时刻关注企业软件开发领域的变化与创新

架构师

5月 ARCHITECT

特别专题

我眼中的网站架构

Sina App Engine数据存储服务架构

又拍网架构中的分库设计

我眼中的云端架构

Trunk.ly CTO董洵谈架构

探索式测试的秘密

项目的故事

虚拟研讨会：Node.js生态系统之框架、库、最佳实践

Java深度历险（七）--Java反射与动态代理

书评和采访：ExtJS in Action

InfoQueue

Enterprise Software Development Community

每月8号出版

篇首语 | Editor's letter

站点架构闲谈

我们正处于战国时代，一个每天都有新站点诞生和消亡的时代，一个名副其实的"网络应用"的战国时代。随着技术的发展，曾被视为"高科技"的网站，现在已经成了随便几个未毕业的大学生就可凭借类似 ROR 或 PHP 这样的工具在几天内搭建出来的"高科技大白菜"。然而，入门门槛的降低并不意味着构建成功的站点就是一门简单的"手艺"。相反，它反而更加拉开了"优秀"和"拙劣"之间的距离，加速了网络世界的新陈代谢。

网络基础设施的成熟，带来了应用的繁荣。如今的站点，不论从数量还是从应用，都远远超过了 10 年前的情形。环顾四周，你认识不认识的人都在嚷嚷着要利用现今发达的网络做点什么。而 Twitter 上也时常会见到新站点发布的消息。难怪著名投资人 Paul Kedrosky 今年 3 月在他那未必是危言耸听的文章里将"记者们正在不断退出新闻行业，转向创业。"列为硅谷又一次走向衰落的迹象的第一条。

但是不绝于耳的警告声并不能扑灭人们的创业激情。每天，一波又一波的站点会照常在网络上升起，等待世俗普罗大众的检验。从这一刻起，它已不是作为一件成品出现在世人的面前，而是作为一个婴儿哇哇落地，开始它不断演变的历程。架构是不断演变的，它是对不断出现的新变化的一种自然的应变。跟传统的应用相比，网站架构在这一点上体现更明显，而且速度要求更快。稍有不慎，就很可能被对手甩下一大截。Trunk.ly 网站的 CTO 董洵在接受采访时表示，在一开始，你所有的重点不是去设计一个如何伸缩性强的架构。恰恰相反，你需要做的是快速实现你的想法，去验证它，而后快速地根据情况调整策略。Trunk.ly 的发展也验证了他的说法，而且这个过程仍然在继续之中。在网络上，像这样的例子并不少见，例如已在 InfoQ 上介绍过的豆瓣、又拍都有这样的过程。

相比起前面说的网站，新浪和百度则走上了另外的道路。在 Google 推出 GAE 之后，两家公司也紧随其后，各自推出了自己的 APP 引擎。网络平台化，在这类网站的身上得到了体现。还记得 Sun 提出的著名的"网络即计算机"吗？我们现在正看着它慢慢地变成现实，虽然此时的 Sun 已经成为历史。这些网站开启了一个新的时代，正如有了淘宝你可以随时成为网络个体户一样，利用它们，你也可以很快的实现创业的想法。虽然，你得按它们的条条框框办事，但也省去了找主机、部署、备份、主机扩容等等一系列的麻烦事。有失必有得。

但人总是有好奇心的，即便是这样，你可能仍然会期望了解你所依托的站点的内部情况。为此，我们在本期特意邀请到了来自新浪的丛磊和百度的肖伟，让他们分别讲述自己的架构故事。

那么，准备好了吗？让我们现在开始这一次的站点架构之旅吧！

本期主编：胡键

目　录

[篇首语]

站点架构闲谈 1

[人物专访]

ROB PIKE谈GOOGLE GO：并发，TYPE SYSTEM，内存管理和GC 3

[热点新闻]

SILVERLIGHT应用整合 13

企业架构到底是什么？ 18

MAVEN实战（七）--常用MAVEN插件介绍（上） 20

TWILIO的云架构原则 24

阅读者(十一):简单之美 26

[特别专题]

SINA APP ENGINE数据存储服务架构 30

又拍网架构中的分库设计 39

我眼中的云端架构 49

TRUNK.LY CTO董洵谈架构 58

[推荐文章]

探索式测试的秘密 63

项目的故事 70

虚拟研讨会：NODE.JS生态系统之框架、库、最佳实践 76

JAVA深度历险（七）--JAVA反射与动态代理 87

书评和采访：EXTJS IN ACTION 92

[新品推荐]

图形数据库NEO4J更新软件许可方式并增强可用性 97

HUDSON以新版本编号 2.0.0 重新登场 97

QUARTZ 2.0 支持FLUENT CONFIGURATION API和工作调度的监控了 97

JAVAFX 2.0 发布近况和未来计划 98

WINDOWS INSTALLER XML（WIX）3.5 简化了COMPONENT AUTHORING..... 98

软件架构模拟器PALLADIO 3.2 版发布 98

VISUAL STUDIO提供对CPYTHON的支持 99

IRONPYTHON完整实现PYTHON 2.7 的功能 99

WSO2 引入新的开源项目：WSO2 MESSAGE BROKER..... 99

[推荐编辑]

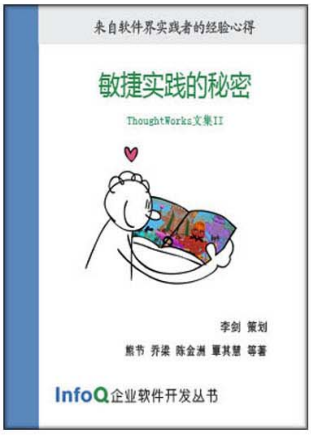
翻译团队编辑 吴宇 100

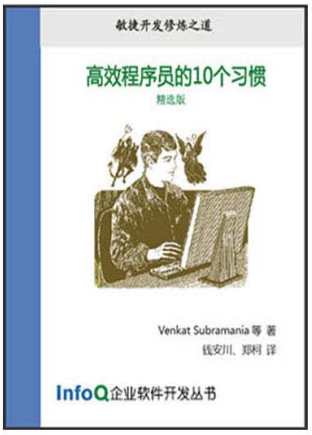
[封面植物]

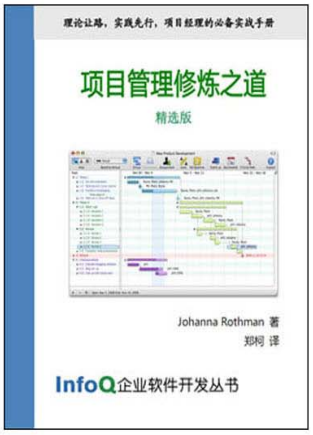
七子花 101

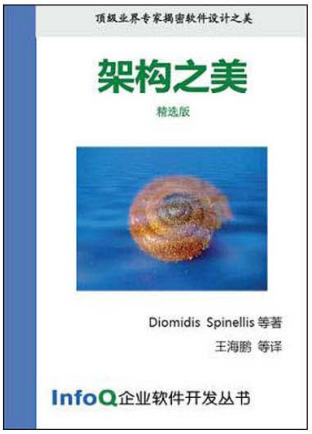
InfoQ企业软件开发丛书

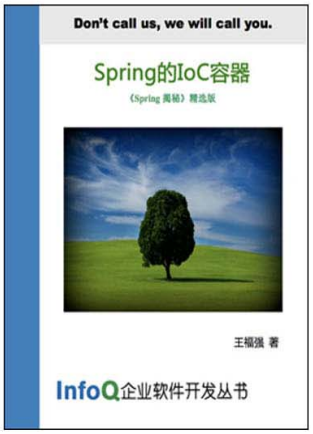
欢迎免费下载











商务合作：sales@cn.infoq.com　　读者反馈/内容提供：editors@cn.infoq.com

人物专访 | Interview

Rob Pike 谈 Google Go：并发，Type System，内存管理和 GC

Rob Pike 是 Google 的首席工程师。最近，他参与开发了 Go 语言。InfoQ 有幸在 GOTO 会议上关于 Go 采访了 Rob Pike。内容涉及：无类 OOP 编程、Go 的接口、采用 Goroutines 和 Channels 的并发特性，以及 Go 中帮助缩短 GC 间歇的一些特性。

Rob Pike 是 Google 的首席工程师。最近，他参与开发了 Go 语言。在此之前，Rob 在贝尔实验室的 CS 研究中心工作，主要从事计算机图形，语言，并发编程等方面的工作，同时他也是 Plan 9 及 Inferno OSes 的架构师。此外，他还与 Brian Kernighan 合著了《The Unix Programming Environment》及《The Practice of Programming》。

InfoQ：Rob，你创建了 Google Go 这门语言。什么是 Google Go？能简明扼要的介绍一下 Google Go 吗？

Rob：我还是讲讲为什么要创建这门语言吧，和你的问题稍有些不同。我在 Google 做了一个有关编程语言的系列讲座，在 Youtube 上有，谈及了我早期所写的一个语言，叫做 Newsqueak，那是八十年代的事，非常早。在做讲座期间，我开始思考为什么 Newsqueak 中的一些想法在我现在以 C++为主的工作环境中无法使用。而且在 Google 我们经常要构建非常大的程序，光构建就要花很多时间，对依赖的管理也有问题，由于链接了本来并不需要的东西，二进制程序包变得很大，链接时间很长，编译时间也很长，而且 C++的工作方式有点古老，其底层实际上 C，C++已经有三十年的历史了，而 C 则更是有四十年了。用现今的硬件做计算，有很多新东西需要考虑：多核机器、网络化、分布式系统、云计算等等。

InfoQ： Go 的主要特点是什么？有什么重要功能？

Rob：对于大多数人来说，他们对 Go 的第一印象是该语言将并发性作为语言原语，这对我们处理分布式计算和多核这类东西来说非常好、也非常重要。我猜许多人会认为 Go 是一门

简单无趣的语言，没有什么特别的东西，因为其构想看起来一目了然。但实际上不能用第一印象来判断 Go。很多用过 Go 的人会发现它是一门非常高产而且有表现力的语言，能够解决我们编写这门语言时期望其所能解决的所有问题。

Go 的编译过程很快，二进制程序包又比较小，它管理依赖的方式如同管理语言本身的东西一样。这里还有一个故事呢，但是在这里就不再展开讨论了，但是这门语言的并发性使其能够以非常简单的模式来处理非常复杂的操作及分布式计算环境。我想最重要的功能可能就是并发性了，后面我们可以谈谈该语言的类型系统，其与 C++、Java 这类传统面向对象类型系统的差异很大。

InfoQ：在我们继续话题之前，能否解释一下为什么 Go 编译器能达到那么快的编译速度呢？有什么法宝？

Rob：它之所以快，有两个原因。首先 Go 有两个编译器——两个单独的实现。一个是按照 Plan 9（http://plan9.bell-labs.com/wiki/plan9/1/） 风格新写的编译器，它有自己独特的工作方式，是个全新的编译器。另一个编译器叫做 GCC Go，它拥有 GCC 前端，这个编译器是 Ian Taylor 后来写的。所以 Go 有两个编译器，速度快是二者的共同特点，但是 Plan 9 风格编译器的速度是 GCC Go 的 5 倍，因为它从头到脚都是全新的，没有 GCC 后端，那些东西会花很多时间来产生真正的好代码。

GCC Go 编译器要产生更好的代码，所以速度慢些。不过真正重要的一点是 Go 编译器的依赖管理特性才是其编译速度快的真正原因。如果你去看一个 C 或 C++程序，便会发现其头文件描述了函数库、对象代码等等东西。语言本身并不强制检查依赖，每一次你都必须分析代码以便清楚你的函数是怎样的。如果你编译过程中想用另一个类的 C++程序，你必须先编译它所依赖的类和头文件等等等等。如果你所编译的 C++程序有许多类，并且内部相关，你可能会把同一个头文件编译数百次甚至上千次。当然，你可以用预编译头文件及其他技巧来回避之一问题。

但是语言本身并不能帮上你的忙，工具可能会让这一问题得到改善，可是最大的问题是并没有什么能保证你所编译的东西就是程序真正需要的东西。有可能你的程序包含了一个并不真正需要的头文件，但是你没办法知道，因为语言并没有强制检查。而 Go 有一个更加严格的依赖模型，它有一些叫做包（packages）的东西，你可以把它想象成 Java 类文件或着类似的东西，或者函数库什么的，虽然他们并不相同，但基本思路是一样的。关键问题是，如果这个东西依赖那个东西，而那个东西又依赖另外一个东西，比如 A 依赖于 B，B 又依赖于 C，那么你必须首先编译最内层的依赖：即，你先编译 C，然后编译 B，最后编译 A。

但是如果 A 依赖 B，但是 A 并不直接依赖于 C，而是存在依赖传递，那么该怎么办呢？这时所有 B 需要从 C 拿到的信息都会被放在 B 的对象代码里。这样，当我编译 A 的时候，我不需要再管 C 了。于是事情就非常简单了：在你编译程序时，你只需将类型信息沿着依赖关系树向上遍历即可，如果你到达树的顶端，则只需编译紧邻的依赖，而不用管其它层级的依赖了。如果你要做算术运算，你会发现在 Objective-C 或 C++或类似的语言里，虽然只包含了一个简单的头文件，但由于依赖传递的存在，你可能会编译数十万行程序。然而在 Go 中，你打开一个文件，里面或许只有 20 行，因为其中只描述了公共接口。

如果一个依赖链里只有三个文件，Go 的优势可能并不明显，但是如果你有成千上万个文件的时候，Go 的速度优势会成指数增长。我们相信，如果用 Go 的话，我们应该能够在数秒内就编译完数百万行代码。然而如果是等量的用 C++编写的程序，由于依赖管理问题，编译的开销会大得多，编译的时间将会长达若干分钟。因此，Go 速度快的根源主要归功于对依赖的管理。

InfoQ： 让我们开始聊聊 Go 里的类型系统吧。Go 里面有结构（struct）、有类型（type），那么 Go 里的类型是什么？

Rob：Go 里的类型与其它传统编程语言里的类型是类似的。Go 里的类型有整数、字符串、struct 数据结构、以及数组（array），我们称之为切片（slice），它们类似于 C 的数组，但更易于使用，更加固定一些。你可以声明本地类型并予以命名，然后按照通常的方式来使用。Go 和面向对象方式的不同之处在于，类型只是书写数据的一种方式，方法则是一个完全独立的概念。你可以把方法放在 struct 上，在 Go 里没有类的概念，取而代之的是结构，以及为此结构声明的一些方法。

结构不能与类混为一谈。但是你也可以把方法放在数组、整数、浮点数或字符串上，实际上任何类型都可以有方法。因此，这里方法的概念比 Java 的方法更加泛化，在 Java 里方法是类的一部分，仅此而已。例如，你的整数上可以有方法，听上去似乎没什么用，但是如果你想在一个叫做 Tuesday 的整数常量上附加上 to_string 方法来打印出漂亮的星期格式；或者，你想重新格式化字符串使其能够以不同的方式打印出自己，这时你就会意识到它的作用。为什么非要把所有方法或者其它好东西都塞进类里面呢，为什么不让它们提供更广泛的服务呢？

InfoQ： 那么这些方法只是在包内部可见喽？

Rob：非也，实际上是这样，Go 只允许你在包内为你所实现的类型定义方法。我不能引入你的类型然后直接把我的方法增加进去，但是我可以使用匿名属性（anonymous field）将其包

裹起来，方法可不是你想加到哪就加到哪的，你要定义类型，然后才能把方法放在上面。正因为如此，我们在包里提供了另一种封装——接口（interface），但是如果你不明白谁能为对象增加方法的严格界限，就很难理解接口。

InfoQ： 你的意思是，我可以给 int 增加方法，但是必须先使用 typedef 吗？

Rob：你要 typedef 一个整数类型，起个名字，如果你正在处理一星期中的七天，可以就叫它“Day”，你可以给你所声明的类型——Day 增加方法，但是你不能直接给 int 增加方法。因为整数类型不是你定义的，不在你的包里，它是引入的但并不在你的包中定义，这就意味着你不能给其增加方法。你不能给不在你包里定义的类型增加方法。

InfoQ： 你们借鉴了 Ruby 里开放类的思想，这很有意思。Ruby 的开放类实际上是可以修改类并增加新的方法，这是有破坏性的，但是你们的方法本质上是安全的，因为创建了新的东西。

Rob：它是安全可控的，而且很容易理解。最初我们觉得类型用起来可能不太方便，我们也希望像 Ruby 那样添加方法，但这又让接口比较难以理解。所以，我们只把方法取出来，而不是放进去，我们想不出有什么更好的办法，于是限制方法只能在本地类型上，不过这种思路确实很容易理解和使用。

InfoQ： 你还提到了 typedef，是叫 typedef 吧？

Rob：应该叫“type”，你所说的类型——Day 的定义方式是这样“type Day int”，这样你就有一个新类型了，你可以在其上增加方法、声明变量，但这个类型不同于 int，不像 C 那样，只是同一事物另起了个名字而已，在 Go 里实际上你创建了一个不同于 int 的新类型，叫做“Day”，它拥有 int 的结构特性，但却有自己的方法集。

InfoQ： Typedef 在 C 里是一种预处理指令吗？【编辑注/免责申明：C 语言里的 typedef 与预处理无关】

Rob：那实际上就是个别名，但在 Go 里不是别名，是新类型。

InfoQ： 我们从底层说起吧，在 Go 里最小的类型是什么？

Rob：最小的类型应该是布尔类型（bool）吧。bool、int 和 float，然后是 int32、float64 之类有尺寸的类型、字符串、复杂类型，可能有遗漏，但这就是基本类型集了。你可以由这些类型构建结构、数组、映射（map），映射在 Go 里是内建类型不是函数库。然后我想就该是接口了，到了接口，有趣的东西才真正开始。

InfoQ： 但是，int 这样的类型是值类型对吧。

Rob：Int 是值类型。在 Go 里，任何类型都是值类型，和 C 一样，所有东西都是按值调用，但是你也可以用指针。如果你想引用某样东西，可以获取其地址，这样你就有了一个指针。Go 也有指针但是比 C 指针有更多限制，Go 里的指针是安全的，因为他们是类型安全的，所以你没法欺骗编译器，而且也没有指针运算，因此，如果你有个指向某物的指针，你无法将其移到对象外，也无法欺骗编译器。

InfoQ： 它们类似 C++的引用吗？

Rob：是的，很像引用，但是你可以按照你预期的方式对它们进行写操作。而且你可以使用结构内部（如缓冲区）中间的某个地址，它和 Java 的引用不一样。在 Java 中，你必须在旁边分配一个缓冲区，这是额外的开销。在 Go 中，你实际上把该对象分配为结构的一部分，在同一内存块中，这对性能是非常重要的。

InfoQ： 它是结构内部一个复合对象。

Rob：是的，如果它是值而不是指针的话，是这样。当然你也可以把指针放在结构内部和外部，但是如果你有 struct A，而把 struct B 放在 struct A 里，那么 stuct B 就是一块内存，而不像 Java 那样，这也是 Java 性能问题的原因之一。

InfoQ： 你提到过接口比较有趣，那下面咱们就谈谈这一部分。

Rob：Go 里的接口真的非常、非常地简单。接口指明了两个不同事情：其一，它表明了类型的构思，接口类型是一个罗列了一组方法的类型，因此如果你要抽象一组方法来定义一个行为，那么就定义一个接口并声明这些方法。现在你就有了一个类型，我们就叫它接口类型吧，那么从现在起所有实现了接口中这些方法的类型——包括基本类型、结构、映射（map）或其它什么类型，都隐含符合该接口要求。其二，也是真正有意思的是，和大多数语言中的接口不同的是，Go 里面没有“implements”声明。

你无须说明“我的对象实现了这个接口”，只要你定义了接口中的那些方法，它就自动实现了该接口。有些人对此感到非常担忧，依我看他们想说的是：知道自己实现（Implement）了什么接口真的很重要。如果你真想确定自己实现了什么接口，还是有技巧可以做到这一点的。但是我们的想法与此截然不同，我们的想法是你不应该考虑实现什么接口，而是应该写下要做的东西，因为你不必事前就决定要实现哪个接口。可能后来你实际上实现了某个现在你尚不知晓的接口，因为该接口还未设计出来，但是现在你已经在实现它。

后来你可能发现两个原先未曾考虑过相关性的类具有了相关性——我又用了类这个词，我思

考 Java 太多了——两个 structs 都实现了一些非常有用的小子集中的相关方法，这时有办法能够操作这两个 structs 中的任意一个就显得非常有用了。这样你就可以声明一个接口，然后什么都不用管了，即使这些方法是在别人的代码中实现的也没问题，虽然你不能编辑这些代码。如果是 Java，这些代码必须要声明实现你的接口，在某种意义上，实现是单向的。然而在 Go 里，实现是双向的。对于接口实际上有不少漂亮而简单的例子。

我最爱用的一个真实例子就是"Reader"，Go 里有个包叫做 IO，IO 包里有个 Reader 接口，它只有一个方法，该方法是 read 方法的标准声明，比如从操作系统或文件中读取内容。这个接口可以被系统中任何做 read 系统调用的东西所实现。显然，文件、网络、缓存、解压器、解密机、管道，甚至任何想访问数据的东西，都可以给其数据提供一个 Reader 接口，然后想从这些资源中读取数据的任何程序都可以通过该接口达到目的。这有点像我们前面说过的 Plan 9，但是用不同的方式泛化的。

与之类似，Writer 也是比较好理解的另一个例子，Writer 由那些要做写操作的人来实现。那么在做格式化打印时，fpringf 的第一参数不是 file 了，而是 Writer。这样，fprintf 可以给任何实现了 write 方法的东西做 IO 格式化的工作。有很多很好的例子：比如 HTTP，如果你正在实现一个 HTTP 服务器，你仅须对 connection 做 fprintf，便可将数据传递到客户端，不需要任何花哨的操作。你可以通过压缩器来进行写操作，你可以通过我所提到的任何东西来进行写操作：压缩器、加密机、缓存、网络连接、管道、文件，你都可以通过 fprintf 直接操作，因为它们都实现了 write 方法，因此，隐含都隐含符合 writer 接口要求。

InfoQ： 某种程度上有点类似结构化类型系统（structural typing）。

Rob：不考虑它的行为的话，它是有点像结构化类型系统。不过它是完全抽象的，其意并不在拥有什么，而是能做什么。有了结构（struct）之后，就规定了其内存的样子，然后方法说明了结构的行为，再之后，接口则抽象了该结构及其它实现了相同方法的其他结构中的这些方法。这是一种鸭子类型系统（duck typing，一种动态类型系统），而不是结构化类型系统。

InfoQ： 你提到过类，但 Go 没有类，对吧。

Rob：Go 没有类。

InfoQ： 但是没有类怎么去写代码？

Rob：带方法的结构（stuct）很像是类。比较有意思的不同之处是，Go 没有子类型继承，你必须学习 Go 的另类写法，Go 有更强大、更有表现力的东西。不过 Java 程序员和 C++程序员刚开始使用 Go 的时候会感到意外，因为他们实际上在用 Go 去编写 Java 程序或 C++程序，

这样的代码工作得并不好，你可以这样做，但这样就略显笨拙了。但是如果你退一步，对自己说“我该怎样用 Go 去编写这些东西呢？”，你会发现模式其实是不同的，用 Go 你可以用更短的程序来表达类似的想法，因为你不需要在所有子类里重复实现行为。这是个非常不同的环境，比你第一眼看上去的还要不同。

InfoQ： 如果我有一些行为要实现，而且想放在多个 structs 里，怎么去共享这些行为？

Rob：有一个叫做匿名域的概念，也就是所谓的嵌入。其工作方式是这样：如果你有一个结构（struct），而又有一些其它东西实现了你想要的行为，你可以把这些东西嵌入到你的结构（struct）里，这样，这个结构（struct）不仅仅可以获得被嵌入者的数据还可以获得它的方法。如果你有一些公共行为，比如某些类型里都有一个 name 方法，在 Java 里的话你会认为这是一组子类（继承来的方法），在 Go 里，你只需拿到一个拥有 name 方法的类型，放在所有你要实现这个方法的结构里，它们就会自动获得 name 方法，而不用在每个结构里都去写这个方法。这是个很简单的例子，但有不少有趣的结构化的东西使用到了嵌入。

而且，你还可以把多个东西嵌入到一个单一结构中，你可以把它想象成多重继承，不过这会让人更加迷惑，实际在 Go 里它是很简单的，它只是一个集合，你可以放任何东西在里面，基本上联合了所有的方法，对每个方法集合，你只需写一行代码就可以拥有其所有行为。

InfoQ： 如果有多重继承命名冲突的问题该怎么办？

Rob：命名冲突实际上并没什么，Go 是静态处理这一问题的。其规则是，如果有多层嵌入，则最高层优先；如果同一层有两个相同的名字或相同的方法，Go 会给出一个简单的静态错误。你不用自己检查，只需留意这个错误即可。命名冲突是静态检查的，而且规则非常简单，在实践中命名冲突发生的也并不多。

InfoQ： 因为系统中没有根对象或根类，如果我想得到一个拥有不同类型的结构的列表，应该怎么办？

Rob：接口一个有意思的地方是他们只是集合，方法的集合，那么就会有空集合，没有任何方法的接口，我们称之为空接口。系统中任何东西都符合空接口的要求。空接口有点类似于 Java 的 Object，不同之处在于，int、float 和 string 也符合空接口，Go 并不需要一个实际的类，因为 Go 里没有类的概念，所有东西都是统一的，这有点像 void*，只不过 void*是针对指针而不是值。

但是一个空接口值可以代表系统中的任何东西，非常具有普遍性。所以，如果创建一个空接口数组，实际上你就有了一个多态性容器，如果你想再把它拿出来，Go 里面有类型开关，你可以在解包的时候询问里面的类型，因此可以安全的进行解包操作。

InfoQ： Go 里有叫做 Goroutines 的东西，它们和 coroutines 有什么区别？不一样么？

Rob：Coroutines 和 Goroutines 是不同的，它们的名字反应了这一点。我们给它起了个新名，因为有太多术语了，进程（processes）、线程（threads）、轻量级线程、弦（chords），这些东西有数不清的名字，而 Goroutines 也并不新鲜，同样的概念在其它系统里已经都有了。但是这个概念和前面那些名字有很大不同，我希望我们自己起名字来命名它们。Goroutine 背后的含义是：它是一个 coroutine，但是它在阻塞之后会转移到其它 coroutine，同一线程上的其它 coroutines 也会转移，因此它们不会阻塞。

因此，从根本上讲 Goroutines 是 coroutines 的一个分支，可在足够多的操作线程上获得多路特性，不会有 Goroutines 会被其他 coroutine 阻塞。如果它们只是协作的话，只需一个线程即可。但是如果有很多 IO 操作的话，就会有许多操作系统动作，也就会有许多许多线程。但是 Goroutines 还是非常廉价的，它们可以有数十万之众，总体运行良好并只占用合理数量的内存，它们创建起来很廉价并有垃圾回收功能，一切都非常简单。

InfoQ： 你提到你们使用了 m:n 线程模型，即 m 个 coroutines 映射到 n 个线程上？

Rob：对的，但是 coroutines 的数量和线程的数量是按照程序所做工作动态决定的。

InfoQ： Goroutines 有用于通信的通道吗？

Rob：是的，一旦有两个独立执行的功能，如果 Goroutine 们要相互协作它们就需要相互对话。所以就有了通道这个概念，它实际上是一个类型消息队列，你可以用它来发送值，如果你在 Goroutine 中持有通道的一端，那么你可以发送类型值给另外一端，那一端则会得到想要的东西。通道有同步和异步之分，我们尽可能使用同步通道，因为同步通道的构思非常好，你可以同时进行同步和通信，所有东西运行起来都步调一致。

但是有时由于效率原因或调度原因，对消息进行缓存也是有意义的。你可以向通道发送整型消息、字符串、结构、指向结构的指针等任何东西，非常有意思的事，你可以在通道上发送另一个通道。这样，我就能够把与他人的通信发送给你，这是非常有意思的概念。

InfoQ： 你提到你们有缓存的同步通道和异步通道。

Rob：不对，同步是没有缓存的；异步和缓存是一个意思，因为有了缓存，我才能把值放在缓存的空间里进行保存。但是如果没有缓存，我必须等着别人把值拿走，因此无缓存和同步是一个意思。

InfoQ： 每个 Goroutine 就像是一个小的线程，可以这么给读者解释吧。

Rob：对，但是轻量级的。

InfoQ： 它们是轻量级的。但是每个线程同样都预分配栈空间，因而它们非常耗费资，Goroutines 是怎么处理的呢？

Rob：没错，Goroutines 在被创建的时候，只有非常小的一个栈——4K，可能有点小吧，这个栈是在堆中的，当然，你知道如果在 C 语言里有这么一个小栈会发生什么，当你调用函数或分配数组之类的东西时，程序会马上溢出。在 Go 里则不会发生这样的事情，每个函数的开头都会有若干指令以检查栈指针是否达到其界限，如果到达界限，它会链接到其它块上，这种连接的栈叫做分段栈，如果你使用了比刚开始启动时更多的栈，你就有了这种栈块链接串，我们称之为分段栈。

由于只有若干指令，这种机制非常廉价。当然，你可以分配多个栈块，但是 Go 编译器更倾向于将大的东西移到堆上，因此实际上典型的用法是，你必须在达到 4K 边界之前调用几个方法，虽然这并不经常发生。但是有一点很重要：它们创建起来很廉价，因为仅有一次内存分配，而且分配的内存非常小，在创建一个新的 Goroutine 时你不用指明栈的尺寸，这是很好的一种抽象，你根本不用担心栈的大小问题。之后，栈会随需求增长或缩小，你不用担心递归会有问题，你也不用担心大的缓存或任何对程序员完全不可见的东西，一切由 Go 语言来打理，这是一门语言的整体构思。

InfoQ： 我们再来谈谈自动化方面的东西，最初你们是将 Go 语言作为系统级语言来推广的，一个有趣的选择是使用了垃圾回收器，但是它速度并不快或者说有垃圾回收间歇问题，如果用它写一个操作系统的话，这是非常烦人的。你们是怎么看这一问题的？

