促进软件开发领域知识与创新的传播

架构师 ARCHITECT

推荐文章 | Article

LinkedIn是如何优化Kafka的

如何打造第三方的开放账号体系

专题 | Topic

深入浅出React（四）

大数据查询引擎Apache Calcite

观点 | Opinion

为什么不在生产项目中转向Go

特别专栏 | Column

OPPO自主研发监控系统

InfoQ Queue

2015年10月

OPPO自主研发监控系统：OPPO Monitor Platform

OPPO公司自主研发了一套监控系统OMP，形成了从App请求开始到后端处理过程的完整监控体系。

作为一名Java程序员，我为什么不在生产项目中转向Go

本文将从一名Java程序员的角度，来探讨一下是不是应该在生产中由Java转向Go。包括CTO去思考。

LinkedIn是如何优化Kafka的

LinkedIn的高级工程主管Kartik Paramasivam撰文分享了他们使用和优化Kafka的经验。

如何打造第三方的开放账号体系

InfoQ采访了Havana与云账号的技术负责人钱磊，了解了他们为何决定开放账号体系，如何保证账号系统的安全和稳定性。

Apache Calcite：Hadoop中新型大数据查询引擎

诸多特性，Calcite项目在Hadoop中越来越引入注目，并被众多项目集成。

深入浅出React（四）：虚拟DOM Diff算法解析

理解虚拟DOM运行机制不仅有助于更好的理解React组件的生命周期，而且对于进一步优化React程序也会有很大帮助。

架构师 2015年10月刊

本期主编 龙永昕

流程编辑 丁晓昀

发行人 霍泰稳

联系我们

提供反馈 feedback@cn.infoq.com

商务合作 sales@cn.infoq.com

内容合作 editors@cn.infoq.com

肖文峰 1978年生，清华大学硕士，2014年加入TalkingData任CTO，负责全业务线的研发工作。

大数据的发展趋势和公司的战略部署

半年前，TalkingData 服务器端每日接收的数据量呈现爆炸式增长的趋势，曾一度让我们的技术支撑团队猝不及防。从 2011 年 TalkingData 创立以来，在智能设备覆盖量上，从零到 10 亿用了整整 4 年，但是从 10 亿到 20 亿，只用了不到 8 个月的时间。我们的合作伙伴和客户对数据的理解大大加深，在使用数据的深度和广度上，相比 2014 年都有一个大的跨越。

2 个月前，国务院常务会议通过 《关于促进大数据发展的行动纲要》，从政府层面推动大数据的开放和共享，为大数据的创业创新提供强大的动力，掀起一股新的大数据热潮。

1 个月前，TalkingData 举办“T11 全球移动大数据峰会”，与行业同仁分享 4 年来在移动大数据应用方面的心得与经验。会议盛况空前，实际客流量超过计划的 3 倍。

如果说 2014 年，大家还处于畅谈大数据概念的层面，那么 2015 年，大数据已经进入落地的阶段。以 Talkingdata 为代表的大数据行业内的公司，已经在逐步消化积累的海量数据，将数据成果应用于各个传统的行业。在一些看似很“土”的行业，比如房地产、奢侈品等，大数据已经产生了实际的价值。

TalkingData 是一个以数据为核心的创业公司，其愿景是要用数据去改变企业做决定的方式，同时要用数据去帮助人们了解周围的环境。在未来，我们依然会继续深化我们的核心价值，聚焦在数据和数据的落地，总体说来有以下两点。

一是继续深入研究智能设备的传感器，包括传感器数据的处理算法，提升数据的深度和广度。过去一段时间，经常听到“场景化”的提法，场景化能够帮助开发者更准确的判断用户当前的状态和目的，把应用做得更加

智能——如果知道“什么样的人在什么时刻做什么”的话，就很容易触发一些有针对性的功能、服务和内容，真正做到“应时应人应景”。比如，如果判断出司机是一个开车很暴躁的人，就给他打一个“激进驾驶员”的标签，这是对“什么样的人”的判断；当前是在下雨天，又是在高速公路上，这是对“什么时刻”的判断。当前正在驾驶中，这是对“做什么”的判断。一个激进的驾驶员，在比较危险的时刻正在驾驶中，这时候一些软件就可以提醒驾驶员，是不是减慢速度，听听音乐，缓解一下情绪，避免可能的风险。这里所有的判断，都是需要基于手机的传感器数据处理以后的结果来判断，有加速度传感器、GPS、光感、音量、温度等等。我们越来越发现，这些传感器的数据分析和处理，存在很高的门槛。比如像我们现在基于加速传感器去研究步态判断的算法，我们尝试好几种算法，K 均值、SVM、决策树等等，也包含综合一些算法进行投票的方法，在不同人群和不同机型上测试了很长时间，才算有一些相对满意的成果。我们希望帮开发者把这个脏活累活分担了，开发者基于我们的 SDK 可以轻松判断用户场景，提升用户体验。

二是如何把数据变得像水一样，能够高效流动。目前数据利用中还有较大的阻碍，比如数据量不够大、数据质量不够高、数据源之间很难匹配、敏感数据较难处理等，这些问题需要由专业团队处理。我们希望 TalkingData 能够成为数据的“水库”，不同来源的“水”（源数据）汇聚到 TalkingData 的“水库”，经过各种手段被加工处理，然后根据客户的不同需求提供不同水质和口味的水（面向用户场景的数据产品），有昂贵的西藏冰川 5100，也有普通的农夫山泉，还有便宜的批量桶装水。“水库”提供数据相关的公共服务，比如清洗、匹配、合规、脱敏、估值、交易等，降低数据交换的门槛。我们也会引入更多的数据合作伙伴，做更多的定向数据服务，探索更新的数据利用模式。我们会把战线推进到客户的现场去，和客户并肩解决问题。当然，研发团队架构也已经有了对应的调整。

经济的寒冬将迫使各行业利用大数据来优化运营、改善管理、降低成本、提升竞争力——这是大数据的春天。TalkingData 会和伙伴们携手并肩，直面挑战，共筑未来。

作为一名Java程序员，我为什么不在生产项目中转向Go

作者 丁雪丰

郑重声明：本文并不是来探讨 Go 或者 Java 谁是更好的语言，每种语言都有自己的设计哲学和适用场景，今天主要是在探讨实际工程中的选择和权衡的问题，所以请不要上纲上线。

自 Google 在 2009 年发布 Go 语言的第一个正式版之后，这门语言就以出色的语言特性受到大家的追捧，尤其是在需要高并发的场景下，大家都会想到是不是该用 Go。随后，在国内涌现出了一批以七牛为代表的使用 Go 作为主要语言的团队，而许式伟大神本人也在各种场合下极力推动 Go 在国内的发展，于是在这种大环境下，中国的 Go 开发者群体逐渐超越了其他地区。

那么问题来了，业余时间好学是一回事，真正要将一个新东西运用到生产中则是另一回事。JavaScript 的开发者可以义无反顾地选择 Node.js，但是对于 Java 开发者来说，在下一个大项目里究竟是该选择 Go，还是 Java 呢？

语言本身

首先，需要说明一下，作为一个技术决策者，在进行技术选型时并不能单方面地根据语言本身的特点直接下结论。实际情况下，大多数人会使用一系列的框架、库及工具，简而言之就是会考虑很多周边生态环境的因素，同时还要结合公司的特点、各种历史问题和实际客观因素等等一系列的考虑点综合下来才能完成决策。所以，接下来我们先从语言开始，一步一步来分析下在你的项目中选择 Go 是否合适。

Go 在高并发编程方面无疑是出众的，通过 goroutine 从语言层面支持了协程，这是 Java 等语言所无法比拟的，这也是大多数人在面对高并发场景选择 Go 的重要原因之一。虽然 Java 有 Kilim 之类的框架，但没有语言层的支持始终稍逊一筹。

除此之外，Go 的其他语法也很有趣，比如多返回值，在一定程度上为开发者带来了一定的便利性。试想，为了返回两到三个值，不得不封装一个对象，或者抹去业务名称使用 Map、List 等集合类，高级一点用 Apache 的 Pair 和 Triple，虽然可行，但始终不如 Go 的实现来得优雅。在此之上，Go 也统一了异常的返回方式，不用再去纠结是通过抛异常还是错误码来判断是否成功，多返回值的最后一个是 Error 就行了。

Go 在语言的原生类型中支持了常用的一些结构，比如 map 和 slice，而其他语言中它们更多是存在于库中，这也体现了这门语言是从实践角度出发的特点，既然人人都需要，为什么不在语言层面支持它呢。函数作为一等公民出现在了 Go 语言里，不过 Java 在最近的 Java 8 中也有了 Lambda 表达式，也算是有进步了。

其他的一些特性，则属于锦上添花型的，比如不定参数，早在 2004 年的 Java 1.5 中就对 varargs 有支持了；多重赋值在 Ruby 中也有出现，但除了多返回值赋值，以及让你在变量交换值时少写一个中间变量，让代码更美观一些之外，其他的作用着实不是怎么明显。

说了这么多 Go 的优点，当然它也有一些问题，比如 GC，说到它，Java 不得不露出洁白的牙齿，虽然在大堆 GC 上 G1 还有些不尽如人意，但 Java 的 GC 已经发展了很多年，各种策略也比较成熟，CMS 或 G1 足以应付大多数场景，实在有要求还能用 Azul Zing JVM。不过从最新的 Go 1.5 的消息来看，Go 的 GC 实现有了很大地提升，顺便一提的是 GOMAXPROCS 默认也从 1 变成了 CPU 核数，看来官方对 Go 在多核的利用方面更有信心了。

许世伟在《Go 语言编程》的前言中预言未来 10 年，Go 会取代 Java，位居编程榜之首，当时是 2012 年，为了看看 2009 年 TIOBE 年度编程语言如今的排名，笔者在撰写本文时特意去 TIOBE 看了下，最近的 2015 年 8 月排行榜，Java 以 19.274% 位居榜首，Go 已经跌出了前 50，这不禁让人有些意外。

但总体上来说，笔者认为 Go 在语言层面的表现还是相当出色的，解决了一些编程中的痛点，学习曲线也能够接受，特别是对于那些有 C/C++ 背景的人，会感觉十分亲切。

工程问题

一个人写代码时可以很随性，想怎么写就怎么写，但当一个人变成一个团队后，这种随性或者说随便就会带来很多问题，于是就诞生了编码规范这玩意儿，大厂基本都有自己的编码规范，比如 Google 就有针对不下十种编程语言的规范。团队内约定一套编码规范能够很大程度上地确保代码的风格，降低阅读沟通的成本。Go 内置了一套编码规范，违反了该规范代码就无法编译通过，可以说只要你是写 Go 的，那你的代码就不会太难看，当然 Go 也没有把所有东西就强制死，还有一些推荐的规范可以通过 gofmt 进行格式化，但这步不是必须的。

虽然 Go 自己解决了这个问题，但并不能说 Java 在这方面是空白，Java 发展至今周边工具无数，并不缺成熟的代码静态分析工具，比如 CheckStyle、PMD 和 FindBugs，它们不仅能扫描编码规范的问题，甚至还能扫描代码中潜在的问题并给出解决方案，并且使用方便，在 Java 开发者社区中有很高地接受度，应该说大多数靠谱地开发者都会使用这些工具。除此之外，一些大厂也有自己的强制手段，比如百度

内部也有很多语言的编码规范，而且大部分情况下如果没有通过编码规范的扫描，你是无法提交代码的；还有一些公司会在持续集成过程中加入代码扫描，有 FindBugs 高优先级的问题时必须修复才能进入下一个阶段。所以说 Go 在这个问题上的优势并不明显，或者说在一个成熟的环境下，这只是合格而已。

这里需要强调笔者的一个观点：

Go 在语言本身和发行包中融入了很多最佳实践，正是这些前人的经验才让它看起来如此优秀。拿这么个海陆空混编特种部队去和 Java、C、Ruby 这些语言本身做对比，显得不太公平，所以本文在考虑问题时都会结合语言及其生态圈中的成员，毕竟这才更接近真实的情况。

Go 本身对项目结构有一套约定，代码放哪里，测试文件如何命名，编译打包后的结果输出到哪个目录，甚至还有 go cover 这种统计测试覆盖率的命令行，开发者不用在这些问题上太过纠结，再一次体现了 Go 注重工程实践的特点。回过头来，Java 方面，Maven、Gradle 都是注重于工程生命周期管理的工具，而且 Maven 更是历史悠久，被广泛用于各种项目之中。以 Maven 为例，不仅能够实现上述所有功能，还有很强的插件扩展能力，这里需要的只是一次性维护好 pom.xml 文件就行了，由于 Maven 的使用群很大，网上有大量的范例，甚至还有很多生成工程的工具和模板，所以使用成本并不高。

这里还要衍生出一个话题，就是依赖管理，在开发代码时，势必需要依赖很多外部的东西，Go 可以直接 import 远程的内容，这个特性很有创意，但并不能很好地解决版本的问题，在 Maven 或 Gradle 里，我们可以直接指定各个依赖项甚至是插件的版本，工具会自动从仓库中下载它们。如果需要同时在同一个系统的不同模块里依赖同一个库的不同版本，我们还能够通过 OSGi 这种略显复杂的手段来实现，在模块化方面，Jagsaw 虽然被一延再延，但估计有望纳入 Java 9，这个特性也会解决不少问题。而根据 Golang 实践群中大家的讨论，似乎 godep、gb 和 gvt 都不尽如人意，在这点上看来 Go 还有一段路要走。

综上所述，Go 在工程方面的确有不少亮点，吸纳了很多最佳实践，甚至可以说用 Go 之后更容易写出规范的代码，有好的项目结构，但与生态圈完备的 Java 相比，Go 并不占优势，因为最终代码的质量还是由人决定的，双方都不缺好的工具，所以这方面的特点并不能影响技术选型的决策。

开发实践

Talk is cheap. Show me the code.

