



2012/10/22

码农

codeMaker ()

第 2 期

- 本期专题：JavaScript
- 七大框架论剑
- Nicholas C. Zakas 和他的
《JavaScript高级程序设计》
- 陈皓：我的精神家园
- 老赵：以“玩”之名
- 迄今为止最伟大的极客
- 深入Markdown

目录

卷首语

- 1 改变世界的好码农

专题：JavaScript

- 3 JavaScript 宝座：七大框架论剑
- 19 JavaScript 模块化开发一瞥
- 27 对话 Nicholas C. Zakas
——《JavaScript 高级程序设计》作者
- 34 通过对象图学习 JavaScript

人物

- 41 我的精神家园——陈皓（@ 左耳朵耗子）专访
- 54 以“玩”之名——赵劼（@ 老赵）专访

践行

- 70 通才还是专才——由摩托裁员引发的讨论

八卦

- 73 为什么 Nikola Tesla 是迄今为止最伟大的极客

鲜阅

- 88 善言的 JavaScript
——选自《善言的 JavaScript: 现代化编程指南》

出版的未来

- 96 Markdown 作者谈“深入 Markdown”

书榜

- 106 看看大家都在看什么？

妙评

- 109 《Erlang/OTP 并发编程实战》

成书手记

- 115 从失败中学习
——《30 天自制操作系统》译者自述

社区动态

- 119 iTran 乐译 10 期
120 专家审读 5 期
121 十月，跟着图灵听课去！

改变世界的好码农



杨帆

文科女，但热情总放在业余爱好上，结果把爱好做成了职业，然后矛盾啊。在 IT 出版圈浸染多年，干过的活还挺多，也算某种手艺人了吧。现任图灵社区执行主编。
图灵社区 ID: [杨帆](#)

图灵出版的多是技术书，惯例是找业内的熟手来翻译（或写作），无论译者还是作者，都是 IT 圈里颇具实力的行家，常与他们联系，再加上经营社区这一年，我算是半只脚踏进了 IT 圈，得以一窥这个时代最大的手艺人——码农们。

与我而言，这是工作的转换，更是人生的惊喜。因为一直以来，我坚信的“一个人可以改变世界”在这里真的常常能看到，而“做个好人也能赢得成功”的结局也时常上演。这个行业有太多正面的效应，能身处其中感受到它们，真是件幸运的事，尤其是在今日之中国。

程序员的工作是解决技术难题，写代码形成产品，而结果如何，写完就能看出好坏，也因此，彼此间的较量更在于专业的比拼，所以聪明、优秀的程序员可以不用考虑其他，秉持自己的良心，坚持正直、诚实的态度，做个可爱的人。本期人物专访的两位受访对象，就亲自向我们展示了这样的人生。

高科技 = 可放大性，这是 Paul Graham 在《黑客与画家》这本书中说的，“一次开发，普遍适用”的模式让程序员的工作可以迅速地扩大影响，更难能可贵的是，软件行业的目标本来就是全世界，在这里，一个人也能有改变世界的潜能。单以出版业来

说，Markdown 的发明者应该不会料到，他设计的这种互联网写作语法会成为技术书电子出版的一个选择，国外的 Leanpub 结合 Markdown 与 Dropbox 推出自出版平台，而我们的社区也选择它作为写作文本的基础语法，现在电子书上线，它更毫不意外地成为电子书的制作工具，并由此改变了我们传统的出版业务流程。所以，诸位自嘲为“码农”的同志，你们耕耘的可是块好地，我们期待看到更多改变世界的榜样。

本期《码农》的主题是 JavaScript，现在有很多基于 JavaScript 改进的语言，也涌现了众多精心设计的开源框架和库。这样自然形成的繁荣局面，让人忍不住去期待这门语言未来的变化。不过，编程语言的用户最终还是程序员，如果诸位有什么想法，不妨来社区谈一谈。■

JavaScript 宝座：七大框架论剑

一周前，[Throne of JS 大会](http://throneofjs.com)在多伦多召开，这应该是我参加过的最有料也最不一样的一次大会。大会官网如是说：

“加载整个页面，然后再‘渐进增强’以添加动态行为，这种构建 Web 应用的方式已经不够好了。要想让应用加载快，反应灵敏，而且又引领潮流，必须彻底检讨你的开发手段。”



没错，应该是 8 个框架，不是 7 个。但到底怎么回事儿，会议主办方也没有明确给我们解释过……

这次大会邀请了七大 JavaScript 框架 / 库的创建人，他们济济一堂，面对面交流各自的技术理念。所谓七大框架 / 库分别是：AngularJS、Backbone、Batman、CanJS、Ember、Meteor、Knockout、Spine。



作者 / Steven Sanderson
微软 .NET 技术专家，现在
微软公司负责 .NET、IIS 等
相关工作开发。他也是流行
的 JavaScript 库 Knockout
的核心开发人员，*Pro ASP.
NET MVC 3 Framework* 一
书作者。个人网站：[http://
blog.stevensanderson.
com/](http://blog.stevensanderson.com/)。

声明：我在会上讲 Knockout，因此我的观点显然不是中立的。在这篇文章中，我重点讨论这些创建人的思路和技术理念，尽量不提我赞成或反对什么。

文章可长啦，先概述一下：

- 对许多 Web 开发人员来说，要构建富 Web 应用，使用客户端框架是理所当然的。如果你什么框架也没用，那要么你不是在做应用，要么就会错过很多好东西。
- 在使用方法上，这些框架很多地方都是一致的（模型 - 视图 - 某某架构、声明绑定，等等——详见下文），因此从某种意义上讲，无论你选择哪一个，都能得到同样的好处。
- 理念上还是有不少差异，特别是在对框架和库的看法上，分歧格外大。你的选择会深刻影响你的架构。
- 会议本身活泼，新颖，技术小组之间有很多交流和对话。我希望能有更多类似的会议。

技术：共识与分歧

随着每个 SPA（Single Page Application，单页应用）技术的逐一展示，一些相当明显的相似性和差异性浮出了水面。

共识：渐进增强不能建立真正的应用

各技术门派一致认为，真正的 JavaScript 应用必须有适当的数据

“不用马拉的车”(horseless carriage) 是汽车刚刚发明的时候，人们对它的称呼。——译者注

模型，并具备客户端渲染能力，而绝不仅仅是服务器处理数据再加上一些 Ajax 和 jQuery 代码那么简单。

用 Backbone 创建人 Jeremy Ashkenas 的话说：“现如今，你说‘单页应用’，都跟说‘不用马拉的车’差不多了。”（意思是，早已经没那么新鲜了。）

共识：模型 - 视图 - 某某

所有技术门派都坚持模型 - 视图分离。有的强调 MVC (Model View Control)，有的提到 MVVM (Model View ViewModel)，甚至有人拒绝明确说出第三个词儿（只提模型、视图，然后加上让它们协调运作的东西）。对各门派而言，最终结果其实是相似的。

共识：推崇数据绑定

除了 Backbone 和 Spine 之外，其他框架都在自己的视图里内置了声明数据绑定的机制（Backbone 的设计理念强调让用户“自选视图技术”）。

共识：IE6 已死

在小组讨论中，大多数框架的创建者说，他们对 IE 浏览器的支持只限于 7+（事实上，Ember 和 AngularJS 的起点是 IE8，Batman 需要 ES5 “垫片”才能在 IE9 之前的 IE 版本中使用）。这也是大势所趋：[jQuery 2 已经不打算支持 IE9 以下的旧版本 IE 了](#)。

只有 Backbone 和 Knockout 还坚定支持 IE6+（我不清楚 Backbone

的内部实现，但 Knockout 会把 IE6/7 那些令人抓狂的渲染及事件方面的怪异行为屏蔽掉）。

共识：许可和源代码控制

大家都使用 MIT 许可，并且托管在 GitHub 上。

分歧：库，还是框架

这是目前最大的分歧。下表对 JavaScript 库和框架进行了归类：

括号中的数字是最近某个时间点 GitHub 上的关注者数量，粗略地代表各自的影响力。

JavaScript 库	JavaScript 框架
Backbone (9552)	Backbone (9552)
Knockout (2357)	AngularJS (2925)
Spine (2017)	Batman (958)
CanJS (321)	Meteor (4172) ——了不起，见下文

什么意思呢？

- **JavaScript 库**，插到既有架构中，补充特定功能。
- **JavaScript 框架**，提供一个架构（文件结构啊，等等），你必须遵守它，只要你遵守，那剩下的就全都是处理通用需求了。

目前来看，鼓吹框架模型最卖力气的是 Ember，其创建人 Yehuda Katz 之前是（理念相似的）Rails 和 SproutCore 项目的开发者。他的观点是，缺少任何组件都不够给力，都不能说是真正在推动技术进步。相反的观点说，库的目的更明确，因而更容易掌握、采用、定制，也有助于把项目风险降到最低，毕竟你的架构不会严重依赖任何一个外部项目。根据我参加对话的情况看，现场观众也分成了两派，有支持框架的，也有支持库的。

请注意，AngularJS 可以说是介于库和框架之间的一种形态：它不要求开发时遵守特定的文件组织方式（与库类似），但在运行时，它提供一个“应用生命周期”，可以对号入座地把代码安排进去（与框架类似）。之所以把它归入框架之列，是因为 AngularJS 团队乐于接受这个说法。

分歧：灵活，还是整合

每个技术门派都有不同程度的强制性规定：

	视图	URL 路由	数据存储
AngularJS	内置基于 DOM 的模板（必选）	内置（可选）	内置系统（可选）
Backbone	自选（最常用的是基于字符串的模板库 handlebars.js）	内置（可选）	内置（可重写）
Batman	内置基于 DOM 的模板（必选）	内置（必选）	内置系统（必选）

CanJS	内置基于字符串的模板（必选）	内置（可选）	内置（可选）
Ember	内置基于字符串的模板（必选）	内置（必选）	内置（可重写）
Knockout	内置基于 DOM 的模板（可选，也可以用基于字符串的模板）	自选（大都使用 sammy.js 或 history.js）	自选（如 knockout.mapping 或只用 \$.ajax）
Meteor	内置基于字符串的模板（必选）	内置（必选？不确定）	内置（Mongo，必选）
Spine	自选基于字符串的模板	内置（可选）	内置（可选？不确定）

不难想见，只要某个库在某方面是开放的，他们的人就会强调只有这样才能从总体上确保跟第三方库兼容。同样，显而易见的反对意见则是，只有内置才能保证无缝整合。再次，根据我参加的对话，现场观众也各持己见，说什么的都有，基本上可以看出每个人对其他技术组合的了解程度。

替代译法

[@ 时 蝇 喜 箭](#) 我们用了很多“法术”，但都是好的“法术”，意味着可以分解成合理的基本组件。

[@ 连城 404](#) 我们的代码技巧性比较高，但绝非旁门左道，

Ember 的 Tom Dale 说：“我们加入了很多魔法，但都是不错的魔法，换句话说，它们完全可以分解为常规的操作。”

分歧：基于字符串的模板与基于 DOM 的模板

（请参考上面的表格。）对基于字符串的模板，大家几乎都选择 Handlebars.js 作为模板引擎，它俨然成了这个领域的霸主，当然

都是些由常规的基本语义元素构成的东西。

@ 玉伯也叫射雕 我们实现了很多巧妙的整合，真的非常巧妙，这些整合都可以分解成普通操作。

CanJS 用的是 EJS。对基于字符串的模板，支持的人认为“它更快”（不一定），而且“理论上，服务器也可以处理它”（也不一定，因为前提必须是在服务器上运行所有模型代码，而实践中根本没人那么做）。

而基于 DOM 的模板呢，意味着纯粹通过在实际标记中绑定来控制流程（each、if，等等），且不依赖任何外部模板库。支持的声音有“它更快”（不一定），另外“代码易读、易写，且标记与模板之间没有隔阂，CSS 如何与之交互也一目了然”。

在我看来，最有吸引力的说法来自 AngularJS 那帮家伙，他们认为在不久的将来，基于 DOM 的模板会得到浏览器原生支持。所以我们最好现在就用，从而可以轻松应对未来。AngularJS 来自 Google，所以他们在开发 Chromium 时会考虑这一点，而且也会说服标准主体接纳这个建议。

分歧：服务器中立到什么程度

Batman 和 Meteor 明显依赖服务器：Batman 是为 Rails 设计的，而 Meteor 本身就是服务器。其他大多数都追求服务器中立。但实际上，Ember 的架构、强制性规定，以及某些工具都倾向于 Rails 开发者。当然，Ember 绝对也能跟其他服务器技术搭配，只不过眼下还需要较多手工配置。

技术门派概览

以下是所有 JavaScript 库 / 框架的基本技术细节。

Backbone

Who: Jeremy Ashkenas 和 DocumentCloud。

What:

- 用 JavaScript 实现模型 - 视图，MIT 许可。
- 只有一个文件，1000 行代码，在所有库中最小！
- 功能极其专一，只提供 REST 可持久模型及简单路由和回调（以便你知道何时渲染视图，但视图渲染机制由你自己选择）。
- 名气最大，很多大牌站点都在用（也许是因为它最小，容易部署）。

Why:

- 非常小，使用它之前，你完全可以通读并理解它的源代码。
- 不会影响你的服务器架构或文件组织方式。可以在页面的某一部分内运行——不需要控制整个页面。
- Jeremy 好像进入了一种禅宗所谓的入定的状态，对一切都能坦然接受。他就像一个大人，看着一群孩子在那里辩论。

Where: [GitHub](#) 及 [自有站点](#)。

When: 至今已诞生近两年了。

Meteor

Who: [Meteor 开发团队](#)（他们[刚募集到 1120 万美元投资](#)，因此可以全职开发）。

What:

- 前瞻性极强的一个框架，想不出有谁那么激进过（也许[Derby](#) 算一个）。
- 将一个服务器端运行时环境（用 Node+Mongo 搭建）和一个客户端运行时环境衔接起来，让你的代码在两端都能运行，还包含数据库。利用 WebSockets 实现所有客户端和服务器之间的同步。
- 在修改代码时就“实时部署”——客户端运行时可以即时更新而不丢失状态。
- 可以[看看这个视频](#)，对它的认识就会更全面。
- 跟会上与我有过交流的所有人一样，我也衷心希望这个框架获得成功——Web 开发就需要这种激进的改革才能真正进步。

Why: 你实在觉得做常规 Web 开发太无聊了，想找点刺激。

Where: [GitHub](#) 和 [自有站点](#)。

When: 诞生时间不长；除了其核心团队在用，不知道还有没有其他站点实际在用 **Meteor**。不过，这个团队真是在严肃地做着一件前无古人的事。

Ember

Who: Yehuda Katz （之前开发过 jQuery 和 Rails）、Ember 团队和 Yehuda 的公司 [Tilde](#)。

What:

- 构建“超级 Web 应用”所需的一切，MIT 许可。
- 功能最多，体积最大。
- 融入了很多设计理念，涉及如何分解并对页面进行层次控制，以及如何利用一个状态机驱动的系统联结各个层次。
- 正在开发一个功能非常完善的数据访问库（Ember.Data）。
- 要在运行时控制整个页面，因此不适合开发大页面上的“富应用区”。
- 对文件、URL 等都有相当严格的一套约束，不过要是不喜欢，你可以重写，只要你知道怎么做就 OK。
- 设计灵感来自 Rails 和 Cocoa。
- 工具：为 Rails 提供项目模板（但如果你手工编写代码，也

可以使用其他服务器端平台）。

Why: 常见的问题应该有通用的解决方案——Ember 提供了所有通用解决方案。

Where: [GitHub](#) 和 [自有站点](#)。

When: 尚未发布 1.0 版，但也快了。然后，API 基本就能稳定下来。

AngularJS

Who: Google（他们内部在使用）。

What:

- 用 JavaScript 实现模型 - 视图 - 其他，MIT 许可。
- 基于 DOM 的模板，具备可观察能力、声明绑定机制，还有准 MVVM 式的代码风格（他们自己说是 Model-View-Whatever）。
- 内置基本 URL 路由和数据持久化能力。
- 工具：附带一个 Chrome 调试器插件，让你在调试的时候能够查看模型；还附带一个 Jasmine 测试框架。

Why:

- 从概念上讲，他们说这个框架相当于一个“填料层”，盖在当前浏览器上，以实现未来的浏览器将可能原生具备的能力

（即声明绑定和可观察能力）。因此，我们现在就应该着手这么来写代码了。

- 对服务器架构或文件组织方式没有影响。可以用在页面的某一小部分中——不需要控制整个页面。

Where: [GitHub](#) 和 [自有站点](#)。

When: 成品级框架，Google 已经搞出来有一段时间了。

Knockout

Who: Knockout 团队和社区（核心团队目前有三个人，包括我）。

What:

- 用 JavaScript 实现模型 - 视图 - 视图模型（MVVM，Model-View-ViewModel），MIT 许可。
- 功能集中在富用户界面元素：基于 DOM 的声明绑定模板，可观察的模型加自动依赖检测。
- 没有限定 URL 路由或数据访问——可组合任意第三方库（例如，用 Sammy.js 做路由，用纯 Ajax 实现存储）。
- 在降低使用门槛方面下了很大工夫，提供详尽的文档和[交互式示例](#)。

Why:

- 只做好一件事（UI），向后兼容到 IE6。

- 对服务器架构或文件组织方式没有影响。可以用在页面的某一小部分中——不需要控制整个页面。

Where: [GitHub](#) 和 [自有站点](#)。

When: 到现在已经正式发布近两年了。

Spine

Who: Alex MacCaw。

What:

- 用 JavaScript 实现 MVC，MIT 许可证。
- 由最早为 O'Reilly 一本书写的示例代码发展而来，已成为一个 OSS（Open Source Software，开源软件）项目。
- 是 Backbone 的一个衍生版（看名字就知道）。

Backbone 和 Spine 都是“脊椎”的意思。

——译者注

Why: 你喜欢 Backbone，但又想要点[不一样的东西](#)。

Where: [GitHub](#) 和 [自有站点](#)。

When: v1.0.0 已经发布。

Batman

Who: [Shopify](#)（一家电子商务平台公司）的团队。

What:

- 在 JavaScript 中实现 MVC，几乎是专门为 Rails+CoffeeScript 开发者定制的，MIT 许可。
- 是所有框架中强制性规定最多的。你必须遵守其约定（例如，怎么组织文件和 URL）。否则，就像他们幻灯片中说的，“你还是用其他框架吧”。
- 非常完善的框架，具有相当丰富的模型、视图和控制器，还有路由。当然，还有可观察机制。
- 基于 DOM 的模板。

Why: 如果你使用 Rails 和 CoffeeScript，你找到亲人了。

Where: [GitHub](#) 和 [自有站点](#)。

When: 当前版本 0.9，几个月内将发布 1.0 版。

CanJS

Who: [Bitovi](#)（一家 JavaScript 咨询 / 培训公司）的团队。

What:

- 用 JavaScript 实现 MVC，MIT 许可。
- REST 可持久模型、基本的路由、基于字符串的模板。

- 知名度不高（我也是上周才听说它的），但它的前身却是原来的 [JavaScriptMVC 项目](#)。

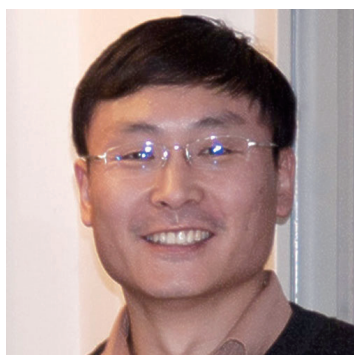
Why: 旨在集上述各技术门派之所长，提供与它们类似的功能，同时又保持体积小巧。

Where: [GitHub](#) 和 [自有站点](#)。

When: 1.0 版已经发布了。

总结

如果你正在考虑选型的问题，想知道上面这些框架 / 库中的哪一个最适合你的新项目，那我建议你重点关注以下两点。



译者 / 李松峰

@ 李松峰，非计算机专业出身的技术爱好者，曾从事 5 年 Web 前后端开发工作，现为北京图灵文化发展有限公司 QA 部主任。2006 年开始涉足计算机相关图书翻译，至今翻译字数超过 458 万字，已出版译著 20 余部。

2007 年建立个人知识分享网站“为之漫笔”

(cn-cuckoo.com)，工作之余除了翻译书，还翻译了大量国外经典技术文章，如被广为转载和引用的“HTML5 设计原理”等。2012 年下半年着手建立“A List Apart 中文版”网站(alistapart.cn)，致力于向中文读者译介这一国际知名的顶级 Web 设计与开发杂志。

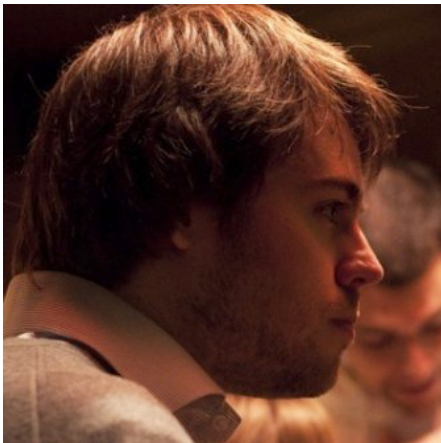
图灵社区 ID: [李松峰](#)

- 功能范围。你想让这个框架或库为你做多少事儿？你的项目是从头做起，因而需要一个能贯穿始终的、完整的、各项功能齐备的架构吗？或者，你其实更喜欢自己来挑选模式和库？对不同的项目，不同的团队，任何选择都有价值，都是正确的。
- 设计美学。你看过它们的代码吗，用没用过自己选择的框架构建出一些小巧的应用？你喜欢这样做吗？不要只看它们的说明或者功能列表就作出选择：那些信息有价值，但不全面。打个比方，如果你置自己主观的编码经验于不顾，那就像在选择小说时只看它有几章几节，或者在找对象时只看其简历或个人描述。

尽管存在分歧，但我认为所有技术门派都有一个重大的共性：它们都践行了模型与视图分离的思想。而这个思想早在 **Web** 诞生之前就已存在，到现在差不多有 **20** 年历史了。这么说吧，就算你只做一个基本的 **Web** 应用的 **UI**，在客户端应用这一思想也永远是正确的。■

查看原文: [Rich JavaScript Applications – the Seven Frameworks](#)

JavaScript 模块化开发一瞥



作者 / David Padbury

他在位于纽约的 [Lab49](#) 公司从事为金融行业创建高级应用程序的工作。他把大部分时间花在开发复杂的 HTML5 及 JavaScript 前端系统上。他的个人博客：<http://blog.davidpadbury.com/>

