译者简介

丁雪丰　网名DigitalSonic，InfoQ中文站编辑，满江红开放技术研究组织核心成员，Spring Framework 2.0 & 2.5文档翻译项目负责人。平时积极投身开源项目，是著名SFTP/SCP软件WinSCP的简体中文汉化者。参与翻译及编著的书籍有《Spring攻略》、《JRuby实战》等。

叶淮光　嵌入式软件工程师一枚，hgye@twitter。

米全喜　现从事金融软件的项目管理工作，关注质量管理和软件过程改进。译作有《SQL解惑》、《团队之美》等。

张　龙　同济大学软件工程硕士，InfoQ中文站Java社区编辑，满江红开放技术研究组织成员。热衷于编程，对新技术有强烈的探索欲，对Java轻量级框架有一定研究。目前对Mac、iPhone/iPad、Android开发、动态语言及算法产生了浓厚兴趣。翻译出版了《Dojo构建Ajax应用程序》、《Spring高级程序设计》等书籍。有5年的Java EE培训讲师经历。联系方式：zhanglong217@yahoo.com.cn。博客地址：http://blog.csdn.net/ricohzhanglong。新浪微博：http:/t.sina.com/fengzhongye，欢迎follow。

吴　珂　中国传媒大学在读研究生。刚刚结束在英国GTI公司的程序员实习生活，回到北京继续修炼。喜读书、电影、音乐、运动，果粉和Google控，热爱互联网和各类电子产品；恶不整洁的厨房和书桌，轻度强迫症。梦想成为一个伟大的程序员。要找到我，请在twitter上找@diamrem。

李琳骁　译有《Linux 命令详解手册》等多本计算机相关图书，目前从事Linux内核驱动及嵌入式系统开发。爱好Python、Vim、技术翻译、给女儿讲故事。个人网站：http://linxiao.net。

李　清　曾攻读英语专业，后辅修计算机，现在做测试开发，对自然语言和计算机语言都比较感兴趣，平时业余喜欢写点小程序玩儿。

郝培强　网名tinyfool，身高180cm，体重240斤，标准的中年老胖子。有妻有女，无房无车，现居上海，程序员，二手经济学家，现任职于盛大创新院。Blog：http://tiny4.org/blog/。twitter： http://twitter.com/tinyfool。

董金乾　软件工程师。关注分布和并发网络应用，对软件设计和开源社区有着浓厚兴趣。

戴　玮　80后SOHO宅男。硕士期间的研究方向是人工智能与模式识别，从此一直对机器学习、数据挖掘、计算机视觉怀有浓厚兴趣，曾经承担为中国医学科学院等单位开发智能系统的课题项目。编程方面喜爱C/C++、C#、Python，对Java有不明原因的厌恶感。闲暇时喜欢翻译（译言网ID是loveisp，欢迎加好友）、琢磨哲学（苏菲的世界莫非就是我们的世界？）、打篮球（不过一年多没碰过了）、玩Flash游戏（Kongregate ID是loveispdvd，欢迎加好友）、用emule下电子书（硬盘就像那什么，挤挤总会有的）。

（本书正文中脚注除说明的以外，均为译者注。）

前　言

李清　译

抛开Ada Lovelace（19世纪的一位伯爵夫人，她为Charles Babbage未完成的分析机设计了算法）的工作不说，人类在计算机编程领域奋斗的时间还不及一个人的寿命长，从1941年Konrad Zuse完成Z3电子机械计算机（首个可运行的通用计算机）算起，只有短短68年。曾有6位女性（Kay Antonelli、Jean Bartik、Betty Holberton、Marlyn Meltzer、Frances Spence和Ruth Teitelbaum）在美军的计算机部队工作，她们手工计算弹道数据表，后来被调去做ENIAC（首台通用电子计算机）最早的程序员。若从这时候算起，人类的编程历史则只有64年。在婴儿潮早期出生的人们以及他们的父母，很多至今仍然健在，他们出生时世界上还不存在计算机程序员。

当然，这些已经是历史了。现在世界上有很多程序员。劳动统计局在2008年对美国125万人进行了统计，大约每106个工作者当中就有1个是计算机程序员或软件工程师。这还没算美国之外的职业程序员、数不清的学生和业余编程爱好者，还有很多人从事其他正式工作，但却花费了一部分或很多时间来试图驯服计算机。虽然有数以百万计的人写过代码，虽然在编程出现后人们写过的代码没有数万亿行也有数十亿行，我们仍然不断地在这一领域进行创造。人们仍然在争论编程到底是数学还是工程，是工艺、艺术还是科学。我们仍然在（经常是带有强烈情绪地）争论编程的最佳方式，因特网上有无数的博客文章和论坛帖子来讨论这些问题。书店也摆满了各种论述新编程语言、新编程方法、新编程思想的书。

本书按照文学期刊《巴黎评论》（*The Paris Review*）的传统，采取了一种不同的方法来讲述什么是编程。这家期刊曾派了两位教授去采访小说家E. M. Forster，这次采访和随后的一系列问答式的采访后来辑录为*Writers at Work*一书。

我采访了15位成就斐然、经验丰富的程序员，其中有些人是系统黑客，如Ken Thompson（Unix的发明者）和Bernie Cosell（ARPANET早期实现者

之一）；有些人既有强大学术实力本身，又是著名黑客，如Donald Knuth、Guy Steele和Simon Peyton Jones；有些人是业界的研究员，如IBM的Fran Allen，爱立信的Joe Armstrong，Google的Peter Norvig，以及曾在施乐帕克研究中心工作过的Dan Ingalls和L Peter Deutsch；有些人是Netscape的早期实现者，如Jamie Zawinski和Brendan Eich；有些人参与设计和实现了现在的万维网，如刚才提到的Eich，以及Douglas Crockford和Joshua Bloch；还有Live Journal的发明人Brad Fitzpatrick（在伴随Web成长起来的一代程序员当中，他是一个当之无愧的典型）。

在采访中，我问他们有关编程的问题，问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。而且我很用心地请他们多谈谈长久以来程序员一直在苦苦思索的那些问题：我们应该如何设计软件？编程语言在帮助我们提高生产力和避免错误方面扮演了什么角色？有什么办法可以更容易地查出难以发现的bug？

这些问题远远还没有解决，所以我的采访对象持有非常不同的观点也就不那么奇怪了。Jamie Zawinski和Dan Ingalls强调尽早让代码跑起来的重要性，而Joshua Bloch则描述了在实现之前，他如何设计API并测试它们能否支持要写的代码。Donald Knuth讲述了他在编写排版软件TeX的时候，怎样在敲代码之前先用铅笔在纸上完整地实现整个系统。Fran Allen大力批判近几十年来人们躺在C语言的脚下对计算机科学的兴趣越来越低，Bernie Cosell称之为“现代计算机最严重的安全问题”，Ken Thompson却认为安全问题是程序员而不是编程语言造成的，Donald Knuth也说C的指针是他所看到过的“最令人赞叹的记法改进之一”。一些受访者对“形式化证明可能有助于改善软件质量”这一观点嗤之以鼻，而Guy Steele则漂亮地展示了这种做法的优点和限制。

然而，仍然有一些主题是大家都认同的。几乎所有人都强调保持代码可读性是很重要的。大部分受访者都认为最难查找的bug出现在并发代码中。没有人认为编程问题已经完全解决了，他们大多数人仍然在寻找编写软件的更好办法，比如怎样自动分析代码，如何让程序员更好地协作，或者寻找（或设计）更好的编程语言。同时几乎所有人都认为多核CPU的大量采用将会给软件开发带来重大改变。

这些谈话发生在计算机发展史的一个特定时刻。因此，本书中讨论的一些话题在当前是紧迫问题，今后将不再是问题而变成了历史。但即使是像编

程这种新兴领域，历史也能为我们提供很多教训。除此之外，我觉得我的受访者们可能有一些共识，包括什么是编程，如何更好地编程，等等，不仅现在的程序员会从中受益，将来几代程序员也将从中受益。

最后提一下本书的书名：*Coders at Work*。这个书名与前面提到的《巴黎评论》出的*Writers at Work*系列以及Apress的*Founders at Work*[①]（该书讲述如何创办技术公司，而本书讨论计算机编程）相呼应。我意识到编程涵盖的范围太广了，而“编码”（coding）则可以特指其中一个很窄的部分。我个人从不认为一个糟糕的程序员会是一个优秀的编码者，也不相信好的程序员会不是出色的设计者、沟通者和思考者。毋庸置疑，这些受访者都是优秀的编码者、程序员、设计者和思考者，而且还不仅仅如此。我相信接下来你在阅读他们的谈话内容时一定能够体会到这一点。希望你能喜欢这本书！

（编辑：谢灵芝）

① 中文版《创业者》由机械工业出版社出版。——编者注

致　谢

李清　译

首先我要感谢我的受访者们，他们为采访慷慨奉献了许多时间，没有他们，本书就只能是一本充满未答问题的小册子。另外要特别感谢Joe Armstrong和Bernie Cosell两家人，他们分别为我在斯德哥尔摩和弗吉尼亚提供了住所。还要特别感谢的是Peter Norvig和Jamie Zawinski，他们不仅接受我的录音采访，还帮我介绍和联系其他受访者。

我在周游世界做采访时，还收到了其他几个家庭的邀请，在此我要感谢他们的热情好客，包括波士顿的Dan Weinreb和Cheryl Moreau，英国剑桥的Gareth McCaughan和Emma McCaughan夫妇，以及我自己的父母，他们为我在纽约市的活动提供了极大便利。在采访的闲暇时间，Christophe Rhodes邀我游览了剑桥大学。他和Dave Fox还陪我吃晚饭，泡剑桥的酒吧。

Dan Weinreb不仅在波士顿招待我，并且还是本书最为勤奋的审稿人，在我准备采访名单的时候，他就开始帮我把关了。Zach Beane、Luke Corrie、Dave Walden和我的母亲也阅读了一些章节，及时给我鼓励。Zach长久以来一直为我的书提供帮助，这次则帮我想出了本书的副标题。Alan Kay建议我去采访Dan Ingalls和L Peter Deutsch，这个提议非常好。Scott Fahlman为我提供了一些有关Jamie Zawinski早期职业生涯的宝贵背景材料。Dave Walden为我寄来了一些有关Bolt Beranek和Newman的历史材料，有助于我准备对Bernie Cosell的采访。对于那些无法在此一一提及的人，我也要表示谢意和歉意。

感谢Apress的朋友，尤其是Gary Cornell，是他首先提议我来写这本书。感谢John Vacca和Michael Banks，他们提了许多好的建议。还要感谢我的文字编辑Candace English，她帮我修改了数不清的错误。

最后要深深地感谢我所有的家人。我母亲和岳母经常来帮我照顾小孩，让我腾出手做更多工作。我父母接我的妻子和孩子住了一星期，使我能够再做一次冲刺。最最感谢的是我的妻子Lily和女儿Amelia，她们是我工作的动力，我爱她们。

（编辑：谢灵芝）

1

Jamie Zawinski

李琳骁 译

名字的三字母简写与全名同样知名的黑客并不多，Lisp黑客、Netscape早期开发者和夜总会老板Jamie Zawinski，又称jwz，便是其中之一。

Zawinski十几岁就开始编程，当时受雇于卡内基·梅隆大学（CMU）人工智能实验室，从事Lisp开发。他在大学没待多久就选择了退学，因为他发现自己厌恶大学。随后近十年他一直投身Lisp和人工智能（AI）领域，阴差阳错地浸染于一种日渐式微的黑客亚文化中，而同年龄段的其他程序员则是伴着微型计算机一起成长。

Zawinski曾在加州大学伯克利分校（UC Berkeley）为Peter Norvig工作过，后者形容他是“自己雇过的最优秀的程序员”。后来Zawinski去了Lisp公司Lucid，最终领导开发了Lucid Emacs。Lucid Emacs后来更名为XEmacs，终成一大Emacs流派，堪称最著名的开源分支之一。

1994年，Zawinski最终离开了Lucid公司和Lisp领域。随后他加入当时羽翼未丰的初创公司Netscape。他是Netscape浏览器Unix版本及其后Netscape

邮件阅读器最初的开发人员之一。

1998年，作为主要推动者之一，Zawinski与Brendan Eich一道，通过mozilla.org促成了Netscape浏览器的开源。一年后，因对发布遥遥无期备感失望，他退出了该项目，在旧金山买了一家夜总会，这就是他现在运营的DNA Lounge。目前，他正集中精力与加州酒类管制局打官司，力争让这家夜总会成为各年龄层都能进入的现场音乐表演场所。

在这次访谈中，我们谈到C++为什么令人厌恶，几百万人使用其软件给他带来的快乐，以及新手程序员多动手实践的重要性。

Seibel：*你是怎么开始学习编程的？*

Zawinski：哇，多久以前的事了，都快没什么印象了。没记错的话，我第一次真正使用计算机编程大概是在八年级。当时学校里有几台TRS-80[①]，我们边玩边学了点BASIC。我记不清是不是专门开了门课，印象里好像只是课后摆弄。我记得那些机器没法保存程序，只能照着杂志或手册什么的，将程序逐行敲进去。当时我看了很多书。书中讲到的一些计算机语言，我没办法实际运行，只好在纸上编写那些语言的程序。

Seibel：*你都学了哪些语言？*

Zawinski：我记得其中一门是APL。我读了一篇讲APL的文章，觉得它非常精妙。

Seibel：*嗯，只在纸上写程序，倒是省得配专用的键盘了。你念高中时上过计算机方面的课吗？*

Zawinski：高中时我学过Fortran，仅此而已。

Seibel：*后来你是怎么开始接触Lisp的？*

Zawinski：我看了许多科幻小说，觉得人工智能实在太迷人了，计算机将统治世界。为此我学了点那方面的东西。我高中时有个朋友叫Dan Zigmond，当时我们俩互相换书看，于是一起学习Lisp。有一次，他去参加Apple用户组在卡内基·梅隆大学举办的活动。所谓活动其实就是大家聚在一起交换

---

① 这是Tandy公司于20世纪七八十年代推出的桌面微型机产品线，拥有QWERTY键盘，体积小，支持浮点BASIC编程语言。

软件，而我朋友就是想去搞点免费的东西。在那里，他还找了个大学生模样的人搭话，那个大学生说："喂，大伙来看，这里有个15岁的孩子会Lisp，真是少见。你该去找Scott Fahlman要份活干。"Dan就照做了。而Fahlman还真给了他一份活。随后Dan又说："对了，我有个朋友你也一起要了吧。"他指的就是我。Fahlman就那么雇了我们。我猜他大概是这么想的，哇哦，有两个高中生居然对这东西感兴趣，让他们在实验室里晃荡也不会有什么大碍。于是我们开始做些简单的活，比如用新版编译器重新编译整套代码。那段经历真是棒极了，就我们两个小毛孩，置身于一群研究语言和人工智能的研究生当中。

Seibel：你是在卡内基·梅隆大学才第一次有机会真正跑Lisp?

