

架构师

ARCHITECT



解读2014 | Review

解读2014之Android篇：连接世界

推荐文章 | Article

新JavaScript库的激动人心之处

高效运维最佳实践

专题 | Topic

Web API设计方法论

亿级用户下的新浪微博平台架构

Netty百万级推送服务设计要点



卷首语

又到年根，作为一位技术媒体人开始频繁拜访各家公司，总结过去，面向未来，在这里把发现的一些新趋势和自己的感受梳理一下，就不透露公司名称了，供大家参考。

公有云的第三方评测将越来越受到重视，我指的并不是知名国际咨询公司提供的研究报告，而是来自于社区圈搞的评测结果。现在信息透明度很高，开源的测试工具也多，从磁盘 IO 到线程锁再到数据库性能，有社区大神开始评测国内主流的云厂商。这些实实在在的数据结果更接“地气”，对于正在对云服务做技术选型的开发者和公司来说，具有较高的参考价值。对于有些云厂商来说，可能面子有些挂不住，不过令人欣喜的是，更多的云厂商持有开放的心态，不忌惮结果，勇于面对，不断改进。

托管云、私有云、混合云将在 2015 年成为云厂商的重点方向。长期以来，公有云的市场营销和品牌战略让国内社区的注意力都放在了公有云上，不过这只是云计算市场的一块，增长很快，但体量不大。从国内现状来看，多数大中型企业因为信息安全、IT 设施复杂性、系统稳定性等原因，短期内不会上线公有云，所以其他类型的云有了发展机会，而且这块市场的体量很大。据我所知，目前几家主流云厂商包括内资和外企都提供非公有云的服务形式，并且也是他们在未来一年关注的重点。所谓“公有云赔本赚吆喝，私有云闷声发大财”的说法有一定的道理。

外资云是困难和机遇并存。困难的是，公有云项目迟迟难以落地，目前只有一家真正落地，其他项目还在评估和运作中，眼看国内的云计算市场发展的如火如荼，外资云有自己的技术和品牌优势，也是着急啊。目前外资云能做的一方面是积极寻求落地，另外一方面是积极发展非公有云服务，本来这几家外资云在这方面就有比较完备的解决方案，最近在通过一系列活动发出自己的声音。

研究性组织越来越重视业务价值。曾几何时，贝尔实验室的辉煌成就了无数个科学家和革命性技术，但对于 AT&T 来说，其产生的企业价值对不起投入。就是几年前，很多 IT 巨头还在强调各种研究院的学术独立性，但是现在的产业转型期中，企业更加现实，科研成果在一段

时间内必须要有看得见的经济效益，鼓励研究人员走出去，面向客户，解决实际问题。对于科学家来说，不知道是好事还是坏事。

O2O 的布局越来越重要，从我走访的几家垂直领域互联网公司来看，即使在线上包括移动端走的很好，也都在布局 O2O，打通线上线下的渠道链，形成比较完整的生态圈，并且能够成为规则的制定者，而不仅仅是参与者，目前面临的挑战包括支付形式的多样性、线下产品的运营等等。

大数据分析逐渐成为企业分析用户行为习惯并提出优化策略的重要手段，搭建和运营大数据分析的平台也成为很多企业的痛点，而第三方的数据分析平台又存在不可靠或者站错队的问题，不敢轻易使用，所以预计大数据这块的人才需要未来将比较旺盛。

云服务的差异化竞争将更加明显，我走访了几家云厂商，发现每家的定位都比较清晰，有的强调市场和运营优势，有的强调技术优势，有的关注普通消费者和个人站长，有的关注中型企业，由于公有云服务的投入成本比较高，BAT 几家财大气粗，其他国内云厂商都在寻找自己的定位，提供差异化的创新服务。

在 2015 年，整个国内的 IT 产业将有更多看点，而 InfoQ 也会从多个角度、通过多种形式进行全面而深入的报道，和大家一起成长。

——InfoQ 中 总编辑 崔康

QCon

全球软件开发大会

2015.4.23-25 北京国际会议中心



8
折
优惠

现在报名

(截至3月1日)

- 针对软件开发领域的热点与趋势，议定18个最具实践价值的专题，覆盖更广的开发者人群，为参会者提供更多选择

/可扩展、高可用架构设计/

/新兴大数据处理技术与工具/

/自动化运维/

/云计算平台构建与应用/

/团队建设/

/软件质量/

/敏捷之后，是什么/

/知名移动案例分析/

/挑战全栈开发/

/移动开发最佳实践/

/安全、隐私与风控/

/互联网金融背后的技术架构/

/微服务和响应式框架/

/Java和新锐语言/

/超越JavaScript/

/思考开源/

/基于大数据的机器学习和数据挖掘/

/技术创业/

抢票热线：010-64738142

会务咨询：qcon@cn.infoq.com

精彩内容策划中，讲师自荐/推荐，线索提供：speakers@cn.infoq.com

更多经典专题 精彩内容 敬请登录：www.qconbeijing.com



促进软件开发领域知识与创新的传播



中文 | 英文 | 日文 | 葡文 |

目 录

解读 2014 | Review

解读 2014 之 Android 篇：连接世界

专题 | Topic

Web API 设计方法论

亿级用户下的新浪微博平台架构

Netty 系列之 Netty 百万级推送服务设计要点

Java 多线程编程模式实战指南：Immutable Object 模式

观点 | Opinion

技术团队的情绪与效率

慎用 Java 日期格式化

Node.js 和 io.js 性能差异巨大

推荐文章 | Article

新 JavaScript 库的激动人心之处

高效运维最佳实践（01）：七字诀，不再憋屈的运维

解读 2014 之 Android 篇：连接世界

作者 郭亮

【编者按】 2014 年，整个 IT 领域发生了许多深刻而又复杂的变化，InfoQ 策划了“解读 2014”年终技术盘点系列文章，希望能够给读者清晰地梳理出技术领域在这一年的发展变化，回顾过去，继续前行。

本文为“解读 2014 之 Android 篇”，Android 从 2008 年发布，到 2014 年末已经 6 岁。经历了前几年的高速发展，Android 已经当之无愧的成为全球用户最多的手机操作系统。2014 年虽然不是 Android 发展最快的一年，却是变化最快、扩张最大的一年。最新版本的 Android 5.0 Lollipop 无论是用户体验还是系统性能都有着颠覆性的改变与提升。Material Design 的出现，更使 Android 设备在体验方面第一次和 iOS 站在了同一个高度。经历了一年多的开发与测试，谷歌也发布了第一款官方正版 IDE——Android Studio，功能强大堪比开发神器。Android Wear、Android TV、Android Auto 已经领先一步进入市场，越来越多的智能硬件都采用 Android 系统，希望借助 Android 生态环境来构建属于自己的市场。谷歌对于国内开发者也变得更加友好，全球最大的 Android 市场 Google Play 已经支持中国的开发者上传 App。本文作为 Android 这一年的总结，从系统本身、开发工具、硬件配置、国内外生态环境四方面介绍了 Android 这一年的发展与改变，并且结合当前市场大胆展望了 2015 年 Android 的发展方向。

系统

谷歌在 2014 年的 I/O 大会上发布了最新的操作系统 Android Lollipop，也就是 Android 5.0。Android Lollipop 是有史以来 Android 最大的一次改变。首先，在感官界面设计上，Android Lollipop 不仅使用了新的配色，同时使用了非常时尚的扁平设计，彻底迎来了 Android 系统的扁平化时代。此外，系统的多任务功能进行了一次基础性的重大调整。Android L 中用户将会拥有一个基于卡片的清单，其中呈现的并不是应用，而是任务。新的任务机制，能够节约大量的系统性能。另外全新的通知中心也不再乏味，当然还有大量的其它新特性，相信第三方系统插件的市场将会越来越小。

系统方面重重之重的改变应该是 Material Design。谷歌将 Material Design 定义为一种设计语言，其特点是能在将整个素材铺平的同时还遵循一定的物理材质的需求。Material Design 的设计风格可以让应用感觉更活泼、具有更丰富的颜色，以及动画效果更真实等等。从技术角度分析，Material Design 完美解决了两个非常大的需求，其一是阴影，它所有的阴影都是默认系统实现的，开发者无需去自定义。另一个是动画，可以说 Android 5.0 将动画应用到了各个角落，动画效果甚至要超过 iOS，并且其效果不是简单的贴图，更像是真实的投影。谷歌自家的应用都已经使用了 Material Design，对于开发商来说，越早使用 Material Design，不仅有机会得到 Google Play 或国内市场设计推荐，更有可能提升 App 的留存率。

Android 5.0 对于硬件的兼容性比之前的版本有了重大提升，原生系统就支持多种设备，同时支持手机、Android Wear、Android TV、Android Auto，并且谷歌发布了这些设备的 SDK，Google Play 已经可以上传 Wear App 和 TV App。基于 Android 5.0 的整个生态链已经全部打通。

工具

一年前 Google 发布 Android Studio 测试版的时候，笔者天真的以为再有三个月就会出正式版，这一等就是一年多的时间。但这种等待是值得的，单从美观上评价，自带的 Darcula 主题炫酷黑界面实在是高大上，极客范，相比而言 Eclipse 的黑色主题太 low 了。Android Studio 亮相之初就支持 Gradle，Gradle 集合了 Ant 和 Maven 的优点，不管是配置、编译、打包都非常棒。Android Studio 的编辑器非常的智能，除了吸收 Eclipse+ADT 的优点之外，还自带了多设备的实时预览，这对 Android 开发者来说简直是神器。提示补全对于开发来说意义重大，Studio 则更加智能，智能保存，从此再也不用每次都 Ctrl + S 了。熟悉 Studio 以后效率会大大提升。

Android 碎片化是让开发者非常痛苦的一件事情，一个 UI 需要去反复测试多个设备，Android Studio 解决了开发者的这个痛点，它支持多款设备的实时预览。Android Studio 还提供更多的特性比如内置终端、完善的插件机制，还可以安装 Markdown，你想要什么插件，直接搜索下载。Android Studio 自带版本控制系统如 GitHub、Git、SVN 等，可以直接 check out 你的项目。

硬件

在美国智能手机市场上，苹果可能是当之无愧的“王者”，但在全球范围内，三星的至尊地位则是无可置疑的。Android 设备包括高、中、低端产品，价格从几百元到五、六千元不等。从市场份额上看要高于 iPhone 系列。而国内市场用户最多的应该是小米，小米凭借 MIUI 成为了世界上最大的第三方 ROM 产商，迅速进入市值 100 亿美金公司队列。2014 年谷歌、三星、HTC、小米、魅族等等都发布了多款手机，这些手机的形态、尺寸、性能规格各异，大部分手机还无法升级到 Android 5.0 系统，Android 2.3 依然不死，Android 的碎片化程度越来越高。对于开发者来说应用的开发更加困难了。不过我们不再有理由去抱怨，因为 iOS 的多屏幕适配繁琐性不低于 Android。

2014 年三月谷歌正式发布了针对智能手机和其它可穿戴设备的全新平台——Android Wear。Android Wear 除了最基本的查看手机通知消息，以及记录用户运动情况以外，还可以通过 Google Play 市场下载应用实现很多用户意想不到的功能。Android Wear 的发布催生了一大批 Android Wear 设备的诞生，年末苹果发布了自己的智能手表，微软同样推出了自己的智能手环。虽然 Android Wear 抢先一步进入市场，但却没有激起太大的浪花。与苹果发布的 Apple Watch 相比，Android Wear 无论从外观上还是从软件功能上都要逊色很多。笔者认为未来还是会是苹果引领智能手表市场。但 Android Wear 的表现并不影响 Android 进入客厅的步伐，乐视 TV、小米电视、天猫盒子已经展开了激烈的竞争，搭载 Android L 系统的 TV 将会是市场的主力军，传统电视厂商都已经开始研发自己的智能 TV，笔者想说的是 Apple TV 竞争非常激烈。

2014 年的硬件产品不全是美好，还充满了各种遗憾。小米没有推出更具吸引力的产品、魅族预售后 2 个多月都拿不到真机、智能汽车还不是真的智能...但对笔者来说，最大的遗憾应该是 Google Glass 消失了。2013 年 Google Glass 被称为是最伟大的发明与改变，那时我们都感觉世界如此之小，每一位极客都希望能体验带着 Google Glass 去环游世界。但 2014 年，Google Glass 没有新的进展，I/O 大会只字未提。有可能是因为体验的问题，有可能是因为材料问题，有可能是用户失去了耐心，Google Glass 发展令人担忧。

生态

谷歌在 I/O 大会推出了最新移动操作系统、设计语言，并正式启动 Android“连接世界”战略，将 Android 带入汽车、客厅、可穿戴设备、健康管理等更广阔的领域，谷歌利用自己的开放优势，借助三星、华

为等第三方厂商正打算抢先在所有智能设备终端上布局。任何的设备，甚至灯泡都可以烧录 Android 系统，每个公司甚至个人都可以利用 Android 系统定制自己的智能硬件，然后利用已经成熟的应用生态圈去带动产品发展。对于 Android 来讲，这样的好处是发展迅速，但不足是产品质量很难保证，碎片化严重。而苹果是软硬件追求极致的公司，无论是 iPhone、Mac 还是 2015 即将上市的 Apple Watch 都堪称极客产品。谷歌显然也意识到了这个问题，从去年谷歌的一系列举措，我们可以发现谷歌将加大对于 Android 的控制，软件层面将会出现更多的谷歌原生应用，而硬件方向，谷歌将会挑选一些大的厂商合作推出相关产品，比如有消息称谷歌将会与国内电视厂商合作在春节时推出 Android TV。

谷歌正在大步前进，连接整个世界，小米搭着 Android 的肩膀，已经成为了巨人，微信已经成为世界上最大的 IM，老罗也火了一把。前两年有人说，很少有 CP 从 Android 身上赚到钱，只愿意开发 iOS App。2014 Google I/O 大会上谷歌公布了一组数据，原话是这样的“In fact, since last year's I/O, we have paid out over \$5 billion to developers on top of Google Play”，比 2013 年的 20 亿美金翻了 2.5 倍，这还仅仅是 Google Play 一个市场，从而可见 Android 市场潜力有多大。国内有大量的公司依靠 Android 不仅赚到了钱，而且抢到了市场份额，比如像各种 App 分发市场。Google Play 已于 2014 年支持中国开发者，国内的开发者终于不用冒着被封号的风险去使用淘宝上购买来的 Google Play 账号。但反观国内市场，整个移动领域的盈利模式还令人担忧，很多开发商还停留在刷数据、冲 KPI 阶段，移动广告公司扣量严重、积分墙无法通过市场等。如果无法赚到钱，投资人还会有多少耐心？

市场前景一片好，但小的团队、个人开发者的发展却变的更加的艰难。移动领域已经度过了发展最快的几年，市场需要求已经基本饱和，越来越多的资源向大的 CP 靠拢。两年前一个技术、一个产品经理开发一款 App，迅速拥有 20 万用户的现象已经很难出现。很多渠道都需要 Money，App 市场也开始搞竞价排名，更是使新入市、低实力的开发者毫无竞争力。市场刷榜从 iOS 感染到了 Android，打包党变的更加猖狂，目前市场上有将近 70% 的 App 都被二次打包，笔者身边有一位朋友，辛辛苦苦开发几个月的 App，上架才一周就被打包，并且自己的 App 说是盗版被下架，而真正的盗版却成了正版。国家对 Android App、分发市场的监管力度目前还比较小，有好处但坏处也不小。

国内外关于 Android 的技术环境还是相对薄弱，这与 Android 开发门槛低有很大的关系，国内的开发者社区活跃度比较低，优秀的技术分享更是非常稀缺。iOS 的技术氛围明显要强与 Android，iOS 有 objc.io 这

样世界一流的开发社区，很多优秀的技术博客产出了大量的高质量文章。如果有能力，组织一个高、精、尖的社区应该有很大的发展潜力。

2015

由于 Android 天生安全性没有 iOS 高，随着手机变成银行卡、支付宝的钥匙，移动安全将更加重要。移动安全一方面是基于手机用户，另一方面是保证大量 CP 的正版利益。开发一款 App 的成本越来越低，国内早已经出现类似于一键建站的站长工具，所以对开发者的要求更加严格。目前入驻移动领域的厂商数不胜数，面向 CP、面向企业级的服务将会很受欢迎，2015 年一定会出现更多类似于 Umeng 分享、ShareSDK、可集成 IM 这种第三方服务产品，这个市场竞争还不太激烈，可做的模块很多。Android App 将会向模块集成方向发展。

苹果的 CloudKit 提供了完善且有弹性的后端解决方案，其目的是帮助开发者减轻编写服务器代码和维护服务器的需求，从而降低开发 iOS 应用的成本，有助于维护 iOS 生态圈的繁荣。谷歌 2014 年 10 份收购了 Firebase，Firebase 与 CloudKit 属于同一款产品——BaaS（后端即服务：Backend as a Service）。BaaS 因是移动互联网才诞生出来的工具。如今 O2O 发展的势头猛烈，很多传统行业本身的核心竞争力不在技术或者 app 层面，移动应用只是为了承载核心服务。所以能够以一种简单的方式搭建一个可用的 App、轻应用、HTML5 页面是最好的选择，就像我们不需要自己去搭建一个推送服务器，不需要自己去做数据统计、数据分析一样。2015 年，更多的 BaaS 产品将会出现，移动产品中的的数据存储、文件管理、消息推送等服务将会直接由 BaaS 产品提供，大量移动开发者不再需要去搭建服务器，开发成本将大大降低。相信 APICloud、bmob 之类的平台会迅速成长起来，将提供更强大的功能。当然，BAT 也可能提供自己的 BaaS 产品，微信或许会为是降低开发门槛提供类似 Parse 的服务，阿里借助 BaaS 产品给开发者提供新的变现方式也是很有可能的。

谷歌会将更多的资源开放给中国市场，并且会和国内的厂商合作，加强对 Android 的控制。无论是因为国内的市场透明度不好，还是因为国内市场已无发展潜力，都是时候开发国际版 App 了。猎豹就是典型的例子，凭借大量海外高质量用户成功上市。

基于 Android 生态的智能硬件、软硬结合产品越来越多，但暂时不会颠覆行业，进入普通消费者的生活还需要市场培养。

Web API 设计方法论

作者 Mike Amundsen, 译者 吴海星

为 Web 设计、实现和维护 API 不仅仅是一项挑战；对很多公司来说，这是一项势在必行的任务。[本系列](#) 将带领读者走过一段旅程，从为 API 确定业务用例到设计方法论，解决实现难题，并从长远的角度看待在 Web 上维护公共 API。沿途将会有对有影响力的人物的访谈，甚至还有 API 及相关主题的推荐阅读清单。

这篇 InfoQ 文章是 **Web API 从开始到结束** 系列文章中的一篇。你可以在[这里](#) 进行订阅，以便能在有新文章发布时收到通知。

设计 Web API 不止是 URL、HTTP 状态码、头信息和有效负载。设计的过程——基本上是为了 API 的“观察和感受”——这非常重要，并且值得你付出努力。本文简要概括了一种同时发挥 HTTP 和 Web 两者优势的 API 设计方法论。并且它不仅对 HTTP 有效。如果有时你还需要通过 WebSockets、XMPP、MQTT 等实现同样的服务，大部分 API 设计的结果同样可用。可以让未来支持多种协议更容易实现和维护。

优秀的设计超越了URL、状态码、头信息和有效负载

一般来说，Web API 设计指南的重点是通用的功能特性，比如 URL 设计，正确使用状态码、方法、头信息之类的 HTTP 功能特性，以及持有序列化的对象或对象图的有效负载设计。这些都是重要的实现细节，但不太算得上 API 设计。并且正是 API 的设计--服务的基本功能特性的表达和描述方式--为 Web API 的成功和可用性做出了重要贡献。