对于那些正在构建大型应用程序，而对 JavaScript 不甚了解的开发者而言，他们最初必须要面对的挑战之一就是如何着手组织代码。起初只要在 `<script>` 标记之间嵌入几百行代码就能跑起来，不过很快代码就会变得一塌糊涂。而问题是，JavaScript 没有为组织代码提供任何明显帮助。从字面上看，C# 有 `using`，Java 有 `import`——而 JavaScript 一无所有。这迫使 JavaScript 编写者试验不同的约定，并使用现有的语言创建了一些切实可行的方法来组织大型 JavaScript 应用程序。

各种模式（pattern）、工具（tool）及惯例（practice）会形成现代 JavaScript 的基础，它们必将来自于语言本身实现之外。

——Rebecca Murphy

模块模式（The Module Pattern）

用于解决组织代码问题、使用最为广泛的方法之一是模块模式（Module Pattern）。我尝试在下面解释一个基本示例，并讨论其若干特性。要想阅读更精彩的说明，并了解

用尽各种不同方法的怪人，那么请参阅 Ben Cherry 的帖子——[JavaScript Module Pattern: In-Depth](#)（[深入理解 JavaScript 模块模式](#)）。

```
(function(lab49) {  
  
    function privateAdder(n1, n2) {  
        return n1 + n2;  
    }  
  
    lab49.add = function(n1, n2) {  
        return privateAdder(n1, n2);  
    };  
  
})(window.lab49 = window.lab49 || {});
```

在上例中，我们使用了一些来自语言的基本功能，从而创造出在 C# 及 Java 等语言中见过的类似结构。

隔离（Isolation）

请注意，这段代码包在被立即调用的函数里（仔细看最后一行）。由于在浏览器中，默认情况下会把 JavaScript 文件置于全局作用域级别上进行计算（evaluated），因此在我们的文件中声明的任何内容都是随处可用的。想象一下，要是先在 lib1.js 中声明了 `var name = '...'`，然后又在 lib2.js 声明了 `var name = '...'`。那么后一句 var 声明就会替掉前一句的值——这可不太妙。然而，由于 JavaScript 拥有函数作用域级别，上例中所声明的一切都位于函数自身作用域内，与全局作用域毫无瓜葛。这意味着，无论系统将来如何变化，位于函数中的任何内容都不会受到影响。

命名空间（Namespacing）

在最后一行代码中会看到，我们要么将 `window.lab49` 赋给其自身，要么将空对象 `{}` 赋给它。尽管看起来有点儿怪，不过让我们一起来看看下这样一个虚构系统，系统中的那些 js 文件一律使用了上例中的函数包装器（function wrapper）。

首个被引入的文件会计算那个或语句（`...||...`），并发现左侧的表达式 `undefined`（未定义）。由于 `undefined` 会被判定为假，因此或语句会进一步计算右侧表达式，在本例中就是空对象。或语句实际上是个表达式，它会返回计算结果，进而将结果赋给全局变量 `window.lab49`。

现在轮到接下来的文件使用此模式了，它会执行或语句，并发现 `window.lab49` 目前已是对象实例——真（对象实例会被判定为真）。此时或语句会走捷径，并返回这个会立即赋给其自身的值——其实什么都没做。

由此导致的结果是，首个被引入的文件会创建 `lab49` 命名空间（就是个 JavaScript 对象），而且所有使用这种结构的后续文件都只是重用此现有实例。

私有状态（Private State）

正如刚才所说，由于位于函数内部，在其内部声明的所有内容都处于该函数的作用域内，而非全局作用域。这对于隔离代码真是棒极了，不过它还带来了一种效果，那就是没人能调用它。真是中看不中用啊！

刚刚还谈到，创建 `window.lab49` 对象是为了用命名空间来有效地管理我们的内容。而且由于变量 `lab49` 被附加到 `window` 对象上，因此它是全局可用的。为了把其中的内容公布给模块外部，或许有人会公开声称，我们要做的全部就是把一些值附加到那个全局变量上。正如上例中所写的 `add` 函数一样。现在，在模块外部就可以通过 `lab49.add(2, 2)` 来调用 `add` 函数了。

在此函数中声明一些值的另一结果是，要是某个值没有通过将其附加到全局命名空间或者此模块外部的某个对象上的方式来显示公开，那么外部代码就访问不到该值。实际上，我们恰好创建了一些私有值。

CommonJS 模块 (CommonJS Modules)

CommonJS 是个社团，主要由服务器端 JavaScript 运行库 (server-side JavaScript runtimes) 编写者组成，他们致力于将模块的公开及访问标准化的工作。值得注意的是，他们提议的模块系统并非标准，因为它不是出自制定 JavaScript 标准的同一社团，所以它更像是服务器端 JavaScript 运行库编写者彼此之间的非正式约定。

我通常会支持 CommonJS 的想法，但要搞清楚的是：它并不是一份崇高而神圣的规范（就像 ES5 一样），它只不过是一些人在邮件列表中所讨论的想法。而且多数想法都未付诸实现。

——[Ryan Dahl](#)，node.js 的创造者

这份[模块规范](#)的核心相当直截了当。所有模块都要在其自身的上下文中进行计算，并且要有个全局变量 `exports` 供模块使用。而

全局变量 `exports` 只是个普通的 JavaScript 对象，甚至可以自行往上面附加内容，它与上面展示的命名空间对象（[lab49](#)）类似。要想访问某个模块，需调用全局函数 `require`，并指明所请求的包标识符。接着会计算此模块，而且无论返回何值都会将其附加到 `exports` 上。然后会缓存此模块，以便后来的 `require` 函数调用。

```
// calculator.js
// 计算器模块--译注
exports.add = function(n1, n2) {

};

// app.js
// 某个需要调用计算器模块的应用程序。
// './calculator' 即包标识符。--译注
var calculator = require('./calculator');

calculator.add(2, 2);
```

要是摆弄过 Node.js，或许会对以上代码有种似曾相识的感觉。这种用 Node 来实现 CommonJS 模块的方式真是出奇地简单，在 [node-inspector](#)（一款 Node 调试器）中查看模块时，会显示其包装在函数内部的内容，这些内容正是传递给 `exports` 及 `require` 的值。非常类似于上面展示的手卷模块内容。

有几个 node 项目（[Stitch](#) 及 [Browserify](#)），它们将 CommonJS 模块带进了浏览器。服务器端组件会把这些彼此独立的模块 js 文件打包到单独的 js 文件中，并把那些代码用生成的模块包装器包起来。

CommonJS 主要是为服务器端 JavaScript 运行库设计的，而且由于以下几个属性使得它们难以在浏览器中组织客户端代码。

- `require` 必须立即返回——要是已经拥有所有内容时这会工作得很好，不过这导致难以使用脚本加载器（script loader）去异步下载脚本。
- 每个模块占一个文件——为了合并为 CommonJS 模块，必须把它们以某种风格组织起来，并包裹到一个函数中。要是没有类似于上面所提及的服务器组件，那么就难以使用它们，并且在许多环境（ASP.NET，Java）下这些服务器组件尚不存在。

异步模块定义（Asynchronous Module Definition）

异步模块定义（Asynchronous Module Definition，通常称为 AMD）已被设计为适合于浏览器的模块格式。它起初源于 CommonJS 社团的提案，不过自从迁移到 [GitHub](#) 上以后，现已加入了配套的 [测试套件](#)，以便模块系统编写者来验证其代码是否符合 AMD 的 API。

AMD 的核心是 `define` 函数。调用 `define` 函数最常见的方式是传入三个参数——模块名（也就是说不再与文件名绑定）、该模块依赖的模块标识符数组，以及将会返回该模块定义的工厂函数。（调用 `define` 函数的其他方式请参阅 [AMD wiki](#)。）

```
// 定义 calculator（计算器）模块。--译注
define('calculator', ['adder'], function(adder) {
    // 返回具有 add 方法的匿名对象。--译注
    return {
        add: function(n1, n2) {
```

```
译注      /*
           * 实际调用的是 adder（加法器）模块的 add 方法。
           * 而且 adder 模块已在前一参数 ['adder'] 中指明了。--
           */
           return adder.add(n1, n2);
       }
   };
});
```

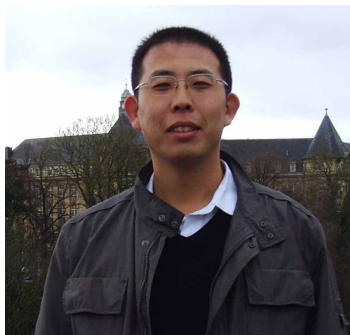
由于此模块的定义包在 `define` 函数的调用中，因此这意味着可以欣然将多个模块都放在单个 `js` 文件中。此外，由于当调用模块工厂函数 `define` 时，模块加载器已拥有控制权，因此它可以自行安排时间去解决（模块间的）依赖关系——对于那些需要先异步下载模块，真可谓得心应手。

为了与原先的 `CommonJS` 模块提案保持兼容，已做出了巨大的努力。有些特殊行为是为了能在模块工厂函数中使用 `require` 及 `exports`，这意味着，那些传统的 `CommonJS` 模块可直接拿来用。

看起来 `AMD` 正在成为颇受欢迎的组织客户端 `JavaScript` 应用程序的方式。无论是如 [RequireJS](#) 或 [curl.js](#) 等模块资源加载器，还是像 [Dojo](#) 等最近已支持 `AMD` 的 `JavaScript` 应用程序，情况都是如此。

这是否意味着 `JavaScript` 很烂？

缺乏语言级别的结构，而无法将代码组织到模块中，这可能会让来自其他语言的开发者觉得很不爽。然而，正因为此缺陷才迫使 `JavaScript` 开发者想出他们自己的模块组织模式，而且我们已经能够随着 `JavaScript` 应用程序的发展进行迭代并改进。欲深入了



译者 / 高翌翔

从 2005 年至今一直从事基于 ASP.NET 的 Web 应用程序开发。长期关注各种设计、开发、测试方法及最佳实践。InfoQ 中文站翻译编辑团队一员。

图灵社区 ID: [高翌翔](#)

解此主题请访问 [Tagneto](#) 的博客。

想象一下，即便在 10 年前就已将此类功能引入语言，那么他们也不可能想到后来的那些需求，例如在服务器上运行大型 JavaScript 应用程序、在浏览器中异步加载资源，或者像 [text templates](#)（文本模板，就是些文本加载器，其功能类似于 RequireJS）那样引入资源等等。

正在考虑将模块（Modules）作为 [Harmony/ECMAScript 6](#) 的语言级别功能。这多亏了模块系统编写者们的奇思妙想以及过去数年中所做的辛勤工作。更有可能的是，我们最终将得到适合于构建现代 JavaScript 应用程序的语言。■

查看原文: [JavaScript Modules](#)

对话 Nicholas C. Zakas

——《JavaScript 高级程序设计》作者



尼古拉斯·泽卡斯
(Nicholas C. Zakas)

前端咨询师、畅销书作者、技术布道者，世界顶级 Web 技术专家，曾在雅虎工作近 5 年，离开该公司前负责 My Yahoo! 和雅虎首页等大访问量站点设计，担任界面呈现架构师。拥有丰富的 Web 开发和界面设计经验，曾经参与许多

图灵社区：对各种 JavaScript 库的优点和缺点一直存在很多争论。在您看来，开发人员应该怎么选择合适的库？

Zakas: 选择 JavaScript 库的时候，有几个非常重要的方面要考虑。首先，就是时间问题，也就是你的项目多长时间必须做完？如果时间很短，那最好选择你最熟悉的 JavaScript 库。如果时间不那么紧迫，那你可以研究一下别的库。此时，必须要回答几个问题。还有谁在使用这个 JavaScript 库？这个库有人在不断完善吗？这个库的文档是否完整详细？这个库的背后有没有一个社区，假如你遇到了问题，能不能获得该社区的支持？是否容易找到具有使用这个库经验的工程师？这个库能解决你的所有问题，还是只能解决其中一部分？所有这些都是在选择某个 JavaScript 库时需要考虑的重要因素。

图灵社区：你对 HTML5 的未来怎么看？现在的宣传是不是太过分了？

Zakas: HTML5 对 Web 发展是一件好事儿。但不好的一点是，一些外行的非技术人员，不分青红皂白地给很多无关的东西都扣上

世界级大公司的 Web 解决方案开发，是《JavaScript 高级程序设计》、《高性能 JavaScript》作者。个人网站：www.nczonline.net，Twitter: @slicknet。

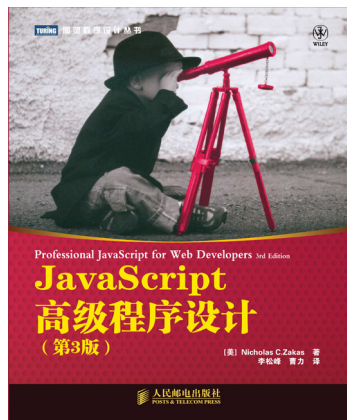
“HTML5”的帽子，这才搞得 HTML5 这个概念满天飞。这与几年前“Ajax”的情况非常相似。从某种角度看，确实宣传上有点过了，仿佛 HTML5 会彻底改变每个人的生活。事实并非如此。HTML5 的意义在于为开发人员提供了更多的工具，利用这些工具能够创建更有吸引力的用户体验。

图灵社区：看来，Mobile Web（移动互联网）开发会成为下一行业焦点，你觉得呢？

Zakas：移动互联网开发已经是焦点了。今天，谁不关注移动用户，谁就要被时代抛弃。移动互联网可不是昙花一现，它将是一个时代。如果你真是在开发 Web 应用，那么就必须考虑移动体验，否则就会让别人抢占先机。

图灵社区：现在有很多基于 JavaScript 改进的语言，比如 Dart、CoffeeScript，等等。你认为 JavaScript 今后的路会朝着哪方面发展呢？是更加类似于 JVM 这种的中间层，还是仍然维持一个强大灵活的编程语言存在？或者说，对于专注于 JavaScript 的前端工程师来说，是否应该投入大精力去研究和使用的 CoffeeScript 这种语言来简化工作，而不是纠结于 JavaScript 本身可能具有的繁复解决方案呢？

Zakas：我没觉得 JavaScript 有一天只会被当成一个中间层。Dart 和 CoffeeScript 很引人关注，这说明开发人员可能更希望 JavaScript 能多适应一些应用场景。最终，我想 JavaScript 会博采众长，吸纳其他语言中更流行的范式，从而使语言核心更完善。但我不认为将来的 Web 开发人员会只用 Dart 或 CoffeeScript 或者其他能编译为 JavaScript 的语言写代码。



作为 JavaScript 技术经典名著，本书承继了之前版本全面深入、贴近实战的特点，在详细讲解了 JavaScript 语言的核心之后，条分缕析地为读者展示了现有规范及实现为开发 Web 应用提供的各种支持和特性。

图灵社区：你觉得 Node.js 怎么样？它会在服务器端开发中发挥重要作用吗？将来，Web 前、后端开发真能只用一种语言来做吗？

Zakas：我认为 Node.js 对未来 Web 应用的重要性难以估量。开发人员一直在寻找一个可以替代 PHP 的方案，以便更迅速、更容易地介入服务器端开发。而在服务器上写 JavaScript 代码就是一种方案。Node.js 不只是一个服务器端的 JavaScript 引擎，它更为高性能、高扩缩的 Web 应用提供了一个解决方案。正因为如此，很多 JavaScript 爱好者可能会转型为后端开发工程师。这样一来，前、后端的沟通会更加顺畅、直接，无论是面对面沟通，还是通过代码交流。

图灵社区：请问用 JavaScript 实现一些实用算法——比如压缩 ZIP 格式，是否可行？

Zakas：不仅可行，而且已经有人做到了！斯图尔特·奈特利（Stuart Knightley）就创建了一个叫 [JSZip](#) 的项目，让我们能够用 JavaScript 来压缩文件。我认为未来还会出现很多类似实用算法的 JavaScript 实现。在实现某些复杂的算法时，可以不使用 JavaScript，但这种可能性是永远存在的。

图灵社区：是否有必要强调 JavaScript 编码风格的一致性？在构建一个大型 B/S 系统时，如何以最佳方式划分 HTML、CSS 和 JavaScript 人员的职责？

Zakas：我觉得任何语言都需要强调编码风格的一致性。只要是团队开发，每个人都以相同方式编写代码就是至关重要的。这样大家才能方便地互相看懂和维护对方的代码。在一个团队中，HTML、CSS 和 JavaScript 的编码风格都应该保持一致。这也是

我为什么要写《可维护 JavaScript》（*Maintainable JavaScript*）这本新书的原因，这本书里就解释了作为团队一分子，应该怎么写 JavaScript。不过，同样的原则也适用于任何语言。

图灵社区：异步、回调编程方式正被广泛使用，但很容易出现复杂的回调函数。虽然有 `deferred` 和 `promise` 这些模式，但对开发人员还是不小的挑战，请问未来是否有可能在语言层面有所改观？

Zakas：经常有人提出建议，希望添加一些语言特性来简化异步编程工作。JavaScript 语言层面到底会不会增加这种特性，现在还说不好，只能拭目以待。目前，有很多人认为回调和异步编程值得提倡，但我不那么认为。如果几乎所有方法都需要一个回调，那会导致代码很难调试和维护。我确实希望在不久的将来，JavaScript 能在语言层面给出一些解决方案。

图灵社区：今天，你涉足 Web 开发已超过 15 年，你当初怎么会选择这个行业呢？能否给中国的开发人员一些职业规划方面的建议（有朝一日也能成为像你一样的专家）？

Zakas：我进入这一行，纯属误打误撞。上大学的时候，我的专业是计算机科学，但 Pascal 和 C 语言这些课让我感觉很无聊。我讨厌整天坐在黑底白字的电脑屏幕前。当时，我想跟高中同学保持联系，有人告诉我有一种新技术，说是叫 Web。于是，1996 年我在 AOL 上建立了自己的第一个网页，然后把网址发给同学，以便他们知道我的近况。我还想知道大家希望通过这个网页了解点其他什么情况，结果维护这个网页就成了我的业余工作。我不断研究、尝试，在此期间自学了 JavaScript。结果大学一毕业，我就知道自己得在互联网行业谋份差事了。

我对 Web 开发人员最大的建议就是：热爱你的工作。热爱跨浏览器开发带来的挑战、热爱互联网技术的种种异端，热爱业内的同行，热爱你的工具。互联网发展太快了，如果你不热爱它的话，就不可能跟上它的步伐。这意味着你必须多阅读，多动手，保证自己的才能与日俱增。下了班也不能闲着，要做一些对自己有用的事儿。可以参与一些开源软件的开发，读读好书，看看牛人的博客。经常参加一些会议，看看别人都在干什么。要想让自己快速成长，有很多事儿可以去做，而且付出一定会有回报。

图灵社区：迄今为止，你已经写了 4 本 JavaScript 书。你怎么会想起来写这些书呢，眼下还有没有写书的计划？

Zakas：我从来没有真正有过写书计划。我第一份工作只干了 8 个月就下岗了（因为公司散伙了）。这就是我当时的处境，大学毕业才 8 个月，而且又失了业。我感觉真正的学习才刚刚开始。于是我就给自己找事儿做，也就是把以前做过的事儿都写出来。写一篇，就在我的网站 (<http://nczonline.net>) 上贴一篇。然后告诉以前的同事都来看。一个朋友回信说：“嘿，你写得非常好，为什么不给杂志投稿呢？”于是，我就上网找到几个征稿的在线杂志。第一篇文章发表在 DevX，后来又有几篇投给了 WebReference。我的文章很受好评，因此我也非常乐意接着写下去。后来有一天，我看了 WebReference 的一篇文章，说有另一位作者，她把自己写过的文章集合起来出了一本书。我心想：“噢，把文章集合在一块就可以出一本书？这事儿我也能干呐！”于是，我就给自己定了一个目标，坚持写，写本书出来。这就是后来的《JavaScript 高级程序设计》（*Professional JavaScript for Web Developers*）。

写另外三本书则隔了很久。《Ajax 高级程序设计》（*Professional*

Ajax) 实际上是 Wrox 的编辑策划的一个选题, 因为我出版过一本书, 所以他就找到了我。一开始我拒绝了, 因为觉得可写的东西还不够多。但他坚持让我写, 我最终还是应承下来。我很高兴又写了这本书, 因为它后来也非常受欢迎。《高性能 JavaScript》(*High Performance JavaScript*) 是通过雅虎出版的, 当时是雅虎内部人员联系我写的。显然, 另外一些人也希望写点相关的东西放在书里, 但当时大家都没有时间动笔。所以, 我就在他们已有成果的基础上做些修改, 改到自己觉得舒服为止。后来又找了几位合著者, 共同完成了这本书。

《可维护 JavaScript》(*Maintainable JavaScript*) 是我最近才出版的一本书, 内容源于 6 年前的一次演讲。我一直都认为这个主题很值得写, 写成一本书都没问题, 但就是不知道从何写起。去年 12 月的一个周六, 我一觉醒来, 突然觉得才思泉涌, 满脑子都是写这本书的想法。我干脆起床, 在电脑前坐了一整天, 终于把脑子里的想法都落实成了文字。那一天, 我写了大约 45 页。12 月份剩下的时间我都花在了写书上, 结果不到一个月就写完了这本书。

目前, 我还没有再写书的计划。我准备先搁笔一段时间, 因为最近我一直都在写, 都连续写了一年多了。确实有几个主题值得写书, 但在此之前, 我得等待灵感爆发的那一刻。

图灵社区: 能否谈谈你的公司 Nicholas C. Zakas Consulting? 很多中国程序员也有创业的梦想, 能分享一下你的经验吗?

