

架构师

ARCHITECT



推荐文章 | Article

[成为一名优秀的Web前端开发者](#)

[Java永久代去哪儿了](#)

专题 | Topic

[Netty案例集锦之多线程篇](#)

[Kafka Consumer解析](#)

观点 | Opinion

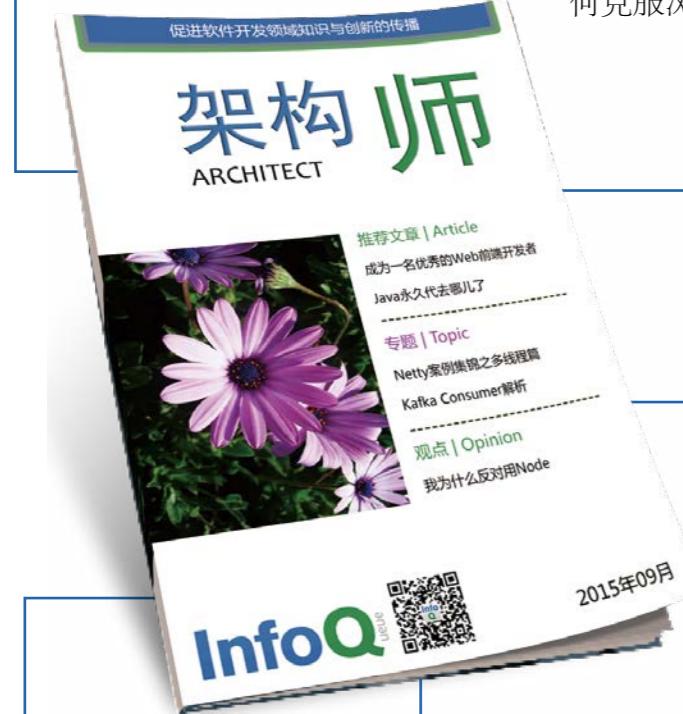
[我为什么反对用Node](#)





Java永久代去哪儿了

随着Java8的到来，我们再也见不到永久代了。它到底去了哪里？



Kafka Consumer解析

文章将详细介绍High Level Consumer, Consumer Group, Consumer Group Rebalance和Simple Consumer, 以及未来新版本中对Kafka High Level Consumer的重新设计——使用Consumer Controller解决Split Brain和Herd等问题。

我为什么反对用Node

随着无线端的快速普及，前后端分离技术走上了前台，而Node由于它的一些特性被工程师快速接受尤其是前端工程师，所以产生了很多Node是否会引发新的技术变革的讨论。

架构师2015年9月刊

本期主编 徐川

流程编辑 丁晓昀

发行人 霍泰稳

成为一名优秀的Web前端开发者

本文记录了两位工程师为Web开发者们所提出的多条建议，其中一位推荐了多种实用的工具与技术，而另一位则对于如何克服浏览器开发时所面临的挑战提出了诸多建议。

Netty案例集锦之多线程篇

Netty案例集锦的案例来源于作者在实际项目中遇到的问题总结、以及Netty社区网友的反馈。

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

QCon [2015] [上海站]

全球软件开发大会

Brought by Geekbang, InfoQ
极客邦科技

9 折优惠
截至9月20日

团购享受
更多优惠

大会介绍

QCon上海2015将于10月15~17日在上海·光大会展中心国际大酒店举行。本次大会共设计了15大热点专题，涵盖大数据、云计算、移动开发、可扩展架构、团队建设、互联网产品设计、安全、开源等热门专题。目前已经确定半数讲师，像安全、编程语言等专题更是确定了全部演讲嘉宾。

专家云集，敬请期待！

主题演讲



畅销书《番茄工作法图解》作者
Staffan Nöteberg
Pixelut优先级方法
——更高效地排序任务优先级



Azul Systems联合创始人兼CTO
Gil Tene
注重实效的性能



Uber首席系统架构师, Voxer联合创始人
Matt Ranney
针对失效做设计
——Uber的系统伸缩之道

演讲嘉宾



余弦 知道创宇技术副总裁
安全专题演讲：
程序员与黑客2



许式伟 七牛云存储CEO
新语言与新发展专题演讲：
Go语言新特性与实践



梁磊 资深专家，高德架构师
可扩展、高可用架构专题演讲：
高德—快速转型时期的稳定性架构实践



黄哲铿 1号店高级技术总监
建设高效团队专题演讲：
如何从零打造高质效互联网技术团队？



梁堰波 明略数据高级工程师
机器学习专题演讲：
基于机器学习的银行卡消费数据预测与推荐



吴永巍 百度网页搜索部架构师
可扩展、高可用架构专题演讲：
百度网页搜索，规模大幅膨胀下的架构优化实践



赵世婧 沪江移动开发架构师
移动开发新趋势专题演讲：
pure-native移动跨平台架构设计与实战



邹光先 广州爱拍网络科技有限公司联合创始人&CTO
技术创业专题演讲：
人生不只一种可能



卷首语

这个月七牛刚刚举办了 D-Future 大会。这是一个以数据为主题的会议。如果要用一个词来概括当前互联网时代的特征，那就是两个字：数据。实际上信息一直都存在，只是它以前存在于原子世界，有了计算机和互联网之后，出现了一个由数据构成的新世界：比特世界。并且这个新的世界正在以每三年翻一倍的速度在膨胀。计算机和互联网扩展了人的逻辑能力，让我们有了很强的分析和预测未来的能力。在这样的数字化洪流下，我们的商业将受到非常巨大的影响。

首先，我们来看一看业务本身。互联网化最基础的一个诉求就是业务上网。我们对比一下传统的商业和新兴的商业形态，最大的一个不同是什么？如果我们用一个词去概括旧的商业形态，我个人想到的一句话是“一手交钱一手交货”，这就是旧的商业形态最基本的特色。但是互联网改变了这一切，它让远程交易成为了可能。互联网时代业务的特征，我也概括了一句话：“非结构化数据是人类最自然的沟通方式”。我们自然表达的语言文本以及图片、音频、视频等媒介，都是非结构化

数据。这些数据人类容易理解，但计算机则很难。非结构化数据忠实地传递人的意愿。比如，我拿起电话说几段语音，告诉对方我想要的东西。或者通过图片和视频，表达一个商品长什么样，商品该怎么使用的。我们可以看到，非结构化数据自然而然会成为交互的一个中介。这些非结构化数据是我们实际存在事物的记录，而这也是原子世界被疯狂地映射到比特世界一个根本原因，因为业务要上网。

第二，当业务上网之后，我们的运营会发生质变。在旧的商业过程当中，大部分的企业会找一些样本客户来做调查问卷。但业务上网后，可以天然地记录每一次的交易过程，能把所有用户的行为都记录下来。今天我们不是取样数据，而是全量地记录用户行为。每天我们都在产生上千万上亿的交易记录，而如何通过这些交易记录去改进我们的商业模式，是一个非常重要的课题。

无论是非结构化数据或者记录用户行为的日志，我们都面临很多机遇和挑战。我们先看一看非结构化数据。如前所述，现



在数据世界正以每三年翻一番的速度在膨胀，而这其中 95% 以上都是非结构化数据，而且这个比例还在不断的提升。如此惊人的数据量，应该如何收集，如何保存，如何进行分析和挖掘，这又是一个很重要的课题。非结构化数据今天主要的用途是用来做交互，但是计算机对于语义的理解非常原始，所以在交互的智能化程度上，仍然有非常巨大的提升空间。我们知道有自然语言分析、NLP 这样的一些技术，我们还有语音识别，有视频或者图片里面对于场景、对于动作的捕捉与识别等，但是这些都还非常早期。这些技术如果能够往前走一步，就会带来巨大的想象空间。用户每一次沟通，每一次交互过程，都沉淀了大量的信息，但限于我们的分析能力还很原始，所以今天几乎所有的非结构化数据都还没有二次分析。

我们再看记录用户行为的日志。日志是计算机生成的，所以它天然可以很容易被计算机去理解，这个理解是全面的，不会损失什么信息。所以日志本身是一个更高含金量的金矿，但是大部分的企业还没有意

识到这一点。另外今天日志的处理能力、分析方法，以及产生对经营有效指导的能力依然存在很多不足。超过半数的企业还没有记录日志，大部分企业对数据的分析仅仅停留在象日活用户、用户留存等基础阶段。绝大部分企业会定期删除日志。

我们刚刚分析了非结构化数据和日志相关的一些挑战，这些挑战绝不是七牛一家公司所能解决的，我们希望有志于去提升数据应用价值的企业，都能够一起共同开拓这个数据世界。我们希望能够分析数据的使用场景，去触及它的方方面面，去构建一个完整的技术栈，构建一个全新的商业形态。

平步云端，数据为先。让我们一起共同发掘数据背后的价值，共同构建新时代的商业文明。

成为一名优秀的Web前端开发者



作者 Abel Avram 译者 邵思华

本文记录了两位工程师为 Web 开发者们所提出的多条建议，其中一位推荐了多种实用的工具与技术，而另一位则对于如何克服浏览器开发时所面临的挑战提出了诸多建议。

Rebecca Murphey 是来自于 Bazaarvoice 的一位软件工程师。今年早些时候，她发布了一篇博客文章“[前端（JS）开发者的基本素质之 2015 版](#)”，为 JavaScript 开发者在进行客户端 web 开发时使用的工具与开发方式提出了一些建议。她在文章的总结中写道：

学习 ECMAScript 2015，推荐的参考资料有：[《Understanding ES6》](#)、[ES6 Rocks](#) 以及 [BabelJS](#)。我们在此还要加上一条，即 Axel Rauschmayer 的著作《[探索 ES6](#)》。考虑到在当前这个时间点上似乎还没有必要了解 ECMAScript 2015 的所有细节，Murphey 建议

开发者更深入地了解如何使用异步调用、回调以及 promise。

使用模块。Murphey 相信，模块毫无疑问应当作为客户端 web 应用程序的构建块。她最近在使用 webpack 以实现模块化的效果，但她希望让每个人都能够使用 ECMAScript 标准模块的那一天能够早日到来。

测试你的代码。在 Murphey 看来，为你的代码编写测试，并且保证代码的可测试性是至关重要的。虽然她对于 Intern “非常中意”，但由于习惯，她还是坚持使用 Mocha。关于这一方面，她也强烈推荐 Michael Feathers 的著作《[修改代码的艺术](#)》。

实现流程自动化。Murphey 曾经尝试使用 Grunt 与 Gulp，但她最终还是选择了 Yeoman。

因为在“使用不熟悉的技术开始一个全新的项目”，或是对第三方 JavaScript 应用的开发进行标准化时，Yeoman 的表现“非常出色”。Murphey 也提到了 Broccoli，认为它将来或许能够取代 Grunt 和 Yeoman。

编写高质量的代码。她的建议是，对“违反了项目中经过良好定义的风格指南”的代码进行重构，还应当使用 lint 工具，例如 JSCS 或 ESLint。

使用 Git。Murphey 建议在 Git 中使用特性分支，因此得以“通过交互式 rebase，在与他人分享提交时对提交进行清理，并且尽可能地在较小的单元上进行工作，以减少冲突的发生机率”。此外还应当通过 ghooks 在 push 操作与 commit 操作前运行钩子操作。

在服务端生成 HTML。出于性能方面的考虑，Murphey 推荐在大型项目中尽可能在服务端生成 HTML。“预先生成这些文件，将其作为静态文件保存，以加快处理请求的速度。随后在客户端的相应事件中可通过客户端代码操作这些 HTML 文件，并在客户端模板中修改。”

拥抱 Node。Murphey 建议 web 开发者熟练掌握 Node.js 的相关知识，至少要了解如何初始化一个 Node 项目、如何搭建一台 Express 服务器、以及如何使用 request 模块转发请求。

Philip Walton 是来自 Google 的一位软件工程师，他最近撰写了一篇博客文章“[如何成为一名优秀的前端工程师](#)”。这篇文章的观点另辟蹊径，他并没有向读者推荐任何工具或框架，而是专注于如何处理这一领域中的某些挑战。在他看来，优秀员工与真正杰出的人才的差别不在于他们的知识量，而在于他们的思维方式。他是这样描述开发者的智慧的：

真正理解背后的过程。对于 Walton 来说，仅仅编写出可以运行的代码算不得优秀。他见过许多编写 CSS 与 JavaScript 的人，他们“只求找到能够运行的代码，然后就继续下一步工作了。”很多时候，开发者并不了解某段代码运行的机制。Walton 建议开发者进行深入钻研：

要充分理解代码的工作原理或许会很耗时间，但我向你保证，从长远来说，这种方式最终将节省你大量的时间。一旦你充分理解你所参与的系统是如何运作的，你就无需不断地进行猜测与检验这些费时的工作了。

预先了解浏览器将产生的改动。Web 开发者应当持续了解有哪些浏览器的改动会破坏现有的代码。以下代码在 IE10 中必然会导致整个 JavaScript 框架的方法出错：

```
var isIE6 = !isIE7 && !isIE8 && !isIE9;
```

仔细阅读规范。Walton 指出，虽然阅读规范是一项艰辛的任务，但一旦出现浏览器对某个页面的渲染不同的情况，这一任务就是不可避免的了。他为此特别举例说明：

最近我遇到这样一个例子，与可伸缩（flex）元素的默认最小尺寸有关。根据规范所说，可伸缩元素的 min-width 与 min-height 的初始值是 auto，而不是 0，这就意味着在默认情况下，这些元素不可能收缩到比其中的内容尺寸还小。而在过去 8 个月中，Firefox 是唯一一个正确地实现了这一特性的浏览器。

如果你遇到了这个跨浏览器的不一致性问题，并且注意到你的网站在 Chrome、IE、Opera 和 Safari 上的展现完全相同，只在 Firefox 上有所差别，那你很可能会认为是 Firefox 的问

题。实际上，我曾多次发现这一情况，在我的 [Flexbugs](#) 项目中，有许多由用户报告的 bug 其实都是由这种不一致性所导致的。而如果我按照用户所提议的那些临时方案来改变实现方式，那么在两周前所发布的 Chrome 44 中又会产生问题。由于这些临时方案选择了违背规范的方式，它们在无形中起到了提倡不正确行为的负面作用。

代码审查。Walton 表示，从阅读他人的代码中可以学到很多知识，它可以拓宽你的思路，了解“新的工作方法”，同时也有助于你在团队中的工作。实际上，这一点确实相当必要，因为“作为一位工程师来说，你的时间大部分都是在一个现有的代码库中添加或修改代码”，而不是从头开始编写全新的代码。

与更聪明的人一起工作。Walton “强烈” 建议你至少在职业生涯的初期阶段要尽量在某个团队中进行工作，向更有经验的团队成员学习，并让他们审查你的代码。如果之后选择了自由

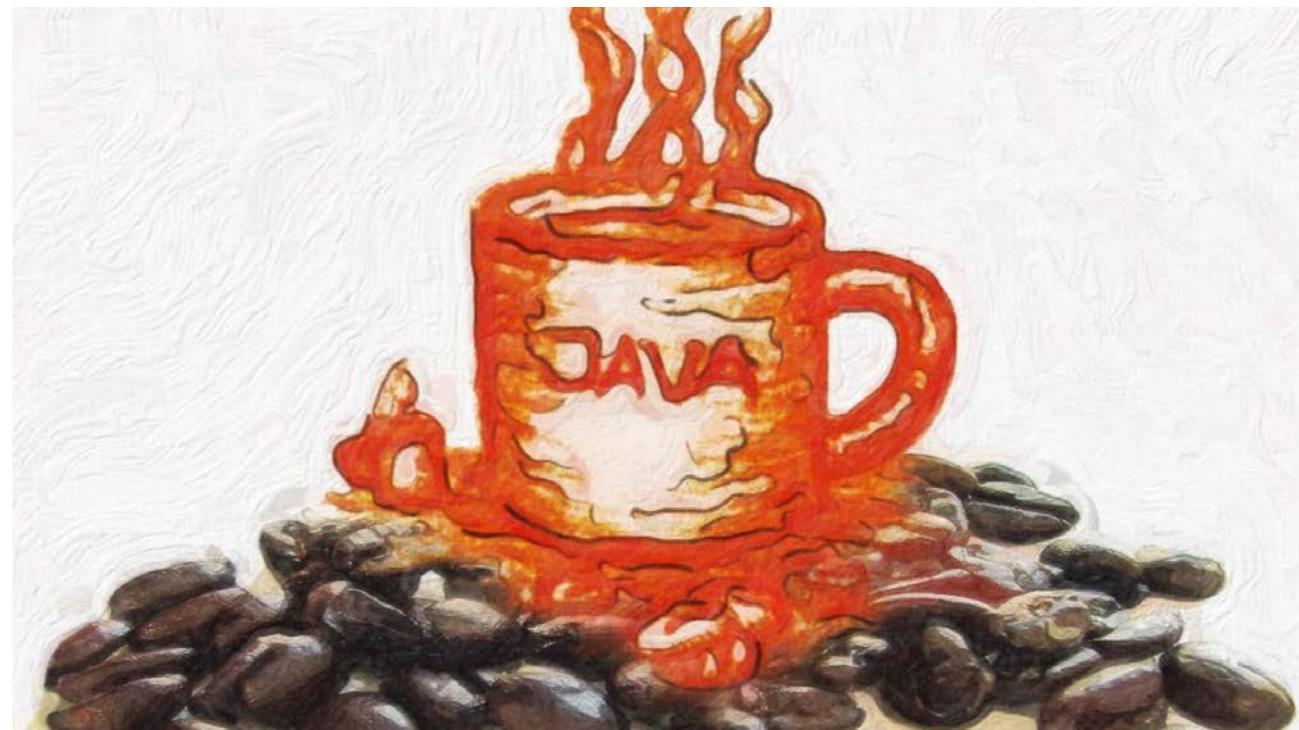
职业者这条职业路线，那么 Walton 建议你参与开源项目，这同样可以感受到在团队中工作的益处。

重复发明轮子。Walton 相信，虽然“重复发明轮子对于业务来说是有害的”，但它对于学习很有好处。在有些情况下，他建议你自己编写代码，而不是依赖于第三方的代码，因为这一过程将让你学到许多东西。当然这也要看情况而定。

将你的经验记录下来。Walton 的最后一条建议是将你所学到的东西用文字记录下来：“按我的经验来看，写作、演讲以及开发 demo，这些方法能够迫使我对知识点进行充分的挖掘，并做到从内到外的完全理解。哪怕你写的东西完全没人看，但写作的过程本身就已经值得你付出的努力了。”

查看英文原文：[Becoming a Great Web Front-end Developer](#)

Java永久代去哪儿了



作者 Monica Beckwith 译者 段建华

在 Java 虚拟机（以下简称 JVM）中，类包含其对应的元数据，比如类的层级信息，方法数据和方法信息（如字节码，栈和变量大小），运行时常量池，已确定的符号引用和虚方法表。

在过去（当自定义类加载器使用不普遍的时候），类几乎是“静态的”并且很少被卸载和回收，因此类也可以被看成“永久的”。另外由于类作为 JVM 实现的一部分，它们不由程序来创建，因为它们也被认为是“非堆”的内存。

在 JDK8 之前的 HotSpot 虚拟机中，类的这些“永久的”数据存放在一个叫做永久代的区域。永久代一段连续的内存空间，我们在 JVM 启动之前可以通过设置 -XX:MaxPermSize 的值来控

制永久代的大小，32 位机器默认的永久代的大小为 64M，64 位的机器则为 85M。永久代的垃圾回收和老年代的垃圾回收是绑定的，一旦其中一个区域被占满，这两个区都要进行垃圾回收。但是有一个明显的问题，由于我们可以通过 -XX:MaxPermSize 设置永久代的大小，一旦类的元数据超过了设定的大小，程序就会耗尽内存，并出现内存溢出错误（OOM）。

备注：在 JDK7 之前的 HotSpot 虚拟机中，纳入字符串常量池的字符串被存储在永久代中，因此导致了一系列的性能问题和内存溢出错误。想要了解这些永久代移除这些字符串的信息，请访问[这里](#)查看。

**国内首部Docker源码分析著作
CNUTCon全球容器技术大会推荐读物**

Docker 源码分析
THE SOURCE CODE ANALYSIS OF DOCKER
孙家俊 著

国内首部Docker源码分析著作
从源码角度全面解析Docker设计与实现
通过Docker理论与实践之间的对比

辞永久代，迎元空间

随着 Java8 的到来，我们再也见不到永久代了。但是这并不意味着类的元数据信息也消失了。这些数据被移到了一个与堆不相连的本地内存区域，这个区域就是我们要提到的元空间。

