

架构师

ARCHITECT



热点 | Hot

DockerCon回顾

WWDC总结

推荐文章 | Article

NGINX引入线程池 性能提升9倍

大数据平台架构实践

专题 | Topic

京东618：Docker扛大旗

深入浅出ES6

观点 | Opinion

姜宁谈红帽绩效考核



扫我，码上开启新世界



Geekbang, 有温度的技术社区。Geek是一种精神, 也是一种态度; Geekbang是一个圈子, 也是一种习惯。在这儿, 你要么是Geek, 要么走在成为Geek的路上。



全球软件开发大会2015

International Software Development Conference

旧金山 上海 伦敦 北京 圣保罗 东京 纽约 里约热内卢

2015.10.15-17 上海·光大会展中心国际大酒店 www.qconshanghai.com

主题演讲：



畅销书《番茄工作法图解》作者
Staffan Nöteberg



Azul Systems联合创始人兼CTO
Gil Tene



Uber首席系统架构师, Voxer联合创始人
Matt Ranney

演讲专题：

机器学习	运维之痛	互联网产品案例研究
安全与风控	容器与云计算	新时代的前端
技术创业	移动开发新趋势	Java优化面面观
开源文化面面观	高可用架构	再谈软件交付
建设高效团队	新兴大数据处理	编程语言选型与实战

[上海站]
8折抢票中...

节省1360元
优惠截至8月16日

5人以上团购 更多给力优惠

Brought by **Geekbang InfoQ**
极客邦科技





全球容器技术大会

剖析容器企业实践
关注容器生态圈开源项目

2015年8月28日~29日

新云南皇冠假日酒店

www.cnutcon.com



大前端：前端与终端开发的融合

在国内，“大前端”这个词在 2011 年就已出现，但未有准确定义，并不为主流业界所接受。这里用它来描述前端与终端的融合，与后端相对，泛指在终端设备上的应用的开发。

为什么说 Web 前端与终端开发正在融合？以移动为例，根据笔者近来的观察，Web 与 Native 之间的鸿沟正在以不同的方式渐渐填补。在之前，Web 与 Native 之间只有 Hybrid，但今天，Hybrid 与 Native 之间有 React Native、Samurai Native，Web 与 Hybrid 之间有轻应用、Hosted App。一个应用里面有多少 Native 的部分，又有多少 Web 的部分，完全由开发者来决定。这些应用被操作系统、甚至应用商店一视同仁——至少在微软的应用商店里。

能够融合的原因，是因为浏览器引擎的功能和使命已经和 Native App 接近，甚至趋同，它们的目标都是为用户提供功能丰富、界面绚丽的应用。随着 HTML5 标准的推进，绝大多数 Native 应用的功能都能在 Web 中实现，在这样的大背景下，前端的专业化迅猛发展，以 yeoman、

gulp 为代表的工程化、以 React 为代表的组件化席卷整个行业，被武装起来的前端开发者试图在更多领域施展拳脚。

除此之外，Web 前端与终端开发也出现越来越多的互相借鉴现象，Web 模拟原生的努力一直在进行，而 Native 则开始借鉴 Web 中的链接和更新功能，iOS 9 的 Deep Linking 和 Android M 的 App Links 补全了 Native 应用缺失的一环，Web 与 Native 越来越相似了。

再来看桌面开发，其实它也在发生融合，越来越多的开发者开始使用 Atom-shell、node-webkit 等工具，使用 Web 技术来开发桌面应用程序。

大前端——前端与终端融合的革命正在发生。一年多之前，月影率领的 360 前端团队奇舞团已经开始“从前端到终端”；鬼道带领的天猫前端团队则在 Web 和 Native 融合上进行探索，在 React Native 发布之后更率先进行应用，取得了不错的成果。

对于前端开发者和移动开发者，也许，是停下 HTML5 vs 原生应用的无聊争论，将精力投入到这场注定影响深远的革命当中的时候了。

目 录

热点 | Hot

Docker、CoreOS 握手言和，共同制定容器标准

WWDC 总结：开发者需要知道的 iOS 9 SDK 新特性

推荐文章 | Article

NGINX 引入线程池 性能提升 9 倍

Facebook 如何向十亿人推荐东西

大数据平台架构实践

专题 | Topic

京东 618：Docker 扛大旗，弹性伸缩成重点

深入浅出 ES6（三）：生成器 Generators

深入浅出 React（二）：React 开发神器 Webpack

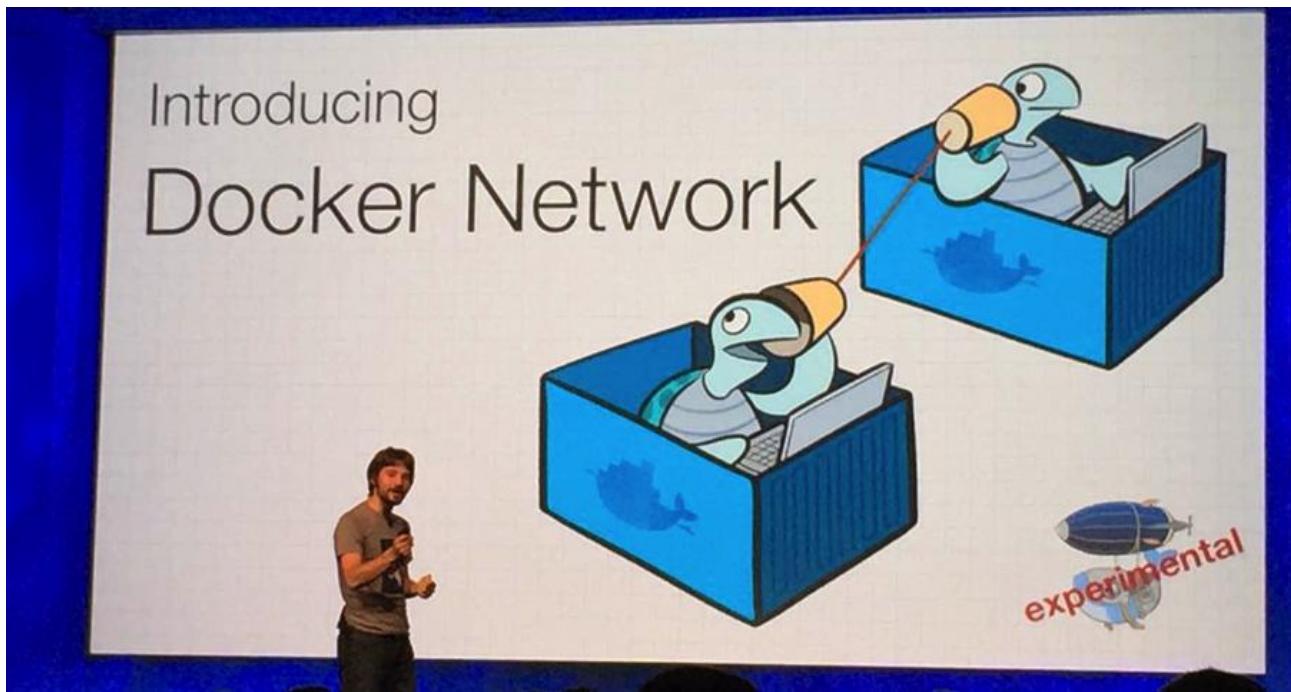
观点 | Opinion

姜宁谈红帽绩效考核：不关心员工具体做什么

冯·诺依曼计算机将渐行渐远？



Docker、CoreOS 握手言和，共同制定容器标准



陈恺，2015 年正式加盟[灵雀云](#)创业团队，任首席技术官。携其十数年大规模、企业级分布式系统/云平台研发经验，打造基于容器技术、面向开发者的云计算平台。加入灵雀云之前，2004 年在微软从事 Windows 操作系统内核（Kernel）的研发，2010 年出任微软云平台 Windows Azure 首席架构师/软件开发部经理，专注于云计算/分布式系统的研发，组建、带领团队开发 Azure 最核心的中控系统（FabricController），管理并支撑整个云平台后端，承载千万级规模应用。

美国时间 6 月 22 日，DockerCon2015 在美国旧金山举行。在大会的 Keynote 上，Docker 创始人兼 CTO Solomon Hykes 宣布：Docker 将联合 CoreOS 等公司共同创建一个完全开放的容器标准，称之为“[开放式容器项目](#)”（Open Container Project）。包括 AWS、Google、Microsoft、Redhat、VMWare、华为等超过 20 家业界巨头同时宣布支持该项目，并成为开放式容器项目创始成员。Docker 率先为 OCP 捐献出了 Docker 所基于的容器格式和运行时（Runtime）代码及文档，成为 OCP 项目的基石。同时，由 CoreOS 发起的 ApplicationContainer（appc）标准将与 OCP 整合。这也就是说 OCP 将会成为了首个业界开放、统一的容器标准。

近两年多来，Docker 引领了一场以容器为核心的，席卷互联网应用乃至整个 IT 界的技术革新，正从根本上改变应用开发和发布的方式。经过持续爆发式的成长，以及迅速壮大的开发者社区和上下游生态，Docker 容器实际上已经成为了容器格式和运行时的标准。之前，以 CoreOS 为代表的容器提供商认为，容器的标准不应该由一家公司掌控，而应更具开放性。

2014 年 12 月，CoreOS 推出自己的容器 rkt，并随后发起了一项开放式的容器标准 ApplicationContainer (appc)，该标准立刻受到了 Google、VMWare、Redhat 等重量级公司的支持。而由于有不同容器标准的存在，所以也引发了容器战争，并很有可能引起社区分裂，从而减慢容器技术的发展进程。OCP 的发布逆转了这一走势，包括 Docker、CoreOS 在内的公司将共同参与开放式委员会的管理，并统一定义容器标准。社区和生态圈合作伙伴将受益于这个开放式的标准。

Docker 创始人兼 CTO Solomon Hykes 在大会上表示，Docker 一贯的方针就是要促进开放式的标准。Docker 最大的价值不在于特定的技术，而是让所有人，包括企业和用户，对于某些标准实现统一。既然 Docker 已经成为了实际上的容器标准，那么 Docker 也肩负着完善这一标准的使命。而 Solomon 认为，完善容器标准的最好方式就是推进该标准的开放性。Solomon 对此提出以下几点原则。

- **正式的标准：**开放基于 Docker 的容器格式，即 OpenContainerFormat (开放式容器格式)，作为 OCP 容器格式定义的起点。
- **独立管理：**OpenContainerProject 会成为 LinuxFoundation 成员项目，由委员会管理。
- **中立的参考实现：**Docker 贡献出其所基于 runC 的代码。
- **受广泛认同、支持：**超过 20 家业界巨头共同参与、支持 OpenContainerProject。
- **广泛接受新的意见和建议：**由 CoreOS 主导的 ApplicationContainer (appc) 标准将与 OCP 整合，成为 OCP 项目的一员。

这一宣布对于容器技术今后发展的影响不可估量。毕竟，相对于容器具体的标准，以及这个标准由谁来定，我们更关心的是容器技术有一个统一的标准可循，这样大家可以放心参考这一标准，在容器上层做更多的，真正产生价值的创新。

在 Keynote 大会上信息量极大，除了最重磅的 OCP，还有大量新内容发布。Solomon Hykes 也是以其一贯的“哲学家”的风格，为来宾细细讲述 Docker 创始的初衷，长远的目标，以及为了实现这些目标将在近期发布的内容。

归根结底，Docker 的使命是“创建用于超大规模创新的工具”而 Solomon 认为创新的最大杠杆就是互联网应用，因此，Docker 将不遗余力地帮助开发者“提升整个互联网的可编程性”。Solomon 提到了几个层面的目标：

- 不断对开发工具进行创新；
- 做开发者的管道工；
- 推动公开的标准；
- 帮助企业用独到的方法解决实际的问题。

围绕这些目标，Docker 在本次大会上发布了一系列新产品和功能。

- 全新的容器网络（DockerNetwork）：自并购 SocketPlane 后，两个团队马不停蹄地将 SDN 与 Docker 容器进行集成，重写了整个网络模块，并将其从 Docker 代码中剥离出来。新的 Docker 网络有几个重大的改动：a) Docker 将原生支持跨主机连接；b) 应用内部可定义多个容器网络，相互之间可实现隔离；c) 支持基于 DNS 的服务发现机制，d) 已经有 11 个网络后端的插件，今后会支持更多。
- 全新的插件框架（DockerPlugins）：用户可以在保持完整的 Docker 体验的同时，通过不同插件来自定义某方面功能，并不影响与其他支持 Docker 工具的兼容性。同时，在多租户环境下，不同容器可以使用不同插件来适应各个场景的需求。目前已有的插件覆盖网络、存储、调度、服务发现等各个方面。

- 试验性发布，每日更新：为了更高效、快速地采纳来自社区的建议，并加快迭代速度，Docker 推出“Docker 试验性发布”（DockerExperimentalReleases），实现每日更新。
- DockerPlumbingProject：为了更有效、重复使用 Docker 某些基础功能层面的模块，Docker 推出“DockerPlumbingProject”，会对 Docker 代码做重构，剥离一些可以独立使用的模块，反馈到社区，用于 Docker 以外的项目。
- Notary：Docker 对安全模块进行重构，剥离出了名为 Notary 的独立项目，用于解决互联网内容发布的安全性。该项目不局限于容器应用，但在容器场景下可对镜像源认证、镜像完整性等安全需求提供很好的支持。
- runC：Docker 对基本容器实现进行重构，剥离出了最底层的容器运行时 runC。在架构层面，Docker 本身基于这个模块，但 runC 对于 Docker 没有任何依赖。更重要的是，Docker 将 runC 贡献给了随后发布的 OCP 项目，形成了该开放性容器运行时标准的基石。

第一天的 Keynote，我们看到了一个更加开放的 Docker，他将拥抱更多的厂商共建生态，尤其是 OCP (OpenContainerProject) 项目的诞生，相信未来社区和生态圈合作伙伴将受益于这个开放式的标准。

WWDC 总结：开发者需要知道的 iOS9SDK 新特性



王巍，微博 ID “[onevcat](#)”，知名 iOS/Unity 开发者，现居日本，就职于 LINE。王巍是 objc 中国项目发起人，开源过广受开发者喜爱的 Xcode 插件 VVDocumenter，主创或参与开发《姬骑士和最后的百龙战争》、《英雄 Slash》、《冒险谜题王国》以及《小熊推金币》等多款游戏，个人应用代表有《番茄工作法》(PomodoroDo) 和《云端记账》(OurMoney)。

年年岁岁花相似，岁岁年年人不同。今年的 WWDC 一如既往的热闹，得益于 Apple 的随机抽选机制，这两年有更多的中国开发者有机会亲临现场进行体验，并与全球开发者取得更多的交流。更多的开发者可能只能在家里或者公司远程关注这一全球 Apple 开发者的盛会，但是这也没有减少大家对于开发的热情。

生命不息，学习不止。从 WWDC 开始受到广大开发者的关注以来，这就是一个开发者们学习和提高的重要途径。可以感受到近年来国内开发者的平均水平越来越高，希望这样的趋势能够保持下去，毕竟只有在社区的支持下，开发者们才会是最强力的存在。

事不宜迟，让我们来看看今年的 WWDC 中开发者可能需要重点关注的一些内容吧。

总览

iOS9 时代开发者面临的最大的挑战和最急切的任务可能有两个方面，首先是如何利用和适配全新的 iPad 分屏多任务特性，其次是如何面对和利用 watchOS2 来构建原生的手表 app。另外的新课题基本就都是现有框架的衍生和扩展，包括从单元测试扩展到 UI 测试，如何进一步占

领和使用系统的通知中心及搜索页面，以及 Swift2 的使用等。

可以说，经过了 iOS7 和 iOS8 连续两次重量级的变革和更新，对普通的 app 开发者来说，iOS9SDK 略归于缓和及平静，新的 SDK 在 API 和整体设计上并没有发生像之前两个系统那样翻天覆地的改变。开发者们也正可以利用这个机会稍作喘息，在这一年里尽快熟悉和至少过渡到使用 iOS8SDK 的特性来构筑自己的 app（比如尝试使用 [SizeClass 和 PresentationController](#) 等）。尽量提升自己的职业能力和制作 app 的水平，并保证能跟上滚滚向前的 Apple 车轮，应该是今年 Cocoa 开发者们的主要任务。从近几年的 WWDC 技术路线图来看，Apple 开发可谓是环环相扣，如果哪一年你的技术停步不前，之后想要再赶上可能要付出的就是成倍的精力了。

Multitasking

这可以说是 iOS9 最大的卖点了。多任务特性，特别是分屏多任务使得 iPad 真正变得像一个堪当重任的个人电脑。虽然在很早以前就已经有越狱插件能让 iPad 同时运行多个程序，但是 Apple 还是很谨慎地到 2015 年才在自己性能最为强劲的移动设备上实装这个功能。iOS9 中的多任务分为三种表现形式，分别是临时调出的滑动覆盖（SlideOver），视频播放的画中画模式（PictureinPicture）以及真正的同时使用两个 app 的分割视图（SplitView）。现在能运行 iOS9 的设备中只有最新的 iPadAir2 支持分割视图方式，但是相信随着设备的更新，分割视图的使用方式很可能成为人们日常使用 iPad 的一种主流方式，因此提早进行准备是开发者们的必修功课。

虽然第一眼看上去感觉要支持多任务的视图会是一件非常复杂的事情，但是实际上如果你在前一年就紧跟 Apple 步伐的话，就很简单了。滑动覆盖和分割视图的 app 会使用 iOS8 引入的 SizeClass 中的 CompactWidth 和 RegularHeight 的设定，配合上 AutoLayout 来进行布局。也就是说，如果你的 app 之前就是 iPhone 和 iPad 通用的，并且已经使用了 SizeClass 进行布局的话，基本上你不需要再额外做什么事儿就已经能支持 iOS9 的多任务视图了。但是如果不幸你还没有使用这些技术的话，可能你会需要尽快迁移到这套布局方式中，才能完美支持了。

视频 app 的画中画模式相对简单一些，如果你使用 AVPlayerViewController 或者 AVPlayerLayer 来播放视频的话，那什么都不用做就已经支持了。但如果你之前选择的方案是 MPMoviePlayerController 或者 MPMoviePlayerViewController 的话，你可能也需要今早迁移到 AVKit 的框架下来，因为 MediaPlayer 将在 iOS9 被标记为 deprecated 并不再继续维护。

watchOS2

在新的 watchOS2 中，WatchApp 的架构发生了巨大改变。新系统中 WatchApp 的 extension 将不像现在这样存在于 iPhone 中，而是会直接安装到手表里去，AppleWatch 从一个单纯的界面显示器进化为了可执行开发者代码的设备。得益于此，开发者们也可以在 extension 中访问到像数

