

架构师

2月 ARCHITECT



特别专题

Go语言

Go语言开发工具LiteIDE

界面漫谈

Go，基于连接与组合的语言（上）

Ruby 20年

欣欣向荣的Ruby家族

深入理解Java内存模型（一）：基础

Node.js分布式聊天服务器搭建

预防Web应用程序的漏洞

敏捷自动化测试

卷首语

还记得上学的时候，经常会和同学们开玩笑地说：贪污浪费，都是犯罪。

最近，一股“反对浪费”的潮流席卷国内很多企业，结果很多人措手不及，本想在年会上看看节目、吃吃饭，再领个奖开心回家，却只能是节目白练了、场地白定了、奖品白准备了，因为年会将造成严重的浪费，所以必须取消，但是却没有想到取消年会本身同样会造成不少浪费。

软件行业中不可避免也有很多浪费，在传统的软件工程模式下，一旦项目前期的环节——像需求分析、概要设计等——出现了一点儿浪费，那么在后期就会被十倍乃至上百倍地放大，广大程序员付出了大量的时间和精力得不到回报，进度也被大大拖延。

一个团队的组织结构图中会有很多种角色，有人认为真正有实际产出的只有程序员，而项目经理、产品经理、质量保证人员等等角色的存在也是一种浪费。管理人员与生产人员的比例，的确是一个值得思考的问题，处理不当，就会造成人这种宝贵资源的浪费。

正是为了消除浪费，敏捷运动应运而生，去除软件系统开发中不必要的角色、不必要的环节、不必要的文档，一切都要尽可能地精简，为了软件系统这个核心而服务，这也取得了相当不错的结果。很多小团队正是因为采用了敏捷的实践，才避免了很多浪费时间和精力的工作，从而在最短的时间内完成最需要的任务，取得成功。

精益方法的核心思想就是要消除浪费，这种思想的应用已经走出了丰田的汽车制造车间，广泛地应用在了很多软件开发团队之中。而且针对初创企业，更是兴起了“精益创业”的做法。快速完成最核心的产品，尽快推向市场，尽快获得用户的反馈，尽快改善，然后不断重复这个循环。这样就避免做出用户不想要的功能，而把更多的资源应用在最核心的内容之上。消除了浪费，初创企业才能够在只具备少量资金和资源的情况下，和那些巨无霸级的企业竞争一下。

由此看来，减少浪费已经成为了人们的共识，也是通向成功之路上所必须要做的一件事。所以，广大程序员朋友们，让我们珍爱生命，远离浪费。

本期主编 侯伯薇

目录

卷首语

人物专访

采访：关于 Go 语言和《Go Web 编程》	6
-------------------------------	---

热点新闻

12306.cn 反制浏览器代码被指“傻大黑粗”	11
devsigh，程序员的一声叹息	14
什么样的登录框才优秀？	17
今时今日，C 还适合当下之所需么？	20
12306 订票助手插件拖垮 GitHub 事件原因始末	26

特别专题

Go 语言开发工具 LiteIDE	31
特别专题	42
界面漫谈	42
热别专题	48
Go，基于连接与组合的语言（上）	48

本期专栏

欣欣向荣的 Ruby 家族	58
---------------------	----

推荐文章

深入理解 Java 内存模型（一）——基础	62
深入浅出 Node.js 游戏服务器开发--分布式聊天服务器搭建	70
推荐文章	81
预防 Web 应用程序的漏洞	81

软件开发生命周期中的安全性.....	82
敏捷自动化测试（2）：像用户使用软件一样享受自动化测试	94

新品推荐

Yeoman：适合现代 Web 应用的现代工作流	103
亚马逊推出 Simple Workflow Service 手册	103
Node.js 包含新的流 API	103
用于展现图表的 50 种 JavaScript 库.....	103
来自荷兰格罗宁根大学的架构决策捕捉工具 RGT	104
新版《Scrum 启动规划书》旨在帮助敏捷团队步入正轨	104
Vert.x 计划加入 Eclipse 基金会	104
使用 Visual Studio 3D 初学者工具包(Starter Kit)进行 Windows 应用商店游戏开发	105
最新的技术雷达趋势	105
沃尔玛实验室开源项目一览	105
推荐编辑 方腾飞	106



促进软件开发领域知识与创新的传播



中文 | 英文 | 日文 | 葡文 |

人物专访 | Interview

采访：关于 Go 语言和《Go Web 编程》

作者 侯伯薇

最近，在网上出现了一本名为《Go Web 编程》的书籍，里面详细地讲述了使用 Go 语言进行 Web 编程的各个方面。很特别的是，这本书是在 GitHub 上以开源的方式撰写的。日前，InfoQ 采访了这本书的作者谢孟军先生，请他来和大家谈谈 Go 语言以及他撰写的开源书籍。

InfoQ：请您先简单和大家介绍一下自己。

谢孟军：大家好，我叫谢孟军，可以叫我Asta，目前就职于盛大云，负责盛大云分发产品的研发。云分发就是我们平常所谓的CDN系统。

InfoQ：是什么原因促使您撰写《Go Web 编程》这本书呢？

谢孟军：我以前主要从事PHP、Python等Web方面的开发，后来到了盛大云之后接触比较多C++等底层的东西，就想学习一下这些底层的技术。再后来看到许式伟的博客，了解到Go语言。Go出自B语言的作者和Unix创始人Ken Thomson之手，而且还有多位牛人在后面支撑，该语言设计的初衷就是系统级别的语言。这正好符合我学习的目标，所以我就开始了自学之路。自学过程中发现Go在网络编程这一块为开发者提供了很多便利，既有C等语言的性能和排错能力，又具备Python等语言的动态特性，还内置了对多核并发的支持。

在学习的过程中，我发现Go在网络编程方面的表现非常好，而我以前主要从事Web开发，所以就想写《Go Web编程》这样一本书，主要还是把以前在Web开发中的一些经验和Go语言结合起来，做一些总结，给后来者一些启示，希望对想学习Go语言的同学们一些帮助。

InfoQ：您撰写的这本书在 GitHub 上开源，为什么采取这样的一种方式呢？

谢孟军：采用GitHub主要是我之前看到过一本书叫做《[PHP扩展开发及内核应用](#)》，这本书就是在GitHub上面写的，我看到之后很震惊，居然还可

以这样来写书，而markdown又是如此的简洁，GitHub对于[markdown](#)的支持也非常好，所以当初就决定在GitHub上面来写这本书。

InfoQ：Go语言的哪些特点最吸引您？哪些特点让您觉得需要进一步改进？

谢孟军：Go语言吸引我的主要有如下几点：

1. 它是系统级别的语言，静态编译，是C系列语言。
2. 具有很多内置库，使用起来和Python很类似。
3. 语法足够简单，入门学习成本很低，适合我这样从PHP和Python切换过来的人。
4. 速度快，就拿简单的页面来说，我用PHP开发并发能够达到500很好了，但是用Go轻松就到上万，这是无法比拟的性能提升，而且用Go开发的效率和PHP差不多。
5. 出自Google之手，而且有一帮牛人在维护，基于BSD开源，社区活跃。
6. 开源项目给我很多自信，举几个开源系统：[vitess](#)（YouTube的数据库proxy系统）、[nsq](#)（bitly的一个实时信息处理系统）、[skynet](#)（轻量级的分布式服务框架）、七牛公司全部用Go开发、360开发的类iMessage应用，支持上千万用户，同时单台服务器长连80w，这些系统都是他们线上跑的，这给我更大的信心能够用Go来开发高性能，高稳定的应用。

主要的缺点：

1. 有些库不成熟，例如图像处理。
2. cgo在Window系统下面编译很麻烦，就拿SQLite的数据库驱动来说，在Window下面编译就会遇到很大的麻烦。
3. runtime还不够成熟，GC还不是很好，不过听说Go 1.1版本会有比较大的性能提升。
4. Go的开源项目还不够多。我觉得一个语言的发展不仅仅是语言本身出色，还要有大公司推动或者好的项目推动。

InfoQ：有人说Go是互联网时代的C语言。与之相比，Go语言有哪些优势，又有哪些需要改进的地方呢？

谢孟军 : Go语言确实吸收了很多C语言的特性，Ken大叔毕竟是Unix的创始人，很多以前看C不习惯的地方，这一次Go里面都改进了，例如{}的换行，()的去掉，强制格式化、去掉结尾分号等。而为什么Go被称为互联网时代的C呢？我认为是Go在语言级别上支持了并发，通过简单的关键字go就可以充分利用多核，这对于硬件不断发展的时代，这么简单就可以充分利用硬件的多核，这是多么重要的一个特性啊！但是相比C而言，Go还缺少一些高质量的第三方包，例如OpenGL等，所以Go内部也支持用cgo直接调用C语言编写的代码。

InfoQ : 有人认为 Go 语言会在以后追赶上 Java。与之相比，Go 语言有哪些优势，又有哪些需要改进的地方呢？

谢孟军 : 我的感觉，Go要赶超Java还不知道要等到什么年代。首先，Java目前在企业应用中非常稳定，而且基于Java的应用很多，库也多，我想Java也会有很多基于JVM的类似语言出现，例如Scala，我想Go要超越Java可能性不大。但是Go相比Java来说优点也是很明显的：简单，足够简单。记得以前有一篇Go创作者Rob写过一篇文章《大道至简》，就是讲为什么创建Go语言的过程。我想这一批老程序员的经验是值得我们学习的，就让我们用Go语言把复杂的事情简单化，简单的事情简单做，Go也许就是这样一种语言。

InfoQ : 您现在是否已经在项目中大量应用 Go 语言，主要在哪些方面呢？

谢孟军 : 我现在开发的项目基本都是用Go开发的，但是页面上我还是用JavaScript来渲染，我现在的开发模式是JavaScript渲染前端+RESTful API，中间用JSON数据进行交互。采用这种模式开发主要有几点原因：

我认为JavaScript渲染页面方面比Go的模板做的好很多，而且JavaScript现有的库非常多，很容易做出很多效果，我的理念就是不管白猫黑猫，会抓老鼠就是好猫，合适的语言用在合适的地方。

Go开发API相當地快速，而且性能很高，是PHP、Python之类的不能比拟的。

设计成RESTful API的话，系统就比较容易扩展，扩展性比较好。

目前内部的短域名服务采用了Go+redis开发，视频直播调度器全部采用了Go开发，内部的系统监控和自动化运维系统采用了JavaScript+RESTful API的形式开发，还有其他一些边缘项目都是采用Go开发。

同时我还开发了两个开源的项目：

[beego](#)：一个模仿Python的tornado系统开发的Go开发框架，现在开发的几个系统都是基于该框架开发。

beedb：一个Go语言的ORM库，可以像操作struct一样操作数据库数据。目前我们内部的API接口我就是采用了这个ORM开发的。

InfoQ：您认为 Go 语言在 Web 编程方面，哪些特点会让它更有优势？

谢孟军：前面也说了Go语言设计的时候是系统级别的语言，所以他本身就有性能上面的优势。其次Go在Web开发中内置的net/http包对于开发Web非常方便，用户可以很方便的就搭建一个Web应用。熟悉PHP的同学可能对于nginx+fastcgi配置都很熟悉，但是Go开发的应用就不需要nginx，因为它自己就可以监控网络，解析数据包，而不依赖任何东西，你编译完之后扔到服务器起来就好了，这省去了一些部署的部分。最后就是Go的并发支持，大家都听说过摩尔定律，硬件只会越来越快，CPU的核数也会越来越多，那么Go的这个特性就让我们这些程序员从以前的多线程处理中解放出来，让Go语言的runtime来帮我们做这个事情，那用使用Go来编写Web何乐而不为呢？

InfoQ：对于开始学习 Go 语言，并期待将其应用在项目中的读者们，您有哪些建议呢？

谢孟军：刚开始学习Go语言的朋友，我建议读一下官方的一些文章，都非常有帮助，如果你对英文敏感的话，那么就看看我写的那本书《Go Web编程》。

在你应用Go到你的项目之前，希望你能够对于Go的特性有一定了解。目前Go语言的特性基本已经确定，接下来的版本都是基于性能的提升，新特性估计不会有。然后，建议大家多读读Go内置的包源代码，因为这些包都是Go语言开发的，对于我们编写Go代码又很大的帮助，看看这些牛人们是怎么编写代码的。有句话说得好：学习牛人，复制牛人，超越牛人，最后我们自己成为牛人。

最后还是希望国内的技术人都学习一下Go语言，不管它现在对你有没有用，但是我可以肯定的说将来肯定有用，对于你开发项目时有扩展思路的作用，我现在也在学习Node.js、Erlang，我学习他们是为了更好的开发Go的项目，借鉴其他语言的一些特性。

原文链接：

<http://www.infoq.com/cn/articles/go-web-programming-interview>

相关内容

- [Google Go 语言推出第一个正式版本：Go 1](#)
- [基于用户反应，Google 调整 GAE 价格条款](#)
- [互联网超级云计算平台](#)
- [Rob Pike 谈 Google Go：并发，Type System，内存管理和 GC](#)
- [H.265 Codec Standard 已获批准](#)
- [李开复谈 Facebook 图谱搜索](#)

热点新闻 | News

12306.cn 反制浏览器代码被指“傻大黑粗”

作者 [郑柯](#)

1月28日，已经是2013年春运正式开始的第三天，那寄托着异乡游子急切心情的粉红纸片，依旧一票难求。

前两天发生的[12306订票助手插件拖垮GitHub事件](#)，风波仍未过去。铁道部为了保证“公平”，他们开始采取技术手段，针对使用插件的浏览器采取反制措施，其结果如何，目前看来，恐怕不能令他们和他们的上峰满意。

网友[码农1999](#)在新浪微吧上发布文章《[傻大黑粗+殃及无辜：12306技术反制猎豹浏览器代码级分析](#)》，列出了12306.cn的具体反制措施，其中主要有四种手段。

手段一：频繁修改具体功能函数名称，让抢票插件调用失效。

这是12306开始技术反制时最常用的小伎俩。码农都知道，这种“小修小补”的技术屏蔽手段，只要针对12306修改的具体功能函数做修改，即可实现插件正常使用。整个破解及插件更新时间不超过1小时。

手段二：1月20日开始屏蔽浏览器 user agent

1月20日进入春运购票最高峰（可订2月8日车票），12306使出了屏蔽浏览器user agent的手段，如此一部分没有用抢票版的用户也被屏蔽，一些用户反应未用插件却登陆12306时白屏，进入不到网站。

针对屏蔽 user agent，只要修改浏览器的用户代理即可实现登陆12306。

手段三：限制连续2个操作的时间间隔，所有浏览器在未使用抢票插件情况下或无辜中枪。

如在提交订单时12306判定输入验证码及提交的时间间隔短，即使验证码输入正确，亦会被12306弹出“验证码错误”。此种屏蔽手段造成的后果是：
1、抢票插件自动提交订单失效；2、无论使用哪种浏览器的用户，无论使不使用抢票插件，只要“被判定”时间间隔短，即提交订单失败...破解此屏蔽

手段，甚至不需要修改插件代码，只需取消抢票插件的“自动提交”功能，手动输入验证码即可。

手段四：12306后台增加一些逻辑判断，会踢出用插件用户的登录状态。

这些逻辑判断包括：

限定操作时间间隔、提交订单时间间隔、查询刷新间隔等。

这些手段均不能保证只屏蔽抢票插件用户，均有可能让无辜的用户，被12306判定为“插件用户”，不能正常登陆，买不到车票。

对于这些反制措施，作者认为：

1. 12306用的那些技术反制手段真心很糙，很临时工。一个词形容：傻大黑粗。
2. 12306这些小花招，在浏览器厂商面前，效果不大。
3. 12306的反制措施还带来更大的网站崩溃可能性，殃及无辜。

文末，作者给12306提出4句话：

1. 每一个登陆12306的用户，都有权利选择使用的浏览器及正当手段提升使用体验，屏蔽只会加剧无谓的劳动量。
2. 请不要再聘用实习生或者临时工进行屏蔽抢票插件，请尊重程序猿的智商。
3. 12306对“无辜”被屏蔽的用户造成极大的不公平。
4. 12306作为投资5亿的网站，请亮出你技术屏蔽的“杀手锏”。

在微博评论中，[Super_Wang](#)指出：

用技术手段屏蔽标准http请求。。。无语，只要它没离开http，找出规律也就是个时间问题。对关键请求加单次有效的短信延时二次认证应该是个不错的办法，客户体验就难说了

[执信飘飘](#)想听听其他人的看法：

大量游戏外挂都不能得到有效解决，何况这是一个甚至没有做验证码适配的脚本程序而已，个人觉得这的确很难做处理。因为不具有很明显的特征性，很难判断。

InfoQ 的读者们，你们遇到过被屏蔽的情况吗？如果有，使用的是哪些浏览器？InfoQ 中文站会继续关注事态后续发展。

原文链接：

<http://www.infoq.com/cn/news/2013/01/12306-anti-browser-failed>

相关内容

- [SoundCloud 前端技术团队分享开发经验](#)
 - [Twitter 开源了基于事件的组件框架 Flight](#)
 - [从可编程到跨终端——QCon 北京 2013 “跨终端的 Web” 专题出品人鄢学鵠专访](#)
 - [Node.js 包含新的流 API](#)
-

热点新闻

devsigh，程序员的一声叹息

作者 [郑柯](#)

程序员，作为以技术谋生存、求发展的群体，常常要面对来自其他不同人群种种看似不合理的需求。类似情况下，程序员所能做的，只有发出一声叹息。

新近出现的 devsigh.com，就是程序员们分享这些叹息的网站。

让我们看看程序员们都有哪些叹息。

本月最佳叹息是：

客户：我很确定，我们在登录框中不需要密码。

所有的叹息中，最佳的前五条是：

第五条：

今天下午在Skype上：

<同事>：这段[bash代码]没问题吧？

<我>：我不知道，你运行的时候啥情况？

<同事>：噢，我还没试过呢。

我不是你的SHELL，滚。。。

第四条：

PHP技术公司的CTO：我们要把所有的MySQL换成SQL Server。

开发人员：这么做背后的原因是什么？现在一切都很正常，SQL Server太烧钱了.....

CTO：MySQL不够“企业级”。

第三条：

[老板]：我要你在这里加一个cookie，【他指着屏幕】

[我] : 啊~~~什么 ?

[老板] : 在这里放一个cookie。【又指着屏幕】

[我] : 呃.....

[团队主管] : 他是说下拉列表。

第二条 :

你在奋力解决一个复杂、困难的问题。

在互联网上找不到帮助 , 你最终决定使用StackOverflow。

把问题拆成核心问题 , 然后抽象出所有特定实现。

因为问题很难 , 而且不是JQuery方面的 , 没人搭理。

放弃了 , 你试着再多Google一次。

第一条搜索结果就是你的问题。

第一条 :

总裁 : 我们不需要两台服务器。

我 : 但是我们需要冗余啊。

总裁 : 搞sir才需要两台服务器。优秀的团队不会让唯一的一台服务器死机。

我 : 唉.....

如果你想去发出你自己的叹息 , devsign 网站[给出如下提示](#) :

你的故事 , 越短越好。没人想读长篇大论。读长篇大论 , 太像是在工作了。

如果我们想工作 , 就不会拖拖拉拉、阅读这样的网站了。

不要提及任何公司或人的名称。如果你这么做了 , 我们可能要删帖 , 那是工作 , 我们讨厌工作。

小段的疯狂代码我们欢迎。不要贴出任何机密或是可能给你带来麻烦的代码。

在对话里面 , 试着用Foo指代名字 , 而不是<name>Foo。

InfoQ 的读者们，你在自己的工作中，遇到或者发出过哪些叹息？不妨去贴在 devsgih.com 上。或者，干脆你去做一个中文版的 devsgih ? 不过别把它当成工作，因为我们讨厌工作!

原文链接：<http://www.infoq.com/cn/news/2013/01/devsigh>

相关内容

- [Facebook 已将 HHVM/JIT 用于其开发和产品中](#)
- [arrayDB，全新而且简单的 PHP ORM 库](#)
- [WebMatrix 2：新模板、智能感知改进、Windows Azure 集成](#)
- [专访 SegmentFault 开发团队：垂直问答社区的架构升级](#)

热点新闻

什么样的登录框才优秀？

作者 侯伯微

在绝大多数应用程序中，都会存在登录框这种组件，而登录框或者登录方式的设计也千差万别。那么，什么样的登录框才称得起“优秀”两个字呢？最近 [torres 在知乎上提出了这个问题](#)，引起了大家广泛的讨论。

torres 在提出这个问题之后，先做了一下“自问自答”，提供了他自己在豆瓣上同标题[日记](#)的链接，其中对几大 SNS 和社区类网站——包括新浪微博、腾讯微博、网易微博、淘宝网、凡客诚品、街旁网、嘀咕网、知乎和 37signals——的登录框做了比较，得出的结论是：优秀的登录框需要简单明了的告诉用户“我该怎么做”，按钮需要重点突出。他还具体总结了三点：

1. 紧密性——界面上的文字按内容清晰分开，避免用户阅读混淆。
2. 复用——复用同一种颜色或者素材，能够使网站风格保持一致，从而引领用户去点击。
3. 对齐——必须条件。

[向翔](#)也根据自己的经验，给出了自己对于优秀登录框的看法：

1. 在合适的时候出现。至少登录过就应该不出现，不抢视觉焦点。
2. （需要用户付出的）时间短，包含几个方面，用户输入的少，tab顺序，回车响应，但很多大网站不注意的一个细节是用户在切换输入法，这个对于用户来说比较麻烦，特别是在输入验证码的时候，这个问题在[百度统计的登录界面](#)显示得特别不友好。（用户名是中文，tab到密码框式默认屏蔽输入法，但到了输入法界面就成了纯英文，但当前输入法还是中文）。验证码要容易辨别，经常看到不少验证码，后台不做智能处理，一定要让用户肉眼来区分O和0，1和l，这些都是让用户最容易输入错误的，都应该在后台屏蔽，允许用户输错o和0.
3. 安全性，使用https，使用验证码，如果能和谷歌一样，自动检测危险，必要时才加入验证码是最好不过。

4. 不要和老版新浪一样，一个礼拜没有登录访问你的页面的时候就变成注册引导框了，明明是用户想登录，却跳转来注册界面。
5. 多种登录方式，支持大部分联合登录，登录相关链接都出来，比如找密码，注册，登录几次不成功后的提示。
6. 登录成功后跳转到该去的地方。特别是有些网站记录的是referrer，导致用户还需要交互一次。比如购物车页面去结算提示登录，登录完又跳到购物车页面了，这就是问题，用户还要点一次。

雷鸣又补充了几点：

1. 支持一键清除，对大写状态要给出提示
2. 不管是什么阅读习惯，登录页只做登录页，啥也不要放了，弹出浮层也不错，总之登录页的焦点只有一个。要想焦点只有一个，就是除了登录功能，别的都不放。
3. 邮箱≠用户名，什么方式登录就提示什么，有几种方式就提示几种，不要多也不要少。多次听到"邮箱就是用户名"这样的傻缺想法，金山词霸登录页中使用邮箱登录，可前面提示却是用户名，一段时间没用后，再登录的时候第一反应八成是输入用户名。
4. 如果使用邮箱登录的话，还是把输入框做大点吧。

Rubi 的想法认为更应该关注的是登录方式：

最好的登录框是没有框。

我的思路不针对现在，我想要的登录框是一个可以多种方式登录的。它含有的功能为：

- 基础输入登录，就是键盘输入。
- 脸部识别登录，扫描我一下就要以登录。
- 声音识别登录，喊一声，就可以登录了。
- 指纹识别登录，扫描一下指纹就可以登录了，现在很多公司都是这样。但识别不高。

石孟宸也表达了自己对最好登录方式的想法：

登录为何要“框”，这明显是拿“技术模型”去映射“心理模型”，而且在天天谈用户体验的今天还没有根本性改变。你可知为何app如此好用，就是因为app默认记住了N多的用户名与密码，扫清了用户与产品的第一道门槛。

这就如同开门要拿着钥匙找锁眼，试问谁明白锁芯是如何工作的？为何要使用如此不人性化的东西。

所以最好的登录方式，应该是“设备主动识别人”。

目前来讲我觉得比较好的是IBM的指纹识别系统。

从讨论中我们可以看出，登录框并不是一个简单的、可以随意设计的组件，而是需要考虑到很多方面。

盛大创新院的[庄表伟](#)也曾经在 QCon 的演讲《[开放平台时代的登录系统设计](#)》中讲述过他在登录框设计方面的经验。

各位 InfoQ 的读者们，你们在日常的工作和生活中也一定少不了和登录框打交道，并且也可能设计过各种系统的登录框，那么，在你的心中，什么样的登录框才优秀呢？欢迎在此分享和讨论。