下面进入编码环节，先从 Go 引以为傲的并发开始，《Go 语言编程》的前言中有这样一段代码：

```
001   func run(arg string) {
002     // ...
003 }
004
005 func main() {
006     go run(“test”)
007     ...
008 }
```

书中与之对比的 Java 代码有 12 行，而且还是线程，不是协程，对比很明显，但那是在 2012 年的时候，时至今日，Java 已经发展到了 Java 8，3 年了，看看如今的 Java 代码会是什么样的：


```
001   public class ThreadDemo {
002     public static void main(String[]
      args) {
003         String str = “test”; // 为了和
      原先的Java版本对照，说明能传参进入线程
      内，在外声明了一个字符串，其实可以直接写
      在Lambda里
004         new Thread(() -> { /* do sth.
      with str */ }).start();
005     }
006 }
```

不是协程仍是硬伤，但有了 Lambda 表达式，代码短了不少。不过话又说回来，这样的比较并没有太多意义，所以各位 Go 粉也不用站出来说 Go 也支持闭包，Go 的版本也能精简。我们比的不是谁写的短，在 Java 实践中，大多数时候大家会选择线程池，而不是自己 new 一个 Thread 对象，Doug Lea 大神的 Java 并发包非常的好用，而且很靠谱。另外，并发中处理的内容才是关键，新启一个线程或者协程才是万里长城的第一步，如果其中的业务逻辑有 10 个分支，还要多次访问数据库并调用远程服务，那无论用什么语言都白搭。所以在业务逻辑复杂的情况下，语言的差异并不会太明显，至少在 Java 和 Go 的对比下不明显，至于其他更高阶、表达力更强的语言（比如 Common Lisp），大家就要拼智商了。

还有一些情况中，由于客观因素制约，完全就无法使用 Go，比如现在如火如荼的互联网金融系统里，与银行对接的系统几乎没有选择，都是 Java 实现的，因为有的银行只会给 Jar 包啊……给 Jar 包啊……Jar 包啊……如果是个 so 文件，也许还能用 cgo 应付一下，面对一个 Jar 你让 Go 该何去何从？

抛开这些让人心烦的问题，让我们再来看看现在比较常见的如何实现 REST 服务。说到这里，就一定要祭出国人出品的 Beego 框架。一个最简单的 REST 服务可以是这样的：

```
001 package main
002
003 import (
004     “github.com/astaxie/beego”
005 )
006
007 type MainController struct {
008     beego.Controller
009 }
010
011 func (this *MainController) Get() {
012     this.Ctx.WriteString(“hello
      world!”)
013 }
014
015 func main() {
016     beego.Router(“/”,
      &MainController{})
017     beego.Run()
018 }
```

既然 Go 方面，我们使用了一套框架，那么 Java 方面，我们一样也选择一个成熟的框架，Spring 在 Java EE 方面基本可以算是事实标准，而 Spring Boot 更是大大提升了 Spring 项目的开发效率，看看同样实现一个 REST 服务，在 SpringBoot 里是怎么做的。

首先，到 start.spring.io 根据需要生成项目骨架（其实完全可以方便地自己通过 Maven 手工配置依赖或者是用 CLI 工具来创建），为了后续的演示，这里我会选上“Web”、“Actuator”和“Remote Shell”，其实就是多了两个 Maven 的依赖，下文运维部分会提到，然后随便找个顺手的 IDE 打开工程，敲入如下代码就行了（import、包和类定义的部分基本都是 IDE 生成的）。

运行下面这段代码会自动启动内置 Tomcat 容器，访问 http://localhost:8080/ 就能看到输出了。因为其实就是 Spring，所以可以毫无压力地与其他各种框架设施组合，也没有太多学习成本。

```
001   package demo;
002
003 import org.springframework.boot.
      SpringApplication;
004 import org.springframework.boot.
      autoconfigure.SpringBootApplication;
005 import org.springframework.web.bind.
      annotation.RequestMapping;
006 import org.springframework.web.bind.
      annotation.RestController;
007
008 @SpringBootApplication
009 @RestController
010 public class DemoApplication {
011     @RequestMapping(“/”)
012     public String sayHello() {
013         return “hello world!”;
014     }
015
016     public static void main(String[]
      args) {
017         SpringApplication.
      run(DemoApplication.class, args);
018     }
019 }
```

可见两者在实现 REST 服务方面，并没有太大的差别，加之上文提到的业务逻辑问题，只要运用恰当的工具，两种语言之间并不会产生质的差异。

Beego 中的 ORM 支持 MySQL、PostgreSQL 和 Sqlite3，而在 Java 里 Hibernate 和 myBatis 这样的 ORM 工具几乎能通吃大多数常见的关系型数据库，且相当成熟，社区配备了各种自动生成工具来简化使用，行业里还有 JPA 这样的公认标准。纵观 Go 的 ORM 工具，大家还是在探讨，究竟哪个才好用呢？切到 NoSQL 方面，双方都有大量的驱动可以使用，比如 MongoDB 和 Redis 都有详尽的驱动列表，MongoDB 还没有官方驱动，但有社区维护的 mgo，算是打成平手吧。再大一点，像用到 Hadoop、Spark 和 Storm 的场景下，似乎 Java 的出镜率更高，或者是直接通过 Streaming 方式就解决了，此处也就不再展开了。

虽然说了这么多问题，但如果真的遇到了大流量、高并发的场景，需要从头开始开发用来处理这些问题的基础设施时，Go 还是不错的选择。比如，七牛这样的云服务提供商，又或者是 BFE（Baidu Front End，号称可能是全世界流量最大的 Go 语言集群 ，在 2015 年的 Velocity 大会上留下了它的身影——图 1 和图 2）这样的硬货，请不要纠结。

运维

写完代码只是万里长征的一小步，后面还有一大堆的事情等着你去解决，比如怎么把写完的代码编译、打包、发布上线。编译打包就不说了，Go 的命令行工具 go build 就能直接把你的代码连同它的所有依赖一起打成一个可执行文件。至于部署，大家都称赞 Go 的部署没有依赖（除了对 glibc 的版本有要求，不考虑需要 cgo 的情况），直接把可执行文件往那里一扔就好了，非常方便。Go 内置了强大的 HTTP 支持，不需要其他 Web 服务器来做支撑就能获得不错的性能。

再来看看 Java，按照常理，一般都会使用 Maven 或者 Gradle 来处理编译、打包，甚至是发布，仍旧以 Maven 为例，mvn package 就能完成编译和打包。可以选择 Jar 包，如果是 Web 项目部署到容器里的话可以是 War 包，也可以将各种资源打包到一起放到压缩包（zip、tar 等等）里，这个步骤并不复杂。

接下来的部署环节，大家就有话要说了，“Write Once, Run Anywhere”这曾是 Java 的宣传语，但正是这句话一直被大家诟病，其实如果代码中不使用平台特定的内容（比如避免绑定在 WebLogic 上），不使用某个特定版本 JDK 的内部类（比如 com.sun 里的东西，这种做法本来就不推荐），Java 的代码还是能够做到编译后在任何地方都能运行的，事实上现在绝大部分情况下，大家也都是这么做的，看看广大的 Java 库都是发布 Jar 到 Maven 仓库的，也没谁让你直接拉源码来编译。在不同的环境下，只需要部署了对应的 JDK 就好了（一般放到装机模板里，或者直接拿安装包部署一下就好了），至于是什么操作系统其实并不重要。

延续上文 REST 服务的例子，Java 的 Web 项目一般都会部署到容器里，比如 Tomcat 或者 Jetty，当然也有用商业容器的（很多银行就是用的 WebLogic），所以大家就都认为部署 Java 程序需要先有容器，这其实是几年前的事情了，后来刮起了一股内嵌容器的风潮，Tomcat 和 Jetty 都可以嵌入到你的程序里，再也不用为有没有容器而烦恼了。Spring Boot 索性把这件事变得更简单了，mvn package 后，一句话就能搞定内置 Tomcat 的启动、完成各种部署，然后一切就变成下面这样（假设最后生成的 Jar 包名为 demo.jar。

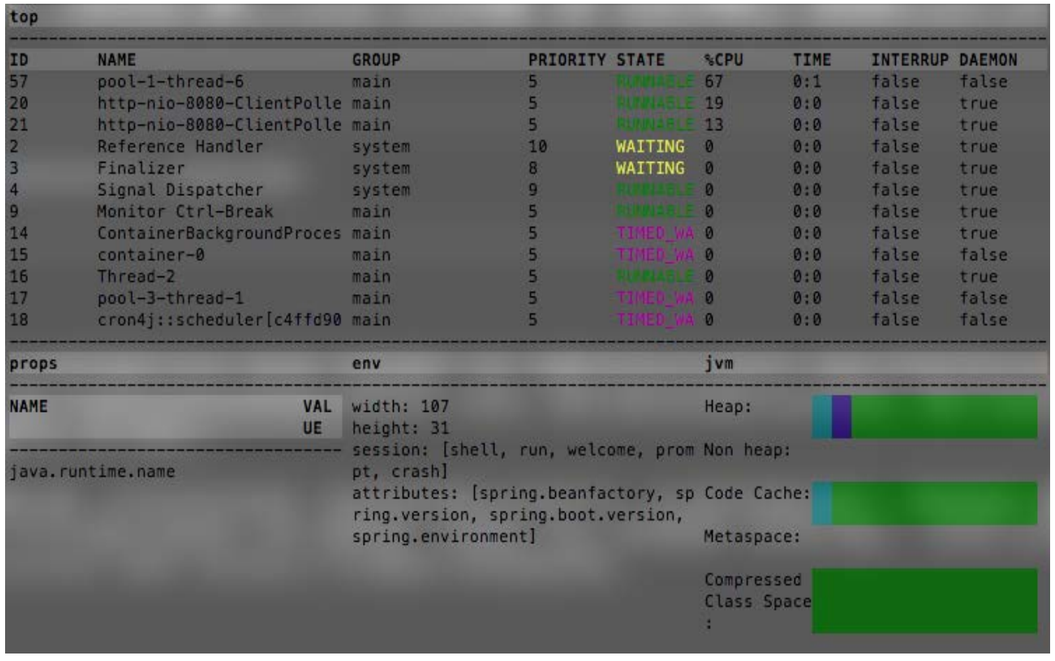
```
java -jar demo.jar
```

在 Spring Boot 1.3 里，还能通过调整 Maven Plugin 的配置，让 Jar 可以直接执行（不要小看这么一个变化，它可以大大提升可运维性）：

```
./demo.jar
```

所以说 Java 程序难部署其实也是历史，现在的 Java 程序部署早已是另一番光景。两者的编译、打包、部署环节完全可以打成平手。笔者认为有些方面 Java 反而更胜一筹，比如 Java 基本就不用操心交叉编译的问题；Go 的库在发布时推荐直接发布源码而非二进制包，遇到天朝特有的网络无法访问的情况，编译个东西还要自备梯子……至于和 Nginx 等等的配合，更是大家都很方便，就不再赘述了。

完成了部署，接下来的日志和监控，都是很常规的问题，日志各自有对应的库，而监控都是依赖专业的监控平台，自己做好信息输出就好了，请容我再秀一下 Spring Boot 的 RemoteShell 终端监控，除了常规的 HTTP 方式输出 JSON 信息（自带了健康检查、仪表数据、Dump、请求跟踪等一系列 REST 输出），还自带了这么个类似 top 的高大上的玩意儿，ssh -p 2000 user@localhost 后执行 dashboard 可以看到这个实时更新的界面。



总结

说了这么多，来总结下全文的观点——虽然 Go 在语言上表现的很出色，也融入了很多最佳实践，但是结合多方考虑，在很多情况下它并不会比 Java 带来更多价值，甚至还不一定能做的比 Java 好，因此作为一个 Java 程序员，我不会在自己的生产项目中转向 Go。

此外，除了本文重点讨论的那些问题，还有更现实的问题摆在那里，比如团队转型成本和招聘的成本，千万不要小看招聘，对于管理者而言，招聘也是工作中的重要内容，试想一下，是招个有经验的 Go 程序员容易，还是招一个有经验的 Java 程序员容易，就算能招到一个会 Go 的正式员工，你能招到一个会 Go 的外包么，特别是在团队急需补充新鲜血液时，结果是显而易见的。

