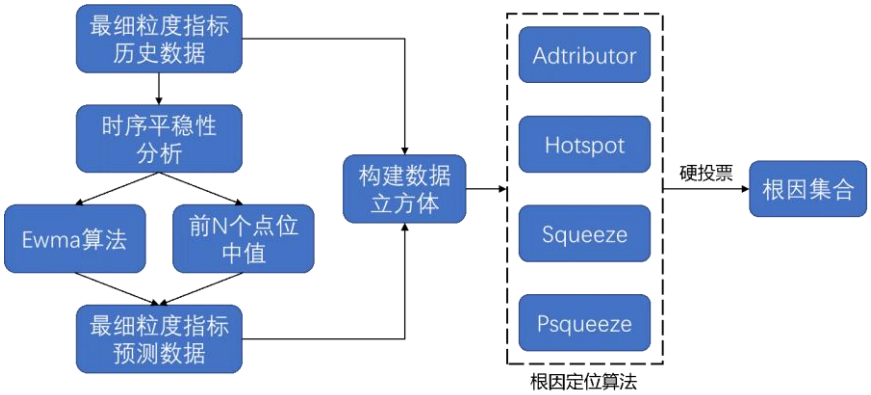


图 4 根因定位系统流程图

最后我们根据火车票业务指标的特点，设计了如图 4 所示的根因定位系统，当异常检测系统检测到某个指标出现异常时，即进入根因定位系统。根因定位是根据指标真实值和期望值的差距大小来确定根因的，它分为数据构建和根因定位算法两个模块:

1）构建数据。首先对指标进行维度拆分，得到最细粒度指标的历史数据时间序列，再根据序列特性采用不同算法进行预测。如果序列具有平稳型，采用指数加权平均法（Ewma）进行预测，否则使用异常点前 3 个点位的中值作为预测值，最后结合历史值（真实值）和预测值（期望值）建立数据立方体。

2）根因定位算法。使用四种常见的根因定位算法分别定位，组成硬投票系统，按照票数倒序输出根因集合。根据火车票核心业务指标维度和维度值多的特点，结合奥卡姆剃刀原则思想，微调了 Adtributor 算法 EP 值阈值，以及对 Hotspot 算法的 PS 评分函数进行修正，在不损失根因定位准确率的情况下尽量精简根因集合。考虑到检测时长，当待检测指标元素组合数量大于 1000 时，不使用 Hotspot 算法，元素组合数量超过 10000 时，只采用 Adtributor 进行根因定位。

三、异动归因实践结果

案例业务背景：出行相关的票量受自然天气影响较大，当出现极端天气时，票量一般会出现突然下跌或上涨，如图 5 所示，该情况需要及时识别异常，并分析出引发指标异常的相关维度和维度值。

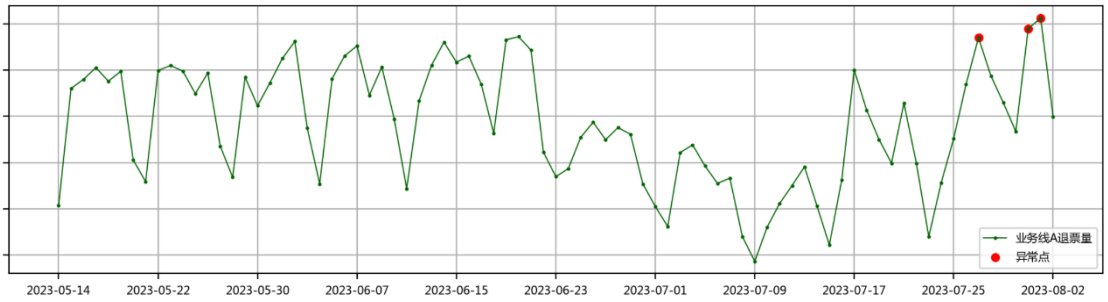


图 5 业务线 A 退票量的异常检测结果

随着台风“杜苏芮“在 7.24 日被中央气象局升级为超强台风，并在 28 日上午登陆福建省晋江市，近日来造成多地区出现极端强降雨现象，导致大量交通工具延误甚至取消。图 5 展示的是业务线 A 在近两个月的退票量情况，退票量有较明显的工作日高于双休日的周期性规律，7 月 27 日，7 月 31 日和 8 月 1 日出现较明显的退票小高峰，符合需要识别异常点的预期。

