TURING

码农

codeMaker()

2012/12/14

第 3 期

代码的未来：专访Ruby之父松本行弘

码农人物：创造的乐趣——郝培强

开源意味着什么？

大教堂，多怪异？

我们的开源项目

玩家也编程：“韦诺之战”的开放软件架构

用Markdown来写自由书籍

目录

专题：开源意味着什么？

人物

九卦

出版的未来

践行

鲜阅

书榜

妙评

社区动态

专题：开源意味着什么？

大教堂？多怪异

作者 /Mike Gancarz

Mike Gancarz 是美国佐治亚州亚特兰大市的一名应用开发顾问。他的团队使用 Linux、Unix 和 Java 工具，为金融服务行业开发出多个获奖的成像解决方案。作为 Unix 应用程序设计专家，他不遗余力地推广 Unix 已达二十多年。作为开发出 X Window System 的团队成员，Mike

“起初他们忽视你，而后嘲笑你，接着打压你，最后，就是你的胜利之日。”

——甘地

猫王、滚石，和拉丁语

Gancarz 还始创了一些至今仍应用在 Linux 上的最新窗口管理器中的可用性观念。Mike 曾经在 DEC 公司位于新罕布什尔州纳舒厄城的 Unix 工程开发项目组工作，主持了将 Unix 命令和程序移植到 64 位 Alpha 处理器的工作。他的首部著作 The Unix Philosophy（Digital Press, 1995）令数以万计的技术人员受益，他也是《Linux/Unix 设计思想》的作者。

译者 / 漆犇
《Linux/Unix 设计思想》译者。
图灵社区 ID: dayanday

2002 年 10 月 1 日，当滚石乐队发布专辑 *Forty Licks* 的时候，每个人都认为这张唱片应该能直接问鼎下一周唱片排行榜的冠军，但它却没能成功，只是屈居第二。滚石的这个故事证实了一个道理，你无法总是得到你想要的东西。这张集合了历史上最伟大摇滚乐队的热门曲目的精选集之所以未能问鼎，是因为广大听众情不自禁地爱上了另一张精选集，它集结的正是伟大歌手猫王 Elvis Aaron Presley 的热门曲目。这就是 2002 年 9 月 24 日发行的名为 *ELV1S 30 #1 Hits* 的专辑。

心存疑问的人们可能想知道为什么像猫王这样的老家伙会让滚石乐队无法得偿所愿呢？为什么他们就无法摆脱猫王的阴影呢？为什么是滚石败给了猫王，而不是猫王败给滚石？

答案很简单。滚石乐队的音乐是在“大教堂”里完成的，而猫王的歌曲却是借用了美国音乐“集市”中的种种元素。咦？在你觉得这个说法是如此怪异之前，让我来解释一下吧。也许你会明白为什么在教堂里哭泣的是滚石乐队而不是猫王。

从滚石乐手 Mick Jagger 和 Keith Richards 处于音乐生涯的早期巅峰开始，他们就在一起度过了无数个不眠之夜，创作出一首接一首让大家热血沸腾的热门单曲。他们的作品只有摇滚这一种风格，但我们还是很喜欢。Jagger 和 Richards 似乎做不了别的。他们只做了一件事情，可他们完成得很好。

然而，对于他们所具备的创作天赋来说，他们的作品只是一种“闭源”的专有做法。我们几乎很少听到滚石与任何第三方一起创作出来的歌曲。几十匹野马也无法将他们从心中的大教堂里拉出来，

去和别人一起开展任何开源的合作。他们可谓是自身 NIH 综合症的受害者。

然而，猫王的妙处就在于他这种大杂烩的集市化创作方式。他喜欢所有令人热血沸腾的音乐元素，他的歌曲风格包罗万象，从摇滚、蓝调、乡村、福音、民谣到流行曲调，一切都取决于他当时热爱的是哪一种。他的魅力源自他多变的风格。不管是在大屏幕上还是工作室的麦克风前，或是去参加著名的电视节目 Ed Sullivan Show，猫王都是激情四射，愿意表演任何节目。

猫王死了。愿他的灵魂不朽。

也许有人会说把滚石乐队、猫王与 Eric Raymond 的著作《大教堂与集市》放在一起来做类比有点儿牵强。毕竟，猫王最近距离接触的集市可能也就是麦克斯韦大道的孟菲斯跳蚤市场。而在大教堂，你最不可能找到的就是那位创作出 Goat’s Head Soup 专辑的人。因此，让我们来看看一个真实的大教堂，或者更确切地说，一种曾经流行于大教堂里的语言：拉丁语。

拉丁语曾经拥有过像今天英语一样的辉煌地位。随着罗马帝国逐步征服世界，从大约公元前 250 年直到 6 世纪，拉丁语的使用便一直在稳步地增长。在这段时间里，罗马天主教会宣称只能用拉丁语这一种语言撰写科学、哲学和宗教作品。因此从那时候起，如果想成为引人注目的牧师或学者，你就必须学会说拉丁语。然而，随着罗马帝国的逐渐衰落，外夷蛮族开始入侵，那些入侵者的好战本性使得他们并没有兴趣让自己显得智力超群，他们通常都是按自己的喜好来修改拉丁语。他们加入了自己民族关于杀戮、伤害和掠夺的词语，因为他们认为上帝也赋予了他们征服这门语言的权利。与此同时，祭司和学者却纷纷谴责这种玷污了他们心爱

的拉丁语的行径，并进一步向不断缩小的宗教人群宣传捍卫这门语言的重要性，直到它最后归隐在宁静的大教堂里，只有大教堂里面的人们才知道如何使用拉丁语。

另一方面，英语逐渐成为一种国际化语言，这并不是因为它的纯粹性或神圣性，而是因为它的适应性。它兼容并包，几乎所有的外来词或概念都能纳入日常的使用。迄今为止，它比世界上任何一门其他的语言都更具备与不同文化对接的能力，而且它做的事情就是连接。在这种我们称之为“生活”的集市中，集合了各种自然多样性，其中这里采用英语的国家是最多的，这主要归功于它连接万物的能力。

这也正是 OSS（开源软件）能够得以蓬勃发展的原因。任何人都能以任意的方式将它们与各种事物连接在一起。这源自它的开放性：开放的标准、开放的协议、开放的文件格式和开放的一切。OSS 世界里，没有什么事物能够隐藏。这种开放性使得软件开发人员能看到他们的前任是如何处理事情的。如果喜欢之前的做法，他们大可沿用，这很棒。他们因此也能找到有用的工具并节省宝贵时间。如果不喜欢，他们也大可以挥洒自如地查看过去的错误并将它们改进。

同时，那些在“大教堂”里面忙着开发“高级”软件的开发人员最终却只能眼睁睁地看着他们的市场被开源世界的开发人员占据。为什么呢？因为这种教堂式的软件开发人员只会为同在“大教堂”里的人们开发软件。不管这个“大教堂”是一个公司、社区还是国家，可迟早这些软件必须要与大教堂之外的软件进行交互，否则就只有死路一条。

我们为什么开源

到现在为止，你可能感受到了 OSS 和遵循 Unix 哲学的软件之间的一些共通之处。与那些大型“企工业级”（像一些人所认为的）闭源的专用软件相比，OSS 的许多作品看起来确实只是些“小儿科”。毕竟，这些作品通常都是由一些个人独自开发的，至少在它们的初期阶段是这样。因此，在早期这些作品只是一些典型的小项目，它们虽然缺乏利于市场营销的闪光点，却拥有扎扎实实的功能。不过没关系。请记住在 Unix 的世界里，小即是美。

Eric Raymond 称这些作品为“挠到程序员痒处”的软件。这似乎是许多 OSS 开发人员的共同目标，他们只是希望编写出他们喜欢的程序。因为日常工作的压力，他们往往是“背水一战”，所以没有太多时间去完成那些花里胡哨的功能。因此，他们通常都会忽略掉那些营销人员最爱的、华而不实的“闪光点”，他们注重的是只做好一件事情，这便意味着他们的作品是精干实用的应用程序，并确实能够解决个人的需求。在那些成功的案例中，这些最基本、最行之有效的解决方案可以激发人们的想象力。这正是热门 OSS 应用程序诞生的典型过程。

一旦点燃了想象力的火花，OSS 开发社区的其他成员就会开始行动，来为这个方案贡献自己的力量。起初，这个程序会吸引为数不多却很忠实的追随者。然后，该软件最终焕发蓬勃的生机，开始超越它最初的设想。在此期间，数十名程序员会把这个“第一个系统”演变成为一个更大更具包容性的“第二个系统”。在撰写本文时，Linux 本身就处于“第二个系统”的阶段。它正在享受自己的快乐时光。

显然，如果有人想将这种个人独自开发的 OSS 项目发扬光大，成为风靡一时的软件热潮，那他就必须尽早建立一个原型。仅仅对软件的伟大想法夸夸其谈是远远不够的。在开源的世界里，最受人们尊敬的是那些编写软件最初版本的原创者，以及在后来作出重大贡献的人士。

如果想要吸引尽可能多的用户，OSS 项目就必须提供适用于多平台的解决方案。因此好的 OSS 开发人员就会舍高效率而取可移植性。利用特定计算机上特定硬件功能优势的软件走不了太远，追随者充其量也不过是那些使用着相同硬件平台的人们。通过从封闭式专有架构走向开放的架构，程序就可以尽可能地出现在更多的平台上，从而最大限度地拓展自己的市场潜力。

在 OSS 开发人员发布那些供公众使用的软件时，他们利用了软件的杠杆效应。这种杠杆效应是双向的。首先，开发人员可以在自己的作品中使用其他 OSS 项目的代码。这样可以节省整体的开发时间，并有助于降低开发成本。其次，发布给全世界的优质 OSS 通常也会吸引到其他的开发人员，他们可以利用这个软件来完成自己的工作，其中的一些作品最终可能会找到一种回归到原始作品的方式。所谓“赠人玫瑰，手有余香”。

Perl 和 CVS

有一种做法能够增加 OSS 项目成功的几率：使用脚本来提高杠杆效应和可移植性。虽然软件世界里有很多核心的底层编码人员，可越来越多的人没有时间或无法尽情地为底层工作。通过为给一两个流行的脚本语言提供接口，这会显著地增强软件的可用性。

如果这些接口能足够流行的话，它们就会成为 OSS 社区里的重要

子社区。这些子社区致力于让人们体验并拓展这些接口。他们会定期召开用户组会议和研讨会，并且在网上邮件列表里发布文章，通常它会成为社区中心，基本上大多数围绕着这个流行接口而展开的活动都是由这个中心来召集的。

开源社区里有两个这样的例子，Perl 社区和 CVS（Concurrent Versions System，一种版本控制系统）社区。在 OSS 社区内，CVS 吸引了不少人。因为几乎所有的 CVS 命令都可以在脚本中使用，于是就出现了众多工具和插件，它们通过新颖的、有趣的方式与 CVS 命令相结合。CVS 的原作者肯定没有预料到人们能够开发出这些奇妙的用途。

CVS 特意避免了 CUI 的使用。只要传递的参数正确，每一条 CVS 命令都可以在命令行中运行。这催生出了更多的 CVS 插件。以基本的 CVS 命令如 checkout、update 和 commit 为基础，构建那些为提高 CVS 性能而开发的集成性工具。众多使用这些命令的插件通常都拥有 CUI，但基本的 CVS 命令却没有。这正是 CVS 得以蓬勃发展的主要原因之一。

作为开源的脚本工具，Perl 曾因晦涩难用而恶评如潮。它的功能比普通的 Unix 脚本如 Bourne 和 Korn Shell 还要强大，非常善于与其他软件交互。几乎所有的 Perl 程序都能充当过滤器。事实上，它的全称 Practical Extraction and Report Language（实用性摘录与报告语言）就恰恰说明了 Perl 的设计开发宗旨。当一个程序从某个地方提取、修改并报告信息（即它的“输出结果”）的时候，它充当的正是过滤器的角色。因为它的可扩展性，所以 Perl 的接受程度也在与日俱增。在这门极其不好用的语言中，黑客们（指的是这个词的褒义方面）仿佛找到了一片名副其实的乐土，他们尽情地添加代码库文件，以让它适应各种各样的情况。今天，你

可以找到能够处理各种事物的 Perl 模块，从复杂的数学内容、数据库开发到为万维网所用的 CGI 脚本。

这一点儿都不奇怪，可以说 Perl 是关于 Unix 哲学中那条次要准则“更坏就是更好”的绝佳例子——它完美地诠释了为什么一个不够好的应用（如 Perl）反而比那些足够好的应用（如 Smalltalk）有着更高的生存几率。几乎没人会否认 Perl 是一门古怪的、有着可怕语法的编程语言，而且谁也都没有想到 Perl 会大获成功。读到这里，你应该不难发现，OSS 往往遵循着 Unix 哲学。OSS 很适合这种早期 Unix 开发人员悉心呵护的“集市”式开发模式，而今天的 Linux 开发人员也都早已全面接受这一风格。不过有别于 Unix 哲学更加侧重设计方法的特点，开源社区中很多令人信服的作品却表明他们更注重软件的开发过程。用一句老话来形容，Unix 哲学与开源开发模式的结合有如天作之合，互相促进，相得益彰。

关于安全

在结束本章之前，我想讨论一个问题，这个问题自 2001 年 9 月 11 日美国世贸中心和五角大楼遭受恐怖分子袭击以来一直萦绕在每个人心目中，而且也是后现代生活和计算机世界的重要方面，那就是：安全。

自那一天开始，安全问题以这样或那样的方式驻留在每个人的脑海中。这种共通的安全意识也已经“植入”进计算机科学家的头脑。也许，Unix 哲学和开源以一种奇特的方式掌握着解决此问题的关键。

在信息安全和加密技术的世界里，有些人坚持认为想要提供最高级别的安全性，开发人员就必须隐藏一切，这是未雨绸缪的行为。

早期 Unix 开发人员的做法却与此大相径庭。他们并没有隐藏用来加密的密码数据和算法，而是选择对所有人公开所有的信息。任何人都可以自由地尝试破解 Unix 的加密算法，而且还能够得到所有相关的工具。在共享的精神作用下，这些找到破解 Unix 口令文件的人会把他们的解决方案提交给最初的开发者，这些开发者就可以在未来版本中纳入这些加密算法的解决方案。多年来，人们反复地尝试去破解 Unix 的密码机制，其中有一些尝试颇为成功，这使得 Unix 逐渐成长为更安全的系统。如果没有别人的帮助，单凭最初的开发人员可无法达到这样的效果。

从某种意义上说，安全是一种数字游戏，闭源系统的胜算非常小。这个世界每出现一个恶意黑客（坏家伙），对应也会有大概 100 倍甚至 1000 倍之多的白客。问题是，闭源的公司只负担得起雇用其中 10 个白客来看管他们的专有源代码。而与此同时，在开源的世界里，比坏人多 1000 倍的白客却都可以查看源代码来解决安全问题。闭源的公司，不管有多大，愿意花多少钱，能够雇用的白客数量也远远比不上开源社区里白客的数量。

开源软件与闭源专有软件的比较中蕴含着深刻的道理。有这么多人盯着 OSS 的安全机制，OSS 最终会证明这两个系统里面哪一个才是更为安全的系统。此外，随着个人、公司和国家越来越了解信息安全工作中对软件进行审核的重要性，很明显，唯一值得信赖的安全软件就是那款你拥有源代码的软件。

在某些时候，每个人都得自行决定是否要信任那些维护信息安全的软件生产商。对软件供应商，每个人都有着自己在道德和伦理上的批判标准。我们能够完全而充分地信任那个供应商吗？当你的财务状况、信誉、个体或国家安全受到威胁时，唯一可以接受的解决方案就是可以逐行验证代码的软件。

我们生活的世界里同时拥有开放和封闭的社会。封闭社会的运作模式就像是“大教堂”式的开发环境，源代码被层层封装，并且只能由少数拥有特权的人来决定哪些该留下、哪些该舍弃。在这样的社会里，新观念的传播极其缓慢。有效安全机制的开发节奏跟不上层出不穷的新威胁，新机制的开发周期可能需要数年。

这个世界的开放性社会看起来就像 OSS 社区一样杂乱无章，但里面也充满了各种天马行空的想象。开源社区的人们非常具有创造力，关于新技术尤其是安全技术的想法层出不穷。开放性社会能够迅速提出创新想法，确保在短时间内解除各种威胁。就像 Unix 密码算法的开发人员一样，让系统处于易受攻击的状态反而最终会成就最高级别的安全水准。■

相关阅读：有人负责，才有质量：写给在集市中迷失的一代

Linux/Unix 哲学的印证

——对话 Mike Gancarz

20 年前，X Window System 的开发者 Mike Gancarz 将 Unix 社区普遍认同的一些准则整理成 Unix Philosophy，让 GNU/Linux 等后来者能站到巨人的肩膀上。10 年前，这本书的第二版《Linux/Unix 设计思想》得以出版。如今，基于 Unix 思想体系的 Linux、iOS、Android 等已经成为全球范围内的主

图灵社区：首先，我想感谢您为我们创作了这样一部关于 Linux 和 Unix 哲学的大作。从您个人角度出发，是什么激励您写出了 *Unix Philosophy* 与 *Linux and the Unix Philosophy* 这两部书呢？

Mike: 谢谢你的夸奖。我很高兴自己的书能带给读者一些启发。当然，我其实也很享受创作这两本书的过程。

当我还在 DEC 公司 Ultrix（Ultrix 是 DEC 的 Unix 产品）工程组工作时，人们早已就 *The Unix Philosophy* 的内在理念讨论了很多年。在享用午餐时，我们会孜孜不倦地相互强调小程序是多么好用，可移植性 / 便携性是如何至关重要，等等。此外，我们不只流于讨论，还在技术生活中贯彻实施了这些准则。

其实，直到我转去 DEC 在佐治亚州阿尔法利塔市的 Ultrix 电话客户支持中心工作之前，我并未真正意识到我是多么地迷恋 Unix 哲学，并遵循其准则来构建系统。在那里，似乎也没有任何人意识到 Unix 的做法与其他系统是大相径庭的。因此，我为大家做了一些幻灯片演示，概要阐述了 Unix 哲学，并将 Unix 系统作为一种教学工具使用，每年还会给组里的新人分发几次教程。后来，在一次“DEC 用户协会（DECUS）”举办的研讨会上，当我做完演示后，有一位编辑提出我教授的这些知识可以作为一本书的创作素材。这便是 *The Unix Philosophy* 的缘起。

流系统，Mike 的远见卓识得到印证。图灵社区有幸邀请 Mike Gancerz 接受采访，谈谈《Linux/Unix 设计思想》、操作系统的前世今生、开源运动的前景等。

Linux and the Unix Philosophy 的情况却有些不同。出版商和我都意识到，尽管哲学是永恒的，但 *The Unix Philosophy* 中的一些素材却有些过时了。于是，我们才有了出第二版的想法。但问题是，作为一个操作系统，Unix 的声望在不断下降，而 Linux 却迅速地得到了人们的普遍认可。在贯彻 Unix 哲学的同时，该系统并未像 Unix 那般商业化。经过与编辑的多次讨论后，我们终于决定将这本书的书名变更为 *Linux and the Unix Philosophy*。

图灵社区：近年来，您为 Linux（包括内核）开发和应用做了哪些新工作?

Mike: 虽然我对 Linux 内核开发者怀有无限敬意，但我本人的关注点却一直是用户空间的应用。其实，我在 DEC 最喜爱的项目之一便是将整个 Unix 指令集移植到 64 位 DEC Alpha 芯片。打个比方说，如果将 Linux 比作是一辆车，那我其实对它的发动机并不感兴趣。我更关心它可以带我去到何方，以及，我是否会享受这些旅程。

近年来，围绕 Linux 的开发活动呈爆发性态势，我们已经很难跟上所有项目的进展情况。比如说，就算你只想专注于 Linux 用户界面这一块，但该领域涵盖的内容多而广，你最多也只可能掌握其中的一部分技能。所以，目前我仅在系统编译及工程项目部署这两方面花了很多时间。在这两个领域里，脚本工具的使用依然很频繁。虽然，现今的脚本工具都略有不同，但脚本仍然是一个很强大的工具。我很享受这种只编写一行代码，却能利用上别人撰写的数千行代码的喜悦感。

图灵社区：您总结概括的这些准则都基于 Unix 的历史和文化、黑客精神、自由软件运动以及您与 Unix / Linux 社区人群普遍采用的做法。素材来源如此之多，再加上 Unix 哲学文化的博大精深，您

是如何将这些精要准则提炼出来的呢？也就是怎么通过一个短语和句子来总结 Unix 哲学的精髓？

Mike: 通过在用户会议及其他论坛上的互动活动，我们大部分人开始体会到 Unix 有着一些核心理念。多年以来，Rob Pike、Kirk McKusick、Brad Cox、Jon “maddog” Hall 与其他人也都在谈论着小程序的重要性，即它们可以轻而易举地组合在一起去完成大规模的任务。我并未发明所谓的 Unix 准则，我只是把这些“大拿”们谈到过的理念如实记录了下来。

如何用一个短语或句子来总结 Unix 哲学呢？这确实是个难题，因为每一条准则单独存在时，根本就体现不出其优势。只有将它们视作一个整体，我们才会发现 Unix 哲学的博大精深之处。我想你们可能会认为，程序应该做且必须要做到的最重要的事情，就是与另一个程序交互。大多数 Unix 版本的开发都是基于这个思想，如果你能正确构建软件，它就永远能与其他软件进行交互，从而始终如一地遵循 Unix 哲学理念。

图灵社区：您提出了一个有趣的概念——“人类创造的三个系统”，您还指出 Linux 同时兼具第二个和第三个系统的特性。那么，目前 Linux 进化到了哪个阶段？它是否已经达到了“第三个系统”的标准呢？

Mike: 相比本书创作时，Linux 现在要更为接近“第三个系统”的状态。事实上，我认为如果你忽略掉某些 Linux 发行版那繁琐的安装机制，其实大多数 Linux 已经算得上是“第三个系统”了。当然，在 Linux 平台上，仍有一些子系统正处于演化进程中。例如，触摸用户界面与移动世界就是两大能说明 Linux 仍在不断发展的代表性领域。

图灵社区：您曾经预测，Unix 将成为计算机世界的首选操作系统。并且，Linux 作为 Unix 的变种，最有可能实现这一预测。十年后，Linux 在企业领域与桌面领域的发展各不相同。您如何看待此前这一预测?

Mike: 虽然我预测得没错，但是它并没有朝着我原本预期的方向发展。Unix 确实是计算机世界的首选操作系统，今天，它已成为除 Windows 之外的所有主流操作系统的根源。 而 Linux 还牢牢雄踞企业级服务器系统的江山。另外，叫好又叫座的 Android 操作系统也是基于 Linux 内核开发的。更不用说 iOS 是从 Mac OS X 演变而来的，要知道 Mac OS X 可是 Unix 的衍生品。单纯以平台数量而论，除掉桌面领域，从 Unix 派生出的操作系统几乎可以说是无处不在。

其中，真正具有讽刺意味的是，虽然 Linux 因传说中用户界面的不好用而未能主宰桌面领域，但 Linux 与 Unix 都在移动环境下得到了蓬勃发展，这主要是因为开发人员重新为该环境定制了一些好用的用户界面。所以 Android 和 iOS 用户总数其实要远超 Windows 的用户数量。尽管大多数用户不知道 Linux 或操作系统到底是什么，可实际情况就是如此。

图灵社区：关于“可移植性 / 便携性”，您曾经说过“使用最频繁的那台机器才是最强大的计算机”。那参照这个标准，iPad 与智能手机岂不就是最强大的计算机么？我们还想知道，目前的移动互联网热潮将会给 Linux 带来何种机遇与挑战？ GNU/Linux 在移动互联网的发展前景如何?

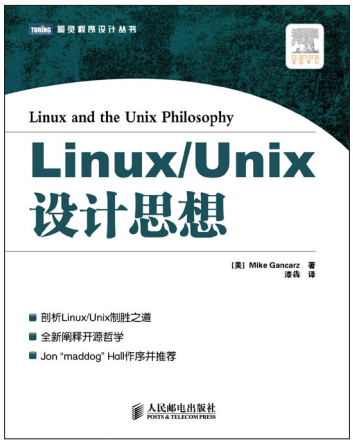
Mike: 是的，按这个标准来衡量，智能手机与平板电脑等移动设备早已赢得了这场战役。这是因为大多数人都是内容消费者，而不

是内容生产者。还记得“内容为王，傻瓜！”这一章节吗？移动设备为用户提供了最高便利，使得他们可以随时随地获取他们喜爱的内容。

此外，用户是内容消费者而不是生产者这一倾向使得移动互联网占据了一定优势，但对 Linux 本身而言影响不大。目前，带宽不足这一普遍问题限制了移动互联网的发展，但我相信它最终还是会解决的。另一个棘手问题就是，很多网页的设计是基于让多数用户连接到同一 Web 服务器的理念。如果同时有一百万用户希望能在一小时内下载某个视频，网站就很难满足所有人的需求。已经有人在着手解决此问题，他们将迫使 Linux 网络领域的人们去重新思考，到底该如何处理海量信息的缓存与传输。

Linux 在移动领域中的最大机会之一就是“云计算”。每个人都希望能在世界各地通过任何设备来获取数据。但与此同时，他们也希望系统能做好数据备份并保证其安全。作为计算基础设施的头号操作系统，Linux 已在“云计算”领域立稳了脚跟。而且，它还会在可预见的未来继续增加市场占有率。这恰恰印证了我们在书中提到的一个观点，计算机应做的最重要事情之一就是与其他计算机交互。可靠的网络通信是“云计算”的一个基本要求。

未来，在移动世界 Linux 要面对一些安全性方面的重大挑战。这同样是具有讽刺意味的，因为此前 Linux 一直被视为比其他同类产品更为安全的产品。其中，也有诸如 Android 这样基于 Linux 的开放系统。目前，开放状态对它们来说可谓是喜忧参半。也许，人们可以只通过认证机构来下载应用程序。亚马逊已经在朝着这个方向努力，更不用说苹果早就采用了这一做法，即该公司旗下的在线应用商店（App Store）。 不过，目前为止我仍然没看到真正的



Linux 和 GNU 项目的理念表面上是 Unix 哲学的下一个发展阶段，实际上它只是生生不息的 Unix 的强势回归。*The Unix Philosophy* 第一版中阐述的准则至今仍确切无误，甚至得到更多的佐证。开源除了可以让你清楚地了解到这些编程大师们创建系统的方式，还可以激励你去创建更快、更强大的系统。到目前为止，没有一本书同时介绍 Unix 和 Linux 的设计理念，本书将这两者有效地结合起来，保留了 *The Unix Philosophy* 中 Unix 方面的内容的同时，探讨了 Linux 和开源领域的新思想。

解决策略，因为每个人都有自己的衡量标准，拥有绝对安全的自由度是一件很难的事情。你需要放弃一些东西，非此即彼。而且，人们都对所谓的安全界限有着不同看法。

图灵社区：在这本书里，你用了很长的篇幅来抨击“强制性的用户界面（CUI）”，但今天 CUI 已成为计算机系统中的主导界面。你会如何解读此情况?