这项改动是很有必要的，因为对永久代进行调优是很困难的。永久代中的元数据可能会随着每一次 Full GC 发生而进行移动。并且为永久代设置空间大小也是很难确定的，因为这其中有很多影响因素，比如类的总数，常量池的大小和方法数量等。

同时，HotSpot 虚拟机的每种类型的垃圾回收器都需要特殊处理永久代中的元数据。将元数据从永久代剥离出来，不仅实现了对元空间的无缝管理，还可以简化 Full GC 以及对以后的并发隔离类元数据等方面进行优化。

移除永久代的影响

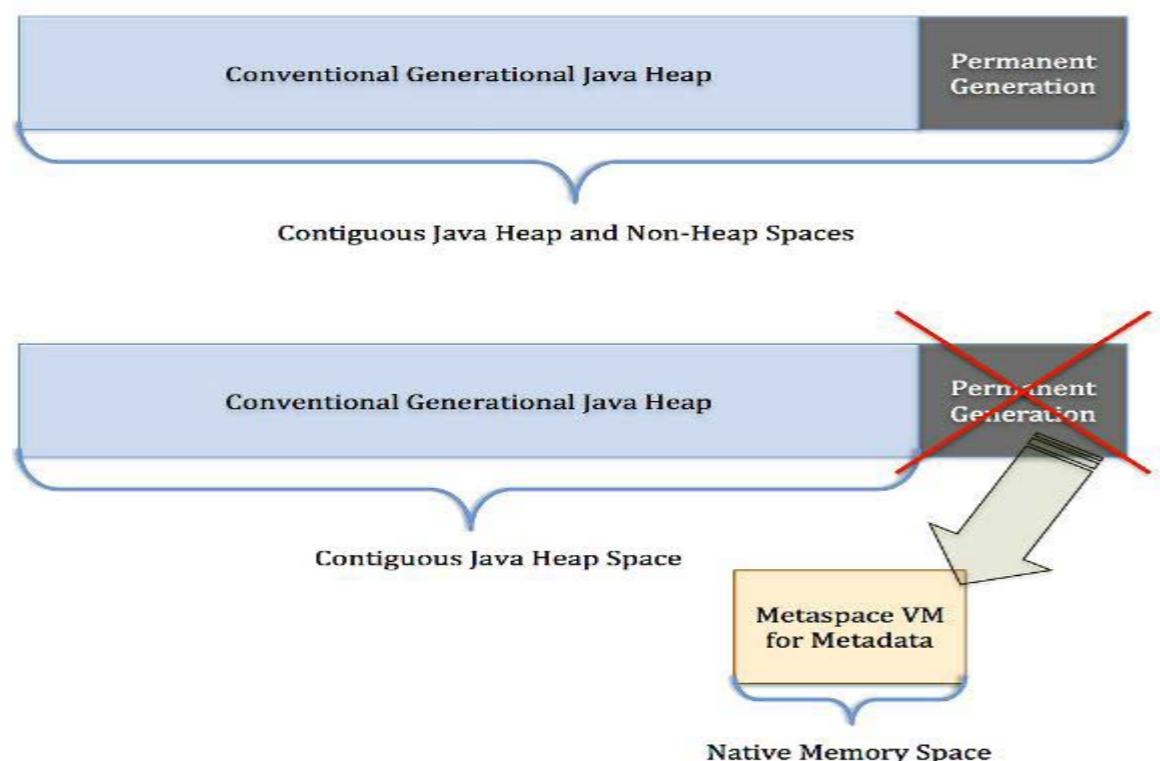
由于类的元数据分配在本地内存中，元空间的

最大可分配空间就是系统可用内存空间。因此，我们就不会遇到永久代存在时的内存溢出错误，也不会出现泄漏的数据移到交换区这样的事情。最终用户可以为元空间设置一个可用空间最大值，如果不进行设置，JVM 会自动根据类的元数据大小动态增加元空间的容量。

注意：永久代的移除并不代表自定义的类加载器泄露问题就解决了。因此，你还必须监控你的内存消耗情况，因为一旦发生泄漏，会占用你的大量本地内存，并且还可能导致交换区交换更加糟糕。

元空间内存管理

元空间的内存管理由元空间虚拟机来完成。先前，对于类的元数据我们需要不同的垃圾回收器进行处理，现在只需要执行元空间虚拟机的 C++ 代码即可完成。在元空间中，类和其元数据的生命周期与其对应的类加载器是相同的。话句话说，只要类加载器存活，其加载的类的元数据也是存活的，因而不会被回收掉。



我们从行文到现在提到的元空间稍微有点不严谨。准确的来说，每一个类加载器的存储区域都称作一个元空间，所有的元空间合在一起就是我们一直说的元空间。当一个类加载器被垃圾回收器标记为不再存活，其对应的元空间会被回收。在元空间的回收过程中没有重定位和压缩等操作。但是元空间内的元数据会进行扫描来确定 Java 引用。

元空间虚拟机负责元空间的分配，其采用的形式为组块分配。组块的大小因类加载器的类型而异。在元空间虚拟机中存在一个全局的空闲组块列表。当一个类加载器需要组块时，它就会从这个全局的组块列表中获取并维持一个自己的组块列表。当一个类加载器不再存活，那么其持有的组块将会被释放，并返回给全局组块列表。类加载器持有的组块又会被分成多个块，每一个块存储一个单元的元信息。组块中的块是线性分配（指针碰撞分配形式）。组块分配自内存映射区域。这些全局的虚拟内存映射区域以链表形式连接，一旦某个虚拟内存映射区域清空，这部分内存就会返回给操作系统。

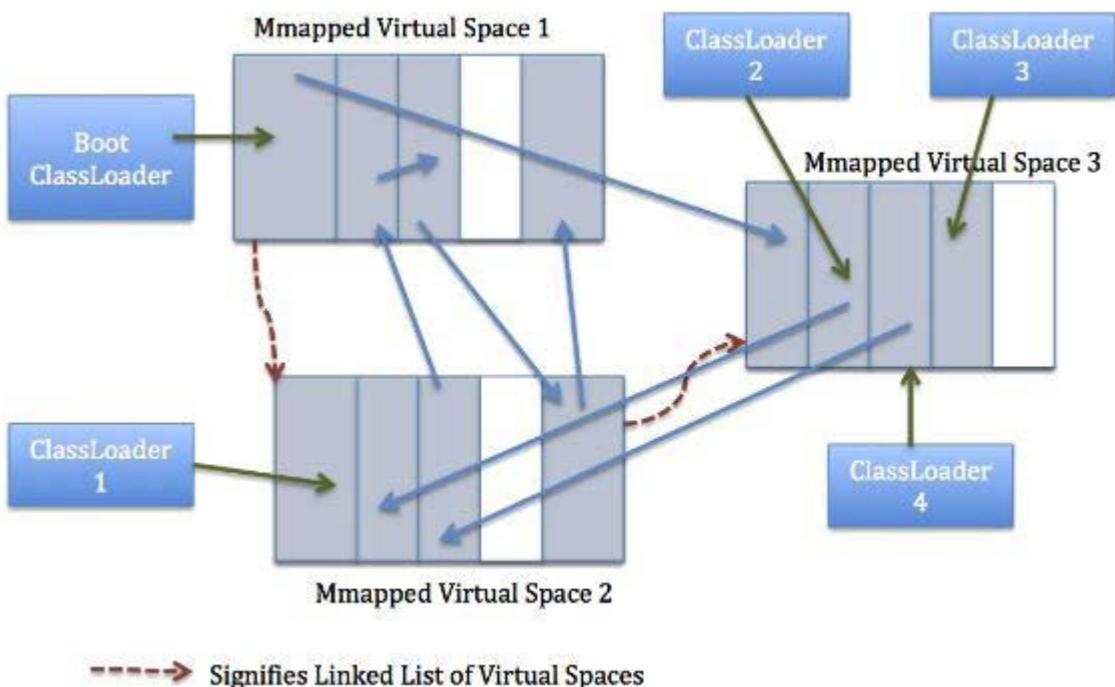
下图展示的是虚拟内存映射区域如何进行元组

块的分配。类加载器 1 和 3 表明使用了反射或者为匿名类加载器，他们使用了特定大小组块。而类加载器 2 和 4 根据其内部条目的数量使用小型或者中型的组块。

元空间调优与工具

正如上面提到的，元空间虚拟机控制元空间的增长。但是有些时候我们想限制其增长，比如通过显式在命令行中设置 -XX:MaxMetaspaceSize。默认情况下，-XX:MaxMetaspaceSize 的值没有限制，因此元空间甚至可以延伸到交换区，但是这时候当我们进行本地内存分配时将会失败。

对于一个 64 位的服务器端 JVM 来说，其默认的 -XX:MetaspaceSize 值为 21MB。这就是初始的高水位线。一旦触及到这个水位线，Full GC 将会被触发并卸载没有用的类（即这些类对应的类加载器不再存活），然后这个高水位线将会重置。新的高水位线的值取决于 GC 后释放了多少元空间。如果释放的空间不足，这个高水位线则上升。如果释放空间过多，则高水位线下降。如果初始化的高水位线设置过



低，上述高水位线调整情况会发生很多次。通过垃圾回收器的日志我们可以观察到 Full GC 多次调用。为了避免频繁的 GC，建议将 -XX:MetaspaceSize 设置为一个相对较高的值。

经过多次 GC 之后，元空间虚拟机自动调节高水位线，以此来推迟下一次垃圾回收到来。

有这样两个选项 -XX:MinMetaspaceFreeRatio 和 -XX:MaxMetaspaceFreeRatio，他们类似于 GC 的 FreeRatio 选项，用来设置元空间空闲比例的最大值和最小值。我们可以通过命令行对这两个选项设置对应的值。

下面是一些改进的工具，用来获取更多关于元空间的信息。

- jmap -clstats PID 打印类加载器数据。（-clstats 是 -permstat 的替代方案，在 JDK8 之前，-permstat 用来打印类加载器的数据）。代码段 1 输出就是 DaCapo Avrora benchmark 程序的类加载器数据。
- jstat -gc LVMID 用来打印元空间的信息，具体内容如下图所示。
- jcmand PID GC.class_stats 一个新的诊断命令，用来连接到运行的 JVM 并输出详尽的类元数据的柱状图。

注意：在 JDK6 build13 下，需要加上 -XX:+UnlockDiagnosticVMOptions 才能正确使用 jcmand 这个命令。（见代码段 2）

提示：如果想了解字段的更多信息，请访问[这里](#)。

```
$ jstat -gc <LVMID>
SOC S1C SOU S1U EC EU OC OU MC MU CCS CCG YGC YGCT FGC FGCT GCT
1024.0 1024.0 0.0 96.0 6144.0 2456.3 129536.0 2228.3 7296.0 6550.3 896.0 787.0 229 0.211 33 0.347 0.558
where, MC: Current Metaspace Capacity (KB); MU: Metaspace Utilization (KB)
```

代码段 1

```
001 $ jmap -clstats
002 Attaching to process ID 6476, please wait...
003 Debugger attached successfully.
004 Server compiler detected.
005 JVM version is 25.5-b02
006 finding class loader instances ..done.
007 computing per loader stat ..done.
008 please wait.. computing liveness.liveness analysis may be inaccurate ...
009 class_loader classes bytes parent_loader alive? type
010
011 655 1222734 null live
0x000000074004a6c0 0 0 0x000000074004a708 dead java/util/ResourceBundl
e$RBClassLoader@0x00000007c0053e20
012 0x000000074004a760 0 0 null dead sun/misc/
Launcher$ExtClassLoader@0x00000007c002d248
013 0x00000007401189c8 1 1471 0x00000007400752f8 dead sun/reflect/
DelegatingClassLoader@0x00000007c0009870
014 0x000000074004a708 116 316053 0x000000074004a760 dead sun/misc/
Launcher$AppClassLoader@0x00000007c0038190
015 0x00000007400752f8 538 773854 0x000000074004a708 dead org/dacapo/
harness/DacapoClassLoader@0x00000007c00638b0
016 total = 6 1310 2314112 N/A alive=1, dead=5 N/A
```

代码段 2

```
001 $ jcmand help GC.class_stats
002 9522:
003 GC.class_stats
004 Provide statistics about Java class meta data. Requires
-XX:+UnlockDiagnosticVMOptions.
005
006 Impact: High: Depends on Java heap size and content.
007
008 Syntax : GC.class_stats [options] []
009
010 Arguments:
011 columns : [optional] Comma-separated list of all the columns to show. If not
specified, the following columns are shown:
012 Options: (options must be specified using the or = syntax)
013 -all : [optional] Show all columns (BOOLEAN, false)
014 -csv : [optional] Print in CSV (comma-separated values) format for
spreadsheets (BOOLEAN, false)
015 -help : [optional] Show meaning of all the columns (BOOLEAN, false)
```

使用 jcmand 的示例输出：

```
001 $ jcmand GC.class_stats
002 7140:
003 Index Super InstBytes KlassBytes annotations CpAll MethodCount Bytecodes
MethodAll ROAll RWAll Total ClassName
004 1 -1 426416 480 0 0 0 0
0 24 576 600 [C
005 2 -1 290136 480 0 0 0 0
0 40 576 616 [Lavrora.arch.legacy.LegacyInstr;
006 3 -1 269840 480 0 0 0 0
0 24 576 600 [B
007 4 43 137856 648 0 19248 129 4886
25288 16368 30568 46936 java.lang.Class
008 5 43 136968 624 0 8760 94 4570
33616 12072 32000 44072 java.lang.String
009 6 43 75872 560 0 1296 7 149
1400 880 2680 3560 java.util.HashMap$Node
010 7 836 57408 608 0 720 3 69
1480 528 2488 3016 avrora.sim.util.MulticastFSMProbe
011 8 43 55488 504 0 680 1 31
440 280 1536 1816 avrora.sim.FiniteStateMachine$State
012 9 -1 53712 480 0 0 0 0
0 24 576 600 [Ljava.lang.Object;
013 10 -1 49424 480 0 0 0 0
0 24 576 600 [I
014 11 -1 49248 480 0 0 0 0
0 24 576 600 [Lavrora.sim.platform.ExternalFlash$Page;
015 ...
016 1300 1098 0 608 0 1744 10 290
1808 1176 3208 4384 sun.util.resources.OpenListResourceBundle
017 2244312 794288 2024 2260976 12801 561882
3135144 1906688 4684704 6591392 Total
018 34.0% 12.1% 0.0% 34.3% - 8.5%
47.6% 28.9% 71.1% 100.0%
019 Index Super InstBytes KlassBytes annotations CpAll MethodCount Bytecodes
MethodAll ROAll RWAll Total ClassName
```

存在的问题

前面已经提到，元空间虚拟机采用了组块分配的形式，同时区块的大小由类加载器类型决定。类信息并不是固定大小，因此有可能分配的空闲区块和类需要的区块大小不同，这种情况下可能导致碎片存在。元空间虚拟机目前并不支持压缩操作，所以碎片化是目前最大的问题。

作者简介

Monica Beckwith 是一位在硬件行业有着 10 多年经验的性能研究工程师。她目前在 Servergy 公司任性能架构师一职。该公司为一家提供高效服务器的创业公司。此外，Monica 曾在

Sun, Oracle 和 AMD 等公司致力于服务器端 JVM 优化。Monica 还是 JavaOne 2013 会议的演讲嘉宾。想要关注的可以在 twitter 上查找 @mon_beck。

查看英文原文：[Where Has the Java PermGen Gone?](#)

Netty案例集锦之多线程篇



作者 李林锋

TalkingData是国内领先的第三方独立移动互联网大数据平台，提供基于移动互联网的数据产品及数据服务。

数据产品

- App Analytics**: 移动应用统计分析
- Game Analytics**: 移动游戏运营分析
- Ad Tracking**: 移动广告效果监测
- Mobile DMP**: 移动数据管理平台

服务体系

- 数据服务** → **Insight**: 移动互联网综合数据服务
- 企业服务** → **Total Solution**: 企业级移动大数据解决方案

微信公众账号: TalkingData
公司网站: www.talkingdata.com
博客地址: blog.talkingdata.com
新浪微博: @TalkingData

1. Netty 案例集锦系列文章介绍

1.1. Netty 的特点

Netty 入门比较简单，主要原因有如下几点：

- Netty 的 API 封装比较简单，将复杂的网络通信通过 BootStrap 等工具类做了二次封装，用户使用起来比较简单；
- Netty 源码自带的 Demo 比较多，通过 Demo 可以很快入门；
- Netty 社区资料、相关学习书籍也比较多，学习资料比较丰富。

但是很多入门之后的 Netty 学习者遇到了很多困惑，例如不知道在实际项目中如何使用 Netty、遇到 Netty 问题之后无从定位等，这

些问题严重制约了对 Netty 的深入掌握和实际项目应用。

Netty 相关问题比较难定位的主要原因如下：

- NIO 编程自身的复杂性，涉及到大量 NIO 类库、Netty 自身封装的类库等，当你需要打开黑盒定位问题时，必须对这些类库了如指掌；否则即便定位到问题所在，也不知道所以然，更无法修复；
- Netty 复杂的多线程模型，用户在实际使用 Netty 时，会涉及到 Netty 自己封装的线程组、线程池、NIO 线程，以及业务线程，通信链路的创建、I/O 消息的读写会涉及到复杂的线程切换，这会让初学者云山雾绕，调试起来非常痛苦，甚至都不知道从哪里调试；
- Netty 版本的跨度大，从实际商用情况看，涉及到了 Netty 3.X、4.X 和 5.X 等多个版本，

每个 Major 版本之间特性变化非常大，即便是 Minor 版本都存在一些差异，这些功能特性和类库差异会给使用者带来很多问题，版本升级之后稍有不慎就会掉入陷阱。

1.2. 案例来源

Netty 案例集锦的案例来源于作者在实际项目中遇到的问题总结、以及 Netty 社区网友的反馈，大多数案例都来源于实际项目，也有少部分是读者在学习 Netty 中遭遇的比较典型的问题。

1.3. 多线程篇

学习和掌握 Netty 多线程模型是个难点，在实

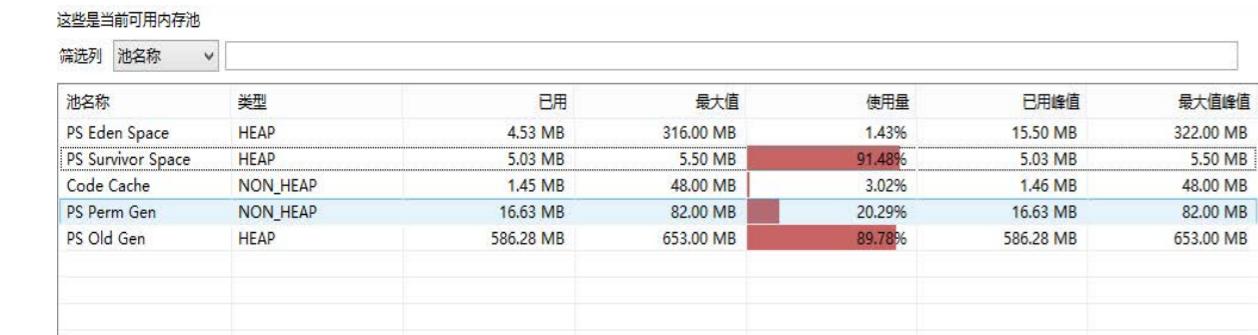
际项目中如何使用好 Netty 多线程更加困难，很多网上问题和事故都来源于对 Netty 线程模型了解不透彻所致。鉴于此，Netty 案例集锦系列就首先从多线程方面开始。

2. Netty 3 版本升级遭遇内存泄漏案例

2.1. 问题描述

业务代码升级 Netty 3 到 Netty4 之后，运行一段时间，Java 进程就会宕机，查看系统运行日志发现系统发生了内存泄露（见下图）。

对内存进行监控（切换使用堆内存池，方便对内存进行监控），发现堆内存一直飙升。



2.2. 问题定位

使用 jmap -dump:format=b,file=netty.bin PID 将堆内存 dump 出来，通过 IBM 的 HeapAnalyzer 工具进行分析，发现 ByteBuf 发生了泄露。

因为使用了 Netty 4 的内存池，所以首先怀疑是不是申请的 ByteBuf 没有被释放导致？查看代码，发现消息发送完成之后，Netty 底层已经调用 ReferenceCountUtil.release(message) 对内存进行了释放。这是怎么回事呢？难道 Netty 4.X 的内存池有 Bug，调用 release 操作释放内存失败？