在检测出异常之后，根因定位系统对出票票量的城市维度进行根因定位，结果如表 2 所示，

日期/定位维度	出发城市	到达城市
7 月 27 日	揭阳&厦门	厦门&南昌
7 月 31 日	北京&南京	北京&厦门
8 月 1 日	北京&成都	北京&广州

表 2 根因定位结果表

为了分析根因定位系统的准确性，我们手动计算城市维度的根因得分，然后对比算法和人工分析的异同。首先根据 adtributor 算法的检测原理，分别计算 EP 值和 S 值，方程式如下，

$$EP_{ij} = (A_{ij}(m) - F_{ij})/(A(m) - F(m))$$

其中，A 为真实值，F 为预测值，下标 i 为维度名，j 为维度下的元素名，m 为异常指标名。

$$S_{ij}(m) = 0.5 \times (p \times log\left(\frac{2 \times p}{p+q}\right) + q \times log\left(\frac{2 \times q}{p+q}\right))$$

其中，p 为先验概率，$p_{ij}(m) = F_{ij}(m)/F(m)$，q 为后验概率，$q_{ij}(m) = A_{ij}(m)/A(m)$。然后以 EP 和 S 的加权和作为根因得分 RT_score，得到不同异常日期的城市维度的人工根因定位结果，如图 6 所示，

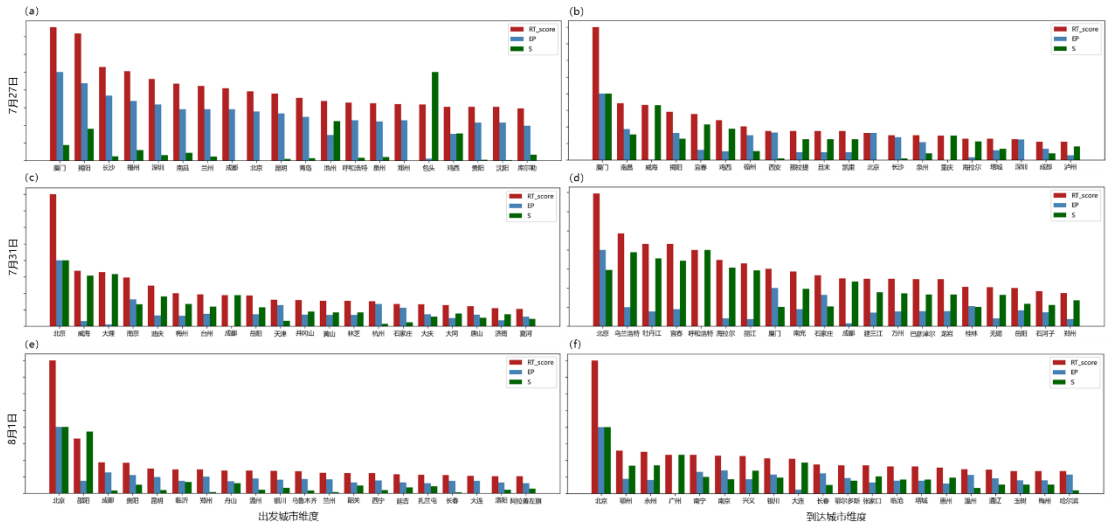


图 6 不同日期城市维度的人工根因定位结果

从图 6 可以看出，7 月 27 日的退票票量异常，从出发城市维度分析，主要是由揭阳和厦门造成的；从到达城市维度分析，主要是由厦门和南昌造成的，人工定位的和算法检测的根因结果相符合。而超强台风杜苏芮正是在 7 月 28 日登陆福建，造成东南沿海部分城市出现暴雨天气，导致大量航班和列车取消或者停运，从而影响所在地区城市的退票票量异常升高。

7 月 31 日和 8 月 1 日的退票票量异常，不论是从出发城市还是到达城市分析，北京都是引起退票退票升高的首要原因。这与根因定位系统的检测结果吻合，并且能将变化幅度大但是体量小的城市剔除。

比如 7 月 31 日的出发城市维度分析，威海和大理的 RT_score 较高，但是由于其退票量体量很小（只占南京 15%的票量），算法没有将其判定为根因。这两天退票量异常的根本原因依然是受超强台风杜苏芮的影响，7 月 29 日至 8 月 1 日杜苏芮在华北地区造成“远距离台风暴雨”，北京地区超 300 趟次航班取消和 43 趟列车停运，导致北京市退票量异常升高。