Zawinski：我想是的。我记得我们还玩过跑在苹果机上的XLISP。不过那好像是后来的事。我在CMU学会了怎样真刀真枪地编程，那时我们用的机器是PERQ工作站，它是Spice项目的一部分，使用的语言是Spice Lisp，后来演变成CMU Common Lisp。当时的环境非常奇特。那时候每周开一次例会，我们就在一边旁听，来学习软件开发是怎么一回事。不过当时那个组里有几个很有意思的"怪人"。比如Rob MacLachlan，算是我们的主管。他身材高大，满头金发，貌似野人，样子有些吓人。他平时话不多。我大部分时间都会坐在开放式隔间里干活，做点杂事，写些Lisp程序。他时常会慢慢踱进来，手拿装满啤酒的陶瓷马克杯，光着脚，然后就站在我身后。我会打声招呼，而他要么咕哝几声要么一言不发，只是站在那里看我敲键盘。有时，我正干着活，他会突然来上一句："噗哧，错了！"说完就走开了。我的感觉就像是被抛入深渊。这倒与禅法颇为相近，大师点到为止，接下来必须自己参悟。

Seibel：我给Fahlman发过邮件，他说你很有天赋，学得非常快。不过他还提到你有些不守纪律。他的原话是："我们曾委婉地教他如何与组内其他成员共事，怎么编写清晰的代码，好让自己或其他人过一个月后仍能看懂。"你还记得他们当时是怎么教你的吗?

Zawinski：过程不记得了。当然，编写自己回头还能理解的代码，这点至关重要。不过，我都快39了，而当时只有15岁，实在记不太清了。

Seibel：去CMU干活是从哪一年开始的?

Zawinski：不是1984年就是1985年。大概是从十到十一年级之间的那个夏天开始的。下午4点左右学校放学后，我会直接去那里，一直待到晚上八九点。

好像不是每天都去，反正前前后后在那里待了很长时间。

Seibel：高中毕业后，你自然是去了CMU。

Zawinski：是的。事情是这样的，我讨厌高中，那是我生命中最糟糕的日子。所以临近毕业时，我去找Fahlman要一份全职工作，他回答说："不大好办，不过我有几个朋友刚开了家公司，可以找他们谈谈。"那家公司叫Expert Technologies，也就是ETI。我猜他是董事之一。他们正在打造一个专家系统，可以自动给电话簿标页码。他们使用Lisp开发，我认识其中几个人，之前都在Fahlman的小组里待过。他们雇了我，一切顺风顺水，约莫过了一年，我开始惶恐不安：哦，天哪，得到这两份工作完全是撞大运，绝不会有下次了。一旦丢了这份工作，没有大学文凭的话，我就只能去打打零工了，看来我应该去拿张文凭。

我原本计划半工半读，一边到ETI上班，一边求学。结果却变成全工全读，前后持续了大概6个或9个礼拜。反正那段时间不短，以致我错过了退课截止期，最后学费一个子儿也没要回来。不过我上大学的时间又不够长，没拿到学分，因此要说我没真正上过大学，我也只能认了。

那段时间真的很糟。上高中时，所有人都自我安慰说："净是些没完没了的老掉牙的标准化测试，上了大学，一切都会好起来的。"结果上大学第一年，跟高中毫无区别。"哦，放心，等你念了研究生，一切都会好起来的。"所以在我看来，大学和高中一样糟糕，换了时间而已，我可受不了。每天早上8点钟起床，就开始往脑子里塞东西。比如，有门叫做外设介绍的课还非上不可，这门课教你怎么用鼠标。我找到他们说："我都在这所大学里工作了一年半，我知道鼠标怎么用。"但所有人都得上，概莫能外，"这是规定"。其他也都差不多，我实在无法忍受，索性退学了事。我觉得自己做得很对。

我在ETI干了大概4年，后来公司开始走下坡路。当时ETI用的Lisp机器是TI Explorer，那时我除了做专家系统的开发工作外，还把大块时间用在捣鼓用户界面上，还有那些Lisp机器的工作机制，我也从里到外学了个遍。我喜欢那些机器，我喜欢折腾操作系统，琢磨各个部分如何融为一体。

那时我已经写了不少代码，便找了个新闻组，发帖子找工作，还顺带提到自己写过不少代码。Peter Norvig[①]看到帖子后安排了面试。我当时的女友已经搬到加州大学伯克利分校求学，我正好可以随她而去。

① Google研发总监，《十年编程无师自通》一文作者，本书第8篇主人公。

Seibel：Norvig当时在伯克利？

Zawinski：是啊。那份工作很奇特。他们有一大群研究生在做自然语言理解方面的研究，大家基本上都是语言学家，偶尔写些程序。因此他们打算找个人接手他们编写的那些零零碎碎的代码，并整合成真正能用的东西。

这活儿对我来说相当困难，因为我没有相关背景，无法理解他们到底在做些什么。因此常常碰到这样的情形：我盯着某样东西，但完全不知所措。我不理解那是什么意思，不知道下一步该做什么，也不了解要读些什么才能真正理解它。于是我跑去问Peter。他很礼貌地回应我："你现在理解不了，这很正常，周二我有时间，到时给你讲解一下。"结果我就无所事事。于是我把大块时间都用在折腾窗口系统、摆弄屏幕保护程序以及之前出于好玩而捣鼓的那些用户界面之类的程序上。

就这么过了6个月或8个月，然后我意识到自己完全是在虚掷光阴。我什么都没为他们做，觉得自己就像在度假。后来有几次我真的忙得一塌糊涂，那种时候回想起在伯克利的那段日子，我就问问自己："你怎么会放弃那份度假般的工作？不会是脑袋短路了吧？他们可是付钱让你写屏保的！"

最后我去了Lucid[①]，当时仅存的两家Lisp环境开发商之一。我决定离开伯克利的主要原因是我觉得自己一事无成，那种感觉很糟。我周围的人都不是程序员。当然他们都不赖，我仍和其中几个人保持着友谊。只不过他们都是语言学家，比起解决实际问题来，他们对抽象的事物感兴趣得多。而我想做出点实际的东西，有一天就能指着某样东西说："瞧，这活儿漂亮吧，是我干的。"

Seibel：你在Lucid的工作成果是XEmacs，不过，你去那里一开始就是做Lisp方面的开发吗？

Zawinski：是的，我在那里做的第一个项目就是用Lisp，哦，我都记不得是什么机器了，不过应该是台有着16个处理器的并行计算机，我们使用的Lucid Common Lisp变体提供了几个控制结构，可以将创建的进程分别部署到不同的处理器上。

我做了一些后端优化工作，主要是减少创建线程的开销，从而让那台机

① 由Richard P. Gabriel于1984年创办，1994年破产，Lucid Common Lisp的所有权被Harlequin收购，后者于1999年被Global Graphics收购，随后Global Graphics将Lucid Common Lisp相关权利卖给Xanalys公司，由此催生了LispWorks公司，现在仍以Liquid Common Lisp为名在出售Lucid Common Lisp。

器可以完成有用的计算，比如实现并行Fibonacci算法，而不用再把时间耗费在为每个线程新建栈组（stack group）上。我非常享受整个过程。那是我第一次有机会使用那么奇特的机器。

在这之前我还负责把Lisp迁移到新机器上。大致过程就是，有人已经针对新架构写好了编译器后端，并且已编译好自举代码。我会拿到这个二进制文件，据称是针对这台机器的可执行代码，接着我必须剖析它们的装载器格式，以便写个简单的C程序，装载拿到的文件，将页面置为可执行，并跳转到那个页面。幸运的话，你就会看到Lisp提示符，之后即可开始手工装载其他东西。

对任何架构来说，这都是件难事，因为装载器几乎没什么对路的文档。你只能找个C程序编译一把，然后用Emacs逐字节分析，编辑其中的字节，尝试把某个字节改成零，看看有什么结果，程序会不会停止运行。

Seibel：*你刚才提到它没什么对路的文档，指的是文档不准确，还是根本就没有文档？*

Zawinski：通常都有文档，不过往往都是错的。或者文档太过陈旧，讲的是三个版本之前的东西，天知道。而遇到问题，往往就是要追究精确细节的时候。有时只是略微改了一下某个文件，装载器就不再认为那是个可执行文件了，你必须弄清楚是怎么回事。

Seibel：*这种事无处不在，从最底层的系统编程到上层API，总有些事跟你预想的不一样，或者不像文档描述的那样。碰到这种情况，你是怎么处理的？*

Zawinski：是啊，这种事情你得料到迟早会来的。你越早意识到自己手里的地图是错的，就能越快发现哪儿走错了。就拿我前面碰到的情况来说，如果要生成一个可执行文件，那好，我知道C编译器会生成一个。挑个正常的可执行文件，不断破坏，直到不可用为止。这是逆向工程的基本方法。

回想起来，我修正过的最难对付的bug，大概就是那段时间遇到的。经过一番努力，可执行文件已经能够运行，它试着引导装载Lisp，并载入了500条指令，之后就崩溃了。于是我不得不靠着S键，单步调试，试图找出崩溃的位置。不过，每次崩溃的位置似乎都不一样，摸不着规律。我开始研读这个自己并不熟悉的体系结构的汇编输出。最后我回过神来："天哪！当我单步调试时，可执行文件的执行顺序是不同的。那个bug有可能是时序相关的。"最终，我找到了真正的原因，那是台支持投机执行（speculative execution）的早期机器，它会执行两条分支路径。而在单步调试分支指令时，GDB总

是只会进入其中一个分支。GDB里有个bug！

Seibel：干得漂亮。

Zawinski：还行。结果我陷入这样的境地："天哪，现在我得调试GDB了，这我可从没干过。"临时的解决办法是碰到分支指令时，在该分支前停止，并在分支的两条路径各设置一个断点，然后继续执行。由此也刚好证实GDB真的出了问题。我大概花了一周来修正GDB，最后还是无果而终。我猜有个寄存器被改写了，导致分支检查时总得到一个正值。

于是我修改了单步调试（step-by-instruction）的命令，让它碰到分支指令时，改为不执行分支检查。这样我就可以只靠S键，最终停在分支之前，我会自己设置断点，再继续执行。当你调试某样东西时，发现不仅原本依赖的一些假设其实是错的，而且连工具都有问题，那真是很有意思。

其实GDB面对Lisp系统时本来就特别古怪，因为GDB拿Lisp代码毫无办法，后者不带任何调试信息，还是由GDB所不知道的编译器编译出来的，我猜有些平台的栈帧结构GDB根本就无法理解。当时GDB基本上就是个汇编单步调试器。因此你会巴不得马上逃离GDB的世界，越快越好。

Seibel：于是你找了个Lisp调试器，一切就全搞定了。

Zawinski：是的，没错。

Seibel：后来某个时候Lucid转变了方向，准备开发一个C++ IDE（集成开发环境）。

Zawinski：在我去那儿工作之前就已经开始了，一直在推进中。大家的工作重心开始从Lisp转移到那个叫做Energize的开发环境上。其实那个IDE很不错，只不过推出的时机不对，早了两三年。至少在Unix平台上，还没有人想到要用这种工具。现在几乎人人都在用IDE了，而那时我们不得不花大量时间向人们解释，为什么我们的工具要比vi和GCC好得多。话说回来，在那之前我已经做了一些Emacs方面的工作。我记得那个时候我已经重写了Emacs字节编译器，因为什么来着？对了，是为了我之前写的那个类似Rolodex电话/地址簿[1]的东西。

Seibel：Big Brother Database（BBDB）[2]？

---

① Rolodex名片架，Rolodex一词取自Rolling和index，详见http://en.wikipedia.org/wiki/Rolodex。

② 详见jwz的介绍，BBDB主页。

Zawinski：是的。它实在太慢了，我决定一探究竟，结果发现问题出在编译器上。于是我重写了编译器，那也是我第一次领教Stallman的固执。所以那时候，我对Emacs已经了解了很多。