Mike: 在命令行 shell 接口中，CUI 指的是那些强制性获取用户注意力，并防止他们执行其他程序的接口。除非他们先完成此前任务才能脱离 CUI 的控制。然而，在图形化环境中，CUI 指的是模态（Modal）界面。例如，锁定用户的强制性对话框，此类对话框在用户给出响应之前是不会释放系统资源的。目前，大多数用户界面已经克服了这一壁垒，哪怕是在资源有限的移动环境也一样。这说明，曾经的哲学理念现已成为常识。

图灵社区：有很多中国程序员渴望能投身到开源活动中，但他们不知道自己是否能依靠此类软件过活。您能够根据自己的经验给他们一些建议吗?

Mike: 虽然我不想假装自己能理解中国程序员所面临的软件市场环境，但我想说的是，正经加入一家公司并为其开发某些应用程序或系统更靠谱。如果公司允许你将代码贡献给开源社区的话，这会是一件好事情。也就是说，程序员的首要任务是生存，他们和芸芸众生一样，先得解决好温饱问题才会有闲情逸致去考虑其他。所以，在满足生存需求的前提条件下程序员还能做出一些无私贡献是最理想的状况。

那么，如何能在为他人提供免费软件之余，还能养活自己呢？开

源社区的普遍做法是为该软件的用户和其他开发人员提供技术支持，这确实可行。当然，我也见证过很多尝试此运营模式，但最终还是失败了的公司。因此，我建议公司与个人都应该先专注于做好自己的本职工作。在项目成功之后，如果你想与大家分享自己的代码，并确实这么做了，那所有的人都会从中受益。

图灵社区：前两本书阐述了相同的哲学原理。如果您有机会就类似主题创作第三本书的话，您会对 *Linux/Unix Philosophy* 的九大基本准则及和十条小准则作修订与补充吗?

Mike: 哲学是永恒的。这些准则依然是真理，我顶多会修改一下其中的规范，让它们变得更为通用。比如，“使用脚本来提高软件杠杆效应”仍然是一个伟大的想法，但 shell 脚本正在被诸多新工具取代。不考虑用户环境，“强制性的用户界面”其实本质上是“模态用户界面”。所以我可能会将该准则改为“避免模态用户界面。”

此外，苹果目前取得的成功恰恰可以证明“哲学是永恒的”。但凡在苹果充分实践了 Unix/Linux 哲学的领域，它都独占鳌头。想要“可移植性 / 便携性”吗? 苹果将用户的整个音乐收藏库放进你的口袋、平板电脑及桌面机中。比起 iTunes 桌面界面，市场上有诸多高效的用户界面。然而，“可移植性 / 便携性”这一因素却屡获成功。那么，“只做好一件事”的小程序又是什么情况? 听起来 iPhone 应用程序不正是在贯彻这条准则吗? 什么是软件杠杆效应? 苹果并没有亲自去撰写运行在其设备上的软件，它只是充分调动其他开发人员来为其编写应用。

当我兴起就类似主题而编写第三本书的念头时，我会问自己一个问题，“Linux/Unix 哲学适用于现实生活的情况吗? ”想想在非计算机世界中去“尽快建立一个原型”的适用性。“让每一个程序都

记者 / 何逸勤
译者 / 漆犇

成为过滤器”在现实生活中可以转化为“从某处获取信息，整合成新资讯，然后发送给相应的消费者”。还有“寻求 90% 的解决方案”变成“不要为所有人建立一揽子解决方案，能处理大部分人的需求即可”。如果人们需要一本在总体生活层面上的 Unix 哲学书，我也许会沿袭这些思路去改写我的书。

图灵社区：在 *Linux and the Unix Philosophy* 的最后，您讨论了一些在当时（2003 年）就已采用了 Unix 哲学的“新技术”。那么，现在您能否为我们介绍一下最近十年间还有哪些技术符合 Unix 哲学吗?

Mike: 好吧，我其实可以与大家讨论一些诸如近场计算（near field computing）与分析，以及整个社交网络热潮中的技术。但是，“授人鱼不如授人以渔”，我宁愿选择后者，让我们从以下这些层面来思考吧。

随着新技术与生活的联系越来越紧密，我们可以仔细观察它们都是如何贯彻 Unix 哲学的。一项技术是否做到了“舍高效率而取可移植性”？公司是否达到了“只做好一件事”这个目标，还是他们试图去取悦所有人的需求？产品在价廉之余，是不是能够高效运行，即满足“更坏就是更好”这一准则？还有，你脑海里冒出的新想法是“第一个系统”吗？或者，你已经看到它有成为“第二个系统”的倾向？请注意，其中又有多少想法能明显转变成“第三个系统”呢？此外，有多少在十年前被人们视作奇思怪想的概念，到现在却被人们视作常识了呢？你是否又有这个鉴别能力，可以洞悉今天的“第二个系统”（比如，社交网络）中真正有价值的部分思想，也就是那些能将其转化为“第三个系统”的核心概念？ ■

玩家也编程

——“韦诺之战”的开放软件架构

作者 /Richard Shimooka
皇后大学“国防管理研究项目”的一位助理研究员，生活在加拿大安大略省金斯顿市。他也是“韦诺之战”项目的秘书和代理负责人。Richard 已经在社会团体的组织文化方面出版了若干本著作，研究的范围包括政府和各种开源项目。

作者 /David White
David 是“韦诺之战”的创始人和开发负责人。David 参与了许多开源的视频游戏项目，包括他和其他人一同创建的 Frogatto。David 是 Sabre Holdings 公司的一名性能调优工程师，该公司是旅行科技领域的领跑者。

编程往往被简单地看作一种解决问题的行为，开发者根据需求编码得到一个解决方案。对代码优美程度的判断一般来自于技术实现上的优雅或者效率，而这本书（《开源软件架构》）中的项目就是它们之中的杰出例子。除了计算，代码还对公众的生活产生了深远的影响。它能够激励人们参与并创造新的事物。但不幸的是，大家在参与各种项目时仍然会遇到很高的门槛。

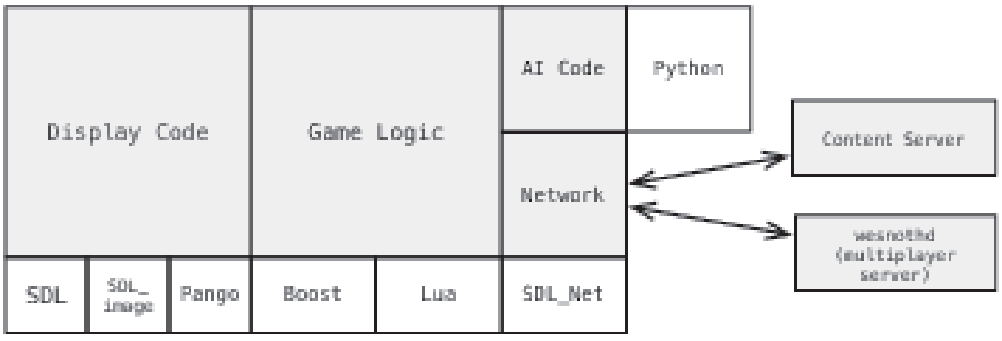
大多数编程语言都需要相当的技术专业知识才能使用，这对许多人来说遥不可及。此外，让所有人都能编程不仅从技术上是困难的，而且对于许多项目也没有必要。这样做并不一定能够得到简洁的代码或是聪明的解决方案。提高项目的参与程度需要开发者在项目和程序的设计时有远见，而这经常是和正常的编程习惯相违背的。再者，大多数项目的核心都是一组熟练的高水平专业程序员。他们并不需要外部资源的帮助。因此，项目的可参与性就变成了可有可无的东西，甚至从没有被考虑过。

我们的项目“韦诺之战”试图从源头上解决这个问题。它是一个基于 GPL2 许可证的在开源模式下开发的回合制奇幻战略游戏，它相当成功，截至这篇文章发表时已经被下载了超过四百万次。虽然这个数据很可观，但我们认为这个项目真正的出彩之处在于其开发模式凝聚了一大批能力水准各异的志愿者。

项目概况

韦诺之战的核心引擎是用 C++ 写的，现在总共约 20 万行代码。这只是游戏的核心引擎，不包含任何游戏内容，约占整个代码库的一半。我们的程序接受由一种叫做“韦诺标记预言”(WML) 的独特的数据语言所定义的游戏内容。游戏在发布时还包含约 25 万行 WML

代码。这个比例在项目中还在不断升高。随着项目的成熟，硬编码在 C++ 中的游戏内容已经越来越多地被重写为 WML 所定义的操作。下图给出了项目的大致结构。绿色的部分是韦诺之战的开发者们维护的，而白色的部分则是由外部的参与者们维护的。



总体而言，我们项目在大多数情况下都会尽量将依赖最小化以最大化应用程序的可移植性。这么做的间接好处是降低了程序的复杂度，并且开发者也无需再学习大量第三方 API 之间的琐碎区别。但同时，谨慎的使用一些第三方库也能达到相同的效果。例如，韦诺之战在处理视频、I/O 和事件中使用了 SDL 库。选择它是因为它使用方便且提供了一个跨平台的公共 I/O 接口。这使得韦诺之战能够被移植到许多平台，而无需为不同的平台专门编写 API。当然，这也是有代价的，那就是很难再利用一些平台专有的特性。韦诺之战还使用了 SDL 附带的一系列类库：

- SDL_Mixer：音频和声音
- SDL_Image：加载 PNG 等格式的图片
- SDL_Net：网络 I/O

此外，韦诺之战还使用一些其他的库：

- Boost：各种高级 C++ 特性
- Pango 和 Cairo：渲染各种字体
- zlib：压缩
- Python 和 Lua：脚本支持
- GNU gettext：国际化

在韦诺之战的引擎中，WML 对象——即用字符串表示的含有子节点的字典——的使用是无处不在的。WML 节点可以构造许多对象，而这些对象也可以被序列化为 WML 节点。引擎的一些部分将数据保存在 WML 字典的格式中并直接翻译它们，而不是将其解析为 C++ 的数据结构。

韦诺之战中有若干个重要的子系统，它们大多数都尽可能地减少外部的依赖。这种高隔离度的结构也有助于提高项目的可参与性。对某个部分感兴趣的人可以工作在其特定领域的代码中，他们提交的修改不会涉及到项目的其他部分。主要子系统包括：

- WML 的预处理器和解析器
- 抽象了底层库和系统调用的基础 I/O 模块：视频、音频和网络
- GUI 模块，包括按钮、下拉列表、菜单等控件的实现
- 显示模块，渲染游戏地图、单位、动画等等
- 人工智能 (AI) 模块
- 寻路模块，包含了许多处理六边型游戏地图的工具函数
- 地图生成模块，用于生成各种随机地图

同时，不同的模块控制着游戏流程中的不同部分：

- titlescreen 模块：控制游戏的主菜单画面的显示
- storyline 模块：显示游戏中的对话、场景切换等的序列
- lobby 模块：在多人游戏服务器上处理游戏的显示和创建
- "play game" 模块：控制整个游戏的主模块

主模块和显示模块是韦诺之战中最大的模块。它们的功能定义是最模糊的，因为它们的功能总是在变化，很难清晰地界定。因此，这两个模块曾多次有变成“上帝对象”的危险——变成一个无法定义其行为的庞然大物。所以，它们的代码会被定期审查并将任何可以独立为模块的部分分离出来。

项目中还有一些附加功能，但它们和主项目是分开的。这包括用于多人网络游戏的多人游戏服务器，以及一个内容服务器。用户可以将他们的进度等信息上传到公共内容服务器并相互分享。两者都是用 C++ 编写的。

韦诺标记语言

作为一个可扩展的游戏引擎，韦诺之战使用了一种简单的数据语言来保存和加载所有的游戏数据。最初我们准备使用 XML，但后来大家希望使用一种对非技术用户更友好且对可视化的数据的描述更加灵活的语言。因此，我们开发了自己的数据语言，叫做“韦诺标记语言”(WML)。我们在设计时就尽量降低了它的技术难度，希望即使连学习 Python 和 HTML 都感到困难的人也能看懂 WML 文件。韦诺之战的所有游戏数据都存储在 WML 中，包括单位的定义、战役、剧情、图形界面以及其他游戏逻辑方面的配置。

WML 和 XML 的基本性质相同，都有元素和属性，但是它不支持元素内的文本 (text)。WML 中的属性可以直接用一个字典式的字符串到字符串的映射表示，而程序逻辑会负责翻译这些属性。下面这个简单的例子是游戏中的“精灵战士”(Elvish Fighter) 这个单位的部分定义：

```
[unit_type]
    id=Elvish Fighter
    name= _ "Elvish Fighter"
    race=elf
    image="units/elves-wood/fighter.png"
    profile="portraits/elves/fighter.png"
    hitpoints=33
    movement_type=woodland
    movement=5
    experience=40
    level=1
    alignment=neutral
    advances_to=Elvish Captain,Elvish Hero
    cost=14
    usage=fighter
    {LESS_NIMBLE_ELF}
    [attack]
        name=sword
        description=_"sword"
        icon=attacks/sword-elven.png
        type=blade
        range=melee
        damage=5
        number=4
```

```
    [/attack]
[/unit_type]
```

由于国际化对于韦诺之战十分重要，所以 WML 直接支持国际化：含有下划线前缀的属性值是可翻译的。所有可翻译的字符串都会在解析时被 GNU `gettext` 转换为适当语言的字符串。

韦诺之战选择将引擎所需的所有主要的游戏数据记录在一份 WML 文档 (document) 中，而非多份文档。这样程序就只需要一个全局变量来表示这份文档，并且在游戏加载时所有内容都会被加载，例如，所有单位的定义都可以在 `units` 元素的所有 name 属性为 `unit_type` 的子元素中找到。

尽管所有数据都是被存储在一份 WML 文档中的，但将它们都保存在同一个文件中则是不明智的。韦诺之战的预处理器会在解析之前遍历所有 WML 文件。它允许一个文件包含另一个文件甚至另一个目录的内容。例如：

```
{gui/default/window/}
```

将会包含 `gui/default/window/` 下的所有 `.cfg` 文件。

由于 WML 可能会变得非常冗长，预处理器还允许定义宏来压缩文件的长度。例如，精灵战士的定义中调用了宏 `{LESS_NIMBLE_ELF}`，这个宏的作用是在特定条件下使某些精灵单位不再敏捷，例如当他们在森林中静止不动的时候。

```
#define LESS_NIMBLE_ELF
    [defense]
```

```
        forest=40
    [/defense]
#enddef
```

这个设计的优点在于游戏引擎无需了解 WML 文档是如何被分解到多个文件中的。如何将游戏数据分割安排到不同的文件和目录中去是 WML 文档的作者们的职责。

当引擎加载 WML 文档的时候，它会根据众多游戏设置定义一些预处理器所使用的符号。例如，韦诺之战中的战役的难度是可设置的，每个难度设置都定义了一个不同的预处理器符号。调节难度的一种常用手段是改变分配给对手的资源量（用“金”表示）。为了简化难度设置，我们定义了一个 WML 宏：

```
#define GOLD EASY_AMOUNT NORMAL_AMOUNT HARD_AMOUNT
  #ifdef EASY
    gold={EASY_AMOUNT}
  #endif
  #ifdef NORMAL
    gold={NORMAL_AMOUNT}
  #endif
  #ifdef HARD
    gold={HARD_AMOUNT}
  #endif
#enddef
```

例如，这个宏的参数可以是 `{GOLD 50 100 200}`。在计算机对手的定义中调用这个宏就可以根据难度级别定义它所拥有的金钱。

因为 WML 的处理是依赖于外部条件的，所以如果韦诺之战的引擎在执行期间发现 WML 文档所需的任何符号发生了改变的话，整个 WML 文档都会被重新加载并处理。例如，当用户启动游戏时，WML 文档就会被加载并得到已知的战役和其他游戏内容。但是，如果用户选择了某个特定的难度——比如简单——来开始某个战役，那么整个 WML 文档都会根据新定义的 EASY 符号被重新加载。

这种设计的方便之处在于单个文档就能够包含所有的游戏数据，并且可以通过各种符号配置该 WML 文档。但是作为一个成功的项目，韦诺之战中的内容越来越多，其中包括许多可下载的内容——这些内容最终会被插入到这棵文档树中，这意味着这份 WML 文档的大小会是若干 MB。它已经成为了韦诺之战的一个性能问题。在某些计算机上，加载文档就可能耗时一分钟。只要需要重新加载文档，游戏就会出现延迟。此外，它也消耗了大量的内存。我们也采取了一些应对措施，例如每个战役都会定义一个唯一的符号。因此可以用 `#ifdef` 来定义仅有这个战役才会用到的资源，这样只有在这个战役启动的时候这些资源才会被加载。

此外，韦诺之战使用了一套缓存系统来保存，有一组定义好的键对应被处理过的 WML 文档。这个缓存系统会检查所有 WML 文件的时间戳，如果任意一个文件发生了改变，就重新生成文档。

韦诺之战的多人游戏

我们用尽可能简单的方式实现了韦诺之战的多人游戏。服务器会尽力阻止恶意的攻击，但并没有在预防作弊上下功夫。韦诺之战的一局游戏中所进行的任何动作——移动单位、攻击敌人、招募单位等等——都能被保存为一个 WML 节点。例如，移动一个单位的一条命令所保存得到的 WML 节点是这样的：

```
[move]
    x="11,11,10,9,8,7"
    y="6,7,7,8,8,9"
[/move]
```

它保存的是单位在玩家的命令下的行进路线。服务器在游戏中会执行接收到的任意类似的 WML 命令。这种设计的实用之处在于，只要保存了游戏的初始状态和随后的所有命令就等于保存了一场完整的游戏录像。重放游戏既能帮助玩家观察其他玩家的玩法，也能帮助我们调试一些错误报告。

我们希望社区的多人游戏的主旋律是友好、休闲的气氛。与其在技术上和那些整天钻研如何破解反作弊系统的反社会分子做斗争，韦诺之战基本没有反作弊。在分析了其他多人游戏之后我们认为，

竞争性的排名系统是这种反社会行为的根源，而在服务器上刻意忽略这种功能则将大大降低人们的作弊动机。此外，社区的管理员也在积极地鼓励参与游戏的玩家互相建立良好的关系。这大大促进了友谊第一比赛第二的社区氛围。这些努力是非常成功的，到目前为止那些尝试恶意攻击游戏的人都被社区孤立了。

韦诺之战的多人游戏的实现是一个典型的 C/S 架构。服务端程序 wesnothd 会接受客户端的连接，并向客户端发送当前游戏的一份列表。韦诺之战会向玩家显示一个游戏大厅，玩家在其中可以选择加入他人的游戏或是创建一个新的游戏并等待他人的加入。当所有玩家都进入游戏且游戏开始之后，韦诺之战的客户端会根据玩家的行动生成 WML 命令。这些命令会被发送到服务器，而服务器则会将它们转发到游戏中的所有其他客户端。服务器的任务仅仅是转发而已。重放系统用于在其他客户端上执行 WML 命令。由于韦诺之战是一个回合制游戏，所有网络通信使用的协议都是 TCP/IP。

服务器还支持观看者观看游戏。观看者可以加入一个进行中的游戏，服务器会将游戏的初始状态和游戏的命令历史全部发送给观看者。这使得新的观看者可以赶上游戏的进度。观看者可以看到游戏的进程，但不会立即到达游戏的当前状态——命令历史可以快进，但仍然需要时间。另一种方法是令某个客户端用 WML 描述一个游戏当前状态的快照并将它发送给新的观看者，但随着观看者的增多这种方法会增加游戏客户端的负担，而且只需让多个观看者同时加入一个游戏就可以达到“拒绝服务”攻击的效果。

当然，由于韦诺之战的客户端相互之间不会共享任何游戏状态，只会发送命令，所以游戏规则的一致性很重要。游戏服务器会根据版本将游戏分区，只有客户端版本相同的玩家才能在一起游戏。

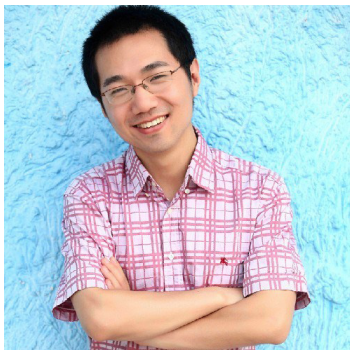
如果有人的客户端版本和其他玩家不同步，所有玩家都会立即收到警告。这也是一种有效的反作弊手段。虽然玩家想通过修改客户端来作弊并不难，但任何版本差异都会被立即通报给所有玩家并由玩家们自己来解决这个问题。

总结

我们认为韦诺之战项目的优秀之处在于所有人都能参与编码。为了实现这一目标，项目经常在代码的优雅方面作出妥协。值得一提的是虽然项目中的许多优秀程序员在见到 WML 的低效语法时都会皱眉头，但这种妥协正是项目的成功之源。今天，韦诺之战的最大财富就是用户创建的数百个战役和时代，而这些用户大都只有很少甚至没有任何编程经验。此外，它也激励了许多人将编程作为一种职业，而将这个项目作为一种学习工具。这种成就是许多其他项目所无法比拟的。

读者从韦诺之战项目的工作中能够学到的重要一课应该是如何去帮助那些不熟练的程序员。要认识到参与者在实际编码和磨练技能时所遇到的困难并不容易。例如，某些人可能想为项目作出贡献，但却不具备任何编程技能。类似 emacs 或 vim 这样的专业编辑器有着显著的学习曲线，只会让人们灰心丧气。因此 WML 的设计初衷就是任何人都能用一个简单的文本编辑器打开 WML 文件并作出贡献。

但是，增强代码库的可参与性并不简单。并没有什么硬性和便捷的规则能够降低代码库的门槛。相反，它需要照顾到方方面面的平衡，否则可能会对社区产生负面的后果，这在项目对依赖的处理上表现得很明显。在某些情况下，依赖会提高项目参与的门槛，但在另一些情况下也可能使参与项目更容易。具体情况需要具体分析。



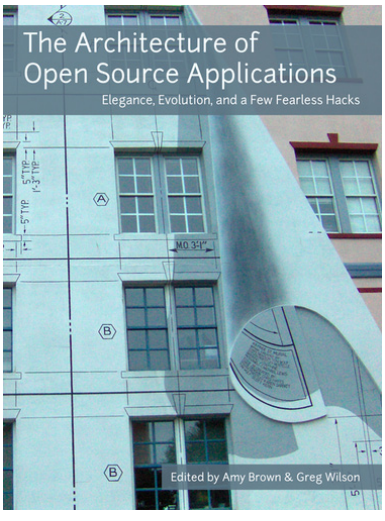
译者 / 谢路云

@一盆花，华中科技大学毕业。Python 程序员，《算法》第四版的译者，现就职于 AppAnnie。自大学起爱看《科幻世界》杂志的译文版，从此发现翻译也是一件特别美妙的事情。他希望把这种快乐也带给其他人。

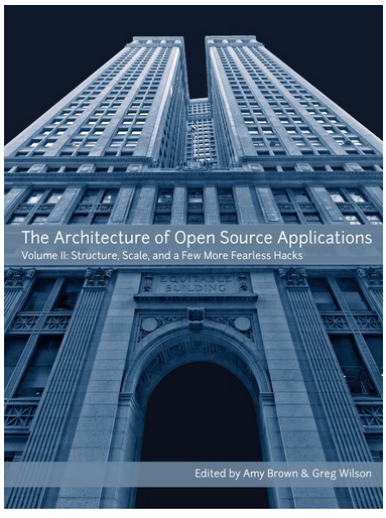
图灵社区 ID：小谢

我们也不应该夸大韦诺之战所取得的成功。这个项目拥有一些其他项目无法轻易复制的优点。降低编写代码的难度并吸引广泛的公众参与的成果也部分来自于项目的规则。作为一个开源项目，韦诺之战在这方面有一定的优势。在法律上，GNU 许可证允许任何人打开代码文件，学习其中的原理并进行修改。这种文化鼓励所有人去实验、学习并分享，但并不一定适合于其他项目。尽管如此，我们希望这个项目中的闪光点帮助了所有开发者并让他们体会到了编程的美妙之处。■

查看原文：the Battle of Wesnoth



《开源软件架构：卷 1》



《开源软件架构：卷 2》

本文选自《开源软件架构》，欢迎加入该翻译计划！在建筑师的培养过程中，他们要看上千座建筑，并研究大师们对这些建筑的评议。但在软件行业，开发者却缺乏这样的经验，使得他们无法利用彼此的成功，只能重复彼此的错误。这两本书恰恰提供了一个了解历史的机会。48 位开源软件作者站出来，解说他们的软件是如何架构的，以及为什么这么做。更多开放书翻译计划，所涉书籍采用 Creative Commons Attribution 3.0 Unported 署名 3.0 Unported (CC BY 3.0) 许可发布。

图灵访谈：我们的开源项目

记者 / 李盼

本文采编自图灵访谈：我们的开源项目系列。受访者分别为：

- 姜宁 @ 姜宁 willem，Apache Member，专职参与 Apache 旗下多个开源项目 Camel, CXF, ServiceMix, ActiveMQ 开发攻城师。
- Freeman Fang @ Freeman 小屋，Apache Committer/PMC member of CXF, Servicemix c, and Karaf, Committer of Camel, Felix, OPS4J Pax Web, Progress/FuseSource Principal Software Engineer.
- Nutz @ zozoh 和 @ 胖五 86，开源项目 Nutz 的发起者。
- 庄表伟 @ 庄表伟，盛大创新院高级研究员。博客：http://www.zhuangbiaowei.com/blog/。
- 魏永明 @ 飞漫魏永明，他主持的 MiniGUI 项目，是国内最具代表性的几大自由软件项目之一，目前已成为支持多种嵌入式操作系统的图形中间件产品，著有《Linux 实用教程》等一批 Linux 技术著作。

你认为中国开源社区的最大问题是什么？

姜宁：我觉得最大的问题是我们仍然大多停留在“用”的状态，而且是悄悄地用，这样一来社区的氛围就不是很浓。国外在开源方面做得比较好，很多牛人他们并不吝啬于自己的好点子，他们乐于分享和讨论。而且大多数这样的人是不爱说空话的，大家要是觉得好，他就会写出来，大家来用，他收到的反馈也会很迅速及时。这和社区的健康发展是分不开的，我们（国内的工程师）人数很多，但是真正能持续做的人很少。 还有就是，**社区一开始可能需要像黑客这样以兴趣为导向，后来会提供一个商业环境来帮助你，这样才是一个成熟的链条。**我认为这有可能是国内比较缺乏的。这再一次说明国内的环境不太好，如果大家只是使用，而不去反哺社区，有些人甚至自己做出了 patch 也不愿意分享出来，这样就无法完成一个循环的全过程，社区也就无法成长起来。

Nutz: 我其实觉得中国开源社区情况挺好的，因为它很残酷。没有什么好的开源项目，大家都奔着挣钱去了。

但是往往在这样的环境下生长出的开源项目就会更加坚强。这就像北方的粮食要比南方的好吃，就是这个道理。所以从另一个角度说，中国的开源环境是世界上独一无二的。

庄表伟：（我们）从来不缺乏好的想法，缺少的是能持续做下去的一种精神。如果要回过头反省自己的话，那就是我自己的那些所谓的好的想法，我自己也不够坚持。如果我自己够坚持，可以埋头苦干一场，说不定最后也就做成了。像我微博和博客里经常提到的，其实有的时候这是一种孤独感，这种孤独感我体会过，很多次都没有坚持下去，这也是很多人经历过的。但是我相信国外的开源爱好者也会有这种孤独感，但是他们就一年两年三年的坚持下去了，后来也就会有人来帮他们。

FREEMAN: 首先肯定存在技术层面上的差距，但是这种差距有其文化方面的原因。如果要细说起来就太多了，环境、教育、个人追求这些都是。比如说一个外国工程师做技术可以很有成就感，但是在中国就经常自嘲为“码农”。我们的价值观并不认同你成为一个技术牛人，这样的人可以对某一个项目或者某一项技术起到推动作用。我能看到的是，**在国外一个业界一流项目的创始人（团队中）有没有技术牛人坐阵实在是太重要了，这些人的想象力是无与伦比的。**

我们受到的教育让我们习惯于系统地来学习一样东西。比如说你要来面试，把《算法导论》和习题做三遍，这样写出来的算法都会让面试官很满意，别人布置的工作你都能完成得很好。但是我们很难以创造性思考的方式产生伟大的思想，一是大多数人没有时间可以闲下来，二是就算闲下来需要考虑的也都是一些琐事。广大开源者除了极少数受雇于开源公司的以外，多数人的生活基本保障都很困难。虽然有些人是在用业余时间做这些事，但是也

有人是全职扑在开源项目上。国内闭源商业软件大家还愿意用盗版，于是，更不用说愿意为开源付费了，都能看见源代码，为什么还要付费呢？所以说开源者只是凭借自己兴趣和爱好的话，也有可能会出好的软件，但是可能性不是很大。

中国的开源社区怎么样才会更好？

姜宁：商业公司之所以愿意雇人在 Apache 上作共享，其实就是因为他们的许可证是商业友好的。如果他们（商业公司）投入一些人力，就可以确保项目朝着期望的方向发展，同时还可以用项目进行商业化的运作，这样就可以给这些人开工资，达到商业上的平衡。**为什么大家愿意用（Apache 许可证），就是因为（商业上）使用起来很方便，这样用的人多了，社区自然就壮大了。**

FREEMAN: 从全球范围来讲，比较好的有生命力的开源软件的背后都有商业公司支持，有些甚至不只一家。很多商业公司也意识到开源的巨大生命力，原来的闭源软件也都贡献出来了。现在的趋势是，不靠软件本身来收费，而是靠服务来收费，（企业）靠的是手上的专家来提供专业解决方案。希望国内可以朝这个方向发展。

魏永明：我认为这里面需要一个引导，需要一个开源社区底层松散的引导团队。这个团队里需要的人既包括技术方面的，也包括商业方面的，甚至是资金方面的。这样的团队可以引导开源软件的开发者，告诉他们这个是不是有意义，用什么样的许可证比较合适，怎么组织，怎么发展，而不是今天做一个轮子，明天再做一个轮子，这是一种浪费。

应该向国外的开源社区学习，比如说 Apache，有一个基金会。国内的话成立基金会可能不那么容易，但是国外的一些模式是可以

借鉴的。现在的发展条件已经基本成熟，最大的困难可能就是如何注册一个 NGO，但是也可以只注册一个公司，通过财务公开，捐款人可以参与决策，就像董事会一样。**我认为这些东西不能靠商业公司来做，而是需要一个非营利机构。**商业公司的参与方式可以通过赞助，并在项目里安排人手来支援开源社区。国内的互联网企业完全有能力有意愿来做这些事。**现在愿意掏钱的人有了，但是问题在于如何形成一个机制，让这些东西变得公正透明。**最差的情况就是有好的组织起了好头，但是有人看到了这里的利润，然后以此牟利。

对于想进入开源这个圈子的人你有什么建议?