考虑到 Netty 内存池自身 Bug 的可能性不大，首先从业务的使用方式入手分析：

- 内存的分配是在业务代码中进行，由于使用到了业务线程池做 I/O 操作和业务操作的隔离，实际上内存是在业务线程中分配的；
- 内存的释放操作是在 outbound 中进行，按照 Netty 3 的线程模型，downstream（对应 Netty 4 的 outbound，Netty 4 取消了 upstream 和 downstream）的 handler 也是

```
final ThreadLocal<PoolThreadCache> threadCache = new ThreadLocal<PoolThreadCache>() {
    private final AtomicInteger index = new AtomicInteger();
    @Override
    protected PoolThreadCache initialValue() {
        final int idx = index.getAndIncrement();
        final PoolArena<byte[]> heapArena;
        final PoolArena<ByteBuffer> directArena;
        //.....此处代码省略
        return new PoolThreadCache(heapArena, directArena);
    }
}
```

由业务调用者线程执行的，也就是说申请和释放放在同一个业务线程中进行。初次排查并没有发现导致内存泄露的根因，继续分析 Netty 内存池的实现原理。

- Netty 内存池实现原理分析：查看 Netty 的内存池分配器 PooledByteBufAllocator 的源码实现，发现内存池实际是基于线程上下文实现的，相关代码如下。

也就是说内存的申请和释放必须在同一线程上下文中，不能跨线程。跨线程之后实际操作的就不是同一块儿内存区域，这会导致很多严重的问题，内存泄露便是其中之一。内存 A 线程申请，切换到 B 线程释放，实际是无法正确回收的。

2.3. 问题根因

Netty 4 修改了 Netty 3 的线程模型：在 Netty 3 的时候，upstream 是在 I/O 线程里执行的，而 downstream 是在业务线程里执行。当 Netty 从网络读取一个数据报投递给业务 handler 的时候，handler 是在 I/O 线程里执行；而当我们在业务线程中调用 write 和 writeAndFlush 向网络发送消息的时候，handler 是在业务线

程里执行，直到最后一个 Header handler 将消息写入到发送队列中，业务线程才返回。Netty4 修改了这一模型，在 Netty 4 里 inbound(对应 Netty 3 的 upstream) 和 outbound(对应 Netty 3 的 downstream) 都是在 NioEventLoop(I/O 线程) 中执行。当我们在业务线程里通过 ChannelHandlerContext.write 发送消息的时候，Netty 4 在将消息发送事件调度到 ChannelPipeline 的时候，首先将待发送的消息封装成一个 Task，然后放到 NioEventLoop 的任务队列中，由 NioEventLoop 线程异步执行。后续所有 handler 的调度和执行，包括消息的发送、I/O 事件的通知，都由 NioEventLoop 线程负责处理。

在本案例中，ByteBuf 在业务线程中申请，在后续的 ChannelHandler 中释放，ChannelHandler 是由 Netty 的 I/O 线程 (EventLoop) 执行的，因此内存的申请和释放不在同一个线程中，导致内存泄漏。

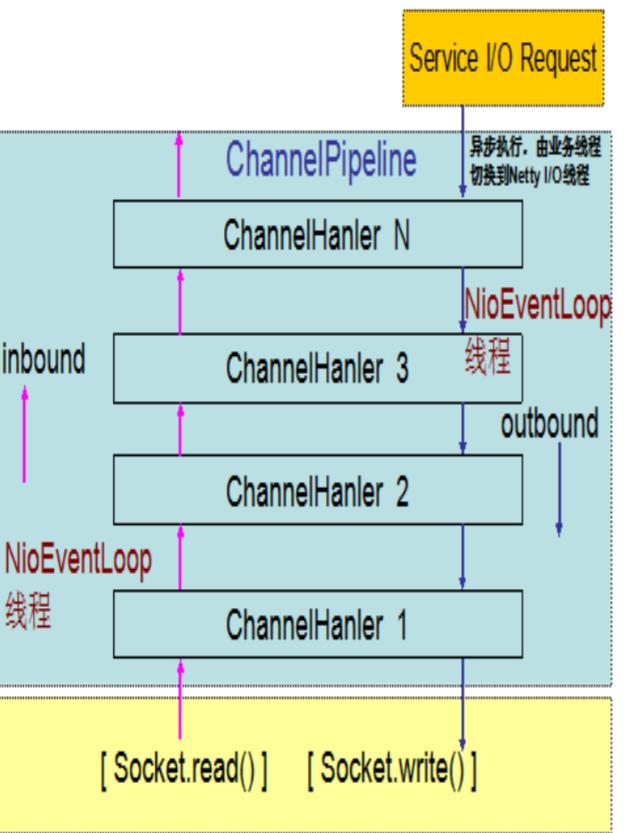
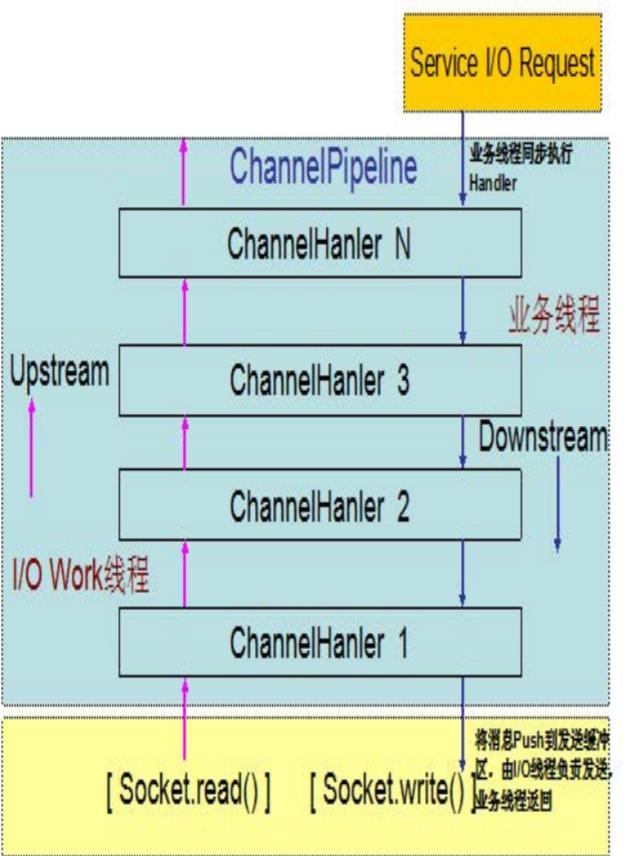
Netty 3 和 Netty 4 的 I/O 事件处理流程（见右图）。

2.4. 案例总结

Netty 4.X 版本新增的内存池确实非常高效，但是如果使用不当则会导致各种严重的问题。诸如内存泄露这类问题，功能测试并没有异常，如果相关接口没有进行压测或者稳定性测试而直接上线，则会导致严重的线上问题。

内存池 PooledByteBuf 的使用建议：

- 申请之后一定要记得释放，Netty 自身 Socket 读取和发送的 ByteBuf 系统会自动释放，用户不需要做二次释放；如果用户使用 Netty 的内存池在应用中做 ByteBuf 的对象池使用，则需要自己主动释放；



- 避免错误的释放：跨线程释放、重复释放等都是非法操作，要避免。特别是跨线程申请和释放，往往具有隐蔽性，问题定位难度较大；

- 防止隐式的申请和分配：之前曾经发生过一个案例，为了解决内存池跨线程申请和释放问题，有用户对内存池做了二次包装，以实现多线程操作时，内存始终由包装的管理线程申请和释放，这样可以屏蔽用户业务线程模型和访问方式的差异。谁知运行一段时间之后再次发生了内存泄露，最后发现原来调用 ByteBuf 的 write 操作时，如果内存容量不足，会自动进行容量扩展。扩展操作由业务线程执行，这就绕过了内存池管理线程，发生了“引用逃逸”；

- 避免跨线程申请和使用内存池，由于存在“引用逃逸”等隐式的内存创建，实际上跨线程申请和使用内存池是非常危险的行为。尽管从技术角度看可以实现一个跨线程协调的内存池机制，甚至重写 PooledByteBufAllocator，但是这无疑会增加很多复杂性，通常也使用不到。如果确实存在跨线程的 ByteBuf 传递，而且无法保证 ByteBuf 在另一个线程中会重新分配大小等操作，最简单保险的方式就是在线程切换点做一次 ByteBuf 的拷贝，但这会造成性能下降。

比较好的一种方案就是如果存在跨线程的 ByteBuf 传递，对 ByteBuf 的写操作要在分配线程完成，另一个线程只能做读操作。操作完成之后发送一个事件通知分配线程，由分配线程执行内存释放操作。

3. Netty 3 版本升级性能下降案例

3.1. 问题描述

业务代码升级 Netty 3 到 Netty4 之后，并没

有给产品带来预期的性能提升，有些甚至还发生了非常严重的性能下降，这与 Netty 官方给出的数据并不一致。

Netty 官方性能测试对比数据：我们比较了两个分别建立在 Netty 3 和 4 基础上 echo 协议服务器。（Echo 非常简单，这样，任何垃圾的产生都是 Netty 的原因，而不是协议的原因）。我使它们服务于相同的分布式 echo 协议客户端，来自这些客户端的 16384 个并发连接重复发送 256 字节的随机负载，几乎使千兆以太网饱和。

根据测试结果，Netty 4：

- GC 中断频率是原来的 1/5: 45.5 vs. 9.2 次 / 分钟；
- 垃圾生成速度是原来的 1/5: 207.11 vs 41.81 MiB / 秒。

3.2. 问题定位

线程名称	线程状态	受限计数
RMI TCP Connection(idle)	TIMED_WAITING	0
RMI TCP Connection(112)-192.168.1.103	RUNNABLE	0
RMI TCP Connection(105)-192.168.1.103	TIMED_WAITING	5
JMX server connection timeout 82	TIMED_WAITING	37
RMI TCP Connection(104)-192.168.1.103	RUNNABLE	0
nioEventLoopGroup-3-1	RUNNABLE	9

选定线程的堆栈跟踪
选定线程 21:49:55 的堆栈跟踪

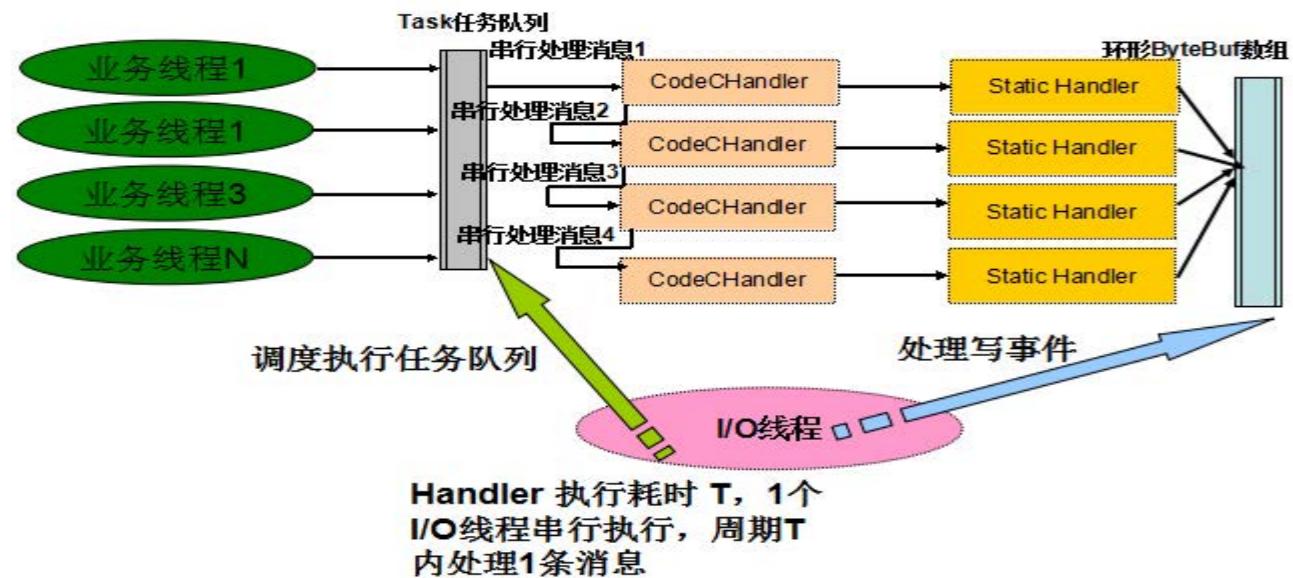
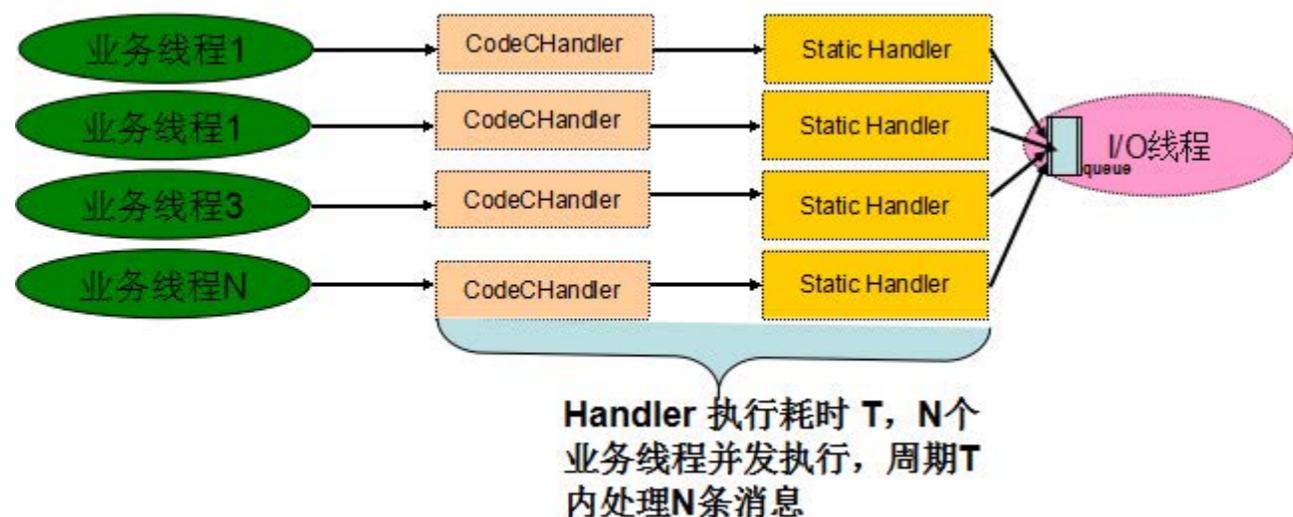
nioEventLoopGroup-3-1 [23] (RUNNABLE) sun.nio.ch.WindowsSelectorImpl\$SubSelector.poll 行: 不可用 [本地方法]
sun.nio.ch.WindowsSelectorImpl\$SubSelector.poll 行: 296
sun.nio.ch.WindowsSelectorImpl\$SubSelector.access\$400 行: 278
sun.nio.ch.WindowsSelectorImpl.doSelect 行: 159
sun.nio.ch.SelectorImpl.lockAndDoSelect 行: 87
sun.nio.ch.SelectorImpl.select 行: 98
io.netty.channel.nio.NioEventLoop.select 行: 596
io.netty.channel.nio.NioEventLoop.run 行: 306

通过对热点方法的分析，发现在消息发送过程中，有两处热点：

1. 消息发送性能统计相关 Handler；
2. 编码 Handler。

对使用 Netty 3 版本的业务产品进行性能对比测试，发现上述两个 Handler 也是热点方法。既然都是热点，为啥切换到 Netty4 之后性能下降这么厉害呢？

通过方法的调用树分析发现了两个版本的差异：在 Netty 3 中，上述两个热点方法都是由业务线程负责执行；而在 Netty 4 中，则是由 NioEventLoop(I/O) 线程执行。对于某个链路，业务是拥有多个线程的线程池，而



3.3. 问题总结

该问题的根因还是由于 Netty 4 的线程模型变更引起，线程模型变更之后，不仅影响业务的功能，甚至对性能也会造成很大的影响。

对 Netty 的升级需要从功能、兼容性和性能等多个角度进行综合考虑，切不可只盯着 API 变更这个芝麻，而丢掉了性能这个西瓜。API 的变更会导致编译错误，但是性能下降却隐藏于无形之中，稍不留意就会中招。

对于讲究快速交付、敏捷开发和灰度发布的互联网应用，升级的时候更应该要当心。

4. Netty 业务 Handler 接收不到消息案例

4.1. 问题描述

我的服务碰到一个问题，经常有请求上来到了 MessageDecoder 就结束了，没有继续往 LogicServerHandler 里面送，觉得很奇怪，是不是线程池满了？我想请教：

```
/*
 * public class ServerChannelInitializer extends ChannelInitializer<SocketChannel> {
 *     private static final EventExecutorGroup executor = new DefaultEventExecutorGroup(500);
 *
 *     @Override
 *     protected void initChannel(SocketChannel sc) throws Exception {
 *         ChannelPipeline pipeline = sc.pipeline();
 *
 *         // logs
 *         pipeline.addLast("logger", new LoggingHandler(LogLevel.DEBUG));
 *
 *         // readTimeoutHandler
 *         pipeline.addLast("readTimeoutHandler", new ReadTimeoutHandler(30));
 *
 *         // Decoders
 *         pipeline.addLast("frameDecoder", new LengthFieldBasedFrameDecoder(2048, 0, 2, 0, 2));
 *         pipeline.addLast("isoMessageDecoder", new MessageDecoder());
 *
 *         // Encoder
 *         pipeline.addLast("frameEncoder", new LengthFieldPrepender(2));
 *         pipeline.addLast("isoMessageEncoder", new MessagePrepender());
 *
 *         // and then business logic
 *         pipeline.addLast(executor, "logicHandler", new LogicServerHandler());
 *     }
 * }
```

- netty 5 如何打印 executor 线程的占用情况，如空闲线程数？

- executor 设置的大小一般如何进行计算的？

业务代码示例如下所示。

4.2. 问题定位

从服务端初始化代码来看，并没有什么问题，业务 LogicServerHandler 没有接收到消息，有如下几种可能：

- 客户端并没有将消息发送到服务端，可以在服务端 LoggingHandler 中打印日志查看；
- 服务端部分消息解码发生异常，导致消息被丢弃 / 忽略，没有走到 LogicServerHandler 中；
- 执行业务 Handler 的 DefaultEventExecutor 中的线程太繁忙，导致任务队列积压，长时间得不到处理。

通过抓包结合日志分析，可能导致问题的原因 1 和 2 排除，需要继续对可能原因 3 进行排查。

Netty 5 如何打印 executor 线程的占用情况，如空闲线程数？回答这些问题，首先要了解 Netty 的线程组和线程池机制。

Netty 的 EventExecutorGroup 实际就是一组 EventExecutor，它的定义如下：

```
public abstract class MultithreadEventExecutorGroup extends AbstractEventExecutorGroup {
    private final EventExecutor[] children;
    private final Set<EventExecutor> readonlyChildren;
    private final AtomicInteger childIndex = new AtomicInteger();
}
```

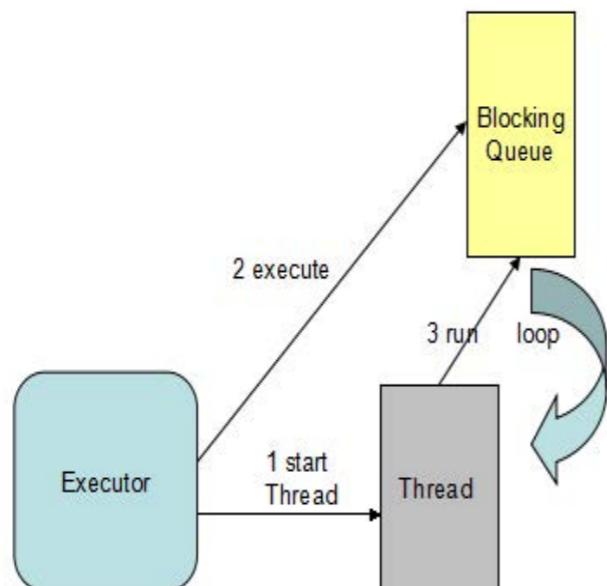
通常通过它的 next 方法从线程组中获取一个线程池，代码如下：

```
@Override
public EventExecutor next() {
    return children[Math.abs(childIndex.getAndIncrement() % children.length)];
}
```

Netty EventExecutor 典型实现有两个：DefaultEventExecutor 和 SingleThreadEventLoop，在本案例中，因为使用的是 DefaultEventExecutorGroup，所以实际执行业务 Handler 线程池就是 DefaultEventExecutor，它继承自 SingleThreadEventExecutor，从名称就可以看出它是个单线程的线程池。工作原理如下：