Seibel：前面提到的对字节编译器的改动，是修改了字节码格式，还是只改了编译器？

Zawinski：实际上有几种方式可供选择，我对C层面和字节码解释器做了部分改动，新增了若干指令，用来加快执行速度。不过编译器可以配置成生成旧的字节码，或是生成充分利用新代码优势的新字节码。

最终我写了个新的编译器，而Stallman的回复是："我觉得这改动没什么必要。"我则回应："你说什么？它产生的代码更快。"接下来他的回复是："好吧，呃，给我发个diff文件，并解释你修改的每一行代码。""算了，我可不会自讨没趣，我之所以重新编写，是因为旧的那个太糟了。"当然这样是过了Stallman那一关的。最终这些修改被合并是因为我直接发布了自己的编译器，成千上万的人开始使用并喜欢上了这个编译器，他们不断催促Stallman，前后持续了两年，最后他被催烦了，才合并了我的改动。

Seibel：你有没有签署文件，将版权转让给自由软件基金会（FSF）？

Zawinski：有啊，我当时立马就签署了。我记得那封邮件开头就是谈签署。Stallman的回复大概是，给我发个diff文件，解释每一行改动，并签署版权转让文件。我签了字，并回邮件说："其他的我办不到，没法给你发diff文件，未免太可笑了。注释说明很详细，好好看看吧。"我猜他从没看过。

关于Lucid和FSF之间存在法律纠纷的传闻毫无根据，我们早就将修改部分的版权转让给FSF。不过，他们完全可以伺机否认我们那么做过。比如，实际上我们提交过好几次纸质文件，因为他们时不时会说："哦，抱歉，我们好像给弄丢了。"我记得大概是后来在版权转让和XEmacs上双方闹过不愉快，不过那时早没我什么事了中细节不得而知。

Seibel：这么说来你一开始就从事Lisp方面的工作。不过，显然你的整个职业生涯并不是只有Lisp，你从事的下一门语言是什么？

Zawinski：嗯，在Lisp之后，我用来正经编程的语言是C。C给我的感觉像是回到了在Apple II上编程时用过的汇编语言。PDP-11汇编器才会把它看作一门语言。总之C相当令人不快。一直以来，我总是尽可能避开C。至于C++，除了叫人反感之外，一无是处。因此我总是尽可能不用C++，在Netscape时，

我用C语言搞定一切。理由很简单，我们的目标平台都是性能不高的机器，没法很好地运行C++程序，C++程序一旦开始采用其他库，就会变得极为臃肿。此外，各个C++编译器千差万别，相互之间存在许多不兼容问题。于是我们一开始就敲定使用ANSI C，它很好地满足了我们的要求。C之后我用的是Java，感觉有点像是回到了Lisp，因为Java不存在你拼命要避开的概念，这下又自在了。

Seibel：比如？

Zawinski：内存管理。函数也更像函数而非子程序。另外，对模块化的要求也高很多。用C写代码，很容易不自觉地写个goto语句，因为用起来实在太顺手。

Seibel：现在你好像主要使用C和Perl。

Zawinski：嗯，其实我已经不怎么写程序了。通常我只写些凌乱简短的Perl脚本，用来维持服务器的正常运转。另外，我还写些简单的代码来处理些琐事，比如为自用的MP3做唱片封面之类的。这都是些短小急就、即用即抛的程序。

Seibel：你是喜欢Perl，还是因为它方便才用的？

Zawinski：哦，我可瞧不上Perl，它太可怕了。不过，它几乎无处不在。随便找台电脑坐下来，你用不着找人安装Perl即可运行自己的脚本，Perl早装上了。这是Perl值得推荐的唯一理由。

Perl拥有的库还凑合。基本上你想做什么，都有库能帮你实现。尽管通常实现得不是很好，但至少有现成的。如果用Java写了些代码，然后试着运行，结果发现在自己的电脑上安装Java遇到问题，这种体验实在令人不快，我就经历过。在我看来，Perl是门卑劣的语言。如果只用Perl很小一部分，你可以让它变得像C那样，或者更像JavaScript。Perl的语法太过古怪，数据结构一团糟。Perl的好处实在不算多。

Seibel：但至少没C++那么糟？

Zawinski：是的，绝对没有。Perl的应用场合不太一样。对于某些活儿，相比C语言，用Perl或类似语言编写要容易得多，因为Perl是面向文本的语言，即所谓的“脚本语言”(scripting language)。我个人并不赞同“程序”和“脚本”的分法，这么分毫无意义。不过，如果你要做的主要操作只是处理文本

或启动程序，比如运行wget下载HTML页面，并对其做模式匹配，那么用Perl实现显然更便捷，即使Emacs Lisp也没法比。

Seibel：*何况，Emacs Lisp也不太适合用作命令行工具。*

Zawinski：是的，虽然我过去常用Emacs随手写些小工具。实际上，在Netscape早期，我们的部分构建过程需要运行emacs-batch来处理某个文件。当然大家都不乐意用。

Seibel：*嗯，我猜他们也不会乐意用。说说XScreenSaver，你还在维护吗？*

Zawinski：我偶尔还会写些新的屏幕保护程序，权当消遣，都是用C语言写的。

Seibel：*你用什么IDE编写那些代码？*

Zawinski：大都只用Emacs。不过最近，我将XScreenSaver移植到了OS X上。具体做法是用Mac图形框架Cocoa重新实现Xlib，这样我就不用修改所有屏保程序的源代码。这些屏保程序仍旧调用X的API，不过这些API的后端是我自己实现的，主要用Objective C编写，这门语言相当不错。我很喜欢用它写程序。就优点而言，它非常接近Java，同时还比较像C。当然Objective C本质上还是C，你仍可以直接链接C代码，调用C函数，不用绕来绕去。

Seibel：*在Lucid公司工作期间，抛开Emacs开发的政治纷争，技术方面你有什么收获？*

Zawinski：毫无疑问，在Lucid期间，我成长为更好的程序员。这主要归功于我身边那些聪明绝顶的同事。在那里工作的人个个才华横溢。在那种环境下工作真是美妙无比，当别人说“真是瞎扯”或者“我们应该这么做”时，你只管言听计从就行，因为你确信他们知道自己在说些什么。那种感觉真的很棒。并不是说以前我身边的人就不聪明，只是因为Lucid那里一直高手云集。

Seibel：*整个开发队伍有多少人？*

Zawinski：估计全公司上下大概有70人，我记不太清了，整个开发队伍应该是40人左右。Energize团队好像有25或20人。我们分工明确，有人负责编译器，有人负责后端数据库，还有人负责与Emacs无关的那部分GUI工作。有一段时间，我和其他两三个人负责将Emacs集成到Energize中。最后变成主要由我负责大部分Emacs相关工作，设法将Emacs 19打造成一个稳定可靠、不易崩溃的编辑器，能真正运行用户想用的所有Emacs包。

Seibel：也就是说，你们想在自己的产品里集成一个功能齐全的Emacs。

Zawinski：我们本来不打算在自己的产品里集成Emacs。用户的机器上已经装了Emacs，然后安装我们的产品，两者互相配合，协同工作。另外，你的机器上已安装GCC，再装上我们的产品之后就能一起工作。我记得产品早期的一个开发代号叫“漫游者”（Hitchhiker），因为出发点是希望它能协调现有的种种工具，将它们揉成一体，通过提供一个通信层让这些工具能互相打交道。

不过这根本行不通。最终我们还是发布了自己的GCC和GDB版本，因为我们无法将修改快速合并到上游，或者根本就合并不了。Emacs的情况也差不多。我们不得已发布了全套工具，最终只好说：“那好，我们替换Emacs。去他的，我想我们没别的选择，那最好让它能用。”其中一项任务是确保vi模拟模式（Viper-mode）可用，那委实费了我不少时间。

Seibel：那几个礼拜是你生命中再也不想重复的吧？

Zawinski：是的，没错，充满挑战。我觉得最后结果还不错。真正的问题是模拟vi出了什么茬子，因为vi用户习惯了不断退出和重启vi。我写的部分根本不可能改变他们的思维习惯。他们大概是这么想的：“我本指望它半秒内启动好，结果却用了14秒，太可笑了，这根本没法用！”

Seibel：后来你怎么离开了Lucid？

Zawinski：Lucid完蛋了。当时裁了不少人。我给自己相识的几个人发邮件：“嘿，看样子我也得快点找份新工作了。”Marc Andreessen刚好是其中一个，他回信说：“太巧了，我们上周刚开了家公司，来我们这儿吧。”大概就是这么回事。

Seibel：于是你去了Netscape。在那里你主要做什么工作？

Zawinski：我一进公司就去开发浏览器的Unix平台部分。我去之前，其他人好像已经写了几天代码。其中Windows和Mac平台的进展稍快。总的原则是后端代码尽可能多，针对三个平台的前端代码尽可能少。

Seibel：所有代码全是从头写起的？

Zawinski：都是全新的代码。绝大部分Netscape创始人之前都是NCSA/Mosaic开发人员，他们写过不同平台的NCSA/Mosaic，实际上是三个独立的程序。那六个人都在Netscape。他们没有重用任何代码，当然他们之前写过这个程序。

Seibel：也就是说，他们就是找了个空白磁盘，从零开始写代码？

Zawinski：没错。我从未读过Mosaic的代码。实际上我们确实因为这被起诉过，NCSA声称我们重用了他们的代码，我想最后大家协商解决了。一直有谣传说我们是基于Mosaic开始的，实际并非如此。

再说了，我们有什么理由那么做？人人都想编写第二个版本，对吧？第一次编写时你绞尽脑汁，而现在有机会扔掉一切，从头开始，毫无疑问你会选择从头开始。这次会干得更漂亮。结果的确如此。其他人做的设计，基本上没法同时载入多张图片。而实际上这种功能相当重要。因此我们着力设计了后端，力求更好。

Seibel：不过，这也往往很容易陷入第二系统综合征[①]。

Zawinski：说得没错。

Seibel：你们又是怎么避免的？

Zawinski：我们大家都以最后期限为上，奉若神明。我们要么在六个月后发布最终产品，要么在不断尝试中等死。

Seibel：你们是怎么确定出那个最后期限的？

Zawinski：嗯，我们分析了业界现状，一致认定，如果我们六个月内搞不定，就会被其他公司打败，我们下定决心，一定要在六个月内搞定。

Seibel：既然事先确定了发布日期，你们就必须在产品功能或者质量上有所取舍了吧。说说你们是怎么做的？

Zawinski：我们花了很长时间讨论功能特性。其实也不是很长，不过感觉很长，因为我们有一个礼拜天天住在一起。我们毫不手软地砍掉一些特性。我们找来白板，写上自己的初步想法，然后一一划掉。好像一共有六七个人，具体人数记不清了。一群聪明、傲慢的家伙聚在一个房间，互相对着吼叫，持续了一个礼拜左右。

Seibel：这六七个人是整个Netscape还是Unix平台开发团队？

Zawinski：整个客户端团队。另外还有服务器开发人员，他们基本上就是实现自己的Apache分支。不过，我们很忙，跟他们聊得不多，也就是一起吃午

---

① 最早出自Fred Brooks的《人月神话》，意指做好第一个初步的系统之后，设计者转而开始制作第二个系统，力求更精巧、更完善，结果却因设计目标过于远大，导致第二系统不断膨胀而无法实现，连带第一个系统也被荒废。

饭而已。我们首先确定自己在项目中的角色，然后分配整项工作，负责项目每个部分的人员不会超过两个，没记错的话。1.0版本之前的团队分工大致为：我负责Unix平台，Lou Montulli实现大部分后端网络功能。Eric Bina负责布局[①]，Jon Mittelhauser和Chris Houck则负责Windows前端，Aleks Totić和Mark Lanett负责Mac前端。1.0版本之后，这些团队都增加了人手。不过我们开完会就各自回到隔间，埋头苦干16个小时，力争搞点东西出来。

那氛围真是太棒了，我打心眼儿里喜欢。每个人都认定自己是对的，我们争吵不断，不过沟通起来也快。有人会倚着你的隔间直嚷嚷："瞧你检入[②]的都是些什么东西，完全是胡来，那么做不行。真是笨得可以。"你则回敬一句："去你的！"接着去查看代码，修正之后重新检入。我们说话虽然生硬粗暴，但沟通起来也快，因为你不用先把人吹捧一番，然后才指出你认为哪里有问题，你完全可以说："全是狗屁！没法用！"这样就能迅速消除问题。尽管这压力不小，不过我们都能很快搞定。

Seibel：要快速交付软件就必须长时间、高强度地工作？

Zawinski：我们的做法当然不够健康。我只知道我们是那么做的，而且确有效果。不妨以另一种方式来回答你的问题，现实世界中，有谁能这么快就发布一款规模不小、质量不赖的软件，同时还能吃住在家，天天睡到自然醒？有过这种事吗？也许有吧，反正我没听说过。

但这并不代表越快越好。如果工作两年之后仍不厌倦，而且能够连续做上十年，那也不错。当然，要是每周工作80多个小时，很难长期坚持。

Seibel：你做过的哪件事最让自己引以为傲？

Zawinski：当然是我们发布了Netscape浏览器，整个系统。我非常专注于自己负责的部分，即Unix前端的用户界面。不过，最关键的是我们发布了Netscape，而且大家喜欢用。人们立即抛开NCSA Mosaic转投我们的产品，并感叹："哇噢，这是我用过的最棒的软件。"Netscape工具栏提供了"精彩站点"按钮，可以展示人们推荐的那些精彩站点。大概有近200个！我倒不太为我们写的代码感到骄傲，关键在于它发布了。从许多方面来看，Netscape的代码不算太好，因为时间紧，写得太快。不过它完成了任务。我们成功发布了产品，这才是关键。

