ctrip tech 携程

无线大前端／大数据和人工智能
框架架构／质量保障／云计算／运维

携程技术
2019年度合辑

携程技术出品

“携程技术”微信公众号

(ID：ctriptech)

分享，交流，成长~

目录

序

2019 年的携程技术合辑要和大家见面了。

我们把过去一年中踩过的坑，成功的经验，汇总起来进行分享，希望能帮助大家少走一些弯路。合辑中的文章来自携程技术各部门的一线工程师，有基础架构部门，有业务部门，有人工智能和大数据……经过 20 年的发展，携程的业务已经遍布全球，IT 系统如何支持业务低成本高效率的扩张，在这些文章中都能一窥端倪。

2019 年，携程的国际化战略快速推进，我们对系统进行了多语言的改造，让产品的体验更符合当地人的使用习惯，同时又不引入过多的代码分支。供应商和客人遍布于全球，情况变得日益复杂，如何通过技术手段来持续提升服务效率，为客户提供高品质的服务，可以在书中看到大数据和人工智能技术与业务场景的深度结合。

对于 2010 年之前创立的互联网企业，大多采用自建数据中心，当公有云快速发展起来之后，是选择继续在私有云上发展，还是彻底转型使用公有云，是不得不思考的一个问题。云计算的快速发展，为企业提供了弹性的资源使用、高效的线上部署、多数据中心的就近接入、透明的 IT 成本，我们采用公有云+私有云的“混合云”方案，以取得用户体验和成本之间的平衡。书中可以看到我们在公有云和私有云打通、系统架构 Cloud Native 化过程中的经验教训。

新的一年，我们希望能与同行伙伴一起携手共进，开放共享，在追求技术极致的道路上一路向前！

携程技术副总裁 马超

2019 年 12 月 23 日

无线大前端篇

近万字长文详述携程大规模应用 RN 的工程化实践

【作者简介】赵辛贵，携程无线平台研发部开发总监。2013 年加入携程，主要负责 App 基础框架研发相关工作，目前重点关注 React Native 技术在公司的推广和研发支持、无线框架和工程架构升级。

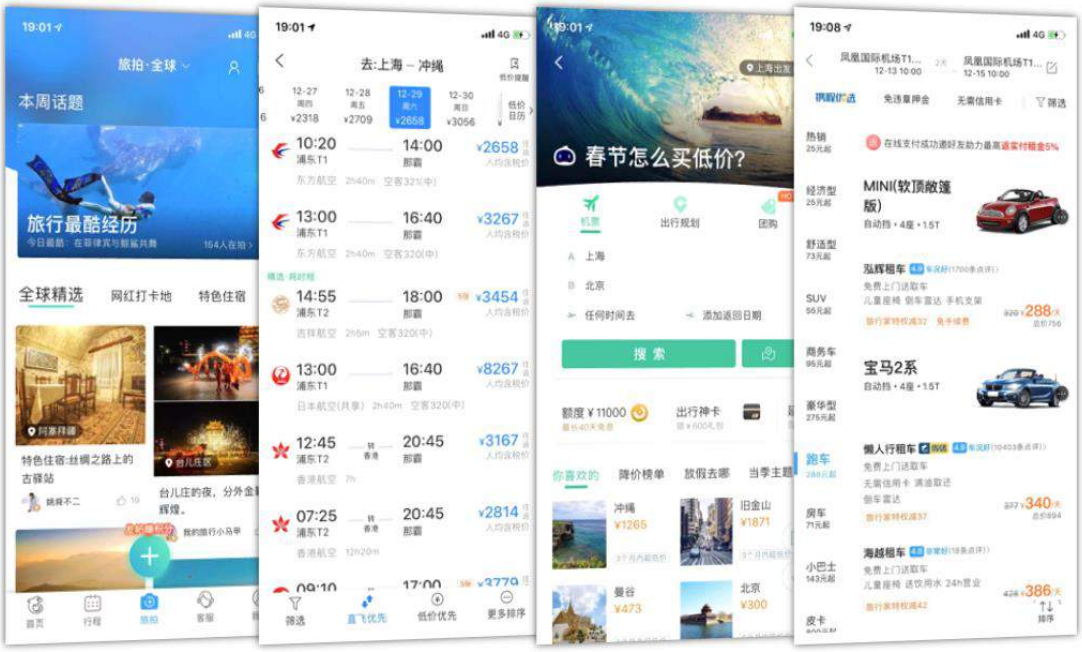
一、RN 在携程的使用情况

2015 年 3 月 React Native iOS 开源，半年之后 Android 开源。携程于 2016 年 6 月份投入资源在 React Native 框架的预研，并于 8 月份正式上线，至今已有 2 年多。

随着业务使用的复杂度增加，各种问题随之而来，我们就这些问题一一提供解决方案，并建设相关配套系统来支撑业务开发团队使用。本文将从携程内部对 RN（ReactNative 简称，下同）的性能稳定性优化以及相关基础设施的建设来做分享。

截止 2018 年 9 月底，使用情况大致如下：

广泛使用：生产环境总共有 104 个 RN 业务 Bundle，其中携程旅行 App 中运行的有 83 个，其它 21 个运行在公司内其它独立 App 中，比如 Trip.com、铁友智行等。从 2016 年 8 月份上线至今，PV 以同比 300%的增速增长，其日 PV 量已是传统 H5 Hybrid 技术的近 2 倍。



深度使用： 全流程使用，比如特价机票、特价酒店、国际机票、租车、旅拍等，已是全流程使用 RN 开发。复杂度高，火车票模块，5.8MB 的 js 代码(uglify 压缩后)，超过 100 个页面，都打包在一个业务 Bundle 中。

总的来说，RN 在携程已经广泛使用于生产环境，并得到业务和用户的认可。

二、CRN 框架

我们基于 React Native 框架优化，定制成适合携程业务的跨平台开发框架 - CRN，提供从开发、发布、运维的全生命周期支持。

开发框架，主要是提供在开发阶段的支持。包括工具&文档、组件和解决方案、跨平台打通和代码托管功能。 工具主要包括 CLI 和 Packer，文档包括 API 文档和设计文档，跨平台主要是抹平平台差异组件间的 API，代码托管是为了方便业务团队，特别是新加入 CRN 开发的团队，可以参考已有业务代码快速上手。

性能优化，主要是为了解决首屏渲染的性能问题和 RN 框架的稳定性问题。为了解决首屏渲染性能问题，我们先后开发了框架拆分和预加载、业务按需加载、业务预加载和渐进式渲染方案，稍后会就这些方案做详细介绍。

发布运维，主要是提供发布系统和性能、错误监控平台，让业务开发同事能够有完备的系统去发现和解决线上问题。

下面会从这几个方面详细介绍。

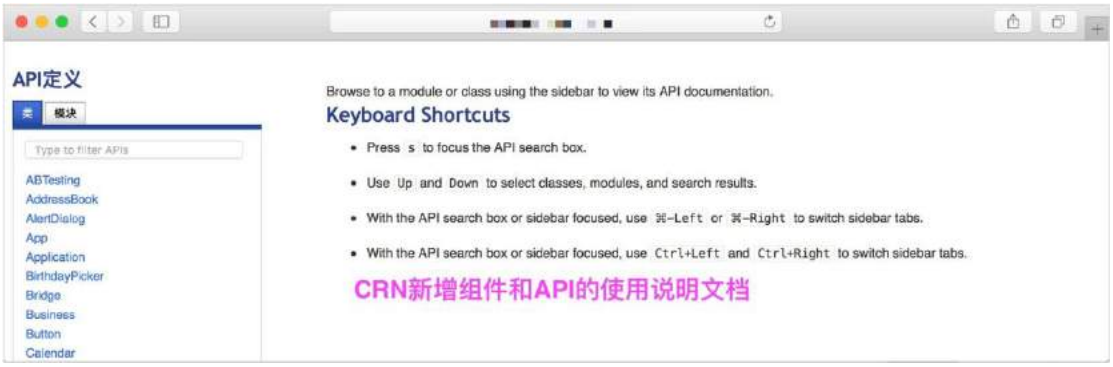
2.1 开发框架

以下是我们的 crn-cli 脚手架，对 RN 原始的 CLI 进行二次包装，提供从工程创建，服务启动，在已集成框架的 App 运行 RN 代码等常用功能，方便开发人员快速上手。

Commands:

init	建立并初始化 CRN 工程，可指定 appId，默认为携程 App
start	启动 CRN 服务,默认端口 5389
run-ios	运行指定 appId 的 IOS App
run-android	运行指定 appId 的 Android App
run-patch	执行 patch,替换 CRN 修改过的 lib 文件
cli-update	更新 cli 版本
example	创建 CRN 组件和 API 调用 Demo 工程
aux	增强型功能入口:log Server、本地打包、上传开发包等 Options:
-h, --help	显示命令帮助
-v, --version	显示版本

文档方面，我们提供 API 文档和设计文档



API 文档采用 YUI doc 根据代码注释自动生成，该文档中主要记录新增组件以及使用示例。

设计文档，主要包含一些组件/API 的设计文档，常见问题解决方案，业务开发常见问题都可以再该文档站点找到对应的解决方案。

2.2 组件和解决方案

提供 100 多个业务和公共组件支持，并保证跨平台提供一致 API。

View组件

- AdView
- Button
- Carousel
- CRNListView
- CRNListViewDataSource
- DatePicker
- DatePickerWidget
- HeaderView
- HtmlText
- LinearGradient
- LoadControl
- LoadingFailedView
- LoadingNoDataView
- LoadingView
- MapView
- Page
- RefreshControl
- ScrollView
- SegmentedControl
- Slider
- SpriteImage
- SwipView
- IconfontView
- .....

API组件

- App
- ABTesting
- AddressBook
- Application
- Calendar
- Channel
- Device
- Encrypt
- Env
- Event
- Fetch
- ImagePicker
- Location
- Log
- Pay
- PhotoBrowser
- QRCode
- ScreenShot
- Share
- Storage
- Toast
- URL
- User
- Zip
- .....

三、CRN 性能优化

分享具体性能优化措施前，先来解释几个基本概念。

- React Native 打包是符合 commonjs 规范的，参考下面的代码：

```
// moduleA.js
module.exports = function( value ){
return value * 2;
}

// moduleB.js
var multiplyBy2 = require('./moduleA');
var result = multiplyBy2(4);
```

简单地说，模块必须通过 module.exports 导出对外的接口或者变量，通过 require()导入其他模块，并同步加载该导入的模块。

- define

简化版 define 实现如下

```
function define(moduleId, factory) {       if (moduleId in modules) {              return;
      }
      modules[moduleId] = {
            factory:factory,
            hasError:false,
            isInitialized:false,
            exports: undefined
      }
}
```

可以看到 define 仅仅是将模块代码嵌入到 factory 中，cache 到 modules 对象内部，并未真正执行。

- require

简化版 require 实现如下：

```
function require(moduleId) {
var module = modules[moduleId];
return (module && module.isInitialized)?module.exports:
      guardedLoadModule(moduleId, module);//代码执行，并赋值给 module.exports
}
```

可以看到，require 是真正模块代码执行的点，JS 模块数越多，耗时越长。guardedLoadModule 内部会使用 try/catch 包裹去执行模块代码，此处可以捕获所有模块的代码异常，RN 内部的 js 错误，都是从此处抛出。

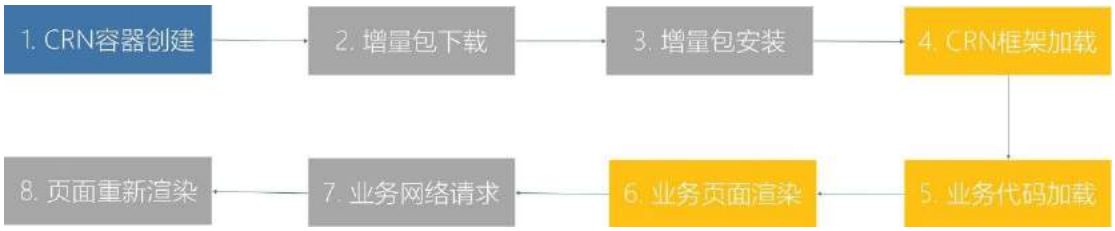
- import

import 在 bundle 编译前后的示例代码如下：

```
/*源码*/
import page1 from './src/Page1.js'
import page2 from './src/Page2.js'

/*编译后*/
var _Page = require(662); //662=./src/Page1.js
var _Page2 = _interopRequireDefault(_Page);
var _Page3 = require(663); //662=./src/Page2.js
var _Page4 = _interopRequireDefault(_Page3);
```

简单地说，编译后 import 等价于 require。

- 页面加载流程



以上是一个 RN 页面加载的全流程，首选是 Native 容器的创建，接着是下载安装最新包(如果有的话)，之后开始 CRN 框架(包含 Native 和 JS 组件)加载，框架加载完成之后，加载业务代码，计算页面虚拟 dom，通知 Native 进行页面首次渲染，如果有网络请求，请求完成之后，再次渲染。

灰色部分是可选的，真实 RN 页面的渲染性能包含 4、5、6 三部分，针对这三部分，我们提供了不同的性能优化方案。

CRN 框架加载：框架和业务代码拆分、框架代码预加载、JSC 执行引擎缓存
业务代码加载：业务代码按需加载、业务代码预加载
业务页面渲染：渐进式渲染、骨架图预渲染

接下来我们一一介绍。

3.1 CRN 框架加载的优化

先看下 react-native bundle 命令打包之后的 bundle 文件结构

```
//头部 - 全局变量定义
(function(global) {
global.require = _require;
global.__d = define; /*...code... */
})(typeof global !== 'undefined' ? global : typeof self !== 'undefined' ? self:this);

//中间 -- 各模块定义部分
__d(/* demo/index.ios.js */function(global, require, module, exports) {
    var _React Native = require(12); // 12 = react-native
    var theCompnent = require(524); // 524 = ./main
    _React Native.AppRegistry.registerComponent('Demo', function () {return theCompnent;});
    module.exports = theCompnent;
    }, 0, null, "crn-demo/index.ios.js");

__d(/* react-native-implementation */function(global, require, module, exports) {
    var React Native = {/*...code... */}
    module.exports = React Native;
    }, 12, null, "react-native-implementation");

// 尾部 -- 引擎初始化和入口模块执行
;require(50); //50 为 InitializeJavaScriptAppEngine 模块
;require(0); //0 为入口 Component 模块
```

结构可以简化为三部分:

为头部全局变量定义;
中间框架/业务模块定义;
尾部引擎初始化/入口函数调用;

3.1.1 框架和业务代码拆分

先来看看我们打包之后的文件目录结构

```
//框架包 rn_common 目录结构
rn_common
    ├—— common_android.js   //Android CRN 框架代码,包括 RN+CRN 扩展 JS 组件+常用第三方组件
    ├—— common_ios.js       //iOS CRN 框架代码,包括 RN+CRN 扩展 JS 组件+常用第三方组件
    └—— pack.config         //打包日志文件，记录打包时间，RN 版本，App 版本等信息
```

```
//业务包 rn_flight_booking 目录结构
rn_flight_booking
    ├─── _crn_config_v2    //配置文件，记录业务代码所在文件夹，默认是 js-modules，同时记录业务代码入口模块文件名
    ├─── _crn_unbundle     //CRN 打包格式标识文件，该文件存在时候，才当做 CRN 包格式加载
    ├─── assets/           //图片资源目录，定制过资源打包/加载流程，iOS/Android 目录一致
    ├─── fonts/            //字体文件目录，每个 js 模块一个文件，文件名为模块 ID.js
    ├─── js-diffs/         //Android 和 iOS 平台差异代码，Android 优先加载该文件夹中的业务代码
    ├─── js-modules/       //业务 js 代码目录
    └─── pack.config       //打包日志文件，记录打包时间，RN 版本，App 版本等信息
```

rn_common 为框架包，可以再后台线程加载，业务包在进入业务的时候才开始加载。

打包部分：

生成框架 jsbundle

业务代码拆分主要是把中间框架/业务模块定义给拆分开来，拆分的思路很简单，用一个空白页面作为入口点，AppRegistry.registerComponent 加载这个入口点。进入业务时，通过这个入口点页面去加载真实的业务代码。把这个空白的入口点页面作为框架的一部分，通过 react-native bundle 命令打包成框架 jsbundle。

抽取业务 js 代码

对 React Native unbundle 的打包过程进行定制，首先让 iOS 支持 unbundle 打包(默认是不支持的)，将生成的业务 js 模块代码单独保存，每个 js 模块一个文件，文件名即为模块 ID.js；

js 模块加载优化

空白页面入口组件，要能加载(require)真实的业务代码，我们需要改造 RN 的 require 方法，简单修改 Native SDK 中的 JSCExecutor（RCTJSCExecutor.mm/JSCExecutor.cpp）文件，调整 nativeRequire 实现即可。

3.1.2 框架代码预加载

RN 框架 instance

RCTBridge/ReactInstanceManager(后文统称为 instance)是 RN 框架中核心的 2 个类，这个类分别控制不同平台的 JavaScriptCore 的执行，同时又都是各自平台 ReactView 的属性，View 的显示于事件靠它来驱动。

所以为了能做到后台预加载 js 代码，首先要做的就是解开台 ReactView 和 instance 之间的耦合解开，能让 instance 在后台独立加载。处理起来不复杂，只需要对 ReactRootView/RCTRootView 接口做简单调整即可。

上图是我们定义的 CRN 框架 instance 的生命周期状态:

框架加载过程，标记为 Loading 状态
框架加载完成，标记为 Ready 状态
框架引擎被业务使用，标记为 Dirty 状态
框架在加载或者业务使用过程中出了异常，会被标记为 Error 状态

App 启动，我们就会预创建一个框架引擎的 instance，创建完成，状态标记为 Ready 并缓存起来，进入业务时候，会优先使用这个缓存的 instance 去加载业务代码，这个时候进入业务页面，只有业务代码的加载执行时间。当这个缓存的 Ready 状态的 instance 被使用之后，后台立即再创建一个，以备后续业务使用。

根据线上数据统计，我们发现 95%的场景，都能直接使用到后台预创建好的框架 instance，或者是已经加载过业务的 instance。也就是说，进入业务页面，只有 5%的用户，需要耗时间加载 RN 框架代码。

3.1.3 业务 instance 缓存

对于加载过业务代码的框架 instance，在用户离开业务时候，会暂时缓存住，这样如果重复进入页面，少了业务代码的加载执行，打开速度提升明显。当暂存的加载过业务的 instance 数量超过 2 个时，会按照创建时间顺序，回收掉最早创建的 instance。根据线上数据统计，有 15%的场景，都会使用到的加载了业务代码的 instance。

框架代码的加载优化已基本完成，来看我们当时测试的一组数据。

3.1.4 一组数据

上图是 2016 年 10 月，基于 RN 0.30 版本，在 iPhone 6 和 Sony Xperia Z5 机型上，多次测试的平均数据。可以看到，优化后，首屏时间比原来都减少 45%左右。后续我们升级 0.41，0.51 版本，该优化都一直在做，方案和思路都是一样的。

3.2 业务代码加载优化

业务代码加载优化我们主要从 2 个方面考虑，业务代码按需加载和预加载，先简单解释两者的差别：

按需加载：是进入业务模块时候，只加载对应页面的代码
预加载： 是尚未进入业务模块前，即把需要进入业务页面的代码在后台加载执行掉

3.2.1 业务代码按需加载

LazyRequire 按需加载方案

先来看一段我们初始化页面路由表的代码

```
import PageA from ("pages/PageA");
import PageB from ("pages/PageB");
import PageC from ("pages/PageC");
import PageD from ("pages/PageD");

//设置页面路由表
let pageList = [PageA, PageB, PageC, PageD];
App.startApp(pageList);
```

早期业务简单，页面数量少，上面的优化方案已经可以是 RN 基本达到 native 的体验，但是随着业务越来越复杂(当时有业务 bundle，包含 70 多个 Page js 代码 uglify 之后达到 3MB)，首屏加载慢的问题又出来，为此我们实现一种懒加载的方案，进入业务时候，只加载当前需要显示的 Page 的代码， 对业务的使用非常简单，下面是我们懒加载的页面路由代码写法。

```
const PageA = lazyRequire("pages/PageA");
const PageB = lazyRequire("pages/PageB");
const PageC = lazyRequire("pages/PageC");
const PageD = lazyRequire("pages/PageD");
//设置页面路由表
let pageList = [PageA, PageB, PageC, PageD];
App.startApp(pageList);
```

对业务开发来说，切换成本非常低，只需要使用 lazyRequire 函数替代 import 指令。怎么做到的呢，其实也很简单。

```
//LazyRequire 函数定义，返回 lazyModule 对象
LazyModule lazyRequire(path)

LazyModule = {
    load(); //代码真正执行的点，返回执行结果
}
```

细心的同学可能发现这里有个问题，lazyRequire 函数传入的文件相对路径，打包之后，还是相对路径，而打包完成之后，每个业务 js 模块都被打成模块 ID.js 文件，这会导致运行时查找不到这些业务页面的模块。是的，在打包过程中，需要开发一个 babel 插件，将 lazyRequire 函数例的文件路径，转换成模块 ID，实现方式和 import 的 babel 插件基本一致。

随着业务代码增加，进入首屏需要加载(require)的代码会增加，前面分析过，require 会导致 JS 代码的执行，是耗时的操作，最终导致首屏变慢。所以，我们就想，进入业务的时候，只加载第一个 Page 相关的代码，其他页面的，路由跳转过去的时候再加载。

Getter API 导出模块

我们先来看看 React Native 模块内的组件导出方式：

```
//原始代码如下
//Module1.js
console.log("Start load module1");
module.exports = {
doJob:()=> {
console.log("doJob called in module1");
    }
}

//Module2.js
import Module1 from "./Module1";
//执行结果：Start load module1
```

这是最常见的模块导出和引用方式，和我们前面说的一样，import 的时候，实际上会去执行对应的代码。接下来，我们创建一个 common.js（文件名无限制），修改下模块的导出方式，参考下面的代码。

```
//common.js
module.exports = {
get Module1() {
return require('./Module1');
    }
}

//回到 Module2.js 的引用
import Module1 from "./common";
//执行结果：没打印任何日志

Module1.doJob();
//执行结果: 打印以下两条日志
//Start load module1
//doJob called in module1
```

可以看到，通过 ES5 的 getter API 来导出模块，在引用时，代码不会立即执行，直到导出对象真正使用时候，才开始执行。所以如果我们有自己的公共组件，多个业务都需要用到，那么使用 getter API 导出模块是一种不错的选择。其实 RN 里面的 ReactNative 模块导出方式也是这样，参考下面的代码。

```
const ReactNative = {
get ActivityIndicator() { return require('ActivityIndicator'); },
get ART() { return require('React NativeART'); },
get DatePickerIOS() { return require('DatePickerIOS'); },
get DrawerLayoutAndroid() { return require('DrawerLayoutAndroid'); },
get Image() { return require('Image'); },
get ListView() { return require('ListView'); },
//...
}
module.exports = React Native;
```

通过 getter API 导出模块实现按需加载是 ES5 默认支持的，对原始 RN 没有任何侵入性修改，是比较推荐的一种方案。

那我们为何需要 LazyRequire 呢？很明显，使用 getter API 导出替换 LazyRequire 是可行的，只是达到不了按需加载的功效了，因为在赋值页面路由表的时候，需要用到所有的 Page 对象，用到这些对象的时候，会直接触发所有 Page 的代码加载执行。

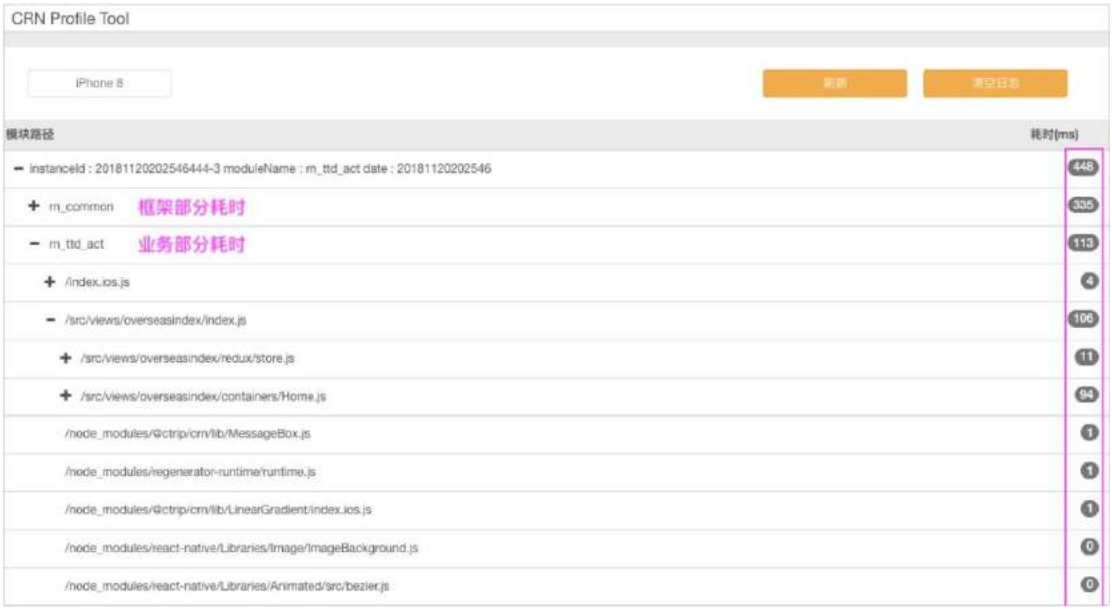
inlineRequire 方案

方案很简单，预先定义模块对象，赋值为 null，在使用时候判断对象是否为 null，null 时候则做真正的 require，进行模块加载。看一段简单示例代码。

```
let VeryExpensiveModule = null;

export default class Optimized extends Component {
    someEvent = ()=>{
        if (VeryExpensiveModule == null) {
            //require('path').default, 动态加载模块代码
            VeryExpensiveModule = require('./VeryExpensive').default;
        }
    }
}
```

3.2.2 工具和数据

为了能方便业务开发同事快速定位到具体是哪个 js 模块加载耗时长，以及具体的调用链是怎样的，我们开发了 CRN Require Profile Tool



如上图所示，业务开发同学，很容易就发现是哪个模块加载耗时长，需要使用按需加载。

按需加载方案是 2017 年，基于 RN 0.41 版本开发的，当时上线前我们也做过首屏性能测试，数据是 iOS 模拟器上跑出来的，由于首次进入业务加载的页面数量猛降，所以首屏时间减少了 2/3。由于这个优化是在 JS 层做的优化，iOS、Android 性能提示基本一致。

3.3 业务代码预加载

经常有这样的业务场景，A 流程订单完成之后，有 B 产品推荐，A、B 业务代码在不同的 RN bundle 里面，A 业务开发完，希望能把 B 业务在后台加载掉，这样用户打开 B 业务首屏速度会更快。为此，我们提供了业务预加载方案。主要两个点，预加载和缓存。

预加载有前面框架代码拆分和预加载的基础，实现起来非常简单，基本没有改造成本。为了能让尽可能多的代码实现预加载，我们在 LazyRequire 里面添加逻辑，让在预加载状态模式下，LazyRequire 等价于 Require，强制加载。

缓存，先前业务代码的缓存是按照业务的 URL 作为 key 来存储的，预加载模式下为了尽可能提高缓存的命中率，我们将缓存的 key 统一成业务 bundle 名，同一业务，同一缓存，这么操作需要业务开发代码也要注意，避免全局变量的使用。

缓存的另外一个问题就是内存占用，我们在提供业务预加载的时候，用一个全局数组来缓存业务 instance，超过限制，或者内存警告时候，会按照 LRU 策略清理没有使用的 instance。实际测试下来，Android 平台，预加载一个业务，会增加 2MB 左右内存(包括框架和业务代码都加载完)，而渲染一个正常页面，占用约 20MB 内存，其中最主要的内存被图片占用。

先前同事在开发这个方案的时候我没在意性能数据，简单测试了下，发现效果非常不错，对于一般页面，业务代码提前预加载后，性能可以达到和 native 基本一致。我们使用了荣耀 7X（千元机，性能偏中低端）进行测试，已经基本感知不到首屏加载和 native 有什么差别了。

3.4 业务页面渲染

我们发现，随着页面复杂度增加，渲染耗时逐渐增加，这也可以理解，要完成页面渲染，需要计算 vitrual dom 的 diff，传输数据给 native，如果数据传输有延迟，就会出现掉帧，为了让页面尽可能快的显示，我们需要简化首次渲染。

渐进式渲染

策略很简单，先渲染 header 部分，setTimeout 去渲染其余部分，如果是 listview/scrollview，先渲染屏幕可视区域，在滑动时候，再渲染其他区域。

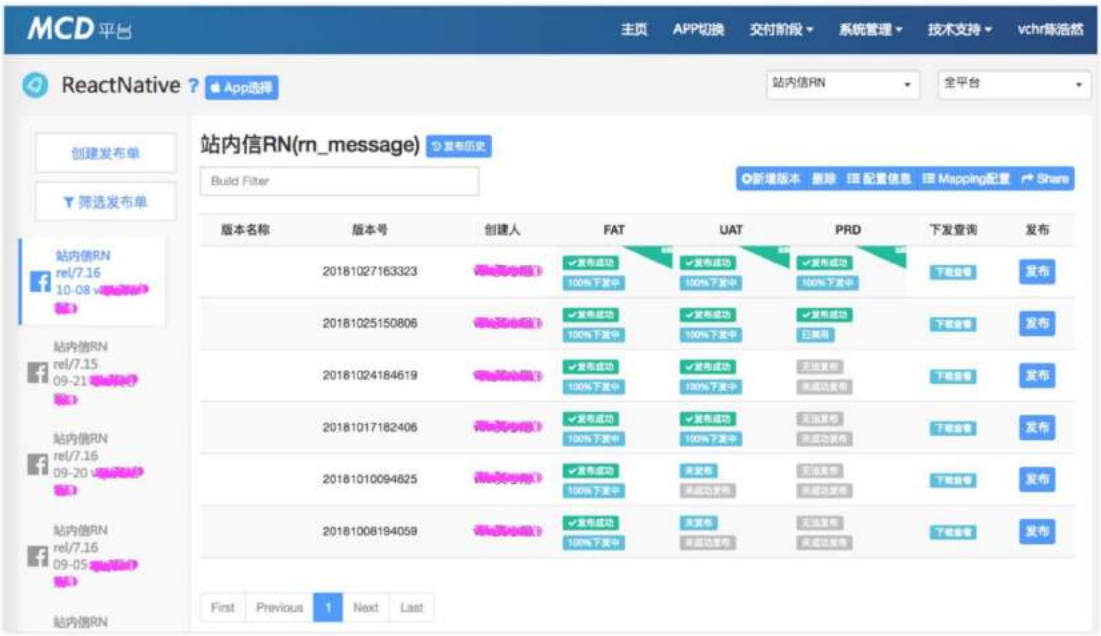
骨架图

先渲染骨架图，由于骨架图相对简单，渲染很快，待请求数据返回后重新渲染界面。骨架图目前没有好的自动渲染框架，需要页面开发同学，根据页面样式，自行开发。

四、发布与运维

一个成熟完善的开发框架，是需要各种配套系统支撑的。我们也为 CRN 开发框架提供了多个内部系统，下面来介绍其中的主要几个。

4.1 发布系统



上图是我们的发布系统页面截图，除了常规的按照版本/平台/环境发布、灰度、回滚支持，我们还增加了发布结果和实时到达率的的报表，方便发布之后，对发布效果评估。

几点说明：

1）不同环境，按照顺便发布，首先发布 FAT(开发环境)、测试通过再发 UAT(跨业务测试环境)、测试通过再发 PRD(生产)。真正的打包只在第一个环境打包，后续的环境都是直接发布前一环境的打包产物，避免重复打包导致的不一致问题，同时也提高发布效率；

2）跨 RN 版本，不支持同时发布，避免新版本 RN 代码发布到老的 RN 版本上，直接在发布系统选择版本的时候做了控制，不能选择 2 个不同 RN 版本的 App；

3）控制发布版本数量，创建发布单时候，可以选择多个版本，经常有发布的同学为了简单，一键勾选所有版本，实际上老版本可能用户量非常小，而回归测试却覆盖不到所有版本，为了避免老版本因为测试不重复导致的问题，我们将版本选择功能做了优化，按照 UV 数量排序，并在版本后面显示 UV 比例，同时默认只能选择 Top5 版本，如果要发布更多版本，需要点击更多，展开其他版本。

监控指标：

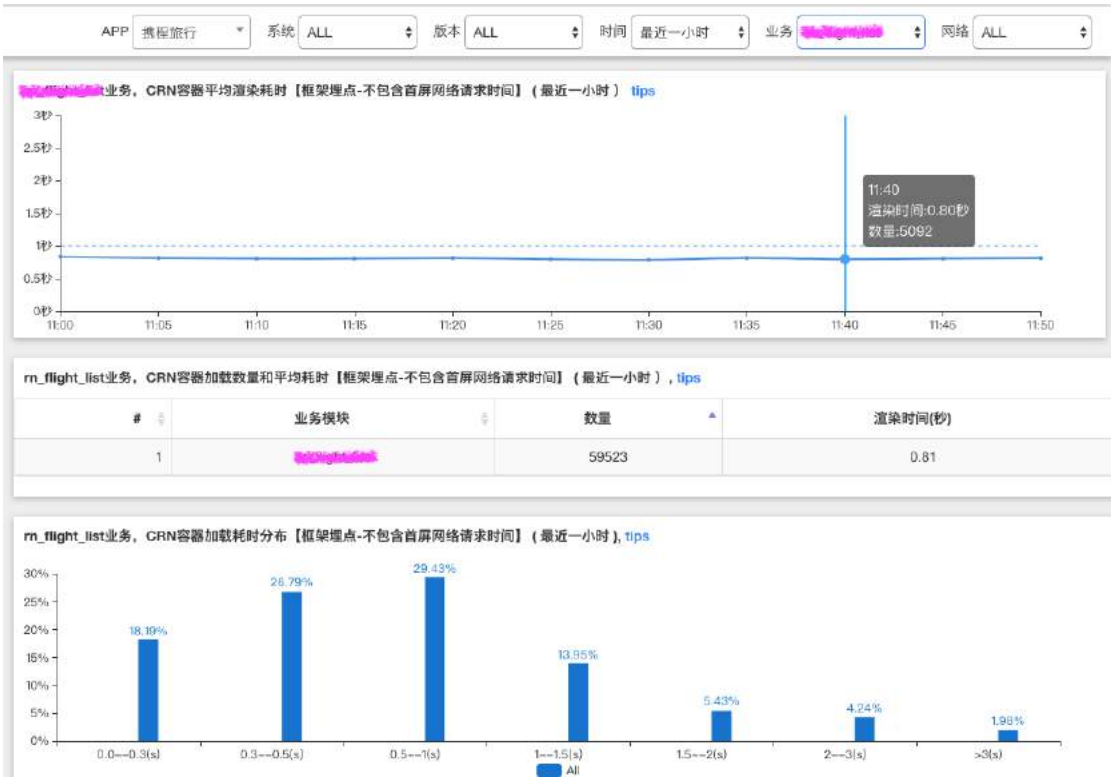
1）发布结果：发布之后，分平台、App 版本展示下载到这个包的成功、失败次数，以及失败的原因分布。

2）实时到达率：这个是业务最应该关注的数据，数据直观的展示，发布之后，实时的有多少比例的用户已经用到最新包。

为了提高实时到达率，我们在打包过程中记录业务模块 ID 和文件名之间的映射，这样可以避免新增文件出现的的大量 JS 文件的文件名(即为模块 ID)变化，从而导致的差分包过大问题。做到只下发真实变更和新增的文件内容。通过线上数据分析，所有首页入口的 RN 模块，新版本发布之后，有 85%的实时到达率，二级及以上入口，实时到达率可以达到 97%。

4.2 性能报表

统计线上业务首屏加载的耗时趋势、分布和使用量，可以支持按照 App/版本/系统过滤查看。



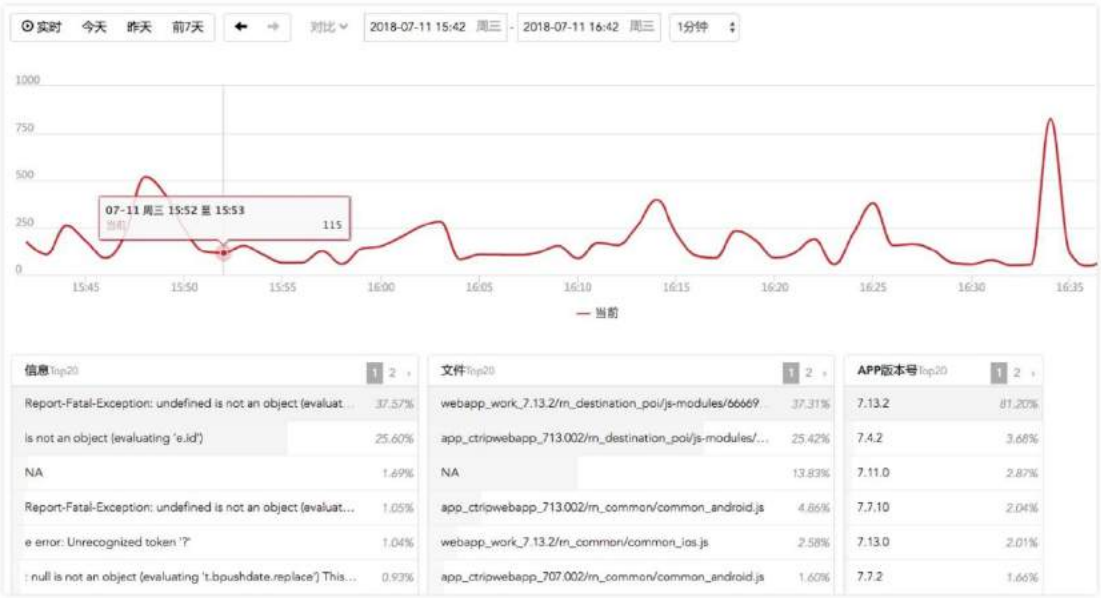
首屏首次渲染完成的时间点，可以在在下面 2 个点添加事件，抛给外层统计。

```
//基于 RN 0.51 版本
//Android ReactRootView.java  添加 dispatchDraw 方法
protected void dispatchDraw (Canvas canvas){
//相对准确，可能会调用多次，内部要做好判断
}

//iOS RCTRootContentView.java
- (void)insertReactSubview:(UIView *)subview atIndex:(NSInteger)atIndex{
//准确,RN 自带的 profile 工具里面的 TT 时间，也是以此处为结束点
}
```

4.3 错误报表

用于收集客户端上报的 RN 错误，包括 JS 执行异常，或者是 native runtime 的一些异常，在业务模块发布之后，必须要到此平台确认自己的发布稳定性是否正常。

除了常规的版本、业务、平台功率，我们在错误堆栈详情页面，还将当前出错的业务包版本和打包记录关联起来，方便开发人员排查问题。



五、其他实践经验

5.1 版本升级

从 2016 年 8 月至今，总共更新 0.28-0.30-0.41-0.51 四个官方 RN 版本，除 0.28 是调研阶段仅使用两个月，其他都使用半年以上。整体升级相对可控，除 0.41 升级 0.51，因为有 PopertyType 组件的移除，需要业务做些适配，其他版本升级对业务都是基本透明的，仅需常规回归测试。

升级流程上，首先是框架团队前期验证(包括打包，SDK 定制，发布，监控全流程确认)、制定升级方案和时间点，接下来是业务团队配合升级和新版本发布，最后是框架团队确认所有业务都在新的 RN 版本重新打包发布过。

升级成本来说，框架团队大约需要 3 名工程师(iOS/Android/前端各 1 人)，2-3 周时间，业务升级和回归测试，一般可在一周内完成。

升级频率上，由于使用的业务团队太多，频繁的升级会对业务造成影响，为了尽可能对业务开发友好，大约 8-12 个月会升级一个 RN 重要版本。当然，如果是有重大的性能升级，比如 RN frabic 的重构版本，我们也会第一时间跟进升级。

5.2 第三方组件版本管理

先看看 npm 模块的版本规则：major.minor.patch， package.json 支持模糊版本，比如>,>=,<,<=,~,^,.x,*，其他都比较好理解，~,^简单解释下(完整本的版本说明参考 semver)

//举例说明
~0.2.0 匹配 [0.2.0, 0.3.0)，有 minor，最大版本为 minor+1, major 不变，patch 为 0
~0 匹配 [0.0.0, 1.0.0)，无 minor，最大版本为 major+1, minor，patch 为 0

^0.2.0 匹配 [0.2.0, 0.3.0)，最大版本为左侧第一个不为 0 的版本号+1
^1.2.0 匹配 [1.2.0, 2.0.0),

我们再看下 react-native-recyclerview-list 这个组件，组件版本和依赖的 RN 版本关系如下。

Component Version	RN Versions
0.1.x	0.45, 0.46
0.2.0 - 0.2.2	0.47, 0.48
0.2.3 - latest	>= 0.49

如果我们使用的 RN 是 0.47 版本，对这个库的依赖方式写成^0.2.0，当组件版本发布到 0.2.2 时候，都使用的很正常，一旦 0.2.3 版本发布，如果再打包发布，则会出现不兼容问题，线上会出大量 JS 报错。

我们就在生产环境出现过类似问题。为了避免类似问题，我们在打包之前做了 preBuildCheck，检测第三方组件的依赖版本，凡是不使用固定版本的，直接报错。

5.3 分平台打包

目的是抹平组件的平台差异，解决资源加载路径不一致的问题。很长一段时间，我们 iOS/Android 的业务代码，只打一次包，以 iOS 平台打包。因为涉及到 Native 代码的新组建的引入，都是由框架团队控制，所以一直以来都没出什么问题。直到公司内部独立 App，他们引入的第三方组件 iOS/Android 有差异，导致发布之后在 Android 上运行有问题。

分平台打包之后，先打包 iOS，再打包 Android，将差异代码存储在 js-diff 目录，加载时，Andorid 先在 js-diff 中查找模块，查找得到直接使用，如果查找不到，再在默认的 js-modules 文件夹中查找。iOS 则只在 js-modules 文件夹中进行模块查找。

5.4 稳定性优化

iOS 平台相对简单，注意解决以下两个 API 相关问题后，绝大部分问题都好处理。

```
//自己注册错误 handler，在此处去进行日志上报，并持续优化
void RCTSetFatalHandler(RCTFatalHandler fatalHandler);

//iOS 所有错误都是通过此次抛出
void RCTFatal(NSError *error);
```

iOS 所有错误都是通过此次抛出 void RCTFatal(NSError *error); ``` iOS RN 注意事项：

- 必须要自己注册错误处理 handler，否则一旦有 RCTFatal 抛出错误，生产环境会有 Crash
- 所有的错误都是 RCTFatal 抛出，为了方便排查问题，需要记录 error 的来源

Android RN 相对复杂，主要注意事项：

- so 加载失败。简单处理可以在原有的 LoadLibrary 加上 try/catch，并在 catch 中再 load 一次，能大幅度降低该问题导致的 Crash；
- ReactInstanceManager 创建过程中的 Native 异常，是通过 DevSupportManager 传递出去，需要处理 DefaultNativeModuleCallExceptionHandler 的 handleException 方法
- JS 执行出错，都是通过 ExceptionsManagerModule 模块抛出，所以需要将该错误和 ReactInstanceManager 相关联，并抛给上层；
- libjsc.so Crash，如果有做 Native Crash 收集，会在后台系统看到不少 libjsc 相关报错，这是由于 RN 自带的 JavaScriptCore 版本为 2014 年的版本，兼容性和稳定性较差，建议参考开源的 jsc-android-buildscrips 项目，将 JavaScriptCore 升级到 2017 年 11 月的版本(WebkitGTK Revision 225067)，我们升级到该版本后，发现该错误降低了 90%；

六、总结

CRN 框架对原生 RN 的大量底层改造优化，解决了性能和稳定性两大核心问题，从落地效果来看，其性能可以做到和 Naitve 基本一致水平，而开发成本却大幅降低。

CRN 框架已在业务团队中广泛使用，为业务的快速迭代提供了强有力支持。对于规模化业务开发团队，使用 RN 作为跨平台开发的解决方案，是切实可行的选择。

2019 年，我们计划根据开发资源情况，适时开源 CRN 框架的部分模块。

携程机票 Node. js 开发实践

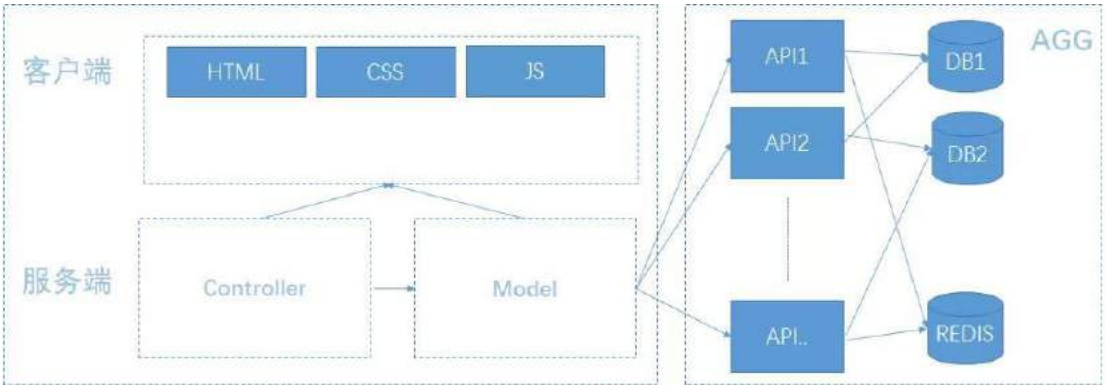
【作者简介】付文平，携程机票研发部前端开发总监。2011 年加入携程，主要负责携程机票 PC、H5、Hybrid 业务方面的开发工作。先后负责机票 PC 前后端分离，H5 Swift 改版，机票 React Native 技术的推进，重点关注 Node.js 技术和产品体验。

Nodejs 自从 2009 年被开发出来以后，至今已经走过了 9 个年头，目前最新的稳定版已经到了 10.13。从问世以后，Nodejs 就深受前端工程师的喜欢。

在携程内部，Nodejs 也是应用广泛，从开发工具到 web 应用，从客户端到服务端，都能见到它的身影。我们也从最初用 Node.js 来完成前后端的架构分离到最近使用 GraphQL 来做微服务，机票部门在 Node.js 的应用探索上越走越宽。

一、前后端分离

在机票事业部前端开发的 web1.0 时代，整个前后端代码耦合在一起，采用的是典型的服务端 MVC 架构。



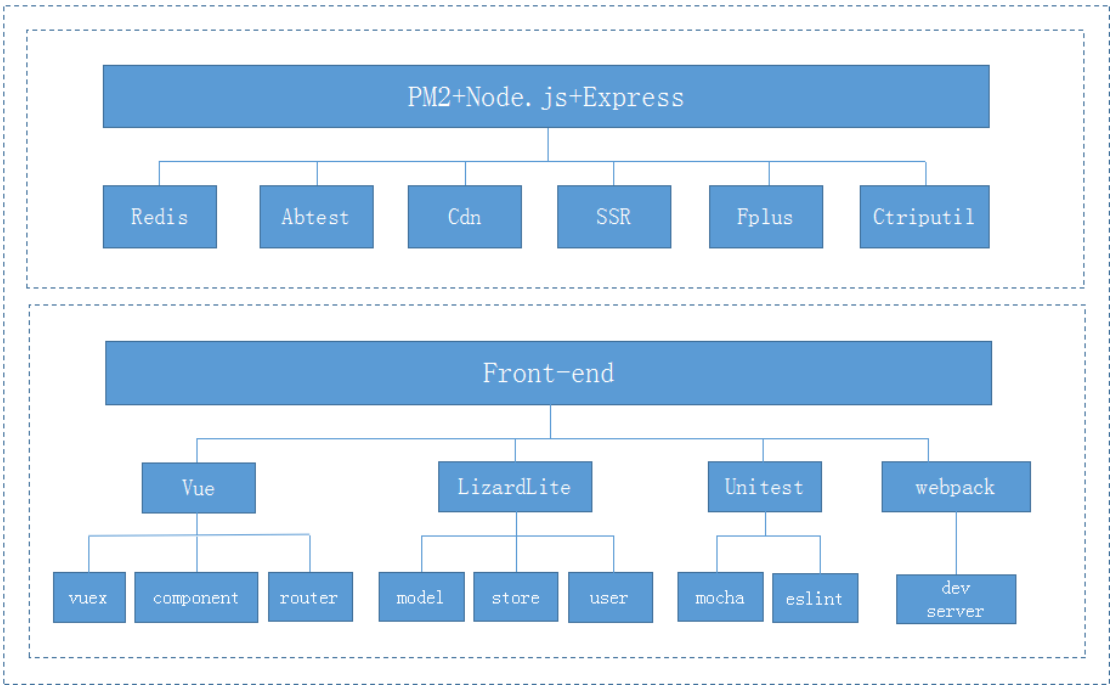
在这样的开发模式下，会存在一些问题和痛点：

- 前后端代码耦合在一起，维护成本比较大，前端的同学不熟悉服务端开发语言，服务端同学也不熟悉前端的交互；
- 展示逻辑和业务逻辑混在在一起，前后端开发同学的职责不明确，有些需求前端说这个逻辑在 view 层，应该后端改，后端说，前端做兼容处理，
- 项目的扩展性比较低，维护性差，迭代速度慢；

在传统的 MVC 模式中，由于 view 层所承载的内容过多，导致 view 层这一块和前端的耦合太多，整体开发效率低下。

能否将这个剥离出来，让前后端集中力量关注自己的领域呢？答案是肯定的，我们将客户端和服务端隔离开，服务端负责数据聚合，提供标准的 restful 接口，前端负责数据渲染。

在机票 H5 实践前后端分离过程中，我们改进了技术架构，在前端的应用层，采用 PM2+Node.js（8.9.4）+Express（4.0）框架，内部基于携程基础框架 ctriputil，同时对一些常用功能的封装，如 Redis 的调用，ABTest 的获取，Qconfig 的集成。



为什么选择 Nodejs 呢?

- Nodejs 采用的是 V8 引擎，运行的是 javascript 代码，对于前端同学来说，学习成本低;
- Nodejs 是事件驱动的，非阻塞性 I/O，非常适合对于前端这种 IO 密集型的应用;
- 社区活跃度高，有大量的库可以被使用
- NPM 生态圈内容丰富
- 客户端代码和应用端可以共享模版和部分逻辑，适合浏览器及服务端代码共用;

在客户端这一层，选用 vue.js ，依托于公司的 lizard.lite 框架，采用 wbebpack 作为构建工具，并通过结合 UT 来提升开发质量。

在 vue 的使用上采用 Vuex 进行状态管理，用 Vue Router 进行路由管理以及用 Lizard.lite 进行 model 层管理（如数据获取、转换、缓存、日志记录、环境切换等）。对于基础数据信息采用 Localstore 进行本地持久化存储，对于状态数据采用 Sessionstore 进行管理，确保状态在当次 session 中是有效的。

自动化代码集成方面我们采用 ESlint\TSlint 做一些基本的语法检查，同时使用 mocha 进行单元测试，确保开发质量，同时按 controller\model\fue 进行分层，确保每个模块之间相对独立。

整个改造后的架构具备以下特性:

模块化： ES6 import + System.import + vue 单文件组件
单页路由控制： vue-router + async component
服务器通信： 同构的 businiess model（LizardLite.AbsModel）
状态管理： vuex store
代码质量： standardjs + eslint + mocha + chai
构建发布： webpack dev server + npm scripts + html-minifier/uglify js/clean css

整个机票 H5 预订流程采用单页+SSR 模式进行开发，获得了 APP-LIKE 式的体验。

针对直接 Landing 页面，采用 APPSHELL 进行服务端加载骨架，提升首屏可视加载速度，对非 Landing 页面采用 SPA 模式，提升后续页面加载速度流畅度，对于搜索引擎的爬虫，会自动识别并进行服务端渲染，做到客户端和服务端代码复用。

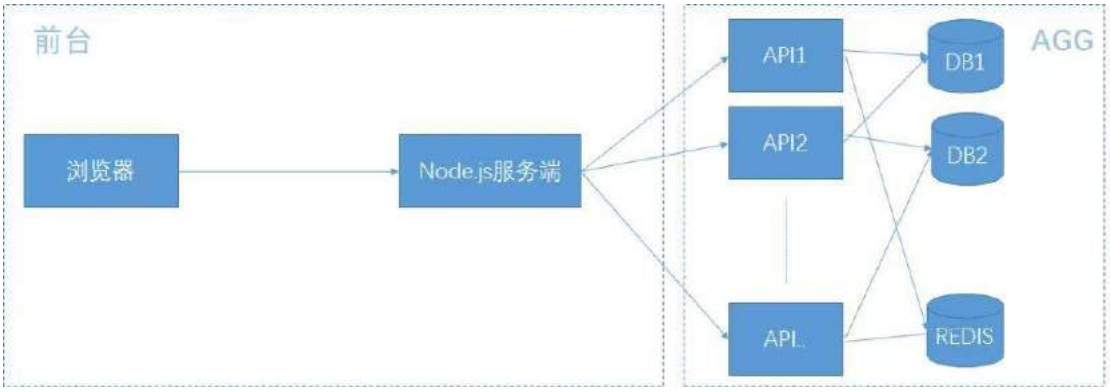
为降低每个页面的资源加载耗时，会对页面资源文件进行拆分和后续页面资源的预加载，同时利用大数据进行用户行为的预测以及接口数据预处理，使得页面速度的加载耗时得到比较大的提升。

二、Node.js 与 restfulAPI

在采用 Node.js 来完成前后端分离后，整个前台的架构分为三大块，一个是以浏览器渲染为主的客户端，二是 Node.js 为主的应用端，三是前台的数据聚合层，在前台的数据聚合层采用 JAVA 作为主要开发语言，对接后台底层的接口。

在 2016 年以来，机票前台开发组开始推行敏捷开发，采用 scrum 的模式进行敏捷管理，并组建比较多的敏捷团队，由于有些敏捷团队比较小，人数相比较少，团队中经常客户端、服务端都只有 1 个或者 2 个同学，如果遇到一个项目是服务端逻辑比较多的时候，服务端资源会爆掉，在遇到改版类项目时，前端资源会爆掉，但由于前后的技术栈不统一，团队内部开发资源相互协调起来比较困难。

如何让团队的效能发挥到最大是我们一直在思考的问题，于是我们在 scrum 团队尝试技术栈统一，将前台的数据聚合层改为用 Node.js 来实现，使得整个团队内部以前端开发工程师为主。



整个 Node 层的架构和 H5 应用层类似，也是采用 PM2+Node.js（8.9.4）+Express（4.0）+CtripUtil，为了提供标准的 restfulAPI，我们在服务入口做了自动化的注册方式，方便客户端接入；

```
/*
 * 发起SOA请求
 * serviceCode/url
 * soa method
 * params
 */
soaAgent: function (serviceCode, method, params, query) {
  const start = Date.now() // 记录请求开始时间

  serviceCode = this.getConfig(`${SOA_PREFIX}_${serviceCode}_${method}`) || this.getConfig(`${SOA_PREFIX}_${serviceCode}`) || serviceCode

  let preifx = `${serviceCode}_${method}_`
  this.logTime(`${preifx}Start`)

  const loginfo = {guid: this.Guid, ClientIP: this.getRemoteIpAddress(), ClientId: this.ClientId, serviceCode, method}

  // 记录发送的请求参数
  clog.info(Object.assign({type:'request'}, loginfo), `soa request: ${serviceCode}\t${method}`, JSON.stringify(params), `开始时间:${start}`)

  params.rejectIncludeResponse = true

  return CtripUtil.SoaAgent(serviceCode).invoke(method, params, true, query).then((res) => {
    this.logTime(`${preifx}End`)

    const end = Date.now()
    const used = end - start

    clog.info(Object.assign({type:'response'}, loginfo), `soa response: ${serviceCode}\t${method}`, `结束时间: ${end}\t耗时:${used}`)

    return res
  }).catch(([e, params, result]) => {
    this.logTime(`${preifx}Error`)

    const end = Date.now()
    const used = end - start

    clog.warn(Object.assign({type:'response'}, loginfo), `soa response: ${serviceCode}\t${method}`, e, JSON.stringify(params), JSON.stringify(result),

    return Promise.reject(e)
  })
},
```

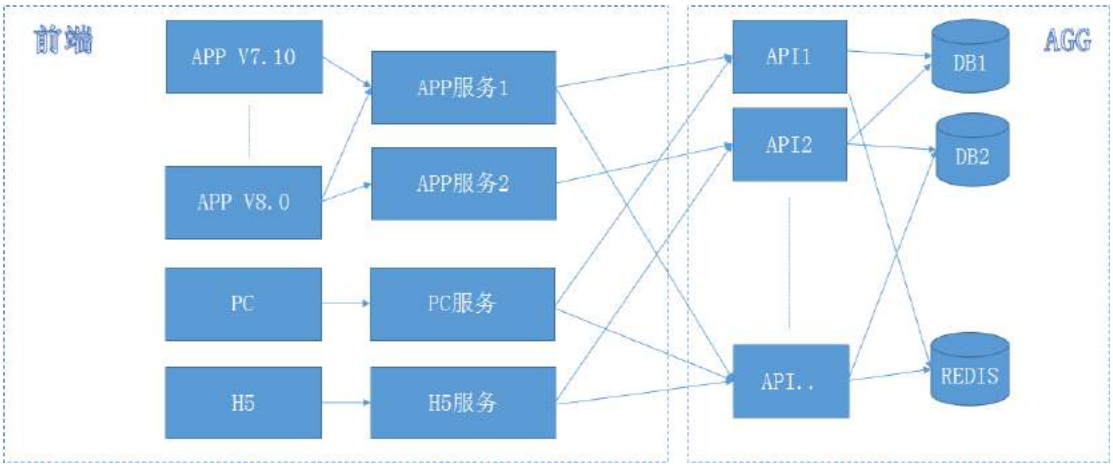
在 Node 层内部针对后台接口的调用做了深度封装，在使用上更加方便快捷，同时接入公司 cat/clog 等通用日志系统。

```
module.exports = function restapi ({ app, vd }) {
  const path = require('path')
  const REST_API_ROOT = path.join(path.resolve('.'), 'restapi')
  const wrap = require('../utils/wrap.js')
  const dir = require('../utils/dir.js')
  const router = require('express').Router()

  const operations = []

  dir(REST_API_ROOT, (route, module) => {
    const { Request, Response } = module

    operations.push({
      Name: route,
      RequestMessage: { Json: JSON.stringify(Request || {}) },
      ResponseMessage: { Json: JSON.stringify(Response || {}) },
    })

    switch (typeof module) { // 注册路由
      case 'object': // 如果模块导出的是一个对象，则判断 是否有 validate/process
        if (module && (typeof module.validate === 'function') && (typeof module.process === 'function')) {
          module = wrap(module)
          router.post('/' + route, module)
          router.post('/json/' + route, module) // 注册 json 类型的 SOA 请求入口
        }
        break
      case 'function': // 如果是个函数，则直接将这个函数做为路由的处理函数
        router.post('/' + route, module)
        router.post('/json/' + route, module)
        break
    }

  })

  app.use(vd, router)

  app.get(vd + '/_operationinfo', (req, res) => {
    res.json(operations)
  })
```

经过对服务端的改造以及技术栈的统一，整个团队的效能也得到了提升，用 Node.js 实现的接口在上线后性能稳定，整体耗时控制在 50ms 以内。

三、RestfulAPI->GraphQL

经过了前面用 Node.js 进行标准的 restfulAPI 开发尝试，有越来越多 Node.js 实现的接口上线，整个前台的架构如下：



在经历过几个版本迭代之后，我们发现了一些新的问题：

- 不同版本的客户端需求不同，相同的接口需要针对不同的版本做不同的处理；
- 不同的客户端对于契约的需求也不一样，比如 PC 由于屏幕尺寸的关系，在界面设计上给用户的信息要比 APP 多的多，PC 与 APP 在显示的信息上是有差异的，相同的契约数据下发对于某一端来说会存在浪费，从而加大网络开销，
- 在 APP 上也会存在着版本之间的差异，比如 7.15 的版本和 7.16 的版本，7.16 上了一些新的功能，加了一些新的 fetch，如果统一下发给前端，对于老的版本也是也是资源上的浪费，
- 客户端在某些时候需要调用多个接口汇总数据一起显示，某些情况下又要分开调用，对于服务来说，动态可扩展的架构尤为重要，
- 前端在 model 层使用的结构和服务端结构可能会存在差异性，如何磨平这些差异，也非常考验开发同学的技术能力；

在这个时候，GraphQL 进入到了我们的视野。GraphQL 是一种新的 API 标准，它提供一种更高效、更加灵活的数据查询方式，在 2015 年被 Facebook 正式开源。

其在本质上是一种基于 API 的查询语言，是对 restfulAPI 的一种封装，目的在于构建一种更加易用的服务，通过 GraphQL，客户端可以很方便的获取所需要的数据。

比如下面的这个例子，获取 ID 为 1 的城市信息，只要返回的 schema 保护 ID,name，Code 即可，其中 name 为重定义 schema，

Request:

```
{
  city: getCityInfo(id: 1) {
    ID
    name: Name
    Code
  }
}
```

Response:

```
{
  "data": {
    "city": {
      "ID": 1,
      "name": "北京",
      "Code": "BJS"
    }
}
```

Schema：

```
module.exports = new GraphQLObjectType({
  'name': 'CityInfo',
  'description': '城市实体',
  'fields': {
    'Code': {
      'description': '城市三字码',
      'type': GraphQLString
    },
    'ID': {
      'description': '城市 ID',
      'type': GraphQLInt
    },
    'Name': {
      'description': '城市名称',
      'type': GraphQLString
    }
  }
})
```

GraphQL 和传统的 restAPI 相比：

数据获取：GraphQL 可以按需获取，通过调用方指定 schema 返回不同报文，RestfulAPI 则是下发相同的结构；

类型系统，强校验：GraphQL 使用 Type System 来定义 API，公开的类型都是通过 SDL 模式进行编写，统一前后端契约结构，便于使用；

URL 入口：Rest 不同的请求入口不同，在请求的 URL 上需要做区分，GraphQL 则是一个入口(/graphql?query=)，通过调用的 request 来区分；

调用方式：Rest 获取多个不同接口数据时，需要并发调用多次，而 GraphQL 可以合并查询，降低网络开销；

于是我们开始在团队内部试点 GraphQL，在技术架构上采用 PM2+Node.js+Express+Express-GraphQL，选用 Express-GraphQL 作为核心中间件，统一客户端的请求入口为/graphql?query=*，由调用端来决定自己需要哪些数据。

四、总结

Node.js 在机票团队从早期的前后端分离到 GraphQL 的实践，目前已经深度应用到前端组的各个模块，现在机票前端应用层已全部采用 Node.js 来实现。

有近 20+接口采用 Node.js 来开发，其中一大半是通过 GraphQL 来实现，日均的流量在 200W 左右，整体 Node 服务端性能稳定，后续我们还将继续拓宽 Node.js 的使用场景，使其发挥更大的价值。

浅谈 React 数据流管理

【作者简介】颜陈宇，携程玩乐高级前端开发工程师，前端架构组成员，目前主要负责玩乐国际化项目的 App、H5 以及 Online 三端技术架构。热衷于 react 技术栈，喜欢阅读和分享。

引言

为什么数据流管理如此重要？react 的核心思想就是：UI=render(data)，data 就是我们说的数据，render 是 react 提供的纯函数，所以用户界面的展示完全取决于数据层。

这篇文章希望能用最浅显易懂的话，将 react 中的数据流管理，从自身到借助第三方库，将这些概念理清楚。我会列举几个当下最热的库，包括它们的思想以及优缺点，适用于哪些业务场景。

这篇文章不是教程，不会讲如何去使用它们，更不会一言不合就搬源码，正如文章标题所说，只是浅谈，希望读者在读完以后就算原先没有使用过这些库，也能大致有个思路，知道该如何选择性地深入学习。

在本文正式开始之前，我先试图讲清楚两个概念，状态和数据：

我们都知道，react 是利用可复用的组件来构建界面的，组件本质上是一个有限状态机，它能够记住当前所处的状态，并且能够根据不同的状态变化做出相应的操作。在 react 中，把这种状态定义为 state，用来描述该组件对应的当前交互界面，表示当前界面展示的一种状况，react 正是通过管理状态来实现对组件的管理，当 state 发生变更时，react 会自动去执行相应的操作：绘制界面。

所以我们接下来提到的状态是针对 react component 这种有限状态机。而数据就广泛了，它不光是指 server 层返回给前端的数据，react 中的状态也是一种数据。当我们改变数据的同时，就要通过改变状态去引发界面的变更。

我们真正要关心的是数据层的管理，我们今天所讨论的数据流管理方案，特别是后面介绍的几种第三方库，不光是配合 react，也可以配合其他的 View 框架（Vue、Angular 等等），就好比开头提到的那个公式，引申一下：UI =X(data)，但今天主要是围绕 react 来讲的，因此我们在说 react 的状态管理其实和数据流管理是一样的，包括我们会借助第三方库来帮助 react 管理状态，希望不要有小伙伴太纠结于此。

一、react 自身的数据流管理方案

我们先来回顾一下，react 自身是如何管理数据流的（也可以理解为如何管理应用状态）：

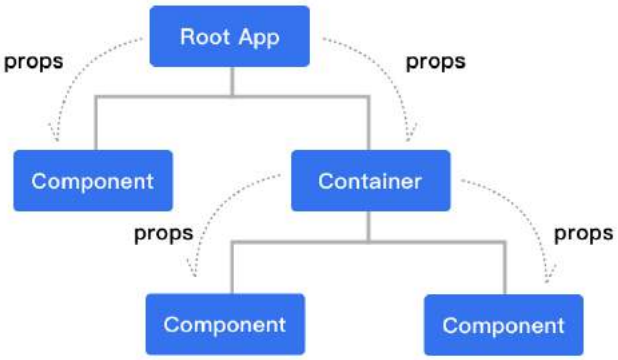
react 是自上而下的单向组件数据流，容器组件&展示组件（也叫傻瓜组件&聪明组件）是最常用的 react 组件设计方案，容器组件负责处理复杂的业务逻辑以及数据，展示组件负责处理 UI 层，通常我们会将展示组件抽出来进行复用或者组件库的封装，容器组件自身通过 state

来管理状态，setState 更新状态，从而更新 UI，通过 props 将自身的 state 传递给展示组件实现通信。

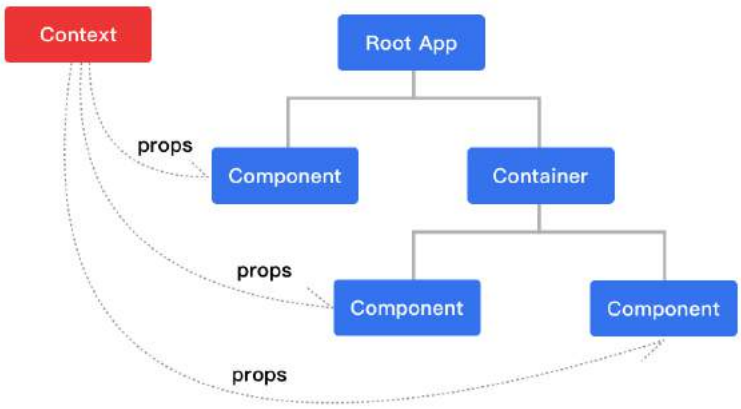
这是当业务需求不复杂，页面较简单时我们常用的数据流处理方式，仅用 react 自身提供的 props 和 state 来管理足矣，但是如果稍微增加一点复杂度呢，比如当我们项目中遇到这些问题：

1）如何实现跨组件通信、状态同步以及状态共享？

react V16.3 以前，通过状态提升至最近的共同父组件来实现。(虽然有官方提供的 contextAPI，但是旧版本存在一个问题：看似跨组件，实则还是逐级传递，如果中间组件使用了 ShouldComponentUpdate 检测到当前 state 和 props 没有变化，returnfalse，那么 context 就会无法透传，因此 context 没有被官方推荐使用)。



react V16.3 版本以后，新版本 context 解决了之前的问题，可以轻松实现，但依然存在一个问题，context 也是将底部子组件的状态控制交给到了顶级组件，但是顶级组件状态更新的时候一定会触发所有子组件的 re-render，那么也会带来损耗。(虽然我们可以通过一些手段来减少重绘，比如在中间组件的 SCU 里进行一些判断，但是当项目较大时，我们需要花太多的精力去做这件事)



2）如何避免组件臃肿?

当某个组件的业务逻辑非常复杂时，我们会发现代码越写越多，因为我们只能在组件内部去控制数据流，没办法抽离，Model 和 View 都放在了 View 层，整个组件显得臃肿不堪，业务逻辑统统堆在一块，难以维护。

3）如何让状态变得可预知，甚至可回溯?

当数据流混乱时，我们一个执行动作可能会触发一系列的 setState，我们如何能够让整个数据流变得可“监控”，甚至可以更细致地去控制每一步数据或状态的变更?

4）如何处理异步数据流?

react 自身并未提供多种处理异步数据流管理的方案，仅用一个 setState 已经很难满足一些复杂的异步流场景;

如何改进?

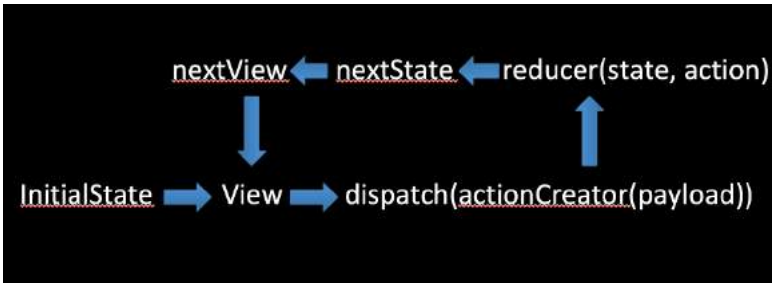
这个时候，我们可能需要一个真正的数据流管理工具来帮助 react 了，我们希望它是真正脱离 react 组件的概念的，从 UI 层完全抽离出来，只负责管理数据，让 react 只专注于 View 层的绘制。

那这也是为什么我们需要使用那些第三方数据流管理工具的原因，接下来我们就来了解一些当前社区比较热门的数据流管理工具。

二、redux

我直接跳过了 flux 来说 redux，主要是因为 redux 是由 flux 演变而来，可以说是 flux 的升级加强版，flux 具备的优势 redux 也做到了。

redux 提供了哪些?



1）store：提供了一个全局的 store 变量，用来存储我们希望从组件内部抽离出去的那些公用的状态;

2) action: 提供了一个普通对象，用来记录我们每一次的状态变更，可日志打印与调试回溯，

并且这是唯一的途径；

3）reducer：提供了一个纯函数，用来计算状态的变更；

为什么需要 redux?

很多人在用了一段时间的 redux 之后，最大的感想就是，redux 要写大量的模板代码，很麻烦，还不如只用 react 来管理。特别是在 react 的新 context 推出以后，许多人更是直接弃用了 redux，甚至觉得 redux 已死。如果说旧版的 context 的弊端，我们通过 redux 配合 react-redux 来实现跨组件的状态通信同步等问题，那确实新版本的 context 可以替换掉这个功能点，但如果你的项目中仅仅是用 redux 做这些，那思考一下，你是否真的需要 redux？也许从一开始你就不需要它。

（虽然新版的 context 功能强大，但是依然是通过一个新的容器组件来替我们管理状态，那么通过组件管理状态的问题依旧会存在，Consumer 是和 Provider 一一对应的，在项目复杂度较高时，可能会出现多个 Provider，更多个 Consumer，甚至会一个 Consumer 需要对应多个 Provider 的传值等一系列复杂的情况，所以我们依然要谨慎使用）

redux 的核心竞争力

1）状态持久化：globalstore 可以保证组件就算销毁了也依然保留之前状态；

2）状态可回溯：每个 action 都会被序列化，Reducer 不会修改原有状态，总是返回新状态，方便做状态回溯；

3）Functional Programming：使用纯函数，输出完全依赖输入，没有任何副作用；

4）中间件：针对异步数据流，提供了类 express 中间件的模式，社区也出现了一大批优秀的第三方插件，能够更精细地控制数据的流动，对复杂的业务场景起到了缓冲地作用；

与其说是 redux 来帮助 react 管理状态，不如说是将 react 的部分状态移交至 redux 那里。redux 彻头彻尾的纯函数理念就表明了它不会参与任何状态变化，完全是由 react 自己来完成。只不过 redux 会提供一套工具，react 照着说明书来操作罢了。

所以这注定了想要接受 redux，就必须按照它的规矩来做，除非你不愿意接受这种 FP 的模式。这种模式有利有弊，有利就是在一个大型的多人团队中，这种开发模式反而容易形成一种规约，让整个状态流程变得清晰，弊端就是对于小规模团队，尤其是着急发布上线的，这种繁重的代码模板无疑是一种负担。

redux 的缺点：

1）繁重的代码模板：修改一个 state 可能要动四五个文件，可谓牵一发而动全身；
2）store 里状态残留：多组件共用 store 里某个状态时要注意初始化清空问题；
3）无脑的发布订阅：每次 dispatch 一个 action 都会遍历所有的 reducer，重新计算 connect，这无疑是一种损耗；
4）交互频繁时会有卡顿：如果 store 较大时，且频繁地修改 store，会明显看到页面卡顿；
5）不支持 typescript；

关于如何优化，网上有很多优秀的案例，redux 官方也提供了很多方法，这里不再赘述。redux 未来不会有太大的变化，那些弊端还是会继续保留，但是这依然不会妨碍忠爱它的用户去使用它。

如果说 redux 那种强硬的函数式编程模式让很多人难以接受，那么当他们开始 mobx 的使用的时候，无疑是一种解脱。

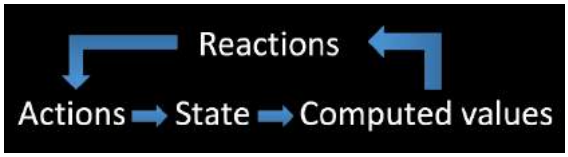
三、mobx

最开始接触 mobx 也是因为 redux 作者 DanAbramov 的那句：Unhappywith redux？try mobx，我相信很多人也是因为这句话而开始了解学习并使用它的。

下面列举一些 mobx 的优势（和 redux 进行一个对比）

1）redux 不允许直接修改 state，而 mobx 可随意修改；
2）redux 修改状态必须走一套指定的流程较麻烦，而 mobx 可以在任何地方直接修改（非严格模式下）；
3）redux 模板代码文件多，而 mobx 非常简洁，就一个文件；
4）redux 只有一个 store，state orstore 难以取舍，而 mobx 多 store，你可以把所有的 state 都放入 store 中，完全交给 mobx 来管理，减少顾虑；
5）redux 需要对监听的组件做 SCU 优化，减少重复 render；而 mobx 都是 SmartComponent，不用我们手动做 SCU；

mobx 的设计思想：



说了这么多，如果你是第一次了解 mobx，是不是听着就感觉很爽！没错，这就是 mobx 的魅力，那它是如何实现这些功能的呢？

这里以 mobx 5 版本为例，实际上它是利用了 ES6 的 proxy 来追踪属性（旧版本是用

Object.defineProperty 来实现的）通过隐式订阅，自动追踪被监听的对象变化，然后触发组件的 UI 更新。

如果说 redux 是把要做的事情都交给了用户，来保证自己的纯净，那么 mobx 就是把最简易的操作给了用户，其它的交给 mobx 内部去实现。用户不必关心这个过程，Model 和 View 完全分离，我们完全可以将业务逻辑写在 action 里，用户只需要操作 Observabledata 就行了。

Observerview 会自动做出响应，这就是 mobx 主打的响应式设计，但是编程风格依然是传统的面向对象的 OO 范式。（熟悉 Vue 的朋友一定对这种响应式设计不陌生，Vue 就是利用了数据劫持来实现双向绑定，其实 React +Mobx 就是一个复杂点的 Vue，Vue 3 版本一个重大改变也是将代理交给了 proxy）

刚刚 mobx 的优势说得比较多了，这边再总结一下：

1）代码量少；
2）基于数据劫持来实现精准定位（真正意义上的局部更新）；
3）多 store 抽离业务逻辑（Model View 分离）；
4）响应式性能良好（频繁的交互依然可以胜任）；
5）完全可以替代 react 自身的状态管理；
6）支持 typescript；

但是 mobx 真的这么完美吗，当然也有缺陷：

1）没有状态回溯能力：mobx 是直接修改对象引用，所以很难去做状态回溯；（这点 redux 的优势就瞬间体现出来了）

2）没有中间件：和 redux 一样，mobx 也没有很好地办法处理异步数据流，没办法更精细地去控制数据流动；（redux 虽然自己不做，但是它提供了 applyMiddleware！）

3）store 太多：随着 store 数的增多，维护成本也会增加，而且多 store 之间的数据共享以及相互引用也会容易出错

4）副作用：mobx 直接修改数据，和函数式编程模式强调的纯函数相反，这也导致了数据的很多未知性

其实现在主流的数据流管理分为两大派，一类是以 redux 为首的函数式库，还有一类是以 mobx 为首的响应式库，其实通过刚刚的介绍，我们会发现，redux 和 mobx 有一个共同的短板，那就是在处理异步数据流的时候，没有一个很好的解决方案，至少仅仅依靠自身比较吃力，那么接下来给大家介绍一个处理异步数据流的高手：rxjs。

四、rxjs

我相信很多人听说过 rxjs 学习曲线异常陡峭，是的，除了眼花缭乱的各类操作符（目前 rxjs

V6 版本有 120+个），关键是它要求我们在处理事务的时候要贯彻“一切皆为流”的理念，更是让初学者难以理解。

这一小节并不能让读者达到能够上手使用的程度，正如文章开头所说，希望读者（新手）能对 rxjs 有一个新的认知，知道它是做什么的，它是如何实现的，它有哪些优势，我们如何选择它，如果感兴趣还需要私下花大量时间去学习掌握各种操作符，但我也会尝试尽可能地相对于前两个说得更细致一些。

在开始介绍 rxjs 之前，我们先来简单地聊聊什么是响应式编程？我以一个很简单的小例子来看：a + b = c。

如果站在传统的命令式编程的角度来看这段公式：c 的值完全依赖于 a 和 b，这时候我们去改变 a 的值，那我们就需要再去手动计算 a + b 的值，a、b 和 c 是相互依赖的。

那么如果站在响应式编程的角度来看，这个公式又会变成这样：c := a + b，a 和 b 完全不关心 c 的值，c 也完全不关心等式那边是 a 或者 b，或者还有什么 d，e，f。。。等式右边改变值了，左边会自动更改数值，这就是响应式编程的思维方式。

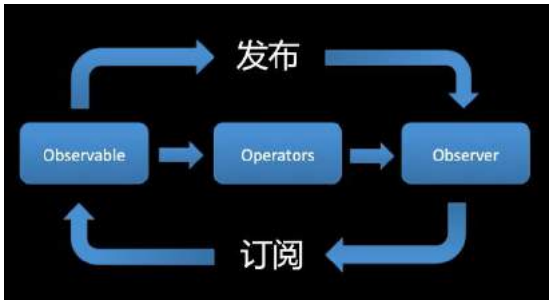
我们再来看前端的框架历史，传统命令式编程的代表：jQuery，在过去我们是如何绘制一个页面的？我们会用 jQuery 提供的一套 API，然后手动操作 Dom 来进行绘制，很精准，但是很累，因为完全靠手动操作，且改动时性能损耗较大，开发者的注意力完全在“如何去绘制”上面了。

那我们再来看响应式编程的 react，它是如何来实现的？

开发者根本不用关心界面如何绘制，只要告诉 react 我们希望页面长什么样子，就可以了，剩下的交给 react，react 就会自动帮我们绘制界面，还记得开头时的那个核心思想吗：UI =render(data)，我们只要操作 data 就可以了，页面 UI 会自动作出响应，而且我们一切的操作都是基于内存之中，不会有较大的性能损耗，这就是 react 响应式编程的精髓，也是为何它叫作 react。

回到我们的 rxjs 上，rxjs 是如何做到响应式的呢？多亏了它两种强大的设计模式：观察者模式和迭代器模式；简单地介绍一下：

1）观察者模式：



在观察者模式中，有两个重要的角色：Observable 和 Observer，熟悉 mobx 的同学对这个一定不陌生(所以我建议想要学习 rxjs 的同学, 如果对 mobx 不熟悉, 可以先学习一下 mobx, 然后再学习 rxjs，这样会更容易理解一些)。

就是可观察对象和观察者，可观察对象（Observable）也就是事件发布者，负责产生事件，而观察者（Observer）也就是事件响应者，负责对发布的事件作出响应，但是如何连接一个发布者和响应者呢?

通过订阅的形式，也就是 subscribe 方法（这也类似于 redux 的 store.subscribe），而在订阅之前，他们两者是毫无关联的，无论 Observable 发出多少事件，Observer 也不会做出任何响应，同样，当这种订阅关系中断时也不会。

2）迭代器模式:

在这里要先引出一个新的概念：拉取（pull）和推送（push），rxjs 官方这两种协议有更详细的解释，我这里就直接引用一下:

拉取 (Pull) vs. 推送 (Push)

拉取和**推送**是两种不同的协议，用来描述数据**生产者 (Producer)**如何与数据**消费者 (Consumer)**进行通信的。

什么是拉取? - 在拉取体系中，由消费者来决定何时从生产者那里接收数据。生产者本身不知道数据是何时交付到消费者手中的。

每个 JavaScript 函数都是拉取体系。函数是数据的生产者，调用该函数的代码通过从函数调用中"取出"一个**单个**返回值来对该函数进行消费。

ES2015 引入了 generator 函数和 iterators（`function*`），这是另外一种类型的拉取体系。调用 `iterator.next()` 的代码是消费者，它会从 iterator(生产者) 那"取出"**多个**值。

	生产者	消费者
拉取	**被动的:** 当被请求时产生数据。	**主动的:** 决定何时请求数据。
推送	**主动的:** 按自己的节奏产生数据。	**被动的:** 对收到的数据做出反应。

什么是推送? - 在推送体系中，由生产者来决定何时把数据发送给消费者。消费者本身不知道何时会接收到数据。

在当今的 JavaScript 世界中，Promises 是最常见的推送体系类型。Promise(生产者) 将一个解析过的值传递给已注册的回调函数(消费者)，但不同于函数的是，由 Promise 来决定何时把值"推送"给回调函数。

RxJS 引入了 Observables，一个新的 JavaScript 推送体系。Observable 是多个值的生产者，并将值"推送"给观察者(消费者)。

拉取和推送实际上对于观察者来说就是一个主动与被动的区别, 是主动去获取还是被动地接收。在 rxjs 中，作为事件响应者（消费者）的 Observer 对象也有一个 next 属性（回调函数），用来接收从发布者那里"推"过来的数据。

（站在开发者的角度，我们一定是希望消息是被动地接收，因为我们倡导的就是通过操作 data 数据层，让 View 层进行一个响应，那么这里 data 数据层一定是事件发布者，而 View 层就是事件响应者，每当 data 数据层发生变化时，都会主动推送一个值给 View 层，这才符合真正意义上的响应式编程，而 rxjs 做到了!）

如何配合 react?

如果说 redux 和 mobx 的出现或多或少是因为 react 的存在，那么不同的是 rxjs 和 react 并没有什么关联，关于 rxjs 的历史这里不多说，感兴趣的可以了解一下 ReactiveExtension，rxjs 只是响应式编程在 JavaScript 中的应用。

那么如何帮助 react 实现状态管理呢，我们只需要将组件作为事件响应者，然后在 next 回调

里定义好更新组件状态的动作 setState，当接收到数据推送时，就会自动触发 setState，完成界面更新，这其实有点类似于 mobx 做的事情。(很多人在 react 项目中并没有完全只使用 rxjs，而是用了这个 redux-observable 中间件，利用 rxjs 的操作符来处理异步 action)

除了响应式编程的魅力，rxjs 还有什么优势呢?

1）纯函数：rxjs 中数据流动的过程中，不会改变已经存在的 Observable 实例，会返回一个新的 Observable，没有任何副作用;

2）强大的操作符：rxjs 又被称为 lodash forasync，和 lodash 一样，拥有众多强大的操作符来操作数据流，不光是同步数据，特别是针对各种复杂的异步数据流，甚至可以多种事件流组合搭配，汇总到一起处理;

3）更独立：rxjs 并不依赖于任何一个框架，它可以任意搭配，因为它的关注点完全就是在于数据流的处理上，而且它更偏底层一些

那 rxjs 有哪些缺点呢?

1）学习曲线陡峭：光是这一点就已经让大多数人止步于此;

2）事件流高度抽象：用 rxjs 的用户反馈一般都是两种极端情况，用得好的都觉得这个太厉害了，用得不好的都觉得感觉有点麻烦，增加了项目复杂度。

五、结语

最后，总结一下各类的适用场景:

1）当我们项目中复杂程度较低时，建议只用 react 就可以了;
2）当我们项目中跨组件通信、数据流同步等情况较多时，建议搭配 react 的新 context api;

3）当项目复杂度一般时，小规模团队或开发周期较短、要求快速上线时，建议使用 mobx;
4）当项目复杂度较高时，团队规模较大或要求对事件分发处理可监控可回溯时，建议使用 redux;
5）当项目复杂度较高，且数据流（尤其是异步数据）混杂时，建议使用 rxjs;

其实回顾全篇，我没有提到一个关键点是，各个库的性能对比如何。其实它们之间一定是有差异的，但是这点性能差异，相对于 react 自身组件设计不当而导致的性能损耗来说，是可以忽略的。

如果你现在的项目觉得性能较差或者页面卡顿，建议先从 react 层面去考虑如何进行优化，然后再去考虑如何优化数据管理层。关于上面提到的三个数据流管理工具，有利有弊，针对弊端，网上也有一大批优秀的解决方案和改进，感兴趣的读者可自行查阅。

30+业务团队，携程无线发布如何做到稳定高效

【作者简介】王雪松，携程技术管理中心 PMO 高级项目经理，主要从事携程技术中心跨 BU 项目集的管理工作。自 2016 年起负责携程主板 app 的项目协调、流程梳理、集成发布，并兼任无线技术委员会助理，负责无线端相关技改项目的推进及对 BU 支持等协调工作。

携程自 2010 年 10 月发布无线战略，到现在 app 已有 8 年左右的发展历史。早期的无线事业部，统一管理 app 从业务需求到研发到发布的整个过程。

2013 年公司推出“拇指+水泥”战略，大力发展无线。2015 年对原统一的无线管理架构进行了调整，将团队拆分到各业务线，此举可称之为携程无线的破和立。

无线团队拆分到各业务线后，加上后续新增业务线，目前涉及到的业务团队大概有 30 条左右，团队人员也很多，对 app 的集成发布形成了不小的挑战。

目前携程的无线发布实践是怎样的呢，本文将重点分享携程主板 app 发布实践。

一、组织架构

2017 年起，携程组建了各垂直领域的技术委员会。无线委员会主要由无线平台和各业务线无线同学组成。作为虚拟组织对垂直技术领域做统一管理，响应技术领域的技术战略、发展方向，新技术论证落地，鼓励技术创新，制定技术规范制订，开展技术培训等。

平台研发中心的无线平台团队更多承担着无线框架、技术创新及对业务线支持的任务，内部也称之为“无线公共团队”。

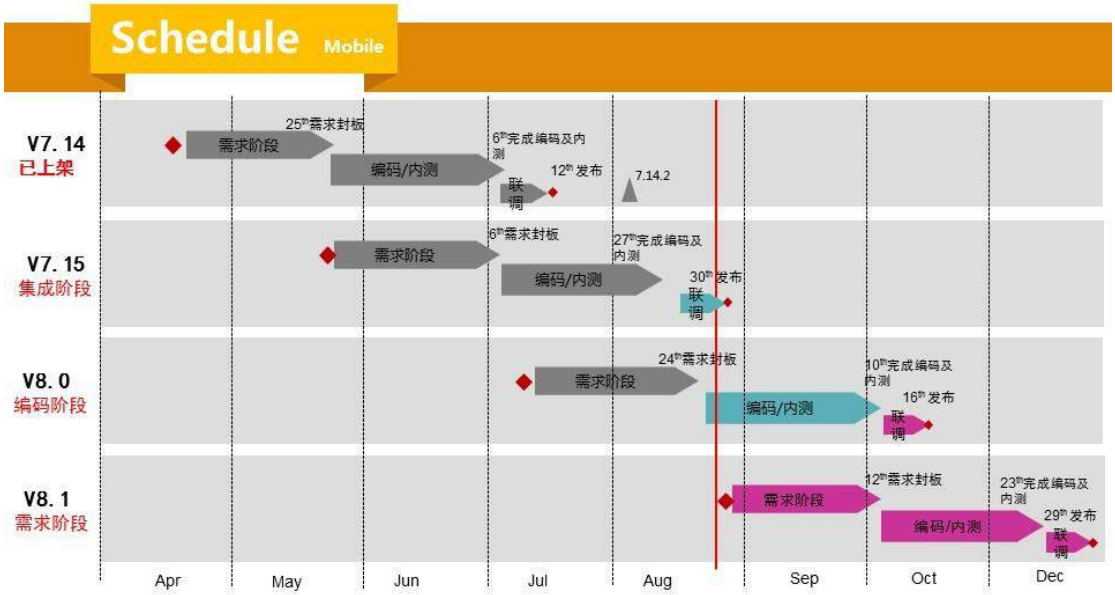
技术管理中心 PMO 更多承担携程技术中心跨业务线的项目管理、SQA、流程等支持。

市场主要负责 App 在市场商务方面的工作，类似 app 上架计划、预装包、渠道包等等。

二、发布流程介绍

流程要点说明：

1、发版计划：发版计划分为大版本和小版本，大版本一般提前半年制订发版计划并通知到业务线，大版本会综合考虑业务线迭代周期及节假日等情况，小版本按需（用途：bug fix 等）穿插在大版本间发布。发版计划主要由市场部和技术管理中心 PMO 负责。大版本迭代图如下：



2、需求部分：框架公共类需求比如大首页宫格分布及入口地址调整等，由无线平台产品团队负责管理，业务线提申请。业务线需求主由各业务线自行管理，跨业务线需求各自协商，公共类的技改有专门的项目立项来推动。

3、迭代发布：目前各业务线迭代周期在 2~3 周左右，各业务线包括平台公共无线框架，业务需求发布和框架类更新发布，都会要求在规定时间内完成测试和发布，进入最后的全业务集成测试。

4、业务线测试：指业务线开发或测试同学内部功能测试，测试通过后可以 release，即可进入全团队集成阶段。集成工具 MCD 支持业务线按需编译和打包。

5、全业务集成测试：全员使用集成包测试（集成包是指集成了所有业务线 release 的功能）。要求各业务在此之前完成内部验证测试，并 release bundle，未 release 的最新功能将不会进入集成包。

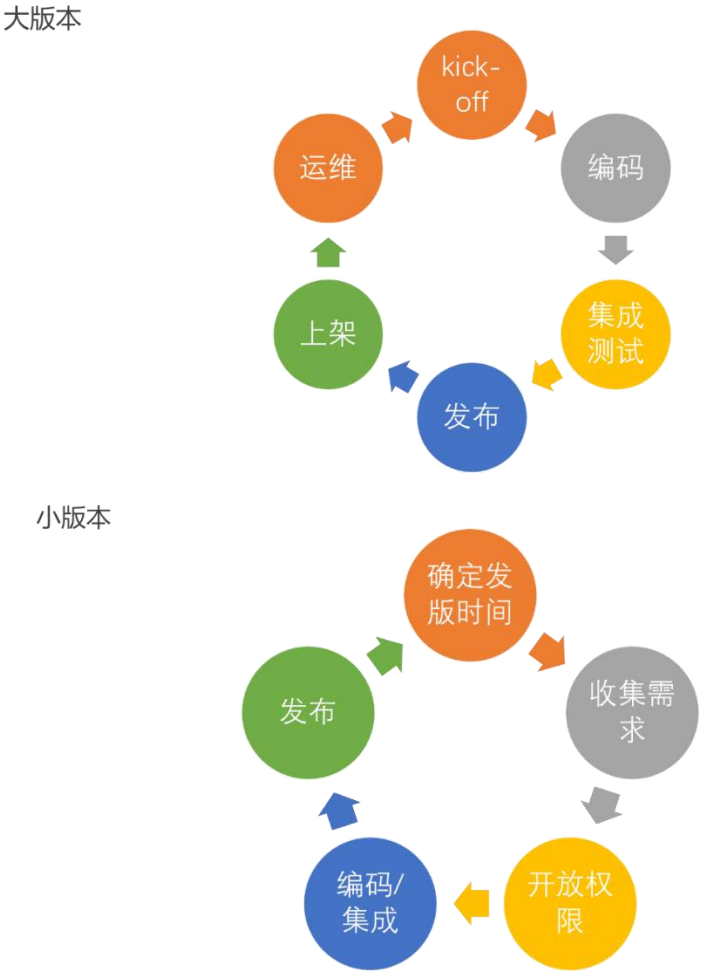
6、Code freeze 封板：为保证发布效率，避免开发后期的改动风险，会在集成发布最后阶段做 code freeze，我们内部也称之为“封板”，封板后出最终包，给到全业务线做最后的测试确认。

7、定版：就封板后的 app 集成包，如全团队测试通过后（需各团队测试负责人在 MCD 确认），我们定版 launch，并在 MCD 标识进入后续渠道包制作等流程。

8、上架：定版后，公共平台团队会处理相应的渠道包和提交审核等工作，市场同学负责各应用市场的上架弹窗等。

9、质量：各业务线 QA 负责，集成期间监控 issue 收敛情况【Jira 平台】。

10、运维阶段，主要指 bug fix、hotfix 等发布相关，均需按相应的流程申请及发布。比如小版本发布，小版本主要以修复大版本 bug 为主。目前采取“搭车需求”模式，即发动小版本车次，业务线提交需求申请，申请通过的开放代码权限。后续开发、发布流程同大版本。



三、工具方面

从工具方面来看，目前无线方面使用到的工具比较多，主要在编译发布、持续集成、日志监控、性能优化、AB 分流、自动化测试等方面。本文重点介绍下集成发布相关工具。

2015 年开始，无线平台团队自研了 MCD（Mobile Continuous Delivery）平台，经过不断实践调整优化，到目前提供了持续集成、编译打包、扫码安装、冒烟测试、白屏检测、size 分析、crash 收集分析、灰度、hotfix 等丰富功能，可以说是目前携程无线的一大利器，极大帮助提升了无线集成发布的效率。

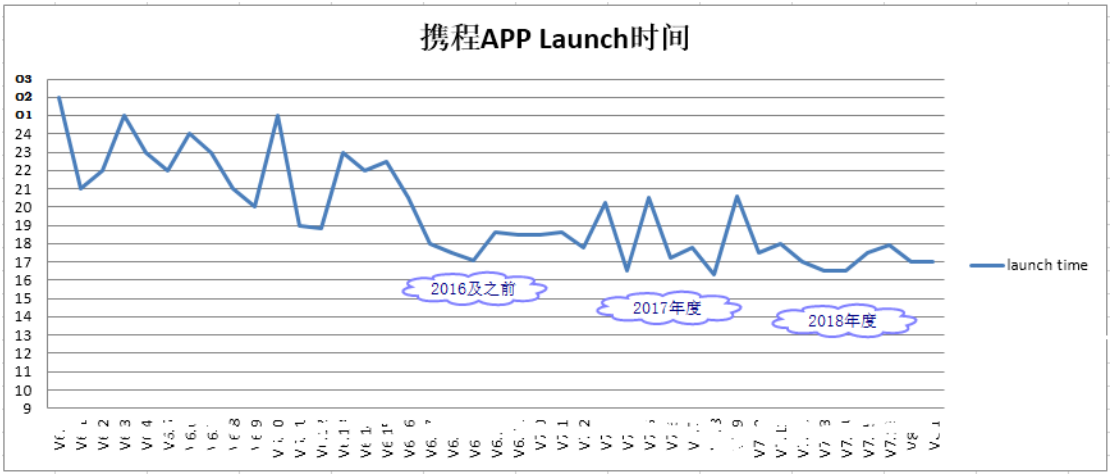
平台涵盖了 app 集成、测试、发布、运营四大阶段。17 年起支持插件化，实现业务解耦，缩短编译时间、减少编译依赖堵塞等问题。所有 BU 业务模块 bundle 化，并辅以 bundle 颗粒度 RC 发布模式，全面支持从项目创建、各业务开发、测试、bundle 发布、集成发布、测试确认的 app 全生命周期管理。

测试阶段，提供白屏检测、远程设备租用、代码质量（结合 sonar）、二维码扫码安装等功能。发布阶段，MCD 还支持 Hybrid、ReactNative 等测试、发布、灰度、下发监控、下发回滚等。运营阶段，支持 app size 分析、崩溃采集、发布记录查询、发布包查询、下发配置、版本占比等运营数据统计。

MCD 目前已全面支持其他独立 app、小程序等发布流程。

四、小结

目前携程无线发布，经过流程梳理、实战打磨、工具利器、集团作战，已形成一套“快而稳”的体制，发布效率高效透明。以下是之 2016 年以来的发布 launch 趋势图。



整体发布流程已经在上面说明，个人认为对发布比较重要的几个点：

1、组织保证：一个高凝聚力的委员会，强大的无线公共服务团队及业务线无线骨干，他们好比是汽车的发动机，给无线技术框架的优化输出源源不断的动力，保证我们无线技术的先进性和实用性。此外，我们也会定期组织技术分享、沙龙等，以一种交流和学习的态度保持与业界的沟通。

2、工具利器：一个高效、信息透明的发布平台对集成发布效率的提升具有非常重要的作用。

需要支持 CI\CD、多技术栈发布、高速编译出包、流程扼要、信息透明等特点。

3、全流程把控，集团作战：目前各业务开发发布流程透明可见，同时高度统一管控发版计划和最后的集成发布阶段。个人认为可以称之为“集团作战”，PMO 作为总指挥所或总枢纽，发出战斗打响号角（集成开始），发布战情（出包啊，家有问题啊），各业务线作战单位自行战斗并及时向指挥所反馈情况（反馈集成情况、确认测试结果），最后指挥所汇总战情，宣布战斗结束（launch）。

4、集成测试周期：从经验来看，集成测试周期长短可能会一定程度影响发布效率，建议是结合企业实际情况，逐步调整改进。携程这边几年来有过几次调整，目前周期也是长期运行调整目前可能比较符合的一个周期。

5、封板：在最后的集成测试阶段，往往因为业务需求的调整而出现开发临近发布还在 commit 的情况，大家都能理解往往最后阶段的代码调整可能带来是质量隐患甚至是巨坑，这也是往往发布 delay 的原因之一。所以我们在 16 年引入了“封板”，做 code freeze。刚开始业务线不太习惯封板，也出现很多次封板延迟的情况，慢慢地也习惯了，需求端开发端都熟知了这个规则也就顺了。

6、沟通：无论从发版计划的调研制定、到最后的定版发布，各环节都离不开沟通。最后集成阶段，PMO 会每天早上邮件发出当天早上编译的集成包（当然业务线也可去 MCD 上随需拿包），并同时会在内部沟通 IM 平台（cchat）广播，全业务线测试同学发现的问题、需要协调找人、问题修复等都可以在群里沟通或广播。

7、坚守原则：因为 app 发布涉及到 30 个左右业务团队，为了确保“集团作战”的效应，在整个发布过程中，对于重要原则必须“严守”。因为一旦某些关键节点“放松”，可能会导致整个发布流程效率降低。这也是 PMO 作为“第三方监管”的职责所在。

原则大家都遵守后，再加上各业务线的敏捷开发、需求封板、代码封板等机制，整体 app 发版流程清晰透明，大家节奏一致，整体发布效率自然也就趋于稳定和高效。

以上是携程无线发布的一些分享，希望对各位小伙伴有所帮助。

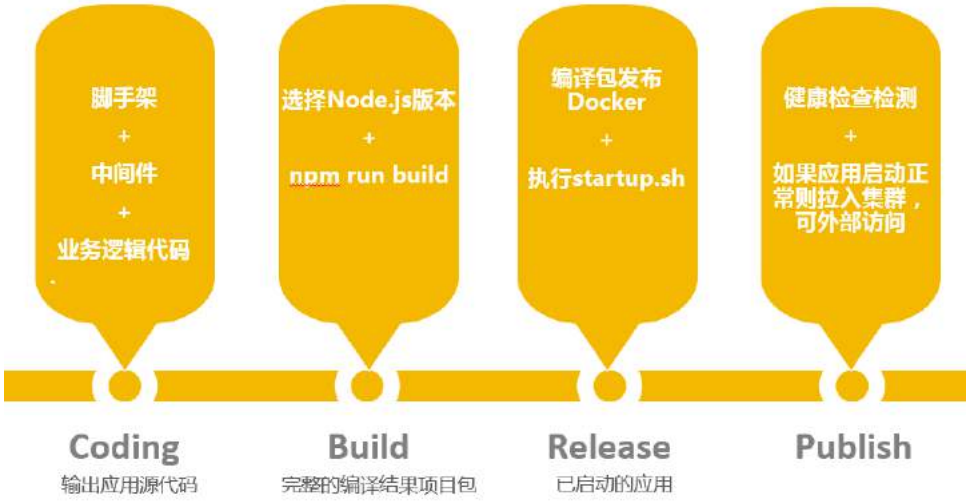
浅谈 Node. js 在携程的应用

【作者简介】潘斐斐，携程无线平台研发部高级研发工程师。2008 年加入携程，目前负责携程 Node.js 技术栈的基础平台研发工作。

携程在 2017 年 9 月份正式上线了 Node.js 应用，本文主要介绍近两年 Node.js 技术栈在携程的应用和体系情况。

一、技术栈

1.1 应用部署



应用部署主要分为以上四个步骤：Develop-> Build ->Release -> Publish

- Coding 阶段会使用脚手架和中间件开发应用，中间件后面会介绍到。
- Build Docker 会负责源码的构建功能，包括一些 C++模块的编译和集成环境，同时会设置构建的缓存机制。
- Release Docker 负责应用的启动和运行，相对轻量级，更需要关注的是 Docker 的数量、CPU、内存等基础信息。
- Publish 负责应用启动之后的健康检查，健康检查完成之后会将 Docker 拉入集群并提供外部访问。

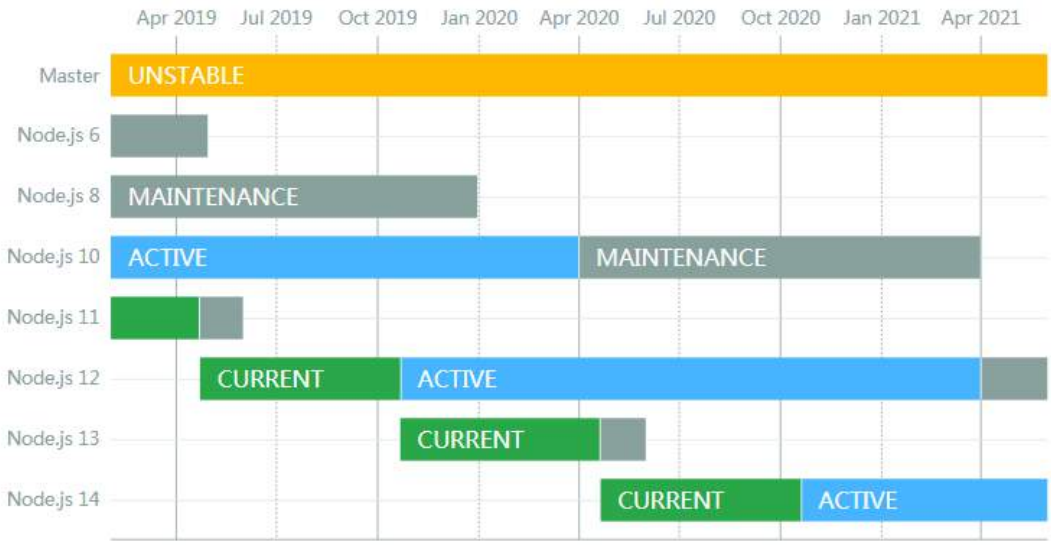
1.2 版本选择

在 Build 阶段，会选择 Node.js 的版本。提供了三个固定版本分别是 v6.10.2，v8.9.4 和 v10.15.1。目前 v6.10.2 版本已经基本进入非维护阶段，并逐步更新下线，版本的更迭与 Node.js 官方几乎保持同步，为保证性能最优，推荐优选最新的 LTS。

当时选择 Node.js 固定版本是考虑到编译环境的简单和稳定性。Node.js 中间件和第三方库

都需要做预编译，为了保证编译环境的简单和应用稳定，会选择固定的某一个版本。

同时针对这 3 个固定的版本，中间件发布的时候，也会一并提供 window/linux/mac 这 3 个平台预编译的包。Linux 预编译包是为了 Build Docker 和 Release Docker 准备的，windows 和 mac 预编译的包是为了开发工程师本地开发的时候准备的。



1.3 构建原则

“靠前构建原则”

如果能在线下编译的尽量线下编译，不要在运行构建。例如：

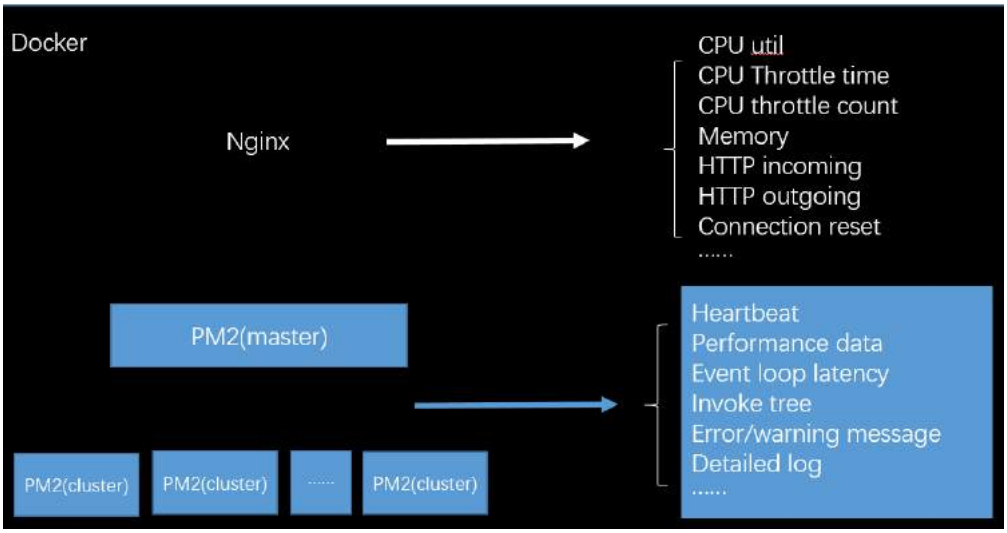
- C++模块的预编译
- 访问 SOA 或者数据库的环境配置
- Babel 或者 TS

二、运维与监控

2.1 Docker 化

Node.js 应用部署在 Docker 上，采用 Nginx+PM2 的模式。

2.2 核心指标



Nginx 会监控整个 Docker 上所有应用的情况：

1）CPU util：CPU 总的使用率

2）CPU throttle count&time：CPU 被限制的次数和 CPU 使用率被限制的总时间。这两个指标的上升一般表示应用有 CPU 密集型操作，需要检查一下是否有大量的计算等操作。

3）Mem RSS used：这个指标上升一般显示应用内存泄漏的问题。

4）HTTP imcoming&outgoging：http request 的数量变化趋势。如果有错误响应或者超过了告警的阈值，则会在趋势图中显示。

5）Connection reset：这个指标如果上升，表示应用出现了大量的拒绝请求，例如是服务器的并发数超过了原本的承载量等原因。

Nginx 中监控的是整个 Docker 的情况，但是我们更需要的是监控应用的指标。

应用一般采用 PM2 cluster – i max 模式启动，最大化利用 CPU。

1）Heartbeat（心跳信息）

每个 worker 一分钟发送一次 Heartbeat（心跳信息）给到 CAT 数据中心。一般来说，如果 Heartbeat 告警的话，需要立刻查看一下错误日志，是不是有异常错误导致进程已经退出了。

Heartbeat 主要包括 CPU、Memory、网络信息等。这些信息和上述提到的 Nginx 信息不是一个维度的。这个更细节的关注了应用的情况，而不是整个 Docker 的情况。如果需要分析应用细节的问题，是需要查看这里的 Heartbeat 信息。

2）性能情况

一般来说，中间件会处理应用常规的性能日志记录。包括：

- 每一个响应的请求耗时（服务端逻辑处理耗时，不包括网络耗时）
- 每一个 Transaction 的耗时。一个 Transaction 可以简单理解为一个有功能意义的代码片段。
- 跨应用调用的请求耗时

3）错误/告警信息

错误告警信息是应用中需要重点关注的，包括：

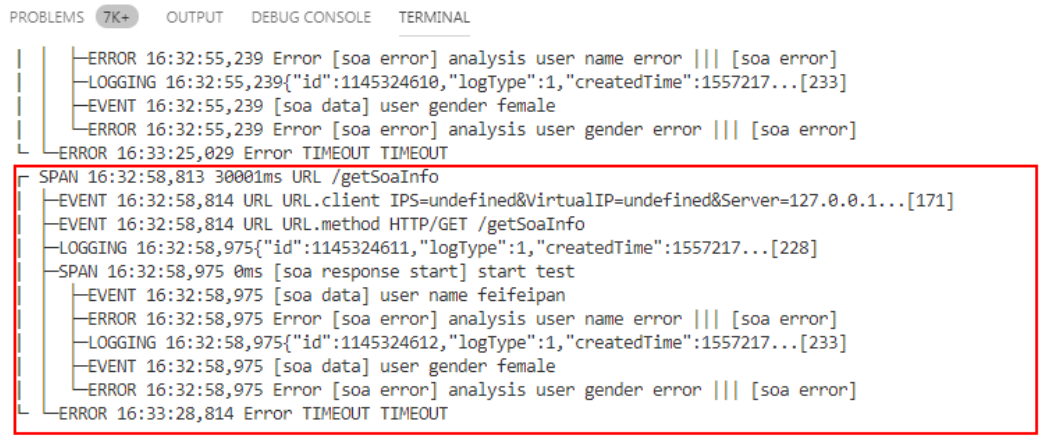
- 应用逻辑出错，例如处理 JSON 数据出错等。
- HTTP 请求出错，会记录状态码、请求地址、返回内容
- 应用中使用了不同版本的同一个包，会报一条告警信息通知开发工程师

4）详细数据日志

详细数据日志一般有开发工程师针对应用的逻辑埋点，而非中间件统一处理。这些日志会包括返回数据的记录，具体运行在哪一段 transaction 中。这些日志一般是故障发生时，用来复盘时的辅助手段。

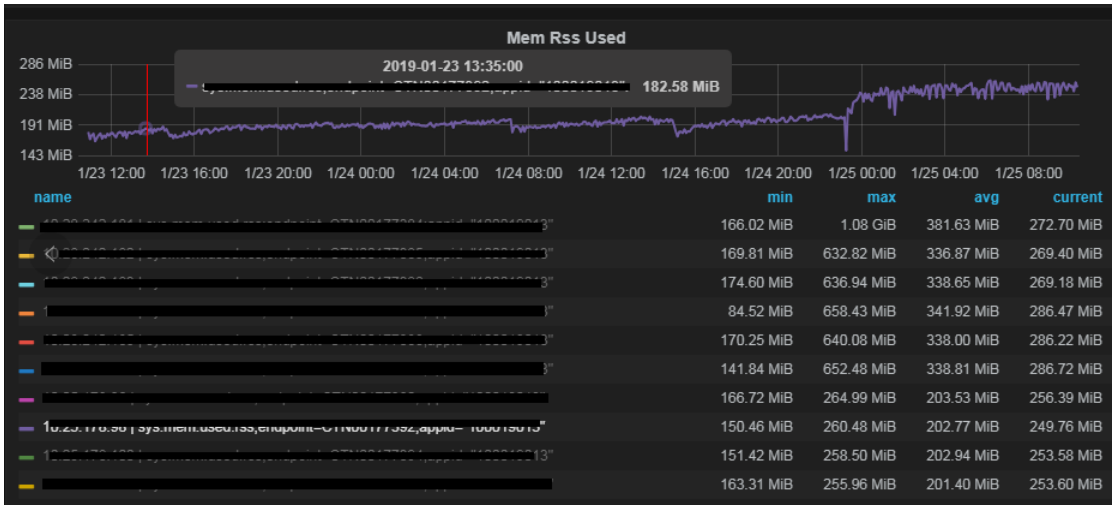
2.3 监控模型

一开始的监控日志是扁平化的，只能看到一条一条简单的日志，但无法将他们的关系串联起来。为了方便排障，设计了调用树的模型，可以将应用中的多个 transaction 串联起来。



2.4 日志排障

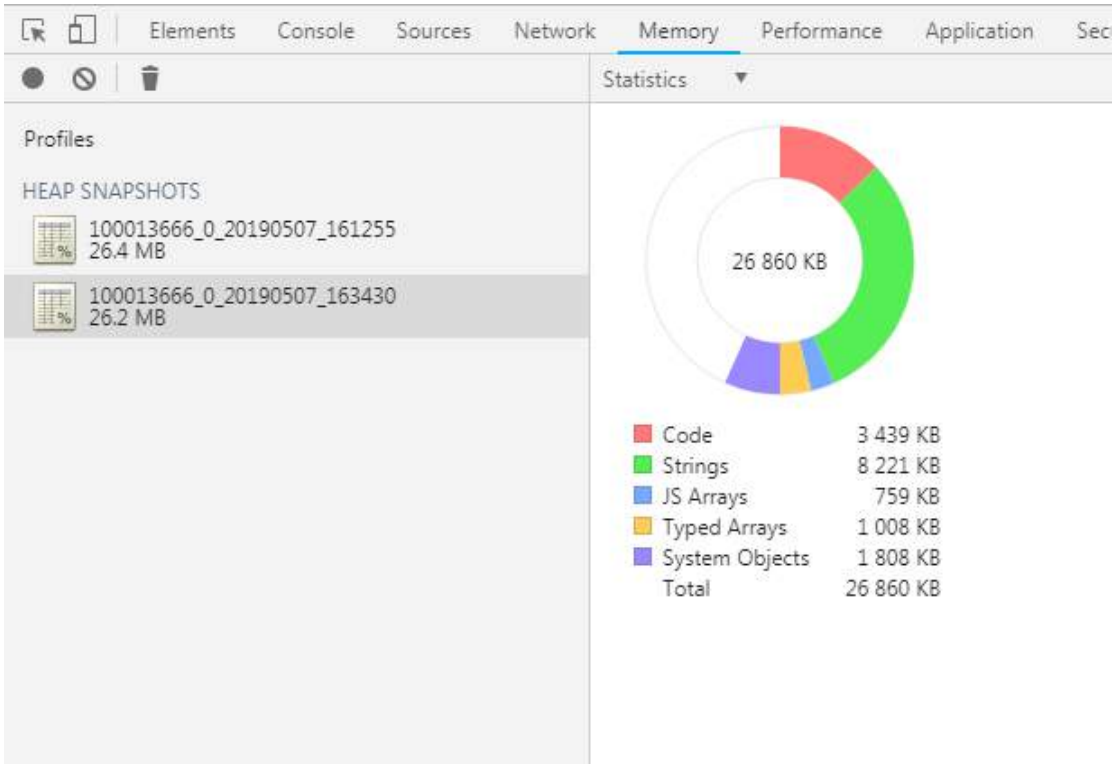
应用发布上线后，需要关注系统的保障通知。经常遇到的故障是发现随着时间的推移，Mem RSS Used 这根线会不停的飙升。



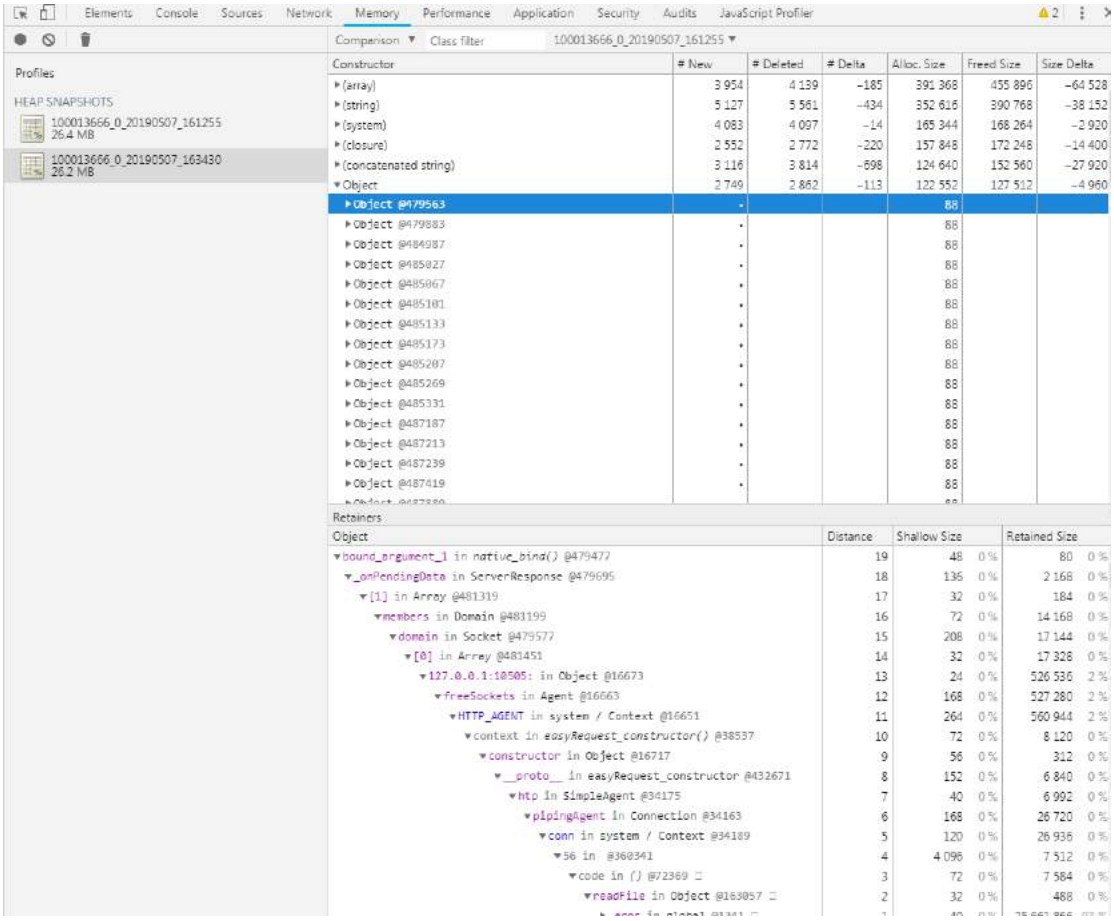
遇到这种情况，基本猜测是发生了 Memory-Leak（内存泄漏）。我们需要分析 heapdump 来定位具体的问题点。

不建议在应用中定期发送 heapdump 的信息来监控，比较消耗内存。所以我们一般在发布到测试阶段，发现问题之后，采样几个不同时间点 heapdumpsnapshot 进行比对。使用的是开源的 heapdump。

首先将两份 snapshot 文件加载到 chrome 中，查看 statistics，对比这里的内存变化和 Docker 中的内存变化。



如果两者的变化一致，那么就说明内存泄漏的确发生在 Heap 区域，那么就可以进行两份 snapshot 的对比。



如果两者变化不一致，Docker 变化量明显比 Heapdump 的多，那么就说明内存泄漏可能出现非 Heap 区域（堆外内存区域），需要查看一下 snapshot 中 Buffer 的数量是否有变化，是不是 buffer 导致的，或者查看一下 Node.js 的官方的 changelog 是否有提到 memory issue。

三、公共服务

3.1 服务调用

SOA client：SOA 客户端主要负责调用 JAVA/.NET/Node.js 等各技术栈的 SOA 服务。主要服务于数据聚合的场景。

3.2 存储服务

1）Ceph（资源存储客户端），主要存储静态资源，包含 JS/CSS/图片等；
2）Redis（Redis 客户端），为应用提供 Redis 缓存服务；
3）Kafka （消息系统） 消息生产者和消费者；

3.3 缓存服务

缓存中间件 Cache 主要解决以下问题：实现跨进程共享内存和跨进程锁。(参考开源模块 node-shared-cache)

3.4 公共业务

1）监控模块目前携程 Node.js 支持三种场景的日志

场景一：CAT 日志埋点，树状结构展示日志,监控服务运行快慢、监控异常、以及自定义事件监控告警。目前携程 CAT 已开源 CAT

场景二：可通过特定事件、特定时间、特定 tag 值过滤查询日志

场景三：可基于时间序列查看各种性能数据聚合结果，如统计某个中间件使用次数、某请求结果的平均值等。

2）foundation-framework 基础模块

- 为不同环境下所有应用提供统一的获取 AppId、环境等基础配置的 API
- 提供 IPv4、IPv6 的检查和 IPv6 的全地址转换

3) qconfig-client 该中间件支持从携程内部服务配置中心获取不同文件类型的配置，支持配置热更新。

4）携程 Node.js 还提供：获取 mysql 数据库连接信息、ABTest、pm2 跨进程通讯等功能模块。

3.5 DR (Disaster Recovery)

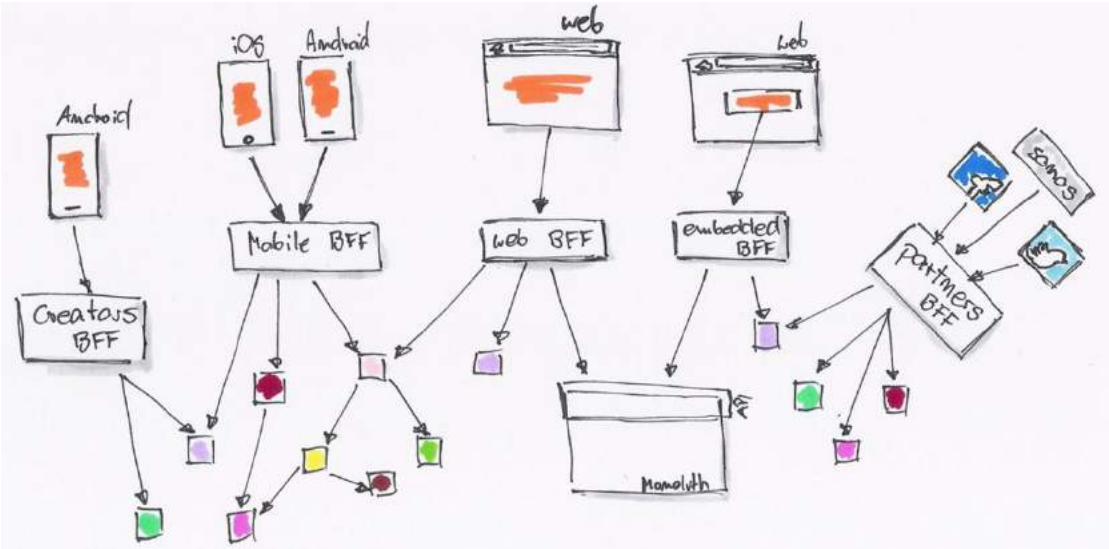
为支持 DR，nodeJS 中间件做以下处理：

1）服务连接失败重试机制。
2）通过 IP 地址访问服务时，需定时重新获取服务 IP 地址。
3）同一服务存在多个 IP 地址时，依次通过不同 IP 地址访问服务，直到成功或全部失败。
4）特定数据缓存。

