Trip.com Group Trip Tech

2024年度合辑

携程技术

- **携程一线技术实践**

  一线技术实践分享，共话研发效能新篇章

- **赋能业务遇见未来**

  赋能业务高质量发展，刷新旅游出行新体验

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

随着科技的飞速发展，旅游业正经历着前所未有的变革。携程，作为中国领先的在线旅游服务平台，一直在不断创新和进步，以提供更好的用户体验和服务质量。在这个过程中，携程的技术团队发挥着至关重要的作用。他们不仅负责构建和维护公司的核心技术平台，还致力于开发创新的解决方案，以应对日益复杂的市场需求和技术挑战。

本书旨在分享携程技术团队在一线技术实践中的经验和见解，涵盖了自动化任务平台、BFF 实践、高性能技术等多个方面。我们希望通过这些真实的案例和深入的剖析，为其他开发者和技术从业者提供有价值的参考和启示。

在携程，技术从来都不是孤立的。我们鼓励团队成员之间的交流和合作，以促进知识的共享和技术的创新。我们相信，通过开放的心态和积极的交流，我们可以共同推动行业的发展和进步。

在本书的编写过程中，我们得到了许多同事和朋友的支持和帮助。我要特别感谢那些为本书提供宝贵意见和建议的人，我希望这本书能够激发读者的兴趣，启发思考，并为你的职业发展提供一些有价值的参考。让我们一起探索技术的奥秘，共同推动行业的进步和发展。

携程集团副总裁/技术委员会主席 马超

大前端

携程前端自动化任务平台 TaskHub 开发实践

【作者简介】
工业聚，携程高级前端开发专家，react-lite, react-imvc, farrow 等开源项目作者。
乐文，携程高级前端开发工程师，专注于前端工程化和性能优化。
克旋，携程资深前端开发工程师，关注效率和质量，追求用科学的方式解决问题。
林雄，携程资深前端开发工程师，关注前端新技术应用领域。

一、前言

本文讨论的自动化，是指通过代码的方式，将原本人工完成的相关操作由机器代为处理，如此达到释放人力和提升效率等目标。

然而实现自动化的过程不总是那么顺利，自动化后的效果未必那么理想。可能有的团队会发现自动化脚本尽管释放了业务人员，但成本却转移到开发人员身上。他们需要投入大量时间去更新失效的自动化逻辑，出现问题后的排障时间和难度也随之增加。如果自动化任务的成功率低、问题修复速度慢，那么大量的工作将不得不降级为人工处理，自动化的业务价值无从体现。

我们发现，通过完善自动化任务的相关基建（包括任务调度引擎和任务管理平台等），上述问题可以得到显著的控制和缓解。本文将分享携程旅游研发在这方面的尝试和经验，希望能为其他开发者和团队提供参考。

二、平台背景

旅游内部不少业务都有自动化需求，部分团队已经上线了自己的自动化项目，这些项目由不同的团队维护，但均有相似的痛点：

无法及时关闭自动化任务：由于缺乏有效的控制手段，自动化任务一旦重启就难以中断。

排障效率低下：目前大部分自动化任务是前端自动化任务，前端由于站点更新、网络波动、代理异常等常见原因导致任务无法完成，出错的原因很杂，不能复现场景，难以查找根因。

日志查找困难：现有的日志系统以时间为锚点记录日志，没有划分任务的边界。一次自动化任务产生的日志通常是一个普通请求的 2 到 5 倍，需要从大量连续的任务执行日志中找到需要的信息，并且随着任务增加，存储的日志的文件数量也随之增长，查找变得更加困难。

复现困难：任务的入参通常和日志柔和在一起，导致复现问题时难以分析。

日志没有权限限制：任何人都可以查看日志，可能导致敏感信息泄露。

三、平台目标

为了解决自动化的共性问题，减少重复开发，我们希望通过创建一个统一的平台来提升自动化效率和可靠性，具体目标如下：

提高排错效率：通过详细的日志记录和辅助工具，低成本快速还原任务场景，便于快速定位和复现。

被动感知任务异常：建立实时监控和通知机制，实时检测并推送任务异常信息，减少人为监控负担，完全释放人力。

聚焦业务本身：让开发者专注于核心业务逻辑，不必担心自动化过程中可能出现的性能问题，通过平台提供的工具确保自动化任务高效、稳定地运行。

基于以上目标，我们设计开发出前端自动化任务平台 TaskHub，一个能够帮助自动化提效的解决方案。

四、TaskHub 介绍

在 TaskHub 中，我们把自动化脚本的执行，称为任务。

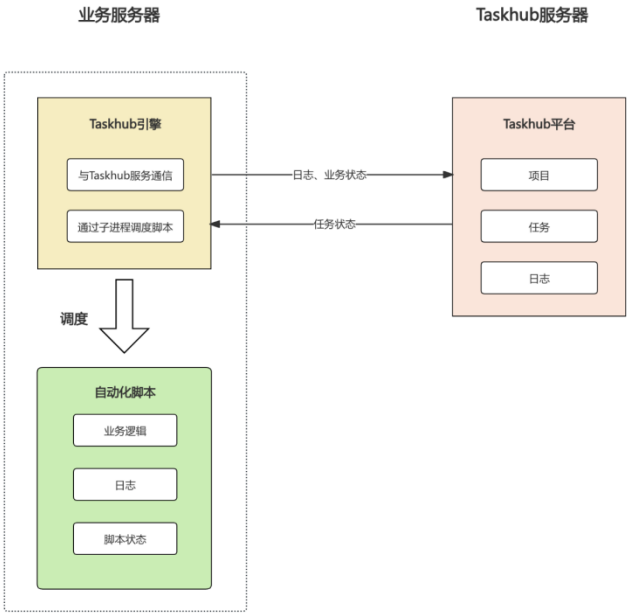
任务是无状态的，任务与任务之间没有直接关系。任务的设计目标是实现某一特定的自动化功能。

在自动化过程中，核心是确保任务的正确执行。任务执行过程中产生的副作用不应阻碍任务的执行。因此，我们将 TaskHub 主要分为两个部分：平台和引擎。平台用于记录和管理任务执行过程中的产物，而引擎专门负责任务的执行。

平台：可以方便查看、配置、中断任务。

引擎：负责调度任务执行，是任务执行的核心。

整体设计：

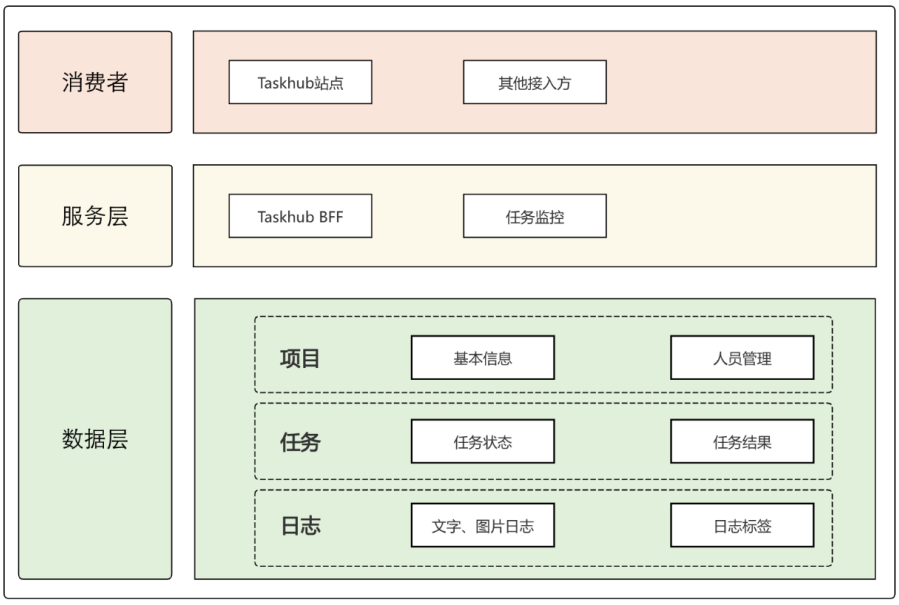


引擎通过 NPM 的方式安装，以 SDK 的方式运行在项目中。这样做有以下几个好处：

1）引擎和平台完全解耦。即使平台不可用也不影响任务运行，确保任务高可用。
2）灵活部署。现有自动化业务不会因为接入 TaskHub 改变原来的部署方式，业务可根据自身需求按需部署。

接下来分别介绍下平台和引擎。

4.1 TaskHub 平台



平台包含三个主要模块：项目、任务和日志。它们之间是一对多的关系：一个项目包含多个

任务，一个任务包含多条日志。每个模块都有严格的权限校验，确保数据安全和访问控制。

项目模块：项目是任务的集合。它包含项目的基本信息，并负责管理项目成员和权限，为同一类任务做统一配置。

任务模块：包含任务的输入、输出、运行时间等信息。任务流程被结构化展示，使任务的输入和输出更加清晰，降低复现难度，提高排错效率。

日志模块：记录任务运行的所有日志，分为业务日志和系统日志两个子模块，方便快速切换和筛查。除了传统的文字日志，TaskHub 还支持图片日志。如果在自动化过程中遇到意外错误，可以截图记录错误页面，留下错误快照，方便后期排查时快速了解错误发生的场景。

4.2 TaskHub 引擎

TaskHub 引擎的核心是调度执行用户的脚本。用户只需编写任务的 .ts 文件，引擎会为每个任务分配一个独立的 Node 子进程，每个任务都在自己的子进程中运行。

这样设计有以下几个好处：

1）数据隔离。借助进程数据隔离的特性，无成本实现业务数据的隔离，带来了数据的安全性。
2）任务易清理。任务执行完成时，使用 process.exit()退出子进程即可释放资源和依赖。此前度假部分自动化任务的方案在释放资源时有较重的心智负担，需仔细编排以避免影响其他任务。

3）任务可远程关闭。度假部分自动化场景有中断执行的需求，独立的子进程使得中断更简单，直接对子进程进行操作即可。

4.2.1 如何初始化项目

在 TaskHub 平台 注册一个新项目后，使用 TaskHub SDK 提供的 engine 对象来初始化 TaskHub 引擎。只需将项目 ID 传入 engine.initProject 方法，引擎就会与 TaskHub 平台 上的项目进行绑定。这样，后续的任务日志、状态等信息就会与相应的项目关联起来。

初始化引擎代码示意如下：

```
import { engine } from '@ctrip/taskhub'

// 初始化引擎
const project = engine.initProject({
  // Taskhub 平台上注册的项目。 上报的日志和任务状态等会与 productId 绑定
  projectId: 1,
  // 引擎执行相关的配置，并发、日志批量时间等
  runnerOptions: {
    scheduler: {
      // 一个 project 实例最多同时执行几个任务。多的任务会被放到本地队列中。默认 10
      workerCount: 10
    },
    logger: {
      // 合并发送日志的最大条数。默认 10 条
      maxBatch: 10,
      // 即使没有达到最大条数，隔特定时间也会执行一次发送。默认 10s
      forceSendTime: 10 * 1000,
    }
  }
})
```

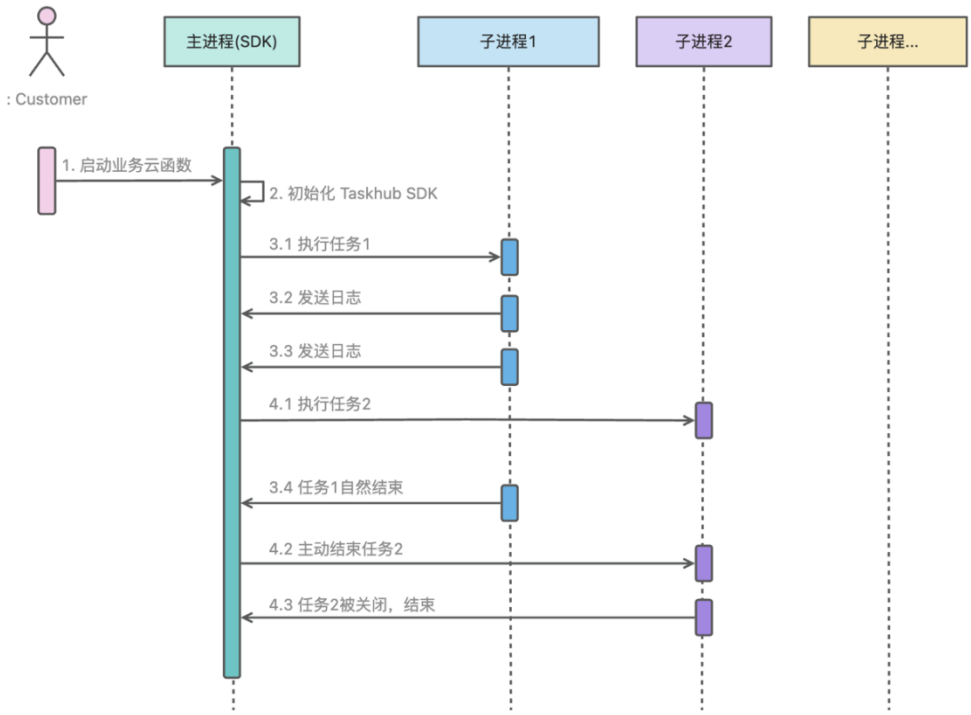
接下来，通过调用 project 的 addTask 方法，即可启动任务，引擎内部会使用子进程运行用户的脚本。

```
const task = project.addTask({
  script: './scripts/task.js',
  description: '',
  env: ''
})
```

以上就是运行 TaskHub 自动化任务的核心代码。

4.2.2 引擎内部设计

在 TaskHub 引擎中，每当一个新任务启动时，引擎会创建一个新的子进程，并在子进程中运行任务，如下图所示：



在简单的自动化场景中，通过主进程启动任务可以满足大多数需求。然而，在复杂的业务场景中，仅仅依靠主进程启动一个子进程来运行任务是不够的。

在任务执行过程中，子进程对于主进程来说是一个黑盒。主进程无法直接了解任务当前运行到哪一步，以及任务的当前状态。为了便于主进程和子进程之间的数据流转，TaskHub 引擎建立了主进程和子进程之间的双向通信机制。

以获取任务运行结果为例，当子进程运行任务后，我们希望在主进程中获取任务的返回结果。

我们可以在主进程中使用之前创建的 task 实例来注册监听事件，等待子进程发送消息。具体代码如下所示：

```ts
/* constants.ts */
export const TASK_RESULT = 'TASK_RESULT'
```

```ts
/* main.ts 主进程 */
import TASK_RESULT from './constants'

// 主进程接收消息
const data = await task.channel.waitForOnce<{ status: 'ok' | 'error' }>(TASK_RESULT)

if (data.status === 'ok') {
  // 处理成功
} else if (data.status === 'error') {
  // 处理失败
}
```

在任务脚本中，根据需要发送任务状态的更新，如下所示：

```ts
/* task.ts 子进程 */
import { childChannel } from "@ctrip/taskhub/child"
import TASK_RESULT from './constants'

let isValid = false

// 业务逻辑处理。。

// 任务脚本子进程发送消息
if(!isValid){
  childChannel.send(TASK_RESULT, {
    stats: 'error'
  })
} else {
  childChannel.send(TASK_RESULT, {
    stats: 'ok'
  })
}
```

需要注意的是，子进程可以向主进程发送消息，而主进程也可以收发子进程的消息。这种双向通信机制在以下场景中非常有用：

1）状态监控：主进程可以实时接收子进程的状态更新，了解任务执行的每一步骤。

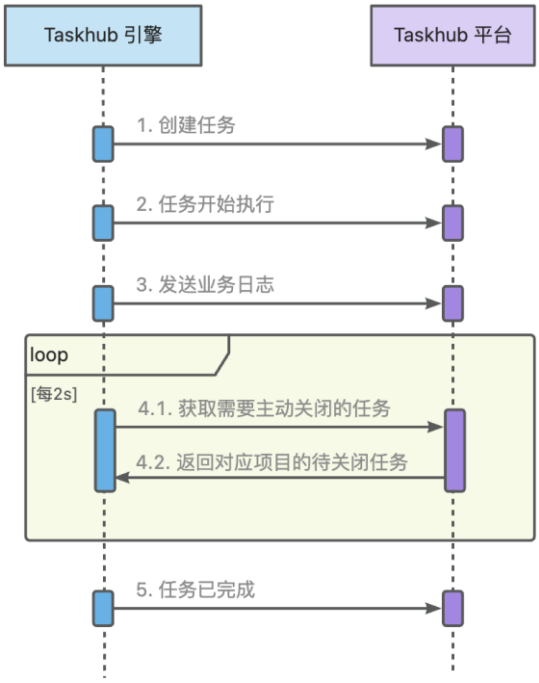
2）任务控制：主进程可以向子进程发送指令，例如暂停、继续或终止任务。

3）异常处理：子进程在遇到问题时，可以立即通知主进程，使得异常情况能够迅速得到响应和处理。

4）数据传输：主进程和子进程之间可以交换数据，确保任务执行所需的信息流畅传递。以上是引擎内部任务调度的实现，接下来介绍引擎如何与 TaskHub 平台进行通信。

4.2.3 引擎外部通信

如上面所说，TaskHub 平台的主要用途是记录自动化脚本执行过程中产生的日志、任务状态，以及其他需要被持久化的状态。所以主要通信的内容包括：任务状态的变更、日志推送、引擎轮询获取需要主动终止的任务。



引擎通信接口

IMessageSender 整体设计中提到 TaskHub 引擎与 TaskHub 平台是解耦的。引擎内部定义了一份接口 IMessageSender，只要实现了接口就能与引擎共同运行，TaskHub 平台只是引擎接口的一份实现。

接口定义如下：

```ts
/**
 * 需要与任务管理平台对接的接口
 */
interface IMessageSender {
  /**
   * 创建一个任务。 返回的是一个有效的 taskId 或者 抛异常。
   * 通常只有 taskhub platform 能返回一个有效的 taskId，其余 sender，直接抛异常就可以了
   */
  createTask(command: CreateTaskCommand): Promise<TaskId>
  /**
   * 传入的 taskIds 中是否有需要关闭的，如果有需要关闭的，则会通过返回
   */
  getTasksToShutDown(taskIds: TaskId[]): Promise<TaskId[]>
  /**
   * 任务关闭后，回传消息给平台
   */
  onShutdown(command: ConfirmTaskShutdownCommand): Promise<boolean>
  /**
   * 发送日志的接口实现
   */
  sendLog(command: SendLogCommand): Promise<{success:TaskId[]; failed: TaskId[] }>
  /**
   * 更新任务状态
   */
  updateTask(command: UpdateTaskCommand): Promise<boolean>
  /**
   * 更新任务结果
   */
  updateTaskBizResult(command: UpdateTaskResultCommand): Promise<boolean>
}
```

我们期望 TaskHub 平台的可用性等级是稳定的，不期望在更高可用性等级要求的应用接入时被迫提升自己的可用性等级。所以，通信接口中关于日志的部分， 引擎对 原有的日志平台 也做了一份实现，作为 TaskHub 日志系统的兜底方案。

当 TaskHub 平台不可用时，引擎与平台的通信会降级到兜底方案，此时部分能力是受限的，例如终止任务的能力。但是这并不会影响自动化任务的执行，引擎的调度能力、脚本的日志留痕等能力仍然可用，并且，所有运行日志可以在原有的日志平台获取。

以上是引擎设计相关的内容，除此之外，TaskHub 还提供了两个辅助 SDK 以补充 TaskHub 平台的使用：

logger：帮助用户记录日志到 TaskHub 平台，支持文字和图片日志。

media：集成了 OSS 平台，提供简单易用的 API 将本地图片或 base64 格式的图片上传至服务器，方便任务运行过程中对图像数据的管理。

五、使用案例

5.1 度假业务自动化数据录入

度假业务内部有一个需求，业务人员需要定期在某个站点录入数据。后来，将数据结构化处理后，通过自动化程序定时进行数据录入，代替之前的人工操作。

虽然这种自动化模式解决了一部分问题，但在实践中也发现了一些新的问题：

1）排障效率低。在自动化的过程中，由于站点加载的资源很多，涉及出错的原因很杂，通过现有日志系统排障需要投入大量时间，效率低下，占用了宝贵的开发时间。

2）无法及时关闭单个任务。目前只能通过关闭整个自动化应用来关闭某个正在运行的任务，这个过程不仅会关闭其他正在运行的任务，而且整个关闭的流程很长，无法及时关闭。

在接入 TaskHub 后，自动化系统的容错率和排障效率得到了显著提升。

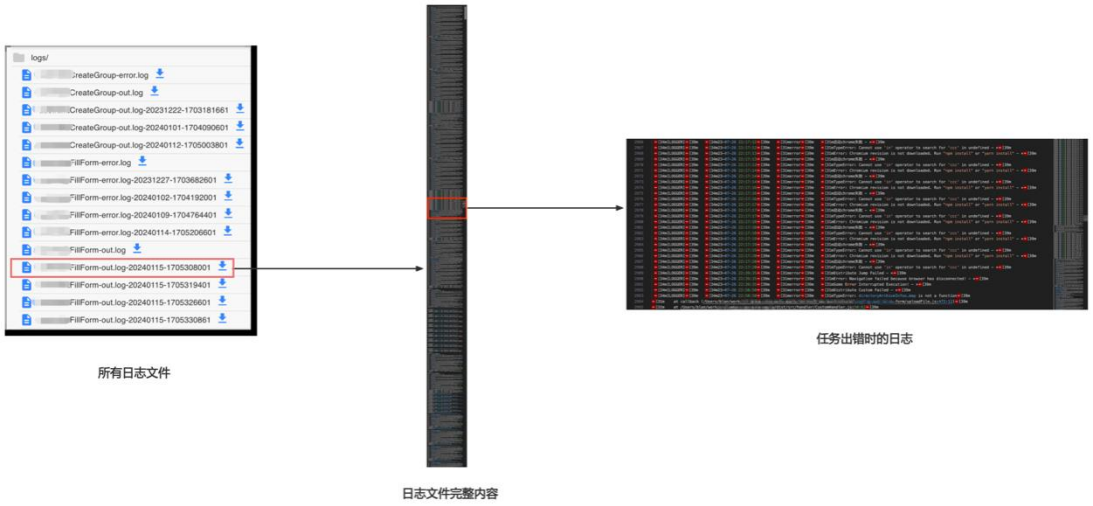
收益：

及时关闭任务：每个任务运行在不同的子进程上，可以杀掉子进程立即关闭单个任务。

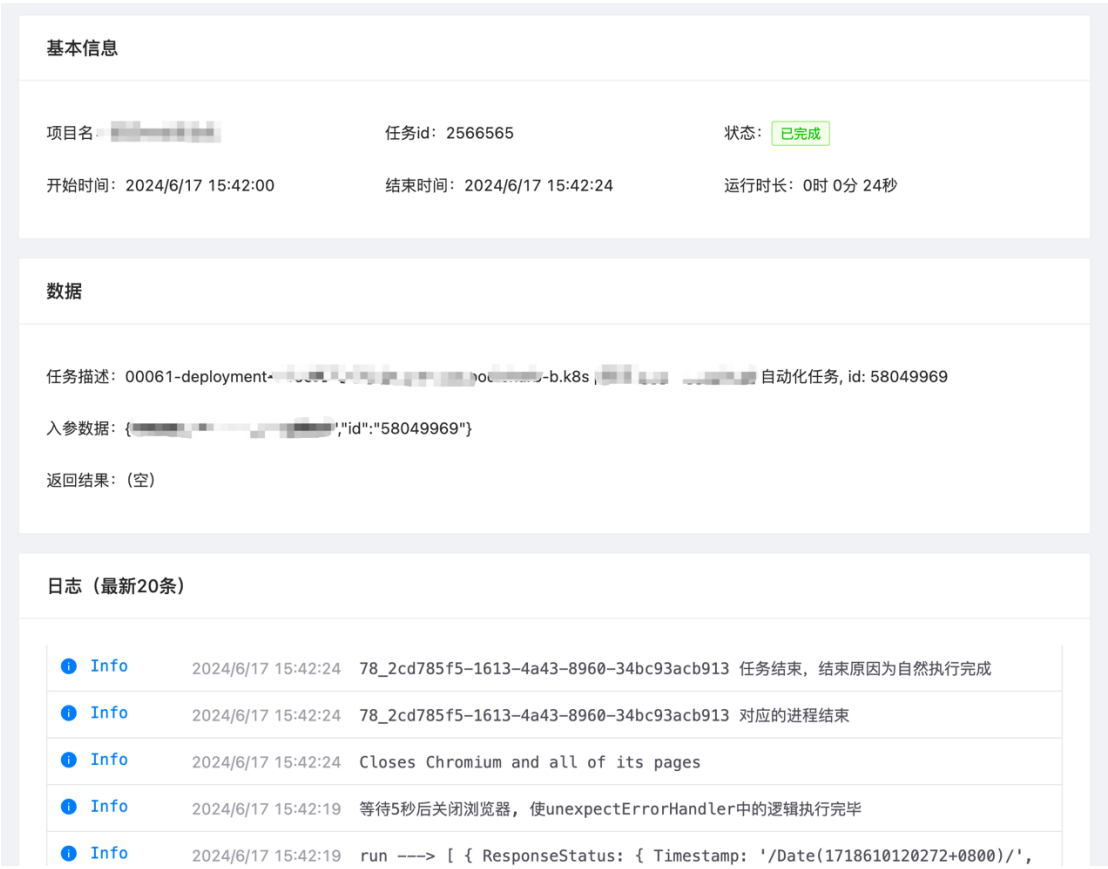
提升日志排障效率：除了可以及时关闭外，TaskHub 显著提升了查找日志排障的效率，节省了大量时间。

以下图表对比了传统日志查找与 TaskHub 任务日志的差异：

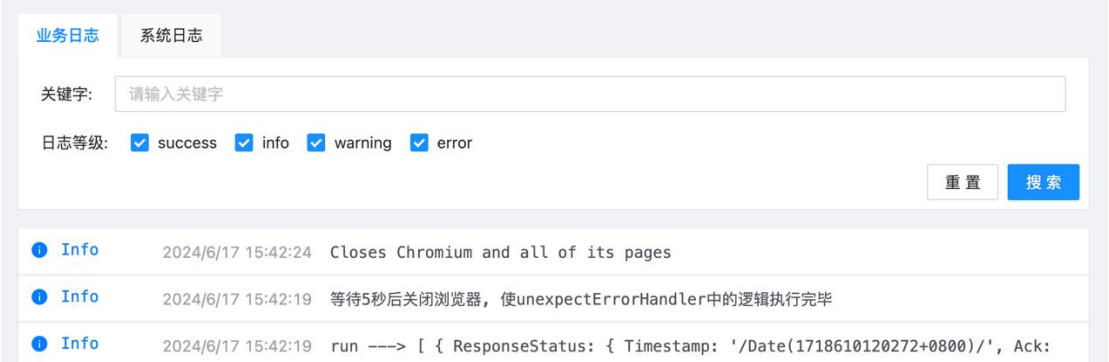
传统日志查找：开发人员需要从大量日志文件中找到错误发生时的日志文件，然后再从文件中定位到异常任务的错误日志，费时费力，容易遗漏关键细节。



TaskHub 任务日志：TaskHub 为不同任务划分边界，不再需要从大量杂乱的日志中寻找关键信息。每个任务的日志被结构化记录，便于快速查找和定位问题。此外，任务的输入输出也清晰可见，可以快速复现场景。



任务详情



任务日志

5.2 复杂业务场景的自动化改造

在更为复杂的业务场景中，自动化需求往往更加多样化。例如，当前有一个需求：人工需要定期在某个站点上查看页面特定元素的状态，如果满足某些条件，则去另一个站点录入数据。

为了将该项目进行自动化改造，可以将场景拆分为以下两个需求：

需求 1：在某个站点上通过查找页面来获得业务数据。

需求 2：定期在某个站点上录入数据，录入数据之前需要判断【需求 1】中的业务结果。

具体实现方案：

需求 1：设计成一个接口，通过调用接口返回的结果来获得业务数据。

需求 2：设计成定时任务，定期执行。每次执行之前先请求【需求 1】的接口，判断条件是否满足。

这是很自然的自动化改造，能够释放人力资源，并且任务也被合理地拆分。然而，在实际运行中，可能无法完全达到预期效果：

1）人工并未完全释放：加入自动化后，人工仍需定时关注自动化任务是否正常运行，无法彻底摆脱手动监控。

2）排障复杂度增加：随着前置任务的增加，自动化排障的复杂度直线上升。当【需求 2】的任务运行出现异常时，如果发现是由于前置的【需求 1】出现问题，就需要根据错误的发生时间去【需求 1】的机器上查找日志。若有多个前置任务，排查成本将大幅增加。

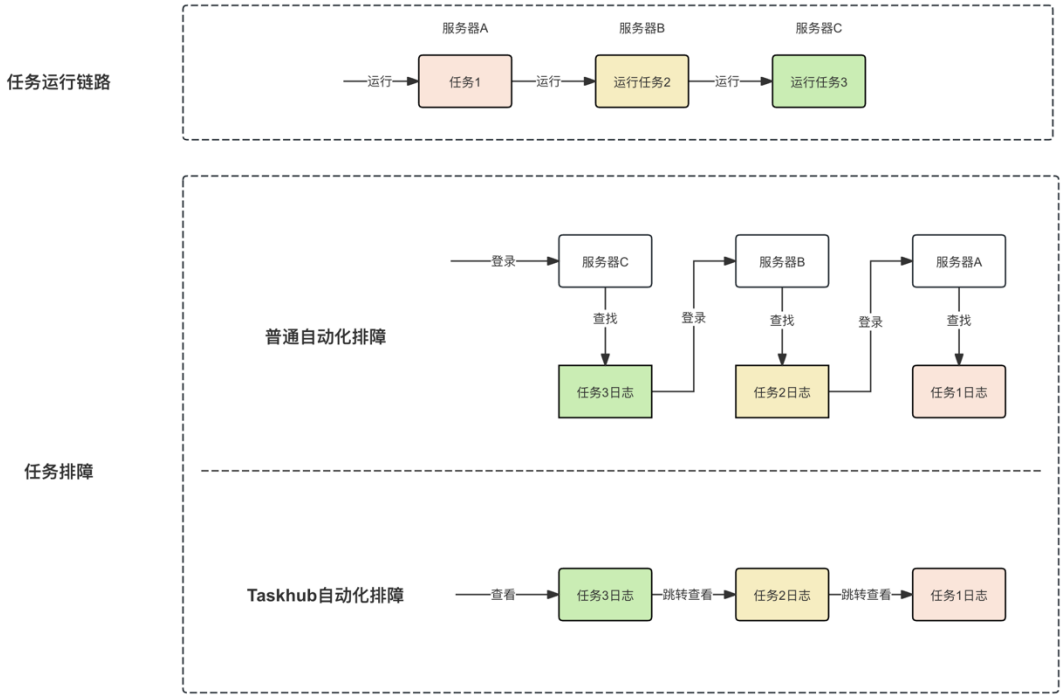
我们来看下加入 TaskHub 之后，会有哪些改变。

1）任务异常主动通知：从原来的主动查看任务状态，转变为被动接收任务异常通知。任务出现异常时，系统会主动提醒，减少了人工监控的负担。

2）快速排障：通过捕捉错误快照，将其记录到 TaskHub 的图片日志中，开发人员能够快速定位并解决问题。

3）清晰的排障链路：TaskHub 不仅提升了单一任务日志的查找效率，对于多个串联任务也同样适用。通过在下游日志中打印上游任务 ID，可以快速查找上游日志，无需在多个机器日志中来回跳转。

下图展示了一个三个串联任务的排障例子。



值得注意的是，TaskHub 并没有改变原有项目的设计，只是将任务的运行方式从 Node 转变为 TaskHub 引擎。

TaskHub 将自动化中的任务概念独立出来，本质上，任务就是脚本的执行。无论任务是通过接口调用、定时函数还是定时器唤起，任务的唤起方式虽然不同，但任务执行的逻辑保持一致。这样一来，开发者可以更专注于任务的逻辑实现，再根据需求唤起任务即可。

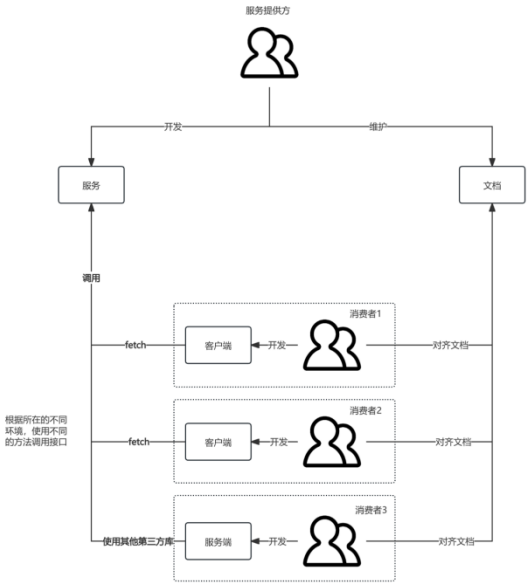
到目前为止，已有 12 个自动化项目使用 TaskHub 运行，累计完成了 48w 次自动化任务，记录了 1300w 条日志。

六、RPC BFF 最佳实践

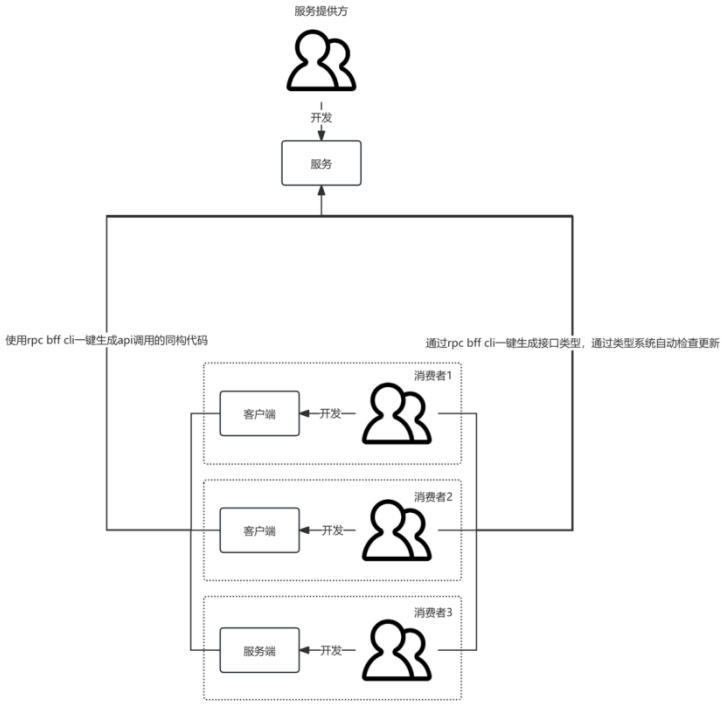
在进行 TaskHub BFF 的技术选型时，我们对比了当前多种主流的技术栈。由于 TaskHub BFF 有多个调用方，并且在客户端和服务端都有消费场景，综合考虑上手成本和多个消费者之间的同步成本等因素，最终选择了 RPC BFF。

关于 RPC BFF 更多介绍可以查看文章《携程度假基于 RPC 和 TypeScript 的 BFF 设计与实践》。

传统服务开发中，服务提供方需要撰写服务代码，然后在第三方平台同步接口文档，消费者再根据文档约定在不同调用环境中使用这些接口。



而在 RPC BFF 模式的开发中，服务提供方撰写服务代码之后，消费者只需通过一个命令就可以获得接口的调用文档和调用函数。这种方式不仅简化了开发流程，还提高了接口调用的一致性和可靠性。



在使用 RPC BFF 的过程中，我们也总结了一些最佳实践，希望能够给大家带来一些启发。

6.1 收敛类型

在设计 RPC 接口的返回值时，有些开发者可能会沿用朴素函数的设计思路，即当接口返回成功时，返回一个成功标识和数据；当失败时，则返回一个失败状态和错误码，让消费端根据不同的错误码进行相应操作，如下所示：

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

// 定义输入类型
class Input extends ObjectType {
  status = Literal('success')
  data = Object
}

// 定义输出类型
class Output extends ObjectType {
  status = Literal('fail')
  errorCode = String
  message = String
}
```

这种设计存在以下几个问题：

1）逻辑不够清晰：简单地将返回结果分为成功和失败，实际开发中可能还有“可接受的错误”或其他复杂的状态，扩展性不佳。

2）错误码维护复杂：前后端都需要维护同一套错误码，增加了开发和维护成本

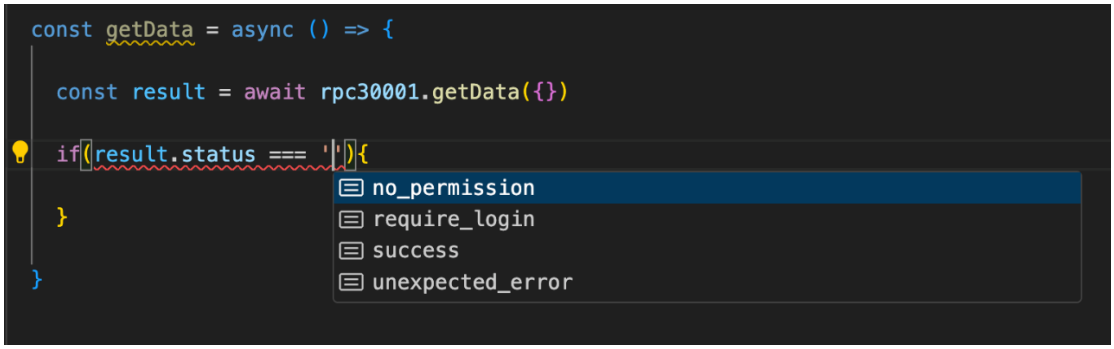
3）错误处理遗漏：编码时容易遗漏未处理的错误状态，增加了系统不稳定性。

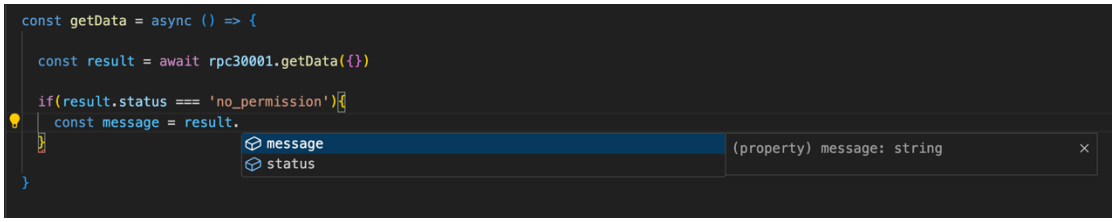
其实，可以使用 RPC BFF 的 Union 类型，将所有可能返回的状态联合起来，合并为最终返回的类型。

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

// 获取数据成功
class Success extends ObjectType {
  status = Literal('success')
  data = Object
}
// 没有权限
class NoPermission extends ObjectType {
  status = Literal('no_permission')
  message = String
}

// 需要登录
class RequireLogin extends ObjectType {
  status = Literal('require_login')
}

// 未知错误
class UnexpectedError extends ObjectType {
  status = Literal('unexpected_error')
  message = String
}

// 聚合输出
const Output = Union(Success, NoPermission, RequireLogin, UnexpectedError)
```

通过聚合所有返回结果，不再需要维护同一套错误码，并且通过类型系统就能确保所有可能的状态都被处理。如下图所示，在消费返回结果时，代码编辑器中有代码提示，不会遗漏任何状态。

如果接口或类型有更新，只需要一个命令就可以同步更新接口和接口类型。





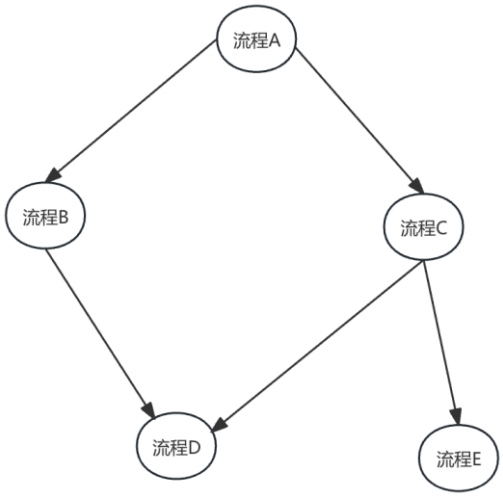
6.2 原子化过程

RPC BFF 的核心理念是面向函数的接口编程，得益于它对底层通信细节的封装，开发者只需考虑函数的功能即可。

与 RESTful 优先资源的理念不同，RPC BFF 的原子是一个过程。REST 将资源暴露出来，而 RPC 是将过程暴露出来。

所以，可以通过分析服务流程（过程），将其拆解为最小不可分割的流程，再将这些流程通过函数实现。

只需保证所有原子化的过程都被很好地处理。在原子过程之上的过程调用，只需按需组合这些原子过程，从而形成一个过程流，最终形成一个有向无环图（Directed Acyclic Graph）。



例如，有一个接口接收一个 ProjectId，并返回该 ID 对应的项目信息。如果未获取到项目，则返回未找到。

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

// 定义输入
export class Input extends ObjectType {
  projectId =  Number
}

// 定义Project类型
class Project extends ObjectType {
  /* Project的属性 */
}

// 定义成功返回类型
class Success extends ObjectType {
  status = Literal('success')
  data? = Project
}

// 定义异常类型
class NotFound extends ObjectType {
  status = Literal('not_found')
  message? = String
}

// 聚合返回类型
export const Output = Union(Success, NotFound)

// 定义RPC函数
export const getProjectById = Api(
  {
    input: Input,
    output: Output,
  },
  async (input) => {
    try {
      // 尝试获取project信息
      const result = await getProjectDao(input)
      if (!result) {
        const message = `未找到项目${input.projectId} 的信息`
        return { status: 'not_found', message }
      }

      // 构造需要的返回数据
      const data = {
        ...result
      }

      return { status: 'success', data }
    } catch (error: any) {
      return { status: 'not_found', message: error.message }
    }
  },
);
```

getProjectById 既可以对外暴露，也可以在其他函数中调用。

假设有一个业务流程，需要首先检查项目是否存在，然后根据项目状态执行不同的操作。如下图所示，在 getProjectSetting 中，我们调用接口的方式就像调用本地函数一样，并且利用前面说的返回值类型组合，清晰地完成代码逻辑。

```
import { Api } from '@ctrip/rpc-bff'
import { getProjectById } from './api'
// 定义RPC函数，获取project setting
export const getProjectSetting = Api(
  {
    input: Input,
    output: Output,
  },
  async (input) => {
    try {
      const { projectId } = input;

      // 直接调用之前的RPC函数，检查project是否存在
      const isProjectExist = await getProjectById({ projectId })
      if (isProjectExist.status === 'not_found') {
        throw new Error(isProjectExist.message)
      }