① layout，指布局引擎，又称绘制引擎，如Firefox之Gecko，Chrome之WebKit，IE之Trident。

② 检入/检出是版本管理系统客户端的常见操作。

推出.96 Beta版的第一个晚上，我们都围坐在房间里，盯着不断增加的下载量，每次下载都会发出一声响声，那真是妙不可言。一个月后，两百万人用上了我们编写的软件。真是不可思议。毫无疑问，我们为Netscape做出的付出都是值得的，我们影响了人们的生活，他们的生活因我们的工作而变得更有意思、更快乐，也更轻松。

Seibel：*在这样马不停蹄的开发阶段过后，想必是要找个时机着手改善代码质量了吧。你们是怎么做的？*

Zawinski：嗯，这方面我们做得不好。我们没时间推倒重写代码。再说推倒再重写也绝非良方。

Seibel：*另外，你还开发过邮件阅读器，对吗？*

Zawinski：开发2.0版本时，Marc走到我的隔间，对我说："我们需要一个邮件阅读器。"我回答道："好啊，听起来不错。我之前做过邮件阅读器。"我当时住在伯克利，大概有几个星期没去办公室。那段时间我就坐在咖啡馆里，胡乱涂鸦，试图勾勒出邮件阅读器要实现的功能。我列出功能列表后，又一项项划掉，估算要用多长时间实现，思考用户界面该如何设计才好。

随后我回到公司，开始写代码。Marc又找到我说："对了，我们又雇了个人，他之前也做过邮件方面的开发。你们俩一起开发吧。"那个人就是Terry Weissman，那家伙太强了，我们合作很愉快。跟早期浏览器开发团队其他人共事相比，这次合作是完全不同的体验。

我们俩不会互相对着嚷嚷。真想不到我们之间的分工方式也行得通，换了别人不知道会怎么样。我已做好初步设计，开始写了些代码，每天或每两天，我们会对一下功能清单，我会说："哦，我来做这块。"他会说："好的，我做那块。"然后我们又各自忙开来。

检入代码后，我们会碰一次头，他会说："我这边都搞定了，你的怎么样了？""唔，我正在做这块。""那好，我就开始做那块了。"我们就以这种方式分配任务。最后看来效果非常不错。

我们也会有分歧，我认为只能把过滤功能仍到文件夹里了，因为时间不够，没法做好。他说："不行，我认为我们应该搞定那个。"而我则答道："时间不够用了！"结果他当天晚上就写好了。

另外，Terry和我之间很少会面，他住在圣克鲁兹，而我住在伯克利。我们到公司的距离差不多，方向刚好相反，我们俩交流并不涉及第三人，于是商量好："你不找我去公司的话，我也就不找你。""一言为定！"

Seibel：你们会发很多电子邮件吗？

Zawinski：是的，邮件不断。那时还没有即时消息，搁现在的话，估计都会用即时消息了，因为我们发的邮件往往只有一行内容。另外我们还通过电话交流。

最后我们发布的2.0版本集成了邮件阅读器，反响不错。接着我们着手开发2.1版，我认为这一版才是完整的，它将实现第一次发布时没能实现的功能。Terry和我刚做到一半，Marc进来对我说："我们刚买了家公司。他们做的邮件阅读器与你们做的差不多。"我答道："哦，好的。不过我们已经有了。"他说："对，是的，不过公司发展太快，要雇到好员工太难，有时直接收购一家公司是条捷径，那些有经验的员工都能为我所用。""那是，这些人准备做哪块？""他们会接手你们现在做的项目。""哦，真糟糕，那我得另外找点事做。"

大体情况是他们收购了Collabra，在我和Terry之上，保留了整套管理层架构。Collabra发布过一款产品，许多方面与我们的产品类似，只不过他们的产品只支持Windows，而且根本没什么市场。

然后他们赢得了创始股，并被Netscape收购。实际上是Netscape把公司的控制权交给了这家公司。结果他们不仅接管了邮件阅读器，最后收编了整个客户端部门。收购Collabra之际，Terry和我还在开发Netscape 2.1，收购之后，他们开始重写。不用说，他们的Netscape 3.0拖了很久，我们的2.1反而成了3.0，因为是时候发布个版本了，我们需要出个主要版本。

最终他们主导开发的3.0变成了4.0，而你知道，那是Netscape遭遇的最严重的灾难，几乎毁了整个公司。尽管此后公司还撑了很长时间，不过总的来说，就是我们收购的这家公司，从未取得过什么成就，却无视我们的所有劳动和成功，他们主导的软件重写，直接陷入第二系统综合征，把我们搞垮了。

他们以为，在Netscape，按他们原来那套做法就注定成功。但是，在之前的公司，他们那套做法就没成功过。结果当取得过成功的人告诫他们"注意，千万别用C++，也不要用线程"时，他们则答道："你胡扯些什么？真是什么都不懂。"

好吧，正是诸如不用C++、不用线程的决定，我们才得以准时发布产品。另外还有一点非常重要，我们从来都是同一时间发布所有平台的版本，对此他们根本不以为然："哦，百分之九十的人使用Windows，我们还是专注于Windows平台，然后再移植到其他平台。"那恰好是其他许多失败公司的做

法。如果你打算发布跨平台产品，历史会告诉你绝不要抱着“以后再移植”这样的想法。真想做到跨平台的话，就必须同时开发。所谓的移植只会令产品在第二平台上蹩脚不堪。

Seibel：4.0版是从零开始重写的？

Zawinski：他们没有从零开始，不过最后也替换了每一行代码。他们一开始就用C++。对此我极力反对，该死的是，结果证明我是对的。使用C++，一切变得臃肿不堪。另外还引入了大量兼容性问题，因为用C++编程时，没人能断定C++哪部分是可以安全使用的。有个家伙说他要用模板，结果你会发现，没有哪两个编译器实现模板的机制是一样的。

当编写的所谓多平台代码，只是支持Windows 3.1和Windows 95时，你根本就意识不到问题究竟有多严重。他们的做法令Unix平台版本成了灾难，谢天谢地，那时我已经不负责那块工作。同样，Mac平台版本也好不到哪儿去。这同时意味着产品无法再在Win16等低端Windows机器上运行。我们不得不开始削减支持的平台。或许也是时候那么做了，不过这个理由太过蹩脚。本来是完全没必要的。

让我带上个人怨恨从自私的角度来评价一下吧，整件事就是我和Terry搭建了这么棒的产品，却因成功而受罚，受罚的方式就是产品被交给一群白痴。那段时间我在Netscape非常郁闷。由此我也开始了在那儿等着被收购的日子。

Seibel：你在Netscape待了五年？

Zawinski：是的。一直待到Netscape被收购的第二年，因为被收购前一天，mozilla.org项目启动，一切又开始变得有意思起来，于是我又待了一阵子。

Seibel：你们最后是不是因使用C++而陷入困境？

Zawinski：没有，是在Java上出了问题。那会儿我们打算用Java重写浏览器。当时我们的想法是：“没问题！继续用4.0代码库会毁了公司，我们要丢弃它，只有这么做才能成功，我们清楚自己在做什么！”

不过最后还是失败了。

Seibel：是因为Java还不够成熟？

Zawinski：不是。我们这部分人又拆分成分工明确的小组，其中三个人负责邮件阅读器。最后我们做好了。邮件阅读器相当不错，速度快，还有大量很

棒的特性，不仅能妥善保存用户的数据，写大文件时也几乎没有停顿。我们还充分利用了Java的多线程，比我预想的好用。整个项目开发感觉很开心。从设计好的API来看，各方面进展顺利。

只有一块没做好，就是邮件阅读器没法显示消息。显示消息时邮件阅读器生成HTML，而显示HTML需要一个HTML显示层，结果这个显示层没搞定，到最后也没做好。页面渲染组完全误入歧途，乱套了，它们成了项目失败的主要原因。

Seibel：*他们大概一直在全力对付当时还不成熟的Java GUI技术。*

Zawinski：我不这么看。因为所有装饰效果都没问题，但是窗口中间空白一片，只能显示纯文本。他们对待项目学究气十足。他们试图从DOM/DTD着手解决问题。“哦，好，我们需要在这里再加一个抽象层，为这个代理对象的代理对象创建代理对象。最终，字符会显示在屏幕上。”

Seibel：*你好像对过度设计非常反感。*

Zawinski：是的。说好今晚发布产品，到时就必须给我发布！重写代码，让它更清晰，反复再三确实会变得非常好，这想法固然不错。但这不是重点，公司付你钱不是为了让你来写代码，而是要发布产品。

Seibel：*沉溺过度设计的人常常会说：“嘿，只要这个框架准备妥当，以后一切自会水到渠成。总的来看，我这么做其实是在节省时间。”*

Zawinski：那终究只是理论而已。

Seibel：*是的，不过有时这个理论也能成真，只要主事者有良好的判断力，框架也不是太过精致，的确能节省时间。你能讲讲自己属于哪一类吗？*

Zawinski：虽然是陈词滥调，不过我还是要重提那句话：更差就是更好。假定你花时间构建了完美的框架，满足了你的全部需求，能从1.0版一直用到5.0版，一切都很棒。猜猜结局如何？1.0版发布用了三年时间，而你的竞争对手只用六个月就发布了他们的1.0版，结果就是你出局了。你再也没机会发布1.0版，因为别人已占得先机，抢占了市场。

你的竞争对手六个月就推出的1.0版，代码质量低劣，他们可能得花上两年时间重写代码，那又怎样？他们有机会重写，而你早就丢了工作。

Seibel：*我想你一定有过彻底弃用大块代码的经历吧，很多时候，也许是期限逼近，时间紧迫，而你认为另起炉灶反而更快。*

Zawinski：是的，当然碰到过，那时你就得赶紧脱手，避免更多损失。我一直觉得这种做法有问题，不过，当你接手别人的代码时，有时自己写确实会比重用他们的还要快。因为掌握别人的代码，学会如何使用，深入理解到能够调试的程度，这些都要花上一定时间。这时你自己从头写起反而用时更短。或许这样一来，你只能完成需求的百分之八十，但这百分之八十可能才是你真正需要的。

Seibel：*就是这一点——比如有人会过来说："这个我理解不了，干脆自己重写一遍。"——导致开源软件开发中你深感遗憾的无休止的重写？*

Zawinski：是的。但撇开效率不谈，从另一角度来看，自己写代码远比弄清楚别人的代码来得有意思。因此发生你说的情形也就容易理解了。不过，整个Linux/GNOME开发介于个人爱好和产品之间。Linux/GNOME开发是不是我们用来不断试验，借以确定桌面应该是什么样子的研究项目？抑或我们是在和Macintosh竞争？到底算哪个？要想兼顾是很难的。

即使描绘得像那么回事，看上去真像有人负责做那个决定，那也根本不是真的。这一切就那么自然而然地发生了。其中一点就是所有东西总是在不断地重写，结果一件都没完成。如果你是那些开发人员之一，那不错，因为总有东西折腾，而你正好是热衷于捣鼓计算机的，但另一些人的感觉就不一样了，在他们看来，计算机只是工具，是用来帮助他们做真正感兴趣的事儿的。

Seibel：*说到捣鼓计算机本身，你现在是否仍以编程为乐？*

Zawinski：有时。我现在净干些系统管理员的脏活，我很受不了，也从没喜欢过。我喜欢做XScreenSaver的相关开发，从某些角度来看，屏幕保护程序（即实际的显示方式而非XScreenSaver框架本身）堪称完美程序，因为它们基本上都是从头写起，养眼好看，绝无所谓的2.0版本。屏保程序几无缺陷可言。它们也会崩溃，哦，有个除零错误（divide-by-zero），修复即可。

没人会要求在屏保程序里加个新功能。"我希望黄色再深点。"你不会收到这种bug报告。做出来是什么样就是什么样。这也是我总写些屏保程序找乐子的原因。屏保程序很单纯，你不用思前想后。它们不会老来烦你。

Seibel：*那么你喜欢做数学计算、求解几何和图形之类的谜题吗？*

Zawinski：是的。我想知道以这种方式显示，这个抽象的小方程式会是什么样子？或者，计算机移动方块的时候总是很生硬，那么怎样才能让他们看起

来更生动？还有，怎么处理这些正弦波，才能让它看上去更像是在弹跳？诸如此类的问题。

此外，我还会写些简单蹩脚的shell脚本，聊以自用。我知道有的工作单击3万个网页然后手工处理也能做到，但何不写个脚本，省点时间？当然这对我来说算不上编程。在那些不会编程的人看来，这像是变戏法。

我倒是很享受XScreenSaver框架移植到Mac平台的过程。实际上，整个移植需要编写大量代码，而且必须仔细考虑API和整体结构。

Seibel：那是你的API吗？你怎么组织代码的结构？

Zawinski：都有。既要弄清楚现有的API，也要确定在X11和Mac系统之间构建中间层的最佳方式。我该怎样组织这个中间层？哪部分Mac API最合适？我已经很久没有做这种活了，那感觉就是："哇哦，有点意思。看来我还是比较擅长做这事的。"