在某个服务器宕机或某个 IDC 机房毁坏情况下，目前 nodeJS 中间件大部分无需任何操作，可自动恢复；部分中间件需重启应用，以保证应用可用性和数据的实时性。

四、应用场景

4.1 DA（DataAggregation）



Introduction of BFF(Backend for Frontend)
(摘自 https://www.jianshu.com/p/eb1875c62ad3)

DA 这一层主要的功能是能够提升加载速度，降低重复的数据逻辑处理。

在 DA 之前，前端展示一般需要请求多条服务做数据聚合。更复杂的情况是，如果需要适配多个平台(Web/Android/IOS)，那么就需要服务写多个接口，造成重复的开发和维护工作。

DA 主要负责将数据做逻辑处理，包括缓存、展示限制等，为前端提供更轻量级的服务。

本着“服务自治，服务于 UI”的原则，我们用前端工程师更熟悉的 JavaScript 开发是非常合适的选择，可以降低学习和开发成本，带来更大的灵活性和高效性。实践下来，引入了数据聚合层之后，性能提升在 20%左右。

4.2 SSR（Server-SideRendering）

服务端在携程的引入主要考量有几点：

1）SEO 的.NET+V8 的老架构
2）SPA 模式首屏性能问题
3）JS 技术栈陈旧等诸多问题
4）不同平台重复编码，无法实现代码同构

所以设计一套 SSR 的框架来解决历史问题，最终带来了 30%的开发效率提升和 20%的性能优化。

4.3 内部工具

Node.js 技术栈的内部工具，主要在几个方向：

1）构建工具，例如发布平台中的 Node.js 应用的构建工具
2）跨平台的 GUI 的工具，一般基于 electron 框架开发
3）静态资源的发布

五、小结

经过一年多的积累，携程已经上线 500+的应用。这一年多，我们比较关注的方向是中间件建设和应用性能的监控优化，后续将计划实践一些 Node.js 技术栈的框架建设和工程化方向，希望能通过更稳定的基础设施，探索新的应用场景，提升开发效率。

Electron 在 DevTools 中的探索与实践

【作者简介】隋丰蔚，携程无线平台研发部前端工程师，现负责开发者工具 NFES Developer Tools 的设计与研发。

引言

目前，主流的桌面应用开发方法有几种，一是使用纯 Native 技术栈进行开发，比如说 Windows 上使用 C++，Mac 上使用 Objective-C。这种方式能够实现最好的性能，但是开发成本比较高，周期也长，而且需要分别开发 Windows 和 Mac 版本，人员投入比较大。

二是基于 Qt 等 Native 框架进行开发，这种方案可以获得接近 Native 的性能体验，但是学习成本仍然较高，而且界面开发效率不高，没有办法满足快速迭代的需求。

第三种则是以 Electron 为代表的，允许我们使用 web 技术开发桌面应用的框架。这种方案背后是整个 web 技术体系，资源丰富，跨平台性好，开发效率高，甚至于我们可以使用一套代码逻辑同时开发桌面应用和 web 应用，特别适合企业用来开发一些偏业务型的桌面应用。而且，Electron 由 GitHub 维护，社区活跃度高，像我们熟悉的 VS Code，Skype 都是基于 Electron 构建的。

因此，如果不是要开发对性能要求很高的桌面应用，团队中 web 开发人员又相对充足，Electron 是一个比较合适的选择。

本文将介绍 Electron、开发过程中可能会遇到的问题和场景，以及 Electron 在 DevTools 中的实践，希望可以为想要开发 Electron 应用的小伙伴们提供一点参考或者思路。

一、初识 Electron

根据官方文档，基于 Electron，我们可以使用 JavaScript，HTML 和 CSS 构建跨平台的桌面应用。目前 Electron 支持的平台有 Mac, Windows 和 Linux。

1.1 Quick Start

先来看一下如何使用 Electron 快速构建一个桌面应用，目录结构如下图所示：

其中，index.html 就是我们平时开发的 web 页面，负责界面展示。main.js 则是整个 Electron 应用的入口文件，如下：

```
import { app, BrowserWindow } from 'electron';

let mainWindow;

const createWindow = () => {
    mainWindow = new BrowserWindow({
        width: 800,
        height: 600,
        webPreferences: {
            nodeIntegration: true
        }
    });

    mainWindow.loadFile('index.html');
    mainWindow.on('closed', () => {
        mainWindow = null;
    });
}

app.on('ready', createWindow);

app.on('window-all-closed', () => {
    if (process.platform !== 'darwin') {
        app.quit();
    }
});

app.on('quit', () => {
    // dispose services
});
```

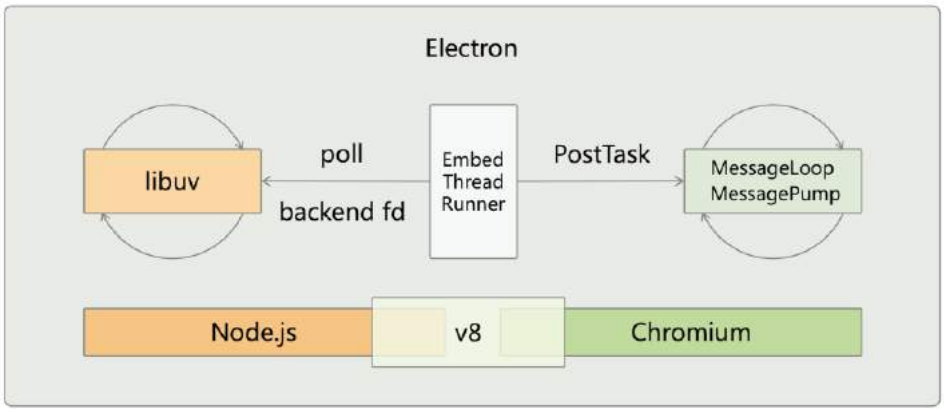
main.js 中首先引入了 app 和 BrowserWindow 模块，app 模块主要负责应用级别的事情，包括应用的生命周期。可以看到，Electron 初始化完成之后，会触发 app ready，这时就可以开始做自己的事情了，比如说在这里我们创建了一个窗口，然后加载了 index.html。如此，一个 Electron 应用就可以运行了，Easy Peasy。

1.2 Electron 工作机制

之所以可以使用 web 技术构建桌面应用，其实是因为 Electron 做了一个整合，它集成了 Chromium 和 Node.js，同时提供了一系列可以操作原生 GUI 的 API。

而能够做这个整合，首先得益于 Chromium 和 Node.js 都是基于 v8 引擎来执行 js 的，所以给了一种可能，他们是可以一起工作的。

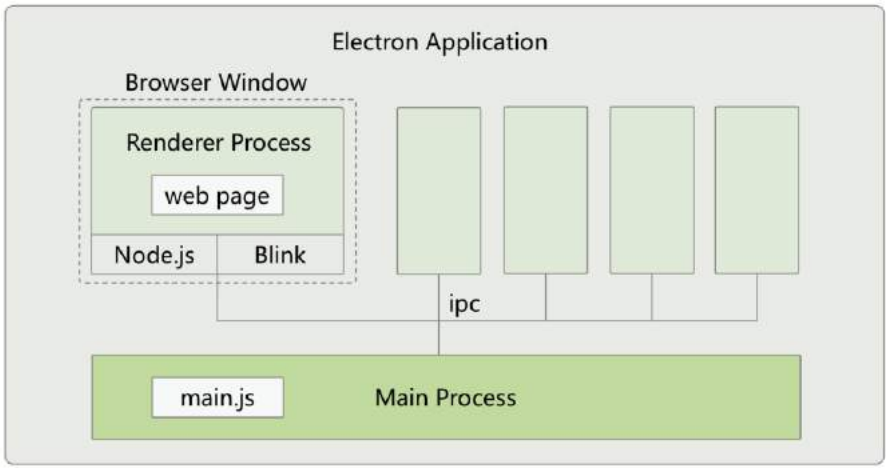
但是有一个问题，Chromium 和 Node.js 的事件循环机制不同。我们知道，Node.js 是基于 libuv 的，Chromium 也有一套自己的事件循环方式，要让他们一起工作，就必须整合这两个事件循环机制。



如上图所示，Electron 采用了这样一种方式，它起了一个新的线程轮询 libuv 中的 backend fd，从而监听 Node.js 中的事件，一旦发现有新的事件发生，就会立即把它 post 到 Chromium 的事件循环中，唤醒主线程处理这个事件。

1.3 Electron 应用架构

Chromium 是多进程模式，每个 Tab 都是一个独立的进程在运行，从而确保了它的稳定性。Electron 延续了多进程的模式，每个窗口对应一个独立的渲染进程，里面运行的就是 web 页面。渲染进程统一由主进程管理，如下图所示。



1.4 通信

在 Electron 中，应用级别的活动以及原生 GUI 模块只能在主进程中运行，渲染进程则主要负责界面展示。这时候就需要解决主进程和渲染进程之间的通信问题。

IPC（Inter-Process Communication）

Electron 提供了两个模块，ipcMain 和 ipcRenderer，它们都是 Node.js EventEmitter 的实例，使用哪个取决于所在的进程。

主进程：

```
import { ipcMain } from 'electron';
ipcMain.on('toMainChannel', (event, arg) => {
    event.sender.send('reply', 'message');
});
win.webContents.send('toRendererChannel', 'message');
```

渲染进程：

```
import { ipcRenderer } from 'electron';
ipcRenderer.on('toRendererChannel', (event, arg) => {});
ipcRenderer.send('toMainChannel', 'message');
```

remote 模块

remote 模块允许在渲染进程中直接调用主进程的模块和方法。从底层实现的角度，remote 其实是对 ipc 做了一层封装，它除了能帮我们避免繁琐的 ipc 消息传递，remote 和 ipc 还有一个本质的区别。

来看一个具体的例子，如下图所示，主进程的 global 上挂了一个 globalData 对象，现在想在渲染进程中获取这个对象中 test 属性的值。

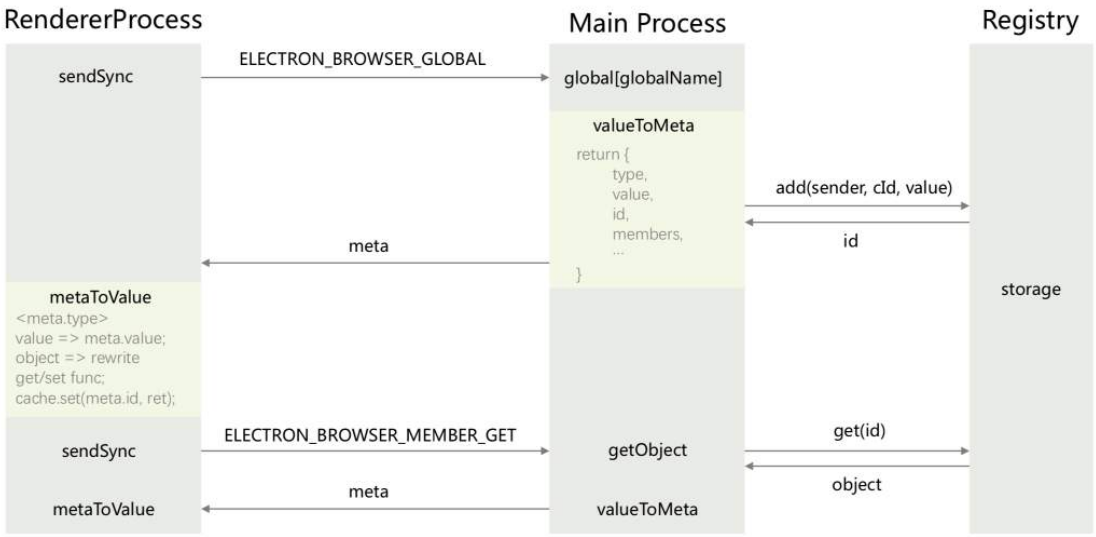
主进程：

```
global.globalData = {
    test: 'testdata'
}
```

渲染进程：

```
import {remote} from 'electron';
let test = remote.getGlobal('globalData').test
```

来看一下 remote 是怎么取值的。如下图所示，首先，渲染进程通过 ipc 给主进程发消息说需要 global 上的 globalData 对象，主进程收到消息后，取到相应的对象，处理之后就把 globalData 相关的信息传回到渲染进程。渲染进程拿到对象之后，直接重写了它的 get 和 set 方法。因此，这时候再获取 test 值的时候，渲染进程会再次发送消息到主进程来获取。



基于这样的机制，可以看出，虽然是在两个进程中，但是完全可以把 remote 取回的对象当作是对主进程中这个对象的引用，因为我们获取到的值总是和主进程中的一致，而使用 ipc 通信，其实是对数据进行了序列化和反序列化，渲染进程拿到的对象和主进程已经没什么关系了。

二、探索 Electron

到这里，大家对 Electron 已经有一个基本的印象了。下面来看 Electron 开发过程中可能会遇到的几个问题和场景。

2.1 启动时间优化

Electron 应用创建窗口之后，由于需要初始化窗口，加载 html，js 以及各种依赖，会出现一个短暂的白屏。除了传统的，比如说延迟 js 加载等 web 性能优化的方法，在 Electron 中还可以使用一种方式，就是在 close 窗口之前缓存 index 页面，下次再打开窗口的时候直接加载缓存好的页面，这样就会提前页面渲染的时间，缩短白屏时间。

但是，优化之后也还是会有白屏出现，对于这段时间可以用一个比较 tricky 的方法，就是让窗口监听 ready-to-show 事件，等到页面完成首次绘制后，再显示窗口。这样，虽然延迟了窗口显示时间，总归不会有白屏出现了。

2.2 CPU 密集型任务处理

对于 cpu 密集型或者 long-running 的 task，我们肯定不希望它们阻塞主进程或者影响渲染进程页面的渲染，这时候就需要在其他进程中执行这些任务。通常有三种方式：

- 使用 child_process 模块，spawn 或者 fork 一个子进程；
- WebWorker；
- Backgroundprocess。在 Electron 应用中，我们可以创建一个隐藏的 Browser Window 作

为 background process，这种方法的优势就在于它本身就是一个渲染进程，所以可以使用 Electron 和 Node.js 提供的所有 api。

2.3 数据持久化存储

为了使应用在 offline 的情况下也可以正常运行，对于桌面应用，我们会将一些数据存储到本地，常见方式有：

- localStorage。对于渲染进程中的数据，可以存到 localStorage 中。需要注意的是主进程是无法获取的。
- 嵌入式数据库。我们也可以直接打包一个嵌入式数据库到应用中，比如说 SQLite，nedb，这种方式比较适合大规模数据的存储以及增删改查。
- 对于简易的配置或者用户数据，可以使用 electron-config 等模块，将数据以 JSON 格式保存到文件中。

2.4 安全性考虑

在 Electron 应用中，web 页面是可以直接调用 Node.js api 的，这样就可以做很多事情，比如说操作文件系统，但同时也会带来安全隐患，建议大家渲染进程中禁用 NodeJS 集成。

如果需要在页面中使用 node 或者 electron 的 api，可以通过提前加载一个 preload.js 作为 bridge，这个 js 会在所有页面 js 运行前被执行。我们可以在里面做很多事情，比如说把需要的 node 方法放到 global 或者 window 中，这样页面中就没办法直接使用 node 模块，但是又可以使用需要的某些功能，如下图所示。

```
let win = new BrowserWindow({
    webPreferences: {
        nodeIntegration: false,
        preload: path.join(__dirname, 'preload.js')
    }
});

// preload.js
const fs = require('fs');
window.nodeUtils = {
    fetchFiles: () => fs.readdirSync(__dirname)
}
```

除此之外，还要注意，使用安全的协议，比如说 https 加载外部资源。在 Electron 应用中，可以通过监听新窗口创建和页面跳转事件，判断是否是安全跳转，加以限制。亦可以通过设置 CSP，对指定 URL 的访问进行约束。

2.5 应用体积优化

对于 Electron 应用打包，首先会使用 webpack 分别对主进程和渲染进程代码进行处理优化，和 web 应用一样。有点区别的地方是配置中主进程的 target 是 electron-main，渲染进程的

target 是 electron-renderer。除此之外，还要对 node 做一些配置，我们是不需要 webpack 来 polyfill 或者 mocknode 的全局变量和模块的，所以设为 false。

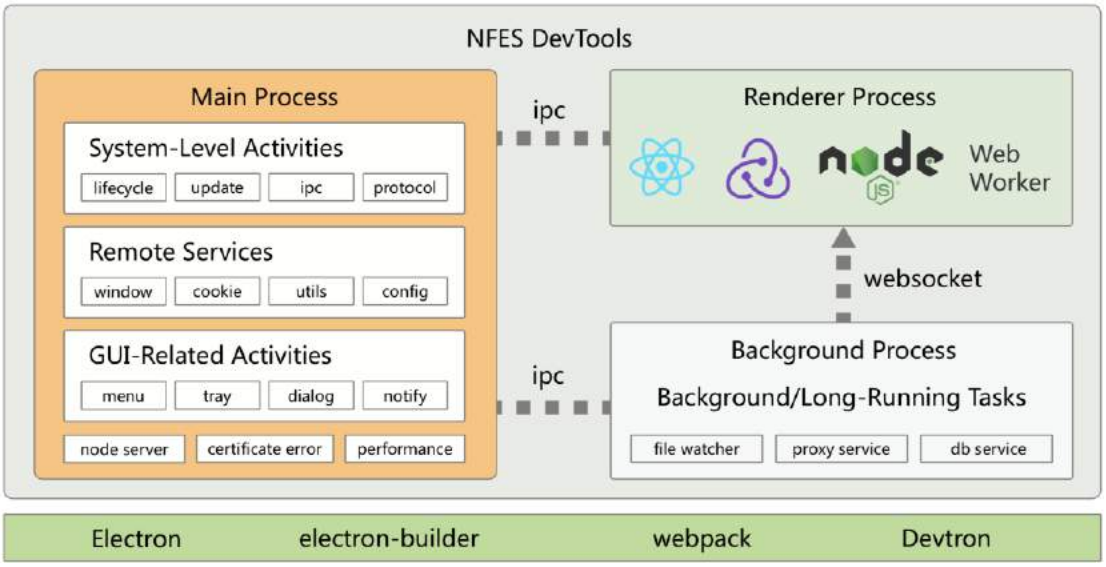
之后，在基于 electron-builder 将应用 build 成不同平台的安装包，需要注意的是，对于 package.json，尽可能地把可以打包到 bundle 的依赖模块，从 dependencies 移到 devDependencies，因为所有 dependencies 中的模块都会被打到安装包中，会严重增大安装包体积。

三、Electron 在 NFES DevTools 中的实践

最后，分享一下 Electron 在 NFES DevTools 中的应用。

NFES 是携程新推出的一整套无线前端解决方案，适用于 pc，h5，hybrid 多个场景，同时支持服务端渲染和单页模式。

NFES DevTools 作为辅助开发平台，能够为开发人员提供稳定的，不受本地 Node.js 版本、全局模块等外部依赖干扰的开发环境，以及 NFES 项目相关的构建，调试，发布，性能监控等功能，从而帮助开发人员更好的开发和维护 NFES 应用。



先从整体应用架构的角度看一下 NFES DevTools。如上图所示，NFES DevTools 是基于 Electron 构建的。

主进程主要做了这几件事情，首先是 app 级别的活动，包括生命周期，自动更新；然后提供了一系列的 remote service，比如说窗口管理，应用级别数据管理，cookie 管理，让渲染进程可以直接调用。其次就是 Native GUI 相关的活动，像创建原生菜单，托盘图标，通知提醒等。最后，在窗口创建之前，我们在主进程中本地起了一个 node server，用来跑 web 应用。

对于渲染进程，主要是基于 React，Redux 写的，Web Worker 用于处理复杂计算，避免阻塞页面渲染。除此之外，我们还启了一个 background 进程，用来执行比如说文件监控这样的

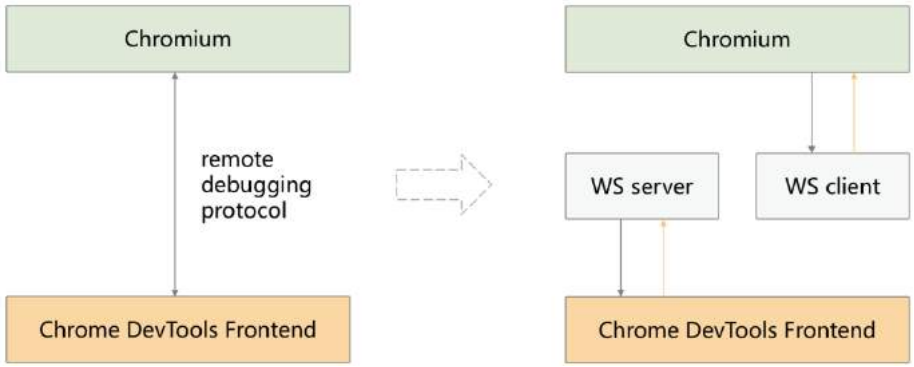
活动。

对于功能模块的实现，主要看下调试功能。对于调试，NFES DevTools 既提供开发态代码的调试，也支持生产态代码的调试。

3.1 开发态调试

开发过程中的调试主要包括以下几点：

- 埋点数据查看
- 性能面板，帮助开发人员在开发时期发现页面可能存在的性能问题。比如说，NFES 支持服务端渲染，所以服务端会传一些数据到客户端，但这样会增大 doc 的体积，影响页面性能。所以，我们会做一个监控，看这些数据是否真的在 render 时被使用了，如果没有我们会提醒开发人员做优化。
- web 和 Node.js 代码调试



对于 web 和 Node.js 代码调试功能的实现，由于 Electron 自身提供的调试 webview 的 api 功能比较弱，不能满足需求，所以我们决定直接使用 Chromium 提供的能力。

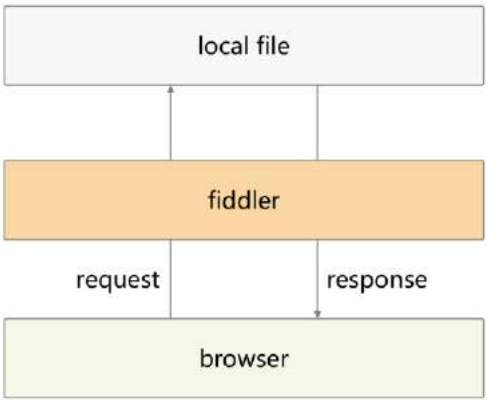
相信大家应该都用过 chrome 开发者工具，其实它本质上就是一个 web app，通过 Chromium 提供的远程调试协议，开发者工具就可以和 chromium 基于 WebSocket 进行通信，如上图左边所示。

调试功能也是基于这个协议实现的，但是如果让调试界面直接和 Chromium 连接会有两个问题，首先是我们没办法完全控制调试过程，不能主动向 Chromium 发送指令；其次是，Chromium 提供的 WebSocket server 只允许一个 client 跟它连接，多个 client 就会出现链接已经被占用的情况。

为了解决这两个问题，我们在调试界面和 Chromium 之间做了一层拦截，如上图右边所示，首先起了一个 WebSocketserver，让它充当 Chromium 跟调试界面连接；然后通过接口获取到真实的 WebSocket 调试地址，起了一个 client 让它跟 Chromium 链接。通过这种方式，就可以拦截掉所有调试界面和 Chromium 之间的 WebSocket 通信，之后做一个转发就可以了。

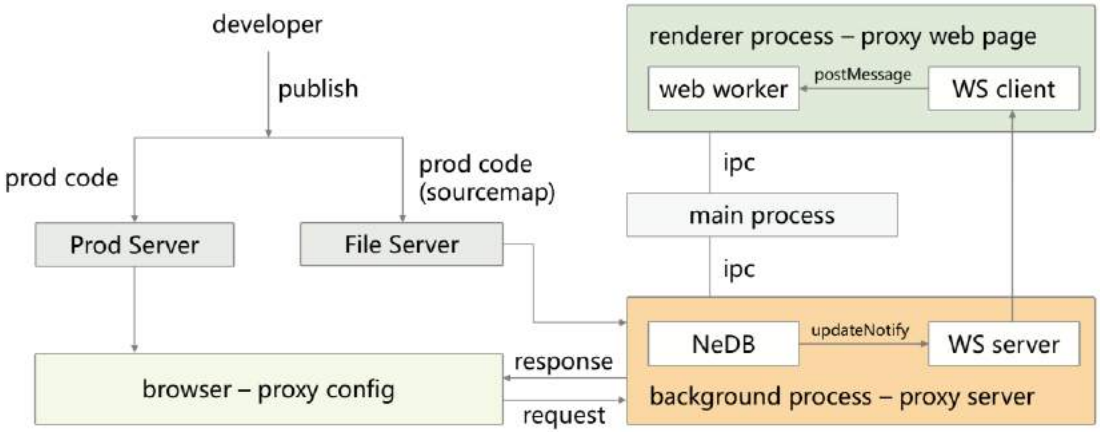
3.2 生产态调试

生产态调试功能是为了帮助开发人员更方便地调试线上代码。通常，调试线上代码会选择使用 fiddler，将线上文件代理到本地来进行调试，如下图所示。NFES DevTools 也支持这种方式，但是这里想跟大家聊一下另一种调试方法。



如下图所示，开发人员在发布 NFES 应用的时候可以选择同步生成一个生产态调试环境，这个调试环境和发布到线上的是一致的，但是多了 sourcemap。

当开发人员需要调试线上的代码的时候，可以开启代理功能，开发人员设置好浏览器代理后，我们会拦截浏览器中的 http/https 请求，把其中与 NFES 应用相关的请求代理到生产态调试环境中，对请求头，响应头，返回值作出相关处理后再返回给客户端，这样开发人员就可以方便的使用 sourcemap 调试线上代码。



代理功能的实现是在 background 进程中，我们基于 Node.js 搭建了代理服务器，并将拦截到的请求数据存储在 nedb 数据库中，因为请求量可能比较大，并且需要根据请求状态的变化对数据进行更新。

数据库插入或者更新数据之后会通知 WebSocket 服务器，实时发送数据到渲染进程，然后在 Web Worker 中计算好需要展示/更新的节点信息之后，重新渲染 http 请求列表。

四、结语

本文简单介绍了 Electron，由于它整合了 Chromium 和 Node.js，所以基于 Electron，可以使用 web 技术开发跨平台的桌面应用。我们也了解了 Electron 的工作机制，以及在开发过程中可能会遇到的白屏，多进程，数据持久化，安全性等问题/场景。

另外也分享了 Electron 在 NFES DevTools 中的实践，包括对 Electron，Chromium，Node 能力的应用，希望可以为想要开发 Electron 应用的小伙伴们有一点启发。

携程酒店 iOS 动态 View 的探索

【作者简介】姜睿东，2009 年加入携程，从事无线研发，现在大住宿事业群负责酒店无线研发工作。

一直以来，Native App 因为审核的原因，新版本不能很及时地上线。尤其是 iOS，碰到点审核问题，有时候一连几天都不能上架，严重影响业务和产品的体验。

大家一直都在寻求能够动态更新业务的方法，关于这方面的框架也是层出不穷。自从 Facebook 推出 React Native 以后，便以其良好的兼容性和性能优势占据了这方面的领先地位，携程也在此基础上开源了 CRN 框架。

如果是新业务，用 CRN 开发是非常合适的，开发效率高，双平台兼容性好。但如果要把已有的 Native 页面转 CRN，复杂的核心页面成本会有点高。在不增加人手的情况下，要想同时进行业务的迭代和 CRN 的转换，会有点力不从心。

如果硬转，周期会很长。以携程酒店主流程页面之一的订单详情页为例，在没有额外增加人手的情况下，前后花了几个月时间，才陆陆续续完成了 90%的功能转 CRN，过程尤为艰辛。订单详情页是主流程页面中相对简单的，如果要转酒店详情页，光是几百行的 ViewModel 就已经让人望而却步了。

对此，我们考虑能不能采用一种让 Native 和 CRN 共存的方式，这样既可以保留 Native 的业务逻辑，又可以在 UI 层面做到灵活应变。最关键的是，可以分模块的开发，而不用像转 CRN 那样必须整个页面一起上。

当然，Native 和 CRN 混合的解决方案早就有了，但是当 CRN 作为一个子 View 出现在 Native 页面里的时候，由于 CRN 的框架比较重量级，在性能上并不是特别理想，而且和 Native 的交互也不是特别方便，所以我们开始考虑有没有更为轻便的解决方案。

在比较了多种跨平台方案之后，首先排除了类似 Lua 这种需要依赖第三方库，且语法非主流的方案，最终决定采用原生系统就自带支持的，且语法有着广泛群众基础的 JavaScript。

在 iOS7 之前，要在 Native 环境中和 JavaScript 交互是非常简单且功能有限的，基本上只有依靠 Webview 的 EvaluateJavaScript 来注入执行一段 JS 脚本。从 iOS7 开始，苹果引入了 JavaScriptCore 这个库，顿时给 iOS 的开发带来了翻天覆地的变化。

为什么会这么说呢，首先来看一下 JavaScriptCore 中所包含的两个关键类，JSContext 和 JSValue：

JSContext

JSContext 提供了一个在 APP 中执行 JavaScript 代码的环境，使得我们可以直接在 Objective-C 或 Swift 代码中直接调用 JavaScript 代码，并得到返回结果，反过来也可以暴露方法和类

供 JavaScript 调用。

JSValue

JSValue 则是一个 JavaScript 数据类型在 Objective-C 或 Swift 中的包装对象，借助于这个对象我们可以在 Native 代码和 JavaScript 代码之间互相传值，这两者之间的对应关系如下图所示：

Objective-C (and Swift) Types	JavaScript Types
nil	undefined
NSNull	null
NSString (Swift String)	String
NSNumber and primitive numeric types	Number, Boolean
NSDictionary (Swift Dictionary)	Object
NSArray (Swift Array)	Array
NSDate	Date
Objective-C or Swift object (id or AnyObject) Objective-C or Swift class (Class or AnyClass)	Object
Structure types: NSRange, CGRect, CGPoint, CGSize	Object
Objective-C block (Swift closure)	Function

简单总结一下，JSContext 提供 JavaScript 和 Native 互相调用的接口，JSValue 提供互相调用之间的数据类型转换，这样的调用方法比之前的 Webview 要强大灵活许多，想象空间也大了很多。所以我们接下去就准备在这基础之上做点文章。

第一步，先创建一个 JavaScript 对象，用来描述对应 iOS 中的 UIView，代码用 ES6 如下：

```
Class View {
    constructor() {
        this.x = 0;
        this.y = 0;
        this.width = 0;
        this.height = 0;
        this.borderWidth = 0;
        this.borderColor = '';
        this.cornerRadius = 0;
        this.masksToBounds = false;
        this.subviews = [];
    }
    initWithFrame(x, y, width, height) {
```

```
        ……
    }
    addSubview(v) {
        ……
    }
    setOnclick(click) {
        ……
    }
    ……
}
```

这些属性和方法都是 iOS 中 UIView 比较常用的，如同在 iOS 中 UILabel 是继承自 UIView 一样，我们继续创建一个 JavaScript 的 Label 对象，并继承自刚才在上面创建的 View 对象。

```
Class Label extends View {
    constructor() {
        super();
        this.text = '';
        this.textColor = '';
        this.textSize = 14;
        this.fontStyle = 0;
        this.textAlignment = 0;
        this.lineBreakMode = 4;
        this.numberOfLines = 1;
    }
}
```

以此类推，我们继续创建诸如 Imageview,Button,ScrollView 等 iOS 中常用的组件，只要愿意，所有的组件都可以用这种方式来描绘。

有了这些基础的 JavaScript 组件，接下去就可以如同在 iOS 中布局一样，开始用这些组件进行布局，如下代码片段示例了如何对一张图片进行布局。

```
createImage() {
        var container = View.initWithFrame(0, 0, 50, 50);
        container.backgroundColor = "#FFFFFF";
        var image = Image.initWithFrame(0, 0, 50, 50);
        image. imageUrl = 'http://m.ctrip.com/xxxxx.png';
        container.addSubView(image);
        return container;
 }
```

对于熟悉 iOS 开发的同学来说，会觉得这段代码非常眼熟。没错，这就是一段用 JavaScript 来写的 iOS 代码，依此类推，稍微复杂一点的布局也可以用这种方式完成。

最后来看一下布局完成以后的返回值，暂时还是先以上面的 Image 控件来做示例：

```
render() {
            varcontainer = View.initWithFrame(0, 0, 50, 50);
          container.backgroundColor = "#FFFFFF";
          var image = Image.initWithFrame(0, 0, 50, 50);
          image. imageUrl = 'http://m.ctrip.com/xxxxx.png';
          container.addSubView(image);

          var demoView = View.initWithFrame(0,0,180,180);
          demoView.addSubView(container)
          return demoView;
}
```

如果在浏览器或者 JavaScript 环境中运行上述代码，会得到一个自定义的递归对象，根对象会包含一个 Subview 数组，数组中的每个元素都有可能是另外一组 UI 对象，当然实际操作中并不建议层次太多，一般 1-2 层。

做到这里，JavaScript 的部分暂告一段落。接下来回到 Native 当中，还记得上文提到的 JSContext 么？这是一个在 Native 当中的 JavaScript 执行环境，我们在 Native 环境中用 JSContext 来执行刚才那个 Demo，就会得到一个对应的 JSValue 值，这个 JSValue 的值用 [JSValuetoObjct]来转换成 Object-C 对象的话，最终就得到了一个字典，NSDictionary。

继续递归地拆解这个字典，拆解到底，每个元素最终都会转成 OC 的 Object，然后根据每个 Object 预先定义好的 Type 类型，实例化成相应的 Native 组件，并且每个组件有一个对应的数据 Model。

还是以上述那个 Label 为例，其对应的 OC Label 代码如下：

```
@implementation Label

- (void)setModel:(HTLDynamicLabelModel *)model{
    self.dynamicViewModel = model;
    self.text = model.text;
    self.textColor = model.textColor;
    self.font = model.font;
    self.lineBreakMode = model.lineBreakMode;
    self.numberOfLines = model.numberOfLines;

     if(model.richText && model.richText.attributedString) {
         self.attributedText = model.richText.attributedString;
     }
}
```

@end

到此为止，就完成了所有之前在 JavaScript 中描绘的控件在 Native 里的转换，剩下的事情就是对这些 Native 组件进行渲染了，具体就不在这里描述了。

总体来说，这个思路在原理上跟 RN 或者 CRN 是一样的，但更为轻量一点，几乎 0 配置就能使用。通过配置增量更新，从服务端下载最新的 JS 文件，可以做到类似 CRN 在线更新的效果。

从性能上来看，因为不需要额外加载任何框架代码，JS 执行的消耗几乎可以忽略，所以和 Native 混合在一起的时候，几乎看不出有任何延迟。

这个方案非常适合做一些轻量级的又需要经常不定期更新的 UI，比如节日氛围或者城市包装的 UI。这些 UI 经常会跟随节假日更新，用这个方案可以轻松在线更新 UI，不用通过服务端下发一堆样式来控制，减轻了服务发布的压力和不必要的服务交互。

综上所述，这是我们团队对新事物的一些探讨和研究，并不存在要代替 CRN 或其他框架一说，每个框架都有其适用的场景，没有绝对的优劣之分。

在研究这个解决方案的过程中，我们也认真地深入了解了 JavaScriptCore 的一些机制，原理都是万变不离其宗的，但可以结合不同的场景，进行不同的演变，就看怎么灵活运用了。

所以，与其说本文是在探索 iOS 中动态 View 的解决方案，也不妨说成是对 JSContex 和 JSValue 如何运用的一些探讨，从实际的摸索中来看，灵活运用好 JavaScriptCore，可以有无限多的可能。

携程机票 App Kotlin Multiplatform 初探

【作者简介】陈琦，携程机票研发部无线研发总监，负责携程 App 机票业务的技术研发和管理工作。

从 2017 年 9 月到 2019 年 5 月，经过一年半的努力，携程机票 App 团队完成 90% 从 Native 到 Ctrip React Native (CRN) 的技术栈转型。

2019 年初，我们开始思考下一步规划。

除了继续做深 React Native 技术，更快更稳定的迭代交付机票业务需求，优化用户体验，我们需要从具体业务逻辑实现层次抽离出来，从一个完整的应用程序架构设计和实现的角度，寻找跨平台技术的未来方向。

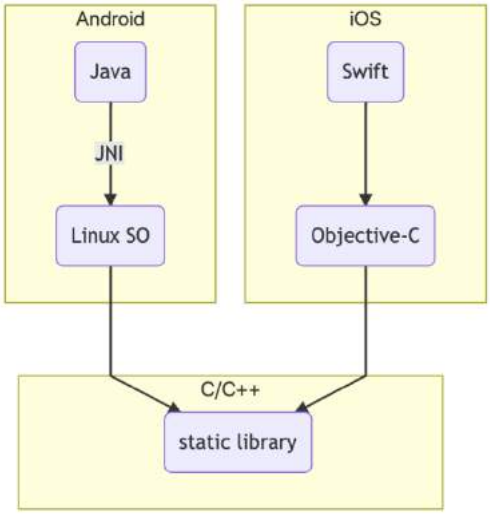
React Native 和 Flutter 这类大前端技术方案已经可以很好的支撑用户界面和组件，业务逻辑需求功能的实现，但是单线程动态脚本语言在以下领域仍显不足。

- 灵活调用强大的平台/厂商 API (AI, AR, mult-core GPU, ...)
- 高性能计算
- 多线程处理
- 后台任务
- 低功耗

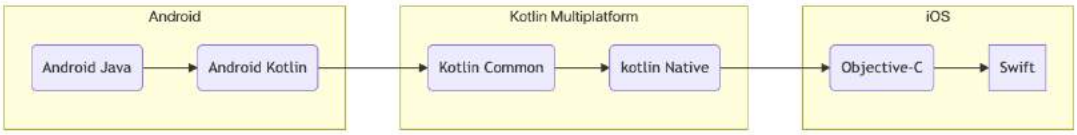
我们希望能够找到一种可靠的跨平台，原生，或能够与原生 API 进行灵活自由双向互操作的技术方案。经过一段时间的针对 Kotlin 及相关开源社区的调研，观察，实践，Kotlin Multiplatform 技术在这方面展现出了良好的发展潜力。

一、Native multiplatform

传统主流的跨平台原生方案是 C/C++，目前依然是最被广泛使用的。 React Native 和 Flutter 的底层实现也是如此。



Kotlin Multiplatform 的跨平台迁移如下图。



二、Kotlin Native

了解 Kotlin Multiplatform 需要先从 Kotlin Native 入手。相比 Kotlin/JVM，Kotlin Native 使用 Kotlin 语言编译器，配合 LLVM backend，将 Kotlin 代码编译为平台原生二进制文件，不依赖虚拟机或运行时环境。当前 LLVM 版本 6.0.1 。官方正在将编译方案从 LLVM 的 backend 转移到 frontend (clang) 。

目前已支持的平台：

- iOS 9.0+ (arm32, arm64, x86_64 模拟器)
- macOS (x86_64)
- Android (arm32, arm64) ， 编译生成 Linux SO 文件
- Windows (mingw x86_64, x86)
- Linux (x86_64, arm32, MIPS, MIPS little endian, Raspberry Pi)
- WebAssembly (wasm32)

三、Kotlin Native 与 C 双向互操作

3.1 cinterop

Kotlin Native 官方附带工具，用于快速生成 Kotlin 与平台 C 库互相调用操作所需的内容。

首先创建一个.def 文件，描述需要包含在语言绑定的内容。然后使用 cinterop 分析 C 头文件，映射生成 Kotlin 语言的类型，函数和常量，完成 Kotlin 绑定。最后通过 LLVM 编译器链接生成最终的可执行文件 *.kexe 或库文件 *.klib。

kexe 是平台相关的可执行程序文件格式。

klib 是平台相关的库文件格式，类似 JAR 的 ZIP 格式，细节详见官网文档：https://kotlinlang.org/docs/reference/native/libraries.html#the-library-format

解压后的文件夹结构如下：

- foo/
- targets/
    - $platform/
    - kotlin/
        - Kotlin compiled to LLVM bitcode.
    - native/
        - Bitcode files of additional native objects.
    - $another_platform/
    - There can be several platform specific kotlin and native pairs.
- linkdata/
    - A set of ProtoBuf files with serialized linkage metadata.
- resources/
    - General resources such as images. (Not used yet).
- manifest - A file in *java property* format describing the library.

3.2 平台库

大多数情况下，我们并不需要使用 cinterop 手动生成所有所需的 C 库绑定。

Kotlin Native SDK 已经提供了大部分平台的原生库绑定。例如：

- Linux POSIX
- Windows Win32
- macOS/iOS Apple Framework, POSIX
- 以及各平台的常用热门库，OpenGL， zlib 等

Kotlin Native 在本机开发时默认下载到 ~/.konan/ 文件夹，例如 ~/.konan/kotlin-native-macos-1.2.1/，平台库文件位于~/.konan/kotlin-native-macos-1.2.1/klib/platform/，已包含以下内容，可见大部分平台 SDK 都已预处理完成。

Android Native Arm32

├── android

```
├── android_arm32.tree.txt
├── builtin
├── egl
├── gles
├── gles2
├── gles3
├── glesCommon
├── linux
├── media
├── omxal
├── posix
├── sles
└── zlib


13 directories, 1 file

iOS Arm64

├── ARKit
├── AVFoundation
├── AVKit
├── Accelerate
├── Accounts
├── AdSupport
├── AddressBook
├── AddressBookUI
├── AssetsLibrary
├── AudioToolbox
├── AuthenticationServices
├── BusinessChat
├── CFNetwork
├── CallKit
├── CarPlay
├── ClassKit
├── CloudKit
├── CommonCrypto
├── Contacts
├── ContactsUI
├── CoreAudio
├── CoreAudioKit
├── CoreBluetooth
├── CoreData
├── CoreFoundation
```

├── CoreGraphics
├── CoreImage
├── CoreLocation
├── CoreMIDI
├── CoreML
├── CoreMedia
├── CoreMotion
├── CoreNFC
├── CoreServices
├── CoreSpotlight
├── CoreTelephony
├── CoreText
├── CoreVideo
├── DeviceCheck
├── EAGL
├── EventKit
├── EventKitUI
├── ExternalAccessory
├── FileProvider
├── FileProviderUI
├── Foundation
├── GLKit
├── GSS
├── GameController
├── GameKit
├── GameplayKit
├── HealthKit
├── HealthKitUI
├── HomeKit
├── IOSurface
├── IdentityLookup
├── IdentityLookupUI
├── ImageIO
├── Intents
├── IntentsUI
├── LocalAuthentication
├── MapKit
├── MediaAccessibility
├── MediaPlayer
├── MediaToolbox
├── MessageUI
├── Messages
├── Metal
├── MetalKit

```
├── MetalPerformanceShaders
├── MobileCoreServices
├── ModelIO
├── MultipeerConnectivity
├── NaturalLanguage
├── Network
├── NetworkExtension
├── NewsstandKit
├── NotificationCenter
├── OpenAL
├── OpenGLES
├── OpenGLES2
├── OpenGLES3
├── OpenGLESCommon
├── PDFKit
├── PassKit
├── Photos
├── PhotosUI
├── PushKit
├── QuartzCore
├── QuickLook
├── ReplayKit
├── SafariServices
├── SceneKit
├── Security
├── Social
├── Speech
├── SpriteKit
├── StoreKit
├── SystemConfiguration
├── Twitter
├── UIKit
├── UserNotifications
├── UserNotificationsUI
├── VideoSubscriberAccount
├── VideoToolbox
├── Vision
├── WatchConnectivity
├── WatchKit
├── WebKit
├── builtin
├── darwin
├── iAd
├── iconv
```

```
├── ios_arm64.tree.txt
├── objc
├── posix
└── zlib


116 directories, 1 file
```

四、Kotlin Native 与 Swift/Objective-C 双向互操作

基于 cinteroop，增加了面向对象的映射。细节详见官网文档：https://kotlinlang.org/docs/reference/native/objc_interop.html#mappings

Kotlin	Swift	Objective-C
class	class	@interface
interface	protocol	@protocol
constructor/create	Initializer	Initializer
Property	Property	Property
Method	Method	Method
@Throws	throws	error:(NSError**)error
Extension	Extension	Category member
companion member <-	Class method or property	Class method or property
null	nil	nil
Singleton	Singleton()	[Singleton singleton]
Primitive type	Primitive type / NSNumber	
Unit return type	Void	void
String	String	NSString
String	NSMutableString	NSMutableString
List	Array	NSArray
MutableList	NSMutableArray	NSMutableArray
Set	Set	NSSet
MutableSet	NSMutableSet	NSMutableSet
Map	Dictionary	NSDictionary
MutableMap	NSMutableDictionary	NSMutableDictionary
Function type	Function type	Block pointer type

五、Kotlin Multiplatform

Kotlin 1.3 重新设计了多平台工程项目架构，以提高工程结构的灵活性和扩展性，更容易共享复用 Kotlin 代码。

Kotlin Native 变成 Kotlin Multiplatfrom 的目标平台之一， 相关库和插件转为内部实现。例如对于应用程序开发人员使用的 Gradle 插件，从 org.jetbrains.kotlin.konan 变更为

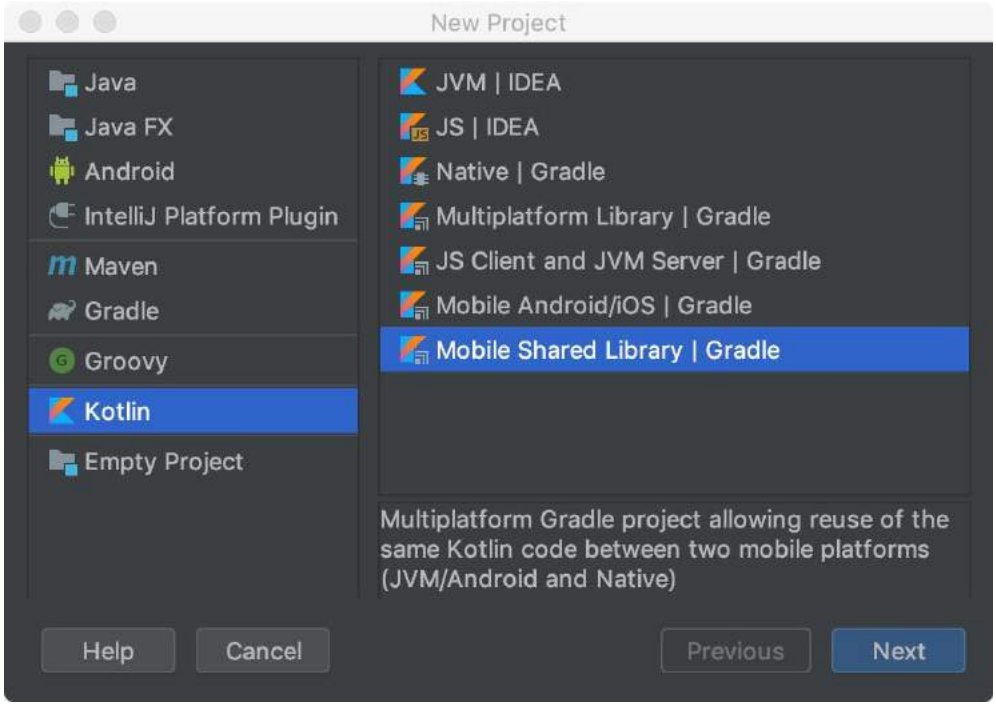
org.jetbrains.kotlin.multiplatform。

konan 变成 multiplatform 内部引用的依赖库，创建 Kotlin Gradle 工程时后台自动下载并保持在本机 ~/.konan/ 文件夹。

为了减少开发人员的误解，对外提供的 SDK 版本号都统一跟随 Kotlin 语言版本号。例如，Kotlin 语言最新版本是 1.3.31，各平台库 SDK 版本号也一样，而 kotlin native macos 1.2.1 仅用于 Kotlin 内部开发人员的版本 Tag，对使用者透明，我们无需关心。

因此网上搜索得到的大部分基于 konan 的文章教程和 GitHub 源码均已过时，需留意 gradle 配置中是否基于 multiplatform plugin。包括官网部分文档。

IntelliJ IDEA 提供了 Kotlin Mulitplatform 的工程模版。实际上 IDEA + Android SDK 可以替代 Android Studio 99%的开发工作。针对 Native 平台有 4 种模版，大同小异，区别仅是 Gradle Modue 结构略有不同。



1）Kotlin/Native 模版是针对单一平台的最小化工程模版。

2）Kotlin (Mobile Android/iOS) 模版沿用 Android 工程的默认结构，将 Android 主工程，Kotlin Common 代码集放在 root/app/src，根目录额外增加了一个 iOS 主工程文件夹。

3）Kotlin (Multiplatform Library) 和 Kotlin (Mobile Shared Library) 非常相似，可以简单的认为后者是前者的子集。前者包含 Kotlin/JS 和 3 个 Native (macOS, Windows, Linux) 平台。后者仅包含 Android，iOS 2 个平台。其中仅有一处细微差异。前者 jvmMain 模块依赖 stdlib-jdk8，后者 jvmMain 模块依赖 stdlib。即前者 JVM 运行环境是 Java 服务端，后者 JVM 运行环境是 Android 设备。

4）Kotlin(Mobile Shared Library) 是最简结构，文件夹如下。

```
├── build.gradle
├── gradle
│   └── wrapper
│       └── gradle-wrapper.properties
├── gradle.properties
├── settings.gradle
└── src
    ├── commonMain
    │   ├── kotlin
    │   │   └── sample
    │   │       └── Sample.kt
    │   └── resources
    ├── commonTest
    │   ├── kotlin
    │   │   └── sample
    │   │       └── SampleTests.kt
    │   └── resources
    ├── iosMain
    │   ├── kotlin
    │   │   └── sample
    │   │       └── SampleIos.kt
    │   └── resources
    ├── iosTest
    │   ├── kotlin
    │   │   └── sample
    │   │       └── SampleTestsNative.kt
    │   └── resources
    ├── jvmMain
    │   ├── kotlin
    │   │   └── sample
    │   │       └── SampleJvm.kt
    │   └── resources
    └── jvmTest
        ├── kotlin
        │   └── sample
        │       └── SampleTestsJVM.kt
        └── resources

27 directories, 10 files
```

接下来我们需要首先解决一些平台相关的上手问题，结合一些简单且实际存在的小场景，探索 Kotlin Multiplatform 的实践现状。

六、场景 1：Logger

打印输出日志是任何平台和技术栈的开发人员每天面对的简单且必需的功能。我们首先看看各平台的基础方案。大致都是定义好日志级别，提供出全局单例或静态方法 API。

6.1 Java Logger

Since Java 1.4

```
import java.util.logging.Logger


private val logger = Logger.getLogger("Module")


fun javaLog() {
    logger.fine("Fine Msg")
    logger.config("Config Msg")
    logger.info("Info Msg")
    logger.warning("Warning Msg")
    logger.severe("Error Msg")
}
```

6.2 Android Log

Since API Level 1

```
import android.util.Log


private val tag = "Module"


fun androidLog() {
    Log.v(tag, "Verbose Msg")
    Log.d(tag, "Debug Msg")
    Log.i(tag, "Info Msg")
    Log.w(tag, "Warning Msg")
    Log.e(tag, "Error Msg")
}
```

实际在 Android 日常开发中，我个人更倾向于使用 Java Logger，相比 Android.util.Log，Java Logger 可以通过注册 ConsoleHandler，FileHandler，SockeHandler，StreamHandler，配合 SimpleFormatter，XMLFormatter，将日志转存到手机本地文件或后端服务器，供后续分析，以及每一行日志代码可以少写一个 Tag 参数。

6.3 iOS os_log

忽略 C print 和 Objective-C NSLog，仅看 iOS 10 提供的 unified logging system。

```
void iOSLog()
{
    os_log(OS_LOG_DEFAULT, "Msg");
    os_log_fault(OS_LOG_DEFAULT, "Fault Msg");
    os_log_error(OS_LOG_DEFAULT, "Error Msg");
    os_log_info(OS_LOG_DEFAULT, "Info Msg");
    os_log_debug(OS_LOG_DEFAULT, "Debug Msg");
    os_log_with_type(OS_LOG_DEFAULT, OS_LOG_TYPE_DEFAULT, "Msg with type");
}
```

6.4 Kotlin Log - Common Expect

参考 android.util.Log，复制一份 Kotlin Common 模块的声明。

跨平台公共模块的实现，除了使用常规的 Interface/Implementation 方案，Kotlin 提供了 expect/actual 声明语法。这里使用 expect 关键字声明一个单例 Log 对象的预期。其类成员方法等同于 Java 的纯虚方法。

```
/**
 * Keep consistent with android.utl.log constant level.
 * */
enum class Level(val value: Int) {
    VERBOSE(2),
    DEBUG(3),
    INFO(4),
    WARN(5),
    ERROR(6),
    ASSERT(7)
}


expect object Log {
    fun isLoggable(tag: String, level: Level): Boolean
    fun v(tag: String, msg: String)
    fun d(tag: String, msg: String)
```

```
    fun i(tag: String, msg: String)
    fun w(tag: String, msg: String)
    fun e(tag: String, msg: String)
    fun wtf(tag: String, msg: String)
}
```

6.5 Kotlin Log - Android Actual

Android 平台的实现直接绑定 android.util.Log。

```
// Kotlin Android Implementation
actual object Log {
    actual fun isLoggable(tag: String, level: Level): Boolean = android.util.Log.isLoggable(tag,
level.value)
    actual fun v(tag: String, msg: String) { android.util.Log.v(tag, msg) }
    actual fun d(tag: String, msg: String) { android.util.Log.d(tag, msg) }
    actual fun i(tag: String, msg: String) { android.util.Log.i(tag, msg) }
    actual fun w(tag: String, msg: String) { android.util.Log.w(tag, msg) }
    actual fun e(tag: String, msg: String) { android.util.Log.e(tag, msg) }
    actual fun wtf(tag: String, msg: String) { android.util.Log.wtf(tag, msg) }
}
```

6.6 Kotlin Log - iOS Actual

由于 cinterop 工具仅处理 C 库的实例和函数的绑定，不能实现 macro 宏定义的绑定。而 os_log()提供的常用 API 实际是 macro 宏定义，所以我们需要找到其内部实际调用的函数 _os_log_internal()。这对新手可能是一个小坑，在未详细了解平台 API 的情况下，在 Kotlin 平台库中费时费力查找绑定方法无果。

```
// <os/log.h> declaration
#define os_log(log, format, ...) \
        os_log_with_type(log, OS_LOG_TYPE_DEFAULT, format, ##__VA_ARGS__)


#define os_log_with_type(log, type, format, ...) __extension__({ \
    _Pragma("clang diagnostic push") \
    _Pragma("clang diagnostic error \"-Wformat\"") \
    _Static_assert(__builtin_constant_p(format), "format argument must be a string constant");
\
    _os_log_internal(&__dso_handle, log, type, format, ##__VA_ARGS__); \
    _Pragma("clang diagnostic pop") \
})
```

_os_log_internal() 的第一个参数 __dso_handle 定义如下。这里有趣了，我们需要调用它的

内存地址指针，kotlinx.cinterop.ptr 就是为此而生。

```
// <os/trace_base.h> declaration
extern struct mach_header __dso_handle;
```

下面这段实现是纯 Kotlin 语言代码，是上文中 Kotlin Common 的 Log 单例对象 expect 声明对应的 actual 实现。调用 Kotlin Native iOS 平台库已提供好的 os_log API 绑定。

```
// Kotlin iOS Implementation
import kotlinx.cinterop.ptr
import platform.darwin.*


actual object Log {
    actual fun isLoggable(tag: String, level: Level): Boolean =
        os_log_type_enabled(OS_LOG_DEFAULT, level.toPlatform())
    actual fun v(tag: String, msg: String) =
        _os_log_internal(__dso_handle.ptr, OS_LOG_DEFAULT, OS_LOG_TYPE_DEFAULT,
"$tag | $msg")
    actual fun d(tag: String, msg: String) =
        _os_log_internal(__dso_handle.ptr, OS_LOG_DEFAULT, OS_LOG_TYPE_DEBUG, "$tag
| $msg")
    actual fun i(tag: String, msg: String) =
        _os_log_internal(__dso_handle.ptr, OS_LOG_DEFAULT, OS_LOG_TYPE_INFO, "$tag |
$msg")
    actual fun w(tag: String, msg: String) =
        _os_log_internal(__dso_handle.ptr, OS_LOG_DEFAULT, OS_LOG_TYPE_INFO, "$tag |
$msg")
    actual fun e(tag: String, msg: String) =
        _os_log_internal(__dso_handle.ptr, OS_LOG_DEFAULT, OS_LOG_TYPE_ERROR, "$tag
| $msg")
    actual fun wtf(tag: String, msg: String) =
        _os_log_internal(__dso_handle.ptr, OS_LOG_DEFAULT, OS_LOG_TYPE_FAULT, "$tag
| $msg")
}
```

6.7 Kotlin Log - AndroidNativeArm Actual

大部分业务场景中，我们使用 Kotlin/JVM 实现 Android 平台的功能，Kotlin Native for AndroidNativeArm32/64 可用但仍不好用。我们简单看看其实现。

与 iOS Native 类似，只需在 build.gradle 文件中添加相应 Target。这里使用了 Gradle Kotlin DSL 新版本。使用 Kotlin 编写 Gradle 配置文件的体验比 Groovy 更佳，IDE 支持语法高亮，自动补全，代码跳转，编译提示等便捷功能。

```
androidNativeArm32() {
        binaries {
            sharedLib()// or staticLib() or executable()
        }
    }
```

Android NDK Log API

```
/**
 * Writes the constant string `text` to the log, with priority `prio` and tag
 * `tag`.
 */
int __android_log_write(int prio, const char* tag, const char* text);


/**
 * Writes a formatted string to the log, with priority `prio` and tag `tag`.
 * The details of formatting are the same as for
 * [printf(3)](http://man7.org/linux/man-pages/man3/printf.3.html).
 */
int __android_log_print(int prio, const char* tag, const char* fmt, ...)
#if defined(__GNUC__)
    __attribute__((__format__(printf, 3, 4)))
#endif
    ;
```

继续纯 Kotlin 语言实现。

```
// Kotlin AndroidNativeArm Implementation
import platform.android.*


@kotlin.ExperimentalUnsignedTypes
actual object Log {
    actual fun isLoggable(tag: String, level: Level): Boolean    = level >= Level.INFO


    actual fun v(tag: String, msg: String) {
        __android_log_write(ANDROID_LOG_VERBOSE.toInt(), tag, msg)
    }
    actual fun d(tag: String, msg: String) {
        __android_log_write(ANDROID_LOG_DEBUG.toInt(), tag,    msg)
    }
```

```
    actual fun i(tag: String, msg: String) {
        __android_log_write(ANDROID_LOG_INFO.toInt(), tag, msg)
    }
    actual fun w(tag: String, msg: String) {
        __android_log_write(ANDROID_LOG_WARN.toInt(), tag, msg)
    }
    actual fun e(tag: String, msg: String) {
        __android_log_write(ANDROID_LOG_ERROR.toInt(), tag, msg)
    }
    actual fun wtf(tag: String, msg: String) {
        __android_log_write(ANDROID_LOG_FATAL.toInt(), tag, msg)
    }
}
```

以上 Demo 源码工程，详见：https://github.com/9468305/log-kotlin

实际生产环境使用，推荐 Jake Wharton's Timber：https://github.com/JakeWharton/timber

七、场景 2：IO File

本地文件读写相比打印输出日志略复杂但也很常用。Android/JVM 和 Java 服务端的 IO File 技术方案一致但场景选型不同。

- Java IO (Blocking IO)
  Default IO Streaming.
- Java NIO (Non-Blocking IO)
  Since 1.4.
- Java NIO2 (Asynchronous I/O, AIO)
  Since 7, Enhancements in 8.

移动端常用 Blocking IO，一方面是因为该方案适合嵌入式平台，另一方面是因为 Android 系统版本对 JDK 高版本的支持更新进展缓慢。长期以来我们需要兼容 JDK 6，最低系统版本升级至 Android 4.4（API Level 19）以上才能够兼容 JDK 7，Android 8.0（API Level 26）才升级至 JDK 8，并且仅支持部分功能 API。

Android NIO 实际广泛应用于网络组件的实现，例如 Google Guava，Square OKHttp。

iOS File 就是 C Posix 使用方式，这里不再赘述。

下面这段代码使用纯 Kotlin 语言调用 iOS 平台 POSIX File API。memScoped{}表示该作用域内的申请的内存空间，当离开作用域后，会被自动释放。这是 Kotlin Native 不依赖 JVM GC 的内存管理方式，即 ARC 自动引用计数。

```
fun sample() {
```

```
    val file = fopen(__filename = "filename", __mode = "r")
    if (file != null) {
        try {
            memScoped { // ARC - Automatic Reference Counting
                val bufferLength = 1024
                val buffer = allocArray<ByteVar>(bufferLength)
                while (true) {
                    val line = fgets(buffer, bufferLength, file)?.toKString()
                    if (line == null || line.isEmpty())
                        break
                    println(line)
                }
            }
        } finally {
            fclose(file)
        }
    }
}
```

那么如何 Kotlin Multiplatform 实现 java.io.file 与 C POSIX File API 的统一？我们看看官方现状。

Package kotlin.io for native 仅有 3 个方法。

```
// Prints the given message to the standard output stream.
fun print()
// Prints the given message and the line separator to the standard output stream.
fun println()
// Reads a line of input from the standard input stream.
fun readLine(): String?
```

Package kotlinx.io 基于 NIO 方案实现，目前仍处于 Experimental 阶段，官方建议配合 kotlinx.coroutines, kotlinx.atomicfu 一起使用，尚未支持 Native 平台。

所以目前我们只能自己实现双平台的统一封装。这部分实现并不难，可参考 OpenJDK 和 AOSP 源码。Java File 底层实现原理也是通过 JNI 调用 C POSIX。Android 源码部分改写了 OpenJDK 的实现。具体细节详见 Android SDK FileInputStream/FileOutputSteam 源码。

另外 Okio 2 正在进行迁移至 Kotlin 和支持多平台，square 团队的最终目标是将 Retrofit 和 OkHttp 运行在多平台。详见：https://github.com/square/okio/issues/370

八、场景 3：SQLite

嵌入式平台主流关系型数据存储方案。

1、Android SQLiteOpenHelper

- Android SDK 默认提供的 SQLite 方案。
- SQLite low-level API
- Raw SQL queries
- 使用比较繁琐

2、Android Jetpack Room

- Jetpack 新组件。
- SQLite 之上的 ORM 抽象层。

3、iOS SQLite library

- 相比 Android，iOS 更接近原始 SQLite C 库。

SQLDelight

https://github.com/square/sqldelight

目前最成熟稳定的 Kotlin 多平台 SQLite 解决方案。作者 Alec Strong, Jake Wharton（又见大神）。不论是 Android Java 开发，还是 Kotlin 多平台开发，我都建议大家了解一下它。

它的思路非常有趣，与 Room 为代表的各种 ORM 方案截然相反。它是从 SQL 查询语句生成代码，而不是从代码生成 SQL 查询。这里不展开介绍，直接放上 Jake Wharton 关于 SQLDelight vs Room 的评论原文。

In my opinion, Room exists at the wrong level of abstraction.

The reason Retrofit and Gson/Moshi/etc. are successful is because there's nothing from which to generate the interfaces and model objects so you write both by hand duplicating an implicit contract. If you switch to protocol buffers, you stop writing models by hand–the tool can generate those. If you switch to gRPC or Swagger, you stop writing Retrofit interfaces by hand–a tool can generate those. When you have an explicit source of schema you no longer need to duplicate that schema by hand in code.

SQL table definitions and queries are a schema. They can define the types and names of both the model objects and the interface through which you interact with queries. Thus, it doesn't really make sense to force the user to duplicate that schema by writing the model objects and interface by hand. You wouldn't do it with protobuf and gRPC. Why are you doing it with your database?

Room is an okay choice. It's far better than all the ORMs people have been using for years.

Alec and I have given talks where the conclusion was that we don't care which you choose, just don't choose an ORM (https://youtu.be/4eUuD7LsqMs).

That being said, it's hard not to see SQLDelight as a step up from Room. It validates more. It has better tooling. It generates Kotlin. And it's multiplatform. Yes, I'm biased, but Alec can tell you how long we spent evaluating what the right level of abstraction is and what the right developer UX is. It's a pleasant experience writing only SQL and having SQLDelight generate the interfaces and model objects that you'd otherwise be forced to write by hand with Room. And when you do that, you also stop (ab)using star selects in your queries and selecting only the minimal amount of columns necessary rather than selecting entire tables so you can reuse entities.

You write the SQL query and you write the method signature. Generating the method signature is a pure function from the SQL query. You have to keep both in sync manually instead of having the method generated automatically based on the SQL bind args and selected columns. SQLDelight generates the interface methods that Room makes you write given the same SQL command.

Similarly, when you write a query that returns results, Room forces you to write a model object which conforms to the selected data. SQLDelight generates the model objects that Room makes you write given the same SQL query.

In summary, you can take all the SQL you're already using with Room, delete all of the interfaces and model objects you had to write manually, and SQLDelight will generate them for you (along with some extra validation that they're correct).

九、The Future

Kotlin 解决方案组件的稳定性和进展，详见：https://kotlinlang.org/docs/reference/evolution/components-stability.html

Kotlin Native 处于 Additions in Incremental Releases (AIR) 阶段。Multiplatform Projects 处于 Moving fast (MF) 阶段。

前不久的 Google IO 2019 大会上，Kotlin 语言在 Android 平台的地位进一步上升。Android Jetpack 系列组件优先支持 Kotlin。Square 的 Okio 2，OkHttp 4.0 正在迁移 Kotlin 并支持多平台。

所以我相信 Kotlin Multiplatform 的未来充满想象力。

十、One more thing

https://gradle.org/kotlin/

Gradle 5.0 已发布 Kotlin DSL v1.0 稳定版，建议尽早迁移 Gradle 工程至 KTS 版本。

前端如何实现业务解耦，携程酒店查询首页的 1.0 到 3.0

【作者简介】何金，携程酒店研发部 Android 资深软件开发工程师，负责酒店代码性能优化、结构改造、疑难问题排查处理，以及 Kotlin 的推广和应用。

酒店查询首页，是用户使用携程 APP 进行酒店预订的第一个页面。它提供了各种类型的酒店筛选入口，让用户进行酒店选择。随着查询首页版本不断迭代，其对应业务，功能和样式经历了由简单到复杂，单一到丰富的过程。

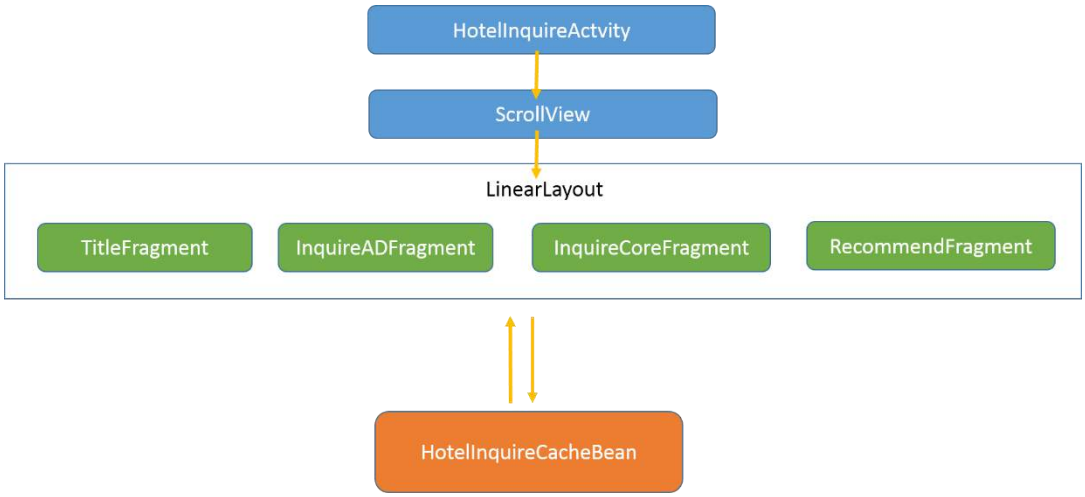
为了更好的适应业务的快速迭代，查询首页的结构也经历了多个版本优化和重构。本文将分享携程酒店是如何根据查询首页自身业务需求特点，进行代码结构优化和重构的。

根据查询首页不同时期业务和代码结构特点，简单的把结构迭代版本划分为三个版本。分别对应简单的 1.0，头疼的 2.0，合适的 3.0。

1.0

1.0 版本是酒店查询首页最早期的一个版本，它所包含的业务可以简单的由上到下做垂直划分，顶部的标题模块，中间提供给用户进行输入的查询模块，以及酒店底部推荐模块。如下图所示：

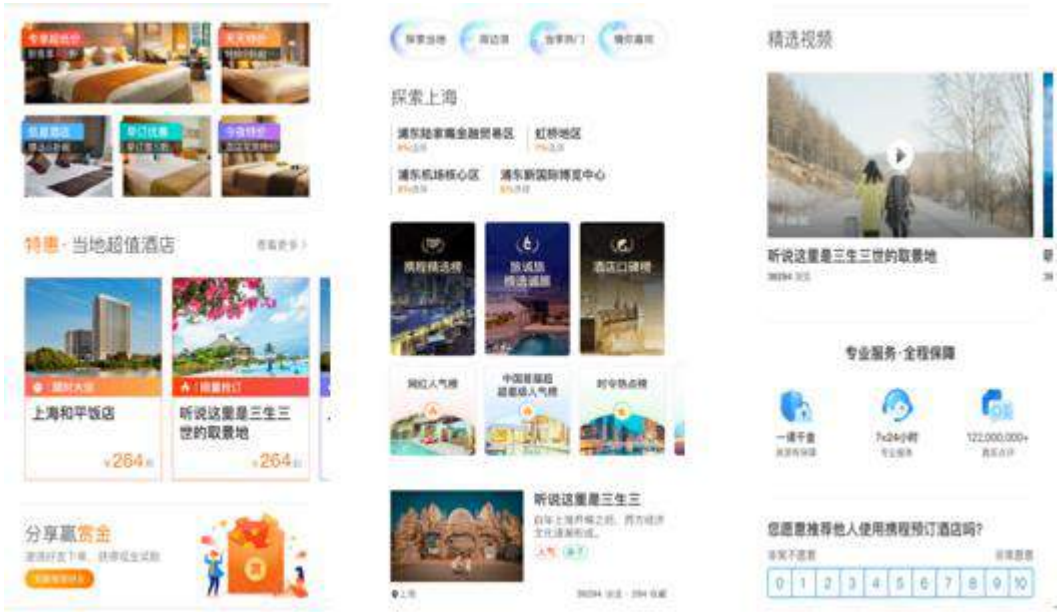
根据这种业务特点，将不同业务模块抽象为不同的 Fragment。将这些 Fragment 统一放到 ScrollView 中进行布局，InuqireCacheBean 用来管理 Fragment 的数据，具体的结构如下图：



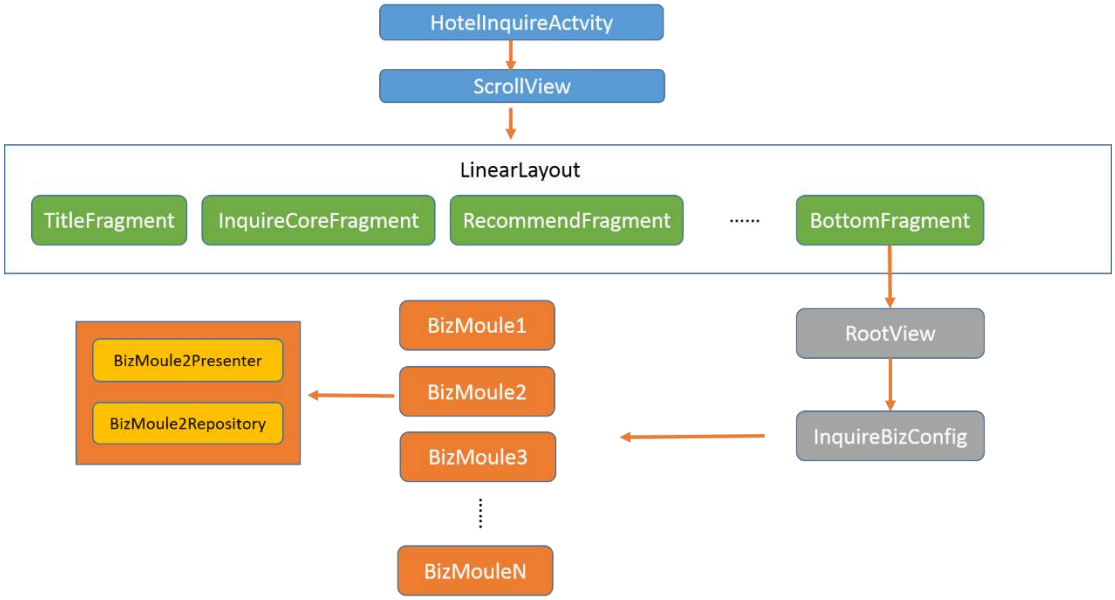
从上图看，1.0 采用的是典型的 MVC 模式。结构清晰明了，但是业务逻辑代码和样式布局全部耦合在相应的 Fragment 里。如果一直维持这个结构，那么随着业务不断迭代和增加，Fragment 里面的代码会越来越臃肿，业务的实现成本和排查问题的难度都会越来越大。

2.0

下面的三张贴图是 2.0 版本时期新增的一些业务所对应的样式。

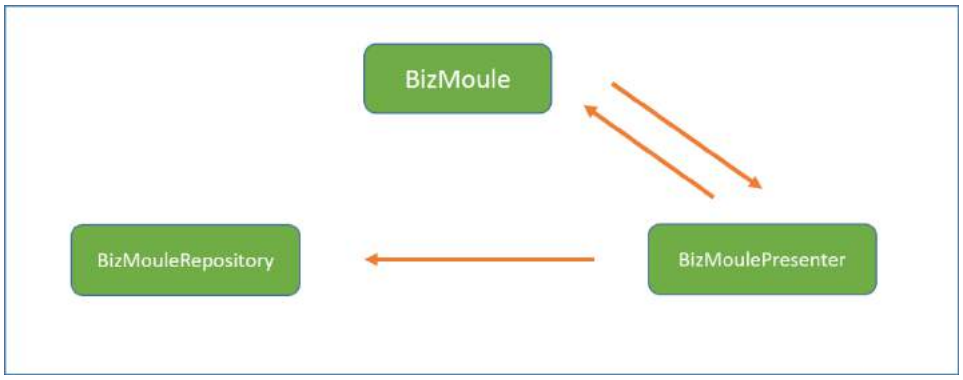


2.0 版本查询页的功能和样式比 1.0 都要丰富和复杂很多。根据这种特点，对查询页进行了模块化拆分，把不同的业务抽象成对应的 Moudle，通过 Module 管理自身业务和 UI 布局。



上图是查询首页 2.0 的架构图，它的容器仍然是 HotelInquireActvity，通过 ScrollView 管理 TitleFragment, InquireCoreFragment，RecommendFragment 和 BottomFragment。

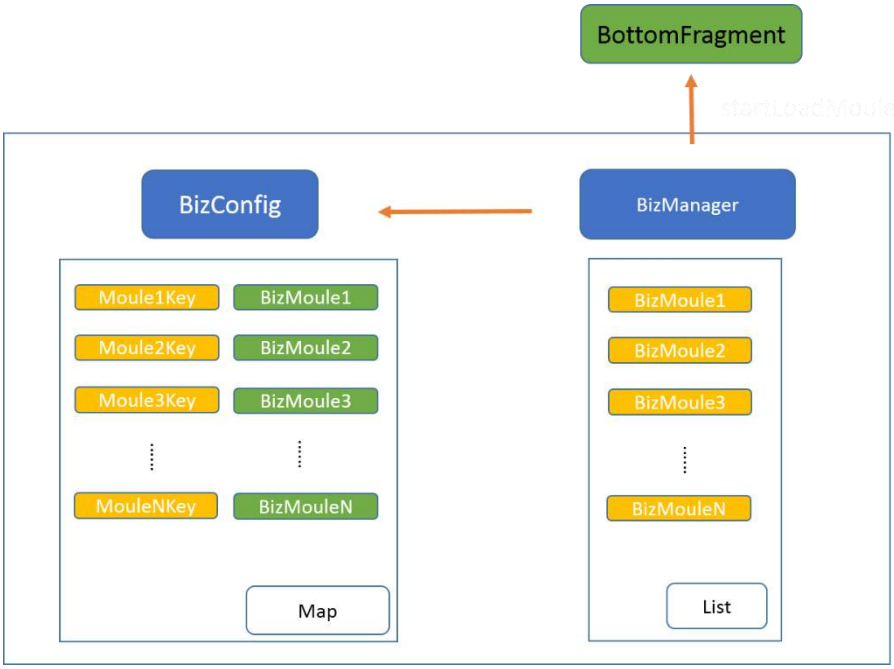
BottomFragment 是 2.0 新增的一个 Fragment，它目的是展示和管理查询首页底部新增的 Moudle。Module 的设计采用 MVP 模式， BizMoudle 代表 V 层，会向外暴露一个 getView 接口，用来展示该业务的样式，BizMoudlePresenter 代表 P 层，用于处理业务逻辑，BizMoudleRepositery 代表 M 层，用于处理数据。Module 的结构如下图:



InquireBizConfig 是各个业务 Module 配置管理器，用于管理加载不同业务对应 Module。InquireBizConfig 由两部分组成，一是 BizConfig，二是 BizManager。

BizConfig 内部核心是一个 Map 数据结构，用于注册查询首页需要用到的所有 Module，BizManager 内部核心是一个 List 数据结构，用于加载和定义各个业务 Moudle 的展示位置顺序。

当 BottomFragment 加载启动的时候，会调用 BizManager 里面的 startLoadMoudles()方法，这个方法会遍历 List,取出相应的 Moudle 加载到 BottomFragment 中，它的结构如下图所示。



从2.0架构图看，对查询首页业务进行了模块化拆分，可以方便团队中不同业务的开发Owner进行同时开发，减少了相互的干扰，提高了业务需求的交付效率，但美中不足的是结构没有做到统一。

有的业务以 Fragment 形式存在，有的业务以 Module 形式存在。该结构还会存在性能黑洞，当 BottomFragment 启动时，会加载所有配置的 Moudle，把所有 Moudle 的 View 加载到布局容器 ScrollView 中，无论这些 Moudle 是否在第一屏展示，影响查询首页的启动性能。

另外由于布局容器采用的 ScrollView，如果业务 Module 里面采用了 ListView 控件，EditText 控件等，那开发必须使用额外的逻辑去处理 ScrollView 和这些控件带来的兼容性问题。布局容器采用的 ScrollView，带来的交互实现成本也很高。

3.0

为了解决 2.0 结构存在的问题，我们又进行了 3.0 版本迭代。

3.0 版本主要围绕 2.0 版本存在的两个问题，一是根布局使用 ScrollView 带来的性能和兼容性问题；二是结构没有统一，业务分别以 Module 和 Fragment 形式存在的问题。

针对 ScrollView 产生的问题，分别选择了三种可替代 ScrollView 的方案。

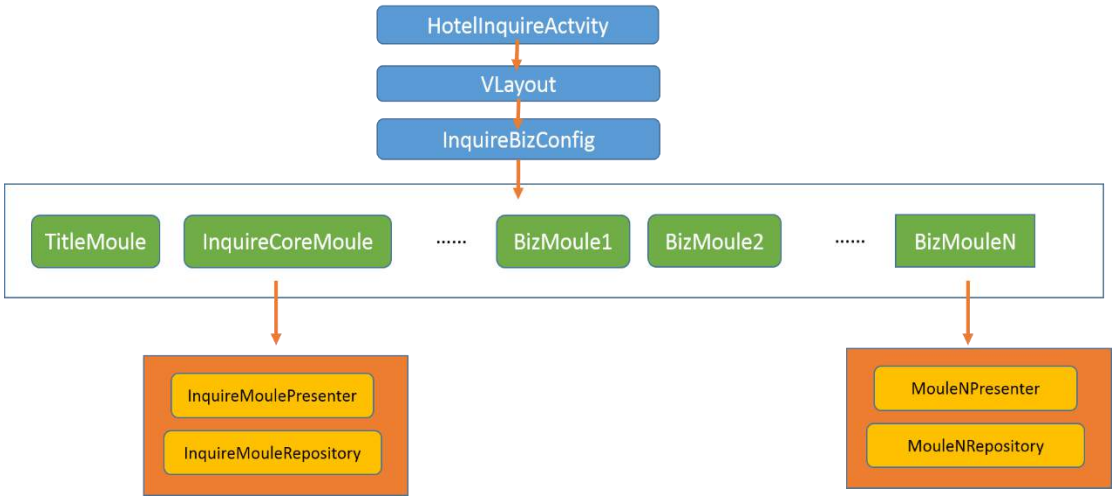
第一种是使用 RecyclerView，通过 Type 来区别各个业务对应的布局，这个方案是理论上行的通的，但是需要改造各个 Moudle 对应的数据结构，需要将不用业务的数据结构进行结构统一，改造成本太高，实现起来难度和风险较高。

第二种采用酒店内部控件 GroupListView 控件，该控件的核心是基于 ListView，它是将不同的 Adapter 融合到一个 Adapter 中。

第三种方案是使用开源组件 Vlayout，该控件的实质是异构的 RecylerView，原理和 GroupListView 相似。

对比第二种和第三种方案，RecyclerView 解决了 ListView 不能局部刷新的问题，另外 RecyclerView 设计上采用四级缓存，在性能上也比 ListView 更优，可为查询首页将来支持流畅性较高的复杂交互做好准备。

基于以上考虑，选择第三种方案替代 ScrollView。对之前 2.0 存在的 TitleFrament，InquireCoreFragment，ReCommendFragment 进行模块化改造，分别改成为 TitleModule，InquireCoreModule，ReCommendModule，移除 2.0 作为 Mudule 容器的 BottomFragment。经过改造后，3.0 的结构图如下：



经过了 3.0 版本的结构迭代和优化，所有的模块都统一抽象为了 Moudle，结构变得更加清晰简单。采用 Vlayout 替代 ScrollView，解决了 2.0 结构版本中潜在的性能黑洞问题，同时 Vlayout 提供了大量的 UI 模板，避免了前端重复造轮子，提高了业务交付效率。

小结

酒店查询首页经过三个版本迭代和优化，结构趋于稳定，为查询首页的未来业务快速交付和生产环境稳定打下了良好基础。

在对酒店查询首页优化和改造时，根据我们团队的特点，采用了渐进式的架构迭代思路，这样既保证了业务需求的交付，也避免了重构带来的风险。

同时也给前端如何实现业务解耦，如何在保证页面性能的前提条件下，承载大量不同的 UI 布局元素提供一种优化借鉴思路。

携程机票 React Native 整洁架构实践

【作者简介】任跃华，携程机票前台软件工程师，从事机票 android、react 和 react native 技术栈相关研发工作。

前言

携程机票前台团队在使用 React Native 实现众多业务的过程中，经历了前期少量探索，中期大量应用，后期架构和性能优化的三个阶段。

在该技术栈积累了一定经验之后，结合不同业务的特点和复杂性，我们重新审视和思考一些前期实践项目的整体优化方向。在 App 国际机票查询列表页的相关业务模块，基于 Clean Architecture 整洁架构之道的思想，进行了一次技术大重构。

一、GUI 架构回顾

GUI 架构模式，一般分为两类：MV* 和 Unidirectional 。

最初的 MVC 将模块划分为展示界面的 View，数据模型 Model 和负责处理二者关系的 Controller 。从 MVC 到 MVP 的过程将 Model 和 View 完全隔离。随着 Databinding 技术的引入，MVP 进化到了 MVVM，使得 View 完全无状态化。

Unidirectional 系列相较于 MV*，则采用了消息队列式的数据流驱动的架构，其中具有代表性的 Redux 采用了统一的状态管理，带来了状态的有序性和可回溯性。

MV* 系列在 iOS、 Android 生态圈中已得到成熟广泛的应用，而在 React 技术栈的 Web 前端领域， Redux 是最主流的数据管理方案。

不同平台选择不同，这其中有框架 API 设计的原因，有编程语言的原因，以及面对的业务逻辑复杂度不同。React Native 是 React 和 Native 的混合体，原有的 Native 框架 API 被映射成 React Component 生命周期，编程语言也发生了变化，不变的是业务场景和逻辑复杂度。

Redux 曾是我们大型 RN 项目的标配，不过实践结果表明， Redux 的一些固有设计并不能很好的应对复杂的应用场景。因此，我们选择了相较于 MV*系列，又对 Presenter/Controller 做了进一步拆分的 Clean Architecture。

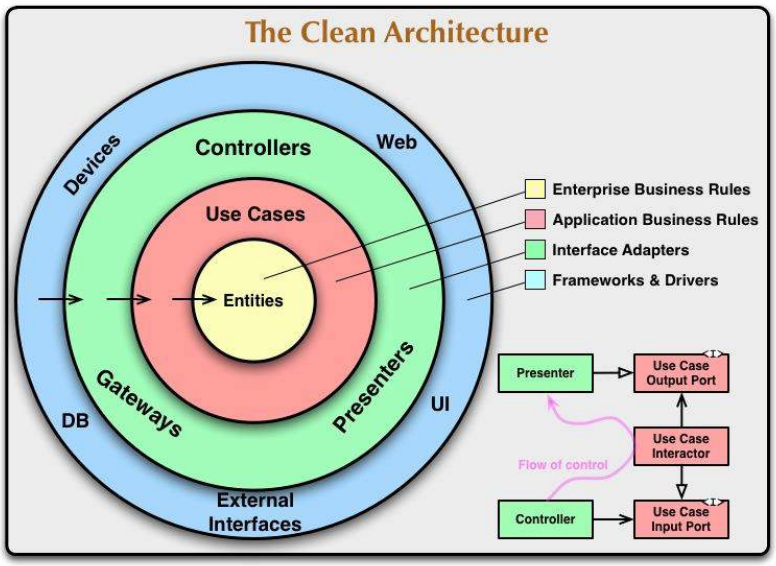
二、Clean Architecture

Clean Architecture (附录 1)是 Uncle Bob 在 2012 年提出的用于构建可扩展、可测试软件系统的概要原则。这些架构产生的系统特点是:

- 框架无关性 - 框架只是一个工具，系统不与框架绑定

- 可被测试 - 业务逻辑与 UI、数据库等隔离，方便单元测试
- UI 无关性 - 不需要修改系统的其它部分，就可以变更 UI，如将 React 替换为 Vue
- 数据库无关性 - 业务逻辑与数据库之间需要进行解耦
- 外部机构 (agency) 无关性 - 系统的业务逻辑，不需要知道其它外部接口，诸如安全、调度、代理等

基于以上原则的系统架构如下图所示，又称洋葱图。



从外到内，分为四层：

- Frameworks & Drivers - 由框架和工具组成，比如各种前端框架，数据库访问工具等。
- Interface Adapters - 作用是转换数据，连接内层与外层
- Application Business Rules - 将多个业务实体封装为高级具体的业务用例
- Enterprise Business Rules - 单个业务实体，可以是具有方法的对象，也可以是一组数据结构和函数

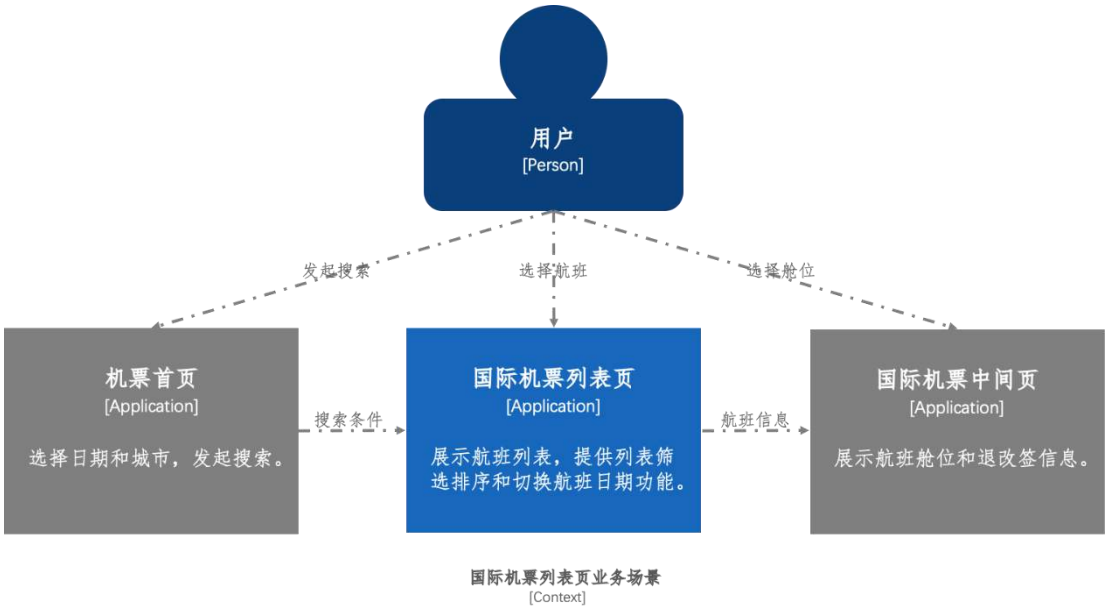
不同层代表软件系统中不同领域，外层是机制(mechanisms)，内层是策略(policies)。

层与层之间遵循一个依赖关系原则：外层指向内层，机制指向策略。内层中的任何东西都不能知道外层中的某些东西。特别是外层中声明的内容的名称不得被内层中的代码提及，包括功能、类、变量或任何其他命名的软件实体。出于同样的原因，外层中使用的数据格式不应该被内层使用，特别是当这些格式是由外层中的框架生成时。外圈中的任何东西不应该影响内圈。

2.1 业务场景

App 国际机票查询预订流程中，列表页负责展示符合用户搜索条件的航班列表，并将用户带入中间页（舱位选择），其业务场景有以下特点：

- 代码量庞大 - 逻辑层 70000 行以上
- 依赖服务多 - 依赖 11 个服务
- 交互复杂 - 筛选、排序、切换日期、低价订阅和查看浮层等
- 展示信息多 - 航班信息、通知公告，推荐航班等
- 页面结构变化 - 单程、往返、多程页面结构不同；不同 ABTesting 页面结构不同

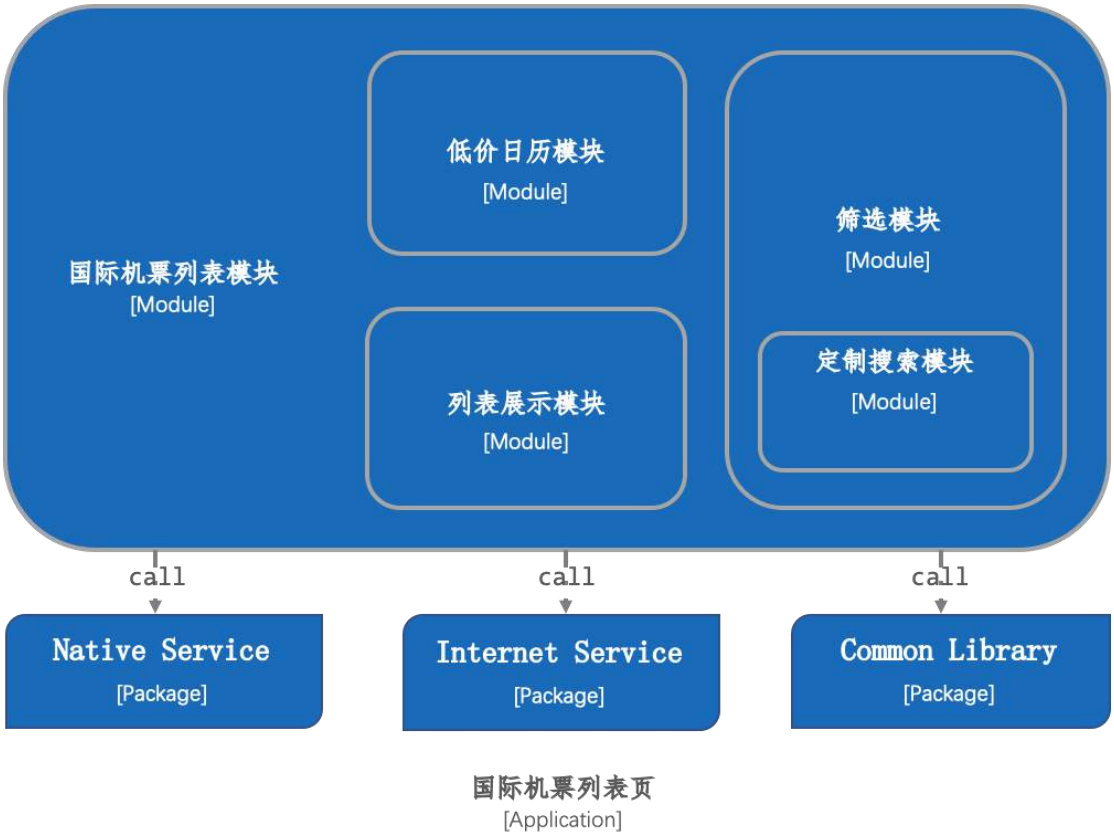


2.2 应用结构

如下图，项目最外层分为公共库和业务两部分。

- 公共库 - 封装了全局可用的公共代码，如与 Native 通信，发起网络请求和其他通用工具类
- 业务部分 - 具体的业务逻辑，由多个同构的业务模块嵌套组合而成，分形结构

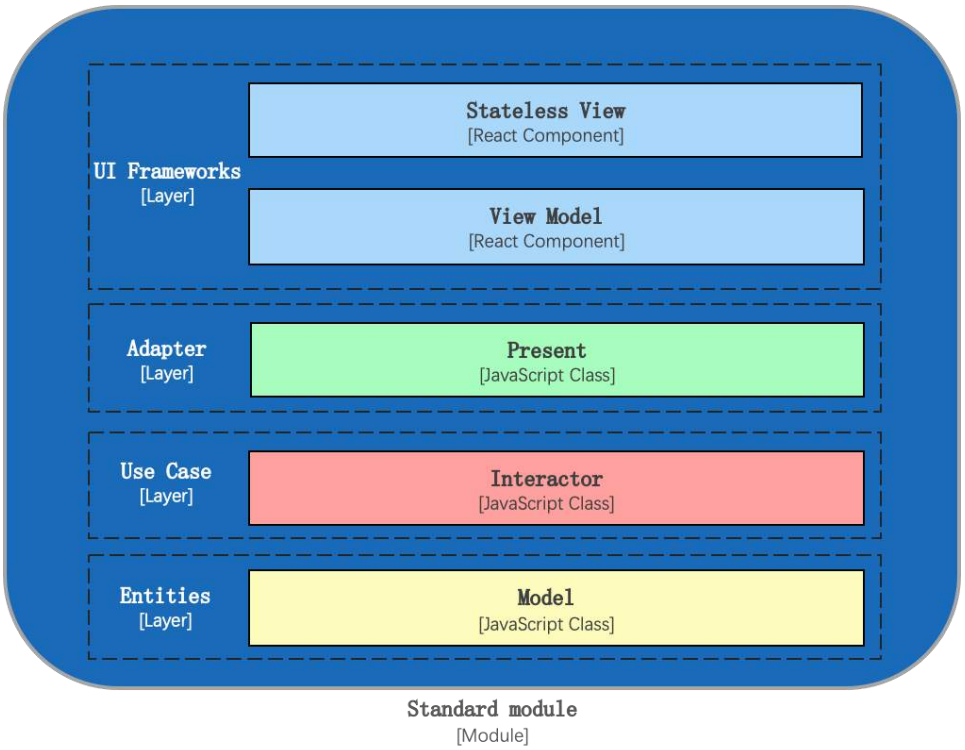
业务部分由多个 Clean Architecture 模块组成，最外层模块处理页面路由和页面初始化数据，低价日历、列表展示和筛选作为子模块嵌套其中。每个模块的内部结构相同，并且可以方便的成为另一个模块的子模块或父模块。



2.3 模块结构

模块内部遵循 Clean Architecture 原则，分为四层：

- ViewModel & StatelessView - React 框架相关代码，只负责界面展示，样式，动画和传递交互事件
- Presenter - 连接 ViewModel 和 Interactor，连接模块内部和外部，不存在业务逻辑
- Interactor - 持有多个 Model，将它们封装成高级的业务逻辑，供 Presenter 调用
- Model - 独立的业务逻辑实体，提供方法给 Interactor 调用



2.4 代码实现

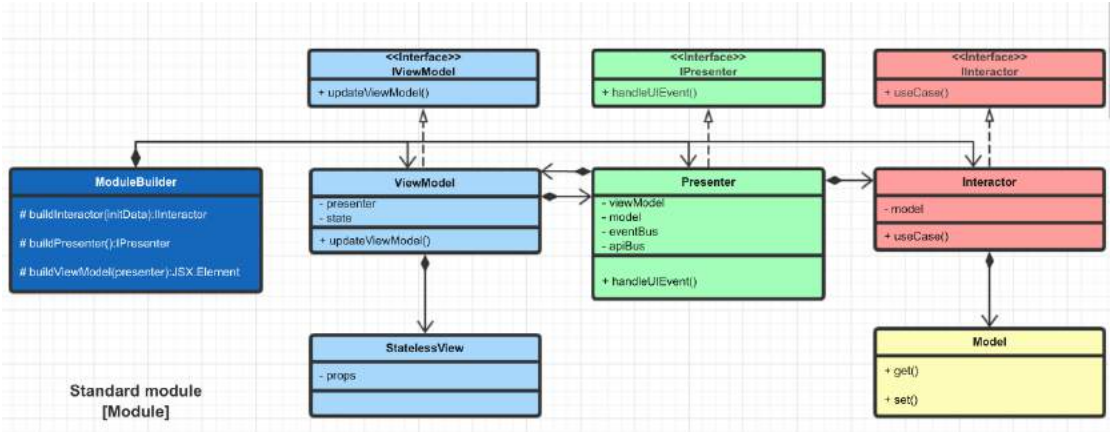
2017 下半年，我们在 React Native 实践初期，就决定全面使用 TypeScript，因为我们期望该技术栈未来能够可靠地支撑大型复杂项目工程。实践证明，Typescript 不负众望，在 2019 年变成了前端技术栈必备技能。

Typescript 补齐了 JavaScript 在数据类型方面的短板，这对大型项目的持续维护和稳定交付非常重要。

TS 类型系统描述了数据结构、function 的入参和返回值的类型和 class 对外暴露的方法，面向接口编程变得可能，我们编码时不再通过阅读代码了解上下文，而是面向接口实现逻辑，消灭 TS error 就好。

TS 对 OOP 友好，对于部分场景，继承和多态是最优解，比如多态的单程、往返、多程列表页。同时，IDE 的支持带来了方便的代码智能提示和跳转，提升了开发效率。

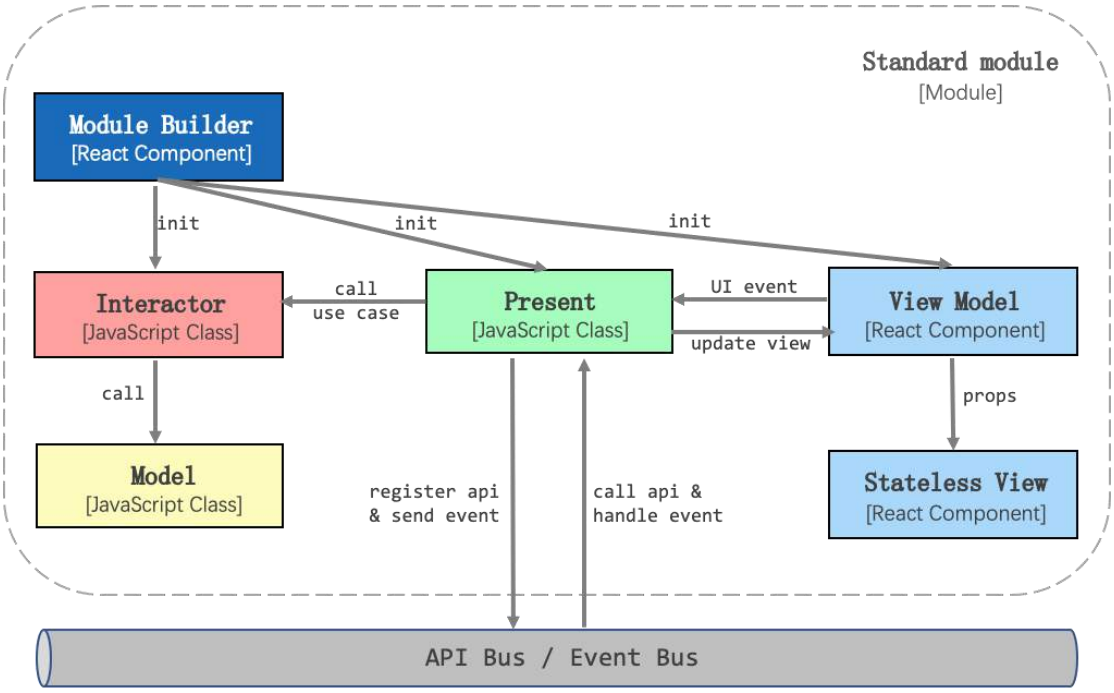
在 TS 加持下，一个标准的模块由以下类和接口组成：



- ModuleBuilder.tsx
  模块的入口，持有父模块传入的初始化参数，通过重写 buildInteractor、buildPresenter、buildViewModel 方法生成 Interactor、presenter、viewModel 实例。
- IViewModel.ts (Interface)
  viewModel 层契约，以接口的形式描述 viewModel 层对 presenter 层暴露的方法，这些方法通常为更新某个 state。
- ViewModel.tsx
  viewModel 层具体实现，持有类型为 IPresenter 的 presenter 实例和多个无状态子组件。UI 交互的响应指向 presenter 暴露的方法，使用 state 持有界面数据，并以 props 的方式下发给无状态子组件。
- StatelessView.tsx
  没有业务逻辑，没有 state，无脑展示 viewModel 下发的 props。
- IPresenter.ts (Interface)
  presenter 层契约，描述暴露给 viewModel 层的方法，通常为响应 UI 交互逻辑。
- Presenter
  presenter 层具体实现，以接口的形式持有 viewModel 和 interactor 对象，关联业务逻辑和界面展示逻辑。持有 eventBus 和 apiBus 对象，用于模块间通信。拥有 onViewModelAttach 和 onViewModelDestroy 生命周期，对应 viewModel 的创建和销毁。
- IInteractor.ts (Interface)
  interactor 层契约，描述暴露给 presenter 层的方法，这些方法表示具体的业务逻辑。
- Interactor.ts
  interactor 层具体实现，持有多个 model 对象，将它们封装为高级的业务用例供 presenter 调用。当只有一个 model 时，interactor 可以不存在，而用唯一的 model 替代。
- Model.ts
  相对独立、内聚的业务实体，暴露方法供 interactor 调用。

2.5 数据流

模块内部数据流、模块与外部通信关系如下：



Dynamic diagram
[Module & Application]

- builder Init
  持有父组件通过 props 传入的模块初始化参数，在生成各层实例时传入对应的构造函数。
- viewModel -> statelessView
  当 viewModel 的 state 被更新时，新的数据通过 props 传递到子组件。
- viewModel -> presenter
  当 viewModel 层监听到交互时，调用 presenter 方法。
- presenter -> viewModel
  当界面需要刷新时，viewModel 的方法被 presenter 调用。
- presenter -> interactor
  当触发某个业务场景时，interactor 的方法被 presenter 调用。
- interactor -> model
  当 presenter 调用 interactor 时，model 的方法被 interactor 调用。
- presenter <-> api bus、event bus
  当模块需要对外暴露 api 和发送事件时，api bus 和 event bus 被 presenter 方法调用；
  当外界需要调用 api 和广播事件时，presenter 的方法被调用。

2.6 具体案例

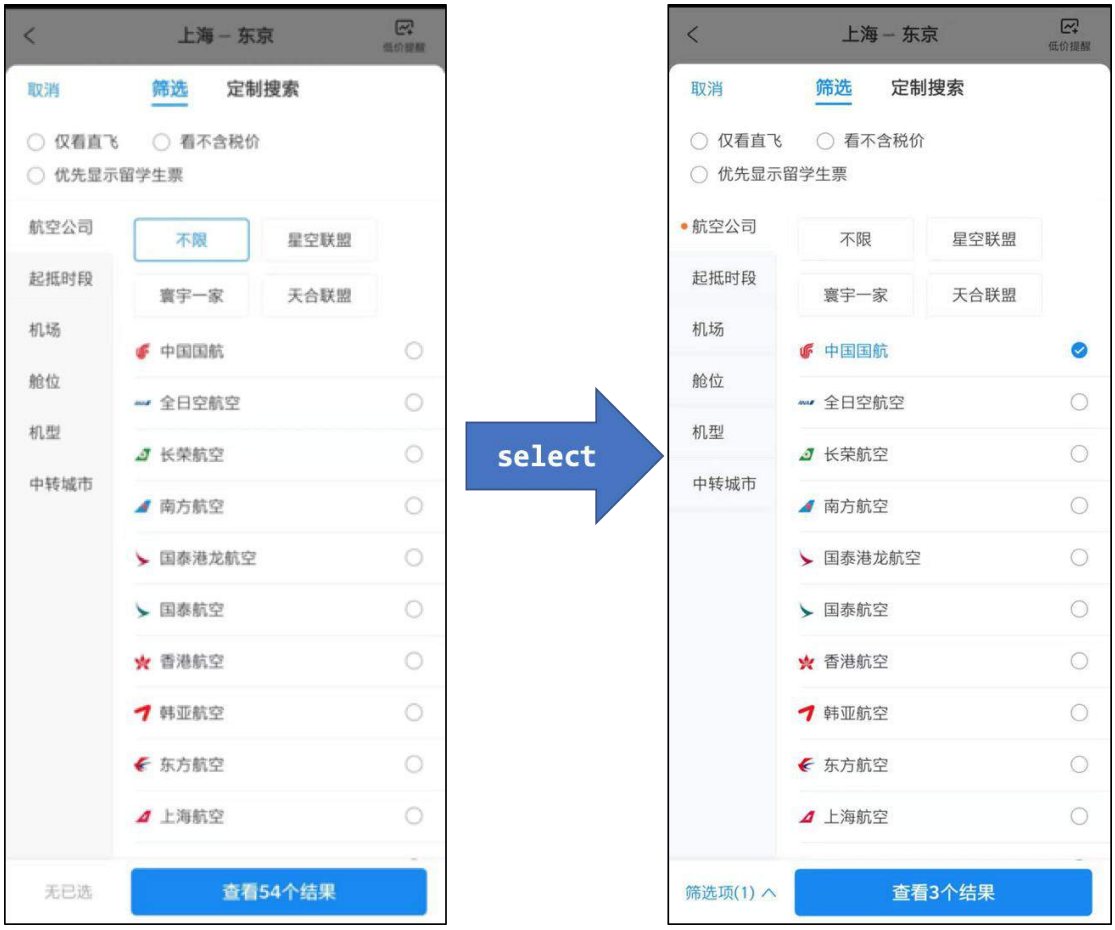
下面以筛选模块为案例，分析模块内部结构设计和数据流向。

筛选模块顶部为三个独立的筛选项；中部左侧为筛选大类栏，中部右侧为已选中大类对应的筛选项列表；底部可展开查看已选筛选项，以及符合当前筛选条件的航班数。

当用户选择中筛选项，如图中选中“中国国航”，会产生四处界面的改变：

- 筛选大类“航空公司” 左侧出现小红点；
- 筛选项“中国国航”被选中；
- 底部查看已选按钮从“无已选”变为“筛选项(1)”
- 底部发起筛选按钮文案从“查看 54 个结果”变为“查看 3 个结果”

这个案例很好地证明了：界面元素在布局关系上的亲密度，与界面状态逻辑的关联性并不成正比。



为了让界面逻辑和业务逻辑都能得到合理的表达，参照 Clean Architecture 原则，模块内部划分为四层。

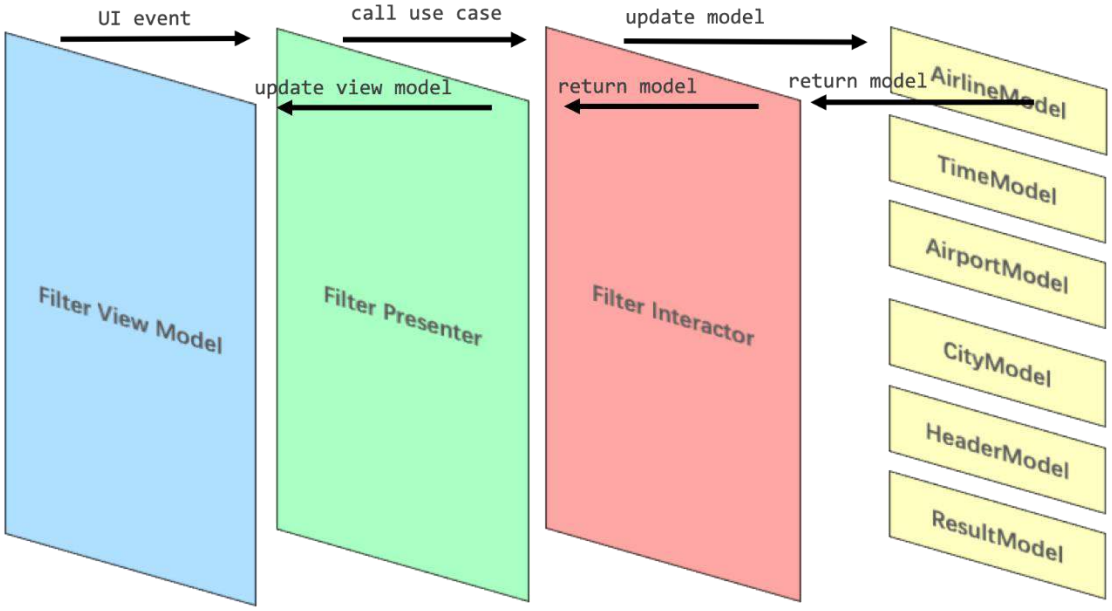
ViewModel 层由多个 React Component 组合嵌套而成，这些勾选框，侧边栏，筛选项列表，按钮等界面元素按照如你所见的布局关系被 JSX 声明式表达为一棵组件树，所见即所得。

Model 层则按照业务逻辑相关性拆分封装为多个业务逻辑高内聚的类：AirlineModel 负责航司筛选逻辑，TimeModel 负责时间筛选逻辑...