但这一切都不妨碍大家来学习 Go，本文开头就已经表达过这一观点，业余时间学习 Go 和在生产项目中不用 Go 并不冲突，Go 还是有很多值得学习和借鉴的地方，而且谁也说不准哪天你就真遇上了适合用 Go 的项目呢。

最后，特别感谢谢孟军与李道兵在本文写作过程中与笔者的各种思维碰撞与交流。

LinkedIn是如何优化Kafka的

作者 张卫滨

在 LinkedIn 的数据基础设施中，Kafka 是核心支柱之一。来自 LinkedIn 的工程师曾经就 Kafka 写过一系列的专题文章，包括它的现状和未来、如何规模化运行、如何适应 LinkedIn 的开源策略以及如何适应整体的技术栈等。近日，来自 LinkedIn 的高级工程主管 Kartik Paramasivam 撰文分享了他们使用和优化 Kafka 的经验。

LinkedIn 在 2011 年 7 月开始大规模使用 Kafka，当时 Kafka 每天大约处理 10 亿条消息，这一数据在 2012 年达到了每天 200 亿条，而到了 2013 年 7 月，每天处理的消息达到了 2000 亿条。在几个月前，他们的最新记录是每天利用 Kafka 处理的消息超过 1 万亿条，在峰值时每秒钟会发布超过 450 万条消息，每周处理的信息是 1.34 PB。每条消息平均会被 4 个应用处理。在过去的四年中，实现了 1200 倍的增长。

随着规模的不断扩大，LinkedIn 更加关注于 Kafka 的可靠性、成本、安全性、可用性以及其他的基础指标。在这个过程中，LinkedIn 的技术团队在多个特性和领域都进行了有意义的探索。

LinkedIn 在 Kafka 上的主要关注领域如下所述。

配额（Quotas）

在 LinkedIn，不同的应用使用同一个 Kafka 集

群，所以如果某个应用滥用 Kafka 的话，将会对共享集群的其他应用带来性能和 SLA 上的负面影响。有些合理的使用场景有可能也会带来很坏的影响，比如如果要重新处理整个数据库的所有数据的话，那数据库中的所有记录会迅速推送到 Kafka 上，即便 Kafka 性能很高，也会很容易地造成网络饱和和磁盘冲击。

Kartik Paramasivam 绘图展现了不同的应用是如何共享 Kafka Broker 的（图 1）。

为了解决这个问题，LinkedIn 的团队研发了一项特性，如果每秒钟的字节数超过了一个阈值，就会降低这些 Producer 和 Consumer 的速度。对于大多数的应用来讲，这个默认的阈值都是可行的。但是有些用户会要求更高的带宽，于是他们引入了白名单机制，白名单中的用户能够使用更高数量的带宽。这种配置的变化不会对 Kafka Broker 的稳定性产生影响。这项特性运行良好，在下一版本的 Kafka 发布版中，所有的人就都能使用该特性了。

开发新的 Consumer

目前的 Kafka Consumer 客户端依赖于 ZooKeeper，这种依赖会产生一些大家所熟知的问题，包括 ZooKeeper 的使用缺乏安全性以及 Consumer 实例之间可能会出现的脑裂现象（split brain）。因此，LinkedIn 与 Confluent 以及其他的开源社区合作开发了一个新的 Consumer。这个新的 Consumer 只依赖于 Kafka Broker，不再依赖于 ZooKeeper。这是一项很复杂的特性，因此需要很长的时间才能完全应用于生产环境中。

在 Kafka 中，目前有两个不同类型的 Consumer。如果 Consumer 希望完全控制使用哪个分区上的 Topic 的话，就要使用低级别的 Consumer。在高级别的 Consumer 中，Kafka 客户端会自动计算如何在 Consumer 实例之间分配 Topic 分区。这里的问题在于，如果使用低级别 Consumer 的话，会有许多的基本任务要去完成，比如错误处理、重试等等，并且无法使用高级别 Consumer 中的一些特性。在 LinkedIn 这个新的 Consumer 中，对低级别和高级别的 Consumer 进行了调和。

可靠性和可用性的提升

按照 LinkedIn 这样的规模，如果 Kafka 的新

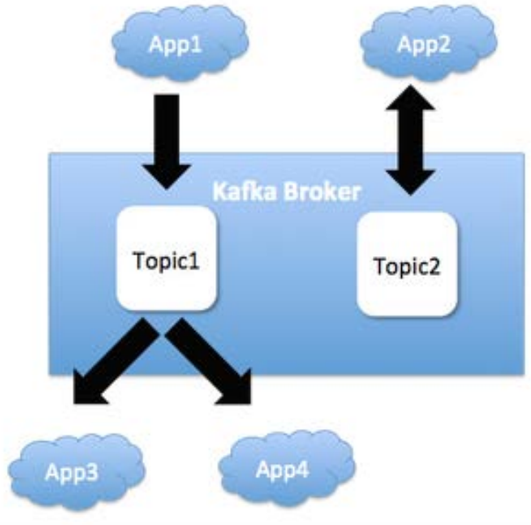


图 1

版本中有什么重要缺陷的话，就会对可靠性产生很大的影响。因此，LinkedIn 技术团队一项很重要的任务就是发现和修正缺陷。他们在可靠性方面所做的增强包括：

Mirror Maker 无损的数据传输：Mirror Maker 是 Kafka 的一个组件，用来实现 Kafka 集群和 Kafka Topic 之间的数据转移。LinkedIn 广泛使用了这项技术，但是它在设计的时候存在一个缺陷，在传输时可能会丢失数据，尤其是在集群升级或机器重启的时候。为了保证所有的消息都能正常传输，他们修改了设计，能够确保只有消息成功到达目标 Topic 时，才会认为已经完全消费掉了。

副本的延迟监控：所有发布到 Kafka 上的消息都会复制副本，以提高持久性。当副本无法“跟上”主版本（master）的话，就认为这个副本处于非健康的状态。在这里，“跟上”的标准指的是配置好的字节数延迟。这里的问题在于，如果发送内容很大的消息或消息数量不断增长的话，那么延迟可能会增加，那么系统就会认为副本是非健康的。为了解决这个问题，LinkedIn 将副本延迟的规则修改为基于时间进行判断。

实现新的 Producer：LinkedIn 为 Kafka 实现了新的 Producer，这个新的 Producer 允许将消息实现为管道（pipeline），以提升性能。目前该功能尚有部分缺陷，正在处于修复之中。

删除 Topic：作为如此成熟的产品，Kafka 在删除 Topic 的时候，会出现难以预料的后果或集群不稳定性，这一点颇令人惊讶。在几个月前，LinkedIn 对其进行了广泛地测试并修改了很多缺陷。到 Kafka 的下一个主版本时，就能安全地删除 Topic 了。

安全性

在 Kafka 中，这是参与者最多的特性之一，众多的公司互相协作来解决这一问题。其成果就是加密、认证和权限等功能将会添加到 Kafka 中，在 LinkedIn，预期在 2015 年能使用加密功能，在 2016 年能使用其他的安全特性。

Kafka 监控框架

LinkedIn 最近正在致力于以一种标准的方式监控 Kafka 集群，他们的想法是运行一组测试应用，这些应用会发布和消费 Kafka Topic 数据，从而验证基本的功能（顺序、保证送达和数据完整性等）以及端到端发布和消费消息的延时。除此之外，这个框架还可以验证 Kafka 新版本是否可以用于生产环境，能够确保新版本的 Kafka Broker 不会破坏已有的客户端。

故障测试

当拿到新的 Kafka 开源版本后，LinkedIn 会运行一些故障测试，从而验证发生失败时 Kafka 新版本的质量。针对这项任务，LinkedIn 研发了名为 Simoorg 的故障引导框架，它会产生一些低级别的机器故障，如磁盘写失败、关机、杀进程等等。

应用延迟监控

Consumer 开发了名为 Burrow 的工具，能够监控 Consumer 消费消息的延迟，从而监控应用的健康状况。

保持 Kafka 集群平衡

LinkedIn 在如下几个维度保证了集群的平衡：

感知机柜：在进行平衡时，很重要的一点是 Kafka 分区的主版本与副本不要放到同一个数据中心机柜上。如果不这样做的话，一旦出现机柜故障，将会导致所有的分区不可用。

确保 Topic 的分区公平地分发到 Broker 上：在为 Kafka 发布和消费消息确定了配额后，这项功能变得尤为重要。相对于将 Topic 的分区发布到同一个 Broker 节点上，如果 Topic 的分区能够均衡地分发到多个 Broker 上，那么相当的它有了更多的带宽。

确保集群节点的磁盘和网络容量不会被耗尽：如果几个 Topic 的大量分区集中到了集群上的少数几个节点上，那么很容易出现磁盘或网络容量耗尽的情况。

在 LinkedIn，目前维护站点可靠性的工程师（Site Reliability Enginee，SRE）通过定期转移分区确保集群的平衡。在分区放置和重平衡方面，他们已经做了一些原始设计和原型实现，希望能够让系统更加智能。

在其他的数据系统中，将 Kafka 作为核心的组成部分。

在 LinkedIn，使用 Espresso 作为 NoSQL 数据库，目前他们正在将 Kafka 作为 Espresso 的备份机制。这将 Kafka 放到了站点延迟敏感数据路径的关键部分，同时还需要保证更高的消息传送可靠性。目前，他们做了很多的性能优化，保证消息传输的低延迟，并且不会影响消息传递的可靠性。

Kafka 还会用于异步上传数据到 Venice 之中。除此之外，Kafka 是 Apache Samza 实时流处理的一个重要事件源，同时 Samza 还使用 Kafka 作为持久化存储，保存应用的状态。在这个过程中，LinkedIn 修改了一些重要的缺陷，并增强了 Kafka 的日志压缩特性。

LinkedIn 的 Kafka 生态系统

除了 Apache Kafka Broker、客户端以及 Mirror Maker 组件之外，LinkedIn 还有一些内部服务，实现通用的消息功能：

支持非 Java 客户端：在 LinkedIn，会有一些非 Java 应用会用到 Kafka 的 REST 接口，去年他们重新设计了 Kafka 的 REST 服务，因为原始的设计中并不能保证消息的送达。

消息的模式：在 LinkedIn，有一个成熟的“模式（schema）注册服务”，当应用发送消息到 Kafka 中的时候，LinkedIn Kafka 客户端会根据消息注册一个模式（如果还没有注册过的话）。这个模式将会自动在 Consumer 端用于消息的反序列化。

成本计算：为了统计各个应用对 Kafka 的使用成本，LinkedIn 使用了一个 Kafka 审计 Topic。LinkedIn 客户端会自动将使用情况发送到这个 Topic 上，供 Kafka 审计服务读取并记录使用情况，便于后续的分析。

审计系统：LinkedIn 的离线报告 job 会反映每小时和每天的事件情况，而事件从源 Kafka Topic/ 集群 / 数据中心，到最后的 HDFS 存储是需要时间的。因此，Hadoop job 需要有一种机制，保证某个时间窗口能够获得所有的事件。LinkedIn Kafka 客户端会生成它们所发布和消费的消息数量。审计服务会记录这个信息，Hadoop 以及其他的服务可以通过 REST 接口获取这一信息。

支持内容较大的消息：在 LinkedIn，将消息的大小限定为 1MB，但是有些场景下，无法满足这一限制。如果消息的发布方和使用方是同一个应用的话，一般会将消息拆分为片段来处理。对于其他的应用，建议消息不要超过 1MB。如果实在无法满足该规则的话，消息的发送和消费方就需要使用一些通用的 API 来分割和组装消息片段，而在 LinkedIn 的客户端 SDK 中，他们实现了一种特性，能够自动将一条大的信息进行分割和重组。

目前，越来越多的国内外公司在使用 Kafka，如 Yahoo!、Twitter、Netflix 和 Uber 等，所涉及的功能从数据分析到流处理不一而足，希望 LinkedIn 的经验也能够给其他公司一些借鉴。

专访阿里钱磊：如何打造第三方的开放账号体系

作者 徐川

如今，使用第三方的云服务对移动开发者来说并不是难以接受的事情，但如果连最核心的账号系统也交给第三方呢？在考虑有没有人敢这么做之前，阿里的百川项目已经将它做出来并对外开放了。

云账号（OpenAccount）是阿里百川项目开放出来的一个业务组件，今年 7 月正式上线，阿里百川是阿里巴巴集团无线开放平台，为移动开发者（涵盖移动创业者）提供快速搭建 APP、加速 APP 商业化、提升用户体验的解决方案。

从云账号的文档可以看到，目前它提供新账号注册、开发者已有账号体系集成、第三方社交平台开放账号体系等功能，可以在开发者已有的账号系统进行双向同步和备份操作，并且提供可定制 UI 的 iOS、Android 客户端集成的 SDK。

据 InfoQ 进一步了解，这套系统是由阿里会员系统 Havana 产品化并开放而来，InfoQ 采访了 Havana 与云账号的技术负责人钱磊，了解了他们为何决定开放账号体系，如何保证账号系统的安全和稳定性，以及后台的架构和性能优化等相关问题。

