

# 架构师 ARCHITECT



## 本期专题 | Topic | 支付宝的测试

40 | 支付宝分布式事务测试方案

44 | 支付宝的性能测试

56 | 数据设计测试分析方案

## 推荐文章 | Article

62 | 一秒钟法则：来自腾讯无线研发的经验分享

68 | 聊聊并发——生产者消费者模式

## 特别专栏 | Column

74 | Node.js 软肋之回调大坑

86 | 快乐 Node 码农的十个习惯



卷首语

# 程序员的理想

(InfoQ 编辑注：本期内容原定用作 5 月刊内容，但 5 月刊由于各种原因未能完成制作，因此作为 6 月刊内容发布。)

4 月的最后一个周末，大连的连一团队策划了一场 TEDx 活动，活动的主题是理想主义。我也作为团队的一员，做了一些微不足道的工作。本来差一点可以邀请 InfoQ 中文站的泰稳同学来和大家做一场关于理想主义的分享，但非常不凑巧的是，活动的日期正好是 QCon 北京的举办日期，所以未能成行。

在那次活动中，多位来自于各个领域的讲师和大家分享了他们心中的理想，那么对于我们这些程序员来说，心中的理想又是什么呢？

有人会说，我的理想就是能够用程序来解决世间一切问题。管他什么业务流程，销售、采购、财务、生产、合同流转、公文审批，只要是用了我写的系统，一切都可以高效、高质，在极其流畅的情况下顺利完成。各种各样的业务人员，都可以没事儿就休息，把工作交给系统做就好啦。

有人会说，我的理想就是能够实现软件行业的世界大同。我们可以按照别人的需要编写程序，而不需要靠这个养家糊口，只是用来帮助别人完成难以完成的工作，实现自己的人生价值。而且，当我们需要的时候，就能够找到需要的各种软件来使用，而不需要因为那而花费任何金钱。这似乎就是 IT 世界中的共产主义。

还有人会说，我没有那么高的理想，我就希望我们这群技术人能够远离各种政治，能够靠着自己的技术赢得丰富的回报，可以衣食无忧，写自己最喜欢写的程序，读自己最喜欢读的书，每天都快快乐乐，和家人朋友一起幸福地生活。

然而，还有句话叫做，理想是丰满的，而现实是骨感的。

所以，我们还是需要承认，计算机、系统、编程语言都是工具，最终是为业务服务的。很多事情并非能够靠系统来解决，至少在我们还没有实现黑客帝国中的情境，每个人都通过终端接入到大大的网络中之前，还是要靠真正意义上的人来做。我们能够做到的只是，在一些情况下，帮助人们解决机械化的工作，或者帮忙存储海量的数据、做大规模的分析和计算，然而，人工智能还没有发达到足够的高度，我们还需要依赖于人来做这一切。

所以，我们还是需要靠编写各种各样的系统和程序来维持生计，在需要使用他人编写的软件时，还是需要花费一些金钱。尽管现在已经有很多开源的软件，但并不足以满足我们的需要；你也可能说现在很多游戏是免费的，但其中可能会隐藏的内购项目，想要真正玩得过瘾，还是需要花费大量资金。

所以，我们还是要承认，有的地方就有政治，很多时候，技术人还是会处于不利的地位，有时候，还是需要为了赚取能够满足我们的衣食住行各个方面所需的费用而努力工作。甚至于，有些时候太忙了，没有时间和家人、朋友相处。

尽管如此，拥有理想还是非常好的事情，至少那是我们的奋斗目标所在。如果都没有了理想，那么又怎么能够体会到风雨后彩虹的美丽？

那么，回过头来说说，我自己的理想是什么呢。

曾经的理想是做好程序员的工作，把程序尽量写得漂亮，没有 bug，能够很好地完成业务客户所需要的功能。

后来的理想是能够和业务人员一些完美地协作，不仅帮助他们用计算机手段解决问题，而且能够和他们一起制定各种制度，完善各种各样的需求。

现在的理想，是希望自己可以把多年来积累的知识和经验，以更好地形式分享给大家，更好地影响更多人做积极的改变。并且也让更多人受到自己的影响，把各自的宝贵经验都分享出来，大家一起提高进步。

似乎现在的理想真的是任重而道远，但我想只要确定了目标，并为之坚持不懈的努力，终究会成功，不是吗？

亲爱的读者朋友们，你的理想又是什么呢？

本期主编：侯伯薇

# 目录

## 卷首语

2 | 程序员的理想

## 人物 | People

6 | 左耳朵耗子谈云计算：拼的就是运维

12 | Javascript 高性能动画与页面渲染

30 | 对话 Facebook 人工智能实验室主任、深度学习专家 Yann LeCun

36 | NoSQL、JSON 和时间序列数据管理：Anuj Sahni 访谈

## 专题 | Topic

38 | 本期专题：支付宝的测试

40 | 支付宝分布式事务测试方案

44 | 支付宝的性能测试

56 | 数据设计测试分析方案

## 推荐文章 | Article

62 | 一秒钟法则：来自腾讯无线研发的经验分享

68 | 聊聊并发——生产者消费者模式

## 特别专栏 | Column

74 | Node.js 软肋之回调大坑

86 | 快乐 Node 码农的十个习惯

## 避开那些坑 | Void

90 | 修复 bug 与解决问题——从敏捷到精益

96 | 不要就这么放弃了 SQL



促进软件开发领域知识与创新的传播



中文 | 英文 | 日文 | 葡文 | .....

# 左耳朵耗子谈云计算：拼的就是运维

作者 陈皓

本文根据 InfoQ 中文站跟陈皓（@ 左耳朵耗子）在 2014 年 3 月的一次聊天内容整理而成。在沟通中，陈皓分享了自己对云计算的理解，包括云计算为什么会分三层，实现一个云平台的难点在什么地方，运维之于云计算的重要性，电商云为什么有价值等。

## 嘉宾简介

陈皓（@ 左耳朵耗子），[CoolShell.cn](#) 博主。15 年软件开发相关工作经验，8 年以上项目和团队管理经验。擅长底层技术架构，团队建设，软件工程，软件研发咨询，以及全球软件团队协作管理。对高性能，高可用性，分布式，高并发，以及大规模数据处理系统有一些经验和心得。喜欢关注底层技术平台和互联网行业应用。技术擅长 C/C++/Java 和 Unix/Linux/Windows。曾于 Amazon 中国任研发经理，负责电子商务全球化业务（全球开店）和全球库存预测系统的研发。曾在阿里巴巴北京研发中心、商家业务部曾任资深专家一职，负责电商云平台、开放平台，云监控和电商多媒体平台。现在阿里巴巴核心系统专家组从事阿里核心系统和阿里云 ECS 相关的虚拟化平台的开发工作。

## 对云计算的定义

云计算其实跟 PC 机有一样的概念，有 CPU、硬盘、操作系统、应用软件。云计算的计算节点（虚拟机）就是 PC 中的 CPU，数据缓存服务就是 PC 的内存，存储节点就是 PC 的硬盘，提供数据服务，让数据不丢、高可用，PC 中的控制器就是云计算的控制系统。PC 机的硬件上面要有操作系统。操作系统很大一块是给开发人员提供系统的 API 接口，提供系统监控以看运行情况，并且还要有系统管理——如用户账号的权限管理、备份恢复等等。操作系统上面要有应用软件，这样才能服务于最终用户，应用软件才是真正落地的业务，这样才会有用户；有了用户，整个体系就运转起来了。

这就是工程师说的 **stack**，也就是我们听到的 IaaS、PaaS、SaaS 三个层。IaaS 层就像 PC 机的基础硬件加驱动程序，PaaS 层就像 PC 机上的操作系统——把基础硬件抽象、包起来并屏蔽硬件和硬件驱动细节、调度基础硬件，而 SaaS 层就是 PC 机里的应用软件。另外，我们还得给开发人员提供各种开发框架、类库和开发环境，这就是为什么 AWS 还做通知、消息、工作流，这是用于粘合操作系统和业务层的，比如可以让你方便地做水平扩展和分布式。云计算自然也会像 PC 机一样，三个层上都会有用于控制和管理的系统。这就是为什么云计算会做成这个样子，其实计算机的发展就在这个圈子里绕。

其实，最终用户基本并不关心你 CPU 用的啥，存储用的是啥，你用什么框架开发，他们关心更多的是可以解决什么问题，有什么样的用户体验。像以前 Windows 用户体验之所以比 Linux 好，就是因为应用层用的舒服；而 Linux 对开发者的用户体验比 Windows 好，就是因为其开放和可以让开发人员更灵活、更自由。我们可以看到 SaaS 层上有的像 SalesForce、Dropbox、Evernote、Netflix 这样的给最终用户的服务，他们更倾向于最终用户和业务。

说到底，云计算的 IaaS、PaaS、SaaS 最后那个 S 都是 Service。就是说，无论你云计算长成什么样，都得要向用户提供“服务”而不仅仅是软硬件和各种资源。

## 云计算的技术难点

到今天，云计算的工业实现已经不太难了。现在有开源软件 **KVM** 和 **Xen**，这两个东西基本把虚拟化搞定；而 **OpenStack** 则把管理、控制系统搞定，也很成熟。**PaaS** 也有相应的开源，比如 **OpenShift**，而 **Java** 里也有 N 多的中间件框架和技术。另外分布式文件系统 **GFS/TFS**，分布式计算系统 **Hadoop/Hbase** 等等，分布式的东西都不神秘了。技术的实现在以前可能是问题，现在不是了。

对于云计算工程方面，现在最难的是运维。管 100 台、1 万台还是 100 万台机器，那是完全不同的。机器少你可以用人管理，机器多是不可能靠人的。运维系统不属于功能性的东西，用户看不见，所以这是被大家严重低估的东西。只要你做大了，就必然要在运维系统上做文章。数据中心 / 云计算拼的就是运维能力。

为什么我说运维比较复杂，原因有这么几个。

一方面，云计算要用廉价设备取代那些昂贵的解决方案。所谓互联网的文化就是屌丝文化，屌丝就是便宜，互联网就是要用便宜的东西搭建出高质量的东西，硬件和资源一定不会走高端路线——比如 **EMC**、**IBM** 小型机、**SGI** 超级计算机等等，你如果用它去搭建云计算，成本太贵。用廉价的解决方案代替昂贵的解决方案是整个计算机发展史中到今天唯一不变的事情。所以如果你要让夏利车跑出奔驰车的感觉，你需要自己动手做很多事，搭建一个智能的系统。用廉价的东西做出高质量的东西，运维好廉价的设备其实是云计算工程里最大的挑战。

另一方面，因为你机器多了，然后你用的又不是昂贵的硬件，所以故障就变成了常态，硬盘、主板、网络天天坏。所以，没什么好想的，运维就必须跟上。云计算的目标是在故障成为常态的情况下保证高可用——也就是我们所说的，你服务的可用性是 3 个 9、4 个 9 还是 5 个 9。

最后，这一大堆机器和设备都放在一起，你的安全就是一个挑战，一方面是 **Security**，另一方面是 **Safety**，保证数十台数百台的设备的安全还好说，但是对于数万数十万台的设计，就没有那么简单了。

所以，面对这样的难题，人是无法搞得定的，你只能依靠技术来管理和运维整个平台。比如必须有监控系统。这跟操作系统一样，对资源的管理，对网络流量、**CPU** 利用率、进程、内存等等的状态肯定要全部收集的。收集整个集群各种节点的状态，是必然每个云计算都有的，都是大同小异的。

然后，你还要找到可用性更好的节点，这需要有一些故障自检的功能。比如阿里云就遇到过磁盘用到一定时候就会莫名其妙的不稳定，有些磁盘的 **I/O** 会变慢。变慢的原因有可能是硬盘不行了，于是硬盘控制器可能因为 **CRC** 校验出错需要多读几次，这就好比 **TCP** 的包传过来，数据出错了，需要重新传。在这种硬盘处理半死不活的状态时，你肯定是需要一个自动检测或自动发现的程序去监控这种事情，当这个磁盘可能不行了，标记成坏磁盘，别用它，到别的磁盘上读复本去。我们要有故障自动检测、预测的措施，才能驱动故障，而不是被动响应故障，用户体验才会好。换句话说，我们需要自动化的、主动的运维。

为了数据的高可用性，你只能使用数据冗余，写多份到不同的节点——工业界标准写三份是安全。然而，你做了冗余，又有数据一致性问题。为了解决冗余带来的一致性问题，才有了 **paxos** 的投票玩法，大家投票这个能不能改，于是你就需要一个强大的控制系统来控制这些东西。

另外，公有云人来人往，里面的资源和服务今天用明天不用，有分配有释放，有冻结，你还

要搞一个资源管理系统来管理这些资源的生命状态。还有权限管理，就像 AWS 的 IAM 一样，如果没有像 AWS 的 IAM 权限管理系统，AWS 可能会不会像今天这样有很多大的公司来用。企业级的云平台，你需要有企业级的运维和管理能力。

## 云计算的门槛

为啥云计算有这么多开源的东西，却不是人人都能做？我觉得有以下原因：

一方面，这就跟盖楼一样。盖楼的技术没什么难的（当然，盖高楼是很难的），但是你没地你怎么盖？我觉得云计算也一样，带宽的价格贵得就像土地的价格。其实云计算跟房地产一样，要占地、占机房、占带宽。如果能把中国所有的机房、机柜、带宽资源都买了，你就不用做云计算了，卖土地就够了——因为这些是有限的。最简单的例子，IP 地址是有限的。你有带宽、有机房，但是如果你没有 IP，这就不好玩了。尤其是你要提供 CDN 服务，这个就更明显，因为有多少物理节点直接决定你的 CDN 服务质量。

另一方面，正如前面所说的，运维是件很难的事，运维这个事并不是一般人能搞的事。没有足够的场景、经验和时间，这种能力很难出现。

从用户的角度来说呢，云计算是一种服务，你需要对用户企业内的解决方案要有很好的了解，这样才能提高很好的服务。能提供“好服务”的通常都是把自己真正当成用户公司。

这跟做汽车一样，底层做引擎、轮子、油箱、控制系统，给你弄一堆零件，上层可以拼装。PaaS 相当于给你一个很快可以打造成的汽车的工作台。而 SaaS 就是成品——两厢、三厢、卡车、轿车，最终用户要的是这个。后面什么 Xen、存储、分布式，跟我一毛钱关系没有，我就要知道汽车是安全的，性能好的，省油的，不会抛锚、耐用的，千万别速度快了或者坡度大了或是别的怎么样就失灵了。

卖汽车也是卖服务。造出汽车来，并不代表你搞定这个事了。如果没有公路、没有加油站、没有 4S 店、没有交通管理、规则等等，你要么用不了，要么就是乱七八糟。不能只让用户在那看着你的汽车好牛啊，但是用户不知道怎么用。所以说，云计算最终旁边必须要有一套服务设施，而这套服务设施也是今天被人低估的。

云计算有两个东西我觉得是被人低估的，一个是运维，一个是那堆服务。做服务的需要有生态环境，有人帮你做。所以做云计算要落地并不简单。

这跟 IBM 一样。IBM 有段时间也是快不行了，他们的 CEO 写了一本《谁说大象不能跳舞》，讲 IBM 的转型，从卖硬件的转成卖服务、解决方案，有流程、咨询，顺便卖硬件，带着一堆系统集成商一起玩。我给你解决方案，谁来实现呢，就是集成商帮你，然后顺便把硬件卖给你。一样。未来是什么样，历史上已经有了。你看，要干那么多事，而且还不是用人堆就可以堆出来的。这就是云计算的门槛。

总之，云计算是需要吃自己的狗食才能吃出来的，绝不是像手机上的 Apps 一样，你想一想、试一试就能搞出来的，你首先需要让自己有这样的场景，有这样的经历，你才可能会有这样的经验能力和。

## 云计算的市场细分

市场细分必然是市场来驱动的。市场变化太快，说不清楚，不过大的方向应该会是这样的：有类是需要玩计算密集型的（比如大数据计算、网络游戏），有类是需要玩 IO 密集型的（比如视频网站），有类就是为了建网站的（比如电子商务、门户网站、无线），有类是为了数

据安全和保密的（比如金融数据）。

从更高的层面来看，社会也需要分工。有的人卖土地，有的人卖房子，有的人装修，有的人是中介。我相信没人愿意把所有的赌注都押在一个地方。云计算也是一样。上面也说过，无论 IaaS、PaaS、SaaS，后面的 S 都是 service，本质上都是提供服务。所以，我认为，市场的细分本质上就是服务的细分。

看看历史我们知道，细分永远是跟着行业走的，也是跟着业务走的，所以，在业务层会出现更多的细分。

## 对阿里云产业细分的看法

政府云、金融云不太清楚，不过我很清楚电商云——就是我之前负责的聚石塔。聚石塔时间不长，2012年9月正式上线，去年是大发展的一年，作为垂直云解决的很好。天猫和淘宝做的都是下单前的东西，下单后，商家每天处理好几百单，需要做订单合并、筛选，有的商家规模不大但订单很多。海尔有 ERP，这些商家没有，但是每天也 1000 多单，如果没有信息化的系统，人肉是处理不了的，必然要有 ERP 系统处理订单。另外还要管理用户，给用户做营销、发展忠实用户。总之，都是卖东西以后的事情。咋办？

淘宝天猫给了一堆开放 API，你可以调我的 API 接入，在你那边有 ISV 帮你做一套东西远程访问淘宝 API，把订单拉过去，仓库进货了之后，通过 API 把库存改一下，就可以连起来了。天猫用户下单，到他的系统、他的仓库，他就发货了，仓库补完货，在他的系统里一改，自动就到天猫店了。这是电子信息化。

但是一到双十一就受不了：订单量太大。正好云平台出现了，再怎么样，阿里的运维能力也要比你商家的要强吧。你看，聚石塔卖的是服务，不是主机。另外是数据安全：商家的系统天天被黑客盯着，如果我们把用户信息都给商家，不是所有的商家的系统安全都做得很好，内部的人插个什么 U 盘，上面一堆木马，数据就被偷走了。偷走了之后，别人还说是阿里搞丢的，这当然不行。所以，我们又要开放，还要保证安全，聚石塔这个云平台就这样出来的：你来我这儿，我才开放给你，因为安全很重要。

保证性能和安全也是商家的利益诉求也在里面，商家也不希望用户数据被偷，他也希望双十一能抗住。

另外，很多商家自己不会做，所以要 ISV（第三方软件开发商）来做，所以这个是卖解决方案，跟 IBM 很相似。银行要一套系统，IBM 提供硬件和解决方案，系统集成商来帮银行写代码和集成系统。聚石塔也很像，聚石塔提供 API、ECS、数据库，第三方的 ISV 进来帮商家集成一个系统。这是很经典的也是很传统的 IBM 的玩法，只不过是玩在了云端。

你看，这也是做自己的长项做出来的细分市场。所以说，吃自己的狗食很重要。

## 对 PaaS 的看法

无论是 Google 的 GAE 还是新浪的 SAE 都是给个容器，给个容器的好处是不用管数据连接、CPU 什么，程序一传就能用，什么水平扩展都不用管。不爽的是，一个是在编程上限制太多：AppEngine 总会阉割很多系统相关的功能，比如 Java、PHP、Python 的很多系统调用都阉割了，因为如果给你这些系统调用，你就可以突破沙箱；另一个是有故障的时候：技术人员遇到问题都恨不得自己上去解决，想看看后面在忙啥，但是看不到，很无助，只能等你解决，就看你的人解决的好不好、快不快。所以如果 IaaS 没做好，运维、故障自动处理、迁移没做好，出了问题用户只能干瞪眼，PaaS 必然不好用。当然 IaaS 层也有这个问题，但是至

少你还可以登到机器上看一看，大不了重启一下。像 AWS，你重启一下就跑到别的物理机，问题也许就解决了。

其实，对于 PaaS 中间这层的确很尴尬。怎么解决？我觉得还是要依赖某种业务场景。单纯一个平台要阉割很多功能，搞得用户不舒服，还不如干脆一步到位，根据业务场景给一个编程框架。比如 SAE 可以就做微博 app，上来就调 API，数据库都 ready；或者微信如果做个 PaaS，上面只玩微信公众平台上的东西，也可以。我觉得 PaaS 层更贴合业务会更成功。给新浪微博做个插件，你去买个 VM、买数据库？这种时候很需要 PaaS。我觉得 PaaS 层要成功就要贴近业务场景。比如：腾讯的风铃系统（虽然不知道企业帐号看见是什么样的），就做无线建站，这样多好。干巴巴的 PaaS 有点高不成低不就。

## 对 SDN 的看法

SDN 其意图是想改变目前超级复杂的网络结构。意图是挺好的。想一想，如果以后我家的网络不用因为买个新的路由器都要重新设计一把，只要一次设置，然后对所有的路由器都通过，的确是挺方便的，这点对企业非常好。不过，不知道在操作上怎么做，也许会从企业内部开始这场革命，这个不得而知。

就像开车一样，机械式的方向盘和刹车油门系统这么多年都没什么变化，也提过很多更好更高科技的解决方案，但是传统还是这样延续下来了。所以，SDN 真不知道未来会怎么样。总之，一个老的事物到一个新的事物需要有一个过程，这个过程中会出现很多过渡产品或是过渡方案，如果没有这些过渡产品和方案，也就没法达到新的事物。未来是什么样，无法预知。

## 对私有云的看法

私有云跟公有云，都会存在。这跟人一样，私人生活和公众生活都会需要的。大公司有 1 万、2 万人，这么多数据，要存，需要一个很稳定的解决方案。要稳定可以买 IBM，但是贵。云计算出来说，我可以写三份，但他不想上公有云，我的数据放在别人那里，总感觉不好的，所以有了私有云做物理隔离，他觉得安全。

安全这个词对应两个英文，**security** 和 **safety**，其实 **security** 和 **safety** 不一样：云计算解决 **safety**，保证数据不丢；宁可数据丢也不让人看到，那是 **security**。比如私人照片我更愿意存家里，有一个小的云存储，所有设备同步，跟老家父母同步，这样比较好。放公网很恐怖。

一定会有公司不愿意上云的，比如金融方面的企业，他们觉得互联网不安全，他们要的更多的是安全。在公网上你的系统的安全攻防能力都要跟上，但如果物理不通的话就不用考虑的太复杂。企业内部私有云肯定有市场。你看，好些企业内部目前还被 EMC、IBM 所垄断着呢。计算机发展史就是廉价的东西取代昂贵的东西，所以私有云一定没问题，而降低私有云的运维复杂度、提供一个或多个方便的运维系统和工具就是重中之重。其中，SDN 之类的东西肯定会是其中一个很重要的一块。

另外，还是那句话，云就是服务，只要提供了好的服务，无论公有还是私有都是会有价值的。

感谢杨赛对本文的策划。

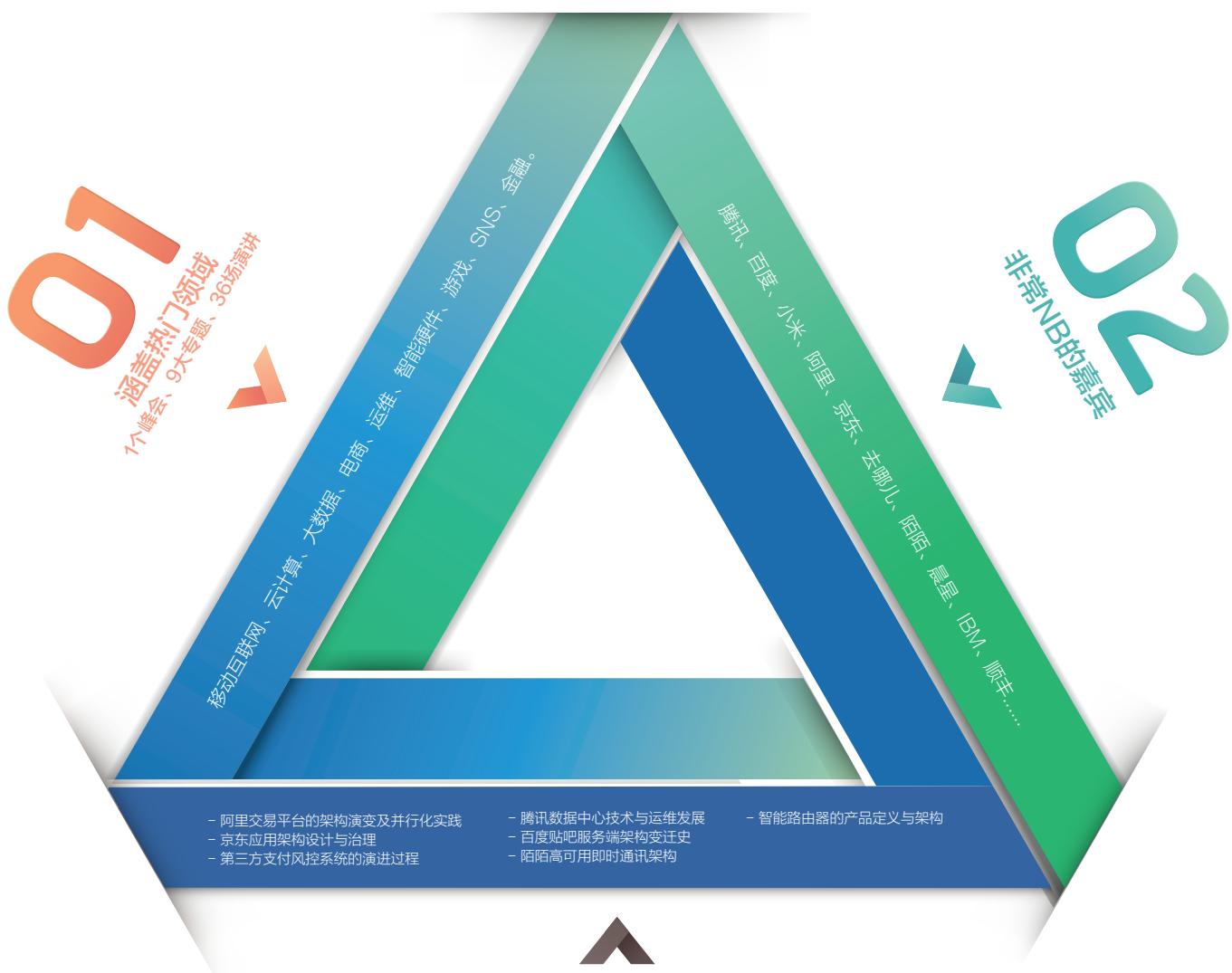
原文：<http://www.infoq.com/cn/articles/chenhao-on-cloud>

# ArchSummit

International Architect Summit

全球架构峰会 2014

2014.07.18-19 中国·深圳 万科国际会议中心



议题提交开放，折扣购票启动，详情查阅大会官网

咨询电话：010-89880682 会务咨询：arch@cn.infoq.com 大会官网：www.archsummit.com

# Javascript 高性能动画与页面渲染

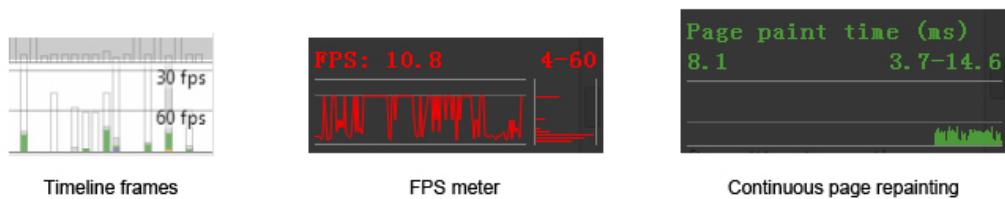
作者 李光毅

## No setTimeout, No setInterval

如果你不得不使用 `setTimeout` 或者 `setInterval` 来实现动画，那么原因只能是你需要精确的控制动画。但我认为至少在现在这个时间点，高级浏览器、甚至手机浏览器的普及程度足够让你有理由有条件在实现动画时使用更高效的方式。

## 什么是高效

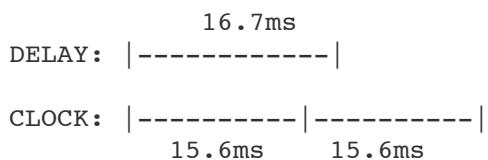
页面是每一帧变化都是系统绘制出来的 (GPU 或者 CPU)。但这种绘制又和 PC 游戏的绘制不同，它的最高绘制频率受限于显示器的刷新频率 (而非显卡)，所以大多数情况下最高的绘制频率只能是每秒 60 帧 (frame per second, 以下用 `fps` 简称)，对应于显示器的 60Hz。60fps 是一个最理想的状态，在日常对页面性能的测试中，60fps 也是一个重要的指标，the closer the better。在 Chrome 的调试工具中，有不少工具都是用于衡量当前帧数：



接下来的工作中，我们将会用到这些工具，来实时查看我们页面的性能。

**60fps** 是动力也是压力，因为它意味着我们只有 16.7 毫秒 (1000 / 60) 来绘制每一帧。如果使用 `setTimeout` 或者 `setInterval` (以下统称为 `timer`) 来控制绘制，问题就来了。

首先，`Timer` 计算延时的精确度不够。延时的计算依靠的是浏览器的内置时钟，而时钟的精确度又取决于时钟更新的频率 (`Timer resolution`)。IE8 及其之前的 IE 版本更新间隔为 15.6 毫秒。假设你设定的 `setTimeout` 延迟为 16.7ms，那么它要更新两个 15.6 毫秒才会触发延时。这也意味着无故延迟了  $15.6 \times 2 - 16.7 = 14.5$  毫秒。



所以即使你给 `setTimeout` 设定的延时为 0ms，它也不会立即触发。目前 Chrome 与 IE9+ 浏览器的更新频率都为 4ms (如果你使用的是笔记本电脑，并且在使用电池而非电源的模式下，为了节省资源，浏览器会将更新频率切换至于系统时间相同，也就意味着更新频率更低)。

退一步说，假使 `timer resolution` 能够达到 16.7ms，它还要面临一个异步队列的问题。因为异步的关系 `setTimeout` 中的回调函数并非立即执行，而是需要加入等待队列中。但

问题是，如果在等待延迟触发的过程中，有新的同步脚本需要执行，那么同步脚本不会排在 `timer` 的回调之后，而是立即执行，比如下面这段代码：

```
function runForSeconds(s) {
    var start = +new Date();
    while (start + s * 1000 > (+new Date())) {}
}

document.body.addEventListener("click", function () {
    runForSeconds(10);
}, false);

setTimeout(function () {
    console.log("Done!");
}, 1000 * 3);
```

如果在等待触发延迟的 3 秒过程中，有人点击了 `body`，那么回调还是准时在 3s 完成时触发吗？当然不能，它会等待 10s，同步函数总是优先于异步函数：

等待 3 秒延迟 | 1s | 2s | 3s | --->`console.log("Done!");`

经过 2 秒 | ----1s----|----2s----| --->`console.log("Done!");`

点击 `body` 后

以为是这样：|----1s----|----2s----|----3s----|--->`console.log("Done!");`--->|-----  
-----10s-----|

其实是这样：|----1s----|----2s----|-----10s-----|--->`console.log("Done!");`

John Resign 有三篇关于 `Timer` 性能与准确性的文章：[1.Accuracy of JavaScript Time](#)，[2.Analyzing Timer Performance](#)，[3.How JavaScript Timers Work](#)。从文章中可以看到 `Timer` 在不同平台浏览器与操作系统下的一些问题。

再退一步说，假设 `timer resolution` 能够达到 16.7ms，并且假设异步函数不会被延后，使用 `timer` 控制的动画还是有不尽如人意的地方。这也就是下一节要说的问题。