Interactor 层是对 Model 层的高级封装，多个 Model 之间存在关联性逻辑包含在这层，例

如“中转城市”与“仅看直飞”选项的互斥关系。

Presenter 层将界面层和逻辑层联系起来，同时也负责筛选模块内部与外界的交流，例如点击“查看 XX 个结果”按钮，就是在 P 层发出 Event，使得监听该事件的模块做出相应。



2.7 易用性

严格分层带来的副作用是要写不少模板代码。为了减少重复模板代码的编写和统一模块结构，我们提供了标准的模板代码。在开发过程中，只需要在模板代码基础上添加业务代码即可，无额外工作量。模板代码目录如下。

```
|____Builder
| |____Builder.tsx
| |____Page.tsx
|____BusStation
| |____OneSimpleEventDescription.ts
| |____OneSimpleApiDescription.ts
|____Contract
| |____IViewModel.ts
| |____IPresenter.ts
| |____IModel.ts
|____Model
| |____ModelOne.ts
| |____ModelTwo.ts
|____Interactor
| |____Interactor.ts
|____Presenter
| |____Presenter.ts
```

```
|____ViewModel
| |____ViewModel.tsx
|____View
| |____StatelessView.tsx
|____Log
| |____LogInfo.ts
```

为了提高模块编码的易用性，我们提供了各层的基类实现。各层派生于以下基类：

- JetModuleBuilder.tsx
- JetViewModel.tsx
- JetPresenter.ts
- JetInteractor.ts

三、Why not use React Component

为什么不采用 React 的组件化设计，将状态逻辑放到 Component 内部？

回顾 Thinking in react (附录 2): 模块由多个 Component 组成，state 放置在负责展示他们的 Component 中。当业务场景变得复杂后，会出现这些问题：

- 在组件之间复用状态逻辑变得困难 - Component 的层次结构，对布局和界面展示友好，对业务逻辑不友好。业务上不相关的 state 组合在一个 Component 中，破坏业务逻辑的内聚，导致业务代码难以测试、复用和维护。

- 混乱的 componentWillReceiveProps - React 的数据流自上而下，当业务逻辑同时依赖 props 和 state 时，必须在 componentWillReceiveProps 中判断是否对应的 props 被改变。

```
// 混乱的 componentWillReceiveProps
public componentWillReceiveProps(nextProps) {
  if (
    nextProps.Filter_Model.changedTopFilter &&
    this.props.Filter_Model.changedTopFilter !== nextProps.Filter_Model.changedTopFilter
&&
    nextProps.Filter_Model.changedTopFilter.length > 0
  ) {
    const directFlightOnly = this.props.Filter_Model.isDirectFlight;
    const changedList = new CompareLists().findDifferentItemNumber(
      this.flightListFilterObj.getOriginalTab(),
      nextProps.Filter_Model.changedTopFilter
    );
    this.hasFilterChanged = changedList.length > 0 || directFlightOnly;
  }
```

```
  if (nextProps.Filter_Model.selectedFilterCount !==
this.props.Filter_Model.selectedFilterCount) {
    this.updateSelectedFilterCount(nextProps.Filter_Model.selectedFilterCount);
  }
  if (nextProps.Filter_Model.AirportList && !this.props.Filter_Model.AirportList) {
    this.setState({
      selectedFilterCount: this.getSelectedFilterCount(nextProps)
    });
  }
}
```

针对以上问题，React 提供了解决方案：状态提升、高阶组件和 Render Props。

参照此思路，多个逻辑关联强的 Component 的 state，被提升到一个 Container 中统一管理，其余 Component 变成了 Stateless Component ，只负责界面展示。但是实践中遇到新问题：

- 复杂组件变得难以理解。随着组件复杂度提高，生命周期中被逻辑不相关的副作用充斥，这很容易产生缺陷。

四、Why not use React Hook

React Conf 2018 会议上，React 的开发者指出 Class Component 存在的 3 个问题：

- Wrapper hell - 现有解决组件间状态逻辑复用的方案会破坏项目的组织结构，使项目变得难理解，抽象层组件会形成“嵌套地狱”。
- Huge components - 充斥各种逻辑的复杂组件难以理解。
- Confusing classes - JS 对 Class 的支持不好，冗余代码多。

并认为这些问题的根因是：React doesn't provide a stateful primitive simpler than a class component。最终给出的解决方案：Hook。

为了复用组件间状态逻辑，可以将逻辑封装为一个 Hook，供其他组件使用。为了 Class Component 的生命周期方法不被不相关的状态逻辑和副作用充斥，则换做在 Function Component 重复使用 Effect Hook ，将这些逻辑进行分类。同时，相较于在 Class 要写类似 bind 的代码，Function Component 可以少写很多代码。

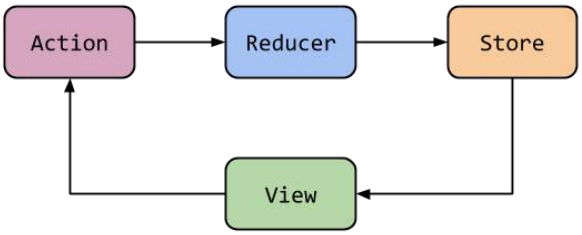
```
// 使用 Class Component 的计数器
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }


  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Click me
        </button>
      </div>
    );
  }
}


// 使用 Hook 的计数器
function Example() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
```

诚然，Hook 的出现，能帮助开发者更好的管理 Component 的 state 和 state logic，但是当面对复杂业务场景时，仍然需要考虑几个问题。

- React 只是构建用户界面的框架。
- 组件树的结构利于描述布局逻辑，但对于业务逻辑不够友好。
- 在完成从 Native 迁移 React Native 技术栈之后，后续如果需要移植到小程序或 Flutter 如何成本最低？

Hook 并不能很好的解决这些问题，而 Clean Architecture 则是参考答案。如果说 Hook 的出现，是为了让开发者更方便地把 state 放入 Component ，那么 Clean Architecture 则是让开发者不要把 state 放入 Component 中。

五、Why not use Redux



同样能做到和业务逻辑和界面展示解耦，为什么不使用 Redux ？

作为 Unidirectional Architecture 类架构的经典，Redux 有其独特的优势：单向数据流和状态可预测。对于逻辑复杂度中等以下的 Web 网站和 App 工程，Redux 可以很好地提升开发体验。但是针对 App 国际机票列表页这样比较复杂（至少我们认为）的业务场景，它略显不足：

- 单一数据源(Store)变大后维护困难。
  单例 Store 在复杂业务场景下会变得庞大，所有全局状态包含其中，所有 Reducer 都拥有修改权限。当我们想修改或删除一个这样的 state 时，不得不把所有的 Reducer 和 mapStateToProps 代码阅读一遍，以确保改动不会影响到其他逻辑。
- Action 和 Reducer 维度的职责划分方式容易导致低内聚。
  Redux 项目中，通常会将所有 Action 放入一个文件，所有 Reducer 放入另一个文件。这样的职责划分无法将业务关联紧密的逻辑封装起来，导致每次修改都要小心翼翼。
- Action 使用字符串区分，留下隐患。
  新建 Action 时，需要人工确认避免用于区分 Type 的字符串冲突。
- 无法独立出子模块。
  所有组件都依赖集中的单例 Store ，当需要将组件改造成为一个独立模块，复用于其他项目时，修改工作量较大。

六、总结

App 客户端技术栈从原生快速迁移到 React Native 之类的混合技术方案，平台 API 变了，编程语言变了，但不变的是业务复杂性。

为了摆脱基于界面元素在布局关系上编写状态逻辑，我们放弃 Component 和 Hook 方案。为了前端模块化和整体分形的项目结构，我们放弃 Redux 方案。

Clean Architecture 不仅带来了逻辑与界面分离和统一的模块结构，还降低了单元测试的难

度，减少了前端技术栈迁移的成本，同时加快了排查问题的速度，方便多团队间代码协作。

目前新架构洋葱版国际机票列表页已经全量上线运行一段时间，效果良好：

- 整个项目由 26 个标准模块组成
- Bug 总数相比旧版减少 71.3%
- UI 自动化测试用例覆盖率达到 86%
- 研发效率相比旧版提升 48.5%

附录

1、Clean Architecture

https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

2、Thinking in react

https://react.docschina.org/docs/thinking-in-react.html

万字长文全面解析 GraphQL，携程微服务背景下的前后端数据交互方案

【作者简介】古映杰，携程研发高级经理，负责前端框架和基础设施的设计、研发与维护。开源项目 react-lite 和 react-imvc 作者。

前言

随着多终端、多平台、多业务形态、多技术选型等各方面的发展，前后端的数据交互，日益复杂。

同一份数据，可能以多种不同的形态和结构，在多种场景下被消费。

在理想情况下，这些复杂性可以全部由后端承担。前端只管从后端接口里，拿到已然整合完善的数据。

然而，不管是因为后端的领域模型，还是因为微服务架构。作为前端，我们感受到的是，后端提供的接口，越发不够前端友好。我们必须自行组合多个后端接口，才能获取到完整的数据结构。

面向领域模型的后端需求，跟面向页面呈现的前端需求，出现了不可调和的矛盾。

我们曾经试图用 RESTful 风格的无线 API 聚合层来解决问题。这一层完全由后端工程师开发和迭代（前端作为下游等待聚合接口的契约），他们既要理解背后对接的微服务体系，又要理解每个前端页面的展现需求。

随着页面数量的增加，接口调用方越来越多，后端 API 聚合层的逻辑越来越重。聚合层自身成为新的瓶颈所在。实际情况是，作为中间层的后端，既难以充分对接背后的微服务体系，也难以充分理解前端的页面展现需求。

他们的整体定位很尴尬，代码里既有 UI 展示相关的话术等信息，又有大量接口相关的业务逻辑处理。每次发布迭代，都要跟上下游做大量的沟通和联调。

我们最终意识到，让最了解页面数据逻辑的前端接管中间层，让后端回到他们擅长的领域模型，可能是更好的做法。可以减少大家沟通成本，提高前后端的专业化程度，降低联调压力，加快彼此的开发效率。

在这种背景下，我们最后选择使用 Node.js 搭建专门服务于前端页面呈现的后端，亦即 Backend-For-Frontend，简称 BFF。

我们面临了很多不同的技术选型，主要围绕在权衡 RESTful API 和 GraphQL。

正如标题所示，我们最终选用的是 GraphQL as BFF。

本文将介绍我们对 GraphQL 所作的考察、探索、权衡、技术选型与设计等多方面的内容，希望能给大家带来一些启发。

一、GraphQL 模式出现的必然性

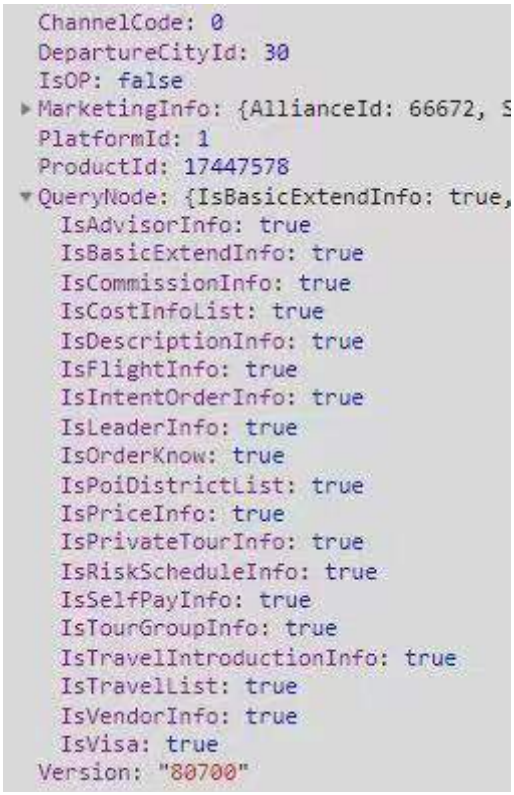
面向前端页面的数据聚合层，其接口很容易在迭代过程中，变得愈加复杂；最终发展成一个超级接口。

它有很多调用方，各种不同的调用场景，甚至多个不同版本的接口并存，同时提供数据服务。

所有这些复杂性，都会反映到接口参数上。

接口调用的场景越多，它对接口参数结构的表达能力，要求越高。如果只有一个 boolean 类型的参数，只能满足 true | false 两种场景罢了。

以产品详情接口为例，一种很自然的请求参数结构如下：



里面包含 ChannelCode 渠道信息，IsOp 身份信息，MarketingInfo 营销相关的信息，PlatformId 平台信息，QueryNode 查询的节点信息，以及 Version 版本信息。最核心的参数 ProductId，被大量场景相关的参数所围绕。

审视一下 QueryNode 参数，很容易可以发现，它正是 GraphQL 的雏形。只不过它用的是

更复杂的 JSON 来描述查询字段，而 GraphQL 用更简洁的查询语句，完成同样的目的。

并且，QueryNode 参数，只支持一个层级的字段筛选；而 GraphQL 则支持多层级的筛选。

GraphQL 可以看作是 QueryNode 这种形式的参数设计的专业化。相比用 JSON 来描述查询结果，GraphQL 设计了一个更完整的 DSL，把字段、结构、参数等，都整合到一起。

仿照格林斯潘第十定律：

任何 C 或 Fortran 程序复杂到一定程度之后，都会包含一个临时开发的、不合规范的、充满程序错误的、运行速度很慢的、只有一半功能的 Common Lisp 实现。https://zh.wikipedia.org/wiki/%E6%A0%BC%E6%9E%97%E6%96%AF%E6%BD%98%E7%AC%AC%E5%8D%81%E5%AE%9A%E5%BE%8B

或许可以说：

任何接口设计复杂到一定程度后，都会包含一个临时开发的、不合规范的、只有一半功能的 GraphQL 实现。

从 SearchParams， FormData 到 JSON，再到 GraphQL 查询语句，我们看到不断有新的数据通讯方式出现，满足不同的场景和复杂度的要求。

站在这个层面上看，GraphQL 模式的出现，有一定的必然性。

二、GraphQL 语言设计中的必然性

作为一个查询相关的 DSL，GraphQL 的语言设计，也不是随意的。

我们可以做一个思想实验。

假设你是一名架构师，你接到一项任务，设计一门前端友好的查询语言。要求：

1）查询语法跟查询结果相近
2）能精确查询想要的字段
3）能合并多个请求到一个查询语句
4）无接口版本管理问题
5）代码即文档

我们知道查询结果是 JSON 数据格式。而 JSON 是一个 key-value pair 风格的数据表示，因此可以从结果倒推出查询语句。

上图是一个查询结果。很显然，它的查询语句不可能包含 value 部分。我们删去 value 后，它变成下面这样。

查询语句跟查询结果拥有相同的 key 及其层次结构关系。这是我们想要的。

我们可以再进一步，将冗余的双引号，逗号等部分删掉。

我们得到了一个精简的写法，它已经是一段合法的 GraphQL 查询语句了。

其中的设计思路和过程是如此简单直接，很难想象还有别的方案比目前这个更满足要求。

当然，只有字段和层级，并不足够。符合这种结构的数据太多了，不可能把整个数据库都查询出来。我们还需要设计参数传递的方式，以便能缩小数据范围。

上图是一个自然而然的做法。用括号表示函数调用，里面可以添加参数，可谓经典的设计。

它跟 ES2015 里的 （Method Definitions Shorthand） 也高度相似。如下所示：

```
class Query {
  UserInfo(userId) {
    let userInfo = fetch('/api/getUserInfo', { userId })
    return {
      ...userInfo
    }
  }
}
```

前面演示的 GraphQL 参数写法，参数值用的是字面量 userId: 123。这不是一个特别安全的做法，开发者会在代码里，用拼接字符串的方式将字面量值注入到查询语句，也就给了恶意攻击者注入代码的机会。

我们需要设计一个参数变量语法，明确参数位置和数量。

```
{
    UserInfo(userId: $id) {
      Uid
      UserName
      UserPhoto
      Contact
      Mobile
      Email
      CountryCode
      CountryName
      MobilePhoneForeign
      CountryCodeForeign
      BindMobilePhone
      BindCountryCode
      BindEmail
    }
}
```

我们可以选用 $xxx 这种常见的标记方法，它被很多语言采用来表示变量。沿用这种风格，可以大大减少开发者的学习成本。

前后端通讯的另一个痛点是，命名。前端经常吐槽后端的字段名过于冗长，或者不知所云，或者拼写错误，或者不符合前端表述习惯。最常见的情况是，后端字段名以大写字母开头，而前端习惯 Class 或者 Component 是大写字母开头，实例和数据，则以小写字母开头。

我们期望有机会进行字段名调整。

别名映射（Alias）语法，正是为了这个目的而出现的。

```
{
    user: UserInfo {
        uid: Uid
        name: UserName
        avatar: UserPhoto
        contact: Contact
        mobile: Mobile
        email: Email
        countryCode: CountryCode
        countryName: CountryName
    }
}
```

上面这种别名映射的语法，在其它语言里也很常见。如果不这样写，顶多就是变成：

uid as Uid 或者 uid = Uid 这类做法，差别不大。我认为选用冒号更佳，它跟 ES2015 的解构语法很接近。

```
const {
  Uid: uid,
  UserName: name,
  UserPhoto: avatar,
  Contact: contact,
  Mobile: mobile,
  Email: email,
  CountryCode: countryCode,
  CountryName: countryName
} = UserInfo
```

至此，我们拥有了 key 层级结构，参数传递，变量写法，别名映射等语法，可以编写足够复杂的查询语句了。不过，还有几个小欠缺。

比如对字段的条件表达。假设有两次查询，它们唯一的差别就是，一个有 A 字段，另一个没有 A 字段，其它字段及其结构都是相同的。为了这么小的差别 ，前端难道要编写两个查询语句？

这显然不现实，我们需要设计一个语法描述和解决这个问题。

它就是——指令（Directive）。

```
{
    UserInfo {
      Uid
      UserName
      UserPhoto
      Contact
      Mobile @include(if: $needMobile)
      Email
      CountryCode
      CountryName
      MobilePhoneForeign
      CountryCodeForeign
      BindMobilePhone
      BindCountryCode
      BindEmail
    }
}
```

指令，可以对字段做一些额外描述，比如

@include，是否包含该字段；
@skip，是否不包含该字段；
@deprecate，是否废弃该字段；

除了上述默认指令外，我们还可以支持自定义指令等功能。

指令的语法设计，在其它语言里也可以找到借鉴目标。Java，Phthon 以及 ESNext 都用了 @ 符号表示注解、装饰器等特性。

有了指令，我们可以把两个高度相似的查询语句，合并到一起，然后通过条件参数来切换。这是一个不错的做法。不过，指令是跟着单个字段走的，它不能解决多字段的问题。

比如，字段 A 和字段 B，拥有相同的总体结构，仅仅只有 1 个字段名的差异。前端并不想编写一样的 key 值重复多次。

这意味着，我们需要设计一个片段语法（Fragment）。

```
fragment User on A {
    Uid
    UserName
    UserPhoto
}

query {
    user1(id: 1) {
      ...User
      Email
    }
    user2(id: 2) {
        ...User
        Mobile
    }
}
```

如上所示，用 fragment 声明一个片段，然后用三个点表示将片段在某个对象字段里展开。我们可以只编写一次公共结构，然后轻易地在多个对象字段里复用。

这种设计也是一个经典做法，跟 JavaScript 里的 Spread Properties 很相近。

```
var obj1 = { foo: 'bar', x: 42 };
var obj2 = { foo: 'baz', y: 13 };

var clonedObj = { ...obj1 };
// Object { foo: "bar", x: 42 }

var mergedObj = { ...obj1, ...obj2 };
// Object { foo: "baz", x: 42, y: 13 }
```

至此，我们得到了一个相对完整的，对前端友好的查询语言设计。它几乎就是 GraphQL 当前的形态。

如你所见，GraphQL 的查询语言设计，借鉴了主流开发语言里的众多成熟设计。使得任何拥有丰富的编程经验的开发者，很容易上手 GraphQL。

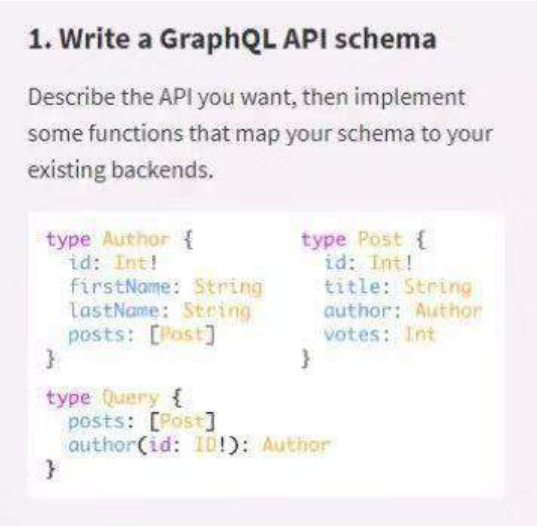
按照同样的要求，重新来一遍，大概率得到跟当前形态高度接近的设计。这是我理解的 GraphQL 语言设计里包含的必然性。

三、GraphQL 的组成与链路

查询语法，是 GraphQL 面向前端，或者说面向数据消费端的部分。

除此之外，GraphQL 还提供了面向后端，或者说面向数据提供方的部分。它就是基于 GraphQL 的 Type System 构建的 Schema。

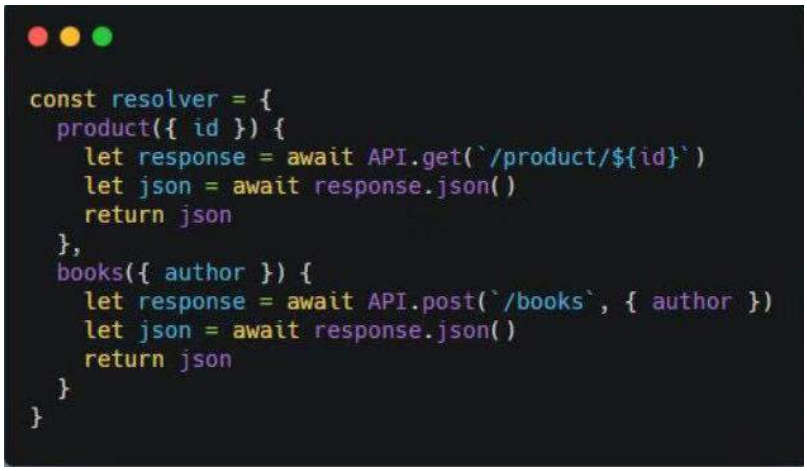
一个 GraphQL 服务和查询的链路，大致如下：



首先，服务端编写数据类型，构建一个数据结构之间的关联网络。其中 Query 对象是数据消费的入口。所有查询，都是对 Query 对象下的字段的查询。可以把 Query 下的字段，理解为一个个 RESTful API。比如上图中的，Query.post 和 Query.author，相当于 /post 和 /author 接口。

GraphQL Schema 描述了数据的类型与结构，但它只是形状 (Shape)，它不包含真正的数据。我们需要编写 Resolver 函数，在里面去获取真正的数据。

Resolver 的简单形式如下



每个 Query 对象下的字段，都有一个取值函数，它能获取到前端传递过来的 query 查询语句里包含的参数，然后以任意方式获取数据。Resolver 函数可以是异步的。

有了 Resolver 和 Schema，我们既定义了数据的形状，也定义了数据的获取方式。可以构

建一个完整的 GraphQL 服务。

但它们只是类型定义和函数定义，如果没有调用函数，就不会产生真正的数据交互。

前端传递的 query 查询语句，正是触发 Resolver 调用的源头。

如上所示，我们发起了查询，传递了参数。GraphQL 会解析我们的查询语句，然后跟 Schema 进行数据形状的验证，确保我们查询的结构是存在的，参数是足够的，类型是一致的。任何环节出现问题，都将返回错误信息。

数据形状验证通过后，GraphQL 将会根据 query 语句包含的字段结构，一一触发对应的 Resolver 函数，获取查询结果。也就是说，如果前端没有查询某个字段，就不会触发该字段对应的 Resolver 函数，也就不会产生对数据的获取行为。

此外，如果 Resolver 返回的数据，大于 Schema 里描绘的结构；那么多出来的部分将被忽略，不会传递给前端。这是一个合理的设计。我们可以通过控制 Schema，来控制前端的数据访问权限，防止意外的将用户账号和密码泄露出去。

正是如此，GraphQL 服务能实现按需获取数据，精确传递数据。

四、澄清关于 GraphQL 的几个迷思

有相当多的开发者，对 GraphQL 有各种各样的误解。在这里挑选几个重要的例子，加以澄清，帮助大家更全面的认识 GraphQL。

4.1 GraphQL 不一定要操作数据库

有一些开发者认为 GraphQL 需要操作数据库，因此实现起来，几乎等于要推翻当前后端的所有架构。这是一个重大误解。

GraphQL 不仅可以不操作数据库，它甚至可以不从其它地方获取数据，而直接写死数据在 Resolver 函数里。查看 graphql.js 的官方文档，我们轻易可以找到案例：

```
var { graphql, buildSchema } = require('graphql');

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

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

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

上图定义了一个 schema，只有一个类型为 String 的 hello 字段，它的 resolver 函数里，无视所有参数，直接 return 一个 hello world 字符串。

可以看到，GraphQL 只是关于 schema 和 resolver 的一一对应和调用，它并未对数据的获取方式和来源等做任何假设。

4.2 GraphQL 跟 RESTful API 不是对立的

在网络上，有相当多的 GraphQL 文章，将它跟 RESTful API 对立起来，仿佛要么全盘 GraphQL，要么全盘 RESTful API。这也是一个重大误解。

GraphQL 和 RESTful API 不仅不对立，还是互相协作的关系。

在前面关于 Resolver 函数的图片中，我们看到，可以在 GraphQL Schema 的 Resolver 函数里，调用 RESTful API 去获取数据。

当然，也可以调用 RPC 或者 ORM 等方式，从别的数据接口或者数据库里获取数据。

因此，实现一个 GraphQL 服务，并不需要挑战当前整个后端体系。它具有高度灵活的适配能力，可以低侵入性的嵌入当前系统中。

4.3 GraphQL 不一定是一个后端服务

尽管绝大多数 GraphQL，都以 server 的形式存在。 但是，GraphQL 作为一门语言，它并没有限制在后端场景。

```
var { graphql, buildSchema } = require('graphql');

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

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

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

上图还是前面展示过的 graphql.js 的官方文档，最下面一行，就是一个普通的函数调用，它发起了一次 graphql 查询，其 response 结果如下：

```
{
  "data": {
    "hello": "Hello world!"
  }
}
```

这段代码，不只能在 node.js 里运行，在浏览器里也可以运行（可访问：https://codesandbox.io/s/hidden-water-zfq2t 查看运行结果）

因此，我们完全可以将 GraphQL 用在纯前端，去实现 State Management 状态管理。Relay 等框架，即包含了用在前端的 graphql。

4.4 GraphQL 不一定需要 Schema

这是一个有趣的事实，GraphQL 语言设计里的两个组成部分：

1）数据提供方编写 GraphQL Schema；
2）数据消费方编写 GraphQL Query；

这种组合，是官方提供的最佳实践。但它并不是一个实践意义上的最低配置。

GraphQL Type System 是一个静态的类型系统。我们可以称之为静态类型 GraphQL。此外，社区还有一种动态类型的 GraphQL 实践。

graphql-anywhere: Run a GraphQL query anywhere, without a GraphQL server or a schema.
https://github.com/apollographql/apollo-client/tree/master/packages/graphql-anywhere

它跟静态类型的 GraphQL 差别在于，没有了基于 Schema 的数据形状验证阶段，而是直接无脑地根据 query 查询语句里的字段，去触发 Resolver 函数。

它也不管 Resolver 函数返回的数据类型对不对，获取到什么，就是什么。一个字段，不必先定义好，才能被前端消费，它可以动态的计算出来。

在某些场景下，动态类型的 GraphQL 有一定的便利性。不过，它同时丧失了 GraphQL 的部分精髓，这块后面将会详细描述。

值得一提的是，不管是静态类型的 GraphQL 还是动态类型的 GraphQL，都是既可以运行在服务端，也可以运行在前端。

4.5 GraphQL 不一定返回 JSON 数据格式

这是另一个有趣的事实。最初我们演示了，如何基于 JSON 数据结果，反推出 GraphQL 查询语法的设计。而现在，我们却说 GraphQL 可以不返回 JSON 数据格式。

没错。当一个新事物出现之后，随着它的不断发展，它可以脱离其初衷，衍生出不同的形态。

```
const query = gql`
  {
    div {
      s1: span(id: "my-id") {
        text(value: "This is text")
      }
      s2: span
    }
  }
`;

assert.equal(
  renderToStaticMarkup(gqlToReact(query)),
  '<div><span id="my-id">This is text</span><span></span></div>'
);
```

上图还是来自 graphql-anywhere 里的例子。

在这里，它实现了一个 gqlToReact 的 Resolver，可以把一个 graphql 查询转换为 ReactElement 结构。

不只是动态类型的 GraphQL 有这个能力，静态类型的 GraphQL 也有可能实现一样的效果。

不过这种做法，目前仅仅停留在能力演示阶段。其妙用还有待社区去挖掘和探索。

五、GraphQL 的几种使用模式

到目前为止，我们见识到了 GraphQL 的高自由度和灵活性。在搭建 GraphQL Server 时，也可以根据实际需求和场景，采用不同的模式。

5.1 RESTful-Like 模式

这个模式就是简单粗暴的把 RESTful API 服务，替换成 GraphQL 实现。之前有多少 RESTful 服务，重构后就有多少 GraphQL 服务。它是一个简单的一对一关系。

默认情况下，面向两个 GraphQL 服务发起的查询是两次请求，而不是一次。举个例子：

前端需要产品数据时，从之前调用产品相关的 RESTful API，变成查询产品相关的 GraphQL。不过，需要订单相关的数据时，可能要查询另一个 GraphQL 服务。

有一些公司拿 GraphQL 小试牛刀时，采取了这个做法；将 GraphQL 用在特定服务里。

不过，这种模式难以发挥 GraphQL 合并请求和关联请求的能力。只是起到了按需查询，精确查询字段的作用，价值有限。

因此，他们在实践后，发现收效甚微；认为 GraphQL 不过如此，还不如 RESTful API 架构简单和成熟。

其实这是一种选型上的失误。

5.2 GraphQL as an API Gateway 模式

在这个模式里，GraphQL 接管了前端的一整块数据查询需求。

前端不再直接调用具体的 RESTful 等接口，而是通过 GraphQL 去间接获取产品、订单、搜索等数据。

在 GraphQL 这个中间层里，我们将各个微服务，按照它们的数据关联，整合成一个基于 GraphQL Schema 的数据关系网络。前端可以通过 GraphQL 查询语句，同时发起对多个微服务的数据的获取、筛选、裁剪等行为。

值得一提的是，作为 API Gateway 的 GraphQL 服务，可以在其 Resolver 内，向前面提到的 RESTful-like 的 GraphQL 发起查询请求。

如此，既避免了前端需要一对多的问题，也解决了 API Gateway GraphQL 需要请求 RESTful 全量数据接口的内部冗余问题。让服务到服务之间的数据调用，也可以做到更精确。

GraphQL 服务是一个对数据消费方友好的模式。而数据消费方，既可以是前端，也可以是其它服务。

当数据消费方是其它服务时，通过 GraphQL 查询语句，彼此之间可以更精确获取数据，避免冗余的数据传输和接口调用。

当数据消费方是前端时，由于前端需要跟多个数据提供方打交道，如果每个数据提供方都是单独的 GraphQL，那并不能得到本质上的改善。此时若有一个 Gateway 角色的 GraphQL，可以真正减少前端调用的复杂度。

5.2.1 两类 GraphQL API Gateway 服务

同样是 API Gateway 角色的 GraphQL 服务，在实现方式上也有不同的分类。

1）包含大量真实的数据操作和处理的 GraphQL
2）转发数据请求，聚合数据结果的 GraphQL

第一类，是传统意义上的后端服务；第二类，则是我们今天的重点，GraphQL as BFF。

这两类 GraphQL 服务的要求是不同的，前者可能包含大量 CPU 密集的计算，而后者总体而言主要是 Network I/O 相关的行为。

许多公司并不提倡使用 Node.js 构建第一种服务，不管是构建 RESTful 还是 GraphQL。我们也一样。

因此，后面我们讨论的 GraphQL，如果没有特别声明，都可以理解为上面所说的第二种类型。

5.3 GraphQL as a Backend Framework

在澄清关于 GraphQL 的迷思时，我们指出，GraphQL 可以不作为 Server。

这意味着，一个包含 GraphQL 实现的 Server，不一定通过 GraphQL 查询语句进行前后端数据交互，它可以继续沿用 RESTful API 风格。

也就是说，我们可以把 GraphQL 当作一个服务端开发框架，然后在 RESTful 的各个接口里，发起 graphql 查询。

不管是前端还是其它后端服务，都不必知道 GraphQL 的存在。前端的调用方式，还是 RESTful API，在 RESTful 服务内部，它自己向自己发起了 GraphQL 查询。

那么，这个模式有什么好处跟价值？

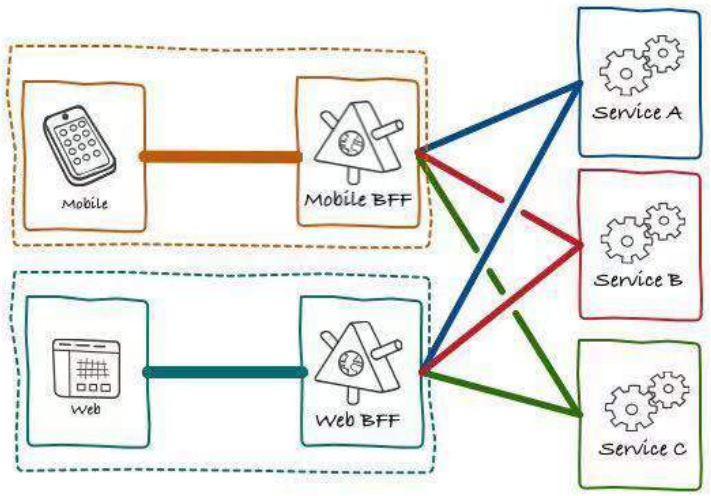
设想一下，你用 RESTful API 风格实现 BFF。由于 PC 端和移动端的场景不同，它们对同一份数据的消费方式差异很大。

在 PC 端，它可以一次请求全量数据。

在移动端，因为它屏幕小，它要分多次去请求数据。首屏一次，非首屏一次，滚动按需加载 N 次，多个 2 级页面里 M 次。

我们要么实现一个超级接口，根据请求参数适配不同场景（即实现一个半吊子的 GraphQL）；要么实现多个功能相似，但又不同的 RESTful 接口。

其中的差异太大了，所以很多公司索性就把 BFF 分成，PC-BFF 和 Mobile-BFF 两个 BFF 服务。



我们可以把 PC-BFF 和 Mobile-BFF 整合成一个 GraphQL-BFF 服务。即便前后端不通过 GraphQL 查询语句进行交互，我们也可以在各个接口里，编写相对简单的查询语句，代替更高成本的接口实现。

也即是说，使用 GraphQL 搭建 BFF，如果出现前后端分工、沟通等方面的矛盾。我们可以将 GraphQL 服务降级为 RESTful 服务，无非就是把需要前端编写的查询语句，写死在后端接口里面罢了。

如果实现的是 RESTful 服务，要转换成 GraphQL 服务，就没有那么简单了。

有了这种优雅降级的能力，我们可以更加放心大胆的推动 GraphQL-BFF 方案。

六、GraphQL 精髓

理解 GraphQL 的精髓所在，可以帮助我们更正确地实践 GraphQL。

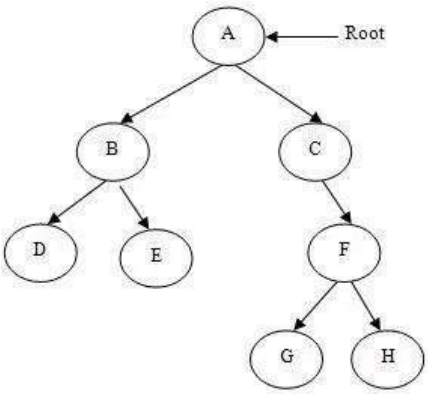
首先来想一下，GraphQL 为什么要叫 GraphQL，其中的 Graph 体现在什么地方？

GraphQL 的查询语句，看起来是 JSON 写法的一种简化。而 JSON 是一个 Tree 树形数据结构。为什么不叫 TreeQL，而是 GraphQL 呢？

6.1 Tree VS Graph

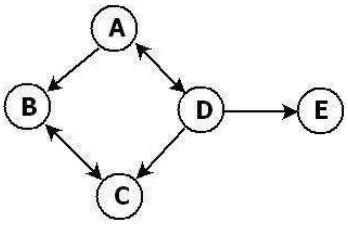
一个重要的前置知识是，什么是 Tree，什么是 Graph，它们有什么关系？

下图是一个 Tree 的结构示意图。



Tree 有且只有一个 Root 节点，并且对于每个非 Root 节点，有且只有一个父节点；它们组成了一个层次结构。其中任意两个节点，有且只有一条连接路径；没有循环，也没有递归引用。

下图是一个 Graph 的结构示意图。



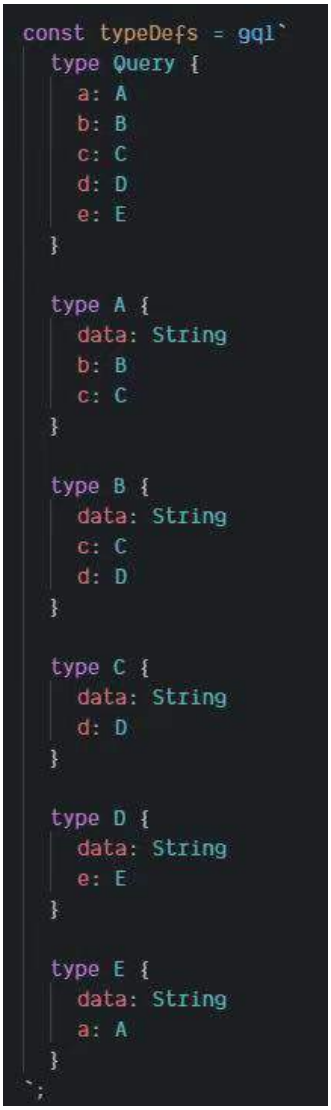
而 Graph 里的节点之间，可能存在不只一种连接路径，可能存在循环，可能存在递归引用，可能没有 Root 节点。它们组成了一个网络结构。

我们可以把 Graph 这种网络结构，通过裁剪连接路径，把它压缩成任意节点只有唯一连接路径的简化形态。如此网络结构退化成层次结构，它变成了 Tree。

也就是说，Graph 是比 Tree 更复杂的数据结构，后者是它的简化形式。拥有 Graph，我们可以按照不同的裁剪方式，衍生出不同的 Tree。而 Tree 里包含的信息，如果不增加其它额外数据，不足以构建足够复杂的 Graph 结构。

6.2 GraphQL 里的 Graph 结构

在 GraphQL 里，承担构建网络结构的，并非 GraphQL 查询语句，而是基于 GraphQL Type System 构建的 Schema。



```
const typeDefs = gql`
  type Query {
    a: A
    b: B
    c: C
    d: D
    e: E
  }

  type A {
    data: String
    b: B
    c: C
  }

  type B {
    data: String
    c: C
    d: D
  }

  type C {
    data: String
    d: D
  }

  type D {
    data: String
    e: E
  }

  type E {
    data: String
    a: A
  }
`;
```

上图是一个 GraphQL Schema，定义了 A, B, C, D 和 E 五种数据类型，它们分别挂载到 入口类型 Query 里的 a, b, c, d 和 e 字段里。

A, B, C, D, E 里面，包含着递归的结构。A 里面包含 B 和 C，B 里面包含 C 和 D，D 里面包含 E，E 里面又包含 A，又回到了 A。

这是一个复杂的关系网络。要构建递归关联，并不需要这么复杂。直接 A 里包含 B，和 B 里包含 A 也行，此处是一个演示。

有了这个基于数据类型的 Graph 关系网络，我们可以实现从 Graph 中派生出 JSON Tree 的能力。

上图是一个 GraphQL 的查询语句，它是一个包含很多 key 的层次结构，亦即一个 Tree。

它从根节点里取 a 字段，然后向下分层，找到了 e。而 e 节点里也包含一个跟根节点同类型的 a 字段，因此它可以继续向下分层，重来一遍，又到了 e 节点，此时它只取了 data 字段，查询中止。

我编写了一个简单的 Resolver 函数，用来演示查询结果。

```
const resolvers = {
  Query: {
    a() {
      return { data: "a" };
    },
    b() {
      return { data: "b" };
    },
    c() {
      return { data: "c" };
    },
    d() {
      return { data: "d" };
    },
    e() {
      return { data: "e" };
    }
  },
  A: {
    b(parent) {
      let data = parent.data + " -> b";
      return { data };
    },
    c(parent) {
      let data = parent.data + " -> c";
      return { data };
    }
  },
```

它很简单。Query 里返回跟字段名一样的字母，任何子节点的数据，都是拼接父节点的字母串。如此我们可以从查询结果看出数据流动的层次。

查询结果如下：


```
{
  "data": {
    "a": {
      "b": {
        "c": {
          "d": {
            "e": {
              "data": "a -> b -> c -> d -> e",
              "a": {
                "b": {
                  "c": {
                    "d": {
                      "e": {
                        "data": "a -> b -> c -> d -> e -> a -> b -> c -> d -> e"
                      }
                    }
                  }
                }
              }
            }
          }
        }
      }
    },
```


第一个 e 节点的 data 字段里，拿到了父节点里的 data 数据，其父节点的 data 数据又是通过它的父节点里获取的，因此有一个数据链条。

而第二个 e 节点同理，它有两段链条。

只要不编写后续字段，我们可以停留在任意节点的 data 字段里。

也就是说，我们用作为 Tree 的 Query 语句，去裁剪了作为 Graph 的 Schema 数据关联网络，得到了我们想要的 JSON 结构。

通过这个角度，我们可以理解为什么 GraphQL 不允许 Query 语句停留在 Object 类型，一定要明确的写出对象内部的字段，直到所有 Leaf Node 都是 Scalar 类型。

这不仅仅是一个所谓的最佳实践，这也是 Graph 本身的特征。对象节点里，可能通过循环或者递归关系，拓展出无限大的数据结构。Query 语句必须写清楚，才能帮助 GraphQL 去裁剪掉不必要的数据关联路径。

6.3 Graph 网络结构的实际价值

前面的 A, B, C, D, E 案例，并不能直观的让大家感受到，Graph 网络结构的实际价值。它看起来像一个连线游戏。

放到 Facebook 的社交网络场景下，其必要性和价值就凸显了。

假设我们要一次性获取用户的好友的好友的好友的好友的好友，基于 RESTful API 我们有什么特别好的方法吗？很难说。

而 Graph 这种递归关联的结构，实现这种查询轻而易举。

```
type User {
  id: ID
  name: String
  friends: [User]
}

type Query {
  user: User
}

query getFriends {
  user {
    id
    nam
    friends {
      id
      name
      friends {
        id
        name
        friends {
          id
          name
          friends {
            id
            name
            friends {
              id
              name

            }
          }
        }
      }
    }
  }
}
```

我们定义了一个 User 类型，挂到 Query 入口上的 user 字段里。User 类型的 friends 字段又是一个 User 类型的列表。这样就构建了一个递归关联。

getFriends 查询语句，可以不断地从任意用户开始，关联其 friends，得到 friends 数组结果。任意一个 friend 也是 User，它也有自己的 friends。查询语句在最外层的 friends 停了下来，它只查询了 id 和 name 字段。

看到这里，另一个经典的关于 GraphQL 的误解出现了：只有像 Facebook，Twitter 这类社交关系网络，才适合 GraphQL，而我们的场景下，GraphQL 并不适用。

其实不然，社交关系网络里使用 GraphQL 特别有效，不意味着其它场景下，GraphQL 不能带来收益。

设想一个电商平台的场景，它有用户、产品和订单这组铁三角，其它库存、价格，优惠券，收藏等先不提。在最简单的场景下，GraphQL 依然可以发挥作用。

```
type User {
  id: ID
  name: String
  orders: [Order]
}

type Product {
  id: ID
  title: String
  description: String
  price: Float
  orders: [Order]
}

type Order {
  id: ID
  created: Date
  price: Float
  user: User
  product: Product
}

type Query {
  user(id: ID): User
  product(id: ID): Product
  order(id: ID): Order
}
```

我们构建了 User，Product 和 Order 三个类型，它们彼此之间有字段上的递归关联关系，是一个 Graph 结构。在 Query 入口类型上，分别有 user, product 和 order 三个字段。

据此，我们可以实现从 user, product 和 order 任意维度出发，通过它们的关联关系，实现丰富而灵活的查询。

比如，查看用户的所有订单及其跟订单相关的产品，Query 语句如下：

```
query getUserOrders {
  user(id: 123) {
    id
    name
    orders {
      id
      created
      price
      product {
        id
        title
        description
        price
      }
    }
  }
}
```

我们查询了 id 为 123 的用户，他的名字和订单列表，对于每个订单，我们获取该订单的创建时间，购买价格和关联产品，对于订单关联的产品，我们获取了产品 id，产品标题，产品描述和产品价格。

当我们的后端人员组织架构是按照领域模型来划分时，用户，产品和订单，通常是 3 个团队，他们各自提供领域相关的接口。通过 GraphQL 我们可以很容易将它们整合到一起。

再比如，查看一个产品下的所有订单及其关联用户，Query 语句如下：

```
query getProductOrders {
  product(id: 123) {
    id
    title
    description
    price
    orders {
      id
      created
      price
      user {
        id
        name
      }
    }
  }
}
```

我们查询了 id 为 123 的产品，它的产品标题，产品描述和价格，以及关联的订单。对于每个关联订单，我们查询了订单的创建时间，购买价格以及下订单的用户，对于下订单的用户，我们查询了他的用户 id 和名称。

如你所见，只要构建出了 Graph 结构的数据网络，它不像 Tree 那样有唯一的 Root 节点。从任意入口出发，它都可以通过关联路径，不断的衍生出数据，得到 JSON 结果。

我们不必疲于编写面向产品详情页的接口，面向订单详情页的接口，面向用户信息的接口。我们编写了一个数据关系网络，就足以适配不同的场景。

此处演示的，只是用户，产品和订单这三个资源的关系网络，已经可以看出 GraphQL 的适用性。在实际场景中，我们能搭建出更复杂的数据网络，它具备更强大的数据表达能力，可以给我们的业务带来更多收益。

七、我们的 GraphQL-BFF 实践模式

在掌握上述关于 GraphQL 的纲领知识后，我们来看一下在实践中 ，GraphQL-BFF 的一种实际做法。

首先是技术选型，我们主要采用了如下技术栈。

开发语言选用了 TypeScript，跑在 Node.js v10.x 版本上，服务端框架是 Koa v2.x 版本，使用 apollo-server-koa 模块去运行 GraphQL 服务。

Apollo-GraphQL 是 Node.js 社区里，比较知名和成熟的 GraphQL 框架。做了很多的细节工作，也有一些相对前沿的探索，比如 Apollo Federation 架构等。

不过，有两点值得一提：

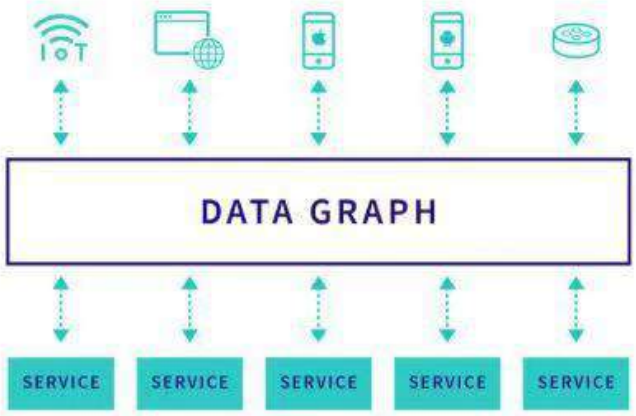
1) Apollo-GraphQL 属于 GraphQL 社区的一部分，而非 Facebook 官方的 GraphQL 开发团队。Apollo-GraphQL 在官方 GraphQL 的基础上进行了带有他们自身理念特点的封装和设计。像 Apollo Federation 这类目前看来比较激进的方案，即使是 GraphQL 官方的开发人员，对此也持保留态度。

2）Apollo-GraphQL 的重心是前文所说的第一类 API Gateway 角色的 GraphQL 服务，本文探讨的是第二类。因此，Apollo-GraphQL 里有很多功能对我们来说没必要，有一些功能的使用方式，跟我们的场景也不契合。

我们主要使用的是 Apollo-GraphQL 的 graphql-tools 和 apollo-server-koa 两个模块，并在此基础上，进行了符合我们场景的设计和改编。

7.1 我们的 GraphQL-BFF 架构设计

GraphQL-BFF 的核心思路是，将多个 services 整合成一个中心化 data graph。



每个 service 的数据结构契约，都放入了一个大而全的 GraphQL Schema 里；如果不做任何模块化和解耦，开发体验将会非常糟糕。每个团队成员，都去修改同一份 Schema 文件。

这明显是不合理的。GraphQL-BFF 的开发模式，应该跟 service 的领域模型，有一一对应的关系。然后通过某种形式，多个 services 自然整合到一起。

因此，我们设计了 GraphQL-Service 的概念。

7.1.1 GraphQL-Service

GraphQL-Service 是一个由 Schema + Resolver 两部分组成的 JS 模块，它对应基于领域模型的后端的某个 Servcie。每个 GraphQL-Service 应该可以按照模块化的方式编写，跟其它 GraphQL-Service 组合起来后，构建出更大的 GraphQL-Server。

GraphQL-Service 通过 GraphQL 的 Type Extensions 构建数据关联关系。

```
const UserService = gql`
  type User {
    id: Int
    name: String
  }

  extend type Order {
    user: User
  }

  extend type Product {
    users: [User]
  }

  extend type Query {
    user(id: Int): User
    users(ids: [Int]!): [User]
  }
`
```

如上所示，我们的 UserService 里面，只涉及到了 User 相关的类型处理。它定义了自己的基本字段，id 和 name。通过 extend type 定义了它在 Order 和 Product 数据里的关联字段，以及定义在 Query 里的入口字段。

从 User Schema 里我们可以看到，User 有两类查询路径。

1）通过根节点 Query 以传递参数的方式，获取到 User 信息。
2）通过 Product 或 Order 节点，以数据关联的方式，获取到 User 信息。

```
const OrderService = gql`
  type Order {
    id: Int
  }

  extend type User {
    orders: [Order]
  }

  extend type Product {
    orders: [Order]
  }

  extend type Query {
    order(id: Int!): Order
    orders(ids: [Int]!): [Order]
  }
`
```

上图是 OrderService 的 Schema，它也只涉及了 Order 相关的类型逻辑。同样是通过 extend type 定义了在 User 和 Product 里的关联字段，以及定义了在根节点 Query 里的入口字段。

Order 数据跟 User 一样，有两种消费路径。一种通过 Query 节点，另一种是通过数据关联节点。

前面我们演示 User, Order 和 Product 铁三角关系时，是在同一个 Schema 里编写它们的关联。我们把多个 GraphQL-Service 的 Schema 整合到一起后，可以生成同样的结果：

```
type User {
  id: Int
  name: String
  orders: [Order]
  collections: [Product]
}

type Order {
  id: Int
  user: User
  product: Product
}

type Product {
  id: Int
  title: String
  description: String
  price: Float
  orders: [Order]
  users: [User]
}

type Query {
  order(id: Int!): Order
  orders(ids: [Int]!): [Order]
  user(id: Int): User
  users(ids: [Int]!): [User]
  product(id: Int): Product
  products(ids: [Int]): [Product]
}
```

上图不是我们手动编写的，而是 merge 多个 GraphQL-Service 的 Schema 后生成的结果。可以看出来，跟之前手写的版本，总体上是一样的。

有了解耦的 Schema 并不足够，它只定义了数据类型及其关联。我们需要 Resolver 去定义数据的具体获取方式，Resolver 也需要解耦。

7.1.2 GraphQL-Resolver

不管是在官方的 GraphQL 文档里，还是 Apollo-GraphQL 的文档里，Resolver 都是以普通函数的形态出现。

```
const resolverMap = {
  Query: {
    author(obj, args, context, info) {
      return find(authors, { id: args.id });
    },
  },
  Author: {
    posts(author) {
      return filter(posts, { authorId: author.id });
    },
  },
};
```

这在简单场景下，没有什么问题。正如在简单场景下，用 node.js 的 http.createServer 就可以创建一个 server。

```
const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});
```

如上，设置状态码，设置响应的 Content-Type，返回内容即可。

然而，在更复杂的真实项目中，我们实际上需要 express、koa 等服务端框架，用中间件的模式编写我们的服务端处理逻辑，由框架将它们整合为一个 requestListener 函数，注册到 http.createServer(requestListener) 里。

在 GraphQL Server 里，虽然 endpoint 只有 /graphql 一个，但不代表它只需要一组 Koa 中间件。

正如一开始我们指出的，每个超级接口里都包含一半功能的 GraphQL 实现。GraphQL 是往超级接口的方向更进一步，不能简单地以普通接口的眼光去看待它。

在 Query 下的每个字段，都可能对应 1 到多个内部服务的 API 的调用和处理。只用普通函数构成的 resolverMap，不足以充分表达其中的逻辑复杂度。

不管是用 endpoint 来表示资源，还是用 GraphQL Field 字段来表示资源，它们只是外在形式略有不同，不会改变业务逻辑的复杂度。

因此，采用比普通函数具有更好的表达能力的中间件，组合出一个个 Resolver，再整合到一个 ResolverMap 里。可以更好的解决之前解决不了，或者很难的问题。

所谓的架构能力，体现在理解我们面对的问题的复杂度及其本质特征，并能选择和设计出合适的程序表达模型。

后面我们将演示，正确的架构，如何轻易地克服之前难以解决的问题。

7.1.3 用 koa-compose 组织我们的 Resolver

或许很多同学并不清楚，express 或 koa 里的中间件模式，可以脱离作为服务端框架的它们而单独使用。正如 GraphQL 可以单独不作为 server，在任意支持 JavaScript 运行的地方使用一样。

我们将使用 koa-compose 这个 npm 模块，去构造我们的 Resolver。

前文里提到的 gql 函数，接受一个 Schema 返回一个 GraphQL-Service，每个 GraphQL-Service 都有一个 resolve 方法：

```
GraphQLService.resolve: (typeName: string, fieldHandlers: FieldHandlers) => GraphQLService
```

resolve 方法，接受两个参数。第一个是 typeName，对应 GraphQL-Schema 里的 Object Type 的类型名称；第二个是 fieldHandlers，每个 handler 支持中间件模式，最终它们将被 koa-compose 整合成一个 Resolver。

以 UserService 为例，其 Resolver 写起来如下：

```
UserService.resolve('Query', {
  user(ctx) {
    let id = ctx.args.id || 1
    ctx.result = find(userTable, { id })
  },
  users(ctx) {
    let predicate: UserPredicate = user => ctx.args.ids.includes(user.id)
    ctx.result = filter(userTable, predicate)
  }
})

UserService.resolve('Order', {
  user(ctx) {
    let id = ctx.parent.userId
    ctx.result = find(userTable, { id })
  }
})

UserService.resolve('Product', {
  users(ctx) {
    let product = ctx.parent
    ctx.result = filter(userTable, user =>
      product.collectedByUsers.includes(user.id)
    )
  }
})
```

作为普通函数的 Resolver 接收的所有参数，都被整合到了 ctx 里面。ctx.result 则是该字段的最终输出，类似于 koa server 里的 ctx.body。我们刻意采用了 ctx.result 这个不同于 ctx.body 的属性，明确区分我们处理的是一个接口还是一个字段。

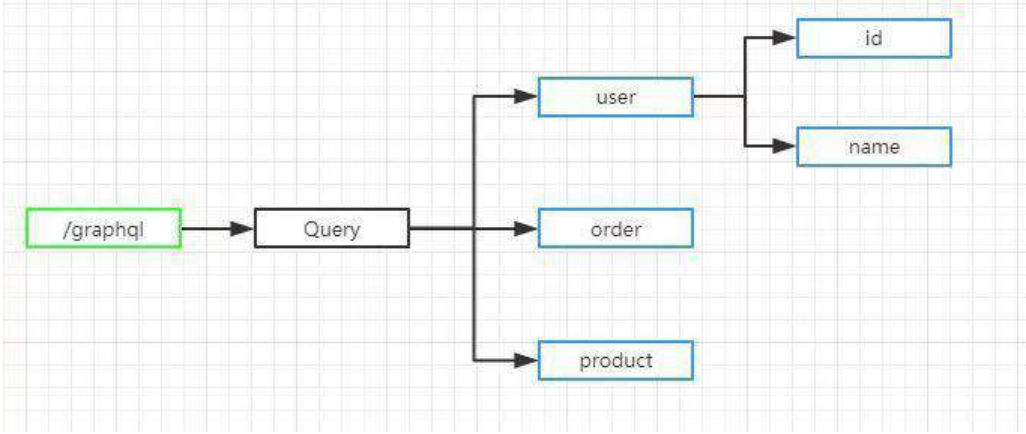
在简单场景下，中间件模式的 Resolver 跟普通函数的 Resolver，仅仅是参数的数量和返回值的方式不同。并不会增加大量的代码复杂度。

```
const logger = async (ctx, next) => {
  let { fieldName } = ctx.info
  let start = Date.now()

  await next()

  let duration = Date.now() - start

  console.log(fieldName, `${duration.toFixed(2)}ms`)
}

UserService.resolve('Query', {
  user: [logger, ctx => {
    let id = ctx.args.id || 1
    ctx.result = find(userTable, { id })
  }],
  users: [logger, ctx => {
    let predicate: UserPredicate = user => ctx.args.ids.includes(user.id)
    ctx.result = filter(userTable, predicate)
  }]
})
```

当我们多个字段要复用相同的逻辑时，编写成中间件，然后将 handler 变成数组形式即可。（在代码里我们用 json 模拟了数据库表，所以是同步代码，实际项目里，它可以是异步的调用接口或者查询数据库）。

上面的 logger，只是一个简单案例。除此之外，我们可以编写 requireLogin 中间件，决定一个字段是否只对登陆用户可用。我们可以编写不同的工具型中间件，注入 ctx.fetch, ctx.post, ctx.xxx 等方法，以供后续中间件使用。

每个 GraphQL Field 字段，都拥有独立的一组中间件和 ctx 对象，跟其他字段互相不影响。我们同时，可以把所有字段共享的中间件，放到 koa server 里的中间件里。



如上图所示，绿框是 endpoint，可以编写 koa server 层面的 middleware。而蓝框是 GraphQL Field 字段，可以编写 Resolver 层面的 middleware。endpoint 层面的 middleware 对 ctx 的修改，会影响到后面所有字段。

```
import Router from 'koa-router'
import fetch from 'node-fetch'

const router = new Router()

router.post('/graphql', logger, async (ctx, next) => {
  ctx.fetch = fetch
  await next()
  console.log('graphql query result', ctx.body)
})

export default router
```

也就是说，我们可以像上面那样。挂接口层面的 logger，可以知道整个 graphql 查询的耗时。编写一个中间件，在 next 之前，挂载一些方法，供后续中间件使用；在 next 之后，拿到 graphql 的查询结果，进行额外的处理。

7.2 解决 mock 难题

GraphQL 是天生 mock 友好的模式，因为其 Schema 里已经指明了所有数据的类型及其关联；很容易可以通过 faker data 之类的手段，自动根据类型生成假数据。

然而，在实践中，实现 GraphQL Mocking 还是有不少的挑战。

```
const { ApolloServer, gql } = require('apollo-server');

const typeDefs = gql`
  type Query {
    hello: String
  }
`;

const server = new ApolloServer({
  typeDefs,
  mocks: true,
});

server.listen().then(({ url }) => {
  console.log(`🚀 Server ready at ${url}`)
});
```

如上图所示，在 Apollo GraphQL 里，mock 看似很简单，只需要在创建服务时，设置 mock 为 true，或者提供一个 mock resolver 即可。但是，一个全局的，跟着服务创建走的 mock，太过粗暴。

mock 的价值在于提供更好的数据灵活性以加速开发效率。它既可以在没有数据时，提供假数据；也可以在真数据的接口有问题时，不用重启服务，也能降级为假数据。它既可以是整个 GraphQL 查询级别的 mock，也可以是字段级别的 mock。

作为超级接口的 GraphQL 服务，全局的，在启动阶段就固化的 mocking，意义不大。

Apollo GraphQL 的 mocking 实践问题，正是它采用普通函数来描述 Resolver 所带来的；它很难简单的通过拓展某个 resolver 而支持 mocking。它不得不在创建服务时，额外新增一个 mock resolver map 去承担 mocking 职能。

而我们的 composed resolver 处理动态 mocking 却异常简单。

```
query {
  user @mock {
    id
    collections {
      id
      title
      description
      price
    }
  }
}
```

它不仅可以在运行时动态确定，它不仅可以细化到字段级别，它甚至可以跟着某次查询走 mock 逻辑（通过添加 @mock 指令）。

```
{
  "data": {
    "user": {
      "id": 91699,
      "collections": [
        {
          "id": 83096,
          "title": "Web open architecture quantifying
transmitter",
          "description": "Computers Unbranded Licensed
quantify",
          "price": 47491.8
        },
        {
          "id": 9334,
```


上图是默认情况下，基于 faker 这个 npm 包，根据数据类型生成的 mock data。

```
if (returnType === GraphQLInt) {
  return faker.random.number()
}

if (returnType === GraphQLFloat) {
  return faker.random.number({ precision: 0.01 })
}

if (returnType === GraphQLString) {
  return faker.random.words(4)
}

if (returnType === GraphQLBoolean) {
  return faker.random.boolean()
}

if (returnType === GraphQLID) {
  return faker.random.uuid()
}
```

在我们的设计里，默认的 mocking，其内部实现方式很简单。我们先是编写了上图，根据 GraphQL Type 调用 faker 模块对应的方法，生成假数据。

```
const createResolver = (middlewares: Handler | Handler[]) => {
  if (!Array.isArray(middlewares)) middlewares = [middlewares]

  if (!hasMockHandler(middlewares)) {
    middlewares = [mocker, ...middlewares]
  }

  let f = compose(middlewares)
  let resolver: IFieldResolver<any, BaseContext> = async (
    parent,
    args,
    context,
    info
  ) => {
    let ctx: BaseContext = proxyContext({
      parent,
      args,
      info,
      result: null,
      context
    })
    await f(ctx)
    return ctx.result
  }

  return resolver
}
```

然后在 createResolver 这个将中间件整合成 resolver 的函数里，先判断中间件里是否存在自定义的 mock handler 函数，如果没有，就追加前面编写的 mocker 处理函数。

我们还提供了 mock 中间件，让开发者能指定 mock 数据来源，比如指定 mock json 文件。

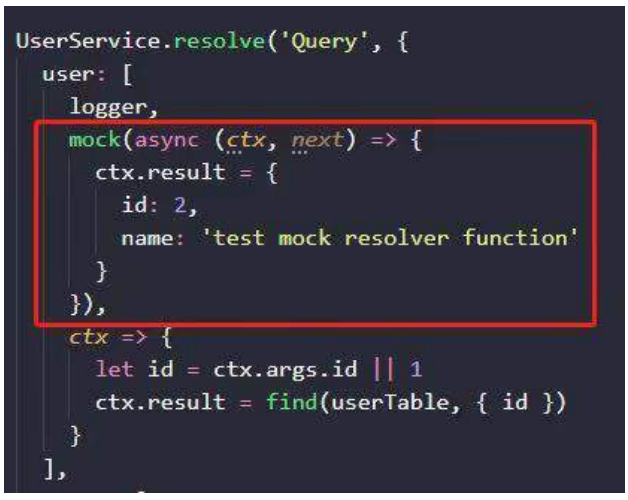
```
UserService.resolve('Query', {
  user: [
    logger,
    mock('user.json'),
    ctx => {
      let id = ctx.args.id || 1
      ctx.result = find(userTable, { id })
    }
  ],
```

mock 中间件，接收字符串参数时，它会搜寻本地的 mock 目录下是否有同名文件，作为当前字段的返回值。它也接收函数作为参数，在该函数里，我们可以手动编写更复杂的 mock 数据逻辑。

有趣的地方是，mock/user.json 文件里，只包含上图红框的数据，其关联出来的 collections 字段，是真实的。这是合理的做法，mock 应该跟着 resolver 走。关联字段拥有自己的 resolver，可能调用自己的接口；不应该因为父节点是 mock 的，子节点也进入 mock 模式。

如此，我们可以在父节点 resolver 对应的后端接口挂掉后，mock 它，让没挂掉的子节点 resolver 正常运行。如果我们希望子节点 resolver 也进入 mock。很简单，添加一个 @mock 指令即可。

如上所示，user 字段和 collections 字段的 resolver 都进入了 mock 模式。



自定义 mock resolver 函数的方式如上图所示，mock 中间件保证了，只有在该字段进入

mock 模式时，才执行 mock resolver function。并且，mock resolver function 内部依然有机会通过调用 next 函数，触发后面的真实数据获取逻辑。

以上所有这些灵活性，都来自于我们选用了表达能力和可组合性更好的中间件模式，代替普通该函数，承担 resolver 的职能。

总结

至此，我们得到了一个简单而灵活的实践模式。我们用 Schema 去构建 Data Graph 数据关联图，我们用 Middleware 去构建 Resolver Map，它们都具备很强的表达能力。

在开发 GraphQL-BFF 时，我们的 GraphQL-Service 跟后端基于领域模型的 Service，具有总体上的一一对应关系。不会产生后端数据层解耦后，在 GraphQL 层重新耦合的尴尬现象。

关于 GraphQL 还有很多话题可以讨论，比如 batching，caching 等。这部分内容在网络上很多 GraphQL 的文档和教程里都可以找到，这里我们不再赘述。

总的而言，根据我们对 GraphQL 的考察和实践，我们认为它可以比 RESTful API 更好的解决我们面对的问题。

我们对 GraphQL 的期望，不仅仅停留在 BFF 层。我们希望通过积累在 BFF 层使用 GraphQL 的成功经验，帮助我们摸索出在 Micro Frontend 架构上使用 GraphQL 模式的合理设计。

如前面所演示的，像 User，Product 和 Order 这种公共级别的数据类型，不可能只由一个团队去维护，它们需要被其它团队所拓展。使得我们可以通过用户，找到它关联的订单，收藏，优惠券等由其它团队维护的数据。

放到 Micro Frontend 架构上，一个支付按钮，也夹杂了多种类型的数据，产品信息，用户信息，库存信息，UI 展示信息，交互状态信息等等，综合了这些信息，支付按钮被点击时，才得到了充分的数据，可以决定是否去支付。

朴素 Micro Frontend 的设计，用 Vue, React, Angular 不同框架，分别维护不同组件，通过

router/message-passing 等方式互相通讯。在我看来，这是对后端微服务架构的拙劣模仿。

后端服务，各自部署在独立环境中，对体积不敏感；因而可以采用不同的语言和技术栈。这不意味着将它简单的放到前端里一样成立。无法共享前端开发的基础设施，这不是微前端，这是一种人员组织架构上的混乱。

GraphQL 让我们看到，基于领域模型的微前端架构，可能是更好的方向。一个简单的支付按钮，也综合了多个领域模型，由多个开发者有组织的协同开发。并不因为它表面上看起来是一个 Button 组件，就由某个团队单独维护。

当然，探索 GraphQL 的其它方向的前提是，GraphQL-BFF 架构得到成功的验证。就现阶段的实践成果来看，我们对此充满了信心。

尽管我们的代码暂无开源计划，不过相信这篇文章，足够完整和清楚地介绍了我们的 GraphQL-BFF 方案。希望它能给大家带来一点帮助。

加载速度提升 15%，携程对 RN 新一代 JS 引擎 Hermes 的调研

【作者简介】储贻锋，携程无线平台研发部基础框架组资深 Android 研发，目前主要负责 CRN Android 端和携程 Android 基础架构的维护与开发工作。

引言

Facebook 在 ChainReact2019 大会上正式推出了新一代 JavaScript 执行引擎 Hermes。Hermes 是个轻量级的 JS 引擎，专门对 Android 上运行 ReactNative 进行了优化。我们第一时间在 CRN 项目中集成了 Hermes，并做了深度调研。

一、Hermes 介绍

自 ReactNative 推出以来，有大量的 APP 接入并使用，其中也包括大型应用的主流程业务。随着业务复杂度不断上升，性能问题变得无法忽视。

在分析性能数据时，Facebook 团队发现 JavaScript 引擎是影响启动性能和应用包体积的重要因素。由于 JavaScriptCore 最初是为桌面浏览器端设计，相较于桌面端，移动端能力有太多的限制，为了能从底层对移动端进行性能优化，Facebook 团队选择自建 JavaScrip 引擎，设计了 Hermes，限于 iOS AppStore 审核限制，目前仅用于 Android 平台。

Chain React 大会上官方给出了 Hermes 引擎一组数据：

- 从页面启动到用户可操作的时间长短（Time To Interact：TTI），从 4.3s 减少到 2.01s
- App 的下载大小，从 41MB 减少到 22MB
- 内存占用，从 185MB 减少到 136MB

CRN 先前做过框架代码拆分和预加载、业务代码懒加载、业务代码预加载等性能优化方案，正困惑于如何更近一步进行性能优化。当看到 Hermes 这三个关键指标都有了显著的提高，非常激动，觉得 Hermes 是非常好的一个方向，接下来我们就来了解 Hermes 的使用和实测性能数据。

二、快速上手 Hermes

Faceback 团队已经将 Hermes 工具上传到了 npm：hermesvm。hemres 工具可以直接运行 JS 代码、转换字节码并且提供非常多的参数进行调优控制。

这里介绍一下 hermesvm 执行 JS 代码和转换 bytecode 功能。

```
// 创建 hermes_test 文件，内容：print("This is Hermes Demo");
vim hermes_test.js
```

```
// 直接执行纯文本 js
~/node_modules/hermesvm/osx-bin/hermes hermes_test.js
This is Hermes Demo

// 转换成 bytecode
~/node_modules/hermesvm/osx-bin/hermes --emit-binary hermes_test.js -out hermes_test.hbc

// 执行字节码
~/node_modules/hermesvm/osx-bin/hermes hermes_test.hbc
This is Hermes Demo
```

三、Hermes 是如何优化的?

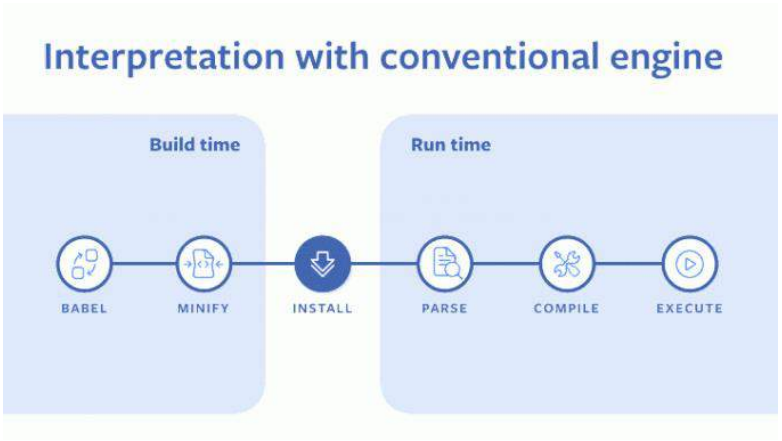
主流 JavaScript 引擎，例如 JSC、V8、SpiderMonkey 等几乎都是为了桌面端浏览器服务的，Hermes 针对移动终端设备的特点做了一些优化，其中最重要的我们认为是以下两点：

3.1 字节码预编译

现代主流的 JavaScript 引擎在执行一段 js 代码的大概流程是：

- 先读取源码文件
- 解析源代码并转换成字节码（bytecode）
- 最后执行

在运行时解析源码转换字节码是一种时间浪费，所以 Hermes 选择预编译的方式在编译期间生成字节码。这样做一方面避免了不必要的转换时间，另一方面多出的时间可以用来优化字节码，从而提高执行效率。



3.2 放弃 JIT

为了加快执行效率，现在主流的 JavaScript 引擎都会使用一个 JIT 编译器在运行时通过转换成机器码的方式优化 JS 代码。Faceback 团队认为 JIT 编译器有主要两个问题：

- 要在启动时候预热，对启动时间有影响；
- 会增加引擎 size 大小和运行时内存消耗；

基于这俩点对性能指标的影响，Faceback 团队决定不实现 JIT 编译器。

这里所谓放弃 JIT，有两点需要再解释一下：

- 纯文本 JS 代码执行效率降低。放弃 JIT，是指放弃运行时 Hermes 引擎对纯文本 JS 代码的编译优化。我们的验证数据也表面，纯文本的 JS 代码执行，Hermes 引擎明显比 JavaScriptCore 慢。
- 对 RN 代码的动态性无影响。由于 Hermes 仍然可以执行纯文本的 JS 代码，并且可以支持动态读取 bytecode， 因此对 RN 的动态性并无影响。

四、如何集成 Hermes?

4.1 从新创建工程集成

1. 升级最新 react-native-cli
npm install -g react-native-cli

2.初始化最新 react-native 工程，最新版为 0.60.3
react-native init HermesDemo

3. 开启 hermes，编辑 HermesDemo 工程 android/app/build.gradl 文件
```
 project.ext.react = [
         entryFile: "index.js",
-        enableHermes: false    // clean and rebuild if changing
+         enableHermes: true    // clean and rebuild if changing
   ]
```

4. 使用 Relase 包体验 Hermes 带来的速度提升
react-native run-android --variant release

4.2 从源码集成

git clone https://github.com/facebook/react-native.git // 需要切换到 Hermes release 节点，比如：eec4dc6
cd react-native
npm install
./gradlew :RNTester:android:app:installHermesRelease // 使用生产环境 hermes

4.3 Hermes 集成过程分析

分析 react-native react.gradle 源码可以看到，如果打开了 Hermes 开关，会在原先打包 RN 代码的 bundleXXXJsAndAsset task 后面追加执行一段 Hermes 转换命令: hermes --emit-binary -out xxx。

```
...
// 1. 执行标准 RN 打包
commandLine(*nodeExecutableAndArgs, cliPath, bundleCommand, "--platform", "android", "--dev", "${devEnabled}",
                        "--reset-cache", "--entry-file", entryFile, "--bundle-output", jsBundleFile, "--assets-dest", resourcesDir,
                        "--sourcemap-output", jsPackagerSourceMapFile, *extraArgs)
...
...
// 2. 将打包好的 jsbundle 文件转换成字节码
if (enableHermes) {
    commandLine(getHermesCommand(), "-emit-binary", "-out", jsBundleFile, jsBundleFile, *hermesFlags)
}
...
```

4.4 执行过程分析

为了进一步抽象 JavaScript 执行层，RN 底层创建了 JSExecutor 和 Runtime 接口，并把大部分业务逻辑放到了实现了 JSExecutor 的 JSIExcutor.cpp 中。对于 JavaScript 执行引擎来说只需要实现 Runtime 接口即可对接 RN 框架。

JavaScriptCore 的 Runtime 实现类是 JSCRuntime。相应的，此次 Hermes 升级，底层创建了 HermesRuntime。

```
// JSCRuntime.cpp jsc Runtime
class JSCRuntime : public jsi::Runtime

// hermes.h hermes Runtime
class HermesRuntime : public jsi::Runtime...
```

每一种 JSExecutor 都提供了创建类 XXXExecutorFactory 来创建相应实例，并且提供了相应的 Java 对象。

RN 框架在初始化 ReactInstanceManager 的时候需要传入 JavaScriptExecutorFactory。如果要切换 JavaScript 执行引擎只需要在 ReactInstanceManager 创建的时候做控制即可。

官方的控制流程是，优先加载 jscexecutorso,如果成功则使用 JSCRuntime，否则使用

HermesRuntime。

```
private JavaScriptExecutorFactory getDefaultJSExecutorFactory(String appName, String deviceName) {
    try {
      // If JSC is included, use it as normal
      SoLoader.loadLibrary("jscexecutor");
      return new JSCExecutorFactory(appName, deviceName);
    } catch(UnsatisfiedLinkError jscE) {
      // Otherwise use Hermes
      return new HermesExecutorFactory();
    }
  }
```

由此可见无论是对于 RN JS 代码的打包还是 Native 代码逻辑的更改，升级 Hermes 的成本都非常低。

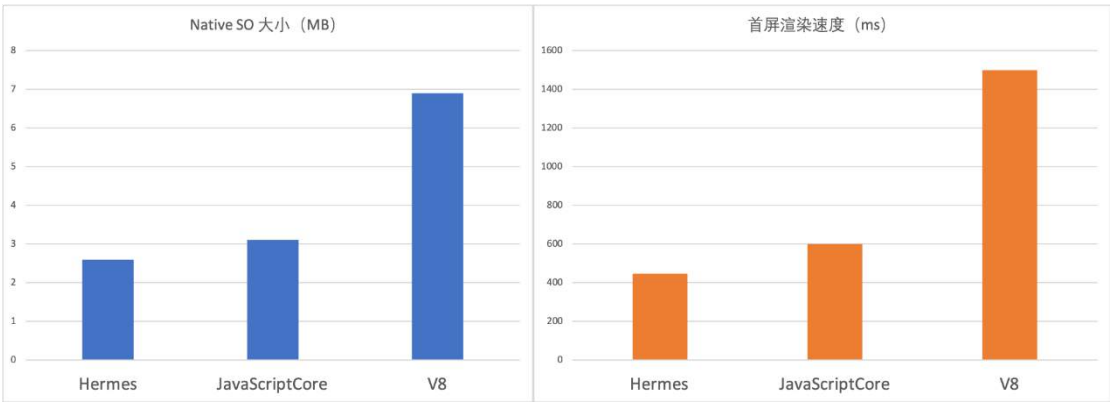
五、Hermes，JavaScriptCore，V8 的对比

通过上面的 Hermes 集成分析可知，Hermes 对整个 RN 原有架构的侵入是极少的，甚至做到了可插拔式接入。我们很快将 Hermes 集成到携程 CRN 框架，并和原先的 JavaScriptCore 引擎以及社区提供的 V8 引擎做了比较。

经过我们的数据验证，Faceback 团队提出的关键性指标相较于原先的 JSC 都有了显著提高。

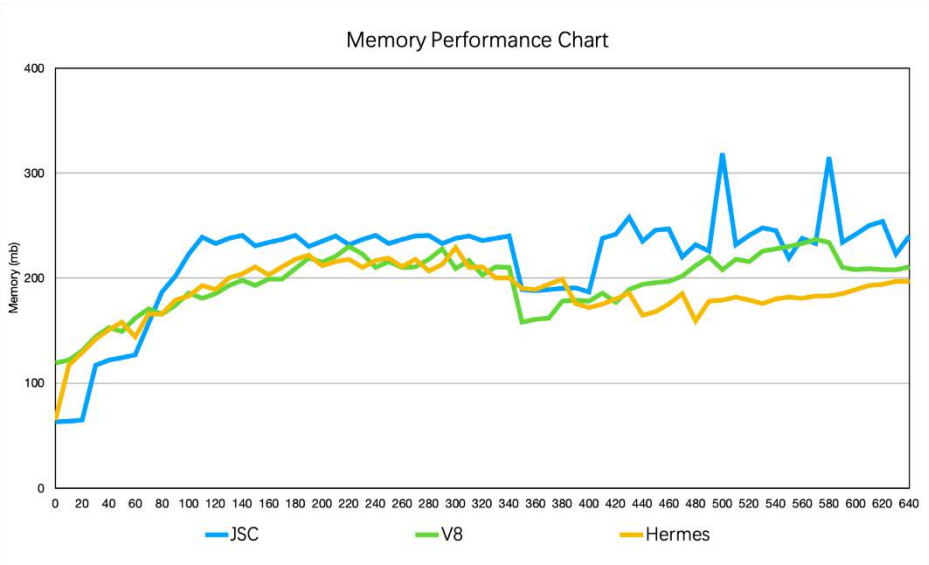
- 首屏渲染速度：bytecode 代码执行情况下，Hermes 比 JavaScriptCore 要快。在携程 App 中，拿门票业务做了验证，在做了预加载的情况下，首屏加载速度依然可以提升约 15%。而 V8 的表现就非常糟糕了。

- Native so size：RN 所依赖的必要 so 库，Hermes 比 JavaScriptCore 减少了约 16%（单 armeabi 架构压缩后降低了 0.5M 左右），V8 则要远大于 Hermes 和 JavaScriptCore。

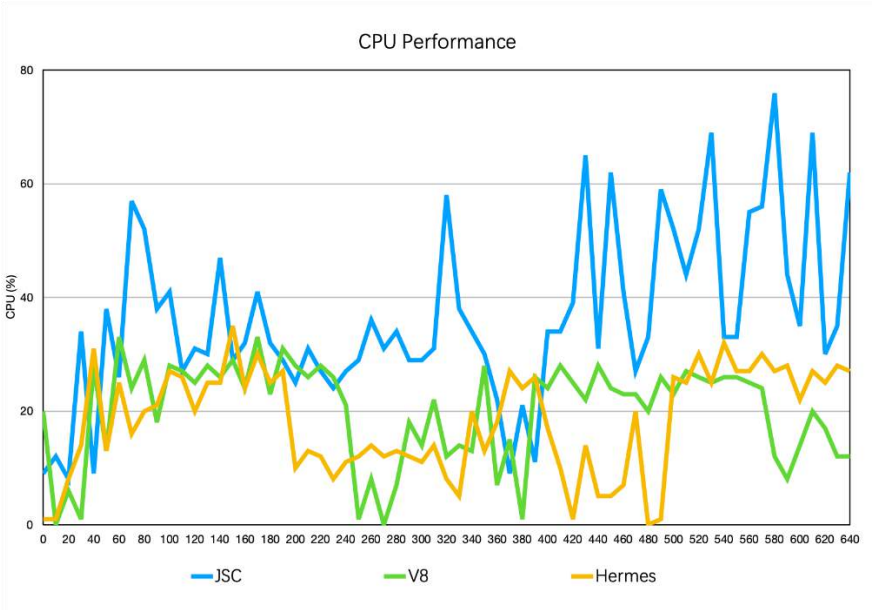


- 内存：拿 RNTester 工程测试进入 RN 页面滑动进入若干页面并退出之后，内存的波动

情况比较可以看到，V8 和 Hermes 内存增长要更加平滑。



- CPU：拿 RNTester 工程测试进入 RN 页面滑动进入若干页面并退出之后，对比 CPU 波动情况。Hermes 明显好于 V8 和 JavaScriptCore。



六、Hermes 引擎的动态性

另外通过我们的测试，Hermes 在执行字节码和文本 JS 上有一些很有意思的特性，这些特性让升级成本变得非常低：

- Hermes 支持执行纯文本的 js
- 支持动态加载纯文本 js 或者 bytecode
- 支持 bytecode 和纯文本 js 混合使用：比如 a.hbc 是 bytecode，模块中引用了 b.js，b 模

块是纯文本 js。在加载的时候可以先加载 a.hbc 文件，然后加载 b.js 文件。可正常执行。

七、Hermes 目前的问题

Hermes 诸多优点让我们团队非常兴奋，几乎觉得应该立马把 JavaScriptCore 下掉，更换至 Hermes。但随着测试和集成的进行，Hermes 带来的问题逐渐显现。

7.1 bytecode 文件占用 size 过大问题

Hermes 编译的字节码文件比纯文本 js 文件增大 100%。

携程旅行 App 的安装包中有 20MB（7z 压缩后）左右的 RN 业务代码，如果都编译成 bytecode，将会再增加 20MB 大小，这是无法接受的。另外，动态下发 RN 增量包时，由于是二进制文件 diff，差分效率极低。

为了解决这个问题，我们根据 Hermes 的特性，转变思路，将 Hermes 的 bytecode 编译放到客户端去做，客户端同时存储 js 和 bytecode 文件，如果有 bytecode 编译完成则使用 Hermes，否则仍然使用 JavaScriptCore。

Hermes 开源项目提供了编译 bytecode 的 complieJS 方法，但这部分代码没有默认打包到 RN 的 Hermes 引擎中，我们稍加整合、封装，通过 JNI 暴露出来，供业务使用。

拿最大的 RN 业务包（1100 个文件，6.5MB 大小），做测试，后台线程执行，小米 9 Android10 耗时 2.49 秒；三星 S6edge+ android 7.0 耗时 6 秒。由于 bytecode 不是必须，因此该耗时尚可接受。

7.2 执行纯文本 js 耗时长

在客户端将纯文本 js 转换成 bytecode 之前，我们让 Hermes 加载纯文本。但实际测试下来，发现 Hermes 加载纯文本的性能比 JavaScriptCore 要慢将近 30%。主要原因是 Hermes 删除 JIT 功能，致使对纯文本 js 代码运行变慢。

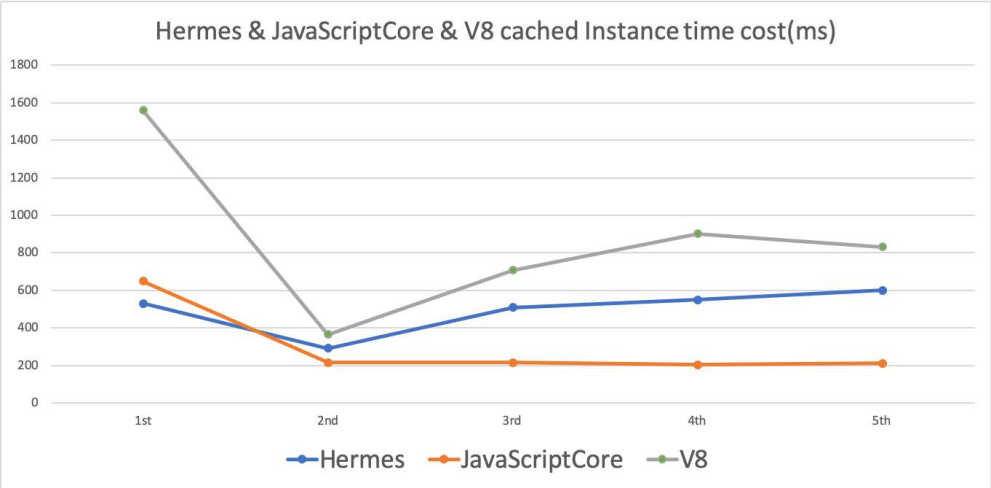
7.3 缓存问题

我们对原生 RN 框架做了大量的优化，缓存使用过的 JS 执行引擎是优化过程非常重要的一环。

拿门票页面举例来说，如果用户启动 App，第一次进入门票业务将会使用一个全新的 JavaScript 引擎并从磁盘读取文件、加载文件、执行 JS 代码。用户退出门票页面之后该引擎被缓存，如果用户再一次进入将会使用缓存的引擎，不用重新读取、加载和执行，仅仅需要创建相关 JS 对象并渲染即可。

遗憾的是，测试 Hermes 的缓存的时候，我们发现使用缓存的 Hermes 引擎加载业务代码表现非常一般，甚至某些情况下比第一次加载还要慢。而使用缓存的 JavaScriptCore 引擎，第

二次打开页面的速度与打开纯 native 页面的速度几乎相当，并且表现相当稳定。



为什么使用缓存的 Hermes 引擎打开页面速度不理想，可能和 Hermes 的设计有关，我们还在进一步分析中。

八、总结与展望

从目前情况来看，在解决缓存问题之前，我们无法在线上版本直接引入 Hermes。

解决缓存问题之后，可以采用 JavaScriptCore+Hermes 双引擎。通过客户端转换 bytecode 字节码。使用 jsc 加载优化之前的纯文本 js，一旦优化完毕切换至 Hermes 引擎。

另外如果使用 Hermes 引擎我们需要充分测试稳定性和兼容性。

Hermes 通过预编译字节码的方式提升 js 执行速度，给了我们新的思路。我们也正在调研 JavaScriptCore 或者 V8 的 bytecode 在移动端的支持度，性能和兼容性。

携程 Trip.com App 首页动态化探索

【作者简介】马增翼，高级软件工程师，目前负责 Trip.com App 公共业务研发和迭代，专注跨平台开发技术研究。热爱和拥抱开源社区。

Trip.com 是携程面向国际市场的全新品牌，受全球化战略影响，随着商业化的脚步加快，首页业务针对产品迭代速度、迭代质量、线上效果验证等方面面临新的挑战。

首页无论在哪个 App 体系下都是主要的流量入口，迭代效率和质量，以及研发成本是我们追求和要解决的问题。

在首页业务迭代中我们遇到以下几个痛点：

1）新的业务需求需要跟版本发布：在商业化越来越快的同时，有大量的产品需求需要快速上线验证其效果，目前在一个大版本需要较长时间周期的大背景下，以及 iOS AppStore 审核机制的存在，最少也需要 1 天，最慢无限期的 Review 时间。在不断缩小发布周期的大方向下，通过动态下发的方式也是我们探索的一大方向。

2）多端的研发成本：在互联网越来越快的趋势下，如何节约人力成本，解放生产力，更好更快的业务需求迭代是我们一直的诉求，多端一致性和快速 UI 搭建也是我们一大痛点。

3）业务埋点维护性和及时性：在验证产品效果的同时，会产生很多难以维护的埋点代码，在页面越来越复杂的情况下，维护成本急剧上升，当线上埋点缺失，产品和运营无法验证产品效果必须等待下个版本搭车上线。

基于以上痛点，我们推出 FoxPage Native 平台，下文将会从方案选择，框架架构，以及对于业务与技术结合思考和实践的经验分享给大家。

名字来源：FoxPage 是 IBU （国际业务部）内部一个在线可视化制作页面的平台。我们决定基于 FoxPage 搭建 Native 平台。

一、框架

1.1、技术选型及思考

首先需要明确定位以及边界，我们需要一个怎么样的框架去解决存在的痛点。

需要的：

- 由于在首页场景使用，高性能和稳定性是最基本的要求；
- 为了不跟随版本发布，所以动态性也是要考虑的；
- 为了解决研发成本，多端渲染也是需要解决的问题；

不考虑的场景：

- 不需要处理复杂的业务逻辑；
- 不支持动画精细的交互场景；
- 不考虑多个组件的联动性；

通过梳理场景和边界使得目标清晰。我们需要一个跨平台支持动态性并且高性能 UI 渲染框架。

站在自身需求的角度，调研业界成熟的方案得出的结论如下表。

方案	性能	动态性	学习成本
React Native	中	高	高
Flutter	高	无	高
Native + DSL	高	高	低

React Native：动态性高，但是学习成本和性能（加载性能、页面性能）不理想；
Flutter：谷歌的跨平台框架，性能高，但是无动态性；

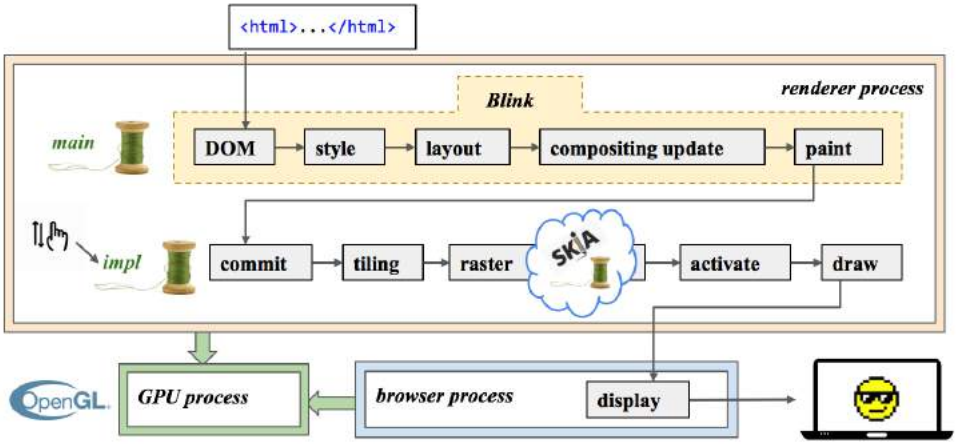
通过以上的调研，我们打算用 Native 解析 JSON + Flexbox 的方式来作为最终方案。

这么做有什么优势？

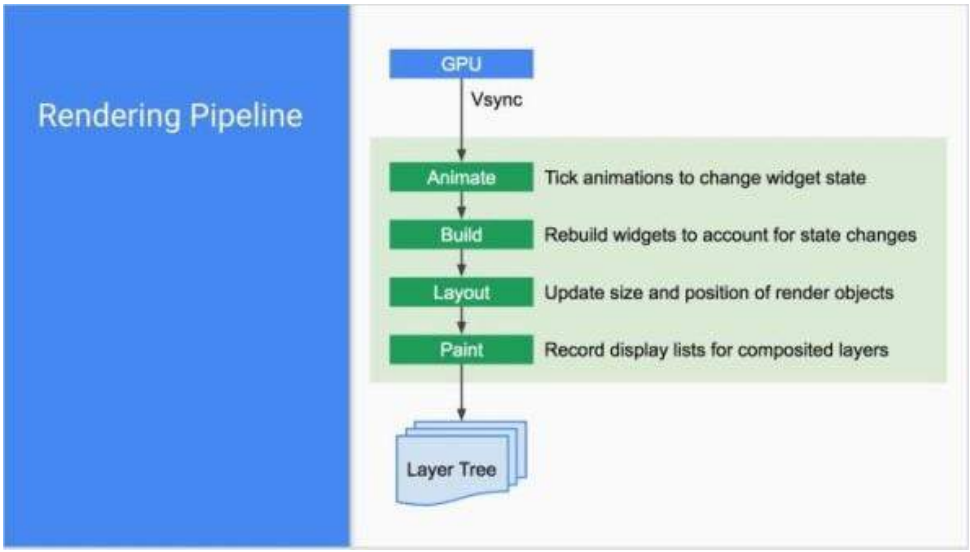
1）学习成本低：Flexbox 布局方式被开发广泛接受（内部跨平台技术栈用的多的是 RN）；
2）开发成本低：JSON 和 Flexbox（Yoga）都有成熟的高性能可靠的第三库直接使用，加快框架开发速度（在一个月内将框架完成并且上线）；
3）兼容性强：Flexbox 完美兼容 Web 端布局的方式，FoxPage 同时支持 Web 端的 DSL 的输出；
4）自定义&扩展强：由于自研，没有包袱，可以在设计上以最符合我们的场景来设计框架；

1.2、架构设计

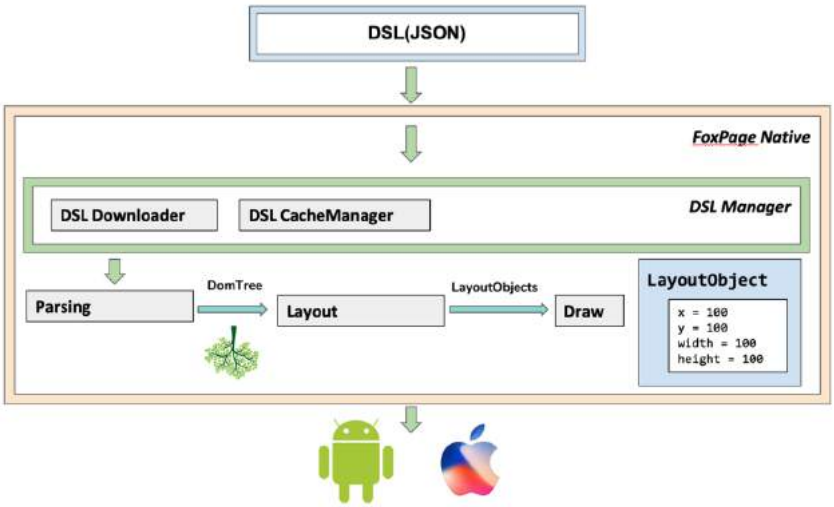
如何做好架构设计，可以先了解下 Chrome 是如何完成一个 HTML 到 UI 的输出。



那么 Flutter 渲染流程是如何呢?



通过调研沉淀下我们的渲染流程:



各个模块的职责清晰且独立：

- Downloader：主要负责 DSL 更新与下载。
- CacheManager：顾名思义，负责 DSL 的缓存管理。
- Parse：这层主要是做 DSL 解析，负责将 JSON 数据组织成节点，供下层使用。
- Layout：此层职责为将 Parse 模块解析之后的数据计算布局，生成布局元素。
- Draw：此层职责为将 Parse 生成节点设置 Layout 层的布局信息输出 Native 视图树并提交系统完成渲染。

遵守职责化最小原则，每个模块被赋予的职责最小化并且彼此独立，使后续的维护，可以将影响范围限制模块内部，而不影响其他模块的稳定性，增强系统维护性。也使得对模块定制优化提供基础。

下文会继续针对框架内部原理做深入介绍。

1.3、DSL 的定义

数据绑定

想象一下，在我们日常开发中，往往是数据对应一个 UI 元素的显示，需要有一定的绑定数据机制。

```
{
    "parentId":"cfb87570-82d8-11e9-811f-0906d1cca8d4",
    "name": "image",
    "props": {
        "url":"{{product.imageUrl}}",
    },
    "extendId": "",
    "conditions": [],
    "type": "trip-app.image",
}
```

变量的格式为{{变量名}}，如以上示例：我们的 image 组件中 url 的属性被设置为 product 对象属性中的 imageUrl 的值。

```
{
    "parentId":"cfb87570-82d8-11e9-811f-0906d1cca8d4",
    "name": "image",
    "props": {
        "url":"{{products[0].imageUrl}}",
    },
    "extendId": "",
```

```
    "conditions": [],
    "type":"trip-app.image",
}
```

数组取值格式为{{数组[Index]}}，如上，我们可以通过此方法获取 products 数组中的第一个元素的图片。

事件

在组件触发事件的时候，我们希望能做一些自定义的事情，如跳转页面，怎么定义呢？

```
{
    "onClick": {
        "name":"router_call",
        "props": {
            "value": {
            "plugin":"router",
            "method":"openURL",
            "args": {
                "url":"{{products.deeplink}}"
            }
            }
        },
        "type":"function.call",
    }
}
```

以上示例表示在点击事件中通过 router 中的 openURL 打开了一个新的页面。

条件判断

在某些条件成立才渲染的场景下，我们也提供了条件判断，示例：

```
{
    "props": {
        "hidden": {
            "name":"render-activity",
            "type": 1,
            "props": {
                "items": [
                    {
                        "key":"{{data.showActivities}}",
                        "operation":"eq",
                        "value":"1"
```

```
                        }
                    ]
                }
            }
        },
        "type":"trip-app.image",
}
```

如上示例： image 组件是否隐藏通过{{data.showActivities}} =="1"来控制。

埋点机制

我们还定义了动态埋点的一些规范，如示例：

```
{
    "name": "image",
    "props": {
        "$traceData": {
            "onClick": {
                "eventName":"home.click.deals.item",
                "data": {
                    "url":"{{data.operatingActivities.0.deeplink}}",
                    "type":"operating_activities"
                }
            }
        }
    },
    "type":"trip-app.image"
}
```

在 image 组件中声明了点击事件，并且把需要的参数，通过 data 字段一并上传服务端。

1.4、布局

我们的目标是为了解决动态性和多端一致性，那么具备一个完备的布局能力是一个基础要求。通过调研，有以下 3 种方案可选。

方式	实现成本	布局效率	通用性
自定义	高	中	差
Web CSS	高	底	高
Flexbox	底（Yoga）	高	高

1) 自定义: 完全自定义一套规则, 实现成本高, 布局效率取决于实现程度, 所以这边是"中", 因为是自定义，所有通用性是三者最差的，几乎独家专属。

2）Web CSS：实现一套 Web CSS 样式集，可想而知，如果实现这样的一套系统代价是极高的，为了兼容众多的 CSS 样式，布局效率必然会下降，但是此方案通用性也是最佳，多端共享。

3）Flexbox：弹性盒子布局，从 Web CSS 子集发展而来，在 RN 已得到充分证明它的适用性，由于 Yoga 的存在，让我们在实现成本上得到下降。

Yoga 是 Facebook 基于 Flexbox 的跨平台布局引擎开源库, 被用于 RN, Weex 等项目中, 也证明了其高性能和可靠性。

在实际使用过程中，Yoga 在 Android 中发现了一些问题，不过我们通过定制源码完美解决，并且在实际体验下来 Yoga 完美的胜任了布局的任务。

一些布局示例:

```
"props": {
      "$layoutStyle": {
            "position":"absolute",
            "bottom": "12",
            "flexDirection":"column",
            "marginLeft": "8",
            "width": "100%",
            "paddingRight":"16"
      }
}
```

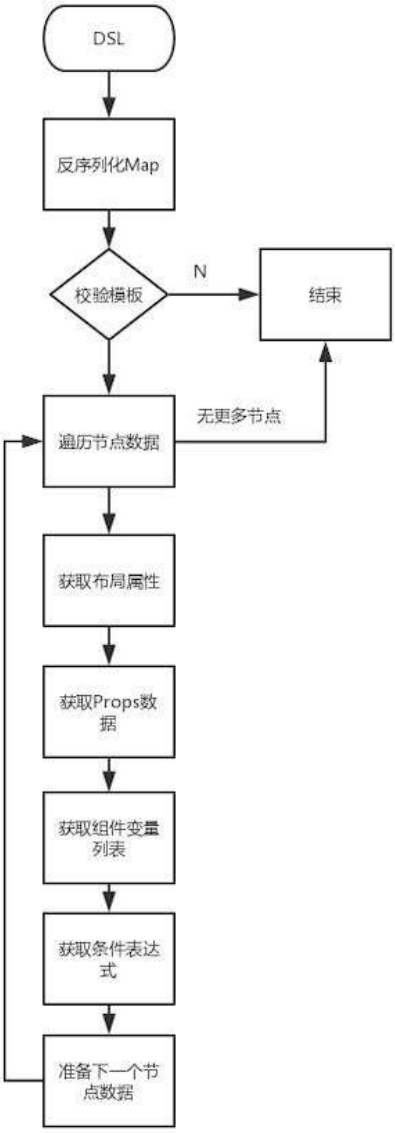
单位定义

width、height 等实际数字单位定义中，我们定义了如下的数字单位。

数字	单位	说明
N/A	Undefined	无定义的数字，如RN的图片宽高为undefined
N/A	Auto	自动缩放类型大小
100	Point	和屏幕像素无关的单位，在 iOS 中表示为pt, Android 为dp
100	Percent	百分比，当前父容器的百分比大小

1.5、DSL 解析

解析层，要做的事情比较简单，为了提高性能，并且对于相同的 DSL 模板，只会做一次解析，之后便会把结果做一个缓存，以下的流程图代表着解析流程。



1.6、视图构建

视图构建相对简单，通过解析层解析之后，每个视图组件都会 ViewNode 节点一一对应视图在虚拟视图树中的状态，包括了视图布局属性，视图属性等元素信息。

以下为 iOS 代码示例：

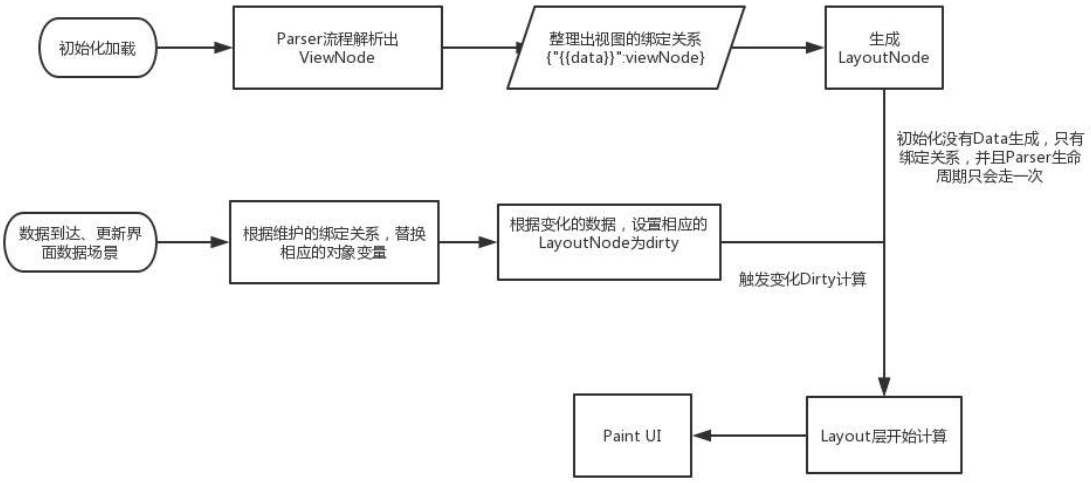
```
/**
 视图节点，映射 FoxPage 中的组件
 */
```

```
@interfaceFPViewNode : NSObject
/**
 视图属性
 */
@property(nonatomic, strong) FPViewAttribute *attribute;

/**
 子视图
 */
@property(nonatomic, copy) NSArray<FPViewNode *> *children;

/**
 绑定关系的值
 */
@property(nonatomic, copy, readonly) NSArray <NSDictionary *> *bindValues;

/**
 绑定关系的值
 */
@property(nonatomic, strong) NSMutableArray<NSString *> *conditions;

/**
 真正的视图引用
 */
@property(nonatomic, weak) UIView<FPViewComponentProtocol> *view;

/**
 添加的父视图
 */
@property(nonatomic, weak) UIView<FPViewComponentProtocol> *superView;

@end
```

在 ViewNode 树准备好之后，我们会将树传递到渲染层中进行渲染操作，在渲染层中，根据 ViewNode 节点的类型，通过代理的方式，从注册的组件之中创建出视图实例，配合 Yoga 布局属性，转换到 Native 视图的映射，由系统完成最终的渲染。

1.7、数据更新

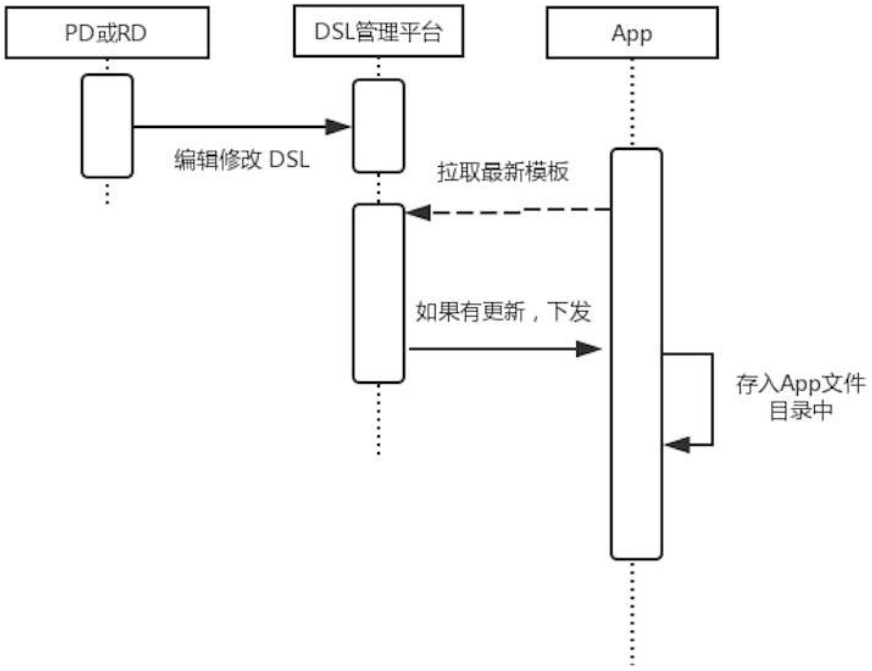
在解析渲染完成之后，关于数据流是怎么处理的呢？以下是处理流程图：



为了优化性能，我们针对 UI 元素有变化的部分做 dirty 处理，会触发 Layout 和 Draw 模块重计算和重绘。

1.8、动态更新

动态更新能力是重要的一环，在云端更新了页面布局或者样式之后，App 需要即时拉取到最新的 DSL 模板。以下是流程中的时序图:



需要注意几点:

1）App 打包需要把线上目前可用的 DSL 模板打包进 App 中，避免第一次打开 App DSL 模板未下载的时候的空窗口现象;

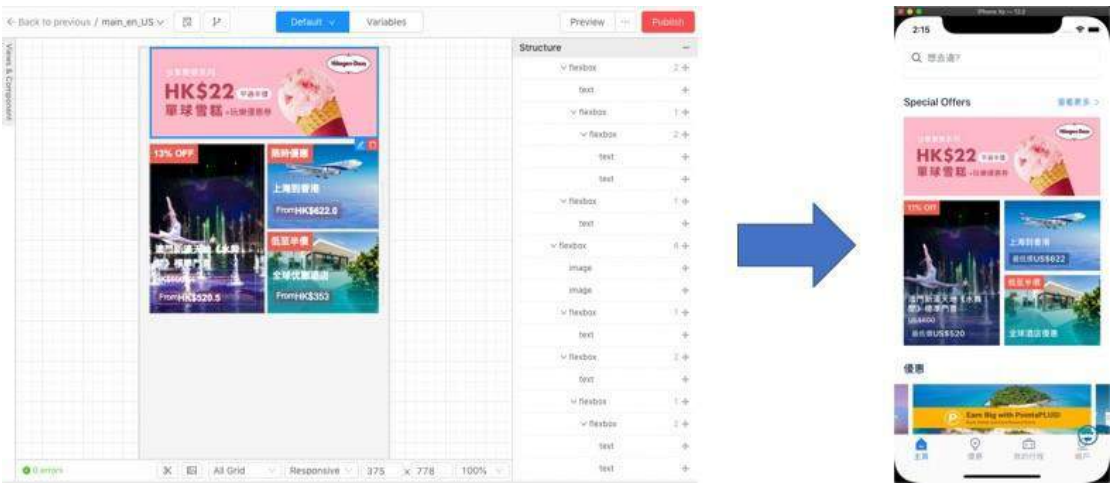
2）版本升级需要做好数据隔离和清除；
3）DSL 最新版本下发，需要做好 backup 与异常校验；

通过动态更新机制，改变了我们发布需要跟随版本的痛点，有问题，修复之后可以直接下发到用户的 App。

1.9、可视化页面搭建平台

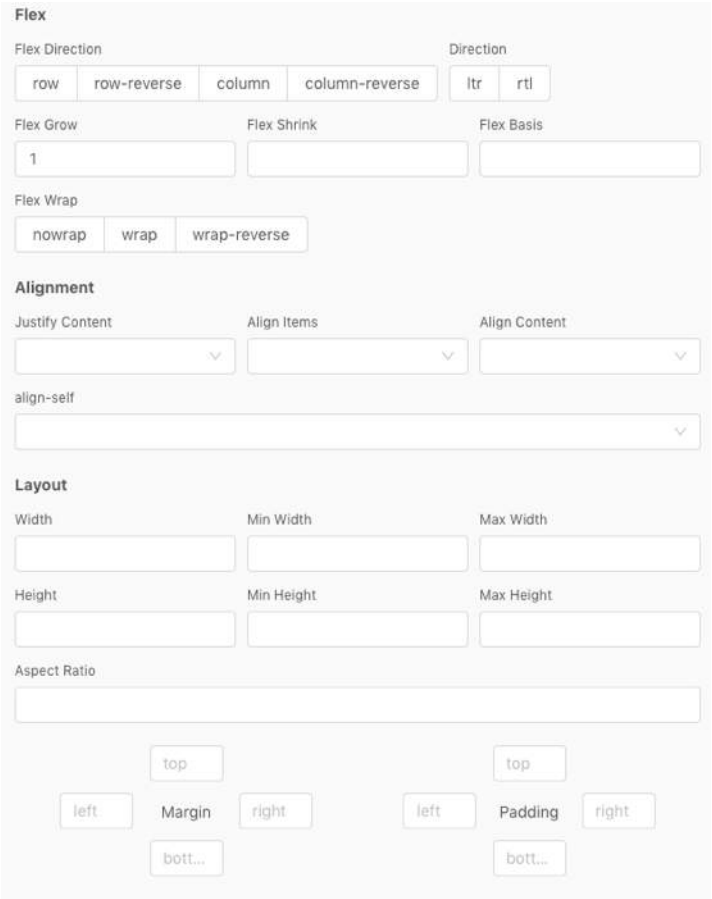
看到这里的看官，心中肯定会有疑问？你们提供变量、布局、事件、埋点追踪，条件，手动来写这很模板复杂度肯定是很高的，不是普通人可以胜任的吧。

是的，我们也意识到了此问题，所以配套了一套可视化的编辑界面。如下面示例图：



左边是可视化编辑页面，右侧为实际在 App 场景的使用效果，可以看出还原度还是很高的。

属性编辑界面：



二、页面工程化的转变

通过动态化的转变之后，首页的业务需求开发的工程模式与研发流程也由此发生变化。

在旧模式下，研发人员更加关注业务需求如何实现，首页的业务需求如何在已有的框架体系之内跑起来。新模式下，研发人员更注重的是，业务组件如何设计，如何完成的一个高质量的业务组件。

将研发人员关注的复杂度从面降为点，使得首页的各个业务模块之间的独立性更高，以及更加稳定。模块之间的复用性提升，如其他业务部门也想用广告组件，他们只需要在其页面做些简单的配置。

在组件生态不断补充的未来，各个业务线之间共享彼此的组件模块，想完成一个新的业务或者产品，只需要像乐高积木一样堆砌即可。

三、构建业务运营闭环

在提供技术基础的条件下，我们继续思考技术和业务之间的关系，如何将业务价值最大化，UI 搭建可以通过平台搭建，是不是可以把产品运营同学也一起参与进来，构建一个业务运营闭环。

1）产品运营同学提出需求；
2）研发人员介入需求开发，开发组件；
3）组件搭建业务上线之后，一站式追踪线上业务价值；
4）根据平台的数据来实时进行运营策略，如修改页面模块，下线模块，添加模块等等；
5）然后反推产品同学提出更合理的产品需求；

如何优化链路，科学的运营体系构建运营业务闭环也是重中之中，并且未来会持续不断在此方向上探索。

四、总结

FoxPage Native 平台上线短短 2 个月，承载了 30+ 次的业务调整，5+ 次临时调整埋点的需求，这是之前做不到的。并且合理系统的设计也增加了框架稳定性，上线至今无任何异常发生。在保证的高质量的交付的同时，也大大减少我们研发成本，如一个复杂的展示模块开发，从原来的双端 4 人日降低到双端 1 人日。

在首页动态化的探索中，遇到了很多的挑战，也有很多收获，后续有很多的功能和需求还需要继续优化和完善，并且需要考虑更多的场景支持。我们相信这是一个好的开头。

从智行 Android 项目看组件化架构实践

【作者简介】陈杰，智行火车票高级开发工程师，目前主要负责智行火车票 Android 客户端的架构和公共基础业务开发，热衷于 Android 技术的研究和开源分享。

一、前言

智行火车票早期以火车票业务起步，随着整体的业务发展和扩张，先后增加了汽车票、机票和酒店模块，逐渐打造成了一个提供出行、旅行和住宿一站式预订服务的 OTA 平台。

在业务扩张过程中，之前 Android 项目单一工程的架构模式慢慢暴露出一些问题，例如业务间耦合较多，整体项目编译耗时等，渐渐无法满足业务开发需求。

为了解决面临的问题，综合主流的 Android 项目架构方案，团队选择了组件化架构方案对项目进行了调整和实践，抽离出基础组件库、独立的业务模块，实现了各独立业务的拆分和独立运行，可以单独进行需求开发，发版时再合并到一起编译打包和发布。

在组件化架构实践过程中，团队解决了组件化调整中遇到的一些难题，对组件化技术在 Android 项目中的应用有一定的参考价值和实践经验。同时根据业务需求，还实现同一个项目进行多个应用差异化适配打包的功能，便于开发和维护团队旗下的其他应用。

二、概述

本文主要根据智行 Android 团队在组件化架构调整中的实践过程以及最终的实践成果，从以下几个方面来进行阐述：

- 为什么要进行组件化架构调整
- 组件化结构调整的实施步骤
- 组件化调整过程中遇到的难题以及解决方案
- 组件化架构调整的成果

2.1 组件化调整的原因和目标

如前面提到的，在调整之前，项目是单一工程的架构模式，这也是常见的 Android 项目架构模式，但是一旦项目整体业务增多，扩张出相对较为独立的业务模块，这种架构就会带来一些问题，例如：

- 业务间代码层面耦合太重，业务之间隔离不明确：由于各业务间代码存在较多的耦合，经常出现某个业务线功能开发迭代影响其他业务线，出现代码冲突，影响其他业务功能。
- 项目整体源码较多，编译耗时久：各业务开发人员主要开发各自业务线需求，但是需要编译整个项目，耗时较多，影响开发效率。
- 多应用差异化适配方案不完善：在业务扩张过程中，还衍生出一些独立应用，例如智行旗下的订票助手、智行机票等应用，实际是使用同一个项目打包，更改一些主题配色和

首页入口，进行差异化的编译打包。之前使用的多应用打包方案存在一些的问题，逐渐无法满足实际需求。

参考技术社区的 Android 架构方案，以及结合项目实际情况和业务场景，我们选择了组件化方案来进行架构的调整。

Android 项目组件化，最早是冯森林老师在 2016 年 MDCC 大会上的《回归初心，从容器化到组件化》演讲中提出来的，当时该方案刚提出，实际应用到项目中的还是比较少得，毕竟一般的公司项目业务不是很复杂，项目结构也是较为单一，没有使用组件化的必要。

但对于此时的智行 Android 项目而言，正是组件化架构最适合实践的项目，多个业务线，项目整体比较庞大，业务间不必要的耦合过多，因此组件化架构的调整方案也就应运而生。

在进行调整之前，团队也定下了调整预期的目标:

1）业务解耦，使得各业务模块可以独立运行，同时可以组合编译打包
2）拆分基础组件，抽离出基础组件 Library
3）各业务间通信和业务交叉调用的实现
4）实现多应用差异化适配打包

以上大的目标点主要是来解决之前遇到的问题，也是项目架构调整的首要目的。

2.2 组件化架构调整的整体规划

2.2.1 基础组件的拆分

智行 Android 项目的基础组件主要分为业务基础组件和功能基础组件，其中业务基础组件包含登录组件、自定义 View 组件、项目网络层组件等，这些和业务有关联，提供给各业务模块的基础组件，根据具体情况拆分成 aar 或者 library，像登录，基础网络层这样较为稳定的组件，一般直接打包成 aar，减少编译耗时。而像自定义 View 组件，由于随着版本迭代会有较多变化，就直接以源码形式抽离成 Library。

基础组件的调整相对较为简单，主要就是按照功能或者业务拆分成 Library ，处理好之前的引用的地方即可，但是对于拆分出来的 Library 的质量和后续维护工作是要求相对较高的，作为基础的组件，是需要为各业务模块提供基础的功能的，重要性是相对较高的。

基础组件库的编译版本设置一般是和主工程同步的，为了方便后续升级和维护配置，可以使用如下的方式来实现 library 使用同一份配置:

```
ext.libDefaultConfig = {
    minSdkVersion 19
    targetSdkVersion 25
    javaCompileOptions {
      annotationProcessorOptions {
```

```
            includeCompileClasspath = true
        }
    }
}
```

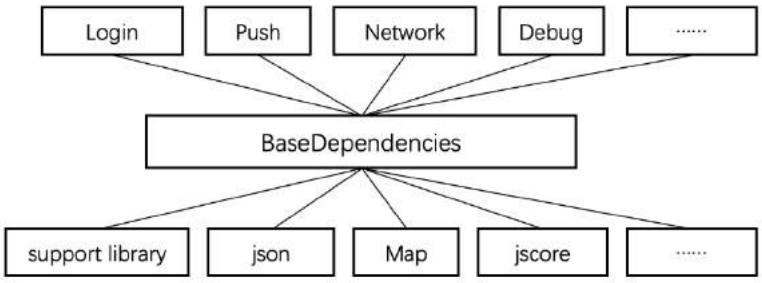
定义一个通用的 DefaultConfig 配置，设置统一的 SDK 版本信息和编译选项，在 Library 的 build.gralde 文件中使用如下方式即可应用到配置：

```
android {
        ...
    defaultConfig libDefaultConfig
}
```

这样既可以保证基础组件库的编译配置统一，也方便后期统一修改和升级。

对于再基础点的组件，例如 Support Library、json 库等，绝大多数基础组件都会使用到。为了避免每个独立基础组件都去引入对应的依赖，还要尽可能得保证版本的统一，我们使用了一个空壳 Library 来一次性引入这些基础的依赖组件。

这个 Library 叫做 BaseDependencies，然后其他的基础组件去依赖 BaseDependencies，这样就可以保证基础组件对这些基础的依赖版本做到一致，后续的升级改动位置也相对较为集中。如此调整后的依赖如下图所示：



当然这样的也有一定的弊端，就是每个基础组件都可能存在引入冗余的依赖，对于后续可能需要提供给第三方的基础组件，还需要进行改动才可以独立出来。

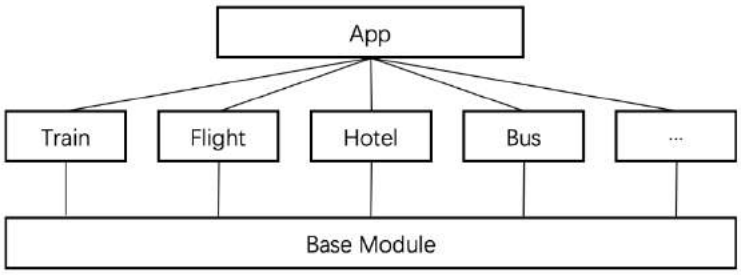
2.2.2 业务模块的拆分和配置

业务模块的调整是组件化中最重要的，这里的模块也是组件，对于这类组件的调整目标就是做到能够独立运行。业务模块的调整主要分两步：

1）业务模块的拆分独立
2）业务模块的配置

由于之前各业务的代码以不同包名来进行区分，各业务代码也是比较集中的，拆分出来还是相对较为容易的。遇到两个业务模块都会使用到的类的话，就将对应的类下沉到 Base Module，当然这种情况也是尽可能去避免，否则 Base Module 会越来越臃肿，如果不加以控制，那么业务模块就变成了一个空壳，失去了组件化原本的意义。

拆分成独立模块业务，彼此之间是平级的关系，无依赖关系，从而从结构层面达成了分离的目的，避免了之前一不小心就出现的类相互引用，耦合严重的问题。



业务模块拆分成独立的 Library 以后，就是对其进行配置，这也是组件化的关键步骤，既要使得各个业务模块可以独立运行，又要保证各个模块作为整体 App 的一部分，关键就在于不同场景下给每个业务模块应用不同的插件类型。

独立运行时，需要使用：

```
apply plugin: 'com.android.application'
```

而分独立运行时，则需要使用：

```
apply plugin: 'com.android.library'
```

为了方便业务模块的在这两种模式下的快速切换和统一调整，我们使用了以下的设置方式：

```
// 项目的 build.gradle 中配置模块是否独立运行
def isSingleCompile = false
ext.isSingleCompile = isSingleCompile

if (isSingleCompile) {
    ext.COMPLIEMODE = 'com.android.application'
} else {
    ext.COMPLIEMODE = 'com.android.library'
}
```

// 业务模块的 build.gradle 中应用
apply plugin: COMPLIE_MODE ```

这样在切换时，只需要修改 isSingleCompile 的值，就可以在独立运行和作为模块运行之间切换。

当业务模块独立运行时，还需要配置独立的 Application 和启动页，以及一些特殊的资源文件，这里同样是根据 isSingleCompile 的值来配置 sourceSets 中的属性：

```
sourceSets {
    main {
        manifest.srcFile MANIFEST_FILE
        res.srcDirs = RESOURCES
        java.srcDirs = JAVA_SOURCES
        ...
    }
}
```

这里配置内容不再赘述，可以参考 Android 官网的说明进行设置，主要是针对独立运行时配置 Manifest 文件和添加入口页面的调整。

2.2.3 业务间的通信

对于拆分成独立部分的业务模块而言，彼此业务出现关联的场景还是比较多的，比如火车票推荐酒店，就需要从火车票模块跳转到酒店模块。

对于这样的业务场景，除了之前提到的将部分业务下沉到 BaseModule 以外，针对页面间跳转，我们采用了路由的方式。由于跳转的场景可能是原生页面、网页和 React Native 页面，我们制定了一套规则来进行通用的跳转。

跳转的链接按照以下的格式来实现：

sy://suanya.cn/xxx?url=xxx&type=1

其中的 type 则是跳转的类型， url 参数的值就是实际的跳转的地址。对于 网页和 React Native 页面，url 的值是比较容易直观的，对于原生页面，则引入了 ARouter 实现，对于需要传递的参数的场景，则采用 query parameters 的方式进行传递，在统一的地方进行处理，转换成 ARouter 传参的形式。

2.3 组件化架构调整中遇到的一些问题

2.3.1 业务模块的 Manifest 文件维护

在之前提到，业务模块独立运行时需要指定 Application 和启动页面，Manifest 文件内容如

下：

```
<application
        android:name=".ModuleApplication"
        android:label="@string/app_name">

        <!-- 模块单独运行需要的 activity -->
        <activity
            android:name=".ModuleLaunchActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
        <activity
            android:name=".ModuleHomeActivity"/>
        <!-- 模块单独运行需要的 activity -->

        <activity
            android:name=".activity.ModuleXXXActivity"/>
            ...
</application>
```

其中的 ModuleApplication、ModuleLaunchActivity 和 ModuleHomeActivity 合并打包时，根据 sourceSets 的设置，是不会编译进来的，需要调整 AndroidManifest 文件，最简单的办法就是写两份 AndroidManifest 文件，通过 sourceSets 中的 manifest.srcFile 来指定。但是这样存在一个问题，例如添加一个 ModuleXXXActivity，这个在独立和非独立运行时都需要的 Activity，则需要在两个 AndroidManifest 中都添加一次，这样显然是不够合理的，对开发而言是不友好的。

我们通过 manifest merge 规则找到解决办法，业务模块只需要维护独立运行时的一份 AndroidManifest 文件即可，在合并的 App 的 AndroidManifest 中，对 application 节点，使用 replace 操作：

```
<application
        android:name=".MainApplication"
        android:theme="@styleAppTheme"
        tools:replace="android:theme,android:name,">
```

而对于只有模块独立运行才使用到的 activity ，则采用 remove 操作进行移除：

```
<activity
    android:name="com.xx.xxx.ModuleLaunchActivity"
    tools:node="remove"/>
```

```
<activity
    android:name="com.xx.xxx.ModuleHomeActivity"
    tools:node="remove"/>
```

如此操作之后，就可以保证最终合并打包时，业务模块设置的 application 被替换成 app 的，而模块独立运行才应用到的 activity 则被移除了，只需要维护一份模块的 AndroidManifest 文件即可。

2.3.2 多应用差异化配置打包

在前面也提到，我们的业务场景对于同一个项目打包出不同的应用，这种需求的我们使用 ProductFlavors 进行了实现。

通过设置不同的 ProductFlavors，通过 manifestPlaceholders 来配置每个应用差异化的参数，例如接入微信的 appId，地图的 key 等。对于每个应用使用不同的主题色和资源问题，则采用在对应的 ProductFlavors 文件夹中以同名文件、同名资源名称的方式进行覆盖设置，这种同名的资源最终以 ProductFlavors 文件夹中的设置为准。

除此以外，还需要针对每个应用的签名配置进行设置：

```
productFlavors.all { flavor ->
    signingConfigs.create(flavor.name, getSigningConfigsByFlavorName(flavor.name))
    flavor.signingConfig signingConfigs.getByName(flavor.name)
}
```

需要定义 getSigningConfigsByFlavorName 方法来根据 flavor name 获取到对应的 signingConfigs。

三、组件化架构的实践成果

根据之前设定的目标，组件化调整后基本都完成的预期的目标：

1）业务模块分离，从结构层面做到了代码隔离，减少了之前不必要的耦合
2）基础组件的拆分，按照业务和功能拆分出基础组件，便于后期开发和维护
3）简单实现了业务间的通信，实现了跨模块的多种类型的通用跳转
4）实现同一个项目多应用差异化适配打包，支持主题适配等

整体项目进行组件化调整以后，模块的划分更为清晰，结构上实现了代码隔离，减少了耦合。业务模块支持独立运行和整体打包，单个模块完整编译耗时约 20 秒左右，合并打包完整编译整个项目耗时约 1 分钟，极大地提升了开发效率。

Node. js 在携程的落地和最佳实践

【作者简介】潘斐斐，Trip.com 高级研发经理。2008 年加入携程，目前工作内容为 Node.js 框架平台整体构建、产品性能优化和创新型项目研发。本文来自在 2019 携程技术峰会上的分享。

本篇主要介绍在携程，Node.js 技术栈是如何实现从 0 到 1 进行技术落地的，以及在不断磨合的过程中，总结出来的最佳实践。

在携程 Node.js 应用根据用户群，主要分两个方向：

DA（数据聚合服务）和 SSR（服务端渲染）是服务于外部用户的，目标是提升用户体验。当然，DA 和 SSR 同时也提升了开发效率，例如前端开发人员可以更加灵活的整合数据，例如同构给开发人员省去了大量重复的开发工作量；

公司桌面工具（例如内部 IM 等）是服务于内部员工的，一般是用 Electron，开发维护成本低，产品迭代快。

一、Node.js 工程化

基于上述三个场景，目前携程有一套 Node.js 的工程化方案。工程化的方案并不是一成不变的， 在任何阶段遇到了实际问题， 都会更新甚至推翻一些步骤，为的就是更好的服务于整个应用开发的生命周期。

工程化涵盖五大部分：开发、构建、测试、发布和运维。

1.1 开发

1）脚手架

有三个类型的脚手架：Web Application、DA Service 和 Desktop Tools。这三种类型的脚手架会服务于上述提到的三种场景。

这三种脚手架有共同点：标准化的 Docker 日志，预置统一的中间件。但同时他们也是有差异的，例如 Desktop Tools 和 Web Application 的应用模型不一样，Desktop 有 UI 层，那么 UI 层和应用层上的应用日志和用户行为如何关联，方便后续的排障；DA Service 需要将应用的健康状况周期性上报给治理中心、熔断机制等等，这些框架层面的差异，脚手架会集成进去，做业务开发同学可以不用关心这些基础设施的接入。

2）核心中间件

核心中间件主要是做基础设施的建设。目前有 20 多个中间件，主要的中间件如下：

图 1 核心中间件介绍

- 存储服务，主要应用于长期的固化存储，例如静态资源。主要提供的是 Ceph 客户端。
- 业务服务，主要应用于 DA 场景，提供 SOA Client 和 SOA Service。SOA Client 用来获取数据，需要重点关注的是读取性能和容错处理；SOAService 用来提供对外的聚合服务，需要重点关注的是稳定性和响应性能。
- 监控服务，涵盖所有的应用，提供三个维度的监控：Tracing、Metrics 和 Logging。具体的介绍请参看下方"运维"部分。
- 公共服务，主要包括配置中心，ABTest 的客户端、数据访问层等。
- 缓存服务，主要用于配置信息的缓存、应用数据的缓存。提供 Redis 客户端和共享内存两个中间件。

1.2 构建

1）Docker 镜像

Node.js 的版本更新频率很快，每 6 个月会发布一个大版本的升级，期间会陆续出很多小版本。如果为每个版本都做一个镜像，会带来极高的开发和运维成本。基于更新频率，我们目前选取 2 个固定版本，在 Node.js 版本更替的时候，可以保证一个稳定的镜像。

2）安装依赖包

为了提升开发效率，在构建时安装依赖包需要保证速度快。如果中间件中用到一部分 C++模块，那么在安装时会做实时编译，这样会导致耗时长，甚至会因为环境问题编译失败。所以我们会将用到 C++模块的中间件做一下预编译，为 windows、linux 和 mac 这三个平台分别编译出 2 个固定版本的预编译包。

3）依赖包扫描

扫描的目的主要解决几个问题：

应用中不同的包如果引用了同一个子包，但是子包的版本不一致，就会导致应用中装了多个版本同一个包，会引发 bug；

中间件缺乏治理能力。通过扫描依赖包，能够做到中间件统一收口。一旦要升级，可以很快

的通知到开发做快速升级。例如第三方依赖包有安全问题，可以在构建环节就提醒开发人员升级版本。

1.3 测试

目前测试环节包括单元测试、集成测试、压力测试和自动化测试。自动化测试主要针对 Service 和 UI 两方面测试。UI 自动化测试使用的是 Puppteer。每次代码更新，会走一遍自动化测试流程，保证代码质量。

1.4 发布

1）携程云和公有云

每个云的部署环境、网络、位置等差异，会带来应用访问差异，例如访问异常，网络延迟等。这些差异需要在基础设施层面抹平，避免放在应用逻辑层面处理。

2）应用一体化发布

一体化发布也可以理解为一键发布。一条发布指令包含了应用核心框架、静态资源、配置的同时发布，而不需要开发人员思考什么步骤需要发什么资源。这样不仅可以提升效率，还能有效的控制发布回滚。

3）私有 npm 包发布

私有包的发布和 GIT 做高度集成。原因是：第一可以通过 git 做快速的发布；第二有历史可查，方便的查看到每个版本发布的时间、人员；第三有权限控制，避免发生生产级别故障。

1.5 运维

运维是整个环节中最重要也是最容易被忽略的环节。一个应用上线只是开始，真正要关注的一定是运维指标。

1）日志监控

三种维度的监控：tracing、logging 和 metric。

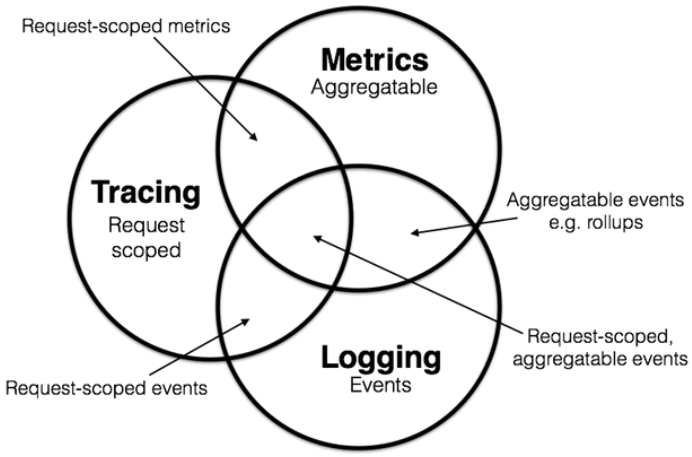


图 2 三种维度的监控

图片来源于网络：http://peter.bourgon.org/blog/2017/02/21/metrics-tracing-and-logging.html

Tracing 提供的是整个请求过程中的数据，例如请求信息（头部、地址）、响应信息（状态码，响应体）、请求耗时、调用链等信息。

Logging 提供的是在请求处理过程中，每一个具体的事件埋点，这些埋点相对是分散的。可以是记录普通的日志，也可以是记录抛出的错误。

Metric 提供的是聚合数据。最大的特征是可聚合的，它展现的是一个时间跨度中的某个维度的指标。一般用来记录量化的指标，例如访问量、性能等数据。

2）应用排障

一般我们排查问题的时候，会先通过 Metric 的聚合指标发掘出异常，然后追踪到某一批有异常的 Tracing，可以查看到调用链、耗时等具体情况，也可以跟踪到某一个请求，查看里面的事件埋点。

也有其他方式的排障，例如下图中展示，可以在线直接通过一个特殊的地址访问到的一张火焰图，可以非常快速地去排障。当有用户说这个页面出现问题，打开这个页面排障，可以定位到底那个对应的地方出现问题。