1. DefaultEventExecutor 聚合 JDK 的 Executor 和 Thread，首次执行 Task 的时候启动线程，将线程池状态修改为运行态；
2. Thread run 方法循环从队列中获取 Task 执行，如果队列为空，则同步阻塞，线程无限循环执行，直到接收到退出信号。

用户想通过 Netty 提供的 DefaultEventExecutorGroup 来并发执行业务 Handler，但实际上却是单线程 SingleThreadEventExecutor 在串行执行业务逻辑，当服务端消息接收速度超过业务逻辑执行速度时，就会导致业务消息积压在



```
Set<EventExecutor> executorGroups = ctx.executor().parent().children();
```

```
for(EventExecutor ext : executorGroups)
```

```
{
```

```
int size = ((SingleThreadEventExecutor)ext).pendingTasks();
```

```
logger.info(ext.toString() + " pending size in queue is : --> " + size);
```

```
}
```

```
七月 23, 2015 11:07:52 下午 io.netty.example.echo.EchoServerHandler printExecutorGroupInfo
信息: io.netty.util.concurrent.DefaultEventExecutor@5a2023f3 pending size in queue is : --> 0
七月 23, 2015 11:07:52 下午 io.netty.example.echo.EchoServerHandler printExecutorGroupInfo
信息: io.netty.util.concurrent.DefaultEventExecutor@741854be pending size in queue is : --> 0
七月 23, 2015 11:07:52 下午 io.netty.example.echo.EchoServerHandler printExecutorGroupInfo
信息: io.netty.util.concurrent.DefaultEventExecutor@1d652020 pending size in queue is : --> 0
```

4.3. 问题总结

事实上，Netty 为了防止多线程执行某个 Handler (Channel) 引起线程安全问题，实际只有一个线程会执行某个 Handler，代码如下所示。

需要指出的是，SingleThreadEventExecutor 的 pendingTasks 可能是个耗时的操作，因此调用的时候需要注意。

```
// Pin one of the child executors once and remember it so that the same child executor
// is used to fire events for the same channel.
ChannelHandlerInvoker invoker = childInvokers.get(group);
if (invoker == null) {
    EventExecutor executor = group.next();
    if (executor instanceof EventLoop) {
        invoker = ((EventLoop) executor).asInvoker();
    } else {
        invoker = new DefaultChannelHandlerInvoker(executor);
    }
    childInvokers.put(group, invoker);
}

/**
 * Return the number of tasks that are pending for processing.
 *
 * <strong>Be aware that this operation may be expensive as it depends on
 * the internal implementation of the
 * SingleThreadEventExecutor. So use it was care!</strong>
*/
```

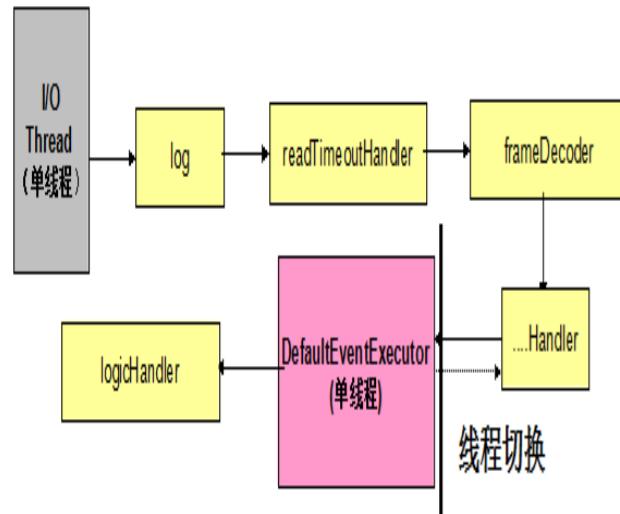
实际就像 JDK 的线程池，不同的业务场景、硬件环境和性能标就会有不同的配置，无法给出标准的答案。需要进行实际测试、评估和调优来灵活调整。

最后再总结回顾下问题，对于案例中的代码，实际上在使用单线程处理某个 Handler 的 LogicServerHandler，作者可能想并发多线程执行这个 Handler，提升业务处理性能，但实

际并没有达到设计效果。

如果业务性能存在问题，并不奇怪，因为业务实际是单线程串行处理的！当然，如果业务存在多个 Channel，则每个 / 多个 Channel 会对应一个线程（池），也可以实现多线程处理，这取决于客户端的接入数。

案例中代码的线程处理模型如下图所示（单个链路模型）。



5. Netty 4 ChannelHandler 线程安全疑问

5.1. 问题咨询

我有一个非线程安全的类 ThreadUnsafeClass，这个类会在 channelRead 方法中被调用。我下

```

public class MyHandler extends ChannelInboundHandlerAdapter {

private ThreadUnsafeClass unsafe = new ThreadUnsafeClass();

public void channelRead(ChannelHandlerContext ctx, Object msg) {
//下面的代码是否ok?
unsafe.doSomething(ctx, msg);
}
...
}

```

面这样的调用方法在多线程环境下安全吗？谢谢！

5.2. 解答

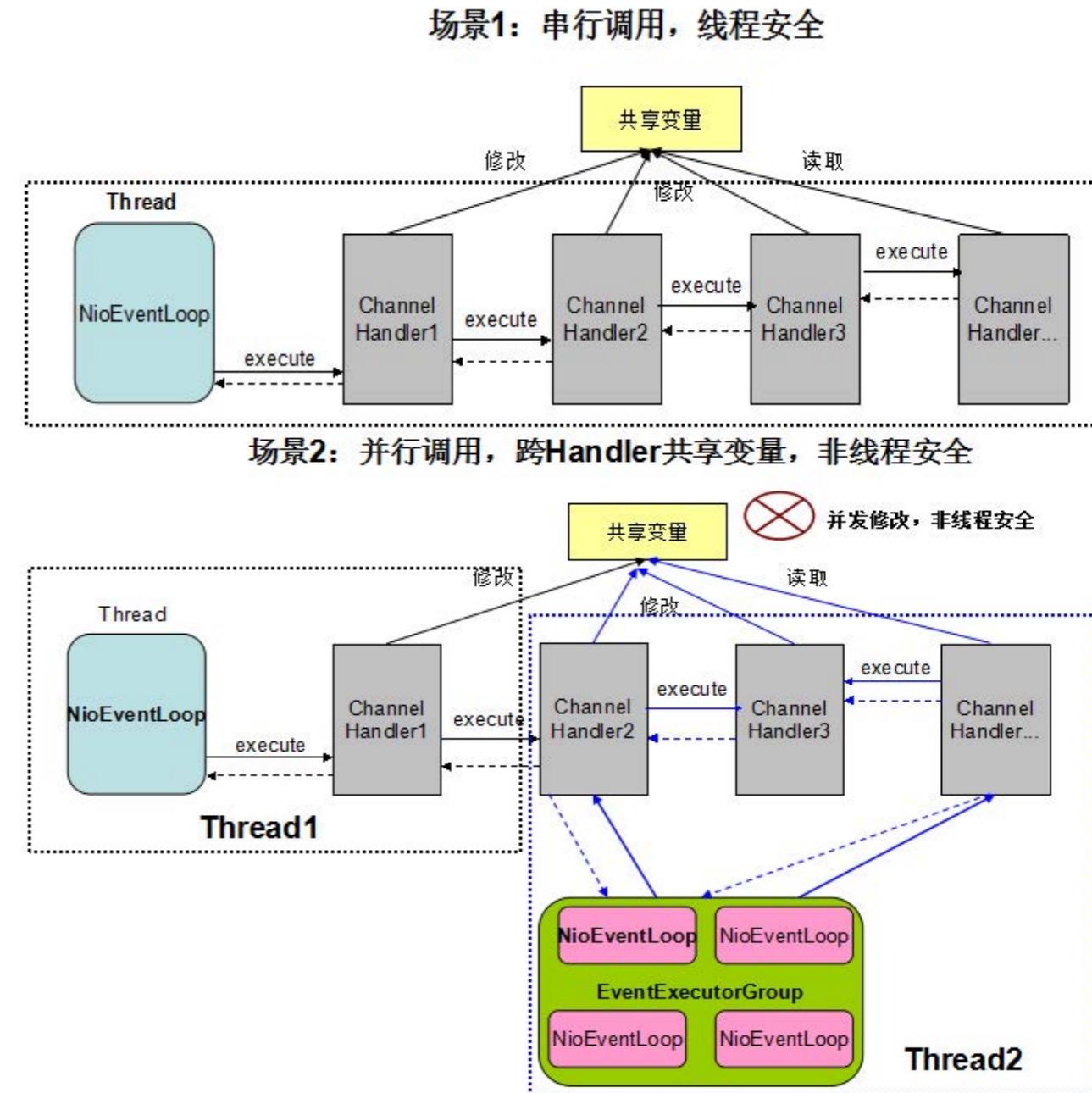
Netty 4 优化了 Netty 3 的线程模型，其中一个非常大的优化就是用户不需要再担心 ChannelHandler 会被并发调用，总结如下：

- ChannelHandler's 的方法不会被 Netty 并发调用；
- 用户不再需要对 ChannelHandler 的各个方面做同步保护；
- ChannelHandler 实例不允许被多次添加到 ChannelPipeline 中，否则线程安全将得不到保证；
- 根据上述分析，MyHandler 的 channelRead 方法不会被并发调用，因此不存在线程安全问题。

5.3. 一些特例

ChannelHandler 的线程安全存在几个特例，总结如下：

- 如果 ChannelHandler 被注解为 @Sharable，全局只有一个 handler 实例，它会被多个 Channel 的 Pipeline 共享，会被多线程并发调用，因此它不是线程安全的；



- 如果存在跨 ChannelHandler 的实例级变量共享，需要特别注意，它可能不是线程安全的。

非线程安全的跨 ChannelHandler 变量原理如下图。Netty 支持在添加 ChannelHandler 的时候，指定执行该 Handler 的 EventExecutorGroup，这就意味着在整个 ChannelPipeline 执行过程中，可能会发生线程切换。此时，如果同一个对象在多个 ChannelHandler 中被共享，可能会被多线程并发操作。（见下图）

6. 作者简介

李林锋，2007 年毕业于东北大学，2008 年进入华为公司从事高性能通信软件的设计和开发工作，有 7 年 NIO 设计和开发经验，精通 Netty、Mina 等 NIO 框架和平台中间件，现任华为软件平台架构部架构师，《Netty 权威指南》作者。目前从事华为下一代中间件和 PaaS 平台的架构设计工作。

联系方式：新浪微博 Nettying 微信：Nettying 微信公众号：Netty 之家。



[北京站]

2015年12月18日-19日
北京·国际会议中心
www.archsummit.com

10月30日前**8折优惠** 立减1360元
团购享受更多优惠

大会介绍

ArchSummit全球架构师峰会是InfoQ中国团队推出的面向高端技术管理者、架构师的技术大会，参会者中超过50%拥有8年以上的工
作经验。ArchSummit秉承“实践第一、案例为主”的原则，展示新技术在行业应用中的最新实践，技术在企业转型中的加速作用，帮助
企业技术管理者、CTO、架构师做好技术选型、技术团队组建与管理，并确立技术对于产品和业务的关键作用。

重量级嘉宾齐助阵(排名不分先后)



周爱民

豌豆荚架构师



段念

宜人贷CTO



崔宝秋

小米首席架构师
小米云负责人



周秋野

民生电商技术总监
小米云负责人



张迅迪

阿里巴巴
高级安全运营专家



李爽

美团云计算部总经理



晁晓娟

新华电商CTO/总裁助理



陈滢

慧科教育集团高级副总裁
慧科教育研究院院长



高自光

小米智能硬件平台
部门负责人



田琪

京东云数据库技术负责人

9大专题论坛倾情巨献

- ▶ 互联网+在线教育
- ▶ 云服务架构探索
- ▶ 移动应用架构
- ▶ 信息安全保障最佳实践
- ▶ 物联网+智能设备
- ▶ 高效运维案例剖析
- ▶ 研发体系构建管理
- ▶ 新金融形态的“颠覆”与创新
- ▶ 新型电商: O2O及其他新型电商模式



垂询电话: 010-89880682

QQ 咨询: 2332883546

E-mail: arch@cn.infoq.com

更多精彩内容, 请持续关注archsummit.com



Kafka设计解析（四）：Kafka Consumer解析

作者 李林锋

High Level Consumer

很多时候，客户程序只是希望从 Kafka 读取数据，不太关心消息 offset 的处理。同时也希望提供一些语义，例如同一条消息只被某一个 Consumer 消费（单播）或被所有 Consumer 消费（广播）。因此，Kafka High Level Consumer 提供了一个从 Kafka 消费数据的高层抽象，从而屏蔽掉其中的细节并提供丰富的语义。

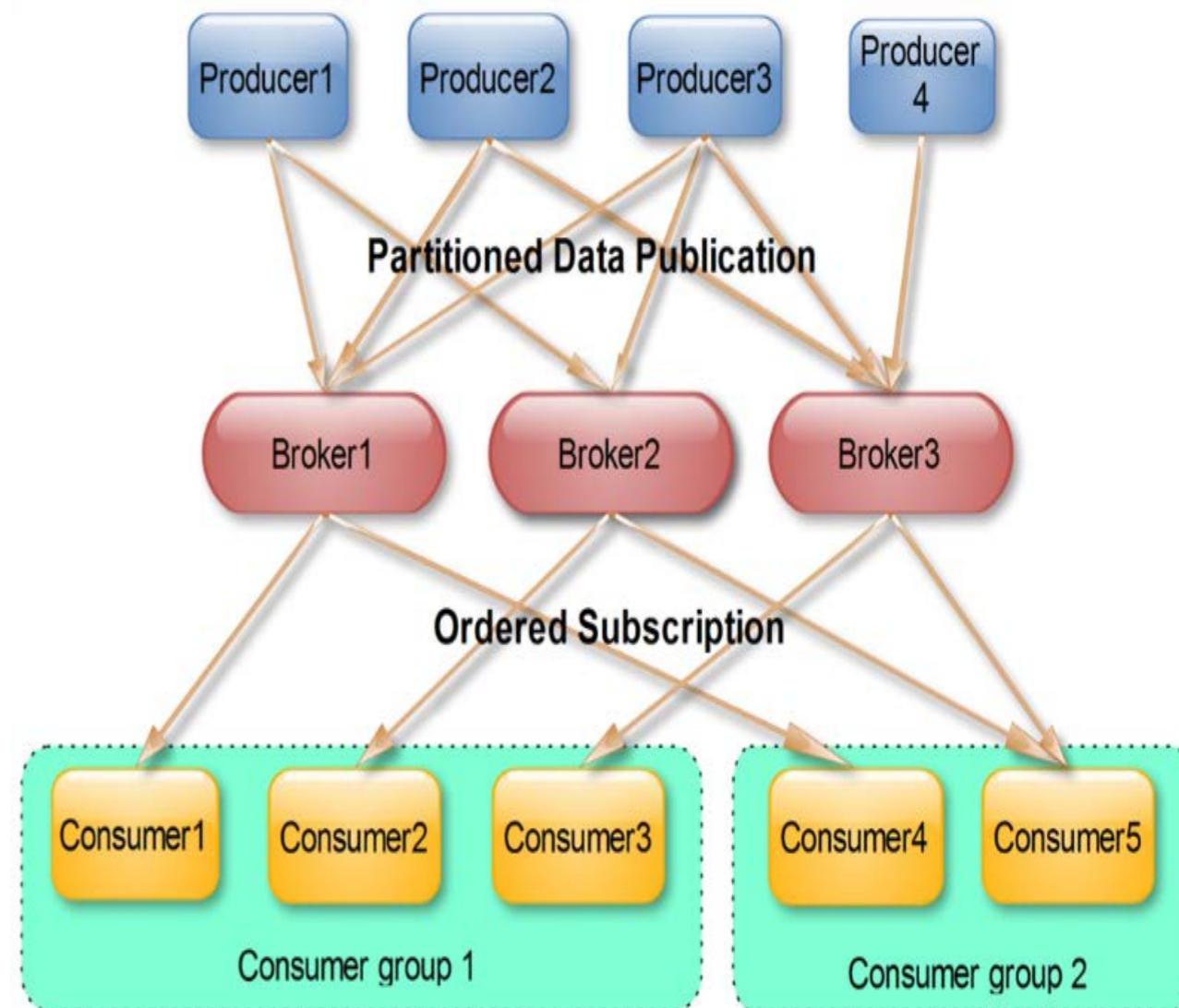
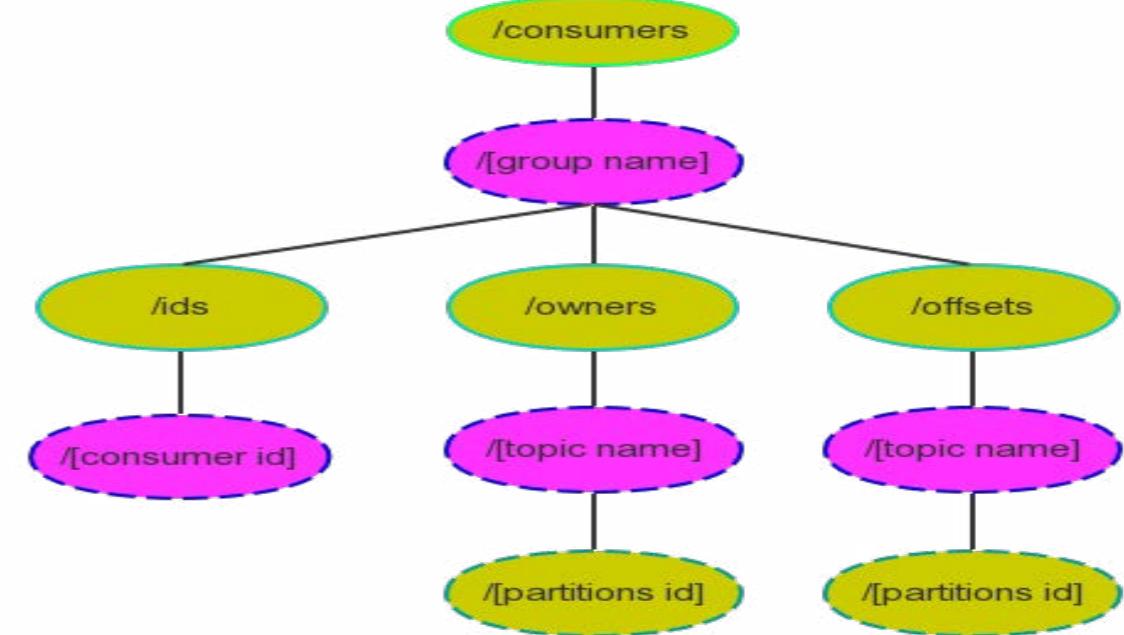
Consumer Group