      // 其他业务逻辑
      return { status: 'success', data }
    } catch (error: any) {
      return { status: 'fail', message: error.message }
    }
  },
);
```

通过这种方式，有以下几个收益：

1）高复用性：原子化的函数可以在不同的业务场景中复用，减少代码重复。

2）清晰明确：每个函数只负责一个具体的操作，逻辑清晰，易于维护和测试。

3）组合灵活：可以根据业务需求，灵活组合原子化过程，构建复杂的业务逻辑。

七、结语

通过重新梳理整个自动化流程，我们拆分出了运行自动化任务的核心部分和辅助模块。在保证自动化任务高可用的基础上，提高了自动化的整体效率和容错率。考虑到自动化场景的复杂性，我们特别设计了主进程与子进程的双向通信机制，以应对各种自动化场景的挑战。

此外，我们还提供了两个自动化场景案例，详细分析 TaskHub 如何帮助提升自动化效率。

最后，我们结合团队在 RPC BFF 的实践，分享了一些使用经验。

未来，我们将继续探索更多的自动化场景，并不断完善 TaskHub。

能效变革，携程酒店前端 BFF 实践

【作者简介】
携程酒店研发前端 BFF 组，专注 BFF 研发实践及效能提升；
携程云函数研发项目组，专注新一代 FaaS 研发模式在携程的落地实践；

引言

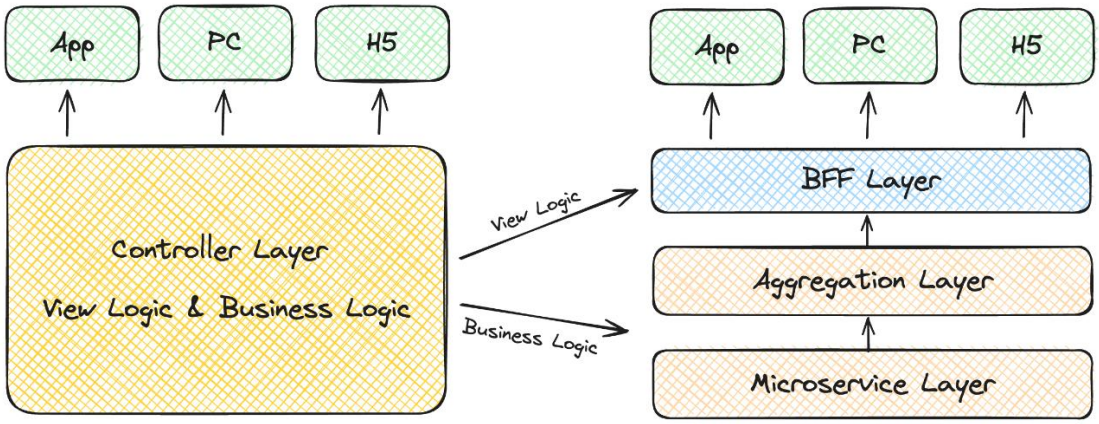
本文概述了携程酒店前端 BFF 层在架构迁移及效能提升过程中面临的挑战和应对方案。第一部分描述了 BFF 实践过程中遇到的问题，分析了两种 BFF 模式的对比并提出了一码多端的 BFF 研发方案；第二部分通过介绍携程云函数平台能力来阐述其如何帮助提升 BFF 研发的效能；第三部分简单介绍了前端动态化能力的未来规划。

一、背景介绍

随着互联网和移动设备的普及，用户对于应用的需求也越来越多样化和个性化，这就要求应用程序在不同的端类型上表现出差异化的交互和 UI。同时，在后端微服务化的变革浪潮下，后端开发人员需要专注在领域服务及建模的工作中。而交互/UI 的变更频率往往要高于领域服务及相关模型，加之前/后端本身的差异性也会无形中加重后端开发人员的心智负担。这就使得后端开发人员既无法专注于领域模型的抽象，又无法敏捷灵活地适应不同的用户界面差异，限制了整体的研发效率提升。

为了解决这个问题，BFF 作为一种研发模式被提出。其作为“面向前端的后端服务”，作为中间层在软件架构上隔离了领域服务层及前端 UI 层。随着架构被隔离，相关的开发人员及开发角色也随之被清晰的分开：前端研发负责 BFF 的视图逻辑，后端研发则可以专注在领域服务层当中。

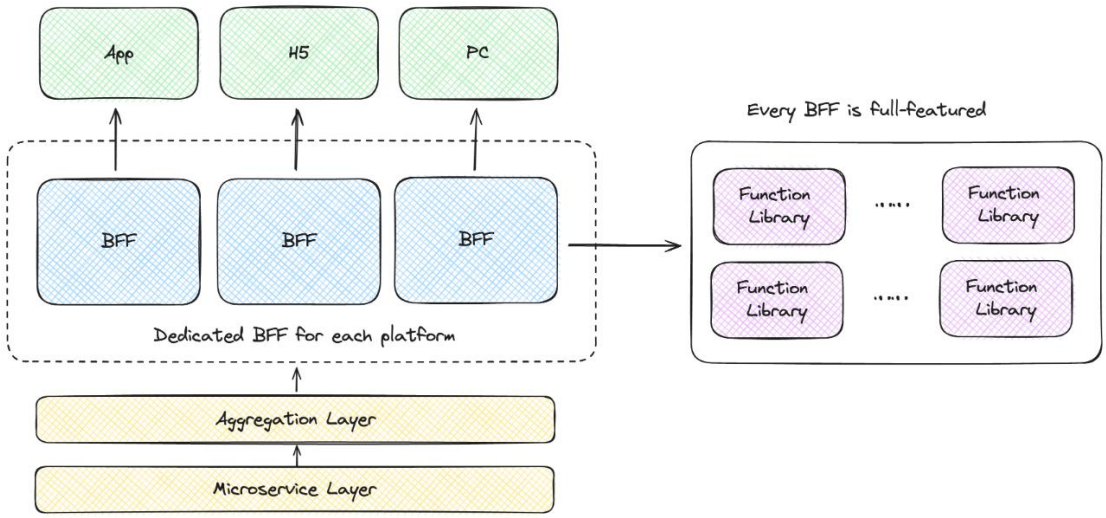
这种模式改变了原来的生产关系，令后端开发人员可以更专注于特定领域模型及服务的搭建，不必过多关注频繁的 UI 层变动。而 BFF 开发人员则可以与前端开发人员更紧密的合作（甚至前端同时兼任 BFF 开发工作），提供更符合前端场景需求的接口及数据结构。BFF 不是一项突破性的生产力变革，更多是一种生产关系变革，通过前后端协作模式的调整来适应互联网领域的敏捷开发节奏。



传统的 API 层包含了视图逻辑和业务逻辑，统一由后端开发进行维护。BFF 层承担其中视图逻辑的职责，而业务逻辑则“下沉”到领域微服务层进行维护。

在现代微服务架构模式下，BFF 作为一类后端服务也被部署在微服务架构中。它作为整个架构中前/后端应用的中间层，也需要考虑其内部各个 BFF 应用的职责分工问题。通常，可以按照‘领域驱动设计’来进行微服务职责的划分：把特定领域范围内的功能划分到一个微服务上做到聚合，把彼此关联性较小的功能划分到不同的服务上满足解耦。在携程酒店业务 BFF 架构实践过程中，针对 BFF 微服务职责的划分问题，出现过 2 种模式：“一码一端”和“一码多端”。

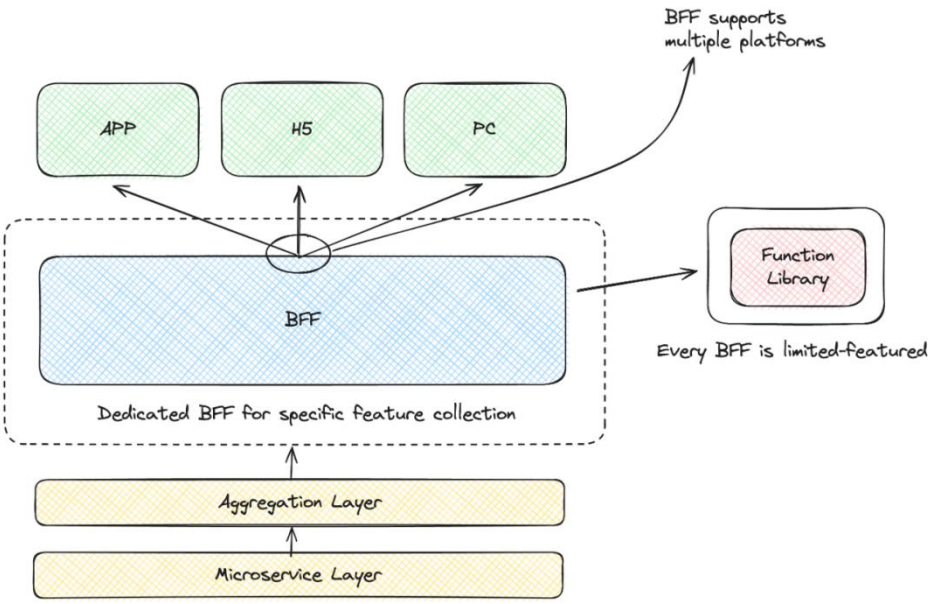
1.1 一码一端



这个模式针对特定客户端提供一个单体 BFF 应用，该应用提供整个酒店预定主流程的全部服务能力。为各端提供了充分的控制端差异的能力，独立开发独立部署。在 Ctrip 小程序端的 BFF 上有过使用，特点是能够快速部署，独立迭代，适合小团队运维和快速上线。

但缺点是出现了单体巨型应用，在一个服务内承载了列表/详情/填写全流程的前端服务能力，整个应用架构显的臃肿。并且针对同一个产品需求，各端的 BFF 应用内部需要各自实现相似的视图控制逻辑，在实现业务需求的时候存在重复开发的弊端，不利于提高人效。

1.2 一码多端



这个模式将预定主流程划分为若干关键阶段，分别由不同的 BFF 提供服务，且一个 BFF 俱备服务多端的能力并在内部处理不同端的差异性，避免了重复开发，从而提升研发侧整体的生产效率。且由于按页面流程维度进行了微服务划分，架构上更为清晰，迭代也更加灵活。

从两种模式的实践结果来看，“一码多端”的模式更符合微服务职责划分的原则，也更有利于提高研发人效。因此，当前酒店的 BFF 层的实践方向主要采用了“一码多端”的模式。曾经采用“一码一端”模式的 BFF 应用也将向“一码多端”模式重构迁移。

二、基于 NestJS 的多端架构

前述已经谈到了 BFF 的概念以及酒店业务 BFF 的微服务划分方式，接下来谈具体实现层面的问题。

首先，我们选择了 NestJS 作为酒店 BFF 基础框架进行二次开发。NestJS 是一个用于构建高效、可扩展的 NodeJs 服务器端应用的框架。它使用渐进式 JavaScript，构建并完全支持 TypeScript。并结合了 OOP（面向对象编程）、FP（函数式编程）的特点。相比其他框架，有如下一些优势：

1）强大的模块化和可扩展性：使用模块化的结构来组织代码，这使得代码更易于组织和维护。此外，NestJS 还提供了一种插件系统，允许开发者扩展框架的功能。

2）内置的依赖注入容器：内置了一个强大的依赖注入（DI）容器，使得服务和控制器之间的解耦变得简单，同时也提高了代码的可测试性。

3）TypeScript 的支持：基于 TypeScript 编写的，这意味着可以利用 TypeScript 的静态类型检查和最新的 ECMAScript 特性。也可以使用纯 JavaScript。

4）装饰器和元数据反射：大量使用了装饰器，这使得代码更易于理解和维护。同时，通过元数据反射，NestJS 可以自动处理很多常见的模式，如路由、依赖注入等。

5）支持微服务：提供了一套完整的微服务解决方案，包括消息模式、传输策略等。

6）测试工具：提供了一套强大的测试工具，使得单元测试和端到端测试变得简单。

7）与其他库的集成：可以轻松地与其他流行的 JavaScript 库和工具集成，如 TypeORM、Passport.js、GraphQL 等。

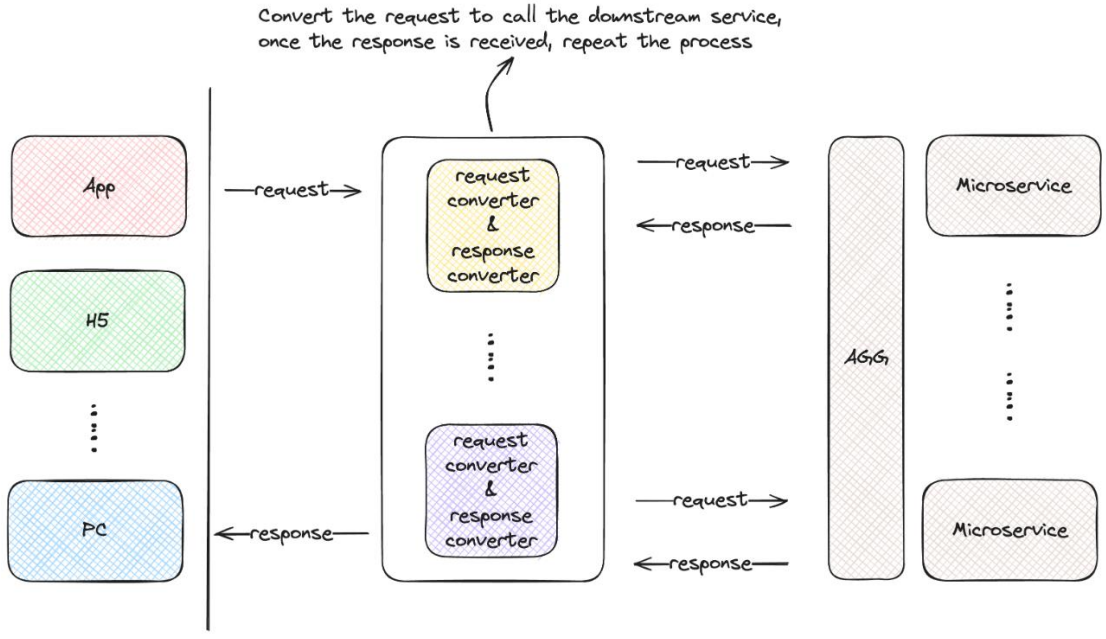
在 Nest 框架提供的 IOC 能力基础上，酒店 BFF 研发组构建了专用于支持"一码多端"研发场景的 BFF 服务框架，试图通过统一的状态数据管理及策略流程模式支持多端适配能力的开发：

1）提供了标准的开发模板，令多端处理逻辑的开发过程趋向标准化。

2）帮助开发者在编写接口时通过流程的横向拆分和数据组件的模块化提高代码可维护性。

3）通过框架模板的约束，促使开发者在系统设计之初就考虑如何处理一致性与差异性。

具体来说，BFF 层作为前后端中间层主要的作用是调用下游微服务接口并进行请求参数的处理并最终组装出视图模型数据返回给前端，典型的模式是：



当前端请求到达 BFF 之后，BFF 会解析参数并做出数据结构转换来向下游微服务发起请求，获得返回后再重复这一过程。

面对这样的调用链路，非常容易写成面向过程的逻辑代码。通过一个个工具函数不停对入参

/出参进行处理并传递给后一个下游服务接口。

整个程序控制流通过工具函数及函数传参被耦合在一起，这种模式在”一码一端“下由单一团队维护尚能接受，同一批开发人员确保自己端的 BFF 链路不出错即可。

但当多个端团队共同维护"一码多端"BFF 应用时，针对同一个 BFF 接口服务的不同特性被耦合在一起，将导致团队协作的困境：

1）下游传参不一致：

由于下游领域服务针对各端可能存在特殊逻辑配置，因此给下游的传参因端而异。

这一点通过 IF/ELSE/SWITCH 在 BFF 层面控制差异尚且能够接受，但多个端的 BFF 开发人员需要共同修改同一段分枝逻辑。

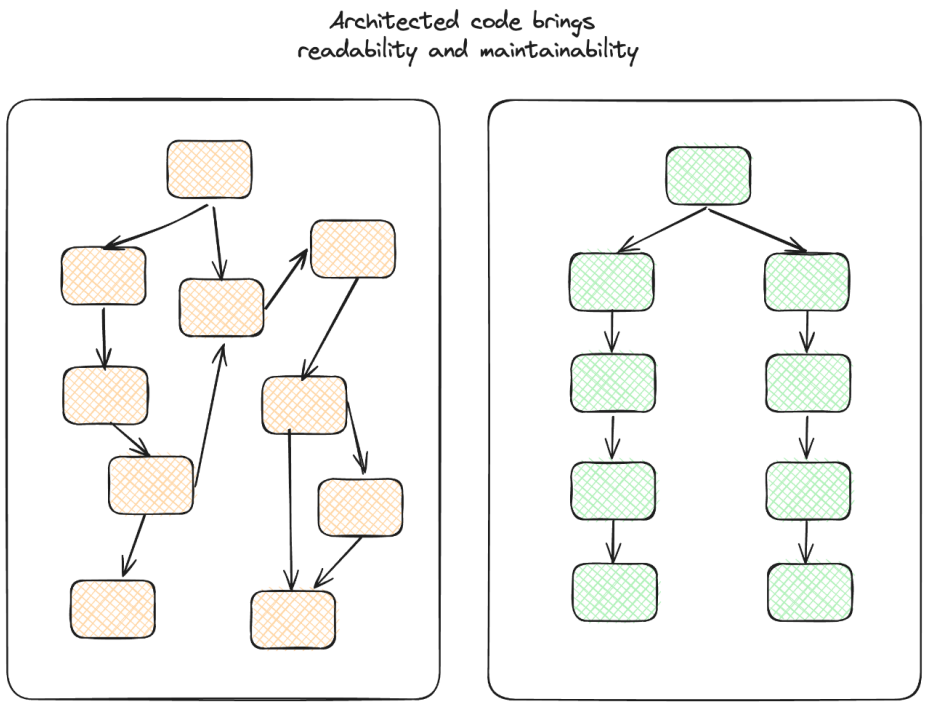
2）调用链路不一致：

各端会存在调用时序差异，例如端 A 某接口强依赖获取用户等级，而端 B 某接口仅依赖用户是否登陆，端 C 同时依赖用户登录态和用户等级。

这种程序控制流程的差异相比与参数的差异更难以处理，可能需要更大更复杂的代码块分枝处理来解决，并更容易引起冲突提高协作成本。

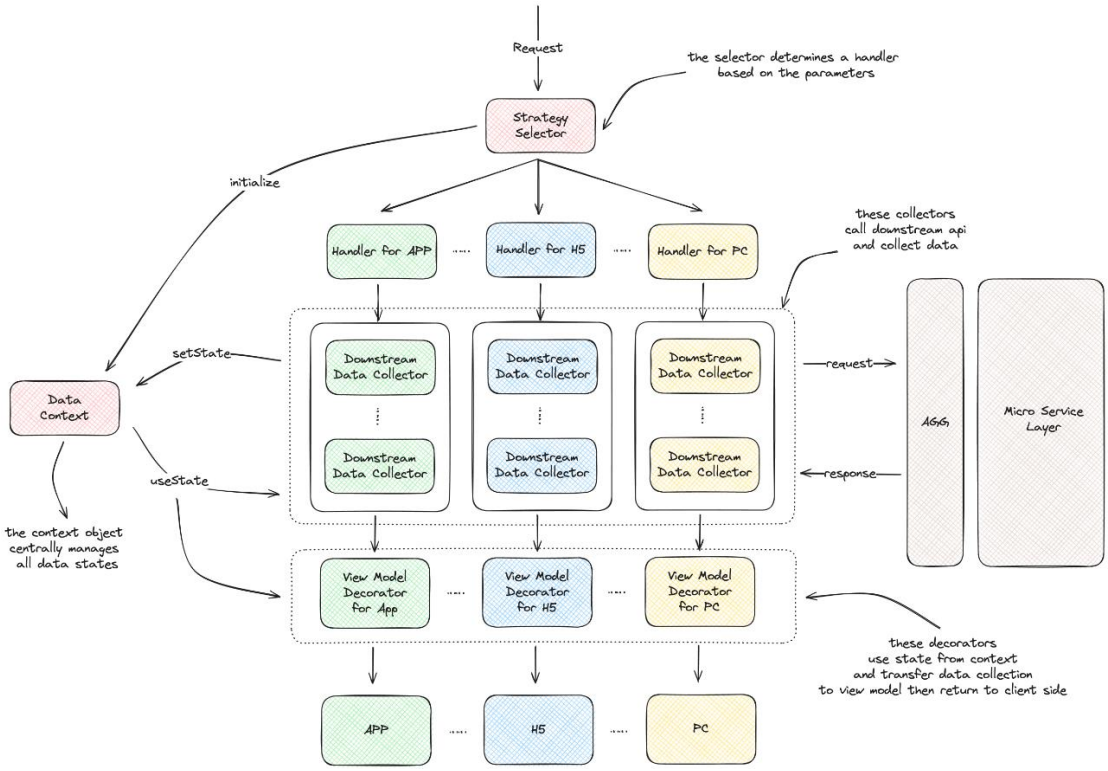
3）视图模型不一致：

由于各个端的 BFF 在不同的时间由不同的团队开发维护，各个版本 BFF 的接口契约难免存在差异，当“一码一端”合并为“一码多端”时，为了视图模型的一致性，必须将（1）（2）中的不一致在 BFF 层处理为一致的视图模型。

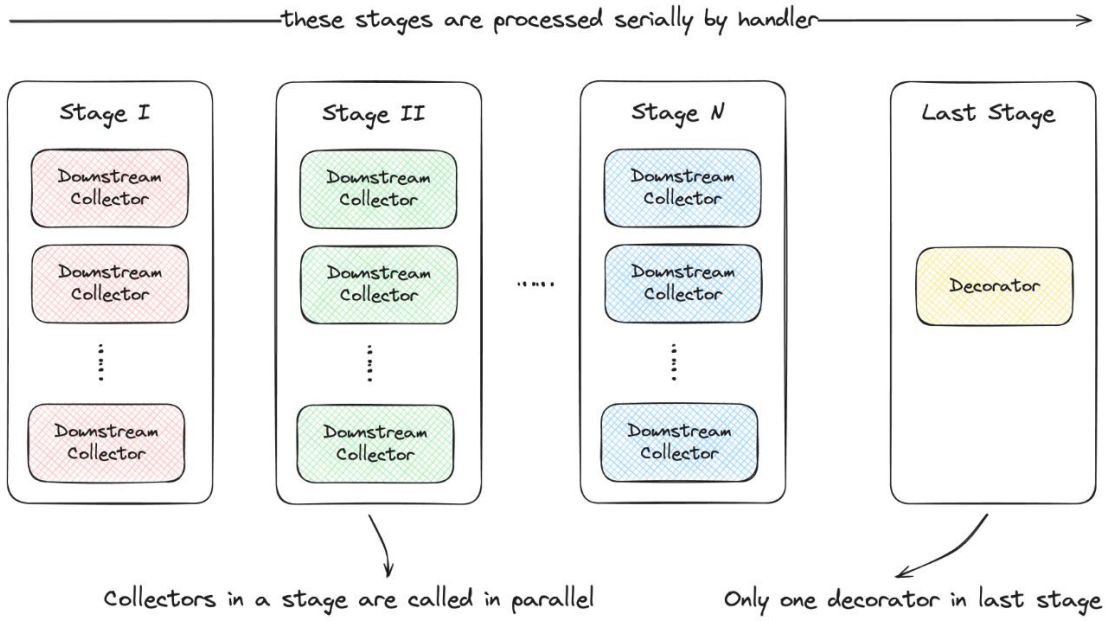


良好的代码组织结构和清晰的调用链路带来可读性和可维护性，降低迁移重构过程中的摩擦成本。

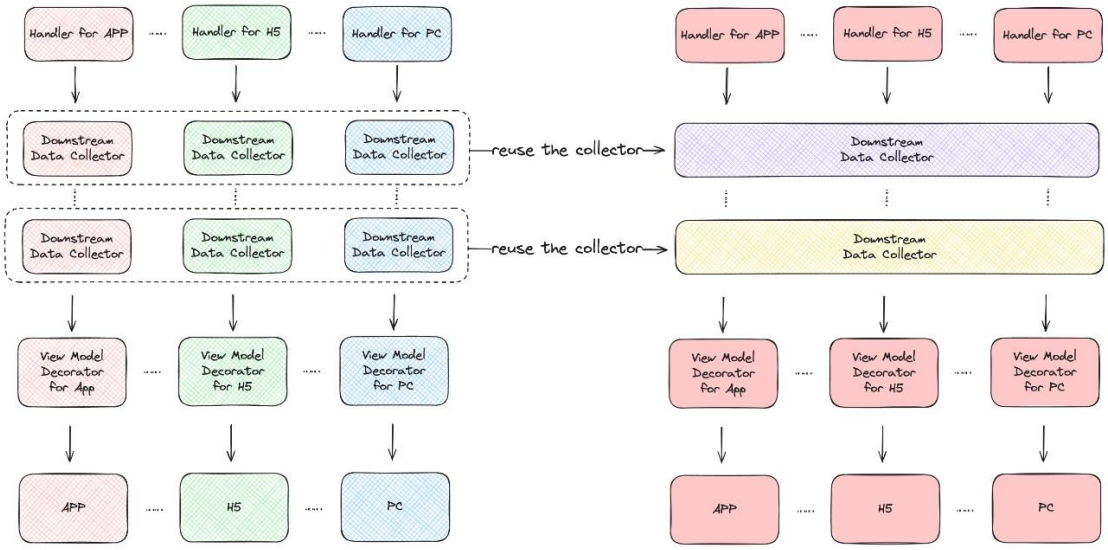
为了降低“一码多端”迁移过程中的协作成本，降低各端分支逻辑之间的耦合性，我们尝试提出一种 BFF 的多端开发模式：



通过多端策略模式将不同端的处理逻辑在接口内进行分流，并将一个接口逻辑内原本对下游的整体链路横向拆分为互不耦合的独立逻辑处理实例，让每一个实例仅需关注自身与下游服务的调用关系。且针对不同端的差异处理可以被文件级的分开，很大程度减少了冲突的发生。基于这个架构，多个端侧团队可以高效安全的进行协同开发。



多个调用下游的逻辑实例可以被并行执行，多个并行阶段可以被串行，数据流最终到达装饰器实例被处理后返回给前端。



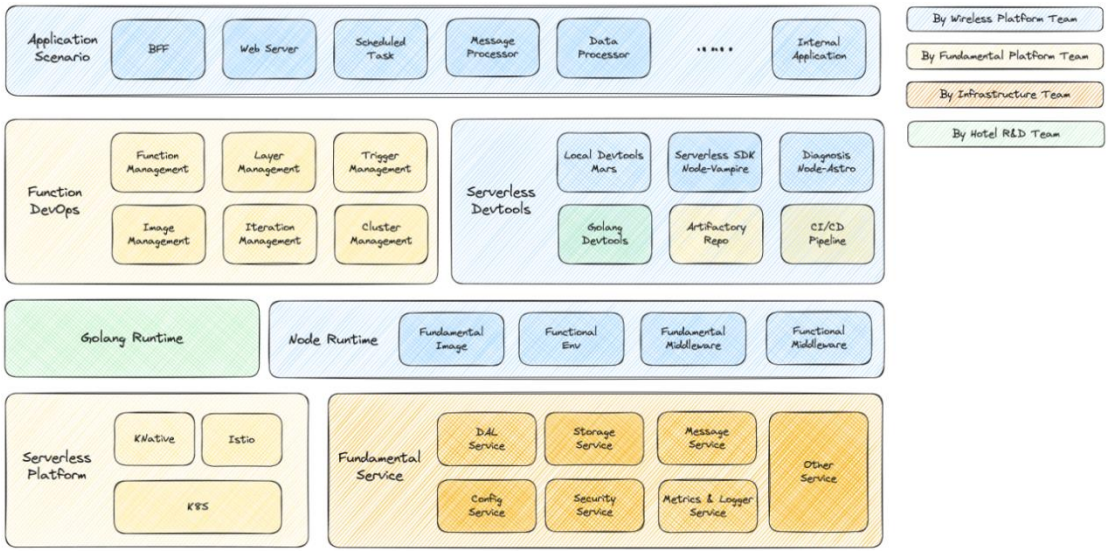
在携程酒店某二级页面 BFF 一码多端项目中，采用前文提到的多端架构模式对原本多套 BFF 应用进行了合并及重构，将原本分散的视图控制逻辑整合收口到了一个 BFF 应用内，大幅减少了后续的迭代维护成本，提升了研发效率。原本由多个 BFF 分别来支持多个端，通过多端开发模式重构之后，多个 BFF 合并成为一个，内部通过多端策略实例来支持各端。原本的过

程化代码被横向解耦并拆分成可复用的数据组件被这多个策略分别组合调用，最后再通过视图模型变化映射成统一的视图模型返回给各端。

在应用代码重构的同时，携程酒店 BFF 团队与携程公共平台团队合作，在他们的支持与帮助下实现了 BFF 的云函数化。

三、云函数平台

携程 Node.js 云函数平台从 2021 年正式立项已开始第四个年头。俱备轻量化的运行时，中间件能力及其他丰富的函数中间件接入能力。



轻量化运行时

云函数的轻量化主要体现在基础镜像、业务代码、发布运维三个方面：

云函数的基础镜像相比传统应用更加轻量化，在最新迭代的函数镜像中仅包含微型系统、js 运行时、函数运行时。相比此前的最小化镜像大小减少 59%，这有利于函数平台更快速的拉起函数镜像和实例。

云函数的代码结构相比传统应用更为简化，只需在定义 json 中定义函数入口。函数内置的运行时将启动一个监听服务器，将必要的请求负载交由函数入口文件运行，并将处理后的函数返回以响应负载的形式返回给客户端。

函数开发平台提供开箱即用的基础设施建设、管理与运维，帮助用户脱离繁冗的开发配置工作，只需关注业务代码逻辑的编写即可快速上线业务服务。

中间件能力

云函数运行时集成了 Tripcore 核心中间件集，并为云函数自动启用部分基础能力，由云函

数接管核心中间件的升级和维护，具有以下优势：

- 自动接入基础能力，无需复杂接入和配置
- 精选常用组件 SDK，无需安装，开箱即用
- 运行时内置，减少项目安装和构建时间
- 定期跟进维护组件升级，保障核心功能稳定性

函数中间件

由于函数代码结构和传统应用有较大差异，为了提升传统应用迁移到云函数的效率，框架团队设计了函数中间件机制，提供了将传统的 Web 应用快速接入到云函数的能力。

使用@ctrip/serverless-app 模块可以将常见的 Express、Koa、Nest.js、Egg.js、NFES、Remix、GraphQL 等 Web 框架的应用使用云函数进行部署，而无需对应用代码进行大刀阔斧的改动。

3.1 函数能力

触发器

云函数通过触发器提供给外部服务进行调用。在部署函数时即可生成函数 URL，用户可以直接使用函数 URL 访问云函数进行测试。此外云函数平台还提供了定时触发器、QMQ 消息队列触发器、SLB 触发器、SOA 触发器满足多种业务场景需求。

层管理

层提供了一种全新的管理依赖和静态文件的方法。通过将不经常变更的依赖库打包成层，可以减少部署镜像的大小，并加快代码的部署速度。目前在云函数平台上提供了 AlmaLinux 的常用运维工具层、Puppeteer 和 Playwright 层、FFCreator 层等，并提供相关能力的解决方案，帮助开发人员快速上线业务功能，而无需过分关注这些依赖库在 Linux 上运行的各种问题。

弹性扩缩

云函数产品支持快速弹性伸缩能力，能帮助业务提升资源利用率，在业务流量高峰时，业务的计算能力、容量自动扩容，承载更多的用户请求，而在业务流量下降时，所使用的资源也能同时收缩，避免资源浪费。

云函数平台支持基于 RPS 和并发度指标进行弹性扩缩，相比传统应用只能按照 QPM 指标和 CPU 指标进行分钟级扩容，云函数平台依靠底层流量监测组件，对流量变化的响应时间最快达到秒级，可以最大限度提升资源利用率。云函数平台支持最小可以将实例缩容到零，在请求到达时，云函数平台挂起请求，并实时拉起函数，待函数实例就绪后再将请求发送给函数

实例进行处理。这对于一些不需要一直运行的函数或者对响应时间不敏感的非业务核心应用来说，可以最大化节省资源。

版本和流量切换

云函数使用蓝绿部署帮助业务提升系统可用性。蓝绿部署时，并不停止掉老版本，而是直接部署一套新版本，等新版本运行起来后，再将流量切换到新版本上。使用蓝绿部署可以更平滑地进行流量切换。

云函数支持快速更改堡垒流量指向，便于开发测试人员在上线前进行堡垒验证。在正式发布前，通过堡垒测试工具验证和观察预发布版本的各项业务与性能指标。

云函数支持灰度发布，通过预先设置的灰度批次和流量比例，云函数可以实现无人值守的灰度发布，在监控到发布产生异常告警时可自动执行发布刹车并通知开发人员进行排查。

云函数支持多集群部署，通过部署在不同地域和机房实现容灾。在单个集群因演练、发布或其他原因导致故障时，可在函数平台进入流量切换页面更改流量指向。

微型实例

云函数平台的实例规格最小可以到 0.125C 和 128MB 内存，可以根据实际业务需求指定不同的实例规格。云函数鼓励通过更精细化的资源管理和快速弹性能力相结合来提升资源利用率。

函数场景

云函数提供了丰富的函数场景和代码模板，预先定义好的代码和流水线模板可以使开发人员在创建函数后等待数分钟即可在测试环境创建指定的场景函数。

目前提供的基础场景有 SOA 微服务、NFES Web 应用函数、QMQ 消息队列函数、定时函数等，此外还有酒店、度假、火车票等业务 BU 提供的定制化函数场景可供选择。

在云平台的函数场景能力支持下，前文提到的基于 NestJS 的多端 BFF 框架被标准化为云函数应用模板提供给其他 BU 开发者使用。

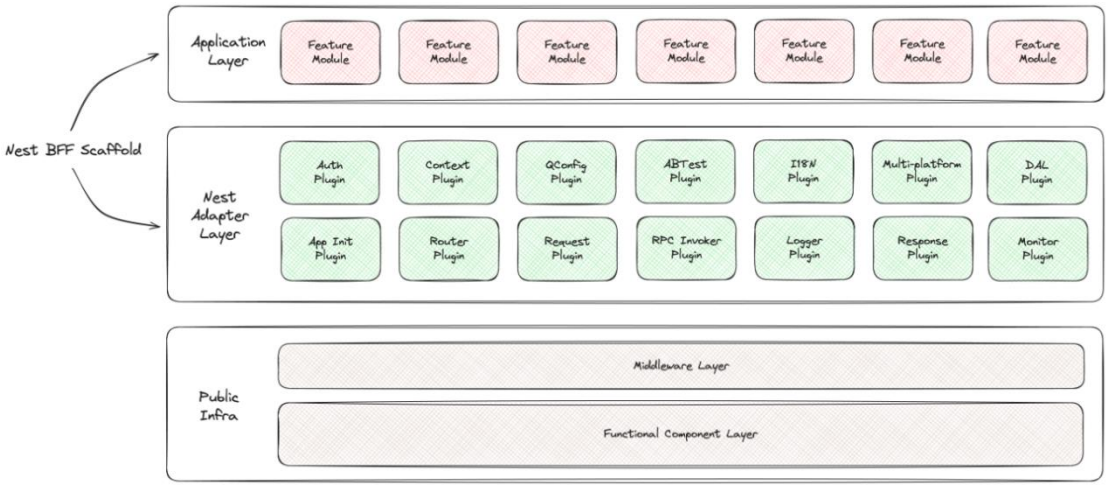
BFF 云函数模板框架采用了分层架构风格：

1）最底层为公司基建，包括 SOA、Clog、NodeJS 基础设施、存储模块等

2）在公司基建之上，通过 NestJS 模块化组织方式，封装常用 NPM PKG，PKG 可拓展，方便业务定制；支持 Template 和脚手架等提效工具，实现开箱即用

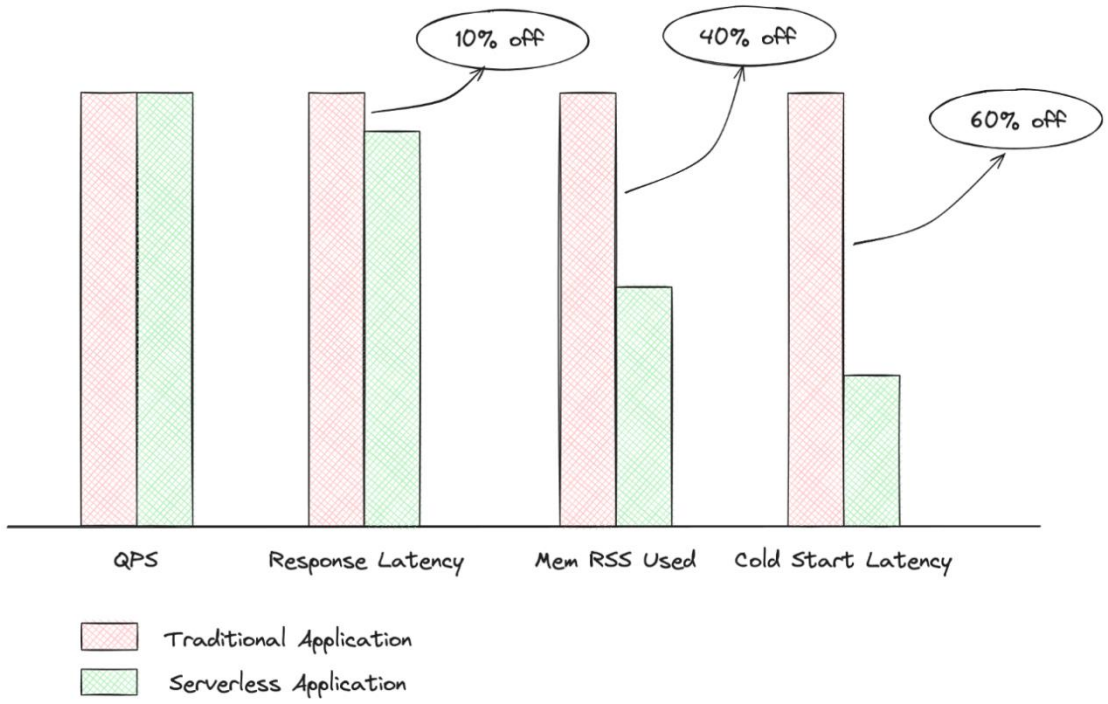
3）最上层为业务模块，使用公共模块开发各 BU 具体 API，实现各自业务逻辑；其中多端模块可支持各端共用一套功能接口

使用方在实现业务逻辑的时候可以做到开箱即用，无需重复造轮子对接公共基础设施。



前文提到的某二级页面 BFF 上云之后在性能方面获得了一定的提升，云函数的基础能力有效提高了 BFF 应用的资源利用率。

在相同 QPS 的运行条件下，云函数应用的响应时间，内存占用，冷启动时间都分别得到了不同程度的优化。



3.2 研发流程和函数生态

迭代管理

云函数迭代管理是具有探索性的一种研发流程一体化的实现方式，旨在为研发人员提供更便捷和灵活的研发流程体验。

研发流程可视化

将开发流程所涉及的各个节点，以流水线的方式展示给用户，使用户能够清楚地知晓整个开发流程所需要经历的节点。

详细的开发引导

在开发过程中，用户可以清晰地知晓当前进度。在开发过程的不同阶段，系统都会提供详细的指导，在遇到发布卡点时可以明确告知用户接下来应该做什么，用户只需按照指引一步一步操作，即可完成整个开发流程。

更专一的开发体验

创建 iDev 任务、创建分支、镜像关联 iDev 需求、分支合并等操作由系统接管，用户可以更加专注于研发工作本身。

运维监控和故障诊断

云函数平台提供的监控面板包含系统指标、Node.js 运行时指标等。开发人员可以实时观测到各个函数实例的基本情况，也可以通过 Web 控制台远程登录机器后进行调试。

云函数平台集成 Node.js Astro 监控面板，提供为 Node.js 增强的监控数据和能力：

- 实时监测到 js 应用的运行情况、点火报告
- 查询应用的 node_modules 依赖、Tripcore 依赖、层依赖等
- 支持镜像依赖和仓库提交比对，快速定位因依赖和代码变更引起的问题
- 支持实时的 CPU 性能分析和内存快照，快速排查性能和内存故障
- 支持在线断点调试，使用开发人员熟悉的调试工具连接到函数实例
- 支持离线调试，将函数镜像拉取到本地进行调试，定位发布环境的问题

Docker 化本地开发

由于 Node.js 函数内置了运行时和中间件，在本地开发时只能通过单元测试的形式测试业务逻辑。

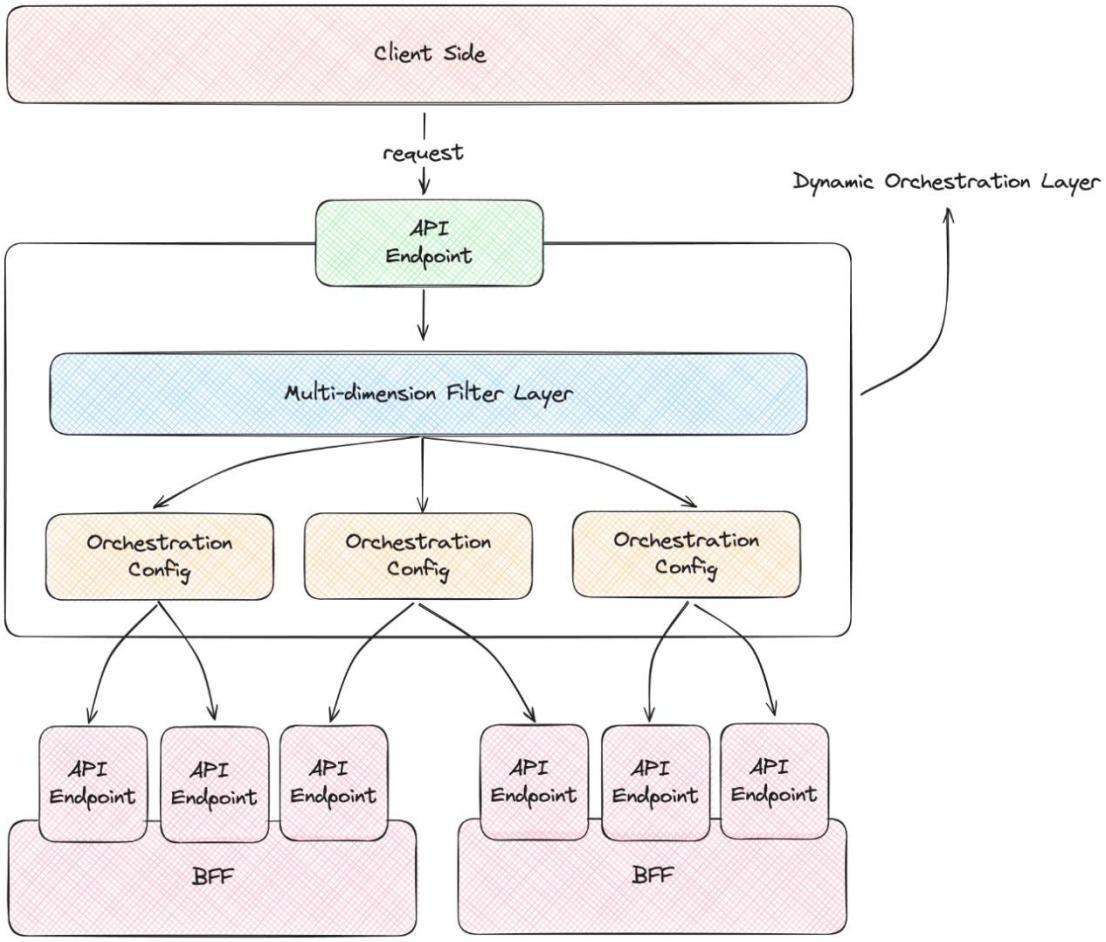
因此框架团队提供了 Mars – Docker 化本地开发工具，Mars 利用 Docker 容器能力在本地

完全模拟了函数实例真实在 CI 流水线和测试环境的运行情况，

使用 Mars 可以帮助开发人员更好地在本地进行云函数的开发调试工作。

四、前端动态化能力

在上述的 BFF 单体应用多端框架能力的基础上，大前端团队还在客户端和 BFF 微服务群中间设计了动态业务网关。如果说多端框架在应用内进行赋能，提供了支持多端 UI 差异的能力，这种差异我们可以称之为“接口内逻辑差异”。那么动态网关层则提供了处理“接口间组合差异”的能力。

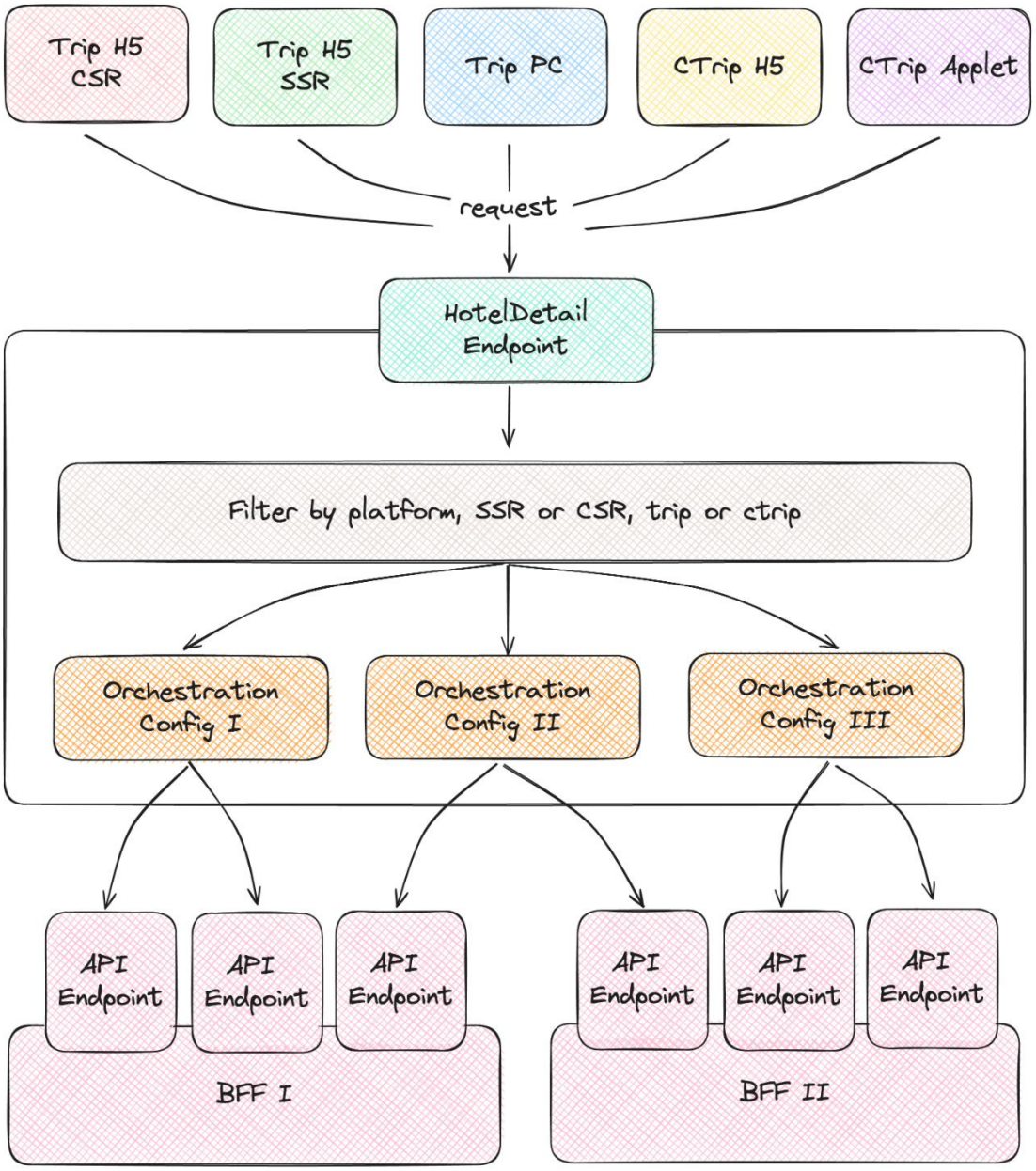


动态网关层支持：跨应用接口组装裁剪，白名单及灰度发布能力以及场景动态能力，支持针对不同参数维度组合场景来提供定制化的接口组装服务，增强接口服务的灵活性和适配性。

“多端模式” + “动态网关”的组合架构模式，分别在应用内，应用间 2 个层级来处理多端的视图控制逻辑差异，从架构维度再一次对多端的差异处理进行了解耦。

在携程酒店某一级页面技改项目中，为了能实现“一码多端”的技术验证目标，采用了”一码多端“+“动态网关”的架构方案。目前 C&T Web 侧共用一套 BFF 功能接口，通过在动态层对接口进行模块化组装来支持差异化的页面 UI 数据需求。

改造范围涉及 Ctrip H5、小程序、Trip Online、Trip H5(CSR/SSR)共 5 个终端，实现 17 个功能模块接口的改造，多端功能模块收口落地 BFF 层，实现多端一致和复用，提高研发能效。

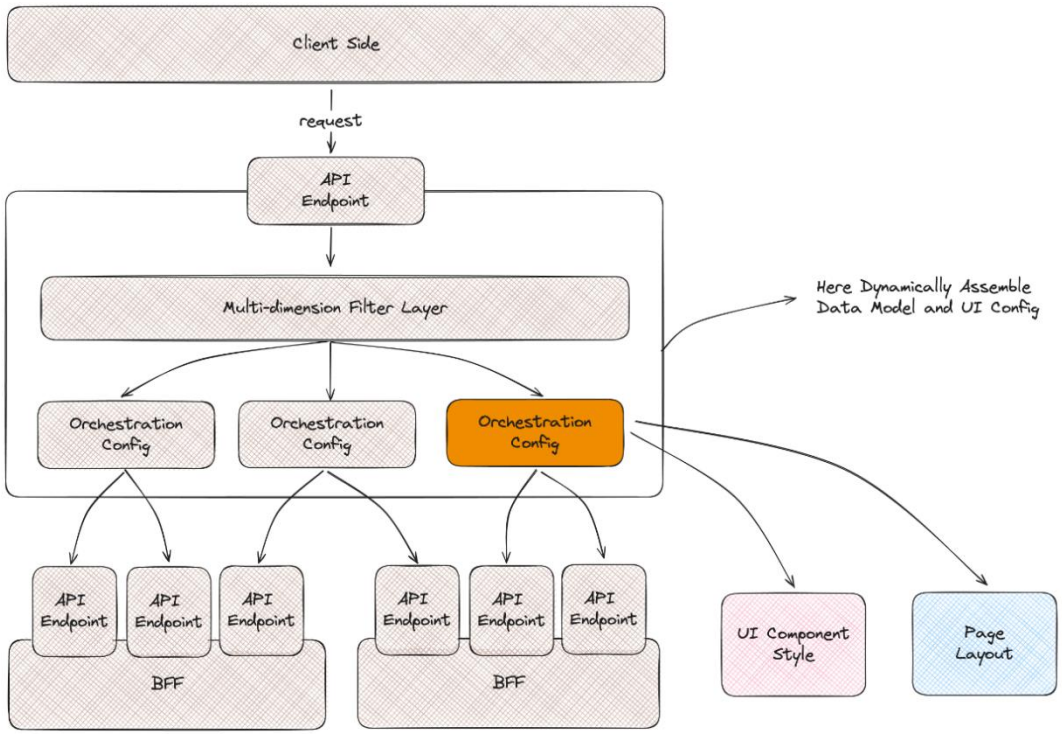


各平台流量在动态层入口处通过请求参数进行分流到不同的 BFF 编排逻辑中，动态层会根据编排配置访问 BFF 接口。

五、One More Thing

前文已经讲到了在“一码一端”到“一码多端”的前端能效变革背景下的 BFF 整体解决方案：应用内多端+动态网关+云函数能力。

这套方案除了能够用来支持将多个 BFF 合并为 1 个之外，还能够提供更强大的 UI 动态化能力：



结合酒店前端团队正在研发的跨端组件库，能够支持以极少的代码量支持多种用户场景，做到组件的跨场景复用，跨端复用。

代码复用率 99%，携程市场洞察平台 Donut 跨多端高性能技术实践

【作者简介】Brain，携程高级软件技术专家，关注前端技术框架、跨端技术方案、前端构建和工程化工具，致力于前沿技术和架构优化。

一、 背景

FlightAI（市场洞察平台）是携程机票大数据团队推出的一款赋能行业的 toB 数据产品。FlightAI 的用户群主要使用微信，但也有移动端和其他企业办公 IM 系统（如企业微信、钉钉等）的使用需求。由于大数据团队的前端工程师人数很少，且现有技术栈主要是 web 技术，缺乏移动端开发经验，因此需要选择一种以微信为主、研发成本低、跨平台（微信小程序、iOS、Android）应用的技术，以满足业务需求并提升研发效率。

FlightAI 涉及大型表格操作与展示、大数据的图表绘制、大数据量数据指标呈现等场景，对查询和展示渲染性能要求较高。此外，还需要在微信公众号、web 系统、小程序、移动端等多维度产品矩阵中实现业务功能、用户登录注册、账号管理与体系打通，形成立体式产品服务，以提升用户体验和运营效率。

1.1 难点与挑战

1.1.1 主要难点

a. 跨平台开发的复杂性：

虽然一些框架如 React Native，Flutter，Weex 等同构了 iOS、Android 的架构，但在独立完成移动端全流程开发和基础设施建设上，仍面临技术栈多样性的问题，特别是跨端架构中还需优先支持小程序。

需要掌握不同平台（iOS、Android、微信小程序）的证书管理、脚本命令调用、法律法规提示和约束差异、真机和模拟器开发、调试、构建、发布技术。

跨平台兼容性问题，如日历优化中遇到样式兼容性，性能问题等，需要深入理解各平台的差异和限制。

不同平台、不同流程下各种 Token、认证和法律法规限制概念较多，较难分清。

UI 组件在各平台的兼容性、生命周期管理和权限系统、SDK 稍有差异，一旦发生 bug 识别出这种差异较有难度。

b. 多端登录态打通：

需要解决不同平台间的登录态同步问题，确保用户在不同设备和平台上有一致的体验。

不同端的登录支持方式多样，需要做好同构和异构，如 APP 端需支持本机号码一键登录、账号登录、小程序登录、微信登录、苹果登录等，以支持各个平台的审核。

Token 种类非常多且复杂，各自用途不同，涉及前台与服务端 code 交换和 token 验证等逻辑。

c. 大数据列表渲染支持：

小程序双线程机制导致涉及交互的大数据列表渲染性能优化成为难题。

在小程序的技术架构下，官方支持最大 DOM 数是 16000 个，text 类节点也被计算在总节点数中，容易超过 DOM 数限制，导致 " Dom limit exceeded " 错误。

1.1.2 新技术平台的挑战

a. 功能覆盖率：

虽然官方号称跨多端，但功能覆盖率、API 覆盖率是否足够，接入是否复杂，能否满足当前和未来的需求？

b. 基础设施：

研发工具、测试工具、部署工具、运维工具是否成熟，遇到问题如何快速找到支持和解决方案？

c. 研发生态兼容性：

除了大的跨端技术框架面临挑战，落地实施会面临更多小型技术选型的挑战，如组件系统、图表库、其他平台支持、兼容性问题。

1.2 技术选型

1.2.1 业务需求与现状

a. 团队规模小，跨多端需求和高研发效率需求：

小型研发团队，现有技术技能主要是 JS 技术栈，缺乏移动端开发经验，必须控制培训成本和研发成本；

考虑集成难度和行业趋势，分析功能需求和性能需求等，FlightAI 用户有强烈的移动端使用需求，仅开发小程序无法满足；

需要选择一款技术复杂度低、研发成本低、技术栈简单的跨平台框架，而 Donut 技术满足跨多端、高性能、低成本需求，且有腾讯官方支持；

b. 业务需求：

建立独立账号体系、独立域名和品牌，要求与微信公众号、web 系统体系化运营，形成产品矩阵；

微信小程序在功能和性能上都要求高优先级支持；

c. 高性能需求：

Donut 在小程序支持度和 App 性能上表现优异。在 App 上底层基于 Flutter，使用和微信小程序相同的一套容器，接近原生渲染性能；

FlightAI 具有大批量数据实时渲染的需求，对渲染性能要求很高；与公司框架计划支持使用 Flutter 提升渲染性能的技术路线一致；

1.2.2 为何选型 Donut 技术

由于用户群体主要以微信小程序为主要使用场景，所以无法支持小程序的多端框架如 React Native，Flutter 等无法选用，而既支持小程序又支持移动端的技术框架，目前市面比较流行的是 taro 和 uni-app，但存在如下顾虑：

使用该类框架后，功能更新完全依赖于框架，微信小程序里有的功能可能无法支持；

使用该类框架后，代码经过二次转换，性能可能不如原生 WXML，而且会带入转换逻辑额外引入的代码和性能忧虑；

该类框架一般都会自定义 DSL，有一定的学习成本，文档是否健全，社区支持是否完善将直接影响配置和自定义功能的效率；

该类框架优势多在跨多端小程序，实际上线 APP 的案例并不多，大坑小坑无法避免，例如 uniapp 的 nvue 原生开发有局限性，特别是样式方面限制比较严重；

跨多端应用开发的全生命周期集成和管理能力是否强大，包括开发，测试，构建，发布，运维，法律法规提示与解读，token 管理等；

而 Donut 是腾讯官方推出的和小程序同源容器，在功能和性能上比这类衍生框架更优秀，在 App 上底层基于 Flutter，接近原生渲染性能；Donut 以小程序 DSL 作为开发语言，可以无缝切换，也没有二次转换的成本，会开发小程序就会写 Donut APP。在研发工具、测试工具、部署工具、运维工具上腾讯做了全方位集成，小型团队可能遇到的问题几乎全部涵盖。官方文档一步一截图，阅读体验良好，总结如下：

a. 功能覆盖率较高

常用需要适配 API（8 个），根据使用情况进行选择支持，Donut 官方提供了非常多的功能 API，从功能支持度、易用性和架构上 App 小程序同构上也可以看出官方的大力投入。

序号	功能名称	适配方案概述
1	•小程序登录（wx.login）	•可使用小程序登录（wx.getMiniProgramCode）代替 •或使用微信登录（wx.miniapp.login）代替
2	•获取手机号（getPhoneNumber）	•可使用一键获取本地号码（phoneOneClickLogin）代替
3	•微信支付（wx.requestPayment）	•可微信支付（wx.miniapp.requestPayment）代替
4	•微信分享（share）	•可微信分享（wx.miniapp.shareXXXX）代替
5	•隐私政策	•框架层提供原生的隐私政策弹窗，内容由开发者自定义实现
6	•Location 相关	•创建腾讯位置服务的 Key 即可使用原来的 jsapi
7	•消息推送（小程序订阅消息）	•创建腾讯云消息推送的 Key + 使用新的 jsapi
8	•流量主广告	•创建腾讯优量汇广告媒体 ID/ 广告位 ID + 使用原来的 jsapi

JSAPI 和组件支持汇总，涉及 6 大方面

组件部分：视图组建，表单，导航，媒体，地图，画布，开发能力，原生组建都能较好支持；

API 部分支持情况：总共大概 507 个 API；支持 371 个，大概占 73%，部分支持或者不支持的部分官方提供一些其他的替代方案；（2024-03 统计）

SDK 部分：基础 SDK 和扩展 SDK 支持较好；这部分是 Native 功能，根据需要进行勾选，不选就不会打入包中，按需打包降低包 size；

NativePlugin：支持 Native 自定义扩展，该部分都是一次性工作，除非有强烈的自定义需求，一般都用不上；

云支持：云开发云托管，Donut 网关防护；

埋点监控：热力图，回放功能支持完善，支持全埋点；

b. 基础设施较完善

Donut 平台覆盖开发、部署、产品体验分析全产品开发周期的各种需求，研发工具、测试工具、部署工具、运维工具上腾讯做了全方位集成，基础设施比较完善，贯穿产研全流程支持，可大幅提升研发效率，特别对于没有移动端研发经验的团队具有引导和指导的作用。

- 多端框架-支持使用小程序原生语法进行一次代码编写，多端编译，实现多端开发。
- 多端身份管理-几行代码，快速实现 App、小程序的身份认证和用户管理。
- 安全网关-提供弱网加速、防爬防刷、流量治理等能力，全方位保障业务安全高效稳定运行。
- 产品体验分析-从真实的用户视角洞察产品问题，寻找产品体验的不足之处，提升用户留存与转化，具有如下功能：

- 零代码、全埋点

无需开发，一键接入，元素自动全埋点，快速开始小程序数据分析。

- 还原用户操作现场

创新可视化日志 & 热力图，还原用户实际操作现场，问题分析一目了然。

- 可视化交互分析

全程无需 SQL 编写，仅需在模拟器上点选交互即可完成分析过程。

- 一键智能分析

无需数据分析背景，根据业务目标智能分析，查找用户漏点，提升转化率。

c. 研发生态兼容性较强

UI 组件系统

UI 组件系统选型 TDesign 还是使用 WeUI，还是使用 Ant Design for Mobile，NutUI，Vant，Material-UI?

主要考虑组件的成熟度，兼容性，组件功能，TDesign 是腾讯官方出品的组件库，而 WeUI 则侧重提供样式库，考虑社区实践经验，最终选择 TDesign，组件相对丰富，常用组件齐全，贴合微信设计规范，在 Donut APP 的兼容性较好，复用组件可以节省不少开发工作量。

图表库选择

图表库使用 echart 还是 g2 移动版图表库，还是选择 wx-chart，ucharts，D3，antV，Threejs 等图表库？从如下几个方面进行了考虑：代码规范：EChart、D3 官网的部分案例还是使用了 ES5 的语法，Antd 遵循了 ES6 新语法规范。灵活度：ECharts < G2 < D3；使用难度：Echart≈G2PLot < G2 < D3；场景：三维图推荐用 Threejs，二维图用 ECharts 或者 G2、G2Plot 均可；考虑到 FlightAI 图表类型为常规图形，而且 PC 版本选型也是 Echart，在小程序集成和兼容性上，Echart 都表现较好，但 Echart 在分辨率处理上和 Donut 系统并不兼容，但在编写适配层解决该难题后，用起来非常丝滑，所以最终选定 Echart。

小程序兼容性

小程序转其他小程序或者 web 的方案调研如 morjs 等转出来的小程序，其他跨多端如 Taro 转 Donut 技术是否可行?

理论上 Donut 提供的是小程序运行容器，其他跨端或者转换技术只要最终产物符合微信小程序规范，应该可以运行，但是转换必然导致功能和性能上有损耗，而且对于 Donut 条件编译等语法需要进行特殊处理，一般转换代码会带来额外的处理逻辑，会增大包 Size，虽然有副作用，但 Donut 本身的扩展能力还是很强的，毕竟提供了和微信一致的小程序容器。

1.3 Donut 简介

Donut 平台是腾讯出品的基于小程序实现跨多端（小程序，IOS，Andord）的技术体系，覆盖了开发、部署、产品体验分析全产品开发周期的各种需求。开发者可以专注于代码逻辑，将复杂的开发构建流程，开发及运行环境等统一管理，有效降低多端应用开发的技术门槛和研发成本，以及提升开发效率和开发体验。包含 4 大特色能力：多端框架，多端身份管理，安全网关，产品体验分析。

1.3.1 Donut 系统模块图:

主要分为客户端和云端功能，客户端主要负责端侧的标注化容器，处理小程序基础库，小程序 SDK 等基础设施，云侧主要处理各种管理和审核功能，流程都集成在了微信开放平台，微信开发者平台，微信公众平台，Donut 管理平台几大平台上。

1.3.2 Donut 工作流程

简单理解，就是微信抽象了一个简化版本的微信 APP 作为容器，底层都是基于 Flutter 进行渲染，承接小程序的能力，并利用微信开发者工具提供研发支持；理论上只要是微信小程序都能运行在该平台上。

1.3.3 Donut 技术适合谁

a. 基于小程序生态开展业务的个人或企业；

b. 需要跨多端支持、高性能和高研发效率的项目；

c. 已经拥有小程序，想扩展到移动端的项目；

d. 希望简化或标准化开发运营流程，利用微信提供的全软件研发周期集成能力的项目。

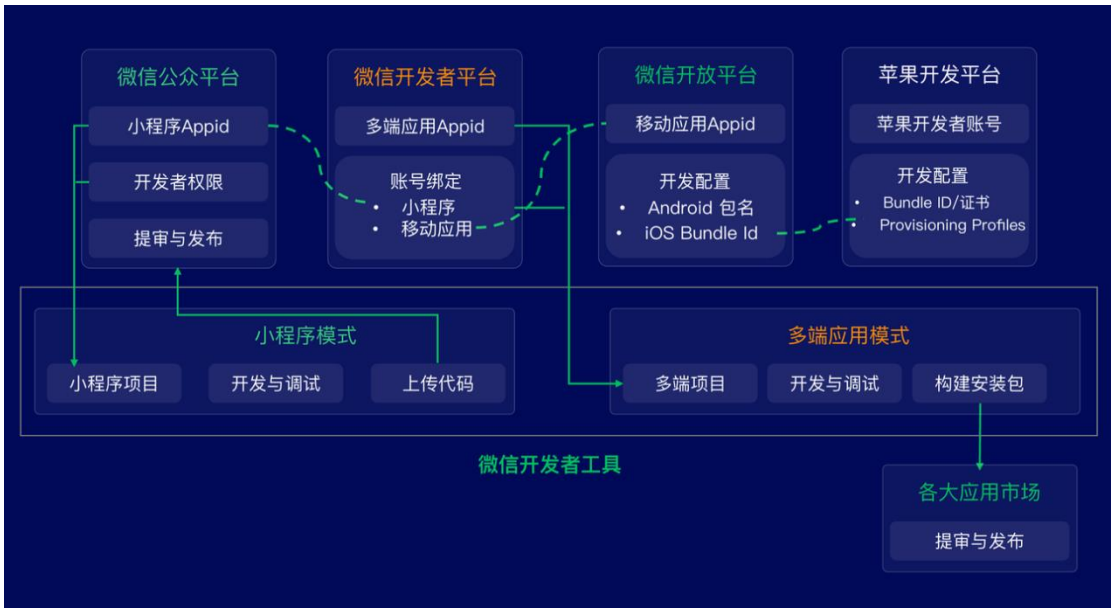
二、FlightAI 如何基于小程序构建一款跨多端移动应用 Donut-APP

首先在微信公众平台（mp.weixin.qq.com）注册小程序，完成注册后可同步进行信息完善和开发。下载微信开发者工具、参考开发文档进行开发和调试。要成为 Donut 跨多端研发人员需要了解如下信息：

2.1 多端应用开发：开发账号准备

序号	账号名称	备注
1	•小程序 Appid（建议企业主体）	•需确保已有开发者权限或者管理员权限
2	•多端应用 Appid（需与小程序同主体）	•可在微信开发者工具-切换为多端应用模式时创建 •可前往微信开发者平台创建并绑定小程序
3	•移动应用 Appid（需与小程序同主体）	•需先创建微信开放平台账号（通常已有，需与小程序同主体） •再在微信开放平台里创建移动应用账号（通常已有）
4	•苹果开发者账号 •或者苹果个人账号	•如只通过数据线将 IPA 安装到手机，则只需要苹果个人账号 •如需分发内测或者需调试微信 OpenSDK 能力，则需开发者账号

2.2 了解账号关系



2.3 多端应用开发：开发资源准备

序号	资源名称	备注
1	•笔记本（ Mac 或 Windows）	•如开发 iOS 应用，建议使用 Mac
2	•微信开发者工具	•需下载安装最新的 nightly 版开发者工具
3	•Android 模拟器（可选）	•通过安装 Android Studio 方式进行安装模拟器
4	•iOS 模拟器（可选）	•通过安装 Xcode 方式进行安装模拟器
5	•手机（ iOS 或 Android）	•用于真机体验
6	•数据线（ iOS 或 Android）	•用于与微信开发者工具将手机进行连接

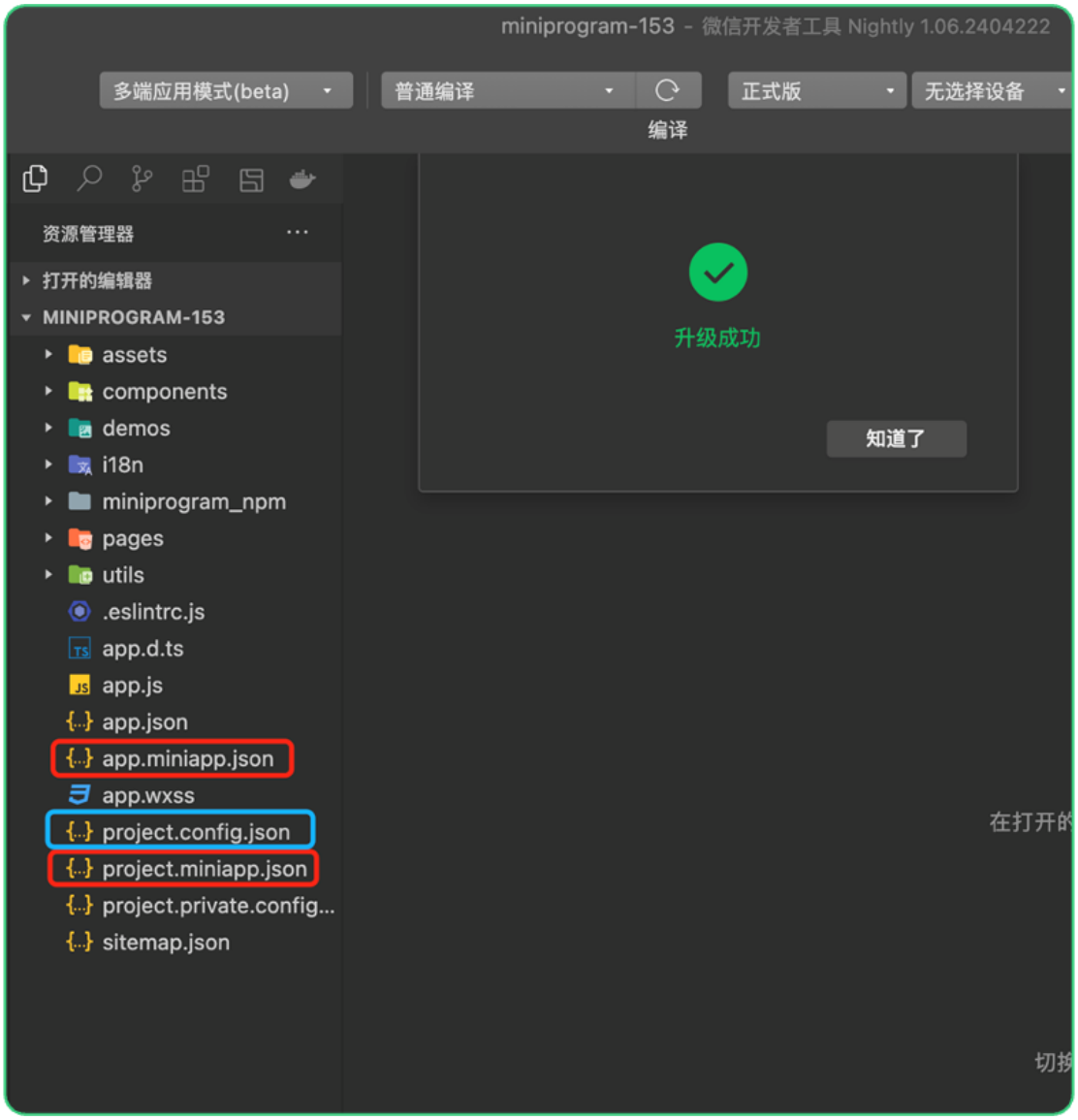
2.4 开启多端应用模式

a. 新建或者升级小程序为多端项目即可

b. Donut 多端项目结构

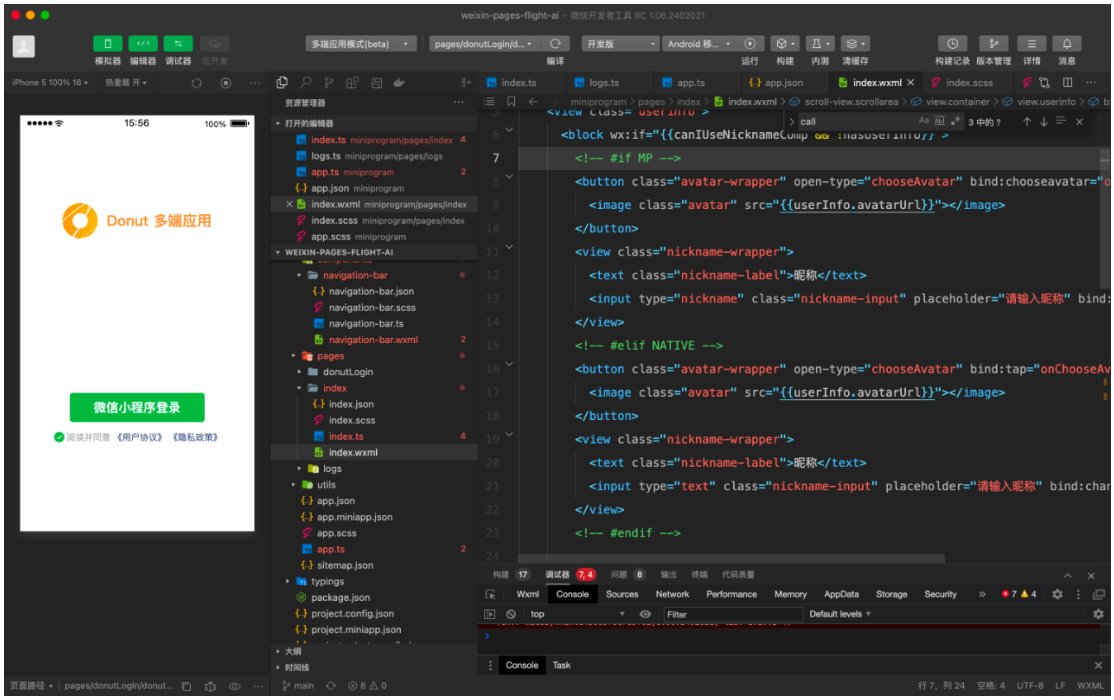
和小程序项目结构一致，只是多了一些跨多端框架的配置文件，app.miniapp.json 主要作用是维护多端应用和小程序绑定关系，配置 App 小程序登录页面地址，授权页等。

project.miniapp.json 则主要用于配制原生插件，多和 native 交互有关；project.config.json 则是小程序相关的配置文件。



2.5 了解条件编译

Donut 支持以条件编译的方式编写特定平台代码；如有些组件只有微信支持，有些业务需求只在微信展示，那么就需要条件编译实现差别需求，在 app 上和小程序使用不同的代码方式实现，一般该类代码占比较少，FlightAI 项目中，我实现了自动统计条件编译代码的功能，可统计上报或者单机运行了解项目状态。

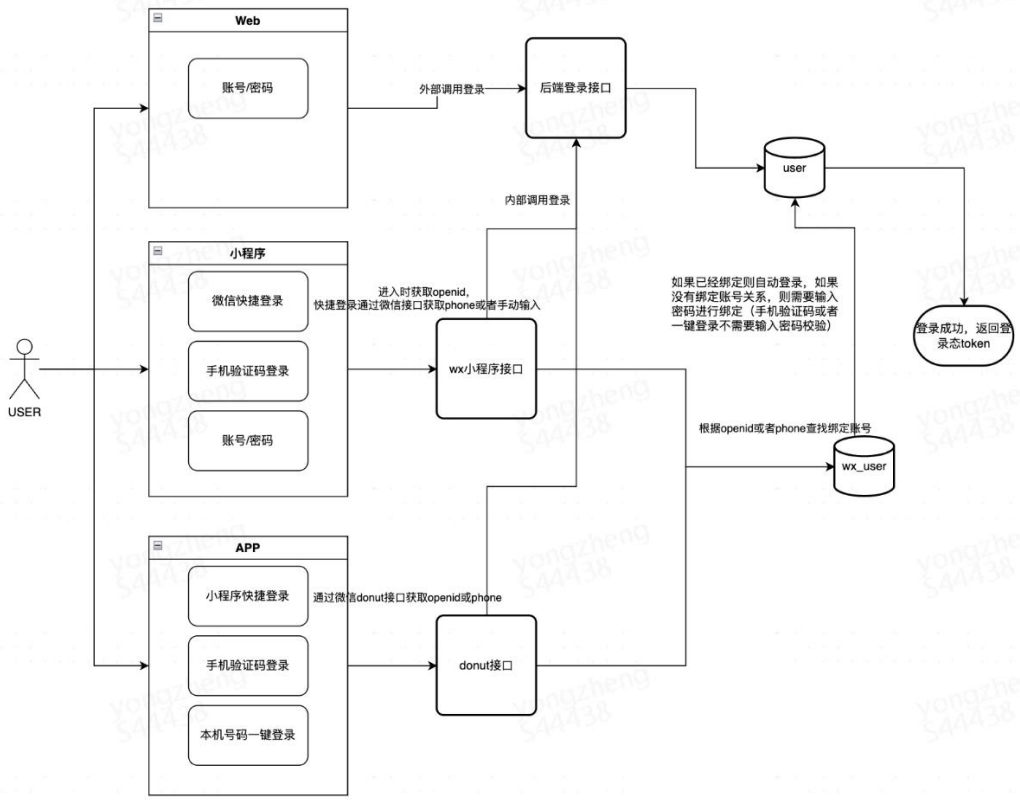


2.6 多种登录方式支持-多端身份管理

应用场景	建议方案	原因
•快速体验	•使用「小程序登录」：wx.getMiniProgramCode	•适配成本更低
•正式上线	•使用「微信登录」：wx.miniapp.login	•更符合 App 用户登录习惯
•正式上线	•使用「一键获取手机号」：phoneOneClickLogin	•登录更便捷，且可获取手机号信息
•正式上线	•使用「手机验证码登录」：sendPhoneSms	•体验不如上，但对接简单，且方便给审核人员测试账号进行测试
•正式上线	•使用「手机验证码登录」：sendPhoneSms	•iOS 需要支持苹果登录

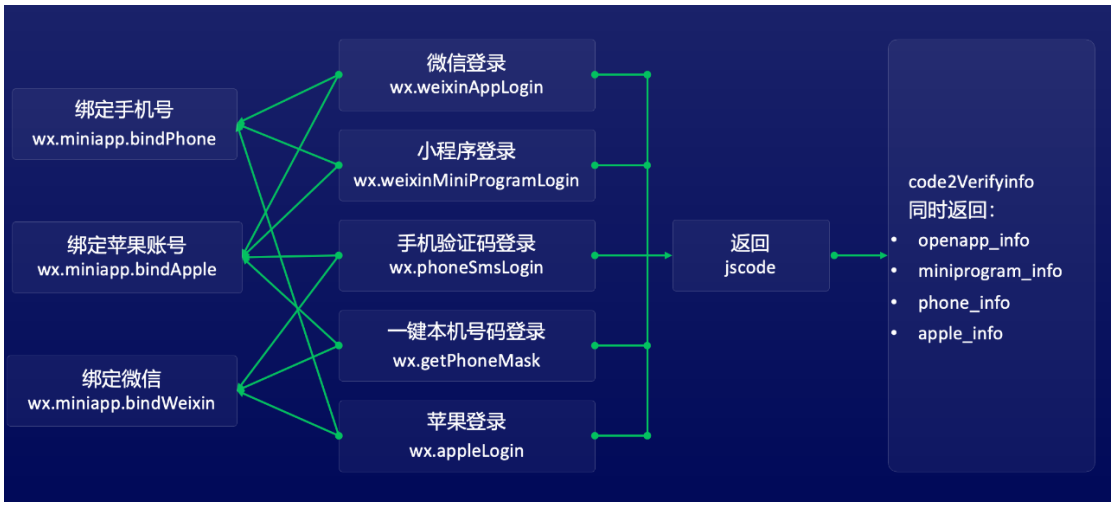
a.FlightAI 登录架构设计

Donut 支持多种登录方式，一般项目完全够用了，并在后台做了较好的集成管理，官方叫做多端身份管理，FlightAI 的场景比较复杂，除了 Donut 提供的 5 登录方式之外，还支持了密码登录等总共 10 种登录方式，给客户最大的便利和可选择性；支持方式多样，但是底层验权，鉴权，授权等接口都实现了统一。



b.小程序和 App 的统一身份识别

小程序的 token 和 code2Session 接口进行交换，App 的 token 和 code2Verifyinfo 接口进行交换，各个接口都有配套 API，比较复杂，不能搞混了，最好单独一整套接完，再接入另外一套。



2.7 大列表渲染

在 webview 的双线程架构下，小程序的官方支持最大 Dom 数是 16000 个，text 类节点也被计算在总节点数中，非常容易就超过 Dom 数限制，导致 Dom limit exceeded， please check if there's any mistake you've made.

无论是模拟器还是真机，小程序还是 Donut APP，在 FlightAI 场景中 view-all 页面，经过测试 442 条渲染 Item 是临界点，超过 442 条就会白屏，在该数据结构下 Dom 数会达到临界点，并且 442 条数据在 IOS 模拟器上渲染耗时 5646ms，这也侧面证实了各端各工具在 Dom 数限制上是一致的。

官方推荐标准

官方推荐说明（也是评分标准），单页面节点尽量不超过 1000 个节点，嵌套不超过 30 层，子节点不超过 60 个，页面深度不超过 10 个；但功能较为复杂的页面，非常容易就超出限制，那么如何突破这个问题呢？

长列表渲染特点

1）列表数据很大，首次 setData 的时候耗时高，双线程模型使得交互性能成为瓶颈；

2）一次渲染出来的列表 DOM 结点多，每次 setData 都需要创建新的虚拟树、和旧树 diff 操作耗时都比较高；

3）渲染出来的列表 DOM 结点总数多，占用的内存高，造成页面被系统回收的概率变大。

优化思路

要么彻底改变双线程的底层架构，要么通过一些手段优化参与渲染的数据和状态，只渲染显示在屏幕的数据，基本实现就是监听 scroll 事件，并且重新计算需要渲染的数据，不需要渲染的数据留一个空的 div 占位元素。如下是 2 个解决方案：

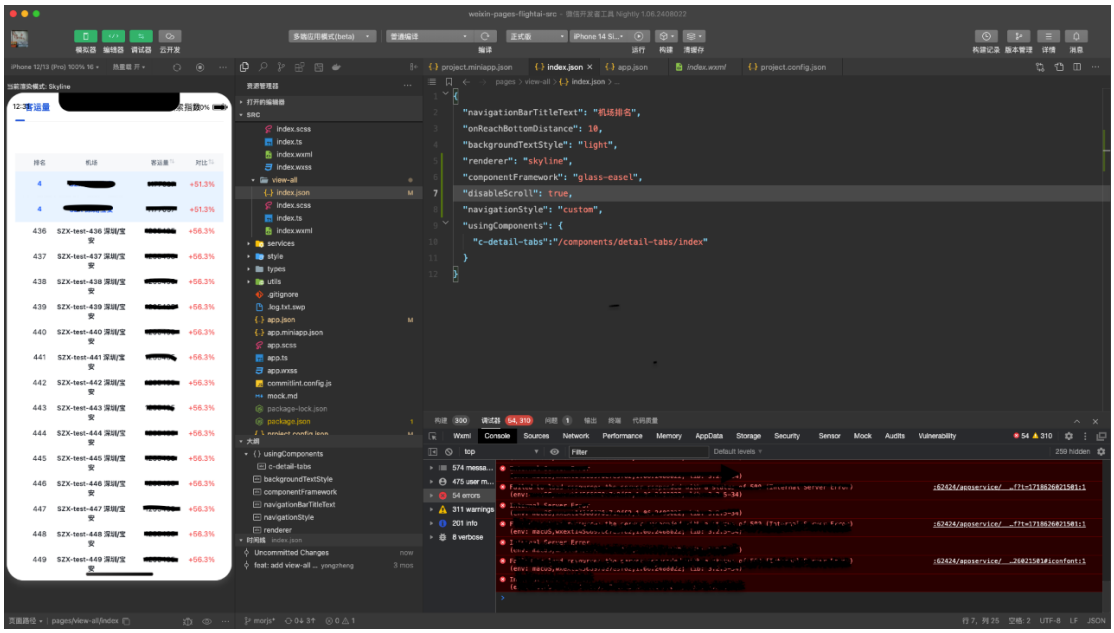
a. 微信官方提供 recycle-view 的解决方案，虚拟列表方案；

在滚动过程中，重新渲染数据的同时，需要设置当前数据的前后的 div 占位元素高度，同时是指在同一个渲染周期内。页面渲染是通过 setData 触发的，列表数据和 div 占位高度在 2 个组件内进行 setData 的，为了把这 2 个 setData 放在同一个渲染周期，用了一个 hack 方法，所以定义 recycle-view 的 batch 属性固定为 batch=" {{batchSetRecycleData}} " 。

b.微信官方为提升渲染速度，开发了 skyline 渲染引擎。

skyline 渲染引擎，使用更精简高效的渲染管线，并带来诸多增强特性，让 Skyline 拥有更接近原生渲染的性能体验。

使用 skyline 之后，不会有 dom 数限制了；而且很明显的一个对比是使用 skyline 后，快速滑动长列表不卡顿，首页没有开启 skyline 快速滑动会卡顿。



注意列表布局容器，仅支持作为 scroll-view 自定义模式下的直接子节点或组件直接子节点，scroll-view 要设置自定义模式 type=" custom " , list-view 要作为 scroll-view 直接子节点。

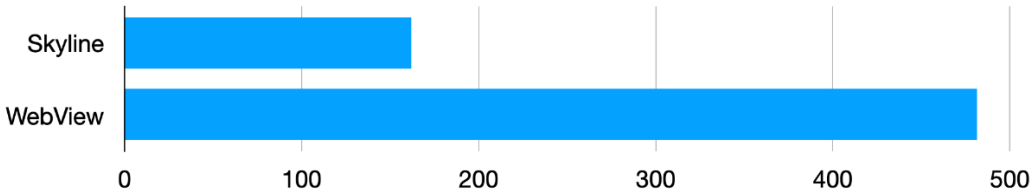
WebView 的 JS 逻辑、DOM 树创建、CSS 解析、样式计算、Layout、Paint （Composite）都发生在同一线程，在 WebView 上执行过多的 JS 逻辑可能阻塞渲染，导致界面卡顿。

Skyline 创建了一条渲染线程来负责 Layout， Paint 和 Composite 等渲染任务，并在 AppService 中划出一个独立的上下文，来运行之前 WebView 承担的 JS 逻辑、DOM 树创建等逻辑，架构深度优化后性能提升显著。

Skyline 长列表官方文档

性能对比

官方示例：小程序助手的线上数据，可以看出 Skyline 的首屏时间比 WebView 快 66%，并且手机性能越低端，差异就越明显。



另外尝试自行计算小程序的 Dom 数，Dom 层级，做统计或者优化，发现 querySeletorAll API 无法使用通配符，另外涉及到 Shadow-Dom，计算变得非常麻烦；向微信官方求证过 Dom 最大 size 的数量是 16000 个，这是编译器层面的限制，无法放开。经过和微信官方沟通结果是他们可以提供小程序的 Dom 数计算 API，目前还没有提供，将来可以提供。

2.8 实现 Push 消息推送

FlightAI 接入 TPNS 或者公司的 Push 系统的一些思考和调研；腾讯云有提供 TPNS，但是这套推送接入成本较高，而且已停止售卖；收费标准大概是 800*DAU/10000，8 分钱/条，按用户收费。

Donut 提供了基于 IM 的新版本推送服务，2024-8 月份开始支持；通过配置插件和证书实现推送功能，支持在线推送和离线推送；IOS 和 Android 的各个厂商需要分别进行配置，这是厂商的规范不同导致。

新版本收费规则根据数据中心位置和不同的套餐有所不同。后台需要根据各厂商申请对应的证书 AppKey 和 AppID，并根据需求制定推送策略。

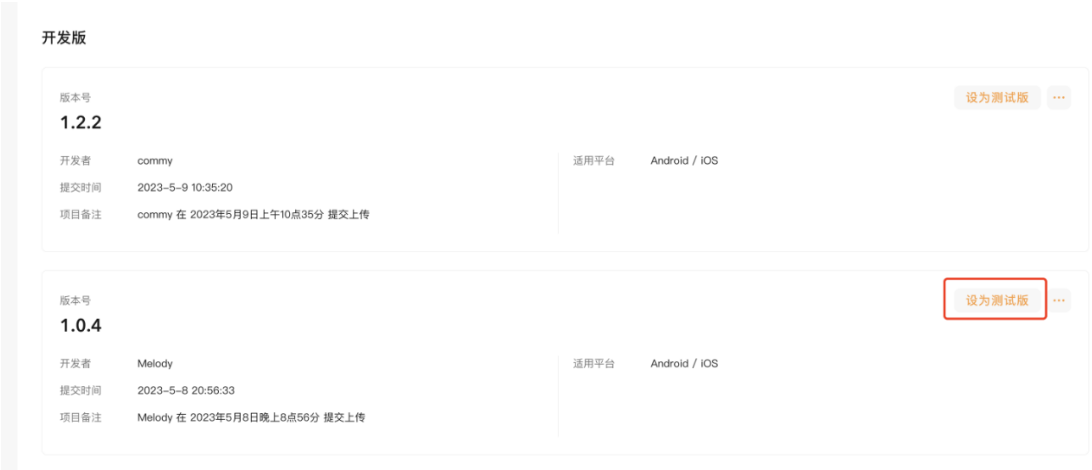
对于接入成本的考虑，在客户端只需简单配置 PluginId 即可，配置示例：

2.9 正式构建 APK 和 IPA

构建移动端 APP 的临时签名和正式签名是区分的，临时签名 Donut 官方直接提供，正式签名证书，需要研发人员按照目标平台进行注册，上传，注意签名文件对目录有要求，必须放在项目内，IOS 的钥匙串必须和配置中的名称完全一致。

2.10 内测分发与提交审核

在开发者工具上 选择 正式版点击「构建」- 「上传资源包」，即可将开发版本的资源包上传至 Donut 资源包管理平台，之后跟进研发流程自行决定测试版本和线上版本。



2.11 完成各平台隐私协议，法律条款，授权弹窗等上架要求，发布 APP

需要注意的是各大电子市场的要求各不相同，为了能够上架，需要满足各平台的各种规范，否则审核无法通过，特别是涉及法律条款需要法务部门介入，通常时间非常长，要做好相关准备。

2.12 踩了哪些坑

一些证书导致的问题

在进行基础设施建设中，完成小程序，Donut 跨多端基建，包括环境搭建、系统配置、开发、测试和发布流程的打通， 技术框架搭建，架构设计模型，开发流水线等。比如证书位置不对，证书信息不匹配，申请证书的 Domain，link 等信息不一致，在构建的时候可能会报一些奇怪的错误，始终无法通过构建，这块的研发信息提示很不明确，但会 Block 研发进程，已建议官方优化；

构建产物，虽然功能增多，业务逻辑变的越来越复杂，包 size 会正常增长，但是如果有自定义管理资源包的需求，那么就会遇到包 size 上传限制的问题，那么就需要分包，比如我们 nephle 最大支持 30M 的包，需要解决分块上传问题；另外如果要实现 ios 自动下载安装的能力，模版地址和包地址需要放在同一目录，否则坑较深，会浪费不少精力和时间；另外真机调试和构建，直连的数据线非常重要，带转换的多功能数据线很有可能带来不少意外的构建问题；

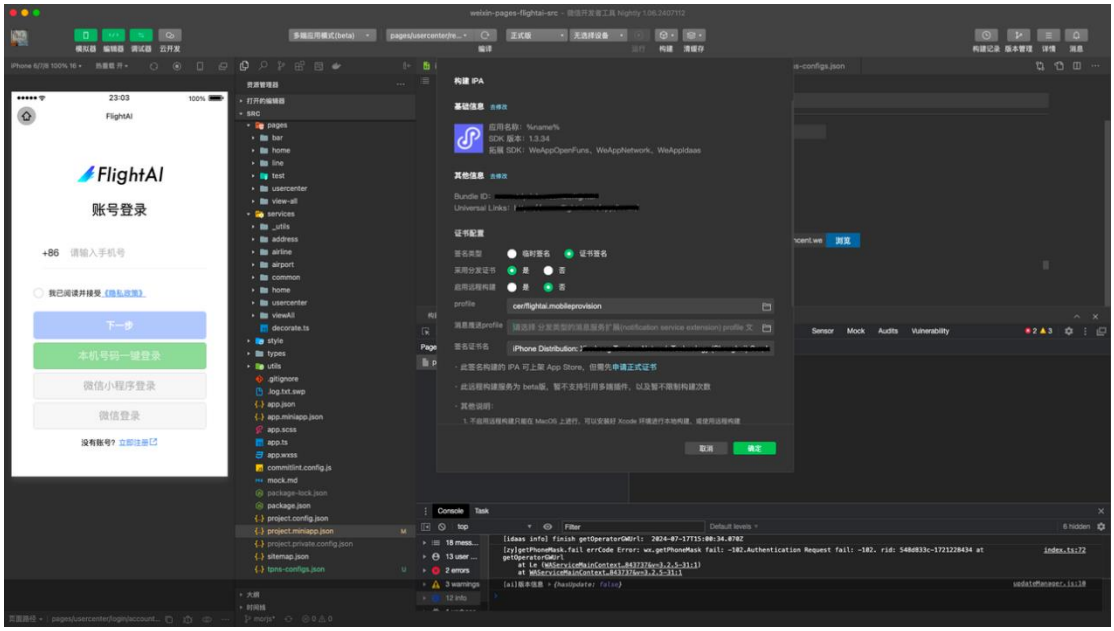
功能开发与性能优化：图表功能开发，日历功能开发与性能优化，TDesign 官方日历组件存在性能问题，官方提供的 format 钩子，在遇到多年份日期遍历的时候，性能问题突出，t-calendar 并且不支持虚拟列表，所以性能不理想，采用 Hash 算法后优化了近 400 倍，提升日历功能的性能，另外 TDesign 和 Donut 是不同的团队，提 issue 的时候要找准团队，否则可能不被受理。

小程序备案，域名备案，微信认证，用户告知，服务条款，隐私协议，软著商标，何时启动？有哪些坑？

法务审核和各种认证需要准备很多企业实体资料，尽早准备提交，完全按照文档操作，否则会被打回重写，再者审核周期不可控，应当尽早启动，隐私弹窗和授权弹窗必须使用模版或者 native 开发，不能使用其他方式，因为 APP 在授权之前是不能加载和运行代码的。请仔细阅读相关章节注意事项；

token 管理

token 是测试一套，生产一套，配置不同，Android 和 IOS 平台各自管理一套，另外 mobileprovision 必须包含 Doman，而且 Donut 和开发工具，APP 中的信息必须一致；



三、成果与经验

3.1 技术创新与效率提升，代码复用率 99%

代码复用率：实现了 99%的代码复用率，特定平台代码代码为非复用代码，自行实现了自动统计的功能；

成功研发一款跨平台高性能 APP：成功开发了业务代码基于小程序，底层框架与微信同源基于 Flutter 的高性能跨多端移动应用，确保 iOS、Android 和微信小程序上的一致性和高性能，为公司探索新跨多端技术方案；搭建了 iOS 和 Android 应用的自动化构建和持续集成（CI/CD）管道，提升了开发和发布效率；

采用 Skyline 新渲染引擎， 首屏时间比 WebView 快 66%，更重要的是 Skyline 没有 Dom 数限制，对于开发大型项目来说非常重要；

一码多端，跨多端真机效果；

3.2 打造产品矩阵，用户体验

为了提升携程市场洞察平台的应用范围和市场竞争力，我们打造了一个全面的产品使用矩阵，实现了移动端、小程序、现有的 Web 端、微信公众号和 API 服务的完整覆盖。通过这种多端覆盖，我们能够更好地满足用户在不同场景下的使用需求，提升产品的市场竞争力。

为了提升用户体验，我们打通了多种设备和多种类型的登录态，支持用户通过不同设备和渠道无缝登录。除了 Donut 提供的 5 登录方式之外，还支持了密码登录等总共 10 种登录方式，给客户最大的便利和可选择性。

在市场洞察平台中基于 Donut 实现了跨多端高性能移动端的技术实践，确保了产品在各种移动场景下的优异表现。

通过这些技术创新和优化，我们不仅提升了产品的应用范围和市场竞争力，还显著改善了用户体验，为用户提供了更加便捷和高效的使用体验。

RN 框架在携程旅行鸿蒙应用的全业务适配实践

【作者简介】携程鸿蒙框架技术团队，负责携程旅行鸿蒙系统原生应用开发，为鸿蒙生态用户提供一站式旅行服务。

引言

携程作为鸿蒙生态在旅游行业的重要合作伙伴，早在鸿蒙服务卡片时期就和华为开始合作。2023 年 9 月，华为宣布鸿蒙原生应用启动开发，同年 12 月，我们完成携程旅行鸿蒙 Beta 版本的开发，技术上基于 Web+部分原生的方案实现。24 年 6 月 HarmonyOS Next 系统正式内测后，为了让鸿蒙生态的用户使用到携程一站式的旅行服务，我们开始在鸿蒙系统上对全业务进行适配。

一、RN 在携程业务使用现状

2019 年，携程开始在线上使用 RN 框架，并结合自身的业场景，对 RN 框架进行了开发和改造，研发了 CRN 框架（以下简称 CRN）。2021 年，CRN 成为携程主流的开发框架。集团内有 20+个 App 接入 CRN 框架，其中核心的 App 都已接入。携程旅行 App 中，200+个业务 Bundle 在线上运行，业务页面数量超过 2000 个，超过 80%的业务使用 CRN。

二、技术选型（为什么选择 CRN）

从新技术的选择到落地的实践上看，业务对技术的要求往往是以下几个方面：

1）功能全，全量业务都能快速的适配上线

2）性能好，用户体验多端一致

3）成本低，复用现有在其他平台的运行的代码

为了满足业务需求，鸿蒙的实现技术上我们选择了 CRN，主要考虑：

1）基建成熟度高：有配套研发/测试/发布/运营监控系统，内部交流活跃，知识沉淀深

2）业务适配成本小：业务不需要重新再开发一遍，可以使用现有的业务代码

3）开发能快速上手：业务开发还是使用原有的技术进行开发，在鸿蒙上运行

4）产品迭代效率：支持每个周期的产品迭代，快速在鸿蒙系统的手机上线

三、CRN 适配实践

3.1 版本升级

线上携程旅行 App 使用的 React Native（RN）版本是 0.70.1，而鸿蒙 RN 版本是 0.72.5。因此，适配鸿蒙的第一步是将 RN 版本从 0.70.1 升级到 0.72.5。版本升级包含了如下几个方面：

3.1.1 RN 版本差异分析

我们对比 RN 0.70.1 和 0.72.5 框架库的差异，整体改动点不多。为了降低业务方升级成本，我们在框架底层对废弃的组件和 API 变更做了兼容，尽可能减少业务使用方的改动。

3.1.2 CRN 框架改造

CRN 框架覆盖了文档、工具、开发框架、发布、监控、排障全链路。对应框架的改造也从这几个方面进行。

1）在文档方面，我们编写了详细的业务升级文档，列出业务方需要关注的点和常见问题。

2）在工具方面，提供了一键式 CLI 升级工具，只需在业务工程执行一行升级命令，即可完成工程升级改造。

3）在开发框架方面，改造涉及点比较多，包括：

- 对 Native 运行时升级，升级 RN 0.72.5 核心库，合并对官方 RN 库的自定义改动点。
- 对 JS 打包工具升级，支持现有的拆包逻辑，合并对官方 RN 库的自定义改动点。
- 梳理使用到的社区三方库，统一三方库版本升级至鸿蒙 RN 三方库要求版本。
- 对 Hermes 引擎进行升级，合并自定义改动点。
- 对 RN 自定义组件和 API 进行新架构改造。

4）在发布方面，对现有的 CRN 发布系统进行改造，支持选择鸿蒙平台进行单独发布。发布的产物下发和线上 IOS/Android 进行隔离，保证测试上线阶段，不影响已经上架的 IOS/Android 应用。

待后续鸿蒙应用稳定，再支持一键同时发布 IOS/Android/HarmonyOS Next 平台。又考虑到业务场景存在一套代码，跨 RN 版本发布。发布系统改造，支持了发布时根据发布单选择的 RN 版本，自动选择依赖配置进行打包发布。提升业务发布效率。

5）在监控方面，实现鸿蒙端的监控数据上报，接入到现有的监控系统，方便线上监控。

6）在排障方面，实现鸿蒙端的异常数据上报，接入现有排障系统，方便线上排障。

3.1.3 业务工程改造

1）业务方按照提供升级文档和工具进行具体业务工程改造。

2）升级改造后，进行本地开发环境测试，发现问题，解决问题。

3）本地测试通过后，进行打包发布，进入集成测试阶段。

在升级过程中，工作量最大的部分是“RN 自定义组件和 API 实现新架构改造”。

这里先介绍下 RN 新架构。RN 新架构是指从 0.68 版本开始后的架构。主要包括：

Turo Modules 模块系统，替换老架构中的 Native Modules，用于 JS 到 Native 的 API 同步调用。

Farbic 组件系统，替换老架构中 Native Component，支持同步渲染。

由于鸿蒙 RN 只支持新架构，所以需要将 RN 自定义组件和 API 实现进行新架构改造。在携程旅行 App 中，我们使用有 100+的自定义组件和 API。这部分的改造工作量非常大，建议在做适配时优先处理这部分工作。

3.2 差异化工作

在 RN 版本升级到 0.72.5 后，开始鸿蒙端特有的适配。

鸿蒙 RN 框架特点：

- 已实现了官方 RN 大部分组件、API
- 已实现社区常用的三方库
- 自定义组件和 API 需要应用开发自行实现

差异化工作：

1）自定义组件和 API 实现

- 100+自定义组件和 API，基于鸿蒙原生开发实现，再封装提供给 RN 调用
- 按优先级分阶段实现这些自定义组件和 API，保持上层 JS 接口不变

2）RN 工程改造

- 添加 react-native-harmony 和 react-native-harmony-cli 依赖库

- 适配 Platform.OS，Platform.select 等 API
- 实现 xxx.harmony.js 文件，逻辑与 IOS 保持一致
- 升级三方库版本，如 react-native-gesture-handler，从 1.X 版本升级到 2.X 版本
- 三方库版本升级后，对不兼容的地方做适配

3.3 原生组件开发

携程 CRN 框架经过近 8 年的迭代，业务线非常复杂，自定义的组件、turboModule 有 100 多个。在鸿蒙中适配 CRN，首先面临的工作就是将这些自定义组件、turboModule 在鸿蒙原生端用 ArkTS 重新实现。

我们面临以下几个挑战：

1）工作量

这些组件经过了近 8 年的迭代，开发负责人可能几经易手。有些复杂组件，如信息流组件、自定义地图、日历组件、多媒体组件等，逻辑异常复杂，经过跟原开发负责人、产品等初步讨论，工作量都超过单人一个半月。而我们面临的是 100 多个组件、turboModule 的重实现。

2）HarmonyOS Next 逐步完善，与 Android、iOS 在某些特性上有差异

开发过程中发现了很多 HarmonyOS Next 功能不完善、存在若干 Bug 的地方，毕竟是一个新系统，我们与华为同学紧密合作，一一解决了问题，这个过程见证了鸿蒙系统的愈发成熟。

出于安全考虑，鸿蒙系统有一些新特性，比如选取图片视频进行编辑的场景，在 Android、iOS 中，申请用户权限之后便可以拿到整个系统相册的图片视频，这确实可能存在一些安全隐患。鸿蒙在最开始就切割了这一操作，即使 App 经用户同意申请了读相册权限，也无法拿到系统主相册的图片视频，本意是让 App 直接跳到系统相册选取图片之后返回，只提供当次选中的图片信息给 App，从而彻底断绝了 App 侵犯用户隐私的可能。

但我们的多媒体场景比较复杂，用户选取图片、视频后会跳入编辑页，且可以重回相册页选择其他图片，也就是说我们的图片视频选择页与编辑页存在联动，鸿蒙提供的这种跳入系统相册的方式显示无法满足我们的需求。

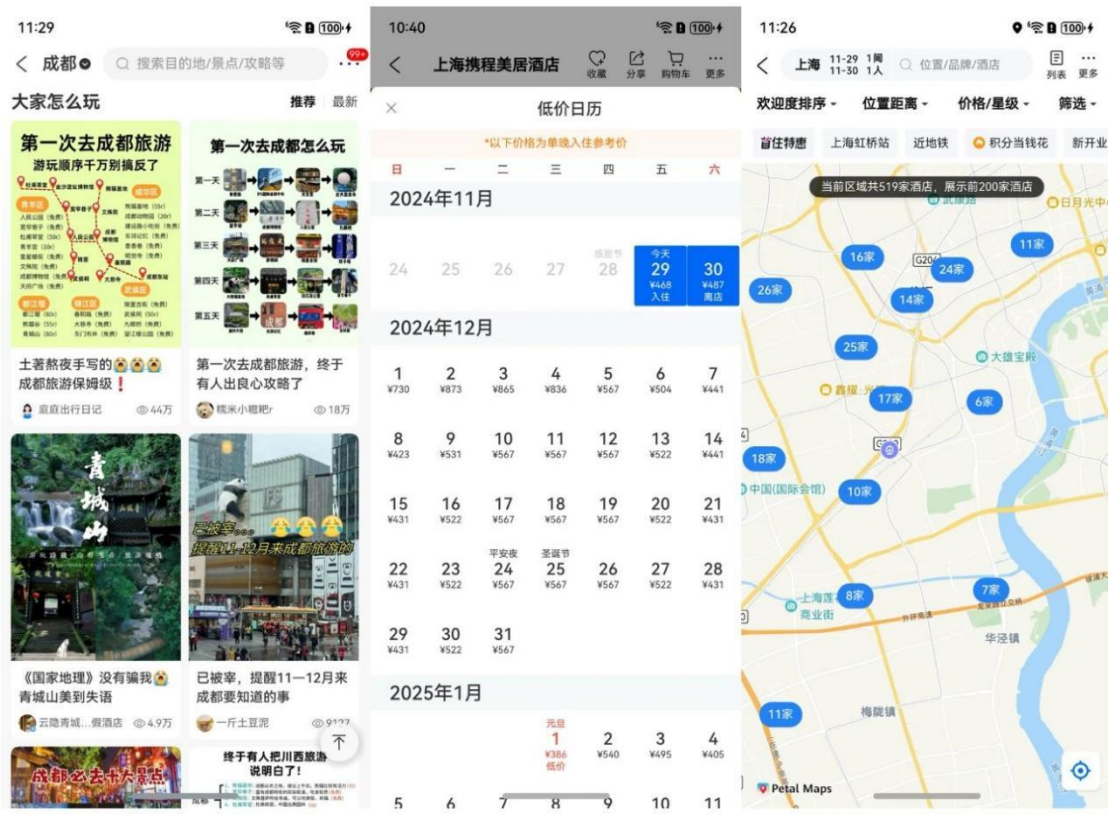
后续经过讨论，鸿蒙提供了相册 Picker 的方案，将系统相册页封装为组件提供给开发者，我们的图片视频选择页可以内嵌相册 Picker，从而解决了联动的问题。但这个需求从开始评审、开发、测试到最终实现，花费了几个月的时间。

3）RN 组件 C 化

在接入 RN 的过程中，发现鸿蒙中 RN 关键性能指标与 Android、iOS 有差距，华为鸿蒙 RN 团队为了解决性能问题，提出了组件 C 化的方案。

简单来讲，就是将 ArkTS 实现的组件用 C-Api 重新实现一遍，华为方面给出的要求是容器结点（RN 代码中存在标签<></>嵌套的组件）需强制 C 化。虽然携程中这种必须 C 化的组件并不多，但也带来了非常多的适配工作。具体可参考下篇-组件 C 化。

部分组件图如下：



Fabric、TurboModule

最开始，我们在实现相关 Fabric、TurboModule 的时候，鸿蒙 RN 框架还没有提供 Spec 文件 CodeGen 工具，全靠手写。不过现在已经提供了相关工具，具体操作步骤可以参考相关文档。

Spec 文件生成之后，剩下的工作就是相关组件、TurboModule 的功能桥接实现，逻辑较为简单，实现相关功能就好。

需要注意的是：

- RN 代码中存在标签<></>嵌套的组件被视为容器结点，此类型组件需使用 C-API 实现。
- 可以通过 this.ctx 获取 RNOHContext，进而获取 RNInstance，从而获取一系列 RN 端 JS

传入的信息，如 View 宽高、style 等，也可执行发送事件、接收事件、获取 TurboModule 进行其他操作等等。

- 在 RNInstanceImpl 构造函数中有一个 arkTsComponentNames 字段，可以传入所有我们自定义叶子结点 Fabric 组件的名称，用于在 RNOH SDK 内部进行指令分发优化。实现 ArkTS 端 Fabric 组件后，需要将 Fabric 组件的名称加入此列表中。

- 假设存在实现过于复杂或者其他原因无法 C 化的容器组件，RNOH SDK 内部指令优化代码需修改（这也意味着 RNOH SDK 需重新打包编译），关键代码见下文‘性能优化-5.3 RN 指令精简章节。

3.4 组件 C 化

经过与华为的详细沟通，RN 代码中存在标签<></>嵌套的组件被视为容器结点，此类型组件需强制 C 化。也就是此类型的组件：

```
<RNComponent>

    <Text/>

    <Image/>

</RNComponent>
```

RNComponent 算为容器结点

经确认，携程端存在四个需强制 C 化的容器结点组件，分别为：

名称	描述
SwipeoutView	可滑动组件
ScrollView	滚动组件
CustomScrollView	自定义列表组件
CRNModal	modal容器

简而言之，需要把 ArkTS 端实现的组件用 C-Api 再次实现。

3.4.1 CRNModal C 化

CRNModal 在开发测试过程中一步步探索了实现方案，经过多轮测试、方案讨论调整，最终

确定了 C 化方案。

方案一：尝试使用系统 Modal 实现这个组件

后续测试过程中发现系统 Modal 的实现方案为系统 Dialog，层级很高，携程业务线会出现这样一种场景，RN 页面打开 Modal 后点击跳转一个其他页面，新打开的页面会出现在 Modal 的下方，不符合需求，方案淘汰。

方案二：尝试通过新跳转一个透明页面的方式实现 modal

测试发现新跳转一个页面后，RN 的点击事件分发出现问题，无法响应任何事件。且我们 App 的路由方案为 Navigation，经与华为方面沟通，Navigation C 化难度非常巨大，短时间不可行。此方案淘汰。

方案三：尝试在 RN JS 端创建 modal

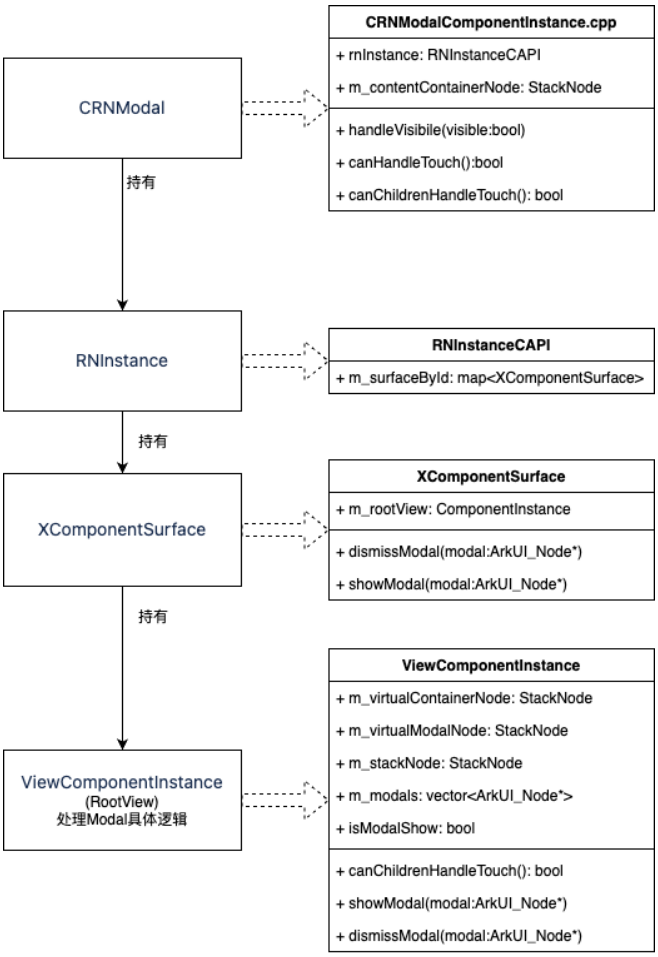
JS 端创建一个 style 为 position: 'absolute', zIndex: 999 的容器，层级提高，显示在其他组件上方来实现 modal。测试发现调用显示 Modal 的地方很多，可能会在一个嵌套很深的层级中，如果在这里尝试展示这个 zIndex: 999 的容器，还是会有被遮挡的情况，最终此方案也被淘汰。

方案四：C++层进行插入

经过内部讨论，这个 modal 应该展示在整个 RN 页面层级的最上方，这在 Android、iOS 中都很好实现，但鸿蒙是一个声明式的语言，无法拿到页面实例，无法拿到父组件，也就无法进行插入。

但研究 RNOH SDK 之后发现，C 化后的 RNInstance 实例在 C++端持有一个 XComponentSurface，所有 RN 页面对应的 Native 组件都被添加显示在这里，而 XComponentSurface 可以获取 rootView 实例 ComponentInstance，这是一个根控件，将这个根控件强转为 ViewComponentInstance 之后，在 ViewComponentInstance.cpp 代码中可以类似 Android，获取 childCount，通过 index 添加 child 等等，进而可以实现在 RN 页面层级最上方添加 modal，最终也是依据此方案，实现了 CRNModal 组件。

流程如下：



3.4.2 开发注意事项

1）在 CAPI instance 中声明的 Node 节点，必须在全局声明，否则会导致 node 节点不能收到 node_event 等消息；

2）设置 node 属性构建 ArkUI_AttributeItem 的时候，如果设置的值是一个 ArkUI_NumberValue 类型，需要指定 size，这个 size 的计算必须除去类型的长度，如下：

```
ArkUI\_NumberValue value\[] = {{.i32 = alignItem}};
ArkUI\_AttributeItem item = {value, sizeof(value) / sizeof(ArkUI\_NumberValue)};
```

3）animateTo 执行动画，在组件析构之后还是会回调，需要控制好生命周期避免 crash；

4）设置 Stack 背景，导致子组件布局错误，是因为 Stack 被作为同级组件从而导致子组件的 postion 参数异常，需要手动处理好 position 问题；

5）可以通过以下方式在 C++层调用 arkTS 方法，获取相关数据：

方案 1：在 ArkTS 里实现一个 TurboModule 方法，然后通过 rnInstance->getTurboModule<XXTurboModule>获取对应的 TurboModule，调用方法，获

取返回值。但此方案涉及 C++与 ArkTS 的跨端调用，性能会差一些，优点是实现简单。

方案 2：通过 ArkTSBridge，添加一个 ArkTS 方法的桥，然后就可以在 C++里直接调用这个 ArkTS 方法。具体实现可以参考 NapiBridget.ArkTSBridgeHandler 里任意方法。此方案性能好，但实现起来稍微麻烦一点。

6）可以通过以下 Api 获取设备的高宽

auto displayMetrics = ArkTSBridge::getInstance()->getDisplayMetrics();
displayMetrics.screenPhysicalPixels.width / displayMetrics.screenPhysicalPixels.scale //直接获取到是 px 单位，需要进行转换，也可以自行修改 TurboModle 的初始化值：

四、遇到的问题和解决办法

在升级适配过程中，我们遇到了一些 RN 新架构问题，还有一些鸿蒙 RN 特有的问题。RN 新架构问题：

- IOS Animated.timing 设置 useNativeDriver:true 后，内嵌按钮无法点击
- IOS TouchableOpacity 内嵌 Aminated.View ，Aminated.View 开启动画变更位置后，无法点击
- IOS Image 样式设置 borderRadius 显示不全
- IOS minimumFontScale maxFontSizeMultiplier 不生效
- Aminated.View 内嵌 Modal 组件，内部 TouchableOpacity 点击不响应
- FlatList、ScrollView stickyHeaderIndices 吸顶功能多次滑动后失效
- Aminated.View 、Animated.ScrollView、layoutAnimation 动画卡顿
- 样式中使用了 zIndex 属性层级可能不生效,尝试添加 position:relative 属性后生效
- 组件需要设置默认高宽，不然布局展示可能发生截断

由于动画、样式、性能影响较大，最终决定在 RN 0.72.5 版本（iOS/Andriod）中只使用 Turbo Modules，不开启 Fabric 模式，来规避掉这些问题。

但鸿蒙 RN 只支持新架构，新架构存在问题有些在鸿蒙端同样存在。我们和华为伙伴紧密沟通来处理这些问题。对于无法规避问题只能业务侧做兼容处理。

鸿蒙 RN 特有的问题：

问题：RN Modal 弹窗显示时，再打开一个 H5 页面会显示在 Modal 下面

解决办法：实现一个 View 层级的 CRNModal 替代 RN Modal

问题：绝对定位中添加 top:“auto”导致元素不显示

解决办法：去除 top:“auto”设置

问题：zIndex:-1 元素不显示

解决办法：在最外层的 View 添加 collapsable={false}属性

问题：position:absolute 样式漂移

解决办法：在外层的 View 添加 collapsable={true}属性

问题：react-native-harmony/metro.config 和现有的自定义 metro 配置冲突

解决办法：提取 react-native-harmony/metro.config 中 harmony 平台相关处理，合并到自定义 metro 插件中

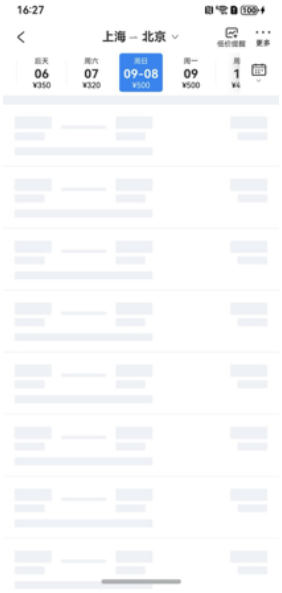
五、性能优化

华为内部对鸿蒙系统寄予厚望，为了追求更好的用户体验，希望鸿蒙 APP 核心业务场景性能指标达成业内最佳水平。对携程来说，大多数业务页面都是 RN，RN 技术栈对性能指标非常敏感，很小的性能优化或劣化，都会大幅影响用户体验。

5.1 CRN 预加载

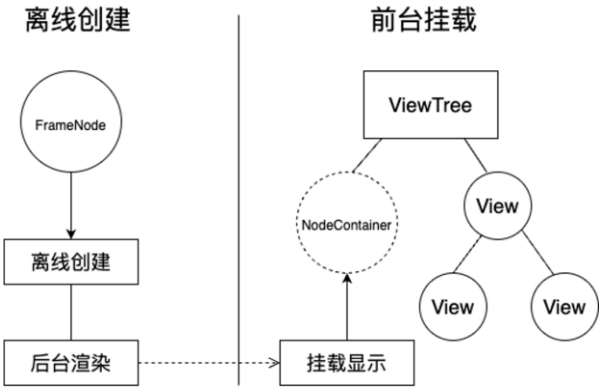
默认情况下，我们会在页面的生命周期中去加载 rn_bundle，因为页面已经进入生命周期开始展示了，加载 bundle 又会有一定的耗时，这种情况下，就会产生白屏现象。

携程也存在某些页面依赖接口数据且接口返回比较慢的情况，比如机票列表页，在进入页面白屏之后又会有长时间的骨架屏，用户体验差。



经过调研，携程端基于系统的 FrameNode 能力，实现了 CRN 预加载方案，解决了上述问题。

5.1.1 FrameNode



鸿蒙中 FrameNode 是一个非常强大的能力，不光是各大厂商在用，官方 ArkUI 中也大量使用了 FrameNode 进行性能优化。

它的特点用一句话可以描述：后台离屏渲染，前台上树展示。

利用这个特点，可以实现组件渲染与页面展示的完全分离。也就是说组件的创建渲染不再依赖页面的生命周期，这样我们就可以做很多事情了。

但正因为 FrameNode 组件会在后台真实渲染，它使用起来会有一定的风险，后台渲染的组件可能会影响前台行为，比如改变状态栏颜色、弹 Toast、弹 Dialog 等，这些都需要人为进行规避。

在 RNSDK 中我们对 Toast、Dialog、状态栏等行为的 TurboModule 调用，根据页面状态进行了拦截，页面不可见时，上述这些 TurboModule 的调用都不会生效，而且会记录最后一次拦截的行为及参数，在页面变为可见时，恢复最后一次被拦截的行为。

基于 FrameNode 能力，实现了鸿蒙中 CRN 预加载的 1.0 和 2.0 方案，下文会详细介绍这两个方案。

另外需要注意的一点是，携程在 RN 中使用 FrameNode 过程中，遇过一个困扰许久的问题：使用 FrameNode 加载 RN 页面时，在某些比较复杂的页面，会发生非常严重的 JS 阻塞现象，用户的点击、返回等操作行为被页面渲染指令阻塞，迟迟得不到响应，极度影响用户体验。

经过与华为方面的联合排查，发现是因为 RNInstance 初始化时传入的参数：disableConcurrentRoot=true 导致。此参数会关闭 React18 一个性能优化的功能：微任务指令批量提交，从而导致在 JS 代码 setTimeout 中进行 setState 时，指令立即提交，总指令数大幅增加，进而大幅影响 RN 指令处理效率。

大家如果也会在项目中用到 FrameNode 进行 RN 页面的性能优化，在初始化 RNInstance 时，disableConcurrentRoot 参数一定要传 false。

5.1.2 CRN 分包

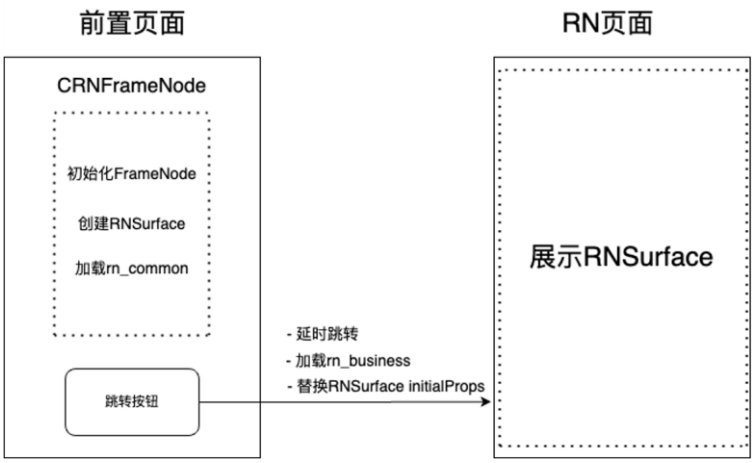
要理解我们 CRN 的预加载方案，首先要了解我们的分包逻辑，具体可参考文章：《近万字长文详述携程大规模应用 RN 的工程化实践》。

总体而言，将业务 bundle 的加载分为两部分：rn_common & rn_business。其中 rn_common 包含完整的基础框架能力，rn_business 则是具体的业务逻辑代码。通过 nativeRequire 的方式，分行加载 rn_business 中的业务代码，然后在加载了 rn_common 的空白页面上进行渲染。

可以发现，CRN 这种分包模式完美契合 FrameNode，我们使用 FrameNode 预渲染一个加载了 rn_common 的空白页面，这个空白页面不会渲染 UI 元素且具备完整的框架能力，不会有任何影响前台页面的行为，等到页面真正展示时，才去加载业务代码 rn_business，进行 UI 渲染，从而完美规避 FrameNode 的使用风险。

也正是基于此，我们实现了 CRN 预加载 1.0 的方案。

5.1.3 CRN 预加载 1.0



预加载 1.0 方案：

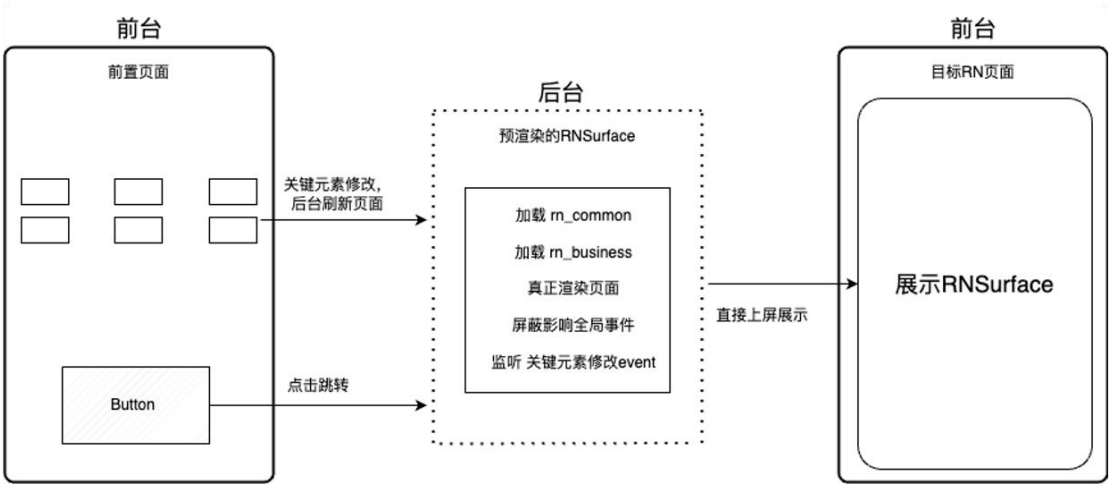
- 在前置页面通过 FrameNode 预加载一个 RNSurface，利用这个 RNSurface 去加载 rn_common，完成后可以理解为后台存在了一个具备所有框架能力的空白页面。
- 用户点击跳转 RN 页面时，添加一个用户几乎不可感知的延时去加载 rn_business。
- 充分利用这个跳转延时 + 页面创建 + 页面切换的动画时间去加载业务 bundle、渲染等。
- 业务 Bundle 加载完成后，动态替换业务自定义的 intialProps
- 做到了 rn bundle 加载、 渲染与页面生命周期的完全隔离。
- 目前预加载 1.0 方案在全业务默认使用，基本解决了 RN 页面首帧白屏问题。

在携程的某些业务线中，页面 UI 依赖网络接口数据，且受外部接口影响，响应较慢。这时候，打开页面会有较长时间的骨架屏 loading，也非常影响用户体验。

如果在前置页面中，我们可以大概率猜到用户下一步跳入的目标页面，那是不是可以利用 FrameNode 将目标页面提前加载，且根据前置页面的参数进行动态刷新，这样用户真正跳入目标页面的时候，就可以直接上屏，达到秒开的效果。

基于此，我们实现了 CRN 预加载 2.0。

5.1.4 CRN 预加载 2.0



预加载 2.0 方案：

- 在前置页面通过 FrameNode 预加载了一个真实的 RN 页面，完成了加载 rn_commom、rn_business、接口请求、渲染等一系列流程。
- 前置页面中影响下一个页面关键参数发生改变时，发消息给后台预加载的的 RN 页面，RN 页面接收到事件，拿到关键参数后进行网络请求，得到数据后对页面进行刷新。
- 用户点击跳转到目标页面时，直接将后台已经预渲染好的页面上屏展示。
- 因为页面已经在后台被真实渲染，有影响前置页面的风险，虽然我们在 RN SDK 层面已经做了一层拦截，但这种拦截不可能 cover 所有场景，所有接入了预加载 2.0 方案的业务都必须在上线前经过完整回归测试。
- 目前，我们在机票列表页及火车票详情页使用了预加载 2.0 方案。

对比视频：

性能优化关闭：

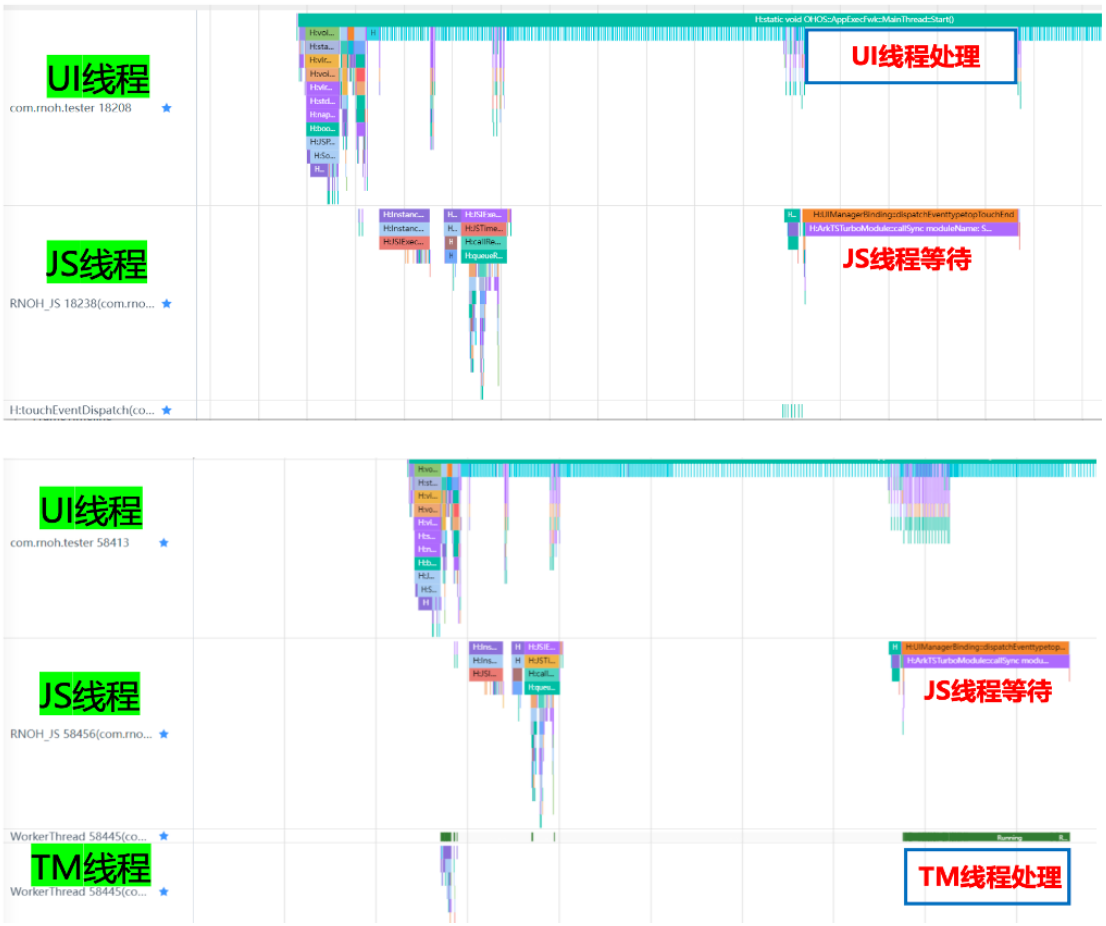

14:34
超值好券，等您来领
暑期玩
机票
特价机票
火车票
汽车·船票
单程
往返
上海
北京
7月4日明天
选择乘机人
快速找到低价票
经济舱
公务舱/头等舱
机票+酒店
更优惠
查 询
机票预订服务由上海华程西南国际旅行社有限公司提供
搜全球低价
搜全国低价
低价提醒
学生特权
展开全部
Hi，下午好
打卡闪光城市 赢5万返现!
03 Jul 2024
今日推荐
未登录
航班助手
我的权益
定制包机
报销凭证
我的订单

性能优化开启:

5.2 RN TurboModule 运行在 Worker 线程

前段时间我们在 RN JS 端对 TurboModule 调用加了一个埋点，统计 TurboModule 方法调用的耗时，后续也是根据这个埋点生成了一个报表，发现在鸿蒙中，TurboModule 同步方法调用耗时比 Android、iOS 耗时长 10 倍，某些方法甚至慢 100 倍。

经过分析，在 Android、iOS 中 TurboModule 都是运行在单独的子线程中，而在鸿蒙中，TurboModule 都运行在主线程，主线程要承载一些别的任务比如页面渲染、用户操作行为响应等，这些行为会导致鸿蒙中 TurboModule 的调用被阻塞，耗时就长。比如下图的 Trace，如果 TurboModule 在 UI 线程运行，那就可能会被阻塞，阻塞的这段时间，js 线程只能等待，而这段等待是毫无意义的。



前段时间，鸿蒙 RN SDK 也是加入了 TurboModule 运行在 Worker 线程这个能力。RNInstance 在创建时会同步创建一个 worker 线程，专门用于 TurboModule 运行。我们要做的是对工程中 TurboModule 代码进行适配改造，使之可以运行到 worker 线程中。

整个适配过程也存在一系列的问题。

首先，鸿蒙的 ArkTS 衍生自 TS 语言，基于 Actor 线程模型，内存不共享，线程间数据通信非常麻烦。

为了解决线程间通信流程繁琐的问题，鸿蒙提供了 Sendable 注解，可以理解为被这个注解修饰的对象会在共享内存创建。但 Sendable 存在一个问题，Sendable 对象的成员变量只能是 Sendable 对象或其他特定的数据类型，也就是说我们如果对一个对象进行 Sendable 改造，就必须对他的所有成员变量进行 Sendable 改造，也需要对成员变量的成员变量进行 Sendable 改造，那这个改造过程就存在指数级扩散的问题。

另外，Sendable 注解提供的时候，我们大部分代码都已经完成了，在这种成熟的大型项目中再重新进行 Sendable 改造的成本非常高，大家各自 App 如果还没开始或者刚开始开发，一定要考虑 Sendable 适配的问题，比如数据类型默认使用 collection 下属 map、array，class 默认添加 Sendable 注解等。

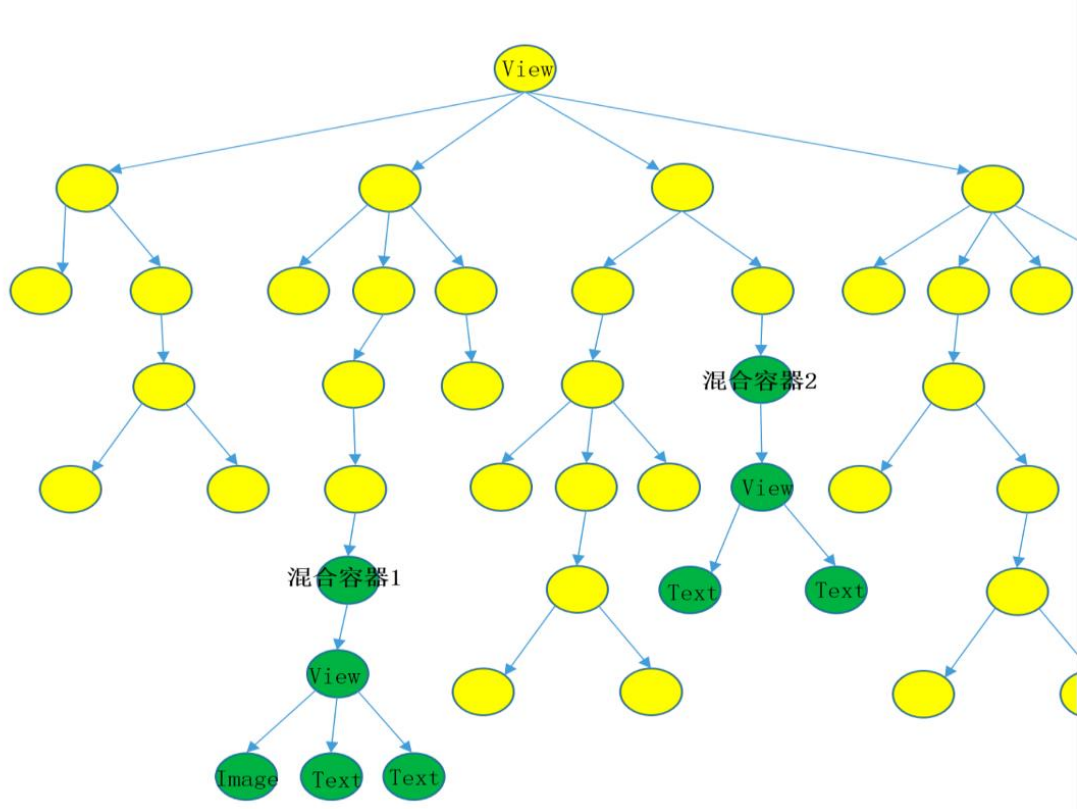
目前，我们适配完成了 7 个 TurboModule，其他 TurboModule 做 Sendable 适配的成本非常高，正在逐步进行中。

5.3 RN 指令精简

在鸿蒙中，RN 支持两套组件：C-API 实现的组件以及原生 ArkTS 实现的组件。

C-API 实现的组件性能更好，华为的支持力度更大，出于性能考虑，大多数厂商使用 RN 时，都会选择 C-API 实现的组件。

C-API 组件的 Create、Insert、Update、Remove 等指令不再需要传递给 ArkTS 侧。仅仅几个自定义的 ArkTS 组件需要将指令传递给 ArkTS 侧。如下图中绿色节点的指令。



鸿蒙 RNOH SDK 中有默认算法，可以保留叶子节点 ArkTS 组件的指令，如果项目内没有 ArkTS 容器组件，RNInstance 初始化时添加配置 arkTsComponentNames 就好。

但在携程的业务中存在 AdatpterMap 这个容器组件，依赖系统花瓣地图，这个地图 C 化难

度巨大，所以我们的 AdatpterMap 暂时也只能由 ArkTS 实现。

这就需要我们自己设计算法。在保证性能的情况下，完整保留 AdatpterMap 及其子组件的相关指令。

以下是关键代码：

RNOHSDK/src/main/cpp/RNOH/MountingManagerCAPI.cpp::getValidMutations：