Zakas: 我的咨询公司实际上只有我一个人。我依靠自己的经验为互联网公司提供前端技术建议, 包括性能评估、架构设计与评审、推行最佳实践等专业 Web 开发公司可能需要的各方面帮助。这为

记者 /@ 李松峰

我积累了宝贵的经验，让我得以接触各式各样的公司，结识他们的团队，了解他们正在做哪些激动人心的事情。

对于想自己开公司的人，我可以给出的最好建议，就是要有自知之明。你必须得知道一些事儿，比如自己开公司没有工资，有时候一连几个月可能都没有收入。自己开公司要应对很多风险，如果你后面有家人支持，可能风险会小得多。我创业的时机很好，因为我还没有成家，也没有其他经济负担，所以我可以承受创业不成功再回头找工作的风险。但并不是所有人都像我一样。如果你真心喜欢做点什么，坚信朝哪个方向努力一定成功，觉得自己能承受得了风险，那就不要犹豫。

图灵社区：你经常上哪些在线社区？请给中国读者推荐一些有用的在线资源。

Zakas：我最近没怎么上网上社区，我是 Twitter 控，关注那些能告诉我 Web 技术发展走向的人。我很愿意多花时间在线下跟人交流，比如在公司里，或者在会场上，这样可以了解到最前沿的东西。

我泡 GitHub 的时间非常多，有时候是看别人的项目，有时候是弄自己的。我在上面经常看到有人讨论代码该怎么写，这样写是为什么之类的，都非常精彩。而通过看别人的代码真的能学习到很多东西。在碰巧遇见自己有感觉的项目，而又认为自己可以提供一些不同思路时，我甚至会为这些项目贡献一些代码。■

[相关阅读：对话《JavaScript 高级程序设计》作者 Nicholas C. Zakas（英文版）](#)

[相关活动：图灵社区征集问题活动](#)

通过对象图学习 JavaScript



作者 / Tim Caswell

[Tim Caswell](#) 从 11 岁起就开始写代码。制作很酷的玩意儿对他来说既为了谋生也为了取乐，目前他正参与多项开源项目的开发（包括 [creationix](#)）。他是 9 月份举行的 [沪JS 2012](#) 主讲人之一。

若想成为一个高效的 JavaScript 开发者，其秘诀之一就是真正理解这门语言的语义。本文将会通过通俗易懂的图表来解释 JavaScript 中最基本的核心内容。

随处可见的引用

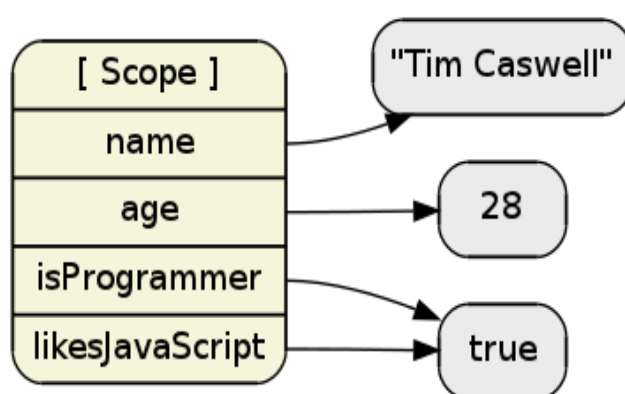
JavaScript 中的变量其实是一个标签，它引用了内存中某个位置的某个值。这些值可以是字符串、数字和布尔值的原始值。它们也可以是对象（object）或函数（function）。

本地变量

在下面这个例子中，我们会在顶级作用域中创建四个本地变量，并将它们指向某个原始值。

```
// 我们在顶层作用域创建一些本地变量
var name = "Tim Caswell";
var age = 28;
var isProgrammer = true;
var likesJavaScript = true;
// 测试一下看看两个变量是否引用了相同的值
isProgrammer === likesJavaScript;
```

输出 => true



* 即 JavaScript 解释器
——译者注

注意两个布尔值指向的是内存中的同一位置，这是因为原始值是不变的，因此虚拟机 * 可以优化它们，使所有的引用共享这个原始值的同一实例。

在这个代码片段中，我们使用 `===` 来判断两个引用是否指向同一个值，得到的结果是 `true`。

外面的框代表最外层的封闭作用域。这些变量是最顶级的本地变量，不要把它们跟 `global/window` 对象的属性相混淆了。

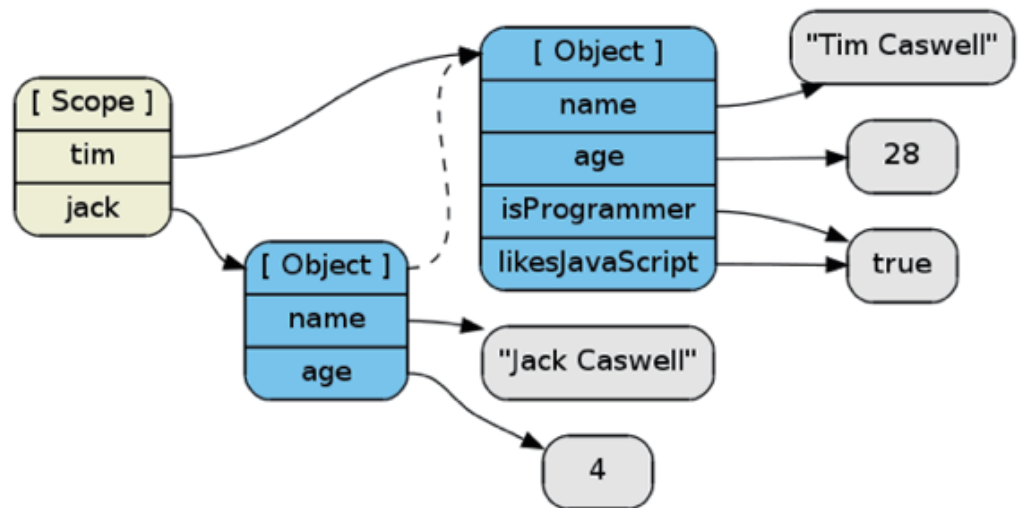
对象和原型链

对象只不过是更多引用的集合，它们指向新创建的对象和原型。它们唯一增添了一点比较特殊的特性就是原型链（**Prototype Chains**），当你试图访问一个不属于本地对象而属于其父对象的属性时，就会用到原型链。

```

// 创建一个父对象
var tim = {
  name: "Tim Caswell",
  age: 28,
  isProgrammer: true,
  likesJavaScript: true
}
// 创建一个子对象
var jack = Object.create(tim);
// 覆盖一些本地属性
jack.name = "Jack Caswell";
jack.age = 4;
// 通过原型链进行查找
jack.likesJavaScript;
输出 => true

```



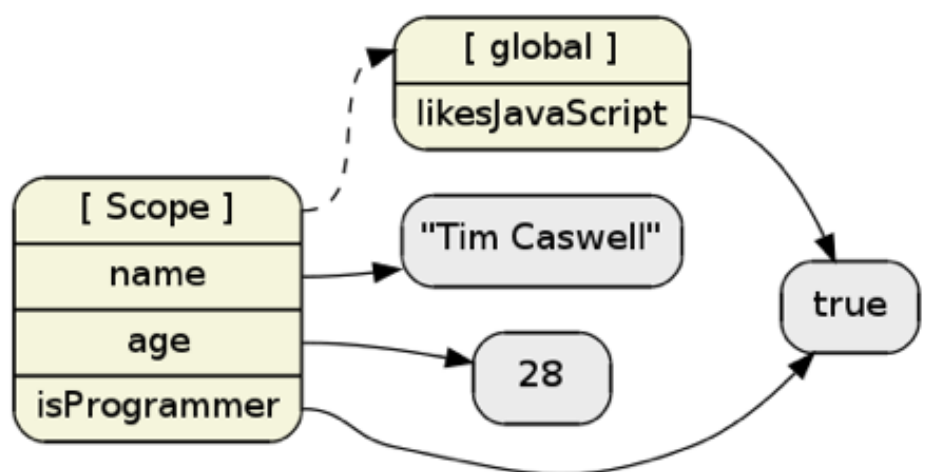
这里，我们有一个包含四个属性的被 `tim` 变量所引用的对象，同时我们创建了一个新的对象，该对象继承自第一个对象并且引用为 `jack`，然后我们覆盖了本地对象的两个属性。

现在，当我们查找 `jack.likesJavaScript` 时，起初会找到 `jack` 所引用的对象，然后会继续查找 `likesJavaScript` 属性。由于本地对象中并不包含该属性，因此我们查找其父对象并找到了该属性，最后则找到了该属性所指向的 `true` 这个值。

全局对象

你想知道为什么像 [jslint](#) 这种工具经常会提示你别忘了在变量的前面增加 `var` 声明吗？好吧，我们看看如果丢掉的话会发生什么情况。

```
var name = "Tim Caswell";  
var age = 28;  
var isProgrammer = true;  
// 不小心丢掉了 var  
likesJavaScript = true;
```



注意，现在 `likesJavaScript` 已经是全局对象的一个属性，而不是外层封闭作用域中的一个自由变量了。尽管这种情况只有在混

搭几段脚本时才会有问题，不过，在任何真正的程序中都会出现混搭的情况。

请牢记一定要添加这些 `var` 声明，这样才能保证你的变量是在当前的作用域及其子对象的作用域中。遵循这个简单的规则将使你受益匪浅。

如果你必须要在全局对象上添点儿东西，那么在浏览器中就明确地使用 `window.woo`，而在 `node.js` 中则使用 `global.goo`。

函数与闭包

JavaScript 并不只是一系列的链式数据结构，它还包含了被称作函数（function）的可执行可调用代码。这些函数会创建链式作用域和闭包（closure）。

可视化的闭包

函数可以被看作是包含可执行代码及属性的特殊对象。每个函数都有一个特殊的 `[scope]` 属性，它代表了函数被定义时的环境。如果一个函数是由另外一个函数返回的，则这个指向旧环境的引用就会在一个“闭包”中被新的函数所终止。

在这个例子中，我们会创建一个简单的工厂方法，它可以生成一个闭包并返回一个函数。

```
function makeClosure(name) {  
  return function () {  
    return name;  
  };  
}
```

```
};  
}  
var description1 = makeClosure("Cloe the Closure");  
var description2 = makeClosure("Albert the Awesome");  
console.log(description1());  
console.log(description2());  
输出 Cloe the Closure Albert the Awesome
```

当我们调用 `description1()` 时，虚拟机会查找它所引用的函数并执行。由于这个函数会查找一个名为 `name` 的本地变量，因此它会在闭包作用域中进行查找。这个工厂方法的好处就是，每个生成的函数都有自己的本地变量空间。

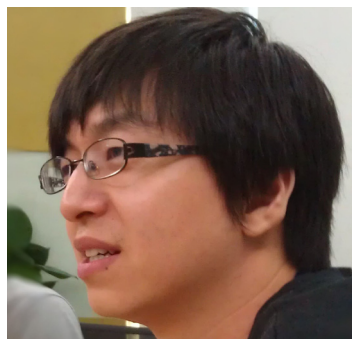
参见这篇“为什么使用闭包” [\(why use closure\)](#) 来获得更多关于闭包及其使用的内容。

共享的函数和 `this` 关键字

有时由于性能的原因，或者因为就是喜欢这种风格，JavaScript 提供了一个 `this` 关键字允许你在不同的作用域中依据函数被调用的形式来重用函数对象。

这里我们创建一些对象，它们全部共享同一个函数，这个函数会在内部引用 `this` 来展示调用过程中的变化。

```
var Lane = {  
  name: "Lane the Lambda",  
  description: function () {  
    return this.name;  
  }  
};  
var description = Lane.description;
```



译者 / 姬光

@ 姬小光，前端开发一枚，
目前就职于腾讯电商控股
用户体验设计部，《精彩绝
伦的 CSS》译者。博客：
<http://44ux.com>。

图灵社区 ID: [姬光](#)

```
var Fred = {  
  description: Lane.description,  
  name: "Fred the Functor"  
};  
// 从四个不同的作用域调用函数  
console.log(Lane.description());  
console.log(Fred.description());  
console.log(description());  
console.log(description.call({  
  name: "Zed the Zetabyte"  
}));
```

输出：

Lane the Lambda Fred the Functor undefined Zed the Zetabyte

在此图中，我们看到即使 `Fred.description` 被设置成 `Lane.description`，它实际上也只是引用了函数本身。因此，三个引用都同样对匿名函数拥有所有权。这就是为什么我没有在构造原型上通过“method”来调用函数的缘故，因为这意味着函数与其构造器和它的“类”之间的某种绑定关系。（详见[“什么是 this” what is this](#) 获得更多关于 `this` 的动态本质的细节。）

结论

我很乐于用图表来使这些数据结构可视化，我希望这些内容可以帮助我们这些视觉学习者更好地掌握 JavaScript 的语义。我曾有过前端开发 / 设计和服务器端架构的经验。我希望我独特的视角能够帮助那些从设计岗位过来，并想深入学习这门被称作 JavaScript 的美妙语言的同学。■

查看原文: [Learning Javascript with Object Graphs](#)

我的精神家园

——陈皓（@左耳朵耗子）专访



经历

“我就想要去经历一些未经历过的事情，这样老了以后才不会后悔。”

98年大学毕业，你找到了一份令旁人羡慕的银行工作，后来为什么离开了？

陈皓

老牌码农代表陈皓（@左耳朵耗子），酷壳 coolshell.cn 博主。芝兰生于深谷，不以无人而不芳；君子修身养德，不以穷困而改志。

14 年软件开发相关工作经验，8 年以上项目和团队管理经验，6 年的软件行业咨询经验。擅长底层技术架

陈皓：我当时在银行做银行网络、银行的电子邮件系统和办公自动化系统。当时正处在银行信息化的阶段，加上互联网和 IT 业刚刚火起来，得到这份工作其实是很幸运的。银行正值扩张电子信息化业务的时候，其实应该有很多事可做，但是当时的主要工作都是由厂商来干。比如说 IBM 或 Cisco 拿下单子来，会把工作外包给系统集成商。作为一位技术人员，其实可以发挥的空间并不大，多数时间我只是出了问题打电话的角色。没有人会教你任何事，出了问题，就是打电话，然后按照他们的指导来完成工作。但这个还不是促使我离职的最主要原因，我离开是因为互联网和 IT 业的兴起让我有些心向往之，有想去看一看的冲动。我还记得当时

构，团队建设，软件工程，软件研发咨询，以及全球软件团队协作管理。对高性能、高可用性、分布式、高并发，以及大规模数据处理系统有一些经验和心得。现于 Amazon 中国任研发经理，负责库存预测和电子商务全球化业务(全球开店)的研发。

的辞职书是这么写的：“本人对现有工作毫无兴趣，申请辞职。”处长说，“你可以这么写，但是要加上‘经调解无效’，另外，分给你的房就不能要了。”我说好啊。就这样辞去了工作，去了上海。老实说，这个决定真不好做，因为几乎所有的朋友和亲人都很反对。

离开了原来的工作，为什么选择来到人生地疏的上海？

陈皓：选择上海是有原因的，我觉得在当时的环境（2000年）下，上海的发展比较不错，没有选择深圳的原因是个人感觉那是因为政治原因凭空冒出来的一座城市，我不是很喜欢，而北京又有很多同学，所以想去一个陌生的地方。但是后来发现上海也不是做技术的地方，过得有些压抑，初来到上海的时候经常会被人瞧不起，毕竟是刚刚来到大城市。

我当时感觉银行束缚了我，想看看自己可以跑多远，能发挥出多大的价值。于是决定出来闯一闯，我就想要去经历一些未经历过的事情，这样老了以后才不会后悔。当时 IT 产业的发展是一个大趋势，我感觉我必须要去一座大城市，去经历一些东西。在小地方基本没有这些机会。要学会游泳就必须跳进水里去呛两口水，所以我就义无反顾地出来了。

在上海你有什么重要的经历吗？

陈皓：我仍然记得自己拎着皮箱站在上海火车站的样子，举目无亲。原来在老家的时候觉得自己还挺厉害的，自以为不愁找不到好工作。不过事实却不是这样的。

我还记得第一次去面试时，（面试官）问了很多和 C 相关的问题，问了我半个小时，我一个问题都答不上来。我一直低着头，好像被审问的犯罪分子一样。我从大学毕业出来就没经历过什么面试，再加上自己内向的性格，所以，整个过程我都在低着头，不敢看别人一眼。最后，面试官问了我一个问题是“有不懂的问题你会怎么办”，这样的问题我都不敢回答，其实这道题的答案不过就是“问别人”或是“自己看书”或是“上网查资料”什么的。很显然，这场面试我肯定是被灭掉了。但这还没完，最后面试官对我说：“你出来干什么，像你这种性格根本不适合（到大城市来）。”我当时被严重地打击了，感觉到自己确实有一些东西很差。第一个是性格差，不知道怎么与人交往；第二个是技术差，很多问题不知道；第三个就是视野狭窄，没见过世面。后面的几家公司的面试都大同小异。一个人在异地他乡，经历了这些事情，心里会非常地恐慌，“我这条路是不是走错了？”我经常这样问自己。

面对这样的情况，我被逼迫着一定要改变自己。因为，离开银行时，我的家人、同学和朋友都很反对我出来，如果这样灰溜溜地回去，我面对不了他们。而前面的人还看不起我。我的处境真的很难堪，就像爬在悬崖中间，上不去也下不来。所以，当时只有一个想法，就是要证明自己不是那么差的人。人被逼到那个份上，活得就比较简单，哪有什么职业发展规划，只想拼命地多学技术，提高自己的能力。这个经历有点像是一剂兴奋剂，同时也相当阵痛。但是回头想想，第一个面试官应该是最感谢的人。

后来，你在当时这种前有敌人后有追兵的状况下怎么坚持下来的？

陈皓：在同学的帮助下我找到了在上海的第一份工作。南天公司，

这是一家给银行做系统集成软件的公司，大学毕业时本来也可以进去，现在绕了一圈而且还是靠同学帮助进去的，所以那时的心态还很不平稳，另一方面因为以前是做银行的，是甲方，现在成了乙方了，两边的人都用异样的眼光看我，心态非常不好。不过，这是个技术不错的企业，国内早期很多搞 Unix/ C 的高手都是从这个公司培养出来的。我当时的技术还是不行，比如说到了用户站点以后，不知道怎么做，我曾经误操作把用户的数据删掉了。经常犯低级错误，不但没做好自己的工作，而且还给别人添了麻烦。这些经历都让我有一种“技术焦虑感”，或者叫“技术忧郁症”。我觉得自己这也不行，那也不行。这也是我今天仍然在拼命学习的原因。这就好像我们经常参加工作多年后还会梦见自己的英语四级没过，或者是期末考试没过一样。我经常梦见的是项目又做砸了，又把用户的系统搞乱了，一大堆人要审我、要训斥我。

因为技术差，沟通差，不会面试，所以，我决定经常出去面试。基本上每周都要去，不管懂不懂，也不管是什么公司，也不管别人是否鄙视我，反正就一有机会就去面试。多见见人这样可以让我性格有所改善，同时，也可以知道社会上需要一些什么样的技能，把别人问我回答不上来的东西都记下来，然后回头找答案。那个时候我会经常去上海书城看书，看很多很多的书。我学的东西很杂，什么做网页，Windows，Unix，Java，.NET，Flash，连 3DMax/Photoshop 我也学，还去考 CCNA 的认证……。这样散乱地学习两年后，我才慢慢确定了要走 C/C++/Unix/Windows 系统底层的路子。而这样扑天盖地学习的结果有一个好处就是，我成长的速度相当之快，重要的是我自己摸索到了适合我学习的方法（从基础和原理上学习），从而不再害怕各种新的技术。那时，所有人都在休黄金周出去玩的时候，我还呆在办公室或住处看书学习。

等到一年半之后，用句赵本山的台词说，我在面试中学会抢答了，他的问题没问完，我就能说出答案。其实，基本上是面一个公司过一个（当然都是一些小公司），此时，我就开始挑公司了。

在那里，感到技术能力不行就去学技术，交往能力不行我就去面试，这两个问题都可以通过大量地实践和努力来弥补，但是眼界这个东西没有办法通过努力来弥补。所以，当时非常想去一些更大的公司去看看，如果能去外企更好。

你从什么时候开始感觉自己已经变得和以前不一样了？

陈皓：我还记得，有一天，有一个和网络相关的技术问题，同事们搞了三四个通宵，也没弄明白，后来想起我好像在看这方面的书，他们就让我去看看、试试，结果我只用了 20 分钟就搞定了。基础真的很重要，这受益于我看了 [《TCP/IP 详解》](#) 这套书。

后来，我去了一家做电信软件的公司，他们让我做 PowerBuilder，尽管我当时想做的是 C++，但是因为当时各种原因很需要这份工作，就去了。进了那里的第一天发现公司里有一个论坛，上面都是一些技术上悬而未决的问题，都是关于 Windows/C++ 的。我一看，都是些很简单的问题，一下午的时间就被我全部解决掉了，我的基础知识发挥了作用。于是，当天下午我一下子就被调到了核心组。不过，我只在那里呆了两个多月，因为那时我已经不愁找工作了，这期间有两家北京的公司录用了我，于是，02 年我就来到了北京，去到一家做分布式计算平台软件的公司。

在上海的这两年的时间，从什么都不是，发展到得到工作上的全面肯定。那段时间感觉自己牛得不得了，有些狂妄和骄傲了，经常上网和不认识的人争论一些很傻的问题，后来发展到对当时的

领导以及银行客户的领导不敬，总觉得这些人太二。现在回头看过，我觉得那是我人生特定时期的记号，人生的痕迹。

关于酷壳

“为什么说我们技术人员是书呆子？其实我们有很多有趣的东西，只不过是
你不知道而已。”

你为什么要创建一个像酷壳这样的网站？

陈皓：我 2002 年在 CSDN 开了一个 blog，当时叫专家专栏。开个专栏很简单，只要发 6 个帖子。我也不是什么专家，只是喜欢看书、喜欢学习而已，也喜欢做一些学习笔记。那时候没有笔记本也没有台式机，市面上好像也没有 U 盘和移动硬盘。正好 CSDN 有这么一个地方，就去 CSDN 的站点上把自己的一些学习笔记放在了上面。后来 03 年的时候技术专栏转到了博客，因为 CSDN 对其博客经营得不好，我 09 年就离开了 CSDN，创建了酷壳。花了 4500 块钱，租了一个 server。我离开那里主要有两个原因，一个是因为当时 CSDN 博客有一些性能上的问题，.NET 架构嘛，大家都懂的（笑）。另外一个原因就是当时出现了很多博客营销的站点，有点像今天的 36 氪。好像那时候出现最早的叫煎蛋，那上面会有一些报纸上不会出现的国外的趣闻，是以博客的方式形成的媒体。这和常规的以日记形式出现的博客大不一样。煎蛋、有意思吧等这些博客让我看到了博客还能这样写，我觉得很好玩儿。而我也经常会去国外社区看一些文章，也能看到一些有意思的东西（因为我当时有了学习瓶颈，国内的网站已经满足不了我了）。心想，既然这些东西这么有意思，我为什么不自己开一个博客呢？