字表冠和（虽然都只是很初级的访问，但是聊胜于无）心跳计数这样的情报。虽然有所进步，但是其实 Apple 在 watchOS2 里表现出来的态度还是十分谨慎，这可能和初代 AppleWatch 的设备限制有很大关系，所以实际上留给 app 开发者的电量和性能空间并不是十分广阔。但是相比起现在的 WatchKit 来说，可以脱离 iPhone 运行本身就是了不起的进步了。而为了和 iPhone 进行通讯，现在还添加了 WatchConnectivity 这个新框架。我们有足够的理由期待 AppleWatch 和 WatchKit 在接下来两三年里的表现。

UITest

在开发领域里，测试一直是保障产品质量关键。从 Xcode4 以来，测试在 app 开发中的地位可谓逐年上升。从 XCT 框架的引入，到测试 target 成为新建项目时的默认，再到去年加入的异步代码测试和性能测试。可以说现在 Xcode 自带的测试框架已经能满足绝大部分单元测试的需求了。

但是这还不够。开发一个 iOSapp 从来都是更注重 UI 和用户体验的工作，而简单地单元测试可以很容易地保证 model 层的正确，却很难在 UI 方面有所作为。如何为一个 app 编写 UI 测试一直是 Cocoa 社区的难题之一。之前的话有像是 [KIF](#), [Automating](#), 甚至是 [FBSnapshotTestCase](#) 这种脑洞大开的方案。今年 Apple 给出了一个更加诱人的选项，那就是 Xcode 自带的 XCUI 测试的一系列工具。

和大部分已有的 UI 测试工具类似，XCUI 使用 Accessibility 标记来确定 view，但因为是 Apple 自家的东西，它可以自动记录你的操作流程，所以你只需要书写最后的验证部分就可以了，比其他的 UI 测试工具方便很多。

Swift2

Swift 经过了一年的改善和进步，现在已经可以很好地担任 app 开发的工作了。笔者自己也已经使用 Swift 作为日常工作的主要语言有半年多时间了，这半年里的总体感觉是越写越舒畅。Swift2 里主要的改动是错误处理方面的变化，Apple 从 Cocoa 传统的基于 `NSError` 错误处理方式变为了 `throwcatch` 的异常处理机制。这个转变确实可以让程序更加安全，新增的 `ErrorType` 也很好地将错误描述进行了统一。但是在实际接触了一两天之后，在语法上感觉要比原来的处理写的代码多一些。可能是长久以来使用 `NSError` 的习惯导致吧，笔者还并没有能很好地全面接受 Swift2 中的异常机制。不过这次 Apple 做的相对激进，把 CocoaAPI 中的 `error` 全数替换成 `throw`。所以不管情不情愿，转型到异常处理是 Swift 开发者必须面对的了。

另外 Apple 新加了一些像是 `guard` 和 `defer` 这样的控制流关键字，这在其他一些语言里也是很实用的特性，这让 Swift 的书写更加简化，阅读起来更流畅。为了解决在运行时的不同 SDK 的

可用性的问题，Apple 还在 Swift2 里加入了 available 块，以前我们需要自己去记忆 API 的可用性，并通过检查系统版本并进行对比来做这件事情。现在有了 available 检测，编译器将会检查出那些可能出现版本不匹配的 API 调用，app 开发的安全性得到了进一步的保障。为了让整个 SDK 更适合 Swift 的语法习惯，Apple 终于在 Objective-C 中引入了泛型。这看似是 Objective-C 的加强，但是实际上却实实在在地是为 Swift 一统 Apple 开发开路。有了 Objective-C 泛型以后，用 Swift 访问 CocoaAPI 基本不会再得到 AnyObject 类型了，这使得 Swift 的安全特性又上了一层台阶。

最后是 Swift2 开源的消息。Swift 的编译器和标准库将在今年年底开源，对于一般的 app 开发者来说可能并不会带来什么巨变，但这确实意味着 Swift 将从一门 app 制作的专用语言转型为一门通用语言。最容易想到的就是基于 Swift 的后端开发，也许我们会在看到 Javascript 一统天下之前就能先感受一下 Swift 全栈的力量？

AppThinning

笔者在日本工作，因为这边大家流量都是包月且溢出的，所以基本不会有对 app 的尺寸介意，无非就是下载 5 秒还是 10 秒的区别。但是在和国内同行交流的时候，发现国内 app 开发对尺寸的要求近乎苛刻。因为 iOSApp 为了后向兼容，现在都同时包含了 32bit 和 64bit 两个 slice。另外在图片资源方面，更是 1x2x3x 的图像一应俱全（好吧现在 1x 应该不太需要了）。而用户使用 app 时，因为设备是特定的，其实只需要其中的一套资源。但是现在在购买和下载的时候却是把整个 app 包都下载了。

Apple 终于意识到了这件事情有多傻，iOS9 中终于可以仅选择需要的内容（Slicing）下载了。这对用户来说是很大的利好，因为只需要升级到 iOS9，就可以节省很多流量。对于开发者来说，并没有太多要做的事情，只需要使用 assetcatalog 来管理素材标记 2x3x 就可以了。

给 App 瘦身的另一个手段是提交 Bitcode 给 Apple，而不是最终的二进制。Bitcode 是 LLVM 的中间码，在编译器更新时，Apple 可以用你之前提交的 Bitcode 进行优化，这样你就不必在编译器更新后再次提交你的 app，也能享受到编译器改进所带来的好处。Bitcode 支持在新项目中是默认开启的，没有特别理由的话，你也不需要将它特意关掉。

最后就是按需加载的资源。这可能在游戏中应用场景会多一些。你可以用 tag 来组织像图像或者声音这样的资源，比如把它们标记为 level1, level2 这样。然后一开始只需要下载 level1 的内容，在玩的过程中再去下载 level2。或者也可以通过这个来推后下载那些需要内购才能获得的资源文件。在一些大型游戏里这是很常见的优化方法，现在在 iOS9 里也可以方便地使用了。

人工智能和搜索 API

如果说这届 WWDCKeynote 上还有什么留给我印象深刻的内容的话，我会给更加智能的手机助理投上一票。虽然看起来还很初级，比如就是插入耳机时播放你喜欢的音乐，推荐你可能会联系的人和打开的 app 等，但是这确实是很有意义的一步。现在的 Siri 只是一个问答系统，如果上下文中断，“她”甚至不记得前面两句话说了些什么。一个不会记住 Boss 习惯的秘书一定不是一个好护士，而 Apple 正在让 iPhone 向这方面努力。好消息是我们大概暂时还不用担心会碰到故意不通过图灵测试的机器，所以在人工智能上还有很大的空间可以发挥。

而 [搜索 API](#) 实质上让 app 多了一个可能的入口。有些用户会非常频繁地使用搜索界面，这是一个绝好的展示你的 app 和提高打开率的机会。如果 app 类型合适的话，这是非常值得一做的追加特性。

游戏相关

游戏类的 app 因为在不同的移动平台上的用户体验并没有鸿沟似的差异，所以是最容易跨平台的-毕竟现在无论哪个开发商都无法忽视安卓的份额。这也是 Apple 自家的 SpriteKit 和 SceneKit 这样的游戏框架一直不温不火的原因。比起被局限在 Apple 平台，更多的开发商选择像是 Unity 或者 Cocos2d-x 这样的跨平台方案。但是今年 Apple 还是持续加强了游戏方面的开发工具支持，包括负责状态机维护和寻路等的 GameplayKit 框架，负责录像和回放游戏过程的 ReplayKit 框架，以及物理建模的 ModelIO 框架。

这些其实都是在 Apple 的游戏开发体系中补充了一些游戏业界已经很成熟的算法和工具，为开发者节省了不少时间。对于个人开发者自制的游戏来说，Apple 的工具提供了相对低的门槛，易于上手。但是在现在大部分游戏开发都需要跨平台的年代，总感觉 Apple 体系是否能顺利走下去还需要进一步观察。

其它

HomeKit, CloudKit, HealthKit 等杂七杂八的框架。如果是 iOSOnly 的 app 的话，使用 CloudKit 做 [BaaS](#) 也许是不错的选择，但是也要面临今后跨平台数据难以共享的风险。其他几个框架专业性相对较强，大部分需要配合硬件支援，其实一直说智能硬件是下一个爆点，但是至少现在为止还没能爆出大的声响，更多的却已经进入到廉价竞争（手环什么的你懂的），只能说期待这些设备的后续表现吧。

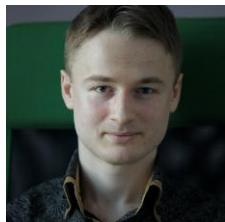
最后是一个对于刚入门或者打算投身到 Apple 开发中的朋友的福利。现在你可以不需要加入付

费的开发者计划就能将 app 部署到自己的设备上了，而在以前这至少需要你加入 99 美金每年的开发者计划，这可以说进一步降低了进行 Apple 开发的门槛。

总结

正如上面提到的，对开发者来说，今年的 WWDC 并没有像 13 年和 14 年那样颠覆性的变化，大多是对已有特性的加强补充和对开发工具链的增强。今年可以说是一个 Cocoa 开发者们沉淀之前知识，增进自己技能的好机会。现在 WWDC15 还在如火如荼的进行之中。如果你打算尽早拥抱新 SDK 的变化的话，请不要犹豫，直接访问 Apple 的[开发者网站](#)，去寻找和观看自己感兴趣的话题吧。

NGINX 引入线程池 性能提升 9 倍



Valentin Bartenev, 毕业于莫斯科物理技术学院（MIPT 州立大学），现在是 NGINX 公司的软件开发工程师。Nginx CT++模块创作者。之前从事过 8 年 Web 开发工作。



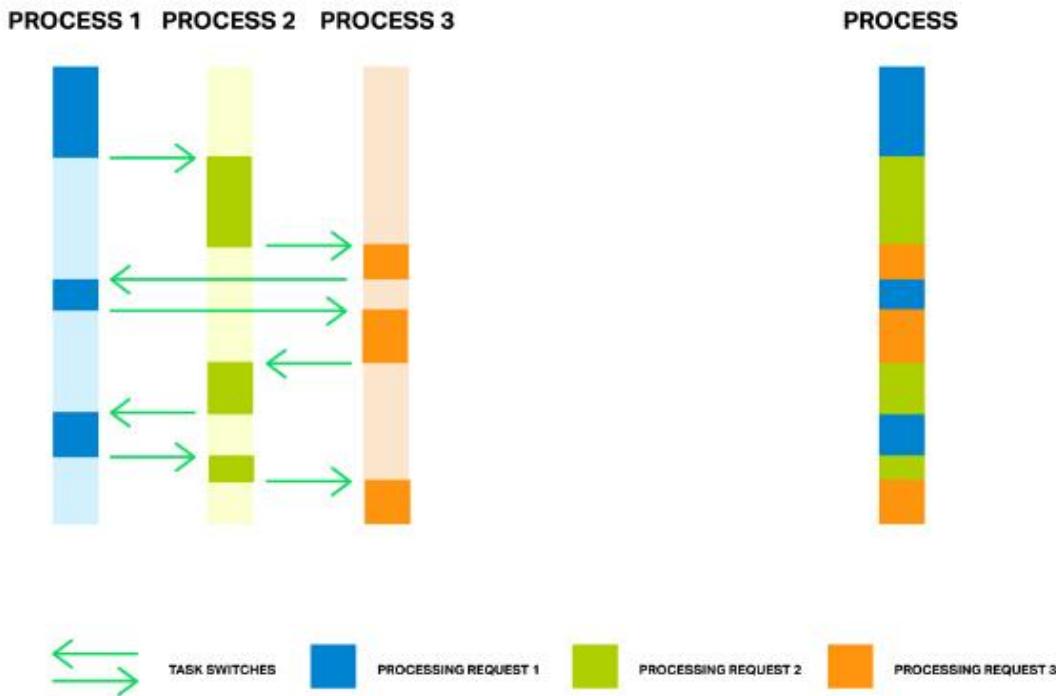
韩陆，北京航空航天大学软件工程硕士。热爱编程，热衷于开源社区的技术交流和分享。曾混迹于用友总部、新浪、Avaya 和 Technicolor 从事研发工作。现就职于阿里巴巴。

1. 引言

正如我们所知，NGINX 采用了[异步、事件驱动的方法来处理连接](#)。这种处理方式无需（像使用传统架构的服务器一样）为每个请求创建额外的专用进程或者线程，而是在一个工作进程中处理多个连接和请求。为此，NGINX 工作在非阻塞的 socket 模式下，并使用了[epoll](#) 和 [kqueue](#) 这样有效的方法。

因为满负载进程的数量很少（通常每核 CPU 只有一个）而且恒定，所以任务切换只消耗很少的内存，而且不会浪费 CPU 周期。通过 NGINX 本身实例，这种方法的优点已经为众人所知。NGINX 可以非常好地处理百万级规模的并发请求。

TRADITIONAL SERVER NGINX WORKER



每个进程都消耗额外的内存，而且每次进程间的切换都会消耗 CPU 周期并丢弃 CPU 高速缓存中的数据。

但是，异步、事件驱动方法仍然存在问题。或者，我喜欢将这一问题称为“敌兵”，这个敌兵的名字叫阻塞（blocking）。不幸的是，很多第三方模块使用了阻塞调用，然而用户（有时甚至是模块的开发者）并不知道阻塞的缺点。阻塞操作可以毁掉 NGINX 的性能，我们必须不惜一切代价避免使用阻塞。

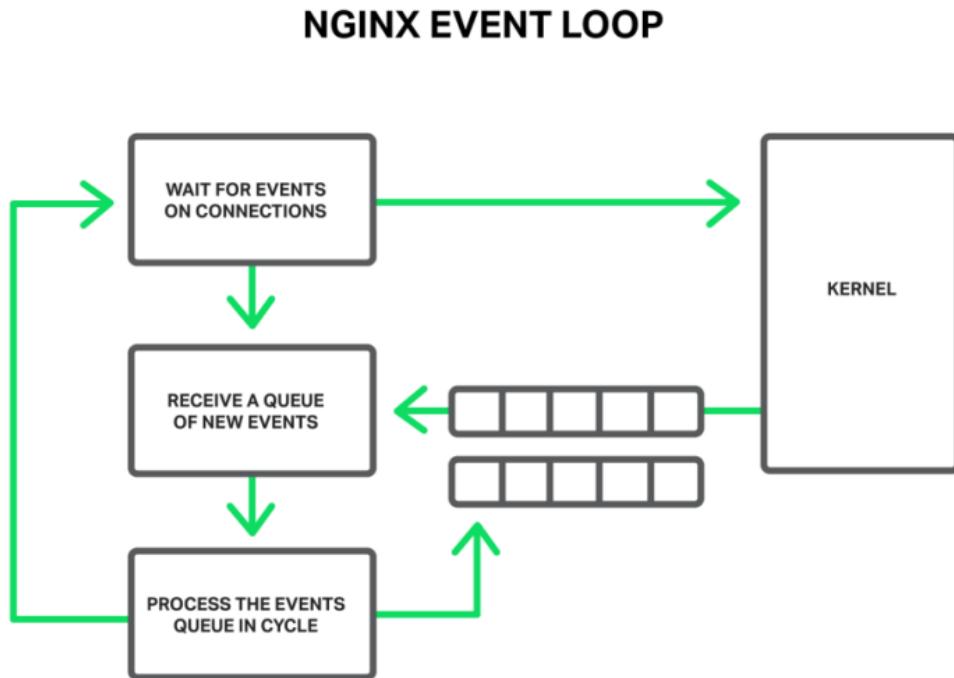
即使在当前官方的 NGINX 代码中，依然无法在全部场景中避免使用阻塞，[NGINX 1.7.11 中实现的线程池机制解决了这个问题](#)。我们将在后面讲述这个线程池是什么以及该如何使用。现在，让我们先和我们的“敌兵”进行一次面对面的碰撞。

2. 问题

首先，为了更好地理解这一问题，我们用几句话说明下 NGINX 是如何工作的。

通常情况下，NGINX 是一个事件处理器，即一个接收来自内核的所有连接事件的信息，然后向操作系统发出做什么指令的控制器。实际上，NGINX 干了编排操作系统的全部脏活累活，而操作系统做的是读取和发送字节这样的日常工作。所以，对于 NGINX 来说，快速和及时的响应

是非常重要的。



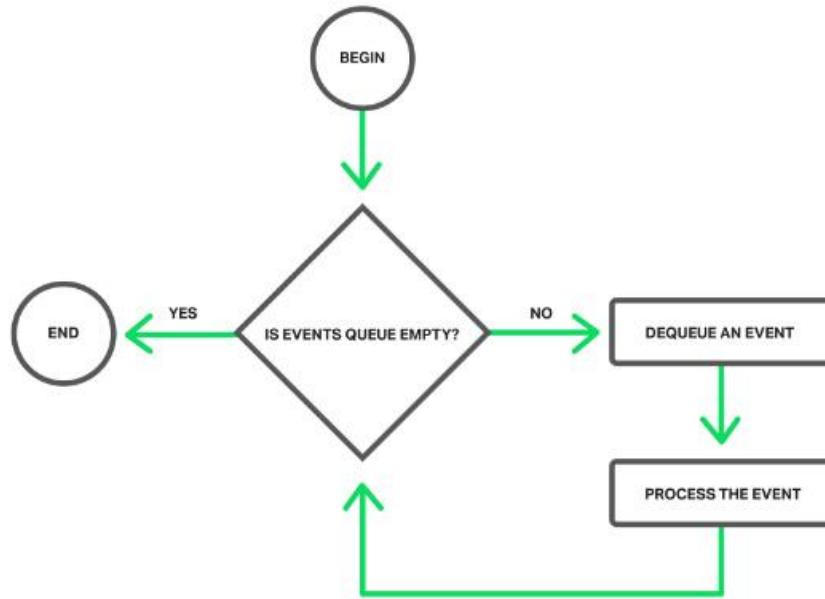
工作进程监听并处理来自内核的事件

事件可以是超时、socket 读写就绪的通知，或者发生错误的通知。NGINX 接收大量的事件，然后一个接一个地处理它们，并执行必要的操作。因此，所有的处理过程是通过一个线程中的队列，在一个简单循环中完成的。NGINX 从队列中取出一个事件并对其做出响应，比如读写 socket。在多数情况下，这种方式是非常快的（也许只需要几个 CPU 周期，将一些数据复制到内存中），NGINX 可以在一瞬间处理掉队列中的所有事件。