原文链接：<http://www.infoq.com/cn/news/2013/01/good-login-design>

相关内容：

- [用于展现图表的 50 种 JavaScript 库](#)
- [豌豆荚余奕霖：Holo 界面与产品测试](#)
- [微软通过 Kinect for Windows 加速器计划助力开发者](#)
- [自我评价 Windows 8 应用的用户体验](#)

热点新闻

今时今日，C 还适合当下之所需么？

作者 [Abel Avram](#) 译者 [张龙](#)

来自 Couchbase 的 Damien Katz 认为 C 依然非常适合于后端编程的一门语言，然而有的开发者则觉得 C 有太多的瑕疵，他们支持 C++ 或是 Java，还有一些人连这两种语言也不喜欢。

在最近一篇题为 [The Unreasonable Effectiveness of C](#) 的博文中，[CouchDB](#) 的创建者 Damien Katz 表示 C 依然非常适合于后端编程的一门语言，虽然现在已经有了很多更加现代化的语言，如 C++、Java、甚至是 Erlang 或是 Ruby，但他还是非常支持 C 语言。Katz 并不是认为 C 就是比其他任何语言都要好，但在“重点考虑性能与可靠性等场景下，C 是很难被打败的”——这段话援引自 Damien Katz 随后的一个帖子，旨在[澄清自己的立场](#)。

虽然一开始使用 Erlang 编写了 CouchDB 的很多代码，但在花费了“2+个人月来处理 Erlang VM 中的一个崩溃问题”之后，Katz 感到非常不爽。

我们浪费了大量时间追踪核心Erlang实现中的一些问题，不敢保证发生什么以及原因，我们觉得也许问题出现在我们自己的插件C代码中，希望我们自己能够发现并修复问题。但事实并非如此，这是核心Erlang中的一个竞态条件Bug。我们只能通过Erlang的代码查看器来找到这一问题。对于那些对计算机进行了过多抽象的编程语言来说，这是个很基本的问题。

出于这一点以及性能因素，Katz 决定逐步重写，“将 Couchbase 的代码换成 C，并将其作为大多数新特性的首选实现语言”。有趣的是，C 证明了“当我们遇到问题、调试与修复问题时，C 更具备可预测性。长远来看，C 的生产力更高”。

Katz 列出了对于后端来说 C 要优于更高层次语言如 C++、Java 等的若干原因：

- **表现力**—— “C 的语法与语义非常强大且具有极强的表现力。凭借 C，我们既能预测高层算法，又能预测低层的硬件。它的语义非常简单，语法足够强大，能够极大降低认知上的负担，让程序员专注在重要的事情上”。
- **简单**—— “C 是一种弱、静态类型语言，其类型系统非常简单。我们所说的弱最后会变成一个优点：C APIs 的“表面”是非常简单且小巧的。相对于大

多数框架来说，C 的一个明显趋势与文化就是创建小型的库，对简单类型进行轻量级的抽象”。

- **速度与内存使用**—— “C 是速度最快的语言，无论是部分还是完整的基准都表明了这一点。它不仅仅运行时是最快的，其内存使用与启动时间也是效率最高的。如果需要在空间与时间上进行折衷，那么 C 并不会对你隐藏任何细节信息，我们可以很容易地做出估计”。
- **更快的开发周期**—— “对于开发者效率与生产力来说最为重要的就是‘构建、运行与调试’周期。周期越快，开发的交互性就越好，你就更容易处在任务的流态上。相对于所有主流的静态语言来说，C 拥有最为快速的开发交互性”。
- **调试**—— “对于纯 C 代码来说，你可以查看调用堆栈、变量、参数、线程局部变量、全局变量，基本上可以查看到内存中的一切。这是非常有用的，特别是当你遇到了问题，这个问题在运行的服务器进程中出现了多日，并且无法重现的情况下更是如此。如果在更为高层的语言中失去了这个上下文，那么你就等着痛苦去吧”。
- **跨平台**—— “有一个标准化的应用二进制接口（ABI），可为现有的所有操作系统、语言与平台所支持。它无需运行时，也不需要其他额外内容。这意味着你使用 C 所编写的代码并非仅仅可由 C 代码中的调用者所调用，还可以由现有的所有库、语言与环境所调用”。

Katz 也认为 C 有“很多瑕疵”：

没有范围检查，不小心就会导致内存出现问题、存在野指针与内存/资源泄露情况、对并发的附加支持、没有模块、没有命名空间。错误处理非常麻烦且冗长。很容易就会搞出一堆错误，调用堆栈也找不到了，恶意输入会控制你的进程。闭包？哈哈

Katz 对 C 的强烈偏爱源自突破 Couchbase 性能极限的需求以及调试问题（问题是 C 插件与 Erlang VM 联合使用所导致的）。他并不认为 C++、Go 或是 D 能够替换 C，但他认为 [Rust](#) 可能会成为“梦想之语言”，只要它能实现“类似于 C 的性能而且能够做到与 Erlang 安全的并发和内建的健壮性”。

Katz 的帖子在 [Reddit](#) 与 [Hacker News](#) 上可谓是一石激起千层浪，有很多开发者谈到了 C 的优点，也有人建议其他语言。[robinei](#) 加入到了[字符串操作](#)与错误检测激战当中：

我总想回到C（从C++等语言中），当我真的这么做了时，我发现不少地方通常都是很简单的，感觉真棒！

但接下来我需要进行字符串操作，或是这类笨拙的方法。

这时会出现很多分配操作，每一个都需要一个显式的free搞得我太痛苦了。

我尝试通过arena allocators树来解决这个问题，就像Go中的slice-strings，但最终C还是缺乏一些语法工具（命名空间前缀函数），这导致结果变得非常笨拙（将一切都分配到arenas中也是非常痛苦的）。

我发现由于要进行显式的错误检测，源代码文件长度增加了一倍（这种情况不常发生，但在诸如sqlite等一些库中，任何操作都有可能失败）。

还有很多方面导致我精疲力竭，我觉得非常不满意。

综上所述，我从哪儿来的还是回哪儿去吧，通常是C++。

[madhadron](#) 提出了“更加现实的 C”：

C能够在PDP-11上很直接地编译成快速的机器码。

C的标准库就是个笑话。它的缺点，特别是字符串相关的处理，是过去40年众多安全漏洞的罪魁祸首。

C的工具根本不值得吹嘘，特别是与同辈的Smalltalk和Lisp相比。人们所使用的大多数C调试器都是命令行。根本没法和Squeak或是Allegro Common Lisp的标准调试器相比。

声明快速的C构建/调试/运行周期令人沮丧。表面上看起来很快，因为C++在这个领域是失败的。如果你想知道如何加快构建/调试/运行周期，那么请看看Turbo Pascal吧。

你可以通过标准ABI在所有Unix上调用C，虽然这么说没错，但原因却是因为C的普遍存在性而已。

[geophile](#) 对上述内容持不同看法：

C/C++/Java，这是程序员视角的石头/剪刀/布。

多年以前，我从C开始，我自己用宏和库为声明与函数提供了很多很有用的组合。我发明了对象，同时也发现了C++。

长久以来，我一直是个快乐的C++用户，很早就开始了（还是cfront时代）。但我被语言的复杂性搞崩溃了，特别是特性之间微妙的交互，我厌倦了内存管理，渴望Java，而它又适时地出现了。

我很开心。在学习语言时，我肯定我漏掉了某些东西。每个对象都在堆中么？

真是如此么？真的没有办法将一个对象在物理上嵌入到另一个对象中么？

但其他一切都很棒，我不介意这一点。

现在我在编写一些系统，这些系统会占用很多内存，包含成千上万的对象，有些对象很小，有些则很大。每个对象的代价快要搞死我了。GC调优是个梦魇，我正在实现子分配模式。我编写了微基准，比较普通对象与序列化为字节数组的对象。由于C++已经变得太恐怖了，比令我焦头烂额的早期版本还要复杂，因此我又渴望C了。

现在的我不再喜欢任何语言了。

对于某些人来说，C看起来瑕疵太多，已经不适应现代的生产力要求了，但还有不少人依然能够很好地使用C，尽管它有很多怪癖。开发者社区还是应该避免语言之争，而是更好地权衡每一种语言，根据项目需求与自身技能选择最合适的语言。毕竟，没有一种语言是完美的。

此文在InfoQ英文站也引来了众多读者的讨论，下面摘取部分读者的评论供大家参考。

读者Mark Peskin说到：

嗯，我喜欢C。它很简单，没有C++那些庞大且有瑕疵的面向对象特性。然而，使用C编写大型、可扩展、模块化的企业应用需要大量规则，这些规则在大型的软件企业中几乎是无法维护的。我认为C的一个问题是它会引诱聪明的开发者编写高度优化的代码，充满了memcpy()与指针运算，以此“打败编译器”，这对于其他开发者来说几乎无法理解（如果不信，你去读读BSD内核代码吧）。

总的来说，如果有很多开发者在开发大型、复杂且长期的项目，那么我认为你最好使用Java（或是Scala等语言）。将C用在那些偶尔出现的场景中吧，这时你可能真的需要本地代码，使用基于消息的系统将二者集成起来（JNI不行）。C++？还是算了吧。

读者Josh Long说到：

回应Katz关于C的问题。

我同意Mark的观点，在某种程度上，也认可Damien的看法。

Damien提到的一点是我们很少在C中看到真正大型、全面的框架，比如APIs。如果构建小型、某个方面的API（通过一些typedefs/structs与函数作为

“契约”），那么我们可以很轻松地将库“导出”并重用。我觉得这是C最适合之处。我从来不会因为性能问题而使用C，但使用C实现某些功能则是更为轻松的事情（内核编程、嵌入式编程、处理硬件、所依赖的APIs并未在更高层次的语言如Java、Ruby、Python等进行过抽象的功能）。

我还尽量不使用C编写具有完整功能的系统，只是因为它对于大型项目来说不具备“可伸缩性”。使用C编写的大型项目最终的结果都是重新编写了很多东西（比如说对象与命名空间等）。恕我直言，真的没有多少领域需要从头到尾都用C不可。当然了，一些例外是系统级组件，比如说操作系统（Linux）或是UI，如GNOME。但对于应用来说，使用更高层次的语言，在高层次语言与平台之间存在缝隙之处再使用低层次APIs进行集成是更容易的做法。Java存在很多这类“缝隙”，但随着APIs逐步成为很多不同操作系统上的常客后，在过去10年间，有些已经被逐步解决了：事件驱动的IO、文件系统通知、文件权限与元数据等等。

Mark为集成C库与模块提出了很好的解决方案。他认为JNI不行，建议使用消息。我是消息的忠实粉丝。本质来说，成功使用消息与成功使用JNI都需要同样的东西：你需要彻底简化导出API。

在使用JNI时，绝不要将任何复杂的C类型“泄漏”到我的Java API中，反之亦然，并且总是通过数值类型与char* -> jstrings进行通信。即便我所公开的本地代码是用C++编写的，我也依然会使用C风格的JNI（而不是C++），因为这种规范化有利于互操作。如果保持C API表面的简洁性，并且避免线程，那么通过Java JNI、CPython或是MRI Ruby等可以将其作为本地扩展。一旦完成了这个过程，接下来通过消息公开C API就变得很简单了，因为根据定义，两个系统之间的消息负载不可能比C库还要复杂。当然了，如果使用消息，这意味着要么使用C编写消息代码，要么将C库公开到更高层的语言上，并在那里实现消息。消息好的一面是能够将高层语言代码与C代码进行隔离，这么做会比Java代码要薄一些。我依然不会直接将使用C APIs编写的代码链接到我的应用中。如果C代码挂掉了，那么消息系统就会接收请求，直到运行着C代码的另一个节点能够进行处理为止。另一方面，如果真的因为性能原因而使用C，那么消息至少会引入一个网络传输，更不必说系统中的另一个组件了，这么做就抵消了使用C编码所带来的优势了。在这种情况下，我们可以编写稳定、行为良好的JNI或是本地扩展，但还是需要保持表面的小巧，并且理解前置与后置条件才行。没有线程。不要在C与Java之间传递指向复杂对象的指针。请确保你自己清楚谁来负责清理内存，什么时候

清理。

总而言之，忘掉C++吧。

读者 Bernd Kolb 说到：

或许你想要看看mbeddr (mbeddr.com)。

mbeddr旨在更好地支持嵌入式软件开发（但并不仅仅限于此），针对基于C语言与IDE的可扩展版本的小型与大型系统。现有扩展包括前置和后置的接口、组件、状态机与物理单元，以及对需求追踪和产品线变化的支持。基于这些抽象，mbeddr还支持基于模型检测与SMT处理的形式验证。

通过这种方式，在大型项目与团队中，C也可以做到“可伸缩”。此外，我们还可以引入现代的编程规则。通过扩展mbeddr，你甚至可以为消息等添加基础信息。

原文链接：<http://www.infoq.com/cn/news/2013/01/C-Language>

相关内容

- [海量视频广告分发的挑战](#)
- [专访 Rust——由 Mozilla 开发的系统编程语言](#)
- [InfoQ 中文站 2011-2012 读者深度调查报告发布并提供下载](#)
- [红帽企业版 Linux 新添 SQL Server 驱动](#)

热点新闻

12306 订票助手插件拖垮 GitHub 事件原因始末

作者 彭超

事件起因

春节临近，12306 订票难的问题再一次被引向风口浪尖。而这一次，各家浏览器厂商不失时机的推出了“春节专版”。这些林林总总浏览器的共同特点，是集成了一位网友 [iFish\(木鱼\)](#) 的“订票助手”插件。

不凑巧的是，这个插件的早期版本使用 [GitHub](#) 的 [Raw File](#) 服务作为 CDN，并对返回 403 错误代码的请求使用非常暴力的 5 秒重试。于是，在 1 月 15 日的时候，第一个订票小高峰到来的时候，GitHub 被间接的 DDoS。

GitHub 的运维工程师 [Jesse Newland](#) 在发现服务器负载异常之后，不得不禁用了这个代码所在 Repo 的 Raw 服务，并在 Repo 里报告了一个 issue —— 他发现 12306 引用了这个 Repo 里的一个资源，由于访问量巨大，这个资源对 GitHub 的服务产生了负面影响，希望有人可以联系到 12306 的工程师去除这个引用。

身在大洋彼岸的 GitHub 工程师在解决了 GitHub 的服务问题之余，显然不太清楚中国的两点情况：

1. 春运是什么，12306 是什么
2. 12306 的工程师是不可能被联系到的

大家疑问

大家的质疑，存在于两点：

1. 为什么一个浏览器插件需要从 GitHub 引用资源？
2. GitHub 的负载能力这么弱吗？

疑难解释

第一个问题，还是让插件作者木鱼自己来解释：

引入自动更新。

由于12306订票助手是个很特殊的东西，依赖于铁道部的网站而存，并且其运行极度依赖网站本身的功能以及页面结构，所以随着铁道部的改进，很容易失效（虽然他们的前台样子从开始到现在，一年多了几乎就没变过……他喵的为什么我又要用年做单位说时间，真伤心）。因此为了保证功能的正常，订票助手在很早的版本开始就引入了自动更新机制（1.4开始）。最开始的更新都是放在自己网站上的，并且区分了Firefox和Chrome。

最初的助手是以UserScript的模式出现的，调试在Firefox下调试。在Firefox下时，Scriptish提供了支持跨域的GMxmlHttpRequest功能，可以直接用ajax访问我的网站。但是在Chrome下，则没有这样的便利，不支持跨域ajax，所以用的是引入script脚本的方式检测更新。在后来Firefox和Chrome分支完全合并后（最开始针对不同的浏览器分离的，后来发现同步实在太麻烦了），舍弃了一些特异的功能（如GMxmlHttpRequest），为一些功能做了适配（如桌面通知），更新也就用下来了。

但是后来不知道什么时候开始，这个更新机制突然失效了。为啥呢，这要从另一件事开始说起。

那就是12306的HTTPS。

作为一个日点击14亿的网站，网宿科技的CDN还是很给力的，根据我收集到的资料，其加速节点上百个。但是，作为一个订票的网站又要CDN的，为什么会用HTTPS协议还是一个自签发的根证书，实在太让人费解。任何一个了解网络知识的人都知道，HTTPS协议下服务器的负载能力要比HTTP的低很多，何况订票又不是什么机密的数据。总会有人跳脚出来说订票啊多机密，我总是很反对，哪门机密了，车次还是余票数据？

这个HTTPS带来了很大的麻烦。

不知道哪个版本Chrome引入的安全机制，对于一个HTTPS网站，其所有引用的资源（Script和StyleSheet之类的），也必须位于HTTPS的服务器上，否则拒绝执行。而我并没有HTTPS服务器，因此，Chrome下自动更新华丽地挂了。然后是Firefox。Firefox下播放不了音乐，我一直以为是Firefox不支持，后来才发现是Firefox的安全机制在作怪：HTTPS的网页拒绝播放

来自于HTTP的多媒体文件。这俩奇葩让我伤透了脑筋。然后无意中瞥见GitHub竟然是HTTPS的，So.....转移过去，变成了顺理成章的事情，我求爹爹告奶奶没求来一台HTTPS的服务器，虽说有免费的SSL证书什么的但是我去申请的时候，连那提供商的网站自己都证书错误了。

于是事情都解决。

而第二个问题，GitHub 的负载能力为什么这么弱，原因在于 GitHub 根本不适合作为 CDN 服务。著名博客[比特客栈的文艺复兴](#)，对此做了详细解释：

1. 它并非静态文件服务器，换句话说，所有请求访问都要先经过一堆服务器代码处理，降低了它的响应速度。
2. 它返回的MIME与文件无关（永远是text/plain），某些浏览器，比如说IE，不会执行MIME类型错误的javascript文件。
3. 它返回的Cache-Control Header不允许浏览器缓存文件，等于失去了CDN最基本的功能。恰恰因为12306订票助手不运行于IE，也不希望更新文件被缓存，Raw file的后两个劣势才没有显现出来。但剩下的那个劣势，却让Github的响应速度大打折扣，不得不暂时封锁Raw file访问。

针对这个问题，原文作者直中要害的提供了两个层面的解决方案：

- 使用GitHub作为CDN的正确之道：

那么，Github作为CDN的正道是什么？Github Pages。通过加入gh-pages branch，你可以修改和发布自己repo的文件。值得提醒，Pages虽然免费，并非资源无限，详见官方[Disk Quota](#)的描述——“虽然我们没设上限，但请各位合理使用。”

Github轻描淡写的说合理使用，而不是严明规章到MB、GB，其实是一种潜意识的相互信任。这是一种在贫富悬殊供求关系紧张的中国日益缺少的东西，12306订票助手的存在就是一种印证：乘客不相信12306，12306不相信乘客，最后逼出一个12306订票助手，每天乃至每半天更新一次来满足中国人订票回家的需求。

- HTTPS下如何引用HTTP资源：

请问Google Reader，是怎么在HTTPS域下播放优酷与土豆等非HTTPS的视频？我们在去年9月发现了同样的问题，Google自己是这样解决的——

Chrome 21之后，在SSL加密页面embed非SSL的Flash会怎样呢？会被默默的屏蔽掉，只留下一句console报告。那Google Reader是怎么绕过这个问题看优酷与土豆视频的？他们iframe了一个非SSL页面，再在里面引用flash（引用页连域名都是不同的）

同理也适用于Javascript，这也是12306订票助手当前的解决办法（Firefox除外）。不再需要SSL下的CDN了。

不是结束

也许一切自有冥冥天意。[Jesse Newland](#)，这位在“订票助手”Repo中发出警告的GitHub员工，早在2012年12月份，就已受InfoQ之邀，确定参加[2013QCon大会（北京站，4月25-27）](#)。除了分享GitHub的架构演进之外，Jesse还会分享他负责的项目——GitHub ChatOps运维机器人。不过看来这次大会的演讲，他将不得不加入关于12306插件的话题。

订票助手的作者木鱼，不堪忍受各界观光团纷纷造访他本开源在GitHub上的订票助手[代码仓库](#)，最终删除了项目。但他表示仍将继续精简/改进这款订票插件。

因为这个插件，铁道部甚至投诉至工信部，要求其责令各家浏览器提供商停止提供附带抢票功能的浏览器的下载。不过就本文发表前，各家浏览器厂商均表示尚未接到相关通知。

这一切还都不是结束。

原文链接：

<http://www.infoq.com/cn/news/2013/01/12306-plugin-ddos-github>

相关内容

- [HTML 5 开发平台](#)
- [周爱民谈 javascript 的发展](#)
- [淘宝的 HTML5 实践](#)

特别专题 | Topic

本期的主题是 Go 语言 ,关于这门语言 ,已经不需要太多的形容和赞美 ,一句 “互联网时代的 C 语言” 已经足够。从 2009 年 11 月 10 日发布以来 ,Go 语言已经有了长足的发展 ,特别是 2012 年 3 月 28 日 ,第一个正式版本 Go1 的发布 ,更标志着这门语言在发展的过程中迈出了非常坚实的一步。

国内的程序员们也对 Go 语言的发展做出了很多贡献。在实践方面 ,七牛云存储团队第一个使用 Go 语言开发了商业化的云存储平台 ,盛大研究院也开始在一些项目中使用它 ;在书籍方面 ,不仅有已经出版的纸质书《 Go 语言编程》、《 Go 语言 ,云动力》 ,还有在 GitHub 上开源的《 Go Web 编程》 ,这对于让更多程序员学习和使用这门语言起到了很大的推动作用 ;在开发工具方面 ,Visualfc 开发的 LiteIDE 是世界上最早出现的 Go 语言 IDE 之一 ,为 Go 语言提供了很好的可视化开发工具 ,而且他还在不断努力 ,向其中添加各种极具工程师风格的特性。

InfoQ 中文站也期望可以为 Go 语言在国内的发展做些努力 ,所以就有了这样的一期《架构师》 ,我们邀请了七牛云存储的 CEO 同时也是《 Go 语言编程》的作者许式伟、《 Go 语言 ,云动力》的作者樊虹剑、《 Go Web 编程》的作者谢孟军以及 LiteIDE 的开发者 visualfc ,共同把这期 Go 语言的大餐献给各位读者。

特别专题

Go 语言开发工具 LiteIDE

作者 [visualfc](#)

2010 年 11 月对外公布，在 2011 年 3 月 16 日发布第一个 release，第一个正式版本 Go1 于 2012 年 3 月 28 日推出。在 Go 语言的正式版本推出后，Eclipse、IntelliJ IDEA、vim、emacs、gedit、SublimeText2、Textmate、Textpad、SciTE、Notepad++ 等 IDE 和编辑器开始纷纷有了各自的 Go 语言插件。

LiteIDE 是一款专为 Go 语言开发而设计的跨平台轻量级集成开发环境（IDE），基于 Qt 开发，支持 Windows、Linux 和 Mac OS X 平台。LiteIDE 的第一个版本发布于 2011 年 1 月初，是最早的面向 Go 语言的 IDE 之一。到 2013 年 1 月为止，LiteIDE 已经发布到版本 X16。

LiteIDE 主要特点

- 支持主流操作系统
 - Windows
 - Linux
 - MacOS X
- Go 编译环境管理和切换
 - 管理和切换多个 Go 编译环境
 - 支持 Go 语言交叉编译
- 与 Go 标准一致的项目管理方式
 - 基于 GOPATH 的包浏览器
 - 基于 GOPATH 的编译系统
 - 基于 GOPATH 的 Api 文档检索
- Go 语言的编辑支持
 - 类浏览器和大纲显示
 - Gocode(代码自动完成工具)的完美支持
 - Go 语言文档查看和 Api 快速检索
 - 代码表达式信息显示 F1

- 源代码定义跳转支持 F2
- Gdb 断点和调试支持
- gofmt 自动格式化支持
- 其他特征
 - 支持多国语言界面显示
 - 完全插件体系结构
 - 支持编辑器配色方案
 - 基于 Kate 的语法显示支持
 - 基于全文的单词自动完成
 - 支持键盘快捷键绑定方案
 - Markdown 文档编辑支持
 - ◆ 实时预览和同步显示
 - ◆ 自定义 CSS 显示
 - ◆ 可导出 HTML 和 PDF 文档
 - ◆ 批量转换/合并为 HTML/PDF 文档

下面，LiteIDE 的作者 visualfc 将介绍 LiteIDE 的安装配置，并展示一个使用 LiteIDE 开发 Go 语言应用的示例。

安装配置

首先需要安装好 Go 语言，Go 语言的项目地址是 <http://code.google.com/p/go>，可以下载二进制文件或通过源码自行编译，按 Go 文档配置好 Go 开发环境。根据需要可以选择安装 Gocode，以支持 Go 语言输入自动完成，`go get -u github.com/nsf/gocode`。

LiteIDE 的下载地址为

<http://code.google.com/p/golangide/downloads/list> ,根据操作系统下载 LiteIDE 对应的压缩文件直接解压即可使用。