我老婆是学新闻编辑的，她鄙视我说，你的博客虽然有很多人读，但是只能算是个书呆子的博客，全是一些书呆子式的文章。我有些不服，为什么说我们技术人员是书呆子？其实我们有很多有趣的东西，只不过是你知道而已。于是我想弄一个有意思的、有娱乐性质的东西，里面都是技术圈里面有意思的事儿，但是很多技术圈以外的人也能看懂。一开始酷壳和 CSDN 博客的风格完全迥然，如果有技术性的文章我还会在 CSDN 上贴，但是后来我就完全抛弃了原来 CSDN 上的博客。酷壳的初衷是希望很多人都可以来上面发表一些东西，但是可能是我写得太多了，别人就被压制住了。

在博客的维护方面一直以来你是怎么做的？

陈皓：现在更新频率是一周一篇，一开始的时候一周三篇。磨刀不误砍柴工，总是有时间来做这些事的。我经常看书，需要把学到的东西整理成学习笔记。自从在 CSDN 上写博客的时候，就有这样的习惯了，而且又有“技术焦虑症”，害怕跟不上，所以维护博客的事对我来说是很自然的。

现在我已经不用自己再租服务器了，由于酷壳的访问量比较有保证，我提供了广告位，就免费得到服务器了（笑）。

关于老的和新的技术

“技术的发展要根植于历史，而不是未来。”

对于日新月异的新技术，你是什么态度？

陈皓：遇到新技术我会去了解，但不会把很大的精力放在这儿。这些技术尚不成熟，我只需要跟得住就可以了。我的团队自己想学什么我都不干涉，但是用到项目里的技术，必须是很成熟的，（技术应用）十年以上可能是一个门槛。有人说技术更新换代很快，我一点儿都不这样想。虽然有不成熟的技术不断地涌出，但是成熟的技术，比如 Unix，40 多年，C，40 多年，C++，30 多年，Java 也有将近 20 年了……，所以，技术并不多啊。还有很多技术比如 ruby，lisp 这样的，它们没有进入主流的原因主要是缺少企业级的应用背景。

如果要捋一个脉络下来，70 年代 Unix 的出现，是软件发展方面的一个里程碑，那个时期的 C 语言，也是语言方面的里程碑。（当时）所有的项目都在 Unix/C 上，全世界人都在用这两样东西写软件。Linux 跟随的是 Unix，Windows 下的开发也是 C。这时候出现的 C++ 很自然就被大家接受了，企业级的系统很自然就会迁移到这上面，C++ 虽然接过了 C 的接力棒，但是它的问题是它没有一个企业方面的架构，否则也不会有今天的 Java。C++ 和 C 非常接近，它只不过是 C 的一个扩展，长年没有一个企业架构的框架。而 Java 发明后，被 IBM 把企业架构这部分的需求接了过来，J2EE 的出现让 C/C++ 捉襟见肘了，后面还有了 .NET，但可惜的是这只局限在 Windows 平台上。这些就是企业级软件方面语言层面这条线上的技术主干。

另外一条脉络就是互联网方面的（HTML/CSS/JS/LAMP…）。这条脉络和上述的那条 C/C++/Java 的我都没有放，作为一个有技术忧虑症的人，这两条软件开发的主线一定不能放弃。无论是

应用还是学术，我都会看，知识不愁多。何必搞应用的和搞学术的分开阵营，互相看不起呢？都是知识，学就好了。

技术的发展要根植于历史，而不是未来。不要和我描述这个技术的未来会多么美好，用这个技术可以实现什么花哨的东西。很多常青的技术都是承前的。所以说“某某（技术）要火”这样的话是没有意义的，等它火了、应用多了咱们再说嘛（笑）。有些人说不学 C/C++ 也是没有问题的，我对此的回应是：如果连主干都可以不学的话，还有什么其他的好学呢？极端一点，我要这么说：这些是计算机发展的根、脉络、祖师爷，这样的东西怎么可以不学呢？大部分学校虽然都会教授 C，但是教得都不好。学校喜欢教微软的东西，老师好教学生好学。我不是说 Windows 不好，但那不是计算机文化的主干，那只是微软的主干、PC 的主干。整个计算机文化的主干肯定是源起于 Unix/C 这条线上（注意，我说的是文化不是技术）。我也写过很多与 Unix 文化相关的文章，大家可以看看我写的 [“Unix 传奇”](#)。

可是在应用环境中，对新技术的需求是很高的，你觉得在教育领域计算机科学的侧重应该是什么样的？

陈皓：学校教的大部分都是知识密集型的技术，但是社会上的企业大部分都是劳动密集型的。什么是劳动密集型企业呢？麦当劳炸薯条就是劳动密集型的工作，用不到学校教授的那些知识。如果有一天你不炸薯条了，而要做更大更专业的东西，学校里的知识就会派上用场。有人说一个语言、一个技术，能解决问题能用就行了，我不这样认为。我觉得你应该至少要知道这些演变和进化的过程。而如果你要解决一些业务和技术难题，就需要抓住某种技术很深入地学习，当成艺术一样来学习。

我在“软件开发‘三重门’”里说过，第一重门是业务功能，在这重门里，的确是会编程就可以了；第二重门是业务性能，在这一重门里，技术的基础就很管用了，比如操作系统的文件管理、进程调度、内存管理，网络的七层模型，TCP/UCP 的协议，语言用法、编译和类库的实现，数据结构，算法等等就非常关键了；第三重门是业务智能，在这一重门里，你会发现很多东西都很学院派了，比如搜索算法、推荐算法、预测、统计、机器学习、图像识别、分布式架构和算法，等等，你需要读很多计算机学院派的论文。

总之，这主要看你职业生涯的背景了，如果你整天被当作劳动力来使用，你用到的技术就比较浅、比较实用，但是如果你做一些知识密集型的工作，你就需要用心来搞搞研究，就会发现你需要理论上的知识。比如说，我之前做过的跨国库存调配，需要知道最短路径的算法，而我现在在亚马逊做的库存预测系统，数据挖掘的那些东西都需要很强的数学建模、算法、数据挖掘的功底。

我觉得真正的高手都来自知识密集型的学院派。他们更强的是，可以把那些理论的基础知识应用到现在的业务上来。但很可惜，我们国内今天的教育并没有很好地把那些学院派的理论知识和现实的业务问题很好地接合起来。比如说一些哈希表或二叉树的数据结构，如果学校在讲述这些知识的时候能够接合实际的业务问题，效果会非常不错，如：设计一个 IP 地址和地理位置的查询系统，设计一个分布式的 NoSQL 的数据库，或是设计一个地理位置的检索应用等等。在学习操作系统的时候，如果老师可以带学生做一个手机或嵌入式操作系统，或是研究一下 Unix System V 或是 Linux 的源码的话，会更有意思。在学习网络知识的时候，能带学生重点学一下以太网和 TCP/IP 的特性，并调优，如果能做一

个网络上的 pub/sub 的消息系统或是做一个像 Nginx 一样的 Web server，那会更好。如果在学图形学的过程中能带领学生实践一个作图工具或是一个游戏引擎，那会更有意思。

总之，我们的教育和现实脱节太严重了，教的东西无论是在技术还是在实践上都严重落后和脱节，没有通过实际的业务或技术问题来教学生那些理论知识，这是一个失败。

精神家园

“当你老了的时候，回想过去，如果你是为自己而活的，你就不会后悔，而且会感觉很踏实。”

你如何在进度压力下，享受技术带来的快乐？

陈皓：中国人中庸的思想，入世和出世，每天的工作就是入世。举个例子，在上海的时候，给交通银行做项目的时候，每周休息一天，早九点到晚十点，每天工作 12 个小时，这样的工作持续了一整年，没有节假日，项目上的技术也没什么意思。当时我晚上十点回到住处，还想学一些 C++/Java 和 Unix/Windows 的技术，于是就看书到晚上 11:30，每天如此，一年下来学到很多东西，时间没有荒废，心里就很开心。我觉得当时是快乐的，因为有成长的感觉是快乐的。

现在的我，工作、写博客、养孩子，事情其实更多。我早上 7:30 起床，会浏览一下国外的新闻，hacker news, tech church, reddit, highavailability 之类的站点，9 点上班。晚上 6、7 点钟下班，开始带孩子。十点钟孩子睡了觉，我会开始重新细读一下这一天都发生了些什么事情。这个时间也有可能用来看书。学习的过程（我）

是不喜欢被打断的，所以从十点到十二点，家人都睡了，这正是我连续学习的好时间。可能从晚上 11:30 开始，我会做点笔记或者写博客。我现在对酷壳文章的质量要求比较高一些，所以大概积累一个星期的时间才可以生成一篇文章。每天我大概都在一两点钟才会睡觉。没办法，我有技术焦虑症。但是觉得这样的生活很充实，也很踏实。

另外，任何一门技术玩深了，都是很有意思的。有些人形成了一个价值取向，“我只做什么，绝不做什么”。前段时间有一个刚来亚马逊的工程师，他原来做的是数据挖掘推荐系统，后来公司重组要他做前端，他不肯。我觉得，前端后端都是编程，JavaScript 是编程，C++ 也是编程。编程不在于你用什么语言去 coding，而是你组织程序、设计软件的能力，只要你上升到脑力劳动上来，用什么都一样，技术无贵贱就是这个意思。

回到问题，怎么才能享受到快乐呢？第一，入世和出世要分开，不要让世俗的东西打扰到你的内心世界，你的情绪不应该为别人所控，也不应该被世俗所污染，活得真实，你才会快乐。第二点就是要有热情，有了热情，你的心情就会很好，加班都可以是快乐的，想一想我们整个通宵用来打游戏的时光，虽然很累，但是你也很开心，这都是因为有了热情的缘故。

有人说你现在的文章仍然说明你是一个躺在自己池子里说话的人，是不是说你仍然没有达到一个很高的层次？

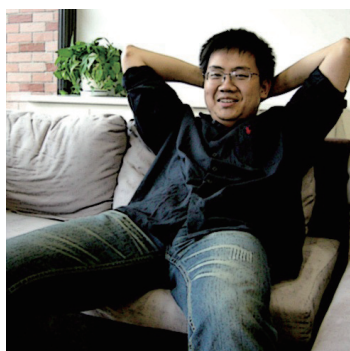
陈皓：我承认我活在我的精神家园里面。我推荐大家看一下王小波的《我的精神家园》，这篇文章对我的影响非常大。看了这篇文章，你就会明白我为什么要躺在自己的池子里，如果不想被这

记者 /@ 谢工、@ 李盼

个社会所污染，就必须要躺在自己的池子里。做大众是很容易的，做自己是最难的。当你老了的时候，回想过去，如果你是为自己而活的，你就不会后悔，而且会感觉很踏实。可能有人会觉得我偏激，没关系，为什么要所有人看法都一致呢？世界因为不同而美丽，多元化的价值观并不冲突。■

以“玩”之名

——赵劼 (@老赵) 专访



赵劼
新派码农代表赵劼，网名 [@老赵](#)，洋名 Jeffrey Zhao，享乐派码农。最爱美女，其次是编程和弹钢琴。
目前就职于 IBM，高级咨询师。InfoQ 中文站编辑，多次受邀于微软 TechED，MSDN WebCast 及各微软官方或社区会议中担任技

编程人生的开始

一开始大家都是每天在一起做类似的事情，但是到后来你就会发现，你们在以后的发展路径中就再也没有交点了。

你是什么时候开始学编程的？

老赵：我小学毕业的时候搞来一台 486 电脑，上面装了一个 DOS 系统，于是我开始搞最简单的编程，当时就是为了好玩。我用 DOS 写了一个 bat，功能大概是在系统启动之后自动执行，生成一个菜单，可以上一页下一页或者按个数字进入某个应用，退出后重新回到这个菜单，目的就是为了避免输入命令，方便操作。当时还没有互联网，全靠一本叫“DOS6.2”的书，这是唯一的渠道，现在回想起来，这些东西好像也挺有难度的，不是那么简单（笑）。后来（写代码）就是为了玩游戏，什么大富翁 3，大航海时代什么的，都是那个时候的经典。我一直对计算机比较感兴趣，小时候还学了十年钢琴，但是我敢说基本没有琴童是对钢琴感兴趣的，可能

术议题讲师。最近致力于 Wind.js 项目的开发与推广，并关注 F#，Scala 语言及 mono 平台在社区中的发展。

长大后会有些兴趣，但是当时肯定都是被逼被打的。后来我发现自己太笨，玩不好游戏。上大学的时候大家流行玩 CS，（玩游戏时）我人都没看到一个，就死了，下一局，还是没看到人，我又死了，我就搞不清楚这个东西的趣味在什么地方，所以觉得不如把时间花在写程序上比较有趣。

我的升学经历就是全考，我的有些同学都是一路直升，没有经过大考，而我每次都是差一点点。小学考初中时，老师总说你们要是不好好学习，到时候就差 0.5 分考不上，结果那个人就是我。高考的时候，我努力搞计算机竞赛，想去交大。我其他成绩比较差，一直以来的打算就是不高考，精力都是用在信息奥赛方面。结果我高考那年交大拿了一个 ACM 的世界冠军，显得很牛，当时有五大联赛，数学、物理、生物、化学、计算机，交大只要数学或者物理的直升，而复旦都要。最后没办法，还得高考，我用仅剩的半年时间拼命读书，同时因为痛恨交大所以报了复旦，还好最后考进了。

你所接受的大学教育有什么可圈可点之处吗？

老赵：进入复旦的软件学院对于我来说是很幸运的，当时的院长很有水平，在理念上很先进。从大一开始我们的所有教材都是国外的，全英文。其中大部分都是 MIT 等国际名校的教材，比如《深入理解计算机系统》那本书就是我们主要的教材，学编程基础课时也建议我们看 *Core Java*，学操作系统的时候用的也都是响当当的恐龙书，现在看来真的是很有价值。我们从大一开始啃大部头的英文原作，我只能硬啃，不能怕，这样学下来收获是很大的。当时人人都这样学习，所以整体环境也是很向上的，我同寝室和隔壁寝室的人大都比我强。可惜现在看来，我大学的时光还是浪

费了很多，算是比上不足，比下有余。

有时候我会很奇怪，为什么有些人没有看过《操作系统概论》，为什么有人觉得写一个语言或者编译器是一件很了不起的事情，为什么很多人也都没有自己写过操作系统。这些其实都是学校里的大作业，虽然我们写的操作系统也是比较弱的，但是那也是一个操作系统，线程调度，虚拟内存，文件系统等等，该有的都会有，也可以在虚拟机里启动。这些在我看来应该是人人都做过的东西，而很多人却没有做过。虽然我大学里专业课成绩也不算差，但是因为当时在学校的学习有时间压力，也贪玩，虽然不玩游戏但也会跟同学或女朋友玩，同时也很懒，所以教科书中的很多内容其实还是没有吃透。

还有一点就是课程的设计，我们还有函数式编程课程，我那时学 LISP，后两年的学生学的是 Haskel。学到后来有不同方向，无论哪个方向都可以毕业。有些人甚至学的是逻辑式编程，prolog 什么的。虽然很多东西对我的帮助可能不会直接感受到，例如我没有做过任何实际的操作系统或函数式开发，但是这些课程设计是很了不起的。

一直都有人问我不知道怎么进步，有什么书可以推荐。我通常的回答就是，你要提高什么我不知道，但我建议你回到大学的课程，也可以参考国外教科书中的内容，跟着教科书你肯定不会错。

在大学里，有些东西我要看三五遍才能懂，而我寝室的兄弟只看一遍就懂了。他大一就看《具体数学》，甚至高德纳那套《计算机程序设计艺术》那样的书，我希望有一天我也能看懂这套书。很多人在大学之初看起来都差不多，一开始大家都是每天在一起做类似的事情，但是到后来你就会发现，你们在以后的发展路径

中就再也没有交点了。我认识的一起在暑假里做过项目的人中就有人去了普林斯顿，走上了科研道路，我觉得我似乎永远都无法企及他那种高度了。

理想？现实？

这段创业经历给我的启示就是，我不想当老板，我就想当码农，快乐地编码。

你有没有什么梦想？

老赵：我从小就想当科学家。科学家是推动人类进步的源动力，我相信世界是由科学家推动的。他们不一定要有很强的工业背景，我所崇拜的科学家就是搞研究的，并不急着把自己研究的东西转化成实际产品，世界上不缺急着赚钱的人。他们（科学家）可能会做 10 年、20 年以后才能得到有实际用途的东西，等到这些东西真正发展成熟了工业界才会拿去用。科学家会把世界向前推动一点点，而不是在已知的领域把已知的东西反复利用，炒作价值。比如华尔街那些用钱生钱、玩数字游戏的一群人，他们可以赚很多的钱，但是他们对人类发展的贡献很小。我尊重个人的选择，有人喜欢赚大钱让自己过得舒服一点，但对我个人而言对于科学家更有认同感。我希望以后可以赚满足够的钱，不用工作就可以生活，那时我愿意重新去读书，重新进入学术界。

我承认科技有它的副作用，但是那是由于人类的滥用。例如，现在由于科技只是发展到一定程度，它能产生让人吃上去舒服但是并不健康的东西，但是如果再往前发展，我相信添加剂可以做到提高口感而且很健康，而目前这些负面的效果可能只是因为科技

发展还不够，人类还没有准备好。现在的社会环境就是做好事可能没有什么直接好处，但是做坏事有可能会拿到很多钱，人们权衡利弊，还是决定做坏事。假如做好事和坏事的成本和收益相差不大，我相信人们还是会选择做好事的。

我认为宗教和科学并不抵触，宗教让人对未知的世界保留一丝敬畏，科学解决不了的东西也是存在的。科学是在探索世界，而不是在发明世界。

谈一下你职业发展的历程吧。

老赵：我第一份工作是在一家国内二线的大型网站，开发他们的媒体平台。我忍不了程序员们无所事事，甚至还不想好好做事情的氛围，所以只呆了六个月左右就离开了。举个例子，他们有的人可能会复制粘贴 50 次，也不会想到要提取出一个公用的方法。这地方的氛围跟我理想里的科研机构根本没法比，就像是你说草履虫不是哺乳动物一样，中间还隔着十万八千里呢。

后来我和陈黎夫一起创业两年多，做一个女性奢侈品网站。这段创业经历给我的启示就是，我不想当老板，我就想当码农，快乐地编码。创业对我来说没什么吸引人的地方。有了这段经历，人家再跟我说创业的时候，我就可以说：创业，我早创过了。当时刚出校门没多久，思路也不清楚，说实话这次创业也只是一个普通的工作，只不过自由一些，工资少些。

后来我去了盛大，差点试用期没有过。因为老板坐在我身后，一抬头就能看见我的屏幕，他就会问：你怎么又在刷微博啊？别人都是试用期过了直接入职，我当时和老板谈老半天。其实我干的活也不比别人少，麦库的架构都是我一点一点搭出来的，第一行

代码就是我提交的，现在的 API 也是我当时绞尽脑汁设计和实现出来的，自诩十分漂亮，十分 Restful。虽然我不喜欢做产品，也不认同只有产品才能体现技术的价值或是技术脱离产品就失去意义等观点，但就算从产品角度来衡量工作成果，好像我还是比很多人干得有价值呢。盛大创新院诞生过很多项目，死掉的不少，突出的不多，但麦库算是其中比较突出的一个吧。

现在我在 IBM，去 IBM 并没有经过猎头，我似乎不是猎头们喜欢的类型（笑），基本没人来找过我。来 IBM 的主要原因是在深圳工作，面向香港客户，我想这样在香港产子会方便一些，这方面对我来说还是很有吸引力的。我现在是香港公民，但不是永久居民。只要有工作签证就能拿到香港身份证，便是所谓的香港公民了，但必须连续六年还是七年是香港公民才能成为“永久居民”。可惜我刚去香港没多久，香港就规定“双非”子女即使在香港出生也不算永久居民，直接摧毁了我南下最主要的动力。我一直说我运气很不好，读书时每次直升都差一点点。我的小学初中关门了，高中大学都堕落了，之前提到我很敬佩的院长也因为某些原因去交大了，至于呆过的公司则要么倒闭要么走下坡路。

我的自我调节能力还是很强的，总能在工作中找到自己喜欢做的事情。在 IBM 的时间对我来说过得很快。我对工作的要求就是不要让我不停地加班，没有自己的时间，晚上七点最晚八点对我来说就应该结束工作了。我是享乐派码农，最爱美女，其次是弹钢琴，接着才是编程，所以我不会去创业，也自觉地去不去加入创业公司。我自诩加入哪家创业公司哪家就必倒，至少工作态度上会被我带坏。创业跟享乐还是很难共存的。

你现在在 IBM 的工作内容是什么？

老赵：很多人其实不知道我做些什么，他们能看到的最多只是我写的文章等东西。我平时的工作既不流行也不火，只是一般的平常工作。我现在在 IBM 里做投资银行的相关项目，完全不涉及 IBM 内部事务。比如说给投行做一些交易系统前端之类，比如说处理交易，聚合一些数据给人看等等。用户则根据这些数据制定下一步的行动决策，设法赚更多的钱。在银行这种非技术为核心的不差钱的机构，很倾向于直接买现成的东西来一用。他们不差钱，可以拿钱换时间、换机会。我不少技术方面的工作实际都是围绕这些现成的产品，而银行的核心业务很少有人能接触到。核心业务，比如说根据大量的历史交易辅助交易员和研究员作出下一步决策，这里面需要大量的数学和算法知识，也需要对业务熟悉，而大部分人都是在做外围，买一个设备、买一个软件，然后使用。

我在 GDC，算是 IBM 的二等部门，这个部门可以简单认为是 IBM 开的外包公司，例如前段时间非常著名的苏宁项目，便是南京 GDC 参与的。GDC 前几年还不算是正式的 IBM 部门，后来才进入 IBM 的正式编制，但对于 IBM 一等部门，例如 IBM China 及 CDL 的一些人来说，GDC 的人都不算是真正的 IBM 员工。这个部门也很难让人产生归属感，虽然我的合同是长期的，但是有些项目的人签的都是短期合同，雇用期随着项目中止而结束。

你看起来精力充沛，你怎么分配你的时间？生活的时间，维护博客的时间，学习新技术的时间？

老赵：我 7 点左右下班，女朋友一般会早下班买菜回家，我到家后做饭，吃完饭可能会弹会儿钢琴，但是 10 点之后就不能再

弹了，邻居会疯掉的，我承诺图灵社区会录一段钢琴曲给大家（笑）。我之前还会花时间健身、减肥，但是减到 80（公斤）就再也减不下去了。他们都说我的体型是“正太”分布的曲线。其实我觉得我的生活还是挺丰富多彩的，例如除了弹琴外我唱歌也不错。之前我还参加了深圳 IBM 唱歌比赛，最后拿了第三名，而且在打分阶段我还是第一，只不过 PK 的时候败了。有人说码农苦逼，我倒觉得还好，我现在的同事有那么一堆人，每个周末都在一起玩，比如看电影啊吃饭啊 K 歌啊桌游啊或是去海边玩什么的，还有什么“吃遍深圳”计划，High 得不得了，反倒是我周末要忙着写代码或是陪女朋友活动，只是偶尔才加入他们。所以关键还是看自己啊，一是要热爱写代码，这样就不会觉得工作太累，二是要热爱生活，没说码农就只能宅在家的。