姜宁：这是一条明摆着的路，但是需要个人投入时间和精力，只要投入，就一定会有回报。这个回报是多方面的，首先你的工作能力肯定会提高，另外还有一点，就是你不用再被别人逼着去做什么事，而是自己在工作中找到乐趣。

找到你感兴趣的点来入手，然后就来社区上混就可以了，否则只是看文档什么的其实是很无聊的。就像混江湖一样，在这过程中你的 meritocracy（声望）就会不断提高，你的等级就会越来越高。这和现在一些回答问题的网站的模式很类似。这样就可以鼓励大家为社区作贡献。像你们图灵社区不是有银子什么的吗（笑），同样都是一种奖励机制，对会员起到正向引导作用。知识改变命运，想通过开源项目获取知识，只要你愿意，地球上没有人能阻挡你。在这里不拼爹，不拼公司背景，拼的是对技术追求的那颗心。

Nutz: 我要说一条建议就是不要做开源项目，不要投身于开源软件，不要混开源社区。如果你只是想把代码写好，就不要投身什么开源社区。因为首先就不存在这样的东西（开源社区）。最后的

结果可能就是人走茶凉，一帮人折腾半天，最后就是散了吧。每个人要对自己负责，要首先想好自己要什么，剩下的事都是自然而然的。

庄表伟：慎入。这个圈子看起来很有意思，但是进来以后你又会发现没有自己设想得那么好。它不像发微博，有两个人转发，五个人评论，你会收到一些反馈。写一个开源软件，三五个月没人理你是很正常的事，因为（如果）你写得不够好，别人可能没兴趣来看你的东西。所以需要熬得住，熬不住的人就会退出去。熬得住的人坚持做下去一定会有回报，但是这个回报一定不是很快的。如果说你想进来玩玩儿,你很有可能觉得没劲很快就退出去了，出去的时候还会说：开源没什么意思，都没人理我。很多人都是开源一个项目，扔在那没人管，也没人理，时间长了他也就不管了。这种事情是再平常不过了的。

魏永明：趁着年轻，30 岁之前的人大可以来尝试一下，十分欢迎。但是如果已经成家立业了，快 40 的人可能需要谨慎一点了（笑）。做开源软件未必能收到现实的利益，但是像我刚才所说的，是有机会获得一些名望的。以我自己为例，我 98 年开始做，99 年发布第一个版本，2000 年的时候就有很多人来找我，希望我去为他们工作，很多人也会愿意提供高薪给我。这个时候，你是否还想继续扑在开源项目上就是个人选择的问题了。但是前提是你要有这个能力，可以做出一个好的软件。所以我说，人在二十五六岁的时候，还在寻找人生的方向，作为一个技术出身的人来说，开源软件是个非常好的介入点。**你通过天分和努力很快就会获得别人的关注，接下来的人生就会有很多机会。可能很好的公司甚至国外公司会邀请你为他们工作，也有可能会有人想收购你的技术，或者从此走上创业之路。**

FREEMAN: 多参加社区讨论，多作贡献，你可以和国外一流的工程师直接打交道，只要你问的问题有质量。大家通常问问题有两个常见的误区，一个是经常会从商业的角度来说，“我需要什么样的一个系统”，问得很大，这种问题是不会有人来回答的；还有一种就是问入门级的问题，这说明你根本没有下功夫学习过。我经常会在微博上发一些“＃提问的智慧＃”之类的东西，里面有一些需要注意的事项。**技术牛人都是很认真地在回答问题，他们对软件的热爱、知识的深度和广度，以及他们心无旁骛的状态都是值得我学习的。** ■

Mozilla 首席技术官的工作进行时

作者 /Peter Seibel

Peter Seibel 是 Common Lisp 专家，Jolt 生产效率大奖图书《实用 Common Lisp 编程》的作者。耶鲁大学英语专业毕业，后投身于互联网行业，曾负责“Mother Jones Magazine”和“Organic Online” 的 Perl 专栏以及 WebLogic 的 Java 专栏，并曾在加州大学伯克利分校成人教育学院教授 Java 编程。2003 年辞职专心研究 Lisp 编程，之后即有了那部 Jolt 大奖图书。现在他是 Gigamonkeys Consulting 公司的首席执行官，和家人幸福地生活在加州伯克利。

译者 / 吴珂

Brendan Eich 现任 Mozilla 公司 CTO，是 JavaScript 的发明者，这种脚本语言是现代 Web 开发中使用最普遍却又最受争议的语言。Mozilla 公司是 Mozilla 基金会的附属公司，专注于火狐浏览器（Firefox）的持续开发。

1998 年，Eich 和 Jamie Zawinsk 带头劝说网景公司将浏览器变成开源项目，并最终成立 mozilla.org 组织，Eich 在该组织中担任首席构架师。

近几年来，Eich 既参与确定 Mozilla 平台发展的大方向，也会深入到底层去开发那个新的 JavaScript 即时虚拟机 TraceMonkey。并且，采访中他还强调正在尝试让 Mozilla 的项目“引领科学方向”，会吸收更多的具有务实精神的研究机构人员参与到 Mozilla 当中，让理论研究和行业实践结合得更紧密。

Seibel: 你是怎么设计代码的呢?

Eich: 反复做原型。我以前通常都会做一些顶层的伪码，然后再自底向上地做实现。但现在很少这样了，因为那些东西都在我脑子里了，我要做的就是把那些脑子里的东西自底层向上一步步实现。

如今我的工作通常是在已有的程序中加入新的子系统或给它拓展一下，大多数我都能从底层开始写起。只有在遇到困难的时候，我才会写一些伪码，然后再从底层代码开始写起。为了能够有足够的测试时间，这一步通常会很快。测试的时候我会检查到每一个环节以确定它们都运行正常。

在上面所说的那层设计之前，可能还有些跟实体关系或者粗略的模块化有关的工作。可能会有两三种算法帮助我弄清工程的复杂度——它是线性增长的，还是常量？每次我写某些线性搜索算法呈二次方地恶化问题，发布在网上的时候，做网络开发的人都会发现这是个问题，他们也都写了足够多的文章来强调这一点。于是我就转而开发那些关于常量时长的数据结构。在那个时候，常量可以不是 1，可以大到你喜欢的程度。

我们设计了很多原型，做了很多自顶向下和自底向上的开发，最后在中间层面上关联起来。在 Mozilla，我们并没有花太多时间去重写，有些保守。而且我们是个开源项目，所以要经营一个开源社区并吸引新人加入。我们有些东西还是被用户认可的，所以不想花上 3 年来重写而使项目停滞不前。如果尝试不断，真的要花 3 年。

但是如果你真的想再进一步，又不知道具体方向在哪里，那么重构吧。多一些尝试才知道自己究竟该怎么走。当你有了足够稳固的设计的时候，就能开始以这个设置为基础继续添砖加瓦。项目大体成熟后，你就会像我们一样不断地扔出补丁了。这其实就是代码进化的必由之路。现有代码作为沉淀资产也许能够支撑好几年，也许会成为急需更换的东西，也许又有更好的开源标准库出现了。

我觉得这才是编程的“手艺”。不能仅靠那些过时的设计编程，要不断地实践，包括对设计进行反思，把编程经验融入设计过程中去。

我对那些象牙塔里的设计和设计模式很反感。当 Peter Norvig 在 Harlequin 公司的时候，他写了一篇文章认为设计模式其实正反应了你的程序语言的错误。他让大家“去找个更好的语言”，这一说法显然很对。盲目崇拜模式，成天想“哦，我要用这个模式”显然是不对的。

Seibel: 所以新的东西会让你更好地前进。但是写程序写到一半结果发现最开始的设计其实有错误的时候怎么办呢?

Eich: 这种情况其实经常发生。而且通常不忍心完全推翻从头再来。就是感觉掉进了一个自己布置的陷阱里面。我在开发 JavaScript 的时候，很匆忙地弄个了一个底层的解释器。其实在刚刚开始做这个东西的时候我就知道我可能会后悔，但是之所以用这个设计是因为它比较容易理解，而我希望其他人也能参与到里面来，所以我一直在质疑这个设计，结果我明白了并不是任何时候都能够彻底反思最初的设计。我想,这也正是我们尝试大规模重写的原因，因为要想通过渐进式重写达到从根本上纠正原始设计的目标简直太困难了。

Seibel: 你是怎么确定什么情况下应该开始重写呢? 由于 Joel Spolsky 的书[1]，网景似乎成了宣扬重写代码坏处的招贴画了。

1 指《软件随想录》。

Eich: 第一个原因，Netscape 完全没必要去收购那些唯设计模式至上的公司，他们以为卖出去几个渲染引擎就是市场赢家了，其实那些东西很初级。用 C++ 和设计模式开发的，初看起来不错，但问题多多。

重写的第二个原因却是我在 mozilla.org 的时候感觉在网景的工作真是太糟糕了。Jamie 也跟我一样，都准备辞职了。我觉得我

们需要开放更多东西给那些第三方的开发者。我们不能再忍受这1994 年就开始用的如学生作业般乱糟糟的代码了。其实我写的那些 Unix 的内核代码风格的解释器也不怎么样。

所以我们需要彻底“重启”一下，可能需要 4 年的时间才能推出新产品。但是那时我们没有告诉管理层，我觉得他们一定会疯掉，所以我们只是含蓄地提了一下。我以为这种要求会让他们伤脑筋，但实际上他们做出了相当好的决策，比我想象中的还好。对于 Mozilla 来说，这个转变也是正确的。

就算是马后炮也好，我们还是挺幸运的，因为我们加快了互联网前进的步伐。微软却试图阻止它的发展——有人说这不是这家企业的本意，主要还跟反垄断案件有关。但不管怎么说，微软的做法给了我们时间去抢占网络标准的制高点，虽然“标准”是把双刃剑，而且有时候也是骗人的——也给了我们时间去重写。跟 Joel 一样，我也对重构持怀疑态度。我觉得要找到共同的兴趣然后拉到投资去开发，最后还不会错过市场，这太难了，而且成功的例子又太少。

我刚刚提到的重写都是在设计原型的时候，这在小范围的设计中很重要。可能只是对一大段代码中的几行的正交变换，但是波及范围很大，所有的不变量都要满足。也许是个新的实时编译器之类的，问题就不大。

Seibel: 你会读别人写的代码吗?

Eich: 我会，我把它当成工作的一部分。代码评审是强制性的预先检查过程，同时，它大多数情况下还是对网景的糟糕招聘水准的一种补救措施，也是在整合代码时的审核步骤之一。在 Mozilla，

当程序员用到很多不熟悉的模块——比如这些模块除了离职的 Joe Schome 之外谁也搞不清楚的时候，我们还有一种单独的“高级评审”。要是有人能弄清楚是怎么回事，你就有了个人可以帮你把握大方向；如果你知道自己到底在做什么，就可以不用经过代码评审，有点像在“绝地武士团”里那样。当然，我们的要求也不会太宽松。

我们本没有设计评审，但是有时就会导致代码写到一半的时候再回去做设计评审。有人会对你说“嘿，去黑板前再看看设计，你写的代码太多了，其实还有更好的方法。”当然这只是例外情形。我们不会死抠“瀑布工作法”，先设计，然后才是实现。但是在 80 年代我刚刚进入这个行业的时候，那种方法很流行，但坦白地说，，如噩梦一般。你先花时间写一堆文档，然后开始写代码，写着写着你经常会意识到不对头，然后就会重头彻底改写代码，把之前写的文档扔到脑后，永不再管。

Seibel: 所以，这样的代码最后就被带到 Mozilla 项目里啦？你读过别人的代码吗？那种不是 Mozilla 内部的，只为了看看的。

Eich: 这就是开源的好处了，我喜欢去看世界上其他地方的程序员的代码。我没有花很多时间在这个上面，但是我还是看了不少服务器端框架的代码，还有 Python 和 Ruby 之类的。

Seibel: 是它们的实现吗？

Eich: 有实现代码，也有库代码。特别是看 Ajax 的那些库，看着会很振奋，你会发现有人竟然那么聪明，居然用诸如闭包、原型和对象等等小工具写出了那么合理、方便非常的抽象应用。这些东西可能还不完善或者不够安全，但确实太方便了。

这是一本访谈笔录，记录了当今最具个人魅力的 15 位软件先驱的编程生涯。包括 Donald Knuth、Jamie Zawinski、Joshua Bloch、Ken Thompson 等在内的业界传奇人物，为我们讲述了他们是怎么学习编程的，在编程过程中发现了什么以及他们对未来的看法，并对诸如应该如何设计软件等长久以来一直困扰很多程序员的问题谈了自己的观点。 本文选编自《编程人生：15 位软件先驱访谈录》。

Seibel: 除了读代码，很多程序员还会读不少相关书籍，你可以给他们推荐几本书吗?

Eich: 我本来可以对各种文献了解更好的，但是我觉得做程序跟学声乐差不多，练习更重要。当然，读别人的代码也可以学到不少。我觉得 Brian Kernighan 的书不错，写得很清晰，书里会从一小段代码开始，然后重复利用这一段代码，最后讲到模块化。还有 Knuth 写的《计算机程序设计艺术》，卷一到卷三。我很喜欢，特别是半数值算法那部分，还有双重散列之类的，关于黄金比例的证明则被留做练习题，很有意思。

但是，读书学编程这种方法的效果我总有点怀疑。我总觉得编程算是工程学，还有点数学。然后这里面有很多应用性的东西，但是还没有上升到土木工程或者机械工程那种程度。或许以后会慢慢地成形的。

计算机科学说到底还是属于科学范畴的，是一整套知识体系。但是 20 多年前在 Usenet 上有人说它是“轻量级科学，三分之一科学”，直到现在还有好多东西经不起时间考验。那些十来页纸的、10 磅字号的、不出版便消亡的论文，往往漏洞百出，倒是一些学术期刊上发表的论文很有价值，因为你要经过专家审阅，他们也不会跟你打哈哈，这些论文在发表之前都是经过仔细的审核的。比如机械证明方面，就让人印象很深。但这种做法还没有影响到程序员。我总觉得计算机科学这个东西里面少了点什么，导致我对读书学编程这种模式有点怀疑，我不该总是这样敌视新东西，但是确实有地方不对劲。

计算机科学里有理论，也有很多重要的知识要学。你需要花不少时间去研究学习。在开发 JavaScript 语言时候，我认识了一些在

理论方面很厉害的人，他们中有人堪称黑客，我觉得这样就很不错。但是也有一部分从来不编程，他们肯定不是实践性人员，他们有很好的见解，有时候也能发挥效力，但是真到了实际写程序，交付给用户，让它能用，让它具有一定的市场竞争能力的时候，空有理论也白搭。但是话说回来，我也喜欢研究理论，理论对改善我们的生活还是很有帮助的。

怎么挑选有天赋的程序员？

Seibel: 下一个问题是关于选拔人才的，你怎么挑选有天赋的程序员？

Eich: 前些时候我们雇了一个人，他是我们公司一个技术大牛的朋友。但是这个人好像只是本科毕业，或许甚至都没有毕业。他和我们公司的这个人都是做 OCaml 开发的，他自己好像有一些项目，而且他好像还对我们正在进行中的静态分析的项目有点想法。后来我们面试了他，我知道他很年轻，但是不能确定到底是多大。有人说：“这个小孩没做什么东西，我们应该只请技术大牛，找他干什么？”

我反驳说：“你们弄错了。这个孩子就像一个很不错的实习生，你要在他们年轻的时候就抓住他们。而且他做过不少东西，会 OCaml，不单单是会这个源码语言，还研究了它的运行时、本地方法，还在用 OCaml 写操作系统，练习用的那种。这个人其实很不错。”所以我没有给他什么笔试，只是听他说他自己的项目，以及做这些项目的原因。他绝不是在仅仅重复那些平淡无奇的 C++ 模式——可是我们这里有不少这类年轻人。他们也都是好人，合格的程序员，天天做着 Java 企业级开发什么的——但是我们需要与众不同的人，这个孩子就是。

所以面试他的时候，最主要的问题是劝人们不被他的年龄所误导，以为他不够格。我们后来招了那个小孩，他表现得超级棒，做了不少静态分析的工作，先是基于 Berkeley Oink 的开源框架的，然后用 GCC 做插件，跟 GCC 的人合作。现在他正在推进移动开发方面的项目，负责给别人做性能分析，从时间戳输出中找出效率不高的地方并加以改进。

我在招聘的时候就知道这个人很有才华，再者来说他是牛人推荐的——要知道们会彼此欣赏，他们也知道怎么判断一个人厉害不厉害。一般他们不会胡乱地说：“雇我的朋友吧，他可不怎么聪明。”厉害的人从来只愿跟厉害的人一起工作。可能这听起来有点取巧，但这也算是我识别天才的一种方法。这也是为什么我们已经请了很多超级牛的程序员的原因。我想我们搜刮了 Valgrind 的所有人才，有些人什么都能干，他们可不是混事儿的主。

Seibel: 你经常会在面试的时候让对方说说他们自己做过的项目之类的吗？

Eich: 的确，我很少出难题考人，当然，我们公司也有人会出题目。要说我们非得这么干，把它作为一种筛选制度，我还是蛮担心的。

Seibel: 那用题目来作第一轮筛选也不好？

Eich: 我持保留态度。Google 经常会这么做，所以他们会招到不少“答题高手”。但是有些人的市井般的精明其实没有必要，我们应该有更成熟的评判方式。所以我对这种方式有些怀疑。是的，出题可以排除一些只会说不会做的应聘者，但是在那之后，你肯定需要听听他们自己的想法，看看他们是否曾经解决过什么问题。所以我们会给出一些实用性很强的问题。但我们不会出脑筋急转弯或者纯数学题，我们会给出较多的编程题目。

一定要考察应聘人的 C++，因为 C++ 很繁琐。当然这一步也只是一个初步筛选，不是这方面表现得好就一定能被录取。当然，通过这一关，说明他们还不错，对于没通过的人，就有理由担心。要决定录用，还要看些其他方面的情况，诸如他做过什么，用的什么方法，采用什么语言等等。

或许我对怪才有偏爱，我不介意一个人有多么不走寻常路。虽然我也不希望跟一个很难相处的人一起工作，但是才能是第一位的，而且公司也需要从不同角度思考的人。

我面试的时候最注意的就是才华，所以我会一直抓住很细节或者很实用的地方。比如上面提到的那个人，他会 OCaml，说明他很聪明，但是这就够了吗？显然不够。结果在接下来的面试中我还发现这个人自己做过不少东西，还有些自己的想法，而且还对编译啊、分析啊等等领域颇有心得，而我们正需要聘请懂这些方面的人。还有一点也很重要，那就是这个人是我们的一个很牛的雇员的朋友，这一层关系是不能被忽略的。

Seibel: 那，你还在享受编程吗？

Eich: 当然，编程会让人上瘾的。这个东西很有挑战性。对现在的我来说，编程不仅仅是写正确的代码，更是一种找到一种有智慧的途径去解决问题的过程。要懂得新泽西风格 [1] 式的 90/10 原则 [2]，这一正确、可爱的理论不可能教你解决所有问题，但它说你若解决了 10% 的问题，就不会陷入绝境。理论上总是有一种方法可以用最简单最短的代码解决问题，但理论和实际还是有距离的。这就是我喜欢编程的地方，直到现在它还吸引着我。编程很有意思，能让我不停地熬夜。■

1 又称“差即是好”(Worse is better)，由 Richard P. Gabriel 提出的关于软件接受度的理论，认为软件质量不见得与功能多少成正比，有时功能少的软件会更实用。

2 由 Stephen Covey 提出的一种理论，认为生活中有 10% 的事情是既定事实，剩下的 90% 的事情取决于我们的行为。

人　物

代码的未来

——专访 Ruby 之父松本行弘

松本行弘（Yukihiro Matsumoto），1965 年 4 月 14 日出生于日本鸟取县。1984 年，就读于筑波大学第三学科信息学系。2 年后休学，成为末日圣徒耶稣基督教会的宣讲师。大学复学后，加入中田育男教授的研究室。1990 年大学毕业。后在岛根大学攻读博士课程，修满学分后退学，未获学位。现任株式会社 Network 应用通信研究所研究员、乐天株式会社乐天技术研究所研究员、Ruby association 理事长、Heroku 首席构架师。

记者 / 周自恒、乐馨

人生经历

高中时代：用自己设计的语言编程

第一次接触电脑，是小学 6 年级。父亲给我买了一个口袋型电脑 L-Kit16，当时真是激动坏了。到了初中 3 年级，父亲又给我买了 Sharp PC-1210，这个时候我第一次知道“编程”这个概念。只要发出指令，计算机就能按照你的指令进行操作，真是有意思极了！

我的父亲，是建筑公司的一个普通的上班族。他之所以买电脑，是用来计算建筑数据的。没想到这两台电脑成了我的玩具（笑）。

回忆起小时候，感觉一天到晚都在看书。我家前面有一个书店，我天天都泡在那里。有时候朋友来玩，我让他到家里来，他却指着书店说：“那不是你家吗？”那时候什么都看，科幻小说呀，漫画呀。百科全书全都记在脑子里。总之就是特别喜欢文字。实在没东西看，就看药品的说明书（笑）。

上了高中之后，就迷上计算机了。成天看计算机杂

志。能让计算机按照自己的指示来运行，这比什么都有意思。特别是对编程语言非常感兴趣。当时对汇编语言、Basic 都不喜欢，不想用它们来编程（笑）。于是我就开始自己发明语言，然后用这种语言来编程，写在笔记本上。不过这本笔记已经不知道去哪儿了，真遗憾。当时连语言的名字都想好了。当然了，不是 Ruby 啦。不好意思说，哈哈。

大学时代：不是去图书馆就是去研究室

大学我考上了信息科学学院。在那里我接触了许多在书中看到但从未使用过的软件和语言。大学里的老师都是我高中读过的书的作者，简直太不可思议了。不过，研究室里我是最不听话的，老是猫在那里设计自己的语言（笑）。

当时是日本泡沫经济崩溃的年代，但是我一点儿都没有闲着。整天泡在图书馆里，看了许许多多书。休息的时候就去电影院或者是书店（笑）。我不擅长运动，也不关心身边的人都在干什么。我从小就喜欢读书，虽然最近拜网络所赐，读得少了（笑）。阅读各领域的书籍，对我的成长影响很大。 另外，我还常常关在研究室了。比如通宵在研究室里弄电脑，到了早上回家洗一下澡，然后再到研究室去（笑）。当时是 80 年代后期，研究室里已经连上网络了，我常常看 BBS、新闻什么的。能接触到很多聪明的人物，觉得非常有意思。

当时是计算机的黎明期。我父亲是反对我考信息科学学院的。“什么计算机，什么编程，当个兴趣就得了。”不过我真是太喜欢计算机了。未来的出路我从未考虑过。可谓是义无反顾了。

经济不景气，开始开发 Ruby

1990 年毕业之后，我进了一家软件公司。我找工作有一个原则：坚决不在东京工作。我从小在农村长大，不喜欢人多的地方。花 1、2 个小时去上班，那简直不可想象。我其实更适合在小地方工作，不过那样的话可以选择的公司就比较少。后来遇到滨松一家软件公司招聘，我就去了那里。

我的主要工作是公司内部 OA 软件的开发。当时的电子邮件只能发送文字，我就弄一个可以粘贴附件的软件。有意思。公司一般会给我一个大致的范围，然后让我自由地去发挥。因为喜欢这个工作，所以常常加班到很晚才回家。当时的梦想就是一辈子做一个程序员。即使将来当爷爷了，也要继续工作。