这个案例较好的展示了异动归因系统对陡升型异常的检测能力，同时能快速并且相对准确的定位到导致异常的原因。

四、总结和展望

4.1 总结

本文主要介绍了异动归因系统在火车票业务指标上的初步应用，首先分析业务指标的时间序列类型，以及不同业务数据出现的异常类型，针对指标的异常类型选择合适的检测算法。然后利用根因定位系统发掘引发异常的原因，根据不同的数据类型选择相应的预测算法，构建根因定位数据集，最后利用四种根因定位算法进行根因定位。

目前异常检测系统处于试运行初期，仅针对核心业务指标做监控，对平稳型时序和强周期型时序的异常检测能力较好。随机挑选了部分指标的异常检测案例，对照人工核验结果，精确率为 67%，召回率为 83%，F1-score 为 74%，为了满足核心指标减小漏报的需求，调低投票

阈值以提高召回率。后期针对接入的非核心指标，可以适当调高阈值提高精确率。

在节假日出行期间，大部分指标迎来一定程度的连续性上涨，这部分日期很容易被算法识别成异常，但绝大部分异常在业务上属于正常的数据波动。目前的做法是对结果做后处理，将节假日的异常设为正常，这样虽然能避免误报，但会漏掉少数真实的异常。一种可以尝试的做法是，对去年同期的数据计算波动情况，根据阈值来判定异常。不过由于火车票业务发展迅速以及疫情原因，往年同期的数据参考意义不大，并且不同业务线指标的阈值设置和调整也是个问题。

最近几年由于疫情的影响，以天为单位的出行相关指标，周期规律难以捕捉，进而影响 STL 分解的准确率，导致检出难度大，这也是目前不选择通过对比实际值与预测值的偏离情况的方法做异常检测的一个重要原因。

根因定位系统可以较准确的定位出维度较少的根因。多维度交叉的根因由于数据量大，细分到细粒度指标的 KPI 统计值就小，而值较小的时序容易引起较大的预测偏差，从而影响根因定位的准确性。

4.2 展望

1）提高其他时间序列类型的检出能力。比如时间序列漂移现象，平稳+漂移，周期+漂移等类型；

2）新增不同类型异常的检出能力。目前异常检测算法对指标陡升和陡降异常具有较好的检出能力，而对整体抬升或整体下降异常，连续性异常的检测能力不佳；

3）在业务需要的情况下，提高多维度交叉的根因定位准确性。

参考文献：

【1】Han S, Hu X, Huang H, et al. Adbench: Anomaly detection benchmark[J]. Advances in Neural Information Processing Systems, 2022, 35: 32142-32159.

【2】Bhagwan R, Kumar R, Ramjee R, et al. Adtributor: Revenue debugging in advertising systems[C]//11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14). 2014: 43-55.

【3】Sun Y, Zhao Y, Su Y, et al. Hotspot: Anomaly localization for additive kpis with multi-dimensional attributes[J]. IEEE Access, 2018, 6: 10909-10923.

【4】Li Z, Luo C, Zhao Y, et al. Generic and robust localization of multi-dimensional root causes[C]//2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE). IEEE, 2019: 47-57.

【5】Li Z, Chen J, Chen Y, et al. Generic and robust root cause localization for multi-

dimensional data in online service systems[J]. Journal of Systems and Software, 2023, 203: 111748.

质量保障

故障召回率提升 34%，携程智能异常检测实践

【作者简介】零一，携程算法工程师，专注于智能告警、容量管理、根因定位等领域。

一、背景

携程作为在线旅游公司，对外提供机票、酒店、火车票、度假等丰富的旅游产品，其系统稳定性关乎用户是否具有顺滑的出行体验。然而，流量激增、代码发布、运维变更等都会给系统稳定性带来挑战。

我们在 2020 年对生产故障的“发现-定位-解决效率”提出了“1-5-10”的目标（即一分钟发现故障，五分钟定位故障，十分钟解决故障），这无疑对监控告警提出了很高的要求。订单量是生产故障异常检测场景中最核心最显性的指标，订单量在自身形态上具有周期性、规律上升和下降、业务高峰和低谷等特点，影响因素包括节假日、促销等。倘若数以万计的业务线通过人工配置规则的方式来覆盖到所有业务场景，并且做到高准确率和召回率，是非常不现实的。因此，迫切需要一套配置费力度低、普适性强、准确率高、时效性强的智能异常检测算法体系来及时发现异常。