受访嘉宾介绍：

钱磊，阿里巴巴资深技术专家，在阿里负责过纯技术的产品研发，产出 Shy3、Dubbo 等优秀的技术产品；其中 Dubbo 产品已经开源。带领

过业务平台的搭建，历经三年打通阿里巴巴全集团的会员体系，阿里巴巴会员体系目前已经承载了淘系几亿级的用户体量，经历了多次双十一的稳定考验。近期负责阿里巴巴百川移动端开放，为移动端创业者搭建集技术、产品、商业为一体的创业孵化平台。

InfoQ：请介绍一下云账号系统的研发经历，为了将它开放出来做了哪些工作？

钱磊：谈这个问题之前，我说下两个背景。首先是百川会放出哪些能力，原则是什么。我们的思考是把阿里做的好 的东西产品化出来让开发者使用，通过百川平台让开发者使用到业界第一流的系统。移动端应用细节很重要，有一流的工程师团队在为你服务，可以放心的把专业的 事交给专业的人，开发者可以专注于自己的核心业务逻辑，把场景做透。

我们已经释放出的云账户、短信、TAE、多媒体云、OpenIM 等产品都是在内部有过长时间沉淀，产品化后供开发者使用。

然后，说到云账号不得不提阿里巴巴的会员体系 Havana，我本人也是会员体系的负责人。最早在整个集团有多套账户体系，淘宝、支付宝各一套、 B2B 有两套，阿里云有一套，一些并购过来的业务，也有自己的账户，头绪很多；多套账户体系对用户造成很多麻烦，要记多套账户名和密码，有时还要绑来绑 去；账户不通很多基础业务系统很难复用，重复建设很严重。11 年开始经过三年的努力将所有的账户系统终于统一到了 Havana，过程中产品和技术层面都做 了大量的工作。Havana 支持了全集团的会员各种业务，从 PC 到移动端、从国内用户到海外用户，整个产品链路非常完整。经历过多年双 11 大促的考验，稳 定性也有保障。

云账号是在 Havana 系统的基础上产品化而来的，然而对内和对外场景上的差异还是很大的，开放给第三方使用也做了大量的改造。例如模型简化、多租户设计、支持定制化、安全方案等等。

InfoQ：对于账号系统，安全性是重中之重，Havana 和云账号在账号安全上做了哪些工作？

钱磊：这个问题提的非常好，移动端应用的安全比 Web 应用难做很多。原因是 Web 应用的客户端是浏览器，能 力不强，以做展现为主，逻辑和数据也都少；体验不太好做，但服务端重，防控的点比较集中；而移动端应用的情况就完全不同了，用户体验可以做极致，数据和逻 辑就很自然的在端上变的重了，所以安全要从端到云做通盘考虑，很考验功力和积累；账户无疑是黑产最想攻击的点，用户在应用内有大量的隐私信息和数据资产， 账户是获取这些资产的钥匙。

账户安全是个整体，有两个方面。一方面是基础安全，移动端应用的终端安全、通讯安全、服务端安全我们都做了大量的工作，使用了阿里集团众多基础 安全产品。例如终端的安全保障就有设备指纹、人机识别、反跟踪调试、代码混淆等方面的设计。通讯安全在性能和安全中我们做了很多取舍，最终选择了使用私有协议，实现会话级的安全通道。另一方面是业务安全，各类账户操作的安全和风控策略上我们都有大量的积累。例如在垃圾注册、防刷库等，我们有长期的攻防经 验，沉淀了很多数据和策略。

InfoQ：云账号的后台架构是怎么样的？有哪些关键系统？如何保证稳定性？

钱磊：整个后台其实分了好几层，处理着不同的问题。大致上有网络接入层、网关层、业务逻辑层、存储层，网络 接入层是复用了手淘的体系；网关层有流控、权限、多租户隔离、灾备等设计；业务层对外的强依赖很少，基本上只会依赖存储，非关键依赖都有降级策略。稳定性 还受益于阿里巴巴整体的技术体系，从网络架构到中间件，到线上监控和运维系统，无论是架构上的稳定性，还是线上运维能力和经验，都是业界一流的。

InfoQ：你们在后台的性能优化上做了哪些工作？

钱磊：后台性能方面有三个关键因素，第一是中间件的选择，受益于阿里强大的中间件体系，我们使用的 HSF、 TDDL、Tair 等，都是业界一流的中间件系统，性能和稳定性上都做的非常好；第二是设计上的考量，其中很多是细节，例如登录场景，我们对账户名做了反 向索引，账密登录只需要访问两次缓存，效率非常高。例如在写请求中状态一致性的处理，我们采用的是同步更新 DB，异步清除缓存，以确保写请求的 RT 是稳定 的，不受缓存设计变化的影响；第三是人的因素，做云账号的团队源于做 Havana 的同学，处理分布式和高并发问题是我们的强项。

InfoQ：用户对于开放账号体系有哪些定制化需求？云账号有什么开发计划？

钱磊：目前看到的定制化需求主要是在功能和 UI 层面，例如用哪几种 SNS 登录方式，登录界面的颜色和文案等，这些我们都已经有支持。

从整体设计上，我们还会提供更多的定制能力，从 功能和 UI 到 业务流程 再到 数据模型，都会逐步的释放出来。

InfoQ 采访视频

深入浅出React（四）：虚拟DOM Diff算法解析

作者 王沛

React 中最神奇的部分莫过于虚拟 DOM，以及其高效的 Diff 算法。这让我们可以无需担心性能问题而”毫无顾忌”的随时“刷新”整个页面，由虚拟 DOM 来确保只对界面上真正变化的部分进行实际的 DOM 操作。React 在这一部分已经做到足够透明，在实际开发中我们基本无需关心虚拟 DOM 是如何运作的。然而，作为有态度的程序员，我们总是对技术背后的原理充满着好奇。理解其运行机制不仅有助于更好的理解 React 组件的生命周期，而且对于进一步优化 React 程序也会有很大帮助。

什么是 DOM Diff 算法

Web 界面由 DOM 树来构成，当其中某一部分发生变化时，其实就是对应的某个 DOM 节点发生了变化。在 React 中，构建 UI 界面的思路是由当前状态决定界面。前后两个状态就对应两套界面，然后由 React 来比较两个界面的区别，这就需要对 DOM 树进行 Diff 算法分析。

即给定任意两棵树，找到最少的转换步骤。但是标准的的 Diff 算法复杂度需要 O(n^3)，这显然无法满足性能要求。要达到每次界面都可以整体刷新界面的目的，势必需要对算法进行优化。这看上去非常有难度，然而 Facebook 工程师却做到了，他们结合 Web 界面的特点做出了两个简单的假设，使得 Diff 算法复杂度直接降低到 O(n)。

1. 两个相同组件产生类似的 DOM 结构，不同的组件产生不同的 DOM 结构；
2. 对于同一层次的一组子节点，它们可以通过唯一的 id 进行区分。

算法上的优化是 React 整个界面 Render 的基础，事实也证明这两个假设是合理而精确的，保证了整体界面构建的性能。

不同节点类型的比较

为了在树之间进行比较，我们首先要能够比较两个节点，在 React 中即比较两个虚拟 DOM 节点，当两个节点不同时，应该如何处理。这分为两种情况：（1）节点类型不同 ，（2）节点类型相同，但是属性不同。本节先看第一种情况。

当在树中的同一位置前后输出了不同类型的节点，React 直接删除前面的节点，然后创建并插入新的节点。假设我们在树的同一位置前后两次输出不同类型的节点。

```
001   renderA: <div />
002 renderB: <span />
003 => [removeNode <div />], [insertNode
      <span />]
```

当一个节点从 div 变成 span 时，简单的直接删除 div 节点，并插入一个新的 span 节点。这符合我们对真实 DOM 操作的理解。

需要注意的是，删除节点意味着彻底销毁该节点，而不是再后续的比较中再去看是否有另外一个节点等同于该删除的节点。如果该删除的节点之下有子节点，那么这些子节点也会被完全删除，它们也不会用于后面的比较。这也是算法复杂能够降低到 O（n）的原因。

上面提到的是对虚拟 DOM 节点的操作，而同样的逻辑也被用在 React 组件的比较，例如：

```
001  renderA: <Header />
002 renderB: <Content />
003 => [removeNode <Header />],
     [insertNode <Content />]
```

当 React 在同一个位置遇到不同的组件时，也是简单的销毁第一个组件，而把新创建的组件加上去。这正是应用了第一个假设，不同的组件一般会产生不一样的 DOM 结构，与其浪费时间去比较它们基本上不会等价的 DOM 结构，还不如完全创建一个新的组件加上去。

由这一 React 对不同类型的节点的处理逻辑我们很容易得到推论，那就是 React 的 DOM Diff 算法实际上只会对树进行逐层比较，如下所述。

逐层进行节点比较

提到树，相信大多数同学立刻想到的是二叉树，遍历，最短路径等复杂的数据结构算法。而在 React 中，树的算法其实非常简单，那就是两棵树只会对同一层次的节点进行比较。如图 1 所示。

React 只会对相同颜色方框内的 DOM 节点进行比较，即同一个父节点下的所有子节点。当发现节点已经不存在，则该节点及其子节点会被完全删除掉，不会用于进一步的比较。这样只需要对树进行一次遍历，便能完成整个 DOM 树的比较。

例如，考虑有下面的 DOM 结构转换（图 2）。

A 节点被整个移动到 D 节点下，直观的考虑 DOM Diff 操作应该是：

```
001   reA.parent.remove(A);
002 D.append(A);
```

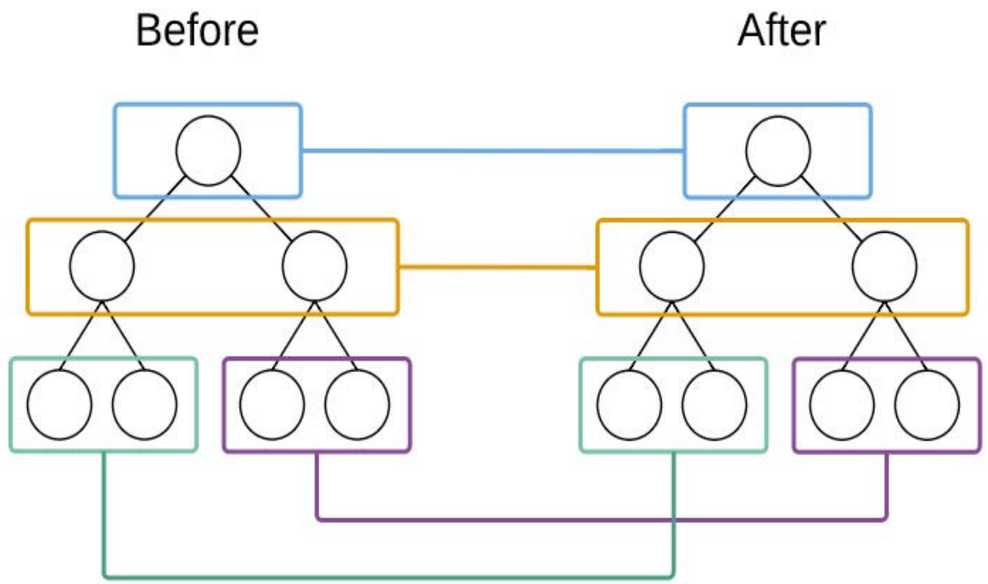



图 1

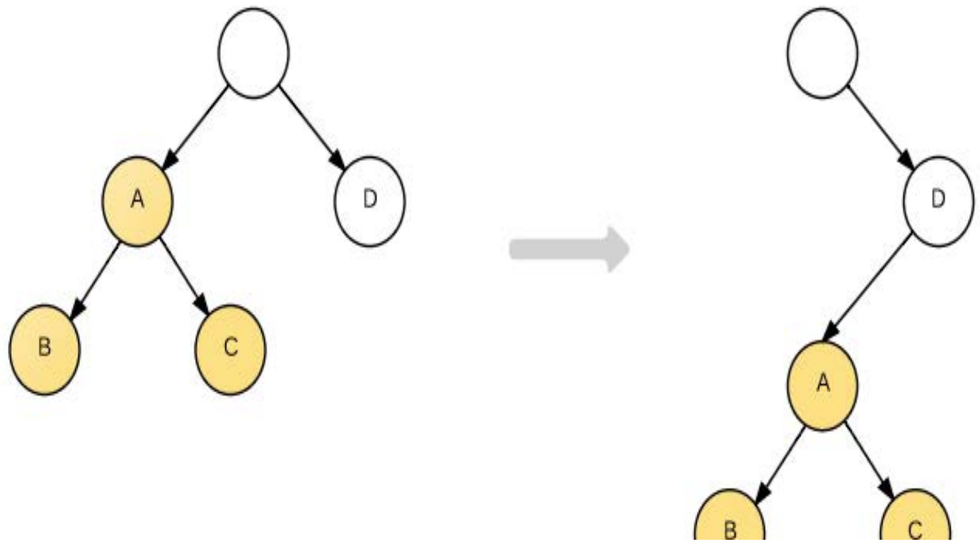


图 2

但因为 React 只会简单的考虑同层节点的位置变换，对于不同层的节点，只有简单的创建和删除。当根节点发现子节点中 A 不见了，就会直接销毁 A；而当 D 发现自己多了一个子节点 A，则会创建一个新的 A 作为子节点。因此对于这种结构的转变的实际操作是：

```
001 A.destroy();
002 A = new A();
003 A.append(new B());
004 A.append(new C());
005 D.append(A);
```