Rob：我认为这是个非常难的问题，我们也还没有解决它，我们的垃圾回收器可以工作，但是有一些延迟问题，垃圾回收器可能会停顿，但是我们的看法是，我们相信尽管这是一个研究课题，虽还没解决但是我们正在努力。对于现今的并行机，通过把机器内核的一些碎片专门分给作为后台任务的垃圾回收来进行并行回收是可行的。在这一领域有很多工作要做，也取得了不少成功，但这是个很微妙的问题，我不认为而我们会把延迟降为 0，但是我相信我们可以让延迟尽可能低，这样对于绝大多数系统软件来讲它不再是个问题。我不保证每个程序都不会有显著延迟，但是我想我们可以获得成功，而且这是 Go 语言中一个比较活跃的领域。

InfoQ： 有没有方法能够避免直面垃圾回收器，比如用一些大容量缓存，我们可以把数据扔进去。

Rob：Go 可以让你深入到内存布局，你可以分配自己的空间，如果你想的话可以自己做内存

管理。虽然没有 alloc 和 free 方法，但是你可以声明一个缓存把东西放进去，这个技巧可用来避免产生不必要的垃圾。就像在 C 语言一样，在 C 里，如果你老是 malloc 和 free，代价很大。因此，你分配一个对象数组并把它们链接在一起，形成一个链表，管理你自己的空间，而且还不用 malloc 和 free，那么速度会很快。你可以做与 Go 所做相同的事情，因为 Go 赋予你与底层事物安全打交道的能力，因此不用欺骗类型系统来达到目的，你实际上可以自己来做。

前面我表达了这样的观点，在 Java 里，无论何时你在结构里嵌入其它东西，都是通过指针来实现的，但在 Go 里你可以把它放在一个单一结构中。因此如果你有一些需要若干缓存的数据结构，你可以把缓存放在结构的内存里，这不仅意味着高效（因为你不用间接得到缓存），而且还意味着单一结构可以在一步之内进行内存分配与垃圾回收。这样开销就会减少。因此，如果你考虑一下垃圾回收的实际情况，当你正在设计性能要求不高的东西时，你不应该总是考虑这个问题。但如果是高性能要求的，考虑到内存布局，尽管 Go 是具有真正垃圾回收特性的语言，它还是给了你工具，让你自己来控制有多少内存和产生了的垃圾。我想这是很多人容易忽略的。

InfoQ： 最后一个问题：Go 是系统级语言还是应用级语言？

Rob：我们是把他设计为一种系统级语言，因为我们在 Google 所做的工作是系统级的，对吧？Web 服务器和数据库系统、以及存储系统等，这些都是系统。但不是操作系统，我不知道 Go 是否能成为一个好的操作系统语言，但是也不能说它不会成为这样的语言。有趣的是由于我们设计语言时所采用的方法，Go 最终成为了一个非常好的通用语言，这有点出乎我们意料。我想大多数用户并没有实际从系统观点来考虑过它，尽管很多人做过一点 Web 服务器或类似东西。

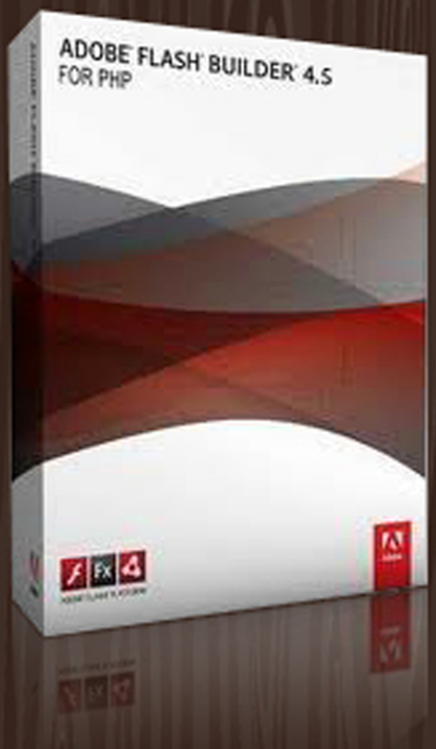
Go 用来做很多应用类的东西也非常不错，它将会有更好的函数库，越来越多的工具以及一些 Go 更有用的东西，Go 是一个非常好的通用语言，它是我用过的最高产的语言。

原文链接：http://www.infoq.com/cn/interviews/pike-google-go-cn

相关内容：

- Google试验新语言--Go、Google Go：初级读本、内存屏障与JVM并发
- 伏威谈淘宝网的高并发处理与压力测试
- 剖析java.util.concurrent锁

Adobe



Adobe Flash Builder 4.5 高级版

隆重发布

更强的功能
更多的设备
更少的编程

免费下载试用版

Adobe官方下载（本地服务器）

- Flash Builder 4.5高级版试用版（Windows，581.2M，点击免费高速下载）
- Flash Builder 4.5高级版试用版（Mac，679.8M，点击免费高速下载）
- Flash Builder 4.5 for PHP高级版试用版（Windows，956.5M，点击免费高速下载）
- Flash Builder 4.5 for PHP高级版试用版免费（Mac，911.3M，点击免费高速下载）

Adobe官方下载（国外服务器）

- Flash Builder 4.5高级版试用版（Windows和Mac版本，免费高速下载）
- Flash Builder 4.5 for PHP高级版试用版（Windows和Mac版本，免费高速下载）

InfoQ

更多下载

热点新闻 | News

Silverlight 应用整合

作者 李玮

Silverlight 与企业信息化

在企业应用系统中，企业客户不在满足于呆板的 Portal、树形结构、数据堆积的表格和满屏的文字描述。企业客户越来越倡导用户体验，Silverlight 迎合了这种需求，实现了炫目的体验和丰富交互的应用，更重要的是 Silverlight 更适合企业应用系统的开发，依靠.Net 平台，使用 C#高级语言代替 JavaScript 脚本语言，实现了强大的客户端计算能力、支持多线程，丰富的控件、特效与动画，更可控的浏览器适应性，更安全的沙箱模式，客户端嵌入式数据库等。

现在企业应用开发有两大阵营，一个是 Java，另一个是.NET 平台，难道 Silverlight 只能服务于后者吗？答案：不是。Silverlight 定位于，可以快速部署到浏览器中的富互联网应用。Silverlight 将要覆盖的领域是“客户端应用程序”和“媒体体验”。Silverlight 是为连接到主机的网页内容而设计的，其特点是不但用户可深入地参与到交互中，而且应用还可在任何浏览器中运行。

图 1 Silverlight 各组件之间的关系

“Silverlight 应用”运行于浏览器中的“XAP 文件”本身需要使用 Microsoft Visual Studio 工具开发，就像“Flash 应用”运行于浏览器中的“SWF 文件”需要使用 AdobeFlex 工具开发一样。“Silverlight 应用”可以和“Flash 应用”一样嵌入到已有的网页中运行，“应用本身”需要数据时再去向后端服务器通讯，例如：“Flash 应用”经常使用的从服务器下载“XML 数据文件”后再显示于应用中的通讯方案。

Silverlight 应用通信原型

Silverlight 内嵌在 HTML 网页中，自然就包含在 web 浏览器的沙箱内。非常重要的一点是，在 Silverlight 运行的客户端机器并不需要安装.NET。运行 Silverlight 应用所需的必要资源全都包含在 Silverlight 浏览器插件中，所有互操作场景都建立在 Silverlight 之上。Silverlight 应用的最终生的 XAP 文件，在被用户下载到本地后所有通信任务都是由前端的 Silverlight 应用通过网络访问后端 Server 的数据，以回传绑定显示在 UI 之上。Silverlight 支持广泛的网络能力如：REST、SOAP、POX，RSS 及标准 HTTP 服务等网络技术的调用，这部分对前端的 Silverlight 特别重要。下面所示是通信原型：

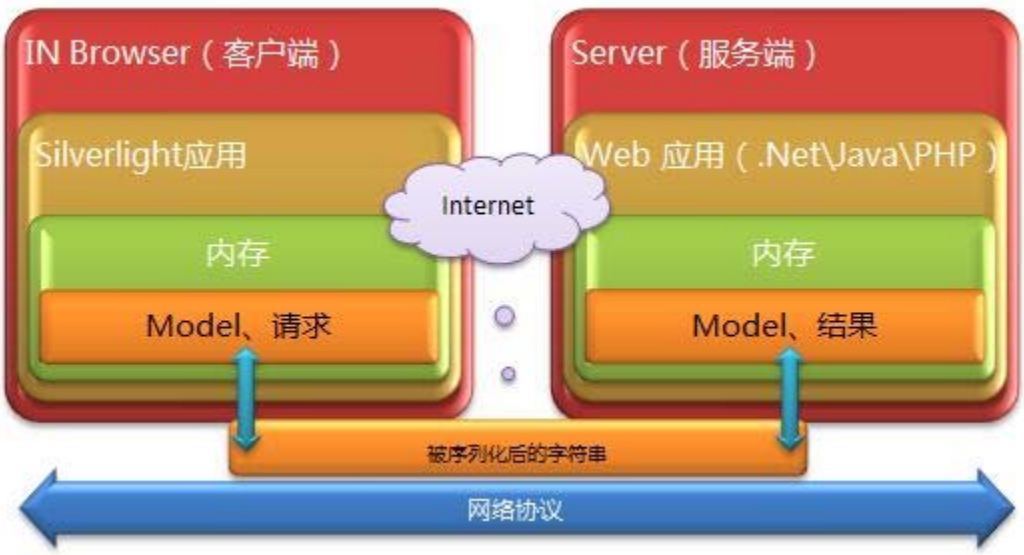


图 2 Silverlight 通信原型

前端的 Silverlight 通过网络访问后端服务器的数据，并以在 UI 之上绑定显示回传数据的过程分为以下步骤：

1. 内嵌在 HTML 网页中 Silverlight 应用发起请求或回传数据。
2. 客户端内存中的“请求”或“数据”被序列化为可传输的字符串，并发送到服务器。
3. 服务器接收并反序列化传输的字符串为请求或数据到服务端内存中处理。

Silverlight 通信技术

Silverlight 提供了丰富的与服务器端通信能力，目前 Silverlight 支持 TCP、UDP、Socket 等网络协议，以及 HttpHandler、WebServices 等多种应用层协议，还可以通过 WCF RIA Service 数据“透明”访问方式（通过向客户端公开数据访问层，屏蔽了通信层手工编码）快速实现实体通信。下面所示为可用于 Silverlight 的通信技术：

图 3 用于 Silverlight 的通信技术

Silverlight 运行时支持若干将网络应用程序与远程主机进行连接的主要方法：

序号	命名空间	类	用途
1	System.Net	WebClient HTTP	这些类使用 HTTP 或 HTTPS 协议来进行网络通信
2	System.Net.Sockets	套接字	这些类提供一个低级别的套接字接口，该接口可用于更常规的使用单播的网络通信
3	System.Net.Sockets	UdpAnySourceMulticastClient UdpSingleSourceMulticastClient	这些类为 UDP 多路广播客户端提供支持

Silverlight 运行时直接支持的连接方法，都可用于非.Net 平台的技术平台如：JAVA 和 PHP 等。

数据或命令要正常交换要具备两个必要条件：渠道和通用语义。这里的渠道就是与远程主机的连接方法，这里的通用语义指的就是：“序列化与反序列化”。当两个进程在进行远程通信

时，彼此可以发送各种类型的数据。无论是何种类型的数据，都会以可保持或传输的格式在网络上传送。把对象转换为可保持或传输的格式的过程称为对象的序列化。把可保持或传输的格式恢复为对象的过程称为对象的反序列化。

序号	类型	通途
1	二进制序列化	保持类型保真度，这对于在应用程序的不同调用之间保留对象的状态很有用例如，通过将对象序列化到剪贴板，可在不同的应用程序之间共享对象。
2	XML 序列化	不保持类型保真度，当您要提供或使用数据而不限制使用该数据的应用程序时。
3	JSON 序列化	不保持类型保真度，一种通用的数据格式，有着存储体积小的特点。

Silverlight 应用整合

在原有应用集成方面，如果服务器端使用.Net 平台，那么 Silverlight 客户端程序可以重用原有通信方式，而不必对服务器端程序有较大改动，如果服务器端使用 Java 平台，Silverlight 也提供标准 WebServices、Http、Socket 多种交互方式进行通信，服务器端只要进行简单的服务封装就可以实现与 Silverlight 客户端的数据通信。因此，Silverlight 基于.Net 提供了快速与服务器端程序集成的能力。事实上，在笔者所开发的项目中，已经实现了 Silverlight 客户端与.Net、J2EE（通过 REST 接口）、Tuxedo（通过 SALT 接口）的数据通信。

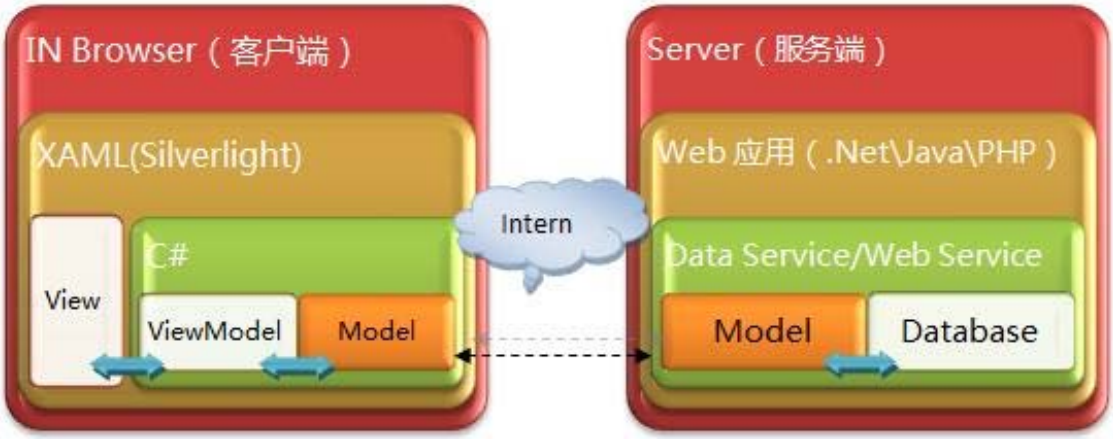


图 4 原有应用集成

因此，Silverlight 基于.Net 提供了快速与服务器端语言集成的能力。事实上，在笔者所开发的项目中，已经实现了 Silverlight 客户端与.Net、J2EE（通过 REST 接口）、Tuxedo（通过 SALT 接口）的数据连通性。

Silverlight 以前对于支持网络的插件的设计将连接性限制到主机或源站点。这意味着，Web 应用程序只能向后与其部署服务器通信，因此，网络应用程序可以连接到从中下载它们的主机。Silverlight 2 和更高版本包括对跨域连接的支持，这使得应用程序可以访问并非源站点的其他位置中的资源。这是一个重要的功能，它使 Silverlight 应用程序可以使用 Web 上的现有服务。

结束语

企业客户越来越认识到 RIA 技术对于企业应用的重要性，而 Silverlight 就是实现企业应用 RIA 重要的选择之一。Silverlight 定位于“客户端应用程序”，无论是互联网应用中，还是企业应用中，都可以为使用者带来的愉悦感和价值感。在 Silverlight 的数据与通信的中，除了通信渠道与序列化的问题外，最值得注意的两点：异步模式与同步模式的区别、不同开发平台的数据类型匹配问题。

关于作者

李玮，东南融通公司架构师，负责 Silverlight 应用程序架构，拥有超过 9 年的项目实施及管理经验，带领过政府、医药、能源、保险、互联网等多个领域的项目，并一直应用.NET 技术进行研发。

原文链接：http://www.infoq.com/cn/news/2011/04/silverlight-app-integration

相关内容：

- Silverlight CoreCLR结构浅析
- SOA与信息风险管理
- 从Silverlight访问Windows 7 的特性
- Silverlight 4 更新：改进了网络延迟与内存使用
- Silverlight的多线程能力（上）

热点新闻 | News

企业架构到底是什么？

作者 Boris Lublinsky 译者 马国耀

Jason Bloomberg最近在 博客 中问道："为什么没有人做企业架构（Enterprise Architecture）呢？"他说：

> 解决方案架构师应该在实施解决方案之前完成解决方案的架构设计。Java 架构师和.NET 架构师做得事情应该先于编程人员。你不能先实施架构再设计架构， 只能先设计再实施……可是，企业架构（Enterprise Architecture）却往往从现有企业开始……当今企业架构师的角色主要面对当前企业，**修补**其中的问题。好吧，也许不全是这样，但却要做某种程度的改进。将企业架构从当前的"糟糕的"状态扭转到"完美的"状态，在这一状态下事情会变得更好。

Bloomberg 认为，虽然解决现有企业的问题既重要又高尚，但是这并非企业架构的工作：

> 架构不是解决问题，而是为设计活动建立一套最佳实践。

所以，在他看来，没有人在真正地做企业做架构。企业在不断成长，

> 每个企业家都知道这个基本道理。当企业首次坐下来，为新的商业投资制定方案时，如果组织(organization)大到可堪称企业（enterprise），他们也许永远也不敢对它做全面的架构设计，因为这里面有太多未知的东西。相反，他们却喜欢建立一个不断成长的框架。播散种子，为之浇灌、除草及施肥。如果运气好的话，沿着这条路走下去，也许能收获一个不错的、健康的、不断发展的企业架构。但是最终的架构看上去可能与最初设想的样子相去甚远。

Bloomberg 继续说到，这与企业架构（Enterprise Architecture）的概念有很大差异，企业架构要定义并建立一组最佳实践，通过它们去实现企业预期的最终目标。问题是：

> 发展企业……意味着它会像任何生物体的生长一样，没有确定的最终状态。一粒橡果最终将会长成橡树是确定的，但是这棵橡树到底长成什么样子，确实无法计划的。相反，橡果的 DNA 决定了会长出橡树这一基本属性，但是其他的东西就取决于后天的变化了。这类变化确定了复杂系统的特征：系统具有各种变化的属性，但这些属性综合起来却不同于任何部分的属性。就像生物界的机体依赖于变化一样，企业的发展同样依赖它。

Bloomberg 认为，改变企业架构的目标是没有意义的，相反应该引入一些新原则：

> 也许，应该为架构的变化确定最佳实践了。毕竟，如果我们可以对传统系统做架构，为何不能对复杂系统做呢？……我们到底能否找到实际做企业架构的方法？毕竟，企业架构需要的是复杂的、系统的方法。最后，还得看“能不能那样做企业架构（Enterprise Architecture）”。

JP Morgenthal在这篇帖子的评论中 说道 ，问题不在于企业架构的原则，而在于企业架构（Enterprise Architecture）这个词本身：

> ……企业架构这个词如果换成**多维度架构（multi-dimension architecture）**可能更好。后者更好地抓住了该活动的 本质，而且没有将它限定在某个特定的范围内——范围视任务的大小而定。我一贯认为业务与多维度架构保持着紧密联系。人们设计的解决方案包括业务流程、工作 流、应用、用户体验、网络连通、灾备/恢复等；但是，思考系统的任何一个部分可能对其他部分造成的影响时，还有哪些东西呢？对我而言，这才是最初创造企业 架构（Enterprise Architecture）这个差劲词汇的本意所在。

我们是可以评论企业架构这个词是好或是坏，但是，而现在当人们已经接受了这个叫法的时候去改它，显然不是一个好建议。其结果只能是带来更多的迷惑和论战。根据 IEEE标准1471-2000 对于软件密集型的系统架构的描述，IEEE建议 ：

> 架构是对系统、系统的内部组件、组件之间的关系、与外部环境间的关系、指导其设计和发展的原则等方面的基本组织。

此定义丝毫没有谈到最终状态——它所关心的是人们改进和发展系统时所遵循的原则，这与 Bloomberg 和 Morgenthal 所提出的定义非常相近。不过，根据该定义，为了使企业符合合适的架构原则，而对他尽心**修补**和**改进**即是**架构**。

原文链接：http://www.infoq.com/cn/news/2011/04/EnterpriseArchitecture

相关内容：人人网移动开发架构、监控和虚拟化技术在"去哪儿"中的应用、QCon北京 2011大会精彩回顾和总结、虚拟面向服务网格的出现、构建企业私有云架构

热点新闻 | News

Maven 实战（七）--常用 Maven 插件介绍（上）

作者 许晓斌

我们都知道 Maven 本质上是一个插件框架，它的核心并不执行任何具体的构建任务，所有这些任务都交给插件来完成，例如编译源代码是由 maven-compiler-plugin 完成的。进一步说，每个任务对应了一个插件目标（goal），每个插件会有一个或者多个目标，例如 maven-compiler-plugin 的 compile 目标用来编译位于 src/main/java/目录下的主源码，testCompile 目标用来编译位于 src/test/java/目录下的测试源码。

用户可以通过两种方式调用 Maven 插件目标。第一种方式是将插件目标与生命周期阶段（lifecycle phase）绑定，这样用户在命令行只是输入生命周期阶段而已，例如 Maven 默认将 maven-compiler-plugin 的 compile 目标与 compile 生命周期阶段绑定，因此命令 **mvn compile** 实际上是先定位到 compile 这一生命周期阶段，然后再根据绑定关系调用 maven-compiler-plugin 的 compile 目标。第二种方式是直接在命令行指定要执行的插件目标，例如 **mvn archetype:generate** 就表示调用 maven-archetype-plugin 的 generate 目标，这种带冒号的调用方式与生命周期无关。

认识上述Maven插件的基本概念能帮助你理解Maven的工作机制，不过要想更高效率地使用Maven，了解一些常用的插件还是很有必要的，这可以帮助 你避免一不小心重新发明轮子。多年来Maven社区积累了大量的经验，并随之形成了一个成熟的插件生态圈。Maven官方有两个插件列表，第一个列表的 GroupId为org.apache.maven.plugins，这里的插件最为成熟，具体地址为：http://maven.apache.org/plugins/index.html。第二个列表的 GroupId 为 org.codehaus.mojo，这里的插件没有那么核心，但也有不少十分有用，其地址为：http://mojo.codehaus.org/plugins.html。

接下来笔者根据自己的经验介绍一些最常用的 Maven 插件，在不同的环境下它们各自都有其出色的表现，熟练地使用它们能让你的日常构建工作事半功倍。

maven-antrun-plugin

http://maven.apache.org/plugins/maven-antrun-plugin/

maven-antrun-plugin 能让用户在 Maven 项目中运行 Ant 任务。用户可以直接在该插件的配置以 Ant 的方式编写 Target，然后交给该插件的 run 目标去执行。在一些由 Ant 往 Maven 迁移的项目中，该插件尤其有用。此外当你发现需要编写一些自定义程度很高的任务，同时又觉得 Maven 不够灵活时，也可以以 Ant 的方式实现之。maven-antrun-plugin 的 run 目标通常与生命周期绑定运行。

maven-archetype-plugin

http://maven.apache.org/archetype/maven-archetype-plugin/

Archtype 指项目的骨架，Maven 初学者最开始执行的 Maven 命令可能就是 **mvn archetype:generate**，这实际上就是让 maven-archetype-plugin 生成一个很简单的项目骨架，帮助开发者快速上手。可能也有人看到一些文档写了 **mvn archetype:create**，但实际上 create 目标已经被弃用了，取而代之的是 generate 目标，该目标使用交互式的方式提示用户输入必要的信息以创建项目，体验更好。maven-archetype-plugin 还有一些其他目标帮助用户自己定义项目原型，例如你由一个产品需要交付给很多客户进行二次开发，你就可以为他们提供一个 Archtype，帮助他们快速上手。

maven-assembly-plugin

http://maven.apache.org/plugins/maven-assembly-plugin/

maven-assembly-plugin 的用途是制作项目分发包，该分发包可能包含了项目的可执行文件、源代码、readme、平台脚本等等。maven-assembly-plugin 支持各种主流的格式如 zip、tar.gz、jar 和 war 等，具体打包哪些文件是高度可控的，例如用户可以按文件级别的粒度、文件集级别的粒度、模块级别的粒度、以及依赖级别的粒度控制打包，此外，包含和排除配置也是支持的。maven-assembly-plugin 要求用户使用一个名为 assembly.xml 的元数据文件来表述打包，它的 single 目标可以直接在命令行调用，也可以被绑定至生命周期。

maven-dependency-plugin

http://maven.apache.org/plugins/maven-dependency-plugin/

maven-dependency-plugin 最大的用途是帮助分析项目依赖，**dependency:list** 能够列出项目最终解析到的依赖列表，**dependency:tree** 能进一步的描绘项目依赖树，**dependency:analyze** 可以告诉你项目依赖潜在的问题，如果你有直接使用到的却未声明的依赖，该目标就会发出警告。maven-dependency-plugin 还有很多目标帮助你操作依赖文件，例如

dependency:copy-dependencies 能将项目依赖从本地 Maven 仓库复制到某个特定的文件夹下面。

maven-enforcer-plugin

http://maven.apache.org/plugins/maven-enforcer-plugin/

在一个稍大一点的组织或团队中，你无法保证所有成员都熟悉 Maven，那他们做一些比较愚蠢的事情就会变得很正常，例如给项目引入了外部的 SNAPSHOT 依赖而导致构建不稳定，使用了一个与大家不一致的 Maven 版本而经常抱怨构建出现诡异问题。maven-enforcer-plugin 能够帮助你避免之类问题，它允许你创建一系列规则强制大家遵守，包括设定 Java 版本、设定 Maven 版本、禁止某些依赖、禁止 SNAPSHOT 依赖。只要在一个父 POM 配置规则，然后让大家继承，当规则遭到破坏的时候，Maven 就会报错。除了标准的规则之外，你还可以扩展该插件，编写自己的规则。maven-enforcer-plugin 的 enforce 目标负责检查规则，它默认绑定到生命周期的 validate 阶段。

maven-help-plugin

http://maven.apache.org/plugins/maven-help-plugin/

maven-help-plugin 是一个小巧的辅助工具，最简单的 **help:system** 可以打印所有可用的环境变量和 Java 系统属性。**help:effective-pom** 和 **help:effective-settings** 最为有用，它们分别打印项目的有效 POM 和有效 settings，有效 POM 是指合并了所有父 POM（包括 Super POM）后的 XML，当你不确定 POM 的某些信息从何而来时，就可以查看有效 POM。有效 settings 同理，特别是当你发现自己配置的 settings.xml 没有生效时，就可以用 **help:effective-settings** 来验证。此外，maven-help-plugin 的 describe 目标可以帮助你描述任何一个 Maven 插件的信息，还有 all-profiles 目标和 active-profiles 目标帮助查看项目的 Profile。

maven-release-plugin

http://maven.apache.org/plugins/maven-release-plugin/

maven-release-plugin 的用途是帮助自动化项目版本发布，它依赖于 POM 中的 SCM 信息。**release:prepare** 用来准备版本发布，具体的工作包括检查是否有未提交代码、检查是否有 SNAPSHOT 依赖、升级项目的 SNAPSHOT 版本至 RELEASE 版本、为项目打标签等等。**release:perform** 则是签出标签中的 RELEASE 源码，构建并发布。版本发布是非常琐碎的工作，它涉及了各种检查，而且由于该工作仅仅是偶尔需要，因此手动操作很容易遗漏一些细节，

maven-release-plugin 让该工作变得非常快速简便，不易出错。maven-release-plugin 的各种目标通常直接在命令行调用，因为版本发布显然不是日常构建生命周期的一部分。

关于作者

许晓斌（Juven Xu），国内社区公认的Maven技术专家、Maven中文用户组创始人、Maven技术的先驱和积极推动者，著有《Maven实战》一书。对Maven有深刻的认识，实战经验丰富，不仅撰写了大量关于Maven的技术文章，而且还翻译了开源书籍《Maven权威指南》，对Maven技术在国内的普及和发展做出了很大的贡献。就职于Maven之父的公司，负责维护Maven中央仓库，是Maven仓库管理器Nexus（著名开源软件）的核心开发者之一，曾多次受邀到淘宝等大型企业开展Maven方面的培训。此外，他还是开源技术的积极倡导者和推动者，擅长Java开发和敏捷开发实践。他的个人网站是：http://www.juvenxu.com。

原文链接：http://www.infoq.com/cn/news/2011/04/xxb-maven-7-plugin

相关内容：

- 创建并扩展Apache Wicket Web应用
- 2010 年InfoQ中文站Java社区回顾：让Java飞一会
- Maven实战
- Maven实战（六）--Gradle，构建工具的未来？
- Maven实战（五）--自动化Web应用集成测试

热点新闻 | News

Twilio 的云架构原则

作者 Jean-Jacques Dubray 译者 丁雪丰

正当很多知名网站都在抱怨 自己受到了AWS问题的影响 时， Twilio的API和服务 并未受到影响 尽管他们也严重依赖于AWS来培育并扩展自己的 云电话平台(cloud telephony platform)。对于Evan Cooke（ Twilio的合伙创始人与CTO ）而言，这不仅展现了云服务在开启当代互联网生态环境方面的惊人成功，还展现了 坚实可靠的分布式架构设计 在构建云服务时的重要性。

> 当我们在 Amazon Web Services 上培育并扩展 Twilio 的时候，我们遵循了一系列的架构设计原则，以便能将底层基础设施中偶发但不可避免的问题所带来的影响降到最小。JQM Alpha 3 现在已经精简到 17K，其中还有相关的 CSS 文件。

- 故障单元是单台主机

> 构建由单台主机构成的简单服务，而非多依赖的主机构成的服务，可以创建复制服务实例来抵御主机故障。

- 短时间超时与快速重试

> 发生故障时，让软件快速识别失败并重试请求。每个服务都运行多个冗余拷贝，短时间内超时，然后绕过失败或不可访问的服务进行重试。

- 幂等的服务接口

> 如果所依赖服务的 API 是幂等的，那就意味着可以安全地对失败请求进行重试。

- 小的无状态服务

> 将业务逻辑分散到小的无状态服务中，这些服务可以被放到简单的同构服务池中。

- 宽松的一致性要求

> 在不需要严格一致性时，为要读取的数据做复制和冗余。

根据 故障的详细说明 ，Evan 还解释了为什么Twilio只针对非关键和非延时敏感的任务使用EBS，因为这不需要符合“故障单元是单台主机”原则。如果EBS遇到了问题，所有依赖它 的服务都会发生故障。他们转而关注于利于EC2 主机上的临时磁盘来做持久化。如果临时磁盘坏了，那么故障的范围仅仅是那台主机。Evan将发表一篇后续文 章来描述他们是如何跨过多个临时磁盘来做RAID0 以提升I/O性能的。