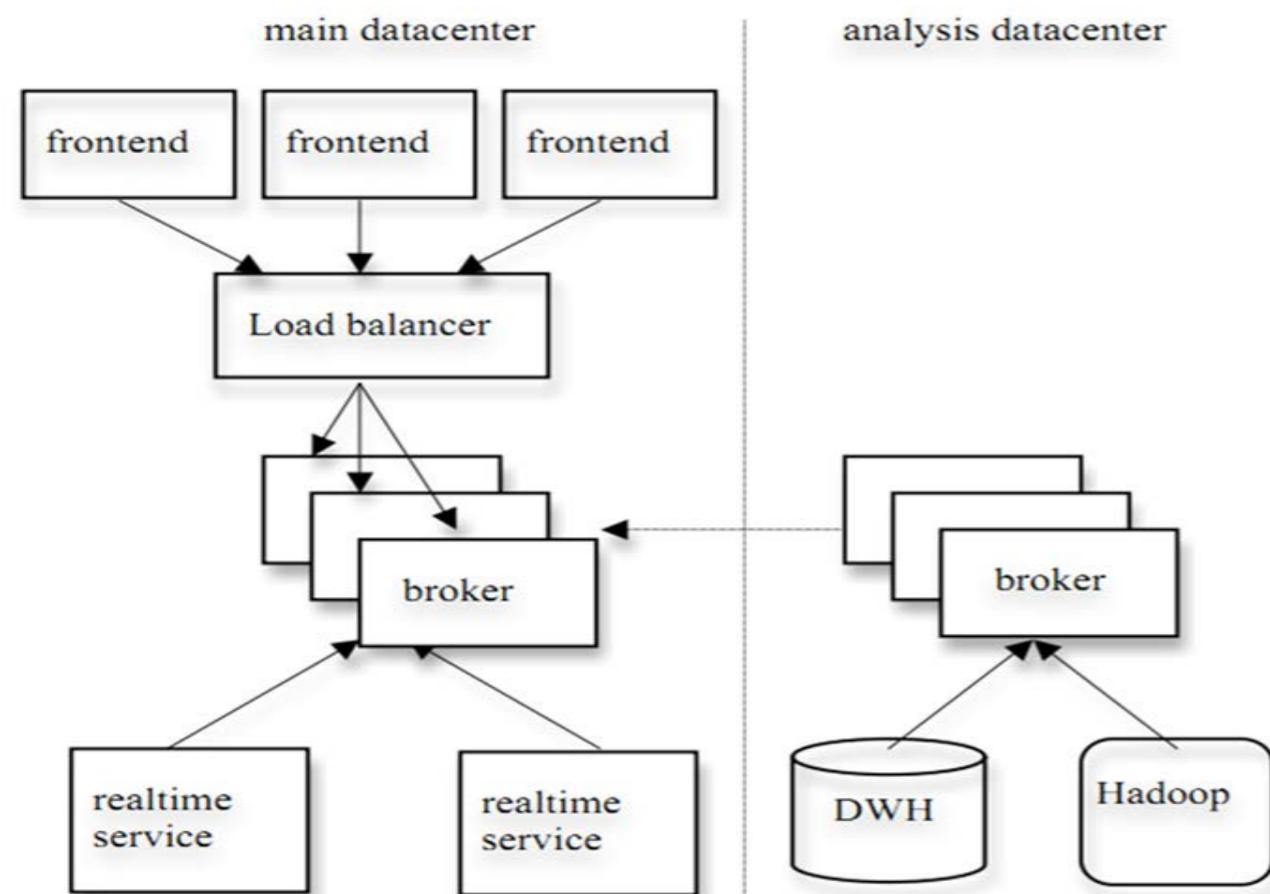
High Level Consumer 将从某个 Partition 读取的最后一条消息的 offset 存于 ZooKeeper 中（Kafka 从 [0.8.2 版本](#) 开始同时支持将 offset 存于 Zookeeper 中与将 offset 存于专用的 Kafka Topic 中）。这个 offset 基于客户程序提供给 Kafka 的名字来保存，这个名字被称为 Consumer Group。Consumer Group 是整个 Kafka 集群全局的，而非某个 Topic 的。每一个 High Level Consumer 实例都属于一个 Consumer Group，若不指定则属于默认的

Group。ZooKeeper 中 Consumer 相关节点如图所示。

很多传统的 Message Queue 都会在消息被消费完后将消息删除，一方面避免重复消费，另一方面可以保证 Queue 的长度比较短，提高效率。而如上文所述，Kafka 并不删除已消费的消息，为了实现传统 Message Queue 消息只被消费一次的语义，Kafka 保证每条消息在同一个 Consumer Group 里只会被某一个 Consumer 消费。与传统 Message Queue 不同的是，Kafka 还允许不同 Consumer Group 同时消费同一条消息，这一特性可以为消息的多元化处理提供支持。

实际上，Kafka 的设计理念之一就是同时提供离线处理和实时处理。根据这一特性，可以使用 Storm 这种实时流处理系统对消息进行实时在线处理，同时使用 Hadoop 批处理系统进行离线处理，还可以同时将数据实时备份到另一个数据中心，只需要保证这三个操作所使用的 Consumer 在不同的 Consumer Group 即可。下图展示了 Kafka 在 LinkedIn 的简化部署模型。





为了更清晰展示 Kafka Consumer Group 的特性，笔者进行了一项测试。创建一个 Topic (名为 topic1)，再创建一个属于 group1 的 Consumer 实例，并创建三个属于 group2 的 Consumer 实例，然后通过 Producer 向 topic1 发送 Key 分别为 1, 2, 3 的消息。结果发现属于 group1 的 Consumer 收到了所有的这三条消息，同时 group2 中的 3 个 Consumer 分别收到了 Key 为 1, 2, 3 的消息，如下图所示。

注 图中每个黑色区域代表一个 Consumer 实例，每个实例只创建一个 MessageStream。实际上，本实验将 Consumer 应用程序打成 jar 包，并在 4 个不同的命令行终端中传入不同的参数运行。

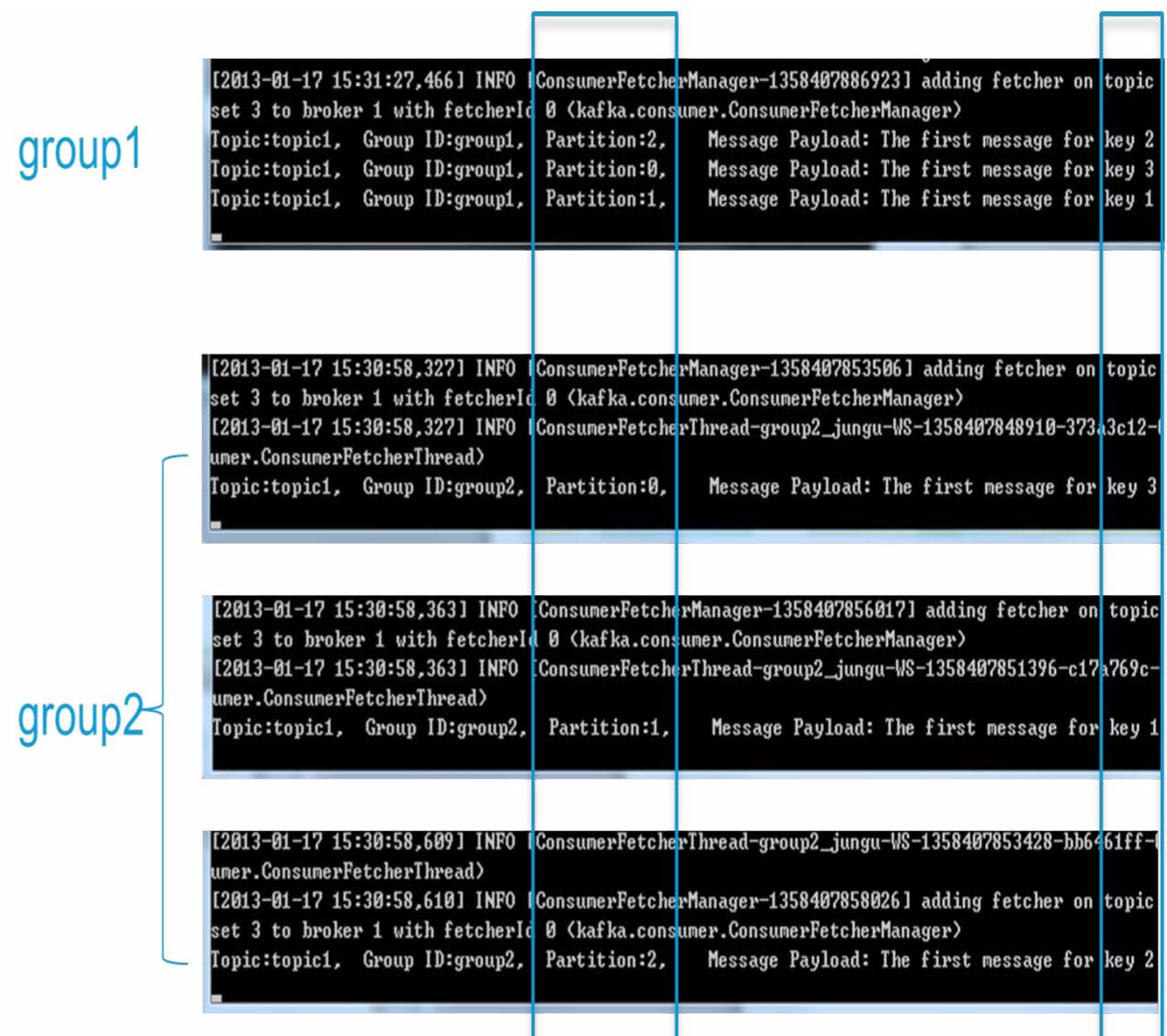
High Level Consumer Rebalance

注：本节所讲述 Rebalance 相关内容均基于

Kafka High Level Consumer。

Kafka 保证同一 Consumer Group 中只有一个 Consumer 会消费某条消息，实际上，Kafka 保证的是稳定状态下每一个 Consumer 实例只会消费某一个或多个特定 Partition 的数据，而某个 Partition 的数据只会被某一个特定的 Consumer 实例所消费。也就是说 Kafka 对消息的分配是以 Partition 为单位分配的，而非以每一条消息作为分配单元。这样设计的劣势是无法保证同一个 Consumer Group 里的 Consumer 均匀消费数据，优势是每个 Consumer 不用都跟大量的 Broker 通信，减少通信开销，同时也降低了分配难度，实现也更简单。另外，因为同一个 Partition 里的数据是有序的，这种设计可以保证每个 Partition 里的数据可以被有序消费。

如果某 Consumer Group 中 Consumer (每个 Consumer 只创建 1 个 MessageStream) 数量少



于 Partition 数量，则至少有一个 Consumer 会消费多个 Partition 的数据，如果 Consumer 的数量与 Partition 数量相同，则正好一个 Consumer 消费一个 Partition 的数据。而如果 Consumer 的数量多于 Partition 的数量时，会有部分 Consumer 无法消费该 Topic 下任何一条消息。

如图 a 所示，如果 topic1 有 0, 1, 2 共三个 Partition，当 group1 只有一个 Consumer (名为 consumer1) 时，该 Consumer 可消费这 3 个 Partition 的所有数据。

图 b 增加一个 Consumer (consumer2) 后，其中一个 Consumer (consumer1) 可消费 2 个 Partition 的数据 (Partition 0 和 Partition

1)，另外一个 Consumer (consumer2) 可消费另外一个 Partition (Partition 2) 的数据。

图 c 再增加一个 Consumer (consumer3) 后，每个 Consumer 可消费一个 Partition 的数据。 consumer1 消费 partition0，consumer2 消费 partition1，consumer3 消费 partition2。

图 d 再增加一个 Consumer (consumer4) 后，其中 3 个 Consumer 可分别消费一个 Partition 的数据，另外一个 Consumer (consumer4) 不能消费 topic1 的任何数据。

图 e，此时关闭 consumer1，其余 3 个 Consumer 可分别消费一个 Partition 的数据。

Consumer 1

```
, partition 2, initOffset 0 to broker 1 with fetcherId 0 <kafka.consumer.ConsumerFetcherManager>
[2013-01-18 11:03:10,814] INFO [ConsumerFetcherManager-1358478190043] adding fetcher on to
, partition 0, initOffset 0 to broker 1 with fetcherId 0 <kafka.consumer.ConsumerFetcherManager>
[2013-01-18 11:03:10,816] INFO [ConsumerFetcherManager-1358478190043] adding fetcher on to
, partition 1, initOffset 0 to broker 1 with fetcherId 0 <kafka.consumer.ConsumerFetcherManager>
Topic:topic1, Group ID:group1, Partition:2, Message Payload: The 1 message for key 2
Topic:topic1, Group ID:group1, Partition:0, Message Payload: The 1 message for key 0
Topic:topic1, Group ID:group1, Partition:1, Message Payload: The 1 message for key 1
```

图 a

Consumer 1

```
, partition 0, initOffset 1 to broker 1 with fetcherId 0 <kafka.consumer.ConsumerFetcherManager>
[2013-01-18 11:05:02,353] INFO [ConsumerFetcherThread-group1_jungu-WS-1358478185425-6bfac3]
tarting <kafka.consumer.ConsumerFetcherThread>
[2013-01-18 11:05:02,354] INFO [ConsumerFetcherManager-1358478190043] adding fetcher on to
, partition 1, initOffset 1 to broker 1 with fetcherId 0 <kafka.consumer.ConsumerFetcherManager>
Topic:topic1, Group ID:group1, Partition:0, Message Payload: The 2 message for key 0
Topic:topic1, Group ID:group1, Partition:1, Message Payload: The 2 message for key 1
```

Consumer 2

```
ducer>
[2013-01-18 11:05:02,593] INFO Disconnecting from 10.75.167.46:49092 <kafka.producer.SyncProducer>
[2013-01-18 11:05:02,604] INFO [ConsumerFetcherThread-group1_jungu-WS-1358478297308-c39893]
tarting <kafka.consumer.ConsumerFetcherThread>
[2013-01-18 11:05:02,604] INFO [ConsumerFetcherManager-1358478301933] adding fetcher on to
, partition 2, initOffset 1 to broker 1 with fetcherId 0 <kafka.consumer.ConsumerFetcherManager>
Topic:topic1, Group ID:group1, Partition:2, Message Payload: The 2 message for key 2
```

图 b

Consumer 1

```
ducer>
[2013-01-18 11:08:57,204] INFO Disconnecting from 10.75.167.46:49092 <kafka.producer.SyncProducer>
[2013-01-18 11:08:57,206] INFO [ConsumerFetcherManager-1358478190043] adding fetcher on to
, partition 0, initOffset 2 to broker 1 with fetcherId 0 <kafka.consumer.ConsumerFetcherManager>
[2013-01-18 11:08:57,207] INFO [ConsumerFetcherThread-group1_jungu-WS-1358478185425-6bfac3]
tarting <kafka.consumer.ConsumerFetcherThread>
Topic:topic1, Group ID:group1, Partition:0, Message Payload: The 3 message for key 0
```

Consumer 2

```
ducer>
[2013-01-18 11:08:57,226] INFO Disconnecting from 10.75.167.46:49092 <kafka.producer.SyncProducer>
[2013-01-18 11:08:57,228] INFO [ConsumerFetcherManager-1358478301933] adding fetcher on to
, partition 1, initOffset 2 to broker 1 with fetcherId 0 <kafka.consumer.ConsumerFetcherManager>
[2013-01-18 11:08:57,228] INFO [ConsumerFetcherThread-group1_jungu-WS-1358478297308-c39893]
tarting <kafka.consumer.ConsumerFetcherThread>
Topic:topic1, Group ID:group1, Partition:1, Message Payload: The 3 message for key 1
```

Consumer 3

```
ducer>
[2013-01-18 11:08:57,522] INFO Disconnecting from 10.75.167.46:49092 <kafka.producer.SyncProducer>
[2013-01-18 11:08:57,533] INFO [ConsumerFetcherThread-group1_jungu-WS-1358478532323-9aed3c]
tarting <kafka.consumer.ConsumerFetcherThread>
[2013-01-18 11:08:57,534] INFO [ConsumerFetcherManager-1358478536931] adding fetcher on to
, partition 2, initOffset 2 to broker 1 with fetcherId 0 <kafka.consumer.ConsumerFetcherManager>
Topic:topic1, Group ID:group1, Partition:2, Message Payload: The 3 message for key 2
```

图 c

Consumer 1

```
[2013-01-18 11:12:01,912] INFO Disconnecting from 10.75.167.46:49092 <kafka.producer.SyncProducer>
[2013-01-18 11:12:01,914] INFO [ConsumerFetcherManager-1358478190043] adding fetcher on to
, partition 0, initOffset 3 to broker 1 with fetcherId 0 <kafka.consumer.ConsumerFetcherManager>
[2013-01-18 11:12:01,914] INFO [ConsumerFetcherThread-group1_jungu-WS-1358478185425-6bfac3]
tarting <kafka.consumer.ConsumerFetcherThread>
Topic:topic1, Group ID:group1, Partition:0, Message Payload: The 4 message for key 0
```

Consumer 2

```
[2013-01-18 11:12:01,893] INFO Disconnecting from 10.75.167.46:49092 <kafka.producer.SyncProducer>
[2013-01-18 11:12:01,900] INFO [ConsumerFetcherManager-1358478301933] adding fetcher on to
, partition 1, initOffset 3 to broker 1 with fetcherId 0 <kafka.consumer.ConsumerFetcherManager>
[2013-01-18 11:12:01,900] INFO [ConsumerFetcherThread-group1_jungu-WS-1358478297308-c39893]
tarting <kafka.consumer.ConsumerFetcherThread>
Topic:topic1, Group ID:group1, Partition:1, Message Payload: The 4 message for key 1
```

Consumer 3

```
[2013-01-18 11:12:01,969] INFO Disconnecting from 10.75.167.46:49092 <kafka.producer.SyncProducer>
[2013-01-18 11:12:01,971] INFO [ConsumerFetcherManager-1358478536931] adding fetcher on to
, partition 2, initOffset 3 to broker 1 with fetcherId 0 <kafka.consumer.ConsumerFetcherManager>
[2013-01-18 11:12:01,971] INFO [ConsumerFetcherThread-group1_jungu-WS-1358478532323-9aed3c]
tarting <kafka.consumer.ConsumerFetcherThread>
Topic:topic1, Group ID:group1, Partition:2, Message Payload: The 4 message for key 2
```

Consumer4

```
erries>
[2013-01-18 11:12:02,143] INFO Fetching metadata for topic Set() <kafka.client.ClientUtils>
[2013-01-18 11:12:02,144] INFO Connected to 10.75.167.46:49092 for producing <kafka.producer.SyncProducer>
[2013-01-18 11:12:02,148] INFO Disconnecting from 10.75.167.46:49092 <kafka.producer.SyncProducer>
```

No data for consumer4

图 d

Consumer 2

```
[2013-01-18 11:16:42,603] INFO Disconnecting from 10.75.167.46:49092 <kafka.producer.SyncProducer>
[2013-01-18 11:16:42,606] INFO [ConsumerFetcherManager-1358478301933] adding fetcher on to
, partition 0, initOffset 4 to broker 1 with fetcherId 0 <kafka.consumer.ConsumerFetcherManager>
[2013-01-18 11:16:42,606] INFO [ConsumerFetcherThread-group1_jungu-WS-1358478297308-c39893]
tarting <kafka.consumer.ConsumerFetcherThread>
Topic:topic1, Group ID:group1, Partition:0, Message Payload: The 5 message for key 0
```

Consumer 3

```
[2013-01-18 11:16:42,607] INFO Disconnecting from 10.75.167.46:49092 <kafka.producer.SyncProducer>
[2013-01-18 11:16:42,608] INFO [ConsumerFetcherManager-1358478536931] adding fetcher on to
, partition 1, initOffset 4 to broker 1 with fetcherId 0 <kafka.consumer.ConsumerFetcherManager>
[2013-01-18 11:16:42,608] INFO [ConsumerFetcherThread-group1_jungu-WS-1358478532323-9aed3c]
tarting <kafka.consumer.ConsumerFetcherThread>
Topic:topic1, Group ID:group1, Partition:1, Message Payload: The 5 message for key 1
```

Consumer 4

```
[2013-01-18 11:16:44,726] INFO Disconnecting from 10.75.167.46:49092 <kafka.producer.SyncProducer>
[2013-01-18 11:16:44,726] INFO [ConsumerFetcherManager-1358478721530] adding fetcher on to
, partition 2, initOffset 4 to broker 1 with fetcherId 0 <kafka.consumer.ConsumerFetcherManager>
[2013-01-18 11:16:44,726] INFO [ConsumerFetcherThread-group1_jungu-WS-1358478716898-1045e3]
tarting <kafka.consumer.ConsumerFetcherThread>
Topic:topic1, Group ID:group1, Partition:2, Message Payload: The 5 message for key 2
```

图 e

图 f 此时关闭 consumer1，其余 3 个 Consumer 可分别消费一个 Partition 的数据。

图 g 接着关闭 consumer2，consumer3 可消费 2 个 Partition，consumer4 可消费 1 个 Partition。

Consumer Rebalance 的算法如下：

- 将目标 Topic 下的所有 Partition 排序，存于 PT；
- 对某 Consumer Group 下所有 Consumer 排序，存于 CG，第 i 个 Consumer 记为 Ci；
- $N = \text{size}(PT) / \text{size}(CG)$, 向上取整；
- 解除 Ci 对原来分配的 Partition 的消费权（从 0 开始）；
- 将第 $i * N$ 到 $(i+1) * N - 1$ 个 Partition 分配给 Ci。

目前，最新版（0.8.2.1）Kafka 的 Consumer Rebalance 的控制策略是由每一个 Consumer 通过在 Zookeeper 上注册 Watch 完成的。每个 Consumer 被创建时会触发 Consumer Group 的 Rebalance，具体启动流程如下：

- High Level Consumer 启动时将其 ID 注册到其 Consumer Group 下，在 Zookeeper 上的路径为 `/consumers/[consumer group]/ids/[consumer id]`；
- 在 `/consumers/[consumer group]/ids` 上注册 Watch；
- 在 `/brokers/ids` 上注册 Watch；
- 如果 Consumer 通过 Topic Filter 创建消息流，则它会同时在 `/brokers/topics` 上也创建 Watch；
- 强制自己在其 Consumer Group 内启动 Rebalance 流程。