过了几年，泡沫经济崩溃的后续影响开始显现出来。公司的业务一下子少了许多。时间非常充裕，于是我就想自己干一点事情，开始开发 Ruby。这些我都是在公司弄的，也和同事交流过，同事亦给我许多帮助。

1 年之后，我去了另一家公司，叫做名古屋 CAD vendor。因为原来那家公司的经营状况已经非常不好了。虽然我对工作很满意，环境呀、人际关系呀都很好，不过当时我结婚有了家庭，考虑的东西就会多一些。万一公司倒闭了，那就比较惨了。而且我也想趁自己比较有优势的时候把自己推销出去。当时能做面向对象的脚本语言的人还是比较少的。

1995 年，松本行弘将 Ruby 公开于众。反响极其热烈。

代码的未来

图灵社区：松本先生今年出版了新书《代码的未来》，这本书的中文版正在由我进行翻译，预计明年会在中国出版。您的上一本书《松本行弘的程序世界》在中国受到了读者的好评，这次的新书和前作相比有哪些不同，又有哪些看点呢？

Matz：《松本行弘的程序世界》一共涉及了 14 个话题，每个话题都是浅尝辄止，内容比较广泛但不是很深入，而这次的新书则是设定了一个大的主题——即对未来即将到来新技术的思考，因此内容比《程序世界》所涉及的范围要窄一些。此外，这本书还在时间尺度上进行了探讨，例如从计算机出现以来，到现在为止经历了怎样的变化，并由此来思考未来可能会发生的变化，也就是对过去和未来两方面都进行了思考。计算机的世界变化非常快，而这本书的目的在于探讨其未来变化的方向。

图灵社区：关于编程语言进化的方向，保罗·格雷厄姆在一篇名叫“一百年后的编程语言”的文章（参见《黑客与画家》P156）中，主张“拥有最简洁最小核心的编程语言”将是未来发展的趋势。对于这一观点，您在书中表示“不同意”，这是为什么呢？您对编程语言发展方向的看法又是怎样的呢？

Matz： 保罗是一个很喜欢 Lisp 的人，而 Lisp 所具备的特性正好符合他所说的“一百年后的编程语言”的样子，因此保罗认为一百年后的编程语言就应该变成 Lisp 这个样子。但实际上，Lisp 这个语言的历史已经有 50 多年了，说实话，Lisp 现在并没有成为一种有很多人在用的主流语言。我觉得这也许是因为 Lisp 对于大多数程序员来说不具备那么大的魅力，也就是说，作为一种“拥有最小核心”的语言，或者从某种意义上说是一种很“美丽”的

语言，和程序员们所期望的语言之间，存在着一定的差距。如果一两年的时间里，Lisp 的魅力没有被大家所接受，那还可以理解，但已经过了 50 年还没有被广泛接受的话，是不是它在本质上就不太符合大家的期望呢？**“对人类来说好用的语言”和“拥有最小核心的语言”之间的这个差距可能是很大的，我觉得可能将来 100 年也没办法消除。**至于未来的编程语言应该是怎样的，我觉得应该是兼具接近 Lisp 的运行模型，以及人类容易理解的语法这两方面特征，这么一看 Ruby 是不是更接近这样一种语言呢？

图灵社区：松本先生被称为 Ruby 之父，我们知道在编程语言的设计过程中，可能要做出很多选择，例如动态还是静态、基于原型还是基于类等等。在 Ruby 的特性中，您认为当初最难做的选择是什么？

Matz：在设计 Ruby 之前，我在上大学的时候还设计过另外一种语言，而那种语言是完全静态的，和 Eiffel 语言非常相似。而我原本也是特别喜欢静态语言的，不过上大学时设计的那种语言是以学术研究为目的的，多年之后，当我想设计一种编程语言作为自己的工具来用的时候，我就觉得还是动态语言实际用起来比较好用。抱着这样的想法，我设计了 Ruby，现在看来这个设计还是正确的。那么当初对于 Ruby 应该是静态还是动态这个问题，也许算不上是最难的吧，但至少是我在设计中做出的“最大”的一个判断。而在此之后，因为是动态语言，那就借鉴一下 Smalltalk 和 Lisp 吧，Perl 有一些功能也不错，于是如此这般吸收了这样一些语言的特性，也就显得比较自然而然了。Ruby 的特点在于 Mixin 模块，而这个特点在 Ruby 诞生当时还算是非常罕见的，因为我不喜欢多继承，总觉得应该有一个更简单的方式，所以就设计了 Mixin 模块。

图灵社区：那么现在回过头来看，Ruby 当中有哪些地方会让您觉得“如果当初设计成这样就好了”呢？

Matz：最开始的时候我的目标只是想实现 Perl 所具备的功能，因此从 Perl 借鉴了很多，比如说用美元符号（$）来修饰变量名之类的，现在看来觉得学得有点过了，搞得和 Perl 太像了。当然，除此之外还有其他一些小地方，但最主要的我觉得就是这个了，也就是跟 Perl 太像了这一点。**刚开始的时候，还没有形成 Ruby 的语法习惯和文化，因此很多东西都是从 Perl“抄”过来的，现在看来好像一股脑拿过来的东西太多了，里面其实有一些是不需要的。**而经过一段时间之后，Ruby 自己的文化已经形成，Rails 出现之后又形成了 Rails 的文化，而到了这个时候再看的话，可能就会觉得这些 Perl 的部分好像没啥必要呢。

图灵社区：大家都认为“Ruby 有现在的人气基本上都是由于 Ruby on Rails 的贡献”，您在书中也认同这个观点，那么您认为 Ruby on Rails 获得巨大成功的原因是什么呢？

Matz：首先是得益于 Web 的快速发展，几乎所有的软件开发平台都在瞄准 Web 这个领域。以往在用 CS（客户端 - 服务器）架构来开发的系统，现在都可以在 Web 上实现了。在 Web 上能够开发的应用变多了，这是一个主要的背景。另外，Ruby 的优势在于进行软件开发非常容易，也就是开发效率比较高。这两点结合起来，我认为就是 Ruby on Rails 成功的主要原因。

此外，Ruby 还有一些比其他语言强大的特性，例如元编程（Metaprogramming）、通过猴子补丁（Monkey patch）所带来的可扩展性等等，通过这些特性，甚至可以对基础的类进行增强。DHH 正是运用了 Ruby 的这些强大之处，开发出了 Rails。而对于

没有接触过 Ruby 的人，比如只用过 Java 这种比较“死板”的语言的人来说，会觉得“唉？居然还可以做到这样吗？”，我觉得这也是 Rails 成功的原因之一。

图灵社区：中国读者很关心的一个话题是，Ruby 目前最广泛应用的领域就是 Web 开发，那么在 Web 开发这个领域之外，Ruby 的发展方向又是什么呢？

Matz： 的确，Ruby 在 Web 开发领域被用得很多，例如 Rails、Sinatra 等开发框架。但编程的世界并非只有 Web 而已，我也一直希望 Ruby 能够从 Web 中走出去。在不久的将来，我认为 Ruby 有望被应用的领域，主要有三个。

1. 科学计算（Scientific computing），也就是大学科研中所要用到的计算。在这个领域，Python、R、matlab 等语言用得非常多。我希望 Ruby 也能够进入这一领域，为此我们正在开发一个叫做 SciRuby 的项目，希望借此推动 Ruby 在大学科研计算领域的应用。

2. 高性能计算（High performance computing）。这个和科学计算有点接近，是运用超级计算机来进行计算的领域。和 C++ 比起来 Ruby 确实要慢很多，所以大家都觉得 Ruby 不可能被用于高性能计算领域。东京大学一个研究生做了一个研究项目，将 Ruby 写的代码编译成 C 语言代码，然后再编译成二进制程序，这个过程中需要用到类型推导等技术，最好的情况下，速度能够达到 C 语言（指用 C 语言人工编写的同等程序）的 90%。这个项目目前只发表了论文，还没有公开源代码，我希望明年这个项目的成果能够全部公开。

3. 嵌入式（Embedded）开发。所谓嵌入式就是指在微型设备，例如手机、医疗器械、机器人，在这些环境下，现在的 Ruby 其实并不是很适合，内存开销很大，API 也不合适，因此才需要开发适合嵌入式开发的，内存开销比较小的，并且具备面向嵌入式开发 API 的 Ruby 引擎，这也就是 mruby。

以这三个领域为首，我希望 Ruby 能够在 Web 开发以外的领域有更多的发展。

图灵社区：Twitter 主要是用 Rails 开发的，最近我看了一则新闻，说美国大选的时候 Twitter 遇到了前所未有的大访问量，Twitter 称为了应付访问量的上升，正在从 Ruby 转移到其他语言，您对这个问题怎么看呢？

Matz： 这里面原因很多吧。首先，Twitter 刚开始开发的时候，没人知道 Twitter 会获得今天的成功，当时很多人觉得，这种只能写 140 个字的博客有什么意思呢？但 Twitter 却出人意料地获得了巨大的成功。在这个过程中，Twitter 增加了很多新功能，在它快速发展的过程中，Ruby 的贡献是相当大的。因为一个新功能从构思出来到付诸实现，可以用很短的时间就能够完成。Twitter 刚开始开发的时候不可能考虑到会有现在这样大的访问量，也就是遇到了设计上的瓶颈了，因为一般的网站也不可能会有每秒上万的访问量，因此可以说现在的 Twitter 发展到当初在设计上的极限了。

为了解决这个问题，Twitter 需要开发一个全新的架构，以应付现在越来越大的访问量。不过，即便要重写架构，我觉得沿用 Ruby 也是可以做到的吧？（笑）话说，**一个网站在遇到设计极限的时候，有很多解决方法，比如重写架构、换其他语言等等，其中重写架构我觉得是最重要的**，而实际上 Twitter 也正是做了这方面的工作。

但在这个过程中，他们的工程师想要挑战一些新的东西，那么从编程语言上来说，就提出要改用 Scala，因为 Scala 是编译型语言，性能也不错，正好适合编写新的架构，我觉得这样也不错。

在我看来，在网站所提供的服务还没有完全成型的时候，最重要的是能够对需求的变化做出快速的反应，这个时候就需要 Ruby 这样灵活性比较高的语言；而在网站获得成功之后，遇到了设计瓶颈，用一种新的语言，比如 Scala，来编写一个新的架构，以节约一定的资源，我认为这也是很好的一个结果。Twitter 转向 Scala 还只是在其核心部分，而在 Web 前端和一些内部工具上还有很多地方在用 Ruby。其实，上个月我还去拜访了一下 Twitter，跟他们的工程师进行了一些交流，Ruby 还是用得很多的哦（笑）。

图灵社区：近年来随着智能手机、平板电脑等移动设备的普及，移动平台开发也变得非常热门。从编程语言来看，Android 上是用 Java，而 iOS 上则是用 Objective-C 来进行开发的，那么作为脚本语言，不仅限于 Ruby，您认为在移动开发上面会有怎样的发挥呢？

Matz：目前，提到移动开发，在 Android 上用 Java，在 iOS 上用 Objective-C 似乎是板上钉钉的事情，不过这也产生了一定的隔阂，比如某个 App 是为 iOS 开发的，如果要移植到 Android 的话，就得全部用 Java 重写才行。现在也逐步产生了一种新的尝试，例如 PhoneGap、Titanium 等框架，通过用 JavaScript、Lua 等脚本语言，编写出来的 App 就可以实现在 iOS 和 Android 的跨平台移植。作为 Ruby 来说，也有一种叫 Rhodes 的框架，通过它就可以用 Ruby 编写出在 iOS、Android 以及 Blackberry 上通用的 App。

以前移动设备和 PC 相比性能差距太大，如果 App 不能全速运行的话，就根本没法用了。但现在移动设备的性能已经得到了大幅度的提升，通过在通用框架的基础上，采用脚本语言来进行开发的方式，性能也逐渐变得可以接受，我想今后通过这种方式，用 JavaScript、Lua、Ruby 等脚本语言来提升移动开发效率的做法，应该会越来越流行吧。对了，刚才我们说到 mruby，其实用 mruby 来编写 iOS 和 Android 应用的项目也已经开始了呢，希望不久的将来这样的 App 能越来越多吧。

图灵社区：目前世界范围内广泛使用的语言大部分都是来自欧美的，作为例外大概只有来自巴西的 Lua 和来自日本的 Ruby，您在书中也说这种情况让人感觉“很寂寞”，那么造成这种情况的原因是什么呢？要改变这种局面，我们应该做出怎样的努力呢？

Matz：关于 Lua 呢，其实如果你要说它是欧美的也可以，巴西是属于“拉丁美洲”嘛（笑）。要说亚洲或者东亚这边的话，那就只有 Ruby 了，真的是蛮寂寞的一件事。从世界范围来看，（对于编程语言来说）欧洲和美国的影响力应该是最强的，而亚洲虽然有众多的人口，但却没有能够出现很多的编程语言，确实挺寂寞的。

如果说对将来的期待，别的国家我不太清楚，至少在日本，其实是有很多人在开发各种各样的编程语言，但除了 Ruby 以外，其他的语言在日本以外几乎就没人知道了。**如果对编程语言感兴趣的人越来越多，所创造出来的编程语言也越来越多的话，这其中应该就会有那么一两个能够取得成功吧。**在日本还存在一个问题就是语言障碍，日本人除了母语以外，精通外语的人不多，有趣的是，居然有用日语来编写程序的编程语言呢。（周：说到这个，其实中国也有用汉语写程序的编程语言呢。）中国也有吗？果然。不过这些语言虽然有趣，却只能给日本人用，也就无法走向世界了。

说句题外话，我曾经收到过一个美国人发给我的一封邮件，他说你是个日本人，但 Ruby 看上去却跟英语没什么区别，因为 Ruby 程序都是用英语写的嘛，难道没有用日语写程序的编程语言吗？为什么没有呢？我只好回答说：有啊，只不过你不知道而已，即便知道你也没办法用嘛。

现在在日本对编程语言感兴趣的人不断增加，大概我总是在网上还有书里说编程语言多么有趣，多少也是受了我的这些言论的影响吧，现在有不少人在挑战设计新的编程语言。在这些新的编程语言中，如果能有千分之一的语言能够最终获得成功，我认为就是很好的结果了。我不知道中国、韩国，以及其他一些亚洲国家中，有多少人想要挑战这一领域，不过如果大家以我的这本书为契机，**能够改变“编程语言是别人给我们的，我们只能被动接受”这种看法，继而抱有“自己创造一种编程语言也不错嘛”这样想法的人能够越来越多的话，这其中一定会有人获得成功。**

说到开源，无论是日本，还是中国、韩国，在世界范围内发表的项目还很少，这也算是一个可以去努力的切入点。这里面可能有很多原因，比如英语很难学，（周：比如 GitHub 也很难用？）哈哈，GitHub 在中国能用吗？（周：能用能用……）唔，还好还好。不过，这个（哔——）的影响还是很大的，有很多资料不太容易获得吧。（周：是啊，比如 Go 语言的官网就上不去呢。）啊！ Go 的官网上不去吗！果然是因为它是 Google 的吧（笑）。总之，需要面对的难题还是很多的。另外，在日本也是如此，程序员把大多数时间都用在了工作（挣钱）上，要进行开源项目之类的活动就比较困难了。10 年前，开源这个概念在日本也没人接受，而现在大家都逐渐明白了开源的主要性，开源项目也就逐渐增多了，我想未来几年中，在中国应该也会产生类似的变化吧，我很期待。

本书是 Ruby 之父松本行弘送给新时代攻城师的技术真经。对云计算、大数据时代下的各种编程语言以及相关技术进行了剖析，涉及 Go、Closure、VoltDB、node.js、CoffeeScript、Dart、MongoDB 等等当今备受关注的话题。

而且，在刚开始做的时候，没人知道到底做什么会成功。我在设计 Ruby 的时候，也不可能想着 Ruby 很不错以后一定会在全世界广泛使用。**因此，一种编程语言生逢其时可能更重要一些，但这种事情，你不去尝试一下是不会知道结果的。在中国，也一定会有一些编程语言或者软件因为生逢其时，从而走向世界并获得成功。**

图灵社区：非常感谢，在采访的最后，请您谈谈对中国程序员的寄语吧。

Matz：在《程序世界》中我也提到了，**我觉得编程的未来应该会以开源的形式发展下去，未来的创新性软件或者编程语言，可能都会以开源的形式出现。**作为开源软件来说，别人做出来我可以免费使用，这一点大家都很开心。在经历了这个阶段之后，如果能够更进一步，将自己做的软件开源，使其对全世界产生影响，如果能做到这一步的话，你可能就会成为一名一流的软件工程师。中国的软件工程师，就我接触的这些人来看，大家都非常认真和努力地学习技术，我希望这些人之中，能够有更多的人迈出这一步，从而成为可以影响世界的一流程序员。■

创造的乐趣

——专访郝培强（@Tinyfool）

郝培强，网名 @ Tinyfool，iApp4Me.com 创始人。身高 180cm，体重 240 斤，人到中年，有些肥胖，有妻有女，无房无车，现居上海。他是程序员、Blogger、二手经济学家、苹果产品义务推销员、Mac/iPhone/iPad 开发者……

记者 / 谢工、李盼

电脑是万能的

那台电脑无所不能，长得像个大吊灯，小孩说什么它都懂，这和我之后接触到的电脑很不一样。

图灵社区：你是怎么和电脑相识的？

我小时候看过一个动画片叫《星球大战》，里面有一个小孩，有一台大电脑。那台电脑是一艘飞船的中控，叫雨果，它无所不能，长得像个大吊灯，小孩说什么它都懂，这和我之后接触到的电脑很不一样。那时候大概是 Apple II 的时代，但我买不起。上初中的时候，出了小霸王学习机，我就让我爸买那个。买回来的虽然不是小霸王，但也差不多。我就想是不是可以用学习机来做游戏呢？那里面有一个 BASIC 的环境可以编程，我自己就写一些小游戏。那时有一种感觉：电脑可以让人无所不能。

到了高中，我开始接触真正的电脑。学习机无法存盘，**我曾经写过一个巨长无比的程序，写了好多代码，结果我表弟跑过来按了下开关，没了。**高中学 PC 的时候，大家喜欢电脑的很少，因为那时候电脑是个不亲民的东西，屏幕上显示的东西都是黑白的。但

是我当时很痴迷，我感觉电脑本质是万能的，只不过我手里的电脑还达不到。Windows 兴起的时候，我父母还给我报了个学习班，就是学 Windows 怎么操作的那种。高考前两天，我有了一台电脑，后来整个假期都在玩电脑、写程序。我写的程序其实解决不了任何问题，但是我就是想写。

图灵社区：你大学的经历是怎么样的？

我高考考得很烂，估分的时候觉得本科线都到不了，那就复读去吧。等到报志愿的时候，我爸说怎么也得找个明白人问问啊。**明白人表示，英语和计算机都是工具，是不能当作专业来报的。**其实不用他说，我的成绩也报不了计算机专业，后来我就去上了一个石油系统学校的机械专业。

高中的时候，我学不下去是因为有点狂，觉得自己成绩特别好。到了大学，学不下去的原因变成了数学。我数学底子不错，第一个学期没有怎么听，分数居然还很高，后半学期我就干脆不去了，每天去大学机房玩，结果高数考了个不及格。当时也没有人告诉我哪个科目很重要，高数灭了之后，我发现所有的科目都无法通过。到了大学，物理其实就是物理常识加微积分。感觉每门课都是在考微积分，而微积分我又不会。

这里面还有一个故事，大三的时候因为挂科太多，需要见家长。上了大学还要被叫家长，父母和我都很尴尬，因为学校已经打算让我退学了。**我去学校大门接我爸妈，他们一脸沉重，我也垂头丧气，这时候忽然有人跑过来说：“郝老师，你好！”**我们全家都愣住了，因为他们来的时候，听到的都是“你儿子不好好学习，天天不知道在干什么”之类的话。我和他们解释说，我在别的专业作培训，教了 400 多个学生怎么使用计算机。本来以为他们会

劈头盖脸地教训我，但是他们却和我深谈了一次。我爸觉得我做的事请还有点意思，但是无论如何也应该先把学位证拿到。毕业的问题是家里的关系帮忙摆平的。学校里一群不是我们专业的老师很喜欢我，但是我们专业的老师都觉得我是个累赘。我当时反思，如果一开始上大学就好好学习，后面又会怎么样呢？**左思右想之后，我觉得我仍然不想学那些东西。我可以端正态度让每一科都通过，但这仍然不是我要做的事。**是有点任性，但这就是我真实的想法。

图灵社区：没有系统地学过计算机，那你是怎么成为一个码农的？

我属于干活上手型。编译原理我是自己看的，搜索这方面我也是自己研究的。弱点我也很清楚，就是和算法、数学相关的，都学得不太好。我自认为分析能力比较强。比如说中学看小说的时候，大家租小说看，有人租了 1 到 20 集，租的人当然自己看第一集，所有其他人就都要从中间开始看。我通常都从第 10 集开始看，最后也能把故事串起来。学计算机也是这样，**一开始写的代码巨烂无比，但是之后的问题要自己解决，所以什么苦都吃过，也学到了不少东西。**另外，通过读、写大量技术文章，也学了不少。

我学习所有的东西都是靠兴趣，而且多学没坏处，知识面广了，要用的时候捡什么都可以捡起来。我的不安全感，也是通过学习新技术来缓解。可以说我没有特别精通的东西，我的强项是分析解决问题。我在面临需要通过学习来解决的问题的时候，我从来不走捷径，我会一行行分析，一行行搞出来。

在培养晚辈的时候，我认为最重要的是大方向。首先你用什么样的语言，要建立在理解的基础上。有人说我们的选择是自由的，

但如果不是在理解的基础上来做选择，选择就是不自由的。我不会告诉谁 Objective-C 就是好，我会说你先去弄懂 Objective-C 是什么再决定。C++ 也是同样道理。**我真的不喜欢 C++，但我不喜欢它有我的道理。对于我来说，它太麻烦了，而我是个怕麻烦的人，我宁可学一个简单的语言。**象耗子这样能把这个复杂度弄懂的人应该去学，我知道他能弄懂。

关于事业的三个故事

我们俩就去和人谈合同，谈着谈着，人家问：多少钱啊？我们才想起来，这个事我们还没商量过呢。

图灵社区：第一份工作是什么样的？

大学毕业的时候特别迷茫，去了无数双选会，参加了无数场面试，但就没有什么工作让我觉得一定要去。我妈勒令我去找工作，就打了份简历，去了一家求职中心。我走到第一个台子，聊了一下，觉得对方很 nice，就把简历给了他，然后直接回家。回家后，我妈说人家已经电话通知决定要我了。

我第一份工作在一家电子厂，他们是给摩托罗拉做配件的。那份工作很简单，其实就是一个网管，我的专业他们也不在乎。工资 1200，租了一个 200 块钱的单间（房间里都没厕所）就开始上班了。**主要工作内容就是，如果有个女孩说我的电脑连不上网了，那我就爬到桌子底下看是不是线被踢了。**比较棘手的是，如果离路由器很远的人网断了，得查出是哪根线断了。我归开发部管，公司只是组装摩托罗拉的手机，但是测试机生产线都要自己做，摩托罗拉不管，甚至我们经常比摩托自己的厂子更早建生产

线，把我们设计的测试机卖给摩托用。其中一种测试机就是要给屏幕供上电，然后检查屏幕上每个检测点。我们要走一段自己的动画，然后看看是不是每个点都看得清。我们当时想做一些新产品，有无数的想法，但是都没有做出来。后来我还做过工厂的工资管理，考勤管理的系统，那时的工作很好玩。后来这个厂发生了一些结构变化，他们觉得我做网管做得太好了，总能解决棘手的问题，于是不想让我写程序，专门做网管，这个事情我无法接受。

图灵社区：后来你怎么和霍炬做起咨询来了？

我后来帮了一个做记者的哥们做一些事，他每次都请我们吃饭，聊一聊，解决完问题就散伙了。有一天他说：“总是这样，久了好像有点对不起你，我还是给你钱好了，反正每个月都会打扰你一两次，不如我给你每个月三四千的兼职工资。”当时正好工作得不顺心，就想，要是这样的工作找上两三份，不就不用坐班了嘛。咬咬牙，辞了职，然后遇到霍炬，他说他也不想干了，于是我们一拍即合，决定一起开个公司。在一次参会的路上，一个同去的哥们知道我俩的情况，给了我们一个两万块钱的项目。公司还没有成立呢，就先拿了一个单子了。终于，我们有公司了，还有了第一单生意。

后来经人介绍认识了一位网站 CEO，本来没觉得会有什么交集。但一个月之后，他找到了我们，因为他们网站流量翻了几番，想要做一些优化。**我们俩就去和人谈合同，谈着谈着，人家问：多少钱啊？我们才想起来，这个事我们还没商量过呢。**我说 8000 一个月就行了，还现编了一套规则：一个月最多四次拜访；电话邮件随意；不写代码，只解决问题。当时的旗号就是做咨询。其实我们只是比普通程序员里眼界稍微开阔一些而已，也没有玩过负载，只是听人说过或者看过一些文章，很多东西对我们来说也都是全新的，就是硬着头皮往上冲。从某种角度来说，最后也都忽悠出来了（笑）。

客户解决了性能问题之后很高兴，他们意识到以后还会遇到其他问题，所以合作还是继续下去了。我们发现他们的搜索很烂，而且有人出价 20 万要帮他们做，于是他们邀请我们来帮忙。我觉得首先这是个挑战，其次我们万一做不来，这单子不是要丢了?