但是，如果 NGINX 要处理的操作是一些又长又重的操作，又会发生什么呢？整个事件处理循环将会卡住，等待这个操作执行完毕。

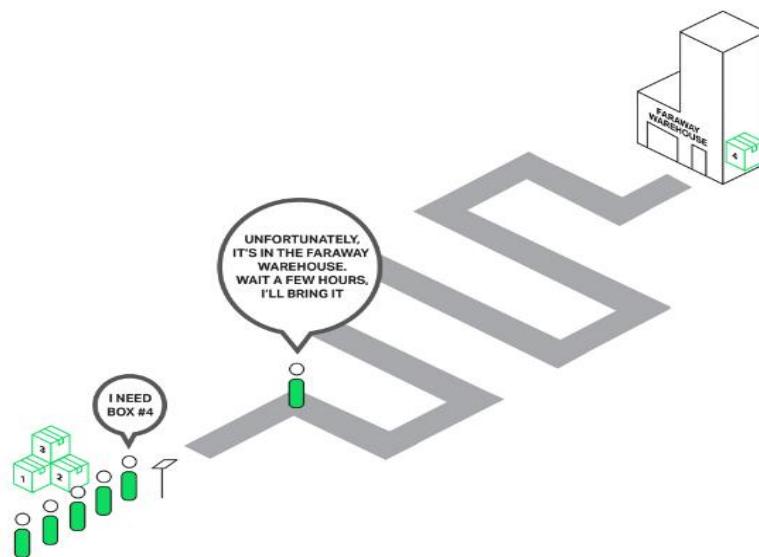
因此，所谓“阻塞操作”是指任何导致事件处理循环显著停止一段时间的操作。操作可以由于各种原因成为阻塞操作。例如，NGINX 可能因长时间、CPU 密集型处理，或者可能等待访问某个资源（比如硬盘，或者一个互斥体，亦或要从处于同步方式的数据库获得相应的库函数调用等）而繁忙。关键是在处理这样的操作期间，工作进程无法做其他事情或者处理其他事件，即使有更多的可用系统资源可以被队列中的一些事件所利用。

THE EVENTS QUEUE PROCESSING CYCLE



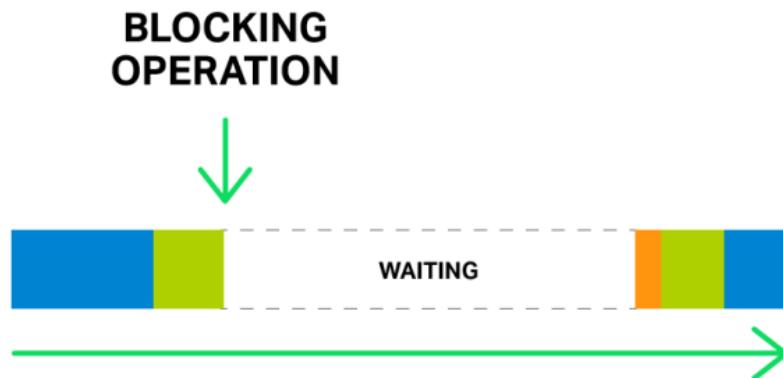
所有处理过程是在一个简单的循环中，由一个线程完成

我们来打个比方，一个商店的营业员要接待他面前排起的一长队顾客。队伍中的第一位顾客想要的某件商品不在店里而在仓库中。这位营业员跑去仓库把东西拿来。现在整个队伍必须为这样的配货方式等待数个小时，队伍中的每个人都很不爽。你可以想见人们的反应吧？队伍中每个人的等待时间都要增加这些时间，除非他们要买的东西就在店里。



队伍中的每个人不得不等待第一个人的购买

在 NGINX 中会发生几乎同样的情况，比如当读取一个文件的时候，如果该文件没有缓存在内存中，就要从磁盘上读取。从磁盘（特别是旋转式的磁盘）读取是很慢的，而当队列中等待的其他请求可能不需要访问磁盘时，它们也得被迫等待。导致的结果是，延迟增加并且系统资源没有得到充分利用。



一个阻塞操作足以显著地延缓所有接下来的操作

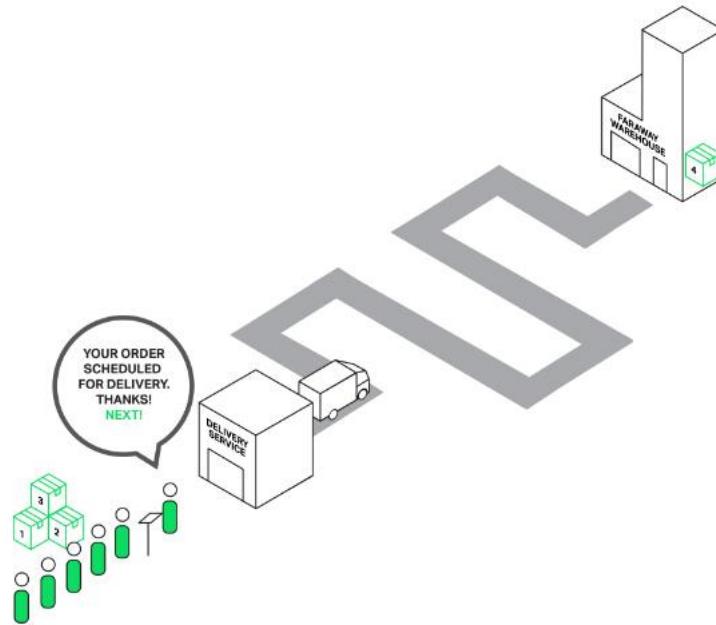
一些操作系统为读写文件提供了异步接口，NGINX 可以使用这样的接口（见 [AIO](#) 指令）。FreeBSD 就是个很好的例子。不幸的是，我们不能在 Linux 上得到相同的福利。虽然 Linux 为读取文件提供了一种异步接口，但是存在明显的缺点。其中之一是要求文件访问和缓冲要对齐，但 NGINX 很好地处理了这个问题。但是，另一个缺点更糟糕。异步接口要求文件描述符中要设置 O_DIRECT 标记，就是说任何对文件的访问都将绕过内存中的缓存，这增加了磁盘的负载。在很多场景中，这都绝对不是最佳选择。

为了有针对性地解决这一问题，在 NGINX 1.7.11 中引入了线程池。默认情况下，NGINX+还没有包含线程池，但是如果你想试试的话，可以[联系销售人员](#)，NGINX+ R6 是一个已经启用了线程池的构建版本。

现在，让我们走进线程池，看看它是什么以及如何工作的。

3. 线程池

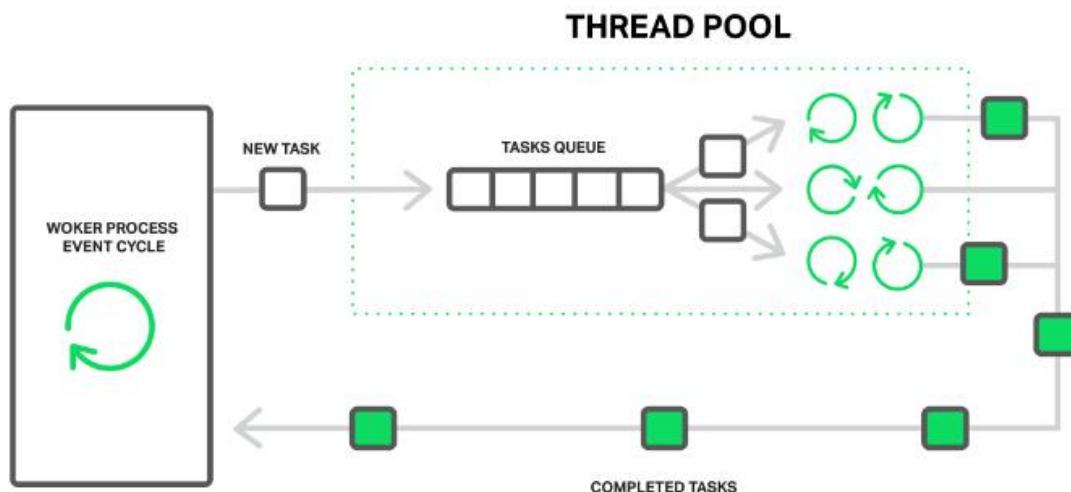
让我们回到那个可怜的，要从大老远的仓库去配货的售货员那儿。这回，他已经变聪明了（或者也许是在一群愤怒的顾客教训了一番之后，他才变得聪明的？），雇用了一个配货服务团队。现在，当任何人要买的东西在大老远的仓库时，他不再亲自去仓库了，只需要将订单丢给配货服务，他们将处理订单，同时，我们的售货员依然可以继续为其他顾客服务。因此，只有那些要买仓库里东西的顾客需要等待配货，其他顾客可以得到即时服务。



传递订单给配货服务不会阻塞队伍

对 NGINX 而言，线程池执行的就是配货服务的功能。它由一个任务队列和一组处理这个队列的线程组成。

当工作进程需要执行一个潜在的长操作时，工作进程不再自己执行这个操作，而是将任务放到线程池队列中，任何空闲的线程都可以从队列中获取并执行这个任务。



工作进程将阻塞操作卸给线程池

那么，这就像我们有了另外一个队列。是这样的，但是在这个场景中，队列受限于特殊的资源。

磁盘的读取速度不能比磁盘产生数据的速度快。不管怎么说，至少现在磁盘不再延误其他事件，只有访问文件的请求需要等待。

“从磁盘读取”这个操作通常是阻塞操作最常见的示例，但是实际上，NGINX 中实现的线程池可用于处理任何不适合在主循环中执行的任务。

目前，卸载到线程池中执行的两个基本操作是大多数操作系统中的 `read()` 系统调用和 Linux 中的 `sendfile()`。接下来，我们将对线程池进行测试（`test`）和基准测试（`benchmark`），在未来的版本中，如果有明显的优势，我们可能会卸载其他操作到线程池中。

4. 基准测试

现在让我们从理论过度到实践。我们将进行一次模拟基准测试（synthetic benchmark），模拟在阻塞操作和非阻塞操作的最差混合条件下，使用线程池的效果。

另外，我们需要一个内存肯定放不下的数据集。在一台 48GB 内存的机器上，我们已经产生了每文件大小为 4MB 的随机数据，总共 256GB，然后配置 NGINX，版本为 1.9.0。

配置很简单：

如上所示，为了达到更好的性能，我们调整了几个参数：禁用了 `logging` 和 `accept_mutex`，同时，启用了 `sendfile` 并设置了 `sendfile_max_chunk` 的大小。最后一个指令可以减少阻塞调用 `sendfile()` 所花费的最长时间，因为 NGINX 不会尝试一次将整个文件发送出去，而是每次发送大小为 512KB 的块数据。

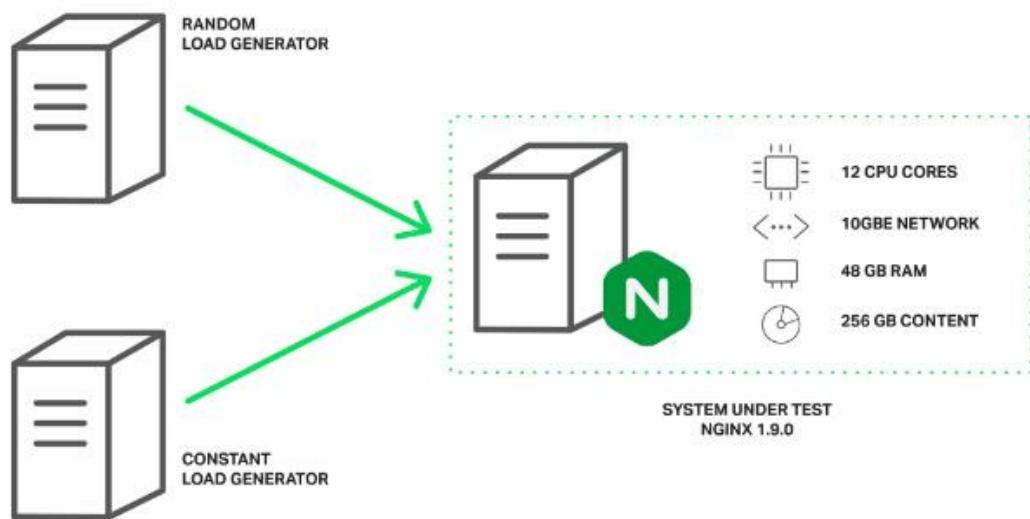
这台测试服务器有 2 个 Intel Xeon E5645 处理器（共计：12 核、24 超线程）和 10-Gbps 的网络接口。磁盘子系统是由 4 块西部数据 WD1003FBYX 磁盘组成的 RAID10 阵列。所有这些硬件由 Ubuntu 服务器 14.04.1 LTS 供电。

```
worker_processes 16;
events {
    accept_mutex off;
}

http {
    include mime.types;
    default_type application/octet-stream;
    access_log off;
    sendfile on;
```

```
sendfile_max_chunk 512k;

server {
    listen 8000;
    location /
    {
        root /storage;
    }
}
```



为基准测试配置负载生成器和 NGINX

客户端有 2 台服务器，它们的规格相同。在其中一台上，在 [wrk](#) 中使用 Lua 脚本创建了负载程序。脚本使用 200 个并行连接向服务器请求文件，每个请求都可能未命中缓存而从磁盘阻塞读取。我们将这种负载称作随机负载。

在另一台客户端机器上，我们将运行 [wrk](#) 的另一个副本，使用 50 个并行连接多次请求同一个文件。因为这个文件将被频繁地访问，所以它会一直驻留在内存中。在正常情况下，NGINX 能够非常快速地服务这些请求，但是如果工作进程被其他请求阻塞的话，性能将会下降。我们将这种负载称作恒定负载。

性能将由服务器上 [ifstat](#) 监测的吞吐率（throughput）和从第二台客户端获取的 [wrk](#) 结果来度量。

现在，没有使用线程池的第一次运行将不会带给我们非常振奋的结果：

```
% ifstat -bi eth2
eth2
Kbps in Kbps out
5531.24 1.03e+06
4855.23 812922.7
5994.66 1.07e+06
5476.27 981529.3
6353.62 1.12e+06
5166.17 892770.3
5522.81 978540.8
6208.10 985466.7
6370.79 1.12e+06
6123.33 1.07e+06
```

如上所示，使用这种配置，服务器产生的总流量约为 1Gbps。从下面所示的 **top** 输出，我们可以看到，工作进程的大部分时间花在阻塞 I/O 上（它们处于 top 的 D 状态）：

```
top - 10:40:47 up 11 days, 1:32, 1 user, load average: 49.61,
45.77 62.89
Tasks: 375 total, 2 running, 373 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0.0 us, 0.3 sy, 0.0 ni, 67.7 id, 31.9 wa, 0.0 hi, 0.0
si, 0.0 st
KiB Mem: 49453440 total, 49149308 used, 304132 free, 98780
buffers
KiB Swap: 10474236 total, 20124 used, 10454112 free, 46903412
cached Mem

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
4639 vbart 20 0 47180 28152 496 D 0.7 0.1 0:00.17 nginx
4632 vbart 20 0 47180 28196 536 D 0.3 0.1 0:00.11 nginx
4633 vbart 20 0 47180 28324 540 D 0.3 0.1 0:00.11 nginx
4635 vbart 20 0 47180 28136 480 D 0.3 0.1 0:00.12 nginx
4636 vbart 20 0 47180 28208 536 D 0.3 0.1 0:00.14 nginx
4637 vbart 20 0 47180 28208 536 D 0.3 0.1 0:00.10 nginx
4638 vbart 20 0 47180 28204 536 D 0.3 0.1 0:00.12 nginx
4640 vbart 20 0 47180 28324 540 D 0.3 0.1 0:00.13 nginx
4641 vbart 20 0 47180 28324 540 D 0.3 0.1 0:00.13 nginx
4642 vbart 20 0 47180 28208 536 D 0.3 0.1 0:00.11 nginx
4643 vbart 20 0 47180 28276 536 D 0.3 0.1 0:00.29 nginx
4644 vbart 20 0 47180 28204 536 D 0.3 0.1 0:00.11 nginx
4645 vbart 20 0 47180 28204 536 D 0.3 0.1 0:00.17 nginx
```

```
4646 vbart 20 0 47180 28204 536 D 0.3 0.1 0:00.12 nginx
4647 vbart 20 0 47180 28208 532 D 0.3 0.1 0:00.17 nginx
4631 vbart 20 0 47180 756 252 S 0.0 0.1 0:00.00 nginx
4634 vbart 20 0 47180 28208 536 D 0.0 0.1 0:00.11 nginx
4648 vbart 20 0 25232 1956 1160 R 0.0 0.0 0:00.08 top
25921 vbart 20 0 121956 2232 1056 S 0.0 0.0 0:01.97 sshd
25923 vbart 20 0 40304 4160 2208 S 0.0 0.0 0:00.53 zsh
```

在这种情况下，吞吐率受限于磁盘子系统，而 CPU 在大部分时间里是空闲的。从 **wrk** 获得的结果也非常低：

```
Running 1m test @ http://192.0.2.1:8000/1/1/1
12 threads and 50 connections
Thread Stats Avg Stdev Max +/- Stdev
Latency 7.42s 5.31s 24.41s 74.73%
Req/Sec 0.15 0.36 1.00 84.62%
488 requests in 1.01m, 2.01GB read
Requests/sec: 8.08
Transfer/sec: 34.07MB
```

请记住，文件是从内存送达的！第一个客户端的 200 个连接创建的随机负载，使服务器端的全部的工作进程忙于从磁盘读取文件，因此产生了过大的延迟，并且无法在合理的时间内处理我们的请求。

现在，我们的线程池要登场了。为此，我们只需在 `location` 块中添加 `aio threads` 指令：

```
location / {
    root /storage;
    aio threads;
}
```

接着，执行 NGINX reload 重新加载配置。然后，我们重复上述的测试：

```
% ifstat -bi eth2
eth2
Kbps in Kbps out
60915.19 9.51e+06
59978.89 9.51e+06
60122.38 9.51e+06
61179.06 9.51e+06
61798.40 9.51e+06
57072.97 9.50e+06
```

```
56072.61 9.51e+06
61279.63 9.51e+06
61243.54 9.51e+06
59632.50 9.50e+06
```