在这种策略下，每一个 Consumer 或者

Broker 的增加或者减少都会触发 Consumer Rebalance。因为每个 Consumer 只负责调整自己所消费的 Partition，为了保证整个 Consumer Group 的一致性，当一个 Consumer 触发了 Rebalance 时，该 Consumer Group 内的其它所有其它 Consumer 也应该同时触发 Rebalance。

该方式有如下缺陷。

• Herd effect

任何 Broker 或者 Consumer 的增减都会触发所有的 Consumer 的 Rebalance。

• Split Brain

每个 Consumer 分别单独通过 Zookeeper 判断哪些 Broker 和 Consumer 宕机了，那么不同 Consumer 在同一时刻从 Zookeeper “看”到的 View 就可能不一样，这是由 Zookeeper 的特性决定的，这就会造成不正确的 Rebalance 尝试。

• 调整结果不可控

所有的 Consumer 都并不知道其它 Consumer 的 Rebalance 是否成功，这可能会导致 Kafka 工作在一个不正确的状态。

根据 Kafka 社区 wiki，Kafka 作者正在考虑在还未发布的 [0.9.x 版本](#) 中使用中心协调器（Coordinator）。大体思想是为所有 Consumer Group 的子集选举出一个 Broker 作为 Coordinator，由它 Watch Zookeeper，从而判断是否有 Partition 或者 Consumer 的增减，然后生成 Rebalance 命令，并检查是否这些 Rebalance 在所有相关的 Consumer 中被执行成功，如果不成功则重试，若成功则认为此次 Rebalance 成功（这个过程跟 Replication Controller 非常类似）。具体方案将在后文中详细阐述。

Consumer 3

```
[2013-01-18 11:19:15,580] INFO [ConsumerFetcherThread-group1_jungu-WS-1358478532323-9aed3c]
[2013-01-18 11:19:15,581] INFO [ConsumerFetcherManager-1358478536931] adding fetcher on to
, partition 1, initOffset 5 to broker 1 with fetcherId 0 (kafka.consumer.ConsumerFetcherManager)
Topic:topic1, Group ID:group1, Partition:0, Message Payload: The 6 message for key 0
Topic:topic1, Group ID:group1, Partition:1, Message Payload: The 6 message for key 1
.
```

Consumer 4

```
[2013-01-18 11:19:15,579] INFO Disconnecting from 10.75.167.46:49092 (kafka.producer.SyncProducer)
[2013-01-18 11:19:15,581] INFO [ConsumerFetcherManager-1358478721530] adding fetcher on to
, partition 2, initOffset 5 to broker 1 with fetcherId 0 (kafka.consumer.ConsumerFetcherManager)
[2013-01-18 11:19:15,584] INFO [ConsumerFetcherThread-group1_jungu-WS-1358478716898-1045e3]
[2013-01-18 11:19:15,585] INFO [kafka.consumer.ConsumerFetcherThread]
Topic:topic1, Group ID:group1, Partition:2, Message Payload: The 6 message for key 2
.
```

图 f

Consumer 4

```
[2013-01-18 11:21:24,538] INFO [ConsumerFetcherThread-group1_jungu-WS-1358478716898-1045e3]
[2013-01-18 11:21:24,539] INFO [kafka.consumer.ConsumerFetcherThread]
[2013-01-18 11:21:24,539] INFO [ConsumerFetcherManager-1358478721530] adding fetcher on to
, partition 0, initOffset 6 to broker 1 with fetcherId 0 (kafka.consumer.ConsumerFetcherManager)
[2013-01-18 11:21:24,539] INFO [ConsumerFetcherManager-1358478721530] adding fetcher on to
, partition 1, initOffset 6 to broker 1 with fetcherId 0 (kafka.consumer.ConsumerFetcherManager)
Topic:topic1, Group ID:group1, Partition:2, Message Payload: The 7 message for key 2
Topic:topic1, Group ID:group1, Partition:0, Message Payload: The 7 message for key 0
Topic:topic1, Group ID:group1, Partition:1, Message Payload: The 7 message for key 1
.
```

图 g

Low Level Consumer

使用 Low Level Consumer (Simple Consumer) 的主要原因是，用户希望比 Consumer Group 更好的控制数据的消费。比如：

- 同一条消息读多次；
- 只读取某个 Topic 的部分 Partition；
- 管理事务，从而确保每条消息被处理一次，且仅被处理一次。

与 Consumer Group 相比，Low Level Consumer 要求用户做大量的额外工作。

- 必须在应用程序中跟踪 offset，从而确定下一条应该消费哪条消息；
- 应用程序需要通过程序获知每个 Partition

的 Leader 是谁；
• 必须处理 Leader 的变化。
使用 Low Level Consumer 的一般流程如下：

- 查找到一个“活着”的 Broker，并且找出每个 Partition 的 Leader；
- 找出每个 Partition 的 Follower；
- 定义好请求，该请求应该能描述应用程序需要哪些数据；
- Fetch 数据；
- 识别 Leader 的变化，并对之作出必要的响应。

Consumer 重新设计。

根据社区 wiki，Kafka 在 0.9.* 版本中，

重新设计 Consumer 可能是最重要的 Feature 之一。本节会根据社区 wiki 介绍 Kafka 0.9.* 中对 Consumer 可能的设计方向及思路。

设计方向

简化消费者客户端

部分用户希望开发和使用 non-java 的客户端。现阶段使用 non-java 发 SimpleConsumer 比较方便，但想开发 High Level Consumer 并不容易。因为 High Level Consumer 需要实现一些复杂但必不可少的失败探测和 Rebalance。如果能将消费者客户端更精简，使依赖最小化，将会极大的方便 non- java 用户实现自己的 Consumer。

中心 Coordinator

如上文所述，当前版本的 High Level Consumer 存在 Herd Effect 和 Split Brain 的问题。如果将失败探测和 Rebalance 的逻辑放到一个高可用的中心 Coordinator，那么这两个问题即可解决。同时还可大大减少 Zookeeper 的负载，有利于 Kafka Broker 的 Scale Out。

允许手工管理 offset

一些系统希望以特定的时间间隔在自定义的数据库中管理 Offset。这就要求 Consumer 能获取到每条消息的 metadata，例如 Topic, Partition, Offset，同时还需要在 Consumer 启动时得到每个 Partition 的 Offset。实现这些，需要提供新的 Consumer API。同时有个问题不得不考虑，即是否允许 Consumer 手工管理部分 Topic 的 Offset，而让 Kafka 自动通过 Zookeeper 管理其它 Topic 的 Offset。一个可能的选项是让每个 Consumer 只能选

取 1 种 Offset 管理机制，这可极大的简化 Consumer API 的设计和实现。

Rebalance 后触发用户指定的回调

一些应用可能会在内存中为每个 Partition 维护一些状态，Rebalance 时，它们可能需要将该状态持久化。因此该需求希望支持用户实现并指定一些可插拔的并在 Rebalance 时触发的回调。如果用户使用手动的 Offset 管理，那该需求可方便得由用户实现，而如果用户希望使用 Kafka 提供的自动 Offset 管理，则需要 Kafka 提供该回调机制。

非阻塞式 Consumer API

该需求源于那些实现高层流处理操作，如 filter by, group by, join 等，的系统。现阶段的阻塞式 Consumer 几乎不可能实现 Join 操作。

如何通过中心 Coordinator 实现 Rebalance 成功 Rebalance 的结果是，被订阅的所有 Topic 的每一个 Partition 将会被 Consumer Group 内的一个（有且仅有一个）Consumer 拥有。每一个 Broker 将被选举为某些 Consumer Group 的 Coordinator。某个 Consumer Group 的 Coordinator 负责在该 Consumer Group 的成员变化或者所订阅的 Topic 的 Partition 变化时协调 Rebalance 操作。

1) Consumer 启动时，先向 Broker 列表中的任意一个 Broker 发送 ConsumerMetadataRequest，并通过 ConsumerMetadataResponse 获取它所在 Group 的 Coordinator 信息。

ConsumerMetadataRequest 和 ConsumerMetadataResponse 的结构如下。

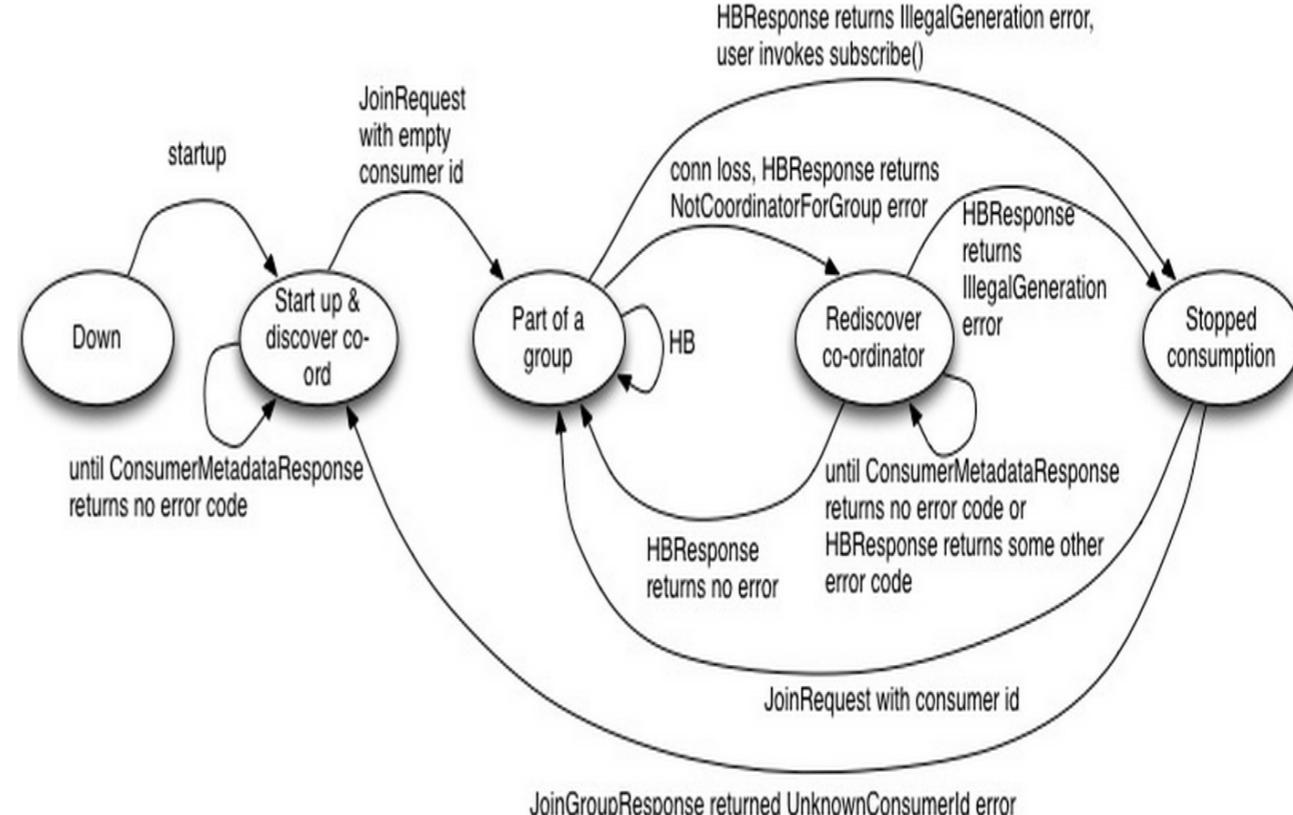
001 ConsumerMetadataRequest

```
002 {
003     GroupId          => String
004 }
005
006 ConsumerMetadataResponse
007 {
008     ErrorCode       => int16
009     Coordinator    => Broker
```

2) Consumer 连接 到 Coordinator 并发送 HeartbeatRequest，如果返回的 HeartbeatResponse 没有任何错误码，Consumer 继续 fetch 数据。若其中包含 IllegalGeneration 错误码，即说明 Coordinator 已经发起了 Rebalance 操作，此时 Consumer 停止 fetch 数据，commit offset，并发送 JoinGroupRequest 给它的 Coordinator，并在 JoinGroupResponse 中获得它应该拥有的所有 Partition 列表和它所属的 Group 的新的 Generation ID。此时 Rebalance 完成，Consumer 开始 fetch 数据。相应 Request 和 Response 结构如下。

001 HeartbeatRequest

```
002 {
003     GroupId          => String
004     GroupGenerationId => int32
005     ConsumerId       => String
006 }
007 HeartbeatResponse
008 {
009     ErrorCode       => int16
010 }
011 JoinGroupRequest
012 {
013     GroupId          => String
014     SessionTimeout   => int32
015     Topics           => [String]
016     ConsumerId       => String
017     PartitionAssignmentStrategy => String
018 }
019 JoinGroupResponse
020 {
021     ErrorCode       => int16
022     GroupGenerationId => int32
023     ConsumerId       => String
024     PartitionsToOwn  => [TopicName [Partition]]
025 }
026 TopicName => String
027 Partition => int32
```



Down: Consumer 停止工作

Start up & discover coordinator:
Consumer 检测其所在 Group 的 Coordinator。一旦它检测到 Coordinator，即向其发送 JoinGroupRequest。

Part of a group: 该状态下，Consumer 已经是该 Group 的成员，并周期性发送 HeartbeatRequest。如 HeartbeatResponse 包含 IllegalGeneration 错误码，则转换到 Stopped Consumption 状态。若连接丢失，HeartbeatResponse 包含 NotCoordinatorForGroup 错误码，则转换到 Rediscover coordinator 状态。

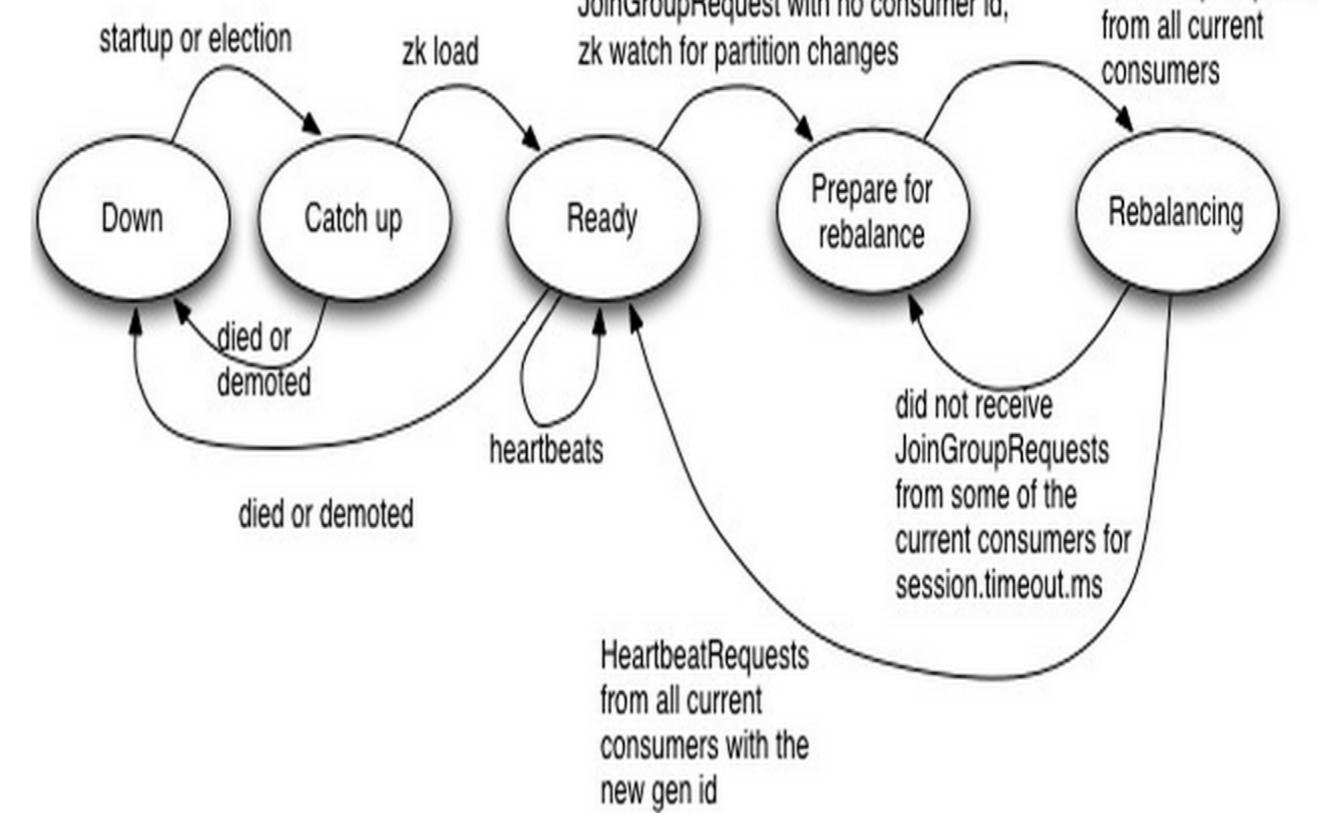
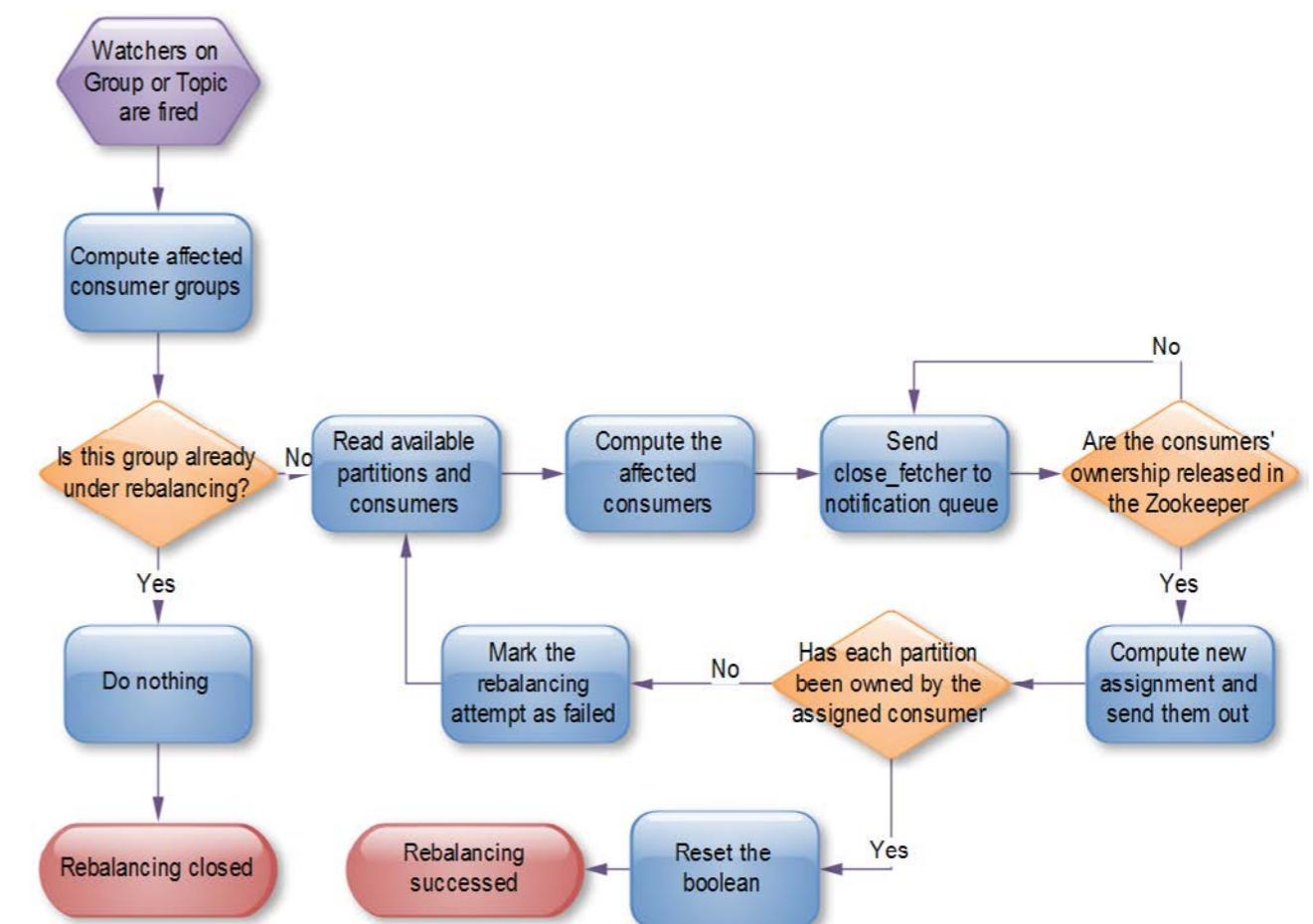
将该 Consumer 标记为宕机状态并为其所在 Group 触发一次 Rebalance 操作。