我博客的内容大部分都是平时在想，真正需要码字的时候其实不费多少时间。博客维护对于我来说是随时随地的事，你经常会看到我拿着一个平板电脑，有可能我就是在做这件事。其实我觉得写作就是在说话，把你想说的东西写清楚就可以了，怎么想就怎么说，怎么说就怎么写。Wind.js 可以说是提升我个人价值的东西。很难统计每周我花在这上面的时间，因为我可能每时每刻都在思考一些东西、构思一些代码，甚至在脑中进行实现，至少每时每刻都因为它在和别人进行交流。我现在在推广它，希望在未来某个时刻，忽然有一个大项目或者大公司决定使用它，或者收编它，这样它就能火了。

对于新的技术，我一般很少看这方面的书，但是我会把源代码花一点时间（比如说一个周末）看一下，里面有一些沟沟坎坎就全都清楚了，该怎么用，靠想也能想明白。对于我比较熟悉的技术，我可能从头到尾执行一遍就知道（怎么回事）了，然后在网上看

一些介绍用法的文章，我就能想到它是怎么做出来的。看一个项目我也知道从哪里看起，因为我知道它的执行过程是怎么样的。书当然也会看，但一般都是看些实现原理、设计思路的书，因为这些内容有时很难从代码里看出来。当然对于大部分我不会深入的技术，我可能就不会读源代码，直接看一些内部实现或是思路分析的书就满足了。

以“玩”之名

我为了玩技术而搞技术，这层次显然比为了做产品才搞技术要高得多嘛。

有人说你不会转换自我价值？

老赵：我觉得还好吧，我写博客，和大家交流也影响了很多人啊。我赚的钱也够花，按照 winter 同学的说法，虽然买不起房，很多人也买不起啊，但我去必胜客点最贵的套餐也丝毫没有压力啊。难道是说我不是分析师管理层，或不是什么 O 吗？其实我现在就是 Wind.js 的 CEO 加上 CTO 加上天使投资人，我还是 Founder，连 co-founder 都没有。

或者是因为我不写书？我还是情愿多写点程序，多写点零碎的文章来讲讲自己的想法。我喜欢自由，而写书会被太多的东西束缚住。我不喜欢在边角上投入精力，有这个时间，我情愿去做一个项目，例如把 Wind.js 的边边角角都做好，然后顺便写一些总结和体会。写书我认为需要面面俱到，把沟沟坎坎都填掉，而我现在还没有那种精力和状态。

我是个纯码农，还是享乐派的，所以我是为了兴趣才写代码，写

代码完全是因为好玩。有人说搞技术是为了做产品，没有产品技术就失去了意义，我倒不觉得。在我看来技术不仅仅是工具，还是玩具，甚至是艺术品。我为了玩技术而搞技术，这层次显然比为了做产品才搞技术要高得多嘛。你看动物只把交配作为繁衍后代的手段，而人类已经把交配当作娱乐和社会活动了。现在很多技术人员喜欢说产品，我觉得这其实挺有问题的。你知道自己在什么场合或者在说什么东西倒也罢了，但我看来很多时候就是一些不怎么样的技术人员在瞎找借口乱讲大道理。举个不怎么好听的例子，在讨论语言设计的时候总有人会嚷嚷“语言之争没有意义”什么的，在我看来这好比人类在讨论某些“技术性问题”的时候，动物们冲出来插嘴说“体位之争没有意义”。但其实呢？它们首先不知道并非所有物种都是为了繁衍才交配的，其次它们也不懂体位对于繁殖的效率也是很重要的。比如人类在帮狗配种之后，都会把母狗倒悬 20 分钟，目的就是为了增加受精成功率。

你看我微博的认证信息，不是那个自己随便填的个人说明，就是“资深码农”，我是真以码农身份为豪的。所以我也不会对什么业界大事或是传言做什么分析什么，在我看来那是互联网分析师做的事情，我挺不喜欢现在网上遍地都是的那种产品经理或者分析师。我甚至不会对技术做什么预测，因为预测什么的我觉得太不靠谱，太廉价，很多时候正过来反过去说都行，都太没意思了。我最多就根据“事实”发表看法，当然也就是在技术方面。

最近有很多关于裁员和失业的新闻，你觉得作为程序员需要掌握什么样的能力，学什么样的技术？

老赵：把自己和任何产业或者某种技术绑定在一起在我看来都是不太可靠的，对于我来说，我干任何工作都是以提高个人能力为

目标，这样安全一点。当然话说回来，现在想要绑定某种技术也不容易吧，搞 .NET 的失业了就不能去搞 Java 了吗？如果某人真把自己和某种技术绑死了，那基本就是自找的。我觉得现在很多同学遇到的困难都可以归结为自找的。例如，说程序员找不到女朋友？那是因为你不敢厚着脸皮去找，我就基本一追一个准，我觉得 IT 男其实挺有优势的。例如，说程序员看不懂英语书？那是因为你没有像我一样硬啃原版的大部头书，我的英语也就四级水平，之前在微博上被一大堆人鄙视的，但我现在英语做点技术方面的双语演讲也够用。例如，说程序员不会写文章作演讲？那是因为没有坚持写坚持说，我刚开始写的博客现在看起来也十分稚嫩，初中时竞选大队长我演讲时双腿发抖，同学都看得到，而现在面向几百人做演讲毫无压力，这都是自己逼出来的。好吧不说了，再说下去变炫耀帖了……

回到技术，我认为技术要根植于现在。有些人觉得专注于稳定的技术不怕找不到工作、吃不饱饭，但是吃得香不香、好不好就是另外一回事儿了。热门的东西可以捞一票就走，然后吃香的喝辣的去。例如，虽然 C 的历史十分悠久，重要性毋庸置疑，但是搞 C 的人一定能找到很好的工作吗？有可能这个市场已经饱和了，只有学得很好的那群人才能到很好的报酬。而如果你做很火的技术比如说 PHP，那里有大量的工作机会，没准可以得到很快的提升，你的生活水平和自身价值也会得到迅速的提高。

我在技术上很倾向于微软，但是就算（微软）真倒了我也不担心找不到好工作，因为我又没有把自己绑死在微软技术上，而且技术多少是相通的，去搞别家技术一样顺利。打个比方，当年 Google 最火的一件事情就是从微软大肆挖人，假如搞微软技术都必须靠微软才能混饭吃，那么那些人去了 Google 之后是做什么的？难道

是写 .NET 或做 Windows 开发么？还有，StackOverflow 总得分第一的 Jon Skeet，也就是我唯二或唯三推荐的 .NET 必读书籍《深入理解 C#》的作者，他搞了好久的 .NET，比我还深入也说不定，结果不也直接被挖去 Google 嘛。说起来我推荐的另一本 .NET 必读书籍 *Framework Design Guideline* 的作者之一 Brad Abrams 也跳槽 Google 了，所以实在不行我也可以去 Google 嘛（笑）。

我学技术唯一的标准就是要有意思，好玩，或者说有美感、符合我的口味。就像我很不喜欢指针和或纠缠于内存地址等大量细节，所以我就对 C 和 C++ 都敬而远之，尤其是后者，前者至少比较容易理解吧，大学里也用不少。其实我十年前在 IE6 上玩 JavaScript 的时候不也是破破烂烂的嘛，谁知道后来 Google 和 Chrome 把它带火了呢？

我相信的事

我不会为了推广自己的概念而忽悠人或者贬低别人。我只会说我认为正确的东西。

你是出了名的微软系，为什么对微软这么推崇？

老赵：我做 .NET 的原因是因为对 Anders Heisenberg 的崇拜，而后来技术这条路线是对微软的欣赏。经常听某些老一代程序员说微软对程序员怎么怎么不好，抛弃了多少技术，但我没这种体会。我应该算是随着 .NET 诞生开始搞微软技术的，之前做的更多的是 Java，中学时则更多用 Delphi 做过一些小游戏以及一些教学用的课件，而从搞 .NET 开始我就没觉得它有放弃的迹象。

我欣赏微软，是因为微软在软件研发，乃至研究方面的投入是最大

的，而且它在研究方面的投入可谓不计回报，所以你会发现微软研究院有很多案例都很有趣。而在开发技术方面，它的 DevLabs 里面也有许多有趣的项目和技术，例如 Reactive Framework，实在是让我眼前一亮，真心佩服那些人的聪明才智。Reactive Framework 的思路是对“拉”模型的接口取逆，成为一种推模型，然后一下子就可以推广至各式推模型和异步操作了。Reactive Framework 的提出者是 Erik Meijer，他是微软级别比较高的研究员，之前在大学里当教授，这种人我相当喜欢。微软的研究成果可能很长时间以后才会变成产品，比如说 Kinect。话说回来，我的 Wind.js 也是受到了他们的 F# 中计算表达式特性的启发，说实话 F# 也没有多火或是多流行，但有多少公司会用心发展一门受众不那么广的语言，并放入自己的主流产品中？像 Google 开发 Dart 或 Go 语言，目的都相当明确，就是要替换现有的 JavaScript 或 C++ 等语言。有人总说苹果公司研发投入比微软少得多，却赚得盆满钵盈，表明转化效率多么惊人，所以多么厉害。我的结论恰好相反，因为商业上的成功不能吸引我，相比起来我更佩服微软那种不计产出——当然肯定不会丝毫不计——的研发投入，这跟我佩服搞科研的人是一种思路，我就欣赏某些二杆子精神。

有人常会问我为什么不去微软工作，我之所以没有去微软是因为，理论上说，在微软你不可以读开源的代码。微软害怕你在看了开源的项目之后，会不知不觉地在工作中应用到开源东西，产生法律纠纷。这点真的很可惜，必须对微软做出严厉批评，还好微软开源的东西也越来越多了，F# 和 ASP.NET 一直开源，现在整个 ASP.NET 技术基本都开源了，还有 Entity Framework 等等。现在微软也在 Node.js 和 HTML5 等开放技术上投入很多。例如，在 iOS 上使用 HTML5 技术开发应用还有因为太简单被苹果拒绝的风险，而 Win8 直接把 HTML5 作为原生开发技术了，最近微软还和

某公司共同向 HTML5 移植了大量经典小游戏，所以我一直自嘲杂牌技术玩家，但搞了半天最后还是没有逃出微软的五指山（笑）。

你和开源社区、开源项目其实有着很多联系，你对国内开源现状怎么看？

老赵：我认为国内的开源属于开源初级阶段，就是说打着开源的旗号来做其他一些事情。举个例子来说，现在的开源项目貌似很多，每个公司都有开源项目，但是很多公司他们只是开源自己的东西，而不用别人的东西。开源自己的东西很容易，因为代码都是自己写的，我可以说我要弄一个开源项目，然后把我的代码放出去，然后这个事情就结束了。但是你有没有在持续更新呢？有没有对这个项目的周边进行持续性投入呢？

开源项目附带有一种正面的宣传效果，好像取得了道德制高点一样。用来体现公司对社会的价值，提高公司的层次。但是问题是，你自己用的是不是你开源出去的东西？有的公司本身在用一套，但是并不及时更新外面的程序，开放源代码很容易，但是配套的东西都没有做到位。这里还会涉及另一些问题，比如说，我从外界可不可以提交代码？外部想用这个东西有没有足够的说明文档？使用这个东西出了问题应该找谁？有没有什么地方可以讨论出一套解决方案？别的公司可以用你的东西吗？而你会用别人的东西吗？比如说 Hadoop，源于 Yahoo，很多公司都在用，Facebook 也在用，也在不断贡献，它同时也在开源自己的一些技术，Facebook 开源的技术 Yahoo 也能用，Yahoo 也会作出贡献。

开源之所以有意义就是因为，每个人都可以提交补丁，为它作贡献，最终达到大家都获益的效果。而国内的现状就是你不用我的，

我也不用你的，我不在意别人用不用我的东西，我也不在意这个东西接下来的发展，我只是作出一个姿态：这个东西开源了。比如说最近两年，大家都号称开源了自己的底层数据库，而为什么大家最终都只是各做各的？说白了这也不过是一种广告而已。而不是我为你的项目贡献，让我自己变得更好。我觉得一个好的状态应该是，一个项目出来，没有人在意这个项目是谁的，没有人算计和计较究竟谁获益较多。其实开源就应该是大公司搞了，用自己的资源和钱来支撑这个项目，最后做出有益的东西，而不是随便玩玩。

你经常被卷入网上的一些争论中，你怎么看待这件事？

老赵：我一直都认为自己是个不太聪明的人，但是虽然这样我仍然在茁壮成长。之所以有些人觉得我很自信，是因为我会花时间学习，我认为我已经掌握了令自己信服的东西之后，才会据理力争。可能有人花一个小时查几个名词解释就可以很有底气，但是我可能要花 5 个小时才能把一些东西搞明白。你可以“拍”我，只要你的道理是通的，我也会经常纠正自己的观点。我希望可以用作研究的态度来争论一些观点，但是实际过程中可能未必控制得好。

我觉得程序员在讨论技术话题时，不要扯太远，能给数据就给数据，没有数据也可以讲道理，不要绕。我很烦讨论的时候偏离主线，绕得很远。比如说，你说你的做法速度快，我测了一下觉得速度不快，你就说“这个做法还有其他好处”，结果就越扯越开。本来谈技术，谈到后来又扯到用户体验。前段时间在讨论内存大小的问题，有人说某某情况下有内存泄漏，然后我和别人做过实验没有内存泄漏，后来对方又说内存大有什么不好，结果实验下来实际上只是系统认为没有必要及时释放而已，接着对方又说这会让用户体验不好会让界面卡，但是为什么会影响用户体验？你必

记者 /@ 谢工、@ 李盼

须拿出一些证据。这种情况下话题越扯越多，本来一开始讨论的是什么都不管了。转移话题也是可以的，但是个中关系应该讲清楚，从内存大到用户体验，中间涉及一系列的为什么，这些就是应该说清楚的。

还有程序员大都比较“机灵”，无论什么结论都能找出理由来。例如，有人在苹果东西卖得不好的时候会说它的产品多么好，而新产品无甚亮点的时候，又会说这种保守策略很正常，你看卖得不照样很好么？这种“争论”就很没意思，正过来反过去都是你说了算，所以我也很讨厌许多产品经理和分析师。话说这方面还是 Amazon 好，其实我最喜欢的公司是 Amazon，它真正推动了云服务产业，而且电子书和平板设备都在不断超越自己，还卖得那么便宜，还没那么多不理智的粉丝。有些争吵的影响力很大，大家转来转去的，而很多这样的争论在我看来都没啥价值，但有时候我也会控制不住。

我不反感别人喷，但是我反感别人乱喷，我微博上也关注了一些喜欢喷的朋友，因为喷得有道理，不乱喷，我觉得挺不错的。我喜欢看别人吵架，但是这要建立在双方都能说出道理的前提下。网络上大部分的吵架都无法让人吸取到营养。很多人吵架都是意气之争，一语不和，就开始争论，都是想赢，而不是关注问题。当然我也不赞同有些人说在争论技术的时候一定要心平气和，口不吐脏字，用最平和的方式来讨论。最重要的是逻辑一定要清晰。我在推 Wind.js 的时候一定会把适用场合说清楚，而不是一概而论说它哪都好。我不会为了推广自己的概念而忽悠人或者贬低别人。我只会说我认为正确的东西。

所以我还真不适合当销售，完全就是个码农。■

通才还是专才

——由摩托裁员引发的讨论



作者 / 池建强

十多年软件从业经验，先后在洪恩软件和用友集团瑞友科技工作，目前任职瑞友科技 IT 应用研究院副院长。

图灵社区 ID: [池建强](#)

近日摩托罗拉强硬裁员像一个重磅炸弹，引发了很多争议和思考，相关新闻[在此](#)。我所在的 Google group 也进行了广泛而严肃的讨论，虽然视角不同，最后大家都落脚到了“专才 VS 通才”上面，IT 从业者到底是要把自己搞成专才——在某个领域独树一帜，还是博采众长，形成通才？

我忍不住也谈了些自己的看法，遂成文一篇，如下。

从事 IT 行业十年以上的，想不成为通才是很困难的，除非你确实是不思进取，而且十年不跳槽。很难想象一个工作近 10 年的技术人员说：我只会 Java！我们回顾一下，从 2002 到 2012，技术领域基本上进行了翻天覆地的变化，革新速度令人目不暇接。各种操作系统、编程语言、数据库（SQL 的、NoSQL 的）、各种框架、平台、业务系统，接踵而来，数不胜数，你得多专才会十年只用一种操作系统、编程语言和数据库啊？

所以对于要么通才要么专才的二选一问题，作为一个 70 后程序员，我是万不赞同的！

比较好的学习方式是全面熟悉你接触或主动学习的内容（非浅尝辄止），同时在其中根据兴趣和方向打造自己的专项特长。

很多同学说这不大可能，人怎么会有那么多时间学习那么多东西。其实这个不可能的设定，是在保证你有足够时间看电视、看美剧、刷微博、上网闲逛的基础之上的。只要把上述这些事情消费的时间减少一半，拿来持续学习，你就会发现学习效果是惊人的。

有一本书叫《奇特的一生》，介绍了一位名为柳比歇夫的科学家，这个兄弟生前发表了七十来部学术著作。其中有分散分析、生物分类学、昆虫学方面的经典著作；各种各样的论文和专著，他一共写了五百多印张。等于一万二千五百张打字稿。涉及内容包括：探讨地蚤的分类、科学史、农业、遗传学、植物保护、哲学、昆虫学、动物学、进化论、无神论。此外，他还写过回忆录，追忆许多科学家

柳比歇夫为什么会有如此多的时间干这么多事情？难道他不睡觉不娱乐每天工作 18 小时么？其实这是个错觉，很多宣称自己每天工作 15 小时的人，真正有效的工作时间可能不到一半。柳比歇夫把时间分为纯时间和毛时间，纯时间是要把工作中的任何间歇都要除去的，他这么描述：

“常常有人说，他们一天工作十四五个小时。这样的人可能是有的。可是拿纯时间来说，我一天干不了那么多。我做学术工作的时间，最高纪录是十一小时三十分。一般，我能有七八个小时的纯工作时间，我就心满意足了。我最高纪录的一个月是一九三七年七月，我一个月工作了三六一十六小时，每日平均纯工作时间是七小时。如果把纯时间折算成时间，应该增加百分之二十五到三十。我逐渐改进我的统计，最后形成了我现在使用的方法……”

“当然，每个人每天都要睡觉，都要吃饭。换句话说，每个人都有一定的时间用在标准活动上。工作经验表明，约有十二——十三小时毛时间可以用于非标准活动，诸如上班办公、学术工作、社会工作、娱乐，等等。”

这个人对时间有精准的把控，他会把自己每天的工作和用时记录成册，时间就像经过严格控制的沙漏一样，在他身边缓缓流过，他仿佛能感知时间的流失。这样的自控能力、学习和工作效率让人叹为观止，常被奉为天人！每当我偷懒时总是对自己说，孙子，你看看人家柳比歇夫子是怎么干的！当然，咱不是天才，有时候懒还是要偷的

所以，综上所述，无论你现在处于哪个境地，是手握优质资源还是即将被裁掉，其实都是自己之前的选择造成的。我们不能改变出生的国家、年代、家庭背景、运气，我们唯一能做的就是充分利用时间，按照自己的兴趣把自己训练成通才之上的专才，这个道理很多人懂得晚，做得晚，但是，就像写博客一样，写得慢不要紧，重要的是持续地写！ ■

为什么

Nikola Tesla

是迄今为止最伟大的
极客

作者 / The Oatmeal 译者 / 朱伟杰



他们修补那些不需修补的东西。极客们抛弃了他们身边的世界，因为他们忙于建造一个新的世界。



他们沉迷于这个世界，但是大部分时间里，他们忍受这个世界。一百年前，一个塞尔维亚裔美国发明家，Nikola Tesla 开始修补那些不需修补的东西。



在这个世界的大部分地方都是靠烛光来照明的时代，一个名叫交流电的系统被发明出来，并且它现在已经照亮了这个世界的每一个角落。



那么我们该感谢谁给我们带来了这个让人类进入第二次工业革命的发明呢，应该是



Nikola Tesla.



"但是我认为 Thomas Edison 才是电器时代之父。"

——每个人都这么认为

错, Tesla 才是。

每个人都认为 Thomas Edison 发明了电灯泡。其实**电灯泡**并不是他发明的,他仅仅是在 22 位前人的基础上,改进了电灯的理念。



Edison 只不过是解决了怎么卖灯泡的问题。

Tesla 早期的确是为 Edison 工作过一段时间。Edison 提出给 Tesla 一笔差不多相当于现在一百万美元作为酬劳来让 Tesla 解决他的直流发电机和直流马达上存在的问题。Tesla 修好了 Edison 的机器,但是当他找 Edison 要自己该得到的报酬时, Edison 对他一笑而过,并且对他说:

“Tesla, 你不懂我们美国人的幽默。”

Edison 是一个在极客世界混得很好的非极客的例子。

他相信他的发明的价值可以通过赚多少钱来衡量。他既不是数学家,也不是科学家,但是他可以雇佣别人来帮他发明和创造。

Edison 不是一个极客,他是一个 CEO。

Tesla 因发现神奇的东西而出名,但是他经常忘记
把它们记录下来。

而 Edison 的出名之处在于他的下属一旦有任何发现,他立马跑到专利局去进行登记。自从和 Edison 闹翻后, Tesla 继续研究他的交流电系统。

这个让 Edison 很不爽,因为这时候 Edison 正在到处售卖他的直流电系统。

Edison 的直流系统不能把电传输超过一英里,因此每一平方英里都需要一个发电厂。但是交流电使用比直流电更细的电线,有着更高的电压,并且可以传输得更远。

因此, Edison 做了什么呢?