## 垂直同步问题

这里请再允许我引入另一个常量 60——屏幕的刷新率 60Hz。

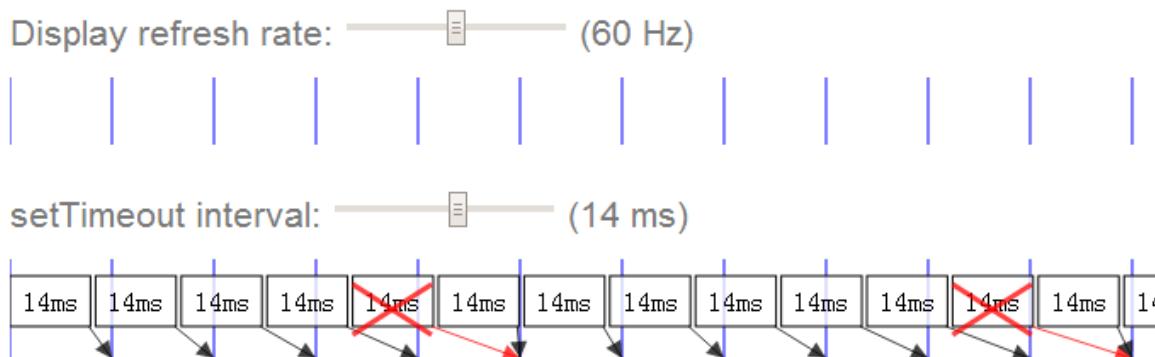
60Hz 和 60fps 有什么关系？没有任何关系。`fps` 代表 GPU 渲染画面的频率，`Hz` 代表显示器刷新屏幕的频率。一幅静态图片，你可以说这副图片的 `fps` 是 0 帧 / 秒，但绝对不能说此时屏幕的刷新率是 0Hz，也就是说刷新率不随图像内容的变化而变化。游戏也好浏览器也好，我们谈到掉帧，是指 GPU 渲染画面频率降低。比如跌落到 30fps 甚至 20fps，但因为视觉暂留原理，我们看到的画面仍然是运动和连贯的。

接上一节，我们假设每一次 `timer` 都不会有延时，也不会被同步函数干扰，甚至能把时间缩短至 16ms，那么会发生什么呢：



在 22 秒处发生了丢帧。

如果把延迟时间缩的更短，丢失的帧数也就更多：



实际情况会比以上想象的复杂的多。即使你能给出一个固定的延时，解决 60Hz 屏幕下丢帧问题，那么其他刷新频率的显示器应该怎么办，要知道不同设备、甚至相同设备在不同电池状态下的屏幕刷新率都不尽相同。

以上同时还忽略了屏幕刷新画面的时间成本。问题产生于 GPU 渲染画面的频率和屏幕刷新频率的不一致：如果 GPU 渲染出一帧画面的时间比显示器刷新一张画面的时间要短（更快），那么当显示器还没有刷新完一张图片时，GPU 渲染出的另一张图片已经送达并覆盖了前一张，导致屏幕上画面的撕裂，也就是上半部分是前一张图片，下半部分是后一张图片：



PC 游戏中解决这个问题的方法是开启垂直同步 (**v-sync**)，也就是让 GPU 妥协，GPU 渲染图片必须在屏幕两次刷新之间，且必须等待屏幕发出的垂直同步信号。但这样同样也是要付出代价的：降低了 GPU 的输出频率，也就降低了画面的帧数。以至于你在玩需要高帧数运行的游戏时（比如竞速、第一人称射击）感觉到“顿卡”，因为掉帧。

## requestAnimationFrame

在这里不谈 `requestAnimationFrame`( 以下简称 rAF ) 用法, 具体请参考 [MDN:Window.requestAnimationFrame\(\)](#)。我们来具体谈谈 rAF 所解决的问题。

从上一节我们可以总结出实现平滑动画的两个因素

1. 时机 (**Frame Timing**): 新的一帧准备好的时机
2. 成本 (**Frame Budget**): 渲染新的一帧需要多长的时间

这个 Native API 把我们从纠结于多久刷新的一次的困境中解救出来 ( 其实 rAF 也不关心距离下次屏幕刷新页面还需要多久 )。当我们调用这个函数的时候, 我们告诉它需要做两件事: 1. 我们需要新的一帧; 2. 当你渲染新的一帧时需要执行我传给你的回调函数

那么它解决了我们上面描述的第一个问题, 产生新的一帧的时机。

那么第二个问题呢。不, 它无能为力。比如可以对比下面两个页面:

### DEMO

### DEMO-FIXED

对比两个页面的源码, 你会发现只有一处不同:

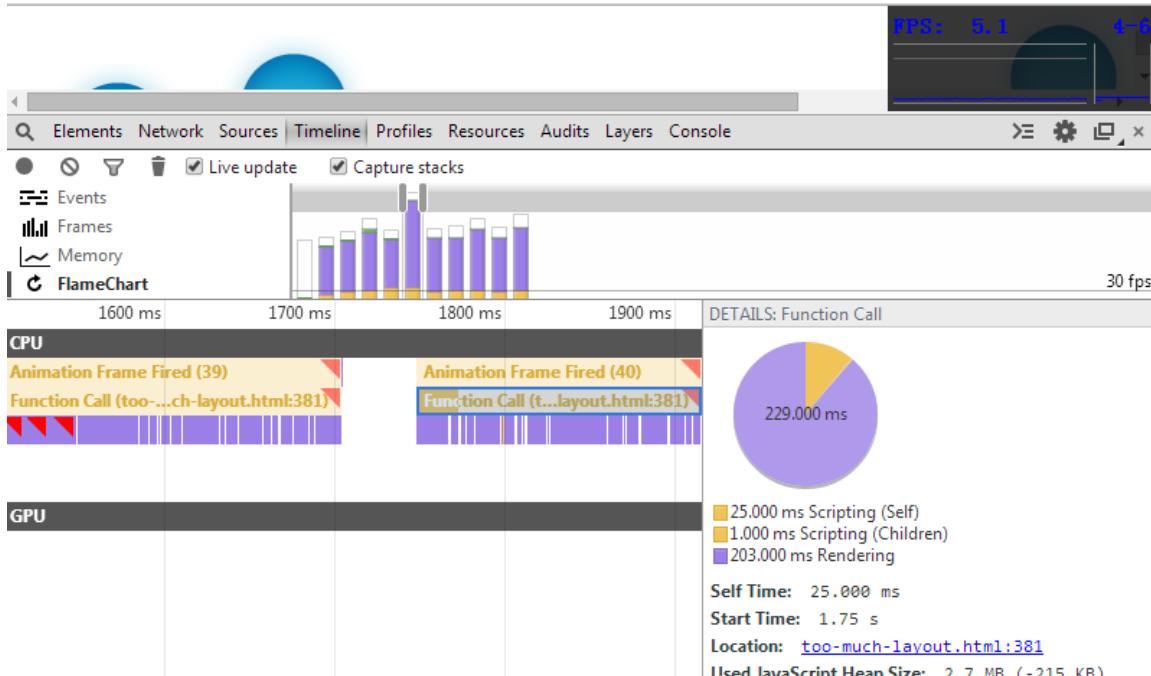
```
// animation loop
function update(timestamp) {
    for(var m = 0; m < movers.length; m++) {
        // DEMO 版本
        //movers[m].style.left = ((Math.sin(movers[m].offsetTop +
timestamp/1000)+1) * 500) + 'px';

        // FIXED 版本
        movers[m].style.left = ((Math.sin(m + timestamp/1000)+1) * 500) + 'px';
    }
    rAF(update);
};

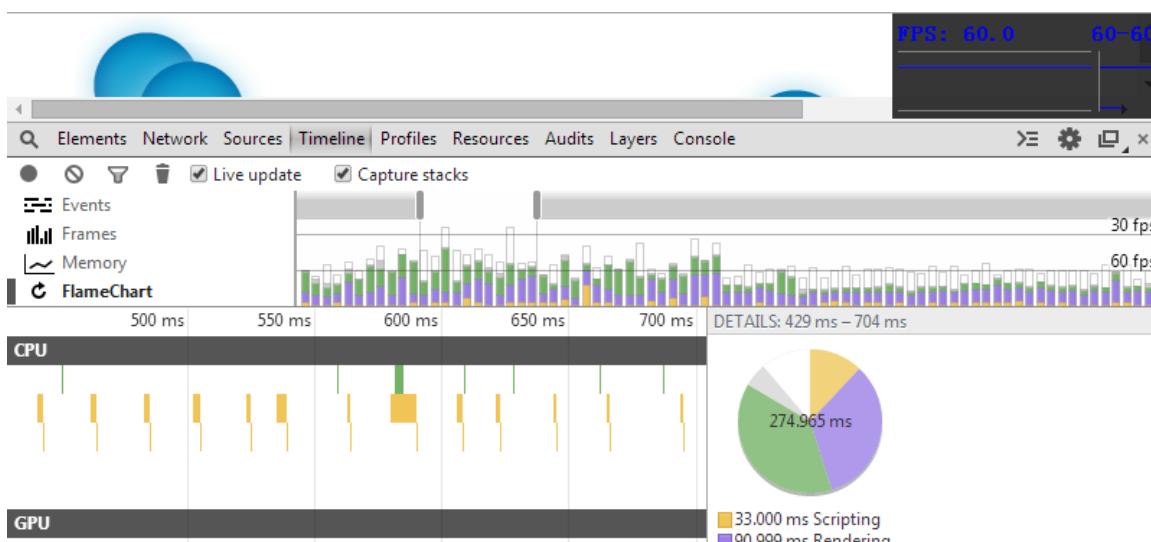
rAF(update);
```

DEMO 版本之所以慢的原因是, 在修改每一个物体的 `left` 值时, 会请求这个物体的 `offsetTop` 值。这是一个非常耗时的 `reflow` 操作 ( 具体还有哪些耗时的 `reflow` 操作可以参考这篇 : [How \(not\) to trigger a layout in WebKit](#) )。这一点从 Chrome 调试工具中可以看出来 ( 截图中的某些功能需要在 Chrome canary 版本中才可启用 )

未矫正的版本



可见大部分时间都花在了 `rendering` 上，而矫正之后的版本：



`rendering` 时间大大减少了

但如果你的回调函数耗时真的很严重，`rAF` 还是可以为你做一些什么的。比如当它发现无法维持 **60fps** 的频率时，它会把频率降低到 **30fps**，至少能够保持帧数的稳定，保持动画的连贯。

## 使用 `rAF` 推迟代码

没有什么是万能的，面对上面的情况，我们需要对代码进行组织和优化。

看看下面这样一段代码：

```

function jank(second) {
    var start = +new Date();
    while (start + second * 1000 > (+new Date())) {}
}

div.style.backgroundColor = "red";

// some long run task
jank(5);

div.style.backgroundColor = "blue";

```

无论在任何的浏览器中运行上面的代码，你都不会看到 `div` 变为红色，页面通常会在假死 5 秒，然后容器变为蓝色。这是因为浏览器的始终只有一个线程在运行（可以这么理解，因为 `js` 引擎与 `UI` 引擎互斥）。虽然你告诉浏览器此时 `div` 背景颜色应该为红色，但是它此时还在执行脚本，无法调用 `UI` 线程。

有了这个前提，我们接下来看这段代码：

```

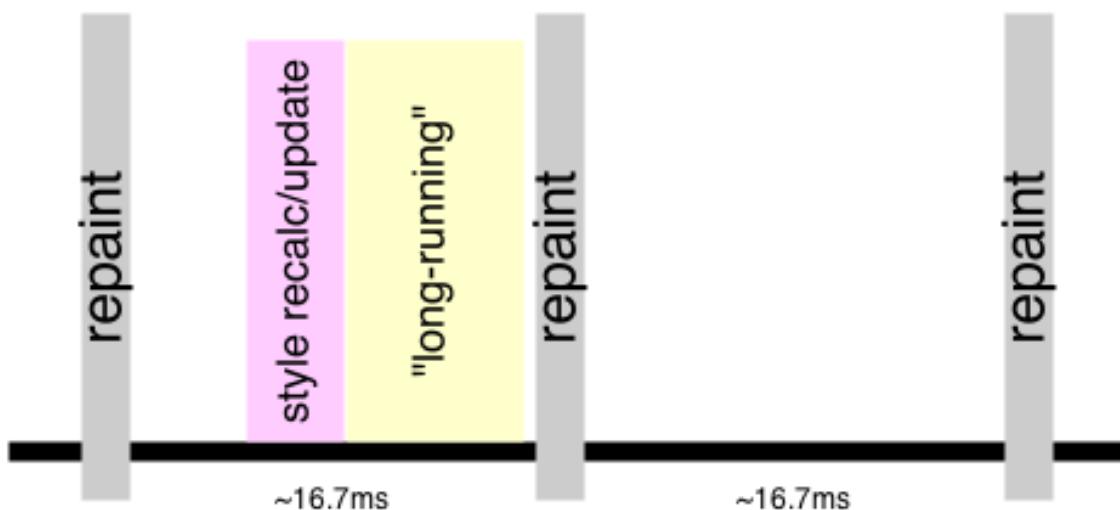
var div = document.getElementById("foo");

var currentWidth = div.innerWidth;
div.style.backgroundColor = "blue";

// do some "long running" task, like sorting data

```

这个时候我们不仅仅需要更新背景颜色，还需要获取容器的宽度。可以想象它的执行顺序如下：



当我们请求 `innerWidth` 一类的属性时，浏览器会以为我们马上需要，于是它会立即更新容器的样式（通常浏览器会攒着一批，等待时机一次性的 `repaint`，以便节省性能），并把计算的结果告诉我们。这通常是性能消耗量大的工作。

但如果我们将 `innerWidth` 放到 `jank` 函数中呢？

上面的代码有两处不足：

1. 更新背景颜色的代码过于提前，根据前一个例子，我们知道，即使在这里告知了浏览器我需要更新背景颜色，浏览器至少也要等到 js 运行完毕才能调用 UI 线程；
2. 假设后面部分的 long runing 代码会启动一些异步代码，比如 setTimeout 或者 Ajax 请求又或者 web-worker，那应该尽早为妙。

综上所述，如果我们不是那么迫切的需要知道 innerWidth，我们可以使用 rAF 推迟这部分代码的发生：

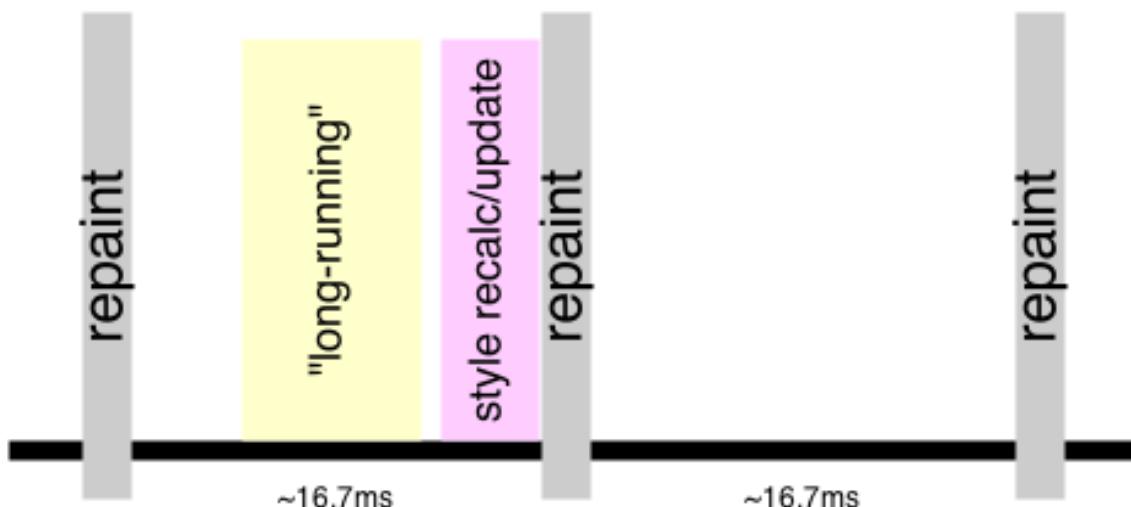
```
requestAnimationFrame(function(){
  var el = document.getElementById("foo");

  var currentWidth = el.innerWidth;
  el.style.backgroundColor = "blue";

  // ...
});

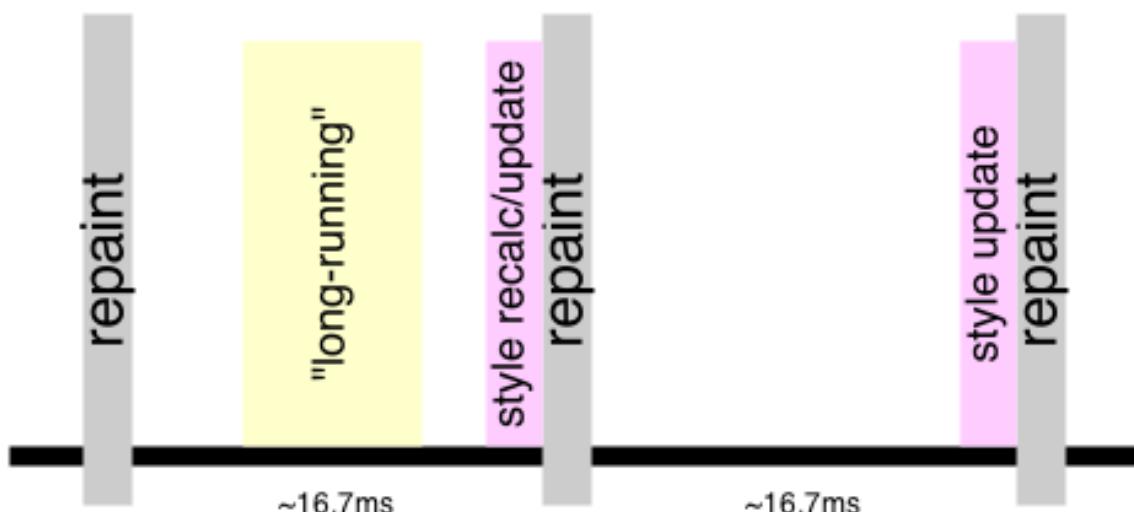
// do some "long running" task, like sorting data
```

可见即使我们在这里没有使用到动画，但仍然可以使用 rAF 优化我们的代码。执行的顺序会变成：



在这里 rAF 的用法变成了：把代码推迟到下一帧执行。

有时候我们需要把代码推迟的更远，比如这个样子：



再比如我们想要一个效果分两步执行：1. div 的 display 变为 block；2. div 的 top 值缩短移动到某处。如果这两项操作都放入同一帧中的话，浏览器会同时把这两项更改应用于容器，在同一帧内。于是我们需要两帧把这两项操作区分开来：

```
requestAnimationFrame(function(){
  el.style.display = "block";
  requestAnimationFrame(function(){
    // fire off a CSS transition on its `top` property
    el.style.top = "300px";
  });
});
```

这样的写法好像有些不太讲究，[Kyle Simpson](#)有一个开源项目 [h5ive](#)，它把上面的用法封装了起来，并且提供了 API。实现起来非常简单，摘一段代码瞧瞧：

```
function qID(){
  var id;
  do {
    id = Math.floor(Math.random() * 1E9);
  } while (id in q_ids);
  return id;
}

function queue(cb) {
  var qid = qID();

  q_ids[qid] = rAF(function(){
    delete q_ids[qid];
    cb.apply(publicAPI,arguments);
  });
  return qid;
}

function queueAfter(cb) {
  var qid;

  qid = queue(function(){
    // do our own rAF call here because we want to re-use the same `qid` for
    both frames
    q_ids[qid] = rAF(function(){
```

```

        delete q_ids[qid];
        cb.apply(publicAPI, arguments);
    });
});

return qid;
}

```

使用方法：

```

// 插入下一帧
id1 = aFrame.queue(function(){
    text = document.createTextNode("##");
    body.appendChild(text);
});

// 插入下下一帧
id2 = aFrame.queueAfter(function(){
    text = document.createTextNode("!!");
    body.appendChild(text);
});

```

## 使用 rAF 解耦代码

先从一个 2011 年 `twitter` 遇到的 bug 说起。

当时 `twitter` 加入了一个新功能：“无限滚动”。也就是当页面滚至底部的时候，去加载更多的 `twitter`：

```

$(window).bind('scroll', function () {
    if (nearBottomOfPage()) {
        // load more tweets ...
    }
});

```

但是在这个功能上线之后，发现了一个严重的 bug：经过几次滚动到底部之后，滚动就会变得奇慢无比。

经过排查发现，原来是一条语句引起的：`$details.find(".details-pane-outer")`；

这还不是真正的罪魁祸首，真正的原因是因为他们将使用的 `jQuery` 类库从 1.4.2 升级到了 1.4.4 版。而这 `jQuery` 其中一个重要的升级是把 `Sizzle` 的上下文选择器全部替换为了 `querySelectorAll`。但是这个接口原实现使用的是 `getElementsByClassName`。虽然 `querySelectorAll` 在大部分情况下性能还是不错的。但在通过 `Class` 名称选择元素这一项是占了下风。有两个对比测试可以看出来：1. `querySelectorAll v getElementsByClassName` 2. `jQuery Simple Selector`

通过这个 bug，`John Resig` 给出了一条（实际上是两条，但是今天只取与我们话题有关的）非常重要的建议

`It's a very, very, bad idea to attach handlers to the window scroll event.`

他想表达的意思是，像 `scroll`, `resize` 这一类的事件会非常频繁的触发，如果把太多的代码放进这一类的回调函数中，会延迟页面的滚动，甚至造成无法响应。所以应该把这一类

代码分离出来，放在一个 `timer` 中，有间隔的去检查是否滚动，再做适当的处理。比如如下代码：

```
var didScroll = false;

$(window).scroll(function() {
    didScroll = true;
});

setInterval(function() {
    if ( didScroll ) {
        didScroll = false;
        // Check your page position and then
        // Load in more results
    }
}, 250)
```

这样的作法类似于 **Nicholas** 将需要长时间运算的循环分解为“片”来进行运算：

```
// 具体可以参考他写的《Javascript 高级程序设计》
// 也可以参考他的这篇博客: http://www.nczonline.net/blog/2009/01/13/speed-up-your-javascript-part-1/
function chunk(array, process, context){
    var items = array.concat(); //clone the array
    setTimeout(function(){
        var item = items.shift();
        process.call(context, item);

        if (items.length > 0){
            setTimeout(arguments.callee, 100);
        }
    }, 100);
}
```

原理其实是一样的，为了优化性能、为了防止浏览器假死，将需要长时间运行的代码分解为小段执行，能够使浏览器有时间响应其他的请求。

回到 `rAF` 上来，其实 `rAF` 也可以完成相同的功能。比如最初的滚动代码是这样：

```
function onScroll() {
    update();
}

function update() {

    // assume domElements has been declared
    for(var i = 0; i < domElements.length; i++) {

        // read offset of DOM elements
        // to determine visibility - a reflow

        // then apply some CSS classes
        // to the visible items - a repaint

    }
}

window.addEventListener('scroll', onScroll, false);
```

这是很典型的反例：每一次滚动都需要遍历所有元素，而且每一次遍历都会引起 `reflow` 和 `repaint`。接下来我们要做的事情就是把这些费时的代码从 `update` 中解耦出来。

首先我们仍然需要给 `scroll` 事件添加回调函数，用于记录滚动的情况，以方便其他函数的查询：

```
var latestKnownScrollY = 0;

function onScroll() {
    latestKnownScrollY = window.scrollY;
}
```

接下来把分离出来的 `repaint` 或者 `reflow` 操作全部放入一个 `update` 函数中，并且使用 `rAF` 进行调用：

```
function update() {
    requestAnimationFrame(update);

    var currentScrollY = latestKnownScrollY;

    // read offset of DOM elements
    // and compare to the currentScrollY value
    // then apply some CSS classes
    // to the visible items
}

// kick off
requestAnimationFrame(update);
```

其实解耦的目的已经达到了，但还需要做一些优化，比如不能让 `update` 无限执行下去，需要设标志位来控制它的执行：

```
var latestKnownScrollY = 0,
    ticking = false;

function onScroll() {
    latestKnownScrollY = window.scrollY;
    requestTick();
}

function requestTick() {
    if(!ticking) {
        requestAnimationFrame(update);
    }
    ticking = true;
}
```

并且我们始终只需要一个 `rAF` 实例的存在，也不允许无限次的 `update` 下去，于是我们还需要一个出口：

```

function update() {
    // reset the tick so we can
    // capture the next onScroll
    ticking = false;

    var currentScrollY = latestKnownScrollY;

    // read offset of DOM elements
    // and compare to the currentScrollY value
    // then apply some CSS classes
    // to the visible items
}

// kick off - no longer needed! Woo.
// update();

```

## 理解 Layer

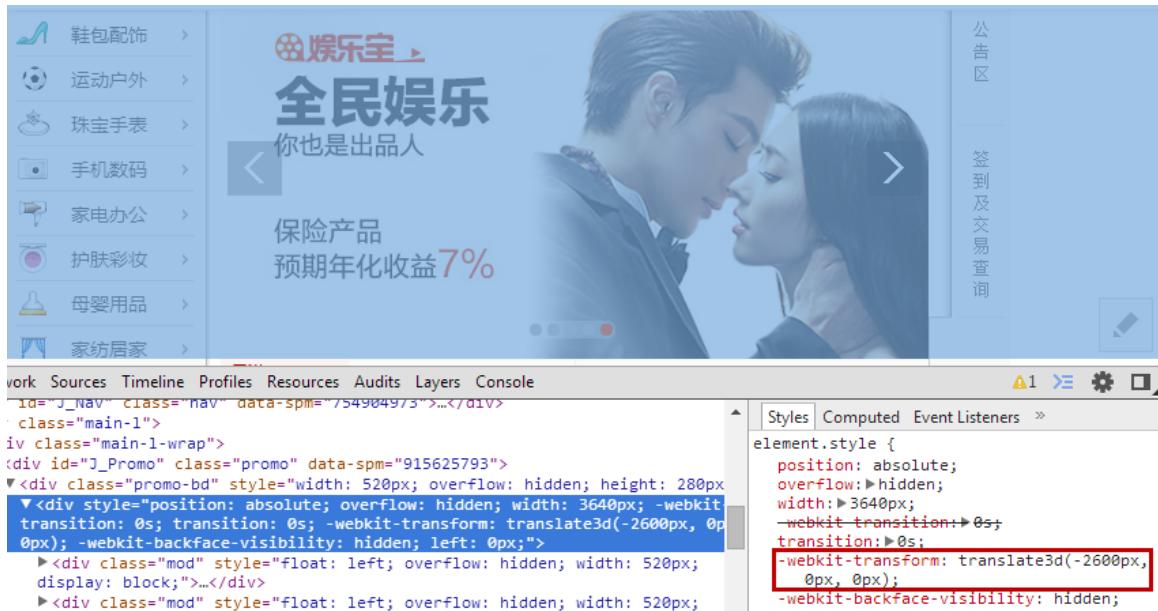
Kyle Simpson 说：

Rule of thumb: don't do in JS what you can do in CSS.

如以上所说，即使使用 rAF，还是会有诸多的不便。我们还有一个选择是使用 **css** 动画：虽然浏览器中 UI 线程与 js 线程是互斥，但这一点对 **css** 动画不成立。

在这里不聊 **css** 动画的用法。**css** 动画运用的是什么原理来提升浏览器性能的。

首先我们看看淘宝首页的焦点图：



我想提出一个问题，为什么明明可以使用 `translate 2d` 去实现的动画，它要用 `3d` 去实现呢？

我不是淘宝的员工，但我的第一猜测这么做的原因是使用 `translate3d hack`。简单来说如果你给一个元素添加上了 `-webkit-transform: translateZ(0);` 或者 `-webkit-transform: translate3d(0,0,0);` 属性，那么你就等于告诉了浏览器用 GPU 来渲染该层，与一般的 CPU 渲染相比，提升了速度和性能。（我很确定这么做会在

Chrome 中启用了硬件加速，但在其他平台不做保证。就我得到的资料而言，在大多数浏览器比如 Firefox、Safari 也是适用的 )。

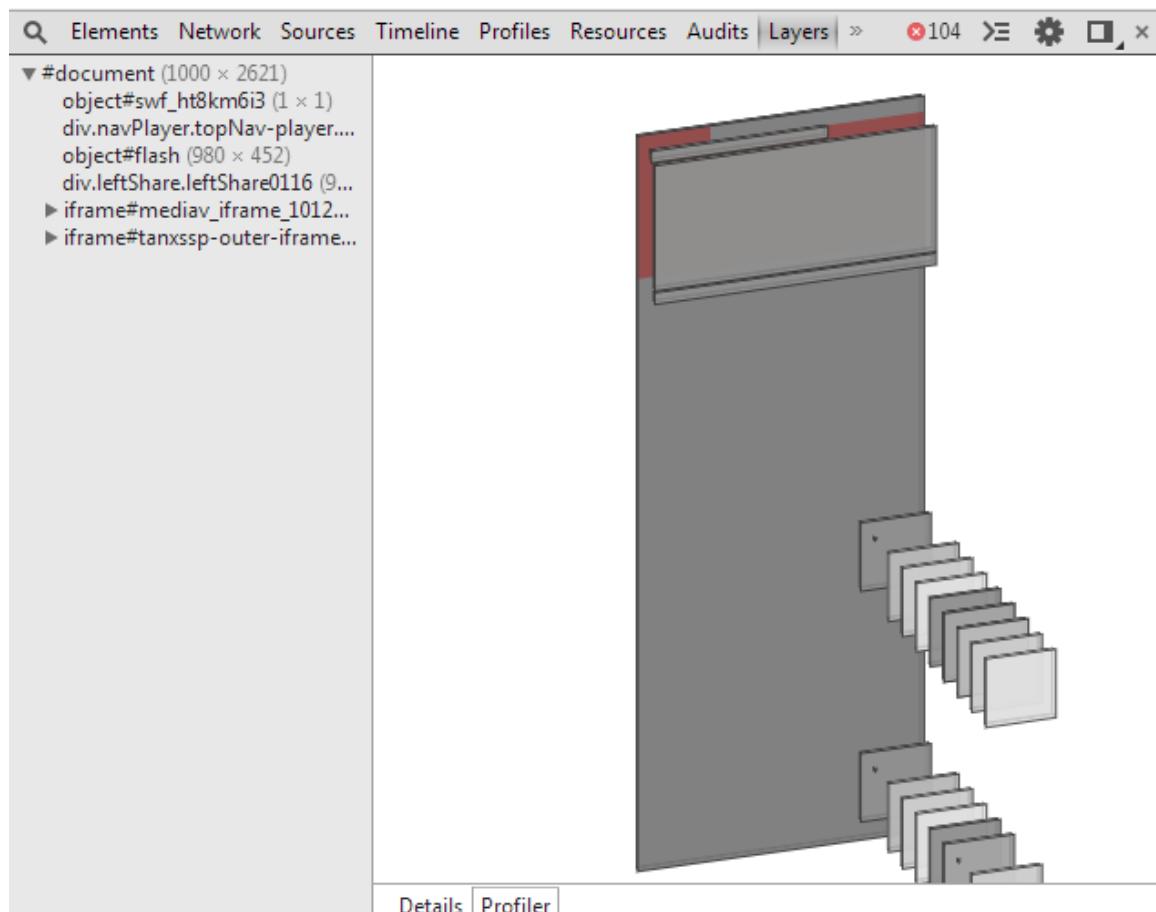
但这样的说法其实并不准确，至少在现在的 Chrome 版本中这算不上一个 **hack**。因为默认渲染所有的网页时都会经过 GPU。那么这么做还有必要吗？有。在理解原理之前，你必须先了解一个层 (Layer) 的概念。

html 在浏览器中会被转化为 DOM 树，DOM 树的每一个节点都会转化为 RenderObject，多个 RenderObject 可能又会对应一个或多个 RenderLayer。浏览器渲染的流程如下：

1. 获取 DOM 并将其分割为多个层 (RenderLayer)
2. 将每个层栅格化，并独立的绘制进位图中
3. 将这些位图作为纹理上传至 GPU
4. 复合多个层来生成最终的屏幕图像 (终极 layer)。