运行 LiteIDE,根据当前系统切换和配置 LiteIDE 当前使用的环境变量。以 Windows 操作系统，64 位 Go 语言为例，工具栏的环境配置中选择 win64，点 编辑环境，进入 LiteIDE 编辑 win64.env 文件

```

GOROOT=c:\go
GOBIN=
GOARCH=amd64
GOOS=windows
CGO_ENABLED=1

PATH=%GOBIN%;%GOROOT%\bin;%PATH%
...
    
```

将其中的 GOROOT=c:\go 修改为当前 Go 安装路径，存盘即可，如果有 MinGW64 ,可以将 c:\MinGW64\bin 加入 PATH 中以便 go 调用 gcc 支持 CGO 编译。

如果当前系统为 Linux 操作系统，64 位 Go 语言，则在工具栏的环境配置中选择 linux64 , 点编辑环境，进入 LiteIDE 编辑 linux64.env 文件

```

GOROOT=$HOME/go
GOBIN=
GOARCH=amd64
GOOS=linux
CGO_ENABLED=1

PATH=$GOBIN:$GOROOT/bin:$PATH
...
    
```

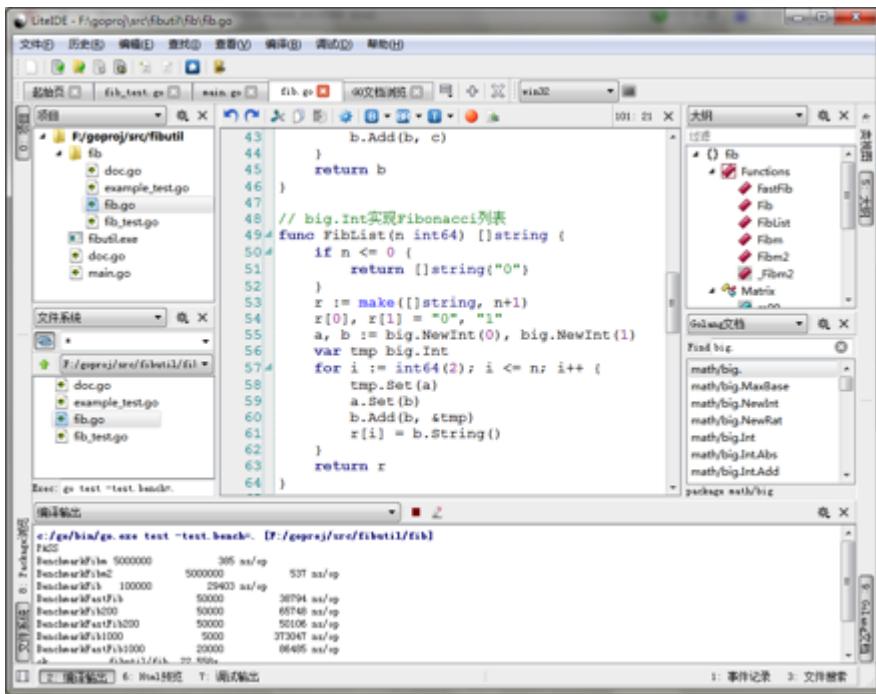
将其中的 GOROOT=\$HOME/go 修改为当前 Go 安装路径，存盘即可。

配置 GOPATH 设置，Go 语言的工具链使用 GOPATH 设置，是 Go 语言开发的项目路径列表，在命令行中输入 go help gopath 快速查看 GOPATH 文档(在 LiteIDE 中也通过可以 Ctrl+, 调出命令输入)。在 LiteIDE 中可以方便的查看和设置 GOPATH。通过菜单 - 查看 - GOPATH 设置，可以查看系统中已存在的 GOPATH 列表，同时可根据需要添加项目目录到自定义 GOPATH 列表中。

使用 LiteIDE 开发一个简单的 Go 语言应用示例

项目简介

我们的目标是实现一个计算斐那契数列(Fibonacci)的程序，主程序为 fibutil ，这是一个简单的命令行程序，算法库为 fibutil/fib ,以包(Package)的形式提供，并实现 fib 包的相关测试文件和函数示例文档，最后实现交叉编译。



建立项目结构

我们先设置 GOPATH，菜单 - 查看 - 设置 GOPATH，将 `f:\goproj` 添加到自定义 GOPATH 中。首先使用向导建立 fibutil 项目，模板选择 Go1 Command Project，GOPATH 目录选择 `f:\goproj`，项目名称添写 fibutil 确定后并加载 fibutil 项目，这将自动生成并加载一个简单的 hello world 项目。然后使用向导建立 fibutil/fib 项目 模板使用 Go Package Project 项目名称添写 fibutil/fib，确定但不需要加载 fib 项目，直接在 fibutil 项目工作即可，如上图中项目窗口所示。与 Go 语言标准一致，在 LiteIDE 中项目就是目录，如果不使用向导而直接在 GOPATH/src 下建立上述目录和文件，效果是完全一样的。

Fibonacci 数列规律

数列列表：

$$\text{数列规律} : F_{n-2} = F_n - F_{n-1}, F_{-n} = (-1)^{n+1} F_n.$$

编写 fib 函数

项目窗口中双击fib.go并编辑 先编写使用int64计算Fib的函数Fibm和Fibm2，分别以非递归方式和递归方式来实现，代码如下：

```
// 非递归方式实现Fibonacci算法
func Fibm(n int64) int64 {
    if n == 0 {
        return 0
    }
    var i, a, b int64
    b = 1
    if n < 0 {
        n = -n
        if n%2 == 0 {
            b = -1
        }
    }
    for i = 2; i <= n; i++ {
        a, b = b, a+b
    }
    return b
}

// 递归方式实现Fibonacci算法
func Fibm2(n int64) int64 {
    if n < 0 {
        if n%2 == 0 {
            return _Fibm2(-n, 0, -1)
        } else {
            return _Fibm2(-n, 0, 1)
        }
    }
    return _Fibm2(n, 0, 1)
}

func _Fibm2(n, a1, a2 int64) int64 {
    if n == 0 {
        return a1
    }
    return _Fibm2(n-1, a2, a1+a2)
}
```

注意一点，Fibm2 递归方式实现中是通过辅助函数_Fibm2 来确保正确的尾调用。函数上方的注释会自动添加到文档中。

编写测试

接下来编写测试文件，Go 语言的测试文件约定命名为 xxx_test.go，测试文件的 Package 有以下两种命名方式：

```
package fib          //称为Test方式，可以直接使用fib包内函数
package fib_test     //称为XTest方式，因为包名与fib不同，必须使用import "fibutil/fib"
```

测试函数约定命名为 TestXXX , 性能测试函数约定命名为 BenchmarkXXX。在项目窗口 fib 目录上右键新建文件 fib_test.go 并编辑 , 输入以下代码 :

```

package fib

import (
    "testing"
)

func TestFibm(t *testing.T) {
    ar := []int64{0, 1, 1, 2, 3, 5, 8, 13, 21}
    for i := 0; i < len(ar); i++ {
        if ar[i] != Fibm(int64(i)) {
            t.Fatalf("%d, %d != %d", i, ar[i], Fibm(int64(i)))
        }
    }
    ar1 := []int64{0, 1, -1, 2, -3, 5, -8, 13, -21}
    for i := 0; i < len(ar1); i++ {
        if ar1[i] != Fibm(int64(-i)) {
            t.Fatalf("%d, %d != %d", -i, ar1[i], Fibm(int64(-i)))
        }
    }
}

func TestFibm2(t *testing.T) {
    ...
}

```

LiteIDE 在保存时会自动格式化源码 , 现在点击工具栏上的测试按钮 (Ctrl+T) 进行测试 (等同于 go test) , 如果测试通过则显示

```

PASS
ok      fibutil/fib 0.036s

```

如果测试不能通过 , 则显示相应的错误信息 , 比如我们将测试函数 TestFibm 中数字 21 修改为 22 , 重新测试会显示错误

```

--- FAIL: TestFibm (0.00 seconds)
    fib_test.go:11: 8, 22 != 21
FAIL
exit status 1
FAIL      fibutil/fib 0.035s

```

双击错误行 fib_test.go:11: 8, 22 != 21 则跳转到编辑器对应行 , 修改后重新测试 , 直到测试通过。

性能测试

我们可以通过性能测试函数来对 Fib 算法的递归和非递归实现性能进行直观比较，在 fib_test.go 文件中输入以下代码并保存。

```
func BenchmarkFibm(b *testing.B) { for i := 0; i < b.N; i++ { Fibm(90) } }
func BenchmarkFibm2(b *testing.B) {
    ...
}
```

使用快捷键 Ctrl+, 调出文件系统的命令窗口输入 go test -bench=. 回车执行，结果如下：

```
PASS
BenchmarkFibm      5000000          358 ns/op
BenchmarkFibm2     5000000          513 ns/op
ok      fibutil/fib 5.286s
```

从性能测试结果中，我们可以看到非递归方式效率优于递归实现方式。

支持大数操作

我们注意到，Fibm 函数使用 int64 运算，意味着 Fibm 函数最大只能支持到 92，可以在 LiteIDE 中查找一下 Go 语言标准库中是否提供了大数计算实现，在 LiteIDE 侧边栏的 Golang 文档窗口中输入 big 进行检索，显示如下列表：

```
math/big
math/big.NewInt
math/big.Int
math/big.Abs
...

```

从名称上可以判断 math/big.Int 支持大数操作，双击 math/big.Int 在 LiteIDE 中将直接打开 math/big 文档并同时定位到 big.Int 函数文档上。

现在使用 big.Int 编写支持大数操作的 Fib 函数和使用矩阵求解的 FastFib 函数，首先加入 import math/big, 函数代码如下：

```
// big.Int实现Fibonacci算法
func Fib(n int64) *big.Int {
    if n == 0 {
        return big.NewInt(0)
    }
    a, b, c := big.NewInt(0), big.NewInt(1), big.NewInt(0)
    if n < 0 {
        n = -n
        if n%2 == 0 {
            b.SetInt64(-1)
        }
    }
    for i := int64(2); i <= n; i++ {
        c.Set(a)
        a.Set(b)
        b.Add(b, c)
    }
    return b
}

// 使用矩阵求解Fibonacci算法
func FastFib(n int64) *big.Int {
    ...
}

// big.Int实现Fibonacci列表
func FibList(n1,n2 int64) []string {
    ...
}
```

同时在 fib_test.go 文件中编写 Fib 函数相应的测试代码和性能测试代码，性能测试结果如下：

```
PASS
BenchmarkFibm      5000000          359 ns/op
BenchmarkFibm2     5000000          513 ns/op
BenchmarkFib       100000          28878 ns/op
BenchmarkFastFib    50000          36354 ns/op
BenchmarkFib200    50000          65398 ns/op
BenchmarkFastFib200 50000          47926 ns/op
BenchmarkFib1000    5000          350344 ns/op
BenchmarkFastFib1000 20000         78159 ns/op
ok      fibutil/fib 21.607s
```

可以看到 big.Int 要比 int64 慢许多，支持大数操作的 Fib 的效率为 $O(N)$, FastFib 效率为 $O(\log(N))$, 在 N 较大时对比非常明显。

编写 Fib 函数代码示例

在项目窗口 fib 目录上右键菜单可以查看 fib 包的 GODOC 文档，我们也可以为文档加入函数示例，方法如下，项目窗口 fib 目录右键新建文件 example_test.go，

输入以下代码：

```

package fib

import (
    "fmt"
)

func ExampleFibList() {
    fmt.Println(FibList(-10, 10))
    // Output: [-55 34 -21 13 -8 5 -3 2 -1 1 0 1 1 2 3 5 8 13 21 34 55]
}

```

函数 FibList 的示例名称约定为 ExampleFibList，如果有标准输出，则使用// Output:来标识输入，测试时会测试 ExampleFibList 的输出是否正确。先运行测试看是否通过测试，右键查看 GODOC 文档时就会看到 Fib 函数的这个代码示例了。

编写 fibutil 主程序

接下来将编写 fibutil 命令行程序，在 fibutil 项目窗口中双击 main.go 进入编辑。在源码中输入 import fibutil/fib 加入包引用，然后按编译菜单 - Get 按钮（对应于按 Ctrl+R 输入 go get . 并回车），这样会自动安装 fibutil 需要的包到 pkg 目录中，以支持 Gocode 自动输入完成，在编辑过程中输入 fib.则会自动显示包函数提示。代码如下：

```

package main

import (
    "fibutil/fib"
    "fmt"
    "os"
    "strconv"
)

func main() {
    switch len(os.Args) {
    case 2:
        n, err := strconv.ParseInt(os.Args[1], 10, 64)
        if err == nil {
            fmt.Println(fib.FastFib(n))
            return
        }
    case 3:
        n1, e1 := strconv.ParseInt(os.Args[1], 10, 64)
        n2, e2 := strconv.ParseInt(os.Args[2], 10, 64)
        if e1 == nil && e2 == nil {
            fmt.Println(fib.FibList(n1, n2))
            return
        }
    }
    fmt.Fprintf(os.Stderr, "%s, fibonacci number util\n\tfibutil n\tfibonacci number\n\tfibutil n1 n2\tfibonacci number list\n", os.Args[0])
}

```

在编辑区的 fib.Fib 上按 F1 会显示 fib.Fib 函数信息，如果按 F2 则会直接跳转到 fib.Fib 源码定义处，这样可以很方便的浏览 Go 源代码。现在点击编译运行（Ctrl+R）开始编译并执行 fibutil 程序，可以在工具栏的编译配置的 TARGETARGS 中输入 -10 10 配置系统与 goproj/src/fibutil 目录关联在一起，

再次编译并执行时，fibutil 程序会带参数运行（也可以 Ctrl+Shift+F5 通过命令行 fibutil -10 10 执行），输出如下：

```
[-55 34 -21 13 -8 5 -3 2 -1 1 0 1 1 2 3 5 8 13 21 34 55]
```

调试

Go 语言编译器 gc 生成 DWARF 格式调试信息，gdb7.1 以上可以很好的调试 Go 语言，LiteIDE 集成了图形化调试功能，提供断点设置、显示变量值、函数调用栈、添加变量监视等功能，同时支持调试控制台直接输入 gdb 调试命令。在对 Go 源程序调试时，如果需要禁用优化和内联，可以使用工具栏编译配置 - 编译自定义 - BUILDARGS 中输入 -gcflags "-N -l" 重新编译后调试即可，在菜单 - 选项 - LiteDebug 中可以选择调试前重编译，这样每次调试时会自动重新编译。

交叉编译

Go 语言支持多平台交叉编译，我们准备将 fibutil 交叉编译为 Linux-amd64 目标。首先要编译 Go 源码以创建目标平台所需要的包和工具，进入 cmd 命令行执行：

```
> set GOOS=linux
> set GOARCH=amd64
> set CGO_ENABLED=0
> cd c:\go\src
> all.bat
```

注意的是 Go1.0x 版本同一份源码仅支持一个本地平台和一个交叉编译平台，go 最新源码 hg-tip 版本则没有这个限制。

编译好目标平台所需要的包和工具后，在 LiteIDE 中切换环境配置为对应的交叉编译环境，即 cross-linux64，编辑配置文件，修改确认 GOROOT 为交叉编译平台的 GOROOT 存盘即可。打开 fibutil 项目内的 main.go 文件，现在使用编译按钮则会编译出 Linux-amd64 平台的目标 fibutil，将生成的 fibutil 二进制文件复制到 Linux-amd64 上即可正确运行，在 LiteIDE 中通过切换不同的配置文件，可随时编译为本地可执行文件和交叉编译目标平台可执行文件。

编写 README

最后，我们可以为 fibutil 写一份 README 简介，选择 Markdown 书写格式，LiteIDE 支持 Markdown 编辑和实时预览，也可以转换输出为 Html/PDF 文档。在 fibutil 项目右键新建文件 README.md 并编辑，写上项目的简介和使用说明，在 Html 预览窗口中可以实时预览，通过切换不同的 CSS 来显示不同的预览效果。本文就是通过 LiteIDE 的 Makdown 编辑器编辑完成。

本文代码

fibutil 源代码可以从 <https://github.com/visualfc/fibutil> 下载，可以使用 go get 安装: go get -v github.com/visualfc/fibutil。github 网站上的代码与本文示例代码区别在于于引用位置不同，即在 fibutil.go 文件中使用 import github.com/visualfc/fibutil/fib 替代 import fibutil/fib

关于作者

visualfc，非计算机专业毕业的狂热程序员，擅长 C++、Lua、Go 等语言，热衷于开源软件，业余时间开发了 WTL 可视化开发插件 VisualFC(<http://code.google.com/p/visualfc>)、基于 Qt 的跨平台轻量级 Go 语言集成开发环境 LiteIDE(<http://code.google/p/golangide>)，他的邮件是 visualfc@gmail.com。

特别专题

界面漫谈

作者 樊虹剑

一件作品的诞生，通常是一个设计师独立完成的。因为这样，一件建筑也好，画作或者音乐舞蹈也好，才能真实反映出其个性。而正是这种不同于其他同类的独特一面，正是这种发自创造者的灵光一现、但又不会背离创作目的和原始架构的新颖实用之处，才使得创新尤为难得。

Go 语言的诞生，是三个有很强个性的设计师共同完成的。Go 语言的定位，就象三维坐标系中的一个点，在强类型、动态和并发这三个特性维度上，分别代表了 Ken、Robert 和 Rob 三人的创造思维的投影。

当然，这样描述不仅是为了表达 Go 语言有这三个特性，也是为了清晰地说明，这三个特性是正交的，也就是它们是彼此独立的，因此可以同时使用而不会彼此制约。当然，这样描述只是形象地比喻，并不是说这三个设计师彼此独立不必彼此制约，就可以得到同一个独立完整的 Go 语言架构。恰恰相反，只有三人共同认可的特性，才会出现在 Go 语言规范，才会发布在 Go 语言的实现上。有趣的是，这种三驾马车的设计组合，也是 Lua 语言所采用的。它使 Lua 成功避免了过度设计的陷阱，能够在保持自身苗条的同时也不会洁身自好，而是能不断的自我更新，提高性能。

如果说 Lua 语言的一个特性是其唯一但又灵活高效的 table 复合类型，那 Go 语言的一个特性我认为就是其唯一但又灵活高效的 interface 动态类型。这种类型使 Go 语言在保持强静态类型的安全和高效的同时，也能灵活安全地在不同相容类型之间转换。在进入正题之前，我们先插播一段轻松的话题。关于 interface 的中文翻译，正统的教育教导我们说是接口。例如，Java 和 C++ 中的对象可以理解为非常自闭的个体或者具有同样遗传基因的同类个体的族谱。此时，接口就能恰如其分地表示：要得到我的遗传基因，必须使用此接口。例如，只有声称和驴马都接口了的那种类，才能自称骡类。接口要在定义类时明确声明。

在 Go 语言里，“接吻需声明（注意口写小了些以便好笑增强记忆）”。所以 Go 的接口和正统的类完全不是一类。为避免误解，也为了和港澳台所代表的国

际译法接轨，我倾向于把 interface 翻译为“界面”。当然，这也符合这个英文的词源：inter 是中界 face 是面。

好了。够了。该讲 Go 了。

要理解动态类型，需要从静态开始。Go 和 C 族语言一样，是强静态类型的编译语言。每一个变量必须预先声明其类型，也只有相同类型的变量才能赋值和参与运算。例如：

```
i := 0
j := 0i
```

分别声明变量 i 和 j 是整型 int 与复数型 complex128。尽管它们的值都是零，尽管我们确信这两个零可以相加并应该能得到正确的零，Go 的编译器却一定会强烈反对。它认为 i 和 j 不是一类不可以运算。这就是强静态类型编译。它把程序员认为可以做的事情一丝不苟的进行强制类型检查，凡是不符合它的规定的一律不予编译，而是举报错误供作者自我检讨。

如果作者要和编译器讨价还价，就要象律师一样研读 Go 的语言规范，才能明白什么是可以通融的、什么是绝对禁止的。例如，Go 语言规范里规定数值类型之间可以有限度的相互转换，例如，整数和浮点数之间，但不包括到复数类型。如果 j := 0.0 声明 j 是浮点数类型，则 float64(i) + j 就可以在强制把整数型的 i 转换为浮点数类型后，再做相同类型变量之间的加法运算。

学过面向对象编程的读者可能会想：嘿，Go 要是能向 XXX 语言一样支持操作符重载或者继承，就不会再有这种加法运算类型不相容的问题了。

真是没问题了吗？还是说问题被抽象了，遮盖了或者说学者除了要学习不同类型之外还有多学一层不同层次的知识了？是简化了还是更复杂更难琢磨了？相信读者会明辨的。

Go 的面向对象不支持重载也只有有限的继承。目的很明确，Go 是要简化类型系统、尤其是已经被过度复杂化了的面向对象的类的类型系统。

这和界面所代表的动态类型系统有关系吗？或者我们问自己，面向对象复杂的类和类型系统所要解决的问题如何用 Go 语言来表达？静态的类和类型，能动态的 interface 吗？

例如，要实现两个不同类型的形状的面积的加运算，在面向对象的语言里，就需要定义一个基类，让这个鸡肋(谐音)有个方法可以相加，再让每个形状去继承，才可以让编译器知道这些类的形状的类型所继承的那个不是任何具体形状的那类形状声明了没有任何具体操作的取得面积的运算，从而可以通融，从而可以从具体类型自己必须已经重新定义的具体的取得自身面积的方法得到具体的数值，才可以把两个具体而且同类的数值相加从而得到面积之和。

如果学者认为是笔者故意把一个简单的道理说得云山雾罩，那学者同志就真的领会了面向对象的精神。让我们拨云见日吧，看看 Go 的界面是怎样解释这个操作的吧。“接吻需声明”或者说“界面勿需声明”。例如只要两个形状都有取面积的方法，就可以把它们的面积相加，就这么简单明确，完全不需组织它们到同类的抽象形状，也无法在 Go 里做这种勾当。具体的例子：

```
package main
import "fmt"

type square struct{ r int }
type circle struct{ r int }

func (s square) area() int { return s.r * s.r }
func (c circle) area() int { return c.r * 3 }

func main() {
    s := square{1}
    c := circle{1}
    fmt.Println(s, c, s.area()+c.area())
}
```

这里所谓的界面 就是方形 square 和圆形 circle 都有 area()int 这样的方法。注意，我们要下面要用到界面类型了：

```

package main

import "fmt"

type square struct{ r int }
type circle struct{ r int }

func (s square) area() int { return s.r * s.r }
func (c circle) area() int { return c.r * 3 }

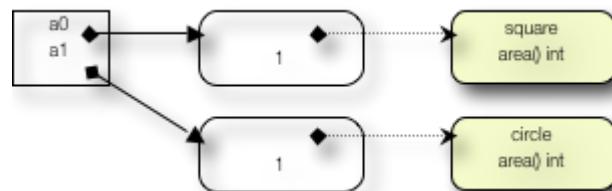
func main() {
    s := square{1}
    c := circle{1}
    a := [2]interface{}{s, c}
    fmt.Println(s, c, a)

    sum := 0
    for _, t := range a {
        switch v := t.(type) {
        case square:
            sum += v.area()
        case circle:
            sum += v.area()
        }
    }
    fmt.Println(sum)
}
    
```

变量 a 是 interface{} 空界面类型的数组变量，类似 C 语言的 void *，可以把任何类型的值放入其单元。此处我们分别放入单位方形和单位圆形变量 s 和 c 的值。

range 是 Go 的遍历语句，此处的变量 t 被依次赋值为数组 a 的单元值，它们还都是空界面类型，所以我们只需用 switch 测试并转换成具体类型的变量 v，就可以使用这个具体类型所定义的 area 方法，得到相应的面积，并进行求和运算了。

这里提到空界面类型类似 C 语言的 void * 空指针类型。实际上，为了能动态地检查类型，就必须让这个指针指向一个结构而不是直接指向对应的具体值。这个结构要同时包括值的类型说明和值本身。例如：



图中的两个实线箭头是从空界面数组类型 a 的两个单元指向它们赋值的两个具体类型的值，分别是 square 类型的变量 s 的值 1，以及 circle 类型的变量 c 的值，刚好也是 1。

由于 s 和 c 赋值给界面类型的变量 a[0] 和 a[1] 在内存中，它们不仅仅就只有值。上文说过，界面类型的值实际上是个结构，包括具体值和方法表指针。图中虚线箭头所所示的，就是方法表指针。正是通过这个指针，Go 程序运行时才可以顺藤摸瓜地从一个界面变量得到具体变量的类型和它们实现的方法，从而能够在动态类型检查安全后，才执行对应的方法操作。如果安检不过关，就会 panic，也就是出现运行态异常，就是类似数组越界或者除 0 所产生的那种异常。Go 的程序可以使用 recover 捕捉并处理这些异常，这里就不再详述了。

熟悉面向对象语言内部实现的学者肯定能嗅到虚拟函数表的味道。事实上，正是由于 Go 是强类型的编译语言，这些类型的方法函数或者可以在编译时就静态的确定，从而不需间接调用；或者就是通过界面变量这种编译时静态分配一个间接的带类型的方法指针表，从而在程序运行时再动态的类型检查，然后“多态的”调用方法函数。这里所谓的多态，并不是 Go 语言的概念，但这种面向对象的概念，实际上 Go 语言可以通过界面类型有限地支持的。