现在，我们的服务器产生的流量是 **9.5Gbps**，相比之下，没有使用线程池时只有约 1Gbps！

理论上还可以产生更多的流量，但是这已经达到了机器的最大网络吞吐能力，所以在这次 NGINX 的测试中，NGINX 受限于网络接口。工作进程的大部分时间只是休眠和等待新的事件（它们处于 top 的 S 状态）：

```
top - 10:43:17 up 11 days, 1:35, 1 user, load average: 172.71,
93.84, 77.90
Tasks: 376 total, 1 running, 375 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0.2 us, 1.2 sy, 0.0 ni, 34.8 id, 61.5 wa, 0.0 hi, 2.3
si, 0.0 st
KiB Mem: 49453440 total, 49096836 used, 356604 free, 97236
buffers
KiB Swap: 10474236 total, 22860 used, 10451376 free, 46836580
cached Mem

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
4654 vbart 20 0 309708 28844 596 S 9.0 0.1 0:08.65 nginx
4660 vbart 20 0 309748 28920 596 S 6.6 0.1 0:14.82 nginx
4658 vbart 20 0 309452 28424 520 S 4.3 0.1 0:01.40 nginx
4663 vbart 20 0 309452 28476 572 S 4.3 0.1 0:01.32 nginx
4667 vbart 20 0 309584 28712 588 S 3.7 0.1 0:05.19 nginx
4656 vbart 20 0 309452 28476 572 S 3.3 0.1 0:01.84 nginx
4664 vbart 20 0 309452 28428 524 S 3.3 0.1 0:01.29 nginx
4652 vbart 20 0 309452 28476 572 S 3.0 0.1 0:01.46 nginx
4662 vbart 20 0 309552 28700 596 S 2.7 0.1 0:05.92 nginx
4661 vbart 20 0 309464 28636 596 S 2.3 0.1 0:01.59 nginx
4653 vbart 20 0 309452 28476 572 S 1.7 0.1 0:01.70 nginx
4666 vbart 20 0 309452 28428 524 S 1.3 0.1 0:01.63 nginx
4657 vbart 20 0 309584 28696 592 S 1.0 0.1 0:00.64 nginx
4655 vbart 20 0 30958 28476 572 S 0.7 0.1 0:02.81 nginx
4659 vbart 20 0 309452 28468 564 S 0.3 0.1 0:01.20 nginx
4665 vbart 20 0 309452 28476 572 S 0.3 0.1 0:00.71 nginx
5180 vbart 20 0 25232 1952 1156 R 0.0 0.0 0:00.45 top
4651 vbart 20 0 20032 752 252 S 0.0 0.0 0:00.00 nginx
25921 vbart 20 0 121956 2176 1000 S 0.0 0.0 0:01.98 sshd
25923 vbart 20 0 40304 3840 2208 S 0.0 0.0 0:00.54 zsh
```

如上所示，基准测试中还有大量的 CPU 资源剩余。

wrk 的结果如下：

```
Running 1m test @ http://192.0.2.1:8000/1/1/1
 12 threads and 50 connections
 Thread Stats Avg  Stdev  Max +/- Stdev

  Latency 226.32ms 392.76ms 1.72s 93.48%
  Req/Sec 20.02 10.84 59.00 65.91%
  15045 requests in 1.00m, 58.86GB read
 Requests/sec: 250.57
 Transfer/sec: 0.98GB
```

服务器处理 4MB 文件的平均时间从 7.42 秒降到 226.32 毫秒（减少了 33 倍），每秒请求处理数提升了 31 倍（250 vs 8）！

对此，我们的解释是请求不再因为工作进程被阻塞在读文件，而滞留在事件队列中，等待处理，它们可以被空闲的进程处理掉。只要磁盘子系统能做到最好，就能服务好第一个客户端上的随机负载，NGINX 可以使用剩余的 CPU 资源和网络容量，从内存中读取，以服务于上述的第二个客户端的请求。

5. 依然没有银弹

在抛出我们对阻塞操作的担忧并给出一些令人振奋的结果后，可能大部分人已经打算在你的服务器上配置线程池了。先别着急。

实际上，最幸运的情况是，读取和发送文件操作不去处理缓慢的硬盘驱动器。如果我们有足够的内存来存储数据集，那么操作系统将会足够聪明地在被称作“页面缓存”的地方，缓存频繁使用的文件。

“页面缓存”的效果很好，可以让 NGINX 在几乎所有常见的用例中展示优异的性能。从页面缓存中读取比较快，没有人会说这种操作是“阻塞”。而另一方面，卸载任务到一个线程池是有一定开销的。

因此，如果内存有合理的大小并且待处理的数据集不是很大的话，那么无需使用线程池，NGINX 已经工作在最优化的方式下。

卸载读操作到线程池是一种适用于非常特殊任务的技术。只有当经常请求的内容的大小，不适

合操作系统的虚拟机缓存时，这种技术才是最有用的。至于可能适用的场景，比如，基于 NGINX 的高负载流媒体服务器。这正是我们已经模拟的基准测试的场景。

我们如果可以改进卸载读操作到线程池，将会非常有意义。我们只需要知道所需的文件数据是否在内存中，只有不在内存中时，读操作才应该卸载到一个单独的线程中。

再回到售货员那个比喻的场景中，这回，售货员不知道要买的商品是否在店里，他必须要么总是将所有的订单提交给配货服务，要么总是亲自处理它们。

人艰不拆，操作系统缺少这样的功能。第一次尝试是在 2010 年，人们试图将这一功能添加到 Linux 作为 [fincore\(\)](#) 系统调用，但是没有成功。后来还有一些尝试，是使用 RWF_NONBLOCK 标记作为 [preadv2\(\)](#) 系统调用来实现这一功能（详情见 LWN.net 上的[非阻塞缓冲文件读取操作](#)和[异步缓冲读操作](#)）。但所有这些补丁的命运目前还不明朗。悲催的是，这些补丁尚没有被内核接受的主要原因，貌似是因为旷日持久的撕逼大战（[bikeshedding](#)）。

另一方面，FreeBSD 的用户完全不必担心。FreeBSD 已经具备足够好的异步读取文件接口，我们应该用这个接口而不是线程池。

6. 配置线程池

所以，如果你确信在你的场景中使用线程池可以带来好处，那么现在是时候深入了解线程池的配置了。

线程池的配置非常简单、灵活。首先，获取 NGINX 1.7.11 或更高版本的源代码，使用--with-threads 配置参数编译。在最简单的场景中，配置看起来很朴实。我们只需要在 http、server，或者 location 上下文中包含 [aio threads](#) 指令即可。

这是线程池的最简配置。实际上的精简版本示例如下：

```
thread_pool default threads=32 max_queue=65536;  
aio threads=default;
```

这里定义了一个名为“default”，包含 32 个线程，任务队列最多支持 65536 个请求的线程池。如果任务队列过载，NGINX 将输出如下错误日志并拒绝请求：

```
thread pool "NAME" queue overflow: N tasks waiting
```

错误输出意味着线程处理作业的速度有可能低于任务入队的速度了。你可以尝试增加队列的最大值，但是如果这无济于事，那么这说明你的系统没有能力处理如此多的请求了。

正如你已经注意到的，你可以使用 [thread_pool](#) 指令，配置线程的数量、队列的最大值，以及线程池的名称。最后要说明的是，可以配置多个独立的线程池，将它们置于不同的配置文件中，用做不同的目的：

```
http {  
    thread_pool one threads=128 max_queue=0;  
    thread_pool two threads=32;  
  
    server {  
        location /one {  
            aio threads=one;  
        }  
  
        location /two {  
            aio threads=two;  
        }  
    }  
}
```

如果没有指定 `max_queue` 参数的值，默认使用的值是 65536。如上所示，可以设置 `max_queue` 为 0。在这种情况下，线程池将使用配置中全部数量的线程，尽可能地同时处理多个任务；队列中不会有等待的任务。

现在，假设我们有一台服务器，挂了 3 块硬盘，我们希望把该服务器用作“缓存代理”，缓存后端服务器的全部响应信息。预期的缓存数据量远大于可用的内存。它实际上是我们个人 CDN 的一个缓存节点。毫无疑问，在这种情况下，最重要的事情是发挥硬盘的最大性能。

我们的选择之一是配置一个 RAID 阵列。这种方法毁誉参半，现在，有了 NGINX，我们可以有其他的选择：

```
# 我们假设每块硬盘挂载在相应的目录中：/mnt/disk1、/mnt/disk2、  
/mnt/disk3  
  
proxy_cache_path /mnt/disk1 levels=1:2 keys_zone=cache_1:256m  
max_size=1024G  
    use_temp_path=off;  
proxy_cache_path /mnt/disk2 levels=1:2 keys_zone=cache_2:256m  
max_size=1024G  
    use_temp_path=off;  
proxy_cache_path /mnt/disk3 levels=1:2 keys_zone=cache_3:256m  
max_size=1024G  
    use_temp_path=off;
```

```
thread_pool pool_1 threads=16;
thread_pool pool_2 threads=16;
thread_pool pool_3 threads=16;

split_clients $request_uri $disk {
    33.3%   1;
    33.3%   2;
    *       3;
}location / {
    proxy_pass http://backend;
    proxy_cache_key $request_uri;
    proxy_cache cache_$disk;
    aio threads=pool_$disk;
    sendfile on;
}
```

在这份配置中，使用了 3 个独立的缓存，每个缓存专用一块硬盘，另外，3 个独立的线程池也各自专用一块硬盘。

缓存之间（其结果就是磁盘之间）的负载均衡使用 [split_clients](#) 模块，`split_clients` 非常适用于这个任务。

在 [proxy_cache_path](#) 指令中设置 `use_temp_path=off`，表示 NGINX 会将临时文件保存在缓存数据的同一目录中。这是为了避免在更新缓存时，磁盘之间互相复制响应数据。

这些调优将带给我们磁盘子系统的最大性能，因为 NGINX 通过单独的线程池并行且独立地与每块磁盘交互。每块磁盘由 16 个独立线程和读取和发送文件专用任务队列提供服务。

我敢打赌，你的客户喜欢这种量身定制的方法。请确保你的磁盘也持有同样的观点。

这个示例很好地证明了 NGINX 可以为硬件专门调优的灵活性。这就像你给 NGINX 下了一道命令，让机器和数据用最佳姿势来搞基。而且，通过 NGINX 在用户空间中细粒度的调优，我们可以确保软件、操作系统和硬件工作在最优模式下，尽可能有效地利用系统资源。

7. 总结

综上所述，线程池是一个伟大的功能，将 NGINX 推向了新的性能水平，除掉了一个众所周知的长期危害——阻塞——尤其是当我们真正面对大量内容的时候。

甚至，还有更多的惊喜。正如前面提到的，这个全新的接口，有可能没有任何性能损失地卸载

任何长期阻塞操作。NGINX 在拥有大量的新模块和新功能方面，开辟了一方新天地。许多流行的库仍然没有提供异步非阻塞接口，此前，这使得它们无法与 NGINX 兼容。我们可以花大量的时间和资源，去开发我们自己的无阻塞原型库，但这么做始终都是值得的吗？现在，有了线程池，我们可以相对容易地使用这些库，而不会影响这些模块的性能。

Facebook 如何向十亿人推荐东西



张天雷，清华大学计算机系博士生，主要研究方向为人工智能，曾参与 MSRA EntityCube 知识图谱和 Q20 群体智能游戏的研发，目前从事无人车研究，参与自然科学基金委视听觉认知重大项目，以及智能车未来挑战赛，多次担任裁判工作。

为了保证用户体验和使用效果，推荐系统中的机器学习算法一般都是针对完整的数据集进行的。然而，随着推荐系统输入数据量的飞速增长，传统的集中式机器学习算法越来越难以满足应用需求。因此，分布式机器学习算法被提出用来大规模数据集的分析。作为全球排名第一的社交网站，Facebook 就需要利用分布式推荐系统来帮助用户找到他们可能感兴趣的页面、组、事件或者游戏等。近日，[Facebook 就在其官网公布了其推荐系统的原理、性能及使用情况](#)。

目前，Facebook 中推荐系统所要面对的数据集包含了约 1000 亿个评分、超过 10 亿的用户以及数百万的物品。相比于著名的 [Netflix Prize](#)，Facebook 的数据规模已经超过了它两个数据级。如何在在大数据规模情况下仍然保持良好性能已经成为世界级的难题。为此，Facebook 设计了一个全新的推荐系统。幸运的是，Facebook 团队之前已经在使用一个分布式迭代和图像处理平台——[Apache Giraph](#)。因其能够很好的支持大规模数据，Giraph 就成为了 Facebook 推荐系统的基础平台。

在工作原理方面, Facebook 推荐系统采用的是流行的协同过滤 (Collaborative filtering, CF) 技术。CF 技术的基本思路就是根据相同人群所关注事物的评分来预测某个人对该事物的评分或喜爱程度。从数学角度而言, 该问题就是根据用户-物品的评分矩阵中已知的值来预测未知的值。其求解过程通常采用矩阵分解 ([Matrix Factorization](#), MF) 方法。MF 方法把用户评分矩阵表达为用户矩阵和物品的乘积, 用这些矩阵相乘的结果 R' 来拟合原来的评分矩阵 R , 使得二者尽量接近。如果把 R 和 R' 之间的距离作为优化目标, 那么矩阵分解就变成了求最小值问题。

对大规模数据而言, 求解过程将会十分耗时。为了降低时间和空间复杂度, 一些从随机特征向量开始的迭代式算法被提出。这些迭代式算法渐渐收敛, 可以在合理的时间内找到一个最优解。随机梯度下降 ([Stochastic Gradient Descent](#), SGD) 算法就是其中之一, 其已经成功的用于多个问题的求解。SGD 基本思路是以随机方式遍历训练集中的数据, 并给出每个已知评分的预测评分值。用户和物品特征向量的调整就沿着评分误差越来越小的方向迭代进行, 直到误差到达设计要求。因此, SGD 方法可以不需要遍历所有的样本即可完成特征向量的求解。交替最小二乘法 ([Alternating Least Square](#), ALS) 是另外一个迭代算法。其基本思路为交替固定用户特征向量和物品特征向量的值, 不断的寻找局部最优解直到满足求解条件。

为了利用上述算法解决 Facebook 推荐系统的问题, 原本 Giraph 中的标准方法就需要进行改变。之前, Giraph 的标准方法是把用户和物品都当作为图中的顶点、已知的评分当作边。那么, SGD 或 ALS 的迭代过程就是遍历图中所有的边, 发送用户和物品的特征向量并进行局部更新。该方法存在若干重大问题。首先, 迭代过程会带来巨大的网络通信负载。由于迭代过程需要遍历所有的边, 一次迭代所发送的数据量就为边与特征向量个数的乘积。假设评分数为 1000 亿、特征向量为 100 对, 每次迭代的通信数据量就为 80TB。其次, 物品流行程度的不同会导致图中节点度的分布不均匀。该问题可能会导致内存不够或者引起处理瓶颈。假设一个物品有 1000 亿个评分、特征向量同样为 100 对, 该物品对应的一个点在一次迭代中就需要接收 80GB 的数据。最后, Giraph 中并没有完全按照公式中的要求实现 SGD 算法。真正实现中, 每个点都是利用迭代开始时实际收到的特征向量进行工作, 而并非全局最新的特征向量。

综合以上可以看出, Giraph 中最大的问题就在于每次迭代中都需要把更新信息发送到每一个顶点。为了解决这个问题, Facebook 发明了一种利用 work-to-work 信息传递的高效、便捷方法。该方法把原有的图划分为由若干 work 构成的一个圆。每个 worker 都包含了一个物品集合和若干用户。在每一步, 相邻的 worker 沿顺时针方法把包含物品更新的信息发送到下游的 worker。这样,

每一步都只处理了各个 worker 内部的评分，而经过与 worker 个数相同的步骤后，所有的评分也全部都被处理。该方法实现了通信量与评分数无关，可以明显减少图中数据的通信量。而且，标准方法中节点度分布不均匀的问题也因为物品不再用顶点来表示而不复存在。为了进一步提高算法性能，Facebook 把 SGD 和 ALS 两个算法进行了揉合，提出了旋转混合式求解方法。

接下来，Facebook 在运行实际的 A/B 测试之间对推荐系统的性能进行了测量。首先，通过输入一直的训练集，推荐系统对算法的参数进行微调来提高预测精度。然后，系统针对测试集给出评分并与已知的结果进行比较。Facebook 团队从物品平均评分、前 1/10/100 物品的评分精度、所有测试物品的平均精度等来评估推荐系统。此外，均方根误差（Root Mean Squared Error, RMSE）也被用来记录单个误差所带来的影响。

此外，即使是采用了分布式计算方法，Facebook 仍然不可能检查每一个用户/物品对的评分。团队需要寻找更快的方法来获得每个用户排名前 K 的推荐物品，然后再利用推荐系统计算用户对其的评分。其中一种可能的解决方案是采用 [ball tree](#) 数据结构来存储物品向量。all tree 结构可以实现搜索过程 10-100 倍的加速，使得物品推荐工作能够在

合理时间内完成。另外一个能够近似解决问题的方法是根据物品特征向量对物品进行分类。这样，寻找推荐评分就划分为寻找最推荐的物品群和在物品群中再提取评分最高的物品两个过程。该方法在一定程度上会降低推荐系统的可信度，却能够加速计算过程。

最后，Facebook 给出了一些实验的结果。在 2014 年 7 月，[Databricks 公布了在 Spark 上实现 ALS 的性能结果](#)。Facebook 针对 [Amazon 的数据集](#)，基于 [Spark MLlib](#) 进行标准实验，与自己的旋转混合式方法的结果进行了比较。实验结果表明，Facebook 的系统比标准系统要快 10 倍左右。而且，前者可以轻松处理超过 1000 亿个评分。

目前，该方法已经用了 Facebook 的多个应用中，包括页面或者组的推荐等。为了能够减小系统负担，Facebook 只是把度超过 100 的页面和组考虑为候选对象。而且，在初始迭代中，Facebook 推荐系统把用户喜欢的页面/加入的组以及用户不喜欢或者拒绝加入的组都作为输入。此外，Facebook 还利用基于 ALS 的算法，从用户获得间接的反馈。未来，Facebook 会继续对推荐系统进行改进，包括利用社交图和用户连接改善推荐集合、自动化参数调整以及尝试比较好的划分机器等。

大数据平台架构实践



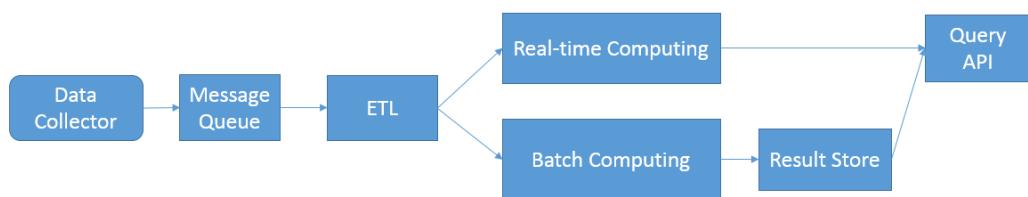
阁志涛，现任 TalkingData 研发副总裁，领导研发了公司的数据管理平台（DMP）、数据观象台等产品，并且负责公司大数据计算平台的研发。目前专注于构建一个融合多种计算模型，支持机器学习和数据挖掘的大数据计算平台。关注 Spark、Hadoop、HBase、MongoDB 等技术。超过 15 年的 IT 领域从业经验，一直从事大规模分布式计算系统、中间件、BI 等相关工作。