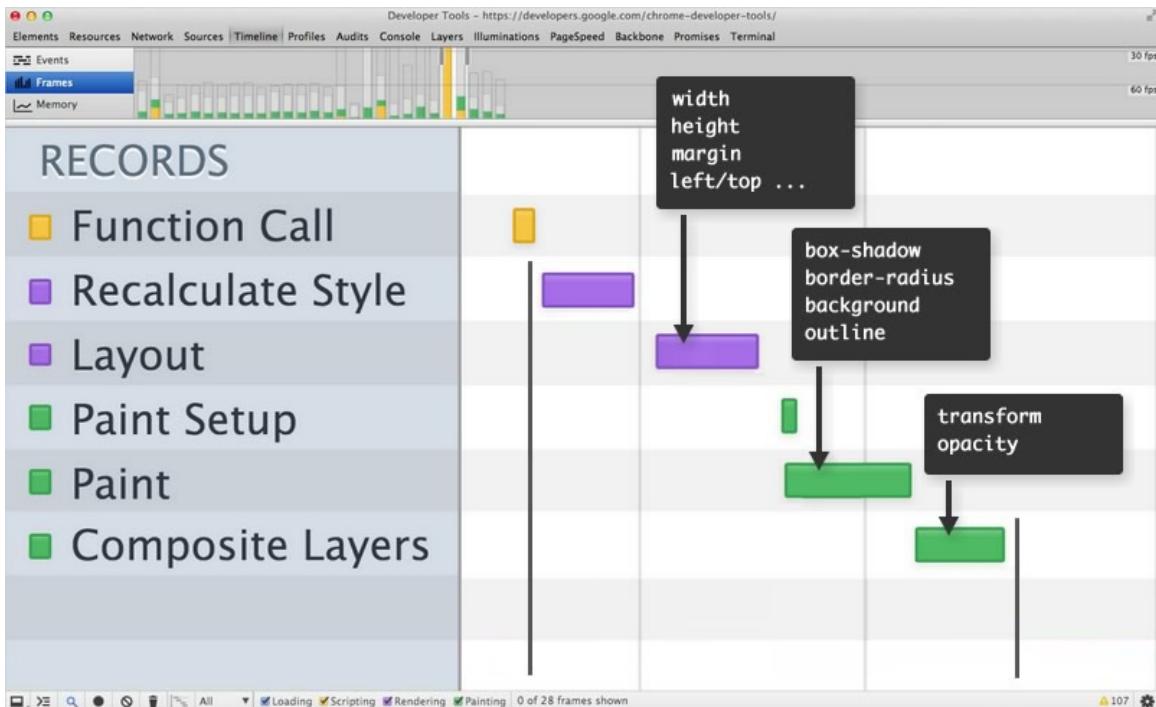
这和游戏中的 3D 渲染类似，虽然我们看到的是一个立体的人物，但这个人物的皮肤是由不同的图片“贴”和“拼”上去的。网页比此还多了一个步骤，虽然最终的网页是由多个位图层合成的，但我们看到的只是一个复印版，最终只有一个层。当然有的层是无法拼合的，比如 flash。以爱奇艺的一个播放页 ([http://www.iqiyi.com/v\\_19rrgyhg0s.html](http://www.iqiyi.com/v_19rrgyhg0s.html)) 为例，我们可以利用 Chrome 的 Layer 面板（默认不启用，需要手动开启）查看页面上所有的层：

我们可以看到页面上由如下层组成：



OK，那么问题来了。

假设我现在想改变一个容器的样式（可以看做动画的一个步骤），并且是一种最糟糕的情况，改变它的长和宽——为什么说改变长和宽是最糟糕的情况呢，通常改变一个物体的样式需要以下四个步骤：



任何属性的改变都导致浏览器重新计算容器的样式，比如你改变的是容器的尺寸或者位置 (**reflow**)，那么首先影响的就是容器的尺寸和位置（也影响了与它相关的父节点自己点相邻节点的位置等），接下来浏览器还需要对容器重新绘制 (**repaint**)；但如果你改变的只是容器的背景颜色等无关容器尺寸的属性，那么便省去了第一步计算位置的时间。也就是说如果改变属性在瀑布图中开始的越早（越往上），那么影响就越大，效率就越低。**reflow** 和 **repaint** 会导致所有受影响节点所在 **layer** 的位图重绘，反复执行上面的过程，导致效率降低。

为了把代价降到最低，当然最好只留下 **compositing layer** 这一个步骤即可。假设当我们改变一个容器的样式时，影响的只是它自己，并且还无需重绘，直接通过在 **GPU** 中改变纹理的属性来改变样式，岂不是更好？这当然是可以实现的，前提是你要有自己的 **layer**

这也是上面硬件加速 **hack** 的原理，也是 **css** 动画的原理——给元素创建自己 **layer**，而非与页面上大部分的元素共用 **layer**。

什么样的元素才能创建自己 **layer** 呢？在 **Chrome** 中至少要符合以下条件之一：

- Layer has 3D or perspective transform CSS properties (有 3D 元素的属性)
- Layer is used by <video> element using accelerated video decoding (video 标签并使用加速视频解码)
- Layer is used by a <canvas> element with a 3D context or accelerated 2D context (canvas 元素并启用 3D)

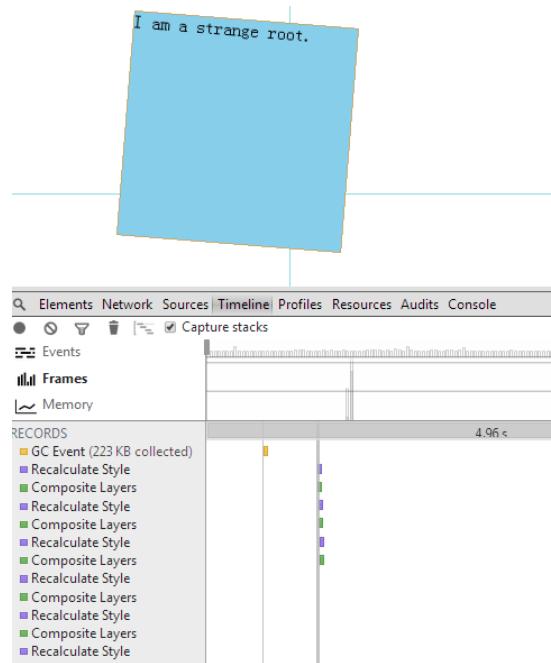
- Layer is used for a composited plugin( 插件, 比如 flash)
- Layer uses a CSS animation for its opacity or uses an animated webkit transform(CSS 动画 )
- Layer uses accelerated CSS filters(CSS 滤镜 )
- Layer with a composited descendant has information that needs to be in the composited layer tree, such as a clip or reflection( 有一个后代元素是独立的 layer)
- Layer has a sibling with a lower z-index which has a compositing layer (in other words the layer is rendered on top of a composited layer)( 元素的相邻元素是独立 layer)

很明显刚刚我们看到的播放页中的 `flash` 和开启了 `translate3d` 样式的焦点图符合上面的条件。

同时你也可以勾选 Chrome 开发工具中的 rendering 选显卡下的 Show composited layer borders 选项。页面上的 layer 便会加以边框区别开来。为了验证我们的想法，看下面这样一段代码：

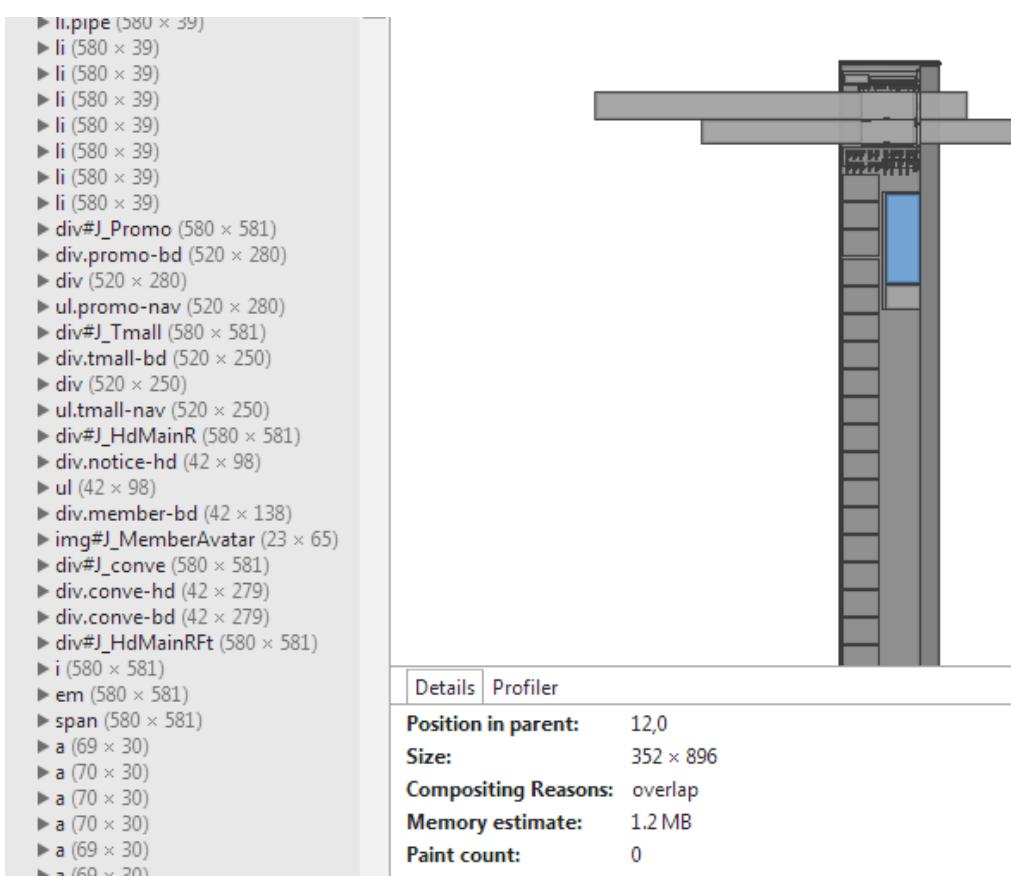
```
<html>
<head>
  <style type="text/css">
    div {
      -webkit-animation-duration: 5s;
      -webkit-animation-name: slide;
      -webkit-animation-iteration-count: infinite;
      -webkit-animation-direction: alternate;
      width: 200px;
      height: 200px;
      margin: 100px;
      background-color: skyblue;
    }
    @-webkit-keyframes slide {
      from {
        -webkit-transform: rotate(0deg);
      }
      to {
        -webkit-transform: rotate(120deg);
      }
    }
  </style>
</head>
<body>
  <div id="foo">I am a strange root.</div>
</body>
</html>
```

运行时的 `timeline` 截图如下：



可见元素有自己的 layer，并且在动画的过程中没有触发 reflow 和 repaint。

最后再看看淘宝首页，不仅仅只有焦点图才拥有了独立的 layer：



但太多的 layer 也未必是一件好事情，有兴趣的同学可以看一看这篇文章：[Jank Busting Apple's Home Page](#)。看一看在苹果首页太多 layer 时出现的问题。

参考文章：

[High Performance Animations](#)

[Accelerated Rendering in Chrome](#)

[Jank Busting for Better Rendering Performance](#)

[Leaner, Meaner, Faster Animations with requestAnimationFrame](#)

[Optimizing Visual Updates](#)

[GPU Accelerated Compositing in Chrome](#)

感谢王保平对本文的审校。

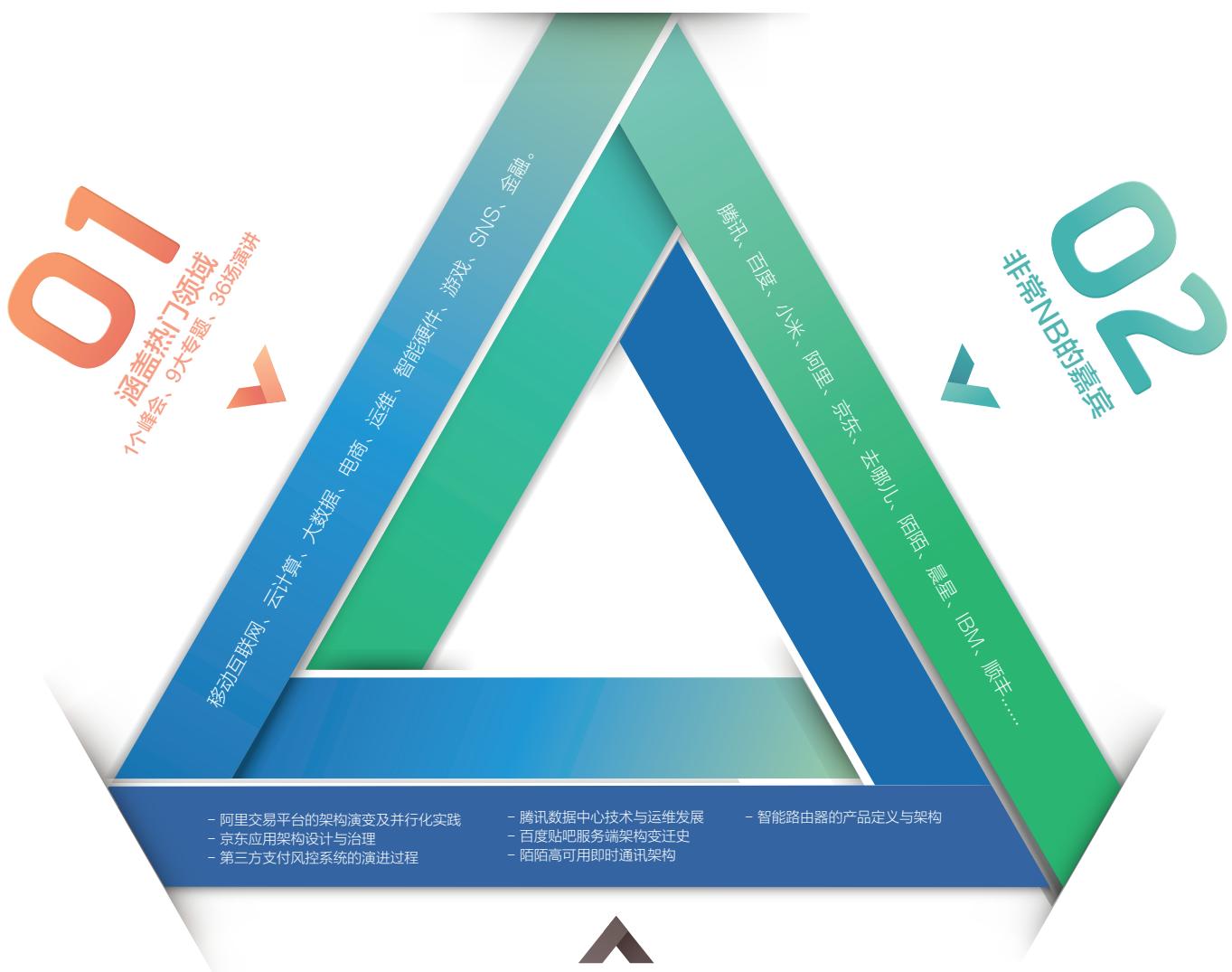
原文：<http://www.infoq.com/cn/articles/javascript-high-performance-animation-and-page-rendering>

# ArchSummit

International Architect Summit

全球架构峰会 2014

2014.07.18-19 中国·深圳 万科国际会议中心



议题提交开放，折扣购票启动，详情查阅大会官网

咨询电话：010-89880682 会务咨询：arch@cn.infoq.com 大会官网：www.archsummit.com

# 对话 Facebook 人工智能实验室主任、深度学习专家 Yann LeCun

作者 [Gregory Piatetsky](#)，译者 张天雷



[Yann LeCun](#)（燕乐存），Facebook 人工智能实验室主任，NYU 数据科学中心创始人，计算机科学、神经科学、电子电气科学教授。他 1983 年在 ESIEE 获得电气工程学位，1987 年在 UPMC 获得计算机博士学位。在多伦多大学做了一段时间博士后，于 1988 年加入位于新泽西州的 AT&T 贝尔实验室。1996 年他成为图像处理研究部的主任，2003 年，在普林斯顿 NEC 研究院经历短暂的 Fellow 生活以后，加入 NYU。2013 年，他被 Facebook 聘请为人工智能实验室主任，同时仍在 NYU 兼职。

他目前的研究兴趣在于：机器学习，计算机认知，移动机器人以及计算神经学。在这些领域他发表了 180 余篇论文和图书，涉及主题有神经网络、手写体识别、图像处理和压缩以及计算机认知的专用电路和架构。他在贝尔实验室研发的字符识别技术，被全世界多家银行用于识别支票，早在 2000 年左右，该程序识别了全美 10%-20% 的支票。他发明的图片压缩技术 DjVu，被数百家网站和出版商采纳，拥有上百万用户。他研发的一个识别方法，卷积网络，是 AT&T、Google、微软、NEC、IBM、百度以及 Facebook 等公司在文档识别，人机交互，图片标注、语音识别和视频分析等等技术的奠基石。

LeCun 教授是 IJCV、PAMI 和 IEEE Trans 的审稿人。CVPR06 的程序主席、ICLR2013 和 2014 的主席。他是 IPAM (Institute for Pure and Applied Mathematics) 的顾问。他是 2014 年 IEEE 神经网络领军人物奖获得者。

本文的采访者是另一位大牛 [Gregory Piatetsky](#)，KDD 会议创始人，是 1989, 1991 和 1993 年 KDD 的主席，SIGKDD 第一个服务奖章获得者，KDnuggets 网站和周刊的维护者。

本文主要内容有，是什么给深度学习带来了今日如此令世人瞩目的成绩，Yann Lecun 和 Vapnik 关于神经网络和核函数（支持向量机）的争论，以及 Facebook 理想中的 AI 是什么样子的。

以下为采访原文：

问：人工神经网络的研究已经有五十多年了，但是最近才有非常令人瞩目的结果，在诸如语音和图像识别这些比较难的问题上，是什么因素让深度学习网络胜出了呢？数据？算法？硬件？

答：虽然大部分人的感觉是人工神经网络最近几年才迅速崛起，但实际上上个世纪八十年代以后，就有很多成功的应用了。深度学习指的是，任何可以训练多于两到三个非线性隐含层模型的学习算法。大概是 2003 年，Geoff Hinton，Yoshua Bengio 和我策划并鼓动机器学习社区将兴趣放在表征学习这个问题上（和简单的分类器学习不同）。直到 2006-2007 年左右才有了点味道，主要是通过无监督学习的结果（或者说是无监督预训练，伴随监督算法的微调），这部分工作是 Geoff Hinton，Yoshua Bengio，Andrew Ng 和我共同进行的。

但是大多数最近那些有效果的深度学习，用得还是纯监督学习加上后向传播算法，跟上个世纪八十年代末九十年代初的神经网络没太大区别。

区别在于，我们现在可以在速度很快的 GPU 上跑非常大非常深层的网络（比如有时候有十亿连接，12 层），而且还可以用大规模数据集里面的上百万的样本来训练。过去我们还有一些训练技巧，比如有个正则化的方法叫做 [dropout](#)，还有克服神经元的非线性问题，以及不同类型的空间池化（spatial pooling）等等。

很多成功的应用，尤其是在图像识别上，都采用的是卷积神经网络（ConvNet），是我上个世纪八九十年代在贝尔实验室开发出来的。后来九十年代中期，贝尔实验室商业化了一批基于卷积神经网络的系统，用于识别银行支票（印刷版和手写版均可识别）。

经过了一段时间，其中一个系统识别了全美大概 10% 到 20% 的支票。最近五年，对于卷积神经网络的兴趣又卷土重来了，很多漂亮的工作，我的研究小组有参与，以及 Geoff Hinton，Andrew Ng 和 Yoshua Bengio，还有瑞士 IDSJ 的 AJargen Schmidhuber，以及加州的 NEC。卷积神经网络现在被 Google，Facebook，IBM，百度，NEC 以及其他互联网公司广泛使用，来进行图像和语音识别。（Gregory Piatetsky 注：Yann Lecun 教授的一个学生，最近赢得了 Kaggle 上猫狗识别的比赛，用的就是卷积神经网络，准确度 98.9%。）

问：深度学习可不是一个容易用的方法，你能给大家推荐一些工具和教程么？大家都挺想从自己的数据上跑跑深度学习。

答：基本上工具有两个推荐：

[Torch7](#)

[Theano + Pylearn2](#)

他们的设计哲学不尽相同，各有千秋。Torch7 是 LuaJIT 语言的一个扩展，提供了多维数组和数值计算库。它还包括一个面向对象的深度学习开发包，可用于计算机视觉等研究。Torch7 的主要优点在于 LuaJIT 非常快，使用起来也非常灵活（它是流行脚本语言 Lua 的

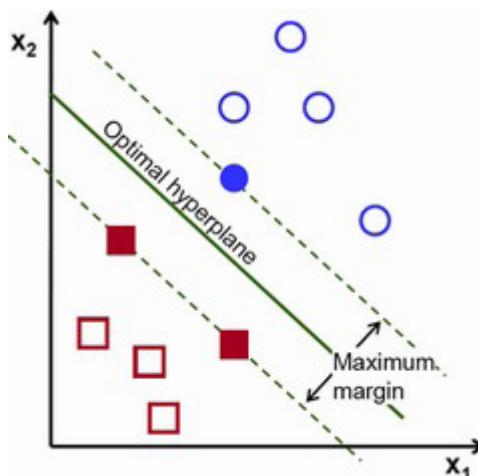
编译版本)。

Theano 加上 Pylearn 先天就有 Python 语言带来的优势 (Python 是广泛应用的脚本语言, 很多领域都有对应的开发库), 劣势也是应为用 Python, 速度慢。

问: 咱俩很久以前在 **KXEN** 的科学咨询会议上见过, 当时 **Vapnik** 的概率学习理论和支持向量机 (**SVM**) 是比较主流的。深度学习和支持向量机 / 概率学习理论有什么关联?

答: 1990 年前后, 我和 Vapnik 在贝尔实验室共事, 归属于 Larry Jackel 的自适应系统研究部, 我俩办公室离得很近。卷积神经网络, 支持向量机, 正切距离以及其他后来有影响的方法都是在这发明出来的, 问世时间也相差无几。1995 年 AT&T 拆分朗讯以后, 我成了这个部门的领导, 部门后来改成了 AT&T 实验室的图像处理研究部。部门当时的机器学习专家有 Yoshua Bengio, Leon Bottou, Patrick Haffner 以及 Vladimir Vapnik, 还有几个访问学者以及实习生。

我和 Vapnik 经常讨论深度网络和核函数的相对优缺点。基本来讲, 我一直对于解决特征学习和表征学习感兴趣。我对核方法兴趣一般, 因为它们不能解决我的问题。老实说, 支持向量机作为通用分类方法来讲, 是非常不错的。但是话说回来, 它们也只不过是简单的两层模型, 第一层是用核函数来计算输入数据和支持向量之间相似度的单元集合。第二层则是线性组合了这些相似度。



第一层就是用最简单的无监督模型训练的, 即将训练数据作为原型单元存储起来。基本上来说, 调节核函数的平滑性, 产生了两种简单的分类方法: 线性分类和模板匹配。大概十年前, 由于评价核方法是一种包装美化过的模板匹配, 我惹上了麻烦。Vapnik, 站在我对立面, 他描述支持向量机有非常清晰的扩展控制能力。“窄”核函数所产生的支持向量机, 通常在训练数据上表现非常好, 但是其普适性则由核函数的宽度以及对偶系数决定。Vapnik 对自己得出的结果非常自信。他担心神经网络没有类似这样简单的方式来进行扩展控制 (虽然神经网络根本没有普适性的限制, 因为它们都是无限的 VC 维)。

我反驳了他, 相比用有限计算能力来计算高复杂度函数这种能力, 扩展控制只能排第二。图像识别的时候, 移位、缩放、旋转、光线条件以及背景噪声等等问题, 会导致以像素做特征的核函数非常低效。但是对于深度架构比如卷积网络来说却是小菜一碟。

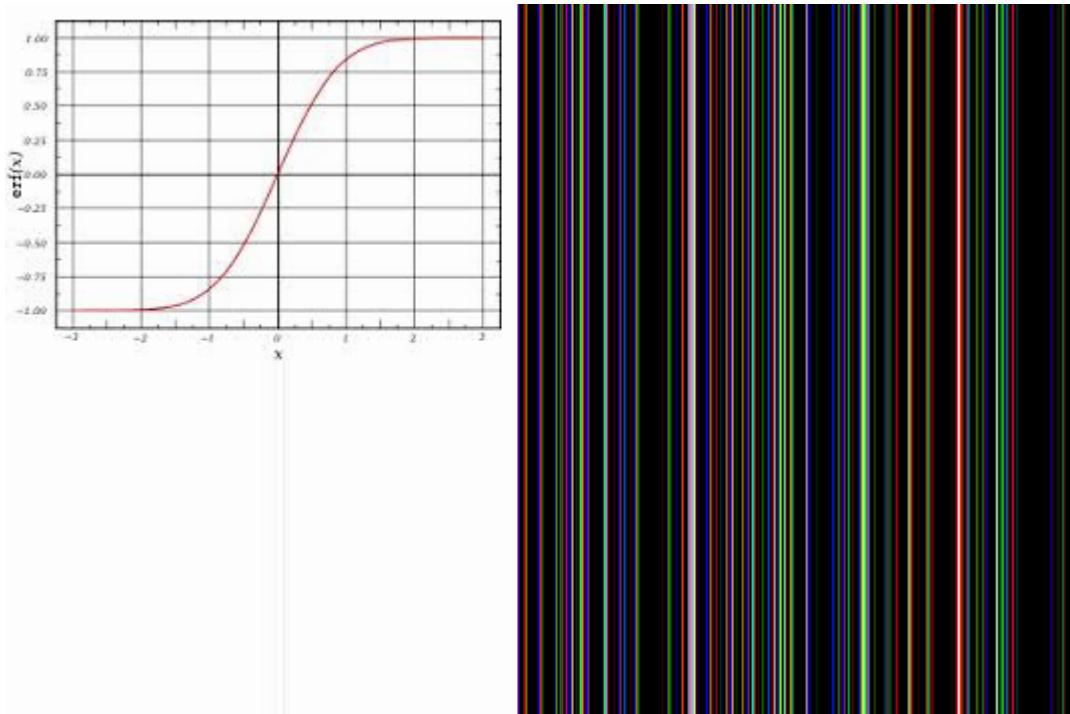
问: 祝贺你成为 **Facebook** 人工智能实验室的主任。你能给讲讲未来几年 **Facebook** 在人工智能和机器学习上能有什么产出么?

答: 非常谢谢你, 这个职位是个非常难得的机会。基本上来讲, Facebook 的主要目标是

让人与人更好的沟通。但是当今的人们被来自朋友、新闻、网站等等信息来源狂哄乱炸。Facebook 帮助人们来在信息洪流中找到正确的方向。这就需要 Facebook 能知道人们对什么感兴趣，什么是吸引人的，什么让人快乐，什么让人们学到新东西。这些知识，只有人工智能可以提供。人工智能的进展，将让我们理解各种内容，比如文字，图片，视频，语音，声音，音乐等等。

问：长期来看，你觉得人工智能会变成什么样？我们会不会达到 Ray Kurzweil 所谓的奇点？

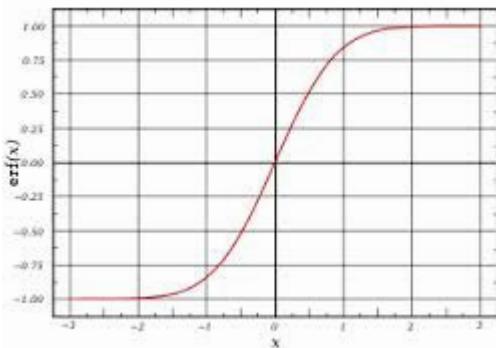
答：我们肯定会拥有智能机器。这只是时间问题。我们肯定会有那种虽然不是非常聪明，但是可以做有用事情的机器，比如无人驾驶车。



至于这需要多长时间？人工智能研究者之前很长的一段时间都低估了制造智能机器的难度。我可以打个比方：研究进展就好像开车去目的地。当我们在研究上发现了新的技术，就类似在高速路上开车一样，无人可挡，直达目的地。

但是现实情况是，我们是在一片浓雾里开车，我们没有意识到，研究发现的所谓的高速公路，其实只是一个停车场，前方的尽头有一个砖墙。很多聪明人都犯了这个错误，人工智能的每一个新浪潮，都会带来这么一段从盲目乐观到不理智最后到沮丧的阶段。感知机技术、基于规则的专家系统、神经网络、图模型、支持向量机甚至是深度学习，无一例外，直到我们找到新的技术。当然这些技术，从来就不是完全失败的，它们为我们带来了新的工具、概念和算法。

虽然我相信我们最终一定会制造出超越人类智能的机器，但是我并不相信所谓的奇点理论。大部分人觉得技术的进展是个指数曲线，其实它是个 S 型曲线。S 型曲线刚开始的时候跟指数曲线很像。而且奇点理论比指数曲线还夸张，它假设的是渐进曲线。线性、多项式、指数和渐进以及 S 曲线的动态演变，都跟阻尼和摩擦因子有关系。而未来学家却假设这些因子是不存在的。未来学家生来就愿意做出盲目的预测，尤其是他们特别渴望这个预测成真的时候，可能是为了实现个人抱负。



问：你还在 NYU 数据科学中心当兼职主任，你怎么权衡或者结合在 Facebook 的工作？

答：我在 NYU 数据科学中心已经不再担任实际职务了，而是名誉主任。在新的主任选举出来以前，代理主任是 [S.R. Srinivasa “Raghu” Varadha](#)，世界上最有名的统计学家。NYU 已经展开了新主任的遴选工作。在数据科学中心的建立过程中，我花费了相当大的精力。我们现在数据科学方面有硕士生项目，未来会有博士生项目。现在中心有 9 个工作空缺，和 Berkeley 和华盛顿大学合作，我们从 Moore 和 Sloan 基金会拿到了非常大的一个五年基金支持，中心现在和 Facebook 等各大公司都有合作伙伴关系，我们马上要盖新大楼。下一任中心主任将会非常热爱自己的工作！

问：“数据科学”这个词，近来经常出现，被认为是统计学、商业智能等学科的交叉。这个数据科学和之前的“数据挖掘”或者“预测分析”有什么不同？它是一个新学科？它的公理和原则有哪些？

答：数据科学指的是自动或半自动地从数据中抽取知识。这个过程涉及很多的学科，每个学科对它都有自己的名字，包括概率估计，数据挖掘，预测分析，系统辨识，机器学习，人工智能等等。

从各个学科的角度，统计学、机器学习以及某些应用数学，都可以声称是数据科学的起源。但是实际上，数据科学之于统计学、机器学习以及应用数学，正如上个世纪六十年代的计算机科学之于电子电气、物理和数学。后来计算机科学变成了一个完全成熟的独立学科，而不是数学或者工程的子学科，完全是因为它对社会非常重要。

当今的数字时代，数据指数级别的疯涨，从数据中自动抽取知识这个问题，已经逐渐成为了人们的焦点。这正促进数据科学成为一个真正独立的学科。也促进着统计学、机器学习和数学重新划定自己的学科界限。数据科学还创造了“方法学科”的科学家和“领域学科”如自然科学、商科、药学和政府的工作人员紧密交流的机会。

我预测，未来十年，很多顶尖大学都会设立数据科学系。

问：您对于“大数据”这个词怎么看？作为一种趋势或者一个时髦词，它有多少成分是夸大，多少是真实的？

答：对于这个词，我觉得最近社交网络上比较流行的那个笑话非常贴切，把大数据比作青少年性行为：每个人都在谈论它，没人知道到底怎么做，每个人都以为其他人知道怎么做，所以每个人都声称自己也在做，这个笑话我是从 Dan Ariely 的 Facebook 上看到的。

我碰到过一些人，哪怕是闪盘可以存下，笔记本可以处理的数据，都坚持使用 Hadoop 来处理。

这个词确实被夸大了。但是如何收集、存储和分析海量数据这个问题是实际存在的。我经常怀疑的是诸如“大数据”这样的名字而已，因为今日的大数据，将成为明日的小数据。还有，很多问题都是因为数据量不足而产生的，比如基因和医疗数据，数据永远都不会够用。

问：数据科学家被称为“二十一世纪最性感的职业”。你给想要进入这个领域的人们提一点建议？

答：如果你是个本科生，多学数学、统计学还有物理学，更重要的是你要学着写代码（学三到四门计算机课程）。如果你有本科学位，那么你可以申请 NYU 数据科学中心的硕士项目。