一个优秀的设计过程或方法论定义了一组一致的、可重复的步骤集，可以在将一个服务器端服务组件输出为一个可访问的、有用的 Web API 时使用。那就是说，一个清晰的方法论可以由开发人员、设计师和软件架构师共享，以便在整个实现周期内帮助大家协同活动。一个成熟的方法论还可以随着时间的发展，随着每个团队不断发现改善和精简过程的方式而得到精炼，却不会对实现细节产生不利的影响。实际上，当实现细节和设计过程两者都有清晰的定义并相互分离时，实现细节

的改变（比如采用哪个平台、OS、框架和UI样式）可以独立于设计过程。

API 设计七步法

接下来我们要对 Richardson 和 Amundsen 合著的《[REST 风格的 Web API](#)》一书中所介绍的设计方法论做简要地概述。因为篇幅所限，我们不能深入探讨这一过程中的每一步骤，但这篇文章可以让你有个大概的认识。另外，读者可以用这篇概述作为指南，根据自己组织的技能和目标开发一个独有的 Web API 设计过程。

说明：是的，7步看起来有点儿多。实际上清单中有5个步骤属于设计，额外还有两个条目是实现和发布。最后这两个设计过程之外的步骤是为了提供一个从头到尾的体验。

你应该计划好根据需要重新迭代这些步骤。通过步骤2（绘制状态图）意识到在步骤1（列出所有组成部分）有更多工作要做。当你接近于写代码（步骤6）时，可能会发现第5步（创建语义档案）中漏了一些东西。关键是用这个过程暴露尽可能多的细节，并愿意回退一步或者两步，把前面漏掉的补上。迭代是构建更加完整的服务画面以及澄清如何将它暴露给客户端程序的关键。

步骤1：列出所有组成部分

第一步是列出客户端程序可能要从我们的服务中获取的，或要放到我们的服务中的所有数据片段。我们将这些称为语义描述符。语义是指它们处理数据在应用程序中的含义，描述符是指它们描述了在应用程序自身中发生了什么。注意，这里的视点是客户端，不是服务器端。将API设计成客户端使用的东西很重要。

比如说，在一个简单的待办事项列表应用中，你可能会找到下面这些语义描述符：

- id: 系统中每条记录的唯一标识符；
- title: 每个待办事项的标题；
- dateDue: 待办事项应该完成的日期；
- complete: 一个是否标记，表明待办事项是否已经完成了。

在一个功能完备的应用程序中，可能还会有很多语义描述符，涉及待办事项的分类（工作、家庭、园艺等），用户信息（用于多用户的实现）等等。不过为了突出过程本身，我们会保持它的简单性。

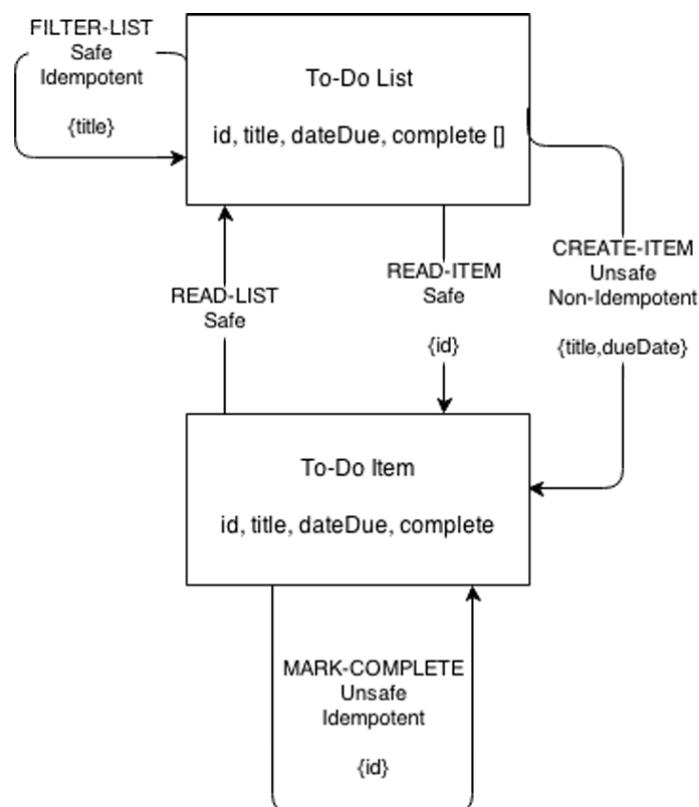
步骤 2：绘制状态图

下一步是根据建议的 API 绘制出状态图。图中的每个框都表示一种可能的表示——一个包含在步骤 1 中确定的或多个语义描述符的文档。你可以用箭头表示从一个框到下一个的转变——从一个状态到下一个状态。这些转变是由协议请求触发的。

在每次变化中还不用急着指明用哪个协议方法。只要标明变化是安全的（比如 HTTP GET），还是不安全/非幂等的（比如 HTTP POST），或者不安全/幂等的（PUT）。

说明：幂等动作是指重复执行时不会有无法预料的副作用。比如 HTTP PUT，因为规范说服务器应该用客户端传来的状态值替换目标资源的已有值，所以说它是幂等的。而 HTTP POST 是非幂等的，因为规范指出提交的值应该是追加到已有资源集合上的，而不是替换。

在这个案例中，我们这个简单的待办事项服务的客户端应用程序可能需要访问可用条目的清单，能过滤这个清单，能查看单个条目，并且能将条目标记为已完成。这些动作中很多都用状态值在客户端和服务器之间传递数据。比如 add-item 动作允许客户端传递状态值 title 和 dueDate。下面是一个说明那些动作的状态图。



这个状态图中展示的这些动作（也在下面列出来了）也是语义描述符--它们描述了这个服务的语义动作。

- read-list
- filter-list
- read-item
- create-item
- mark-complete

在你做这个状态图的过程中，你可能会发现自己漏掉了客户端真正想要或需要的动作或数据项。这是退回到步骤 1 的机会，添加一些新的描述符，并/或者在步骤 2 中改进状态图。

在你重新迭代过这两步之后，你应该对客户端跟服务交互所需的所有数据点和动作有了好的认识和想法。

步骤 3：调和魔法字符串

下一步是调和服务接口中的所有“魔法字符串”。“魔法字符串”全是描述符的名称--它们没有内在的含义，只是表示客户端跟你的服务通讯时将要访问的动作或数据元素。调和这些描述符名称的意思是指采用源自下面这些地方的，知名度更高的公共名称：

- [Schema.org](#)
- [microformats.org](#)
- [Dublin Core](#)
- [IANA Link Relation Values](#)

这些全是明确定义的、共享的名称库。当你服务接口使用来自这些源头的名称时，开发人员很可能之前见过并知道它们是什么意思。这可以提高 API 的可用性。

说明：尽管在服务接口上使用共享名称是个好主意，但在内部实现里可以不用（比如数据库里的数据域名称）。服务自身可以毫不困难地将公共接口名称映射为内部存储名称。

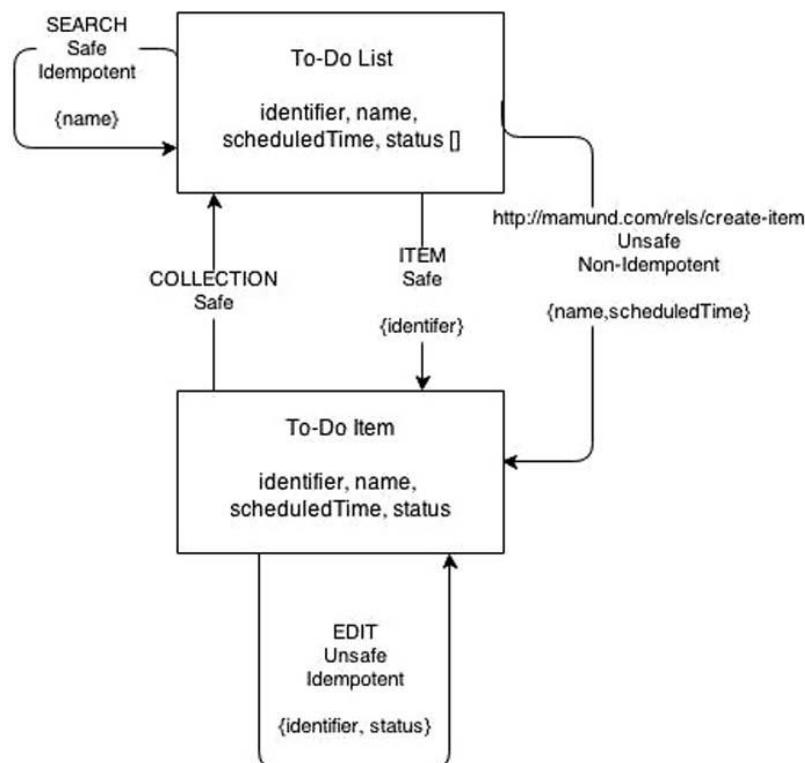
以待办事项服务为例，除了一个语义描述符 create-item，我能找到所有可接受的已有名称。为此我根据 [Web Linking RFC5988](#) 中的规则创建了一个具有唯一性的 URI。在给接口描述符选择知名的名称时需要折中。它们极少能跟你的内部数据存储元素完美匹配，不过那没关系。

这里是我的结果：

- id -> [来自 Dublin Core 的 identifier](#)

- title - 来自 Schema.org 的 name
- dueDate -> 来自 Schema.org 的 scheduledTime
- complete -> 来自 Schema.org 的 status
- read-list -> 来自 IANA Link Relation Values 的 collection
- filter-list -> 来自 IANA Link Relation Values 的 search
- read-item -> 来自 IANA Link Relation Values 的 item
- create-item -> 用 RFC5988 的 <http://mamund.com/rels/create-item>
- mark-complete - 来自 IANA Link Relation Values 的 edit

经过名称调和，我的状态图变成了下面这样：



步骤 4：选一个媒体类型

API 设计过程的下一步是选一个媒体类型，用来在客户端和服务器端之间传递消息。Web 的特点之一是数据是通过统一的接口作为标准化文档传输的。选择同时支持数据描述符（比如"identifier"、"status"等）和动作描述符（比如"search"、"edit"等）的媒体类型很重要。有相当多可用的格式。

在我写这篇文章时，一些顶尖的超媒体格式是（排名不分先后）：

- 超文本标记语言 ([HTML](#))
- 超文本应用程序语言([HAL](#))

- Collection+JSON ([Cj](#))
- [Siren](#)
- [JSON-API](#)
- 交换表达式的统一基础 ([UBER](#))

让所选择的媒体类型适用于你的目标协议也很重要。大多数开发人员喜欢用 [HTTP](#) 协议做服务接口。然而 [WebSockets](#)、[XMPP](#)、[MQTT](#) 和 [CoAP](#) 也会用--特别是对于高速、短消息、端到端的实现。

在这个例子中，我会以 HTML 为消息格式，并采用 HTTP 协议。HTML 有所有数据描述符所需的支持(用于列表，用于条目，用于数据元素)。它也有足够的动作描述符支持 (<A>用于安全链接，<FORM method="get">用于安全转变，<FORM method="post">用于非安全转变）。

注意：在这个状态图中，“编辑”动作是幂等的（比如 HTTP PUT），并且 HTML 仍然没有对 PUT 的原生支持。在这个例子中，我会添加一个域来将 HTML 的 POST 做成幂等的。

好了，现在我可以基于那个状态图创建一些样例表示来“试试”这个接口了。对我们的例子而言，只有两个表示要渲染：“待办事项列表”和“待办事项条目”表示。

```
<html>
<head>
    <!-- for test display only -->
    <title>To Do List</title>
    <style>
        .name, .scheduledTime, .status, .item {display:block}
    </style>
</head>
<body>
    <!-- for test display only -->
    <h1>To-Do List</h1>
    <!-- to-do list collection -->
    <ul>
        <li>
            <a href="/list/1" rel="item" class="item">
                <span class="identifier">1</span>
            </a>
            <span class="name">First item in the list</span>
            <span class="scheduledTime">2014-12-01</span>
            <span class="status">pending</span>
        </li>
        <li>
            <a href="/list/2" rel="item" class="item">
                <span class="identifier">2</span>
            </a>
            <span class="name">Second item in the list</span>
            <span class="scheduledTime">2014-12-01</span>
            <span class="status">pending</span>
        </li>
    </ul>
</body>
```

```

<li>
  <a href="/list/3" rel="item" class="item">
    <span class="identifier">3</span>
  </a>
  <span class="name">Third item in the list</span>
  <span class="scheduledTime">2014-12-01</span>
  <span class="status">complete</span>
</li>
</ul>
<!-- search transition --&gt;
&lt;form method="get" action="/list/" class="search"&gt;
  &lt;legend&gt;Search&lt;/legend&gt;
  &lt;input name="name" class="identifier" /&gt;
  &lt;input type="submit" value="Name Search" /&gt;
&lt;/form&gt;
<!-- create-item transition --&gt;
&lt;form method="post" action="/list/" class=""&gt;
  &lt;legend&gt;Create Item&lt;/legend&gt;
  &lt;input name="name" class="name" /&gt;
  &lt;input name="scheduledTime" class="scheduledTime" /&gt;
  &lt;input type="submit" value="Create Item" /&gt;
&lt;/form&gt;
&lt;/body&gt;
&lt;/html&gt;
</pre>

```

图 1 用 HTML 表示待办事项列表集合

```

<html>
  <head>
    <!-- for test display only -->
    <title>To Do List</title>
    <style>
      .name, .scheduledTime, .status, .item, .collection {display:block}
    </style>
  </head>
  <body>
    <!-- for test display only -->
    <h1>To-Do Item</h1>
    <a href="/list/" rel="collection" class="collection">Back to List</a>
    <!-- to-do list collection -->
    <ul>
      <li>
        <a href="/list/1" rel="item" class="item">
          <span class="identifier">1</span>
        </a>
        <span class="name">First item in the list</span>
        <span class="scheduledTime">2014-12-01</span>
        <span class="status">pending</span>
      </li>
    </ul>
    <!-- edit transition -->
    <form method="post" action="/list/1" class="edit">
      <legend>Update Status</legend>
      <input type="hidden" name="etag" value="q1w2e3r4t5y6" class="etag" />
      <input type="text" name="status" value="pending" class="status" />
      <input type="submit" value="Update" />
    </form>
  </body>
</html>

```

图 2 用 HTML 表示待办事项条目

记住，在你做状态图的表示样例时，可能会发现之前的步骤中有所遗漏（比如漏掉描述符，动作描述符中有幂等之类的变化等）。那也没关系。现在就是解决所有这些问题的时机-- 在你把这个设计变成代码之前。

等你对表示完全满意之后，在开始写代码之前还有一个步骤——创建语义档案。

步骤 5： 创建语义档案

语义档案是一个文档，其中列出了设计中的所有描述符，包括对开发人员构建客户端和服务器端实现有帮助的所有细节。这个档案是一个实现指南，不是实现描述。这个差别很重要。

服务描述符的格式

服务描述文档格式已经出现了相当长一段时间了，并且当你想给已有的服务实现生产代码或文档时很方便。确实有很多种格式。

在我写这篇文章时，顶级竞争者有：

- Web 服务定义语言([WSDL](#))
- 原子服务描述([AtomSvc](#))
- Web 应用程序描述语言([WADL](#))
- [Blueprint](#)
- [Swagger](#)
- REST 风格的应用程序建模语言([RAML](#))
- 档案的格式
- 现在只有几种档案格式。我们推荐下面两种：
- 应用级语义档案 ([ALPS](#))
- [JSON-LD + Hydra](#)

这两个都比较新。JSON-LD 规范在 2014 年早期达成了 W3C 推荐状态。Hydra 仍是一个非官方草案（本文写成时还是），有一个活跃的开发者社区。ALPS 仍处于 IETF 的早期草案阶段。

因为档案文档的理念是要描述一个问题空间的现实生活方面（不只是那一空间中的单一实现），所以其格式跟典型的描述格式十分不同。

```
<html>
<alps version="1.0">
  <doc>
    ALPS profile for InfoQ article on "API Design Methodology"
  </doc>
  <!-- data descriptors -->
  <descriptor id="identifier" type="semantic" ref=" />
  <descriptor id="name" type="semantic" ref=" />
  <descriptor id="scheduledTime" type="semantic" ref=" />
  <descriptor id="status" type="semantic" ref=" />
  <!-- action descriptors -->
  <descriptor id="collection" type="safe" ref=" />
  <descriptor id="item" type="safe" ref=">
    <descriptor href="#identifier" />
  </descriptor>
  <descriptor id="search" type="safe" ref=">
    <descriptor href="#name" />
  </descriptor>
  <descriptor id="create-item" type="unsafe" ref=">
    <descriptor href="#name" />
    <descriptor href="scheduledTime" />
  </descriptor>
  <descriptor id="edit" type="idempotent" ref=">
    <descriptor href="#identifier" />
    <descriptor href="#status" />
  </descriptor>
</alps>
```

图 3 ALPS 格式的待办事项列表语义档案

你会注意到，这个文档就像一个基本的词汇表，包含了待办事项服务接口中所有可能的数据值和动作--就是这个理念。同意遵循这个档案的服务可以自行决定它们的协议、消息格式甚至 URL。同意接受这个档案的客户端将会构建为可以识别，如果合适的话，启用这个文档中的描述符。

这种格式也很适合生成人类可读的文档，分析相似的档案，追踪哪个档案用得最广泛，甚至生成状态图。但那是另外一篇文章的课题了。

现在你有完整的已调和名称的描述符清单，已标记的状态图，以及一个语义档案文档，可以开始准备编码实现样例服务器和客户端了。

步骤 6：写代码

到了这一步，你应该可以将设计文档（状态图和语义档案）交给服务器和客户端程序的开发人员了，让他们开始做具体的实现。

HTTP 服务器应该实现在第 2 步中创建的状态图，并且来自客户端的请求应该触发正确的状态转变。服务发送的每个表示都应该用第 3 步中选好的格式，并且应该包含一个第 4 步中创建的指向一个档案的链接。响应中应该包含相应的超媒体控件，实现了在状态图中显示、并在档案文档中描述的动作。客户端和服务器端开发人员在这时可以创建相对独立的实现，并用测试验证其是否遵守了状态图和档案。

有了稳定的可运行代码，还有一步要做：发布。

步骤 7：发布你的 API

Web API 应该至少发布一个总能给客户端响应的 URL -- 即便是在遥远的将来。我将其称为“看板 URL” -- 每个人都知道的。发布档案文档也是个好主意，服务的新实现可以在响应中链接它。你还可以发布人类可读的文档、教程等，以帮助开发人员理解和使用你的服务。

做好这个之后，你应该有了一个设计良好的、稳定的、可访问的服务运行起来了，随时可以用。

总结

本文讨论了为 Web 设计 API 的一组步骤。重点是让数据和动作描述正确，并以机器可读的方式记录它们，以便让人类开发人员即便不直接接触也能轻松为这个设计实现客户端和服务器端。

这些步骤是：

- 1、列出所有组成部分
收集客户端跟服务交互所需的所有数据元素。
- 2、绘制状态图
记录服务提供的所有动作（状态变化）
- 3、调和魔法字符串
整理你的公开接口以符合（尽可能）知名的名称
- 4、选择媒体类型
评审消息格式，找到跟目标协议的服务转变最贴近的那个。
- 5、创建语义档案
编写一个档案文档，定义服务中用的所有描述符。

6、写代码

跟客户端和服务器端开发人员分享档案文档，并开始写代码测试跟档案/状态图的一致性，并在有必要时进行调整。

7、发布你的 API

发布你的"看板 URL"和档案文档，以便其他人可以用他们创建新的服务以及/或者客户端程序。

在设计过程中，你可能会发现有遗漏的元素，需要重做某些步骤，以及要做一些折中的决定。这在设计过程中出现得越早越好。将来开发人员要求用新的格式和协议实现时，你还有可能用这个 API 设计。