```
facebook::react::ShadowViewMutationList MountingManagerCAPI::getValidMutations(
    facebook::react::ShadowViewMutationList const& mutations) {

...
//需要特殊处理，保留容器组件及其子组件所有指令的容器组件名称
std::unordered_set<std::string> whiteListArkTsComponentNames = {
      "AdapterMap", "AdapterMapMarkersContainer", "AdapterMapMarker"};
//第一次遍历：只遍历 create，从前到后找到混合组件名称，只保存 tag
for (auto mutation : mutations) {
    if (mutation.type == facebook::react::ShadowViewMutation::Create) {
        ...
        //特殊保留地图容器组件 tag
        if (whiteListArkTsComponentNames
                    .count(newChild.componentName)) {
            arkTsComponentTags.push(newChild.tag);
        }
    }
}

if (!arkTsComponentTags.empty()) {
    // 第二次遍历：只遍历 insert，找到混合组件 tag 和它的子组件的 tag。采用广度遍历方式，这里也只保存 tag
    for (auto mutation : mutations) {
        if (mutation.type == facebook::react::ShadowViewMutation::Insert) {
            ...
            //保存地图容器组件 tag 和它的子组件的 tag
        }
    }
}

//第三次遍历：根据 2 中齐全的 tag，重新过滤所有指令，保留这些 tag 的 create、insert、update、remove 指令。
for (auto mutation : mutations) {
    ...
    //根据组件 Tag，保留需要传递给 ArkTS 的所有指令
```

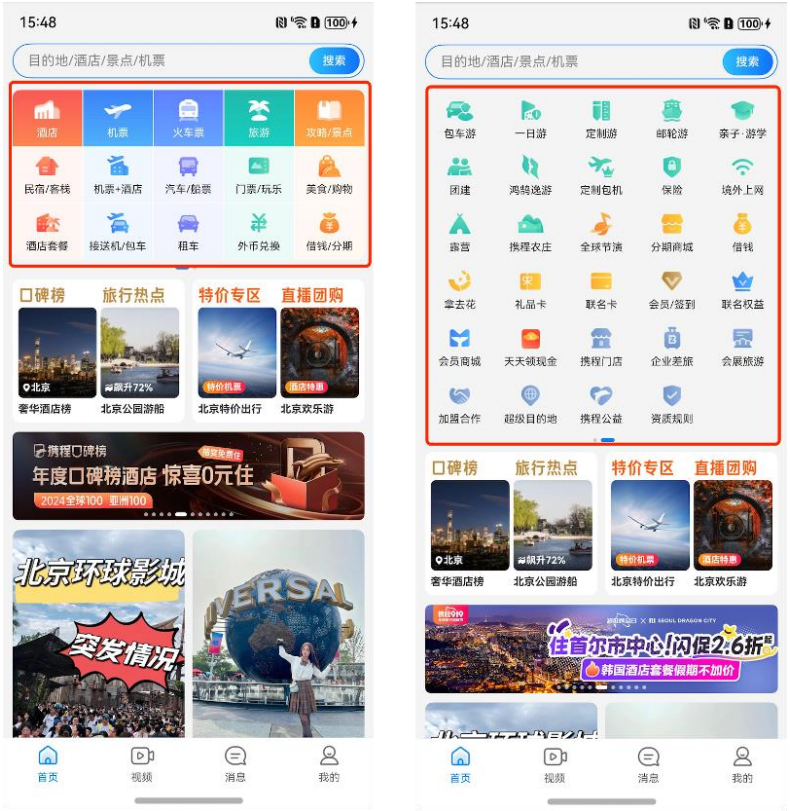
```
}
  return validMutations;
}
```

再来看下成果，测试 RN 页面中，算法过滤的不需要传递到 ArkTS 侧的指令数超过 99.9%。

页面	优化前指令数	优化后指令数
酒店首页	3092	11
酒店套餐	2181	0
机票+酒店	496	2
酒店	3838	3
美食/购物	1542	3

5.4 分帧渲染

分帧渲染主要用在 App 的启动优化中。首页宫格存在两屏，二屏在刚开始是不可见的，可以在首页加载完毕之后再去加载宫格二屏。



分帧渲染可以监听到帧渲染的回调，这样就可以对页面元素的加载优先级进行定制，将重要

的元素优先加载，不重要或者不可见的元素后续加载。进而提升页面的性能。

关键代码：

```
private myDisplaySync?: displaySync.DisplaySync

updateStage() {
    if (this.stages == 0) {
      this.myDisplaySync = displaySync.create();
      this.myDisplaySync.start();
      this.myDisplaySync.on('frame', (frameInfo: displaySync.IntervalInfo) => {
        this.updateStage();
      });
    }

    this.stages++;
    if (this.stages == 3) {
      this.myDisplaySync?.stop();
    }
}
...

  build() {
    Column() {
      Scroll(this.scroller) {
        Row() {
          //默认加载宫格首屏
          if (this.stages > 0){
            this.genFirstCell(0)
          }
          //三个渲染帧之后，加载宫格二屏
          if (this.stages > 2){
            this.genFirstCell(1)
          }
        }
        ...
        }
```

接入分帧渲染，控制宫格二屏的渲染时机后，首页的启动耗时减少了 20ms。

5.5 后续性能优化

华为鸿蒙 RN 团队规划有一个性能优化的 feature，在这里简单介绍下。

5.5.1 更换 RN JS 执行引擎：JSVM（基于 V8）

JSVM 相较 hermers，预计可以提升 20%的 JS 解析性能。前段时间华为提供了一个 rn sdk，我们新建了一个分支验证了一下这个 JSVM，js 的加载速度确实比 hermes 要快一些。但 RN 产物 jsbundle 的加载比 hermes 要慢，这意味着页面的首屏性能会受到一定影响。这是我们非常关注的一个性能指标，问题得到解决之后，我们也会进行切换。

六、成果和未来规划

经过 4 个月鸿蒙版本的开发和适配，2024 年 6 月 18 日携程在鸿蒙应用商店上架了首个全业务全场景的携程旅行鸿蒙版应用。业务方在 Android/iOS 上的一套 CRN 代码，只需经过简单的适配，就能正常在鸿蒙系统上运行，甚至有些业务不需要修改，现有的代码直接在鸿蒙系统上能完整的跑完业务流程。

未来，我们还会在以下两个方面持续对鸿蒙 CRN 框架进行优化：

用户体验

用户体验和性能一直是我们关注的重点，CRN 在鸿蒙系统上还有很大的优化空间。我们会持续在性能上继续打磨和提升。

技术布局

为了追求高效率、低成本的研发模式，未来携程业务开发会大量使用一码多端的框架 xTaro。后续 xTaro 会支持鸿蒙系统，真正实现让业务的一套代码能在多端多平台多应用场景上全矩阵运行。

鸿蒙生态的发展是一个持续且快速的过程。随着鸿蒙系统的不断迭代升级和生态的逐步完善，我们会持续为用户提供更加智能、安全、便捷的一站式的旅行应用。

携程弱网识别技术探索

【作者简介】Aaron，携程移动开发专家，关注网络优化、移动端性能优化。

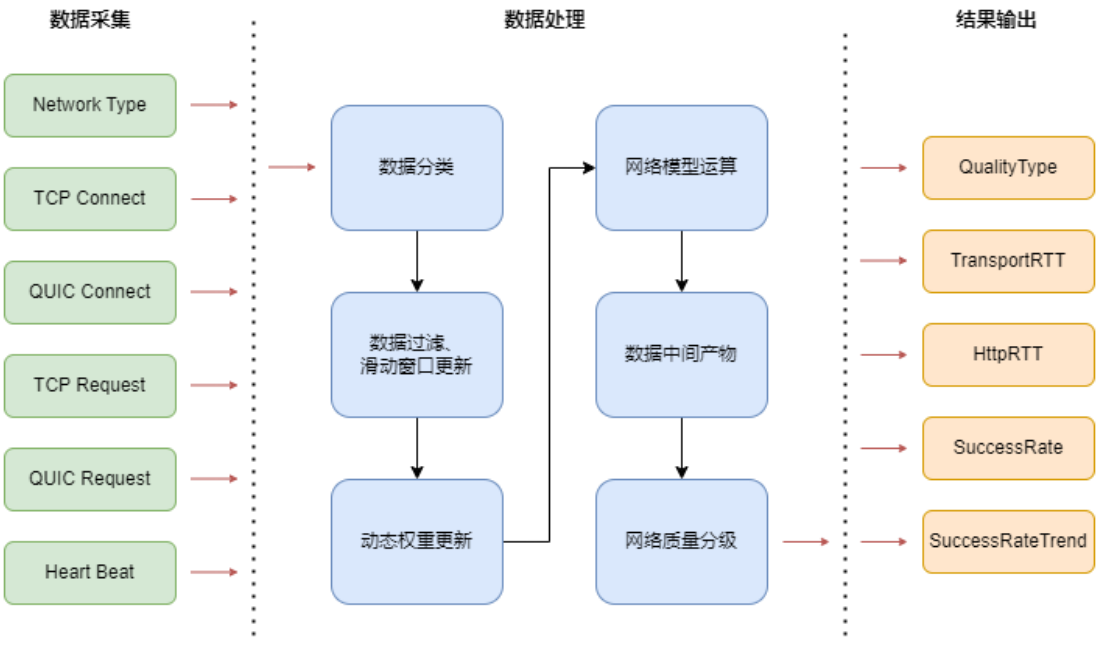
一、背景

自从 2010 年携程推出"无线战略"，并发布移动端 APP 以来，无线研发团队对于客户端网络性能的优化就一直没有停止过。经过这十几年持续不断的优化，目前携程的端到端网络性能已经处于一个相当不错的水平，大盘数据趋于稳定，优化也随之进入 "深水区"，提升难度巨大。

结合线上的一系列客诉反馈，我们发现即使大盘的数据再优秀，用户网络表现不佳的个例 case 仍然层出不穷，排查后大部分被我们归因到"弱网"。这部分"弱网"长尾数据相比大盘均值仍有巨大的提升空间，如果可以针对性优化的话，对于提升整体用户体验和减少客诉都有非常明确的价值。

既然要优化"弱网"，那第一步一定是建立相应的"弱网识别模型"，准确识别出弱网场景，本文即探讨携程在弱网识别方面的技术探索，包含技术选型细节和关键的路径思考，欢迎沟通交流。

二、技术方案



携程弱网识别模型的整个工作流程由数据采集、数据处理、结果输出三部分组成，接下来我们顺着流程来逐个剖析相关细节。

2.1 数据采集

2.1.1 可以客观反映网络质量的指标有哪些

说到可以客观反映网络质量的指标，业内定义清晰且获得公认的有如下这些：

HttpRTT：Http 请求一次网络往返的耗时，具体口径是客户端从开始发送 RequestHeader 到收到 ResponseHeader 第一个字节的时间差

TransportRTT：网络通道上一次数据往返的耗时，具体口径是客户端从开始发送数据到收到服务端返回数据第一个字节的时间差，要减去服务处理的耗时

ThroughPut：网络吞吐量，是指定位时间内网络通道上下行的数据量，具体口径是单位位时间内上行或者下行的数据量除以单位时间，由于上行数据量受到业务因素的影响较大，我们一般仅关注下行

BandwidthDelayProduct：带宽时延乘积，顾名思义是指网络带宽乘以网络时延的结果，即当前网络通道里正在传输的数据总量，是一个复合指标，可以客观反映当前网络承载数据的能力，计算方式是 ThroughPut * TransportRTT

SignalStrength：信号强度，移动互联网时代，设备依靠 Wifi 或者蜂窝数据接入互联网，信号强度会影响用户的网络表现

NetworkSuccessRate：网络成功率，剔除业务影响的纯网络行为成功率，与网络质量呈正相关，包含建联成功率、传输成功率等

2.1.2 携程选择了哪些指标作为模型输入？为什么

对于网络质量识别，业内做的比较早的是 Google 的 NQE（Network Quality Estimator），国内大多数网络质量识别方案也都参考了 Google NQE，NQE 中识别模型的输入主要是 HttpRTT、TransportRTT、DownstreamThroughput 这三个指标。

对于 HttpRTT、TransportRTT，在应用层和传输层都有很多方式可以采集到，且口径清晰，所以这两个指标被我们纳入采集范围。

对于 DownstreamThroughput，我们实践过程中发现，该指标受到用户行为的影响很大，当用户集中操作大量发送网络请求的时候，该指标就偏高，当用户停止操作阅读数据时，该指标就会偏低甚至长时间得不到更新，考虑到指标的波动性，我们不将此指标纳入采集范围。

既然 DownstreamThroughput 被排除在外，那由他参与计算的 BandwidthDelayProduct 也被我们排除。

SignalStrength 信号强度由于 iOS 无法准确获取，考虑到多端一致性，也被我们 Pass。

NetworkSuccessRate 网络成功率这个指标，可能很少被其他方案提及到，我们提出这个指标并将他纳入采集范围的主要原因是，基于 RTT 的网络识别模型，在遇到网络波动导致的用户大面积请求失败时，无法获取到有效的 RTT 值，导致识别的准确性和实时性都收到影响，引入网络成功率可以很好的弥补这个缺陷，最终线上生产环境验证也证明了该指标的必要性。

最终，携程的网络质量识别模型采集 HttpRTT、TransportRTT、NetworkSuccessRate 作为输入指标。

2.1.3 输入的网络指标如何采集

携程的网络请求，主要有 Tcp 代理通道、Quic 代理通道、Http 通道三种网络通道。对于上述提到了三个输入指标，我们从如下网络行为中进行数据采集：

TransportRTT：通道心跳耗时、Tcp 通道建联耗时、Http 通道建联耗时（Tips：建联耗时不涉及业务处理，近似于纯网络传输耗时，所以我们把他作为 TransportRTT；Quic 建联约等于 Tls 握手耗时，且存在 0RTT 等特性干扰，所以不采用）

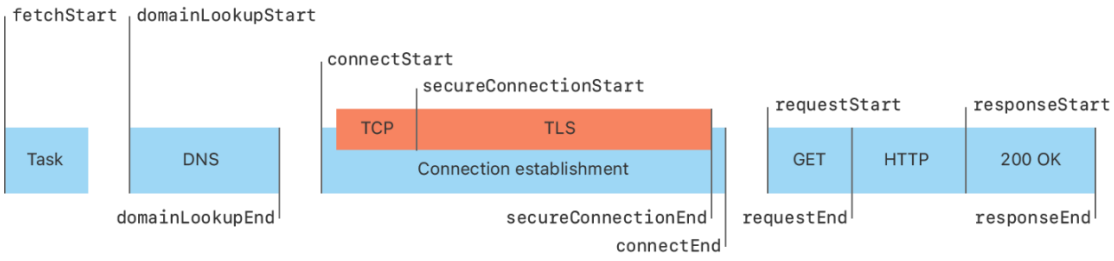
HttpRTT：标准 Http 请求的 responseHeader 开始接收时间减去 RequestHeader 的开始发送时间、自定义网络通道请求的开始接收时间减去开始发送时间

NetworkSuccessRate：Tcp 建联成功状态、Quic 建联成功状态、心跳成功状态、Http 请求是否完整接收到 Response

对于自定义的 Tcp、Quic 代理通道，按照上述口径在网络通道相关状态回调内统计数据即可，自定义实现参考价值不大，这里就不过多赘述。

对于标准的 Http 请求，我们可以通过获取系统网络框架返回的 Metric 信息或者监听请求的状态流转来获取网络指标。

对于 iOS，iOS 10 之后 NSURLSession 支持通过 NSURLSessionTaskDelegate 的协议方法 URLSession:task:didFinishCollectingMetrics:获取到请求的 Metric 信息，详细信息见附录 1，单次请求的 Metric 定义如下图：



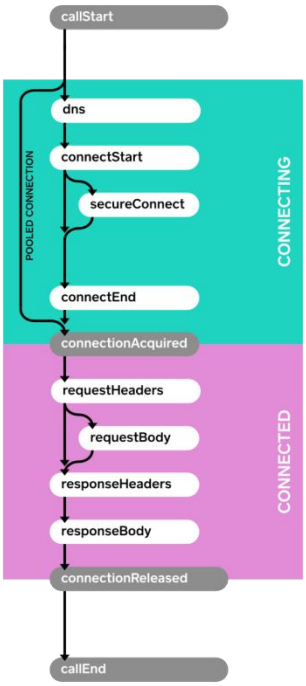
TransportRTT = connectEnd - connectStart - secureConnectionEnd + secureConnectionStart；建联耗时要减去 Tls 的耗时，连接复用时，相关字段为空值，

不纳入计算

HttpRTT = responseStart - requestStart

NetworkSuccessStatus = responseEnd 且没有传输错误；网络成功率只关心传输是否成功，不需要关注 Response 的 http 状态码

对于 Android，系统网络框架 OkHttp 支持添加 EventListener 来获取 Http 请求的状态流转信息，可以在各状态回调内记录时间戳来计算 RTT，详细信息见附录 2，单次请求的 Events 定义如下图：



TransportRTT = connectEnd - connectStart - secureConnectEnd + secureConnectStart

HttpRTT = responseHeadersStart - requestHeadersStart

NetworkSuccessStatus = responseBodyEnd 且没有传输错误

依照上述方法收集到网络数据后，我们把数据封装成对应的结构体，注入识别模型，携程对于网络数据结构体的定义如下，方便大家参考：

```
typedef enum : int64_t {

    NQEMetricsSourceTypeInvalid = 0,                    // 0
    NQEMetricsSourceTypeTcpConnect = 1 << 0,            // 1
    NQEMetricsSourceTypeQuicConnect = 1 << 1,           // 2
```

```
    NQEMetricsSourceTypeHttpRequest = 1 << 2,        // 4
    NQEMetricsSourceTypeQuicRequest = 1 << 3,        // 8
    NQEMetricsSourceTypeHeartBeat = 1 << 4,          // 16
    ......
} NQEMetricsSourceType;

struct NQEMetrics {

    // 本次采集到的数据来源，可以是多个枚举值的或值
    // 例如一次没有连接复用 http 请求，source = TcpConnect|HttpRequest，同时存在 transportRTT 和 httpRTT        NQEMetricsSourceType source;
    // 本次数据的成功状态，用作成功率计算
    bool isSuccessed;
    // httpRTT，可为空
    double httpRTTInSec;
    // transportRTT，可为空
    double transportRTTInSec;
    // 数据采集时间
    double occurrenceTimeInSec;
};
```

2.2 数据处理

2.2.1 数据过滤和滑动窗口

网络数据采集后，注入到识别模型内，需要一个数据结构来承载，我们采用的是队列。

进入队列前，我们需要先进行数据过滤，筛选掉一些无效的数据，目前采用的筛选策略有如下这些：

- 单条 NQEMetrics 数据，在 isSuccessed=true 的情况下，httpRTT、transportRTT 至少有一条不为空，否则为无效数据

- RTT 必须大于最小阈值，用来过滤一些类似 LocalHost 请求的脏数据，目前采用的阈值为 10ms

- RTT 必须小于最大阈值，用来过滤前后台切换进程挂起导致的 RTT 数值偏大，目前采用的阈值为 5mins

数据过滤后加入队列，为了实时性和结果准确性，我们处理数据时，会根据两个限制逻辑来确定一个具体的滑动窗口，只让窗口内的数据参与计算，具体窗口限制逻辑如下：

- 最小数量限制，当窗口内数据过少时，会放大单条数据的影响，导致结果毛刺增

多，所以我们限制最小计算窗口的数据条数为 5

- 最大时间限制，为了数据实时性的考虑，比较旧的数据不参与计算，目前采用的阈值为 5mins

每次计算网络质量时，可以根据这两个限制来确定计算窗口，窗口外的数据可以实时清理出队列，减少内存占用。

上文提到的各种阈值设置，均可通过配置系统更新。

2.2.2 动态权重计算

弱网识别模型的原理简单来说就是将窗口内的一组数据经过一系列处理后，得出一个最终值，再用这个最终值与对应的弱网阈值比较来得出是否是弱网。

出于实时性的考虑，我们希望距离当前时间越近的数据权重越高，所以要用到动态权重的算法，这里我们比较推荐的是”半衰期动态权重“和”反正切动态权重“两种算法。

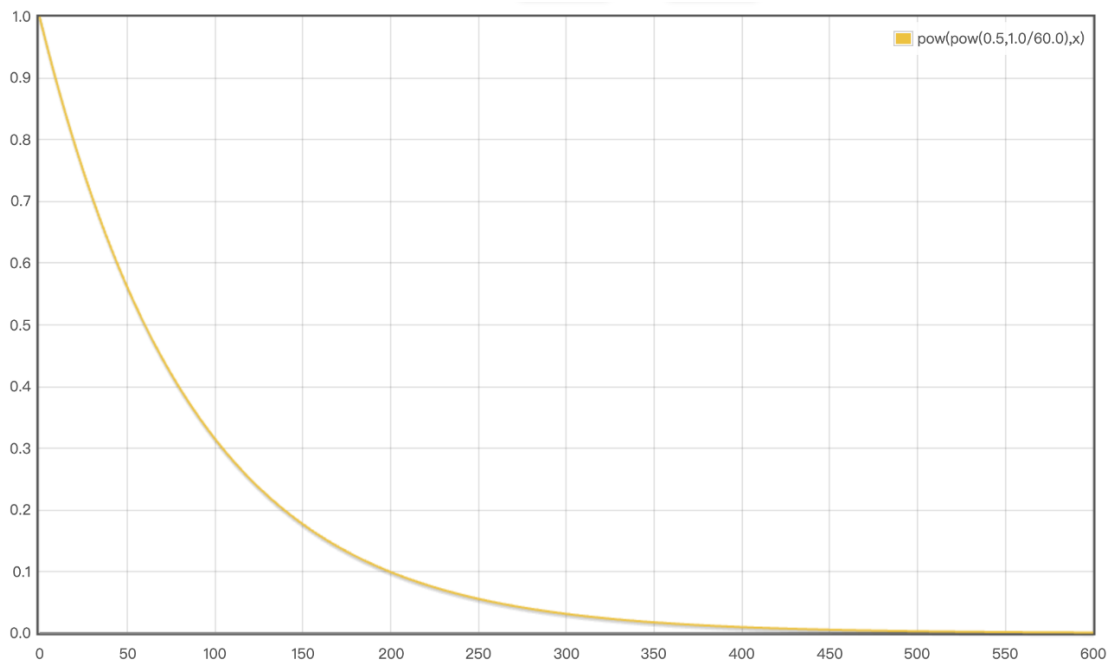
半衰期动态权重

半衰期顾名思义，即每经过一个固定的时间，权重降低为之前的一半。这里衰减幅度和周期都是可以自定义的，计算公式如下：

- 每秒衰减因子 = pow(衰减幅度, 1.0 / 衰减周期)；衰减幅度为浮点型，取值范围 0~1，衰减周期为整形，单位为秒

- 动态权重 = pow(每秒衰减因子, abs(now - 数据采集时间))

以衰减幅度为 0.5，衰减周期为 60 秒为例，对应的函数曲线如下：



横坐标为数据采集时间距今的时间差，纵坐标为权重，从图上可以清晰看到，随着时间差增大，权重无限趋近于 0。

半衰期动态权重也是 Google NQE 采用的权重计算方案，Google 采用的周期是每 60 秒降低 50%，相关代码详见附录 3，部分核心代码如下：

```
double GetWeightMultiplierPerSecond(
        const std::map<std::string, std::string>& params) {
    // Default value of the half life (in seconds) for computing time weighted
    // percentiles. Every half life, the weight of all observations reduces by
    // half. Lowering the half life would reduce the weight of older values
    // faster.
    int half_life_seconds = 60;
    int32_t variations_value = 0;
    auto it = params.find("HalfLifeSeconds");
    if (it != params.end() && base::StringToInt(it->second, &variations_value) &&
            variations_value >= 1) {
        half_life_seconds = variations_value;
    }
    DCHECK_GT(half_life_seconds, 0);
    return pow(0.5, 1.0 / half_life_seconds);
}

void ObservationBuffer::ComputeWeightedObservations(
        const base::TimeTicks& begin_timestamp,
        int32_t current_signal_strength,
        std::vector<WeightedObservation>* weighted_observations,
```

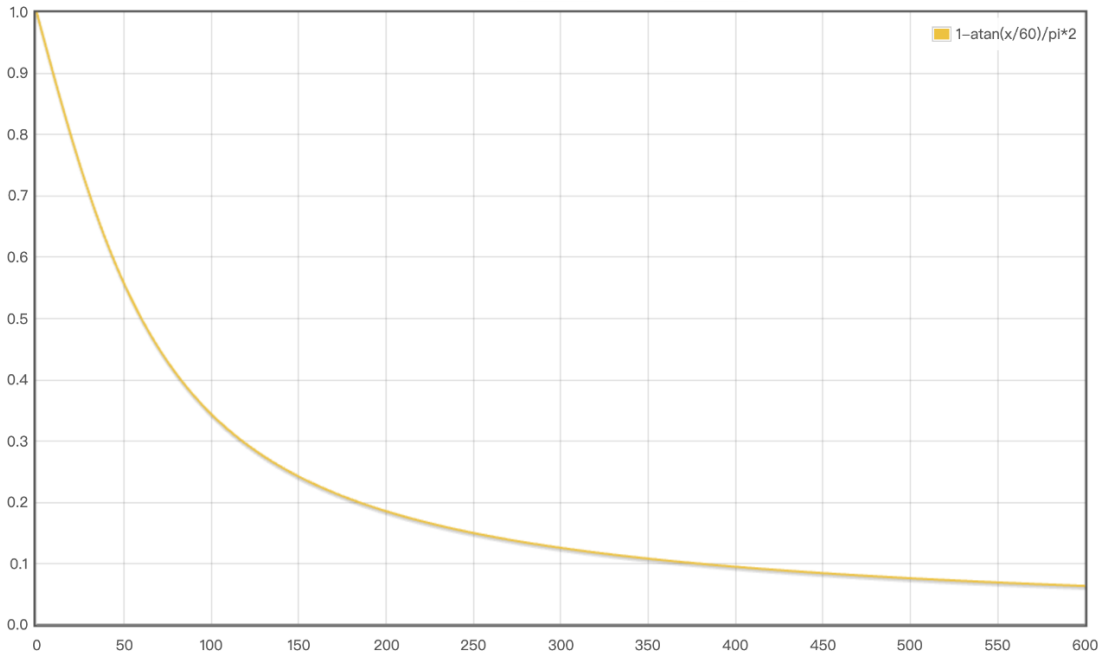
```
        double* total_weight) const {

        base::TimeDelta time_since_sample_taken = now - observation.timestamp();
        double time_weight =
            pow(weight_multiplier_per_second_, time_since_sample_taken.InSeconds());
...
}
```

反正切动态权重

y=arctan(x)反正切函数在第一象限的取值范围为 0~Pi/2，我们将 arctan(x)取反，向上平移 Pi/2，然后除以 Pi/2，函数曲线即可在第一象限随着 x 增大 y 的取值从 1 趋近于 0。我们还可以使用一个斜率系数来控制权重降低的趋势快慢，公式推导过程如下：

动态权重 = (Pi / 2 - arctan(abs(now - 数据采集时间) * 斜率系数)) / (Pi / 2) = 1 - arctan(abs(now - 数据采集时间) * 斜率系数) / Pi * 2；斜率系数为浮点型，取值范围为 0~1，系数越小，权重降低的越缓慢。

以斜率系数为 1/20 为例，对应的函数曲线如下：



和前文的半衰期动态权重相同，横坐标为数据采集时间距今的时间差，纵坐标为权重，随着时间差增大，权重趋近于 0，两种动态权重算法效果类似。

反正切动态权重的实现代码如下：

```
static double _nqe_getWeight(double targetTime) {
        ......
```

```
    double interval = now - targetTime;
    /// 曲率系数，数值越小权重降低的越缓慢
    double rate = 20.0 / 1;
    return 1.0 - atan(interval * rate) / M_PI_2;
}
```

从上图的代码实现可以看出，反正切相关的代码实现要简单很多，但是由于存在推导过程，所以理解起来比较困难，代码维护成本较高（数学功底对于程序员来说也是非常重要的），大家可以酌情自行选择。

携程最终采用的也是半衰期动态权重的方案，出于实时性考虑，最终线上验证后采用的衰减幅度为 0.3，衰减幅度为 60 秒，供参考。

2.2.3 RTT 指标加权中值计算

在确定了单条数据的权重之后，对于 RTT 的数值计算，我们第一个想到的是加权平均，但是加权平均很容易收到高权重脏数据的影响，准确性堪忧，所以我们改用了“加权中值”。

加权中值的计算方式是，将窗口内的数据按照数值大小升序排列，然后从头遍历数据，累加权重大于等于总权重的一半时，停止遍历，当前遍历到的数值即为最终的加权中值。

NQE 对于 TransportRTT 和 HttpRTT 处理，也是使用的这种方式，相关代码详见附录 4，部分核心代码如下：

```
std::optional<int32_t> ObservationBuffer::GetPercentile(
    base::TimeTicks begin_timestamp,
    int32_t current_signal_strength,
    int percentile,
    size_t* observations_count) const {

……
  // 此处的 percentile 值为 50，即取中值
  double desired_weight = percentile / 100.0 * total_weight;
  double cumulative_weight_seen_so_far = 0.0;
  for (const auto& weighted_observation : weighted_observations) {
    cumulative_weight_seen_so_far += weighted_observation.weight;
    if (cumulative_weight_seen_so_far >= desired_weight)
      return weighted_observation.value;
  }
  // Computation may reach here due to floating point errors. This may happen
  // if |percentile| was 100 (or close to 100), and |desired_weight| was
  // slightly larger than |total_weight| (due to floating point errors).
```

```
    // In this case, we return the highest |value| among all observations.
    // This is same as value of the last observation in the sorted vector.
    return weighted_observations.at(weighted_observations.size() - 1).value;
}
```

2.2.4 成功率指标加权平均计算

对于成功率，我们的 NQEMetrics 结构体内定义了单次成功状态 isSuccessed，单条数据的加权成功率为（NQEMetrics.isSuccessed ? 1 : 0) * weight，整体的加权成功率为加权成功率总和除以总权重。

相关代码实现如下：

```
extern double _calculateSuccessRateByWeight(const vector<CTNQEMetrics> &metrics, uint64_t types, const shared_ptr<NQEConfig> config) {
    ……
    uint64_t totalValidCount = 0;
    double totalWeights = 0.0;
    double totalSuccessRate = 0.0;

    for (const auto& m : metrics) {
        /// 过滤需要的数据
        if ((m.source & types) == 0) {
            continue;
        }
        /// 累计总权重和总成功率
        totalValidCount++;
        totalWeights += m.weight;
        totalSuccessRate += (m.isSuccessed ? 1 : 0) * m.weight;
    }

    /// 数据不足
    if (totalValidCount < config->minValidWindowSize) {
        return NQE_INVALID_RATE_VALUE;
    }
    if (totalWeights <= 0.0) {
        return NQE_INVALID_RATE_VALUE;
    }

    return totalSuccessRate / totalWeights;
}
```

2.2.5 引入成功率趋势提高实时性

网络质量识别不仅需要准确，实时性也非常重要，在网络质量切换时模型识别的时间越短越好。前文已经提到了 TransportRTT、HttpRTT、NetworkSuccessRate 三个核心指标的计算，但是在线上实际验证的过程中，我们发现在网络完全不可用成功率跌 0 后，识别模型对于网络状态的恢复感知很慢，原因是成功率的攀升需要较长的时间。

针对这个极端的 case，我们引入了一个“成功率趋势”的新指标，来优化模型的实时性，在成功率未达阈值当时有明显趋势时，提前切换网络质量状态。成功率趋势是指一段时间内成功率连续上升或者下降的幅度，浮点类型，取值范围-1 ~ +1。

成功率趋势初始值为 0，计算方式如下：

1）在每次更新成功率时，计算更新前后成功率的差值

如果差值为正，则成功率向好

- 如果当前成功率趋势值为正，则向好趋势持续，成功率趋势加上当前差值
- 如果当前成功率趋势值为负，则成功率趋势由坏转好，成功率趋势重置为当前差值

如果差值为负，则成功率向坏

- 如果当前成功率趋势值为正，则成功率趋势由好转坏，成功率趋势重置为当前差值
- 如果当前成功率趋势值为负，则向坏趋势持续，成功率趋势加上当前差值（负值）

2）当然还需要过滤一些毛刺数据，避免趋势变化过频

具体代码实现如下：

```
void NQE::_updateSuccessRateTrend() {
    auto oldRate;
    auto newRate;

    if (oldRate < 0 || newRate < 0) {
        _successRateContinuousDiff = 0;
        return;
    }
    auto diff = newRate - oldRate;
    /// 数据错误，不做处理
    if (abs(diff) > 1) {
        _successRateContinuousDiff = 0;
        return;
    }
    /// diff 小于 0.01，作为毛刺处理，不影响趋势变化
```

```
        if (abs(diff) < 0.01) {
            _successRateContinuousDiff += diff;
            return;
        }
        /// 计算连续 diff
        if (diff > 0 && _successRateContinuousDiff > 0) {
            _successRateContinuousDiff += diff;
        } else if (diff < 0 && _successRateContinuousDiff < 0) {
            _successRateContinuousDiff += diff;
        } else {
            _successRateContinuousDiff = diff;
        }
}
```

在网络成功率和成功率趋势的加持下，我们的识别模型实时性大幅度提升。我们控制相同请求频率和请求数据量，线下模拟弱网切换进行测试，测试结果如下：

- 单 RTT 识别模型，网络质量切换后识别较慢，且存在连续切换场景识别不出弱网的情况

- RTT+成功率模型，切换识别速度较单 RTT 模型提升约 50%，成功率跌 0 后的 Bad 切 Good 识别明显较慢

- RTT+成功率+成功率趋势模型，切换识别速度较单 RTT 模型提升约 70%，Bad 切 Good 识别速度明显提升

所以，最终携程弱网识别模型计算的指标有 TransportRTT、HttpRTT、NetworkSuccessRate、SuccessRateTrend（成功率趋势）四个。

2.3 结果输出

2.3.1 网络质量定义

识别模型对外输出的是一个网络质量的枚举值，Google NQE 对于网络质量的定义如下，源码详见附录 5：

```
enum EffectiveConnectionType {
  // Effective connection type reported when the network quality is unknown.
  EFFECTIVE_CONNECTION_TYPE_UNKNOWN = 0,

  // Effective connection type reported when the Internet is unreachable
  // because the device does not have a connection (as reported by underlying
  // platform APIs). Note that due to rare but    potential bugs in the platform
  // APIs, it is possible that effective connection type is reported as
```

```
  // EFFECTIVE_CONNECTION_TYPE_OFFLINE. Callers must use caution when using
  // acting on this.
  EFFECTIVE_CONNECTION_TYPE_OFFLINE,

  // Effective connection type reported when the network has the quality of a
  // poor 2G connection.
  EFFECTIVE_CONNECTION_TYPE_SLOW_2G,

  // Effective connection type reported when the network has the quality of a
  // faster 2G connection.
  EFFECTIVE_CONNECTION_TYPE_2G,

  // Effective connection type reported when the network has the quality of a 3G
  // connection.
  EFFECTIVE_CONNECTION_TYPE_3G,

  // Effective connection type reported when the network has the quality of a 4G
  // connection.
  EFFECTIVE_CONNECTION_TYPE_4G,

  // Last value of the effective connection type. This value is unused.
  EFFECTIVE_CONNECTION_TYPE_LAST,
};
```

Google 枚举定义的最大问题是理解成本比较高，其他开发同学看到这个所谓的“3G”、“4G”，他依然不知道网络是好是坏，是不是他认为的“弱网”。

所以我们在定义接口的时候，对于枚举的设计考虑最多的就是理解成本，结合开发同学最想知道的“是不是弱网”，我们的接口定义如下：

```
typedef enum : int64_t {
    /// 未知状态，初始状态或者无有效计算窗口时会进入此状态
    NetworkQualityTypeUnknown = 0,
    /// 离线状态，网络不可用
    NetworkQualityTypeOffline = 1,
    /// 弱网状态
    NetworkQualityTypeBad = 2,
    /// 正常网络状态
    NetworkQualityTypeGood = 3
} NetworkQualityType;
```

这样是不是看起来简单明了多了，我们接下来就讲讲怎么计算得出这几个枚举的结果。

2.3.2 网络质量计算方式

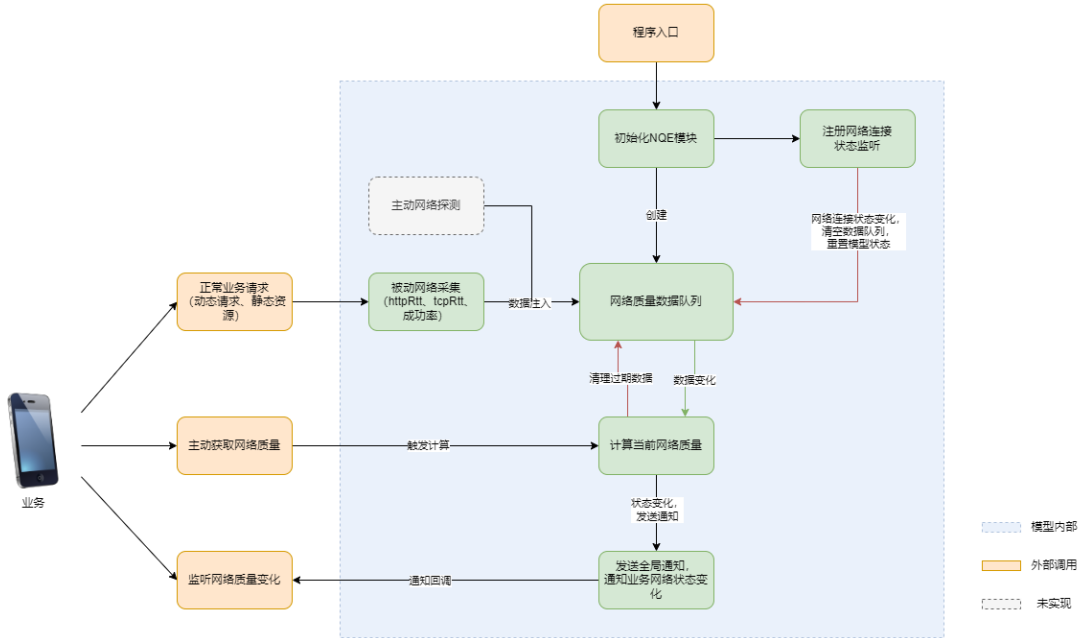
NetworkQualityTypeUnknown 是在初始化或者网络切换后的一段时间内，数据不足无法得出网络质量，会进入此状态。

NetworkQualityTypeOffline 的触发条件很单一，就是操作系统识别到无网络连接，具体的获取方式由各平台自行实现，例如 iOS 可以通过 Reachability 获取，官方 Demo 详见附录 6。

NetworkQualityTypeBad 也就是我们最核心的“弱网”状态，计算方式是上文提到的 TransportRTT、HttpRTT 两个指标任一指标触发弱网阈值，或者 NetworkSuccessRate 和 SuccessRateTrend 同时满足弱网阈值。

NetworkQualityTypeGood 是指正常的网络质量状态，上述三种网络质量类型讲完后，这个类型就简单了，即非上述三种情况的场景，归类到 Good，这也是设计上占比最高的网络质量类型。

识别模型的运转流程如下：



- 数据队列在初始化和网络连接状态变化两个时机会被重置，充值后网络质量类型进入 NetworkQualityTypeUnknown

- 网络连接状态进入无连接状态时，数据队列被清空，网络质量类型直接进入 NetworkQualityTypeOffline，在网络类型变为可用类型前，数据队列不接受数据注入，且不进行计算

- 数据队列的数据变化触发质量计算，出于资源开销考虑，要限制计算的频次，我们采用计算间隔和新增数据量两个阈值限制，计算间隔大于 60s 或者新增数据量

超过 10 条才会触发计算；同时也暴露了对外接口，业务可按需强制刷新计算结果

- 关于主动网络探测，可结合自身的业务需求按需实现，目前携程的 APP 在使用时网络数据更新比较频繁，无需补充主动探测数据

2.3.3 弱网阈值制定

模型核心的计算逻辑，就是将加工后得到的各网络指标与对应的弱网阈值进行对比，从而获得是否进入弱网的结果，关于弱网阈值的制定上，我们经历了如下两个阶段：

第一阶段，主要参考 NQE EFFECTIVE_CONNECTION_TYPE_2G 的阈值定义

- HttpRTT > 1726ms
- TransportRTT > 1531ms
- 成功率按照内部讨论的预期设置为 NetworkSuccessRate < 90%
- 成功率趋势阈值设置为 SuccessRateTrend < 0.1，即成功率连续向好增加超过 10pp，即使成功率小于 90%，也从 Bad 切换为 Good，这个指标主要是为了提升 Bad 切 Good 的速度

第二阶段，我们通过线上的网络质量分布监控，和一些具体 case 的分析，不断迭代我们的阈值，我们需要制定一个识别准确率的指标来指引阈值的调整工作，达到逻辑准确与自洽。

理论上，我们希望识别模型的入参与当下计算出的网络质量类型所匹配，例如当前注入 NQEMetrics 数据的 HttpRTT <= 1726ms ，那我们预期当前计算出的网络质量类型就是 Good。但是弱网的决策逻辑是相对复杂的，需要考虑到各种因素，以下两点会造成弱网状态下的入参数据不一定符合弱网阈值定义：

1）弱网的计算是对过去已经发生网络行为的分析，具有一定的滞后性，所以在识别结果切换附近，必然有部分的原数据已经满足下一阶段的网络质量定义

2）弱网的决策是对多个指标的复合计算，所以在识别到弱网状态时，不一定所有的指标原数据都符合弱网定义，比如由 HttpRTT 触发弱网时，当前的 TransportRTT 数据可能表现良好

终上两点原因，弱网分类下必然有一定的非弱网数据，这里的误差数据占比与识别准确率负相关，误差数据占比越低，识别准确率越高。所以想到这里，我们的模型识别准确率的指标计算口径就有了：

模型弱网识别准确性 = 100% - 弱网状态下不符合弱网阈值定义原数据占比

有了这个指标指引，我们在模型上线后进行了数个版本的数据统计，通过各指标阈值的微调和 case by case 解决异常场景，误差数据从刚上线的 15%+降低到 10%以下，即模型识别准确率优化至 90%以上。

最终携程 90%准确率的模型对应的弱网阈值如下（不同业务场景的网络请求差别较大，仅供大家参考）：

- HttpRTT > 1220ms；这个值是线上 HttpRTT 的 TP98 值，与弱网占比相近
- TransportRTT > 520ms；同线上 TP98 值
- NetworkSuccessRate < 90%
- SuccessRateTrend < 0.2；之前的 0.1 导致模型的结果切换过于频繁，最终调整到了 0.2

Tips：对于类似携程这种自定义的弱网识别模型，弱网标准也是考虑业务现状的定制标准，所以不需要太多和外部的弱网标准对齐，重点是自洽和符合业务预期。

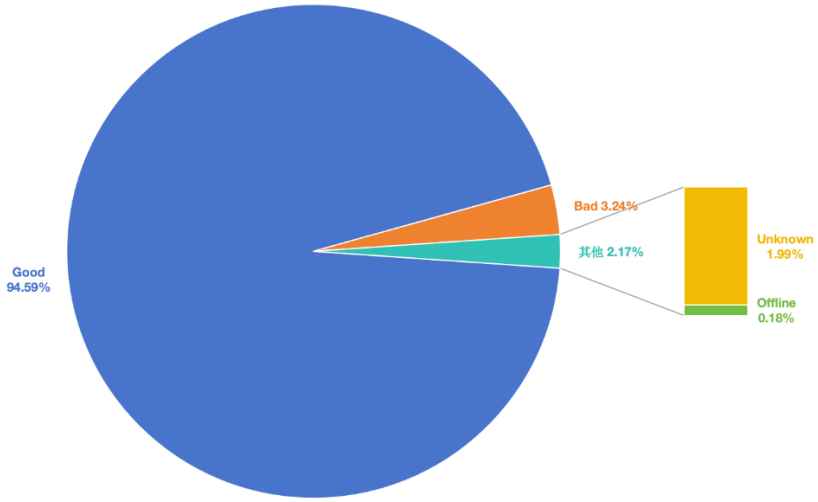
三、落地效果

考虑到识别模型要支持多平台（iOS、Android、Harmony 等），所以我们在一开始实现方案时就采用了 C++作为开发语言，天然支持了多平台，各平台只需要实现上层数据采集和注入模型的少量逻辑即可完成模型的接入。相同的代码实现和弱网标准，也方便我们在不同的平台间直接对标数据，发现各平台的问题针对性优化。

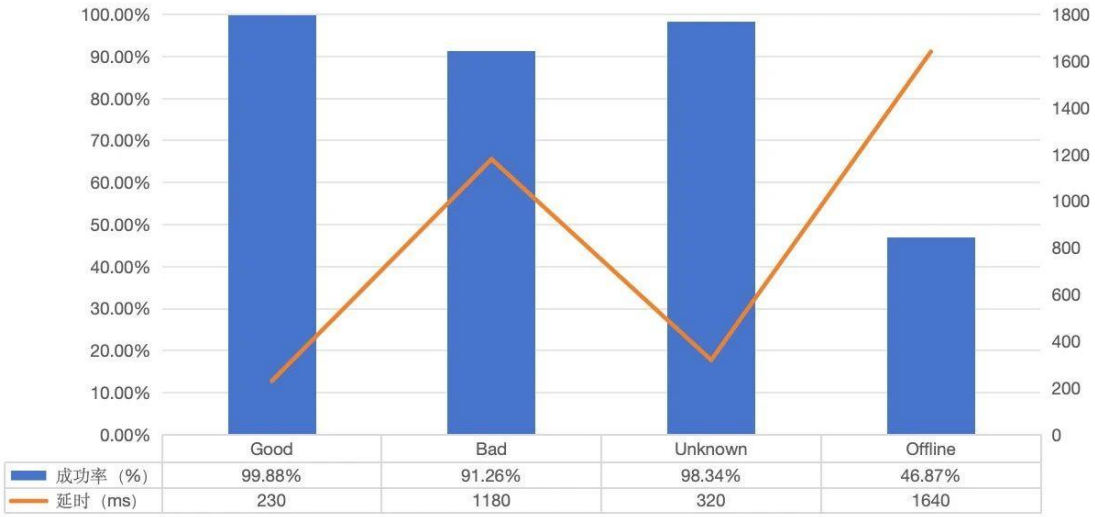
目前携程的网络质量识别模型，已经在 iOS、Android 平台完成接入并大面积投产，网络质量数据与集团的 APM 监控平台打通，形成了携程官方统一的网络质量标准，在网络排障、框架网络优化、业务网络优化等多种场景下扮演重要角色，弱网优化相关的内容我们会在后面相关的专题内继续分享，此处不再赘述。

最终网络质量相关的分布数据如下（数据为实验采集，不代表携程真实业务情况，仅参考）：

网络质量分布：



各网络质量下对应的请求性能数据：



四、未来展望

网络质量识别模型的完成只是我们网络优化的开始，后续还有很多的工作需要我们继续努力，未来一段时间我们会从以下几个方面继续推进：

1）持续推进各平台、各独立 APP 的网络质量识别模型接入，完成携程终端全平台的网络质量模型覆盖

2）做好识别模型的防劣化工作，解决各业务场景的 bad case，坚守现阶段识别准确

率和实时性的标准水位

3）推出携程内部的“网络性能白皮书”，从 APP、系统平台、网络质量、成功率、全链路耗时等各维度解析公司内部各业务线的网络表现，形成内部的网络性能数据基线，为业务优化提供参考

4）借助现有的弱网标准和识别能力，从网络框架侧和业务侧两个不同的角度进行弱网优化，提高整体网络表现；当下海外市场是业务发力的重点，海外场景的网络表现也明显弱于国内，我们会针对海外场景从弱网的角度进行重点优化。

携程目前已经针对弱网场景推出了一系列优化策略，部分策略已经取得非常不错的收益，后续我们会继续推进，也会持续分享输出。

架构

为业务系统赋能，携程机票最终行程系统架构演进之路

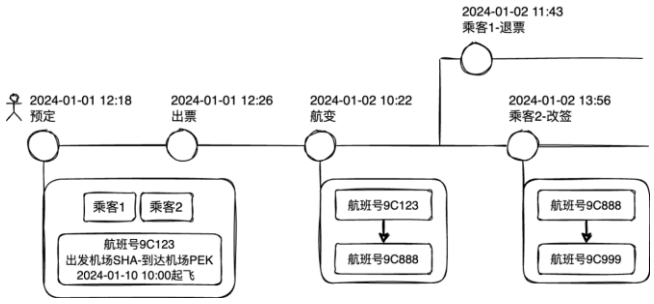
【作者简介】
Stephen，携程资深后端开发工程师，专注新技术挖掘，持续推动业务创新。
Scott ，携程资深研发经理，负责订单系统架构升级和优化。

一、背景

携程机票订单系统是由多个业务子系统组成，包括出票、改签、航变等等，获取订单行程信息复杂度较高。

例如：用户预订了一个包含了 2 个乘客的机票订单，该订单发生了航变，其中用户 A 选择了退票，用户 B 选择了改签。

业务系统需要获得该订单最新的行程信息以及行程变化轨迹，以进行展示和进一步处理。

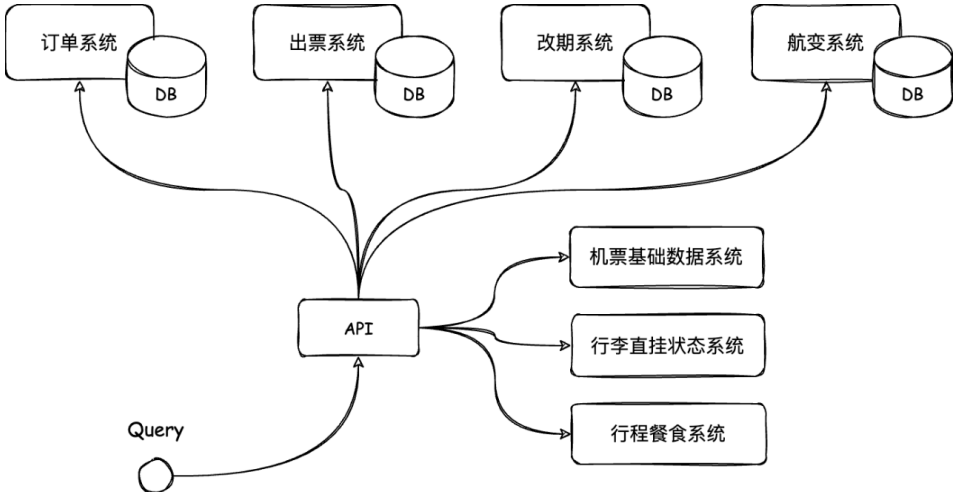


上述例子用户的最新行程信息为：

乘客 1：航班号 9C888，SHA-PEK，已退票

乘客 2：航班号 9C999，SHA-PEK，已改签

历史的系统设计需要通过 API 对各业务子系统的数据进行实时的聚合和计算，如果要获取上述例子的最终行程与轨迹，需要至少调用订单、出票、改期、航变系统等，流程复杂且耗时高，并且针对一些复杂的业务场景还可能导致错匹配、漏匹配等问题。



总结下来有如下几个问题:

- 数据私有（分散），数据模型不统一
- 按照时间线进行聚合的难度大，需要动态计算，耗时长
- 数据存储周期不一致，完整性不高
- 数据分析困难，报表逻辑复杂

二、目标

总的来说，我们需要设计一个用户行程系统来满足以下要求:

- 完整准确的行程信息

信息丰富完整，并保证更新及时、准确

- 使用便利

一站式获取，使用方效率提升，方便使用方快速接入

- 性能可靠

系统性能良好，可靠性高

- 提升业务系统自动化率

提升自动化率，上线灵活

- 快速实现复杂业务流程

对于大量动态数据的分析与过滤需要快速实现并上线

三、实施方案

3.1 设计思路

Q1：系统需要提供什么样的能力？

1）提供准确的用户最新行程信息

用户和相关的业务系统需要及时和方便的获取到完整、准确行程信息

2）输出历史行程变化轨迹

对于退票等场景，需要了解用户完整的行程变化轨迹，以便于自动化处理相关数据

3）通过行程信息进行模糊匹配

对于航变场景，航司通知某个具体航班发生了变化，系统需要通过这些信息匹配到对应的订单并进行后续的处理

Q2：如何确保信息的丰富和准确？

1）在丰富性方面，可以接入大量的数据源并提供便捷的接入方式，及时有效的采集数据，提升系统数据的完整性

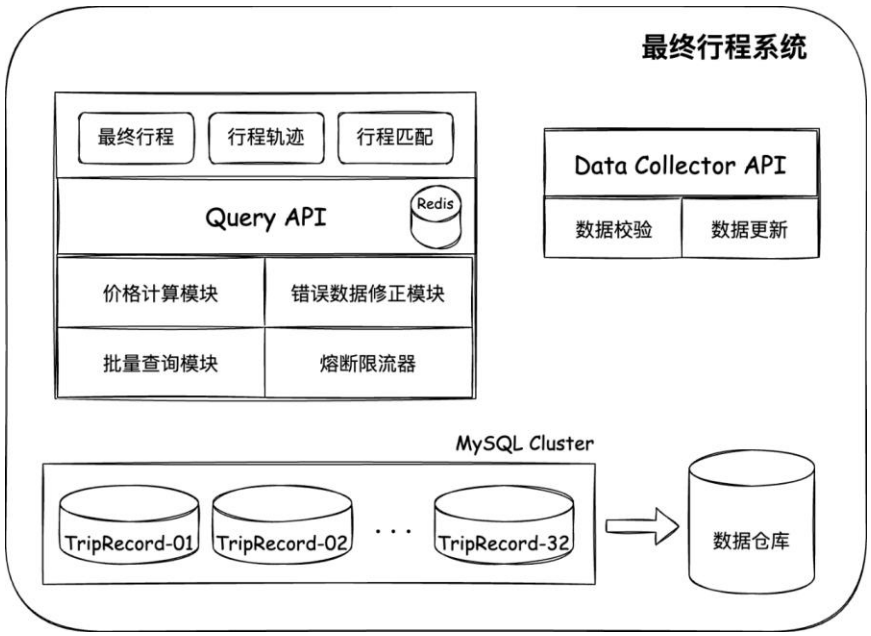
2）在准确性方面，可以采用主动 + 被动等方式，多维度的对数据进行校验、修复，提升数据的准确性

Q3：如何提升系统的稳定性和可扩展性？

1）通过分布式缓存、结构化并发等技术提升系统的性能与稳定性

2）通过数据库的 sharding、数据仓库的赋能等方式提供在线和离线的数据处理能力，进一步扩充数据的应用场景

3.2 系统架构图



最终行程系统主要有以下几个方面组成：

1）最终行程数据通知与更新系统

即上图中的 Data Collector API，通过收集各种来源，如订单库、出票系统、改签系统等的数据，更新或者落地在最终行程系统数据库中。同时在落地的时候也会进行被动 + 主动相结合的数据校验机制，保证数据的准确性。

2）最终行程查询系统，即上图中的 Query API，其中包含三大功能与若干个模块

- 最终行程查询，对外输出该订单的最终行程信息，该接口流量最高，包含有缓存组件、熔断器、限流器等，保障其性能的稳定；
- 行程溯源轨迹查询，对外输出该订单下所有行程变化的历史轨迹，使用方可以通过该接口拿到这个订单的行程关系图，感知所有变化轨迹；并且整合了价格计算模块、错误数据修正模块；
- 行程匹配查询，通过给定的行程要素条件，匹配能够对应上的最终行程记录，并支持批量查询；

3）数据存储架构，通过分库提升数据库的水平扩展能力，并且结合数据仓库为业务赋能

3.3 信息丰富性

支持多种更新机制，方便接入多种类型的通知方，提升信息的丰富度，目前已经接入了出、退、改、航变、票号中心等 22 个数据源。

策略 1: 系统主动通知，适用于对于数据新鲜度要求较高的场景，查询性能较好

策略 2: 消息通知消费，适用于数据新鲜度要求不太高的场景，通过反查保证数据最终一致，方便系统解耦

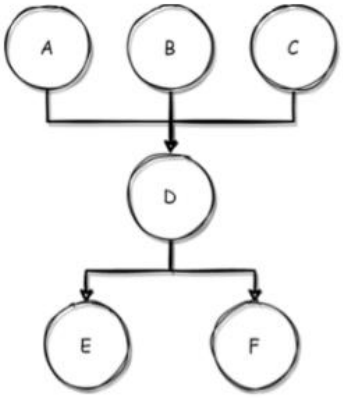
策略 3: 实时查询，适用于数据变化非常频繁，新鲜度要求高的场景；减少了数据冗余，但是在查询和使用上存在依赖

策略 4: 动态数据的过滤通知，适用于存在规则变更，但变化维度和订单维度不同，需要扫描海量数据来获取更新记录的场景

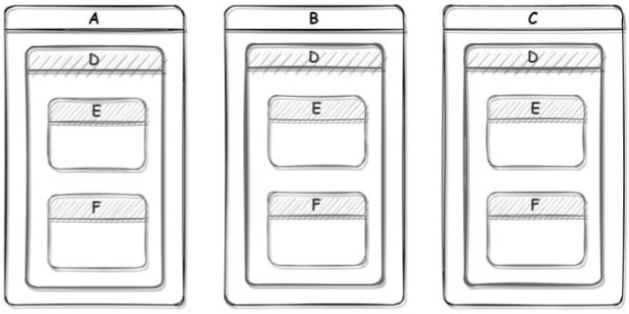
3.4 便利度增加和业务提升

3.4.1 降低溯源接口接入复杂度

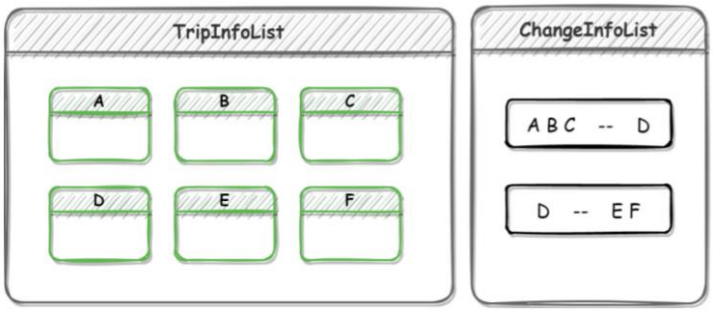
溯源轨迹接口对于行程关系图的输出形式，对于使用方的便利度影响非常大，比如如下的行程关系图。



历史的输出形式为一种无限层级的树形结构，这样的结构虽然能对向下的溯源查询以及对一变多的行程变化关系提供支持，但是对于向上的溯源查询、多变一、多变多的行程变化关系不友好，许多使用方都需要使用 DFS 等算法来解析数据，不够简洁易用，容易出错；并且树形结构已经不能直观的反映出类似二变一（中转变直飞）的行程变化场景，而且这样的结构还会出现数据的冗余，如下图所示：

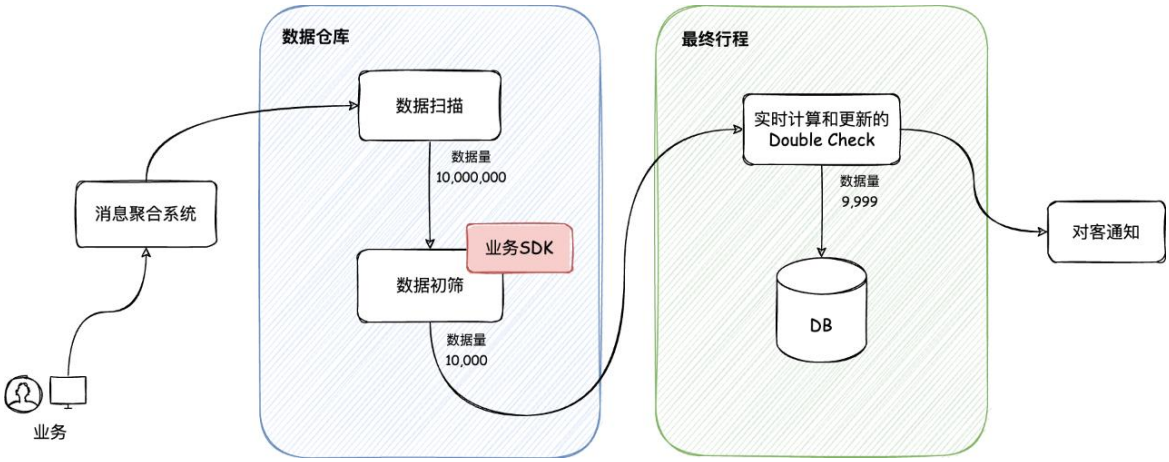


基于以上的情况，新溯源接口选择了类似图的邻接矩阵来表述行程溯源变化关系，通过 TripInfo 节点来表示顶点数组，平铺出行程溯源关系图中各个节点的行程信息；通过 ChangeInfo 节点来表示边数组，主要描述行程变化关系。这样的描述更加通用、结构清晰并且对使用方更友好。



3.4.2 支持大量动态数据的扫描与过滤

在实际的业务场景中需要维护这样一部分数据，它会发生变化，但引起变化的规则维度与订单维度不一致，所以需要扫描海量数据来获取需要被更新的记录。同时，扫描依赖的数据可能还需要跨库才能拿到，按照现有的数据库结构实现起来非常复杂。通过调研，最终采用数仓并结合业务 SDK 过滤的动态数据主动更新机制，实现了业务场景主动更新与通知的功能，该流程有如下几个特点：



- 轻松整合所有依赖数据项，通过数据仓库的大数据分析能力，可以轻松整合所有依赖的数据项

- 对数据进行筛选，在数据仓库处理的流程中，添加了业务 SDK 的过滤机制进行数据的初筛，将海量数据进行过滤，并结合 Double Check 机制进行进一步的筛选，得到真正受影响的记录

- 触发消息的聚合机制，同时考虑到了业务误操作后又修改一次的情况，所以增加了消息聚合机制，聚合一段时间的消息后再真正触发数仓进行处理

该流程具有很强的通用性，通过简单替换不同的 SQL 语句，切换不同的 SDK，就可以轻松将该流程移植到其他业务项目中，实现了功能的快速上线。

3.5 性能优化

3.5.1 提升数据库的水平扩展能力

最终行程系统在之前使用的是单库存储，但是随着数据量的不断增加，当业务信息扩充时，新增数据字段在数据库层面上变的难以操作；并且如果按照业务期望的存储时间，硬盘使用率会过高，造成了存储瓶颈。

经过调研，决定对最终行程数据库做 Sharding 处理，将数据平均分配到多个分片就可以满足存储要求并兼顾性能指标。

1）数据切分，基于最终行程数据特性，即订单号访问占比较高，同时在订单号分布均匀的前提下，最终采用了订单号对数据库总分片数取模的方式，以保证数据分布的均匀性。

2）数据兼容，对于 sharding 库和非 sharding 库双写新数据的操作，并考虑数据库存在异常的情况，需要增加异常补偿处理机制；并且对于历史存量数据，也进行了分批次的数据迁移以及补偿功能，同时为了保证数据一致性，在迁移完成后也进行了多批次的数据对比与接口对比工作，保证 Sharding 数据的准确性和可靠性。

3）查询性能，多分库的查询性能是分库存在的典型问题，对于最终行程来说，采用非订单号查询操作，分库后就涉及到多个分片的 All Shard 查询，极大地增加了数据库压力和影响查询性能。经过数据统计，分析得到特定的业务字段查询其实就涵盖了非订单号查询的大多数，从而增加其二级索引表就可以有效解决 All Shard 查询性能的问题。

3.5.2 接入 Redis 缓存提升系统性能

总体上采用先操作数据库，后删除缓存；先查询缓存，查询不到缓存则查询数据库，并回填缓存的方式进行处理。

1）提升新鲜度，在行程更新流程时、接收 BinLog 消息时、接收业务变更消息时都会将缓存删除。

2）采用分级储存查询的模式，查询时根据调用方所需的数据级别进行获取，缩小 Redis 获取数据的大小，减少网络开销。

3）异步回填，启用专用的线程对缓存数据进行异步回填，这样可以不拖累查询请求本身的耗时。

4）优化缓存容量，对 Json 序列化器定制规则，不输出值为 null 的字段；将序列化对象中的字段通过@JsonProperty 注解取一个简短的别名，来简化 Json 字符串 Key 的大小；使用 Zstd 压缩算法对序列化后的数据进行压缩；通过前期调研命中率和生存时间的关系，得出达到预

期命中率的最小缓存生存时间，从而进一步减少 Redis 的容量。

3.5.3 结构化并发在匹单接口中的探索

最终行程匹单接口允许使用方传入多组条件进行匹配，接口内部对于这多组条件采用的是 for 循环的方式顺序执行的，存在并发改造的空间；且匹单接口操作数据库存在多 shard 查询的情况，对于多 shard 查询，Dal 底层会使用线程池并发调用，对线程的开销较大。综合上述问题，并结合近期发布的新的长期支持版本 JDK21，发现了其预览功能中的结构化并发比较适用于匹单场景的优化。

1）简化多线程编程，增强可观察性。

一般而言，如果我们想要实现并发操作，需要使用异步编程的方式来实现，但是使用这样的方式对于代码阅读性和调试来说都比较差。在目前的多线程开发中，常用的方式是使用 CompletableFuture 的级联方式编写。与单线程的代码相比，这样的写法并不直观，并且“任务终止不干净”和“等待超过必要时间”的问题仍然存在，如果要解决这些问题还需要自己实现一系列模版代码，费力度大大增加。

而结构化并发的一大特点就是让开发人员以类似单线程的方式来编写多线程代码，他引出了一个结构化任务作用域（Scope）的概念，在这个作用域中创建并执行任务，这些任务的生命周期都由作用域来负责管理，开发人员可以不用关系细节问题。对于作用域的任务使用 try-with-resources 块，如果在执行中出现错误，会自动调用 StructuredTaskScope 的 shutdown 方法来终止执行，调用 shutdown 方法会阻止新任务的执行，同时取消正在运行中的任务。

```
int doAction(String input) {
    try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
        Subtask<Integer> v1 = scope.fork(() -> callOp1(input));
        Subtask<Integer> v2 = scope.fork(() -> callOp2(input));
        scope.join().throwIfFailed();
        return callOp3(v1.get(), v2.get());
    }
}
```

2）使用虚拟线程解决阻塞问题。

StructuredTaskScope 底层默认采用了虚拟线程进行实现，在我们原来的认知中，线程的使用都是昂贵的，而虚拟线程是 JVM 中 Thread 类的实现，它是轻量级的，当使用虚拟线程进行代码执行时，如果遇到阻塞操作，便会释放掉载体线程；并当该阻塞操作可用时，虚拟线程又将被安排在载体线程上去继续处理执行。即在虚拟线程中，阻塞不是问题，因为阻塞时底层的载体线程已经被释放了

虚拟线程和结构化并发的组合将非常强大，虚拟线程使阻塞不再是一个问题，而结构化并发为我们提供了更简单的多线程编写方案，以更直观的方式处理异步编程。

3.6 优化前后数据支撑

- 数据库 QPS 降低 30%
- 数据库 CPU 平均利用率下降 20%
- 平均响应时间降低 40%，P95 降低 30%
- 减少机器线程数 41%，CPU 利用率降低 25%，显著减少机器压力
- 快速支持了业务功能，人力成本节约至少 50%以上

四、后续规划

1）易用性优化

增加行程变化订阅通知机制，进一步提升易用性。

2）可靠性与性能提升

- 细化熔断和降级的策略
- 和框架团队协作，积极推广新技术在生产系统上的规范化落地
- 探索新的数据库结构与数据库选型，提升关系链路的存储能力

3）可视化

实现整体客人行程的可视化界面，依托最终行程数据的力量，帮助业务/产品开发更快了解到订单全貌，帮助提升问题解决效率。

领域化、中台化和多 Region 化，携程账号系统演进之路

【作者简介】Scai，携程高级研发经理，多年深耕于账号中台，持续推进中台的技术架构演进及性能优化。

一、前言

在互联网早期时代，账号系统的功能非常广泛，包括账号管理、登录认证相关能力以及维护各类用户信息，比如头像、昵称、积分、等级等。随着业务的发展，每个功能逐渐分化出自己的需求和架构侧重点，独立出各自的领域服务也成了业界共识。

本文分享的账号系统，指的是提供用户账号管理、登录认证相关能力的系统。介绍了携程在不断发展的过程中，账号系统在领域化、中台化和多 Region 化方向上的演进、探索和一些思考。

二、领域化

微服务迅猛发展阶段，账号系统分裂出了很多应用。比如，专门支持三方登录的应用，专门保存账号实名信息的应用，针对不同平台的接入应用。一开始确实可以满足迅速开发上线的需求，当应用裂变到几十个的时候，应用分层不合理，领域逻辑不内聚带来的问题逐渐显现出来：

1）用户请求会经历多个应用之间的 RPC 调用，性能和稳定性受影响。

2）操作无法原子性，易出现脏数据。

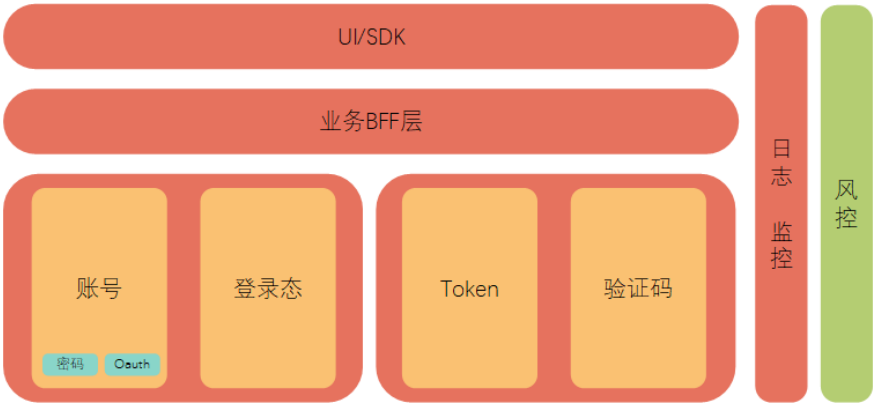
3）应用过多，开发、运维、测试范围大，影响效率。

领域化是对账号系统的全面重构，有以下两个目标：

1）合理划分领域，逻辑内聚。

2）改造需要对业务透明。

2.1 全面梳理，重新划分领域



账号系统功能主要分为 3 个类型：

1）核心功能：负责管理和维护账号系统的核心功能和数据。由于涉及到用户的核心数据，相关插入、变更接口只可暴露给业务 BFF 层。

- 账号领域：包括账号信息查询；账号注册/注销；手机、邮箱、三方等数据的绑定/解绑；openid 的生成；密码的管理和认证等功能。
- 登录态领域：负责登录态生成、验证、续期、踢出、过期删除等功能。
- 日志监控领域：负责记录账号相关业务埋点和日志。

2）辅助功能：不仅可以被账号相关业务依赖，也可以开放出去供类似业务场景的接入。

- Token（临时凭证）领域：负责 Token 的生成、验证、过期删除等功能。
- 验证码领域：负责验证码生成、发送、验证等功能。

3）接入功能：负责账号相关的业务功能接入，包括端上接入和内网服务接入。

- BFF 层：主要负责各类业务逻辑的组合（注册、登录、改绑手机等）以及接入方权限的控制。
- 前端 UI、SDK：前端显示 UI 以及提供出去供业务接入的 SDK。直接对接 BFF 层。

其他一些业务中必须依赖的模块，如风控模块，依赖公司相应团队提供的能力即可。

2.2 读写对比，透明改造

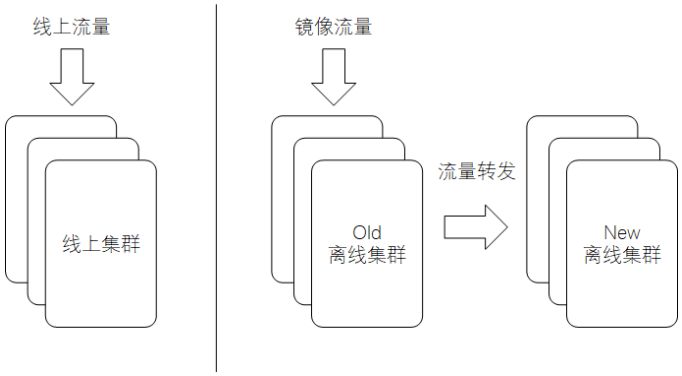
账号系统非常核心，上游是公司的各类业务，依赖方非常多，牵一发而动全身。同时，业务也在急速发展，不会停下来等待系统的改造。对账号系统的改造无疑是在快速开动的汽车上更换轮胎。因此，对账号系统的改造需要一套完整的比对流程，需要满足两个条件：

1）完整性。比对需要覆盖 100%的场景，避免场景遗漏。

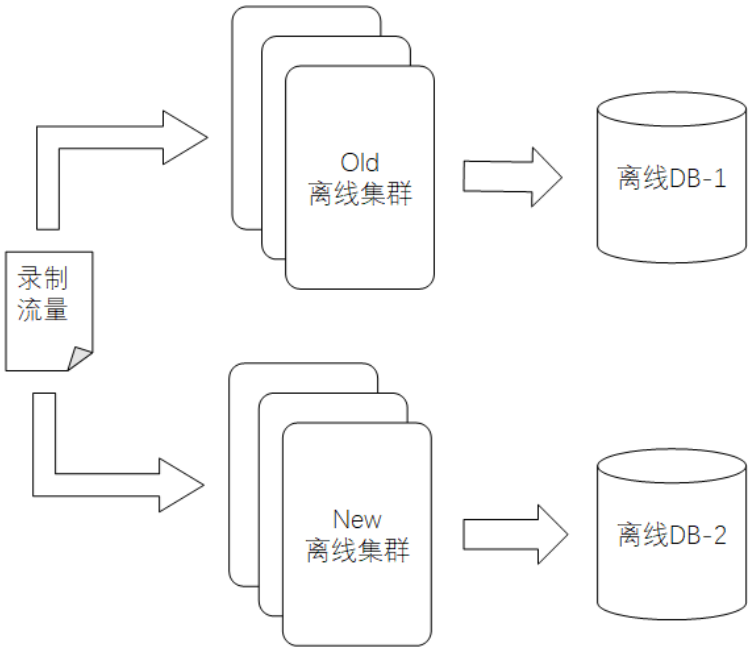
2）隔离性。比对需要在离线集群和存储上进行，避免对在线系统造成影响。同时，需要屏蔽掉离线集群不必要的对外请求，以免对下游造成影响（包括 RPC 调用，消息，监控数据等）。

在此基础上，完成了账号系统的读写比对流程：

读对比：转发-比对。利用镜像流量的能力，将镜像流量导入离线 Old 代码集群。Old 代码集群在处理流量的同时，会转发到离线 New 代码集群，完成接口返回数据比对。



写比对：录制-回放-比对。提前录制生产环境的流量，记录输入、输出和发出的消息等数据；分别部署 New/Old 代码离线集群和两套相同的离线 DB（里面数据为同一时间点的 Snapshot）；将录制好的流量（DB Snapshot 时间点后）回放到两套集群上，比对输出、存储、发出来的消息等数据，确保 New/Old 集群和录制的结果三方一致。



完成了领域化改造后，核心数据的变更沉淀到对应的领域服务中，相关操作可以满足原子性，

避免脏数据的产生；应用减少以及引入 BFF 层，减少了应用间，用户端和服务端的交互次数，提升了系统稳定性，提升了开发、运维、测试的效率。

三、中台化

和大部分互联网公司一样，随着集团的发展，会出现不同的品牌，需要一套独立的账号体系。也有业务团队会将自己研发的账号系统交给账号团队统一管理。如果账号系统没有做好中台化的准备，势必会在接入的过程中手忙脚乱。不仅代码中会存在大量的判断逻辑，接入时间也会很长，甚至可能无法满足业务的需求。中台化的改造需要考虑以下三点：

1）降低改造复杂度

- 减少系统的架构改造复杂度
- 降低业务接入的复杂度

2）配置化

- 将需求抽象为功能，减少对业务需求的定制化开发
- 简单配置即可快速接入

3）提供多样化的接入方式，以满足不同业务方的接入需求

3.1 降低改造复杂度

中台化改造过程中，账号体系 ID 是最核心的概念。

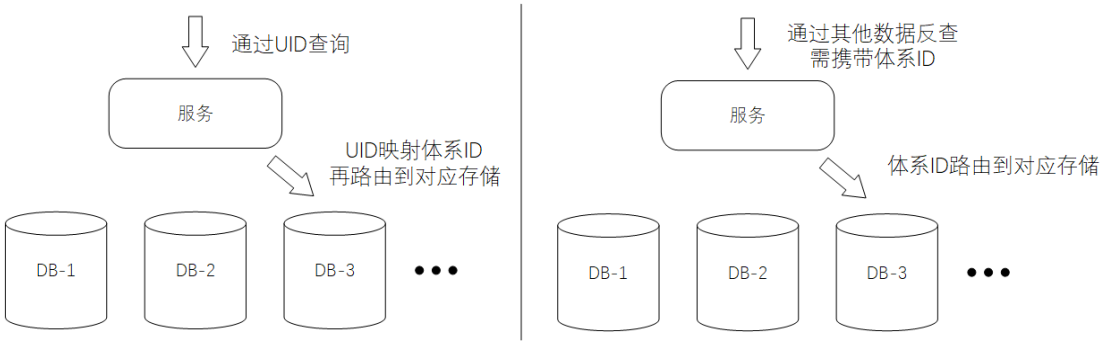
- 标志着账号所属的业务体系，相互之间隔离

- 控制账号体系的策略和权限

- 路由每一个账号体系对应的存储

因此，对于账号相关查询请求，如果不知道账号所在的体系 ID，就无法找到对应的存储。要么进行全存储查询，要么需要一个大表存储 UID 到体系 ID 的映射关系，这会引入额外的依赖，提高成本的同时，也会使得系统变得复杂。

另外，要求所有上游新增一个参数需花费大量的资源推动。



UID 全局唯一，并且通过编码区分不同的体系 ID，可以大大降低改造和接入的复杂度

- 新接入的账号体系，通过 UID 的编码可以快速判断账号体系 ID。
- 对于存量 UID，默认属于最早的账号体系。

同时，UID 全局唯一也可以提高排障和 TS 时的效率（不用反复确认某一个 UID 属于哪个账号体系）。

当然，一些不通过 UID 进行的查询接口，如通过手机号查询账号的场景，还是需要业务方传递体系 ID，但通过这样的设计已极大的缩小了需要改造的范围。

3.2 配置化

账号中台化后主要提供以下能力：

1）账号管理：管理账号的完整生命周期，包括注册，验证，注销。支持账号绑定手机号，邮箱，第三方账号，以及对应属性的变更、解绑操作。管理账号密码，支持多种加密逻辑。管理 Oauth 相关数据。

2）多样化登录方式：包括账号密码登录，手机验证码登录，手机一键登录，扫码登录，社交账号登录等登录方式。特别的，在社交账号登录方式中，账号系统已完成了与多个平台的对接，常见的比如微信、支付宝、QQ、微博、华为等。

3）登录态管理：包括登录态的生成、验证，登录态信息维护，按需续期、踢出等功能。

4）安全&监控体系：账号中台具有完善的日志体系，并已完成对接前端滑块和后端实时风控。

在中台化建设的过程中，虽然已经全量梳理了中台应该提供的能力，仍然会有新的需求需要支持。在接到新的需求，而现有的功能无法支持的时候，不要急于解决本次需求，需要思考本次需求涉及的具体功能，从而在实现的过程中避免定制化逻辑，沉淀为中台的新能力。

比如，某次需求为：一个账号体系全平台需要保证登录态是唯一的，即新的登录产生后，会

踢出之前的登录态。可以抽象为需要中台提供对某一个账号体系的登录态数量进行控制。进一步的，可以按照平台（App、小程序、H5 等）分别进行控制。