这与 SmugMug采用的原则和方法 是一致的，正如Don McAskill所说的那样，SmugMug也选择不用EBS。

Mike Kavis（M-Dot Network的CTO）认为 Amazon的IaaS已经变成了PaaS：

> Amazon拥有为数众多的服务，那些费时费力的任务都可以被简化并自动化进一次简单的调用中，开发者仅需一次调用就可以了。Cloudwatch（监控与自动扩展）和 RDS（数据库管理）就是其中的两个。当你开始使用这些服务后，你就已经置身于一个PaaS场景中了，你使用了某个厂商所独有服务。

对他而言，此类依赖和可能发生的故障都应该被纳入架构和业务模型之中，因为要构建一个云提供商未知的架构，如果不自己重新构建这些服务，几乎就是不现实的。

很明显，就算在云端，灾难恢复 计划也是必须的，架构无论现在还是以后，都会是构建基于云的解决方案的必备内容，这并不是什么新鲜观点。Twilio的原则是否足够？从中你是如何理解云架构的演进的？更多的冗余？自己开发服务？更多的架构原则？这将如何转变为基于PaaS的解决方案？

原文链接：http://www.infoq.com/cn/news/2011/04/twilio-cloud-architecture

相关内容：

- RIA的强力后盾：REST+海量存储
- Cloud Foundry遭遇存储故障
- RightScale提供基于Zend PHP的PaaS服务
- 张雷博士谈IBM沃森背后的AI技术
- 2010 年架构社区回顾：悠长的一年

热点新闻 | News

阅读者(十一):简单之美

作者 张逸

本书的内容有些另类，绝不似书名所呈现的中规中矩，但确实体现了一种美，是一种简单到极致的优雅，似乎又繁复如星空般的深邃，包容如峭立千仞之高的山壁。这是一本可以称之为轻松加愉快的思想随笔，又是一篇如杜拉拉升职记般的职场小说，它还贯穿了整个软件开发过程，揭露了从方法论、需求、架构设计、编码实现，到测试与维护以及团队管理的诸多要诀。这正是本书的另类之处。

我在阅读本书时，情不自禁地被放在书中每章篇首的实践场景所吸引，甚至忽略了本书的重要内容，直接根据提示转到下一个章节的实践场景，一气呵成，直到将这些实践场景阅读完毕。作者挥洒自如的文笔，入木三分的人物刻画，以及细腻含蓄的情感描写，将我彻底吸引住了。在大结局中，孔如之与儿子在阳光中巴黎圣母院前的对话，让人意犹未尽，似乎满怀希望，却又历尽沧桑，真是让我产生“情何以堪”的感慨。

这是本书感性的一面了。只是看完这 9 篇由实践场景片段组成的小说，就已经值回票价了。而从技术书籍的角度来看，本书的意义显然并不在于此，作者完整地勾勒出软件开发的全貌，诸多感悟与体会都可以成为软件开发人员的重要借鉴。作为本书理性的一面，这些内容需要反复阅读和分析琢磨，才能引起你的共鸣，许多模糊在心头的概念，在作者简明扼要的叙述下，或许就会产生“拨开云层见月明”的感悟。

以本书第 4 章为例。作者给出了一个简单的实例描述了框架构建的过程。首先从背景描述出发，展现了对保险业务中对保单进行处理的需求功能。这段背景描述将复杂的保险需求阐述得非常清楚而富有条理，体现了作者撰写文档的高超能力。

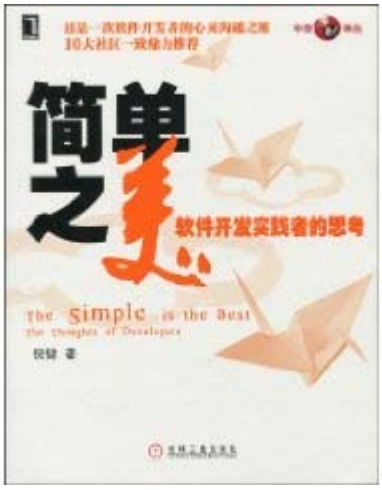
接下来是作者对这一背景描述的抽象。这段抽象有理有据，较好地体现了从需求捕获到分析的过程演变，利用抽象搭建了基本的领域模型。紧接着是对约束的思考，这是架构师必须完成的工作。根据对需求的抽象和关于约束的思考，就能够做出合理的架构决策。作者在本书中反复强调的“使用自然语言和讲故事的方式”，通过实例得到了具体的展现。事实上，在 Joel on software 一书中，Joel Spolsky 也提出了同样的观点，认为通过讲故事的方式描述用例场景，可以更好地促进理解与交流。本书作者扩大了这种方式的应用范围，引入到架构设计过

程中。对这一做法，我深表赞同。事实上，我在架构过程中，也常常采用类似手法，通过在文档撰写设计的故事场景，帮助我梳理设计思路，有时候，甚至在文档中自问自答，在这样的编写过程中我慢慢找到了解决方案。

本书对领域模型的讨论也有着个人独到的见解。例如他对静态模型和动态模型的分类，又例如他提出了使用贫血模型的好处。在 Martin Fowler 提出贫血模型之后，业界曾经掀起过对贫血模型与充血模型的争论。然而，争论到了最后，也没有一个确切的结果。从经典的 OO 原则来看，它要求将对象的数据和行为组织在一起，这正是批判贫血对象的主要论据。我比较倾向于这个观点，认为对象没有行为，就是"死"的，缺乏自治的能力。但在实际开发过程中，我也常常体会到贫血模型的好处，尤其是在模型重用与解耦方面，贫血模型都有其显著的优势。本书作者认为，贫血模型的"第一个好处是，有利于信息交换。第二个好处是，清晰了对象的职责。第三个好处是，实体对象（贫血对象）的实现更加灵活。第四个好处是，可以确保实体对象（贫血对象）只能在内存中用于计算。"这些好处都说到了点子上。虽然，我对于贫血对象的使用仍然抱有谨慎态度，但本书对此的阐述依旧给我提供了不错的参考。

在项目管理方面，书中强调了"负责制度"的实施。这首先关系到责任定位的问题。项目延迟或失败，究竟是项目管理的问题，还是架构设计的问题？是编码实现的问题，还是测试维护的问题？作者认为，负责制度的缺失可能会影响项目的质量。书中提到："在软件开发过程中，人是最重要的因素，而责任、权利和利益是保证这个因素发挥作用的关键。""建立负责制度的目的，不是为了惩罚某人，也不是为了永久取消某人的职业发展权利，它只是通过责任人利益损失的形式，来表明这样一个事实：没有金刚钻，别揽瓷器活。"事实上，负责制度的关键不在于制度的确立，而在于执行。如果没有创建公平、公开、公正的执行环境，这种制度只会给软件开发带来负面影响。这也是作者仅仅提出问题，却没有给出好的答案原因所在。相对而言，我个人更倾向于 Scrum"回顾会议"，在基于迭代与渐进式开发的基础上，这种方式更能够有效解决项目开发中存在的问题。

倘若是新手阅读本书，由于缺乏足够的工作阅历与开发经验，很难理解作者写作的意图。但我们绝对不能因为这种认识上的障碍，而将本书拒之千里之外。事实上，越早阅读本书，越能够开拓读者的眼界，提前感受业界的真实与谎言，反而能够帮助新手更快地确立自己的职业生涯规划。对于混迹行业多年的老鸟而言，阅读本书，一定能找到那些似曾相识的画面。作者对技术的深入探讨，也一定能给予我们启发，即使观点不同，也可以求同存异。所谓"嘤其鸣矣，求其友声"，这是我在阅读本书时收获的如遇旧友般的快乐！



图书信息

作者： 倪健

出版社：机械工业出版社

出版日期：2010 年 4 月

内容简介

本书不是一本关于方法论的理论性书籍（尽管已经尝试在大量的思考上进行一些理论归纳），也不是一本关于具体技术的操作手册。本书为读者呈现的是作者在软件 开发实践中的思考和体验，目的在于探究实践中的问题的根源，并给出思想上的解决之道。

本书以软件开发的基本顺序为主线，以简单、想象和文化的开发思想为线索，以为软件开发实践提供指导价值为目标。本书力图做到传递经过组织的、可操作的软件开发经验，使软件开发领域的新手在知识（软件开发思想）积累过程中少走弯路，也使软件开发领域的老手可以对照自己的经验把可能的困惑降到最低。有一些有趣例子为那些有价值的思考提供支持。

本书中，作者尝试对简单、想象和文化做一些形而上的思考。这些思考会比较抽象，但却是很多话题的基础。

作者介绍

倪健，资深软件工程师、项目经理和架构师，拥有 12 年 IT 行业从业经验。从事过电气设计与装配、单片机开发、软件系统设计与开发、项目管理、软件架构、软件培训以及性能调优相关的工作。精通汇编语言、VB、VC、Java 和 C#，自主设计过一些框架。12 年的工作经验帮助他建立了一个完整的认识世界的思想 体系，这个体系的核心内容就是简单、想象和文化。他擅长想象，善于归纳，内心坚持，追求有意义的生活。

原文链接：http://www.infoq.com/cn/news/2011/04/beauty-of-simple

相关内容：

- 书评：《代码之道》、书评：领域驱动设计与模式实战
- SOA治理：实现并维持业务及IT的敏捷性、王坚谈如何设计成功的互联网产品
- 微软云计算虚拟学院

InfoQ中文站

www.infoq.com/cn

我们的**使命**：成为关注软件开发领域变化和创新的专业网站

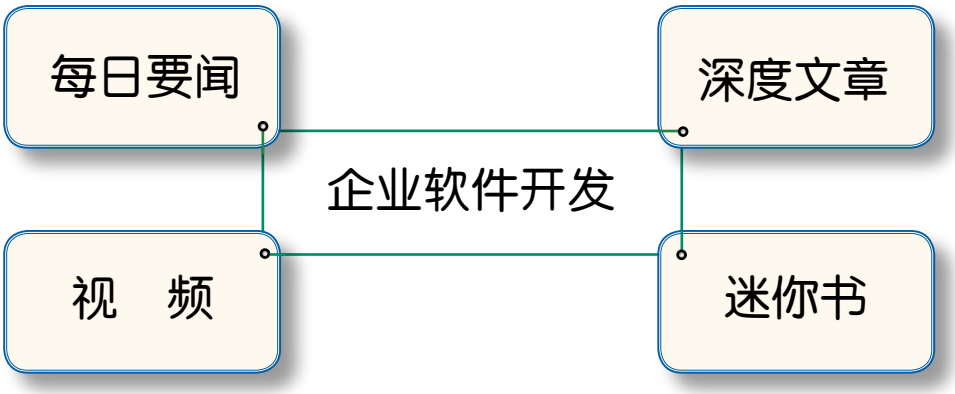
我们的**定位**：关注高级决策人员和大中型企业

我们的**社区**：Java、.NET、Ruby、SOA、Agile、Architecture

我们的**特质**：个性化RSS定制、国际化内容同步更新

我们的**团队**：超过30位领域专家担当社区编辑

……



讨 论 组：groups.google.com/group/infoqchina
编辑电邮：editors@cn.infoq.com
广告合作：sales@cn.infoq.com

特别专题 策划：InfoQ 中文站； 执行：胡键

我们正处于战国时代，一个每天都有新站点诞生和消亡的时代，一个名副其实的“网络应用”的战国时代。随着技术的发展，曾被视为“高科技”的网站，现在已经 成了随便几个未毕业的大学生就可凭借类似 ROR 或 PHP 这样的工具在几天内搭建出来的“高科技大白菜”。然而，入门门槛的降低并不意味着构建成功的站点就 是一门简单的“手艺”。相反，它反而更加拉开了“优秀”和“拙劣”之间的距离，加速了网络世界的新陈代谢。

本期《架构师》月刊邀请一些专家从自己的实践经验出发，讲述了自己对网站架构的看法。那么，准备好了吗？让我们现在开始这一次的站点架构之旅吧！

我眼中的网站架构

Sina App Engine 数据存储服务架构 — 又拍网架构中的分库设计

我眼中的云端架构 — Trunk.ly CTO 董洵谈架构

特别专题

Sina App Engine 数据存储服务架构

作者 丛磊

一、什么是 Sina App Engine

Sina App Engine（简称SAE）是新浪研发中心于 2009 年上旬开始内部开发，并在 2009 年正式推出第一个Alpha版本的国内首个公有云计算平台（http://sae.sina.com.cn），SAE是新浪云计算（简称浪云）战略的核心组成部分。

SAE作为国内的公有云计算，从开发伊始借鉴吸纳Google、Amazon等国外公司的公有云计算的成功技术经验，并很快推出不同于他们的具有自身特色的云计算平台。SAE选择在国内流行最广的Web开发语言PHP作为首选的支持语言，Web开发者可以在Linux/Mac/Windows上通过SDK或者Web版在线SDK进行开发、部署、调试，团队开发时还可以进行成员协作，不同的角色将对代码、项目拥有不同的权限；SAE提供了一系列分布式计算、存储服务供开发者使用，包括分布式文件存储Storage、分布式数据库集群RDC、分布式缓存MemcacheX、分布式定时服务Cron、分布式异步队列TaskQueue、邮件服务Mail等，这些服务将大大降低开发者的开发成本。同时又由于SAE整体架构的高可靠性和新浪的品牌保证，大大降低了开发者的运营风险。另外，作为典型的云计算，SAE采用"所付即所用，所付仅所用"的计费理念，通过日志和统计中心精确的计算每个应用的资源消耗（包括CPU、内存、磁盘等）[1]。

总之，SAE 就是分布式 Web 服务的开发、运行平台。

附，SAE 和传统的虚拟主机托管 VPS 的主要区别：

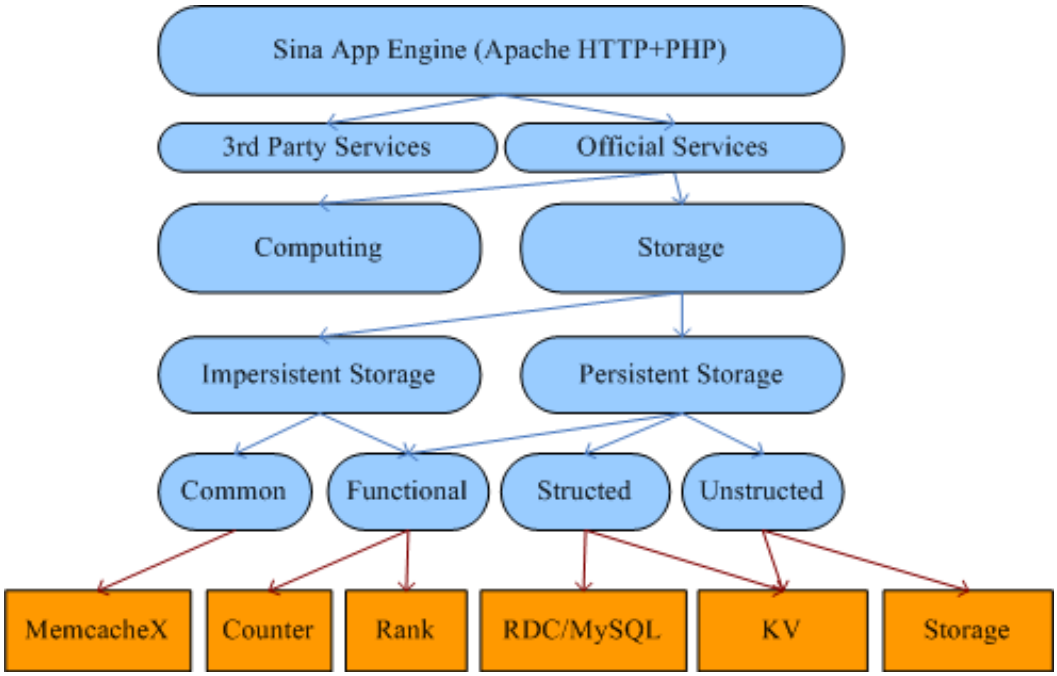
类项	SAE	VPS
核心用户	Web 开发者	无核心用户
使用方式	服务使用	设备租用
目标	力争覆盖 Web 服务所有需求，提供多种服务给开发者使用	仅基本需求
SLA（服务承诺）	高可靠性及严格的服务承诺	依服务商变化，无严格协议

计费方式	所付即所用，所付仅所用	预付费，无精确计费

SAE 在 2009 年 11 月 3 日发布了 alpha1 版本，2010 年 2 月 1 日发布了 alpha2 版本，2010 年 9 月 1 日发布了 beta 版本，截止到 2011 年 4 月 30 号，注册开发者已经接近 10000 人，活跃 App 接近 3000 个，每天独立代码部署行为超过 3000 次。

二、SAE 的数据存储服务

众所周知，Web 应用的最核心的是数据，数据对用户异常重要，对开发者本身也是最重要的信息。所以，保证用户和应用数据的可用性、可靠性、安全性就是云计算平台实现的重中之重。



SAE 数据存储关系图

如上图所示，SAE作为Web计算中心通过Apache HTTP+PHP提供给用户Web服务，Web服务通过API调用SAE内置的各种服务和外置的第三方服务（图中所示仅为SAE官方内置服务的数据存储部分）[2]。服务分成两类：计算型服务Computing和存储类服务Storage。而计算型服务又分成同步计算服务（如FetchURL、Image等）和异步计算（如Cron、TaskQueue等）服务，而存储服务则分成非持久化存储服务和持久化存储服务，本文只针对介绍存储服务。

非持久化存储服务包括了传统意义上的缓存服务，它能够提供大并发和高速读写，但是不保证数据的完全可靠，也就是说，当发生系统故障时可能会发生数据丢失和不一致。按照功能非持久化存储分成了通用缓存和功能性存储，SAE 上的 MemcacheX 就是通用性缓存，大家可以理解为 LAMP 架构中常见的 memcached。计数器服务 Counter 和排行榜服务 Rank 属于

功能性缓存，可以为特定场景下的需求提供更方便的服务，值得一提的是，计数器服务和排行榜服务的 HA 可靠性都比通用的 MemcacheX 高，具有一定的容灾和数据修复迁移能力，所以他们也兼有持久化存储的特性。

持久化存储服务用来存储用户需要落地的数据，这类服务强调高可靠性和高可用性，在分布式环境中以最终一致性来保证 HA。持久化存储服务根据存储数据是否为结构化又分成两类：结构化数据存储包括传统意义的关系型数据库 MySQL（在 SAE 中，以 RDC 提供）+NoSQL 数据库（KV，在开发中）和非结构化数据存储（分布式文件存储 Storage）。

从目前国内外流行的 Web 应用和网站看，MySQL 和 memcached 是最重要的服务组件，MySQL 提供实际数据存储，memcached 提供高速访问缓冲，甚至有些游戏网站直接拿 memcached 当实际存储，然后定时将其中的数据刷到 MySQL 中实现落地。从 Facebook、Twitter 等网站的公开文档中，也都可以看到，memcached 在其架构中被广泛使用，而且也没有放弃 MySQL，MySQL 仍作为其核心数据的存储服务。

所以，MySQL 和 memcached 服务对应的 RDC(Relational Database Cluster)和 MemcacheX 也是 SAE 的所有数据存储服务中最重要的组件，从实际的运维经验中也确实如此，像数据库的主从同步、缓存的失效和无法写入等问题都直接关系到线上应用的用户体验，所以本文着重介绍 RDC 和 MemcacheX，了解了这两个服务，也就了解了 SAE 数据存储的核心。

三、RDC（Relational Database Cluster）

随着 NoSQL 概念的兴起，传统 MySQL 关系型数据库在云计算时代遇到了挑战，无论是亚马逊还是 Google，在其云计算服务伊始都不约而同的避免了关系型数据库而主推其 NoSQL 数据库。但是，从 AWS SimpleDB 等服务的实际效果来看，普通开发者对于传统 MySQL 的依赖仍然十分严重。所以，出于降低开发者学习成本和迁移成本的考虑，SAE 从一开始就确定了“不能放弃 MySQL”，要对用户提供 MySQL 服务的目标，而这也就是 RDC 的设计和实现的初衷 。

对于 RDC，用户可以简单的理解其为 MySQL 的中间层或者代理层，类似 MySQL Proxy。所有的 SQL 请求都由 RDC 传递到后端的实际存储上。

RDC 作为对外提供服务的数据库集群，具有和传统 MySQL Proxy 截然不同的目标和要求，那是解决公有云数据库集群的安全性，为用户提供的强隔离性，而这个特性是私有云计算平台上的数据库集群所不具备的。除此之外，RDC 要求还具有对数据库的监控、管理功能，实时监控用户的 DB 的健康状况和主从同步延迟情况，一旦发现异常，将会在15秒内完成切换工作，用户的请求会被转发到其他正常的机器上。和某些私有云计算平台上的数据库集群提供的水平扩展和垂直划分不同，RDC 不提供用户数据库水平扩展，而是通过提供以下两个地址：

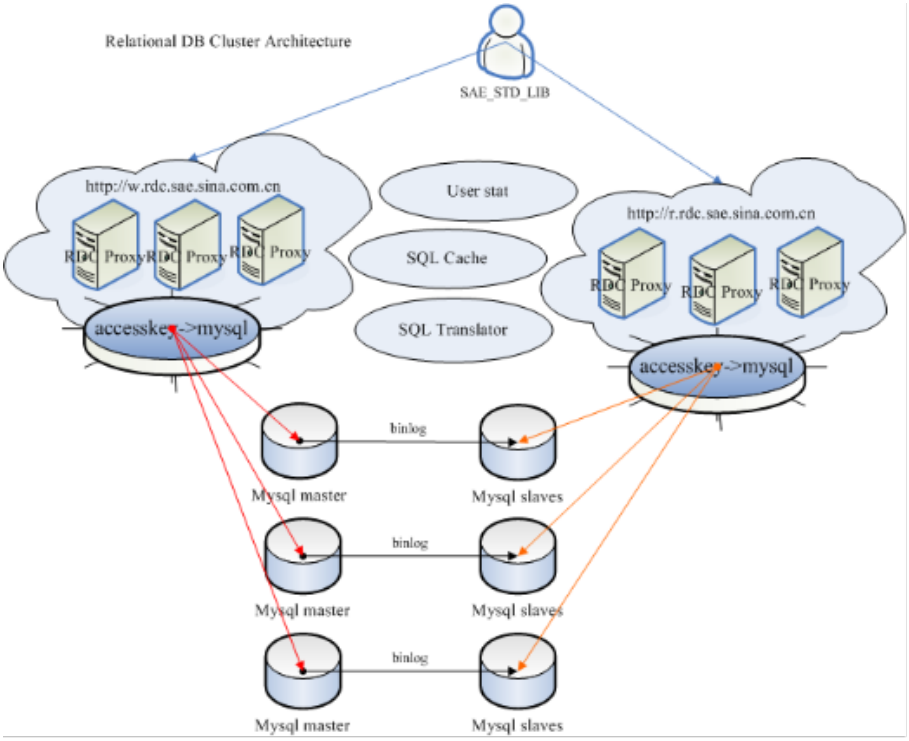
```
w.rdc.sae.sina.com.cn:3307 => 写库（主库）
r.rdc.sae.sina.com.cn:3307 => 读库（从库）
```

由应用层自行处理，对于 Sina App Engine 上的开发者，其实可以使用 PHP 语言级的数据库类(SaeMysql)自动实现读写分离。

RDC 从2010年9月开始完成设计并开始编码，到2010年底完成测试，其自身采用的是多进程非阻塞方式的通讯框架，自身性能消耗较少，根据内部性能测试，开启 SQL cache 时对比直接访问 MySQL 的性能损耗在5%以内。

RDC 在2010年底正式上线，目前 SAE 上所有的数据库访问都通过 RDC 来提供，并且已经稳定运行了半年。RDC 虽然为分布式数据库集群，但是对终端用户完全透明，用户在使用 RDC 时，不会感觉到和使用传统的 MySQL 数据库有任何差异，甚至用户都不用知道 RDC 的存在。用户可以使用所有 MySQL 标准客户端（MySQL5以上）操作 RDC，如 mysql_query、mysql_connect 等，错误处理和标准 MySQL 客户端处理模式一样，用户操作 RDC 的示例代码：

```
$con=mysql_connect('w.rdc.sae.sina.com.cn:'.$_SERVER['HTTP_MYSQLPORT'],SAE_
ACCESSKEY,SAE_SECRETKEY);
if (!$con)
    die('could not connect: ' . mysql_error());
$sql = "SELECT * FROM Person";
if(!mysql_query($sql,$con))
    echo mysql_errno() . ": " . mysql_error() . "\n";
```



RDC 整体架构图

RDC的整体架构如上图所示，开发者通过SAE StdLib（自动支持主从分离）或者标准MySQL PHP函数（手工访问w.rdc.sae.sina.com.cn:3307和r.rdc.sae.sina.com.cn:3307来控制主从分离）在SAE平台上调用RDC。当一条SQL语句到达RDC时，RDC先将其翻译解析，然后查看cache的解析结果，如果命中则根据结果做出判断（SQL预判，见下文），如果没有命中则送到后端（根据accesskey到DB的映射关系）进行判断。判断结果完成后，如果是拦截则直接返回用户MySQL Error并通过errmsg告知拦截原因，否则就进行正常的SQL访问。

RDC支持的数据库后端的数量为百万级，其自身设计为无状态结构，并且可以随时水平扩展（而后端数据库的水平扩展由用户自己负责），accesskey到DB的映射关系由SAE自身的支持分布式一致性的trigger服务保障。

RDC后端的DB为标准的MySQL一主多从结构，具体几个从库由用户级别对应的SLA决定（默认是两个从库），主从之间通过binlog做最终一致性保证。目前RDC上的MySQL的存储引擎为MyISAM，以后会支持InnoDB。RDC不支持非SAE环境直连，这里主要是因为用户客户端如果不在SAE的环境中就很难保证和数据库的网络状况。用户如果想管理数据库可以通过SAE应用页面里的PHPAdmin进行操作。

在编码实现上，RDC类似MySQL5（除去存储引擎部分），但在细节上和标准MySQL还是有几个显著区别：

1. RDC采用多进程实现（SAE上的服务几乎都是多进程结构），而标准MySQL采用多线程
2. RDC对SQL语句的解析是词法解析，而标准MySQL parser则是标准的语法解析，这样做的好处是词法分析更快，而RDC并不需要完整的挂接语义动作
3. RDC的cache层在词法分析后，这意味着即使用户的SQL语句有些改动，比如多了1个空格，仍会命中cache；而标准MySQL的query cache[3]则在语法分析前，SQL语句即使多了1个空格，仍然会使cache失效

RDC自带一个监控进程，监控分成两部分：健康监控和主从监控。健康监控会定期连接DB，发现故障时，自动做切换。主从监控通过分析slave stat和binlog pos来确定主从延迟情况，发现高延迟时，将请求自动切换到低延迟的DB。RDC的监控还自带了网络状况检查和跨IDC配置，以支持跨IDC部署RDC需求。

安全性保证

安全性是RDC作为公有云计算数据库服务的最大挑战，试想，如果一个初学用户在上千万记录的表上执行不带where语句的select是什么后果，如果一个恶意用户故意频繁执行大量的join是什么后果。为了解决这些问题，RDC设计了三道“防护线”的策略来保护用户数据库

的隔离性。

SQL 预判机制

RDC 会根据自身算法预先评估 SQL 语句的效率，当发现有可能是性能不高的 SQL 语句时，会对开发者做出提示（这在 RDC 内部叫软拦截），而当发现可能对整个数据库平台造成伤害的 SQL 语句时，会直接在 RDC 层将其屏蔽掉，并以标准 MySQL 的错误返回方式告知开发者。
RDC 会对以下几种语句做出预判：

select

update

insert

delete

replace

create table

alter table

create index

以最复杂的 select 语句为例，RDC 的判断依据包括了：select 语句是否带 where 语句、查询条件是否带了恰当的索引、是否带有子查询、是否带有连表 join 操作、子查询和 join 和主查询的关系、join 和子查询的复杂度、是否有不恰当的排序、是否有 limit 限定等。
目前，该判断算法已经申请专利。

并发执行时间和

RDC 率先提出这个概念，来取代传统 MySQL 的并发连接数限制，原因主要是因为传统的连接数限制较为粗鲁，不能区分用户，换句话说， 无法区分"SQL 优化得好的用户"和"SQL 执行性能低下的用户"。而并发执行时间和很好的解决了这个问题，
并发执行时间和=当前所有活的请求已经执行的时间和
举个例子，如果 SQL 并发执行时间和为10000ms：