我这个人什么都没有学得很深入，但在 Google 刚火的时候我就研究过搜索，到处看文章。01、02 年有一个叫 Lucene 的东西，很多人介绍过，我也觉得很有意思，但当时觉得它没什么用武之地，所以也没有真正用起来。借着这个机会，我买了两本书，《Java 语言入门》和《Lucene 实战》，边看书边写，九天时间我给他们写了一套系统。Java 和 Lucene 都很简单，很多人玩不明白是因为他们对搜索不理解。我看了很多年的搜索，虽然从来没有动过手，但我对搜索的概念门儿清。

这个系统我很骄傲，因为它不是为这家公司量身定做的，而是一个通用的架构。虽然用的是 Java，但我觉得未来的客户未必会局限于此，所以一定要有 Web 接口。当时我不会玩 Tomcat，也来不及学习，于是我找了一个 Python HTTP Server，这是一个开源的架构，照着它写了一个 Java 的 HTTP Server 作为前端，这样我们的程序不需要跟 Apache Tomcat 组合，自己启动就可以接上了。我和我的合作伙伴有一场争论，他说做 C 的更好，我说那个太新了，缺乏社区经验，也许不靠谱。Lucene 再慢也是个成熟的东西。于是我们分头去用不同的方式来实现，最后他那个没按时写出来。

客户上线之后，效果很好。**原来每天 4000 次搜索，换了这个系统之后，第二天就变成了 8000 次，第三天 40 000 次，第四天 80 000 次**，客户也很信服。

图灵社区：你们用这个搜索系统做了些什么？创业了吗？

这个事完成之后，我们觉得除了咨询以外，搜索可以作为我们的一项产品了。一开始我们打算出租搜索，后来客户的竞争对手点评找上门来，直接签了一年的服务走了。然后我们融了一笔小钱，开始招人。我们还和六间房签了一笔单子，是按搜索量收费的。当时他们一天 2000 多万搜索，按照这个量我们的月租是很高的。

我们很重视这个项目，做了很多次大改造，那段时间真的很辛苦。我对 Java 的研究已经相当深入了，当时的学习方法是这样的：我在 Google 上搜 Java 性能、Java 内存、Java 高性能、Java 并发，以及把 IBM 开发站上 Java 性能调优的内容全部弄回来，把所有 Java 调优工具全部都过一遍。我们把搜索过程中每个部分，查询词、分词、查询词的解析等都单独加一个 timer，然后用用户数据 log 几万条跑压力测试。跑完之后，把数字拿过来打成一张图片，研究上面的高低点分布，这个时候再把压力翻一倍，看看这个压力怎么变化。**改一句，跑一遍测试，就这样假设、试验、确定问题，不断调试了两个星期，终于把性能调上去了。**最后一台机器跑 2000 万是没有问题的。悲催的是，我们虽然调好了，但六间房的视频牌照没有拿到，导致他们无法再融资，流量每日俱下。

我们做的工作虽然有价值，但是再也找不到有这么大需求的客户了。而且当时还有好多小客户都被拔线了（拔线门：一个网站出问题，整个机柜甚至整个机房无关的网站一起被拔线）。而且 A 轮融资的时候，本来融到了一家，但是觉得对方条件过于苛刻，害怕之后会有问题，就重新找。可是重新开始找的时候，美国发生了次贷危机，影响到了中国的投资界，大家都不肯出钱了。**当时我对创业有些灰心，小公司可能拿不到牌照，做网站会被人拔线，创业公司在这样的大环境下实在是太难了，我觉得还是继续打工好了。**

苹果和乔布斯

乔布斯的执念从来都没有变过，他要把科学家用的机器变成你手中的爱物。后来，他只是学会了顺应当前环境做事情的方式而已。

图灵社区：你是怎么和苹果结缘的？是什么东西打动了你？

我在高中的时候很爱看书，我看了三本传记，一本关于 IBM，一本关于微软，一本关于苹果。其中我最喜欢苹果那本传记，那本书是关于乔布斯的。这本书其实是把乔布斯当作悲剧英雄来描写的，他刚被苹果赶走，谁也不知道他之后会干什么。我很喜欢那本书的调调，悲情嘛。里面说了他对于这个行业的贡献，当时大家对乔布斯的印象和现在是不一样的，那时普遍认为他年轻时很聪明，但是后来就变扯淡了，很嚣张，结果弄得自己的公司都容不下他。

我认为苹果有一点像这个行业被淹没的传奇。现在很多人认为，乔布斯回苹果之后做出很多牛逼的东西，从这里开始才是乔布斯的传奇。其实，错了，乔布斯一直是个传奇，只不过中间中断了一段。他的精神世界从来没有变过，他要做的事也没有变过。

有人说苹果不是第一个开发电脑的，也不是第一个开发 PC 的。据我了解，最开始的时候有一种东西叫做阿朗星，很多黑客都沉迷于此，阿朗星就是一个方盒子，上面有一堆灯，一堆可以扳的开关，没有鼠标、键盘、屏幕。那些开关就跟插线是一个道理，就是这么写程序的，灯就是输出方式。**而乔布斯和沃兹尼亚克在很早的时候就提出，电脑一定要有键盘，人不应该用扳开关的方式输入程序。**他们觉得这个东西不装上键盘和显示器就玩不起来。苹果 I

和 II 其实也不带显示器，但是可以直接插到电视上，所以说苹果是最早把 PC 应该是什么样的构建出来的公司。

如果到这里乔布斯就收手的话，可能就不会有后来的盖茨。乔布斯认为计算机是个好东西，但是它是不能直接给普通人用的。85 年出的 Windows 一代是很可怕的，就是线框图，感觉完全不对。而他在 84、85 年做的 Lisa，水平已经和 91、92 年的 Win32 差不多了，但他太执着于做出心中的好东西，成本过高，又和主流产品 Apple II 产生竞争关系，造成了业绩问题。**但董事会的错误在于，这种视野宽广的初创企业依仗的不应该是职业经理人，而是有创造力的人。**我对盖茨评价不高的原因也正因为此，我认为他的视野并不开阔，他只是跟随乔布斯当时的想法而已。但是他有他务实的一面，他就没有因为价格而拒顾客千里之外。他视野狭窄有几件事都可以反映出来，其中之一就是跟随 Netscape 做浏览器，**这个东西没火的时候他没兴趣，火了之后就想抄，抄也没关系，但是 IE 做到 IE6，他又要把团队解散，份额全是他的，刀枪入库。**等到人家搞开源，搞出了 Firefox，花了三年时间把市场做到 10%，这时候盖茨又清醒了，又要重新做 IE。这说明他不是要把一个产品做好，而是要获得市场份额。而乔布斯当年却是牺牲了公司的利益，要成就一个完美的产品。

乔布斯回来做了一个播放器，很多人说乔布斯回来是做播放器的。但是在我看来，这个播放器也是一台计算机，它就是完成专门功能、和人交互的计算机。把电脑播音乐这部分抽离出来做到极致，让你觉得很舒服。iPod 后来就变成了 iPod touch，再后来就变成了 iPhone，这是有传承的。**乔布斯的执念从来都没有变过，他要把科学家用的机器变成你手中的爱物。后来，他只是学会了顺应当前环境做事情的方式而已。**

图灵社区：苹果的成功在你看来有什么原因？

苹果的销售数据显示，他们每一次的新产品销量总额都是上代产品两倍还多，这个数据在 iPhone4 和 iPhone4S 的时候都很明显。买过一个 iPhone 的人，下一个产品仍然使用 iPhone 的概率很高，达到 80%，而其他机器可能也就是百分之四五十。我认为这个翻番的销量很大一部分来自于在国外非常普遍的合约机。这些合约的周期通常是两年，而苹果新产品发布的周期和这个时间是很协调的。这非常巧妙，在全球规模来看，都很少有公司做产品做得步调如此精准。**每一代机器（一款新机型）和上一代机器的间隔都是一年，而每隔两代就会有一个大的变化。**一开始可能还不明显，到了 iPhone3 和 3GS，4 和 4S 的时候就很明显了。这个销售周期太合理了，合约完成了，下一代产品也出来了，用户只需要把合约继续就可以换新产品了。

一个产品的好坏很大程度上可以从顾客会不会续买这个产品看出来，首先是这个产品的质量有没有下降，然后就是凝结顾客自己的价值。2003 年 iPod 开始支持 Windows，同年苹果开始开拓音乐市场。当年 Mac 的占有率很低，虽然 iPod 很火，但是前三代其实卖不了多少台，因为没有 Mac 的话根本无法同步音乐。销量还没起来，苹果就敢卖音乐了。0.99 美金一首歌，一个 iPod 一百美金不到，可以存 1000 首歌，很多正版用户可能就会买一千首歌。你还会再买索尼吗？这点在苹果的成功里至关重要，到了后来，苹果已经和 1.4 亿张信用卡连在一起了。所以说乔布斯第二季的时候做事已经特别成熟了，很多事都是顺理成章的。**如果他们当年不卖那么多音乐，今天的软件会那么好卖吗？**卖软件，卖内容是一个持久生意。我的 iPhone 为什么不换，因为我的应用已经买了几千美金，已经超过了任何一款 iPhone。这是一个进去就出不来的产品。

但是光有产品就够了吗？重要的是苹果改变我们的生活。比如原来我打算教我女儿怎么用 Mac，但是当她在 1 岁半的时候，我给了她一台 iPad，我不需要教她怎样使用，她直接就会用。我曾经尝试过教我父母学电脑，老年人学电脑相当困难，总是有各种各样的问题。从这件事上我觉得，我们不应该让一个设备包打天下。用 iPad 几乎可以完成所有娱乐、轻松的事情，但是程序员、设计师可能仍然需要电脑来完成自己的工作。

创造的乐趣

> 我就马上对这门课失去了兴趣，太无聊了，设什么计啊，这不就是抄吗？结果整个大学我只有革命史得了优。

图灵社区：你自认为是个爱折腾、爱鼓捣的人吗？

我小时候也很喜欢物理、化学。我喜欢拆收音机，得到磁铁和一堆线圈，我会把线圈连到一块，看看是不是真的能把磁铁吸起来。我用我妈缝毛衣的针和两个铁片用胶水绳子缠到一起，还真的做了一个电动机的模型。从一开始不转，到后来一点点调角度、调相位让它可以运转起来。我还玩过口服液的瓶子，用它当试管装满水，放进金属片当电极，电解水，收集氢气，然后拿火柴来点燃。我的方案都是自己琢磨出来的，家里的电都是 220 伏，我就用了一个变压器弄成 12 伏电来电解水，但是我们同学听说可以这么玩之后，自己也去弄，结果**他按照书里说的没用变压直接接出两根线，加个灯泡来降压，接成了并联，结果可想而知，他们家的保险丝当场就被烧了。**自从有了计算机之后，有了其他渠道来消耗我的“创造”精力，这些东西玩得就相对少了。计算机不需要耗费材料，也没有危险。

我之所以报机械，是因为我认为也许机械这件事和我爱折腾的性格有着某种契合点。但是直到我上了设计课，**老师说，中国的设计模式叫做模仿式设计，模仿式设计是指我们从老外那里买一台机器回来，拆碎了，量，然后再装回去，根据这些尺寸画图。**这样做出来的机器其实往往跑不起来，轴不够结实，因为我们的钢不行。所以美国用 1 公分直径的轴，我们就用 1.5 公分，这就是设计余量，就是套公式。所有这些参数都是这样推出来的，其实没什么道理，就是死记硬背。**我就马上对这门课失去了兴趣，太无聊了，设什么计啊，这不就是抄吗？结果整个大学我只有革命史得了优。**

图灵社区：你现在在做什么，有什么想做的事情吗？

我现在正在做 app，**我愿意成为这种改变世界的力量的一部分，我相信这是做好事，但是由于做得还不够好，所以还没有收到钱。**虽然现在国内市场不是很好，但是我认为这个东西会改变我们所有人。

我想拿 iBooks Author 写一本教人动手的书，把我小时候鼓捣的那些乱七八糟的事写进去，我小时候做过电动机，物理课本里的图是很难看懂的，用平面图来表示一个立体的东西是很困难的。我对书的理解是这样的，书的未来应该是交互的。我所期待的电子书，是代码放在那里，按个按钮它就会 run 起来了，可以是假的，但是感觉是完全不一样的。教小孩的时候也可以这么想，音乐书、英语书怎么能没有声音呢？其实有一些已经上架的欧美教科书已经非常赞了，生物书里面的蚂蚁属于六足纲，哪里是头部，哪里是中部，还可以和同为六足纲的甲虫一一对应。这些应该会使老师的教学变简单，学生用 3D 都能看了。

图灵社区：美国一位著名投资人曾经表示，他认为现在这个时代是“不思进取”的，看似繁荣的 IT 产业没有给科技进步带来任何实质推进，这个观点你怎么看?

首先我认为这个人经济学没学好，世界应该是每个人提供自己所能提供的服务，然后其他人来购买，这个世界才是美好的。亚当？斯密第一本书第一章讲的就是分工。这个社会的分工应该类似于一个村子，如果你要一个人又做镰刀，又做斧头，还要种地，就会很悲惨，感觉非常累，虽然是自给自足，但是你家的所有东西都要自己做。原来我们每家都有缝纫机，其实是一种很惨的生活状态，一点都不浪漫。回到一开始的问题，谁的服务更好，让这个社会更有价值呢？谁也不知道。没有一个宏观的绝对值可以告诉我们什么更重要。而市场可以告诉我们，这就是钱的价值。供求关系和随时随地浮动的价格会告诉我们，这个社会缺少的是什么。**为什么现在游戏很火，是因为这个东西还比较新，说明我们之前不是随时随地都可以玩游戏的，原来我们不能放松的时间，现在可以拿来放松了。**如果有一天游戏多到我们在地铁上找不到想玩的游戏了，那游戏自然而然就不会再赚钱了。所以我觉得那个人的说法太共产主义了，而资本主义的说法不是这样的。同理，你给女孩子生产花花绿绿的衣服是错的吗?

图灵社区：你认为自己是一个果粉吗？或者这种感情中有什么类似于信仰的东西?

如果说苹果有一天背离了这些我喜欢的特质，那么我就不会再喜欢它了。我原来是很喜欢 Google 的，但是我发现它有一点认知是我不喜欢的——那就是这个世界是免费的。**开源不坏，全免费也不坏，但是任何一个执念不能被完全垄断了。**苹果是垄断不了的，因为它做收费，**只要有人做收费，就一定有人在做免费，而收费**

是垄断不掉免费的，但是反之却成立。以 iPhone 这个价格，永远都还有低端机的市场。有人说如果所有人都用苹果，那这个世界就千篇一律、没有未来了，但我认为，苹果虽然都一样，但里面的东西是不一样的。**只有机器一样了，很多东西标准化了，才能把里面的区别发挥到最大。**

我认为 IT 产业是这样的，一种模式靠内容收钱，一种是靠流量收钱。靠版权收钱的东西再大，也无法挤压靠流量收钱的。现在免费的已经快把收费的挤死了。我不认同免费的地方是，**免费不是一份简单生意，免费也不便宜。**Textmate 卖 39 欧元，15 万份是多少钱？而做免费软件的人，用户 10 万是没有收益的，20 万也没有收益。你想想这些用户里面有多少人天天用，又有多少人会点入广告，做这样的软件连生活费都挣不回来，做到 1000 万可以有收益了，但那是非常难的。所以**做免费软件的门槛相当高。你做任何东西，新浪有一个差不多的，一百万用户，你挣什么钱？**而这样的大公司可以等，开始的时候可以不挣钱。如果一直这么玩下去，我不知道怎么才能做创新。

如果国家想扶持 IT 行业，第一件事就应该是尊重知识产权，我们的价值就会翻番地涨。现在这个行业很不值钱，虽然听起来好像有一堆上市公司。我对盗版的痛恨也正在于此，我知道我的呼吁不会解决任何问题，但是我会把这件事当做行善和尽义务来做。我们国家做出来的创新少之又少，而聪明人又那么多。只有尊重知识产权，让做创新的人挣到钱，才能把创新发扬光大。**原来 PC 流氓软件横行，后来出来一个大流氓，把小流氓都杀光了，我们将永远生活在这个循环里头。**而苹果在这点上做得最好，一个程序无法访问另一个程序，这样也就没有人能做流氓软件了。我对乔布斯的欣赏源于我认同他对世界的理解。**我希望挣大钱，也希望让世界变得更好，如果两者可以一起发生，那就更好了。这点乔布斯做到了。** ■

九　卦

程序员的时间换算表

——为什么程序员不擅长估算时间

作者 / Anders Abel

Anders 是一位生活在瑞典斯德哥尔摩、在 Kentor IT 工作的咨询师，他是开发与系统架构方面的专家。Anders 自认为有一颗程序员的心，编程年龄 20 年有余，并仍然以此为乐。为此，他的博客叫做 Passion for Coding（编程的激情）。

一个曾经与我一起工作过的经验丰富的项目经理声称，他拿到程序员的时间估算以后，先将它乘以 π，转化下一个时间数量级后，才能得到真正的值。1 天转化成 3.14 周。他过去因为程序员不擅长估算时间而吃尽了苦头。我创建了一个用来翻译程序员时间估算的表格，来尽量缩小估算错误。

时间估算是困难的。每一个程序员都有一个现实的估计区间。低于这个区间的估计意味着（构件、测试、检查代码的）时间开销被低估了。超过这个区间的估计意味着这个任务太大而很难预估。

对于初级开发者来说，这个区间甚至都不存在。他们忽略（构件、测试、检查代码的）时间开销，同时困难的任务他们却又无法预估。我想说一个有经验的开发者应该在 0.5 至 24 小时将事情做完。超过 24 小时，就需要细分。这项工作应该在开发者的头脑中完成，然后总和到 60 小时。但是即使是有经验的开发者也需要利用管理时间块来思考。

同样重要的是要明白：编程经验不等同于估算经验。一个不被包含在估算流程中的开发者将不会擅长估算。同样，如果实际的时间花费不被测量和用于与估算比较，那么将没有反馈来学习。

估算时间	程序员所想象的	程序员所忘记的	实际时间
30 秒	只需要做一个很小的代码改动。我准确地知道怎么改，在哪里改。花费 30 秒敲键盘即可。	启动计算机，开发环境和获取正确源码的时间。用于构件，测试，检查和文档修复的时间。	1 小时
5 分钟	小事一桩，我只要上谷歌查一下语法就可以修复它了。	很少有一次就能找到完全正确的信息。即使找到，在它能工作前，也需要做一些调整。外加构件，测试等等时间。	2 小时
1 小时	我知道怎么做，但是写这些代码需要花费一些时间。	面对未来可能发生的问题，1 小时稍纵即逝。有些东西总是会出错。	2 小时
4 小时	需要写一些代码，但是我粗略地知道步骤。我知道标准框架中的 Wizzabanga 模块可以做到，不过我得查看文档，了解它的准确地调用方式。	这个大概是唯一现实的估算。它为意外的错误留下了足够大的余地，而这个任务也小到足以把握。	4 小时
8 小时	我先要把 Balunga 类重构成 2 个，然后为 Wizzabanga 模块加一个调用，最后为 GUI 加一些字段。	总会有许多系统的不同部分依赖着 Balunga 类。大概有 40 个不同的文件需要修改。为 GUI 新加的字段，同样也需要加到数据库中。8 小时太长，无法完全把握。总会有比程序员估算时更多的步骤出现。	12-16 小时
2 天	真的有一大堆代码要写。我需要往数据库里加一些新 table，显示 table 的 GUI，还有读写 table 的代码逻辑。	对于大多数开发者来说，两天的工作量已经大到难以估算了。肯定会有什么东西被遗漏掉。不仅仅是一些小事情，而是整个一大块主要功能会被遗忘在估算中。	5 天
1 周	哎哟，这真是一项艰巨的任务。虽然我还没有思路，但我不能说我不知道。一周应该够了，我希望，我真心希望，但是我不能要求更多了，否则他们会认为我不够称职。	这个任务已经大到超过大多数程序员的理解了。它应该被发回给架构师，帮忙将它划分成更小的部分，然后提供一些解决问题的方向。架构师可能会发现一种更简单的方法来完成它，或者发现其实有更多超乎想象的工作。。。	2-20 天

译者 / 刘建平

IT 项目经理，程序经理，目前处于创业阶段，曾任职于奥迪斯电梯、日本理光、IBM 等企业。从事了将近十年的软件开发，工作领域涵盖汇编、嵌入式、通信协议实现、算法设计、互联网应用等各个方面，同时也有丰富的项目管理经验，尤其擅长带领团队高质量地完成一些极具挑战性的项目。虽然已经是有点年纪的程序员，但是写起代码来还是乐在其中。生命不息，编码不止。业余爱好是看书，弹琴，打游戏。 图灵社区 ID：Liszt

最后，每个程序员都应该具备估算的技能。为磨练这个技能，接手每个任务时，先决定你要做什么，然后在开始之前估算任务所需时间，最后测量实际花费时间，并与估算相比较。同样比较你实际完成的与计划完成的。这样你将会既提高你对一个任务包含细节的理解，同样也提高了你的估算技能。■

查看原文：Programmer Time Translation Cheatsheet -or- Why Programmers Are Bad at Estimating Times

为什么写作自由书籍?

作者 / Allen B. Downey
美国欧林工程学院计算机科学系教授，也是 Google 公司前客座科学家。 他写了 Think Stats、How to Think Like a Computer Scientist series (Java, C++, Python 等版本)、The Little Book of Semaphores、Complexity and Computation、Physical Modeling in MATLAB、Learning Perl the Hard Way 等计算机科学书籍，这些书

本文是 Downey 发表在绿茶出版社网站上的一篇文章，说明了选择写作自由书籍的理由，并根据自己的经历提出一些建议。

1998 年，我为我教的一门课写了一本很短的教科书，然后采用自由许可发布了这本书。通过这种方式，我给予读者通过任何形式复制、修改和再次分发那本书的权限，而限制只有两个：衍生作品须说明原作，再发布时也必须采用这个许可。

我想到，其他教授可能需要从几本书中挑选一些章节，或者针对特别的课程来定制教材。我猜想他们中的一些人或许会补充一些资料，另外也希望读者能够帮我挑出错误。

那时对于自由书籍，我完全没有什么深刻的想法，当其他作家来问我的时候，我并不是一个多大的案例，对我来说，这不过是个好主意罢了。

八年之后，那些当时不很明显的益处变得益发清晰，而本文要说的正是它们。

- 自由许可不只是一个发布产品的不同方式。采用自由许可的书根本就是不同的产品。

都采用了 GNU 自由文档许可发布，允许读者可以根据自己的需要进行改编。

译者 / 杨帆

- 自由书籍会产生更多的自由书籍，这是常规书籍做不到的。自由书籍让读者变成作家。

现在，如果你有兴趣，来看看这个结论是怎么来的吧：

尽早发布，时常更新

我花了 14 天的时间写了《像计算机科学家一样思考》的前 13 章。在前 13 天我每天写一章，每章 10 页，最后用了一天的时间来编辑，然后发给一个短版印刷厂。他承诺在开课前交给我。我的朋友 Scott Reed 是位艺术家，他答应让我扫描一副他的作品当作封面。

在给学生这本书的时候，我说你们可以每周读一章。跟我写这书相比，你们有 7 倍的时间来读它。如果我们之间是有比赛的，我开始写的时候你们开始读，我会领先 11 周的。学生们当时没被逗笑。

所以自由书籍的优势之一就是周转时间，而另一个优势则是开始时需要投入的精力比较低。我可没有时间来写一本常规的教科书，也许也从来没想过要写。但我有时间来写一本自由教科书（也是勉强吧），但结果证明，它开启了一个意外积累的过程。

后一个学期的时候，我写了另外关于数据结构的 6 章。后来我报名参与计算机科学先修（美国高中阶段的大学先修课程）测验的评分，在那里我遇到了一些很棒的人，他们在教高中水平的计算机科学。我想我的书可能会有用，但那时先修测验采用 C++，所以我又为先修课程专门写了一个 C++ 版本。不过它并没有流行起来，特别是在先修测验转而采用 Java 之后。

但那时我已经有了一本 200 页的书，还有两个版本，而且这两个版本都发行了好几版。

放开控制

第一个真正的惊喜在 1999 年 4 月到来，我从 Jeff Elkner 那里听说了这个消息。Jeff 与 Richard Stallman[1] 谈了我的项目。Richard Stallman 曾看过我给 Jeff 关于这个项目的传单。

于是，Jeff 和我开始合作，Jeff 着手将《像计算机科学家一样思考》转为 Python 版，同时，他的一个学生也开始将 Java 版译成西班牙语。

我很快意识到，随着多种编程语言和多种自然语言的诞生，译文的版本已经有点超出控制了。

当那些可能的译者联系我时，我都鼓励他们公开他们的作品，并让我知道。有些这样做了，有些则没有。我愿意维护版本的序列，但我已经明白我无法控制这个过程。

Jeff 给我 Python 第一版时，我从未用过 Python，用自己的书来学一门新的语言真够古怪。我喜欢上了这门语言，但对这本书，感觉有点矛盾。Jeff 的风格与我不同，一般而言，我更喜欢我的。

我感觉有义务做一个基本修订版，虽然我没有时间完成详尽的工作，但我做了一个 14 天通，并把它发回给 Jeff。

对于这个结果我很高兴，但现在想起当时的回应，真是个错误。我是一个拙劣的编辑，而 Jeff 当时不是很开心。收回项目控制权的同时，我抹掉了很多 Jeff 的贡献。

1 美国自由软件运动的精神领袖、GNU 计划以及自由软件基金会的创立者

幸运的是，Jeff 非常有雅量，他还是继续与我合作。但想想吧，我的失策真的可以轻易毁掉这个项目。

如果你的书就是你的孩子，交给陌生人任由他们处理，确实很难。但如果你想有创意的人使用你的素材，就必须允许他们取得所有权。

可能必须做点监管

2000 年 11 月，一位比利时的高中老师 Gerard Swinnen 联系了我，他那时已经将 Python 版译为法语，所以请求获得我的允许，以自由书籍的形式发布他的翻译。实际上我已经在我的许可中给予了这种授权，所以我说可以的。

之后这件事就被我抛诸脑后，直到 2005 年 7 月，我 Google 了一下，才发现 O'Reilly 法国出版了这本书，名为 *Apprendre? programmer avec Python*。

当时我很兴奋，但也有点担心，因为 O'Reilly 的网页上并没有任何关于原书或翻译许可的信息。

我从 Amazon.fr 订购了一本，失望地发现，这本书还是遵循常规的版权协议，读者没有权利复制、修改或再分发原文。

在同 Swinnen 交换邮件后，我才了解他之前开始翻译我们的书，后来逐渐改写，直到最终不再使用大部分的原始资料。

一方面，我很高兴，这种使用自由书籍的方式我之前并没有想到。我不想阻碍 Swinnen 的成果或否认他的荣誉，但在我个人的烦恼之外，我认为有必要强制执行我的许可。

我发了一系列的电子邮件给 Swinnen 的编辑、O'Reill 法国的主编，并最终送到了 Tim O'Reilly 本人手中，在邮件中我这样写道：

“如果没有我的书作为开始，我想 Jeff [Elkner] 和 Chris [Meyers] 不会写出一本书，如果没有我们的书作为开始，Swinnen 先生也不会写出他的。这展现了自由书籍的力量，但同样也说明，处理违反 FDL（自由文档许可）行为的重要性。

作者会采用 FDL 是因为益处大过风险，但如果作品的滥用得到广泛传播或者被容许，这将坐实作者的担忧。原始作品的作者将继续采用严格的许可，而那些本来可能创作出重要的、有益的改编作品或翻译作品的人就再没有机会了。”

值得表扬的是，O'Reilly 先生之后指示法国办事处在那本法文书的网页上加上了我们的链接，并且他们同意重印时遵守许可。自那之后，我再没收到他们的消息，所以就我所知，他们还在销售那版我认为是侵权的书。

我希望这是唯一一个不遵守许可的使用，但其实还是有其他的。在这个案例中我能得到一些妥善处理，但对于大多数的情况，我没有时间去监管。而且，我只能看懂英语、法语和一点点的德语，所以使用其他语言的作者不在我的检查范围内，是相对安全的。

维护贡献者名单

从乐观的角度想，我每周都会收到几封读者的电子邮件，大多数都恭维了这本书，听到这些总是好的，尽管也有人专门写信可能只是对我的好意有偏见。

礼貌之外，很多通信人也会指出错误并提供建议，很多真的有帮助。我的第二本自由书，*The Little Book of Semaphores*，尤其满是错误。现在少一些了，但同步的困难就在于它很难确定。

早期的时候，Jeff Elkner 曾提议维护一份贡献者名单，并附加在书中。这绝对是一个好主意，很遗憾在最初几版我忽略了这一点。现在我的所有书都有贡献者名单。

想象一下好书如果是自由的

这学期我将开一门关于哥德尔不完备定理的迷你课程。在准备的时候，我发现了另一个支持自由书籍的例子。

内格尔和纽曼的《哥德尔证明》第一版于 1958 年出版，封底简介称其为“注解的杰作”。我认同这句话，想来 1959 年侯世达 14 岁读到这本书的时候，也是这样认为的。

《哥德尔证明》是侯世达那本广为流传的《哥德尔、艾舍尔、巴赫：集异璧之大成》的“父母”之一，侯世达从 1972 年开始写作这本书，并于 1979 年出版。

回顾《哥德尔证明》，侯世达曾说“被某些段落困扰过……过了一会，然后意识到，让他感到困惑的地方并不完全是自己的错。”侯世达写道，“意识到这本我喜欢的书有些瑕疵，而很明显，我却不能为它做任何事情，感觉很糟糕。”

对于一本去除瑕疵就非常棒的书，人们却觉得无法为它做任何事是明摆的事，这种想法让我感到难过。

但至少在这个例子中，最后证明侯世达是错的。纽约大学出版社后来请他编辑《哥德尔证明》的修订版，该版于 2001 年面世。

结果是令人惊讶的。在不牺牲第一版的简洁风格的前提下，他阐明了几个表述模糊的实例，清除了不太理想的举例，并将一些模糊之处变得更精确。修订版的成就毋庸置疑。这个例子说明即使在常规出版领域，一本书也能激发另外一本书，而且读者也是有助于改进的。

想象一下，如果周期小于 40 年，如果人们不用先拿到博士学位，并写过一本畅销书，就可以修改、改写和改进先前的作品，世界将会是怎样?