问：你最近对哪本书比较感兴趣？不接触计算机和手机的时候你都在干些什么？

答：在我空闲的时候，我会造一些微型飞行器，我非常喜欢 3D 打印，我还经常研究带微控制器的电路板，我还希望能更好的制造音乐（我收集电子风门控制器）。大多数非小说的作品我都看，还听可多的爵士乐（或者类似的音乐）。

查看英文原文：<http://www.kdnuggets.com/2014/02/exclusive-yann-lecun-deep-learning-facebook-ai-lab.html>

感谢[吴甘沙](#)对本文的审校，感谢[包研](#)对本文的策划。

原文：<http://www.infoq.com/cn/articles/interview-yann-lecun>

# NoSQL、JSON 和时间序列数据管理：Anuj Sahni 访谈

作者 [Srinivas Penchikala](#)，译者 吴海星

时间序列数据是以固定的时间间隔按顺序排列的变量值。时间序列数据正从四面八方向我们涌来：传感器、移动设备、Web 追踪、财经事件、工厂自动化以及各种工具——只列举这几个吧。

时间序列数据管理最近比较受关注，InfoQ 采访了 Oracle NoSQL 数据库的高级产品经理 Anuj Sahni，请他介绍时间序列数据，以及如何对这种类型的数据建模。

**InfoQ：**什么是时间序列数据，它跟结构化数据有什么区别？

Sahni: 时间序列是按统一的时间间隔采集的数据点序列。比如来自全球证券交易所的股票行情信息，定期采集的贵金属价格，按周期采集的特定经 / 纬度的天气详情，从制造机械或石油钻井平台的传感器上源源不断输出的数据。时间序列信息不一定会跟结构化数据有什么不同，但它是 a) 数量和速度出众 b) 信息的稀疏性使得很难用传统的数据存储方式保存它，因为那种方式是为定义良好的结构化数据设计的。

**InfoQ：**将一个数据集建模为时间序列数据有什么好处？

Sahni: 并不是所有的数据都需要用时间序列表示，但如果你有持续的信息项进来（比如说来自于传感器），并且你想要基于时间维度分析那些数据，那么比较稳健的做法是保留每条信息项的到达时间，并为之建立索引以便提高查询速度。

**InfoQ：**为了把时间序列类型的数据保存在我们的数据库中，设计时要考虑哪些问题？

Sahni: 时间序列数据，一般来说，都是大量生产的，所以需要特别考虑如何保存在数据库中，以及如何很好地访问它们。所以你应该尽量事先明确数据的访问方式，比如在证券交易的演示中，我们知道每只股票的行情信息都是每秒生成一次的，即每只股票每天有 86K 的行情信息。如果我们把那么多条记录都各自存在不同的行中，那访问这些信息的时间复杂度将会非常巨大，所以我们可以按 5 分钟，1 小时，或者一天将这些记录分组为一条向量记录。把信息存在更大数据块里的好处显而易见，因为你要从 NoSQL 存储中获取特定时间段的数据时，所做的查询次数更少。还有一点需要记住，如果你开的窗口太小，那你就要做很多的读写操作；如果太大，那耐久力就是个问题，如果出现系统故障，你就要有信息丢失了。所以你得掌握好两者的平衡。

**InfoQ：**在使用时间序列数据时，开发人员应该遵循什么样的设计或架构模式呢？

Sahni: 在设计任何一个时间序列应用时，最重要的就是要了解客户端应用的访问模式，以及它所要求的信息粒度。尽管我们可能需要持久化所有的时间序列数据，但一般我们不需要把每个数据点都存成数据库里的单独记录。如果我们可以定义一个时间窗口，然后把那个时间段的所有读取都存为一个数组，那么我们可以极大削减存储在数据库中的实际记录数量，提升系统的性能。继续前面那个例子，如果每支股票每天生成 84K 的数据点，那么你在访问这些信息时就需要在硬盘上做很多的随机 I/O，简直太多了，但如果你可以把同样的信息存为一个数组，每个数组放一个小时的信息，那么取一天的数据只需要查询 24 次就行了。

另外一个需要平衡的是数组对象的大小（不管是 JSON, XML 还是其他什么东西）。你应该不想让数组的窗口大小大到最终返回的数据超出你的需要太多，那样就浪费了宝贵的带宽。但如果你把窗口定的太精细，那很可能会降低吞吐量，并增加系统的延迟。所以要处理好两边的平衡，得出合适的时间窗口大小，从 / 往数据库中读 / 写时间序列数据集合。

**InfoQ：** 把数据集存为时间序列数据有什么限制吗？

Sahni: 我得说处理时间序列数据所面临的真正挑战是基于需求和访问模式微调系统。如果将来访问模式变了，那你可能必须重建索引，重新计算数组的大小来优化查询。所以每个时间序列应用的定制性都非常强，你可以采用最佳实践，但却不能指望只通过引入数据建模模板来解决不同的时间序列问题。

Anuj 在去年的 JavaOne 大会上讲了如何在应用中管理时间序列数据。

## 关于受访者

Anuj Sahni 是 Oracle 的高级产品经理，他负责管理公司在 NoSQL 数据库和大数据产品上的领导地位。Anuj 在世界 500 强公司里有超过 12 年的产品管理 / 开发经验。他领导过高度成功的、分布在全球的跨职能团队开发新软件和基于云的服务。Anuj 在佛罗里达大学取得了计算机工程硕士学位，还在生物信息学领域发表过学术论文。在闲暇时，他喜欢骑行、追踪，还有跟他的两个女儿玩。

查看原文链接：[NoSQL, JSON, and Time Series Data Management: Interview with Anuj Sahni](#)

原文：<http://www.infoq.com/cn/articles/nosql-time-series-data-management>

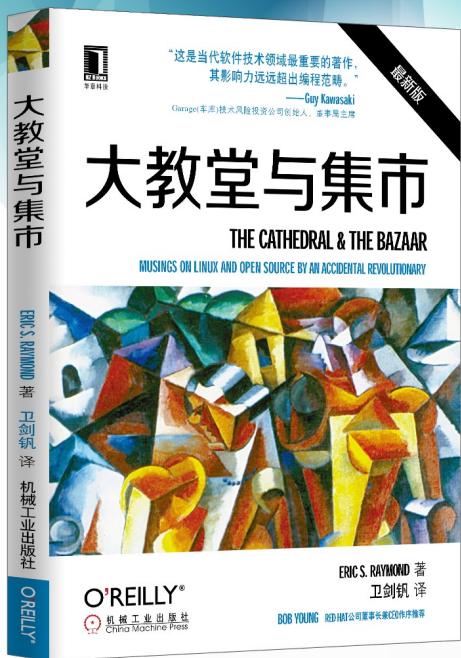
专题 | Topic

## 本期专题：支付宝的测试

提起支付宝，大家都知道。现在网上支付的环节已经越来越离不开它了。而最近两年来的双十一、双十二，更是让大家认识到支付宝系统的强大之处。

想要保证系统能够在各种各样的压力下正常运行，除了架构、代码方面的工作之外，还有一项非常重要的工作就是测试。这期《架构师》的专栏中，就让我们来一起了解一下支付宝在测试方面做出的努力。

本期专栏中选入的三篇文章都是由支付宝测试团队一线工程师撰写的，都是他们宝贵经验的总结，希望能够让各位读者吸取其中的知识，为自己的实际工作带来启示。



# 开源运动的《圣经》 当代软件技术领域最重要的著作 中文版首次出版!

这是对当代软件技术最重要的著作，其影响力远远超出编程领域。

——Guy Kawasaki  
Garage(车库)技术风险投资公司创始人、董事局主席

本书是开源运动的独立宣言，它清晰、透彻和准确地描述了开源运动的理论与实际应用，对开源软件运动的成功和Linux操作系统的广泛采用都起到了至关重要的作用。本书在开源运动中的地位相当于基督教的圣经，用黑客们的话说，这是“黑客藏经阁”的第一收藏。

本书不只是在讲开源和黑客，所有关心软件开发和IT发展的人都应该花些时间通读全书，书中给出了大量充满智慧的观点和经过验证的概念，如命令体系、礼物文化、以少成多、内部市场、竞次、反公地模型、委员会设计、同侪声誉、模因工程、SNAFU现象、进化不利条件、软件业是服务行业、组织结构决定产品结构、准入门槛越低稳定性越高、程序员是资产而非成本等，这些内容一定会给你带来新的启示和思考。



机械工业出版社  
China Machine Press

咨询信箱: yannan@hzbook.com 联系人: 闫南

# 支付宝分布式事务测试方案

作者 李跃

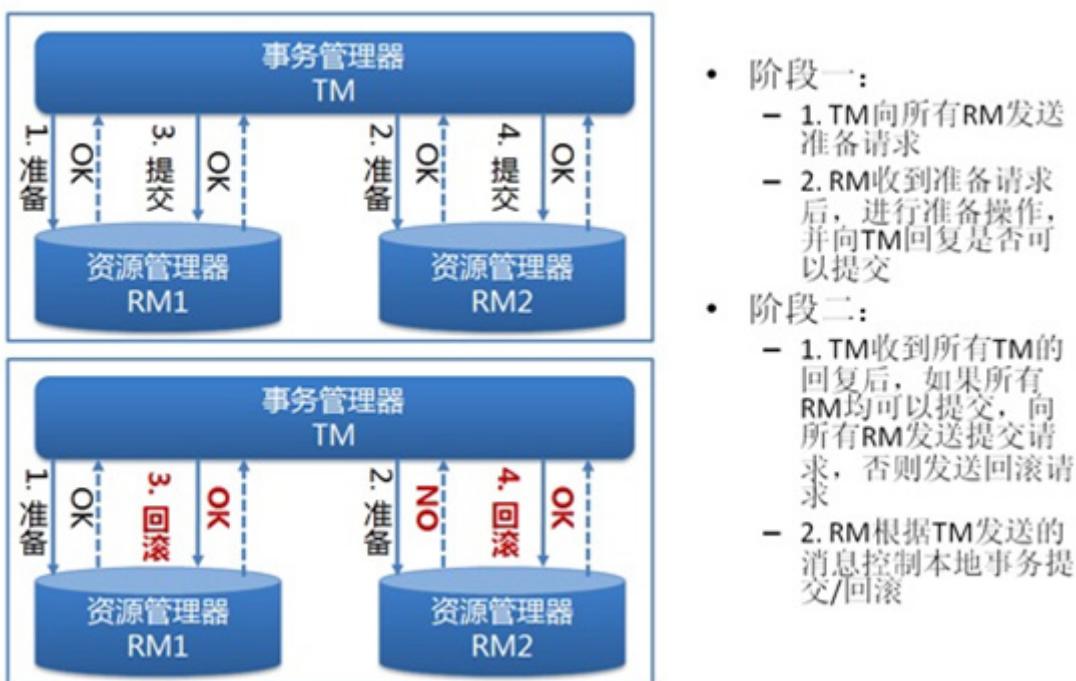
传统的基于数据库本地事务的解决方案只能保障单个服务的一次处理具备原子性、隔离性、一致性与持久性，但无法保障多个分布服务间处理的一致性。因此，我们必须建立一套分布式服务处理之间的协调机制，保障分布式服务处理的原子性、隔离性、一致性与持久性。

基于 SOA 架构，整个支付宝系统会拆分成一系列独立开发、自包含、自主运行的业务服务，并将这些服务通过各种机制灵活地组装成最终用户所需要的产品与解决方案。

在多个服务协同完成一次业务时，由于业务约束（如红包不符合使用条件、账户余额不足等）、系统故障（如网络或系统超时或中断、数据库约束不满足等），都可能造成服务处理过程在任何一步无法继续，使数据处于不一致的状态，产生严重的业务后果，所以我们需要一个分布式事务的解决方案，用来协调多个服务的业务一致性。

## 支付宝的分布式事务框架

支付宝开发的分布式事务是基于两阶段提交的理论（Two Phase Commit），首先给出两阶段提交的逻辑图：

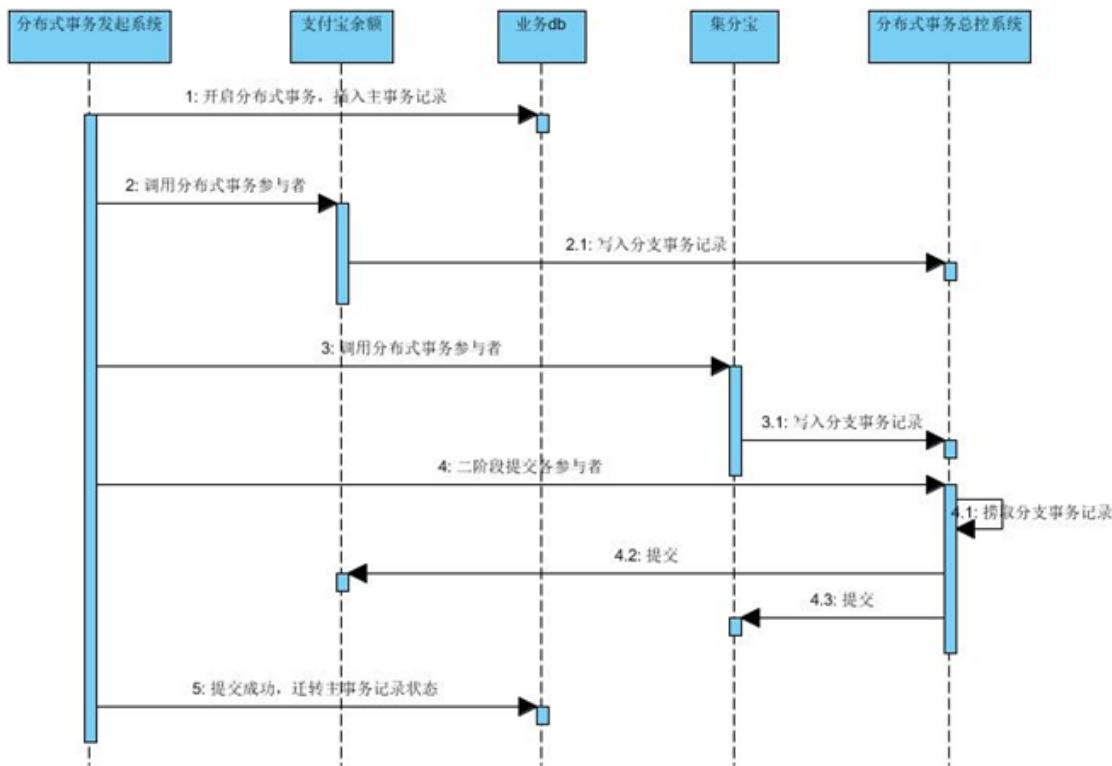


为了能够有效的让框架进行分布式事务的提交、回滚等动作，框架需要在整个两阶段执行过程中记录下足够的信息，设计了两张表来记录相关信息：

分布式业务控制活动主表：记录了全局事务的活动状态；

原子业务活动表：记录了原子业务活动的状态；

我们用一个例子来说明：看一个典型的分布式事务场景。业务场景描述：用户购买商品，使用支付宝余额支付；



## 测试方案

### 分析步骤

- 角色定位
- 各分支的业务活动记录状态
- 梳理业务各个场景
- 验证梳理场景
- 恢复 & 回查机制

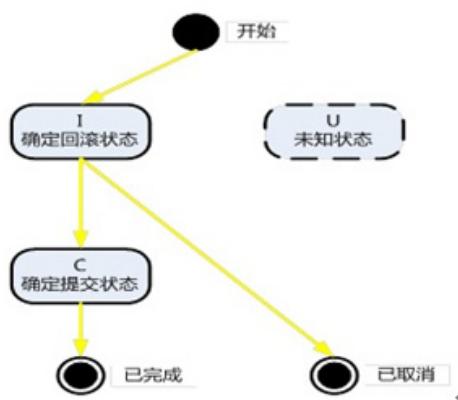
### 角色定位

首先测试人员需要分析所测试的系统处于分布式事务中的哪一个环节中，是处于事务的发起者，还是事务的参与者，不同的角色的定位对于测试分析角度不同，主要有以下的区别：

**发起者：**分为同库 / 异库模式，主要区分是控制全局事务状态的主事务记录是否持久化在自己系统的 db 中；

**参与者：**分为本地 / 远程模式，主要区分是否可以创建嵌套的分布式事务；

## 各分支的业务活动记录状态



### 主事务记录：

根据业务场景的不同，主事务记录状态也会相应改变，主要的状态机变化如图所示，测试人员需要模拟业务场景来验证状态机的迁转是否正确；

同库：初始状态： I；提交成功： C；提交失败： I

异库：初始状态： U；提交成功： U；提交失败： U

### 梳理 & 验证业务场景

- 分析维度
- 一阶段：预处理：成功 / 失败；
- 二阶段：提交 / 回滚；
- 预期结果
- 各个状态场景

### 恢复 & 回查

- 恢复：应用使用分布式事务，出现处理失败的业务活动，为了确保产生的影响不破坏业务一致性，我们必须对这些记录进行恢复处理
- 回查：对于异库模式，事务状态为 U，若提交或回滚失败，分布式事务总控系统无法感知这笔分布式事务是否执行成功，需要业务系统提供相应的回查接口；

恢复及回查接口需要特别关注，对于分布式事务的正常二阶段提交或回滚，业务场景覆盖时多半都能 `check` 到，但是对于恢复及回查逻辑，很多时候都会遗漏，所以测试人员需要对这块特别做一个分析。

感谢侯伯薇对本文的审校。

原文：<http://www.infoq.com/cn/articles/distributed-transaction-testing-scheme-of-zhifubao>



促进软件开发领域知识与创新的传播



中文 | 英文 | 日文 | 葡文 | .....

# 支付宝的性能测试

作者 付丽华 孙玉星

## 一、性能测试支付宝场景介绍

2013 年双 11 过程当中，促销开启的第一分钟内支付宝的交易总额就突破了一亿元，短时间内大量用户涌入的情况下，如何保证用户的支付顺畅，是对支付宝应用系统的一个极大的挑战。

支付宝的性能测试场景分为性能基线测试，项目性能测试。

任意一笔交易过来，我们都需要对交易进行风险扫描，对于有可能是账户盗用的交易，我们会把这笔支付直接拒绝掉，或者通过手机校验码等方式进行风险释放。

我们有一个老的扫描平台 A，现在需要构建一个新的扫描平台 B，对 A 中关键技术进行升级，并增加额外的功能。扫描的策略是存储在 DB 中的，需要通过发布来更新到应用服务器的内存中。

## 二、性能测试需求分析和方案制定

### a. 需求挖掘

1)，查看业务方的显性需求。业务方给到的需求为平台 B 的分析性能要优于平台 A 的性能。除此之外无其它的需求。

2)，挖掘隐性需求。了解业务架构，了解业务流程。为了保证扫描的性能，大量的存储类的需求被设计为异步处理，但是结果类的扫描需要使用到前面落地的数据，那么在系统正常运行时是否会产生落地数据读取不到的问题，在存储抖动时是否会导致后续的分析扫描全部失效？

首先我们通过运维监控平台拿到平台 A 的分析性能， $RT < 130ms$ ,  $TPS > 35$ .

基于以上需求挖掘，我们确认的性能测试场景为

1. 扫描性能场景。（单场景）
2. 发布性能场景。（单场景）
3. 扫描过程中发布性能场景。（混合场景）

### b. 技术方案

- 1). 评估我们的系统架构，系统调到链路，定位可能存在问题的瓶颈点。
- 2). 掌握详细技术实现方案，了解具体技术方案可能存在的性能问题。

比如我们是否使用到了脚本动态编译，是 Java 脚本还是 groovy 脚本。是否使用到了线程池等异步处理，系统幂等性是如何控制的，数据结构是如何存储与读取的，是决策树还是图型结构。

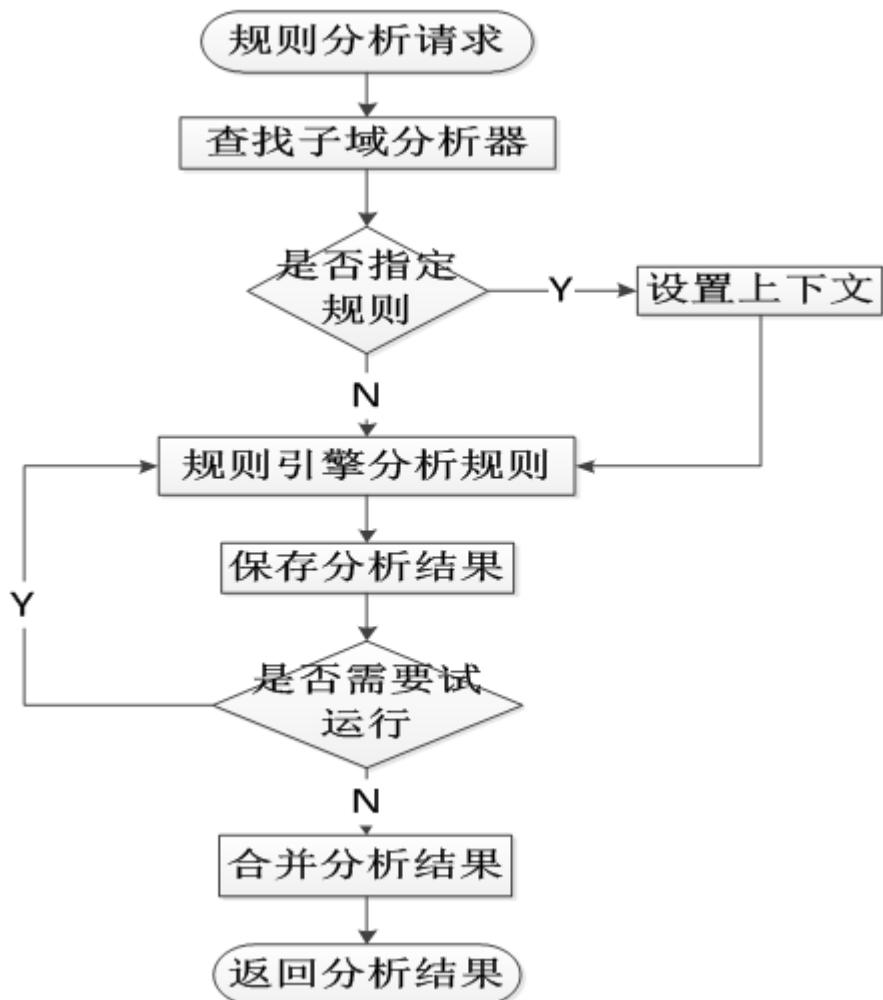
3). 了解系统环境的差异，比如服务器位数、CPU、内存的差异，JDK 版本及位数的差异。

基于以上的技术方案，我们确认了上述 3 个性能测试场景可能存在的性能问题

### 1. 扫描性能场景

技术方案为扫描引擎 drools2 升级到了 drools5.

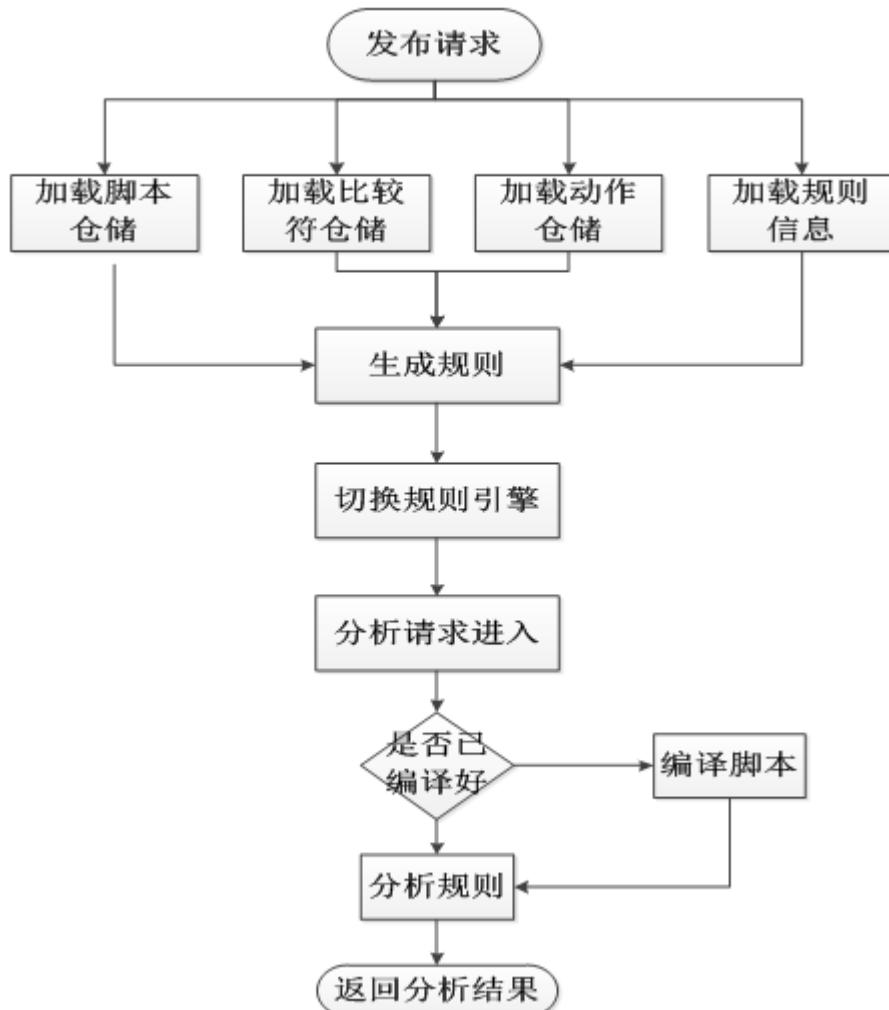
性能关注点为请求扫描 RT，TPS 是否满足我们的需求；JVM Old 区内存溢出，Old 区内存泄露；GC 频率过高。CPU 使用率，load.



### 2. 发布性能场景

技术方案为规则 DB 捞取 -> 规则加载 -> 规则引擎切换 -> 规则脚本编译。

性能关注点为 CPU 使用率，load。JVM Perm 区内存溢出，Perm 区内存泄露，GC 频率过高。GC 暂停应用时间。



### 3. 扫描过程中发布性能场景。

性能关注点为请求扫描 RT, TPS。规则发布耗时, CPU 使用率, load, JVM GC 频率。

### c. 性能测试方案制订

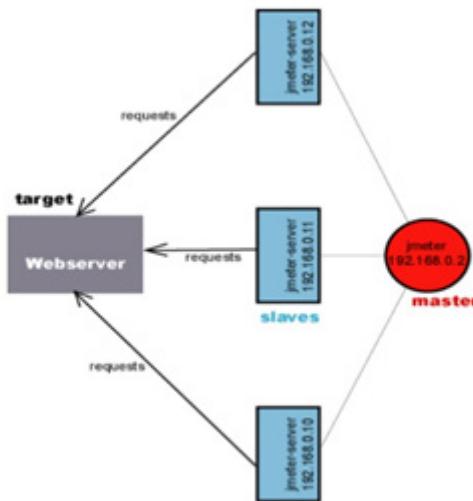
1. 分布式压测, 参数自动化, 使用单元测试脚本, 接口测试脚本, jmeter 脚本等进行压测。
2. 性能结果收集及统计。
3. 性能测试通过标准。

基于以上的分析

#### 1. 扫描性能场景

性能测试方案:

使用 jmeter 脚本进行分布式压测, 一台 master, 三台 slaver. 参数自动构建, 使用高斯定时器模拟真实场景。



使用 **jmeter** 收集分析性能数据，使用 **nmon** 收集服务器性能数据，使用 **jconsole** 收集 **JVM** 数据。

通过标准：

**RT<130ms, TPS>35.**

**JVM old 区内无内存泄露，无内存溢出。GC 时间间隔 >30min，暂停应用时间 <150ms.**

**CPU<70%， load < core\*1.5。**

## 2. 发布性能场景

性能测试方案：

发布时间间隔时间限制从 **1min** 调整为 **3s**，更快的暴露问题。

使用单元测试类推送发布消息。

服务器 **shell** 脚本收集发布模块性能数据。

使用 **nmon** 收集服务器性能数据。

使用 **jconsole** 收集 **JVM** 数据。

通过标准：

**JVM Perm 区内无内存泄露，无内存溢出。GC 时间间隔 >10min，暂停应用时间 <200ms.**

发布时间 <30s

**CPU<70%， load < core\*1.5。**

## 3. 扫描过程中发布性能场景

性能测试方案：

使用 `jmeter` 脚本进行分布式压测，同时提交发布请求进行发布。

同时使用扫描性能场景和发布性能场景收集数据功能。

通过标准：

`RT < 扫描性能场景结果 RT * 110%.`

`TPS > 扫描性能场景结果 TPS * 90%.`

发布时间 < 40s。

## d. 发现的问题

### 1. 扫描性能场景

`AVG RT = 473ms, CMS GC = 90ms`, 应用暂停时间 = 1s, 因此测试未通过。

问题定位：

dump 内存，使用 `ibm memory analyzer` 分析。

确认 `cms gc` 的原因为 `drools` 引擎的 `finalize` 方法。`Finalize` 方法不能正确的释放对象的引用关系，导致引用关系一直存在，无法释放。

调优方案：

根据 `drools` 的升级文档，升级 `drools` 引擎后解决此问题

### 2. 发布性能场景

`CMS GC` 回收失败，内存无法被释放，应用宕机。

问题定位：

`GC` 回收比例为默认值 68%，`OLD` 区内存 1024M，那么回收的临界值为  $1024 \times 0.68 = 696.32M$ 。系统的 `JVM` 内存占用为 500M，扫描策略相关的内存为 120M，在切换的过程中，依赖额外的 120M，因此只有在可用内存大于 740M 时才能正常回收。

解决方案：

调整 `JVM` 参数，扩大 `GC` 回收比例。

后续技术方案改造，使用增量发布解决此问题。

### 3. 扫描过程中发布性能场景

问题定位：

扫描平台发布流程，当首次请求进来时执行脚本动态编译过程，由于脚本较多，因此所有脚本的动态编译时间较长，在此过程中，进来的所有请求都会被 hand 住，造成大量超时

解决方案：

把脚本的动态编译提前到首次请求调用进来之前，编译通过后再切换扫描引擎，保证首次请求进来前一切准备就绪。

## 三：性能测试的执行和结果收集

### 3.1 性能测试的执行

性能测试的执行需要具备以下几个条件：施压工具，测试环境以及对测试结果的收集工具。

#### 3.1.1 施压工具

我们先来说说施压工具，支付宝使用的主流施压工具是开源工具 **Apache JMeter**，支持很多类型的性能测试：

1. Web – HTTP, HTTPS

2. SOAP

3. Database via JDBC

4. LDAP

5. JMS

6. 任何用 **java** 语言编写的接口，都可二次开发并调用。

支付宝大部分接口是 **webservice** 接口，基于 **soap** 协议，且都是 **java** 开发，所以使用 **jmeter** 非常方便，即使 **jmeter** 工具本身没有自带支持的协议，也可以通过开发插件的方式支持。

#### 3.1.2 测试环境

测试环境包括被压机和施压机环境，需要进行硬件配置和软件版本确认，保证系统干净，无其他进程干扰，最好能提前监控半小时到 1 小时，确认系统各项指标都无异常。

另外除了被压机和施压机，有可能应用系统还依赖其他的系统，所以我们需要明确服务器的数量和架构，**1** 是方便我们分析压力的流程，帮助后面定位和分析瓶颈，**2** 是由于我们线下搭建的环境越接近线上，测试结果越准确。但是通常由于测试资源紧张或者需要依赖外围，例如银行的环境，就会比较麻烦，通常我们会选择适当的进行环境 **mock**。当然，**Mock** 的时候尽量和真实环境保持一致，举个简单的例子，如果支付宝端系统和银行进行通信，线上银行的平均处理时间为 **100ms**，那么如果我们在线下性能测试时需要 **mock** 银行的返回，需

要加入 100ms 延迟，这样才能比较接近真实的环境。

另外除了测试环境，还有依赖的测试数据也需要重点关注，数据需要关注总量和类型，例如支付宝做交易时，db 中流水万级和亿级的性能肯定是不一样的；还有 db 是否分库分表，需要保证数据分布的均衡性。一般考虑到线下准备数据的时长，一般性能测试要求和线上的数据保持一个数量级。