A 用户的平均每条 SQL 消耗100ms，那么 A 获得的最大并发为100

B 用户的平均每条 SQL 消耗1000ms，那么 B 获得最大并发仅为10

并发执行时间和从技术层面突出一个产品引导：“对于 SQL 效率高的用户支持更大的并发，

而 SQL 执行效率低的用户则可能不会获得大的并发”，以鼓励用户优化自己的 SQL，优化自己的表结构，提高 SQL 的执行效率，减少对系统的消耗。

慢查询保护

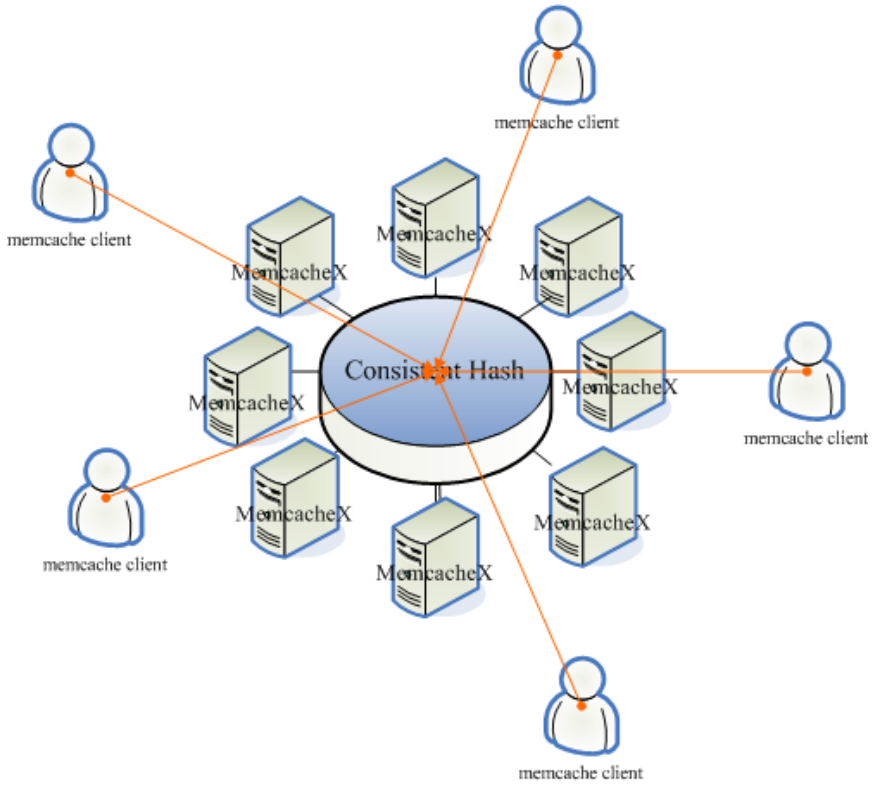
MySQL 自身会产生慢查询日志，正常应用的慢查询肯定不会超过一定额度，那么超过这个额度我们就会给出建议和报警，如果应用超过慢查询系统上限，则可能导致 MySQL 服务暂时禁用（过十分钟会自动开启），以保护其他的数据库服务正常使用并督促应用的开发者尽快修改代码或作 MySQL 结构优化。

通过上面三种不同层面的策略，有效的保护了 RDC 的后端实际数据库。从实际运行效果看，98%正常应用并不会受到 RDC SQL 保护的影响，而%1的应用因为数据库设计编码经验不足而触发了大量的 RDC 拦截，用户可以在 SAE 日志中心实时的看到这些日志，日志中心还会针对这些日志给出 MySQL 优化改进建议。事实证明，通过 RDC 使 SAE 避免了多次重大数据库故障。

四、MemcacheX

传统的公有云计算平台的 memcache 服务，往往是给开发者提供一个类 memcached 的实例，用户可以自主使用其 memcache 而不影响其他用户的 memcache。但这种实现有个致命缺陷：即当 memcached 实例出现故障时，会使应用全部缓存 key 穿透数据库，而很可能造成重大故障。除此之外别的缺点也不少，比如成本过大，每个 memcached 都有额外的内存 overhead，从而浪费了不少空间，再比如扩容时必须重启服务，这样导致扩容会导致用户的 key 失效等等。

SAE MemcacheX 服务设计和实现的目的就是克服上述缺点，代替现有的 memcached 服务，从而降低运行成本，并且为用户提供高可靠性的缓存服务。



MemcacheX 架构图

从上图可以看出，用户通过标准memcache客户端使用MemcacheX服务，用户的key通过客户端的一致性hash分散在所有的MemcacheX节点上，这点和标准的memcached服务几乎一样，但不同的是每个用户的key加上了他们的应用名做前缀区分，这样多个用户的key可以共享一个MemcacheX实例，甚至共享同一套slab内存池，提高了内存使用率。每个用户都独立享有实例的统计信息和LRU信息，也就是说，当某个用户的memcache内存使用到了其申请的大小上限时，key只会和本用户的key做LRU交换，而不会影响其他的key。MemcacheX还对slab算法进行了改进，因为memcached的slab算法并没有伙伴系统[4]的回收逻辑，所以可能会导致在明明还有内存的时候，分配不成功，MemcacheX对此的改进有效的降低了这种可能性。MemcacheX支持内存自动扩容，扩容过程不用重启服务，所以用户无需担心扩容后的穿透问题。

因为每个用户的 key 都会被一致性 hash 在所有 MemcacheX 的 N 个节点上，所以当某个节点出现故障时，理论上只有 1/N 的 key 受到影响，而当 N 足够大时，就不会对用户的应用造成致命影响，从而提高了 HA。

MemcacheX 还具备一个功能，就是 connection LRU 保护。新浪动态应用平台的实际运维经验表明，因为系统的不可靠性和 Web 开发人员代码质量的不可控性，传统 memcached 容易造成连接资源耗光的的情况，比如下面的代码：

```
$memcache_obj=new Memcache;
$memcache_obj->connect('memcache_host',11211);
if($memcache_obj->get('key')!=false)
    do_mysql_query()
$memcache_obj->close();
```

试想，当数据库操作 do_mysql_query()因为某种原因阻塞住时，该代码就会造成 memcached 的连接资源没能及时释放，当并发访问量很大时，可能造成瞬间 memcached 文件描述符资源耗尽，从而影响服务，即使当数据库恢复正常时，memcached 仍然不可用。

MemcacheX 的 connection LRU 就是解决这个问题的，connection LRU 是指当极端情况下的连接资源耗尽时，可以淘汰旧的连接以保证可以正确建立新的连接。这样做固然会影响某些应用的正常逻辑，但可以在特殊时期保证新用户的正常使用。

2011 年 3 月，MemcacheX 已经在 SAE 上线。经过测试和实践检验，MemcacheX 和传统 memcached 拥有同样的性能，并且具有更高的可靠性。

通过RDC和MemcacheX，SAE很好的支持了近万开发者的数据库和缓存服务，RDC每天请求量超过 6000 万，MemcacheX的每天请求量则超过 1 亿，两者共同承担了 90%以上数据读写需求，也正是因为这两个服务，才保证了公有云计算平台上的用户数据读写的可用性和可靠性。另外，除了RDC和MemcacheX，Sina App Engine技术团队近期还研发了很多重量级产品[5]：支持系统层文件排重的分布式强一致性文件部署系统CodeFS、计数器服务Counter、排行榜服务Rank，以及正在开发中的分布式NoSQL存储SAE KV。对于这些服务的架构和技术细节，我会在以后的文章和大家分享。

欢迎使用 SAE：

1. Sina App Engine注册页面，http://sae.sina.com.cn/
2. Sina App Engine体验账号申请页面，http://sae.sina.com.cn/?m=home&a=exp
3. 微博开发平台SAE账号申请页面，http://open.t.sina.com.cn/apps/new

参考资料：

1. Sina App Engine架构-云计算时代的分布式Web服务解决方案
2. Sina App Engine-deep inside cloud service
3. MySQL query cache
4. Buddy System
5. SAE服务总体说明

特别专题

又拍网架构中的分库设计

作者 周兆兆

又拍网是一个照片分享社区，从 2005 年 6 月至今积累了 260 万用户，1.1 亿张照片，目前的日访问量为 200 多万。5 年的发展历程里经历过许多起伏，也积累了一些经验，在这篇文章里，我要介绍一些我们在技术上的积累。

又拍网和大多数Web2.0 站点一样，构建于大量开源软件之上，包括 MySQL、PHP、nginx、Python、memcached、redis、Solr、Hadoop和 RabbitMQ等等。又拍网的服务器端开发语言主要是 PHP和 Python ，其中 PHP用于编写Web逻辑（通过HTTP和用户直接打交道）， 而 Python则主要用于开发内部服务和后台任务。在客户端则使用了大量的Javascript， 这里要感谢一下 MooTools这个JS框架，它使得我们很享受前端开发过程。 另外，我们把图片处理过程从 PHP进程里独立出来变成一个服务。这个服务基于 nginx，但是是作为 nginx的一个模块而开放REST API。

图 1：开发语言

由于 PHP的单线程模型，我们把耗时较久的运算和I/O操作从HTTP请求周期中分离出来， 交给由 Python实 现的任务进程来完成，以保证请求响应速度。这些任务主要包括：邮件发送、数据索引、数据聚合和好友动态推送（稍候会有介绍）等等。通常这些任务由用户触 发，并且，用户的一个行为可能会触发多种任务的执行。 比如，用户上传了一张新的照片，我们需要更新索引，也需要向他的朋友推送一条新的动态。PHP通过消息队列（我们用的是 RabbitMQ ）来触发任务执行。

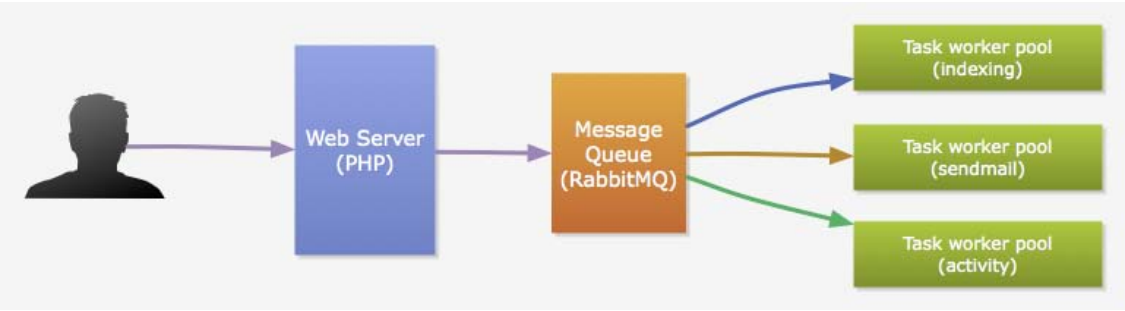


图 2：PHP 和 Python 的协作

数据库一向是网站架构中最具挑战性的，瓶颈通常出现在这里。又拍网的照片数据量很大，数据库也几度出现严重的压力问题。 因此，这里我主要介绍一下又拍网在分库设计这方面的一些尝试。

分库设计

和很多使用 MySQL的 2.0 站点一样，又拍网的 MySQL集群经历了从最初的一个主库一个从库、到一个主库多个从库、 然后到多个主库多个从库的一个发展过程。

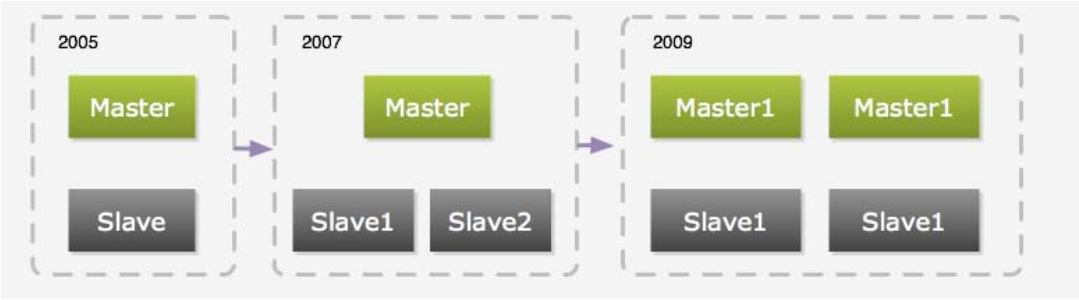


图 3：数据库的进化过程

最初是由一台主库和一台从库组成，当时从库只用作备份和容灾，当主库出现故障时，从库就手动变成主库，一般情况下，从库不作读写操作（同步除外）。随着压力的增加，我们加上了 memcached，当时只用其缓存单行数据。 但是，单行数据的缓存并不能很好地解决压力问题，因为单行数据的查询通常很快。所以我们把一些实时性要求不高的Query放到从库去执行。后面又通过添加多个从库来分流查询压力，不过随着数据量的增加，主库的写压力也越来越大。

在参考了一些相关产品和其它网站的做法后，我们决定进行数据库拆分。也就是将数据存放到不同的数据库服务器中，一般可以按两个纬度来拆分数据:

垂直拆分：是指按功能模块拆分，比如可以将群组相关表和照片相关表存放在不同的数据库中，这种方式多个数据库之间的**表结构不同**。

水平拆分：而水平拆分是将同一个表的数据进行分块保存到不同的数据库中，这些数据库中的**表结构完全相同**。

拆分方式

一般都会先进行垂直拆分，因为这种方式拆分方式实现起来比较简单，根据表名访问不同的数据库就可以了。但是垂直拆分方式并不能彻底解决所有压力问题，另外，也要看应用类型是否合适这种拆分方式。如果合适的话，也能很好的起到分散数据库压力的作用。比如对于豆瓣我觉得比较适合采用垂直拆分， 因为豆瓣的 各核心业务/模块（书籍、电影、音乐）相对独立，数据的增加速度也比较平稳。不同的是，又拍网的核心业务对象是用户上传的照片，而照片数据的增加速度随着 用户量的增加越来越快。压力基本上都在照片表上，显然垂直拆分并不能从根本上解决我们的问题，所以，我们采用水平拆分的方式。

拆分规则

水平拆分实现起来相对复杂，我们要先确定一个拆分规则，也就是按什么条件将数据进行切分。 一般 2.0 网站都以用户为中心，数据基本都跟随用户，比如用户的照片、朋友和评论等等。因此一个比较自然的选择是根据用户来切分。每个用户都对应一个数据库，访问某个用户的数据时， 我们要先确定他/她所对应的数据库，然后连接到该数据库进行实际的数据读写。

那么，怎么样对应用户和数据库呢？我们有这些选择：

按算法对应

最简单的算法是按用户 ID 的奇偶性来对应，将奇数 ID 的用户对应到数据库 A，而偶数 ID 的用户则对应到数据库 B。这个方法的最大问题是，只能分成两个库。另一个算法是按用户 ID 所在区间对应，比如 ID 在 0-10000 之间的用户对应到数据库 A， ID 在 10000-20000 这个范围的对应到数据库 B，以此类推。按算法分实现起来比较方便，也比较高效，但是不能满足后续的伸缩性要求，如果需要增加数据库节点，必需调整算法或移动很大的数据集， 比较难做到在不停止服务的前提下进行扩充数据库节点。

按索引/映射表对应

这种方法是指建立一个索引表，保存每个用户的 ID 和数据库 ID 的对应关系，每次读写用户数据时先从这个表获取对应数据库。新用户注册后，在所有可用的数据库中随机挑选一个为其建立索引。这种方法比较灵活，有很好的伸缩性。一个缺点是增加了一次数据库访问，所

以性能上没有按算法对应好。

比较之后，我们采用的是索引表的方式，我们愿意为其灵活性损失一些性能，更何况我们还有 memcached， 因为索引数据基本不会改变的缘故，缓存命中率非常高。所以能很大程度上减少了性能损失。

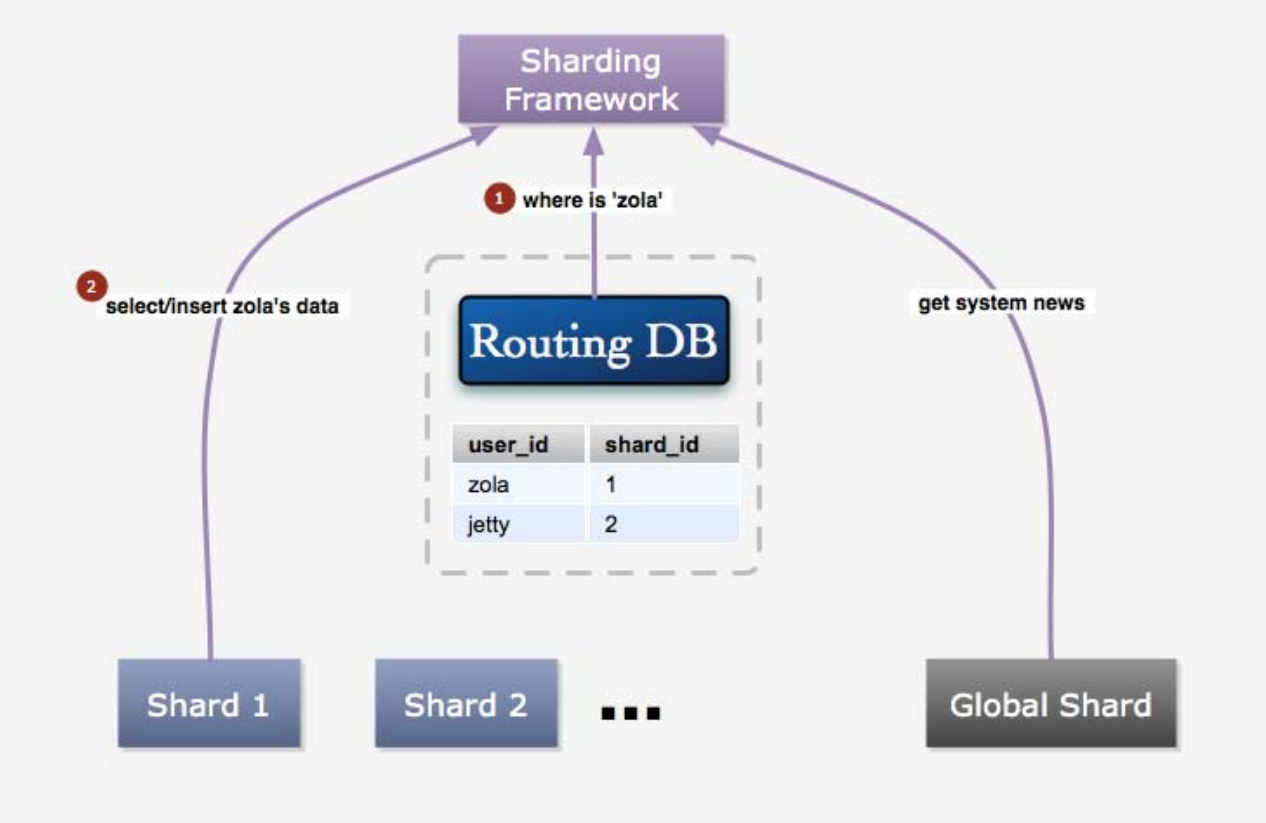


图 4：数据访问过程

索引表的方式能够比较方便地添加数据库节点，在增加节点时，只要将其添加到可用数据库列表里即可。 当然如果需要平衡各个节点的压力的话，还是需要进行数据的迁移，但是这个时候的迁移是少量的，可以逐步进行。要迁移用户 A 的数据，首先要将其状态置为迁移数据中，这个状态的用户不能进行写操作，并在页面上进行提示。 然后将用户 A 的数据全部复制到新增加的节点上后，更新映射表，然后将用户 A 的状态置为正常，最后将原来对应的数据库上的数据删除。这个过程通常会在临晨进行，所以，所以很少会有用户碰到迁移数据中的情况。

当然，有些数据是不属于某个用户的，比如系统消息、配置等等，我们把这些数据保存在一个全局库中。

问题

分库会给你在应用的开发和部署上都带来很多麻烦。

不能执行跨库的关联查询

如果我们需要查询的数据分布于不同的数据库，我们没办法通过JOIN的方式查询获得。比如要获得好友的最新照片，你不能保证所有好友的数据都在同一个数据 库里。一个解决办法是通过多次查询，再进行聚合的方式。我们需要尽量避免类似的需求。有些需求可以通过保存多份数据来解决，比如User-A和User- B的数据库分别是DB-1 和DB-2， 当User-A评论了User-B的照片时，我们会同时在DB-1 和DB-2 中保存这条评论信息，我们首先在DB-2 中的photo_comments表 中插入一条新的记录，然后在DB-1 中的user_comments表中插入一条新的记录。这两个表的结构如下图所示。这样我们可以通过查询 photo_comments表得到User-B的某张照片的所有评论， 也可以通过查询user_comments表获得User-A的所有评论。另外可以考虑使用全文检索工具来解决某些需求， 我们使用 Solr来提供全站标签检索和照片搜索服务。

photo_comments

column	type
photo_id	int
comment_id	int
author_id	int
posted_at	datetime
content	text

user_comments

column	type
user_id	int
comment_id	int
photo_owner_id	int
photo_id	int
posted_at	datetime

图 5：评论表结构

不能保证数据的一致/完整性

跨库的数据没有外键约束，也没有事务保证。比如上面的评论照片的例子， 很可能出现成功插入 photo_comments 表，但是插入 user_comments 表时却出错了。一个办法是在两个库上都开启事务，然后先插入 photo_comments，再插入 user_comments， 然后提交两个事务。这个办法也不能完全保证这个操作的原子性。

所有查询必须提供数据库线索

比如要查看一张照片，仅凭一个照片 ID 是不够的，还必须提供上传这张照片的用户的 ID(也就是数据库线索)，才能找到它实际的存放位置。因此，我们必须重新设计很多 URL 地址，而有些老的地址我们又必须保证其仍然有效。我们把照片地址改成

/photos/{username}/{photo_id}/的形式，然后对于系统升级前上传的照片 ID， 我们又增加一张映射表，保存 photo_id 和 user_id 的对应关系。当访问老的照片地址时，我们通过查询这张表获得用户信息，然后再重定向到新的地址。

自增 ID

如果要在节点数据库上使用自增字段，那么我们就不能保证全局唯一。这倒不是很严重的问题，但是当节点之间的数据发生关系时，就会使得问题变得比较麻烦。我们可以再来看看上面提到的评论的例子。如果 photo_comments 表中的 comment_id 的自增字段，当我们在 DB-2.photo_comments 表插入新的评论时， 得到一个新的 comment_id，假如值为 101，而 User-A 的 ID 为 1，那么我们还需要在 DB-1.user_comments 表中插入(1, 101 ...)。 User-A 是个很活跃的用户，他又评论了 User-C 的照片，而 User-C 的数据库是 DB-3。 很巧的是这条新评论的 ID 也是 101，这种情况很用可能发生。那么我们又在 DB-1.user_comments 表中插入一行像这样(1, 101 ...)的数据。 那么我们要怎么设置 user_comments 表的主键呢（标识一行数据）？可以不设啊，不幸的是有的时候（框架、缓存等原因）必需设置。那么可以以 user_id、comment_id 和 photo_id 为组合主键，但是 photo_id 也有可能一样（的确很巧）。看来只能再加上 photo_owner_id 了， 但是这个结果又让我们实在有点无法接受，太复杂的组合键在写入时会带来一定的性能影响，这样的自然键看起来也很不自然。所以，我们放弃了在节点上使用自增字段，想办法让这些 ID 变成全局唯一。为此增加了一个专门用来生成 ID 的数据库，这个库中的表结构都很简单，只有一个自增字段 id。 当我们要插入新的评论时，我们先在 ID 库的 photo_comments 表里插入一条空的记录，以获得一个唯一的评论 ID。 当然这些逻辑都已经封装在我们的框架里了，对于开发人员是透明的。 为什么不用其它方案呢，比如一些支持 incr 操作的 Key-Value 数据库。我们还是比较放心把数据放在 MySQL 里。 另外，我们会定期清理 ID 库的数据，以保证获取新 ID 的效率。

实现

我们称前面提到的一个数据库节点为Shard，一个Shard由两个台物理服务器组成， 我们称它们为Node-A和Node-B，Node-A和Node-B之间是配置成Master-Master相互复制的。 虽然是Master-Master的部署方式，但是同一时间我们还是只使用其中一个，原因是复制的延迟问题，当然在Web应用里，我们可以在用户会话里放置一个A或B来保证同一用户一次会话里只访问一个数据库， 这样可以避免一些延迟问题。但是我们的 Python任务是没有任何状态的，不能保证和 PHP应 用读写相同的数据库。那么为什么不配置成Master-Slave呢？我们觉得只用一台太浪费了，所以我们在每台服务器上都创建多个逻辑数据库。 如下图所示，在Node-A和Node-B上我们都建立了shard_001 和shard_002 两个逻辑数据库， Node-A上的shard_001

和Node-B上的shard_001 组成一个Shard，而同一时间只有一个逻辑数据库处于Active状态。这个时候如果需要访问Shard-001 的数据时，我们连接的是Node-A上的shard_001， 而访问Shard-002 的数据则是连接Node-B上的shard_002。以这种交叉的方式将压力分散到每台物理服务器上。 以Master-Master方式部署的另一个好处是，我们可以不停止服务的情况下进行表结构升级， 升级前先停止复制，升级Inactive的库，然后升级应用，再将已经升级好的数据库切换成Active状态， 原来的Active数据库切换成Inactive状态，然后升级它的表结构，最后恢复复制。 当然这个步骤不一定适合所有升级过程，如果表结构的更改会导致数据复制失败，那么还是需要停止服务再升级的。

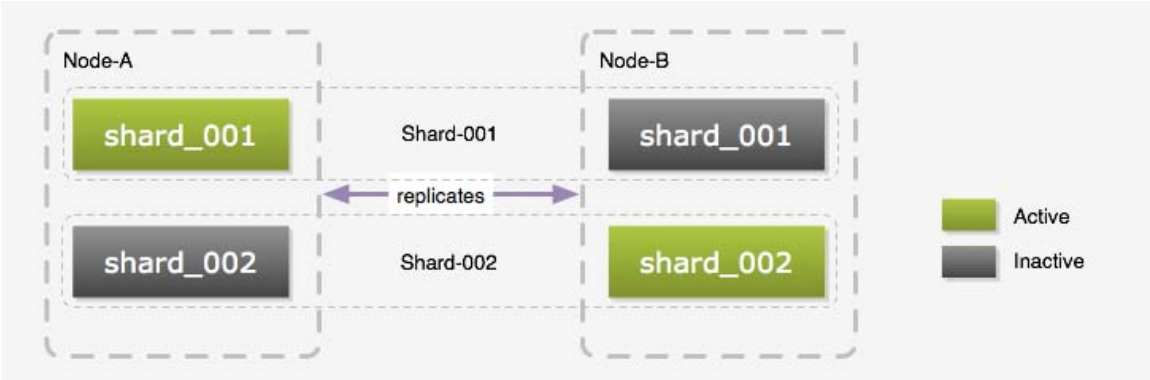


图 6：数据库布局

前面提到过添加服务器时，为了保证负载的平衡，我们需要迁移一部分数据到新的服务器上。为了避免短期内迁移的必要，我们在实际部署的时候，每台机器上部署了 8 个逻辑数据库，添加服务器后，我们只要将这些逻辑数据库迁移到新服务器就可以了。最好是每次添加一倍的服务器， 然后将每台的 1/2 逻辑数据迁移到一台新服务器上，这样能很好的平衡负载。当然，最后到了每台上只有一个逻辑库时，迁移就无法避免了，不过那应该是比较久远的事情了。

我们把分库逻辑都封装在我们的 PHP 框架里了，开发人员基本上不需要被这些繁琐的事情困扰。下面是使用我们的框架进行照片数据的读写的一些例子:

```
<?php
   $Photos = new ShardedDBTable('Photos', 'yp_photos', 'user_id',
                    array(
                       'photo_id'    => array(
                               'type' => 'long',
                               'primary' => true,
                               'global_auto_increment' => true),
                       'user_id'     => array('type' => 'long'),
                       'title'       => array('type' => 'string'),
                       'posted_date' => array('type' => 'date'),
```

```
                ));

    $photo = $Photos->new_object(
                array('user_id' => 1,
                        'title' => 'Workforme'));
    $photo->insert();

    // 加载ID为10001的照片，注意第一个参数为用户ID
    $photo = $Photos->load(1, 10001);

    // 更改照片属性
    $photo->title = 'Database Sharding';
    $photo->update();

    // 删除照片
    $photo->delete();

    // 获取ID为1的用户在2010-06-01之后上传的照片
    $photos = $Photos->fetch(
                  array('user_id' => 1,
                    'posted_date__gt' => '2010-06-01'));
?>
```

首先要定义一个 ShardedDBTable 对象，所有的 API 都是通过这个对象开放。第一个参数是对象类型名称， 如果这个名称已经存在，那么将返回之前定义的对象。你也可以通过 get_table('Photos')这个函数来获取之前定义的 Table 对象。 第二个参数是对应的数据库表名，而第三个参数是数据库线索字段，你会发现在后面的所有 API 中全部需要指定这个字段的值。第四个参数是字段定义 其中 photo_id 字段的 global_auto_increment 属性被置为 true，这就是前面所说的全局自增 ID， 只要指定了这个属性，框架会处理好 ID 的事情。

如果我们要访问全局库中的数据，我们需要定义一个 DBTable 对象。

```
<?php
   $Users = new DBTable('Users', 'yp_users',
               array( 'user_id'  =>
                      array('type' => 'long',
                          'primary' => true,
                          'auto_increment' => true),
               'username' => array('type' => 'string'),
              ));
?>
```

DBTable 是 ShardedDBTable 的父类，除了定义时参数有些不同（DBTable 不需要指定数据库线索字段），它们提供一样的 API。

缓存

我们的框架提供了缓存功能，对开发人员是透明的。

```
<?php
    $photo = $Photos->load(1, 10001);
?>
```

比如上面的方法调用，框架先尝试以 Photos-1-10001 为 Key 在缓存中查找，未找到的话再执行数据库查询并放入缓存。当更改照片属性或删除照片时，框架负责从缓存中删除该照片。这种单个对象的缓存实现起来比较简单。稍微麻烦的是像下面这样的列表查询结果的缓存。

```
<?php
    $photos = $Photos->fetch(array('user_id' => 1, 'posted_date__gt' =>
'2010-06-01'));
?>
```

我们把这个查询分成两步，第一步先查出符合条件的照片 ID，然后再根据照片 ID 分别查找具体的照片信息。 这么做可以更好的利用缓存。第一个查询的缓存 Key 为 Photos-list-{shard_key}-{md5(查询条件 SQL 语句)}， Value 是照片 ID 列表（逗号间隔）。其中 shard_key 为 user_id 的值 1。目前来看，列表缓存也不麻烦。 但是如果用户修改了某张照片的上传时间呢，这个时候缓存中的数据就不一定符合条件了。所以，我们需要一个机制来保证我们不会从缓存中得到过期的列表数据。我们为每张表设置了一个 revision，当该表的数据发生变化时（调用 insert/update/delete 方法） ， 我们就更新它的 revision，所以我们把列表的缓存 Key 改为 Photos-list-{shard_key}-{md5(查询条件 SQL 语句)}-{revision}， 这样我们就不会再得到过期列表了。

revision 信息也是存放在缓存里的，Key 为 Photos-revision。这样做看起来不错，但是好像列表缓存的利用率不会太高。因为我们是以整个数据类型的 revision 为缓存 Key 的后缀，显然这个 revision 更新的非常频繁，任何一个用户修改或上传了照片都会导致它的更新，哪怕那个用户根本不在我们要查询的 Shard 里。要隔离用户的动作对其他用户的影响，我们可以通过缩小 revision 的作用范围来达到这个目的。 所以 revision 的缓存 Key 变成 Photos-{shard_key}-revision，这样的话当 ID 为 1 的用户修改了他的照片信息时， 只会更新 Photos-1-revision 这个 Key 所对应的 revision。