最后，这个方法论只是为 Web API 设计过程创建一种可靠、可重复、一致的设计过程的一种可能方式。在你做这个例子时，可能会发现插入一些额外的步骤，或者缩减一些会更好用，并且--当然--消息格式和协议决策在不同案例中可能也会发生变化。

希望这篇文章能给你一些启发，让你知道如何给自己的组织以及/或者团队创建一个最佳的 API 设计方法论。

关于作者

Mike Amundsen 是 Layer 7 科技的首席 API 架构师，帮助人们为 Web 创建优秀的 API。国际知名作者和讲师，Mike 在欧美四处旅行，就分布式网络架构、Web 应用程序开发、云计算和其他题目提供咨询和演讲。他的名下有十多本书。

为 Web 设计、实现和维护 API 不仅仅是一项挑战；对很多公司来说，这是一项势在必行的任务。[本系列](#)将带领读者走过一段旅程，从为 API 确定业务用例到设计方法论，解决实现难题，并从长远的角度看待在 Web 上维护公共 API。沿途将会有对有影响力的人物的访谈，甚至还有 API 及相关主题的推荐阅读清单。

这篇 InfoQ 文章是 **Web API 从开始到结束** 系列文章中的一篇。你可以在[这里](#)进行订阅，以便能在有新文章发布时收到通知。

查看英文原文：[A Web API Design Methodology](#)

亿级用户下的新浪微博平台架构

作者 卫向军

【编者按】《博文共赏》是 InfoQ 中文站新推出的一个专栏，精选来自国内外技术社区和个人博客上的技术文章，让更多的读者朋友受益，本栏目转载的内容都经过原作者授权。文章推荐可以发送邮件到 editors@cn.infoq.com。

序言

新浪微博在 2014 年 3 月公布的月活跃用户（MAU）已经达到 1.43 亿，2014 年新年第一分钟发送的微博达 808298 条，如此巨大的用户规模和业务量，需要高可用（HA）、高并发访问、低延时的强大后台系统支撑。

微博平台第一代架构为 LAMP 架构，数据库使用的是 MyIsam，后台用的是 php，缓存为 Memcache。

随着应用规模的增长，衍生出的第二代架构对业务功能进行了模块化、服务化和组件化，后台系统从 php 替换为 Java，逐渐形成 SOA 架构，在很长一段时间支撑了微博平台的业务发展。

在此基础上又经过长时间的重构、线上运行、思索与沉淀，平台形成了第三代架构体系。

我们先看一张微博的核心业务图（如图 1），是不是非常复杂？但这已经是一个简化的不能再简化的业务图了，第三代技术体系就是为了保障在微博核心业务上快速、高效、可靠地发布新产品新功能。

第三代技术体系

微博平台的第三代技术体系，使用正交分解法建立模型：在水平方向，采用典型的三级分层模型，即接口层、服务层与资源层；在垂直方向，进一步细分为业务架构、技术架构、监控平台与服务治理平台。图 2 是平台的整体架构图。

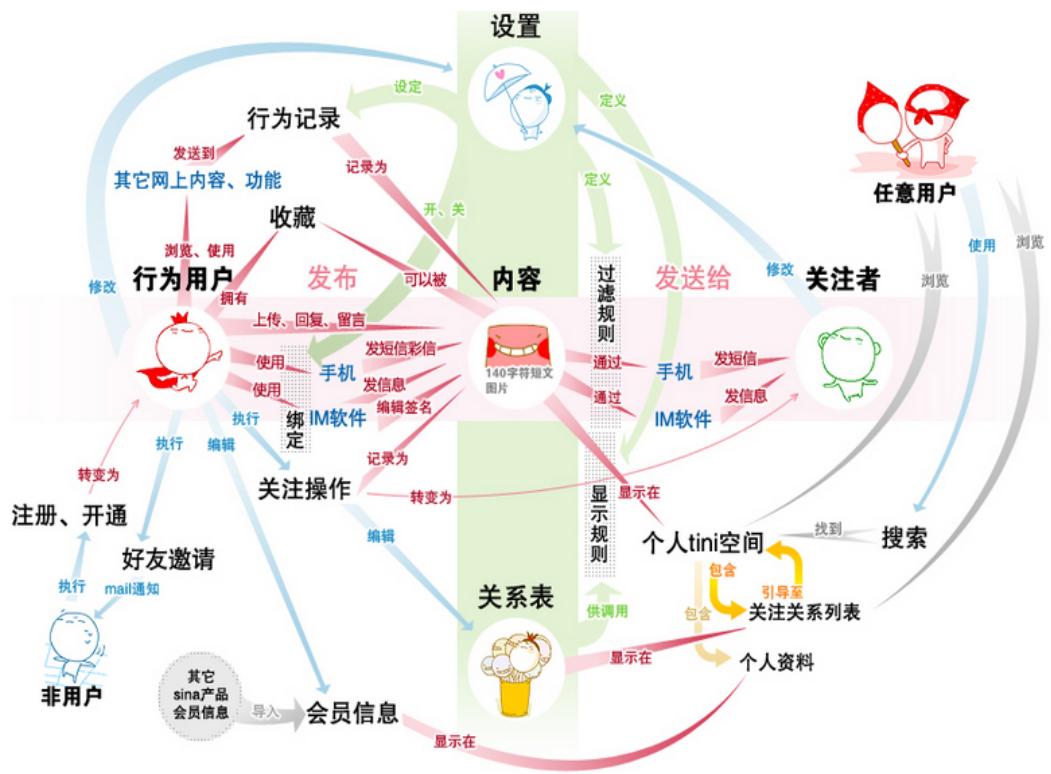


图1

微博平台架构

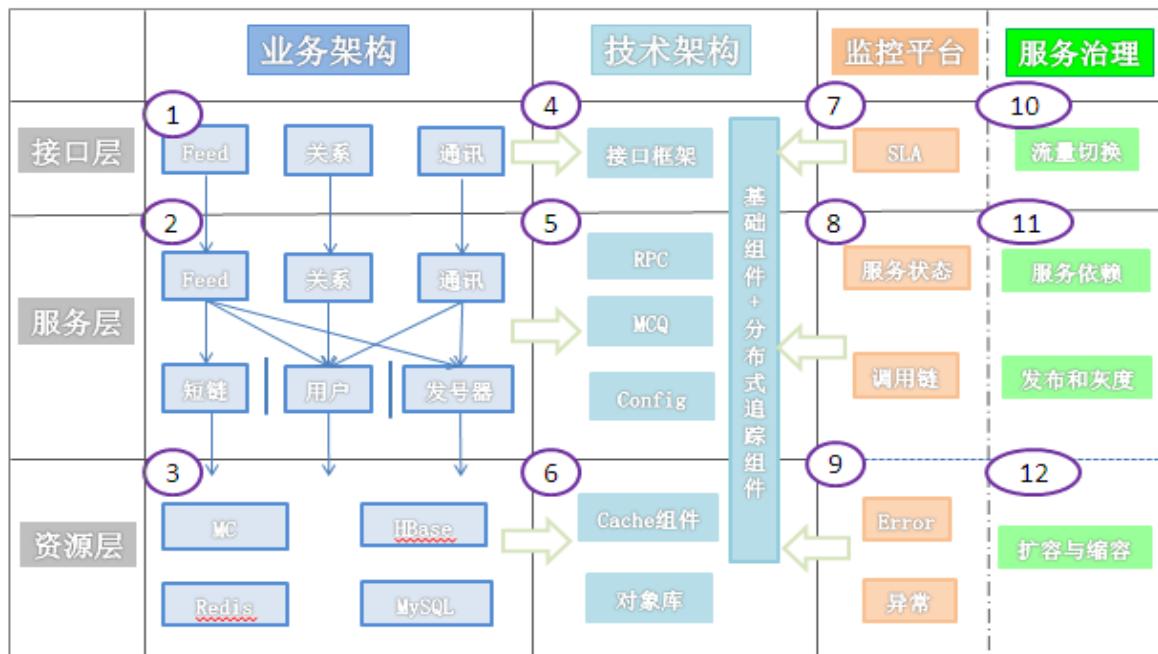


图2

如图 2 所示，正交分解法将整个图分解为 $3 \times 4 = 12$ 个区域，每个区域代表一个水平维度与一个垂直维度的交点，相应的定义这个区域的核心功能点，比如区域 5 主要完成服务层的技术架构。

下面详细介绍水平方向与垂直方向的设计原则，尤其会重点介绍 4、5、6 中的技术组件及其在整个架构体系中的作用。

水平分层

水平维度的划分，在大中型互联网后台业务系统的设计中非常基础，在平台的每一代技术体系中都有体现。这里还是简单介绍一下，为后续垂直维度的延伸讲解做铺垫：

接口层主要实现与 Web 页面、移动客户端的接口交互，定义统一的接口规范，平台最核心的三个接口服务分别是内容（Feed）服务、用户关系服务及通讯服务（单发私信、群发、群聊）。

服务层主要把核心业务模块化、服务化，这里又分为两类服务，一类为原子服务，其定义是不依赖任何其他服务的服务模块，比如常用的短链服务、发号器服务都属于这一类。图中使用泳道隔离，表示它们的独立性。另外一类为组合服务，通过各种原子服务和业务逻辑的组合来完成服务，比如 Feed 服务、通讯服务，它们除了本身的业务逻辑，还依赖短链、用户及发号器服务。

资源层主要是数据模型的存储，包含通用的缓存资源 Redis 和 Memcached，以及持久化数据库存储 MySQL、HBase，或者分布式文件系统 TFS 以及 Sina S3 服务。

水平分层有一个特点，依赖关系都是从上往下，上层的服务依赖下层，下层的服务不会依赖上层，构建了一种简单直接的依赖关系。

与分层模型相对应，微博系统中的服务器主要包括三种类型：前端机（提供 API 接口服务）、队列机（处理上行业务逻辑，主要是数据写入）和存储（mc、mysql、mcq、redis、HBase 等）。

垂直延伸技术架构

随着业务架构的发展和优化，平台研发实现了许多卓越的中间件产品，用来支撑核心业务，这些中间件由业务驱动产生，随着技术组件越来

越丰富，形成完备的平台技术框架，大大提升了平台的产品研发效率和业务运行稳定性。

区别于水平方向上层依赖下层的关系，垂直方向以技术框架为地基支撑点，向两侧驱动影响业务架构、监控平台、服务治理平台，下面介绍一下其中的核心组件。

接口层 Web V4 框架

接口框架简化和规范了业务接口开发工作，将通用的接口层功能打包到框架中，采用了 Spring 的面向切面（AOP）设计理念。接口框架基于 Jersey 进行二次开发，基于 annotation 定义接口(url, 参数)，内置 Auth、频次控制、访问日志、降级功能，支撑接口层监控平台与服务治理，同时还有自动化的 Bean-json/xml 序列化。

服务层框架

服务层主要涉及 RPC 远程调用框架以及消息队列框架，这是微博平台在服务层使用最为广泛的两个框架。

MCQ 消息队列

消息队列提供一种先入先出的通讯机制，在平台内部，最常见的场景是将数据的落地操作异步写入队列，队列处理程序批量读取并写入 DB，消息队列提供的异步机制加快了前端机的响应时间，其次，批量的 DB 操作也间接提高了 DB 操作性能，另外一个应用场景，平台通过消息队列，向搜索、大数据、商业运营部门提供实时数据。

微博平台内部大量使用的 MCQ (SimpleQueue Service Over Memcache) 消息队列服务，基于 MemCache 协议，消息数据持久化写入 BerkeleyDB，只有 get/set 两个命令，同时也非常容易做监控 (stats queue)，有丰富的 client library，线上运行多年，性能比通用的 MQ 高很多倍。

Motan RPC 框架

微博的 Motan RPC 服务，底层通讯引擎采用了 Netty 网络框架，序列化协议支持 Hessian 和 Java 序列化，通讯协议支持 Motan、http、tcp、mc 等，Motan 框架在内部大量使用，在系统的健壮性和服务治理方面，有较为成熟的技术解决方案，健壮性上，基于 Config 配置管理服务实

现了 High Availability 与 Load Balance 策略（支持灵活的 FailOver 和 FailFast HA 策略，以及 Round Robin、LRU、Consistent Hash 等 Load Balance 策略），服务治理方面，生成完整的服务调用链数据，服务请求性能数据，响应时间（Response Time）、QPS 以及标准化 Error、Exception 日志信息。

资源层框架

资源层的框架非常多，有封装 MySQL 与 HBase 的 Key-List DAL 中间件、有定制化的计数组件，有支持分布式 MC 与 Redis 的 Proxy，在这些方面业界有较多的经验分享，我在这里分享一下平台架构的对象库与 SSD Cache 组件。

对象库

对象库支持便捷的序列化与反序列化微博中的对象数据：序列化时，将 JVM 内存中的对象序列化写入在 HBase 中并生成唯一的 ObjectID，当需要访问该对象时，通过 ObjectID 读取，对象库支持任意类型的对象，支持 PB、JSON、二进制序列化协议，微博中最大的应用场景将微博中引用的视频、图片、文章统一定义为对象，一共定义了几十种对象类型，并抽象出标准的对象元数据 Schema，对象的内容上传到对象存储系统（Sina S3）中，对象元数据中保存 Sina S3 的下载地址。

SSDCache

随着 SSD 硬盘的普及，优越的 IO 性能使其被越来越多地用于替换传统的 SATA 和 SAS 磁盘，常见的应用场景有三种：1) 替换 MySQL 数据库的硬盘，目前社区还没有针对 SSD 优化的 MySQL 版本，即使这样，直接升级 SSD 硬盘也能带来 8 倍左右的 IOPS 提升；2) 替换 Redis 的硬盘，提升其性能；3) 用在 CDN 中，加快静态资源加载速度。

微博平台将 SSD 应用在分布式缓存场景中，将传统的 Redis/MC + Mysql 方式，扩展为 Redis/MC + SSD Cache + Mysql 方式，SSD Cache 作为 L2 缓存使用，第一降低了 MC/Redis 成本过高，容量小的问题，也解决了穿透 DB 带来的数据库访问压力。

垂直的监控与服务治理

随着服务规模和业务变得越来越复杂，即使业务架构师也很难准确地描述服务之间的依赖关系，服务的管理运维变得越来越难，在这个背景

下，参考 google 的 dapper 和 twitter 的 zipkin，平台实现了自己的大型分布式追踪系统 WatchMan。

WatchMan 大型分布式追踪系统

如其他大中型互联网应用一样，微博平台由众多的分布式组件构成，用户通过浏览器或移动客户端的每一个 HTTP 请求到达应用服务器后，会经过很多个业务系统或系统组件，并留下足迹（footprint）。但是这些分散的数据对于问题排查，或是流程优化都帮助有限。对于这样一种典型的跨进程/跨线程的场景，汇总收集并分析这类日志就显得尤为重要。另一方面，收集每一处足迹的性能数据，并根据策略对各子系统做流控或降级，也是确保微博平台高可用的重要因素。要能做到追踪每个请求的完整调用链路；收集调用链路上每个服务的性能数据；能追踪系统中所有的 Error 和 Exception；通过计算性能数据和比对性能指标（SLA）再回馈到控制流程（control flow）中，基于这些目标就诞生了微博的 Watchman 系统。

该系统设计的一个核心原则就是低侵入性（non-invasiveness）：作为非业务组件，应当尽可能少侵入或者不侵入其他业务系统，保持对使用方的透明性，可以大大减少开发人员的负担和接入门槛。基于此考虑，所有的日志采集点都分布在技术框架中间件中，包括接口框架、RPC 框架以及其他资源中间件。

WatchMan 由技术团队搭建框架，应用在所有业务场景中，运维基于此系统完善监控平台，业务和运维共同使用此系统，完成分布式服务治理，包括服务扩容与缩容、服务降级、流量切换、服务发布与灰度。

结尾

现在，技术框架在平台发挥着越来越重要的作用，驱动着平台的技术升级、业务开发、系统运维服务，本文限于篇幅限制，没有展开介绍，后续会不断地介绍核心中间件的设计原则和系统架构。

关于作者

卫向军（[@卫向军](#) [微博](#)），毕业于北京邮电大学，现任微博平台架构师，先后在微软、金山云、新浪微博从事技术研发工作，专注于系统架构设计、音视频通讯系统、分布式文件系统和数据挖掘等领域

Netty 系列之 Netty 百万级推送服务设计要点

作者 李林锋

1. 背景

1.1. 话题来源

最近很多从事移动互联网和物联网开发的同学给我发邮件或者微博私信我，咨询推送服务相关的问题。问题五花八门，在帮助大家答疑解惑的过程中，我也对问题进行了总结，大概可以归纳为如下几类：

- 1) Netty 是否可以做推送服务器？
- 2) 如果使用 Netty 开发推送服务，一个服务器最多可以支撑多少个客户端？
- 3) 使用 Netty 开发推送服务遇到的各种技术问题。

由于咨询者众多，关注点也比较集中，我希望通过本文的案例分析和对推送服务设计要点的总结，帮助大家在实际工作中少走弯路。

1.2. 推送服务

移动互联网时代，推送(Push)服务成为 App 应用不可或缺的重要组成部分，推送服务可以提升用户的活跃度和留存率。我们的手机每天接收到各种各样的广告和提示消息等大多数都是通过推送服务实现的。

随着物联网的发展，大多数的智能家居都支持移动推送服务，未来所有接入物联网的智能设备都将是推送服务的客户端，这就意味着推送服务未来会面临海量的设备和终端接入。

1.3. 推送服务的特点

移动推送服务的主要特点如下：

- 1) 使用的网络主要是运营商的无线移动网络，网络质量不稳定，例如在地铁上信号就很差，容易发生网络闪断；
- 2) 海量的客户端接入，而且通常使用长连接，无论是客户端还是服务端，资源消耗都非常大；

- 3) 由于谷歌的推送框架无法在国内使用, Android 的长连接是由每个应用各自维护的, 这就意味着每台安卓设备上会存在多个长连接。即便没有消息需要推送, 长连接本身的心跳消息量也是非常巨大的, 这就会导致流量和耗电量的增加;
- 4) 不稳定: 消息丢失、重复推送、延迟送达、过期推送时有发生;
- 5) 垃圾消息满天飞, 缺乏统一的服务治理能力。

为了解决上述弊端, 一些企业也给出了自己的解决方案, 例如京东云推出的推送服务, 可以实现多应用单服务单连接模式, 使用 AlarmManager 定时心跳节省电量和流量。

2. 智能家居领域的一个真实案例

2.1. 问题描述

智能家居 MQTT 消息服务中间件, 保持 10 万用户在线长连接, 2 万用户并发做消息请求。程序运行一段时间之后, 发现内存泄露, 怀疑是 Netty 的 Bug。其它相关信息如下:

- 1) MQTT 消息服务中间件服务器内存 16G, 8 个核心 CPU;
- 2) Netty 中 boss 线程池大小为 1, worker 线程池大小为 6, 其余线程分配给业务使用。该分配方式后来调整为 worker 线程池大小为 11, 问题依旧;
- 3) Netty 版本为 4.0.8.Final。

2.2. 问题定位

首先需要 dump 内存堆栈, 对疑似内存泄露的对象和引用关系进行分析, 如下所示:

Name	Instance count	Difference
char[]	3,400,328	-1,448,882 (-30 %)
java.util.HashMap\$Entry	3,956,168	+2,597,922 (+191 %)
io.netty.channel.ChannelOutboundBuffer\$Entry	2,624,096	+2,240,000 (+583 %)
java.nio.DirectByteBuffer	1,529,161	-672,671 (-31 %)
java.lang.Object[]	1,502,173	-343,175 (-19 %)
int[]	242,299	+5,121 (+2 %)
io.netty.util.concurrent.ScheduledFutureTask	1,101,894	+1,089,886 (+9076 %)
byte[]	644,180	-328,374 (-34 %)
io.netty.channel.DefaultChannelHandlerContext	656,009	+560,000 (+583 %)
sun.misc.Cleaner	1,135,417	-463,772 (-29 %)
java.util.HashMap\$Entry[]	511,599	+308,732 (+152 %)
io.netty.buffer.UnpooledUnsafeDirectByteBuf	550,545	-290,964 (-35 %)
java.lang.String	1,605,781	-607,173 (-27 %)
java.nio.DirectByteBuffer\$Deallocator	1,135,415	-463,774 (-29 %)
io.netty.util.concurrent.PromiseTask\$RunnableAdapter	1,101,886	+1,089,886 (+9082 %)

我们发现 Netty 的 ScheduledFutureTask 增加了 9076%，达到 110W 个左右的实例，通过对业务代码的分析发现用户使用 IdleStateHandler 用于在链路空闲时进行业务逻辑处理，但是空闲时间设置的比较大，为 15 分钟。

Netty 的 IdleStateHandler 会根据用户的使用场景，启动三类定时任务，分别是：ReaderIdleTimeoutTask、WriterIdleTimeoutTask 和 AllIdleTimeoutTask，它们都会被加入到 NioEventLoop 的 Task 队列中被调度和执行。

由于超时时间过长，10W 个长链接链路会创建 10W 个 ScheduledFutureTask 对象，每个对象还保存有业务的成员变量，非常消耗内存。用户的持久代设置的比较大，一些定时任务被老化到持久代中，没有被 JVM 垃圾回收掉，内存一直在增长，用户误认为存在内存泄露。

事实上，我们进一步分析发现，用户的超时时间设置的非常不合理，15 分钟的超时达不到设计目标，重新设计之后将超时时间设置为 45 秒，内存可以正常回收，问题解决。

2.3. 问题总结

如果是 100 个长连接，即便是长周期的定时任务，也不存在内存泄露问题，在新生代通过 minor GC 就可以实现内存回收。正是因为十万级的长连接，导致小问题被放大，引出了后续的各种问题。

事实上，如果用户确实有长周期运行的定时任务，该如何处理？对于海量长连接的推送服务，代码处理稍有不慎，就满盘皆输，下面我们针对 Netty 的架构特点，介绍下如何使用 Netty 实现百万级客户端的推送服务。

3. Netty 海量推送服务设计要点

作为高性能的 NIO 框架，利用 Netty 开发高效的推送服务技术上是可行的，但是由于推送服务自身的复杂性，想要开发出稳定、高性能的推送服务并非易事，需要在设计阶段针对推送服务的特点进行合理设计。

3.1. 最大句柄数修改

百万长连接接入，首先需要优化的就是 Linux 内核参数，其中 Linux 最大文件句柄数是最重要的调优参数之一，默�单进程打开的最大句柄数是 1024，通过 ulimit -a 可以查看相关参数，示例如下：

```
[root@lilinfeng ~]# ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority      (-e) 0
file size                (blocks, -f) unlimited
pending signals          (-i) 256324
max locked memory        (kbytes, -l) 64
max memory size          (kbytes, -m) unlimited
open files               (-n) 1024
```

.....后续输出省略

当单个推送服务接收到的链接超过上限后，就会报“too many open files”，所有新的客户端接入将失败。

通过 vi /etc/security/limits.conf 添加如下配置参数：修改之后保存，注销当前用户，重新登录，通过 ulimit -a 查看修改的状态是否生效。

```
* soft  nofile  1000000
* hard   nofile  1000000
```

需要指出的是，尽管我们可以将单个进程打开的最大句柄数修改的非常大，但是当句柄数达到一定数量级之后，处理效率将出现明显下降，因此，需要根据服务器的硬件配置和处理能力进行合理设置。如果单个服务器性能不行也可以通过集群的方式实现。

3.2. 当心 CLOSE_WAIT

从事移动推送服务开发的同学可能都有体会，移动无线网络可靠性非常差，经常存在客户端重置连接，网络闪断等。

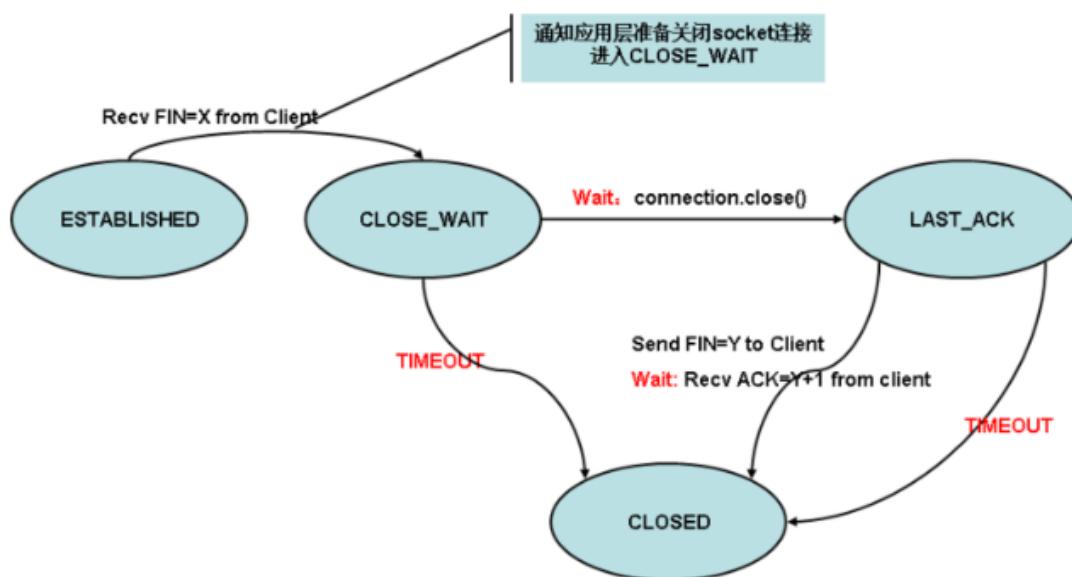
在百万长连接的推送系统中，服务端需要能够正确处理这些网络异常，设计要点如下：

- 1) 客户端的重连间隔需要合理设置，防止连接过于频繁导致的连接失败（例如端口还没有被释放）；
- 2) 客户端重复登陆拒绝机制；

3) 服务端正确处理 I/O 异常和解码异常等，防止句柄泄露。

最后特别需要注意的一点就是 close_wait 过多问题，由于网络不稳定经常会导致客户端断连，如果服务端没有能够及时关闭 socket，就会导致处于 close_wait 状态的链路过多。close_wait 状态的链路并不释放句柄和内存等资源，如果积压过多可能会导致系统句柄耗尽，发生“Too many open files”异常，新的客户端无法接入，涉及创建或者打开句柄的操作都将失败。

下面对 close_wait 状态进行下简单介绍，被动关闭 TCP 连接状态迁移图如下所示：



被动关闭 TCP 连接状态迁移图

close_wait 是被动关闭连接形成的，根据 TCP 状态机，服务器端收到客户端发送的 FIN，TCP 协议栈会自动发送 ACK，链接进入 close_wait 状态。但如果服务器端不执行 socket 的 close() 操作，状态就不能由 close_wait 迁移到 last_ack，则系统中会存在很多 close_wait 状态的连接。通常来说，一个 close_wait 会维持至少 2 个小时的时间（系统默认超时时间的是 7200 秒，也就是 2 小时）。如果服务端程序因某个原因导致系统造成一堆 close_wait 消耗资源，那么通常是等不到释放那一刻，系统就已崩溃。

导致 close_wait 过多的可能原因如下：

- 1) 程序处理 Bug, 导致接收到对方的 fin 之后没有及时关闭 socket, 这可能是 Netty 的 Bug, 也可能是业务层 Bug, 需要具体问题具体分析;
- 2) 关闭 socket 不及时: 例如 I/O 线程被意外阻塞, 或者 I/O 线程执行的用户自定义 Task 比例过高, 导致 I/O 操作处理不及时, 链路不能被及时释放。

下面我们结合 Netty 的原理, 对潜在的故障点进行分析。

设计要点 1: 不要在 Netty 的 I/O 线程上处理业务 (心跳发送和检测除外)。Why? 对于 Java 进程, 线程不能无限增长, 这就意味着 Netty 的 Reactor 线程数必须收敛。Netty 的默认值是 CPU 核数 * 2, 通常情况下, I/O 密集型应用建议线程数尽量设置大些, 但这主要是针对传统同步 I/O 而言, 对于非阻塞 I/O, 线程数并不建议设置太大, 尽管没有最优值, 但是 I/O 线程数经验值是 [CPU 核数 + 1, CPU 核数*2] 之间。

假如单个服务器支撑 100 万个长连接, 服务器内核数为 32, 则单个 I/O 线程处理的链接数 $L = 100/(32 * 2) = 15625$ 。假如每 5S 有一次消息交互 (新消息推送、心跳消息和其它管理消息), 则平均 CAPS = $15625 / 5 = 3125$ 条/秒。这个数值相比于 Netty 的处理性能而言压力并不大, 但是在实际业务处理中, 经常会有一些额外的复杂逻辑处理, 例如性能统计、记录接口日志等, 这些业务操作性能开销也比较大, 如果在 I/O 线程上直接做业务逻辑处理, 可能会阻塞 I/O 线程, 影响对其它链路的读写操作, 这就会导致被动关闭的链路不能及时关闭, 造成 close_wait 堆积。

设计要点 2: 在 I/O 线程上执行自定义 Task 要当心。Netty 的 I/O 处理线程 NioEventLoop 支持两种自定义 Task 的执行:

普通的 Runnable: 通过调用 NioEventLoop 的 execute(Runnable task)方法执行;

定时任务 ScheduledFutureTask: 通过调用 NioEventLoop 的 schedule(Runnable command, long delay, TimeUnit unit)系列接口执行。

为什么 NioEventLoop 要支持用户自定义 Runnable 和 ScheduledFutureTask 的执行, 并不是本文要讨论的重点, 后续会有专题文章进行介绍。本文重点对它们的影响进行分析。

在 NioEventLoop 中执行 Runnable 和 ScheduledFutureTask，意味着允许用户在 NioEventLoop 中执行非 I/O 操作类的业务逻辑，这些业务逻辑通常用消息报文的处理和协议管理相关。它们的执行会抢占 NioEventLoop I/O 读写的 CPU 时间，如果用户自定义 Task 过多，或者单个 Task 执行周期过长，会导致 I/O 读写操作被阻塞，这样也间接导致 close_wait 堆积。

所以，如果用户在代码中使用到了 Runnable 和 ScheduledFutureTask，请合理设置 ioRatio 的比例，通过 NioEventLoop 的 setIoRatio(int ioRatio) 方法可以设置该值，默认值为 50，即 I/O 操作和用户自定义任务的执行时间比为 1: 1。

我的建议是当服务端处理海量客户端长连接的时候，不要在 NioEventLoop 中执行自定义 Task，或者非心跳类的定时任务。

设计要点 3：IdleStateHandler 使用要当心。很多用户会使用 IdleStateHandler 做心跳发送和检测，这种用法值得提倡。相比于自己启定任务发送心跳，这种方式更高效。但是在实际开发中需要注意的是，在心跳的业务逻辑处理中，无论是正常还是异常场景，处理时延要可控，防止时延不可控导致的 NioEventLoop 被意外阻塞。例如，心跳超时或者发生 I/O 异常时，业务调用 Email 发送接口告警，由于 Email 服务端处理超时，导致邮件发送客户端被阻塞，级联引起 IdleStateHandler 的 AllIdleTimeoutTask 任务被阻塞，最终 NioEventLoop 多路复用器上其它的链路读写被阻塞。

对于 ReadTimeoutHandler 和 WriteTimeoutHandler，约束同样存在。

3.3. 合理的心跳周期

百万级的推送服务，意味着会存在百万个长连接，每个长连接都需要靠和 App 之间的心跳来维持链路。合理设置心跳周期是非常重要的工作，推送服务的心跳周期设置需要考虑移动无线网络的特点。

当一台智能手机连上移动网络时，其实并没有真正连接上 Internet，运营商分配给手机的 IP 其实是运营商的内网 IP，手机终端要连接上 Internet 还必须通过运营商的网关进行 IP 地址的转换，这个网关简称为 NAT（NetWork Address Translation），简单来说就是手机终端连接 Internet 其实就是移动内网 IP，端口，外网 IP 之间相互映射。

GGSN（GateWay GPRS Support Note）模块就实现了 NAT 功能，由于大部分的移动无线网络运营商为了减少网关 NAT 映射表的负荷，如果一个链路有一段时间没有通信时就会删除其对应表，造成链路中断，正是这种刻意缩短空闲连接的释放超时，原本是想节省信道资源的作用，没想到让互联网的应用不得以远高于正常频率发送心跳来维护推送的长连接。以中移动的 2.5G 网络为例，大约 5 分钟左右的基带空闲，连接就会被释放。

由于移动无线网络的特点，推送服务的心跳周期并不能设置的太长，否则长连接会被释放，造成频繁的客户端重连，但是也不能设置太短，否则在当前缺乏统一心跳框架的机制下很容易导致信令风暴（例如微信心跳信令风暴问题）。具体的心跳周期并没有统一的标准，180S 也许是个不错的选择，微信为 300S。

在 Netty 中，可以通过在 ChannelPipeline 中增加 IdleStateHandler 的方式实现心跳检测，在构造函数中指定链路空闲时间，然后实现空闲回调接口，实现心跳的发送和检测，代码如下：

```
public void initChannel{@link Channel} channel){  
    channel.pipeline().addLast("idleStateHandler", new {@link  
        IdleStateHandler}(0, 0, 180)); channel.pipeline().addLast("myHandler",  
        new MyHandler());  
}
```

拦截链路空闲事件并处理心跳：

```
public class MyHandler extends {@link ChannelHandlerAdapter}  
{    {@code @Override}  
    public void userEventTriggered{@link ChannelHandlerContext} ctx, {@link Object} evt) throws {@link Exception} {  
        if (evt instanceof {@link IdleStateEvent}) {  
            //心跳处理  
        }  
    }  
}
```

3.4. 合理设置接收和发送缓冲区容量

对于长链接，每个链路都需要维护自己的消息接收和发送缓冲区，JDK 原生的 NIO 类库使用的是 `java.nio.ByteBuffer`，它实际是一个长度固定的 Byte 数组，我们都知道数组无法动态扩容，`ByteBuffer` 也有这个限制，相关代码如下：

```
public abstract class ByteBuffer
    extends Buffer
    implements Comparable
{
    final byte[] hb; // Non-null only for heap buffers
    final int offset;
    boolean isReadOnly;
```

容量无法动态扩展会带来一些麻烦，例如由于无法预测每条消息报文的长度，可能需要预分配一个比较大的 `ByteBuffer`，这通常也没有问题。但是在海量推送服务系统中，这会给服务端带来沉重的内存负担。假设单条推送消息最大上限为 10K，消息平均大小为 5K，为了满足 10K 消息的处理，`ByteBuffer` 的容量被设置为 10K，这样每条链路实际上多消耗了 5K 内存，如果长链接链路数为 100 万，每个链路都独立持有 `ByteBuffer` 接收缓冲区，则额外损耗的总内存 $Total(M) = 1000000 * 5K = 4882M$ 。内存消耗过大，不仅仅增加了硬件成本，而且大内存容易导致长时间的 Full GC，对系统稳定性会造成比较大的冲击。

实际上，最灵活的处理方式就是能够动态调整内存，即接收缓冲区可以根据以往接收的消息进行计算，动态调整内存，利用 CPU 资源来换内存资源，具体的策略如下：

`ByteBuffer` 支持容量的扩展和收缩，可以按需灵活调整，以节约内存；

接收消息的时候，可以按照指定的算法对之前接收的消息大小进行分析，并预测未来的消息大小，按照预测值灵活调整缓冲区容量，以做到最小的资源损耗满足程序正常功能。

幸运的是，Netty 提供的 `ByteBuf` 支持容量动态调整，对于接收缓冲区的内存分配器，Netty 提供了两种：

`FixedRecvByteBufAllocator`：固定长度的接收缓冲区分配器，由它分配的 `ByteBuf` 长度都是固定大小的，并不会根据实际数据报的大小动态收缩。但是，如果容量不足，支持动态扩展。动态扩展是 Netty `ByteBuf` 的一项基本功能，与 `ByteBuf` 分配器的实现没有关系；

AdaptiveRecvByteBufAllocator: 容量动态调整的接收缓冲区分配器，它会根据之前 Channel 接收到的数据报大小进行计算，如果连续填充满接收缓冲区的可写空间，则动态扩展容量。如果连续 2 次接收到的数据报都小于指定值，则收缩当前的容量，以节约内存。

相对于 FixedRecvByteBufAllocator，使用 AdaptiveRecvByteBufAllocator 更为合理，可以在创建客户端或者服务端的时候指定 RecvByteBufAllocator，代码如下：

```
Bootstrap b = new Bootstrap();
    b.group(group)
        .channel(NioSocketChannel.class)
        .option(ChannelOption.TCP_NODELAY, true)
        .option(ChannelOption.RCVBUF_ALLOCATOR, AdaptiveRecvByteBufAllocator.DEFAULT)
```

如果默认没有设置，则使用 AdaptiveRecvByteBufAllocator。

另外值得注意的是，无论是接收缓冲区还是发送缓冲区，缓冲区的大小建议设置为消息的平均大小，不要设置成最大消息的上限，这会导致额外的内存浪费。通过如下方式可以设置接收缓冲区的初始大小：

```
/**
 * Creates a new predictor with the specified parameters.
 *
 * @param minimum
 *          the inclusive lower bound of the expected buffer size
 * @param initial
 *          the initial buffer size when no feed back was received
 * @param maximum
 *          the inclusive upper bound of the expected buffer size
 */
public AdaptiveRecvByteBufAllocator(int minimum, int initial, int maximum)
```

对于消息发送，通常需要用户自己构造 ByteBuf 并编码，例如通过如下工具类创建消息发送缓冲区：



构造指定容量的缓冲区

3.5. 内存池

推送服务器承载了海量的长链接，每个长链接实际就是一个会话。如果每个会话都持有心跳数据、接收缓冲区、指令集等数据结构，而且这些实例随着消息的处理朝生夕灭，这就会给服务器带来沉重的 GC 压力，同时消耗大量的内存。

最有效的解决策略就是使用内存池，每个 NioEventLoop 线程处理 N 个链路，在线程内部，链路的处理时串行的。假如 A 链路首先被处理，它会创建接收缓冲区等对象，待解码完成之后，构造的 POJO 对象被封装成 Task 后投递到后台的线程池中执行，然后接收缓冲区会被释放，每条消息的接收和处理都会重复接收缓冲区的创建和释放。如果使用内存池，则当 A 链路接收到新的数据报之后，从 NioEventLoop 的内存池中申请空闲的 ByteBuf，解码完成之后，调用 release 将 ByteBuf 释放到内存池中，供后续 B 链路继续使用。

使用内存池优化之后，单个 NioEventLoop 的 ByteBuf 申请和 GC 次数从原来的 $N = 1000000/64 = 15625$ 次减少为最少 0 次（假设每次申请都有可用的内存）。

下面我们以推特使用 Netty4 的 PooledByteBufAllocator 进行 GC 优化作为案例，对内存池的效果进行评估，结果如下：

垃圾生成速度是原来的 1/5，而垃圾清理速度快了 5 倍。使用新的内存池机制，几乎可以把网络带宽压满。

Netty4 之前的版本问题如下：每当收到新信息或者用户发送信息到远程端，Netty 3 均会创建一个新的堆缓冲区。这意味着，对应每一个新的缓冲区，都会有一个 new byte[capacity]。这些缓冲区会导致 GC 压力，并消耗内存带宽。为了安全起见，新的字节数组分配时会用零填充，这会消耗内存带宽。然而，用零填充的数组很可能会再次用实际的数据填充，这又会消耗同样的内存带宽。如果 Java 虚拟机（JVM）提供了创建新字节数组而又无需用零填充的方式，那么我们本来就可以将内存带宽消耗减少 50%，但是目前没有那样一种方式。