### 3.1.3 测试结果收集工具

测试结果收集主要包括以下几个指标：

响应时间、tps、错误率、cpu、load、IO、系统内存、jvm (java 虚拟内存)。

其中响应时间、tps 和业务错误率通过 jemter 可以收集。

Cpu、load、io 和系统内存可以通过 nmon 或 linux 自带命令的方式来监控。

Jvm 可以通过 jdk 自带的 jconsole 或者 jvisualvm 来监控。

总体来说，监控了这些指标，对系统的性能就有了掌握，同样这样指标也可以反馈系统的瓶颈所在。

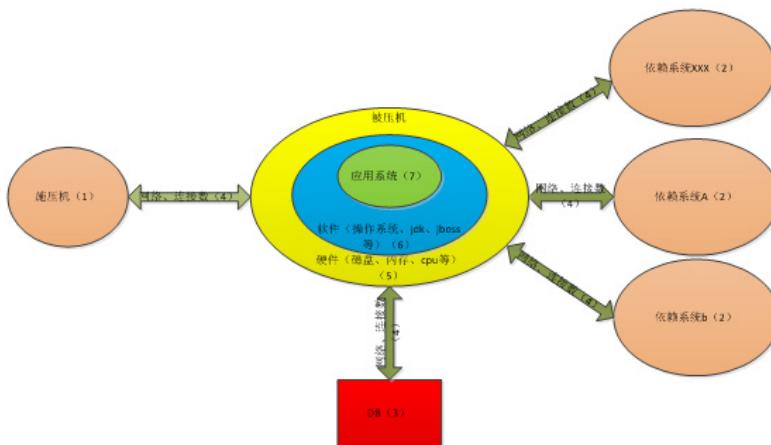
## 四. 性能测试瓶颈挖掘与分析

我们在上面一章中拿到性能测试结果，这么多数据，怎么去分析系统的瓶颈在哪里呢，一般是按照这样的思路，先看业务指标：响应时间、业务错误率、和 tps 是否满足目标。

如果其中有一个有异常，可以先排除施压机和外围依赖系统是否有瓶颈，如果没有，关注网络、db 的性能和连接数，最后关注系统本身的指标：

1. 硬件：磁盘是否写满、内存是否够用、cpu 的利用率、平均 load 值
2. 软件：操作系统版本、jdk 版本、jboss 容器以及应用依赖的其他软件版本
3. Jvm 内存管理和回收是否合理
4. 应用程序本身代码

先看下图：是一般性能测试环境部署图



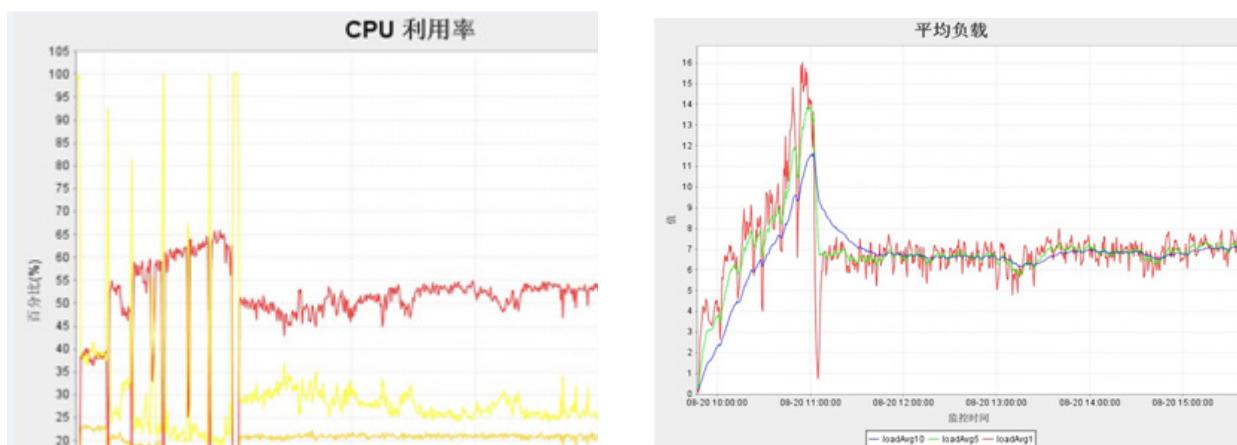
我们在定位的时候，可按照标注中的 1、2、3 数字依次进行排查，先排查施压机是否有瓶颈、接着看后端依赖系统、db、网络等，最后看被压机本身，例如响应时间逐渐变慢，一般来说是外围依赖的系统出现的瓶颈导致整体响应变慢。下面针对应用系统本身做下详细的分析，针对常见问题举 1~2 个例子：

#### 4.1 应用系统本身的瓶颈

##### 1. 应用系统负载分析：

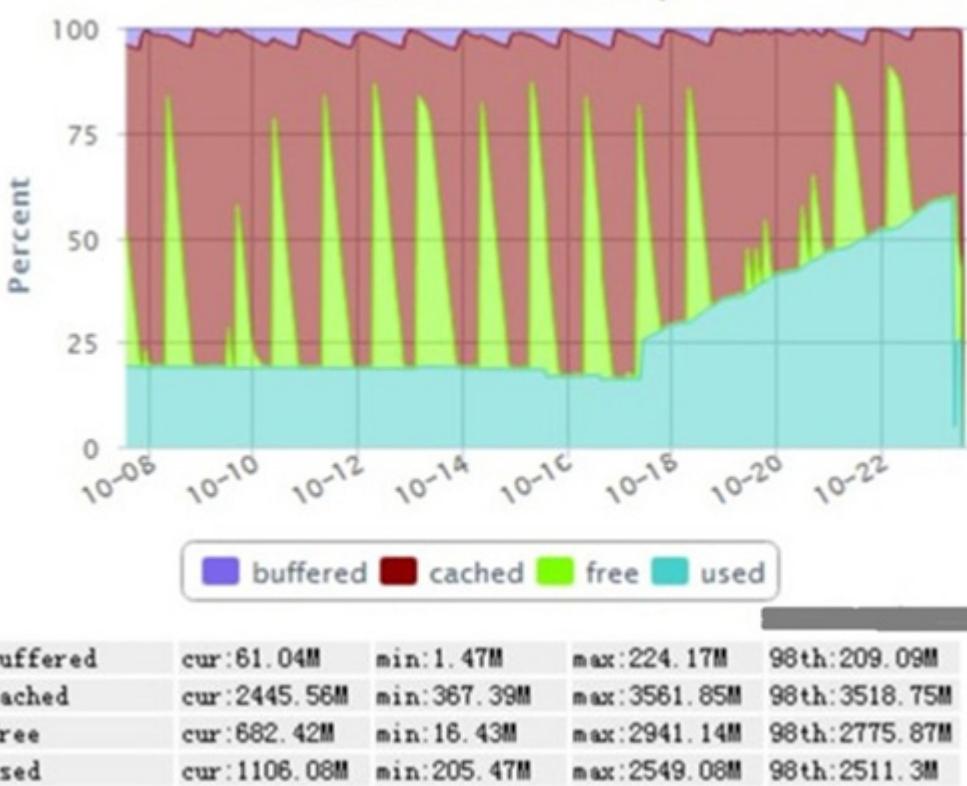
服务器负载瓶颈经常表现为，服务器受到的并发压力比较低的情况下，服务器的资源使用率比预期要高，甚至高很多。导致服务器处理能力严重下降，最终有可能导致服务器宕机。实际性能测试工作中，经常会用以下三类资源指标判定是否存在服务器负载瓶颈：

- CPU 使用率
- 内存使用率
- Load 一般 cpu 的使用率应低于 50%，如果过高有可能程序的算法耗费太多 cpu，或者某些代码块进行不合理的占用。Load 值尽量保持在  $\text{cpus}+2$  或者  $\text{cpus}*2$ ，其中 cpu 和 load 一般与并发数成正比（如下图）



- 内存可以通过 2 种方式来查看： 1) 当 `vmstat` 命令输出的 `si` 和 `so` 值显示为非 0 值，则表示剩余可支配的物理内存已经严重不足，需要通过与磁盘交换内容来保持系统的稳定；由于磁盘处理的速度远远小于内存，此时就会出现严重的性能下降；`si` 和 `so`

的值越大，表示性能瓶颈越严重。2) 用工具监控内存的使用情况，如果出现下图的增长趋势（**used** 曲线呈线性增长），有可能系统内存占满的情况：



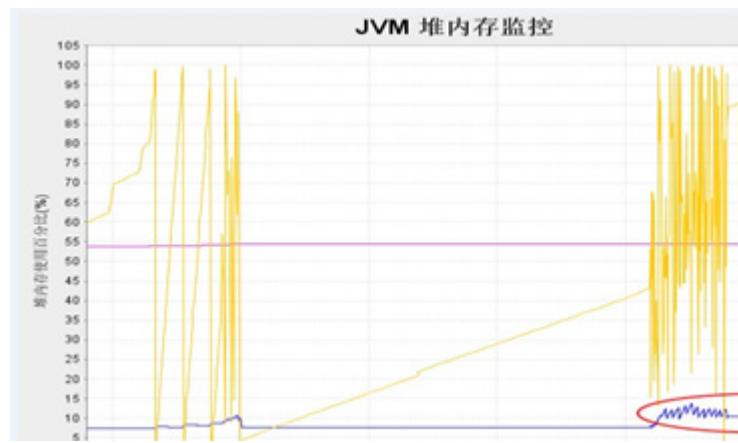
- 如果出现内存占用一直上升的趋势，有可能系统一直在创建新的线程，旧的线程没有销毁；或者应用申请了堆外内存，一直没有回收导致内存一直增长。

## 4.2 JVM 瓶颈分析

### 4.2.1 GC 频率分析

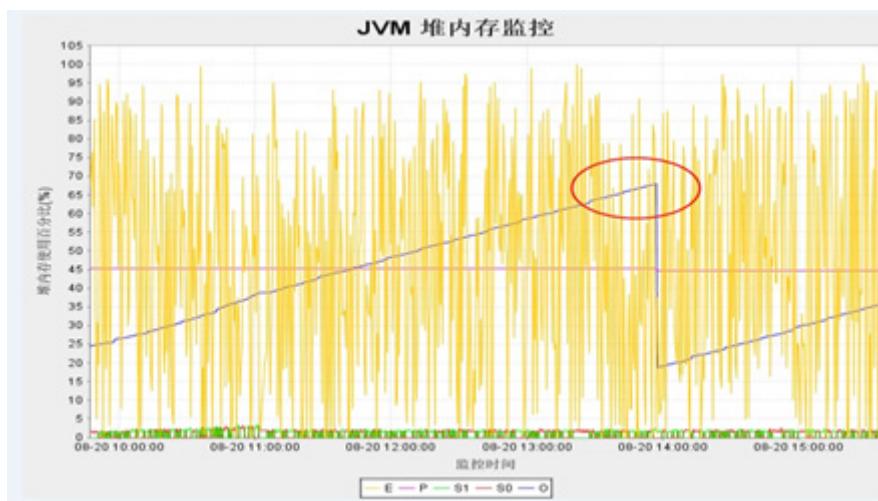
对于 java 应用来说，过高的 GC 频率也会在很大程度上降低应用的性能。即使采用了并发收集的策略，GC 产生的停顿时间积累起来也是不可忽略的，特别是出现 **cmsgc** 失败，导致 **fullgc** 时的场景。下面举几个例子进行说明：

1. **Cmsgc** 频率过高，当在一段较短的时间区间内，**cmsgc** 值超出预料的大，那么说明该 JAVA 应用在处理对象的策略上存在着一些问题，即过多过快地创建了长生命周期的对象，是需要改进的。或者 **old** 区大小分配或者回收比例设置得不合理，导致 **cmsgc** 频繁触发，下面看一张 **gc** 监控图（蓝色线代表 **cmsgc**）



由图看出：`cmsGC` 非常频繁，后经分析是因为 jvm 参数 `-XX:CMSInitiatingOccupancyFraction` 设置为 15，比例太小导致 `cms` 比较频繁，这样可以扩大 `cmsgc` 占 old 区的比例，降低 `cms` 频率注。

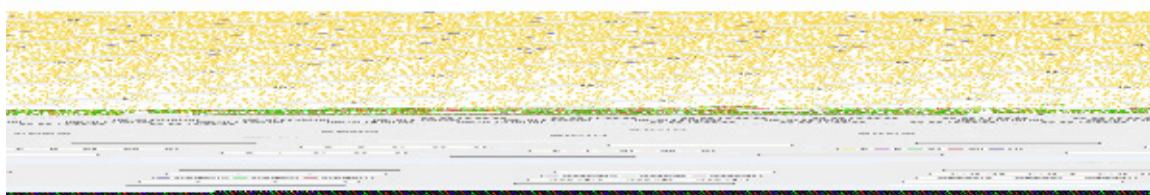
调优后的图如下：



## 2. fullgc 频繁触发

当采用 `cms` 并发回收算法，当 `cmsgc` 回收失败时会导致 `fullgc`：

```
2013-09-05T09:22:34.771+0800: [GC [1 CMS-initial-mark: 209530K(696320K)] 220794K(1249280K)], 0.0136260 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]
2013-09-05T09:22:35.150+0800: [CMS-concurrent-mark: 0.365/0.365 secs] [Times: user=0.37 sys=0.00, real=0.37 secs]
2013-09-05T09:22:35.157+0800: [CMS-concurrent-preclean: 0.006/0.007 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]
CMS: abort preclean due to time 2013-09-05T09:22:40.173+0800: [CMS-concurrent-abortable-preclean: 2.461/5.016 secs] [Times: user=2.43 sys=0.05, real=5.01 secs]
2013-09-05T09:22:40.175+0800: [GC[YG occupancy: 60044 K (552960 K)][Rescan (parallel), 0.0260900 secs][weak refs processing, 0.0806340 secs][class unloading, 0.0160600 secs][1 CMS-remark: 209530K(696320K)] 269574K(1249280K), 0.1466890 secs] [Times: user=0.22 sys=0.00, real=0.15 secs]
2013-09-05T09:22:40.580+0800: [CMS-concurrent-sweep: 0.253/0.258 secs] [Times: user=0.27 sys=0.02, real=0.26 secs]
2013-09-05T09:22:40.585+0800: [CMS-concurrent-reset: 0.004/0.004 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]
```



由上图可以看出 `fullgc` 的耗时非常长，在 6~7s 左右，这样会严重影响应用的响应时间。

经分析是因为 **cms** 比例过大，回收频率较慢导致，调优方式：调小 **cms** 的回比例，尽早触发 **cmsgc**，避免触发 **fullgc**。调优后回收情况如下

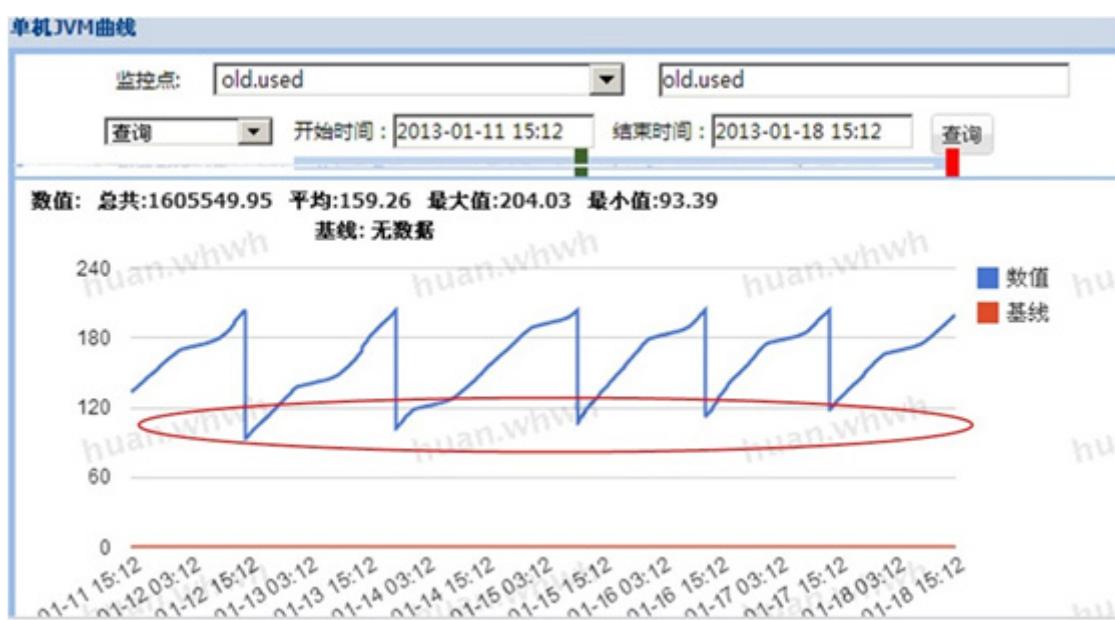
```

2013-09-05T09:22:34.771+0800: [GC [1 CMS-initial-mark: 209530K(696320K)] 220794K(1249280K), 0.0136260 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]
2013-09-05T09:22:35.150+0800: [CMS-concurrent-mark: 0.365/0.365 secs] [Times: user=0.37 sys=0.00, real=0.37 secs]
2013-09-05T09:22:35.157+0800: [CMS-concurrent-preclean: 0.006/0.007 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]
CMS: abort preclean at time 2013-09-05T09:22:40.173+0800: [GC[CMS occupancy: 60044 K (552960 K)][Rescan (parallel), 0.0260900 secs][weak refs processing, 0.0806340 secs][class unloading, 0.0160600 secs] [1 CMS-remark: 209530K(696320K)] 269574K(1249280K), 0.1466890 secs] [Times: user=0.22 sys=0.00, real=0.15 secs]
2013-09-05T09:22:40.580+0800: [CMS-concurrent-sweep: 0.253/0.258 secs] [Times: user=0.27 sys=0.02, real=0.26 secs]
2013-09-05T09:22:40.585+0800: [CMS-concurrent-reset: 0.004/0.004 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]

```

可以看出 **cmsgc** 时间缩短了很多，优化后可以大大提高。从上面 2 个例子看出 **cms** 比例不是绝对的，需要根据应用的具体情况来看，比如应用创建的对象存活周期长，且对象较大，可以适当提高 **cms** 的回收比例。

### 3. 疑似内存泄露，先看下图



分析：每次 **cmsgc** 没有回收干净，**old** 区呈上升趋势，疑似内存泄露

最终有可能导致 OOM，这种情况就需要 dump 内存进行分析：

- 找到 oom 内存 dump 文件，具体的文件配置在 jvm 参数里： `-XX:HeapDumpPath=/home/admin/logs`
- `-XX:ErrorFile=/home/admin/logs/hs_err_pid%p.log`
- 借助工具：MAT，分析内存最大的对象。具体工具的使用这里就不再介绍。

感谢侯伯薇对本文的审校。

原文：<http://www.infoq.com/cn/articles/performance-test-of-zhifubao>



促进软件开发领域知识与创新的传播



中文 | 英文 | 日文 | 葡文 | .....

# 数据设计测试分析方案

作者 蒋雷

## 数据拆分机制

### 一、背景

支付宝从 2009 年开始进行 1111 大促开始，从 2009 年大促 600w 的交易量到 2013 年大促那天 2 亿的交易量；在这四年中，支付宝系统的容量经历了每年指数级的提升；如果需要支持这么大的容量的话，初步估算，一个支付系统至少得支持每天 30 亿次的数据库事务，150 亿次的 dao 访问，30T 的数据包传输；如果不进行数据库拆分，这么大的开销单靠一台物理 db 完全支撑不了的，所以必须对单点的物理 db 进行拆分。

### 二、数据拆分的方案

Sharding 的基本思想就要把一个数据库切分成多个部分放到不同的数据库 (server) 上，从而缓解单一数据库的性能问题；不太严格的讲，对于海量数据的数据库，如果是因为表多而数据多，这时候适合使用垂直拆分，即把关系紧密（比如同一业务模块）的表切分出来放在一个物理 db 上；如果表并不多，但每张表的数据非常多，这时候适合水平拆分，即把表的数据按某种规则（比如按 ID 散列）拆分到多个物理 db 上；当然，现实中更多是这两种情况混杂在一起，这时候需要根据实际情况做出选择，也可能会综合使用垂直与水平拆分，从而将原有数据库切分成类似矩阵一样可以无限扩充的数据库 (server) 阵列。

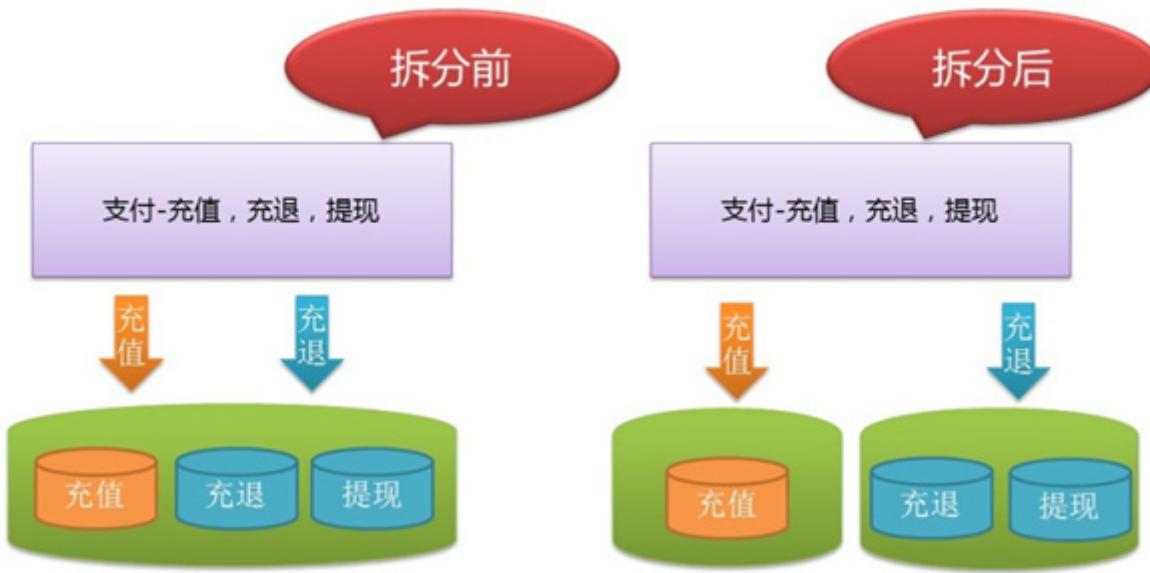
垂直拆分的最大特点就是规则简单，实施也更为方便，尤其适合各业务之间的耦合度非常低，相互影响很小，业务逻辑非常清晰的系统，也可以对故障进行有效的隔离；在这种系统中，可以很容易做到将不同业务模块所使用的表分拆到不同的数据库中；根据不同的表来进行拆分，对应用程序的影响也更小，拆分规则也会比较简单清晰。

水平拆分更多的是解决 db 带来的整体容量和性能问题，对于表中的数据按照某一种规则拆分到不同的物理表或者物理 db 中，从而把应用对数据库的访问压力分散到不同的 db；其实本质还是进行分布式的部署架构思路；但是这种规则拆分后的数据必须尽量达到平均，否则就失去拆分的意义。

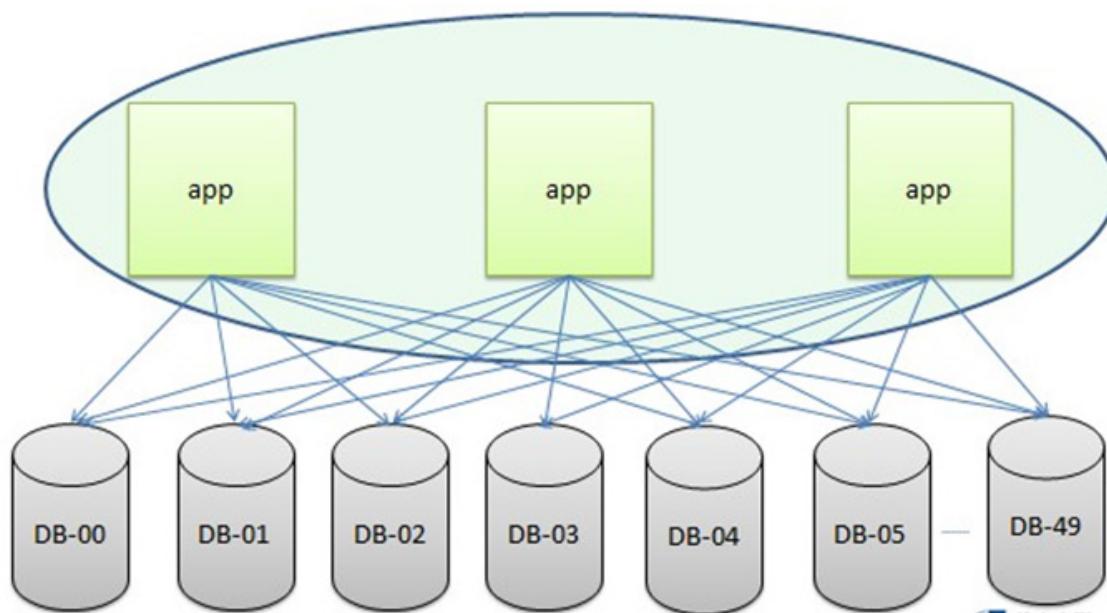
### 三、数据拆分的实例介绍

在支付宝的架构演进过程中也进行了很多数据拆分这一方面的改造。

垂直拆分的案例：支付宝系统，该系统上面运行了三大块的业务：充值 / 充退 / 提现；充值业务非常重要，业务的流量也非常大，是个非常关键的业务；充退和提现的业务重要性相对低一点，但是这类业务会进行一些大数据量的操作，对于 db 的影响非常大；所以为了确保充值业务的稳定性，db 层面进行了垂直拆分的改造，把充值和充退提现数据分两个物理 db 进行存储和访问，具体如下图所示：



水平拆分的案例：支付宝系统，支付关键的业务都在该系统上，而且每天的业务量都非常大；根据业务量增长的趋势，在不久的将来，单个 db 已经支撑不了这么庞大的业务吞吐量，所以我们对支付的数据按照用户 id 的维度进行了水平的拆分，具体如下图所示：



#### 四、 水平拆分应用测试分析

本次我们会对水平拆分的方案做详细的测试关注点分析。

##### 功能性需求测试分析

###### 1. 分库分表功能逻辑的覆盖

考虑到拆分前的数据是落地到一个物理库一个物理表，拆分后其实是落地到多个物理库的多个物理表中，所以对于不同拆分维度的数据落地到正确的数据库和数据表是水平拆分类改造

最重要也是最关键的测试覆盖点；但是有些同学可能会产生疑问，如果需要覆盖 100 个甚至更多的数据库表该怎么覆盖；其实这个可以用自动化脚本直接从数据访问层对每个 `sql` 进行全量覆盖，这个点也主要是对路由规则和数据源配置的覆盖校验。

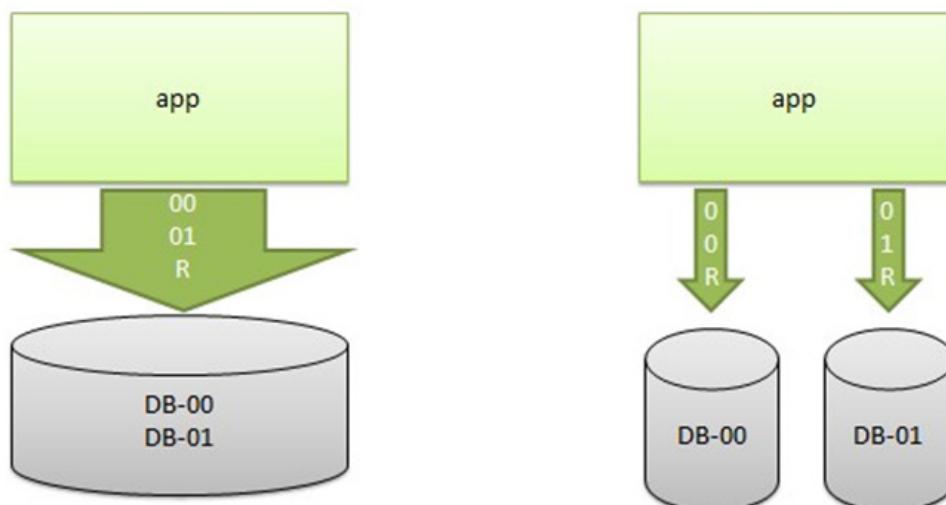
## 2. 跨库的事务无法支持

事务是完整性的单位，一个事务的执行是把数据库从一个一致的状态转换成另一个一致的状态，但是如果在一个事务模板中出现了多次数据库的操作，并且这些操作对应的数据源针对的是不同的物理存储，那么这个事务一致性肯定不能被保证的；在应用分库之后，原先在一个事务中的多次数据库的操作有可能被指向不同的物理存储，所以针对这样的逻辑是肯定需要改造的，测试分析的时候一定要重点分析这样的场景。



## 3. 业务数据 `merge` 需要覆盖所有分库

拆分前，一份表中所有的数据都是存放在一个物理表中；而数据库表拆分后，原来在一个表中的数据有可能会分布在不同的物理数据库和物理数据表中，所以在进行一些大数据的查询操作时，从原先的一条 `sql` 执行就能获取全量数据，拆分后就可能需要通过针对不同表的多个 `sql` 获取并在应用逻辑中进行合并，所以测试的时候一定需要关注到这部分的改造是否到位，是否有分析遗漏的地方。



## 非功能性需求测试分析

一般数据库拆分的项目实施过程中，在应用发布后，其实数据层面还是没有真正的做到物理上的切分，需要逻辑分库的这么一个过程主要是考虑到发布过程中的一些数据和业务兼容性，还有一些可能是由于物理机器采购和调试的进度和项目发布的时间点不匹配等客观因素，所以在真正做物理拆分前会有一个逻辑分库运行的阶段。

### 1. 逻辑拆分过程中数据一致性

逻辑拆分阶段，从应用系统的视角来看对应的数据源都是独立的，应用的业务逻辑也都是按照真正分库分表的逻辑去实现的，但是这些独立数据源底层还是同一套物理数据源，物理 **db** 的部署也都是按照拆分前的方式，虽然应用看到的是多个库多份表，但是 **db** 层面还是单库单表；对于发布过程中新的代码访问的数据库表和老代码访问的数据库表都是同一份，虽然应用上看到的是不同的，所以通过这种方式就可以避免发布过程中新老代码对于数据访问兼容性的问题，测试对于逻辑拆分的验证主要还是通过对新老代码业务的验证，确保访问的数据是同一份的。

### 2. 物理拆分过程中数据一致性

物理拆分是数据库拆分项目的最终目标，一般在物理拆分之前，会进行历史数据的迁移工作，确保新的数据库保存和原先物理库一致的数据，但是对于增量的数据处理还需要其他配套的方案支持；一般进行物理拆分的方案有两种：第一种是比较暴力的方式，直接通过数据源连接的 **dns** 切换到新机器，不过这种方式有个缺点，需要进行新老库增量数据的同步和对比，这个过程可能比较长，所以对业务造成损失的时间会比较长；还有一种方式就是需要应用进行 **failover** 的改造来配合数据的拆分，对所需要迁移的数据库先进行 **failover** 的切换，然后再进行 **dns** 的切换，这样业务只会在 **failover** 切换和回切瞬间产生影响。

### 3：新老库 **seq** 同步性问题

在做物理拆分的时候，由于会涉及到老数据的迁移和同步，所以对于在新库产生的数据要避免和老库迁移过来的数据产生冲突，这个冲突主要是唯一性主键相关的冲突，在 **oracle** 中我们一般会通过数据库提供的 **seq** 和其他的特定位来组成这个主键，对于这个 **seq** 在新老库又是两套，所以我们在确定新库的 **seq** 时一定要和原先老库的 **seq** 进行隔离，而这个梳理和明确的过程需要测试的同学和 **dba** 一起明确。