```
"platformConfigs": [
    {
        "platform": "h5",
        "maxNo": 5
    },
    {
        "platform": "miniapp",
        "maxNo": 4
    },
    {
        "platform": "pc",
        "maxNo": 2
    },
    {
        "platform": "default",
        "maxNo": 3
    }
]
```

中台化建设完成后，不同的功能都可以通过配置进行控制，也可以对每一个功能进行细节上的调整。同一体系的配置放置在同一配置文件中，便于维护。如果没有特别的需求，直接使用默认配置接入即可。

3.3 多样化接入方式

为了适应业务多样化的接入需求，中台提供了 3 种接入方式：

1）UI 接入：在携程的主 Web 站点、App 和小程序，统一使用中台开发的前端界面，业务方按需拉起。

2）前端 SDK 接入：有少量定制需求，如显示的 LOGO 需要调整，协议需要变更等，可以使用中台的前端 SDK 接入。此 SDK 已包含所有的流程逻辑，接入时仅需替换掉对应的元素。

3）后端对接：若业务方有过多的与业务藕合的逻辑，则不适合将逻辑放在中台。此时，业务方可以自行开发一层 BFF，利用中台 BFF 层和辅助系统（验证码、Token）提供的能力，

组合出合适的业务逻辑。

中台化改造完成后，新账号体系需要申请接入时，仅需选择需要的能力，中台通过调整配置，小时级即可完成接入。大大减少了新增一套账号体系的支持成本。

四、多 Region 化

两地三中心是近期比较热门的部署方案，一方面可以更好的应对城市级别的故障，另一方面可以更好的服务当地用户（例如，一个产品定位于服务西部用户，相应的应用和数据部署在西部城市可以提供更好的用户体验）。账号作为业务的基石，需要第一时间完成多 Region 部署，为各业务的部署做好准备。

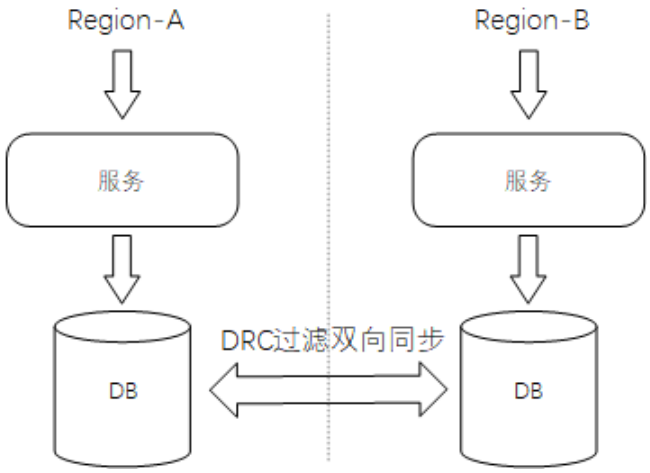
多 Region 部署，账号中台制定了两个目标：

1）数据支持多 Region 部署，请求正确路由。结合业务需求，账号中台的应用和数据按需部署到指定的 Region 中，相应的用户请求需要准确的落到对应的 Region。

2）架构需要同构部署，不能因为多 Region 部署引入开发和维护上的额外成本。

4.1 用户识别及请求路由

为了满足数据不同 Region 部署的需求，需要对用户数据进行识别并打标，利用公司 DB 数据同步组件（DRC）进行带过滤的双向同步（有的数据仅需要本 Region 使用，会过滤不进行同步），将数据部署到需要的 Region 上。后续的更新也由 DRC 进行同步。

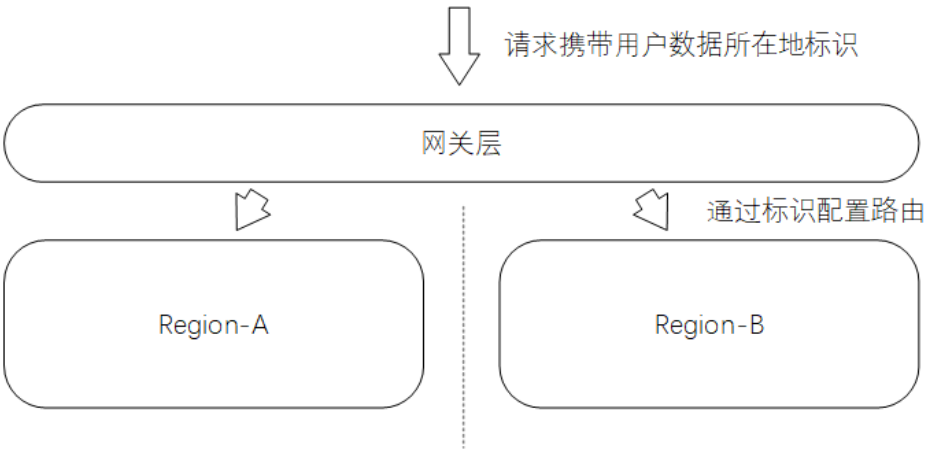


在数据部署完成后，如何保证用户的请求落到了正确的 Region。

方案一：每次请求经过网关的时候，网关进行一次用户到 Region 的映射查询，再将请求正确的路由到数据所在 Region。这样的做法不仅会对请求引入一次额外的查询，还会使得网关这一及其关键的节点引入一个依赖，会影响整个网站的稳定性。

方案二：基于用户的数据一定完整的存在某一个 Region 的前提，在用户登录的时候，将之

前识别时打上的标识下发到端上。请求的时候，网关只需对标识进行简单的路由配置，即可正确的路由到对应的 Region。



4.2 多 Region 架构

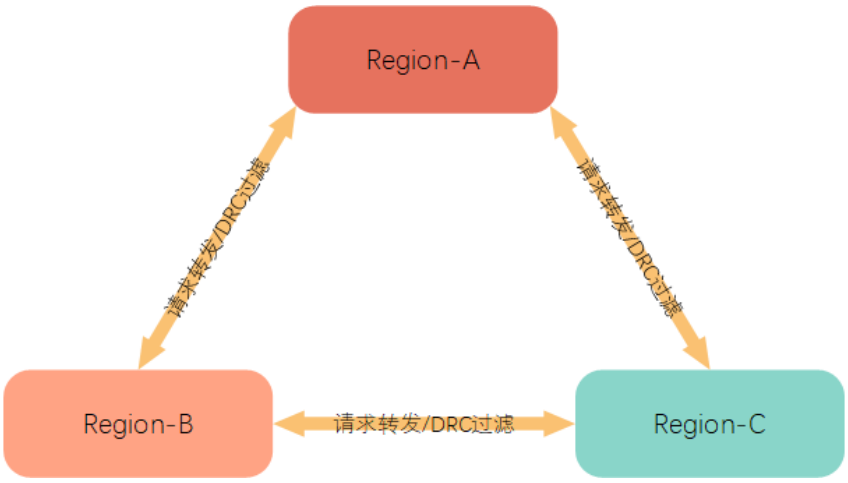
在多 Region 的部署中，账号中台实现了同构部署。

1）网关层。利用网关根据用户登录下发的标识，将请求路由到正确的 Region。

2）内网服务。通过完整的部署，内网请求在同一 Region 内完成，实现 Region 闭环，提高服务性能。同时也避免了 DB 多 Region 写入引起数据冲突的问题。

3）数据层。

- DB 数据。DB 表结构完全一致，通过 DRC 根据用户标识进行带过滤的多向复制。
- Redis 缓存数据。本地使用，不需要同步。当一个 Region 的数据有变更的时候，其他 Region 接受 DRC 的同步消息，对本 Region 的 Redis 进行删除。



在这样的多 Region 部署架构下，可以根据业务的使用场景和数据部署需求，实现用户数据的单 Region 或多 Region 存储。

五、结语

账号系统从最开始的巨大单体应用中剥离出来，经历了若干年的演进，变成了现在支持多 Region 部署的账号中台，这是业务发展和互联网技术进步的一种体现和必然趋势。

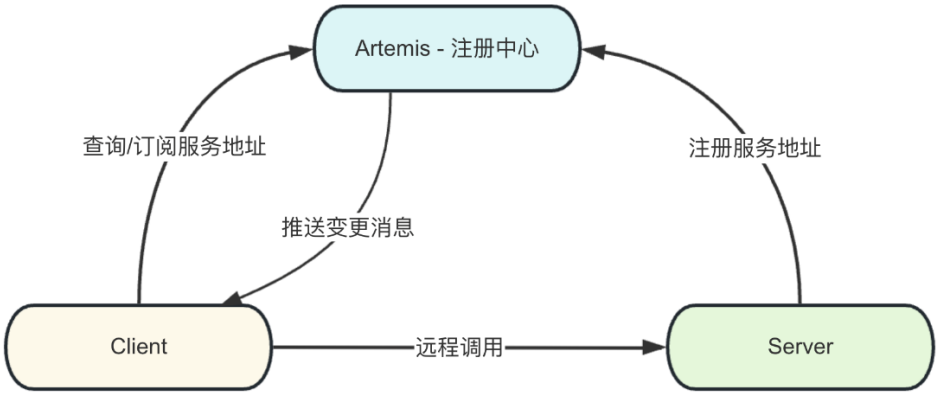
当然，这种趋势也不会停止于此，账号系统的功能和架构必将继续演进，其他系统也同样如此。回望过去，基于现在，展望未来，敢为人先，为各业务的发展打好基石，这正是账号等基础系统的意义所在，技术团队的价值也存在于此。

携程注册中心整体架构与设计取舍

【作者简介】Siegfried，携程软件技术专家，负责携程注册中心的研发。

一、 前言

目前，携程大部分业务已经完成了微服务改造，基本架构如图。每一个微服务的实例都需要和注册中心进行通讯：服务端实例向注册中心注册自己的服务地址，客户端实例通过向注册中心查询得知服务端地址，从而完成远程调用。同时，客户端会订阅自己关心的服务端地址，当服务端发生变更时，客户端会收到变更消息，触发自己重新查询服务端地址。



疫情刚过去那会，公司业务回暖迹象明显，微服务实例总数在 1 个月左右的时间里上涨 30%，个别服务的单服务实例数在业务高峰时可达万级别。按照这个势头，预计全公司实例总数可能会在短时间内翻倍。

实例数变大会引起连接数变大，请求量变高，网络报文变大等一系列现象，对注册中心的性能产生挑战。

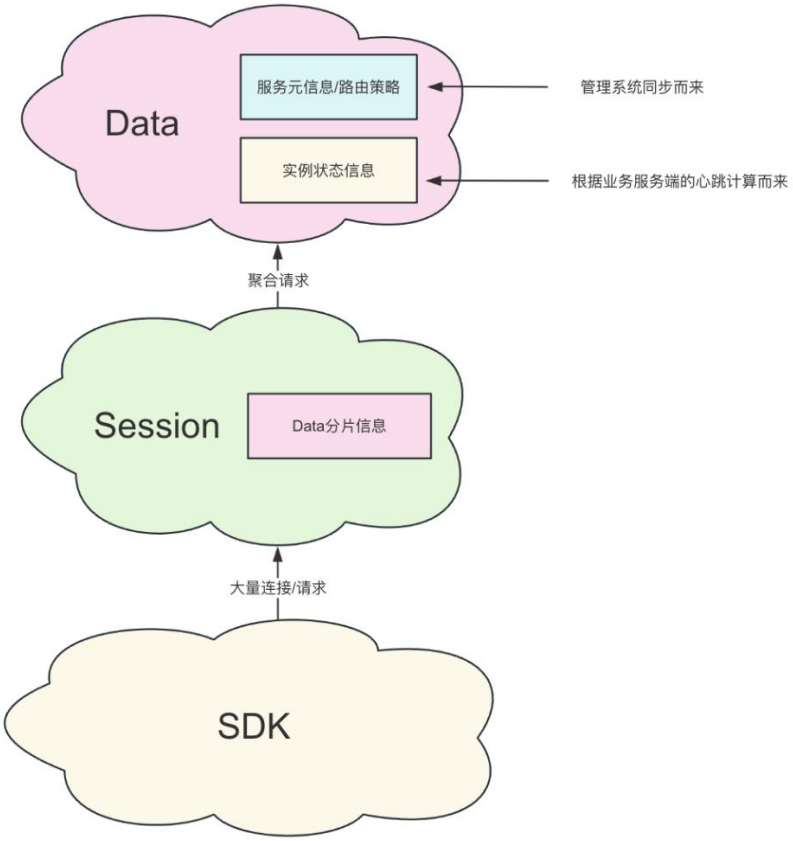
如果注册中心遇到性能瓶颈或是运行不稳定，从业务视角看，这会导致新增的实例无法及时接入流量，以至被调方紧急扩容见效慢；或者导致下线的实例不能被及时拉出，以至调用方业务访问到已下线的实例产生报错。

如今，业务回暖已经持续接近 2 年，携程注册中心稳定运行，强劲地支撑业务复苏与扩张，特别是支撑了业务日常或紧急情况下短时间内大量扩缩容的场景。今天就来简单介绍一下携程注册中心的整体架构和设计取舍。

二、整体架构

携程注册中心采用两层结构，分为和数据层(Data)和会话层(Session)。Data 负责存放被调方的元信息与实例状态、计算 RPC 调用相关的路由策略。Session 与 SDK 直接通讯，负责扛连

接数，聚合转发 SDK 发起的心跳/查询请求。



注册 – 定时心跳

微服务架构下，服务端的一个实例（ 被调方）想要被客户端（调用方）感知，它需要将自己注册到注册中心里。服务端实例会发起 5 秒 1 次的心跳请求，由 Session 转发到对应分片的 Data。如果数据层能够持续不断的收到一个实例的心跳请求，那么数据层就会判断这个实例是健康的。

与此同时，数据层会对这一份数据设置 TTL，一旦超过 TTL 没有收到后续的心跳请求，那么这份数据也就会被判定为过期。也就是说，注册中心认为对应的这个实例不应再被调方继续访问了。

发现 - 事件推送/保底轮询

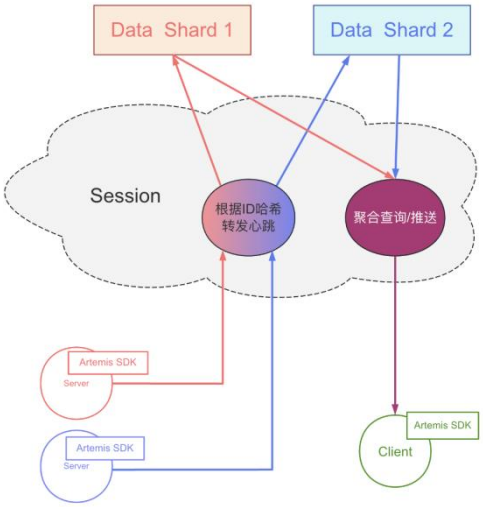
当收到新实例的第一个心跳时，数据层会产生一个 NEW 事件，相对应地，当实例信息过期时，数据层会产生一个 DELETE 事件。NEW/DELETE 事件会通过 SDK 发起的订阅连接通知到调用方。

由于网络等一些不可控的因素，事件推送是有可能丢失，因而 SDK 也会定时地发起全量查询请求，以弥补可能丢失的事件。

多分片方案

如图所示，Data 被分成了多分片，不同分片的数据互不重复，从而解决了单台 Data 的垂直瓶颈问题（比如内存大小、心跳 QPS 等）。

Session 会对服务 ID 进行哈希，根据哈希结果将心跳请求、订阅请求、查询请求分发到对应的 Data 分片中。调用方 SDK 对多个被调方进行信息查询时，可能会涉及到多个 Data 分片，那么 Session 会发起多个请求，并最终负责将所有必要信息聚合起来一并返回给客户端。

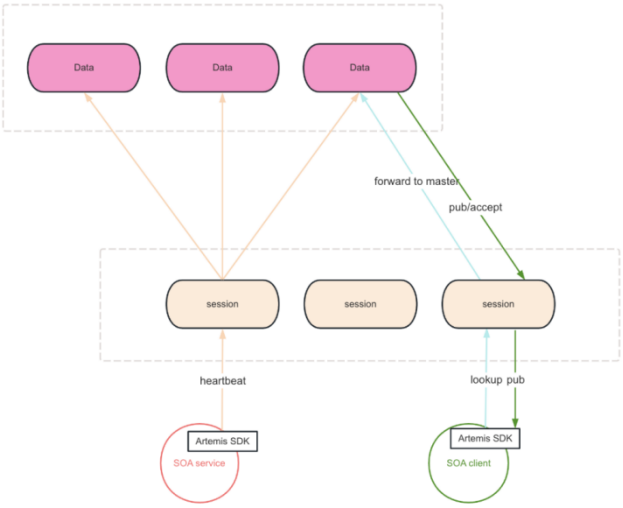


单点故障

与很多其他系统类似，注册中心也会遇到故障/维护等场景从而遭遇单点故障。我们把具体情况分为 Data 单点故障和 Session 单点故障，在两种情况下，我们都需要保证系统整体的可用性。

单点故障 – Data

如图所示，SDK 发起的心跳请求会被复制到多台 Data 上，以保证同一分片中每一台 Data 的数据完整性。也就是说，同一个分片的每台 Data 都会拥有该分片对应的所有服务的数据。当任一 Data 出现故障，或是参与到日常运维被踢出集群的情况下，其他任一 Data 能够很好的接替它的工作。



这样的多写机制相比于之前版本注册中心采用的 Data 间复制机制更加简单。在 Data 层发生故障时，当前方案对于集群的物理影响会更小，可以做到无需物理切换，因而也更加可靠。

在当前多写机制下，Data 层的数据是最终一致的。心跳请求被分成多个副本后是陆续到达各个 Data 实例的，在实例发生上线或者下线时，每台 data 变更产生的时间点通常会略有不同。

为了尽可能避免上述情况对调用方产生影响，每台 Session 会在每个 Data 分片中选择一台 Data 进行粘滞。同时，SDK 对 Session 也会尽可能地粘滞。

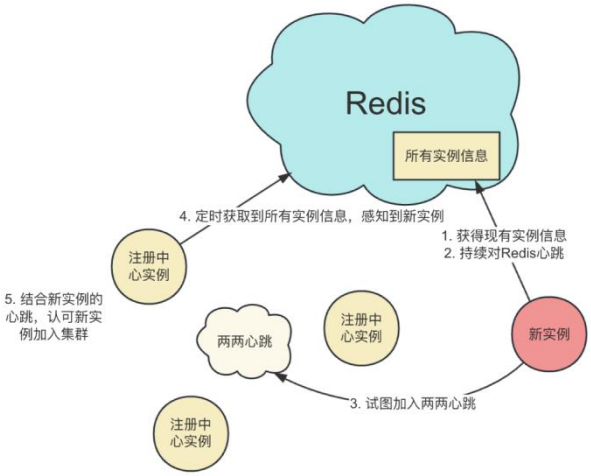
单点故障 – Session

参考上文提到 Data 分片方案，任一 Session 都可以获取到所有 Data 分片的数据，所有 Session 节点都具备相同的能力。

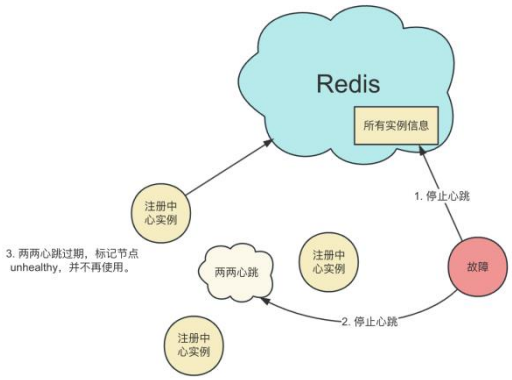
因此，任一 Session 故障时，SDK 只需要切换到其他 Session 即可。

集群自发现

携程注册中心是基于 Redis 做集群自发现的。如下图所示，Redis 维护了所有注册中心实例的信息。当一个注册中心实例被创建时，新实例首先会向 Redis 索要所有其他实例的信息，同时开始持续对 Redis 发起心跳请求，于是 Redis 维护的实例信息中也会新增新实例。新实例还会根据从 Redis 拿到的数据向其他注册中心实例发起内部的心跳请求。一旦其他实例从 Redis 获得了新实例的信息，再加上收到的心跳，就会认可新实例加入集群。



如下图所示，当时注册中心实例需要维护或故障时，实例停止运行后不再发起内部心跳。其他实例在该节点的内部心跳过期后，标记该节点为 unhealthy，并在任何功能中都不会再使用该节点。这里有一个细节，节点下线不会参考 Redis 侧的数据，Redis 故障无法响应查询请求时，所有注册中心实例都以两两心跳为准。



我们可以了解到，注册中心实例的上线是强依赖 Redis 的，但是运行时并不依赖 Redis。在 Redis 故障和运维时，注册中心的基本功能不受影响，只是无法进行扩容。

二、设计取舍

新增代理还是 Smart SDK?

注册中心设计之初只有 Data 一层，由于要引入分片机制，才有了 Session。那么是不是也可以把分片的逻辑做到 SDK，而不引入 Session 这一层呢?

这也是一种方式，业界也一直有着代理和 Smart SDK 之争。我们基于注册中心所对应的业务场景，认为新增一层代理是更加合适的。

最重要的一点，注册中心的相关功能运行不在 BU 业务逻辑主链路上，其响应时间并非直接

影响业务的响应时间。因此我们对注册中心的请求响应时间并没有极致的要求，代理层引入的几百微秒的延迟可以被接受。

其次注册中心的请求是一定程度容忍失败的，SDK 请求数据失败后可以继续使用内存中的老数据，不会对业务线产生致命影响。因此代理层引入的失败率也可以被接受。

另一侧，代理的加入带来了诸多好处。最直接地，落地分片逻辑不需要所有的 SDK 升级，分片逻辑迭代时，对业务也是无感。

其次，代理层也隔离了连接数这一瓶颈，当 SDK 层的实例不断变多，连接数不断增加时，只需要扩容代理层就能解决连接数的问题。这也是我们将它取名为 Session 的原因。

同时，我们也希望作为物理层的 SDK 逻辑更加轻量，比较重的逻辑放在逻辑层，这样稳定性更强更不容易出错。比如后续会提到的“Data 按业务隔离分组”就是在 Session 层实现的。

普通哈希还是一致性哈希？

携程注册中心的数据分片是采用普通哈希的，并没有采用一致性哈希。

我们知道，一致性哈希相比普通哈希的最大卖点是当节点数量变化时，不需要迁移所有数据。

结合注册中心的场景，我们用服务 ID 做哈希，而服务数量（也包括实例数量）是相对稳定的，因此哈希节点的扩容周期会比较长，基本用不到一致性哈希的优势特性。哪怕一段时间内业务迅速扩张，只要提前做好预估，留好余量一次性扩容就好了。

我们选择普通的固定的哈希，并让每一个分片都具备多个备份节点，这样就基本可以认为每个分片都不会彻底挂掉，不用去实现数据迁移的逻辑，整个机制更简单了。

要知道，数据迁移需要对注册请求、查询请求和订阅请求进行同步切换，要处理好各种状态，避免在数据迁移过程中错查到空数据或者丢失变更事件，非常复杂危险。

自发现是否强依赖 Redis？

前面也提到，注册中心自发现的运行时是不依赖 Redis 的。有的同学可能会想到，如果运行时强依赖 Redis，就可以去掉两两注册了。

两两注册确实是一个不好的设计，随着集群的节点数越来越大，其产生的性能开销肯定也会更大，影响整个注册中心集群的拓展能力。

但在目前规模下，内部心跳占用的系统资源并不可观。哪怕规模再拓展，通过降低心跳的频率，进一步降低资源开销。

最大的好处是，Redis 集群故障或者维护时，并不会对注册中心的功能产生影响。

基于 Redis 还是用 Java 写？

目前注册中心的 Data 是用 Java 实现的。有的同学可能会想，Data 层主要就是维护微服务实例的存活状态，能不能直接用 Redis 实现呢？如果用 Redis，不就可以直接复用 Redis 体系的扩容/切换能力了吗？

比如基于 Redis 6.0 的 Client Cache 功能，通过 Invalidate 机制通知 SDK 重新更新服务信息。

不过在携程注册中心设计之初，Redis 版本还比较老，没有这些新 feature，感觉基于 pub/sub 机制做注册中心还挺麻烦的。现在注册中心已经稳定运行了好久，加了很多功能，比如路由策略一部分的计算过程就是在 Data 层完成的，暂时没有必要推倒重建。

总的来说，用 Java 写更可控，后续自定义程度更高。

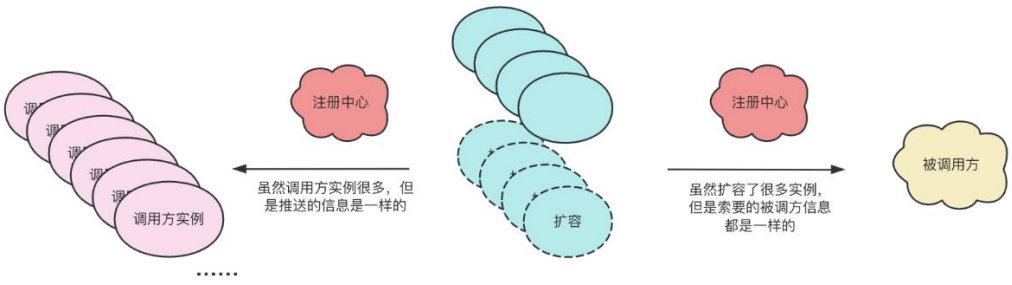
四、需要注意的场景

突发流量

在遇到节假日，或是公司促销活动，亦或是友商故障的情况下，公司集群会因为业务量急剧上升而迅速自动扩容，因而注册中心会受到强劲的流量冲击。

期间因为系统资源被榨干，注册/发现请求可能会偶发失败，事件推送延迟和丢失率会上升。严重时，部分调用方业务会无法及时感知到被调方的变动，从而导致请求失败，或流量无法被分摊到新扩容的被调方实例。

我们发现，这些场景产生的流量有着很高的重复度，比如某个被调方实例扩容，调用方的众多实例需要知道的信息是完全一样的，又比如调用方实例扩容，这些新扩的实例部署着相同的代码，它们依赖的被调方信息也是完全一样的。



因此我们针对性的做了不少聚合与去重，大大降低了突发流量情况下的资源开销。

流量不均衡

关于 Data 粘滞，这里有一个细节。那么多 Data 机器，Session 选谁呢？目前 Session 是用类似随机的方式选择 Data 的。那就会有一个场景，我们对 Data 层进行版本更替，逐个实例

重新发布，当一个实例被重置时，Session 就会因为丢失粘滞对象而重新随机选择。

我们会发现，最后一个 Data 实例完成发布时，它不会被任何 Session 选中。而第一个发布的 Data 实例，它倾向于被更多的 Session 选中。

通常来说，越早发布的 Data 实例，就会被越多的 Session 选中。也正因为如此，更早发布的 Data 会承担更多的流量，而最后发布的 Data 一般不承担流量。这显然是不合理的。

解决这个问题的方法也很简单，我们引入拥有全局视角的第三者，整体调控 Session 的粘滞，保证 Data 尽可能地被相同数量的 Session 选中。

全局风险

前面也提到，Data 层被分成了多分片，Session 会对服务 ID 进行哈希，将心跳请求、订阅请求、查询请求分发到对应的 Data 层分片中。

当程序出现预期外的问题（程序 bug，OOM 等等）导致某个 Data 无法正常的履行功能职责时，那些被分配到这个 Data 实的服务就会受到影响。

如果调配方式是对服务 ID 做哈希，那么所有业务线的任意服务都可能参与其中，从业务视角去看，就是整个公司都受到了影响。

对服务 ID 做哈希是有它的优势的，它无需引入过多的外部依赖，只需要一小段代码就能工作。但我们还是认为避免全局故障更加重要。

因此我们最近对 Data 引入了业务语义，将 Data 分为多个组，以各个业务线命名。且我们可以按服务粒度对数据进行分配。默认情况下，我们会将服务分配到自己 BU 的分组上。

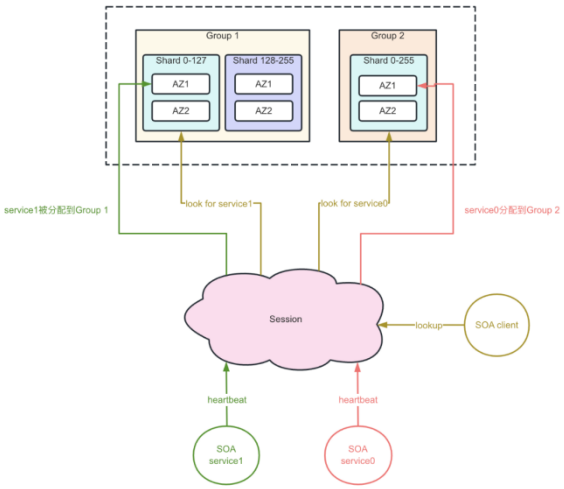
这样，我们就具备了以下能力：

1）不同业务线的数据可以被很好的隔离，任一业务线的 Data 数据出现问题，不会影响到其他业务线。

2）注册中心将获得故障切换的能力，当个别服务的数据出现问题时，我们可以将它单独切走。

3）我们可以将一些不重要的应用单独隔离到一个灰度分组，新代码可以先发布到灰度分组上，尽可能避免新代码引入的问题直接影响核心业务分组。

4）注册中心将获得应用粒度的部署能力。在集群分配上，具备更强的灵活度，针对业务规模的大小合理分配系统资源。



从图中可以看到，我们在引入分组逻辑的同时也兼容老的分片逻辑，这样做是为了在分组逻辑上线过程初期，服务信息在 Data 层的分布可以尽可能保持不变，可以让少数的服务先灰度切换到新增的分组上进行验证。

当然，从去复杂度的角度考虑，最终分片逻辑还是要下线，垂直扩容的能力也可以由分组实现。

五、后续规划

因为注册中心引入了分组机制，并对各个业务线数据进行了隔离，注册中心的集群规模也在因此膨胀，分组数量较多，运维成本也随之上升。

后续我们计划进一步优化优化单机性能，精简优化一些不必要的机制，降低机器数量。

同时，我们也希望注册中心能够支持弹性，能够在业务高峰时自动扩容，在业务低峰时自动缩容。

携程门票秒杀系统的设计与实践

【作者简介】Liang，携程技术专家，专注系统性能、稳定性、承载能力和交易质量，在技术架构演进、高并发等领域有丰富的实践经验。

本文概述了携程门票预订交易系统在应对秒杀活动中面临的挑战与应对策略。第一部分阐述了业务激增对系统架构的考验；第二部分深入剖析了系统架构的优化路径，涵盖读热点、写入性能瓶颈、强一致性事务处理及流量精细化控制等关键问题的解决方案，并总结了确保系统高可用性与持续性的治理措施。希望这些内容能够对大家有所帮助或启发。

一、背景

后疫情时代旅游行业快速复苏，各类营销秒杀活动变得越发频繁，面对亿级流量的冲击，系统架构面临挑战。研发团队需要保障大流量下的功能稳定性，为国内外用户提供流畅的预订体验，因此需要对核心的预订交易系统进行应用架构升级，从而确保系统在高并发情况下仍能稳定高效运行。

本文将介绍在应对流量高峰、突破系统瓶颈、强化系统稳定性等方面的应对策略与优化效果。

二、秒杀活动案例分析

回顾大家曾经参与过的秒杀或大促活动，如双十一、618、12306 节假日抢票、演唱会抢票时，会有相似的感受：

1) 紧张刺激：活动通常定时开售，期待与紧张并存。

2) 系统压力：在高峰期，系统容易出现卡顿、宕机或提示“太火爆”或需要排队等待，让人倍感焦虑。

3) 结果未知：尽管全力以赴，但结果往往不尽如人意，有时抢到了票无法支付或者可能被退单。

这些活动在预订交易系统中也会呈现相似的特征：

1) 大流量、高并发：大流量、高并发、强事务性，对系统性能提出严峻挑战。

2) 时间敏感性：准时开售，用户争抢热点资源，系统需要确保实时、准确地响应。

3) 履约保障：从订前到订后，系统需要确保履约的顺利进行，避免用户因系统问题而遭受损失。

与传统电商相比，携程门票交易系统具有两大特点：

1) 强一致性：用户预订后保证出票且尽可能快速确认，确保每一笔交易都能履约。

2) 多维度和跨商品组合限购：限购规则复杂多变，例如多维度和跨商品组合限购，保障每位用户有公平购票的机会，避免囤票行为。

接下来回顾历史上有过的携程门票大型秒杀/活动案例。

1) 2020 年 8 月 8 日~9 月 1 日："惠游湖北"活动，携程独家承办，首次面对日常流量 45 倍 (数十万 QPS) 峰值的流量挑战，虽然刚开始系统出现不稳定的情况，但最终还是成功应对。

2) 2021 年 9 月 14 日：北京环球影城开业开售活动，携程门票在与其他友商的同期竞争中，成为唯一稳定出票且销量最高的交易平台。

3) 2023 年 9 月 15 日：武汉动物园开园，在供应商系统出现异常、友商页面卡顿有大量退单的情况下，携程门票预订依然能保持顺畅下单。

4) 2024 年 4 月 10 日：IU（李知恩）全球演唱会门票在 Ctrip.com 和 Trip.com 国际站同时秒杀，携程门票再次表现稳定，预订过程丝滑流畅，10 秒内售罄。

以下是部分历史秒杀活动峰值流量与日常峰值流量的对比数据:

日期	活动名称	峰值流量与日常流量的倍数
2020年	惠游湖北0元票活动	45倍
2021年	北京环球影城开业	5倍
2023年	五一假期活动	5倍
2023年	武汉动物园开业	35倍
2023年	十一假期活动	5倍
2024年4月10日	IU演唱会CT同售	73倍

数据显示出活动的流量激增通常远超系统日常处理的极限，如果没有针对预订交易系统进行优化，用户可能会遇到各种问题，例如:

1) 页面打开慢、卡顿、宕机：直接影响用户购物体验，系统会出现 Redis 或 DB 超负载，供应商接口不稳定等情况。

2) 付款后不能确认/退款：付款后，无法及时确认订单状态或进行退款操作，系统出现库存超卖/少卖等情况。

要避免出现上述情况，就要求系统具备高度的可扩展性和灵活性，同时在架构、缓存、数据库、流量控制等多方面进行全面优化。接下来我们通过具体场景来分析系统遇到的问题和应对策略，了解系统架构设计与演进过程。

三、系统架构设计与演进

整体而言，预订交易系统的目标是：稳、准、快。

- 稳：确保系统稳定可靠，保障售卖流程无间断。
- 准：实现数据一致性，确保履约准确无误。
- 快：提供流畅的预订体验，实现快速确认。

在大流量高并发场景下，要达到这些目标就可以进行有针对性的改造升级，接下来展开阐述。

3.1 系统稳定性挑战与应对策略

当系统遇到洪峰流量时，容易出现页面打开慢、卡顿等问题，主要原因有以下几点：

1) Redis 超负载与缓存热点。

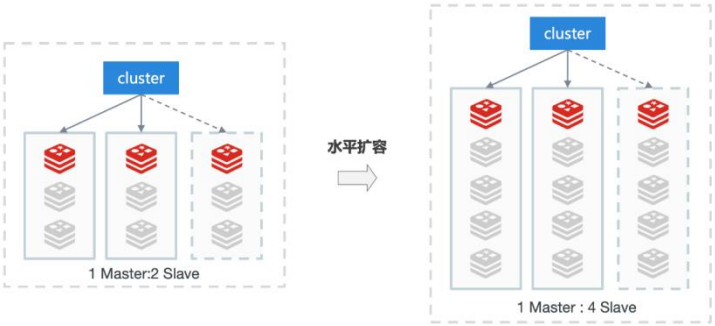
2) 数据库超负载。

3) 供应商系统不稳定。

接下来针对这 3 个常见问题，阐述相应的应对策略。

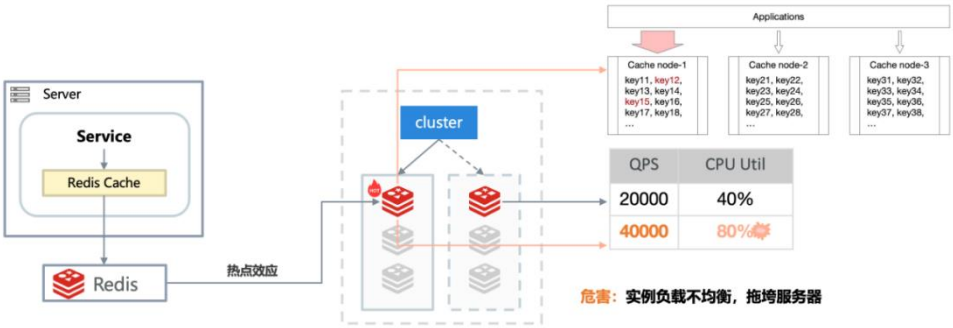
问题一：Redis 超负载与缓存热点

当 Redis 面临负载问题时，可以使用水平扩容这种常规手段让流量分摊到更多实例。然而扩容虽能降低大多数实例的 CPU 使用率，但在处理特定热点数据时，各实例的 CPU 使用率仍然可能出现不均衡的情况，即缓存热点问题；此外还会存在缓存大 Key 问题。



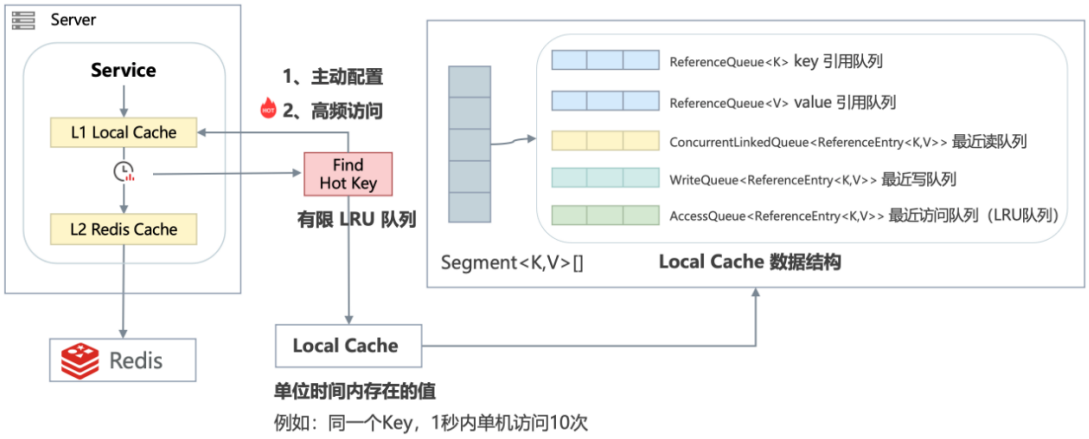
1) 缓存热点问题

如下图所示，node-1 节点存在 2 个热点访问，请求量远高于其他节点。缓存热点会导致实例负载不均衡，从而严重影响响应速度。



缓存热点应对方案：热点识别自动构建多级缓存

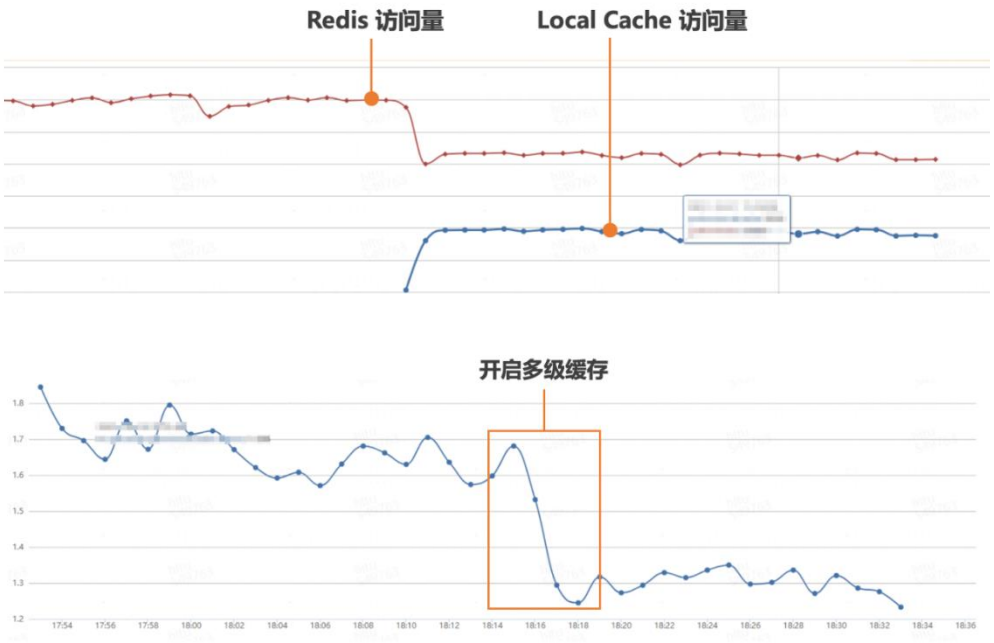
将单位时间内高频访问的 Key，识别出来。例如：同一个 Key，1 秒内单机访问 10 次。



如上图所示，自动发现 Hot keys 或将指定的 Key 加入到本地缓存。

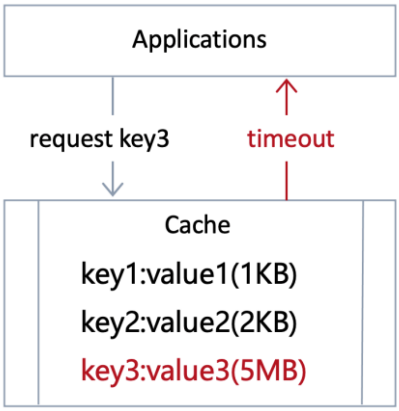
秒杀时：短暂的本地缓存可以减少 Redis 单实例热点，对数据的一致性不会有较大影响。

优化效果：开启多级缓存后，同一个缓存 key 访问性能明显提升，对应 Redis 访问量也明显降低（如下图所示）。

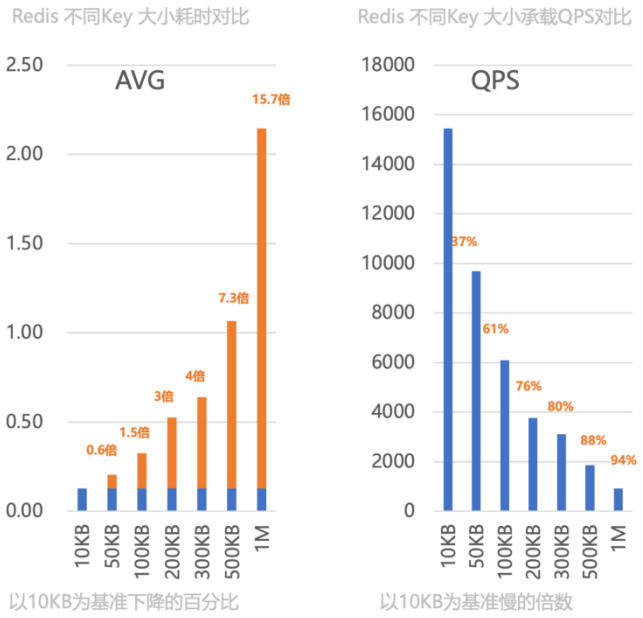


2) 缓存大 Key 问题

缓存大 key 的危害主要包括：阻塞请求、内存占用大、阻塞网络等。不仅会降低 Redis 的性能，还可能影响整个系统的稳定性（如下图所示）。



通过 memtier-benchmark 工具在生产环境下压测：200KB 以上比 10KB 以内的性能慢 3 倍，吞吐能力也下降 76%（如下图所示）。



缓存大 Key 应对方案：

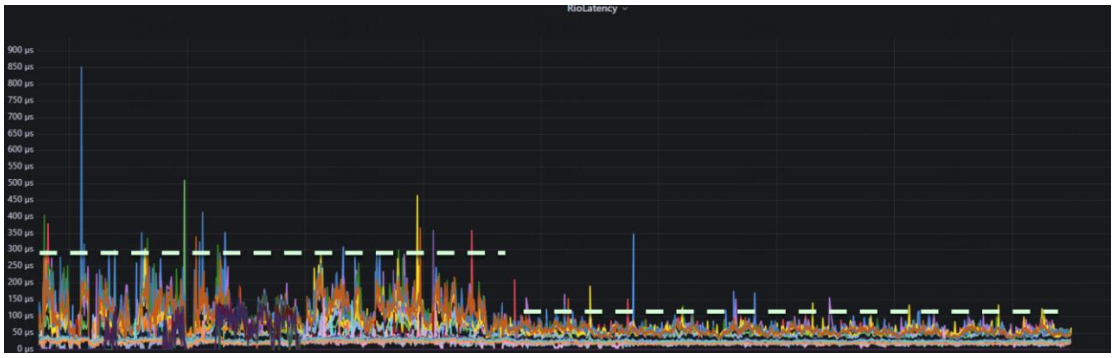
a）精简缓存对象：去除缓存中的冗余字段。

b）压缩缓存对象：采用压缩比更高的压缩方式，缩小缓存对象。

c）拆分大 Key：若精简和压缩后还是过大，根据业务逻辑，将大 Key 拆分成多个小 Key。(注意拆分后 IO 次数会增加，高负载下性能不一定会变好，需要根据压测结果来评估最终性能)

d）长期治理：建立长期治理机制，定期扫描 Redis 中的大 Key，每周跟进，将隐患在日常治理中消除。

优化效果：在大 Key 优化后，Redis 查询性能有较为明显的提升（如下图所示，缓存查询耗时从 300μs 优化至 100μs）。



问题二：数据库超负载

系统中商品信息的变更往往伴随着缓存失效的问题，尤其在高并发和秒杀场景下，大量请求可能直接穿透缓存层，对数据库造成巨大压力，甚至引发性能故障。

缓存更新策略优化：应对商品变更导致的数据库压力

1) 常见的缓存架构设计问题

监听器收到消息后删除相应的缓存 Key。这种方式在一般情况下是有效的，但在高并发和大流量场景下，它存在几个突出的问题：

a) 缓存击穿：由于缓存的 Key 被立即删除，大量请求在缓存未更新之前会直接访问数据库，导致数据库压力骤增。

b) 消息处理延迟：在高并发场景下，消息处理可能产生延迟，导致缓存更新不及时，进一步加剧数据库压力。

2) 缓存更新策略的优化

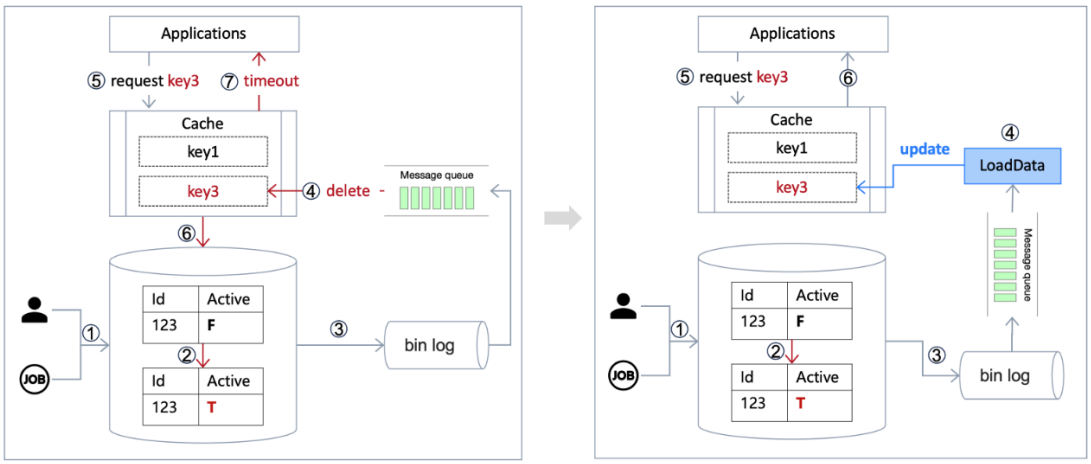
为了应对这些挑战，采取了一系列优化措施，主要包括：

a) 缓存覆盖更新策略：替代直接删除缓存 Key 的做法，采用了缓存覆盖更新策略。当商品信息发生变更时，系统不再删除缓存 Key，而是直接更新该 Key 对应的缓存值。避免了流量穿透到底层数据库。

b) 消息聚合：针对商品变化消息量过大的问题，引入了消息聚合机制。将商品多次变化消息在一段时间窗口内合并成一个，减少消息处理的频率。

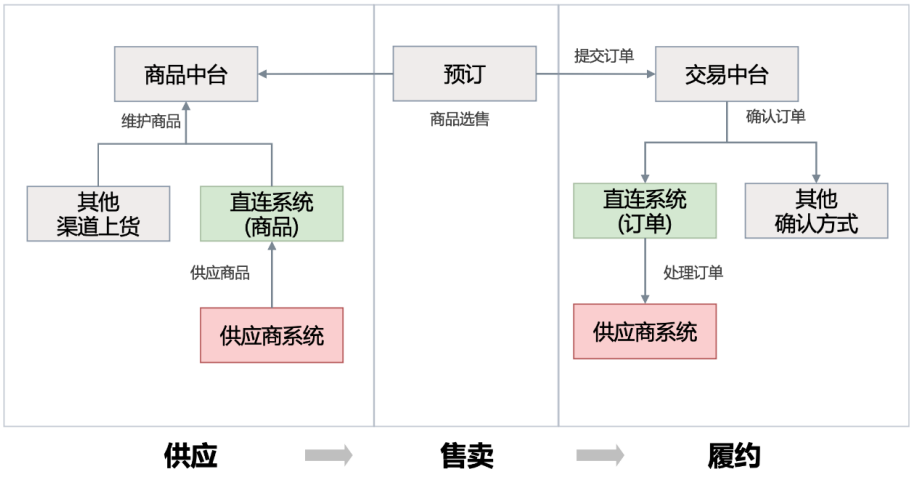
c) 异步更新缓存：为了进一步降低对数据库的实时压力，采用了异步更新缓存的策略。当商品信息发生变更时，系统不会立即更新缓存，而是将更新任务放入一个异步队列中，由后台线程异步处理。

缓存更新策略变化如下图所示：



问题三：供应商系统不稳定

供应商系统因大流量导致响应缓慢或被限流，影响整体系统的稳定性。



应对供应商系统不稳定性的技术策略优化

当供应商系统面临大流量冲击时，往往会出现响应缓慢甚至被限流的情况，这直接影响了我们自身系统的稳定性和用户体验。

供应商订单对接问题

当与供应商进行订单对接时，可能会遇到以下问题：

a）被供应商限流：在高并发场景下，供应商系统可能会对我们限流。这会导致我们的订单提交受阻，影响业务流转。

b）供应商系统不稳定：由于各种原因，供应商系统可能会出现不稳定的情况，导致订单处理延迟或失败。

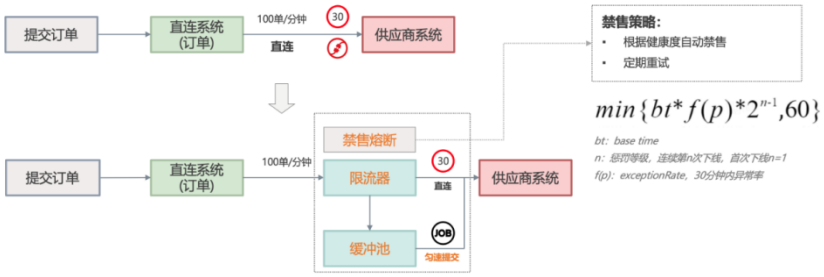
为了缓解上述问题，我们采取以下技术策略：

1）削峰填谷/缓冲池：利用消息队列作为订单提交的缓冲池，将订单信息先写入队列，再由后台服务异步处理。这样可以将订单提交的高峰流量削平，减少对供应商系统的瞬时压力。

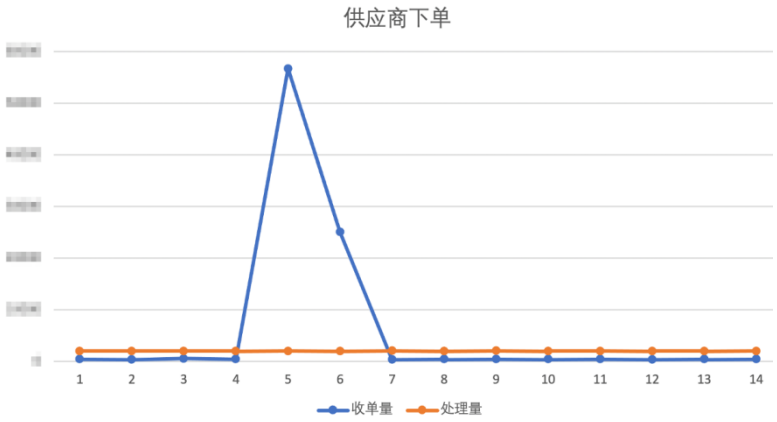
2）禁售策略

• 自动禁售：建立对供应商系统的健康度监控机制，实时监测其响应速度、错误率等指标。一旦发现供应商系统出现不稳定或限流的情况，及时触发禁售策略。

• 定期重试：对于因供应商系统问题而失败的订单，设定了一个重试机制，定期尝试重新提交。同时，根据供应商系统的恢复情况，动态调整重试的频率和次数。



优化效果：通过实施上述技术和策略优化，可以有效确保供应商系统能力不影响下单吞吐量(如下图所示)。

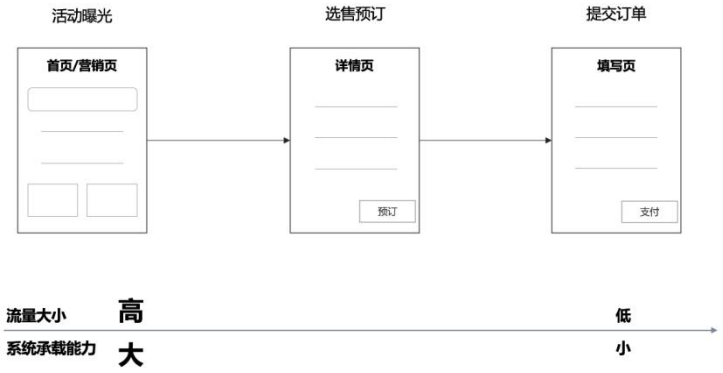


上述的优化措施落地后能够提升系统的稳定性，然而鉴于流量的不确定性，即使流量超过系统负载能力，系统也要正常运行，因此仍然需要有相应的流量控制策略。

流量控制策略优化：确保秒杀活动稳定运行

如下图所示，不同页面对应的流量和系统(承载能力)是不同的，需要控制好每个过程的流量，

确保整体系统的稳定性。



以 70 万人购买 5000 张票的秒杀活动为例，可采取以下限流策略：

1) SOA 限流：接口与应用级限流

通过服务治理框架对服务接口进行限流（SOA 限流），在秒杀/活动等场景会影响到其他商品的正常售卖。对此可针对秒杀活动的特殊需求，设计自定义的限流策略，如按秒杀商品限流、页面级限流等，细化商品维度的流量控制。

2) 自定义限流：商品级限流

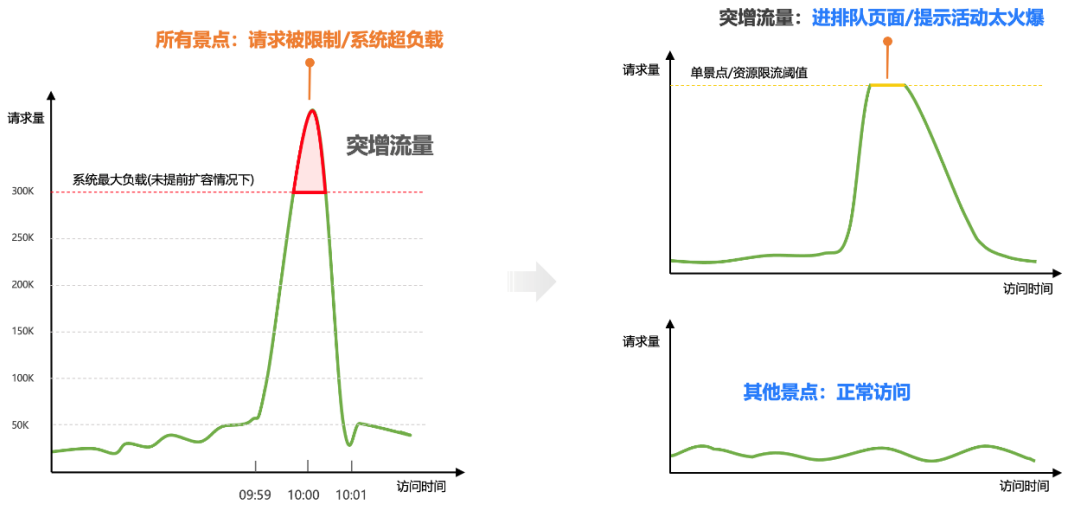
a）针对单个秒杀商品设置独立的限流阈值，即使某个商品超负载，也不会影响整体系统的可用性。

b）同时，对于未知的秒杀突增流量，也可以支持热点商品自动限流，与 Redis 热 Key 发现类似，自动识别热点访问的商品，并添加到商品级限流中，从而确保整体系统的稳定运行。

如下图所示，我们采用了商品维度的自定义限流策略，该策略将 1 秒内的请求流量划分为 10 个独立的 100 毫秒(可配置)滑动窗口。每个窗口都会平分一部分流量，以确保下游服务的并发量得到有效控制。这种方法不仅降低了下游服务的压力，也为用户提供更加均衡的流量分配。

结合商品级限流能力，控制进入每一个页面的流量，形成多层次的限流防护体系，根据秒杀库存预估售卖时长，控制进入到每一个页面的流量比例，这样也能够大幅减少服务器资源投入。

优化效果：自定义限流可控制进入每一个页面的流量，超负载也不影响整体的可用性，服务器资源的投入也可控。



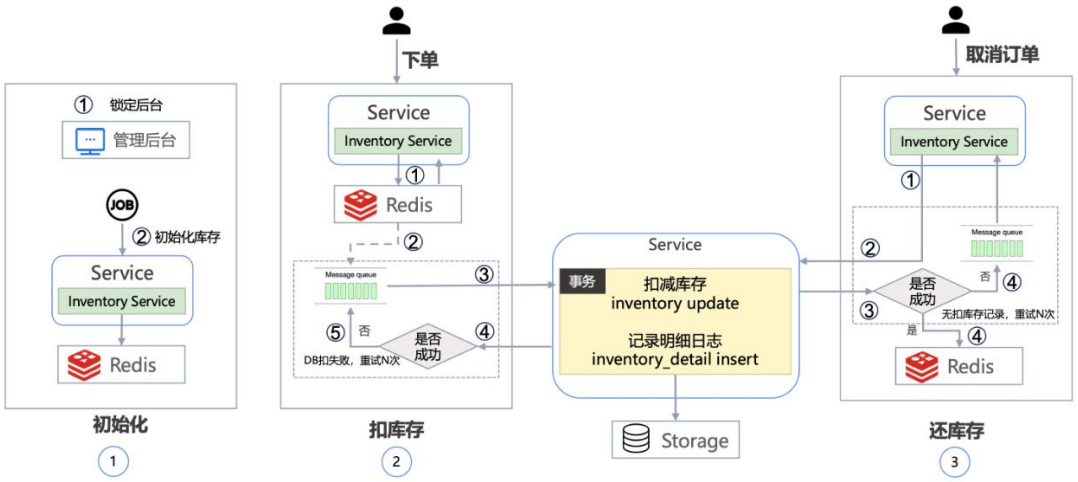
本部分阐述了系统稳定性的挑战及优化，包括 Redis 超负载与缓存热点、数据库超负载、供应商系统不稳定等。通过热点识别自动构建多级缓存、缓存覆盖更新策略、削峰填谷/缓冲池、自定义限流等多种技术策略，使得系统稳定性问题得到有效解决。

3.2 写数据一致性挑战与应对策略

下单过程中的库存扣减的精确执行，这种数据一致性的实现效果会直接影响订单是否能够成功履约，而传统关系型数据库的并发更新存在显著瓶颈，因此需要专项优化。

扣减库存问题：性能瓶颈 – MySQL 热点行扣减库存（行级锁）。

技术策略：扣减库存异步化，异步扣库存主要分 3 步（见下图）：



1）初始化：秒杀商品设置好活动场次，将秒杀库存同步至 Redis。

2）扣库存：活动开始时，先从 Redis 扣减库存，在通过消息通知异步扣减 DB 库存，削除 DB 更新同一行的峰值。

3）还库存：如果有退订，先判断 DB 中是否有扣减记录，如果有，则先退 DB 再退 Redis；如果没有，重试多次。

扣还库存过程中也会存在超时等未知情况，此处细节过多不再展开。按照业务“可少买不超卖”的原则，即使在这个过程中数据可能存在短暂的延时，但能够确保最终一致性。

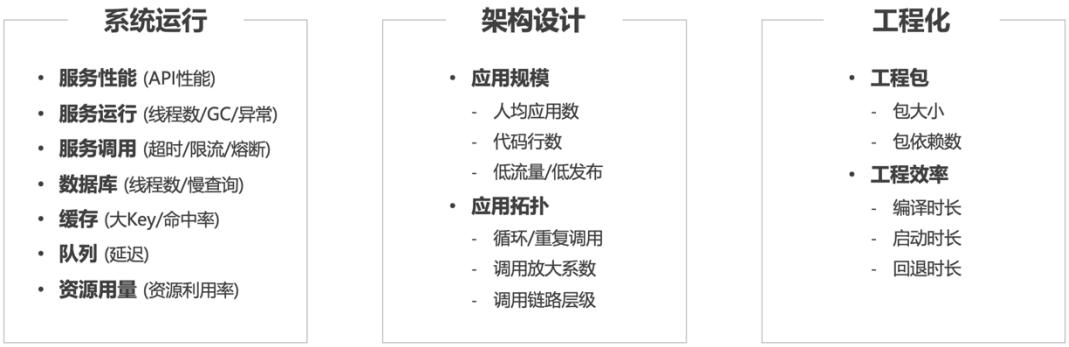
优化效果：库存扣减异步化，消除行级锁瓶颈。现在系统能够轻松支撑数十万单/分钟交易流量。

3.3 实现高可用的可持续性

系统是不断演进的，如何保持并持续优化系统能力就成为新的课题。因此日常架构健康度持续治理、以及大型活动和节假日保障体系是实现高可用“可持续性”的关键。

3.3.1 架构健康度治理

基于架构健康度实现系统质量的量化管理，实现研发生命周期各个环节的跟踪和优化，如下图所示可细分为三部分：



a) 系统运行健康度：通过系统各个维度运行时的健康状态和问题来反映系统质量。

b) 架构设计健康度：服务数量、调用关系的复杂度、循环依赖、调用层级过深等因素都会影响系统的稳定性和性能。

c) 工程化健康度：基于应用的工程质量和效率状态，反应出开发的工程化水平。

3.3.2 大型活动和节假日保障体系

无论大型活动还是节假日，都需要提前准备好应急预案，做好压测，提前保证系统的高可用。

日常开发 + 大型活动节假日保障形成闭环

四、总结

本文总结了携程门票的预订交易系统在承接秒杀活动中面临的挑战与应对策略。重点解决了读热点、写瓶颈、强事务、流量控制等诸多细节问题，同时通过日常的架构健康度治理和制定专项的保障计划，持续对系统进行优化，确保系统在高负载下依然能够稳定运行，实现系统的持续高可用。

Trip.com QUIC 高可用及性能提升

【作者简介】章磊，携程高级后端开发工程师，关注网络协议、算法优化、云原生等领域，对开源框架源码、高性能系统设计与优化有浓厚兴趣。

本文详细介绍了 QUIC 协议在携程 Trip.com App 上的实践方案，以及团队在 QUIC 高可用及性能提升方面所做的各类优化。首先介绍了 QUIC 多进程部署架构，随后分析了 QUIC 网络架构在生产应用中遇到的问题及其优化方案。在性能提升方面，分享了 QUIC 全链路埋点监控的实现思路及其收获，QUIC 拥塞控制算法开发与调优思路等等。希望这些内容能够帮助大家了解 QUIC 协议及其在实际应用中的优化思路，并从中获得启发。

一、前言

1.1 QUIC 在 Trip.com APP 的落地简介

QUIC（Quick UDP Internet Connections）是由 Google 提出的基于 UDP 的传输层协议，为 HTTP3 的标准传输层协议。相较于 TCP 协议，QUIC 主要具备以下优势：

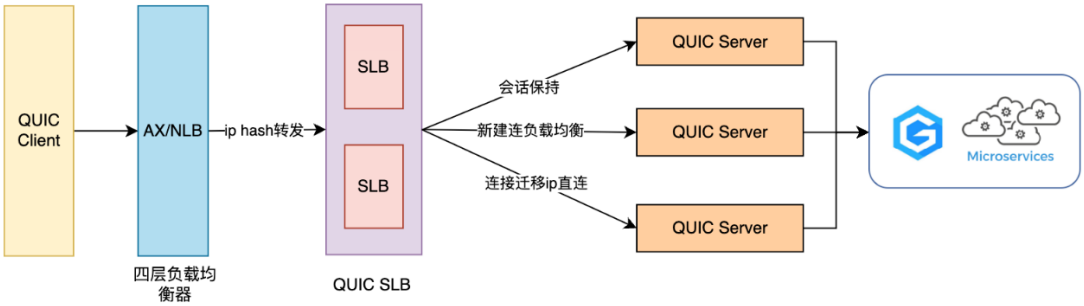
1）多路复用：QUIC 允许在单个连接上并行传输多个数据流，解决了 TCP 的队头阻塞问题，从而提高了传输效率；

2）快速建连：新建连时，QUIC 握手与 TLS 握手并行，避免了 TLS 单独握手所消耗的 1 个 RTT 时延。当用户连接过期失效且在 PSK(pre-shared key)有效期内再次建连时，QUIC 通过验证 PSK 复用 TLS 会话，实现 0-RTT 建联，从而可以更快的建立连接，此特性在短连接，单用户低频请求的场景中收益尤为明显；

3）连接迁移：TCP 通过四元组标识一个连接，当四元组中的任一部分发生变化时，连接都会失效。而 QUIC 通过连接 CID 唯一标识一个连接，当用户网络发生切换(例如 WIFI 切换到 4G)时，QUIC 依然可以通过 CID 找到连接，完成新路径验证继续保持连接通信，避免重新建连带来的耗时；

4）拥塞控制：TCP 的拥塞控制需要操作系统的支持，部署成本高，升级周期长，而 QUIC 在应用层实现拥塞控制算法，升级变更加灵活；

上述优质特性推动了 QUIC 协议在 IETF 的标准化发展，2021 年 5 月，IETF 推出了 QUIC 的标准化版本 RFC9000，我们于 2022 年完成了 QUIC 多进程部署方案在 Trip.com APP 的落地，支持了多进程下的连接迁移和 0-RTT 特性，最终取得了 Trip.com App 链路耗时缩短 20% 的收益，大大的提升了海外用户的体验。我们的初期网络架构如下：



其中有两个重要组成部分，QUIC SLB 和 QUIC Server：

- QUIC SLB 工作在传输层，具有负载均衡能力，负责接收并正确转发 UDP 数据包至 Server。当用户进行网络切换，导致连接四元组发生变化时，SLB 通过从连接 CID 中提取 Server 端的 ip+port 来实现数据包的准确转发，从而支持连接迁移功能；

- QUIC Server 工作在应用层，是 QUIC 协议的主要实现所在，负责转发及响应客户端请求，通过 Server 集群共享 ticket_key 方案实现 0-RTT 功能；

1.2 QUIC 高可用及性能提升

随着全球旅游业的复苏，携程在国内外的业务迎来了成倍的增长，业务体量越来越庞大，QUIC 作为 Trip.com APP 的主要网络通道，其重要性不言而喻。为了更好地应对日益增长的流量，更稳定地支持每一次用户请求的送达，我们建立了 QUIC 高可用及性能提升的目标，最终完成了以下优化内容：

QUIC 集群及链路高可用优化：

- 完成 QUIC Server 容器化改造，具备了 HPA 能力，并定制化开发了适用于 QUIC 场景的 HPA 指标

- 优化 QUIC 网络架构，具备了 QUIC Server 的主动健康监测及动态上下线能力

- 通过推拉结合的策略，有效提升了客户端 App 容灾能力，实现网络通道和入口 ip 秒级切换

- 搭建了稳定可靠的监控告警体系

QUIC 成功率及链路性能提升：

- 支持 QUIC 全链路埋点，使 QUIC 运行时数据更加透明化

- 通过优化拥塞控制算法实现了链路性能的进一步提升

- 通过多 Region 部署缩短了欧洲用户 20%的链路耗时
- 客户端 Cronet 升级，网络请求速度提升明显

QUIC 应用场景拓展：

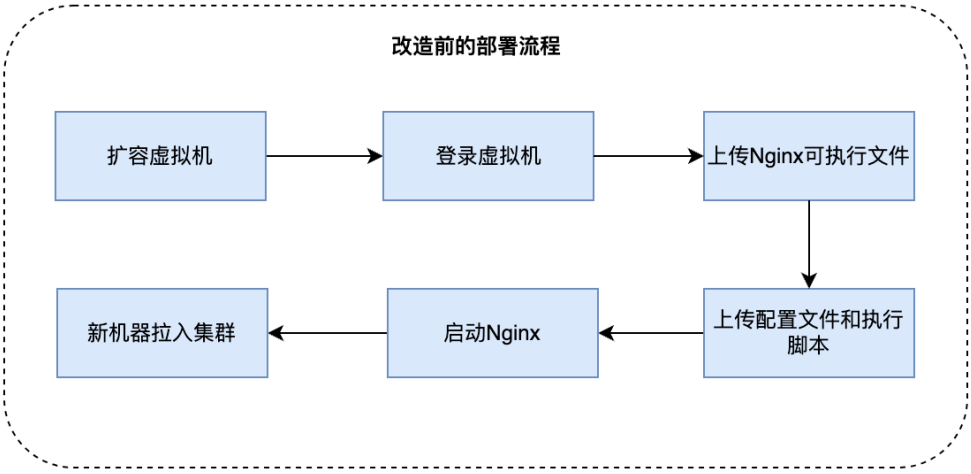
- 支持携程旅行 App 和商旅海外 App 接入 QUIC，国内用户和商旅用户在海外场景下网络成功率和性能大幅提升

下文将详细的介绍这些优化内容。

二、QUIC 网络架构升级

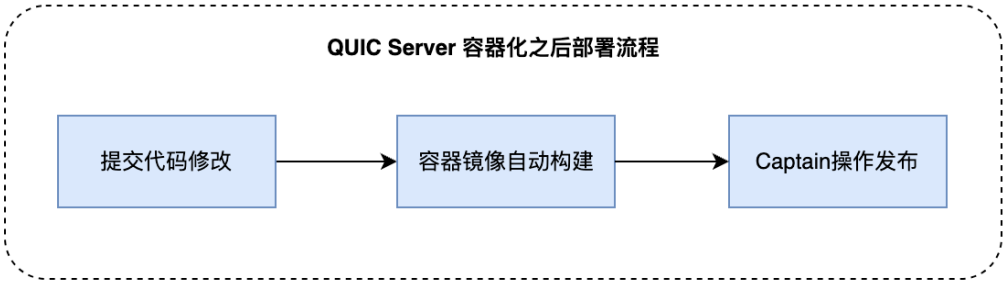
2.1 容器化改造

改造前我们统一使用 VM 部署 QUIC SLB 和 QUIC Server，具体部署流程如下：

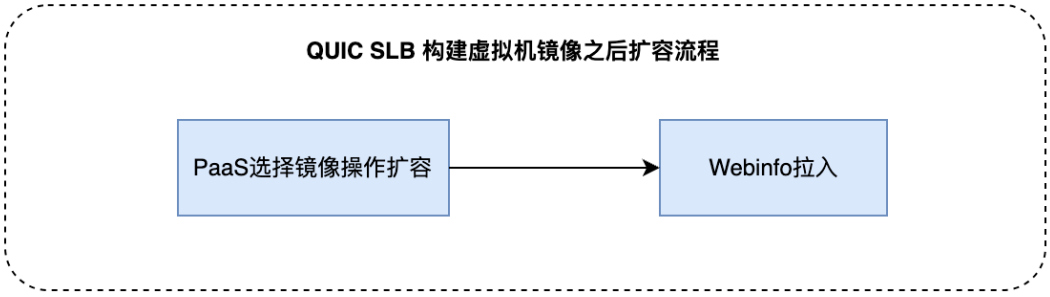


QUIC 实践早期我们经常需要在机器上执行自定义操作，这种部署方案的优点是比较灵活。但随着 QUIC 服务端功能趋于稳定，以及业务流量的日益增长，此方案部署时间长、不支持动态扩缩容的弊端日益显现。为了解决以上问题，我们对 QUIC SLB 以及 QUIC Server 进行了容器化改造：

QUIC Server 承载了 QUIC 协议处理以及用户 http 请求转发这两项核心功能，我们将其改造成了容器镜像，并接入到内部 Captain 发布系统中，支持了灰度发布，版本回退等功能，降低了发布带来的风险，同时具备了 HPA 能力，扩缩容时间由分钟级缩短到秒级



QUIC SLB 作为外网入口，主要负责用户 UDP 包的转发，负载较小，对流量变化不敏感。并且由于需要 Akamai 加速，QUIC SLB 需要同时支持 UDP 以及 TCP 协议，当前容器是无法支持双协议的外网入口的。因此我们将 QUIC SLB 改造成了虚拟机镜像，支持在 PaaS 上一键扩容，大大降低了部署成本



2.2 服务发现与主动健康监测

在 QUIC SLB 中，我们使用 Nginx 作为 4 层代理，实现 QUIC UDP 数据包的转发以及连接迁移能力。

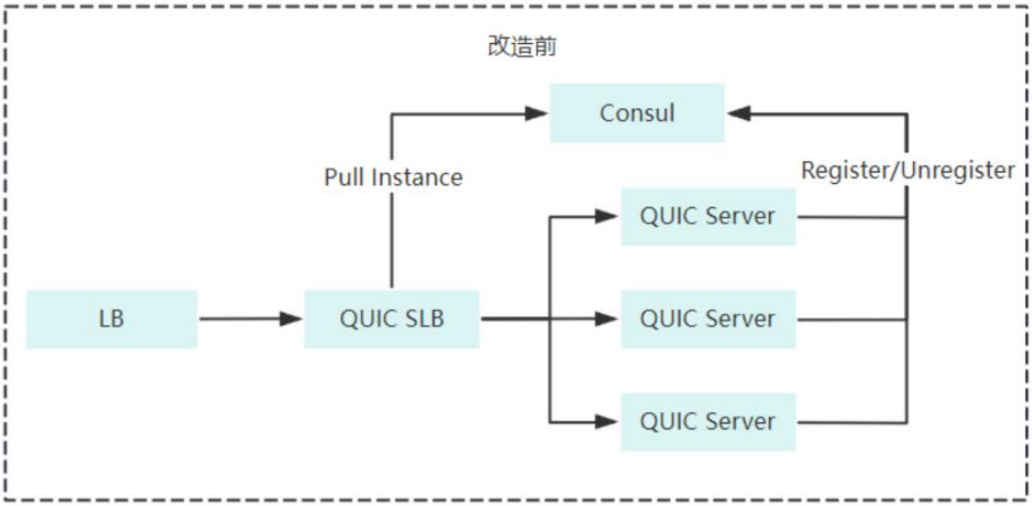
容器化后的前期我们使用 Consul 作为 QUIC Server 的注册中心，Server 会在 k8s 提供的生命周期函数 postStart 和 preStop 中分别调用 Consul 的注册和删除 API，将自身 ip 在 Consul 中注册或摘除。QUIC SLB 会监听 Consul 中 ip 的变化，从而及时感知到每个 Quic Server 的状态，并实时更新到 Nginx 的配置文件中，这样就实现了 QUIC Server 的自动注册与发现。

但在实际演练场景中，我们发现当直接对 QUIC Server 注入故障时，由于 Server 所在 pod 并没有被销毁，因此不会触发 preStop API 的调用，故障 Server 无法在 Consul 中摘除自身 ip，导致 QUIC SLB 无法感知到 Server 的下线，因此 QUIC SLB 的 nginx.conf 中依旧会保留故障的 Server ip，这种情况在 Nginx 做 TCP 代理和 UDP 代理时所产生的影响不同：

- 当 Nginx 做 TCP 代理时，Nginx 与故障 Server 之间建立的是 TCP 连接，Server 故障时 TCP 连接会断开，Nginx 会与故障 Server 重新建联但最终失败，此时会自动将其拉出一段时间，并每隔一段时间进行探测，直至其恢复，从而避免了 TCP 数据包转发至错误的 Server，不会导致服务成功率下降；

- 当 Nginx 做 UDP 代理时，由于 UDP 是无连接的，QUIC SLB 依旧会转发数据包至故障

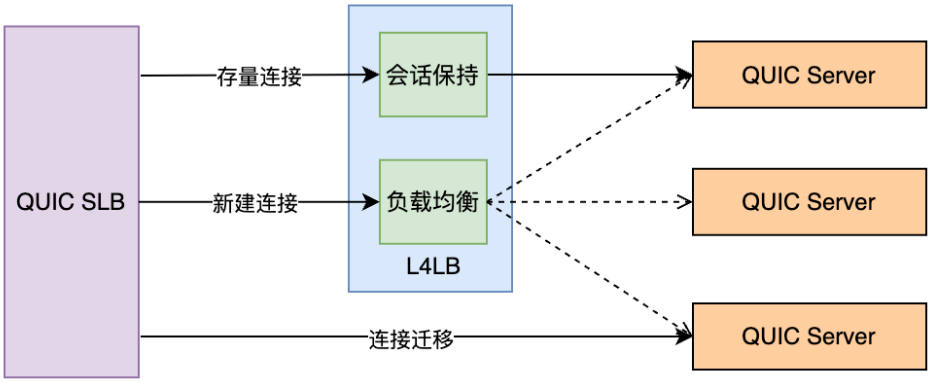
Server，但 SLB 不会收到任何响应数据包，由于 UDP 协议特性，此时 QUIC SLB 不会判定 Server 为异常，从而持续大量的 UDP 包被转发到故障 Server 实例上，导致 QUIC 通道的成功率大幅下降；



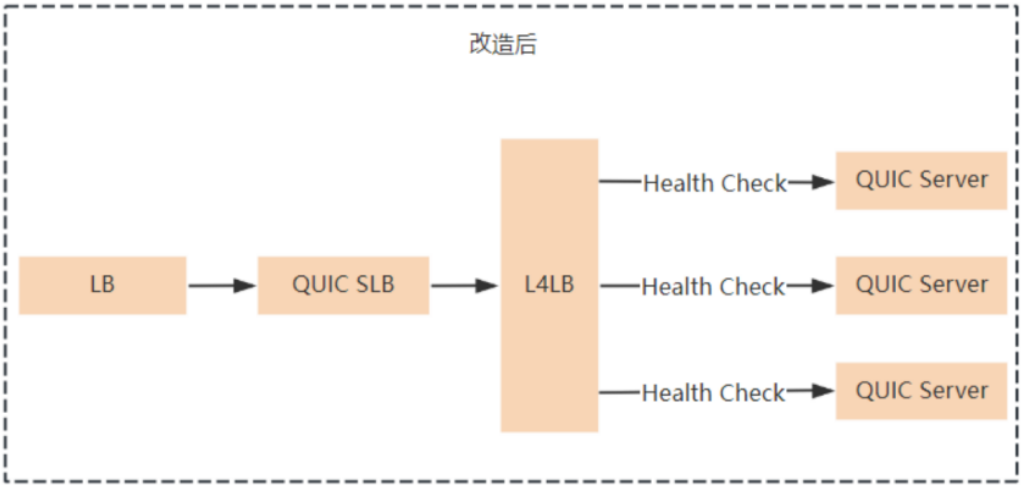
经过上述分析，我们知道了使用 UDP 进行健康监测存在一定弊端，期望使用 TCP 协议对 QUIC Server 进行主动的健康检测。所以开启了新方案的探索，其需要同时支持 UDP 数据转发，基于 TCP 协议的主动健康监测，支持服务发现与注册，并且能够较好的适配 QUIC SLB 层。

调研了很多方案，其中较适配的是开源的 Nginx UDP Health Check 项目，其同时支持 UDP 数据包转发和 TCP 的主动健康检测，但是其不支持 nginx.conf 中下游 ip 的动态变更，也就是不支持 QUIC Server 的动态上下线，这直接影响了 Server 集群的 HPA 能力，因此舍弃了这个方案。

最终通过调研发现公司内部的 L4LB 组件，既能同时支持 TCP 的主动健康检测和 UDP 数据包转发，还支持实例的动态上下线，完美适配我们的场景，因此最终采用了 L4LB 作为 QUIC SLB 和 QUIC Server 之间的转发枢纽。



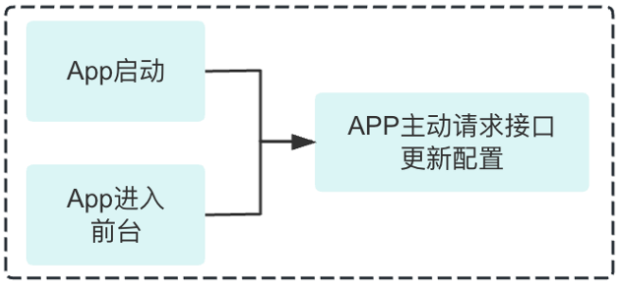
具体实现是为 QUIC Server 的每个 group 申请一个 UDP 的内网 L4LB 入口 ip，这些 ip 是固定不变的，那么对于 QUIC SLB 来说只需要将 UDP 数据包转发至固定的虚拟 ip 即可。L4LB 开启 TCP 的健康检测功能，这样当 group 中的 QUIC Server 实例故障时，健康检测失败，L4LB 就会将此实例拉出，后续 UDP 包就不会再转发到此实例上，直至实例再次恢复到健康状态。这样就完美解决了 QUIC Server 的自动注册与主动健康监测功能。



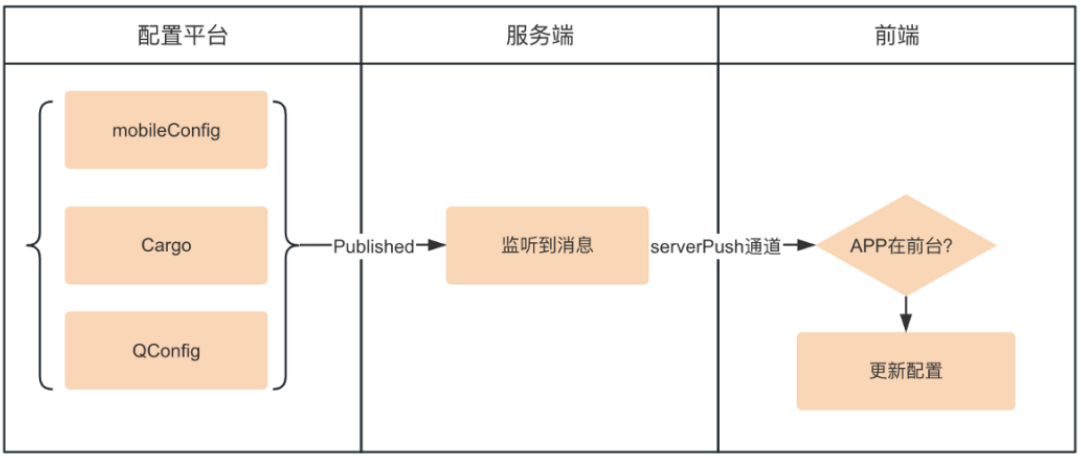
2.3 推拉结合方式提升客户端用户侧网络容灾能力

Trip.com App 的网络请求框架同时支持 QUIC/TCP/HTTP 三通道能力，其中 80%以上的用户请求都是通过 QUIC 通道访问服务器的，日均流量达到数亿，在现有的多通道/多 IP 切换能力的基础上，进一步提升容灾能力显得尤为重要。于是我们设计了一套推拉结合的策略方案，结合公司配置系统实现了秒级通道/IP 切换。下面是简化过程：

在客户端 App 启动和前后台切换等场景，根据变动情况获取最新的配置，网络框架基于最新的配置进行无损通道或 IP 的切换。



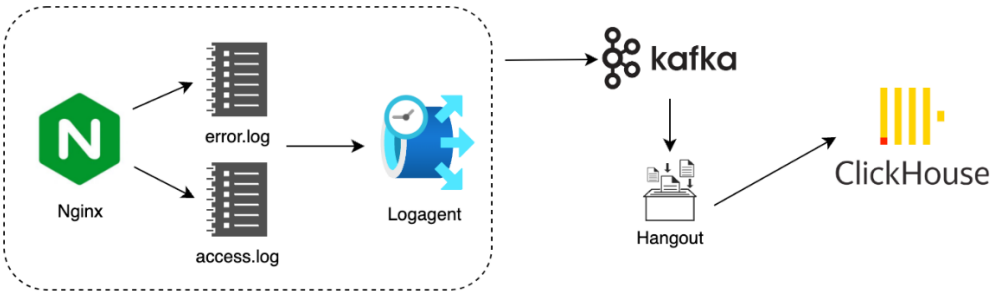
同时当用户 APP 处于前台活跃状态时，通过对用户进行主动的配置更新推送，让在线用户可以立即感知到变化并切换至最新的网络配置上面，且此切换过程对用户是无感的。



这样一来，我们的 QUIC 客户端网络框架进一步提升了容灾能力，当某个 IP 发生故障时，可以在秒级通知所有用户切离故障 IP，当某通道发生异常时，用户亦可以无感的切换至优质通道，而不会受到任何影响。

2.4 监控告警稳定性保证及弹性扩缩容指标建设

QUIC 数据监控系统的稳定性，对于故障预警、故障响应起到至关重要的作用。通过将埋点数据写至 access log 和 error log 来完成 QUIC 运行时埋点数据的输出，再通过 logagent 将服务器本地的日志数据发送至 Kafka，随后 Hangout 消费并解析，将数据落入 Clickhouse 做数据存储及查询，这给我们做运行时数据观察及数据分析提供了很大的便利性。

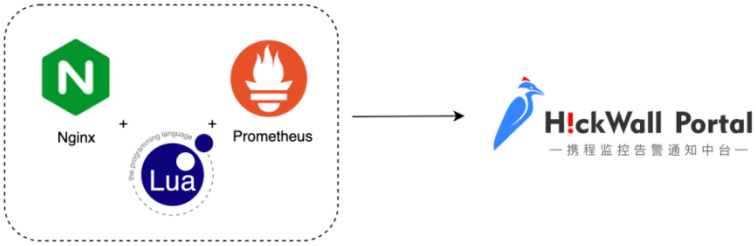


在完成 QUIC 网络架构升级之后，只依靠上述日志体系遇到了下面两类问题：

第一，在应用场景下，单纯依靠这部分数据做监控告警，偶发由于某些中间环节出现波动导致监控告警不准确， 例如 Hangout 消费者故障导致流量骤降或突增的假象，这可能会影响监控告警的及时性和准确性；

第二，容器化之后具备了弹性扩缩容能力，而 HPA 依赖于扩缩容数据指标，仅仅使用 CPU、内存利用率等资源指标，无法充足的反映 QUIC 服务器状态。根据 QUIC 服务特性，仍需要自定义一些 HPA 数据指标，例如空闲连接占比，空闲端口号占比等，以建立更合理、更稳定的扩缩容依赖；

基于上述两方面的考量，在经过调研之后，我们将 Nginx 与 Prometheus 进行整合，支持了关键数据指标、扩缩容数据指标通过 Prometheus 上报的方案。在 Nginx 中预先对成功率，耗时，可用连接数等等重要指标进行了聚合，只上报聚合数据指标，从而大大缩小了数据体量，使 Nginx 整合 Prometheus 的影响可以忽略不计。另外我们支持了空闲连接占比，空闲端口号占比等等 HPA 指标，使 QUIC 集群在流量高峰期，能够非常准确迅速的完成系统扩容，在低流量时间段，也能够缩容至适配状态，以最大程度的节约机器资源。



这样一来，QUIC 系统的监控告警数据来源同时支持 Prometheus 和 Clickhouse，Prometheus 侧重关键指标及聚合数据的上报，Clickhouse 侧重运行时明细数据的上报，两者相互配合互为补充。

在支持 Prometheus 过程中我们遇到了较多依赖项的版本搭配导致的编译问题，以 nginx/1.25.3 版本为例，给出版本匹配结果：

组件名	描述	版本	下载链接
nginx-quic	nginx官方库	1.25.3	https://github.com/nginx/nginx/releases/tag/release-1.25.3
nginx-lua-prometheus	lua-prometheus 语法库	0.20230607	https://github.com/knyar/nginx-lua-prometheus/releases/tag/0.20230607
luajit	lua即时编译	v2.1-20231117	https://github.com/openresty/luajit2/releases/tag/v2.1-20231117
lua-nginx-module	lua-nginx框架	v0.10.25	https://github.com/openresty/lua-nginx-module/releases/tag/v0.10.25
ngx_devel_kit	lua-nginx依赖的开发包	v0.3.3	https://github.com/vision5/ngx_devel_kit/releases/tag/v0.3.3
lua-resty-core	lua-resty核心模块	v0.1.27	https://github.com/openresty/lua-resty-core/releases/tag/v0.1.27
lua-resty-lrucache	lua-resty lru 缓存模块	v0.13	https://github.com/openresty/lua-resty-lrucache/releases/tag/v0.13

三、全链路埋点

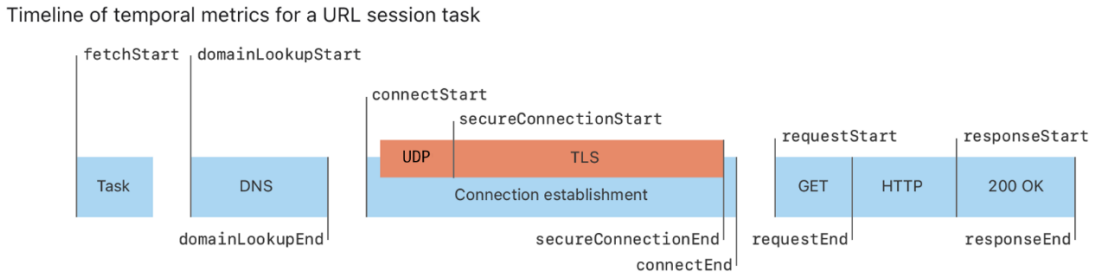
3.1 落地实践

我们基于优化用户链路耗时，寻找并优化耗时短板的目标出发，开始抽样分析耗时较久的请求，并根据所需，在服务端 access.log 中逐渐添加了较多的数据埋点，nginx 官方对单个 http 请求维度的数据埋点支持性较好，但仅仅分析单个请求维度的信息，难以看清请求所属连接的各类数据，仍需要观察用户连接所处网络环境，握手细节，数据传输细节，拥塞情况等等数据，来协助对问题进行定位。

QUIC 客户端之前仅有端到端的整体埋点数据，并且存在 QUIC 埋点体系和现有体系 mapping 的问题，我们收集过滤 Cronet metrics 的信息，整合进现有埋点体系内。

3.1.1 收集过滤 QUIC 客户端 Cronet Metrics 埋点数据

端到端流程支持了 DNS、TLS 握手、请求发送、响应返回等环节细粒度的埋点，QUIC 端到端各环节数据一目了然。



3.1.2 改造服务端 nginx 源码

我们在连接创建到连接销毁的全生命周期内进行了详细的数据埋点，另外通过连接 CID 实现了连接级别埋点和请求级别埋点数据的串联，这对进行问题定位，性能优化等提供了可靠的数据支持。下面分类列举了部分服务端全链路埋点，并简要概述了其用途：

1）连接生命周期时间线

连接类型(1-rtt/0-rtt)，连接创建时间(Server 收到 Client 第一个数据包的时间)，连接发送第一个数据包的时间，连接收到第一个 ack 帧的时间，连接握手耗时，连接收到及发送 cc 帧(Connection Close Frame)的时间，连接无响应超时时间，连接销毁时间等等；这一类埋点主要帮助我们理清用户连接生命周期中的各个关键时间点，以及握手相关的耗时细节。

2）数据传输细节

(以下皆为连接生命周期内)发送和接收字节、数据包、数据帧总数，包重传率，帧重传率等等。这类数据帮助分析我们的数据传输特性，对链路传输优化，拥塞控制算法调整提供数据参考。

3）RTT(Round-trip time)和拥塞控制数据

平滑 RTT，最小 RTT，首次和最后一次 RTT，拥塞窗口大小，最大 in_flight，慢开始阈值，拥塞 recovery_start 时间等等。这些数据可用来分析用户网络状况，观察拥塞比例，评估拥塞控制算法合理性等等。

4）用户信息

客户端 ip，国家及地区等。这帮助我们对用户数据进行区域级别的聚合分析，找到网络传输的区域性差异，以进行一些针对性优化。

3.2 分析挖掘

通过合并聚合各类数据埋点，实现了 QUIC 运行时数据可视化透明化，这也帮助我们发现了诸多问题及优化项，下面列举几项进行详述：

3.2.1 0-rtt 连接存活时间异常，导致重复请求问题

通过筛选 0-rtt 类型的连接，我们观察到此类连接的存活总时间，恰好等于 QUIC 客户端和服务端协商之后的 max_idel_timeout，而 max_idel_timeout 的准确定义为"连接无响应（客户端服务端无任何数据交互）超时关闭时间"，也就是说正常情况下，当一个连接上最后一次 http 请求交互完毕之后，若经过 max_idel_timeout 时间仍未发生其他交互时，连接会进入关闭流程；当连接上不断的有请求交互时，连接的存活时间必定大于 max_idel_timeout(实际连接存活时间 = 最后一次请求数据传输完成时间 - 连接创建时间 + max_idel_timeout)。

为了论证上述现象，我们通过连接的 dcid，关联筛选出 0-rtt 连接生命周期中所有 http 请求列表，发现即使连接上的请求在不断的进行，0-rtt 连接仍然会在存活了 max_idel_timeout 时无条件关闭，所以断定 0-rtt 连接的续命逻辑存在问题。我们对 nginx-quic 源码进行阅读分析，最终定位并及时修复了问题代码：