Coordinator Failover 过程中，Consumer 可能在新的 Coordinator 完成 Failover 过程之前或之后发现新的 Coordinator 并向其发送 HeartbeatRequest。对于后者，新的 Coordinator 可能拒绝该请求，致使该 Consumer 重新探测 Coordinator 并发起新的连接请求。如果该 Consumer 向新的 Coordinator 发送连接请求太晚，新的 Coordinator 可能已经在此之前将其标记为宕机状态而将之视为新加入的 Consumer 并触发一次 Rebalance 操作。

Coordinator

- 稳定状态下，Coordinator 通过上述故障探测机制跟踪其所管理的每个 Group 下的每个 Consumer 的健康状态。
- 刚启动时或选举完成后，Coordinator 从 Zookeeper 读取它所管理的 Group 列表及这些 Group 的成员列表。如果没有获取到 Group 成员信息，它不会做任何事情直到某个 Group 中有成员注册进来。
- 在 Coordinator 完成加载其管理的 Group 列表及其相应的成员信息之前，它将为 HeartbeatRequest, OffsetCommitRequest 和 Join Group Requests 返回 CoordinatorStartupNotComplete 错误码。此时，Consumer 会重新发送请求。
- Coordinator 会跟踪被其所管理的任何 Consumer Group 注册的 Topic 的 Partition 的变化，并为该变化触发 Rebalance 操作。创建新的 Topic 也可能触发 Rebalance，因为 Consumer 可以在 Topic 被创建之前就已经订阅它了。

Coordinator 发起 Rebalance 操作流程如下所示。



故障检测机制

Consumer 成功加入 Group 后，Consumer 和相应的 Coordinator 同时开始故障探测程序。Consumer 向 Coordinator 发起周期性的 Heartbeat (HeartbeatRequest) 并等待响应，该周期为 session.timeout.ms/heartbeat.frequency。若 Consumer 在 session.timeout.ms 内未收到 HeartbeatResponse，或者发现相应的 Socket channel 断开，它即认为 Coordinator 已宕机并启动 Coordinator 探测程序。若 Coordinator 在 session.timeout.ms 内没有收到一次 HeartbeatRequest，则它

Coordinator 状态机

Down: Coordinator 不再担任之前负责的 Consumer Group 的 Coordinator

Catch up: 该状态下, Coordinator 竞选成功, 但还未做好服务相应请求的准备。

Ready: 该状态下, 新竞选出来的 Coordinator 已经完成从 Zookeeper 中加载它所负责管理的所有 Group 的 metadata, 并可开始接收相应的请求。

Prepare for rebalance: 该状态下, Coordinator 在所有 HeartbeatResponse 中返回 IllegalGeneration 错误码, 并等待所有 Consumer 向其发送 JoinGroupRequest 后转到 Rebalancing 状态。

Rebalancing: 该状态下, Coordinator 已经收到了 JoinGroupRequest 请求, 并增加其 Group Generation ID, 分配 Consumer ID, 分配 Partition。Rebalance 成功后, 它会等待接收包含新的 Consumer Generation ID 的 HeartbeatRequest, 并转至 Ready 状态。

Coordinator Failover

如前文所述, Rebalance 操作需要经历如下几个阶段:

- Topic/Partition 的改变或者新 Consumer 的加入或者已有 Consumer 停止, 触发 Coordinator 注册在 Zookeeper 上的 watch, Coordinator 收到通知准备发起 Rebalance 操作。
- Coordinator 通过在 HeartbeatResponse 中返回 IllegalGeneration 错误码发起

Rebalance 操作。

- Consumer 发送 JoinGroupRequest。
- Coordinator 在 Zookeeper 中增加 Group 的 Generation ID 并将新的 Partition 分配情况写入 Zookeeper。
- Coordinator 发送 JoinGroupResponse。

在这个过程中的每个阶段, Coordinator 都可能出现故障。下面给出 Rebalance 不同阶段中 Coordinator 的 Failover 处理方式。

1) 如果 Coordinator 的故障发生在第一阶段, 即它收到 Notification 并未来得及作出响应, 则新的 Coordinator 将从 Zookeeper 读取 Group 的 metadata, 包含这些 Group 订阅的 Topic 列表和之前的 Partition 分配。如果某个 Group 所订阅的 Topic 数或者某个 Topic 的 Partition 数与之前的 Partition 分配不一致, 亦或者某个 Group 连接到新的 Coordinator 的 Consumer 数与之前 Partition 分配中的不一致, 新的 Coordinator 会发起 Rebalance 操作。

2) 如果失败发生在阶段 2, 它可能对部分而非全部 Consumer 发出带错误码的 HeartbeatResponse。与第上面第一种情况一样, 新的 Coordinator 会检测到 Rebalance 的必要性并发起一次 Rebalance 操作。如果 Rebalance 是由 Consumer 的失败所触发并且 Consumer 在 Coordinator 的 Failover 完成前恢复, 新的 Coordinator 不会为此发起新的 Rebalance 操作。

3) 如果 Failure 发生在阶段 3, 新的 Coordinator 可能只收到部分而非全部 Consumer 的 JoinGroupRequest。Failover 完成后, 它可能收到部分 Consumer 的 HeartRequest 及另外部分 Consumer 的 JoinGroupRequest。与第 1 种情况类似, 它将

发起新一轮的 Rebalance 操作。

4) 如果 Failure 发生在阶段 4, 即它将新的 Group Generation ID 和 Group 成员信息写入 Zookeeper 后。新的 Generation ID 和 Group 成员信息以一个原子操作一次性写入 Zookeeper。Failover 完成后, Consumer 会发送 HeartbeatRequest 给新的 Coordinator, 并包含旧的 Generation ID。此时新的 Coordinator 通过在 HeartbeatResponse 中返回 IllegalGeneration 错误码发起新一轮 Rebalance。这也解释了为什么每次 HeartbeatRequest 中都需要包含 Generation ID 和 Consumer ID。

5) 如果 Failure 发生在阶段 5, 旧的 Coordinator 可能只向 Group 中的部分

Consumer 发送了 JoinGroupResponse。收到 JoinGroupResponse 的 Consumer 在下次向已经失效的 Coordinator 发送 HeartbeatRequest 或者提交 Offset 时会检测到它已经失败。此时, 它将检测新的 Coordinator 并向其发送带有新的 Generation ID 的 HeartbeatRequest。而未收到 JoinGroupResponse 的 Consumer 将检测新的 Coordinator 并向其发送 JoinGroupRequest, 这将促使新的 Coordinator 发起新一轮的 Rebalance。

作者简介

郭俊 (Jason), 硕士, 从事大数据平台研发工作, 精通 Kafka 等分布式消息系统, Storm 等流式处理系统及数据库性能调优。个人博客: <http://www.jasong.j.com>。

更多与 Kafka 相关的文章



我为什么反对用Node



作者 许令波

随着无线端的快速普及，前后端分离技术走上前台，而 Node 由于它的一些特性被工程师快速接受尤其是前端工程师，所以产生了很多 Node 是否会引起新的技术变革的讨论。我本人是淘系的一个 Web 开发人员，基本上经历了淘系关于 Node 和 Java 技术选型讨论的过程，所以今天我给大家推演一下在像淘系这个环境下 Node 能否会成为主流的 Web 开发技术，当然后面也给出了我认为比较适合的场景。

Node 火了

在百度中搜索 Node 可以得到 105w 个结果，图书出版方面 13 年 3 月到 15 年 6 月 2 年时间有近 20 种相关的 Node 书出版，实践方面国外公司 PayPal、LinkedIn、groupon 也都在使用，国内大公司阿里、腾讯、百度也都有实践项目在尝试。这让我想起了当初 Nosql 新出来是一样的场景，大家都一窝蜂的涌上去拥抱新技术，获取新技术带来的红利。

Node 很火的另一个推手是当前的无线技术流行，很多应用从传统的 PC 开发要转到无线等多端，这种情况下渲染层和逻辑层的分离变得重要起来，而 Node 刚好可以很好的渲染前端页面，所以我们的开发同学不遗余力的在推广 Node 技术。

Node 能够火起来最重要的原因还是它的确给我们的开发带来了很多好处：

- 基于事件驱动；
- 无阻塞 I/O。

Node 还有其它一些优点如单线程，总体来说 Node 是为轻量级的分布式的实时数据服务这类应用提供运行容器而设计的，这类应用很容易想到微博、Facebook 这类典型场景，需要非常实时化、个性化、高并发的数据服务。

为什么火了

今年由于无线终端的兴起，后端要提供基于 JSON API 的数据接口非常普遍。目前来看公司还存在两种形态：一个是无线作为新系统独立存在于传统 PC 系统；另外一种是将无线系统合并到老的 PC 系统，在一个系统里同时支持多端服务。长期来看无线和 PC 系统的合并是必然，业务上以无线为主，PC 仅仅作为一个终端而已，不可能存在无线和 PC 两套业务逻辑。

那基于无线和 PC 业务合并，由一个系统提供多终端、多语言适配的角度来看，Node 能否在其中扮演传统服务端 MVC 中的 V 角色？

要回答这个问题，我们再设想一下，在多端情形下，需要怎样的交互：PC、手机端、Pad、TV、Car、Watch 等其它移动终端。

是 Native 还是 H5

当前移动端主要还是以 Native 实现为主，从用户体验角度来考虑 Native 的实现要比 H5 更流畅，同时 Native 还可以基于本地做很多在浏览器里不能做的优化，如大数据的存储、可以定制的通信协议、更方便的保持长连接以及更容易实现的实时消息推送。

当然 H5 也有其无法比拟的优势，客户端更轻

量级，服务端发布更迅速，不需要用户升级版本等。长期来看移动端能否会向早期 PC 那样也从富客户端转向浏览器呢？

我的判断是未必，有几个因素，首先 Native 实现性能优势相比 H5 会好很多，当前移动端都在追求极致体验的时代，无疑 APP 会比 H5 有很多的优势；其次，移动端屏幕较小，基于网页的交互和 APP 相比还有很多限制。最重要的是不同的商家是主推带有品牌标识的 APP 还是向统一的浏览器靠拢，从目前的趋势看，APP 会是手机端上争夺的重点。所以推测直接基于手机端的浏览器的应用不会成为主流前端。

如何实现快速迭代

基于 APP 的 Native 如何解决客户端更新和服务端的快速迭代，这个问题是当前正在着力解决的，目前为止有两种思路：一种是客户端用同一种技术开发，然后通过工具编译技术把它编译成不同平台上能够执行的代码，如当前的 React Native；另一种思路是将客户端中经常需要更新的模块做成动态推送的，用模板 + 数据的方式，在不同的客户端平台上实现一个小小的解析引擎来实现快速个性化的定制，目前手游主要采用后面一种实现方式，当然前一种也正在尝试。

那么再说回来，基于前面的这些推断，多终端和服务端交互主要是数据 + 模板的方式为主，那么服务端提供格式化的数据将成为必然选项。所以涉及到的问题就是服务端既要提供格式化的数据（Http JSON 数据），又要支持传统的 PC 的方式：基于 JSON 数据渲染出 HTML 页面，所以很容易想到将渲染层独立出来用 Node 完成。

真的靠谱吗

既然 Node 可以带来这么多好处，那么我们不妨就继续向下推演，看是否真的很靠谱？下面看下 Node 在我们的实际的开发环境中如何使用，在引入 Node 之前我们有必要先介绍下当前 Web 服务端架构。（见图 1）

在当前这种架构下 Node 怎么融入进去呢？最保守的一种方案是将当前的 Java Web 中的 VIEW 层从 MVC 中独立出来，交给 Node 来完成，Java Web 只提供基于 JSON 数据接口给 Node 调用，架构图变成了如下的形式。（见图 2）

那么 Node 能否取代前台的 Web 系统，成为很明显的差别是在我们当前的访问路径上多增加了一个 Web 代理层，而这一层和当前的 Web

服务器层怎么看都有点别扭，两者同时存在始终觉得有一个是多余的，那么 Node 能否替代掉 Nginx 成为 Web 代理服务器呢？理论上是完全没问题的，就像我们用 PHP 来代替 Java Web 开发一样，不过你放到具体的公司运维体系中，你会发现目前在 Nginx 上的防攻击、限流、数据埋点、热点 cache 等模块都要在 Node 上重新开发一遍，最重要的是用 Node 取代 Nginx 并不能带来额外的好处，如果说 Node 可以渲染页面，那么 Nginx 的开发会和你讨论 Nginx lua 模块和 Node 哪个更合适，所以用 Node 取代 Nginx 作为代理服务器也不太现实。

那么 Node 能否取代前台的 Web 系统，成为主流的 MVC 框架呢？

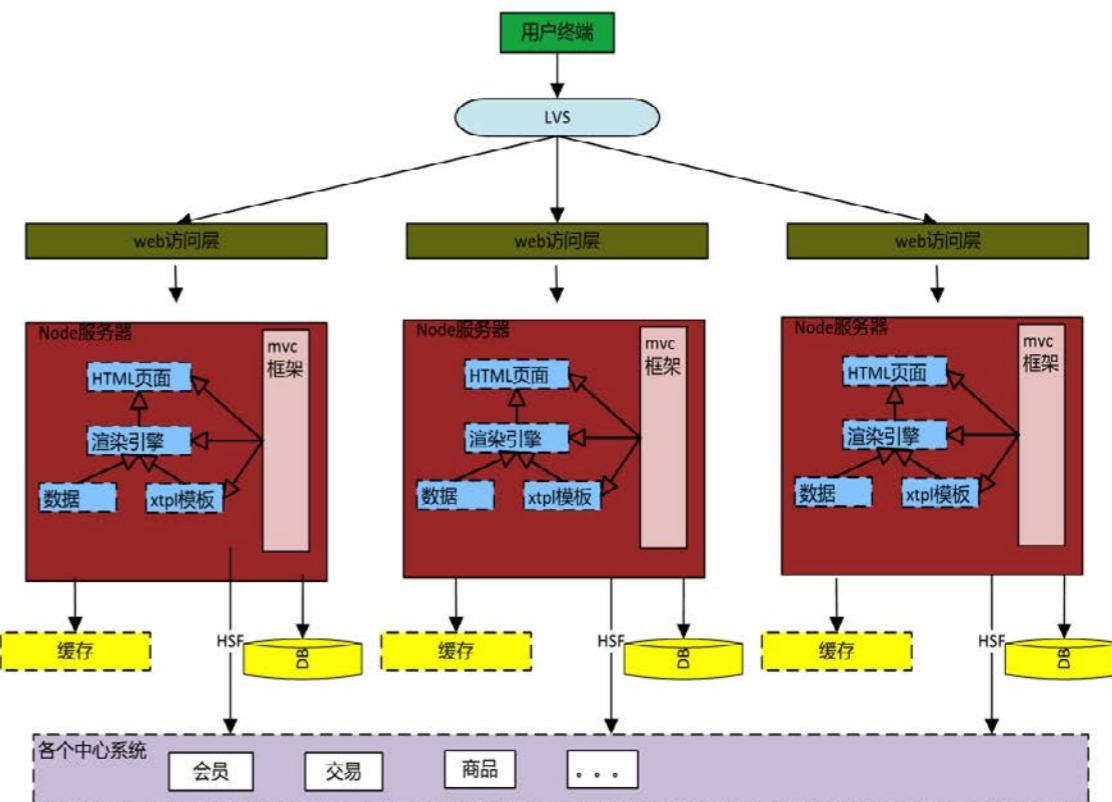
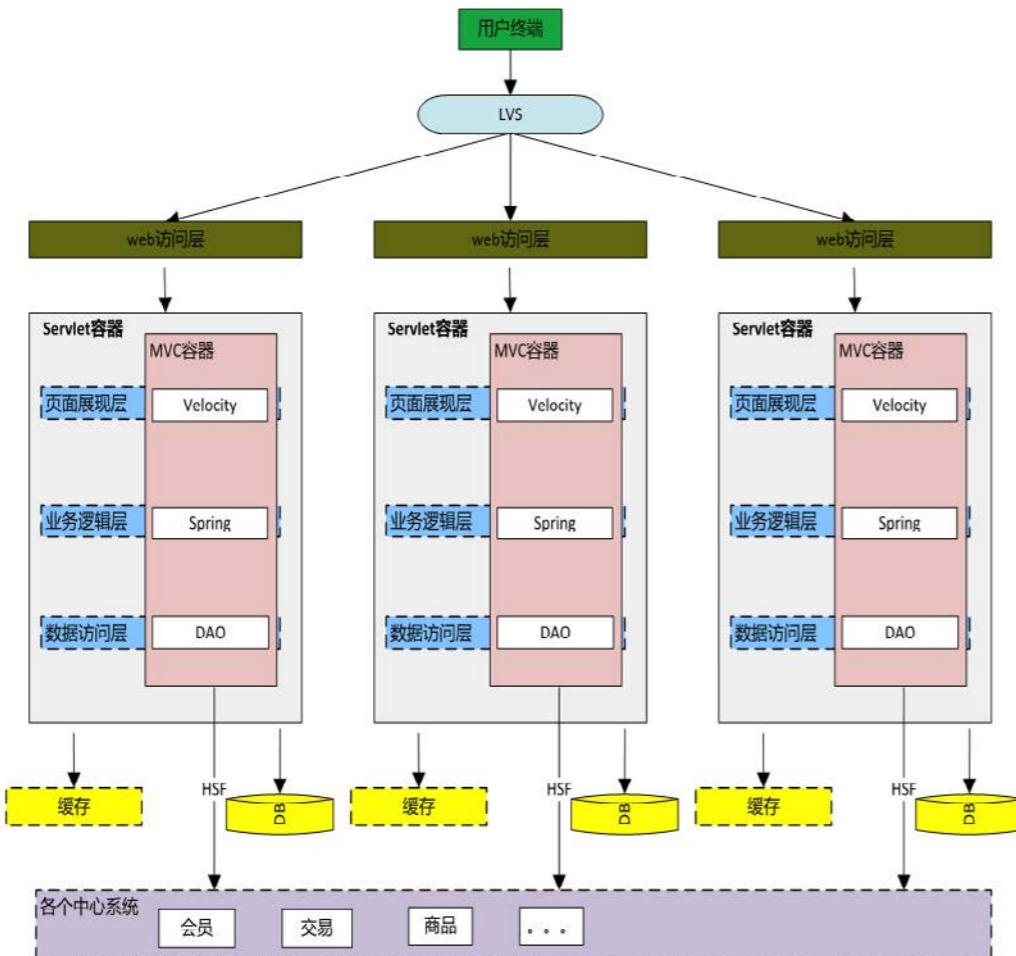


图 2 Node 作为渲染层加入到传统架构中

未必

Node 和 Java Web 一样可以提供 MVC 管理功能，一个系统中同时存在两套 MVC 框架显然不合理，那么如果用 Node 来替换 Java Web 的话，服务端的架构变成如图 3。

从技术实现上这种架构没什么大问题，就是用 Node 上的 MVC 框架如 express 来替代 Java Web 中 Webx，也就是用 JavaScript 替换 Java，以及整个运行容器和中间件都要替换，那么是否真的带来那么大的好处呢？

我们从语言特性、开发效率和成本因素三个方面来看，Node 作为后来者能否比我们现在的 Java 更优秀。

语言特性

JavaScript 作为 Node 上运行的语言和 Java

相比，优缺点很明显。JavaScript 语法简单，很容易编写基于事件的驱动的实现。但是 JavaScript 基于面向对象的描述能力偏弱，不像 Java 是真正的面向对象语言，同时 JavaScript 的对数据类型定义也比较单一，要么是数值类型要么是字符类型。很明显 Java 更擅长构建复杂逻辑的大型应用程序，在这一点上 JavaScript 明显落于下风。