住在 Edison 实验室附近的人慢慢发现他们的宠物都不见了。那是因为 Edison 付给学校的男孩子 25 美分用于收购一只活狗或者活猫。

然后,他公开展示这些猫和狗,

并且用 Tesla 的交流电来电击它们。

他的目的是为了公开污蔑 Tesla 的交流电,并且告诉人们在家用交流电是很危险的。

简而言之, Edison 仅仅是使用
暴力的先驱。

你是否听说过 Marconi?
他获得了诺贝尔物理学奖, 因为他发明了



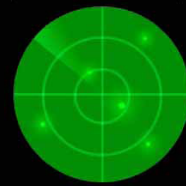
但是你是否又知道, 他所作的一切都是基于 Tesla 的工作? 在
Marconi 因为发送了第一条跨越大西洋无线电信息而闻名世
界时, Tesla 做了如下回应:

"Marconi 是个好人, 让他继续下去,

他使用了我 17 项专利呢。"

基本上, Tesla 是世界上脾气最好的发明家。

你又是是否听说过雷达？



这个神奇的技术可以让我们侦测到巡航导弹，也可以检测到在限速 45 的路上开到 85 的 SUV 车。

一个英国的名叫 Robert A. Watson-Watt 的人，在 1935 年，被誉为发明雷达的人。但是你能想到是谁在 1917 年就已经有了这个想法吗？他比 Watson-Watt 足足早了 18 年。

Nikola Tesla。



他把雷达的发明递交给了美国海军就在一战刚刚开始的时候，整个世界的菊花都被德国的 U 型潜艇爆的很惨。

很不幸

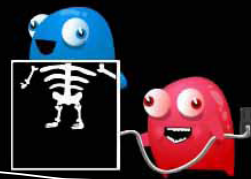
Thomas Edison 是当时美国海军研发中心的掌门人，他说服美国海军雷达技术在战争中没有实际的应用。

干得不错，Edison！

你个自以为是的蠢蛋。

我真希望一个纳粹的鱼雷射进你孙子的嘴巴里。

Wilhelm Rontgen 普遍地被人们
认为是 X 射线
的发现人。



➤ 但是一个有着小胡子的发明家早就发现了 X 射线，但是并没有获得半分荣誉。



还是 Nikola Tesla。



并且，在 X 射线刚被发现的时候，人们相信它能够治好失明，也相信它能够治好其他疾病。Tesla 警告人们 X 射线是危险的，

并且他拒绝将 X 射线用于医疗试验。但是 Edison 并不放过任何一个机会，他直接在人体上进行 X 射线试验。他的一个雇员，Clarence Dally，因为受到太多辐射而不得不截去双臂来保住性命。但是即使这样也没有用，他最终死于纵膈腔肿瘤。Dally 被认为是第一个死于放射性试验的人终于，Edison 有了自己原创性的发明！**除了杀死自己的助手，**

Edison 差点把自己弄瞎，因为他经常在自己眼前发射 X 射线。后来，当别人在他面前提起 X 射线时，他回答说：“别和我提 X 射线，我很怕它们。” --Thomas Edison, 1903

真是个操蛋的蠢货。

你是否好奇是谁在尼亚加拉瀑布建造了第一

个水力发电站，并且向整个世界证明
水力是个很好的能源？

他就是 Nikola Tesla。

是谁在冷冻技术发明近半个世纪前已经在进行试验了呢？也是 Tesla。是谁在 100 多年前拥有的专利后来用在了晶体管的开发中？

晶体管就是让信息时代变成可能的设备，是它让你能够做刷新 Facebook 页面，下载毛片等等操作。



还是 Tesla。

是谁第一个记录了从外太空来的无线电信号？（一不小心让他成为了射电天文学之父）

依然是 Tesla。

是谁发现了地球^{地球}的谐振频率呢?



也是 Tesla。

这个发现当时的科学家没法证实,直到 50 年后,科技的发展才跟上 Tesla 绝顶聪明的脑袋,这个发现才被证实。

是谁发明了人造地震机,当它启动的时候差点摧毁当时在纽约城的所有房子?

不用说,还是 Tesla

你是否听说过球形闪电^{球形闪电}? 这是一种以球形的形式存在的,运行很缓慢,并且会在地面上方几英尺的地方飘着的闪电。这是一种极其罕见的现象,并且直到今天也没有科学家能够在实验室里产生它。



不过,除了 Tesla,他在 19 世纪 90 年代就实现了。

是否想过是谁发明了遥控装置?

Tesla。

霓虹灯？也是 Tesla。

现代发电机呢？Tesla。无线通信呢？Tesla。你是否知道当你家里需要用电的时候，电可以向雨一样从地球的电离层下来，



然后通过无线的方式给你家里的电器充上电。

对，Tesla 发明了这项技术，

但是他并没有把他公诸于世，也许是因为他害怕那些没有灵感的蠢蛋盗取了他的专利。

毫无疑问，Tesla 是个天才。

他能说八门语言：塞尔维亚语，英语，捷克语，德语，法语，匈牙利语，意大利语还有拉丁语。

我们大部分人只会一种语言（可能都说不好）。

他可以记下整本书，并且能够随意背诵。

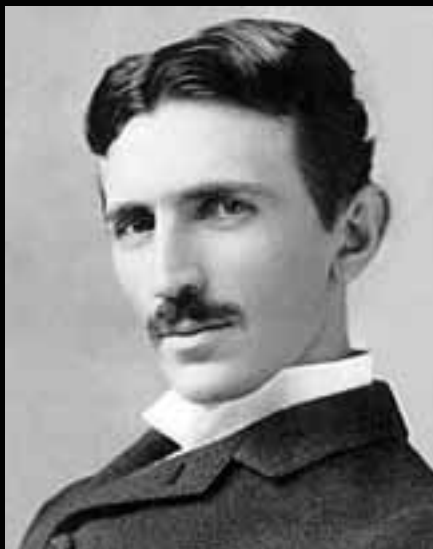
我们大部分人甚至连自己的密码都记不住。

他能够在大脑中设想出整个设备的样子, 然后
在不写下任何东西的情况下, 构造出这个设备。

我们大部分人只会花时间设想女人的
裸体以及油腻的三明治。

更让人惊奇的是, 他活了 86 岁, 而且始终是单身。

即使在 19 世纪九十年就身高 6'6" (200cm), 并且极度受女士
们的欢迎, 但是 Tesla 拒绝约会, 因为他相信那
会干扰他的工作。

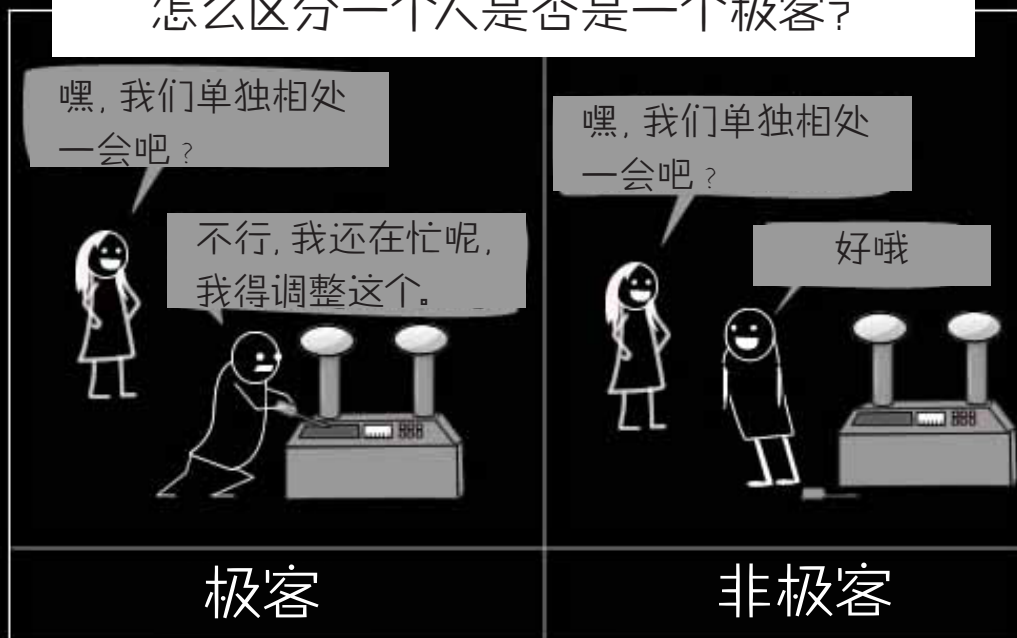


Tesla: 一个英俊潇洒的男人, 禁
欲了 86 年, 因为他一直在实验
室里忙于制造人造暴风雨。

PS: Thomas Edison 娶了一个年
仅 16 岁的女孩。

PPS: 这是我在这篇文章里最后
一次骂 Edison, 我发誓。

怎么区分一个人是否是一个极客？



有个这么聪明的脑袋以及这么多发明，Tesla 肯定非常富裕，并且很有名，对吧？

很不幸，事实恰恰相反。

Tesla 生于一个需要有着实际用途以及能够产生利润发明的时代。那个时代不需要射电天文，只需要电灯泡和面包烤箱。

Tesla 的贡献不是改良性的，

它们都是革命性的！

Tesla 留给世界的最后的礼物之一是纽约城附近的一座塔，它

可以给整个星球提供免费的无线能量。但是出钱建造这座塔的人，在听说了没有办法去管理别人对这座塔发出去的能量
的使用，因而也就没法赚钱的说法后，把它给关闭了。

人们的这种占有欲和贪婪困扰了 Tesla 的一生，并且这一生他都过得很苦恼。除此之外，Tesla 还忍受着一种不好的状态，
我们现在一般称为“**精神错乱**”。

Tesla 经常出现幻觉，因而导致他有一段时间无法区分现实和
自己的想象，这就是为什么他一直一个人在实验室里昼夜不
分地进行工作的原因。

他经常说，他唯一开心的时候就是关在自己的实验室里。
Tesla 悲惨而又孤独地死在纽约城的一个房间里。他一直以牛
奶和 nabisco 饼干为食，在他最后几次的访谈里，他说了一些
非常私人的事情：

我好几年一直都在喂鸽子，成百上千的鸽子。这中间有一只
鸽子，一只漂亮的鸽子，纯白色羽毛，

有着浅灰色的翼尖；那只很特别。她是只雌鸽子。无论到
哪儿我都能感觉到她。

无论我在哪儿，她都能找到我，当我需要她的时候，我只需
要想她，并且唤她，她就会飞到我身边。我们彼此心心相映。
我爱那只鸽子。

是的, 我爱她就像一个男人爱一个女人那样, 并且, 她也爱我。
当我看着她的时候, 我知道她想告诉我 -- 她要死了。然后, 当
我明白了她的想法之后, 她的眼睛里出来一道光——
一束很强的光。

仅仅以咸饼干为食, 并只和一只幻想出来的激
光鸽子聊天? !

这就是 Tesla 为人类所作
贡献的回报? !

亲爱的 Nikola Tesla,

我很难过,
我非常非常之难过。

你是个生不逢时的人;

一个阿基米德, 史蒂夫·沃兹尼亚克 (译者注: 苹果公司另
一个创始人),

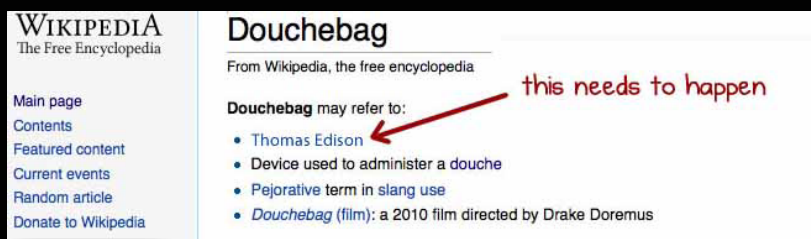
19 世纪的托尼·斯塔克 (译者注: 钢铁侠)。

你是迄今为止最伟大的极客。你生在一个人类比较下流的时代。

在英语里, 没有足够的名词能够放在 " 傻 " 后面来描述
Thomas Edison, 但是我还是要试一试:

傻逼, 傻 x, 傻帽, 傻蛋, 傻杂种……
(译者注: 原文的词太多了, 凑不够)

7 月 10 号是 *Nikola Tesla* 的诞辰, 我将很荣



幸地编辑他的 *wikipedia*。

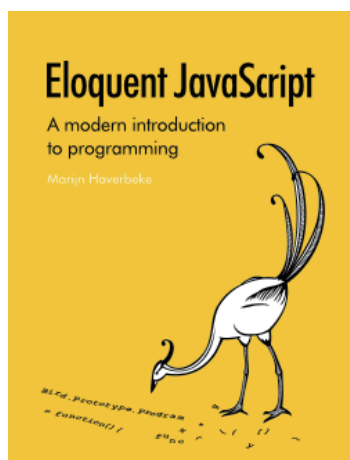
为了那些沉迷于修补那些不需要修补的东西
的人, 我用我自己的方式来
表达对你们的敬意:

谢谢你, *Nikola Tesla*。 ■

[-The Oatmeal](#)

善言的 JavaScript

——《善言的 JavaScript: 现代化编程指南》序言



樊虹剑（《Go 语言云动力》作者）“唯一”推荐的入门书。本书将教你如何学习 JavaScript 这个 Web 时代的 C 语言，以及计算机专业应该怎样思考问题。JavaScript 发明者 [Brendan Eich](#) 亦对此书赞誉有加。

个人电脑刚出现时，几乎都带着简单的编程语言，通常是 BASIC 的一种。此语言与操作电脑紧密相连，因此每个电脑用户，不管乐不乐意，都得尝试。可现如今电脑泛滥，平常用户却只给个老鼠点点。这对绝大多数人已是很不错了。但对我们这种有摆弄技术倾向者，日常电脑去除了编程，就成了种障碍。

还好，随着万维网的发展，每台装有现代化浏览器的电脑，碰巧就有 JavaScript 编程的环境。本着不用技术细节麻烦用户的精神，它藏得好好的，但一张网页就可以让它展现，并用它来作学习编程的平台。

这就是这本书想要做的。

不愤不启，不悱不发，举一隅不以三隅反，则不复也。

—— 孔子

除了讲解 JavaScript，本书还想要介绍基本的编程原则。你会发现，编程很难。大多时候，基础规则简单清楚。但基于这些基本规则的程序，常会复杂到产生自己的规则和自身的复杂度。正因如此，

本书采用 [Creative Commons Attribution 3.0 Unported](#) 署名 3.0 Unported (CC BY 3.0) 许可发布。加入图灵社区[开放书翻译计划](#)。

作者 /Marijn Haverbeke, Marijn Haverbeke 是一位通晓多种编程语言的专家。他在最近赢得了 JS1K (JavaScript demo in 1024 bytes) 竞赛, 并且是很多开源软件的作者。

编程极少会简单可期。正如这个领域的奠基者之一的高德纳所说的, 这是一门艺术。

要从本书获益良多, 单单读读是不够的。试着仔细看, 花些功夫做做练习, 直到真正搞懂了才继续。

程序员自创宇宙、自付全责。用计算机程序的方式创造出可以无比复杂的宇宙。

—— Joseph Weizenbaum, 《计算机威力与人类理性》

程序是多面一体。它是程序员键入的文字, 它指导电脑做要做的事, 它是计算机记忆体的数据, 也控制着对相同记忆体的操作。拿它类比我们熟悉的事物有些牵强, 但大致上程序类似一种机器。机械手表的齿轮巧夺天工地组合在一起, 如果表匠够好, 表就可以很多年保持准时。程序的组件也是像这样结合的, 如果程序员清楚自己的工作, 那程序运行就不会垮掉。

计算机被制造出来, 作为这些非物质机器的宿主。计算机自身只能做些蠢笨直接的事情。它们能如此有用, 是因为这些事情做得迅即无比。一个精巧地组合了这些简单操作的程序, 可以完成非常复杂的工作。

对某些人而言, 编程是种令人着迷的游戏。程序是思维的建筑物。不费一文就可以搭建起来, 没有丝毫重量, 能够轻易地成长在我们的按键之上。可如果我们玩过头, 那它的庞杂就会失去控制, 连造它的人都不能搞定了。这是编程的主要问题。这是当今软件容易垮掉、失效、搞得一团糟的原因。

程序干活时, 它是美丽的。编程的艺术是对复杂度的把握。程序

大物被制服了，简自繁中出。

现在，很多程序员认为，如果在他们的程序中只使用一块儿可以透彻理解的技术，就能把复杂度控制到最佳。他们编制了合格程序必须遵守的严格规定，他们中的狂热分子会把违反规定者斥责为“烂”程序员。

这是多么敌视编程丰富内涵的行为啊！要将其缩编到某种直接可期的事物，要屏蔽掉所有诡异与美丽的程序。编程技术的地平线是那么广阔，花样是那么纷繁，而且还有大片处女地。那里自然会散布着一些圈套和陷阱，误导不成熟的程序员进入各种可怕的误区，但这些只说明你一定要保持机警，谨慎前行。你会学到，新的挑战新的疆土一定会出现。不想探索的程序员一定会停滞，失去乐趣，不再希望编程（而去当了主管）。

据我所知，程序的正确性是其根本标准。效率、清晰和大小也都重要，但如何配置保持均衡则是一种权衡，是每个程序员必须自行作出的权衡。经验法则虽然有用，但也别怕把它打破。

最初，计算机刚诞生时，是没有编程语言的。程序看起来是这个样子的：

```
00110001 00000000 00000000
00110001 00000001 00000001
00110011 00000001 00000010
01010001 00001011 00000010
00100010 00000010 00001000
01000011 00000001 00000000
01000001 00000001 00000001
00010000 00000010 00000000
01100010 00000000 00000000
```

这个程序从一加到十再打印出结果 ($1 + 2 + \dots + 10 = 55$)。它能运行在非常简单的电脑上。要编程早期的计算机，就得把一大排的开关扳到正确的位置，或者在一长条的卡片上打满孔，再喂给计算机。可以想像要得出这种流程多么繁琐，也容易出错。连写个简单的程序都要求要很聪明很自律，复杂点儿的就别想啦。

那些 0 和 1 通常称为比特。当然，手动输入这些晦涩的比特图案，确实给程序员一种极强的无敌巫师的感觉。从工作满意度的角度来说，值得这么干。

每一行程序都是一条指令。写成文字是这样的：

1. 存数 0 到内存格 0
2. 存数 1 到内存格 1
3. 存内存格 1 的值得到内存格 2
4. 内存格 2 的值减数 11
5. 如果内存格 2 的值是数 0, 继续指令 9
6. 内存格 0 值加上内存格 1 的值
7. 内存格 1 的值加数 1
8. 继续指令 3
9. 输出内存格 0 的值

这比比特汤容易读些，但也会令人不舒服。使用名字代替指令和内存格数字，可能有些帮助：

置‘总数’为 0
置‘计数’为 1
[兜圈处]

置‘比较’为‘计数’
从‘比较’减 11
如‘比较’为 0, 从 [结束处] 继续
加‘计数’到‘总数’
加 1 到‘计数’
从 [兜圈处] 继续
[结束处]
输出‘总数’

至此不难看出程序是如何工作的，是吧？前两行给出两个内存格的开始值：“总数”用来加起程序的结果，“计数”一直跟踪着我们在用的数。使用“比较”的那几行可能是最奇怪的。程序想做的，是看“计数”是不是等于 11，这样才能决定是否该停下来。因为机器太过原始，它只能检查一个数是不是 0，然后据此做一个判断（跳转）。所以它是用标着“比较”的内存格，来计算“计数” - 11 的值然后根据此值做一个决策。下面的两行把“计数”的值加入结果，然后每次程序决定还没到 11 的时候，就给“计数”加 1。

此处用 JavaScript 写这个程序：

```
var total = 0, count = 1;
while (count <= 10) {
    total += count;
    count += 1;
}
print(total);
```

这使我们又有所进步。最主要的，是再也没必要指定程序来回跳动了。魔力字 **while** 做到了这点。它持续执行下面的几行，只要给它的条件没变：`count <= 10`，它是指“count 小于或等于 10”。显然，也没必要创造一个临时的值来和 0 比较。这是个笨笨的小细节，而编程语言的力量，就是替我们照顾这些笨笨小小的细节。

最后，如果我们刚好有 `range` 和 `sum` 这种便捷操作，也就是分别可以创建某个范围的数集，以及计算某个数集的总和，程序是可以这样写的：

```
print(sum(range(1, 10)));
```

这个故事的寓意是说，同样的程序可以写长或者写短，能够读懂或者不知所云。此程序的第一版最为晦涩，而最终版看着很像英语：打印（`print`）出总和（`sum`），范围（`range`）从 1 到 10。（稍后章节我们会看到如何能做出 `sum` 和 `range` 这种东西。）

优秀的编程语言帮助程序员用更宏观的方式表达自己。它藏起无味的细节，提供便捷的建筑模块（例如 `while` 结构），并且几乎总能让程序员自己铸造模块（例如 `sum` 和 `range` 操作）。

JavaScript 在如今做的大多是些万维网网页中各种机灵和讨厌的事情。[一些人](#)声称下一版的 JavaScript 会成为别的工种的重要语言。我不那么确信这会发生，但只要你在乎编程，JavaScript 就肯定是种值得学习的语言。即便最后你没有做什么 Web 编程，我在此书中展示给你的令人迷幻的程序，会一直跟着你，让你鬼迷心窍，影响你用其他语言写的程序。

总有些人说些 JavaScript “糟糕” 的事。这些事很多确实没错。我第一次需要用 JavaScript 写点什么的时候，几乎立刻就鄙视了这个语言。它可以全盘接收我键入的一切，但却用非我所愿的方式解读。这当然是因为我自个儿都不懂自己在干些什么，但这确实是一件大事：JavaScript 极端自由。这种设计的背后是为了让入门编程更容易。但实际上，它只是让你更难发现自己程序的问题，因为系统不会为你指出来。

但是，语言的灵活也是一种优势。它能完成那些更加僵直的语言无法完成的动作，这可以用来克服 JavaScript 的一些短处。在适当地学习和使用一段时间之后，我已经喜欢上这门语言了。

与名字的暗示相反，JavaScript 几乎与那个叫 Java 的编程语言无关。名字的类似出自市场考量，而非真知灼见。1995 年 Netscape 推出 JavaScript 时，Java 正在大造其市也广受欢迎。显然，某人觉得搭顺风车不赖。而我们现在只好忍着这个名字了。

和 JavaScript 相关的一个东西称为 ECMAScript。当 Netscape 之外的浏览器开始支持 JavaScript 或者貌似之物时，出现了一份文档精确地表述了此语言应该做什么。这份文档描述的语言，在被机构标准化之后，就被称为 ECMAScript 了。