```
void
ngx_quic_run(ngx_connection_t *c, ngx_quic_conf_t *conf)
{
    ngx_int_t               rc;
    ngx_quic_connection_t  *qc;

    ngx_log_debug0(NGX_LOG_DEBUG_EVENT, c->log, 0, "quic run");
    rc = ngx_quic_handle_datagram(c, c->buffer, conf);
    if (rc != NGX_OK) {
        ngx_quic_close_connection(c, rc);
        return;
    }
    /* quic connection is now created */
    qc = ngx_quic_get_connection(c);

    ngx_add_timer(c->read, qc->tp.max_idle_timeout);
    ngx_add_timer(&qc->close, qc->conf->handshake_timeout);

    ngx_quic_connstate_dbg(c);
    c->read->handler = ngx_quic_input_handler;
    return;
}
```

在源码 ngx_quic_run()函数中，存在两个连接相关的定时器：

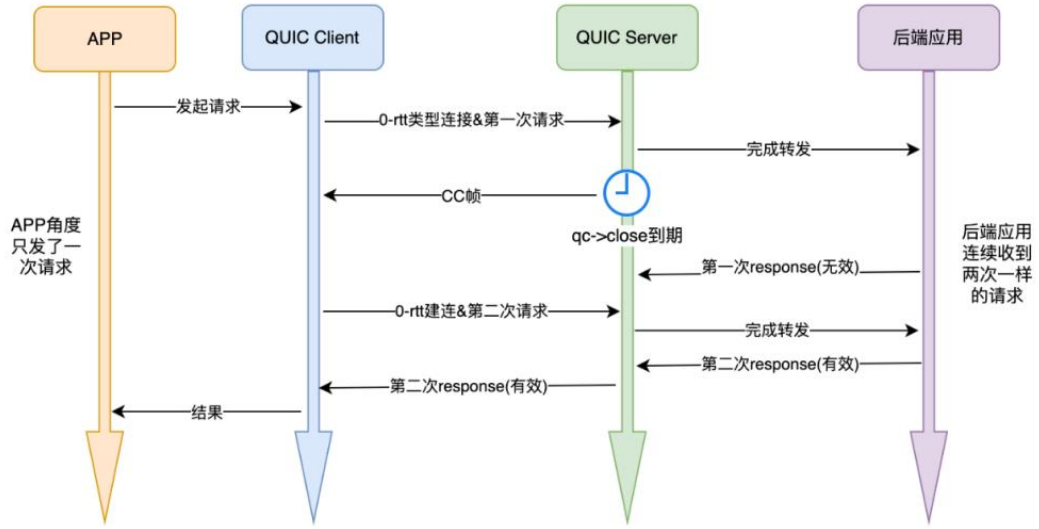
```
ngx_add_timer(c->read, qc->tp.max_idle_timeout);
ngx_add_timer(&qc->close, qc->conf->handshake_timeout);
```

两者都会影响连接的关闭，其中 c->read 定时器存在续命逻辑，会随着连接生命周期内，请求的不断发生而刷新定时器。qc->close 在源码中不存在续命逻辑，有且仅有一处删除逻辑，即在执行 ngx_quic_handle_datagram()函数过程中，若完成了 ssl 初始化，则调用 ngx_quic_do_init_streams()进行 qc->close 定时器的删除操作；

- 若为 1-rtt 建联，第一次执行 ngx_quic_handle_datagram()函数不会完成 ssl 初始化，所以 qc->close 的创建发生在 ssl 初始化完毕之前，在后续数据包交互过程中能够正常完成删除逻辑；

- 若为 0-rtt 建联，第一次执行 ngx_quic_handle_datagram()函数会完成 ssl 初始化逻辑，所以仅一次的删除逻辑，发生在了 qc->close 定时器设置之前，所以导致 qc->close 不能被正常移除，从而导致 max_idel_timeout 时间一到，连接立即关闭的现象；

这个 bug 除了导致较多无效 0-rtt 新建连之外，在我们的应用场景下，经过对全链路埋点数据聚合分析发现，还会导致重复请求问题，下面介绍发生重复请求的原因：



quic client 发起某个 request 的第一次请求，quic server 端收到并转发给后端应用，在 server 收到后端应用 response 之前，恰好 qc->close 定时器到期，导致 server 立即向 client 发送 cc 帧，client 在收到 cc 帧后，按照 quic 协议应无条件立即关闭当前连接，所以 client 认为第一次请求失败，从而发起新建连，开始第二次请求，从而导致重复请求问题。而这个过程从客户端业务角度来说只请求了一次，但后端应用却连续收到两次一模一样的请求，这对于幂等性要求较高的接口影响较大。

我们在 2024 年 2 月发现并修复了上述定时器 bug，修复之后连接复用比例提升 0.5%，不必要的 0-rtt 建连比例降低 7%。目前 nginx-quic 官方分支于 2024/04/10 也提交了对此问题的修复，Commit 链接：https://hg.nginx.org/nginx-quic/rev/155c9093de9d

3.2.2 客户端 App Cronet 升级给 95 线用户带来体验提升

经过数据分析发现，长尾用户在 0-RTT 上占比不高，大多为 1-RTT 新建连接，经过分析判断可能和 QUIC 客户端 Cronet 裁剪有关。Trip.com App 上 Cronet 库在优化前使用的是 2020 年的旧版本，并且由于包大小问题对其进行了裁剪（比如：重构了 PSK 相关逻辑，转为 session 级别），在保留关键功能的前提下尽可能剔除了无用代码十几万行。同时经过与其他 Cronet 使用方沟通，得到 Cronet 升级后有不错的性能提升表现，于是时隔近四年，客户端对 Cronet 库进行了一次大升级。

另外，由于 Chromium 官方在 2023 年 11 月份官宣不再提供 iOS 的 Cli 工具，所以此次升级目标就定位选一个尽可能靠近官方删除 iOS 构建工具之前的版本，最终我们选定了 120.0.6099.301。

chromium / chromium / src / **refs/tags/120.0.6099.301**

```
commit  7a11e66c1b2c35ab2bb18e5e87d91eae981fc7f3                                [log] [tgz]
author  Keren Zhu <kerenzhu@chromium.org>                                       Thu Feb 29 22:42:19 2024
committer  Chromium LUCI CQ <chromium-scoped@luci-project-accounts.iam.gserviceaccount.com> Thu Feb 29 22:42:19 2024
tree  639790651ba71348b0ae661b4f0444da51fb2b3f
parent  7d9a4b5fec6cc4b9b2336c91e82565bcc7cc63c5 [diff]
```

经过 Cronet 升级及相关适配性改造，对线上升级前后版本进行对比，升级后用户侧 95 线请求耗时降低了 18%。

3.2.3 Nginx-quic 分支中拥塞控制算法实现具有较大的优化空间

通过对 nginx-quic 源码的研究以及链路埋点的数据分析，我们发现源码中的拥塞控制算法为 Reno 算法的简化版，初始传输窗口设置较大为 131054 字节(若 mtu 为 1200，初始就有 109 个包可以同时传输)。若发生拥塞事件，降窗的最小值仍为 131054 字节，在网络较好时，网络公平性不友好，在网络较差时，这会加剧网络拥堵。在发现此问题后，我们开始着手改造源码中的拥塞控制算法逻辑，这一优化内容将在第四部分详述。

四、拥塞控制算法探索

nginx-quic 官方分支中，对于拥塞控制算法的实现目前仍处于 demo 级别，我们结合 QUIC 在应用层实现拥塞控制算法而不依赖于操作系统的特性，对 Nginx 官方代码中拥塞控制相关逻辑进行了抽象重构，以方便拓展各种算法，并支持了可配置式的拥塞控制算法切换，可以根据不同的网络状况，不同的 server 业务场景配置不同的拥塞控制算法。

4.1 拥塞控制算法简介

目前主流的拥塞控制算法大抵可分为两类，一类是根据丢包做响应，如 Reno、Cubic，一类是根据带宽和延迟反馈做响应，如 BBR 系列。这里简要介绍下 Reno，Cubic 和 BBR 的工作原理，适用场景及优缺点：

1）Reno 算法是 TCP 最早的拥塞控制算法之一，基于丢包的拥塞控制机制。它使用两个阈值（慢启动阈值和拥塞避免阈值）来控制发送速率。在慢启动阶段，发送方每经过一个往返时间（RTT），就将拥塞窗口大小加倍。一旦出现拥塞，会触发拥塞避免阶段，发送速率会缓慢增长。当发生丢包时，发送方会认为发生了拥塞，将拥塞窗口大小减半；适用于低延时、低带宽的场景，对于早期互联网环境比较适用。其优点是简单直观，易于理解和实现。缺点是对网络变化反应较慢，可能导致网络利用率不高，且在丢包率较高时性能不佳。

2）Cubic 算法在 Reno 算法的基础上进行改进，利用网络往返时间（RTT）和拥塞窗口的变化率来计算拥塞窗口的大小，使用拟立方函数来模拟网络的拥塞状态，并根据拥塞窗口的大小和时间来调整拥塞窗口的增长速率。适用于中度丢包率的网络环境，对于互联网主流环境有较好的性能表现。优点是相对于 Reno 算法，能更好地适应网络变化，提高网络利用率。缺点是在高丢包率或长肥管道环境下，发送窗口可能会迅速收敛到很小，导致性能下降。

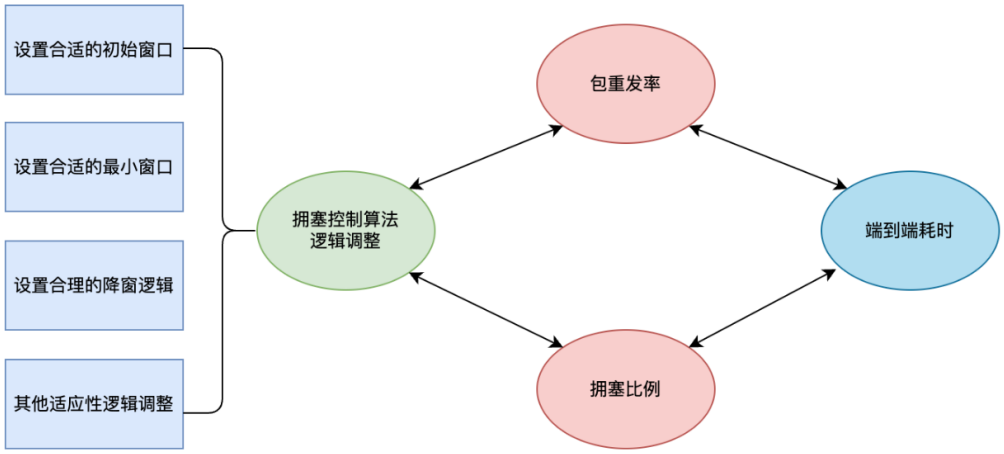
3）BBR（Bottleneck Bandwidth and RTT）算法是 Google 开发的一种拥塞控制算法，通过测量网络的带宽和往返时间来估计网络的拥塞程度，并根据拥塞程度调整发送速率。BBR 算法的特点是能够更精确地估计网络的拥塞程度，避免了过度拥塞和欠拥塞的情况，提高了网络的传输速度和稳定性。适用于高带宽、高延时或中度丢包率的网络环境。优点是能够更精确地控制发送速率，提高网络利用率和稳定性。缺点是可能占用额外的 CPU 资源，影响性能，且在某些情况下可能存在公平性问题。

4.2 优化实现及收益

源码中有关拥塞控制逻辑的代码分散在各处功能代码中，未进行抽象统一管理，我们通过梳理拥塞控制算法响应事件，将各个事件函数抽象如下：

```
typedef struct ngx_quic_cg_ctrl_callback_s {
    /* 获取当前拥塞控制算法 */
    char *(*ngx_quic_cg_ctrl_get_type)(void);

    /*  初始化环节用来分配内存空间 */
    size_t (*ngx_quic_cg_ctrl_size)(void);

    /* 拥塞控制算法初始化 */
    void (*ngx_quic_cg_ctrl_init)(ngx_connection_t *c);

    /* 发生丢包事件回调 */
    void (*ngx_quic_cg_ctrl_on_lost)(ngx_connection_t *c, ngx_quic_frame_t *f);

    /* 收到ack包事件回调 */
    void (*ngx_quic_cg_ctrl_on_ack)(ngx_connection_t *c, ngx_quic_frame_t *f);

    /* 获取当前拥塞窗口，用以记录数据或判断当前包是否能够正常发送 */
    uint64_t (*ngx_quic_cg_ctrl_get_cwnd)(ngx_connection_t *c);

    /* 拥塞窗口重置 */
    void (*ngx_quic_cg_ctrl_reset_cwnd)(ngx_connection_t *c);

    /* 获取当前慢开始阈值 */
    size_t (*ngx_quic_cg_ctrl_get_ssthresh)(ngx_connection_t *c);

    /* 获取当前recovery_start */
    ngx_msec_t (*ngx_quic_cg_ctrl_get_recovery_start)(ngx_connection_t *c);

    /* 连接处于 slow start 状态 */
    int (*ngx_quic_cg_ctrl_in_slow_start)(ngx_connection_t *c);

    /* 连接处于 recovery state 状态 */
    int (*ngx_quic_cg_ctrl_in_recovery)(ngx_connection_t *c);

    /* 多包ack事件回调 */
    void (*ngx_quic_cg_ctrl_on_ack_multiple_pkts)(ngx_connection_t *c);

    /* BBR初始化 */
    void (*ngx_quic_cg_ctrl_init_bbr)(ngx_connection_t *c, ngx_sample_t *sampler);
} ngx_quic_cg_ctrl_callback_t;
```

我们基于上述抽象结构，先后实现了目前主流的拥塞控制算法 Reno，Cubic 及 BBR，并且我们根据埋点数据对拥塞控制算法进行了参数和逻辑调优，包括设置合理的初始窗口和最小窗口，设置最优的拥塞降窗逻辑等等，这些调整会引起包重发率和连接拥塞率的数据变化，而这些数据变化皆会影响到链路传输性能。



我们通过对 QUIC 拥塞控制算法的优化，SHA 环境连接拥塞比例降低了 15 个点，实现了 SHA 端到端耗时降低 4%的收益。后续将继续基于 Trip.com 的数据传输特性，根据每个 IDC 的不同网络状况进行适应性定制化开发，并通过长期的 AB 实验，探索每个 IDC 下最优的拥塞控制逻辑。

五、成果与展望

我们不断的优化 QUIC 链路性能，不断的提升 QUIC 通道稳定性，旨在为 Trip.com 日益增长的业务提供优质的网络服务，同时我们也在不断的探索支持更多的 QUIC 应用场景。

1）通过容器化改造，将扩缩容由手动改为自动，扩缩容时间缩短 30 倍，在 20s 内就能拉起并上线大批服务器；

2）通过开发全链路埋点，聚合分析出了较多优化项，修复 0-rtt 连接问题，连接复用比例提升 0.5%，优化拥塞控制算法，端到端耗时缩短 4%；

3）通过在 FRA(法兰克福)部署 QUIC 集群，降低了欧洲用户耗时 20%以上，提高了网络成功率 0.5%以上；

4）支持了携程旅行 App 和商旅海外 App 接入 QUIC，国内用户和商旅用户在海外场景下网络成功率和性能也大幅提升；

5）客户端升级 Cronet 之后，综合上述优化项，用户侧端到端整体耗时 95 线降低 18%；

Trip.com 在 QUIC 上的探索将持续的进行，我们将密切关注社区动态，探索支持更多的 QUIC 应用场景，挖掘更多的优化项目，为携程国际化战略贡献力量。

携程搜广推算法策略开发平台

【作者简介】携程搜广推中台框架（Eagle）技术团队。

在携程的搜广推业务中，Eagle 技术生态扮演着核心角色，不断地应对业务扩展带来的新挑战。本文首先剖析了 Eagle 算法策略平台的架构创新，包括流程组件化、编排可视化和逻辑算子化，这些设计有效提高了开发效率并确保生产稳定性。进一步通过推荐信息流业务的实践案例，展示了策略平台在性能提升和资源优化方面的实际效益。最后，对平台的未来发展进行了展望，包括优化调度、增强编排、灵活部署、全链路监控和提升用户体验等，以持续引领技术创新，满足业务发展需求。

一、背景

目前 Eagle（携程搜广推中台框架）技术生态在集团多个业务线搜广推业务中发挥了关键作用。它在模型训练/推理，特征生产/服务，在线策略服务、分层实验以及运维监控等方面提供了统一且易用的基础框架，显著提升了各业务团队在研发效率和业务效果上的表现。

在效率提升方面的核心点包括：首先它改变了传统的算法和开发协作模式，使得算法同学能够直接参与到线上召回和重排模块算法策略代码的开发，大幅降低了沟通成本和一致性问题。其次，通过分层实验平台实现了 A/B 实验参数化、配置化高效做实验。这在一定程度上提升了策略逻辑的透明性和实验效率。然而，随着业务场景的扩展、业务需求量和复杂度的增加，同时更多的算法和产品同学参与进来，在代码质量、参数管理以及沟通协作出现了一些新的问题。这些对工程质量、迭代效率和性能带来了新的挑战。主要体现在以下几个方面：

1）代码冗余、参数爆炸：由于各场景在召回和重排阶段存在策略逻辑的相似性、导致了大量的代码复制现象。这种状况不仅使得服务变得过于臃肿，也大幅增加了维护成本和迭代难度。

2）逻辑黑盒和沟通成本：业务逻辑的掌握仅限于开发者本人，其他人一般都是通过文档和口述同步逻辑。反之则需要投入大量时间来理解和掌握现有的代码逻辑，这无疑增加了团队的协作难度和沟通成本。比如，日常的 case Debug 和策略解释路径很长，产品找算法找开发一层层传递，效率和沟通成本问题严重。

3）迭代风险与难度：代码缺少流程和模块化设计，比如：某一个召回策略逻辑几十行甚至几百行代码都写在一个类里面。这样当需要更改这块逻辑时，无疑增加了出错的风险，也使得迭代过程变得更加困难和低效。

4）质量和性能问题：算法同学（包括一部分工程同学）工程能力层次不齐，算法同学更多的是关注策略逻辑的实现，在代码设计和质量没有太多的优化经验。导致上线运行时的各种问题逐渐显现，如：时空复杂度、频繁 GC、潜在的代码漏洞，系统的稳定性、资源利用率以及性能瓶颈等问题等。

二、算法策略平台是什么？

Eagle 算法策略平台是一个高度配置化和透明化的算法策略开发和部署平台。平台专为搜索、广告和推荐系统业务领域设计，通过实现搜广推全流程的配置化和透明化，简化算法策略的迭代流程，使得算法团队能够更加专注于策略的效果优化和业务目标的实现。平台提供了视图和工具，使算法团队能够直观地理解和监控流程中的每个环节（数据召回、排序逻辑、模型预测等），同时集成了自动化测试、实时监控和告警、分层实验和问题排查功能，确保了平台服务的高稳定性和可靠性。

三、整体设计

策略开发平台为用户提供一套从算子开发，任务编排到策略上线的完整解决方案，实现了策略上线流程标准化平台化管理，沉淀策略通用能力。策略开发人员可以根据不同的业务场景对相应的业务流程进行编排和发布。相对于传统的策略开发上线流程，该方案具有以下优势：

- 流程组件化：将策略上线流程划分为三个阶段：算子开发，策略编排，任务运行；三阶段分别封装为完全解耦的三个组件，使方案整体具有高扩展性和低成本维护。

- 编排可视化：策略的编排采用标准的 DAG 模型，在页面直观的展现策略节点逻辑信息及节点间的逻辑关系，实现了策略逻辑完全透明。

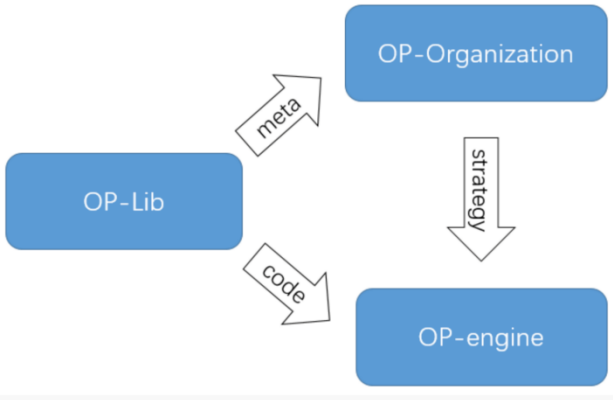
- 逻辑算子化：框架提供一套统一的 OP 算子接口，实现了每个算子参数协议独立，功能单一，进一步减少策略代码的冗余度和提升代码的复用性。

因此，我们将整个开发平台划分为三部分：

1）OP-Organization：通过可视化的页面管理逻辑算子（OP-Lib）代码库，策略组件管理，策略逻辑编排（DAG）。执行策略标准化上线流程；

2）OP-Lib：实现统一 OP 接口的代码库。通过接口及注解定义了统一的 OP 代码开发规范，元数据声明，耗时及异常实时监控。大幅提高代码的开发效率和复用性；

3）OP-engine：实时监听策略配置（DAG）变更，支持 DAG 的自动优化（节点合并），动态裁剪，节点限流，熔断，实时监控，数据回流等功能，确保策略高效运行；



3.1 策略编排：简单，直观，安全，高效

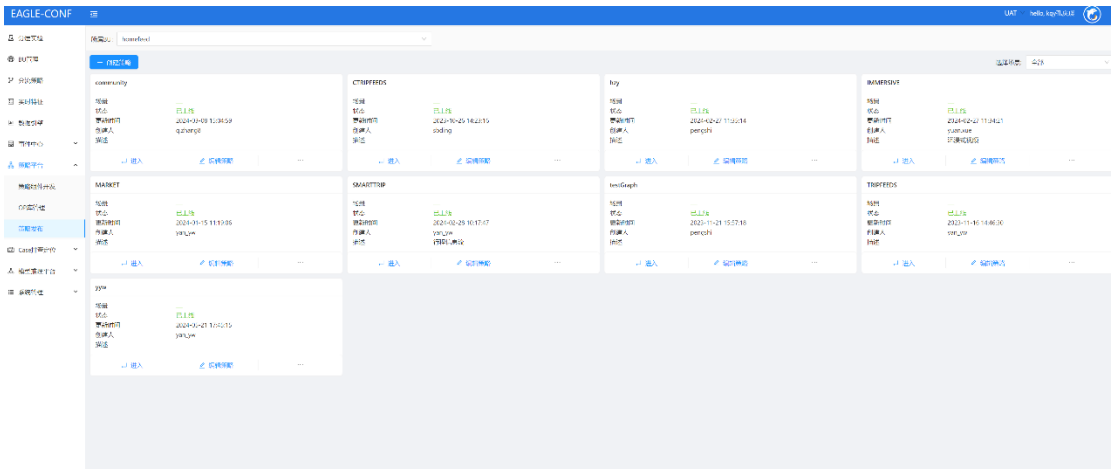
策略逻辑作为支撑线上效果的主要逻辑，需要高效的迭代上线，不断的验证正向收益，在底层的框架支持上，已有分层实验平台和 ABtest 实验可以满足高效的实验和结果验证，但在涉及到逻辑执行层，没有统一的管理平台，以适应策略的高频迭代，为解决上述痛点，设计并开发了策略开发平台。在搜广推场景中，针对不同的调用阶段，将整体流程分为了召回，粗排，精排，重排几个阶段，下面就以召回阶段的视角来介绍策略开发平台的特点。

3.1.1 策略编排可视化，所见即所得

召回首先要解决的问题是如何将策略逻辑透明化，需要宏观视角去呈现线上服务运行的策略链路，它们之间的调用关系是什么？如何快速地调整不合理的策略逻辑。

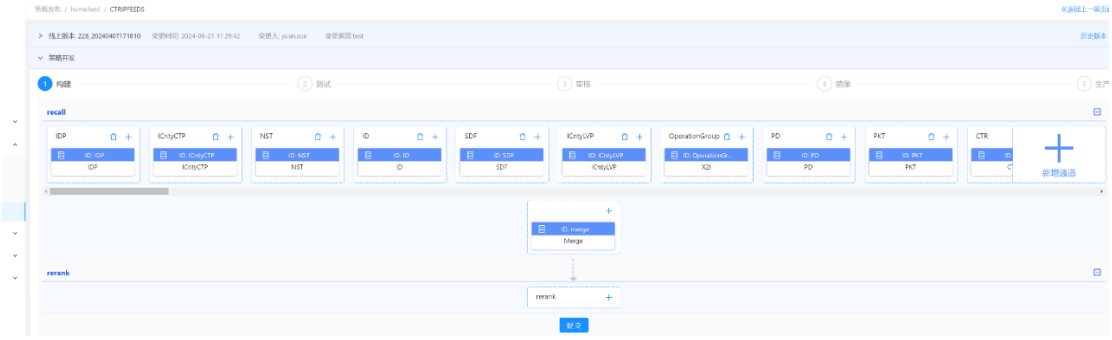
3.1.1.1 应用场景可视化

进入 BU 下的策略首页，即可看见以卡片形式展示出的策略梗概信息(场景，上线状态，更新时间等)，拥有权限人员可以编辑对应卡片。



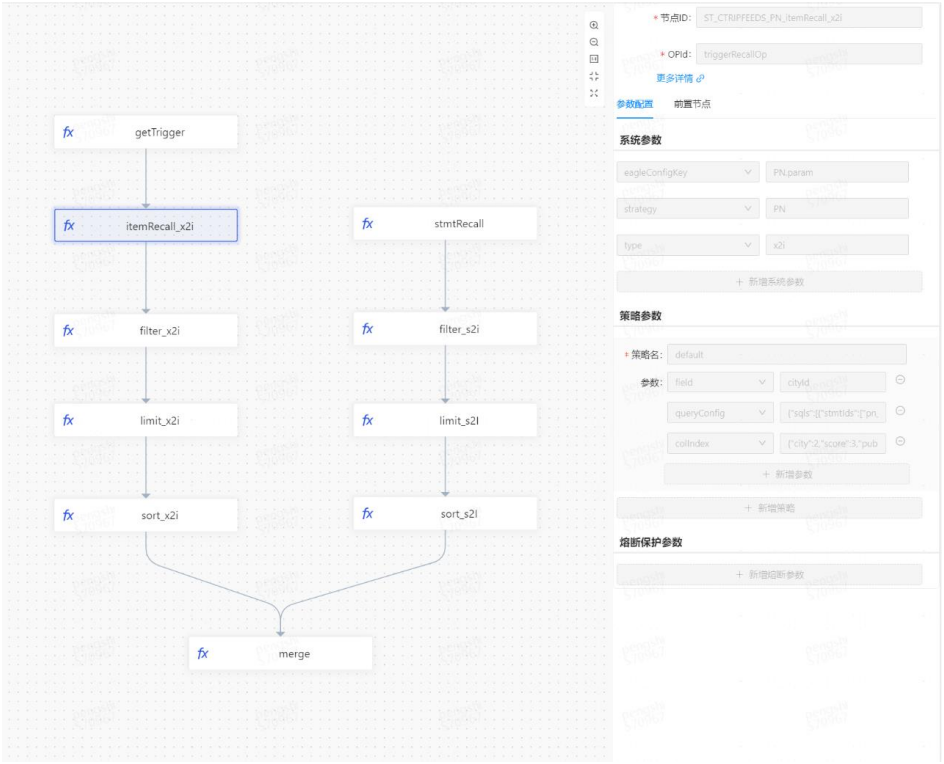
3.1.1.2 策略组件可视化

进入具体策略，可以查看编辑该策略，同时可以追踪、回退历史版本。在当前页面中，为了增加策略逻辑的可读性，使用自定义组件将逻辑功能相同的策略节点封装在一起，使用户对逻辑策略的认识更加直观。例如下图中，在召回阶段按召回通道将策略封装成一个个并行组件，使用户直观了解到该策略下的所有召回通道。



3.1.1.3 策略节点可视化

组件中的策略节点是平台中的最小单元，是由策略 OP 算子（OP-Lib 代码库）和策略参数组成。通过此页面不仅可以方便的选择一个 OP 来生成策略节点，同时可以以拖拽的方式编排各个节点的调用关系，通过 OP 算子上报了的元数据信息，平台确保节点调用关系的正确性，保证策略上线的安全。



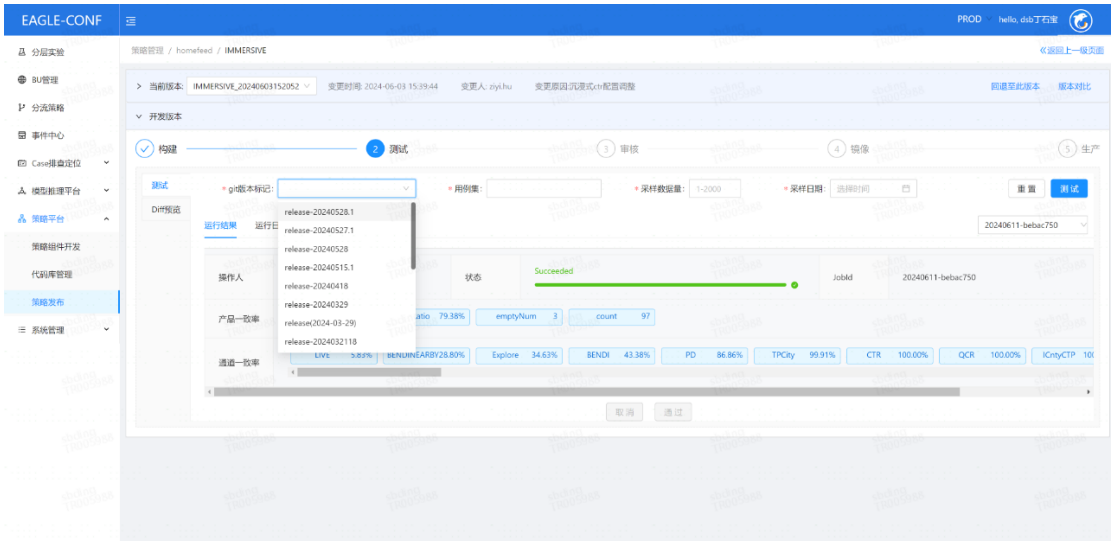
3.1.2 标准化上线流程，提升上线效率

标准化上线流程在提升上线效率的同时，保证每次配置发布都按照规定的步骤和规范进行，降低人为错误的风险，并通过验证和测试来提高配置发布的质量和稳定性。

3.1.2.1 自动化测试

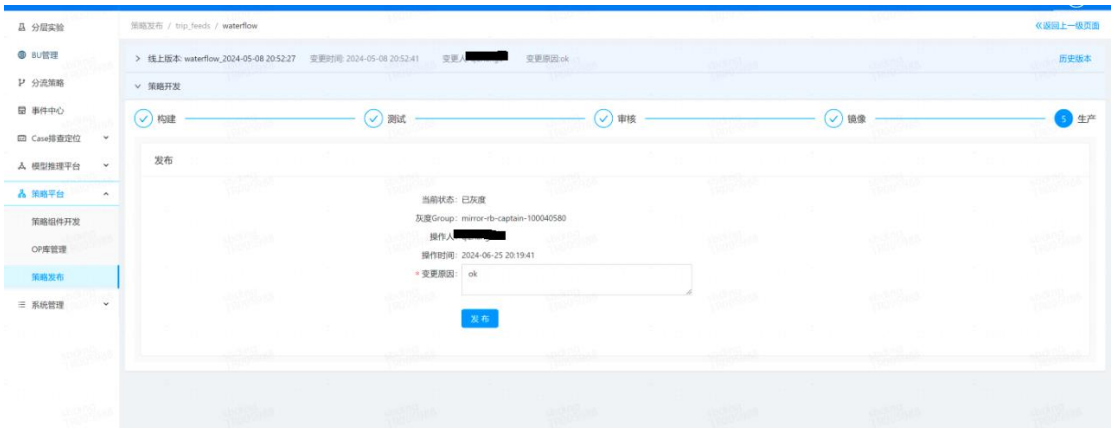
自动化测试工具将服务代码打包成 Docker 镜像并部署到 k8s 容器中。根据线上服务的历史

请求数据，构建请求报文并发送到该服务。获取到结果后，对不同配置获取到的结果进行比较。最终生成的测试结果将作为策略开发人员判断配置变更对服务性能、功能、稳定性的影响的评估依据。如果测试结果不符合预期，策略开发人员可以及时调整配置，确保服务可以正常运行。



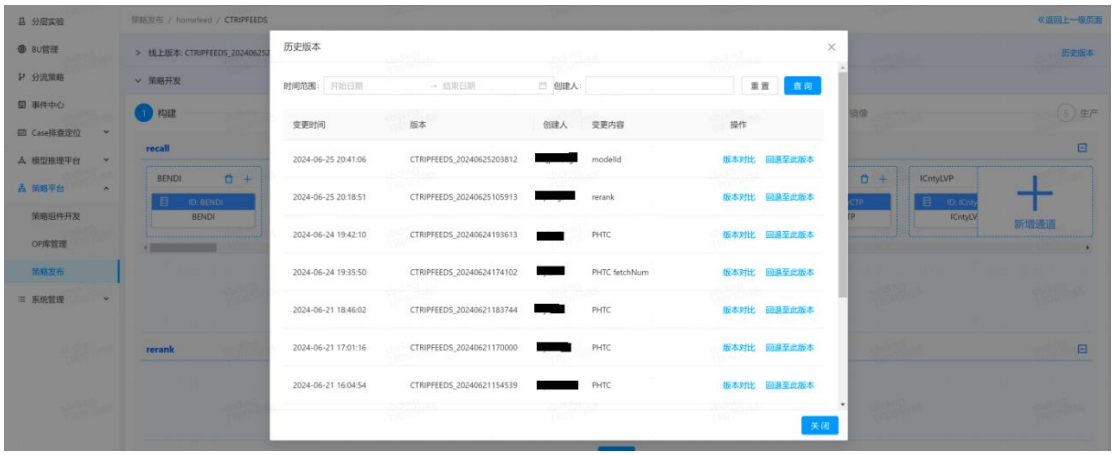
3.1.2.2 灰度发布

策略开发人员可以将新配置发布到镜像集群或者测试集群，在正式上线前尽可能发现并解决潜在的问题，确保基于新配置的服务能够稳定运行，功能和各项指标符合预期。当新配置通过测试和验证，即可发布到生产环境。



3.1.2.3 配置回滚

当配置发布到生产环境后不符合预期或者出现问题，可以通过回滚功能将配置回退到上一个稳定版本，降低配置对服务的影响。



3.1.3 先设计、再开发（Design First）

平台提供了在线编排可视化功能，极大降低了用户上线策略的学习成本，用户只需对策略 OP 进行编排，即可完成策略上线。这种开发模式的转变，要求用户更多的去思考如何合理编排策略 OP，以及如何设计可复用的策略 OP，避免了老的开发模式（边开发边设计）带来的需求演变、迭代时大量修改和补丁的问题。

同时，策略 OP 的动态组合也带来了一些问题：1）如何确保组合后的各个 OP 能正常调用，避免出现不合理的调用关系甚至参数类型不兼容，2）编排好的 DAG 图在运行时却找不到挂载的 OP 实例。在高并发流量的首页信息流场景中，每一个问题都足以致命，为了解决上线的安全性，这就需要依靠完善的 OP 元数据上报机制以及 OP 代码库的版本验证来保证。

3.1.3.1 OP 开发流程和规范

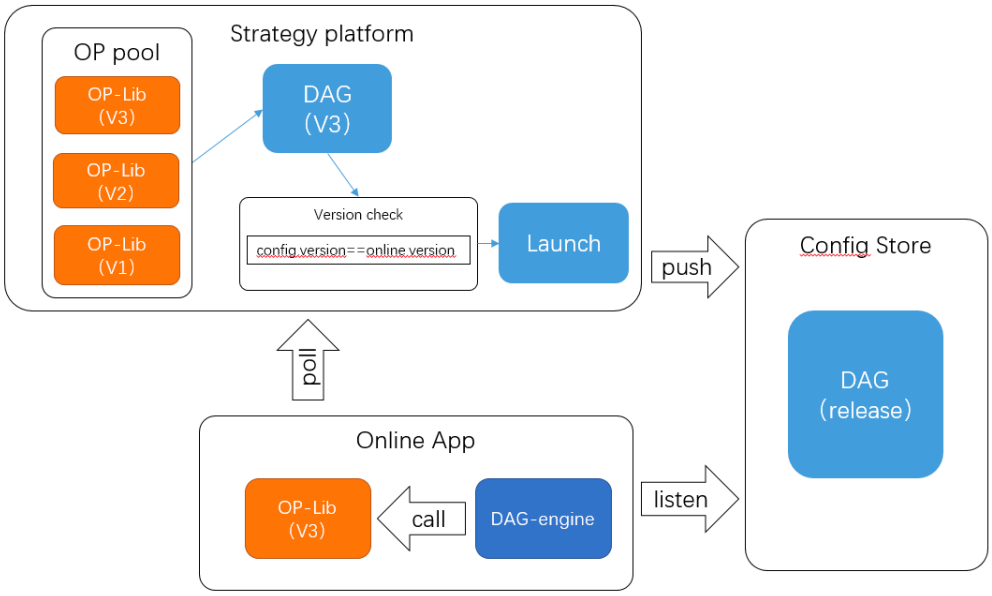
为了保证用户在操作策略节点编排时的安全性，在设计 OP 时，首先定义元数据信息，包括但不限于：OP 类信息，输入输出类型和校验逻辑、方法详细功能描述等。当 OP 设计完成后再是代码开发。

3.1.3.2 OP 代码库版本验证

由于 OP 代码库在线上服务中为预加载方式，即线上 OP 实例在一个运行周期内是完全恒定且单例的，需要在配置平台在推送策略 DAG 异步上线时确保每个 DAG 节点挂载的 OP 实例在服务中是存在的，平台使用的方案是在推送策略变更前做一次版本验证，保证线上 OP 版本与配置版本完全一致。

OP 代码版本验证流程图

策略发布代码验证机制



i. 用户编排策略（DAG）时从 OP 池选定某一个 OP-Lib 代码版本构图，如选择最新版 V3。

ii. 策略发布时进入版本验证流程（version check），该流程从在线服务中拉取当前线上 OP 代码版本与构图 OP 版本对比，版本相同时策略进入 launch 流程推送至配置存储系统（Config store）。

iii. 在线应用实时监听，将内存中的策略配置（DAG）更新，完成一次策略发布。

3.1.4 数据引擎

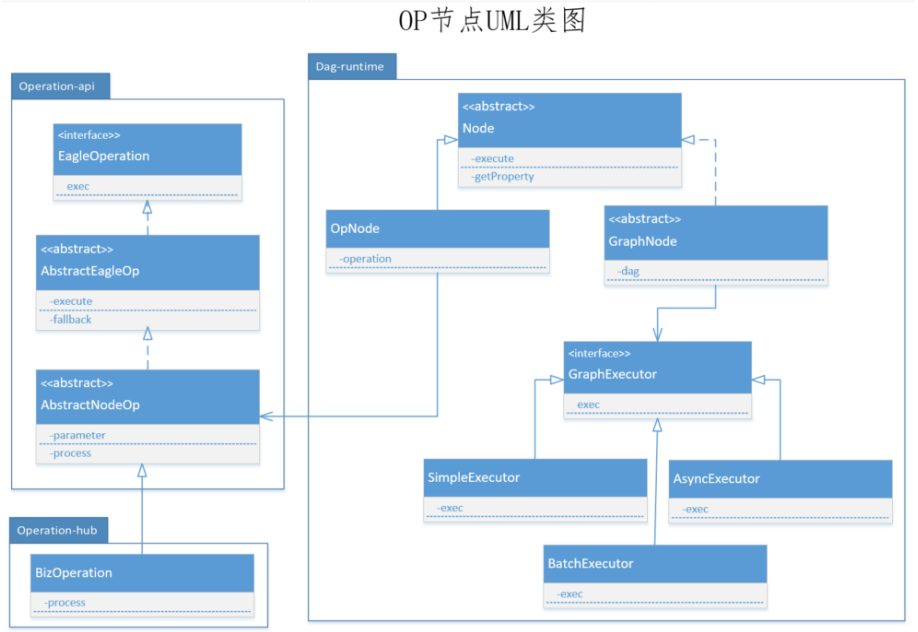
平台集成了数据引擎 matrix，为在线策略提供线上数据访问能力。matrix 负责管理数据上线流程和提供统一的类 sql 查询访问，对应用屏蔽底层存储，大大降低了策略开发成本，开发人员能够专注于业务实现。

3.2 OP-Lib：面向运行时的统一代码库

在敏捷开发模式下，开发人员更加关注的是每次迭代需求的快速交付，往往容易忽略新增代码与现有代码的统一风格规范，代码冗余度，整体运行效率等问题。容易出现不同功能代码间的强耦合，编码相互入侵，上下文运行环境不兼容等一系列问题。这些问题带来的最直接的后果就是代码维护成本不断提高，线上运行效率的不断下降，同时大量的补丁代码也增加了线上 BUG 出现的概率，因此我们需要提供一套标准的策略算子统一规范，在满足业务功能的前提下将异常的影响控制在最小的范围内。

3.2.1 OP-API：功能完善的 OP 接口

为了使策略逻辑代码和开发平台解耦，我们设计了一套完整的 OP 接口规范，通过这套接口的隔离，用户只需定义 OP 的输入输出并实现对应的 OP 接口，而请求上下文信息，参数验证，异常异常处理都由框架自动处理。



- EagleOperations 是 OP 算子的统一接口，作为 OP 与调用方的交互规范，为了增加 OP 的通用性，OP 接口只有一个 exec 方法
- AbtractEagleOp 封装了对 OP 算子的通用处理逻辑，包括参数验证，上下文解析，运行时监控埋点以及异常处理
- AbtractNodeOp 是 OP 接口对策略执行引擎的扩展，主要面向执行引擎的 DAG 节点实现，在 OP 接口的基础上扩展了节点相关参数

3.2.2 OP-Lib：开放的策略 OP 代码库

OP-Lib 是所有策略业务实现的代码库，在 OP 接口的规范下，策略的实现可交由各个业务开发来完成，同时在发布代码时上报该策略 OP 的元数据信息，如逻辑说明，输入输出参数类型等。框架将在服务启动时以预加载的形式实例化这些 OP 算子，并在请求进入时根据 DAG 执行计划统一实现调度策略 OP 开发流程遵循：OP 设计与实现、单元测试、（框架）集成测试、 Snapshot 包、Release 包。

3.2.2.1 OP 定义与实现

策略 OP 作为 DAG 的执行图节点，包含图节点基本要素，同时作为逻辑单元，为提高策略复用性、策略上线效率，通过配置化驱动。同时为避免参数爆炸、策略实验等问题，引入“参数组”概念，OP 代码提供了逻辑模板，配合参数组完成具体业务策略。

i. 元数据信息定义

开发 OP 前，首先定义 OP 元数据信息，将定义的 OP 元数据信息通过配置文件的形式保存，用于后续上报。

ii. 代码实现

框架提供了一套完整的 OP 开发接口，屏蔽了底层图执行相关实现细节，同时提供了异常、超时等机制，用户只需关心 OP 的内部的逻辑功能实现。

策略 OP 可继承的 OP 基类可分为两类：

1）NodeOp0<I,O,P>

表示接收多个同类型上游 OP 的输出结果集合作为入参，上游 OP 的数量不限制，并且接受的上游的输出是无序的。

I：输入类型，O：输出类型，P：参数类型

2）NodeOpN<I1,I2,....,IN,O,P>

表示接收 N 个不同类型上游 OP 的输出结果作为入参，N 在这里是泛指多个，具体基类包括：NodeOp1，NodeOp2，NodeOp3.....等，实际执行时上游 OP 的数量要同该 OP 定义的数量保持一致，且有序。

I1，I2，....，IN：输入类型，O：输出类型，P：参数类型。

基类提供 3 个方法：

1）Opout<O> process()：具体的功能逻辑

2）Opout<O> fallback()：失败回调方法，用于异常处理，降级处理。当上游节点抛出异常、或者自身逻辑抛出异常时，会执行该方法，方法的返回即节点的输出。

3）Function<JsonNode,P> paramFn()：定义策略参数解析器，用于将策略平台定义的 json 参数结构转化为 OP 定义的参数实体。

3.2.2.2 调试与监控

1）在线 debug 调试

为方便用户在线调试、排障，我们提供了 debug 模式，可以在线查看整个图中各节点的输出的结果，验证各节点结果的正确性，快速定位问题。

在构建 DAG 图的执行请求时，通过设置 debug 参数为 true 来开启 debug 模式。

同时考虑到某些节点的输出可能是函数、延迟计算等不可直接查看的对象，我们提供了 SyncOpOutput 接口，实现 OP 输出的自定义转换，同时不影响线上正常输出。

2）自动化测试

OP 上线前，我们要保证对线上现有的策略和性能上无影响，可借助自动化测试完成。

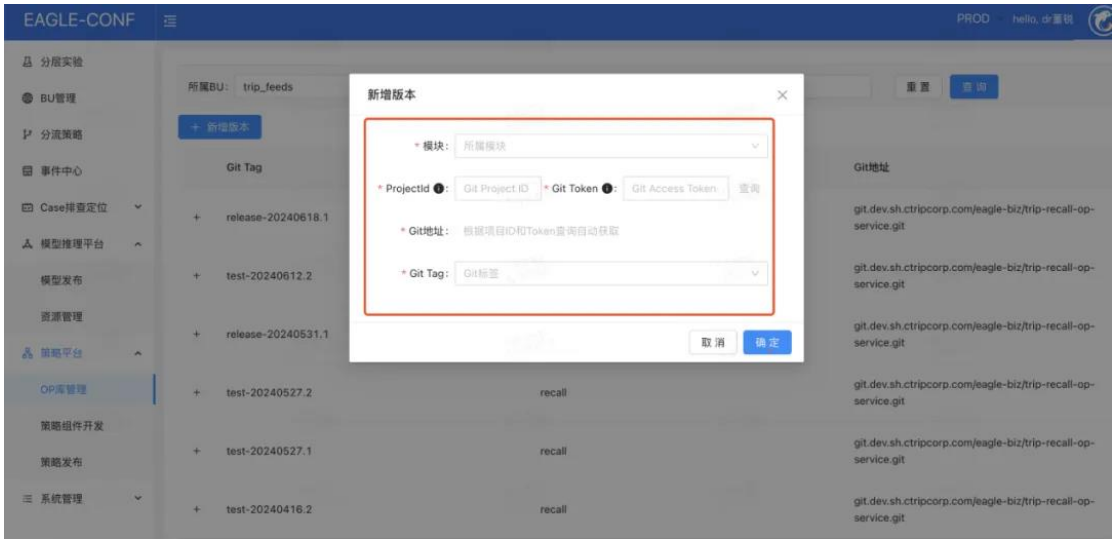
目前平台已集成自动化测试功能，支持多种测试需求，包括：功能测试、回归测试、结果一致性测试、性能测试。上线前需要将测试包/分支上传至测试平台，同时设置样本用例、测试规则等参数即可开始自动化测试任务。

3）监控埋点

平台提供了丰富全面的监控埋点（流量监控，耗时监控，异常监控等等），帮助用户观测 OP 在线运行情况。

3.2.2.3 部署与注册

最后需要将包含 OP 元数据信息的配置文件跟随代码一起发布到代码仓库，发布一个正式的代码仓库包，然后在策略平台的 OP 管理库中升级版本，将刚创建的包注册进去，完成 OP 上线。

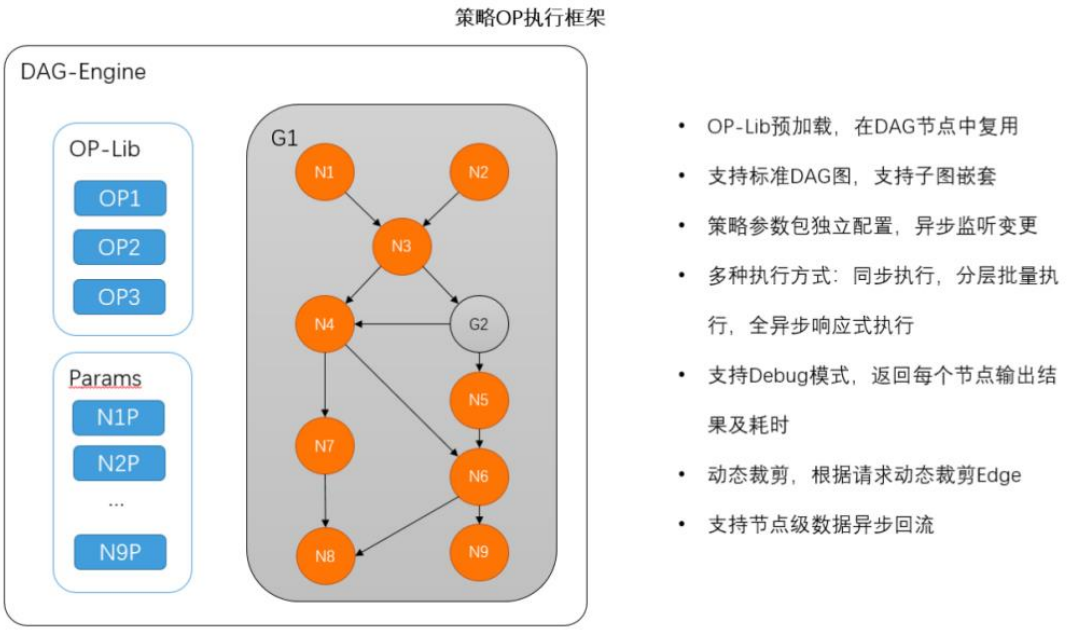


3.3 DAG 执行器：高效的通用策略执行引擎

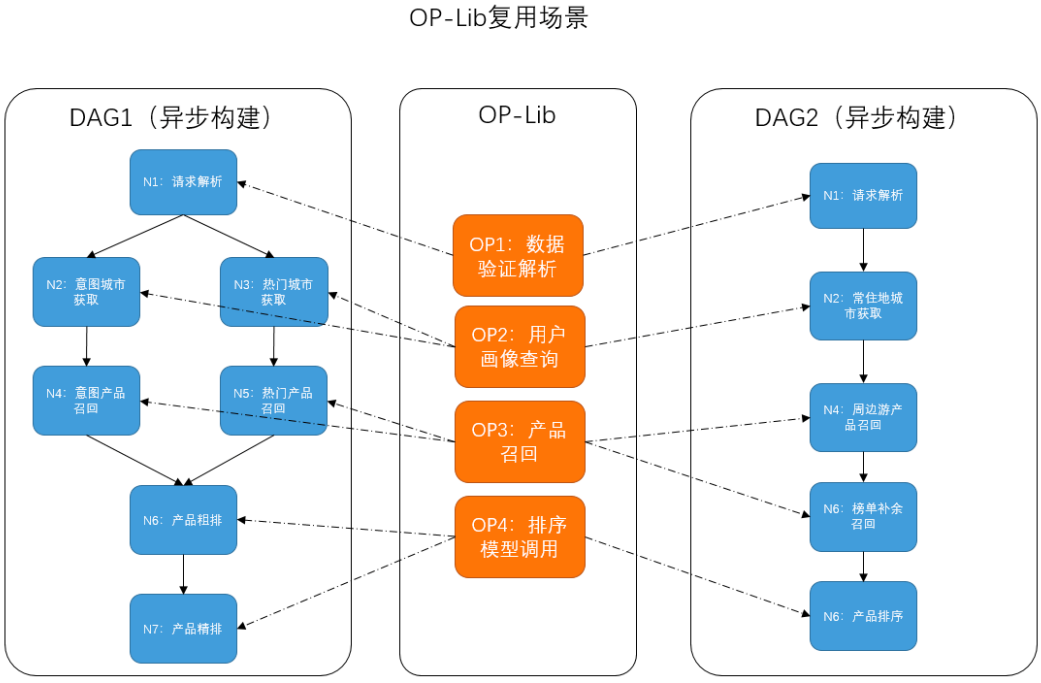
策略的执行引擎是一款基于有向无环图（DAG）的数据结构实现的策略节点调度框架，为了满足和覆盖搜广推全场景策略的编排，在引擎的设计上充分突出灵活易扩展的特点，既可以快速应用于现有的策略场景中，又可以实现快速扩展附加功能。该执行引擎具有如下特点：

- 标准的 DAG 规范：框架接收所有符合 DAG 规范的执行计划。具有高通用性
- OP 复用，为了增加 OP 的复用性，降低代码冗余，框架从设计上将 OP 实例作为单例模式预加载，并可以挂载到多个相同功能的节点上，通过执行节点参数来执行不同的逻辑分支
- DAG 嵌套：支持在 DAG 中将节点配置成另一个 DAG 子图，形成 DAG 嵌套结构。该功能在复杂的 DAG 中可避免节点平铺，具有更好的可读性，同时隔离资源及异常
- 动态图裁剪：支持根据请求动态裁剪 DAG 的边对象，满足节点维度的 ABtest 实验
- 可扩展的图优化器：框架监听到图配置变更后默认启动责任链模式的图优化器，用户根据应用场景扩展需要的节点优化规则

OP 策略执行框架结构设计示例图



OP 算子复用示例图



3.3.1 应对复杂场景的执行引擎多种实现

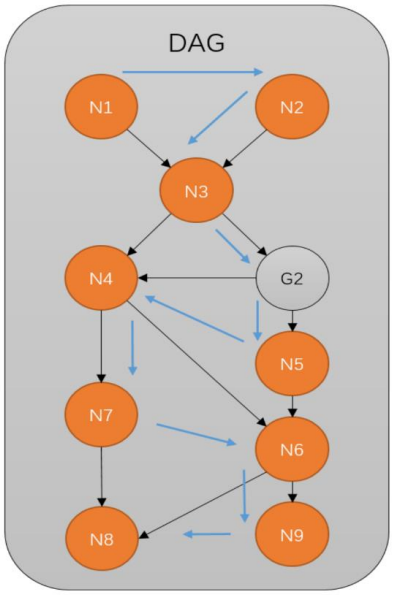
由于框架设计目标是可以执行任何标准的 DAG 图，为了保证各种简单或复杂的 DAG 能以最高的效率运行，我们在执行器引擎设计上采用了统一接口的多种运行策略实现，并支持动态切换，目前已实现了三种执行引擎：

3.3.1.1 广度优先串行调度

在广度优先(BFS)的访问策略下，执行器会以入度为 0 的顶点作为入口，直接使用请求线程以串行方式依次调用当前层的所有节点，如果同一层访问完毕，再调用下一层，直到所有节点访问完毕。该策略在执行周期内完全使用请求线程，没有线程切换开销，占用资源较少。但所有节点串行执行会增加执行器整体耗时，比较适用于对响应时间要求不高或无状态的纯计算场景。

3.3.1.2 广度优先并行调度

此调度实现同样使用 BFS 的算法访问节点，并绑定一个线程池资源，调度时会将同一层的节点打包提交到线程池并行执行，对于并行度较大的 DAG，此策略可以有效提升 DAG 的执行效率。



广度优先同步调度：N1-N2-N3-G2-N5-N4-N7-N6-N9-N8

广度优先并行调度：[N1,N2]-[N3]-[G2]-[N5,N4]-[N7,N6]-[N9,N8]

3.3.1.3 全异步响应式调度

虽然并行调度的方式可以并发执行 DAG 中平行的节点，但仍然存在“短板效应”,由于提交批次间仍然是串行的，如果某一批次中的节点 N 执行耗时较大，则该批次所有节点都会阻塞等待，尽管下一批次中大部分节点不依赖节点 N。为解决此短板，我们使用异步阻塞队列实现一个全异步响应式执行器，执行过程如下：

主线程逻辑：

1）主线程（请求线程）进入执行器，创建一个阻塞任务队列 Q。并找出 DAG 的所有根节

点（入度为 0 的顶点）作为任务队列的初始任务；

2）主线程进入事件队列开始调度：主线程尝试从任务队列等待可执行的节点，如果获取到节点则将节点提交给异步线程（子线程）执行（如果等待超时，则退出）；

3）如事件队列还存有未执行事件则重复 2）操作，如当前请求所有事件（任务）全部执行完成，则获取叶子节点的返回结果并退出 DAG，执行后续的流程。

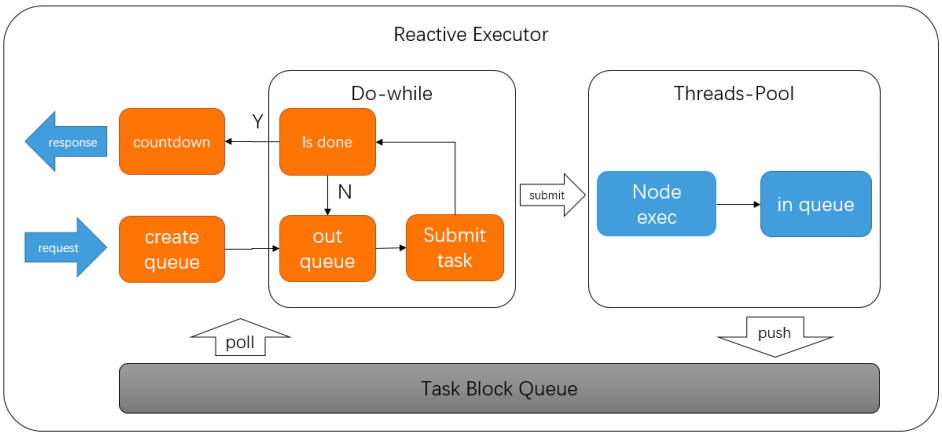
子线程逻辑：

1）子线程接收到主线程提交的任务后开始执行，并将返回值存入临时缓存区。

2）子线程在完成当前节点任务后，从 DAG 中找出当前节点所有可执行的下级节点集合，压入任务队列，（触发主线程步骤 2）

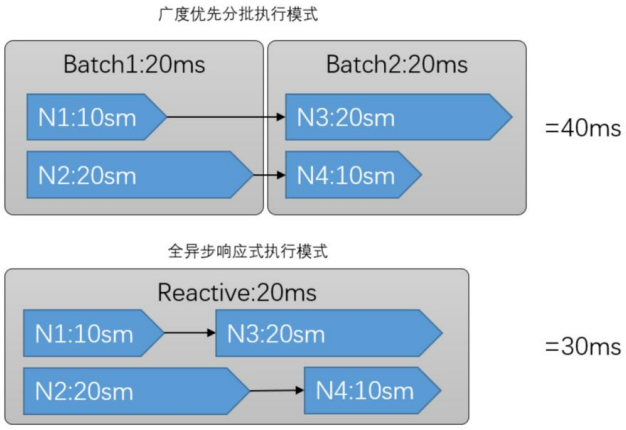
3）子线程标记后被回收（如是虚拟线程则直接释放）

异步响应式图执行器设计



异步响应式调度的优势在于，在复杂 DAG 下，每个节点只要满足了前置条件（入度全部完成）即可触发运行，不受同层平行节点的耗时干扰，进一步提升了执行器运行效率。同时该策略下由于每个节点都是异步运行，在传统的平台线程池模式下，高并发意味着占用更多的线程资源，需要合理设置线程池规模和资源降级策略，另外框架支持在 java21 环境下将线程池配置为虚拟线程模式，该模式下节点将使用无池化的虚拟线程异步执行，关于虚拟线程相关概念可通过 JDK 官网了解

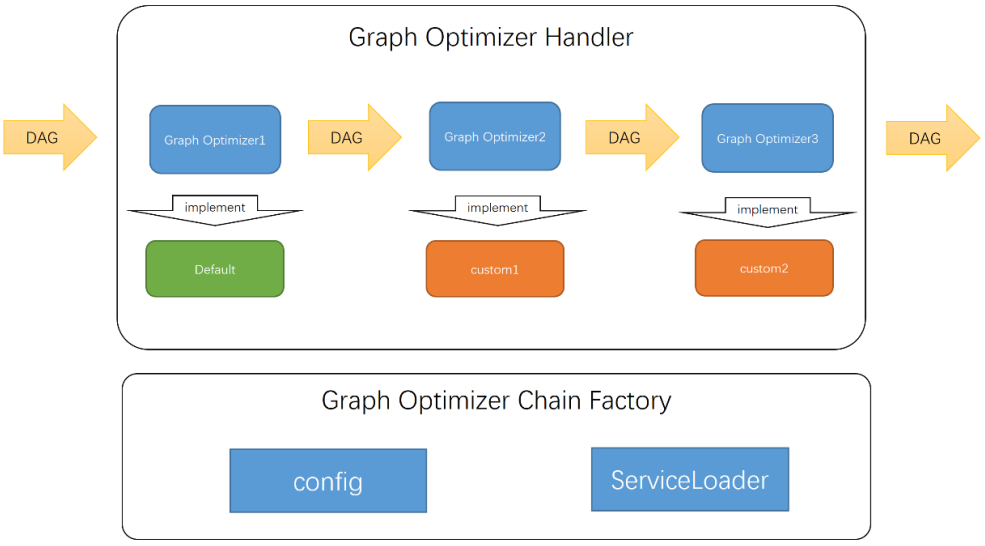
下图对比了分层调用和响应式调用的区别，可见响应式调用可明显减少调用时间：



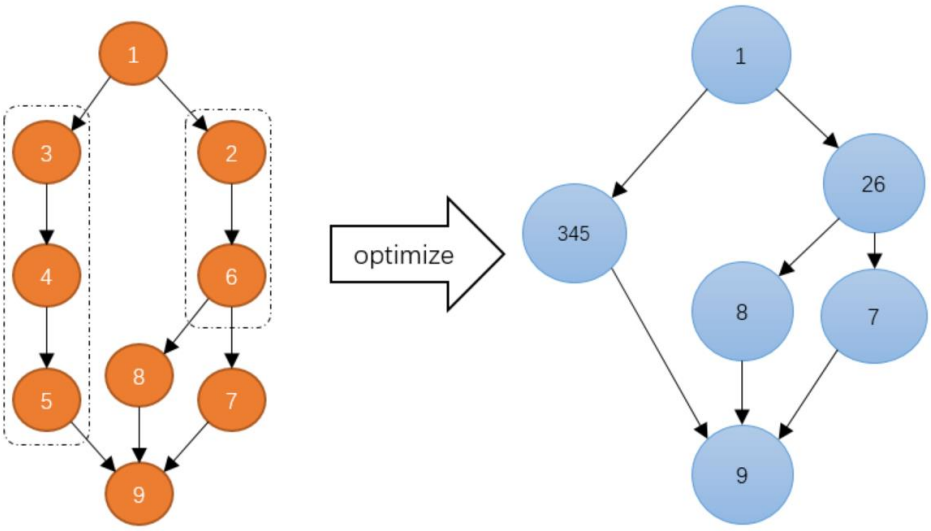
3.3.2 执行效率提升：可扩展的 DAG 图优化器

在策略开发平台的策略编排（DAG 构图）完全由用户自定义，这意味着推送到执行引擎中的 DAG 规模可能非常大，如果用户同时使用的是异步执行器，节点的并行规模可能会耗尽线程资源，因此，我们引入了一个异步图优化流程，执行引擎监听到 DAG 图变更后，会将此图的节点按照需要的规则进行合并，最终生成实际的物理执行计划；从设计上看，优化器采用了责任链的开发模式，除框架预设的优化器外，用户可以扩展实现自定义规则的优化器并设置顺序。合理的执行顺序能有效提高优化器的执行效率。

图：基于责任链的图优化器



目前框架预设的优化器为串行节点优化器，可以将关系唯一的两个相邻节点合并为一个包装节点，减少执行器的调度次数和线程资源。

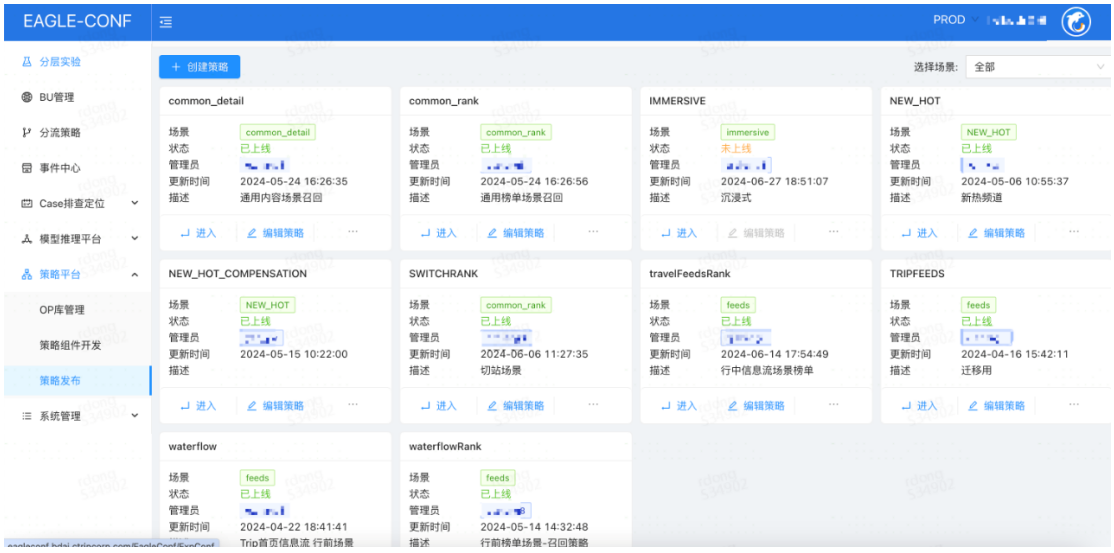


在实际案例中证明，用户的 DAG 图越复杂，默认优化器的优化效率越高：

2024-05-30 18:05:21.414	GraphOptimizer[SerialOptimizer],Graph[SMARTTRIP] optimized Vertexes:[44 -> 12], Edges:[50 -> 18]	qconfig-worker-remote#-thread-3
2024-05-30 18:05:21.413	GraphOptimizer[SerialOptimizer],Graph[TEST] optimized Vertexes:[70 -> 20], Edges:[83 -> 33]	qconfig-worker-remote#-thread-3
2024-05-30 18:05:21.410	GraphOptimizer[SerialOptimizer],Graph[IMMERSIVE] optimized Vertexes:[136 -> 42], Edges:[167 -> 73]	qconfig-worker-remote#-thread-3
2024-05-30 18:05:21.407	GraphOptimizer[SerialOptimizer],Graph[MARKET] optimized Vertexes:[29 -> 10], Edges:[34 -> 15]	qconfig-worker-remote#-thread-3
2024-05-30 18:05:21.376	GraphOptimizer[SerialOptimizer],Graph[CTRIPFEEDS] optimized Vertexes:[805 -> 357], Edges:[1062 -> 614]	qconfig-worker-remote#-thread-3

3.4 策略平台在推荐信息流中的实践

目前整个推荐策略已完成策略 OP 化改造工作，策略 OP 涵盖了召回和重排等多个阶段，已有 80 多个推荐信息流业务场景接入了策略平台，涵盖了多条业务线。



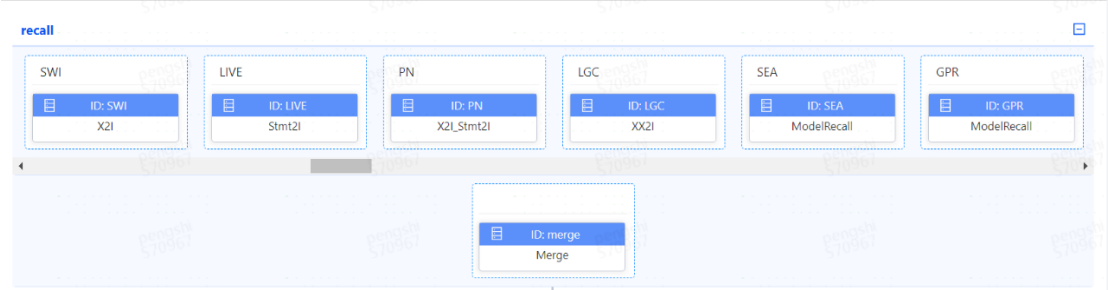
3.4.1 策略上线效率提升

以信息流召回阶段的策略上线为例，在原有开发模式下，需要重新定义一个新的通道，涉及代码逻辑定制，要经历开发、测试、发布的一个完整上线流程，至少两天时间。接入策略平台之后，完善的策略 OP 池基本覆盖了召回各种需求策略，一个新通道上线，只需添加配置，测试验收即可，简单的策略 1 小时内可以上线，复杂的策略半天时间，极大的提升了策略上线效率。

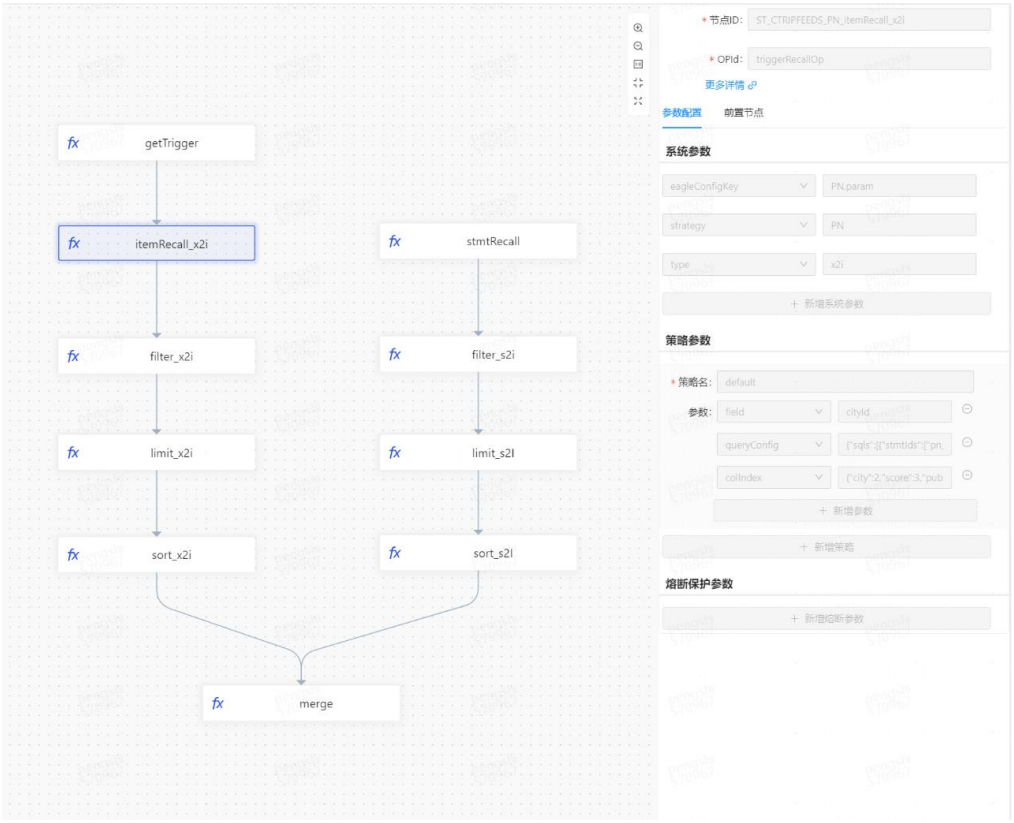
3.4.2 策略逻辑透明

策略逻辑通过 DAG 图化展现，根据每个节点所选择的 OP 以及定义的参数组能够快速了解该节点内部逻辑，策略整体逻辑清晰明了，大大降低了学习成本低，沟通成本。

下图为首页召回阶段平台视角，可以清楚看到线上生效的策略（SWI，LIVE，PN，LGC，SEA，GPR 等），以及各策略使用的召回组件模板，可大致了解通道逻辑，比如 SEA 使用的是模型召回组件（ModelRecall），通过模型来召回产品；SWI 使用的是 trigger 召回组件（X2I），通过指定条件召回产品。



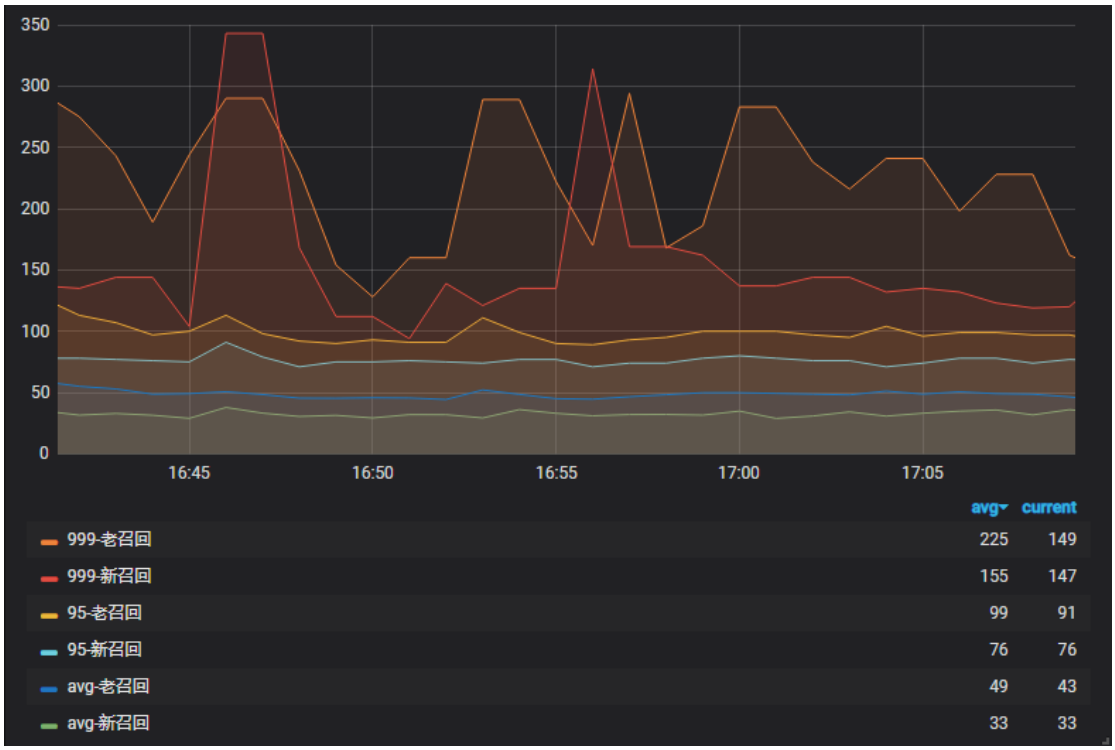
点击通道可查看该通道具体的策略逻辑，下图是 PN 通道的策略逻辑，可以看出该通道由两路查询策略合并组成。



3.4.3 性能提升，资源优化

首页推荐信息流召回服务接入策略平台后，服务性能得到提升，资源利用率得到优化。

性能相关指标如下：



整体耗时提升 30%左右，性能得到明显提升。

资源相关指标如下：

下图左边为老版召回应用核数变化，初始核数为 2400，右边为新版应用核数变化，流量完全切至新版后核数为 900，CPU 核数从 2400 缩减至 900，缩减比例 60%。

同时策略平台底层框架引入了 java21 虚拟线程技术，对于召回这种重 IO 的业务场景，资源得到进一步优化。下图为新版召回服务使用虚拟线程前后资源对比，因虚拟线程切换，CPU 核数从 900 缩减至 600，缩减比例 30%，CPU 利用率由 20%提高至 40% ，提升 100%。

经过上述两次优化，召回服务的 CPU 总核数从 2400 缩减至 600，缩减比例 75%，资源得到优化。

3.5 未来规划

3.5.1 平台建设

平台未来会持续建设、优化，主要目标包括：

- 继续优化图执行引擎，提升图执行效率；
- OP 性能优化和组件沉淀：优化 OP 性能，沉淀业务组件、提升新场景构建效率；
- 全链路 Debug：为全流程提供统一的 Debug，帮助用户快速定位链路问题，提高问题解决效率；
- 平台易用性：为提升用户体验，进一步提升易用性；

3.5.2 平台推广

平台专为搜广推业务领域设计，目前首页推荐业务场景已接入策略平台，未来将继续推动和支持更多业务线搜广推业务场景接入。

携程度假商品千亿日志系统架构演进

【作者简介】cd，携程资深后端开发工程师，度假商品系统研发，专注于后端系统性能提升。

在携程旅游度假的线路类商品系统中，由于商品结构复杂，涉及底层数据表上千张，在日常供应商以及业务维护过程中，每日产生 6 亿+的数据变动记录。这些数据的变动留痕，不但可供录入方查看，也对日常产研的排障起着至关重要的作用，同时也可以提供给 BI 做数据进一步分析。商品日志系统建设尤为重要，随着商品日志系统不断发展迭代，已经积累达到 1700 亿条日志。

本文将介绍线路商品日志系统的演进过程以及在其中遇到的问题。

一、发展轨迹

线路商品日志系统的发展大致可以分为以下三个阶段：

2019 年以前：单表日志

在 2019 年以前，商品系统尚无统一的日志系统来记录商品的变更，在系统中使用 DB 日志表，该表以非结构化的方式记录商品基本信息的变动。

2020 年~2022 年：平台化

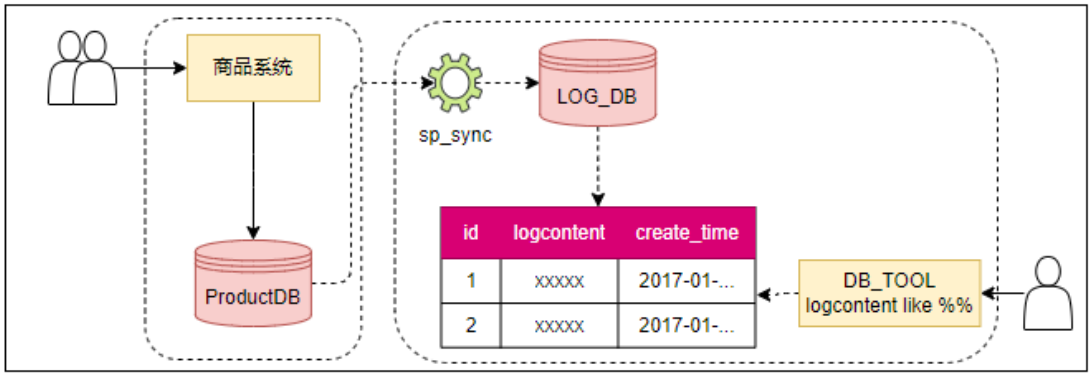
在系统改造过程中，建立统一的商品系统日志平台，通过配置的方式记录商品的数据变动日志，覆盖商品维护的全部流程。

2023 年~2024 年：开放

经过在线路商品系统的实践，商品系统日志平台经历百亿级数据的考验，并以灵活的配置方式记录数据变动日志，同时支持自定义索引字段，具有接入和使用成本较低的优势。为此，通过对商品系统日志进行改造，逐步向门票、用车等业务线开放使用。

二、演进过程

2.1 V1.0 DB 单表存储



图：单表日志

在 2019 年以前，记录线路商品的变动日志较为简单，在 DB 中建立一张日志表（id,LogContent）来记录日志，数据变动以非结构化的文本记录在 LogContent 字段内，且仅覆盖商品最基本的信息，在使用时通过数据库查询工具执行 sql 语句 like 关键字进行查询，这种方式带来的问题也显而易见：

数据量大，性能低

由于是单表文本字段存储，导致表的数量非常巨大，达到单表 10 亿+（370GB）的数据，查询超时问题严重，不得不进行定期归档。

可读性差，仅开发人员使用

由于日志内容以文本字段存储，在进行日志查询时，一般由开发人员使用 like 语句直接查询 DB，例如：select id, LogContent from log where LogContent like '%1234%'。查询速度缓慢，严重影响日常排障流程。

扩展性差

由于日志写入与业务代码强耦合，且采用非结构化存储。对于新增日志，需要对业务代码进行改动，在接入时存在一定的成本，且接入后无法直接提供给供应商或业务人员直接使用，最终仍需要开发人员进行查询转换。

2.2 V2.0 平台化

2.2.1 技术选型

针对 V1.0 遇到的问题，重点在于海量日志数据的存储与查询，业内解决海量日志数据存储与查询的方案一般有以下几个：

ES+Hbase

HBase 提供高并发的随机写和支持实时查询，是构建在 HDFS 基础上的 NoSql 数据库，适用于海量日志数据的存储，可支持到 PB 级别的数据存储。但其查询能力有所欠缺，支

持 RowKey 快速查询，若有复杂查询则需要自建索引。ES 提供强大的搜索能力，支持各种复杂的查询条件，适合快速检索及灵活查询的场景。ES + HBase 的组合，利用各个组件的优势，结合起来解决海量日志数据存储及查询的问题，但架构较为复杂，需要保证两个组件间数据的一致性。

MongoDB

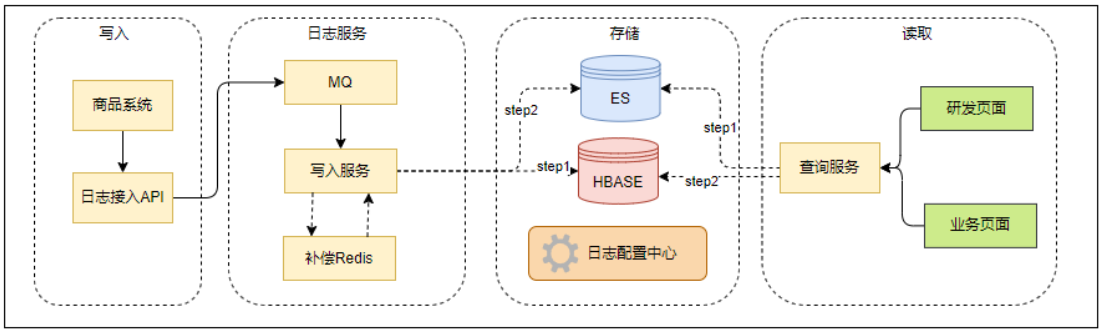
支持多种查询，具有文档型及嵌套的数据结构，但其支持的数据量级一般在 10 亿级别，对比 HBase 要欠缺得多。如果想要处理 TB 级以上的数据量，需要进行适当的架构设计和优化，例如利用分片集群来水平扩展数据等，付出的成本会比较高。

ClickHouse

Clickhouse 是一个开源的列式数据库，采用列存储的数据组织方式，具有高性能、可伸缩性、容错性和低延迟查询等特点。查询性能出色，可实现秒级甚至毫秒级的查询性能，对于数据压缩和存储效率高，可节省成本。适用于海量数据的存储及查询场景。

通过对以上方案进行对比，我们的数据量级已经超过 MongoDB 一般的处理能力，因此该方案被淘汰。对比 ES + HBase 与 ClickHouse，这两个方案都比较适合海量日志的存储与查询，但是受限于内部成本控制，CK 集群的日志保存时长被控制在一定天数内，无法满足我们业务场景的需求。最终，我们选择 ES + HBase 的方案。

2.2.2 整体架构



图：整体架构

基本原理即利用 HBase 解决存储问题，利用 ES 解决搜索问题，并将 ES 的 DocID 与 HBase 的 RowKey 关联起来。通过发挥各个组件的优势，相互结合解决海量日志的存储与查询问题。如上图所示，在接入日志 API 后，所有日志均经过 MQ 进行异步处理，如此既能够将日志写入与业务代码的逻辑解耦，又能确保写入速度的平稳，避免高峰流量对整个 ES + HBase 集群的写入造成压力。

2.2.3 RowKey 设计

RowKey 设计原则：

唯一性：RowKey 应保证每行数据的唯一性；

散列性：数据均匀分布，避免热点数据产生；

顺序性：可以提高查询性能；

简洁性：减少存储空间及提高查询性能；

可读性：以便人工查询及理解；

对于线路商品日志，对于直接可读性要求不高，查询的场景我们是从 ES 中先查出 RowKey，再用 RowKey 去 hbase 查询日志原文，整个过程 RowKey 是人工不可见的，结合我们实际的场景，线路商品数据日志的 RowKey 由五部分构成{0}-{1}-{2}-{3}-{4}

{0}：传入的 pk 转换为 md5[pk]值 16 进制字符串，取前 8 为

{1}：tableId 补 0 至 8 位

{2}：pk+4 位随机值补 0 至 24 位

{3}：log 类型补 0 至 16 位

{4}：时间戳

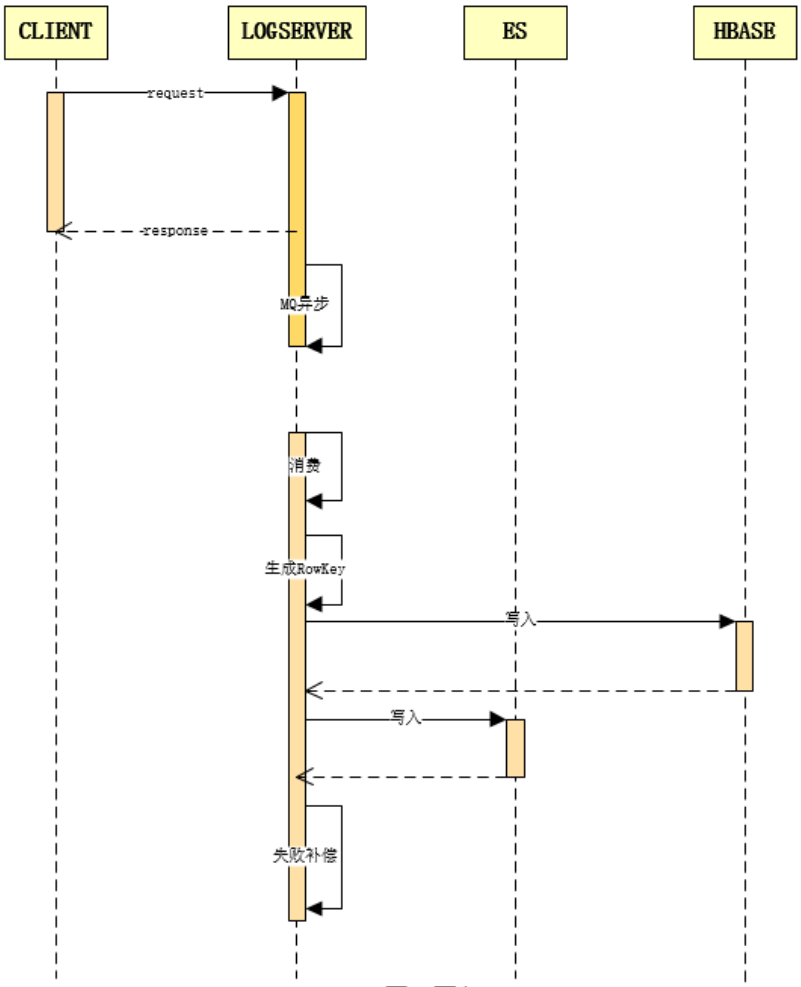
RowKey					Column Family:tpl				
					logtime	logcontent	logtype	...	...
md5[pk]	tableid	pk	logtype	timestamp	169331...	{"pk":2.....	3	...	...
md5[pk]	tableid	pk	logtype	timestamp	169331...	{"pkg":1.....	4	...	...

图：RowKey

2.2.4 扩展

对于线路商品信息的维护分散于不同的模块中，例如录入模块、直连模块等。鉴于此，我们抽象出统一的数据写入服务，并提供统一的日志接入 API，API 内部异步写入日志。在底层的数据写入服务中，将所有的写入操作接入日志 API。通过这个方式，将扩展性统一到日志配置中心。

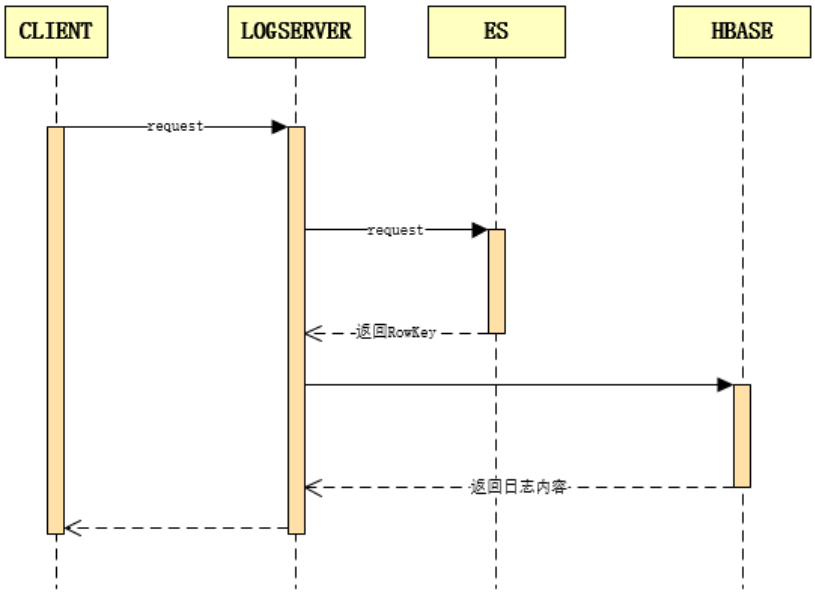
写入流程



图：写入

日志的写入流程如上图所示，客户端调用日志 API 以进行数据变动日志的写入操作。日志服务在接收到请求后，将其抛入 MQ，由后续的消费组进行消费处理。消费组件在接收到消息后，会进行相应的消费处理，并根据上述的 RowKey 生成策略为该条日志生成 RowKey，随后将日志文本内容写入 HBase，在写入成功之后，再将索引数据写入到 ES。其中，若 HBase 或 ES 中的任何一个写入失败，都会将此条日志写入补偿 redis 集群，再由补偿逻辑进行后续补偿，以确保整个日志的写入成功。

查询流程



图：读取

日志的查询流程如上图所示：客户端调用查询 API 并传入查询参数，日志服务接收到请求参数后，将其转换为 ES 分页查询请求，从 ES 集群中查出 RowKey，再汇总 RowKey 并从 HBase 中批量查出日志全文内容。

此外，我们利用上述查询 API，建立一个日志查询页面，供研发人员使用。在该页面，相关开发人员可以便捷地进行数据变动日志的查询。上述日志平台的建立，相对完美地解决线路商品海量数据变动日志的存储及查询问题。同时，抽象日志的配置中心，解决一定的扩展性问题。

整个系统的优点在于：基于表级别日志的商品日志记录，覆盖全面，配置灵活，索引结构化存储，支持海量日志数据的存储及查询。

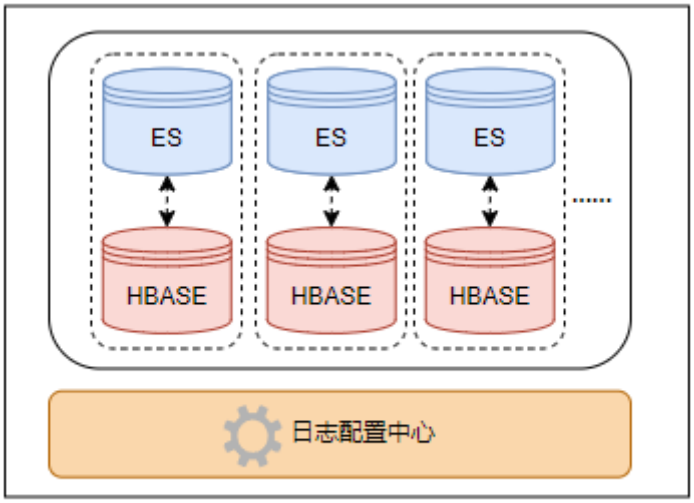
缺点是：对于使用方而言存在一定的局限性，过于“技术化”，开发人员使用较为方便，但供应商与业务人员使用困难。

2.3 V3.0 赋能

2.3.1 业务赋能

存储能力

随着日志写入量的增加，日志查询效率逐渐下降，对 ES 和 HBase 的拆分势在必行。如下图所示，我们对 ES 和 HBase 进行横向的拆分与扩容，并在日志配置中心制定匹配规则，根据接入日志类型的不同，将其匹配到不同的集群进行写入。此外，对于接入方，我们也支持独立集群申请，使用方可以根据自身情况决定是使用独立的集群部署，还是使用公用集群。

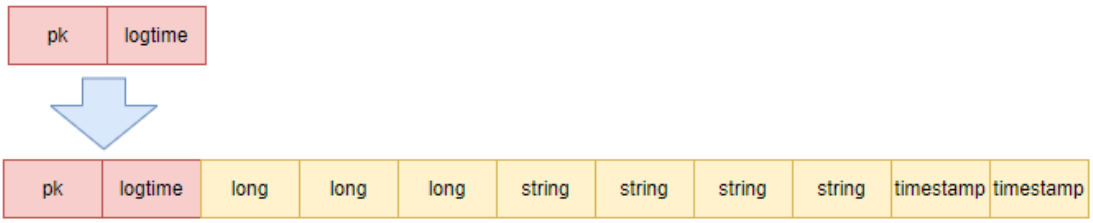


图：存储多集群部署

搜索能力

搜索能力的提升主要由以下两个部分：

索引字段扩展：支持的索引字段更多。前期我们绝大部分场景的日志的索引条件是产品 id 或者资源 ID，随着接入的日志变多，索引字段也变的丰富起来。对于日志搜索场景我们进行梳理，预留 10 个可支持不同查询的索引字段（其中四个数值型、4 个字符型，2 个日期型）供使用方使用，覆盖绝大多数的查询场景。



图：索引字段

ES 索引分区：随着接入的日志增多，单个索引文件也愈发庞大，直接影响日志的查询性能。一般而言，对于日志类型数据，常见的方案是依据时间建立索引，该方案的优势如下：

1）提升查询性能。若日志携带时间范围进行查询，则可仅搜索特定时间段的索引，避免全量索引的查询开销。

2）便于数据管理。可以按照时间删除旧的索引，从而节省存储空间。

通过对日志进行分类，主要包括商品信息、开关班、价格库存等模块。随后结合业务使用场景、每天产生的增量数据以及服务器资源进行评估，最终决定按周建立索引，且索引数据保留一年。

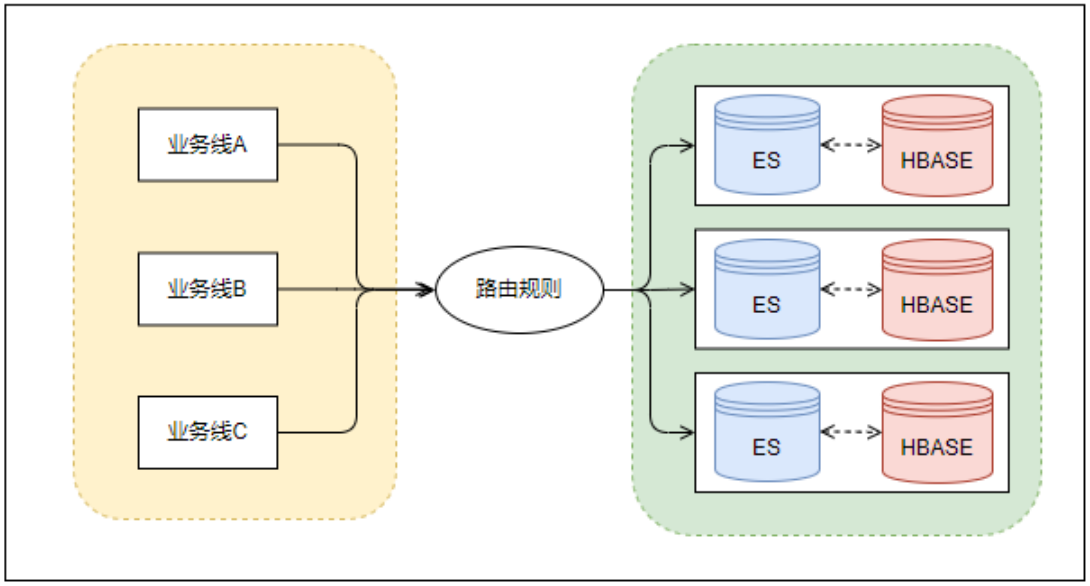
1）利用定时任务在每周一时创建下一周的索引；

2）利用定时任务每周删除已过期的索引；

时间	ESclusterA--indices				ESclusterB--indices	
2024第一周	plog20241	mlog20241	dlog20241	...	tlog20241	...
2024第二周	plog20242	mlog20242	dlog20242	...	tlog20242	...
2024第三周	plog20243	mlog20243	dlog20243	...	tlog20243	...
2024第四周	plog20244	mlog20244	dlog20244	...	tlog20244	...
...	...	...	...	...	...	...

图：indices

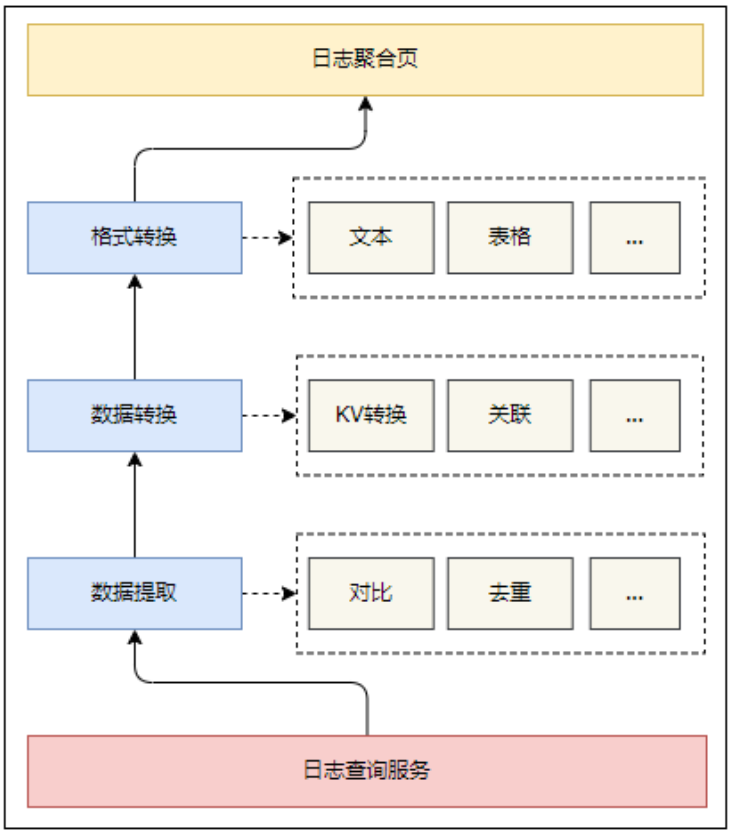
基于以上存储能力与搜索能力的扩展提升之后，我们在日志配置中心定制了【业务线<-->日志集群】的路由规则，来决定接入的其它业务线日志最终存储的日志集群，提供了更加灵活与具有弹性的业务线接入能力。



图：路由

2.3.2 供应商赋能

展示能力



图：展示

在 V2.0 版本中，日志页面仅限于研发人员使用，底层数据过于技术化，业务与供应商难以理解。通常情况下，如果能将日志的查询前置到供应商及业务环节，将极大地减少研发人员平时工作中的排障时间。

为此，我们提供 B 端的日志查询页面，给供应商及业务人员平时排查问题使用。我们对日志内容进行格式化的转换处理，将其转换为供应商和业务人员能够理解的信息，包括行转列、新旧对比、KV 转换、关联数据查询等。日志内容不再是抽象的文本，而是展示为与平时使用的商品系统相对应的内容。这对业务和供应商更加友好。

扩展能力

基于上一步展示能力的提升，对于新接入的日志如何能快速为业务及供应商提供 B 端的页面进行查看，这一步将大幅节省开发排查时间。对底层日志数据需要转换为业务及供应商能看懂的信息的场景进行分析，并总结 7 种数据展示的方式，分别如下：

1）文本字段类：此种展示内容最为简易，无需进行转换，用户可直接理解日志记录的内容，前端展示的即为此内容。

图：直接展示

2）数据关联类：此类日志内容中记录的是一个 id，但实际内容存在于另一个关联表的数据中，例如 id：1 表示的是跟团游，不能将 id：1 的日志展示给用户，而需转换为“跟团游”，这就需要进行一步关联 db 表的查询转换。

图：关联查询

3）枚举类：此类日志内容记录的是一个 key 值，实际用户能理解的是该 key 值所代表的含义，例如产品钻级：0，就需要转换为：“不分级”，这就需要关联枚举值的配置文件进行查询转换。

图：枚举转换

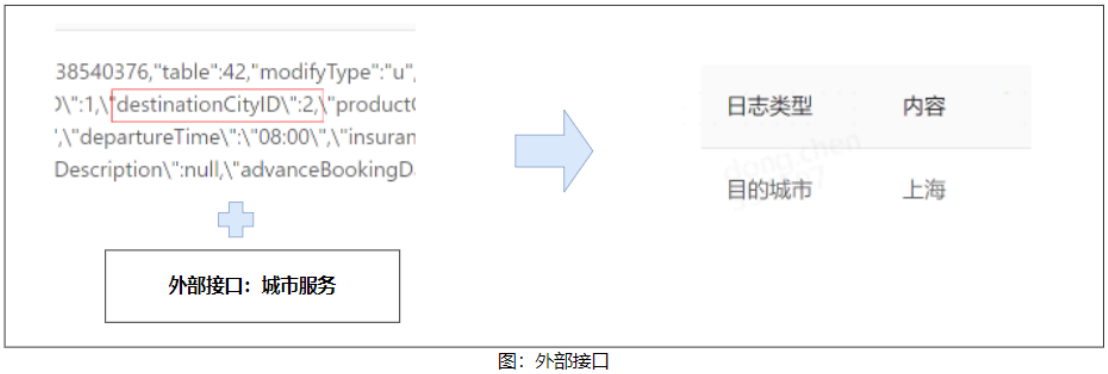
4）位存储类：此类日志内容记录的是一种计算后的结果数值。例如支付方式是通过按位与计算然后累加的结果。这种情况就需要按照一定的计算方式将其还原回去。

图：位存储

5）字段组合：此类日志记录的是分散的数据，但实际需要将数据结合在一起查看才会更具业务意义，例如资源适用人档，日志中分为最大、最小记录。实际展示时需要结合到一起展示范围。

图：组合

6）外部接口：此类日志记录的是一种依赖外部接口的值，例如日志记录的是城市 id：2，代表的是上海，这就需要调用外部接口将 2 转换为上海。



图：外部接口

7）差异对比类：此类日志需对结果进行解析以作对比，从而使用户能够更为直观地理解。通常存在两种情形：其一，日志内容所记录的即为两份对比数据，此种情况仅需依循规则予以解析即可；其二，若日志数据属于当次的快照数据，则需与前一次快照数据进行对比，以找出差异。最终达成如下图所示的展示效果。

图：对比

针对以上这些日志解析的场景，我们最终构建一个日志转换配置。对于新加入的日志，在绝大多数场景下，我们只需修改底层的数据提取及转换配置，便可较为快速的配置出日志查询页面，提供给供应商使用。

三、结语

本文详细介绍度假商品日志平台的演进历程，以及在各个阶段遇到的问题及解决方案。在整个演进过程中，针对海量日志数据存储与搜索的技术挑战，我们采取一系列措施，实现千亿级数据查询在 500ms 内的响应。

同时将日志系统开放，将问题查询解决前置到供应商及业务人员，极大降低一些数据变动查询需求的复杂度，减轻研发及 TS 同事重复性的工作。此外，我们还对日志平台进行横向的扩容配置，以支持更多的业务线可以接入。截至目前，多个业务线总数据存储量达到千亿级别。

携程国际机票基础数据中台化：构建高效的数据管理和应用平台

【作者简介】空歌白石，携程资深研发经理，关注性能和效率提升、架构优化。

本文概述了携程机票在基础数据处理所做的中台化设计方案。第一部分阐述了中台化的背景和面临的挑战，第二部分介绍了中台化设计的原则以及目标，第三部分详细介绍了中台化架构的关键技术实践，涉及数据一致性、数据时效性、系统健壮性、系统自动化等方面的探索和尝试，最后是整体的技术架构概览以及线上运行效果。希望以上内容能够对大家有所帮助和启发。

一、背景与挑战

随着携程国际机票业务的快速发展与全球化战略的深入推进，需要使用的数据种类以及对应的复杂度也随之显著增加。这一增长趋势不仅带来了数据管理的挑战，同样对数据准确性和实时性提出了更高的要求。

接下来，我们从生产者和消费者两个角度具体看下有哪些挑战。

生产者视角下的挑战：对于基础数据的生产者，每一种数据类型都有其独特的业务逻辑，涵盖数据的获取、处理、存储和匹配等环节。数据更新周期长，业务复杂性高，和基础数据相关的应用就达到了几十个，维护成本不断攀升。同时，还存在着数据访问效率低、上云过程复杂、数据回滚困难以及无法应对大规模机器重启或数据刷新等问题。

消费者视角下的挑战：对于基础数据的消费者，每一个应用都或多或少依赖于基础数据。任何一种数据的误差都可能引发广泛的线上问题，而且问题的发现往往滞后，导致生产问题被不断放大。此外，还存在着业务开发接入复杂、测试环境不稳定、服务启动缓慢、垃圾回收频繁、上云过程繁琐以及数据不一致等问题。

二、原则与目标

面对上述问题，构建一套中台化系统是其中一个解决方案，我们希望实现数据资源的高效整合，消除数据孤岛，提高数据处理效率，确保数据质量标准。在系统设计的初期阶段，我们首先从数据生产者和消费者角度出发确立了一系列核心目标：

数据生产者角度：

数据一致性：确保数据在各个环节保持一致性，避免偏差。

数据时效性：保障数据的实时更新，满足业务对数据的即时性需求。

系统健壮性：构建稳定可靠的系统架构，以应对各种运行环境和负载条件。

- 数据可追溯性：实现数据的全流程追踪，便于问题定位和历史分析。

- 数据可回滚性：提供数据版本控制，允许在出现问题时快速回退至稳定状态。

- 监控完善性：建立全面的监控体系，实时监控数据流和系统状态。

降低成本：通过优化资源配置，降低机器和存储的成本；简化系统维护流程，减少人力和时间的投入。

数据消费者角度：

优化消费流程：简化数据消费流程，提高数据处理的便捷性和效率。

- 接入方式简化：提供直观易用的接入方式，降低数据使用的门槛。

- 统一数据模型：建立统一的数据模型，确保数据的一致性和可理解性。

解决环境问题：解决不同运行环境下的数据同步的问题。

- 测试环境完善：提供完善的测试环境，确保数据的准确性和稳定性。

- 云服务便捷性：优化云服务接入，提高数据服务的灵活性和可扩展性。

提升服务性能：通过技术优化，提升服务的响应速度和处理能力。

- 启动耗时降低：减少服务启动时间，提高系统的快速响应能力。

- 减少 GC 次数：优化内存管理，减少因数据更新引起的垃圾回收（GC）操作，提升系统性能。

三、关键技术实践

在基础数据中台化建设的实践过程中，遇到了一系列的问题，本章节将介绍一些关键的技术实践。

3.1 数据一致性

3.1.1 版本控制

同一集群下不同的机器在更新缓存时由于调度时间不一致，会导致不同机器在同一时间使用的数据不一致，为解决此问题，我们使用数据版本控制策略来解决此问题。每当数据发生变更，我们便认定一个新的数据版本已经诞生。这个新版本可以是包含了变更和未变更数据的完整副本（下文简称为"全量数据"），也可以是仅包含变更内容的更新（以下简称

为"增量数据"）。不论是全量数据还是增量数据，我们都会将数据记录在 BLOB（Binary Large Object）文件中，BLOB 文件将作为数据传输的媒介。

实施数据版本控制后，我们能够收获以下收益：

版本追踪：确保每一次数据更新都有迹可循，并且在新版本出现问题时能够迅速恢复到之前的版本。

数据一致性：保证所有数据消费者能访问到相同版本的数据，从而减少因数据版本不一致而引发的问题。

容错与恢复：能够快速识别出问题数据，并利用多种通知机制，促使产品或开发团队及时介入，解决问题。

性能监控：基于数据版本，构建性能监控体系，以评估数据传输和处理的效率。

数据安全：在数据传输和存储过程中，通过加密和访问控制机制，确保数据安全，防止未授权访问。

3.1.2 去中心化

在业界，如果某个服务想要消费基础数据时，主流解决方案是采用直连数据库或调用应用程序接口的方式，如下图 1 和图 2，这两者均属于 C/S 架构。C/S 架构虽被广泛采用，却面临着如数据库承载压力大、数据一致性难以保障、系统扩展性不足、开发与集成过程复杂、硬件成本高昂、缓存穿透、中心服务的读压力，以及难以应对流量高峰等问题。

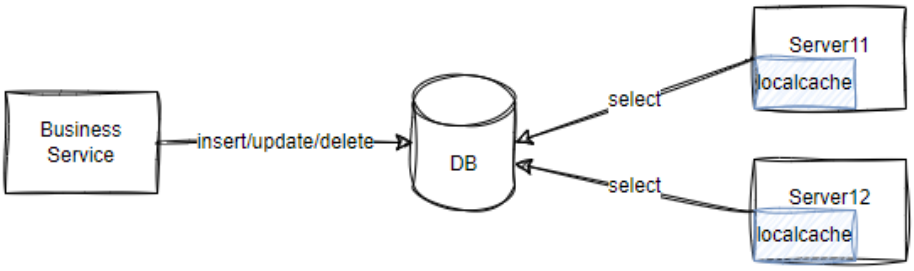


图 1

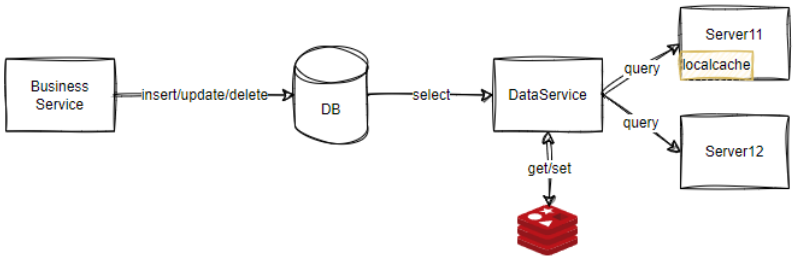


图 2

图 1 为了克服以上问题，我们引入了 P2P 架构（Peer-to-Peer Architecture）。与 C/S 架构相比，P2P 架构实现了去中心化。在 P2P 网络中，每个节点都具备客户端和服务器的双重身份，能够直接与其他节点进行通信和数据交换，无需依赖中央服务器。同时，P2P 架构下的数据查询耗时并不会因为客户端数量的增加而线性增长。图 3 直观地展示了 C/S 架构与 P2P 架构的本质区别。

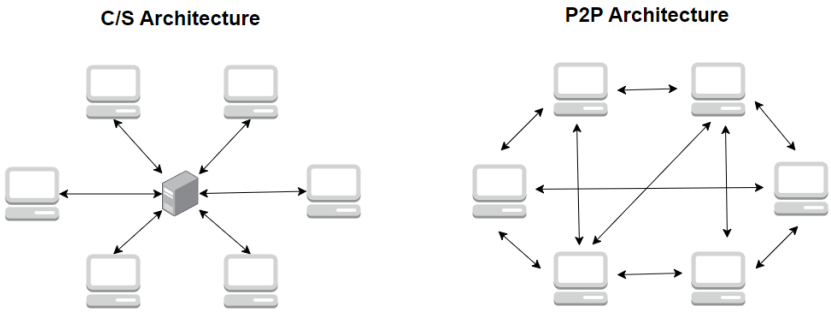


图 3

基于前文提到的 BLOB 文件版本控制机制，我们选择了 BitTorrent 协议作为 P2P 网络中的文件共享与分发标准。BitTorrent 以其高效的数据传输能力，特别适用于大规模数据的快速分发，具体架构如图 4，客户端可以根据实际需求控制是否要加入 Peer 网络，如图 4 中的红色机器就仅仅下载数据，并不分享给其他 peer。

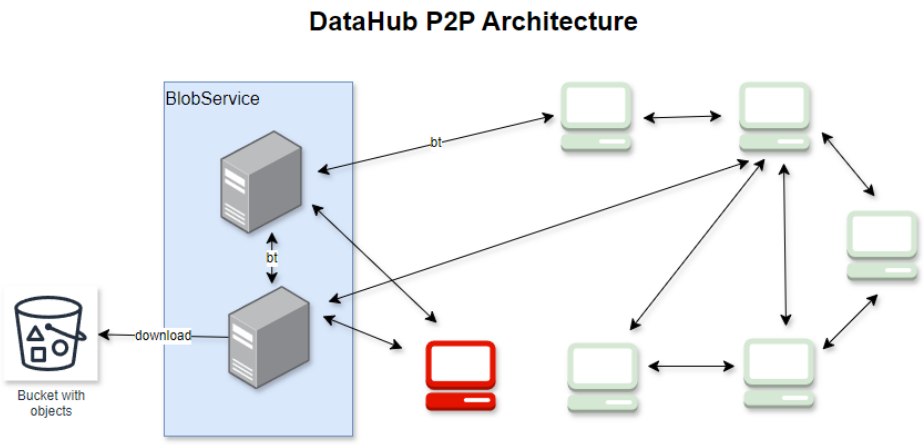


图 4

3.1.3 数据组合策略优化

在业务实践中，我们有时候会需要处理多种基础数据的组合场景，比如想要在消费国家数据时一同消费国家所在大洲数据，当组合中的数据版本出现不一致时，会导致数据不准确。

为应对这一挑战，我们采取的方法是通过业务规则，将所有相关联的组合数据整合到单一的 torrent 文件中，消费方可以一次性下载到所有关联数据。这种整合方式的优势在于，它允许消费方一次性获取整个数据集合，而不需要分别从不同来源或版本中收集和协调数据。通过这种方式，我们不仅简化了数据获取过程，而且更重要的是确保了数据的一致性和准确性。

3.1.4 单点数据生成策略

集群内不同的机器间隔几秒钟查询同一数据库，查询结果便可能会有所不同，为保证消费方数据一致性，我们在生成全量和增量数据时采取了单点数据加载模式，如图 5。仅有单一的机器负责从数据库中查询数据，这种设计一方面显著降低了数据库的读取压力，其效果相当于将压力分散至消费方机器数量的 1/n（n 代表消费方机器的总数），另一方面结合版本控制策略，在同一时间内中台系统中仅仅有一个有效的版本数据可以被使用。

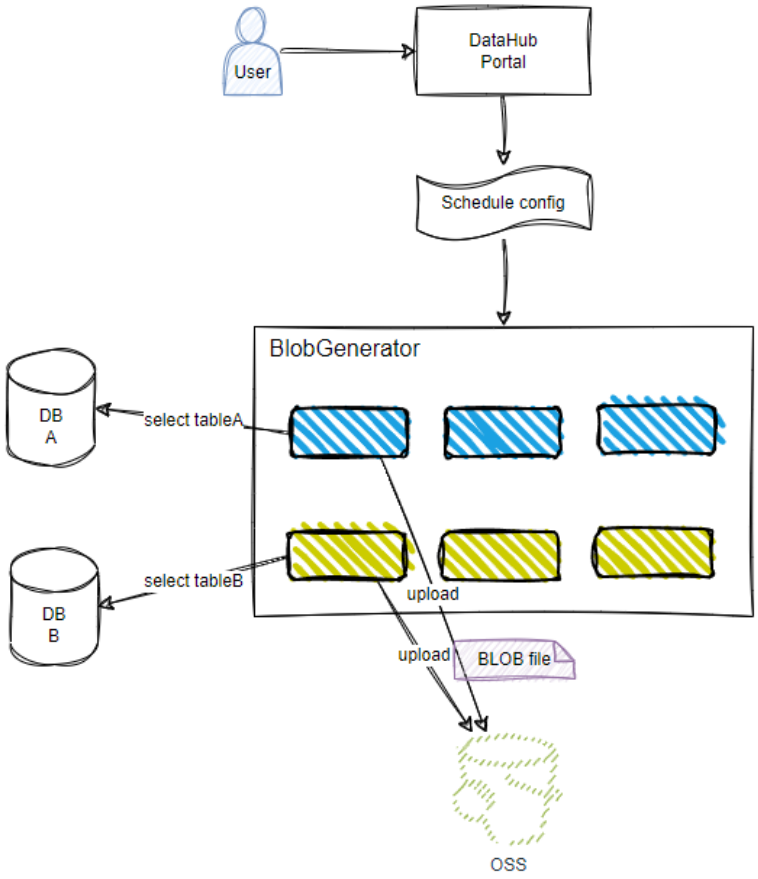


图 5

此外，不同种类的数据更新频率各异，这通常由具体的业务需求所决定。因此，中台系统提供了自定义数据生成配置的功能，这种灵活性的好处在于：

减少不必要的数据加载：通过精确控制数据生成的频率和时机，避免了资源的浪费。

降低客户端缓存刷新压力：减少了因频繁数据更新导致的客户端垃圾回收（GC）操作，从而提高了客户端的性能。

缓解网络带宽压力：通过优化数据生成和传输策略，减轻了整个网络的带宽负担，确保了数据传输的高效性。

3.2 数据时效性

3.2.1 推拉机制

为确保数据生产至消费的全流程时效性，防止数据更新滞后对业务造成损失，我们在架构设计中引入了推拉接合模式。前序系统完成数据处理后，即通过消息中间件向后续系统发出通知，这一连贯流程在各子系统中依次触发，直至数据被消费。

为提高流程可靠性，我们引入了可配置的基于定时任务中间件的拉状态逻辑，确保了任何环节的延迟或异常都能被及时补偿。

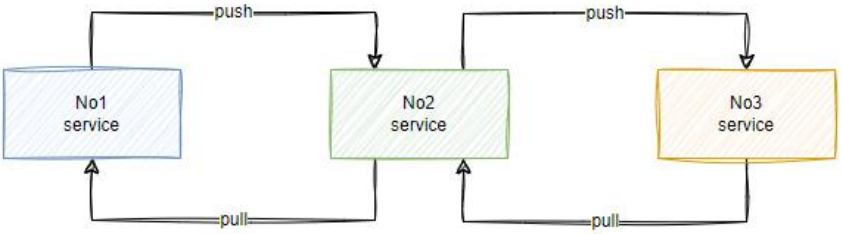


图 6

3.2.2 数据云端迁移

随着携程国际业务的不断拓展，我们的系统架构正逐步向混合云模式转型。传统的基础数据上云是复制数据库到云端，虽然可行，但往往会带来成本上升和数据复制延迟等问题。为应对这些挑战，我们将 Blob 文件分发到不同的 Region，避免了对昂贵数据库实例的依赖，在保证数据时效性的前提下成本降低 98%以上，榆次同时使用此架构消费方在上云过程中可以做到无缝迁移。具体架构如图 7。

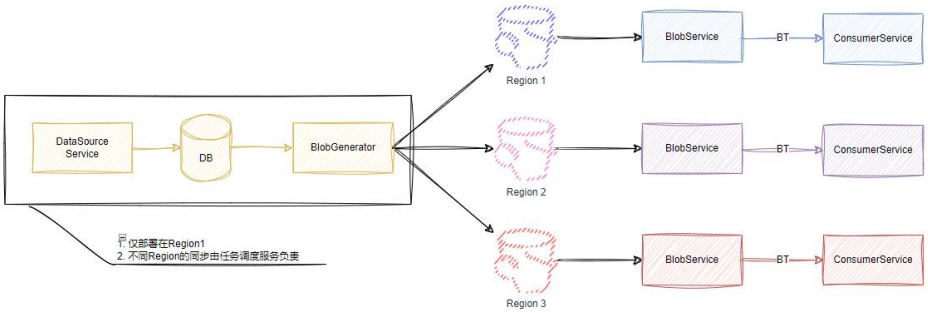


图 7

3.3 系统健壮性

3.3.1 数据校验和拦截

在数据的生命周期中，无论是业务维护的数据，还是由外部数据提供商提供的数据，数据错误总是一个不可忽视的问题。这些错误表现为非法字符、数据缺失、数据重复等。为有效解决此问题，我们开发了一套数据校验机制，针对数据的每个字段执行严格的合规性检查。系统一旦监测到异常数据，将立即通过 TripPal（携程自研的 IM 系统）、电子邮件、短信等多种通信渠道，向数据负责人发出警报，这样可以迅速响应并处理问题。

在问题未解决前，系统会自动暂停出问题数据的更新，防止错误数据的扩散。此外，为了进一步提升数据校验的准确性，我们结合了统计学算法和人工智能（AI）预测模型，对数据变化进行分析和智能判断。具体的架构如图 8。

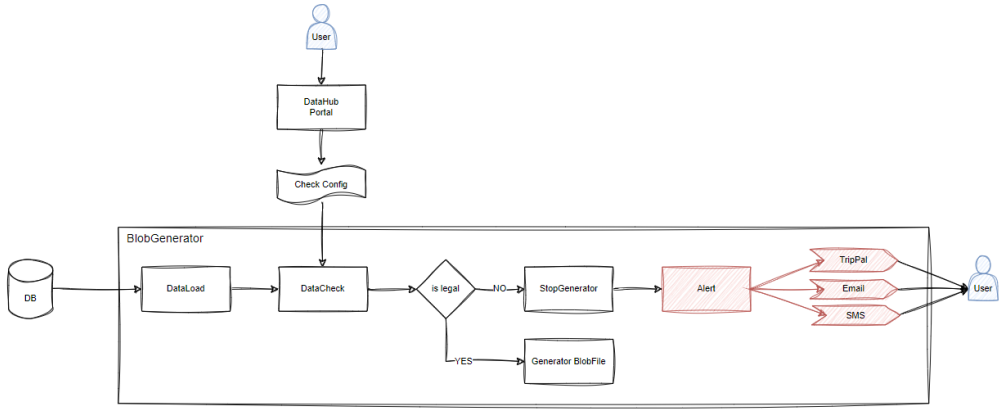


图 8

3.3.2 数据回滚

在生产环境中，面对突发的系统故障，实施回滚操作是最迅速且有效的应对策略。中台系统为此提供了一套数据回滚功能，将回滚版本的数据视为一个完全正常的版本，通过中台的 Portal 界面，用户可以依据时间戳追溯并查询到所需的历史数据版本。其中，整个回滚过程无需对数据库进行任何数据层面的修改，这一点与依赖于二进制日志（binlog）的回滚方法相比，提高了效率和安全性。如图 9。

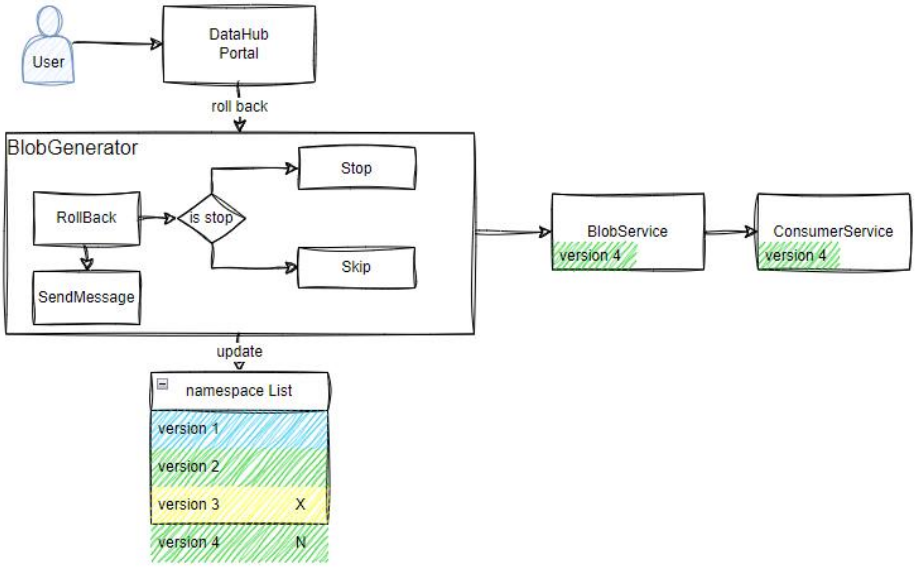


图 9

3.4 消费流程优化

3.4 1 统一数据模型

数据模型会出现新增、修改和删除字段的场景，通常我们会通过编码的方式实现，不仅过程繁琐，而且随着时间推移，会显著增加系统的维护成本。为了降低消费方消费数据的复杂度，需要支持任意数据模型的自动化生产和消费。我们通过脚本化手段实现自动化构建（build）和部署（deploy）jar 包，从而简化流程。具体的流程如图 10。

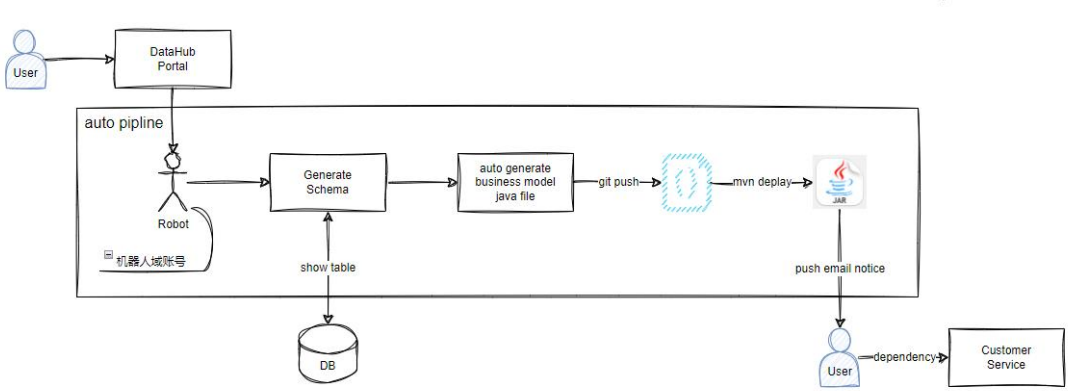


图 10

3.4.2 简化接入方式

当需要消费某个数据时，引入通过图 10 流程生成的独立 model 包，通过以下代码即可获取全量数据集或者按条件查询的数据集。