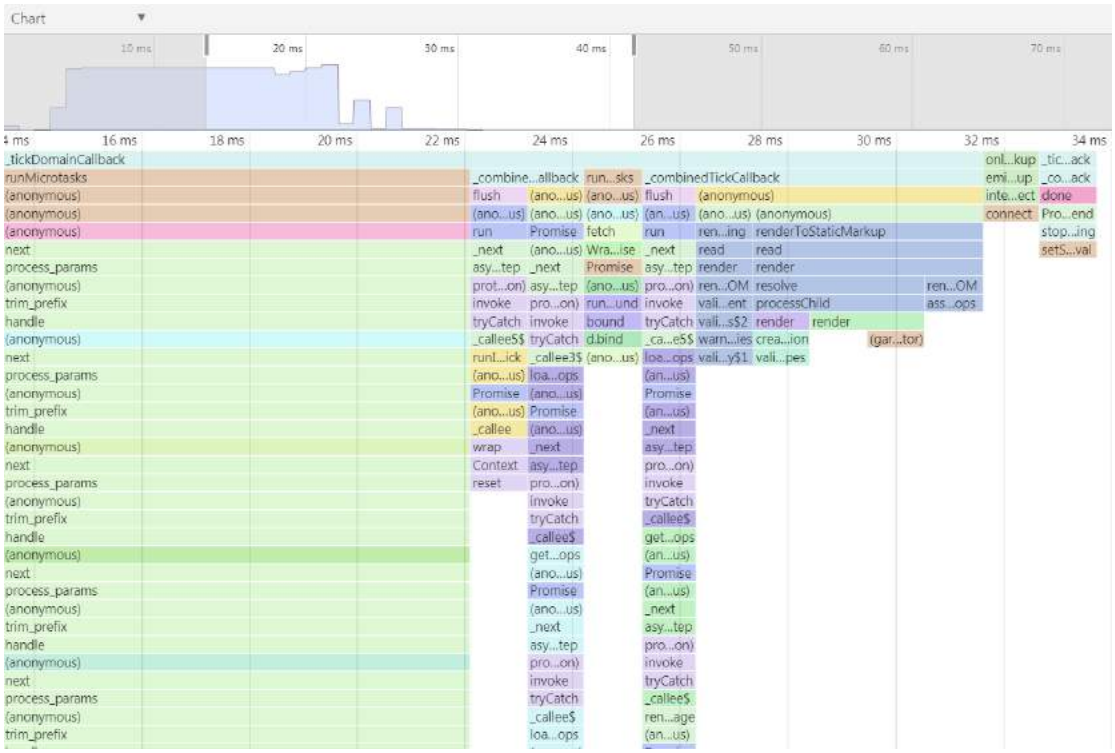


图 3 火焰图

二、Node.js 最佳实践

2.1 部署模型

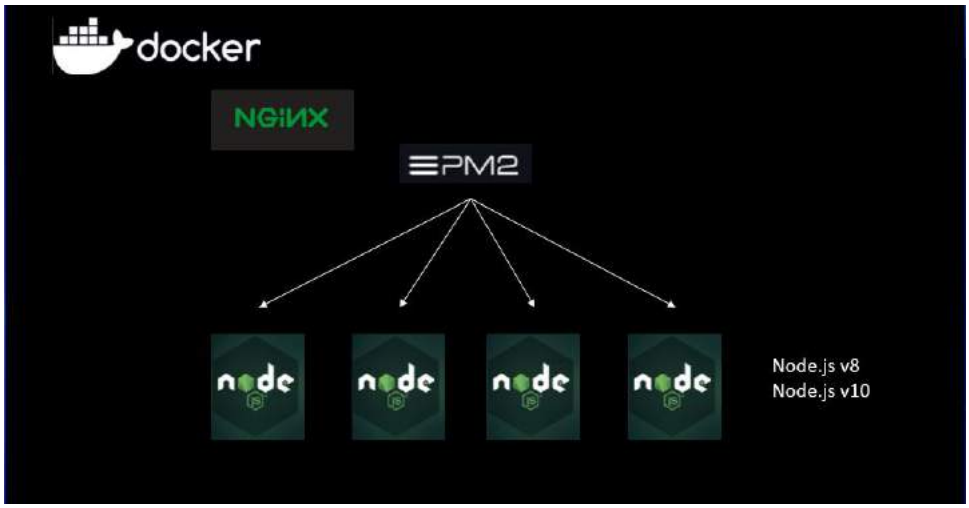


图 4 部署模型

Node.js 应用部署在 Docker 上，采用 Nginx+PM2 的模式。

2.2 问题一：多进程通信

多进程通信主要用于数据交换，最常见的有 2 种场景：

1）提供 SequenceId：在单台机器需要提供唯一的并且按时间序列排列的 ID。
2）提供远端配置信息：当获取远端配置信息时，需要考虑多进程的共享分发。

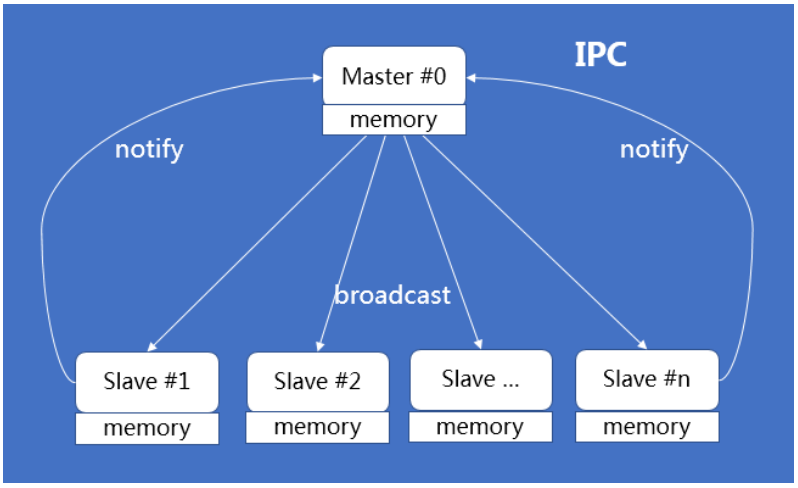


图 5 多进程通信 V1.0

在第一版本设计中，我们采用的是 IPC 机制进行多进程的通信。Master 作为一个中转站，当 Slave 有消息分发时，通知给 Master，再由 Master 分发给各 Slave，从而达到进程之间通信的效果。

但是上线之后发现，这样的机制会遇到几个问题：数据量必须控制好体积；数据的同步会有延迟；Master 必须时刻在线，一旦 Master 进程挂掉，就需要等待重启再重连。

基于这些问题，我们重新设计了第二个版本：

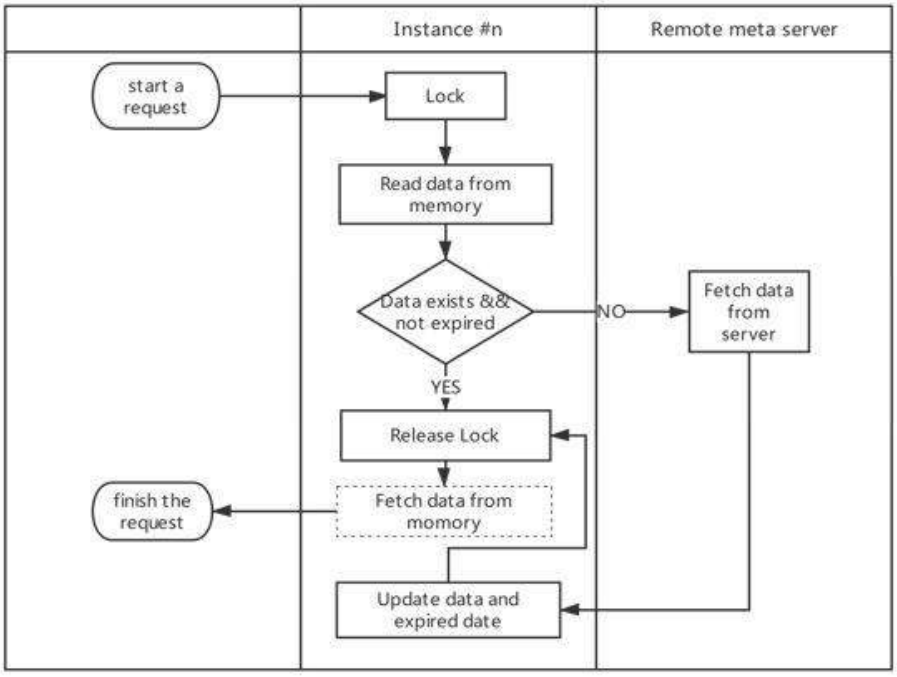


图 6 多进程通信 V2.0

在第二个版本的设计中，我们使用了共享内存（shared memory）。举一个场景为例，当需要获取某个配置的时候，先将这块内存锁定，尝试从内存中获取数据。如果判断数据存在且在有效期内，那么解锁并从内存中读取数据返回，否则从服务端获取数据，当服务端有数据返回时，将数据和有效期更新到内存中，解锁并从内存中读取数据返回。通过共享内存的机制，可以非常轻量级且高效的实现多进程之间的数据共享。

2.3 问题二：监控什么内容

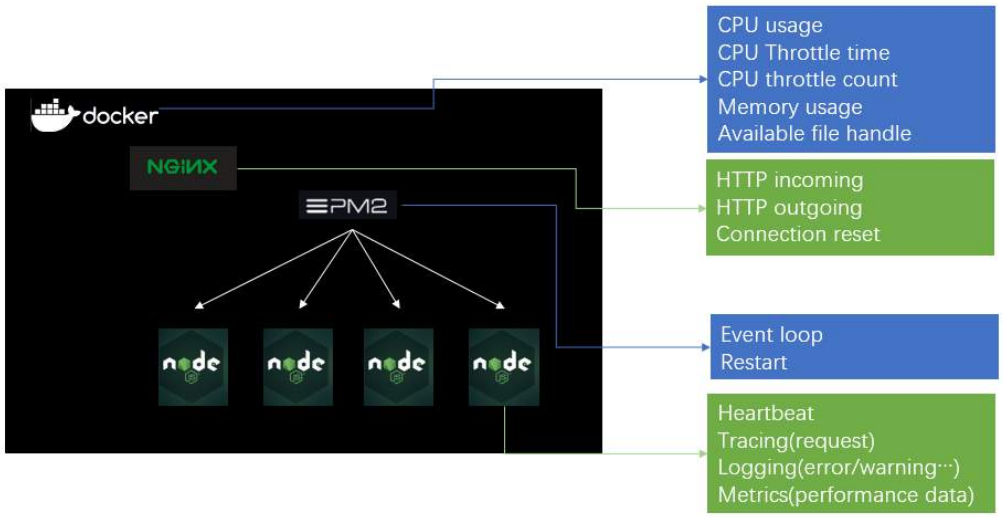


图 7 监控指标

Nginx 会监控整个 Docker 上所有应用的情况：

- CPU util：CPU 总的使用率。
- CPU throttle count&time：CPU 被限制的次数和 CPU 使用率被限制的总时间。这两个指标的上升一般表示应用有 CPU 密集型操作，需要检查一下是否有大量的计算等操作。
- Mem RSS used：这个指标上升一般显示应用内存泄漏的问题。
- HTTP imcoming&outgoging：http request 的数量变化趋势。如果有错误响应或者超过了告警的阈值，则会在趋势图中显示。
- Connection reset：这个指标如果上升，表示应用出现了大量的拒绝请求，例如是服务器的并发数超过了原本的承载量等原因。

Nginx 中监控的是整个 Docker 的情况，但是我们更需要的是监控应用的指标。应用一般采用 PM2 cluster –i max 模式启动，最大化利用 CPU。

Heartbeat（心跳信息）

每个 slave 一分钟发送一次 Heartbeat（心跳信息）给到 CAT 数据中心。一般来说，如果 Heartbeat 告警的话，需要立刻查看一下错误日志，是不是有异常错误导致进程已经退出了。

Heartbeat 主要包括 CPU、Memory、网络信息等。这些信息和上述提到的 Nginx 信息不是一个维度的。这个更细节的关注了应用的情况，而不是整个 Docker 的情况。如果需要分析应用细节的问题，是需要查看这里的 Heartbeat 信息。

性能情况

一般来说，中间件会处理应用常规的性能日志记录。包括:
1）每一个响应的请求耗时（服务端逻辑处理耗时，不包括网络耗时）。
2）每一个 Transaction 的耗时。一个 Transaction 可以简单理解为一个有功能意义的代码片段。
3）跨应用调用的请求耗时。

错误/告警信息

错误告警信息是应用中需要重点关注的，包括:
1）应用逻辑出错，例如处理 JSON 数据出错等。
2）HTTP 请求出错，会记录状态码、请求地址、返回内容。
3）应用中使用了不同版本的同一个包，会报一条告警信息通知开发工程师。

详细数据日志

详细数据日志一般有开发工程师针对应用的逻辑埋点，而非中间件统一处理。这些日志会包括返回数据的记录，具体运行在哪一段 transaction 中。这些日志一般是故障发生时，用来复盘时的辅助手段。

2.4 问题三：全链路监控

全链路监控指的是端到端的监控，监控的是一系列的调用链情况。

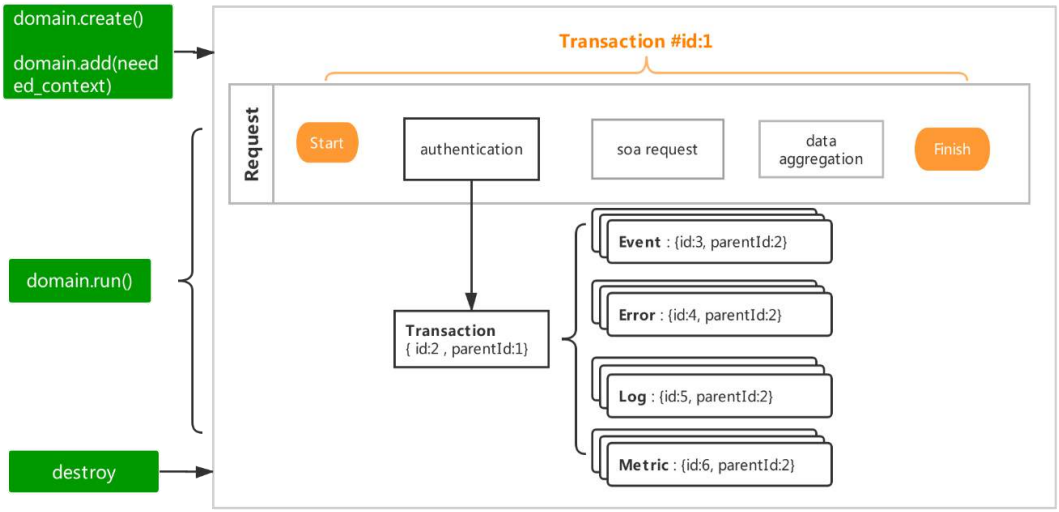


图 8 Tracing 模型

在介绍全链路模型之前，首先介绍 Tracing 模型（图 8）。Tracing 模型是一个树状结构的模型。以一个场景为例，当用户发起一个请求，这个请求的处理中有三段逻辑（authentication、soa request 和 data aggregation）。

在整个请求体外层会有一个 Transaction#1，记录请求响应等信息。每一个逻辑段会对应一个 Transaction#2，Transaction#2 的父节点是 Transaction#1。Transaction#2 中可以有多个 Logging 信息，根据类型可以分为 Event/Error/Log，也可以包含 Metric 信息。这些 Logging 和 Metric 都有父节点，是 Transacation#2。按照这样的结构可以将一整个 request 的过程的监控信息记录下来。

要做全链路监控，就是需要将每个 request 和调用链做关联。

在过程中遇到的最核心的问题是，如何将上下文进行关联。第一个版本使用的是 domain 的模块，使用 domain 的 add api 将上下文信息记录下来，使用 run api 运行逻辑代码块。第二个正在测试中的版本是使用 async_hook 的模块，引入了生命周期的概念，通过 executionAsyncId 和 ttriggerAsyncId 可以追踪每个函数体。

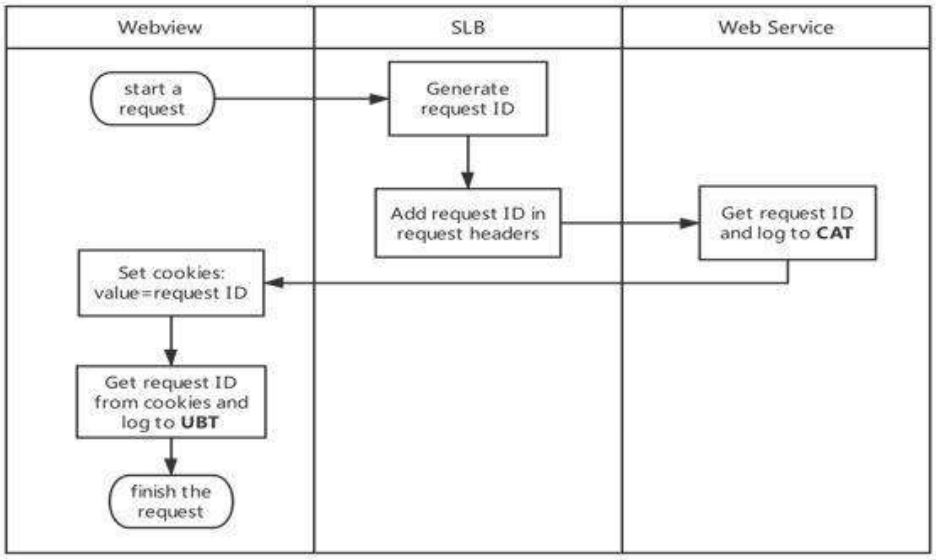


图 9 页面请求模型

通过上图的页面请求模型可以将每个请求做关联，从而达到全链路监控的效果。

三、总结

- Node.js 工程化需要结合业务，反复磨合；
- 设计好运维指标，做好 Tracing/Logging/Metric 的结合；
- 密切关注上线之后的监控指标，防止内存泄漏；
- 发掘出 Node.js 技术栈的差异，有针对性的解决问题；
- 不要盲目相信同一个技术栈，合适才是最好的。

注：“携程技术”微信公众号后台回复“nodejs”，可下载讲师 PPT。

大数据和人工智能篇

每天十亿级数据更新，秒出查询结果，ClickHouse在携程酒店的应用

【作者简介】蔡岳毅，携程酒店大数据高级研发经理，负责酒店数据智能平台研发，大数据技术创新工作。喜欢探索研究大数据的开源技术框架。

一、背景

1）携程酒店每天有上千表，累计十多亿数据更新，如何保证数据更新过程中生产应用高可用；
2)每天有将近百万次数据查询请求,用户可以从粗粒度国家省份城市汇总不断下钻到酒店，房型粒度的数据，我们往往无法对海量的明细数据做进一步层次的预聚合，大量的关键业务数据都是好几亿数据关联权限，关联基础信息，根据用户场景获取不同维度的汇总数据；
3）为了让用户无论在 app 端还是 pc 端查询数据提供秒出的效果，我们需要不断的探索，研究找到最合适的技术框架。

对此，我们尝试过关系型数据库，但千万级表关联数据库基本上不太可能做到秒出，考虑过Sharding，但数据量大，各种成本都很高。热数据存储到 ElasticSearch，但无法跨索引关联，导致不得不做宽表，因为权限，酒店信息会变，所以每次要刷全量数据，不适用于大表更新，维护成本也很高。Redis 键值对存储无法做到实时汇总，也测试过 Presto，GreenPlum，kylin，真正让我们停下来深入研究，不断的扩展使用场景的是 ClickHouse。

二、ClickHouse 介绍

ClickHouse 是一款用于大数据实时分析的列式数据库管理系统，而非数据库。通过向量化执行以及对 cpu 底层指令集（SIMD）的使用，它可以对海量数据进行并行处理，从而加快数据的处理速度。

主要优点有：

1）为了高效的使用 CPU，数据不仅仅按列存储，同时还按向量进行处理；
2）数据压缩空间大，减少 io；处理单查询高吞吐量每台服务器每秒最多数十亿行；
3）索引非 B 树结构，不需要满足最左原则；只要过滤条件在索引列中包含即可；即使在使用的数据不在索引中，由于各种并行处理机制 ClickHouse 全表扫描的速度也很快；
4）写入速度非常快，50-200M/s，对于大量的数据更新非常适用；

ClickHouse 并非万能的，正因为 ClickHouse 处理速度快，所以也是需要为“快”付出代价。选择 ClickHouse 需要有下面注意以下几点：

1）不支持事务，不支持真正的删除/更新；
2）不支持高并发，官方建议 qps 为 100，可以通过修改配置文件增加连接数，但是在服务器足够好的情况下；

3）sql 满足日常使用 80%以上的语法，join 写法比较特殊；最新版已支持类似 sql 的 join，但性能不好；
4）尽量做 1000 条以上批量的写入，避免逐行 insert 或小批量的 insert，update，delete 操作，因为 ClickHouse 底层会不断的做异步的数据合并，会影响查询性能，这个在做实时数据写入的时候要尽量避开；
5）Clickhouse 快是因为采用了并行处理机制，即使一个查询，也会用服务器一半的 cpu 去执行，所以 ClickHouse 不能支持高并发的使用场景，默认单查询使用 cpu 核数为服务器核数的一半，安装时会自动识别服务器核数，可以通过配置文件修改该参数；

三、ClickHouse 在酒店数据智能平台的实践

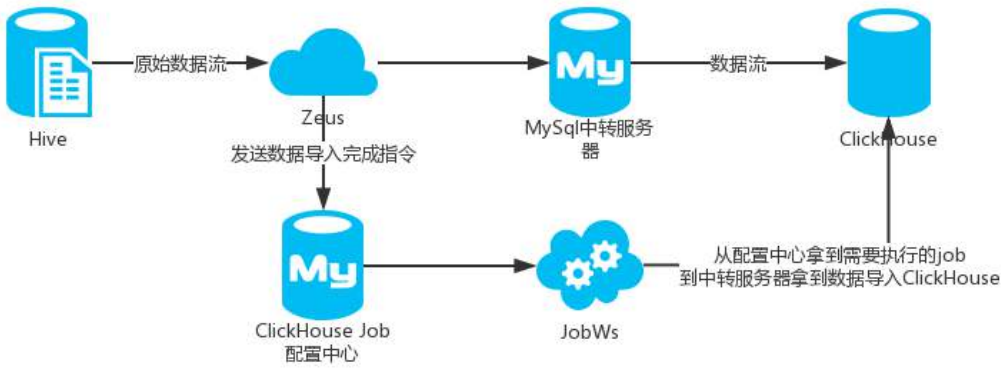
3.1 数据更新

我们的主要数据源是 Hive 到 ClickHouse，现在主要采用如下两种方式：

1）Hive 到 MySql，再导入到 ClickHouse

初期在 DataX 不支持 hive 到 ClickHouse 的数据导入，我们是通过 DataX 将数据先导入 mysql，再通过 ClickHouse 原生 api 将数据从 mysql 导入到 ClickHouse。

为此我们设计了一套完整的数据导入流程，保证数据从 hive 到 mysql 再到 ClickHouse 能自动化，稳定的运行，并保证数据在同步过程中线上应用的高可用。

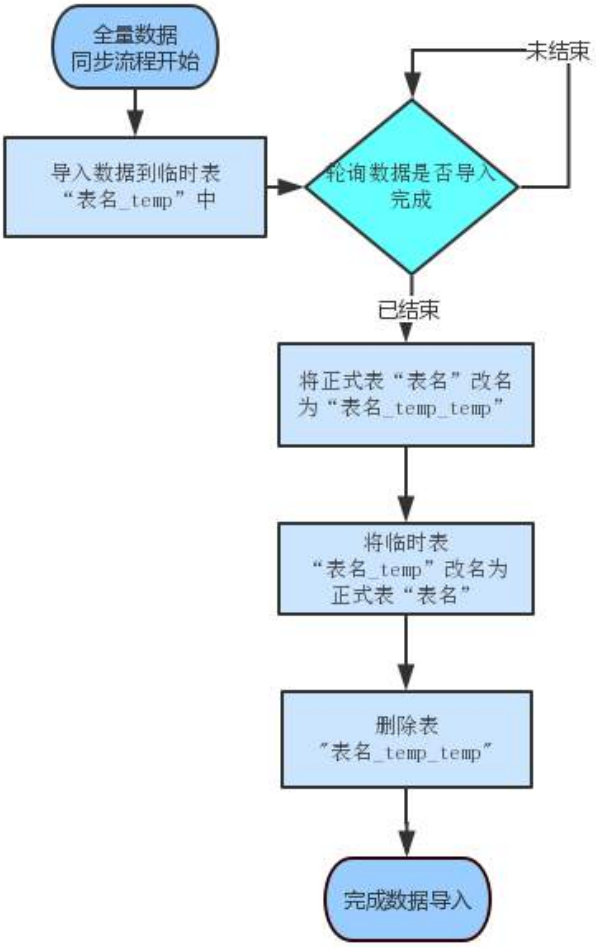


2）Hive 到 ClickHouse

DataX 现在支持 hive 到 ClickHouse，我们部分数据是通过 DataX 直接导入 ClickHouse。但 DataX 暂时只支持导入，因为要保证线上的高可用，所以仅仅导入是不够的，还需要继续依赖我们上面的一套流程来做 ReName，增量数据更新等操作。

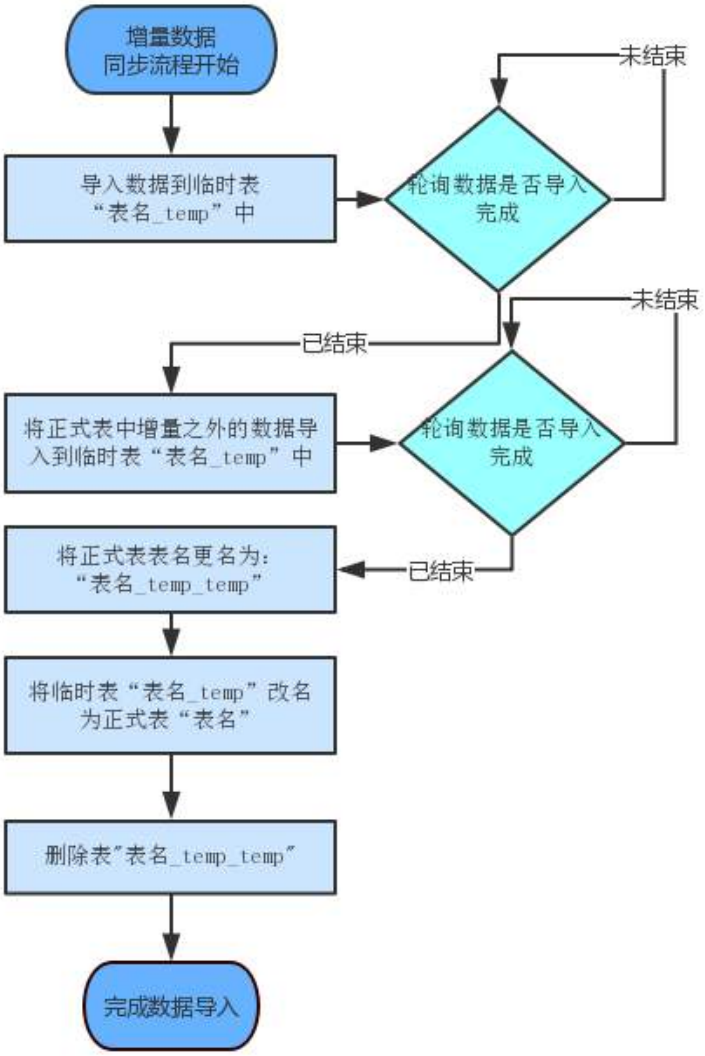
针对数据高可用，我们对数据更新机制做了如下设计：

3.1.1 全量数据导入流程



全量数据的导入过程比较简单，仅需要将数据先导入到临时表中，导入完成之后，再通过对正式表和临时表进行 ReName 操作，将对数据的读取从老数据切换到新数据上来。

3.1.2 增量数据的导入过程



增量数据的导入过程，我们使用过两个版本。

由于 ClickHouse 的 delete 操作过于沉重，所以最早是通过删除指定分区，再把增量数据导入正式表的方式来实现的。

这种方式存在如下问题：一是在增量数据导入的过程中，数据的准确性是不可保证的，如果增量数据越多，数据不可用的时间就越长；二是 ClickHouse 删除分区的动作，是在接收到删除指令之后内异步执行，执行完成时间是未知的。如果增量数据导入后，删除指令也还在异步执行中，会导致增量数据也会被删除。最新版的更新日志说已修复这个问题。

针对以上情况，我们修改了增量数据的同步方案。在增量数据从 Hive 同步到 ClickHouse 的临时表之后，将正式表中数据反写到临时表中，然后通过 ReName 方法切换正式表和临时表。

通过以上流程，基本可以保证用户对数据的导入过程是无感知的。

3.2 数据导入过程的监控与预警

由于数据量大，数据同步的语句经常性超时。为保证数据同步的每一个过程都是可监控的，我们没有使用 ClickHouse 提供的 JDBC 来执行数据同步语句，所有的数据同步语句都是通过调用 ClickHouse 的 RestfulAPI 来实现的。

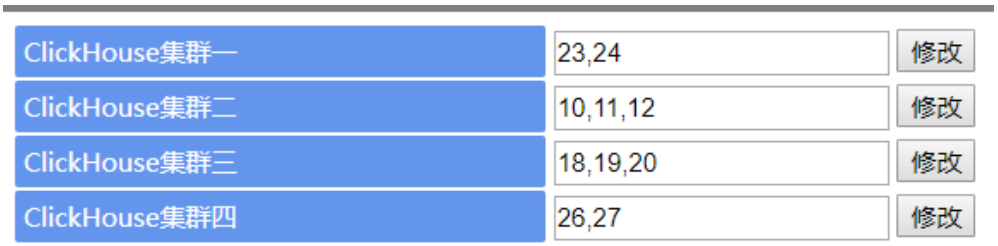
调用 RestfulAPI 的时候，可以指定本次查询的 QueryID。在数据同步语句超时的情况下，通过轮询来获得某 QueryID 的执行进度。这样保证了整个查询过程的有序运行。在轮询的过程中，会对异常情况进行记录，如果异常情况出现的频次超过阈值，JOB 会通过短信给相关人员发出预警短信。

3.3 服务器分布与运维

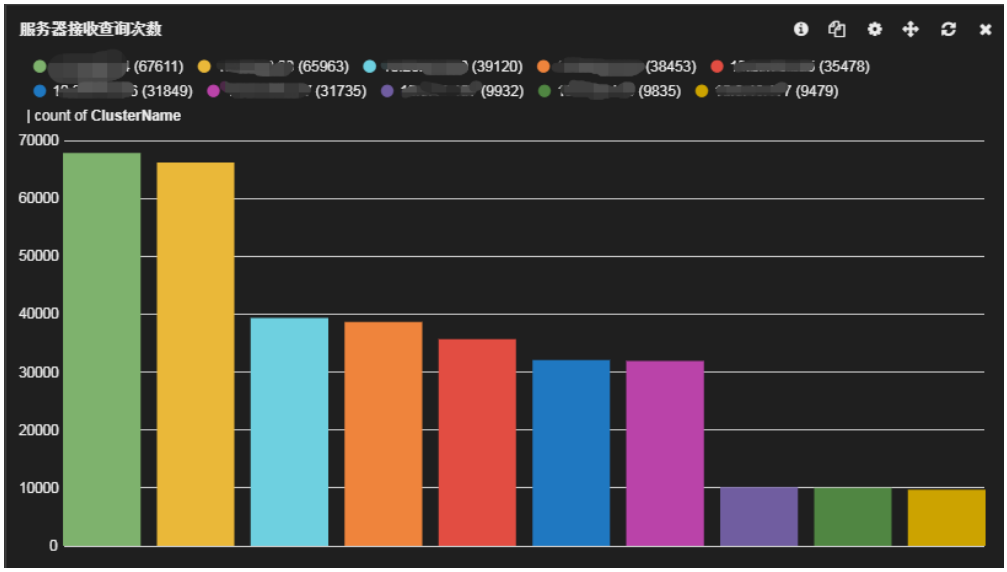
现在主要根据场景分国内，海外/供应商，实时数据，风控数据 4 个集群。每个集群对应的两到三台服务器，相互之间做主备，程序内部将查询请求分散到不同的服务器上做负载均衡。

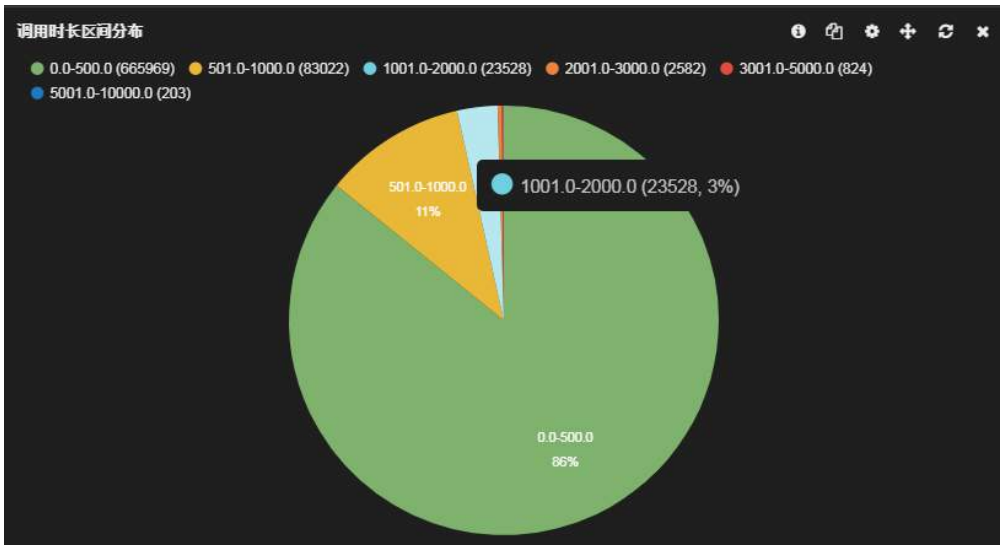
假如某一台服务器出现故障，通过配置界面修改某个集群的服务器节点，该集群的请求就不会落到有故障的服务器上。如果在某个时间段某个特定的数据查询量比较大，组建虚拟集群，将所有的请求分散到其他资源富于的物理集群上。

下半年计划把每个集群的两台机器分散到不同的机房，可以继续起到现有的主备，负载均衡的作用还能起到 dr 的作用。同时为了保障线上应用的高可用，我们会实现自动健康检测机制，针对突发异常的服务器自动拉出我们的虚拟集群。



我们会监控每台服务器每天的查询量，每个语句的执行时间，服务器 CPU，内存相关指标，以便于及时调整服务器上查询量比较高的请求到其他服务器。





四、ClickHouse 使用探索

我们在使用 ClickHouse 的过程中遇到过各种各样的问题，总结出来供大家参考。

1）关闭 Linux 虚拟内存。在一次 ClickHouse 服务器内存耗尽的情况下，我们 Kill 掉占用内存最多的 Query 之后发现，这台 ClickHouse 服务器并没有如预期的那样恢复正常，所有的查询依然运行的十分缓慢。

通过查看服务器的各项指标，发现虚拟内存占用量异常。因为存在大量的物理内存和虚拟内存的数据交换，导致查询速度十分缓慢。关闭虚拟内存，并重启服务后，应用恢复正常。

2）为每一个账户添加 join_use_nulls 配置。ClickHouse 的 SQL 语法是非标准的，默认情况下，以 Left Join 为例，如果左表中的一条记录在右表中不存在，右表的相应字段会返回该字段相应数据类型的默认值，而不是标准 SQL 中的 Null 值。对于习惯了标准 SQL 的我们来说，这种返回值经常会造成困扰。

3）JOIN 操作时一定要把数据量小的表放在右边，ClickHouse 中无论是 Left Join 、Right Join 还是 Inner Join 永远都是拿着右表中的每一条记录到左表中查找该记录是否存在，所以右表必须是小表。

4）通过 ClickHouse 官方的 JDBC 向 ClickHouse 中批量写入数据时，必须控制每个批次的数据中涉及到的分区的数量，在写入之前最好通过 Order By 语句对需要导入的数据进行排序。无序的数据或者数据中涉及的分区太多，会导致 ClickHouse 无法及时的对新导入的数据进行合并，从而影响查询性能。

5）尽量减少 JOIN 时的左右表的数据量，必要时可以提前对某张表进行聚合操作，减少数据条数。有些时候，先 GROUP BY 再 JOIN 比先 JOIN 再 GROUP BY 查询时间更短。

6）ClickHouse 版本迭代很快，建议用去年的稳定版，不能太激进，新版本我们在使用过程中遇到过一些 bug，内存泄漏，语法不兼容但也不报错，配置文件并发数修改后无法生效等问题。

7）避免使用分布式表，ClickHouse 的分布式表性能上性价比不如物理表高，建表分区字段值不宜过多，太多的分区数据导入过程磁盘可能会被打满。

8）服务器 CPU 一般在 50%左右会出现查询波动，CPU 达到 70%会出现大范围的查询超时，所以 ClickHouse 最关键的指标 CPU 要非常关注。我们内部对所有 ClickHouse 查询都有监控，当出现查询波动的时候会有邮件预警。

9）查询测试 Case 有：6000W 数据关联 1000W 数据再关联 2000W 数据 sum 一个月间夜量返回结果：190ms；2.4 亿数据关联 2000W 的数据 group by 一个月的数据大概 390ms。但 ClickHouse 并非无所不能，查询语句需要不断的调优，可能与查询条件有关，不同的查询条件表是左 join 还是右 join 也是很有讲究的。

五、总结

酒店数据智能平台从去年 7 月份试点，到现在 80%以上的业务都已接入 ClickHouse。满足每天十多亿的数据更新和近百万次的数据查询，支撑 app 性能 98.3%在 1 秒内返回结果，pc 端 98.5%在 3 秒内返回结果。

从使用的角度，查询性能不是数据库能相比的，从成本上也是远低于关系型数据库成本的，单机支撑 40 亿以上的数据查询毫无压力。与 ElasticSearch，Redis 相比 ClickHouse 可以满足我们大部分使用场景。

我们会继续更深入的研究 ClickHouse，跟进最新的版本，同时也会继续保持对外界更好的开源框架的研究，尝试，寻找到更合适我们的技术框架。

携程全局搜索推荐系统实践

【作者简介】葛荣亮，携程搜索部门高级研发工程师。2015 年加入携程，目前主要负责搜索平台的前端+数据挖掘工作。

一、前言

随着旅游业的发展，人们对搜索的要求越来越高。智能化大趋势下，个性化的推荐系统的应用及用户需求也越来越广泛。

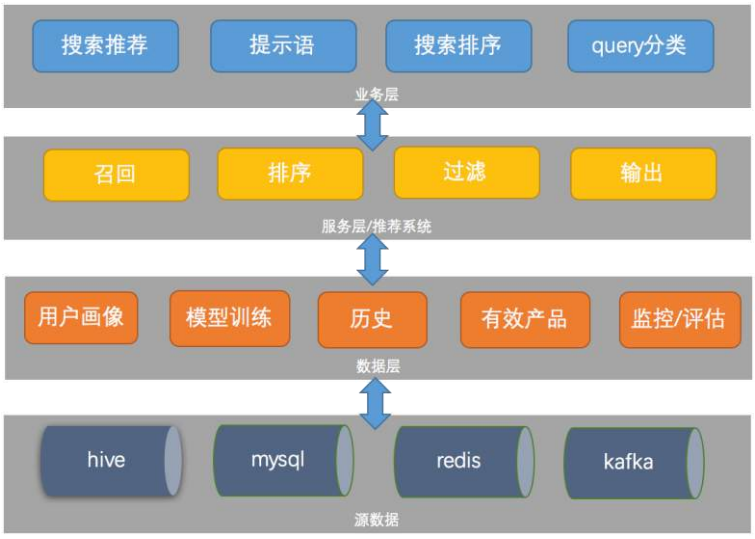
旅游推荐系统主要面临的问题及挑战包括：

- 用户维度，用户的需求多种多样，如本地异地的差异，年龄、家庭结构的差异等；
- 时间、地理维度，每个时间点的需求都是不同的，如季节(冬季的温泉，夏季避暑...)、早中晚的需求差异，不同城市用户对同一目的地的旅游产品类别需求可能不同；
- 产品维度，如何输出多样性的产品也是推荐系统考虑的重点，如相似的酒店、景点等。

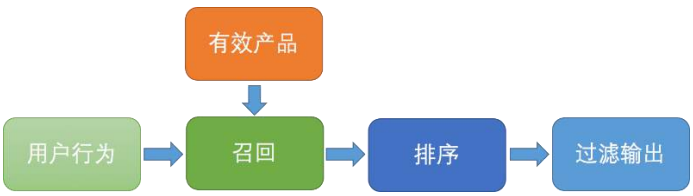
针对以上面临的问题和挑战，本文将分享携程推荐系统的更新迭代过程。

二、推荐系统架构

携程搜索推荐系统架构如下：



抛开业务和数据部分，这里只简单介绍推荐服务的结构，其简要构造如下：



2.1 用户行为

用户行为数据展示了用户的操作习惯和偏好。对这部分数据进行离线分析，可以更好地理解用户，以此来做线上产品的推荐源。

对线上需要的行为数据，可以取一个月或者近 7 天的历史数据，以保证数据的时效性。

2.2 可用产品

这部分指的是可供用户使用的产品及可以提供帮助的文章等。主旨在于告诉系统，我们有什么产品，哪些产品是可以提供给用户的，及哪些是优质的产品。产品的定义比较广泛，可以不限定具体的售卖产品，也可指定用户偏好，比如用户对酒店、景点的偏向等。

2.3.召回

这部分是整个系统的重点，也是规划场景最多的地方。这部分可以细分成几大召回策略（以推荐实际酒店、文章、景点的系统为例）：

2.3.1 补充策略

这部分主要输出当前热门的产品信息，比如当季热门的酒店、景点等。

在具体实现的时候可以考虑季节性的变化，比如以两周为周期，统计产品的点击情况，当用户对于温泉搜索量增加时，可以输出一些热门的温泉景点。

这部分补充策略，只是为了解决冷启动问题，即当用户没有行为，或者没有地理位置信息时，做最基本的补充。

2.3.2 基于位置召回

当得到具体位置信息之后，可以做更具体的补充召回：

1）根据当前用户所在地，推荐当地的热门产品；

2）判断用户是否在常住地。如常驻上海的用户，在上海搜索产品时，更喜欢周边游，而常驻北京的用户，在上海搜产品时，更喜欢东方明珠和迪士尼。

具体分类为：

本地需求(定位城市=常驻城市)，输出当地人热搜/点击的产品；
外地需求(定位城市!=常驻城市)，输出外地人热搜/点击的产品；

3）根据地理位置信息，输出用户周边的几公里内的产品。

2.3.3 基于历史关联策略

这部分内容是基于用户历史行为，推出相关的产品。需要对数据和行为进行总结，并提供相应的产品展示逻辑，丰富推荐召回的内容。比如用户预定迪士尼乐园的门票，可以推迪士尼附近的酒店等。

2.3.4 协同过滤

协同过滤是推荐系统经典的算法。其对用户行为、产品的相关性做了抽象和泛化。协同过滤算法主要分为 USER CF 和 ITEM CF，即基于用户的协同过滤和基于物品的协同过滤。

在这里我们主要用到基于物品的协调过滤，相比用户的协同过滤，物品的内容属性和数量更便于统计和计算。具体算法可以参看《推荐系统实践》这本书。

大体可以理解为，定了某一酒店的用户，又定了哪些酒店，及通常订了又订的逻辑。比如，以用户一个月的点击或订单数据为基础，计算出物品的相似度，当用户搜了某条产品时，推荐与其相似的其他产品。具体示例为：假设东方明珠、外滩、迪士尼产品相似，当用户搜索东方明珠的时，推荐外滩和迪士尼。

2.4 排序

上述召回策略，会召回大量的产品，如何对这些产品进行合理排序，是推荐系统的核心部分，同时也是反映系统优劣的指标。

这部分，经历几次迭代。

在 1.0 时代，在排序策略上进行了几次变动：

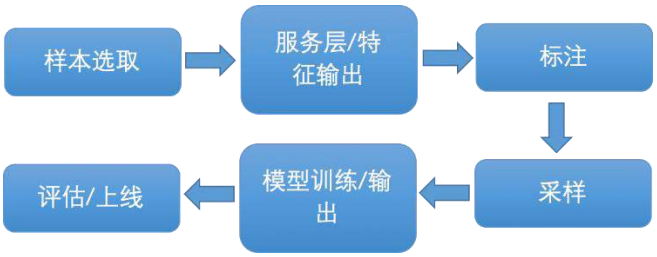
1）对召回产品按照类别，对相同类型产品，进行销量排序；
2）考虑到操作时间问题，加入操作时间权重。对历史行为的时间进行归一化得出权重，最大为 1。操作时间越近，权重越大；
3）考虑规则的重要性，加入规则分；

上述排序策略取得一定效果，但很难完善排序问题。

最终，选取了机器学习的排序算法。其基本实现为：对每个输出产品，规划特征，输出特征集。比如季节特征，当地人/外地人特征，一天内的早、中、晚特征等。接下来根据订单和点击数据，输出训练样本，为每个召回产品做一个打分，最高 5 分，最低 1 分。最后使用 XGBoost 工具，对样本进行训练，这样就能得出基本模型。

通过模型，对线上每个召回产品进行打分并排序，得出最终结果。同时在系统上线后，定期的进行更新，并通过 ABTest 系统不断对模型进行迭代。

大致流程如下：



2.5 过滤输出

这部分内容，主要做格式化输出，并过滤一些无效，黑名单产品。

每个场景的输出，都不太一样，就需要对其数据进行筛选。比如进入搜索默认页时，提前给出推荐产品，减少用户操作。还可以在用户搜某个具体城市时，输出相应的结果。

这里需要注意的是马太效应。由于推出的内容有限，对于一些产品，会导致点击多的会越来越多，而点击少的，则慢慢退出推荐序列。这里需要对那些不常用产品做展示规划。比如随机出一两条，给一定曝光，消除一部分马太效应。

三、展望

目前推荐系统已经运用在多个场景，但对场景及产品的引入毕竟有限，同时对 query 分析还不够完善，后续将丰富产品，并引入更多机器学习的内容，让系统更智能化和自动化。同时会加入更多深度学习内容，在搜索意图和 NLP 相关方面做更进一步的分析。

强化学习在携程酒店推荐排序中的应用探索

【作者简介】宣云儿，携程酒店排序算法工程师，主要负责酒店排序相关的算法逻辑方案设计实施。目前主要的兴趣在于排序学习、强化学习等领域的理论与应用。

前言

目前携程酒店绝大部分排序业务中所涉及的问题，基本可以通过应用排序学习完成。而其中模型训练步骤中所需的训练数据集，一般是通过线下收集数据来完成的。

然而在实际当中，往往存在业务新增或者业务变更，这就使得使用历史数据训练的模型，并不能很好地用于变更后的应用场景。形成该问题的主要原因，是过去所收集的数据与实际排序场景并不一致。

为了应对类似问题，我们尝试在城市欢迎度排序场景中引入了强化学习。通过实验发现，增加强化学习后，能够在一定程度上提高排序的质量。

一、实际面临的问题

在目前大部分的实践中，我们解决排序问题所诉诸的办法，基本都可以归为传统意义上的“排序学习”（learning to rank， L2R）。排序学习基本的建模过程，是通过收集数据，构建特征，选择模型，训练模型等步骤实现。

其中第一步收集数据，往往是以离线的方式完成。常见方法是通过埋点服务器收集用户行为数据，再进行模型训练数据的构建。

在收集数据的时候，一般存在一个假设要求：即我们所收集到的数据是和总体数据分布保持一致的，模型只要在这个基础上进行训练，所得到的结果在实际线上业务使用时是可靠的。我们暂且称其为“分布一致假设”。

当业务体量足够大、收集数据足够多时，一般会认为这一假设是成立的。然而在实际当中，要满足分布一致假设会面临一系列的挑战。这其中包含且不限于各种埋点服务准确性难以保证，数据处理中无法完全避免的数据损失，以及收集数据时间空间上的限制等问题。

此外，当排序应用的目标是预测现有数据集中不存在的情况时，传统的排序学习将变得无能为力。举一个商品排序的例子。在排序 list 结果展现中，我们设置业务限制：top10 的 item 只能是 1000 元以上的商品。假设现在的排序目标为点击率（CTR），从现有数据进行模型训练后，模型很难准确地告诉我们一件 10 元商品出现在 top10 时它的 CTR 是多少，因为它根本没有在这个位置出现过。

这种“预测不曾在历史中出现的事件”的需求在实际当中并不少见：在多指标融合的排序指标中（例如排序指标是要在 CTR 和转化率 CVR 之间做加权），如何设置不同子指标之间的权重就是一例。

由于新的权重下得到的数据分布是不存在现有的数据集当中的，因此无法充分知晓用户在新的排序结果下的行为结果，在新权重的条件下训练一个模型满足业务目标也就无从谈起了。

更重要的是，实际上我们很难充分收集数据：试想，对于同一个用户，难道我们要将 100 种不同的权重参数所得到的排序结果，都一一展现给同一个用户，然后收集其给出的行为结果么？答案当然是否定的。

携程酒店排序业务中，同样也存在这样的问题。具体来说有两点最为明显：

1）对于内外网比价结果为优势或者劣势的酒店，我们是否应该调整该酒店的排序位置、以及应该如何调整。这个问题的答案并不是直观的，因为对不同用户来说，可能会存在不同的偏好，以及对于不同酒店来说，这一个问题的结果也可能是不同的。

2）对于历史上由于业务设置的原因排名靠后的酒店，在个性化排序或者广告业务中若将其位置提前，如何准确预测用户对这些酒店的行为。

这两个业务需求，都可以归结为前面提到的无法预先收集同分布数据集的问题，传统的 L2R 并不能很好地进行支持解决。

二、可能的解决方案

我们再仔细考虑下“将 100 个不同排序结果丢给同一个用户”这种做法的弊端。毫无疑问，这种暴力的做法虽然似乎能够充分收集数据，但事实上是有着极高的成本：极为伤害用户体验。

然而，为了能够让模型有机会预测那些未知的情况，又必然是需要一定的“随机探索”，只有这样我们才能知道实际中用户的反馈。因此，随机探索所带来的短期损失是无法完全避免的，但最终的目标是在于探索所带来的收益能够弥补并超过其带来的损失。

而“强化学习”的目标，恰好和我们的需求不谋而合。

三、谈谈 RL 的背景，它解决的问题

为了方便后的表述，先简单介绍下强化学习（reinforcement learning, RL）的背景，对其概念熟悉的同学可以略过这一部分。

“强化”的概念起源于行为心理学派。行为心理学派主要的概念是说，人的复杂高级行为，例如恐惧，是可以用过“强化”进行塑造的。同巴普洛夫的条件反射不同，心理学派认为生物的行为不需要通过直接的刺激所引起，行为更多地是因为行为本身所带来的结果，生物体自发地执行该行为。

举个例子来说，迷信行为的出现，主要的原因是人们认为迷信行为会为自己带来好运，而不是真的会直接让我们的彩票中奖。

机器学习（machinelearning，ML）领域的 RL，就好比是利用这种行为生成理论，训练一个机器人（agent），让它能出现我们所期望的行为，以实现我们的目标。

简单来说，RL 组成要素包括执行动作（action）的 agent，agent 所在环境（environment），以及环境给与 agent 的反馈（reward）。Agent 根据对环境的观察（observation）通过给出 action 到 environment，得到 environment 对该 action 的 reward，由此 agent 判断自己 action 是否有利并由此进行迭代修正，再发出新的 action。

那么它和传统 ML 方法有什么差别呢？在我的理解来看，RL 之所以能够处理一些传统 ML 无法做到的事情，主要是在于其具有探索（exploration）能力，以及其长期累计收益最大化（maximize long term cumulative reward）的学习目标。

我们耳熟能详的 RL 应用例子 Alpha Go 为例，之所以它能够击败人类职业棋手，依靠的就是 RL 可以通过探索获得更好的长期收益，而不是完全地从人类历史棋谱中进行监督学习，从而实现对人类的超越。

对于我们前面提到的 L2R 无法处理的问题，也同样可以诉诸 RL 途径。例如前面提到电商排序问题，通过 RL 的 exploration 机制，让排序在后面的商品有机会以一定的概率在靠前位置曝光，并且在长期收益最大化的目标保证下，能够让我们随机探索的收益大于其带来的代价。

四、RL 和业务结合应用的思考过程，详述实践内容和碰到的问题，分步骤的实践方案

再回到前面提及的酒店排序中遇到的两个问题。我们认为 RL 是解决这两个问题的不二途径。成功实施 RL 的关键点在于如何设计 RL 中各要素以满足业务的需求。为此我们预先设计了几套循序渐进的方案。

第一个方案，我们就姑且称其为“方案 A”吧。方案 A 我们主要将其作为一个小规模探索，摸清流程和潜在的问题。在这个方案里，我们将不会花过多的时间在特征构建等问题上，主要是将 RL 的流程实现，并观察其带来的结果。

具体来说，我们计划在主排序的线性模型基础之上，对线性模型的若干维度进行权重调整，以实现对排序结果的调整。而 RL 的目标，就是学习这些对权重做出调整的“超参”，从而能够依照不同的输入数据，得到更优的排序序列。在粒度控制上，我们以城市为单位进行 action 输出，这样做的主要考量是数据部分的工程复杂性。

算法选择上，我们选择使用 DQN。DQN 作为 value base 的 RL 算法，无法给出连续区间的控制输出，这就意味着我们给出的权重调整需要预先定义一些固定的区间，但其易用性是我们在方案 A 中选择它的理由之一。

在工程实现上，RL 最大的挑战在于 action 发出与 reward 回收的环节对实时性的要求较高。在方案 A 我们打算利用现有 Kafka 环境，通过消息同步的方式连接前后端，进行 action 与 reward 数据匹配。这样的额外工程量最小。

但问题也很明显：Kafka 当时是对全业务埋点进行统一定时处理，势必存在更新缓慢延迟过

大的问题，且发出 action 后与实际权重更新发送到前端存在不可控的时间差，由此带来的缺陷便是 RL 模型 update 频率被明显拖慢，agent 可能需要上线很久才能达到可用的状态。

在第二步，我们称为“方案 B”，将主要改造 action 与 reward 数据的发送回收环节。方案 A 中的 Kafka 环境实时性不高，在方案 B 中，我们将采用 storm 实现流式处理，从而实现较为实时的 action 发送。在获取 reward 数据时，我们也能够更便捷地匹配到其对应的 action。

这样一来便能够显著提升 agent 学习频次。此外，由于避免了复杂前后端消息同步，算法的调整粒度也能够从城市粒度进一步提升为酒店粒度至用户粒度，从而提升 RL 的潜在效能。

在方案 B 中，我们也将对数据维度做进一步的丰富化。我们当前正在进行对酒店以及用户的 embedding 表征学习，在现有模型的线下测试中取得了一定效果。这些 embedding 维度也将在第二步方案中融合成为 RL 模型的输入维度。

第三步方案 C，我们则会将精力集中在算法的调优上。在解决了数据同步以及粒度问题后，我们打算在方案 C 中尝试不同的 RL 模型，目标是保证效果的前提下，能够用更少的参数量、更少的算法训练时间，以优化整个 RL 部分。

首先考虑可以改进替换的算法为 DDPG。DDPG 为基于 value 的 actor-critic RL 算法。首先来说，由于开销限制，即使使用了流式处理，模型在线更新频次也不能做得非常高，那么同样具有 experience replay 机制的 DDPG 能够更好地实现较高的数据训练效率。此外，DDPG 的 actor 部分可以实现连续值输出，从而能够让控制更为平滑。

方案 A 的 DQN 模型，其输出值是离散的，因而需要人为将参数调整范围进行分段，分段的多少以及每一段区间设置需要人为指定。DDPG 还有其改进算法 TD3，能够进一步稳定 Q 网络输出，缓解 TD error 的大幅度波动。

另一方面，我们可以使用 policy gradient 的算法，例如 A2C，TRPO 等，与前面 value base 的算法进行对比选优。在有一些工作中，提到相比 DDPG，TRPO 等能够减低由于 Q-value 估计的波动带来的不稳定性。但在我们看来，这些 policy base 方法由于其 action 输出存在随机性，对训练数据量的要求可能会更高。

最后，我们也将会考虑尝试基于高斯过程(Gaussian process, GP)的汤普森采样（Thompsonsampling）的算法方案。领英在它的首页混排排序问题中，就应用了这样的构架。

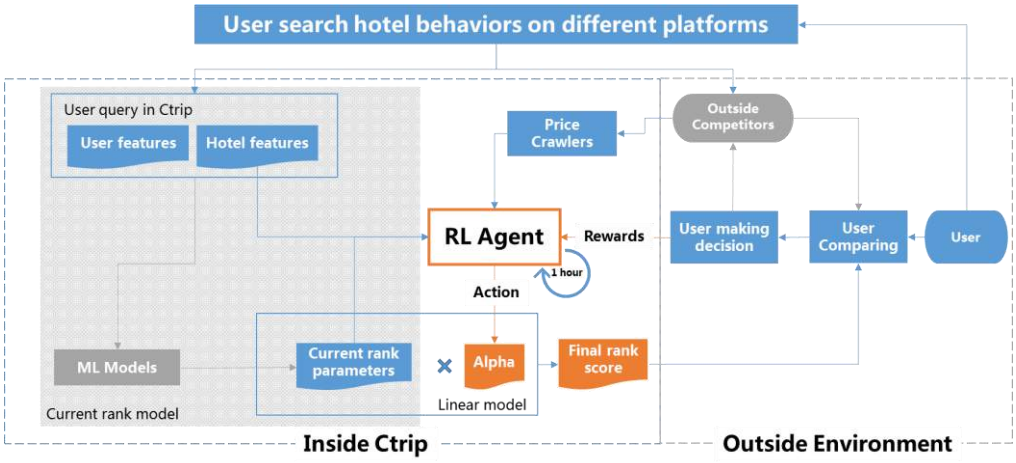
前面提到的一系列 RL 算法都是需要构建深度神经网络（DNN），需要依赖 TensorFlow 或者 PyTorch 来实现。而基于 GP 的方案基本只涉及相对简单的数值计算。当然，作为非参数模型，需要额外空间来存放样本数据。

总结一下我们的 RL 实施规划，主要包含了前期的流程探索，打通流式数据处理并丰富数据维度，以及后期的各种算法尝试及调优这样三个阶段。

五、最后的实践说明，初步探索

现在我们已经完成了方案 A 的实施，通过结果初步说明了 RL 起到了一定的作用。接下来将详细介绍下我们的做法，以及过程中遇到的问题。

整体上 RL 模型将会依据输入数据，调整现有模型的某些重要的权重值。RL 模型的输入值包括了全网比价结果，以及城市粒度的默认排序相关统计维度。



首先，我们定义 agent 的 action 用于调整现有线性排序模型的参数 WBase_model。

$$w_{new} = w_{Base\_model} \times \alpha_i$$

其中，alpha 即为 agent 输出值。

接下来，我们构造 reward 函数为：

$$Maximize \sum_{c \in cities, i \in sequence} (CR\_experiment_{c,i} - CR\_base_{c,i})$$

$$=> Maximize \sum_{c \in cities, i} (CR\_advantage_{c,i})$$

即我们的优化目标为增加了 RL 模型的流量分桶，能够比原模型流量分桶的转化率（CR）更高，也就是最大化 CR 优势 $CR\_advantage_{c,i}$。

模型选择及参数设置。在方案 A 中，我们没有对算法选择和参数设置上做太多的调整。值得说明的一点是，我们将 DQN 中 Bellman Equation 更新部分的衰减参数 γ 设置为 0，意味这我们的优化的目标是一步达成更优的 CR，而不考虑多步马尔科夫决策过程（MDP）。

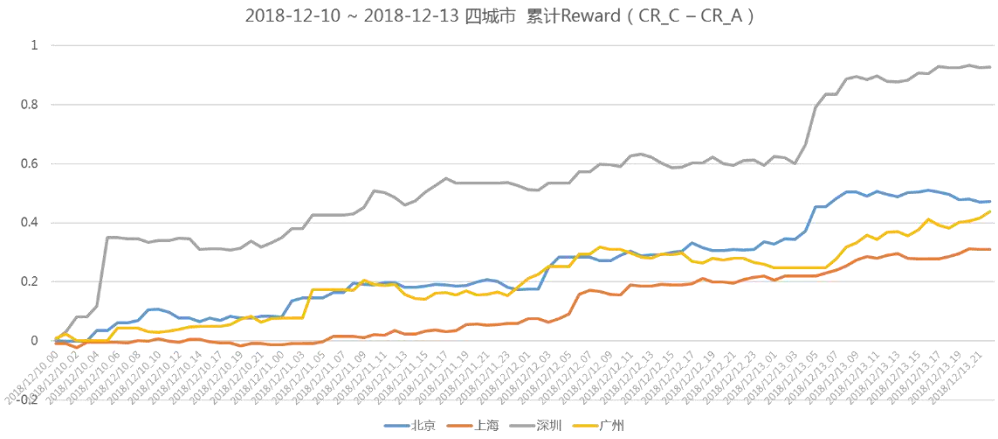
显然这是一个简化操作，但我们认为在城市粒度上，追求多步 MDP 意义不大，设置 γ=0 能够简化模型。当粒度细化到单独一个用户时，考虑 MDP 将更为有价值。

模型调整的粒度。我们前面讲到，方案 A 是按照城市粒度进行参数调整的。当 agent 对某个城市给出一个 action 后，经过一个时间片回收 reward 结果，作为模型的训练依据。考虑城市粒度，主要是为了平衡数据量和同步难度：如果将整个默认排序进行调整，RL 作用的价值有限；而若以更细粒度进行，数据实时同步又会比较困难。

更新时间片长度的设置。模型所需的 reward 反馈依赖于由 Kafka 给出的埋点记录数据，而当时是全业务埋点统一定时处理的，造成 reward 收集存在很大的时间间隔。此外，在模型给出 action 到最终在前端权重更新也存在较大延迟。

为了应对这个问题，我们增加了前端的消息同步，以确保 action 与回收的 reward 能够一一对应。但这并不能改善模型更新频次少、更新延迟的问题。我们原计划是每一个小时进行数据更新，但在极端情况下，模型更新间隔被拉长到了 3 个小时，对模型的效果形成了较大的影响。

最后，我们来看下线上实验的效果。我们选择了北上广深四个城市观察，发现这四个城市的累积 reward 虽有上下波动，但总趋势为正向增长。但同时也应注意，累积 reward 值绝对数值较小，对其评估还需要更多的实验。



六、总结

我们从传统 L2R 应对探索问题能力不足谈起，介绍了 RL 的背景以及其对这类平衡探索与收益问题的适应性。在此基础上，我们结合排序业务出发，制定了循序渐进的若干 RL 应用方案，以期能够利用 RL 的优点，满足排序业务的需求。

此外，对初步探索中我们的实践与碰到的问题做了详细的讨论，并在最后通过对线上结果实验的分析，说明了 RL 能够起到一定的作用，但还需要更进一步的应用和实验，以加强 RL 能够带来正向作用的结论。

接下来我们将首先对数据处理构架方面进行改进，利用流式处理构架，并尝试探索更为适合算法的以进一步释放 RL 的效能，从而实现更优的排序结果，进一步降低用户费力度、提升用户使用体验。

千人千面营销系统在携程金融支付的实践

【作者简介】房英明，携程支付中心数据负责人，目前负责支付离线数据仓库建设及 BI 业务需求。主要兴趣点为并行计算、数据处理及应用等领域。

一、引言

携程金融核心产品为：拿去花、借去花、信用卡、理财。其中拿去花提供携程产品分期支付服务，借去花提供现金借款服务，信用卡提供携程联名卡、理财则给用户提供有竞争力的理财产品。除此之外还有闪游卡、二维码、程金币等小的业务线。

如何把这些创新性的金融产品，推荐给有兴趣的潜在用户，成为一个需要解决的难题。在此背景下，支付中心数据组开发了一套用户精准营销系统。

二、系统设计目标

1）不侵入业务系统，与业务系统解耦

业务方只要提供符合规范的 restful 接口，即可接入系统。接口分为两类：一些强业务关联的规则，由业务研发实现；另外一些分析相关的接口由数据组负责完成实现。

2）灵活的营销页面投放方法

可以灵活进行分群及投放。投放策略分为：共性策略，个性化策略。共性策略配置实时生效，产品运营可以根据业务需要及时调整相关策略，无需数据/研发参与；而较为复杂的个性化策略可以由数据组基于画像指标进行扩展与定制，免去了漫长的开发、测试、发布等阶段，提高了投放效率。

3）支持多种投放模式

基于优先级、随机、模型等，其中模型效果可以通过 Abtest 量化效果以及迭代优化。

4）报表监控/离线数据采集

报表监控包括：实时流量使用、实时点击、离线点展转化、业务接入 api 性能监控；离线数据采集主要是为了追踪策略效果及转化。

为了方便其他业务方接入，数据组定义了一套输入/输出标准，业务方提供的 restful 接口只需满足输入输出的定义，即可接入本系统，这样接口开发与数据组策略制定可以解耦，运营可以针对每种接口进行热插拔式投放。

2.1 整体架构

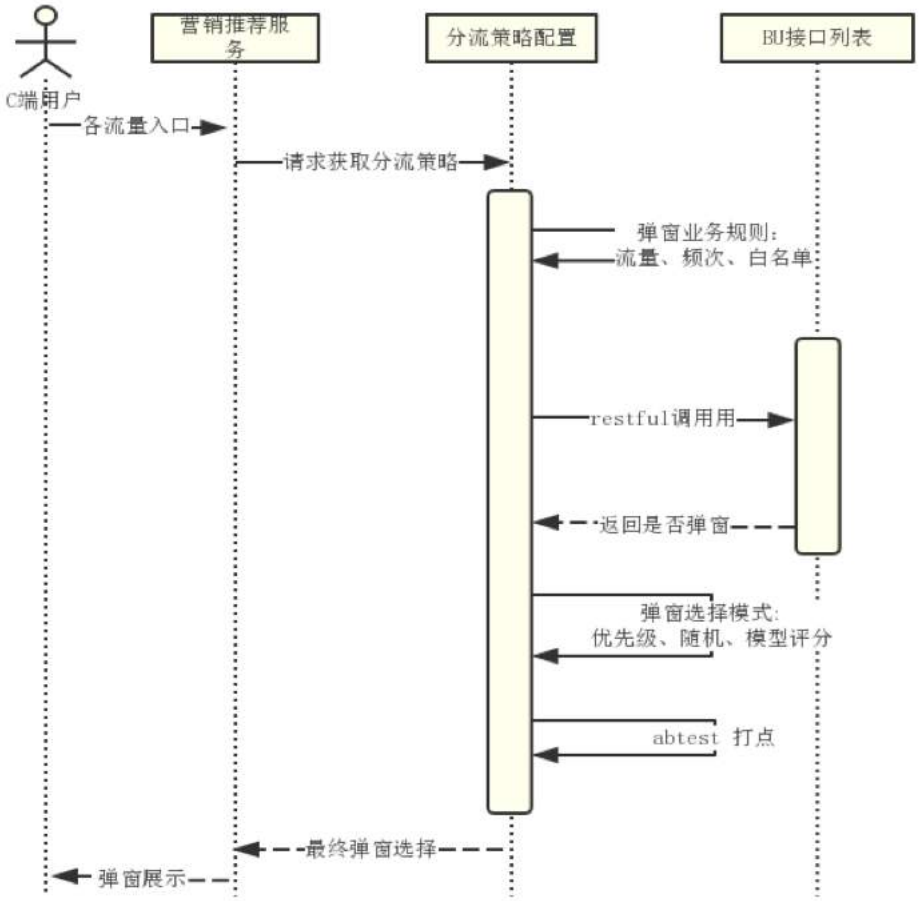
1）策略管理模块：数据组收集业务相关需求，沉淀相关的业务规则。目前核心功能是：借助分流策略提高流量利用率及使用模型评分提高开户转化率。

2）模型训练模块：根据不同业务线的特点，定义模型黑白样本，使用机器学习算法，对用户业务线的开通情况进行评分并进行标准化。

3）数据收集/分析模块：建立了完善的离线及实时监控，如实时流量使用、实时点击、离线流量/开通转化率数据；通过后端埋点，埋点日志落地至 hive 中，数据组基于日志分析各个步骤数据转化效果，进而优化策略。

2.2 交互时序

通过各接入业务方接口拦截是其营销页面展示的必要条件，为了提高接口的响应性能，采用了线程池对每种类型的弹窗分别遍历各自的规则，同时对于业务线接口返回超时时间进行限制，防止系统性能出现较大的真震荡。



三、核心功能

本系统关注两个方面：流量使用率和流量转化率，前者实现流量精细化运营，能够更加精细的控制每个产线所分配的流量；后者致力于提高流量转化，使流量能够最大化其价值，流量使用率主要通过分流实现，而流量转化率主要借助机器学习算法。

3.1 流量使用率-分流

分流的本质在于提高流量的利用率，在流量总量有限的情况下，如何根据一定的策略实现不同流量的分配，流量是通过百分比进行划分，所有产品线共享流量比例为 100%。当前支持的策略有：随机均匀、基于优先级、基于推荐算法实现。

3.1.1 随机均匀

均匀性：保证各业务线不仅能够在投放总量上保持均匀分配还要求在同一时点尽可能的均匀，不同时点的流量质量会有差异，此处可以借鉴各种随机化函数实现(random/Collections.shuffle())。

一致性：保证同一个用户多次进入都会显示相同的营销内容，常用的算法为 hash 取模，如

果对于并发度要求高，可以使用 murmurhash3。

3.1.2 预测当天流量

当一个用户能够通过多个业务线策略拦截时，随机选择一个业务线的营销页面进行投放。针对任一用户 uid，各业务线接口返回值会有不同，且不同业务线的拦截策略也会有所不同，最终能够通过业务线拦截的业务线个数会不同，故无法保证完全的均匀性和一致性。

如果仅仅借助于均匀性和一致性会导致部分产线的流量分配不合理，因此需要对各个业务线流量的上限进行严格控制。此时需要预测当天流量才能借助于当天流量总量及所分配比例，确定流量预算，考虑到携程 app 流量周期性影响，可以借助于时间序列模型去预测。

3.2 开通转化率-模型

3.2.1 模型 Y 值定义

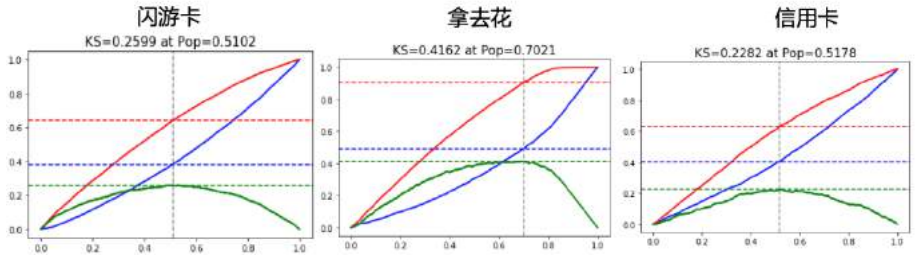
为了提高营销页面的转化率，我们借助机器学习算法，在选择模型的正负样本的时候，我们没有以是否点击作为模型的 target，而是以是否开通作为模型的 target，原因是我们通过离线数据分析发现，UED 重新设计页面，可能导致页面的点击率变动较大，当对于整体的开通转化率影响较小。

3.2.2 关于模型开发及部署

我们测试了多种业内算法，最终采用了 GBDT+LR，由于目标客户量大，而模型训练的特征也有近百个，且需要对每天的全量用户进行打分并推到生产环境，故直接使用 spark ML 在 hadoop 平台进行模型训练及部署。

模型效果如下：其中拿去花 KS 较高，是由于已激活的拿去花用户较多，能够拿到的特征相对比较全面。

弹窗类型	KS	AUC
闪游卡	0.2599	0.6755
拿去花	0.4162	0.7735
信用卡	0.2282	0.6361



3.2.3 评分的标准化

由于大部分产线都会单独进行模型评分，各自评分的度量需要映射到同一个基线上进行评分才合理，这个统一基准包括：开通转化率的测算，然而不同业务线的开通转化率差异也会很大，我们对评分进行分段，对不同分段的开通转化率进行测算，最终根据测算结果选择阈值，且利用最大最小值标准化不同产线的评分。

四、策略制定需要关注的问题及解决方案

4.1 流量预算少时模型阈值的选择

倘若某个业务线所分配的流量占比非常少时，是否可以大幅度提高其转化率？也就是尽量选择高质量的流量进行曝光，防止流量过早在相对较低质量时段中耗尽，此时如果该业务线上了模型，可以将模型分值调整到一个较大值，从而保证其转化率。

4.2 业务线共存

各产线营销页面投放需求差异较大，部分营销页面投放周期较短，或不以“开通”为目的时，可能就不需要为这种类型的业务线训练模型，而营销页面投放选择时会遇到：拥有模型业务线和不拥模型业务线共存问题，此时要合并这两种类型的营销页面，进行页面随机均匀选择。

4.3 新业务线的接入

新接入的业务线由于没有样本，需要事先分配部分流量，待样本积累一段时间后，才会进行模型评分。

4.4 模型效果量化-abtest

理论上一个新的用户端产品上线前都应该进行有效的 abtest，然后基于 T/Z 检验 abtest 产品效果，实际上 abtest 效果评估也需要根据业务进行调整。

如在一般的信用类场景中，使用 abtest 进行效果评估会很不合适，因为用户变坏的过程较为缓慢，且策略变更也较为频繁，而效果却难以短时间内衡量，具体到本系统，定义好目标值，如开通转化率，在保证 AB 样本接近 1:1 时，很容易可视化模型效果，无需 T/Z 检验。

4.5 如何构建近似 1：1 的 abtest 流量

Abtest 衡量转化率时，需要尽量保证 A 流量和 B 流量近似于 1：1，这需要对 A、B 样本进行埋点，然后线下分析，进而调整各业务线的模型分值，间接影响 AB 流量分配。

五、总结

本营销系统，基于携程用户信息及机器学习算法，灵活进行分群及推荐，解决了流量利用率和转化率难点，且能够通过 abtest 量化/迭代推荐算法效果，业务线接入简单，支持运营产品实时调整营销策略。

数据同学能够基于收集的完整后端/UBT 日志，跟踪分析营销系统每一步的转化情况，然后进行策略调整及优化投放。

数据质量良莠不齐？携程是这样来做多场景下的内容智能发现的

【作者简介】朱登龙，携程 AI 研发部高级算法工程师。负责 NLP 内容化的相关工作，主要专注领域为文本分类，文本抽取，文本生成，文本内容信息挖掘等。协同完成多场景智能内容抽取和生成项目，并交付多业务线的不同场景使用。

一、背景

目前业界内容化的应用场景愈发丰富，大家试图通过特色化的内容来吸引用户，引导用户与产品增加交互（浏览、点击、购买等）。虽然各大互联网公司的内容数据已足够丰富，但数据质量良莠掺杂，难以直接用于内容化场景落地，因此如何实现优质内容的发现、抽取和生成，便成为重要的技术课题。

本文将从多场景（短亮点、长推荐理由、正式语句表达）和多纬度（主题/产品特色）角度，分享携程在智能内容抽取和生成中的技术实践。

一般而言，内容智能发现可以分为抽取式和生成式两种方法，其中抽取式是指从数据源数据中抽取高质量的语句并且不会改变语句的原始结构；而生成式是指利用深度学习基于 seq2seq 的方法，根据相应的语料，进行语句生成，生成的是新的语句。

内容智能发现的价值在于：

1）充分发挥算法价值，借助算法自动快速的从自然语言数据中发现和生成高质量的推荐理由语句，结合业务和场景展示给用户，帮助用户快速了解该产品其他用户的评价内容并以此做出自己的判断。

2）帮助运营人员进行内容化工作，大大节约人力成本，节省时间。

这些推荐结果的应用场景也较多，如产品展示页，评论弹幕，产品详情页，副标题等等。随着内容化的重要性越来越大，可用的场景也会越来越多。

二、携程马可波罗中台

马可波罗中台是携程 AI 研发部自主研发的集 AI 算法能力于一身，并适应多场景需求，对外提供 AI 服务的人工智能服务平台，如图 1。

该中台承载着内容化的主要功能。

架构上，该中台分为数据层，算法层和应用层：其中数据层是指中台灵活对接公司的 50 多个数据源，并且支持动态调整，这些数据量级达到了 10 亿+；算法层是整个平台的大脑，集成各种算法模块，主要为情感模型、实体识别、图片算法等等；将业务和算法层的组合就能

够完成应用层的实现，现在的应用层主要是实现 NLP 内容化的工作和图片视觉等方面的工作。

将马可波罗中台的应用层进行具体划分，又可以分为底层计算、特色挖掘、图片部分和主题内容化四个部分。考虑到中台的数据量较大，因此整个算法都是基于 spark 等大数据平台运算；产品特色挖掘方面是结合知识图谱和实体识别建立各个产品的特色部分供业务使用；而图片部分主要是视觉方面的工作包含图片分类、图片搜索、优美度判断等等应用；最后的智能内容发现是指平台根据用户设置的主题利用算法在用户选择的数据源范围内，进行内容发现。

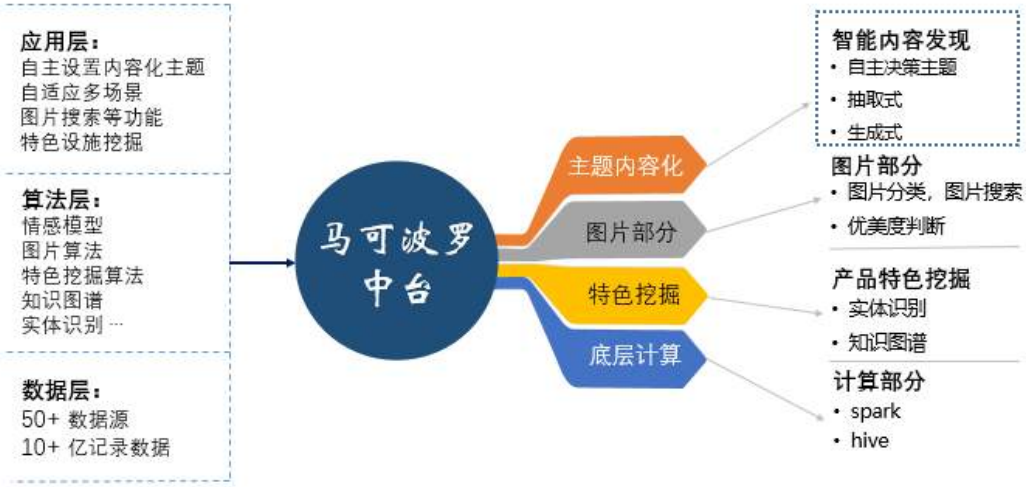


图 1 马可波罗中台

三、智能内容抽取

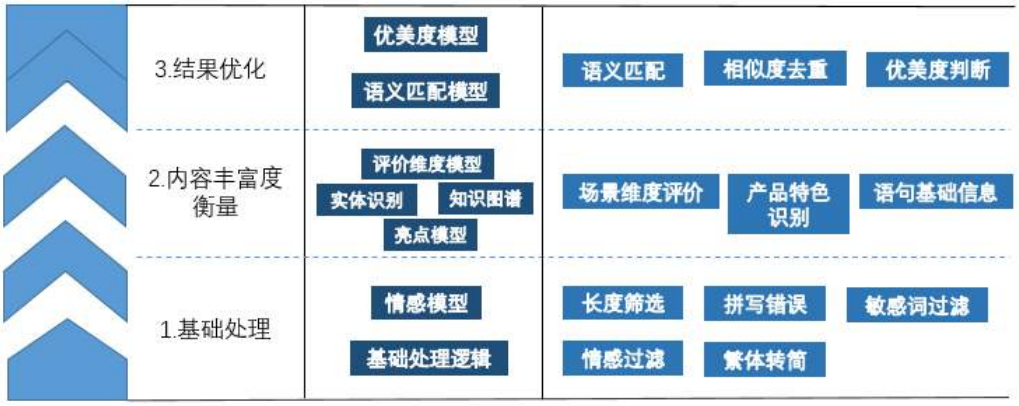


图 2 智能内容抽取示意图

根据图 2 所示，抽取方式的流程涉及到的模块比较多，主要可以分为三个阶段：

1）基础梳理；
2）内容丰富度衡量；

3）结果优化三部分；

预处理阶段的目的是对语句进行比较基础的处理，主要包含情感过滤，敏感词检测，拼写错误等等，这些模块是为了保证语句基本在表达上是规范的，情感倾向是满足需求的并且不会触犯法律法规问题的。

内容丰富度模块是较为核心的模块，内容丰富度的程度会直接影响改句子的得分，进而直接影响这句话是否被召回，经过统计和思考我们确立了三个层次内容丰富度衡量体系，即分别从产品维度、产品特色及语句信息含量来衡量句子的内容丰富度，而用到的模型及算法包含实体识别、类别评价维度模型、知识图谱等等。

结果优化处理模块，是对初步召回的结果进行优化处理如去重，语义匹配，优美度判断等等。

在抽取过程中我们同业务充分沟通，设计这些模块计算法能够动态匹配相应的产品，因此基本上能够满足不同产品和场景的抽取要求，并且在最终的结果处理方面，能够根据业务需求进行多样性的结果优化。

3.1 预处理过程

在预处理过程中，比较重要的是情感检测模块。在不同的业务场景下，对结果语句的情感要求也随之变化。因此我们构建情感模型来掌握和控制语句的情感倾向，来匹配结果的需求，满足不同业务的需求。

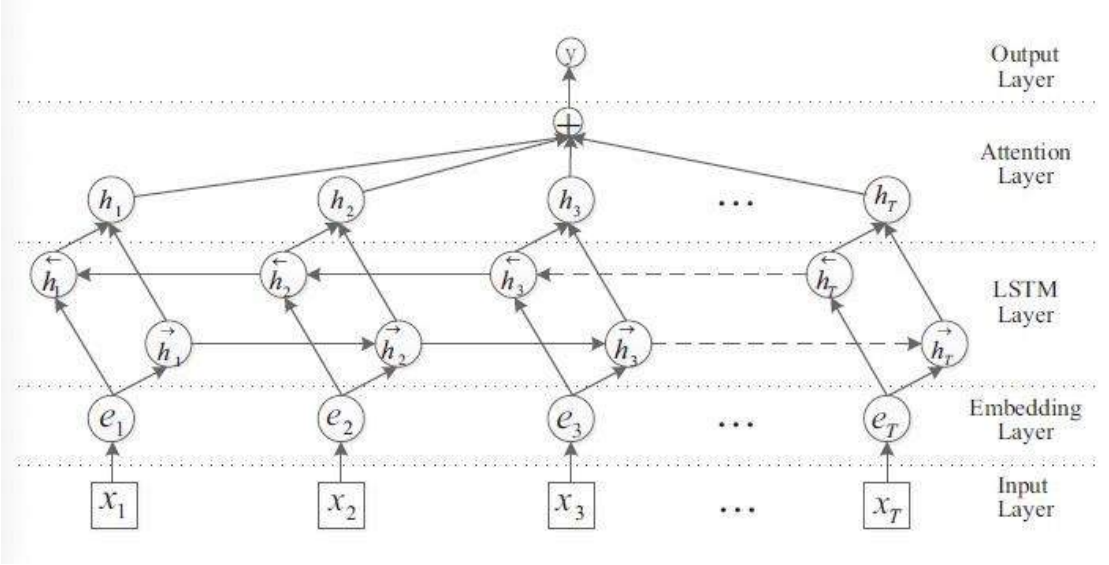


图 3 Bllstm 示意图

在构建情感模型上面，我们一开始使用的是传统的机器学习 tfidf 抽取文本的特征，使用 chi2 筛选特征，进而使用 svm 作为分类器，来解决情感分类问题。

当数据量变大以后，我们采用了现在较为流行的深度学习 W2C+LSTM+attenion 的框架来搭建情感模型。bilstm 结构图如图 3 Bllstm 示意图所示，不同于传统的 lstm，该网络结构分别

从前到后和从后到前进行 lstm，这样的结果等同于同时考虑了词语前向和后向的关系。

而 Long ShortTerm 网络，一般就叫做 LSTM ，是一种 RNN 特殊的类型，可以学习长期依赖信息。这个模型非常的深，12 层，并不宽(wide)，中间层只有 1024，而之前的 Transformer 模型中间层有 2048，深而窄比浅而宽 的模型更好。

MLM（Masked Language Model），transformer 同时利用左侧和右侧的词语，15%单词进行 Mask（遮挡）技术在语言模型上应用。在 encoder 端输入三种 embedding 如图 4 所示，另外在进行分类模型时候，只是用结果的第一个向量如图 5 中的 C。

1. Token embedding 表示当前词的embedding
2. Segment Embedding 表示当前词所在句子的index embedding
3. Position Embedding 表示当前词所在位置的index embedding

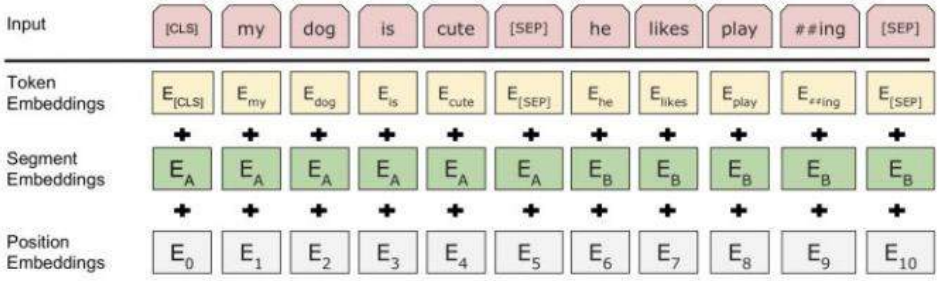


图 4 bert embedding

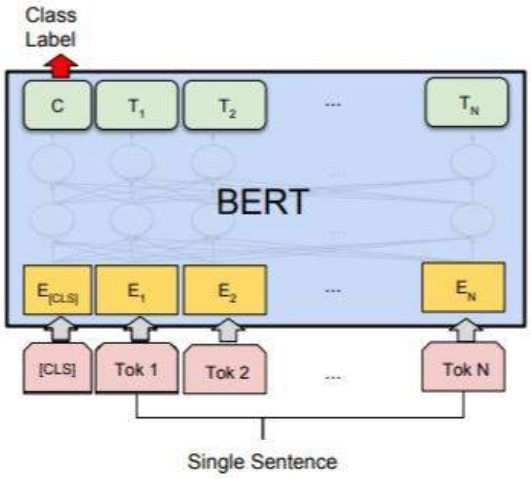


图 5 bert 分类模型输出

在分类模型选择上，我们分别调研了：

1）Tfidf+Chi2+SVM；
2）Lstm+Attention，CNN 等深度学习方法；
3）Bert 的方法；

不同方法的优劣点具体如下：

对于方法 1，使用传统机器学习的方法，使用 TFIDF 来构建文本特征，使用 Chi2 的方法进行特征筛选（feature selection），在对比了随机森林、boost、svm 众多的机器学习方法后，最终发现 SVM 的效果较好。该方法的优点是即使样本量比较少（小于 5000），效果依然比较好，正向的召回和准确率均在 90%以上。总体来讲，传统机器学习的方法，便于实现、解释性强。另一方面我们的数据很多是离线处理的，数据量很大（上亿级别），因此要使用大数据技术，结合 spark 进行大批量的运行。在结合 spark 过程中，传统机器学习的优势比较明显，方便部署和并且批量处理的时间能够接收。

对于方法 2，在深度学习方面调研了 CNN、LSTM、RCNN、fasttext 主要的四种编码方法，从结果看 lstm+attention 的效果最好，RCNN 效果次之，并且当数据量大于 1.5W 后 lstm+attention 能够超过传统机器学习方法。但是 RCNN 的训练特别耗时，因此后期主要选择 LSTM+attention 的方法。在将深度学习和 spark 结合过程中，通过优化 spark 后，深度学习的方法在效率和准确度上也能达到令人满意的效果。

对于方法 3，考虑到 bert 在很多数据及上表现出来的碾压式的优势，我们也调研了 bert 的分类能力，使用同样的数据量，bert 的正确率大幅度高于前两类方法高出 3-4 个百分点。但是和深度学习一样，难以部署，尝试过结合 spark 部署，但是速度奇慢，也只作为调研对象。

经过对比较结果图 1-6 所示，考虑到 bert 的优异表现，最终我们选用 bert 的方法。

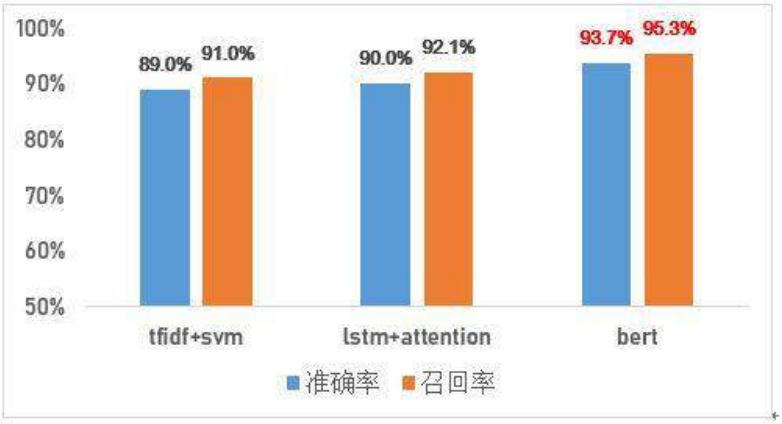


图 6 结果对比

3.2 内容丰富度

传统衡量文本内容有很多方法，长度，词性，句法分析，交叉熵等等都能衡量。但是衡量结果并不能同时满足满足场景需要及满足运营需要，因此需要考虑增加更多的维度、更多新的方法来实现内容丰富度衡量。

具体参看图 7，我们建立了三层体系的内容丰富度衡量的评价体系，即分别从语句层面、产品层面和场景层面较为完备的衡量内容丰富度情况。

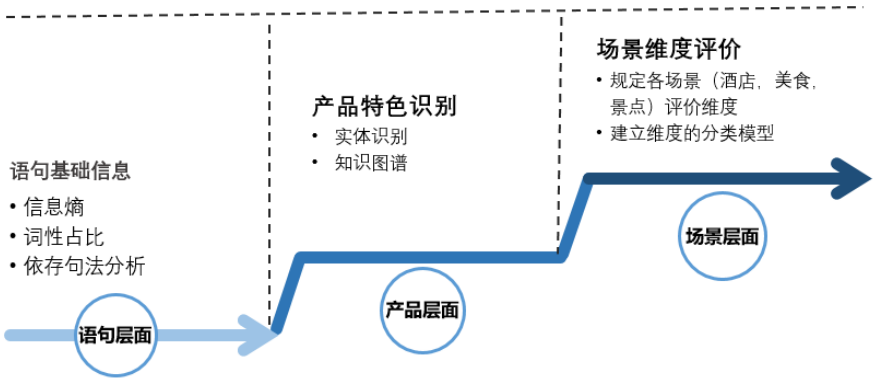


图 7 内容丰富度评价体系

3.2.1 内容信息度

$$H_s = \sum_{i=1}^{n} p_i I_e = -\sum_{i=1}^{n} p_i \log_2 p_i$$

图 8 信息熵

该模块分别从信息熵图 8 所示，基于统计方法的词性结果和句法依存关系占比三方面来衡量文本基础信息度。信息熵经常被用来作为衡本信息度的指标，而分词后文本不同词性的占比也能从一定程度反应该文本的信息度。

从推荐理由的目的出发，我们期望抽取的句子含有一定比例词性的词语，并且也能够包含一定的依存关系情况。考虑到推荐理由的结果大概率的同时包含名词和形容词，并且包含一定的句子结构。因此考虑分词后词性和依存关系分析和计算内容表征分数。

其中词性重点考虑名词（n），形容词（adj），副词（adv），并且给与同时含有名词和形容词的词高的权重；依存关系方面，重点考虑 'ATT', 'ADV', 'COO', 'POB', 'RAD', 'LAD', 'VOB'等结构。

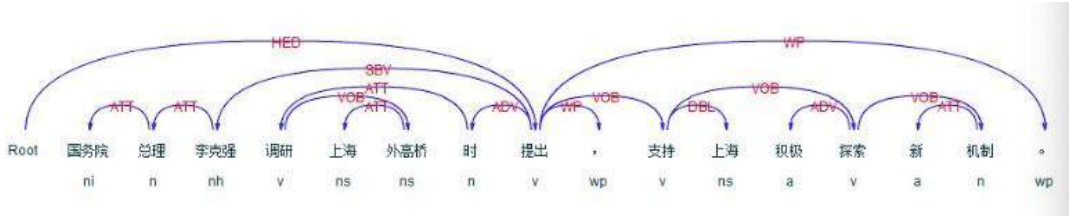


图 9 句法依存关系

3.2.2 产品特征

在产品特征方面，我们结合知识图谱和实体识别两方面来进行综合抽取。具体流程是一方面同业务商议各产品类别的实体词的确定、数据标注和模型训练；另一方面结合知识图谱来判断语句中是否存在当前产品的知识图谱中的特征属性。

具体流程如图 10 所示，实体识别和知识图谱的综合使用能够全面的获取语句中包含的产品层面的特征信息。

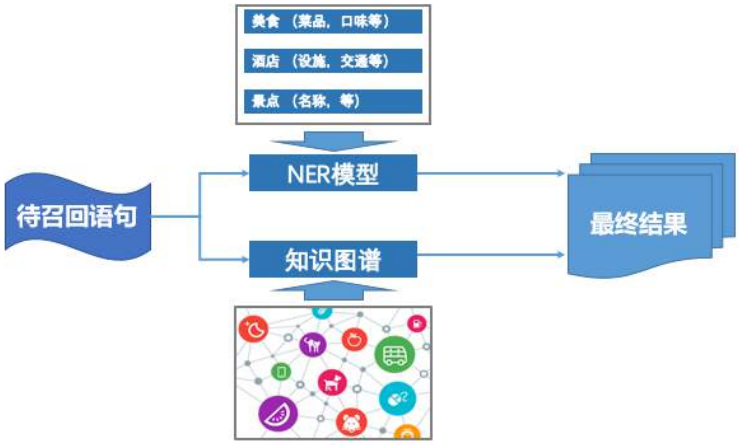


图 10 产品特征发现逻辑

目前 biLSTM+CRF 的方法成为基于深度学习的 NER 方法中的最主流模型。经过 bilstm 抽取后的特征，传入 crf 层，然后 Crf 使用状态转移矩阵和动态规划的思想进行求解，效果比较好。具体的知识图谱和实体识别相结合的案例如图 11 所示。

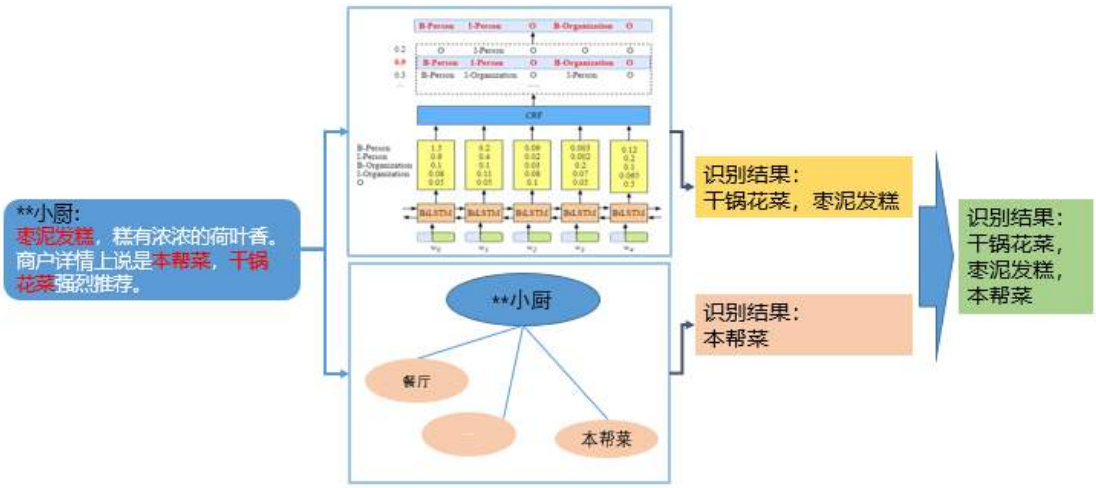


图 11 产品特征发现逻辑

3.2.3 类别维度评价

该流程主要包含两个组成部分——维度评价模型和亮点模型，目的是保证抽取的结果具有明显的表达某场景某一方面的情况，并且该语句有着一定的推荐倾向。

类别维度评价模块是产品特征维度更高层次的内容，并且需要分别结合不同的产品类别制定维度特征，利用语句在产品类别维度的得分来反映文本内容丰富度的模块。

该模块的出发点为，考虑用户希望看到的推荐理由应该是反映该产品某一方面的评价信息，因此先根据不同的类别的产品（酒店，餐饮，美食）明确各大类别的主题维度，而后进行数据标注及模型训练。在预测期间将取最高的维度得分（各类 softmax 后）作为该语句在该模块上的得分，具体流程见图 12。

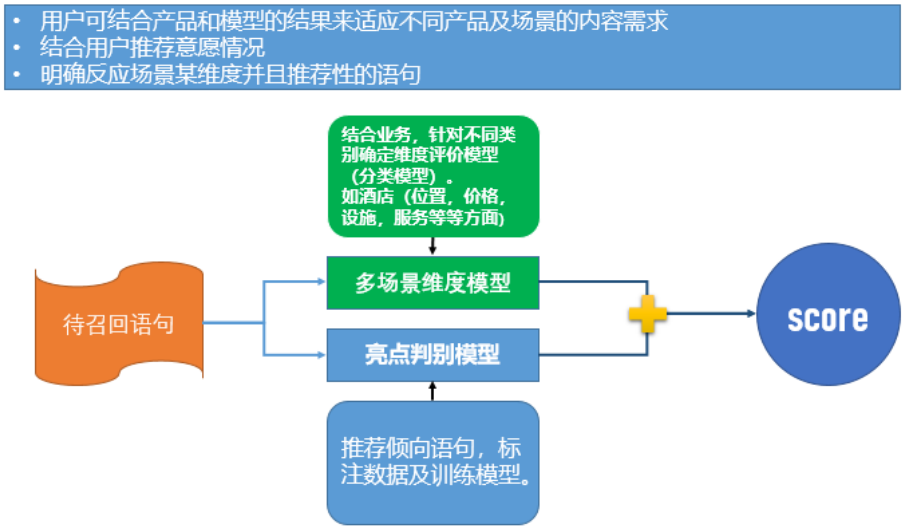


图 12 类别维度评价

3.2.4 效果展示

经过上述处理后的结果展示如下，以“和平饭店”抽取结果前后对比（左前右后）：

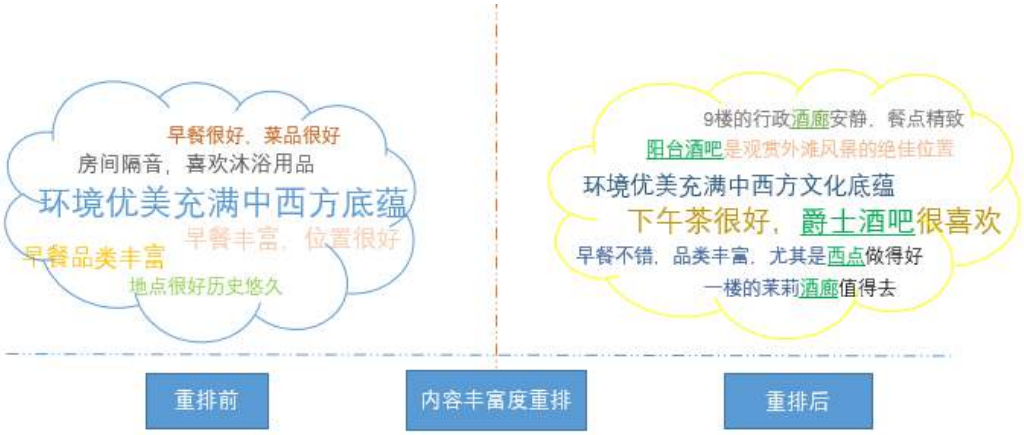


图 13 “和平饭店”效果展示

从图 13 结果看出，加入该逻辑后，能够避免抽取单一的例如早餐丰富的语句，更能够抽取出产品自身的特点的语句，如阳台酒吧、爵士酒吧、酒廊等等实体词汇。

3.3 语义匹配

在 AI 挖掘平台上，用户抽取推荐理由是基于主题语境的的，即用户给出一些列的关键词，

然后算法依据这些关键词对候选语料进行召回。因此需要模型来判断主题和结果的语义匹配度，进而对结果根据语义匹配度进行重排。

在做语义匹配模型时候我们分两步进行，在缺少标注数据时候首先使用无监督的方法，直接计算平均词向量的余弦值作为匹配分值。在阶段二，利用标注数据训练了匹配模型，将匹配模型的得分作为最终的匹配分数。

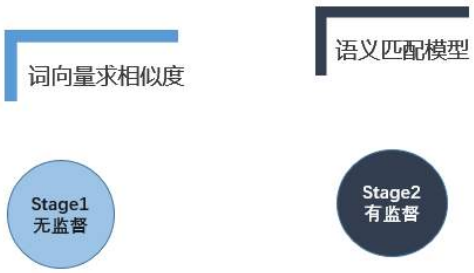


图 14 语义匹配的两个阶段

目前匹配模型从 q,a（q 是问句，a 是回答语句）分别抽取特征后到计算相似度之间是否反生交互过程，分为并行方式，交互方式两类；而按照 loss 函数设计又分为 pointwise 和 pairwise 方法。

在并行方式中 q,a 一般使用同样的网络抽取特征，但是两者并不相互产生关系；而在交互方式中，两者会进行一些计算比如 ATTENTION，pooling 等然后计算相似度。

pointwise 和 pairwise 的主要区别在于 loss 的设计，前者会将问题转化为 2 分类问题，loss 函数使用交叉熵，模型的结果是判断两个文本是否匹配；后者一条训练记录包含 q 和匹配语句 a+和不匹配的语句 a-，然后分别计算相似度 sim，损失函数为 loss = max{(margin+sim（q,a+）-sim(q,a-))，0}。

这 loss 的具体意义是令匹配的得分高，不匹配的得分尽可能低，margin 表示超出这个区间就认为能够区分匹配与不匹配的界限。从结果可以看到如果两者差值超过这个 margin，loss 为 0 训练停止。另外其中 SIM 为相似值，一般会用 cos 计算距离。

在实践中我们分别根据图 15 和图 16 使用 lstm+attention 和 CNN 方法搭建匹配模型框架，然后根据图 17 的网络结构将两阶段拼接，最后同时训练两者的 loss。

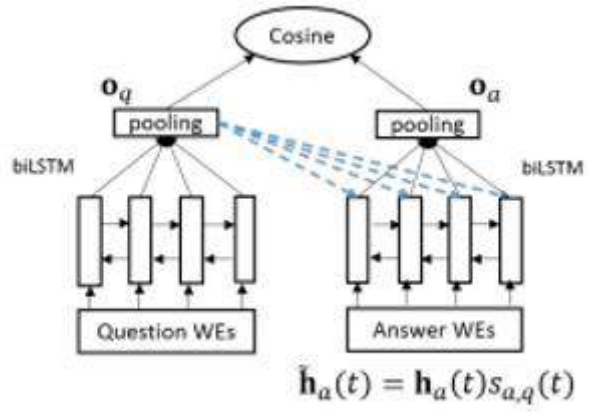


图 15 语义匹配 lstm+attention

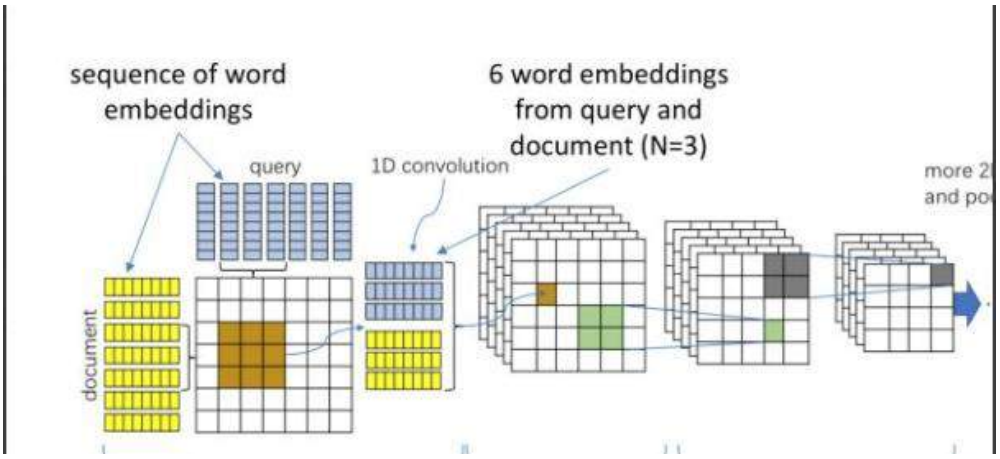


图 16 语义匹配-cnn

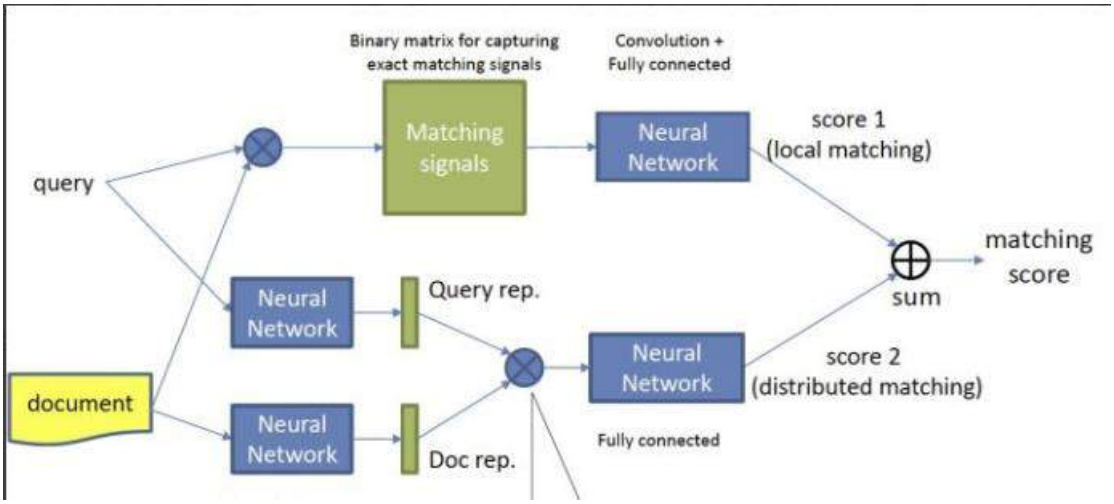


图 17 语义匹配完整网络结构

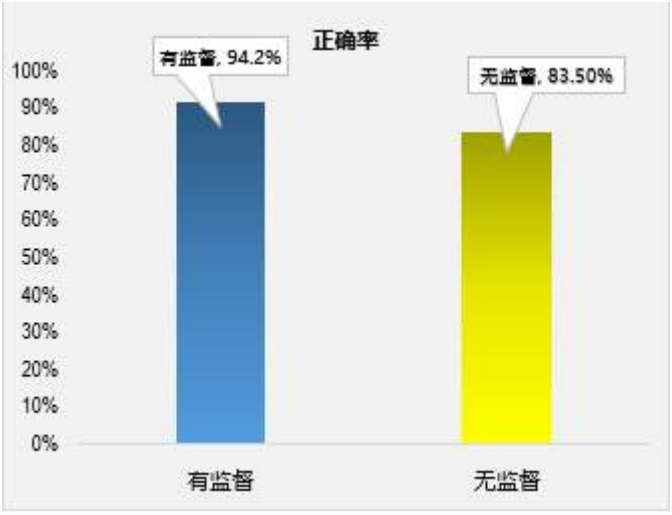


图 18 语义匹配重排效果图

我们使用 1000 个匹配的 q,a，然后每个匹配的 q,a 随机增加 3-4 条不匹配的数据。评判标准为分别使用监督学习和费监督学习的方法为每个 q,a 计算匹配得分，判断最高得分的 q,a 是否为真是匹配对。最终的结果表明，监督学习的方法正确率为 94.2%，远高于费监督学习的正确率 83.5%。

四、推荐理由生成

4.1 copynet

随着深度学习在 NLP 上的大放异彩，我们也基于深度学习的 seq2seq 框架在文本生成方面做了一些探索。因为仅仅做抽取的话，结果就只能从原始语料产生，因此结果太过生硬，并且可能出现语法问题。

NLP 深度学习的发展特别是 seq2seq 的发展，让文本生成成为了较为容易实现的事情。因此我们也希望能够利用生成的方法克服抽取的缺点，为了保证一定的限制性的输出，才用了 copynet 的方法。翻阅受限生成的方法，其实在 encoder 的改动不大，最多是增加一些新的信息或者机制进行 encoder，主要的改动都在 decoder 端，在生成部分倾向于某些词汇进行生成，接下来借助 copynet 讲解。

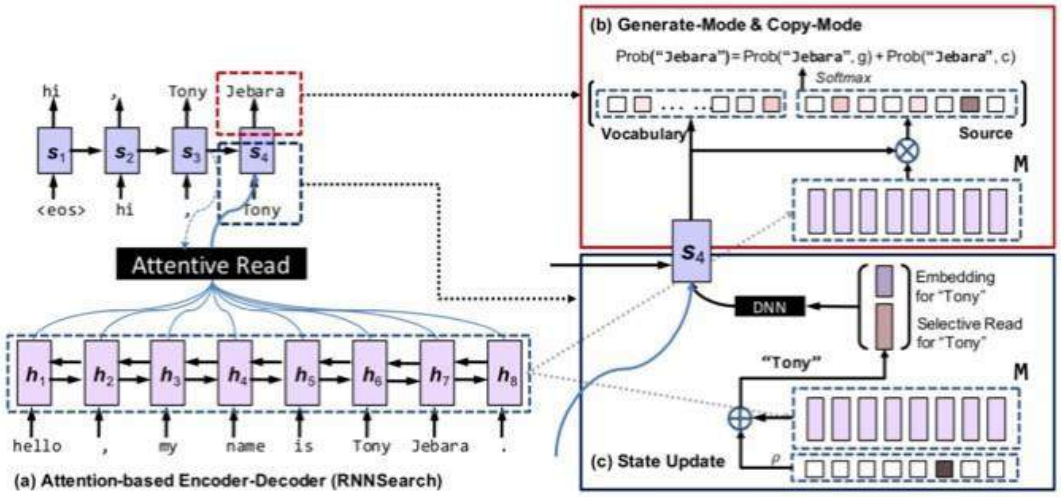


图 19　copynet 结构图

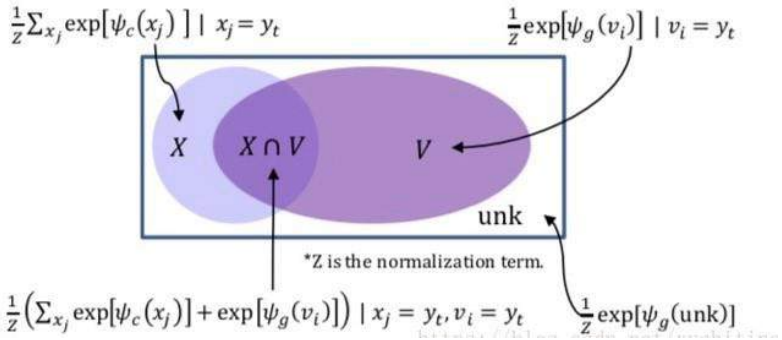


图 20 结果概率公式

状态更新

decoder状态更新的公式是

$$\mathbf{s}_t = f(y_{t-1}, \mathbf{s}_{t-1}, \mathbf{c})$$
$$p(y_t|y_{<t}, X) = g(y_{t-1}, \mathbf{s}_t, \mathbf{c})$$

不同的是这里的t-1时刻的y由下式表示：

$$[\mathbf{e}(y_{t-1}); \zeta(y_{t-1})]^\top$$

后面的部分是M矩阵中与t时刻y相关的状态权重之和，如下：

$$\zeta(y_{t-1}) = \sum_{\tau=1}^{T_S} \rho_{t\tau} \mathbf{h}_\tau$$
$$\rho_{t\tau} = \begin{cases} \frac{1}{K} p(x_\tau, \mathbf{c}|\mathbf{s}_{t-1}, \mathbf{M}), & x_\tau = y_{t-1} \\ 0 & \text{otherwise} \end{cases}$$

图 21 状态更新

4.2 TA-seq2seq

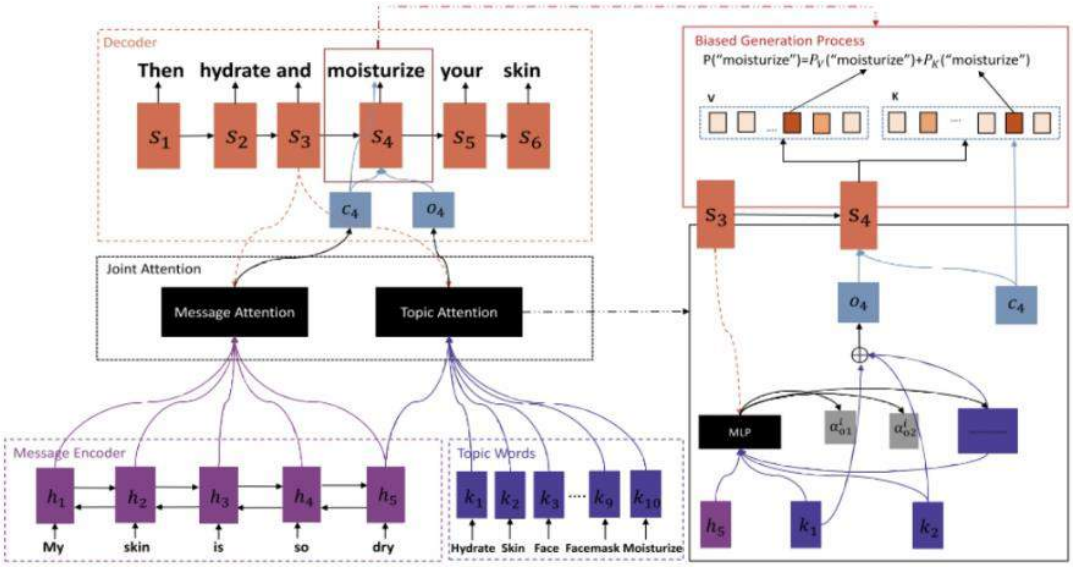


图 22 TA-seq2seq 网络结构

该算法全名为 Topic Aware Neural ResponseGeneration，基本思想是引入主题词来影响结果的生成，让生成结果更有"营养"。具体做法是在 encoder 端不仅输入常规的语句，还输入一些列的主题关键词，这些关键词是通过 LDA 模型得来，这两部分都需要和 decoder 每一步状态做 attention；并且再预测端设计两个概率函数进预测，一个是常规高频词，一个是主题词组成，而两者的结果作为最后的概率。

具体的改变如下图所示：

decoding 时，每一步时隐向量 h 被转化为 ci：

$$\mathbf{c}_i = \Sigma_{j=1}^{T} \alpha_{ij} \mathbf{h}_j,$$

主题向量 k 也被线性组合为 oi，每个 kj 对应的系数为：

$$\alpha_{oj}^{i} = \frac{exp(\eta_o(\mathbf{s}_{i-1}, \mathbf{k}_j, \mathbf{h}_T))}{\Sigma_{j'=1}^{n} exp(\eta_o(\mathbf{s}_{i-1}, \mathbf{k}_{j'}, \mathbf{h}_T))}.$$

图 23 decoder 端联合 attention

p(yi) = pV (yi) + pK (yi),

$$p_V(y_i = w) = \begin{cases} \frac{1}{Z} e^{\Psi_V(\mathbf{s}_i, \mathbf{y}_{i-1}, w)}, & w \in \mathbf{V} \cup \mathbf{K} \\ 0, & w \notin \mathbf{V} \cup \mathbf{K} \end{cases}$$

$$p_K(y_i = w) = \begin{cases} \frac{1}{Z} e^{\Psi_K(\mathbf{s}_i, \mathbf{y}_{i-1}, \mathbf{c}_i, w)}, & w \in \mathbf{K} \\ 0, & w \notin \mathbf{K} \end{cases}$$

$$\mathbf{s}_i = f(y_{i-1}, \mathbf{s}_{i-1}, \mathbf{c}_i, \mathbf{o}_i).$$

其中 V 是回复词汇表，f 是一个 GRU 模型， Z 是正规化因子，并且：

$$\Psi_V(\mathbf{s}_i, y_{i-1}, w) = \sigma(\mathbf{w}^T(\mathbf{W}_V^s \cdot \mathbf{s}_i + \mathbf{W}_V^y \cdot y_{i-1} + \mathbf{b}_V)),$$

$$\Psi_K(\mathbf{s}_i, y_{i-1}, \mathbf{c}_i, w) = \sigma(\mathbf{w}^T(\mathbf{W}_K^s \cdot \mathbf{s}_i + \mathbf{W}_K^y \cdot y_{i-1} + \mathbf{W}_K^c \cdot \mathbf{c}_i + \mathbf{b}_K)).$$

图 24 概率函数

优点：

1）引入了 topic 的影响，提高指定词的生成概率；

2）topic attention 利用 topic words 的状态信息和 input message 的最终状态作为额外的输入来减弱不相关主题词并加强相关主题词的概率；

3）在最后对话生成的过程中，采取了有偏于 topic words 的生成概率来增大 topic words 的出现几率；

4）不同于传统的 seq2seq 模型，在生成对话的第一个单词时，采用了两者组合，可以产生更为准确的第一个词，来给后续的词及整个句子的产生带来了更好的效果，因为后续词的产生会依赖于前面生成的词。

五、场景展示

经过上述的处理经过，通过算法模型抽取/生成的推荐理由主要的上线场景如图 25 所示。主要是四个场景——酒店首页二屏，酒店短亮点轮播，餐厅的推荐理由和 IM+的酒店推荐理由。通过算法和模型能够帮助产品特点的露出，吸引消费者的停留和转化，并且也大大的降低了运营人员的工作和运营周期。

酒店 – 首页二屏　　酒店 - 推荐短亮点轮播　　餐厅 - 推荐理由　　IM+ - 酒店推荐理由

图 25 线上应用场景效果图

六、总结

通过上文分享的方法及流程，我们完成了多场景的抽取及生成工作，并且在诸多的场景中上线，取得了较好的效果。通过算法和模型能够帮助产品特点的露出，吸引消费者的停留和转化，并且也大大的降低了运营人员的工作和运营周期。

但是同时实践中也存在一些不足：

1）结果中还是存在很多的比较单一的语句，例如在抽取部分存在例如：早餐丰富，孩子喜欢等缺乏细节或者特色的语句；在生成部分还存在句子句法错误的问题。
2）缺少用户 CTR（点击率）数据，所有的流程和算法的结果验证主要是依赖业务及运营人员检验，因此不能真实了解用户的喜好和意图。
3）模型更新问题，大量的数据还是希望能使用更加深层的模型，但是现在又因为 spark 结合，整个项目较大，大的模型或造成内存溢出或者效率不能满足生产需要的情况，暂时还不能解决这个问题。
4）抽取和生成结合：后面的方向需要生成方式的探索。并且还需要合理的将生成结果和抽取结果进行融合。

面对现有的不足，我们在实践中也会重点在这几方面进行探索和突破。

携程度假智能客服机器人背后是这么玩的

【作者简介】雷蕾，携程度假研发部资深算法工程师，负责智能客服算法工作。鞠剑勋，携程度假研发部算法经理，负责智能客服、知识图谱、NLP 算法等工作。

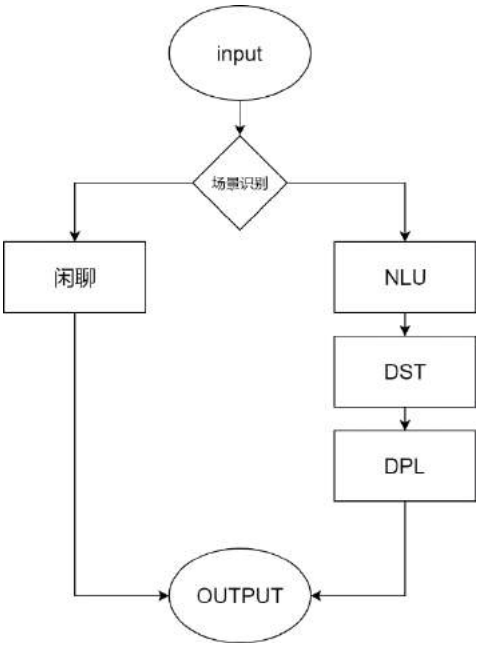
随着人工智能的发展，人机交互技术愈发成熟，应用场景也越来越多。智能客服是人机交互在客服领域的一个应用，服务于客人以及相关的客服人员。本文将介绍智能客服在旅游场景下的主要技术和应用。

当前度假的智能客服主要用于 C 端（客户端）面向客人，以及客服端辅助客服的两个角色。

面向客服端的是智能客服助手，用于对话窗口的侧边栏，提示客服人员当前客人问题的答案，客服人员可视情况来采纳；而面向 C 端的智能客服则是直接服务于客人，回答客人问题。

智能客服又分为单轮问答的 QA Bot 和多轮对话的 Task Bot，在携程的旅游场景下，以多轮对话的 Task Bot 居多。一般多轮对话的智能客服系统会切分为以下几个模块：客人的问题（Query）进来后首先经过 NLU 模块抽象化为客人的意图（intent）以及关键信息槽位（slot），意图及槽位传给 DM 模块后，经过 DST、DPL、NLG 模块返回答案。

- NLU（Nature Language Understand 自然语言理解），通过模型或规则的方式获取客人的意图和槽位；
- DST（Dialog State Tracking 对话状态追踪）存储对话状态，包括每一轮对话的意图以及已经抽取到的槽位信息、历史机器人的行为；
- DPL（Dialogue Policy Learning 对话策略选取），DPL 根据 DST 传输的内容决策机器人在该轮的行为；

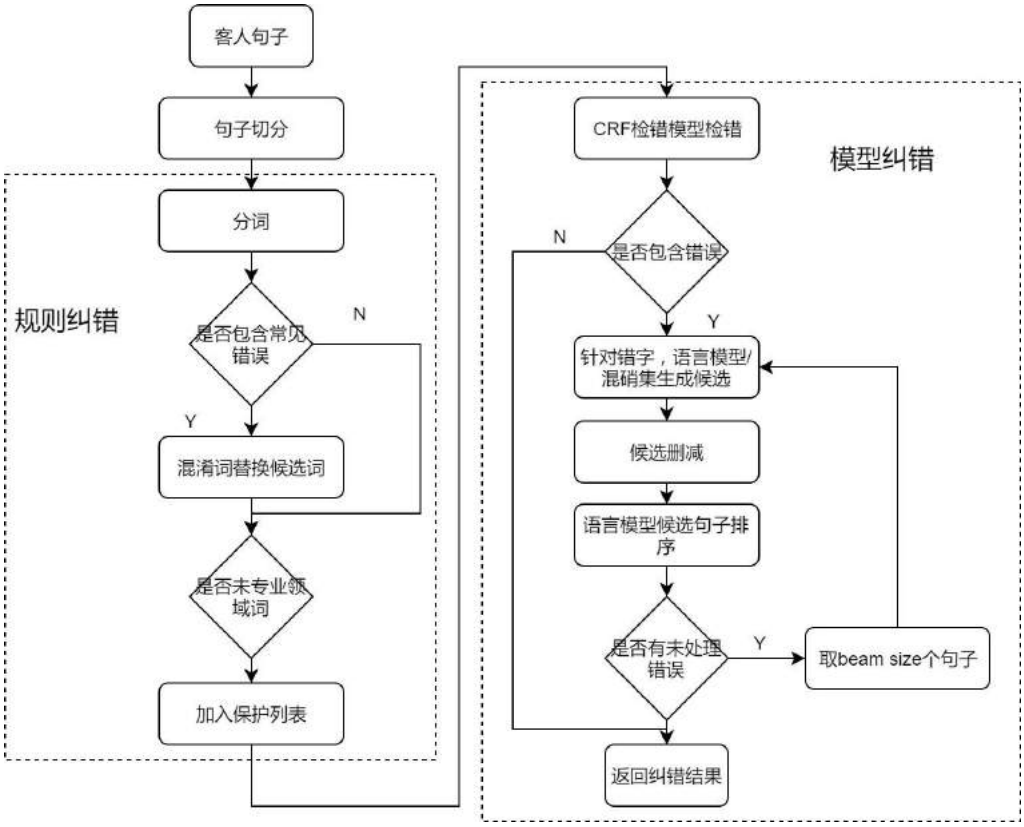


一、NLU

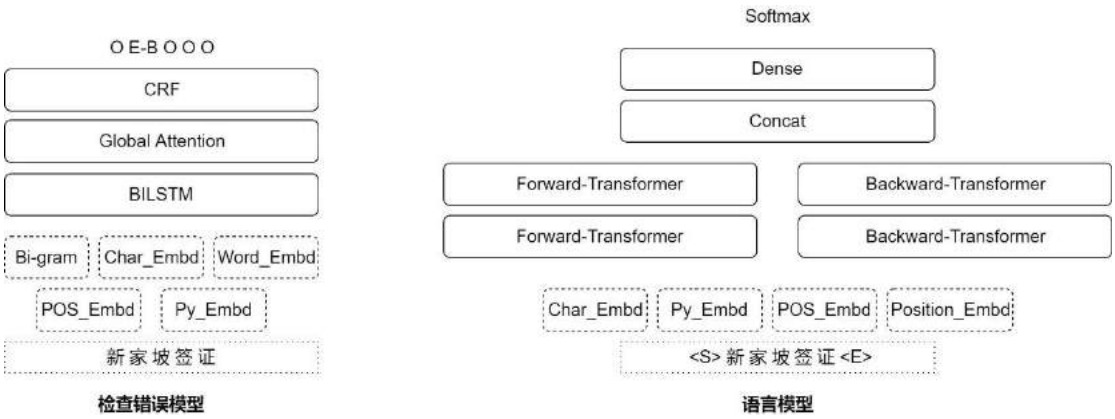
NLU 模块基础功能是获取客人的问题的意图及槽位信息，在业务比较复杂的场景，相对应客人可能问的问题维度也会很复杂。如果答案涉及的条件很多的情况，往往单轮的 QA 无法很好的解决客人问题。因此在度假业务的场景下，多轮次的 Task Bot 会占大多数。