## 性能需求测试分析

说到性能提升，其实这是我们做分库分表改造的初衷，在未进行分库分表的情况下，随着时间和发展，库中的表会越来越多，表中的数据量也会越来越大，相应地，数据操作，增删改查的开销也会越来越大；另外，由于无法进行分布式部署，而一台服务器的资源（CPU、磁盘、内存、IO 等）是有限的，最终数据库所能承载的数据量、数据处理能力都将遭遇瓶颈，所以分库分表就是为了解决单个 **db** 性能的瓶颈。

### 1. 关注 **tps**, 内存, 连接数等性能指标

在针对分库分表的项目做性能压测的时候，我们主要关注的还是应用整体的 **tps**，响应时间，内存情况，还有数据源连接数等；而对于分库之后物理数据源的连接数大小的设定，我们也得出了一个基本的公式，如果物理数据源的连接数调的太大可能会增加应用的内存的开销，如果物理数据源的连接数调的太小，则会影响应用整体的 **tps** 和响应时间，所以对于连接数的设置必须要通过多轮压测得到一个理想的值。



## 2. 连接数, `ps-cache`, `fetch_size` 设置合理性

`ps-cache`: `PreparedStatement` (开源框架的类) 是 JDBC 里面提供的对象, 而 JBOSS 里面引入了一个 `PreparedStatementCache`, `PreparedStatementCache` 即用于保存与数据库交互的 `prepareStatement` 对象; `PreparedStatementCache` 使用了一个本地缓存的 LRU 链表来减少 SQL 的预编译, 减少 SQL 的预编译, 意味着可以减少一次网络的交互和数据库的解析 (有可能在 `session cursor cache hits` 中命中, 也可能是 `share pool` 中命中), 这对应用的 DAO 响应延时是很大的提升, 另外, `PreparedStatementCache` 也不是设置越大越好, 毕竟, `PreparedStatementCache` 是会占用 JVM 内存的; 之前出现过一个核心系统因为在增加了连接数 (拆分数据源) 后, 这个参数设置没有修改, 导致 JVM 内存被撑爆的情况; (JVM 内存占用情况 = 连接总数 \* `PreparedStatementCache` 设置大小 \* 每个 `PreparedStatement` 占用的平均内存)。

`fetch_size`: 当我们执行一个 SQL 查询语句的时候, 需要在客户端和服务器端都打开一个游标, 并且分别申请一块内存空间, 作为存放查询的数据的一个缓冲区; 这块内存区, 存放多少条数据就由 `fetchsize` 来决定, 同时每次网络包会传送 `fetchsize` 条记录到客户端; 应该很容易理解, 如果 `fetchsize` 设置为 20, 当我们从服务器端查询数据往客户端传送时, 每次可以传送 20 条数据, 但是两端分别需要 20 条数据的内存空闲来保存这些数据; `fetchsize` 决定了每批次可以传输的记录条数, 但同时, 也决定了内存的大小; 这块内存, 在 oracle 服务器端是动态分配的, 而在客户端, `PS` 对象会存在一个缓冲中 (LRU 链表);

`fetchsize` 的设置, 跟具体业务系统有关系, 没有一个最好的值可以供各个应用都可以使用, 一般的系统, `fetchsize` 使用 `jdbc` 的默认值就可以了。在某个特定的条件下测试的 `fetchsize`, 得出一个值, 然后所有人都用这个值来设置自己的应用系统, 一般情况下, 这仅仅只是一些资源的浪费, 但是在数据源拆分改造中, 当 `fetchsize` 设置的太大, 有可能会导致性能的急剧下降, 甚至会导致应用上可怕的 JVM 内存溢出, 在不少公司发生过这种惨痛的教训, 建议在设置这个值之前, 先做一个 JVM 内存的 DUMP, 以便能够对内存的占用情况有一个清晰的了解。

## 五、总结

功能性:

1. 分库分表功能逻辑的覆盖
2. 跨库的事务无法支持
3. 业务数据 `merge` 需要覆盖所有分库

非功能：

1. 逻辑拆分过程中数据一致性
2. 物理拆分过程中数据一致性
3. 新老库 seq 同步性问题

性能：

1. 关注 tps，内存，连接数等性能指标
2. 连接数，ps-cache，fetch\_size 设置合理性

感谢侯伯薇对本文的审校。

原文：<http://www.infoq.com/cn/articles/data-design-test-analysis-scheme>

# 一秒钟法则：来自腾讯无线研发的经验分享

作者 刘昕

在 2014 年 4 月 11 日的腾讯分享日活动上， 来自腾讯 MIG 的移动互联网事业群运营总监 /T4 专家，负责运营 QQ 手机浏览器、腾讯 PC 浏览器、腾讯手机安全管家、腾讯电脑管家产品的刘昕介绍了移动无线产品研发中的“一秒钟法则”。本文根据该演讲内容整理形成。

移动互联网的一个很大问题在于无线网络跟以前的有线网络不一样，无论是网络的组织形态、架构、通讯机制，跟有线网络都有很大差异，这带来很多挑战。今天介绍的“一秒钟法则”就是根据我们在移动互联网研发、运营过程中总结出来的一条解决的原则。

## 手机接入服务器的流程

首先，手机要通过无线网络协议，从基站获得无线链路分配，才能跟网络进行通讯。

无线网络基站、基站控制器这方面，会给手机进行信号的分配，已完成手机连接和交互。

获得无线链路后，会进行网络附着、加密、鉴权，核心网络会检查你是不是可以连接在这个网络上，是否开通套餐，是不是漫游等。核心网络有 SGSN 和 GGSN，在这一步完成无线网络协议和有线以太网的协议转换。

再下一步，核心网络会给你进行 APN 选择、IP 分配、启动计费。

再往下面，才是传统网络的步骤：DNS 查询、响应，建立 TCP 链接，HTTP GET，HTTP RESPONSE 200 OK，HTTP RESPONSE DATA，LAST HTTP RESPONSE DATA，开始 UI 展现。

这是手机通过无线网络接入服务器的全过程。整个过程当中有几个困扰开发者的问题：

1. 无线网络是怎么给手机分配到无线链路的？
2. 核心网络有接入点（APN），这里的 CMNET 和 CMWAP 有什么区别，仅仅是协议不同吗吗？数据转发又有什么区别？一个数据包在不同网络上传输有不同吗？
3. 用户怎么最快的找到正确的服务器？内容怎么快速有效的加载，在第一时间显示出来？

这几个问题的重点在于其中的几个连接点：

1. 无线链路分配。这是一个物理实连接。
2. IP 层链接。这是一个逻辑虚连接。
3. TCP 层链接。这是一个逻辑虚连接。
4. HTTP 层链接。这是一个逻辑虚连接。

## 5. 用户在线。这是一个逻辑虚连接。

即使 TCP 连接建立，看到用户在线，也必须在手机获得无线链路分配的情况下，一个完整的通信才能真正完成，上行下行数据才能发送。这是移动互联网非常重要的特性。在现实中，手机已经分配 ip 也可能是没有无线链路，为什么？无线网络的资源是有限的，必须有效利用，这里由无线网络的信令机制完成无线网络资源的分配与释放。

以用手机打电话的场景示例：用户在手机上拨号出去后，手机会跟网络申请无线链路，呼叫申请会发给电路域的核心网，通过电话交换机找寻被叫电话，被叫方接通电话，无线链路建立；完成通话，挂断的时候，手机给网络发送指令，表示服务使用结束，把已经分配的无线链路释放。

上网的情况就比较复杂一些了。什么时候决定无线链路的分配？什么时候决定通讯完成？对于这两个时间点，不同的网络制式、不同的运营商都是不同的，不过大致上有几个区间值：

在 2G Edge 网络下，差不多是 1 秒钟不传数据，就释放物理连接，回收给其他人备用。3G 网络会延长几秒钟。

这样的设定是有原因的。比如现在我们这个会场里有 200 人，那么我们 200 人同时上网的前提是共享同一个基站的资源，共享资源必须要有规则，比如要有排序，根据资源情况、用户链接活跃决定分配还是回收，这都是通过无线网络信令控制的。

给一个手机分配无线信道的信令又有好几个情况，比如基站跟手机，基站跟基站控制器、核心网。举个例子，服务器从后台发送 push 消息，移动网络可能不知道这个手机是否活跃，不知道在哪个小区，移动网络就会发一个寻呼，在各个小区找这个手机，当然这个不能基于 IP，而是其他的网络标识。找到了之后，这个手机再去申请信道资源，然后才能接受 push。所以，这种场景下信令的消耗可能会在很多小区产生。

根据以上情况，就形成无线网络的一大特点：秒级状态管理，秒级状态转换。这两个操作都在几百 ms 到几秒之间进行，对于维持连接来说时间太短，对于从无连接到有连接的转换来说时间又太长。

相比之下，有线网络的状态管理如 ip 分配、tcp 连接释放，都是分钟级，而状态转换则是毫秒级。

这些通讯机制，同时加上无线网络的高延迟、高丢包。如何保证移动互联网的产品提供稳定的、可预期的服务质量，成为非常大的挑战：

1. 2G 网络上无线部分数据传输的延迟有几百 ms，4G 网络上无线部分传输延迟减少到几十 ms，核心网状态转换、协议转换 30~100ms，IP 骨干网上的延迟又跟物理距离以及运营商互联互通质量有关，跨运营商 50~400ms，同运营商 5~80ms，这个还要取决于网络拥塞的情况。
2. 无线网络误码率比有线高两个数量级，在不同时间段的波动也非常巨大。

怎么基于移动网络的特性去优化服务？这就是我们总结的一秒钟法则：在一秒内要完成的规定动作。

1. 2G 网络：1 秒内完成 dns 查询、和后台服务器建立连接

2. 3g 网络：1 秒内完成首字显示（首字时间）

3. wifi 网络：1 秒内完成首屏显示（首屏时间）

这些指标需要在终端度量，必须跟用户体验相关：首字时间、首屏时间都必须是用户可以直观感受到的。

## 优化思路

### 接入调度优化

接入调度优化首先要考虑的是减少 DNS 的影响。移动网络的 DNS 有如下特点：

- 骨干网无法识别移动用户在哪个城市，东西南北各个地方的调度没有充分调用。目前有一部分全国范围的 DNS 承载了超过 40% 的全网用户
- 很多山寨机的终端 local dns 设置是错误的

另外还有一些有线网络也一样会遇到的问题，如终端 DNS 解析滥用、域名劫持、DNS 污染、老化、脆弱等。不过对于这些问题，桌面的自愈性会比较好，而在手机上则比较难以解决。

对于 DNS 的问题，有两条主要的解决思路：

- 减少 DNS 的请求、查询、更新，也就是做 DNS 缓存
- 在终端配置 server list，直接访问 IP，不用 DNS

但仅仅这么做还不够，因为用户可能来自国内外不同的运营商，还需要进一步优化调度策略：

1. DNS 缓存需要多建立接入点，用不同域名区分

2. IP 列表需要更新以适应不同网络情况，要做到主动调度。好比最早我们只服务好移动用户就行，保证移动用户的接入质量优先，因为绝大多数用户集中在移动；现在国内有三个运营商，用户分布的比例在慢慢接近，要区分清楚；智能手机会用 wifi，接入的是电信、联通还是哪个运营商，不知道，所以你不可能预先设置场景再 if then，必须通过后台调度能力来解决。

再进一步优化，就产生一种融合的方式：

1. 先做域名解析，客户端直接连接解析的 IP，可以用 http 协议，也可以用 tcp socket

2. 多端口、多协议组合：不同协议有不同的限制，有些只能 http，有些只能 tcp socket，各种环境都要适应，客户端不能只支持一种协议

3. 终端测速：接入点越来越多，接入哪个合适，要选择，可以通过终端测速来选择最快的。  
你当然可以每一次新建连接都做测速，但是这样建立连接时间可能会很长；我们可以给用户先建立连接后，在后台根据长期速度监控、当前测速的结果，来做动态调度。也就是说，第一次连接可能不是最优，连接建立后动态测速，再转移到最快接入点。更进一步就是建立网络 profile，终端学习的思路。

测速采样的粒度我也说一下，移动互联网取 IP 段是没用的，比较好的粒度是到网元级别，比如广东有 20 多个 wap 网关，每一个网关的情况都不一样，这就是一个比较合适的粒度。

另外我们后面还有一个 SET 模型，可以就近提供服务。

最后想强调一个所有的接入调度原则：不要把调度逻辑写死在客户端，一定要由后台完成。

## 协议优化

协议参数优化这块就简单列一下，是我们长期运营过程中总结的一些经验，在启动移动互联网服务时作为运营的规范，可以少走很多弯路：

- 关闭 TCP 快速回收
- Init RTO 不低于 3 秒
- 初始拥塞控制窗口不小于 10。因为大部分页面在 10kB 以下，很多请求在慢启动阶段已经结束，改为 10 可以降低小页面资源传输时延。内容越大，这个选项的效果就比较不明显。
- Socket buffer > 64k
- TCP 滑动窗口可变
- 控制发包大小在 1400 字节以下，避免分片

协议优化的原则总结下来是这么几条：

- 连接重用
- 并发连接控制
- 超时控制
- 包头精简
- 内容压缩
- 选择更高效率的协议。无论是 TCP、HTTP、UDP、长连接、GZIP、SPDY、WUP 还是 WebP，每一种协议、方案都有其道理，没有最优，只有是否适合你的产品和服务特点，需要大家在运营过程验证和取舍。

## WAP 接入点优化

关于 WAP 接入点优化，可能有些人会说，我们的 App 是高端大气上档次的应用，是不是就不用做 WAP 优化？实际上我们的统计显示，目前有 5%-20% 的用户选择的接入点是 \* WAP (CMWAP、3GWAP、CTWAP)，这甚至包括一些 iPhone 终端。实际上，WAP 网关本质是个代理，不完全是落后的东西，随着技术的进步也在演进，以后在组网架构中可能有综合网关、内容计费网关来取代目前的 WAP 网关，所以建议也要一并考虑。以下是做 WAP 优化需要注意的一些问题：

- 资费提醒页面
- 302 跳转处理
- X-Online-Host 使用与处理
- 包大小限制
- 劫持与缓存
- 正确获取资源包大小

## 业务逻辑优化

简化逻辑：交互繁琐的内容尽量用标识更新。举一个例子，我们在老版的手机 QQ 上做过一个测试：假如我有 100 个好友，用手机 QQ 完成登陆，完成好友列表更新一遍，需要 3.5 分钟。这肯定是不合理的。建议用信令状态来通知是否需要更新，同时合理利用缓存。在比如玩游戏，好友给你送了很多星星，是让用户一次一次点还是批量点？从优化的角度肯定是批量点，从用户体验的角度这也更加舒服。

另一方面，延长域名图标的缓存时间也可以有效地优化访问次数。我们把手机腾讯网图标的缓存时长从 120 分钟延长到 2 天后，访问次数优化了差不多 35%。

柔性可用：这个意思就是在网络质量好的时候给高清大图，不好的时候先给用户看小图，点一下再拉取原图。举一个极端的例子，比如万一地震了，基站毁掉 20%，用户要给家人报平安，这时候产品上就必须优化，比如只发送文字，合理降低网络消耗。另外在响应很慢的时候，需要给用户一些合理的页面提示，比如提示用户再过 5 秒会发送，所以你不要一直刷屏，这也减少访问对后台服务、对网络的冲击。

其实腾讯公司的很多产品在业务逻辑优化，更好的适应移动互联网场景上，有很多非常好的思路，今天由于时间关系就不再展开来。

最后谈谈对优化方法的实践和结果的评估。QQ 手机浏览器从 4.5 版本、5.0 版本到 5.1 版本，我们对 2G 网络下的连接时间、3G 网络下的首字耗时、wifi 网络下的首屏耗时进行持续监控，耗时降到一秒钟以下还在不断的改进，每个新的版本平均值均有所压缩。这个结果是从每天用户实际使用的运营数据中得到的，覆盖到绝大多数的手机终端和网络环境。不过平均值只是一方面，我们另一方面还要看“有多少比例的数据满足了一秒钟法则”这个维度，因为无线网络的长尾数据波动很大，这一个维度也非常重要。目前现状是我们 2G 网络做到 79%，3G 网络做到 73%，wifi 网络做到 69%。目前我们的目标是达到 80%，实现之后，再进一步挑战 90% 的比例，不断追求极致。

感谢杨赛对本文的整理。

原文：<http://www.infoq.com/cn/articles/1sec-rule-from-tencent>

# QClub

我们影响有影响力的人

北京 上海 广州 大连 西安 太原 成都 杭州 武汉 南京 深圳...

QClub

邀请  
业内知名专家

自由开放的  
讨论氛围

定期举办的线下活动

结识  
圈内技术好友

InfoQ



中文 | 英文 | 日文 | 葡文 | .....

# 聊聊并发——生产者消费者模式

作者 方腾飞

## 为什么要使用生产者和消费者模式

在并发编程中使用生产者和消费者模式能够解决绝大多数并发问题。该模式通过平衡生产线程和消费线程的工作能力来提高程序的整体处理数据的速度。

在线程世界里，生产者就是生产数据的线程，消费者就是消费数据的线程。在多线程开发当中，如果生产者处理速度很快，而消费者处理速度很慢，那么生产者就必须等待消费者处理完，才能继续生产数据。同样的道理，如果消费者的处理能力大于生产者，那么消费者就必须等待生产者。为了解决这个问题于是引入了生产者和消费者模式。

生产者消费者模式是通过一个容器来解决生产者和消费者的强耦合问题。生产者和消费者彼此之间不直接通讯，而通过阻塞队列来进行通讯，所以生产者生产完数据之后不用等待消费者处理，直接扔给阻塞队列，消费者不找生产者要数据，而是直接从阻塞队列里取，阻塞队列就相当于一个缓冲区，平衡了生产者和消费者的处理能力。

这个阻塞队列就是用来给生产者和消费者解耦的。纵观大多数设计模式，都会找一个第三者出来进行解耦，如工厂模式的第三者是工厂类，模板模式的第三者是模板类。在学习一些设计模式的过程中，如果先找到这个模式的第三者，能帮助我们快速熟悉一个设计模式。

## 生产者消费者模式实战

我和同事一起利用业余时间开发的 **Yuna** 工具中使用了生产者和消费者模式。首先我先介绍下 **Yuna** 工具，在阿里巴巴很多同事都喜欢通过邮件分享技术文章，因为通过邮件分享很方便，同学们在网上看到好的技术文章，复制粘贴发送就完成了分享，但是我们发现技术文章不能沉淀下来，对于新来的同学看不到以前分享的技术文章，大家也很难找到以前分享过的技术文章。为了解决这问题，我们开发了 **Yuna** 工具。**Yuna** 取名自我喜欢的一款游戏最终幻想里的女主角。

首先我们申请了一个专门用来收集分享邮件的邮箱，比如 `share@alibaba.com`，同学将分享的文章发送到这个邮箱，让同学们每次都抄送到这个邮箱肯定很麻烦，所以我们的做法是将这个邮箱地址放在部门邮件列表里，所以分享的同学只需要象以前一样向整个部门分享文章就行，**Yuna** 工具通过读取邮件服务器里该邮箱的邮件，把所有分享的邮件下载下来，包括邮件的附件，图片，和邮件回复，我们可能会从这个邮箱里下载到一些非分享的文章，所以我们要求分享的邮件标题必须带有一个关键字，比如 [内贸技术分享]，下载完邮件之后，通过 `confluence` 的 `web service` 接口，把文章插入到 `confluence` 里，这样新同事就可以在 `confluence` 里看以前分享过的文章，并且 **Yuna** 工具还可以自动把文章进行分类和归档。

为了快速上线该功能，当时我们花了三天业余时间快速开发了 **Yuna1.0** 版本。在 **1.0** 版本中我并没有使用生产者消费模式，而是使用单线程来处理，因为当时只需要处理我们一个部门的邮件，所以单线程明显够用，整个过程是串行执行的。在一个线程里，程序先抽取全部的邮件，转化为文章对象，然后添加全部的文章，最后删除抽取过的邮件。代码如下：

```

public void extract() {
    logger.debug("开始" + getExtractorName() + "。。");
    // 抽取邮件
    List<Article> articles = extractEmail();
    // 添加文章
    for (Article article : articles) {
        addArticleOrComment(article);
    }
    // 清空邮件
    cleanEmail();
    logger.debug("完成" + getExtractorName() + "。。");
}

```

**Yuna** 工具在推广后，越来越多的部门使用这个工具，处理的时间越来越慢，**Yuna** 是每隔 5 分钟进行一次抽取的，而当邮件多的时候一次处理可能就花了几分钟，于是我在 **Yuna2.0** 版本里使用了生产者消费者模式来处理邮件，首先生产者线程按一定的规则去邮件系统里抽取邮件，然后存放在阻塞队列里，消费者从阻塞队列里取出文章后插入到 **confluence** 里。代码如下：

```

public class QuickEmailToWikiExtractor extends AbstractExtractor {

private ThreadPoolExecutor threadsPool;

private ArticleBlockingQueue<ExchangeEmailShallowDTO> emailQueue;

public QuickEmailToWikiExtractor() {
    emailQueue= new ArticleBlockingQueue<ExchangeEmailShallowDTO>();
    int corePoolSize = Runtime.getRuntime().availableProcessors() * 2;
    threadsPool = new ThreadPoolExecutor(corePoolSize, corePoolSize, 10L,
TimeUnit.SECONDS,
                new LinkedBlockingQueue<Runnable>(2000));

}

public void extract() {
    logger.debug("开始" + getExtractorName() + "。。");
    long start = System.currentTimeMillis();

    // 抽取所有邮件放到队列里
    new ExtractEmailTask().start();

    // 把队列里的文章插入到 Wiki
    insertToWiki();

    long end = System.currentTimeMillis();
    double cost = (end - start) / 1000;
    logger.debug("完成" + getExtractorName() + "，花费时间：" + cost + "秒");
}

/**
 * 把队列里的文章插入到 Wiki
 */
private void insertToWiki() {
    // 登录 wiki，每间隔一段时间需要登录一次
    confluenceService.login(RuleFactory.USER_NAME, RuleFactory.PASSWORD);

    while (true) {

```

```

        //2 秒内取不到就退出
        ExchangeEmailShallowDTO email = emailQueue.poll(2, TimeUnit.
SECONDS);
        if (email == null) {
            break;
        }
        threadsPool.submit(new insertToWikiTask(email));
    }
}

protected List<Article> extractEmail() {
    List<ExchangeEmailShallowDTO> allEmails = getEmailService().
queryAllEmails();
    if (allEmails == null) {
        return null;
    }
    for (ExchangeEmailShallowDTO exchangeEmailShallowDTO : allEmails) {
        emailQueue.offer(exchangeEmailShallowDTO);
    }
    return null;
}

/**
 * 抽取邮件任务
 *
 * @author tengfei.fangtf
 */
public class ExtractEmailTask extends Thread {
    public void run() {
        extractEmail();
    }
}
}

```

## 多生产者和多消费者场景

在多核时代，多线程并发处理速度比单线程处理速度更快，所以我们可以使用多个线程来生产数据，同样可以使用多个消费线程来消费数据。而更复杂的情况是，消费者消费的数据，有可能需要继续处理，于是消费者处理完数据之后，它又要作为生产者把数据放在新的队列里，交给其他消费者继续处理。如下图：



我们在一个长连接服务器中使用了这种模式，生产者 1 负责将所有客户端发送的消息存放在阻塞队列 1 里，消费者 1 从队列里读消息，然后通过消息 ID 进行 hash 得到 N 个队列中的一个，然后根据编号将消息存放在到不同的队列里，每个阻塞队列会分配一个线程来消费阻塞队列里的数据。如果消费者 2 无法消费消息，就将消息再抛回到阻塞队列 1 中，交给其他消费者处理。

以下是消息总队列的代码；

```
/**
 * 总消息队列管理
 *
 * @author tengfei.fangtf
 */
public class MsgQueueManager implements IMsgQueue{

    private static final Logger           LOGGER
= LoggerFactory.getLogger(MsgQueueManager.class);

    /**
     * 消息总队列
     */
    public final BlockingQueue<Message> messageQueue;

    private MsgQueueManager() {
        messageQueue = new LinkedTransferQueue<Message>();
    }

    public void put(Message msg) {
        try {
            messageQueue.put(msg);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    public Message take() {
        try {
            return messageQueue.take();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        return null;
    }
}
```

启动一个消息分发线程。在这个线程里子队列自动去总队列里获取消息。

```
/**
 * 分发消息，负责把消息从大队列塞到小队列里
 *
 * @author tengfei.fangtf
 */
static class DispatchMessageTask implements Runnable {
    @Override
    public void run() {
        BlockingQueue<Message> subQueue;
        for (;;) {
            // 如果没有数据，则阻塞在这里
            Message msg = MsgQueueFactory.getMessageQueue().take();
            // 如果为空，则表示没有 Session 机器连接上来,
            // 需要等待，直到有 Session 机器连接上来
            while ((subQueue = getInstance().getSubQueue()) == null) {
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
            // 把消息放到小队列里
            try {
                subQueue.put(msg);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }
}
```

使用 Hash 算法获取一个子队列。

```
/**
 * 均衡获取一个子队列。
 *
 * @return
 */
public BlockingQueue<Message> getSubQueue() {
    int errorCount = 0;
    for (;;) {
        if (subMsgQueues.isEmpty()) {
            return null;
        }
        int index = (int) (System.nanoTime() % subMsgQueues.size());
        try {
            return subMsgQueues.get(index);
        } catch (Exception e) {
            // 出现错误表示，在获取队列大小之后，队列进行了一次删除操作
            LOGGER.error("获取子队列出现错误", e);
            if ((++errorCount) < 3) {
                continue;
            }
        }
    }
}
```

使用的时候我们只需要往总队列里发消息。

```
// 往消息队列里添加一条消息
IMsgQueue messageQueue = MsgQueueFactory.getMessageQueue();
Packet msg = Packet.createPacket(Packet64FrameType.
TYPE_DATA, "{}".getBytes(), (short) 1);
messageQueue.put(msg);
```

## 小结

本章讲解了生产者消费者模式，并给出了实例。读者可以在平时的工作中思考下哪些场景可以使用生产者消费者模式，我相信这种场景应该非常之多，特别是需要处理任务时间比较长的场景，比如上传附件并处理，用户把文件上传到系统后，系统把文件丢到队列里，然后立刻返回告诉用户上传成功，最后消费者再去队列里取出文件处理。比如调用一个远程接口查询数据，如果远程服务接口查询时需要几十秒的时间，那么它可以提供一个申请查询的接口，这个接口把要申请查询任务放数据库中，然后该接口立刻返回。然后服务器端用线程轮询并获取申请任务进行处理，处理完之后发消息给调用方，让调用方再来调用另外一个接口拿数据。

另外 Java 中的线程池类其实就一种生产者和消费者模式的实现方式，但是实现方法更高明。生产者把任务丢给线程池，线程池创建线程并处理任务，如果将要运行的任务数大于线程池的基本线程数就把任务扔到阻塞队列里，这种做法比只使用一个阻塞队列来实现生产者和消费者模式显然要高明很多，因为消费者能够处理直接就处理掉了，这样速度更快，而生产者先存，消费者再取这种方式显然慢一些。

我们的系统也可以使用线程池来实现多生产者消费者模式。比如创建 N 个不同规模的 Java 线程池来处理不同性质的任务，比如线程池 1 将数据读到内存之后，交给线程池 2 里的线程继续处理压缩数据。线程池 1 主要处理 IO 密集型任务，线程池 2 主要处理 CPU 密集型任务。

感谢张龙对本文的策划与审校。

原文：<http://www.infoq.com/cn/articles/producers-and-consumers-mode>

# Node.js 软肋之回调大坑

作者 吴海星

**Node.js** 需要按顺序执行异步逻辑时一般采用后续传递风格，也就是将后续逻辑封装在回调函数中作为起始函数的参数，逐层嵌套。这种风格虽然可以提高 CPU 利用率，降低等待时间，但当后续逻辑步骤较多时会影响代码的可读性，结果代码的修改维护变得很困难。根据这种代码的样子，一般称其为 "callback hell" 或 "pyramid of doom"，本文称之为回调大坑，嵌套越多，大坑越深。

## 坑的起源

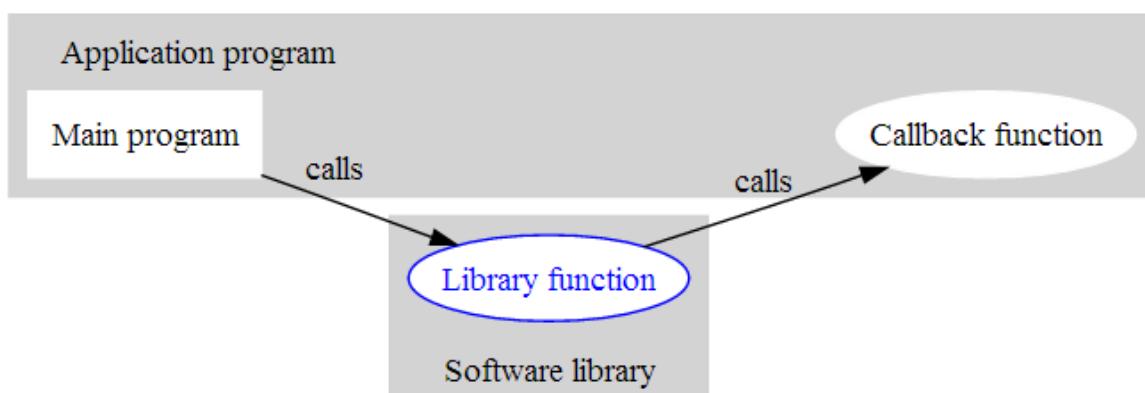
### 后续传递风格

为什么会有坑？这要从后续传递风格（continuation-passing style--CPS）说起。这种编程风格最开始是由 Gerald Jay Sussman 和 Guy L. Steele, Jr. 在 AI Memo 349 上提出来的，那一年是 1975 年，Schema 语言的第一次亮相。既然 JavaScript 的函数式编程设计原则主要源自 Schema，这种风格自然也被带到了 Javascript 中。

这种风格的函数要有额外的参数：“后续逻辑体”，比如带一个参数的函数。CPS 函数计算出结果值后并不是直接返回，而是调用那个后续逻辑函数，并把这个结果作为它的参数。从而实现计算结果在逻辑步骤之间的传递，以及逻辑的延续。也就是说如果要调用 CPS 函数，调用方函数要提供一个后续逻辑函数来接收 CPS 函数的“返回”值。

### 回调

在 JavaScript 中，这个“后续逻辑体”就是我们常说的回调（callback）。这种作为参数的函数之所以被称为回调，是因为它一般在主程序中定义，由主程序交给库函数，并由它在需要时回来调用。而将回调函数作为参数的，一般是一个会占用较长时间的异步函数，要交给另一个线程执行，以便不影响主程序的后续操作。如下图所示：



在 JavaScript 代码中，后续传递风格就是在 CPS 函数的逻辑末端调用传入的回调函数，并把计算结果传给它。但在不需要执行处理时间较长的异步函数时，一般并不需要用这种风格。我们先来看个简单的例子，编程求解一个简单的 5 元方程：

```

x+y+z+u+v=16
x+y+z+u-v=10
x+y+z-u=11
x+y-z=8
x-y=2

```

对于  $x+y=a$ ;  $x-y=b$  这种简单的二元方程我们都知道如何求解，这个 5 元方程的运算规律和这种二元方程也没什么区别，都是两式相加除以 2 求出前一部分，两式相减除以 2 求出后一部分。5 元方程的前一部分就是 4 元方程的和值，依次类推。我们的程序写出来就是：

#### 代码清单 1. 普通解法 -calnorm.js

```

var res = new Int16Array([16,10,11,8,2]),l= res.length;
var variables = [];
for(var i = 0;i < l;i++) {
    if(i === l-1) {
        variables[i] = res[i];
    }else {
        variables[i] = calculateTail(res[i],res[i+1]);
        res[i+1] = calculateHead(res[i],res[i+1]);
    }
}
function calculateTail(x,y) {
    return (x-y)/2;
}
function calculateHead(x,y) {
    return (x+y)/2;
}