```
// 引入客户端
@DataResource
private CityClient cityClient;
// 全量查询
List<City> list = cityClient.queryList();
// 条件查询
List<City> list = cityClient.queryList(cityCode);
```

3.5 统一数据治理

在实际的工作中，开发团队和产品团队常常面临一个共同的挑战：如何在种类繁多的基础数据中找到目前生产上实际使用的基础数据？一般会通过口口相传或维护共享文档的方式来解决，但是这种方法不仅效率低，而且容易因人为失误或信息更新不及时导致生产问题，为此，数据中台实现了一个统一的模型入口，简化了数据和模型的搜索和引用过程。用户可以在 Portal 中按照数据库名、数据表名、接口名等检索条件轻松搜索所需的数据类型，如果存在，可以直接引用；如果不存在，可以根据具体的业务需求新增。

对于新的业务接入中台，我们也进行了流程的优化。优化后，接入中台只需调整数据源，无需进行额外的开发工作。这一改进显著减少了业务接入的复杂性和工作量，最小化了接

入流程。与直接连接数据库或调用应用程序接口的方式相比，中台系统在数据接入效率上实现了 90%以上的提升。

四、技术架构概览

在前文所述的基础上，本节从宏观的系统架构视角，阐述携程国际机票基础数据中台化建设的关键技术实现。我们的系统精心设计为若干个互相协作的核心模块，每个模块承担着特定的职责，共同构成了数据处理和分发的中台化平台。

1）DataSource 模块：作为数据流的起点，此模块负责数据的初始写入和确保数据一致性。它通过定时任务或消息通知机制触发数据操作流程。

2）BlobGenerator 模块：专注于数据的生产过程，提供全面的服务，包括数据校验、BLOB 文件生成、版本控制以及回滚操作的拦截。

3）BlobService 模块：作为数据分发的核心，处理来自 DataClient 的数据请求，充当 BlobGenerator 与 DataClient 之间的桥梁，确保数据流畅、高效地传递。

4）DataClient 模块：负责数据的消费端，提供包括 BitTorrent 下载、缓存管理、以及支持精确查询等多种功能，满足不同场景下的数据使用需求。

5）DataQuery 模块：为那些无法通过 BitTorrent 下载方式获取数据的消费者提供了 API 查询接口，支持全量数据输出、条件筛选输出以及逻辑计算等高级功能。

6）Dispatcher 模块：作为系统的调度协调中心，确保 DataSource、BlobGenerator、BlobService 等模块的任务有序执行，保障整个数据处理流程的顺畅和同步。

通过这些模块的紧密协作，我们的技术架构不仅提升了数据处理的效率和准确性，还增强了系统的可扩展性和可维护性。

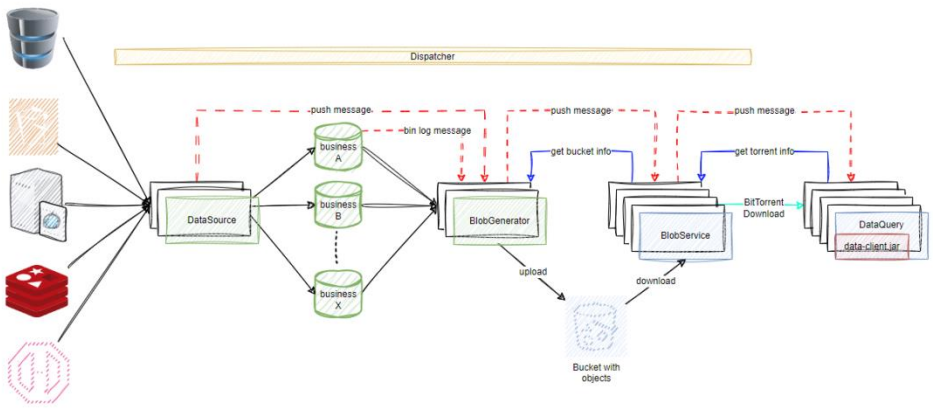


图 11

五、成效

数据中台实现了从数据生成到消费的全生命周期覆盖。系统化的数据治理举措提升了数据系统在治理、成本控制、安全性和运营效率方面的性能。

数据生产者角度：我们对分散的业务流程进行了梳理和优化，根据优先级分批整合至中台。在这一过程中，我们不仅重构了业务流程，还通过重写和优化，挖掘并解决了多个之前未被发现的问题。数据分发的效率实现了质的飞跃，平均分发时间降至 23 秒，对于小规模数据，我们更是实现了 5 秒内的端到端快速传输，极大提升了数据的实时性和新鲜度；整体服务器成本降低了 95%以上；系统的维护成本降低 66%。

数据消费者角度：新数据源的接入效率提升了 90%，上云过程无需进行特殊改造，加快了研发进度，提高了开发效率。解决不同运行环境下的数据同步的问题，减少了对生产环境的依赖。优化了调度策略，减少了 98%以上的无效调度任务，降低了 GC 的频率。

六、未来计划

国际机票数据中台在上线体现了一些在研发效能、数据治理、性能优化、业务提升、降低成本等方面优势，未来我们计划从以下几个关键方面对数据中台进行深入迭代和优化：

1）自动化：进一步提升数据处理流程的自动化程度，减少人工干预，提高整体效率，特别会利用大语言模型等技术进行数据校验以及，增强数据准确性。

2）稳定性：加强系统的稳定性，确保数据中台在高并发和大数据量处理场景下的可靠性。

3）健壮性：构建更加健壮的系统架构，提高系统对异常情况的容错能力和自我恢复能力。

4）时效性：优化数据更新和分发机制，确保数据的实时性和时效性。

5）可视化：通过可视化技术，直观展示数据流动和处理过程，提高数据可读性和易用性。

- 更友好的 Portal 界面：设计和实现更加人性化的 Portal 界面，提升用户体验，简化用户操作。

- 处理流程可视化：实现数据处理流程的可视化展示，使用户能够清晰地追踪数据处理的每个环节。

运维

携程 IT 桌面全链路工具研发运营实践

【作者简介】
Spring，携程软件技术专家，专注于 IT 自动化工具及服务效率提升系统的研发，在公司 IT 架构演进和工具开发方面经验丰富。
Soldier，携程资深软件工程师，专注于自动化工具和服务质量提升系统的研发，在自动化服务工具开发方面经验丰富。

本文概述了携程 IT 管理数万台办公 PC 时面临的挑战及应对方案，介绍了通过全链路工具实现故障主动发现和自动修复的运营理念。详细阐述了背景、系统架构选型及各部件，深入说明了工具实践过程中面对的大数据量、脚本运行权限、交互弹窗等问题及其解决方案，并阐明了在运维故障定位和脚本统计优化方面的支持，希望能为大家带来一些启发。

一、前言

随着企业规模的增长，在满足合规、信息安全的要求下，管理大量电脑并满足多样化的员工需求，同时确保员工能够高效、稳定地工作，已成为一项很具有挑战性的任务。

二、现状

公司拥有大量的员工和电脑，电脑故障量较多，虽有各种自动化工具帮助员工自行解决，但仍有不少故障需要运维工程师人工排障。这种被动式的排障管理模式不仅效率较低、耗时长，而且容易遗漏，影响员工的办公效率。因此需考虑采用一种更加主动、高效的桌面运维模式，如自动化的故障检测和主动修复，以提高员工的办公效率和运维团队的工作效率。

三、全链路研发运营实践

为降低公司电脑故障量，引入主动监测电脑健康度的机制，及时发现电脑存在的已知问题，可以主动自行修复，或提醒用户当前电脑的健康情况，由用户自行处置。对一些特定的故障做到提前预警并提前解决，以保证用户能够持续高效办公。

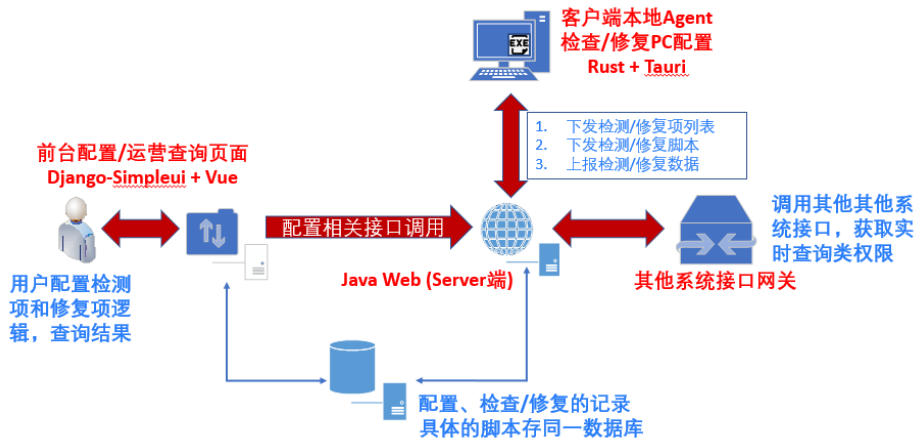
3.1 架构选型

数据采集需开发客户端 Agent 常驻用户 PC，并定期执行，考虑到公司电脑系统的多样性(Windows/Mac/Linux)，为避免多个平台的重复开发而带来的工作量，Agent 选用新兴的内存安全和高性能的 Rust 语言，配合其生态下的跨平台桌面应用程序框架 Tauri。实现具有跨平台能力的 Agent;

Server 端使用 SpringBoot，为所有的 Agent 提供各类接口，完成策略下发、数据收集等;

客户端采集/修复脚本支持 Powershell、Bat、exe 等类型，脚本均可通过全链路 Server 端管理页面单独管理发布，功能脚本和 Agent 独立开，可以最大程度上做到 Agent 和脚本之间的松耦合，增加后续业务的可扩展性，提高运行效率；

管理后台选用 Django、Django-Simpleui、Vue 配合开发，通过配置化的框架，快速构建功能丰富的管理后台，并使用 Vue 对需定制的页面进行完善和补充，提高开发效率；



3.2 前期业务评估和设计

公司电脑体量大，每台电脑数十个检测项，加上一小时采集周期，系统将在短时间内产生数千万条数据，在数据库索引设计、整体架构、多线程高并发编程等方面需做好充分设计。采用多线程、异步队列机制对上报数据实现高效落库，提高整体稳定性和时效性；

全链路数据采集有两种模式：定时客户端采集上报和其他系统接口的实时查询。后者依赖外部接口较多且响应时间相差较大，需要在用户检索查询时使用多线程同步等技巧，保证查询速度和用户体验；

全链路 Agent 和 Server 端交互数据包括数据采集、脚本下载、策略下发，各场景都需要保证其安全性，各接口均采用双向不对称加密对上报的数据和下载的脚本实现加解密，同时通过 Server 端的 Token 实现客户端的会话鉴权机制，保证系统的安全性；

3.3 实战业务流程

1) 客户端

执行调度：客户端用户登录后，Agent 定时在客户端自动调度运行，执行权限为 System；

下载任务：客户端 Agent 向服务端请求，获取客户端所需要检测的相关任务列表（检查项，如：C 盘剩余空间、账号锁定状态等），任务列表中包含检查项 Key、检测脚本名、脚本 MD5 值、脚本运行参数等信息；

采集脚本：客户端 Agent 获取对应任务列表后，首先检查本地缓存中是否存在相同 MD5 的文件，如果存在则直接运行，如果不存在则 Agent 发起请求从服务端下载最新的脚本文件检查校验 MD5 后缓存到本地，脚本类型可为：Powershell、bat、exe 等；


```
{
    "checkKey": "AtrustServerAddress",
    "cmd": "{file}",
    "timeOut": 600,
    "admin": false,
    "md5": "[redacted]",
    "fileName": "AtrustServerAddressFix.bat ",
    "params ": "",
    "reminderVerbal ": "提醒话术 ",
    "reminderVerbalEn ": "Reminder Language ",
    "fixJson ": ""
}
```


采集数据：客户端 Agent 依次调度执行各检查项对应的脚本，采集客户端数据；

数据上报：所有检测项数据采集完成后 Agent 将数据加密后上报到 Server 端，由 Server 端进行解密、数据校验和存储；

修复脚本：Server 端对采集数据结果检查，校验后会将检查结果异常需修复的项，下发到客户端 Agent，返回数据中包含了检查项 Key、修复脚本名、修复脚本 MD5 值、脚本参数、异常处理、超时时间类型等信息，参数类似采集脚本；

异常处理：修复脚本信息中，异常处理类型有“直接修复”、 “提醒修复”、“仅提醒”三种处理方案（处理类型可再后台页面配置）。Agent 依次执行所有“直接修复”相关的脚本，执行后修复结果统一上报 Server 端进行修复结果存储。“提醒修复”、“仅提醒”相关异常信息在桌面右下角弹框提醒，并对“提醒修复”类型的检查项开放用户确认自助修复，用户自助修复的结果同时上报给 Server 端进行数据存储；

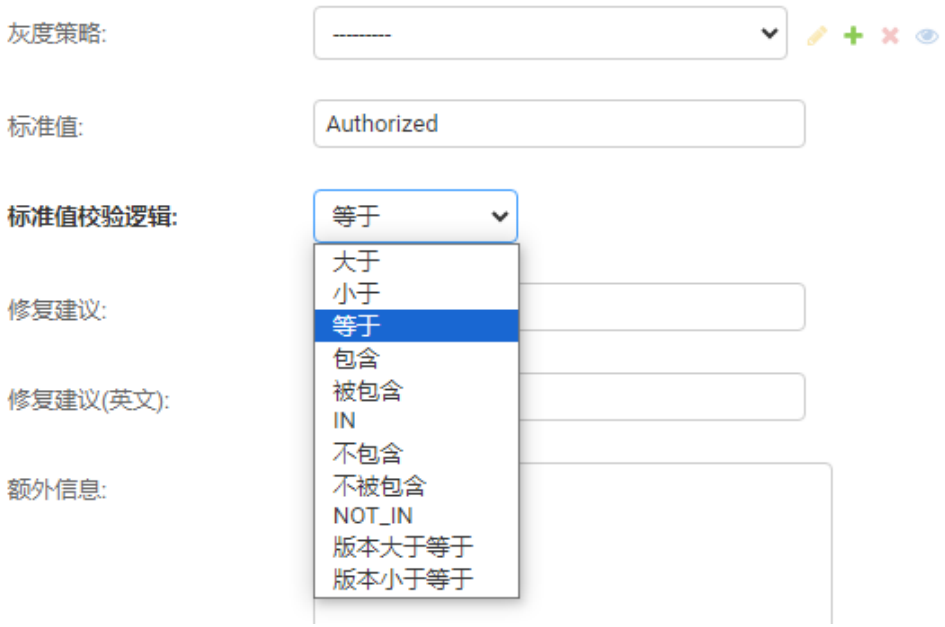
任务结束：数据采集完成后 Agent 退出运行，关闭与服务端的会话，释放系统资源；

2) Server 端

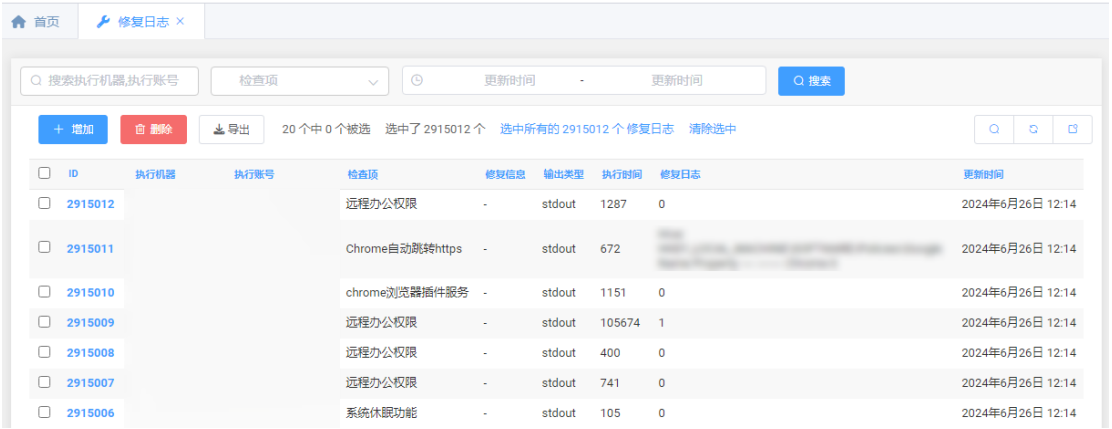
下发采集任务：Server 端缓存所有检查项信息（检测项信息中会包含脚本的 MD5 值，这在下载流程中会用到），缓存保存 2 分钟，当接收到客户端任务查询请求时，优先去缓存中查询，缓存中没有则去数据库查询并更新缓存，即能保证查询的速度，同时减轻数据库的压力；

下发采集脚本：检测脚本&修复脚本均为小文件，以 BLOB 二进制大对象存数据库，同时记录每个脚本的 MD5 值，与检查项一致，脚本信息同时缓存，客户端请求下载文件时，根据请求体中的 MD5 值匹配脚本；

数据校验保存：每个检查项均可独立配置检测结果的标准值和校验逻辑，系统对“标准值校验逻辑”列表中的校验项实现了不同的匹配逻辑，客户端上报检测结果时，Server 端根据检查项配置的校验规则对每条检查数据进行校验，所有采集数据计算出正常/异常后保存数据库，数据采用了异步队列的方式落库存储，提高接口的时效性，减低客户端接口响应时长，同时所有检测异常的数据，Server 端将相关检查项信息（包括修复脚本信息、异常处理类型等）返回给客户端进行进一步的修复工作；



保存修复结果：Server 端接收客户端上报给修复结果，Server 端保存修复结果；



3) 运维管理

全链路最终交付对象是桌面运维团队，运维管理模块必不可少，全链路提供的管理页面包含了检查项、脚本、灰度管理、修复日志、检测结果查询等模块。

检查项管理：工程师可以根据业务情况自定义新增检查项，检查项关键字段包括 CheckKey（检查项唯一标识）、开关（检测项是否可用）、提醒类型（直接修复、提醒修复、提醒）、实时查询（检查项是否实时查询，只有非实时查询类的检查项才会在客户端采集数据）、灰度策略（检测项上线或变更时的灰度功能，保证检查项的平稳上线）、标准值（每个检查项的检测结果标准值或标准值集合）、校验逻辑（包括大于、小于、等于、不包含、被包含等，用于校验数据结果）；

脚本管理：该模块统一处理客户端检测、修复脚本。上传后的脚本会关联检查项 CheckKey 和脚本类型（检测脚本、修复脚本），以关联检查脚本、修复脚本和检查项的关联关系。此模块还关联到具体灰度策略（每个脚本均有独立灰度功能）、操作系统选项（包含 Windows、Mac、Linux，目前只用到 Windows，其它为未来扩展预留）、脚本参数（如 hostname、username 等，用于客户端执行脚本时的固定参数）和超时时间（限制客户端脚本的执行时间，保证采集过程的正常进行）。为确保流程完整性和安全性，脚本上传会启动审批流程，只有审批通过的脚本才能投入生产；

灰度策略管理：用于支持检查项和脚本的生产灰度，灰度维度包括：员工、邮件组、团队、工种、国家、员工批次、电脑名，灰度策略包含黑名单和白名单，黑白名单可以多维度单独配置，并支持黑白名单同时生效，可提高灰度场景的灵活性；

员工批次管理：员工批次是灰度策略中的一种，可以灵活自定义某些没有职级或组织结构关联关系的员工为一个批次，该逻辑可以覆盖补齐所有灰度场景，进一步提升灰度策略的灵活性；

数据查询：运维管理界面提供数据采集结果的查询、筛选，每条结果包机器名、用户、检查项、采集结果、校验结果、上报时间等；

其他：除以上的主要功能以外，全链路运维模块还提供了一些额外的板块，包含提醒配置（用于记录客户端用户选择提醒冷却周期）、权限管理（用于配置和管理每个管理板块的用户权限，提高系统的安全性）等；

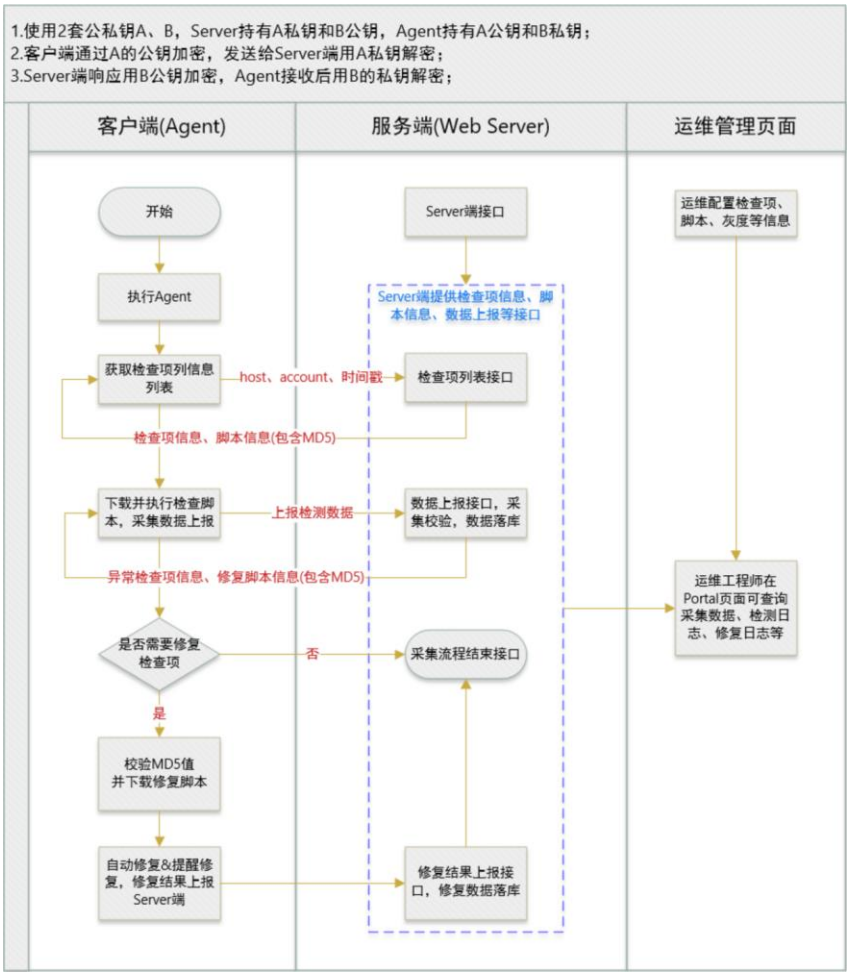
4) 安全措施

客户端 Agent 和 Server 端之间的数据交互采用双向不对称加密方式，客户端和 Server 端分别持有一对公钥和私钥（Agent 拥有 Agent 的私钥+Server 端的公钥，Server 端拥有 Agent 的公钥+Server 端的私钥）。并增加 Server 端 Token 鉴权，实现会话机制，当 Server 端接收到 Agent 请求时，需要验证 Token 的有效性及时效性；

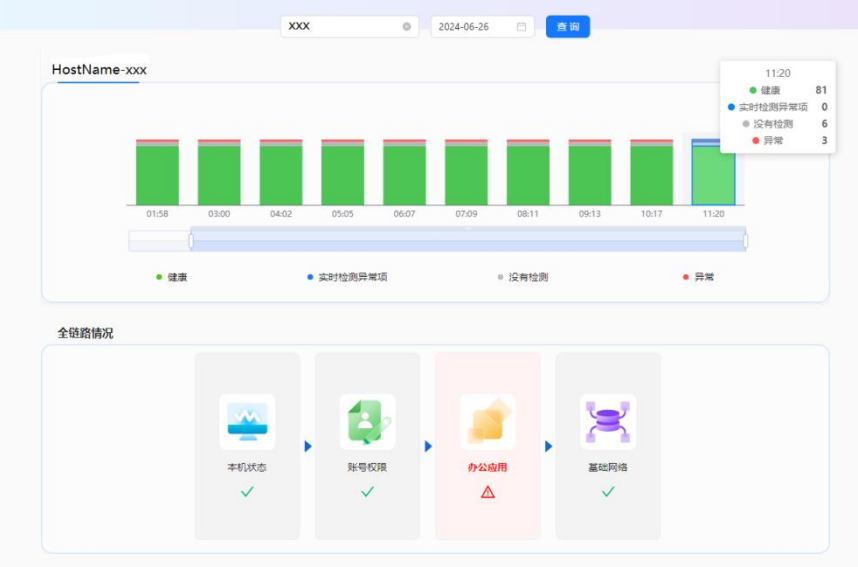
公司办公环境复杂（有内网、独立组网、零信任网络环境等），员工办公电脑可能同时存在于各种网络环境中，为了保证 Agent 能在各种环境内正常工作，Server 端同时开通内、外网域名，Agent 第一个请求首先使用内网域名，无法通信则自动切换外网域名，本次会话中的后续请求均使用确定好的域名。该机制实现了 Agent 在各种网络环境的自动适配，保证了系统的可用性；

每次 Agent 调度执行相关脚本时，都计算本地缓存文件和 Sever 端接口中的 MD5，确保本地缓存文件不会被篡改，避免高权限执行危险命令和程序;

3.4 全链路业务流程图（FLT）



3.5 员工电脑健康度查询



3.6 困难和挑战

1) 采集数据量激增，数据库性能挑战

随着检查项和接入 PC 数量增加，累计的数据量急剧增加（超过 5000W），数据库的性能和系统稳定受到严峻挑战。经过仔细研究后优化，采用数据增量更新策略，同一 PC、用户、检测项的结果，如果与前一次执行结果一致，则只更新数据的采集时间，只有在数据不一致时才新增数据记录。通过额外字段标识记录采集结果是否为当前活跃结果，并配合每个客户端 Agent 执行日志，统计每次 Agent 采集明细，此机制在保证数据完整性的同时，大大减缓了数据量的增长，据统计数据量减少超过 70%;

2) 实时检查项查询逻辑优化

实时检查项的数据均通过外部接口调用获取，工程师在线查询各个检测项的执行结果时后台实时触发接口调用查询，对查询的时效性要求较高，随着实时检查项数量的增加，查询速度逐渐难以满足工程师的要求。采用多个实时检查项并行查询机制，并配合查询结果缓存策略，提高了查询速度。

3) 弹框显示 GUI 交互问题处理

全链路 Agent 以 System 权限调度运行，System 权限启动的应用不能直接与当前用户的 GUI 进行交互，导致用户客户端右下角的弹框提示也无法显示。对 Agent 调度程序进行了优化改造，拆分为 FLT-System.exe 和 FLT-User.exe，分别以 system 权限运行和当前登录用户账号运行，FLT-System.exe 主要负责执行系统级别的检测修复脚本（例如:网络信息检测），FLT-User.exe 负责执行用户级别的检测修复脚本，同时用户弹框界面通过 FLT-User.exe 唤起。

Agent 改造逻辑如下:

FLT-System.exe、FLT-User.exe 为两个程序模块，依次调度执行，同时两者之间通过 RPC 通信实现数据交互；

FLT-User.exe 向 Server 发送请求获取客户端所需要检测的相关任务列表及脚本的下载，脚本运行权限分为 system 权限、user 权限，其中 system 权限的脚本 FLT-User.exe 向 FLT-System.exe 发起 RPC 调度请求，而 user 权限的脚本 FLT-User.exe 会直接调度运行；

FLT-System.exe 启动开启心跳监测，FLT-User.exe 每过 5 秒会向 FLT-System.exe 请求心跳状态，确保程序正常运行，防止系统资源占用。如果不存在心跳，则 FLT-User.exe 自行退出，释放系统资源。同时 FLT-System.exe 如果在连续 3 次都未获得心跳检测请求，也自行退出，释放系统资源；

两个 exe 之间交互采用双向不对称加密方式通信，RPC 调用为 localhost 网络地址，防止其他应用调用 exe 接口，保证系统的安全；

FLT-User.exe 可在电脑右下角弹出窗口，解决 GUI 交互的问题，实现用户交互；

Agent 拆分的具体设计图：

客户端交互弹窗：

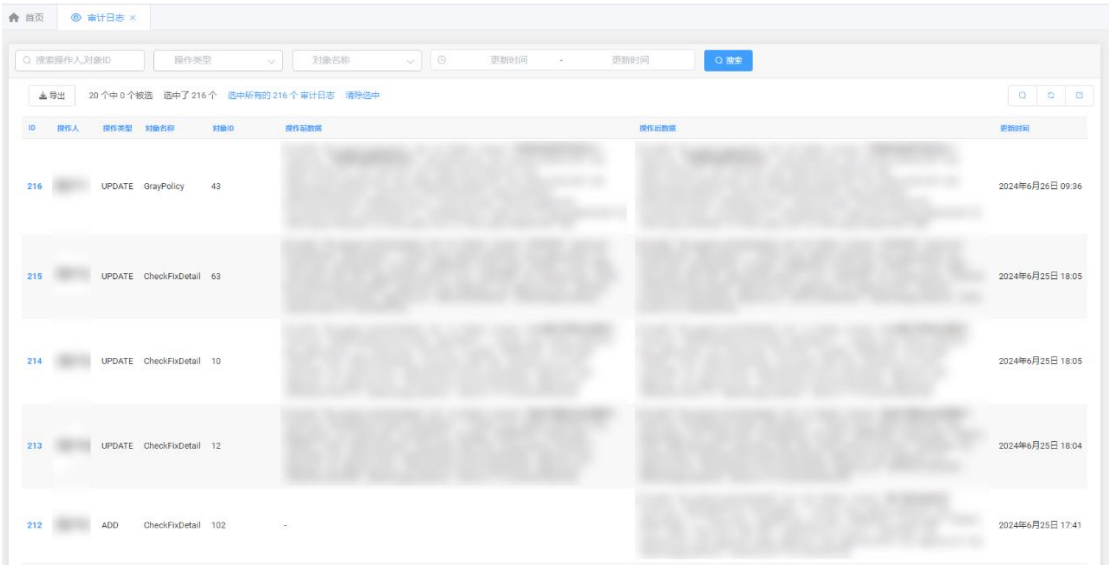
4) 客户端覆盖率提升

全链路工具 FLT-User.exe 使用当前登录用户权限启动，故在注销状态下无法运行，导致注销状态下的客户端数据无法采集，为了提升客户端的覆盖面，再次对全链路 Agent 调度逻辑进行优化。注销状态时 FLT-System.exe 和 FLT-User.exe 均采用 System 权限执行，同时限制只运行 System 权限相关的脚本(User 权限相关脚本可能会执行异常，故在此情况下不再调度)，注销状态下无需弹框提醒，GUI 交互无问题；

5) 运维故障快速定位和解决

全链路工具覆盖公司所有 Windows 客户端，全链路运维工作很容易造成线上故障，风险较大，例如：工程师无意修改错检查项标准值，就会导致采集的数据产生大量的异常值，对运维工作会造成极大的负面影响，且排障难度较大。为能快速定位和解决运维故障，在运维管理系统中增加了运维日志模块（审计日志），记录后台页面操作的所有日志，可以快速查询所有的配置修改，为系统长期稳定运行打好基础。

审计日志查询页面如下：



6) 采集链路脚本执行情况统计

由于采集数据是通过客户端单线程队列顺序执行脚本实现，脚本执行的耗时会直接影响采集的效率，再对所有采集工作耗时情况统计分析，正常情况单次采集工作的总耗时平均在 2min ~ 5min，非正常的采集耗时有的超过 1 小时，这种非正常采集降低了全链路的可用性和准确性。对 FLT-System、FLT-User 调度脚本采集数据的逻辑上进行优化，新增脚本执行耗时统计、脚本执行超时限制（在脚本管理后台可配置），同时数据采集记录表新增“采集耗时”字段，记录检测项脚本执行的时间，为后续各个检查、修复脚本的优化提供数据支持。同时对执行超时的脚本进行主动中止，避免过多消耗客户端资源。优化后提升了整个采集工作的效率，进一步完善了全链路工具。

四、成果

通过全链路工具的运营，实现了电脑故障的自动检测、提前发现、自动修复，改变了以往用户报障后被动式排障的运营管理模式。通过自动化故障检测和主动修复机制，全面提高了电脑的稳定性和安全性，整体人工事件量降低明显，提高了员工满意度。

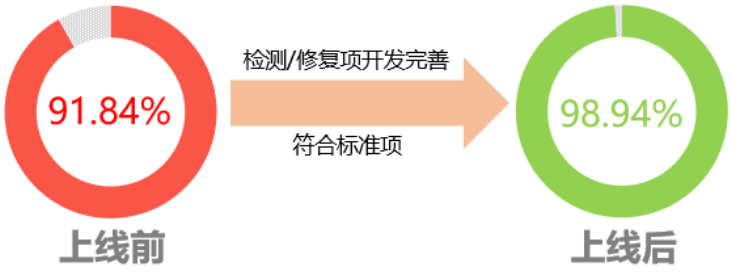
全链路工具上线后，使得 IT 运维团队能够通过技术手段，更加有效的管理全公司数万台用途各异、配置不同的办公电脑，大大提高运维团队的工作效率。并且提供了准实时的电脑运行数据监控，同时为网络质量检测、办公电脑数据收集等业务场景提供了有力的支撑。确保满足多样化员工需求和信息安全的前提下，能够高效、稳定的开展工作。

系统上线前后，各类 PC 主要问题，周平均故障量下降 20%-30%。运维团队人工服务量下降超过 10%，业务效果显著。

上线前后主要检查修复项故障量下降:

内容	上线前月平均	上线后台月平均	平均下降
客服插件	114	46	59.65%
磁盘空间不足	69	43	37.68%
Admin 权限问题	48	34	29.17%
根证书问题	13	10	23.08%
远程服务/权限/端口	58	51	12.07%
域账号问题	197	180	8.63%

全链路检测项范围内客户端健康度提升:



五、未来

全链路数据采集脚本的执行需要依赖于客户端的环境，相同的脚本在不同的客户端执行后可能有不同的结果（成功或者异常），后续需要具体分析脚本异常问题，提升数据有效性。在未来需要持续不懈地优化和完善系统，不断突破性能的极限，赋予全链路更加强大的功能，为企业提供更为卓越的桌面系统运营解决方案。

大数据

携程数据基础平台 2.0 建设，多机房架构下的演进

【作者简介】cxzl25，携程高级软件技术专家，关注数据领域生态建设，对分布式计算和存储、调度等方面有浓厚兴趣，Apache Kyuubi PMC Member，Apache Celeborn / ORC Committer。

一、背景

携程数据基础平台主要组件包括：HDFS 分布式存储集群，YARN 计算集群，Spark、Hive 计算引擎。数据基础平台 1.0 版本的架构从 2017 年开始逐步成型，2018 年至 2021 年数据基础团队基于 1.0 的架构做了性能优化和各类 Bug 修复，支撑集群数据和计算任务高速增长。

进入 2023 年以来，随着业务恢复，数据平台存量数据也不断增长，单日数据量净增长超过数 PB，增速前所未见，2 个 IDC 的数据机房物理机架位告急。

在 OPS 团队的大力支持下，启动了第三个 IDC 数据机房建设项目，2 个月内交付了新 IDC。

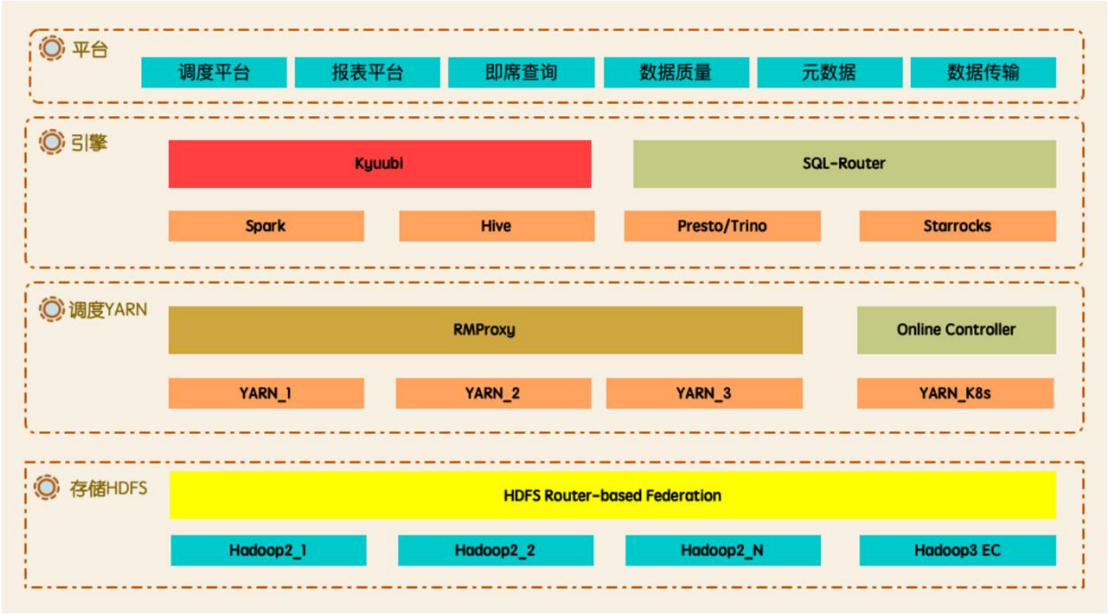
二、面临的问题

随着集群规模不断增长，2022-2023 年亟待解决的基础平台几大痛点：

- 多机房架构支持三数据中心架构，数据存储和计算调度
- 数据迅速增长、机房需要建设周期，冷数据搬迁上云上对象存储可以有效缓解整体存储容量压力，降低综合成本
- 数据量增长导致算力资源缺乏，需扩大离线在线混部资源规模且能实时互相借调
- 计算引擎 Spark2 需要平滑升级 Spark3

三、整体架构

在 2022-2023 年持续演进过程中，数据平台 2.0 整体架构如下图所示。存储层支持多机房架构， 热/温/冷三分层数据，透明迁移，并且具备读取缓存，透明加速的能力。调度层支持灵活的优先级调度，NodeManager 节点混部，离线和在线节点混部，还引入了 Celeborn 作为新的 Shuffle service。引擎层从 Spark2 升级到 Spark3，使用 Kyuubi 作为 Spark 的查询入口。



四、存储

4.1 多机房架构升级：支撑三个以上数据中心架构

Hadoop 多机房架构升级，数据支持按 IDC 或者跨多个 IDC 共享，Client 支持就近读写，避免产生跨机房流量，新增数据中心对使用数据平台的用户无感知。

4.2 分层存储：热/温/冷三分层数据存储架构落地，对接云上对象冷归档存储，降低存储成本

存储和计算引擎多方联动升级改造：支持热/温/冷分层，热数据放私有云热节点，温数据放私有云 Erasure Coding (EC) 冷节点，冷数据周期性搬迁至云上对象存储的超冷归档存储。

4.3 透明迁移

HDFS 原生支持的多个迁移工具，在迁移过程对用户来说完全透明。

- Balancer - 平衡不同的 DataNode 数据分布
- Mover - 迁移数据到期望的存储类型的 DataNode，比如迁移在 DISK 类型的数据到 ARCHIVE 类型的 DataNode
- Disk Balancer - 平衡同一个 DataNode 不同磁盘的数据分布

但是在 HDFS Router-based Federation (RBF) 的架构下，因为单组 NameNode 存储的文件数有上限，又或者因为 RPC 导致 NameNode 响应慢，通常的做法是增加一组或者多组 NameNode，并把不同的部门的数据拆分到不同的 Namespace，所以经常会有跨 HDFS 集群 Namespace 迁移。

或者是因为新增 IDC，需要将某个部门的数据和计算任务都迁移到新的 IDC，以缓解数据和计算能力的不足。

又或者存储的数据转换成 Erasure Coding 编码的数据，以节约数据存储成本。

这里涉及到多种数据迁移，是原生的 HDFS 迁移工具不支持的，怎么做成对用户较为透明的迁移?

对于新增数据，可以利用 RBF 的 Mount Table 特性，让新增的数据指向新的 Namespace，旧的数据可以逐步迁移，或者随着 Time to live (TTL) 旧的数据自动删除，不需要迁移。但是如果存在跨目录的 rename 操作，可能不太适用。

对于全量数据，基于 FaceBook 开源版本的单机版 FastCopy，在 DistCp 的基础上扩展了分布式的 FastCopy 的方案，并修复和支持多个开源版的问题，比如不支持非 DISK 类型的 DataNode 的 FastCopy，addBlock RPC 的重试导致 missing block。FastCopy 的原理是选择对源文件的 Block 对应的多个 DataNode 执行 Hard link，并汇报到新的集群，以达到快速迁移 HDFS 集群元数据，但是不需要复制 Block 数据的目的。

跨机房的数据迁移，实现跨机房副本分布配置，支持路径的数据可以按单机房或者多机房策略放置，实现 Client 可以就近读和写，对用户无感。

转 EC 编码，实现数据迁移工具，通过配置近多少天读取次数小于 X 次的规则，实现自动化的迁移。迁移完成后切换 RBF 的 Mount Table，Client 就可以直接读取 EC 编码的数据，并且支持静态分区，动态分区的历史分区数据回刷。

4.4 读取透明加速

存储在 HDFS 集群的数据大多数是一次写入多次读取，由于 HDFS 本身提供的 HDFS 集中式缓存管理 (Centralized Cache Management) 功能较有限，所以引入了 Alluxio 组件，基于社区版实现了透明 URI 访问，多 IDC 自动选 master，单集群多租户等功能，无需更改 Location，与计算引擎集成打通，用户可以直接透明使用缓存读取功能。

五、调度

5.1 优先级调度

与 ETL 作业调度，元数据管理平台打通，基于表的重要等级自动提升任务链路的优先级，对 P0，P1，PX 任务分类，在 YARN 调度器实现优先级调度，保证任务 SLA。

5.2 NodeManager 节点混部

在白天业务低峰的时候，集群会临时下线部分 NodeManager 节点，切换成 Presto，Trino，StarRocks 的计算节点，以应对白天较多的 Ad-Hoc 查询及报表查询。

这带来几个问题，下线 NodeManager 需要快速清理已调度的 Container，但是 Spark 或者 MapReduce 依赖的 Shuffle service 可能会因为 NodeManager 停止服务而无法拉取 Shuffle data，导致计算任务的 Task 局部失败并重试，拉长任务整体完成时间。

对此，实现了临时下线 NodeManager 时，仍然保留了 Shuffle service 的服务线程和端口，这样保证了 Shuffle 过程不失败。并且对 P0，P1 重要的任务则实现了 NodeManager 混部节点黑名单机制，保证重要作业申请的 Container 资源不会调度到这些混部的计算节点，防止 Task 在下线的过程中运行失败。

5.3 离线和在线节点混部

在线服务应用的资源使用情况随着终端用户的访问数量而变化，不少应用存在夜间 CPU 利用率较低，具备潮汐特性，而数据计算任务通常都在凌晨有较高的资源需求，YARN 集群经常出现 App 或者 Container pending 等资源不足的问题。

通过对离线作业 Spark、MapReduce 和 Kyuubi Spark Engine 的画像分析，收集读取，Shuffle，写入等作业指标，区分任务优先级，与 ETL 作业调度平台联动，提交到在线集群基于 K8s 部署的的 YARN 集群，并通过 YARN Node Label 特性实现灵活 Container 调度。

5.4 Remote shuffle service Celeborn

在 Spark on YARN 的方案，开启 Spark 动态资源分配时，往往需要在 NodeManager 部署 Spark External Shuffle Service (ESS)，在 Executor 闲置回收之后提供 Shuffle 数据的读取服务。

ESS 虽然经过一系列优化，比如 Shuffle write 结束合并成一个大文件，以避免在 NM 创建大量的小文件，但是仍然无法避免几个问题。Shuffle read 存在大量的随机读，NM 有大量的磁盘 IOWait，导致 FetchFailed，进而 Stage 需要重新计算。并且一次 Shuffle read 会创建 M*N 次的连接数，当 MapTask 和 Shuffle partition 较大规模时，作业经常因为 Connection Timeout 或者 Reset 而触发 FetchFailed。

在 Spark on K8s 的方案，目前还不支持 External Shuffle Service，所以目前要想在 K8s 开启 Spark 动态资源分配，只能开启 spark.dynamicAllocation.shuffleTracking.enabled=true，这样 Executor 当没有 active 的 shuffle 数据，就可以被释放回收，整体资源释放时间被拉长。

基于上述多个问题，引入了 Remote shuffle service(RSS) Celeborn 组件，在多个 IDC 的 NodeManager 节点，混部了 Celeborn，与 Spark 引擎集成，并在 Ad-Hoc 查询平台，ETL 调度平台灰度开启。

Celeborn 服务可以解决和优化目前 ESS 存在的问题。Celeborn 优势如下：

- 使用 Push-Style Shuffle 代替 Pull-Style，减少 Mapper 的内存压力

- 支持 IO 聚合，Shuffle Read 的连接数从 M*N 降到 N，同时更改随机读为顺序读
- 支持两副本机制，降低 Fetch Fail 概率
- 支持计算与存储分离架构，与计算集群分离
- 解决 Spark on Kubernetes 时对本地磁盘的依赖

六、计算引擎

6.1 Spark3

2017 年引入 Apache Spark 2.2，基于此版本做了不少定制化的开发，实现多租户的 Thrift Server，基本替代了 Hive CLI/HiveServer2 SQL，成为携程主流的 SQL 引擎，服务于 ETL 计算，Ad-Hoc 查询和报表。

在 2020 年 6 月，Spark3.0 正式发布，有强大的自适应查询执行 (Adaptive Query Execution) 功能，通过在运行时对查询执行计划进行优化，允许 Spark Planner 在运行时执行可选的执行计划，这些计划将基于运行时统计数据进行优化，比如动态合并 Shuffle Partitions，动态调整 Join 策略 ，动态优化倾斜的 Join，从而提升性能。

2021 年 10 月随着 Spark 3.2 发布，开始着手调研升级的可行性，最终经过一系列的探索，移植多个 Spark2 定制需求，完成了 Spark2 到 Spark3 的平滑升级。

6.1.1 Spark3 平滑升级

1）使用 Kyuubi plan only mode 重放线上 SQL，分类语法不兼容的类型

Kyuubi Spark Engine 设置 kyuubi.operation.plan.only.mode=OPTIMIZE，结合元数据，获取提交的 SQL 的优化之后的执行计划，可以按 SQL 错误类型归类。

2）与 Hive SQL 、Hive meta store、Spark2 SQL 兼容

- 扩展 BasicWriteTaskStats，收集和记录非分区表、分区表（静态分区，动态分区） 多种写入类型写入的行数，文件数，数据大小 (numRows，numFiles，totalSize)。
- Spark 建的视图与 Hive 兼容

在 Spark 在 USE DB 之后建的视图，会导致 Hive 读 View 失败，因为 viewExpandedText 没有完全重写，当前 DB 的信息存储在 Hive meta store 的 View 的 table properties，Hive 读取 View 对应的 Table 因为没有 USE DB 而找不到对应的表。

在 Hive 执行 DDL 修改 Spark 视图的类型定义，会导致 Spark 读取 View 失败，因为

Spark 建 View 的时候会把当前 schema 存储在 View 的 table properties 的 spark.sql.sources.schema，Spark 读取 View 时 schema 再从此属性恢复，由于 Hive 修改 View 不会同步修改这个属性，这导致 Spark 读取 Hive 修改后的 View 失败。

这里采用 Hive 的做法，重写 viewExpandedText，补全当前的 DB 信息，同时去掉存储在 table properties 的 schema，保证多个引擎可以修改，可以读取。

- 避免全量永久 UDF 加载

Spark 在某些模式下启动可能会从 Hive meta store 拉取所有 DB 的永久 UDF 定义，这导致 Spark 启动较慢，对 Hive meta store 负载有一定影响。需要避免直接初始化 Hive Client，这样能避免全量永久 UDF 加载。

[SPARK-37561][SQL] Avoid loading all functions when obtaining hive's DelegationToken

- 避免创建 0 Size 的 ORC 文件

Hive 的实现 OrcOutputFormat 在 close 方法，如果该 Task 无数据可以写，在 close 的时候会创建一个 0 size 的 ORC 文件，较低的 Hive 版本或者 Spark2 依赖的 ORC 较低版本不支持读。

虽然 ORC-162 (Handle 0 byte files as empty ORC files) 补丁可以修复此问题，但是对多个组件的低版本进行升级是一件较为困难的事，所以采取了对 Spark3 依赖的 Hive 版本进行修复，创建一个无数据空 schema 的 ORC 文件，保证灰度升级的时候，Spark3 产出的数据文件，下游 Spark，Hive 都可以正常读取该表的数据。

3）移植 Spark2 自定义特性，部分 Rule 通过 SparkSessionExtensions 注入

在早期二次定制开发 Spark2 的时候，Spark2 还没有丰富的 API 接口供开发者注入自定义的实现，这导致了一些个性化的特性直接耦合在 Spark2 的源码中，这给升级 Spark3 移植特性带来诸多不便，代码散落在各个代码文件，移植的时候可能会遗漏，缺少一些端到端的测试。

在 Spark3 升级的过程中，重新梳理定制化需求，尽可能剥离出来新的代码文件，并抽离出一些 SQL Rule，包装成 Spark plugin，注入到 SparkSessionExtensions，方便后续的升级及维护。

4）基于 SBT 在 GitLab 构建 CI/CD，快速集成

在二次开发 Spark 或者 backport 社区 Patch，Spark 需要一个完整的测试工作流，社区版的 CI 是基于 GitHub action 构建的，在内部的 GitLab 参考了类似的 workflow，因为 SBT 构建和测试速度比 Maven 快很多，所以基于 SBT，拆出 10+个 Module，可以并行测试，并且一旦编译通过，自动化部署对应的分支的 jar 到验证环境，供开发者进一步调试。极大提高了 Spark Merge request 合并代码的稳定性和 Code review 的效率，也使得生产环境

的 Spark 更为健壮。

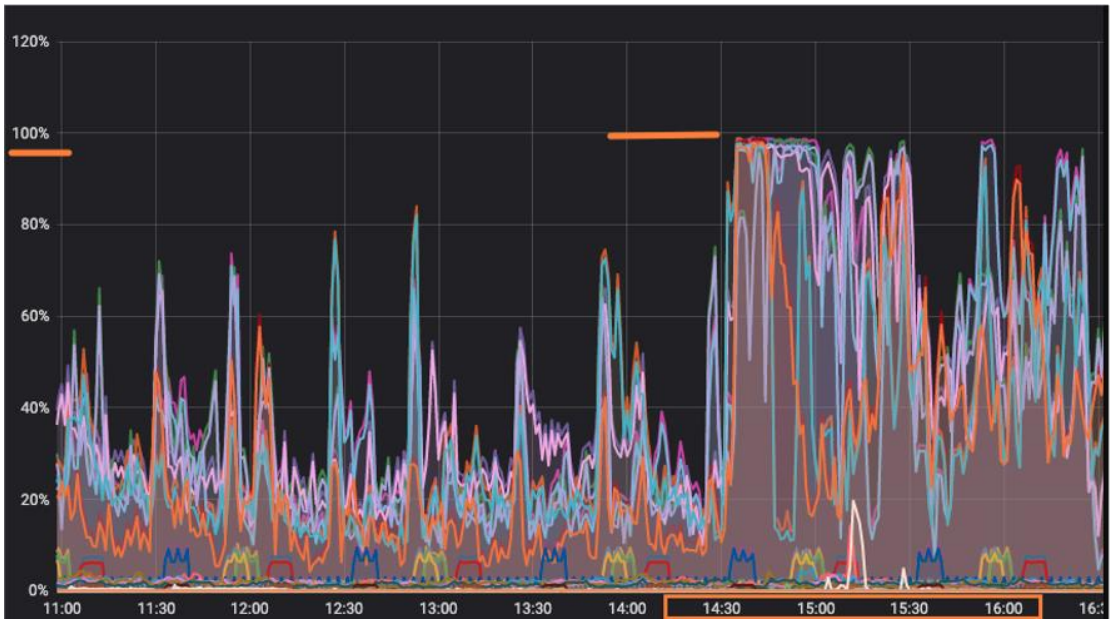
5）灰度升级策略，任务粒度切换

与 ETL 调度平台联动，支持任务级别或者按任务优先级的百分比，从 Spark2 灰度切换 Spark3，失败可自动 fallback，并且有数据质量平台，每个任务完成之后，都有相应的数据校验保证，另外还有一些运行时间对比，错误监控。

6.1.2 分区过滤函数优化

查询一张数万个分区表，在 Hive 查询引擎使用函数 substr 对分区字段 d 进行过滤，它使用 Hive meta store 提供的 get_partitions_by_expr RPC 进行分区裁剪，最终 Client 只需要获取少量的符合条件的几个分区。

但是在 Spark 实现的分区裁剪，不支持函数，所以如果有 where substr(d,1,10) = '2023-01-01' 函数过滤分区的 SQL，会造成 Hive meta store 因为需要获取大量分区而导致 CPU 被打爆到 100%，并且 Client 会因为获取太多分区详情会导致 OOM 而失败。



分析 Spark 关于分区裁剪的调用链路，Spark 先是把支持的算子转换成 Hive 支持过滤的 Filter SQL，如果支持转换，就直接使用 get_partitions_by_filter RPC 获取分区详情。如果不支持转换，则使用 get_partitions RPC 获取所有分区详情，再通过 Spark 的算子进行分区值的过滤，调用代价太高。

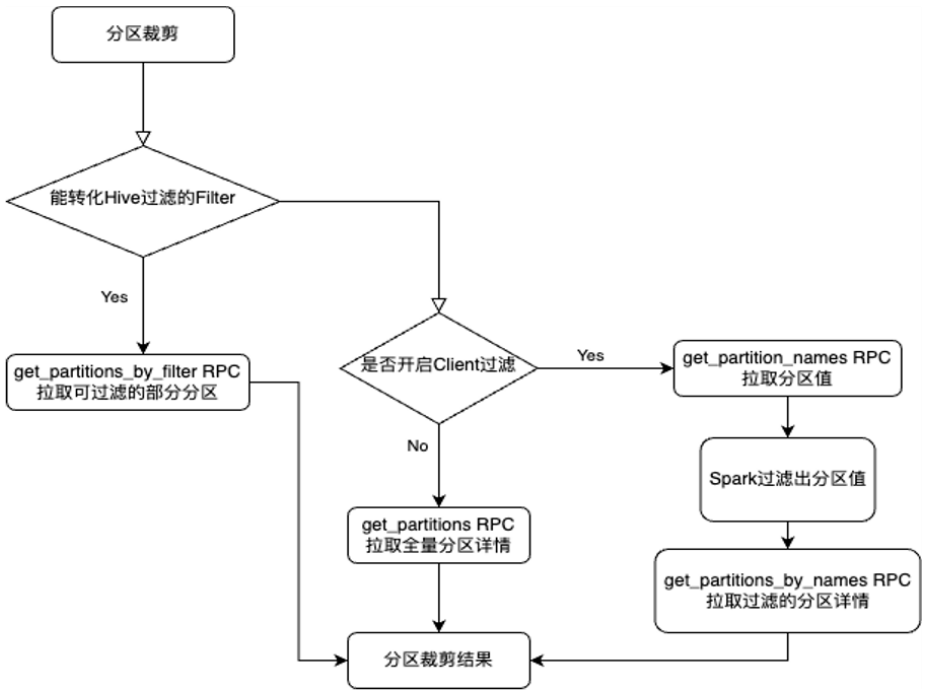
如果是借鉴 Hive 的实现，因为 Spark 的函数和 Hive 提供的函数定义不一定一样，Spark 的函数可能在 Hive 没有实现，所以 Hive 实现的 get_partitions_by_expr 在 Spark 侧不太适用。

[SPARK-33707][SQL] Support multiple types of function partition pruning on hive metastore

这里采取了另外一个思路，在不支持转换 Filter SQL 的时候，先是获取调用 get_partition_names RPC 获取分区列表，再通过 Spark 算子过滤出所需的分区值，接着调用 get_partitions_by_names RPC 获取过滤后对应的分区值的详情，调用耗时从数十分钟降到秒级别，极大的提升了分区裁剪的效率。

社区版本提供了一个配置项，需要通过 spark.sql.hive.metastorePartitionPruningFastFallback=true 打开此特性。

[SPARK-35437][SQL] Use expressions to filter Hive partitions at client side



6.1.3 数据倾斜

虽然在 Spark3 AQE 可以自动优化倾斜 Join，但是在部分场景仍然存在倾斜 Key 的情况，比如 Stage 没有 Shuffle，缺少运行时统计信息，而 Skew join 需要通过统计信息计算出不同的 Partition 是否存在倾斜才可以进行优化。

首先在 Spark 实现定位数据倾斜 Key，在 SortMergeJoin 注入 JoinKeyRecorder，采集每个 Task join 的 key 的行数和最大行数的 key，类似于 Hive 的 JoinOperator 的实现。

接着在诊断平台的 Event log parser 实现相应的解析，提取 Join key 和行数，当用户诊断作业的时候，可以显示是否存在倾斜 Key 和倾斜行数。

诊断平台是基于罗盘 (compass) 开源二次开发，集成在 Spark History UI 和企业 IM 的诊断机器人，用户可以自助诊断，输入调度系统的作业 Id 或者 App Id，Bot 即可生成诊断报告。

诊断机器人

生成的诊断报告

6.2 Kyuubi

6.2.1 Spark2 Thrift Server

与 Hive 提供的 HiveServer2 对应的 Spark Thrift Server (STS) 是 Apache Spark 社区基于 HiveServer2 实现的一个 Thrift 服务，目标是做到无缝兼容 HiveServer2。与 HiveServer2 类似，通过 JDBC 接口提交 SQL 到 Thrift Server。

相比于 HiveServer2，Spark Thrift Server 是比较脆弱的。Spark Driver 比 Hive Driver 更为繁忙一点。Hive 负责编译和优化 SQL，提交 MapReduce Job，轮询结果，而 Spark Driver 不仅仅要做 Hive 的类似事情，还需要管理资源调度，按需增加和减少 Executors，调度 Job、Task 执行，广播变量、小表，这也导致了 Spark Driver 更容易有 OOM 的问题，当这个问题出现在 Driver 与 Server 绑定的同个进程中，问题就更为严峻，Server crash 的话可能导致多个 Session 的查询直接失败。

原生的 STS 还存在下列的问题：

- Server 单点问题

不支持类似 HiveServer2 通过 Zookeeper 实现 High Availability

[SPARK-11100] HiveThriftServer HA issue,HiveThriftServer not registering with Zookeeper

- 不支持多个不同的用户

Thrift Server 不能以提交查询的用户取代启动 Thrift Server 的用户来执行查询语句，类似 HiveServer2 hive.server2.enable.doAs

[SPARK-5159] Thrift server does not respect hive.server2.enable.doAs=true

- 不支持 Cluster 模式，受限于 Driver 启动的机器的内存

基于上述原生 Spark Thrift Server 不能够满足需求，在 Spark2 扩展了一些实现，比如支持多租户，基于 Zookeeper 实现 High Availability。

实现多租户的功能，是在 Client 发起 openSession 时，Server 在 SparkSQLSessionManager.openSession 对当前的 Session User 申请 HDFS DelegationTokens 和 Hive DelegationTokens。这一块 Token 传递和刷新和 Spark2 Streaming 更新 Token 逻辑类似。

然后在 DAGScheduler submit job 的时候关联 SQL，JobId，User 信息，并绑定到 Task。

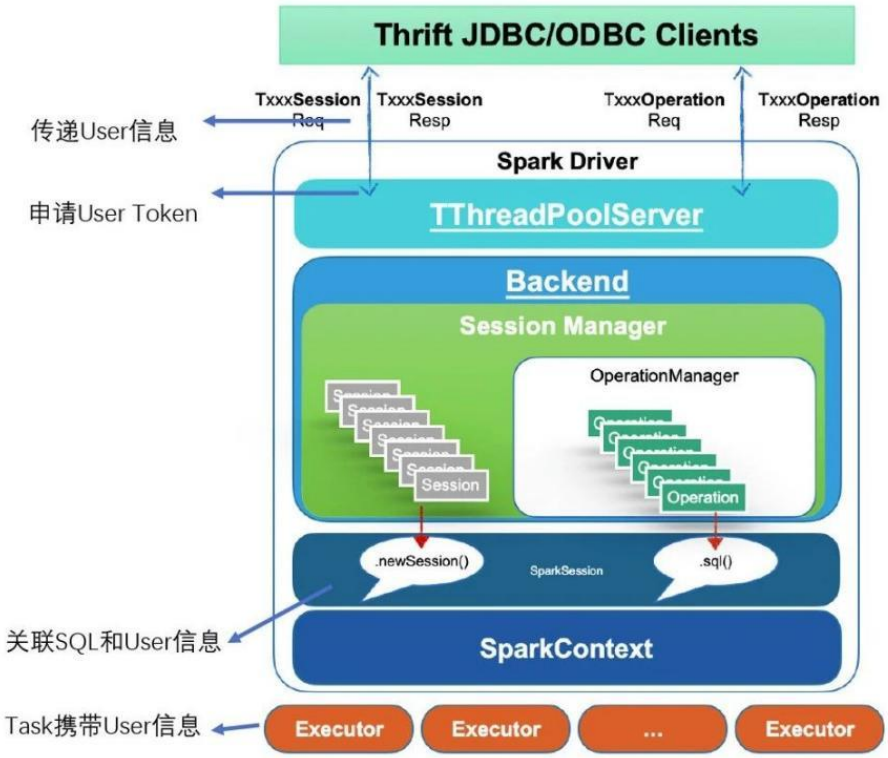
接着在 Executor 使用 Task 对应的 UGI doAs 执行。

由于 Spark2 还有多处的实现用到了线程池，这里也需要模拟成不同的用户去执行。

- BroadcastExchangeExec.executionContext 全局的线程池
- UnionRDD.partitionEvalTaskSupport 全局的 ForkJoinPool
- HIVE-13120: propagate doAs when generating ORC splits

这样的实现也存在了不少的局限

- 在 YARN 层面 App 对应的用户是超级用户，不能细粒度划分资源
- Spark Jars、Files 是全局共享的，这导致了 UDF 隔离性不是很好
- 扩展特性对 Spark Core 、SQL、ThriftServer 模块改动较多，与 Spark 版本深度绑定



6.2.2 Kyuubi Spark3 Thrift Server

在升级 Spark3 的时候，决定废弃原有的 Spark2 的 Thrift Server 的改造实现，引入 Apache Kyuubi 项目。

Kyuubi 有如下的优点

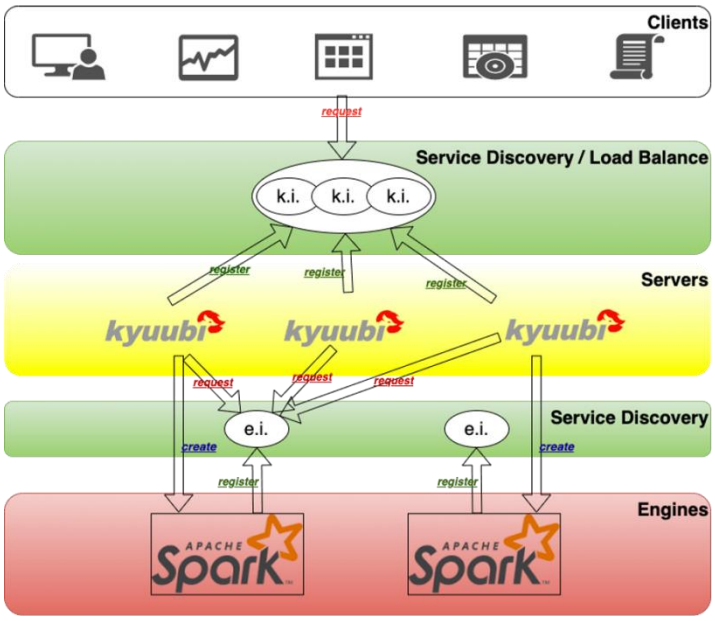
- 隔离性好，支持资源队列隔离，Engine 隔离
- 设计天然多租户，计费友好，支持 Cluster 模式
- 不与 Spark 具体版本绑定，支持 N 个大小 Spark3 版本
- 使用 Explain 模式，可以预解析 SQL
- 支持 Server、Engine graceful stop
- 可以按不同的用户进行个性化配置

Kyuubi 的架构分为两层，一层是 Server 层，一层是 Engine 层。

Server 层和 Engine 层都有一个服务发现层，Kyuubi Server 层的服务发现层用于随机选择一个 Kyuubi Server，Kyuubi Server 对于所有用户来共享的。

Kyuubi Engine 层的服务发现层对用户来说是不可见的。它是用于 Kyuubi Server 去选择对应的用户的 Spark Engine，当一条用户的请求进来之后，它会随机选择一个 Kyuubi Server，Kyuubi Server 会去 Engine 的服务发现层选择一个 Engine。如果 Engine 不存在，它就会创建一个 Spark Engine，这个 Engine 启动之后会向 Engine 的服务发现层去注册，然后

Kyuubi Server 和 Engine 之间的再进行一个内部连接。所以说 Kyuubi Server 是所有用户共享，Kyuubi Engine 是用户之间资源隔离。



目前 Kyuubi 完全替换了原先的 Spark2 Thrift Server 服务，作为即度查询，质量校验，报表系统的 Spark 入口。

- 动态远程配置

基于远程配置中心，推送各种配置，按用户，用户组开启

- 动态分时注销 Engine

白天允许 Engine 闲置时间更长，避免冷启动 Engine 较慢

- 动态调度 Engine 集群

历史画像分析，使用资源较小的 Engine 允许调度到离线在线混部集群

6.2.3 Kyuubi 全链路血缘跟踪

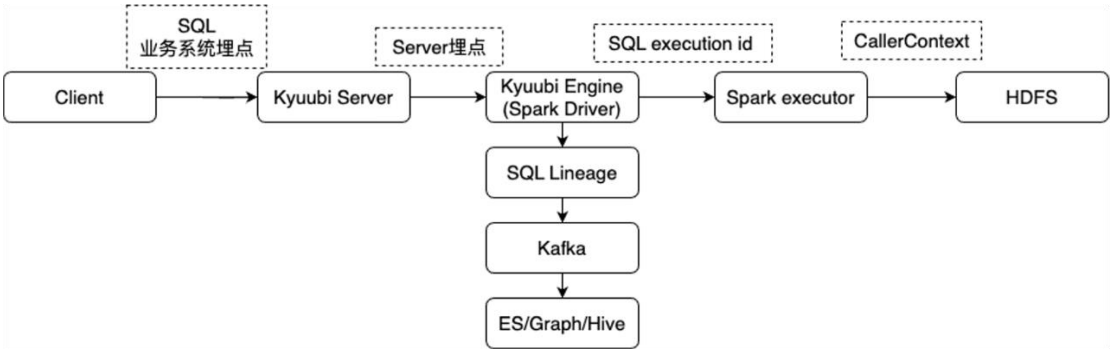
在多租户共享 Engine 的情况，如何精细化跟踪每条 SQL？对此，实现了一个全链路的血缘跟踪。

在 SQL Lineage 层面，基于 spark.sql.queryExecutionListeners 的 API 实现，采集了 Kyuubi Server/Engine IP，Session Id，Operation Id 和当前的 Spark 的 YARN Application Id，SQL 执行的 Id。

在 HDFS 的 Audit log 层面，拓展了 Spark Task 的 CallerContext 的实现，埋入 SQL 执

行的 Id。

这样可以基于每条的 SQL execution id 关联整条链路，在 SQL lineage 层面可以知道哪个 session 的哪次执行读取了什么数据，写入哪张表，在 HDFS 的 Audit log，可以看到具体是哪个 SQL 对应的 ID 访问了哪些数据文件，以达到精细化追踪和运营的目的。



SQL lineage