在 Go 语言中有一个非常重要的界面类型，也是 Go 语言内置的唯一界面类型，error 类型。而 Go 语言库函数以及使用惯例，是返回这个 error 类型的 nil 值表示没有错误，否则就返回一个具体的值表示特定的错误。例如我们定义一个 Err 类型符合 error 界面，也就是要有一个返回 string 的叫 Error 的方法：

```
type Err struct {}
func (_ *Err) Error() string {
    return "To err is human"
}
```

当函数报错时，我们就返回这个 Err 类型的值，而没有错误时，就返回 nil。注意 Err 类型的值是 error 界面类型所指向的具体值，而 nil 代表这个 error 不指向任何具体值。所以：

```

func NoErr(ok bool) error {
    if !ok {
        return &Err{}
    }
    return nil
}

func main() {
    fmt.Println(NoErr(true))
    fmt.Println(NoErr(false))

    // Output:
    //           // To err is human
}
    
```

但如果我们不小心写了如下的错误例子：

```

func ToErr(ok bool) error {
    var e *Err = nil
    if ok {
        e = &Err{}
    }
    return e
}
    
```

如果我们错误地返回一个 Err 类型但值为 nil 的具体值，而不是直接返回 nil，就会发现依靠返回的 error 是否是 nil 来判断是否出错不再有效：

```

func main() {
    fmt.Println(ToErr(true) == nil) //false
    fmt.Println(ToErr(false) == nil) //false
}
    
```

这是因为 nil 也是 Err 类型的有效值，而 Err 类型实现了 error 界面的方法 Error()，所以这个 nil 值也一样会调用 Error() 方法返回 “To err is human” 这个字符串，而不是 nil。

本文只是提纲挈领地展示了一点点 Go 语言界面类型的特色，并添油加醋了一大堆闲言碎语。相信学者朋友们的智商要比作者敝人的高些，能自己去莞存蓄，也能举一反三地明白 Go 语言如何简单地用一个界面的概念实现了面向对象和动态类型编程。因为本文只是篇漫笔，并非面面俱到地全面讲述，希望读者朋友们能对本人的不周甚至荒唐走板之处一笑了之。谢谢。

热别专题

Go，基于连接与组合的语言（上）

作者 许式伟

到目前为止，我做过不下于 10 次关于 Go 的讲座，大多数的主题都会与“设计哲学”这样的话题有关。之所以这样，是因为我对自己的定位是 Go 语言的“传教士”，而不是“培训师”。我的出发点在于引起大家对 Go 的关注与兴趣，至于如何去一步步学习 Go 语言的语法知识，我相信兴趣是最好的老师。现今我们学习的平台足够强大，只要你真的很有兴趣，就一定能够学好 Go 语言。

Go 语言是非常简约的语言。简约的意思是少而精。Go 语言极力追求语言特性的最小化，如果某个语法特性只是少些几行代码，但对解决实际问题的难度不会产生本质的影响，那么这样的语法特性就不会被加入。Go 语言更关心的是如何解决程序员开发上的心智负担。如何减少代码出错的机会，如何更容易写出高品质的代码，是 Go 设计时极度关心的问题。

Go 语言也是非常追求自然(nature)的语言。Go 不只是提供极少的语言特性，并极力追求语言特性最自然的表达，也就是这些语法特性被设计成恰如多少人期望的那样，尽量避免惊异。事实上 Go 语言的语法特性上的争议是非常之少的。这些也让 Go 语言的入门门槛变得非常低。

今天我的话题重心是关于 Go 语言编程范式的流派问题。这仍然是关于“设计哲学”方面的。到目前为止，大家可能听过的编程范式主要如下：

- 过程式（代表：C）
- 面向对象（代表：Java、C#）
- 面向消息（代表：Erlang）
- 函数式（代表：Haskell、Erlang）

过程式编程的代表概念是过程（函数）。这是一个非常古老的流派，基本上所有语言都有过程式的影子。但是比较纯粹的过程式的主流语言比较少，通常比较古老，C 是其中最典型的代表。

面向对象编程是目前广为接受、影响极其深远的流派。面向对象有很多概念：如类、方法、属性、重载、多态（虚函数）、构造和析构、继承等等。Java、C#是其中最典型的代表。Go 语言支持面向对象，但将特性最小化。Go 语言中有结构体（类似面向对象中的类），结构体可以有方法，这就是 Go 对面向对象支持的所有内容。Go 语言的面向对象特征少得可怜。结构体是构成复合对象的基础，只要有组合，通常就由结构体，像 C 这样的过程式语言，照样有结构体。所以 Go 身上没有多少面向对象的烙印，Go 甚至反对继承，拒绝提供继承语法。

面向消息编程是个比较小众的编程流派，因为分布式与并发编程的强烈诉求而崛起。面向消息编程的主体思想是推荐基于消息而不是基于锁和共享内存进行并发编程。Erlang 语言是面向消息编程的代表。Go 语言中有面向消息的影子。因为 Go 语言中有 channel，可以让执行体（goroutine）之间相互发送消息。但 channel 只是 Go 语言的基础语法特性，Go 并没有杜绝锁和共享内存，所以它并不能算面向消息编程流派。

函数式编程也是一个小众的流派，尽管历史非常悠久。函数式编程中有些概念如：闭包、柯里化、变量不可变等。Haskell、Erlang 都是这个流派的代表。函数式编程之所以小众，个人认为最重要的原因，是理论基础不广为人知。我们缺乏面向函数式编程的数据结构学。因为变量不可变，数据结构学需要用完全不同思维方式来表达。比如在传统命令式的编程方式中，数组是最简单的基础数据结构，但函数式编程中，数组这样的数据结构很难提供（修改数组的一个元素成本太高，Erlang 语言中数组这个数据结构很晚才引入，用 tree 来模拟数组）。Go 语言除了支持闭包外，没有太多函数式的影子。

Go 语言有以上每一流派的影子，但都只是把这些流派的最基础的概念吸收，这些特性很基础，很难作为一个流派的关键特征来看。所以从编程范式上来说，个人认为 Go 语言不属于以上任何流派。如果非要说一个流派，Go 语言类似 C++，应该算“多范式”流派的。C++ 是主流语言中，几乎是唯一一门大力宣扬多范式编程理念的语言。C++ 主要支持的编程范式是过程式编程、面向对象编程、泛型编程（我们上面没有把泛型编程列入讨论的流派之中）。C++ 对这些流派的主要特性支持都很完整，说“多范式”名副其实。但 Go 不一样的是，每个流派的特性支持都很基础，这些特性只能称之为功能，并没有形成范式。

Go 语言在吸收这些流派精华的基础上，开创了自己独特的编程风格：一种基于连接与组合的语言。

连接，指的是组件的耦合方式，也就是组件是如何被串联起来的。组合，是形成复合对象的基础。连接与组合都是语言中非常平凡的概念，但 Go 语言恰恰是在平凡之中见神奇。

让我们从 Unix 谈起。Go 语言与 Unix、C 语言有着极深的渊源。Go 语言的领袖们参与甚至主导了 Unix 和 C 语言的设计。Ken Thompson 甚至算得上 Unix 和 C 语言的鼻祖。Go 语言亦深受 Unix 和 C 语言的设计哲学影响。

在 Unix 世界里，组件就是应用程序（app），每个 app 可大体抽象为：

- 输入：stdin（标准输入），params（命令行参数）
- 输出：stdout（标准输出）
- 协议：text（data stream）

不同的应用程序（app）如何连接？答案是：管道（pipeline）。在 Unix 世界中大家对这样的东西已经很熟悉了：

```
app1 params1 | app2 params2
```

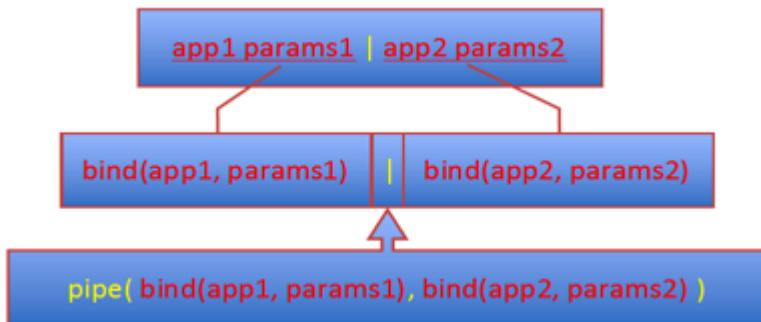
通过管道（pipeline），可以将一个应用程序的输出（stdout）转换为另一个应用程序的输入（stdin）。更为神奇的一点，是这些应用程序是并行执行的。app1 每产生一段输出，立即会被 app2 所处理。所以管道（pipeline）称得上是最古老，同时也是极其优秀的并行设施，简单而强大。

需要注意的是，Unix 世界中不同应用程序直接是松散耦合的。上游 app 的输出是 xml 还是 json，下游 app 需要知道，但并无任何强制的约束。同一输出，不同的下游 app，对协议的理解甚至都可能并不相同。例如，上游 app 输出一段 xml 文本，对于某个下游 app 来说，是一颗 dom 树，但对 linecount 程序来说只是一个分行的文本，对于英文单词词频统计程序来说，是一篇英文文章。

为了方便理解，我们先尝试在 Go 语言中模拟整个 Unix 的管道（pipeline）机制。首先是应用程序（app），我们抽象为：

```
func(in io.Reader, out io.Writer, args []string)
```

我们按下图来对应 Unix 与 Go 代码的关系：



也就是说，Unix 中的

```
app1 params1 | app2 params2
```

对应 Go 语言中是：

```
pipe( bind(app1, params1), bind(app2, params2) )
```

其中，bind 函数实现如下：

```
func bind(
    app func(in io.Reader, out io.Writer, args []string),
    args []string
) func(in io.Reader, out io.Writer) {
    return func(in io.Reader, out io.Writer) {
        app(in, out, args)
    }
}
```

要理解 bind 函数，需要先理解“闭包”。Go 语言中，应用程序以一个闭包的形式体现。如果你熟悉函数式编程，不难发现，这个 bind 函数其实就是所谓的柯里化（currying）。

pipe 函数如下：

```

func pipe(
    app1 func(in io.Reader, out io.Writer),
    app2 func(in io.Reader, out io.Writer)
) func(in io.Reader, out io.Writer) {
    return func(in io.Reader, out io.Writer) {
        pr, pw := io.Pipe()
        defer pw.Close()
        go func() {
            defer pr.Close()
            app2(pr, out)
        }()
        app1(in, pw)
    }
}
    
```

要理解 pipe 函数，除了“闭包”外，需要知晓 defer 关键字和 goroutine (go 关键字)。defer 语句会在函数退出时执行（无论是否发生了异常），通常用于资源的清理操作（比如关闭文件句柄等）。有了 defer 语句，Go 语言中的错误处理代码显得非常优雅。在一个正常的函数调用前加上 go 关键字，就会使得该函数在新的 goroutine 中并行执行。理解了这些背景，这个 pipe 函数不难理解，无非是：先创建一个管道，让 app1 读入数据 (in)，并向管道的写入端 (pw) 输出，启动一个新 goroutine，让 app2 从管道的读入端读取数据，并将处理结果输出 (out)。这样得到的 app 就是 app1 和 app2 的组合了。

你甚至可以对多个 app 进行组合：

```

func pipe(apps ...func(in io.Reader, out io.Writer)) func(in io.Reader, out io.Writer) {
    if len(apps) == 0 { return nil }
    app := apps[0]
    for i := 1; i < len(apps); i++ {
        app1, app2 := app, apps[i]
        app = func(in io.Reader, out io.Writer) {
            pr, pw := io.Pipe()
            defer pw.Close()
            go func() {
                defer pr.Close()
                app2(pr, out)
            }()
            app1(in, pw)
        }
    }
    return app
}
    
```

我们举个比较实际的例子，假设我们有 2 个应用程序 tar(打包)、gzip(压缩)：

```

func tar(io.Reader, out io.Writer, files []string)
    
```

```
func gzip(in io.Reader, out io.Writer)
```

那么打包并压缩的代码是：

```
pipe( bind(tar, files), gzip )(nil, out)
```

通过对管道 (pipeline) 的模拟我们可以看出，Go 语言对并行支持是非常强大的，这主要得益于 Go 的轻量级进程 (goroutine)。

为了体现 Go 语言在连接方式上的特别之处，我准备拿经典的面向对象设计案例（故事名不妨叫 Shape ）来比较。

在 Java 中，Shape 这个故事是这样的：

首先定义 Shape 接口：

```
interface Shape {
    double area(); // 为简化，我们只是求面积
}
```

然后提供各种具体的 Shape 类型：

```
class Circle implement Shape {
    private double x, y, r;
    public double area() { return math.Pi/2 * this.r * this.r; }
}

class Rect implement Shape {
    private double x, y, w, h;
    public double area() { return this.w * this.h; }
}
```

最后，我们写一个算法，求一系列 Shape 的总面积：

```

class Algorithm {
    static public double area(Shape... shapes) {
        double result = 0;
        for (Shape shape: shapes) {
            result += shape.area();
        }
        return result;
    }
}
    
```

在 Java 中，组件 Circle、Rect、Algorithm.area 之间，是通过 Shape 接口串联的。比较麻烦的是，Circle、Rect 都需要显式地声明自己实现了 Shape 这个契约，这加深了组件间的耦合。

我们来看看 Go 语言怎么做：

首先，我们定义各种具体的 Shape 类型：

```

type Circle struct {
    x, y, r float64
}
func (this *Circle) Area() float64 {
    return math.Pi/2 * this.r * this.r
}

type Rect struct {
    x, y, w, h float64
}
func (this *Rect) Area() float64 {
    return this.w * this.h
}
    
```

然后我们实现求多个 Shape 总面积的 Area 组件：

```

type Shape interface {
    Area() float64
}

func Area(shapes ...Shape) float64 {
    var result float64
    for _, shape := range shapes {
        result += shape.Area()
    }
    return result
}
    
```

看到这里，你可能误解我在做代码交换的游戏——看起来和 Java 版本的差别只是 Shape 这个 interface 挪了下位置。但这实际上是思维方式的变化。为了凸显这种变化，我把 Area 组件的代码改变下：

```
func Area(shapes ...interface{ Area() float64 }) float64 {
    var result float64
    for _, shape := range shapes {
        result += shape.Area()
    }
    return result
}
```

现在和 Java 版本的差异就很显著了：Shape 接口不见了！！！再独立去看看 Circle、Rect、Area 组件，每个组件都很独立，看不到任何额外的耦合代码。但神奇之处在于他们确实可以工作在一起：

```
area := Area( &Rect{w: 20, h: 30}, &Circle{r: 20} )
fmt.Println("area:", area)
```

我们再回头看看 Unix 的组件的连接方式，不难发现 Go 和 Unix 类似的地方：组件之间的连接是松散耦合的，彼此之间有最自然的独立性。但 Go 和 Unix 也有不同的地方，Unix 的契约基于文本流来表达，可能出现连接错误（比如上游输出 xml 文本，但下游期望是 json），在运行时才发现。但 Go 语言是静态编译语言，组件间的协议通过 interface 描述，并在编译期进行检查，如果不合适直接编译失败，如：

```
area := Area( &Rect{w: 20, h: 30}, &Circle{r: 20}, &Foo{} )
fmt.Println("area:", area)
```

如果上面的 Foo 类型没有 Area 方法，则编译不能通过。这种编译期检查是代码品质重要保障。作为对比的是，动态语言如 Python、PHP 等，可以很容易做到类似 Go 语言的松散耦合方式，但是由于没有强制的契约检查，如果误传一个对象实例，就可能出现各种非预期的行为，包括运行时崩溃。

一些面向对象语言的拥护者可能会认为 Java 这种明确指定 implement 的接口，使得程序更规范、在工程管理上更容易受控。但我个人持完全相反的观点，认为这完全是方向性的错误。原因是这种接口定义方式违背了事物的因果关系。举个

例子，假设我们实现了一个 File 类，它有 4 个方法 :Read、Write、Seek、Close。那么 File 类需要从哪些接口继承呢？Reader（注：我这里是按 Go 语言风格来命名接口，在 Java 中可能会倾向于 Readable，C++ 中则可能会倾向于 IRead 这样的名字）？Writer？Seeker？Closer？ReadWriteWriter？ReadWriteSeeker？ReadWriteSeekCloser？脱离实际需求，这个问题并无正确答案。要满足所有可能的用况，File 类最好从所有这些接口继承，总共需要继承的接口达 $2^4 - 1 = 15$ 种。这很恐怖，因为当某个类有 10 个公开方法的时候，需要继承的接口达 $2^{10} - 1 = 1023$ 种之多！

Go 语言对组件连接方式的调整是革命性的。可以预期的是，它必将慢慢影响目前各种主流的静态类型语言，逐步从错误的方向上修正过来。



全球软件开发大会

International Software Development Conference
[北京站] 2013

- Douglas Crockford

JSON发明人、《JavaScript精粹》作者

- Jesse Newland

GitHub资深运维工程师

- Kevlin Henney

《97件程序员必须知道的事》作者

- Chris Richardson

《POJOs in Action》作者

× 更有国内讲师，来自——百度、阿里巴巴、豆瓣、Redhat、新浪、网易等

专题及出品人：

- 知名网站案例分析
- 新锐编程语言
- 持续集成与持续交付
- 跨终端的Web
- 企业级开发框架
- 大数据与NoSQL
- 深入移动开发
- 自动化运维

- 冯大辉
- 陈皓
- 乔梁
- 鄢学鶴
- 方越
- 孙振南
- 崔海斌
- 黄冬

- 构建高效能团队
- 机器学习与推荐算法
- 优秀测试实践分析
- 敏捷嘉年华
- 云计算迷思
- 开放平台进行时
- Server端Node.js
- 用户体验与产品设计

- 吴永强
- 谷文栋
- 高楼
- 熊妍妍
- 蒋炜航
- 朱建庭
- 田永强
- 石岩



2013年4月25-27日 | 北京·国际会议中心

主办方：



本期专栏 | Column

欣欣向荣的 Ruby 家族

作者 丁雪丰

诞生于 1993 年的 [Ruby](#) 即将迎来自己的 20 岁生日，估计松本行弘（ Matz ）先生 20 年前也没有想到 Ruby 能成为一门流行的语言，长期出现在 [TIOBE 编程语言排行榜](#) 前 20 之列，并且有 [逐步上升之势](#)。Ruby 的爱好者遍布世界各地，在中国也有庞大的 [RubyChina 社区](#)。而且，除了的 MRI Ruby 之外，还诞生了很多与其兼容的 Ruby 实现，有的旨在提升性能，而有的则是为了充分利用其他平台提供的资源，还出现了专门针对移动设备和嵌入式设备的版本。

更好的性能

早期的 Ruby 虽然受到很多人的追捧，但是性能并不理想，很多人都质疑 Ruby 的性能，建议系统中的关键部分用 C 来写。而且 Ruby 的 GC 也存在很多问题，打不打 [MBARI 补丁](#)，差异巨大。出于性能的考虑，很多人开始从动手实现性能更好的虚拟机，正式他们的努力，让 Ruby 的性能获得了质的飞跃。

首当其冲的是 Koichi Sasada 开发的 [YARV](#)（ Yet Another Ruby VM ），该项目的唯一目的就是要打造世界上最快的 Ruby 虚拟机。从早期的一些[评测](#)来看，YARV 为 Ruby 带来了巨大的性能提升，而它也成为了后来 Ruby 1.9 的官方解释器，自然不必多说了。

其次是 [Rubinius](#)，在它的首页最上方有着这么一句话：

An environment for the Ruby programming language providing performance, accessibility, and improved programmer productivity.

可见它的关注点是性能、可访问性以及开发者的生产效率。Rubinius 拥有自己的字节码虚拟机，运行时，Ruby 代码会先变为字节码，随后再通过 LLVM 将字节码编译为高效的字节码。而且，它还有一套精确的、带压缩的分代垃圾收集器，进一步提升了 GC 效率。

除此之外，Rubinius 还为我们带来了另一个重要的贡献，它创建了 [RubySpec](#)，为其他 Ruby 实现提供了一套可以参考的实现规范，后续很多组织也参与其中，添砖加瓦，贡献了很多测试。

第三个登场的是 Phusion 出品的 [REE](#) (Ruby Enterprise Edition)，REE 的焦点更多地集中在服务端的 Rails 应用上，与 [Phusion Passenger](#) 结合在一起，可以极大程度上降低 Rails 应用的内存开销，并且提升服务响应速度。REE 的主页上提供了一套[官方的对比](#)，Apache (worker MPM) + Ruby Enterprise Edition + Phusion Passenger 的组合在这两方面要明显优于其他组合。

REE 还集成了很多第三方的补丁，比如之前提到的 MBARI、[RailsBench](#) 等等，可惜它仅兼容到 Ruby 1.8.7，有点跟不上时代的节奏。

在性能上表现出众的还有 JRuby，考虑到它是运行于 Java 平台之上的，因此会在后续小结中进行重点介绍。

更多的平台

[JRuby](#) 是运行于 Java 平台之上 Ruby 实现，最近刚刚发布了 1.7.2 版本，能很好地兼容 Ruby 1.8.7 和 1.9.2 (JRuby 从 1.7.0 开始默认使用 1.9 模式，之前一直默认 1.8 模式)。JRuby 让 Ruby 程序能够充分利用 Java 的庞大资源，同时还提供了更好的性能。如今的 Ruby 拥有庞大的类库，但在 Rails 刚刚让 Ruby 成为众人焦点之时，Ruby 的资源并没有这么充分，即使是现在，能够借助 Java 的力量还是非常有优势的。

在 JRuby 诞生初期，开发者的主要精力集中在对 MRI 的兼容性上，但在兼容性不再是个问题时，他们就努力提升性能，他们做到了，还越做越好。例如，借助 Java 的力量，JRuby 能真正做到多线程，而不是 “Green Thread”；Java 7 中新增的 invokedynamic，能够为基于 JVM 的脚本语言带来不小的性能提升，JRuby 1.7 就开始支持该特性了。JRuby 的核心开发者 [Charles Oliver Nutter](#) 经常会在博客上发表一些[文章](#)，介绍 JRuby 的优化方法和经验，从最近的一篇[文章](#)表明，JRuby 1.7.2 在很多方面的性能要远好于尚未正式发布的 Ruby 2.0。

有不少 Ruby 开发者之前都是 Java 开发者(Bruce A. Tate 就写过一本很出名的 [《From Java to Ruby》](#))，因此如果能够充分利用之前 Java 的各种经验，也

不失为一件好事。一个有着丰富 Java 经验的开发者 知道怎么对 JVM 进行调优，那他也一定能把 JRuby 的程序[调校](#)得很好。

Android 上运行的是 Java 程序，那么应该也能运行 JRuby，于是就诞生了 [Ruboto](#)——这是一个用来开发原生 Android 应用的框架及工具链。此外，将 Ruby 代码编译为 Java 字节码后交付给用户，还能在一定程度上保护源代码，这也算是个额外的收获吧。

讲完了运行在 Java 平台上的 Ruby，再来看看.NET 平台上的 Ruby——[IronRuby](#)。与 JRuby 类似，IronRuby 让 Ruby 能利用.NET Framework 上的资源，.NET 开发者也能够使用 Ruby 快速地完成很多任务。可惜 IronRuby 在 2011 年 3 月之后就再也没有更新过，因此这里就不再阐述了。

Mac OS 自带了 MRI Ruby，但其实也有构建于 Mac OS X 核心技术（例如 Objective-C 运行时）之上的 Ruby 1.9 实现，即 [MacRuby](#)。它的目标是在不牺牲性能的前提下享受 Ruby 的乐趣，用它来创建完善的 Mac OS X 应用程序。从官方的[入门教程](#)来看，能够很方便地使用 MacRuby 构建带 GUI 的 Mac 应用。

除了 Mac 应用，你一定也很想知道能否用 MacRuby 来开发 iOS 端的应用，到目前为止 Objective-C 仍然是官方的首选，也有人[泼过 MacRuby 的冷水](#)，但 MacRuby 的开发者并没有放弃，他们开发了 [RubyMotion](#)，让开发者能方便地使用 Ruby 来开发 iPhone 和 iPad 的原生 iOS 应用，且[得到众多好评](#)。

上面提到了 Ruboto 和 RubyMotion，就不得不提另一个让开发者能够使用 Ruby 来开发移动应用的 [MobiRuby](#) 了。MobiRuby 使用了松本行弘开发的 [mruby](#)，这是一个轻量级的 Ruby 实现，可以运行 Ruby 程序，但主要的目的是嵌入其他程序，并且运行在内存受限的小型设备上，比如松本先生介绍过的 Sakura Board。MobiRuby 旨在替代移动平台上的 Objective-C/C 和 Java，目前已经能够编写 iOS 应用，更重要的是[已经有 MobiRuby 的程序被 AppStore 接受了](#)。开发团队承诺，后续也会有针对 Android 的 MobiRuby 出现。