可以看到，以 A 为根节点的树被整个重新创建。

虽然看上去这样的算法有些“简陋”，但是其基于的是第一个假设：两个不同组件一般产生不一样的 DOM 结构。根据 React 官方博客，这一假设至今为止没有导致严重的性能问题。这当然也给我们一个提示，在实现自己的组件时，保持稳定的 DOM 结构会有助于性能的提升。例如，我们有时可以通过 CSS 隐藏或显示某些节点，而不是真的移除或添加 DOM 节点。

由 DOM Diff 算法理解组件的生命周期

在上一篇文章中介绍了 React 组件的生命周期，其中的每个阶段其实都是和 DOM Diff 算法息息相关的。例如以下几个方法：

- constructor：构造函数，组件被创建时执行；
- componentDidMount：当组件添加到 DOM 树之后执行；
- componentWillUnmount：当组件从 DOM 树中移除之后执行，在 React 中可以认为组件被销毁；
- componentDidUpdate：当组件更新时执行。

为了演示组件生命周期和 DOM Diff 算法的关系，笔者创建了一个示例：https://supnate.github.io/react-dom-diff/index.html ，大家可以直接访问试用。这时当 DOM 树进行如下转变时，即从“shape1”转变到“shape2”时。我们来观察这几个方法的执行情况（图 3）。

浏览器开发工具控制台输出如下结果：

```
001 C will unmount.
002 C is created.
003 B is updated.
004 A is updated.
005 C did mount.
006 D is updated.
007 R is updated.
```

可以看到，C 节点是完全重建后再添加到 D 节点之下，而不是将其“移动”过去。如果大家有兴趣，也可以 fork 示例代码。从而可以自己添加其它树结构，试验它们之间是如何转换的。

相同类型节点的比较

第二种节点的比较是相同类型的节点，算法就相对简单而容易理解。React 会对属性进行重设从而实现节点的转换。例如：

```
001 renderA: <div id=”before” />
002 renderB: <div id=”after” />
003 => [replaceAttribute id “after”]
```

虚拟 DOM 的 style 属性稍有不同，其值并不是一个简单字符串而必须为一个对象，因此转换过程如下：

```
001 renderA: <div style={{color: ‘red’}}
      />
002 renderB: <div style={{fontWeight:
     ‘bold’}} />
003 => [removeStyle color], [addStyle
     font-weight ‘bold’]
```

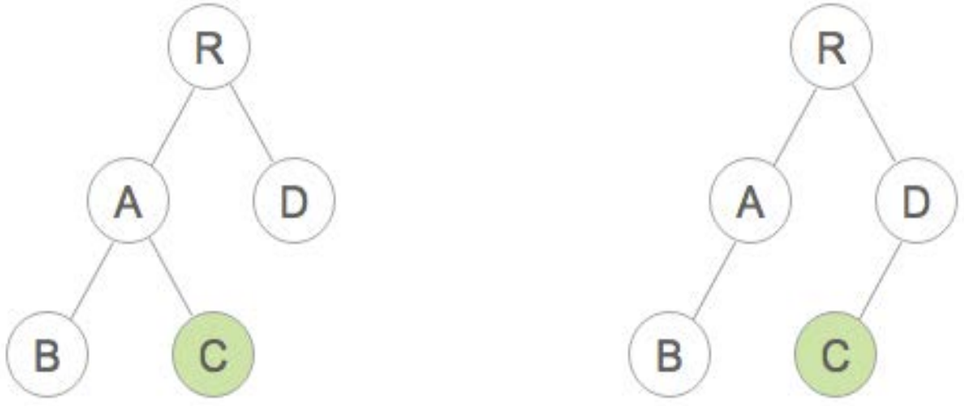



图 3

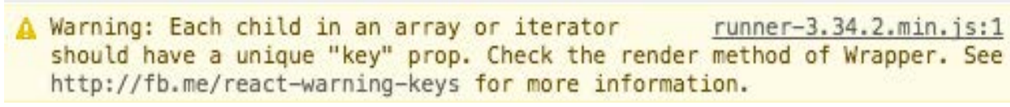


图 4

列表节点的比较

上面介绍了对于不在同一层的节点的比较，即使它们完全一样，也会销毁并重新创建。那么当它们在同一层时，又是如何处理的呢？这就涉及到列表节点的 Diff 算法。相信很多使用 React 的同学大多遇到过图 4 这样的警告。

这是 React 在遇到列表时却又找不到 key 时提示的警告。虽然无视这条警告大部分界面也会正确工作，但这通常意味着潜在的性能问题。因为 React 觉得自己可能无法高效的去更新这个列表。

列表节点的操作通常包括添加、删除和排序。例如下图，我们需要往 B 和 C 直接插入节点 F，在 jQuery 中我们可能会直接使用 $(B).after(F) 来实现。而在 React 中，我们只会告诉 React 新的界面应该是 A-B-F-C-D-E，由 Diff 算法完成更新界面。（图 5）

这时如果每个节点都没有唯一的标识，React 无法识别每一个节点，那么更新过程会很低效，即，将 C 更新成 F，D 更新成 C，E 更新成 D，最后再插入一个 E 节点。效果如图 6 所示。

可以看到，React 会逐个对节点进行更新，转换到目标节点。而最后插入新的节点 E，涉及到的 DOM 操作非常多。而如果给每个节点唯一的标识（key），那么 React 能够找到正确的位置去插入新的节点，如图 7 所示。

对于列表节点顺序的调整其实也类似于插入或删除，下面结合示例代码我们看下其转换的过程。仍然使用前面提到的示例 ，我们将树的形态从 shape5 转换到 shape6（图 8）。

即将同一层的节点位置进行调整。如果未提供 key，那么 React 认为 B 和 C 之后的对应位置组件类型不同，因此完全删除后重建，控制台输出如下。

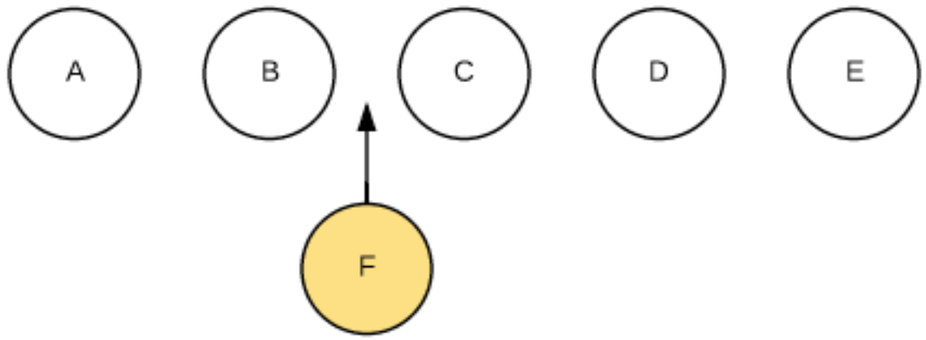


图 5

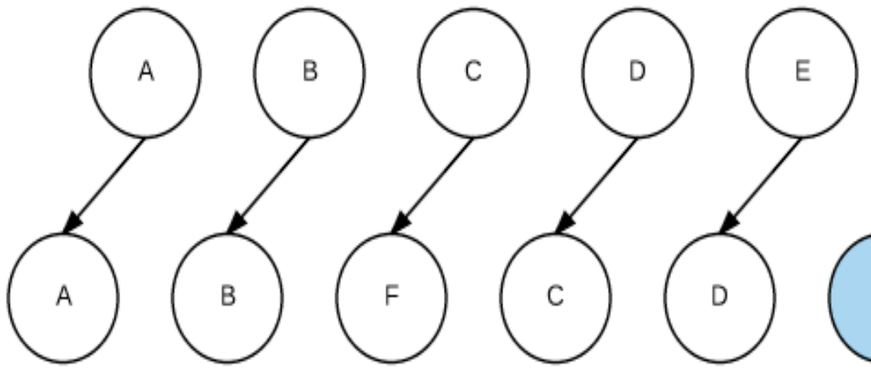


图 6

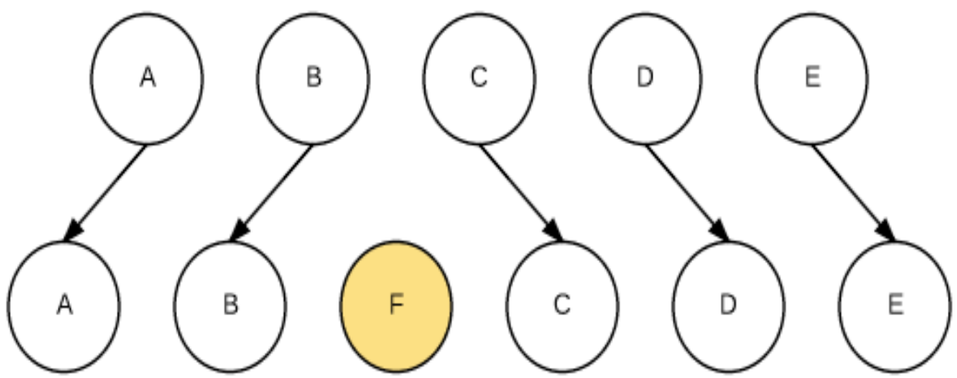


图 7

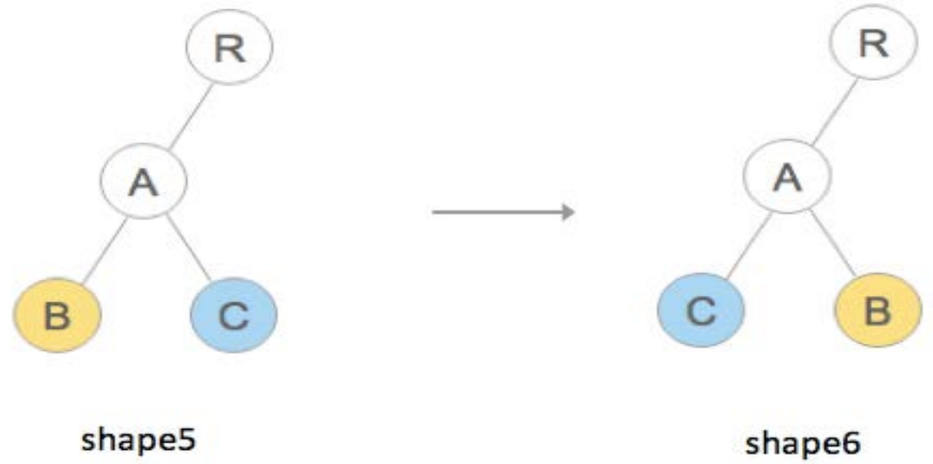


图 8

```
001 B will unmount.
002 C will unmount.
003 C is created.
004 B is created.
005 C did mount.
006 B did mount.
007 A is updated.
008 R is updated.
```

```
001 C is updated.
002 B is updated.
003 A is updated.
004 R is updated.
```

而如果提供了 key，如下面的代码：

```
001 shape5: function() {
002   return (
003     <Root>
004       <A>
005         <B key=”B” />
006         <C key=”C” />
007       </A>
008     </Root>
009   );
010 },
011
012 shape6: function() {
013   return (
014     <Root>
015       <A>
016         <C key=”C” />
017         <B key=”B” />
018       </A>
019     </Root>
020   );
021 },
```

那么控制台输出如下：

可以看到，对于列表节点提供唯一的 key 属性可以帮助 React 定位到正确的节点进行比较，从而大幅减少 DOM 操作次数，提高了性能。

小结

本文分析了 React 的 DOM Diff 算法究竟是如何工作的，其复杂度控制在了 O（n），这让我们考虑 UI 时可以完全基于状态来每次 render 整个界面而无需担心性能问题，简化了 UI 开发的复杂度。而算法优化的基础是文章开头提到的两个假设，以及 React 的 UI 基于组件这样的一个机制。理解虚拟 DOM Diff 算法不仅能够帮助我们理解组件的生命周期，而且也对我们实现自定义组件时如何进一步优化性能具有指导意义。

Apache Calcite：Hadoop中新型大数据查询引擎

作者 楚晗

Apache Calcite是面向Hadoop新的查询引擎，它提供了标准的SQL语言、多种查询优化和连接各种数据源的能力，除此之外，Calcite还提供了OLAP和流处理的查询引擎。正是有了这些诸多特性，Calcite项目在Hadoop中越来越引入注目，并被众多项目集成。