在 Netty 4 中实现了一个新的 ByteBuf 内存池，它是一个纯 Java 版本的 [jemalloc](#)（Facebook 也在用）。现在，Netty 不会再因为用零填充缓冲区而浪费内存带宽了。不过，由于它不依赖于 GC，开发人员需要小心内存泄漏。如果忘记在处理程序中释放缓冲区，那么内存使用率会无限地增长。

Netty 默认不使用内存池，需要在创建客户端或者服务端的时候进行指定，代码如下：

```
Bootstrap b = new Bootstrap();
    b.group(group)
        .channel(NioSocketChannel.class)
        .option(ChannelOption.TCP_NODELAY, true)
        .option(ChannelOption.ALLOCATOR, PooledByteBufAllocator.DEFAULT)
```

使用内存池之后，内存的申请和释放必须成对出现，即 retain() 和 release() 要成对出现，否则会导致内存泄露。

值得注意的是，如果使用内存池，完成 ByteBuf 的解码工作之后必须显式的调用 ReferenceCountUtil.release(msg) 对接收缓冲区 ByteBuf 进行内存释放，否则它会被认为仍然在使用中，这样会导致内存泄露。

3.6. 当心“日志隐形杀手”

通常情况下，大家都知道不能在 Netty 的 I/O 线程上做执行时间不可控的操作，例如访问数据库、发送 Email 等。但是有个常用但是非常危险的操作却容易被忽略，那便是记录日志。

通常，在生产环境中，需要实时打印接口日志，其它日志处于 ERROR 级别，当推送服务发生 I/O 异常之后，会记录异常日志。如果当前磁盘的 WIO 比较高，可能会发生写日志文件操作被同步阻塞，阻塞时间无

法预测。这就会导致 Netty 的 NioEventLoop 线程被阻塞，Socket 链路无法被及时关闭、其它的链路也无法进行读写操作等。

以最常用的 log4j 为例，尽管它支持异步写日志（AsyncAppender），但是当日志队列满之后，它会同步阻塞业务线程，直到日志队列有空闲位置可用，相关代码如下：

```
synchronized (this.buffer) {
    while (true) {
        int previousSize = this.buffer.size();
        if (previousSize < this.bufferSize) {
            this.buffer.add(event);
            if (previousSize != 0) break;
            this.buffer.notifyAll(); break;
        }
        boolean discard = true;
        if ((this.blocking) && (!Thread.interrupted()) &&
(Thread.currentThread() != this.dispatcher)) //判断是业务线程
        {
            Try
            {
                this.buffer.wait(); //阻塞业务线程
                discard = false;
            }
            catch (InterruptedException e)
            {
                Thread.currentThread().interrupt();
            }
        }
    }
}
```

类似这类 BUG 具有极强的隐蔽性，往往 WIO 高的时间持续非常短，或者是偶现的，在测试环境中很难模拟此类故障，问题定位难度非常大。这就要求读者在平时写代码的时候一定要当心，注意那些隐性地雷。

3.7. TCP 参数优化

常用的 TCP 参数，例如 TCP 层面的接收和发送缓冲区大小设置，在 Netty 中分别对应 ChannelOption 的 SO_SNDBUF 和 SO_RCVBUF，需要根据推送消息的大小，合理设置，对于海量长连接，通常 32K 是个不错的选择。

另外一个比较常用的优化手段就是软中断，如图所示：如果所有的软中断都运行在 CPU0 相应网卡的硬件中断上，那么始终都是 cpu0 在处理软中断，而此时其它 CPU 资源就被浪费了，因为无法并行的执行多个软中断。

	CPU0	CPU1	CPU2	CPU3	CPU4	CPU5	CPU6	CPU7	
255:	2278111556	0	0	0	0	0	0	0	Dynamic-irq timer0
257:	1593008	0	0	0	0	0	0	0	Dynamic-irq resched0
258:	30	0	0	0	0	0	0	0	Dynamic-irq callfunc0
259:	680	0	0	0	0	0	0	0	Dynamic-irq xenbus
260:	0	776074	0	0	0	0	0	0	Dynamic-irq resched1
261:	0	90	0	0	0	0	0	0	Dynamic-irq callfunc1
262:	0	460767536	0	0	0	0	0	0	Dynamic-irq timer1
263:	0	0	784517	0	0	0	0	0	Dynamic-irq resched2
264:	0	0	92	0	0	0	0	0	Dynamic-irq callfunc2
265:	0	0	610961913	0	0	0	0	0	Dynamic-irq timer2
266:	0	0	0	945497	0	0	0	0	Dynamic-irq resched3
267:	0	0	0	92	0	0	0	0	Dynamic-irq callfunc3
268:	0	0	0	705281800	0	0	0	0	Dynamic-irq timer3
269:	0	0	0	0	729973	0	0	0	Dynamic-irq resched4
270:	0	0	0	0	92	0	0	0	Dynamic-irq callfunc4
271:	0	0	0	0	6121593865	0	0	0	Dynamic-irq timer4
272:	0	0	0	0	0	747793	0	0	Dynamic-irq resched5
273:	0	0	0	0	0	91	0	0	Dynamic-irq callfunc5
274:	0	0	0	0	0	739413487	0	0	Dynamic-irq timer5
275:	0	0	0	0	0	0	705970	0	Dynamic-irq resched6
276:	0	0	0	0	0	0	90	0	Dynamic-irq callfunc6
277:	0	0	0	0	0	0	543901390	0	Dynamic-irq timer6
278:	0	0	0	0	0	0	0	707045	Dynamic-irq resched7
279:	0	0	0	0	0	0	0	67	Dynamic-irq callfunc7
280:	0	0	0	0	0	0	0	499288362	Dynamic-irq timer7
281:	35	0	0	0	0	0	0	0	Dynamic-irq xencons
282:	3138	0	0	0	0	0	0	0	Dynamic-irq xenfb
283:	0	0	0	0	0	0	0	0	Dynamic-irq xenkbd
284:	2601367	606	0	2611	0	0	0	0	Dynamic-irq blkif
285:	112133	4153	0	0	60	0	0	0	Dynamic-irq blkif
286:	19	0	0	0	0	0	0	0	Dynamic-irq blkif
287:	14259441	0	0	0	0	0	0	0	Dynamic-irq eth0

中断信息

大于等于 2.6.35 版本的 Linux kernel 内核，开启 RPS，网络通信性能提升 20%之上。RPS 的基本原理：根据数据包的源地址，目的地址以及目的和源端口，计算出一个 hash 值，然后根据这个 hash 值来选择软中断运行的 cpu。从上层来看，也就是说将每个连接和 cpu 绑定，并通过这个 hash 值，来均衡软中断运行在多个 cpu 上，从而提升通信性能。

3.8. JVM 参数

最重要的参数调整有两个：

- 1) -Xmx:JVM 最大内存需要根据内存模型进行计算并得出相对合理的值；
- 2) GC 相关的参数：例如新生代和老生代、永久代的比例，GC 的策略，新生代各区的比例等，需要根据具体的场景进行设置和测试，并不断的优化，尽量将 Full GC 的频率降到最低。

4. 作者简介

李林峰，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 6 年 NIO 设计和开发经验，精通 Netty、Mina 等 NIO 框架。Netty 中国社区创始人，《Netty 权威指南》作者。

联系方式：新浪微博 Nettying 微信：Nettying

Java 多线程编程模式实战指南： Immutable Object 模式

作者 黄文海

多线程共享变量的情况下，为了保证数据一致性，往往需要对这些变量的访问进行加锁。而锁本身又会带来一些问题和开销。Immutable Object 模式使得我们可以在不使用锁的情况下，既保证共享变量访问的线程安全，又能避免引入锁可能带来的问题和开销。

Immutable Object 模式简介

多线程环境中，一个对象常常会被多个线程共享。这种情况下，如果存在多个线程并发地修改该对象的状态或者一个线程读取该对象的状态而另外一个线程试图修改该对象的状态，我们不得不做一些同步访问控制以保证数据一致性。而这些同步访问控制，如显式锁和 CAS 操作，会带来额外的开销和问题，如上下文切换、等待时间和 ABA 问题等。Immutable Object 模式的意图是通过使用对外可见的状态不可变的对象（即 Immutable Object），使得被共享对象“天生”具有线程安全性，而无需额外的同步访问控制。从而既保证了数据一致性，又避免了同步访问控制所产生的额外开销和问题，也简化了编程。

所谓状态不可变的对象，即对象一经创建其对外可见的状态就保持不变，例如 Java 中的 String 和 Integer。这点固然容易理解，但这还不足以指导我们在实际工作中运用 Immutable Object 模式。下面我们看一个典型应用场景，这不仅有助于我们理解它，也有助于在实际的环境中运用它。

一个车辆管理系统要对车辆的位置信息进行跟踪，我们可以对车辆的位置信息建立如清单 1 所示的模型。

清单 1. 状态可变的位置信息模型（非线程安全）

```
public class Location {
```

```
private double x;
private double y;

public Location(double x, double y) {
    this.x = x;
    this.y = y;
}

public double getX() {
    return x;
}

public double getY() {
    return y;
}

public void setXY(double x, double y) {
    this.x = x;
    this.y = y;
}
}
```

当系统接收到新的车辆坐标数据时，需要调用 Location 的 setXY 方法来更新位置信息。显然，清单 1 中 setXY 是非线程安全的，因为对坐标数据 x 和 y 的写操作不是一个原子操作。setXY 被调用时，如果在 x 写入完毕，而 y 开始写之前有其它线程来读取位置信息，则该线程可能读到一个被追踪车辆根本不曾经过的位置。为了使 setXY 方法具备线程安全性，我们需要借助锁进行访问控制。虽然被追踪车辆的位置信息总是在变化，但是我们也可以将位置信息建模为状态不可变的对象，如清单 2 所示。

清单 2. 状态不可变的位置信息模型

```
public final class Location {
    public final double x;
    public final double y;
    public Location(double x, double y) {
        this.x = x;
        this.y = y;
    }
}
```

使用状态不可变的位置信息模型时，如果车辆的位置发生变动，则更新车辆的位置信息是通过替换整个表示位置信息的对象（即 Location 实例）来实现的。如清单 3 所示。

清单 3. 在使用不可变对象的情况下更新车辆的位置信息

```
public class VehicleTracker {  
    private Map<String, Location> locMap  
        = new ConcurrentHashMap();  
    public void updateLocation(String vehicleId, Location  
newLocation){  
        locMap.put(vehicleId, newLocation);  
    }  
}
```

因此，所谓状态不可变的对象并非指被建模的现实世界实体的状态不可变，而是我们在建模的时候的一种决策：现实世界实体的状态总是在变化的，但我们可以用状态不可变的对象来对这些实体进行建模。

Immutable Object 模式的架构

Immutable Object 模式的主要参与者有以下几种。其类图如图 1 所示。

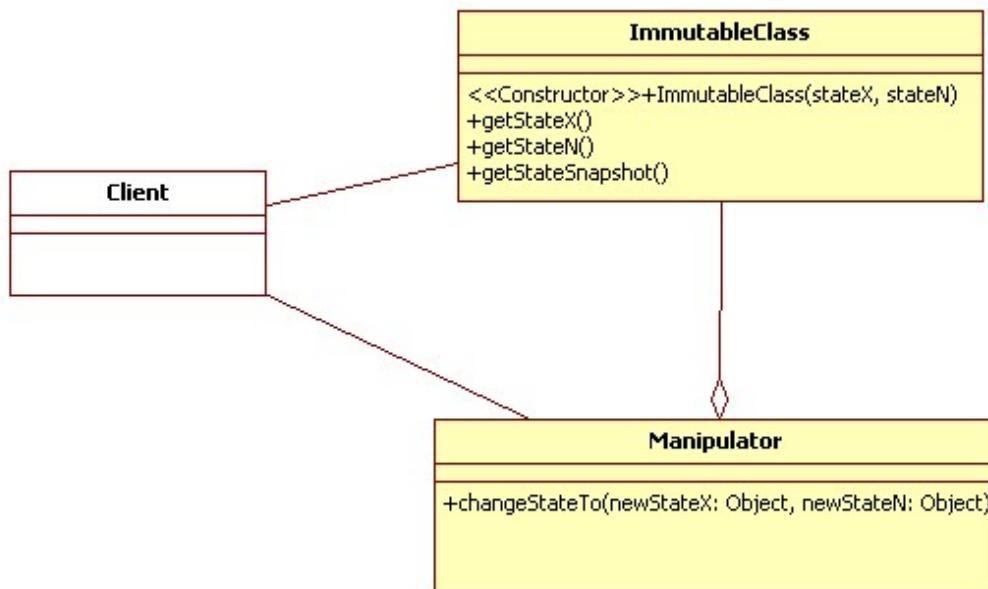


图 1. Immutable Object 模式的类图

ImmutableClass: 负责存储一组不可变状态的类。该类不对外暴露任何可以修改其状态的方法，其主要方法及职责如下：

getStateX, getStateN: 这些 getter 方法返回该类所维护的状态相关变量的值。这些变量在对象实例化时通过其构造器的参数获得值。

getStateSnapshot: 返回该类维护的一组状态的快照。

Manipulator: 负责维护 ImmutableClass 所建模的现实世界实体状态的变更。当相应的现实世界实体状态变更时，该类负责生成新的 ImmutableClass 的实例，以反映新的状态。

changeStateTo: 根据新的状态值生成新的 ImmutableClass 的实例。

不可变对象的使用主要包括以下几种类型：

获取单个状态的值：调用不可变对象的相关 getter 方法即可实现。

获取一组状态的快照：不可变对象可以提供一个 getter 方法，该方法需要对其返回值做防御性拷贝或者返回一个只读的对象，以避免其状态对外泄露而被改变。

生成新的不可变对象实例：当被建模对象的状态发生变化的时候，创建新的不可变对象实例来反映这种变化。

Immutable Object 模式的典型交互场景如图 2 所示。

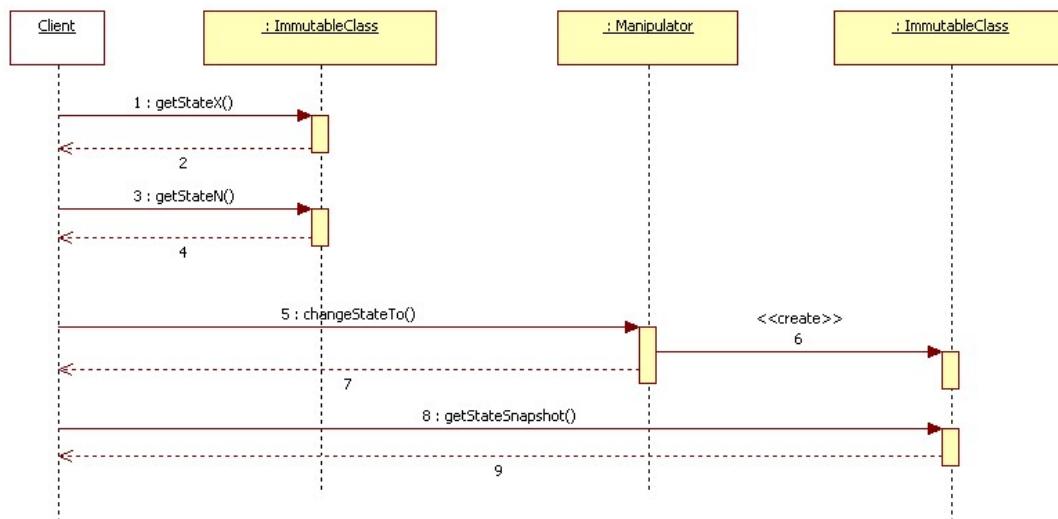


图 2. Immutable Object 模式的序列图

- 1~4、客户端代码获取 ImmutableClass 的各个状态值。
- 5、客户端代码调用 Manipulator 的 changeStateTo 方法来更新应用的状态。
- 6、Manipulator 创建新的 ImmutableClass 实例以反映应用的新状态。
- 7~9、客户端代码获取新的 ImmutableClass 实例的状态快照。

一个严格意义上不可变对象要满足以下所有条件：

- 1) **类本身使用 final 修饰：**防止其子类改变其定义的行为；
- 2) **所有字段都是用 final 修饰的：**使用 final 修饰不仅仅是从语义上说明被修饰字段的引用不可改变。更重要的是这个语义在多线程环境下由 JMM（Java Memory Model）保证了被修饰字段的所引用对象的初始化安全，即 final 修饰的字段在其它线程可见时，它必定是初始化完成的。相反，非 final 修饰的字段由于缺少这种保证，可能导致一个线程“看到”一个字段的时候，它还未被初始化完成，从而可能导致一些不可预料的结果；
- 3) **在对象的创建过程中， this 关键字没有泄露给其它类：**防止其它类（如该类的匿名内部类）在对象创建过程中修改其状态；
- 4) **任何字段，若其引用了其它状态可变的对象（如集合、数组等），则这些字段必须是 private 修饰的，并且这些字段值不能对外暴露。若有相关方法要返回这些字段值，应该进行防御性拷贝（Defensive Copy）。**

Immutable Object 模式实战案例

某彩信网关系统在处理由增值业务提供商（VASP, Value-Added Service Provider）下发给手机终端用户的彩信消息时，需要根据彩信接收方号码的前缀（如 1381234）选择对应的彩信中心（MMSC, Multimedia Messaging Service Center），然后转发消息给选中的彩信中心，由其负责对接电信网络将彩信消息下发给手机终端用户。彩信中心相对于彩信网关系统而言，它是一个独立的部件，二者通过网络进行交互。这个选择彩信中心的过程，我们称之为路由（Routing）。而手机号前缀和彩信中心的这种对应关系，被称为路由表。路由表在软件运维过程

中可能发生变化。例如，业务扩容带来的新增彩信中心、为某个号码前缀指定新的彩信中心等。虽然路由表在该系统中是由多线程共享的数据，但是这些数据的变化频率并不高。因此，即使是为了保证线程安全，我们也不希望对这些数据的访问进行加锁等并发访问控制，以免产生不必要的开销和问题。这时，**Immutable Object** 模式就派上用场了。

维护路由表可以被建模为一个不可变对象，如清单 4 所示。

清单 4. 使用不可变对象维护路由表

```
public final class MMSCRouter {  
    // 用 volatile 修饰，保证多线程环境下该变量的可见性  
    private static volatile MMSCRouter instance = new MMSCRouter();  
    //维护手机号码前缀到彩信中心之间的映射关系  
    private final Map<String, MMSCIInfo> routeMap;  
  
    public MMSCRouter() {  
        // 将数据库表中的数据加载到内存，存为 Map  
        this.routeMap = MMSCRouter.retrieveRouteMapFromDB();  
    }  
    private static Map<String, MMSCIInfo> retrieveRouteMapFromDB() {  
        Map<String, MMSCIInfo> map = new HashMap<String,  
MMSCIInfo>();  
        // 省略其它代码  
        return map;  
    }  
    public static MMSCRouter getInstance() {  
  
        return instance;  
    }  
    /**  
     * 根据手机号码前缀获取对应的彩信中心信息  
     *  
     * @param msisdnPrefix  
     *      手机号码前缀  
     * @return 彩信中心信息  
     */  
    public MMSCIInfo getMMSC(String msisdnPrefix) {  
        return routeMap.get(msisdnPrefix);  
    }  
}
```

```
 }、  
/**  
 * 将当前 MMSCRouter 的实例更新为指定的新实例  
 *  
 * @param newInstance  
 *         新的 MMSCRouter 实例  
 */  
public static void setInstance(MMSCRouter newInstance) {  
    instance = newInstance;  
}  
  
private static Map<String, MMSCInfo> deepCopy(Map<String,  
MMSCInfo> m) {  
    Map<String, MMSCInfo> result = new HashMap<String,  
MMSCInfo>();  
    for (String key : m.keySet()) {  
        result.put(key, new MMSCInfo(m.get(key)));  
    }  
    return result;  
}  
public Map<String, MMSCInfo> getRouteMap() {  
    //做防御性拷贝  
    return Collections.unmodifiableMap(deepCopy(routeMap));  
}  
}
```