前言

随着移动互联网时代的到来，越来越多的与人、与物、与环境有关的数据产生，大数据技术也变得越来越重要。在国内，大数据也由几年前的概念阶段逐渐的在不同的企业和行业落地，并且对企业的运营、发展起到了越来越重要的作用。从 2011 年创业之初，TalkingData 就坚信一句话：“In God we trust, everyone else must bring data”。期望通过数据去改变人们做决策的方式，通过数据让人们更好的了解自己。创业这 4 年，对于 TalkingData 技术团队来讲，也是对大数据技术架构的认识逐步深入的过程。经过 4 年的发展，我们每天处理的新增数据由几个 GB 逐渐的增加到如今的数个 TB。而数据计算类型也从最初的统计分析的类型到支持多维交叉、即席查询、机器学习、广告归因等等多种计算类型。一路跌跌撞撞走来，TalkingData 技术团队在大数据技术上踩过不少坑，有过许多个不眠之夜，也逐渐总结了一些自己的经验，并在这里分享给大家。

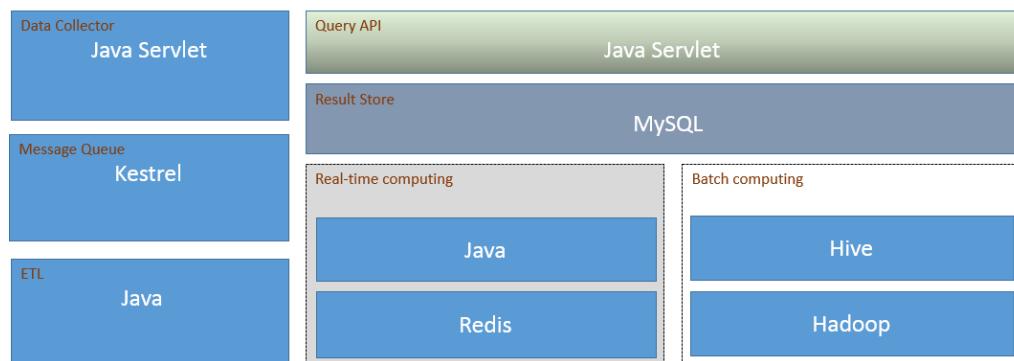
大数据平台之初试

提到大数据平台，首先需要考虑的是采用什么样的技术架构。TalkingData 在成立之初，主要业务是移动统计分析业务，主要帮助我们的客户分析移动应用的新增、活跃、留存、流失等等指标。包括这些指标在渠道、版本等不同维度的分布。在 2011 年，大部分的分析系统还都是纯粹离线的分析系统，所有的指标都是 T+1 才能获取。但是考虑到用户的体验，所有的数据都是 T+1 显然是不能够满足用户的需求的，用户需要知道当天的实时数据的分析结果。为了解决这个问题，大数据平台的高层架构如下：



数据通过数据收集器进行接收，接收后的数据会发送到消息队列中。ETL 负责将数据进行规范化和逻辑组织、抽取，然后发送给实时计算部分和离线批量计算部分。实时计算部分主要负责计算当天的实时数据，而离线批量处理部分则主要计算当天以前的数据。查询 API 则提供统计分析业务对计算结果的查询。

具体到采用的技术，平台的组件架构图如下：



在这个架构当中，不同的组件采用的技术如下所述。

□ Data Collector (数据收集器)

Data Collector 主要负责收集从 SDK 发送来的各种数据，以日志的形式保留在本地，然后再将数据发送到消息队列中。整个 Data Collector 组件包括 Nginx 作为负载均衡器，接收所有从 SDK 发送来的请求数据并发送到后方的真正处理数据的 Data Collector 中。Data Collector 是运行在 Jetty 容器中的 java servlet，利用容器提供的多线程的支持，接收并处理 SDK 发送上来的数据。数据会先以日志的形式存储在本地磁盘，然后再将数据发送到 Message Queue 中。Data Collector 由于承担了数据接收的工作，在设计实现中不承担任何的计算逻辑，主要承担的是存储和转发的逻辑，从而能够高效的接收数据。

□ Message Queue (消息队列)

消息队列主要是为了解耦数据接收和数据计算的逻辑，在第一个版本中，采用的是轻量级的消息队列 Kestrel。作为一个轻量级的消息队列，Kestrel 非常的轻巧和方便使用，并且能够支持消息的存储，对消息的访问支持 memcached 协议，并且有非常不错的读写性能。对于快速构建一个支持异步处理的分布式系统来讲，Kestrel 无疑是一个非常简单方便的选择。

□ Batch Computing (离线处理)

Batch Computing 主要是对非实时要求的数据做批量处理，在 2011 年，想做离线批量计算，能够选择必然是 hadoop 生态系统中的某种技术。可以选择自己写 MapReduce，也可以选择 Pig 或者 Hive 来完成对应的工作。考虑到开发的方便性，Hive 因为其支持类似于标准 SQL 的 HQL 最终被我们选择为离线处理的计算平台。批量计算的结果，会存储到 Result Store 中。为了解决多维交叉的问题，在批量处理过程中，我们会对每个维度生成对应设备的 bitmap 索引，同时也会将索引存储到 Result Store 中。

□ Real-time Computing (实时计算)

Real-time Computing 主要是为了解决客户需要看到实时分析数据的结果需求而引入的组件。在离线处理部分，我们通过 Hive 来计算一天以前的数据，包括各种时间跨度比较长的指标。不过对于用户当天当时的各种指标，Hadoop 生态系统中的各种技术，因为其设计就是为离线计算而生，就不能够满足实时计算的要求了。在 2011 年，还没有非常好的开源的实时处理框架。我们能够选择的只能是自己去根据业务的需求开发自己的实时计算的组件。整个实时计算组件是采用 Redis 内存数据库为基础实现的。利用 Redis 提供的高速的访问能力，以及对能够对 key 的值进行增加计数，可以设置 key 过期等能力，我们将实时的计算指标通过组织 Redis key 来完成。

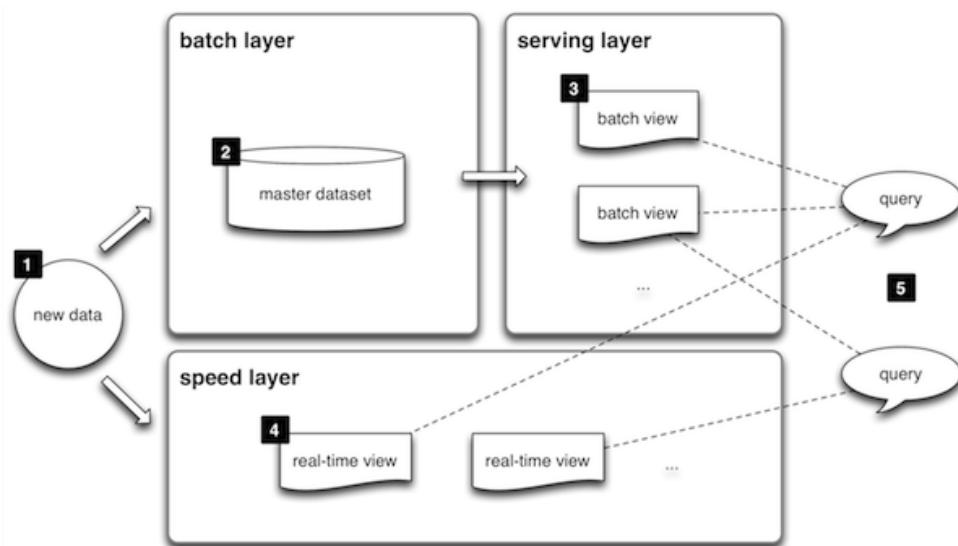
□ Query API (查询接口)

数据在离线计算和实时计算完成之后，会被 Query API 进行查询。Query API 会分别从 Redis 当中查询当日的实时结果，同时会从 Result DB 查询离线计算的结果。为了方便查询，我们将 Query

API 进行了封装，封装成了一个支持类似于 SQL 的查询引擎。分析业务会通过类 SQL 的表达将查询发送给查询 API，查询 API 会根据时间切片的不同，决定是从 Redis 还是从 Result DB 中查询数据，并且将结果拼装后返回给分析业务系统应用。

现在回头看我们的架构，实际上像极了后来 Storm 的开发者 Nathan Marz 提出的 Lambda 架构，其架构如下图所示。

从图中可以看到，我们在 2011 年采用的架构和 Lambda 架构非常的像，只是采用的技术实现不尽相同。



大数据平台之改良

随着业务的发展和数据的增加，2011 年我们那套技术架构也逐渐出现各种各样的问题，对架构的重构也就变得越来越重要。这套架构主要存在的问题主要包括：

□ 数据一致性问题

由于数据计算存在实时和离线两个部分，实时计算采用的 Redis，而离线部分采用的是 Hive，由于 SDK 上传的数据可能出现延迟，这样实时计算的时候可能当天没有上传的数据，在以后会上传，这样实时计算的当天的数据和当天过后通过 Hive 进行批量计算的结果就会产生偏差，有时会造成用户的困惑。

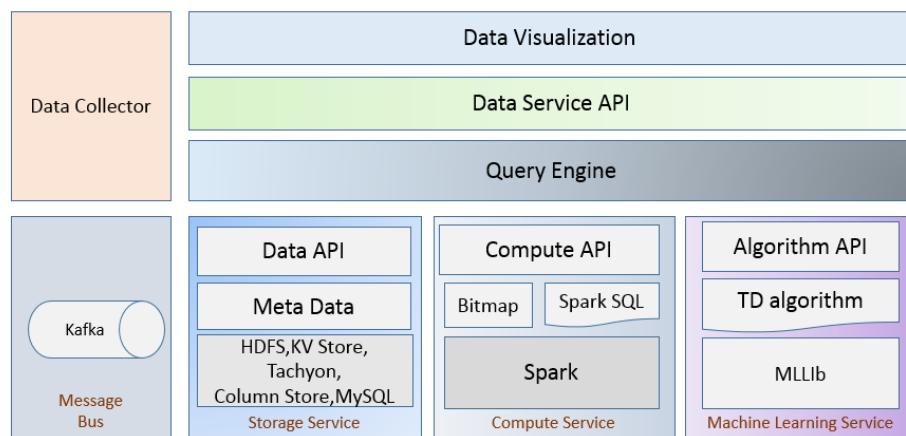
□ 数据处理能力不足问题

随着数据量的逐渐增加，这套技术架构也开始面临数据处理能力的考验。Kestrel 作为一个轻量级的队列，能够满足初期的要求。可是数据量增多后，kestrel 的平行扩展能力的不足开始体现，

另外随着数据业务的增加，消息队列模型需要更灵活的支持多消费者的的消息队列。而 `kestrel` 在这方面也很难满足业务的需求。

为了解决这些问题，我们决定对架构进行重构，于是一套自行研发的计算框架 Torch（火炬）系统应运而生。这套系统采用微批次的概念，主要解决大数据场景下统计分析业务的需求。整体的技术架构如下图所示。

这个架构中，大家可以看到，消息队列从 Kestrel 变为 Kafka。采用 Kafka，在数据量每天都在增加的时候，更方便的进行平行扩展。另外，业务可靠性的要求也越来越高，而 Kafka 本身的高度可靠性的特点也更适合业务的需求。



在这个改良的架构中，不再存在实时和离线处理两个数据计算路径，所有的数据计算都是通过 Torch 的计算引擎来完成。整个计算引擎分为 Counter 引擎和 Bitmap 引擎两个部分。计算是以分钟为单位的微批次的计算，Counter 引擎主要进行汇总类型的计算，而 Bitmap 引擎则负责生成数据的 Bitmap 索引，并将结果存储在存储当中。计算过程是基于预先定义好维度和度量的事实表来进行的。而某些不能预先进行索引的数据，则存储在列式数据库当中，从而可以在没有预先计算的情况下，高效的执行分析型的计算。

大数据平台之进阶

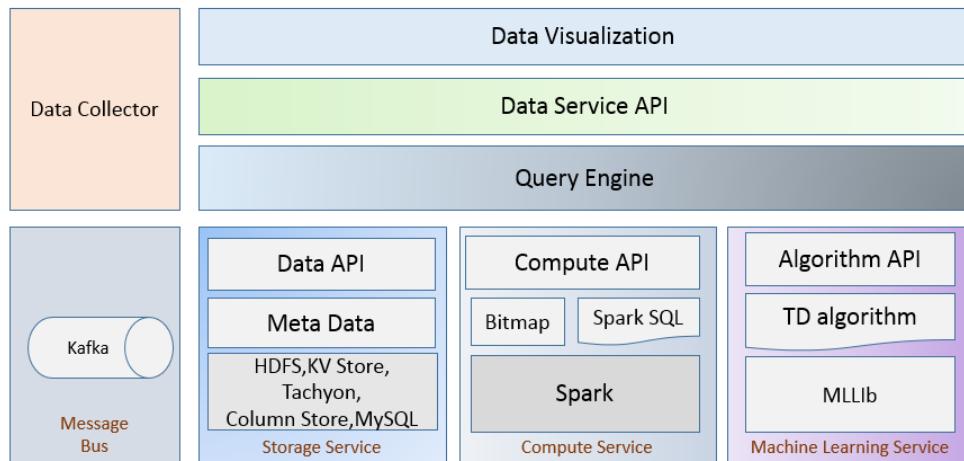
随着公司业务的进一步发展和扩充，对于数据计算的需求已经不仅仅是统计分析类型的业务，对数据价值的探索变得越来越重要。面向统计分析业务的平台已经不能够满足业务发展的需求，我们需要对平台进行进一步重构，使得大数据平台能够满足：

- 统计分析业务
 - 交互式分析

□ 机器学习

□ 数据可视化

基于这些需求，TalkingData 新的 π 系统应运而生。整个平台的架构如下图所示。



在新的 π 系统架构中，整个大数据平台除了能够支持统计分析业务，还增加了对机器学习、以及交互式分析的支持。不同的技术组件描述如下。

□ Data Collector

在新的架构中，为了提高数据收集的效率，Data Collector 在新的架构中从基于 java servlet 改为基于 actor 模型的 node.js 进行实现。另外 Data Collector 本身分为前置节点和中心节点两级，从而可以实现数据收集的分布式部署。前置节点分布式部署在多个区域，使得 SDK 可以选择网络连接更快的节点发送数据。而前置节点和中心节点采用高压缩比的数据传输，从而更好的利用中心机房的带宽资源。

□ Message Bus

在新的架构中，消息队列还是采用基于 Kafka 的消息总线，从而保证平行扩展、高可靠性，另外支持多消费者。

□ Storage Service

在新的架构中，我们将使用到的存储做为服务进行了封装。整个存储部分根据数据的冷热时间不同，进行分区。热数据存储在分布式缓存 Tachyon 中，而冷数据则以 Parquet 格式存储在 HDFS 当中。为了更好的支持多维交叉的分析型业务，TalkingData 开发了针对 bitmap 的 bitmap 存储。所有的存储可以通过封装好的 API 进行统一的访问。另外，引入了基于 HCatalog 进行封装的元数据管理，从而方便对数据的管理和访问。

□ Compute Service

计算服务基于分布式计算框架 Spark，其中融合了 Torch 系统中的 bitmap 引擎，从而可以对流式数据生成 bitmap 索引，并将索引存储在 bitmap 存储中。另外将流式消费的数据转化为列式存储结构，存储在 Tachyon 中。Tachyon 中存储的数据有有效期，过期的数据会迁移到 HDFS 当中，并且在 Tachyon 中做清除。即时数据请求会根据请求类型和时间，决定是从 bitmap 存储、Tachyon、还是 HDFS 中读取数据。所有的数据计算封装为统一的数据计算 API。

□ Machine Learning Service

为了更好的发挥数据的价值，我们的架构中引入了机器学习服务。机器学习服务包括了 Spark 提供的 MLLib，另外也包括公司自己开发的一些高效的机器学习算法，比如随机决策森林、LR 等等算法。所有的算法都封装为算法库，通过 API 的方式提供调用。

□ Query Engine

查询引擎则是对存储 API、计算 API 和机器学习 API 进行封装，上层业务可以通过类似于 SQL 的语句进行数据计算，查询引擎会对查询进行解析，然后转化为对应的下层 API 调用和执行。

□ Data Service API

数据服务 API 则是各数据业务系统对数据进行业务化封装的 API，这些 API 一般都是 Restful API。数据可视化层可以通过这些 Restful API 获取数据，进行数据展现。

□ Data Visualization

数据可视化服务包含标准的数据可视化组件，通过对数据可视化组件化封装，业务系统的开发变得更为高效。数据可视化组件通过与数据服务 API 交互，获取需要的数据，完成数据的可视化展现。

后记

新的 π 系统的架构是 TalkingData 技术团队第一次以更为面向全局的视角进行的一次架构重构。整个架构的设计和实现也融合了公司不同技术团队的集体力量，整个架构目前还在逐步完善中，期望我们能够将这个架构变得更加成熟，实现的更加灵活，变成一个真正的可平行扩展的支持多种大数据计算能力的大数据平台。

京东 618：Docker 扛大旗，弹性伸缩成重点



刘海峰，京东云平台首席架构师、系统技术部负责人。系统技术部专注于基础服务的自主研发与持续建设，包括分布式存储、以内存为中心的 NoSQL 服务、图片源站、内容分发网络、消息队列、内部 SOA 化、弹性计算云等核心系统，均大规模部署以支撑京东集团的众多业务。



郭蕾，InfoQ 技术编辑，文艺范儿程序员。在 CRM 行业厮混 3 年多，喜欢技术写作和社区运营，我狂妄，我自负，我信奉见城彻先生的那句话：偏执、冒险、狂妄的人终是英雄。我不是英雄，但我会努力成为英雄。

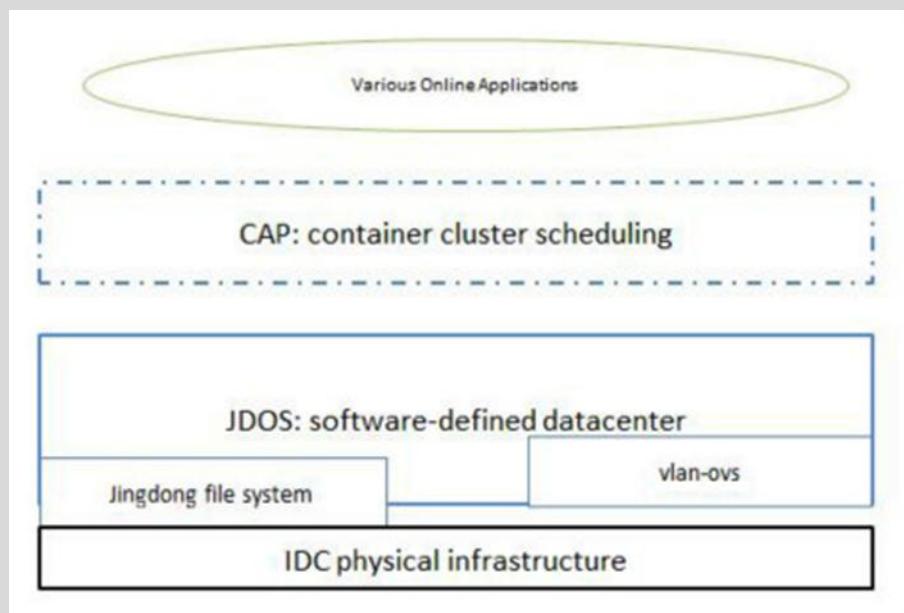
不知不觉中，年中的 618 和年终的 11.11 已经成为中国电商的两大促销日，当然，这两天也是一年中系统访问压力最大的两天。对于京东而言，618 更是这一年中最大的一次考试，考点是系统的扩展性、稳定性、容灾能力、运维能力、紧急故障处理能力。弹性计算云是京东 2015 年研发部战略项目，它基于 Docker 简化了应用的部署和扩容，提高了系统的伸缩能力。目前京东的图片系统、单品页、频道页、风控系统、缓存、登录、团购、O2O、无线、拍拍等业务都已经运行在弹性计算云系统中。