因为全局库没有 shard_key，所以修改了全局库中的表的一行数据，还是会导致整个表的缓存失效。 但是大部分情况下，数据都是有区域范围的，比如我们的帮助论坛的主题帖子， 帖子属于主题。修改了其中一个主题的一个帖子，没必要使所有主题的帖子缓存都失效。 所以我们在 DBTable 上增加了一个叫 isolate_key 的属性。

```
<?php
$GLOBALS['Posts'] = new DBTable('Posts', 'yp_posts', array(
       'topic_id'    => array('type' => 'long', 'primary' => true),
       'post_id'         =>  array('type'  =>  'long',  'primary'  =>  true,
'auto_increment' => true),
       'author_id'   => array('type' => 'long'),
       'content'     => array('type' => 'string'),
       'posted_at'   => array('type' => 'datetime'),
       'modified_at' => array('type' => 'datetime'),
       'modified_by' => array('type' => 'long'),
   ), 'topic_id');
?>
```

注意构造函数的最后一个参数 topic_id 就是指以字段 topic_id 作为 isolate_key，它的作用和 shard_key 一样用于隔离 revision 的作用范围。

ShardedDBTable 继承自 DBTable，所以也可以指定 isolate_key。ShardedDBTable 指定了 isolate_key 的话，能够更大幅度缩小 revision 的作用范围。 比如相册和照片的关联表 yp_album_photos，当用户往他的其中一个相册里添加了新的照片时， 会导致其它相册的照片列表缓存也失效。如果我指定这张表的 isolate_key 为 album_id 的话， 我们就把这种影响限制在了本相册内。

我们的缓存分为两级，第一级只是一个 PHP 数组，有效范围是 Request。而第二级是 memcached。这么做的原因是，很多数据在一个 Request 周期内需要加载多次，这样可以减少 memcached 的网络请求。另外我们的框架也会尽可能的发送 memcached 的 gets 命令来获取数据， 从而减少网络请求。

总结

这个架构使得我们在很长一段时间内都不必再为数据库压力所困扰。我们的设计很多地方参考了 netlog和 flickr的实现，因此非常感谢他们将一些实现细节发布出来。

关于作者

周兆兆（Zola,不是你熟知的那个），又拍网架构师。6 年 IT 从业经验，不太专注于某项技术，对很多技术都感兴趣。

原文链接：http://www.infoq.com/cn/articles/cloud-computing-soa

相关内容： 洪强宁谈豆瓣网技术架构、Raven——一个.NET 上的文档数据库、百度大规模数据处理、实施云计算之前需要问的 6 个问题、豆瓣数据存储实践

特别专题

我眼中的云端架构

作者 肖伟

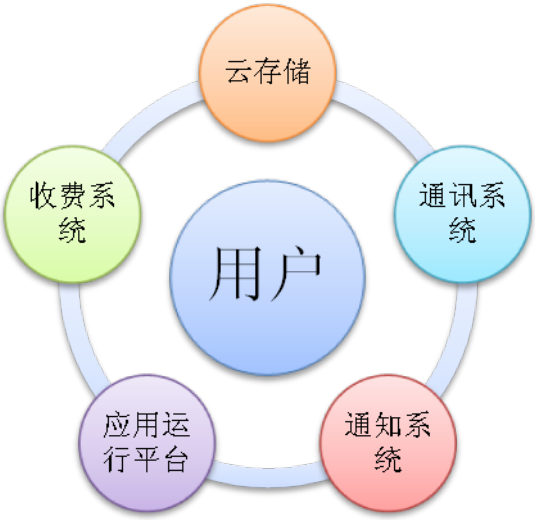
背景

在看阿凡达的时候，感叹着他们接口的统一，和获取知识的便利性。有时候在想，现在很多企业所做的工作，不就是要提供这类服务吗。设想一下，我们有一朵公有云，存储了用户的数据、逻辑关系，提供标准的通讯接口，然后大家各自开发丰富的展现逻辑，让云端变的丰富多彩。这次很荣幸能接到这个议题，谈谈我个人对这朵云的理解。

每个人心中都有自己的一朵云，在我设想中，应该存在这么一种公有服务，它能够帮助用户随时随地的获取自己的数据，与朋友交流，获取好友最新状态。在这服务之上，我们有这么一个平台，它能够给用户提供二次开发的接口，让开发者根据用户数据开发丰富的展现层，并且提供这些展现层的运行平台。

我们需要的云端服务

为了完成这个功能，我们需要什么准备？



云存储：提供用户数据的存储功能。让用户方便的获取自己的数据。

通讯系统：提供以 Mail，IM 为基础的通讯方式

通知系统：好友行为推送，能够把握好友最新动态，或者告知好友你在干什么。

在这三个基本服务之上，用户可以开发大量的运用。比如“音乐盒”用于在线播放云存储的 MP3，图片系统用于管理、分享，美化自己的照片……然而，用户开发完逻辑应用之后，需要机器运行这个运用。因此，第四个基本服务**运行平台**孕育而生，它提供所有云应用运行的基本资源，包括内存、CPU、操作系统等。

这四个基本要素构建成一个面向终端用户的操作系统平台（也就是我们的云），它能够随时被访问，通过浏览器或者手机的 App。满足用户在任意时刻团购，玩三国杀，看视频，听音乐等需求。为了方便开发者开发更多的应用，我们抽象一种编程模式，提供丰富的 SDK，加速运用的开发。由于云端服务，有很大一部分会被手机等嵌入式设备访问，于是需要各种平台的编程框架（android、IOS）。编程框架将更加关心业务逻辑，屏蔽分布式细节和运维问题。

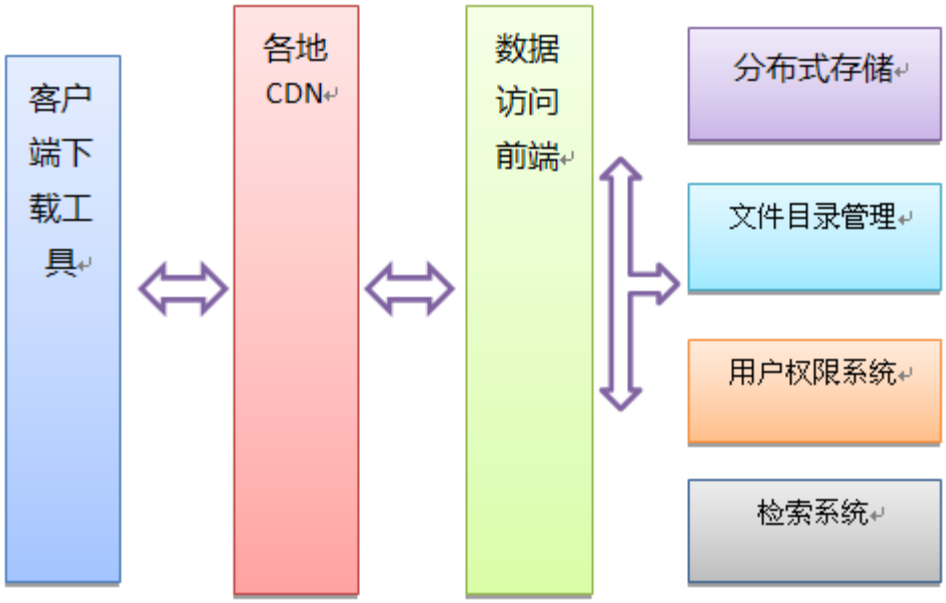
在满足这些开发便利性的前提下，为鼓励用户开发，提高 APP 质量和数目，需要一套良好的**收费系统**，帮助开发者更好的盈利。

围绕这这朵云，一点点的展开，发现想说的东西太多，今天我们就谈谈其中的两个核心架构：云端存储和运用运行平台（App Engine）。为什么要选这两个？因为云的核心就是存储和计算，其它都构建在存储和计算之上的基础服务和用户运用。

云端存储架构

云端存储主要是为了存储用户数据，方便用户访问。它涉及了三方面的技术

- 底层架构。包括：分布式存储、文件目录管理、用户权限系统
- 下载优化：各地 CDN 支持、客户端下载技术（P2P）
- 数据访问前端优化



底层架构设计的要点

首先我们比较一下跟传统离线存储的设计指标差异

1. **单个文件体积不大**

   存储互联网用户的数据，注定文件不会很大。我们只要支持 0-100G 左右的单文件大小即可。为什么用户文件会到 100G？因为我们要保证用户能分享高清电影。另外相对于海量的容量，如果单文件过小，那么海量空间也没啥意义。多媒体是促进磁盘发展的动力。

2. **文件数会很多**

   跟 GFS 不一样，云存储的文件数是海量的。因为每个人都会存储他们的文档、mp3、图片……这注定了单机保存全部文件的 node 是不可能的。

3. **需要目录管理**

   我们需要呈现传统操作系统类似的目录管理方式。另外根据云存储文件数量多的特点，我们要提供可靠的检索做文件管理。

4. **读写模式特殊性**

   用户对文件的访问模式是一次写入，多次读取，读取支持随机位置的读取（比如视频从中间开始播放）等。另外考虑在用户带宽条件下，100M 的文件也算是大文件了，我们要需要支持断点续传功能。另外，存在对单文件的高并发访问。

5. **检索和访问的实时性**

   用户上传的数据，在上传成功之后，就应该能访问到完整的数据。并且在检索的时候就

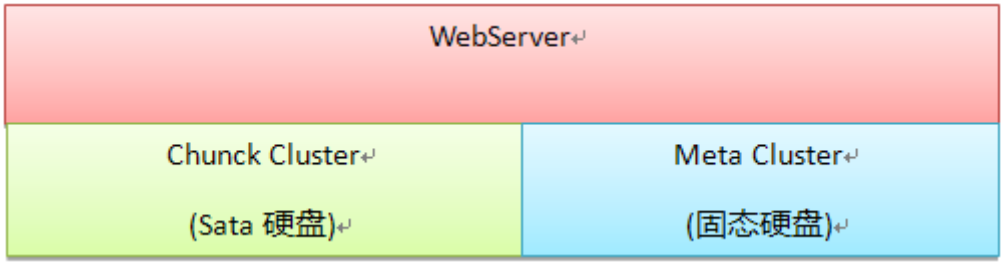
能够体现出来。因此不仅要求存储系统要求实时性，而且检索系统也有要求实时性。归纳一下，因为文件太小导致文件数过多，需要专门的目录存储；针对文件的访问模式，我们需要设计一个比较合理的文件格式；提升检索的实时性。

文件格式介绍

一个文件需要的存储数据：Meta 信息和数据块。Meta 信息存储这个文件的详细信息，包括文件名、大小、文件类型（doc 或者 mp3）、MD5、创建者、具体数据块的存放位置、数据块大小，以及该文件格式的版本信息等。数据块是真正存储的文件数据。

我们将一个完整的文件，物理切成多块。比如一个 1G 的文件，我们按照 1M 为块大小，切成 1024 块，然后将 1024 个块数据散列到 N 台机器中去。从而保证文件具备高并发的特点，而且也能够方便的为整个集群提供扩展能力。然后我们会将这 1024 个块的具体位置记录到文件的 meta 信息中，方便访问。

因此，我们需要一个逻辑文件的访问入口（WebServer），和存储这些数据块与 Meta 信息的集群 Chunk Cluster 和 Meta Cluster。



将一个不大的文件分散到各台机器上存储有什么好处？

1. 方便做负载均衡和集群扩容
2. 将热门文件的流量分散到各台机器上，使热门文件的高频访问对后端影响降低。

这个文件格式的设计，大家可能会觉得文件很大的话，Meta 信息因需要存储的块位置而导致体积过大。其实这个问题，可以通过二级索引块来解决。

存储架构的工作原理

如上图，WebServer 在接收 Http 请求的时候，会解析参数，然后根据 Meta Cluster 提供的 Meta 信息，读取相关块，返回给请求者。Chunck Cluster 和 Meta Cluster 的设计都是一样的，就是提供一套 NoSQL 系统，支持针对 Key（字符串） - value（二进制）的增删改查。但是

考虑到访问频率的不同，我们需要针对不同的硬件做单机的优化，比如廉价的 Sata 盘存放相对静止的数据，SSD 盘存放访问频率过高的数据。

NoSQL 集群不是我们这篇文件要简述的话题，有机会可以详谈。不过即使是分布式系统，我们也应注重模块的单机性能。因为如果我们的模块单机性能提高一倍，那么我们的集群规模就会下降一倍。在上万台机器中，节约的成本是非常可观的。我们如何衡量这个存储系统的单机引擎性能呢？方法很简单，如果一个单机模块，能够将网卡吞吐跑满或者磁盘顺序读写吞吐跑满，对于存储模块本身来说，可以了。

目录管理系统实现

海量文件的目录管理，很难。这里，我们采用一个分布式有序表的方式来解决，分布式有序表也是 NoSQL 的一种。它对存储的数据，提供基于字典序的游标查询。比如：我们将所有的用户文件名放入有序表中，该系统就会产生根据文件名排序的分布式数组，如下：

[/a.doc; /a/a.doc; /a/b.doc; /a/c.doc; /b.doc; /b/b/b.doc]

在执行 ls /a/命令的时候，我们会寻找/a/的游标得到/a/a.doc，接着我们开始遍历这个游标，直到不是/a/打头为止。如果该过程中碰到子目录，程序会会通过二分查找直接跳过子目录，从而防止遍历过多。如果数目过多，我们会展现 100 条，其它隐藏。目录管理，主要是给用户组织自己数据的时候用的，理论上，用户不会在一个目录下放太多的文件，即使太多，也没关系，我们就显示 100 条，然后提供下一页的按钮(*因为下一页的游标位置我们是知道的*)。

实时检索系统

讨论这个议题的时候，需要假设我们已经有一个传统的检索系统，然后想办法提高检索的实时性。我们设计一个内存索引，把用户新增的文件，对文件名切词后放到内存中检索，检索的结果参与最终的合并。每隔五分钟 merge 到传统检索系统中，然后释放内存。云存储，不像互联网网页，在 5 分钟之内，仅文件名的索引，数据量不可能太大，所以内存不会是瓶颈。进一步的，我们可以对文件的内容作检索，但是文件内容没有必要做到实时。

权限系统

用户权限系统，对于云存储来说，也是个用户文件，所以没什么特别的，只不过我们需要专门的缓存做访问优化。因为每一次读写请求，都要判断访问者是否有相关的权限。

CDN 技术

P2P 技术和 CDN 支持，主要是为了减少带宽成本而做的，在云存储这种数据量巨大的服务中，这两种技术，显的尤为重要。这两块是两个专题，我们在这里不多做介绍。不过这两个技术在解决热点问题效果比较好，但是海量文件并不是所有文件都放 CDN 的，因此有些工作，数据访问前端不得不做。

数据访问与客户端优化考虑

客户端访问速度差别，是我们要考虑的问题。如果是内部的访问，带宽可以保证是 1000M 以上，但是面向互联网用户，各种各样的带宽需求都有，比如 GPRS、3G、ADSL，从 20k-16M 不等。这就要求我们的前端技术，在处理这些请求下要工作的很好。另外我们还要考虑在正常服务下，网络带宽最小化，比如一个视频是 100 分钟，我们就应该保证 100 分钟内传完，满足正常播放，不能太快，因为太快，你不能保证用户有耐心看完，可能他就看 10 分钟，然后就关了，于是后面传输的带宽全浪费了。如果是用户下载，那么当然是越快越好。这些控制，我们都通过 WebServer 来实现。

WebServer 最主要的功能就是高并发支持，限速。再加上云存储的数据是海量的，传统的 Apache 做 WebServer 肯定不适合，这里我们采用异步的 WebSever 比如 lighttpd 或者 nginx，然后对客户端句柄进行速度控制。为了支持大文件的断点上传，我们需要有一个专门的客服端，能够将文件分块上传。Webserver 必须支持根据 md5 查询这个文件哪些块已经上传了，哪些没上传，从而通知客户端正常工作。

云存储有很多节约带宽的优化，比如上传文件的时候，先上传 md5，如果云端已经存在，就不需要上传了，这样可以做到大文件的秒传，节约网络带宽。另外它提供对外标准的 Http 协议，可以采用迅雷等 p2p 软件下载，从而提高访问速度，减少服务器带宽冲击。为了数据安全性，我们还得提供 https 协议的数据访问。

App Engine

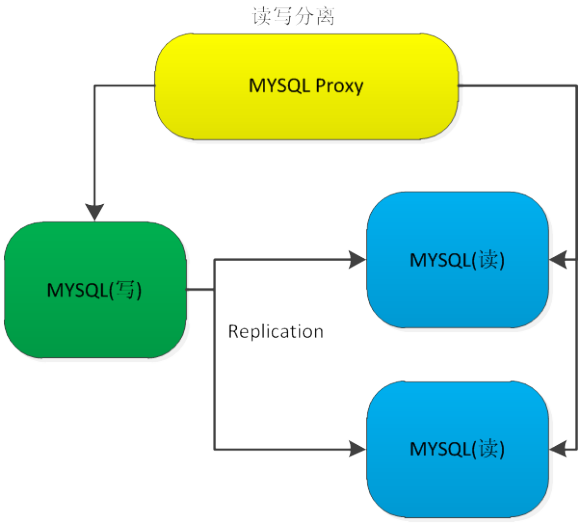
完成云存储的设计之后，我们需要一个开发平台，这个开发平台提供用户逻辑的运行环境。这环境包括

1. MYSQL 集群化管理
2. 离线任务的处理

3. PHP 的运行环境

MySQL 集群化管理

因为云存储没有提供关系数据的存储功能，为了降低用户的开发门槛，我们需要一个 MYSQL 的集群化来完成类似的功能。MYSQL 的集群化，主要是完成 MYSQL 读写分离和主从同步功能。



通过这种架构，保证了开发者不需要关心 MYSQL 的数据故障等问题。因为 MYSQL Proxy 会自动的进行主从切换和读写分离。这里我们要开发的就是解析 SQL 语句，完成相关的用户认证，并完成相关的后台转发、接收。

MYSQL 的集群化管理没有解决分布式的问题，这个地方我们认为不需要解决。因为互联网在线业务类的关系数据不会太大，大的数据都放到云存储里面了，数据库只存索引。还有，数据库的分库分表也相对成熟，索引数据也很难快速膨胀。如果用户有对分布式索引的需求，可以考虑前面我们谈到的有序表。

离线任务处理

离线任务处理主要解决，用户需要做大量的 cpu 密集型的工作，包括图片转化，视频转化等。我们这里采用了一套消息队列的方式进行离线处理。用户将处理请求扔给消息队列，执行机获取消息队列的消息之后，会执行相关的的用户代码。

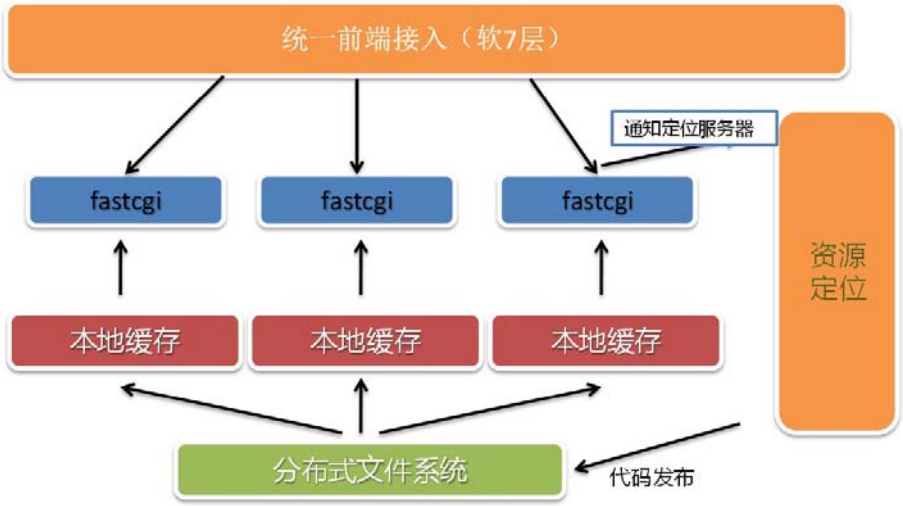
php 运行环境

PHP 运行环境，主要解决 PHP 的分布式化问题。云平台上跑的服务，千奇百怪，可能因为没有流量，只用到实际机器的千分之一，也可能拥有巨额流量，需要上百台机器支持。当然大部分服务没有什么流量。对于传统的虚拟化来说，1 台机器能虚拟化成 32 台，已经慢的不行。这样，一台物理机只能部署 32 个 app 运用，对于基础架构来说是不可接受的。因为互联网上的云端运用，会急剧膨胀，所以我们需要一种新的虚拟化架构，能将机器的粒度切的更细。

这里我们使用 PHP 为开发语言展示一个轻量级的虚拟化技术。我们通过轻量级虚拟化技术，为每个用户分配一组 FAST CGI 进程资源，通过 Web 端的调度，将请求引到各自的 FAST CGI 进程组中处理。这样一台机器能启动多少个进程，我们就能虚拟化多少份。

如果网站流量很大，单机处理不了，该如何解决？我们是通过 FAST CGI 进程个数来调度的，单机资源不够的情况下，我们会在多台机器上分配进程，组成一个 FAST CGI 组，然后通知 Web 端，这个网站的请求可以分流到哪些 FAST CGI 中去。我们会有一个总控的 Master 来观察各台机器的负载，从而判断是否要迁移 FAST CGI 进程。FAST CGI 进程的迁移是简单的，这台机器 KILL，在另外一台机器重启即可。

这个是我们 PHP 执行环境的架构。



架构依赖与资源定位服务。资源定位通知前端接入，哪些机器负载还行，可以引流，哪些已经故障，或者压力过大，不能引流。当 A.baidu.com 的流量过来的时候，前端接入会解析域名，并且根据资源定位获取的数据（本地有缓存），分发到对应的某台机器的 FAST CGI 端口上，执行 PHP 代码后返回。因为 FAST CGI 读取的用户代码存储在网络文件系统中，所以前端接入无论选择哪台 FAST CGI 都能够有效的做处理。在实际过程中，我们发现网络文件系统

对性能，尤其是 HTML 的访问性能影响很大，因此我们对每台机器做了单机缓存。不过 Cache 失效是个非常难解决的问题，我们这里采用的方法是一但文件发生修改，资源定位会通知所有客户机的该文件缓存失效，而且更新必须走同一的入口。由于我们只存储代码，效果还可以。有了分布式的网络文件系统，用户代码更新也变得异常简单。只要更新完成，通知缓存失效即可。

另外一个问题，如果运行平台跑大量的垃圾网站，比如很多一天只有少量请求的网站。用轻量级虚拟化，即使 1 台机器切出 2000 千份资源，还是很浪费的。对于这种运用，我们采用了 FAST CGI 复用，即很多小网站的请求都落到 1 个 FAST CGI 上，然后 FAST CGI 根据目前处理的网站，来获取相关的配额控制。真正做到资源消耗跟访问量成正比，没访问没成本。这里可能用 CGI 更好点，不过为了架构统一下，这点优化不算麻烦。

小结

简单介绍了 App Engine 所具有的能力，一个 PHP 的轻量级虚拟化，能够将 1 台机器虚拟成万分之一，也能将万台机器合成 1 个大的虚拟环境。实现从万分之一到万倍计算资源的渐进分配。它成功解决了一个网站，从小到大的计算能力的无限扩展问题。

也介绍了云存储所具备的能力，它支持海量的数据存储，成功解决了一个网站，从小到大的存储能力无限扩展问题。

整个云端运用，就是基于强悍可伸缩的计算能力和存储能力之下构建的网站。我们真正做的就是逻辑开发，和各个终端下的特殊展现形式。用这套架构，实现一个视频分享网站，电子书阅读网站是容易的。减少了云端应用的开发门槛，整个云端产品也将丰富多彩。它们都用统一的架构，计算和存储分离，程序开发无状态化，持久性存储放在云存储中。

有给力的基础架构，云端运用随手拈来。目前在百度公司内部，这种架构已经有成型的运用，它极大的提高了应用服务的开发效率，降低服务运维成本和开发人员的技术门槛。百度内部云平台迁移了大量的在线服务，有关百度的最新进展和数据，大家可以参考《QCon北京 2011 全球企业开发大会》公布的一些关于BAE的资料。

相关内容：

- 关于ESB实施的几点建议、云计算七问七答
- 云架构尚未经过深思熟虑、为云应用重构系统
- 白话SOA

特别专题

Trunk.ly CTO 董洵谈架构

作者 胡键

【编者按】：本期的架构师杂志我们请到了 Trunk.ly 的 CTO 董洵（Alex Dong，@alexdong）来谈谈他眼中的架构和架构师。

InfoQ：你好，Alex。能向我们的读者介绍一下你自己和目前正在从事的工作吗？

Alex：大家好，我是董洵，目前是 Trunk.ly 网站的 CTO，它是一个专门提供社会化书签服务的站点。由于目前公司的主要员工只有两人，因此，即便我是公司的 CTO，我仍然会每天花很多时候编写代码。就网站后端来讲，目前主要做的是搜索；前端则进行一些站点 UI 的设计和开发。就像大多数创业公司一样，我们非常关注我们的功能特点、站点流量，而且会经常跟投资人打交道。至于架构师这个头衔，其实谈不上，只不过做架构是我们平常工作的一部分吧。在这之前，也曾经用过.net 之类的技术做过企业项目。

【编者注：如果想进一步了解 Alex，读者可以访问他的个人站点以及《程序员》杂志对他的访谈。关于 Trunk.ly 域名的含义，@36 氪上的这篇文章中有这样一段描述：我问 @alexdong 为什么用 trunk.ly 这个域名，他说："是个一语双关. 在航海时代，每个周游世界的人随身要带一个箱子,这个箱子就叫 trunk。现在英联邦国家仍然叫后备箱 trunk。我们就是一个链接的百宝箱。"】

InfoQ：作为互联网的创业者，在同行业中有哪些站点的架构会引起你的注意，让你觉得值得钦佩？

Alex：这一点怎么说呢，像 Google。就我个人讲，我会花些时间（虽然我也没有太多的空闲时间）去读 Google Research 上发表的这些论文。像一些大的互联网公司，比方说 Google、Yahoo 和 Amazon 都有自己的 research 部门，他们经常会把自己解决某个问题的经验写成论文发表出来。对于一些架构上的实际问题，比方说 Timeline，这些公司会做一些科研实验，比较各种实现方式的优劣，最终把这些内容总结成文。我会比较多的看这些论文，从中了解这些公司如何解决架构中的某个问题，以及这些问题都有哪些解决方案，以及每种解决方案的好坏。从网站的架构来说，我会比较关注这些内容。除此之外，还有一些东西的架构我也

注，这也是我个人比较感兴趣的方面。比如说MongoDB、Linux或者是HyperTable，像这些新型的数据库、应用服务器等，我也花了不少时间去研究了解它们的架构是如何演进的。

就钦佩这个词来讲，可能有点不太准确。在我看来，它更多的是一种静态的感觉。比方说，当你看到大概有 5 年或 10 年历史的一个架构，第一眼看上去可能会觉得“了不起，想得真是很全面。”，这时你会产生出一种钦佩的感觉。但对我来讲，像MongoDB，在它还是 1.0 的时候，我就在自己的系统中用它们。更多的时候，我看到了它发展的轨迹。在这种情况下，说钦佩可能不太合适。更多的一种感觉是他们做的这些事情很有意思，很有特点。这也是为什么我会在邮件中提到，相比起架构本身，我会更关注架构背后的人，觉得他们更有意思。通过这些，你可以看到，同样一个问题，由不同的人来解决，最后的解决方案会特别的不一样，而且对应于它的社区，你会发现用这些解决方案的人也特别的不一样。所以这些事情是最令我感兴趣的。

InfoQ：是的，架构本身也是动态发展而来的。人本身有其自身的缺陷，很难设计出来一种面面俱到，一劳永逸的架构。就架构决策而言，应该是架构师在当时环境所能做出的最好决策了。随着时间的发展，周围的环境也会发生变化，这时就需要他来根据环境进行调整。

Alex：没错，这也是我觉得一位好的架构师，他真正的价值不在于从第一天开始就拿出一个蓝图，它有多漂亮、多干净。我认为好的架构师首先应该是知识面比较宽广，需要清楚有哪些选择，每个选择在解决当前问题的同时还会带来哪些影响，也就是每种解决方案好的一面和坏的一面。其次，当遇到架构上的问题时，能够通盘的考虑，发现问题的本质，进而提出自己的解决办法。比如像Linus，他实现的Git其实已经融合了他对于分布式开放源代码项目的管理以及协作方式的思考和理念。

InfoQ：那么做架构决策时，就有点像权衡各方面的因素，从而使架构达到一个比较好的状态？

Alex：说达到一个比较好的状态可能有点欠妥。我可以举一个反面的例子。我之所以说它反面，并不是说这个抉择或决定本身不好，而是说它有不可忽视或是很致命的缺点。像Python、Ruby和V8 Engine它们都使用了GIL（Global Interpreter Lock），它的问题在于若是我用Python写一个计算密集型的多线程程序，它的性能可能会比不使用多线程还要差。这就是它的运行环境或者说编程语言本身设计理念中埋下的引子。像Python之所以用GIL，是因为Van Rossum一开始最主要的设计目标是要非常容易的写C Extension，即写一个C模块，在Python中可以很容易的去调用它。有了这个锁之后，使得写C模块变得非常的容易。假如没有这种锁，要想实现从C调用Python或是从Python调用C，就没法做。在我看来，这是Python为了实现这种设计目标而不得不做出的极大牺牲。而这一点，就目前看来，其实还是非常致命的，对我们