```

方程式的结果放在了一个整型数组中，我们在循环中依次遍历数组中的头两个值 `res[i]` 和 `res[i+1]`，用 `calculateTail` 计算最后一个单值，比如第一和第二个等式中的  $v$ ；用 `calculateHead` 计算等式的“前半部分”，比如第一和第二个等式中的  $x+y+z+u$  部分。并用该结果覆盖原来的差值等式，即用  $x+y+z+u$  的结果覆盖原来  $x+y+z+u-v$  的结果，以便计算下一个 `tail`，直到最终求出所有未知数。

如果 `calculateTail` 和 `calculateHead` 是 CPU 密集型的计算，我们通常会把它放到子线程中执行，并在计算完成后用回调函数把结果传回来，以免阻塞主进程。关于 CPU 密集型计算的相关概念，可参考本系列的上一篇 [Node.js 软肋之 CPU 密集型任务](#)。比如我们可以把代码改成下面这样：

#### 代码清单 2. 回调解法 -calcb.js

```

var res = new Int16Array([16,10,11,8,2]),l= res.length;
var variables = [];
(function calculate(i) {
    if(i === l-1) {
        variables[i] = res[i];
        console.log(i + ":" + variables[i]);
        process.exit();
    }else {
        calculateTail(res[i],res[i+1],function(tail) {
            variables[i] = tail;
            calculateHead(res[i],res[i+1],function(head) {
                res[i+1] = head;
                console.log('-----'+i+'-----')
                calculate(i+1);
            });
        });
    }
})

```

```

        });
    }
})(0);
function calculateTail(x,y,cb) {
    setTimeout(function(){
        var tail = (x-y)/2;
        cb(tail);
    },300);
}
function calculateHead(x,y,cb) {
    setTimeout(function(){
        var head = (x+y)/2;
        cb(head);
    },400);
}

```

跟上一段代码相比，这段代码主要有两个变化。第一是 `calculateTail` 和 `calculateHead` 里增加了 `setTimeout`，把它们伪装成 CPU 密集型任务；第二是弃用 `for` 循环，改用函数递归。因为 `calculateHead` 的计算结果会影响下一轮的 `calculateTail` 计算，所以 `calculateHead` 计算要阻塞后续计算。而 `for` 循环是无法阻塞的，会产生错误的结果。此外就是 `calculateTail` 和 `calculateHead` 都变成后续传递风格的函数了，通过回调返回最终计算结果。

这个例子比较简单，既不能充分体现回调在处理异步非阻塞操作时在性能上的优越性，坑的深度也不够恐怖。不过也可以说明“用后续传递风格实现几个异步函数的顺序执行是产生回调大坑的根本原因”。下面有一个更抽象的回调样例，看起来更有代表性：

```

module.exports = function (param, cb) {
    asyncFun1(param, function (er, data) {
        if (er) return cb(er);
        asyncFun2(data, function (er,data) {
            if (er) return cb(er);
            asyncFun3(data, function (er, data) {
                if (er) return cb(er);
                cb(data);
            })
        })
    })
}

```

像 `function(er,data)` 这种回调函数签名很常见，几乎所有的 `Node.js` 核心库及第三方库中的 `CPS` 函数都接收这样的函数参数，它的第一个参数是错误，其余参数是 `CPS` 函数要传递的结果。比如 `Node.js` 中负责文件处理的 `fs` 模块，我们再看一个实际工作中可能会遇到的例子。要找出一个目录中最大的文件，处理步骤应该是：

用 `fs.readdir` 获取目录中的文件列表；

循环遍历文件，获取文件的 `stat`；

找出最大文件；

以最大文件的文件名为参数调用回调。

这些都是异步操作，但需要顺序执行，后续传递风格的代码应该是下面这样的：

### 代码清单 3. 寻找给定目录中最大的文件

```

var fs = require('fs')
var path = require('path')
module.exports = function (dir, cb) {
  fs.readdir(dir, function (er, files) { // [1]
    if (er) return cb(er)
    var counter = files.length
    var errored = false
    var stats = []
    files.forEach(function (file, index) {
      fs.stat(path.join(dir, file), function (er, stat) { // [2]
        if (errored) return
        if (er) {
          errored = true
          return cb(er)
        }
        stats[index] = stat // [3]
        if (--counter == 0) { // [4]
          var largest = stats
            .filter(function (stat) { return stat.isFile() }) // [5]
            .reduce(function (prev, next) { // [6]
              if (prev.size > next.size) return prev
              return next
            })
          cb(null, files[stats.indexOf(largest)]) // [7]
        }
      })
    })
  })
}

```

对这个模块的用户来说，只需要提供一个回调函数 `function(er,filename)`，用两个参数分别接收错误或文件名：

```

var findLargest = require('./findLargest')
findLargest('./path/to/dir', function (er, filename) {
  if (er) return console.error(er)
  console.log('largest file was:', filename)
})

```

介绍完 **CPS** 和回调，我们接下来看看如何平坑。

## 解套平坑

编写正确的并发程序归根结底是要让尽可能多的操作同步进行，但各操作的先后顺序仍能正确无误。服务端的代码一般逻辑比较复杂，步骤多，此时用嵌套实现异步函数的顺序执行会比较痛苦，所以应该尽量避免嵌套，或者降低嵌套代码的复杂性，少用匿名函数。这一般有几种途径：

1. 最简单的是把匿名函数拿出来定义成单独的函数，然后或者像原来一样用嵌套方式调用，或者借助流程控制模块放在数组里逐一调用；
2. 用 `Promises`；
3. 如果你的 `Node` 版本  $>= 0.11.2$ ，可以用 `generator`。

我们先介绍最容易理解的流程控制模块。

## 流程控制模块

**Nimble** 是一个轻量、可移植的函数式流程控制模块。经过最小化和压缩后只有 837 字节，可以运行在 **Node.js** 中，也可以用在各种浏览器中。它整合了 **underscore** 和 **async** 一些最实用的功能，并且 **API** 更简单。

**nimble** 有两个流程控制函数，**\_.parallel** 和 **\_.series**。顾名思义，我们要用的是第二个，可以让一组函数串行执行的 **\_.series**。下面这个命令是用来安装 **Nimble** 的：

```
npm install nimble
```

如果用 **.series** 调度执行上面那个解方程的函数，代码应该是这样的：

```
...
var flow = require('nimble');
(function calculate(i) {
    if(i === l-1) {
        variables[i] = res[i];
        process.exit();
    }else {
        flow.series([
            function (callback) {
                calculateTail(res[i],res[i+1],function(tail) {
                    variables[i] = tail;
                    callback();
                });
            },
            function (callback) {
                calculateHead(res[i],res[i+1],function(head) {
                    res[i+1] = head;
                    callback();
                });
            },
            function(callback){
                calculate(i+1);
            }]);
    }
})(0);
...
```

**.series** 数组参数中的函数会挨个执行，只是我们的 **calculateTail** 和 **calculateHead** 都被包在了另一个函数中。尽管这个用流程控制实现的版本代码更多，但通常可读性和可维护性要强一些。接下来我们介绍 **Promise**。

## Promise

什么是 **Promise** 呢？在纸牌屋的第一季第一集中，当琳达告诉安德伍德不能让他做国务卿后，他说：“所谓 **Promise**，就是说它不会受不断变化的情况影响。”

**Promise** 不仅去掉了嵌套，它连回调都去掉了。因为按照 **Promise** 的观点，回调一点也不符合函数式编程的精神。回调函数什么都不返回，没有返回值的函数，执行它仅仅是因为它的副作用。所以用回调函数编程天生就是指令式的，是以副作用为主的过程的执行顺序，而不是像函数那样把输入映射到输出，可以组装到一起。

最好的函数式编程是声明式的。在指令式编程中，我们编写指令序列来告诉机器如何做我们想做的事情。在函数式编程中，我们描述值之间的关系，告诉机器我们想计算什么，然后由机器（底层框架）自己产生指令序列完成计算。**Promise** 把函数的结果变成了一个与时间无关的值，就像算式中的未知数一样，可以用它轻松描述值之间的逻辑计算关系。虽然要得出一个函数最终的结果需要先计算出其中的所有未知数，但我们写的程序只需要描述出各未知数以及未知数和已知数之间的逻辑关系。而 **CPS** 是手工编排控制流，不是通过定义值之间的关系来解决问题，因此用回调函数编写正确的并发程序很困难。比如在代码清单 2 中，**caculateHead** 被放在 **caculateTail** 的回调中执行，但实际上在计算同一组值时，两者之间并没有依赖关系，只是进入下一轮计算前需要两者都给出结果，但如果不用回调嵌套，实现这种顺序控制比较麻烦。

当然，这和我们的处理方式（共用数组）有关，就这个问题本身而言，**caculateHead** 完全不依赖于任何 **caculateTail**。

这里用的 **Promis** 框架是著名的 **Q**，可以用 `npm install q` 安装。虽然可用的 **Promis** 框架有很多，但在它们用法上都大同小异。我们在这里会用到其中的三个方法。

第一个负责将 **Node.js** 的 **CPS** 函数变成 **Promise**。**Node.js** 核心库和第三方库中有非常多的 **CPS** 函数，我们的程序肯定要用到这些函数，要解决回调大坑，就要从这些函数开始。这些函数的回调函数参数大多遵循一个相同的模式，即函数签名为 `function(err, result)`。对于这种函数，可以用简单直接的 **Q.nfcall** 和 **Q.nfapply** 调用这种 **Node.js** 风格的函数返回一个 **Promise**：

```
return Q.nfcall(FS.readFile, "foo.txt", "utf-8");
return Q.nfapply(FS.readFile, ["foo.txt", "utf-8"]);
```

也可以用 **Q.denodeify** 或 **Q.nbind** 创建一个可重用的包装函数，比如：

```
var readFile = Q.denodeify(FS.readFile);
return readFile("foo.txt", "utf-8");

var redisClientGet = Q.nbind(redisClient.get, redisClient);
return redisClientGet("user:1:id");
```

第二个是 **then** 方法，这个方法是 **Promise** 对象的核心部件。我们可以用这个方法从异步操作中得到返回值（履约值），或抛出的异常（拒绝的理由）。**then** 方法有两个可选的参数，分别对应 **Promis** 对象的两种执行结果。成功时调用的 **onFulfilled** 函数，错误时调用 **onRejected** 函数：

```
var promise = asyncFun()
promise.then(onFulfilled, onRejected)
```

**Promise** 被解决时（异步处理已经完成）会调用 **onFulfilled** 或 **onRejected**。因为只会有一种可能，所以这两个函数中仅有一个会被触发。尽管 **then** 方法的名字让人觉得它跟某种顺序化操作有关，并且那确实是它所承担的职责的副产品，但你真的可以把它当作 **unwrap** 来看待。**Promise** 对象是一个存放未知值的容器，而 **then** 的任务就是把这个值从 **Promise** 中提取出来，把它交给另一个函数。

```
var promise = readFile()
var promise2 = promise.then(readAnotherFile, console.error)
```

这个 **promise** 表示 **onFulfilled** 或 **onRejected** 的返回结果。既然结果只能是其中之一，所以不管是什么结果，**Promise** 都会转发调用：

```

var promise = readFile()
var promise2 = promise.then(function (data) {
  return readAnotherFile() // if readFile was successful, let's readAnotherFile
}, function (err) {
  console.error(err) // if readFile was unsuccessful, let's log it but still
readAnotherFile
  return readAnotherFile()
})
promise2.then(console.log, console.error) // the result of readAnotherFile
因为 then 返回的是 Promise, 所以 promise 可以形成调用链, 从而避免出现回调大坑:
readFile()
  .then(readAnotherFile)
  .then(doSomethingElse)
  .then(...)

```

第三个是 `all` 和 `spread` 方法。我们可以把几个 `Promise` 放到一个数组中, 用 `all` 将它们变成一个 `Promise`, 而 `spread` 跟在 `all` 后面就相当于 `then`, 只是它同时接受几个结果。如果数组中的 N 个 `Promise` 都成功, 那 `spread` 的 `onFulfilled` 参数就能收到对应的 N 个结果; 如果有一个失败, 它的 `onRejected` 就会得到第一个失败的 `Promise` 抛出的错误。

下面是用 Q 改写的解方程程序代码:

```

.....
var Q = require('q');
var qTail = Q.denodeify(calculateTail);
var qHead = Q.denodeify(calculateHead);
(function calculate(i) {
  Q.all([
    qTail(res[i],res[i+1]),
    qHead(res[i],res[i+1]))]
  .spread(function(tail,head){
    variables[i] = tail;
    res[i+1] = head;
    return i+1;
  })
  .then(function(i){
    if(i === l-1) {
      variables[i] = res[i];
      process.exit();
    }else {
      calculate(i);
    }
  });
})();
function calculateTail(x,y,cb) {
  setTimeout(function(){
    var tail = (x-y)/2;
    cb(null,tail);
  },300);
}
function calculateHead(x,y,cb) {
  setTimeout(function(){
    var head = (x+y)/2;
    cb(null,head);
  },400);
}
...

```

注意 `calculateTail` 和 `calculateHead` 中的 `cb` 调用, 为了满足 `denodeify` 的要求,

我们给它增加了值为 `null` 的 `err` 参数。此外还用到了上面提到的 `denodeify`、`all` 和 `spread`、`then`。其实除了流程控制，`Promise` 在异常处理上也比回调做得好。甚至有些开发团队坚决反对在代码中使用 `CPS` 函数，将 `Promise` 作为编码规范强制推行。

再来看一下那个找最大文件的例子用 `Promise` 实现的样子：

```
var fs = require('fs')
var path = require('path')
var Q = require('q')
var fs_readdir = Q.denodeify(fs.readdir) // [1]
var fs_stat = Q.denodeify(fs.stat)
module.exports = function (dir) {
  return fs_readdir(dir)
    .then(function (files) {
      var promises = files.map(function (file) {
        return fs_stat(path.join(dir, file))
      })
      return Q.all(promises).then(function (stats) { // [2]
        return [files, stats] // [3]
      })
    })
    .then(function (data) { // [4]
      var files = data[0]
      var stats = data[1]
      var largest = stats
        .filter(function (stat) { return stat.isFile() })
        .reduce(function (prev, next) {
          if (prev.size > next.size) return prev
          return next
        })
      return files[stats.indexOf(largest)]
    })
}
}
```

这时这个模块的用法变成了：

```
var findLargest = require('./findLargest')
findLargest('./path/to/dir')
  .then(function (er, filename) {
    console.log('largest file was:', filename)
  })
  .fail(function(err){
    console.error(err);
  });
});
```

因为模块返回的是 `Promise`，所以客户端也变成了 `Promise` 风格的，调用链中的所有异常都可以在这里捕获到。不过 `Q` 也有可以支持回调风格函数的 `nodeify` 方法。

## generators

`generator` 科普

在计算机科学中，`generator` 实际上是一种迭代器。它很像一个可以返回数组的函数，有参数，可以调用，并且会生成一系列的值。然而 `generator` 不是把数组中的值都准备好然后一次性返回，而是一次 `yield` 一个，所以它所需的资源更少，并且调用者可以马上开始处理开头的几个值。简言之，`generator` 看起来像函数，但行为表现像迭代器。

**Generator** 也被称为半协程，是协程的一种特例（比协程弱），它总是把控制权交回给调用者（同时返回一个结果值），而不是像协程一样跳转到指定的目的地。如果说得具体一点儿，就是虽然它们两个都可以 `yield` 多次，暂停执行并允许多次进入，但协程可以指定 `yield` 之后的去向，而 `generator` 不行，它只能把控制权交回给调用者。因为 `generator` 主要是为了降低编写迭代器的难度的，所以 `generator` 中的 `yield` 语句不是用来指明程序要跳到哪里去的，而是用来把值传回给父程序的。

2008 年 7 月，Eich 宣布开始 `ECMAScript Harmony`（即 `ECMAScript 6`）项目，从 2011 年 7 月开始，`ECMAScript Harmony` 草案开始定期公布，预计到 2014 年 12 月正式发布。`generator` 就是在这一过程中出现在 `ECMAScript` 中的，随后不久就被引入了 V8 引擎中。

`Node.js` 对 `generator` 的支持是从 v0.11.2 开始的，但因为 `Harmony` 还没正式发布，所以需要指明 `--harmony` 或 `--harmony-generator` 参数启用它。我们用 `node --harmony` 进入 REPL，创建一个 `generator`:

```
function* values() {
  for (var i = 0; i < arguments.length; i++) {
    yield arguments[i];
  }
}
```

注意 `generator` 的定义，用的是像函数可又不是函数的 `function*`，循环时每次遇到 `yield`，程序就会暂停执行。那么暂停后，`generator` 何时会再次执行呢？在 REPL 中执行 `o.next()`:

```
>var o = values(1, 2, 3);
>o.next();
{ value: 1, done: false }
>o.next();
{ value: 2, done: false }
>o.next();
{ value: 3, done: false }
>o.next();
{ value: undefined, done: true }
>
```

第一次执行 `o.next()`，返回了一个对象 `{ value: 1, done: false }`，执行到第四次时，`value` 变成了 `undefined`，状态 `done` 变成了 `true`。表现得像迭代器一样。`next()` 除了得到 `generator` 的下一个值并让它继续执行外，还可以把值传给 `generator`。有些文章提到了 `send()`，不过那是老黄历了，也许你看这篇文章的时候，本文中也有很多内容已经过时了，IT 技术发展得就是这样快。我们再看一个例子，还是在 REPL 中：

```
function* foo(x) {
  yield x + 1;
  var y = yield null;
  return x + y;
}
```

再次执行 `next()`:

```
>var f = foo(2);
>f.next();
{ value: 3, done: false }
>f.next();
```

```
{ value: null, done: false }
> f.next(5);
{ value: 7, done: true }
```

注意最后的 `f.next(5)`, `value` 变成了 7, 因为最后一个 `next` 将 5 压进了这个 `generator` 的栈中, `y` 变成了 5。如果要总结一下, 那么 `generator` 就是:

1. `yield` 可以出现在任何表达式中, 所以可以随时暂停执行, 比如 `foo(yield x, yield y)` 或在循环中。
2. 调用 `generator` 只是看起来像函数, 但实际上是创建了一个 `generator` 对象。只有调用 `next` 才会再次启动 `generator`。`next` 还可以向 `generator` 对象中传值。
3. `generator` 返回的不是原始值, 而是有两个属性的对象: `value` 和 `done`。当 `generator` 结束时, `done` 会变成 `true`, 之前则一直是 `false`。

可是 `generator` 和回调大坑有什么关系呢? 因为 `yield` 可以暂停程序, `next` 可以让程序再次执行, 所以只需稍加控制, 就能让异步回调代码顺序执行。

## 用 `generator` 平坑

`Node.js` 社区中有很多借助 `generator` 实现异步回调顺序化的库, 比如 `suspend`、`co` 等, 不过我们重点介绍的还是 `Q`。它提供了一个 `spawn` 方法。这个方法可以立即运行一个 `generator`, 并将其中未捕获的错误发给 `Q.onerror`。比如前面那个解方程的函数, 用 `spawn` 和 `generator` 实现就是:

```
....
(function calculate(i) {
    Q.spawn(function* () {
        i = yield Q.all([qTail(res[i],res[i+1]),qHead(res[i],res[i+1])])
        .spread(function(tail,head){
            variables[i] = tail;
            res[i+1] = head;
            return i+1;
        })
        if(i === l-1) {
            variables[i] = res[i];
            console.log(i + ":" + variables[i]);
            process.exit();
        }else {
            calculate(i);
        }
    });
})(0);
...
```

代码和前面用 `Promise` 实现时并没有太大变化, 只是去掉了 `then`, 看起来更简单了。不过记得执行时要用 `>=v0.11.2` 版本的 `Node.js`, 并且要加上 `--harmony` 或 `--harmony-generator`。你会看到和前面相同的结果。至于寻找最大文件那个例子, 在 `spawn` 里定义一个 `generator`, 然后在有 `then` 的地方放上 `yield` 就行了。具体实现就交给你了。

由于篇幅所限, 本文没有展开介绍 `Promise` 和 `generator` 在错误处理上的优势, 这在实际工作中也是很重要的部分, 应该认真研究。参考资料 7 中对此介绍得比较详细, 建议认真阅读。

## 参考资料

[Managing Node.js Callback Hell with Promises, Generators and Other Approaches](#), Marc Harter, 2014.02.03

在 Node.js 中用 Q 实现 Promise – Callbacks 之外的另一种选择

指令式 Callback, 函数式 Promise: 对 node.js 的一声叹息

[How-to Compose Node.js Promises with Q](#), Marc Harter, 2013.08.14

[Analysis of generators and other async patterns in node](#)

[generators in v8](#), Andy Wingo, 2013.05.08

[A Study on Solving Callbacks with JavaScript Generators](#), James Long, 2013.06.05

[V8 JavaScript Engine-Issue 2355:Implement generators](#)

本文源码

原文: <http://www.infoq.com/cn/articles/nodejs-callback-hell>



促进软件开发领域知识与创新的传播



中文 | 英文 | 日文 | 葡文 | .....

# 快乐 Node 码农的十个习惯

作者 Zeke sikelianos , 译者 吴海星

从问世到现在将近 20 年，JavaScript 一直缺乏其它有吸引力的编程语言，比如 Python 和 Ruby，的很多优点：命令行界面，REPL，包管理器，以及组织良好的开源社区。感谢 Node.js 和 npm，现如今的 JavaScript 鸟枪换炮了。Web 开发者有了强大的新工具，接下来就看他们的想象力了。

下面这个提示和技巧清单，能让你和你的 node 程序保持快乐。

## 1. 用 npm init 开始新项目

npm 有个 init 命令，可以引导你完成创建 package.json 文件的过程。即便你非常熟悉 package.json 和它的属性，也可以把 npm init 当作将你的新程序或模块导入正轨的简便办法。它可以聪明地为你设置默认值，比如通过上层目录的名称推断模块名，从 ~/.npmrc 中读取创作者的信息，以及用你的 git 设置确定代码库。

```
mkdir my-node-app
cd my-node-app
npm init
```

## 2. 声明所有依赖项

在将模块安装到项目本地时坚持使用 --save ( 或 --save-dev ) 是个好习惯。这些选项会将指定的模块添加到 package.json 的 dependencies ( 或 devDependencies ) 清单中，并使用合理的默认 semver 范围。

```
npm install domready --save
```

注意，现在 npm 使用插入符风格的 semver 范围：

```
"dependencies": {
  "domready": "^1.0.4"
}
```

## 3. 指定启动脚本

在 package.json 中设定 scripts.start，你就可以在命令行中用 npm start 启动程序了。这个非常方便，因为克隆了你的程序的其他 node 开发人员不用猜就能轻松运行它。

额外奖励：如果在 package.json 中定义了 scripts.start，你就不需要 Procfile 了 (Heroku 平台用 Procfile 来声明在你程序的 dynos 上运行什么命令)。使用 npm start 会自动创建一个作为 web 进程的 Procfile。这里有个启动脚本示例：

```
"scripts": {
  "start": "node index.js"
}
```

## 4. 指定测试脚本

就像团队中的所有人都应该可以运行程序一样，他们也应该可以测试它。`package.json` 中的 `scripts.test` 就是用来指定运行测试套件的脚本的。如果你用 `mocha` 之类的东西运行测试，一定要确保把它包含在 `package.json` 里的 `devDependencies` 中，并且指向安装在你项目本地的文件，而不是全局安装的 `mocha`：

```
"scripts": {
  "test": "mocha"
}
```

## 5. 不要把依赖项放在源码的版本控制中

很多 `node` 程序使用的 `npm` 模块带有 C 语言写的依赖项，比如 `bson`、`ws` 和 `hiredis`，这些依赖项必须在 `Heroku` 的 64 位 `Linux` 架构下进行编译。编译过程可能非常耗时。为了让构建过程尽可能的快，`Heroku` 的 `node buildpack` 在下载和编译完依赖项后会缓存它们，以便在后续部署中重用。这个缓存是为了降低网络流量并减少编译次数。

忽略 `node_modules` 目录也是模块创作者推荐的 `npm` 实践。应用程序和模块之间少了一个区别！

```
echo node_modules >> .gitignore
```

## 6. 用环境变量配置 `npm`

以下内容摘自 `npm` 配置：

所有以 `npm_config_` 开头的环境变量都会被解释为配置参数。比如说环境中有 `npm_config_foo=bar` 时，会将配置参数 `foo` 设置为 `bar`。任何没有给出值的环境配置的值都会设置为 `true`。配置值对大小写不敏感，所以 `NPM_CONFIG_FOO=bar` 也一样。

最近在所有的 `Heroku` 构建中都有程序的环境。这一变化让 `Heroku` 上的 `node` 用户无需修改程序代码就可以控制他们的 `npm` 配置。习惯 #7 是这一方式的完美例证。

## 7. 带着你自己的 `npm` 注册中心

最近几年公共 `npm` 注册中心出现了突飞猛进式的增长，因此会偶尔不稳定。所以很多 `node` 用户开始寻求公共注册中心之外的方案，他们或者是出于开发和构建过程中速度及稳定性方面的考虑，或者是因为要放置私有的 `node` 模块。

最近几个月冒出了一些可供选择的 `npm` 注册中心。`Nodejitsu` 和 `Gemfury` 提供收费的私有注册中心，此外也有一些免费的，比如 `Mozilla` 的只读 `S3/CloudFront` 镜像和 `Maciej Małecki` 的欧洲镜像。

在 `Heroku` 上配置 `node` 程序使用定制注册中心很容易：

```
heroku config:set npm_config_registry=http://registry.npmjs.eu
```

## 8. 追踪过期的依赖项

如果你编程的时间足够长，可能已经领教过相依性地狱的厉害了。好在 `Node.js` 和 `npm` 接

纳了 **semver**, 即 语义化版本管理规范 , 设置了一个健全的依赖项管理先例。在这个方案下, 版本号和它们的变化方式传达的含义涉及到了底层代码, 以及从一个版本到下一版本修改了什么。

**npm** 有一个很少有人知道的命令, **outdated**。它可以跟 **npm update** 结合使用, 能够找出程序的那些依赖项已经过期了, 需要更新:

```
cd my-node-app
npm outdated
```

Package	Current	Wanted	Latest	Location
express	3.4.8	3.4.8	4.0.0-rc2	express
jade	1.1.5	1.1.5	1.3.0	jade
cors	2.1.1	2.1.1	2.2.0	cors
jade	0.26.3	0.26.3	1.3.0	mocha > jade
diff	1.0.7	1.0.7	1.0.8	mocha > diff
glob	3.2.3	3.2.3	3.2.9	mocha > glob
commander	2.0.0	2.0.0	2.1.0	mocha > commander

如果你做的是开源的 **node** 程序或模块, 可以看看 **david-dm**, **NodeICO** 和 **shields.io**, 你可以用这三个优秀服务所提供的图片徽章在项目的 **README** 或网站上显示生动的依赖信息。

## 9. 用 **npm** 脚本运行定制的构建步骤

随着 **npm** 生态系统的持续增长, 开发和构建过程的自动化选择也会随之增长。**Grunt** 是迄今为止 **node** 世界中最流行的构建工具, 但像 **gulp.js** 这种新工具, 以及普通的老式 **npm** 脚本也因为较轻的负载受到欢迎。

在你把 **node** 程序部署到 **Heroku** 上时, 要运行 **npm install --production** 命令以确保程序的 **npm** 依赖项会被下载下来装上。但那个命令也会做其它事情: 它会运行你在 **package.json** 文件中定义的所有 **npm** 脚本钩子, 比如 **preinstall** 和 **postinstall**。

```
{
  "name": "my-node-app",
  "version": "1.2.3",
  "scripts": {
    "preinstall": "echo here it comes!",
    "postinstall": "echo there it goes!",
    "start": "node index.js",
    "test": "tap test/*.js"
  }
}
```

这些脚本可以是行内 **bash** 命令, 或者也可以指向可执行的命令行文件。你还可以在脚本内引用其他 **npm** 脚本:

```
{
  "scripts": {
    "postinstall": "npm run build && npm run rejoice",
    "build": "grunt",
    "rejoice": "echo yay!",
    "start": "node index.js"
  }
}
```

## 10. 尝试新东西

ES6，也就是被大众称为 **JavaScript** 的 **ECMAScript** 语言规范的下一版，其工作名称为 **Harmony**。 **Harmony** 给 **JavaScript** 带来了很多振奋人心的新特性，其中很多已经出现在较新版本的 **node** 中了。

**Harmony** 实现了很多新特性，比如块作用域、生成器、代理、弱映射等等。

要在你的 **node** 程序中启用 **harmony** 的特性，需要指定一个比较新的 **node** 引擎，比如 **0.11.x**，并在启动脚本中设置 **--harmony** 选项：

```
{  
  "scripts": {  
    "start": "node --harmony index.js"  
  },  
  "engines": {  
    "node": "0.11.x"  
  }  
}
```

## 11. Browserify

客户端 **JavaScript** 有乱如麻团般的遗留代码，但那并不是语言本身的错。由于缺乏合理的依赖项管理工具，让 **jQuery-** 插件拷贝 - 粘帖的黑暗时代延续了好多年。感谢 **npm**，带着我们步入了前端振兴的年代：**npm** 注册中心像野草一样疯长，为浏览器设计的模块也呈现出了惊人的增长势头。

**Browserify** 是一个让 **node** 模块可以用在浏览器中的神奇工具。如果你是前端开发人员，**browserify** 将会改变你的人生。可能不是今天，也不是明天，但不会太久。如果你想开始使用 **browserify**，请参阅这些文章。

你有哪些习惯？

不管你已经做过一段时间 **node** 程序，还是刚刚开始，我们都希望这些小技巧能对你有所帮助。如果你有一些（健康的）**node** 习惯想要跟大家分享，请在发 **tweet** 时带上 **#node\_habits** 标签。编码快乐！

## 作者简介

本文最初由 **Zeke sikelianos** 发表在 **Heroku** 上。**Zeke** 在 **Heroku** 工作，用 **ruby** 和 **coffeescript** 编写开源软件。他从事设计师这一职业已经有 14 年了，信奉信息自由的精神，并且相信用计算机可以创造出美好的事物。

查看英文原文：[10 Habits of a Happy Node Hacker](#)

原文：<http://www.infoq.com/cn/articles/node.js-habits>

# 修复 bug 与解决问题——从敏捷到精益

作者 [Cecil Dijoux](#)，译者 邵思华

关于精益的定义有许多，但其中最令我感到鼓舞的是精益企业研究所主席 [John Shooke](#) 在它的著作《管理精益》中所描述的一段话：精益通过提高员工的水平来保证产品开发。在这个定义的基础上，这篇论文接下来解释了精益是怎样提高人员的水平的：方法就是解决问题。这一定义揭示了以下管理实践的美妙之处：仔细设计你的工作，让你能够清晰地看见所发生的问题（以及同时出现的学习机会），并在问题出现后以科学的方式解决。

在与使用敏捷方法进行软件开发的团队共同工作时，我曾经有过一些误解：起初时，我混淆了 **bug** 和问题的概念，并且确信敏捷过程就是精益，因为它能够使 **bug** 变得可见。在最后的几个月里，在我头脑中的概念开始渐渐清晰起来，回想起当初的情景，我开始相信，我所在的敏捷团队产生的 **bug**，与精益系统产生的学习机会并不是一回事：后者表明在我的团队中确实存在着质量问题，而在其它许多团队身上我也看到了同样的问题。

写这篇文章的目的，是为了描述我对 **bug** 与质量问题这一点上的思考方式是怎样逐渐变化的。这对于读者更好地理解造成 **bug** 产生的质量问题，并相应地提高绩效能够起到一些启示作用。并通过一些真实的故事描述来看清楚真正的问题所在。（先声明一点：我们并不假设所有的敏捷团队都对此问题抱有类似的误解）

## 什么是 **bug**？

在软件工业中，一个 **bug** 可以代表任何形式的系统错误（`NullPointerException`、`Http 404` 错误代码或是蓝屏……）、功能性错误（在我单击 **B** 的时候，系统本应执行 **Z**，却最终执行了 **Y**） 、性能问题以及配置错误等等。

在精益的术语中，一个 **bug** 必须能够按照下一节提到的定义进行清晰的表达，才能说它是一个问题。请相信我，我所见过的（和自己产生的） **bug** 中，95% 以上都不像是某种问题——性能问题或许是个常见的例外情况，但有趣的是，它们也是绩效的一部分，不是吗？