```
duration          482
endTime           2023-09-18 14:48:00.035
engineName        spark
exData            {"kyuubiServerIp":"10.xx.xx.xx","kyuubiOperationId":"88981051-edeb-4ad8-a8c0-abe
                  492dacca9","adhocQueryUser":"sychen","kyuubiSessionId":"4e97b06e-a44e-4f1f-993c-
                  2c451beba3da","sparkInternalVersion":"3.2.0-vXX","sparkSqlExecutionId":"13","use
                  rDefine":"{\"realUser\":\"sychen\",\"engine\":\"spark3\",\"ip\":\"10.xx.xx.xx
                  \"}"}
inputCols         [tmp_testdb.tmp_test_query.c1, tmp_testdb.tmp_test_query.c2, tmp_testdb.tmp_test
                  _query.d, tmp_testdb.tmp_test_query.h]
inputTables       [tmp_testdb.tmp_test_query]
loginUserName     admin
queryId           application_1692154506624_3280208
queryString       SELECT *
                  FROM tmp_testdb.tmp_test_query
sourceSystemId    40638715
sourceSystemName  adhoc
startTime         2023-09-18 14:47:59.553
```


HDFS audit log


```
2023-09-18 14:48:00,004 INFO FSNamesystem.audit: allowed=true   ugi=admin (auth:PROXY) via
(auth:KERBEROS)        ip=/          cmd=open        src=/user/hive/warehouse/tmp_testdb.db/tmp_test_query/d=2023-09-18/h=14/p
art-00000-40081541-eeae-495f-bea6-4d2126549a11.c000       dst=null        perm=null       proto=rpc       c=          SPARK_TASK_applicati
on_1692154506624_3280208_1_JId_2_SId_2_0_TId_7_0_QId_13
2023-09-18 14:48:00,008 INFO FSNamesystem.audit: allowed=true   ugi=admin (auth:PROXY) via
(auth:KERBEROS)        ip=/          cmd=open        src=/user/hive/warehouse/tmp_testdb.db/tmp_test_query/d=2023-09-17/h=11/p
art-00000-3fdd91d4-8e1f-408b-bd7b-48b06fddd13c.c000       dst=null        perm=null       proto=rpc       c=          SPARK_TASK_applicati
on_1692154506624_3280208_1_JId_2_SId_2_0_TId_6_0_QId_13
```


七、总结

在多个基础组件协同联动，齐头并进，取得了如下的收益：

1）架构层面优化收益

数据基础平台 1.0 架构从 2017 年到 2022 年稳定运行 5 年，达到瓶颈，新的 2.0 架构预期在 2023 落地建设完成后，具备可扩展性，预期在近几年内可以为集团数据保驾护航，确保集团数据计算任务持续、稳定、高效运行，在数据量快速增长的情况下，多数据中心+冷数据上云的架构也将具备很高的韧性。

2）存储引擎优化收益

具备热、温、冷数据，缓存分层存储的能力，支持多数据中心存储和迁移。

3）调度引擎优化收益

优先级调度保证了 P0、P1 任务整体按时达成率，2023 年新版本离线在线混部工具开发完成后，日均借调 CPU 逻辑核心数也有数万核，用在线集群闲置资源为离线计算提速。

4）计算引擎优化收益

- 从 Spark2 无感升级到 Spark3，支撑日均运行超过 60 万 Spark 任务，提升运行速度约 40%
- 落地数据服务网关 Kyuubi，动态分时扩缩容，动态调度集群，日均超过 30 万查询量
- 落地 Alluxio，实现透明访问 Hive 表，自动冷热分离，部分场景下提升 30-50%读取速度
- 落地 Celeborn，计算引擎 Spark 集成，可支持更小粒度的离线在线混部
- 支持多种数据湖组件，支持多种存储类型，热数据，EC 冷数据，云上冷数据读取多种特性

未来将持续深入数据组件生态，并适时引入新的技术栈，通过不断探索和创新，致力于优化系统架构，以提升集群的稳定性和提高数据处理效率，确保系统的可靠性和性能，满足不断增长的业务需求，为用户提供更优质的服务体验。

性能指标提升 50%+，携程数据报表平台查询效率治理实践

【作者简介】携程 OLAP 引擎开发组，专注于大数据 OLAP 引擎 trino/starrocks 的开发与大规模部署和运维。

本文概述了面对公司数据报表平台遇到的查询性能挑战，数据平台组围绕数据缓存、物化视图、查询策略、SQL 质量等方向所做的一系列治理工作，以提升平台的查询效率和稳定性。通过这些工作，平台的查询响应时间得到了显著的改善，其中平均响应时间从原来的 8 秒降低至 4 秒，响应时间 90 线由原先的约 18 秒降低至约 8 秒，总体性能指标提升幅度达 50%以上。本文在各个小节中对各治理策略的关键原理和思路进行了阐述，希望能够为读者提供一定的参考和启发。

一、背景

数据报表平台（代称 Nova，后同）用于支持携程内部数据分析、数据挖掘、数据可视化等业务需求，目前每日承载数十万 Hive 表 AP 查询，所涉数据量达万亿级别。随着用户基数逐步提升，承载查询量不断增大，平台查询性能面临挑战，具体表现如下：

1）平均响应时间延长，大查询在业务高峰期存在阻塞现象，超时数量增多；

2）查询所需时间不稳定，性能波动较大，在业务高峰期可能出现响应时间突增现象；

3）查询负载集群资源占用率高，CPU、内存资源吃紧，I/O 请求排队等待，进而导致集群稳定性下降，时有节点宕机现象出现。

针对上述现象，我们从平台自身服务、SQL 路由分发组件、SQL 执行引擎等方面入手，采用了一套“全方位组合拳”对平台的查询性能进行治理，目标有二：

1）从用户体验角度：改善查询性能，提升查询效率和稳定性；

2）从集群维护角度：提升集群稳定性，增强查询结果复用能力，提高算力使用效率。

二、平台设计概览

数据报表平台执行查询的主要链路如图 1 所示，其中有几个关键构件：

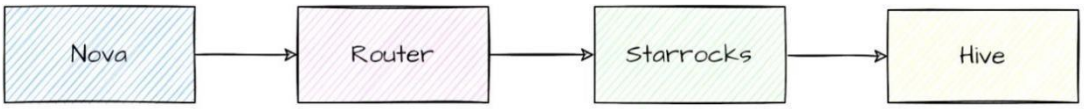


图 1：Nova 数据查询链路

1）Nova：应用本体，提供可视化用户界面，包含报表即时查询、执行离线定时任务等功能；
2）Router：用于分离指向不同引擎的查询请求，起到 SQL 路由功能；
3）Starrocks & Hive：平台使用 Starrocks 作为主要查询引擎，向 Hive 外表发起查询请求。

三、多维度数据缓存

在硬件资源有限的情况下，要提升查询性能，最直观的思想是对重复的查询进行结果复用。在对平台的查询请求数据进行统计分析后，可发现存在相当数量的查询请求在不同时段内重复出现，这为我们引入缓存机制提供了实践基础。

若将在执行过程中可能遭遇瓶颈的查询进行划分，可将大致分为 I/O 型、计算型和高频型三类，其中 I/O 型查询对网络和磁盘带宽的要求较高，往往涉及大规模数据的扫描；计算型查询对 CPU 和内存资源的要求较高，往往涉及大量连接、分组、聚合、筛选、再计算操作；高频型查询的单次调用开销可能较小，但在单位时间内发起的次数显著高于均值，在涉及远程调用（如元数据获取）的环节可能遭遇性能瓶颈，且在单位时间造成的资源开销可能与大查询相当。

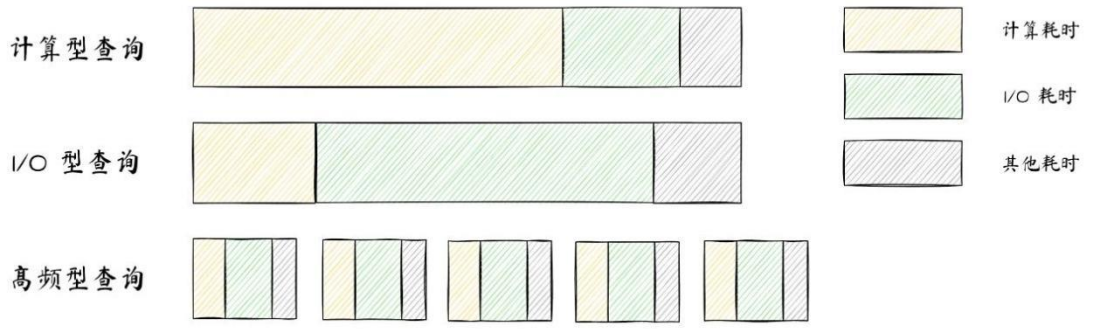


图 2：受限查询分类

目前，在整个数据查询链路中，我们在以下几个环节引入了缓存机制，以应对不同类型查询所带来的挑战。

3.1 底表 Data Cache 预热

当 Starrocks 从 Hive 外表进行数据查询时，Scan 算子会将所需数据文件以块的形式读取至本地。对于典型的 I/O 型查询而言，这个过程所需时间可达整个查询流程耗时的 70%以上。在业务高峰期，由于大量查询请求同时发起，I/O 型查询的堆积将导致其他查询请求读取数据文件的等待时间增长，进而影响查询响应时间。

针对这类情况，我们在 Starrocks 集群中开启了 Data Cache，将每次读取得到的文件块标识并临时存储在本地磁盘中，在下次查询请求需要相同文件块时，若发现该文件块没有更新，则直接从本地磁盘读取，避免了经由网络和 Hive 带来的文件读取延迟。

通过 Data Cache 缓存的文件块在 Starrocks 中由带冷热分区的 LRU 队列维护，当队列满时，将根据文件块的访问频率和时间戳进行淘汰，以保证缓存的命中率。

从缓存一致性角度，Starrocks 在使用 Data Cache 时，会通过元数据判断底表数据是否发生更新，若发现数据文件已更新，则将废弃缓存数据，重新拉取底表数据文件，以保证查询结果的准确性。

3.1.1 预热机制

通过对查询请求命中底表的情况进行统计，可发现其中热点表的使用呈现一定的规律性（如：每日相近时刻、每周固定几日访问量达峰等）。为此，我们为统计得到的各热点表建立了用户画像，记录并预测其访问高峰。

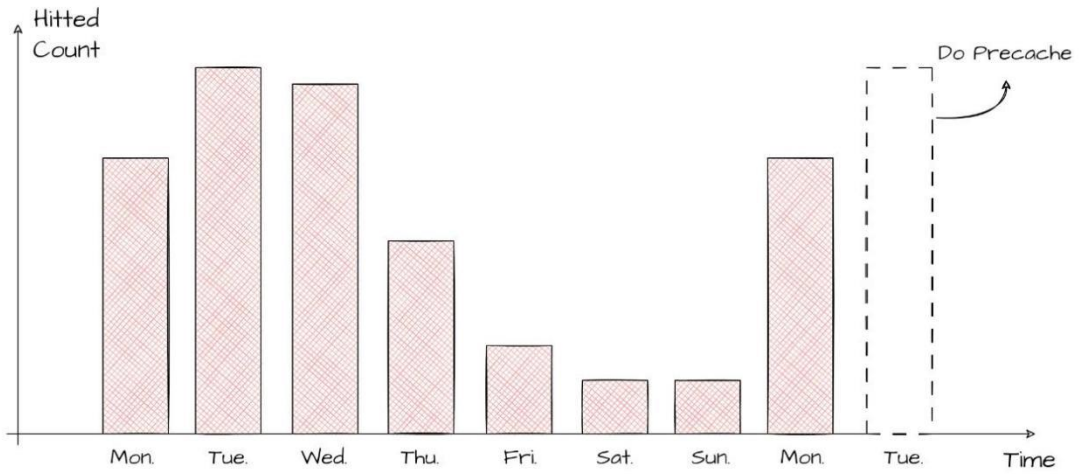


图 3：预热机制

通过在业务高峰到来前将热点表数据主动 Cache 预热，可进一步分散业务高峰期的 I/O 压力。如图 4 所示，这部分主动指定 Cache 的数据文件将会优先被置入 LRU 队列的热区，以保证其在高峰期的查询中能够被快速命中。

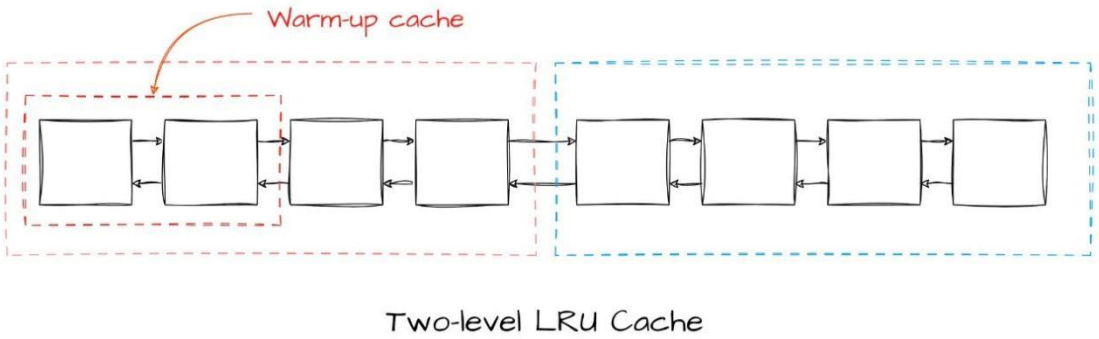


图 4：Data Cache LRU 队列

3.2 HDFS 元数据缓存

在 Starrocks 查询引擎执行查询时，需要获取 Hive 表的元数据信息，如表的列信息、分区信息、表的存储格式；HDFS File 的元数据信息，如 block 块属性等。Starrocks 将通过这些信息来生成最优的执行计划。在业务高峰期，大量并发查询可能导致 Hadoop Namenode 的元数据请求压力过大，进而影响查询的执行效率。

Starrocks 原有的元数据缓存时间较短，这是因为其无法实时感知 HDFS File 文件变化。为防止缓存不一致，原有的 Remote File 元数据缓存时间不宜设置过长，但这会导致即使元数据未更新，在某些场景下 Starrocks 也会频繁发起重复的元数据请求。

为此，我们选择在 Starrocks 的 FE 侧通过元数据接口（该接口调用开销远小于元数据的获取）对元数据的新鲜度进行检测，仅在远端元数据发生更新时拉取元数据，此功能可使得 Remote File 缓存时长延长至 6 小时，在保证缓存一致性的同时，提升相同元数据的复用程度。

3.3 Router Redis 缓存

当数据从查询引擎返回至 Router 时，Router 会将查询结果进行缓存，以便后续面对完全相同的查询请求，可直接从缓存中获取结果，避免重复计算。

在缓存一致性方面，Router 同样可通过血缘信息实时获取底表数据的更新时间，从而判断缓存数据是否过期。在缓存数据过期时，Router 将弃用缓存数据，重新执行查询，以保证查询结果的准确性。

3.4 Download 缓存

对于查得的报表数据，平台支持用户将数据下载至本地，以便用户进一步分析。在实践过程中我们发现，部分报表数据除了在查询后被即时下载外，还可能需要在未来的某个时间点被再次下载（如浏览器关闭后）。这种行为将触发二次查询，导致相同请求被反复执行以满足下载需求。

针对这种现象，我们在 Download 服务中同样引入了缓存机制，将下载的数据进行缓存，以便后续相同的下载请求直接从缓存中获取数据。

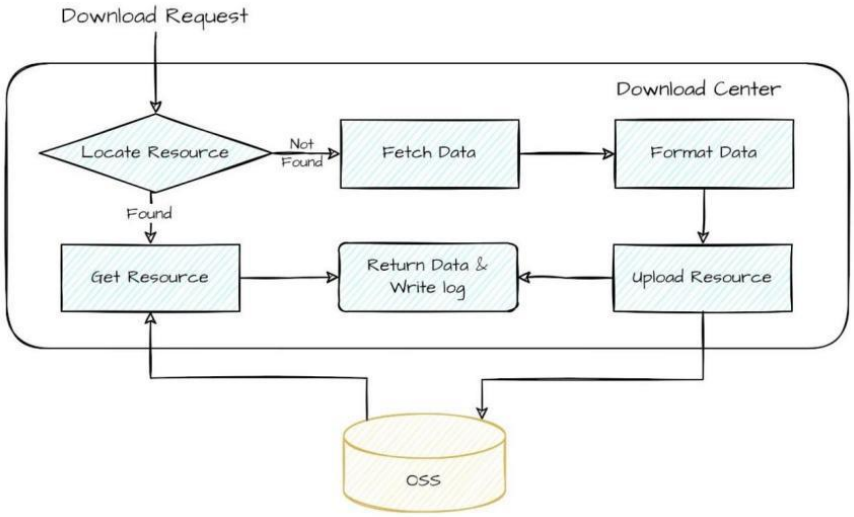


图 5：Download 缓存

3.5 小结

如图 6 所示，通过引入多维度数据缓存机制，我们在平台的数据传输的全链路中，尽可能实现了对重复数据的复用。而为了进一步提升在计算过程中，数据和算力的使用效率，我们进一步在物化视图的使用上进行了探索。

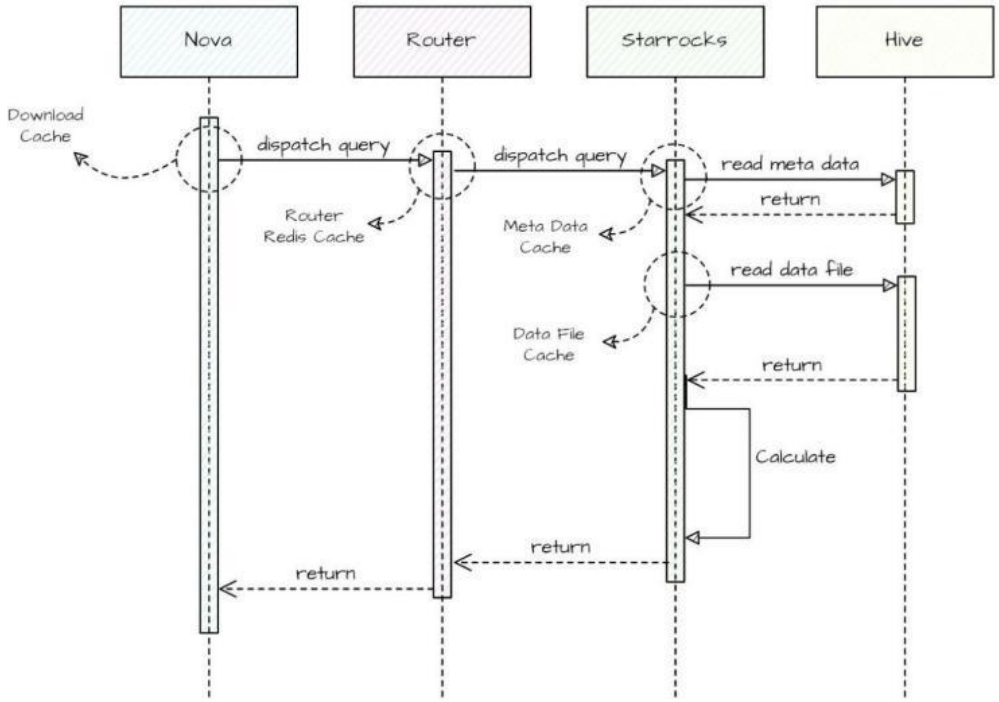


图 6：查询链路缓存总览

四、使用物化视图加速查询

物化视图（Materialized View, MV）作为一种预先计算的结果集，可以有效减少查询时的计算量，提升查询性能。面对计算型查询，MV 可以将查询计算的部分结果进行固化存储，避免复杂计算逻辑的重复执行；面对 I/O 型查询，MV 可以将查询所需数据进行预聚合，减少数据扫描的规模，或者更简单地，将需要频繁查询的数据底表保存为 MV，作为优先级更高的 Data Cache 来使用，使得对 Hive 外表的查询达到和 Starrocks 内表一致的查询性能。不过，想要发挥 MV 的优势，需要解决如下几个问题：

1）视图定义：如何创建有效的 MV；

2）视图利用：如何在查询时高效地利用 MV；

3）视图维护：如何保证 MV 的新鲜度。

4.1 视图定义：为数据集构建 MV

在 Nova 平台构建数据报表时，用户首先需要创建数据集，最终在数据集定义的范围内进行数据查询、生成可视化报表。这一特点使得在选择 MV 的目标时，将数据集作为构建对象是一个较为合适的选择，因为它代表着一系列特定业务逻辑的公共数据查询需求。

在用户自定义的数据集中，视构建 MV 的难度，可将数据集分为以下三类：

1）静态数据集：数据集的定义不随使用时间、用户及其他环境因素发生变化，在底表数据不发生变化的前提下，执行任意次查询都将返回相同结果。此类数据集在构建一次 MV 后便无需再次修改视图定义；

2）半静态数据集：数据集的定义按日（或更长的时间单位）进行更新，但其可变部分仅限于日期（配置了日期变参）。面对此种类型的数据集，需要构建模板，按日重新渲染视图；

3）动态数据集：即使数据集所涉底表的数据未有更新，但随查询的执行时间不同（如包含 CURRENT_TIMESTAMP 函数或 RAND 函数等）、执行查询的用户不同（数据集配置了用户变参）、查询的上下文参数不同（数据集配置了自定义变量）等，查询结果也会发生变化。此类数据集的 MV 构建较为困难，需要进一步拆解数据集的定义，分离得到其中静态的部分以构建 MV，其极端情况为仅对数据集的底表进行物化。

通过将半静态或动态数据集转换为静态形式，并对静态数据集构建 MV，可将各类查询的中间计算结果进行固化，在确保 MV 所涉数据底表中的数据没有更新的前提下，可将 MV 中的数据进行重复利用。

4.2 视图利用：MV 自动改写

面对一个潜在的可利用已有 MV 数据的查询，Starrocks 提供 MV 自动改写的能力，可将查询计划中的相关计算逻辑改写为直接从 MV 进行读取，从而减少查询的计算量。

Starrocks 提供两类视图改写规则：SPJG 改写和文本匹配改写。

4.2.1 SPJG 模式改写

SPJG 模式改写是一种基于逻辑计划的改写规则，其原理基于这篇论文：《Optimizing Queries Using Materialized Views: A Practical, Scalable Solution》。

这种改写规则的核心思想在于：首先确保查询（或某一逻辑子树构成的查询）所需的全部数据均可由 MV 查询得到，随后计算查询与 MV 间的谓词差异（称为补偿谓词），应用至可用于改写的视图上，构成一个新的查询计划。图 7 是改写规则生效的一个示例。

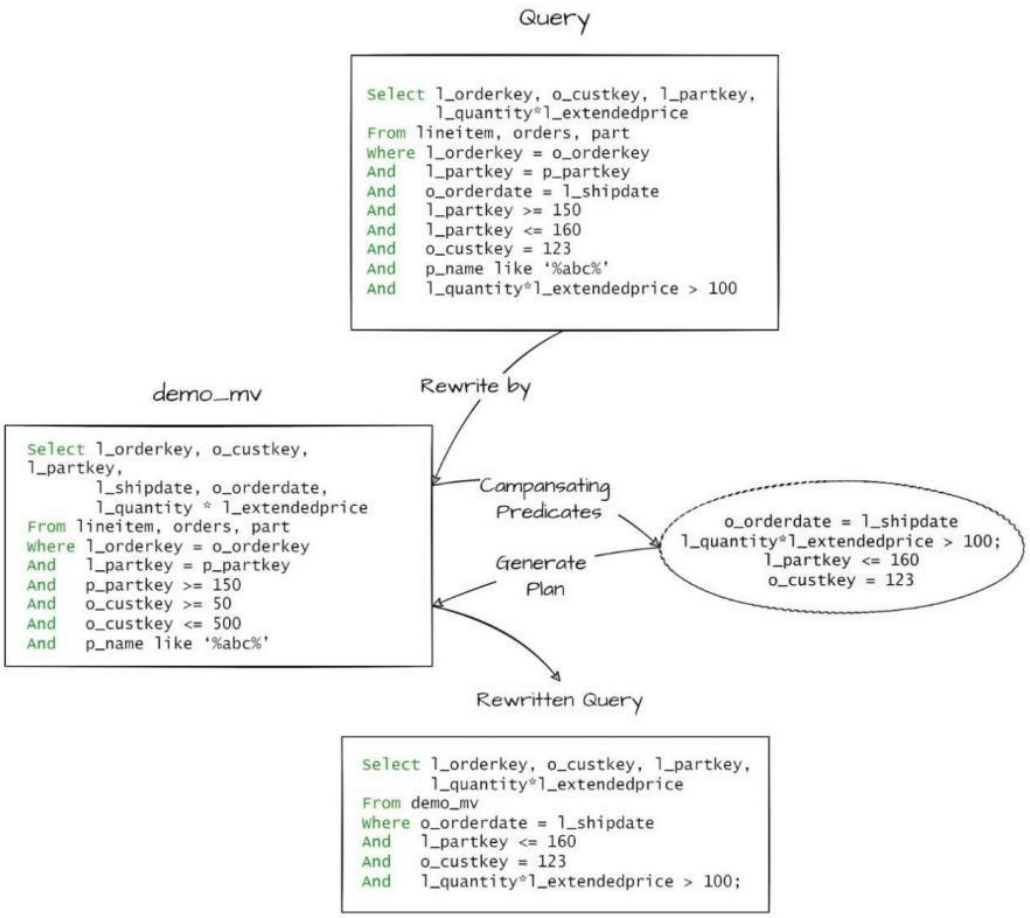


图 7：SPJG 模式改写示例

SPJG 模式改写的优势在于，面对建立在相同数据集上的各类不同查询，Starrocks 可根据实际情况灵活复用已经在 MV 中完成预计算的数据，从而达成“一次计算，多次使用”的目的。

不过，由于这种改写的复杂性，目前仅支持对只包含 Select、Project、Join 和 Group-By 这四类算子（SPJG）的查询计划树（或逻辑子树）进行改写，而在涉及 Union、Order-By、Limit 等算子的查询改写时能力受限。面对复杂的数据集，更具实践性的做法是抽取出其中的 SPJG 模式子树用于创建 MV，以扩展 MV 的适用范围，增大匹配改写成功率。

4.2.2 文本匹配改写

文本匹配改写是一种基于抽象语法树（AST）的改写规则，通过比对 Query 和 MV 规范化后的 AST 是否一致，可判断是否可以将查询改写至 MV 上。

这种改写的优势在于可支持更多类型算子的查询改写，但其改写的自由度和适应性不如 SPJG 模式改写，一旦数据集或查询的 SQL 结构在处理过程中发生变化，则文本匹配改写可能无法发挥作用。

4.2.3 改写数据一致性

通过合理利用上述两类 MV 改写规则，可做到以用户无感的方式，自动复用中间计算结果，在节约计算资源的同时大幅提升复杂查询的效率。

值得一提的是，在数据一致性方面，Starrocks 在执行查询改写流程时，会自动检测 MV 中的数据是否过期，若是，则放弃改写，执行原有查询计划。这使得引入 MV 查询改写机制后，在达成查询加速效果的同时，依然能够保证查询结果的准确性。

而由于此机制的存在，截至目前，Starrocks 在执行 MV 改写时，若发现 MV 本身的定义中包含非确定返回值（Non-deterministic）函数，例如 CURRENT_DATE 和 RAND 等，将弃用此 MV，这是因为 Starrocks 无法保证 MV 返回的结果在当前时刻下依然可用。因此，在选择数据集进行物化时，必须先确保数据集的定义为静态，这也是“视图定义”小节对数据集分类的重要意义所在。

4.2.4 CURRENT_DATE 函数改写问题

在构建 MV 时，我们发现部分数据集的定义中包含 CURRENT_DATE 函数，而并未使用平台提供的日期变参（$EFFECTDATE）。这将导致即使数据集 SQL 的文本定义未发生变化，随着时间推进，数据集本身所指代的数据范围却按日发生变化。

与平台提供的日期变参不同的是，CURRENT_DATE 函数在执行时不会被替换为具体日期，而是在查询时由执行引擎动态计算，这使得查询引擎在执行改写操作时，将由于涉及非确定返回值函数而弃用此类数据集对应的 MV，使得这类本该满足半静态数据集条件的数据集需要被作为动态数据集处理。

经统计，报表平台中目前有相当数量的数据集定义，其可变部分仅为 CURRENT_ DATE 函数，为提升对此类数据集的物化能力，我们在平台侧引入了重渲机制，在将查询请求提交至引擎前，会将 CURRENT_DATE 函数以和 EFFECTDATE 变参类似的方式重渲为具体日期，从而保证查询引擎的改写机制能够正常生效。

4.3 视图维护：MV 自动刷新

在确保 MV SQL 为静态的前提下，MV 的数据新鲜度仅和底表数据是否更新有关。通过分

析 MV 创建语句中 SQL 所涉及的底表追溯血缘依赖关系，并通过元数据服务获取底表的最近更新时间，可创建自动化应用实时监控 MV 是否过期，并根据实际情况选择是否刷新 MV。

4.4 MV 价值发掘

为数据集创建 MV 并不一定必然为平台的查询性能带来优化，原因可罗列为以下几点：

1）MV 的创建和维护需要消耗额外的计算资源，每一次刷新都对应一次对数据集的查询操作。若 MV 的使用率较低，其带来的性能提升可能无法弥补其维护成本；

2）部分数据集在定义时所涉及的数据范围远大于真正使用所需，在直接执行查询操作时，可通过谓词下推等优化操作过滤掉不必要的扫描范围，而如果为此类数据集创建 MV，将反而导致更大的全局计算开销；

3）Starrocks 集群能够提供的磁盘空间有限，不可能为所有数据集都创建 MV 来优化。在有限的资源下，需保证“好钢用在刀刃上”，优先创建有较大查询性能提升潜力的 MV 进行创建。

为此，我们在 MV 的选择过程中，引入了 MV 价值评估机制，通过综合分析数据集的使用频率、数据集计算及相关查询的计算代价、数据集的数据规模等多个维度，为数据集的 MV 创建提供参考。以查询所消耗的 CPU 代价为例，建立 MV 前后的所节省的计算代价可使用如下公式计算：

$$\Delta_{cost} = \sum_{i=1}^{N} (Cost_{cpu}(Q_i) - Cost_{cpu}(Q_i)') \times Freq(Q_i) - Cost_{cpu}(V) \times Freq(V)$$

为精确捕获查询执行或数据集刷新的计算代价，我们会将统计得到的待改写查询（高代价或高频查询）在独立环境下执行预跑，获取量化数据作为评估的依据，大致测试流程如图 8 所示。

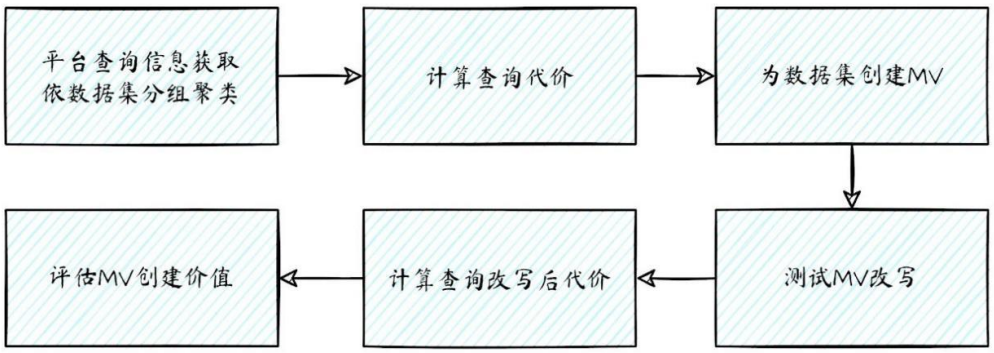


图 8：MV 价值评估流程

4.5 小结

通过选取合适的数据集构建 MV，并利用 Starrocks 的自动改写能力加速查询，可在一定程度上规避平台在计算过程中的出现的短板，补足全查询链路的数据复用能力，使得数据报表平台面对不同类型的高负载查询，有更加成熟、可靠、易维护的应对方案。

五、查询策略优化和 SQL 质量治理

目前，数据报表平台所承载的查询任务主要分为两类：一是即时报表查询，用户通过 Web 界面即时运行查询并获取数据报表；二是离线定时任务调度，用户通过配置定时任务执行查询，任务触发后用户可通过邮件等途径获取数据。

这两类查询任务分别存在以下特征：

1）即时报表查询：触发时间随机，由用户自主触发，查询请求所涉计算量相对较小，但对查询响应时间更为敏感，一般要求在数秒内返回结果；

2）离线定时任务调度：触发时间固定，由预注册的计划周期性自动触发，查询需要计算的数据量往往较大，但对查询响应时间要求相对较低，一般可容忍的查询响应延迟较宽。

立足于平台自身的业务场景思考，可发现以下痛点：

1）负载不均问题：平台执行查询的高峰和低谷期分明，需要调整定时任务调度策略，以平滑负载；

2）资源争用问题：相同时间段内，不同查询间存在资源争用，例如离线定时任务的执行可能影响同期即时报表查询的性能；

3）慢查询问题：部分查询请求自身所需的计算资源过大，或是配置的计算逻辑不佳，可能对平台造成过量负载。

对于慢查询问题，其成因可能有多种，包括但不限于以下几点：

1）平台查询策略问题：面对特定查询，原有的实现或查询策略在执行时性能较差，需要进行改进；

2）查询 SQL 实现欠优：查询 SQL 在实现上存在质量问题，在执行时消耗的计算量远大于业务逻辑所需，可能引入大量非必要的底表扫描和计算操作，需要整治优化；

3）业务需求：即使优化了查询策略和 SQL 质量，仍有部分查询请求所需计算量较大，对于其中涉及关键业务的部分，可能需要整合计算资源，以专门提升这类查询的性能。

在本节中，将分别对目前我们针对上述痛点所做的工作进行阐述。

5.1 整点调度问题治理

Nova 平台所提供的离线任务定时调度功能，由用户根据所需自行配置任务的触发时间。但出于用户习惯，可发现大量查询都被设置在整点进行调度，这导致平台在整点出现查询负载高峰。

通过对此业务场景的痛点进行分析，我们采用了两种策略来解决整点调度问题。

一方面，通过与用户沟通，我们提出了“错峰调度”策略，即在原有配置的基础上，对部分任务进行时间错峰调度，以减少整点负载高峰。例如，原定于每日九点执行的调度任务，可能被延迟至九点十分执行。这种策略在一定程度上缓解了整点负载高峰的问题，而不至于为用户带来过多的使用不便。

另一方面，我们提供了“依赖调度”方案，即不再以时间，而是以数据更新为触发条件进行调度。报表元数据完成更新时间的随机性，使得这种调度方式可变相地起到“错峰调度”的作用，且通过数据更新作为触发条件，可使得报表数据新鲜度具有更加便捷的维护方式。

5.2 查询流量切分

资源隔离是避免查询请求相互干扰的有效手段。通过将平台的查询请求按照类型进行切分，能够有效提升查询请求的执行效率。

平台的原有的查询请求路由策略仅根据用户指定的查询引擎类型进行请求分流，而根据即时报表查询和离线调度查询的特征，我们选择进一步改进路由策略，将这两类查询请求分发至不同的 Starrocks 集群，防止离线调度过高的查询负载对即时报表查询的响应时间造成影响。

另外，通过对平台每日查询性能数据进行分析，可识别得到一系列慢查询请求，这些 SQL 需要我们根据实际情况细分，并逐步对其进行优化。为避免这些查询请求对平台中其他查询的性能造成影响，我们选择将这类查询分发至专门用于处理待优化查询的独立 Starrocks 集群，以减轻对主集群的压力。

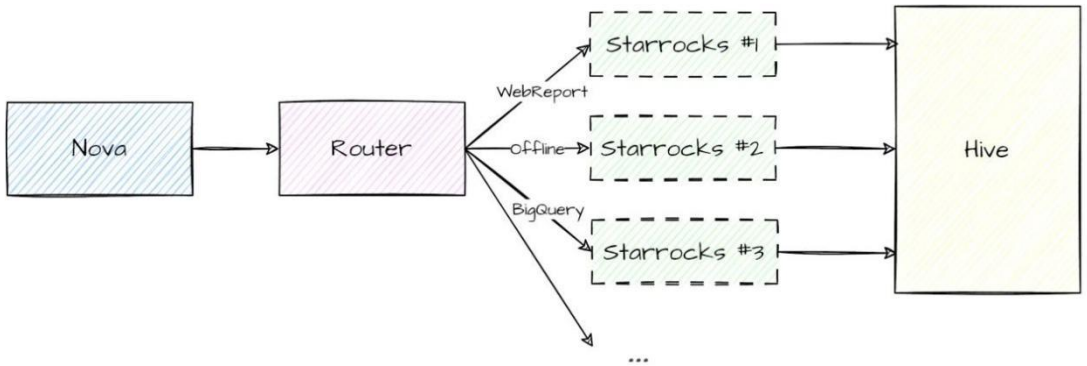


图 9：查询流量切分

5.3 Max-d 查询专项治理

在各类数据集的查询中，有一类典型的 SQL 在实际查询时将引入极高的查询代价。这类查询在对数据条目的日期 d 进行筛选时，使用 max(d)子查询作为谓词判断依据，以获取数据集中所包含的最新一日的数据进行计算，SQL 示例如下所示：

```
SELECT * FROM dataset WHERE d = (SELECT max(d) FROM dataset);
```

此种查询结构将导致查询引擎在实际执行查询时，对数据表的每一行数据都触发一次全表扫描的子查询调用，随着数据表数据量的上升，这种查询的查询代价将呈二次函数倍率增长，产生难以忽视的性能开销。

面对这类查询，我们对其执行策略分两期进行了专项优化。

第一期，在 Router 侧，通过对查询 SQL 的 AST 进行模式匹配，可筛得存在 Max-d 问题的查询请求。在此基础上，我们将这类查询的执行策略拆分为两阶段：首先执行 max(d)子查询，获取最新日期 d 的值，随后将此实值作为谓词条件对原查询进行改写后执行。

通过这种方式，不但可解决原有的嵌套查询问题，将查询扫表的时间复杂度由 $O(N^2)$降至 O(N)，还可进一步触发 Starrocks 的列分区裁剪行为，大幅减少这类查询的计算代价。

第二期，在 Nova 平台侧，直接通过配置变参方式替换 Max-d SQL，然后获取 MetaStore 元数据最新分区替换变量。此举可提升这种执行机制的可维护性，且 max(d)值在计算前即可获取，能够进一步提升优化后的查询性能。

5.4 SQL 实现优化

查询效率优化的核心在于提升查询的执行速度、降低资源消耗，然而单从查询的执行侧进行优化，对于存在潜在质量问题的 SQL，仍然难以解决根本病因。通过分析归纳用户提交的 SQL 中可能导致查询效率低下的原因，我们与用户协调，开展了以下几类 SQL 语句优化工作。

5.4.1 distinct * 语句删减

在撰写 SQL 时，为保证查询结果的唯一性，用户可能会习惯性地添加 distinct * 子句进行去重，然而这种操作涉及到全表数据行、全字段域的扫描去重，当目标数据集合行数或列数较多时，将产生大量的 cpu 和内存资源开销。

为减少这类查询的资源消耗，我们向用户提出了两项修改建议：

1）检查在使用 distinct * 前，计算结果是否已由上游数据源或计算操作去重，避免重复引入去重操作；

2）检查当前去重操作是否必要，是否必须对全部字段进行去重，尽可能减少对去重操作的依赖。

5.4.2 数据表拆分

通过对部分复杂的查询逻辑进行分析，可发现其主要原因是数据模型设计不合理，对应的数据表拆分不当，例如一些权限表和静态信息表存在严重耦合，致使计算逻辑复杂，查询数据量大。

对于这类问题，重新设计底表数据模型，将非必要的耦合部分进行子表拆分，重新定义数据集和报表计算流程，往往可以起到理想的性能优化效果。

5.4.3 限制查询分区

部分用户在创建 SQL（主要是定义数据集）时，未对数据查询范围进行限制，或所做限制缺乏发展性考虑。例如，对一个按日分区的数据表，简单地将数据表查询范围设置为 d > 2022-01-01，而未配置动态参数。随着时间积累，此类数据集的查询资源开销将逐步上升，出现查询性能劣化现象。

在一个 SQL 被执行时，所涉底表分区的数量可能会对查询性能产生重要影响，因为它对应着查询涉及的目标点位数量。为提升此类数据集的查询性能，我们向用户提出了限制查询分区的建议，例如根据所需添加动态日期范围限制，保证报表数据计算的可持续发展性。

5.5 潜在慢查询阻断机制

为防止未经优化的 SQL 对平台的查询性能造成影响，我们在平台侧引入了慢查询阻断机制，对用户提交的 SQL 进行检查，判断其是否满足执行标准，目前主要涉及以下两个点位：

1）平台规范卡点：当用户在平台新增或修改数据集/报表时，平台将对用户提交的 SQL 进行检查，判断其是否满足规范。目前，上述规范将检查用户提交查询的扫描行数、查询耗时和占用内存，而后续将进一步补充对 Join 连接数、底表个数、Union 操作个数、所涉分区数等指标的检查，以保证用户提交的 SQL 质量符合平台的执行标准。

2）Starrocks 熔断机制：当 Starrocks 为查询请求生成执行计划，发现待扫描的文件数和分区数量过大时，将触发熔断机制，跳出此次查询的执行，并返回错误信息。

六、成效

通过以上一系列的查询性能治理措施，我们在数据报表平台的查询性能上达成了阶段性目标，平台查询性能逐步稳定，具体表现如下：

1）查询平均响应时间降低：在业务高峰期，平台的查询平均响应时间从原来的 8 秒降低至 4 秒；

2）查询超时数量显著降低：查询时间 90 线由原先的约 18 秒降低至约 8 秒，由超时导致的查询失败量由日均 7000 次缩减至日均 1400 次；

3）查询性能波动幅度减小：平台每日平均查询性能指标趋于平滑，全平台查询时间标准差由原来的约 25 秒缩短至 14 秒左右；

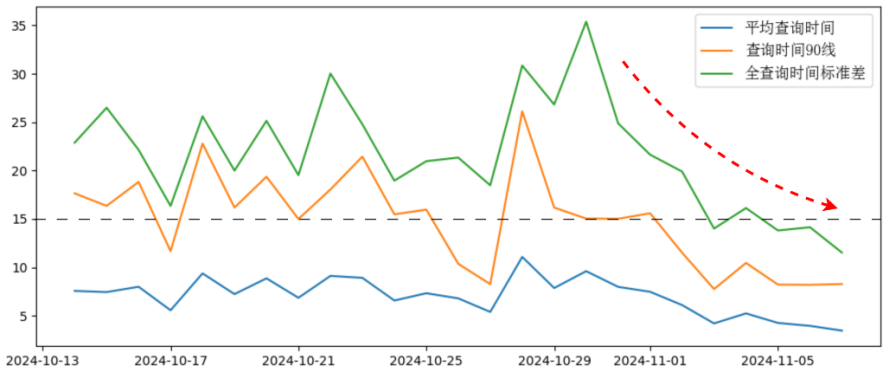


图 10：查询响应时间统计

对平台所承载的各类大查询而言，本文所述的治理策略起到的优化作用尤为明显，由图 11 可见，长耗时查询数量在治理措施实施后呈明显下降趋势。

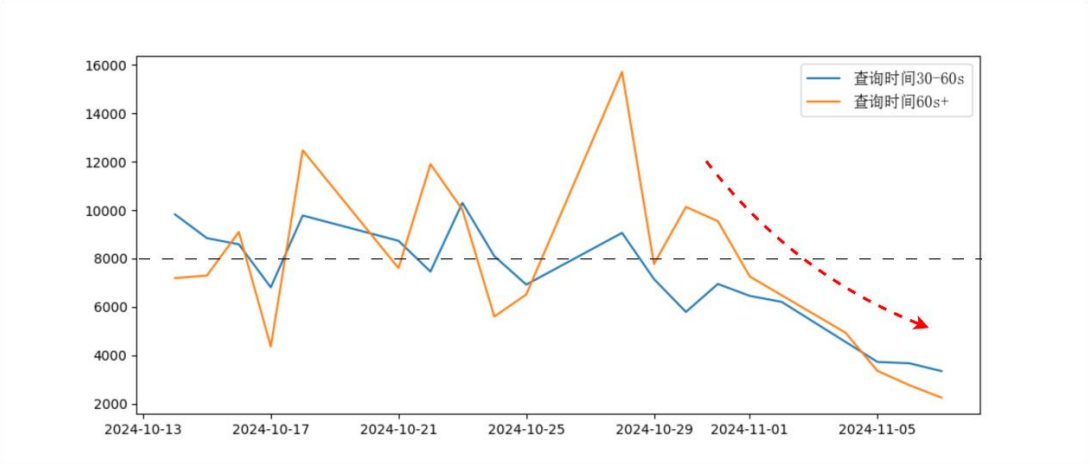


图 11：长耗时查询数量变化趋势

七、总结

通过本文所述内容，我们采取了多项措施来对数据报表平台的查询效率进行治理。通过建立缓存机制、使用物化视图，可提升查询性能和算力使用效率；通过对查询策略进行优化，可裁剪非必要开销，解放平台查询瓶颈；通过切分流量和对 SQL 质量进行管控，可更好地实现资源隔离，提升平台查询质量。

未来，随着治理措施的不断演进，这些优化策略将被逐步规范化、集成化、自动化，以更好地服务于平台的查询需求。

质量保障

携程代码分析平台，快速实现精准测试与应用瘦身

【作者简介】Kevin，携程后端开发专家，追求通过深入业务来简化系统，对底层算法、数据分析有浓厚兴趣。

一、引言

1.1 背景

微服务架构下，产研分工精细，需求迭代频繁，随着需求的不断迭代，应用数、代码量及测试用例越积越多；需求迭代（尤其是有新人加入）的过程中，产品经理需要通过开发了解现状和历史逻辑，开发人员翻阅历史代码花费的时间和精力越来越大，测试人员上线前需要回归的用例也越来越多，严重影响了需求迭代的效率。

1.2 现状分析

目前携程旅游 BG 的后端开发人均应用数超过 4 个，人均维护的代码行近 20 万行；每月平均需求迭代的发布超过 2 千次，其中核心应用数占比及其发布次数占比都超过 8 成。

为了提高需求迭代的效率，旅游技术团队设计开发代码分析平台，对应用的现状（主要是源代码和测试用例）进行综合分析发现：生产应用中高达三分之一的代码属于 dead 代码（没有被引用，也没有任何生产流量），严重影响开发效率；日常迭代中 68%的自动化回归用例与当前迭代无关，但是为了保障上线质量，测试人员需要对每条失败用例进行分析排查，不仅影响当前的交付进度，而且随着需求的快速迭代，自动化测试的可持续性堪忧。

二、代码分析规划

本文主要基于后端 Java 应用介绍如何实现代码分析平台化，并借助平台工具实现精准测试和应用瘦身。平台可以帮助开发人员识别无效代码，在短时间内以最小的风险完成应用瘦身，极大的提高研发的效率；同时通过平台的用例知识库进行精准测试，在需求迭代过程中只执行本次改动相关的用例，极大的提高自动化回归的效率和可持续性。如下图：

图 1 代码分析与精准测试、应用瘦身

2.1 分析应用现状

通过对应用系统综合分析，形成知识库。分析的对象包括源代码（不含第三方引用）、对外提供的服务（包括 api、任务以及消息）、自动化用例、日常迭代变更以及方法维度的生产流量等。知识库包含应用基本信息、统计信息（例如代码规模、方法规模、用例规模）、方法链路信息、用例链路等。

2.2 工具化及流程闭环

利用分析得到的知识库，针对特定场景进行工具化和流程闭环，辅助应用治理。

例如精准测试场景，平台可以与发布流程结合起来，开发提测后自动识别变更内容，并智能推荐自动化用例并执行，将执行结果实时同步给开发和测试人员，实现变更→发布→用例推荐、执行、反馈→修复变更的闭环。

应用瘦身场景，从开发角度来看，平台需提供多视角的辅助分析工具，帮助开发人员确定方法/类是否可以安全的删除；从管理角度来看，平台需划定应用治理前的基线，并将无效代码比例作为应用长期治理对象，从而实时评估下属治理的进度和效果，形成治理过程的闭环。

三、代码分析原理

代码分析的基本单元是方法，主体是应用的整个生命周期，从应用的代码仓库建立以及研发完成代码开发，到测试发布，再到生产运行，我们对不同阶段方法的关联信息进行分析，最终得到一个完整的知识库，分析流程及定义如下图：

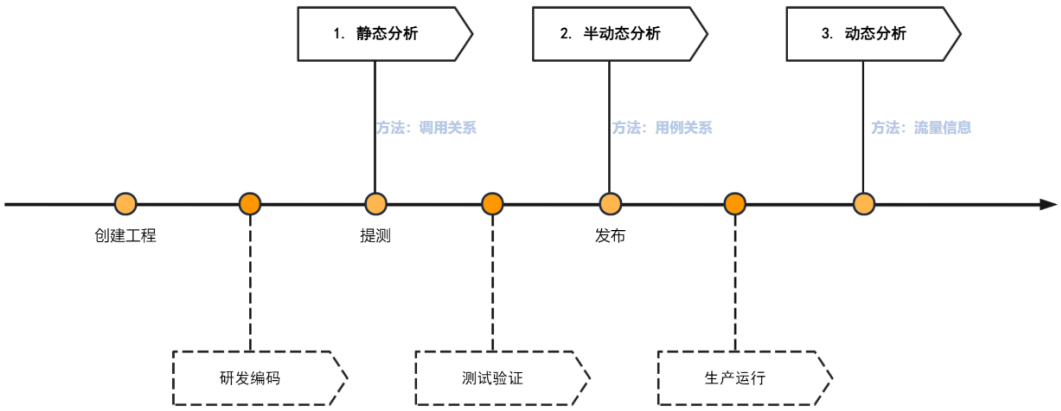


图 2 代码分析原理

3.1 静态分析

通过源代码解析工具解析出所有的方法声明及调用关系。

针对 Java 语言常见的解析工具及原理如下：

工具	原理	适用场景	缺点
FindBugs	解析字节码文件，分析代码的控制流和数据流	检测 Java 代码中的潜在缺陷和错误，如空指针引用、资源未关闭等	工具，不易扩展
PMD	解析源代码，分析代码结构和语义	检测 Java 代码中的潜在问题和错误，如未使用的变量、重复代码等	工具，不易扩展
Checkstyle	解析源代码，分析代码结构和语义	强制执行代码样式和规范，如缩进、命名规范等	工具，不易扩展
IntelliJ IDEA	解析源代码，分析代码结构和语义	检测 Java 代码中的潜在问题和错误，如未使用的变量、空指针引用等	插件，不易扩展
JavaParse	解析源代码成抽象语法树(AST)，提供访问 AST 的 api	Java 官方提供，适用于学习和自研	功能基础
java-callgraph2	解析源代码或动态跟踪程序执行过程来收集方法之间的调用关系	方法、类及调用关系，可基于开源 api 自行扩展	

推荐使用 java-callgraph2，理由是 java-callgraph2 专注于类和方法之间调用关系的分析，解决了很多常见问题，例如 thread、lambda、stream 使用场景的调用关系缺失等，并且在 git 上开源，引入源码可实现定制化。

3.2 半动态分析

通过字节码增强技术（如下图）和用例回放相结合，获取用例执行的方法链，再基于静态分析进行方法映射，达到半动态的效果。半动态分析工具推荐使用携程的开源平台 AREX。

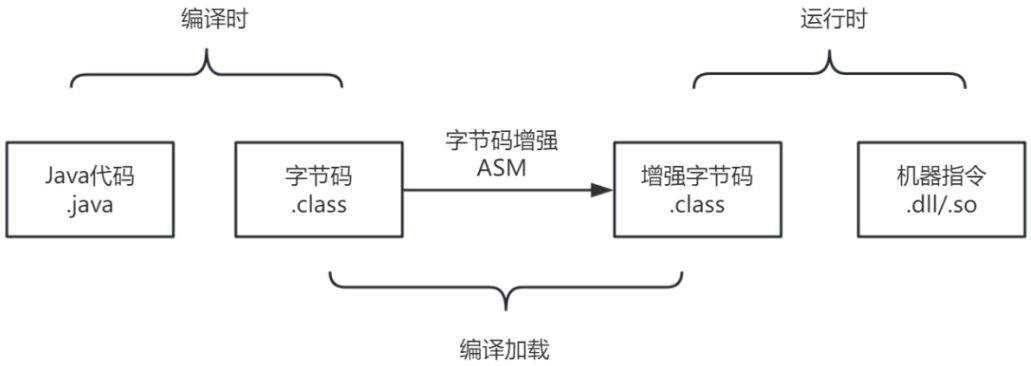


图 3 字节码增强技术

3.3 动态分析

动态分析是一种代码运行时的采集分析，主要方式是收集生产环境方法的执行次数，以确认方法是否有效。目前主要有两种做法，一种是通过打桩的方式，类似于半动态分析。该

方式虽然能够获取准确完整的运行时信息，但考虑到存在代码入侵并且可能对生产服务器性能产生影响，不建议采用这种方法。

另一种方法是利用 Java 虚拟机（JVM）的方法计数器，我们知道 JVM 采用的是 JIT（Just-In-Time）编译机制，方法执行过程如下图：

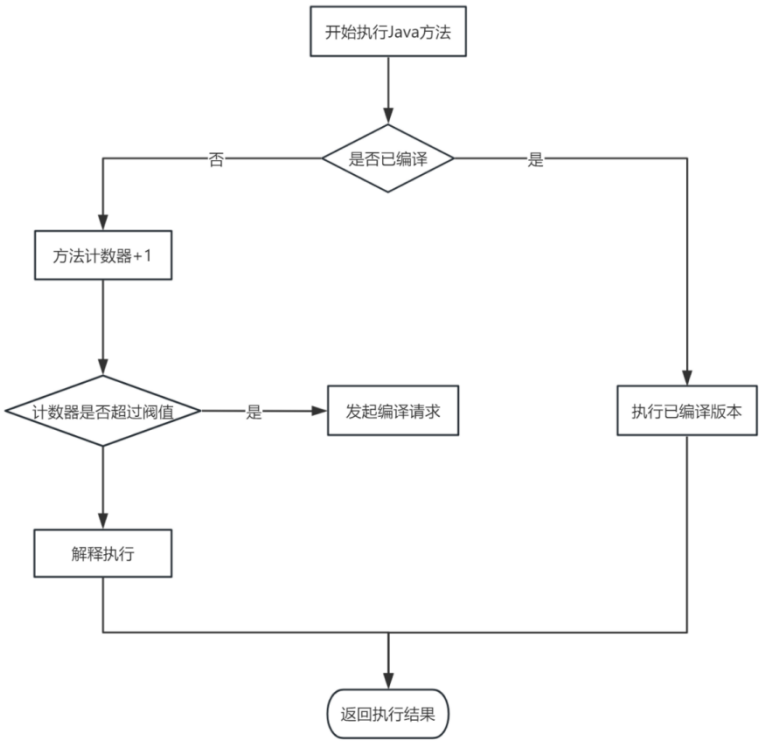


图 4 JVM-JIT 方法编译执行流程

这种方式对代码无入侵，缺点是访问 JVM 方法计数器需要 attach 虚拟机进程导致 STW（Stop The World），并且方法计数并不代表真正流量，只能反映方法有没有被执行以及执行的频度（幸运的是这对我们的场景已经足够了）。

综合考虑，推荐使用第二种方式，另外为了最大程度的降低采集流量期间 STW 对业务的影响，需要选取最适合采集的实例并提前停止对外服务（集群部署可以通过实例拉出实现）。

四、代码分析平台化

确定了代码分析平台化的目标，并阐述了代码分析的基本原理，接下来我们重点剖析平台化的三个关键步骤。

4.1 步骤一：建立知识库

建立知识库是代码分析平台化的基础，知识库可以将需求迭代的流程串连起来，并为后续分析数据（用例、流量等）的落地提供载体。

4.1.1 获取应用入口

应用入口指的是应用对外提供的服务，通常包括对外提供的 api、应用定时调度 job、消息（例如 qmq）的消费者；应用入口一般都是通过注解标记并自动注册上线，原理如图所示，运行时主动向注册中心注册实例和服务，被动接受调度和请求。

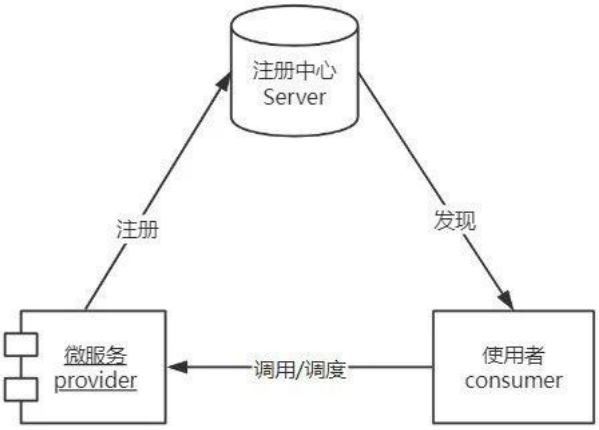


图 5 微服务注册发现流程

获取应用入口的最简单方式是通过代码分析根据注解识别。另外，多团队协作场景的 api 契约往往采用集中管理模式，应用通过第三方包引入 api 契约定义，为了避免大量的第三方引用解析，建议通过注册中心获取应用入口。

4.1.2 获取源代码

镜像指的是源代码经过编译、打包、检测验证后得到的容器加载对象，镜像是静态分析的主要输入。获取源代码则是为了得到准确的源码统计信息及变更信息。

考虑到开发人员在特定需求迭代过程中会多人协作、多次提交代码，因此获取源代码及镜像的时机建议在集群部署完成后、对外提供服务前，这样可以减少不必要分析、节约资源、简化分析流程以及减少对开发和测试的干扰。

4.1.3 静态分析及存储

通过静态分析可以得到方法间的调用关系，以及对方法进行标记（api、job、consumer、属性等）和染色（重写、继承、引用、可达等）。静态分析流程如下图所示：

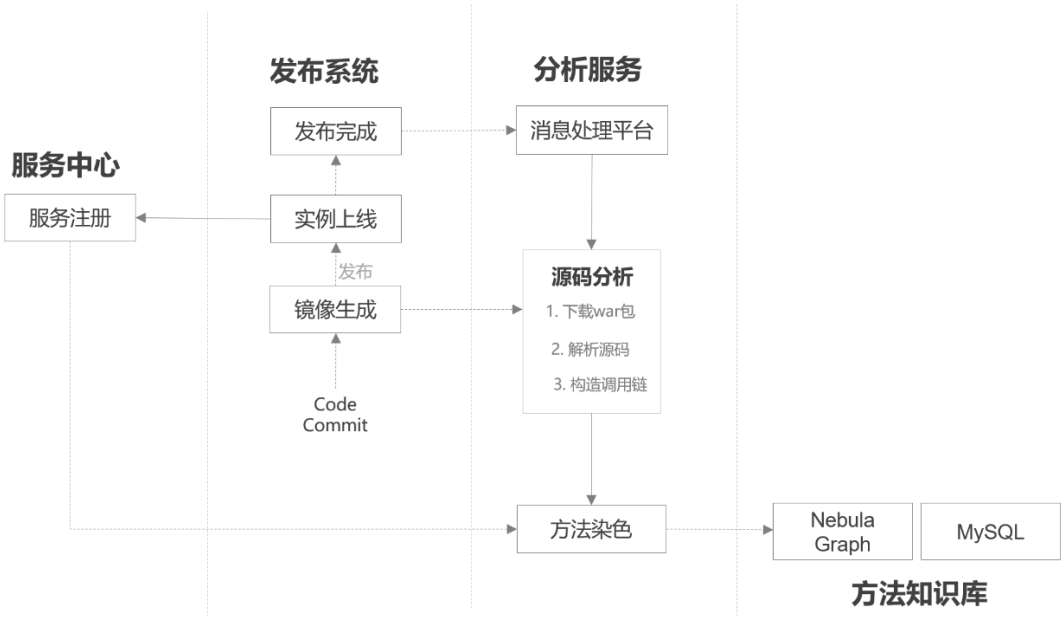


图 6 静态分析流程

实体数据建议使用关系数据库存储，考虑到方法间的调用关系复杂多变且层级深，推荐使用图数据库存储方法调用关系，不仅检索的复杂度更低、性能更好，而且能够比较直观的反映系统现状（通过 Nebula-Graph 存储并检索举例如下图）。

图 7 方法调用链路图展示

4.2 步骤二：完善知识库用例信息

在建立知识库的基础上，测试作为需求上线前的必备步骤，对测试用例的分析并融入知识库至关重要。这个步骤我们主要通过用例回放收集用例经过的内部方法和对外 api，结合源码对比得到的变更方法分析出需求改动直接、间接影响的入口和用例。

4.2.1 用例回放

用例回放指的是在用例执行的同时收集代码执行信息。执行用例回放需要满足两个前提条件，一是需要有一套自动化用例测试平台，能够维护并调度执行自动化用例；二是需要在

系统运行时进行打桩，能够在用例执行的过程中识别用例和方法调用信息，并对外输出。

大多数互联网企业都有自建的自动化测试平台，这里不做展开；系统运行时打桩的实现推荐使用开源 AREX，不需要修改业务代码，仅需系统镜像打包时加载代理服务，对系统运行时的影响安全可控。

4.2.2 分析流程

通过半动态分析（流程如下图），获取用例执行过程中途经的方法链路，补充知识库中通过用例建立起来的方法关联关系。

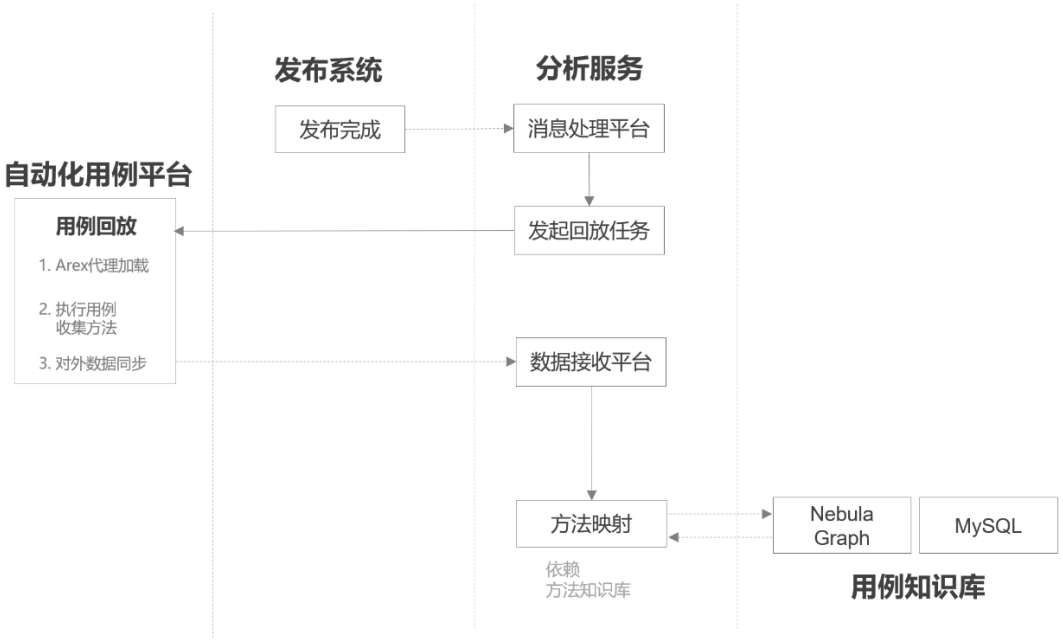


图 8 半动态分析流程

基于方法调用关系在图中的存储，用例和方法的关系也采用图数据库的存储，只需要再补充新类型的点（用例）和边（用例调用方法）即可，其表现方式更为直观（如下图）。

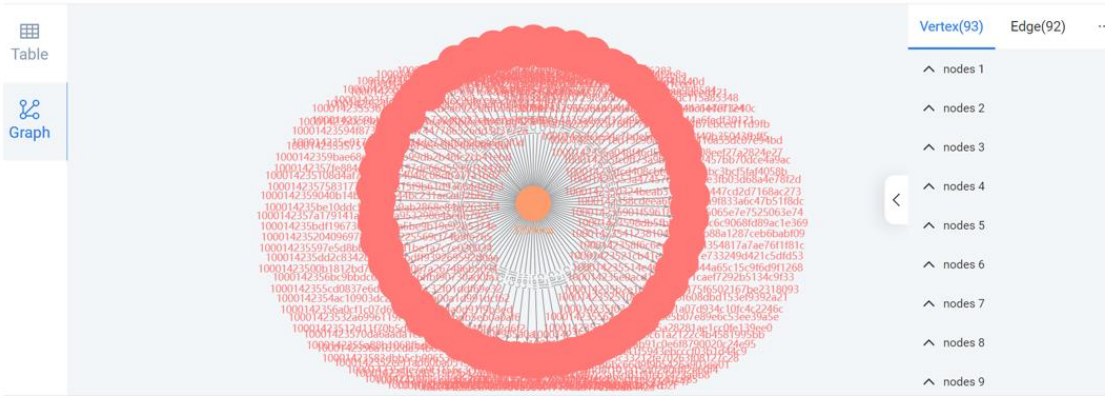


图 9 用例方法调用链图展示

4.3 步骤三：完善知识库流量信息

对源代码、用例的分析是建立在冷数据加载的基础上，应用代码的质量、测试的有效性最终体现在应用对外提供服务的过程中，运行时的数据不可或缺。这个环节我们主要介绍基于动态分析的原理，如何进行生产流量采集，如何将采集数据跟知识库结合起来，为后续的工具化和流程闭环提供数据支撑。

4.3.1 生产流量采集

生产流量采集主要包含两部分内容，入口流量采集和应用内部方法流量采集。

入口流量主要指 api（job 任务/消息处理）被外部调度的情况。作为日常排查问题和监控的重点，这部分数据通常作为微服务架构的基础能力，可以直接通过公共基础服务获取，这里不做展开。

应用内部方法流量采集的原理（动态分析）前面已经介绍过，这里重点介绍集群部署的场景下，采集实例选取的三个基本原则。

首先是保障采集对生产影响最小。主要基于采集需要暂停实例服务的考量，实例拉出前要做集群服务能力评估，确保服务能力不能下降过多（例如集群实例数少于 3 个的情况下不建议自动拉出），拉出后要给未完成的业务线程保留一定的处理时间，采集异常或者时间超过一定时长能够及时中断恢复拉入。另外针对 job 类应用建议 owner 选择合适时机手工采集。

其次是确保采集内容有效。方法流量采集本质上是 JVM 底层方法计数信息，因此如果实例创建时间过短（例如自动扩容）或者集群本身只针对特定场景服务（例如操作路由），很多场景都没有被执行到，采集的意义就不大。

最后是保障采集过程可持续。随着业务快速迭代，生产流量是不断变化的，因此流量采集需要周期性的持续进行。

4.3.2 分析流程

通过动态分析（流程如下图），将方法的流量信息补充到图数据库的点（方法）上，可以动态的反映方法被执行情况，间接的反映方法及自动化用例的有效性。

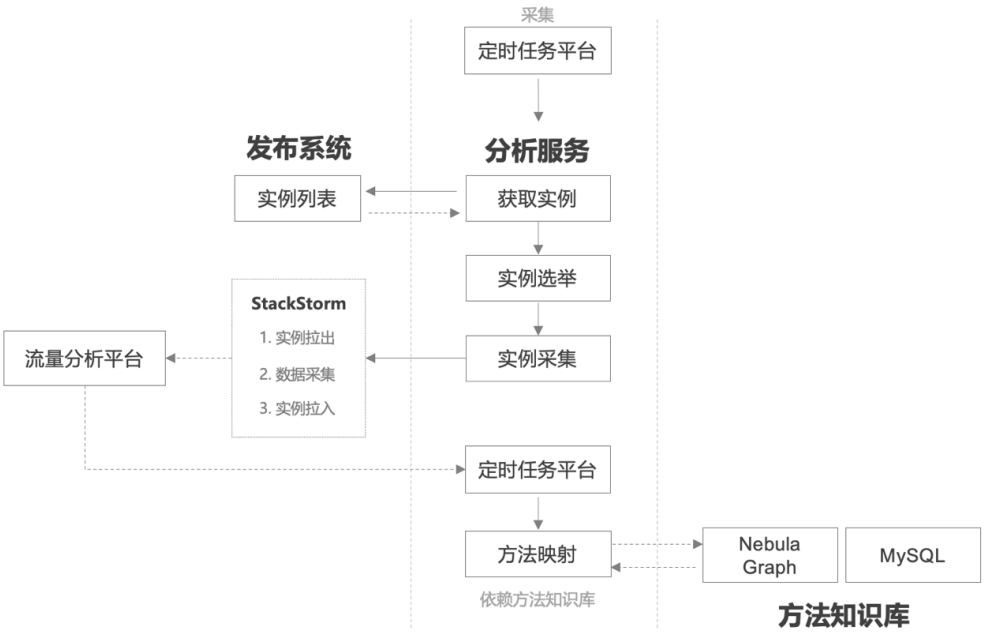


图 10 动态分析流程

五、应用场景

在知识库的基础上，结合精准测试和应用瘦身两个具体的应用场景，实现工具化和流程闭环，最终完成代码分析平台化建设。

5.1 精准测试

5.1.1 用例执行现状

自动化用例回归作为应用发布生产前的必经环节，有两项重要的评估标准：用例执行成功率和新增代码行的覆盖率。在没有用例推荐之前，一般采用人工选取执行和全量执行两种方式。

人工选取的优点是用例有针对性，缺点是不仅效率低而且容易漏选；全量执行虽然可以避免用例选取缺漏，同时可以提高增量代码行的覆盖率，但是用例执行成功率无法保障，往往需要对执行失败的任务一一排查，花费大量的时间和精力，大多数情况下失败的用例与迭代的需求并不相关，更令测试同学头疼的是随着需求迭代自动化用例在不断增加，费力度逐渐升高。

5.1.2 用例推荐

基于代码分析平台，应用生产发布前，可以通过对源代码进行对比分析获取变更的具体方法，获取变更通常需要代码版本管理工具和对比组件，目前互联网企业应用比较广泛的代码仓库管理工具是 git，对比组件推荐使用 code diff。

方法声明不变的变更（例如修改、删除），知识库中已经收集了方法调用链、用例方法

链，并且对方法进行了入口标记和染色，我们可以准确的识别其关联的入口及用例；方法声明变化的变更（例如新增、增加入参），我们利用实时静态分析可以通过调用链追溯到影响的入口，通过入口找到关联的用例。用例推荐功能如下图，每次只执行代码变更相关的用例。

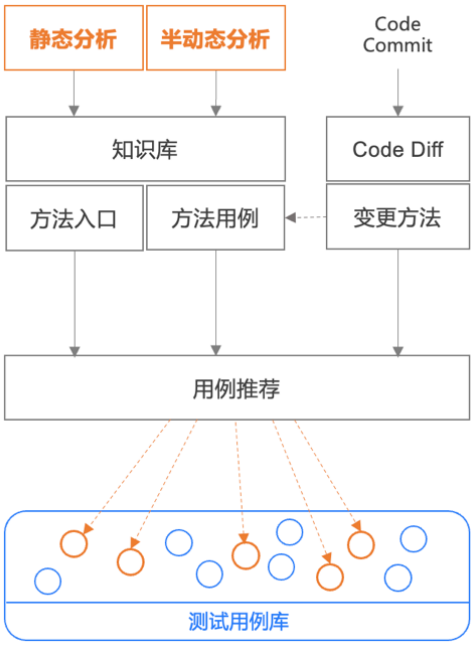


图 11 用例推荐流程

5.1.3 流程闭环

解决了如何准确的推荐用例，接下里需要结合需求迭代的基本流程和应用发布流程将代码变更、用例推荐、用例执行以及结果反馈串连起来（如下图），实现流程的闭环，才能更好的发挥代码分析平台的作用，最终提高需求迭代效率和质量。

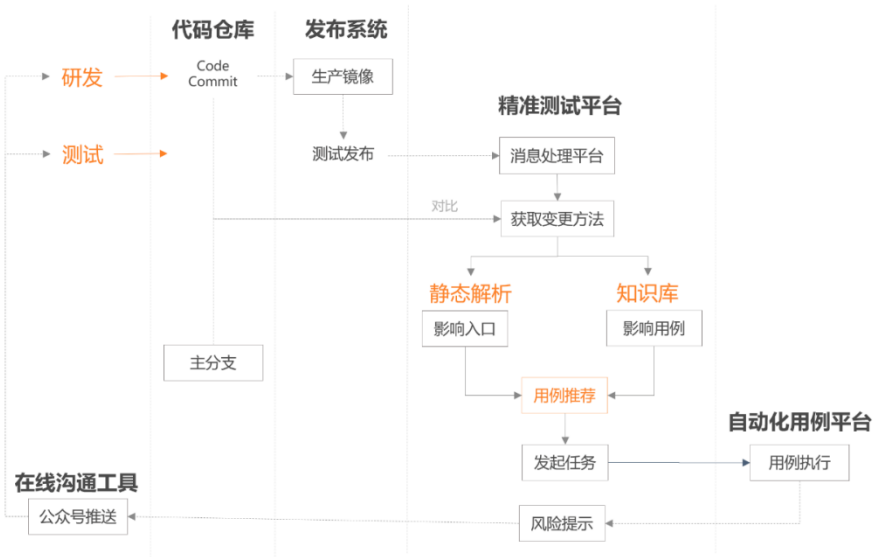


图 12 精准测试流程闭环

5.1.4 度量方法及效果

前面提到自动化用例执行效果的评估标准主要是用例执行成功率和增量行覆盖率，成功率主要靠用例本身的质量来保障，增量行覆盖率可以作为衡量精准测试准确性的度量方法之一。理论上精准测试的增量行覆盖率只要不能接近全量执行的增量行覆盖率，就说明推荐用例存在缺失。

经过验证，目前我们精准测试的增量行覆盖率可以达到全量执行的 99.2%（偏差主要来自于环境依赖），全面推广后平均减少了 68%与当次需求迭代无关的用例回归。随着自动化用例的不断增加，精准推荐已经成为自动化回归不可或缺的一环。

5.2 应用瘦身

5.2.1 应用代码现状

应用经历一段时间的需求迭代之后无效代码就会开始累积，究其原因主要有三个方面，一是需求变更后部分分支不会再被执行到，由于种种原因没有来得及重构；二是项目上线初期的临时检测对比的分支，项目上线后没有及时清除；三是上游场景变化导致下游部分场景不再被执行，但下游自身又无法识别。

过高比例的无效代码不仅影响系统的编译速度，而且严重影响开发的效率，尤其是对于新接手的同学，需要了解大量的代码历史背景才能熟悉系统现有的逻辑，而且代码量越大逻辑越复杂，修改的风险也就越高，即使经验丰富的工程师也不敢轻言重构。

5.2.2 工具化

基于代码分析的知识库信息，以方法为基本单位进行引用、入口以及生产流量的多维度分析（如下图），并提供应用分析工具，辅助开发人员快速的识别无效代码，实现应用瘦身。

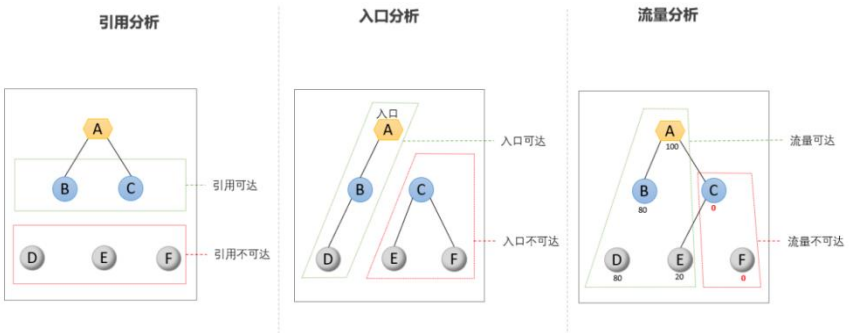


图 13 方法可达分析

5.2.3 流程闭环

从研发的角度看，删除代码存在一定的风险，如果能够便捷的通过工具获取代码的生产流量情况以及外部依赖情况，将极大的降低这种风险，增强其应用瘦身（包括代码重构和删除）的信心；从团队管理的角度来看，如何衡量治理的效果、把控治理过程的风险以及长远地评估无效代码的合理范围也同样重要，因此平台从研发和管理两个角度实现了闭环（如下图）。

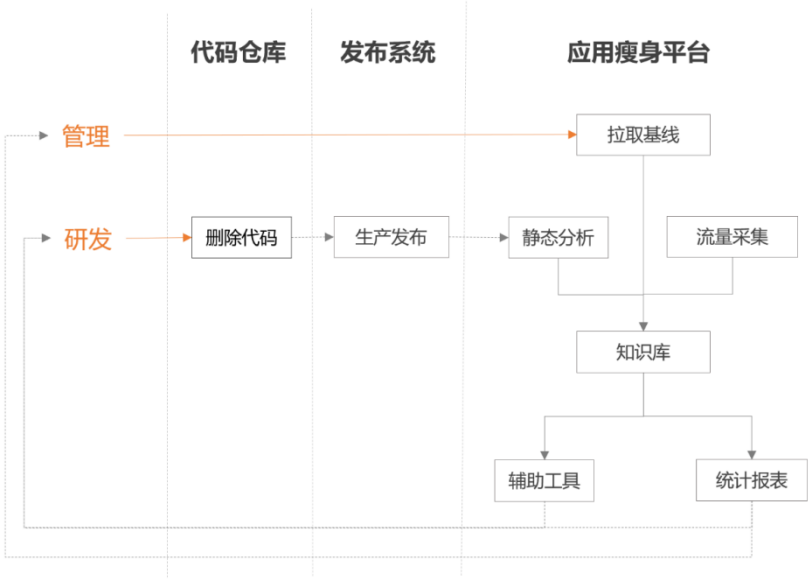


图 14 应用瘦身流程闭环

5.2.4 试点效果及经验

完成工具化和流程闭环，我们拿团队内部 10%的应用（100+）经过 1 个月的试点，轻松实现了零故障删除百万级代码行的目标。其中 15%的大规模应用（代码行大于 20 万行）经过瘦身后系统镜像的生成时长从几十分钟级降至到几分钟（耗时包括编译、UT 执行、合规扫描等）。代码链路复杂度也明显降低，如下图所示。

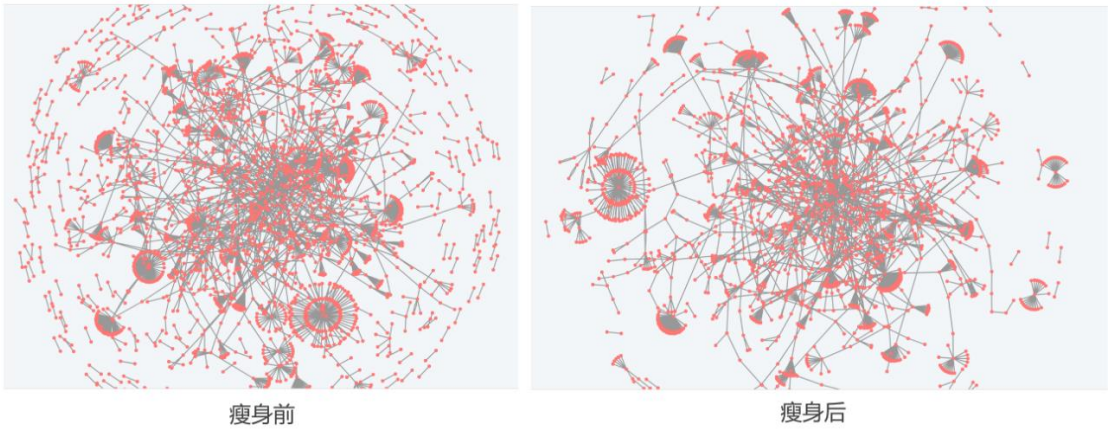


图 15 应用瘦身前后方法链对比

经过试点，我们总结了应用瘦身需要严格遵守的三个原则，一是瘦身工具只是辅助，代码删除一定要由对应用背景有一定了解的同学进行；二是删除代码一定要经过合作伙伴或 leader 的 review；三是应用瘦身是一个长期治理的过程，不能急于一时或一次了事。

六、总结

本文主要基于微服务架构下为了提高需求迭代效率，通过代码分析形成知识库，针对精准推荐和应用瘦身两个场景进行工具化和流程闭环，初步完成代码分析平台化建设。

目前已支持的场景还需要进一步细化（例如应用瘦身可以支持本地开发工具的插件化），结合当前的知识库，后续还可以支持更多的场景（例如工程复杂度、用例质量等等）。本文只是抛砖引玉，为应用治理提供了一种全新的思路，代码分析平台化是一个长期持续的工程，需要走的路还很长。

通过实时调试，让 AI 编写有效的 UI 自动化

【作者简介】Thales Fu，携程高级研发经理，致力于寻找更好的方法，结合 AI 和工程来解决现实中的问题。

引言

在快速迭代的软件开发周期中，用户界面（UI）的自动化测试已成为提高效率和确保产品质量的关键。然而，随着应用程序变得日益复杂，传统的 UI 自动化方法逐渐显露出局限性。AI 驱动的 UI 自动化出现了，但仍面临着准确性和可靠性的挑战。在这个背景下，本文提出一个创新的视角：通过实时调试技术，显著提升 AI 编写的 UI 自动化脚本的有效性。

这个问题不仅仅是技术上的挑战，它关系到如何在保证软件质量的同时加速软件的交付。本文将探讨实时调试如何帮助 AI 更准确地理解和执行 UI 测试脚本，以及这种方法如何能够为软件开发带来革命性的改变。

一、UI 自动化的现状

从最初的记录与回放工具到复杂的脚本编写框架，UI 自动化经历了显著的发展。然而，尽管技术进步，传统的 UI 自动化方法在应对快速变化的应用界面时仍然面临诸多挑战。

手动编写测试脚本不仅效率低下，而且在应用更新时需要大量的重新工作。据行业调查显示，UI 自动化测试脚本的维护可能占到整个测试工作的 60%至 70%。在一个典型的敏捷开发环境中，每次应用更新可能需要超过 100 小时来重新编写和测试现有的自动化脚本。这种高昂的维护成本凸显了传统 UI 自动化方法的低效性和资源消耗。

二、行为驱动开发 BDD 的引入

行为驱动开发（BDD）是一种敏捷软件开发的实践，它鼓励软件项目的开发者、测试人员和非技术利益相关者之间进行更有效的沟通。Cucumber 是实现 BDD 方法论的一个流行工具，它允许团队成员使用自然语言编写明确的、可执行的测试用例。

Cucumber 使用一种称为 Gherkin 的域特定语言（DSL），这种语言是高度可读的，使得非技术背景的人员也能理解测试的内容和目的。测试场景被写成一系列的 Given-When-Then 语句，描述了在特定条件下系统应该如何响应。

例如，一个在线购物网站的购物车功能可能有如下的 Gherkin 场景：