自己应用的影响也很大。

我举这个例子的原因在于，权衡这个词在很多时候看上去很优美，没有负面感觉。可就对我个人来说，很多架构的决定，包括我现在用的很多软件，权衡这个词实际上有着很大的负面感觉，或者说你接受它【架构决策】时不得不承受很大的痛苦。而且很多时候是现实情况下不得不接受，或者说是一种对现实的妥协。因为，就当前情况而言，面对各式各样的约束条件，已经没有其他更好的选择了。假设我有充足的时间和资源，我可能会要求做得更好。

InfoQ：的确，每种工具或架构有其本身适应的环境或者说是前提条件。假设你当前应用的环境跟这个前提或环境有冲突时，就会造成你刚才所说的那种在应用上非常痛苦的一面。

Alex：你说得非常的准确，就是这种感觉。每个架构决策就会使架构僵硬一点，将你推向某个极端或某条路。当然，实在不行的时候，我就会必须将某些东西扔掉。我们做Trunk.ly也是沿路扔了不少东西，而且每次也都是在实在没有办法不得不这样做的情况下采取的这项行动。

InfoQ：这些优秀的架构给你的工作带来的怎样的影响？

Alex：对于这个问题，我想通过《设计模式》来谈一谈。我读这本书大约是在 8、9 年前的**事情了，当时的第一感觉是，这些模式设计得真是不错，真的很好。由于当时我用C++/C#**的机会比较多，在之后的几年里，我都尽量地去使用它们。可后来我认识到，这些模式其实是只是一种规则。我觉得最理想的，作为架构师，你在读这些规则或者是在用这些规则的时候，要知道这些规则之后的规则是什么。然后将这些背后的规则去用到自己的工作中，而不是单单生搬硬套，为用而用。

也就是说，我希望架构师能够理解这些规则之所以呈现出现在这个样子的原因。一名优秀的架构师，当然是要多看别人设计的架构，但是在看的同时，需要了解到使其架构成型的现实的限制和现实的原因。除去你看到的架构本身，你需要看到一种动态的架构，知道其背后的推理和它的思维。这样，等遇到问题时，就可以应用这些思维方式，而不是简单的应用架构。以Trunk.ly的后台搜索引擎的存储为例，到目前为止总共经历了 4 次大的变更，从最初的MySQL，到Sphinx，到Lucene和Solar，再到目前的HyperTable。每次修改都是因为我们知道我们遇到了哪些问题，又因为知道业界都有哪些解决方案，每个解决方案的优缺点是什么，最后有针对性的进行抉择和行动。我举这个例子想说明，正是因为我关注业界目前的动态，以及每种解决方案的优缺点，这样，当我遇到问题时，我就能想得起来去用它们。而作为架构师，是需要通过这种方式去建立起自己的工具箱，而不是单单让自己的手中就握着个锤子。

InfoQ：前面你曾提到相比起架构本身，你对于导致当前架构成型背后的人和故事更感兴趣。

能详细谈谈吗？

Alex：我觉得这是件挺有意思的事情。当你做架构做到某一天，就会遇到所谓“治理（Governance）”的挑战。尤其对于开源软件，在这样一种松散的组织结构，谁也不能将自己的设计理念强迫别人接受的条件下，要保证设计目标的达成、版本、特性不断地推陈出新，这时所面临的挑战和协调的艺术，要远比企业内实现同样目的要难得多。大部分的开源软件，到最后都会有一个所谓的“善意的独裁者（benevolence dictator）”，他们来决定特别艰难的决定该如何做。但是，不同的人做“独裁”决定的时候会特别的不一样。

像Linux社区，这个“独裁者”会要求你在邮件中把自己的设计理念讲得特别清楚，然后他会通过邮件来进行讨论。但像Python就不一样，它一开始就确定了一些理念。作为社区，如果你要贡献源代码就必须承认它们，否则就不会被接受。还有一些开源软件则采用了“不同意就fork”的道路，假设一两年之后，他们觉得这些fork不错，这些fork还有可能会重新被接受。比如Python社区就有几个著名的fork，如pypy，它是用Python来写Python，现在的性能也越来越好了。

通过看这些开源社区的意见领袖如何用“非强制”的手段来保证架构的一致性，消除意见分歧，我觉得是一件特有意思的事情。这就类似在现场看一个直播，对我来讲这比起架构本身成为什么样子要有意思得多。而且在这个过程中，你也可以了解这些软件解决方案诞生的全过程，大家对于问题/特性应用场景的讨论和投票，给你带来一种参与感，这是特别过瘾的一件事情。

InfoQ：刚才说了那么多其他人的故事，现在是不是可以说一下你们自己的故事？谈谈Trunk.ly经历了几个阶段，每个阶段都有哪些，未来它准备朝什么方向发展？

Alex：做创业公司是有两个阶段的。第一个阶段是摸索阶段，在这个阶段，我还不清楚我的商业模式是否行得通，不知道今天的用户是否就是给我带来收益的那些用户。这个阶段是非常重要的，英文叫做“product/market fit”，在之前的所有架构都不是为了架构本身，也不是为了站点的Scale，而是为了能让我尽快的去做实验，更快地验证我的假设。

InfoQ：更像一个原型？

Alex：是的，但它不是那种可抛弃的原型，因为我没有这个奢侈去把它抛弃重头再来。我们在去年 12 月份Trunk.ly上线的时候，我们对其的定位是一个书签服务，但它是一种透明的书签服务。跟传统的delicious不一样的是，你不需要显式地给它去打标签，它会自动收集你在Facebook、Twitter和Delicious上共享的链接，把它们弄进去进行全文检索。这样，当你想找以前的链接时，你可以通过我们的搜索引擎去找到它。所以，在第一阶段时，为了满足这个

商业目标，我们就拿Django和MySQL快速地把它给实现了。后来数据库就有点顶不住了，于是就换成了MongoDB，同时有一部分功能就采用了异步的方式实现。

接下来，我们发现之前的商业目标虽然不错，但是这个市场的价值并不是特别的大。然后，通过日志和分析用户的行为特征，我们发现有些用户经常会去看别人共享了哪些链接，所以我们很快就增加一个Timeline的功能，这样你在一个地方就能看到你关注的人每天都共享了哪些链接，这样用户就不用跑到每个人那里去看了。这个时候，关于网站的商业目标和定位就有了一点点的变化。这个时候，我前面说的那些网站的论文提供了一些帮助。而且，这个时候大概每周都有百分之二三十的用户增长，比起原来要快多了。而且，很多人会邀请他的朋友来注册Trunk.ly。你要是在Twitter上搜索trunkly，你会看到每天有四五十条这样的信息，这就是我们的第二个阶段。

第三个阶段就是我们发现这个市场还是不够大。因为，要想把Trunk.ly做成一个让人们每天都来的地方，这样做的市场开销特别大，而且不是每个人都有这样的需要。所以，我们开始着手做Social Search。与Google这样把所有人的链接都搜索出来不同，我们的这个功能允许你搜你朋友的链接，这样搜出来的结果相比起前者来讲要更准确，而且跟你的相关性更强。我自己在测试这个功能的时候，也经常因此而“走神”，开始访问起那些链接的内容来。

在这个功能完成上线之后，我们还准备做的一件事情是“Topic”，即允许用户自己来建立自己的主题页面和链接。我们相信，从广告和利润分成来说，这对我们是一个非常大的市场。从我刚才说的这些内容，你可以看出Trunk.ly的整个发展过程。我们目前还处在“product/market fit”之前，我想如果一旦证明我们设想的商业模式是正确的话，可能架构还要再变，到时候可能就会更Scalable一些，而且架构挑战也就不一样了。

InfoQ：对于未来的互联网架构师，作为过来人，你有什么经验之谈是值得他们注意和避免的？

Alex：我想我上面讲的内容应该已经是我的经验之谈了。如果非要总结的话，那就是，视野放宽一点，不要手里拿着榔头就满眼尽看到钉子；另一点就是好的架构都需要一个过程，多关注一下架构背后的事情。这就是我能想到的经验之谈，也不是什么非常了不起的道理。

InfoQ：非常感谢你能抽空接受我的采访，谢谢！

Alex：不客气。

相关内容： Twitter，架构的变迁、理解REST软件架构、你是个软件架构师吗？、架构师修炼之道、虚拟研讨会：软件架构文档

推荐文章 | Articles

探索式测试的秘密

作者 季哥

前段时间的 QCon北京 2011 大会 里面有个探索式测试的分享，是 Erik Petersen 演讲（资料下载） 的，我由于一些原因，没有在现场向大师学习，之后发现他讲的非常好，效果也很好，引起了很多人对于探索式测试的兴趣。这里也感谢InfoQ的给力，我把了 Erik的PPT看了两遍，有非常深刻的体会。也确实在某些方面开导了我。自己本身学习探索式测试快一年有余了，也不乏和国外的ET的一些大师 (James Bach, CemKaner)进行交流。

这里我先抛一个问题出来：为什么产品提交到测试这边，我们的测试工程师可以发现很多开发或用户都发现不能的 Bug 呢？大家可能觉得这个原因肯定有很多，比如有这些：

1. 我们是测试工程师，我们是专门干这个的
2. 我们很好地理解了需求，我们使用了很多且很好的测试设计技术
3. 我们有较多时间进行测试执行
4. 开发人员或用户没有时间去进行测试
5. 开发人员或用户没有时间不知道怎么去进行测试
6. 我们的运气好，突然发现了这些好的 Bug

(当然还有很多其他的因素，这里就不列出所有的了)

那么再有一个问题出来 :为什么有些测试人员能发现其他测试人员不能发现的 Bug 呢？大家可能觉得这个原因肯定有很多，比如有这些：

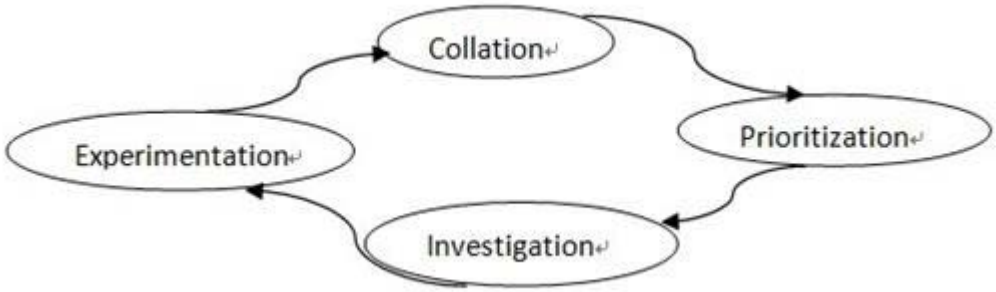
1. 其他测试人员对于业务需求不熟悉，该测试人员对于业务非常精
2. 其他测试人员没有很好的状态，注意力不够集中
3. 其他测试人员的测试经验较少，该测试人员测试经验非常丰富
4. 其他测试人员的运气不怎么好，该测试人员是突然灵感出来发现的

5. 其他测试人员的测试环境和测试数据不一样

(当然还有很多其他的因素，这里就不列出所有的了)

同样的问题还有很多，比如：为什么我们很多好的 bug 都不是通过我们已经写好的测试用例发现的呢？其实这里面我并不是想去搞清楚很多不同的原因，根据 80/20 原则，我只想关注一个主要的，那就是经验丰富的测试工程师是如何进行测试的，是如何进行探索式测试的，他们的思维过程到底是什么样的，和我们通常的测试有什么不一样的。

很多人都会问到底什么是探索式测试，也有很多人知道很多时候我们就是在做探索式测试(只是我们自己不知道而已)，不管怎样，我们都期望把很好的测试方法或手段传承下去，让新加入测试行业的同学都可以吸收这个武林秘籍。根据看 Erik 的 PPT，我这边大概抽象了下探索式测试的思维过程：



这个思维模型简称为 CPIE

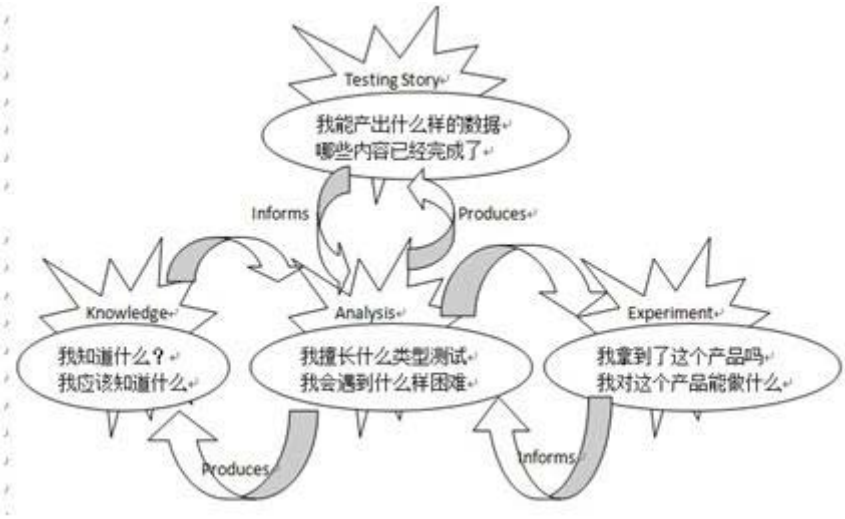
- 简要说明：Collation，这个 action 注意就是我们需要收集所有关于 SUT 的所有信息，去了解和理解。
- Prioritization，我们要对所有需要测试的任务或模块或特性进行优先级的划分，这里不说划分原则。
- Investigation，划分好后，就需要对应确定即将测试的任务进行仔细的分析并预测其可能输出的结果。
- Experimentation，这个 action 就是需要我们实际的去进行测试，看看我们的预测是否正确，我们的信息是否正确，就会变化到去影响 Collation 阶段。

这里面部分人看的出来，还是有点站在整个项目或产品测试的角度去进行测试，那实际上对于某个小的需求和功能，是如何进行探索式测试的呢？

这里面还需要强调的是探索式测试是一个测试方法，不是测试技术，大家都知道有很多测试

技术，特别是测试设计技术，这样的话，就意味着进行着探索式测试，就可以完全使用这些测试技术，可以融合这些测试技术。

下面来看下 ET 的大师 James Bach 是怎么来看待 ET 的思维过程的：



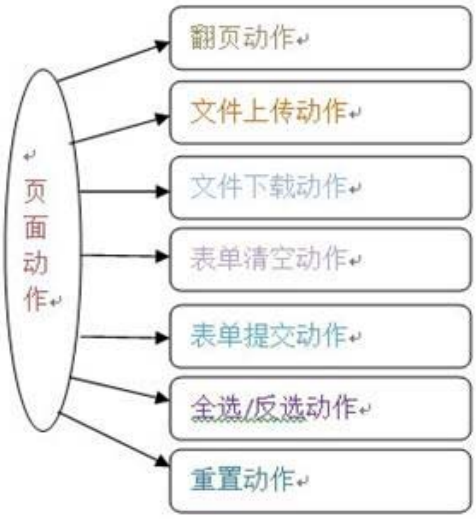
这里面可以看到和 Erik 的观点还是有很多类似的地方的，都强调 Experiment，也就是说之前我们做的再好的测试设计和用例，只要在测试执行的时候才知道好还是不好，还有没有更好的测试思路。同样可以发现这些都是一个循环的过程，ET 过程中，测试设计和测试执行是互相驱动和完善的过程，这也是和我们平时的 SBT(Script Based Testing)的最大区别。

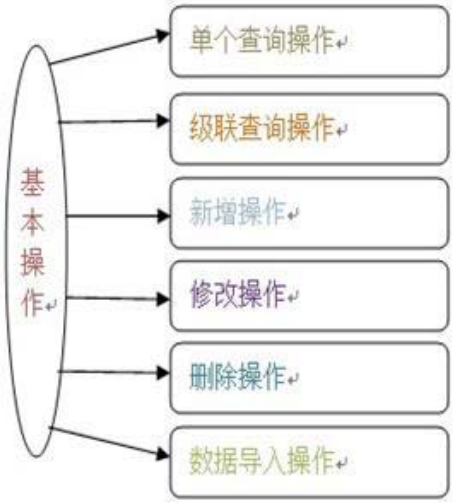
个人认为之所以优秀的测试人员能发现一些隐含比较深的 bug，主要有以下几个关键的因素：

1. 对于基本的测试设计技术的使用达到炉火纯青的地步
2. 对于错误猜测测试方法有一定的理解和应用
3. 对于开发思维习惯和异常的用户使用思路有一定的了解

这里面谈到了错误猜测方法，Erik 也提到了，这个方法在 ET 过程中使用得非常普遍，也许很多人会认为该方法很大程度上依赖于个人的测试经验积累，不错，但不意味着新的测试人员不能很好地使用该方法去进行 ET。这是又回到了模型的概念了，个人理解的模型有三个层面，如下：

1. 功能测试模型：一些常用功能的测试思路的大集合(增删改查，web 页面测试等)





2. 线下 bug 模型：基于线下测试发现的优秀 bug 抽象的模型(主要针对于常出现 bug 的地方，提出注意事项)

模型解释：

- 多线程创建，更新，删除某类数据，多线程方式包括同时打开多个页面或浏览器；使用工具多线程并发操作来校验数据操作的原子性
- 多浏览器或一个浏览器里多个 Tab 进行测试，考虑 cookie 里面的值的变化是否影响后续的操作；或使用场景探索模型去多角度校验页面信息和数据库数据的正确性

3. 线上故障模型：基于线上事故产生的原因进行抽象的模型(环境较复杂和特殊，完善我们的测试设计思路和创新出新的测试方法)

目前淘宝也在做这三方面的测试模型，以应用于 ET 的培训资料，使测试经验不丰富的同学也能够快速掌握错误猜测方法去进行 ET 测试，把这些模型完全掌握且能够应用得好，那我

们还怕我们的功能覆盖率不够全吗。

说到覆盖率 接下来说下 ET 在 Coverage 上是怎么考虑的。我们说的 Coverage 一般就是 Product coverage，同样也是这个被测产品的一部分。那么对于 Product coverage 又包括哪些方面的 coverage 呢？

第一个就是 **Structure**，也就是产品的一个因素，对于这个 Structural Coverage，我们到底是测试什么呢？我们到底要 cover 什么呢？我们要测试的就是这个产品是怎么构成的，我们要 cover 的就是构成这个产品的部 分。下面以打印机产品为例，看看 Structural Coverage 到底要考虑什么：

- 打印需要用到的文件
- 实现打印功能的代码模块
- 在这个模块里面的代码语句
- 在这个模块里面的代码分支

可以看到这个时候我们关注的是产品的内部结构。

第二个就是 **Function**，也是产品的一个因素，对于这个 Functional Coverage，我们到底是测试什么呢？我们到底要 cover 什么呢？我们要测试的就是这个产品能够做什么？我们要 cover 的就是这个产品做得什么 样。同样以打印机产品为例，看看 Functional Coverage 到底要考虑什么：

- 打印，页设置，打印预览
- 打印 range，打印复制，zoom
- 打印所有的，当前页，或指定的 range

可以看到这个时候我们关注的是产品的功能或特性。

第三个就是 **Data**，也是产品的一个因素，对于这个 Data Coverage，我们到底是测试什么呢？我们到底要 cover 什么呢？我们要测试的就是这个产品能够对数据方面有什么考虑？我们要 cover 的就是这个 产品能够处理什么样的数据。同样以打印机产品为例，看看 Data Coverage 到底要考虑什么：

- 打印文档的类型
- 文档里面的元素，文档的大小和结构

- 关于怎么打印的数据(比如 zoom factor; no. of copies)

可以看到这个时候我们关注的是产品使用过程中不同的数据处理。

第四个就是 **Platform**，也是产品的一个因素，对于这个 Platform Coverage，我们到底是测试什么呢？我们到底要 cover 什么呢？我们要测试的就是这个产品依赖什么才能使用？我们要 cover 的就是这个产品怎么 处理不同的依赖的。同样以打印机产品为例，看看 Platform Coverage 到底要考虑什么：

- 打印机，Spoolers，network behavior
- 计算机
- 操作系统
- 打印机驱动程序/设备

可以看到这个时候我们关注的是产品使用过程中不同的环境和依赖。

第五个就是 **Operation**，也是产品的一个因素，对于这个 Operations Coverage，我们到底是测试什么呢？我们到底要 cover 什么呢？我们要测试的就是这个产品怎么使用的？我们要 cover 的就是这个产品使用的步骤 是否合理/正确。同样以打印机产品为例，看看 Operations Coverage 到底要考虑什么：

- 默认情况下使用
- 真实环境下使用
- 真实的场景下使用
- 复杂的流程下使用

可以看到这个时候我们关注的是产品使用的场景(包括稳定性，可用性，安全性，可扩展性，性能，可安装性，兼容性，可测性，维护性，本地性等)。

第六个就是 **Time**，也是产品的一个因素，对于这个 Time Coverage，我们到底是测试什么呢？我们到底要 cover 什么呢？我们要测试的就是这个产品在什么时间情况下会受影响？我们要 cover 的就是这个产品在不同的时间下会表现什么样。同样以打印机产品为例，看看 Time Coverage 到底要考虑什么：

- 尝试在不同的网络或端口的速度使用

- 一个文档打印完，紧接着打印另一个文档，或隔很长时间再打印
- 尝试与时间相关的限制，比如使用 spooling, buffering, timeouts
- 尝试 hourly，daily，月底，或年底打印报告
- 尝试从不同的 2 个工作站同时打印

可以看到这个时候我们关注的是产品使用的时候是否受时间影响。

上面我们可以看到 ET 在考虑覆盖率上还是有一点自己独特的角度，但感觉不是很具体，无细节，对于某一个类型的 Coverage 需要做出全面的分析，还有一个就是这些不同类型的 coverage 会经常组合在一起来使用的。至于组合的策略，需要在实际项目过程才能体会。

原文链接：http://www.infoq.com/cn/articles/exploratory-testing-secret

相关内容：

- 阅读者（五）：测试驱动开发的艺术
- QCon北京 2011 大会精彩回顾和总结
- 颠覆者生存：互联网产品测试观点
- 窦涵之谈ATDD及CI的整合
- 基于静态依赖的测试

推荐文章 | Articles

项目的故事

作者 Olivier Mallassi 译者 侯伯薇

这是关于一个项目的故事，与其它项目相比，既不非常复杂，也不是很简单： 一个应用程序与数据库以及其它两个系统通信。这在技术和架构角度都是主流，而在管理角度则是标准情况： 所有工作都应该在昨天完成，但还有很多没有完成的。简而言之，“这很难！”（这是开发者经常表达的一种情绪，但是谁都不会太大声地把它喊出来。）

> 为此我们创建了团队。雇佣了四十位员工并指定了他们的角色。我们把团队分为小组，并在不同的小组之间设置了一种契约。每个小组负责应对特定类型的要求。这样就形成了要求的流程。指定的小组会承受很大压力，并成为瓶颈： 在上游小组中会创建要求，而下游的小组会等待这些要求。对于那些有压力的小组中的要求，重要的事情会变为紧急的事情。我们必须在紧急的事情之中做出选择，以解决最亟待解决的问题。我们经常需要在任务之间切换，最终，这导致流程变得很慢。
>
> 然后发布的最终日期临近了： 就在两个月之后。用户接受测试刚刚开始，但是不同组件之间冗长而痛苦的整合过程导致了测试的延迟。可能正是在团队之间创建的契约让整合变得如此复杂： 其中缺少一些必须的参数，日期格式不合适，只有部分错误代码能够得到解释……
>
> 在任何情况下，用户接受测试都会检测出很多的 bug，而开发团队来不及解决，这样就无法对系统进行彻底的测试。
>
> 因此我们增加了更多人力。我们设置专门的团队来解决 bug，在另一个地方的团队会完成开发，还有一个团队会整合不同的组件。但所有这些团队共享同样的代码，对某些代码做出的改变会影响其他团队所做出的修正。
>
> 接下来我们可以预测出这样的做法导致的结果： 开发者通宵加班，周末也要工作；最终日期被一再推迟，最初的工作任务范围发生了变更（减少了内容）；然而，多亏了计算机科学的奇迹，应用程序最终发布并运行了！

这是多年以前的事儿了。我认为这是“要走的路”，并且所有项目都以相同的方式管理。从那开始，我经历了各种不同的情况，读了很多资料，并与很多人做过相关的讨论。现在我能够从不同的思想角度来看待这个故事。

创建IT系统是一种非常复杂的机制，其中不仅混合了技术、架构技能，还有管理以及与人相关的技能。在这两个领域有多种文化，但是关于构架系统的管理和人的部分，我认为Tom De Marco是我最喜欢的作者之一。我还记得他出版的两本书。在第一篇无标题的“软件工程：关于谁的时代来临和消失的想法”中，De Marco讨论了对软件项目的控制（的幻想）。在第二篇“放松（Slack）”中，De Marco研究了管理学中的经典问题，并说明更多的放松能够如何帮助组织具有更强的适应性，最终获得更高的效率。

但是，带着这两篇文章中的想法，让我们回到故事中，看看如何才能避免很多典型的问题。

没有优先级，或者是著名的“我们全都需要”

Tom Demarco 向我们展示了一种方法来避免整体上陷入困境，也就是“每项功能都非常重要”的项目：

例如，你和团队的领导说，我时刻记着完成时间，但是我不会与你分享这个信息。当有一天我告诉你项目会在一周内完成时，你需要准备好打包，并把你所得到的东西作为最终产品发布。你的工作就是持续地为项目工作，按照相对值的顺序向其中添加内容，并在过程中持续地完成集成、编写文档以及验收测试工作。

简言之： 工作的目的是第二天就可以发布

如果不考虑实际的组织和技术上的问题，你需要有意识地对软件进行持续构建。团队之间的合同责任很有限，从而无法在技术上或者任务上组织它们，而只能在业务特性上做到这一点。从技术的角度，每个团队都是这样，要负责让完整的功能可以很好地运行。从管理的角度，经理和业务人员需要作出选择： 什么是必须需要的特性。根据我自己的经验，你在需要符合交付日期的环境中工作的时间越长，这种特性团队和组织就会对你起到更大的帮助。

管理压力和恐惧文化

我们在这里还要看看 De Marco 关于恐惧文化和它的特征是怎么说的：

“……在具有恐惧文化的组织中，会具有如下特征：

- 说特定的问题是不安全的（例如，我很怀疑是否能够达到这个限额）。

- 目标设定得非常高，看起来没有完成的可能。
- 允许权利超越一般的常识……”

当我想到恐惧管理的时候，就会想象到一种专横的身体语言，他会在位置上对协作者大喊大叫，会用拳头敲桌子……这是非常具有讽刺意义的样子。这看起来像是在背后说人坏话，但是我们不得不承认，人们经常会处于这样的压力之下，难于提出警告，设定截止日期而不考虑团队是否能够承受，最终，团队会处于经理根据他们自己的地位所做出的提交压力之下。

当然，了解问题只是第一步。我们怎么才能解决呢？ 当经理不了解风险所在，并且拒绝接受那些不可规避的风险，我们应该怎么办呢： 你需要选择，按优先级排序，并协商范围，以确保截止期限，或者把它延后。

这个任务绝对不是个简单的任务，我现在所能够给出的最佳答案就是 backlog 和燃尽图的组合。依我所见，在以下各种情况下，这会有些好处：

1. 把所有任务集合在一起（技术上的、功能上的任务等等）。当然，我们可以通过用例或者特性来组织和统一这些任务。
2. 把任务与所有项目参与者共享。换句话说，让更多人知道哪些工作是必须要完成的。
3. 显示出信心，并制定切实可行的截止日期，从而让项目经理可以有效地排定任务的优先级。
4. 显示所有添加的任务，那可能会推迟最初制定的截止日期。

在上线前的几周，当前组织的压力比较大，因此我们决定要增加人手

Brook 在 1975（我还没有出生）年声明了一条规则：

“向已经延迟了的软件项目中添加人手，只会使其延迟更严重。”

我们都已经对此深有体会。但是我们不得不承认，一般我们还是想要通过增加人力来保证在截止日期之前发布，而不是改变最初定义的范围，并保持最优化和合适的团队规模。Brooks 对他的规则做出解释，其中有两个主要的观点。 第一点关注于新的团队成员，我们需要对他们进行培训，那会耗费现有人员的工作时间。第二点是一个“神话”，让我们相信开发任务可以任意分解，而不需要考虑工作中很多部分都需要智慧，并且还需要所有开发者之间必要的沟通。此外，我们还能发现更多与组织开发相关的困难，并且需要在所有开发人员之间共享同样的代码。这么多的细节会降低团队的生产力。

Tom de Marco 为我们做出了他对 Brooks 的规则的理解，他称之为“人浮于事（overstaffing）”。

“按时完成并非是最重要的。重要的是看起来你已经做出最大的努力来按时完成。在这个“吃得少跑得快”的时候，对于你来说，使用少数（优化的）人员来运转项目是很不安全的。”

请再次读一下，这次慢一些。你不认为这是一种很有趣的观点吗？ De Marco 所建议的，或者说我的理解是，我们之所以不增加人力，是因为我们认为那会更快更好地完成工作。我们添加人力只是作为一旦失败时的借口。就像小孩子喜欢说的：“ 那不是我的错，我已经尽力了。”

这是多么自然的反应啊！ 想要改变这种行为，首先需要改变我们的教育方式（并学着从失败和错误中吸取教训），并且组建一些组织，让失败成为其中的一种选择（当然不要太多）.....