1.1 错别字纠正

原始语句中难免会出现错字，错字可能会改变最终输出的答案。在识别意图之前首先通过纠错模块对错别字进行纠正。



兼容速度和准确率考虑，纠正分为规则部分和模型部分，度假业务中涉及到的地点比较多，在规则部分就能够覆盖大部分错别字的情况。模型部分首先会经过一个 CRF 模型输出字级别存在错误的可能性，生成候选集后，通过语言模型计算候选集句子的置信度，重排序得到最终纠正的结果。



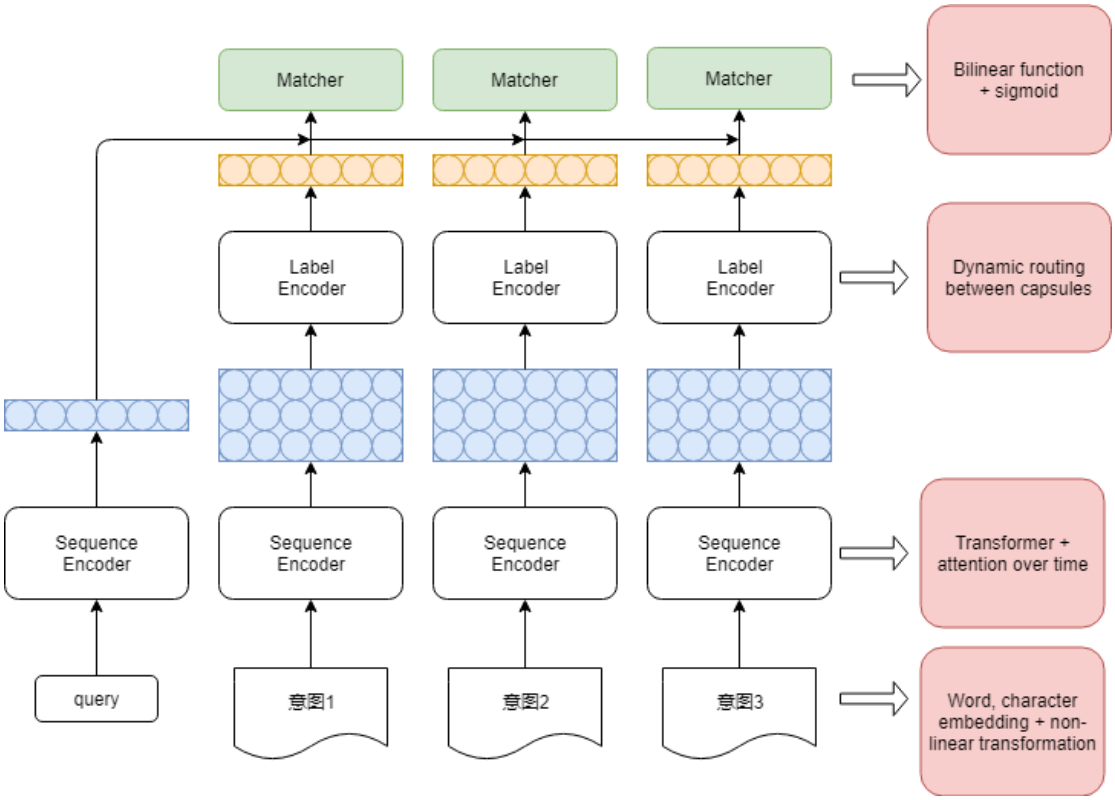
• 检查错误模型M_c主要使用了五种特征向量连接后进入 Bi-LSTM-ATT-CRF 模型，得到对每个字是否错误的判断。

• 语言模型M_l计算候选集替换为该字的情况在语言模型内的得分，文本转为特征向量后经两层向前和后向前的 Transformer，最后全连接计算 softmax。

1.2 意图识别

意图实质上是对客人问题的抽象化，比如常见的客人问及“这个产品多少钱？”，可转换为“询问价格”意图。而在直接服务客人的 C 端上，对回答答案准确率有较高的要求，高质量的服务背后首先是高准确率，而且通常在设计意图前期会存在意图训练语料不足的问题，因此一个高准确率并且弱监督的意图识别模型非常重要。

意图识别模型



意图识别模型整体采用上图的类似 matching network 框架，每个意图会有一个类别表示l_i，新的 query 经计算获得其句向量q_j，通过计算q_j和每个类别l_i的相似度得到该客人问题的意图。在意图识别模型优化过程中发现，相较于以前常使用的分类模型会有一定提高，同时更方便迁移到多意图的情景。

在训练阶段，共有已知C个类别，每个类别N个样本，语句$\{s_{ij}\}_{i=1,\dots C,j=1\dots N}$经特征向量经过 Bi-LSTM 层后再通过 Transformer-Attention 把一句话映射为一条向量e_{ij}，最后经胶囊网络获得每个类别的类别向量l_i。每个训练语句v_j得到句向量q_j后，再通过 Bilinear-Function-Sigmoid 计算q_j与l_i的相似度得分，最后采用二分类的对数似然损失函数计算损失。

模糊意图的处理

我们研究表明，客人在和机器人对话及与人对话的时候一些行为习惯是不同的。在面对机器人的时候，客人倾向于把机器人作为一个“搜索引擎”，常常输入关键词来获得回答，但关键词的信息不完整，通过模型或模板都无法返回切合的意图。针对于此，我们采用了“联想问”和“猜你想问”的功能来引导客人的提问方式。

联想问

客人在聊天输入栏输入问题的同时，显示相关的一些问题以供客人点选，由于是实时显示，对速度的要求较高，这里我们使用的是检索算法计算文本相似度。

我们会为每一个意图人工设置一些用户常问问题，当用户输入的时候，我们会用文本相似度

的算法，算出和用户输入最接近的三个常问问题，提示给用户供其选择。

猜你想问

对于“猜你想问”功能，主要是处理问句过短的语义不明的情况。举个例子，在签证领域，客人会输入“照片”，而和照片相关的意图有“是否需要照片”、“照片要求”、“照片尺寸大小”等等能够涉及到的十几个意图。在触发“猜你想问”后，会返回 4 个最关联的问题供客人点选。

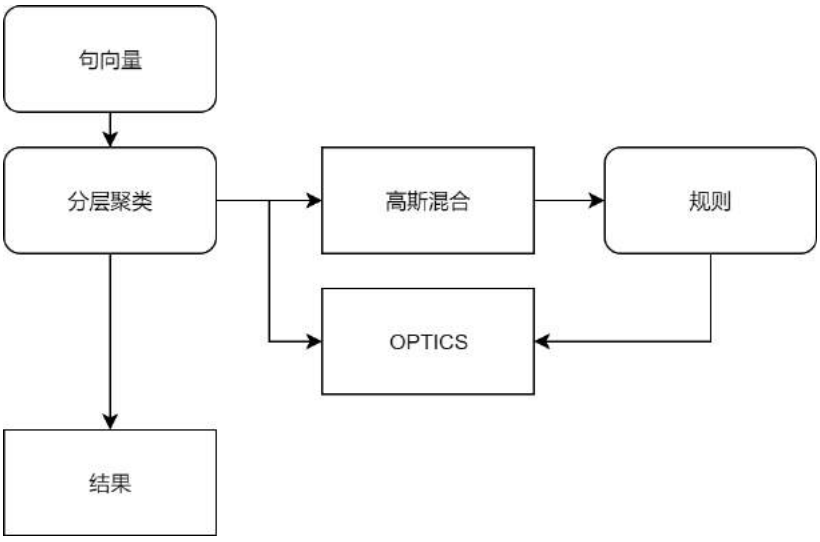
在使用“猜你想问”和“联想问”的机制后，可以引导部分客人的用户输入习惯，提升单轮次下信息输入的完整性及纯净度。

1.3 发现新意图

一个新业务线设计意图的时候，不可能把所有会出现的意图都理清楚，而是循序渐进，逐步增加。业务人员本身对业务的熟悉程度可提供新业务线的意图大框架，在小细节上难免存在漏缺，或是因为实时政策的变化产生的新问题。

比如说，在今年六月份大陆禁止发放台湾自由行签证，这段时间就新产生了很多类似于“已办的台湾签证是否还可使用”、“是否还能办台湾 G 签证”等这些新的客户问题。

层次聚类



我们采用的是对原始问题聚类的方法，把相似句聚集在一起。经过数据预处理后，生成句向量，第一层使用高斯混合得到一个初步的聚类结果，再通过规则判断是否需要再进行一次聚类，随后在第二层使用 OPTICS 聚类。

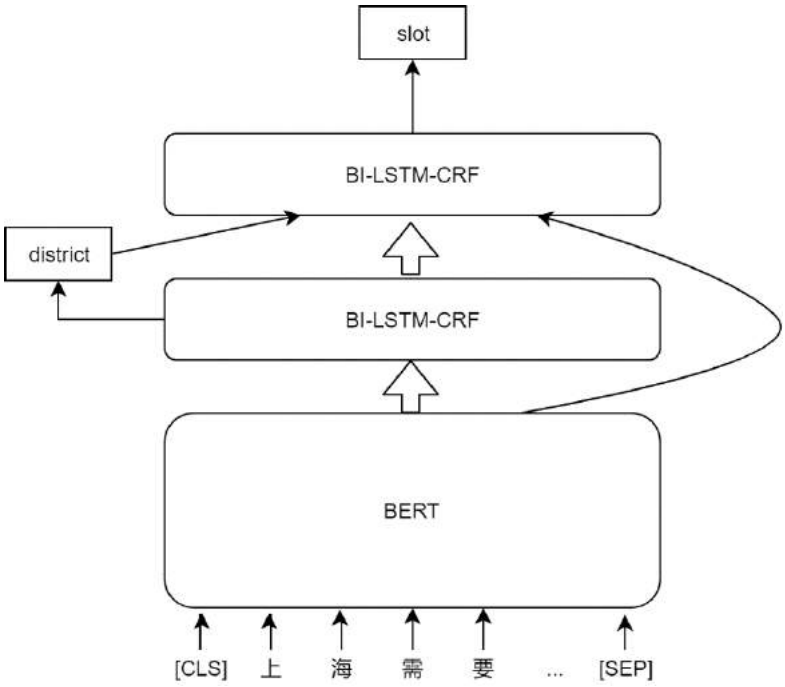
在用算法发现意图后，并不会即刻投入使用，而是业务做重审确定，整体上新意图的定位在于辅助业务对意图体系的完善。

1.4 槽位抽取

在 Task Bot 中，槽位信息抽取主要是服务于检索答案。比如签证一个常见问题“办签条件”，需要确定客人的办签国家、户口所在地、居住地等信息后才能给出最终回答。

有时客人的问题中直接会涉及相关槽位，目前槽位抽取采用的是规则+模型的方式。在实际应用中，规则能够覆盖 70%的情况，剩下的则由模型来负责。

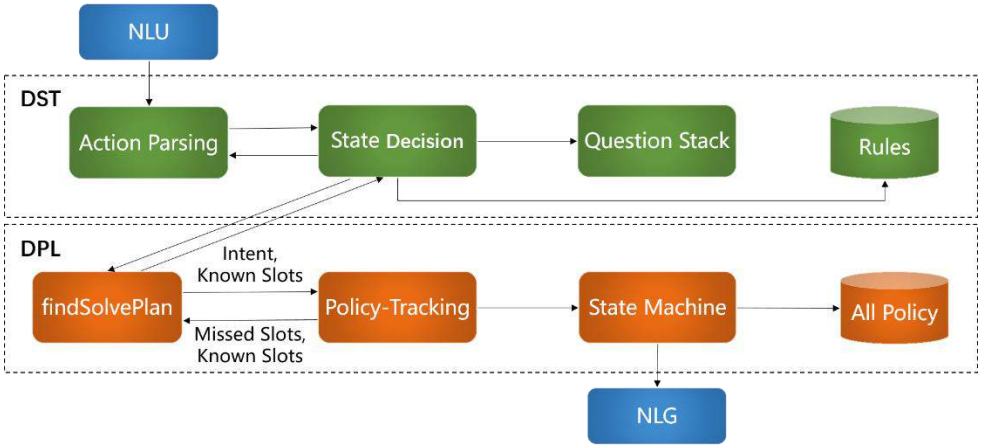
在度假业务里需抽取的槽位词有一个明显的层次关系，比如地点-送签地、地点-办签国家、职业-在职、职业-自由职业等，在模型的设计上会先抽第一层，第二层才是对最终结果的二级识别，通过多任务的学习，实际上每一层的任务是在对特征进行自动抽取。



大型的语言模型，比如说今年大热的 BERT，在很多 NLP 任务中大放光彩。在这个词槽抽取任务中，语句中会先经过 BERT 得到字向量b_i后，第一层经 Bi-LSTM-CRF 模型得到第一类的结果r_i以及 Bi-LSTM 的编码结果e_i^1，r_i会映射为对应的类向量l_i，经l_i、e_i^1、b_i连接后进入第二层 Bi-LSTM-CRF 后得到最终的词槽。在加入语言模型后，对于语料比较少以及地点比较多的情况提升会比较大，尤其是一些语料中没出现过的地点，加入语言模型后也能识别出来。

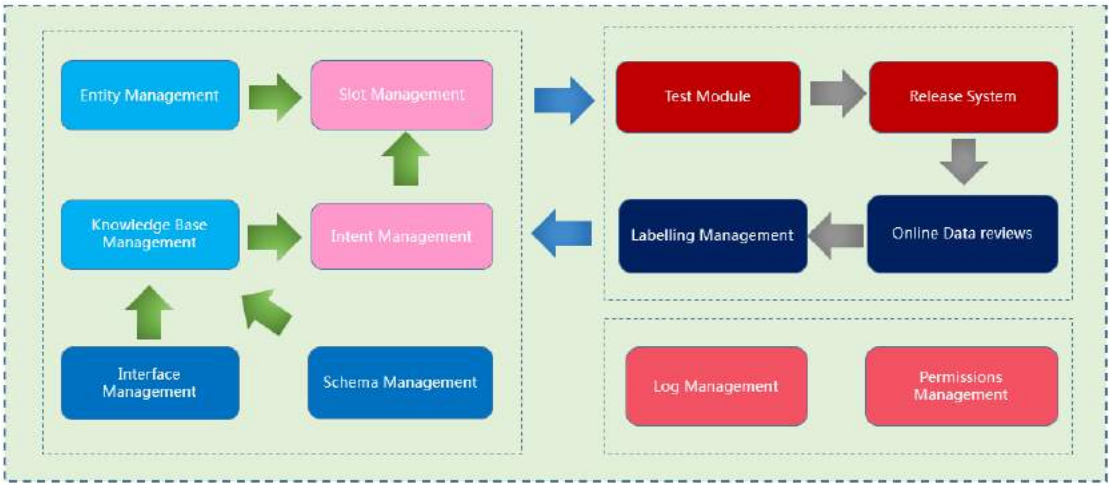
二、对话管理系统

对话管理系统模块主要负责对话状态追踪 DST（每轮意图、槽位的存储）、对话策略选取 DPL（反问或给出答案）、答案生成 NLG。在这部分接收 NLU 识别的意图和槽位结果，DST 把对话状态信息发送给 DPL，DPL 根据知识库中的规则返回机器人在下一轮的决策（回答问题、反问或其它操作）。



三、智能客服平台

在整体上，智能客服业务和技术的部分是解耦的。业务相关信息的设定和操作都是通过智能客服平台，包括不同业务线的意图和词槽的设定、答案配置、数据审核、测试、标注等。新建一条业务线的智能客服应用，只需要在平台上新建项目，输入设定的意图、对应的语料、必要的槽位和对应的答案。



此外，平台上的答案配置也很灵活，可以是固定回答，可以是知识图谱的 schema，可以是外部的接口，或是随不同词槽设定的回复等等。

四、结语

以上是度假人机交互的主要技术和成果，目前我们已经完成了一个智能客服项目落地的闭环，其中还有很多内容可以持续完善，比如多轮的意图识别、更多主动对话的探索等等。

未来的智能客服机器人将往多模态和多语言方向发展，支持语音和图像等模态的解析，支持英法日韩等多国的语言。智能客服还将提供主动服务模式、人机协同模式、群聊功能等多种模式。此外，采用大规模挖掘和生产的方式降低人工标注成本也是未来的主要方向之一。

XGBoost 在携程搜索排序中的应用

【作者简介】曹城，携程搜索部门高级研发工程师，主要负责携程搜索的个性化推荐和搜索排序等工作。

一、前言

在互联网高速发展的今天，越来越复杂的特征被应用到搜索中，对于检索模型的排序，基本的业务规则排序或者人工调参的方式已经不能满足需求了，此时由于大数据的加持，机器学习、深度学习成为了一项可以选择的方式。

携程主站搜索作为主要的流量入口之一，是用户浏览信息的重要方式。用户搜索方式多样、对接业务多样给携程主站搜索（下文简称大搜）带来了许多挑战，如：

- 搜索方式多样化
- 场景多样化
- 业务多样化
- 意图多样化
- 用户多样化

为了更好满足搜索的多样化，大搜团队对传统机器学习和深度学习方向进行探索。

说起机器学习和深度学习，是个很大的话题，今天我们只来一起聊聊传统机器学习中 XGBoost 在大搜中的排序实践。

二、XGBoost 探索与实践

聊起搜索排序，那肯定离不开 L2R。Learning to Rank，简称（L2R），是一个监督学习的过程，需要提前做特征选取、训练数据的获取然后再做模型训练。

L2R 可以分为：

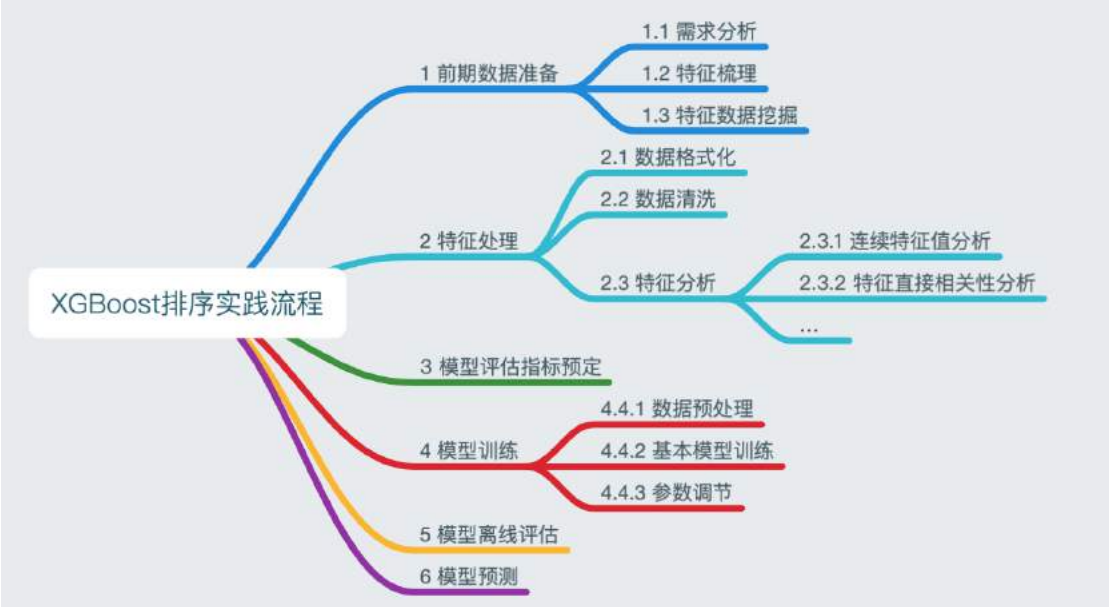
- PointWise
- PairWise
- ListWise

PointWise 方法只考虑给定查询下单个文档的绝对相关度，不考虑其他文档和给定查询的相关度。

PairWise 方法考虑给定查询下两个文档直接的相对相关度。比如给定查询 query 的一个真实文档序列，我们只需要考虑任意两个相关度不同的文档直接的相对相关度。相比 PointWise，PairWise 方法通过考虑任意两个文档直接的相关度进行排序，有一定的优势。

ListWise 直接考虑给定查询下的文档集合的整体序列，直接优化模型输出的文档序列，使得尽可能接近真实文档序列。

下面先简单介绍 XGBoost 的应用流程：



XGBoost 是一个优化的分布式梯度增强库，增强树模型，灵活便捷。但对高维度的稀疏矩阵支持不是很好，另外调参确实有点麻烦。

三、特征工程实践

在传统机器学习下，特征工程显的尤为重要，不论后续模型工程做的多好，如果前期的特征工程没有做好，那么训练的结果不会有多好。所以对特征处理的总体逻辑如下：

3.1 前期数据准备

首先，我们需要进行需求分析，就是在什么场景下排序。假设我们需要针对搜索召回的 POI 场景进行排序，那么需要确定几件事情：

数据来源：搜索的数据就是各种 POI，然后需要确定我们有哪些数据可以用来排序，比如最近半年的搜索 POI 的曝光点击数据等；

特征梳理：需要梳理影响 POI 排序的因子，例如查询相关特征、POI 相关特征、用户相关特征等；

标注规则制定：每次搜索召回的每个 POI，会有曝光和点击数据，简单点，比如：我们可以将曝光位置作为默认标注分，当有点击，就将标注分在原来的基础上加一；

数据埋点/数据抽取：这是两种方式，可以根据实际需求进行选择；

- 数据埋点：可以在线上实时生成特征，然后进行日志埋点，离线分析的时候可以直接从日志中拉取即可，这种方式，需要提前进行埋点。

- 数据抽取：可以通过大数据平台拉取历史数据，然后进行离线计算抽取所需特征，这种方式虽然慢点，但是可以拉取历史数据进行分析。

3.2 特征处理

前期的数据准备工作完成了，接下来可以开始看看数据质量了。

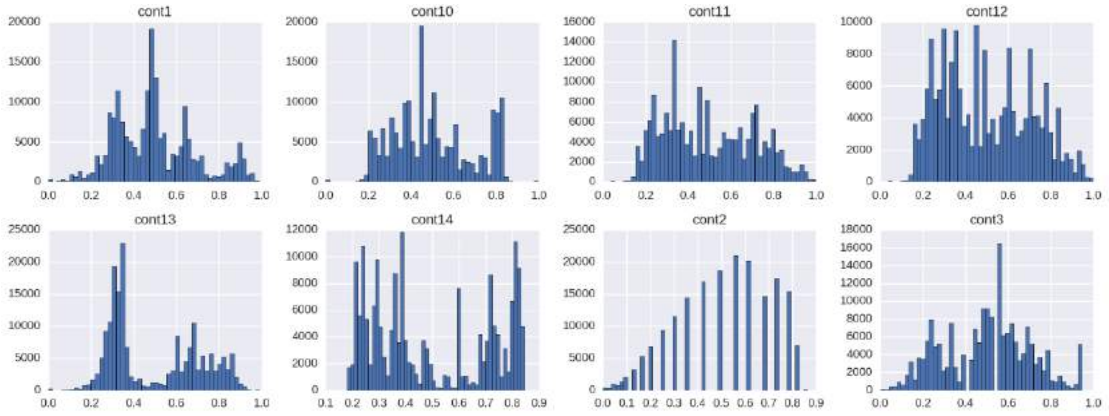
查看缺失值：绝大多数情况下，我们都需要对缺失值进行处理；

特征归一化处理：监督学习对特征的尺度非常敏感，因此，需要对特征归一化用来促进模型更好的收敛；

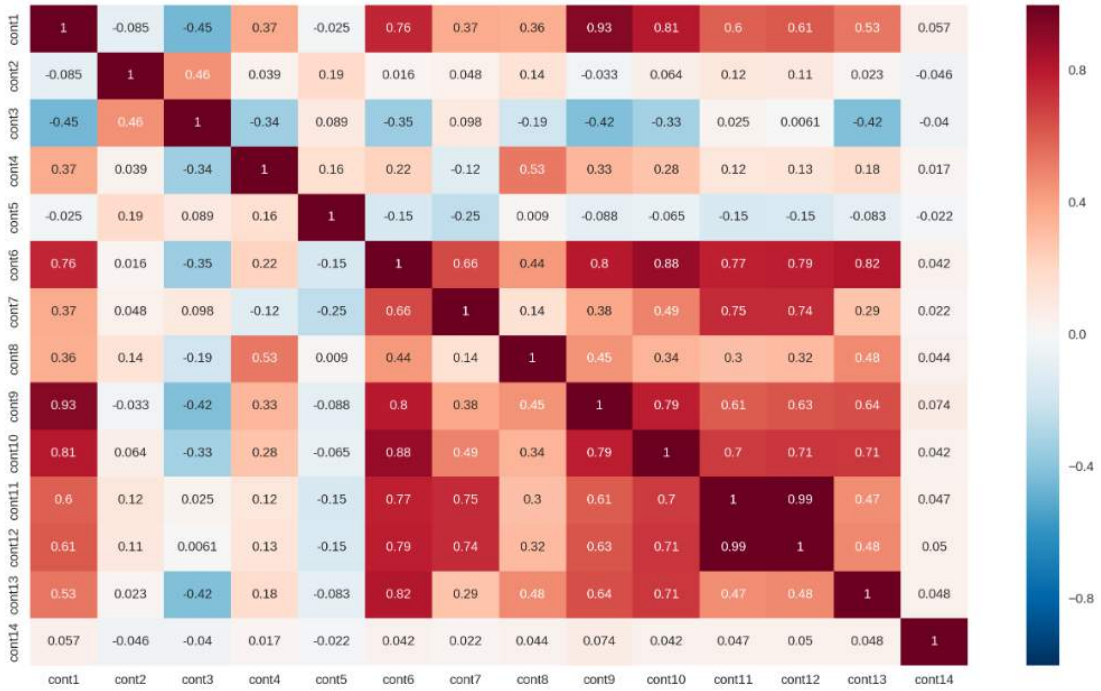
噪声点处理：异常的数据会影响模型预测的正确性；

特征连续值分析：分析特征的值分布范围是否均匀；

特征之间的相关性分析；



通过连续值特征可以分析每个特征值的大致分布范围，有利于对相关特征进行数据补充或者重新筛选。



通过特征相关性的分析，如上我们看到几个特征之间有很高的相关性，那么可以帮助我们做特征组合或者特征筛选等等方面决策。

四、模型工程实践

4.1 评估指标制定

在搜索业务中，考虑的有以下两种情况：

- 看重用户搜索的成功率，即有没有点击；
- 看重页面第一屏的曝光点击率；

在文章开头提到的 L2R 的三种分类中，我们在 XGBoost 参数 objective 配置“rank:pairwise”，同时使用搜索系统常用的评估指标 NDCG (Normalized Discounted Cumulative Gain) 。

```
def _to_list(x):
    if isinstance(x, list):
        return x
    return [x]

def ndcg(y_true, y_pred, k=20, rel_threshold=0):
    if k <= 0:
        return 0
```

```
    y_true = _to_list(np.squeeze(y_true).tolist())
    y_pred = _to_list(np.squeeze(y_pred).tolist())
    c = list(zip(y_true, y_pred))
    random.shuffle(c)
    c_g = sorted(c, key=lambda x: x[0], reverse=True)
    c_p = sorted(c, key=lambda x: x[1], reverse=True)
    idcg = 0
    ndcg = 0
    for i, (g, p) in enumerate(c_g):
        if i >= k:
            break
        if g > rel_threshold:
            idcg += (math.pow(2, g) - 1) / math.log(2 + i)
    for i, (g, p) in enumerate(c_p):
        if i >= k:
            break
        if g > rel_threshold:
            ndcg += (math.pow(2, g) - 1) / math.log(2 + i)
    if idcg == 0:
        return 0
    else:
        return ndcg / idcg
```

4.2 初始模型训练

前期通过基础的模型训练，可以初步得出一些初始参数和相关特征的重要度等相关信息。

```
train_dmatrix = DMatrix(x_train, y_train)
valid_dmatrix = DMatrix(x_valid, y_valid)
test_dmatrix = DMatrix(x_test)

train_dmatrix.set_group(group_train)
valid_dmatrix.set_group(group_valid)
test_dmatrix.set_group(group_test)

params = {'objective': 'rank:pairwise', 'eta': 0.5, 'gamma': 1.0,
                    'min_child_weight':    0.5,    'max_depth':    8,'eval_metric':'ndcg@10-
','nthread':16}
xgb_model = xgb.train(params, train_dmatrix, num_boost_round=1000,
                                    evals=[(valid_dmatrix, 'validation')])
```

```
[16:59:17] src/tree/updater_prune.cc:74: tree pruning end, 1 roots, 154 extra nodes, 56 pruned nodes, max_depth=8
[995]   validation-ndcg@10-:0.902236
[16:59:19] src/tree/updater_prune.cc:74: tree pruning end, 1 roots, 138 extra nodes, 106 pruned nodes, max_depth=8
[996]   validation-ndcg@10-:0.902233
[16:59:20] src/tree/updater_prune.cc:74: tree pruning end, 1 roots, 134 extra nodes, 120 pruned nodes, max_depth=8
[997]   validation-ndcg@10-:0.902241
[16:59:22] src/tree/updater_prune.cc:74: tree pruning end, 1 roots, 230 extra nodes, 72 pruned nodes, max_depth=8
[998]   validation-ndcg@10-:0.902261
[16:59:23] src/tree/updater_prune.cc:74: tree pruning end, 1 roots, 260 extra nodes, 78 pruned nodes, max_depth=8
[999]   validation-ndcg@10-:0.902291
```

```
import pandas as pd

print('特征名称', '特征权重值')
feature_score = xgb_model.get_fscore()
pd.Series(feature_score).sort_values(ascending=False)
```

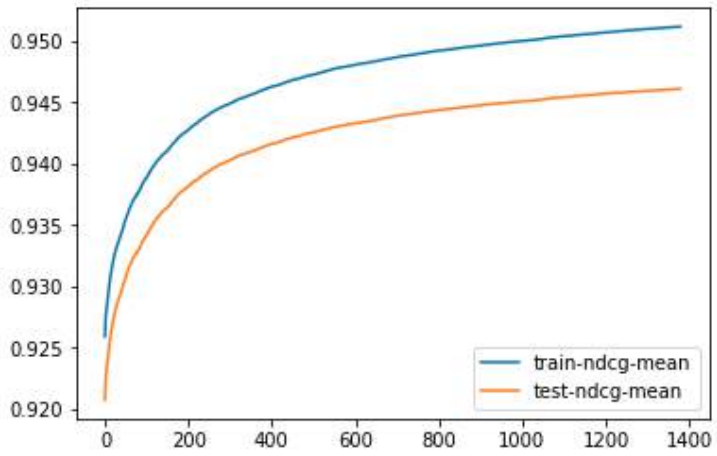
```
特征名称 特征权重值
f1      52632
f2      31876
f10     31091
f3      23172
f13     22050
f5      18675
f4      14639
f6      13974
f11     10936
f18      8696
f16      7088
f12      6990
f14      6434
f24      4696
f15      4478
f22      2080
f20      1798
f19       883
f0        744
f21       663
f9        463
dtype: int64
```

4.3 模型调优五部曲

通过上述基础的模型训练，我们可以得出相关的初始参数，进入到五部曲环节，XGBoost 参数调节基本为五个环节：

Step 1：选择一组初始参数；
Step 2：改变 max_depth 和 min_child_weight；
Step 3：调节 gamma 降低模型过拟合风险；
Step 4：调节 subsample 和 colsample_bytree 改变数据采样策略；
Step 5：调节学习率 eta；

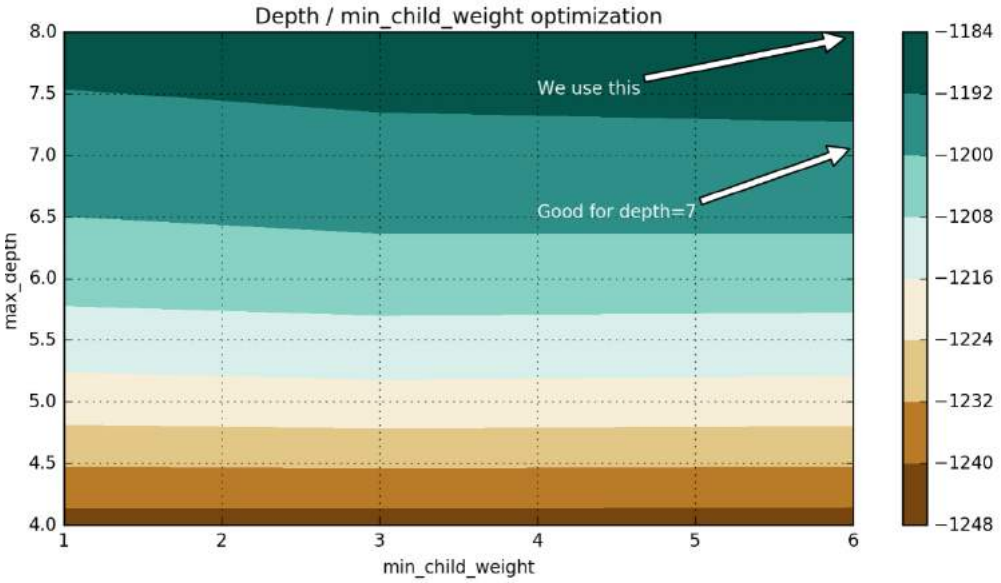
例如我们在通过 step1，可以观察弱分类数目的大致范围，看看模型是过拟合还是欠拟合。



通过 step2 调整树的深度和节点权重，这些参数对 XGBoost 性能影响最大，我们简要概述他们：

- max_depth：树的最大深度。增加这个值会使模型更加复杂，也容易出现过拟合，深度 3-10 是合理的；
- min_child_weight：正则化参数。如果树分区中的实例权重小于定义的总和，则停止树构建过程。

可以通过网格搜索发现最佳结果，当然也可以通过其他方式。



我们看到，从网格搜索的结果，分数的提高主要是基于 max_depth 增加。min_child_weight 稍有影响的成绩，但是 min_child_weight = 6 会更好一些。

4.4 模型离线评估

通过调优五部曲，训练，生成最终的模型之后，就要进入离线评估阶段。离线拉取线上生产用户的请求，模拟生产，对模型预测的结果进行检验，根据在之前评估指标制定环节所提到的用户点击率和第一屏曝光点击率，比对线上用户点击产品的位置和模型预测的位置，同时对比两者之间的第一屏曝光点击率。

4.5 模型预测

通过 AB 实验，对模型进行线上预测，实时监测效果评估，方便之后的迭代和优化。

五、总结与展望

对于传统的机器学习，最重要的一步是要做好需求分析，评估这种算法是否能有效解决该问题，否则得不偿失；

前期的特征工程非常关键。通过多次的摸索发现，特征没有选取好，或者数据覆盖不全，标注没做好，导致后续模型不论怎么调优，都无法达到预定的效果；

选定目标后，可以先尝试一些优秀的开源工具、优秀的数据分析工具。直观的图表能帮助你做更好的决策，优秀的算法库，能避免重复造轮子；

单一的算法无法满足搜索排序应用场景，多模型融合以及深度学习方向需要做更深入的探索与实践；

框架架构篇

携程框架团队对于应用监控系统的探索与思考

【作者简介】鄞劭涵，携程框架架构研发部高级软件工程师，目前主要从事应用监控系统以及消息队列相关基础框架的研发。

一、为什么需要应用监控系统

随着市场环境的变化以及国际化的进程，企业的各种对内、对外需求也日益增长。服务化的架构以及容器化的应用加速了各种功能、产品的迭代与更新。随之而来，我们也面临着一个不断膨胀，日渐复杂的系统。

复杂度的成倍增加对故障的根因分析、执行流程的调优以及数据链路的追踪带来了极大的挑战。因此，对于一个企业级的应用监控系统来说，也应该持续地发展、演化，才能更好地解决痛点，提高用户的整体效率。

如今，应用的执行流程往往由种种内外部依赖、软硬件结合构成。相应的，针对不同的需求，监控领域也有着业务指标监控、应用监控以及基础设施监控等等类别。

对于应用监控系统来说，它的主要职责是管理、监控一个软件应用的性能与可用性 [1]。在服务化场景下，它应致力于快速监测并诊断出一个复杂的服务调用链路中的潜在问题，帮助研发人员更好地维护服务的质量 [2]。

二、应用监控系统的内容

一个完整的应用监控体系往往包含着多种组成，例如客户端的日志以及宿主机的心跳状态等。随着系统复杂度的增加，每一个相关的组件都有可能含有帮助用户定位系统异常根因的线索。因此，对于种种不同来源的原始数据，都应该进行归类、量化，这样才能帮助用户在遇到问题或者维护应用的时候更加有的放矢。

不同的系统有着不同的监控粒度以及目标着重点，因而它们的度量也不尽相同。依托对于携程当前服务应用的场景以及对于外部资源的参考，我们对于应用监控系统的作用域进行了如下定义 [3]:

1、Trace：即一次完整的事务调用请求。比方说一个用户的下单请求，经过层层服务预处理，到支付服务成功，数据落库，成功返回，这就是一条完整的 Trace 。Trace 最大的特点就是它含有上下文环境，通常来说会由一个唯一的 ID 来进行标识。一个 Trace 内可能有多个不同的事务 (Transaction) 以及标志事件 (Event) 组成。

2、Log：即日志。代表了用户主动记录的离散的数据。通常来说就是用户采用 logging 组件输出到日志文件的具有 WARN，INFO，ERROR 等用来表示不同执行状态级别的信息。这些日志信息在用户进行问题分析判断时可以提供更为详尽的线索。

3、Metric：代表了用户定义关心的或者通用的预定义的一些运行时指标。通常来说，Metric

具有时序可累积性。根据不同的粒度需求，Metric 可以做到小时级、分钟级、秒级等。通常来说， Metric 是数据采集项目的聚合，旨在为用户展示某个指标在某个时段的运行状态。

4、Report：报表是针对某种特定领域，经过对收集而来的各种数据进行统计、分析而产生的关键信息展示载体。报表含有丰富的信息，通过报表用户可以获得关于特定领域指标集的多维信息，从而更好地作出排障、调优决策。

三、分布式链路追踪

服务化架构下，以往的单一应用被拆分为一个个不同维度的服务，每个完整的事务逻辑往往由多种外部请求构成。调用关系的复杂性增加了发现应用问题的难度。同时，多个服务间的依赖关系，依赖合理性与调用性能分析以及资源容量规划也成为了需要考虑的问题。

由此可见，一个完整的分布式调用链路追踪是应用监控体系中举足轻重的一环。如果说 Metric 和 Report 是具有统计性的较粗粒度的数据归纳结果，分布式链路追踪 (Trace) 就是一个细粒度的独立完整的调用记录。因此，Trace 提供的翔实信息对于理解系统行为、应用长尾分析以及故障根因分析有着很大助益。

当下也有许多的监控组件支持分布式调用链追踪。例如开源的 Zipkin，Skywalking，Pinpoint，Jaeger，CAT 以及商业的 NewRelic APM 等产品。

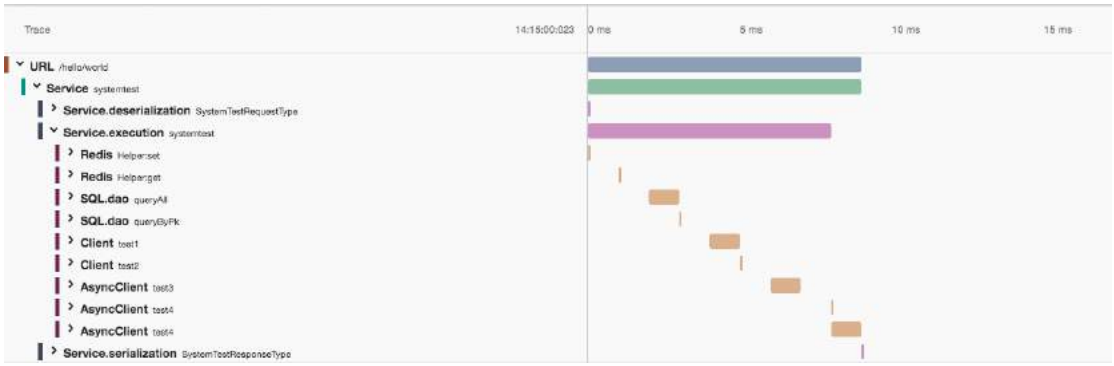
它们的理论模型大多都是借鉴或者近似于 Google Dapper 这篇论文中提到的思想 [4]。每个 Trace 由树形结构组织的 Span 构成，每个 Span 代表了某次具有特定耗时的外部调用，例如一次 RPC 请求或者一次 DB 操作。与 Dapper 以及 OpenTracing 的思想类似 [5]，我们将 Trace 内的调用链路组织成一个个事务(Transaction)，每个事务含有自己的操作(Operation)，目标(Target) 以及属性(Properties)。

比方说，如今有许多服务都会大量的采用 Redis 作为数据缓存，一次这样的操作就是一种操作 Redis 的 Transaction。通过设置该事务的目标以及属性(比方说将某次 Redis 操作的 Key 置为某种属性)，用户可以更详尽地了解这次操作的流程与内容。

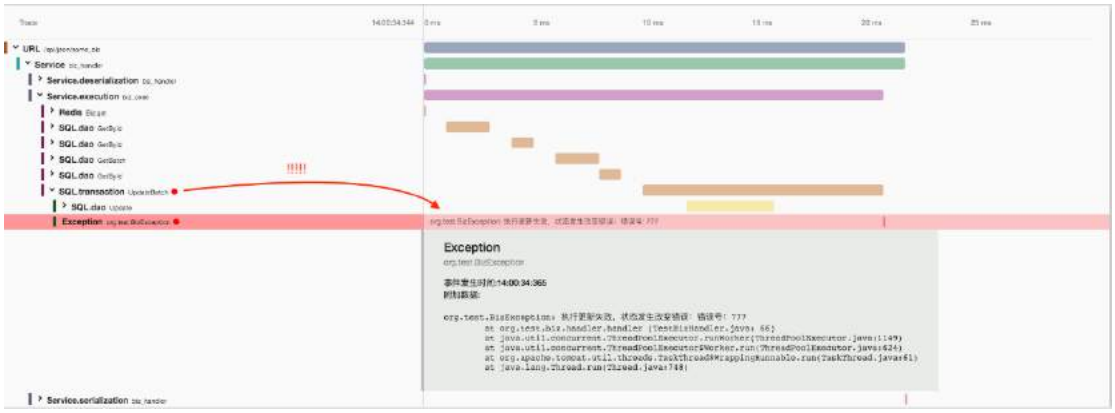
同时，对于不具备时间跨度的某些标识类事件，我们采用 Event 进行标识。这可以帮助我们快速定位、了解到该操作的执行路径。如下就是一个典型的 Redis Transaction：

通过 Trace 及其丰富的层级、属性，我们能够更好的理解整个系统的运行流程，更快地识别

耗时过长的链路。通过链路抽样的分析比较不同版本下服务的性能表现。



当服务发生异常时，一个完整的 Trace 也可以帮助用户快速定位问题根因。异常状态会从异常产生的具体节点上浮到对应的事务层级。这样的组织方式有助于排障人员发现到底是哪一个下游服务，哪一个 DB 还是自身执行发生了问题，同时异常抛出时的 StackTrace 也会被附加到对应的 Transaction 中，方便用户回溯异常产生的现场。



四、Trace 如何针对开发、测试、运维人员进行设计

分布式的调用链路追踪具有很明显的层级关系，无论是性能调优亦或是异常分析，往往都能通过不断地定位、下钻，缩小目标范围，最终找到根因。

作为一个应用监控系统重要的一环，Trace 能够对关注于服务性能、质量的人提供一定的帮助。对于开发人员来说，每个应用的研发人员都应该关注并维护好自己的服务质量。

所谓服务质量，包含了响应时间（性能），错误率，稳定性等多种特性。对于性能需求，Trace 的记录天然的展示了该调用链路的耗时分布，通过对调用链路中对应目标的事务的分析，开发人员可以快速地定位系统交互链路上的瓶颈，从而更好地对症下药。

与此同时，一个良好的服务也应该具有高稳定性。特别是对于直接与客户打交道的业务来说，一次缓慢的调用就有可能意味着一个潜在客户的流失。监控系统应该收集并分析包含长尾调用等的 Trace ，通过这样的记录我们能够较快地发掘出造成长尾的原因，尽可能的使每次调用维持在预期的延时之内。

与开发人员不同，测试人员更关注于系统的准确性，以及新旧版本应用在执行上的差异。通过具有完整层级组织以及特殊事件标识(Event)的 Trace，测试人员可以更好地了解系统行为以及执行路径。一方面，Trace 可以展示出该次执行的正确性，另一方面，通过设计、实施不同执行路径的测试，可以做到更高的覆盖率，减少系统出现异常的可能。

对于应用监控运维人员来说，故障的及时发现以及根因分析相对来说是一个更为重要的目标。Trace 的异常节点高亮以及异常状态上浮可以更好的辅助定位问题的发生源。通过事务记录下的属性，问题可以更进一步地被缩小到调用目标、对应操作甚至是发生的基础设施的级别(Redis 操作、Key、所在 IP，或者是 MySQL 对应的 DB、执行语句等)，进而可以制定出对应的解决方案。

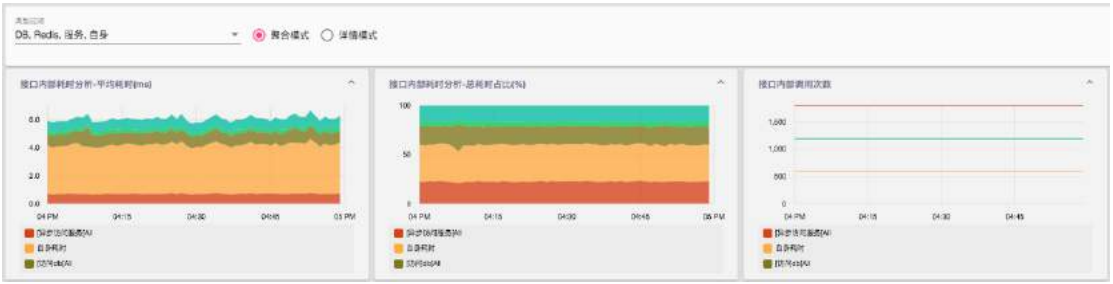
五、服务成分分析

应用监控系统的主要目的是帮助用户监控、维护一定的服务质量。稳定性对于一个服务来说也是一个至关重要的指标。

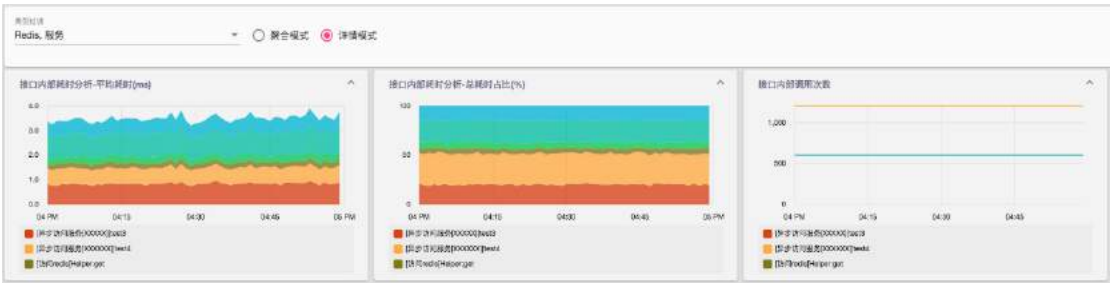
对此，单次的 Trace 并不能很好的展示一定时段内的服务表现。在分布式、服务化的场景下，针对一些常用的调用，我们引入了“组成成分”的概念。

比方说，在一段时间内，针对某个特定的 DB 的 SELECT 操作就属于这个服务的调用链路的一个成分。与此相近，特定的 Redis 操作，或者外部服务调用也是成分的一种。

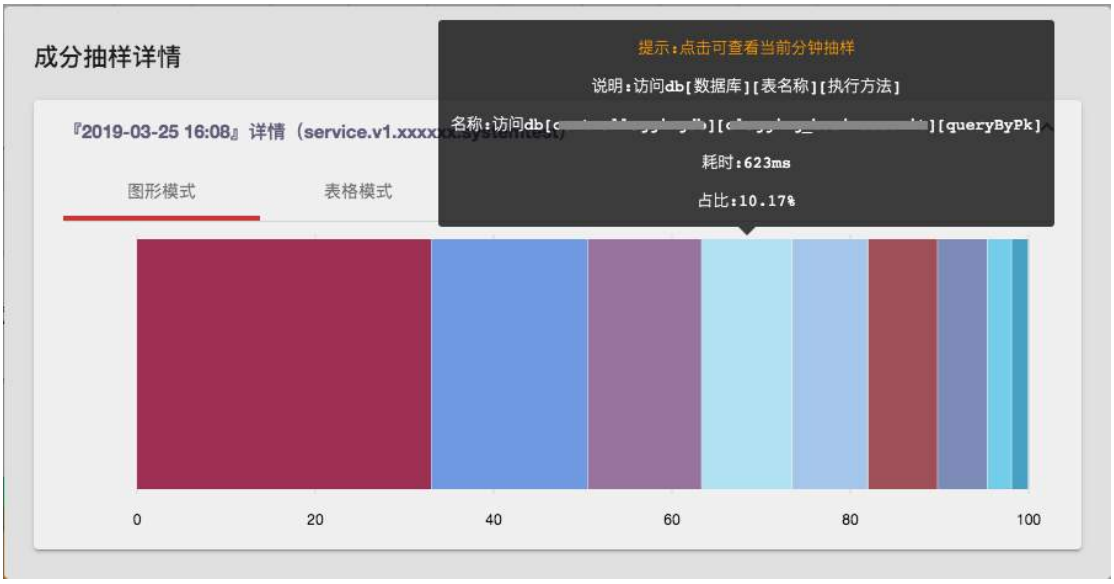
通过对于成分指标的分析，将其聚合成一个成分分析的报表。成分分析报表展示的是对应服务在其执行路径上各个不同事务的表现情况。一个典型的成分报表构成如下:



成分报表分析了各种不同成分的响应时间、被调用次数以及该成分在总体耗时内的占比。成分能够帮助用户整体掌握系统的运行状态。多样化的成分聚合、下钻能够为分布式系统的各调用链路分析提供依据。

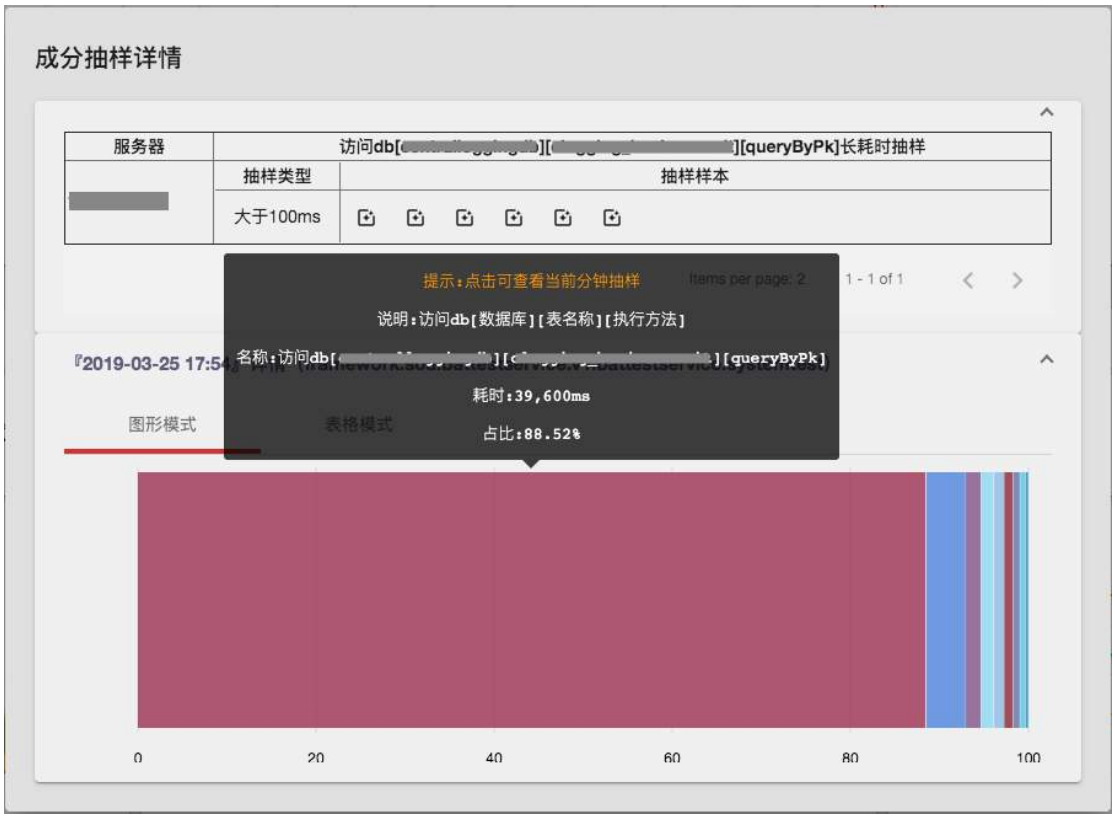


目前来说，成分数据做到了秒级收集，分钟级聚合。在异常发生时，通过对于特定时段的成分进行下钻，也可以帮助我们分析、追溯调用链路中出现问题的根源。



当系统出现故障时，往往也伴随着成分指标的异常。如下述示例中，服务的耗时出现了突增，收到了告警。为了定位到是整个调用链路中的哪一部分出现了问题，可以查看对应时间段的成分分析。

观察得到，在这一分钟内服务对各个外部链路的访问次数没有明显异常。但是数据库的访问时间出现了突增，可以明显看到访问数据库对应的成分的耗时占比在整个调用中的比例扩大了。继而，我们去查询对应现象发生的当前分钟的情况，通过详情面版继而可以定位到是针对特定的数据库、表的访问出现了异常。在此之后，通过收集而来的异常 Trace，可以进而为后续定位服务发生耗时异常的根因分析提供线索。



六、其它思考

在持续演进的过程中，我们也对现有的系统在易用性、可用性方面进行了反思。

对于一个应用监控系统来说，不断地做“加法”，收集、分析并不断完善从软件到硬件等方方面面的数据指标固然不可或缺，但是善用“减法”，去粗取精反而有的时候更能使用户有的放矢。

现如今，大多数应用监控系统都能够收集并提供多种指标的展示。对于较为没有经验的用户来说，他们往往在遇到问题时会不知道应该从哪里下手。与之相反，有经验的同事往往有他们自己的排障习惯，他们能在各种系统的各种指标看板中来回穿梭，最终一步步缩小范围，找到原因。

因此，我们将思维反转了一下，不通过堆砌数据，而是依托于问题的场景，通过与有丰富排障经验的同事进行沟通，将数据整合成一个个具有特定排障目的的模块。

对于一个服务化的场景中，可能会发生服务耗时突增，服务出现错误以及服务调用存在长尾等多种潜在问题。举例说来，针对服务耗时异常场景，我们组织整理了一个专门用于耗时问题排障的模块。外部调用方的异常流量，内部调用链路成分的异常，应用的 GC 情况以及运行时宿主机的负载都可能导致服务时间出现异常。通过将对应指标进行有机聚合，用户可以学习他人的经验,从而进行更有效的排障。

七、附言

对于应用监控领域我们仍在不断地探索、完善中。文中内容如有任何错误之处还望不吝斧正。有任何的意见、建议，也欢迎在评论区探讨。

引用

[1] Wikipedia contributors. “Application performance management.”/Wikipedia, The Free Encyclopedia/. Wikipedia, The Free Encyclopedia, 7 Mar. 2019. Web. 24 Mar. 2019.

[2] Wikipedia contributors. “Application service management.”/Wikipedia, The Free Encyclopedia/. Wikipedia, The Free Encyclopedia, 2 May. 2018. Web. 24 Mar. 2019.

[3] Peter Bourgon. “Metrics, tracing, and logging.”, 21 Feb 2017.

[4] Sigelman, Benjamin H., et al. “Dapper, a large-scale distributed systems tracing infrastructure.” (2010).

[5] OpenTracing contributors. “https://opentracing.io/docs/overview/spans/”.

聊聊携程升级 Dubbo 的踩坑历程

【作者简介】顾海洋，携程框架架构研发部技术专家，负责携程分布式服务化领域的工作。目前主要负责 Dubbo 在携程的二次开发和推广工作。

一、什么是 CDubbo

携程从 2017 年 11 月左右开始调研，真正落地是在 2018 年 4 月发布的 CDubbo 0.1.1 版本。在携程内部，我们管他叫 CDubbo，言下之意就是携程版的 Dubbo。考虑到以后升级的问题，CDubbo SDK 是对 Dubbo SDK 的扩展和包装，保留了 Dubbo 所有的扩展和配置能力。

目前，生产环境已经从第一个 0.1.1 版本，到目前的 0.13.3 版本，历经十几个版本的迭代，服务端有 156 个应用，客户端 170 个应用，生产实例数 2000 个左右。

二、升级 2.7.3 的几个理由

2.5.10 版本在携程只用了一年半左右，业务的应用也不算很多，这么快就大版本升级，主要是遇到了下面的几个问题。

1）2.5.10 的异步也是有阻塞的
2.5.10 版本只支持客户端异步，而且是基于 JDK 1.6 的 Future，并不是真正意义上的异步，本质上还是阻塞的，只不过是从 DubboClientHandler 线程切换到了业务线程。

2）支持服务端异步
对于微服务来说，一般又会调用外部服务，在网络 IO 比较多的场景下异步服务的优势会很明显，可以充分利用 CPU 资源，提高系统吞吐量，降低响应时间。部分机票、酒店等业务同学明确表示需要服务端异步。

3）现阶段不兼容问题带来的副作用较小
不兼容问题大概率是服务端和客户端的版本不一致，比如服务端 2.7.0，客户端 2.5.10。考虑到携程目前 CDubbo 服务的比例还不太高，早一点升级对业务的影响会比较小。

4）为之后的升级做铺垫
携程业务场景很广泛，部分业务已经明确表示需要 2.7.0 的服务端异步，也有业务在尝试 3.0 的 Reactive 了。如果先升级到 2.7.0，以后再升级 3.0 会比较容易些，如果直接从 2.5.10 升级到 3.0 版本，可能升级不过去，或者无法透明升级。

5）支持三中心
2.5.10 只有注册中心，注册数据和配置数据对注册中心的压力比较大。2.7.0 对模型重构，拆分成注册中心、元数据中心、配置中心，职责划分更合理。为了接入公司的测试平台，需要用到服务的元数据信息，2.7.0 正好提供了这个能力。

三、第一阶段升级及踩坑历程

注：为了表达方便，后续提到的 Apache 代表 org.apache 的 package，Alibaba 代表 com.alibaba 的 package。

第一阶段升级 2.7.0 是在 2019 年 3 月份左右，大概花了三周的时间，我们先来看下所遇到的几个问题吧。

3.1 变更 package 导致的不兼容

2.7.0 把 package 从 com.alibaba 改成了 org.apache，虽然对低版本做了兼容，但是还是会发现部分 class 找不到了，例如：Alibaba 的 DubboComponentScan 就已经被删掉了。取而代之的是 Apache 的 DubboComponentScan，不过这个问题在编译时就会报错了。

3.2 Apache 的 Constants 常量类被拆分

升级到 2.7.0 版本之后，Alibaba package 的 Constants 还是没变，但是如果要用新功能升级到 Apache package，你会发现 Constants 被拆分成 RegistryConstants, CommonConstants, RemotingConstants 等多个常量类。新的常量类只是分散到不同的 class 中，只要换个引用就可以解决了。

3.3 Apache 的 Router 接口新增了部分方法

如果扩展是基于 Alibaba 的 Router 接口，Dubbo 已经做了默认实现，应该不会存在兼容性问题。这次，我们直接换成了 Apache 的 Router 接口，因为新加了 isRuntime、isForce、getPriority 方法编译时就报错了。

3.4 Apache 的 ProxyFactory 接口新增了 getProxy 方法

我们这次升级是把 Alibaba 的 ProxyFactory 换到了 Apache 的 package 下，2.7.0 版本中对该接口新增了 getProxy 方法，编译时会报错。如果不需要扩展这部分功能，可以通过 delegate 机制保留默认实现就可以了。

3.5 限制 ApplicationConfig 必须全局唯一

2.5.10 版本对于 ApplicationConfig 没有限制，服务端起多个服务时可以配置独立的 ApplicationConfig。但是从 2.7.0 开始 ApplicationConfig 就会要求全局唯一，如果一个应用定义了多个不同的 ApplicationConfig 就会报错。

Apache 的 ConfigManager 的 setApplication 会检查是否 duplicate。

```
public void setApplication(ApplicationConfig application) {
    if (application != null) {
        checkDuplicate(this.application, application);
        this.application = application;
    }
}
```

3.6 JDK 1.8

2.7.0 为了支持真正的异步，用到了 JDK 1.8 的 CompletableFuture，也用到了 1.8 的 Supplier、Consumer 等操作符。如果业务的应用还是基于 JDK 1.7 打包的，升级后就会导致发布失败。由于我们这次是公司层面的整体升级，就需要所有业务应用都升级到 1.8 才可以发布。

3.7 默认升级到 Netty4

为了接入公司的 CAT 监控系统，需要把 Codec 的监控埋点数据通过 ThreadLocal 传递下去。

但是，2.7.0 把 Netty 的版本从默认的 Netty3 升级到了 Netty4，这两个版本的线程模型是不一样的，Netty3 的 decode 是在 New IO worker 线程，Netty4 是 NettyServerWorker 线程，导致原有逻辑的监控埋点数据传不过来。为了暂时解决这个问题，我们把默认的 Netty 版本降回了 Netty3。

注：CAT 是点评开源的实时应用监控平台，目前在携程也有落地，在 Github（https://github.com/dianping/cat）上也已经超过 1 万颗星。

3.8 异步请求一直 hang 住

扩展 2.5.10 版本的时候，为了支持对客户端异步的埋点，我们对 RpcContext 的 Future 重新包装了，用户拿到的 Future 已经是被我们包装过的 FutureAdapter 了。

```
Future<?> f = RpcContext.getContext().getFuture();

if (f instanceof FutureAdapter) {
  ResponseFuture future = ((FutureAdapter<?>) f).getFuture();
  RpcContext.getContext().setFuture(new FutureAdapter<>(new AsyncResponseFutureDelegate(future)));
}
```

在 2.7.0 版本中， AsyncRpcResult 在 recreate 的时候也会给 RpcContext 设置 Future。导致用户拿到的 Future 跟实际的不是同一个，客户端一直拿不到响应，请求被 hang 住。

```java
@Override
public Object recreate() throws Throwable {
    RpcInvocation rpcInvocation = (RpcInvocation) invocation;
    FutureAdapter future = new FutureAdapter(this);
    RpcContext.getContext().setFuture(future);
    if (InvokeMode.FUTURE == rpcInvocation.getInvokeMode()) {
        return future;
    }
```

2.7.0 对这部分的重构很好，支持了异步 Filter 链，通过 ListenableFilter 回调机制比现在的代码结构更清晰，可以把同步和异步埋点的逻辑进行统一整合。

```java
public abstract class ListenableFilter implements Filter {

    protected Listener listener = null;

    public Listener listener() { return listener; }
}
```

3.9 服务端无法指定客户端的调用方式

Issue：https://github.com/apache/dubbo/issues/3650

如果服务端设置了默认 ASYNC，升级到 2.7.0 版本后客户端会拿不到响应。例如：服务端配置了 async=true，客户端默认配置。

<dubbo:service interface="..." async="true">
<dubbo:reference interface="...">

2.5.10 版本的客户端，通过 client.sayHello() 会返回 null，RpcContext 的 Future 可以拿到响应。

基于 2.7.0 版本测试下来， client.sayHello 拿到了响应，但是 RpcContext 的 Future 却是 null。

经过研究发现，2.7.0 版本在 ClusterUtils 的 mergeUrl 过程中把服务端传递过来的 ASYNC_KEY 给删掉了，所以客户端仍然以同步方式去调用。

这个是新老版本兼容性的 Bug，已经在 2.7.2 修复了，验证下来没问题了。

3.10 @Service 注解无法设置 parameters 参数

Issue：https://github.com/apache/dubbo/issues/3778

用户通过 Annotation 方式启动服务，在 @Service 注解的 parameters 属性，服务端启动的时候拿不到用户配置的参数。

```
@Service(parameters = {"someKey","someValue"})
public class DemoServiceImpl implements DemoService {
}
```

并且报了下面这样的错误。

```
Caused by:
org.springframework.beans.ConversionNotSupportedException:
Failed to convert property value of type 'java.lang.String[]' to required type 'java.util.Map' for
property 'parameters';
nested exception is java.lang.IllegalStateException:
Cannot convert value of type 'java.lang.String[]' to required type 'java.util.Map' for property
'parameters':
no matching editors or conversion strategy found
```

2.7.0 版本对这部分机制进行了重构，BeanDefinitionBuilder 把这段 parameters 的参数转换代码给漏掉了。加上这段逻辑之后，我们测试下来已经 OK 了，这个问题已经在 2.7.2 解决掉了。

```
private AbstractBeanDefinition buildServiceBeanDefinition() {
    BeanDefinitionBuilder builder =rootBeanDefinition(ServiceBean.class);
    ...
    // Convert parameters into map

builder.addPropertyValue("parameters",convertParameters(serviceAnnotationAttributes.getS
tringArray("parameters")));
}
```

3.11 当客户端发现服务时出现异常，即使服务端启动后也不会恢复

Issue：https://github.com/apache/dubbo/issues/4068

API 方式比 XML 和 Annotation 更灵活，可以在不重启进程的情况下多次初始化客户端。

服务端没有启动的情况下，通过 API 的方式启动了客户端，这个时候客户端会报 No Provider 的错误。然后启动服务端，客户端通过 API 的方式再次初始化，仍然会报 No Provider 的错误。

```
ReferenceConfigCache cache = ReferenceConfigCache.getCache();
DemoService demoService = (DemoService) cache.get(reference);
```

通过翻阅 ReferenceConfig 的代码，服务发现的时候可能会抛异常导致直接跳出 init 过程，但是 initialized 标志位已经被置为 true 了，导致下次不会再重新初始化。

```
if (urls.size() == 1) {
    invoker = REF_PROTOCOL.refer(interfaceClass, urls.get(0));
} else {
    List<Invoker<?>> invokers = new ArrayList<>();
```

修复方案：只有在 init 方法的最后，客户端代理创建完成才会设置 initialized 为 true。

```
    ref = createProxy(map);

    String serviceKey = URL.buildKey(interfaceName, group, version);
    ApplicationModel.initConsumerModel(serviceKey, buildConsumerModel(serviceKey, attributes));
    initialized = true;
}
```

这个问题已经在 2.7.2 版本修复，验证下来已经 OK 了。

3.12 客户端服务发现失败，重试会有 OOM 的风险

Issue：https://github.com/apache/dubbo/issues/4107

如果服务端没有启动的情况下启动了客户端，客户端会报 No Provider 的错误，如果一直不停的重试可能会有 OOM 的风险。

Dubbo 在创建代理的时候会缓存 urls，每次启动失败都会把 url 加到 urls，但是由于 dubbo 的 URL 是有时间戳的，就导致 urls 队列不停的增长，甚至引起 Heap OOM 的风险。

```
if (url != null && url.length() > 0) { // user specified URL, could be peer-to-peer address, or register center's address.
    String[] us = SEMICOLON_SPLIT_PATTERN.split(url);
    if (us != null && us.length > 0) {
        for (String u : us) {
            URL url = URL.valueOf(u);
            if (StringUtils.isEmpty(url.getPath())) {
                url = url.setPath(interfaceName);
            }
            if (REGISTRY_PROTOCOL.equals(url.getProtocol())) {
                urls.add(url.addParameterAndEncoded(REFER_KEY, StringUtils.toQueryString(map)));
            } else {
                urls.add(ClusterUtils.mergeUrl(url, map));
            }
        }
    }
}
```

解决方案：每次创建代理之前，都把 urls 给予清空，这个问题已经在 2.7.2 中解决了。

总结：第一轮升级过程中大概历时 3 周左右，发现的几个 Issue 导致我们的 Test Case 无法继续下去，升级过程暂停了两三个月。

四、第二阶段升级及踩坑历程

直到六月中旬，阿里团队把上述几个问题在 2.7.2 修复了，我们重新开始了第二轮的升级过程。

4.1 性能测试，吞吐量下降了 40%

服务端：8C24G 的物理机，响应报文大小为 10Bytes，queue 设置为 -1 无界队列。
客户端：10 台 4C8G 的 Docker，请求报文大小也是 10Bytes。

基于原生的 2.5.10 版本，我们的压测环境下可以达到 8 万 QPS 左右。由于 CDubbo 扩展了熔断、配置、监控等功能，吞吐量下降到 5.5 万 QPS 左右。

升级到 2.7.2 版本后，最高只压测到 3 万多，吞吐量下降了差不多 40% 左右。

这个问题是因为 JDK 1.8 的 Bug 导致的，JDK 1.8 的 CompletableFuture 在 get 时会等到 256 次 countDown 执行完毕，影响了性能。

Issue：https://github.com/apache/dubbo/issues/4279

总结：第二阶段遇到的性能下降的问题肯定要解决后才可以上线，问题反馈给阿里团队后，他们需要讨论新的 Hotfix 发布机制。

五、第三阶段升级及踩坑历程

这次改变了合作模式，跟阿里团队基于 2.7.3-SNAPSHOT 版本一起讨论，一起修复，一起验证。下面的几个问题都是基于 SNAPSHOT 验证过程中发现的问题，并且在正式版中修复掉了。

5.1 对于同步的请求，方法级超时不生效

Issue：https://github.com/apache/dubbo/issues/4435

如果服务级设置的 timeout 为 1000ms，sayHello 方法设置的 timeout 为 800ms。理论上来说，sayHello 方法的请求应该在 800ms 就会超时了，但是实际上我们发现直到 1000ms 才会超时。

```
<dubbo:reference id="demoService" interface="com.ctrip.Demo" timeout="1000">
  <dubbo:method name="sayHello">
    <dubbo:parameter key="timeout" value="800"/>
  </dubbo:method>
</dubbo:reference>
```

同步的请求是在 AsyncToSyncInvoker 中执行了同步等待，修复前的代码如下，取的是整个服务的超时时间，也就是 1000ms。

```
asyncResult.get(getUrl().getParameter(TIMEOUT_KEY,DEFAULT_TIMEOUT),TimeUnit.MILLISECONDS);
```

解决方案：同步请求，除了 AsyncToSyncInvoker 在 get 时被设置了超时时间，

DubboInvoker 的 CompletableFuture 也被设置了超时时间。其实，只要一个地方能够超时就足够了，所以 AsyncToSyncInvoker 被设置到 Integer.MAX_VALUE 永不超时，所有的超时机制都通过 CompletableFuture 实现。

5.2 异步超时的情况下，不会回调 listener 的 onError 方法，导致埋点丢失

Issue：https://github.com/apache/dubbo/issues/4152
https://github.com/apache/dubbo/issues/4306

在修复前的版本中，ProtocolFilterWrapper 的 Filter 链中，只处理了正常的 onResponse 响应，并没有处理 onError 情况，就导致异常发生时不会回调 ListenableFilter 的 onError 方法。

修复后，会对正常响应和异常响应进行回调。

```
try {
    if (t == null) {
        listener.onResponse(r, filterInvoker, invocation);
    } else {
        listener.onError(t, filterInvoker, invocation);
    }
} catch (Throwable filterError) {
    t = filterError;
}
```

5.3 @Reference 注解的方式，客户端不会初始化

Issue：https://github.com/apache/dubbo/issues/4330

基于 2.7.2 版本，如果用的 Annotation 的方式，先要把 DubboComponentScan 换成 Apache 的，不然编译时就会因为找不到 class 而报错 。

如果客户端的 @Reference 用的还是 Alibaba 的 package，所拿到的 proxy 代理是 null，导致 service.sayHello 调用时抛 NPE 的 exception。

```
import com.alibaba.dubbo.config.annotation.Reference;

@RestController
public class DubboController {

  @Reference
  public CDubboDemo2Service service;

  @RequestMapping("/")
  public String get(@RequestParam("id") String msg) throws Exception {
    return service.sayHello(msg);
  }
}
```

这个问题，主要是由于 Apache 的 DubboComponentScan 没有兼容 Alibaba 的 @Reference 注解，目前 2.7.3 正式版实现了对 Alibaba 的 @Reference 和 @Service 的兼容。

5.4 服务端 executes 限流失效

Issue：https://github.com/apache/dubbo/issues/4277

我们的测试场景把服务的 executes 设置为 1，然后客户端多线程发起请求到服务端。第一次发起的多线程请求，只有一个请求能通过，符合预期。第二次再发起多线程请求，所有请求都通过了，并没有被限流。

```
<bean id="demoService" class="com.xxx."/>
<dubbo:service interface="com.xxx" ref="demoService" executes="1">
</dubbo:service>
```

这是因为服务端抛异常的时候，除了正常请求结束后释放掉的计数器，异常处理时又减了一次，之后的限流一直处于失效的状态，所有请求都可以通过了。

这个问题已经在 2.7.3 解决了，解决方案就是在 onError 的时候不要重复减。

```
@Override
public void onError(Throwable t, Invoker<?> invoker, Invocation invocation) {
    if (t instanceof RpcException) {
        RpcException rpcException = (RpcException)t;
        if (rpcException.isLimitExceed()) {
            return;
        }
    }
    RpcStatus.endCount(invoker.getUrl(), invocation.getMethodName(), getElapsed(invocation), false);
}
```

5.5 现有服务框架生成的 ListenableFutre 异步服务接口，Dubbo 无法支持

携程现有几千个 SOA 服务，服务端异步用的是 Guava 的 ListenableFuture，但是 2.7.0 支持的服务端异步用的是 CompletableFuture，这就导致现有服务接口迁移过来，无法支持 Dubbo 协议的服务端异步了。

针对这个问题，我们想到了几个方案。

方案 1：让 Dubbo 既支持 CompletableFuture 又支持 ListenableFuture。首先，需要 Dubbo 支持 ListenableFuture，这个改动成本比较高。其次，对用户多一个选择也会提高他们的学习成本，以及犯错的概率。

方案 2：只支持 CompletableFuture。如果用户从 SOA 服务迁移到 CDubbo 框架，就需要

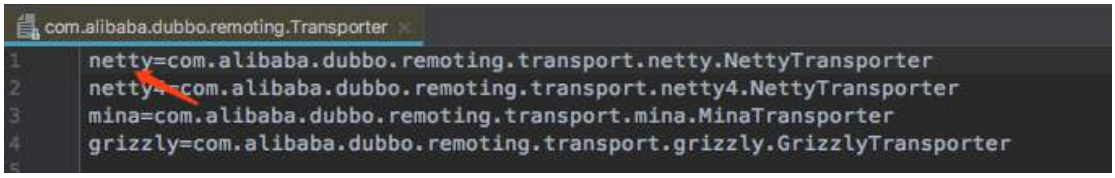
把服务接口的 Future 类型改为 CompletableFuture。

最终跟业务沟通下来，选择了方案 2，业务迁移到 CDubbo 的时候手工修改服务接口的 Future 类型。

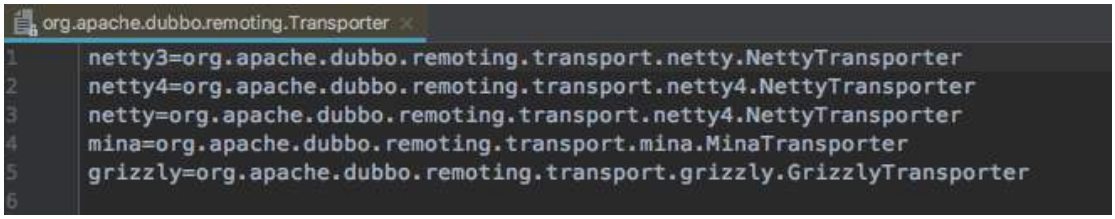
5.6 服务端新版本，客户端老版本，报 Netty3 找不到的异常

这个问题的根因是前面监控打点失败，我们把 Netty 默认版本降回了 Netty3。服务端 2.7.3 版本，客户端 2.5.10 版本的情况下会报 Netty3 找不到的异常。

在 2.5.10 版本中， Netty3 在 resource 配置文件中的名字叫 netty，具体如下图：



但是，2.7.3 版本把 Netty3 在 resource 配置文件中的名字改成了 netty3，而不是 netty 了。



服务端注册的时候会包括 Netty 版本，通过注册中心推送到了客户端，客户端的 2.5.10 版本不存在 netty3 的资源文件，通过 SPI 加载的时候因为找不到 netty3 而报错了。

解决方案：Netty 的版本不应该被推送到客户端，我们修改了动态配置的推送规则，不允许 Netty 参数推送到客户端，问题就解决了。

六、兼容性测试

第三轮测试把所有的 Test Case 都通过了，接着我们手工验证了新老版本的兼容性测试。以下场景都是基于服务端升级 2.7.3，客户端仍然是 2.5.10 场景下的测试验证。

6.1 注册发现机制

服务端可以正常注册到注册中心，客户端也可以发现到新版本的服务端。

6.2 同步请求是否正常

如果服务端返回的是 Response 对象，客户端以同步的方式可以正常调用。

6.3 异步请求是否正常

如果服务端返回的是 Response 对象，客户端以异步的方式可以正常调用。

6.4 监控打点

除了不支持 CompletableFuture，其他都正常。

6.5 服务端升级到新版本，客户端老版本，在超时场景下的异常测试

因为服务端抛的是 org.apache.dubbo.rpc.RpcException，这个 package 在 2.5.10 版本中是不存在的，就会报 java.lang.ClassNotFoundException 的错误。

这个错误似乎也没法避免，这也是我们优先升级 Dubbo 2.7.3 的原因，我们要忍受这种阵痛，等全部升级完就不存在这个问题了。

七、性能压测

兼容性测试也通过了，我们紧接着开始了第二轮压力测试：(基于 2.7.3-SNAPSHOT)

从服务端的压测数据来看，在低于 4 万 QPS 的时候性能没啥区别，在 5 万左右的时候响应时间有所下降，主要是由于 YGC 导致的。

2.5.10 服务端

QPS	Avg(ms)	P95(ms)	P99.9(ms)	CPU 使用率(%)	YGC(ms)
10000	0.1	1	1	23	25~35
20000	0.1	1	2	40	50-70
30000	0.5	2	5	55	90-110
40000	0.5	2	13	67	110-130
50000	1	3	19	78	150-200

2.7.3 服务端

QPS	Avg(ms)	P95(ms)	P99.9(ms)	CPU 使用率(%)	YGC(ms)
10000	0.1	1	1	23	25~50
20000	0.2	1	2	41	60~70
30000	0.5	2	7	56	90~110
40000	0.8	2	16	69	110~130
50000	1.6	5.4	36.5	80	150~170

从客户端的性能来看，吞吐量基本没啥变化，响应时间在 5000QPS 的时候下降稍微有点明显，主要也是 GC 导致的。

2.5.10 客户端

QPS	Avg(ms)	P95(ms)	P99.9(ms)	CPU 使用率(%)	YGC(ms)
1000	0	0	4	11	0~10
2000	0	0	8.2	19	0~20
3000	0.5	2	14	30	10~20
4000	1.3	4.2	29	36	10~30
5000	1.9	5	54	44	20~35

2.7.3 客户端

QPS	Avg(ms)	P95(ms)	P99.9(ms)	CPU 使用率(%)	YGC(ms)
1000	0	0	5	16	0~500
2000	0.1	0.4	10.2	18	500~800
3000	0.8	3.1	18.4	35	500~1000
4000	1.6	5.4	36.5	50	600~1200
5000	2.4	5.2	73.5	52	1200~2000

八、集成测试

到现在为止，CDubbo 单个组件已经完成了所有 Test Case，兼容性测试也全部通过了，压力测试的结果勉强可以接受。公司有一些中间件也会依赖 Dubbo，除了这些组件要升级 Dubbo 到 2.7.3，我们还遇到其他一些问题。

8.1 ApplicationConfig 的冲突再次出现

前面只是解决了 CDubbo 单个组件的 ApplicationConfig 冲突问题，在一个组件中保证只会有一个 ApplicationConfig。但是，不同组件在暴露本地服务的时候也需要设置 ApplicationConfig，用户可能会只引用一个组件，也可能两个同时引用，无法保证不同组件只初始化一个 ApplicationConfig。

看了 ApplicationConfig 的 equals 方法，可以知道冲突是因为 name 不一致，我们只要保证 name 一致就行了。

```
@Override
public boolean equals(Object obj) {
    if (obj == null || !(obj.getClass().getName().equals(this.getClass().getName()))) {
        return false;
    }

    Method[] methods = this.getClass().getMethods();
    for (Method method1 : methods) {
        if (MethodUtils.isGetter(method1) && ClassUtils.isPrimitive(method1.getReturnType())) {
            Parameter parameter = method1.getAnnotation(Parameter.class);
            if (parameter != null && parameter.excluded()) {
                continue;
            }
            try {
                Method method2 = obj.getClass().getMethod(method1.getName(), method1.getParameterTypes());
                Object value1 = method1.invoke( obj: this, new Object[]{});
                Object value2 = method2.invoke(obj, new Object[]{});
                if ((value1 != null && value2 != null) && !value1.equals(value2)) {
                    return false;
                }
            } catch (Exception e) {
                return true;
            }
        }
    }
    return true;
}
```

8.2 开源和订制版的冲突

在携程，大部分业务用的是我们提供的开源版的 Dubbo，还有部分业务使用的是基于 Dubbo 代码直接修改过的订制版本。

因为，我们这次是公司级的升级，用了订制版 Dubbo 的应用，如果引入公司其他中间件，这些中间件又依赖了开源版的 Dubbo，就会导致业务的应用类冲突。

对于这个问题，没有统一的解决方案，需要跟业务同事进行讨论来解决。

8.3 服务端启动时端口连不上

Issue：https://github.com/apache/dubbo/issues/4775

在集成测试时，服务端用的默认协议，客户端通过 20880 端口发起的连接，结果客户端报连接失败。后来看了下服务端的 20880 端口的确没有打开，本地打开的端口号是 20xxx。

经过调试代码发现拿到的默认协议是 QSchedule 组件设置的 ProtocolConfig。看下 ConfigManager 的代码，addProtocol 的时候会把第一个协议作为默认协议缓存下来了，之

后再 getDefaultProtocol 的时候拿到的并不是默认的协议了。

```
public void addProtocol(ProtocolConfig protocolConfig) {
    ...
    if (protocols.containsKey(key)&& !protocolConfig.equals(protocols.get(key))) {
        logger.warn("···");
    } else {
        protocols.put(key, protocolConfig);
    }
}
```

这个问题可以把 ProtocolConfig 默认值设置为 false，就不会被 put 到 protocols 作为默认协议了。但是，对于不知道背景的同学可能还是会掉坑里，不过这个问题会在 2.7.4 版本中修复。

九、感兴趣的几个话题

写到这里，我们已经通过了 Test Case、回归测试、压力测试和集成测试，发布了 SNAPSHOT 版本给到业务同事去试用，预计九月初会发布正式版本。

除了我们踩到的坑，下面可能也是你感兴趣的话题。

9.1 注册中心

注册中心，我们在去年落地 2.5.10 的时候就扩展了携程自己的注册中心。

服务端注册实例信息到注册中心，每隔 5s 发送一次心跳来续约，如果注册中心 30s 没有收到心跳，会将其从注册中心反注册，并通知到客户端。

客户端向注册中心发起订阅，当注册信息发生变化时会通过长连接推送到客户端。

这套机制是基于 2.5.10 扩展的，在升级 2.7.3 的过程中没有任何变更，可以完全兼容 2.7.3，服务端和客户端都可以正常的注册和发现。

9.2 从一个中心拆分成三中心

1）注册中心：前面已经提到，升级 2.7.3 没有变更，可以完全兼容。

2）配置中心：CDubbo 的 0.2.0 版本，为了接入携程自己的配置中心，就已经通过 override 协议实现了动态配置方案，这套机制目前没有发现问题，所以这次没有对接 2.7.3 提供的动态配置推送能力。方案可以参考：http://dubbo.apache.org/zh-cn/docs/user/demos/config-rule-deprecated.html

3）元数据中心：TCP 协议的测试不像 HTTP 协议那么方便，服务提供者必须得自己写个

Client 才能测试验证，大多数业务同事都反馈过这个痛点。

在 2019 年 1 月份的时候，CDubbo 对接了携程的测试平台，支持 Dubbo 协议的测试。当时为了透明升级 2.7.0 版本，就已经提前把元数据中心的代码拷贝到内部的版本了，这次升级 2.7.3 版本很平滑，没有发现有啥问题。

9.3 为什么敢于升级大版本

业界对大版本升级的普遍做法就是，等其他大厂试试看，或者等发布几个 hotfix 之后再考虑。携程在这次升级过程中有一套自己的保障，事实也证明我们的单元测试和集成测试在 2.7.3 升级过程中发挥了重要作用。

1）单元和集成测试覆盖率 93%：刚开始落地 CDubbo 的时候就非常重视测试覆盖率。目前为止，我们的测试覆盖率仍然达到了 93%。

在这次升级过程中，很多藏的很深的 Bug 都是通过我们的测试代码发现的，前面谈到的升级过程中遇到的 Bug 基本都是通过测试代码发现的。不但保证了质量，也提高了我们升级的效率。

2）Benchmark 压测：我们有一套稳定的压测环境，服务端是一台 8C24G 的物理机，客户端是 10 台 4C8G 的 Docker 机器。服务器都比较稳定，测试结果也能真实准确的反应出性能的问题。

每次发布新功能的时候，都要经过 Benchmark 至少 5 万 QPS 以上持续一天的稳定性压测。

携程 Redis 跨 IDC 多向同步实践

【作者简介】祝辰，携程框架架构研发部资深研发工程师，主要负责 Redis 跨站点容灾方面的工作，目前致力于研究分布式系统中的一致性问题以及相关理论和解决方案。此前曾就职于 EMC 混合云部门。对底层技术比较感兴趣，乐于研究操作系统和各种数据库的实现思路。

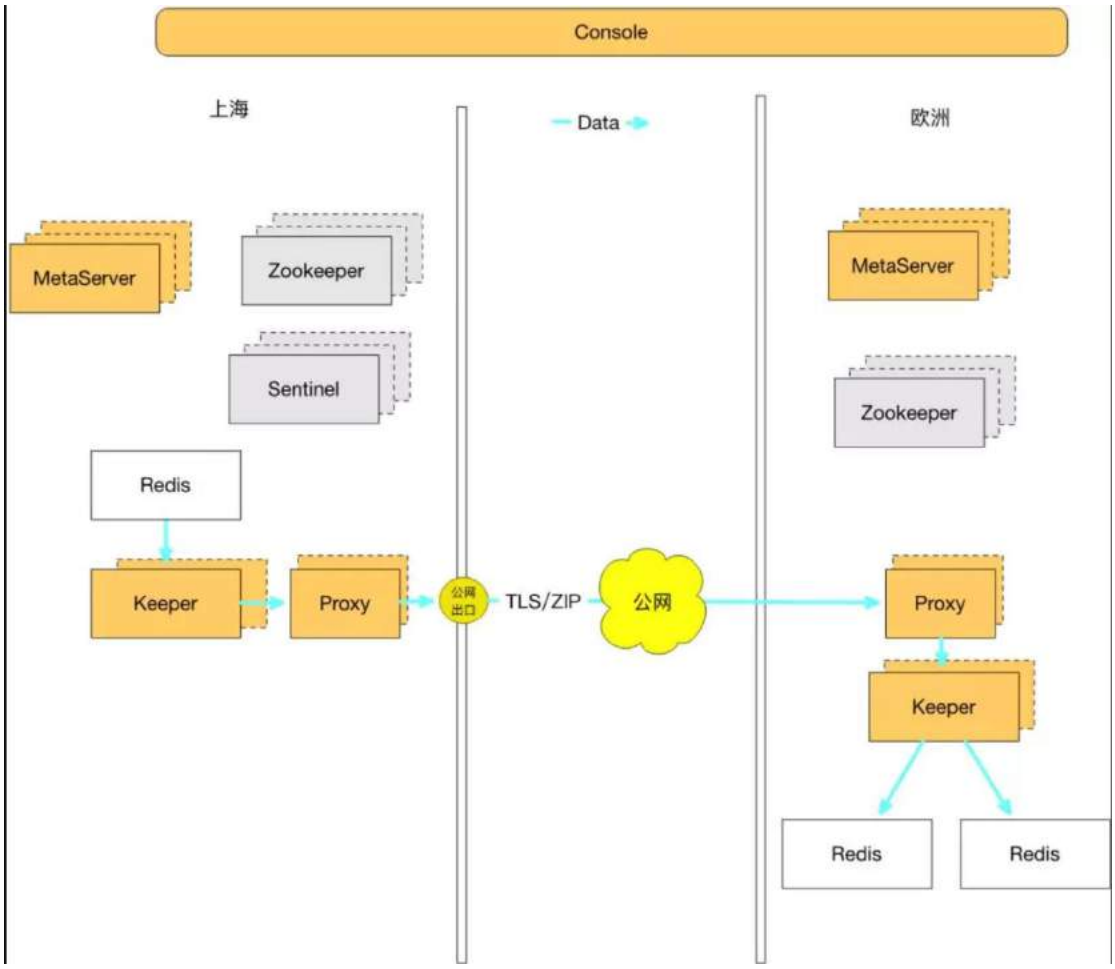
一、前言

跨 DC（数据中心）的数据同步是企业提升容灾实力的必备手段。随着携程业务向海外发展的速度越来越快，应用架构能够快速全球部署的能力也愈发重要。对于服务而言，我们可以尽量做到无状态的部署架构，来达到灵活拓展，快速部署的目的，比如 server-less。

然而，对于业务应用来讲，大量的业务逻辑是有状态的，仅仅做到服务的灵活拓展还不够，需要数据也拥有多站点共享的能力。

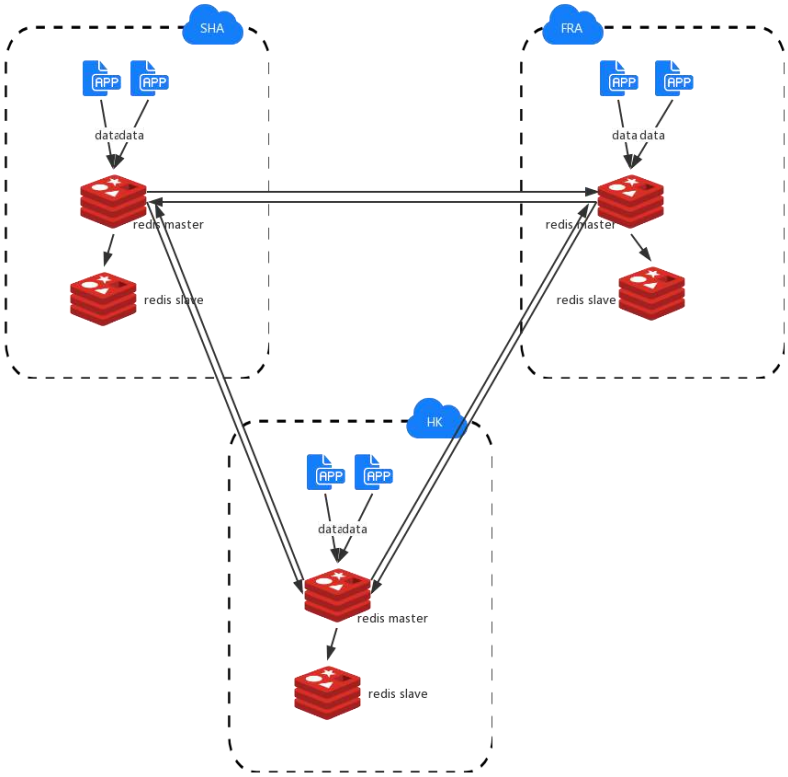
携程在一年前已经可以实现 Redis 单向跨区域同步，从上海将数据复制到美国加州，德国法兰克福，以及新加坡等众多海外站点，延迟稳定在 180ms 左右，支持单个 Redis 5MB/s 的传输带宽。

关于跨公网传输以及跨区域单向复制，详情参考这篇文章：携程 Redis 海外机房数据同步实践。



但是，技术人的目标就是不断突破自己。在 Redis 出海同步一年之后，业务上有了新的需求：能否在每个站点都可以独立地写入和读取，所有数据中心之间互相同步，而跨区域复制的一致性等问题，也可以由底层存储来解决？

单向同步的 Redis 固然可以解决问题，然而，大量的海外数据需要先回流上海，再从上海同步至各个数据中心，这一来一去，不仅给业务开发带来了额外的复杂性和代码的冗余性，也给数据本身的时效性以及跨区域传输的费用带来了问题（携程的数据回源需要走专线，而 Redis 同步通过公网传输，由携程的框架中间件 - XPipe 来解决跨公网传输的安全和不稳定性因素）。[1]



二、技术选型

所以，我们需要的是一个分布式的 Redis 存储，能够实现跨区域多向同步。面对大型分布式系统，不免要讨论 CAP 理论，在跨区域多活的场景下如何取舍？

显然 P（网络分区）是首要考虑因素。其次，跨区域部署就是为了提高可用性，而且对于常见的一致性协议，不管是 2PC、Paxos 还是 raft，在此场景下都要做跨区域同步更新，不仅会降低用户体验，在网络分区的时候还会影响可用性。

对于 Redis 这种毫秒级相应的数据库，应用希望能够在每个站点都可以“如丝般顺滑”地使用，因此 C 必定被排在最后。

那是不是 C 无法被满足了呢？事实并非如此，退而求其次，最终一致也是一种选择。经过调研，我们决定选用“强最终一致性”的理论模型来满足一致性的需求。[2]

关于“最终一致性”(Eventually Consistency) 和“强最终一致性”(Strong Eventually Consistency)，大家可以参考 wiki 百科给出的释义：

(Strong Eventually Consistency) Whereas eventual consistency is only a liveness guarantee (updates will be observed eventually), strong eventual consistency (SEC) adds the safety guarantee that any two nodes that have received the same (unordered) set of updates will be in the same state. If, furthermore, the system is monotonic, the application will never suffer rollbacks.

三、问题

有了目标，自然是开始计划和设计，那么在开始之前有什么问题呢?

3.1 跨数据中心双向同步共同的问题

各种数据库在设计双向数据同步时，均会遇到的问题:

1）复制回源：A -> B -> A
数据从 A 复制到 B，B 收到数据后，再回源复制给 A 的问题。

2）环形复制：A -> B -> C -> A
我们可以通过标记来解决上一问题，然而系统引入更多节点之后，A 发送到 B 的数据，可能通过 C 再次回到 A。

3）数据一致性
网络传输具有延迟性和不稳定性，多节点的并发写入会造成数据不一致的问题。

4）同步时延（鉴于是跨国的数据同步，这一项我们先忽略）

3.2 Redis 的问题

1）Redis 原生的复制模型，是不能够支持 Multi-Master 的理论架构的。
开源版的 Redis 只能支持 Master-Slave 的架构，并不能支持 Redis 之间互相同步数据。

2）Redis 特殊的同步方式（全量同步+增量同步），给数据一致性带来了更多挑战。
Redis 全量同步和增量同步都基于 replicationId + offset 的方式来做，在引入多个节点互相同步之后，如何对齐互相之间全量同步和增量同步的 offset 是一个巨大的问题。

3）同时支持原生的 Master-Slave 系统
在新版系统上，同时兼容现存的 Master-Slave 架构，两种同步方式和策略的异同，也带来了新的挑战。

四、解决方案

由于篇幅的限制，这里只对数据一致性的解决方案做下介绍，对于分布式系统或者是双向同步感兴趣的同学，可以关注我们后续的技术文章和技术大会。

4.1 一致性的解决方案

CRDT（Conflict-Free Replicated Data Type）[3] 是各种基础数据结构最终一致算法的理论总结，能根据一定的规则自动合并，解决冲突，达到强最终一致的效果。

2012 年 CAP 理论提出者 Eric Brewer 撰文回顾 CAP[4]时也提到，C 和 A 并不是完全互斥，建议大家使用 CRDT 来保障一致性。自从被大神打了广告，各种分布式系统和应用均开始尝试 CRDT，redislabs[5]和 riak[6]已经实现多种数据结构，微软的 CosmosDB[7]也在 azure 上使用 CRDT 作为多活一致性的解决方案。

携程的框架部门最终也选择站在巨人的肩膀上，通过 CRDT 这种数据结构，来实现自己的 Redis 跨区域多向同步。

4.2 CRDT

CRDT 同步方式有两种：

1）state-based replication
发送端将自身的“全量状态”发送给接收端，接收端执行“ merge” 操作，来达到和发送端状态一致的结果。state-base replication 适用于不稳定的网络系统，通常会有多次重传。要求数据结构能够支持 associative（结合律）/commutative（交换律）/idempotent（幂等性）。

2）operation-based replication
发送端将状态的改变转换为“操作”发送给接收端，接收端执行“update”操作，来达到和发送端状态一直的结果。op-based replication 只要求数据结构满足 commutative 的特性，不要求 idempotent（大家可以想一想为什么）。op-based replication 在接收到 client 端的请求时，通常分为两步进行操作：

a. prepare 阶段：将 client 端操作转译为 CRDT 的操作；
b. effect 阶段：将转译后的操作 broadcast 到其他 server；

两者之间在实现上，界限比较模糊。一方面，state-based replication 可以通过发送 delta 减少网络流量，从而做到和 op-based replication 比较接近的效果；另一方面，op-based replication 可以通过发送 compact op-logs 将操作全集发送过去，来解决初始化的时候同步问题，从而达到类似于 state-based replication 的效果。

我们的系统需要借助两种同步方式，以适用于不同的场景中：

1) state-based replication 通常是基于“全量状态”进行同步，这样的结果是造成的网络流量太大，且同步的效率低下。在同步机制已经建立的系统中，我们更倾向于使用 op-based replication，以达到节省流量和快速同步的目的。

2) op-based replication 是基于 unbounded resource 的假设上进行论证的学术理念，在实践过程中，不可能有无限大的存储资源，将某个站点的全部数据缓存下来。

这样就带来一个问题，如果新加节点或者网络断开过久时，我们的存储资源不足以缓存所有值钱的操作，从而使得复制操作无法进行。此时，我们需要借助 state-based replication 进行多个站点之间，状态的 merge 操作。

4.3 CRDT 的数据结构

Redis 的 String 类型对应的操作有 SET, DEL, APPEND, INCRBY 等等，这里我们只讲一下 SET 操作（INCRBY 会是不同的数据类型）。

Register

先来讲一下，CRDT 理论中如何处理 Redis String 类型的同步问题。

Redis 的 String 类型对应于 CRDT 里面的 Register 数据结构，对应的具体实现有两种比较符合我们的应用场景：

- MV(Multi-Value) Register：数据保留多份副本，客户端执行 GET 操作时，根据一定的规则返回值，这种类型比较适合 INCRBY 的整型数操作。
- LWW(Last-Write-Wins) Register：数据只保留一份副本，以时间戳最大的那组数据为准，SET 操作中，我们使用这种类型。

还记得上文提到的两种不同的同步方式么，关于两种不同的同步方式，对于 LWW Register，实现方式会稍有不同。

Op-based LWW Register [8]

Specification 9 Op-based LWW-Register

```
payload X x, timestamp t                          ▷ X: some type
    initial ⊥, 0
query value () : X w
    let w = x
update assign (X x')
    atSource () t'
        let t' = now()                            ▷ Timestamp
    downstream (x', t')                           ▷ No precond: delivery order is empty
        if t < t' then x, t := x', t'
```

State-based LWW Register [8]

Specification 8 State-based Last-Writer-Wins Register (LWW-Register)

```
1:  payload X x, timestamp t                      ▷ X: some type
2:      initial ⊥, 0
3:  update assign (X w)
4:      x, t := w, now()                          ▷ Timestamp, consistent with causality
5:  query value () : X w
6:      let w = x
7:  compare (R, R') : boolean b
8:      let b = (R.t ≤ R'.t)
9:  merge (R, R') : payload R''
10:     if R.t ≤ R'.t then R''.x, R''.t = R'.x, R'.t
11:     else R''.x, R''.t = R.x, R.t
```

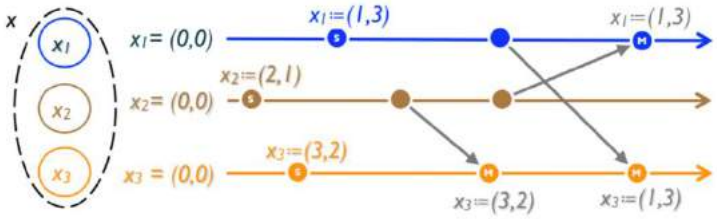


Figure 7: Integer LWW Register (state-based). Payload is a pair (value, timestamp)

4.4 CRDT Register 在 Redis 中的落地

讲完了 CRDT 的传输类型和一个基本的数据结构，那么具体这样的理论是如何落地到 Redis 中间的呢?

在最终的实现中，我们采用了 OR-Set(Observed-Remove Set) + LWW(Last-Write-Wins) Register 来实现 Redis 中的 String 操作。

4.4.1 Redis K/V

以下是理论上的数据结构，并不是 redis 中真正的结构体，仅仅作为说明使用。

```
struct CRDT.Register {
        string key;
        string val;
        TAG delete-tag;
        int timestamp;
}
```

1）key 既是 SET 操作中的 key
2）val 用来存储相应的 value
3）delete-tag 是逻辑删除的标记位，具体的理论来源是 OR(Observed-Remove) Set
4）timestamp 用于 LWW(Last Write Wins)机制，来解决并发冲突

由于目前我们的多写 Redis 还没有开源，这里我们拿 Java 程序举个栗子[9] 详细可以访问 github。

LWW Register

processCommand 是这个 CRDT 框架的核心函数，基本定义了每一种类型，是如何进行 merge/update 等操作的。

在这里我们可以看到，每一个 command 过来时，会携带一个自身的时钟，由本地的程序进行判定，如果时钟符合偏序(partial ordered)，就进行 merge 操作，并储存元素。

java
public class LWWRegister extends
AbstractCrdt<LWWRegister, LWWRegister.SetCommand> {

private T value;
private StrictVectorClock clock;

public LWWRegister(String nodeId, String crdtId) {
    super(nodeId, crdtId, BehaviorProcessor.create());
    this.clock = new StrictVectorClock(nodeId);
}

protected Option<SetCommand<T>> processCommand(SetCommand<T> command) {
    if (clock.compareTo(command.getClock()) < 0) {
        clock = clock.merge(command.getClock());
        doSet(command.getValue());
        return Option.of(command);
    }
    return Option.none();
}

public T get() {
    return value;
}

public void set(T newValue) {
    if (! Objects.equals(value, newValue)) {
        doSet(newValue);
        commands.onNext(new SetCommand<>(
                crdtId,
                value,
                clock
        ));
    }
}

private void doSet(T value) {
    this.value = value;
    clock = clock.increment();
}

}

OR-SET

而 Observed-Remove SET 相较 LWW-Register 就复杂一些，主要涉及两个概念，一个是正常的原生储存的地方，在 Set<Element<E>> elements 中。而另一个重要的概念，是 tombstone（墓地），用来存放会删除的元素 Set<Element<E>> tombstone OR-SET 的 tombstone，就可以解决并发删除的问题，而 LWW-Register 则可以解决并发添加的问题。

```
java ublic class ORSet extends AbstractSet implements
Crdt<ORSet, ORSet.ORSetCommand> /*, ObservableSet */
{private final String crdtId;
private final Set<Element<E>> elements = new HashSet<>();
private final Set<Element<E>> tombstone = new HashSet<>();
private final Processor<ORSetCommand<E>, ORSetCommand<E>> commands =
ReplayProcessor.create();

public ORSet(String crdtId) {
    this.crdtId = Objects.requireNonNull(crdtId, "Id must not be null");
}

@Override
public String getCrdtId() {
    return crdtId;
}

@Override
public void subscribe(Subscriber<? super ORSetCommand<E>> subscriber) {
    commands.subscribe(subscriber);
}

@Override
public void subscribeTo(Publisher<? extends ORSetCommand<E>> publisher) {
    Flowable.fromPublisher(publisher).onTerminateDetach().subscribe(command -> {
        final Option<ORSetCommand<E>> newCommand =
processCommand(command);
        newCommand.peek(commands::onNext);

    });
}

private Option<ORSetCommand<E>> processCommand(ORSetCommand<E> command) {
    if (command instanceof AddCommand) {
        return doAdd(((AddCommand<E>) command).getElement())?
Option.of(command) : Option.none();
    } else if (command instanceof RemoveCommand) {
```

```
            return doRemove(((RemoveCommand<E>) command).getElements())?
Option.of(command) : Option.none();
        }
        return Option.none();
}

@Override
public int size() {
        return doElements().size();
}

@Override
public Iterator<E> iterator() {
        return new ORSetIterator();
}

@Override
public boolean add(E value) {
        final boolean contained = doContains(value);
        prepareAdd(value);
        return !contained;
}

private static <U> Predicate<Element<U>> matches(U value) {
        return element -> Objects.equals(value, element.getValue());
}

private synchronized boolean doContains(E value) {
        return elements.parallelStream().anyMatch(matches(value));
}

private synchronized Set<E> doElements() {
        return elements.parallelStream().map(Element::getValue).collect(Collectors.toSet());
}

private synchronized void prepareAdd(E value) {
        final Element<E> element = new Element<>(value, UUID.randomUUID());
        commands.onNext(new AddCommand<>(getCrdtId(), element));
        doAdd(element);
}

private synchronized boolean doAdd(Element<E> element) {
        return (elements.add(element) | elements.removeAll(tombstone)) &&
(!tombstone.contains(element));
```

```
}

private synchronized void prepareRemove(E value) {
    final Set<Element<E>> removes =
elements.parallelStream().filter(matches(value)).collect(Collectors.toSet());
    commands.onNext(new RemoveCommand<>(getCrdtId(), removes));
    doRemove(removes);
}

private synchronized boolean doRemove(Collection<Element<E>> removes) {
    return elements.removeAll(removes) | tombstone.addAll(removes);
}

}
```

4.4.2 用户视角

正常同步的场景

Data Type: Strings
Use Case: Common SETs Conflict Resolution: None

Time	InstanceA	InstanceB
t1	SET 'key1' 'val1'	
t2	sync	sync
t3	GET 'key1' => 'val1'	GET 'key1' => 'val1'
t4		SET 'key1' 'val2'
t5	sync	sync
t6	GET 'key1' => 'val2'	GET 'key1' => 'val2'

并发冲突的场景

Data Type: Strings
Use Case: Concurrent SETs
Conflict Resolution: Last Write Wins (LWW)

Time	InstanceA	InstanceB
t1	SET 'key1' 'val1'	
t2		SET 'key1' 'val2'
t3	sync	sync
t4	GET 'key1' => 'val2'	GET 'key1' => 'val2'

五、未完待续

由于篇幅限制，我们详细介绍了 CRDT 同步的理论基础以及一个 Redis 的 K/V 数据结构在 CRDT 中是如何展现出来的。对 CRDT 或分布式数据库感兴趣的同学，请关注携程的公众号，也可以支持一下我们目前已经开源的的 redis 同步产品 -- XPipe (https://github.com/ctripcorp/x-pipe)。

附录

[1] 携程 Redis 海外机房数据同步实践
[2] CRDT——解决最终一致问题的利器
[3] CRDT: https://hal.inria.fr/inria-00609399/document
[4] Eric Brewer: https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed
[5] redislabs, Developing Applications with Geo-replicated CRDBs on Redis Enterprise Software(RS): https://redislabs.com/redis-enterprise-documentation/developing/crdbs/
[6] riak: https://docs.basho.com/riak/kv/2.0.0/developing/data-types/
[7] cosmosDB: https://docs.microsoft.com/en-us/azure/cosmos-db/multi-region-writers
[8] A comprehensive study of Convergent and Commutative Replicated Data Types(https://links.jianshu.com/go?to=http%3A%2F%2Fhal.upmc.fr%2Ffile%2Findex%2Fdocid%2F555588%2Ffilename%2Ftechreport.pdf)
[9] CRDT Java: https://github.com/netopyr/wurmloch-crdt

携程的 Dubbo 之路

【作者简介】董艺荃，携程框架架构研发部技术专家。目前负责携程服务化框架的研发工作。本文来自董艺荃在 Dubbo 社区开发者日上的分享。

一、缘起

携程当初为什么要引入 Dubbo 呢？

实际上从 2013 年底起，携程内主要使用的就是基于 HTTP 协议的 SOA 微服务框架。这个框架是携程内部自行研发的，整体架构在这近 6 年中没有进行大的重构。受到当初设计的限制，框架本身的扩展性不是很好，使得用户要想自己扩展一些功能就会比较困难。

另外，由于 HTTP 协议一个连接同时只能处理一个请求。在高并发的情况下，服务端的连接数和线程池等资源都会比较紧张，影响到请求处理的性能。

而 Dubbo 作为一个高性能的 RPC 框架，不仅是一款业界知名的开源产品，它整体优秀的架构设计和数据传输方式也可以解决上面提到的这些问题。正好在 2017 年下半年，阿里宣布重启维护 Dubbo 。基于这些原因，我们团队决定把 Dubbo 引入携程。

二、Dubbo 落地第一步

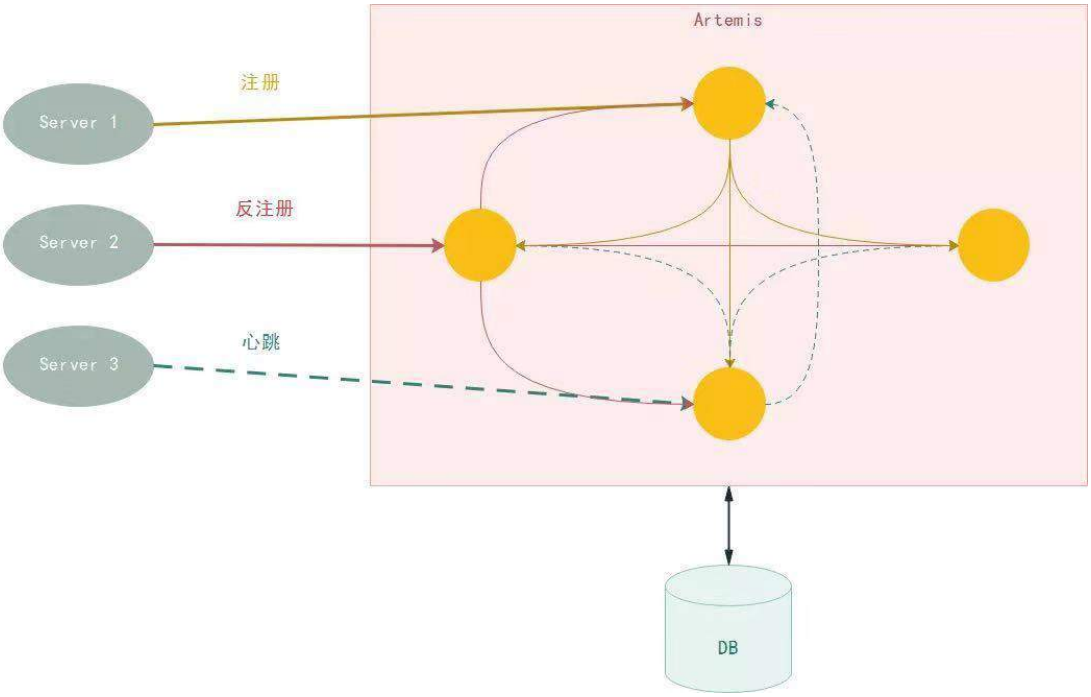
要在公司落地 Dubbo 这个新服务框架，第一步就是解决服务治理和监控这两个问题。

2.1 服务治理

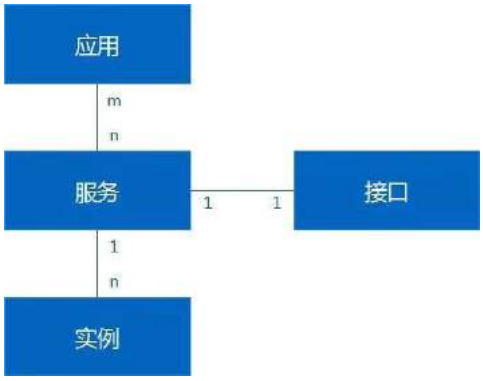
在服务治理这方面，携程现有的 SOA 框架已经有了一套完整的服务注册中心和服务治理系统。

对于服务注册中心，大家比较常用的可能是 Apache Zookeeper 。而我们使用的是参考 Netflix 开源的 Eureka 自行研发的注册中心 Artemis 。

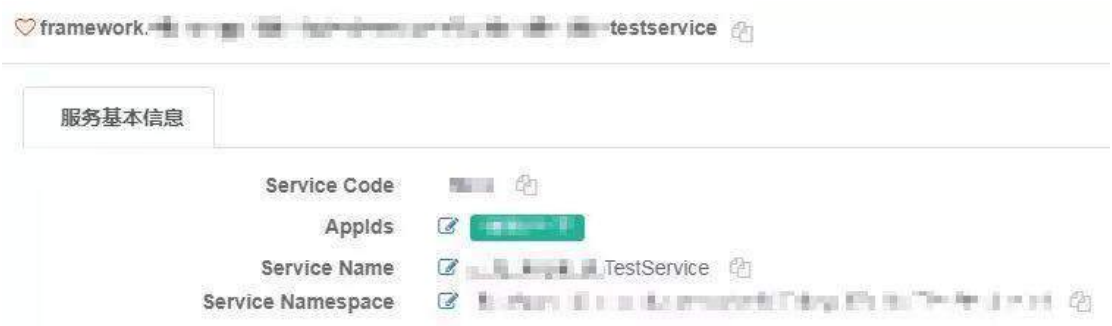
Artemis 的架构是一个去中心的对等集群。各个节点的地位相同，没有主从之分。服务实例与集群中的任意一个节点保持长连接，发送注册和心跳信息。收到信息的节点会将这些信息分发给其他节点，确保集群间数据的一致性。客户端也会通过一个长连接来接受注册中心推送的服务实例列表信息。



在服务数据模型方面，我们直接复用了现有 SOA 服务的数据模型。如图所示，最核心的服务模型对应的是 Dubbo 中的一个 interface 。一个应用程序内可以包含多个服务，一个服务也可以部署在多个服务器上。我们将每个服务器上运行的服务应用称为服务实例。



所有的服务在上线前都需要在治理系统中进行注册。注册后，系统会为其分配一个唯一的标识，也就是 ServiceID 。这个 ServiceID 将会在服务实例注册时发送至注册中心用来标识实例的归属，客户端也需要通过这个 ID 来获取指定服务的实例列表。



由于 Dubbo 本身并没有 ServiceID 的设计，这里的问题就是如何向注册中心传递一个 interface 所对应的 ServiceID 信息。我们的方法是在 Service 和 Reference 配置中增加一个 serviceId 参数。ArtemisServiceRegistry 的实现会读取这个参数，并传递给注册中心。这样就可以正常的与注册中心进行交互了。

```
<!-- 服务端 -->
<dubbo:service interface="com.ctrip.cdubbo.demo.CDubboTestService" ref="testService">
    <dubbo:parameter key="serviceId" value="framework.xxx.yyy.zzz.testservice"/>
</dubbo:service>

<!-- 客户端 -->
<dubbo:reference id="testService" interface="com.ctrip.cdubbo.demo.CDubboTestService" init="true">
    <dubbo:parameter key="serviceId" value="framework.xxx.yyy.zzz.testservice"/>
</dubbo:reference>
```

2.2 服务监控

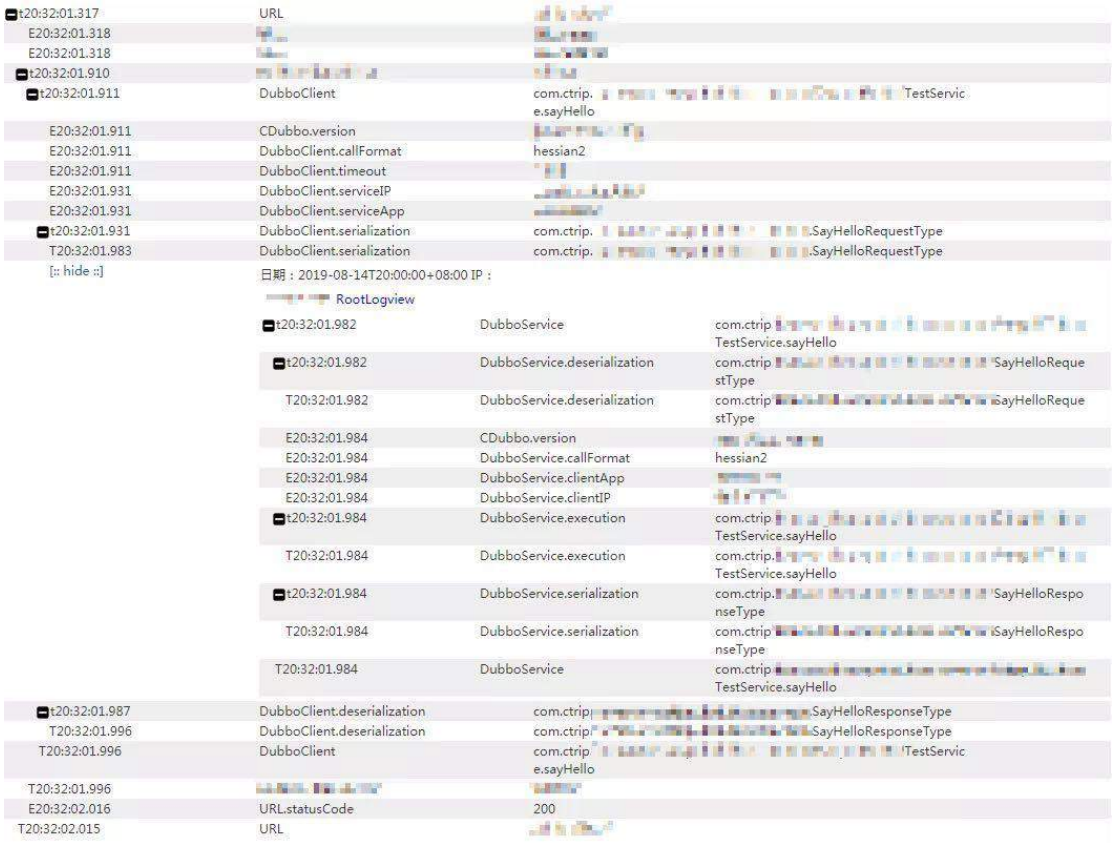
在服务监控这方面我们主要做了两部分工作：统计数据层面的监控和调用链层面的监控。

统计数据指的是对各种服务调用数据的定期汇总，比如调用量、响应时间、请求体和响应体的大小以及请求出现异常的情况等等。这部分数据我们分别在客户端和服务端以分钟粒度进行了汇总，然后输出到 Dashboard 看板上。同时我们也对这些数据增加了一些标签，例如：Service ID、服务端 IP 、调用的方法等等。用户可以很方便的查询自己需要的监控数据。

在监控服务调用链上，我们使用的是 CAT 。CAT 是美团点评开源的一个实时的应用监控平台。它通过树形的 Transaction 和 Event 节点，可以将整个请求的处理过程记录下来。

我们在 Dubbo 的客户端和服务端都增加了 CAT 的 Transaction 和 Event 埋点，记录了调用的服务、SDK 的版本、服务耗时、调用方的标识等信息，并且通过 Dubbo 的 Attachment 把 CAT 服务调用的上下文信息传递到了服务端，使得客户端和服务端的监控数据可以连接起来。在排障的时候就可以很方便的进行查询。

在图上，外面一层我们看到的是客户端记录的监控数据。在调用发起处展开后，我们就可以看到对应的在服务端的监控数据。



2.3 初版发布

在解决了服务治理和监控对接这两个问题后，我们就算完成了 Dubbo 在携程初步的一个本地化，在 2018 年 3 月，我们发布了 Dubbo 携程定制版的首个可用版本。在正式发布前我们需要给这个产品起个新名字。既然是携程（Ctrip）加 Dubbo ，我们就把这个定制版本称为 CDubbo 。

三、CDubbo 功能扩展

除了基本的系统对接，我们还对 CDubbo 进行了一系列的功能扩展，主要包括以下这 5 点：Callback 增强、序列化扩展、请求熔断、服务测试工具、堡垒测试网关。

下面我来逐一给大家介绍一下。

3.1 Callback 增强

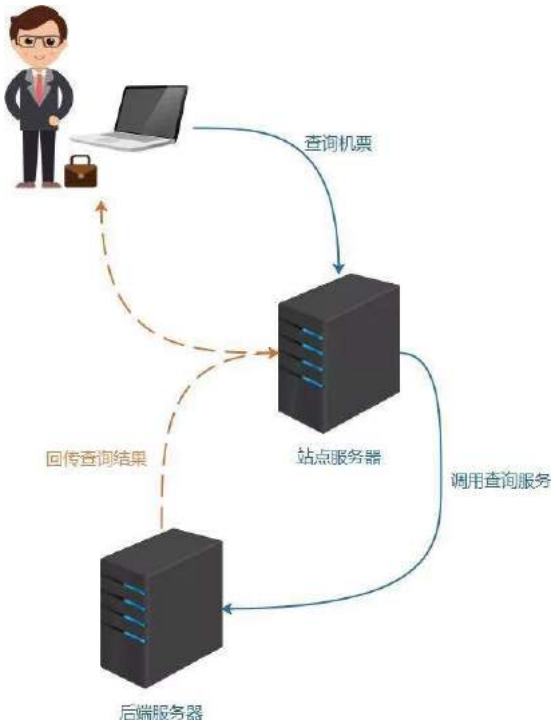
首先，我们看一下这段代码。请问代码里有没有什么问题呢？

```
// Callback接口
public interface MessageCallback {
    void onMessage(String message);
}

// 服务接口
public interface DemoService {
    void callbackDemo(MessageCallback callback);
}

public class Demo {
    private static final Logger logger = LoggerFactory.getLogger(Demo.class);

    private final MessageCallback callback = new MessageCallback() {
        @Override
        public void onMessage(String message) {
            System.out.println(message);
        }
    };
    private DemoService demoService;

    void foo() {
        demoService.callbackDemo(callback);
    }

    void bar() {
        demoService.callbackDemo(new MessageCallback() {
            @Override
            public void onMessage(String message) {
                logger.info(message)
            }
        });
    }
}
```

这段代码里有一个 DemoService 。其中的 callbackDemo 方法的参数是一个接口。下面的 Demo 类中分别在 foo 和 bar 两个方法中调用了这个 callbackDemo 方法。

相信用过 Callback 的朋友们应该知道，foo 这个方法的调用方式是正确的，而 bar 这个方法在重复调用的时候是会报错的。因为对于同一个 Callback 接口，客户端只能创建一个实例。

但这又有什么问题呢？我们来看一下这样一个场景。



一个用户在页面上发起了一个查询机票的请求。站点服务器接收到请求之后调用了后端的查询机票服务。考虑到这个调用可能会耗时较长，接口上使用了 callback 来回传实际的查询结果。然后再由站点服务器通过类似 WebSocket 的技术推送给客户端。

那么问题来了。

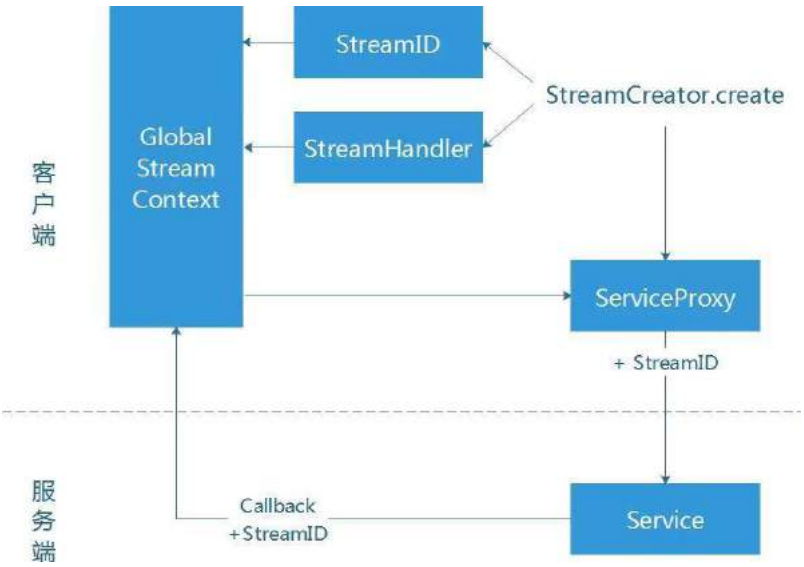
站点服务器接受到回调数据时，需要知道它对应的是哪个用户的哪次调用请求，这样才能把数据正确的推送给用户。但对于全局唯一的 callback 接口实例，想要拿到这个请求上下文信息就比较困难了。需要在接口定义和实现上预先做好准备。可能需要额外引入一些全局的对象来保存这部分上下文信息。

针对这个问题，我们在 CDubbo 中增加了 Stream 功能。跟前面一样，我们先来看代码。

```
public interface DemoService {
    void callbackDemo(StreamContext<String> stream);
}

public class Demo {
    private DemoService demoService;

    void bar() {
        StreamContext<String> streamContext = StreamCreator.create(StreamOptions.newBuilder().build()
                , new StreamHandler<String>() {
                    @Override
                    public void onNext(String message) {
                        System.out.println("Message received: " + message);
                    }

                    @Override
                    public void onError(Throwable e) {
                        System.out.println("Something goes wrong with the stream: " + e);
                    }

                    @Override
                    public void onCompleted() {
                        System.out.println("Message receiving is completed.");
                    }
                });
        demoService.callbackDemo(streamContext);
    }
}
```

这段代码与前面的代码有什么区别？首先， callback 接口的参数替换为了一个 StreamContext 。还有接受回调的地方不是之前的全局唯一实例，而是一个匿名类，并且也不再是单单一个方法，而是有 3 个方法，onNext、onError 和 onCompleted 。这样调用方在匿名类里就可以通过闭包来获取原本请求的上下文信息了。是不是体验就好一些了？

那么 Stream 具体是怎么实现的呢？我们来看一下这张图。



在客户端，客户端发起带 Stream 的调用时，需要通过 StreamContext.create 方法创建一个 StreamContext。虽然说是创建，但实际是在一个全局的 StreamContext 中登记一个唯一的 StreamID 并关联对应回调的实际处理逻辑。在发送请求时，这个 StreamID 会被发送到服务端。服务端在发起回调的时候也会带上这个 StreamID 。这样客户端就可以知道这次回调对应的是哪个 StreamContext 了。

3.2 序列化扩展

携程的一些业务部门，在之前开发 SOA 服务的时候，使用的是 Google Protocol Buffer 的契约编写的请求数据模型。Google PB 的要求就是通过契约生成的数据模型必须使用 PB 的序列化器进行序列化。

为了便于他们将 SOA 服务迁移到 Dubbo ，我们也在 Dubbo 中增加了 GooglePB 序列化方式的支持。后续为了便于用户自行扩展，我们在 PB 序列化器的实现上增加了扩展接口，允许用户在外围继续增加数据压缩的功能。

整体序列化器的实现并不是很难，倒是有一点需要注意的是，由于 Dubbo 服务对外只能暴露一种序列化方式，这种序列化方式应该兼容所有的 Java 数据类型。而 PB 碰巧就是那种只能序列化自己契约生成的数据类型的序列化器。所以在遇到不支持的数据类型的时候，我们还是会 fallback 到使用默认的 hessian 来进行序列化操作的。

3.3 请求熔断

相信大家对熔断应该不陌生。当客户端或服务端出现大范围的请求出错或超时的时候，系统会自动执行 fail-fast 逻辑，不再继续发送和接受请求，而是直接返回错误信息。

这里我们使用的是业界比较成熟的解决方案：Netflix 开源的 Hystrix 。它不仅包含熔断的功能，还支持并发量控制、不同的调用间隔离等功能。单个调用的出错不会对其他的调用造成影响。各项功能都支持按需进行自定义配置。CDubbo 的服务端和客户端通过集成 Hystrix 来做请求的异常情况进行处理，避免发生雪崩效应。

3.4 服务测试工具

Dubbo 作为一个使用二进制数据流进行传输的 RPC 协议，服务的测试就是一个比较难操作的问题。要想让测试人员在无需编写代码的前提下测试一个 Dubbo 服务，我们要解决的有这样三个问题：如何编写测试请求、如何发送测试请求和如何查看响应数据。

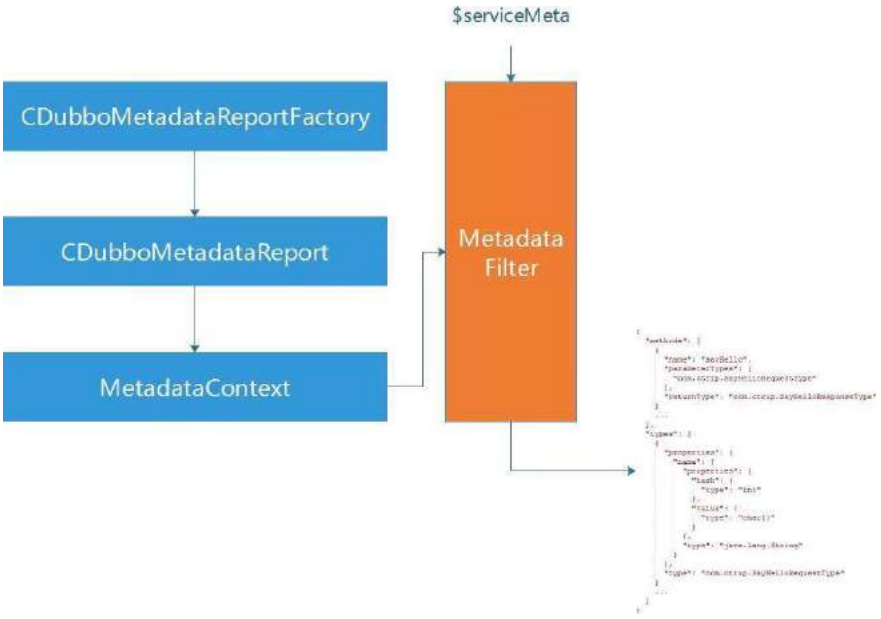
首先就是怎么构造请求。这个问题实际分为两个部分。一个是用户在不写代码的前提下用什么格式去构造这个请求。考虑到很多测试人员对 Restful Service 的测试比较熟悉，所以我们最终决定使用 JSON 格式表示请求数据。

那么让一个测试人员从一个空白的 JSON 开始构造一个请求是不是有点困难呢？所以我们还是希望能够让用户了解到请求的数据模型。虽然我们使用的是 Dubbo 2.5.10 ，但这部分功能在 Dubbo 2.7.3 中已经有了。

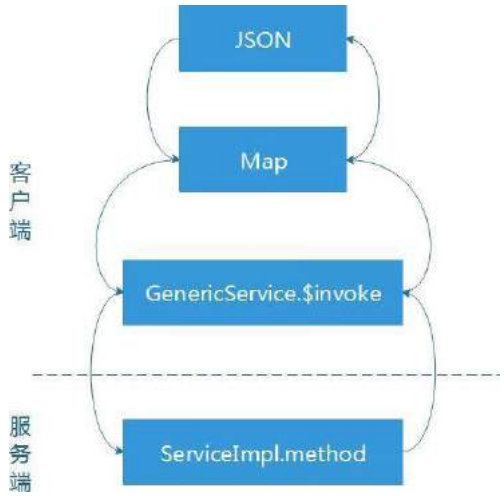
所以我们将这部分代码复制了过来，然后对它进行了扩展，把服务的元数据信息保存在一个全局上下文中。并且我们在 CDubbo 中通过 Filter 增加了一个内部的操作，$serviceMeta，把服务的元数据信息暴露出来。

这部分元数据信息包括方法列表、各个方法的参数列表和参数的数据模型等等。这样用户通

过调用内部操作拿到这个数据模型之后，可以生成出一个基本的 JSON 结构。之后用户只需要在这个结构中填充实际的测试数据就可以很容易的构造出一个测试请求来。



然后，怎么把编辑好的请求发送给服务端呢？因为没有模型代码，无法直接发起调用。而 Dubbo 提供了一个很好的工具，就是泛化调用， GenericService 。我们把请求体通过泛化调用发送给服务端，再把服务端返回的 Map 序列化成 JSON 显示给测试人员。整个测试流程就完成了。顺便还解决了如何查看响应数据的问题。



为了方便用户使用，我们开发了一个服务测试平台。用户可以在上面直接选择服务和实例，编写和发送测试请求。另外为了方便用户进行自动化测试，我们也把这部分功能封装成了 jar 包发布了出去。

其实在做测试工具的过程中，还遇到了一点小问题。通过从 JSON 转化 Map 再转化为 POJO 这条路是能走通的。但前面提到了，有一些对象是通过类似 Google Protobuf 的契约生成的。它们不是单纯的 POJO ，无法直接转换。所以，我们对泛化调用进行了扩展。

首先对于这种自定义的序列化器，我们允许用户自行定义从数据对象到 JSON 的格式转换实现。其次，在服务端处理泛化调用时，我们给 Dubbo 增加了进行 JSON 和 Google PB 对象之间的互相转换的功能。现在这两个扩展功能有已经合并入了 Dubbo 的代码库，并随着 2.7.3 版本发布了。

3.5 堡垒测试网关

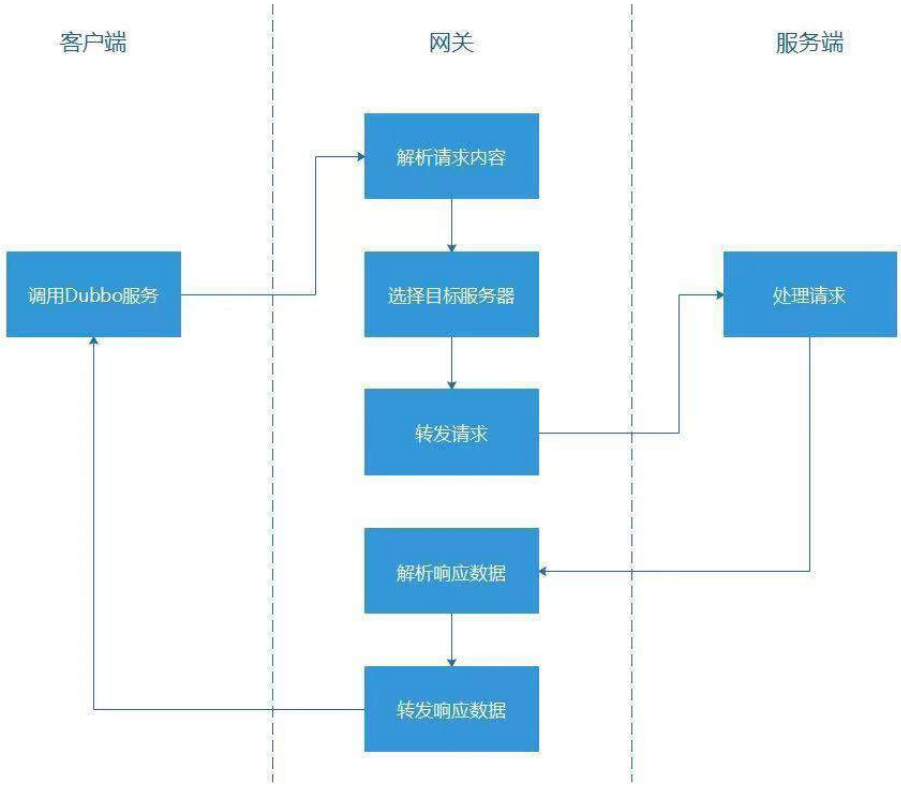
说完了单纯针对服务的测试，有些时候我们还希望在生产的实际使用环境下对服务进行测试，尤其是在应用发布的时候。

在携程有一个叫堡垒测试的测试方法，指的是在应用发布过程中，发布系统会先挑出一台服务器作为堡垒机，并将新版本的应用发布到堡垒机上。然后用户通过特定的测试方法将请求发送到堡垒机上来验证新版本应用的功能是否可以正常工作。

由于进行堡垒测试时，堡垒机尚未拉入集群，这里就需要让客户端可以识别出一个堡垒测试请求并把请求转发给指定的堡垒服务实例。虽然我们可以通过路由来实现这一点，但这就需要客户端了解很多转发的细节信息，而且整合入 SDK 的功能对于后续的升级维护会造成一定的麻烦。

所以我们开发了一个专门用于堡垒测试的服务网关。当一个客户端识别到当前请求的上下文

中包含堡垒请求标识时，它就会把 Dubbo 请求转发给预先配置好的测试网关。网关会先解析这个服务请求，判断它对应的是哪个服务然后再找出这个服务的堡垒机并将请求转发过去。在服务完成请求处理后，网关也会把响应数据转发回调用方。



与一般的 HTTP 网关不同， Dubbo 的服务网关需要考虑一个额外的请求方式，就是前面所提到的 callback 。

由于 callback 是从服务端发起的请求，整个处理流程都与客户端的正常请求不同。网关上会将客户端发起的连接和网关与服务端之间的连接进行绑定，并记录最近待返回的请求 ID 。这样在接收到 callback 的请求和响应时就可以准确的进行路由了。

四、后续功能规划

截止到今天，CDubbo 一共发布了 27 个版本。携程的很多业务部门都已经接入了 Dubbo 。

在未来， CDubbo 还会扩展更多的功能，比如请求限流和认证授权等等。我们希望以后可以贡献更多的新功能出来，回馈开源社区。

跨多业务线挑战下，携程订单索引服务的 1.0 到 2.0

【作者简介】唐巍，携程用户平台部订单服务组资深后端开发，在互联网尤其是移动互联网方面有丰富的经验，目前主要负责 OrderIndex 的维护和架构升级工作。

经过团队几个月的努力，我们最近终于完成了 OI（订单索引服务）从 1.0 到 2.0 升级的里程碑，上线了新的数据同步平台和对应的数据查询服务。在这里我们总结了其中的经验和心得，希望能给大家，尤其是有做跨多业务线或者复杂系统需要升级改造的同学们，一点启发或者是帮助。

一、什么是 OI?