Calcite之前的名称叫做optiq，optiq起初在Hive项目中，为Hive提供基于成本模型的优化，即CBO（Cost Based Optimizatio）。2014年5月optiq独立出来，成为Apache社区的孵化项目，2014年9月正式更名为Calcite。Calcite项目的创建者是Julian Hyde，他在数据平台上有非常多的工作经历，曾经是Oracle、 Broadbase公司SQL引擎的主要开发者、SQLStream公司的创始人和主架构师、Pentaho BI套件中OLAP部分的架构师和主要开发者。现在他在Hortonworks公司负责Calcite项目，其工作经历对Calcite项目有很大的帮助。除了Hortonworks，该项目的代码提交者还有MapR、Salesforce等公司，并且还在不断壮大。

Calcite的目标是“one size fits all（一种方案适应所有需求场景）”，希望能为不同计算平台和数据源提供统一的查询引擎，并以类似传统数据库的访问方式（SQL和高级查询优化）来访问Hadoop上的数据。

Apache Calcite具有以下几个技术特性：

- 支持标准SQL语言；
- 独立于编程语言和数据源，可以支持不同的前端和后端；
- 支持关系代数、可定制的逻辑规划规则和基于成本模型优化的查询引擎；

- 支持物化视图（materialized view）的管理（创建、丢弃、持久化和自动识别）；
- 基于物化视图的 Lattice 和 Tile 机制，以应用于 OLAP 分析；
- 支持对流数据的查询。

下面对其中的一些特性更详细的介绍。

基于关系代数的查询引擎

我们知道，关系代数是关系型数据库操作的理论基础，关系代数支持并、差、笛卡尔积、投影和选择等基本运算。关系代数是 Calcite 的核心，任何一个查询都可以表示成由关系运算符组成的树。 你可以将 SQL 转换成关系代数，或者通过 Calcite 提供的 API 直接创建它。比如下面这段 SQL 查询：

```
001 SELECT deptno, count(*) AS c, sum(sal)
      AS s
002 FROM emp
003 GROUP BY deptno
004 HAVING count(*) > 10
```

可以表达成如下的关系表达式语法树：

```
001 LogicalFilter(condition=[>($1, 10)])
002   LogicalAggregate(group=[{7}],
      C=[COUNT()], S=[SUM($5)])
003     LogicalTableScan(table=[[scott,
      EMP]])
```

当上层编程语言，如 SQL 转换为关系表达式后，就会被送到 Calcite 的逻辑规划器进行规则匹配。在这个过程中，Calcite 查询引擎会循环使用规划规则对关系表达式语法树的节点和子图进行优化。这种优化过程会以一个成本模型作为参考，每次优化都在保证语义的情况下利用规则来降低成本，成本主要以查询时间最快、资源消耗最少这些维度去度量。

使用逻辑规划规则等同于数学恒等式变换，比如将一个过滤器推到内连接（inner join）输入的内部执行，当然使用这个规则的前提是过滤器不会引用内连接输入之外的数据列。图 1 就是一个将 Filter 操作下推到 Join 下面的示例，这样做的好处是减少 Join操作记录的数量。

非常好的一点是 Calcite 中的查询引擎是可以定制和扩展的，你可以自定义关系运算符、规划规则、成本模型和相关的统计，从而应用到不同需求的场景。

动态的数据管理系统

Calcite 的设计目标是成为动态的数据管理系统，所以在具有很多特性的同时，它也舍弃了一些功能，比如数据存储、处理数据的算法和元数据仓库。由于舍弃了这些功能，Calcite 可以在应用和数据存储、数据处理引擎之间很好地扮演中介的角色。用 Calcite 创建数据库非常灵活，你只需要动态地添加数据即可。

同时，前面提到过，Calcite 使用了基于关系代数的查询引擎，聚焦在关系代数的语法分析和查询逻辑的规划制定上。它不受上层编程语言的限制，前端可以使用 SQL、Pig、Cascading 或者 Scalding，只要通过 Calcite 提供的 API 将它们转化成关系代数的抽象语法树即可。

同时，Calcite 也不涉及物理规划层，它通过扩展适配器来连接多种后端的数据源和处理引擎， 如 Spark、Splunk、HBase、Cassandra 或者 MangoDB。简单的说，这种架构就是“一种查询引擎，连接多种前端和后端”。

物化视图的应用

Calcite 的物化视图是从传统的关系型数据库系统（Oracle/DB2/Teradata/SQL server）借鉴而来，传统概念上，一个物化视图包含一个 SQL 查询和这个查询所生成的数据表。

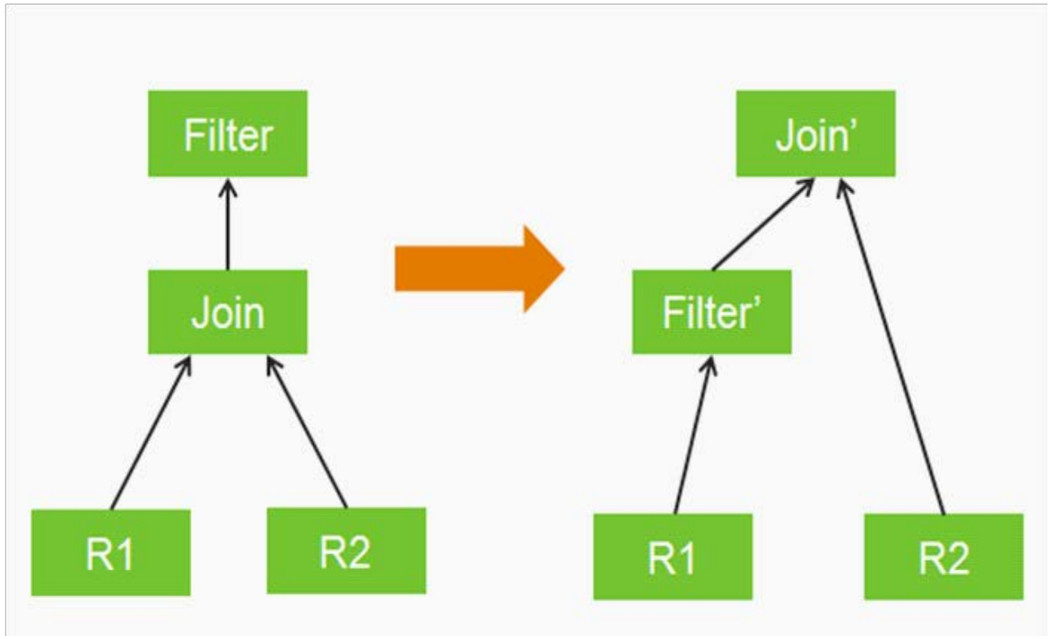


图 1：一个逻辑规划的规则匹配（Filter 操作下沉）

下面是在 Hive 中创建物化视图的一个例子， 比如下面这个对员工表（emp）的查询（女性的平均工资）：

```
001 SELECT deptno, AVG(salary) AS average_
      sal
002 FROM emp WHERE gender = ‘F’
003 GROUP BY deptno;
```

它按部门、性别统计出相应的员工数量和工资总额：

```
001 CREATE MATERIALIZED VIEW emp_summary
      AS
002 SELECT deptno, gender, COUNT(*) AS c,
      SUM(salary) AS s
003 FROM emp
004 GROUP BY deptno, gender;
005 ;
```

因为物化视图本质上也是一个数据表，所以你可以直接查询它，比如下面这个例子查询男员工人数大于 20 的部门：

```
001 SELECT deptno FROM emp_summary
002 WHERE gender = ‘M’ AND c > 20;
```

更重要的是，你还可以通过物化视图的查询取代对相关数据表的查询，可参见图 2。由于物化视图一般存储在内存中，且其数据更接近于最终结果，所以查询速度会大大加快。

可以被Calcite规划器改写成对物化视图(emp_summary）的查询：

```
001 SELECT deptno, s / c AS average_sal
002 FROM emp_summary WHERE gender = ‘F’
003 GROUP BY deptno;
```

我们可以看到，多数值的平均运算，即先累加再除法转化成了单个除法。

为了让物化视图可以被所有编程语言访问，需要将其转化为与语言无关的关系代数并将其元数据保存在 Hive 的 HCatalog 中。HCatalog 可以独立于 Hive，被其它查询引擎使用，它负责

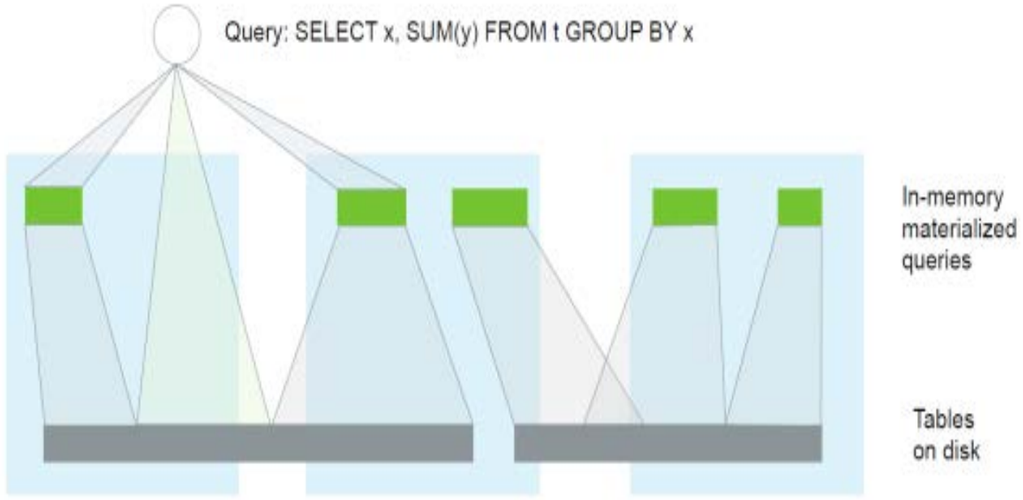


图 2：查询、物化视图和表的关系

Hadoop 元数据和表的管理。

物化视图可以进一步扩展为DIMMQ。简单地说，DIMMQ 就是内存中可丢弃的物化视图，它是高级别的缓存。相对原始数据，它离查询结果更近，所占空间更小，并可以被多个应用共享，并且应用不必感知物化视图存在，查询引擎会自动匹配它。物化视图可以和异构存储结合起来，即它可以存储在 Disk、SSD 或者内存中，并根据数据的热度进行动态调整。

除了上面例子中的归纳表（员工工资、员工数量），物化视图还可以应用在其它地方，比如 b-tree 索引（使用基础的排序投影运算）、分区表和远端快照。总之，通过使用物化视图，应用程序可以设计自己的派生数据结构，并使其被系统自动识别和使用。

在线分析处理（OLAP）

为了加速在线分析处理，除了物化视图，Calcite 还引入Lattice（格子）和 Tile（瓷片）的概念。Lattice 可以看做是在星模式（star schema）数据模型下对物化视图的推荐、创建和识别的机制。这种推荐可以根据查询的频次统计，也可以基于某些分析维度的重要等级。Tile 则是 Lattice 中的一个逻辑的物化视图，它可以通过三种方法来实体化：

1. 在 lattice 中声明；
2. 通过推荐算法实现；
3. 在响应查询时创建。

图 3 是 Lattice 和 Tile 的一个图例，这个 OLAP 分析涉及五个维度的数据：邮政编码、州、性别、年和月。每个椭圆代表一个 Tile，黑色椭圆是实体化后物化视图，椭圆中的数字代表该物化视图对应的记录数。

由于 Calcite 可以很好地支持物化视图和星模式这些 OLAP 分析的关键特性，所以 Apache 基金会的 Kylin 项目（Hadoop 上 OLAP 系统）在选用查询引擎时就直接集成了 Calcite。

支持流查询

Calcite 对其 SQL 和关系代数进行了扩展以支持流查询。Calcite 的 SQL 语言是标准 SQL 的扩展，而不是类 SQL（SQL-like），这个差别