而彩信中心的相关数据，如彩信中心设备编号、URL、支持的最大附件尺寸也被建模为一个不可变对象。如清单 5 所示。

清单 5. 使用不可变对象表示彩信中心信息

```
public final class MMSCInfo {  
    /**  
     * 设备编号  
     */  
    private final String deviceID;  
    /**  
     * 彩信中心 URL  
     */  
  
    private final String url;
```

```
/*
 * 该彩信中心允许的最大附件大小
 */
private final int maxAttachmentSizeInBytes;
public MMSCInfo(String deviceID, String url, int
maxAttachmentSizeInBytes) {
    this.deviceID = deviceID;
    this.url = url;
    this.maxAttachmentSizeInBytes = maxAttachmentSizeInBytes;
}
public MMSCInfo(MMSCInfo prototype) {
    this.deviceID = prototype.deviceID;
    this.url = prototype.url;
    this.maxAttachmentSizeInBytes =
prototype.maxAttachmentSizeInBytes;
}
public String getDeviceID() {
    return deviceID;
}
public String getUrl() {
    return url;
}
public int getMaxAttachmentSizeInBytes() {
    return maxAttachmentSizeInBytes;
}
}
```

彩信中心信息变更的频率也同样不高。因此，当彩信网关系统通过网络（Socket 连接）被通知到这种彩信中心信息本身或者路由表变更时，网关系统会重新生成新的 MMSCInfo 和 MMSCRouter 来反映这种变更。如清单 6 所示。

清单 6. 处理彩信中心、路由表的变更

```
/*
 * 与运维中心（Operation and Maintenance Center）对接的类
 *
public class OMCAgent extends Thread{
    @Override
    public void run() {
        boolean isTableModificationMsg=false;
        String updatedTableName=null;
        while(true){
```

```
//省略其它代码
/*
 * 从与 OMC 连接的 Socket 中读取消息并进行解析,
 * 解析到数据表更新消息后,重置 MMSCRouter 实例。
 */
if(isTableModificationMsg){
    if("MMSCInfo".equals(updatedTableName)){
        MMSCRouter.setInstance(new
MMSCRouter());
    }
}
//省略其它代码
}
}
}
```

上述代码会调用 MMSCRouter 的 setInstance 方法来替换 MMSCRouter 的实例为新创建的实例。而新创建的 MMSCRouter 实例通过其构造器会生成多个新的 MMSCInfo 的实例。

本案例中， MMSCInfo 是一个严格意义上的不可变对象。虽然 MMSCRouter 对外提供了 setInstance 方法用于改变其静态字段 instance 的值，但它仍然可视作一个等效的不可变对象。这是因为， setInstance 方法仅仅是改变 instance 变量指向的对象，而 instance 变量采用 volatile 修饰保证了其在多线程之间的内存可见性，这意味着 setInstance 对 instance 变量的改变无需加锁也能保证线程安全。而其它代码在调用 MMSCRouter 的相关方法获取路由信息时也无需加锁。

从图 1 的类图上看， OMCAgent 类（见清单 6）是一个 Manipulator 参与者实例，而 MMSCInfo、MMSCRouter 是一个 ImmutableClass 参与者实例。通过使用不可变对象，我们既可以应对路由表、彩信中心这些不是非常频繁的变更，又可以使系统中使用路由表的代码免于并发访问控制的开销和问题。

Immutable Object 模式的评价与实现考量

不可变对象具有天生的线程安全性，多个线程共享一个不可变对象的时候无需使用额外的并发访问控制，这使得我们可以避免显式锁（Explicit Lock）等并发访问控制的开销和问题，简化了多线程编程。

Immutable Object 模式特别适用于以下场景。

被建模对象的状态变化不频繁：正如本文案例所展示的，这种场景下可以设置一个专门的线程（Manipulator 参与者所在的线程）用于在被建模对象状态变化时创建新的不可变对象。而其它线程则只是读取不可变对象的状态。此场景下的一个小技巧是 Manipulator 对不可变对象的引用采用 volatile 关键字修饰，既可以避免使用显式锁（如 synchronized），又可以保证多线程间的内存可见性。

同时对一组相关的数据进行写操作，因此需要保证原子性：此场景为了保证操作的原子性，通常的做法是使用显式锁。但若采用 Immutable Object 模式，将这一组相关的数据“组合”成一个不可变对象，则对这一组数据的操作就可以无需加显式锁也能保证原子性，既简化了编程，又提高了代码运行效率。本文开头所举的车辆位置跟踪的例子正是这种场景。

使用某个对象作为安全的 HashMap 的 Key：我们知道，一个对象作为 HashMap 的 Key 被“放入”HashMap 之后，若该对象状态变化导致了其 Hash Code 的变化，则会导致后面在用同样的对象作为 Key 去 get 的时候无法获取关联的值，尽管该 HashMap 中的确存在以该对象为 Key 的条目。相反，由于不可变对象的状态不变，因此其 Hash Code 也不变。这使得不可变对象非常适于用作 HashMap 的 Key。

Immutable Object 模式实现时需要注意以下几个问题。

被建模对象的状态变更比较频繁：此时也不见得不能使用 Immutable Object 模式。只是这意味着频繁创建新的不可变对象，因此会增加 GC（Garbage Collection）的负担和 CPU 消耗，我们需要综合考虑：被建模对象的规模、代码目标运行环境的 JVM 内存分配情况、系统对吞吐率和响应性的要求。若这几个方面因素综合考虑都能满足要求，那么使用不可变对象建模也未尝不可。

使用等效或者近似的不可变对象：有时创建严格意义上的不可变对象比较难，但是尽量向严格意义上的不可变对象靠拢也有利于发挥不可变对象的好处。

防御性拷贝：如果不可变对象本身包含一些状态需要对外暴露，而相应的字段本身又是可变的（如 HashMap），那么在返回这些字段的方法还是需要做防御性拷贝，以避免外部代码修改了其内部状态。正如清单 4 的代码中的 getRouteMap 方法所展示的那样。

总结

本文介绍了 Immutable Object 模式的意图及架构。并结合笔者工作经历提供了一个实际的案例用于展示使用该模式的典型场景，在此基础上对该模式进行了评价并分享在实际运用该模式时需要注意的事项。

参考资源

本文的源代码在线阅读：

<https://github.com/Viscent/JavaConcurrencyPattern/>

Brian Göetz,*Java theory and practice: To mutate or not to mutate?*:

<http://www.ibm.com/developerworks/java/library/j-jtp02183/index.html>

Brian Göetz et al.,*Java Concurrency In Practice*

Mark Grand,*Patterns in Java*, Volume 1, 2nd Edition

Java Language Specification, 17.5. final Field Semantics:

<http://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html>

作者简介

黄文海，有多年敏捷项目管理经验和丰富的技术指导经验。关注敏捷开发、Java 多线程编程和 Web 开发。在 InfoQ 中文站和 IBM DeveloperWorks 上发表过多篇文章。其博客：<http://viscent.iteye.com>。

技术团队的情绪与效率

作者 申玉宝

引：为什么工程师的效率有那么明显的波峰波谷？为什么会有负面情绪？负面情绪与工作效率有什么关系？团队 Leader 应该怎样保证整体的效率输出与大家的成长？为什么醉心于技术的同学做项目总是虎头蛇尾？

对工程师来说经常会有明显的效率差异，有时一天能搞定好几个模块，顺带加了好几个新的技能点，而有时一个简单的功能投入了两三天还和之前没什么区别。虽然任务并不复杂，但忍不住会刷会微博，聊会 QQ，即使硬着头皮去做，往往效率也不如意，甚至引入一些新的低级 Bug。这个差异与技能水平和工作态度无关，在绝大多数工程师身上都会看到。

效率的影响因素非常多，首先是焦虑。

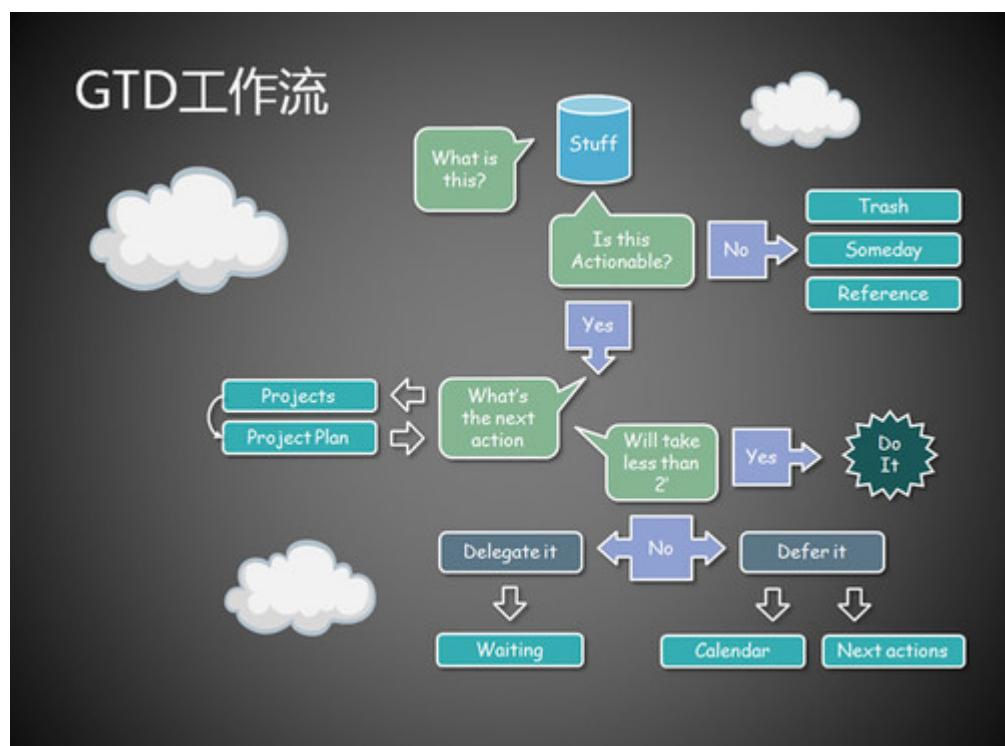
1. 焦虑，执行力崩溃，GTD

当任务单一时大家的效率往往很高，例如『今天下班前只提供一个用户获取接口就行，传入城市编码，分页返回用户』，这个对绝大多数同学没什么心理负担。但现实情景不会这么简单，尤其是在创业型公司，每天会有各种任务，可能运营一会会要一份数据，产品一会报一个 Bug，或者老大又提了几个新的优化点，这些任务单个来看工作量不大，但是持续而无序的任务到达一个工程师身上时，完全可以摧毁他一天的效率和心情。尤其是研发需要注意力集中，频繁的任务切换会浪费大量的时间和精力。

在 GTD (Get Things Done) 中对此有阐述『压力不是来自于任务本身，而是任务在大脑中的堵塞，带来的焦虑和心理的抵触』。当一件任务还没有完成时，持续到来的新任务会带来很大的心理压力，意志不够强大时，很容易导致执行力崩溃，进入一种任务怎么做都做不完的绝望状态。

知道原因了，自然也有解决方案，GTD 提供了一套很可行的执行方案。简化后如下。

把任务放在『待归类』『今日待办』『日程』『等待』几个盒子中收集：每次收到新任务先做一个判断，如果这个任务 5min 可以搞定的话直接干掉，否则都放在『待归类』盒子里。整理：每天开始的时候从『待归类』盒子中开始过滤任务，挑出来今天需要做的 3 件事，放进『今日待办』。如果今天不需要做再根据有没有明确的执行时间，放入日程或者等待盒子里。执行：只盯着『今日待办』即可，再有新任务执行 收集步骤。回顾：定期整理自己的『日程』『等待』盒子。完整的流程图：



这套解决方案能将杂乱地任务明确下来，一定程度上减轻心理压力。

Tools：符合 GTD 的时间管理工具很多，Doit.im 是其中的佼佼者，全平台覆盖，强烈推荐。OmniFocus 则是功能最强大的，支持无限级目录等功能，不过只支持 Mac/iPhone/iPad，且价格不菲。也可以使用印象笔记/OneNote 来自己规划管理，这样相对灵活。

Doit.im:



上面说到的是在任务压力面前个人可以做什么，那作为公司/项目经理/产品经理，也需要为避免『执行力崩溃』做一些事情，那就是保持开发的节奏。

2. 节奏，情绪的体力值

第一次听到『开发的节奏』是在微博的 Scrum 项目流程培训上，这个概念解释了以前大学时我们学生外包团队遇到的诸多问题。简而言已，可以给每个人的情绪量化出一个体力值）。每个开发任务/每个会议/每次报告 都会消耗这个体力值，当体力透支时，后面可能会需要几天不等的时候来恢复体力（我们说的恢复干劲也是这个东西），当透支次数过多时，可能会引发更恶劣的情绪问题。

所以一个健康的团队需要维持开发的节奏，具体操作可以是 每 1-2 周为一个周期，进行大的项目规划，研发任务占用时间最好不高于 80%，之后每个人能有休息/自我充电的时间，在下个周期开始时，团队又能进入满体力值的状态。

具体到我现在的团队，我们以一周为一个单位，每周一产品经理提完本周的需求，我们进行分工消化后，存进需求系统。这周的其他时间内，产品应最大量减少对开发的干扰，下周一的时候对上周的任务进行回顾和总结。这套方案起到了一定的效果，团队成员没有明显的疲惫感，每周能自由支配一些时间（任务能早早完成的话，自由适配时间更多）。

Tools：团队的需求管理系统 我们先后试过 Onenote 多人协作 / Team ambition / Team.oschina / c 禅道，但普遍不理想，或者功能太复杂，或者无法同时集成 Bug 系统，目前采用的是开源的 Cynthia，Cynthia 也是我们团队的 Bug 管理系统。具体工具的选择有时间单拉一篇 Blog 来讲。

Cynthia:

The screenshot shows a web-based application interface for managing bugs. The top navigation bar includes 'Cynthia' logo, search bar, and various menu items like '导出', '发送', '更多操作', '标记为', '配置显示', '设为默认', and '刷新'. A sidebar on the left lists categories: '已处理(未关闭)(3)', '已处理(已关闭)(10)', '待处理(18)', '待处理(3)', and '查看更多...'. Below this is a section titled '我的标签' with color-coded boxes for '重要紧急' (blue), '重要不紧急' (red), and '不重要不紧急' (green). The main content area displays a table of bugs with columns: '序号', '编号', '标题', '状态', '创建人', '创建时间', and '报障人'. There are 18 items listed, each with a yellow progress bar indicating its status. The first three items are from the '待处理(18)' category.

序号	编号	标题	状态	创建人	创建时间	报障人
1	744888	██████████	待处理	申玉宝	2014-11-26 10:12:32	周总
2	744886	██████████	待处理	申玉宝	2014-11-26 10:10:36	周总
3	744785	██████████	待处理	申玉宝	2014-11-13 09:41:47	周总
4	744880	██████████	任务创建	申玉宝	2014-11-26 09:44:36	刘峰
5	744887	██████████	任务创建	申玉宝	2014-11-26 10:11:56	周总
6	744885	██████████	任务创建	申玉宝	2014-11-26 10:08:01	周总
7	744883	██████████	任务创建	申玉宝	2014-11-26 10:00:46	刘峰
8	744881	██████████	任务创建	申玉宝	2014-11-26 09:45:35	刘峰
9	744879	██████████	任务创建	申玉宝	2014-11-26 09:43:24	陈志利
10	744878	██████████	任务创建	申玉宝	2014-11-26 09:35:09	李华强
11	744877	██████████	任务创建	申玉宝	2014-11-26 09:34:37	周总
12	744873	██████████	任务创建	申玉宝	2014-11-19 09:33:56	周总
13	744872	██████████	任务创建	申玉宝	2014-11-19 09:31:48	周总

3. 情绪

影响效率的另一个问题是情绪，情绪问题危害很大，最直接的在于：

- 1) 情绪很容易泛化：单一诱因导致的问题会影响各个方面：工作积极性，工作效率，工作质量，等等；
- 2) 情绪很容易传染：小圈子内，情绪很容易传染（QQ 群功不可没）；
- 3) 情绪不好消除：后面会看到，导致情绪的问题多是之前小问题的日积月累，或者就是现阶段不好解决的问题。

情绪的影响因素很多，简单列举几个很常见的。

- 1) 研发节奏过于紧凑：在上一节中提到当开发的情绪体力持续透支时，会有恶劣的情绪问题。这个在开发团队中并不少见。当开发节奏太过紧凑，团队不注意休整时，团队很容易负面情绪弥漫，而情绪一旦形成印象，便不会那么好消散。
- 2) 薪酬倒挂：这个也是大家诟病 HR/Leader 的重要原因，当一个团队薪酬内部增长太乏力时，内部人员会有流出，团队需要再招聘新人，而市场上平均待遇已经和之前不同，所以新招来的

人员待遇往往也会水涨船高。这个是很致命而且不好消解的。
HR 太过节约成本，往往会对团队有致命的伤害。

- 3) 与 Leader 理念/习惯不合。
- 4) 工作内容安排不当，太困难或太简单，或者与职业发展规划不符。
- 5) 纯粹发泄。
- 6)

情绪问题暴露后，也不是不能解决，有明确的诉求时直接去解决问题本身。没有明确诉求的可能是抱怨性格或者与公司方向不合，那也无法强求。

而真正可怕的是团队 Leader（或者需要对这些问题负责的人）对团队本身情绪的不知情。当大家私密的 QQ 群/讨论组 都没有你，聚会也没有参加，不会有真心话交流，只有工作上例行的接触时，就已经是挺危险的信号，成员离职时再去寻找原因已然太晚。

4. 纠结的 Leader

Leader 这个词并不是太贴切，这个职位的职责应该是服务团队的开发同学，找到并解决大家开发不爽的地方，做好技术和业务的架构，保证整体研发输出的质量和时间点。

而且 Leader 其实并不容易当。要获得工程师的尊重，需要满足下面一项或多项。

- 1) 技术过硬，能解决团队遇到的各种技术问题。
- 2) 情商逆天，有能力和意愿感知团队成员的情绪，并能不断给积极的反馈，团队保持很强的凝聚力。
- 3) 资历深厚，业内有影响力或者披荆斩棘创下了公司的基业，能为团队争取到资源。而在没有得到工程师的充分尊重前，各种措施的执行都会受到影响，技术决策的讨论更得充分尊重大家的意见。

5. 技术驱动

技术驱动业务是产生颠覆式创新的动力之一，工程师更清楚技术的边界在哪里，哪些情景已经可以被成熟（或者半成熟，但可驾驭）的技

术方案来解决了，这些会把公司与竞争对手拉开一个或者半个技术时代，输出更酷炫的产品。

这个时代对于工程师来说是最好的时代，Github 等开源社区的兴起，让新技术的学习成本变得很低。数据挖掘，自然语言分析，图数据库，数据可视化，虚拟化，移动互联等技术的发展更给业务带来了无限的可能，而美国市场与中国市场还存在 5-10 年的时间差，也为我们提供了很多可以参照的模板。

技术驱动有更多实际可以做的事情，放到二手车行业，例如当其他产品靠用户自己填购车需求时，你实现了通过用户的行为轨迹挖掘用户的需求；当其他产品还是几张图片来展示车况，你实现了低成本的全景照片，当其他产品还在要经销商自己维护关系时，你通过图数据库计算出了他可能的朋友圈...