为什么选择自由书籍?

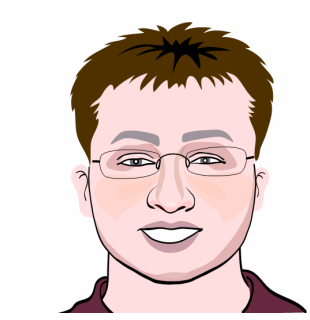
一本自由的书就像一棵大树的根，从它开始，分支出可能的改写本、翻译本，甚至是全新的书。自由书籍将读者变成校对、编辑、编者、通讯记者、投稿人、合作人、撰稿人和作者。

如果你正考虑写一本书，马上开始吧，尽早发布，时常更新，放开控制，但也要不时监管一下，维护好贡献者名单，让这本书自由开来。■

查看原文：Free Books, Why Not?

用 Markdown 来写自由书籍

——开源技术的方案



作者 / 蔡煜

蔡煜是上海爱立信研发中心的软件开发高级专家，作为软件实践的先行者，主要工作就是探索软件开发的最好、最适合的方法和工具。同时他是一个开源、协作和敏捷的布道者。 图灵社区 ID：larrycai

背景

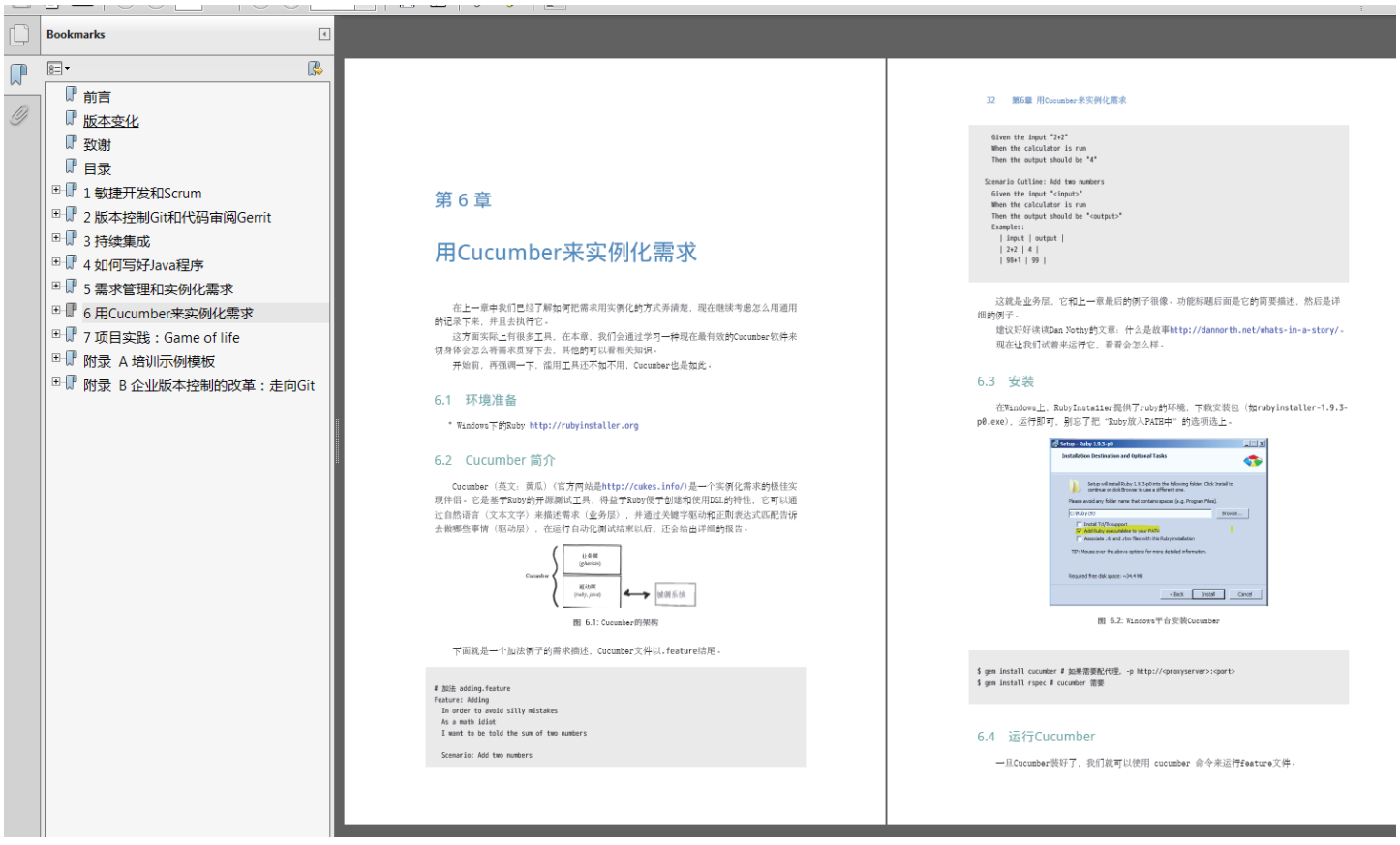
随着互联网的进步和技术积累，国内技术圈交流的气氛越来越浓，微博博客上到处可见有质量的技术贴，但是有时候想系统阅读时却不是很方便，如果能够整理成册自由分享该有多好。虽然少数人有幸通过出版社出书了，但代价太大，也不能普及大众，现在技术那么发达，有没有办法自己自助出版呢?

看了图灵社区的文章为什么写作自由书籍?，作为爱书之人，我为什么不能想个办法来解决呢?

技术无极限，办法现在有好几种，在本文中，我就介绍用现在流行的 Markdown 格式的方式来产生专业的书籍，而且还可以做到利用互联网，不需要本地机器参与的完美方案。如果你喜欢微软的 Word，觉得用它已经足够了，那我们不是同道，不用往下看了。

markdown 能做出什么效果呢?

多说无益，先来看看效果吧。下面是我去年写的一本小册子：跟我学企业敏捷开发在 PDF 阅读器中的效果。注意还有完整的目录和页眉。



书的内容就全是用 markdown 格式写的，上图相对应的文件内容（6.2 节）片断如下。

```
## Cucumber 简介 ##

Cucumber（英文：黄瓜）( 官方网站是 <http://cukes.info/>) 是一个实例化需求的极佳实现伴侣。它是基于 Ruby 的开源测试工具，得益于 Ruby 便于创建和使用 DSL 的特性，它可以通过自然语言（文本文字）来描述需求（业务层），并通过关键字驱动和正则表达式匹配告诉去做哪些事情（驱动层），在运行自动化测试结束以后，还会给出详细的报告。

Insert 18333fig0601.png
```

图 6-1. Cucumber 的架构

下面就是一个加法例子的需求描述，Cucumber 文件以 `.feature` 结尾。

```
# 加法 adding.feature
Feature: Adding
  In order to avoid silly mistakes
  As a math idiot
  I want to be told the sum of two numbers
```

下面就是书的全部 markdown 文件，每一章一个文件。我把它们分成了序、前言、致谢、正文和附录，蛮像回事吧。有兴趣的朋友可以看看作译者手册，来了解标准的书是怎么组成的。

```
$ find zh
zh
zh/0preface
zh/0preface/00-chapter1-preface.markdown
zh/0preface/00-chapter2-changes.markdown
zh/0preface/00-chapter3-acknowledgement.markdown
zh/1chapters
zh/1chapters/01-chapter1-agile-scrum.markdown
zh/1chapters/01-chapter2-git-gerrit.markdown
zh/1chapters/01-chapter3-ci.markdown
zh/1chapters/01-chapter4-java.markdown
zh/1chapters/01-chapter5-sbe.markdown
zh/1chapters/01-chapter6-cucumber.markdown
zh/1chapters/01-chapter7-workshop.markdown
zh/2appendix
zh/2appendix/02-chapter1-sample.markdown
zh/2appendix/02-chapter2-cc2git.markdown
```

全书的内容够可以在 github 上找到 https://github.com/larrycai/sdcamp/tree/master/zh

样式的产生全部有模板完成，一般不需要改动，这个稍后讲。

什么是 markdown

有兴趣了，就可以聊聊 markdown 了，简单来说，markdown 格式的文件看着像一般的文本文件，里面只是加了很少的格式标记，因此看文本文件也不影响理解，这种格式也有很多工具帮你去转化，而且很容易自动化解决。并且这些技术大多数是开源或免费的。

如 ## Cucumber 简介 ## 就可以转换成 html 的 <h2>Cucumber 简介 </h2> 或者书中的小节，空四个就是表示代码，是否很简单。具体可以看图灵社区的文章 Markdown 语法说明（详解版）

为什么要用 markdown

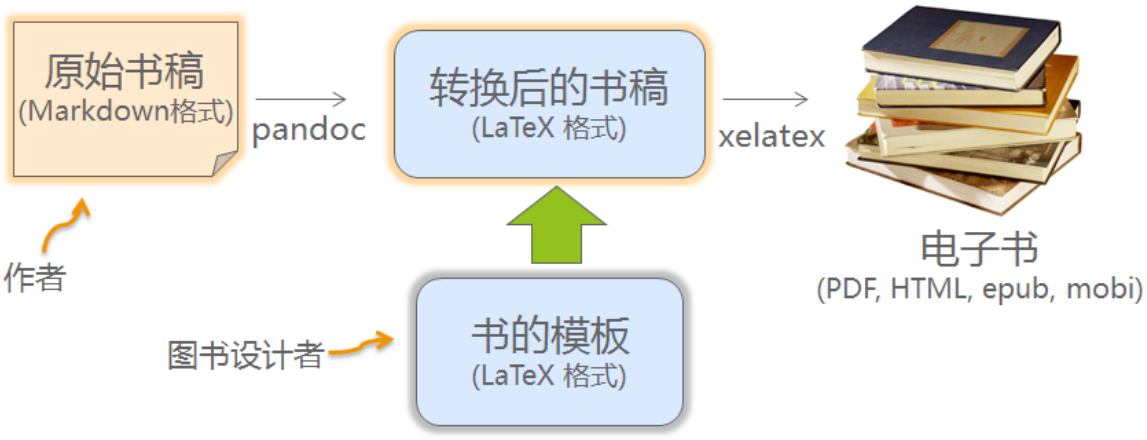
原因也很简单，因为简洁和流行。markdown 格式的普及要归功于 Github 和 StackOverflow。因为它们越来越流行，它们支持 markdown 格式也越来越流行。这里要赞一个的是，国内的图灵社区也支持 markdown，用起来超级方便。

我的体会是，它让你关注内容，格式怎么显示不是要你在写得时候关注的。Word 让我讨厌的原因就是老要关注格式。

实际上核心是软件思想中的分离，就像 HTML 关注内容，CSS 关注格式表现一样。

背后的魔法

从 Markdown 源文件产生电子书一般有两种方案：一种是产生 HTML 中间格式再转换出电子书，另一种是对应地转换产生出 LaTeX 文件格式，再和 LaTeX 模板合在一起，最后转换产生 PDF，达到标准书籍出版的质量（完整的封面，目录，页眉等等），这里主要讨论第二种。



LaTeX（就是 Donald E. Knuth（高德纳）发明的）是一个出版界（至少是科技界）常用的格式，PDF 也能很容易地产生出来，它的好处是内容和形式是很容易分开的，感谢大师的设计。

Markdown 可以通过工具自动转换出 LaTeX 的标准内容。

```
\section{Cucumber 简介} %
```

就是上面由 ## Cucumber 简介 ## 自动生成的 LaTeX 内容，表示小节。

余下书的表现形式就有预先调好的 LaTeX 的模板来处理，像页眉、目录、字体之类的和内容无关的就全部有 LaTeX 搞定了。不同的出版社或者图书系列都会有特定的模板，这和作者的内容无关。

这有点像 HTML 和 CSS 的关系。下面这一段是调整小节标题显示：

```
\definecolor{colorsection}{RGB}{95,158,160}      %
CadetBlue
\setromanfont[Mapping=tex-text,BoldFont="WenQuanYi
Micro Hei"]
\titleformat{\section}
    {\color{colorsection}\normalfont\Large\bfseries}
    {\color{colorsection}\thesection}{1em}{}
```

\bfseries 是让 LaTeX 使用粗体字排版，这儿选择了文泉驿的微黑。

下面这一段是调出页眉左上角（LE=Left Head）的：显示页码和内容，颜色设定为钢铁蓝，

```
\definecolor{colorheader}{RGB}{70,130,180} % SteelBlue
\fancyhead[LE]{\color{colorheader}\quad\small\textbf\
thepage\quad\quad\small\leftmark} %页眉左上角
```

初看有点复杂，学习曲线蛮陡的。不懂的话，知道里面含有样式就可以了，反正你可以用现成的模板，这也是出版的专业所在，不用写书的负责。

工具软件 pandoc 能帮着从 markdown 转换出 LaTeX 格式，然后通过 TexLive 软件中的 xelatex 再转成 PDF 格式。

强烈建议你到 Pandoc 的在线实验环境试一下。

这里主要讨论 PDF 的格式，实际上 epub/mobi 格式的用 pandoc 生成也很方便。

其他常用格式的出版方案

计算机类图书对格式要求不是很多，图文、章节、源代码基本就够了，就算有些复杂公式，也可用图来显示。这也从理论上说明，它不需要复杂的格式。现在对这类技术书出版我的理解主要有几种：

1. Microsoft 的 Word 格式，虽然国内出版界如日中天，缺省就认它（对技术没追求，鄙视）。简单好学，但是不擅长自动化，是开源的死敌。

2. 直接用 LaTeX 格式，这是很棒的东西，特别适合学术类的各种复杂的公式等，不过学习曲线很高，国内也只有几家学术期刊使用。

3. DocBook 格式是最有名的（从 SGML 演化过来），O'Reilly 和 Pragmatic 出版社缺省就用它，它能很方便地转化出出版要的各种样式。如 Jenkins - the definition guide 开源书就是采用 docbook。但由于是 XML 格式，很多人不习惯，而且多人网上协作不是很方便。

4. 通过蒋鑫的 Got Github 开源书，我也了解 reStructureText 也是和 markdown 差不多纯文本（plain text) 的，也是蛮流行的，结合 sphinx 也所向无敌。

自己动手产生电子书

讲了那么多，作为码农，该动手实践一下，其他读者可以跳过这一章。

你需要一台 Linux 机器（虚拟机就可以了）和简单的 Linux 命令就可以试验了。有 git 和 ruby 的知识那就更方便了。

我用的试验环境是 Ubuntu 12.04 (Precise) 版本。

下载书的源代码

很简单，git clone 一下就可以了，下载它的源文件包我觉得还是烦了点。

$ git clone https://github.com/larrycai/sdcamp.git

生成 PDF 是一个比较复杂的东西，用到了 pandoc 和 TexLive 软件，用 Ubuntu 库里就可以了。关于 Pandoc，可以看图灵社区翻译的文章关于通用文档转换器 Pandoc，TexLive 就是 LaTeX 的工具集。

$ sudo apt-get install pandoc

$ sudo apt-get install texlive-xetex texlive-latex-recommended texlive-latex-extra # 安装 texlive 2011

因为是中文 PDF，需要把字体嵌入在文件中，因此需要安装字体文件（如果不是 Ubuntu 中文版），具体可看我在图灵社区的文章开源书和开源技术 -PDF 中蛋疼的中文字体

$ sudo apt-get install ttf-arphic-gbsn00lp ttf-arphic-ukai ttf-wqy-microhei ttf-wqy-zenhei

现在你就可以生成 pdf 文件了。

$ ./mkbok

mkbok 脚本

pandoc 本身也支持模板，但很多情况下，还需要调整，写个脚本就方便多了。我用的脚本 mkbok 是基于 Pro Git 里面的脚本 makeebooks 和 makepdfs 开发的。原书作者 Scott 原来还用 calibre 产生 epub 文件，我统一用 pandoc。并原有的基础上进行了扩展，统一为 mkbok 脚本。

$ ./mkbok --build pdf,html,epub --lang zh --template latex/template.tex

这样就可一次性完成电子书的活，而且还改造了 LaTeX 模板（加了前言、致谢和页眉等，使它最后的结果更像一份标准的书。主要功能差不多，但是扩展应该会更好些，特别是有机会更方便地采用不同的专业模板。

基于 Github 和 Travis-ci 的互联网在线方案

你可以建一套基于上述技术的方便的出版系统。不过也可以利用互联网的服务，混搭一下。Github 和 Travis-ci 就是我要利用的混搭服务。

Github 和 Travis CI 这里就不介绍怎么使用了，具体可以先看晓斌的博客和蒋鑫的 Got Github，强烈建议你先试一下。要不下期码农吧。

简单道理就是当你把代码推送到 Github 时，就可以触发 Travis-ci 的构建。Travis-ci 会启动一个基于 Virtualbox 的 Ubuntu 的虚拟机（当前是 12.04 版本），然后根据你的 .travis-ci.yml 中的配置来构建你的产品，也就是执行上面的步骤来产生 PDF 文件。

构建结束后，虚拟机会被删除掉，虽然 Travis-ci 网站本身没有提供归档功能，但是 Github 的 API 提供了极好的方法来上传文件。所以我们可以用 Github API 和 Travis-ci 的保密环境变量的方式来把 Travis-ci 的结果上传回 Github。

具体请看我的博客把 Travis-ci 的结果上传回 Github。

李明老师的源码开放学 ARM 也是此方案的忠实用户，基本上可以全用浏览器解决。

结尾

里面的技术细节还有蛮多的，不知是否你都明白，实际上照着试一遍，你会发现不是那么复杂。如果有问题，尽管找我（@larrycaiyu）。

我一直设想中有一天上面提到的各个环节都能打磨得非常流畅，吸引更多的使用者。然后能够有很多人有兴趣用它来写书，可能是自娱自乐，也可以免费出版。不管怎样都是促进图书业。

出版界或者 LaTeX 的高手可以提供设计些更好的 LaTeX 模板（可以收费）供大家使用。

图灵出版社也提供网上的编辑服务（当然是可以收费的），对一些想把书写得专业点的作者提供额外的帮助，这样或许是一个很好的增值服务。至少像我这样的文笔超烂的技术人员很想得到的。

这些本是我 2012 想做到的事情：设想中的开源书项目，而且域名都买好了，留个遗憾放到明年吧。

技术无极限，只怕没追求。■

英雄出少年：
大金刚、小金刚和跳跳人儿

作者 / 张玳
澳大利亚卧龙岗大学计算机科学学士。主要关注 IT 产业以及电子游戏产业领域（特别是与任天堂相关的部分），并有多年 Web 及用户体验相关的经验。热爱读书、设计、培训以及翻译，译著有《任天堂传奇：游戏产业之王者归来》，并曾参与翻译图灵公司的书籍《软件之道：软件开发争议问题剖析》。 图灵社区 ID：张玳

张玳在图灵社区第一次尝试用 Running Lean 介绍的方法循序渐进地写一本书（或者证明没必要写一本书）。他给自己设定的目标是 50 个推荐：如果能收集到 50 个推荐，他就继续写下去。刊登本文的第一天，他收到了接近 20 个推荐。现在，推荐数已超过 50，他正在准备后面部分的资料，其他章节将会逐步整理好之后按顺序放出。欢迎来图灵社区查看本文完整版和这本书的最新进展。

1952 年，宫本茂出生于京都府船井郡园部町（现在属于南丹市）。这是个自然风光优美的小镇，不但有成片的水稻田，还有一座长满了绿树的小麦山，而从山下流过的则是水产丰富的圆部川。宫本茂曾经就读的小学圆部市立小学就座落在圆部川的旁边，背靠小麦山。孩提时代的宫本茂很喜欢读书和画画。他非常喜欢迪士尼的卡通人物，也常常自己学着画。放学过后，他也喜欢和附近的小朋友们一起打棒球，以及在家附近的大自然中去冒险。

当时的日本正处于二战之后的重建阶段，物资缺乏，而且园部町也属于比较偏僻的地区。不过，正因为这样，宫本茂也得以在和大自然非常亲近的环境中生活和玩耍，这成了他无尽灵感的来源。（日本儿童这种亲近自然的生活方式还将造就很多其他的东西，我们后面会讲到。）

宮本茂傳說

The Legend of Miyamoto Shigeru

宫本茂（马里奥之父）的才华是无可争议的，他在任天堂前期的游戏设计，以及后期的游戏制作两个层面都做得非常出色。在这本书中，作者希望能淡化经营层面的东西，从微观和平视的角度来看待任天堂的各大游戏系列，并进行一些背景、起源的分析。这本书将以游戏和游戏机为主线，结合宫本茂的设计思维和言论来一步步铺开。

随着年龄的增长，宫本茂的爱好也有了一定的改变，但是总是和绘画、艺术以及设计有关。上小学的时候，他看到了人偶剧团的表演，并希望自己也能够去表演人偶剧。心灵手巧的他曾经用父亲的工具自己制作木头人偶和玩具。而在上了中学之后，他又添加了一项新爱好——漫画。后来他又爱上了工程学，并希望能够成为工业设计师。

十八岁的时候，宫本茂进入了金泽美术工艺大学，主修工业设计。不过，他花在学习上的时间并不多，反而是经常花时间画漫画和玩乐队。1977 年，24 岁的他从大学毕业。在这个时候，宫本茂的志愿是做一个职业漫画家，对于电子游戏他并不十分感冒，更没有想过自己以后会做电子游戏。

毕业以后没有着落的宫本茂发现京都有一家做花札和各种玩具的公司——任天堂。不仅是玩具，他发现任天堂还做游戏机器。从小手工活就很好而且也喜欢自己做玩具的宫本茂希望能进入这家公司。这个时候任天堂并没有在招聘设计师，所以宫本茂走了个后门，请他的父亲找到了父亲的朋友，即任天堂的社长山内溥，希望能够引荐一下自己。看在这个面子上，山内溥给了宫本茂一次面谈的机会。没想到这一谈，山内溥——宫本茂职业生涯中的伯乐——发现：这个年轻人很不错！然后，他又约宫本茂谈了一次。

这一次，宫本茂准备充分。他拿着一本画册和一个大口袋到了山内溥面前，并从口袋中拿出了“儿童专用晾衣架”、“游乐园中的奇异时钟”，又向山内溥展示了“可以同时供三个儿童游乐的带秋千的跷跷板”的设计图。听了宫本茂的说明之后，山内溥被这个激情四射的年轻人的创意和设计给深深地打动了，二人一拍即合。虽然当时任天堂并不需要美术方面的人才，但是宫本茂仍然顺利地进入了任天堂，成为了任天堂的第一个美术设计师。

初入任天堂

在进入任天堂之后，宫本茂进入了企划部，主要任务是做平面出版物设计（比如海报）、设计花札的图案以及游戏机的外壳设计（图 1-1），还参与了一些玩具和模型的设计。1978 年的时候，日本的 Taito 公司推出了街机游戏《太空侵略者》（图 1-2）。这个射击外星侵略者的游戏颠覆了当时人们对电子游戏的看法，每年光是投币的收入都能达到 6 亿美元（如果计算通货膨胀的话这个数字在今天将会更大）。同样被这个游戏打动的还有年轻的宫本茂，他对电子游戏产生了兴趣。

图 1-1

图 1-1　1979 年 4 月 23 日，任天堂版的《打砖块》游戏正式发售，橙黄色的机身圆润、简洁、大方，给人一种非常温暖的感觉，非常切合这台游戏机的定位——家庭。机身底部前方有一个凸起，使之朝玩家微微倾斜，让人可以非常轻松地进行操作。这个游戏机的设计还有一个非常独特的地方，那就是左利手也可以轻松使用，只需要把游戏机转一圈就行了。这样兼具功能和美感，带着 70 年代特有的“幻想科技”味道的外观设计正是宫本茂的作品（他正是左右互利手，可能正因如此，才会把这台游戏机设计成左利手也可以顺利使用吧，后面我们还会讲到宫本茂的左右互利手对游戏设计的影响）。这也是任天堂第一次在游戏机上打上自家的标识。

图 1-2

图 1-2　京都一台仍在使用的《太空侵略者》游戏机，单人玩一次 50 日元，对战 100 日元，摇杆控制方向，红色键发射。这是日本常见的街机样式，常常放在酒吧、咖啡馆里，供客人喝酒、喝咖啡之余解闷，这样的游戏机在东野圭吾的小说《白夜行》里也有提及。（图片作者 [Tomomarusan][3]，在创作共用 2.5 协议下使用）

图 1-3

图 1-3　任天堂 1979 年推出的《太空狂热者》，是一款纯粹的山寨游戏，抄袭了当时的热门游戏《太空侵略者》，顺应了当时的潮流。和原版相比，这款游戏把敌人分成了多个颜色，然后增加了两个模式：侵略者分成两组分别朝反方向移动，以及一次只出现一行。宫本茂为这款游戏设计了人物和贴画（也许此时还算不上真正的“人物设计”）。

翌年，任天堂乘着这股热潮推出了山寨游戏《太空狂热者》。宫本茂为这个游戏做了角色设计和机箱贴画设计（图 1-3），这是有据可查的宫本茂和电子游戏设计的第一次接触，也拉开了宫本茂游戏设计师生涯的序幕。

在加入任天堂三年之后，宫本茂得到了一次和大师合作的机会。当时，任天堂的硬软件双料设计天才横井军平发明了一款使用液晶显示器的微型游戏机（即掌机）Game & Watch，并希望能够开发 G&W 版的《大力水手》。他找到了宫本茂和他一起合作。不过，天有不测风云，就在他们努力开发这个游戏的时候，任天堂的美国分部出问题了。

任天堂本打算在美国销售一款街机游戏《雷达阵地》（这款游戏是《太空侵略者》和《大蜜蜂》的结合体），但是由于日本和美国之间物流不畅顺导致货物在途时间过长，错过了最佳市场机会，任天堂的美国分部仓库中的三千台机器卖不出去。任天堂美国分部的负责人，山内溥的女婿荒川实拼尽了全力，也只卖出去一千台，还剩两千台在仓库吃灰。此时的任天堂美国分部已经耗尽了财力，没办法，只能求助京都总部，希望总部能开发新的游戏软件来替换已经过时的《雷达阵地》。