最后再来看看 [MagLev](#)，它是构建于 VMware GemStone/S 上的 64 位 Ruby 实现，让开发者能充分发挥 GemStone/S 的优势，比如有更好的性能、分布式共享缓存、企业级 NoSQL 数据管理能力等等。更重要的是它能透明地管理远大于内存上限的 TB 级别的数据和代码，出于这个原因，也许可以将 MagLev 看成一个能够运行 Ruby、存储 Ruby 原生对象的分布式数据库。

在看了这么多 Ruby 家族的成员之后 , 不知您有何感想 ? 一定觉得这是个充满活力的语言吧。如果您一直在使用官方的 Ruby 实现 , 那么不妨试试其他的实现 , 也许会有别样的感觉。

推荐文章 | Article

深入理解 Java 内存模型（一）——基础

作者 程晓明

并发编程模型的分类

在并发编程中，我们需要处理两个关键问题：线程之间如何通信及线程之间如何同步（这里的线程是指并发执行的活动实体）。通信是指线程之间以何种机制来交换信息。在命令式编程中，线程之间的通信机制有两种：共享内存和消息传递。

在共享内存的并发模型里，线程之间共享程序的公共状态，线程之间通过写-读内存中的公共状态来隐式进行通信。在消息传递的并发模型里，线程之间没有公共状态，线程之间必须通过明确的发送消息来显式进行通信。

同步是指程序用于控制不同线程之间操作发生相对顺序的机制。在共享内存并发模型里，同步是显式进行的。程序员必须显式指定某个方法或某段代码需要在线程之间互斥执行。在消息传递的并发模型里，由于消息的发送必须在消息的接收之前，因此同步是隐式进行的。

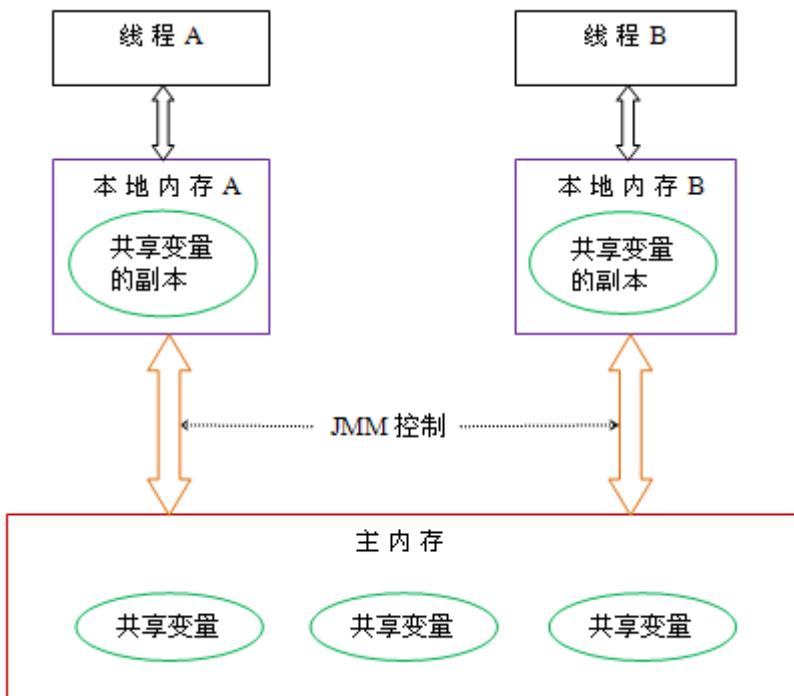
Java 的并发采用的是共享内存模型，Java 线程之间的通信总是隐式进行，整个通信过程对程序员完全透明。如果编写多线程程序的 Java 程序员不理解隐式进行的线程之间通信的工作机制，很可能遇到各种奇怪的内存可见性问题。

Java 内存模型的抽象

在 java 中，所有实例域、静态域和数组元素存储在堆内存中，堆内存在线程之间共享（本文使用“共享变量”这个术语代指实例域、静态域和数组元素）。局部变量（Local variables）、方法定义参数（java 语言规范称之为 formal method parameters）和异常处理器参数（exception handler parameters）不会在线程之间共享，它们不会有内存可见性问题，也不受内存模型的影响。

Java 线程之间的通信由 Java 内存模型（本文简称为 JMM）控制，JMM 决定一个线程对共享变量的写入何时对另一个线程可见。从抽象的角度来看，JMM 定义了线程和主内存之间的抽象关系：线程之间的共享变量存储在主内存（main

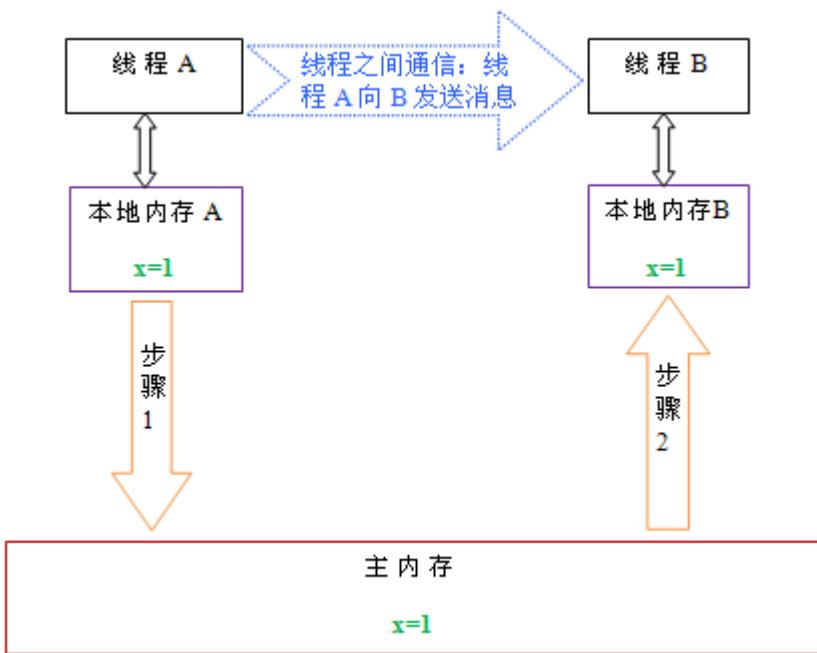
memory) 中，每个线程都有一个私有的本地内存 (local memory)，本地内存中存储了该线程以读/写共享变量的副本。本地内存是 JMM 的一个抽象概念，并不真实存在。它涵盖了缓存、写缓冲区、寄存器以及其他硬件和编译器优化。Java 内存模型的抽象示意图如下：



从上图来看，线程 A 与线程 B 之间如要通信的话，必须要经历下面 2 个步骤：

1. 首先，线程 A 把本地内存 A 中更新过的共享变量刷新到主内存中去。
2. 然后，线程 B 到主内存中去读取线程 A 之前已更新过的共享变量。

下面通过示意图来说明这两个步骤：



如上图所示，本地内存 A 和 B 有主内存中共享变量 x 的副本。假设初始时，这三个内存中的 x 值都为 0。线程 A 在执行时，把更新后的 x 值（假设值为 1）临时存放在自己的本地内存 A 中。当线程 A 和线程 B 需要通信时，线程 A 首先会把自己本地内存中修改后的 x 值刷新到主内存中，此时主内存中的 x 值变为了 1。随后，线程 B 到主内存中去读取线程 A 更新后的 x 值，此时线程 B 的本地内存的 x 值也变为了 1。

从整体来看，这两个步骤实质上是线程 A 在向线程 B 发送消息，而且这个通信过程必须要经过主内存。JMM 通过控制主内存与每个线程的本地内存之间的交互，来为 java 程序员提供内存可见性保证。

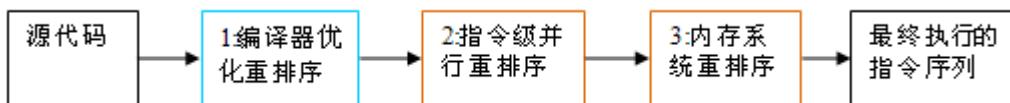
重排序

在执行程序时为了提高性能，编译器和处理器常常会对指令做重排序。重排序分三种类型：

1. 编译器优化的重排序。编译器在不改变单线程程序语义的前提下，可以重新安排语句的执行顺序。
2. 指令级并行的重排序。现代处理器采用了指令级并行技术（Instruction-Level Parallelism，ILP）来将多条指令重叠执行。如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。

3. 内存系统的重排序。由于处理器使用缓存和读/写缓冲区，这使得加载和存储操作看上去可能是在乱序执行。

从 java 源代码到最终实际执行的指令序列，会分别经历下面三种重排序：



上述的 1 属于编译器重排序，2 和 3 属于处理器重排序。这些重排序都可能会导致多线程程序出现内存可见性问题。对于编译器，JMM 的编译器重排序规则会禁止特定类型的编译器重排序（不是所有的编译器重排序都要禁止）。对于处理器重排序，JMM 的处理器重排序规则会要求 java 编译器在生成指令序列时，插入特定类型的内存屏障（memory barriers，intel 称之为 memory fence）指令，通过内存屏障指令来禁止特定类型的处理器重排序（不是所有的处理器重排序都要禁止）。

JMM 属于语言级的内存模型，它确保在不同的编译器和不同的处理器平台之上，通过禁止特定类型的编译器重排序和处理器重排序，为程序员提供一致的内存可见性保证。

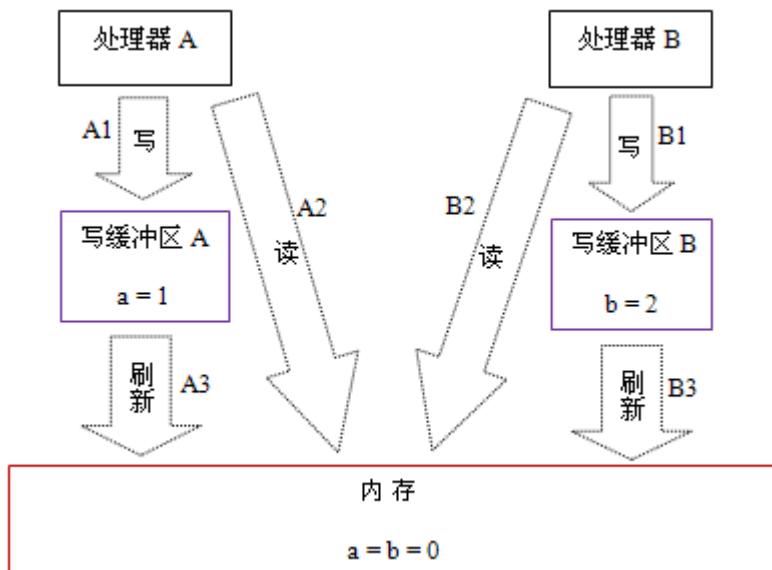
处理器重排序与内存屏障指令

现代的处理器使用写缓冲区来临时保存向内存写入的数据。写缓冲区可以保证指令流水线持续运行，它可以避免由于处理器停顿下来等待向内存写入数据而产生的延迟。同时，通过以批处理的方式刷新写缓冲区，以及合并写缓冲区中对同一内存地址的多次写，可以减少对内存总线的占用。虽然写缓冲区有这么多好处，但每个处理器上的写缓冲区，仅仅对它所在的处理器可见。这个特性会对内存操作的执行顺序产生重要的影响：处理器对内存的读/写操作的执行顺序，不一定与内存实际发生的读/写操作顺序一致！为了具体说明，请看下面示例：

Processor A	Processor B
a = 1; //A1 x = b; //A2	b = 2; //B1 y = a; //B2
初始状态 : a = b = 0	

处理器允许执行后得到结果： $x = y = 0$

假设处理器 A 和处理器 B 按程序的顺序并行执行内存访问，最终却可能得到 $x = y = 0$ 的结果。具体的原因如下图所示：



这里处理器 A 和处理器 B 可以同时把共享变量写入自己的写缓冲区(A1 ,B1)，然后从内存中读取另一个共享变量 (A2 , B2)，最后才把自己写缓存区中保存的脏数据刷新到内存中 (A3 , B3)。当以这种时序执行时，程序就可以得到 $x = y = 0$ 的结果。

从内存操作实际发生的顺序来看，直到处理器 A 执行 A3 来刷新自己的写缓存区，写操作 A1 才算真正执行了。虽然处理器 A 执行内存操作的顺序为：A1->A2，但内存操作实际发生的顺序却是：A2->A1。此时，处理器 A 的内存操作顺序被重排序了（处理器 B 的情况和处理器 A 一样，这里就不赘述了）。

这里的关键是，由于写缓冲区仅对自己的处理器可见，它会导致处理器执行内存操作的顺序可能会与内存实际的操作执行顺序不一致。由于现代的处理器都会使用写缓冲区，因此现代的处理器都会允许对写-读操作重排序。

下面是常见处理器允许的重排序类型的列表：

	Load-Load	Load-Store	Store-Store	Store-Load	数据依赖
sparc-TSO	N	N	N	Y	N

x86	N	N	N	Y	N
ia64	Y	Y	Y	Y	N
PowerPC	Y	Y	Y	Y	N

上表单元格中的“N”表示处理器不允许两个操作重排序，“Y”表示允许重排序。

从上表我们可以看出：常见的处理器都允许 Store-Load 重排序；常见的处理器都不允许对存在数据依赖的操作做重排序。sparc-TSO 和 x86 拥有相对较强的处理器内存模型，它们仅允许对写-读操作做重排序（因为它们都使用了写缓冲区）。

※注 1 : sparc-TSO 是指以 TSO(Total Store Order)内存模型运行时，sparc 处理器的特性。

※注 2 : 上表中的 x86 包括 x64 及 AMD64。

※注 3 :由于 ARM 处理器的内存模型与 PowerPC 处理器的内存模型非常类似，本文将忽略它。

※注 4 : 数据依赖性后文会专门说明。

为了保证内存可见性，java 编译器在生成指令序列的适当位置会插入内存屏障指令来禁止特定类型的处理器重排序。JMM 把内存屏障指令分为下列四类：

屏障类型	指令示例	说明
LoadLoad Barriers	Load1; LoadLoad; Load2	确保 Load1 数据的装载，之前于 Load2 及所有后续装载指令的装载。
StoreStore Barriers	Store1; StoreStore; Store2	确保 Store1 数据对其他处理器可见（刷新到内存），之前于 Store2 及所有后续存储指令的存储。
LoadStore	Load1; LoadStore;	确保 Load1 数据装载，之前于 Store2 及所有后续

Barriers	Store2	的存储指令刷新到内存。
StoreLoad Barriers	Store1; StoreLoad; Load2	确保 Store1 数据对其他处理器变得可见（指刷新到内存），之前于 Load2 及所有后续装载指令的装载。StoreLoad Barriers 会使该屏障之前的所有内存访问指令（存储和装载指令）完成之后，才执行该屏障之后的内存访问指令。

StoreLoad Barriers 是一个“全能型”的屏障，它同时具有其他三个屏障的效果。现代的多处理器大都支持该屏障（其他类型的屏障不一定被所有处理器支持）。执行该屏障开销会很昂贵，因为当前处理器通常要把写缓冲区中的数据全部刷新到内存中（buffer fully flush）。

happens-before

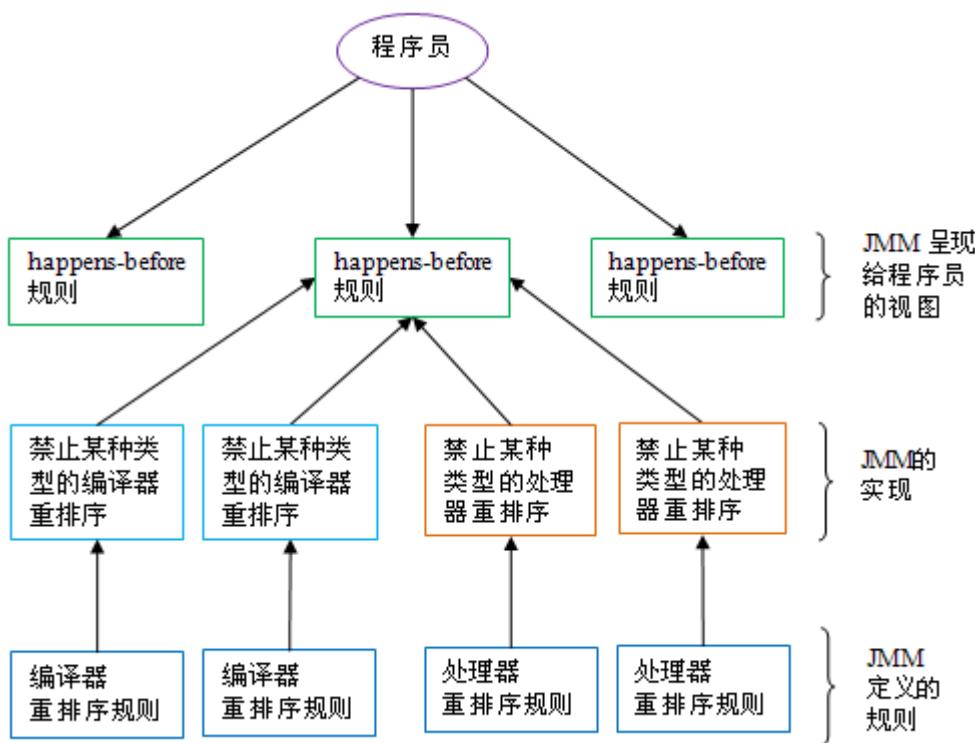
从 JDK5 开始，java 使用新的 JSR -133 内存模型（本文除非特别说明，针对的都是 JSR- 133 内存模型）。JSR-133 提出了 happens-before 的概念，通过这个概念来阐述操作之间的内存可见性。如果一个操作执行的结果需要对另一个操作可见，那么这两个操作之间必须存在 happens-before 关系。这里提到的两个操作既可以是在一个线程之内，也可以是在不同线程之间。与程序员密切相关的 happens-before 规则如下：

- 程序顺序规则：一个线程中的每个操作，happens- before 于该线程中的任意后续操作。
- 监视器锁规则：对一个监视器锁的解锁，happens- before 于随后对这个监视器锁的加锁。
- volatile 变量规则：对一个 volatile 域的写，happens- before 于任意后续对这个 volatile 域的读。
- 传递性：如果 A happens- before B，且 B happens- before C，那么 A happens- before C。

注意，两个操作之间具有 happens-before 关系，并不意味着前一个操作必须要在后一个操作之前执行 !happens-before 仅仅要求前一个操作（执行的结果）对后一个操作可见，且前一个操作按顺序排在第二个操作之前（the first is

visible to and ordered before the second)。happens-before 的定义很微妙，后文会具体说明 happens-before 为什么要这么定义。

happens-before 与 JMM 的关系如下图所示：



如上图所示，一个 happens-before 规则通常对应于多个编译器重排序规则和处理器重排序规则。对于 java 程序员来说，happens-before 规则简单易懂，它避免程序员为了理解 JMM 提供的内存可见性保证而去学习复杂的重排序规则以及这些规则的具体实现。

关于作者

程晓明，Java 软件工程师，国家认证的系统分析师、信息项目管理师。专注于并发编程，就职于富士通南大。个人邮箱：asst2003@163.com。

原文链接：<http://www.infoq.com/cn/articles/java-memory-model-1>

相关内容

- [Java 那些事儿 - JavaOne 2011、CDI 和 Google Dart 语言](#)
- [Java 那些事：Java 7、JavaFX 2.0 以及 Vaadin 框架](#)

推荐文章

深入浅出 Node.js 游戏服务器开发--分布式聊天服务器搭建

作者 [谢骋超 彭阳](#)

在上一篇文章中， 我们介绍了游戏服务器的基本架构、相关框架和 Node.js 开发游戏服务器的优势。本文我们将通过聊天服务器的设计与开发，来更深入地理解 pomelo 开发应用的基本流程、开发思路与相关的概念。本文并不是开发聊天服务器的 tutorial，如果需要 tutorial 和源码可以看文章最后的参考资料。

为什么是聊天服务器？

我们目标是搭建游戏服务器，为什么从聊天开始呢？

聊天可认为是简化的实时游戏，它与游戏服务器有着很多共通之处，如实时性、频道、广播等。由于游戏在场景管理、客户端动画等方面有一定的复杂性，并不适合作为 pomelo 的入门应用。聊天应用通常是 Node.js 入门接触的第一个应用，因此更适合做入门教程。

Pomelo 是游戏服务器框架，本质上也是高实时、可扩展、多进程的应用框架。除了在 library 中有一部分游戏专用的库，其余部分框架完全可用于开发高实时 web 应用。而且与现在有的 Node.js 高实时应用框架如 derby、socketstream、meteor 等比起来有更好的可伸缩性。

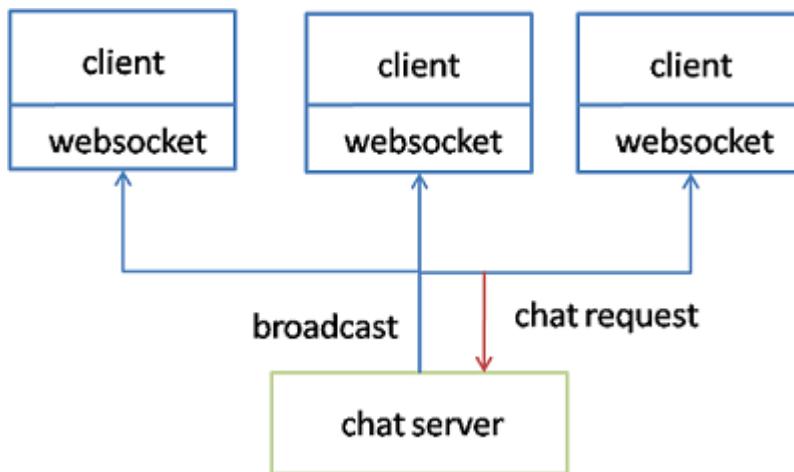
对于大多数开发者而言，Node.js 的入门应用都是一个基于 socket.io 开发的普通聊天室， 由于它是基于单进程的 Node.js 开发的，在可扩展性上打了一定折扣。例如要扩展到类似 irc 那样的多频道聊天室， 频道数量的增多必然会导致单进程的 Node.js 支撑不住。

而基于 pomelo 框架开发的聊天应用天生就是多进程的，可以非常容易地扩展服务器类型和数量。

从单进程到多进程，从 socket.io 到 pomelo

一个基于 socket.io 的原生聊天室应用架构，以 [uberchat](#) 为例。

它的应用架构如下图所示：



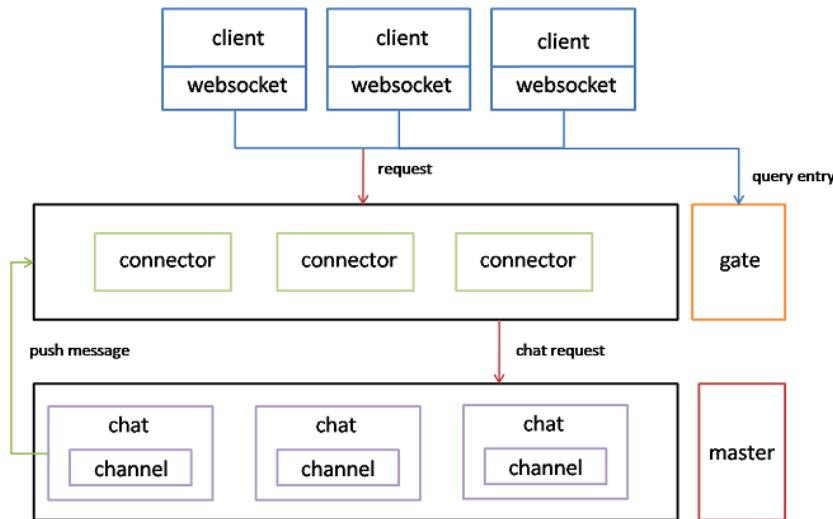
服务端由单个 Node.js 进程组成的 chat server 来接收 websocket 请求。

它有以下缺点：

1. 可扩展性差：只支持单进程的 Node.js，无法根据 room/channel 分区，也无法将广播的压力与处理逻辑的压力分开。
2. 代码量大：基于 socket.io 做了简单封装，服务端就写了约 430 行代码。