“一切都一直在失败”。你为什么不积极处理，而只是忽略它呢？

和通常情况一样，批评很容易，而艺术地处理则很难。那样，我们会注意到相同的错误一次又一次发生，而很少试一下其他方法（当然，这些方法也会存在其他限制）。“风险管理是为失败做计划的方法”（Slack, Tom de Marco)，并且这可能正是我们擅长之处。 Werner Vogels 说过：“一切都一直在失败”。

他们所要表述的是： 我们很难避免失败。就像往山上滚石头的人一样，如果想要爬得更高，走得更远，就必须接受石头会落下的事实，那是通往成功的必经之路。所以拥抱它、处理它，并从中学习吧。

Tom De Marco 告诉我们，想要管理风险，首先需要把它们识别出来，然后对其进行监控，设置指示器，当失败发生的时候会为我们提出警告。有时，我们需要找到其他方法。有些人需要培训。我们会开发软件的并行版本，这样，在做决定的最后时刻，我们就能够选择最适合的解决方案（精益管理把这叫做“基于设计设置（set based design）”原则）。

但是风险管理不仅仅是计划。从架构的角度来看，风险管理当然还意味着可依赖性的管理：从灾难中恢复.....但这种风险管理的方法意味着要为我们的系统制定架构——与业务部门的人员紧密合作——从而管理并拥抱所有这些错误。换句话说，就是尽可能地预测我们架构中可能发生的错误情况：

- 如何管理一种降级的模式，从而在整个系统的某个部分不可用，或者访问者的数量超出预期时让系统仍然可用？
- 在发生错误的时候，需要执行哪些过程（如果需要手工过程的话）来终止业务处理？

- 想要完成业务处理，需要哪些命令信息？
- 警告机制是什么样的，从而可以根据最终用户的需要提前激活，通知他发生了错误，并帮助他恰当地终止正在处理的工作。
- ...

在现存的系统中，我们需要让它发展（而不是变革），从而确保这些已经检测出来的错误情况都被彻底修正。

但让我们现实一些。如果你是成本驱动的，那么你会发现所有非业务的需求都没有用。但是，最终我们对于应用程序的信心不正是就它管理错误的能力（弹性和可靠性）吗？绝对不是其它标准吗。

结论： 那又怎么样呢？

还有另外一个故事，属于另一个项目，既不非常复杂，也不是很简单。和其它所有项目的情况一样，其中有很多需要做的工作。开始时，IT 和市场团队一起定义了清晰的路线图。

- “我们希望在四个月内发布新平台，其中会实现最简化的市场特性。对于访问量，我们期望每小时会有 3000 次，而 80%的访问者会使用新特性。”这是市场团队所说的。
- “OK。”CTO 说。“我们需要添加一些非功能性的特性来限制对平台的访问，以防我们的预测出现错误。在这些额外的特性中也会有一些技术风险，我们需要尽快确认。可能我们需要在下一次发布中看是否能够完成这些内容。”
- “没问题。尽快告诉我们关于这些风险的信息。在接下来的月份中，我们就会需要这些新特性……”
- ...

随着特性团队负责实现完整的用户故事、修正 bug 并部署他们的代码，平台变得越来越大。

当然，IT 也遇到了技术问题。业务和 IT 部门需要一起来对路线图进行修正。最后期限已经到了，而有些特性被推迟实现了。那些都是优先级最低的特性。

当然，市场预测也不怎么好。CTO 向平台添加的非功能特性（例如，记录所有用户行为的日志，或者当达到已知平台限制的时候限制访问）在发布平台的过程中非常有用，帮助市场团队改进了预测。用户愿意使用这个新平台，并且信任它。

在开放的空间中，墙上也充满了信息。燃尽图和累计流程图显示了团队的速度和问题，以及

下次发布的目标……

在这个项目中，还雇佣了一位技术专家： 那是 CTO 不想放过的机会。

不管怎样，项目进行良好，并且还在运行中。技术人员们对自己所做的工作感到自豪……

原文链接：http://www.infoq.com/cn/articles/mallassi-project-story

相关内容：

- 云应用的两个案例
- InfoQ中文站 2011 年技术趋势展望
- 云计算那点事儿
- 基于Adobe AIR的社交应用开发及案例研究
- FourSquare经历两次宕机

推荐文章 | Articles

虚拟研讨会：Node.js 生态系统之框架、库、最佳实践

作者 Dionysios G. Synodinos 译者 王丽娟

Node.js是一个服务器端框架，基于 Google的V8 JavaScript引擎 创建，旨在利用事件触发、非阻塞的I/ O帮助开发人员构建高度可伸缩的网络程序。目前很多流行的第三方库和框架都使用了Node.js， InfoQ联系了其中几个的创建者， 与他们展开了一次虚拟研讨会。

Node.js的创建者Ryan Dahl在JSConf 2010 上做了演讲，简单介绍了 什么是Node.js（PDF） ：

- 服务器端的 JavaScript
- 基于 Google 的 V8 创建
- 事件触发、非阻塞的 I/O。 类似于 EventMachine 或 Twisted。
- CommonJS 模块系统。
- 有 8000 行 C/C++代码， 2000 行 Javascript 代码，14 个贡献者。

Node.js最近越来越受关注，Yahoo! Mail的首席工程师Peter Griess也提到， 他们公司正在研究如何在产品中使用Node.js。

目前 在GitHub上，大约有三百个项目都与Node.js有关 ，InfoQ联系了其中几个最受欢迎项目的创建者， 与他们展开了此次讨论。

参与者包括：

- 来自 Express的TJ Holowaychuk， Express是受Sinatra影响的Node.js Web开发框架。（译注：Sinatra是用Ruby编写的开源Web应用框架、 领域特定语言。）
- Socket.IO的Guillermo Rauch，Socket. IO是一个简单的HTTP套接字接口实现及服务器。
- 来自 Geddy的Matthew Eernisse，Geddy是个Node. js的Web开发框架，类似于Merb、Rails、

Pylons、 Django等框架。

- node-xmpp的Astro，node- xmpp是遵循XMPP协议的Node.js库。
- StackVM的Peteris Krumins和James Halliday，StackVM是一家初创公司， 正尝试用Node.js来简化虚拟机在Web上的使用。

InfoQ：能向大家简单介绍一下你们的项目么？ 项目试图解决什么问题？又是如何去做的呢？

> **TJ（Express）：**正如你们所了解的，Express 很大程度上受到了 Sinatra 的影响，最初主要是想让开发人员在新的平台上自如使用新的语法。不过我们在 1.x 版本做了很多改进，Express 现在利用了流行的中间件框架 Connect（ 我是共同编写者），并移除了一些不适合项目的依赖。
>
> 有些 Node 框架只关注自身，但大部分还是想成为“最重要的”库。我期望 Express 是可选的，不会带来什么局限。

> **Guillermo（Socket.IO）：**Socket. IO 提供了一个简单的 API ,可以抽象出各个浏览器在 HTTP 传输实现上的不同，借此让实时 JavaScript 应用的构建成为可能。

> **Matthew（Geddy）：**Geddy 是个全 JavaScript 堆栈的 MVC Web 框架。Geddy 想让开发人员很容易地用 JavaScript 编写复杂的 Web 应用、在客户端和服务器之间共享代码。Geddy 使用了现有 MVC 框架中的常见模式， 比如 Rails 和 Django 中用控制器/行为、模型、模板化的函数。不过 Geddy 中的这些功能都是用 JavaScript 完成的，所以模型、验证、模板之类的内容也都可以在浏览器中使用。
>
> Geddy 有简单、基于资源的路由机制，还有准确的内容协商， 所以用它来开发传输结构化数据的轻量 Web Service 会超级简单。

> **Astro（node-xmpp）：**我的目标是创建一个能在 Node.js 环境中容易使用的 XMPP 库。先前的库以适用于浏览器的 Strophe.js 为基础，不符合 Node.js 的约定；也没有利用 EventEmitter 和支持 SRV 的 DNS 解析器，最重要的是，它既不支持 XMPP 客户端，也不支持组件连接。
>
> Peteris 和 James（StackVM）：我们的项目叫 StackVM，它可以借助 HTML 和 JavaScript 让虚拟机在浏览器上运 行。StackVM 显示了虚拟机不绑定到桌面时可以进行的操作。比如说，用户可以在他们的网站里嵌入虚拟机、与合作者共享工作内容、为应用创建在

线的产品演示，还可以创建以前不可能实现的 Mashup。

InfoQ：你们项目里的主要模块是什么？它们怎样和其他模块、 Node.js 进行交互？

TJ（Express）: Connect 中间件框架在 Expre ss 1.x 里发挥了重要作用，以前称为“插件” 的组件现在都是 Connect“中间件”。这种抽象意味着社区可以在更短的时间周期内借助所有通用的内容生成强大的框架。Express 1.0 测试版的核心组件和功能有：

- 强大的路由
- 视图系统，包括对局部视图和集合视图的支持
- 基于环境的配置
- 基于会话的持久、迅速的通知
- 内容协商
- 响应工具
- 重定向工具
- 能快速生成应用骨架

Guillermo（Socket.IO）: Socket. IO 包括一个在浏览器上运行的 JavaScript 客户端，还有一个在服务器（Socket.IO-node） 上运行的 JavaScript 组件。

Matthew（Geddy）: Geddy 是个 MVC 框架。你可以在“控制器”上定义函数，把 URL 映射到动作上。 还可以创建“模型”，定义控制器中要使用的对象。最后，你可以通过渲染“视图”对请求作出响应，“视图” 可以是某种模板，或者仅仅是格式化的数据。Geddy 用 Node.js 实现了它的 HTTP 服务器，在服务器端执行所有的函数、控制器动作、模板渲染时也使用了 Node.js。由于是全 JavaScript 堆栈， 很多相同的代码也是在浏览器中运行的。

Astro（node-xmpp）: Node. js 追求可伸缩的服务器软件性能，这点对我来说非常重要。这就是为什么我在评估可用的 XML SAX 解析器库之后，自己创建了 node-expat 的原因。Expat 用 C++编写，和快速的解析器库 libexpat 进行了绑定。

我过去经常使用 Ruby XMPP 库 xmpp4r，所以 node- xmpp 仿照了 xmpp4r 的一点儿内容。客户端和组件都来自于连接，连接进行 XML 流处理。连接基于普通的 TCP 客户端流，所以用户仍然能访问原始数据管理、发送队列管理等功能。

Peteris和James（StackVM）：我们的主要组成部分就是StackVM本身，它使用了我们为Node.js编写的几个其他模块：

- dnode：DNode为远程方法调用提供了一个简单、面向对象的接口，并借助透明包装的回调来传送数据， 从而简化了StackVM进程和Web浏览器之间的通信。
- node-bufferlist： 使用bufferlist，我们可以用异步方式构建一个二进制的流解析器。
- node-rfb：这个模块用node- bufferlist告诉客户端VNC使用的RFB协议。
- node-png：node- rfb捕获到屏幕更新后，我们用该模块渲染PNG图片。
- node-jpeg：这个模块使用node-rfb的输出，生成JPEG格式的屏幕更新图片。
- node-video：借助这一模块， 我们可以为node-rfb捕获到的屏幕更新保存截屏记录。
- node-base64：该模块对node-png、node- jpeg的输出进行base64 编码，并把结果发送给浏览器（ 因为没有什么简单的方法可以对二进制数据进行流传输）。

接着我们用Socket.IO 和别人编写的Socket.IO-node模块来完成浏览器和StackVM之间的通信。

StackVM 的工作原理串起来就是：StackVM 使用 node-bufferlist 和 node- rfb 打开一个到虚拟机 VNC 端口的连接。屏幕出现更新时，node-rfb 将更新信息转发给 node-png 或 node-jpeg，由它们生成图像。接着，图像由 Socket. IO 转发给浏览器。DNode 对 Socket. IO 的传输进行了抽象，用更加异步的方式在服务器和浏览器之间路由消息。

InfoQ：你们选用 Node.js 的主要原因是什么？ 日常工作中会有哪些好处？

TJ（Express）：大概在一年前，当我开始做 Express 的时候我对 Node 产生了兴趣。我希望用熟悉的语言、熟悉而简单的方式去定义 Web 应用。我听说 Node 不久后就下载了源码、简单阅读了一下，我喜欢它所引导的方向。用了几年 Ruby 之后，我发现我越来越喜欢用 JavaScript 了。

Guillermo（Socket.IO）：用 Node 编写实时应用真的非常简单。我们选择 Node 的主要原因是， 在 Node 里你是司机，它能让你只用几行代码就创建出一个自己的 HTTP 服务器。传统堆栈都没有这种能力，比如常见的 Apache+PHP，因为 PHP 只能在请求完成时执行，而且要运行一小段时间，才能生成响应并退出。

另一方面，不用牺牲性能和可伸缩性，Node 的请求就可以延长一段时间，因为 Node 是个完全异步、单线程、单进程的框架。

Matthew（Geddy）：我选择 Node.js 是因为，它是第一个可靠的服务器端 JavaScript 实现，不以 JVM 为基础。我在日常工作中编写了很多 JavaScript 代码，一个很大的优点就是，不需要针对客户端和服务器端在两种语言之间换来换去。

Astro（node-xmpp）：对 IO 操作比较多的应用，我想控制它们的并发问题。典型的选择是线程编程，但这很昂贵。Erlang 和 GHC 把轻量级的绿色线程映射到了本地线程上（一个 CPU 内核有一个本地线程），以此来解决并发问题。尽管有这一功能，开发人员仍要忍受同步问题、避免竞态条件。

异步编程时不用开发串行的“线程”代码，你会觉得这有些不自然。但不处理那些麻烦的同步问题确实让我觉得轻松了很多。一旦你理解了这个概念，就会习惯它带来的“阶梯效应”，功能会逐层嵌套实现。

Peteris 和 James（StackVM）：不用刻意展开或编写 C 代码，Node.js 就有非常棒的性能。Node. js 没有任何需要担心的遗留代码，所有内容自底向上都是异步的。还有很多库在尝试一些新的事情，比如 websockets，社区也非常活跃。 在浏览器和服务器端使用相同的语言让代码重用变得更为简单，而且 JavaScript 语言很简单，能迅速完成原型。

InfoQ：你觉得 Node.js 成为主流的时机成熟了么？ 你怎么看它周围的生态系统呢？

TJ（Express）：这个问题不太好回答，和其他很多社区相比，Node.js 仍然很小。我个人觉得这个项目（和社区）有很大的潜力，而且 Django、Drupal 或 Ruby on Rails 最终也很有可能会跟进，甚至做出更大的贡献。

Guillermo（Socket.IO）：Node正在非常迅 速地成熟起来。在 GitHub上，它的追随者现在差不多是 Rails追随者的一半， Rails无疑是最流行的Web应用开发框架之一。而这只用了一段很短的时间。

Matthew（Geddy）：对有些目标来说它已经很成熟了，但人们要像用 Ruby 或 Python 那样去进行通用的开发还为时尚早。也许还需要一两年时间。

Astro（node-xmpp）：Node. js 有朝一日会很容易成为主流的，因为 JavaScript 是个相当流行的语言。

Node.js 一直在发展，你应该经常针对新版本测试你的代码。如果你打算让任何人都能运行你的代码，却不管用户的 Node. js 版本到底是新还是旧，我觉得这会有问题。

Peteris 和 James（StackVM）：它要是在某个方面还不够成熟，我们会让它更加成熟的。它是开源的，所以我们要是发现一个 Bug、或是需要一些新特性，必要的话我们会自己进行处理。社区驱动的生态系统非常棒：开源、很多想法、大量参与其中的开发人员、一大堆文档和博客文章， 还有 GitHub 上每天都会有更新的 Node.js 库。

API 在不同的版本之间确实会略有变化，但 Ryan Dahl 承诺，即将发布的 0.2 版本会具备全新的稳定性。

InfoQ：你在项目或日常工作中使用其他的 Node. js 库或框架么？除了你自己的库，你觉得有没有其他的 Node. js 库或框架是特别有用、别的开发人员也应该去看一看的？

TJ（Express）：日常工作中我用了很多自己的库，还有下面几个库，我对这几个项目和它们的开发人员怀有深深的敬意：

- Socket-IO（Guillermo Rauch）
- Formidable （Felixge）
- Less.js（Cloudhead）
- Redis客户端 （Fictorial）

Guillermo（Socket.IO）：我们天天都用Node。除了Socket.IO，我支持 Mongoose和其他一些较小的组件， Mongoose是一个ORM框架，针对NoSQL、面向文档的数据库MongoDB。

Matthew（Geddy）：Node-Jake是个类似于Make/ Rake的JavaScript构建工具。Logan是个小型的跨环境测试框架， 可以用它在浏览器里测试只能运行在Node.js中的代码（还有TheRubyRacer，它把V8 嵌套在一个Ruby进程中）

Astro（node-xmpp）：有几个需要提一下。第一个是 Isaac Schlueter 开发的包管理器 npm。无论什么地方， 你都想拥有包管理器所带来的便利。

TJ Holowaychuk 开发的 ext.js 相当有用：它用很多方便的方法对基本的 JavaScript 类型进行了增强 ，所有人都会发现这很有用。

不要错过 Senchalab 的 Connect Web 框架。Node.js 内置的 HTTP 支持是它的杀手锏，Connect 想要达到 Rack 在 Ruby 世界中的水平。当然，这并不像看起来的那么简单，因为它完全是异步的。

Peteris和James（StackVM）：借助Node包管理器 npm 去使用、发现其他开发人员编写的库是一种非常好的方式。 我们使用Socket.IO，它对WebSocket进行了抽象， 还对尚没有WebSocket的浏览器提供了应变方案，以支持WebSocket式的通信。我们也一直在尝试基于Socket.IO的node- compress，以便加快把数据传输到浏览器的速度。

InfoQ：如果一个团队要开始使用 Node.js， 他们需要留心哪些陷阱？有提示或“最佳实践”么？

TJ（Express）：一般来说都有环境问题。 我认为Git是必需的，其他SCM都让我觉得不太满意， 实际是有点儿反感。搭建一个持续集成的服务器，哪怕像 CIJoe那么简单，那也是必需的； 还有就是测试、测试、再测试！我着重强调测试，我所有的Node项目都用“Expresso”进行测试，“Expresso” 项目支持代码覆盖率报告，能让你很明确地知道哪些代码没有覆盖到， 这些简单的步骤应该能让项目不停向前滚动。

Guillermo（Socket.IO）：我的建议是：

- 对 JavaScript 语言要有非常强烈、近乎狂热的求知欲。
- 深刻理解保证网站运转的各个层面。这意味着要理解底层的知识（ 熟知 HTTP、协议和网络通常会有较大的帮助）。
- 对于某些功能，试着“忘记”其他语言实现它们的方式，学习用“ Node”的方式去实现它们（ 比如在 Java 里用线程去解决的问题， 你可以用 Node 以不同的方法去轻松解决）

Matthew（Geddy）：Node. js 仍然处于发展初期，所以仍然会有 API 的变化， 这会破坏代码。要准备好修改代码， 或者用 Geddy 或 Express 等框架，框架的作者会屏蔽 Node 的变化，保证 API 的一致性。

Astro（node-xmpp）：尽管 JavaScript 的语法微不足道，看过用大括号划分程序代码的任何人都能看懂， 但基于原型的遗留系统一开始还是会造成混乱。一定要告诉你的伙伴会有什么样的影响。

要是你仍然觉得基于类的遗留系统更容易使用， 那你可能需要找一个处理方式类似于用 JavaScript 处理遗留 Ruby 的库。

Peteris 和 James（StackVM）：阻塞 I/O 可能是需要留意的最大陷阱，因为整个程序等到调用完成之后才会执行其他操作。只在初始化时使用少量的阻塞调用，尤其是在编写注重性能的网络服务时。异步 I/ O 和事件驱动编程需要的思维方式和大多数编程都不太相

同。 拥抱它吧。

InfoQ：根据你的经验， JavaScript 代码库规模如何像项目那样变得越来越大？ 你用的工具有哪些？你觉得工具对现实中的 Node. js 项目支持得怎样？

TJ（Express）：我觉得这完全由开发人员来决定。JavaScript 过去曾是一门“丑陋”的语言，但现在已经不是了。我觉得行内注释非常重要，有助于创建可见的空白，而且不会让代码看起来混乱不堪。至于整个项目的组织，把逻辑块分布到单独文件中绝不会有错，对任何语言来说这都是通用的做法。

Node 采用了 CommonJS 的模块模式，这其实是我在语言中使用过的最喜欢的组织工具。最典型的，大多数环境都包含一个扁平的名字空间，所以在现实中你仍然要随时处理名字空间的冲突，而 CommonJS 的模块系统就很好地避免了冲突。

Guillermo（Socket.IO）：让 JavaScript 同时在服务器、客户端上运行是很让人兴奋的。随着招聘变得越来越容易，整个团队可以处理同一应用的很多不同方面，我觉得这对公司来说特别重要。Node 也为专业环境中的开发提供了很多工具：

- 简单却强大的模块系统（基于 CommonJS）
- 很多测试框架（Expresso、Vows 等）

Matthew（Geddy）：在服务器端， JavaScript 现在和 Ruby、Python 一样，已经有很多工具了—— CommonJS 模块提供了让代码保持模块化的机制。Node- Jake 等构建工具能让你的构建和打包自动化，以可预测的方式进行。

客户端的 JavaScript 还需要再多做一点儿工作。但 JavaScript 对象能很自然地封装功能，而且 JavaScript 很久以前就有了在客户端进行“ 大规模编程”的最佳实践。Facebook 等大公司创建了他们自己的框架，但 Dojo 或 YUI 等更复杂的工具包则需要更多这样的支持。

Astro（node-xmpp）：我推荐在大型项目中使用静态类型。

我也很喜欢测试驱动开发；不知道 Node. js 社区是否已经选择了测试库的事实标准。

Peteris 和 James（StackVM）：Node. js 有一个模块系统，它对项目的可伸缩性大有裨益，但大型软件项目不论用什么语言，一般都有更多的 Bug，而且比小型项目更难测试。对于 StackVM，我们将功能分解为可重用、开源的模块，以此来尽可能长久地避免大型代码库的这种缺陷。到目前为止，这种方法让 StackVM 的主要代码库更为灵活， 对支持库进行单独测试也更容易。 社区里使用我们模块的人已经为我们提出了很多 Bug。 现在对 Node. js 来说最有用的工具也许是 Node 包管理器 npm。 使用 npm，我们能轻松使用其他程序员编写的库， 我们也可以很容易地和 Node.js 社区分享我们自己的模块。

InfoQ：你觉得 Node.js 现在还缺些什么、 需要添加进来呢？你认为它应该如何发展？

TJ（Express）：我觉得应该继续保持原样，为框架做越来越有影响的底层基础。它应该从“空白”开始， 不受开发人员的影响。

Guillermo（Socket.IO）：HTTP 服务器对 SSL 的支持还不完整，还有很多 Bug。除此之外，它最终是要让人们去用的，要考虑通用用法、修复 Bug，继而实现最佳模式。

Matthew（Geddy）：大家都在使用包管理器（NPM），所以希望在 NPM 稳定之后，我们能以一种简单的方式安装库、 为构建的项目指定依赖关系。

很多刚起步的项目都在尝试处理数据存储和持久化，但还没有最佳解决方案。我们仍需要一个稳定的 ORM， 它能为关系型存储和非关系型存储同时提供通用的 API。

服务器端 JavaScript 还有很多尚未开始开发的需求， 所以我觉得生态系统会很自然地转去填补目前的这些空白。

Astro（node-xmpp）:我特别想从 node- xmpp 中删除对 node-base64 的依赖。暴露 Base64 编码/解码的补丁已经发布，但还没有合并进来。

我发现到处都在提 CommonJS。这个项目旨在对跨多个 JavaScript 接口/ 框架的库进行标准化。他们已经实现了模块系统，NPM 也使用他们的 package.json 规范。除 CommonJS 之外，恐怕不会有太多的共享模式，因为相对很多其他的实现来说，Node. js 里的所有内容都必须是异步的。

对很多人来说，找到合适的 Web 框架也很有意思。

异步编程有时候非常麻烦，我明白开发人员有时候忽略回调引用，是否只是想让控制流能更加直截了当一些。

Peteris 和 James（StackVM）：每当我们想出一 些有潜力的 Node.js 新库，我们会自己开发并发布出来，这样其他开发人员就不用浪费时间去重新发明轮子了。其他很多程序员也都在这么做，库的支持生态系统就会很自然地演进。Node. js 核心就是要尽可能小，由支持库来提供大量的功能，对社区驱动的项目来说，这是一个分配工作负担、 鼓励参与的很好的方式。

InfoQ：你项目以后的路线图是怎样的？

TJ（Express）：现在还是让 Express 保持精简。对于那些适合项目的重点功能，我都会接受它们的新思路和补丁。开发人员现在甚至可以把 Express 看成是“框架的框架”，因为对那些更大型、偏向于自己使用的框架来说，它本身就很容易成为构建块。最后我要感谢 Ciaran 和 Aaron 的巨大贡献，还有我的老板 Sencha 允许我和 Tim 回馈社区。

Guillermo（Socket.IO）：我现在正努力让 So cket.IO 尽可能地迅速、精简、易于测试。

Matthew（Geddy）：Geddy 还有很多工作要做：

1. ORM 查询 API

   目前模型代码中的持久化部分是非常重要的。ORM 非常复杂，任何既要和关系型存储交互， 又要和非关系型存储交互的内容都越来越难构建。

2. 插件 API

   Geddy 需要一种简单的方式，去支持第三方构建的插件功能。

3. 认证

   认证就是应该用插件去实现的绝好例子。

4. 更好的模块化

   编写 Geddy 是为了提供一种集成的、“开箱即用”的体验，让开发人员编写真实应用时不那么手忙脚乱。但 Geddy 提供组件应该更容易选择和使用，或者某个特定的部分能很容易地被其他内容替换。

   除了简单的 EJS，还要支持其他的模板库

   人们喜欢用奇妙的东西，比如 Haml 或 Moustache。Geddy 应该让大家能使用这些内容。

5. 测试覆盖率

Geddy 有大量针对路由器的测试，还有一套针对模型和验证的测试套件，但其余代码还没有覆盖到。Geddy 的代码需要进行重构、要更容易测试，以便我们有更完整的覆盖率。

> **Astro（node-xmpp）：**我很高兴在发布项目之后，Github 上马上就有了关注者。除此之外， 我一直在期待一些实质性的贡献。
>
> 我自己的目标已经达到了，因为我只想完成 XML 解析和认证，并不是在库的基础上实现所有特定的 XMPP XEP 规范。XMPP XEP 规范的实现就留给应用开发人员去完成吧。

> **Peteris 和 James（StackVM）：**短期内，我们会侧重于添加更多的功能，还有性能优化。以后我们很可能会关注硬件和可伸缩性。

在 InfoQ上，你可以找到更多关于 性能和可伸缩性 、或 Java Script的内容！

原文链接：http://www.infoq.com/cn/articles/nodejs-frameworks

相关内容：

- Node.js开源项目获得Joyent资助，发展进入快车道
- Node.js基于Google V8 提供了基于事件的I/O处理
- Javascript的兴起是否意味着LAMP的终结？
- 使用Apache Avro
- FireAtlas：ASP.NET AJAX查看器

推荐文章 | Articles

Java 深度历险（七）--Java 反射与动态代理

作者 成富

在上一篇文章中介绍 Java 注解的时候，多次提到了 Java 的反射 API。与 javax.lang.model 不同的是，通过反射 API 可以获取程序在运行时刻的内部结构。反射 API 中提供的动态代理也是非常强大的功能，可以原生实现 AOP 中 的方法拦截功能。正如英文单词 reflection 的含义一样，使用反射 API 的时候就好像在看一个 Java 类在水中的倒影一样。知道了 Java 类的内部 结构之后，就可以与它进行交互，包括创建新的对象和调用对象中的方法等。这种交互方式与直接在源代码中使用的效果是相同的，但是又额外提供了运行时刻的灵 活性。使用反射的一个最大的弊端是性能比较差。相同的操作，用反射 API 所需的时间大概比直接的使用要慢一两个数量级。不过现在的 JVM 实现中，反射操作的性能已经有了很大的提升。在灵活性与性能之间，总是需要进行权衡的。应用可以在适当的时机来使用反射 API。

基本用法

Java 反射 API 的第一个主要作用是获取程序在运行时刻的内部结构。这对于程序的检查工具和调试器来说，是非常实用的功能。只需要短短的十几行代码，就可以遍历 出来一个 Java 类的内部结构，包括其中的构造方法、声明的域和定义的方法等。这不得不说是一个很强大的能力。只要有了 java.lang.Class 类 的对象，就可以通过其中的方法来获取到该类中的构造方法、域和方法。对应的方法分别是 getConstructor、getField 和 getMethod。这三个方法还有相应的 getDeclaredXXX 版本，区别在于 getDeclaredXXX 版本的方法只会获取该类自身所声明的元素，而不会考虑继承下来的。Constructor、Field 和 Method 这三个类分别表示类中的构造方法、域和方法。这些类中的方法可以获取到所对应结构的元数据。

反射 API 的另外一个作用是在运行时刻对一个 Java 对象进行操作。 这些操作包括动态创建一个 Java 类的对象，获取某个域的值以及调用某个方法。在 Java 源代码中编写的对类和对象的操作，都可以在运行时刻通过反射 API 来实现。考虑下面一个简单的 Java 类。

```
class MyClass {
    public int count;
    public MyClass(int start) {
```

```
        count = start;
    }
    public void increase(int step) {
        count = count + step;
    }
}
```

使用一般做法和反射 API 都非常简单。

```
MyClass myClass = new MyClass(0); //一般做法
myClass.increase(2);
System.out.println("Normal -> " + myClass.count);
try {
    //获取构造方法
    Constructor constructor = MyClass.class.getConstructor(int.class);
    //创建对象
    MyClass myClassReflect = constructor.newInstance(10);
    //获取方法
    Method method = MyClass.class.getMethod("increase", int.class);
    //调用方法
    method.invoke(myClassReflect, 5);
    //获取域
    Field field = MyClass.class.getField("count");
    //获取域的值
    System.out.println("Reflect -> " + field.getInt(myClassReflect));
} catch (Exception e) {
    e.printStackTrace();
}
```

由于数组的特殊性，Array类提供了一系列的静态方法用来创建数组和对数组中的元素进行访问和操作。

```
Object array = Array.newInstance(String.class, 10); //等价于 new String[10]
Array.set(array, 0, "Hello");  //等价于array[0] = "Hello"
Array.set(array, 1, "World");  //等价于array[1] = "World"
System.out.println(Array.get(array, 0));  //等价于array[0]
```