ECMAScript 描述的是一种通用编程语言，根本没提如何与互联网浏览器结合。JavaScript 就是 ECMAScript，外加一些与互联网网页以及浏览器窗口打交道的工具。

还有些软件也使用 ECMAScript 文档所描述的语言。最重要的是，Flash 所用的 ActionScript，它基于 ECMAScript（但却没有精确遵循标准）。Flash 的系统用来在网页上添加可移动的物件，并制造大量噪声。如果你某天发现自己要学着做做 Flash 电影，了解些 JavaScript 没有害处。

JavaScript 还在发展。本书写完之后，ECMAScript5 面世了。它和此处描述的版本是兼容的，但添入了一些内置方法，这些功能我们会自己来写。最新一代的浏览器支持这个扩展版本的 JavaScript。截止到 2011 年，"ECMAScript harmony"，作为一种语言的激进扩展，仍在进行标准化。你不必过多担心这些新版本

译者 / 樊虹剑

网名 fango, 《Go 语言·云运算》一书作者。大叔级码农, 半瓶醋宅男。少年偶得志, 立命修身, 不与蝇苟。卖身软件开发十余年, 以程序自娱终日。闲暇时图谋调动文艺细胞, 怎奈 DLL 缺失, 异常为冷笑果。常感“独悦了不如众阅乐”, 偶发豆腐块文章, 愚己娱人。

图灵社区 ID: [fango](#)

会让学自此书的内容过时。原因之一, 它们都是现有语言的扩展, 所以此书所写的几乎所有内容都可以保持不变。

此书的绝大部分章节都带有大量代码。我个人经验是, 阅读和编写代码是学习编程的重要一环。不要只是一眼扫过这些示例, 要用心读并加以理解。刚开始会很慢也会很困惑, 但很快你就可以把握它了。练习也是一样。先别假设你明白了, 等真正写出可以干活的方案再说。

Web 的工作方式, 使我们总能查看别人写在网页中的 JavaScript 程序。这可以是学习某事如何完成的好方法。但是因为大部分 Web 程序员不是“职业”程序员, 或者是些认为 JavaScript 编程太没劲不值得好好学的人, 你这样查到的大量代码, 其质量是相当糟糕的。如果学习丑陋或者错误的代码, 那种丑乱就会混进你的代码。所以, 要小心自己在学谁。■

Markdown 作者谈

深入 Markdown



作者 / John Gruber

John Gruber，知名科技博客 [Daring Fireball](#) 的作者，发明了标记语言 Markdown，为编辑器 BBEdit 和博客软件 Movable Type 开发过多款插件。

“有时候一件事情的真相，不是来自于对它的思考，而是来自于对它的感觉。”

——Stanley Kubrick

要点 1

这儿有个问题：你最近一次听到足以改变你内心想法的论述是什么时候？不仅仅是关于那些你没怎么想过的问题，也包括在听到这个论述之前你原本相当确信的观念。

也就是说，你最近一次认识到自己在某个观念上彻底错了是什么时候？

如果你的回答是“从来没有”，或者是“很久以前”，那么这是否就意味着你总是正确的？

这里有我的一个故事。

一月份的时候，关于 Postel 法则在解析 XML 聚合信息流（比如

[RSS](#) 和 [Atom](#) 这样的格式) 的软件中的应用, 曾经有过一次争论。这次争论简而言之, 就是 [Postel 法则](#)认为: “自己做的时候要谨慎, 对别人的东西要宽容。” 而 XML 规范则明确说明: “一旦发现致命错误, 解析器就不能再继续正常的解析流程了 (比如, 绝不能继续将字符数据和文档结构描述信息还用标准方式传递给应用程序)。”

那么, 解析 XML 聚合信息流的软件该如何抉择呢? 是依照 [Postel 法则](#), 包容遇到的错误, 还是依照 XML 标准, 一遇到严重错误就终端处理?

[Brent Simmons](#) (Mac 系统上的主流新闻聚合阅读器 [NetNews Wire](#) 的开发者) 和 [Nick Bradbury](#) (Windows 系统上主流新闻聚合阅读器 [FeedDemon](#) 的开发者) 都认为在处理 Atom 和 XML 信息流时, 他们的软件应该采取严格的标准。很多牛人都同意他们的看法。

“客户端的标准应该严格” 的论点, 我觉得可以归结为以下几点:

1. 正确、规整的 XML 要好于有语法错误的 XML;
2. 要写出能生成正确、规整的 XML 的软件并不是那么难 (就像 [Tim Bray](#) 说的, “任何不能用规整的 XML 写聚合信息流的人都是无能的蠢货”);
3. 如果主流的 XML 信息流阅读软件都要求正确、规整的 XML 数据, 那么就会促使生成信息流的应用不再输出糟糕的 XML。

这三点我都同意, 因此, 我坚定地站在 “客户端解析时应该有严

格标准”的阵营中。

但之后我读了 Mark Pilgrim 的“思想实验”。Pilgrim 不仅是信息流解析应该宽松的倡议者，而且他公开了自己的所有代码，还写出了著名的开源信息流解析器 Universal Feed Parser。

Pilgrim 的观点，至少在我看来是这样的：如果你在做处理 XML 信息流的软件，而你的解析器非常严格，那么你的用户在遇到异常数据的时候就会遭殃。事实上，你的用户终究是会遇到异常数据的，这种情况发生的话，也许错在造成数据异常的人，但最终受害的却是你的用户，就因为客户端被强加了严格检查。

读过之后，我认真思考了他的思想实验，最终意识到我之前彻底错了，他是对的。（Simmons 也改变了他自己的想法，可以到[这里](#)和[这里](#)看后续的故事。）

Pilgrim 仅凭借他在“思想实验”中的论述，就让我认识到自己错了。但有意思的是，他说服我的时候，并没有反驳任何一个让我一开始站在“严格解析”阵营的事实。

这就是我要说的第一点：当我最终改变某个观点的时候，通常并不是因为我掌握的事实有误，而是因为我选错了事实。

要点 2

回头来看，所有博客软件背后的基本理念，其核心似乎都很简单。你是在管理一系列的文章，而不是有一系列网页的网站，博客软件会帮你将文章转换成网页。

博客软件对一般人的吸引力很容易理解。没有博客软件，他们就没法发布站点。

但博客软件对行家们——就是完全有能力用 HTML 手写出一个博客网站的专家们——的吸引力在哪儿呢？当然了，确实有[一些人坚持手写](#)。但是对大多数人，甚至包括世界上最[博学](#)和[著名](#)的 Web 工程师来说，都在用博客软件包（或者是[开发自己的](#)博客发布程序）。

答案是方便，灵活。博客软件免去了更新站点的过程中绝大多数的单调劳动。我必须承认——直到我 2002 年发布 Daring Fireball 的几个月前才明白这点。我能手写一整个网站出来而且还觉得写代码很容易，这样的事实让我忽视了在这个过程中有很多很多机械重复性的工作。

下面是我每次往 Daring Fireball 上发布一篇新文章实际上会发生的事：

1. 新文章出现在首页的顶部（同时从首页移去最旧的文章）；
2. 新文章有一个单独的永久页面；
3. 在我的 RSS 信息流中更新这篇文章；
4. 标题和新页面的链接被添加到我的[存档](#)页面；
5. 新页面的标题和链接作为“下一篇”被添加到上一篇文章中。

我只需要一个操作——发布一篇文章——然后 Movable Type 就会创建一个新文件，并且更新其他四项。这些任务都不难手动完成，其实很大程度上就是拷贝 - 粘贴的事儿。这些工作确实很无聊——

而且即使确实不难，我仍然很可能出错。我会犯错，因为我可能烦了，或是忘了操作其中哪一步。单调的重复性工作正是计算机所擅长，而不适合人类的。

另外，博客软件的这种封装也显得很自然。写一篇文章——或是对现有文章做出修改——感觉上就应该是一项任务。我要编辑的是文章，而不是文章所出现的页面。

因此，我要说的第二点是：不能仅仅因为一件事情不难，就认为它应该这样做。

下面我要试着将要点 1 和要点 2 结合起来，说说 Markdown 的事儿

让我们先回退一步。之前我说了，博客软件让你将站点当作文章的集合来管理，而不是页面的集合。

那么什么是一篇文章？

常见的说法是，一篇文章就是一个 HTML 代码段。不是完整的 HTML 文档，仅仅是一段 HTML 格式的文本——博客软件负责生成完整的 HTML（或者 XML）文档。你的博客模板里有关于文档结构的所有标签——`<html>`，`<head>`，`<body>`，等等——以及给你文章预留的位置。一旦发布，你的博客软件就把你的文章当作 HTML 代码段插入你的 HTML 模板中。

在我写 Daring Fireball 的第一年里，即从 2002 年 8 月到 2003 年 8 月的整整一年，我非常认可“文章就是 HTML 代码段”的说法。

我甚至从来没有考虑过别的可能性。我在 Daring Fireball 中写的每一篇文章，都是标准的 HTML 格式。（其实是 XHTML，但这种区别现在不重要。）

当然，除了一点之外，那就是 HTML 代码段不能被作为正确性验证，因为 HTML 是一种文件类型。你可以写出规整的 HTML 代码——确保关闭每一个标签，转换每一个 & 和 <> 符号——但你却不能将 HTML 代码段放到 [W3C HTML Validator](#) 上做检查，也不能用 BBEdit 的 HTML 语法检查工具。

我期望的工作流程是：

1. 在 BBEdit 中写文章，编辑，修订；
2. 搞定后，登录 MT 的 Web 界面，将文章粘贴进去，然后发布。

但我的实际工作流程看起来却是这样的：

1. 在 BBEdit 中写文章；
2. 在浏览器中预览；
3. 切换到 BBEdit 中修订；
4. 重复上面的步骤，直到搞定；
5. 登录 MT，粘贴文章，然后发布。

最终，我领悟到：这太愚蠢了。用电脑写文章的最大优势就是编辑起来比较直接。写、读、修订，都在同一个窗口中、同一种模式下完成。

用 HTML 格式直接写文章的理由——我用了很多年的理由——是 HTML 并不难。这点我仍然认同。HTML 很容易学会，而且一旦学会，就能马上用起来。专业的 Web 开发？那确实很难。但 HTML 的基本 tag 和规则——足够能让你用 HTML 代码直接撰写博客的 HTML 知识——是很简单的。

但像 [Lynx](#) 这样的纯文本浏览器也不直接给你显示 HTML 源代码，是有原因的。因为 HTML 本身就不是要作为一种可读格式存在的。用一种没有可读性的格式来写作，你不觉得很奇怪吗？突然间，我感到自己很荒唐。

要点 1 的应用：我没说写 HTML 源代码很难，而且仍然同意它很容易。我在论述完全不相关的另一个观点——我们所说的容易，是指文章标记上各种标签很容易，而不是阅读和排版容易。

要点 2 的应用：即使你仍然坚持认为，用 HTML 源代码写文章很容易，但难道这样不是很繁琐吗？要写 “AT&T” 而不是直接写 “AT&T”，这难道不无聊吗？（更别提要把 URL 的 & 符号进行编码。）

现在是 2004 年了！你的电脑难道不应该有能力判断你的文章哪里是分段，哪里是副标题吗？

别跟我说 Movable Type 的“转换分段符”功能可以实现这个效果。2.661 版 MT 的“转换分段符”功能将会把输入的如下两行：

```
<h2>This is a header.</h2>  
This is a paragraph
```

转换成：

```
<p><h2>This is a header.</h2></p>
<p>This is a paragraph.</p>
```

这效果真是够矬的。TypePad 显然能做到不在块级别的 HTML 标签外再嵌套一层没用的 `<p>` 标签——因此 MT3.0 大概也能做到——但这仍然无法改变其所生成 HTML 代码的冗余和难看。

为什么桌面版的博客编辑器需要提供“预览”模式呢？你在发送一封电子邮件之前，不需要“预览”——你写完邮件，回过头读一下，然后编辑修改，就行了。

实际上，我很喜欢写电子邮件。电子邮件是我最喜欢的写作媒介。在过去的 5 年里，我发送的邮件超过 16000 封。电子邮件的纯文本传统让我能清楚、准确地表达我自己，中途不会有别的东西来干扰我。

就这样，Markdown 诞生了。电子邮件风格的 Web 写作方式。

其他多数的“文本 -HTML”转换器都基于这样的假设，即 HTML 的标签很难用，所以他们用自己的标签来替代 HTML 的标签，结果是相较于 HTML 既不“更容易”也不“更可读”。而且，最终的结果是当用户遇到问题时，很难自如地翻阅手册，不得不再次捡起 HTML 来。

其他的转换器都意在替换 HTML，而 Markdown 则考虑的是别的东西。它希望能最有效地解决问题，既能让人在必要的时候很容易地使用原生的 HTML，也能让你在只需要写字的时候可以专心地写纯文本。

多数博客应用提供的一些标签快捷按钮——斜体、加粗、链接、图片、引用——并不是你一开始就需要考虑的。把插入这些标签

多数博客应用提供的一些标签快捷按钮——斜体、加粗、链接、图片、引用——并不是你一开始就需要考虑的。把插入这些标签做得这么容易，并不会让写作更容易，相反，会让你的文章结构更难被辨识。

但当你真的需要使用内嵌的 HTML 源码时——比如说，要用自定义的类属性来创建一个特殊样式的有序列表——你应该能直接就写 HTML。不用做字符转义，不用特殊的模式转换标记，就直接用标签好了。Markdown 让你能这样做，因为它是专门被设计为只作为 HTML 预处理器的。

（如果你确实需要将 Markdown 格式的文档转换为非 HTML 格式，只需要先将它转换为 HTML，然后使用现有的 HTML 转换器就行了。）

尽管我确实认为 HTML 很简单，但在一个特定领域它真的很棘手（如果不能说难的话）：用 HTML 标记语言来写关于 HTML 标记的东西实在很让人头痛。当你写有关代码的东西的时候，你应该只用关注示例代码本身——而不是每一个涉及 `<`，`&`，`<`；以及 `&`；符号的转义问题。

除了让在文档中添加内嵌 HTML 变得容易之外，Markdown 也让在代码段中添加示例 HTML 标签变得容易了。

感觉 vs 思考

纯文本在印刷上的局限性——单一的字体，单一的字号，没有斜体和加粗——跟打字机的局限性非常类似。设想有个不错的人给你买了个礼物：一个用原始打字机打印出来的一本经典小说的手

译者 / 张重骐

北邮人。学过设计，技术在学；前产品设计师，现任开发工程师。热爱美好，努力成为创造美好的人。

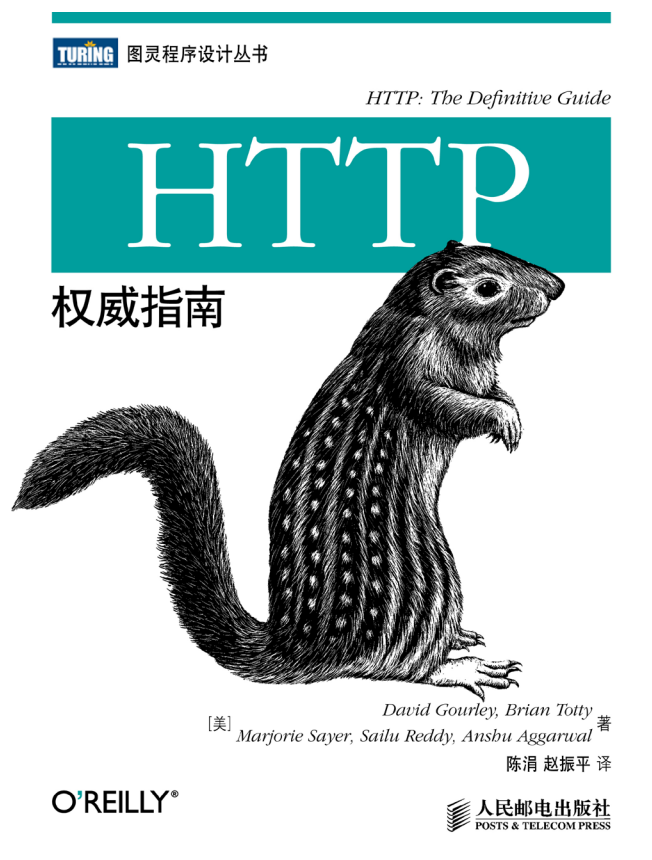
图灵社区 ID: [zhongqi](#)

稿，就比如说是 Fitzgerald 的《了不起的盖茨比》吧。你可以坐下来静静读这本手稿，从头到尾，这样获得的阅读体验，与你阅读一本精心排版和包装后的书的体验几乎是一样的。没错，那种感觉贯穿在打字机充满油污的、等宽 Courier 式的字体中，以及用下划线代替斜体的习惯，等等。——但文字依旧流畅，从纸面飞跃到你的心中，就像 Fitzgerald 所期望的那样。

我在文章开头引用的 Stanley Kubrick 的名言是我最喜欢的话之一。当你读或者写用 HTML 标签标记的文字时，这些标记在强迫你集中注意力去思考它们。而我希望 Markdown 格式的文本所传达的是同一种感觉。■

查看原文: [Dive Into Markdown](#)

书 榜



HTTP 权威指南

作者：David Gourley, Brian Totty

译者：陈涓，赵振平

书号：978-7-115-28148-7

定价：109.00 元

图灵社区推荐：32

HTTP (HyperText Transfer Protocol, 超文本传输协议) 是 Web 客户端与服务器交互文档和信息时所使用的协议, 是每个成功 Web 事务的幕后推手。众所周知, 我们每天访问公司内部网络、搜索绝版书籍、研究统计信息时所使用的浏览器的核心就是 HTTP。但 HTTP 的应用远不仅仅是浏览 Web 内容。由于 HTTP 既简单又普及, 很多其他网络应用程序也选择了它, 尤其是采用 SOAP 和 XML-RPC 这样的 Web 服务。

本书是 HTTP 协议及相关 Web 技术方面的权威著作, 主要内容包括: HTTP 方法、首部以及状态码; 优化代理和缓存的方法; 设计 Web 机器人和爬虫的策略; Cookies、认证以及安全 HTTP; 国际化及内容协商; 重定向及负载均衡策略。



30 天自制操作系统

作者：川合秀实

译者：周自恒，李黎明，曾祥江，张文旭

书号：978-7-115-28796-0

定价：99.00 元

图灵社区推荐：17

自己编写一个操作系统，是许多程序员的梦想。也许有人曾经挑战过，但因为太难而放弃了。其实你错了，你的失败并不是因为编写操作系统太难，而是因为没有人告诉你那其实是一件很简单的事。那么，你想不想再挑战一次呢？这是一本兼具趣味性、实用性与学习性的书籍。



Go 语言编程

作者：许式伟，吕桂华等

书号：978-7-115-29036-6

定价：49.00 元

图灵社区推荐：26

“《Go 语言编程》这本书应当说是作者多年编程经验的沉淀和反思。通过 Go 语言构建的‘七牛云存储平台’项目，对这些沉淀和反思进行了实践和验证，最终形成文字总结。Go 语言作为一个工程化的编程语言，正是需要这样以工程化思想为依托的图书来向世人展示其优雅之处……”

——邢星，Go 语言社区积极推动者，39 健康网技术部副总监



思考的乐趣：Matrix67 数学笔记

作者：顾森

书号：978-7-115-27586-8

定价：45.00 元

图灵社区推荐：17

“本书一大特色，是力图把道理说明白。作者总是用自己的语言来阐述数学结论产生的来龙去脉，在关键之处还不忘给出饱含激情的特别提醒。数学的美与数学的严谨是分不开的。数学的真趣在于思考……”

——张景中，中国科学院院士



我是设计师

5

作者：刘松，王蕾
书号：978-7-115-28785-4
定价：65 元
图灵社区推荐：7



60 秒学脑科学常识——《科学美国人》专栏文集

6

作者：Emily Anthes, Scientific American, Steve Mirsky
译者：蒋苹，王兴
书号：978-7-115-28243-9
定价：29.00 元
图灵社区推荐：3



短！微讯息时代写作的艺术

7

作者：Christopher Johnson
译者：赵燕飞
书号：978-7-115-28802-8
定价：39.00 元
图灵社区推荐：5



编程珠玑 (第2版)

8

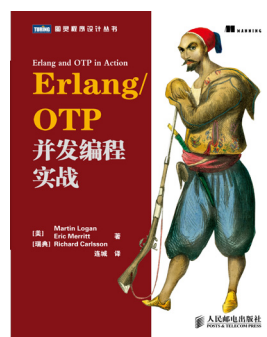
作者：Jon Bentley
译者：钱丽艳，黄倩
书号：978-7-115-17928-9
定价：39.00 元
图灵社区推荐：2



番茄工作法图解：简单易行的时间管理方法

9

作者：Staffan Noteberg
译者：大胖
书号：978-7-115-24669-1
定价：29.00 元
图灵社区推荐：7



Erlang/OTP 并发编程实战

10

作者：Martin Logan, Eric Merritt, Richard Carlsson
译者：连城
书号：978-7-115-28559-1
定价：79.00 元
图灵社区推荐：11

《Erlang/OTP 并发编程实战》



作者 / 连城

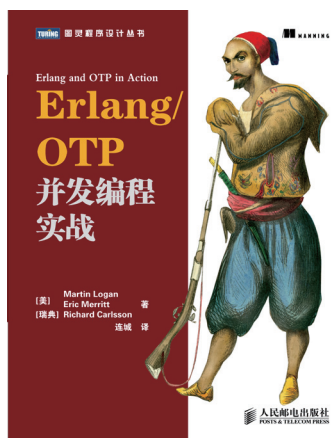
自由开发者，前百度资深工程师，《Erlang/OTP 并发编程实战》社区翻译项目组织者及主要译者。对分布式存储、分布式消息系统、程序语言设计实现抱有浓厚兴趣。个人博客：<http://blog.liancheng.info/>。

图灵社区 ID：[连城](#)

为了配合《Erlang/OTP 并发编程实战》一书的发布，译者从亚马逊上摘录并翻译了 Erlang and OTP in Action 的三篇书评，以飨读者。三篇之中好评两篇、差评一篇：倒不是偏心，实在是因为这本书在亚马逊上只有这么一篇差评……

身为本书的译者，同时也作为一名曾被 Erlang/OTP 官方文档绕得云里雾里的开发者，个人认为本书最大的优点有二：第一是清晰地拆解了 OTP 中最为繁琐的内容，如发布镜像、版本控制、部署；第二是在阐述 OTP 概念的同时给出了大量产品级（而非 Hello World 级）的代码实例。