很久没这么做了，因为我已经彻底厌倦了软件行业。部分原因是商业公司和自由软件界都存在着政治上的明争暗斗。我受够了。我想做点正事，不用为鸡毛蒜皮的事在网上打口水战，也不想让产品因为自己无力插手的官僚决定而被毁掉。

Seibel：你有没有想过重新回去开发Mozilla浏览器？

Zawinski：没有。我实在不想再与人争辩，也不愿再参加Bugzilla扯皮大战。真没劲。而这些又是构建大型软件无法避免的。只要有一人以上参与开发，比如Mozilla这种项目，你就必须面对。但是我不想再卷入那种争斗。我不做这行很多年了。另外，做程序员就得给人打工，而我没必要那么做，那就不做。我终于摆脱了最糟糕的日子。要是自己开公司，我也没法做程序员，我得运营公司。

Seibel：除了有两百万人用上你的软件，你还以编程的哪些方面为乐？

Zawinski：这个问题有点难。我想大概是解决问题的过程。这不太像是谜题，谜题类游戏我也玩得不多。弄清楚如何从A点到B点，怎么让机器照你的想法去做，这是编程令人快乐的主要原因。

Seibel：你感觉到代码的美吗？美感是否在可维护性之上？

Zawinski：是，当然是。任何东西只要表达恰当，不论精练，抑或平淡，都是具有美感的。例如组织合理的语句、一幅涂鸦，或是寥寥数笔勾勒出的极其逼真的漫画，这些都有共通之处。

Seibel：*你认为编程和写作是类似的智力活动？*

Zawinski：从某些角度看，我认为是这样。当然，编程要严格得多。但就表达思维的整体能力而言，两者非常相似。不要不着边际，说出口之前先想一下准备说什么，然后尽量言简意赅。我认为这正是编程和写散文之间的共同点。

我感觉两者都使用大脑的同一块区域，不过很难准确表述那是什么。很多时候，我读的文章与糟糕的代码无异。比如大部分合同：格式死板，重复又重复。看到这些，我心想，为什么不抽取成子例程（即所谓的段落）？基本套路就是先从定义开始，然后甲乙丙丁，参照子丑寅卯。

Seibel：*好吧，接下来我们聊点编程的细节。你怎么设计自己的代码？如何组织代码的结构？不妨以你最近OS X上的XScreenSaver移植工作为例讲讲。*

Zawinski：好，首先我会随意鼓捣一番，写些短小的演示程序，那些都是最后不会再用的。比如搞清楚怎样将窗口显示到屏幕上，等等。既然我是要实现X11，第一件事就是挑个屏幕保护程序，列出它调用的所有X11函数。

接着，为每个X11调用创建一个空函数（stub），然后再开始慢慢逐个实现，弄清楚自己准备如何实现这个，怎么实现那个。

另外，在Mac平台上，需要编写启动代码。窗口怎么显示到屏幕上？某个时候，它会调用X代码。更棘手的一项任务其实是弄清楚怎样搭建构建系统，以合理的方式让它工作起来。为此我做了大量实验，代码也挪来挪去。有时，我会将这段代码放在上层，这段代码又会被它调用。然后那些代码又可能得彻底修改。总之，期间会有大量剪切和粘贴操作，直到我觉得控制流程合理为止。然后我会回头清理代码，将相关的代码放在相应的文件里。

这有点像粗线条绘制，搭建基础框架。剩下的就是一个接一个地移植屏保程序。有时遇到一个屏保程序调用了之前没用到的三个函数，那我就必须实现这三个函数。这些任务都相当简单直观。其中也有些比较棘手，比如针对在屏幕上显示文本和四处移动矩形框等操作，X11 API提供了大量选项，那段代码就会越来越乱，不过大部分都非常简单。

Seibel：*这么说来，针对每个X11调用，你都要编写相应的实现。你有没有发现自己累积了大量非常相似的代码？*

Zawinski：有，当然有。通常当你第二或第三次剪切粘贴同一段代码时，就得停下来把这段代码抽取成子程序了。

Seibel：*假定再次开发规模与邮件阅读器相当的软件，你前面提到开始会写下几段文字，列出一组功能特性，这是你着手编写代码之前所做的最细粒度的准备吗？*

Zawinski：是的。也许还会简略描述库和前端的差别。不过也可能不会。如果是独自一人开发，我不会关心这些，毕竟那部分我再清楚不过了。随后，我采取的第一步是自顶向下或自底向上开始。无论采用哪种方式，我都会先在屏幕上显示一个窗口，包含若干按钮，然后逐步深入，开始构建那些按钮的功能。或者也可以从另一边入手，先编写解析邮箱和保存邮箱的功能。采用哪种方式都可以，还可以两边齐头并进，到中间会合。

我发现尽早在屏幕上显示一些东西，有助于集中注意力去解决问题。这能帮我决定下一步做什么。只是盯着满眼的待办事项，我会手足无措，不知道先做哪项，会去为"先做哪项有关系吗"之类的问题操心。相反，如果能真切地看到一些东西，即使只是邮箱解析器的调试输出，心里也会觉得踏实。我会想：好，这一步完成了，下一步该做什么？好的，现在我或许该输出HTML或其他什么，而不是只显示树形结构。或者解析邮件头的更多细节。你会直接看到下一步该构建什么。

Seibel：*你是否经常重构以保证代码内部结构的一致性？或者你一开始就很清楚如何将代码各部分整合在一起？*

Zawinski：通常一开始就很清楚。我很少遇到这种情形："哦，我得整个推翻重来。看来我必须好好整理一番。"当然，有时还是会碰到。

编写程序的第一个版本时，我一般会把所有代码写在一个文件里。随后，我开始分析那个文件的结构，比如有几段代码非常相似。再比如那些代码已有上千行，何不把它们挪到另一个文件里。前面提到的API就是这样逐步构建起来的。毫无疑问，设计是个持续不断的过程，程序不完成，永远不知道设计是什么样的。因此我喜欢尽早实践，在屏幕上显示东西，这样我就可以边看边做。

另外，一旦开始编写代码，你就会意识到："不，这想法够烂。为什么我会认为这个模块相当简单，而实际情况却远比我预想的复杂？"这种感觉是在实际的编码工作前无法了解的，那时你会发现情况变得不受自己掌控。

Seibel：*哪些迹象表明情况正脱离你的控制？*

Zawinski：当你探究某个模块时，心里会想："哦，这得用上我半天时间，

代码量大概是这个规模。”随后你开始实现，情绪却逐渐低落：“哦，好吧，我还需要另外一段代码，我最好先去写那段。呃，好像是个大问题。”

Seibel：*我注意到优秀程序员和差劲程序员的一个重要区别是，优秀程序员在不同抽象层之间切换自如，游刃有余，修改时仍能保证各层独立，并且选择最合适的那一层进行修改。*

Zawinski：显然，决定在什么位置进行修改很有讲究，而且可能关系非常重大。设法只在靠近用户的上层直接修改，还是选择可能影响底层的大幅修改？两者可能都是正确答案，很难说哪种做法更好。我这次所做的修改只是小小的特例？或者还会碰到十几种同类情形？

对我而言，我认为最重要的一点是在构建全新的程序时，应当设法尽快写好自己能用的程序，哪怕只实现一点功能。这样你就能真正知道下一步做什么。一旦有窗口显示在屏幕上，加上了一个已具备功能的按钮，你就能大概知道接下来该做哪个按钮。当然，我这里谈的内容主要是从GUI的角度出发的。

Seibel：*我们聊过你追踪过的几个非常棘手的bug，比如GDB自身的bug。下面，我们再来谈谈调试相关的话题。对初学者而言，你推荐哪个调试工具？打印语句？符号调试器？形式化方法证明程序正确性？*

Zawinski：过去这些年里变化太大。当初我使用Lisp机器时，无非就是运行程序，停止运行，探查数据，可以使用检查器（inspector）工具浏览内存。我做了一些修改，基本上Lisp侦听器（listener）就成了检查器。因此，只要它打印出对象，就会出现一个上下文菜单，你可以点击菜单项，返回指定的值。这么一来，很容易跟踪一组相关对象和类似的东西。这就是我早期思考问题的方式。深入代码内部，来回修改，不断试验。

后来，我开始编写Emacs内部的C代码，使用GDB时，我试图沿用同样的方式。我们也是围绕那个模型来构建Energize的。不过一直没有取得很好的效果。随着时间的推移，我渐渐地不再使用这类工具，而是直接插入打印语句，再重新运行程序。如此反复，直到搞定为止。特别是在你接触越来越多比较原始的环境，这是你唯一的选择，因为那些环境根本就没有调试器，比如JavaScript和Perl这样的语言。

如今，人们似乎不清楚何谓调试器：“哦，要那玩意儿干嘛？它能做些什么，帮你插入打印语句？我搞不懂。你用的这些稀奇古怪的词到底是什么意思？”如今大多数人都用打印语句。

Seibel：这中间有多少是因Lisp和C语言的区别而非工具的区别所致？一处区别是在Lisp里你可以测试一小部分，调用你不确定工作正常与否的小函数，然后中断函数执行，检查运行状态。而C代码呢，复杂之极，必须运行整个程序，然后在某个地方设置断点。

Zawinski：与C语言相比，Lisp这类语言本身更适合调试。另外Perl、Python和类似语言在这方面也更具Lisp的特质，不过我还没看到多少人真的按照Lisp的方式去做。

Seibel：不过GDB是能帮你检查一些程序状态的。对你来说，GDB哪些方面导致它没法用？

Zawinski：我一直对GDB不满意。部分原因在于它本质上是C。浏览数组时，满眼都是数字，我不得不逐个确认，把数组元素强制转型成相应的类型。这方面GDB始终没做好，没借鉴更好的语言采取的做法。

Seibel：相反，在Lisp里查看数组时，数组元素会按其类型一一列出，因为数组知道其元素是什么类型。

Zawinski：非常对！总感觉使用GDB时要在栈里上蹿下跳，栈里的内容看起来乱七八糟。有时等你从栈的深处回来以后会发现上层的数据完全变了，通常是因为GDB出了什么错。或是，有时寄存器和栈帧完全对不上号，这样就什么也做不了了。

我一直不怎么相信调试器告诉我的东西。它会打印点东西，瞧，这是个数字。至于是否属实？我不得而知。而且很多时候，你根本没有调试信息。比如你碰到一个栈帧，看似没有参数，然后我会花上十分钟试图记起参数零放在哪个寄存器里。然后我就放弃了，重新连接，插入一条打印语句。

随着时间的推移，调试工具看起来好像越变越糟，每况愈下。不过，另一方面，现在人们终于认识到，人工分配内存的做法并非明智之举，这种做法也不再那么重要了，因为需要你深入数据结构的真正复杂的bug不会经常发生，这是因为这些bug，特别是在C中，常常可以算作内存崩溃问题。

Seibel：你们是否使用断言，或者比较不那么正式的文档编写方式，或者实践过检查不变量的方式？

Zawinski：对于怎么处理Netscape代码库里的断言，我们做过一番研究。显然，插入断言语句对调试而言是个好主意，如你所说，还有利于文档化。它能表达意图。我们用得很多。但随之而来的问题是，在产品代码中，断言失

败时会怎么样？你会怎么做？我们商定的做法是返回零，好让它继续运行下去。让浏览器崩溃的做法很糟糕，还不如返回到空循环，哪怕泄漏些内存或其他什么也好。怎样都比浏览器直接崩溃来得好。

许多程序员都有种本能："我必须呈现错误消息。"不，根本不用。没人会在意那个。这类东西在Java等语言中很容易管理，这些语言都有真正的异常系统。当顶层循环处于空闲状态时，你可以捕获所有异常，轻松搞定。不需要去打扰用户，告诉他们某个值为零。

Seibel：*你是只在调试程序的时候才会去逐句查看代码吗？还是如有些人建议的那样，写好程序之后，把逐句查看作为检查代码的方法？*

Zawinski：不会，没必要。只有调试程序时，我才单步查看代码。当然，这么做有时也是为了确认代码写得没问题。但很少这么做。

Seibel：*那你是怎么调试代码的？*

Zawinski：我会先审读代码。通读代码，直到我认为一切正常，不可能出错。然后，我会插入一些代码，尝试解决存在的问题。如果审读代码时没发现纰漏，我就会停在中间或其他什么位置，看看问题出在哪里。总之，视情况而定，很难一概而论。

Seibel：*就断言来说，你看得有多正式？有的人使用断言很随意——这里有个变量我觉得应该为真，于是就加个断言。而有的人则看得非常正式——函数有前置条件和后置条件，还有全局不变量等。你在这方面是怎么做的？*

Zawinski：我绝对不会以数学上可证明的方式考虑问题。我显然做得更随意。当然，给函数传入参数时，至少应该考虑到这些参数的取值范围，这一点很有帮助。这会是个空字符串吗？诸如此类的检查。

Seibel：*与调试相关的是测试。在Netscape，你们有专门的QA（质量保证）组，还是你们自己测试所有项目？*

Zawinski：两个都有。我们会一直让软件运行着，那是最好的一线QA。另外我们有个QA组，他们会从头到尾做详细的正式测试。每次一有新版本发布，他们就会对着清单做测试。比如转到这个页面，点击这个。你应该看到这个，或者不应该看到这个。