指标异常检测是智能运维领域的重要落地场景，携程 AIOPS 团队致力于提升告警质量，寻找告警效率、准确率和真实故障召回率三者之间的平衡点。我们将统计学方法和机器学习方法结合，根据指标的历史数据，将训练的多个模型组成一套异常检测系统，在覆盖真实故障的基础上，减少告警数量，产生更有价值的告警。

二、告警质量提升

2.1 更准确的预测

时序类异常检测算法通常是对指标进行预测，通过对比实际值与预测值的偏离情况来判断时刻 T 是否异常。与其他算法相比，时序类异常检测算法适用于更多的数据类型，并且检测效果也是最优的。

为实现对业务线的精准预测，前期我们尝试了 ARIMA、Holt-Winter、LSTM 等多种时间序列预测方法。携程订单业务指标具有较强的周期性，LSTM 模型捕获序列长期和短期模式的特性导致其在周期性指标上的预测效果优于其他模型。我们以离当前时刻最近的 10 个时间序列数据作为 LSTM 模型的输入，采用滑动窗口不断预测未来时刻指标的取值。

在绝大多数场景下，LSTM 模型的预测效果是非常好的，然而，当指标出现缓慢下跌时，由于在短时间内很难判断出异常，随着窗口的不断滑动，训练数据中包含了异常值，从而导致预测值被带偏。

为了解决这个问题，采用假设检验的方式对指标当前走势做出判断。我们认为，短期内指标的取值应具有一定的随机性，不会具有明显的上升或下降趋势。当假设检验认为指标在

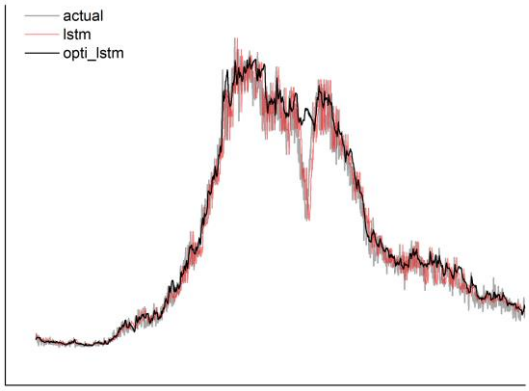
当前时段存在上升或下降的趋势时，不再对 LSTM 模型的输入数据作更新，依然采用上一个窗口的输入数据进行预测。

当假设检验认为当前数据不存在上升或下降趋势时，才会对 LSTM 输入数据进行更新。在做趋势分析时，不应将指标自身的趋势性包括进来，所以，我们可以利用的时间序列数据是非常有限的，此场景下，非参数检验方法要优于参数检验，因此，我们采用 Mann-Whitney U 统计量进行趋势分析。为了评估不同模型的性能，我们选择了三条业务线，并使用平均绝对误差（MAE）作为评估指标进行对比，实验结果表明，经过趋势分析调整的 LSTM 模型的 MAE 值最小，说明预测效果最优。

	ARIMA	Holt-Winter	LSTM	LSTM-Adjust
AA 业务线	0.01519	0.02024	0.01995	0.01483
BB 业务线	0.02975	0.03064	0.02919	0.02612
CC 业务线	0.02314	0.02542	0.02108	0.01690

表 1 不同模型的预测效果

趋势分析的引入，加强了 LSTM 模型的预测精度（如图 1），无论是对于点异常（显著有别于其他点的数据异常）的识别还是对于连续性异常（指标在正常范围内波动，但波动模式却发生了显著变化）的识别，均做了较好的铺垫。



2.2 自适应阈值计算

起初，研发人员会基于自己对业务的理解，设置“人为指定规则”的专家体系，同时，为保障故障 100%召回，阈值设置的都比较敏感。该体系不仅存在准确率低、维护成本高的问题，还可能产生告警风暴，导致告警接收人警惕性麻痹，产生更严重的故障。

为解决这个问题，我们从数据本身出发，基于数据本身的波动性，构造合适的统计学模型，实现阈值的自适应计算。

为便于对异常程度进行衡量，我们定义统计量 Z，假设当前值为y，预测值为y'。当预测值与实际值很接近，Z 值接近于 0；偏离越大，Z 的绝对值也越大（如图 2）。异常事件发生