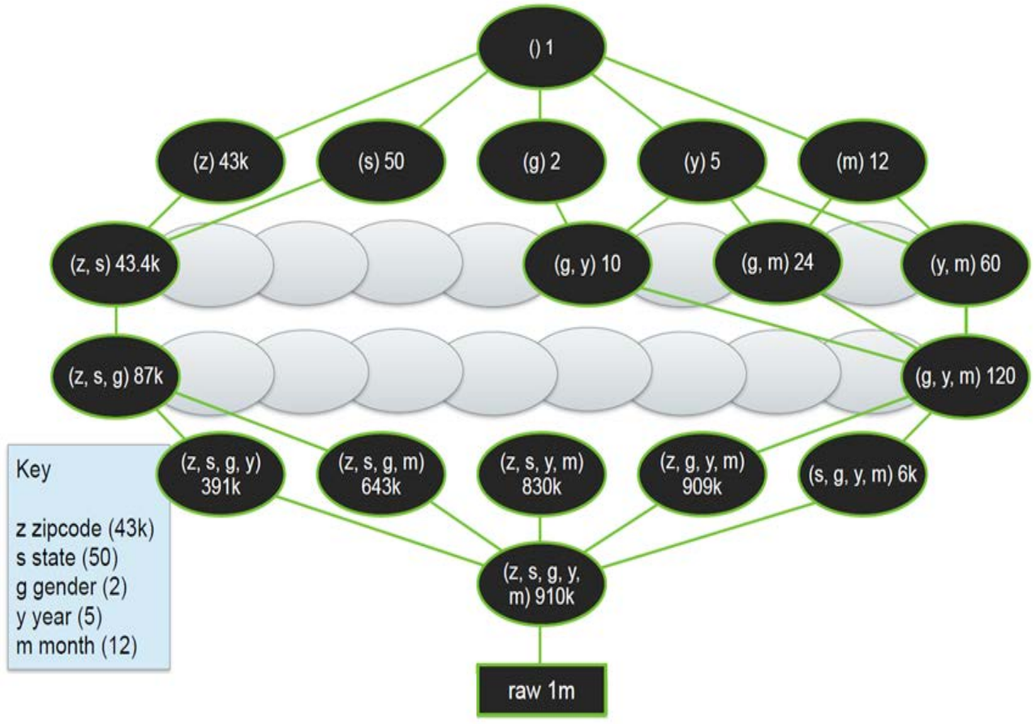


图 3：Lattice 和 Tile 的示例图

非常重要，因为：

- 如果你懂标准 SQL，那么流的 SQL 也会非常容易学；
- 因为在流和表上使用相同的机制，语义会很清楚；
- 你可以写同时对流和表结合的查询语句；
- 很多工具可以直接生成标准的 SQL。

Calcite 的流查询除了支持排序、聚合、过滤等常用操作和子查询外，也支持各种窗口操作，比如翻滚窗口（Tumbling window）、跳跃窗口（Hopping window）、滑动窗口（Sliding windows）、级联窗口（Cascading window）。其中级联窗口可以看作是滑动窗口和翻滚窗口的结合。

总结

Calcite 是一种动态数据管理系统，它具有标准 SQL、连接不同前端和后端、可定制的逻辑规划器、物化视图、多维数据分析和流查询等诸多能力，使其成为大数据领域中非常有吸引力的查询引擎，目前它已经或被规划集成到 Hadoop 的诸多项目中，比如 Lingual（Cascading 项目的 SQL 接口）、Apache Drill、Apache Hive、Apache Kylin、Apache Phoenix、Apache Samza 和 Apache Flink。

OPPO Monitor Platform

从应用请求到后端处理，自研解决服务化架构系统监控难题

作者 罗代均

众所周知，系统监控一直是拥有复杂 IT 架构的企业所面临的一个重要问题，而这也并不是每家企业都能够轻松解决的技术挑战。OPPO 作为一家国际智能终端设备及移动互联网服务供应商，推出过多款外观精细、功能可靠的智能手机产品，其品牌知名度也一直名列前茅。但实际上 OPPO 公司与其他快速发展的现代企业一样面临着自己的IT挑战，而更加鲜为人知的，则是其品牌背后同样出色的 IT 团队与信息化支持能力。

OPPO 后端系统规模近几年快速发展，系统重构以后采用了服务化的架构，各系统之间耦合降低，开发效率得到了很大提升。然而在服务化带来了好处的同时，难于监控的问题也一并出现。由于服务之间调用关系错综复杂，接口出现问题，多个系统报错，因此很难定位真正的故障源头。整个请求调用链就像一个黑盒，无法跟踪请求的整个调用路径，发现性能瓶颈点。为了解决这些问题，OPPO 公司自行开发了一套监控系统，并结合第三方监控系统，形成了从 App 请求开始到后端处理过程的完整监控体系。OPPO 监控系统的简称为 OMP(OPPO Monitor Platform)，历时半年开发，分为两期上线，现在已全面接入 OPPO 线上项目。

三大理由决定自主研发

之所以选择自主研发监控系统，主要是考虑到三方面的原因：定制化需求、易用性、以及开发成本低。

首先，在对比之后发现现有的开源监控软件无法满足 OPPO 的需求。对于监控系统来说最核心的一条需求，就是要能够监控每个 App 请求的完整调用链，从 App 发起请求，到后端的负载均衡接入、API Server、微服务调用、缓存、消息队列、数据库访问时间等。系统架构微服

务化以后，服务跟踪和服务调用链监控尤为重要，否则系统故障及性能瓶颈就很难排查了。

为了打通用户请求的完整调用链，需要在API框架、RPC框架、缓存操作、数据库操作、队列消费等代码埋点，以及高性能处理和存储系统，而目前的开源软件无法满足需求，各大公司也因此才开发了自己的监控平台。由于服务调用跟踪功能跟开发框架深度关联，各公司选用的框架并不相同，所以业界鲜有类似开源的产品。

第二个原因是考虑到权限及一体化管理界面的需求。监控平台不仅仅面向运维人员，开发人员、运营人员、测试人员也需要经常使用。例如根据监控平台采集到JVM Young GC/Full GC次数及时间、耗时Top 10线程堆栈等信息，经常查看监控平台，开发、测试人员便可以评估代码质量，排除隐患。

监控平台面向用户众多，安全性及权限管理要求较高，同时需要一体化的管理界面，简洁易用，而组合多个开源软件，权限和管理便捷性很难满足需求。

第三，监控系统的开发难度比较低。自行研发的监控平台虽有千般好处，但是如果开发的难度太大，以至于无法持续的投入，那也是没有意义的。基于Sigar、kafka、Flume、HBase、Netty等技术，开发高性能、可伸缩的系统难度实际上并不大，需要投入的资源不需要很多。

六项目标内容实现线上应用全面监控

OMP的最终目标是提供一体化的监控系统，在同一套管理界面及权限体系之下，对线上应用系统进行多维度的监控。OMP现阶段主要监控内容包括：主机性能指标监控、中间件性能指标监控、服务调用链实时监控、接口性能指标监控、日志实时监控、业务指标实时监控。

主机性能指标监控方面的开源软件非常多，比如Zabbix、Cacti等。主要采集主机的CPU负载、内存使用率、各网卡的上下行流量、各磁盘读写速率、各磁盘读写次数（IOPS）、各磁盘空间使用率等。

借助开源的Sigar库，可以轻松采集主机信息，为了保证整个监控系统体验的一致性，以及系统扩展性、稳定性的要求，我们没有直接采用Zabbix等开源监控系统，而是自己开发Agent程序，部署在主机上采集信息。

Sigar（System Information Gatherer And Reporter），是一个开源的工具，提供了跨平台的系统信息收集的API。核心由C语言实现的，可以被以下语言调用： C/C++、Java 、Perl 、NET C# 、Ruby 、Python 、PHP 、Erlang 。

Sigar可以收集的信息包括：

1. CPU信息，包括基本信息（vendor、model、mhz、cacheSize）和统计信息（user、sys、idle、nice、wait）；
2. 文件系统信息，包括Filesystem、Size、Used、Avail、Use%、Type；
3. 事件信息，类似Service Control Manager；
4. 内存信息，物理内存和交换内存的总数、使用数、剩余数；RAM的大小；
5. 网络信息，包括网络接口信息和网络路由信息；
6. 进程信息，包括每个进程的内存、CPU占用数、状态、参数、句柄；
7. IO信息，包括IO的状态，读写大小等；
8. 服务状态信息。
9. 系统信息，包括操作系统版本，系统资源限制情况，系统运行时间以及负载，JAVA的版本信息等。

JVM	堆内存、永久代内存、老年代内存、线程CPU时间、线程堆栈、Yong GC、Full GC
MySQL	慢查询、QPS、TPS、连接数、空间大小、表锁、行锁...
Redis	QPS、命中率、连接数、条目数、占用内存...
Memcached	QPS、命中率、占用内存、条目数、连接数...
Nginx	每秒请求数、连接数、keepalive连接数、持久连接利用率...

对于中间件性能指标监控，目前根据业务使用中间件的情况来看，主要采集的中间件包括Nginx、MySQL、MongoDB、Redis、Memcached、JVM、Kafka等。实现方式为部署独立的采集服务器，通过中间件的Java客户端执行状态查询命令，解析出相应的性能指标，采集的部分指标上表所示。

系统架构微服务化以后，服务调用错综复杂，出了问题或性能瓶颈，往往很难定位。所以服务调用链实时监控极为重要。

服务调用链监控是从一个App发起请求开始，分析各环节耗时及错误情况，包括负载均衡接入、API Server耗时、微服务调用耗时、缓存访问耗时、数据库访问耗时、消息队列处理耗时等，以及各环节的错误信息，便于跟踪性能瓶颈及错误。

由于服务调用量巨大，同时便于管理员查看，监控系统不能存储所有请求的调用链，主要存储以下几种请求：

- 周期内最慢Top 1000请求：通过分析最慢的top 1000请求，可以判断主要的性能瓶颈环节，比如数据库访问，或者调用第三方公司接口耗时过多。
- 采样请求：根据设置采样比例，随机选取部分请求，存储请求的调用链。
- 关键字：满足关键字规则，存储请求的调用链。

接口性能指标监控，主要监控接口的可用性和响应时间，由内部监控和外部监控两部分组成。

外部监控：外部监控由第三方公司负责，分为两种，一是App中埋点，采集真实的业务请求性能指标。二是通过第三方公司部署在各地的采集点，主动监控接口在各地区的可用性和性能指标。

外部监控只能监控负载均衡器对外的最终接口服务地址的可用性和性能指标，如果要监控机房内部接口服务器，则需要机房内部部署第三方公司的Agent，这样会带来非常大安全风险，所以机房内部节点监控由内部监控完成。

内部监控：内部监控采用OMP，监控负载均衡层后面的接口服务器的可用性和性能指标，及时发现异常节点，同时OMP根据异常原因，回调业务系统提供的恢复URL，尝试恢复系统。

应用产生的日志分散在各应用服务器当中，由于安全管理非常严格，开发人员查看线上系统的日志非常不方便，同时日志内容匹配关键字需要发送告警通知相关人员。OMP将日志统一采集存储到Elastic Search集群，实现日志检索。OMP日志实时监控主要包括如下功能：

- 日志实时在线查看：监控平台可以实时查看日志文件的内容，效果类似tail -f命令，同时屏蔽内容中的敏感信息（如密码等）。
- 日志全文检索：全文检索日志内容及高亮显示。

- 关联日志查看：查看日志产生时刻，日志所属应用关联组件和应用的日志。
- 关键字告警：用户自己定义告警规则，符合匹配规则发送邮件和短信通知。

最后一项监控内容，是业务指标实时监控。除了监控系统主动采集的信息，还有业务层指标需要进行监控，如周期内订单数量、第三方数据同步结果等。这些业务层的指标数据，由各业务系统负责采集，然后上报到监控系统，监控系统完成图表展现及告警通知。

四大方面详解 OPM 系统设计

首先来了解一下 OPM 的系统体系架构，如图 1 所示。

1. 中间件采集器：独立部署多台中间件性能指标采集器，通过 Zookeeper 实现故障转移和任务分配。中间件采集器通过中间件的 Java 客户端执行状态查询命令，解析命令结果得到性能指标，由于状态查询得到的是最新累计值，采集器还负责计算周期内的均值、最大值、最小值等周期数据。中间件采集将采集到的数据实时上报到接收器集群。
2. Agent 监控代理：Agent 监控代理部署在各服务器上，实时采集服务器的日志文件内容、CPU 负载、内存使用率、网卡上下行流量、磁盘读写速率、磁盘读写次数（IOPS）等。Agent 采集到的数据实时上报到接收器集群，对于日志文件，为防止阻塞，上传过程还需要做流控和丢弃策略。
3. 代码埋点：代码埋点主要采集服务调用链数据，通过封装的缓存访问层、数据库访问层、消息队列访问层，以及分布式服务框架（RPC），获得服务调用链耗时和错误信息。代码埋点采集数据本机暂存，一分钟合并上报一次到接收器集群。
4. 业务指标上报：业务指标由各业务系统负责采集，上报到接收器集群，上报周期和策略由各业务决定。
5. 接收器集群：OPPO 自研的 Data Flow 组件，架构参考 Flume，内部包括输入、通道、输出三部分，将接收到的数据输出到 Kafka 队列，后文将作详细介绍。
6. Kafka 消息队列：由于监控数据允许丢失和重复消费，所以选择高性能的 Kafka 做为消息队列，缓冲消息处理。
7. 消息处理集群：消息处理集群订阅 Kafka 主题，并行处理消息，处理告警规则、发送通知、存储到 HBase 和 ES。
8. Hbase：HBase 存储指标类数据，管理控制台通过查询 HBase 生成实时图表。
9. Elastic Search：存储日志内容，实现日志全文检索。

OPPO Data Flow（图 2）实现了数据流配置和管理，设计参考 Flume，内部包括 Source（输入）、通道（Channel）、输出（Sink）三部分，通道是一个队列，具备缓冲数据的功能。之所以不采用 Flume，主要考虑如下几个原因：