```
功能：购物车
  作为一个在线购物网站的用户
  我希望将商品添加到我的购物车中
  以便我可以随时查看我想购买的商品列表

  场景：将商品添加到购物车
    假如我已经将一本书添加到我的购物车
    当我查看我的购物车时
    那么我应该看到该书列在购物车中
```

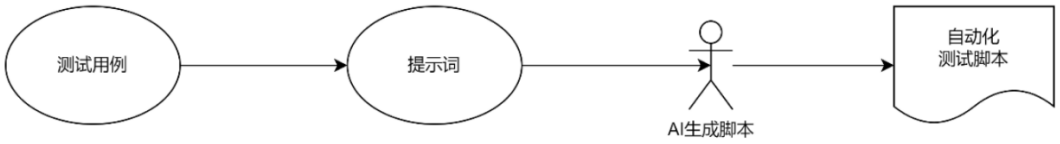
这种方法通过使用自然语言描述功能，帮助技术和非技术团队成员之间建立更好的理解和沟通。自然语言的测试场景也充当了项目文档，帮助新团队成员快速理解项目功能。让非技术人员可以直接参与测试用例的编写和验证过程，确保开发工作与业务需求紧密对齐。

但是它也存在着局限性，尽管测试场景用自然语言编写，每个步骤背后的实现（步骤定义）仍然需要技术人员使用编程语言来编写。这意味着实现测试逻辑可能涉及复杂的代码编写工作。随着应用程序的发展和变化，维护和更新与之相对应的测试步骤可能会变得繁琐。特别是在 UI 频繁更改的情况下，相关的步骤定义也需要相应地进行更新。还有灵活性和适应性限制：Cucumber 测试脚本依赖于预定义的步骤和结构，这可能限制测试的灵活性。对于一些复杂的测试场景，实现特定的测试逻辑可能需要创造性地规避框架的限制。

```java
public class ShoppingCartSteps {

    WebDriver driver; // 假设这里已经初始化了WebDriver

    @Given("用户在商品页面")
    public void user_is_on_the_product_page() {
        driver.get("http://example.com/product_page");
    }

    @When("用户点击“添加到购物车”按钮")
    public void user_clicks_the_add_to_cart_button() {
        WebElement addToCartButton = driver.findElement(By.id("add-to-cart-button"));
        addToCartButton.click();
    }

    @Then("该商品应该被添加到购物车")
    public void the_item_should_be_added_to_the_shopping_cart() {
        WebElement cart = driver.findElement(By.id("shopping-cart"));
        Assert.assertTrue(cart.getText().contains("1 item in your cart"));
    }
}
```

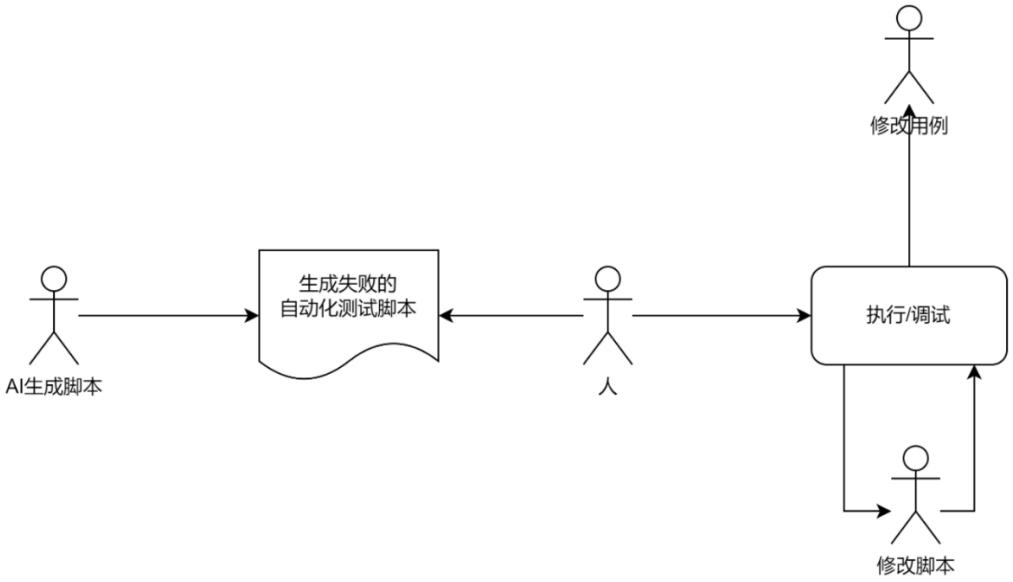
三、当前 AI 在 UI 自动化中的应用

近年来，AI 技术被集成到 UI 自动化中，特别是以 GPT 为代表的大模型出现后，因为它本身就有代码生成能力。业界也开始试着通过大模型来直接把 Gherkin 的测试用例描述语言生成成测试代码。



不过，当前大模型生成的测试代码并不能完全达到预期，主要有几个问题：首先，生成出来的脚本，因为语法错误可能无法运行；其次，也可能没有准确的覆盖到测试用例需要它去测试的校验点。在我们的实践下，真正能第一次就成功的比例不超过 5%。

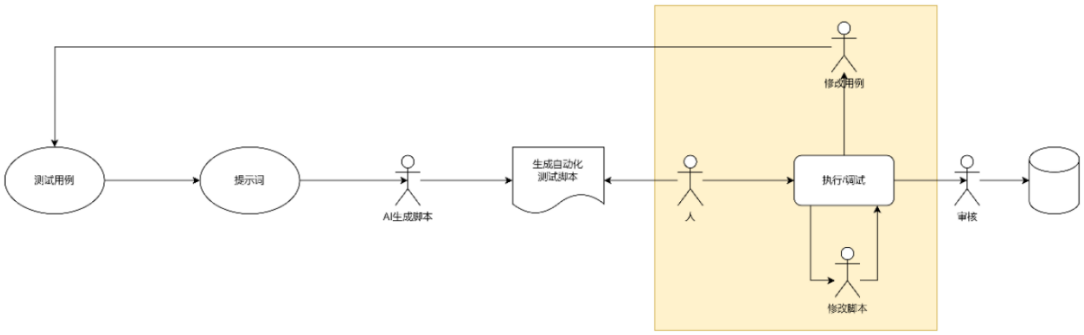
它生成失败后，接着就需要人介入再进行一些补救的工作。包括：调试，修改用例重新生成，或者直接修改生成的脚本。



而这些工作本身也需要消耗不少的人力，和我们系统通过 AI 来自动生成测试脚本的初衷相违背。

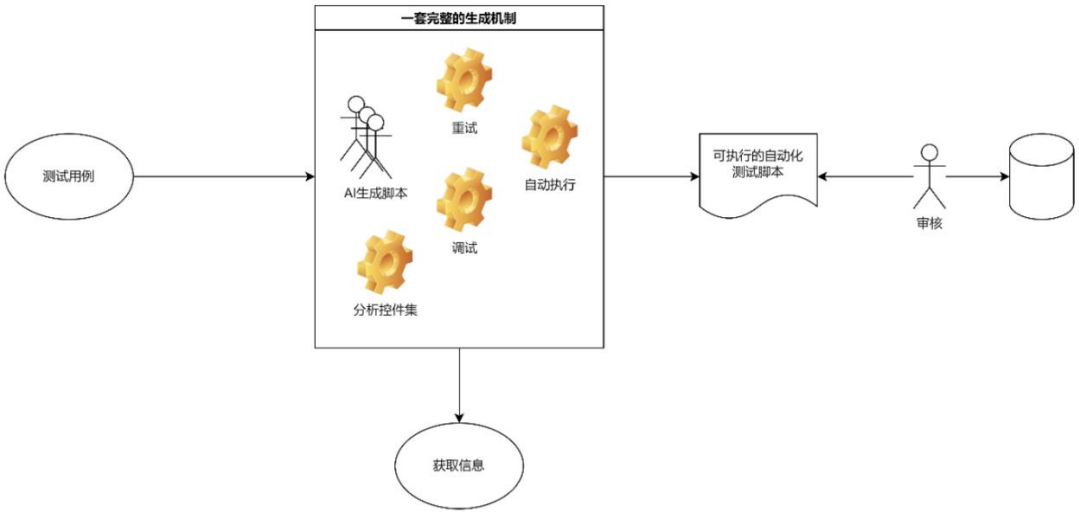
四、AI 全自动的来编写有效的测试脚本

为了解决这个问题，我们重新思考了 AI 生成测试脚本的整个过程。



我们把人的工作也放在里面一起考虑。人在系统中做了调试和修改的工作，那这部分工作是不是可以让 AI 来做呢，让系统自己运行生成的代码，让 AI 来调试和修改自己生成的错误代码。

因此，我们调整了系统设计，让 AI 代替人自主地来做这些工作。最终，对于携程酒店订单详情页的全部用例，在无人参与的情况下，生成可以执行成功的占全部的 83.3%，在生成脚本过程中，有 8%的 case 就已经发现了 Bug。我们连续生成这些用例三次，成功率分别在 84.3%，81.4%和 83.3%，系统是稳定有效的。



具体的测试用例和代码如下：

BDD描述

```
Given订单用户是黑钻用户
When 用户滑动到"用户权益"模块
And 点击"订房优惠" 进入价格明细浮层
then 页面显示 黑钻贵宾 文案
```

首先，需要滑动到订单详情页下放的用户权益模块，然后点击订房优化区域，来弹出价格浮层。

然后再看，费用明细里面是否包含黑钻贵宾。

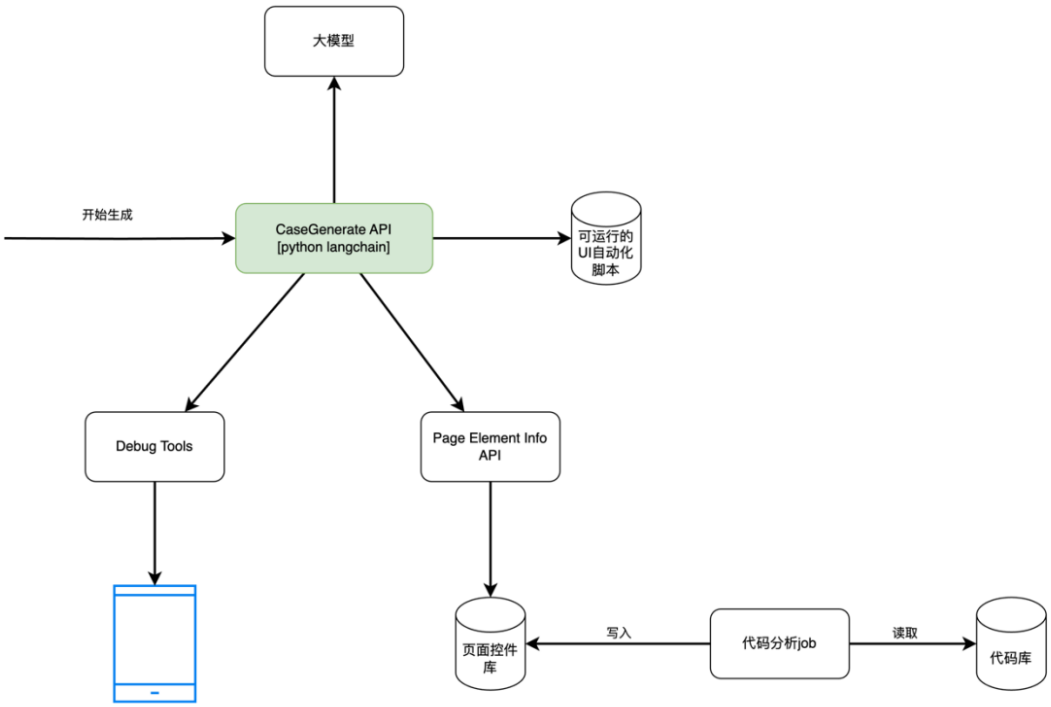
最终生成的测试代码如下：

```
def runTest(self):
    hotelDetailUICaseManager = HotelDetailUICaseManager(self)
    hotelDetailUICaseManager.commonCheck.waitPageLoad()
    # 滑动屏幕找到"用户权益"模块
    root_element = self.look_for_element(CtripMemberAwardModuleElements.ctrip_reward_module, endpoint='')

    # 查找"订房优惠"控件
    element = self.findAnyElement(CtripMemberAwardModuleElements.book_discount)

    # 点击"订房优惠"控件，进入价格明细浮层
    self.click(element)

    # 断言页面上存在"黑钻贵宾"文案
    self.assert_exist('黑钻贵宾')
```

五、系统实现

整个系统的核心架构示意图如下。系统的核心部分是一个 langchain 框架的程序。它会去访问大模型，我们给它配备了多个工具，主要分成两类，一类是页面信息的获取工具，一类是调试工具。

Langchain 会自动根据需要，使用页面信息获取工具，去拿页面的数据，来判断当前的操作需要具体哪个控件，来生成代码。然后再使用调试工具在手机中真实的执行代码，基于调试的反馈来判断自己生成的代码是否正确。

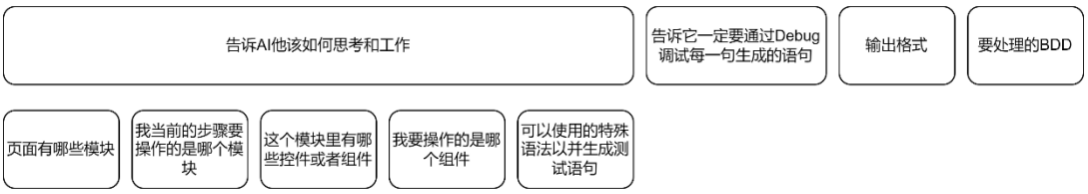


5.1 提示词

有了基本的架构后，我们需要提示词，来把这些工具粘合起来，让 AI 理解它该如何

工作。我们的提示词从结构上来说包含了几部分内容：首先告诉 AI 它该如何思考和工作，其次告诉它一定要通过 Debug 调试它每一句生成的语句，再次告诉它输出格式是什么，最后是告诉 AI 要处理的完整用例文本。

对于告诉 AI 它该如何思考和工作，展开包含以下部分：首先看页面有哪些模块，我要操作的这个步骤应该是哪个模块，这个模块里有哪些控件和组件，我当前要操作的是哪个控件或组件，我要操作的动作是什么，以及我可以用的特殊的语法是什么，然后生成语句。



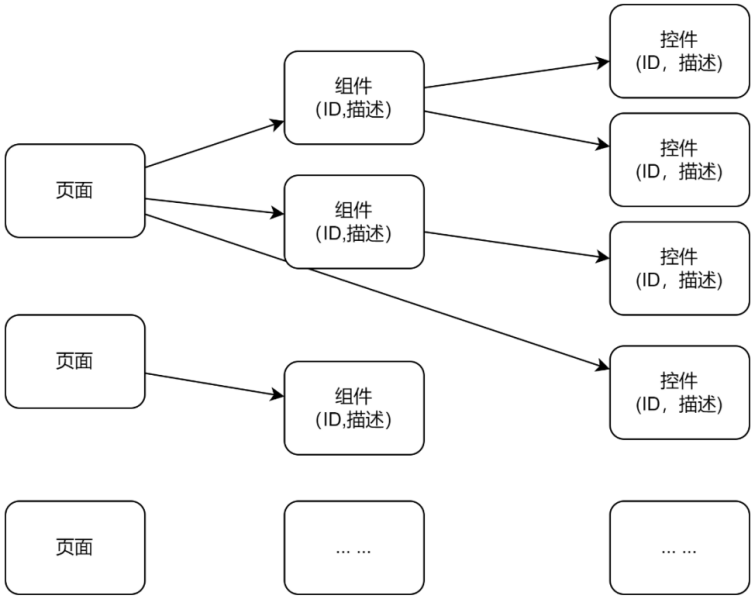
5.2 调试工具

调试工具的本质是通过 adb 工具远程连接到手机上。连接后，我们就可以把 AI 生成的指令发送给手机去运行，并且读取到运行后的结果给到 AI，让 AI 去判断自己生成的指令是否正确。

5.3 页面信息获取工具

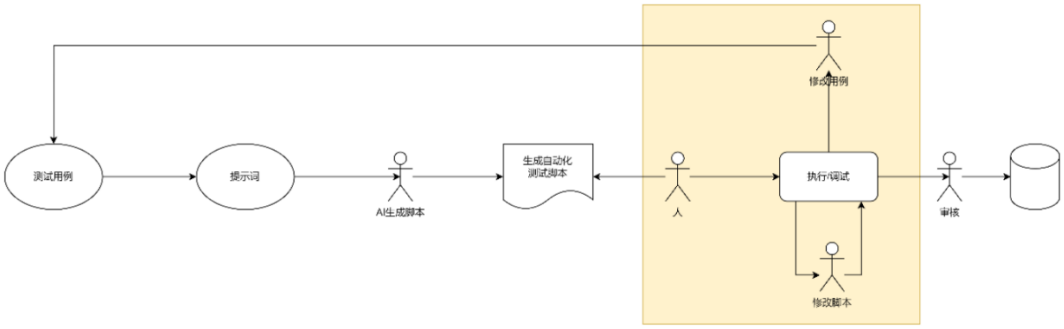
页面信息获取工具的最终目的是帮助 AI 判断出，BDD 的用例上面写得要操作的内容，它具体要操作的控件的 ID 是什么，有了 ID 才能基于 ID 生成后续的程序指令。而为了拿到 ID，我们需要有个控件和组件库，这个库里面的核心是每个控件和组件的 ID 以及它们的描述。有了这两项内容后，才能帮助 AI 看了 BDD 用例后，基于控件的描述去猜需要的是哪个控件。

为了达到这个目的，我们建立了一个页面控件库。这个库除了包含页面上每个控件的 ID 和描述外，还包含了页面和组件的关系，以及组件和控件的关系。能方便 AI 一步步的进行查询。



而这个控件库本身是基于我们通过 job 对代码进行静态分析来生成的。不过实际应用中，因为页面当前真正展示的控件会根据场景状态的不同而不同，在某些场景下页面上的控件会隐藏。因此页面信息获取工具会把页面当前真实存在的控件和控件库中查询出来的控件做交集，从而获取到当前页面真实展示出的控件和它的描述信息。

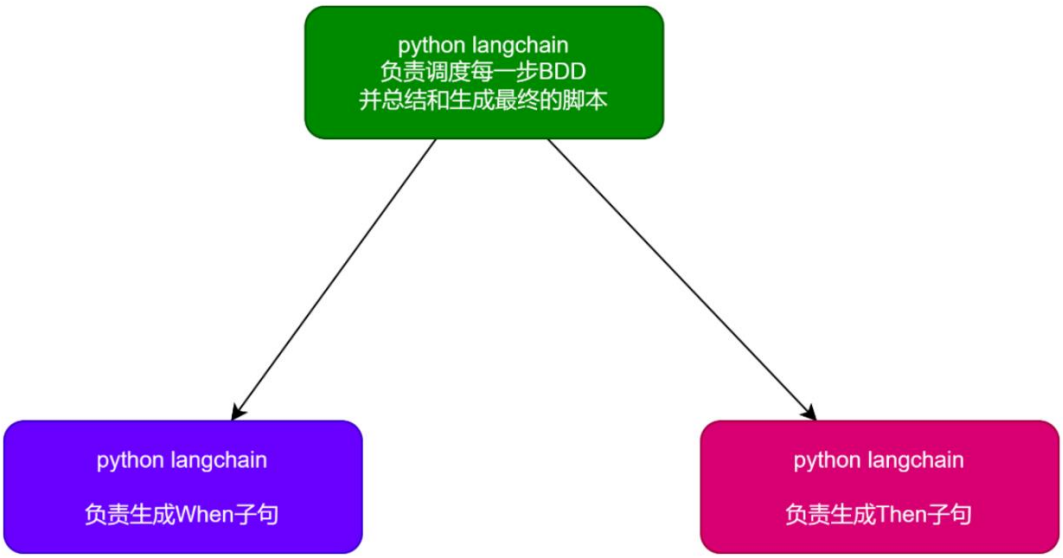
5.4 进一步拆分 AI



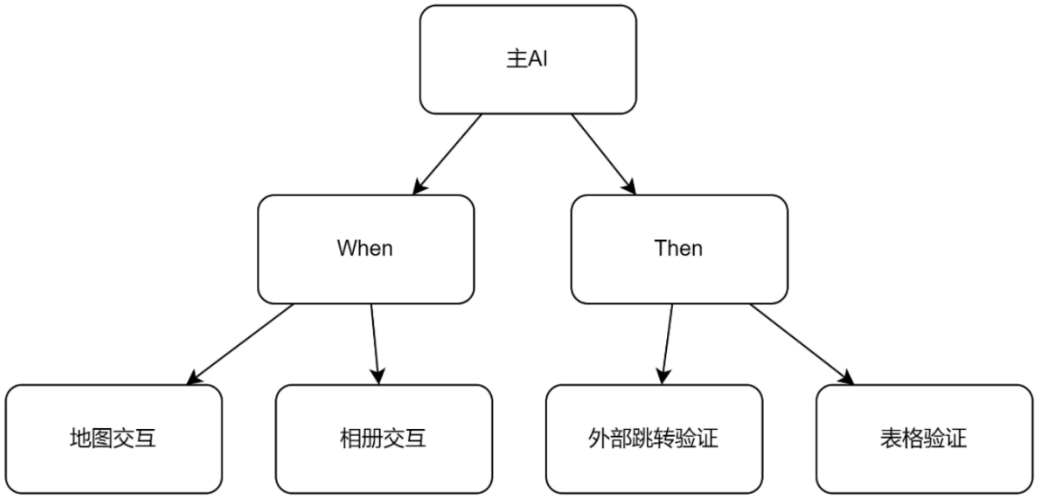
当做了这些工作后，AI 基本上已经可以把上面这张图黄色的部分，也就是人的工作自动去做了。生成成功率也从 5%提升到了 55%，但是 55%的成功率还是不够的。

我们进一步分析了失败的 case。发现主要问题是 AI 的幻觉，虽然提示词已经比较详细了，但是 AI 有时会没有按照要求处理，有的时候会自己胡说八道。

我们的结论是，给 AI 的责任太多了，它要考虑的东西太多。倒不是说它的 Token 不够，而是让它做的事情太多，会遗忘，无法精准完成要求。因此我们考虑进行拆分，还是利用了 langchain 的 function 的功能，既然 AI 能通过工具去完成功能，那这个工具为什么本身不能也是个 AI 呢。



甚至还可以把它再进行拆分。



通过这些拆分，我们让每一个 AI 需要考虑的工作变得更少更简单，也让它处理得更加精准，最终生成成功率提升到了 80%以上。

六、后续的发展

当前，通过我们的工作，能让 AI 在无人参与下以 80%左右的成功率去生成自动化测试的代码，很让人振奋，但还有很多问题需要继续去解决。

1）大模型的调用成本还是不低，是否有更好的办法，更低的成本去完成工作。

2）当前还有些比较难处理的操作或者校验，成功率 80%还有不小的提升空间，以及目前最后还是需要人来复核生成结果。

3）除此之外，其他方面也都有提高的空间，值得我们继续去完善。

流量回放平台 AREX 在携程的大规模落地实践

【作者简介】携程 AREX 团队，机票质量工程组，主要负责开发自动化测试工具和技术，以提升质量和能效。

导语

AREX 是一款由携程开源的流量回放平台，孵化于机票 BU 内部。聚焦录制回放核心链路的建设，从基础方案建设到核心事业线的深入落地验证，在集团复杂业务场景下不断迭代和优化下，积累了大量经验，取得了可见的成果。在携程落地至今已有 4000+ 应用接入，交付率和缺陷数均有所改善。

本篇文章主要介绍 AREX 在携程内部落地实践过程中遇到的一系列挑战和解决方案，以及如何通过 AREX 快速部署一站式流量录制回放解决方案来降低接入成本，快速落地。

一、背景

流量录制回放技术在性能测试、回归测试、自动化测试以及线上问题快速修复方面有广泛的应用前景，可以帮助技术团队解决复杂业务场景和系统架构下的稳定性保障及研发过程中的效率问题。

然而在技术方案落地时，会面临很多的挑战，比如基础设施建设难度大、前期投入成本和收益不成正比、落地场景模糊不清等。

二、方案

目前市场上已知的开源解决方案大部分都是在 Jvm-Sandbox-Repeater 基础上进行二次开发和改造，核心原理也都是通过录制线上真实流量然后在测试环境进行回放，验证代码逻辑正确性。那么可能有人会问：既然已有成熟的解决方案，为什么还要“重复造轮子”?

首先，JVM SandBox 支持的组件有限，远不能满足携程内部广泛使用的中间件和框架。且 JDK 底层的支持也不够彻底，比如异步线程上下文传递，需要依赖其他第三方组件。

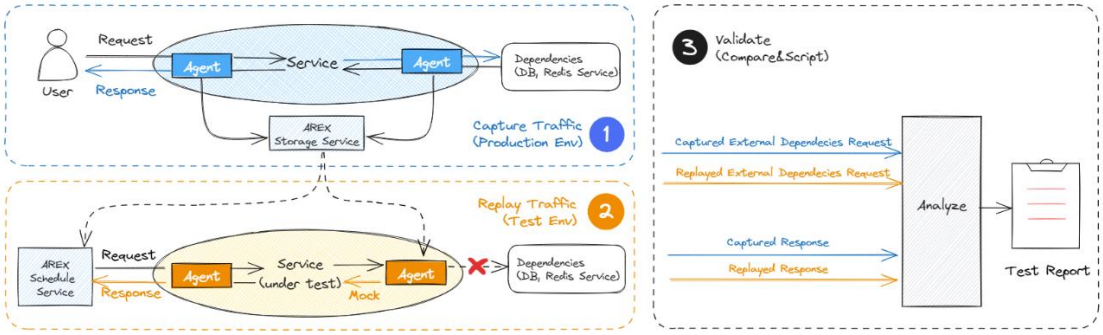
其次，Jvm-Sandbox-Repeater 虽然提供了基本的录制和回放功能，但若要构建一个完整的业务回归测试解决方案，我们还需要一个完善的后台支持系统，负责数据的采集、存储和比对工作。

最后，官方文档的缺乏以及社区活跃度的不足，使得我们在后续的二次开发过程中可能面临无法及时获得官方支持的风险。

基于这些考虑，我们决定自主研发流量录制回放平台 AREX:

1）支持更广泛的中间件和组件录制与回放，而且能够模拟各种复杂的业务场景，如本地缓存、当前时间等。

2）作为一个全面的解决方案，还要配备有完善的配套设施，如前端界面、回放服务和报告分析等，实现从流量采集、流量回放，到比对验证、生成报告的一站式工作流程（如下图所示）。



下面，我们将深入探讨实施过程中遭遇的挑战、针对性的解决策略，以及携程内部的应用实例，希望可以为大家的实践提供实质性的帮助和指导。

三、技术挑战

3.1 跨线程、异步场景下的流量采集

流量录制需要把一次业务请求里涉及到的所有链路节点采集下来，不仅是主入口的，还有内部调用各种框架的请求和响应，如 Mybatis、Redis、Dubbo 等。然而公司很多项目会使用到线程池，异步编程的场景，比如在一次请求中主流程会 Fork 出很多子任务/线程并行工作，有些任务查询 Redis，有些会调用 RPC 接口、有些去操作数据库等完成不同的业务场景，底层也会牵涉到大量线程的切换。

这样就需要保证在一次请求中把这些在不同线程里执行的操作都采集下来，我们是通过 Trace 传递的思路解决这个问题的，即通过修饰各种线程池和异步框架，使用一个 recordId 在线程间传递的方式串联起来，完成一次完整的用例录制。比如 Java 里的 CompletableFuture、ThreadPoolExecutor、ForkJoinPool、第三方的 Tomcat、Jetty、Netty 使用的线程池，以及异步框架 Reactor、RXJava 等，实现不同线程间的传递。

3.2 非幂等接口回放不产生脏数据

例如，在订单落库和调用第三方支付接口等关键场景中，流量回放时需确保利用 Mock 来避免实际数据交互。这样做可以防止在测试过程中产生不必要的数据，从而避免对正常业务流程造成干扰。流量回放的核心机制在于拦截并 Mock 框架调用，使用录制的数据来替代真实的数据请求，确保测试过程中不会发生任何真实的外部交互，如数

据库写入操作或第三方服务调用，从而有效防止回放测试中脏数据的写入。

目前我们的 Java Agent 已支持 Spring、Dubbo、Redis、Mybatis 等开源框架，完整列表请参考下方。

Foundation	Cache Library	Spring	Http Client	Redis Client	Netty
Java Executors	Caffeine Cache	Spring Boot [1.4+, 2.x+]	Apache HttpClient [4.0,)	Jedis [2.10+, 4+]	Netty server [3.x, 4.x]
System time	Guava Cache	Servlet API 3+、5+	OkHttp [3.0, 4.11]	Redisson [3.0,)	
Dynamic Type	Spring Cache		Spring WebClient [5.0,)	Lettuce [5.x, 6.x]	
			Spring Template		
			Feign [9.0,)		
Persistence framework	**NoSQL**	**RPC**	**Auth**	**Config**	
MyBatis 3.x	MongoDB [3.x, 4.x]	Apache Dubbo [2.x, 3.x]	Spring Security 5.x	Apollo Config [1.x, 2.x]	
TkMyBatis		Alibaba Dubbo 2.x	Apache Shiro 1.x		
MyBatis-Plus			JCasbin 1.x		
Hibernate 5.x			Auth0 jwt 3.x		
			JWTK jjwt 0.1+、jjwt-api 0.10+		

3.3 因登录鉴权、token 过期问题引起的回放失败

在实际的流量回放过程中，我们经常遇到这样的问题：许多 Web 应用在接口访问前实施了登录鉴权校验。如果鉴权失败或登录的 token 已经过期，接口访问将被拒绝，这可能导致大量用例在回放时失败。虽然可以通过配置白名单来解决部分问题，但我们寻求的是一种更为通用的解决方案。

理想的方案是在回放过程中，能够 Mock 如 Spring Security、Apache Shiro、JWT 等鉴权框架，从而绕过鉴权和 token 校验步骤，确保接口能够在回放环境中正常执行。

3.4 时间敏感业务，如支付超时场景回放

如果录制时的当前时间和回放时的当前时间不一致，可能会导致一些超时判断逻辑出现预期外的差异。例如，在判断订单是否超时未支付的场景中，我们通常会使用 currentTime - orderCreateTime > 30 分钟 作为判断依据。如果在录制时订单尚未超时，但在半小时后进行回放时，由于系统当前时间的变化，可能会错误地触发支付超时的处理逻辑。

为了解决这一问题，我们提出了一种解决方案：在录制过程中，同时记录下当时的当前时间，并仅录制一次。在回放过程中，通过 Mock 与当前时间相关的类，如 Date、Calendar、LocalTime、joda.time 等，使得回放时使用的当前时间实际上是录制时记录的时间。这样可以保证在回放过程中，与时间相关的逻辑判断能够与录制时保持一致，从而确保测试结果的准确性和可靠性。

3.5 本地缓存问题

在应用中，为了提高性能，通常会将一些常用数据存储在本地缓存中以供快速访问。然而，在流量录制回放的场景中，缓存的行为可能会对回放结果产生影响。

在录制过程中，如果请求的数据已经被缓存，那么系统会直接从缓存中提供数据，而不会触发对数据库或外部接口的查询。但在回放环境中，由于缺乏预先加载的缓存数据，相同的请求可能会导致应用程序去查询数据库或调用外部接口，产生新的调用（new call），导致回放失败。

为了解决这一问题，我们实现了对流行缓存框架的支持，如 Guava Cache 和 Caffeine Cache，确保在回放时能够模拟缓存的行为并保持一致性。这样一来，在回放过程中，即使是对缓存的请求也能按照录制时的状态返回预期的结果，避免了不必要的新调用。

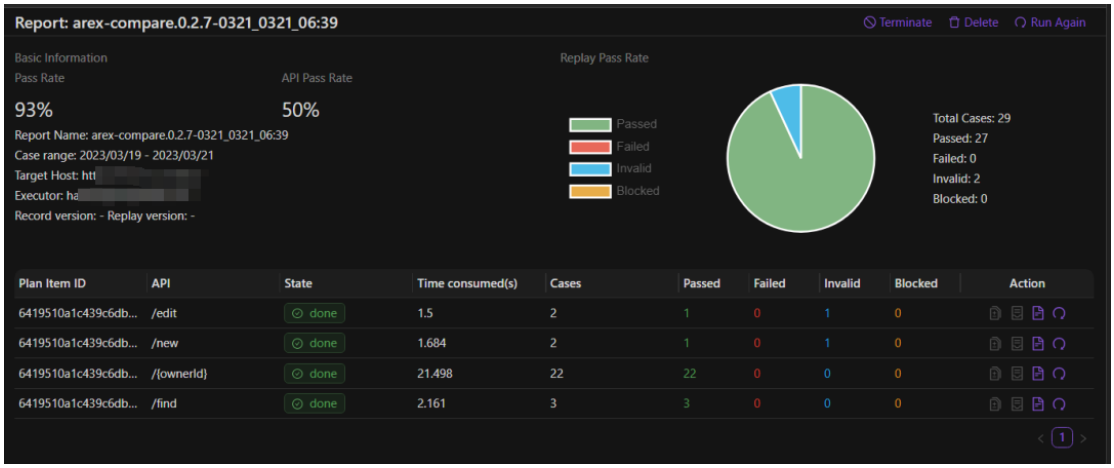
对于那些使用自定义缓存框架的情况，AREX 平台提供了灵活的配置选项，允许通过动态类的方式进行适配。这意味着即使是非标准的缓存实现，也能够被 AREX 平台兼容并正确地进行流量回放。

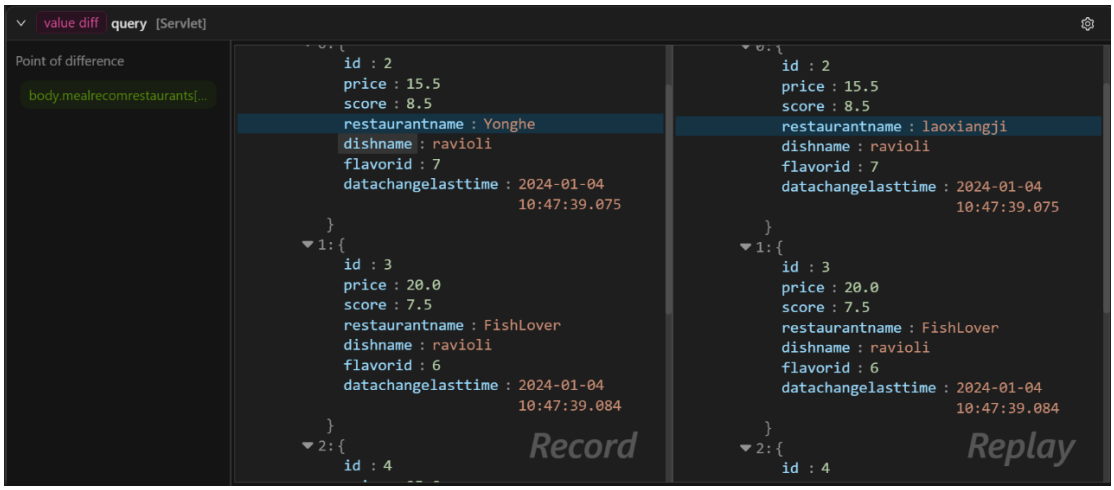
以上解决方案都是默认支持，基本不需要额外处理，另外如果是公司内部研发的框架也需要录制回放的话，可以以插件的方式扩展。

四、落地挑战

4.1 安装部署要做到简单便捷，快速上手，减少接入成本

AREX 是一套完整的解决方案，除基本的录制回放功能外，还有前端、调度、报告分析、存储等配套服务。本着开箱即用、快速接入的原则，AREX 提供了多种部署方式：一键部署、非容器部署、私有云部署的方式，安装完成后只需配置一些基础参数即可自动采集流量和进行回放对比差异：

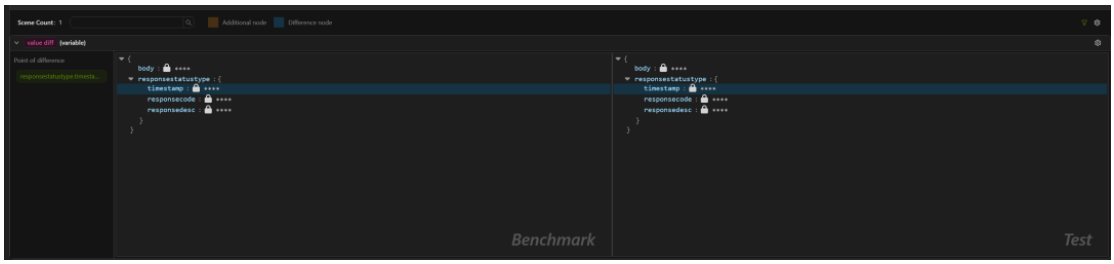




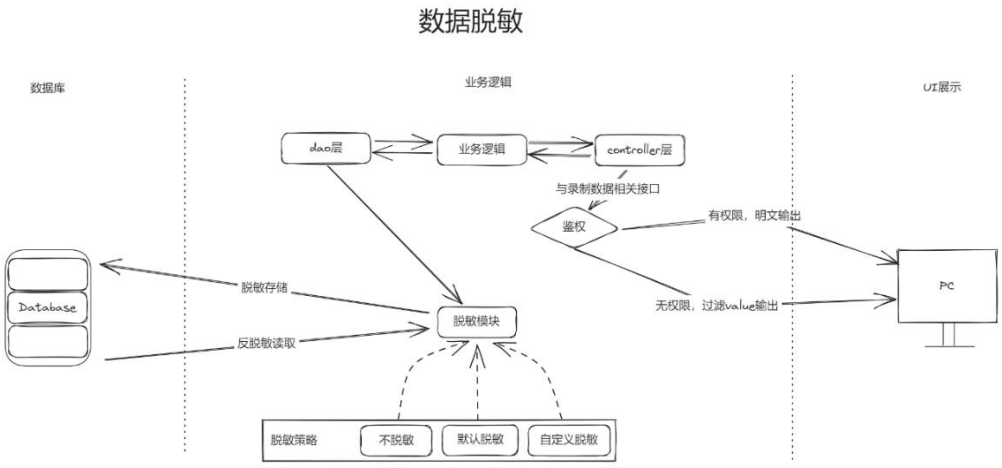
此外 AREX 还支持单机模式，可以在本地不需要安装的情况下快速上手体验。

4.2 符合公司风控、数据安全要求

录制生产上真实流量时，在涉及安全或者一些商业性敏感数据的情况下，还需要针对某些敏感信息通过脱敏规则进行数据的变形，实现敏感隐私数据的可靠保护。

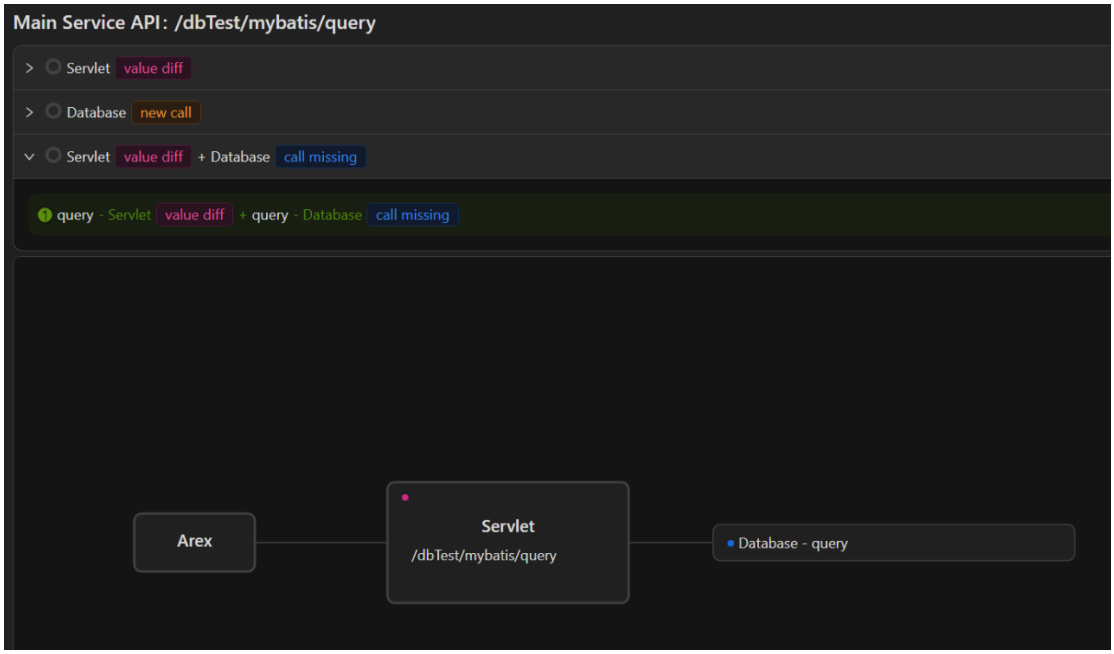


AREX 选择在进行数据落库时对数据进行脱敏，以保护敏感信息的安全性。具体实现方式是通过 SPI 机制，加载外挂 JAR 包，动态加载加密方式。



4.3 提高用户体验，快速定位问题

在实际使用过程中，录制和回放的用例数量巨大，为了减轻使用者分析差异时的工作量，AREX 对存在相同差异的场景用例进行了聚合，加快排查问题的速度。



通过调用链可以快速定位问题所在范围，并且对时间戳、uuid、ip 等噪音节点进行降噪，减少干扰。

如果是一些业务比较复杂的应用线上问题本地难以复现时，AREX 也支持在本地进行调试快速排查问题。

4.4 技术方案是否成熟、安全、可靠

AREX 基于 Java Agent 技术，采用业界成熟的字节码修饰框架 ByteBuddy，安全稳定，代码隔离，带有自我保护机制，在系统繁忙时会智能降低或关闭数据采集频率。且在携程集团内部已稳定运行 2 年有余，线上得到充分验证。

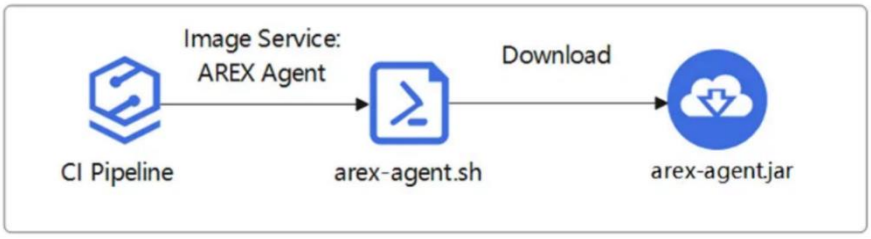
五、最佳实践

目前流量录制回放服务作为独立的选项集成到公司的 CI/CD 系统中：

1）首次接入流程：在首次接入流量录制回放时，只需在 CI Pipeline 选择 Flight AREX Agent 服务，这样在应用打包成镜像的过程中，会把 AREX 启动脚本 arex-agent.sh 包含在发布包内。

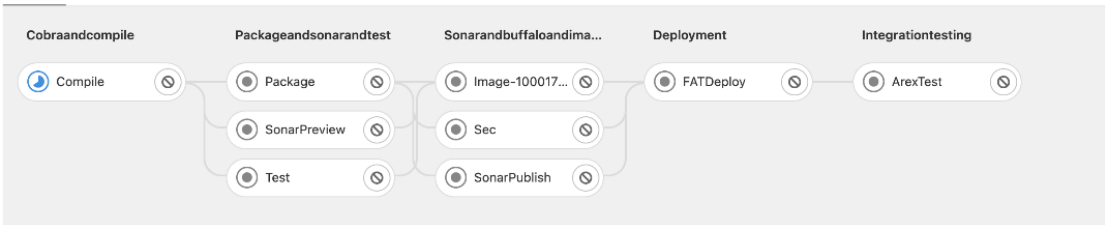
2）发布与 Agent 加载：在应用发布过程中，先前的脚本启动后会拉取最新的 arex-agent.jar，并通过修改 JVM Options 挂载 AREX Agent（-javaagent:/arex-agent.jar）。

3）版本控制与灰度发布：启动脚本后会根据应用的 AppId 拉取与之匹配的 arex-agent.jar 版本，实现灰度发布和按需加载，比如只有某些特定的应用会加载 Agent 新功能。



同样，如果是首次回放，操作也很简单：

1）创建 Pipeline：在 Gitlab 或 Jenkins 中，创建一个 Pipeline，在 ArexTest Job 脚本中调用 AREX 提供的回放地址，定时执行流水线。

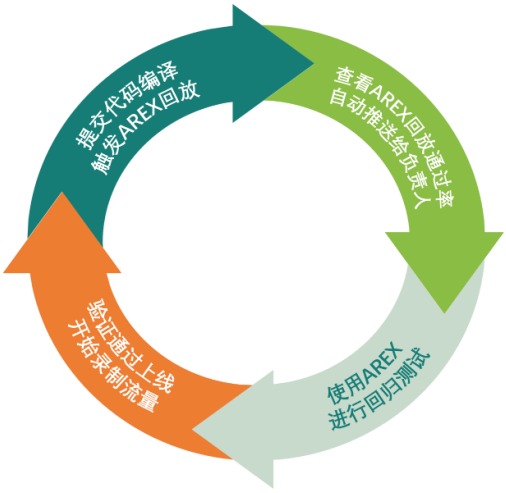


2）自动触发流量回放：研发人员在提交代码后会自动触发流量回放。

3）回放结果推送与发布控制：回放完成后 AREX 会把回放用例数、通过率、失败率等指标推送给相关人员做统计和分析，只有当通过率达到预定标准时，代码才被允许发布到生产环境。

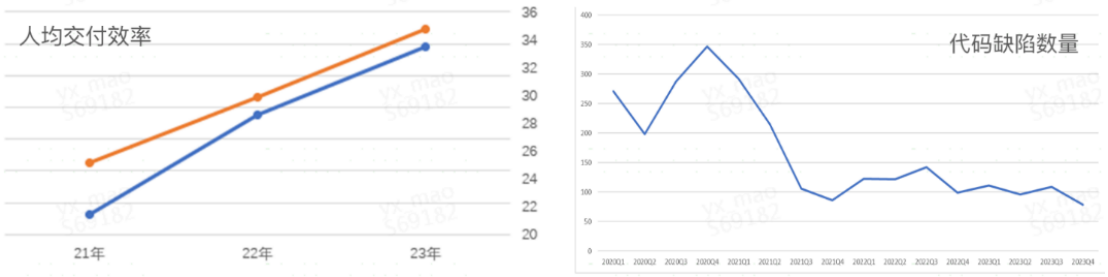
下图是 AREX 流量录制回放平台在公司研发测试发布各个环节如何发挥作用的，供大家参考：

针对每次迭代，代码提交后测试自动执行，并反馈测试报告，开发和测试人员只需要关注在新业务的研发、验证上即可，脱离那些繁琐的数据和脚本，通过流量回放在软件研发全生命周期内进行多环节针对性优化、合力赋能，形成一个自动化测试和持续集成的闭环。



六、落地成果

在携程集团复杂业务场景不断迭代和优化下，目前已有 4000+ 应用接入，交付率和缺陷数均有所改善：



七、拥抱开源

在携程内部经过长期稳定运行并验证其可靠性后，我们在 2023 年将 AREX 平台开源（https://github.com/arextest），希望能够帮助更多企业高效、低成本地把流量录制回放技术解决方案落地。

过去一年，我们致力于开源社区的建设，目前已有上千个外部用户接入使用 AREX。

AREX 的愿景是是在需求快速迭代的同时保障质量，降低成本，提升效能。这一愿景已在携程及众多开源用户的实践中得到验证，带来了显著的业务价值。

展望未来，我们将持续依托活跃的社区力量，响应并解决用户的疑问，不断优化 AREX。在此诚邀每一位开发者加入社区并试用，共同见证 AREX 的成长与进步。

准确率 89%，携程酒店大前端智能预警归因实践

【作者简介】
SunnyZhou，携程资深测试开发工程师，专注于数据应用，以数据维度保障质量。
zw_jin，携程算法工程师，关注自然语言处理算法、大模型 AIGC。

携程酒店前端存在大量监控，但对于监控问题的排查成本，随着量级的增加而变得不可控。因此引入了智能预警归因系统，以数据池统一数据结构及标准；以预警规则池保证预警的准确性、低噪音；以算法模型进行根因分析，直接给出归因结果，从而提高整体排障效率。

一、背景

携程酒店大前端为了监测生产用户运行的稳定性和流畅度，对大量的系统指标、业务场景配置了监控。在运作的过程中我们发现，监控确实能发现很多的问题，但由此带来的排查成本随着量级的增加也变的不可控。

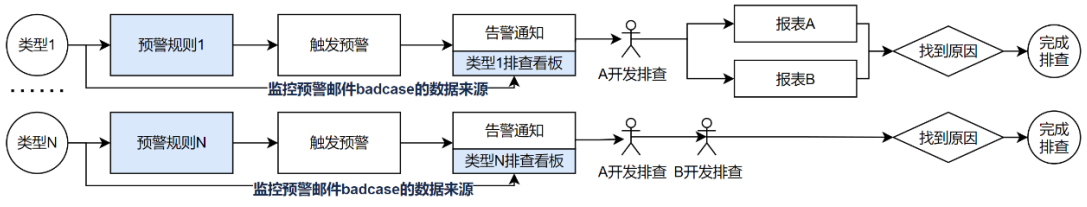
以系统指标监控说明：

目前约有 30+类型，如页面慢加载、白屏等。每种异常监控都是独立的数据处理、预警机制、问题排查分析流程，互相之间没有关联。这一现状导致了一个个的数据孤岛，管理的高度碎片化，且无法复用，让维护和新增的成本较高。

同时，问题的原因不尽相同，排查链路也各自独立。举个例子：页面白屏异常上涨时，通过白屏的监控规则，触发预警后通知相关人员。排查人员从现有的白屏报表中进行 badcase 分析，排查锁定初步原因后，需要跨多个报表进行交叉验证，直至找到根本原因，整个过程中会耗费大量人力和时间成本。

在对页面白屏问题的深入剖析中，发现其根本原因往往与服务失败、服务耗时增长、内存溢出、增量数据拉取失败等因素密切相关。在现有的链路繁琐、异常之间没有关联性、分析效率低下、分析结果也不尽如人意。

现有监控触发及排查流程

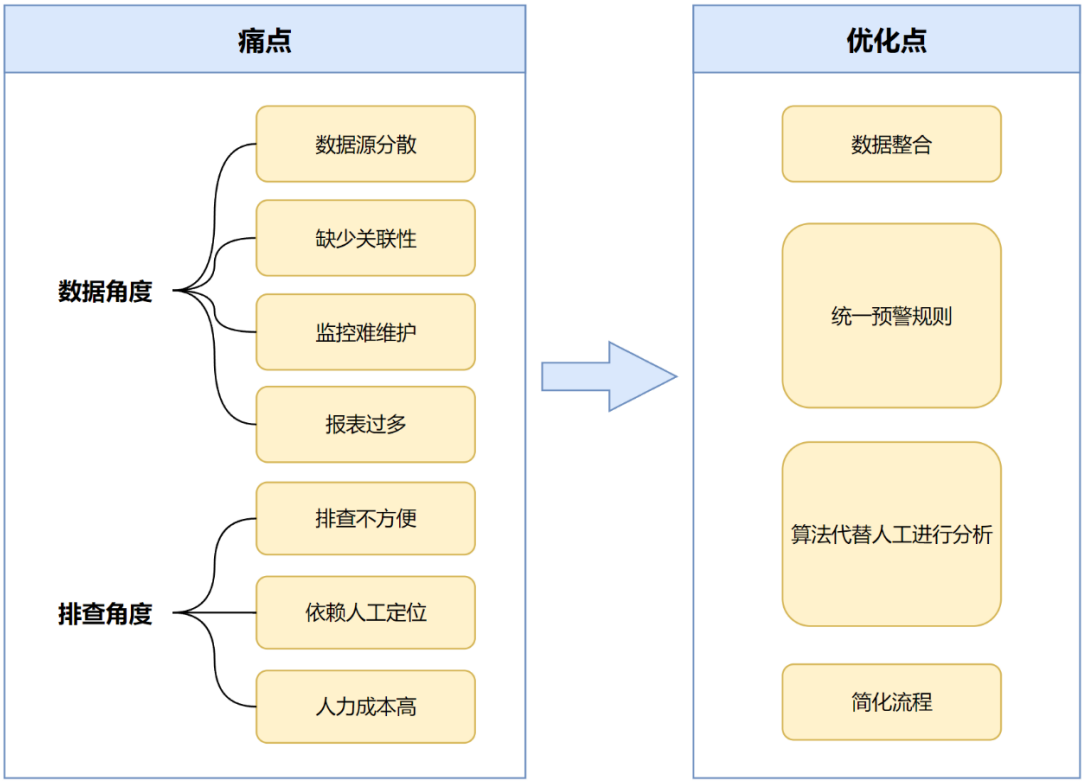


基于目前情况，该如何找到解决方案呢？

经过对流程及问题的梳理，罗列了一些当前存在的核心痛点。

数据角度：数据源分散，缺少关联性，报表过多，监控难维护&增加成本高

排查角度：现有流程链路长排查不方便，非常依赖人工定位，系统并不能很好的给出实际有问题点的方向。



针对上述问题，优化策略已明确成形，聚焦于几个核心要点：

数据整合：整合分散的数据源，确保数据结构的一致性与准确性，为后续分析奠定坚实基础。

统一预警规则：抽取预警规则库，形成统一的规则，减少重复编写预警流程和调整规则，提高预警的效率和准确性

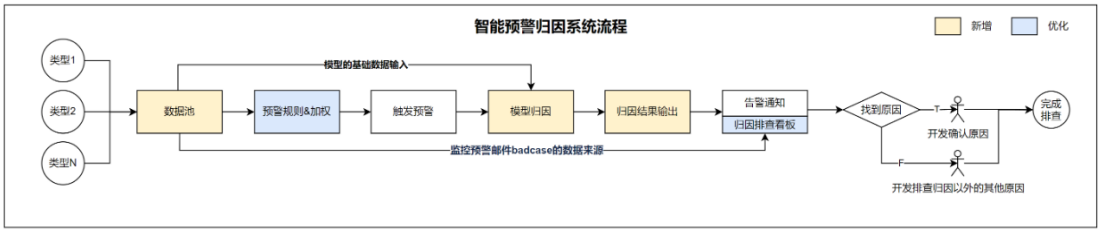
算法归因分析：引入算法模型，深入挖掘不同数据与维度间的内在联系，精准归因问题根源，有效规避人工判断的主观性与不确定性，提高故障排查的效率。

简化流程：缩短故障排查路径，通过优化聚合排查报表，让问题定位更加直观快捷。

二、智能归因体系实现

2.1 整体方案

基于上述的优化方向，我们搭建了酒店前端智能预警归因系统，将多个数据预警孤岛串联起来，形成数据池。抽取统一且丰富预警规则，形成预警规则库，保证预警准确性。针对预警的异常，引进算法模型，实现算法模型归因分析，快速定位问题根源，并生成详尽的归因报告，给到开发明确排查方向。最终，将分析结果转化为清晰的告警通知邮件&排查报表，助力团队迅速响应，开展有效的故障排查与解决，大大提高了排查效率。



2.2 数据池

在整个体系中，搭建数据池是其重要的基石。数据池起到整合不同数据源的作用，将不同格式、维度的数据集中存储在同一个数据池中，对其定义统一的数据结构及标准，确保数据一致性和准确性。

拆解来看，将数据池划分为六大字段维度：数据类型、平台、指标、核心监控维度、基础字段以及业务特有字段。细致的数据维度分类不仅使得数据汇总更加直观，同时也大大提升了数据的结构化处理能力、降低后续使用成本。

数据池字段解构	数据类型	页面慢加载、页面白屏……
	平台	APP、PC、H5……
	指标	业务指标……
	核心监控维度	页面、页面名称……
	基础字段	操作系统、版本、触发时间……
	业务特有字段	加载时间、模块、大小……

数据池的数据，为以下 3 个流程，提供了核心的数据支持：

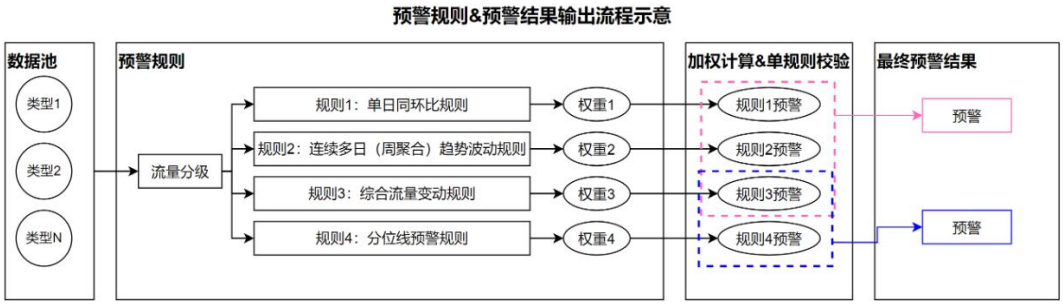
1）算法模型的基础输入：为算法模型提供必要的输入数据，确保模型运算数据的准确性和有效性。

2）预警规则的底表：作为预警系统的基础数据，为规则设定提供坚实的数据支撑，确保预警系统的精确性和可靠性。

3）监控预警邮件 badcase 的数据来源：针对监控预警邮件中的 badcase，提供了详细且多维度的数据来源，以便进行深入的分析。

2.3 预警规则&加权

预警数据的准确性，是预警归因系统中的重要一环，如何让监控保证准确、不遗漏，且降低误报的概率呢？为了达到这个目标，把数据进行流量分级，且构建了多类预警规则，以权重加权的方式，进行预警判断，最后把多个规则的预警结果汇总起来，成为最终的结果。



首先说说流量分级和预警规则。

从数据池获取数据时，把数据本身的流量进行分级，分为高、中、低、仅关注四级。根据类型不同，并针对不同流量级别设置了合理的变化率阈值。流量越大，则对变化率越敏感。

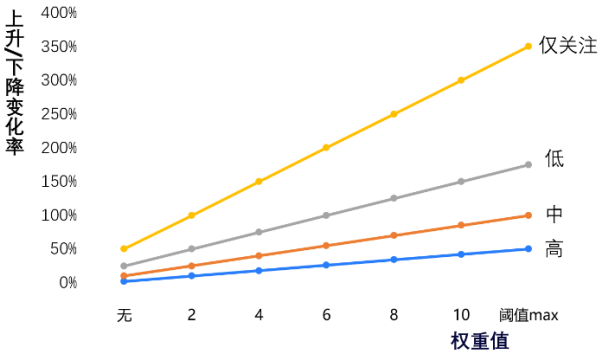
权重计算公式：权重=流量分级系数*变化率

举个例子：数据变化率上升 50%，通过规则，会计算出高级的权重值为 max，中级的权重值为 5，低级的权重值则为 2，仅关注的权重值则为 0。

流量分级

类别	范围
高	万级及以上流量
中	千级流量
低	百级流量
仅关注	小流量

不同流量&变化率的阈值设定



在预警规则上，构建了四种不同维度的规则，以确保数据异常&变化的及时发现与处理：

1）单日同环比规则：通过对比当前数据与上周同日（同比）或相邻日期（环比）的数据，发现单日内的显著变化，如激增、骤减、新增或异常波动。

2）连续多日（周聚合）趋势波动规则：除了关注单日的数据变化外，还制定了周维度的数据趋势。通过对比每周的数据变化，能够更准确地评估数据的长期或持续性的趋势异常。这一预警规则不仅关注数据的变化幅度，还关注数据的变化方向和速度，从而更全面地把握数据的动态变化，确保不漏掉那些单日波动中被忽略的异常情况。

3）综合流量变动规则：该规则不仅考虑了数据本身的变化，还结合了与其相关指标/大盘流量，从而降低因用户流量放大、缩小而导致的数据变动预警噪音，确保预警的准确性。例如，针对页面白屏异常，监测异常率（异常数/页面流量）。

4）分位线预警规则：针对非报错类数据源，引入了分位线规则，以天为单位设定最优分位线作为基准，实时监测高于或低于该基准的数据占比变化及波动，确保数据的细微变化可以反馈出来。

基于以上四种预警规则，再结合数据本身的流量大小，设置了合理的变动权重。通过对权重进行加权计算后，能够精准得出每日的预警数据，确保系统与业务中的数据变动得到及时发现和处理。

在得到各规则的对比权重值后，同样根据流量分级进行分类加权，对加权结果进行阈值或权重的过滤，以确定在该规则下是否触发预警（具体参考“触发预警标准表”）。再根据不同的业务需求，叠加多个规则进行最终的数据预警，以确保预警的精准性，并有效降低预警的噪音。

触发预警标准						
流量分级	单日同环比规则		连续多日（周聚合）趋势波动规则		综合流量变动规则	分位线预警规则
	阈值	权重	阈值	权重	权重	权重
高	次数或人数达到阈值次数	权重加总分级判定	次数或人数达到阈值次数	权重加总分级判定	人数权重分级判定	次数权重分级判定
中						
低	次数和人数达到阈值次数					
仅关注						

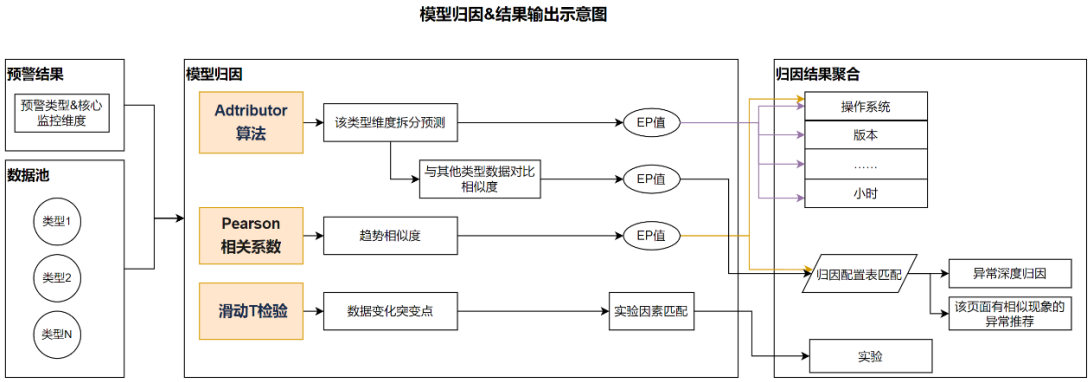
2.4 模型归因

有了实际预警的类型后，基于预警结果及预警池中全量异常数据，将其作为算法模型的数据输入，经过不同的模型进行归因分析，最终获得归因结果值。

在实践中，考虑到对异常数据进行本身根因的分析、与其相关的其他问题导致情况、趋势问题等各类问题归因，使用了以下三种算法模型：

- Adtributor 算法

- Pearson 相关系数
- 滑动 T 检验



2.4.1 Adtributor 算法

Adtributor 算法是微软研究院于 2014 年提出的一种多维时间序列异常根因分析方法，在多维度复杂根因的场景下具有良好的可靠性。在任务中，首先对监控的各种关键性能指标(KPI)进行实时异常检测，当一个 KPI 发生异常时，想要解除异常，定位出其根因所在的位置是非常关键的一步。归因系统需要对检测到的异常指标进行维度分析和根因定位，而 Adtributor 算法非常适用于这个场景。

首先对异常指标进行维度拆分，维度拆分意在从预警结果的诸多维度中，分析出哪几个维度对一条预警造成的影响最大。

使用 Adtributor 算法来实现维度拆分主要包括以下几个步骤：

1）数据收集：以 pv、uv 作为参考指标 KPI，将预警结果和预警池作为数据源，按维度构造原始数据集。

2）异常检测：采用 ARMA 时间序列模型对 KPI 进行实时预测，将预测值 F 和真实值 A 对比，判断 KPI 是否发生异常。

3）维度分析：Adtributor 对异常 KPI 的所有维度和元素计算 EP 值和 S 值，从而定位到影响程度最大的维度。

下面对重要细节展开阐述：

数据收集

对于发生预警的数据，通过关联预警池，获取到每条预警在某个维度下，过去 60 天内每天的明细 pv、uv 值。

异常检测

ARMA（auto regressive moving average model，自回归移动平均）模型是研究时间序列的重要方法，它可以基于历史数据来预测未来的事情。

在每个维度下，基于过去 59 天的历史数据，使用 ARMA 模型推理得到当天的预测 pv、uv 值。将预测值与真实值进行对比，可以判断预警是否发生异常。

维度分析

将多维分析问题分解为多个单维分析问题，采用 EP 值和 S 值定位出每个维度下的异常元素集合，最后根据每个维度总的 S 值大小汇总输出相关维度集合。

1）对于异常 KPI，计算所有维度下所有元素的真实值和预测值的差异，即 S 值(Surprise，意外性)，将每一维度下的元素依据 S 值大小降序排列；

2）对于每一维度，从上到下依次对排序后的所有元素计算其变化在 KPI 变化总量中的占比，即元素对于 KPI 异常的解释能力 EP 值(Explanatory power，解释能力)。

EP 值

对于每一维度，如果元素的波动变化在异常 KPI 的波动变化中的占比越大，则认为元素越能解释 KPI 异常的发生。EP 值(Explanatory power，解释能力)用于衡量元素对异常的解释能力。EP 值计算公式如下：

$$EP_{ij} = (A_{ij}(m) - F_{ij}(m))/(A(m) - F(m))$$

式中，A 为真实值，F 为 ARMA 时间序列模型正常预测值，下标 i 为维度、j 为元素、m 为异常指标。EP 值可以为正、为负、大于 100%，但是每个维度下的所有元素 EP 之和必须为 100%。EP 为正表示可能是异常维度，为负表示不是异常维度，大于 100%表示与 KPI 异常有非常明显的正相关关系。

S 值

如果 KPI 在某一维度下的真实值和预测值相差越大，则越有可能是异常维度。

KPI 先验概率和后验概率的相似度可以采用相对熵(relative entropy)来衡量，进而判断维度是否具有意外性。JSD 散度(Jensen-Shannnon divergence)是相对熵的一种变形，其对称性好、对于零值具有适应性、对相似度判别更加准确，这里基于 JSD 散度，得到 S 值(Surprise，意外性)公式如下：

$$S_{ij}(m) = 0.5\,(p\,log(\frac{2p}{p+q}) + q\,log(\frac{2q}{p+q}))$$

其中，预测概率或先验概率计算公式为：

$$p_{ij}(m) = F_{ij}(m)/F(m), \forall E_{ij}$$

真实概率或后验概率计算公式为：

$$q_{ij}(m) = A_{ij}(m)/A(m), \forall E_{ij}$$

维度的 S 值为维度下所有元素的 S 值之和，即：

$$S_i = sum(S_{ij})$$

在得到维度拆分的分析结果之后，还需要做深度根因的定位，根因定位意在从预警结果中，找出造成该预警的深度原因。例如：预警类型为页面白屏，而深度原因在于内存溢出，算法需要通过页面白屏与内存溢出之间的关联性，定位到深度原因。

通过计算预警结果所属类型的 uv、pv 行为，与其他深度类型 uv、pv 的相似度，来匹配深度原因。深度归因的实现是基于维度拆分的结果，具体来说：首先得到了一条告警影响最大的一个或者几个维度，然后只获取这些维度下的 uv、pv 指标作为特征向量，忽略其他影响不大的维度。对于所有的候选深度类型，也获取相同维度下的特征向量，然后计算向量间的余弦相似度，作为数据类型之间的关联性指标。相似度最接近的深度类型，即是该条预警关联性最高的深度原因。

2.4.2 Pearson 相关系数（Pearson Correlation Coefficient）

考虑到通过 Adtributor 算法定位深度原因，没有充分利用 uv、pv 的趋势变化，而事实上趋势的变化是预警判断时的重要依据，因此这里补充使用了另外一种基于趋势的相似度计算方法来定位预警的深度原因。

首先对预警明细表基于每个维度聚合，得到 21 天的 pv、uv 值作为特征表。然后对于每一条预警数据，以及待匹配的深度类型，都从特征表中获取相关特征，然后计算特征间的相似度。

由于主要考虑特征在趋势上相似性，因此选择 Pearson 相关系数（Pearson Correlation Coefficient）作为相似度指标。Pearson 相关系数的绝对值越大，相关性越强，正数表示正相关，负数表示负相关。

总体相关系数公式如下：

$$\rho_{X,Y} = \frac{\operatorname{cov}(X,Y)}{\sigma_X \sigma_Y} = \frac{E[(X-\mu_X)(Y-\mu_Y)]}{\sigma_X \sigma_Y}$$

其中 cov(X,Y)是 X，Y 的协方差，σX 是 X 的标准差，σY 是 Y 的标准差。

2.4.3 滑动 T 检验(Moving T test , MTT)

上述两种方法分别通过维度拆分、考虑趋势变化来定位预警的深度原因，已经比较周全，但仍存在问题，例如：开启的实验和页面的预警不能完全匹配。因此针对实验维度，启用了另外一种方案来解决，该方案通过匹配 KPI 的突变时刻与业务变更时间来实现。

具体来说，对于每条预警信息，首先获取当天分钟级别的 pv、uv 值，分别作为两组特征变量。对于每一组特征变量，使用滑动 T 检验(Moving T test , MTT)算法来定位发生突变的时刻。

滑动 t 检验是通过考察两组样本平均值的差异是否显著来检验突变。其基本思想是把序列中两段子序列均值有无显著差异看作来自两个总体均值有无显著差异的问题来检验。如果两段子序列的均值差异超过了一定的显著性水平，则可以认为有突变发生。

找到突变时刻后，可以将其与业务变更时间点进行匹配。若在时间范围内匹配成功，则数据变化可能与该业务变更有一定关联。如：结合了 AB 实验的数据，获取了所有 AB 实验流量开启、分流结果等的操作信息。若数据突变点前 10 分钟内，预警页面有相关的实验操作变更，则输出相应的实验归因数据。

2.4.4 归因结果聚合输出

在得到上边四个算法的结果后，按照既定的根因分类，根据根因实际情况，使用单个模型或多个模型的 EP 值结果限制卡点，进行归因结果聚合。

接下来，拆解这些根因的实际算法使用考量。

- 核心优化的异常深度归因和相似异常推荐。首先会根据配置表中的“对该类会有影响的根因类型”和“其他不相关异常推荐类型”，进行算法结果的过滤。在该场景下，主要使用趋势相似性算法结果，深度归因结果作为辅助过滤条件。

- 操作系统等不会短期内变化的维度，则会主要使用趋势相似度模型，再把维度拆分模型的结果作为辅助参考，获得更准确的归因结果

- 版本等维度，由于他们的变化情况及异常波动较大，相比之下就并不适合使用相似度模型，因此仅使用维度拆分模型结果，反而会更精确。

- 实验维度，直接使用突变点与实验操作变更匹配数据进行输出。

有了归因结果后，最终会以预警邮件或消息的形式发送给相关开发，并以监控报表的形式，来展示多维度的分布数据和异常明细，以提高整体排障效率。

看下实际案例：

案例 1：6 月 9 日，某详情页白屏上升，归因到代码报错升高导致

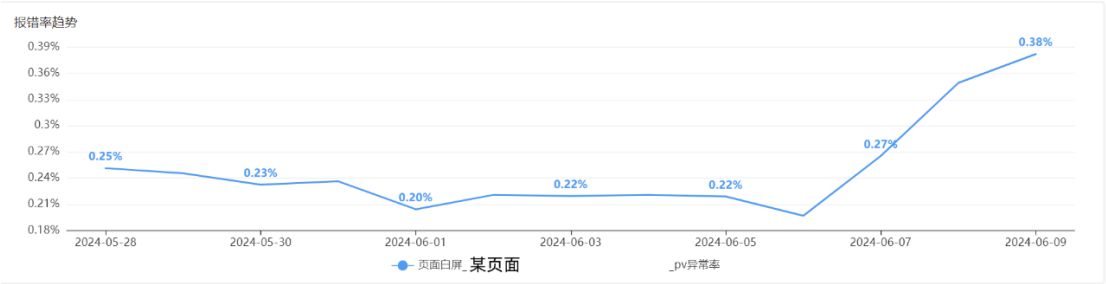
该日，通过多天的同环比规则权重叠加，监控出该页面白屏上升，报错率从 0.22%上升至 0.38%。

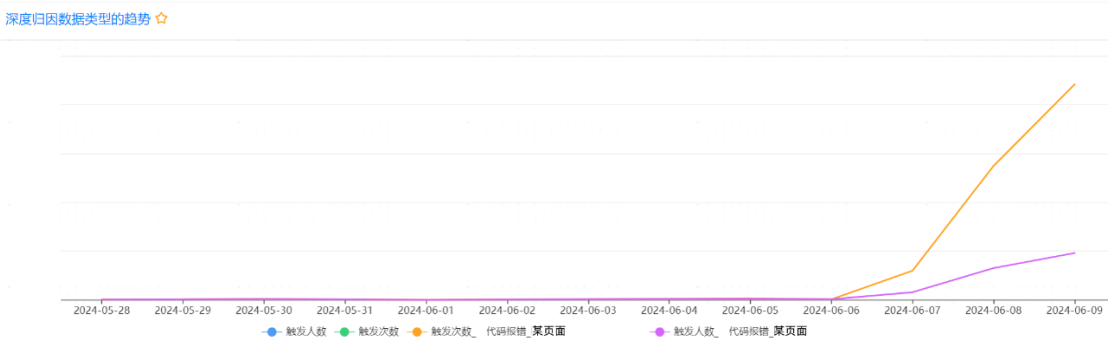
通过算法归因结果输出：锁定了操作系统、版本及频道；且深度归因发现，该类报错，是由于底层代码报错升高导致，两者间的关联度为 97%。

归因结果给到开发有指向性的排查方向，提高排障效率。

预警表明细

日期(分区)	预警原因	页面名称	归因类型	原因明细
2024-06-09	满足上升权重（天）	某详情页	操作系统	Android，iOS
			版本	某版本
			频道名	某频道
			深度归因	代码报错【关联度:0.97】





案例 2：5 月 21 日，某页面图片慢加载上升，归因到与业务实验开启有关

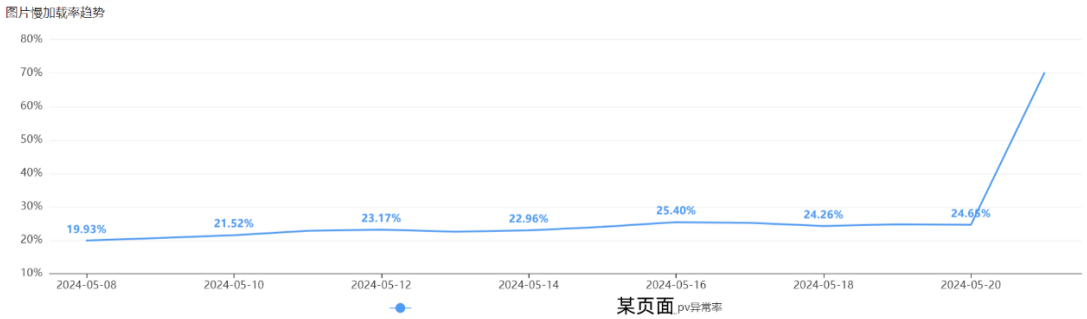
该日，通过多天的同环比规则阈值限制，监控出该页面图片慢加载上升，报错率从 24%上升至 70%。

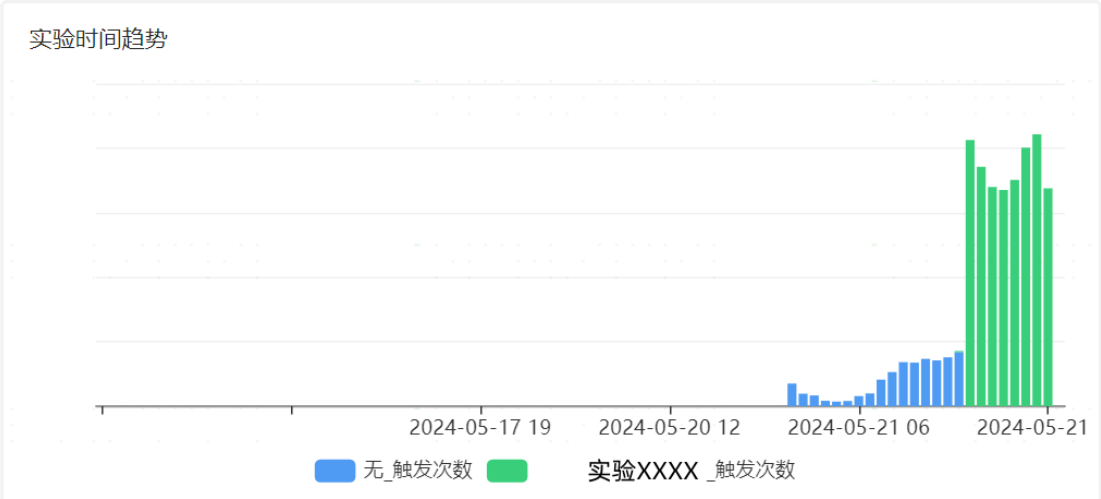
通过算法归因结果输出，该异常升高与平台、系统、及实验开启有关。

归因结果直接定位到了实验，精确到影响该异常的需求改动及关系方，以更好的评估新需求对系统性能是否带来影响。

预警表明细

日期(分区)	预警原因	页面名称	归因类型	原因明细
2024-05-21	满足大流量上升阈值（天）	某页面	实验	2024-05-21 15:59:05 实验 XXXXXXXXXX 变更，状态：实验中（开启实验）
			操作系统	Android，iOS
			版本	某版本





三、阶段成果&后续规划

3.1 阶段成果

1）预警的准确性升高：老版监控的准确度约为 60%，会有数据正常波动的场景也误触预警的情况；新版预警归因系统的准确率约为 89%，有比较明显的改善。

2）从归因结果提供、排查报表数据整合、排查链路缩减三个方面来看，整体排查时间缩短约 40%。

3）数据池中接入 40+的数据类型。

4）已有 19 个传统预警、资源消耗监控，接入到新版智能预警归因系统中。

5）新版归因系统已发现问题约 70+个，其中图片相关问题 20+、页面慢加载问题 10+、资源消耗问题 7 个……。预警后系统能直接找到影响的根本原因，如：内存上涨与图片 size 变大有关，流量上涨与服务调用次数变多有关；异常变化与实验开启有关。

6）该系统降低了不同数据源增加监控的开发成本。

3.2 后续规划

- 丰富预警规则，提高预警的精确度
- 传统预警继续接入新版归因系统中
- 多类指标大盘监控数据，接入智能预警归因系统
- 单日预警接入 bug 缺陷提交系统