Seibel：*那么开发人员那一级的测试呢，比如单元测试？*

Zawinski：没有，我们从来不做那类测试。我偶尔会对某些模块做这种测试。比如邮件头日期解析器的测试用例就非常详尽。那时没人真正在意标准，因此你拿到的邮件头五花八门。不论你扔给我们什么样的邮件头，都得能解析，否则邮件排序出错只会惹恼用户。为此，我从网上收集了大量实例，拼凑在一起，得到一张巨大的链表，包含格式繁多的日期，和我认为应该转换成的数字。每次修改这块代码，我都会跑一遍测试，有些测试会失败。这时我得决定，嗯，我要不要继续修改？

Seibel：*那些测试有没有整合成自动化测试？*

Zawinski：没有，我编写代码的这类单元测试时，只有我运行单元测试，它们才会运行。后来我们开发Java重写版本Grendel时做了一些单元测试，因为那时编写新建类的单元测试非常容易。

Seibel：*回头看的话，你认为你们是因为这而受苦吗？要是对测试要求更严，开发是不是会更轻松或快捷？*

Zawinski：我不这么看。我认为如果我们在那上面花时间，只会拖延我们的进度。第一次就把它做对的好处不胜枚举。早期我们非常注重速度。即使产品不够完善，我们也得发布。当然我们可以晚点发布，质量也会更好，但那时别人早已捷足先登。

要是我们做单元测试或编写更小的模块或其他，有些进度无疑会更快。原则上这一切看似很棒。只要开发进度不是很紧，当然有办法办到。但是你要知道，“我们可是得从无到有，六个星期内搞定”，除非砍掉一些东西，否则根本做不完。而要砍掉的肯定是那些无足轻重的。显然单元测试可有可无。即使没有单元测试，用户也不会抱怨。那只是产品开发过程的一部分。

我希望你别误解，以为我暗指“只有笨蛋才做测试”。我并不是这个意思。只是优先级高低的问题。你是要设法编写好的软件呢，还是设法在下周之前搞定？你不可能一举两得。我们在Netscape经常开的一个玩笑是：“我们绝对是百分百力求高质量。我们要在3月31日前发布我们所能及的最高质量的产品。”

Seibel：*说到这个，正好聊聊软件维护相关的话题。你怎样设法理解别人的代码？*

Zawinski：我会直接一头扎进去，开始阅读代码。

Seibel：*那么你会从哪里开始呢？从第一页开始，按顺序读下去？*

Zawinski：有时会这么做。更常见的是学习如何使用某个新的库或工具包。幸运的话，你能找到一些文档，还有API。最后弄清楚自己可能会用到的那部分，或者弄懂它是怎么实现的。按这个思路一直做。或者，对于诸如Emacs的程序，有可能从底层着手。cons cell由什么组成？怎么用？然后以此为基础拓展开来。有时，从构建系统下手，你就可以了解整体的结构。让自己专注于代码有个好方法，那就是挑一项你感兴趣的任务，然后尽力搞定它。

对于Emacs这类软件，你或许可以找个现有模块，剖析实现机制。好，现在我已经掌握这段代码。然后抽取真正实现功能的那部分，就能得到样板。至此，我弄明白了这个系统的组件是什么样的，接着就可以开始把我的东西放回到系统里。基本上就是不断抽取，直到现出骨架。

Seibel：*你最终重写了Emacs的字节码编译器和部分字节码虚拟机。我们刚才讨论过重写东西为何比修正更有趣，但重写并不总是好主意。我想知道你怎么拿捏两者之间的界限？你认为之所以选择重写整个编译器，是因为重写真的比局部修正更容易吗？或者原因很简单，因为写个编译器会比较有趣？*

Zawinski：其实到最后需要重写的地步是有个过程的。一开始我只是修正缺陷，并试着做些优化。结果原有代码就不见了踪影。在原有的API消失以前，我都是一直坚持使用的。我觉得字节码编译器效果挺好。部分原因在于它是个非常孤立的模块，只有一个入口：编译并保存。

当然，我在Lucid Emacs里添加的许多东西不像重写字节码编译器那样有充足的理由。实际上，我做许多东西的动机在于让它更像Lisp机器，更像我熟悉的Emacs，而那才是我熟悉的Lisp环境。于是我添加了大量东西，设法让Emacs在许多方面不再是半吊子的Lisp，比如应该用事件对象取代包含数字的链表。用包含数字的链表来表示事件对象，这样的实现实在无趣，令人作呕。现在回想起来，那些修改当属最大的问题。那类修改导致与第三方库的兼容性问题。

Seibel：*当然，那时你不知道会出现两个Emacs。*

Zawinski：的确。不过即使没有XEmacs，也不会只有一个Emacs。仍会有两个Emacs：Emacs 18和Emacs 19，它们之间也无法避免兼容性问题。事后看来，如果早点意识到那些修改有那么大的影响，我也许会采取不同的做法。或者多花些时间，让原来的方式也能工作。大概就是这样。

Seibel：*代码可读性事关维护，前面你谈到一些编写易读代码的做法，有哪*

些特性可以让代码更易读？

Zawinski：嗯，显然是注释。写下期望是什么，实际又做了什么。如果是创建数据结构，那就描述其成员布局。很多时候，我发现这么做帮助很大。尤其是在编写Perl代码时（比如散列表），值是对链表的大量引用，要知道Perl里的数据结构很难对付。“这里是否需要右箭头？”我发现这类注释很有帮助。

我总是希望人们能多加注释，不过令人厌烦的是有些注释只是复述一下函数名。比如push_stack函数的注释为“本函数将参数推入栈中”。还真辛苦你了。

你得在注释里写点不是一目了然的东西。它是做什么用的？可以是偏上层或偏下层的描述，具体看哪个最重要。有时最重要的是，这是做什么用的？我为什么会用它？而有时最重要的是，期望的输入范围是什么？

要使用长变量名。我并不热衷匈牙利命名法，不过我认为应该使用真实的英语单词描述事物，循环迭代除外，因为它很容易辨认。总之尽可能详细吧。

Seibel：*那么结构呢，最终代码总是以某种线性的形式组织起来的，但程序实际上并不是线性的。你是以自顶而下还是自底向上的方式组织自己的代码？*

Zawinski：通常我最后是把最底层、最细节的东西放在文件顶部，尽力保持那种基本结构。然后，通常是在顶部之上，编写API注释说明。这个文件或模块最顶层的入口有哪些？对于面向对象语言，这些语言本身就支持。使用C语言的话，你就得自己明确声明。在C语言里，我常常会尽量为每个.c文件创建一个.h文件，包含.c文件的所有外部声明。没在.h文件里导出的变量和函数一律为静态。之后，如果发现哪个静态变量或函数需要被调用到，我再把它改回到.h里去。不过，你得确认有这么做的必要，不能疏忽大意。

Seibel：*在组织文件的时候，你会在文件最开始放在最底层的细节，不过那是你的思考方式吗？你会从底层开始逐步构建程序？*

Zawinski：不一定。有时我会从顶部开始，有时则从底部下手，视情况而定。其中一种方式是，我清楚自己需要这些构件块（building block），我会先把它们组合在一起。另一种思考方式则是，我心里已经大概有个谱，然后逐步实现。这两种方式我都用。

Seibel：那么假如说，你准备重新出来工作，建立一个开发团队。你会怎么组织安排？

Zawinski：在我看来，最好的安排是一个团队不超过三个或四个人，成员之间紧密合作，每天一起共事。这种做法可以按比例放大很多倍。假设你有个项目，可以切分成25个真正不同的模块。那么，你就可以找25个团队，也许少点，比如10个。只要这些团队能互相配合，我认为你能放大到多少倍根本没什么限制。最终这看起来像是多个项目，而不是一个。

Seibel：这么说来，你会有多个团队，每个团队不超过四名成员。你怎么协调这些团队呢？会找个总架构师，负责管理依赖性，调和各个团队之间的关系？

Zawinski：我们会约定好各模块之间的接口。要让模块化的做法行之有效，模块之间的接口就必须清晰且简单。顺利的话，这就意味着大家不用太多争吵就能达成一致，同时遵守模块契约也不致太难。就我的理解，要让模块之间交互自如，最佳做法就是保持模块本身真正简单，减少可能出错的途径。

至于怎么划分则完全视项目而定。某些Web应用可能划分成UI、数据库、服务器上运行的那部分，以及服务器背后的机器上运行的那部分。桌面应用程序的分工也差不多，可以分为文件格式、GUI和基本的命令结构。

Seibel：你怎么识别人才？

Zawinski：这个我不大懂。我从来都没真正雇过人。我参加过面试人，不过始终没感觉。通过面谈我能判断自己与这个人处不处得来，但是看不出他们够不够好，只通过交谈无法下结论。我总觉得这很难。

Seibel：差劲的程序员呢？有没有可靠线索可资鉴别？

Zawinski：有时会有。通常我的看法是，如果某个人是C++模板的忠实拥趸，那就离他远点。不过那也可能只是我个人的仓促判断。也许在他们使用模板的情景中，模板确实很管用。另外从我共事过的人来看，据理力争的能力也相当重要，因为我们这群人都非常好辩。在那种环境里，那种能力好处多多。当然，那和编程能力毫无瓜葛，只与人际互动有关。

Seibel：换一个团队，可能会很不利。

Zawinski：是的，当然。

Seibel：这么说来，在Netscape，你们会把事情分割开来，每个人负责软件

不同的模块。有的人认为那么做很重要，有的人则认为团队共同对代码承担责任的做法更好。你怎么看？

Zawinski：这两种方法我都用过，各有各的优势。让每个人都对全部代码负责，我认为不切实际，因为代码实在太多了。人们还是得专攻一处，有时你需要的是专家，那么做总是能解决问题。总有些代码是你熟悉的，因为那个模块的代码你刚好比其他人写得多。或者有些部分你更拿手。要是你自己不打算一直维护那些代码，其他人插手自然是件好事。因为种种原因，代码要转交给其他人维护，因此将知识传播开来自有好处。不过，还有个“好处”是可以找个人推脱责任。如果每个人都要负责全部代码，那就没人会在需要的时候提出反对意见了。

Seibel：你当过项目经理吗？

Zawinski：严格来说没有。我在Lucid做Emacs相关开发时，许多人写的模块都会被纳入Lucid Emacs。那些人其实并不为我工作，不过这有点像是管理。其中许多人资历尚浅，这种做法之所以行之有效，是因为他们做的都是自己最喜欢做的事情，而我主要是给予反馈：“好，我想要纳入这个模块，不过首先我需要其中的这几部分。”

Seibel：而你则给他们充分自由，放手让他们去做？你告诉他们你想要功能*X*、*Y*和*Z*，然后他们设法弄清楚怎么做？

Zawinski：对。如果我要决定准备发布的产品是否包含这个模块，我就得提出模块的需求。底线是那玩意真正能工作。因此我会给他们建议：“我觉得你不妨试试这种做法，效果会比那种好。”不过我想要它工作，但又不想自己动手写。要是他们想采用某种疯狂的做法，只要能搞定，那也没问题，因为这能实现我第二个想法：我不用自己动手写。不过，大部分情况下，我给他们的反馈只是，这能工作吗，行得通吗。

Seibel：另一方面，当你还是资历尚浅的程序员时，你导师做了哪些对你有帮助的事情？

Zawinski：我觉得关键在于他们能意识到什么时候该提升员工的等级。我为Fahlman效力时，他总是给我安排一些无聊琐碎的活，最后才安排了些相对重要的任务，其实也不是那么重要。

Seibel：我记得你提到过Rob MacLachlan，他路过时会说：“错了！”是不是也会有人多一些鼓励和指导？

Zawinski：嗯，他不完全是野人一个。实际上，他也会给我一些指导。我记得自己最后阅读了大量代码，不停地问问题。我认为有一点非常重要，就是不要害怕自己的无知。如果你理解不了某件东西的工作原理，那就找做这块的人问。许多人都害怕这么做，但那样毫无裨益。不知道某件东西并不代表你笨，只是暂时还不知道罢了。

Seibel：*你阅读代码主要是因为你正在开发相关功能，还是说你只是想探究它是怎么工作的？*

Zawinski：后者，只是到处看看，我想知道它是怎么工作的。把东西拆开看看的冲动是将人们带入这一行当的一大原因。

Seibel：*你小时候真的像那些拆过烤面包机的孩子一样？*

Zawinski：是的。我还做了个电话机，学会了用罐头盒做的电报键拨号。小时候，我从车库甩卖摊上淘了不少旧书还有其他什么的，比如上世纪30年代出的《男孩自己的科学书》(*Boy's Own Science Book*)，我记得自己从中得到很多乐趣。实际上，那是上世纪二三十年代的黑客文化，这些书会告诉你怎样在自己的房间和车库之间拉条电话线，怎么制作莱顿瓶。

Seibel：*正好我要再问个标准问题：作为程序员，你认为自己是科学家、工程师、艺术家、手艺人还是其他什么角色？*

Zawinski：这个嘛，肯定不是科学家和工程师，两者都有非常正式的内涵。我数学搞得不多，也不画蓝图，也不做什么证明。我觉得自己处在手艺人和艺术家之间，具体看是什么项目。我写过不少屏保程序，那不是手艺，而是在制作精美图片，算是跟艺术沾点边。

Seibel：*在你自学的时候，你觉得自己是在学计算机科学，还是只是学习了编程？*

Zawinski：嗯，这些年我的确学了些计算机科学，但目标是学习编程。让机器做事情是目标，计算机科学则是达成目标的手段。

Seibel：*你是否认为那是种缺失，有没有过但愿自己曾更系统地学习过计算机科学的想法？*

Zawinski：有时的确会那么想，尤其是在Lucid工作期间，当时听到那些家伙讨论的，整个是大黑洞，我完全不懂，因为之前我根本不需要知道这东西。过后，我会去了解术语，对他们讨论的东西有个基本的概念，如果那正好是