## 什么是问题？

让我们在这里做一个标准的定义吧。在《丰田模式：精益模式的实践》（[Toyota Way Field book](#)）这本书中，[Jeffrey Liker](#) 定义了一个问题所需的四个方面的信息：

1. 当前的实际绩效
2. 预期的绩效（标准绩效或目标绩效）
3. 以当前绩效和目标绩效之差所体现的问题严重程度
4. 问题的范围和特点

正如 [Brenée Brown](#) 在 TED 所做的一次关于漏洞的演讲中所说的一样，如果你不能评估某个漏洞，那么它就不存在。从更实用的角度来说，如果你不能解释在绩效差距上的问题所在，那么很可能是由于你并没有花足够的时间去思考它。

在开始着手解决一个问题之前，重要的一点是要清晰地表达它，花一定时间去理解它（按照精益专家 Michael Ballé 的说法：要善待它），并且克制住直奔解决方案的冲动。我们都听说过爱因斯坦的名言：“如果我只有一小时的时间去解决一个问题，我会首先用 55 分钟去思考问题，最后用 5 分钟去思考解决方案。”没有人说这是件容易的事。

在软件开发敏捷团队的情境中，绩效指标或许是一张燃尽图（表示工作量与延迟）、bug 数量、系统响应时间（质量）、客户对已提交的用户故事的评价（以总分 10 分来表示客户满意度），以及每个 Sprint 提交的用户故事（或用户故事总点数）数量（生产力）。

按照这些指标，可以有以下这些问题存在：

质量：这个页面的响应时间目标是在 500ms 以内，而在 5000 个并发用户的情况下，我们测量到的结果是 1500ms。

质量：在 Sprint 结束时仍未解决的 bug 数量（2 个，而不是 0 个）

工作量 / 延迟：我们预计这个用户故事需要 3 天时间完成，而实际上用了 8 天才完成

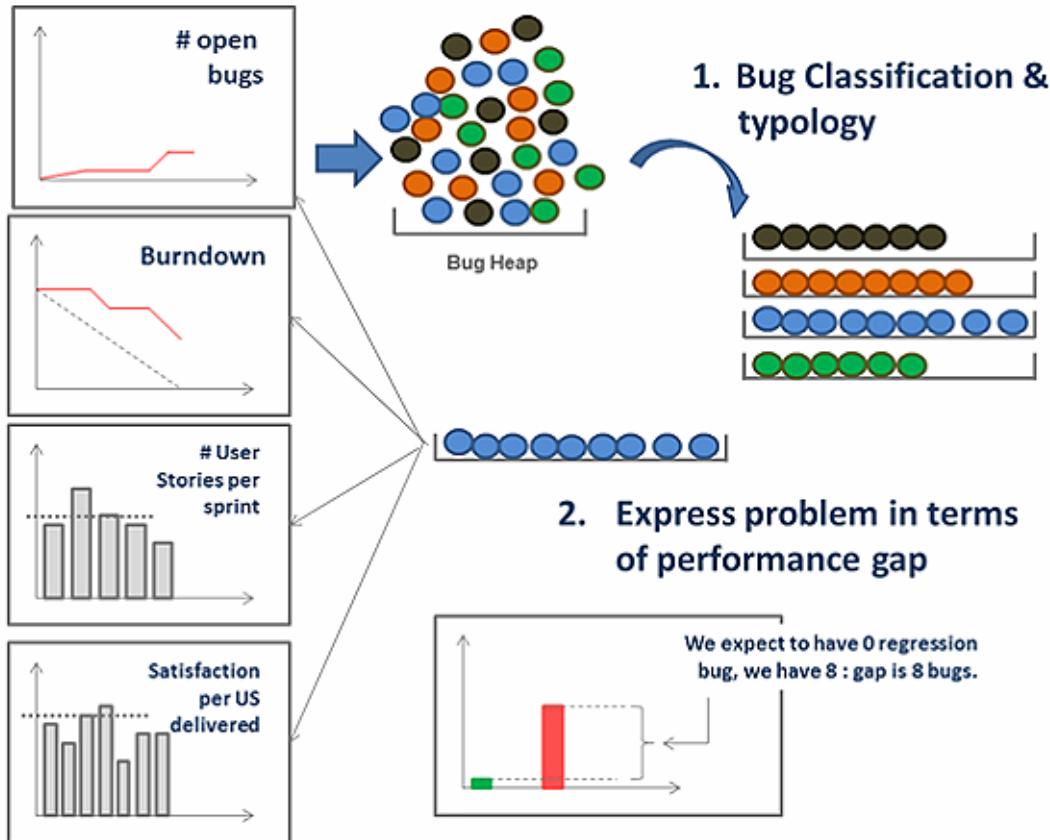
生产力：在 Sprint 结束时，整个团队共提交了 5 个完成的用户故事，而之前的计划是完成 7 个。

客户满意度：我们希望每个用户故事都能够得到 8 分以上（满分 10 分），而在上个 Sprint 结束后，有两个用户故事的客户满意度低于这个分数（6.5 分和 7 分）。

## 怎样从 bug 中分析问题所在？

Bug 的出现往往是系统中产生了更常见问题的一种症状，而对于精益团队来说，将这些症状与真正的问题相关联起来是至关重要的。可以这么说，正如米开朗基罗从大理石碎片中发现它的美丽，并最终打造成迷人的雕像一样，精益团队的任务（例如某些团队会将持续改进作为他们每日工作内容的一部分）就是发挥他们的洞察力，从大量的 bug 中发现问题，并将其抽取出来。实现这一点需要进行细致地分析，并将这些原始的问题转化为学习的机会。

我发现了一种着手进行这种分析的良好方式，就是将所有 bug 分门别类，并且理解每个 bug 类别的权重。大多数情况下，某个 bug 类别就体现了造成某个现有问题的原因，或者它本身就是一个问题。这种关联性可以帮助你以正确的顺序处理这些问题，并首先从对整个操作绩效影响最大的问题开始解决。如果你仍然不确定应该从何处着手，那么优先解决质量问题是比较保险的做法。



## 示例 1：敏捷开发中的情景

当时我在这个使用敏捷开发的团队中担任经理一职。和许多团队一样，我们团队也不是一个跨职能的团队（典型的 **Scrum-but**），而是一个负责后台的团队。它在某个迭代内负责构建基础服务端软件，以便让应用团队在之后的迭代中使用这部分功能。

我们按照 **Pareto's Principle**（即 80-20 原则）对产生的 **bug** 进行了一些分析，并且找出了一个占总数约 20% 的 **bug** 类别：这些 **bug** 都是由应用团队所提出的，与我们团队所建立的后台软件所暴露的 **API** 对“隐式”这一概念的定义有关。当应用团队在使用我们提供的功能时，经常会发生遗漏了某些输入参数，或者是缺少了某些输出数据等问题……因此他们就会为我们创建一些 **bug**，而我们的团队则会说：嘿！这个 **API** 已经隐式地表明了它不会返回这些数据。

我们同时注意到了这些 **bug** 的持续时间，通常从创建直到关闭为止一共持续了大约 4 个星期。（在最好的情况下）在以一个月为周期的迭代的最后阶段会进行代码发布，客户端团队则可以在下一个迭代时使用这些代码。因此当客户端团队创建了 **bug**，并指派给原来的开发者时，往往距离她开发那些代码时已经过去了两三个星期，开发者不得不再度拾起这段代码……

为了处理这种情况，我们决定改变一下工作的方式，将相关人员组织在一起，而产生一个相关联、跨职能并且跨技能的团队。

采用了新的方式之后，我们注意到这些“隐式 **API**”相关的 **bug** 数量大幅下降了（约 50%）。最令人欣慰的是，这种类型的 **bug** 的持续时间下降到了几个工作日以内。当然，这个数字有一定的水分，有些 **bug** 虽然被发现了，但是并没有记录下来，因为开发者们现在进行结对编程，于是许多 **bug** 直接在座位上就解决了。

虽然成果是显著的，但我总感觉到还有些不适之处，却说不出究竟是哪里出了问题。之后不

久我才发觉，从精益的角度来说，我们目前还有两个不足之处：

1. 首先，我们的系统中依然存在 **bug**，因此我们不得不重复劳动，这使得整个开发系统出现了生产力的浪费。但是由于缺乏内建的质量标准，我们无法保证服务端开发者所开发的 **API** 不存在问题。此外，对于错误的处理也没有真正的标准，我们的解决手段就是：遇到问题就坐下来一起解决。
2. 尽管结果非常显著且令人振奋，但它与团队的每日绩效没有什么直接的关联，团队也无法立即采取行动并在第二天直接看到结果。我们只是从宏观上在 6 个月结束后的发布中才能够看到这一效果：即在 **bug** 总数中与 **API** 相关的 **bug** 只占少数。因此我们看到：建立一个跨技能的团队确实能够在某种程度上改进质量，但我们还未能提供一种有效的方法，让我们能够每天监控它的情况，并采取相应的行动。

### 示例 2：精益开发中的情景

时间转眼间过去了几年，我还是任职于同一家公司中，但目前的职位是项目主管及教练，负责一个大型的多团队、多种技术的敏捷项目的实施。某一个团队遇到了一个很有挑战的技术难题，他们要与某个大家都没有什么经验的技术进行整合。整个团队在过去的两个 **Sprint** 中没有交付任何用户故事，他们深陷于质量问题（例如 **bug**）中难以自拔。当第二个 **Sprint** 结束后，依然没有任何完成的用户故事（比方说，按照我们对完成的定义来看，该用户故事在功能性需求上需要做到没有任何 **bug**）可以交付。因此在回顾会议中，整个团队一致决定，将每周进行 **bug** 评审（在精益中称为红箱分析）。

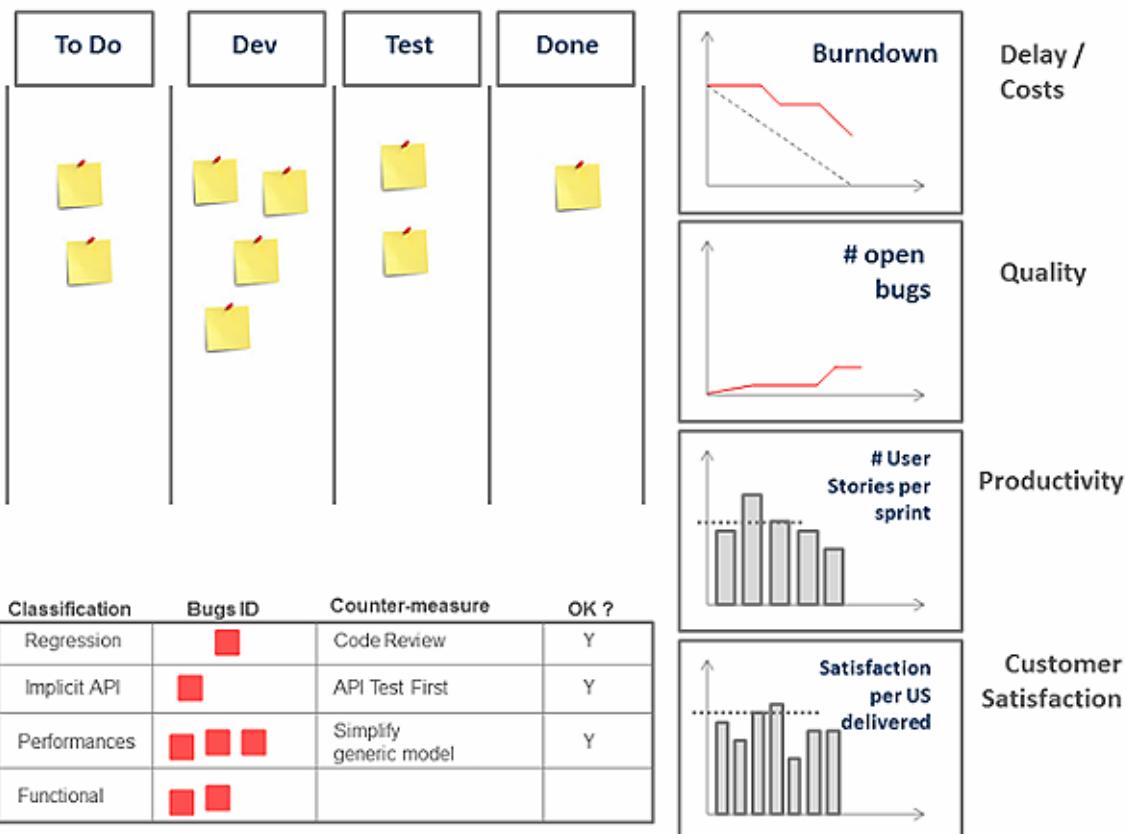
在第一次会议中，团队为所遇到的问题建立了一个 **Pareto** 模型。我们创建了一张表格，将 **bug** 类别放在一列里，**bug** 的数量和 **bug ID** 则分别用余下的几列来表示。

之后的目标是逐个排除每种 **bug** 类别背后的根本问题，首先从发生次数最频繁的开始。为了鼓励团队成员就这一话题展开交流，**Scrum Master** 决定将这张 **Pareto** 表格贴在 **Scrum** 公告板与 **bug** 数量的旁边，并且每天对其进行更新。在每天早上的站立会议上，团队都会报告当前的 **bug** 情况，而新产生的 **bug** 都会按照其分类添加到该表格中。这种方式能够使团队更明显地意识到每日质量性能的变化情况，同时也是实现 **PDCA** 中的 C——**Check**（检验）的一种良好方式。当问题被根除之后，这方面的 **bug** 应该至少在一周之内不复存在了。不过，某些时候还是会发生在这些 **bug**，而这也是需要学习的地方。

举一个例子，该团队已经认识到了 **bug** 类别中有一种属于回归缺陷，即对软件的改动破坏了原本能够正常工作的特性。这种 **bug** 多数情况下发生在图形用户界面端，因为对这一部分进行自动化测试是非常困难的事。我们所找出的一个根本问题在于，初级程序员并不总是完全理解他们对代码的改动可能会造成的影响。对此问题的解决措施是在流程中加入一个新的步骤，在提交代码之前先让某个更资深的开发者进行代码复审。这一步骤大概只需要 15 分钟，但能够大幅降低回归缺陷出现的次数。此外还将对每次发布的 **bug** 数量进行每日评估（每天发布两次）。这种方式还能够提高初级开发者的技能水平。

最终，所有的问题都得到了解决，结果是令人惊叹的：所有的问题都通过标准流程（在提交代码之前进行代码复审）得以一一根除。每日的 **bug** 数量直线下降，每个迭代周末能够提交的包括完整功能并且无 **bug** 的用户故事数量也在上升。3 个月之后，该团队就从之前产生 **bug** 数量最多的困境中摇身一变，成为了整个项目中高质量、高效率团队的代名词。

这种方式相比之前的方法显得更为精益。因为它对每日绩效（质量）和生产力（提交的用户故事数量）产生了直接的影响，并且为团队带来了新的操作标准。



## 将一个敏捷团队转变为学习团队

经历过了以上两个示例之后，加上我从这次经历中所学到的经验，我将为你推荐一种将敏捷团队转变为精益和学习团队的路线图：

- 对绩效进行评估，让它可为众人所见，并且每天都要对它展开讨论。我能够理解这一点对于某些非主流的敏捷教练来说是难以忍受的，但事实可能会令你感到沮丧：如果我们要进行改进，那么首先要做的第一件事就是评估。此外，最重要的一点是，只有面对现实，才能进行深刻的学习。网络巨擎（谷歌、亚马逊、Twitter 及 Facebook）或者实践领导者（Etsy）都是这样做的：他们对每件事情都进行评估，如果他们仅仅关注于计算用户故事的点数，就不可能达到如今的绩效。在敏捷团队方面有个实际的例子可供参考：除了 Sprint 燃尽图之外，还要展示质量绩效（没有关闭的 bug 数量、每次发布的 bug 数量、每种类别的 bug 数量，等等）、客户满意度（例如对交付的用户故事按照总分 10 分进行评分），并且每天都对燃尽图没有达到预期目标的原因进行分析。
- 确保使用精益的方式表达问题 对于某个问题的表达必须包含两个方面：所观察到的绩效和目标绩效。Pareto 是一种将原始的 bug 进行分类处理的优秀工具，但还要专门进行分析，以理解每个类别是如何影响到绩效的。
- 这种方式可以保证你已经清晰地为划分了问题的类型，并且从商业绩效的角度以正确的次序分别进行处理。
- 当问题出现时逐一分析解决 精益式解决问题方法的关键之一，就在于不要试图同时解决多个问题。你只需要专注于一个问题，理解它如何影响你的绩效指标，并确保你理解造成该问题的原因所在。

进行校验 很遗憾，根据我的经验来看，我们通常会倾向于忽略这一步骤。如果你的预估与现实不符、你的软件不能正常工作，那很好！你是否可以从中学到些什么？如果你所想象中会发生的事与实际发生的事产生了偏差，那这一段偏差就是可以从中进行学习的地方。这正是在第二个示例中的团队所做的事。正如 **Stephen J. Spear** 在他的著作《Chasing the Rabbit》中所写的一样，这是你的组织中的系统在向你发出的一种声音：“在我身上还有一些你所不了解的东西，但如果你愿意倾听，我就会告诉你。”团队正是这样才能够从工作与流程中快速地培养自己的专业技能，并真正地成为一支梦想中的团队。

## 从敏捷到精益

我从 2004 年开始成为一名敏捷实践者，而在过去的几年中，我的思维方式渐渐转为精益。正是它帮助我跨越了一些单纯依靠敏捷无法跨过的障碍。

按我的经验来看，精益已经被证明是一种有效的手段，它能够帮助你超越敏捷，建立起一种持续改进的实践，并为团队带来直接的绩效提高和激励作用。而明确地区分 **bug** 与问题这一方式已经被证实是对持续改进的一大助力。

如果你也开始了这一相同的过程，你是否能指出 **bug** 与问题之间有哪些关键的区别因素吗？

## 关于作者

**Cecil Dijoux** 在 **Operae Partners** 担任精益 IT 教练，他是一位具有 25 年国际经验的 IT 专家。他对探索 21 世纪的管理方式（精益、敏捷、Enterprise 2.0 等等）充满热情。**Cecil** 在他的博客 <http://thehypertextual.com/> 中写了大量有关于互联世界中的组织文化方面的文章，有法语和英语两个版本。纽约时报在线和 **Read Write Enterprise** 曾经专门报道过他的一篇博客文章。**Cecil** 还是一位国际性的演讲者，并且还是一份法语电子书“#hyperchange - petit guide de la conduite du changement dans l'économie de la connaissance”的作者，书中主要描写了关于知识经济时代的管理方式的变化。

查看英文原文：[Bug Fixing Vs. Problem Solving - From Agile to Lean](#)

原文：<http://www.infoq.com/cn/articles/bug-fixing-problem-solving>

# 不要就这么放弃了 SQL

作者 Lukas Eder , 译者 石岩

**SQL** 这门语言已经稳定地发展了的 20 多年。同时, **Java** 客户端代码中冗长的 **JDBC API** 和 **Java** 语言缺乏一流的 **SQL** 支持的情况, 使得 **ORM** 开发框架开始渐渐占据一席之地, 例如后来作为标准加入到 **JPA** 和 **Criteria API** 中的 **Hibernate**。然而, 出于性能上和表述上的考虑, 一些用户还通过复杂的 **SQL** 查询语句和数据库进行交互 - 这种复杂性和 **JPA** 中所覆盖到的特性是相悖的。如果 **SQL** 和 **JPA** 越来越背离, 我们的数据交互模式应该何去何从?

## 最近的 **TopConf 2013** 大会对我的影响

软件大会是我们行业在未来的 5 到 10 年将如何进展的新趋势和新思维的指向标。**Topconf** 是一场在美丽的爱沙尼亚, 塔林新举办的令人期待的大会, 相当多的技术演讲都是关于大规模、分布式数据处理, 以及新兴的技术和范例。第二天的开场主题演讲是由 **GridGain** 公司的创始人和 **CEO Nikita Ivanov** 所做的一场关于记忆体内运算方面的演讲。之后不久来自 **Hazelcast** 的 **Christoph Engelbert** 举办了一个比赛演讲, 不少的演讲都是关于异步编程和响应式编程, 后者是一种新兴的范式, 起源于 **Scala** 的“**subculture**”, 通过共用服务器上的可计算资源, 来处理水平伸缩的情况。

无论何时, 只要“垂直伸缩”和“水平伸缩”这样时髦的名词出现, 一些广为人知的 **NoSQL** —— 比如 **MongoDB** 或者 **Cloudera (Hadoop)** 也会跟随出现。**Nikita Ivanov** 和 **Christoph Engelbert** 都提到了一个很好的观点, 当我们在考虑伸缩问题的时候, 这一观点将对我们的想法起到更真实和实际的影响。

## 垂直伸缩还是水平伸缩? 集中化还是分布式?

首先, **DRAM** 的价格在过去几年飞速下跌, 你都不敢想象十年前的记忆体内计算 (那时候大概 1 美元 / **MB**) , 而现在很多公司有能力把他们所有的在线交易数据全放到内存中 (大概 1 美元 / **GB**) 。就像 **Nikita** 说的, 99% 的公司都没有 9**TB** 大的在线数据量, 这是在可支持的范围内的。这就允许把大量的数据从磁盘中 (毫秒级别的访问) 转移到内存中 (纳秒级别的访问), 而不用实际上改变软件架构! 换句话来说, 不需要使用什么新奇的技术, 就可以使遗留系统得到量级的提速。这对正在从 **COBOL** 迁移至 **Java**, 而又不太情愿替换关系数据库的大公司非常重要。再换句话来说, 也许实际上我们并不需要新的数据存储范式。不是吗?

另一方面, 数据量仍然在不断变得越来越大, 而且人们想进行数据分片和数据分布式, 物理上让数据更接近用户。即便是真的有这么一台可以计算的机器, 可以迅速地处理越来越大的记忆体内计算, 将巨大的数据通过网线传输, 也会连续遇到网络延迟、包丢失和别的原因导致的性能问题等方面所造成的困扰。

**Hazelcast** 实施了一种“新的”范式, 即把计算转向数据, 而不是把数据转向计算。这也是 **OLAP** 数据库一直以来长时间在做的事情。现代的 **RDBMS** 融入基于 **SQL** 的语言, 在接近数据的位置执行存储过程, 这种语言很复杂而且很具有表述力。实际上, 这甚至很像早期的共享主机系统, 比如 **IBM 7094**, 那时候开发者在中央计算引擎上分享时间槽。所以这并不是新的范式。我们只是在体验分布式和集中化计算的周期振荡, 基于可用的资源和这些资源的力量。如果说 **NoSQL** 是分布式的强大力量, 那么下降的 **DRAM** 价格就是集中化的强大力量。

## 同时，企业级 Java 中发生了什么？

这些主题是怎么关联到最近 Charles Humble 在 InfoQ 的调查，关于我们怎么在 Java 中访问关系数据库的呢？根据他和 Martin Fowler 的观察，通常人们似乎对 ORM 变得越来越不满了。这些情感变化很大一部分是因为 ORM 会发生是一种泄漏的抽象（Leaky Abstractions）的这一事实。Joel Spolsky 很久之前就已经观察到了这一点。

最近 JEE 7 中的 JPA 2.1 标准升级引入了一些的功能，可以更好地和“高级”的数据库特性进行集成，例如存储过程。关系数据库变得更符合 SQL 的标准，这个标准在不断地朝无法用 JPA 表达式的方向发展。通过 ISO/IEC SQL:1999 标准，我们可以利用分组集和（递归式的）通用表表达式。有了 SQL: 2003 标准，我们已经有了非常复杂的开窗函数和 MERGE 语句。有了 SQL:2008 标准，我们可以执行分段的 JOIN 语句。通过 SQL:2011 标准，现在我们可以和临时数据库进行交互操作（现在 IBM DB2 和 Oracle 都已实现）。上述这些特色没有一个在 JPA 中或者别的大多数 Charles Humble 所列举的 ORM 中有所体现，除了 jOOQ——Java 建模 SQL 中一种类型安全的内部领域特定语言（在 Topconf 大会上也有相关主题的演讲）。

也就是说，当数据管理市场以快速的步伐扩大的时候，企业级 Java 社区还在慢慢地以小版本号的幅度更新着持久化 API。

## 未来会发生什么？

我们可以说的是，在未来几年软件技术会快速地变化。以下是一些将会被质疑的范式：

### 数据存储

RDBMS 已经很好地服务于我们，并且在将来继续很好地服务于我们。它们的创建基于很复杂的和很可靠的理论——关系理论，这种模型能够适用于许多问题的解决。

其它的类型的数据存储则会继续浮现，在非 RDBMS 市场中争夺新的优势。关于为什么应该选择别的数据存储而不是调整现有的，有两个基本的原因：

- 对于高度层次型的或者非结构化的数据，关系数据模型不够用。
- ACID 不能满足极端的水平伸缩情况，网络潜在问题导致在分布式系统中所有的 A-C-I-D 都变成难以解决的问题。

但是，当说到集成列存储（也被称为 NewSQL），更多的像是让老象学习新技能，关系型数据库制造者最终也会集成经过验证的 NoSQL 功能。

### 语言

命令式（和面向对象）编程在最近几年，已经受到函数式编程极大的挑战。显著的挑战者有 Scala, Javascript, C# 3.0, VB 9 和 Java 8。一个简单的解释是，函数式编程让计算逻辑更加容易地接近数据。同时，产生的副作用是期望中的，这也被 Simon Peyton Jones 自己确认，当时他在和 LINQ 的创造者 Erik Meijer 讨论编程语言的未来。

另一方面，Erik Meijer 创造的 LINQ 的人气，已经证明声明式查询语言是复杂的数据交互的一种好范式。SQL 本身在之前提到的多样的新标准上不断提升。眼下，Facebook 已经开源了他们自己的 ANSI-SQL，它是建立在 Apache Hadoop 之上的，基于自己编写

的查询语言。

可以说，这些语言范式中没有一个可以被替代。

## SQL 无所不在

在数据处理上 **SQL** 无所不在，这让我们更加了解到为什么 **Charles Humble** 发现，在使用类似 **Hibernate** 这样传统的 **ORM** 时，会让人越来越不爽的重要原因。这些工具已经解决了以下两个问题：

- CRUD 的重复性
- 缓存，可以提高磁盘访问速度。

智能缓存是很困难的，即便是实现了很多实体级别的 **SQL** 缓存功能的 **Hibernate**。**Hibernate** 或者 **JPA** 的缓存机制之所以能够现实，是因为 **ORM** 策略已经深深地和 **JPQL** 集成。由于缺乏原生查询功能，很难绕过缓存的核心机制。

但是别忘了内存变得多么的便宜？缓存大多数数据可以保证很少量地访问磁盘。如果数据不再从磁盘上访问（毫秒级），而是从内存中访问（纳秒级）会怎么样？当数据库已经把所有相关的数据缓存到内存中，我们依然需要完整的和复杂的二级缓存吗？

数据库不仅有能力把在线交易数据一直放在内存中，而且它们早就已经可以实现一些复杂的缓存机制了。想想 **Oracle** 的游标缓存，查询结果缓存，和标量子查询缓存吧。

虽然 **JPA** 依然是 **CRUD** 操作的优秀工具，但以 **SQL** 为中心的工具正在被越来越多的人选择使用，比如：**MyBatis** 或者 **jOOQ**（都是“后 **JPA** 框架”）表明了一种需要再次更加接近 **SQL** 的确定需求。**ANSI-SQL** 标准不断发展，把不断革新的数据库的各种功能进行标准化，比如：**Oracle**, **SQL Server**, **PostgreSQL**, 通常是那些很难集成到 **JCP** 甚至是今后的 **JPA** 版本中的一些特性。**JPA 2.1** 支持存储过程，但是在客户端代码管理的这些支持，看起来和在 **JDBC** 中实现的可调用语句（**CallableStatements**）一样单调乏味。另外，**JPA 2.1** 虽然有了各种新的与存储过程相关的注解，但并没有把嵌入式的用户自定义函数加入到 **SQL** 语句中，而这些却被不知名的 **CriteriaQuery API** 不充分的表述出来。大会上的所有关于 **jOOQ** 演讲，我都在问观众他们使用 **CriteriaQuery** 幸福吗。他们的回答是一致的不幸福。

## 总结

**RDBMS** 仍旧在不断拥抱新的功能，按照 **ANSI SQL** 进行自我标准化，并逐渐放弃 **JPA 2.x**。在这几次变化中，**JPA** 规范看起来局限了数据存储市场进行的革新。**EclipseLink** 最近通过 **JPA** 扩展支持 **MongoDB**，表明了标准制定者并不确定他们将走向何方。

但是有一件事情是确定的。我们不会这么快放弃 **SQL**。那么为什么不再次拥抱它？

查看英文原文：[Don't jump the SQL ship just yet](#)

感谢邵思华对本文的审校。

原文：<http://www.infoq.com/cn/articles/SQL-relevance-NoSQL>

封面植物

# 栀子花



灌木，高 0.3-3 米；嫩枝常被短毛，枝圆柱形，灰色。叶对生，革质，稀为纸质，少为 3 枚轮生，叶形多样，通常为长圆状披针形、倒卵状长圆形、倒卵形或椭圆形。产于山东、江苏、安徽、浙江、江西、福建、台湾、湖北、湖南、广东、香港、广西、海南、四川、贵州和云南，河北、陕西和甘肃有栽培；生于海拔 10-1 500 米处的旷野、丘陵、山谷、山坡、溪边的灌丛或林中。国外分布于日本、朝鲜、越南、老挝、柬埔寨、印度、尼泊尔、巴基斯坦、太平洋岛屿和美洲北部，野生或栽培。模式标本采自我国，具体地点不详。

本种作盆景植物，称“水横枝”；花大而美丽、芳香，广植于庭园供观赏。干燥成熟果实是常用中药，其主要化学成分有去羟栀子甙，又称京尼平甙（GENIPOSIDE）、栀子甙（Gardenoside）、黄酮类栀子素（Gardenin）、山栀子甙（Shanzhjside）等；能清热利尿、泻火除烦、凉血解毒、散瘀。叶、花、根亦可作药用。从成熟果实亦可提取栀子黄色素，在民间作染料应用，在化妆等工业中

用作天然着色剂原料，又是一种品质优良的天然食品色素，没有人工合成色素的副作用，且具有一定的医疗效果；它着色力强，颜色鲜艳，具有耐光、耐热、耐酸碱性、无异味等特点，可广泛应用于糕点、糖果、饮料等食品的着色上。花可提制芳香浸膏，用于多种花香型化妆品和香皂香精的调合剂。

栀子花含挥发油，可提取芳香浸膏，用作调香剂；根、花、叶、果可入药，果有消炎解毒的功效，主治肺热咳嗽；根可清热、利湿、凉血、解毒，主治感冒高热、黄疸型肝炎等；成熟的栀子花果实为橙红色，可作黄色染料。

促进软件开发领域知识与创新的传播

# 架构师 ARCHITECT



本期专题 | Topic | 支付宝的测试  
40 | 支付宝分布式事务测试方案  
44 | 支付宝的性能测试  
56 | 数据设计测试的分析方案  
推荐文章 | Article  
62 | 一秒钟法则：来自腾讯无线研发的经验分享  
68 | 腾讯开放平台产品消费者模式  
特别专栏 | Column  
74 | Node.js 故障之面试大坑  
86 | 快乐 Node 码农的十个习惯

InfoQ



2014年6月

## 架构师 2014.6月刊

每月 8 号出版

本期主编：侯伯薇

责任编辑：杨赛

发行人：霍泰稳

读者反馈 / 投稿：[editors@cn.infoq.com](mailto:editors@cn.infoq.com)

InfoQ 中文站新浪微博：

<http://weibo.com/infoqchina>

商务合作：[sales@cn.infoq.com](mailto:sales@cn.infoq.com) 15810407783

本期主编：侯伯薇，InfoQ 中文站译者团队主编

侯伯薇，生于丹东凤城，学在春城长春，工作在滨城大连；虽已年过而立，但自问童心未泯；对代码热情不减，愿与天下程序员共同修炼，不断提升。译有《学习 WCF》、《Expert C# 2008 Business Objects》。