过去的一段时间里，弹性计算云项目在京东内部获得了广泛应用，并且日趋稳定成熟。一方面，这个项目可以更有效地管理机器资源，提高资源利用率；另外还能大幅提高生产效

率，让原来的申请机器上线扩容逐渐过渡到全自动维护。京东弹性计算云项目将深刻影响京东未来几年的基础架构。

InfoQ：能否介绍下京东弹性计算云项目的情况，你们什么时候开始使用 Docker 的？目前有多大的规模？

刘海锋：弹性计算云项目在去年第四季度开始研发，今年春节后正式启动推广应用。经过半年多的发展，逐渐做到了一定规模。截至 6 月 17 日，我们线上运行了 9853 个 Docker 实例（注：无任何夸大）以及几百个 KVM 虚拟机。京东主要的一些核心应用比如商品详情页、图片展现、秒杀、配送员订单详情等等都部署在弹性云中。弹性计算云项目也作为今年 618 的扩容与灾备资源池，这估计是国内甚至世界上最大规模的 Docker 应用之一。随着业务的发展以及 IDC 的增加，预计今年年底规模会翻两番，京东大部分应用程序都会通过容器技术来发布和管理。



系统架构可以这样简洁定义：弹性计算云 = 软件定义数据中心 + 容器集群调度。整个项目分成两层架构，底层为基础平台，系统名 JDOS，通过『OpenStack married with Docker』来实现基础设施资源的软件管理，Docker 取代 VM 成为一等公民，但这个系统目标是统一生产物理机、虚拟机与轻量容器；上层为应用平台，系统名 CAP，集成部署监控日志等工具链，实现『无需申请服务器，直接上线』，并进行业务特定的、数据驱动的容器集群调度与弹性伸缩。

InfoQ: 能否谈谈你们的 Docker 使用场景？在 618 这样的大促中，Docker 这样的容器有什么优势？618 中有哪些业务跑到 Docker 中？

刘海锋：目前主要有两类场景：无状态的应用程序，和缓存实例。这两类场景规模最大也最有收益。不同的场景具体需求不同，因此技术方案也不相同。内部我们称呼为“胖容器”与“瘦容器”技术。从资源抽象角度，前者带独立 IP 以及基础工具链如同一台主机，后者可以理解为物理机上面直接启动 cgroup 做资源控制加上镜像机制。

618 这样的大促备战，弹性计算云具备很多优势：非常便捷的上线部署、半自动或全自动的扩容。Docker 这样的操作系统级虚拟化技术，启动速度快，资源消耗低，非常适合私有云建设。

今年 618，是京东弹性计算云第一次大促亮相，支持了很多业务的流量。比如图片展现 80% 流量、单品页 50% 流量、秒杀风控 85% 流量、虚拟风控 50% 流量，还有三级列表页、频道页、团购页、手机订单详情、配送员主页等等，还有全球购、O2O 等新业务。特别是，今年 618 作战指挥室大屏监控系统都是部署在弹性云上的。

InfoQ: 你们是如何结合 Docker 和 OpenStack 的？网络问题是如何解决的？

刘海锋：我们深度定制 OpenStack，持续维护自己的分支，称之为 JDOS (the Jingdong Datacenter Operating System)。JDOS 目标很明确：统一管理和分配 Docker、VM、Bare Metal，保证稳定高性能。网络方面不玩复杂的，线上生产环境划分 VLANs + Open vSwitch。SDN 目前没有显著需求所以暂不投入使用。我们以『研以致用』为原则来指导技术选择和研发投入。

InfoQ: 能否谈谈你们目前基于 Docker 的 workflow？

刘海锋：弹性计算平台集成了京东研发的统一工作平台（编译测试打包上线审批等）、自动部署、统一监控、统一日志、负载均衡、数据库授权，实现了应用一键部署，并且全流程处理应用接入，扩容、缩容、下线等操作。支持半自动与全自动。

InfoQ: 这么多的容器，你们是如何调度的？

刘海锋：容器的调度由自主研发的 CAP (Cloud Application Platform) 来控制，并会根据应用配置的策略来进行调度；在创建容器的时候，会根据规格、镜像、机房和交换机等

策略来进行创建；创建完容器后，又会根据数据库策略、负载策略、监控策略等来进行注册；在弹性调度中，除了根据容器的资源情况，如 CPU 和连接数，还会接合应用的 TPS 性能等等来综合考虑，进行弹性伸缩。

目前已经针对两大类在线应用实现自动弹性调度，一是 Web 类应用，二是接入内部 SOA 框架的服务程序。大规模容器的自动化智能调度，我们仍在进一步做研究与开发。

InfoQ：目前主要有哪些业务使用了 Docker？业务的选择方面有什么建议？

刘海锋：目前有 1000 个应用已经接入弹性云，涵盖京东各个业务线，包括很多核心应用。目前我们主要支持计算类业务，存储类应用主要应用到了缓存。数据库云服务也将通过 Docker 进行部署和管理。

特别强调的是，业务场景不同，技术方案就有差别。另外，有些对隔离和安全比较敏感的业务就分配 VM。技术无所谓优劣和新旧，技术以解决问题和创造业务价值为目的。

InfoQ：你们的缓存组件也跑在 Docker 中，这样做有什么好处？IO 什么的没有问题吗？有什么好的经验可以分享？

刘海锋：我们团队负责一个系统叫 JIMDB，京东统一的缓存与高速 NoSQL 服务，兼容 Redis 协议，后台保证高可用与横向扩展。系统规模增长到现在的三千多台大内存机器，日常的部署操作、版本管理成为最大痛点。通过引入 Docker，一键完成容器环境的缓存集群的全自动化搭建，大幅提升了系统运维效率。

技术上，缓存容器化的平台并不基于 OpenStack，而是基于 JIMDB 自身逻辑来开发。具体说来，系统会根据需求所描述的容量、副本数、机房、机架、权限等约束创建缓存容器集群，并同时在配置中心注册集群相关元数据描述信息，通过邮件形式向运维人员发出构建流水详单，并通知用户集群环境构建完成。调度方面，不仅会考虑容器内进程，容器所在机器以及容器本身当前的实时状况，还会对它们的历史状况进行考察。一旦缓存实例触发内存过大流量过高等扩容条件，系统会立即执行扩容任务创建新的容器分摊容量和流量，为保证服务质量，缓存实例只有在过去一段时间指标要求持续保持低位的情况下才会缩容。在弹性伸缩的过程中，会采用 Linux TC 相关技术保证缓存数据迁移速度。

InfoQ：使用过程中有哪些坑？你们有做哪些重点改进？

刘海锋：坑太多了，包括软件、硬件、操作系统内核、业务使用方式等等。底层关键改进印象中有两个方面：第一，Docker 本地存储结构，抛弃 Device Mapper、AUTFS 等选项，自行定制；第二，优化 Open vSwitch 性能。比如，优化 Docker 镜像结构，加入多层合并、压缩、分层 tag 等技术，并采用镜像预分发技术，可以做到秒级创建容器实例；优化 Open vSwitch 转发层，显著提升网络小包延迟。



ArchSummit
全 球 架 构 师 峰 会

应用性能管理(APM) 技术实践专场

2015年7月17日（周五）
深圳·大梅沙京基海湾大酒店

[了解更多>>](#)

深入浅出 ES6：生成器 Generators



Jason Orendorff，
Mozilla JS 引擎黑客，
4 个孩子的老爸。



刘振涛，Web 开发领域新生，近期关注 Ecmascript 6，爱好摄影、网球，重度信息癖（Infomania）患者。

今天的这篇文章令我感到非常兴奋，我们将一起领略 ES6 中最具魔力的特性。

为什么说是“最具魔力的”？对于初学者来说，此特性与 JS 之前已有的特性截然不同，可能会觉得有点晦涩难懂。但是，从某种意义上来说，它使语言内部的常态行为变得更加强大，如果这都不算有魔力，我不知道还有什么能算。

不仅如此，此特性可以极大地简化代码，它甚至可以帮助你逃离“回调地狱”。

既然新特性如此神奇，那么就一起深入了解它的魔力吧！

ES6 生成器（Generators）简介

什么是生成器？

我们从一个示例开始：

```
function* quips(name) {
  yield "你好 " + name + "!";
  yield "希望你能喜欢这篇介绍 ES6 的译文";
  if (name.startsWith("X")) {
    yield "你的名字 " + name + " 首字母是 X，这很酷！";
  }
  yield "我们下次再见！";
}
```

这是一只[会说话的猫](#)，这段代码很可能代表着当今互联网上最重要的一类应用。（试着点击[这个链接](#)，与这只猫互动一下，如果你感到有些困惑，回到这里继续阅读）。

这段代码看起来很像一个函数，我们称之为生成器函数，它与普通函数有很多共同点，但是二者有如下区别：

- 普通函数使用 `function` 声明，而生成器函数使用 `function*` 声明；
- 在生成器函数内部，有一种类似 `return` 的语法：关键字 `yield`。二者的区别是，普通函数只可以 `return` 一次，而生成器函数可以 `yield` 多次（当然也可以只 `yield` 一次）。在生成器的执行过程中，遇到 `yield` 表达式立即暂停，后续可恢复执行状态。

这就是普通函数和生成器函数之间最大的区别，普通函数不能自暂停，生成器函数可以。

生成器做了什么？

当你调用 `quips()` 生成器函数时发生了什么？

```
> var iter = quips("jorendorff");
[object Generator]
> iter.next()
{ value: "你好 jorendorff!", done: false }
> iter.next()
{ value: "希望你能喜欢这篇介绍 ES6 的译文", done: false }
> iter.next()
{ value: "我们下次再见！", done: false }
```

```
> iter.next()  
{ value: undefined, done: true }
```

你大概已经习惯了普通函数的使用方式，当你调用它们时，它们立即开始运行，直到遇到 `return` 或抛出异常时才退出执行，作为 JS 程序员你一定深谙此道。

生成器调用看起来非常类似：`quips("jorendorff")`。但是，当你调用一个生成器时，它并非立即执行，而是返回一个已暂停的生成器对象（上述实例代码中的 `iter`）。你可将这个生成器对象视为一次函数调用，只不过立即冻结了，它恰好在生成器函数的最顶端的第一行代码之前冻结了。

每当你调用生成器对象的 `.next()` 方法时，函数调用将其自身解冻并一直运行到下一个 `yield` 表达式，再次暂停。

这也是在上述代码中我们每次都调用 `iter.next()` 的原因，我们获得了 `quips()` 函数体中 `yield` 表达式生成的不同的字符串值。

调用最后一个 `iter.next()` 时，我们最终抵达生成器函数的末尾，所以返回结果中 `done` 的值为 `true`。抵达函数的末尾意味着没有返回值，所以返回结果中 `value` 的值为 `undefined`。

现在回到[会说话的猫的 demo 页面](#)，尝试在循环中加入一个 `yield`，会发生什么？

如果用专业术语描述，每当生成器执行 `yields` 语句，生成器的堆栈结构（本地变量、参数、临时值、生成器内部当前的执行位置）被移出堆栈。然而，生成器对象保留了对这个堆栈结构的引用（备份），所以稍后调用 `.next()` 可以重新激活堆栈结构并且继续执行。

值得特别一提的是，**生成器不是线程**，在支持线程的语言中，多段代码可以同时运行，通常导致竞态条件和非确定性，不过同时也带来不错的性能。生成器则完全不同。当生成器运行时，它和调用者处于同一线程中，拥有确定的连续执行顺序，永不并发。与系统线程不同的是，生成器只有在其函数体内标记为 `yield` 的点才会暂停。

现在，我们了解了生成器的原理，领略过生成器的运行、暂停恢复运行的不同状态。那么，这些奇怪的功能究竟有何用处？

生成器是迭代器！

上周，我们学习了 ES6 的迭代器，它是 ES6 中独立的内建类，同时也是语言的一个扩展点，通过实现 `[Symbol.iterator]()` 和 `.next()` 两个方法你就可以创建自定义迭代器。

实现一个接口不是一桩小事，我们一起实现一个迭代器。举个例子，我们创建一个简单的 range 迭代器，它可以简单地将两个数字之间的所有数相加。首先是传统 C 的 for(;;)循环：

```
// 应该弹出三次 "ding"
for (var value of range(0, 3)) {
  alert("Ding! at floor #" + value);
}
```

使用 ES6 的类的解决方案（如果不清楚语法细节，无须担心，我们将在接下来的文章中为你讲解）：

```
class RangeIterator {
  constructor(start, stop) {
    this.value = start;
    this.stop = stop;
  }

  [Symbol.iterator]() { return this; }

  next() {
    var value = this.value;
    if (value < this.stop) {
      this.value++;
      return {done: false, value: value};
    } else {
      return {done: true, value: undefined};
    }
  }
}

// 返回一个新的迭代器，可以从 start 到 stop 计数。
function range(start, stop) {
  return new RangeIterator(start, stop);
}
```

[查看代码运行情况](#)

这里的实现类似 [Java](#) 或 [Swift](#) 中的迭代器，不是很糟糕，但也不是完全没有问题。我们很难说清这段代码中是否有 bug，这段代码看起来完全不像我们试图模仿的传统 for (;;)循环，迭代器协议迫使我们拆解掉循环部分。

此时此刻你对迭代器可能尚无感觉，他们用起来很酷，但看起来有些难以实现。

你大概不会为了使迭代器更易于构建从而建议我们为 JS 语言引入一个离奇古怪又野蛮的新型控制流结构，但是既然我们有生成器，是否可以在这里应用它们呢？一起尝试一下：

```
function* range(start, stop) {
  for (var i = start; i < stop; i++)
    yield i;
}
```

[查看代码运行情况](#)

以上 4 行代码实现的生成器完全可以替代之前引入了一整个 **Rangelterator** 类的 23 行代码的实现。可行的原因是：**生成器是迭代器**。所有的生成器都有内建.next()和[Symbol.iterator]()方法的实现。你只须编写循环部分的行为。

我们都非常讨厌被迫用被动语态写一封很长的邮件，不借助生成器实现迭代器的过程与之类似，令人痛苦不堪。当你的语言不再简练，说出的话就会变得难以理解。**Rangelterator** 的实现代码很长并且非常奇怪，因为你需要在不借助循环语法的前提下为它添加循环功能的描述。所以生成器是最好的解决方案！

我们如何发挥作为迭代器的生成器所产生的最大效力？

1. 使任意对象可迭代。编写生成器函数遍历这个对象，运行时 `yield` 每一个值。然后将这个生成器函数作为这个对象的[Symbol.iterator]方法。
2. 简化数组构建函数。假设你有一个函数，每次调用的时候返回一个数组结果，就像这样：

```
// 拆分一维数组 icons
// 根据长度 rowLength
function splitIntoRows(icons, rowLength) {
  var rows = [];
  for (var i = 0; i < icons.length; i += rowLength) {
    rows.push(icons.slice(i, i + rowLength));
  }
  return rows;
}
```

使用生成器创建的代码相对较短：

```
function* splitIntoRows(icons, rowLength) {
  for (var i = 0; i < icons.length; i += rowLength) {
    yield icons.slice(i, i + rowLength);
  }
}
```

行为上唯一不同的是，传统写法立即计算所有结果并返回一个数组类型的结果，使用生成器则返回一个迭代器，每次根据需要逐一地计算结果。

- 获取异常尺寸的结果。你无法构建一个无限大的数组，但是你可以返回一个可以生成一个永无止境的序列的生成器，每次调用可以从中取任意数量的值。
- 重构复杂循环。你是否写过又丑又大的函数？你是否愿意将其拆分为两个更简单的部分？现在，你的重构工具箱里有了新的利刃——生成器。当你面对一个复杂的循环时，你可以拆分出生成数据的代码，将其转换为独立的生成器函数，然后使用 `for (var data of myNewGenerator(args))` 遍历我们所需的数据。
- 构建与迭代相关的工具。ES6 不提供用来过滤、映射以及针对任意可迭代数据集进行特殊操作的扩展库。借助生成器，我们只须写几行代码就可以实现类似的工具。

举个例子，假设你需要一个等效于 `Array.prototype.filter` 并且支持 DOM `NodeLists` 的方法，可以这样写：

```
function* filter(test, iterable) {
  for (var item of iterable) {
    if (test(item))
      yield item;
  }
}
```

你看，生成器魔力四射！借助它们的力量可以非常轻松地实现自定义迭代器，记住，迭代器贯穿 ES6 的始终，它是数据和循环的新标准。

以上只是生成器的冰山一角，最重要的功能请继续观看！

生成器和异步代码

这是我在一段时间以前写的一些 JS 代码

```
  };
}
});
});
});
});
});
```

可能你已经在自己的代码中见过类似的片段，[异步 API](#) 通常需要一个回调函数，这意味着你需要为每一次任务执行编写额外的异步函数。所以如果你有一段代码需要完成三个

任务，你将看到类似的三层级缩进的代码，而非简单的三行代码。

后来我就这样写了：

```
}).on('close', function () {
  done(undefined, undefined);
}).on('error', function (error) {
  done(error);
});
```

异步 API 拥有错误处理规则，不支持异常处理。不同的 API 有不同的规则，大多数的错误规则是默认的；在有些 API 里，甚至连成功提示都是默认的。

这些是到目前为止我们为异步编程所付出的代价，我们正慢慢开始接受异步代码不如等效同步代码美观又简洁的这个事实。

生成器为你提供了避免以上问题的新思路。

实验性的 [Q.async\(\)](#) 尝试结合 promises 使用生成器产生异步代码的等效同步代码。举例：

```
// 制造一些噪音的同步代码。
function makeNoise() {
  shake();
  rattle();
  roll();
}

// 制造一些噪音的异步代码。
// 返回一个 Promise 对象
// 当我们制造完噪音的时候会变为 resolved
function makeNoise_async() {
  return Q.async(function* () {
    yield shake_async();
    yield rattle_async();
    yield roll_async();
  });
}
```