用 pomelo 来写这个框架可完全克服以上缺点，并且代码量只要区区 100 多行。

我们要搭建的 pomelo 聊天室具有如下的运行架构：



在这个架构里，前端服务器也就是 connector，专门负责承载连接，后端的聊天服务器则是处理具体逻辑的地方。这样扩展的运行架构具有如下优势：

- * 负载分离：这种架构将承载连接的逻辑与后端的业务处理逻辑完全分离，这样做是

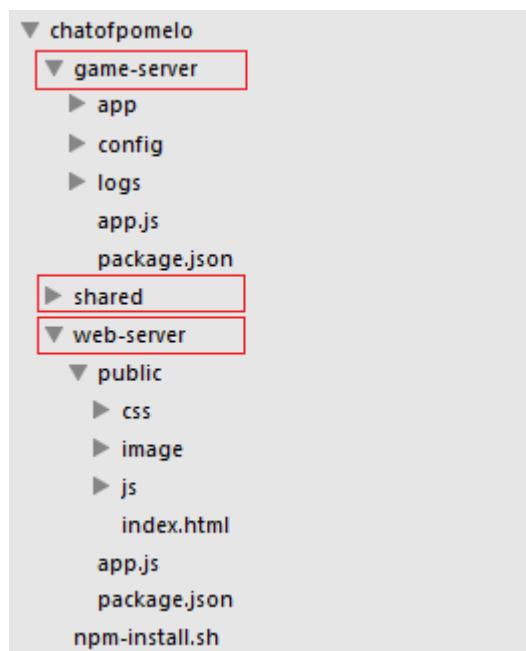
非常必要的，尤其是广播密集型应用（例如游戏和聊天）。密集的广播与网络通讯会占掉大量的资源，经过分离后业务逻辑的处理能力就不再受广播的影响。

- 切换简便：因为有了前、后端两层的架构，用户可以任意切换频道或房间都不需要重连前端的 websocket。
- 扩展性好：用户数的扩展可以通过增加 connector 进程的数量来支撑。频道的扩展可以通过哈希等算法负载均衡到多台聊天服务器上。理论上这个架构可以实现频道和用户的无限扩展。

聊天服务器开发架构

game server 与 web server

聊天服务器项目中分生成了 game-server 目录、web-server 目录与 shared 目录，如下图所示：



这样也将应用天然地隔离成了两个，game server 与 web server。

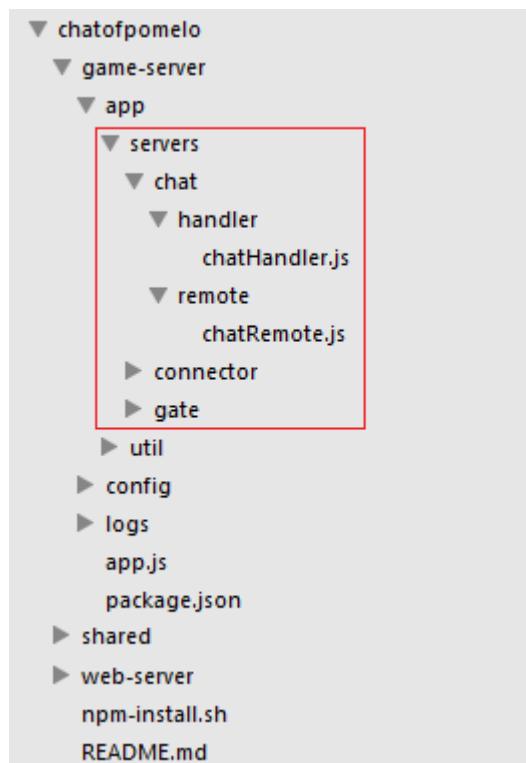
- Game server，即游戏服务器，所有的游戏服务器逻辑都在里实现。客户端通过 websocket(0.3 版会支持 tcp 的 socket)连到 game server。game-server/app.js 是游戏服务器的运行入口。
- Web server，即 web 服务器，也可以认为是游戏服务器的一个 web 客户端，所有客户端的 js 代码，web 端的 html、css 资源都存放在这

里，web 服务端的用户登录、认证等功能也在这里实现。pomelo 也提供了其它客户端，包括 ios、android、unity3D 等。

- Shared 目录，假如客户端是 web，由于服务端和客户端都是 javascript 写的，这时 Node.js 的代码重用优势就体现出来了。shared 目录下可以存放客户端、服务端共用的常量、算法。真正做到一遍代码，前后端共用。

服务器定义与应用目录

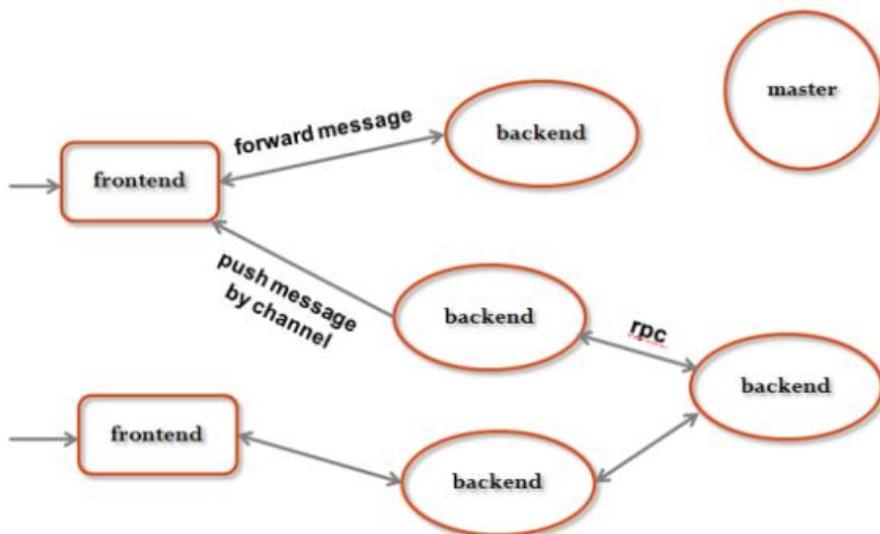
Game server 才是游戏服务器的真正入口，游戏逻辑都在里，我们简单看一下 game-server 的目录结构，如下图所示：



servers 目录下所有子目录定义了各种类型的服务器，而每个 servers 目录下基本都包含了 handler 和 remote 两个目录。这是 pomelo 的创新之处，用极简的配置实现游戏服务器的定义，后文会解释 handler 和 remote。

通过 pomelo，游戏开发者可以自由地定义自己的服务器类型，分配和管理进程资源。在 pomelo 中，根据服务器的职责不同，服务器主要分为前端服务器(frontend)和后端服务器(backend)两大类。其中，前端服务器负责承载客户端的连接和维护 session 信息，所有服务器与客户端的消息都会经过前端服务器；

后端服务器负责接收前端服务器分发过来的请求，实现具体的游戏逻辑，并把消息回推给前端服务器，最后发送给客户端。如下图所示：



动态语言的面向对象有个基本概念叫鸭子类型。在 pomelo 中，服务器的抽象也同样可以比喻为鸭子，服务器的对外接口只有两类，一类是接收客户端的请求，叫做 handler，一类是接收 RPC 请求，叫做 remote，handler 和 remote 的行为决定了服务器长什么样子。因此开发者只需要定义好 handler 和 remote 两类的行为，就可以确定这个服务器的类型。例如 chat 服务器目前的行为只有两类，分别是定义在 handler 目录中的 chatHandler.js，和定义在 remote 目录中的 chatRemote.js。只要定义好这两个类的方法，聊天服务器的对外接口就确定了。

搭建聊天服务器

准备知识

pomelo 的客户端服务器通讯

- pomelo 的客户端和服务器之间的通讯可以分为三种：

request-response

pomelo 中最常用的就是 request-response 模式，客户端发送请求，服务器异步响应。客户端的请求发送形式类似 ajax 类似：

```
```
pomelo.request(url, msg, function(data){});
```

第一个参数为请求地址，完整的请求地址主要包括三个部分：服务器类型、服务端相应的文件名及对应的方法名。第二个参数是消息体，消息体为 json 格式，第三个参数是回调函数，请求的响应将会把结果置入这个回调函数中返回给客户端。

- **notify**

notify 与 request—response 类似，唯一区别是客户端只负责发送消息到服务器，客户端不接收服务器的消息响应。

```
```
pomelo.notify(url, msg);
```

- **push**

push 则是服务器主动向客户端进行消息推送，客户端根据路由信息进行消息区分，转发到后。通常游戏服务器都会发送大量的这类广播。

```
```
pomelo.on(route, function(data){});
```

以上是 javascript 的 api，其它客户端的 API 基本与这个类型。由于 API 与 ajax 极其类似，所有 web 应用的开发者对此都不陌生。

## session 介绍

与 web 服务器类似，session 是游戏服务器存放用户会话的抽象。但与 web 不同，游戏服务器的 session 是基于长连接的，一旦建立就一直保持。这反而比 web 中的 session 更直接，也更简单。由于长连接的 session 不会 web 应用

一样由于连接断开重连带来 session 复制之类的问题，简单地将 session 保存在前端服务器的内存中是明智的选择。

在 pomelo 中 session 也是 key/value 对象 其主要作用是维护当前用户信息，例如：用户的 id，所连接的前端服务器 id 等。session 由前端服务器维护，前端服务器在分发请求给后端服务器时，会复制 session 并连同请求一起发送。任何直接在 session 上的修改，只对本服务器进程生效，并不会影响到用户的全局状态信息。如需修改全局 session 里的状态信息，需要调用前端服务器提供的 RPC 服务。

## channel 与广播

广播在游戏中是极其重要的，几乎大部分的消息都需要通过广播推送到客户端，再由客户端播放接收的消息。而 channel 则是服务器端向客户端进行消息广播的通道。可以把 channel 看成一个用户 id 的容器。把用户 id 加入到 channel 中成为当中的一个成员，之后向 channel 推送消息，则该 channel 中所有的成员都会收到消息。channel 只适用于服务器进程本地，即在服务器进程 A 创建的 channel 和在服务器进程 B 创建的 channel 是两个不同的 channel，相互不影响。

## 服务器之间 RPC 通讯

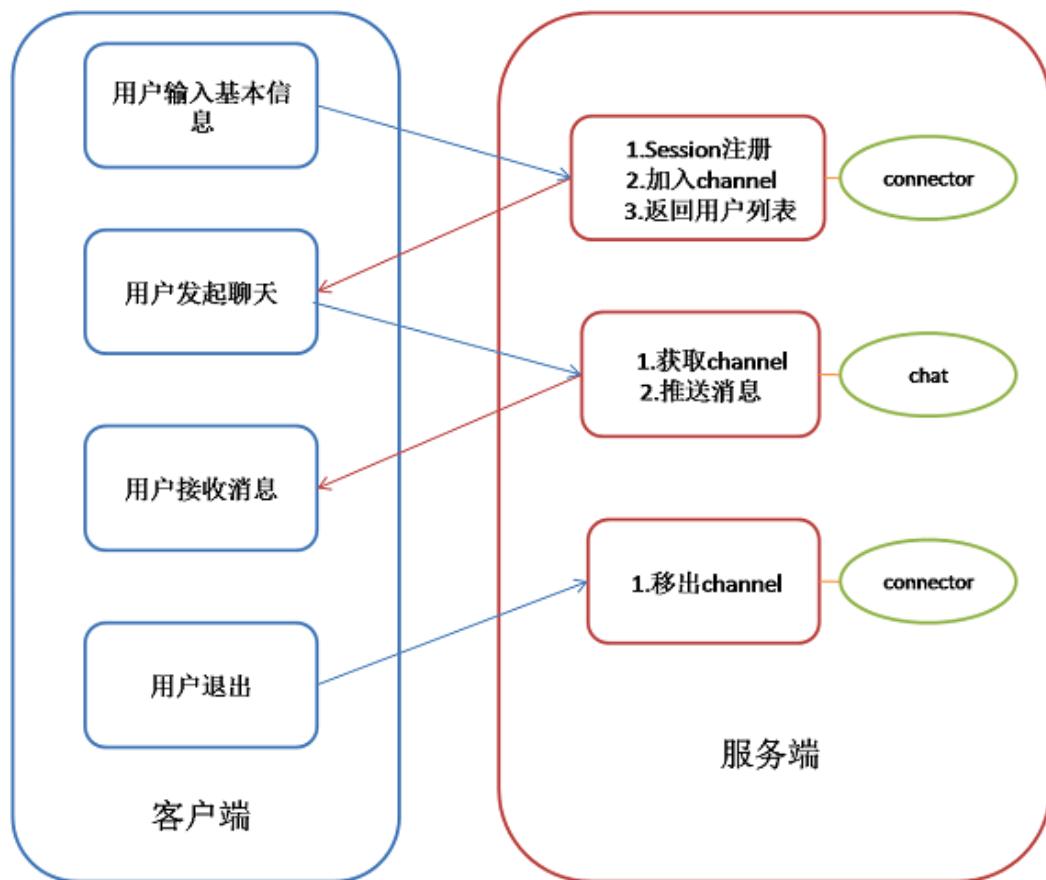
从之前的文章可以了解到，在 pomelo 中，游戏服务器其实是一个多进程相互协作的环境。各个进程之间通信，主要是通过底层统一的 RPC 框架来实现，服务器间的 RPC 调用也实现了零配置。具体 RPC 调用的代码如下：

```
```javascript
app.rpc.chat.chatRemote.add(session, uid, app.get
('serverId'), function(data){});
```

其中 app 是 pomelo 的应用对象，app.rpc 表明了是前后台服务器的 Remote rpc 调用，后面的参数分别代表服务器的名称、对应的文件名称及方法名。为了实现这个 rpc 调用，则只需要在对应的 chat/remote/中新建文件 chatRemote.js，并实现 add 方法。

聊天室流程概述

下图列出了聊天室进行聊天的完整流程：



通过以上流程，我们可以看到 pomelo 的基本请求流程和用法。本文不是聊天室的 tutorial，因此下面列出的代码不是完整的，而是用极简的代码来说明 pomelo 的使用流程和 api。

进入聊天室

客户端向前端服务器发起登录请求：

```

```javascript
pomelo.request('connector.entryHandler.enter',
 {user:userInfo}, function(){}
);
```

```

用户进入聊天室后，服务器端首先需要完成用户的 session 注册同时绑定用户离开事件：

```
```javascript
session.bind(uid);
session.on('closed', onUserLeave.bind(null, app));
```
```

另外，服务器端需要通过调用 rpc 方法将用户加入到相应的 channel 中；同时在 rpc 方法中，服务器端需要将该用户的上线消息广播给其他用户，最后服务器端向客户端返回当前 channel 中的用户列表信息。

```
```javascript
app.rpc.chat.chatRemote.add(session, uid,
serverId, function(){});
```
```

发起聊天

客户端向服务端发起聊天请求，请求消息包括聊天内容，发送者和发送目标信息。消息的接收者可以聊天室里所有的用户，也可以是某一特定用户。

服务器端根据客户端的发送的请求，进行不同形式的消息广播。如果发送目标是所有用户，服务器端首先会选择 channel 中的所有用户，然后向 channel 发送消息，最后前端服务器就会将消息分别发送给 channel 中取到的用户；如果发送目标只是某一特定用户，发送过程和之前完全一样，只是服务器端首先从 channel 中选择的只是一个用户，而不是所有用户。

```
```javascript
if(msg.target == '*')
 channel.pushMessage(param);
else
 channelService.pushMessageByUids(param,
 [{uid:uid, sid:sid}]);
```
```

接收聊天消息

客户端接收广播消息，并将消息并显示即可。

```
```javascript
pomelo.on('onChat', function() {
 addMessage(data.from, data.target, data.msg);
 $('#chatHistory').show();
});
```
```

退出聊天室

用户在退出聊天室时，必须完成一些清理工作。在 session 断开连接时，通过 rpc 调用将用户从 channel 中移除。在用户退出前，还需要将自己下线的消息广播给所有其他用户。

```
```javascript
 app.rpc.chat.chatRemote.kick(session, uid, serverId,
 channelName, null);
```

```

聊天服务器的可伸缩性与扩展讨论

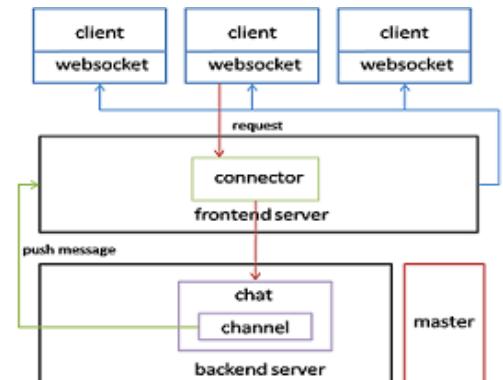
上一讲已经谈到 pomelo 在提供了一个高可伸缩性的运行架构，对于聊天服务器同样如此。如果想从单频道聊天室扩展到多频道聊天室，增加的代码几乎为零。大部分工作只是在进行服务器和路由的配置。对于服务器配置只需要修改 json 配置文件即可，而对于路由配置只需要增加一个路由算法即可。在 pomelo 中，开发者可以自己配置客户端到服务器的路由规则，这样会使得游戏开发更加灵活。

我们来看一下配置 json 文件对服务器运行架构的影响：

- 最简服务器与运行架构：

```
"development": {
    "connector": [
        { "id": "connector-server-1", "host": "127.0.0.1", "port": 3050 }
    ],
    "chat": [
        { "id": "chat-server-1", "host": "127.0.0.1", "port": 6050 }
    ]
}

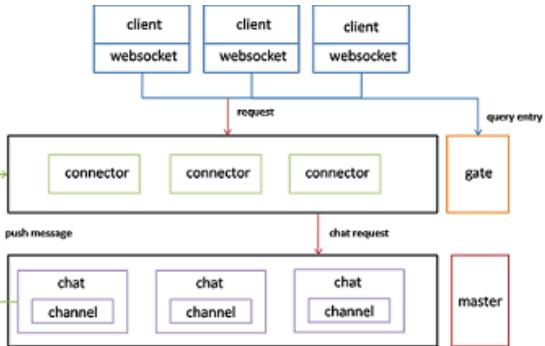
```



- 扩展后的服务器与运行架构：

```

"development": {
    "gate": [
        { "id": "gate-server-1", "host": "127.0.0.1", "port": 3014 }
    ],
    "connector": [
        { "id": "connector-server-1", "host": "127.0.0.1", "port": 3050 },
        { "id": "connector-server-2", "host": "127.0.0.1", "port": 3051 },
        { "id": "connector-server-3", "host": "127.0.0.1", "port": 3052 }
    ],
    "chat": [
        { "id": "chat-server-1", "host": "127.0.0.1", "port": 6050 },
        { "id": "chat-server-2", "host": "127.0.0.1", "port": 6051 },
        { "id": "chat-server-3", "host": "127.0.0.1", "port": 6052 }
    ]
}
    
```



另外，在0.3版本的pomelo中增加了动态增删服务器的功能，开发者可以在不停服务的情况下根据当前应用运行的负载情况添加新的服务器或者停止闲置的服务器，这样可以让服务器资源得到更充分的利用。

总结

本文通过聊天服务器的搭建过程，分析了pomelo开发应用的基本流程，基本架构与相关概念。有了这些知识我们可以轻松地使用pomelo搭建高实时应用了。在后文中我们将分析更复杂的游戏案例，并且会对架构中的一些实现深入剖析。

原文链接 <http://www.infoq.com/cn/articles/game-server-development-2>

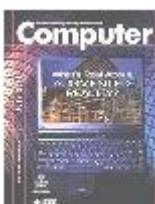
相关资源：

- [pomelo 构建聊天服务器 tutorial](#)
- [聊天服务器的源码](#)
- [线上聊天服务器 demo](#)

推荐文章

预防 Web 应用程序的漏洞

作者 [Nuno Antunes and Marco Vieira](#) 译者 张卫滨



本文最早发表于 [《Computer》](#) 杂志，现在由 InfoQ 以及 IEEE Computer Society 合作为你呈现。

如今的 Web 应用程序可能会包含危险的安全缺陷。这些应用程序的全球化部署使其很容易遭受攻击，这些攻击会发现并恶意探测各种安全漏洞。

Web 环境中两个主要的风险在于：注入——也就是 SQL 注入，它会让黑客更改发往数据库的查询——以及跨站脚本攻击（XSS），它们也是最危险的（[Category:OWASP_Top_Ten_Project](#)）。注入攻击会利用有问题代码的应用程序来插入和执行黑客指定的命令，从而能够访问关键的数据和资源。当应用程序将用户提供的数据不加检验或编码就发送到浏览器上时，会产生 XSS 漏洞。

尽管 2009 年 OWASP（Open Web Application Security Project）的一个报告表明安全方面的投资在增加

（[Category:OWASP_Security_Spending_Benchmarks](#)），但是 NTA Monitor 的 2010 Web 应用安全报名表明 Web 的安全性跟前一年相比实际在下降。实际上，Web 应用的漏洞给公司和组织带来了很多的问题。按照 WhiteHat Security 最新的 [Web 站点安全性](#) 数据报告显示，被评估网站的 63% 是有漏洞的，每个平均有六个未解决的缺陷。（[WhiteHat Website Security Statistics Report](#)）。这些漏洞创建并维持了一个基于攻击窃取数据和资源的地下经济链。

Web 应用程序需要有深度防御的措施来避免和减少安全性漏洞。¹ 这种方式假设所有的安全预防措施都可能失败，所以安全性依赖于多层的机制从而能够覆盖其他层的失败。为了减少成功攻击的可能性，软件工程师团队必须做出必要的努力来引入适当的安全性防护措施。要达到这一点必须使用各种技术和工具来确保安全性涵盖软件产品开发生命周期的所有阶段。

软件开发生命周期中的安全性

尽管软件开发的生命周期有多种不同的划分方式，但正如图 1 所示，它通常包含如下的阶段：初始化、规范和设计、实现（编码）、测试、部署以及停用，这些阶段应用开发人员可以不断地重复迭代。²

尽管开发人员应该在产品的整个生命周期中都关心代码安全性，³但是他们应该特别关注三个关键阶段：¹

- **实现。** 在编码过程中，软件开发人员必须使用特定应用领域内避免关键漏洞的最佳实践。这种实践的例子包括输入和输出校验、识别恶意字符以及使用参数化的命令。⁴ 尽管这些技术在避免大多数安全漏洞方面很有效，但因为缺乏安全相关的知识，开发人员通常并不使用它们或者使用得不正确。边栏“为什么开发人员不使用安全编码实践？”更详细地讨论了这个问题。
- **测试。** 有很多技术可以在测试阶段使用，包括渗透测试（目前最流行的技术）、静态分析、动态分析以及运行时的异常检测。⁴ 问题在于开发人员通常会关注需求功能的测试而忽略安全方面。另外，现有的自动化工具要么在漏洞探测覆盖度方面比较差要么产生太多的误报。
- **部署。** 在运行时环境中，会有不同的攻击探测机制。这些机制可以按照不同的级别运行并使用不同的探测方式。它们的使用障碍在于性能开销以及不准确的结果会打乱系统的正常行为。

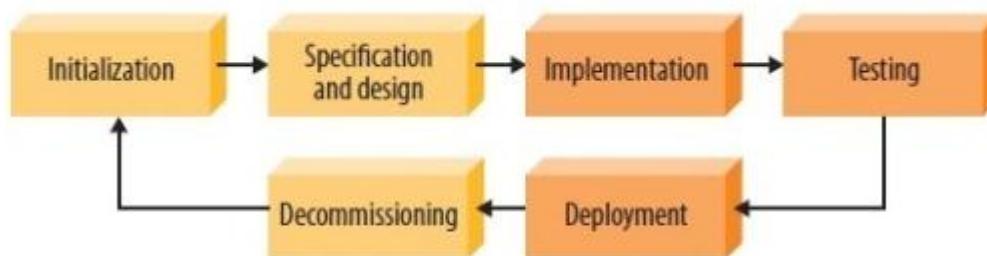


Figure 1. Simplified version of a software development life cycle.