那么问题来了，应该如何推动产生更多的技术推动型的产品呢

宽松的学习氛围：技术驱动型一般借助于相对前卫的技术，大多数同学对这些技术都没有多少经验，依赖于持续的学习，而学习就需要有学习的氛围，尤其是时间的保障。

优秀的工程师：技术驱动对工程师的自我实现需求要求的更高，只想完成现有任务不想多事的工程师显然不合适。

技术与业务的结合：最理想的是工程师本身有商业思维，能够主动将新技术与业务结合起来，寻找最大价值的结合点；其次是工程师定期宣讲技术成果，与产品同学共同讨论。例如，我们已经将 20 万经销商数据全部存入图数据库，支持宽度遍历，深度遍历这些查询方式，他们的时间复杂度是 $O(n+e)$ 。我们可以对这几十万条评论内容进行分析，分辨出褒义还是贬义，还可以匹配上我们数据库中的品牌车系，准确率能有 60%。

技术驱动也有一些硬伤，或者说工程师同学主导项目时都很容易出现的硬伤：优先级，时间点，任务管理。

优先级：醉心于技术的同学会被问题本身吸引，例如，MongoDB 还支持数据分片，那我搭个集群试试。我试试这里能不能承载 1w qps 的压力。虽然我正在看 iBeacon，但是 Arduino 也好酷哇，我做个 Demo 先，等等。在这种吸引下，工程师很难对套页面，修数据这种任务感

兴趣，而这些对项目来说优先级可能会更高。（心理学中也有类似结论，当难度降低到一定程度，动机的强度也会降低。）

时间控制：同时因为要使用的很多是大家没用过的技术，技术本身可能不成熟，大家经验也不多，有时候一些坑要好长时间才能埋上，这样固定的时间点很难保证产出。

任务管理：许多热衷于解决问题的同学同时也是挖坑小能手，他们能预见一种更优雅的解决方案，但是没有时间和精力去完成，在这个过程中还挖了更多新的坑，于是这些坑一直没有时间埋...

也因为以上几个原因，我们会发现很多醉心于技术的同学在做项目时会出现虎头蛇尾，总也结束不了的样子。这种情况需要技术同学自己注意每月确定团队的大方向，定期汇报，发周报或者半月报。

如何提高个人与团队的效率。是会伴随行业发展长久存在的问题，每个团队都要去寻找自己的答案，大家一直在努力。

慎用 Java 日期格式化

作者 曹知渊

2014 年 12 月 29 日, [Reddit](#) 上一条寥寥几语的留言引起了大量的回复, 这条留言说道:

今天有可能变成 2015 年 12 月, 快点修复它。

这条留言实际指向了 Twitter 上的一个[帖子](#), 这个帖子提醒大家, 如果使用了 YYYY 的格式符来格式化日期, 那么就有可能用错格式了。

Reddit 的一位读者解释说, Twitter 由于误用格式符, 把当天的日期变成了 2015 年 12 月的某天。

那么日期为什么忽然变得不对了? 原因是开发人员误用的格式符代表的是一种不同的日历系统。现行的公历通常被称为格里高利历 (Gregorian calendar), 它以 400 年为一个周期, 在这个周期中, 一共有 97 个闰日, 在这种历法的设计中, 闰日尽可能均匀地分布在各个年份中, 所以一年的长度有两种可能: 365 天或 366 天。而本文提到的被错误使用的历法格式, 是国际标准 ISO 8601 所指定的历法。这种历法采用周来纪日, 样子看起来是这样的: 2009-W53-7。对于格里高利历中的闰日, 它也采用“闰周”来表示, 所以一年的长度是 364 或 371 天。并且它规定, 公历一年中第一个周四所在的那个星期, 作为一年的第一个星期。这导致了一些很有意思的结果, 公历每年元旦前后的几天, 年份会和 ISO 8601 纪年法差一年。比如, 2015 年的第一个周四 是 1 月 1 日, 所以 1 月 1 日所在的那周, 就变成了 2015 年的第一周。代表 ISO 8601 的格式符是 YYYY, 注意是大写的, 而格里高利历的格式符是小写的 yyyy, 如果不小心把这两者搞混了, 时间就瞬间推移了一年! 维基百科上也有[词条](#)专门解释 ISO 8601。

作为 Java 开发者, 只要搞清楚 YYYY 和 yyyy 区别, 准确地使用两者, 就不会出现这种错误。Reddit 的评论中也有读者提到, 在 [Joda Time](#) 中, 使用 YYYY 是没有问题的。

Node.js 和 io.js 性能差异巨大

作者 Michael Schöbel 译者 谢丽

[io.js 是 Node.js 的一个分支](#)。近日，一名具有 20 多年软件开发经验的工程师 [Michael Schöbel](#) 通过一组测试[比较了两者的性能](#)。

他分别使用 io.js 和 Node.js 实现了质数查找算法“埃拉托斯特尼筛法（[Sieve of Eratosthenes](#)）”，并且每一种实现又有三种实现形式，分别使用了常规数组、“类型数组（typed-array）”和 buffer。

io.js 和 Node.js 有一个重要的共同点，就是它们都基于 [Chrome 的 V8 JavaScript 引擎](#)。但它们基于不同的 V8 版本。在本次测试中，Schöbel 使用了 Node 0.10.35 和 io.js 1.0.2，都是最新版本。

他在相同的环境下将每一种实现形式执行 7 次，然后取平均执行时间，结果如下（单位：秒）。

	Node.js 0.10.35	io.js 1.0.2
Buffer	4.259	5.006
Typed-Array	4.944	11.555
Regular Array	40.416	7.359

从中可以看出：

- 类型数组和常规数组性能差异巨大；
- 使用 buffer 时，io.js 用时比 Node.js 长 18%；
- 使用类型数组时，io.js 用时是 Node.js 的 2 倍多；
- 使用常规数组时，Node.js 用时是 io.js 的 5 倍多。

对于这个结果，Schöbel 表示：

这不是一个专业的综合性的测试，该测试所用的算法与现实开发中的用法可能完全不同；

由于两种框架基于不同的 V8 JavaScript 引擎，所以这个测试实际上也比较了 V8 的不同版本；

如果一个项目大量使用 CPU，并且部署在 AWS 或 Azure 这样的服务器上，那么选择合适的 Node.js/io.js 版本可以降低成本。

最后请谨记：务必使用不同版本的 Node.js/io.js 测试代码。

新 JavaScript 库的激动人心之处

作者 Philip De Smedt , 译者 侯伯薇

近期在 GitHub 上出现了大量新的 JavaScript 库，我们要来看一下其中非常棒的一些库。

QuaggaJS: 完全使用 JavaScript 编写的条形码扫描程序。

QuaggaJS 是一种条形码扫描程序，完全使用 JavaScript 编写，支持对各种类型的条形码——像 EAN 和 CODE128——的实时定位和解码。

尽管已经存在各种各样的条形码库，但它还是从头编写的，而并没有从流行的 [zxing](#) 库移植过来。 QuaggaJS 实现的特性是一种条形码扫描程序，它能够在图形中找到类似于条形码的样式，得到估计的边界框，包括旋转的情况。 这样，你就可以在图形中做很智能的二维码识别。

如果你想要完全利用 QuaggaJS 的优势，那么浏览器需要支持 getUserMedia 这个 API，它在最新版本的 Firefox、Safari、Chrome 和 Opera 中都已经实现。

这个库暴露了以下 API：

Quagga.init(config, callback)

这个方法会根据给定的配置和回调函数对库进行初始化，回调函数会在载入过程完毕后调用。 如果配置了实时检测，那么初始化过程还会请求访问相机。

Quagga.start()

当库初始化之后，start()方法会启动视频流，并开始对图像进行定位和解码。

Quagga.stop()

如果当前解码器在运行，那么在调用 stop()之后，解码器就不会再处理任何图像。

Quagga.onDetected(callback)

注册一个回调函数，它会在成功定位和解码一个条形码模式之后触发。在调用回调函数的时候，解码后的数据会作为第一个参数。

Quagga.decodeSingle(config, callback)

和上述的方法不同，这个方法不会依赖于 getUserMedia，而会在单独的图像上处理。提供的回调函数和 onDetected 中的一样，会包含解码后的数据作为第一个参数。

[The QuaggaJS 示例库](#)中包含了更多示例和用例。

Lining.js：为 CSS web 排版所用的 JavaScript 插件

[Lining.js](#) 是用来处理带有元素的单独行的库。你只需要在元素上增加 data-lining 属性，就可以使用 Lining.js，然后使用 CSS 来设置它的样式。

在 CSS 中我们已经拥有::first-line 这个选择器，可以在元素的第一行上应用样式。但并没有类似于::nth-line()、::nth-last-line()或者::last-line 之类的选择器。Lining.js 对你的文本提供了完整的行控制，如下示例所示：

```
<div class="poem" data-lining="">Some text...</div>
<style type="text/css">.poem .line[first] { /* ` .poem::first-line` */ }
.poem .line[last] { /* ` .poem::last-line` */ }
.poem .line[index="5"] { /* ` .poem::nth-line(5)` */ }
.poem .line:nth-of-type(-n+2) { /* ` .poem::nth-line(-n+2)` */ }
.poem .line:nth-last-of-type(2n) { /* ` .poem:::nth-last-line(2n)` */ }
</style>
<script src="YOUR_PATH/lining.min.js"></script>
```

现在，Lining.js 只支持主要浏览器，像 Chrome、Firefox、Safari 和 Opera。但不支持 IE。

InteractJS：使用 JavaScript 实现拖放、缩放和多点触控手势

InteractJS 是一个 JavaScript 模块，它为最新的浏览器（包括 IE8 以上版本）增加了拖放、缩放和多点触控手势，并带有惯性和快照功能。

这个库的主要目的是替换 jQuery UI 所提供的功能。因此，使用 InteractJS 来编写的 web 应用在智能手机和平板上会更加易用。

InteractJS 是一个轻量级的库，可以与 SVG 技术协作，处理多点触控输入，而把渲染元素以及设置其样式的任务留给了应用程序。

官方的 [InteractJS 站点](#) 提供了拖拽、快照、缩放和多点触控旋转的示例和用例。

TreeJS：构建和操作可挂钩的树

Tree.js 是一种用来构建和操作可挂钩的树的 JavaScript 库。对于查找和遍历目录结构，它是一种很有用的插件。

想象一下，你在 web 应用程序中有一个管理部分，需要浏览文件系统。那么使用 Tree.js，你就可以像下面这样来展示文件系统：

```
javascript

var myTree = Tree.tree({
  children: [
    {
      name: 'dupuis',
      children: [
        {
          name: 'prunelle',
          children: [
            {
              name: 'lebrac',
              job: 'designer'
            },
            {
              name: 'lagaffe',
              firstname: 'gaston',
              job: 'sleeper'
            }
          ]
        }
      ]
    }
});
```

为了找到一个节点，你可以传递任意一个有效的目录结构作为参数，就可以在这个树中搜索。

```
javascript

var lebrac = myTree.find('/dupuis/prunelle/lebrac');

lebrac.data() // { name: 'lebrac', job: 'designer' }

lebrac.attr('job'); // designer

lebrac.path(); // /dupuis/prunelle/lebrac
```

GitHub 上的 [The Tree.js 库](#) 提供了其他使用挂钩和保证 (promises) 的案例。

查看英文原文: [What's Exciting in New JavaScript Libraries](#)

高效运维最佳实践（01）：七字诀，不再憋屈的运维

作者 萧田国

【专栏介绍】“高效运维最佳实践”是 InfoQ 在 2015 年推出的精品专栏，由触控科技运维总监萧田国撰写，InfoQ 总编辑崔康策划。

前言

做运维的那么多，快乐的能有几个？

我们那么努力，为什么总感觉过得那么憋屈、苦闷？做的事情那么多，为什么业务部门、直接领导和公司貌似都那么不领情？怎么做才能自己更加开心些？

本专栏的主线实际是一个运维人员的十年成长史，从菜鸟到运维总监。但不是基础技术教学，也不会在运维技术的某一方面过深涉及。更多的是应用技巧、实践经验及案例剖析。专栏中的系列文章，包含作者在运维各个细分领域的技术和个人成才的心得体会。因此也可以成为广大运维朋友的工具书，伴随大家从初级运维成长为高级技术型运维管理人才。

技术专栏就非得那么中规中矩么？咱这个专栏试图以更轻松活泼的文字，徐徐道来，就当是个老朋友，轻松愉快中，希望能给大家带来收获和启发。

前段时间有位 IT 大佬在网络上发声，我这么有钱，为什么不幸福？诚然，有钱是幸福的最重要条件之一，但有钱就一定幸福么？真的是充分必要条件？当然更悲催的是运维行当，技术好是被认可（幸福）的最重要条件，但技术再好，外部门不说咱们“坏话”，已经是很不错的了。

补充一句：本文实际上既是本专栏的开篇，也是综述。后续专栏文章，会就各个部分，进行详细阐述，敬请期待。

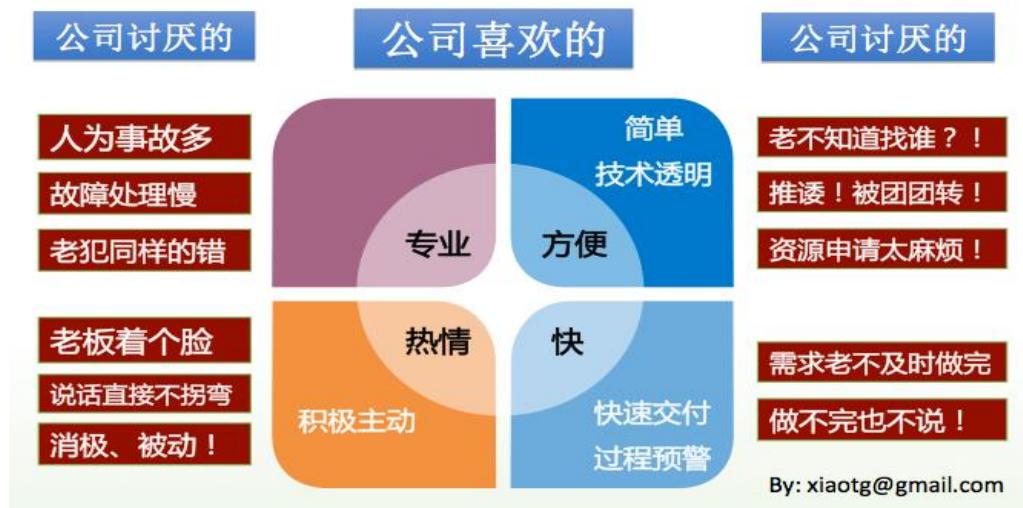
本文略长，主要目录结构如下：

1. 什么是高效运维
2. 为什么难以做到高效运维
 - 2.1 糟糕的分工及连环反应
 - 2.2 做 vs 说的困境
 - 2.3 资源错配
3. 如何做到高效运维
 - 3.1 明确分工/职责
 - 3.2 技术的专业化
 - 3.3 管理的专业化
 - 3.4 良好的客户界面
4. 小结

好吧，我们正式开始。

1. 什么是高效运维

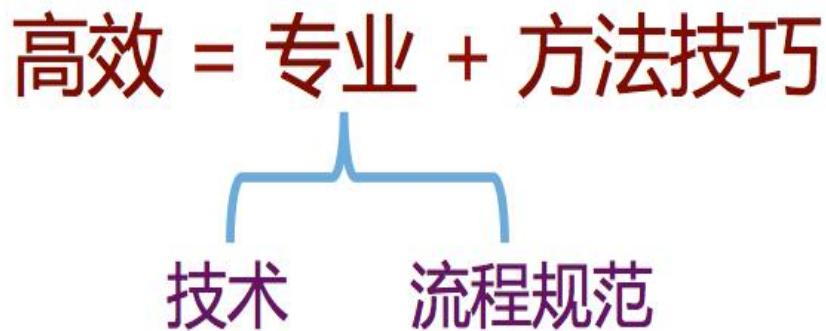
我们收集了一些来自外部门对运维的印象（tou）象（su），如下图所示。其中，大家看是否也多少有自己的影子？



往往看自己都很美，但从外部门来看，槽点多到乃至无力吐槽。首先，做事情不专业，人为事故多（更多是低级的人为事故）；很多时候，都是我们业务部门告诉运维，运维才知道发生故障了，而且故障解决

时间过长；做个调试，老超出调试时间，超时也不说，是不是完成了也不知会一声；部门内老玩踢皮球的游戏，做个需求，老让我挨个找人；申请个服务器，老费劲了，扔我一个申请表，当自己是衙门呢？或者扔我一个技术文档，我哪看得懂？

专业、热情、方便、快，这是为根治上述各种疑难杂症，经多年自我治疗并综合各方经验，得出的高效运维七字诀。我们用一个简单的公式来表示高效和专业的关系。专业是高效的基石，否则无从谈起高效与否，而技术是专业的基石。但这恰恰也是运维技术人员的误区所在，误以为，技术比较强，就足够了，并因此而忽视其他重要方面。



实际上，对外部门而言，运维是个黑盒子，是一个输入输出的关系：外部部门提出需求，运维给出结果：完成、或未完成。本质上而言，外部部门不关心（也无法关心）我们采用什么技术来实现的，只关心是否如期完成。

合理的流程规范，就像血液，能让部门稳定而高效的运转，大家都觉得开心，这也是专业与否的重要组成部分。但如果希望做到高效运维，良好的客户界面、合适的方法技巧，也非常有必要。这就像网站的 UI，给人感觉舒服了，后面很多事情也能轻松愉快、顺理成章地进行。

2. 为什么难以做到高效运维

做不到高效运维，公司和业务部门不满意，上级领导不满意，自己也不满意。原因很多，我们从管理者和员工角度分别来讲。

2.1 糟糕的分工及连环反应

发生在中小公司的糟糕情况，往往从不明确的分工，开始悲剧之旅。有些游戏创业公司，刚开始时运维人员也就2、3个，基本每人都得会运维的各个工种，游戏运维、网站运维（Nginx/PHP等）、数据库运维（MySQL等）、系统运维（Linux/Windows等）、服务器上架、故障报修、甚至做网线。

公司业务扩大很多后，如果运维组织结构不随之而变，分工不明确，就会发现大家都在疲于奔命，什么都会的结果就是什么都不精。在运维技术如此庞杂的今天，就是把人活活的架在火上烤。这样引发的是多米诺骨牌效应：分工不明确 → 职责不清楚 → 考核不量化→ 流程不合理→ 缺规范、少文档。

2.2 做 vs 说的困境

一般运维技术人员都不善于沟通（至少表面上，虽然大家都普遍有火热的心，呵呵）。在微信、QQ 大行其道的今天，这个问题变得更严重，而不是减轻。这也和工作性质有关，想想，一天到晚和服务器说话的时间，比和人说话时间都多。

另外，从人脑结构来看，做和说两难全，也是合理的。控制计算、推理能力的是左脑，而表现力等由右脑控制。如果强行要求会做还会说，说不定会导致紊乱、崩溃甚至“脑裂”呢，呵呵（当然，这个问题也是有解决方法的）。

更严重的是，很多同学没意识到自己的沟通表达是有问题的，说句话能把人呛死，也不知道如何有效表达。这样就谈不上热情了。

2.3 资源错配

管理者和员工都可能存在资源错配的问题。对管理者而言，包括人员错配和时间错配，员工主要是时间错配。

管理者如果把错误的人安排在错误的岗位，那么注定是个错误。例如，某位同学喜欢钻研技术，不喜表达，非得让他作为和外部门的接口人，那自然费力不讨好，大家都不开心。

管理者的时间错配包括三种情况。

1) 沉迷解决技术问题。这一般发生在刚从技术岗位提拔为管理者的时候，忘记自己是管理者了。解决复杂技术问题，能带来愉悦感，否则

就是挫折感。于是遇到技术问题时，非得死磕到底，然后一周过去了，而部门其他同学却放羊一周。

2) **一心扑在管理上**。这又是一个极端了，忘记自己的技术身份。把自己变成一个项目经理，整天只关心时间节点，不关注技术人员的小情怀，不协助他们解决具体的技术问题。