二者主要的区别是，异步版本必须在每次调用异步函数的地方添加 `yield` 关键字。

在 `Q.async` 版本中添加一个类似 `if` 语句的判断或 `try/catch` 块，如同向同步版本中添加类似功能一样简单。与其它异步代码编写方法相比，这种方法更自然，不像是学一门新语言一样辛苦。

如果你已经看到这里，你可以试着阅读来自 James Long 的[更深入地讲解生成器的文章](#)。

生成器为我们提供了一个新的异步编程模型思路，这种方法更适合人类的大脑。相关工作正在不断展开。此外，更好的语法或许会有帮助，[ES7 中有一个有关异步函数的提案](#)，它基于 promises 和生成器构建，并从 C# 相似的特性中汲取了大量灵感。

如何应用这些疯狂的新特性？

在服务器端，现在你可以在 io.js 中使用 ES6（在 Node 中你需要使用--harmony 这个命令行选项）。

在浏览器端，到目前为止只有 Firefox 27+ 和 Chrome 39+ 支持了 ES6 生成器。如果要在 web 端使用生成器，你需要使用 [Babel](#) 或 [Traceur](#) 来将你的 ES6 代码转译为 Web 友好的 ES5。

起初，JS 中的生成器由 Brendan Eich 实现，他的设计参考了 [Python 生成器](#)，而此 Python 生成器则受到 [Icon](#) 的启发。他们[早在 2006 年](#)就在 Firefox 2.0 中移植了相关代码。但是，标准化的道路崎岖不平，相关语法和行为都在原先的基础上有所改动。Firefox 和 Chrome 中的 ES6 生成器都是由编译器 hacker [Andy Wingo](#) 实现的。这项工作由[彭博](#)赞助支持（没听错，就是大名鼎鼎的那个彭博！）。

yield；

生成器还有更多未提及的特性，例如：.throw() 和 .return() 方法、可选参数.next()、yield* 表达式语法。由于行文过长，估计观众老爷们已然疲乏，我们应该学习一下生成器，暂时 yield 在这里，剩下的干货择机为大家献上。

下一次，我们变换一下风格，由于我们接连搬了两座大山：迭代器和生成器，下次就一起研究下不会改变你编程风格的 ES6 特性好不？就是一些简单又实用的东西，你一定会喜笑颜开哒！你还别说，在什么都要“微”一下的今天，ES6 当然要有微改进了！

深入浅出 React：React 开发神器 Webpack



王沛，2007 年研究生毕业于南京大学计算机与科学技术系，之后工作于 IBM，从事多种企业级产品前端框架的设计和开发。期间参与 Dojo 开源项目，成为代码贡献者。并参与创建和设计了基于 Dojo 的 [GridX 项目](#)。

本文介绍用于 React 开发和模块管理的主流工具 Webpack。称之为 React 开发神器有点标题党了，不过 Webpack 确实是笔者见过的功能最为强大的前端模块管理和打包工具。虽然 Webpack 是一个通用的工具，并不只适合于 React，但是很多 React 的文章或者项目都使用了 Webpack，尤其是 [react-hot-loader](#) 这样的神器存在，让 Webpack 成为最主流的 React 开发工具。

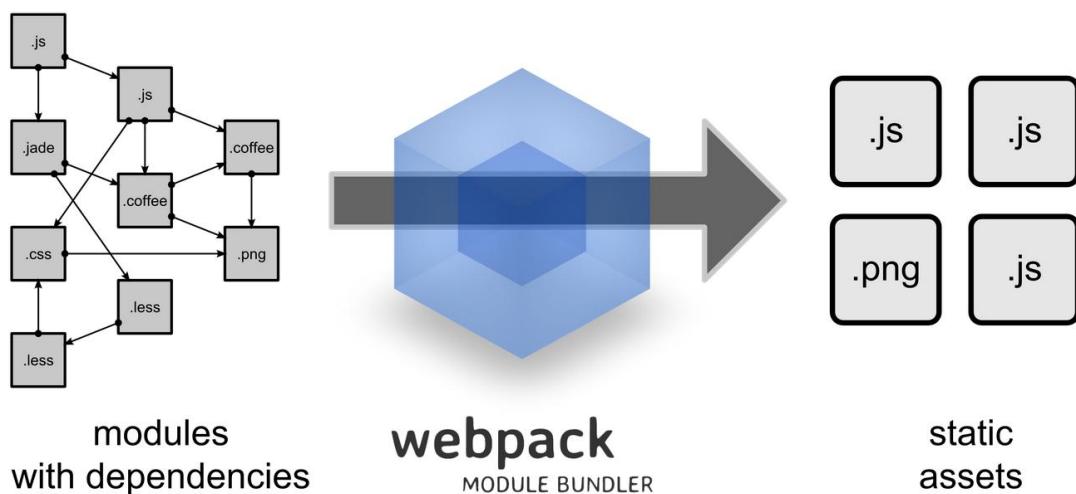
CommonJS 和 AMD 是用于 JavaScript 模块管理的两大规范，前者定义的是模块的同步加载，主要用于 NodeJS；而后者则是异步加载，通过 requirejs 等工具适用于前端。随着 npm 成为主流的 JavaScript 组件发布平台，越来越多的前端项目也依赖于 npm 上的项目，或者自身就会发布到 npm 平台。因此，让前端项目更方便的使用 npm 上的资源成为一大需求。于是诞生了类似 [browserify](#) 这样的工具，代码中可以使用 require 函数直接以同步语法形式引入 npm 模块，打包后再由浏览器执行。

Webpack 其实有点类似 browserify，出自 Facebook 的 Instagram 团队，但功能比 browserify 更为强大。其主要特性如下：

1. 同时支持 [CommonJS](#) 和 [AMD](#) 模块（对于新项目，推荐直接使用 CommonJS）；

2. 串联式模块加载器以及插件机制，让其具有更好的灵活性和扩展性，例如提供对 CoffeeScript、ES6 的支持；
3. 可以基于配置或者智能分析打包成多个文件，实现公共模块或者按需加载；
4. 支持对 CSS，图片等资源进行打包，从而无需借助 Grunt 或 Gulp；
5. 开发时在内存中完成打包，性能更快，完全可以支持开发过程的实时打包需求；
6. 对 sourcemap 有很好的支持，易于调试。

Webpack 将项目中用到的一切静态资源都视之为模块，模块之间可以互相依赖。Webpack 对它们进行统一的管理以及打包发布，其官方主页用下面这张图来说明 Webpack 的作用：



可以看到 Webpack 的目标就是对项目中的静态资源进行统一管理，为产品的最终发布提供最优的打包部署方案。本文就将围绕 React 对其相关用法做一个总体介绍，从而能让你将其应用在自己的实际项目之中。

安装 Webpack，并加载一个简单的 React 组件

Webpack 一般作为全局的 npm 模块安装：

```
npm install -g webpack
```

之后便有了全局的 webpack 命令，直接执行此命令会默认使用当前目录的 webpack.config.js 作为配置文件。如果要指定另外的配置文件，可以执行：

```
webpack --config webpack.custom.config.js
```

尽管 Webpack 可以通过命令行来指定参数，但我们通常会将所有相关参数定义在配置文件中。一般我们会定义两个配置文件，一个用于开发时，另外一个用于产品发布。生产环境下的打包文件不需要包含 sourcemap 等用于开发时的代码。配置文件通常放在项目根目录之下，其本身也是一个标准的 CommonJS 模块。一个最简单的 Webpack 配置文件 webpack.config.js 如下所示：

```
module.exports = {
  entry:[
    './app/main.js'
  ],
  output: {
    path: __dirname + '/assets/',
    publicPath: "/assets/",
    filename: 'bundle.js'
  }
};
```

其中 `entry` 参数定义了打包后的入口文件，数组中的所有文件会按顺序打包。每个文件进行依赖的递归查找，直到所有相关模块都被打包。`output` 参数定义了输出文件的位置，其中常用的参数包括：

- **path:** 打包文件存放的绝对路径。
- **publicPath:** 网站运行时的访问路径。
- **filename:** 打包后的文件名。

现在来看如何打包一个 React 组件。假设有如下项目文件夹结构：

```
- react-sample
+ assets/
- js/
  Hello.js
  entry.js
index.html
webpack.config.js
```

其中 `Hello.js` 定义了一个简单的 React 组件，使用 ES6 语法：

```
var React = require('react');
class Hello extends React.Component {
  render() {
    return (
      <h1>Hello {this.props.name}</h1>
    );
}
```

entry.js 是入口文件，将一个 Hello 组件输出到界面：

```
var React = require('react');
var Hello = require('./Hello');
React.render(<Hello name="Nate" />, document.body);
```

index.html 的内容如下：

```
<html>
<head></head>
<body>
<script src="/assets/bundle.js"></script>
</body>
</html>
```

在这里 Hello.js 和 entry.js 都是 JSX 组件语法，需要对它们进行预处理，这就要引入 webpack 的 JSX 加载器。因此在配置文件中加入如下配置：

```
module: {
  loaders: [
    { test: /\.jsx?$/, loaders: ['jsx?harmony']}
  ]
}
```

加载器的概念稍后还会详细介绍，这里只需要知道它能将 JSX 编译成 JavaScript 并加载为 Webpack 模块。这样在当前目录执行 webpack 命令之后，在 assets 目录将生成 bundle.js，打包了 entry.js 的内容。当浏览器打开当前服务器上的 index.html，将显示“Hello Nate!”。这是一个非常简单的例子，演示了如何使用 Webpack 来进行最简单的 React 组件打包。

加载 AMD 或 CommonJS 模块

在实际项目中，代码以模块进行组织，AMD 是在 CommonJS 的基础上考虑了浏览器的异步加载特性而产生的，可以让模块异步加载并保证执行顺序。而 CommonJS 的 require 函数则是同步加载。在 Webpack 中笔者更加推荐 CommonJS 方式去加载模块，这种方式语法更加简洁直观。即使在开发时，我们也是加载 Webpack 打包后的文件，通过 sourcemap 去进行调试。

除了项目本身的模块，我们还需要依赖第三方的模块，现在比较常用第三方模块基本都通过 npm 进行发布，使用它们已经无需单独下载管理，需要时执行 npm install 即可。例如，我们需要依赖 jQuery，只需执行：

```
npm install jquery --save-dev
```

更多情况下我们是在项目的 package.json 中进行依赖管理，然后通过直接执行 npm install 来安装所有依赖。这样在项目的代码仓库中并不需要存储实际的第三方依赖库的代码。

安装之后，在需要使用 jquery 的模块中需要在头部进行引入：

```
var $ = require('jquery');
$('body').html('Hello Webpack!');
```

可以看到，这种以 CommonJS 的同步形式去引入其它模块的方式代码更加简洁。浏览器并不会实际的去同步加载这个模块，require 的处理是由 Webpack 进行解析和打包的，浏览器只需要执行打包后的代码。Webpack 自身已经可以完全处理 JavaScript 模块的加载，但是对于 React 中的 JSX 语法，这就需要使用 Webpack 的扩展加载器来处理了。

Webpack 开发服务器

除了提供模块打包功能，Webpack 还提供了一个基于 Node.js Express 框架的开发服务器，它是一个静态资源 Web 服务器，对于简单静态页面或者仅依赖于独立服务的前端页面，都可以直接使用这个开发服务器进行开发。在开发过程中，开发服务器会监听每一个文件的变化，进行实时打包，并且可以推送通知前端页面代码发生了变化，从而可以实现页面的自动刷新。

Webpack 开发服务器需要单独安装，同样是通过 npm 进行：

```
npm install -g webpack-dev-server
```

之后便可以运行 webpack-dev-server 命令来启动开发服务器，然后通过 localhost:8080/webpack-dev-server/ 访问到页面了。默认情况下服务器以当前目录作为服务器目录。在 React 开发中，我们通常会结合 react-hot-loader 来使用开发服务器，因此这里不做太多介绍，只需要知道有这样一个开发服务器可以用于开发时的内容实时打包和推送。详细配置和用法可以参考[官方文档](#)。

Webpack 模块加载器（Loaders）

Webpack 将所有静态资源都认为是模块，比如 JavaScript，CSS，LESS，TypeScript，JSX，CoffeeScript，图片等等，从而可以对其进行统一管理。为此 Webpack 引入了加载器的概念，除了纯 JavaScript 之外，每一种资源都可以通过对应的加载器处理成模块。和大多数包管理器不一样的是，Webpack 的加载器之间可以进行串联，一个加载器的输出可以成为

另一个加载器的输入。比如 LESS 文件先通过 less-load 处理成 css，然后再通过 css-loader 加载成 css 模块，最后由 style-loader 加载器对其做最后的处理，从而运行时可以通过 style 标签将其应用到最终的浏览器环境。

对于 React 的 JSX 也是如此，它通过 jsx-loader 来载入。jsx-loader 专门用于载入 React 的 JSX 文件，Webpack 的加载器支持参数，jsx-loader 就可以添加?harmony 参数使其支持 ES6 语法。为了让 Webpack 识别什么样的资源应该用什么加载器去载入，需要在配置文件进行配置：通过正则表达式对文件名进行匹配。例如：

```
module: {
  preLoaders: [
    {
      test: /\.js$/,
      exclude: /node_modules/,
      loader: 'jsxhint'
    },
    {
      loaders: [
        {
          test: /\.js$/,
          exclude: /node_modules/,
          loader: 'react-hot!jsx-loader?harmony'
        },
        {
          test: /\.less/,
          loader: 'style-loader!css-loader!less-loader'
        },
        {
          test: /\.css$/,
          loader: 'style-loader!css-loader'
        },
        {
          test: /\.(png|jpg)$/,
          loader: 'url-loader?limit=8192'
        }
      ]
    }
}
```

可以看到，该使用什么加载器完全取决于这里的配置，即使对于 JSX 文件，我们也可以用 js 作为后缀，从而所有的 JavaScript 都可以通过 jsx-loader 载入，因为 jsx 本身就是完全兼容 JavaScript 的，所以即使没有 JSX 语法，普通 JavaScript 模块也可以使用 jsx-loader 来载入。

加载器之间的级联是通过感叹号来连接，例如对于 LESS 资源，写法为 style-loader!css-loader!less-loader。对于小型的图片资源，也可以将其进行统一打包，由 url-loader 实现，代码中 url-loader?limit=8192 含义就是对于所有小于 8192 字节的图片资源也进行打包。这在一定程度上可以替代 [Css Sprites](#) 方案，用于减少对于小图片资源的 HTTP 请求数量。

除了已有加载器，你也可以自己[实现自己的加载器](#)，从而可以让 Webpack 统一管理项目特定的静态资源。现在也有很多第三方的加载器实现常见静态资源的打包管理，可以参考 Webpack 主页上的[加载器列表](#)。

React 开发神器：react-hot-loader

Webpack 本身具有运行时模块替换功能，称之为[Hot Module Replacement \(HMR\)](#)。当某个模块代码发生变化时，Webpack 实时打包将其推送到页面并进行替换，从而无需刷新页面就实现代码替换。这个过程相对比较复杂，需要进行多方面考虑和配置。而现在针对 React 出现了一个第三方[react-hot-loader](#)加载器，使用这个加载器就可以轻松实现 React 组件的热替换，非常方便。其实正是因为 React 的每一次更新都是全局刷新的虚拟 DOM 机制，让 React 组件的热替换可以成为通用的加载器，从而极大提高开发效率。

要使用 react-hot-loader，首先通过 npm 进行安装：

```
npm install -save-dev react-hot-loader
```

之后，Webpack 开发服务器需要开启 HMR 参数 hot，为了方便，我们创建一个名为 server.js 的文件用以启动 Webpack 开发服务器：

```
var webpack = require('webpack');
var WebpackDevServer = require('webpack-dev-server');
var config = require('../webpack.config');
new WebpackDevServer(webpack(config), {
  publicPath: config.output.publicPath,
  hot: true,
  noInfo: false,
  historyApiFallback: true
}).listen(3000, '127.0.0.1', function (err, result) {
  if (err) {
    console.log(err);
  }
  console.log('Listening at localhost:3000');
});
```

为了热加载 React 组件，我们需要在前端页面中加入相应的代码，用以接收 Webpack 推送过来的代码模块，进而可以通知所有相关 React 组件进行重新 Render。加入这个代码很简单：

```
entry: [
  'webpack-dev-server/client?http://127.0.0.1:3000', // WebpackDevServer host and port
  'webpack/hot/only-dev-server',
  './scripts/entry' // Your app's entry point
]
```

需要注意的是，这里的 client?<http://127.0.0.1:3000> 需要和在 server.js 中启动 Webpack 开发服务器的地址匹配。这样，打包生成的文件就知道该从哪里去获取动态的代码更新。下一步，我们需要让 Webpack 用 react-hot-loader 去加载 React 组件，如上一节所介绍，这通过加载器配置完成：

```
loaders: [
  {
    test: /\.js$/,
    exclude: /node_modules/,
    loader: 'react-hot!jsx-loader?harmony'
  },
  ...
]
```

做完这些配置之后，使用 Node.js 运行 server.js：

```
node server.js
```

即可启动开发服务器并实现 React 组件的热加载。为了方便，我们也可以在 package.json 中加入一节配置：

```
"scripts": {
  "start": "node ./js/server.js"
}
```

从而通过 npm start 命令即可启动开发服务器。示例代码也上传在 [Github](#) 上，大家可以参考。

这样，React 的热加载开发环境即配置完成，任何修改只要以保存，就会在页面上立刻体现出来。无论是对样式修改，还是对界面渲染的修改，甚至事件绑定处理函数的修改，都可以立刻生效，不得不说是提高开发效率的神器。

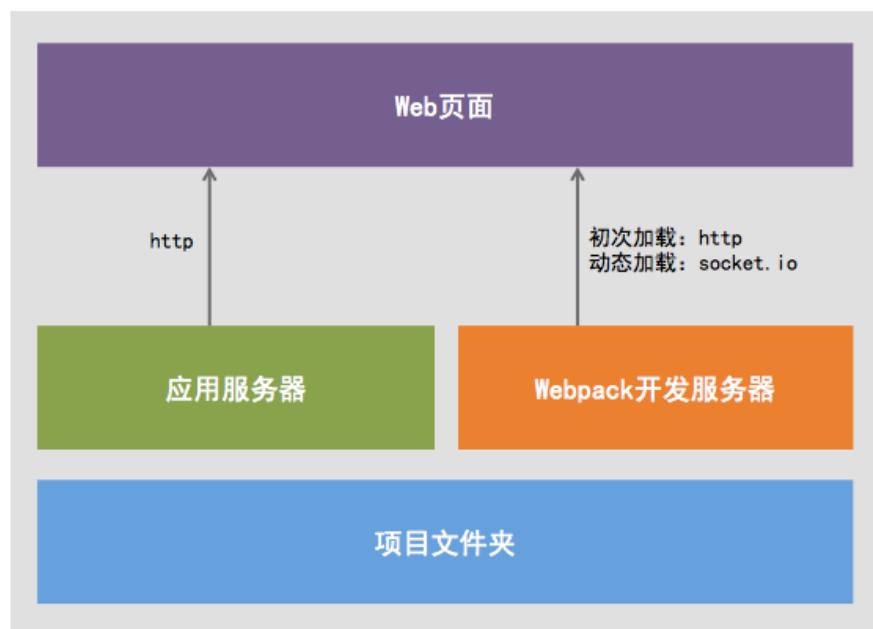
将 Webpack 开发服务器集成到已有服务器

尽管 Webpack 开发服务器可以直接用于开发，但实际项目中我们可能必须使用自己的 Web 服务器。这就需要我们能将 Webpack 的服务集成到已有服务器，来使用 Webpack 提供的