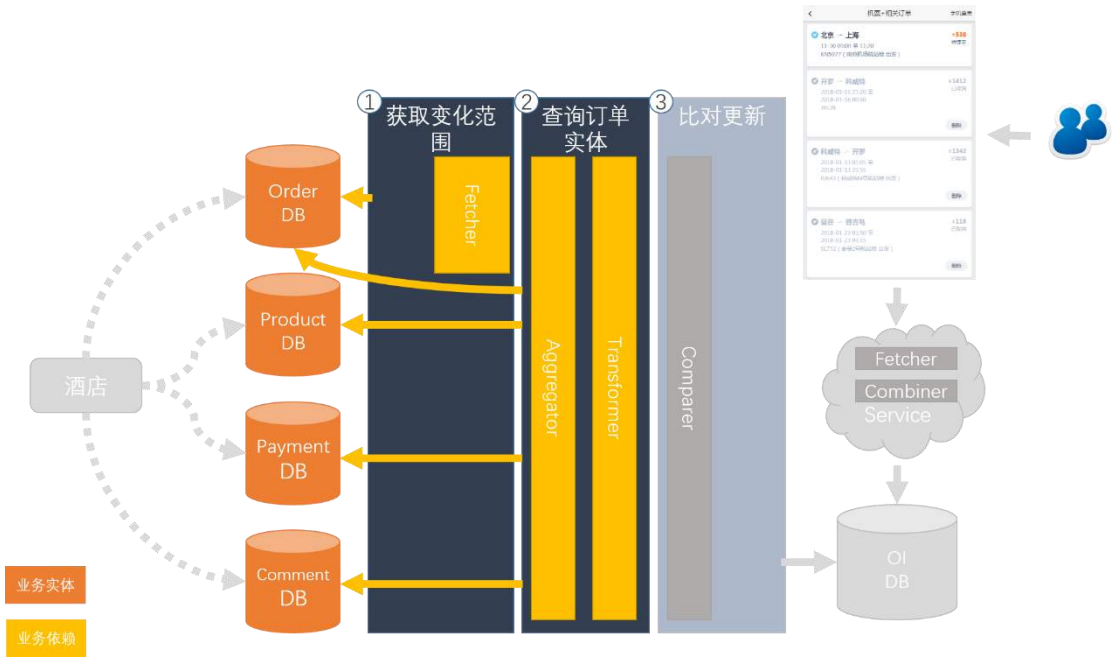
携程的众多业务线订单信息分布在各业务线不同的订单系统之中，有各自独立的查询服务，而公司内部又存在着大量跨业务线查询统一订单信息的诉求，为解决这样的痛点，OI 项目应运而生。

OI 的全称是 OrderIndex（订单索引服务），是“携程 APP-我的携程-订单列表”的订单信息数据源，是一个以“聚合不同数据来源的订单信息，并提供统一分类查询功能”为核心业务的系统。

二、OI 1.0

OI1.0 是目前支撑 OI 业务的主力系统，上线于 2013 年。查询服务基于公司的 SOA 服务治理平台开发。订单同步和辅助 Job 基于公司的作业调度平台（JosWS）开发。

其中核心机制如下图。



OI 1.0 的订单同步机制

数据同步以及下发流程如下：

1）对应业务线订单的订单同步 Job，从业务线的订单数据库通过扫描相关表的时间戳，来感知订单变化（获取变化的订单号 orderid）；
2）通过业务线提供的拉取订单详情的 SQL 从业务线订单数据库读取订单详情相关数据；
3）根据业务线提供的业务，将从业务线数据库拉取的订单详情相关数据转化为实际的订单数据，并规整后保存到 OI 的数据库中；
4）在 OI 的接口调用方通过 SOA 服务来查询订单信息的时候，将我们同步过来的订单信息透传下发；

基于这样的架构和机制，截止目前，OI 1.0 聚合了超过 50+数据源，160+业务线的订单数据，总量超过 5 亿条。并通过实时数据查询服务以及订单变更消息通知以毫秒级的订单同步延迟，支撑了下游超过 200 个系统共计日均 4 亿+次的订单查询需求，并且，这些数字还在不断的变大。可以预见，OI 在之后还会承担更重要的责任。

虽然目前 OI 1.0 看起来仍然运行良好，但是随着 OI 业务的不断扩展，我们也发现了一些问题。

1）业务线提供的数据源只支持直连 DB，并且需要提供的接入信息非常复杂

- 需要 Db 提供生产核心订单库的访问权限，有安全风险
- 需提供所有相关表的表结构以及字段说明，并提供跟实际订单信息之间的关联转化逻辑
- 绝大部分情况下，订单的信息变更必须反映到订单主表的时间戳变更上，否则无法感知到订单变化

2）业务线提供的订单数据源结构各不相同，还需结合配套业务使用

- 订单接入和修改需要我方产品、开发、和测试人员理解业务线的所有相关业务，并理解其原始数据到 OI 数据的转化过程，沟通成本和出错率高，响应也相对较慢（上线周期长），易出错。
- 由于各接入方数据不标准，验证数据和业务的正确性需要熟悉业务的开发、测试人员人工进行，比对点和工作量都很大。

3）大量的订单接入方（业务线），和 OI 服务的调用方希望我们接入大量的非标准字段，由于旧系统的各种限制，往往难以支持或者周期漫长

- 1.0 系统并不支持扩展字段的添加，所有的扩展字段添加都需要定制开发（从指定 db 指定表的指定字段获取数据，再经过某种业务进行处理，最后落我们空余的某个 db 字段），若无空余字段，则无法支持
- 由于 1.0 系统中我们的字段大多有固定含义，所以能借用存放的空余字段不多
- 由于借用空余字段的存放，所以下发的部分字段，在不同场景下有不同的含义，数据使用方需知业务线原始业务，以及 OI 的字段命名映射逻辑（OI 下发的数据字段名跟业

务方字段名不同）

4）OI 感知订单变化依赖于业务线提供的 Sql 定制开发，出于对性能考虑通常只能扫描订单主表的变更，若订单变更有独立于订单主表之外的，感知订单变化的部分的实现会特别复杂。

- 一旦业务线修改感知订单变化的业务或者新加渠道，OI 都需要重新修改代码，或者新增 Job 支持
- 订单直接物理删除，感知不到变化，需业务线配合，新增物理删除订单信息同步

5）同步 Job 依赖于公司的 Job 调度平台，由于调度平台的限制，

- 每新增一个 Job 都需要开发一个对应的 Job 类，并发布代码，流程长，风险大
- 若一个实例需要多活，需要在 Job 调度平台中针对每个实例，单独手工配置 Job 调度任务

6）公司内的合作部门，有数据接入 OI 的需求，但是因为目前系统的实现机制评估下来工作量，以现有的人力完全无法支撑。

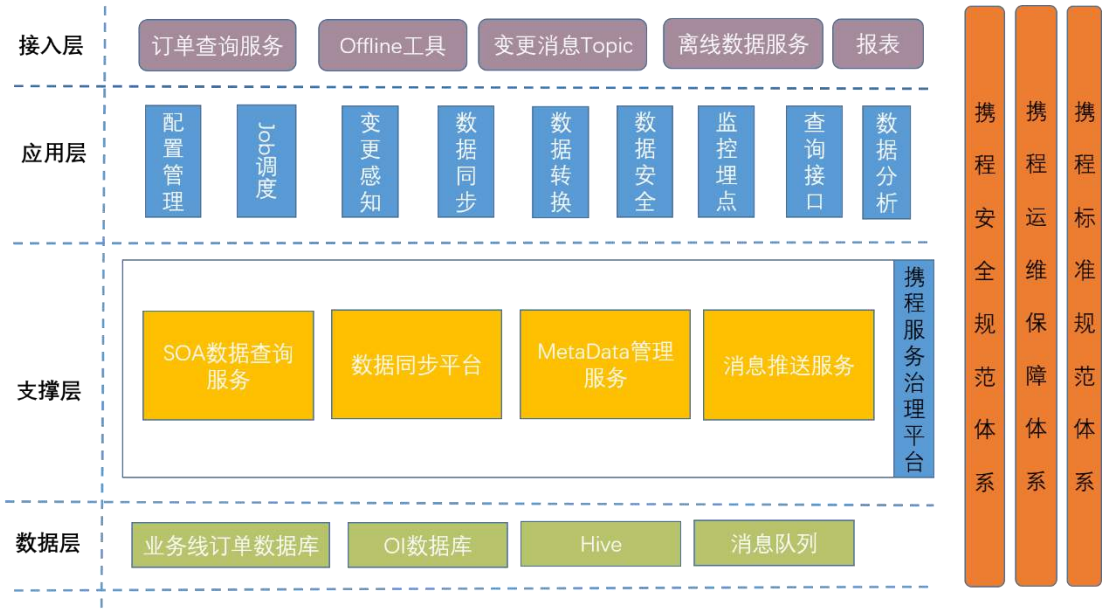
下图是我们迫切需要解决的问题：

于是我们决定对 OI 进行改造升级，彻底解决这些问题。

三、OI 2.0

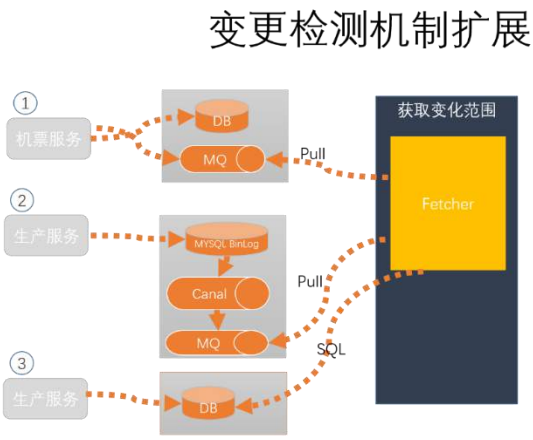
首先，我们对于 OI 进行了重新定义：OI 是一个提供了基于标准化流程接入，针对订单数据提供统一汇聚、检索、输出、管理的数据平台。

基于全新的定位，重新设计了 OI，新的架构如下：



OI 2.0 的系统架构

主要改造方向有如下几点：



针对订单变更检测重新设计，提供新的接入方案。

1）业务线订单服务在更新订单时推送变更消息。

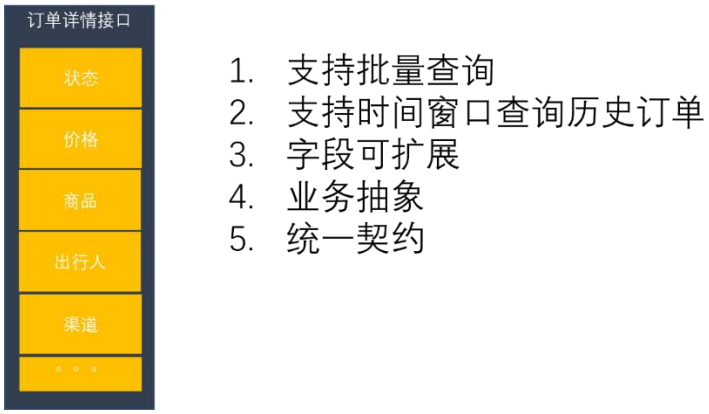
优点：时延最短
缺点：需业务线配合，开发成本高

2）基于订单数据库相关表的 Binlog 通过 Canal 组件推送变更消息。

优点：时延可以接受<500ms，开发成本极低
缺点：中间环节多，故障点增多，并且只支持 mysql 数据库

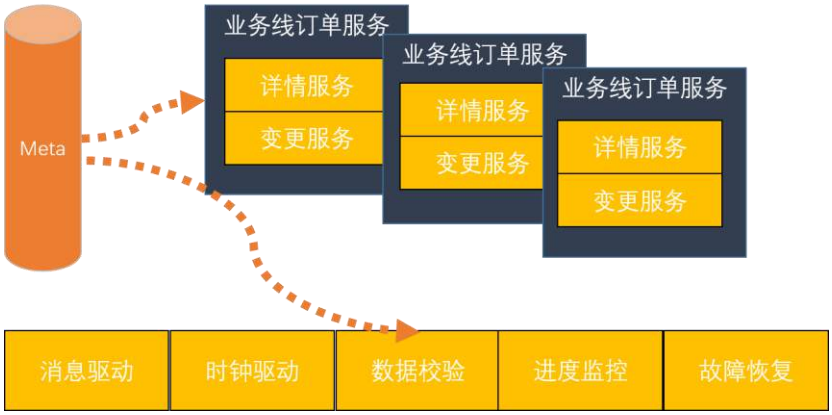
3）扩展了基于 sql 的变更检测，无需业务线再提供 sql，而是提供关联 db 信息。

优点：中间环节少，时延较低(<200ms)
缺点：耦合高，依赖业务线数据库访问权限



订单详情标准服务化改造

我们针对获取订单详情方式进行了重新设计，抛弃了过去通过 sql 直连 db 读取数据，在同步 Job 中进行转化的模式，重新抽象了订单的基础 MetaData，并以此为基础设计了一套新的标准详情服务接口契约，以此将业务进行抽像，将具体的业务实现从 OI 中剥离了出来，返还给了业务线。

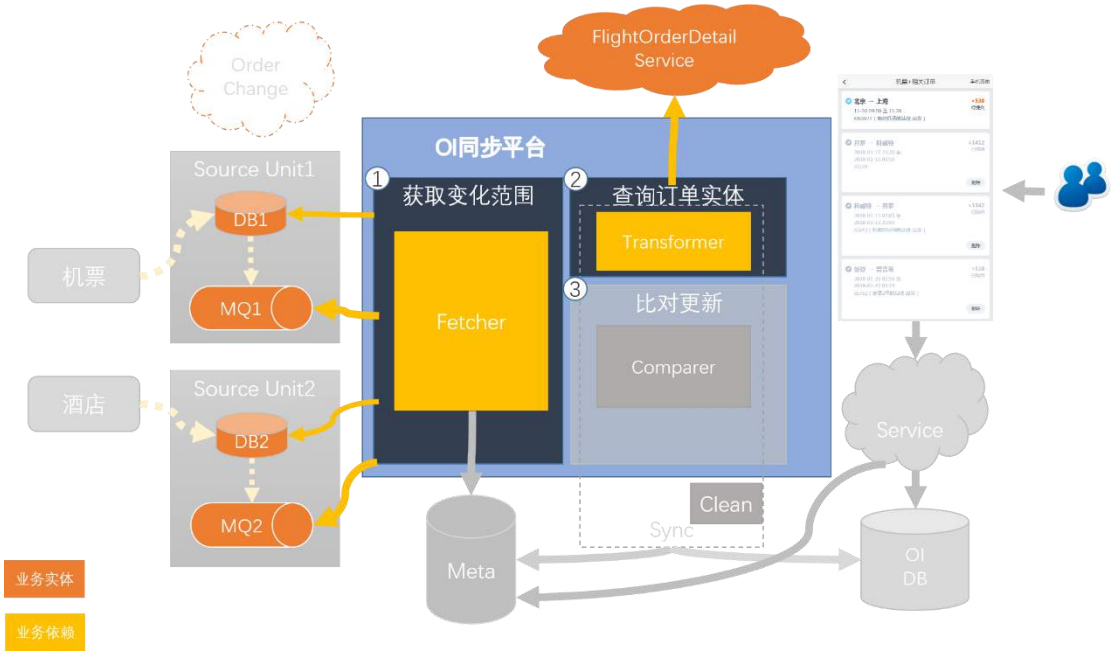


OI 平台化

将订单变更检测和订单详情数据源，以及订单的部分特殊标准信息（如订单状态等）全部作为 meta 通过统一配置管理平台来进行管理，并且自研了 Job 调度模块以支持根据配置的数据源 MetaData，动态的生成同步 Job 来执行数据同步任务，并进行数据校验。

同样，SOA 数据查询服务基于相同的 MetaData， 来提供数据查询服务。

基于新的设计，订单同步流程也采用了新的模式。



平台化后的订单同步流程

1）首先，将需要同步的数据源 Meta 信息通过配置管理录入。
2）OI 同步平台根据数据源 Meta 信息生成同步 Task 和对应的同步 Job （若同一数据源有配置多种感知渠道，则生成多组同步 Task 及对应 Job）。
3）通过配置的订单感知渠道（db，消息队列，SOA 服务），感知订单变化。
4）通过统一的标准 Facade Order Detail Service 获取，标准 MetaData 描述的订单信息。
5）将订单信息经过校验和变更比对后，更新（加入）到 OI 数据库。
6）通过标准 MetaData 的订单实体信息查询接口，将订单数据下发给服务调用方。

四、OI 2.0 平台化，我们做了哪些具体的工作?

4.1 重新设计了统一接入信息模板，通过统一配置管理服务管理

基于平台化后的系统，重新设计了一份标准订单接入的信息模板，用来取代之前跟每个业务线针对所有数据源逐一讨论商定的接入信息文档。（不过暂时不支持接入方自主录入，需线下提供，审核完毕后，再录入 Meta 信息）。通过这个改变，将过去接入过程中贯穿始末的业务讨论时长从数周缩短到了一两次会议（电话或者面谈）就能沟通完毕的程度。

4.2 自研 Job 调度系统（JobHost）

我们放弃了使用公司已有的成熟通用 Job 调度系统，自研了一套 OI 定制的 Job 调度系统，通过定制开发，支持了一些原本很难实现的功能。

1）结合统一信息模板，极大的简化了订单同步 Job 的复杂度，OI 的 JobHost 默认会基于配置动态生成常规（Normal）同步 Task，同时也支持手工指定要同步的数据源来生成任意多个自定义的同步 Task（如针对部分数据进行数据清洗的补偿 Job，OI 1.0 由于设计原因仅支持一个），并支持通过对数据源的开启标志的控制，实现对相关 Job 的实时联动控制（公

司的调度系统需手工操作)。

2）运行状态监控粒度更细，由于定制开发，所以针对各种 Job 的不同运行步骤，设置了更丰富的监控点，更好的了解 Job 的运行状态。

3）除了守护 Task 会保持心跳，检查 JobHost 中运行的 Job 健康状态，尝试进行自动恢复外，我们还设计了人工干预的机制，可以针对运行在每一个实例中的任意 Job 进行更细微的调整。

4.3 自研的数据查询引擎（Sql+内存过滤，根据策略编译查询条件）

为增加数据平台对异构数据存储结构的支持以及提供特定场景下的 db 数据读取优化，我们自研了一套简易的数据查询引擎。

上层服务只需要将数据查询请求，转化成平台统一的 OrderQuery，再注入一套 OrderQuery 编译策略，以支持将 OrderQuery 中的查询请求和过滤条件，转化为 DB 执行的 SQL 语句以及内存中过滤的 OrderFilter（同一过滤条件，在不同场景下通过 SQL 直接过滤，或者内存中过滤，效率可能会不同，查询引擎会根据编译策略，选择将该过滤条件 build 进 sql 语句中，或者是生成对应的 OrderFilter)。

五、改造完成后的成果

前面零零散散讲了很多细节，大家对改造效果可能没有很直观的认识，我们再来看一张图。

这是我们在确认改造方案后，第一阶段的整体目标。

1）通过平台化改造以后，OI 不用再针对每一个业务线（订单业务类型）做定制开发，通过平台提供统一支持即可，整体复杂度降到了 O(1)，即使不熟悉 OI 的订单接入方，或者组内同学，又或者测试的同学，凭着简单的文档，也可以直接上手了，不用业务线再提供 sql 和 db 字段说明，也不用再提供 db 字段到实际订单数据的业务逻辑。

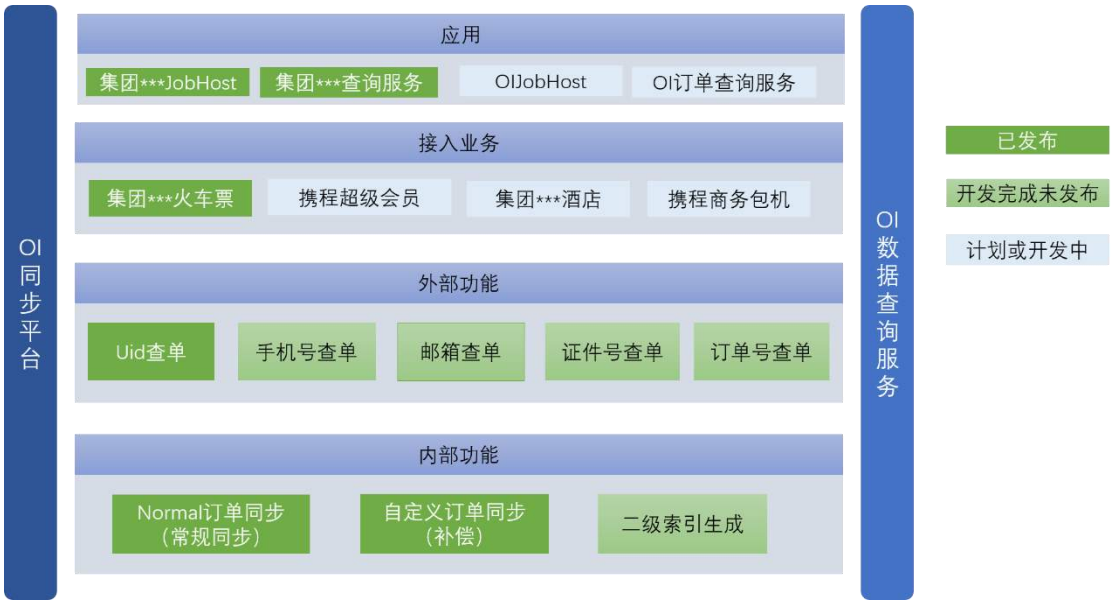
2）改造完成后，新接入一种订单（数据源），或者下线一个数据源只需要通过配置管理配置好数据源的 MetaData 即可，无需修改代码，也无需重启服务，实现了热插拔和热更新。

3）通过标准详情接口，将溢出到 OI 的业务线订单业务彻底归还业务线，业务线业务发生变动的时候，直接同步修改详情接口实现即可，无需再拉上 OI 一起排期改造，效率大大增加。

4）结合 1-3 带来的效率提升，在 OI2.0 数据平台上进行业务变动，在业务线准备好的情况下，基本上都能实现 T+0 上线。

5）基于全新的标准化改造，我们可以针对不同运行环节增加统一的监控点，实现更灵活的监控扩展。

最后，来看一张图：



这是之前，第一个里程碑完成时的 OI2.0 交付情况。目前 OI 数据平台，已经能够多订单系统复用，支持携程订单体系和集团的某关联企业订单体系。

基于这还算坚实的第一步，我们会继续一步一个脚印，继续去完成我们所描绘的蓝图。

计算密集型服务的负载均衡策略

【作者简介】罗茂林，携程国际机票后台研发总监，主要负责国际机票引擎的研发工作。致力于系统性能优化和研发效率提升。

一般情况下，在计算密集型服务中，即使处理单个请求也需要使用到服务器的所有 CPU。如果单台服务器连续接收到两个请求，要么两个请求互相争抢 CPU，要么后来的请求排在前面的后面等待处理。最终，会导致平均处理时间变长。常规的负载均衡策略（如轮询、随机等）下，负载均衡器不关心服务器的负载情况，这就很容易造成服务器同时收到多个请求，从而使服务器的服务质量下降。

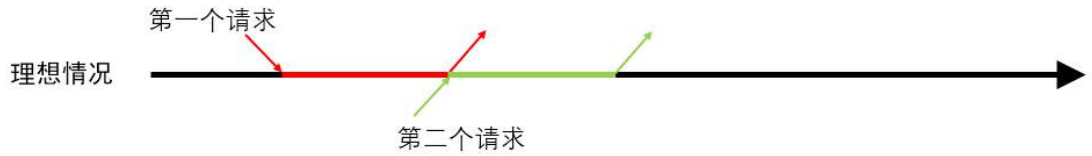
一、背景

有一天，携程国际机票查询引擎经过一次改造后，虽然平均响应时间得到了提升，但是响应时间也有非常大的波动。从监控图上看，非常明显的尖刺持续存在。如下图：

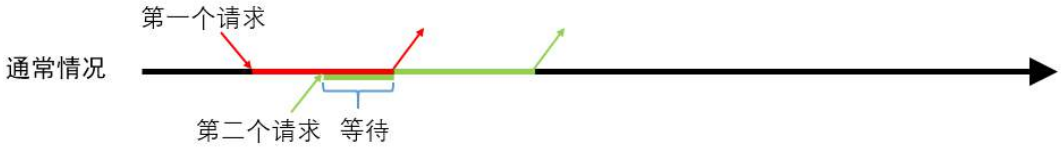
经过分析，我们发现这次改造深度优化了服务的并行计算能力，使得引擎成为了一个完全的计算密集型服务，它的最大并发处理能力为 1。然而，我们却没有相应的修改负载均衡策略，而是继续使用的轮询策略。

对于计算密集型服务，如果使用轮询策略，有如下三种情况：

理想情况是连续两个请求之间无间隔、无重叠，既下一个请求刚好在上一个请求处理完成的时刻到达。这种情况下，后来的请求没有等待时间，服务器也没有空闲时间，得到了充分的利用。



通常情况下，由于请求的到达普遍服从泊松分布，如果使用轮询、随机等负载均衡策略，单机的请求也服从泊松分布，即连续两个请求间总会存在间隔或者重叠，导致服务器资源空闲或者请求响应时间上升。



在极端情况下，如果某个请求的处理时间特别长，后续的一大串请求将产生积压，最终导致这些请求的响应时间也变得特别长，甚至超时。

我们发现，引擎的响应时间尖刺是由极端情况的 case 造成的。引擎有一类请求 A，它 qps 不高，但是却需要 CPU 满负荷运转长达几秒甚至 10 秒才能算出结果。另有一类请求 B，它 qps 非常高，只需要 CPU 满负荷运转几十毫秒就能算出结果。

当一台服务器正在处理一个 A 类请求时，在接下来的几秒内，它将继续收到几十个 B 类请求，而且所有的 B 类请求都要排队，直到 A 类请求完成。这就导致大批 B 类请求的响应时间由应该的几十毫秒升高到几秒，从而造成了严重的尖刺。

二、pooling

为了解决这个问题，我们使用了一种新的负载均衡策略，在这种策略下，服务器不再被动的接收请求，而是主动的去获取请求，这种方式非常容易做到服务器同一时刻只处理一个请求。在我们内部，这种方式被称为 pooling（它和线程池类似，可以叫做服务器池）。

在 pooling 模式中，有三个主要角色：submitor、queue、worker。

- submitor

submitor 一方面用于接收请求方的调用，它收到请求后，不直接处理请求，而是把这个请求提交给 queue。

另一方面，submitor 接收 worker 的回调，submitor 收到 worker 的结果后，直接把它转发给请求方。

- queue

pooling 的关键是引入了一个 queue，queue 是一个全局唯一队列，用于暂时缓冲请求。

我们使用了 redis 的 list 结构来实现 queue。入队操作为 lpush，出队操作为 brpop。brpop

是阻塞式的操作，当队列为空时，brpop 会阻塞直到队列非空。队列非空时，如果有该队列有多个 brpop 操作阻塞，只有其中一个会被唤醒并且返回数据。

- worker

worker 是实际的请求处理者。在旧的模式下，worker 是被动接收请求。在 pooling 模式下，worker 要主动去 queue 获取请求。worker 启动时，要创建一个线程，这个线程启动后，便进入一个无限循环，循环的主要内容为：

1）从 queue 获取一个请求，当 queue 没有请求时，worker 被阻塞。
2）worker 处理这个请求。
3）把结果返回给 submitor。

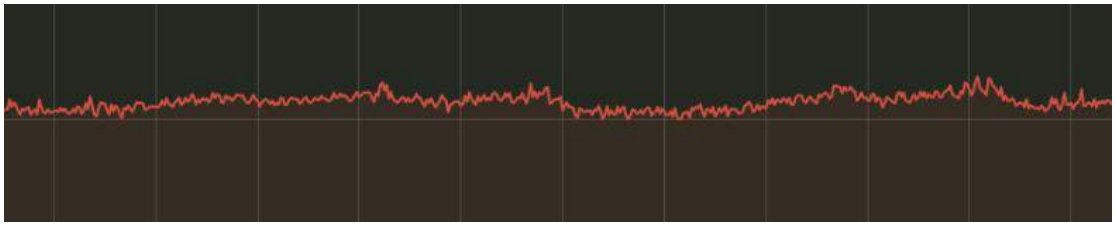
如此往复。可以看到，worker 要么正在处理一个请求，要么正在等待一个请求。

三、效果

国际机票查询引擎的负载均衡策略由轮询改为 pooling 后，效果非常好。系统的平均响应时间降低了大约 20%，并且完全消除了响应时间尖刺。

轮询方式：

pooling 方式：



质量保障篇

节省 55%测试时间，携程酒店比对平台介绍

【作者简介】黄文杰，携程酒店研发部高级测试经理，主要从事测试框架和平台的研发，现在负责自动化与工具平台，热衷于研究技术提升测试工作效率。

一、前言

当初我们为什么想到要开发比对平台，主要遇到以下几个问题：

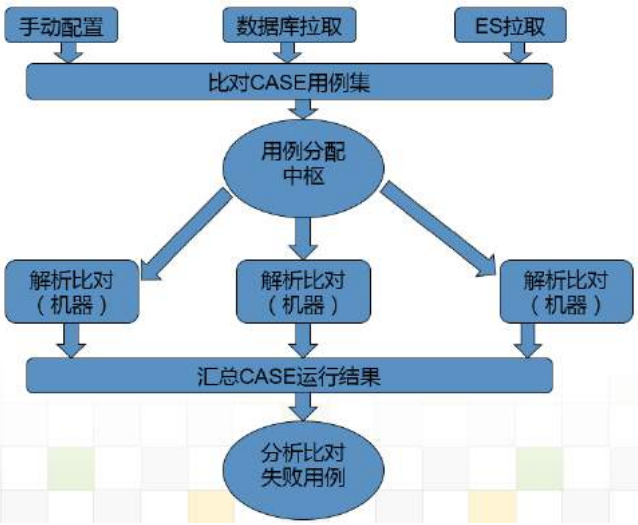
1）很多.net 应用转 java 返回的报文相差不大；
2）采用常规的方法测试测试量太大，测试时间太长；
3）可以补充多的测试场景；
4）自动化测试的补充；
5）实现比对监控；

为此我们根据以上测试问题设计开发了接口比对平台。在使用过程中极大的提高了测试效率，在此基础上根据业务人员需求相继开发了数据库比对模块，埋点比对模块，缓存比对模块。

本文将从各模块如何实现比对以及各个模块相互关系的方面，分享携程酒店比对平台，希望能给遇到同样问题的小伙伴带来一些启发和借鉴。

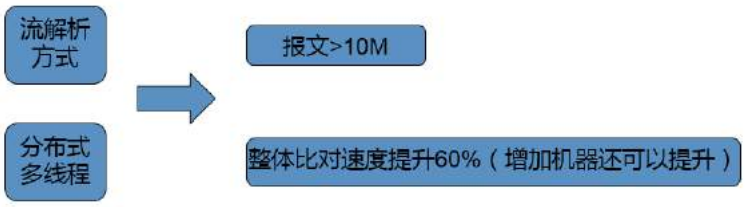
二、整体介绍

2.1 比对流程图



2.2 相关技术

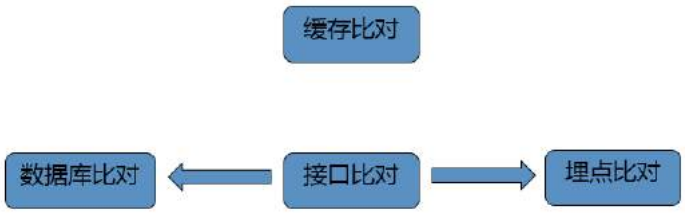
比对中我们主要采用流的报文解析方式，以及分布式多线程运行。



通过该方式可以解决以前比对中以下一些瓶颈：

1）不能忽略节点比对；
2）不能忽略节点排序比对；
3）不能选择节点比对；
4）不能比对大于 10M 的报文；
5）比对速度比较慢；

2.3 模块关系图



比对平台由四个模块组成：接口比对、数据库比对、埋点比对，缓存比对。

根据用户配置，接口比对的同时可以生成数据库比对用例，当接口用例集比对完成接下来就运行数据库比对用例集，极大提高了测试效率，当然数据库比对也可以独立使用。

根据用户配置，接口比对的同时也可以生成埋点比对用例，当接口用例集比对完成接下来就运行埋点比对用例集，埋点比对也可以独立使用。

缓存比对模块目前是单独使用的，主要比对数据库数据和缓存数据，增，删，改，查等操作。

三、模块介绍

3.1 接口比对

比对原理：同一个请求报文，访问二个部署在不同机器的不同版本服务，比较二个服务的返回报文。

比对用例数据来源：

- 手动配置
- ES 保留
- ES 实时拉取
- 数据库拉取

用例集可以配置以下规则对报文处理后进行比较：

- 忽略大小写：节点值比对时，忽略配置节点值的大小写
- 忽略节点排序：对报文中 list 需要忽略排序的，可以配置节点忽略排序，list 比较的时候我们会进行排序处理然后进行报文比对
- 忽略节点：忽略配置的节点（以及所有子节点）的比较
- 忽略属性：忽略配置的节点中属性的比较
- 数值比较：将配置节点值转化为数值进行比较
- 属性数值比较：将配置的属性值转化为数值进行比较
- 白名单：只比较配置了白名单的节点（以及所有子节点）
- 节点替换：将返回报文的节名称进行替换

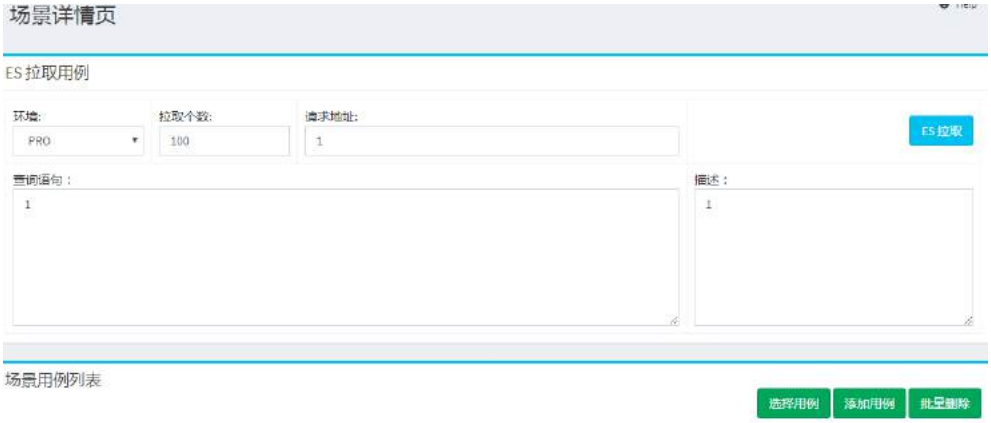
ES 用例集详情页配置点

- 获取用例环境
- 拉取用例数
- ES 请求地址
- ES 查询语句

DB 用例集详情页配置点

- DB 名称
- 拉取用例个数
- 数据库服务地址
- SQL 查询语句

用例详情页面也可以选择手动用例或者新加手动用例进入用例集。



比对运行方式

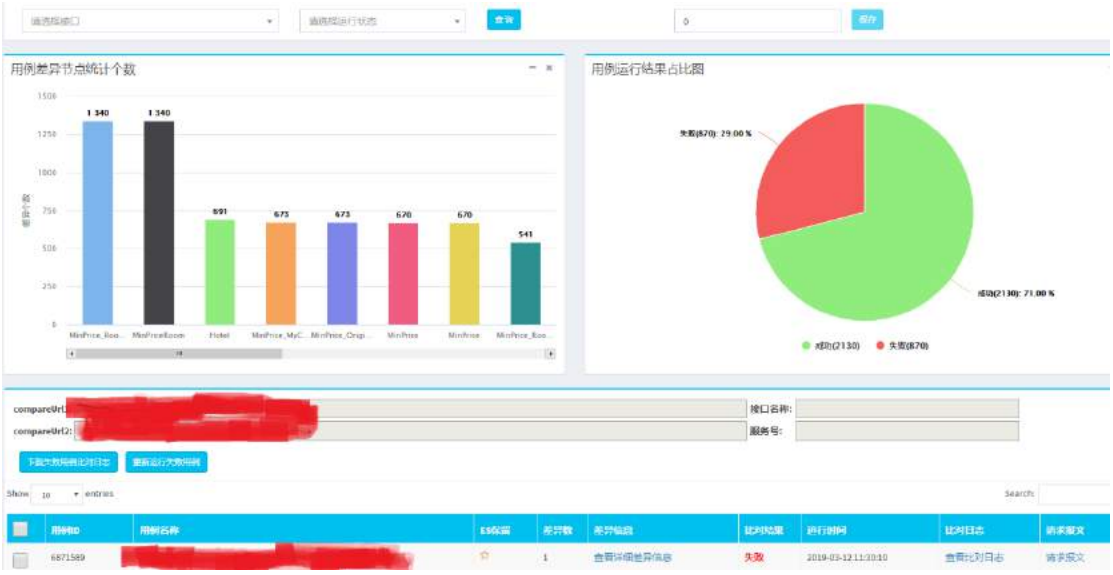
- CI 自动运行（和自动化体系打通，服务发布后自动运行）
- CI 手动运行
- 手动运行场景用例集
- 定时器运行（可实现比对监控）
- 单用例手动运行

执行方式

- 分布式
- 多线程

运行结果查看页

- 根据接口，运行状态筛选运行用例
- 根据用例差异节点统计个数筛选运行用例
- 差异信息展示节点值不同的列表
- 比对日志展示返回比对报文
- 请求报文展示当时请求报文信息
- 可以重新运行失败用例
- 可以下载失败用例报文日志



3.2 数据库比对

比对原理：二个相同类型订单在数据库中各个需要比较的字段进行比较。

比对用例数据来源:

- 手动配置
- 接口比对生成

用例集配置点

- 比较的库表
- 忽略比较的库表字段（可不填）
- 需要比较的库表字段（可不填）

比对运行方式

- 手动执行
- 接口比对完成触发数据库比较执行

执行方式

- 分布式
- 多线程

运行结果查看页

- 根据表比较不同个数展示筛选运行用例
- 差异信息展示用例表字段具体比较不同详细信息

3.3 埋点比对

比对原理：接口比对生成的埋点用例，根据埋点 key 从 ES 中获取埋点信息进行报文比对。

比对用例数据来源：

- 接口比对生成

用例集配置点

- 应用 ID
- 公共规则（可以不填）

比对运行方式

- 接口比对完成触发埋点比较执行

执行方式

- 分布式
- 多线程

运行结果查看页

- 根据用例差异节点统计个数筛选运行用例
- 差异信息展示节点值不同的列表
- 比对日志展示返回比对报文
- 可以重新运行失败用例

3.4 缓存比对

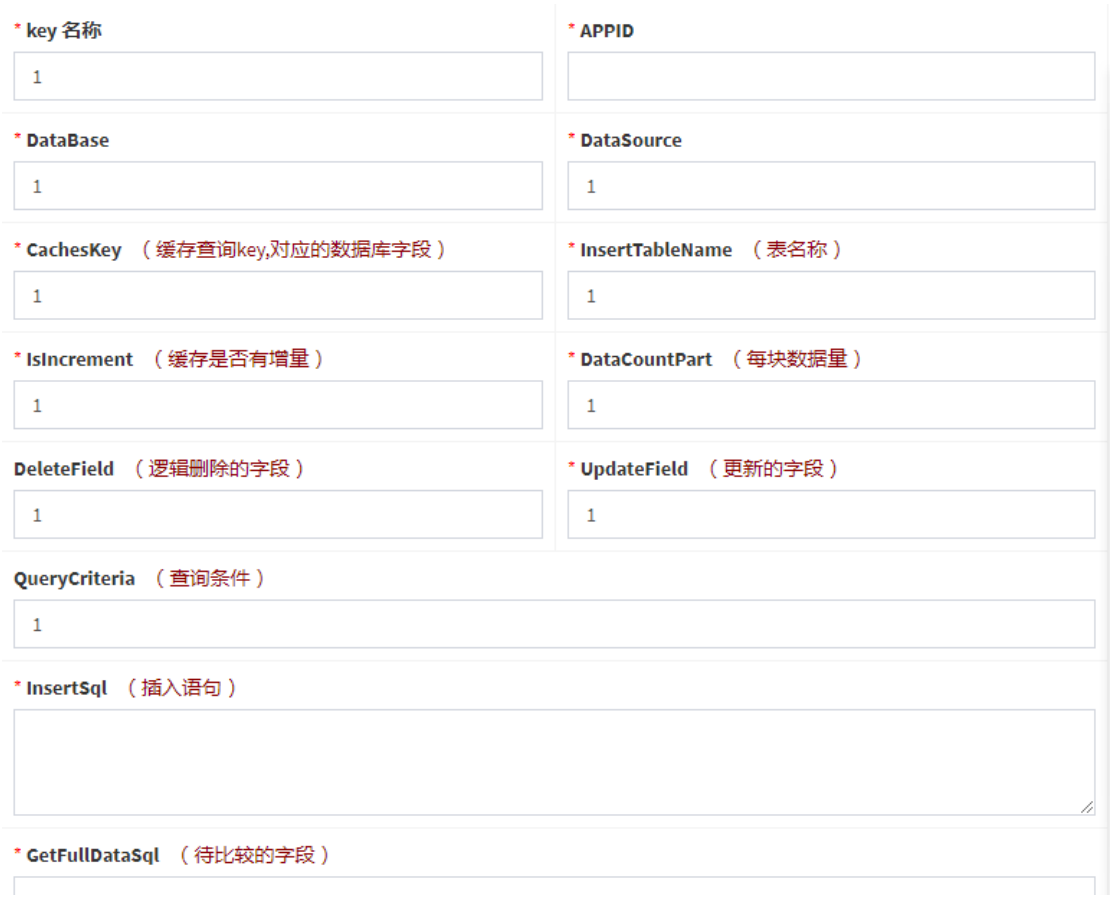
比对原理：数据库查询的数据和缓存中数据进行比较。

比对用例数据来源：

- 手动配置

用例配置点：

- 查询 SQL
- 缓存 Key
- 比较的字段
- 等等（和具体业务相关就不具体描述）

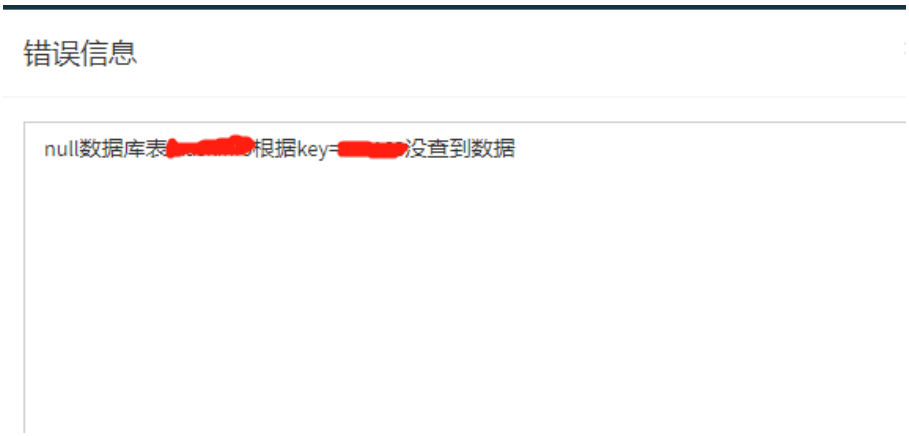


比对运行方式

- 定时运行
- 手动运行

运行结果查看页

- 具体错误信息展示弹出层中



四、总结

目前比对平台已稳定运行一年多:

- 用例基本不需要维护，每次拉最新（除手动用例）
- 用例基本都是 ES，DB 实时拉取不需要编写（除手动用例）
- 场景配置简单，基本不需要学习成本
- 失败用例自动归类降低分析成本
- 用户接入简单方便
- 比对测试丰富了测试场景，构成自动化测试体系中重要一环
- 1000 个用例平均 4 分钟内运行完成
- 目前 Case 量 50 万+
- 到目前发现 BUG 数 1000+
- 节省测试时间 55%+

为了更好的满足用户需求，仍在持续迭代中。

携程酒店 MOCK 全链路实践

【作者简介】刘晓攀, 携程酒店性能测试负责人, 专注性能测试分析和辅助测试工具的开发。

一、前言

Mock 在整个软件开发测试周期中已经非常普遍，我们也会经常有意无意地使用它。譬如开发了一段代码，这段代码强依赖了其他服务，在对方服务完成之前，肯定是期望代码能够同步开发。那么在开发的过程中一定会根据约定固定对方服务的返回，这种在代码中的模拟行为，是一种 mock。

另外当前很多应用为了提高性能，普遍采用 cache 的方式。有些 cache 数据是需要很多 database 的数据经过一定的逻辑运算得到的，而在测试过程中，为了快速验证测试场景，直接取修改 cache 数据的方式，也是一种 mock。

在当前五花八门的 mock 方式中，我们希望有一种方便快捷的 mock 方式，便于开发测试使用，特别是对于多依赖、多变化的场景，更需要 mock 的协助。否则将需要打通一系列的外部依赖来满足一个场景的需要。

于是，我们推出了 mock 全链路。mock 全链路既是一种 mock 方式，又是 mock 在不同阶段的扩展使用。当然 mock 全链路还面临着一些些问题，我们将在后面的章节中详细展开。

二、技术背景

技术的进步更多源于底层技术, 比如正是由于蒸汽机的发明, 才有了现在各式各样的机动车, 挖土机，收割机等等，工业上各种技术也有了不同形式的发展。同理，酒店 mock 也是伴随着基础架构、日志系统的更新换代而不断发展。

mock 全链路就是在框架能够完整跟踪应用调用链，指定 header 在链路上无限透传，ES 日志系统大范围应用的背景下产生的 。当然不管 mock 怎么变化，最基础的原理还是建立在常规 mock 的基础上的（如图 1）。

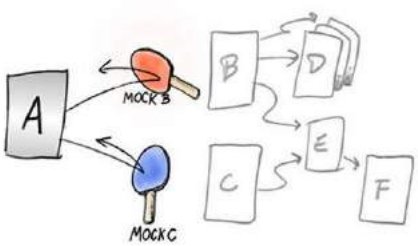


图 1 常规 mock 示意图

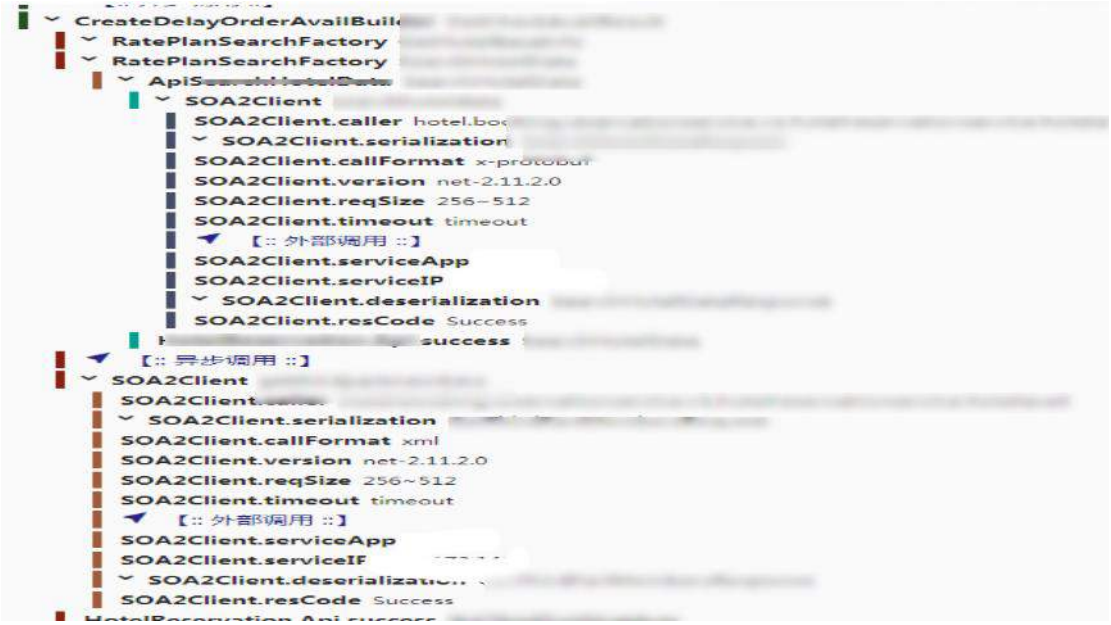
首先在多依赖的应用中，特别是强依赖服务的业务逻辑比较复杂，依赖服务所在的组还有一堆任务的情况下，通过构造打通依赖服务的行为会变得非常耗时。严重的时候，会拖累整个

项目的进度。

ES 在公司的大规模应用，为我们解决此类问题提供了契机。ES 的埋点数据为 mock 提供了很好的种子。框架的链路完整埋点为全链路提供最基础的管道,没有这个最基础的管道，所有的链路都将是模糊不清，无法识别的。同时 header 在链路中的透传为整个 mock 实现铺平了道路，使得 mock 能够快速根据 header 中的规则来识别不同的链路调用。

测试的不同场景在这里也有了很好的区分。而这些数据又都可以在 ES 中通过埋点来解决。mock 可以通过 ES API 把这些数据拉来使用，从而基础的 API 调用链路已经成形并且明晰。这就为后续的实现打下基础。

基础调用链路图如下：



Es 链路埋点如图 2 所示：

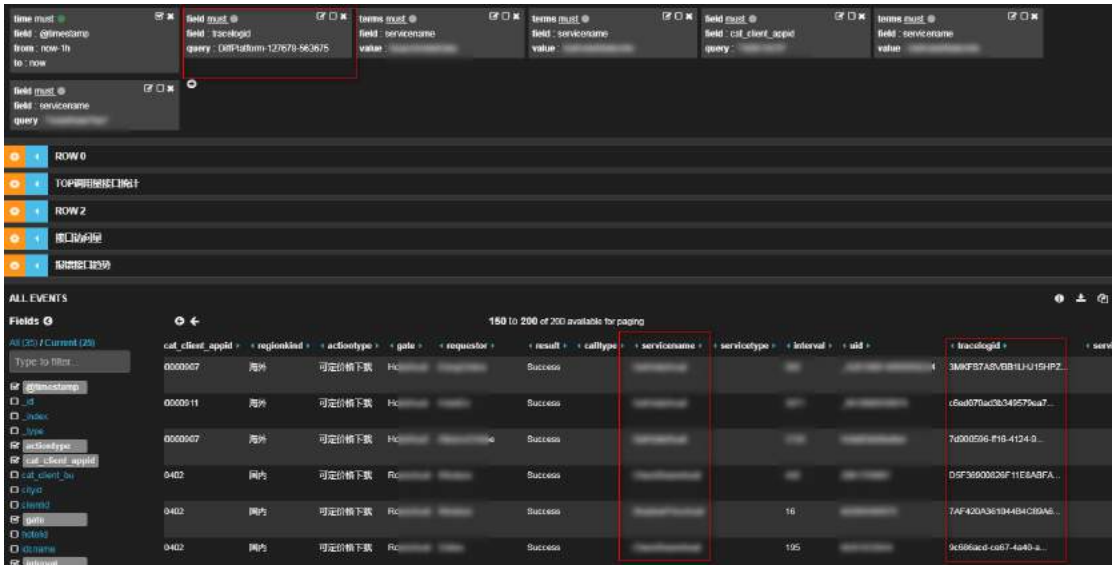


图 2 ES 埋点链路示意图

当所有的链路设施都准备好之后，影响整个使用过程的就是怎么样把 mock 快速无缝的切入真实调用链路。

我们知道，无论是 slb 还是 ip 直连，最基本的原理还是所有的服务都需要注册，注册的最终目的就是所有的服务中心化。当调用依赖服务的时候，在中心里获取对方的服务，提供给调用方使用。

这样的情况下，应用切入 mock 最直接的想法就是把中心中依赖方地址修改成 mock 地址，当然这就需要框架提供支持。这是一种侵入式的 mock 应用方法。

另外一种方式是通过 PROXY，类似于网络代理，所有经过代理的网络包在代理中进行收编和整理，该走 mock 的走 mock，不走 mock 的放回原来的网络中。这种方式是当前相对比较理想的方式。把对应用的侵入，变成系统层面配置的更改，相对适应性更强一些。

综上两点解决后，整个 mock 链路的基本技术方案有了一个比较可行的落点。mock 全链路的整体架构如图 3。

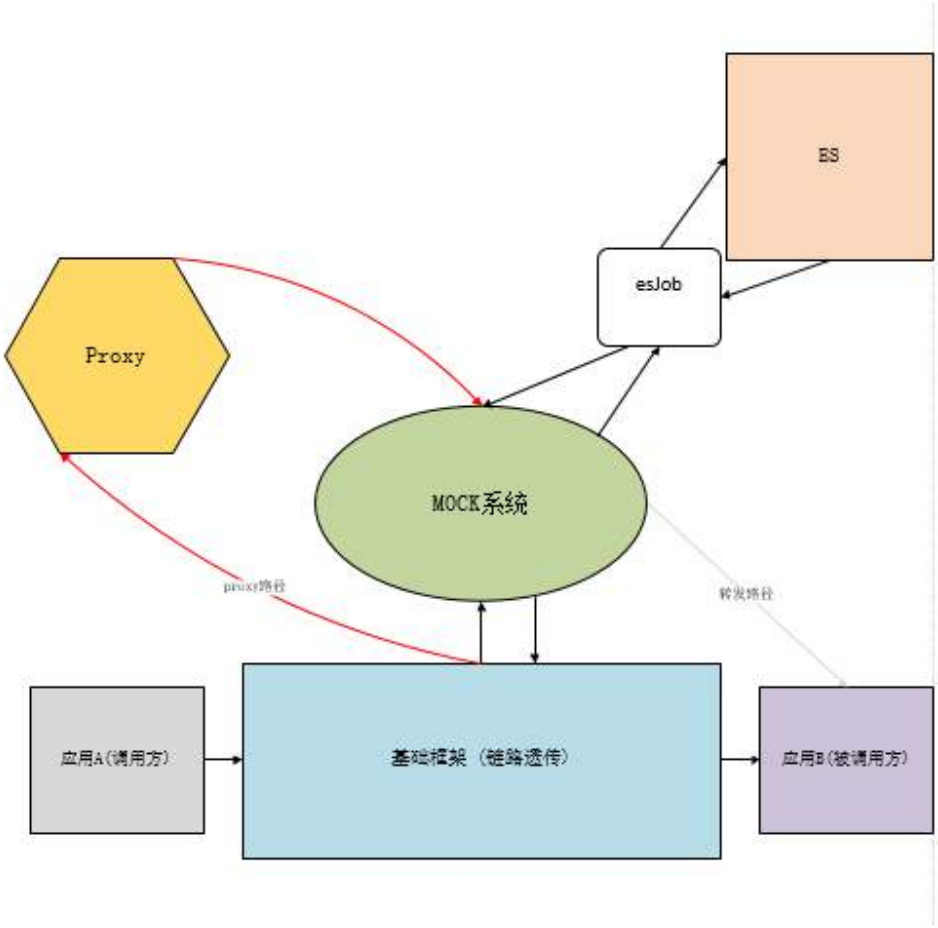


图 3 MOCK 链路架构图

三、技术实现

Mock 的技术实现要注意以下三点:

3.1 Mock 的响应速度

响应速度要满足服务调用基本要求，不能 timeout。正常情况下，mock 不需要业务逻辑处理，只需要做规则的简单匹配，响应速度要快才对，这是建立在规则相对少的情况下。

如果在千万亿级的规则进来之后，规则的获取，场景的匹配速度会拖慢整个 mock 的响应，现在的 mock 系统同样要依赖缓存技术来提高规则场景的匹配速度。

另外是 pb 格式请求，我们要进行多次的序列化和反序列化，来保证配置的明文 json 或者 xml 能够正确的在请求中传递，而序列化和反序列化本身就是一个消耗 cpu 的动作，这会给响应造成极大的延迟。

当前情况下，我们尽量提前进行序列化和反序列化操作，操作结果存入缓存，这样能够在此等格式请求传递和返回传递的过程中，保持快速的字节流传递而不需要额外的序列化反序列化操作。

最后在转发的情况下，相对正常服务调用多了一次网络请求，时间消耗相对较长。在这方面我们尽量采用减少连接等待，保持连接状态等方式来减少多次 socket 连接造成的时间损耗。

3.2 契约依赖问题

对于依赖契约的请求格式，如 pb 等，需要契约的更新速度必须在 ES 埋点落库之前。

契约延迟更新是我们面临的很大问题，契约的不及时更新，会造成数据节点的丢失或者直接调用报错。发布订阅和编译包下载替换是我们目前解决这个问题的主要方式。

当前这需要维护一个所有应用契约文件的列表，以便我们在编译包中进行查找替换。而替换动作成功发生的前提是当前契约文件没有被进程占用。

3.3 系统操作的易用性

操作简单是用户使用的基础，也是系统的目标，当然操作习惯也会影响系统在不同人群中的使用。但最终的目标，是希望用户用尽量少的鼠标或者键盘动作，来完成整个 mock 的操作。链路使用操作相对简单是系统能够被使用的重要环节。

譬如 ES 数据拉取，我们采用了只让用户输入 ES 共享链接来代替繁琐复杂的 ES 过滤器配置。

最原始的设计 UI 如下图所示：需要有十几个配置项需要用户取配置：

改良后的交互如图 4 所示，只需要输入 ES 共享连接即可，其余的事情都由系统来协助用户完成，从而减少用户的使用费力度。

图 4 ES 数据拉取

四、面临的一些问题

在 mock 全链路的使用过程中，遇到一些新的问题，比如在真实调用链中重复调用的问题。由于重复调用的返回结果不一致，导致我们在 mock 全链路的时候无法知晓哪个是第一次要返回，哪个是第二次要返回，从而影响全链路的真实性。

缓存问题。之前提到过很多应用都会在本地或者其他缓存服务器做数据缓存，很多处理逻辑是缓存数据存在的情况下，拉取缓存数据，否则走依赖接口调用，在 mock 切入应用之后，发现很多缓存未过期导致 mock 无法被调用，从而影响全链路的正确性。

转发 302 问题。有些服务需要在服务 API 之间做 302 转发调用，而 mock 正常是要返回 http code 200 的状态的。

针对不同业务的问题，我们只能针对性的进行个性化或者通用兼容性修补，在通用系统的基础上，进行个性业务适应。

总之，mock 全链路系统需要在实际业务中进行不断的锻造、更新，在使用中不断调整，从而保证 mock 全链路能够适应不同业务的需要。

五、后记

Mock 的设计特别要考虑压测的需求，压测对 mock 系统的要求会更高，所以我们在设计之初，要把这种需求优先考虑，对系统扩展和快速资源扩容留下足够的空间。

另外匹配规则的设计要尽量简单，譬如正则和 xpath, jpath 此类的功能，从当前的统计数据来看，使用率不高；而简单的规则，譬如 header 匹配，包含等则被用户大面积使用。

行为驱动开发在携程机票前端研发流程中的实践

【作者简介】任跃华，携程机票前台软件工程师，参与了国际机票 RN Clean Architecture 落地和 MEC 中文测试框架研发工作。

前言

过去，在携程机票前台团队保障研发质量的体系中，采用先开发后测试的模式，测试验收环节以手工测试为主。

机票预订研发流程中 BDD（行为驱动开发）模式的引入，统一了技术人员和非技术人员对软件行为描述的语言，均衡了自动化测试与手工测试之间的关系；入门级中文编程易读易用，且支持细颗粒度用例及海量用例复用。

一、困境

传统的敏捷软件开发，产品经理根据用户诉求和商业目的撰写 PRD 文档，测试工程师基于 PRD 文档考虑边界值和场景排列组合产出测试用例文档，软件工程师按照自己对需求的理解实现代码，最后的验收环节由手工测试完成。

在这个过程中，容易出现这些问题：

- 各方低质量的沟通

产品经理和技术人员分别站在不同的角度，使用不同的专业术语描述软件的行为，这使得沟通往往反复进行。

- 难以维护与线上逻辑一致的文档

PRD、用例文档和软件代码都是对软件行为的描述，要保持这三份文档逻辑的同步是投入高收益少的事情，实际上他们通常是不同步的。如果遇到项目重构或团队人员变动，需要花费较多的时间才能整理与线上软件行为一致的文档。

- 先开发后测试放大风险

实际项目经验表明：问题暴露的时间越临近发布时间，修复问题的成本越大。

- 手工测试限制迭代速度

每次发布前，投入手工测试做回归，周期长，成本高，限制了发布的次数。

- UI 自动化成本高覆盖低

自动化测试需要较高的编程能力，对于功能测试人员门槛较高。

我们尝试通过 BDD 缓解这些问题。

二、什么是 BDD

什么是 BDD ？参考维基百科的描述：

行为驱动开发（英语：Behavior-driven development，缩写 BDD）是一种敏捷软件开发的技术，它鼓励软件项目中的开发者、QA 和非技术人员或商业参与者之间的协作。BDD 最初是由 Dan North 在 2003 年命名，它包括验收测试和客户测试驱动等的极限编程的实践，作为对测试驱动开发的回应。在过去数年里，它得到了很大的发展。

BDD 的重点是通过与利益相关者的讨论取得对预期的软件行为的清醒认识。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。行为驱动开发人员使用混合了领域中统一的语言的母语语言来描述他们的代码的目的。这让开发者得以把精力集中在代码应该怎么写，而不是技术细节上，而且也最大程度的减少了将代码编写者的技术语言与商业客户、用户、利益相关者、项目管理者等的领域语言之间来回翻译的代价。

使用 BDD 的敏捷软件开发包括以下关键步骤：

- 需求各利益方（产品，测试，开发）对需求进行充分讨论
- 讨论基于软件的行为展开
- 讨论的产出为自然语言书写的非程序员可读的测试用例文档
- 产出的测试用例能在自动化测试平台上执行
- 程序员专注于编写代码通过测试用例

BDD 是一种软件过程的思想或者方法，而不是一个技术框架或者系统。

为了建立“自然语言测试用例文档”和“自动化测试代码”间的关联关系，需要用到支持 BDD 工具，我们使用了 Cucumber。

为了实现 BDD 中“测试用例能在自动化测试平台上执行”，需要用到 UI 自动化测试框架，我们使用了 Macaca。

三、BDD 改造过程

3.1 Cucumber

Cucumber 是一种支持行为驱动开发的工具。Cucumber 提供了一套名为 Gherkin 的语法规则，一个功能的描述由多个场景组成，一个场景由多个语句组成。

以 "假如(GIVEN)" 开头的语句描述的是场景的前提条件、初始状态， 以 "当(WHEN)" 开

头的语句表示采取某个动作或者是发生某个事件， 以 "那么(THEN)" 开头用来描述一种期望的结果。每条自然语句将和一个代码编写的自动化测试方法对应，这让整个文档变得可执行。

如下 feature 文档描述了在机票单程列表页的直飞优先排序功能：

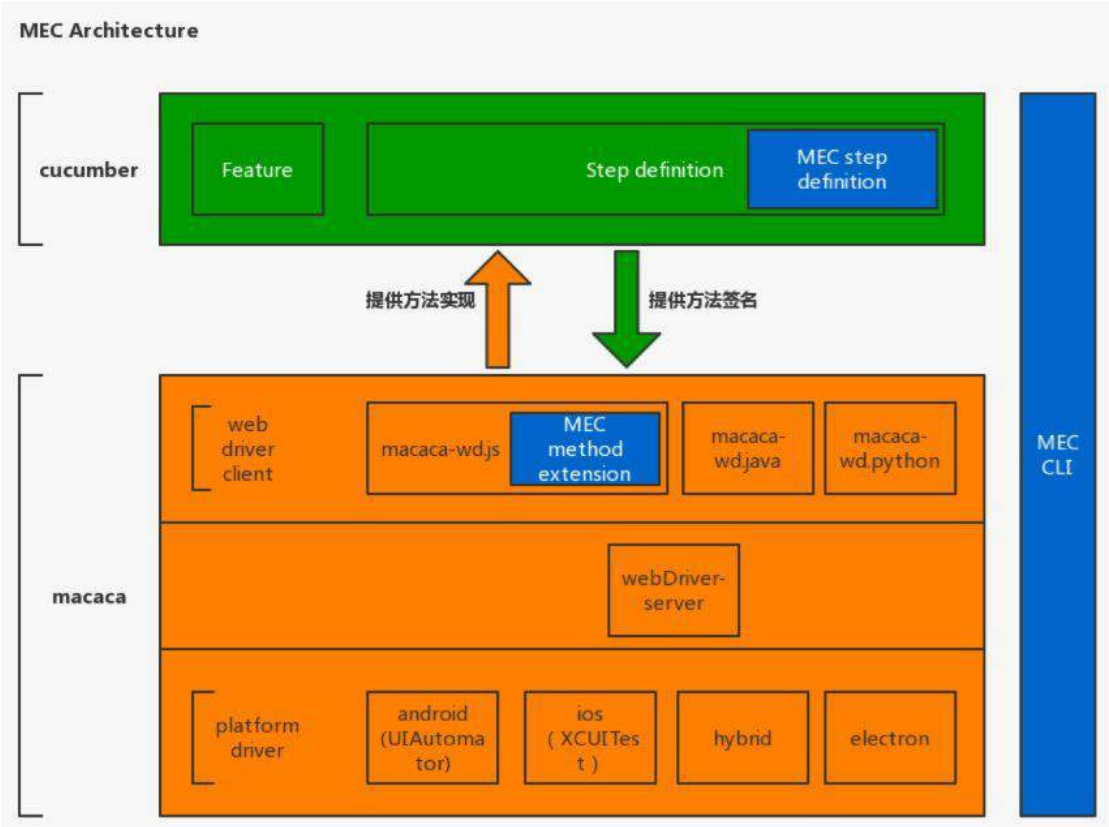
```
# language: zh-CN
功能: 排序-单程列表页

  场景:
    假如 跳转页面到[机票单程列表页]
    当   点击[直飞优先按钮]
    那么 第[1]程列表页按照直飞航班在前，中转航班在后排序
```

3.2 Macaca

Macaca 是阿里开源的前端 UI 自动化测试框架。业内优秀的自动化测试框架不少，最终我们选择 Macaca 有以下几个原因：

- 跨平台 API 统一性 — Macaca 支持 PC、Android、iOS 和 Hybrid，并从 API 的设计上抹平了各平台的差异，这样的设计对测试跨平台的 React Native 应用有利；
- 文档和周边工具丰富 — Macaca 官方网站提供了丰富的中英文文档，有利于框架的快速接入使用，同时提供了 app-inspector 等常用工具，方便了控件的查找定位；
- 多语言支持 — Macaca 支持使用 Java、JS 和 Python 编写测试脚本，其中 Java 和 JS 是团队中常用的开发语言，降低了学习成本；
- 开源 — 能看到源代码，方便问题定位和功能扩展。

3.3 MEC



我们在 Cucumber 和 Macaca 的基础上，整合出一系列通用的工具和完善的文档，取名为 MEC (macaca eating cucumber)。为了让 BDD 变得轻松和高效， MEC 做了这些事情：

1）扩展 Macaca Api

支持在携程 app 中打开 Schema，绑定服务 Mock，登陆账号等功能。

2）提供开箱即用的公共自然语句

30+ 条，涉及交互、页面元素断言、服务 Mock、页面跳转等场景的可执行自然语句。

3）提供 CLI 改善使用体验

提供 10 个命令，涵盖项目初始化、打补丁、运行、下载 app、编译、生成报告等场景。

4）支持基于 UI ViewModel 的校验提升自动化测试覆盖率

基于 UI 自动化框架，能方便的对可视区域内的单个页面元素进行测试，但对以下场景则不能很好覆盖：

- 基于多个页面元素的关系的判断 - 比如判断航班列表按照低价优先展示，航班在列表中的顺序越靠后，价格越高；
- 长列表 - 需要把要校验的元素滑动到可视区域，才能获取；

- 更快的执行速度 - 运行在移动设备上的 UI 自动化稳定性和执行效率不理想;

我们的解决方案是将页面上展示的信息用数据的方式发送给 MEC Server， 如 React 中把 state 发送出来，测试用例的断言部分，直接校验界面数据，而不再通过 UI 自动化框架实现。

5）实现 Cucumber 场景片段复用

编写 feature 有一个痛点：有的固定语句组合会出现在多个 feature 中。Cucumber 没有提供类似编程可以抽象公用方法的功能，这不利于用例的编写和维护。我们的解决方案是在原来的语法规则上做扩展，通过新增编译过程，把使用了场景片段复用功能的 feature 转义成标准语法的 feature。

6）支持业务方做语句扩展

MEC 作为通用的 BDD 解决方案，提供了 30+ 条开箱即用的自然语句，但对于业务方，也有封装与业务强相关的语句的需求。针对这样的使用场景，MEC 提供了 API，方便业务方对自然语言做扩展。

7）执行报告

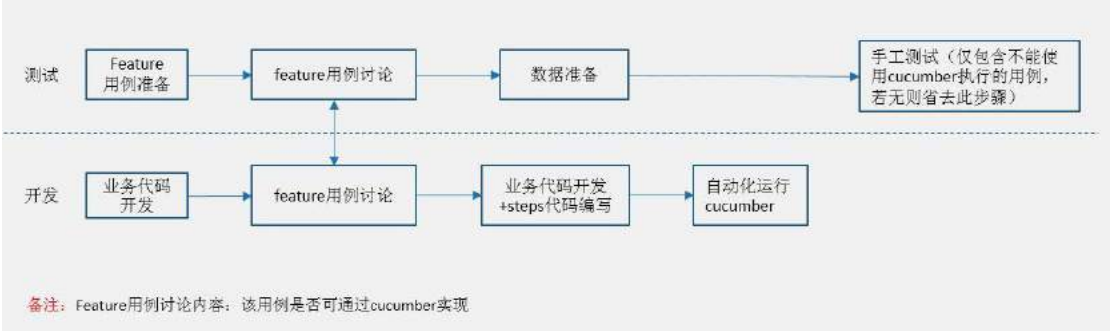
MEC 提供了报告模板，用例运行结束会生成直观的运行结果报告。

8）文档和推广

为了向团队中的非技术人员和技术人员推广 BDD 模式，帮助手工测试利用 BDD 转型自动化测试，我们提供了接入文档。

四、测试开发同时进行

BDD 意味着相对于开发环节，测试可以同步进行或者先行；使用自然语言编写 feature 意味着原本的功能测试人员可以较容易的参与自动化测试。现在，研发流程从之前的先开发后测试演变为测试开发同时进行：



五、回顾

随着软件过程中引入 BDD，feature 文档统一了各方的沟通语言并作为一份活的文档，保持着与线上软件行为的一致，让各方更容易达成共识；研发模式的改变让测试开发工作可以同时进行，减少了发布前夕才发现问题带来的风险；质量保证环节从手工测试为主到自动化为主，降低了发布的成本并提高了准确性。

现在，国际机票预订主流程 UI 自动化测试用例覆盖率达到 90%+，集成测试成本降低了 75%。

附录

1、Macaca https://macacajs.github.io

2、Cucumber https://cucumber.io

注：“携程技术”微信公众号后台回复“bdd”，下载讲师 PPT。

携程酒店 DevOps 测试实践

【作者简介】王幸福，携程酒店研发部高级测试经理，负责无线自动化测试相关工作。在测试框架和平台研发、移动测试、DevOps 等领域有着丰富的经验。

如今很多大型互联网公司、创新型企业都在积极地进行 DevOps 实践和落地。为什么 DevOps 如此受青睐？我们该如何实施 DevOps？DevOps 中 Dev 代表开发，Ops 代表运维，那么在这个崭新的流程体系中，QA 又该如何找到自己的位置？带着这些疑问和困惑，我们希望在本文中都能进行探索和解答。

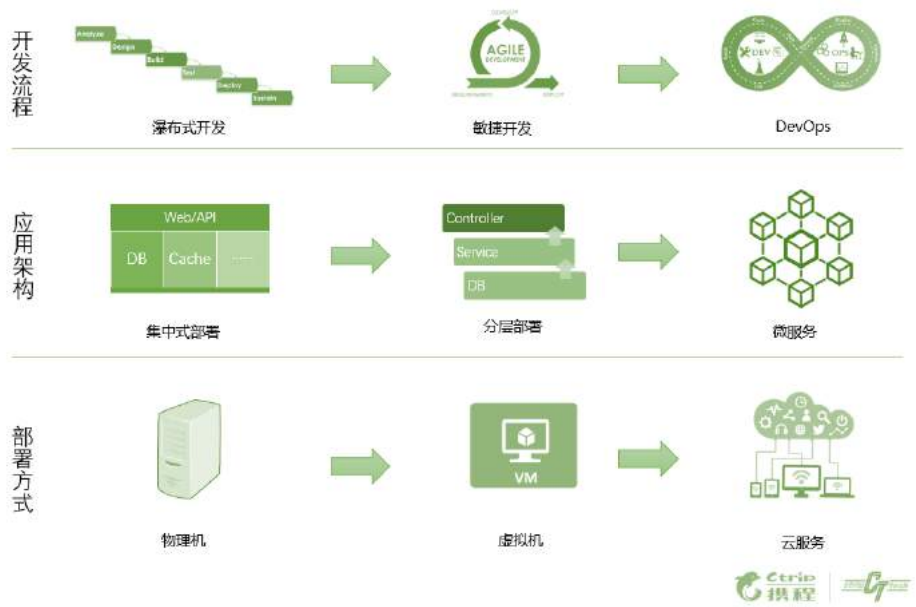
一、业务和技术变革驱动流程的变革

以往在软件开发的世界里，以月甚至以年为周期进行发布是一种常态。但随着近些年由云、移动互联网、AI、社交技术以及区域链等技术推动的业务变革呈现爆炸式的发展。在这种大背景下，即使是大型的互联网公司也随时面临着业务上被淘汰的危险，持续的业务创新，快速的上线，卓越的用户体验以及快速的获得反馈才是企业制胜的法宝。

业务在高速变革，那么技术怎么样呢？技术的变革比之业务，有过之而无不及。应用架构从以往的服务集中到如今盛行的微服务，IT 架构从物理机、虚拟机到如今的容器化、云服务，开发技术栈无论是前端还是后端也都呈现百花齐放的姿态。

无论是业务变革还是技术变革，最终都会对企业的开发流程造成影响，并进而推动其进行变革。从早期的瀑布式开发，到敏捷开发，再到如今的 DevOps，其产生的背景无一不都有着业务和技术变革的影子。

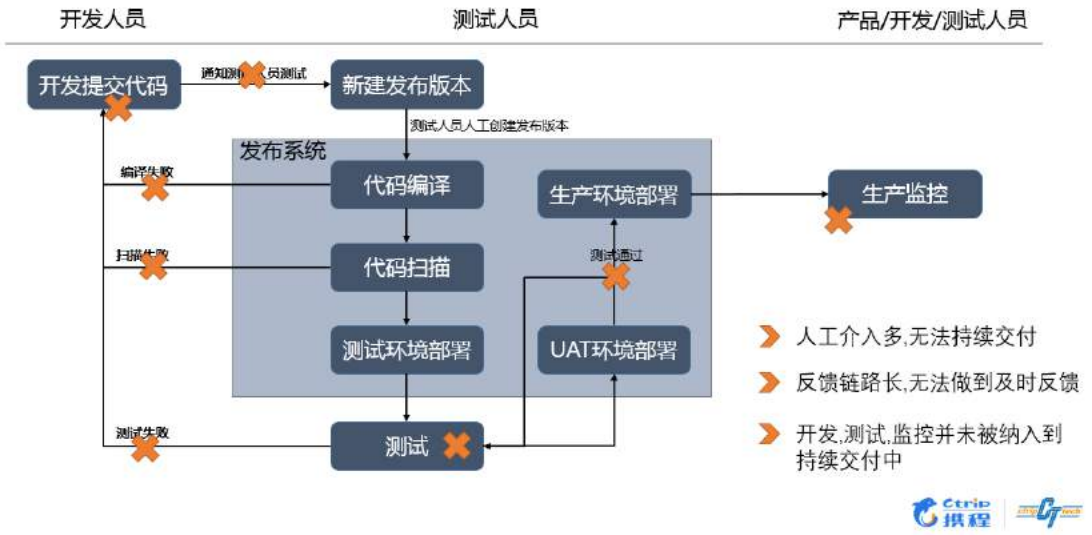
为什么当前我们需要 DevOps，甚至很多大型的互联网公司也在进行 DevOps 转型，其中最关键是因为其核心思想能够满足当前业务和技术变革的需要，那就是“快速的交付价值，灵活的响应变化”。“快速的交付价值”意味着能先人一步占领市场，“灵活的响应变化”亦意味着减少变化带来的不利因素，使企业立于不败之地。



业务和技术变革推动流程变革

二、携程持续交付的现状和挑战

携程在很久以前就已经开始进行持续交付的建设，应用部署全部实现了容器化，并实现了一套自动化程度较高的持续集成发布系统-Ctrip CD(后面简称CD)，CD发布流程如下图所示：



携程发布流程图

开发人员在功能开发完成并提交代码后，可以自己操作或通知测试进行环境部署。进行环境部署的人员可以在CD中创建发布版本，然后由CD自动进行代码编译，代码扫描，安全扫描，测试环境部署等操作。测试人员完成测试后进行测试结果的反馈。如果测试通过继续通过CD进行UAT环境的部署，进行验证测试。测试通过后，发布生产环境。

从上面流程图可以看出，整个发布过程自动化程度还是较高的，相关人员只要在CD中操作

新建版本，关注发布状态就可以了。但我们仔细分析这个过程还是能发现不少问题：

1）持续集成的反馈链路过长。我们往往希望在开发人员每次提交代码时就进行代码编译，代码扫描，单元测试等过程，而不是在功能开发完成后进行。

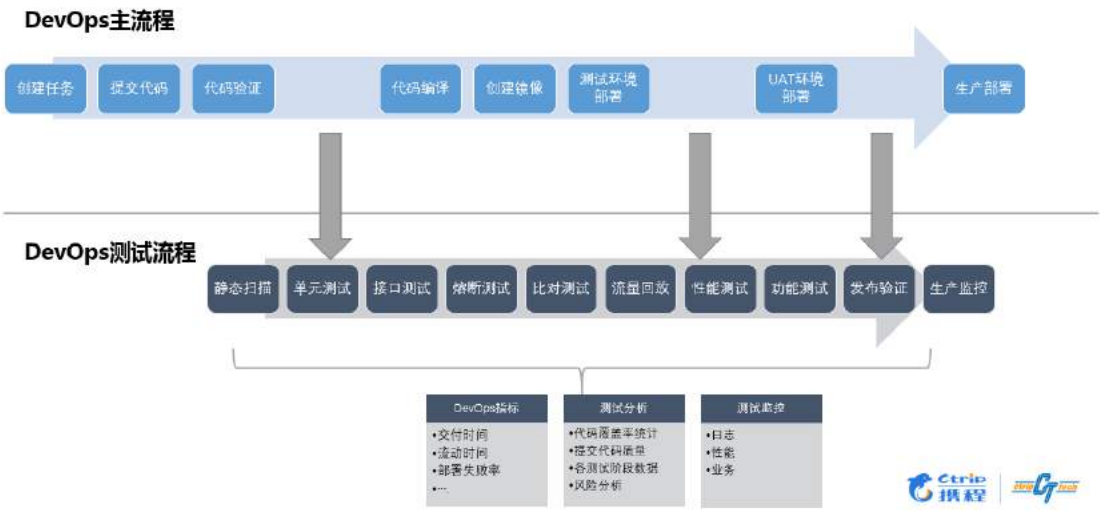
2）人工介入依然过多。虽然在 CD 中可以完成大部分的编译，发布，部署等繁复且人工易出错的工作，但是否可以省略人工创建版本，测试环境手动测试，进而每次提交代码都触发一系列的操作，发布到 UAT 环境，甚至是生产环境（对于业务简单，单元测试和接口测试的应用）。

3）CD 发布系统解决了编译，部署，环境治理等大部分 OPS 相关的工作，但却没有考虑到如何把开发，测试以及发布后的监控等活动整合起来。

上面提到的这些问题，也是携程希望引入 Auto DevOps 的原因之一。DevOps 所提倡的“持续集成，持续测试，持续交付，持续部署”可以很好地解决这些问题，使整个研发效率提升。

三、DevOps 测试流程

携程 DevOps 是基于 GitLab CI/CD 为主干实现的，并针对携程内部的情况进行了二次开发，实现 auto devops 的能力。本文关注的重点是在 DevOps 过程中的测试实践，所以在此就不赘述携程 DevOps 的实现细节，仅列出携程 DevOps 的主干流程。



携程 DevOps 流程及测试流程图

DevOps 虽然从字面意义上看更着重于开发与运维的融合，但事实上却并非如此。DevOps 可以看成是开发、运维和 QA 三者的交集，所以 DevOps 实施成功的关键在于各个阶段都不能有短板。DevOps 通过自动化来实现“持续交付”的流程，那么自动化的手段中自然也包括测试的自动化。其倡导的“持续测试”也需要我们尽可能多的使用自动化手段来快速的发现和反馈问题，从而保障交付产品的质量。

我认为“持续测试”并不仅仅是频度上的持续，还包括开发过程上的持续。我们希望在开发过程的各个阶段都可以有测试的介入，“测试左移”和“测试右移”的思想也由此而来。那么在携程 DevOps 流程中，我们根据自身的情况分三个阶段来进行测试的介入。

第一阶段开发提交代码时，触发静态扫描（Sonar，Infer，代码风险扫描等），安全扫描，单元测试。如果这些测试不通过，将通知开发进行修复。

第二阶段开发提交代码后，经过编译并部署到测试环境时，触发接口测试、熔断测试、比对测试、性能测试等。

第三阶段测试环境测试通过后，发布到生产的镜像环境，在此环境进行流量回放测试，并进行发布前的验证确认工作，验证通过后可以进行生产发布。同时进行生产环境各项指标的监控工作。

在整个过程中，我们也会收集 DevOps 指标数据，日志，性能，测试数据，进行测试分析，通过算法进行风险评估，从而为提高测试决策质量，效率提升提供依据。

俗话说“无规矩不成方圆”，流程的制定只是搭了个架子。在这个架子下，我们还需要制定一系列流程标准来充实它，这也是相对比较困难的部分。因为制定的这些标准需要取得整个部门甚至整个公司的认可，并作为规范来严格的执行，这势必对现有的流程和规范造成很大的影响，推广难度还是比较大的。所以如有必要，甚至可以成立质量委员会来统一制定这些标准，并密切监控实施的过程，遇到问题和困难可以一起想办法解决。

那么通常的标准有哪些呢？归纳下来，这些标准包括：

提测标准：静态扫描结果，单元测试通过率，覆盖率，接口测试结果…
测试规范：探索测试，用例执行，接口测试分析，性能测试…
发布规范：风险分析，发布检查项…
监控规范：业务，性能，日志数据收集，预警的条件….

四、携程酒店 DevOps 测试平台– Moss

有了流程和标准，我们就夯实了实施 DevOps 的基础。接下来需要一个平台来实践这些流程和标准，可以选择 Gitlab CI/CD，也可以选择 Jenkins，亦或者 Gitlab 与 Jenkins 结合。我们选择了自建平台，理由如下：

1）无论是 Gitlab 还是 Jenkins 都需要进行较复杂的配置文件设置，对于开发和测试人员都有一定的学习成本，所以我们希望通过可视化配置的方式来简化配置过程，这样既能提高配置的效率，也能减少推广的难度。

2）携程酒店测试使用的工具和平台很多都是内部研发的，市面上的 DevOps 平台整合这些工具和平台并没有现成的方案可用。

3）我们希望 DevOps 测试过程并不仅仅是给测试看的，我们希望开发，测试，产品都可以

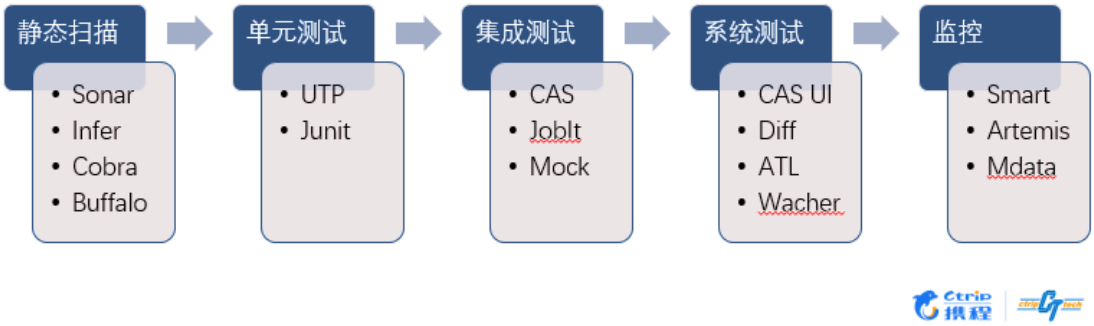
从这个平台中看到自己需要的东西。

4）DevOps 最理想的状态是可以直接自动发布到生产。可目前现实的情况却很少有应用可以做到，那么我们希望提供尽可能多的评估和反馈数据来缩短发布确认的过程。

Moss 平台

4.1 DevOps 测试工具链

在实施 DevOps 过程中会涉及到很多的工具，我们把这些工具形象的称为工具链，而合理的整合工具链中的工具也是 DevOps 是否成功的关键因素之一。在测试各个阶段常用的测试手段通常包括静态代码扫描，单元测试，接口测试，UI 自动化测试，流量回放等等。而这些测试手段在业界都有比较成熟的开源框架，比如 SonarQube、Junit、Selenim、Appnium…. 携程酒店测试根据自身情况，结合这些开源框架开发了一系列的平台和工具。



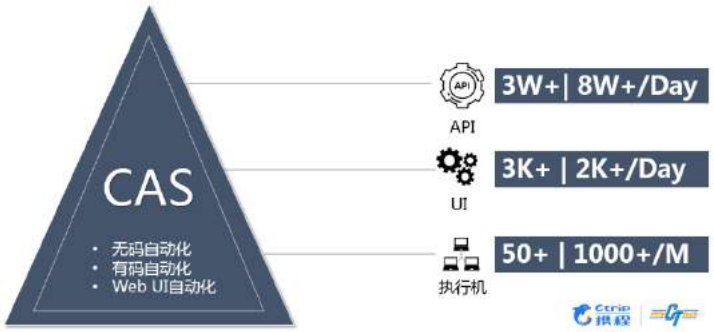
携程酒店 DevOps 测试工具链

静态扫描作为一种近乎零成本的测试手段，可以在早期发现代码中存在的代码缺陷，安全漏洞等问题。在静态扫描领域，SonarQube 已经深耕多年，在这方面已经近乎成为标配。携程通过对原有 SonarQube 代码规范库中的规范进行筛选和扩展，形成了自己代码规范库。我们还有基于开源框架开发的安全扫描工具 Cobra 和 Buffalo。在我们的 DevOps 流程中，开发人员在提交代码后就会触发 Sonar，Infer，Cobra，Buffalo 等一系列的静态扫描手段进行

代码检测。

单元测试随着敏捷开发的盛行而引起了大家的重视，虽然目前在国内对单元测试的重视程度依然欠缺，但从众多大型的开源项目可以看出单元测试确实在软件的开发质量保障方面有着积极的作用。我们为了整合单元测试的编写，执行和结果而开发了 UTP 单元测试平台。该平台由 Junit 扩展库 UtpJunit，IDEA 插件 UtpGenerator 以及 Utp 站点组成。该平台实现了 BDD 驱动，代码分析，在线 WebIDE，单元测试执行，覆盖率统计，报告展示，持续集成等功能。

集成测试阶段主要进行接口测试，数据库测试，Job 测试等等。无论是 RPC，SOA 还是目前流行的微服务，都是在强调对外提供服务的能力。而这种能力主要通过提供对外接口来实现，这也决定了接口测试的重要性。我们为此构建了 CAS 平台，CAS 平台是一个同时支持有码和无码接口自动化测试的平台。



CAS 自动化测试平台

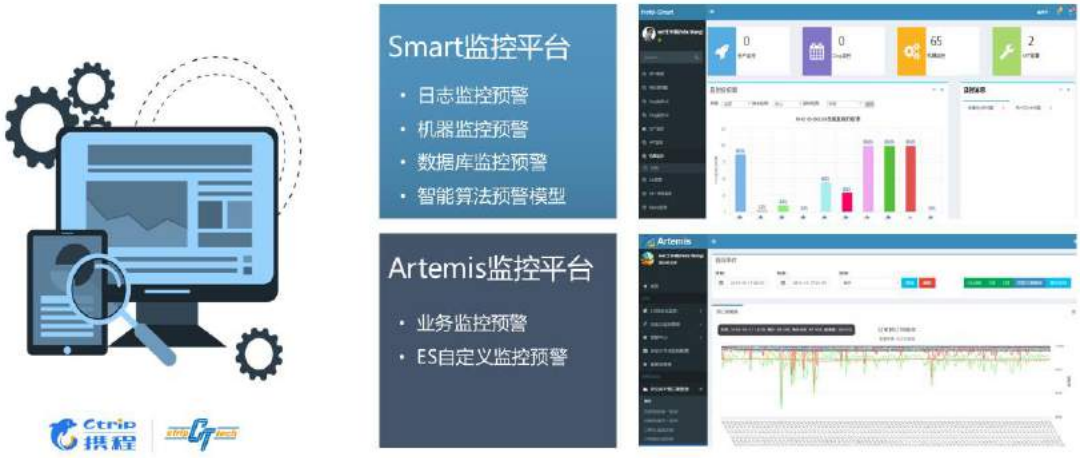
测试过程中一个比较难以解决的困难是测试数据问题。为了保障接口测试和 UI 自动化测试数据的可用性，我们开发了 MOCK 系统，用于测试人员配置和管理测试数据。

系统测试阶段是测试人员介入比较多的阶段，也是测试人员比较热衷做自动化测试的阶段。因此这个阶段的自动化测试框架也比较的多。常用的 Web 自动化框架有 Selenim，Jest，Jasmine…，常用的 APP 自动化测试框架有 Appnium，Airtest，Clabash，UIAutomator…. 而这些自动化测试框架是百花齐放，各有所长，要根据自身团队的实际情况慎重选择适合自己的框架。

在 Web 自动化测试方面，我们选择了 Selenium 框架作为基础进行二次开发，而在 APP 自动化测试方面，我们构建了自己内部的测试云平台- ATL(APP Test Lab)，该平台支持设备管理，也同时支持 Appnium 和 Airtest 的用例管理，执行和报告查看。

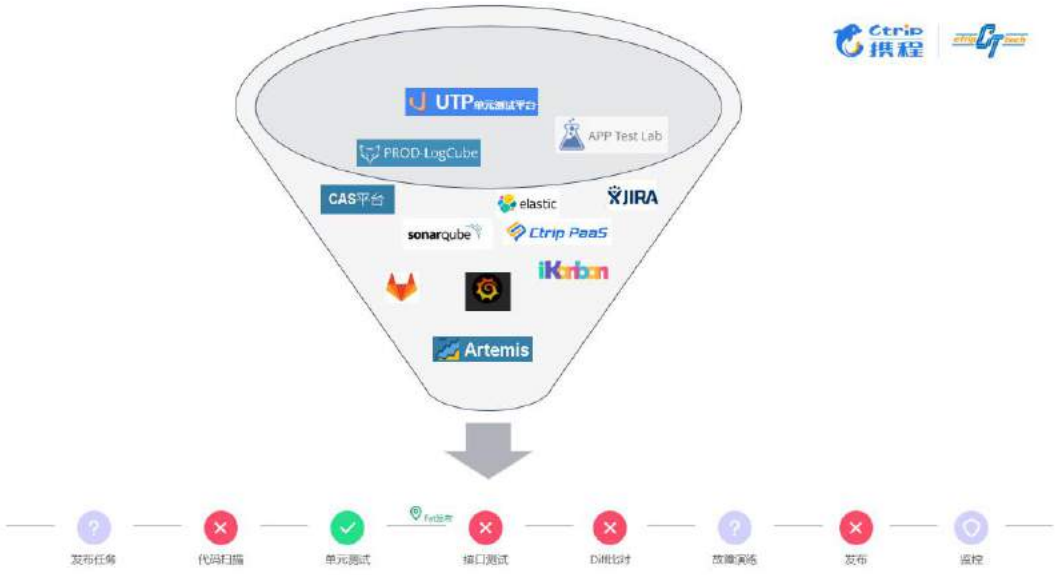
ATL 自动化测试平台

线上监控作为"测试右移"的重要手段之一，正越来越引起很多公司的重视。通常在服务器，网络，框架，性能等方面，OPS 会有众多的监控和预警机制。但在业务，功能等一些特定指标上却无法兼顾到，那么我们就需要自己去监控和预警，这些监控大致可以分为数据库数据监控，埋点监控，接口监控，UI 监控等。



携程酒店测试监控平台

除了以上的工具和平台，我们还有一些经常使用的工具和平台，限于篇幅，不在这里一一介绍。而这么多的工具和平台，以往都是测试人员在各个平台切换使用，容易混乱，效率也低，工具之间无法产生化学反应。我们需要通过 DevOps 把这些工具整合起来，形成工具链，这也就是我们经常提到的 pipeline.