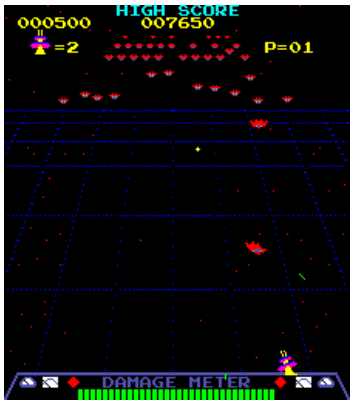


图 1-4

图 1-4　在日本国内曾经红火一时的《雷达阵地》，是一款仿 3D 的效果的设计游戏，这在当时是非常具备创新精神的。可惜，文化的不同加上物流的不畅导致了这款街机在北美市场的惨败。不过，塞翁失马，焉知非福，这款游戏涅槃之后，任天堂的新基因开始形成。宫本茂为这款游戏做了人物（飞船）设计。

山内溥很生气，但是也没办法，只好把准备在 G&W 平台推出的《大力水手》转到《雷达阵地》的平台上。可是屋漏偏逢连夜雨，大力水手角色的游戏使用权最终没有谈下来。眼看就要火烧眉毛，救星出现了。宫本茂提出了一个建议，那就是保留大力水手、奥利芙和布鲁托之间的三角关系，但是把人物换一下，特别是把布鲁托换成一只非常有爱的大猩猩。这个游戏就是《大金刚》。

图 1-5，一切都是从这里开始的……G&W 版大力水手。

大金刚

宫本茂设计方法小贴士

借鉴法。案例：《大金刚》。在做设计的初期，设计师常常很难推出百分之百自创的设计。这时候，可以借鉴已为人熟知的故事、人物等等，提供一个方向，并顺着方向进行演化、修改和调整。借鉴和抄袭的区别在于，借鉴是将一个东西（在这个案例中是1933年的《金刚》电影和《美女与野兽》的童话故事）消化之后，进行再创造的过程，而抄袭则是简单地复制。

在设计这个游戏的过程中，宫本茂使用了“借鉴法”。首先，大金刚的故事在西方已经传诵了多年，也出过小说，拍摄过多部电影，而在日本，由于也拍摄过很多关于金刚的电影，所以“金刚”二字有时候也被作为“体型硕大的猩猩”的代名词。也就是说，这是一个具有广阔知名度的故事，在需要快速出成果的状况下，借鉴这样一个故事比重新创造一个新的故事要容易得多，玩家们也更容易接受。其次，金刚、英雄、美女三者形成了三角关系，而金刚带走美女，英雄拯救美女这样的故事，正是法国作家乔治·博迪（Georges Polti）所写的《戏剧场景三十六则》中的“绑架”，是很常见也很容易被人们理解和接受的一种戏剧冲突——更重要的是，大力水手所使用的也是同一种戏剧模式。这样一来，故事情节也符合了。

单纯的复制人物和故事不能称之为借鉴，只能说是抄袭。在这一点上，日本的艺术家们非常地用心，他们常常会将一个东西完全消化之后，再根据自己的需要进行改造、加工，最后推出来一个可以称之为原创的东西。比如被誉为“漫画之神”的手冢治虫在创造阿童木这个角色的时候就参考了迪士尼的米老鼠，二者有很多相似之处，比如说都带着手套、都是四根手指（迪士尼在设计米老鼠的时候故意设计成了四根手指，因为动画动起来之后四根看上去就像是五根；这个设计在后来的阿童木动漫中被修改了过来），而且不管从哪种角度来看，阿童木头上的两个尖角都能同时看到，这一点也是借鉴了米老鼠总是露出两个耳朵这个设计。尽管有这些相似之处，我们还是很难把阿童木和米老鼠联系起来，因为二者的背景、故事、性格等等都有很大的不同。

宫本茂在设计《大金刚》的时候，采用了和手冢治虫同样的策略。他认为大力水手、奥利芙和布鲁托三个人之间并不是敌对关系，而更像是“欢喜冤家”，所以他笔下的大金刚看上去一点也不凶狠，反而很傻很天真。这也是《大金刚》英文原名叫做“Donkey Kong”的原因，Donkey 有“蠢、笨拙”的意思。此外，金刚的故事情节也被做了修改：金刚的对手不再是英雄，而是一个木匠大叔；金刚不再攀爬帝国大厦，而是跑到了建筑工地（即大叔工作的地方）上；金刚不再刀枪不入地抓飞机，而是从上往下扔木桶；金刚的外观也有所不同等等。这里面还有一个有趣的设定——大金刚其实是木匠大叔养的宠物猩猩，只是趁大叔不注意偷偷跑掉了。这样的设定进一步减弱了二者的敌对性。

《大金刚》创造了游戏史上的多个第一。首先，这个游戏是第一个有跳跃键的游戏，跳跳人儿可以用跳跃来拿木槌和躲避障碍物，并从一个平台跳到另一个平台，也就是说，这是第一个真正的平台游戏。其次，这是第一个有故事情节的游戏，此前虽然也有很多游戏有所谓背景故事，但是《大金刚》却是第一个在游戏过程中讲述故事情节的游戏。

年轻的玩家们可能会很惊讶，这也能成第一？是的，游戏其实是一个非常年轻的行业，不只是跳跃键和故事情节，其他我们已经习惯了的东西其实出现也并不早，比如第一个有背景音乐的游戏是 1980 年的《Rally-X》，第一个有过场动画的游戏则是同年推出的《吃豆人》。这些元素在电子游戏业刚刚开始腾飞的时候，是绝对的新鲜事物。

《大金刚》火了，不但拯救了岌岌可危的任天堂美国分部，还一跃成为了热门游戏，大金刚和马里奥都成为了电子游戏界的新星。■

书　榜

大数据：互联网大规模数据挖掘与分布式处理

作者：Anand Rajaraman, Jeffrey D. Ullman

译者：王斌

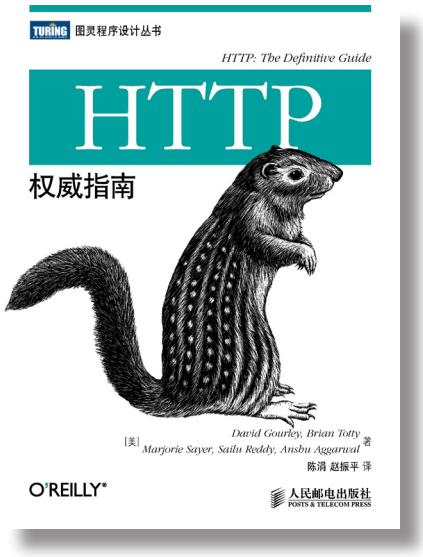
书号：978-7-115-29131-8

定价：59.00 元

图灵社区推荐：15

大数据时代的及时雨
全球著名数据库技术专家最新力作
理论与实际算法实现并重

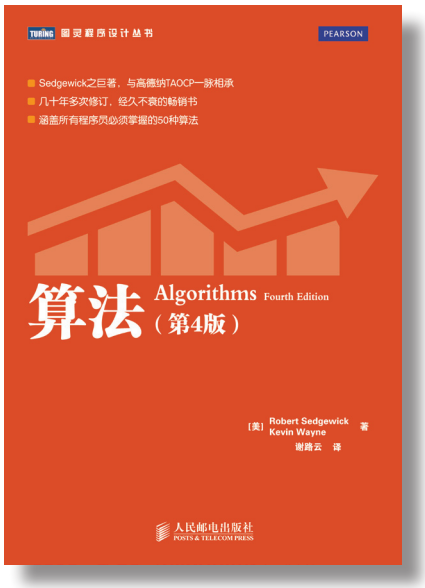
本书由斯坦福大学的“Web 挖掘”课程的内容总结而成，主要关注极大规模数据的挖掘。主要内容包括分布式文件系统、相似性搜索、搜索引擎技术、频繁项集挖掘、聚类算法、广告管理及推荐系统。其中相关章节有对应的习题，以巩固所讲解的内容。读者更可以从网上获取相关拓展材料。本书适合本科生、研究生及对数据挖掘感兴趣的读者阅读。



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：33

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；重定向及负载平衡策略。



算法（第 4 版）

作者：Robert Sedgewick, Kevin Wayne
译者：谢路云
书号：978-7-115-29380-0
定价：99.00 元
图灵社区推荐：11

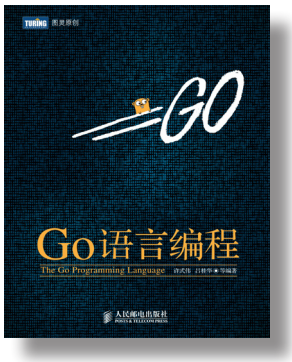
Sedgewick 畅销著作的最新版，反映了经过几十年演化而成的算法核心知识体系。全面介绍了关于算法和数据结构的必备知识，并特别针对排序、搜索、图处理和字符串处理进行了论述。第 4 版具体给出了每位程序员应知应会的 50 个算法，提供了实际代码，而且这些 Java 代码实现采用了模块化的编程风格，读者可以方便地加以改造。

30 天自制操作系统

作者：川合秀实
译者：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：19

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。特别适合作为大学操作系统课程的参考书。

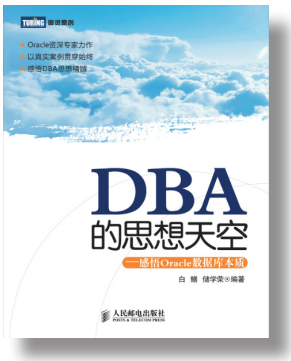
5



Go 语言编程

作者：许式伟
书号：978-7-115-29036-6
定价：49.00 元
图灵社区推荐：27

6



DBA 的思想天空——感悟 Oracle 数据库本质

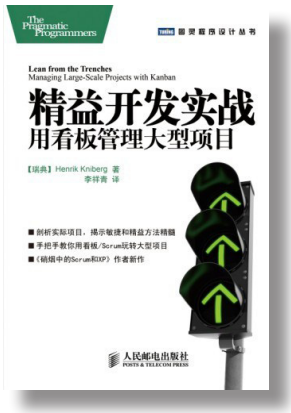
作者：白鳝，储学荣
书号：978-7-115-29443-2
定价：89.00 元
图灵社区推荐：8

7

实例化需求：团队如何交付正确的软件

作者：Gojko Adzic
译者：张昌贵，张博超，石永超
书号：978-7-115-29026-7
定价：49.00 元
图灵社区推荐：11

8



精益开发实战：用看板管理大型项目

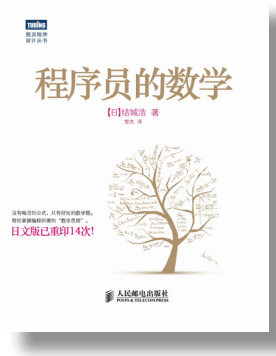
作者：Henrik Kniberg
译者：李祥青
书号：978-7-115-29177-6
定价：39.00 元
图灵社区推荐：13

9

iOS 应用开发攻略

作者：Matt Drance, Paul Warren
译者：刘威
书号：978-7-115-29178-3
定价：35.00 元
图灵社区推荐：7

10



程序员的数学

作者：结城浩
译者：管杰
书号：978-7-115-29368-8
定价：49.00 元
图灵社区推荐：14

精益 VS Scrum

——《精益开发实战：用看板管理大型项目》读书笔记

作者 / 曹刘阳
任职盛大游戏研究员，从事 HTML5、Web app、Web game 方向的研究，擅长领域：Web 前端开发、游戏开发、敏捷、项目架构。著有《编写高质量代码——Web 前端开发修炼之道》。
图灵社区 ID：阿当

编者按：笔者在阅读了《精益开发实战》（后称《精》）这本书之后，将之与作者的前一本书《硝烟中的 Scrum 和 XP》（后称《硝》）进行了对比。他结合本人的开发经历，总结出精益开发和 Scrum 的 8 点不同和联系。

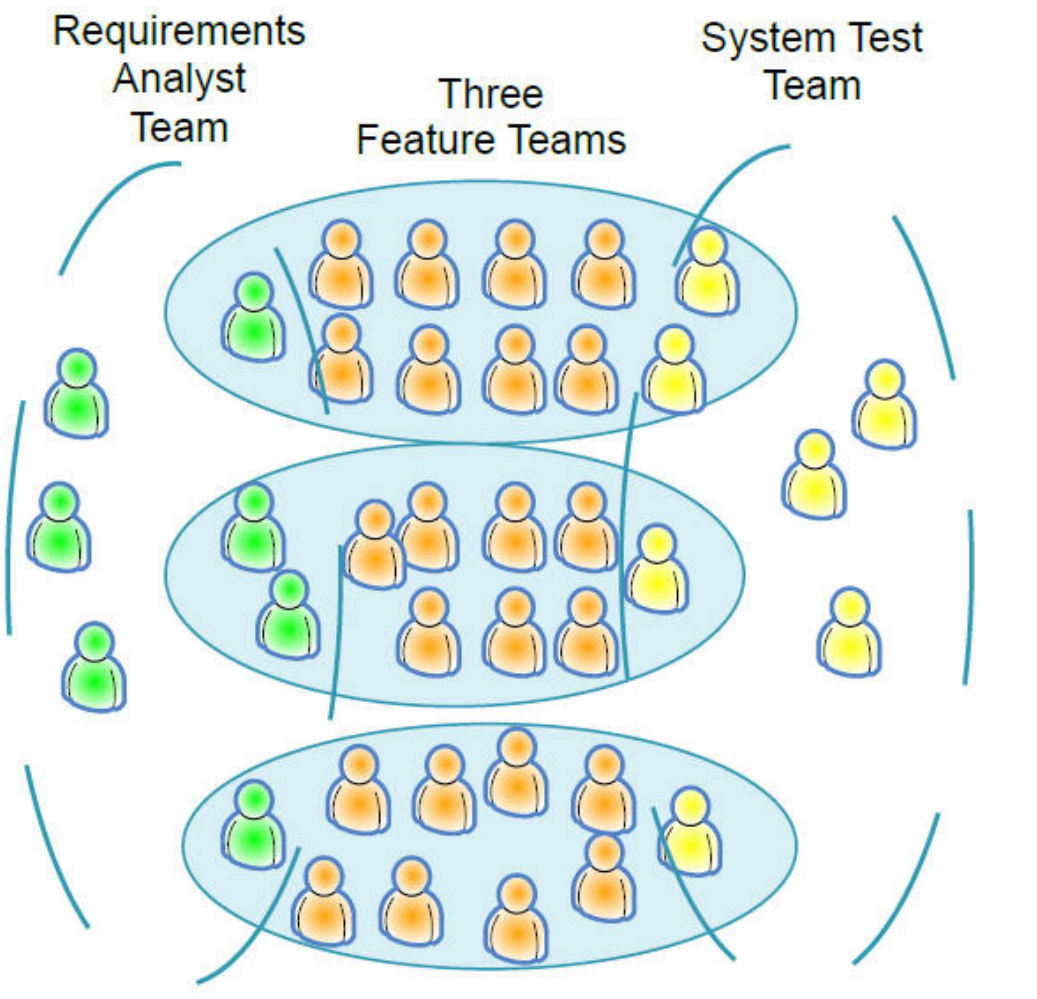
一口气读完。对比上一本讲 Scrum 的书，这本书中有很多地方值得注意：

1）团队规模 和 Scrum of Scrums

无论是Scrum之父肯施瓦伯的书《应用Scrum进行软件项目管理》，还是《硝》，都重点讲的是Scrum本身，而Scrum团队的规模都不大，按 Henrik 在《硝》中的说法，团队规模应该在 5~9 人之间，最好是 7 个人。如果人数特别多，就应该使用 Scrum of Scrums——将团队拆成几个更小的团队，每个小团队各自使用 Scrum，然后各个团队再派负责人参加团队之间的 Scrum。两本书关于 Scrum of Scrums 讲的都并不多，《硝》讲了一些，但它其实变化很大——多团队之间并不是每日站立会议，而是每周一次，全员参与，15 分钟以内，沟通的方式不像是“交流”而更像是“汇报”。

本书从实践角度展示如何使用看板管理大型项目。书中内容共分为两大部分。第一部分是案例研究，讲述看板和精益原则在具体项目中的运用；第二部分是技术详解，详细介绍第一部分提到的因果图等实践做法。

《精》中讲的案例则正好填补了这个空白，讲的是一个由 60 人组成的团队，这个大团队拆成了 5 支小团队，共同配合。本书很细致地讲解了怎么进行Scrum of Scrums——当然，它管这个叫精益，并不是典型的 Scrum。



5 支团队分别是：1 支产品分析团队，1 支测试团队和 3 支功能开发团队。

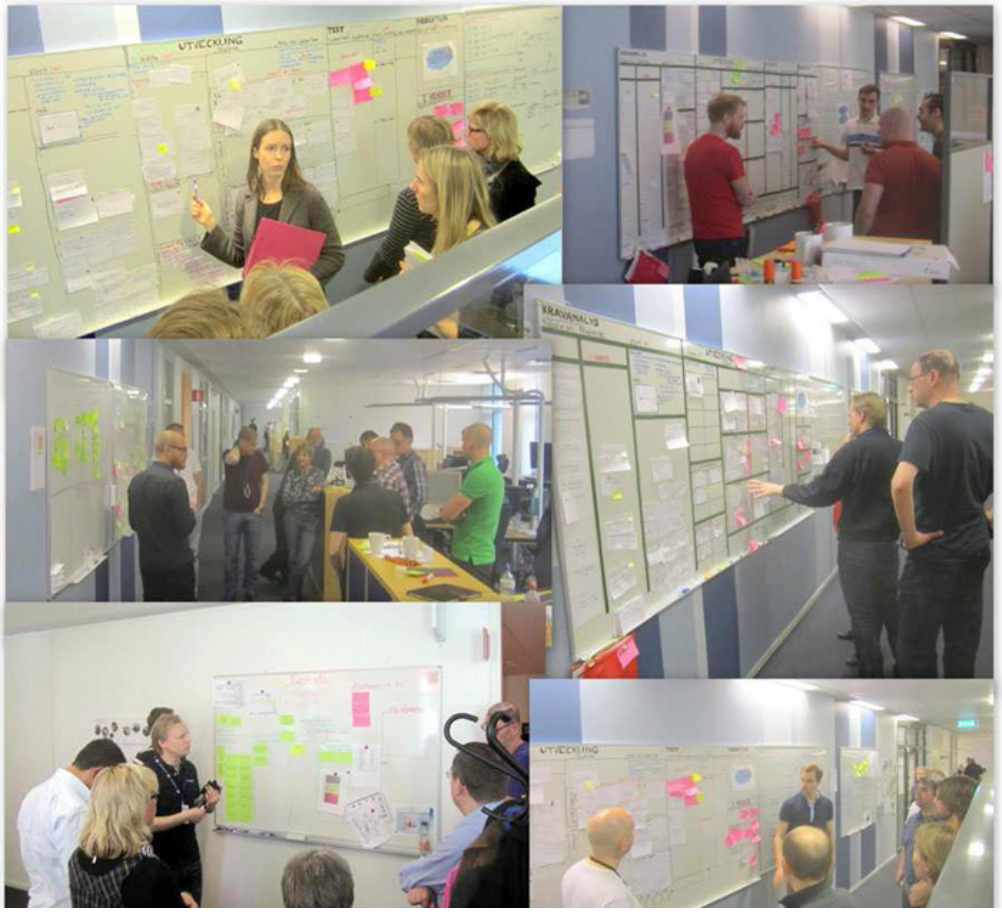
其中产品分析团队和测试团队都会派人进入功能开发团队，这些人是交叉团队，即属于垂直的功能开发团队，又属于横向的支持团队。他们是 Scrum of Scrums 的重要元素。

Scrum 中要求团队是跨职能的，所以这样的安排对于三支功能开发团队来说，是非常好的。但产品分析团队和测试团队又该做何解释呢？它们可不是 Scrum 中所说的跨职能团队呀。书上说之所以会有这两支团队的存在，是因为在产品分析和测试方面，不仅需要深入开发细节的人，也需要站在全局把握进度和方向的人。

Scrum of Scrums 的每日站立会议会开三轮，首先是 3 支垂直功能团队们开自己的站立会议——包括属于这支团队的产品分析人员和测试人员。接着是 2 支横向团队分别开会，由交叉团队的成员，分别汇报各个功能开发团队的当前进度和瓶颈，横向团队据此了解了全局的进度，进行相应的调整和计划。最后是各个团队的负责人和项目的总负责人碰头。

这三轮会议的时间是串行的，每轮会议的时间都严格控制在 15 分钟以内。也就是说，每天早上都会开 45 分钟的会，分三次开完。有些人只会出席一个会议，有些人会出席两到三个会议。其中第一轮和第两轮会议，分别有三支和两支团队同时进行，这些团队都会集中在看板前，相距不过几米，如果有问题需要跨团队沟通，走两步即可。

《硝》中每周一次的全体站立会议，从管理上更加扁平，全员参与，但这个真的感觉不到“每日站立会议”的精神了。《精》中每天早上三次早立会议，不要求全员参加，从管理上来说，让组织架构更深了，多少有点官僚的影子进来了，但至少更能感受到“每日站立会议”的精神。



我知道每日站立会议对于 Scrum 的重要性，但从没想过 Scrum of Scrums 时的壮观景象，呵呵。

2）测试团队如何融入 Scrum

关注 Scrum 的人都会问到这个问题，理论上如果一切都那么理想的话，Scrum 团队编写出来的代码应该是质量非常高的，团队成员是跨职能的，自己能做测试工作，每个轮次结束时代码都是可马上发布的。但——现实是，专业的 QA 这一环肯定是必不可少的，专业的测试人员具备一些工程师没有的技能，工程师写单元测试的确可以帮助提高一下代码质量，降低 bug 数量，但专业的 QA 是不可替代的。

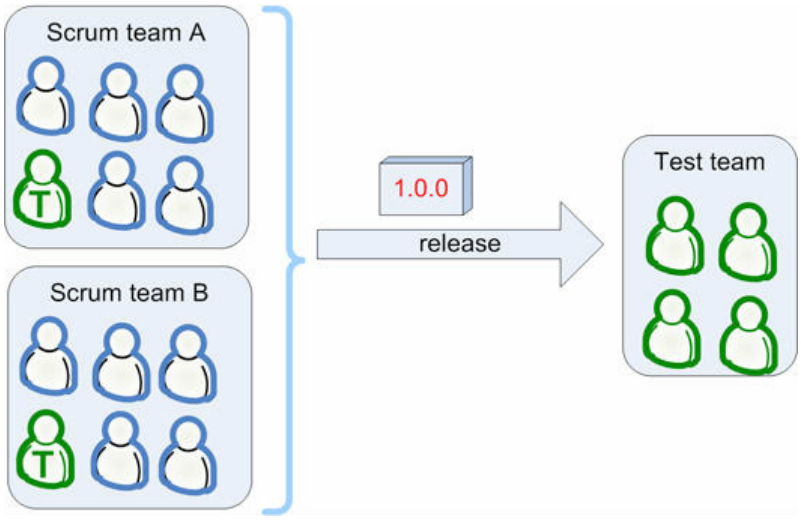
肯施瓦伯没有讲如何让 QA 融入进来，《硝》倒是有比较详细的讲解。《硝》中的建议如下。

1. 提高代码质量，尽量少产生点 bug。

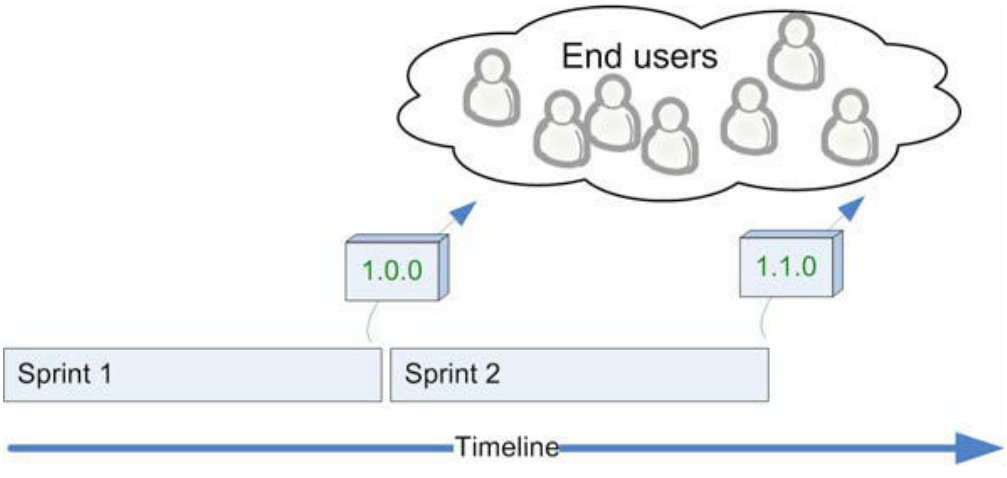
2. 每个轮次少接受一点任务，保证这个轮次完成的功能质量都很高，可发布，不抢功能，重质量，不抢速度。

3. 把测试人员放到项目组中，让他在日常的工作中就开始检验其他成员的代码，只有他检验通过的，才能放到“完成”一栏去，如果测试工程师没有事情做了，就让他做一些辅助的工作，比如组织文档，编写发布、打包之类的脚本，拆分 sprint backlog。

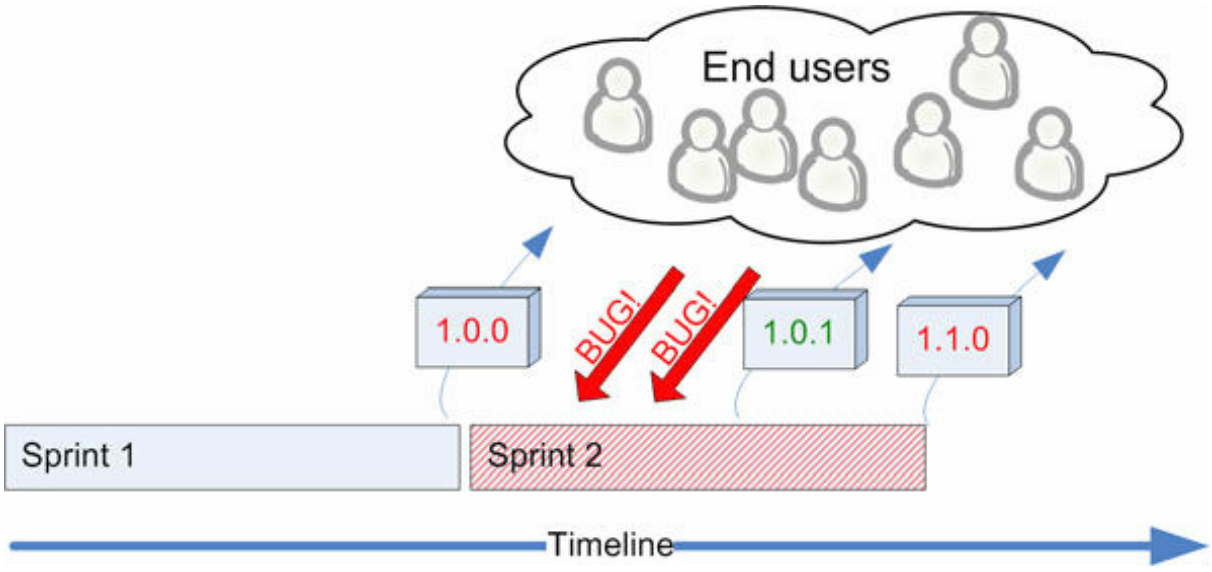
4. 除了项目组中包含测试人员，在项目开发完成后，还是需要再由测试组进行一轮测试。