1. Flume 提供了良好的 Source—channel—Sink 框架，但具体的 Source、Sink 需要自己去实现，以兼容 oppo 线上使用软件版本，以及优化的参数配置。
2. Flume 资源占用较大，不适合作为 Agent 部署在业务服务器上
3. Flume 配置文件采用 properties 方式，不如 xml 配置直观，更不能管理界面来配置
4. Flume 管理界面不友好，不能查看输入、输出的实时流量图表以及错误数量。

参考 Flume 的设计思想，OPPO Data Flow 是更易管理、配置更便捷的数据流工具。使用开源软件，并不只是拿来就用这一种方式，学习其设计精华，从而进一步改进也是一种方式。

实际上，Agent 监控代理、中间件采集器、接收器集群都是 OPPO Data Flow 组件，组合不同的 Source 和 Sink。Source、Sink 采用 OSF 服务框架开发，实现 Agent—接收器的自动发现、负载均衡及故障转移功能。（表 2）

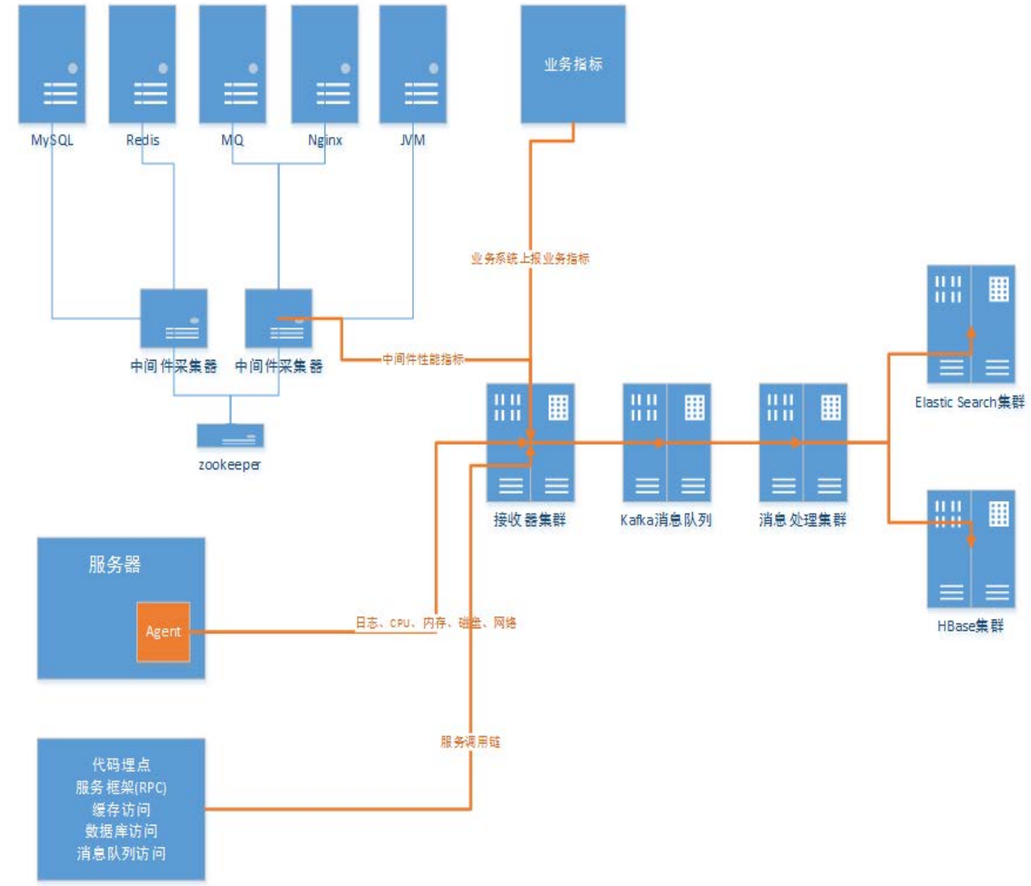


图 1

图 3 为 Data Flow 内嵌管理界面，可以查看数据流量和错误信息，点击名称可以查看历史流量。

服务调用链是监控的重点，核心的核心，为了打通服务调用链，OPPO 开发了 OSF（OPPO Service Framework）分布式服务框架，并对缓存、数据库、消息队列操作进行封装埋点，目的是透明的实现服务调用跟踪。实现方式如下：

1. 在 App 请求的入口生成唯一 requestID，放入 ThreadLocal
2. 缓存访问层代码埋点，从 ThradLocal 取出 requestID，记录缓存操作耗时
3. 数据库访问层代码埋点，从 ThradLocal 取出 requestID，记录数据库操作耗时
4. 调用其它微服务（RPC），将 requestID 传递到下一个微服务，微服务将接收到的 requestID 存入 ThreadLocal，微服务内部的缓存、数据库操作同样记录 requestID 操作耗时及错误信息。
5. 消息队列写入、消费代码埋点，传递 requestID，记录消息消费耗时。

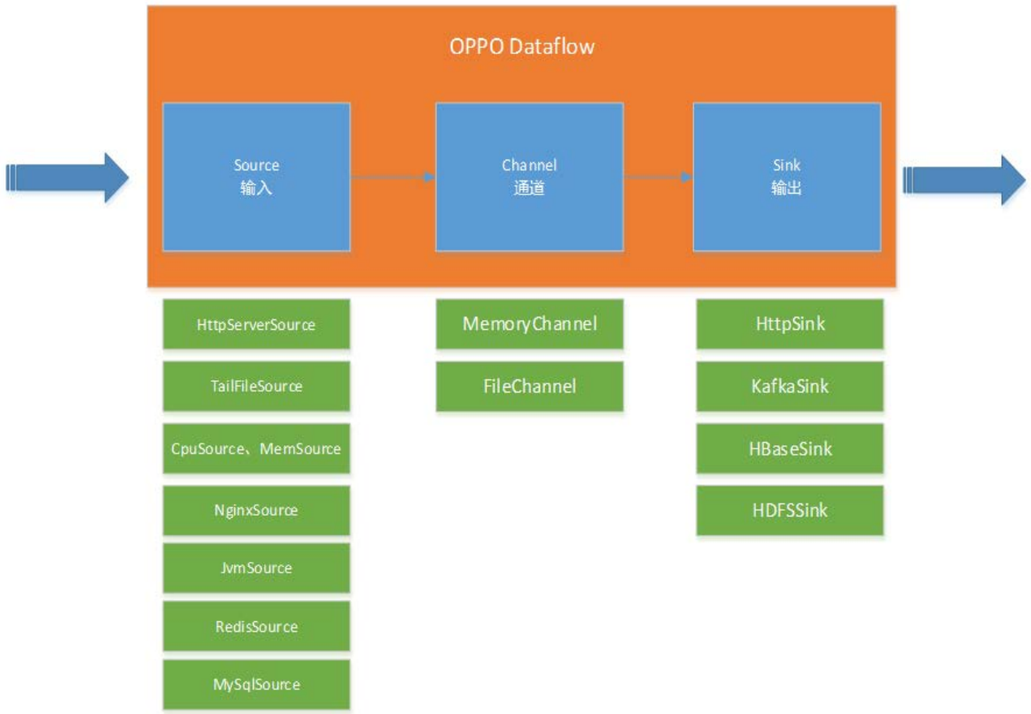


图 2

图 3

	输入(Source)	通道(Channel)	输出(Sink)
Agent 监控代理	TailFileSource CPUSource MemorySource NetworkSource DiskSource	MemoryChannel	HttpSink
中间件采集器	NginxSource MySqlSource MongoDBSource RedisSource JvmSource MemcachedSource	MemoryChannel	HttpSink
接收器	HttpSource	FileChannel	KafkaSink

表 2

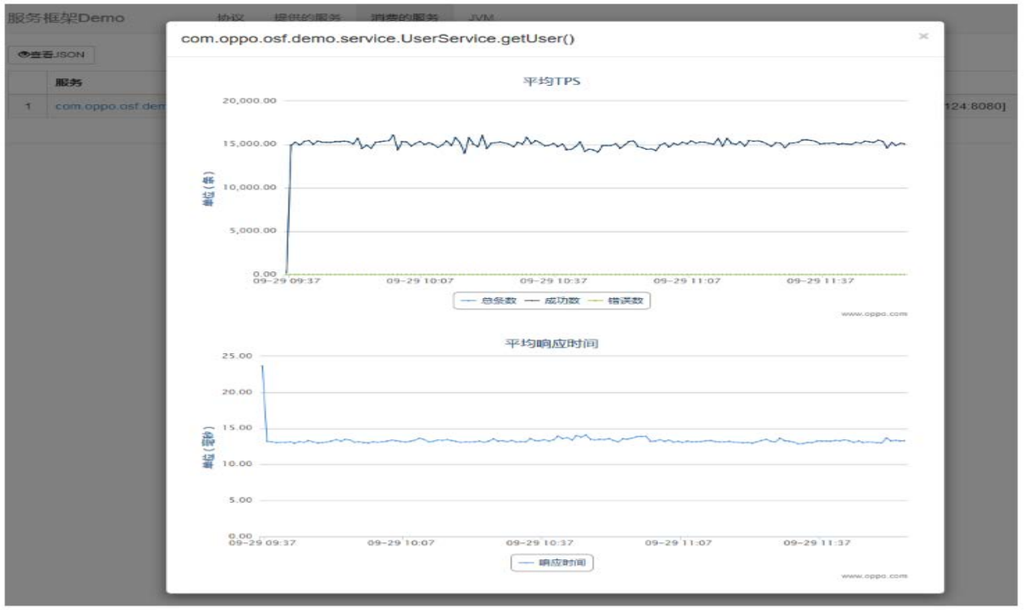
调用链数据庞大，无法全量存储，监控系统将周期内最慢 Top1000 请求，采样的部分请求以及符合关键字规则请求的服务调用链存储在 HBase 中，管理控制台可以快速分析查看。

分布式服务框架是打通服务调用链的关键。开源的 Dubbo 应用广泛，考虑到 Dubbo 版本较长时间没有更新（有些 Dubbo 依赖库已经跟开发生态的其他开源组件版本冲突）、代码量较大，而且服务治理能力较弱，很难完全掌控 Dubbo 的所有细节，而前文提到的 OPPO 自行开发的分布式服务框架 OSF，代码精简满足核心需求，与监控系统深度集成。

OSF 实现微服务 RPC 调用 requestID 的传递，记录每个服务的调用耗时及错误信息，框架每分钟汇总上报微服务调用耗时及错误信息到监控平台。

OSF 主要特性如下：

1. 支持 RESTFul 协议，容器支持 Tomcat、Netty、JDK Http Server；
2. 支持 TCP 二进制协议，容器支持 Netty；
3. 支持 HTTP/2 协议，测试中；
4. 支持 Protobuf、JProtobuf、Kryo、FST、MessagePack、Jackson、GSON、Hessian 序列化实现由消费方决定序列化方式；
5. 注册中心基于 MySQL，同时提供推送、client 拉取两种方式，保证服务发现可靠性；
6. 注册中心提供 HTTP API，支持多语言、移动设备；
7. 支持多数据中心部署；
8. I/O 线程与工作线程池分离，提供方忙时立即响应 client 重试其它节点。



服务调用统计

图 4

从可靠性及伸缩性角度来看，主要包括以下内容：

1. 接收器：接收器的输入采用 OSF RESTFul 协议开发，通过注册中心，client 能够自动发现接收器节点的变化，通过 client 实现负载均衡及故障转移，从而保证接收器的可靠性、伸缩性。
2. 中间件采集器：中间件采集器通过 zookeeper 选举 Master，由 Mater 来分配采集任务，采集器节点变化，选举的 Master 重新分配采集任务，这样任意增减采集器节点，都能重新平衡采集任务，保证采集任务的持续可靠运行。
3. 消息处理：由于多个节点均分同一个 kafka topic 的消息并且实现高可用比较困难，OMP 预先定义了若干个 kafka topic，消息处理节点通过 zookeeper 选举 Master，由 Master 来分配 Topic 数量，当某个消息处理节点宕机，该节点负责的 topic 转移到其他节点继续处理。
4. Agent 监控代理：服务器上 shell 脚本定期检查 Agent 状态，不可用时自动重启 Agent，同时 OMP 维持与 Agent 之间的心跳消息，超过 3 个周期没有收到 Agent 的心跳消息，OMP 发送告警通知相关人员处理。

从 OPPO 的自主研发监控系统的实践案例来看，一切应当从业务需求出发，目的是解决业务遇到的问题。面对开源软件的选择，要有所“为”，有所“不为”。业界有很多成熟的开源软件，也有一些比较大胆的设计思想可供借鉴，但开源软件并不是拿过来就能用好这么简单的，选择的原则可“管”可“控”。一个开源软件，如果不能“掌控”，不够简单，那就不如不用，自己用土办法也许反而会更好，出了问题至少还能想想应急的办法。同样要具备“管理”性，不然黑盒子般运行，心里没底，那作为 IT 管理人员来说就睡不安心了。

作者简介

罗代均，现就职于 OPPO 基础技术团队，从事监控平台、服务框架等基础技术开发工作。2005 年毕业后，先后主导过通信、移动金融、应用商店、PaaS 平台等领域多个产品系统设计开发、项目管理工作。