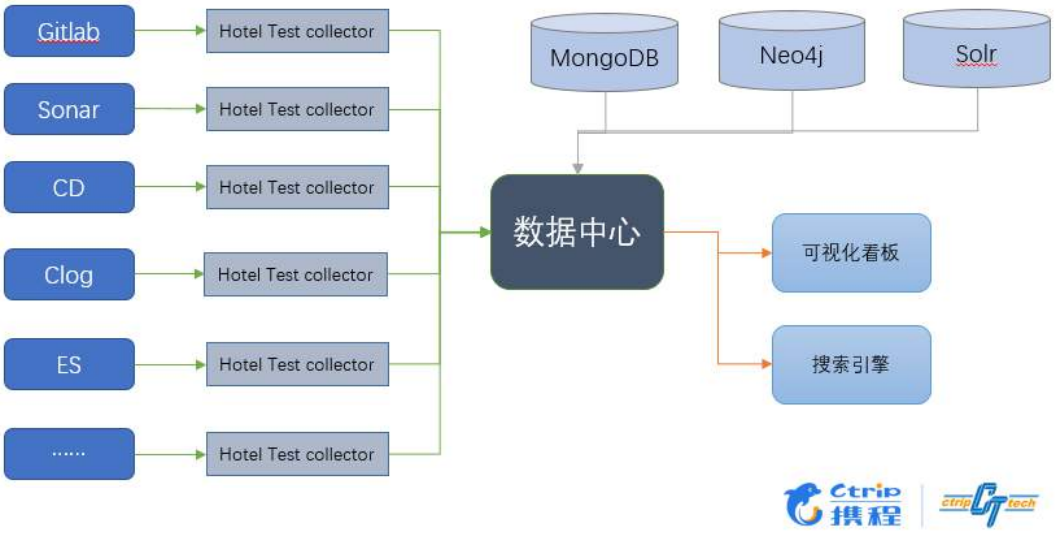


Moss 平台的 pipeline 整合了众多的工具和平台

4.2 DevOps 数据中心

DevOps 的精益思想需要数据支持来减少不必要的浪费，DevOps 是否成功得到实施需要数据来反馈各项指标数据，公司的领导需要知道当前团队代码问题，覆盖率情况，Bug 等数据…等等这些都需要数据。

这些数据来源在哪里？自然来自 DevOps 所整合的各个平台和工具。所以我们需要一个 DevOps 数据中心来收集和分析这些数据，并把数据以可视化的方式展示给相关的人员，让相关人员可以看到自己需要的数据。

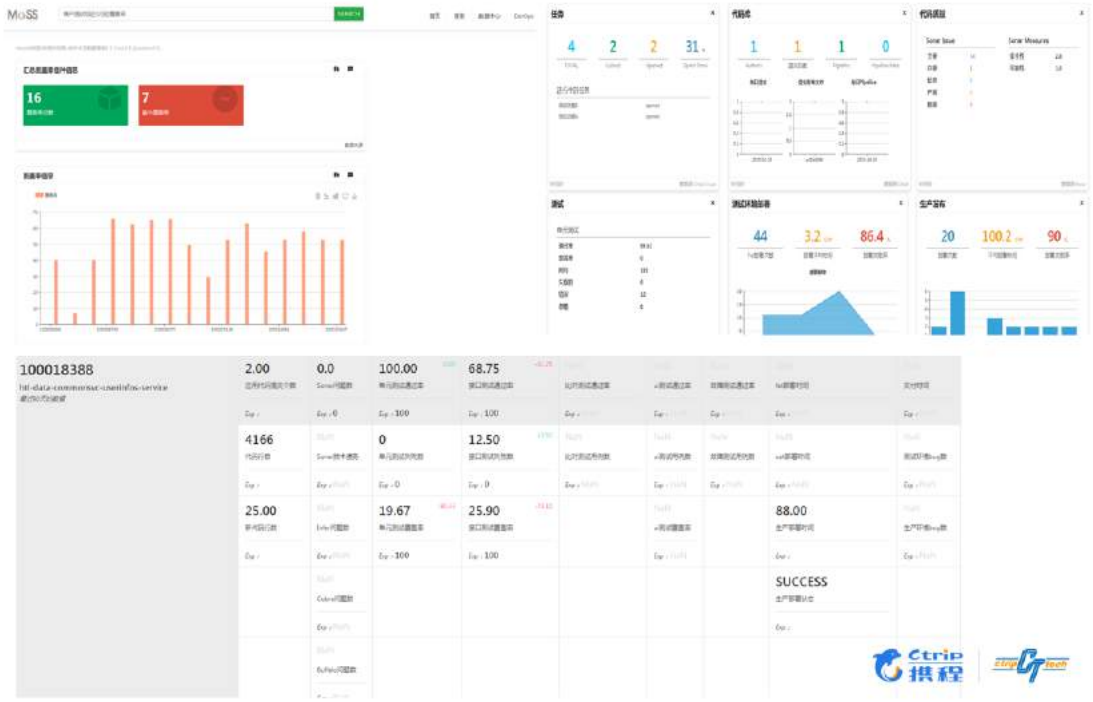


DevOps 数据中心架构示意图

Moss 平台的 DevOps 数据中心通过收集器从各个工具链平台中拉取数据存放到 MongoDB

中。Neo4j 是一款 NOSQL 图形数据库，用于存放人与人，人与应用，应用与应用之前的关系和数据，为以后聚合团队数据，数据关联提供支持。

同时 DevOps 数据中心还提供了可视化数据编辑功能，可以让用户以可视化配置的方式来配置数据的可视化看板。而且秉承着一切数据都是可以被搜索的理念，我们提供一个搜索引擎让用户搜索到自己想要的数据，并以可视化的形式展示出来。



应用看板，技术价值流看板

五、总结和未来展望

测试在历经了瀑布式开发，敏捷开发阶段后，其测试体系的基础并没有受到太多的冲击和改变，但在“来势汹汹”的 DevOps 浪潮中，我认为测试的根基已经受到了一定的动摇，过去那种固有的测试思维已经难以适应当下测试的需要。作为测试人，如果不想被时代淘汰，就需要主动去适应这种转变，去积极挖掘测试在 DevOps 体系中的价值。

在实施 DevOps 的过程中，我们也遇到了很多的困难和挑战，同时也收获了很多的经验和教训。总结下来主要有这么几点：

- 高度自动化，尽可能减少人为干预
- 需要快速且准确的反馈问题
- 要制定 DevOps 流程中可行的规范
- 关注 DevOps 指标，优化流程和提高效率

目前，我们实施的 DevOps 还处于初期阶段，很多方面尚未完全达到预期。在不久的将来我们还有很多的工作需要去做：

- 进一步完善流程标准和挖掘数据来提高效率和软件质量
- 采用机器学习来实现基于风险和变更的测试策略
- 进一步加强质量可视化实现
- 基于 Moss 的数据整合能力，实现监控一体化
- 开发 Chrome 插件 Moss Detector，进一步加强用户在 DevOps 中的交互和效率提升

注：“携程技术”微信公众号后台回复“devops”，可下载讲师分享 PPT

云计算篇

云计算时代携程的网络架构变迁

【作者简介】赵亚楠，携程云平台资深架构师。2016 年加入携程云计算部门，先后从事 OpenStack、SDN、容器网络（Mesos、K8S）、容器镜像存储、分布式存储等产品的开发，目前带领 Ctrip Cloud Network & Storage Team，专注于网络和分布式存储研发。

本文介绍云计算时代以来携程在私有云和公有云上的几代网络解决方案。希望这些内容可以给业内同行，尤其是那些设计和维护同等规模网络的团队提供一些参考。

本文将首先简单介绍携程的云平台，然后依次介绍我们经历过的几代网络模型：从传统的基于 VLAN 的二层网络，到基于 SDN 的大二层网络，再到容器和混合云场景下的网络，最后是 cloud native 时代的一些探索。

一、携程云平台简介

携程 Cloud 团队成立于 2013 年左右，最初是基于 OpenStack 做私有云，后来又开发了自己的 baremetal（BM）系统，集成到 OpenStack，最近几年又陆续落地了 Mesos 和 K8S 平台，并接入了公有云。

目前，我们已经将所有 cloud 服务打造成一个 CDOS — 携程数据中心操作系统的混合云平台，统一管理我们在私有云和公有云上的计算、网络 、存储资源。

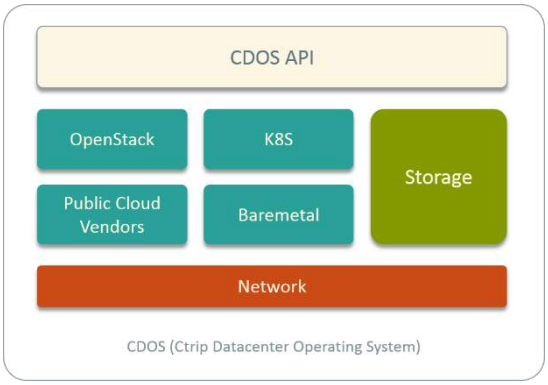


Fig 1. Ctrip Datacenter Operation System (CDOS)

在私有云上，我们有虚拟机、应用物理机和容器。在公有云上，接入了亚马逊、腾讯云、UCloud 等供应商，给应用部门提供虚拟机和容器。所有这些资源都通过 CDOS API 统一管理。

网络演进时间线

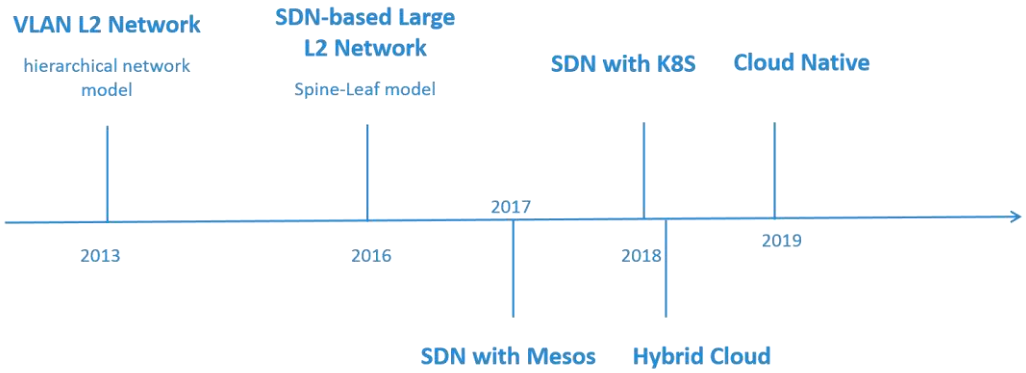


Fig 2. Timeline of the Network Architecture Evolution

图 2 是我们网络架构演进的大致时间线。

最开始做 OpenStack 时采用的是简单的 VLAN 二层网络，硬件网络基于传统的三层网络模型。

随着网络规模的扩大，这种架构的不足逐渐显现出来。因此，在 2016 年自研了基于 SDN 的大二层网络来解决面临的问题，其中硬件架构换成了 Spine-Leaf。

2017 年，我们开始在私有云和公有云上落地容器平台。在私有云上，对 SDN 方案进行了扩展和优化，接入了 Mesos 和 K8S 平台，单套网络方案同时管理了虚拟机、应用物理机和容器网络。公有云上也设计了自己的网络方案，打通了混合云。

最后是 2019 年，针对 Cloud Native 时代面临的一些新挑战，我们在调研一些新方案。

二、基于 VLAN 的二层网络

2013 年我们开始基于 OpenStack 做私有云，给公司的业务部门提供虚拟机和应用物理机资源。

2.1 需求

网络方面的需求有：

首先，性能不能太差，衡量指标包括 instance-to-instance 延迟、吞吐量等等。
第二，二层要有必要的隔离，防止二层网络的一些常见问题，例如广播泛洪。
第三，实例的 IP 要可路由，这点比较重要。这也决定了在宿主机内部不能使用隧道技术。
第四，安全的优先级可以稍微放低一点。如果可以通过牺牲一点安全性带来比较大的性能提升，在当时也是可以接受的。而且在私有云上，还是有其他方式可以弥补安全性的不足。

2.2 解决方案：OpenStack Provider Network 模型

经过一些调研，我们最后选择了 OpenStack provider network 模型 [1]。

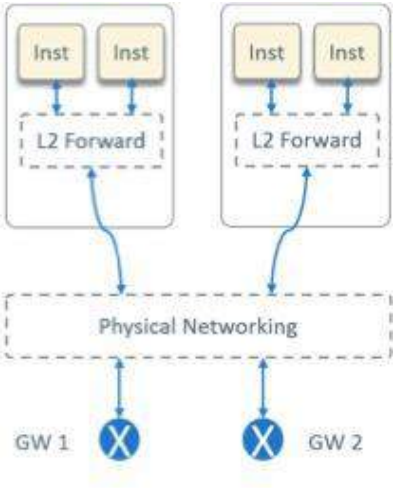


Fig 3. OpenStack Provider Network Model

如图 3 所示。宿主机内部走二层软交换，可以是 OVS、Linux Bridge、或者特定厂商的方案；宿主机外面，二层走交换，三层走路由，没有 overlay 封装。

这种方案有如下特点：

首先，租户网络的网关要配置在硬件设备上，因此需要硬件网络的配合，而并不是一个纯软件的方案；
第二，实例的 IP 是可路由的，不需要走隧道；
第三，和纯软件的方案相比，性能更好，因为不需要隧道的封装和解封装，而且跨网段的路由都是由硬件交换机完成的。

方案的一些其他方面：
1）二层使用 VLAN 做隔离
2）ML2 选用 OVS，相应的 L2 agent 就是 neutron ovs agent
3）因为网关在硬件交换机上，所以我们不需要 L3 agent（OpenStack 软件实现的虚拟路由器）来做路由转发
4）不用 DHCP
5）没有 floating ip 的需求
6）出于性能考虑，我们去掉了 security group

2.3 硬件网络拓扑

图 4 是我们的物理网络拓扑，最下面是服务器机柜，上面的网络是典型的接入-汇聚-核心三层架构。

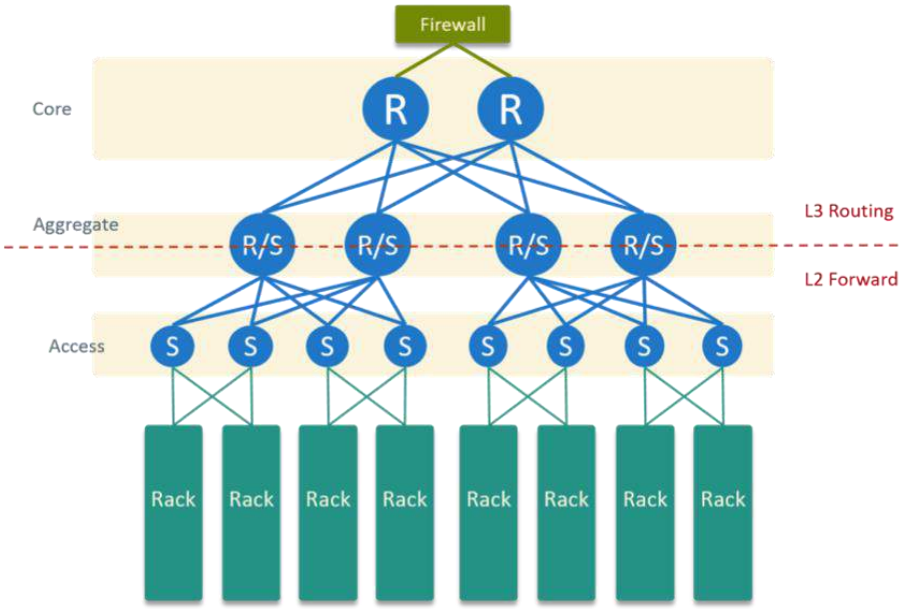


Fig 4. Physical Network Topology in the Datacenter

特点:

1）每个服务器两个物理网卡，直连到两个置顶交换机做物理高可用

2）汇聚层和接入层走二层交换，和核心层走三层路由

3）所有 OpenStack 网关配置在核心层路由器

4）防火墙和核心路由器直连，做一些安全策略

2.4 宿主机内部网络拓扑

再来看宿主机内部的网络拓扑。图 5 是一个计算节点内部的拓扑。

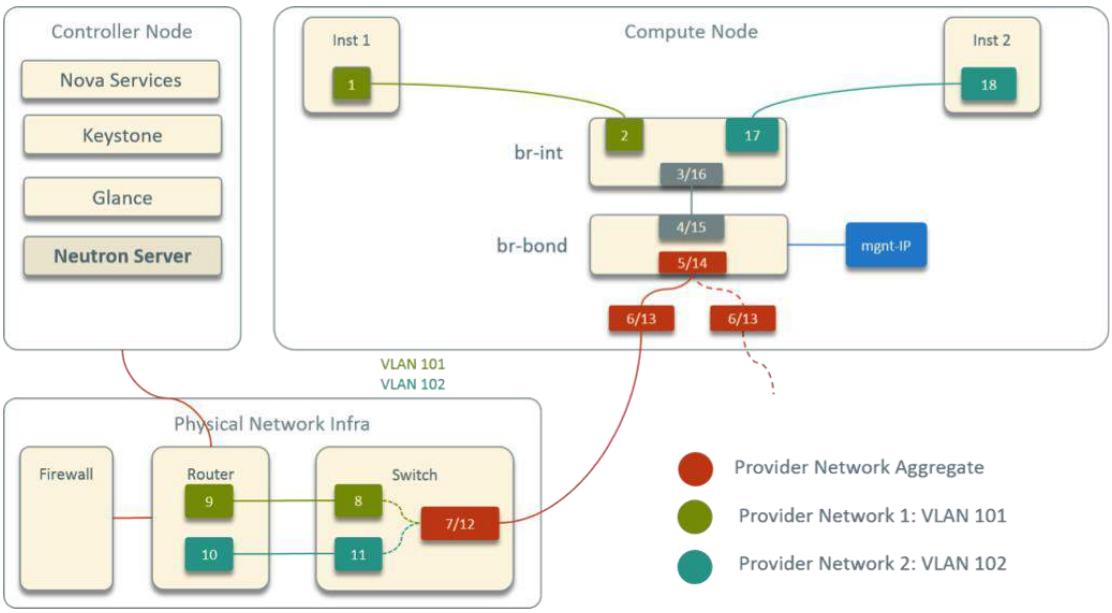


Fig 5. Designed Virtual Network Topology within A Compute Node

特点：
1）首先，在宿主机内有两个 OVS bridge：br-int 和 br-bond，两个 bridge 直连
2）有两个物理网卡，通过 OVS 做 bond。宿主机的 IP 配置在 br-bond 上作为管理 IP
3）所有实例连接到 br-int

图中的两个实例属于不同网段，这些标数字的（虚拟和物理）设备连接起来，就是两个跨网段的实例之间通信的路径：inst1 出来的包经 br-int 到达 br-bond，再经物理网卡出宿主机，然后到达交换机，最后到达路由器（网关）；路由转发之后，包再经类似路径回到 inst2，总共是 18 跳。

作为对比，图 6 是原生的 OpenStack provider network 模型。

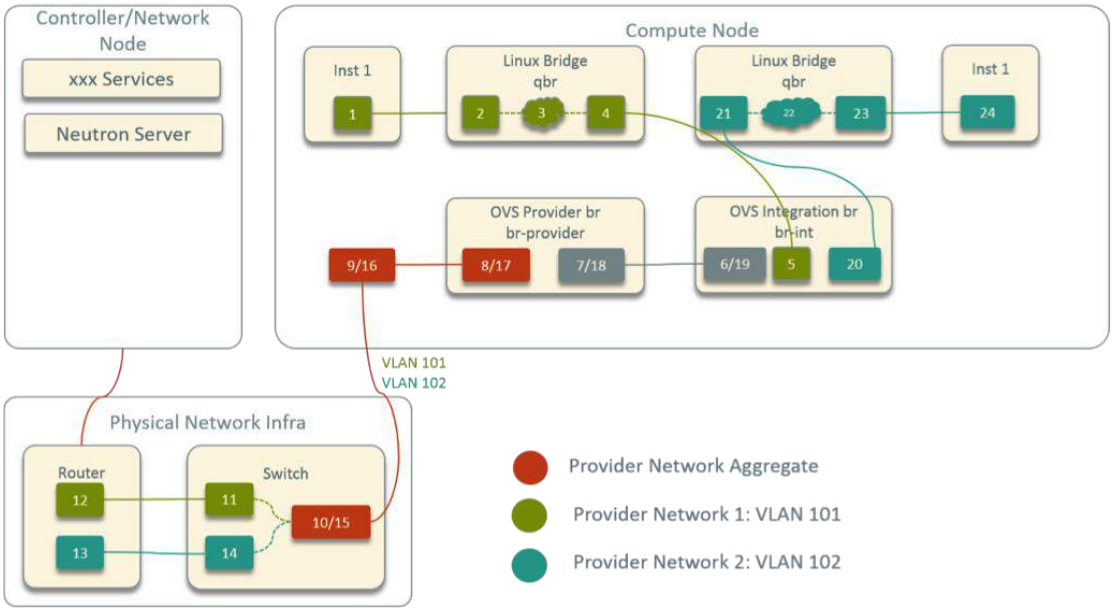


Fig 6. Virtual Network Topology within A Compute Node in Legacy OpenStack

这里最大的区别就是每个实例和 br-int 之间都多出一个 Linux bridge。这是因为原生的 OpenStack 支持通过 security group 对实例做安全策略，而 security group 底层是基于 iptables 的。OVS port 不支持 iptables 规则，而 Linux bridge port 支持，因此 OpenStack 在每个实例和 OVS 之间都插入了一个 Linux Bridge。在这种情况下，inst1 -> inst2 总共是 24 跳，比刚才多出 6 跳。

2.5 小结

最后总结一下我们第一代网络方案。

优点：

首先，我们去掉了一些不用的 OpenStack 组件，例如 L3 agent、HDCP agent, Neutron meta agent 等等，简化了系统架构。对于一个刚开始做 OpenStack、经验还不是很丰富的团队来说，开发和运维成本比较低。

第二，上面已经看到，我们去掉了 Linux Bridge，简化了宿主机内部的网络拓扑，这使得转发路径更短，实例的网络延迟更低。

第三，网关在硬件设备上，和 OpenStack 的纯软件方案相比，性能更好。

第四，实例的 IP 可路由，给跟踪、监控等外围系统带来很大便利。

缺点：

首先，去掉了 security group，没有了主机防火墙的功能，因此安全性变弱。我们通过硬件防火墙部分地补偿了这一问题。

第二，网络资源交付过程还没有完全自动化，并且存在较大的运维风险。 provider network 模型要求网关配置在硬件设备上，在我们的方案中就是核心路由器上。因此，每次在 OpenStack 里创建或删除网络时，都需要手动去核心交换机上做配置 。虽然说这种操作频率还是很低的，但操作核心路由器风险很大，核心发生故障会影响整张网络。

三、基于 SDN 的大二层网络

以上就是我们在云计算时代的第一代网络方案，设计上比较简单直接，相应地，功能也比较少。随着网络规模的扩大和近几年我们内部微服务化的推进，这套方案遇到了一些问题。

3.1 面临的新问题

首先来自硬件。做数据中心网络的同学比较清楚，三层网络架构的可扩展性比较差， 而且我们所有的 OpenStack 网关都配置在核心上，使得核心成为潜在的性能瓶颈，而核心挂了会影响整张网络。

第二，很大的 VLAN 网络内部的泛洪，以及 VLAN 最多只有 4096 个的限制。

第三，宿主机网卡比较旧，都是 1Gbps，也很容易达到瓶颈。

另外我们也有一些新的需求：

首先，携程在这期间收购了一些公司，会有将这些收购来的公司的网络与携程的网络打通的需求。在网络层面，我们想把它们当作租户对待，因此有多租户和 VPC 的需求。

另外，我们想让网络配置和网络资源交付更加自动化，减少运维成本与运维风险。

3.2 解决方案：OpenStack + SDN

针对以上问题和需求，数据中心网络团队和我们 cloud 网络团队一起设计了第二代网络方案：一套基于软件+硬件、OpenStack+SDN 的方案，从二层网络演进到大二层网络。

- 硬件拓扑

在硬件拓扑上，从传统三层网络模型换成了近几年比较流行的 Spine-Leaf 架构，如图 7 所示。

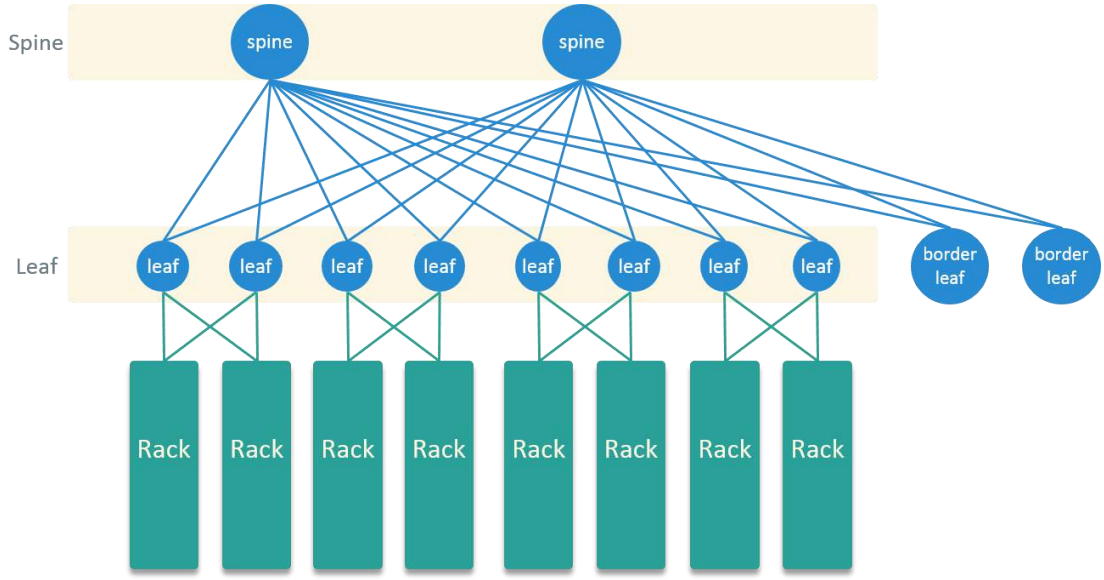


Fig 7. Spine-Leaf Topology in the New Datacenter

Spine-Leaf 是 full-mesh 连接，它可以带来如下几个好处：

第一，转发路径更短。以图 7 的 Spine-Leaf（两级 Clos 架构）为例，任何两台服务器经过三跳（Leaf1 -> Spine -> Leaf2）就可以到达，延迟更低，并且可保障（可以按跳数精确计算出来）。

第二，水平可扩展性更好，任何一层有带宽或性能瓶颈，只需新加一台设备，然后跟另一层的所有设备直连。

第三，所有设备都是 active 的，一个设备挂掉之后，影响面比三层模型里挂掉一个设备小得多。

宿主机方面，我们升级到了 10G 和 25G 的网卡。

- SDN：控制平面和数据平面

数据平面基于 VxLAN，控制平面基于 MP-BGP EVPN 协议，在设备之间同步控制平面信息。网关是分布式的，每个 leaf 节点都是网关。VxLAN 和 MP-BGP EVPN 都是 RFC 标准协议，更多信息参考 [2]。

VxLAN 的封装和解封装都在 leaf 完成，leaf 以下是 VLAN 网络，以上是 VxLAN 网络。

另外，这套方案在物理上支持真正的租户隔离。

- SDN：组件和实现

开发集中在以下几个方面。

首先是自研了一个 SDN 控制器，称作携程网络控制器（Ctrip Network Controller），缩写 CNC。CNC 是一个集中式控制器，管理网络内所有 spine 和 leaf 节点，通过 neutron plugin 与 OpenStack Neutron 集成，能够动态向交换机下发配置。

Neutron 的主要改造：
1）添加了 ML2 和 L3 两个 plugin 与 CNC 集成
2）设计了新的 port 状态机，因为原来的 port 只对 underlay 进行了建模，我们现在有 underlay 和 overlay 两个平面
3）添加了一下新的 API，用于和 CNC 交互
4）扩展了一些表结构等等

图 8 就是我们对 neutron port 状态的一个监控。如果一个 IP（port）不通，我们很容易从它的状态看出问题是出在 underlay 还是 overlay。

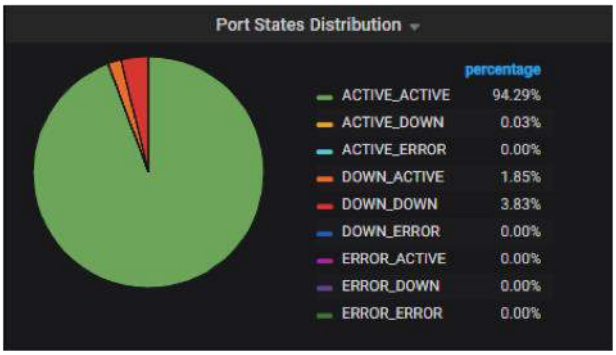


Fig 8. Monitoring Panel for Neutron Port States

3.3 软件+硬件网络拓扑

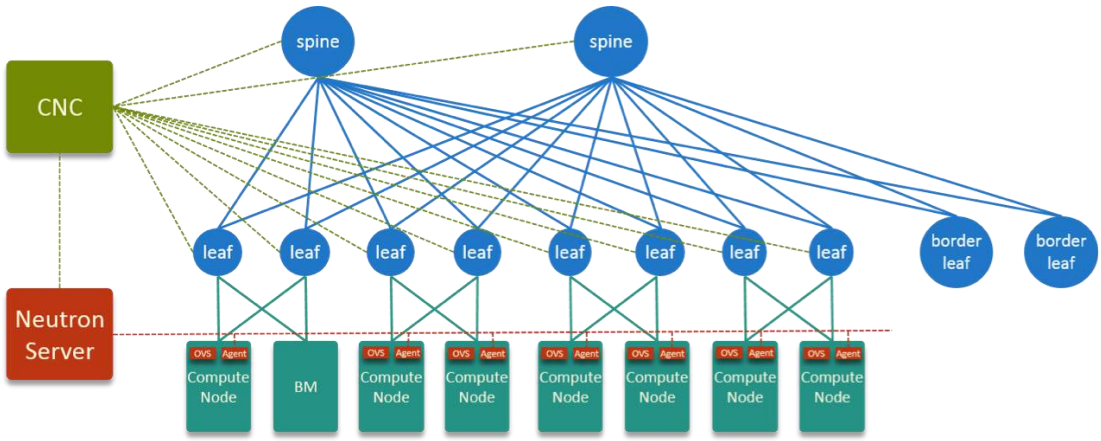


Fig 9. HW + SW Topology of the Designed SDN Solution

图 9 是我们软件+硬件的网络拓扑:

1）以 leaf 为边界，leaf 以下是 underlay，走 VLAN；上面 overlay，走 VxLAN
2）underlay 由 neutron、OVS 和 neutron OVS agent 控制；overlay 是 CNC 控制
3）Neutron 和 CNC 之间通过 plugin 集成

3.4 创建实例涉及的网络配置流程

这里简单来看一下创建一个实例后，它的网络是怎么通的。图 10 中黑色的线是 OpenStack 原有的逻辑，蓝色的是我们新加的。

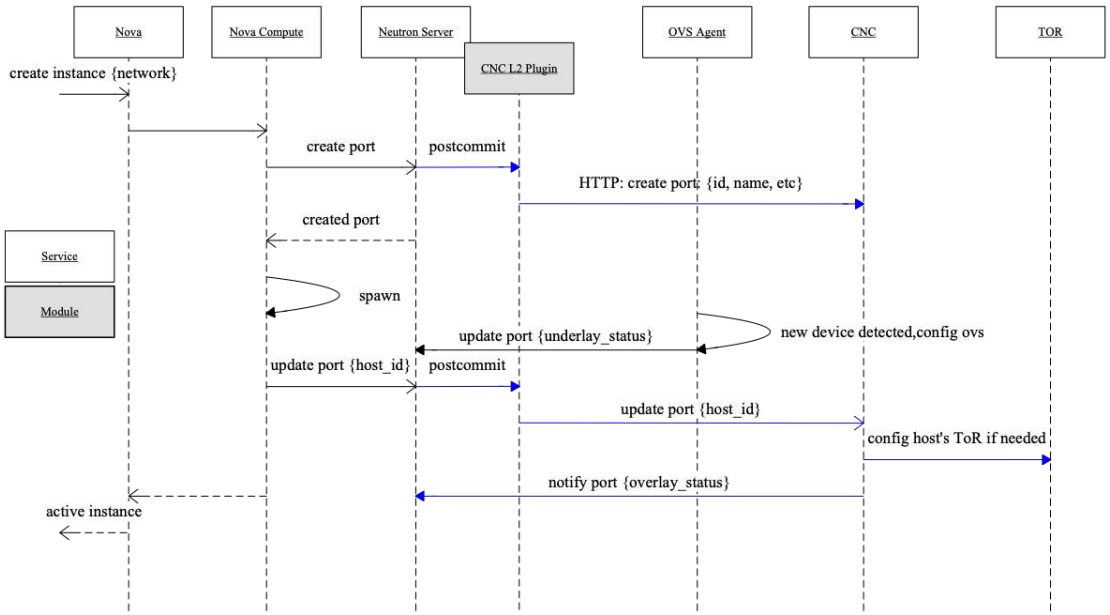


Fig 10. Flow of Spawn An Instance

1) 控制节点: 从 nova 发起一个创建实例请求, 指定从哪个网络分配 IP 给这个实例。nova 调度器将任务调度到某台计算节点
2）计算节点：nova compute 开始创建实例，其中一步是向 neutron 发起一个 create port 请求，简单认为就是申请一个 IP 地址
3）Neutron Server: neutron 创建一个 port，然后经 create port 的 postcommit 方法 到达 CNC ML2 plugin；plugin 进一步将 port 信息同步给 CNC，CNC 将其存储到自己 的 DB
4）计算节点：port 信息从 neutron server 返回给 nova-compute
5)计算节点:nova-compute 拿到 port 信息, 为实例创建虚拟网卡, 配置 IP 地址等参数 ，并将其 attach 到 OVS
6）计算节点：ovs agent 检测到新的 device 后，就会为这个 device 配置 OVS，添加 flow 等，这时候 underlay 就通了，它会将 underlay 状态上报给 neutron server
7）计算节点：nova-compute 做完网络配置后，会发送一个 update port 消息给 neutron server，其中带着 host_id 信息，表示这个 port 现在在哪台计算节点上
8）Neutron Server：请求会通过 postcommit 发给 CNC

9）CNC：CNC 根据 host_id 找到这台计算节点所连接的 leaf 的端口，然后向这些端口动态下发配置，这时候 overlay 就通了，最后将 overlay 状态上报给 neutron server

在我们的系统里看，这时 port 就是一个 ACTIVE_ACTIVE 的状态，表示 underlay 和 overlay 配置都是正常的，网络应该是通的。

3.5 小结

总结一下我们这套方案。

- 硬件

首先，我们从三层网络架构演进到 Spine-Leaf 两级架构。Spine-Leaf 的 full-mesh 使得服务器之间延迟更低、容错性更好、更易水平扩展。另外，Spine-Leaf 还支持分布式网 关，缓解了集中式网关的性能瓶颈和单点问题。

- 软件

自研 SDN 控制器并与 OpenStack 集成，实现了网络的动态配置。这套方案同时支持虚拟机和应用物理机部署系统，限于篇幅这里只介绍了虚拟机。

- 多租户

有硬多租户（hard-multitenancy）支持能力。

四、容器和混合云网络

以上方案最开始还是针对虚拟机和应用物理机设计的。到了 2017 年，我们开始在私有云和公有云上落地容器平台，将一部分应用从虚拟机或应用物理机迁移到容器。

容器平台（K8S、Mesos 等）有不同于虚拟机平台的一些特点，例如：

1）实例的规模很大，单个集群 10k ~ 100k 个容器是很常见的；
2）很高的发布频率，实例会频繁地创建和销毁；
3）实例创建和销毁时间很短，比传统的虚拟机低至少一个数量级；
4）容器的失败是很常见，总会因为各种各样的原因导致容器挂掉。容器编排引擎在设计的时候已经把失败当做预期情况处理，例如将挂掉的容器在本机或其他宿主机再拉起来， 后者就是一次漂移；

4.1 私有云的 K8S 网络方案

容器平台的这些特点对网络提出了新的需求。

4.1.1 网络需求

首先，网络服务的 API 必须要快，而且支持较大的并发。

第二，不管是 agent 还是可执行文件，为容器创建和删除网络（虚拟网络及相应配置）也要快。

最后是一个工程问题：新系统要想快速落地，就必须与很多线上系统保持前向兼容。这给我们网络提出一个需求就是：容器漂移时，IP 要保持不变。因为 OpenStack 时代， 虚拟机迁移 IP 是不变的，所以很多外围系统都认为 IP 是实例生命周期中的一个不变量， 如果我们突然要改成 IP 可变，就会涉及大量的外围系统（例如 SOA）改造，这其中很多不 是我们所能控制的。因此为了实现容器的快速落地，就必须考虑这个需求。而流行的 K8S 网络方案都是无法支持这个功能的，因为在容器平台的设计哲学里，IP 地址已经是一个被 弱化的概念，用户更应该关心的是实例暴露的服务，而不是 IP。

4.1.2 解决方案：扩展现有 SDN 方案，接入 Mesos/K8S

在私有云中，我们最终决定对现有的为虚拟机和应用物理机设计的 SDN 方案进行扩展，将容器网络也统一由 Neutron/CNC 管理。具体来说，会复用已有的网络基础设施，包括 Neutron、CNC、OVS、Neutron-OVS-Agent 等待，然后开发一个针对 Neutron 的 CNI 插件 （对于 K8S）。

一些核心改动或优化如下。

- Neutron 改动

首先是增加了一些新的 API。比如，原来的 neutron 是按 network id 分配 IP，我们给 network 添加了 label 属性，相同 label 的 network 我们认为是无差别的。这样，CNI 申请 IP 的时候，只需要说“我需要一个 ‘prod-env’ 网段的 IP”，neutron 就会从打了“ prod-env” label 的 network 中任选一个还没用完的，从中分一个 IP。这样既将外部系统 与 OpenStack 细节解耦，又提高了可扩展性，因为一个 label 可以对应任意多个 network 。

另外做了一些性能优化，例如增加批量分配 IP 接口、API 异步化、数据库操作优化等待。

还有就是 backport 一些新 feature 到 neutron，我们的 OpenStack 已经不随社区一起升级了，都是按需 backport。例如，其中一个对运维和 trouble shooting 非常友好的功能 是 Graceful OVS agent restart。

- K8S CNI for Neutron 插件

开发了一个 CNI plugin 对接 neutron。CNI 插件的功能比较常规：

1）为容器创建 veth pairt，并 attach 到 OVS
2）为容器配置 MAC、IP、路由等信息

但有两个特殊之处：

1) 向 neutron（global IPAM）申请分配和释放 IP，而不是宿主机的本地服务分配（local IPAM）
2）将 port 信息更新到 neutron server

- 基础网络服务升级

另外进行了一些基础架构的升级，比如 OVS 在过去几年的使用过程中发现老版本的几个 bug，后来发现这几个问题在社区也是有记录的：

1）vswitchd CPU 100% [3]
2）流量镜像丢包 [4]

注意到最新的 LTS 版本已经解决了这些问题，因此我们将 OVS 升级到了最新的 LTS。 大家如果有遇到类似问题，可以参考 [3, 4]。

4.1.3 容器漂移

创建一个容器后，容器网络配置的流程和图 10 类似，Nova 和 K8S 只需做如下的组件对应：

- nova -> kube master
- nova-compute -> kubelet
- nova network driver -> CNI

其流程不再详细介绍。这里重点介绍一下容器漂移时 IP 是如何保持不变的。

如图 11 所示，保持 IP 不变的关键是：CNI 插件能够根据容器的 labels 推导出 port name，然后拿 name 去 neutron 里获取 port 详细信息。port name 是唯一的，这个是我们改过的，原生的 OpenStack 并不唯一。

第二个宿主机的 CNI plugin 会根据 name 找到 port 信息，配置完成后，会将新的 host_id 更新到 netron server；neutron 通知到 CNC，CNC 去原来的交换机上删除配置 ，并向新的交换机下发配置。

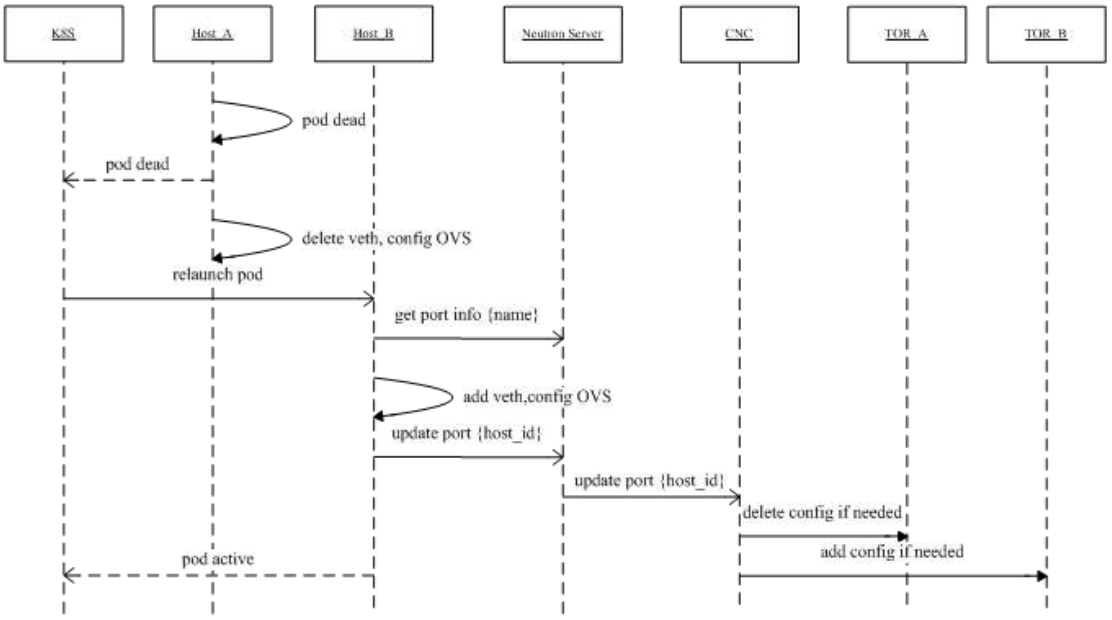


Fig 11. Pod drifting with the same IP within a K8S cluster

4.1.4 小结

简单总结一下:

1）在保持基础设施不变的情况下，我们快速地将容器平台的网络接入到已有系统
2）一个 IPAM 同时管理了虚拟机、应用物理机和容器的网络

目前这套新方案的部署规模:

1）4 个可用区
2）最大的可用区有超过 500 个节点（VM/BM/Container 宿主机），其中主要是 K8S 节点
3）单个 K8S 节点最多会有 500+ pod（测试环境的超分比较高）
4）最大的可用区有 2+ 万个实例，其中主要是容器实例

4.1.5 进一步演进方向

以上就是到目前为止我们私有云上的网络方案演讲，下面这张图是我们希望将来能达到的一个架构。

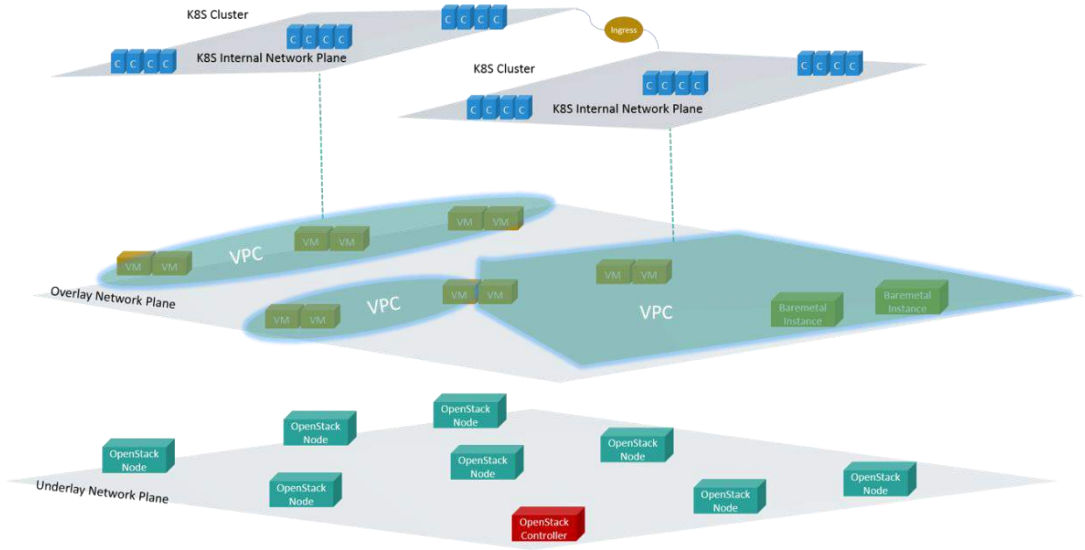


Fig 12. Layered view of the future network architecture

首先会有 underlay 和 overlay 两个平面。underlay 部署各种基础设施，包括 Openstack 控制器、计算节点、SDN 控制器等，以及各种需要运行在 underlay 的物理设备；在 overlay 创建 VPC，在 VPC 里部署虚拟机、应用物理机实例等。

在 VPC 内创建 K8S 集群，单个 K8S 集群只会属于一个 VPC，所有跨 K8S 集群的访问都走服务接口，例如 Ingress，现在我们还没有做到这一步，因为涉及到很多老环境的软件和硬件改造。

4.2 公有云上的 K8S

接下来看一下我们在公有云上的网络。

4.2.1 需求

随着携程国际化战略的开展，我们需要具备在海外部署应用的能力。自建数据中心肯定是来不及的，因此我们选择在公有云上购买虚拟机或 baremetal 机器，并搭建和维护自己的 K8S 集群（非厂商托管方案，例如 AWS EKS [10]）。 在外层，我们通过 CDOS API 封装不同厂商的差异，给应用部门提供统一的接口。这样，我们的私有云演进到了混合云的阶段。

网络方面主要涉及两方面工作：一是 K8S 的网络方案，这个可能会因厂商而已，因为不同厂商提供的网络模型和功能可能不同；二是打通私有云和公有云。

4.2.2 AWS 上的 K8S 网络方案

以 AWS 为例来看下我们在公有云上的 K8S 网络方案。

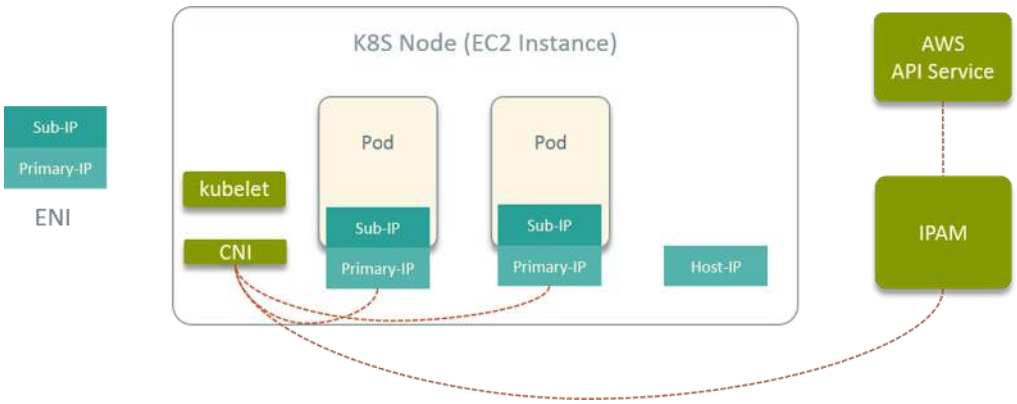


Fig 13. K8S network solution on public cloud vendor (AWS)

首先，起 EC2 实例作为 K8S node，我们自己开发一个 CNI 插件，动态向 EC2 插拔 ENI，并把 ENI 作为网卡给容器使用。这一部分借鉴了 Lyft 和 Netflix 在 AWS 上经验 [5, 6]。

在 VPC 内，有一个全局的 IPAM，管理整个 K8S 集群的网络资源，角色和私有云中的 neutron 类似。它会调用 AWS API 实现网络资源的申请、释放和管理。

另外，我们的 CNI 还支持 attach/detach floating IP 到容器。还有就是和私有云一样， 容器漂移的时候 IP 保持不变。

4.2.3 全球 VPC 拓扑

图 14 是我们现在在全球的 VPC 分布示意图。

在上海和南通有我们的私有云 VPC；在海外，例如首尔、莫斯科、法兰克福、加州（美 西）、香港、墨尔本等地方有公有云上的 VPC，这里画的不全，实际不止这几个 region。

Fig 14. VPCs distributed over the globe

这些 VPC 使用的网段是经过规划的，目前不会跟内网网段重合。因此通过专线打通后，IP

可以做到可路由。

五、Cloud Native 方案探索

以上就是我们在私有云和混合云场景下的网络方案演进。目前的方案可以支持业务未来一段的发展，但也有一些新的挑战。

首先，中心式的 IPAM 逐渐成为性能瓶颈。做过 OpenStack 的同学应该很清楚，neutron 不是为性能设计的，而是为发布频率很低、性能要求不高的虚拟机设计的。没有优化过的话，一次 neutron API 调用百毫秒级是很正常的，高负载的时候更慢。

另外，中心式的 IPAM 也不符合容器网络的设计哲学。Cloud native 方案都倾向于 local IPAM（去中心化），即 每个 node 上有一个 IPAM，自己管理本节点的网络资源分配，calico、flannel 等流行的网络方案都是这样的。

第二，现在我们的 IP 在整张网络都是可漂移的，因此故障范围特别大。

第三，容器的高部署密度会给大二层网络的交换机表项带来压力，这里面有一些大二层设计本身以及硬件的限制。

另外，近年来安全受到越来越高度重视，我们有越来越强的 3-7 层主机防火墙需求。目前基于 OVS 的方案与主流的 K8S 方案差异很大，导致很多 K8S 原生功能用不了。

针对以上问题和需求，我们进行了一些新方案的调研，包括 Calico，Cilium 等等。Calico 大家应该已经比较熟悉了，这里介绍下 Cilium。

5.1 Cilium Overview

Cilium [7]是近两年新出现的网络方案，它使用了很多内核新技术，因此对内核版本要求比较高，需要 4.8 以上支持。

Cilium 的核心功能依赖 BPF/eBPF，这是内核里的一个沙盒虚拟机。应用程序可以通过 BPF 动态的向内核注入程序来完成很多高级功能，例如系统调用跟踪、性能分析、网络拦截等等。Cilium 基于 BPF 做网络的连通和安全，提供 3-7 层的安策略。

Cilium 组件：

1）CLI
2）存储安全策略的 repository，一般是 etcd
3）与编排引擎集成的插件：提供了 plugin 与主流的编排系统（K8S、Mesos 等）集成
4）Agent，运行在每台宿主机，其中集成了 Local IPAM 功能

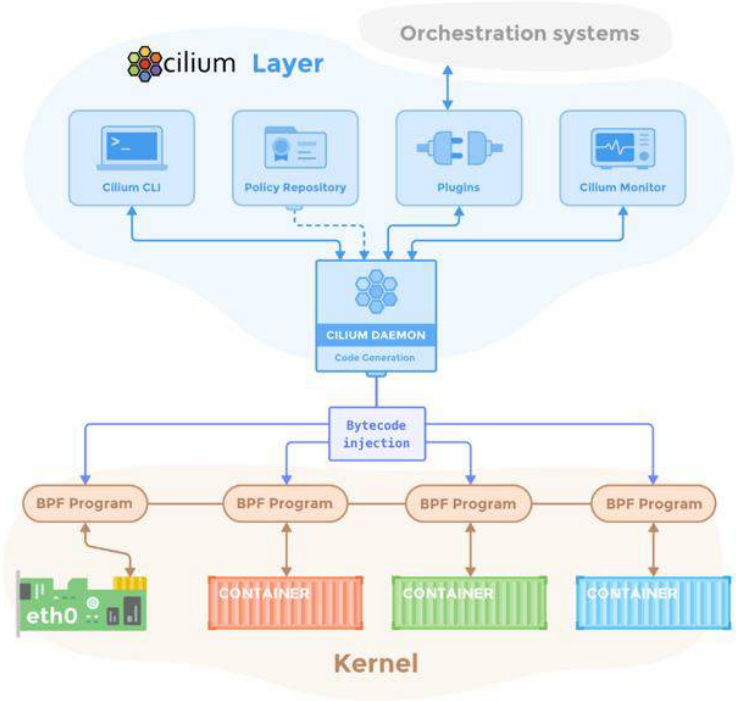


Fig 15. Cilium

5.2 宿主机内部网络通信（Host Networking）

每个网络方案都需要解决两个主要问题：

1）宿主机内部的通信：实例之间，实例和宿主机通信
2）宿主机之间的通信：跨宿主机的实例之间的通信

先来看 Cilium 宿主机内部的网络通信。

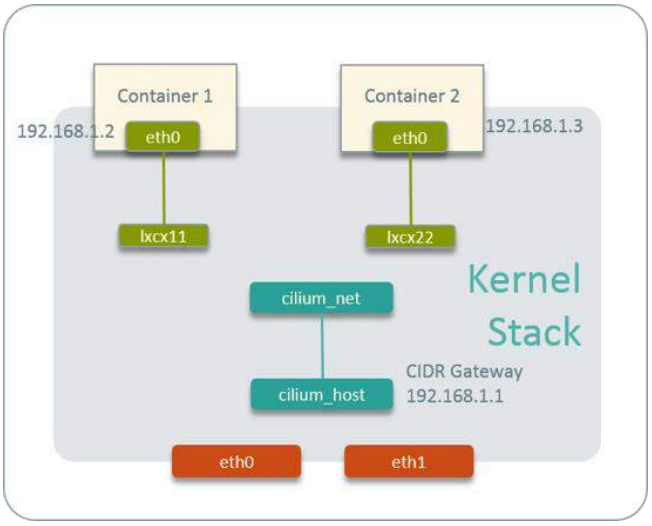


Fig 16. Cilium host-networking

Agent 首先会创建一个 cilium_host <---> cilium_net 的 veth pair，然后将它管理的 CIDR 的第一个 IP 作为网关，配置在 cilium_host 上。对于每个容器，CNI 插件会承担创建 veth pair、配置 IP、生成 BPF 规则等工作。

同宿主机内部的容器之间的连通性靠内核协议栈二层转发和 BPF 程序。比如 inst1 到 isnt2，包首先从 eth0 经协议栈到达 lxc11，中间再经过 BPF 规则到达 lxc22， 然后再经协议栈转发到达 inst2 的 eth0。

以传统的网络观念来看，lxc11 到 lxc22 这一跳非常怪，因为没有既没有 OVS 或 Linux bridge 这样的二层转发设备，也没有 iptables 规则或者 ARP 表项，包就神奇的到达了另一个地方，并且 MAC 地址还被修改了。

类似地，容器和宿主机的通信走宿主机内部的三层路由和 BPF 转发，其中 BPF 程序连接容器的 veth pair 和它的网关设备，因为容器和宿主机是两个网段。

5.3 跨宿主机网络通信（Multi-Host Networking）

跨宿主机的通信和主流的方案一样，支持两种常见方式：

- VxLAN 隧道
- BGP 直接路由

如果使用 VxLAN 方式，Cilium 会创建一个名为 cilium_vxlan 的 device 作为 VTEP， 负责封装和解封装。这种方案需要评估软件 VxLAN 的性能能否接受，以及是否需要 offload 到硬件加速。一般来说，软件 VxLAN 的方式性能较差，而且实例 IP 不可路由。

BGP 方案性能更好，而且 IP 可路由，但需要底层网络支持。这种方案需要在每个 node 上起一个 BGP agent 来和外部网络交换路由，涉及 BGP agent 的选型、AS（自治系 统）的设计等额外工作。如果是内网，一般就是 BGP agent 与硬件网络做 peering；如果 是在 AWS 之类的公有云上，还可以调用厂商提供的 BGP API。

5.4 优劣势比较（Pros & Cons）

最后总结一下 Cilium 方案的优劣势。

- Pros

首先，原生支持 K8S L4-L7 安全策略，例如在 yaml 指定期望的安全效果，Cilium 会自动将其转化为 BPF 规则。

第二，高性能策略下发（控制平面）。Calico/iptables 最大的问题之一就是集群规模大了之后，新策略生效非常慢。iptables 是链式设计，复杂度是 O(n)；而 Cilium 的复杂度是 O(1) [11]，因此性能非常好。

第三，高性能数据平面 (veth pair, IPVLAN)。

第四，原生支持双栈 (IPv4/IPv6)。事实上 Cilium 最开始只支持 IPv6，后面才添加了对 IPv4 的支持，因为他们一开始就是作为下一代技术为超大规模集群设计的。

第五，支持运行在 flannel 之上：flannel 负责网络连通性，Cilium 负责安全策略。如果你的集群现在是 flannel 模式，迁移到 Cilium 会比较方便。

最后，非常活跃的社区。Cilium 背后是一家公司在支持，一部分核心开发者来自内核社区 ，而且同时也是 eBPF 的开发者。

- Cons

首先是内核版本要求比较高，至少 4.8+，最好 4.14+，相信很多公司的内核版本是没有这么高的。

第二，方案比较新，还没有哪家比较有名或有说服力的大厂在较大规模的生产环境部署了这种方案，因此不知道里面有没有大坑。

第三，如果要对代码有把控(稍大规模的公司应该都有这种要求)，而不仅仅是做一个用户 ，那对内核有一定的要求，例如要熟悉协议栈、包的收发路径、内核协议栈数据结构、 不错的 C 语言功底（BPF 程序是 C 写的）等等，开发和运维成本比基于 iptables 的方案 （例如 Calico）要高。

总体来说，Cilium/eBPF 是近几年出现的最令人激动的项目之一，而且还在快速发展之中。我推荐大家有机会都上手玩一玩，发现其中的乐趣。

References

OpenStack Doc: Networking Concepts
Cisco Data Center Spine-and-Leaf Architecture: Design Overview
ovs-vswitchd: Fix high cpu utilization when acquire idle lock fails
openvswitch port mirroring only mirrors egress traffic
Lyft CNI plugin
Netflix: run container at scale
Cilium Project
Cilium Cheat Sheet
Cilium Code Walk Through: CNI Create Network
Amazon EKS - Managed Kubernetes Service
Cilium: API Aware Networking & Network Security for Microservices using BPF & XDP

携程万台规模容器云平台运维管理实践

【作者简介】周昕毅，携程系统研发部云平台高级研发经理。现负责携程容器云平台运维，Cloud Storage 及 Cloud Network 基础设施研发及运维。

本文来自于周昕毅在 GOPS 全球运维大会上的分享，由高效运维公众号整理，略有修改

前言

本文将分享携程在私有云平台管理实践过程中踩过的坑和遇到的问题，包含：

第一部分，携程容器云概览
第二部分，容器云管理实践
第三部分，云平台运维管理发展方向展望

一、携程容器云概览

携程使用混合云架构，自建数据中心结合公有云实现弹性资源管理。机票、酒店、商旅、度假等业务线，数以千记的应用运行在携程容器云上。业务研发基于容器云可以实现快速功能迭代，按需扩容缩容，整个平台每周变更次数超过 1 万次。

携程容器云技术选型主要分为三个阶段：

第一个阶段，携程从 2013 年开始实施 OpenStack ，2014 年所有数据中心都具有了 OpenStack Provision 的能力，因此我们对 OpenStack 特别熟悉。容器云平台第一个版本也是尝试了基于 OpenStack 技术来实施。

2015 年当时容器在生产环境上运行的还较少，各大厂都在试水的阶段，基于性能和稳定性的考虑，大家不希望对基础设施做太大变革。

为了尽快迈出容器化这一步，我们采用了相对折中的方式，通过 OpenStack 部署一种类似于轻量级虚拟机的方式来跑容器。当然这也有不好的地方，就是调度方式不灵活，因为容器时代跟虚拟机的调度和编排的需求不太一样，虚拟机部署出来它的生命周期就是上线周期到下线，容器需要更强的调度迁移的能力。

第二个阶段，2016 年。当时我们觉得 OpenStack 管理容器的方式很难支撑下去了，当时业界最火的调度技术是 Mesos ，我们调研了 Mesos 并在它的基础上自研了调度 framework 。

第三个阶段，从 2017 年到 2018 年两年时间，我们发现 Mesos 的社区越来越不活跃了，遇到问题也是自己解决，无法依靠社区的力量。

调研之后我们决定全面投向 Kubernetes ，同时给用户灌输概念：不再是单独申请，你申请的是服务，容器云给你提供服务，这个 PAAS 服务包括应用、缓存、数据库服务、日志与监控服务等。

我们容器云的中间一层是 CDOS (Ctrip DataCenter Operation System) ，容器云把所有数据中心管控起来，像操作系统一样，把底层的计算资源、网络资源、存储资源以容器云 PAAS 的方式提供给客户使用，在 CDOS 上层有各大系统和业务框架的系统，包括 SOA，相当于携程容器云在携程里处于数据中心的统一交付接口。

当然我们自身也集成了调度编排、日志监控、存储资源、镜像管理，用统一的方式交付给研发团队比较一致的交付体验。

IAAS 的理念是把基础设施以服务的方式提供出来，这样对业务的体验还是不够好；现在全面转向 PAAS 的理念 ，用户不再需要去申请一个机器或者申请 IP，直接给用户提供服务，你不需要关注服务跑在物理机上还是容器上，它后台服务的IP地址是什么都不需要去关注。

容器云还有一个比较大的好处，我们具备了弹性计算的能力和应用容量预估的技术基础，最终可以实现提升资源利用率的目标。

在几年前业务团队申请虚拟机的时候，我们一旦给了用户虚拟机就很难动它了，虚拟机热迁移是需要很高成本的，而且用户究竟在虚拟机里装了什么软件，做了什么特殊的配置我们是很难控制的。

容器云的时代该问题得到很大改善，每一次的容器发布都是做重新的调度，每一次可以给全新的容器，而且确保跟上一个版本完全一样，因为用户没有权限做修改。

二、携程容器云运维实践

2.1 面对的挑战

容器云面对各种各样的挑战，从基础设施角度来讲，我们需要解决计算、网络、存储三个方面的问题。

首先，网络层面，IP 的数量会有指数级的上升，虚拟机的时代可以单机多应用，部分应用可以部署在同一个虚拟机上；单容器单应用架构会导致容器的数量比较多，IP 数量也会多十倍，这样要上一套 SDN 的管理方案来进行多个 VPC 的隔离，传统虚拟机的管理方式已经不适用了。

第二个挑战是计算资源隔离相关的问题。每个业务的峰值是不一样的，如何保证每个业务容器 CPU 需求和网络流量突发过来的时候，它不影响其他的应用？这也是非常大的挑战。

由于容器本身的特性，假设你用程序获取 Mem/CPU/Disk 的值，可能获取到宿主机的值。还有比较麻烦的 DefunctProcess 问题，一旦容器中的某个进程变为 Defunct Proccess，只能通过重启宿主机的方式来结束它、此时同宿主机其他容器也需要被重启。

还有如何平衡好 CPU Throttle Time 也是一大挑战。 Throttle Time 出现的时候也就是代表某一个容器没有办法申请到 CPU period、在等待状态。因为容器的特性，同宿主机部署应用密度上升，导致系统调用对内核增加额外系统调用次数，在 3.10 内核上遇到过触发内核死锁的 BUG。Docker 版本升级非常快，从 1.6 到 1.7、1.8 到后面 17.0、18.0，我们是否跟着升级每一个版本？是单独维护分支，还是紧跟社区的步伐？

2.2 基础运维

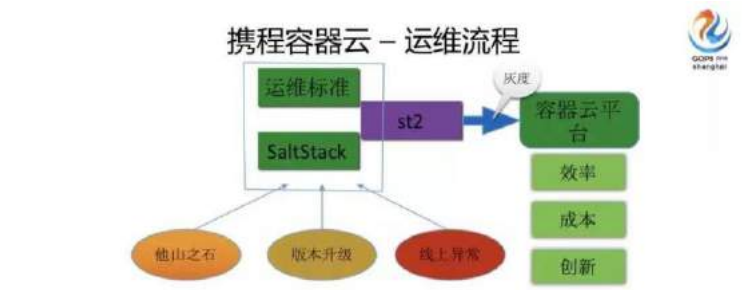
2.3 运维工具

从运维工具角度来说，配置管理工具我们用了 SaltStack 和 Rundeck 两个比较常用的配置工具，监控告警携程运维团队有一个自研的 Ctrip-Hickwall 工具，开源的是 Prometheus ，稍微做一些配置和改造就可以监控整个云平台各种指标，包括系统层面、调度成功率、资源可靠性。

日志系统也是用的比较主流的技术栈，用 ELK、TIGK 和 ElasticBeats ，监控告警和日志系统采集的结果会放在中心的数据库，为 AIOPS 提供数据基准。

同时我们基于 StackStorm 开发了系列工具，进行 ChatOps 工程实践，也达到了比较好的效果。

2.4 运维流程



刚才提到了工作流的工具 StackStorm ，它跟传统化运工具有些区别。我们开发的聊天机器人也可以和 StackStorm 工具做集成，提高排障的效率。

上图是我们的一个聊天机器人的例子，普罗米修斯对接的机器人，这个机器人对接都会为云平台各个服务做采集，发现某个服务异常，会自动发送到 IRC Chat。工程师可以点击链接跳转相关环境的监控页上去，看一下当前到底发生什么样的事情，判断故障是什么原因导致的，回复一个消息给机器人。

故障从发生到被响应应处理是有纪录的，这样就可以很全面的做一个知识库，新加入团队的工程师可以翻聊天室看一看历史上出现了哪个故障，是否有预案应对？

这一块每个业务容器发布失败三次，基本上可以认为是云平台本身的问题导致的，那就需要工程师看一下错误日志，这种错误是很难处理的，但是我们可以让工程师很快发现问题，因为我们把所有相关的错误日志汇总在一个页面，通过点击连接，工程师可以快速把故障建立一个关联。

这里总结一下我个人认为 ChatOps 的好处吧，对于老板来说每个团队都是有困难的，每个团队都是希望招人的，但是往往很难。运维工程师也不愿意做去重复的 routine task ，我会跟他们说你们去招聘一个机器人，工程师觉得这样说也不错，我们的 ChatOps 的工具也提供了这样的可能，对团队协作方面也有很大的提升。

2.5 关注变化

我们容器云运维最关注平台发生的变化，因为平台的变化往往都是故障的先兆，对于异常的事情需要让工程师做深度的挖掘，往往是暂时没有出现影响业务的异常，在深度挖掘之后会是非常大的坑，在不久的将来就会让业务受到比较大的影响。

我们鼓励工程师花时间做深入挖掘而不是满足正常运行就可以了，灰度运行一定要落实的，这个时候也需要提供工具让工程师落地灰度变更，通过流程和工具来做保障。

我们会组织一些会议回顾近期踩过的坑，有时候需要把节奏慢下来，在今年年底之前需要把所有宿主机干掉，在这样的工作压力下，也不能让节奏变的太快，也需要不同的回顾。实现关注变化的技术手段依赖于各种监控系统、巡检工具以及 CI/CD 的工具链。

上图是我们镜像的监控，我们认为一个镜像服务维护的水准是一个容器云运维能力非常重要的指标，因为镜像对于容器来说特别关键。我们会从每个数据中心建立一个单独的 Harbor 集群，让用户从第二个备份的数据中心拉容器，第二个数据中心出问题也会有保障，它的带宽变化、相应时间变化都会有分析。

容器云的网络也是我们特别关注的一个点，这个指标会去看每一个端口创建的时间，包括 IP 资源的情况都会做单独的监控。

2.6 关注趋势

前面讲的是关注变化，关注变化算是比较基础的一个工作，因为做运维的工程师都会关注我们是否有告警，平台是否有变化，而趋势往往会被工程师所忽视。

我们关注趋势可以设定长期目标，让运维工作比较有计划性，运维工作有计划性是非常重要的，怕就怕的是运维工程师每天都在处理突发的工作，没有时间对你管理的平台做前瞻性的规划，把事情做在前面，不要把压力留给自己。所以需要建立技术储备，让工程师有一定的前瞻性，必须要在异常真正发生之前能够解决潜在问题。

云平台的组件实在太多了，包括存储、计算每一个环节都会出现问题，监控做的太细的话，让你淹没在日常事件里，核心的指标会被忽视，用户关注的是整个服务的交付的速度和服务整体的可控性。我们目前用的是 TIGK 、StackStorm 、SaltStack 和 ChatOps 。

上图是我们做的机器负载变化趋势的看板，可以快速帮助我们发现某一个集群的利用率，这个集群 CPU 还是比较富裕的，它的内存也只有 71.47%。我们会做一些分析，看看是不是给他分配的容器太多了。像左下角这些宿主机要主动的维护，持续增加的话机器某一天就会坏掉了，造成非常大的影响。

这是应用维度的变化趋势，像这个应用在线容器数有三百个，但是 CPU 的表现是很稳定的，CPU 突然出现 50%左右的增长，我们也会告警他是否做一些分析或者扩容。

2.7 容量管理

下面介绍一下我们的容量管理，容量管理往往是老板比较关心的，因为涉及到花钱。每个季度到底投多少钱买宿主机？动态调度的能力有没有？应用有波峰和波谷的，尽量把波峰的应用挫开，我们需要对它进行资源使用情况的预判，这样才能实现弹性计算。

为了实现容量管理我们主要借助了监控系统、 Hadoop 平台以及自己运维的监控工具，最终通过 PAAS 平台实现容器弹性的调度。最终目标是把资源合理利用出去，同时又保证一定的稳定性。

这是我们容量的整体情况，会发现它的内存基本上用的非常满，这个集群它的 CPU 资源也是用的比较满，我们会从一些维度去做整体的分析和个别的分析。

三、总结与展望

前面基本上就是运维相关的事情，下面简单说一下我个人的思考。之前做 OpenStack 私有云的管理，做完之后我们要把运营工具做成产品化，不能用工具的思维做事情，这样不能很好的解决用户的问题。

所以我们现在也是在尝试做一些日志产品和监控产品，在云原生的 DevOps 工作方式。我们运维人还是要以用户至上的，整体出发点保证平台稳定、持续、高效运行。还有一个总结是对 ChatBot 与事件的整合有比较好的效果，一方面让事件可追溯，另外一方面让工程师有更好的热情。

展望团队工作的话，接下来会有混合云运维的实践，携程这些采购公有云的产品，阿里云、AWS 还没有做很好的整合，下一步把混合云管理起来，真正做到云原生。

另一方面业界都在转 Kubernetes ，但还要进行思考，在 Kuberentes 时代下一步要做什么。做容器云的价值就是实现了弹性计算，而我们要在弹性计算这条路上做更多的事情，真正体

现出容器的价值。

日部署 6000 次，携程持续交付与构建平台实践

【作者简介】周光明，携程高级技术经理，目前负责携程 CI/CD 系统，致力于通过技术手段提高公司研发质量与效率， 对 Docker，K8s，Gitlab，Jenkins 等 DevOps 技术有浓厚的兴趣，Ruby 语言狂热爱好者。

本文来自周光明在 DevOps 国际峰会上的分享

一、携程持续交付

我们目前有 8000 多个应用，研发人员 3000 多位，每天在各个环境上部署的次数有 6000 多次，持续交付对于我们来说是一个非常重要的能力。

持续交付的意义，首先是效率的提升。部署是一个很麻烦的事情，如果是有多个环境需要部署，部署的难度也会直线上升。这时候如果有一个工具去做这样的事情，研发人员就可以将更多的精力投入到研发它的功能上面，让产品的迭代更加迅速。

第二是质量保障，我们在持续交付的过程中穿插了一些代码扫描、单测或者集成测试的过程，可以让整个产品的质量在交付过程中得到很好的保障，也可以让我们在交付的时候更加有信心。

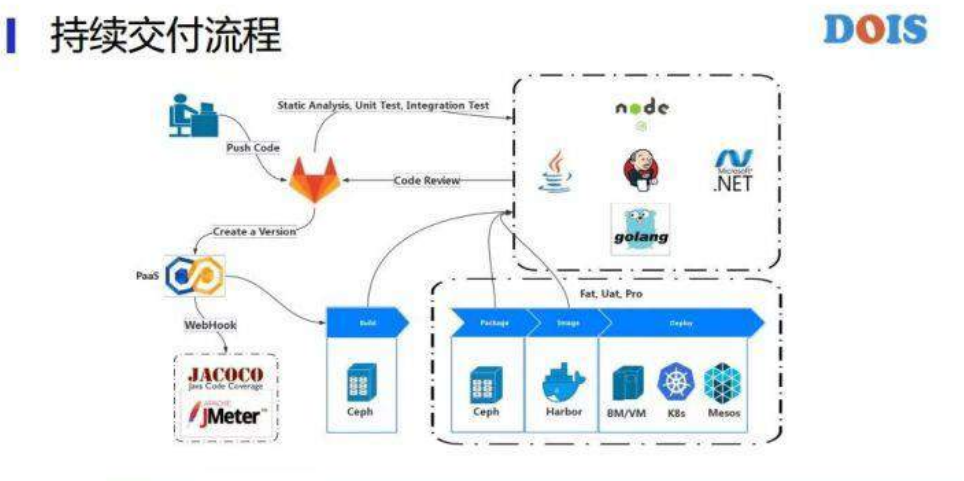
第三是安全可靠，如果没有机器就要人工跑上去进行部署，会对线上系统增加很多误操作的隐患。

第四是团队协作，传统交付模型从产品讨论到上线需要经过很长时间，有可能出现一个现象，在开发阶段的时候开发人员在闷头写代码，测试人员没有什么事情做，到了测试阶段这个现象又会反过来。如果我们采用小步快走的方式，可以让各个团队之间的协作更加紧密和紧凑。

最后是流程更加透明，因为我们使用的是统一的规范、统一的工具，在交付过程当中的每个细节都可以被暴露出来。不管谁多写一个 bug 或者少写一个单测都会被系统记录下来。

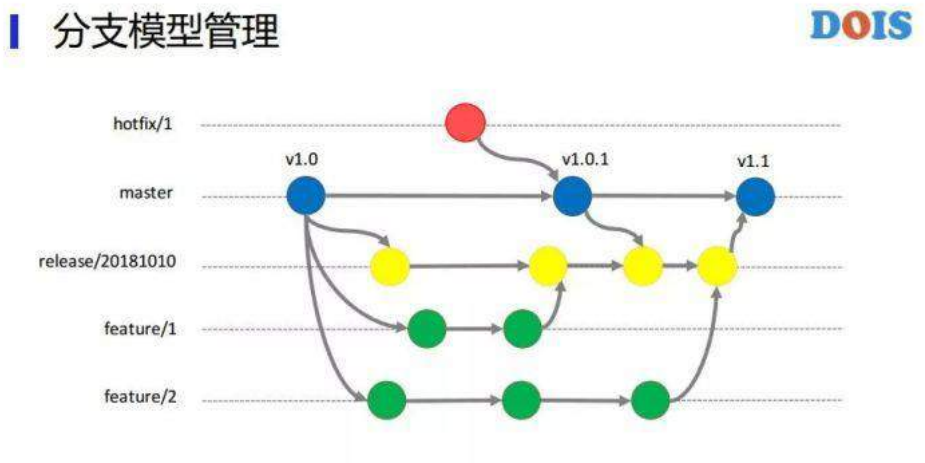
下图是目前我们简单的交付流程，首先是研发人员 Push 代码，扫描单测集成测试，再将结果反馈，之后创建一个版本，版本是什么概念待会再说。

创建版本之后进行打包，再部署到测试环境，部署成功之后我们会通知周边的自动化测试平台或者性能平台，项目测试人员、QA 根据测试结果进行审批工作，就可以将项目部署到下一个测试环境或者生产环境当中。



对于研发阶段来说，我们目前主要推崇的分支管理模型是 Master 分支和 Feature 分支，多个 Feature 分支可以同时进行功能的开发，并且可以被临时合并到一个分支。之所以这样做，是因为这样可以让代码的冲突在合并之前就暴露出来并提前解决。

如果这个时候线上有一些紧密的 bug 要修复，也可以通过 Hotfix 分支提交代码，Hotfix 分支被 Merge 回 Master 之后也会被 Merge 到上面提到的临时分支中。



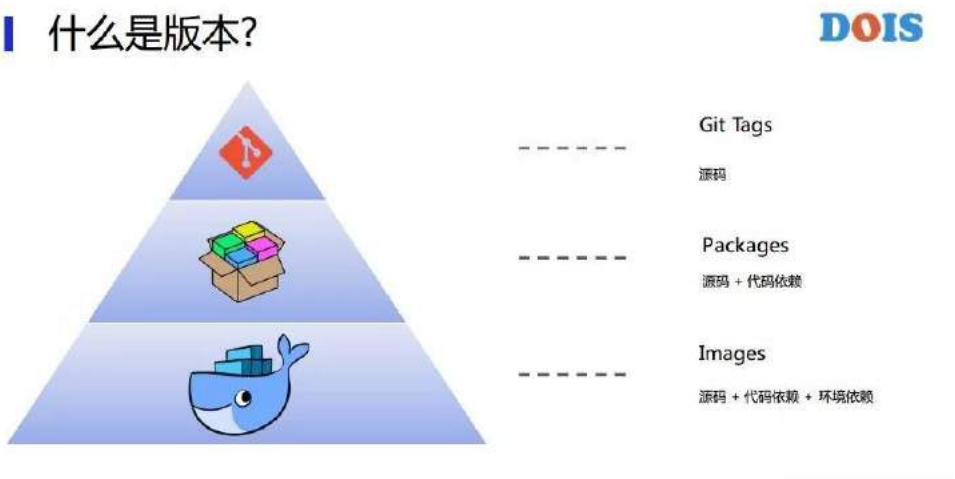
接下来解释一下刚才提到的版本概念，我们知道很多开源软件是使用 Git Tags 作为开源版本的，因为一个 Git Tag 可以快速找到仓库某一时刻的内容。如果一些项目比较复杂，可能会有一些其它代码的依赖。因此我们可以将源码构建打包成一个版本，打包的东西就会比使用 Git Tag 更加准确。

因为有些代码依赖，不是一个特定版本号，可能是一个范围。上个月打包出来的结果和今天打包出来的结果就会不一致，那么部署就增加很多不稳定的因素。

第三点是在一些比较特殊的项目里面，除了语言依赖之外，还会有环境依赖。

在容器没有出现之前，我们将环境依赖的过程写到项目原码的脚本，通过部署运行多个程序安装那些依赖。但是有了容器之后，我们就可以将环境的依赖也作为版本的因素，容器就可以很好地帮我们解决这个问题。

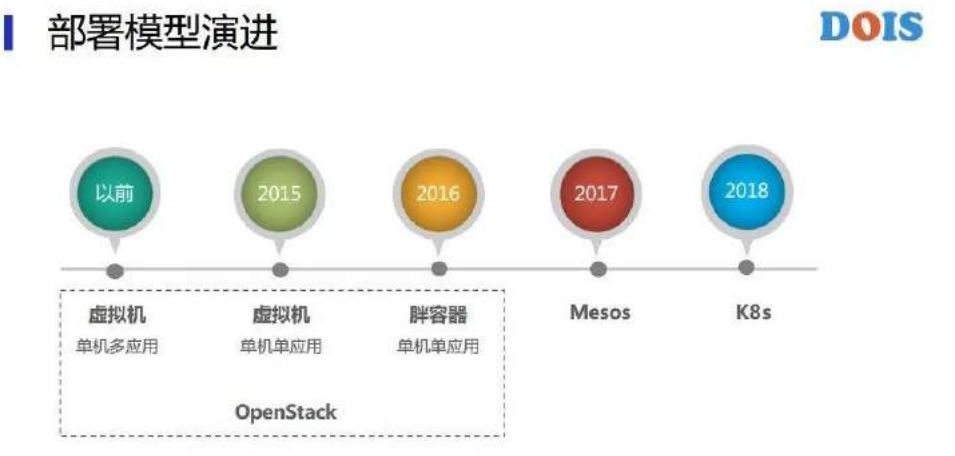
因此一个明确版本的概念，对于交付来说也是相当重要的。



下面介绍一下我们这些年的部署模型的演进。

2015 年之前我们使用虚拟机做单机多应用的部署。对于运维来说成本相当高，2015 年我们重新做了一个发布系统，也是主推单机单应用的部署模型。2016 年开始研究容器，但是将容器作为一个虚拟机的方式，我们叫它“胖容器”，以这样的方式部署应用。

整个部署过程，总体来说还是有一些复杂的。我将几个比较关键的概念整理出来。



首先是 Group，一组暴露同一服务的集合，对于单机单应用，我们可以简单理解是一组机器，Group 也是我们部署的基本单元。

第二是拉入拉出，Group 中中的某个成员是否接受流量和请求，流量可能来自 SLB 或者消

息系统推送的消息。

第三是堡垒机，是指生产环境 Group 中第一台被发布验证的机器，有点像金丝雀部署模型中金丝雀的角色。

第四是点火，是指应用初始化、预热、加载数据等过程，我们认为点火成功才是应用部署成功的一个最终状态。

第五是分批，我们将同一个 Group 分成多个批次进行滚动部署，减少线上变更对于线上的影响。

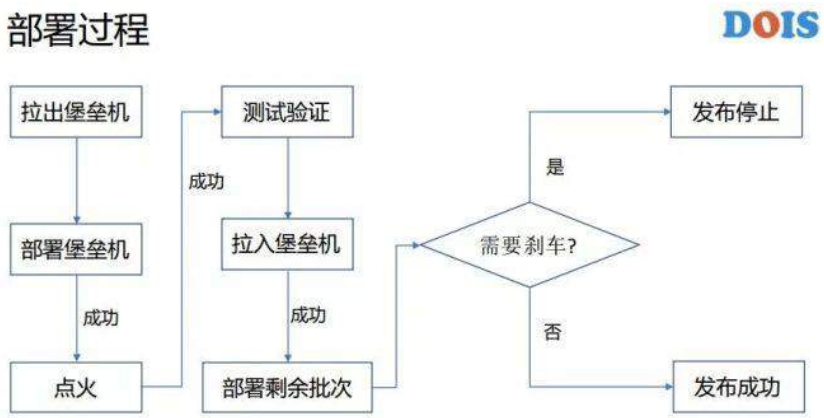
第六是降级，刚才也提到降级的事情，比如我们的发布需要对应用进行拉入拉出，如果这个时候 SLB 出现了故障，并且有应用需要紧急发布，我们可以通过降级的方式忽略拉入拉出，虽然会丢失一些线上流量，但是可以保证应用被成功的部署到生产上，因此也减少了线上的损失。

第七是刹车，刹车是如果线上的部署失败的机器大于一个比例，我们都会停止部署行为，人工排查到底是什么原因，是需要回滚还是修改 Bug 之后打另外一个版本进行修复。

第八是回滚，回滚的概念就不用多说了，我们需要稳定的符合预期的回滚逻辑。

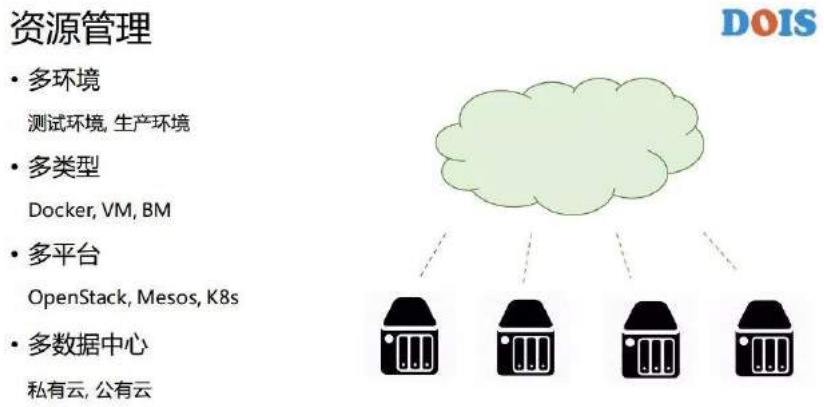
部署过程，首先是拉出堡垒机，部署堡垒机成功之后需要点火，进行测试验证后拉入堡垒机，堡垒机会作为一台正常机器进行工作。

这些都没问题之后我们再将剩余批次进行滚动部署，滚动部署的时候发现部署失败的机器比较多就需要进行刹车判断。



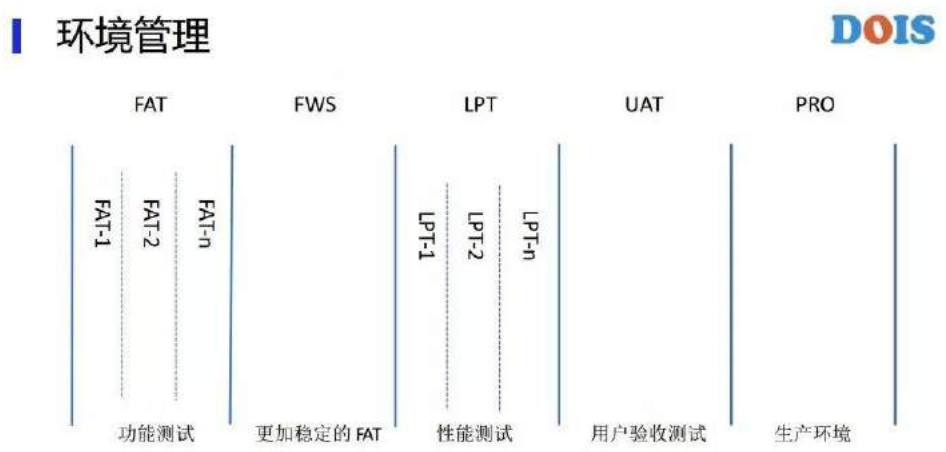
目前我们的 PaaS 平台上支持了测试和生产多个环境的资源管理，这些资源当中既有容器，也有虚拟机，甚至还有物理机的管理，因此我们的后端需要对接 OpenStack、Mesos 等管理平台。

目前我们可以将资源放在私有云的多个数据中心，也可以将资源放在像 AWS 公有云之上。



下面说一下环境管理，对于功能测试我们有一个 FAT 的环境，FAT 又分成多个 FAT 子环境，可以满足用户同时进行多个功能测试的需求。在 FAT 之上有一个 FWS 环境，它是一个更加稳定的 FAT 环境。

对于性能测试也是有多套性能测试环境。FAT 环境部署成功之后，需要 QA 人员的测试验收，才可以将应用发布到 UAT 环境，UAT 是一个相对更加接近生产的测试环境。最后是生产环境。



二、统一构建平台设计

我们可以看到刚才的流程图上很大一部分工作是通过统一构建平台实现的，接下来介绍下统一构建平台。相信很多同学是 Jenkins 用户或者爱好者，先说说 Jenkins。

首先 Jenkins 非常的方便，一个 War 包就可以轻松搞定部署这件事情。Jenkins 已经发展了很多年，非常的成熟稳定，插件也非常丰富，基本上满足各种各样的需求。当然这些来自社区活跃人员，强大的 Pipeline 可以将配置转化成代码，也是大大增强了我们的生产力。

但 Jenkins 也不是完美的，也有一些问题，其中就有单点故障和单机性能的问题。我们是怎么看待和解决这两个问题的？

先是单点故障，很多团队都是采用一主一备的 Jenkins 模式，出现故障的时候需要以切换的方式将故障转移，稍微成熟一点的团队会用 Keepalived+Virtual IP 保证 HA。还有可以将 Jenkins 打包放在 Mesos 或 K8S 上面，也可以购买 CloudBees 服务，比较省心一点。

解决了单点故障的问题，Jenkins Master 的上线总归是有限的，随着业务的增长每天的数量越来越成为负担。官方提供了几个维度拆分 Jenkins Master 的方式，分别是从环境、组织结构、产品线、插件可制定性、人员访问权限控制、出现故障时的影响等几个方面，分析了它的利弊。当然每个团队各自的情况不一样，需要根据各自的情况作出决策。

Master 拆分策略

DOIS

	By environment	By org chart	By product lines
Pros	Can tailor plugins on masters to be specific to that environment's needs Can easily restrict access to an environment to only users who will be using that environment	Can tailor plugins on masters to be specific to that team's needs Can easily restrict access to a division's projects to only users who are within that division	Entire flows can be visualized because all steps are on one master Reduces the impact of one master's downtime on only affects a small subset of products
Cons	Reduces ability to create pipelines No way to visualize the complete flow across masters Outage of a master will block flow of all products	Reduces ability to create cross-division pipelines No way to visualize the complete flow across masters Outage of a master will block flow of all products	A strategy for restricting permissions must be devised to keep all users from having access to all items on a master.

来源 https://jenkins.io/doc/book/architecting-for-scale/

拆分了之后，要预估一下 Jenkins Master 的单台机器的承载能力，这边也提供一个比较有意思的公式。

预估 Job, Master 和 Executors 的数量

DOIS

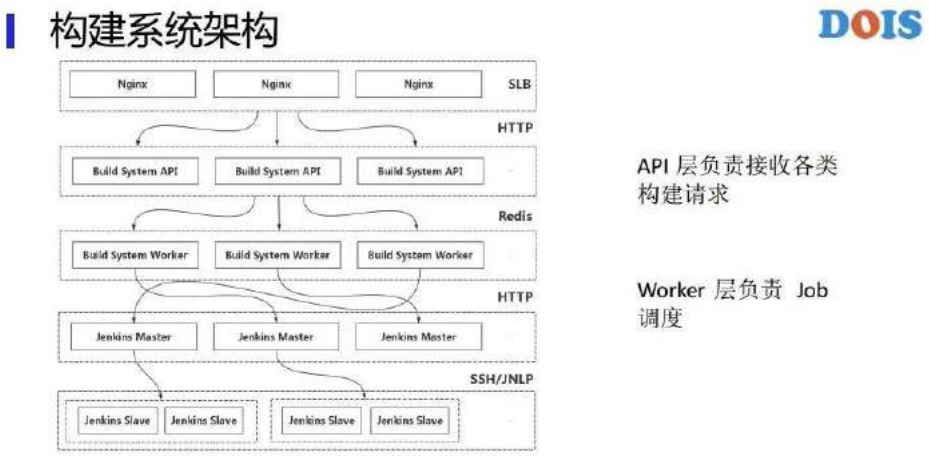
- Number of Jobs = Number of Developers * 3.333
- Number of Masters = Number of Jobs/500
- Number of Executors = Number of Jobs * 0.03

来源 https://jenkins.io/doc/book/architecting-for-scale/

根据研发人员的数量预估 Job、Master 和 executors 的数量，根据这个公式大概推导有多少个 Job 和 Master。我们每天大概是 20000 次构建数量，包括编译打包镜像等任务，管理了 40000 多个 Jobs ，这些 Jobs 跑在几十个 Jenkins Master 上。

我们选择自己做一个平台满足大量的运维工作，自己动手丰衣足食。

先看一下构建系统的整体架构，也是一个比较简单、比较传统的架构模型，在上层封装了一层 API 层，负责各种类型的构建请求，在 Worker 层，将每种构建类型调度到不同的 Jenkins Master 上。调度到 Jenkins Master 之后，就是 Jenkins Master 发挥自己能力的时候了。



接下来看下 Worker 层处理了哪些事情。有些同学可能会疑惑，为什么我们有这么多 Jobs？最早的时候我们不是按照这样的方式，是按照每种类型一个 Job，这样的好处是可以维护比较少的 Job。但是这样也有一些缺点，比如说我要更新 Job 配置的时候，影响范围太大，可能几千个应用都是依赖一个 Job。

还有就是大家都共用一个 Job，构建现场的保留也会成问题。当前面一个构建失败以后，下一个构建就会把前面的构建 Workspace 冲掉。此外保留了 Workspace 之后，也可以加快代码下载的速度。

当一个 Job 创建的时候或者每次构建任务进来的时候，我们都会对比当前 Job 的配置是不是最新的可用的，如果不是就将它更新。如果目前线上没有这个 Job，我们就会根据配置模版创建一个新的 Job，有了这个机制就可以将 Jenkins Master 作为一个没有状态的服务来看待。

接下来我们参考了 Labeling 模型，可以根据标签匹配找到满足条件的 Master。也可以把 Job 与 Master 标签进行匹配。做到这些之后下面的工作就会比较容易，我们在系统中同时注册了多个 Master，势必有多台 Master 满足构建条件。

此外我们还在 Master 上配置了容量配比，比如说新建 Job 时按照 Master 容量可以承接多少数量。最后是故障转移，当构建任务进来的时候看之前用过的 Master 是不是健康的，如果健康会优先把它调度到这台上面，如果不健康会选择另外条件满足的 Master。我们也会做 Master 的检测，如果某台 Master 不健康会拉出整个 Master 集群。

我们在多个维度做了监控和告警，第一个维度是操作系统层面的，也就是一些常规的指标，像 CPU 之类的。

第二个是应用系统层面的，包括 API 层的可用性、Worker 可用性、Jenkins 可用性等等。

第三个是业务逻辑层面的，主要检测的是比如说每一个构建队列是否堵塞，系统容量是否达到瓶颈，因为我们对每台 Master 都做了容量预估，希望当有大面积的构建请求进来的时候，可以提早知道进行扩容。

除此之外还有 Pipeline 关键 Step 是否超时，进行容器调度的时候，是不是创建时间比我们预期的要长等，接下来会细讲如何进行容器调度。

这是构建系统的简单界面，首页包含了目前各种状态的构建数量，还有一些简单的统计。比如构建系统中所有 Job 的列表情况，包括它之前构建了多少次、它是什么类型的。以及构建的任务情况，当前 Jenkins Master 线上集群的情况，包括支持的类型情况、监控指标等等。

三、Jenkins on K8s 实践

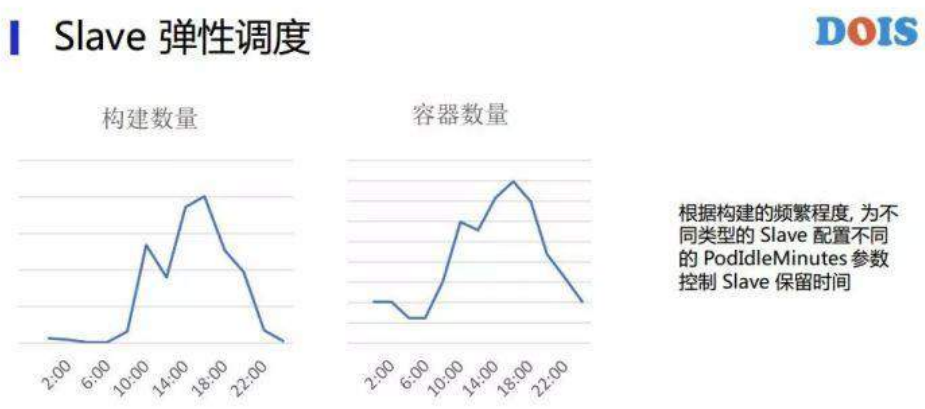
接下来是我们如何使用 K8S 进行 Jenkins 管理。

首先是 Jenkins 集成的演进，跟我们刚才看到的应用集成演进是类似的，但是时间上面稍微比他们快一些，因为在公司级别技术演进的时候经常会使用 Jenkins 作为一个试验田，因为它也比较合适。

接下来主要讲以下两个方面，第一是 Slave 弹性调度，第二是 Workspace 的问题，我们看一下为什么存在这两个问题以及如何处理好。

下面是单日构建数量以及容量数量趋势图。每一种类型根据调用频繁程度和特性，配置出合适的 Podidle Minutes 参数，控制 Slave 保留时间。

可以看到构建数量趋势明显比容器数量趋势缓和一些。我们的容器没有实时的创建和销毁，这样既满足我们对弹性调度的要求，也不会让整个系统的性能受到太大的影响。



既然实现了弹性调度，对于每个 Slave 创建的时间我们是特别关心的，因为它不像静态 Slave 可以有请求进来就可以直接拿来用。但是我们发现采用弹性调度的方式之后，Slave 的创建逻辑并不总是符合预期。

举个例子，当在空闲的时候，Jenkins 可以创建 Slave 并且正常执行构建任务，但是隔了几秒钟又有一个新的任务进来了就需要等一段时间。不知道有多少同学发现过这个问题？

我们首先梳理了一遍调度的逻辑，通过改变上面的几个参数，是可以达到目的的。接下来介绍一下几个参数的逻辑，有些人可能是对这个逻辑比较清楚的。

首先是 initialDelay，它是动态调度等待连接静态 Slave 的时间参数，在我们集群的 Jenkins Master 是没有任何静态 Slave 的，所以我们将这个参数设置成 0。

第二个参数是 Decay，它是 Jenkins 负载统计公式中指数移动平均值中的平滑指数，我刚才说的现象是因为 Jenkins 在内部维护多个负载情况的序列，这些序列的数据有等待队列数量、各种状态的数量等等。它们的值是通过 EMA 公式计算出来的，比如说 history0 是前一时刻的值，等待队列数量就是 0。

这个时候来了一个数量就是 1，假设 decay 是 0.2，那么计算出来的值负载值就是 0.8。我们可以观察到 decay 的值越大，当前负载越接近实际值。因为我们不希望 Jenkins 的保守创建逻辑增加整体的构建时间，因此需要让负载统计中的值更加接近于当前的实际情况。但是这个值也不能太小，如果太小 Jenkins 就过于敏感，有一个请求过来就帮你创建，还没创建好下一次还会创建一个，这样浪费很多资源。

第三个公式是 Jenkins 判断是否创建 Slave 的不等式，不等式右边为什么是 1-m 呢？m 是作为一个参数来用的，如果根据 EMA 的值计算，它是永远不会等于 1 的，只是会无限接近于 1，因此我们需要一个偏移量控制它是不是应该创建 Slave。

m 的公式也和上面三个参数有关的，最后的参数是 totalSnapshot，这是当前可以用的 Excutor 数量，经过调整这三个参数可以很好的工作。

Slave Provision 时间优化

DOIS

```
-Dhudson.slaves.NodeProvisioner.initialDelay=0    //Default 100
-Dhudson.model.LoadStatistics.decay=0.5           //Default 0.9
-Dhudson.slaves.NodeProvisioner.MARGIN=50         //Default 10
-Dhudson.slaves.NodeProvisioner.MARGIN0=0.85      //Default 0.5
-Dhudson.slaves.NodeProvisioner.MARGIN_DECAY=0.5 //Default 0.5
```

initialDelay: Jenkins 启动留给静态 Slave 连接的时间

decay: Jenkins 负载统计公式 “指数移动平均值(EMA)” 中的平滑指数:

$$history[0] * decay + newData * (1 - decay)$$

判断是否应该创建 Slave:

$$excessWorkload > 1 - m \ (excessWorkload = qlen - plannedCapacity - connectingCapacity$$

m 的计算公式:

$$MARGIN + (MARGIN0 - MARGIN) * Math.pow(MARGIN_{\downarrow}DECAY, totalSnapshot)$$

经过长时间的观察，我们发现这样一个调整对于创建参数是比较平稳的，也没有太大波动。创建一个 Slave 大概是 20 多秒时间，因为采用的调度方式不是立即创建和消毁，所以每天大概有几十个创建时间，相对于每天的构建数量是可以被接受的。

长时间运行之后，我们还会发现有个别的 Slave 的创建时间会超过 5 分钟，这是为什么呢？一开始我也不知道，所以我们又重新梳理了一下整个创建流程，也是找到了其中的原因。Jenkins 的调度逻辑是通过一个轮训逻辑做的，遍历 Labels。

Slave Provision 时间优化

DOIS

经常有 Slave 的创建时间超过 5 分钟!

2018/10/22 (周一) 1:34
build@ctrip.com
[BuildPortal] Schedule slave for k8s job: java-s[redacted] exceed 5 minutes --- production
收件人 [redacted]

Kubernetes slave 调度时间超过五分钟，编译终止

Build ID: 4126227

Job Name: java[redacted]onnector

Job Type: j[redacted]r

查看详情

如果在系统当中这个Labels是没有出现过的，它要创建一个新的Labels，它是不会更新Labels集合的。但是 Jenkins 每隔 5 分钟会更新一次 Labels 集合，最后我们在创建中 Label 时主动调用 reset 方法解决这个问题。

目前已经运行了差不多一年时间，后来刚刚看到那些问题再也没有出现过了。

接下来讲一下 Workspace 保留问题，什么是 Workspace 保留问题？Workspace 对于整个系统来说是非常重要的。首先用户排障需要现场，另外我们通过复用 Workspace 可以减少下载源码的次数。

但是一旦 Workspace 消毁之后，之前没有做任何处理，在这个 Slave 上面所有数据就跟着一起被消毁了。我们首先想到的方式是将上面的数据挂载到 Slave 上面，消毁也不会有影响，下一次重新被挂载进去。

但是这样只解决了一个问题，Slave 不会被删，但是通过 Jenkins Master 怎么看？我们目前的做法是让 Slave 与 Master 在同一个 Node 且共享同一 Workspace，通过 Master 查看 Workspace 的能力，查看上面运行的 Slave Workspace。

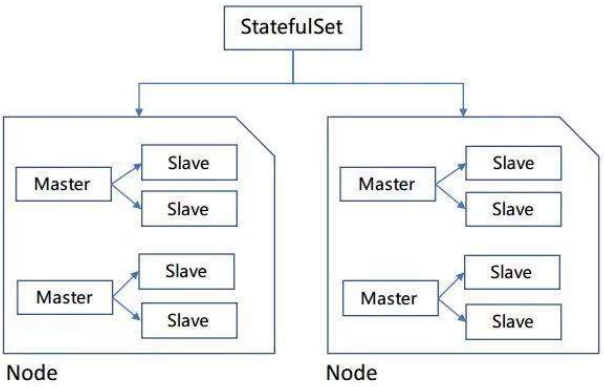
但是我们遇到一个问题，一个 Job 同一时间只能在一个 Master 上面运行，因为我不可能把一个目录同时给两个 Master，这样可能会产生无法预期的结果。

解决这个问题的方法是，通过上层来做一个调度，将之前 Jenkins Master 控制并发的这一阶段放在系统上面。因为 Job 已经拆分得很细了，因此对于单个 Job 的并发需求来说不算大。

接下来再介绍一下我们是如何通过 StatefulSet 管理 Jenkins Master 的，是不是可以通过 StatefulSet 维护 Jenkins Master 集群，因为我们希望更加自动化，所以解决了以下两个问题。

StatefulSet 管理 Master

DOIS



使用 StatefulSet 管理 Jenkins Master 集群降低维护成本

第一是 Job 会对之前成功运行的 Jenkins Master 有亲和性，它会默认跑到之前运行过的 Master，因此我们为了尽可能的使用之前的 Workspace，需要将 Master pod 尽可能固定在 Node 上面。

我们做了一个调度器，创建出来的 Master pod 后面有序号，可以很好的映射到每一台机器上面。

Sticky Scheduler

DOIS

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  namespace: build
  name: jenkins-master
  annotations:
    jenkins-master-0: svr01
    jenkins-master-1: svr01
    jenkins-master-2: svr01
    jenkins-master-3: svr02
    jenkins-master-4: svr02
    jenkins-master-5: svr02
spec:
  serviceName: jenkins-master
  replicas: 6
  selector:
    matchLabels:
      app: jenkins-master
```

让 Master Pod 固定分配到指定的 Node 上

第二是我们需要将上面的目录既挂载在 Master，又挂载到 Slave 上面，因此创建集群的时候需要事先将挂载目录准备好，所以我们也是开发了一个插件 CHostpath Volume Driver。除了这些之外，还将 Java 依赖的东西也是放在上面，每次创建 Slave 的时候也可以挂载到 Slave 上面，可以提高构建的性能。

CHostpath Volume Driver

DOIS

```
deploy@ubuntu:/var/lib/k8s/storage$ tree
.
└── chostpath
    └── build
        └── StatefulSet
            └── jenkin-master
                └── Pod
                    ├── jenkins-master-1
                    │   └── jenkins-home
                    ├── jenkins-master-2
                    │   └── jenkins-home
                    └── jenkins-master-3
                        └── jenkins-home
```

按照规则自动创建 Pod 的挂载目录

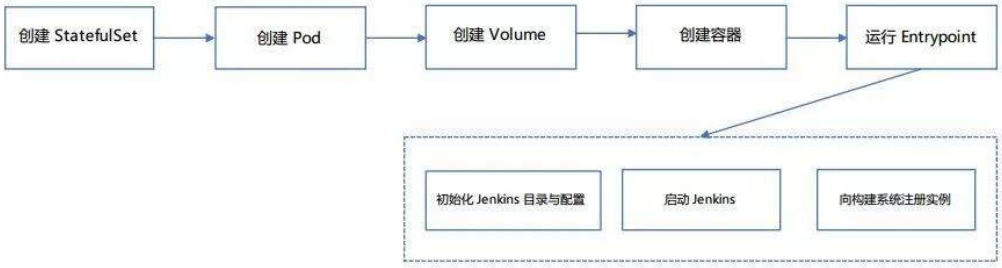
/var/lib/k8s/storage/chostpath/$(NAMESPACE)/$(RESOURCE_TYPE)/$(RESOURCE_NAME)/Pod/$(POD_NAME)/$(VOLUME_NAME)

有了这些之后，再看一下整个集群创建的流程，其实只要维护一个 StatefulSet 就可以了。首先是创建一个 StatefulSet，后面创建、更新或者扩容的时候都要创建一个 Pod，再创建出 Volume，再创建容器，运行 Entrypoint。因为刚才提到 Jenkins Job 的配置，将 Jenkins Master 的配置放在上面也是支持版本的控制。

下载下来到本地初始化目录，因为初始化之后有些配置需要根据当前机器的情况做修改，比如说 IP，最后启动 Jenkins，向构建系统注册实例，StatefulSet 更新完之后只需要在系统里面拉入就可以对外配比服务了，中间不需要做过多手工干预。

Jenkins Master 启动过程

DOIS



四、问题与改进

上面大概介绍了持续交付、构建平台、Jenkins on K8S 使用实践，接下来说一下问题与改进。

第一是多环境应用镜像问题，我们现在是一个环境有一个镜像，为什么会这样？因为在早些

年没有配置中心的时候，配置是写在代码里的，在发布的时候会根据每个环境将配置重新做修改，再打成包，再打成镜像。

当然这样就会造成比较多的问题，比如环境之间的差异，因为一个应用的发布最好以镜像作为版本，这样就会作为测试环境与其它测试环境不一致，也会降低发布效率。

第二是资源混部，我刚刚说的整个构建系统的调度，Master 是非常必要的，我们希望未来构建系统的机器在业务比较紧张的时候，可以使用其它的业务计算资源，在晚上构建空闲的时候可以腾出来给其他的业务跑 Job，这样也可以提升整个数据中心的资源利用率。

携程容器偶发性超时问题案例分析

【作者简介】李剑，携程系统研发部技术专家，负责 Redis 和 Mongodb 的容器化和服务化工作，喜欢深入分析系统疑难杂症。
周昕毅，携程系统研发部云平台高级研发经理。现负责携程容器云平台运维，Cloud Storage 及 Cloud Network 基础设施研发及运维。

上篇

前言

随着携程的应用大规模在生产上用容器部署，各种上规模的问题都慢慢浮现，其中比较难定位和解决的就是偶发性超时问题，下面将分析目前为止我们遇到的几种偶发性超时问题以及排查定位过程和解决方法，希望能给遇到同样问题的小伙伴们以启发。

一、问题描述

某一天接到用户报障说，Redis 集群有超时现象发生，比较频繁，而访问的 QPS 也比较低。紧接着，陆续有其他用户也报障 Redis 访问超时。在这些报障容器所在的宿主机里面，我们猛然发现有之前因为时钟漂移问题升级过内核到 4.14.26 的宿主机 ServerA，心里突然有一丝不详的预感。

二、初步分析

因为现代软件部署结构的复杂性以及网络的不可靠性，我们很难快速定位“connect timout”或“connectreset by peer”之类问题的根因。

历史经验告诉我们，一般比较大范围的超时问题要么和交换机路由器之类的网络设备有关，要么就是底层系统不稳定导致的故障。从报障的情况来看，4.10 和 4.14 的内核都有，而宿主机的机型也涉及到多个批次不同厂家，看上去没头绪，既然没什么好办法，那就抓个包看看吧。

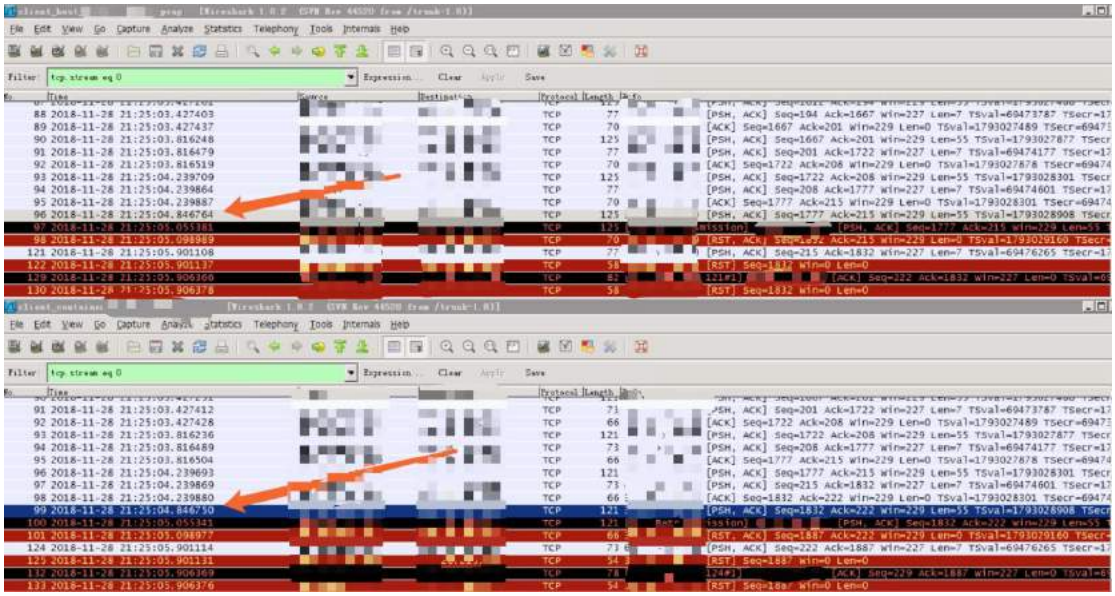


图 1

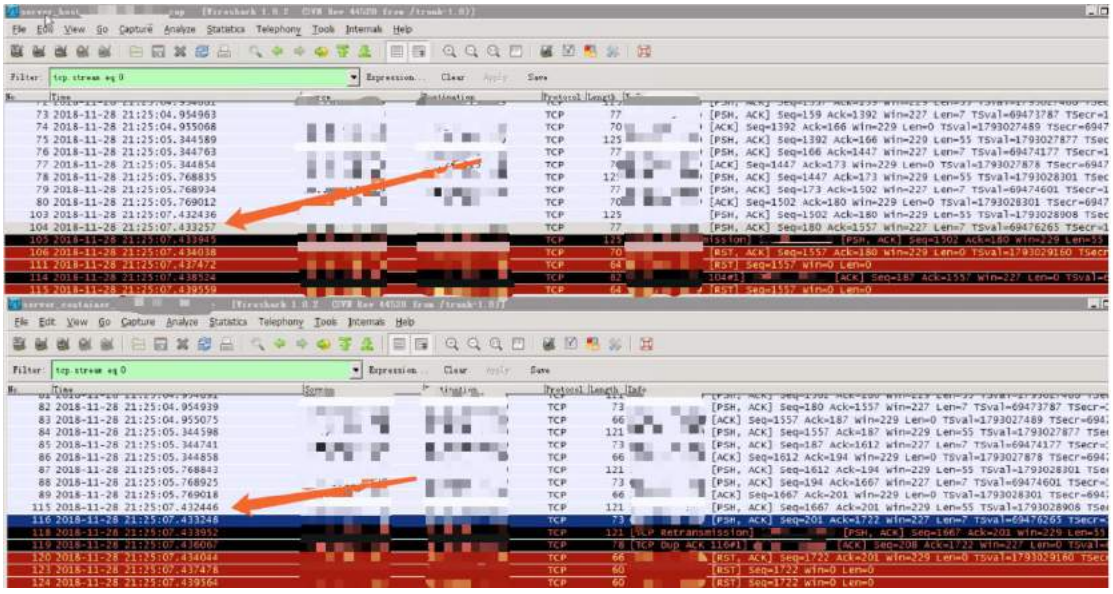


图 2

图 1 是 App 端容器和宿主机的抓包，图 2 是 Redis 端容器和宿主机的抓包。因为 APP 和 Redis 都部署在容器里面（图 3），所以一个完整请求的包是 B->A->C->D，而 Redis 的回包是 D->C->A->B。

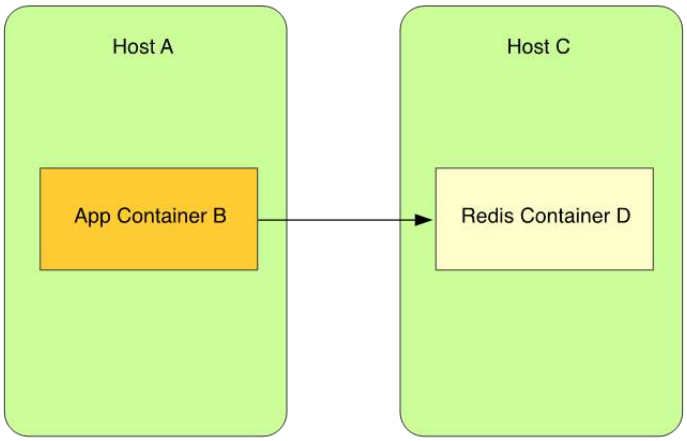


图 3

上面抓包的某一条请求如下：

1）B（图 1 第二个）的请求时间是 21:25:04.846750 #99
2）到达 A（图 1 第一个）的时间是 21:25:04.846764 #96
3）到达 C（图 2 第一个）的时间是 21:25:07.432436 #103
4）到达 D（图 2 第二个）的时间是 21:25:07.432446 #115

该请求从 D 回复如下：

1）D 的回复时间是 21:25:07.433248 #116
2）到达 C 的时间是 21:25:07.433257 #104
3）到达 A 点时间是 21:25:05:901108 #121
4）到达 B 的时间是 21:25:05:901114 #124

从这一条请求的访问链路我们可以发现，B 在 200ms 超时后等不到回包。在 21：25:05.055341 重传了该包#100，并且可以从 C 收到重传包的时间#105 看出，几乎与#103 的请求包同时到达，也就是说该第一次的请求包在网络上被延迟传输了。大致的示意如下图 4 所示：

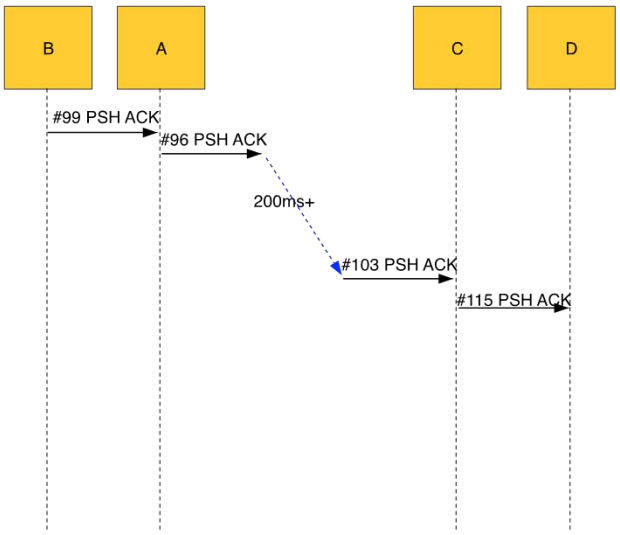


图 4

从抓包分析来看，宿主机上好像并没有什么问题，故障在网络上。而我们同时在两边宿主机，容器里以及连接宿主机的交换机抓包，就变成了下面图 5 所示，假设连接 A 的交换机为 E，也就是说 A->E 这段的网络有问题。

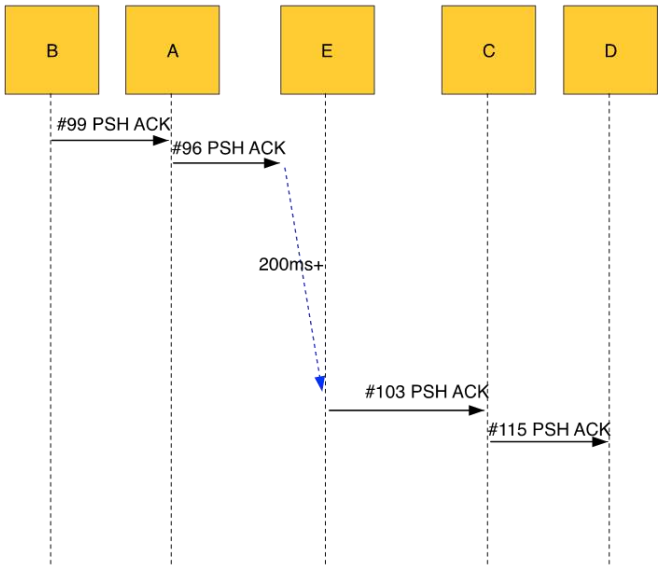


图 5

三、陷入僵局

尽管发现 A->E 这段有问题，排查却也就此陷入了僵局，因为影响的宿主机遍布多个 IDC，这也让我们排除了网络设备的问题。我们怀疑是否跟宿主机的某些 TCP 参数有关，比如 TSO/GSO，一番测试后发现开启关闭 TSO/GSO 和修改内核参数对解决问题无效，但同时我们也观察到，从相同 IDC 里任选一台宿主机 Ping 有问题的宿主机，百分之几的概率看到很高的响应值，如下图 6 所示：

图 6

同一个 IDC 内如此高的 Ping 响应延迟，很不正常。而这时 DBA 告诉我们，他们的某台物理机 ServerB 也有类似的的问题，Ping 延迟很大，SSH 上去后明显感觉到有卡顿，这无疑给我们解决问题带来了希望，但又更加迷惑：

1）延迟好像跟内核版本没有关系，3.10，4.10，4.14 的三个版本内核看上去都有这种问题。
2）延迟和容器无关，因为延迟都在宿主机上到连接宿主机的交换机上发现的。
3）ServerB 跟 ServerA 虽然表现一样，但细节上看有区别，我们的宿主机在重启后基本上都能恢复一段时间后再复现延迟，但 ServerB 重启也无效。

由此我们判断 ServerA 和 ServerB 的症状并不是同一个问题，并让 ServerB 先升级固件看看。在升级固件后 ServerB 恢复了正常，那么我们的宿主机是不是也可以靠升级固件修复呢？答案是并没有。升级固件后没过几天，延迟的问题又出现了。

四、意外发现

回过头来看之前为了排查 Skylake 时钟漂移问题的 ServerA，上面一直有个简单的程序在运行，来统计时间漂移的值，将时间差记到文件中。当时这个程序是为了验证时钟漂移问题是否修复，如图 7：

```c
#include <stdio.h>

#include <unistd.h>
#include <time.h>


struct timespec diff(struct timespec start, struct  timespec end)
{
    struct timespec temp;
    if ((end.tv_nsec-start.tv_nsec)<0) {
        temp.tv_sec = end.tv_sec-start.tv_sec-1;
        temp.tv_nsec = 1000000000+end.tv_nsec-start.tv_nsec;
    } else {
        temp.tv_sec = end.tv_sec-start.tv_sec;
        temp.tv_nsec = end.tv_nsec-start.tv_nsec;
    }
    return temp;
}
 int main()
{
    struct timespec time1, time2;
    int temp;
    FILE *ff = fopen("logout_clock","wt");
    while (1){
    clock_gettime(CLOCK_MONOTONIC_RAW, &time1);
    usleep(0);
    clock_gettime(CLOCK_MONOTONIC_RAW, &time2);
    usleep(100000);
    fprintf(ff,"time1:%lld,time2:%lld,seconds:%d,nano seconds:%ld\n",time1.tv_sec,time2.tv_sec,diff(time1,time2).tv_sec,diff(time1,time2).tv_nsec);
    fflush(ff);
    }
    return 0;
}
```

图 7

这个程序跑在宿主机上，每个机器各有差异，但正常的时间差应该是 100us 以内，但 1 个多月后，时间差异越来越大，如图 8，最大能达到几百毫秒以上。这告诉我们可能通过这无意中的 log 来找到根因，而且验证了上面 3 的这个猜想，宿主机是运行一段时间后逐渐出问题，表现为第一次打点到第二次打点之间，调度会自动 delay 第二次打点。

```
time1:8239869,time2:8239869,seconds:0,nano seconds:62127
time1:8239870,time2:8239870,seconds:0,nano seconds:55051
time1:8239870,time2:8239870,seconds:0,nano seconds:55734
time1:8239870,time2:8239870,seconds:0,nano seconds:55623
time1:8239870,time2:8239870,seconds:0,nano seconds:54874
time1:8239870,time2:8239870,seconds:0,nano seconds:54834
time1:8239870,time2:8239870,seconds:0,nano seconds:56361
time1:8239870,time2:8239870,seconds:0,nano seconds:3583498
time1:8239870,time2:8239870,seconds:0,nano seconds:54789
time1:8239870,time2:8239870,seconds:0,nano seconds:55415
time1:8239870,time2:8239870,seconds:0,nano seconds:55526
time1:8239871,time2:8239871,seconds:0,nano seconds:54904
time1:8239871,time2:8239871,seconds:0,nano seconds:55798
time1:8239871,time2:8239871,seconds:0,nano seconds:54906
time1:8239871,time2:8239871,seconds:0,nano seconds:55301
time1:8239871,time2:8239871,seconds:0,nano seconds:54596
time1:8239871,time2:8239871,seconds:0,nano seconds:54955
```

图 8

五、TSC 和 Perf

Turbostat 是 intel 开发的，用来查看 CPU 状态以及睿频的工具，同样可以用来查看 TSC 的频率。而关于 TSC，之前的文章《携程一次 Redis 迁移容器后 Slowlog"异常"分析》中有过相关介绍，这里就不再展开。

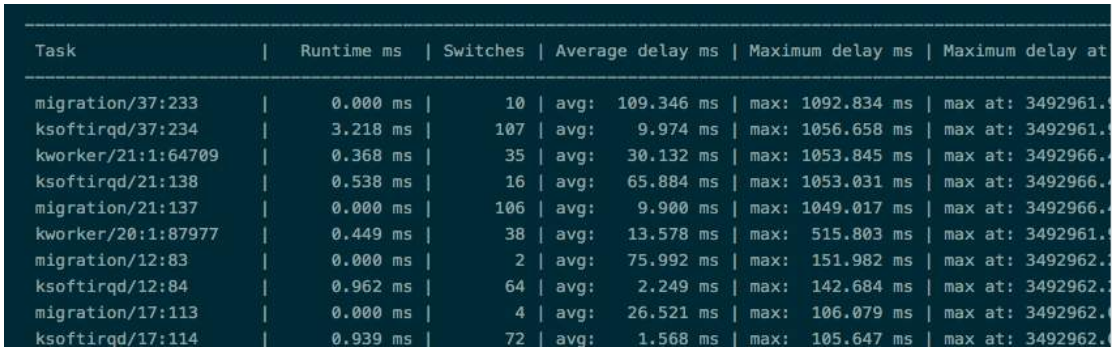
在有问题的宿主机上，TSC 并不是恒定的，如图 9 所示，这个跟相关资料有出入，当然我们分析更可能的原因是，turbostat 两次去取 TSC 的时候，被内核调度 delay 了，如果第一次取时被 delay 导致取的结果比实际 TSC 的值要小，而如果第二次取时被 delay 会导致取的结果比实际值要大。

Avg_MHz	Busy%	Bzy_MHz	TSC_MHz	IRQ	S
168	10.07	1539	1844	217282	0
253	12.78	1533	2196	29565	0
266	13.32	1543	2196	34804	0
270	16.41	1540	1815	33539	0
245	14.77	1553	1815	34009	0
208	13.04	1495	1815	34055	0
216	13.51	1502	1812	30473	0
72	4.50	1508	1812	3163	0
63	3.75	1570	1812	2318	0
140	8.58	1572	1764	4632	0
169	10.74	1581	1696	5712	0
58	3.65	1592	1696	2263	0
56	3.64	1540	1696	2749	0

图 9

Perf 是内置于 Linux 上的基于采样的性能分析工具，一般随着内核一起编译出来，具体的用法可以搜索相关资料，这里也不展开。用 perf sched record -a sleep 60 和 perf sched latency -s max 来查看 linux 的调度延迟，发现最大能录得超过 1s 的延迟，如图 10 和图 11 所示。用户态的进程有时因为 CPU 隔离和代码问题导致比较大的延迟还好理解，但这些进程都是内核态的。尽管 linux 的 CFS 调度并非实时的调度，但在负载很低的情况下超过 1s 的调度

延迟也是匪夷所思的。



Task	Runtime ms	Switches	Average delay ms	Maximum delay ms	Maximum delay at
migration/37:233	0.000 ms	10	avg: 109.346 ms	max: 1092.834 ms	max at: 3492961.
ksoftirqd/37:234	3.218 ms	107	avg: 9.974 ms	max: 1056.658 ms	max at: 3492961.
kworker/21:1:64709	0.368 ms	35	avg: 30.132 ms	max: 1053.845 ms	max at: 3492966.
ksoftirqd/21:138	0.538 ms	16	avg: 65.884 ms	max: 1053.031 ms	max at: 3492966.
migration/21:137	0.000 ms	106	avg: 9.900 ms	max: 1049.017 ms	max at: 3492966.
kworker/20:1:87977	0.449 ms	38	avg: 13.578 ms	max: 515.803 ms	max at: 3492961.
migration/12:83	0.000 ms	2	avg: 75.992 ms	max: 151.982 ms	max at: 3492962.
ksoftirqd/12:84	0.962 ms	64	avg: 2.249 ms	max: 142.684 ms	max at: 3492962.
migration/17:113	0.000 ms	4	avg: 26.521 ms	max: 106.079 ms	max at: 3492962.
ksoftirqd/17:114	0.939 ms	72	avg: 1.568 ms	max: 105.647 ms	max at: 3492962.

图 10

Task	Runtime ms	Switches	Average delay ms	Maximum delay ms	Maximum delay at
kworker/8:2:22841	3.841 ms	185	avg: 7.247 ms	max: 1332.951 ms	max at: 9595693.23
ksoftirqd/8:60	2.430 ms	173	avg: 7.654 ms	max: 1322.959 ms	max at: 9595693.23
kworker/11:2:1028	0.749 ms	45	avg: 45.537 ms	max: 1284.833 ms	max at: 9595708.51
migration/8:59	0.000 ms	9	avg: 134.107 ms	max: 1206.944 ms	max at: 9595693.23
migration/11:77	0.000 ms	2	avg: 968.031 ms	max: 1192.832 ms	max at: 9595708.51
kworker/11:1H:772	0.394 ms	8	avg: 47.417 ms	max: 379.286 ms	max at: 9595710.93
watchdog/11:76	0.000 ms	8	avg: 23.915 ms	max: 191.294 ms	max at: 9595710.93
handler29:1174	49.499 ms	888	avg: 0.147 ms	max: 92.127 ms	max at: 9595692.00
watchdog/8:58	0.000 ms	8	avg: 8.326 ms	max: 66.576 ms	max at: 9595690.80
kubelet:(8)	6073.132 ms	14910	avg: 0.035 ms	max: 19.201 ms	max at: 9595709.67
udevadm:(5)	9.779 ms	47	avg: 0.476 ms	max: 19.042 ms	max at: 9595706.63
dockerd:(7)	151.344 ms	1977	avg: 0.077 ms	max: 15.467 ms	max at: 9595698.91

图 11

根据之前的打点信息和 turbostat 以及 perf 的数据，我们非常有理由怀疑是内核的调度有问题，这样我们就用基于 rdtscp 指令更精准地来获取 TSC 值来检测 CPU 是否卡顿。rdtscp 指令不仅可以获得当前 TSC 的值，并且可以得到对应的 CPU ID。如图 12 所示：

```
#include <stdio.h>

unsigned long tacc_rdtscp(int *chip, int *core)

      {

        unsigned long int x;

        unsigned a, d, c;
  __asm__ volatile("rdtscp" : "=a" (a), "=d" (d), "=c" (c));

         *chip = (c & 0xFFF000)>>12;

         *core = c & 0xFFF;
  return ((unsigned long)a) | (((unsigned long)d) << 32);;

      }
int main()
{
      unsigned long tsc_start, tsc_end;
       int start_chip=0,start_core=0,end_chip=0,end_core=0;
      for (;;)
       {
            tsc_start = tacc_rdtscp(&start_chip,&start_core);
            usleep(100000);
            tsc_end = tacc_rdtscp(&end_chip,&end_core);
           if (tsc_end-tsc_start > 1700686502){
            printf("In 0.1 seconds, the TSC advanced by %lu,start chip core:%d-%d, end chip core:%d-%d\n",tsc_end-tsc_start,start_chip,start_core,end_chip,end_core);
           }
       }
      return 0;
}
```

图 12

上面的程序编译后，放在宿主机上依次绑核执行，我们发现在问题的宿主机上可以打印出比较大的 TSC 的值。每间隔 100ms 去取 TSC 的值，将获得的相减，在正常的宿主机上它的值

应该跟 CPU 的 TSC 紧密相关，比如我们的宿主机上 TSC 是 1.7GHZ 的频率，那么 0.1s 它的累加值应该是 170000000，正常获得的值应该是比 170000000 多一点，图 13 的第五条的值本身就说明了该宿主机的调度延迟在 2s 以上。

```
In 0.1 seconds, the TSC advanced by 248916960,start chip core:1-61, end chip core:1-61
In 0.1 seconds, the TSC advanced by 223017784,start chip core:1-61, end chip core:1-61
In 0.1 seconds, the TSC advanced by 241442826,start chip core:1-61, end chip core:1-61
In 0.1 seconds, the TSC advanced by 486876966,start chip core:1-61, end chip core:1-61
In 0.1 seconds, the TSC advanced by 4708212362,start chip core:1-61, end chip core:1-61
In 0.1 seconds, the TSC advanced by 505758500,start chip core:1-61, end chip core:1-61
In 0.1 seconds, the TSC advanced by 803742164,start chip core:1-61, end chip core:1-61
```

图 13

六、真相大白

通过上面的检测手段，可以比较轻松地定位问题所在，但还是找不到根本原因。这时我们突然想起来，线上 Redis 大规模使用宿主机的内核 4.14.67 并没有类似的延迟，因此我们怀疑这种延迟问题是在 4.14.26 到 4.14.67 之间的 bugfix 修复掉了。

查看 commit 记录，先二分查找大版本，再将怀疑的点单独拎出来 patch 测试，终于发现了这个 4.14.36-4.14.37 之间的（图 14）commit:
https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?h=v4.14.37&id=b8d4055372b58aad4a51b67e176eabdcc238fde3

x86/acpi: Prevent X2APIC id 0xffffffff from being accounted

commit 10daf10ab154e31237a8c07242be3063fb6a9bf4 upstream.

RongQing reported that there are some X2APIC id 0xffffffff in his machine's ACPI MADT table, which makes the number of possible CPU inaccurate.

The reason is that the ACPI X2APIC parser has no sanity check for APIC ID 0xffffffff, which is an invalid id in all APIC types. See "Intel® 64 Architecture x2APIC Specification", Chapter 2.4.1.

Add a sanity check to acpi_parse_x2apic() which ignores the invalid id.

Reported-by: Li RongQing <lirongqing@baidu.com>
Signed-off-by: Dou Liyang <douly.fnst@cn.fujitsu.com>
Signed-off-by: Thomas Gleixner <tglx@linutronix.de>
Cc: stable@vger.kernel.org
Cc: len.brown@intel.com
Cc: rjw@rjwysocki.net
Cc: hpa@zytor.com
Link: https://lkml.kernel.org/r/20180412014052.25186-1-douly.fnst@cn.fujitsu.com
Signed-off-by: Greg Kroah-Hartman <gregkh@linuxfoundation.org>

图 14

从该 commit 的内容来看，修复了无效的 apic id 会导致 possible CPU 个数不正确的情况，那么什么是 x2apic 呢？什么又是 possible CPU？怎么就导致了调度的延迟呢？

说到 x2apic，就必须先知道 apic，翻查网上的资料就发现，apic 全称 Local Advanced

ProgrammableInterrupt Controller，是一种负责接收和发送中断的芯片，集成在 CPU 内部，每个 CPU 有一个属于自己的 local apic。它们通过 apic id 进行区分。而 x2apic 是 apic 的增强版本，将 apic id 扩充到 32 位，理论上支持 2^32-1 个 CPU。简单的说，操作系统通过 apic id 来确定 CPU 的个数。

而 possible CPU 则是内核为了支持 CPU 热插拔特性，在开机时一次载入，相应的就有 online，offline CPU 等，通过修改/sys/devices/system/cpu/cpu9/online 可以动态关闭或打开一个 CPU，但所有的 CPU 个数就是 possible CPU，后续不再修改。

该 commit 指出，因为没有忽略 apic id 为 0xffffffff 的值，导致 possible CPU 不正确。此 commit 看上去跟我们的延迟问题找不到关联，我们也曾向该 issue 的提交者请教调度延迟的问题，对方也不清楚，只是表示在自己环境只能复现 possible CPU 增加 4 倍，vmstat 的运行时间增加 16 倍。

这时我们查看有问题的宿主机 CPU 信息，奇怪的事情发生了，如图 15 所示，12 核的机器上 possbile CPU 居然是 235 个，而其中 12-235 个是 offline 状态，也就是说真正工作的只有 12 个，这么说好像还是跟延迟没有关系。

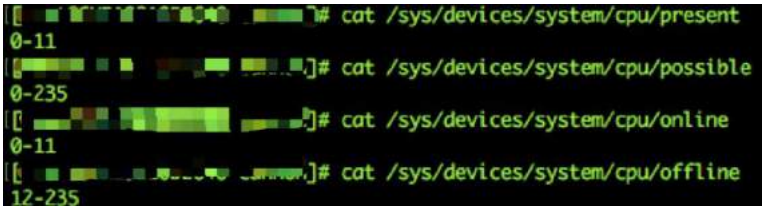


图 15

继续深入研究 possbile CPU，我们发现了一些端倪。从内核代码来看，引用 for_each_possible_cpu()这个函数的有 600 多处，遍布各个内核子模块，其中关键的核心模块比如 vmstat，shed,以及 loadavg 等都有对它的大量调用。而这个函数的逻辑也很简单，就是遍历所有的 possible CPU，那么对于 12 核的机器，它的执行时间是正常宿主机执行时间的将近 20 倍！该 commit 的作者也指出太多的 CPU 会浪费向量空间并导致 BUG（https://lkml.org/lkml/2018/5/2/115），而 BUG 就是调度系统的缓慢延迟。

以下图 16，图 17 是对相同机型相同厂商的两台空负载宿主机的 kubelet 的 perf 数据（perf stat -p $pid sleep 60），图 16 是 uptime 2 天的，而图 17 是 uptime 89 天的。

```
Performance counter stats for process id '17564':

      4911.119674      task-clock (msec)         #    0.082 CPUs utilized
            33,814      context-switches          #    0.007 M/sec
             3,388      cpu-migrations            #    0.690 K/sec
            31,001      page-faults               #    0.006 M/sec
     8,235,208,977      cycles                    #    1.677 GHz
    11,217,407,462      instructions              #    1.36  insn per cycle
     2,374,349,738      branches                  #  483.464 M/sec
        47,377,239      branch-misses             #    2.00% of all branches

      60.001901173 seconds time elapsed
```

图 16

```
Performance counter stats for process id '29620':

      1359.985833      task-clock (msec)         #    0.023 CPUs utilized
            34,253      context-switches          #    0.025 M/sec
             2,168      cpu-migrations            #    0.002 M/sec
             1,842      page-faults               #    0.001 M/sec
     2,210,146,665      cycles                    #    1.625 GHz
     2,475,897,721      instructions              #    1.12  insn per cycle
       503,869,447      branches                  #  370.496 M/sec
         9,120,867      branch-misses             #    1.81% of all branches