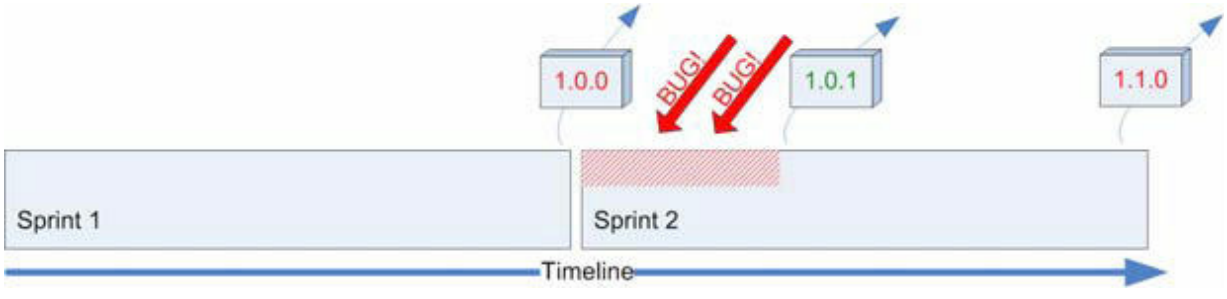
5. 在每个轮次开始时做计划会议时，都预留足够的时间用于修复上个轮次结束时遗留的 bug，理想情况下，Scrum 的情况是这样的：



但实际情况，在 sprint2 的时候，会受到测试团队反馈的关于 sprint1 的 bug 困扰。



在“开发新功能”还是“修复 bug”这个问题上，采用“可以开始构建新东西，但是要给‘将旧功能产品化’分配高优先级”的策略。

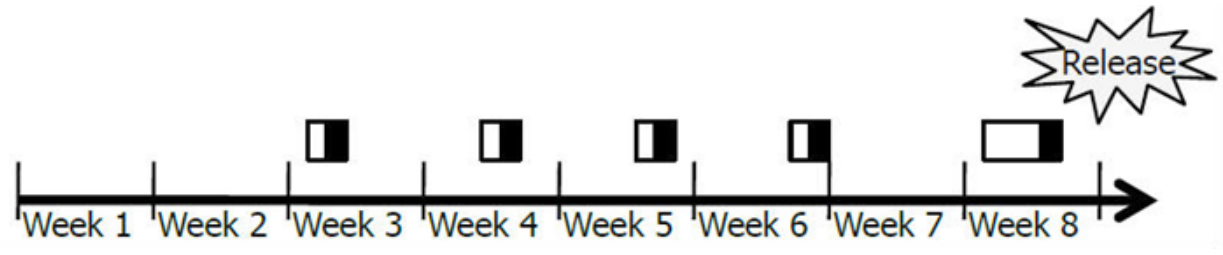


出于这个现实情况的考虑，Scrum 所推崇的“每个 Scrum 周期结束后，代码都可直接发布”，其实是很难做到的，应该说不可能做到。在这一点上，无需过多纠结。

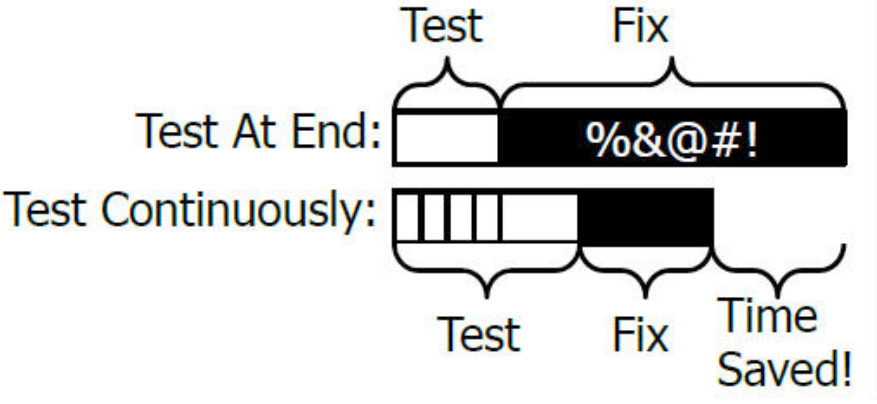
《精》在讲测试这一块时，有类似的建议，希望测试团队能定期进行测试，不要等到最后阶段，等开发团队代码全提测了再进行测试。从第一张图上，三支功能开发团队的队伍中，都含有测试人员，就能看出这点。

这种方式会招来测试人员的反感，认为工作量太大，而且代码会反复修改，并不稳定，测试工作不好做。

《精》比较了一下两种测试方式的时间成本：



白色的部分表示的是测试用去的时间，黑色的部分表示的是修复 bug 用去的时间。



由这张图可以看出，使用传统的测试方法，测试的时间是比较短，但用于修复 bug 的时间却会非常地长，因为 bug 越晚被发现，修复的成本就越高。而后者虽然测试时间变长了，但在修复 bug 上的用时却会明显变短。

3）迭代周期 VS 前十的任务

在 Scrum 中，有一个特别重要的概念就是 Sprint 周期。整个 Scrum 就是围绕迭代周期来进行的，每个周期之始进行本轮次的计划会议，周期结束时进行验收会议和回顾会议，在周期之内，进行每日站立会议。开发团队会有非常强的节奏感，重视在一个很短的周期内完成一些功能，并且在周期结束时，代码可达到发布状态。

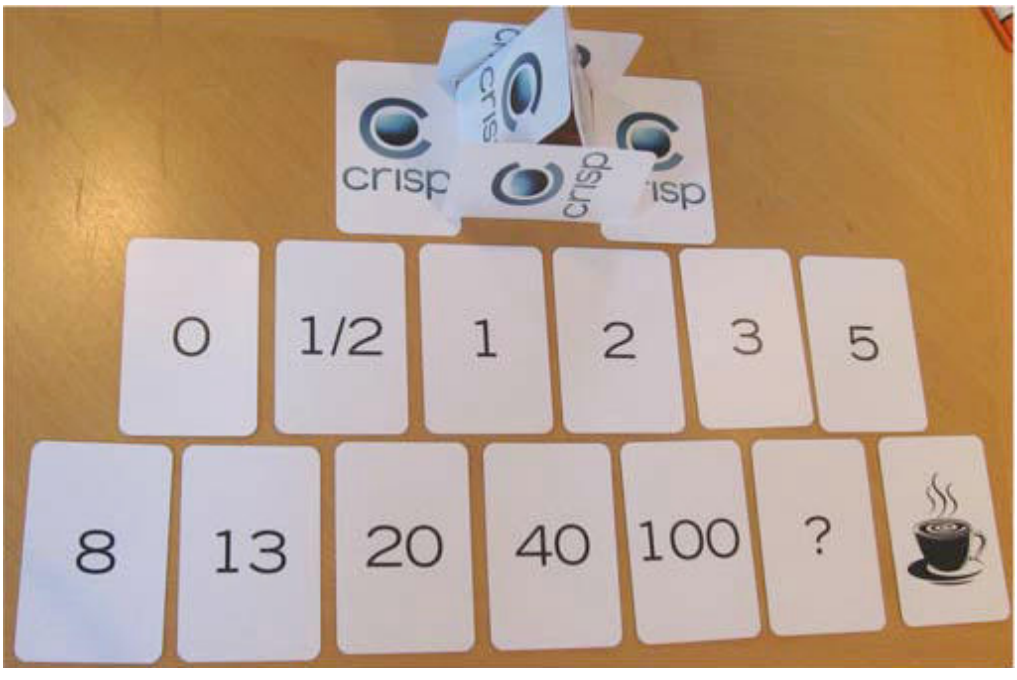
《精》在很大程度上和 Scrum 有相似的地方，比如也会开会挑选优先级高的用户故事，压入待开发队列栈，但《精》并没有 Sprint 周期的概念，不要求开发团队估算下个轮次（也就是未来几周内）要开发完成哪几个任务，不要求开发团队估算故事点。《精》认为估算故事点非常耗时，但意义却并不大。在计划会议上，唯一要做的事，就是排出接下来要做的十个任务，而这些任务什么时候完成，并没有时间上的任何估算和承诺。

这一点和 Scrum 有非常大的区别。而事实上，我更喜欢《精》的方式。Scrum 的计划会议，虽然一再对工程师强调在未来的这个周期内，大家只用做一个“估算”，这个不是“承诺”，完不成也没关系，希望借此来让工程师放松，更愉快地进行工作。但在实际执行的时候，“估算”很难不被“承诺”关联上，工程师们总是会对“估算”有压力。而且如果估算过于乐观，在回顾会议上面对没有完成的估算，沮丧感还是很强烈的。

《精》只重视优先级，并不重视“一定周期内完成多少功能”。这可以让工程师毫无压力，其实更符合敏捷的“人性化”精神。

4）估算扑克牌

《硝》和《精》中都提到了估算扑克牌。而且也都是在计划会议中使用的，所以功能大致相同。但其实两者又有区别。《硝》中讲的是 Scrum 实践，Scrum 的计划会议需要工程师们估算出 Sprint backlog 的耗时，根据团队生产力和各个 Sprint backlog 的耗时，排出本轮次可完成的任务。《硝》中的估算扑克牌就是用于团队估算每个 Sprint backlog 需要的故事点的，它的单位是数字。

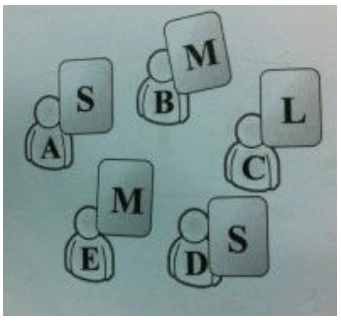


而《精》的计划会议是用于确定接下来最重要的十个开发任务。有 Scrum 中，产品 backlog 拆分成 Sprint backlog 时，Sprint backlog

的粒度有要求，不能太大，如果大了，就需要进一步拆分。有《精》中也一样，产品负责人列出的开发任务，需要被开发团队估算是不是太大了——不用像 scrum 那样，进行过于细致地拆分和时间估算，《精》并不提倡时间估算。《精》认为开发任务的大小和开发实际使用的时间没有直接关系，有可能一个小功能因为各种原因，前后花了非常多的时间，而一个大功能却可能很快就开发完成了。《精》只是需要在排开发任务时，保证这十个开发任务的粒度不要太大就行。

《精》提供的估算扑克牌，只有这么几张。

其中问号和咖啡的意思和 Scrum 中一样，但没有了代表故事点的牌，只有表示“小”“中”“大”的三张牌。产品负责人问开发团队某个开发任务的大小时，开发团队抽出自己的牌压在桌上，当大家一起亮牌时，如果意见不一致，那么进一步讨论，如果结论是开发任务粒度太大的话，那么产品负责人进一步将开发任务拆小。



扑克牌玩法还是一样，但意义已经完全不同了。

5）任务墙 VS 看板

无论是任务墙还是看板，都不鼓励使用电子系统，而是使用实体墙，这一点上两者是一样的。事实上，两者非常相似——开发团队的每日站立会议都是在这面墙面前进行的，而且墙上也都会划分“待开发”、“开发中”、“待测试”、“测试中”、“完成”等栏目，也都会配上表示进度的图表。

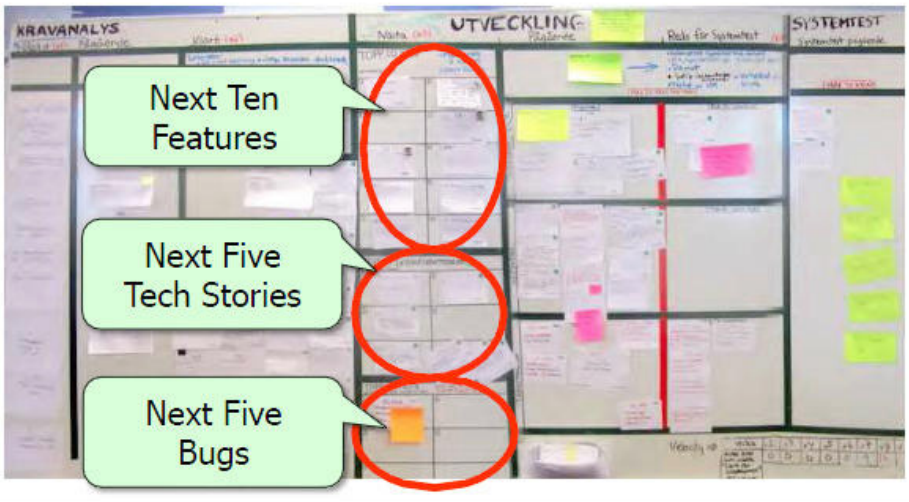
只是看板比任务墙更进一步。

1. 上面会陈列更多的任务，比如总体产品 backlog 居然也在这面墙上（根据项目需要，可以按自己意愿自由扩展这面墙的栏目），所以看板会比 scrum 的任务墙更长，更大。先看个任务墙：

再看个看板：

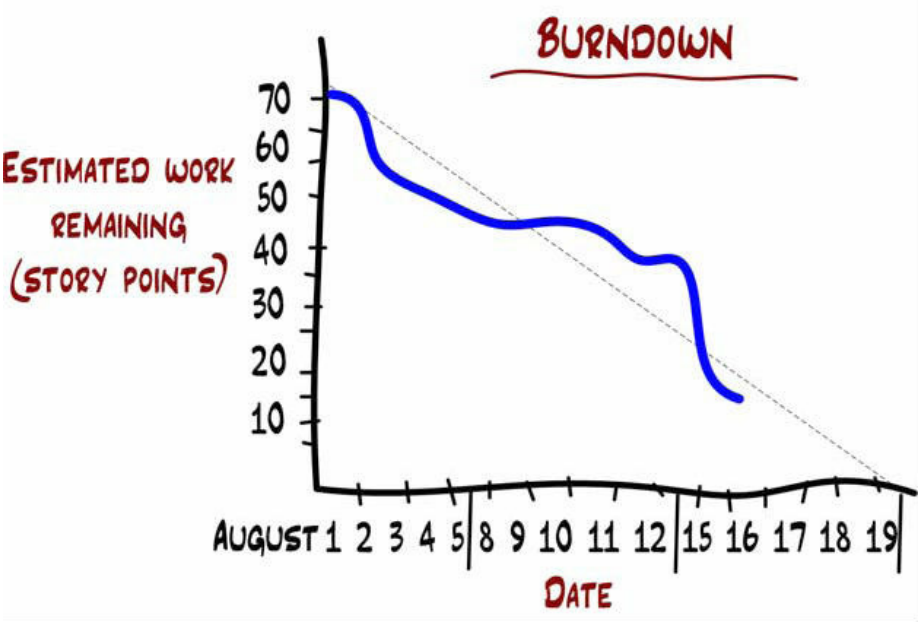
怎么说呢？任务墙是看板的一个子集。看板将任务墙这种形式带到了一个更高的高度，一种极致。

2. 看板将待开发功能进一步做了个整理，清晰地划分出“下十个新功能”、“下五个技术故事”、“下五个 bug”。“新功能”是可以看到的新的功能，“技术故事”是迁移数据库，编写自动构建脚本，重构代码等用户看不到，但技术上确定占开发量的工作，“bug”是遗留的待解决 bug。

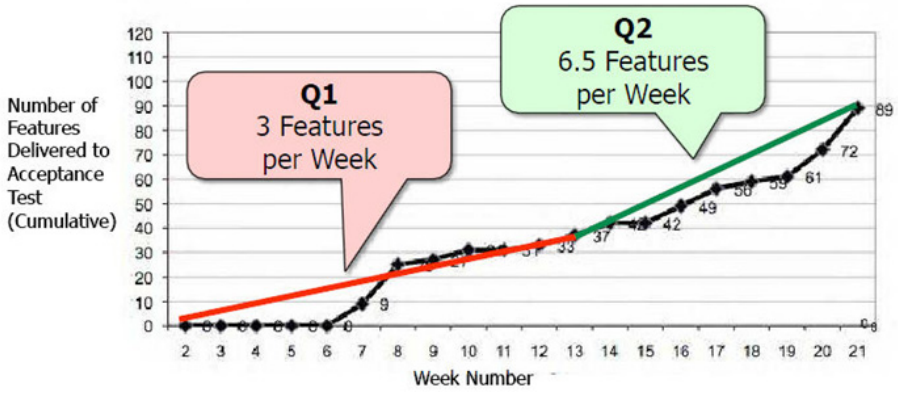


6）燃尽图 VS 开发速率图

Scrum 用一条左上至右下的斜线来表示开发进度，用每轮次多少个故事点来表示团队的开发速度。



而《精》用一条左上至右上的斜线来表示开发进度，用每周或每月多少个开发任务来表示团队的开发速度。

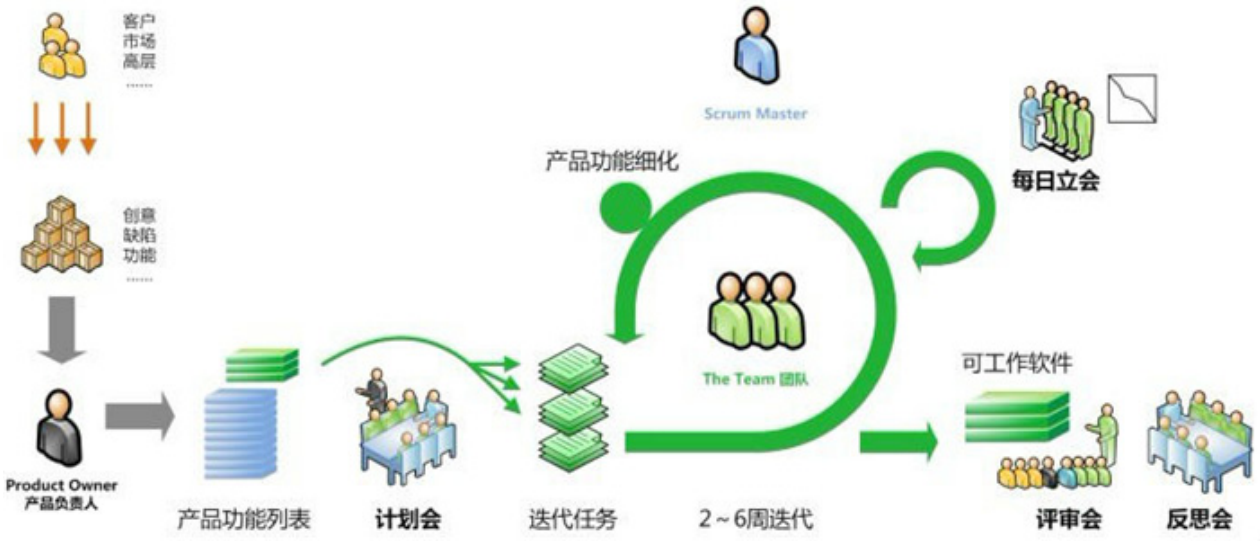


7） 发布计划

《精》和 Scrum 一样，只保证当前正在进行最重要的功能，开发的代码质量高，可持续将功能集成起来，产品可持续迭代，小步快跑地发布新功能。只保证当前在做最正确的事，但不保证整体的开发时间。所以《精》也需要在一开始就搞定领导，如果领导坚持“x 月 x 日前，xxx 功能要全部开发完成”，那就没法玩了。这应该是敏捷所有方法论都会遇到的问题。

8）Sprint 周期 VS 限额

Scrum 的核心是 Sprint，确定一个较短的 Sprint 周期，然后通过计划会议将优先级最高的需求压到最近的 Sprint 周期中，然后每日站立会议，最后验收和回顾会议。借 Sprint 周期来做项目的照明弹，保持团队的成就感和活力，同时保证新增的功能可持续集成至生产环境。



《精》没有 sprint 周期的概念，不是通过时间，而是通过限额来“降低开发团队在可视范围内的待开发任务量”的，比如看板上只能压入“最新十个新功能”、“最新五个技术问题”、“最新五个 bug”。这些“最新”都是优先级最高的，超过这个限额的，就不往看板上放。这样，可以保证开发团队不会承受过大压力。

因为 Scrum 是有周期的，而且每个周期内都有设计、开发、测试、验收，所以某种意义上来说，Scrum 的每个 Sprint 轮次都是一次小型的瀑布，很完整。而精益没有这种感觉，它更像一种源源不断的“流”，没有起点，也没有终点，任何时候看板上都填满了接下来需要开发的限额任务，但高质量的产出却源源不断。

《精》第 17 章的一组图很好地反映了看板的精髓。图不好找，建议大家自己买来看看。很棒的一本书。■

社区动态

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 12 期

APL 编程语言源代码：纪念《APL》出版 50 周年

新 Microsoft.com 的故事

不要再劝导菜鸟程序猿使用 vim 或者 Emacs 了

NGINX 发明人 Igor Sysoev 访谈

Stack Overflow 创始人 Atwood：拒绝 ToDo 列表

软件即数学的意思是?

专家审读

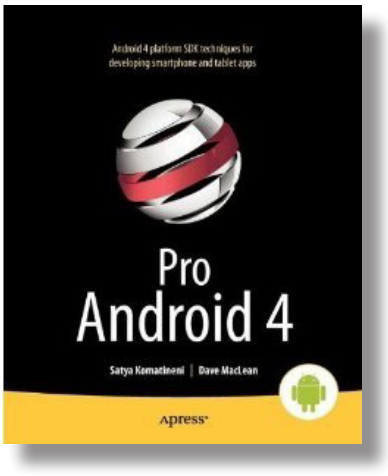
“专家审读”是图灵编辑出版流程的重要环节，即邀请图灵社区会员以专业读者身份阅读全稿，并修正之前环节未查出的问题。事实证明，这个环节能有效地保证书籍质量。

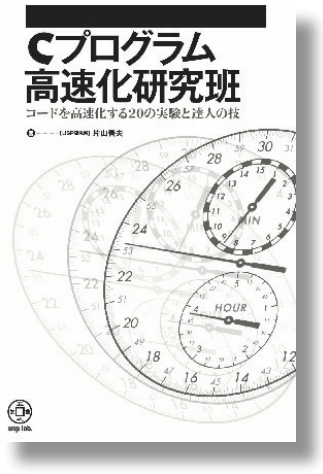
专家审读 6-8 期

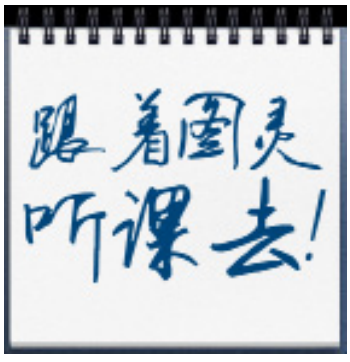
这三期共有 13 位会员加入图灵技术专家审读小组，在我们的新书付梓前，以专业读者身份审读了《明解 C 语言》、《精通 Android 4》、《C 程序性能优化：20 个实验与达人技巧》和《征服 C 语言指针》的纸稿，并给予了积极的反馈和评价，他们是（社区 ID）：

- 《明解 C 语言》：veldts、白龙、kenvi、疯人院主任、邓国平
- 《精通 Android 4》：蒋麒霖、zvivi521、简单、风林火山
- 《C 程序性能优化：20 个实验与达人技巧》：staryin，acid-free
- 《征服 C 语言指针》：zangxt，Liszt

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）。







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。

12 月，跟着图灵听课去！

1 日 Hulu/ResysChina 2012：推荐系统实践

- 图灵出版的《推荐系统实践》作者项亮与大家进行话题分享
- Facebook、百度、腾讯的技术牛人与你分享推荐系统的探索过程

4 日 -5 日 Velocity China 2012：Web 性能与运维

- 构建更快速的 Web 应用：带给你实际工作中优化网页的最佳实践，尤其是 Ajax、CSS、JavaScript、图片的性能。
- 与专家面对面：Web 性能和运维专业人士相聚的之地，向同行学习，和业界的顶尖专家进行面对面交流，将你的 Web 性能和运维水平提高到新的层次。

7 日 -8 日 SpringOne & Cloud Foundry 开发者大会：开发框架和云平台

- 热门技术：Spring 框架和 Cloud Foundry 两大热门技术，Spring 框架的最新发展，Spring 在互联网、移动设备、企业集成和大数据领域的实战应用和开发架构，Cloud Foundry 与 Spring 的集成开发，vFabric 企业级云计算应用平台等丰富内容。
- 国外牛人：活动汇集了国内外牛人，Spring 开发人员技术布道师，也是 SpringSource 的提交者和贡献者——Josh Long；CloudFoundry.com 的创始人，Java 顶尖开发人员——Chris Richardson 等。

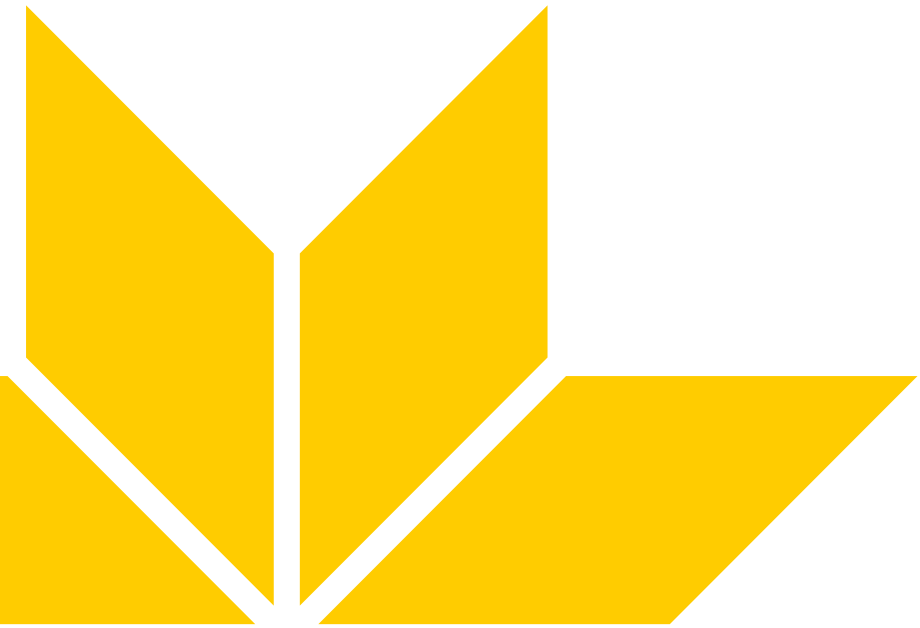
8 日 HiUED 用户体验交流会：指尖上的交互设计

- 经验分享：来自百度、腾讯、网易的设计师和产品经理，与你分享移动端交互方面的实践体会
- 好书抢先看：可以看到图灵出版的交互设计新书《亲爱的界面》、《自然用户界面设计》

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、李松峰
设计：大胖



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