开发安全的代码

为了编写没有漏洞的安全代码，⁴ 基于 Web 基础设施的关键业务开发人员就要遵循编码实践，这个实践包括了深度防御的措施，它假设所有的安全性预防措施都会失败。在实现阶段依赖多层的安全机制是特别重要的，使用一个预防或保护措施来避免安全漏洞是不够的。

Web 应用程序的特征在于需要三层不同的安全防线：输入校验、热点保护以及输出校验。

输入校验

大多数的安全漏洞是因为目标应用程序没有正确地校验输入数据。¹ 所以，应用程序要考虑到所有恶意的输入直到能证明其合法，这要涵盖不可信环境中的所有数据。

输入校验是第一道防线，总体来讲就是缩小应用程序允许输入的范围，它会直接作用在用户提供的数据上。这种类型的防御要依赖输入参数在一个合法的范围内，或者如果用户提供了超出了范围的值就会停止执行。在 Web 应用程序中，这首先要标准化输入将其转换到基线字符集和编码。接下来，应用程序必须对标准化的输入使用过滤策略，拒绝那些值在合法范围之外的输入。这种方式能够避免很多 Web 应用程序中的问题，在执行输入校验时会使用正向模式匹配或正向校验。在这种情况下，开发人员建立规则来识别那些可接受的输入而不是识别有什么输入是不可接受的。尽管开发人员不能预测所有类型的攻击，但他们应该能够说明所有类型的合法输入。

关键问题在于，输入校验通常使用地并不充分，这是因为输入参数的数据域允许存在恶意数据，这是与校验执行相独立的。例如，在 SQL 注入漏洞中，大多数的 SQL 语句使用引号作为字符串分隔符，这就意味着黑客可以使用它来执行 SQL 注入攻击。⁴ 但是，在有些情况下，字符串输入域必须允许存在引号值，所以应用程序不能排除所有包含引号的值。

热点防护

为了应对输入校验的局限性，有必要采用第二道防线

任何类型的攻击都是以热点为目标的，热点指的就是应用程序中可能会有某种类型漏洞的代码。通用的输入校验会在应用程序中进行或者在整个 Web 应用程序上下文中修改输入，与之相比，第二道防线关注于保护重要的热点，例如保护那些真正使用输入域值的代码行。

一个具体的例子就是 SQL 注入攻击，它们大多数会使用单引号或双引号。有些编程语言提供了对这些字符的转码机制，这样它们就能用在 SQL 语句中了，但是只能用来在语句中分隔值。⁴但是这些技术有两个问题。第一，更高级的注入技术，例如联合使用引号和转义字符，可以绕过这些机制。第二，引入转义字符会增加字符串的长度，如果结果字符串的长度超过数据库限制的话，可能会导致数据截断。

正确使用参数化命令是预防注入攻击最有效的方式。¹在这种情况下，开发人员定义命令的结构，并使用占位符来代表命令的变量值。稍后，当应用程序将对应的值关联到命令上时，命令解释器会正确地使用它们而不会涉及到命令的结构。

这种技术最著名的用法是数据库的预处理语句，也被称为参数化查询。⁴当应用程序创建预处理语句时，语句发送到了数据库端。应用程序使用占位符来表示查询的可变部分，占位符通常会是问号或标签。随后，每次查询执行时，应用程序都要往对应的可变部分绑定值。不管数据的内容是什么，应用程序会一直使用这个表达式作为一个值而并没有 SQL 代码。因此，不可能修改查询的结构。

为了确保正确使用数据，很多语言允许类型绑定。但是预处理语句本身并不能修复不安全的语句——开发人员必须正确地使用它们。例如，像传统语句一样使用预处理语句——也就是使用字符串拼接来绑定 SQL 查询——而不是对查询的可变部分使用占位符会导致类似的漏洞。

输出校验

在将一个进程的输出发送之前进行校验能够避免用户收到他们不应该看到的信息，例如应用程序内部的异常细节，这些信息有助于发起其他的攻击。在输出校验的另一个例子当中，保护系统会搜索应用程序输出的关键信息，如信用卡号，并在发送给前端之前用星号代替。将信息编码是能够避免 XSS 漏洞的一种输出校验方式。⁴如果发送给浏览器的数据要显示在 Web 页面上，它应该进行 HTML 编码或百分号编码，这取决于它在页面的位置。通过这种方式，XSS 所用的恶意字符不再具有破坏性，而且编码会保留数据的原来意义。

探测漏洞

识别安全的问题要求不仅测试应用程序的功能还要寻找代码中可能被黑客利用的隐藏的危险缺陷。⁵探测漏洞的两个主要方式是白盒分析和黑盒测试。

白盒分析

白盒分析需要在不执行的情况下检查代码。开发人员可以按照以下两种方式中的某一种来进行：在代码的审查或评审时以手动方式进行或者借助自动分析工具自动化进行。

代码审查（Code inspection）指的是程序员的同伴系统检查交付的代码，查找编码错误。⁶ 安全审查是减少应用程序中漏洞最有效的方式；当为关键的系统开发软件时，这是重要的过程。但是，这种审查方式通常是很费时间的、代价昂贵并需要深入了解 Web 的安全知识。

代码检查（Code review）是代价稍为低廉的替代方案，⁶ 它是一种简化版本的代码审查适用于分析不像前面那么重要的代码。检查也是手动进行的，但是它不需要正式的审查会议。几个专家分别进行检查，然后由主持人过滤和合并结果。尽管这是一个有效的方式，但代码检查的成本依旧是很高的。

为了减少白盒法分析的成本，开发人员有时依赖自动化工具，如静态代码分析器。静态代码分析工具会检查软件代码，要么是源码格式要么是二进制格式，并试图识别出常见的编码级别缺陷。⁴ 使用现有工具所执行的分析会因为它们的复杂性而有所不同，这种差异体现在考虑单条语句和命令或考虑代码行之间的依赖。除了模型检查和数据流分析等功能之外，这些工具还会自动关注可能的编码错误。它的主要问题在于细致的分析是很困难并且因为源码的复杂性和缺乏动态（运行时）的视角有很多安全缺陷很难被发现。

尽管使用静态代码分析工具很重要，但是它有时会降低开发人员的生产效率，这主要是因为误报，这会导致没有用处的额外工作。⁷ 为了避免这种情况，除了要有足够的时间学习怎样使用这些工具以外，开发人员需要一些策略来保证正确使用这些工具。例如，有必要指定规则来分类和选择开发人员应该处理的警告信息。同时，开发人员还要配置分析工具只报告那些与当前开发上下文相关的警告。没有接受怎样使用静态分析训练的开发人员最终会低估它的真正效益并且通常不能发挥它的所有功能。

黑盒测试

黑盒测试指的是从外部的视角分析程序的执行。简而言之，它会比较软件执行的输出与期望的结果。⁵ 对于软件的检验和确认来说，测试可能是最常用的技术了。

对于黑盒测试来讲，有多种级别，从单元测试到集成测试再到系统测试。测试方式可以是正式的（基于模型和定义良好的测试规范）也可以不那么正式（被称为“冒烟测试”，一种粗糙的测试目的是快速暴露简单的缺陷）。

健壮性是黑盒测试一种特殊形式，它的目标是查看系统在错误输入条件下的行为。渗透测试是特殊类型的健壮性测试，它会分析在遇到恶意输入时的代码执行并查找潜在的漏洞。在这种方式中，测试人员使用模糊技术，这包含通过 HTTP 请求，提交意料之外的或非法的数据项到 Web 应用程序上并检查它的响应。⁴ 测试人员不需要了解实现细节——他们在用户的角度来测试应用程序的输入。对于每种漏洞类型，可能会有上百次甚至上千次的测试。

渗透测试工具会自动搜索漏洞，这避免了手工为每种类型的漏洞构建上百个甚至上千个测试所带来的重复和乏味的工作。Web 应用的常见自动化安全测试工具一般会称为 Web 应用或 Web 安全扫描器。这些扫描器可以很容易地测试应用程序以发现漏洞。对于目标应用，它们会有一些预定义的测试用例，所以用户只需要配置一下扫描器并让它测试应用即可。一旦扫描器完成测试，它会报告所探测到的漏洞。大多数的扫描器都是商业产品，尽管也有免费的应用程序扫描器，但是与商用版本相比，它们缺少大多数的功能所以用的很有限。

漏洞检测的局限性

渗透测试和静态代码分析可以是手动的也可以是自动化的。因为手动测试或检查需要特殊的安全资源并且很费时间，所以对于 Web 应用的开发人员来说自动化工具是常见的选择。当考虑漏洞检测工具的局限性时，很重要的一点就是安全测试是很困难的。确实，衡量应用程序的安全性是很有挑战性的：尽管发现一些漏洞可能很容易，但是保证应用没有漏洞是困难的。¹

渗透测试和静态代码分析工具都有其固有的局限性。渗透测试依赖于有效地代码执行，但是在实践中，漏洞识别时只会检查 Web 应用的输出。所以，缺少查看应用的内部行为会限制渗透测试的有效性。

另一方面，详尽的源代码分析可能比较困难。代码的复杂性以及缺少动态（运行时）的观察可能会阻止发现很多安全缺陷。当然，渗透测试不需要查看源码，但是静态代码分析需要。

使用错误的检测工具会导致部署的应用含有未检测出的漏洞。图 2 比较了在 Web 服务中，知名的并广泛使用的渗透测试和静态分析工具在检测 SQL 注入漏

洞中的表现。⁸结果显示静态代码分析工具——包括 FindBugs、Fortify 360 以及 IntelliJ IDEA (在图中匿名为 SA1 到 SA3) ——的覆盖度通常高于渗透测试工具，包括 HP WebInspect、IBM Rational AppScan、Acunetix Web Vulnerability Scanner 以及科英布拉大学开发的一个原型工具 (在图中匿名为 VS1 到 VS4)。这两种方式都有的一个问题就是误报，但是在静态分析中更明显。一个重要的发现在于相同方式下的不同工具对于相同的代码通常报告不同的漏洞。

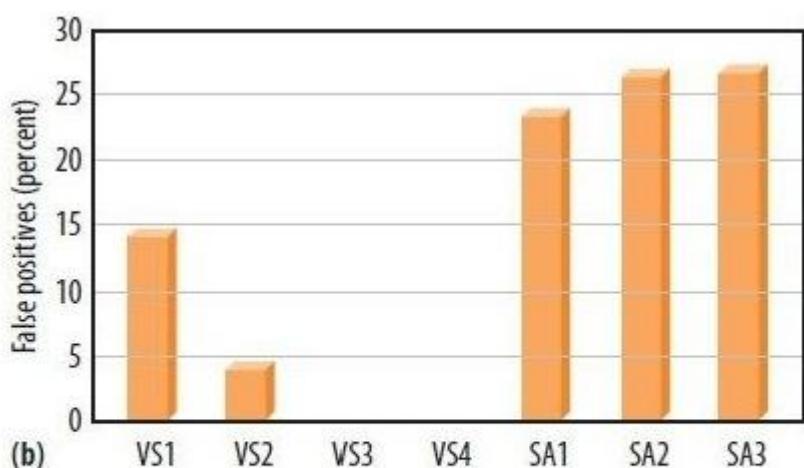
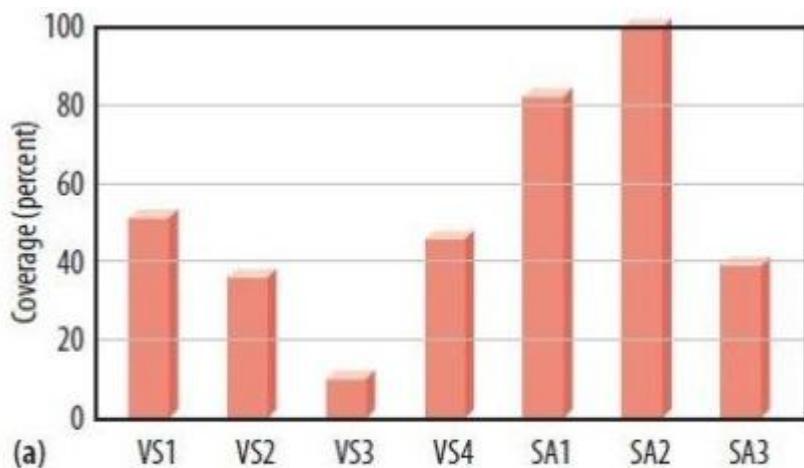


Figure 2. (a) Penetration testing versus (b) static code analysis.

根据研究结果，需要强调工具的局限性使得有必要提高漏洞检测的有效性，比如这可以通过联合使用多种方式来实现。另外，开发人员需要定义一种机制来评估和比较不同的工具，这样它们才能选择最适合各种开发场景的工具。

检测攻击

为了防止对 Web 应用的攻击，软件工程师必须实施攻击检测机制，通常称为入侵检测系统(intrusion detection system ,IDS)或 Web 应用防火墙(WAF)。不同的工具可以作用在应用或网络级别甚至在应用的资源上，如数据库，它们可以使用不同的方式如异常检测或签名匹配来检测攻击。

检测攻击的方法

检测攻击要区分出与所学习行为的差别。攻击检测工具所使用的方式要么基于异常检测要么基于签名。⁸

异常检测通常需要一个训练阶段。训练阶段会展现系统的无恶意请求，工具会在给定的架构级别观察它的行为并学习正常的操作。这些工具会考虑到每个 Web 应用程序的细节，但如果应用程序的正确行为发生了变化或学习不完整的话，会产生很多错误的警告。

相比之下，基于签名的工具会查找预定义的一组规则模式或标示攻击的签名。因为这些签名通常是独立于应用的，所以工具的成功与应用程序的运行配置文件或任何训练过程无关。

在网络级别进行操作的工具通常会监视和分析网络流量，以保证攻击在到达 Web 应用之前检测到。工作在应用级别的攻击检测工具会分析发送给应用的请求并试图利用服务端程序和请求中参数的特定关系。工作在资源层的工具会保护与每种漏洞类型相关的资源。这些工具会在应用层之下并接近受保护的资源。一个常见的例子是监控对数据库服务器的访问来检测 SQL 注入的 IDS。

工具使用各种策略来收集应用请求以及可能收到攻击的信息。一些工具会使用嗅探策略来监控和分析通过网络传输的数据以此来观察 HTTP 流量，但是加密、编码以及封装可能会限制其有效性。同时，网络上可能会承载大量与受保护应用无关的数据。另一些工具会分析应用产生的日志甚至是应用所在服务器所产生的日志。尽管这种策略不会直接延迟对应用的请求，但是受限于日志中可以得到的信息。

而另一种策略就是在请求的来源和受保护的应用或资源间引入一个代理。这能够很容易地阻止攻击，因为它提供了关于目标应用或资源的有用信息，但是，它引入了不良的延时从而会影响应用的正常行为。

检测攻击的局限性

因为每个 Web 应用程序的细节会影响到攻击检测工具的表现，同时工具所运行的架构级别也会有所影响，所以它们的实际效果通常是未知的。¹⁰ 大多数工具的检测覆盖比较低（在很多场景下，低于 20%），同时它们还会有很多误报（高达所产成警报的 50%）。此外，有些工具在特定的场景下展现的结果很好，但是在其他场景下所提供的结果很差。

数据库级别的工具通常比应用级别的工具表现更好一些，¹⁰ 不过它们会产生一些关于请求的误报，这些请求是不会成功攻击数据库的。基于异常检测的工具对于简单的应用表现得更好，而基于签名的工具对于复杂的应用表现更好。在简单的应用程序中，工具能够学习并且更好地描述行为，因此从模式中检测偏差会更准确。实际上，异常检测的成功取决于训练阶段。如果训练不完整或者应用的正常操作配置在训练后发生了变化，那么攻击检测工具的有效性会降低。

使用这些工具的开发人员有时缺乏创建适当配置的培训。这会减少工具的有效性，这凸显了评估和对比不同工具及配置的重要性。¹⁰

新的趋势和方向

要达到更好的结果并提高有效性需要新的技术来克服漏洞检测工具的局限性。但是要克服这些局限性不容易，因为它需要将传统方式改为颠覆性的方法。关键在于释放一些约束并将不同的方法结合起来以克服单个方法的局限性。

[Acunetix AcuSensor](#) 就是一个商业技术的例子，它将黑盒扫描和测试执行反馈结合起来。反馈来自于植入到目标应用程序代码中的传感器(sensor)。Acunetix 声称这种技术能够发现更多的漏洞并能够精确表明漏洞在代码中的位置，而且误报也会更少。

一项最近提出的技术试图以更小的侵入性实现类似的效果，它联合使用攻击签名和接口监控来克服渗透测试对注入攻击漏洞测试的局限性。¹¹ 这是一种黑盒测试技术，因为它只会监控应用程序和漏洞相关资源的接口（如数据库接口）。

[Analysis and Monitoring for Neutralizing SQL-Injection Attacks\(Amnesia \)](#) 工具组合了静态分析和运行时监控来检测 SQL 注入攻击。¹² 它对 Web 应用的源码进行静态分析，构建一个由应用生成的合法查询模型。在运行时，它监控动态生成的查询，检查是否与静态生成的模型相符。这个工具认为违反模型的查询为攻击并阻止它访问数据库。

为了应对 Web 应用安全的新威胁，开发流程必须也要有所发展。例如，微软安全开发生命周期（ Microsoft Security Development Lifecycle ）完善了公司的开发流程并特别针对安全问题的解决，例如明确了开发团队的安全培训。¹³ 按照微软的说法，这个流程的采用减少了软件中的安全缺陷。尽管这只是一个例子，但是它表明在这个行业中，对软件执行安全流程是很重要的事情。

在整个软件产品的开发和部署生命周期中，开发人员必须要考虑安全性。他们必须要使用安全编码的最佳实践、执行足够的安全测试并使用安全检测系统在运行时保护应用程序。在这个任务中，开发人员需要得到一些帮助来获取需要的技术和能够提高生产率的工具。

研究人员应该提出创新的工具，能够在开发过程中方便地使用并满足部署时有效性和生产效率的要求。这个演变的中心是安全测试工具，对于检验和确认应用程序以检查安全漏洞来讲，它们是至关重要的。不过，必须要探索新的假设。一个可以预见的可能性就是开发编译器，使其不仅能强制使用最佳编码实现，还能自动化修改存在的安全漏洞。

参考资料

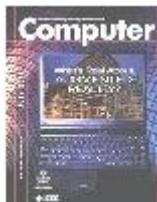
1. M. Howard and D.E. Leblanc, *Writing Secure Code*, Microsoft Press, 2002.
2. C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*, Prentice Hall, 2002.
3. G. McGraw, *Software Security: Building Security In*, Addison-Wesley, 2006.
4. D. Stuttard and M. Pinto, *The Web Application Hacker' s Handbook: Discovering and Exploiting Security Flaws*, John Wiley & Sons, 2007.
5. B. Arkin, S. Stender, and G. McGraw, "Software Penetration Testing," *IEEE Security & Privacy*, Jan.-Feb. 2005, pp. 84-87.
6. D.P. Freedman and G.M. Weinberg, *Handbook of Walkthroughs, Inspections, and Technical Reviews: Evaluating Programs, Projects, and Products*, Dorset House, 2000.
7. N. Ayewah and W. Pugh, "A Report on a Survey and Study of Static Analysis Users," Proc. *Workshop Defects in Large Software Systems* (DEFECTS 08) ACM, 2008, pp. 1-5.

8. N. Antunes and M. Vieira, "Comparing the Effectiveness of Penetration Testing and Static Code Analysis on the Detection of SQL Injection Vulnerabilities in Web Services," *Proc. 15th IEEE Pacific Rim Int' / Symp. Dependable Computing* (PRDC 09), IEEE CS, 2009, pp. 301-306.
9. E. Biermann, E. Cloete, and L.M. Venter, "A Comparison of Intrusion Detection Systems," *Computers & Security*, Dec. 2001, pp. 676-683.
10. I.A. Elia, J. Fonseca, and M. Vieira, "Comparing SQL Injection Detection Tools Using Attack Injection: An Experimental Study," *Proc. 21st IEEE Int' / Symp. Software Reliability Eng.* (ISSRE 10), IEEE CS, 2010, pp. 289-298.
11. N. Antunes and M. Vieira, "Enhancing Penetration Testing with Attack Signatures and Interface Monitoring for the Detection of Injection Vulnerabilities in Web Services," *Proc. IEEE Int' / Conf. Services Computing* (SCC 11), IEEE CS, 2011, pp. 104-111.
12. W.G.J. Halfond and A. Orso, "Preventing SQL Injection Attacks Using AMNESIA," *Proc. 28th Int' / Conf. Software Eng.* (ICSE 06), IEEE CS, 2006, p. 798.
13. M. Howard and S. Lipner, *The Security Development Lifecycle*, Microsoft Press, 2006.

关于作者

Nuno Antunes 是葡萄牙科英布拉大学信息科学与技术系的在读博士生，在这里他获得了信息工程的理科硕士。他的研究兴趣包括开发安全 Web 应用程序和服务的方法论和工具。Antunes 是 IEEE 计算机学会的会员。可以通过 nmsa@dei.uc.pt 联系他。

Marco Vieira 是葡萄牙科英布拉大学信息科学与技术系的助理教授 (assistant professor)。他的研究兴趣包括可靠性和安全基准测试、实验可靠性评估、错误注入、软件开发过程以及软件质量保证。Vieira 在科英布拉大学获取了计算机工程的博士学位。他是 IEEE 计算机学会和 ACM 的会员。可以通过 mvieira@dei.uc.pt 联系他。



*Computer*是IEEE计算机学会的旗舰出版物，出版了很多为同行热议且广受赞誉的文章。这些文章大都由专家执笔撰写，代表了计算机技术软硬件及最新应用的领先研究。它提供较商业杂志更多的技术内容，而较研究学术期刊具有更多的实践性思想。*Computer*传递着可适用于日常工作环境的有用信息。

QClub

我们影响有影响力的人

北京 上海 广州 大连 西安 太原 成都 杭州 武汉 南京 深圳...

QClub

邀请
业内知名专家

自由开放的
讨论氛围

定期举办的线下活动

结识
圈内技术好友

InfoQ



中文 | 英文 | 日文 | 葡文 |

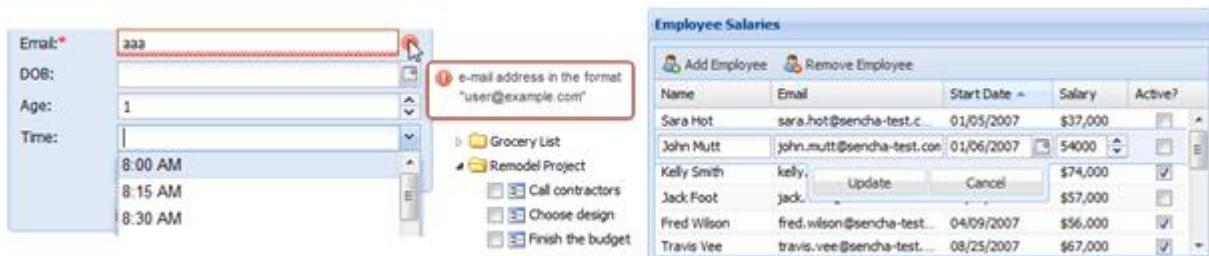
推荐文章

敏捷自动化测试（2）：像用户使用软件一样享受自动化测试

作者 殷坤

在本系列的第一篇文章“[我们的测试为什么不够敏捷](#)”中，根据实例总结出敏捷自动化的两大阻碍：“脚本维护困难”、“断言条件繁琐”。本文针对如何降低脚本维护难度分享一些实践经验。

近几年，Web 技术发展势头迅猛，浏览器市场群雄争霸、各种 UI 组件库也如雨后春笋。现在互联网上已经很少有仅支持一种浏览器，并且不基于任何可复用的 UI 组件库进行开发的应用了。开发人员基于各种优秀的 UI 组件库（如 [JQuery](#)、[Dojo](#)、[ExtJS](#)）可以很容易的开发出功能强大、展现绚丽、兼容各种浏览器的页面（如下图）：



UI 组件库的广泛采用在提高开发效率的同时，也极大的提升了用户体验。基于 UI 组件库之所以能快速开发出功能强大的页面，是因为 UI 组件库可以自动生成海量、结构类似的 HTML 源码（如下图）：

```

<div id="formContent" class="content" style="border:1px solid black; padding:10px; margin-bottom:10px;">
    <div class="form-item">
        <div class="label">姓名:</div>
        <div class="input">张三</div>
    </div>
    <div class="form-item">
        <div class="label">年龄:</div>
        <div class="input">25</div>
    </div>
    <div class="form-item">
        <div class="label">性别:</div>
        <div class="input">男</div>
    </div>
    <div class="form-item">
        <div class="label">爱好:</div>
        <div class="input">篮球</div>
    </div>
    <div class="form-item">
        <div class="label">地址:</div>
        <div class="input">上海市徐汇区淮海中路 1234 号</div>
    </div>
    <div class="form-item">
        <div class="label">电话:</div>
        <div class="input">13812345678</div>
    </div>
    <div class="form-item">
        <div class="label">邮箱:</div>
        <div class="input">zhangsan@example.com</div>
    </div>
    <div class="form-item">
        <div class="label">备注:</div>
        <div class="input">无</div>
    </div>

```

Form 控件


```

<div id="treeContent" class="content" style="border:1px solid black; padding:10px; margin-bottom:10px;">
    <ul style="list-style-type: none; padding-left: 0;">
        <li>根节点
            <ul style="list-style-type: none; padding-left: 20px;">
                <li>子节点 1
                    <ul style="list-style-type: none; padding-left: 20px;">
                        <li>孙节点 1.1</li>
                        <li>孙节点 1.2</li>
                    </ul>
                <li>子节点 2</li>
            </ul>
        <li>子节点 3</li>
    </ul>