我需要知道的东西，我可能还会查阅一些相关资料。因此，有时的确会有那种想法，特别是早期，我会想："天哪，我什么都不懂。"这只会叫人窘迫，还会让人缺乏安全感。当年我还是个年轻小伙，周围却全是拥有博士学位的同事，"啊，我什么都不懂！真是傻瓜一个！我怎么会混到这里来？"

要是多花点时间在求学上，我的生活的确会全然不同，不过那时我做了我该做的。

Seibel：*你有过相反的感觉吗，觉得身边的计算机科学家并不像自己那样懂得编程的真谛？*

Zawinski：我经常有那种感觉，不过，"哇，你们这些家伙搞错对象了。"这种想法其实并不多，更多是觉得："哇，我们只是兴趣不同。"我不想当数学家，但也不会去批评数学家。

奇怪的是人们常常混淆这两种追求，将深入研究计算机科学理论的人和做桌面应用的开发人员等而视之。其实两者之间并没太多可比性。

Seibel：*你主要是靠自学的。对那些自学的程序员，你有什么建议？*

Zawinski：这个问题很难回答，如今的情况大不相同。谈起"我是这么做的"，我总会觉得不自在。我不知道自己的做法是否正确。但人们总是听成"像我那样"。

我不知怎么就跌跌撞撞做了程序员。我当时做的一些决定，导致了其他决定，环环相扣，最终成了现在的我。

我不时收到邮件，内容大体是"我想成为程序员，我该怎么做？"或是"我该不该上大学？"我怎么回答得上？要是在1986年问我，我也许还会有不错的答案。不过，现在人们已不可能循着我当年的路走，因为那条路已经无迹可寻。

换在十年前，我会说第一要务是学习汇编语言。你得理解机器运转的根本。现在这还重要吗？我真不知道。也许还重要，但也许完全不重要了。如果今后十年软件的方向就是Web应用，或是一段分布式代码，寄存在某个租来的计算集群上，在十几个不同的Google服务器之间迁移，派生一些副本，得到结果后再融合在一起，人们还需要懂汇编语言吗？是否会抽象到完全和汇编语言无关的地步吗？我不得而知。

当我发现有人拿到计算机科学学位却从未写过C程序，我倒是着实惊呆了。他们开始学的是Java，就没想学点其他的。这真是荒诞不经，错得离谱。但我也不确定。也许这并没有错。或许那是野人的落伍想法："在

我那个年代，我们编程靠的是九伏电池和沉稳的双手！”

Seibel：*计算机方面的书呢？哪些计算机科学或编程书籍是每个人必读的？*

Zawinski：老实说，计算机方面的书我读得并不多。我一直都推荐的一本书是《计算机程序的构造和解释》，许多人都怕读这本书，因为Lisp味太浓了。不过，这本书在不教语言的前提下教授编程，我认为做得非常到位。我发现大量入门课程或书籍只关注语法，在高中的课堂上，在卡内基·梅隆大学（我只待了几个月）的入门课上，我都亲身经历过。

这不是在教大家编程，而是在教大家分号该放在什么位置。光是这种做法就会吓走不少人，因为教的东西太无聊。即使那些知道他们用意的人也会受不了。

还有一本书，名字叫什么来着，是关于调试的，微软公司的员工写的[①]。那本书主要介绍如何有效地使用断言。我当时觉得那真是本好书，不是因为我从中学到了什么，而是你巴不得自己那班白痴同事都读过那本书。

还有本书叫《设计模式》，人人追捧，奉为圭臬。不过，在我看来，这本书一派胡言，给人的感觉好像编程只需剪切粘贴就能搞定。你不用全盘考虑要做的任务，只要看看这本“配方书”，找个有几分相近的模式，直接套用就行了。那根本就不是编程，而是涂色书。不过，似乎许多人都对这书着了魔。参加各种会议时，他们嘴里不时蹦出从书中读到的术语。比方说，反转-翻转-两次后空翻模式（the inverse, reverse, double-back-flip pattern）等。哦，你指的不是循环吧？是的，就是循环。

Seibel：*有程序员必须具备的关键技能吗？*

Zawinski：嗯，好奇心，把东西大卸八块的好奇心。渴望弄明白底层是怎么回事。我认为那是技能之根本。缺了好奇心，就如同在浮沙之上筑高塔。好奇心是你获取知识的主要途径。把东西拆开，用心研习，你才能做好自己的东西。至少我是这么做的。计算机方面的书我读得很少。我的经验主要来自不断地挖掘源代码和参考手册。首先确定目标，没问题，实现这个目标我需要弄清楚这东西都做什么。然后，我会随意折腾一番，直到确定方向为止。

Seibel：*你读过Knuth的《计算机程序设计艺术》吗？*

---

① 疑为微软出版社1993年出版的、Stephen A. Maguire的*Writing Solid Code - Microsoft Techniques for Developing Bug-free C Programs*，中译本为《编程精粹：编写高质量C语言代码》。

Zawinski：没读过。这也许是我真正该读的一本书。可惜我一直没读。

Seibel：这本书很难读，需要很好的数学功底才能真正读懂。

Zawinski：数学我可不在行。

Seibel：有意思。很多程序员都有数学背景，许多计算机科学理论都离不开数学。这么看来，数学也并非不可或缺，你就是很好的例子。想成为优秀程序员，得具备多少数学知识或数学方面的思维？

Zawinski：嗯，那要看你怎么划分，什么算是数学的，什么不算。擅长模式匹配算得上数学在行吗？从直觉上把握数量级和组合数学很重要。不过，我要是参加相关科目的入门测试，肯定考不及格。我已经很久没做过那么正式的东西。

实际上我只上过高中数学课。我学过代数，学了点微积分，但学得不太好。还好勉强过了那门课。我上了高中物理课，我们学习力学，做实验，比如在砂纸或其他物体上拖动方块。在实验课上，我非常投入，如痴如醉，我真的很喜欢那门课。我做实验很有一手，步骤准确无误，但是一遇到数学运算就傻眼了。

我算出来的结果差了三个数量级。我交上自己的作业，但不知道哪里做错了。我只拿到一半学分，收集到的数据虽然准确，但后面的计算有误。数学向来就不是我的强项。

不过，我还不至于认为做程序员不需要数学。很显然，世上的编程种类繁多。没有除我之外的那些人，一切都将是空中楼阁。但是，比起数学，我总觉得编程与写散文有更多共通之处。这就好比你正在写故事，尝试向非常愚钝的人——词汇有限的计算机——表述观念。你已经掌握自己想表达的观念和用于表达的那些工具，你会使用哪些词，序论和总结陈述会写成什么样？编程也大抵如此。

谈到散文，口味问题就凸显出来了。用一段文字描述某样东西，可以描述得体，也可以描述出彩，很有特色。这些同样适用于程序。程序可以只是完成任务，或者，只要组合得当，还能做到容易理解。

Seibel：为什么口味很重要？只是为了让自己满意，还是从实践角度来说优美的代码更好？

Zawinski：在很大程度上，优美和容易维护可以划上等号，或者说息息相关。一篇作品之所以优美，原因之一在于组织合理，容易理解。各项事实是在文

前一一摆明，还是散布在文章各个段落？要是回头查阅，比如浏览一本书，你能否快速找出自己隐约记得的要点位于全书什么位置？“大概在书里的中间部分，作者在那儿讲到了这一点。”或者，你要找的东西散布在全书各个地方。编程的情形与此非常相像。

Seibel：*你是否认为，如今在编程上能获得成功的人已不同往常？*

Zawinski：当然，现在已经不太可能从无到有编写没有任何依赖的程序。工具包、函数库、框架等数量激增，即使最简单的软件也要用到这些。程序被分解得支离破碎。现在，一切都开始变成Web应用。编程的方式完全变样了。

总之，这样一来，快速掌握别人的代码并弄清楚其用法，这项必备技能变得更加重要。“这个我理解不了，干脆自己写一个”的做法过去很管用。不管这个想法好坏与否，你还是可以这么去做。但现在，这么做要难得多。

Seibel：*我在想，现在拆卸东西以理解方方面面是不是也需要更多耐心。试图读懂自己面对的每段代码，那几乎是没有尽头的。现在你必须有勇气说：“它的工作机制我略懂一点，我打算先放一放，等到更急迫时，再好好弄明白。”*

Zawinski：是的。正因为如此，我的第一反应是，这样成长起来的一代程序员根本不知效率为何物，也不会知道程序真正分配了什么东西。当他们意识到“哦，我的程序越来越臃肿”时，他们会怎么做？他们不知道从哪里下手。那只是我的第一反应，或许我野人一个，落伍了。也许那根本就无关紧要，只要多配点内存就行了。

Seibel：*也许人们将真正学会从更复杂的角度思考那些问题的真意。比如，这里分配六个还是四个字节也许真的无关紧要，真正要紧的是我们有没有限定程序的大小，好让它能部署到集群的一个节点上，不用占两个节点。*

Zawinski：没错，千真万确。从那个角度来看，我认为编程的确已不同往昔。你以前关注的事情不一样。以前，你会集中精力计算字节数，还有“我的对象有多大？也许这个地方需要做特殊处理，毕竟数组头部要计算在内”，诸如此类的顾虑。现在没人再关心这些。对正向和反向指针做异或运算从而指向同一个字，这类奇技淫巧在现代程序员眼里形同巫术，他们会想怎么会有人那么做，真是疯狂。其实眼下需要掌握的这组完全不同的技能，过去也要用到，只不过现在更显重要。我认为，如今能够探究API，找出自己需要哪部分，哪部分用不到，这才是重中之重。

Seibel：要是你现在只有13岁，看到现在编程的方式，你还会被编程吸引吗？

Zawinski：这太难回答了。我不认识13岁大的孩子，也不知道当下的世界变成什么样了。如今的东西更难拆卸。现在10岁大的孩子不会再像我小时候捣鼓电话那样，拆开自己的手机，弄清楚话筒是怎么工作的。况且，现在的手机里头也没什么用户可自行调整的部件。

我觉得正是这些折腾捣鼓，比如拆开磁带仓，看看里头的齿轮是怎么啮合在一起的，这种探索吸引我走上了编程之路。在我看来，除了玩乐高机器人，现在人们已经没什么机会循着我当年那条路成长。不过，也许我是错的，我刚说过不认识13岁大的小孩，也不知道他们玩什么玩具。倒是有层出不穷的视频游戏，还有许多带遥控的装置。至今我还未看到过真正让人眼前一亮的建造类玩具。真叫人沮丧。

Seibel：另一方面，编程本身也变得更容易。只是让计算机做些常规任务的话，你根本不必一开始就掌握晦涩复杂的汇编语言。

Zawinski：没错。我觉得今天的孩子想要编程可以从搭建Web应用或编写Facebook插件等开始。搭建LiveJournal的Brad Fitzpatrick是我朋友。当初写LiveJournal时，他只是出于好玩，写了个Perl脚本，这样他和朋友就可以用来写些“我准备吃午饭去了”之类的留言。他开始的方式就是写个简短的Perl脚本，然后放到Web服务器上。这种现象也许会愈演愈烈。

（编辑：谢灵芝）

2

Brad Fitzpatrick

丁雪丰 译

Brad Fitzpatrick是所有受访者中最年轻的一位，也是其中唯一一位从未在没有因特网或个人电脑的世界里生活过的。他出生于1980年，很早就开始了自己的程序员生涯，5岁时就在一台自制的Apple II克隆机上学习编程。在十几岁时，正好赶上因特网革命的大潮，他一头扎入其中，在高中时就建立了自己的第一个商业网站，在进入大学前的那个夏天创立了著名社区LiveJournal。

LiveJournal的日渐流行迫使Fitzpatrick走上了学习构建可伸缩网站的艰难之旅，期间他和他创办的Danga交互技术公司里的程序员们开发了几个开源软件，其中包括memcached、Perlbal和MogileFS，现在被用于很多世界上最繁忙的网站的服务器上。

Fitzpatrick是个典型的极有才华的世纪之交的Web程序员，他的主要编程语言是Perl和C，需要时也会用Java、C++、Python、JavaScript和C#。他做的所有编程工作基本都与网络相关，比如为网站构建更好的后端基础设施，

设计协议和软件来让博客阅读软件获知博客更新，甚至为他的手机编写代码以便在摩托车上就能自动打开车库门。

我们将谈到他在读著名儿童系列丛书*Clifford the Big Red Dog*的年龄就开始学习编程，为什么能够很高兴地一边念大学，一边运行LiveJournal，以及他是如何学会不惧怕去阅读他人的代码的。

Seibel：你是怎么成为一个程序员的？

Fitzpatrick：我父亲曾在Mostek[①]工作。这个公司是制造内存的，他对电脑很感兴趣。他做了台Apple II电脑，材料几乎都是多余的废弃部件。他和我母亲坐在电视机旁把部件焊起来，这个工作花了他们好几个月，只是把它们焊起来而已。然后我父亲从公司里拿了些不能卖的ROM，这些ROM有一位或几位不能用，有的在高位，有的在低位。后来不知怎么着他们弄到了Apple II ROM，接着就不停地将ROM烧到无法工作的芯片上，直到找到一块能用的为止，损坏的那位正巧是好的。最终，他和他的一帮同事终于做成了自制Apple II。我差不多从两岁起就在上面玩或者看他编程。