3) **沉迷单个业务模块**。这是另一个特例。一般发生在内部提拔时。例如某位同学，之前是 DBA 组的负责人，提拔为运维部经理后，还是习惯于抓其擅长的数据库工作，这也是不应该的，否则就没必要提拔了嘛。

员工的资源错配主要体现在时间安排上。事情多了，分不清轻重缓急，没有一个合理的排序原则、指导思想；混淆技术进步和工作要求（有时过分追求技术进步），简单的问题复杂化，降低客户满意度。

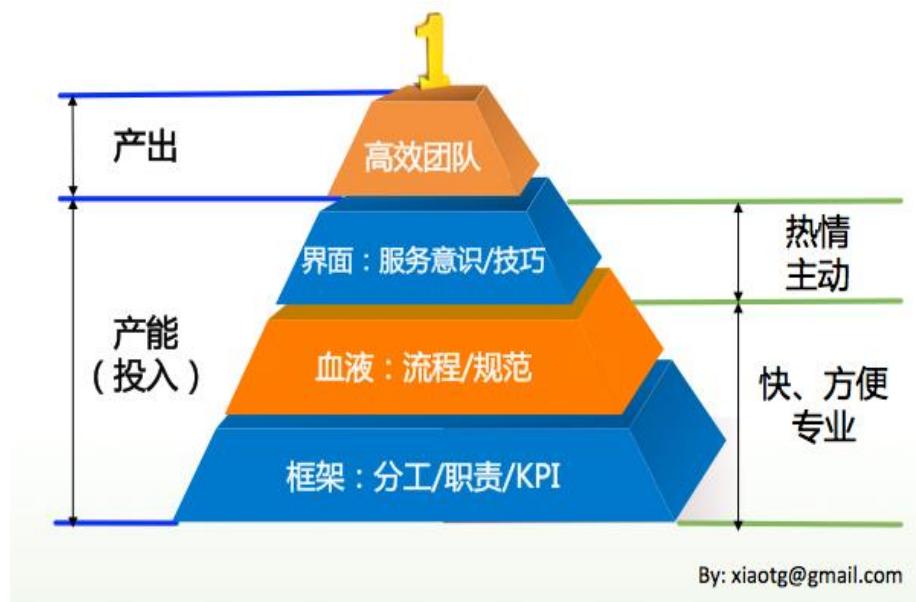
3. 如何做到高效运维

高效运维从来不是一个简单的事情，需要多方面共同努力来实现，本文先择其要点简述之，以后专栏系列文章会有更多深入阐述。

3.1 明确分工/职责

美国著名管理学者史蒂芬·柯维在畅销书《高效能人士的七个习惯》中提出了产出/产能平衡原则。想多产出，先得扩大产能。想金鸡多下蛋，就不能杀鸡取卵。那么对于高效运维而言，产能是什么呢？

包括三部分：1) **框架**，即合理的分工/职责/KPI，抱歉我提到了 KPI，多么让人如此爱恨交织的词语；2) **血液**，即专业的流程/规范；3) **界面**，即良好的服务意识/技巧。这些投入足够多，才会得到心仪的产出——高效运维。在贯彻实施这些一段时间后，外部门会诧异的感觉：哟，怎么运维变化这么大。虽然他们不知道原因，但我们可以微微一笑，呵呵。



具体到运维部门而言，我们的分工，区别于内网 IT 部。一个是服务外部客户，一个是服务内部客户，差别还是蛮大的。根据部门分工，拆解出各个小组的分工，再落实到每个员工头上。有章法，大家也觉得舒心。

运维是支持部门，成本中心，难以产生利润。所以其中重要的考核指标其实是客户满意度，请相关业务部门给运维同学打分，运维内部根据分工，也可以相互打分，这对应着外部满意度和内部满意度。KPI 虽然令人不舒服，但总的来说，还是有存在的合理性。



3.2 技术的专业化

技术上的专业化运维，涉及面也很广，下面仅列举几例。

3.2.1 优化监控系统

谁来监控监控系统？怎么保证比业务部门先发现问题？是否需要添加业务监控？URL 监控是否返回状态码 200 即万事大吉？是否需要文件监控？短信报警、邮件报警是否足够？是否需要自动语音报警及垂直升级功能？

监控是门学问，是专业运维的入口。展开说可以很大篇幅，先抛砖引玉，提出这些问题。实际上，对于资深、聪明的运维同学，看到问题，就已经有了自己的答案。

3.2.2 减少人为事故

人为事故是运维最头疼、最不专业的事情之一。例如网站运维中，如果每次更新都需要登录服务器，svn update/git pull，难免会出差错。所以可以用类似 Jenkins 的工具，实现 Web 更新，这样，除非重大更新（包括数据库更新），否则都只需要点点鼠标即可。甚至，可以把网站更新外包回开发部门，这样还能减少运维操作带来的沟通成本、时间成本。

3.2.3 运维自动化

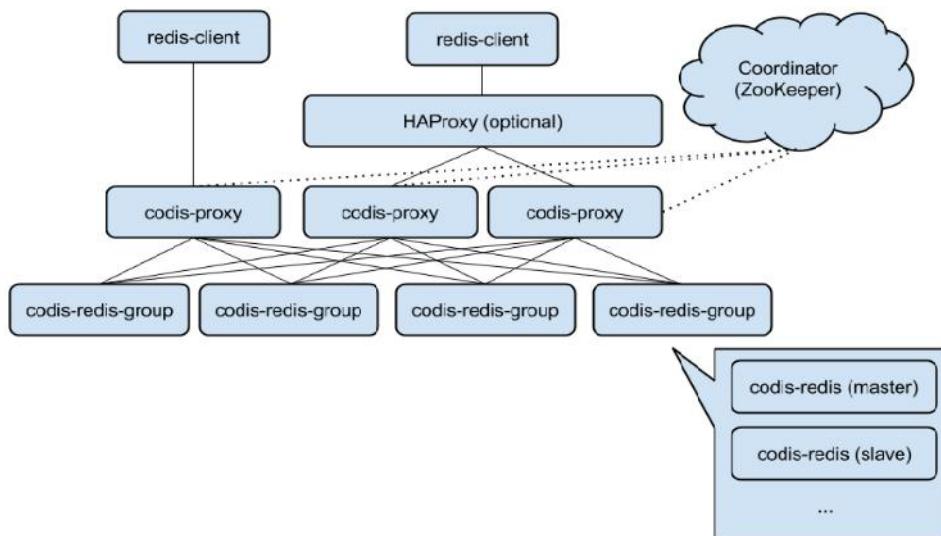
运维自动化是个大课题，网络上的讨论也很多。建议选择合适自己的方式、方法。轻量级工具如 ansible，无需在被管理服务器安装客户端程序，这在针对多台服务器进行分发管理（特别是管理仅有临时账号权限的服务器时），具有较大优势。另一个吸引人的地方是，操作结果和操作日志集中存储。

3.2.4 合理优化架构

近几年国内优秀的开源软件层出不穷，设计和优化架构，很多时候并不是非得自己从零起步来搞。例如 Redis，以其高效、稳定，已成为缓存系统的最好选择之一，但 Redis 单实例的支撑能力有限，目前 Redis 集群的实现，大多采用 Twemproxy，但使用起来老感觉有些美中不足，那么，有没有一个取而代之的产品？

Codis 就是其一，Codis 由豌豆荚开源，并广泛用于其自身的业务系统。Codis 刚好击中 Twemproxy 两大痛点（无法 sharding，运维不友好）。Codis 可以平滑的扩容/缩容，随时增减 Redis 服务器；并提供友好的运维界面，不仅能看到 Codis 系统运行情况，还能进行数据迁移、主备切换等操作。

Codis Architecture



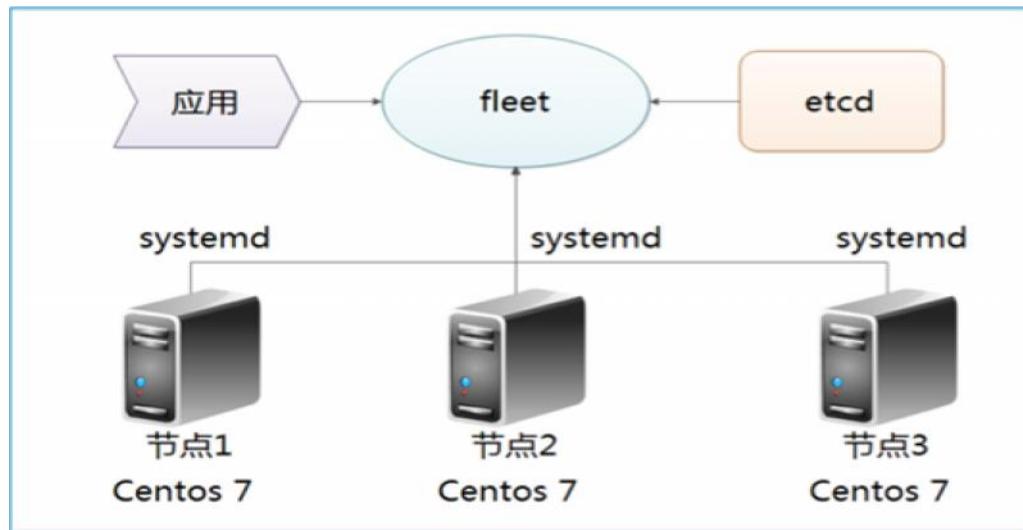
另外，Codis 还提供工具，将依赖于 Twemproxy 的 Redis 集群，平滑的迁移至 Codis（太酷了，那画面太美，我简直不忍看）。性能方面，经我们实测，在正常 Value 长度下，Codis 的 get/set 性能，优于 Twemproxy。

3.2.5 代码持续部署

线上系统程序代码可否自动打包、持续部署？测试环境的新版本发布可否由开发人员自己来做，甚至自己来做测试？这些无疑可以很大提升运维和开发效率。

Docker 高可用集群，加上 Jenkins 发布，可以把这些需求变成现实。Centos 7 的 systemd 用来底层支持 Docker 高可用，etcd 实现了配置文件的集中存储，而不是分散在各台服务器的本地。fleet 作为 etcd 和 systemd 之间的桥梁，并通过 systemd 来控制集群服务器。

Jenkins 从 svn 服务器获取到新代码版本后，通过 shell 脚本，打包成 image，放入 Docker 私有库，从而被 Docker 集群服务器 update 并使用。



3.3 管理的专业化

管理上的专业化运维，甚至包括调试通报和故障通报，都很有说法。系统运行一段时间后趋于稳定，调试/更新就变成了故障的主要来源之一，怎么让调试少出人为事故，顺利如期的完成？这是个技术活。

3.3.1 运维 345 法则

故障通报是细究故障的不二法门，一次长时间的故障，往往有很多细节可以推敲，我们总结出运维 345 法则。3 是指故障时长被分成三部分，4 是指对应的四个故障时刻点，5 是指在这个过程中我们可以做的五件事。这样，我们就可以有的放矢地进行优化解决了。



3.3.2 不要让流程吞噬责任

流程规范是很好，不可或缺，好处谁都晓得。只是，流程有时会成为挡箭牌，会让人变得本位，不愿担当，也不愿从事个人职责之外的事情。

【不要让流程吞噬责任】

流程的价值，在于能规避一些人性的弱点；但拘泥于流程，则同样会埋葬人性的光辉——责任感，以及基于责任才会迸发的判断力与行动力。

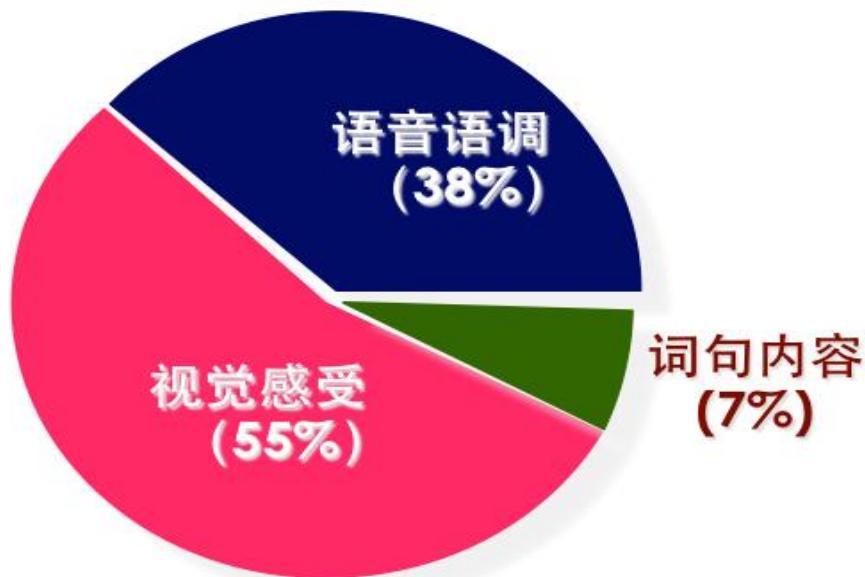
这其实是错误的、短视的，“害人害己”的。如果真的出了一个非常严重的故障，个人就能“出污泥而不染”么？没戏。如果是顶级故障，老板想的甚至是把整个运维部门端掉，皮之不存、毛将焉附？

3.4 良好的客户界面

伸手不打笑脸人。合适的言语表达，可以大事化小、小事化了，反之亦然。只是对做技术的运维同学而言，这是很不容易的事情，甚至有人宁愿多加班，也不去和人沟通。但，工作的要求有时往往需要善于表达，其实也可以换个角度想，把良好的沟通当做一门技术来攻克，如何？

3.4.1 当面沟通

即时聊天工具如 QQ、微信实际上是加剧了沟通成本。大家变得更加依赖于此，本来当面沟通或电话沟通，几分钟就能说明白的事情，来来去去几十分钟，更有甚者，还能吵起来，没法愉快的玩耍了。根据国外一项调查，一次有效沟通中，词句内容仅占据一小部分。



我们一般都会要求素未谋面的小伙伴，先当面聊一下。举个真实的例子，有位同学之前和某位运营同学一直 QQ、邮件沟通，某次实在说不清楚，于是面聊，发现对方居然是个美女，于是之后合作很愉快（虽然美中不足的是，该女士已有男友）。

3.4.2 来的都是客

做运维的，应该放下身段，不一定非得低三下气地做事情，但至少意识得到位。运维的沟通中，也适应心理学的投射原理：越是觉得别人盛气凌人、服务不到位，其实自己也往往是这样的。

来的都是客。如果自己实在忙不开，响应慢。礼貌用语总是可以的嘛，不好意思，对不起，抱歉，谢谢。

4. 小结

絮絮叨叨说了这多，也不知大家看烦木有。运维很苦闷，让苦闷的人变得更苦闷。但不管怎样，也是一门技术。这年头，有门手艺，虽然发达需良机，但至少生存无忧。话说回来，哪行都不容易。

本人做运维这么些年，结合各种失败与成功、痛苦与苦痛的经验，终于悟出高效运维的七字诀：专业、热情、方便、快。不一定完全适合您，但终归是多年的领悟，自成一个小体系，如各位盆友喜欢，以后逐一阐述，如能对大家有所裨益，幸莫大焉。

对了，下一篇篇文章的主题类似《我技术能力这么好，为什么不幸福（员工篇）》，如您愿意，可以持续关注。

关于作者

萧田国，男，硕士毕业于北京科技大学，目前为触控科技运维负责人。拥有十多年运维及团队管理经验。先后就职于联想集团（Oracle 数据库主管）、搜狐畅游（数据库主管）、智明星通及世纪互联等。从 1999 年开始，折腾各种数据库如 Oracle/MySQL/MS SQL Server/NoSQL 等，兼任数据库培训讲师若干年。

曾经的云计算行业从业者，现在喜欢琢磨云计算及评测、云端数据库，及新技术在运维中的应用。主张管理学科和运维体系的融合、人性化运维管理，打造高效、专业运维团队。

近来有时参加一些大小技术会议，做做演讲嘉宾或主持人（有空找我来玩呀）。

对了，我计划在 2015 年 4 月 23-25 日的 [QCon 全球软件开发大会](#) 上，就自动化运维话题[进行分享](#)哦。

我的微信号 xiaotianguo。如需更多了解，请百度“萧田国”。另外，我也有微信公众号了，微信搜索“开心南瓜 by 萧田国”或扫描头像边的二维码，和我微信互动。公众号里将有我原创的各类技术和非技术文章，以及我喜欢的文章、帖子。一起来吧。

肉肉的植物



多肉植物亦称多浆植物、肉质植物，在园艺上有时称萌肉肉、多肉花卉，但以多肉植物这个名称最为常用。全世界共有多肉植物一万余种，它们都属于高等植物(绝大多数是被子植物)。在植物分类上隶属几十个科，个别专家认为有 67 个科中含有多肉植物，但大多数专家认为只有 50 余科。多肉植物是指植物营养器官的某一部分，如茎、叶或根(少数种类兼有两部分)具有发达的薄壁组织用以贮藏水分，在外形上显得肥厚多汁的一类植物。它们大部分生长在干旱的地区，每年有很长的时间根部吸收不到水分，仅靠体内贮藏的水分维持生命。有时候人们喜欢把这类植物称为沙漠植物或沙生植物，但是不太确切，虽然多肉植物确实有许多生长在沙漠地区，但是也有不生长在沙漠的。

多肉植物喜欢凉爽的半荫环境，主要生长期在春、秋季节，为多肉植物中的“中间型”品种。耐干旱，不耐寒，忌高温潮湿和烈日暴晒，怕荫蔽，也怕土壤积水。生长期应给予充足的光照，若过于荫蔽，会造成株形松散，不紧凑，叶片瘦长。如光照过强，则叶片生长不良，呈浅红褐色，有时强烈的直射阳光还会灼伤叶片，留下难看的斑痕，在半荫处生长的植株，叶片肥厚饱满，透明度高。生长期浇水掌握“不干不浇，浇则浇透”，避免积水，更不能雨淋，尤其是不能长期雨淋，这些都是为了避免烂根。但也不宜长期干旱，否则植株虽然不会死亡，但叶片干瘪，叶色黯淡，缺乏生机。

架构师

www.infoq.com/cn/architect

每月8号出版



促进软件开发领域知识与创新的传播

架构师

ARCHITECT



解读2014 | Review
解读2014之Android篇：连接世界

推荐文章 | Article
新JavaScript库的激动人心之处
高效运维最佳实践

专题 | Topic
Web API设计方法论
亿级用户下的新浪微博平台架构
Netty百万级推送服务设计要点

InfoQ 

2015年02月

架构师 2月刊

每月 8 号出版

本期主编：崔康

流程编辑：丁晓昀

发行人：霍泰稳

读者反馈/投稿：editors@cn.infoq.com

微博、微信： @infoqchina

商务合作：sales@cn.infoq.com

本期主编

崔康 InfoQ 中国总编辑，致力于中国 IT 领域知识与创新的传播，目前负责 InfoQ 整体内容的品牌和质量，同时担任 QCon、ArchSummit 大会的总策划。技术人出身，毕业于天津大学计算机学院，在加入 InfoQ 之前，在某大型外企长期担任协作软件平台的技术负责人和架构师，在相关技术领域积累了一定的经验。从 2008 年开始参与国内社区的技术传播，在多家科技媒体先后发表过数十篇文章，并出版多本译著，总产量超过 50 万字。可以通过 tyler.cui@infoq.com 或者微博（“崔康 Tyler”）与他联系。