      60.002397596 seconds time elapsed
```

图 17

我们一眼就看出图 16 的宿主机不正常，因为无论是 CPU 的消耗，上下文的切换速度，指令周期，都远劣于图 17 的宿主机，并且还在持续恶化，这就是宿主机延迟的根本原因。而图 17 宿主机恰恰只是图 16 的宿主机打上图 14 的 patch 后的内核，可以看到，possible CPU 恢复正常(图 18)，至此超时问题的排查告一段落。

图 18

总结

我们排查发现，不是所有的宿主机，所有的内核都在此 BUG 的影响范围内，具体来说 4.10（之前的有可能会有影响，但我们没有类似的内核，无法测试）-4.14.37（因为是 stable 分支，所以 master 分支可能更靠后）的内核，CPU 为 skylake 及以后型号的某些厂商的机型会触发这个 BUG。

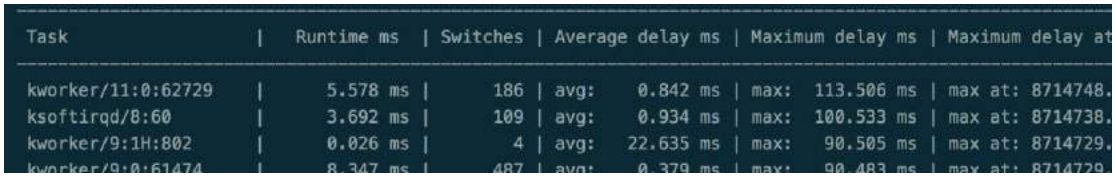
确认是否受影响也比较简单，查看 possible CPU 是否是真实 CPU 即可。

下篇

前言

随着内核升级到 4.14.67，看上去延迟的问题彻底解决了，然而并没有，只是出现的更加缓慢。几周后，超时报障又找了过来，我们用 perf 来分析，发现了一些异常。

如图 1 所示是一个空负载的宿主机升级内核后 8 天的 perf 的数据，明显可以看到 kworker 的 max delay 已经 100ms+，而这次有规律的是，延迟比较大的都是最后四个核，对于 12 核的节点就是 8-11，并且都是同一 D 厂的宿主机。而上篇中使用新内核后用来验证解决问题的却不是 D 厂的宿主机，也就是说除了内核，还有其他的因素导致了延迟。



Task	Runtime ms	Switches	Average delay ms	Maximum delay ms	Maximum delay at
kworker/11:0:62729	5.578 ms	186	avg: 0.842 ms	max: 113.506 ms	max at: 8714748.
ksoftirqd/8:60	3.692 ms	109	avg: 0.934 ms	max: 100.533 ms	max at: 8714738.
kworker/9:1H:802	0.026 ms	4	avg: 22.635 ms	max: 90.505 ms	max at: 8714729.
kworker/9:0:61474	8.347 ms	487	avg: 0.379 ms	max: 90.483 ms	max at: 8714729.

图 1

一、NUMA 和 CPU 亲和性绑定

NUMA 全称 Non-Uniform Memory Access，NUMA 服务器一般有多个节点，每个节点由多个 CPU 组成，并且具有独立的本地内存，节点内部使用共有的内存控制器，节点之间是通过内部互联（如 QPI）进行通信。

然而，访问本地内存的速度远远高于访问其他节点的远地内存的速度，通常有几十倍的性能差异，这也正是 NUMA 名称的由来。因为 NUMA 的这个特性，为了更好地发挥系统性能，应用程序应该尽量减少不同节点 CPU 之间的信息交互。

无意中发现，D 厂的机型与其他机型的 NUMA 配置不一样。假设 12 核的机型，D 厂的机型 NUMA 节点分配如下图 2 所示：

图 2

而其他厂家的机型 NUMA 节点分配如下图 3 所示：

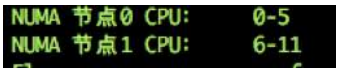


图 3

为什么会出现 delay 都是最后四个核上的进程呢?

经过深入排查才发现，原来相关同事之前为了让 k8s 的相关进程和普通的用户的进程相隔离，设置了 CPU 的亲和性，让 k8s 的相关进程绑定到宿主机的最后四个核上，用户的进程绑定到其他的核上，但后面这一步并没有生效。

还是拿 12 核的例子来说，上面的 k8s 是绑定到 8-11 核，但用户的进程还是工作在 0-11 核上，更重要的是，最后 4 个核在遇到 D 厂家的这种机型时，实际上是跨 NUMA 绑定，导致了延迟越来越高，而用户进程运行在相关的核上就会导致超时等各种故障。

确定问题后，解决起来就简单了。将所有宿主机的绑核去掉，延迟就消失了，以下图 4 是 D 厂的机型去掉绑核后开机 26 天 perf 的调度延迟，从数据上看一切都恢复正常。

Task	Runtime ms	Switches	Average delay ms	Maximum delay ms	Maximum delay at
revalidator34:1232	161.104 ms	465	avg: 0.240 ms	max: 12.007 ms	max at: 2409980.82
kubelet:(8)	1277.912 ms	14535	avg: 0.025 ms	max: 11.434 ms	max at: 2409979.90
revalidator36:1234	163.901 ms	360	avg: 0.353 ms	max: 10.959 ms	max at: 2409983.85
revalidator37:1235	154.776 ms	402	avg: 0.319 ms	max: 10.915 ms	max at: 2409983.85
rsyslogd:(110)	100.207 ms	7165	avg: 0.017 ms	max: 10.422 ms	max at: 2409979.60
revalidator35:1231	211.293 ms	352	avg: 0.250 ms	max: 9.564 ms	max at: 2409986.10
rs:main Q:Reg:865	4.403 ms	49	avg: 0.510 ms	max: 9.094 ms	max at: 2409984.75
redis-server:(30)	6240.949 ms	211674	avg: 0.002 ms	max: 8.992 ms	max at: 2409984.53

图 4

二、新的问题

大约过了几个月，又有新的超时问题找到我们。有了之前的排查经验，我们觉得这次肯定能很轻易的定位到问题，然而现实无情地给予了我们当头一棒，4.14.67 内核的宿主机，还是有大量无规律超时。

三、深入分析

perf 看调度延迟，如图 5 所示，调度延迟比较大但并没有集中在最后四个核上，完全无规律，同样 turbostat 依然观察到 TSC 的频率在跳动。

Task	Runtime ms	Switches	Average delay ms	Maximum delay ms	Maximum delay at
kworker/9:2:144924	0.610 ms	71	avg: 12.576 ms	max: 892.574 ms	max at: 2435977.385304 s
migration/9:64	0.000 ms	41	avg: 25.693 ms	max: 887.871 ms	max at: 2435977.385295 s
ksoftirqd/9:65	1.619 ms	64	avg: 16.397 ms	max: 886.577 ms	max at: 2435977.385313 s
kworker/29:1:676298	0.368 ms	40	avg: 3.424 ms	max: 136.692 ms	max at: 2435977.526106 s
migration/29:185	0.000 ms	107	avg: 1.224 ms	max: 130.680 ms	max at: 2435977.526095 s
ksoftirqd/29:186	2.076 ms	142	avg: 0.944 ms	max: 126.453 ms	max at: 2435977.526114 s
ksoftirqd/26:168	1.299 ms	91	avg: 0.144 ms	max: 5.986 ms	max at: 2435973.417433 s
ksoftirqd/14:96	1.340 ms	122	avg: 0.051 ms	max: 4.987 ms	max at: 2435975.323424 s

图 5

在各种猜想和验证都被一一证否后，我们开始挨个排除来做测试:

1、我们将某台 A 宿主机实例迁移走，perf 看上去恢复了正常，而将实例迁移回来，延迟又

出现了。
2、另外一台 B 宿主机上，我们发现即使将所有的实例都清空，perf 依然能录得比较高的延迟。
3、而与 B 相连编号同一机型的 C 宿主机迁移完实例后重启，perf 恢复正常。这时候看 B 的 TOP，只有一个 kubelet 在消耗 CPU，将这台宿主机上的 kubelet 停掉，perf 正常，开启 kubelet 后，问题又依旧。

这样我们基本可以确定 kubelet 的某些行为导致了宿主机卡顿和实例超时，对比正常/非正常的宿主机 kubelet 日志，每秒钟都在获取所有实例的监控信息，在非正常的宿主机上，会打印以下的日志。如图 6 所示：

```
I0613 16:36:43.336181   12039 container.go:529] [/system.slice] Housekeeping took 456.815669ms
I0613 16:36:45.574415   12039 container.go:529] [/] Housekeeping took 647.602654ms
I0613 16:36:56.419252   12039 container.go:529] [/] Housekeeping took 677.489223ms
```

图 6

而在正常的宿主机上没有该日志或者该时间比较短，如图 7 所示：

```
I0613 20:53:21.500358   12039 container.go:529] [/] Housekeeping took 110.780762ms
I0613 20:53:31.779227   12039 container.go:529] [/] Housekeeping took 123.377625ms
```

图 7

到这里，我们怀疑这些 LOG 的行为可能指向了问题的根源。查看 k8s 代码，可以知道在获取时间超过指定值 longHousekeeping (100ms)后，k8s 会记录这一行为，而 updateStats 即获取本地的一些监控数据，如图 8 代码所示：

```
func (c *containerData) housekeepingTick(timer <-chan time.Time, longHousekeeping time.Duration) bool {
  select {
  case <-c.stop:
    // Stop housekeeping when signaled.
    return false
  case finishedChan := <-c.onDemandChan:
    // notify the calling function once housekeeping has completed
    defer close(finishedChan)
  case <-timer:
  }
  start := c.clock.Now()
  err := c.updateStats()
  if err != nil {
    if c.allowErrorLogging() {
      klog.Warningf("Failed to update stats for container \"%s\": %s", c.info.Name, err)
    }
  }
  // Log if housekeeping took too long.
  duration := c.clock.Since(start)
  if duration >= longHousekeeping {
    klog.V(3).Infof("[%s] Housekeeping took %s", c.info.Name, duration)
  }
  c.notifyOnDemand()
  c.statsLastUpdatedTime = c.clock.Now()
  return true
}
```

图 8

在网上搜索相关 issue，问题指向 cadvisor 的消耗 CPU 过高，
https://github.com/kubernetes/kubernetes/issues/15644
https://github.com/google/cadvisor/issues/1498
而在某个 issue 中指出

（https://github.com/google/cadvisor/issues/1774），echo2 > /proc/sys/vm/drop_caches

可以暂时解决这种问题。我们尝试在有问题的机器上执行清除缓存的指令，超时问题就消失了，如图 9 所示。而从根本上解决这个问题，还是要减少取 metrics 的频率，比较耗时的 metrics 干脆不取或者完全隔离 k8s 的进程和用户进程才可以。

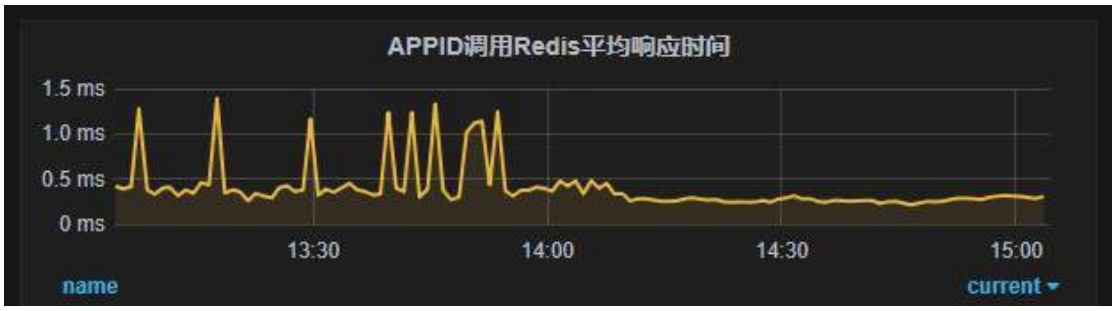


图 9

四、硬件故障

在排查 cadvisor 导致的延迟的过程中，还发现一部分用户报障的超时，并不是 cadvisor 导致的，主要表现在没有 Housekeeping 的日志，并且 perf 结果看上去完全正常，说明没有调度方面的延迟，但从 TSC 的获取上还是能观察到异常。

由此我们怀疑这是另一种全新的故障，最重要的是我们将某台宿主机所有业务完全迁移掉，并关闭所有可以关闭的进程，只保留内核的进程后，TSC 依然不正常并伴随肉眼可见的卡顿，而这种现象跟之前 DBA 那台物理机卡顿的问题很相似，这告诉我们很有可能是硬件方面的问题。

从以往的排障经验来看，TSC 抖动程度对于我们排查宿主机是否稳定有重要的参考作用。这时我们决定将 TSC 的检测程序做成一个系统服务，每 100ms 去取一次系统的 TSC 值，将 TSC 的差值大于指定值打印到日志中，并采集单位时间的异常条目数和最大 TSC 差值，放在监控系统上，来观察异常的规律。如图 10 所示。

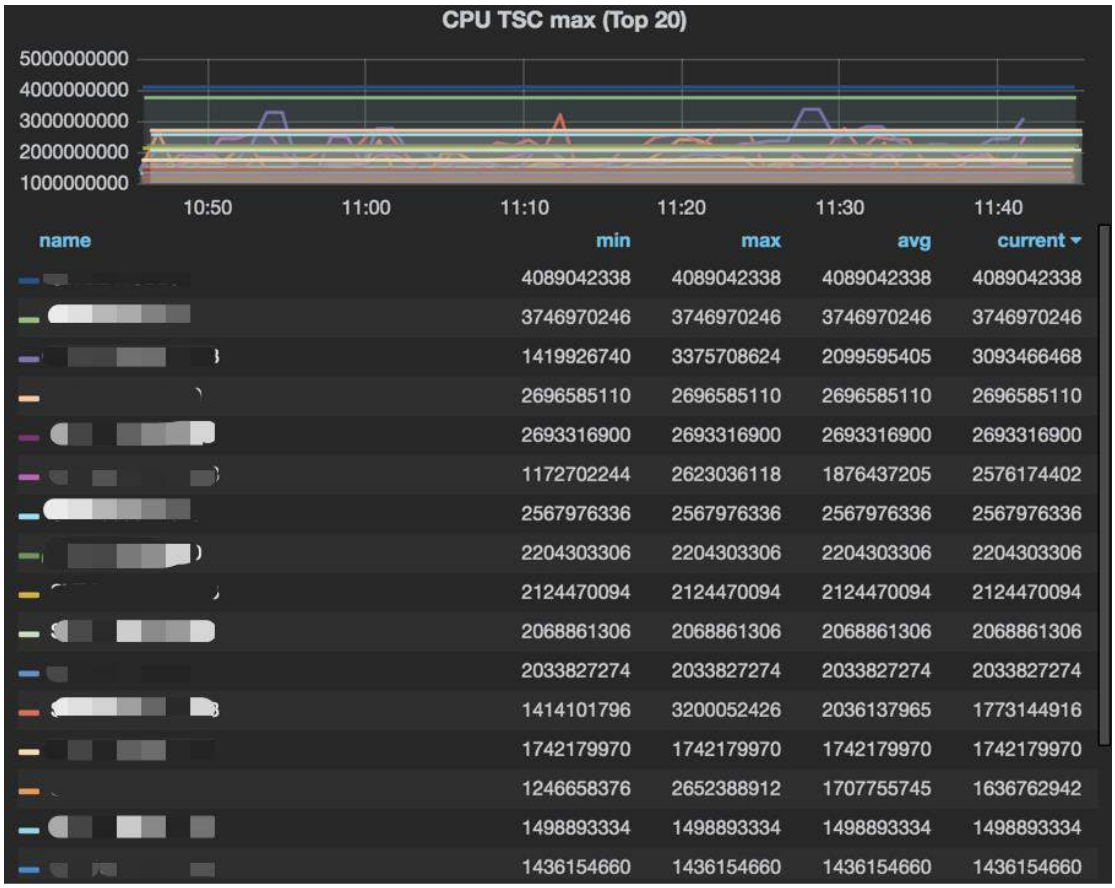


图 10

这样采集有几个好处：

1、程序消耗比较小，仅仅消耗几个 CPU cycles 的时间，完全可以忽略不计；
2、对于正常的宿主机，该日志始终为空；
3、对于有异常的宿主机，因为采集力度足够小，可以很清晰地定位到异常的时间点，这样对于宿主机偶尔抖动情况也能采集到；

恰好 TSC 检测的服务上线不久，一次明显的故障说明了它检测宿主机是否稳定的有效性。如图 11，在某日 8 点多时，一台宿主机 TSC 突然升高，与应用的告警邮件和用户报障同一时刻到来。如图 12 所示：

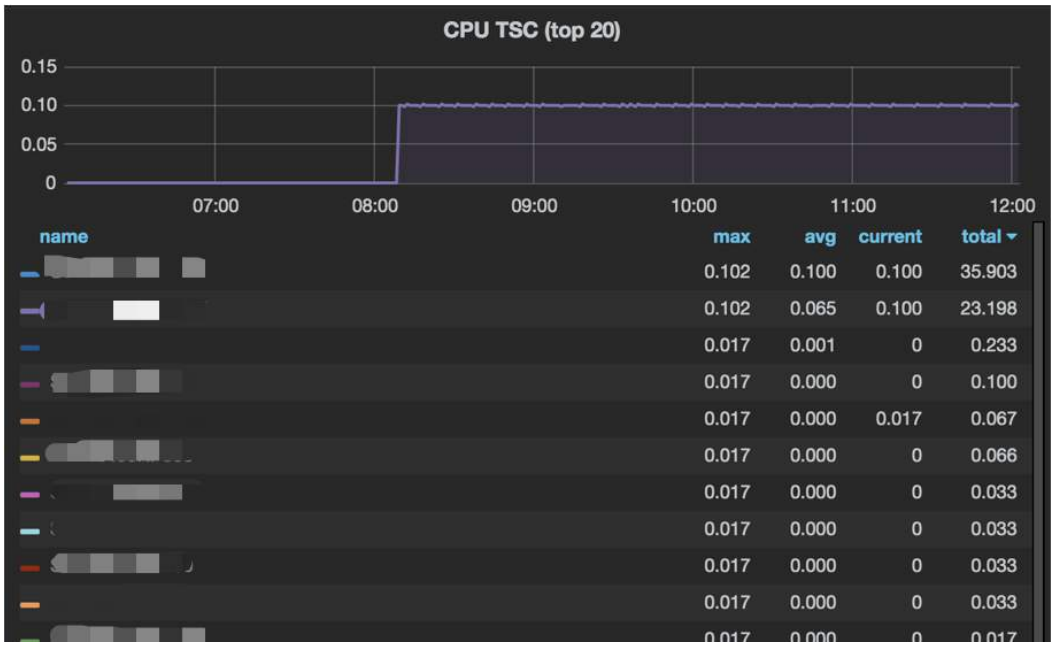


图 11

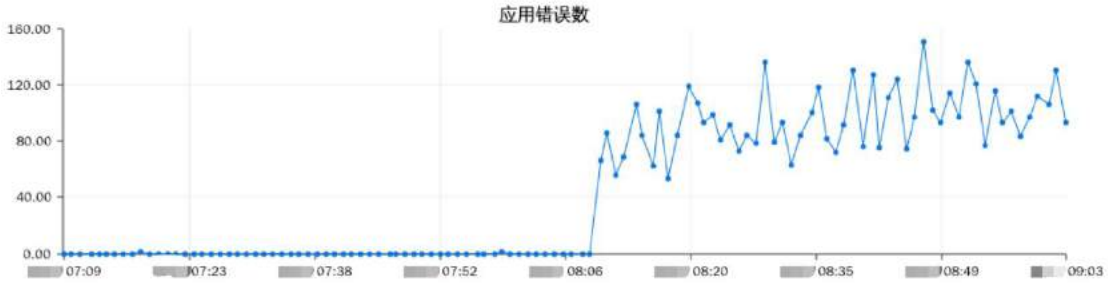


图 12

将采集的日志这样展示后，我们一眼就发现问题都集中在某几批次的同一厂商的宿主机上，并且我们找到之前 DBA 卡顿的物理机，也是这几批次中的一台。我们收集了几台宿主机的日志详情，反馈给厂商后，确认是硬件故障，无规律并且随时可能触发，需升级 BIOS，如图 13 厂商技术人员答复的邮件所示，至此问题得到最终解决。

【分析结论】服务器短时间产生大量UR错误，上报SMI中断到CPU，占用大量CPU资源导致OS卡顿，建议升级BIOS到的105以后版本，该版本开始在服务器端屏蔽了UR错误。

【理论分析】

1、 服务器当前版本处理UR错误的机制，当BIOS检测到UR错误时会发送SMI中断请求到CPU, 如果短时间内UR错误太多（UR风暴），就会使得CPU一直处理中断而无法响应OS，导致OS 卡顿（hang）。

2、 OS TSC时钟，内核在启动过程中会根据既定的优先级选择时钟源。优先级的排序根据时钟的精度与访问速度。
其中CPU中的TSC寄存器是精度最高（与CPU最高主频等同），访问速度最快（只需一条指令，一个时钟周期）的时钟源，因此内核优选TSC作为计时的时钟源。其它的时钟源，如HPET, ACPI-PM，PIT等则作为备选。
正常来说，TSC的频率很稳定且不受CPU调频的影响（如果CPU支持constant-tsc）。内核不应该侦测到它是unstable的。但是，计算机系统中存在一种名为SMI（System Management Interrupt）的中断，该中断不可被操作系统感知和屏蔽。如果内核校准TSC频率的计算过程quick_ pit_ calibrate ()被SMI中断干扰，就会导致计算结果偏差较大（超过1%），结果是tsc基准频率不准确。最后导致机器上的时间戳信息都不准确，可能偏慢或者偏快。

总结：服务器短时间产生大量UR错误时，上报给CPU的SMI中断太多，会导致OS卡顿，并且CPU TSC时钟异常。

图 13

总结

本系列的两篇文章基本上描述了我们遇到的容器偶发性超时问题分析的大部分过程，但排障过程远比写出来要艰难。

总的原则还是大胆假设，小心求证，设法找到无规律中的规律性，保持细致耐心的钻研精神，相信这些疑难杂症终会被一一解决。

携程云原生基础设施演进之路

【作者简介】周昕毅，携程系统研发部云平台高级研发经理。目前负责携程 K8S 平台运维管理、分布式存储和云平台网络组件研发及运维管理。熟悉云基础设施建设，从事运维自动化及 DevOps 工具研发工作十年以上。长期关注云原生技术领域，Infrastructure AS Code 理念的坚定践行者。

随着虚拟化技术和云计算技术的普及，IT 互联网基础设施发生了很大的变化，计算、存储、网络等底层架构也越来越复杂。本文主要介绍携程云平台团队在追求云原生的道路上不断演进携程云基础设施的历程，包括架构选型的思路和踩坑经验的总结。

一、Generation1.0：OpenStack & IAAS 平台建设

携程云 1.0 时代，云平台团队基于 OpenStack 构建 IAAS 平台，旨在提升资源交付效率、统一资源交付标准。虚拟化网络方案基于 OpenStack Neutron 及 Openvswitch 技术实现，网络架构是传统的大三层架构。网络架构如下图：

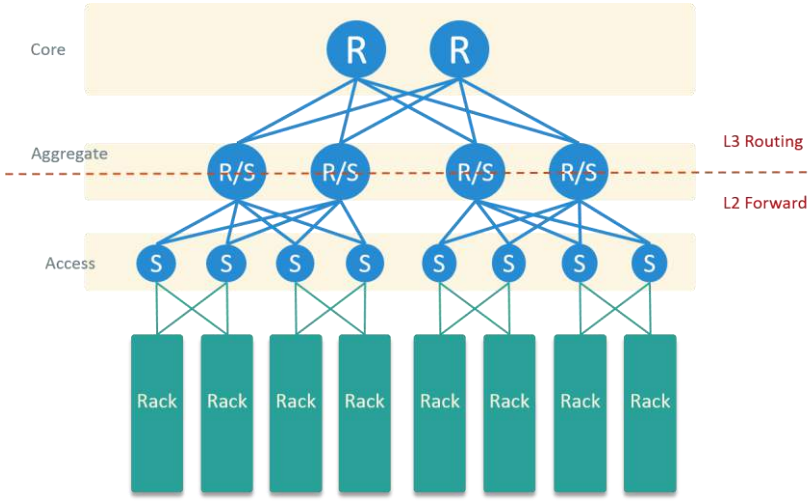


图 1 传统大三层网络架构示意图

经过对 OpenStack nova 调度器的优化、KVM/Vmware 镜像自动下发，虚拟机的交付时长稳定在 2 分钟至 5 分钟左右。但由于虚拟机和物理机镜像的管理成本较高，IAAS 平台仅负责交付标准的预装 OS(如 Ubuntu1604/CentOS71/CentOS76/Windows2012 等)，在 postinstall 过程中安装 Provision 系统的 agent（比如 SaltStack Minion），代码运行环境的标准化是由 Provision 系统来实施。再加上 CMDB、负载均衡系统、自动发布系统等对接的步骤，从用户申请虚拟机到可以提供线上服务，预期时间在 20 分钟到 30 分钟之间。

二、Generation2.0：容器化推进 & Mesos 平台建设

2015 年底开始，云平台开始进行容器化的尝试。由于前几年在 OpenStack 平台积累了很多经验，团队在第一时间选择用 novadocker 组建来进行容器化落地。

容器的网络方案与 Generation1.0 时代的虚拟机网络模型保持一致，继续使用 Neutron+Openvswitch 来落地，这样节省了很多网络选型和测试的时间。

Novadocker 的调度模式与虚拟机类似，第一次实例创建后即落地在固定的宿主机上，因此内部又将 novadocker 创建出的容器定义为“胖容器”。胖容器接入了部分生产业务并稳定运行之后，大家觉得在稳定性方面容器和 VM 并没有太大差异，很快我们启动容器调度平台的选型，让容器实例“动”起来。

2016 年初，在调研了 Docker Swarm、Mesos、Kubernetes 之后，团队普遍认为 Mesos 最为成熟，于是着手开始进行 Mesos 在携程的落地，大家分工进行各技术栈容器镜像规范制定、镜像打包平台建设、PAAS 平台与 Mesos 平台对接、下一代网络方案调研及落地。

在验证了大规模 Mesos 集群的稳定性之后，团队将之前运行在虚拟机上的 java 应用分批次迁移到 Mesos 集群。至此，团队的工作重心从资源交付转向应用交付，研发新申请容器实例到投产平均仅需 30 秒的时间。

在 Generation2.0 的时期，大家逐步认识到容器化带来的实际优势，不可变基础设施的理念不仅帮助了运维团队减轻工作，也为研发团队提升了交付效率，各业务积极配合容器化改造，JAVA 应用容器化覆盖率在很短的时间内超过 90%。

三、Generation2.5：Ctrip Paas + Kubernetes 平台化

在基于 Mesos 平台快速完成 JAVA 应用容器化之后，我们也挖掘出了更多的需求：Python/Nodejs/golang 等多语言技术栈需要进行容器化改造；调度系统的需求越来越复杂（应用分散策略，CPU 绑定，网络 Qos，指定 CPU 指令集的调度）；部分应用有强烈的自动扩容缩容的需求等。越来越多的应用侧需求需要下沉到基础设施层（在当前阶段是容器编排调度平台）来实现。

在连续几个月进行 Mesos 调度器的二次开发工作之后，我们发现 Kubernetes 社区也越来越活跃，Kubernetes 很快演进成了一套 Production Ready 的容器调度平台。在评估了工作成本和风险之后，团队在 2018 年初启动了 Mesos 至 Kubernetes 的迁移。同时，Ctrip PAAS 平台已借助 Kubernetes 的底层能力，为研发提供了更丰富的服务。

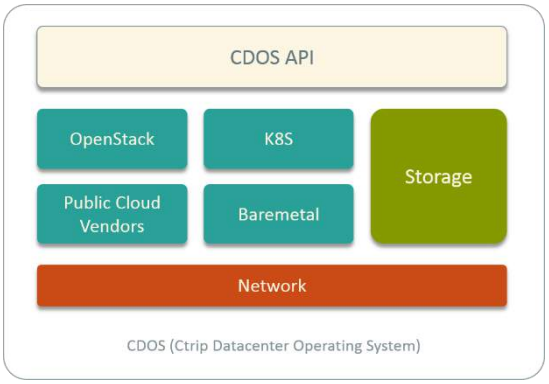


图 2 Ctrip DataCenter OS & Kubernetes 整合

图 3 Ctrip 研发服务平台

在 Generation2.5 时代，我们完成了数千个 Node 规模的 Kubernetes 集群建设，接入了携程超过 1 万个线上应用，覆盖了 Java/Nodejs/Python/golang 等主流技术栈。期间团队在 DockerDaemon 稳定性、内核方面踩坑多次，投入很多精力进行了相关问题的排查和解决，确保平台上业务的持续、高效、稳定运行。在集群规模大了之后，底层网络的问题陆续多了起来，大集群的快速扩容、高频次的 HPA，现有 Neutron 集中式 IPAM 分配管理机制已经成为了瓶颈。

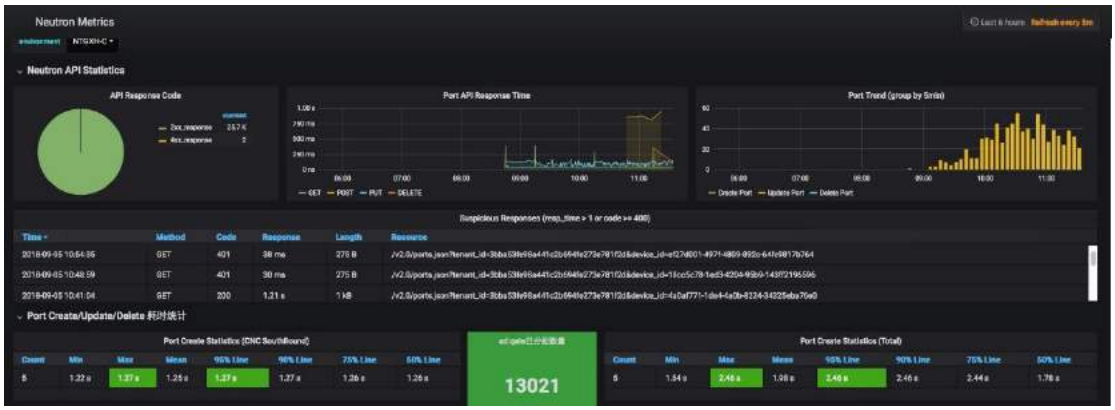


图 4 Generation2.5 Neutron 组件监控看板

我们发现部分应用也仅仅是完成了容器化，即部署模式从物理机或虚拟机换成了 Docker 容器（Cloud Hosting 任务达成）。而 Kubernetes 基础设施平台化能够带来的好处还包括：中心化的资源编排可以最大限度的减少资源浪费(前提是 Kubernetes 覆盖足够大)；强大的管控能力让各领域专家的精力可以集中在领域本身，运维、部署、容灾、分布式、资源分配等繁琐工作由 Kubernetes 来统一解决；Kubernetes 社区的活跃度也可以让业界一流的架构和技术快速在自己的组织中落地。这几大好处促使我们不满足于 Cloud Hosting，掀开了下一阶段工作重心：Cloud Native basedon Kubernetes 的序幕。

四、Generation3.0：Cloud Native & Kubernetes

1991 年 9 月 Linux0.0.1 版本的发布震动了整个开源界，Linux 自由与分享的哲学在开源界传承至今。2015 年,Google 发布了 Kubernetes 1.0 版本，Kubernetes 在 2017 年开始成为了容器编排系统的事实标准。在携程云平台基础设施从 2013 年至今不断演进的过程中，我们愈发认同 Jim Zemlin 的观点：‘Kubernetes is becoming the Linux of the cloud’。我们团队目前致力于打造 Cloud Native 基础设施，追求更高资源利用率、更快部署速度、更强应用治理能力。

4.1 关于更高资源利用率

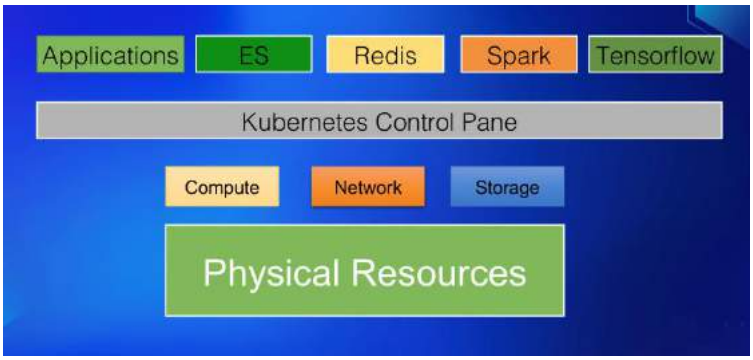


图 5 Kubernetes 统一控制平面

目前携程已经接入 Kubernetes 的应用类型包括: 在线应用、ElasticSearch 集群、Redis 集群、Spark 离线计算 job 集群、AI 协作平台等，其中在线应用集群具有显著的波峰和波谷，可以利用混部将波谷资源释放给 CPU 密集型的离线计算 Job，可以显著提升集群资源利用率。如下图:

图 6 混步后的资源利用率提升

集中式的 Kubernetes 集群监控工具可以快速识别各种潜在异常:

图 7 携程云管平台 Kubernetes 集群体检中心

4.2 关于更快的部署速度

在提升 Provision 速度方面，团队主要围绕两个方面进行优化工作：一是打造容器镜像管理平台，快速构建新镜像，按需分发至多地数据中心，基础镜像提前下发预热，宿主机镜像多数据中心就近下载。二是进行新一代容器网络架构调整，解决集中式 IPAM 的瓶颈，提供更加"云原生"的网络解决方案。

Cilium on Kubernetes 的网络架构基于 ebpf 技术、容器间通信的链路更短；Node 级别的 IPAM 与 Neutron 相比更高效，也降低了全局性网络故障的潜在风险。

Cilium 是 Kubernetes Native 的一套高性能、高动态的网络解决方案，可以原生支持私有云和公有云，同时可以适配 OpenStack、为后续管理 VM 和物理机管理提供了备选方案。Application 层面，Cilium 原生支持所有 Kubernetes 的资源，可以在应用无感知的情况下对网络进行透明的拦截(ebpf 技术)，对混沌工程的支持非常方便。

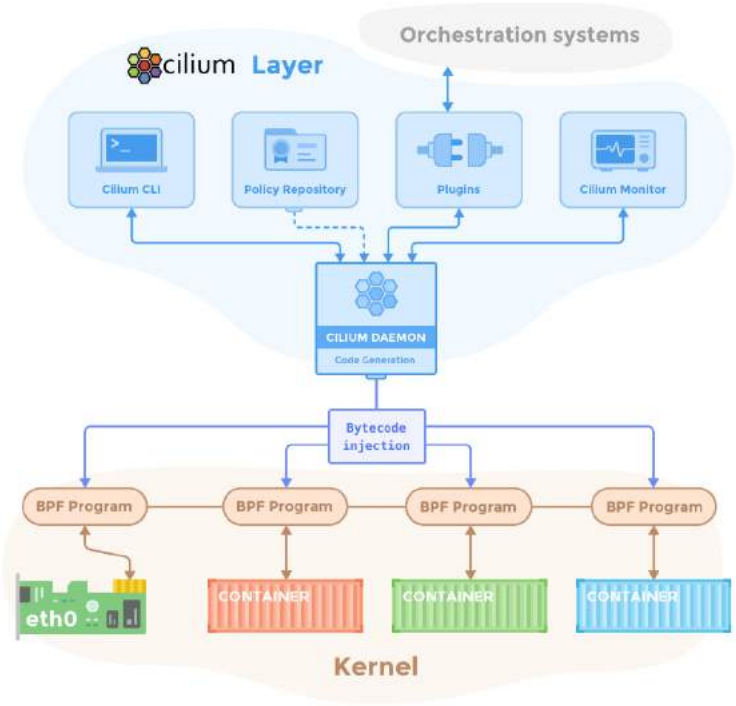


图 8 Cilium on Kubernetes

4.3 关于更强的应用治理能力

Infrastructre as Code 实践:

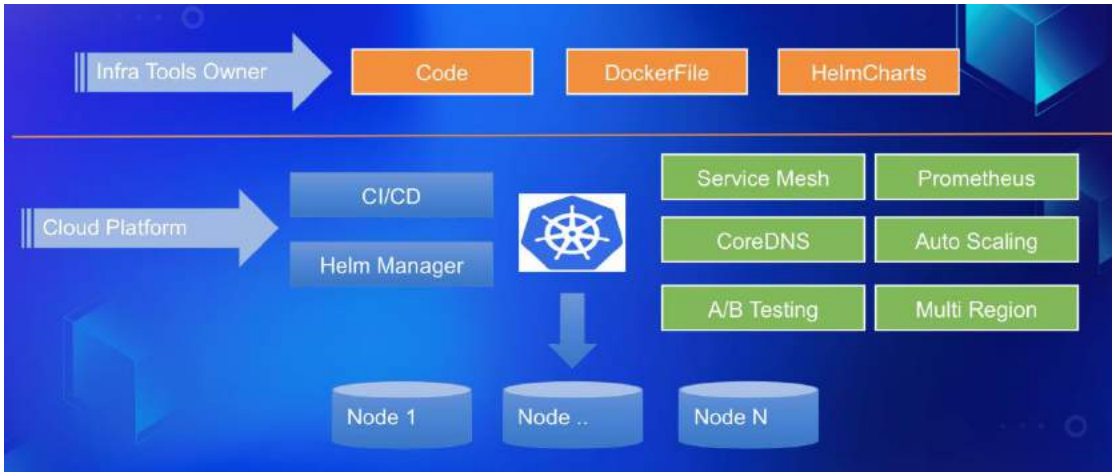
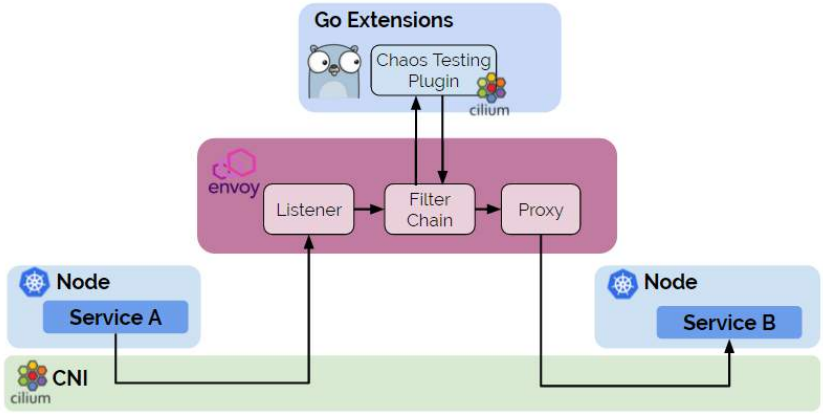


图 9 Infra as Code 实践

Kubernetes 平台为 Infrastructure as Code 实践提供了绝佳的平台，我们基础设施的各类管理服务都在使用 IAC 的方式进行部署和统一管理，在提升稳定性和运维效率的同时享受云原生带来的社区红利。

基于 Cilium，我们同时进行 Network Control as Code 实践，为混沌工程中的网络故障模拟

场景提供了便捷落地的可能；同时，基于内核和 ebpf 我们可以从底层进行网络排障工具研发，从底层定位各类网络侧疑难杂症的根因。




```yaml
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
[...]
specs:
- endpointSelector:
  matchLabels:
    app: myService
  ingress:
  - toPorts:
    - ports:
      - port: "8000"
        protocol: TCP
        l7proto: chaos
        l7:
        - probability: "0.8"
          delay-response: 50ms
        - probability: "0.2"
          delay-response: 1s
```

图 10 & 图 11：Network Control as Code 实践

五、写在最后

在线应用的底层基础设施变更有一定风险，需要投入较多人力进行迁移方案设计，“给高速飞行中的飞机换零件”挑战在于：运行中的系统给迁移方案带来了诸多先天限制，原则上只允许成功，不允许失败。

现在云计算的发展日新月异，不断有新的技术和架构出现，各领风骚三五年，要求架构师们具备一定的技术前瞻性和敏锐的嗅觉，同时在设计架构时需要考虑底层变更的成本。Cloud Native 架构的开放性和包容性及其对于更高的资源利用率、更快的部署速度、更强的管控能力的不断追求，让我们相信这是值得持续投入的正确道路。

注：携程技术微信公众号后台回复“云原生”，可下载讲师分享 PPT。

运维篇

AIOps 在携程的探索与实践

【作者简介】徐新龙，携程技术保障中心资深 SRE，复旦大学信号处理方向硕士研究生。负责携程多个 AIOps 项目的设计与研发，对人工智能、机器学习、神经网络及数学有浓厚的兴趣，对人工智能技术结合运维场景的实践有深入研究。

携程的应用数量众多、架构复杂，规模效应和时间维度上的积累会导致运维数据（日志、监控数据、应用信息等）体量异常庞大，传统基于经验规则的方式已经不能很好地胜任某些特定的运维场景。特别是在大数据时代背景下，这种挑战尤为严峻。

本文将分享携程在 AIOps 方面的一些探索和典型的实践场景，希望通过分享，让大家对 AIOps 以及目前行业发展水平有个宏观的认识，也给对 AIOps 感兴趣的小伙伴一些借鉴和启发。

一、运维面临的挑战

运维数据的体量随着运维规模的快速增长呈现出爆发式地增长。除了对持续交付、持续集成、资源调度、监控能力等提出很高的要求外，面对海量的运维数据，其查找和获取成本也变得非常高。

另外，运维数据的价值和数据成本之间如何平衡、如何取舍，以及如何挖掘有价值的信息，也给运维提出了一定的挑战。

二、AIOps 的理解、定位和现状

结合自身实践以及通过对行业整体水平的分析，介绍下对 AIOps 的理解、定位和现状，以及发展 AIOps 面临的一些挑战。

2.1 运维技术的发展趋势

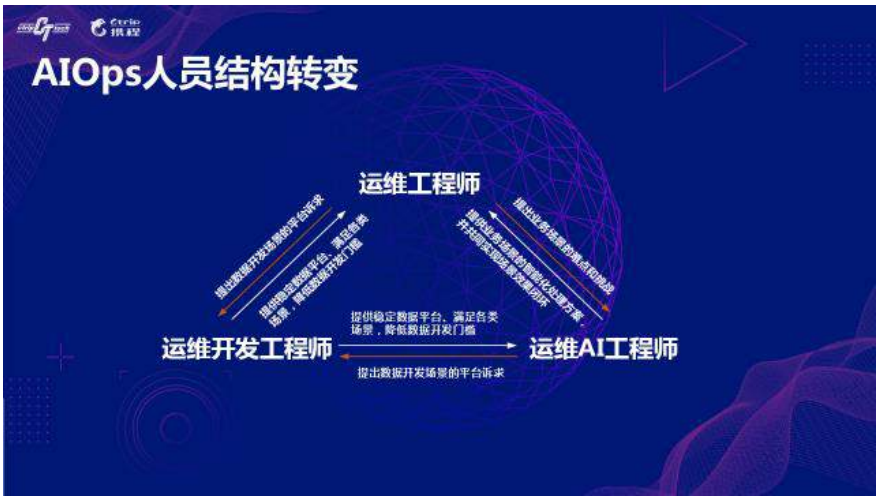
和运维行业中普遍的经历一样，携程的运维方式主要经历了：人肉运维的脚本运维时代，针对特定运维场景的工具化运维时代，打通端到端应用交付的自动化运维时代，到现在正在进行中的智能化运维时代。

AIOps 属于跨领域结合的技术，正式被提出是在 2016 年，随后有多家互联网公司参与实践，2018 年上海的 GOPS 全球运维大会上 70%的分享主题都和 AIOps 有关，2018 年也被业内成为 AIOps 的元年。

2.2 运维人员构成转变

行业中对人员的构成上也按照职能的侧重点不同，划分为运维工程师、运维开发工程师和运维 AI 工程师，大家专业领域不同，分工的侧重也不同。

从个人理解看，这种划分大多数情况下只是一种逻辑划分，例如一个人可以身兼多种角色，而这种复合型的人才是目前非常紧缺的。



2.3 AIOps 现状及实践内容

从行业分享的最佳实践内容看，AIOps 主要围绕质量、效率和成本这三个方面展开，实践包括异常检测到诊断自愈，以及容量到成本的优化。

从行业整体实践水平来看，目前的 AIOps 还处于初级阶段，实践的内容主要还是针对单个应用的场景展开。

2.4 发展 AIOps 的挑战

因为是跨领域结合的技术，所以发展的难度也主要是两个领域的知识都需要有比较深刻的理解和认识。数据质量和算法积累是一方面挑战，复合型人才的稀缺是另一方面。

三、AIOps 在携程的探索与实践场景介绍

单场景实践方面罗列了几个相对成熟的解决方案。

首先是应用监控指标的智能异常检测。传统基于规则方式的告警泛化能力差，经常会导致告警漏告或者误告，智能告警是通过机器学习的方式替代传统基于固定阈值的告警方式。

当检测到应用指标异常之后，我们通过智能应用诊断系统，诊断系统是基于专家经验和相关性检测等算法实现的故障分析诊断系统，可以快速定位故障根源，从而达到快速止损。

另一实践场景是在线资源和离线作业之间的混合部署，基于对在线应用的资源和离线作业的画像，分时动态地将离线作业调度到在线资源上运行，从而达到提升在线资源利用率的目的，同时也提高了离线作业的执行效率。

最后一个要介绍的是智能弹性扩缩容，通过对线上资源构建容量模型，定期生成容量报告，根据容量报告自动执行扩容和缩容。

从 AIOps 整体架构设计和规划方面，主要分为四个逻辑层实现。自顶向下分别包括：运维 KPI 层、运维场景解决方案层、平台自动化层、以及底层基础架构和数据监控层。

底层服务为上次的实现提供能力和平台支撑，未来重点主要会放在平台能力的建设和优化、以及更多智能化运维场景的挖掘方面。

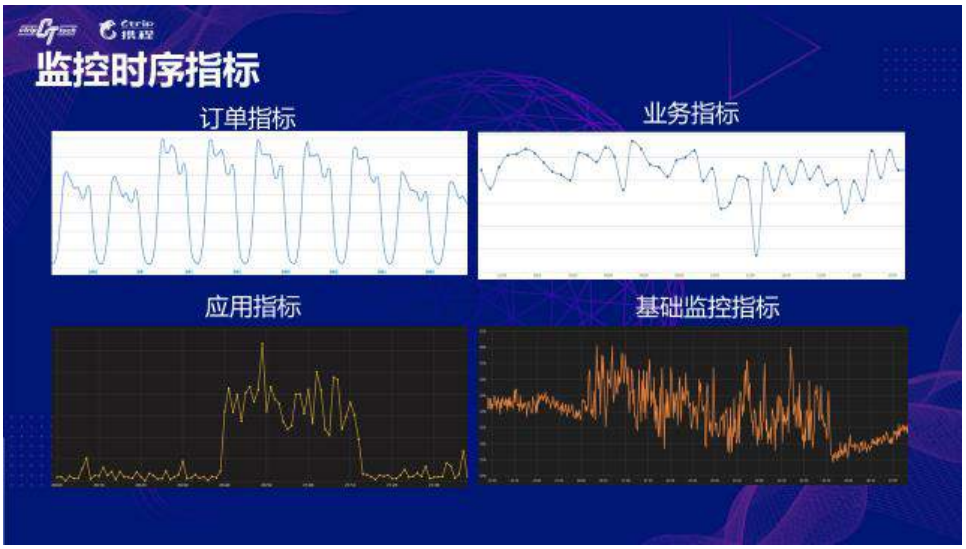
具体的实践运维场景以监控时序的异常检测和应用的智能诊断为例展开介绍:

3.1、监控时序的异常检测

数据体量很小的时候，基于规则的告警方式尚可胜任，当数据体量不断增长之后，规则的泛化能力就会变弱。做监控时序的智能异常检测主要是为了提高告警的质量，减少误报和漏告数量、提高告警的及时性、降低阈值管理的复杂度。

3.1.1 监控时序指标

首先总结下常见的监控时序指标，携程是国内最大的在线旅游互联网公司，和大部分互联网公司一样，监控指标根据功能的不同主要分为以下几类:



订单指标，也是最核心的监控指标，从监控曲线看有非常强的周期性;

应用指标和业务指标，大部分是开发基于框架中间件做的一些业务埋点。这些指标正常情况下都会表现的很平稳，当有突发状况或异常时，指标会剧烈抖动；

基础监控指标，涉及底层各种类型的监控，包括服务器 CPU、内存、磁盘 IO、网络 IO 等指标，以及 DB、Redis、代理、网络等相关监控指标。

3.1.2 异常检测流程

前面提到了常见几种监控时序类型，其中最能够及时反映应用健康程度的就是应用的监控指标（错误数、请求量以及相应时间等），也是应用运维最关心的指标。

异常检测实践的主要内容也是在围绕业务指标跟应用指标展开的，因为携程的应用数量众多，这部分指标的数量也是非常庞大，非常具有挑战性。这里我们主要介绍从数据流向的角度来介绍下时序指标的异常检测。

数据源配置：对于一个通用的异常检测平台而言，待检测的时序数据源可能存在不同的物理介质上，也可能是不同的系统中，为了避免对业务系统的侵入性，异常检测的逻辑一般都是旁路来实现。首先需要将这些不同系统、不同存储介质中的时序进行采集（数据源可以是 DB、HBase、消息、API、以及特定的监控系统等），在异常检测平台中保存一段副本数据，留作构建数据仓库使用。

数据集过滤：实践中我们并不会对所有的数据集都配置智能检测算法，是因为在很多真实的场景中，有些指标很难用被异常检测的算法检测，主要的原因是数据质量不高，有算法经历的道友应该都清楚，数据质量的好坏决定了算法效果的上限。我们会事先配置了一个数据集过滤模块，过滤掉一些数据质量不高的数据集。实现的原理主要是基于数据集的一阶和二阶统计量、偏度、峰度、信息熵等指标，将满足一定统计特性的数据集筛选到后续流程处理。

异常检测算法集：针对预筛选环节过滤得到的数据集，我们准备了常见的异常检测算法集，这些算法大都是通用的机器学习算法根据实际情况和需要做了一定的二次定制，更详细的介

绍我们会在接下来的内容中展开。

告警状态机：这个模块的功能主要是将时序异常转变为一个有效的告警。从事过监控告警的道友应该有类似的共识，异常数据从统计角度看只是离群较远的分布，能不能当做一个业务告警处理呢？大部分时候是需要业务同事来给出规则，将一个无语义的时序异常转变成一个业务告警。例如将连续三次或五次的时序异常转变成一个业务告警，连续多少次之后恢复告警，同时告警状态机会维护每个告警的生命周期，避免重复的告警通知等。

告警质量评价：告警质量的评估可以说是最具挑战性的工作了。一方面，我们检测的指标基本都是无标注的数据集，产生的告警准确与否必须有人来判断；另一方面，因为应用数量众多，每天的告警量也非常的庞大，靠人力逐个去判断几乎是不可能实现的。如果不做告警质量的评价，就无法形成闭环，算法效果也无法得到后续的优化与提升。目前的评价一方面是靠专家经验抽样判断，一方面是邮件将告警推送给监控负责人，通过一定的奖励机制调动用户来反馈告警结果。

3.1.3 异常检测算法介绍

习惯上，按照待检测的数据集有无标签标注可以分为监督式学习、无监督学习以及半监督学习；按照算法模型有无参数将算法分为有参模型和无参模型。

具体算法基本都是大家耳熟能详的，其中大部分算法在实际使用的时候都做过一些二次开发和参数优化，例如某些场景下我们需要将有的算法改写成递归方式实现，用来适配流方式的处理。

上面只是罗列了部分算法，具体的实现算法要远多于这几种。但就这些异常检测算法的思想进行分类的话，无外乎两大类：

一类是监督式的异常检测，这类算法的数据集因为已经打上了标签的，分成了训练数据集和测试数据集，利用训练数据集和对应的标签训练出模型，然后利用学习到的模型再对测试数据集进行检测；

另外一类算法可以归结为基于分布和统计特性的异常检测，这类型的算法针对的是无标注数据集的检测，一个很简单的道理，我们要判断某个指标正常与否，一定需要和某个基准进行比对，这个基准可以是固定阈值，也可以是动态阈值。基于分布和统计特性的异常检测使用的基本都是动态化的阈值。

在大部分的实践场景中，监控指标都是没有标签标注的，这里我们重点介绍下基于分布和统计特性的异常检测原理。

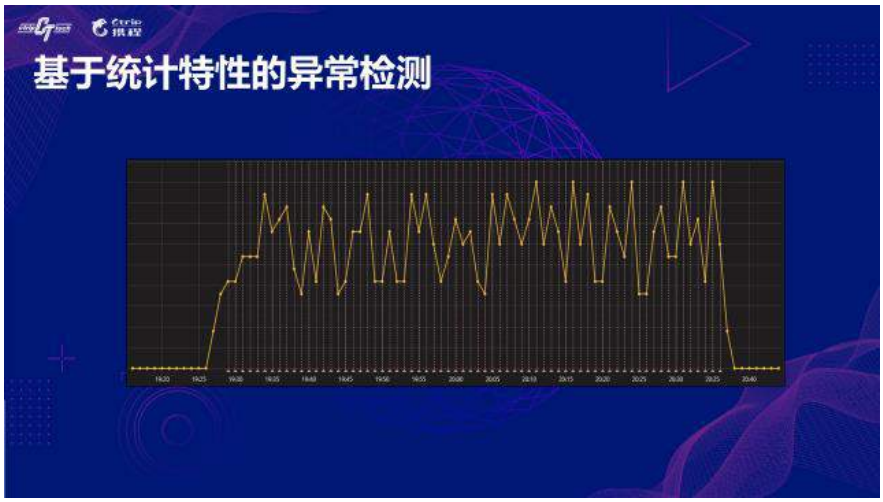
我们提到的绝大部分监控指标，经过统计分析后都是能够近似满足正态分布或者超高斯分布。利用统计特性做异常检测主要是看分布情况，时间轴上的监控序列投影到纵轴上，就可以得到相应的概率密度分布函数，样子如下图所示。

可以看到水平方向很高的点，对应的概率密度非常小，而大部分监控数据的分布都比较集中。直观的看，连续多次小概率事件发生就可以认为是异常事件。

我们借助工业上常用的 3Sigma（标准差）准则作为是否是异常点的检验标准，对标准的正态分布而言，3Sigma 准则可以包括整个样本集 99.7%的分布，也就是说有千分之三的样本会被判定为异常。

对于超高斯分布，也就是形状上比标准正态分布更尖的分布，可能不用 3Sigma，2Sigma 甚至 1Sigma 就可以满足检测需求。除了使用标准差外，四分位差也是经常被用作异常检测的动态阈值。

下面是从我们生产系统里截取的一张图，是某个应用的某项监控指标，竖着的虚线标识的时间点代表该指标有监控告警。可以看到正常情况下这个指标是比较小，按照以往固定阈值的告警方式很难发现这类故障，因为固定阈值动辄就是成百上千的设置阈值，像这种 Case，很容易漏掉告警，但是通过分布和统计特性来检测就很容易发现异常。

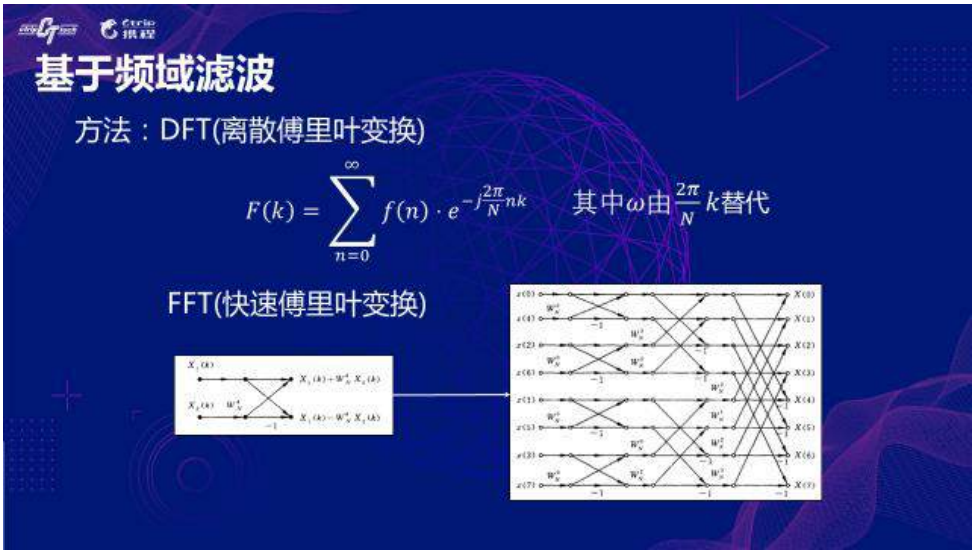


前面介绍了基于分布和统计特性的异常检测规则，原理上就是基于 N Sigma 准则方式实现的动态阈值，其中动态阈值是根据预测基线和 N 倍标准差计算得到的。接下来这个算法主

要是跟大家分享下，我们是如何基于时频变换得到预测基线。

时频变换对很多人来讲可能是个比较陌生的概念，用到的技术叫做傅里叶变换。大家可能或多或少都有一点印象，高等数学有一章级数，曾经提到过傅里叶级数。系统讲解傅氏变换的技术是在信号处理这门课程里边介绍的。

理解傅氏变换的物理意义是很有挑战性，这里简单介绍下如何应用和实现，具体实现需要对监控序列加窗，然后做离散傅里叶变换。下面也给出了具体的计算公式，但由于直接计算相当的复杂，实现上都是通过做快速傅里叶变换实现下的，简称 FFT。很多编程语言都有现成的函数库实现。



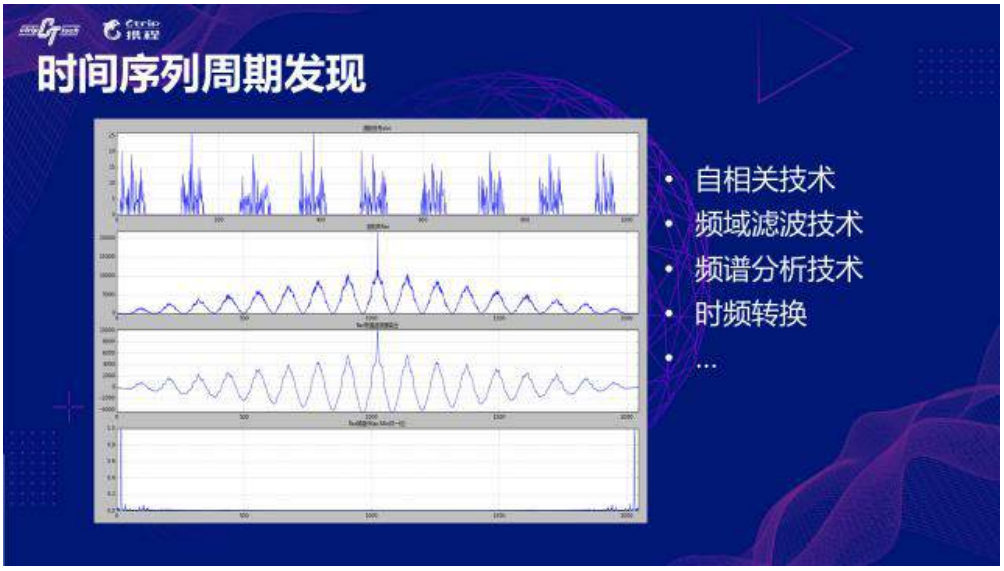
通过前面介绍的时频转换技术，将监控时序变换到频率域之后再对频谱做相应的过滤，去除掉频率较高的成分，然后在时间域重构时序，就可以得到一条相对平稳的基线了。

这里的前提假设就是我们的监控指标正常情况下都是平稳的，渐变的。从我们生产系统里边截取了一段基于频域滤波的监控结果，黄色的曲线是原始的监控指标，绿色的曲线是通过频域滤波之后的基线，可以看到是一条非常平滑和稳定的曲线。从图中可以看到，使用这个技术可以有效的剔除掉异常监控点，确保基线平稳。

基于时频变换的技术，除了前面讲到的可以过滤掉时序中的高频成分生成比较平稳基线时序外，还可以自动发现时序是有包含周期性特征。

以这幅图为例为例，这是从生产系统里边截取的一段真实的监控指标。直观的看，确实是存在明显的周期性，现在我们要做的事情是让程序自动的来识别这个指标的周期。

首先对这个时序做一个自相关，如图中第 2 幅所示，然后去掉自相关序列中频率过低和过高的成分，再去掉均值，如图中第 3 幅所示，这时候结果看上去有点接近正弦波的形状。最后再对上面的结果做一次时频变换就可以得到对应序列的频率谱，如最后一幅图所示。



频率谱左右对称，单位一般取赫兹，频谱上有明显的谱线就说明对应的时间序列存在比较强的周期性，通过一定的数学公式转换，就可以计算出相应的周期大小。

3.1.4 异常检测实践的经验总结

针对前面介绍的关于异常指标检测实践内容，我们简单总结下实践的成果和积累的一些经验，以及识别到的一些问题。

通过智能化告警的实践，应用告警的准确率、召回率相比之前基于规则和固定阈值方式的告警得到了很大的提升。

现在携程默认的应用告警方式已经全替换成了智能告警。大部分实践场景中，这些时序数据都是没有标注的，都是需要结合基于分布和统计特性的异常检测方式。

另外并不是所有的时序都是要采用智能化的方式被检测，这里涉及到算力和成本的投入，如果基于规则的方式可以满足某些场景下的告警检测，那么做智能化检测的意义就不是很大，做成这件事主要还是为了解决“痛点”。

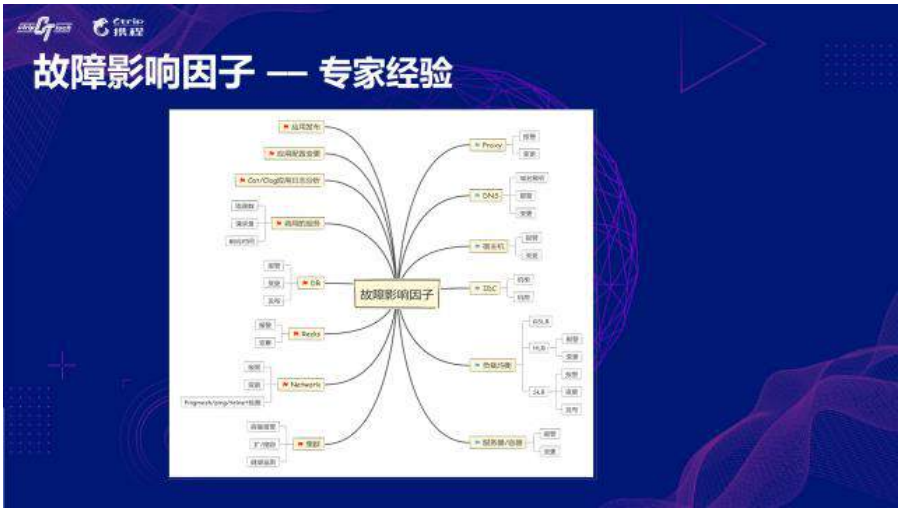
另外就是不同时序的特征可能有所不同，而不同的算法适用场景也有所差异，所以针对不同特征的数据集就需要配置不同的检测算法。

再提一下检测结果的质量评估问题，对于没有标签标注的数据集来说，一直是一个难点和挑战，只有这个环节打通，异常检测才算是真正的闭环了。也只有不断地收集和利用反馈结果，这件事才能越来越智能。

3.2 应用告警的智能诊断

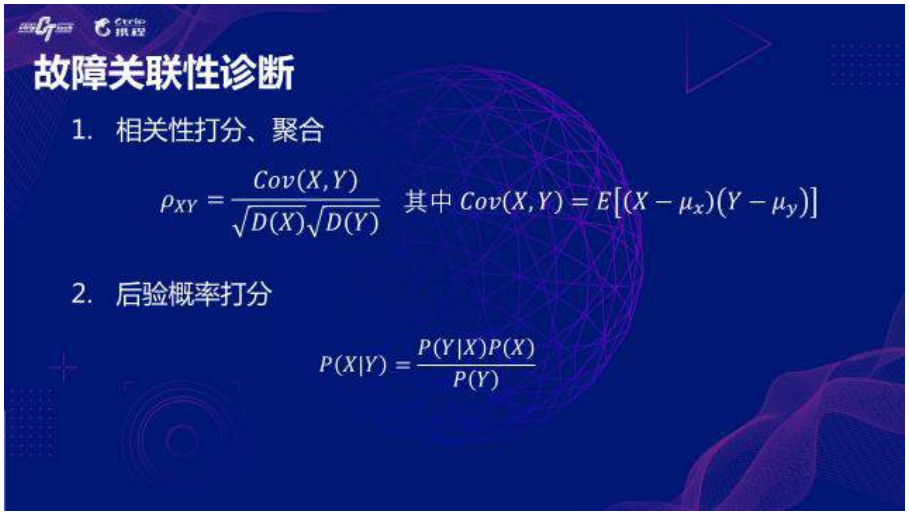
我们先来看一个例子，下面是一张大量应用告警时的应用大盘截图，标红的每个方格表示当前有告警的应用。实际应用之间的调用错综复杂，究竟是哪个应用或什么操作导致了此次故障，需要能快速排查出故障原因，这样就可以为网站快速恢复和止损。

作为坚定的唯物主义因果论者，我们相信万事皆有因果。借助专家经验，我们把所有可能会影响到某个应用异常的因素罗列出来，每个子项称为一个因子分析器，其中包括了应用发布、配置变更、调用链异常、代理故障、数据库发布、DNS 故障、负载均衡器故障、网络变更等等。



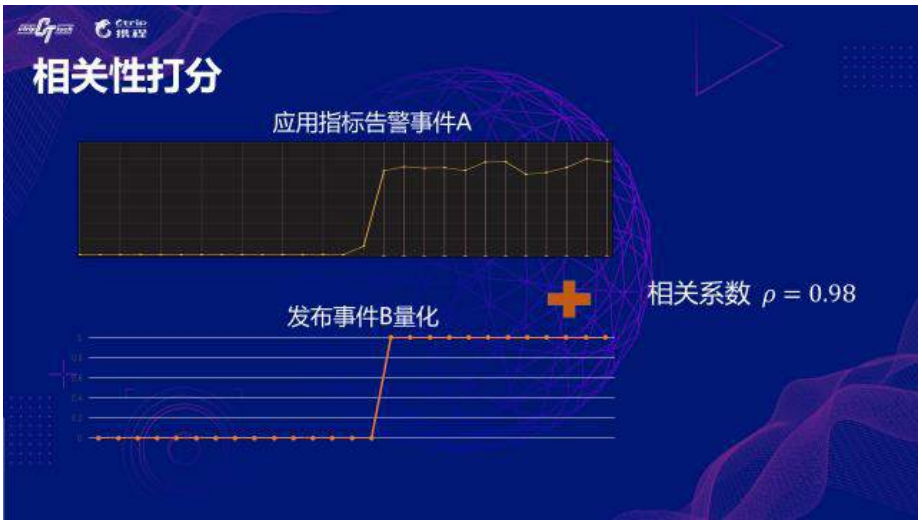
每当发生应用告警的时候，就会自动触发因子分析器分析。引子分析器主要是做运维事件及告警等和应用告警的关联分析，分析结果用百分制打分给出。

主要的算法有两个：一种是基于皮尔逊相关系数计算得到的相关系数；另外一种是基于贝叶斯公式计算得到的后验概率。这两种技术计算的结果，其绝对值都是在 0 和 1 之间，相当于对结果做了归因化的处理，然后对这个结果再乘以 100 就可以直接计算出因子分析器输出的关联分数。



3.2.1 应用的相关性打分、聚合

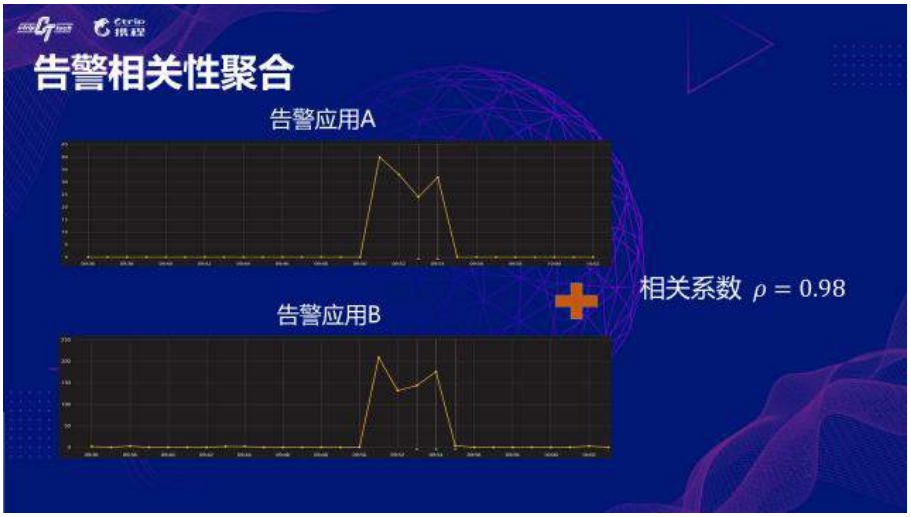
因子分析器种类特别多，这里我们以应用告警指标和发布事件之间的关联分析为例，说明下相关性打分的原理。



上面黄色的曲线代表了某个告警应用的监控指标，记做序列 A；下面的红色的曲线代表了发布事件在时间维度上的量化结果，记做序列 B。此时我们就得到了 A 和 B 两个时间序列，然后计算下皮尔逊相关系数即可计算得出关联分析结果。

同样的思路我们还可以运用到多个告警应用之间的关联性分析。图中上线两条曲线分别代表了两个告警应用的监控指标序列，对这两个监控时序直接计算皮尔逊相关系数，即可求得两个告警之间的关联程度。

另外，我们还会通过框架中间记录的埋点数据分析两个应用之间是否存在调用关系，再结合之前计算得到的相关系数，以此来完成将多个应用告警事件进行聚合和收敛。



3.2.2 后验概率打分

利用后验概率打分，需要积累相当长时间的历史运维事件和关联应用告警的数据，记录并收集在运维知识库。这里我们主要对贝叶斯公式的使用做下说明，使用贝叶斯公式的目的主要是希望从似然概率计算得出后验概率。

似然概率是可以通过对知识库的训练和学习得出的某个运维事件发生的时候，各应用告警发生的概率；后验概率刚好是反过来的，是在应用告警的时候，某个运维事件发生的概率大小，而这些运维事件大部分情况下就是对应应用告警的根源。

上面我们介绍了多种故障关联的方法和实现。实际效果如何呢？这幅图是从我们生产系统里边截的一张故障时候快速定位根源的快拍。

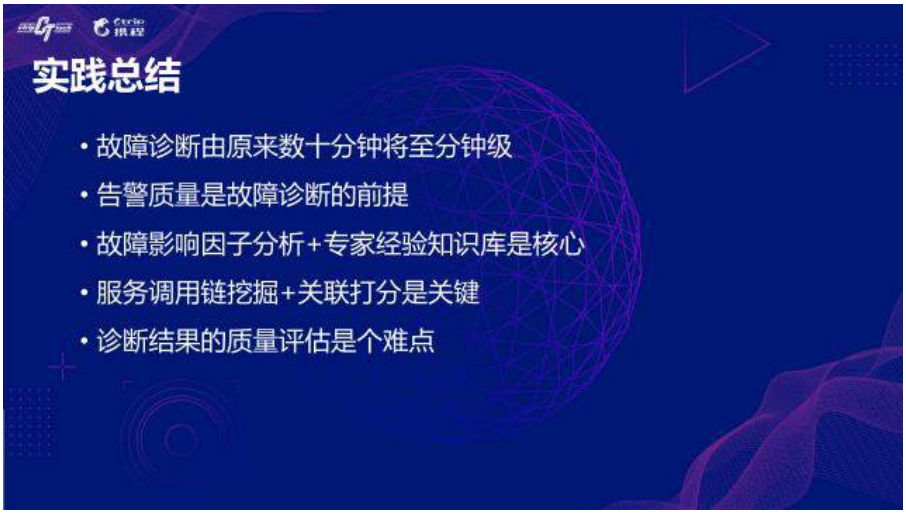
某天某时突然有很多应用告警同时告警，在我们故障诊断系统中一键分析就得出了图中的结果，定位的结果非常明显地指向了中间某个应用，而这个应用当时关联到正在做发布。

3.2.3 故障诊断总结

和前面应用告警的智能检测实践一样，我们总结下故障智能诊断的实践成果、经验和识别到的一些问题。

目前大部分故障发生时，我们已经可以快速的定位出故障根源，大大缩短了恢复故障的时间。因子分析器的设计、专家经验知识库的构建、关联打分、调用链挖掘等都是非常关键的技术点。

要提到的一点是，诊断质量的结果评估和告警质量的结果评估类似，也是一个技术难点。未来的计划是随着反馈数据的不断积累以及知识库的完善，相信这个问题会逐步得到更好地解决。



四、AIOps 未来展望

通过携程在 AIOps 方面的实践和探索，最后简单总结下我们在 AIOps 方面的一些思考和展

望。

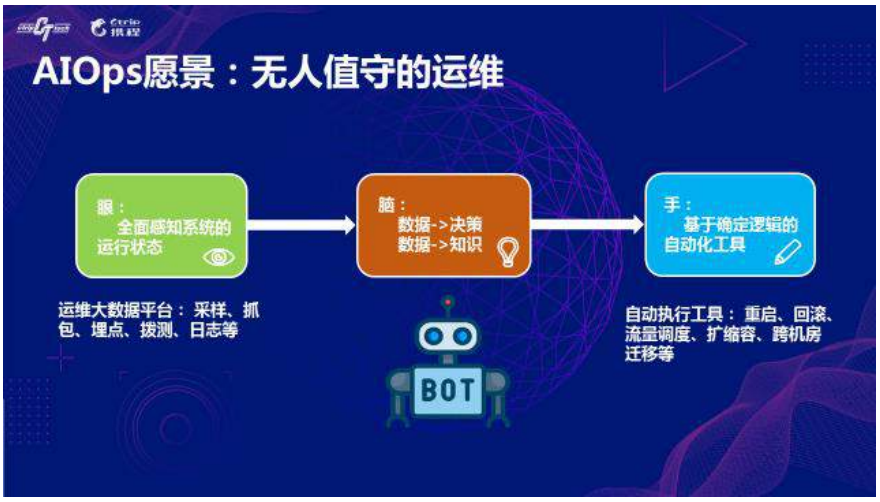
AI 是“他山之石”：AIOps 是一个跨领域结合的技术，AI 只是解决运维问题的一种思路和工具，实践的出发点和落脚点还是 Ops。另外，较高的自动化程度是 AIOps 实践的前提。

实践一定要结合自身场景：我们一直是主张场景优先的原则，实践 AIOps，首先一定要明确自身运维过程中的场景和痛点，不能跟风；警惕拿来主义，AIOps 目前没有明确和统一的实践规范，实践前最好要搞清楚机制和原理，必要的时候做一些二次开发。

紧跟行业动态：大型互联网企业有相当的财力和人力做这件事情，而且一般也很乐意分享相关的实践经验。紧随行业的一些最佳实践，结合自身场景分析讨论落地的可行性，可以避免大方向上走弯路。

学术界跟工业界各贡献一半力量：这是清华裴丹教授在各种 AIOps 会议上分享的时候一直呼吁的，学术界贡献算法，工业界贡献数据和场景，最终实现 AIOps 的美好愿景。

AIOps 总体来说是一个比较新和初期的技术，预测 5 到 10 年后，运维必将是另外一番景象。相信通过构建敏锐的“眼”、智慧的“脑”以及自动化的“手”，“无人值守的运维”的美好愿景终将能实现。


AIOps愿景：无人值守的运维
眼：
全面感知系统的运行状态
脑：
数据->决策
数据->知识
手：
基于确定逻辑的自动化工具
运维大数据平台：采样、抓包、埋点、拨测、日志等
BOT
自动执行工具：重启、回滚、流量调度、扩缩容、跨机房迁移等

为了让携程上万员工上好网，他们做了这些

【作者简介】孙颖， 携程技术保障中心网络管理团队高级工程师。从事 IT 互联网网络运维工作十余年，目前负责 IT 网络及 WiFi 网络设计、建设及运维。

引言

随着移动互联网的飞速发展，WiFi 也已经成为企业办公网络必不可少的基础设施。越来越多的企业对无线办公网产生了极为刚性的品质需求。曾经“WiFi 不好影响工作”的玩笑，放在今天已成为事实。

遗憾的是，许多无线办公网建成后的使用品质与预期存在不小的差别，“网络不好”的抱怨不绝于耳又难以解决，究其原因，主要是由于“交付与运维没有到位”。

对于任何网络系统来说，在设备规格满足需求的情况下，规划、交付和运维水平决定了实际使用效果。但相对于有线网络，WiFi 网络的质量受到更多因素的干扰，更易引起质量下降且排查困难。这给 Wifi 网络的交付与运维带来了很大的挑战。

本文基于实践经验，定位于为 WiFi 组网提供从交付到运维阶段的技术赋能，从以下两方面为企业 WiFi 的 IT 管理者提供一些借鉴与启发。

1）了解 WiFi 运维方法论；
2）提升 WiFi 运维能力。

一、携程 WiFi 平台概述

2015 年携程总部进驻凌空 SOHO。依托主流厂商解决方案，完成无线 WiFi 全面覆盖。目前共计部署 AP 信息点 600+，覆盖达 10+万平方米，,日均活跃终端突破 7000，峰值下行吞吐量超过 1Gbps。

逻辑拓扑（见图 1）

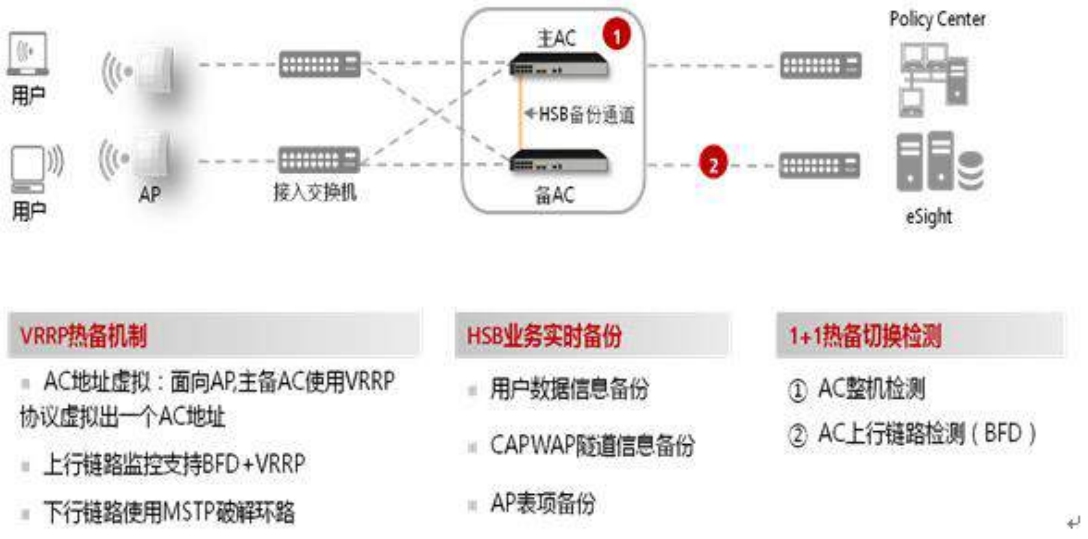


图 1

二、开局篇

首先需要明确，文档主要专注于 WiFi 品质的优化工作，其它相关工作应结合企业自身环境及需求完成基础建设。

开局涉及网络规划、网络交付两个阶段。开局似建筑过程中的地基环节，只有地基打好了，才能起高楼。

大型企业实现 WiFi 高密度部署，确保用户体验的主要挑战：
1）无线的全面覆盖；
2）无线容量与干扰的平衡；
3）内网的安全威胁。

2.1 无线的全面覆盖

以携程为例，办公区域面积大，结构复杂。无线信号的覆盖需要综合考虑建设结构、穿透损耗及布线等具体情况。WiFi 有效覆盖涉及 AP 覆盖、AP 容量两方面，了解 AP 覆盖的基础知识，对 360 度的无死角覆盖及 AP 选型将有极大帮助。

AP 覆盖的有效范围取决于 AP 和终端之间的链路预算。链路预算计算公式如下：终端接收信号强度 = AP 发射功率 + AP 发射天线增益 − 空间传输距离衰减 − 障碍物损耗 + 终端接收天线增益。其中，空间距离对信号的衰减如下：

频段	不同空间传播距离下的无线信号衰减							
	1m	5m	10m	20m	40m	100m	200m	320m
2.4G	46dB	64.2dB	72dB	79.8dB	87.6dB	98dB	105.8dB	113.6dB
5G	56dB	74.2dB	82dB	89.8dB	97.6dB	108dB	115.8dB	123.6m

为满足移动办公场景下 BYOD 不同类型终端（便携笔记本、智能手机、PAD、哑终端等）的接入体验，终端信号强度及 AP 覆盖半径建议值如下：

- 重点覆盖区域终端接收信号电平应大于－65dBm；
- 普通覆盖区域终端接收信号电平应大于－75dBm；
- 空间开阔且用户较少时，AP 覆盖半径＜20 米；
- 空间开阔且用户密集时，AP 覆盖半径 5~8 米为宜；
- 存在少量障碍物遮挡且用户数分布适中时，AP 覆盖半径以 8~12 米；
- 存在大量障碍物遮挡时，重点考虑障碍物对信号的衰减，建议对小空间单独 AP 覆盖。

2.2 信道的配置优化

对于密集和数据流量需求高的场景，密集布放 AP 是提升用户体验的一种重要手段。但密集布放常常导致信道之间的相互干扰，从而影响用户体验。大型移动办公必须对 WLAN 信道进行统一规划并实施。

携程网 WiFi 遵循：双拼蜂窝覆盖、交叉复用原则（见图 2），保证信道间不相互干扰。

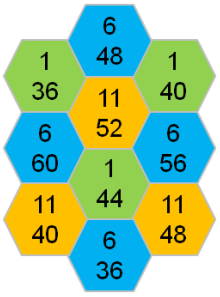


图 2 双频蜂窝覆盖

WiFi 系统主要应用两个频段：2.4GHz 和 5.0GHz。由两个频段自身信道的特性，在高密度的场景下需要尽量的抑制 2.4G 射频，避免低速用户传输对网络传输的影响。

2.3 内网的安全威胁

同一 vlan 内、不同 vlan 间通讯的终端应采用隔离技术，有效防止终端之间传输大量文件损耗 AP 有限的带宽资源，也防止终端之间的任意互访有可能导致的数据窃取、文件中毒等恶意行为，最大限度地确保办公安全，提高办公效率。

三、运维篇

开局篇网络优化只是打好了地基。长期良好的 WiFi 上网品质，是以贯穿整个 WiFi 系统生命周期的优化工作为基础，需要持续投入。

WiFi 运维的痛点：

1）设置参数多，网络优化难。WiFi 网的优化相对来说复杂，包含了射频领域的专业知识，甚至多数情况下无法直接找到优化网络的设置项及设置值，只能通过多维度的数据看到幕后端倪。

2）网络体验数据难以收集和展现。单凭文字描述已经很难达到预期效果，如何量化网络服务水平，将直接制约网络信息部门的工作成果评估。

以上两大烦恼，揪其主要原因在于大多数企业对 WiFi 的运维简单拷贝有线网运维经验，主要依靠厂商提供的网优功能，仅从系统设备层面对系统的健壮性进行监控，而很少从提供用户服务体验的角度建立、健全监控机制。

3.1 规划有效的 KPI 参数

任何网络平台的搭建都有其原生的管理系统管理平台。多数情况，原生管理系统仅从设备性能角度出发，列举尽可能的参数指标。WiFi 系统环境多变，参数繁杂，监控数据的搜集涉及许多层面的知识（诸如功率、信道规划等）。

如果不对其进行梳理，只是简单实现对其有无的监控，则很难发挥这些数据价值，对整个系统缺乏有效评估：一方面导致运维处于被动式的排障；另一方面导致排障阶段出现类似“瞎子摸象”的困局。

解决问题的关键是把“概况-体验”结合，一方面借鉴有线网络运维经验，甄别原有监控平台的各项指标，遴选出全局、局部二个层面的 KPI 综合评分，建立全网主动运维能力；另一方面加强对用户体验的关注，利用自身开发平台，纵深收集用户网络层指标，从用户可用性角度建立用户层面的 KPI 指标。

携程结合自身的运维经验，全局、局部的 KPI 考量、汇总表如下：

维度	指标名称	适用场景
认证服务器	服务器基础	全局WiFi路径设备级监控
	服务状态（死活）	
AC	AC名称	全局WiFi基础网络质量监控
	在线时长	
	CPU实时利用率	
	内存实时利用率	
	接口流量统计	
AP	AP名称	区域性WiFi基础网络质量监控
	AP CPU利用率	
	AP 内存利用率	
	AP 接口速率	
	接入终端	
	接入成功率	
	接入掉线率	
	上/下线监控	
射频	信道利用率	
	噪声强度	

自建监控平台，为用户提供用户角度 KPI 呈现入口，同时将生硬、专业的参数指标转化为网络可达性、可用性指标：（示例见图 3）

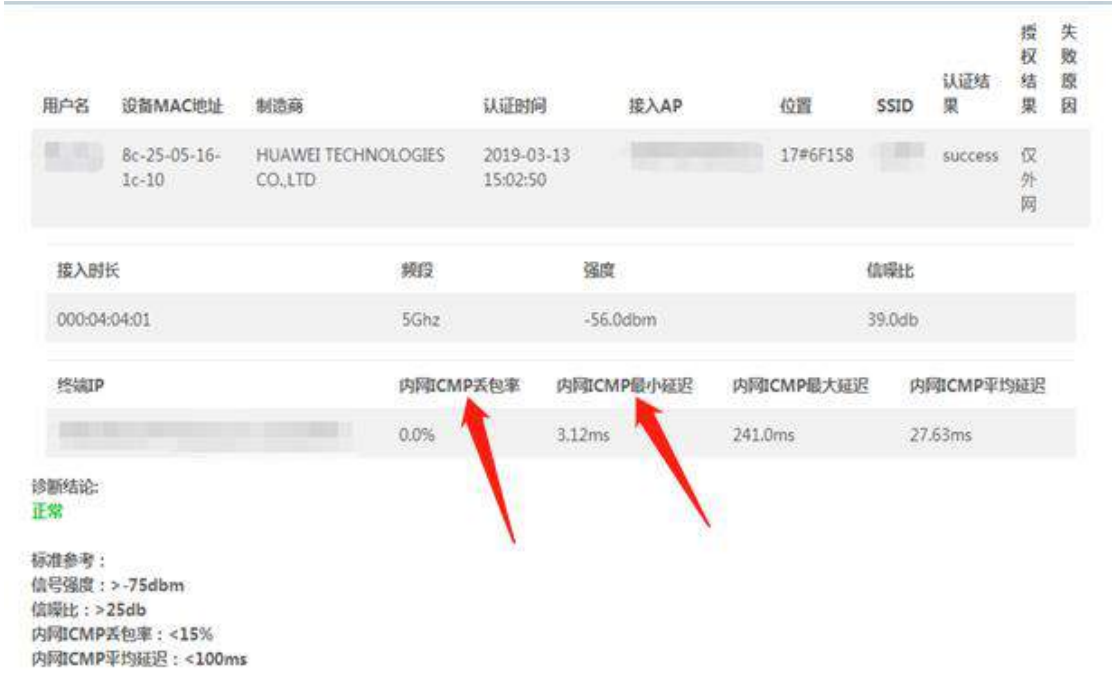


图 3

3.2 量化系统基准及用户评估体验

WiFi 网缺乏量化的数据评估，一直以来是无线网用户体验难有提升空间的原因。WiFi 运维下经常会听到用户反馈“上网慢”等模糊性体验的抱怨之声。在此情况下，因为缺乏有效的基准数据和用户体验量化值，从而造成网络运维人员心理评估基线与用户实际需求管理之间的沟通障碍。

一方面报障阶段数据缺失，运维人员不能准确理解用户抱怨点，造成疲于奔命的解释和漫无目的的查找原因；另一方面解决效果缺乏数据支撑，对用户模棱两可的回答造成用户被忽悠的感觉。WiFi 运维工作处于两难的困境。

3.3 部署“探针“，量化服务基准值

建立用户体验指标，我们就需要广泛收集终端网络访问闭环周期内的相关指标。但由于用户终端设备的私有属性及手机平台的限制，无法通过实际用户终端持续有效的获取用户信息。

对此，携程网络运维团队另辟蹊径，基于“树莓派”产品进行开发，模拟用户 Http 访问,通过拨测方式收集、统计 DNS 解析时长、WEB 连接时长、下载速度等信息，从而实现“基准分析“模块，用直观的方式呈现 WiFi 网络的运行情况。

用户微信的使用效果经常是企业“WiFi 好不好”的直接体现。微信通讯协议：为保证稳定，微信用长链接和短链接相结合，微信划分了 http 模式（short 链接）和 tcp 模式（long 链接），分别应对状态协议和数据传输协议。

1）short.weixin.qq.com 主要用途：

- 用户登录验证;
- 好友关系（获取，添加）;
- 消息 sync (newsync)，自有 sync 机制;
- 获取用户图像;
- 用户注销;
- 行为日志上报。
- 朋友圈发表刷新

2）long.weixin.qq.com 主要用途：

- 接受/发送文本消息;
- 接受/发送语音;
- 接受/发送图片;
- 接受/发送视频文件等。

基于上述说明，携程利用探针程序，通过以下指标，从 DNS 解析-->TCP 连接-->客户端准备-->服务器响应-->数据传输进行阶段监测。（见图 4）

图 4

3.4 量化用户体验值

针对用户反馈无量化问题，携程在内部“程里人”系统下嵌入无线自检工具。用户可主动在终端发起测试，将问题时段的“信号采样“及”WEB 下载速度“直接上报至后台系统，解决用户体验与数据量化之间的矛盾。（示例见图 5）

图 5

四、排障篇

WiFi 排障存在两大难点：

1）网络故障难以重现。很多时候用户反映 WiFi 网问题，需要至现场反复确认，很多问题由于无法重现当时情景，导致无法及时得到处理，从而影响用户体验和服务效率；

2）企业 WiFi 多采用有线无线融合运维，WiFi 存在“背锅”问题。对很多终端用户来说，WiFi 就是互联网，一旦有问题他们就会反馈“WiFi 不好”。“WiFi 不好”背后存在太多可能性，例如互联网接入等出现问题，但由于用户终端缺乏检测手段，很难有效将故障从有线、无线层面进行界定。

解决上述问题的关键在于对用户数据包历史的留存。

4.1 建立用户数据流量包追踪

有线环境对于个体问题定位的终极解决方案就是抓包分析。借鉴该思路，WiFi 排障问题上我们也希望尽可能获取靠近用户终端侧数据包。 考虑 WiFi 传输层的加密及终端环境多变，故障现象短暂等因素，WiFi 环境下终端抓包具有很大局限性。为此则需要从网络层对用户的数据包进行留存。

有了上述思路，数据采样收集点的位置选择则尤为重要。综合三方面考虑：1、尽可能靠近用户侧；2、规避加密传输；3、明确划分有线、无线端。

对此，携程在无线与有线对接点部署“流量采集器”（逻辑图示见图 6），以上帝视角忠实记录了从现在往前一端时间内无线网络的完整数据，排障阶段不管是对历史记录的回溯，还是对复现过程中的模型建立，提供了有效的数据样本。

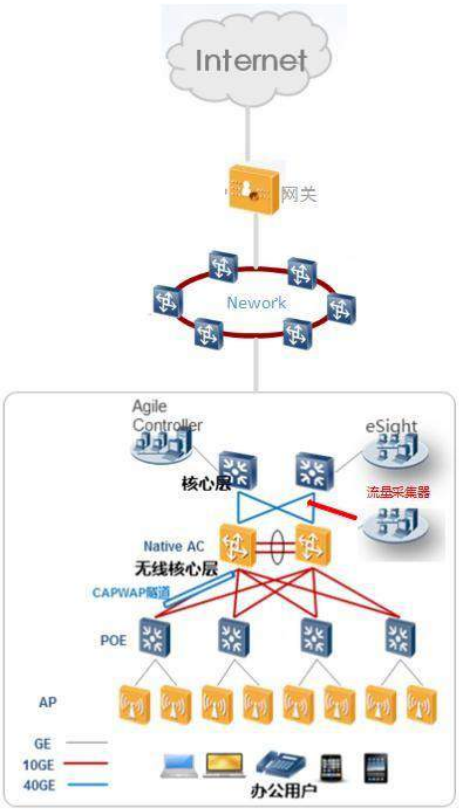


图 6

五、案例篇

通过上述 WiFi 全生命周期监控健全与优化，经过内部实践，切实对问题的排查起到了的事半功倍之效。

案例一，利用“流量采集器”，对 PTK 兼容性引发的网络故障的定位和解决。

内部某用户反馈：iPhoneXS 在连接一段时间后概率性无法上网。通过基础监控平台，我们发现问题时段，故障用户关联的无线设备及用户自身终端的信号状态均正常，但网络通讯中断。

通过“流量采集器”回溯故障时间的用户数据包（见图 7），通过分析，发现其数据流具有以下特点：1）故障前用户存在较大的流量下载行为；2）故障时间段 AC 层面转发正常。

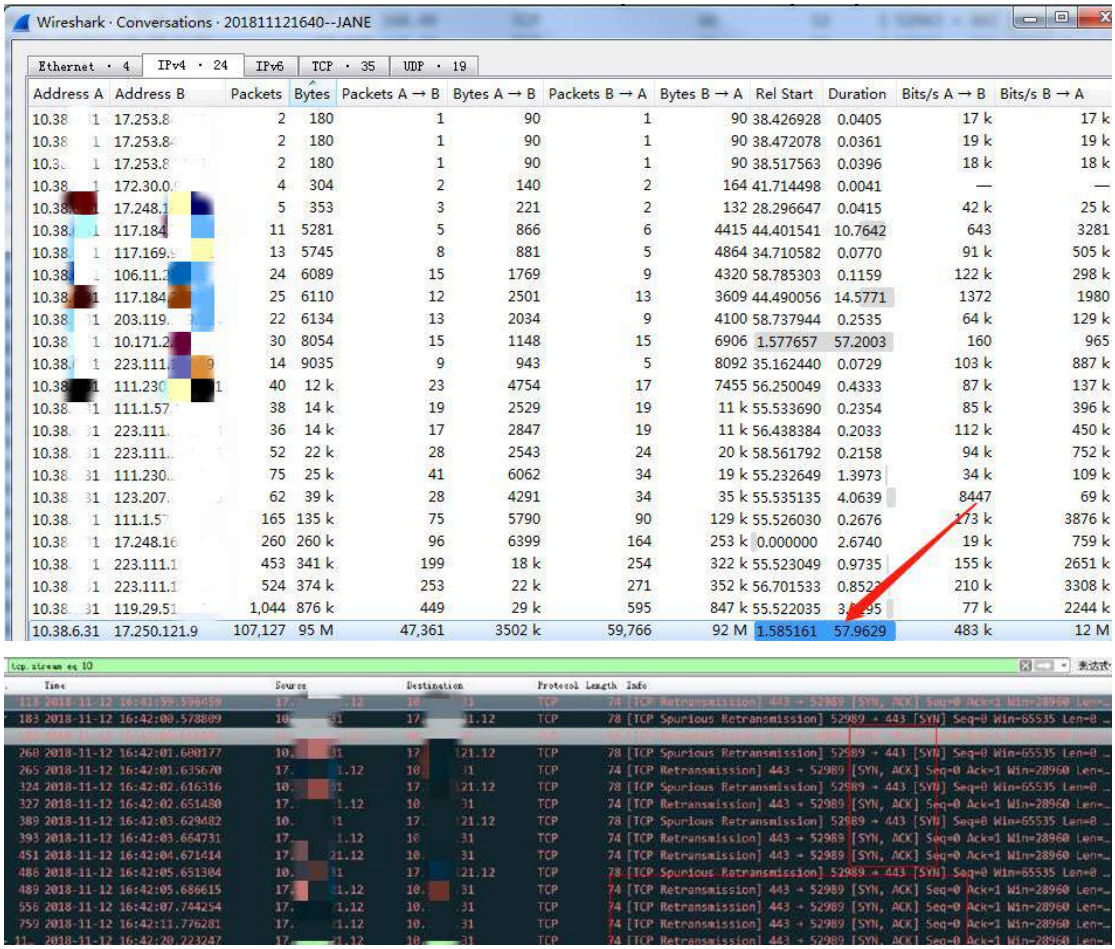


图 7

基于存档数据包分析，故障有效定位在 AC 与终端之间。模拟故障前后用户数据特性，结合实际环境配置参数，问题很快在厂商实验环境得到复现，至此发现问题的根本原因为：iPhone BCM 芯片终端不支持 PTK 密钥更新，PTK 定时更新会触发终端概率性不回报文，导致通信中断。通过关闭设备 PTK 定时更新功能，故障问题得到根本解决。

案例二，结合监控指标及数据流分析，定位跨 AC 访问优化。

某用户终端上报某时间段 WiFi 通讯中断。我们通过无线设备综合评分情况，定位该区域网络整体质量达标，故障现象属于个体问题。

进一步向下通过日志绘制出用户的漫游轨迹，发现问题发生在终端跨 AC 建联后。结合“流量采集器”的数据包，可以观察到终端的下行报文还会转发到漫游前 AC 设备。分析组网结构（见图 8），怀疑跨 AC 前后 MAC 表项与 ARP 表项不统一导致。经过问题复现，上述怀疑得到确认。

经过厂商跟进，确认为交换机存在 CPUCAR 设备偏小问题，导致 ARP 上送过程中有限速丢弃情况，交换机上 arp 表项无法及时刷新到漫游后的流量接口上，导致流量转发异常。

针对上述问题，我们主要通过以下优化措施，对问题进行了有效解决：

1）优化 AP 点位拓扑，尽可能避免同区域的跨 AC 漫游；
2）适当调整 CP car 避免 arp 丢包；
3）网关设备部署 mac 联动 arp，解决 arp 刷新问题.；
4）进行端口隔离，避免资源消耗。

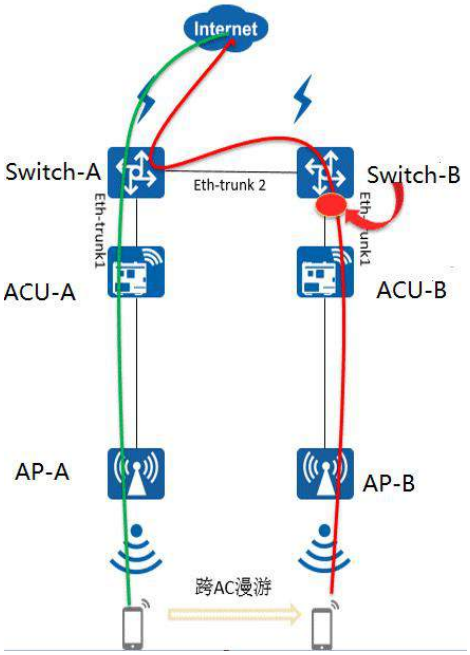


图 8

六、展望

WiFi 优化操作应该基于广泛全面的数据支撑，而不是凭感觉、凭经验，虽然在此之上我们已探索一二，但 WiFi 运维仍大有可为。

如何依托有效的数据搜集，通过机器学习，感知指标变化，提供基于用户体验闭环的智能运维将成为未来之路。携程网将与其它大型网络平台，携手并进演进之路，让“无线办公”变得“无限精彩”。

携程一次 Dubbo 连接超时问题的排查

【作者简介】
顾海洋，携程框架架构研发部技术专家，负责携程分布式服务化领域的工作。目前主要负责 Dubbo 在携程的二次开发和推广工作。
李伟，携程云平台技术专家，2012 年加入携程，目前专注于云原生方向技术的研究和落地，先后参与负责过携程部署架构改造、接入层架构升级等项目的设计和推广。

工作中，常常会遇到连接超时的问题，一般都是先检查端口状态，然后再检查 CPU、Memory、GC、Connection 等机器指标是否正常。如果都在合理范围内就会怀疑到网络或者容器上，甩手丢给网络组同事去排查。

今天，我们想分享一个高并发场景导致的 connect timeout，对原因以及过程的分析或许可以帮助大家从容地面对类似问题。

一、问题背景

携程度假事业部的某个核心服务在两个机房一共有 80 台机器，每台机器都是 4C8G 的 docker 容器。这个服务的调用方比较多，几十个调用方的机器加起来大概有 1300 多台。

SOA over CDubbo 是将现有 SOA 框架的 HTTP 传输协议切换到 TCP 协议，能够解决长尾问题以及提供更好的稳定性。大概实现原理是，服务端通过 CDubbo 启动代理服务，客户端在服务发现后与服务端同步建立 TCP 长连接，请求也会在 TCP 通道传输。

但是，度假事业部的这个服务每次发布总是会有部分客户端报 connect timeout，触发大面积的应用报警。

```
com.alibaba.dubbo.rpc.RpcException: Fail to create remoting client for service(dubbo://ip:port/bridgeService) failed to connect to server /ip:port, error message is:connection timed out: /ip:port
at com.alibaba.dubbo.rpc.protocol.dubbo.DubboProtocol.initClient(DubboProtocol.java:364)
at com.alibaba.dubbo.rpc.protocol.dubbo.DubboProtocol.getSharedClient(DubboProtocol.java:329)
at com.alibaba.dubbo.rpc.protocol.dubbo.DubboProtocol.getClients(DubboProtocol.java:306)
at com.alibaba.dubbo.rpc.protocol.dubbo.DubboProtocol.refer(DubboProtocol.java:288)
```

从日志分析，是 CDubbo 代理服务 TCP 连接失败，还好当初设计的时候考虑到降级机制，没有影响到用户流量。有同事提到既然没有影响，是否可以考虑把日志降级。这么诡异的问题，不知道是否会有其他层面的问题需要去优化的呢，作为执着的技术人员，我们决定排查到底。

二、服务的端口是否异步打开

调用方的每台机器都要跟 160 个服务端实例建立连接，但是客户端看到的报错量只有几个。所以，最开始怀疑客户端的连接发到服务端，但是端口没有来得及打开，导致少量的连接失败了。

翻了下 SOA 框架在处理实例注册的代码，启动 CDubbo 代理是在注册之前，而且是同步启动的，这样的话就否定了端口没打开的可能。

```
// should run after policy service configuration
                    e().setTopLevelLog("Init CDubbo Soa Configuration ...");
                   til.getInstance(CDubboSoaInitializer.class).initialize();

            getIn  ance().setTopLevelLog("Begin registry ...");
                   try.getInstance().register();
```

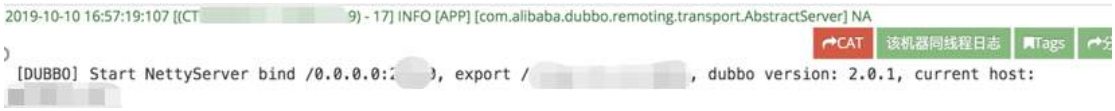
三、怀疑注册中心推送出现了问题

正常情况下的注册发现机制是在服务端健康检查通过后，再把实例推送到客户端。是否注册中心推送出了问题，服务没注册完就把实例推送到客户端了？或者，客户端实例缓存出现问题导致的呢？

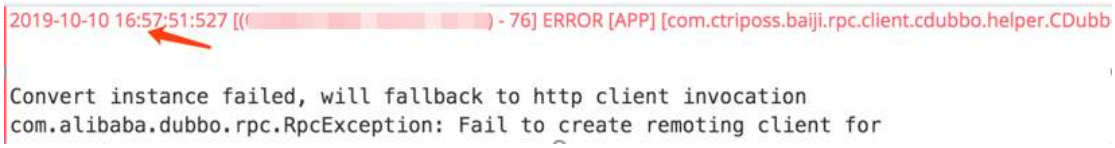
这类问题还是要从日志入手，翻了下 Dubbo 的代码，如果 Netty 打开端口之后，是会记录端口打开时间的。

```
public AbstractServer(URL url, ChannelHandler handler) throws RemotingException {
    super(url, handler);
    localAddress = getUrl().toInetSocketAddress();

    String bindIp = getUrl().getParameter(Constants.BIND_IP_KEY, getUrl().getHost());
    int bindPort = getUrl().getParameter(Constants.BIND_PORT_KEY, getUrl().getPort());
    if (url.getParameter(ANYHOST_KEY,  defaultValue: false) || NetUtils.isInvalidLocalHost(bindIp)) {
        bindIp = ANYHOST_VALUE;
    }
    bindAddress = new InetSocketAddress(bindIp, bindPort);
    this.accepts = url.getParameter(ACCEPTS_KEY, DEFAULT_ACCEPTS);
    this.idleTimeout = url.getParameter(IDLE_TIMEOUT_KEY, DEFAULT_IDLE_TIMEOUT);
    try {
        doOpen();
        if (logger.isInfoEnabled()) {
            logger.info( msg: "Start " + getClass().getSimpleName() + " bind " + getBindAddress() + ", export " +
        }
    } catch (Throwable t) {
        throw new RemotingException(url.toInetSocketAddress(), null, "Failed to bind " + getClass().getSimpleNam
                + " on " + getLocalAddress() + ", cause: " + t.getMessage(), t);
    }
    //fixme replace this with better method
    DataStore dataStore = ExtensionLoader.getExtensionLoader(DataStore.class).getDefaultExtension();
    executor = (ExecutorService) dataStore.get(Constants.EXECUTOR_SERVICE_COMPONENT_KEY, Integer.toString(url.ge
}
```

从日志系统可以看到端口是在 16:57:19 就已经被打开了。



客户端在 16:57:51 发起的连接居然失败了，这个时候端口肯定是已经被打开了。从这个层面推断注册中心或者缓存机制应该是没有问题的。



那么，是否端口打开后又被莫名其妙的关闭了呢？

四、怀疑端口打开后又被莫名其妙的关闭

不确定是否服务启动后，会有某些未知的场景触发端口被莫名其妙的关闭。于是，在本地模拟服务启动，启动过程中通过 shell 脚本不停的打印端口的状态。

通过以下这段脚本，每 1s 就会打印一次 20xxx 端口的状态。

```
for i in {1..1000}
do
lsof -nPi | grep 20xxx
sleep 1
done
```

从结果中，可以看到 20xxx 端口一直处于 listen 状态，也就是正常情况下并不会被莫名其妙的关闭。

```
TCP *:20xxx (LISTEN)
```

五、增加连接被 accept 的日志

Dubbo 已经打印了前面看到的端口打开的日志，如果再能够看到服务端连接被 accept 的情况就好了。

继续翻了 Dubbo 的代码，对 Netty3 的版本来说，连接被 accept 之后会执行 channelConnected 的。那么，只要在这里加点日志，就可以知道端口什么时候被打开，以及连接什么时候进来的了。

以下是基于 Dubbo 2.5.10 版本增加的日志。

```java
NettyHandler.java
public Map<String, Channel> getChannels() { return channels; }

@Override
public void channelConnected(ChannelHandlerContext ctx, ChannelStateEvent e) throws Exception {
    if (logger.isInfoEnabled()) {
        logger.info( msg: "Received the connection from " + ctx.getChannel().getRemoteAddress().toString());
    }
    NettyChannel channel = NettyChannel.getOrAddChannel(ctx.getChannel(), url, handler);
    try {
        if (channel != null) {
            channels.put(NetUtils.toAddressString((InetSocketAddress) ctx.getChannel().getRemoteAddress()),
        }
        handler.connected(channel);
    } finally {
        NettyChannel.removeChannelIfDisconnected(ctx.getChannel());
    }
}

@Override
public void channelDisconnected(ChannelHandlerContext ctx, ChannelStateEvent e) throws Exception {
    if (logger.isInfoEnabled()) {
        logger.info( msg: "Received the disconnection from " + ctx.getChannel().getRemoteAddress().toString()
    }
    NettyChannel channel = NettyChannel.getOrAddChannel(ctx.getChannel(), url, handler);
    try {
        channels.remove(NetUtils.toAddressString((InetSocketAddress) ctx.getChannel().getRemoteAddress()));
        handler.disconnected(channel);
    } finally {
        NettyChannel.removeChannelIfDisconnected(ctx.getChannel());
    }
}
```

业务同事帮忙升级了版本之后，服务端在 16:57:51:394 已经有连接被 accept 了，连接报错时间是 16:57:51:527，也就是 accept 连接过程中只有一部分被拒绝了。

```
2019-10-10 16:57:51:394 [(                    ) - 74] INFO [APP] [com.alibaba.dubbo.remoting.transport.netty.NettyHandler] NA
                                                                                    该机器同线程日志  Tags
[DUBBO] Received the connection from ,                    dubbo version: 2.0.1, current host:
```

那么，是没有收到这个连接的 syn，还是把 syn 给丢弃了呢，必须要抓包看看了。

六、服务端的 TCP 抓包

正常情况下，需要服务端和客户端同时抓包才有意义。但是，客户端数量实在太多，也不知道哪台机器会报超时，两端一起抓的难度有点打，所以决定先只抓服务端试试。

首先摘掉服务的流量，然后在 Tomcat 重启的过程中抓 TCP dump。从 TCP dump 的结果中可以看到，服务端有一阵子收到了 TCP 的 syn，但是全部没有回 ack。可是 HTTP 的 syn 却正常的回复了 syn+ack，难道是应用层把 syn 包给丢了？

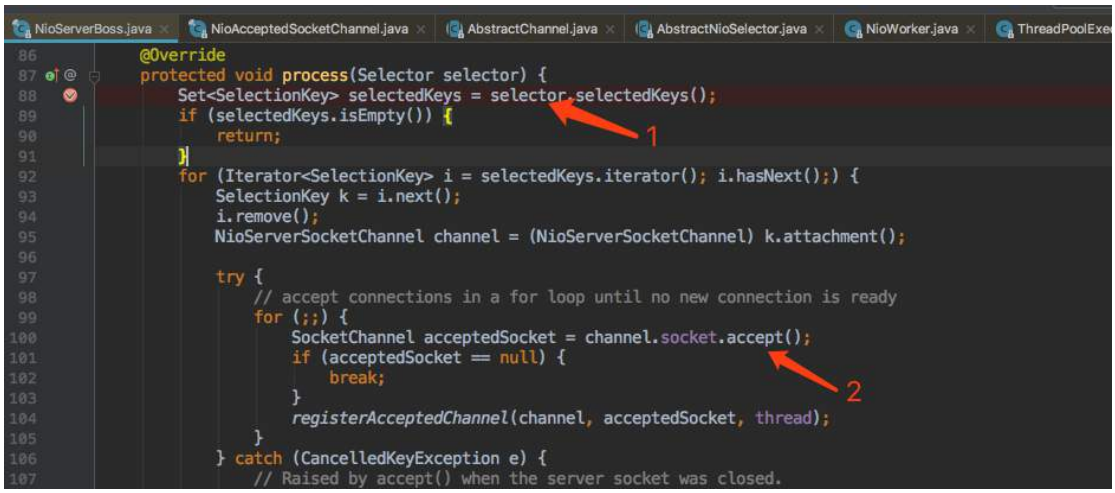
```
5862 41.653245  10.27    .56    10.6    233   TCP    74 4023  → 2    0 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=673563268 TSec
5863 41.653299  10.7     .8     10.6    233   TCP    74 49704 →      0 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=1288214895 TSe
5864 41.653370  10.     .50    10.6    233   TCP    74 51078 →      0 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=3708236043 TSe
5865 41.653374  10.     .229   10.     .233  TCP    74 40220 →      0 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=3079450197 TSe
5866 41.653390  10.     .37    10.     .233  TCP    66 49474 →      [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
5867 41.653393  10.     .16    10.     .233  TCP    74 52186 →      0 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=1775005778 TSe
5868 41.653401  10.     233    10.     9.37   TCP    66 8080  →      [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460 SACK_PERM=1 WS=128
5869 41.653414  10.     97     10.6    233   TCP    74 56674 →      0 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=2196610446 TSe
5870 41.653422  10.     .182   10.6    233   TCP    74 53592 →      0 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=349714822 TSec
5871 41.653417  10.     .97    10.6    233   TCP    74 50164 →      0 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=2330986231 TSe
5872 41.653450  10.     .191   10.6    233   TCP   988 52540 →      0 [PSH, ACK] Seq=1 Ack=1 Win=29312 Len=922 TSval=77475025 TSecr=3975054
5873 41.653455  10.     233    10.6    .191  TCP    66 20990 →      0 [ACK] Seq=1 Ack=923 Win=31872 Len=0 TSval=3975054262 TSecr=77475025
5874 41.653471  10.     210    10.6    233   TCP    74 36662 →      0 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=2362946558 TSe
5875 41.653480  10.28    207    10.6    .233  TCP   994 54312 →      0 [PSH, ACK] Seq=1 Ack=1 Win=29312 Len=928 TSval=3441971044 TSecr=10595
```

没有回 syn+ack 是谁的问题呢，Netty 丢掉的吗？还是操作系统呢？为此，我们做了个小实验。

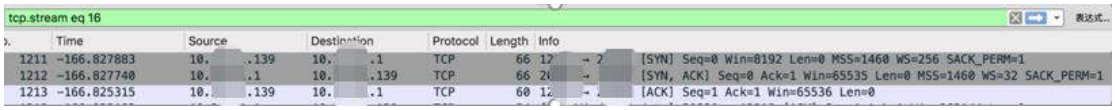
小实验:

如果是应用层丢掉的，那么肯定要从 Netty 的入口处就调试代码。Netty3 的 NioServerBoss 收到请求，会在以下箭头 2 处对连接进行 accept，所以计划在 1 处打上断点。

启动服务端后，再启动客户端，连接进来的时候的确会被箭头 1 处 block 住。



通过 TCP 抓包发现在 accept 之前就已经回复 syn+ack 给客户端了。



这个时候，顺便看了下本机的 20xxx 端口情况，只有一个 listen 状态，没有任何客户端跟它连接。

```
$ lsof -nPi | grep 20xxx
java   24715   Tim   217u   TCP   *:20xxx (LISTEN)
```

之后，继续执行代码，Netty 在 socket 的 accept 执行之后，就可以看到连接已经 ESTABLISHED 成功了。Netty 在 accept 连接之后会注册到 worker 线程进行 IO 处理。

```
$ lsof -nPi | grep 20xxx
java    24715   Tim   0t0   TCP   10.xx.xx.1:20xxx->10.xx.xx.139:12918 (ESTABLISHED)
java    24715   Tim   0t0   TCP   *:20xxx (LISTEN)
```

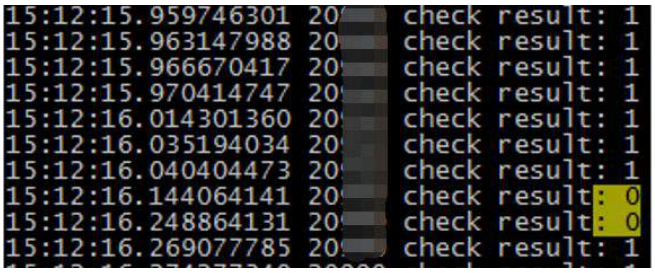
这就证明连接失败不是应用层丢掉的，肯定是操作系统层面的问题了，那么容器内的连接是否会成功呢？

七、从容器内发起的连接是否能成功

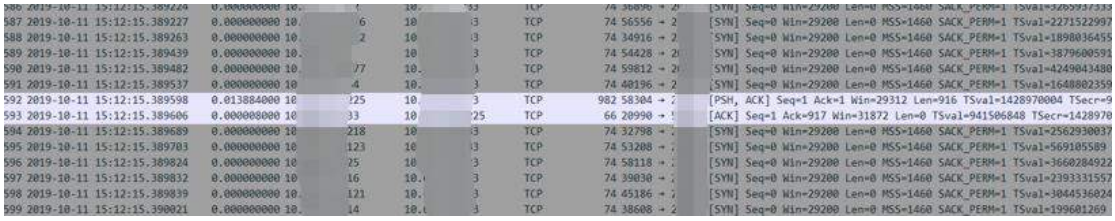
通过重启服务的时候，脚本不停的对服务端端口发起连接，看看是否会有失败的情况。

```
#!/bin/bash
for i in`seq 1 3600`
do
t=`timeout0.1 telnet localhost 20xxx    </dev/null 2>&1|grep -c 'Escapecharacter is'`
echo$(date) "20xxx check result:" $t
sleep0.005
done
```

从脚本的执行结果看到，容器内发起的连接有时也是会失败的，以下黄色高亮的 0 就是失败的连接。



同时，从服务端的抓包结果看到，也会有 syn 被丢弃的情况。

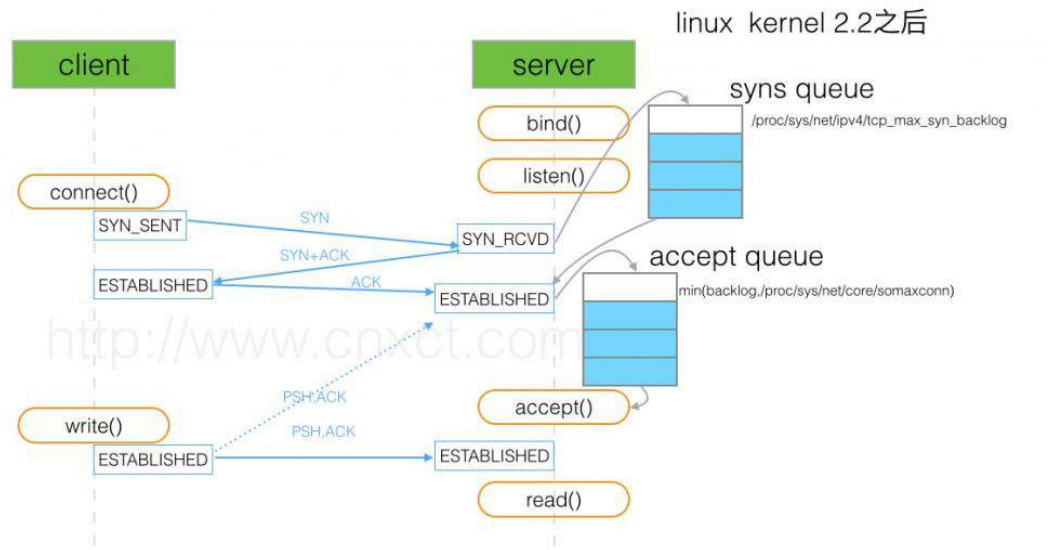


八、全连接队列满导致的 SYN 丢包

syn 包被操作系统丢弃，初步猜测是 syn queue 满了，通过 netstat -s 查看队列的情况。

```
$ netstat -s
3220 times the listen queue of a socket    overflowed
3220 SYNs to LISTEN sockets dropped
```

问题的原因基本找到了，可是导致 syn 被丢弃的原因还是不知道，这里我们先复习下三次握手的整个过程。



上图结合三次握手来说：

1、客户端使用 connect()向服务端发起连接请求（发送 syn 包），此时客户端的 TCP 的状态为 SYN_SENT；

2、服务端在收到 syn 包后，将 TCP 相关信息放到 syn queue 中，同时向客户端发送 syn+ack，服务端 TCP 的状态为 SYN_RCVD；

3、客户端收到服务端的 syn+ack 后，向服务端发送 ack，此时客户端的 TCP 的状态为 ESTABLISHED。服务端收到 ack 确认后，从 syn queue 里将 TCP 信息取出，并放到 accept queue 中，此时服务端的 TCP 的状态为 ESTABLISHED。

我们大概了解了 syn queue 和 accept queue 的过程，那再看上面的问题，overflowed 代表 accept queue 溢出，dropped 代表 syn queue 溢出，那么 3220 SYNs to LISTEN sockets dropped，这个就是代表 syn queue 溢出吗？

其实并不是，我们翻看了源码：

```
exit_overflow:
        NET_INC_STATS(sock_net(sk), LINUX_MIB_LISTENOVERFLOWS);
exit_nonewsk:
        dst_release(dst);
exit:
        tcp_listendrop(sk);
        return NULL;
put_and_exit:
        newinet->inet_opt = NULL;
        inet_csk_prepare_forced_close(newsk);
        tcp_done(newsk);
        goto exit;
}
EXPORT_SYMBOL(tcp_v4_syn_recv_sock);
```

可以看到 overflow 的时候 TCP dropped 也会增加，也就是 dropped 一定大于等于 overflowed。但是结果显示 overflowed 和 dropped 是一样的（都是 3220），只能说明 accept queue 溢出了，而 syn queue 溢出为 0（3220-3220=0）。

从上图的 syn queue 和 accept queue 的设计，accept queue 满了应该不影响 syn 响应，即不影响三次握手。

带着这个疑问我们再次翻看了内核源码：

```
int tcp_conn_request(struct request_sock_ops *rsk_ops,
                     const struct tcp_request_sock_ops *af_ops,
                     struct sock *sk, struct sk_buff *skb)
{
        struct tcp_fastopen_cookie foc = { .len = -1 };
        __u32 isn = TCP_SKB_CB(skb)->tcp_tw_isn;
        struct tcp_options_received tmp_opt;
        struct tcp_sock *tp = tcp_sk(sk);
        struct net *net = sock_net(sk);
        struct sock *fastopen_sk = NULL;
        struct request_sock *req;
        bool want_cookie = false;
        struct dst_entry *dst;
        struct flowi fl;

        /* TW buckets are converted to open requests without
         * limitations, they conserve resources and peer is
         * evidently real one.
         */
        if ((net->ipv4.sysctl_tcp_syncookies == 2 ||
             inet_csk_reqsk_queue_is_full(sk)) && !isn) {
                want_cookie = tcp_syn_flood_action(sk, skb, rsk_ops->slab_name);
                if (!want_cookie)
                        goto drop;
        }

        if (sk_acceptq_is_full(sk)) {
                NET_INC_STATS(sock_net(sk), LINUX_MIB_LISTENOVERFLOWS);
                goto drop;
        }

        req = inet_reqsk_alloc(rsk_ops, sk, !want_cookie);
        if (!req)
                goto drop;
```

可以看到建连接的时候，会判断 accept queue，如果 acceptqueue 满了，就会 drop，即把这个 syn 包丢掉了。

九、全连接队列怎么调整？

我们再看下服务器的实际情况，通过 ss -lnt 查看当前 20xxx 端口的 accept queue 只有 50 个，这个 50 是哪里来的呢？

```
$ ss -lnt
State Recv-Q Send-Q Local    Address:Port Peer Address:Port
LISTEN 0 50 *:20xxx *:*
```

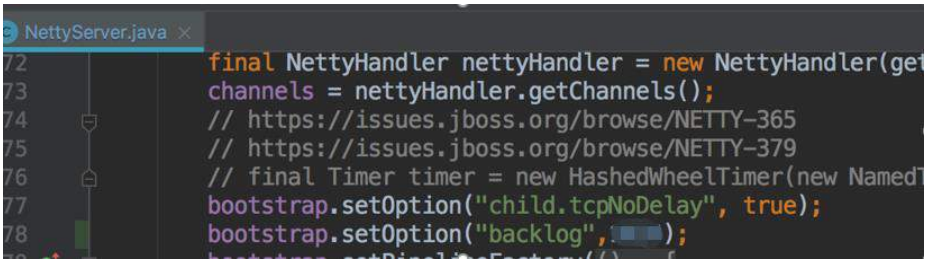
然后看了下机器内核的 somaxconn 也远远超过 50，难道 50 是应用层设置的？

```
$ cat   /proc/sys/net/core/somaxconn
 128
```

Accept queue 的大小取决于：min(backlog, somaxconn)，backlog 是在 socket 创建的时候传入的，somaxconn 是一个内核级别的系统参数。

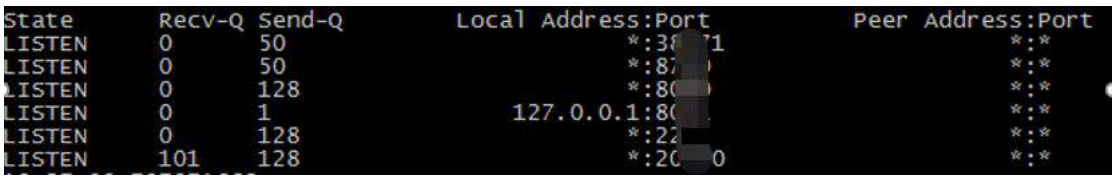
Syn queue 的大小取决于：max(64,tcp_max_syn_backlog)，不同版本的 os 会有些差异。

再研究下 Netty 的默认值，可以发现 Netty3 初始化的时候 backlog 只有 50 个，Netty4 已经默认升到 1024 了。



业务换了新的包，重新发布后发现 accept queue 变成了 128，服务端 syn 被丢弃的问题已经没有了，客户端连接也不再报错。

在应用启动时间，通过 shell 脚本打印队列大小，从图片中可以看到，最大 queue 已经到了 101，所以之前默认的 50 个肯定是要超了。



从这个截图，也可以知道为啥前面提到的 HTTP 请求没有 syn 丢包了。因为 Tomcat 已经设置了 backlog 为 128，而且 HTTP 的连接是 lazy 的。但是，我们对 TCP 连接的初始化并不是 lazy 的，所以在高并发的场景下会出现 accept queue 满的情况。

十、调整 backlog 到底有多大效果？

针对这个问题，我们还专门做了个试验了下，从实验结果看调整带来的效果还是比较明显的。

服务端：1 台 8C 的物理机器
客户端：10 台 4C 的 docker

Backlog	每秒并发建连数	SYN包被丢?
128	3000	无
128	5000	少量丢包
1024	5000	无
1024	10000	无

可以看到，对 8C 的机器 backlog 为 128 的情况下，在每秒 5000 建连的时候就会出现 syn 丢包。在调整到 1024 之后，即使 10000 也没有任何问题。当然，这里提醒下，不要盲目的调整到很高的值，是否可以调整到这么高，还要结合各自服务器的配置以及业务场景。