在语言运行效率上，JavaScript 本来是解释执行，而 Java 是编译执行，但是由于 Node 做了优化，所以运行效率差别不大。

开发效率

开发效率可以从语言的复杂度、程序员培养、开发工具包的丰富性以及编码效率几个方面比较。

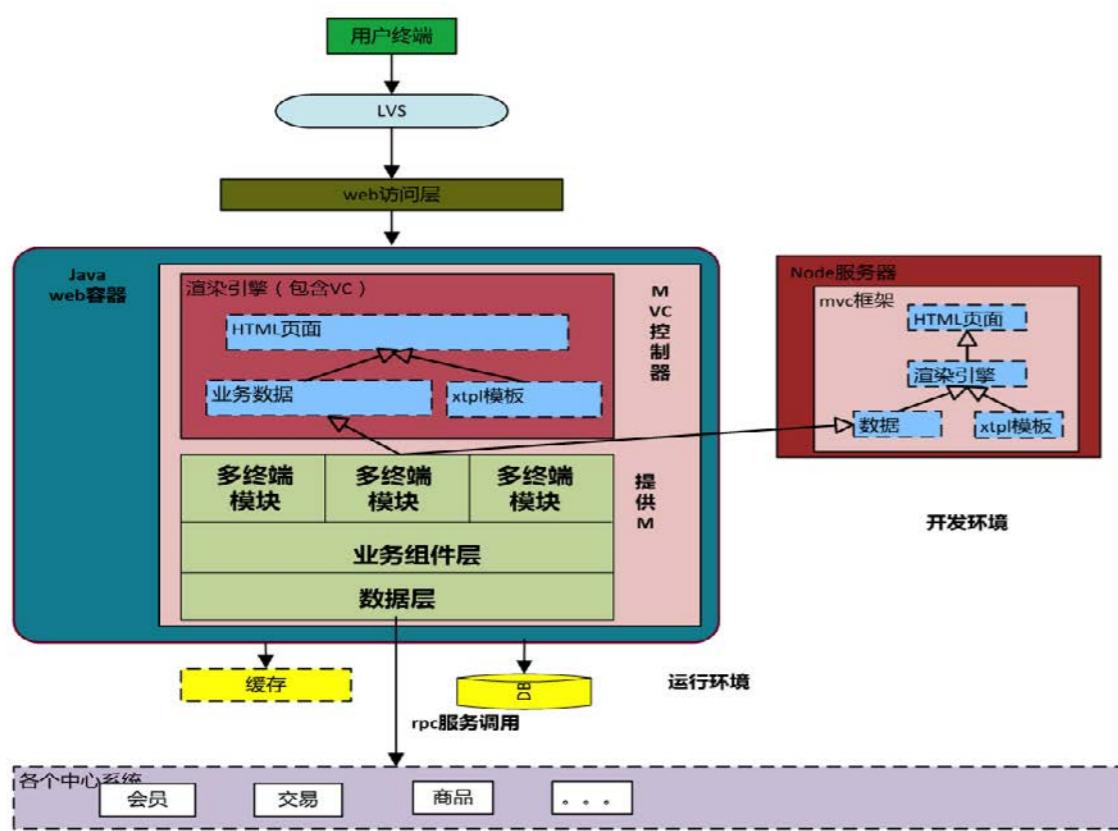


图3 Node 替代 Java Web

- 语言的复杂度。Java 和 JavaScript 从开发角度都不需要关心内存的管理，都是基于虚拟机来管理内存，从并发角度来看：JavaScript 是基于事件触发的，而 Java 是基于线程的，JavaScript 更占优势。另外 JavaScript 是无阻塞 I/O 的，在 I/O 效率上 JavaScript 也更占优势，虽然 Java8 也将更好的支持异步 I/O。
- 程序员培养。目前 Java 语言仍然是仅次于 C 语言的第二大编程语言，而 JavaScript 排第 10 位，Java 程序员队伍要比 JavaScript 大很多，很显然 Java 程序员招聘要比 JavaScript 更容易一点。
- 开发工具包。一个语言的开发效率很多时候要这个语言的支持工具包和组件的丰富性，Java 经过这么多年的发展，工具类库已经非常丰富，几乎你想要的工具类库都能在网上找到。JavaScript 虽然也发展很长时间，但是基于 JavaScript 的工具类库主要集中在前端，能够直接用于 Node 上的仍然很少，

成本因素

前面主要是从技术角度考虑，但是如果往 Node 迁移的话，成本也是一个考虑因素。

- 首先是学习成本。公司大部分是 Java 程序员要往 Node 迁移很明显这个学习成本会非常巨大，即使这个迁移是逐渐的，长期来看仍然是要将一部分 Java 程序员替换成 JavaScript 程序员，不管这个程序员是公司内培养的还是从外部招聘的，我们可以算一下帐，公司招聘一个程序员的成本有多大：一个普通的工程师的年薪假定有 10w 以

当然 Node 的社区非常活跃，可以预见 Node 的工具类库增长也会非常迅速。但是短时间内要达到 Java 的规模尚需时日。

- 编码效率。Java 语言的运行基于 JVM，但是 Java 的部署效率稍差，而 JavaScript 使测试更加简单，容器重启更快，但是 debug 机制仍然不完善，所以很难做 debug 测试。

上，猎头费一般是年薪的 20% 以上，也就是 2w，再加上一个月的实习成本 1w，加在一起约 3w，对于一个有 1w 以上开发的大公司成本可想而知，即使是招聘应届生，由于应届生的培养周期更长，所以学习成本会更高。

- 其次是环境成本。公司的基础服务产品，如中间件都是基于 Java 开发的，如果要替换成 JavaScript 必然要再开发一套，还有配套的运维工具等，这个成本也可想而知。
- 最后是维护成本。Java 和 JavaScript 都是基于容器运行，JVM 和 v8 引擎相比，程序员显然对 JVM 更熟悉。另外从排查问题的难度程度来看针对 JVM 的工具显然更完善。

从上面几个方面来看，当前在阿里要让 JavaScript 完全取代 Java 作为后端开发语言，基于 Node 用 JavaScript 实现整个服务层逻辑显然成本会比较高。

换个角度

我们再换一种角度推演一下，假如我们现在的 Web 系统都用 Node 实现，那必然有很多 Java 工程师会做 Node 的开发，因为我们现有的前端工程师人数肯定是支撑不了现有的业务发展的。我们假定一部分 Java 工程师愿意学习 JavaScript 成为全栈工程师，但是是否同一个人愿意用两种不同的语言完成同一个任务呢，正常来说，如果我能用同一个方式完成全部工作，我为什么要把一个任务分成两种不同的方式来完成，这显然也不太合理。

怎么办

基于前面的分析，Node 不管是在现有基础上单独增加一层还是要整个替换 Java Web 层都不太合理，那是否意味着这种前后端分离的思路有没有合理之处？有没有更好的实现呢？

传统的 MVC Web 软件架构将渲染层独立出来交由前端同学控制有其合理性，在当前的多终端开发模式下，将业务逻辑层和前端渲染层分离有利于提升后端的开发效率，后端只需要关注后端的业务逻辑和数据的输出，因为在 Native 开发下服务端只需要输出 JSON 数据，客户端的页面渲染有前端同学完成。H5 和 PC 需要的 HTML 渲染统一交给前端同学完成有利于前端更好的开发模板，从以往的先画好模板（HTML），给后端同学转化成相应的模板（如 Velocity 或 JSP），然后再基于复杂的 Java Web 工程下调试页面（前端要独立的运行整个 Java Web 工程还是相当的困难）。而渲染层全部交给前端的话，前端同学和后端只需要约定好数据后，页面完全由前端同学完成，减少了交流成本（不过从淘宝的基于 Node 实践来看，整个效率提升还不是那么明显，大部分是把原本是后端的开发工作量转嫁给前端了）。

还有一个重要的理由是前端有了渲染层的控制权，前端的开发体验有了不小的提升，说白了就是前端从以往的配合角色转变成一个 Web 渲染层的 Owner，更加有了主人翁角色，如果再维护 Node 的话，和以往的后端 Java 开发几乎一样了，而这种前端职责的提升恰恰是从后端削弱过来的，所以第一个出来反对的肯定是前端同学，当前在阿里 Node 发展比较缓慢我想也有一点关系吧。

再说回来，我们一直讨论的基于 Node 实现的前端分离方案，可以把他分解一下 Node 技术和前后端分离。很明显前后端分离在当前多终端背景下有其合理性，但是是否一定要用 Node 来实现呢？答案是不一定。当前还有两种方案。

方案一将 Node 层代码抛到 Java 体系上，如图 4。

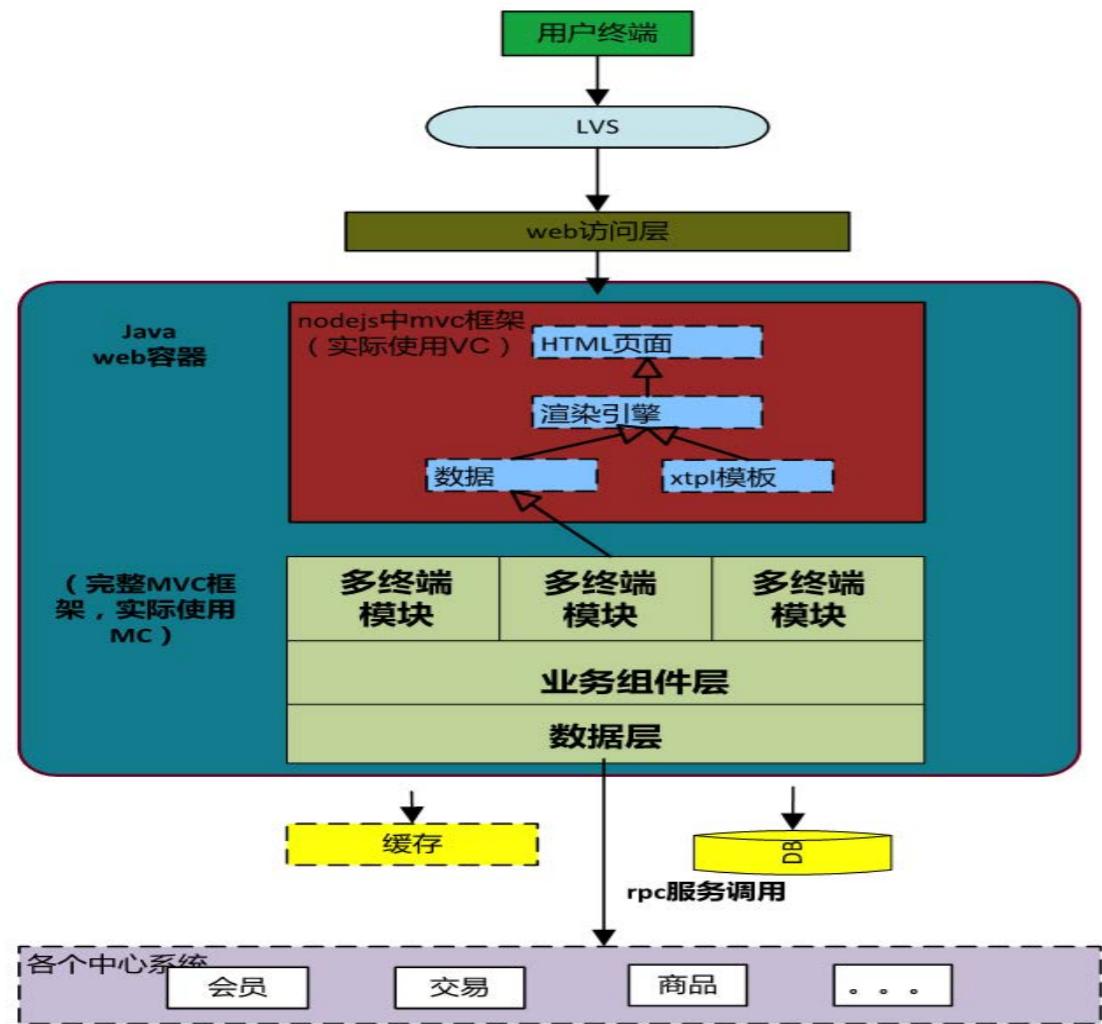


图4 Node嵌入到Java中

当前的 Java8 (Nashorn) 已经可以支持 JavaScript 的解析，而 Avatar.JS 可以将 Node 无缝迁移过来，但是经过测试，Nashorn+Avatar.JS 的执行效率太差，有 4 到 10 倍的性能下降，最好也有一半的下降，这样很难达到工程级别的使用。

另外一种就是仍然在基于 Java Web 体系下，而是将渲染层独立出来，渲染层和业务逻辑层仍然通过 JSON (或者大对象) 数据交互，使得渲染层既可以在 Java 上渲染也可以在 Node 上执行。如下图 5 所示。

这种方式与前一种的区别是只做渲染引擎的适配，即模板在 Node 和 Java 上都可以解析，而不是把 Node 的整个 MVC 都搬过来，由于 Node

和 Java Web 上都有控制逻辑 (即 MVC 中的 C)，所以如果 Node 和 Java 中逻辑不一致会导致两边渲染出来的 HTML 不一致，所以需要把 URL 改造成满足 RESTFULL 的格式，尽量把 C 的逻辑简单化。

图 5 的架构正是目前我们在详情系统上做一个实践，成功的关键是将 XTPL 的模板从 Node 上无缝迁移到 Java 上，另外就是保持页面的路由尽量简单，这样前端在 Node 上开发的重点只是 XTPL 模板。这种解决方案达到几个目的：

- 我们的后端系统完成的组件化改造，PC 和无线逻辑统一起来了；

- 将渲染层独立，渲染层后业务量逻辑层通过 JSON 数据对象交互；
- 前端开发同学完全掌握了 XTPL+JS 逻辑，有

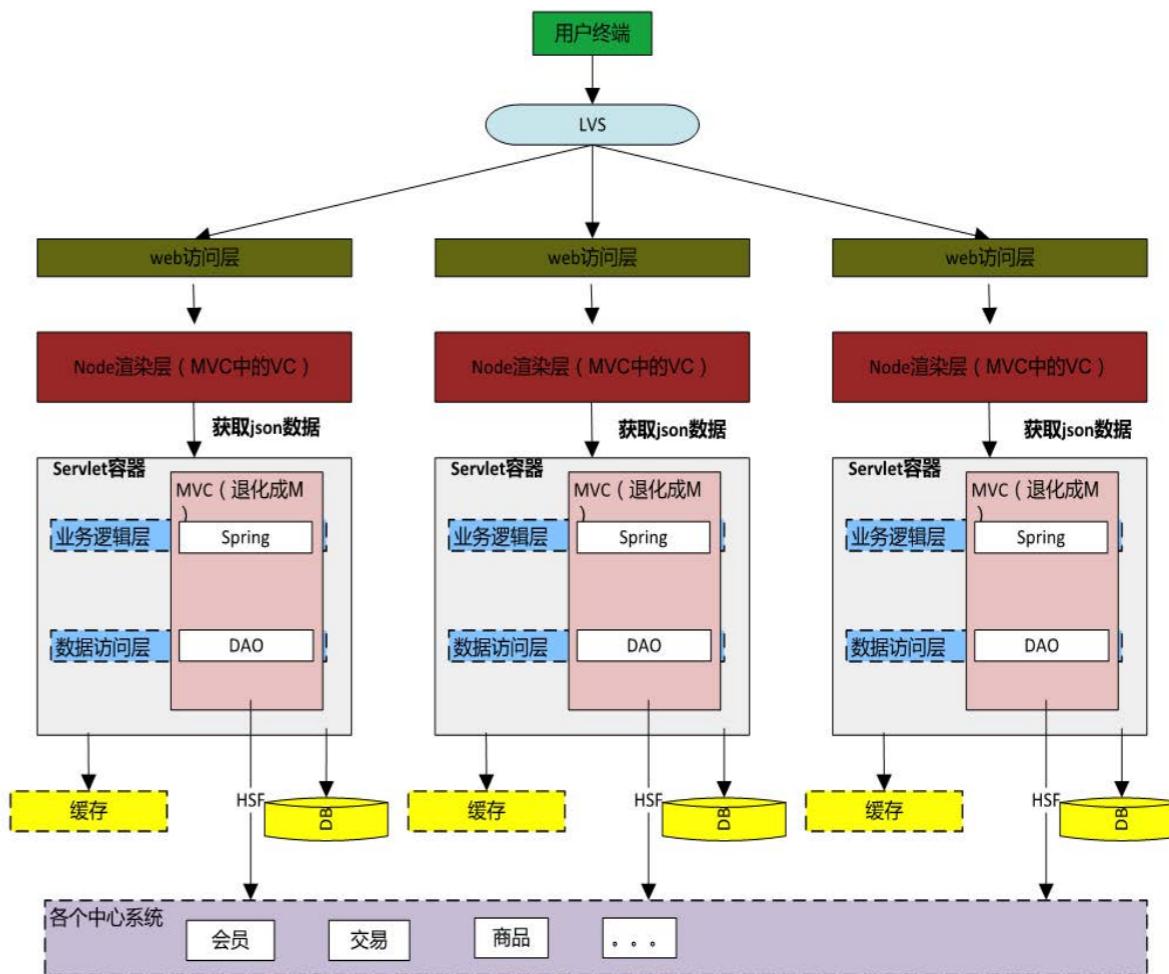


图5 Java Web兼容Node的模板层

- 更多的掌控力；
- 前端开发页面不需要依赖后端的 Java 系统，调试页面可以在 Node 中完成；
 - 系统的架构并没有增加一层，运维上也没有引入新的 Node 系统。

没有无往不能的神器

前面介绍了在现有 Java 体系下，要将 Node 替换 Java Web 其实是比较困难的，所以没有一个技术是无往不能的神器，但是是否意味着 Node 就没有应用场景了呢？肯定不是，下面这些情况下 Node 很有用武之地。

创业公司很合适，尤其当创始人之一是熟悉前端的同学的话，用 Node 实现 Web 系统很合适，

Node 和 PHP 一样具备快速发布的优势，代码 copy 上去就生效，甚至不需要重启服务器，这一点相比 Java 有很大的优势。当业务逻辑还没有非常复杂时，JavaScript 语言的弱点也没有暴露的那么明显，从系统的维护角度来说，不需要一个工作有两个角色的工程师完成，可以提升开发效率。

重页面交互轻业务逻辑的系统也适合 Node 来开发，说白了如果 Web 系统如有一半以上的工作量都是需要前端同学来完成的话，那还不如把整个系统都交给前端同学来维护。

如果公司的工程师都是全栈工程师能在不同语言之间自由切换，那么也就没有为的成本一说了。当然这个仍然要受到公司基础环境的约

束，如运维和中间件产品仍然不会同时开发两套。

小结

随着技术的不断进步我们的开发模式也在一直发生着变化，早期的页面渲染和业务逻辑全部集中在一起，如 ASP、PHP、JSP 技术，后来由于业务逻辑不断变得复杂，出现了 MVC 的开发框架，前后端工程师分工也越发清楚。中间也有过前端工程师负责整个渲染层和控制层的实现如 Extjs+Ajax 的开发模式，但是由于整个渲染是在浏览器端完成的受制于客户端渲染性能和搜索引擎的收录页面的硬缺陷很难成为主流。一直到今天前后端开发模式仍然是后端工程师管理 M 和 C，而由前端来实现 V，开发环境和运行环境是一套，所以开发上的耦合导致沟通和调试成本增加。直到 Node 的出现缓解了前后端开发上的耦合，但是这种分离仍然

是以增加运行时的维护成本来换取开发时的便利，所以我觉得还不是最佳实践。

本文给出的解决方案也是想兼顾开发时的便利而同时也不增加运行时的维护成本为出发点，当然每种方案都不是完美的，找到适合才是最重要的，随着 Java 中执行 JS 技术的不断成熟，我想开发环境和运行环境的分离肯定不久就将实现，前后端开发的耦合度也就最终解决。

作者简介

许令波，2009 年毕业加入淘宝，目前负责商品详情业务和稳定性相关工作，长期关注性能优化领域，参与了淘宝高访问量 Web 系统主要的优化项目，著有《深入分析 Java Web 技术内幕》一书。 @淘宝君山、<http://www.xulingbo.net>、xulingbo0201@163.com 可以联系到我。





你有想法，我来实现。

mart.coding.net



海量开发者
候选替补任你挑



专属项目经理
全程跟踪，进度监控



整合云端工具
开发流程全透明



扫一扫，关注码市微信服务号

Coding 码市

省钱 无需组建团队，海量开发者任你挑选，直接沟通！

省心 专属项目经理全程沟通监管，帮助梳理需求思路，提供最贴心服务！

省时间 全云端的开发管理工具，随时随地沟通、监控项目进度，快速高质完成项目！

InfoQ 中文站 2015迷你书



本期内容推荐：Docker背后的容器集群管理——从Borg到Kubernetes（一），解析微服务架构（二）微服务架构综述，多范式编程语言 - 以Swift为例，Oracle专家谈MySQL Cluster如何支持200M的QPS，杂谈：创业公司的产品开发与团队管理



开源启示录 第一季



顶尖技术团队访谈录 第二季



云生态专刊

《云生态专刊》是InfoQ为大家推出的一个新产品，目标是“打造中国最优质的云生态媒体”。