```

Tree 控件


```

<div id="gridContent" class="content" style="border:1px solid black; padding:10px; margin-bottom:10px;">
    <table border="1">
        <thead>
            <tr>
                <th>序号
                <th>姓名
                <th>年龄
                <th>性别
                <th>爱好
                <th>地址
                <th>电话
                <th>邮箱
                <th>备注
            
        <tbody>
            <tr>
                <td>1
                <td>张三
                <td>25
                <td>男
                <td>篮球
                <td>上海市徐汇区淮海中路 1234 号
                <td>13812345678
                <td>zhangsan@example.com
                <td>无
            </tr>
            <tr>
                <td>2
                <td>李四
                <td>28
                <td>女
                <td>游泳
                <td>北京市朝阳区建国门大街 123 号
                <td>13912345678
                <td>lisi@example.com
                <td>无
            </tr>
            <tr>
                <td>3
                <td>王五
                <td>30
                <td>男
                <td>足球
                <td>广州市天河区珠江新城 123 号
                <td>13712345678
                <td>wangwu@example.com
                <td>无
            </tr>
        </tbody>
    </table>

```

Grid 控件

开发人员是幸福的，因为这一切对于他来说完全透明。于是只剩下自动化测试人员独自面对这样“恐怖”的页面源码。

如前文“[我们的测试为什么不够敏捷](#)”中所言，业界常见测试工具的脚本本质上还是针对页面源码的，因此原本就举步维艰的自动化测试在开发使用 UI 组件库之后变得雪上加霜：

- 页面 DOM 结构非常复杂

导致所录制/编写脚本的复杂度变的更大、可读性变得更差。

- UI 框架的升级很可能导致 DOM 结构的变化

因此即使开发人员没对代码做任何改动，测试脚本也会因为 UI 框架的升级变得无法回放。

- 控件 ID 是自动生成的，甚至在每次刷新页面后都会变化

大部分自动化测试工具在“录制”脚本时，都会优先使用 ID 定位策略，自动生成的 ID 会导致这种关键的控件定位策略变得无效。

- UI 框架在各种浏览器下自动生成页面源码可能不完全相同

为了在不同浏览器下“看起来一样”，实际的 DOM 结构有时也可能是不同的，因此所录制脚本的浏览器兼容性会比较差。

技术的发展是为了让生活变得越来越轻松。从用户的角度来看确实如此：Web 应用的功能覆盖范围越来越广、操作越来越方便、界面越来越美观。

为什么自动化测试人员没有感觉工作变得轻松呢？

要回答这个问题，首先要分析“用户使用软件”与“自动化测试软件”之间的一些重要差异：

- 用户使用软件时只关注界面上能“看”到的，而不用总是“查看页面源码”；
- 用户会更关注整体业务的正确性、稳定性，而不仅仅是每个孤立页面功能的正确性；
- 用户对页面样式、响应时间、浏览器兼容性要求越来越高；如果我们能像用户使用软件一样进行自动化测试，测试就会变得更敏捷；
- 根据界面快速编写测试脚本：敏捷应对需求的变化；
- 降低对技术实现（UI 框架、页面样式/布局）的依赖：敏捷应对设计/开发的变化；
- 测试脚本可以稳定支持各种浏览器：敏捷应对环境的变化；

(一) 根据界面快速编写测试脚本的实现思路

还是以前文“[我们的测试为什么不够敏捷](#)”中用到的“用户增加”界面为例：



如果你是作为用户在使用上述功能时，心里一定也会默念下面内容吧：

“账号”输入***、 “密码” 输入***、 “姓名” 输入***、 “性别” 选择***、 “生日” 输入***、 “国籍” 选择***，点击 “保存” 按钮。

这就可以理解为 “根据界面编写的测试脚本” 。

我们在使用 Web 应用时 心里常常默念的还包括：“展开/收拢/选中” 树(Tree) 的某个节点、数据表格 (Grid) 的下一页/上一页、选中数据表格 (Grid) 的某一行、点击/关闭某个 Tab 页，等等。

很多人在与笔者交流的过程中都会问 “为什么还要手工编写脚本，怎么不提供脚本录制工具呢？”

因此在这里想澄清一下大家对 “编写” 脚本的误解，与传统自动化测试工具相比， “此脚本” 非 “彼脚本” 。

传统的测试脚本是给特定测试工具执行时使用的，对人类而言可读性极差，更别谈手工编写了，因此 “录制/回放” 工具往往都是必备组件。即便如此，此类 “录制/回放” 工具的实际使用效果，也是不尽如人意的。

从这种角度来看，“录制” 脚本是 “下策” ， “上策” 应该是让测试脚本大幅简化，并且具备良好的可读性和可维护性，以达到人工编写很容易的程度。

——以上只是笔者的一点差异化见解，对业界优秀的测试工具没有任何冒犯之意！

(二) 降低测试脚本对技术实现依赖度的实现思路

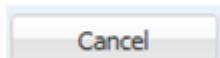
Web 页面开发中的技术实现细节主要包括 UI 框架的选择及版本升级、页面样式设计、页面布局设计，因此我们的主要目标就是降低测试脚本对这些因素的依赖。

为了便于测试脚本的解析和组织管理，目前还不能直接写形如 “点击(保存)按钮” 这样的纯自然语言，但可以设计成接近自然语言的领域专用语言 (DSL:Domain Specific Language)，本文采用 XML 来实现这种 DSL。比如：

```
<event id="[button]保存" name="click"/>
```

这种超级清晰、简短的测试脚本与实际海量的页面源码之间有一条鸿沟，我们可以通过“封装”来屏蔽。下面以 [ExtJS](#) 的 Button 控件为例来示意如何完成这种封装：

- Button 的界面展现如下：



- 实际生成的页面源码 DOM 结构如下（省略部分非关键属性）：

```
<div id="button-1032" class="x-btn x-box-item x-btn-default-small
    x-noicon x-btn-noicon >
    <em id="button-1032-btnWrap">
        <button id="button-1032-btnEl" class="x-btn-center"
            autocomplete="off" role="button">
            <span id="button-1032-btnInnerEl">Cancel
            <span id="button-1032-btnIconEl" class="x-btn-icon" />
        </button>
    </em>
</div>
```

我们封装所要做的主要工作就是解析出测试脚本中定义的关键信息（button、保存、click），结合对应页面源码的 DOM 结构，翻译成 W3C 标准的定位方式（如，XPath）。通过 XPath 就可以定位到页面上的任何控件。

对于测试脚本（`<event id="[button]Cancel" name="click"/>`），翻译后的 XPath 可以是（`//button/span[text()='Cancel']`）。

同理，其它测试脚本还可以包括：

- 树节点展开/关闭

```
<event id="[treeNode]部门" name="open"/>
```

```
<event id="[treeNode]部门" name="close"/>
```

- Grid 翻页

```
<event id="[grid]人员列表" name="nextPage"/>
```

它们的封装实现比 Button 稍微复杂一些，但原理是一样的。

(三) 增强测试脚本浏览器兼容性的实现思路

这个就比较简单了，只要选用标准化程度高、厂商支持度高、扩展性强的第三方底层实现即可，笔者推荐采用 [Selenium WebDriver](#)。

通过上面的介绍，有些读者或许会有个疑问：“如果一个 Web 应用没有采用任何 UI 框架、而且编码又极为不规范，那么你这种方案岂不就不适用了？”

答案是肯定的。但笔者认为此类 Web 应用如果想要发展下去，其瓶颈还停留在开发环节，对其进行自动化回归测试的投入产出比不是很大。

此外，为了进一步提高 Web 应用的可测试性，笔者在实际工作中总结了一些前台页面开发的最佳实践，供大家参考：

- 页面采用单帧实现

如果页面上的 frame/iframe 嵌套很多的话，控件的定位会比较麻烦，并且影响展现速度。

- 兼容 Firefox

Firefox 有很多强大插件，如 Firebug、FirePath，非常有助于在开发、测试过程中的调试问题。

- 采用统一的 UI 框架开发复杂控件

对于复杂控件，比如 Tree、Grid，如果不基于统一的 UI 框架实现，开发、测试工作量都会很大。

- 对于需要设置 ID 的控件，一定要设置唯一、并且有业务意义的 ID

当然对于一般不需要设置 ID 的控件（如，div）或者控件 ID 由 UI 框架自动生成的情况除外。

- 对于 Form 中最常见的 label+input 组合，尽量让 label 和 input 有逻辑对应关系

推荐为 label 设置 for 属性，属性值等于 input 的 id，这样对最终用户也比较友好：双击 label 的时候，鼠标焦点能定位到对应的 input 中。

- 页面设计时考虑用户体验，往往用户使用方便的时候，自动化测试也会方便

比如，一个页面上不要有多个同名的按钮、通过点击上下微调按钮（箭头）改变输入域值的控件也支持直接输入值、日期选择控件支持用户直接输入、下拉列表控件支持输入过滤，等等

对以上几点最佳实践有如下补充说明：

1. 这些最佳实践不仅能提高 Web UI 的可测试性，也非常有助于提升用户体验；
2. 这些最佳实践如果能满足更好，即使不能全部满足，也可以开展自动化测试；

按照本文给出的方案，前文“[我们的测试为什么不够敏捷](#)”中用到的“用户管理（增加、删除）”功能的自动化测试脚本就可以改造为如下样式（示意代码）：

1. <event id="[button]新增" name="click"/>
2. <event id="[input]帐号" name="setValue" value="\${account}"/>
3. <event id="[input]密码" name="setValue" value="1"/>
4. <event id="[input]姓名" name="setValue" value="\${accountName}"/>
5. <event id="[input]性别" name="select" value="女"/>
6. <event id="[input]生日" name="setDate" value="1980-01-01"/>
7. <event id="[input]国籍" name="select" value="中国"/>
8. <event id="[button]保存" name="click"/>
9. <event id="[gridRow]\${account}" name="assertExist"/>
10. <event id="[gridRow]\${account}" name="check"/>
11. <event id="[button]删除" name="click"/>
12. <event id="[gridRow]\${account}" name="assertNotExist"/>

以上测试脚本完全基于界面上“可见”的内容进行编写，大家应该不需要看下面的解读，也能明白脚本的意思：

- 第1、8、11行表示点击“新增”、“保存”、“删除”按钮；
- 第2~7行表示为输入域赋值，赋值方式有输入文本、输入日期、选择下拉选项；
- 第10行表示选中表格中的指定行，即，选中指定行前面的Radio按钮；
- 第9、12行表示断言表格中指定的行“存在”或“不存在”；
- \${account}表示使用外部的变量account；

自动化测试中最关键的两部分是“脚本”和“断言”。至此，自动化测试脚本的编写、维护以及执行已经可以跟上敏捷开发的步伐了。本系列的下一篇文章会就“如何让断言不再成为自动化测试的负担”这个话题来分享一些实践经验，敬请关注！

作者简介

殷坤，东软集团资深测试经理、技术讲师，10年软件研发、实施、测试及项目管理工作经验。目前专注于敏捷项目管理及质量控制、过程改善、自动化测试、持续集成、用户体验提升等方面。

原文链接：<http://www.infoq.com/cn/articles/Agile-test-automation-2>

相关内容：

- [社区热议自动化测试发展前景](#)
- [Robot Framework 作者建议如何选择自动化测试框架](#)
- [Android 自动化测试解决方案](#)

架构师

www.infoq.com/cn/architect

每月8号出版

时刻关注软件开发领域的变化与创新

架构师

11月 ARCHITECT

特别专题
光棍节狂欢的背后——
电商系统深探
1号店B2C电商系统深造之路
百万点推荐引擎——从需求到架构
麦当劳购物系统浅谈分享
REST的远程API设计案例
大型Rails与VoIP系统架构
与部署实践
什么是Node.js
扩展Oozie
浅谈dojox中的一些小工具

InfoQ 每月8号出版

时刻关注企业软件开发领域的变化与创新

架构师

10月 ARCHITECT

特别专题
大数据时代
大数据
大数据时代的数据管理
阿里巴巴数据架构设计经验与挑战
大数据时代的创新者们
关系数据库还是NoSQL数据库
向Java开发者介绍Scala
HTML 5 or Silverlight?
解析JDK 7的Garbage-First收集器
了解云计算的基础

Steve Jobs
1955-2011

InfoQ 每月8号出版

时刻关注企业软件开发领域的变化与创新

架构师

9月 ARCHITECT

特别专题
QCon全球企业大会精华点滴
QCon在中国的三年回顾
麦肯锡对阿里巴巴国际站架构演进
畅销书《IPS》和《技术流年》
敏捷团队建设的虚与实
沐泽宁谈主观决策架构
跟着李树学Oozie
如何查看我的订单—
REST的远程API设计案例
通用系统思考，走上改善之路
Redis内存使用优化与存储

InfoQ 每月8号出版

时刻关注企业软件开发领域的变化与创新

架构师

8月 ARCHITECT

特别专题
云计算的安全风险
圆桌会议：云计算的安全风险
设计一种云级别的身份认证结构
云应用和平台的现状：
云采纳者如是说...

Java虚拟机家族考
专家视角看IT转型
为什么使用 Redis及高产品定位
架构演化之谜

InfoQ 每月8号出版

时刻关注企业软件开发领域的变化与创新

架构师

7月 ARCHITECT

特别专题
深入理解Node.js
为什么要用后端工程语言Node.js
虚拟机设计：Node.js生态系统之
秘密，操作实践
使用Java连接JavaScript并行编程
Node.js的起源和实践应用—
专访Node.js创始人Ryan Dahl

Java深度剖析十：Java对集群划分与热切换
将数据打散之一：关于横向的数据设计
分布式平台的服务PaaS
社区驱动的开源计划
来自Padmanabha的真实声音

InfoQ 每月8号出版

新品推荐 | Product

Yeoman：适合现代 Web 应用的现代工作流

作者 Tom Lane 译者 孙镜涛

前端开发者应当关注一下 Yeoman 的发布，这是一个前端开发工作流。

原文链接：<http://www.infoq.com/cn/news/2013/01/yeoman>

亚马逊推出 Simple Workflow Service 手册

作者 Boris Lublinsky 译者 薄海

来自亚马逊的一份最新的白皮书列出了一些常用的编程模式，这些模式可以被用在应用的决策逻辑当中，用于告知 SWF 应如何协调该应用的工作。

原文链接：<http://www.infoq.com/cn/news/2013/01/swfrecipes>

Node.js 包含新的流 API

作者 James Campos 译者 孙镜涛

Node.js 是一个基于 Google V8 引擎构建的服务器端 JavaScript 平台，在 0.9.4 这个不稳定版本中包含了一个新的流 API。新 API 称为 “streams2”，计划在 0.10 稳定版中正式发布。

原文链接：

<http://www.infoq.com/cn/news/2013/01/new-streaming-api-node>

用于展现图表的 50 种 JavaScript 库

作者 侯伯薇

在很多项目中都会有在前端展现数据图表的需求，而在开发过程中，开发者往往会使用一些 JavaScript 库，从而更有效地达到想要的目标。最近，TechSlide 上的一篇文章总结了 50 种用于展现图表的 JavaScript 库，并对每种库做了简要的说明。这对于想要选择合适 JavaScript 库的开发者很有参考意义。

原文链接：

<http://www.infoq.com/cn/news/2013/01/50-javascript-chart-lib>

来自荷兰格罗宁根大学的架构决策捕捉工具 RGT

作者 Michael Stal 译者 薄海

格罗宁根大学的 Dan Tofan 向软件架构师提供了开源软件工具 RGT (Repertory Grid Tool)，这种工具用于捕获和评估他们的架构决策。这个工具可以帮助架构师更好的文档化他们的决策及对决策进行回顾。

原文链接：<http://www.infoq.com/cn/news/2013/01/rug-rgt-tool>

新版《Scrum 启动规划书》旨在帮助敏捷团队步入正轨

作者 Craig Smith 译者 康锦龙

新版《Scrum 启动规划书》刚刚由 Adam Weisbart 发布，目的是促使团队围绕创建敏捷团队或项目的重点展开讨论。

原文链接：

<http://www.infoq.com/cn/news/2013/01/scrum-kickoff-planner>

Vert.x 计划加入 Eclipse 基金会

作者 Alex Blewitt 译者 李彬

在 VMWare 确定不会阻止后，项目创始人 Tim Fox 选择将 Vert.x 落户于 Eclipse 基金会。

原文链接：<http://www.infoq.com/cn/news/2013/01/vertx-eclipse>

使用 Visual Studio 3D 初学者工具包(Starter Kit)进行 Windows 应用商店游戏开发

作者 Anand Narayanaswamy 译者 蔡坚安

Visual Studio 3D 初学者工具包是游戏开发示例的集合，指导开发者如何在 Visual Studio 2012 项目的帮助下为 Windows 应用商店开发基础游戏。

原文链接：<http://www.infoq.com/cn/news/2013/01/vs-3d-starter-kit>

最新的技术雷达趋势

作者 Aslan Brooke 译者 张龙

ThoughtWorks 最新的“技术雷达”聚焦在移动、可访问性分析、简单架构、可再生环境与数据持久化。

原文链接：

<http://www.infoq.com/cn/news/2013/01/thoughtworks-radar-1012>

沃尔玛实验室开源项目一览

作者 郑柯

众所周知，沃尔玛是世界第一大零售商，但少为人知的是，沃尔玛有一个实验室，该实验室在开源项目上有不少贡献。这些项目中，大部分都与 Node.js 和 JavaScript 有关。

原文链接：

<http://www.infoq.com/cn/news/2013/01/walmart-lab-opensource-projects>

推荐编辑 | 方腾飞

在写作中成长



记得第一次向 InfoQ 投稿已经是 1 年前的事情了，当时花了几星期的时间写了一篇文章“深入分析 volatile 的实现原理”，准备在自己的博客中发表时，在同事金建法的建议下，怀着试一试的心态投向了 InfoQ，庆幸的是半小时后得到 InfoQ 编辑郑柯采纳的回复，高兴之情无以言表。这也是我第一次在专业媒体上发表文章，而后在 InfoQ 编辑张龙的不断鼓励和支持下，我陆续在 InfoQ 发表了几篇并发编程相关的文章，于是便形成了“聊聊并发”专栏，在这个专栏的写作过程中，我得到了非常多的成长和帮助，在此非常感谢 InfoQ 的编辑们。

接下来我想分享下我的写作历程和我对写作的看法。

通过写作来学习。不知何时便爱上了写作，翻开自己的博客，从 06 年到现在已经写了 164 篇博文。为什么会写博客？因为我的学习方法是每个输入都尽量有所输出，无论是读书，看文章，参加技术讲座，还是看电影，我都会或多或少的总结输出出来，输出形式要么是一篇博客，要么是一篇微博，要么是一篇笔记。我总是在想如果这篇文章我看完了，但是我却说不出他讲的是什么，那我岂不是在浪费时间。自己完全掌握的知识应该是自己能表达出来的知识。

在写作中成长。但是对于我博客中的大多数博文不能称之为文章而是笔记。所以从 2011 年起我开始认真对待写作这件事情，尝试着花一个星期，甚至一个月的时间来写一篇文章，在这个过程中为了把一些东西写清楚，我翻阅了大量的外国文献，阅读了 java 虚拟机的源码，并做了一些试验来验证自己的理论是否正确，虽然为写一篇文章花的时间非常长，但是收获颇丰，对于读者能通过阅读这样的文章能够深入了解某项技术，对于自己也能获得对这项技术的深入理解。

写作能帮我真正理解知识。其实写作的过程就是一个逐渐总结知识过程，我们学过的东西很快便会忘记，那是因为这些知识没有经过大脑的推理总结来转化为自己大脑容易存储的知识，当你在下笔时，你的大脑会不停的反问自己，这样写对吗？是不是还需要再读一遍再理解一下？只是看一遍，却没有总结，停留在自己

脑海的知识是很难成体系的，或者经不起推敲的，所以把它们写下来，然后进行重组，会更有收获。没有真正理解这个知识，在运用的时候就很难举一反三，形成有效的解决方案。

写作能帮我把知识记得更深。 大脑是一块很大的硬盘，存储的信息非常多，但是想访问到全部的知识却非常慢，而通过写作来总结知识相当于在为大脑里存储的数据建立索引，当需要用到某项知识时，能快速的通过索引把知识检索出来。而索引就是通过一句话或几句话来对知识进行高浓缩度的总结，比如本文中每一个段落的第一句话，只要能在大脑里检索出这句话，相信这句话后面的详细内容也更容易拉出来。而这种索引也非常容易被检索出来，因为大脑喜欢存储这种有层次感的数据。

如何写作？ 对于技术人员来说，写作的确不是很擅长的事情，那么如何把知识总结得很有层次感呢？在此给大家推荐一本书《[金字塔原理](#)》，一种思考、表达和解决问题的方法，我目前的写作方式就是尽量遵循金字塔原理的。

分享完写作之后，我想介绍下我现在正在做的事情。目前我的业余时间除了在 InfoQ 和程序员发表文章之外，还在维护一个[并发编程网](http://ifeve.com) (<http://ifeve.com>)，该网站致力于推动并发编程在中国的研究和推广，网站的内容除了在 InfoQ 发表过的文章外，还包括很多并发编程相关的译文，比如并发大师 Doug Lea 写的并发文献的译文和 Disruptor 官方资料的译文。

最后，感谢大家的阅读，希望大家能喜欢。

封面植物

水仙（学名：Narcissus tazetta var. chinensis），属石蒜科水仙属植物，原产



中国，在中国已有一千多年栽培历史，为中国传统名花之一，分布于东亚以及中国大陆的浙江、福建沿海岛屿等地。多年生草本，地下部分的鳞茎肥大似洋葱，卵形至广卵状球形，外皮棕褐色皮膜；阔线形扁平叶子，先端钝，二列状着生；冬季抽花穗，近顶端有膜质苞片，苞开后放出花数朵，苞花萼中空，扁筒状，通常每球有花萼数支，多者可达10余支，每萼数支，至10余朵，组成伞房花序，白色芳香花，内部有黄色杯状突起物（副冠）。

水仙花主要有两个品种：一是单瓣，花冠色青白，花萼黄色，中间有金色的冠，形如盏状，花味清香，所以叫“玉台金盏”，花期约半个月；另一种是重瓣，花瓣十余片卷成一簇，花冠下端轻黄而上端淡白，没有明显的付冠，名为“百叶水仙”或称“玉玲珑”，花期约二十天左右。水仙性喜温暖、湿润，又要排水良好。

以疏松肥沃、土层深厚的冲积沙壤土为最宜， $\text{pH} 5 \sim \text{pH} 7.5$ 均宜生长。

水仙具有宜人的芳香，鲜花为制造高级芳香油的原料，是水仙雕刻艺术的主要材料，而且是很受欢迎的年花，因为其在春节期间开放，象征来年好运。

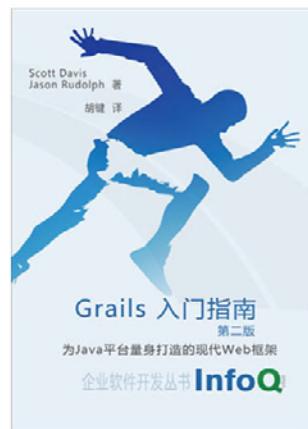
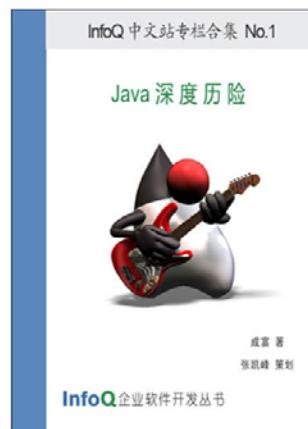
1kg.org 多背一公斤

爱自然 | 更爱孩子



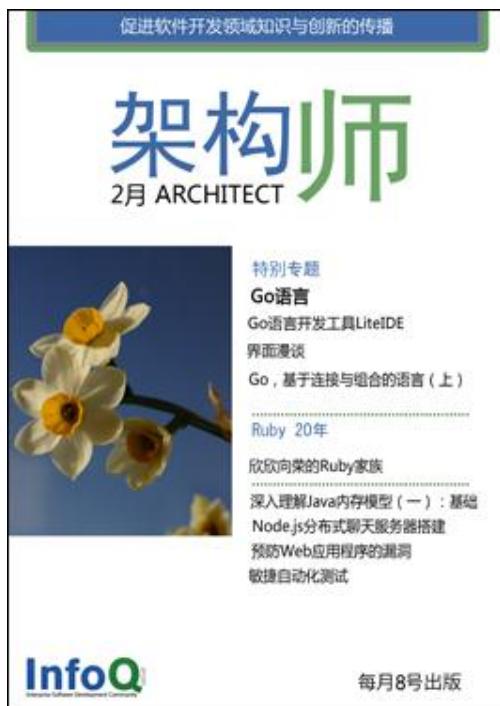
InfoQ 软件开发丛书

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



架构师 2 月刊

每月 8 日出版

本期主编：侯伯薇

责任编辑：杨赛

美术/流程编辑：水羽哲

总编辑：贾国清

发行人：霍泰稳

读者反馈：editors@cn.infoq.com

投稿：editors@cn.infoq.com

InfoQ 中文站新浪微博：

<http://weibo.com/infoqchina>

商务合作：sales@cn.infoq.com 15810407783



本期主编：侯伯薇，InfoQ 中文站译者团队主编

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