使用Java反射API的时候可以绕过Java默认的访问控制检查，比如可以直接获取到对象的私有域的值或是调用私有方法。只需要在获取到Constructor、Field和Method类的对象之后，调用 setAccessible方法并设为true即可。有了这种机制，就可以很方便的在运行时刻获取到程序的内部状态。

处理泛型

Java 5 中引入了泛型的概念之后，Java反射API也做了相应的修改，以提供对泛型的支持。由于类型擦除机制的存在，泛型类中的类型参数等信息，在运行时刻是不存在的。JVM看到的都是原始类型。对此，Java 5 对Java类文件的格式做了 修订 ， 添加了Signature属性，用来包含不在JVM类型系统中的类型信息。比如以java.util.List接口为例，在其类文件中的Signature属性的声明是<E:Ljava/lang/Object;>Ljava/lang/Object;Ljava/util /Collection<TE;>;; ，这就说明List接口有一个类型参数E。在运行时刻，JVM会读取Signature属性的内容并提供给反射API来使用。

比如在代码中声明了一个域是 List<String>类型的，虽然在运行时刻其类型会变成原始类型 List，但是仍然可以通过反射来获取到所用的实际的类型参数。

```
Field field = Pair.class.getDeclaredField("myList"); //myList的类型是List
Type type = field.getGenericType();
if (type instanceof ParameterizedType) {
    ParameterizedType paramType = (ParameterizedType) type;
    Type[] actualTypes = paramType.getActualTypeArguments();
    for (Type aType : actualTypes) {
        if (aType instanceof Class) {
            Class clz = (Class) aType;
            System.out.println(clz.getName()); //输出java.lang.String
        }
    }
}
```

动态代理

熟悉设计模式的人对于 代理模式 可 能都不陌生。 代理对象和被代理对象一般实现相同的接口，调用者与代理对象进行交互。代理的存在对于调用者来说是透明的，调用者看到的只是接口。代理对象则可以封装一些 内部的处理逻辑，如访问控制、远程通信、日志、缓存等。比如一个对象访问代理就可以在普通的访问机制之上添加缓存的支持。这种模式在 RMI 和 EJB中 都得到了广泛的使用。传统的代理模式的实现，需要在源代码中添加一些附加的类。这些类一般是手写或是通过工具来自动生成。JDK 5 引入的动态代理机制，允许开发人员在运行时刻动态的创建出代理类及其对象。在运行时刻，可以动态创建出一个实现了多个接口的代理类。每个代理类的对象都 会关联一个表示内部处理逻辑的 InvocationHandler接口的实现。当使用者调用了代理对象所代理的接口中的方法的时候，这个调用的信息会被传递给InvocationHandler的invoke方法。在 invoke方法的参数中可以获取到代理对象、方法对

应的Method对象和调用的实际参数。invoke方法的返回值被返回给使用者。这种做法实际上相 当于对方法调用进行了拦截。熟悉AOP的人对这种使用模式应该不陌生。但是这种方式不需要依赖 AspectJ等AOP框架。

下面的代码用来代理一个实现了 List 接口的对象。所实现的功能也非常简单，那就是禁止使用 List 接口中的 add 方法。如果在 getList 中传入一个实现 List 接口的对象，那么返回的实际就是一个代理对象，尝试在该对象上调用 add 方法就会抛出来异常。

```
public List getList(final List list) {
   return (List) Proxy.newProxyInstance(DummyProxy.class.getClassLoader(),
new Class[] { List.class },
       new InvocationHandler() {
           public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
               if ("add".equals(method.getName())) {
                   throw new UnsupportedOperationException();
               }
               else {
                   return method.invoke(list, args);
               }
           }
       });
 }
```

这里的实际流程是，当代理对象的 add 方法被调用的时候，InvocationHandler 中的 invoke 方法会被调用。参数 method 就包含了调用的基本信息。因为方法名称是 add，所以会抛出相关的异常。如果调用的是其它方法的话，则执行原来的逻辑。

使用案例

Java 反射 API 的存在，为 Java 语言添加了一定程度上的动态性，可以实现某些动态语言中的功能。比如在 JavaScript 的代码中，可以通过 obj["set" + propName]()来根据变量 propName 的值找到对应的方法进行调用。虽然在 Java 源代码中不能这么写，但是通过反射 API 同样可以实现类似 的功能。这对于处理某些遗留代码来说是有帮助的。比如所需要使用的类有多个版本，每个版本所提供的方法名称和参数不尽相同。而调用代码又必须与这些不同的版本都能协同工作，就可以通过反射 API 来依次检查实际的类中是否包含某个方法来选择性的调用。

Java 反射 API 实际上定义了一种相对于编译时刻而言更加松散的契约。如果被调用的 Java 对象中并不包含某个方法，而在调用者代码中进行引用的话，在编译时刻就会出现错误。而反射 API 则可以把这样的检查推迟到运行时刻来完成。通过把 Java 中的字节代码增强、类加

载器和反射 API 结合起来，可以处理一些对灵 活性要求很高的场景。

在有些情况下，可能会需要从远端加载一个 Java 类来执行。比如一个客户端 Java 程序可以通过网络从服务器端下载 Java 类来执行，从而可以实现自动更新的机制。当代码逻辑需要更新的时候，只需要部署一个新的 Java 类到服务器端即可。一般的做法是通过自定义类加载器下载了类字节代码之后，定义出 Class 类的对象，再通过 newInstance 方法就可以创建出实例了。不过这种做法要求客户端和服务器端都具有某个接口的定义，从服务器端下载的是 这个接口的实现。这样的话才能在客户端进行所需的类型转换，并通过接口来使用这个对象实例。如果希望客户端和服务器端采用更加松散的契约的话，使用反射 API 就可以了。两者之间的契约只需要在方法的名称和参数这个级别就足够了。服务器端 Java 类并不需要实现特定的接口，可以是一般的 Java 类。

动态代理的使用场景就更加广泛了。需要使用AOP中的方法拦截功能的地方都可以用到动态代理。Spring框架的 AOP实现 默认也使用动态代理。不过JDK中的动态代理只支持对接口的代理，不能对一个普通的Java类提供代理。不过这种实现在大部分的时候已经够用了。

参考资料

- Classworking toolkit: Reflecting generics
- Decorating with dynamic proxies

原文链接：http://www.infoq.com/cn/articles/cf-java-reflection-dynamic-proxy

相关内容：

- 阅读者（九）: 编程人生
- Simon探讨编程语言与研究工作
- Java深度历险（六）--Java注解
- CORBA宗师谈REST、Web服务和Erlang
- 讨论 5 种跟踪Java执行的方法

推荐文章 | Articles

书评和采访：ExtJS in Action

作者 Dionysios G. Synodinos 译者 崔康

Jesus Garcia 编写的《ExtJS in Action》 是一本介绍跨浏览器JavaScript库ExtJS的书，ExtJS用于构建富互联网应用（Rich Internet Application）。ExtJS包含大量的组件、可扩展的组件模型，提供了易用的API来构建针对基于JavaScript的Web应用的完整、稳定的平台。

本书作者在IT和企业web应用开发领域拥有超过 16 年的经验。他在数个大公司部署过基于ExtJS的应用，如万豪国际、洛克希德马丁、摩根大通和美国金融监管局。业余时间，Jay喜欢帮助ExtJS社区，他是 ExtJS非官方雇员中最活跃的发帖者之一（超过 13000 个帖子）。

本书包括如下内容：

- 使用核心组件如布局面板构建动态 UI
- 创建动态表单，提交数据无需页面刷新
- 使用树状组件、菜单和工具栏的有用组件
- 创建定制的复合组件
- 全面解析 Ext JS 3.0

读者会了解 ExtJS 组件模型和布局，学习核心组件如 Container 类在构建复杂用户界面时作为构建模块的运行机制。本书覆盖了 ExtJS 常用类、AJAX、Observable (ExtJS 事件模型 I)、DOM 辅助类和辅助函数，并介绍了如何使用 JavaScript Object Notation (JSON)。它还探讨了定制或者扩展 ExtJS 组件的可行办法。

本书的出版商Manning为InfoQ读者提供了第三章试读《组件模型和生命周期》。

InfoQ 采访了本书作者：

InfoQ：Hi Jesus，能否介绍一下自己，以及编写这本有关 ExtJS 书籍的初衷？

Jesus：我从 2006 年初就开始使用 ExtJS，当时它只是 YUI 库的一个扩展集。那时，它被称为

“YUI- ext”，由一个名为 Jack Solcum 的家伙作为边缘项目开发。像许多刚接触该框架的新人一样，我最初被数据网格吸引。真正让我倾心该库的是文档，可能是最好最新的 Ajax 框架 API。

在我接触该框架不久，我决定成为一名专家，专注我的事业，成为一名独立咨询顾问。做出决定之后，我开始活跃于早期社区，花费大量时间在帮助社区上，回答人们的提问解决他们的问题。通过这种方式，我学到了比只是应用开发更多的有关框架的知识。直到现在，通过帮助别人来回馈社区仍然是我学习大量知识的一种方式。

活跃于 ExtJS 社区使我赢得了一些声誉，因此被来自 Manning 的 Michael Stephens 看中。他联系我，还有其他活跃的社区成员，来审阅一本书的提案。在阅读了提案内容表格的前几行之后，我就意识到这本书不是“Ext JS in Action”。在提交了我的想法和推荐的目录之后，Michael 举行了一个审查会议，在那里他问我是否愿意提交一个自己的目录，然后我就提交了。然后两年之后，现在我们有了这本书。

InfoQ：什么是 ExtJS，它想解决什么问题？

Jesus：Ext JS 是一个针对 Web 的 UI 库。它是市场上的轻量级工具，因为它通过抽象，支持开发人员不必强制编写 HTML 的源代码，而且允许开发人员灵活地扩展组件库或者通过简单强大的插件架构创建真实的插件。

Ext JS 解决了管理 HTML 和 CSS 的问题，提供了媲美桌面应用的 UI 外观。它支持开发人员在现有 HTML 网页中注入组件或者开发富企业级 RIA，其中一些应用于关键任务应用，如实时监控和网络基础设施的管理。

框架也包含许多实用工具，如数据存储、模板和 xTypes，进一步提高了应用开发能力。

InfoQ：在书中，你提到 Ext JS 遵循的组件和容器模型是它比其他 Ajax 库与众不同的主要原因，你能详细说说吗？

Jesus：组件和容器模型（还有其他特点）使 Ext JS 在其他同类 Ajax 库中别具一格，因为它们使库的 UI 部分结构化。没有它，Ext JS 不会这么与众不同。

组件模型为框架引入了“组件生命周期”，为组件带来了一套定义良好和可靠的行为。它分为三个阶段，初始化（Initialization）、渲染（Render）和销毁（Destruction）。初始化指组件新实例的初始化时刻。渲染阶段指组件被渲染或者绘制到屏幕上的时候。最后，销毁阶段支持组件执行清理任务，如所有注入的 HTML 节点的删除和清除事件监听器。

容器模型支持组件通过父子关系模式来管理其他模型。通过容器模型，开发人员可以轻松地

创建动态 UI——利用它的 add()或者 remove()来添加或者删除一个或几个子元素。为了管理子元素的视觉组织，容器使用了布局（Layout）。

InfoQ：你能透漏一下工作中如何使用 Ext JS 吗？你了解其他人如何使用它吗？是否存在 RIA 应用不适合采用 Ext JS？Ext JS 适合移动 web 应用吗？

Jesus：我通常使用 Ext JS 为许多客户开发应用。我最近一个任务是为客户开发界面管理成员信息。因为 Ext JS 功能齐全，所以我可以利用它开发各种应用，比如定制 UI 从 IBM 的 Tivoli Enterprise Console 和 HP Universal CMDB 中合并数据、为 EMC 的 Documentum 产品开发前端界面。我已经开发了几个应用来帮助公司制定考试内容。

开发人员可以充分利用 Ext JS 来开发 RIA 中所需的功能。这几年，我已经看到了各种各样的应用，大部分是基于业务的。

Ext JS 本身不是为移动 web 应用而定制的，但是它的兄弟 Sencha Touch 是。我已经看到并且实验了在 iPad 上运行有限功能的 Ext JS 应用。它不支持触摸功能，而且 UI 主题不是以移动为中心的。

InfoQ：Ext JS 在各种浏览器平台上的运行情况如何？你想对那些由于当前的兼容性问题或者害怕因未来升级失去兼容性而惧怕使用 Ext JS 的朋友说些什么？

Jesus：这可是个老问题了。我想说 Ext JS 在最好的环境中运行最好。一般来讲，Ext JS 在基于 Webkit 的浏览器中运行最好，在 IE 中最差。当然，浏览器还要更加努力解决这些问题，但是在 Ext JS 3.0 的最初版本中存在一些代码问题导致 IE 的缓慢性能。

众所周知，Ext JS 3.0 的最初版本的性能比 2.X 差得多，但是在后续版本中好得多。Ext JS 团队非常重视来自于客户和社区成员的反馈，并采取必要修改在不限制功能的情况下获得更好的性能。

我也提到框架的性能需要开发人员配合做大量的工作。 初级开发人员往往易犯简单错误，如所谓的“overnesting”。这种情况是指不必要的通过父组件包装子组件，并且为 UI 添加了不必要的层。另一个常见错误是加载数以千计的记录到视图中（GridPanel、DataView、ListView 或者 ComboBox），导致屏幕中出现“JavaScript is running too long”消息。

从 2.x 升级到 3.x 非常简单，在 3.x 不同版本中升级也是如此。以我的经验来说，大家遇到的大部分升级问题是他们存在代码依赖、使用或者重载了私有方法，这些方法标记为私有是有原因的。显然，这种做法不值得推荐给团队甚至资深的开发人员，你应该做些工作使升级更容易。

InfoQ：对于开发 Ext JS 应用，你喜欢哪种过程？你使用哪些工具？你觉得现在的 JavaScript/HTML 工具好用吗？

Jesus：通常我在笔记本电脑上开发应用，这样可以尽可能快地收到反馈。我遵循框架模式来开发应用。例如，我喜欢根据类的命名空间在文件系统（和项目）中安排文件。这种做法能够快速访问文件。同样的模式适用于插件、用户扩展（ux）或者应用于框架的重载。

开发 JavaScript 代码有一些优秀的工具，但是我最喜欢使用 Jetbrains' Intellij IDEA 开发应用。虽然它的运行速度不是最快的，但是我爱上了它。它支持比如动态代码补足和重构类成员等，还有自己的 JavaScript 调试器，虽然我一般不用它。

我听过开发人员喜欢使用 NetBeans 甚至 Aptana（包装了 Eclipse）。

InfoQ:使用大型的客户端框架如 Ext JS 会导致调试时有困难吗？代码遇到问题时，你如何调试？

Jesus:困难有很多原因。首先，开发人员使用 ext-all.js(框架的精简版)而不是 ext-all- debug.js（保留了空格，删除了注释）。如果异常被 JS 解释器抛出，异常消息因为 JS 的简化通常晦涩难懂。如果在 IE 中查看错误，一般看不出任何意义，增加了难度。通常，为了解决问题，可以使用解压缩的 ext-all-debug.js 来开发应用，这会使错误消息好懂些，至少可以找到异常抛出的准确行数。

一旦 ext-all-debug.js 抛出异常，我通常会查看异常发生的位置，从中找到异常对应的方法，然后找到该方法归属的类。绝大多数情况下，都是些低级错误。

因为我使用 WebKit，所以查看 ext-all-debug.js 文件成百上千的 JS 代码轻而易举。可悲的是，Firefox 的当前版本妨碍了 Firebug 达到同样的性能水平，因此几个月之前我从 Firefox 转移到了 WebKit。我也在 IE8 中尝试过，但是结果让我很崩溃。

InfoQ：在工作中使用功能测试方法吗？感觉如何？

Jesus：答案是没有。我曾经想在客户的应用开发周期中加入单元测试，但时间是个问题。花费在编写测试的时间可以节省未来修补缺陷的更多时间，但是做出初始投资对客户很困难。

InfoQ：在书中，你描述了使用 Ext JS 时框架之外的若干组件，易于扩展吗？易于创建新组件吗？

Jesus：我觉得易于扩展框架，因为我从 2006 年就开始使用它了。开发 Ext JS 的时候就被设

计为可扩展的，存在针对它编写的插件，而且采用了一些最佳的 JavaScript 开发模式。

扩展组件的难度水平与开发人员对 Object Oriented JavaScript 模式和框架的经验有关。你必须熟悉 OOJS（原型继承）才能够扩展组件。毕竟，如果你不理解原型继承，在 JavaScript 内扩展的概念会比较生疏。还需要能够理解例如执行上下文概念和关键字“this”的使用方法。

同样，你应该熟悉 Component、Container 和布局模式来正确的扩展组件同时避免例如 overnesting 的常见错误。熟悉框架的其他部分也有帮助。

我写这本书主要是因为看到很多初识框架的开发人员想要构建复杂的应用。然而，他们发现在完成简单任务时就磕磕绊绊，这都是因为没有必要的基础知识来完全理解和使用框架。

对于想提高 JavaScript 速度的人来说，我推荐 Dustin Diaz 的《Pro JavaScript Techniques》（Apress）。这本书会详述现代 JavaScript 编程，是你充分了解一些高级代码。

InfoQ：随着 HTML5 的出现，你认为 Ext JS 会如何发展？

Jesus：Sencha 团队的一个优点是向前看。他们努力使 4.0 兼容 HTML5 和 CSS3。Ext JS 4.0 与 Sencha Touch 共用同一个代码库，而 Sencha Touch 已经对这些技术有大量的支持。

InfoQ :你觉得本书对于那些初识 Ext JS 的人和想在 Ext JS 开发中更进一步的人有哪些帮助？

Jesus：对于 Ext JS 的新手来说，我认为深入理解 Component、Container 和 Layout 模型非常有用。我尽可能清晰地讲解这些内容，并从读者中得到了好评。

对于老手来说，我认为本书最有价值的部分是后三章（15-17）。第 15 章介绍了创建 Ext JS 扩展的基本知识，并使用现实例子开发插件。最后两章构建了一个虚构应用，使读者了解核心的决策过程，如安排命名空间和开发抽象来减少冗余代码。

不同经验水平的开发人员在阅读完本书之后都给予了积极的反馈，他们表示其中一些概念在不同的方面提供了帮助。

你可以在InfoQ找到更多有关 **JavaScript** 和 **Rich Internet Applications的内容。**

原文链接：http://www.infoq.com/cn/articles/extjs-in-action

相关内容：

- Ext JS 4 即将正式发布，值得期待、互联网开放平台技术趋势和讨论
- WebBeans（JSR-299）：与规范领导者GavinKing的问答、GWT-Ext 2.0 发布: SanjivJivan访谈

新品推荐 | New Products

图形数据库 Neo4j 更新软件许可方式并增强可用性

作者 Jai Hirsch 译者 吴宇

Neo科技发布了Neo4j 1.3GA（通用版本）。该次发布更新了Neo4j的软件许可并增添了一些新功能。其社区版目前基于GPLv3 许可，与MySQL相同。

原文链接：http://www.infoq.com/cn/news/2011/04/neoej-13

Hudson 以新版本编号 2.0.0 重新登场

作者 Alex Blewitt 译者 马国耀

Hudson 2.0.0发布了！该版本使用了与OSGi/Semantic一致的版本编号。在此之前，Hudson（Jenkins至今都是）遵循JDK的版本编号规则（具体而言，第一位版本号永远是 1，每次发布时第二位数字递增）。另外，此版本中包括JSR330 依赖注入模型，所以它能更方便地运行在OSGi运行时中并且不再依赖于特定的Hudson标注。

原文链接：http://www.infoq.com/cn/news/2011/04/hudson-20

Quartz 2.0 支持 Fluent Configuration API 和工作调度的监控了

作者 Srini Penchikala 译者 王丽娟

开源工作调度器Quartz的最新版本开始支持Fluent Configuration API和工作调度行为的监控、管理了。Terracotta最近发布了这个调度框架的 2.0 版本。Quartz的新版本还提供了两个商业模块：Quartz Manager和Quartz Where。

原文链接：http://www.infoq.com/cn/news/2011/04/quartz-2.0

JavaFX 2.0 发布近况和未来计划

作者 Charles Humble **译者** 黄玲艳

Oracle 已经开始 JavaFX 2.0 的 beta 测试了，这将是它富客户端平台的下一个版本??，试用版只开放给 JavaFX 合作伙伴计划的成员。InfoQ 对话 Richard Bair，了解有关 JavaFX 2.0 的发布和未来计划。

原文链接：http://www.infoq.com/cn/news/2011/04/jfx2ea

Windows Installer XML（WiX）3.5 简化了 Component Authoring

作者 Jenni Ripley **译者** 张龙

近日，Rob Mensching和Bob Arnson发布了Windows Installer XML（WiX）3.5——与Visual Studio集成的安装包创建器。与 之前的报道 不同，WiX并未捆绑到Visual Studio 2010，而是作为一个开源的扩展，兼容于IIS 7.0 和Visual Studio 2010。

原文链接：http://www.infoq.com/cn/news/2011/04/WiX-35

软件架构模拟器 Palladio 3.2 版发布

作者 Michael Stal **译者** 杨晨

Palladio Bench能够帮助架构师和开发者预测软件的各项指标，例如性能，可靠性以及可维护性，还能根据内置的开发模型计算出开发成本。最新版本 3.2 是基于Eclipse 3.5（Galileo），修复一些bug，以及比之前的版本更加稳定，而且还引入了稳定性分析，新的图形编辑器以及新的工作流引擎，还有全新的规范检测架构。

原文链接：http://www.infoq.com/cn/news/2011/04/palladio_tool

Visual Studio 提供对 CPython 的支持

作者 Roopesh Shenoy 译者 朱永光

微软技术计算部门（Technical Computing Group ）最近宣布了 Python Tools for Visual Studio (PTVS)开源项目。PTVS在IronPython Tools for Visual Studio （在 IronPython 2.7中引入)的代码基础上进行了增强，并添加了对CPython、集群（Cluster）的支持，以及诸如NumPy和SciPy这样的新模块。

原文链接：http://www.infoq.com/cn/news/2011/04/VS-CPython

IronPython 完整实现 Python 2.7 的功能

作者 Roopesh Shenoy 译者 朱永光

IronPython 2.7 已经发布，新特性包括 Visual Studio 中开发工具的提升，同 LINQ 和扩展方法更好的互操作，更完整的文档，并提供了和 Python 2.7 同等的完整特性。

原文链接：http://www.infoq.com/cn/news/2011/04/IronPython-2.7

WSO2 引入新的开源项目：WSO2 Message Broker

作者 Jean-Jacques Dubray 译者 丁雪丰

Paul Fremantle 在他的博客上宣布了一个新的开源项目和产品：WSO2 Message Broker。MB 基于 Apache Qpid 并且支持 Amazon SQS API 和 WS-Eventing。

原文链接：http://www.infoq.com/cn/news/2011/04/wso2-mb

时刻关注企业软件开发领域的变化与创新

Java — .NET — Ruby — SOA — Agile — Architecture

Java社区：企业Java社区的变化与创新

.NET社区：.NET和微软的其它企业软件开发解决方案

Ruby社区：面向Web和企业开发的Ruby，主要关注Ruby on Rails

SOA社区：关于大中型企业内面向服务架构的一切

Agile社区：敏捷软件开发和项目经理

Architecture社区：设计、技术趋势及架构师所感兴趣的话题

讨 论 组：groups.google.com/group/infoqchina
编辑电邮：editors@cn.infoq.com
广告合作：sales@cn.infoq.com

推荐编辑 | 翻译团队编辑 吴宇

大家好，很高兴能借此次《架构师》和大家见面。首先做一个自我介绍，我是来自 SOA 社区的吴宇，主要负责 SOA 社区的新闻和文章的本地化工作。目前在 IBM CDL 北亚太地区 SOA 战略合作部，从事 SOA 项目技术咨询和支持工作。

说起来在加入 InfoQ 中文站之前，关注 InfoQ 中文站已经有一段时间了。InfoQ 对业界最新资讯的介绍、各社区对其技术的深度报道以及各种深入浅出的原创文章都深深吸引了我。在一次偶然的浏览中，惊奇的发现了我现在的同事——也是我 InfoQ 的引荐人——马国耀先生的文章，当时就产生了加入 InfoQ 中文站的念头，后来经过他的推荐，在试译过几篇新闻和一篇文章后成为了一名 InfoQ 编辑。

在 InfoQ 中文站翻译的过程，其实也是学习和自我提升的过程。在这个集体里你会接触到一批热爱技术、严谨仔细又乐于分享的朋友，在线下大家会热情激烈的讨论，常常为了某个翻译细节而唇枪舌战，但目的只有一个：创建国内一流的技术社区网站，为广大读者提供高质量的翻译和原创文章。近年来，InfoQ 中文站在国内技术社区的影响力越来越大，其引入的 QCon 技术大会吸引了志同道合的朋友前来 交流讨论。在我的工作中，有机会接触公司国外一些程序员，他们大多年纪都很大，有的甚至白发苍苍，但他们对技术的热情和见解给我留下了深刻的印象。反观国内 IT 发展的时间还不长，InfoQ 中文站作为 IT 技术的布道者，我们并没有自满，路途任重道远。

最后，欢迎大家加入 Info 中文站或踊跃投稿，与大家分享一个我喜欢的签名，与君共勉：

If I have a dollar and you have a dollar;

and we exchange our dollars, we only have a dollar each.

If I have an idea and you have an idea;

if we exchange our ideas, we have two ideas each.

封面植物

七子花

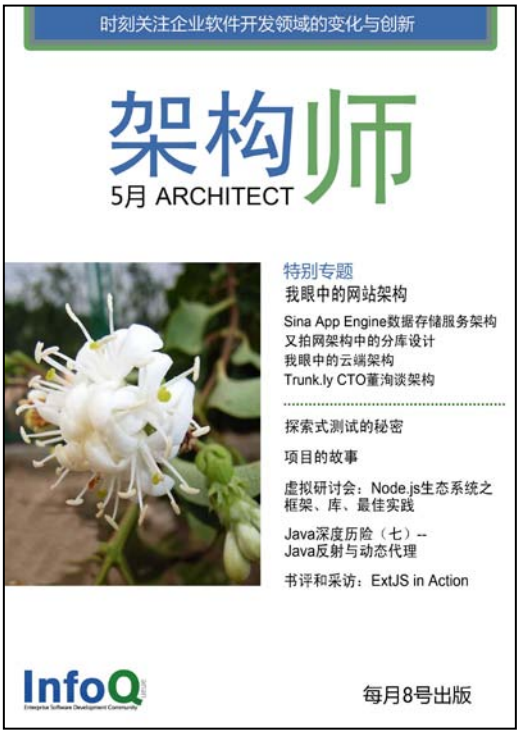
七子花，中国特有的忍冬科单种属植物，属国家二级重点保护植物，先后被列入中国被子植物关键类群中高度濒危种类和中国多样性保护行动计划中优先保护的物种。

生长分布：本种分布于丘陵低山。分布区雾多而凉爽，年平均温 15.4--16.2℃，1 月平均温 2.9--4.4℃，7 月份平均温 27.4--28℃，极端最低温-6.1℃，年降雨量 1278--1547 毫米，雨日较多，湿度大，夏季相对湿度常在 90%左右。土壤由红壤逐步过渡到黄壤，酸性反应。常生于低山坡、山沟溪边疏林灌丛中或毛竹林边缘，很少长在山顶与山脊灌丛中。主要分布于湖北、安徽、浙江的大盘山、北山、天台山以及泾县、宣城等地区，在模式标本产地——湖北兴山已不存在七子花了。

植物形态：落叶小乔木，高达 7 米；树皮灰褐色，片状剥落；幼枝略呈四棱形，红褐色。叶对生。圆锥花序顶生，长达 15 厘米。由多数密集呈头状的穗状花序组成；穗状花序有 12 轮，每轮有 7 朵花，包括 1 对有 3 朵花的聚伞花序和 1 朵顶生的单花，外面包有 10 一 12 枚鳞片状苞片和小苞片，小苞片各对形状大小不等，最外一对有缺刻；萼筒长约 2 毫米，被白色刚毛，萼齿 5，长圆形；花冠 白色，稍芳香，筒状漏斗形，外面密生倒向短柔毛，裂片 5，近唇形；雄蕊 5；子房下位，3 室，仅 1 室能育。果为瘦果状核果，长圆形，长 1 一 1.5 厘米，外具 10 条纵棱和疏生糙毛，冠以宿存而增大的 5 萼裂片，裂片紫红色七子花姿态优美，花期长；树干洁白、光滑，可与紫薇媲美；花形奇特，花色红白相间，繁花集于长花序，远望酷似群蜂采蜜，甚为奇观。

保护价值：七子花是单种属植物，为中国特有，对研究忍冬科系统发育有科学价值。又是优良的观赏树木。叶片、花等器官提取物具有抑菌性。

1kg.org
多背一公斤
爱自然 | 更爱孩子



架构师 5 月刊

每月 8 日出版

本期主编：胡键

总编辑：霍泰稳

美术编辑：胡伟红

读者反馈：editors@cn.infoq.com

投稿：editors@cn.infoq.com

InfoQ 中文站新浪微博：
http://t.sina.com.cn/infoqchina

商务合作：sales@cn.infoq.com 13581658359

本期主编：胡键，InfoQ 中文站翻译团队编辑

胡键，热心开源技术，《开源技术选型手册》SOA 部分作者，《SOA 实践指南--应用整体架构》第一译者。目前致力于 Groovy/Grails 的研究和推广，InfoQ 的迷你书《Grails 入门指南（第二版）》译者，创建了关注 Groovy 社区动态的中文站点 GroovyQ：http://www.groovyq.net，欢迎大家前往交流。其在 InfoQ 的编辑特辑为：胡键。

InfoQ
Enterprise Software Development Community

《架构师》月刊由 InfoQ 中文站出品。

所有内容版权均属 C4Media Inc.所有，未经许可不得转载。

时刻关注企业软件开发领域的变化与创新

架构师

www.infoq.com/cn/architect

每月8号出版

InfoQ neue
Enterprise Software Development Community

商务合作：sales@cn.infoq.com
读者反馈/投稿：editors@cn.infoq.com