的概率是非常小的，一般情况下，预测值与实际值非常接近，也就是说，不同时刻 Z 统计量的取值围绕 X 轴上下随机波动。数据分析发现，基于不同时间段计算的 Z 统计量的均值和方差均是一个与时间 T 没有关系的常数，这完全符合平稳时间序列的定义，因此，我们认为，时间序列 Z 属于平稳时间序列。

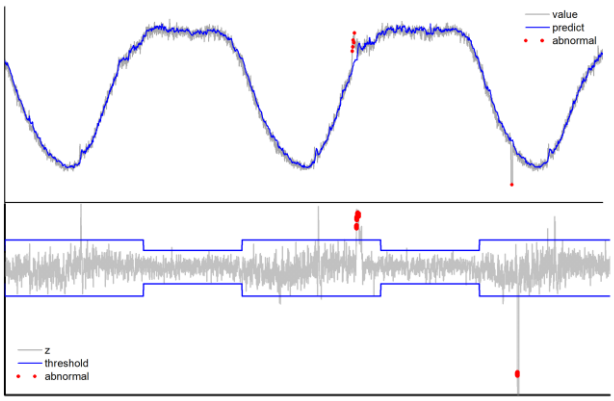
$$Z = \frac{y' - y}{\sqrt{y'}}$$

当数据平稳时，业界一般会对指标的分布情况做出估计，通过上限阈值或下限阈值来实现上升或下降场景的异常检测。对指标的分布情况做出估计的方法称作密度估计。统计学中，密度估计方法包括参数方法和非参数方法两种。参数估计方法假定样本服从某个分布，然后基于假定分布作出区间估计和点估计；而非参数方法一般不利用有关数据分布的先验知识，对数据分布不做任何假设，从数据本身出发做出估计。

实践证明，非参数方法核密度估计（KDE）对指标 Z 的拟合效果要好于高斯，因此，我们采用 KDE 方法对指标 Z 的分布情况进行估计，考虑到异常事件出现的概率远小于正常事件出现的概率，经不断验证和调整，我们一般选择 99.99 分位数作为异常判断的阈值。

我们发现，指标在一天内不同时间段的波动性存在显著差异，业务低谷期间指标随机性更强，波动性相对也更大一些。倘若不考虑指标在不同时间的波动性，每条业务线仅配置一个阈值，由于阈值被平均，可能导致在业务低谷期间误告增多，业务高峰时段真实故障无法被召回。

在统计学中，采用标准差和变异系数来衡量指标的波动性，相比标准差，变异系数剔除了量纲的影响，便于在不同指标间进行波动性的比较。我们基于指标在不同时段变异系数的取值情况，将一天 24 小时划分为低波动区和高波动区，也就是说，一条业务线只要适配两组阈值，便可做到 7*24 小时不间断监控。



通过上述方法，实现了告警体系阈值的自适应计算和优化，告警准确性和故障召回率均有显著提升。与人为指定阈值的方式相比，故障召回率提升了 34%。

2.3 业务趋势分析

我们发现，采用单一异常检测算法在多指标上很难实现通用，为进一步提升告警质量，需要结合其他算法实现异常的层层过滤。下面将讲述我们是如何利用线性回归模型对订单业务指标进行异常检测的。

宏观上看，订单随时间周期性波动，倘若将时间窗口圈定在最近的有限分钟内，便可以采用线性模型对业务趋势进行拟合。在业务高峰来临前，观测值随着时间的推移不断增加，高峰过后，观测值随时间的推移不断下降。由于 Huber-Regression 稳健回归算法为识别为异常点的观测值分配较小的权重，所以，无论训练数据中是否包含异常值，均能得到比较理想的回归线。

因此，我们最终采用 Huber-Regression 算法对指标短期窗口内的观测值进行预测分析。为了对指标在时刻 T 的异常情况做出判断，需要对窗口内指标的波动性进行衡量，观测点与拟合直线之间的距离是指标波动性的一种体现。对于强周期性指标，可以设置较为敏感的阈值；当指标整体量纲比较小，用户行为对曲线形态的影响较大，可以适当调低异常检测敏感度。