下文中那篇差评的部分观点我也很赞同，这本书的确不太适合完全没有接触过 Erlang/OTP 的初学者。我想，这是“实战”这一定位决定的。虽然全书花费了 1/3 的篇幅来讲解 Erlang 语言的基础知识，这一部分却略显鸡肋。作为完整的语言参考，它还略显不足——正如作者在第二章开头处所说：“我们希望这些材料足以助你消化本书后续的内容；不过在将 Erlang 用到正式项目上之前，你最好还是备上一本更完整的 Erlang 编程指南。”尽管如此，2.15 小节仍然非常适合尚未习惯函数式编程的初学者阅读，这一小节很好地解释了尾递归，以及怎样用尾递归来代替常规命令式语言



本书侧重生产环境下的 Erlang 开发，主要讲解如何构建稳定、版本控制良好、可维护的产品级代码，凝聚了三位 Erlang 大师多年的实战经验。本书主要分为三大部分：第一部分讲解 Erlang 编程及 OTP 基础；第二部分讲解如何在实际开发中逐一添加 OTP 高级特性，从而完善应用，作者通过贯穿本书的主项目——加速 Web 访问的分布式缓存应用，深入浅出地阐明了实践中的各种技巧；第三部分讨论如何将代码与其他系统和用户集成，以及如何性能调优。

中的迭代结构（如“for”、“while”等）。

如果你对 Erlang 的基本语法已经相当熟悉，初步接触过 OTP，却在官方文档中屡撞南墙，那么这本书就是为你准备的了。OTP 中各种关键概念之间的关系、常用行为模式的最佳实践、发布和部署环节中各种错综复杂的配置，本书悉数给出了细致入微的阐述。

王婆卖瓜到此为止，下面来看看亚马逊上的读者们对本书的评价吧。

好一本菜鸟高手两相宜的 Erlang/OTP 开发大作！

- 作者：Mr. Bookish, Mild and Meek
- 评分：★★★★★
- 反馈：14 人中有 13 人认为这篇书评有用

这本书我刚看完一半，后半部分还只是草草浏览了一遍，但我已经爱上这本书了。

Erlang 跟 C#、Java 等面向对象 / 命令式语言大相径庭，当初在看 Joe Armstrong 的 *Programming Erlang: Software for a Concurrent World* 的时候我就觉得学习曲线应该会比较陡。那本书总体上还不错，但有些地方我还是没太看明白，即便是最初几章也同样存在这种情况。于是我又找来了 O'Reilly 的 *Erlang Programming*。这本稍微好点儿，但还是碰到了一些问题。80 年代末 90 年代初的时候我曾经读过计算机科学的研究生，对合一（Unification，数理逻辑和计算机科学中的一个概念——译者注）、演绎型数据库（Deductive

Database)、函数式编程、Lambda 演算、Gul Agha 的 Actor 并发编程以及分布式数据库等领域都有一定程度的了解。即便如此，我还是碰上了这么多困难。尤其是这两本书中和 OTP 相关的内容，实在是把我给弄晕了。直到后来，我开始从头拜读 Logan 等人的这本书。几位作者的那种以务实的口吻和实际的示例洗练地阐述问题的能力，着实令我叹服。这下好了，短短几天，Erlang 和 OTP 就都没问题了。第 2 章简明扼要地介绍了 Erlang 编程。讲解 OTP 及相关工具的章节更是没的说。相对于仅对 Erlang 抱有“学术兴趣”的读者，本书更适合于那些打算将 Erlang 应用到实际产品代码中去的开发者。

就本书再多说两句：Erlang/OTP 相关的工具很多，比如 Mnesia、附带 EXMPP 库的 XMPP 服务器 Ejabberd、Web 服务器 Mochiweb 和 YAWS（即所谓的 LYME 平台）等等；在把玩了 Erlang/OTP 及这些工具之后，我认为它们共同构成了一套非常棒的应用系统开发平台（依我之愚见，我敢说是最棒的），基于这一平台，开发者可以干净利落地开发出健壮的世界级应用系统。很多人对这一平台的性能、可伸缩性、并发支持、分布式支持、容错性以及集成之便利大加赞赏；但就个人而言，我最在乎的还是它鹤立鸡群的开发效率。毕竟只要赔上足够的复杂度和精力，在 Java EE、LAMP 以及 .NET 等其它平台上一样能够换取那些架构层面的特性。一个开发平台，只需不到 400 页便能把大部分问题讲得一清二楚，本书作者固然功不可没，但平台自身的简练与表达能力才是更为关键的因素。不信你针对 .NET 或者 Java EE 写本同类的书试试！！

那么，你可能就要问了，面对那么多高成本、高难度、高出错率的开发项目，开源 LYME 套件的优势又那么明显，为何历经二十多年都未能席卷全球呢？尤其是那些受资源和市场时机制约的独立开发者以及小型开发团队和软件企业，他们为何不采用 LYME 呢？很快你便会意识到，Erlang 的函数式特征、异乎寻常的编程

思想以及相对短缺的文献，造成了一道陡峭的入门门槛。我想这本言简意赅的书将可以帮助更多普通程序员迈过这道门槛，并最终让软件开发社区中的一大部分人享受到 LYME 的高效和便利。听上去我好像是被 Erlang 给洗了脑了似的？各位不妨亲自看看这本书，试试 Ejabberd、CouchDB 等 NoSQL 数据库，还有 YAWS 等等；再拿它们跟你所熟知的同类产品做个比较，眼见为实。

最后，如果你和我一样正进入 Erlang/OTP 领域，那么不妨再看看 Mitchell Hashimoto 的 Erlang 博客，上面有一系列和 OTP 相关的文章。Logan 等人的书中有些地方并未着重说明，而是让读者参考 Erlang 的在线文档，Mitchell 的这些文章是这部分内容的一个很好的补充。

涉猎广泛无出其右

- 作者：Richard J. Wagner
- 评分：★★★★☆
- 反馈：15 人中有 14 人认为这篇书评有用

一部着眼于产品级代码开发的 Erlang 大作。

全书分为三大部分：

1. Erlang 及 OTP 基础
2. 构建产品级 Erlang 应用
3. 集成和调优

第一部分介绍了 Erlang 和 OTP 框架相关的基础知识。内容编排

得当，非常适合初学者阅读。书中每隔几页便会穿插一副简图，用以诠释重要概念。OTP 相关的内容尤为详细，中高级用户也会有所斩获。这一部分本身就是一份不错的 Erlang 指南，但又远远超出了指南的范畴。

第二部分以一个完整的应用构建场景为基础展开。书中描述了一个饱受性能问题拖累的 Web 应用，正打算用 Erlang 搭建一套缓存以解燃眉之急。网络通信正是 Erlang 的强项，本书将带着读者一起自力更生，从头搭建一切必要的组件。众所周知，从掌握语法到掌握应用开发的最佳实践还有很长一段路要走，而这正是这一部分的价值所在。（本书既阐述了最佳实践也涵盖了语言的基础知识。）这一部分的几个章节惯用的手法都是先描述若干亟待实现的功能，再阐述如何用 Erlang 实现这些功能，最后一步步地给出完整实现。读起来就像是一组逐步深入的教程。

第三部分讲的是如何在 Erlang 应用中集成外部组件。示例中消息层面的集成借由 JSON 完成，更深层次的集成则在介绍 Erlang 语言层面的集成机制时进行了阐述。值得一提的是：Erlang-Java 接口 JInterface 独占了一个章节（相较之下，其余 Erlang 书籍在这一方面的介绍都较为有限）。

这本厚达 400 多页的大部头编排精良、内容翔实、讲解清晰，非常值得一读。在此我向各级 Erlang 开发人员推荐这本书。

另类看法

- 作者：NewLibertarian
- 评分：★☆☆☆☆

- 反馈：8 人中有 6 人认为这篇书评有用

最近，我花了几周的业余时间来学习 Erlang（着实令人印象深刻）。我算是个资深程序员（从业 20 多年了），并且有 5 年的专业 LISP 经验。我曾经读过并强烈推荐了由 Thompson 和 Cesarini 合著的 *Erlang Programming*。对于 Erlang 初学者来说，那是本极好的入门书籍。然而，书中却没有深入讲解 OTP。由于急于了解这一重要主题，我买来了 *Erlang and OTP in Action*。首先，如果你从来没有写过 Erlang 程序，那么它不适合作为你的第一本入门书籍。书中对 Erlang 的基础知识只做了概要介绍。读完跟 OTP 相关的第一个章节之后（费了牛劲儿了！），在那些金光灿灿的书评的反衬之下，我感到非常的恼火和失望。（我猜这些书评都是已经对 OTP 有所了解的人写的吧？）问题出在哪儿呢？我原以为作者会以言简意赅的方式阐述和 OTP 相关的概念并配上一两个简洁的示例，结果呢，作者却搬出了一堆冗长晦涩的 TCP 服务器之类的东西，把读者彻底给弄晕了。对于不熟悉 OTP 的人来说，这种又臭又长的示例只会让人抓不住重点。我当时就想——哇哦，看上去很难缠啊。为了尝试换个角度，我转而求助于 erlang.org 在线文档中的相关章节。我早该这么干的！！在线文档没几页就把问题讲清楚了，给出的一系列示例也同样简单明了。

也许对那些相对熟悉 OTP 的人来说这会是本好书。过段时间我应该还会再看看这本书，如果情况真是那样，我会再给高分的。■

[更多来自国外亚马逊网站读者的评价](#)

从失败中学习

——《30 天自制操作系统》译者自述



作者 / 周自恒

@ 馒头家的花卷，生于天津，毕业于上海外国语大学，现居上海。IT、编程爱好者，自称伪 Geek，初中时曾在 NOI（国家信息学奥赛）天津赛区获一等奖。大学学习语言，毕业后曾任 IT 系统咨询顾问，精通英语和日语，译著有《30 天自制操作系统》。图灵社区 ID: [Shyujikou](#)

《30 天自制操作系统》中文版终于和国内读者见面了，作为本书的 4 位译者之一，我负责翻译了本书约三分之二的內容。这是我参与翻译的第一本译著，感到激动之余也颇为紧张，因为我知道译者的水平对于一本译著质量的重要性——好的翻译可以成就一部作品，蹩脚的翻译也可以毁掉一部作品——正如当初引来无数臭鸡蛋的第一版《乔布斯传》中译本一样。

这本书的标题一出，立刻引起了很多反响。一些读者表示，以前看过类似“21 天学会 C 语言”之类的书，感到很坑爹，因此对这类形如“XX 天”标题的书往往怀有戒心，认为这多半只是噱头，很不靠谱。说实话，我没有看过其他以 XX 天命名的书，不过我也大概能理解这些读者的感受。在我看来，这本书的标题并不仅仅是一个噱头。类比一下，“30 天学会核物理”可能看起来很“假大空”，因为你没办法定义“学会”这个概念的边界。怎样算学会呢？每个读者会有各自不同的理解，因此往往有的人对结果表示满意，而另外一些人则正好相反。如果说“30 天自制微型反应堆”又怎么样呢？虽然你可能还是觉得太难了，但至少这个标题能给你一个具体的目标，30 天之后能做出一个反应堆来，不管它多么简陋，一定具备一个反应堆的关键性质。我认为这本书正是属于



这是一本兼具趣味性、实用性与学习性的书籍。作者从计算机的构造、汇编语言、C 语言开始解说，让你在实践中掌握算法。在这本书的指导下，从零编写所有代码，30 天后就可以制作出一个具有窗口系统的 32 位多任务操作系统。

后者，不管这个操作系统多简单，它是一个真正意义上的操作系统——更何况它还真不那么简单。正如封面上所说的，这个系统虽然很小，但却能实现图形界面、多任务等高级功能，这些都是实实在在的，跟着作者的思路和脚步，保证人人都做得出来的——即便你只是抄抄代码。

这本书的定位是零基础的读者，作者甚至找了初中生和高中生志愿者来试读这本书，以保证这本书的语言尽可能通俗易懂，可想而知作者把这本书的阅读门槛设定得有多么低。书中的语言风格非常轻松幽默，作为译者，我很喜欢这样的风格，因为可以把很多好玩的口语或者网络用语代入进去，让大家看起来更有意思，翻译起来也惊喜连连。从技术角度来看，这本书并没有过多地解释技术细节，作者认为对某些细节先有一个大概的认识就行，然后通过实践再加深理解，编写操作系统这件事的最终目的还是为了有趣、好玩，技术细节讲得太多自然就不好玩了。因此，想从这本书系统学习计算机原理、汇编语言、C 语言等知识是不现实的，但通过编写操作系统这个过程，你一定能够获得与系统学习这些知识完全不同的宝贵体验。

这本书的一大特色是“从失败中学习”，每一次我们为这个操作系统实现一些功能，一开始总是不顺利的，里面会有一些漏洞和缺陷，甚至根本不能工作。这些漏洞应该说都是作者刻意安排的（或者是作者以前开发 OSASK 系统时所亲身经历过的），作者花了很大篇幅来引导读者去寻找并发现这些漏洞，并从这些错误中学习如何想方设法让系统变得更加完善。我认为这种思路是非常有趣的，也是符合实际开发的过程的，一个东西怎么可能一下子就做得那么完美呢？用作者的意思来说，先体验了不完美的东西再努力去改进它，才能让你更有成就感呀！市面上的技术类书籍，

大部分都没有这种“试错”的过程，主要是因为“试错”本身需要精心的安排，而且需要占用大量的篇幅。我认为这本书所采用的写作手法是非常难得的，也是我认为值得向大家推荐这本书的主要理由之一。

如果你是一位高手，或者是在编写操作系统方面有所造诣的读者，你可能会觉得这本书的讲解并不是那么系统和有条理。的确，这本书并没有对每一个技术细节进行详细的讲解，就连汇编语言和 C 语言的语法也只是避重就轻地介绍，你甚至觉得这个操作系统在很多方面处理都很简陋（比如文件系统、内存管理、设备驱动等等），根本算不上一个实用的操作系统，甚至连作者自己都说：

“这本书无论在哪个方面都只有半瓶醋。”不过我们必须注意到，作者是在带领大家从零开始编写一个操作系统，而并不是用一个现成的 Linux 等内核为基础来做，后者也许才是目前所谓自制（定制？）操作系统的主流方式，但是这样真的能让你了解系统底层的真正机制吗？我看未必，一个 Linux 内核已经帮你搞定大部分底层工作了，你大可以把内核看作一个黑箱，只要往上面连接各种模块就 OK 了。只有从零开始，才能真正了解系统底层是如何运作的，而这些知识，对于你以后在其他内核基础上制作操作系统来说，也一定是非常有帮助的。如果你看了这本书觉得很坑爹，千万别忘了看一看最后一章中一个叫做“这也能叫自制操作系统？太坑爹了！”的专栏，作者早就预料到了大家的各种吐槽，在终点等着你们呢！（笑）看了这个专栏，你就会明白作者在有限的篇幅中，是如何谋划布局和取舍内容的，真心不容易，赞一个。

可能大家从目录上面也发现了，这本书涉及到了“日文显示”的相关知识。在这个部分如何翻译的问题上，我和出版社讨论了很多次，由于操作系统的编写都是在底层动刀子，可以说是牵一发

而动全身，我不想在这里破坏原书的结构，也不想擅自改动原来的代码，因此，在原汁原味保留原书文字的基础上，我补充了一些实现中文显示的相关内容，以体现日文显示和中文显示在实现上的异同。好在基本上只要替换字库和编码方式，就可以实现中文显示，甚至比日文实现起来还简单些（比如不用考虑不同的编码标准和半角片假名问题）。这部分补充的内容是我自己写的，但我在这方面也不是专家，不敢班门弄斧，补充的内容篇幅也并不多，如果有错误或者疏漏，也欢迎各位高手随时拍砖。

在这本书最后的编辑过程中，为了确认某些术语的译法，我还和作者川合秀实先生通过几封邮件（这说明作者在书里给大家留的那个邮箱地址是保证可用的哟！）。川合先生是个很认真很友好的人，他不像 **Matz**（松本行弘）那样在技术界鼎鼎有名，只能算是一个普通的技术者吧，因此能够写出这样一本好书，实在是非常难得的。最后也感谢其他 3 位译者，以及图灵公司的各位编辑的共同努力，使得这本书能够和中国的读者见面，希望大家都能从中获益。■

周自恒 2012 年 8 月 于上海



好文章要给更多人看，用你的笔传递最新 IT 好文。阅读量前 5 名的译人可任选图灵出版图书一本。

iTran 乐译 10 期

恭喜 Liszt、zhongqi、木子木、acid-free、周庆成五位童鞋拿走本期大奖

- [深入 Markdown](#)
- [程序员的时间换算表](#)
- [编织如编程](#)
- [理解 Git 工作流](#)
- [开源项目之憾](#)
- [互联网时代的 Word 5.1](#)
- [Markdown 的未来](#)
- [麻瓜的工具](#)
- [Martin Fowler 谈 CMS 系统中编辑 - 发布模块的分离](#)

专家 审读

“专家审读”是图灵编辑出版流程的重要环节，即邀请图灵社区会员以专业读者身份阅读全稿，并修正之前环节未查出的问题。事实证明，这个环节能有效地保证书籍质量。

专家审读 5 期

本期共有 11 位会员加入图灵技术专家审读小组，在我们的新书付梓前，以专业读者身份审读了《算法》中文版的纸稿，并给予了积极的反馈和评价，他们是（社区 ID）：

ola, Liszt, attachkover, 浮生小憩, 查无此人, 王腾超, acid-free, 小鱼 vs 婷婷, It, RalphGao, staryin

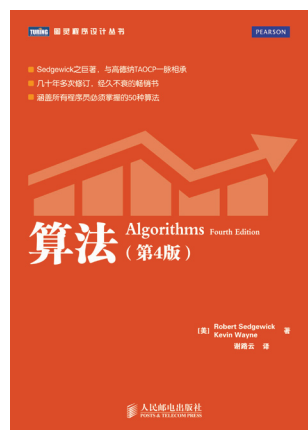
为了表达我们的谢意，审读结束后各位专业读者可任选一本自己喜欢的书（图灵出版）。

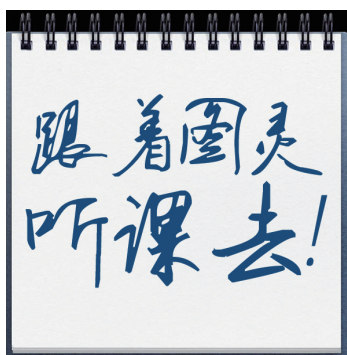
算法（第 4 版）

Sedgewick 之巨著，与高德纳 TAOCP 一脉相承

几十年多次修订，经久不衰的畅销书

涵盖所有程序员必须掌握的 50 种算法





图灵每个月都会举办或参与一些技术会议，“跟着图灵听课去”让您跟随图灵的脚步提前了解这些会议的亮点，在现场和图灵互动，并在会后得到第一手的会议报到和相关资源。

十月，跟着图灵听课去！

19-20 日 2012 MDCC 移动开发者大会：又一场移动开发者的盛会

面向创业者的平台，让更多技术创业项目浮出水面。为技术创业者对接天使和 VC 投资资源，并为他们提供各种有价值的服务。

20 日 PyCon China 2012：蟒人大聚会，京沪 Python 开发者盛宴

播放来自美国、欧洲、澳洲，以及来自北京、深圳等地的 Python 高手代表的视频演讲，认识全国、亚洲甚至全球的 Python 开发者，感受 Python 开发者的精神，领会 Python 开发的精髓。

25-27 日 QCon 杭州 2012 大会：全球企业开发大会·杭州站

知名网站案例分享。本主题由冯大辉主持。或许技术人员都想一窥那些大型网站背后的架构信息，“大规模”、“高性能”、“海量”之类的字眼的确够吸引技术人的注意，但是这些成功的网站发展历程中的经验教训才真的是值得我们关注的地方。

29 日 《持续交付》作者 Jez Humble 北京见面会：知无不言，言无不尽

Jolt 大奖图书《持续交付》作者 Jez Humble 亲临北京告诉你他所理解的持续交付。本次讲座将要讨论持续交付对于商业的价值。然后将要介绍关于持续交付相关的理论和实践，包括价值流图、部署管道、验收测试驱动开发、宕机发布、增量式开发。And most importantly, it's free.

[更多最新活动，请来社区活动版](#)

图灵社区 出品

出版人：武卫东

执行主编：杨帆

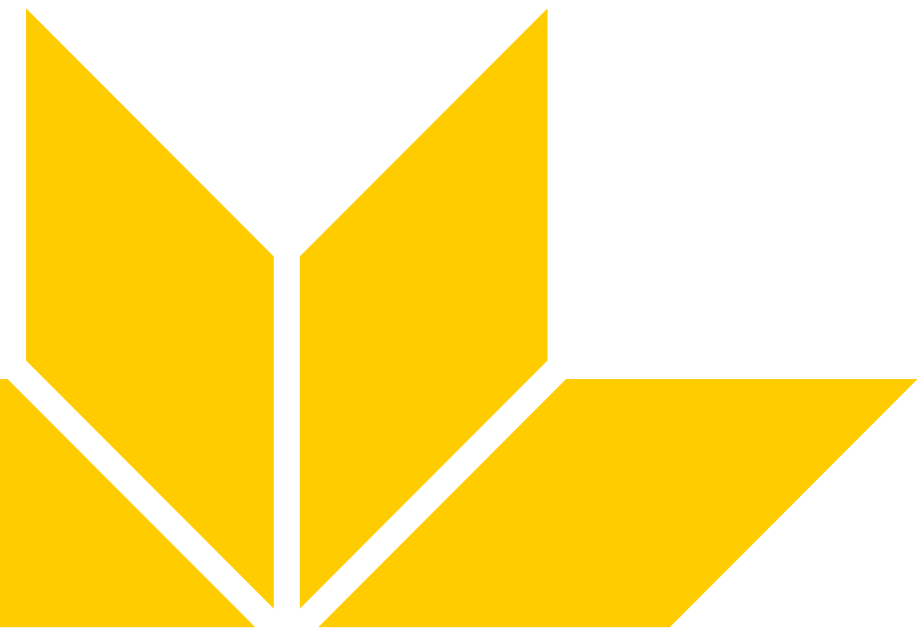
编辑：李盼

顾问：谢工、傅志红、姜丁坤

设计：大胖

本刊只用于行业交流，免费赠阅。

署名文章及插图版权归原作者所有。



地址：北京市朝阳区北苑路13号院领地OFFICE C座603室

电话：010-51095181

微博：weibo.com/ituring

Email: ebook@turingbook.com