模块打包和加载功能。要实现这一点其实非常容易，只需要在载入打包文件时指定完整的 URL 地址，例如：

```
<script src="http://127.0.0.1:3000/assets/bundle.js"></script>
```

这就告诉当前页面应该去另外一个服务器获得脚本资源文件，在之前我们已经在配置文件中指定了开发服务器的地址，因此打包后的文件也知道应该通过哪个地址去建立 Socket IO 来动态加载模块。整个资源架构如下图所示：



打包成多个资源文件

将项目中的模块打包成多个资源文件有两个目的：

1. 将多个页面的公用模块独立打包，从而可以利用浏览器缓存机制来提高页面加载效率；
2. 减少页面初次加载时间，只有当某功能被用到时，才去动态的加载。

Webpack 提供了非常强大的功能让你能够灵活的对打包方案进行配置。首先来看如何创建多个入口文件：

```
{
  entry: { a: "./a", b: "./b" },
  output: { filename: "[name].js" },
  plugins: [ new webpack.CommonsChunkPlugin("init.js") ]
}
```

可以看到，配置文件中定义了两个打包资源“a”和“b”，在输出文件中使用方括号来获得输出文件名。而在插件设置中使用了 CommonsChunkPlugin，Webpack 中将打包后的文件都称之为“Chunk”。这个插件可以将多个打包后的资源中的公共部分打包成单独的文件，这里指定公共文件输出为“init.js”。这样我们就获得了三个打包后的文件，在 html 页面中可以这样引用：

```
<script src="init.js"></script>
<script src="a.js"></script>
<script src="b.js"></script>
```

除了在配置文件中对打包文件进行配置，还可以在代码中进行定义：require.ensure，例如：

```
require.ensure(["module-a", "module-b"], function(require) {
  var a = require("module-a");
  // ...
});
```

Webpack 在编译时会扫描到这样的代码，并对依赖模块进行自动打包，运行过程中执行到这段代码时会自动找到打包后的文件进行按需加载。

小结

本文结合 React 介绍了 Webpack 的基本功能和用法，希望能让大家对这个新兴而强大的模块管理工具有一个总体的认识，并能将其应用在实际的项目开发中。笔者也将其应用在之前提供的 [React 示例组件](#)项目中，大家可以参考。除了这里介绍的功能，Webpack 还有许多强大的特性，例如插件机制、支持动态表达式的 require、打包文件的智能重组、性能优化、代码混淆等等。限于篇幅不再一一介绍，其[官方文档](#)也非常完善，需要时可以参考。

姜宁谈红帽绩效考核：不关心员工具体做什么



作者郭蕾



姜宁，有十余年企业级开源中间件开发经验，有丰富的 Java 开发和使用经验，函数式编程爱好者。从 2006 年开始一直从事 Apache 开源中间件项目的开发工作，先后参与 Apache CXF、Apache Camel 以及 Apache ServiceMix 的开发。2007 年开始参与 Apache Camel 项目开发，目前是 Apache Camel 项目的主要维护者。对 WebServices、Enterprise Integration Pattern、SOA、OSGi 有比较深入的研究。

开源软件是指某个由社区驱动的开放源代码的产品，而开源文化是指开源社区所衍生出的团队沟通、协作和管理的理念。在开源社区中，程序代码直接决定着项目的成败，但人更是核心，没有人便没有代码。这一点和企业是相同的。越来越多的企业意识到，开源所提倡的协作和管理方式同样适用于企业管理，开源文化能够提高员工的积极性，从而提高生产效率。那到底什么是开源文化，企业应该建立怎么样的开源文化？为了回答这些问题，InfoQ 编辑采访了一直深耕于开源社区的 RedHat 工程师姜宁。另外，姜宁还将在 ArchSummit 全球架构师峰会上分享题为 [《如何在企业开发中引入开源项目成功模式》](#) 的演讲，敬请关注。

InfoQ: 你认为公司是否应该有开源文化?

姜宁: 开源文化对公司的影响非常大，开源提倡的是开放和协作。如果一个公司不具备这样的开源文化，那就很难真正把开源项目做好，也很难真正靠开源来推动实现公司的商业目标。以我的个人经历来说，我先在国企待了三年，然后一直都在外企上班，在外企期间先后被换了三家公司（主要是公司被并购），但我做的主要工作一直都是维护 Apache 上面的中间件项目。在我待过的这三家外企公司中，红帽软件是一家有着开源基因的公司（本身是靠做 Linux 发行版起家的公司），也是一家真正把开源作为企业文化并在管理组织架构中应用开源文化来提升效率的公司。红帽软件的企业目标和口号是『成为用户、贡献者以及合作伙伴社区的催化剂，让大家能够以开源的方式创造出更好的科技』。

InfoQ: 开源文化对公司有怎么样的帮助?

姜宁: 开源打破了信息壁垒，让创新的火花四溅。在开源社区中大家通过分享收获成长，开源文化是参与开源项目的人们认同的东西，如果公司认同了开源文化并鼓励开源，相当时是公司与员工在价值观上是相互认同的。我们现在处在一个信息爆炸的时代，公司为了能够跟上时代的步伐必须通过创新来保证其核心竞争力，而开源文化能够最大限度的鼓励创新，我想这也是现在越来越多的公司开始拥抱开源的最主要的原因。

InfoQ: 要建立开源文化，是不是公司相应的组织架构、管理模式都应该做相应的调整?

姜宁: 传统的公司组织架构是一个自顶向下的，这样的组织架构是为管理者服务的，对于底层的员工来说大家只需要按照领导的意志或者安排做事情就可以了。这种组织是由以前常规的制造企业创建，组织架构只需要员工按照标准重复生产就可以，所有的决策都是由高层最终决定的。这样的组织架构一旦上层做出决定，那就很难被改变，它既不适合快速的市场变化，也很难激发员工的创新热情。反观大多数开源项目，由于做的东西都是新的，没有人在最开始就知道项目最终要做什么样的，大部分成功的开源项目都是先通过一个好点子构建一个原型，在社区的需求下不断激发出新的点子，不断修复原有的问题，不断演进，最终成长起来。在开源社区中，由于大家都没有公司的层级概念，靠升职加薪来激励员工的管理方式很难奏效。开源项目的成功靠的是大家发自内心对开源项目的认同，靠的是技术领导以及开发人员、社区贡献者的热情与努力。开源的项目中没有项目经理，只有技术领导，而技术领导的话语权是通过其长期在社区中的贡献而建立起来的。由此可见开源的文化和我们传统的管理组织架构有太多的不同，红

帽软件的 CEO Jim WhiteHust 最近写了一本叫做《[The Open Organisation](#)》的书，在书中他详细阐述了红帽软件是如何将开源文化应用到公司架构中去的。

InfoQ：红帽是一家开源文化极浓的公司，能谈谈红帽是如何管理员工绩效的吗？

姜宁：的确，红帽雇佣了大量专职参与上游（红帽提供的企业版软件基本都是基于开源社区的项目经过打包测试之后制作的发行版，我们把那些社区项目成为上游）开发的工程师，有意思的是这些工程师大部分都是在家上班的。这些工程师如果按照传统企业绩效考核基本上都不合格，因为首先经理看不到你是否工作，也不知道你什么时候上班什么时候下班，更重要的是经理不会给你安排工作内容。红帽软件对于这些工程师的绩效考核核心内容是工程师在社区的活跃程度，以及工程师的影响力，而不是他具体完成了多少工作。红帽有一个叫做 Compass 的绩效考核网站，经理和员工会定期讨论工作重点，制定发展计划。有意思的是我在和经理进行绩效考评的过程中，经理问得最多的是不是你完成哪些事情，而是你对自己的未来有什么规划，你打算学点什么，你的成长目标是什么。这些讨论的最终结果是将你的发展规划和公司工作目标结合起来，在保证公司目标实现的前提下，你可以任意选择你感兴趣的工作内容。在《[只是为了好玩](#)》这本书中提到了一个问题，为什么人们愿意甚至渴望在互联网上为 Linux 这样的项目工作？问题的答案是 Linux 能使人们通过挑战智力而获得乐趣，又能通过产业开发工作获得一种被社会需要的满足感。我觉得问题的答案可以比较好的解释为什么红帽的绩效考核会是这样的。

InfoQ：公司应该如何培养员工的开源文化、开源意识？

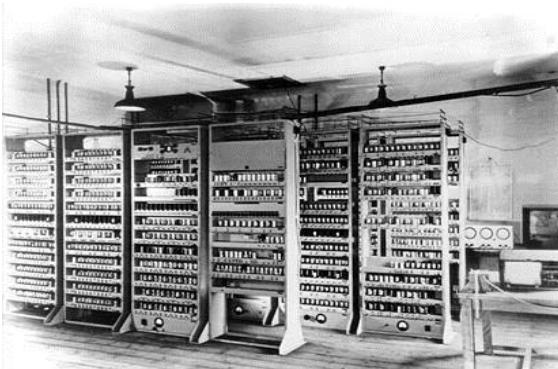
姜宁：我觉得首先公司管理层需要了解开源，认同开源文化。开源的核心是开放与协作。公司应该鼓励大家分享，当然是在不泄露公司商业机密的前提下。另外大部分开源项目的协作是跨越公司边界的，公司应该鼓励大家跨部门沟通协作。当然在参与开源过程中，有时可能出现公司和员工有分歧的情况，比如某个社区功能可能会很好玩，但是这项目或者这功能和公司目标并不一致，它很难直接为公司产生效益。如果公司的高层和员工在认识和处理这些差异的过程中发生了很大的分歧的话，就会给员工参与开源社区的开发带来很大的困扰。在这里我的建议是让管理层尽量少的干涉员工的具体工作，把大部分管理精力放在如何让员工认同公司的发展策略上，这样才能最大限度地将员工的发展目标与公司的发展目标统一起来，让员工能把所有的热情都投入到日常开发工作中去。

InfoQ: 相比于开源软件的开发和管理模式，你认为企业在软件研发过程中，有哪些值得改进的地方？

姜宁：传统的企业研发过程基本上就是先做需求，再做概念验证，最后实施。在传统的组织架构下，项目的参与者可能很难有机会影响到项目的决策，如果相关项目是一个比较新的项目（缺乏借鉴），往往比较容易失败。在开源项目中虽然也有失败的例子，但是这种一开始就方向性失败的例子很少见，这是因为开源项目的反馈非常及时。通过社区的使用，一旦发现设计缺陷，就会立即修正，周期不会很长。

如果说如何改进的话，我觉得主要是管理层需要从如何提高员工工作热情入手，创造条件鼓励员工之间协作，让创新的火花四溅。这样的改进不是一朝一夕就能完成的，我曾经帮助过一家传统的软件开发企业改进研发流程，很多时候我认为是习以为常的东西，但如果我没有开源项目开发经验的话，大家接受和实施过程中都感觉困难重重。这里先做一个小小的预告，我会在这次的 [ArchSummit 全球架构师峰会](#) 上，结合我的经历为大家介绍如何将开源软件的管理方式应用到企业软件开发中去。

冯·诺依曼计算机将渐行渐远？



作者 张天雷

如果说图灵 (Alan Turing)奠定的是计算机的理论基础，那么冯·诺依曼 (John Von Neumann)则是将图灵的理论物化成为实际的物理实体，成为了计算机体系结构的奠基者。从第一台冯·诺依曼计算机诞生到今天已经过去了将近 70 年，计算机的技术与性能也都发生了巨大的变化，但整个主流体系结构依然是冯·诺依曼结构。

冯·诺依曼体系结构的特点：采用二进制，硬件由 5 个部分组成（运算器、控制器、存储器、输入设备和输出设备），提出了“存储程序”原理，使用同一个存储器，经由同一个总线传输，程序和数据统一存储同时在程序控制下自动工作。特别要指出，它的程序指令存储器和数据存储器是合并在一起的，程序指令存储地址和数据存储地址指向同一个存储器的不同物理位置。因为程序指令和数据都是用二进制码表示，且程序指令和被操作数据的地址又密切相关，所以早先选择这样的结构是合理的。

但是，随着对计算机处理速度要求的提高和对需要处理数据的种类、量级的增大，这种指令和数据共用一个总线的结构，使

得信息流的传输成为限制计算机性能的一个瓶颈，制约了数据处理速度的提高。由此，体现出了冯·诺依曼体系结构的局限性：

- 1、目前 CPU 的处理速度和内存容量的成长速率要远大于两者之间的流量，将大量数值从内存搬入搬出的操作占用了 CPU 大部分的执行时间，也造成了总线的瓶颈。
- 2、程序指令的执行是串行的，由程序计数器控制，这样使得即使有关数据已经准备好，也必须遵循逐条执行指令序列，影响了系统运行的速度。
- 3、存储器是线性编址，按顺序排列的地址访问，这是有利于存储和执行机器语言，适用于数值计算。但高级语言的存储采用的是一组有名字的变量，是按名字调用变量而非按地址访问，且高级语言中的每个操作对于任何数据类型都是通用的，不管采用何种数据结构，多维数组、二叉树还是图，最终在存储器上都必须转换成一维的线性存储模型进行存储。这些因素都导致了机器语言和高级语言之间存在很大的语义差距，这些语义差距之间的映射大部分都要由编译程序来完成，在很大程度上增加了编译程序的工作量。
- 4、冯·诺依曼体系结构计算机是为逻辑和数值运算而诞生的，它以 CPU 为中心，I/O 设备与存储器间的数据传送都要经过运算器，在数值处理方面已经达到很高的速度和精度，但非数值数据的处理效率比较低，需要在体系结构方面有革命性突破。

科学家们一直在努力突破传统的冯·诺依曼体系结构框架，对冯·诺依曼计算机进行改良，主要体现在：

1、将传统计算机只有一个处理器串行执行改成多个处理器并行执行，依靠时间上的重叠来提高处理效率，形成支持多指令流、多数据流的并行算法结构。

2、改变传统计算机控制流驱动的工作方式，设计数据流驱动的工作方式，只要数据准备好，就可以并行执行相关指令。

3、跳出采用电信号二进制范畴，选取其他物质作为执行部件和信息载体，如光子、量子或生物分子等。

近几年，在计算机体系结构研究方面也已经有了重大进展，越来越多的非冯计算机相继出现，如光子计算机、量子计算机、神经计算机以及 DNA 计算机等等。

光子计算机（Photonic computer）是一种采用光信号作为物质介质和信息载体，依靠激光束进入反射镜和透镜组成的阵列进行数值运算、逻辑操作和信息的存储和处理。它可以实现对复杂度高、计算量大、实时性强的任务的高效、并行处理，比普通电子计算机快 1000 倍，在图像处理、模式识别和人工智能方面有着非常巨大的应用前景。

神经计算机（Neural computer）是一种可以并行处理多种数据功能的神经网络计算机，它以神经元为处理信息的基本单元，将模仿大脑神经记忆的信息存放在神经元上。神经网络具有自组织、自学习、自适应及自修复功能，可以模仿人脑的判断能力和适应能力。美国科学家研究出的神经计算机可以模拟人的左脑和右脑，能识别语言文字和图形图像，能控制机器人行为，进行智能决策。它的左脑由 100 万个神经元组成，用于存储文字和语法规则，右脑由 1 万多个神经元组成，适用于图形图像识别。这将有可能成为人工智能

硬件发展的主攻方向。

量子计算机（Quantum computer）是遵循量子力学规律进行高速数学和逻辑运算、存储及处理量子信息的物理装置。量子计算机本身的特性，扩充了逻辑和数学理论，通过核自旋、光子、束缚离子和原子等制成的量子位，创造出经典条件下不可能存在的新的逻辑门。与经典的比特位不同，对量子位操作 1 次等同于对经典位操作 2 次，因为量子不像半导体只能记录 0 和 1，它可以同时表示多种状态。这些都为新的算法实现提供了条件，也为人工智能的发展提供了可能的硬件条件。

冯·诺依曼计算机以其技术成熟、价格低廉、软件丰富和大众的使用习惯，可能在今后很长的一段时期里还将为人类的工作和生活发挥着重要作用。当然，为了满足人们对计算机更快速、更高效、更方便的使用要求，为了让计算机能够模拟人脑神经元和脑电信号脉冲这样复杂的结构，就需要突破现有的体系结构框架并寻求新的物质介质作为计算机的信息载体，才能使计算机有质的飞跃。随着非冯计算机的商品化问世，我们将会迎来一个崭新的信息时代。

米兰花：有爱，生命就会开花



架构师 2015 年 7 月刊

每月 8 号出版

本期主编：徐川

流程编辑：丁晓昀

发行人：霍泰稳

读者反馈/投稿：editors@cn.infoq.com

InfoQ 中文站新浪微博：<http://weibo.com/infoqchina>

商务合作：sales@cn.infoq.com 15810407783

米兰花，(拉丁文名：*Aglaia odorata* Lour) 别名：四季米兰、碎米兰，.楝科、米仔兰属常绿灌木或小乔木；属楠科，米仔兰属常绿小乔木，适应温暖多湿的气候条件，成年植株需充足阳光。其枝叶茂密，叶色葱绿光亮，一年内多次开花，夏秋最盛。开花时清香四溢，气味似兰花。小枝顶部常有星状锈色小鳞片；

羽状复叶互生，花呈黄色，味极香，直径约 2 毫米。两性花梗稍短而粗，花萼 5 裂，花瓣 5 枚，长圆形；浆果近球形，长 10~12 毫米，花期为夏秋间。

在南方庭院中米兰又是极好的风景树。

大话“技术选型”

PHP是不是最好的语言



深圳大梅沙京基海湾大酒店

2015年7月19日

EGO EXTRA GEEKS' ORGANIZATION
NETWORKS

Brought by **Geekbang** 极客邦科技 **InfoQ**



ArchSummit

全球架构师峰会 2015

中国·北京
International Architect Summit

北京站/即将启动

2015.12.18-19

北京国际会议中心

www.archsummit.com