引入业务趋势分析，可以有效减少指标抖动产生的误报，从而提高告警准确性。业务趋势分析可以帮助我们更好地理解业务当前趋势，从而更准确地判断异常情况。通过业务趋势分析的引入，告警准确率提升了 30%。

2.4 周期性异常

周期性异常是指在一定时间范围内，某个或某些指标出现了周期性的波动或变化，通常与正常的趋势和规律不符，表现为短期内突然上升或下降。周期性告警属于一种较为明显的误报，在总告警数量中占有不少比例，如果不进行有针对性解决，将会严重影响用户体验。

为了实现周期性异常的过滤，需要考虑周期性异常在时间维度上的偏移。动态时间规整（DTW）是一种用于比较两个时间序列相似性的算法，它可以对两个时间序列进行时间轴上的对齐，从而消除时间上的偏移，使得它们在时间上的对应点更加接近。为实现周期性异常的检测，可以采用以下步骤:

- 利用 DTW 算法实现当前时段数据和历史数据的最优匹配，从而消除时间上的偏移，提高周期性异常的检测准确性。

- 基于当前数据和最优匹配数据进行异常特征提取，例如周期、幅度、相位等，这些特征要尽可能描述当前异常。

- 使用分类算法对异常特征进行判断，从而得出当前异常是否处于周期性异常的结论。

通过上述方法，实现了对周期性异常的准确过滤，周期性误报减少 80%，对于提升告警质量有非常重要的作用。

三、小结

智能异常检测系统包括线下训练和线上实时检测两部分（如图 3）。由于业务变化较快，较远的历史数据参考性不大，因此采用 14 天的历史数据训练预测模型和自适应计算阈值。

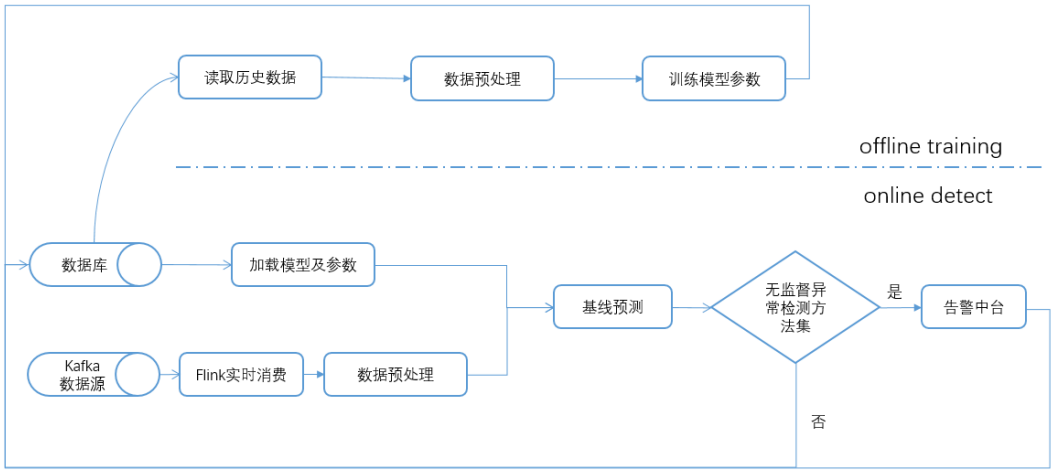
历史数据需要进行预处理，主要包括以下几个方面：

1）异常事件产生的异常数据需要剔除，比如，RCA 时间段、秒杀、考试等外面事件引起指标的突升。同时，采用滑动窗口的方式对异常区间进行缺失值补全。

2）为增强模型鲁棒性，消除数据毛刺点产生的影响，使用滑动窗口对数据进行平滑处理。

3）对数据进行归一化处理，保证 LSTM 预测模型能够快速收敛，提高模型预测精度。

线上检测算法包括：基线预测模块和无监督异常检测方法集。基线预测模块的主要功能是时间序列预测和异常特征提取，无监督异常检测方法集包括一些统计类异常检测算法，如 Boxplot、K-sigma、KDE。无论是点异常还是连续性异常，均会基于异常特征集，采用多种无监督检测方法实现检测，最终以投票的方法得出时刻 T 是否异常的结论。



智能异常检测系统在携程投入使用三年有余。在这期间，我们有遇到各式各样的挑战。经过千锤百炼，取得了不错的效果。告警准确率和召回率均有了显著提升，大部分故障可以在一分钟内被发现。