Seibel：他是个程序员还是个硬件工程师？

Fitzpatrick：他是个电气工程师，偶尔也写写程序。我五岁时他就教我编程，搞笑的是我六七岁时就超过他了。我母亲说我是一边读*Clifford the Big Red Dog*，一边读从图书馆借来的Apple II程序员手册。我会把“变量”念成“贝量”。我早期的一些记忆就是和父亲一起编程。比如他把我拖进厨房，在纸上写下一段程序，问我：“你觉得这段程序是什么意思？”我记得那程序好像是“10 PRINT HELLO, 20 GOTO 10”。

Seibel：那么说你是从BASIC开始的？

Fitzpatrick：是的，就是BASIC。我当时还不能使用鼠标、高级图形模式和彩色，直到我们家的一个朋友向我介绍了C并给了我Turbo C。那年我大概八岁或者十岁。我父亲在1984年去了Intel，我们就搬去了波特兰。他帮助设计了386和486，现在仍在Intel。我们总是能有新的有趣的电脑。

Seibel：那你有没有试过汇编语言呢？

Fitzpatrick：我在计算器上做过些汇编，比如TI计算器上的Z80，但仅此而已。

---

① 集成电路制造商，成立于1969年，在其巅峰时期曾占据全球85%的DRAM内存芯片市场份额。

Seibel：你还记得是什么吸引你开始编程的吗？

Fitzpatrick：我不记得了，只是好玩吧。我母亲不得不限制我使用电脑，好让我出去和朋友们一起玩。我的朋友们会跑过来说："Brad又在玩电脑。他太无聊了。"我母亲则会对我说："到外面去玩吧。"

Seibel：你还记得写的第一个比较有意思的程序吗？

Fitzpatrick：我们以前有台Epson打印机，它配有几本又大又厚的手册，手册最后是程序员指南。我就在Apple上写了点东西，我可以在高级图形模式下画些东西，当程序完成绘制（线段、图案或别的什么）之后，按下Control C，在后台一个不会显示的帧缓冲区里键入一段内容，加载另一个程序，它会读取屏幕并打印出来。

在那之前，我记得还写过一个程序，每当我敲击一个键，它就移动稿面，我按退格稿面会向回退，这样打字时就感觉像是在用打字机一样了。

这是我的第一个程序，好比方K是抓取的下一个字符，如果K等于a，打印a；如果K等于b，打印b。我几乎处理了每个字母、数字和一些标点。后来一个念头一闪而过："等等，我可以说'打印一个变量！'"然后用1行代码替换掉了40行。"天啊，这太棒了！"对一个六岁的孩子来说，这已经是抽象能力的极限了。

那些都是比较有意思的早期作品。到了中学，我开始开发游戏，为朋友们制作图形编辑器和关卡编辑器，他们会把图形做进关卡中，随后我们再把它卖给其他同学。我记得我不得不检测EGA和VGA。如果VGA跑不了，就退回到EGA模式，使用另一组适合当前屏幕的贴图，为此我们必须为所有的东西做两组图形。学校的人会出大概5美元买它，然后安装，接着发现它不能用，他们的家长就会给我爸妈打电话大喊："你儿子拿那没用的东西从我孩子那里骗了5美元。"我母亲就开车带我过去，坐在死胡同里等我进同学家调试并修复我的程序。

Seibel：那段日子里你有没有上过关于编程的课？

Fitzpatrick：没有。就是从图书馆里借了一两本书，然后随便玩玩。当时没有真正的论坛或因特网。后来我连上了一个BBS，但上面并没什么内容，它没有联上因特网，只是一群人在玩棋类游戏而已。

Seibel：你的学校有AP[①] C.S.（计算机科学）或类似的课程么？

① Advanced Placement项目，在美国和加拿大的高中提供大学级别的课程。

Fitzpatrick：呃，我们没有AP C.S.课程，但我们有计算机程序设计课。有一个老师在上这门课，不过我都可以在教室后面讲高级课程了。他们还在用我写的图形编辑器和图形库，他们要做的项目是开发一个游戏。我偶尔还会碰到那个计算机老师，他是我家的一个朋友，我会在我哥哥的足球比赛上看到他，他会说："是啊，我们还在用你的库呢。"

我的确参加了AP C.S.考试，那是在考试从Pascal换成C语言前的最后一年，一年后又从C换到了Java之类的语言。我不懂Pascal，所以去附近有AP C.S.课程的高中上了一些夜间班，大概三四次吧。后来我找了本书来学Pascal，我把大多数时间花在用Pascal画星形线上，因为那时我刚学三角学。我会说："哇哦，正弦和余弦太有趣了。我又可以一显身手了。"

Seibel：*那你考得怎么样？*

Fitzpatrick：我得了5分[①]。考试内容是写一个大整数类。现在这是我招聘人员的一道面试题："写一个能够进行任意大整数乘法及除法的类。"既然我能在高中的一次AP考试中做出这道题，那么他们在这儿应该也能做得出来。

Seibel：*你大学第一年的夏天在Intel工作，那在高中时期有没有做过程序员的工作呢？*

Fitzpatrick：是的，我在Tektronix[②]工作过一段时间。在正式工作之前，我有几个主机账号。我写了些机器人程序，往聊天室里灌水，把AOL惹翻了，这么做确实让人讨厌。我在另一个Windows程序中编写AOL客户端脚本，还写机器人程序猛提交AOL的线上表单，获赠CD。因为不想让他们发现这是重复表单而只寄一张CD，我用了自己名字的各种变体。这些账户有100个免费小时，或者是5000个免费小时。在这几千次表单提交后，整整一个星期，邮递员会天天带着很多CD过来。

我妈妈说："见鬼，Brad，你会有麻烦的。"我回答："呃，这是他们的错，对吗？"后来有一天，有一个找我的电话，我接了（通常我不去接电话），电话那头是一个AOL的人，他对我大叫："别再向我们提交表单了！"我平时的反应并不敏捷，但这次我立刻回敬了他："为什么你们会寄给我这些废物？每天都有邮递员过来，扔下这些CD！"他说："对不起，先生。不会再发生这样的事情了。"然后我把这些CD都用来布置大学宿舍了，它们现在还放在

---

① AP考试最高5分。

② 测试、测量与监测领域的知名公司。

我家车库的一个盒子里呢。我记得它们曾是很好的装饰品，所以没有扔掉。

恶搞完AOL后，我得到了一个当地ISP的shell账号。大概上我就是在那时学的Unix。虽然不能运行CGI脚本，但我可以用FTP上传东西，所以我就在家里的台式机上运行Perl程序来生成整个Web站点，接着再上传到服务器上。后来我在Tektronix得到了一份工作，类似暑期实习。当时我已经很懂Perl和Web了，但还没有做过动态Web。在1994年、1995年时，Web还是个很新的东西。

我去Tektronix上班的第一天，他们给我介绍我的办公用品："这是你的电脑。"那是一台大的SPARC工作站，也可能是别的什么运行了X和Motif的机器。"这是你的浏览器。"可能是Netscape 2，我记不清了。"如果你有CGI的东西，放到这个目录里。"我记得那天晚上我写了个最基本的Hello World的CGI程序，大约就三行吧，我感叹道："天哪，这太有意思了。"第二天早上六点我就来工作了，继续疯狂地写CGI。

后来我开始自己做动态Web编程。当时我找了台支持CGI的Windows Web服务器。我最终说服了我的ISP（也许是和他们交情不错，或者之前给了他们不少帮助让他们信任我）："OK，我们会运行你的CGI，但开始前会对它们做审核。"他们仔细查看了代码，然后把那些代码扔到他们的目录里。那是一个投票站的脚本，你可以用它来创建投票程序，例如："你最喜欢的电影是哪部？"添加完选项后就能开始投票了。后来的几年里它变得越来越流行了。

Seibel：是FreeVote吗？

Fitzpatrick：是的，在弄爆我的主机后它变成了FreeVote。那时Banner广告真的很流行，也许就是那段时间开始流行起来的，我通过它赚到了越来越多的钱，得到越来越好的合同，每次点击的收益也越来越高。最高时每点击一次广告我就能赚到27美分，就算是按今天的标准，我觉得也是相当夸张的。有了它，我每个月在Banner广告点击方面能有2.5万～2.7万美元的收入。

这些都是在高中时做的，我整个高中时期都在私下做这事。我还在Intel做了两个夏天，随后在进入大学前的最后一个夏天，我办起了LiveJournal。为此，在进入大学的第一年，我卖了FreeVote，基本上算是送给一个朋友的，大概只要了1.1万美元，因为我想摆脱它，还有相应的法律责任。

Seibel：你有了ISP并开始使用Unix，这有没有对你编程带来什么变化？

Fitzpatrick：Unix并没让我抓狂。我没办法理解Windows上都发生了什么，

也许你看过Windows API——每个函数都有差不多20个参数，它们都是标志位，而且一半都赋了0值。完全不知道发生了什么。当有东西无法正常工作时，你根本没办法知道究竟是怎么回事。

Seibel：你早期的编程方法或编程风格和你现在所想的有什么明显的不同之处吗？

Fitzpatrick：我用过很多种编程风格，面向对象的、函数式的，现在用的这种有点怪异，混合了面向对象和函数式编程。这是我热爱Perl的原因，虽然语法很丑陋，有很多历史包袱和瑕疵，但它从不限制我写代码的风格。用你喜欢的风格去写就是了。你能让代码优雅一致，却没有和特定语言相关的风格。直到我进入Google，我写的Perl代码才慢慢少了下来。

运营LiveJournal后，我也做了很多测试工作。尤其是当我开始和他人共事时，我意识到永远也甩不掉自己写的代码，要一辈子维护它们时，我开始写测试了。在十年前的博客文章上，我收到了这样的评论："嘿，我看到这段代码，发现了一个问题。"然后我立刻着手维护代码。

我现在维护着很多代码，还有其他很多人在和它们打交道，如果有什么地方不清楚，我会假设有人理解不了我写的某些不变式。因此，当我要搞些小聪明时，通常我都会保证在代码出现问题时有测试及时跳出来。我也强迫其他人写测试，他们基本上都为我工作。我会为自己的代码写测试，当别人写代码时我会告诉他们："你确定这段代码能工作吗？写个测试证明给我看。"有时，他们会意识到"天哪，这么做太有用了"，尤其是在后期维护的时候。

Seibel：你是从什么时候开始和他人一同共事的？

Fitzpatrick：差不多是在大学结束的时候，我开始雇用其他人，尤其是毕业搬回波特兰后。

早期的雇员是客户支持，所以他们不用写任何代码。慢慢地，我开始雇用程序员。我雇佣的第一个人是我的一个网友，他的名字是Brad Whitaker，我们都有名为BradleyLand或BradleyWorld的网站，因此我们找到了对方的网站。我比他早几年开始Web编程，也可能是早一年，他问我："嘿，你的网站是怎么做的？"就是说它到底是HTML、Frame、CGI还是Perl的。后来我开始接很多合同项目，我就把一些我不做的项目给他。有一次我们有个大项目，谁都没有办法独自完成，我找到他说："这个项目需要两个人来做。"他让我飞去宾夕法尼亚，也许是匹兹堡？我对东海岸完全没概念，我是个生活

在西海岸的人。也许是费城？有牛肉奶酪三明治[1]的地方。

Seibel：是费城。

Fitzpatrick：是的，我们第一次见面是在一家便宜的旅馆里，我感觉好像早就认识他一样。他和我打招呼："嘿，最近怎么样？"他走了进来，在我旅馆的卫生间里上了个厕所，当时我就站在那儿，可他连门都不关。我回答道："还不错。你还真惬意。"虽然我们从未谋面，可就像认识了四五年一样。然后，我们就开始干活了。

他住到我空余的一间卧室里，我们差不多把厨房给搬空了，架起几张桌子，放上电脑就开工了。我们通常10点或11点起床，干到中午，看会儿电视（穿着短裤坐着看电视），然后不间断地工作到早上三四点钟。后来，我的另一个朋友从华盛顿大学过来度暑假。他是我大学一年级后认识的，我们三个在一起忙碌着。这个朋友住在市区，早上坐轻轨过来，然后滑滑板到我家。他就坐在外面用Wi-Fi上网、写程序，直到我们醒过来去给他开门，让他进来。

一起有了三个人，房子就有点挤了，我就说："哦，好吧，我们搞间办公室吧。"于是我们弄了间办公室，"我们既然有了这些空间，不如再雇点人吧！"在随后的两年里，我们慢慢扩大到了12个人，而LiveJournal也逐渐流行了起来，当然压力也更大了，因为我还得处理人事问题。

后来，我妈妈来处理人事，随后我们的关系就有点紧张了，因为她为我工作。我给她定了几条规矩："如果你给我打电话，要分清楚是私事还是公事。要么只谈私事，要么只谈公事。你不能在工作与私人问题间换来换去的。"如果她转变话题，我就会挂电话。然后她再打回来，我会说"你搞混了"。这真的搞得很紧张，当我把公司卖掉时她真的很高兴，她不用再为我工作了，我们也不用争吵了。

Seibel：那时你的公司还接合同工项目吗，还是说这就是整个LiveJournal了？

Fitzpatrick：差不多这就是整个LiveJournal了。我们还打算开个照片托管服务，这方面Flickr做的比我们好，我们的那个有些过度设计了：漂亮的抽象，能结合到任何东西中。为LiveJournal做的每个新的基础设施，我们都会问自己："它怎么和FotoBilder结合到一起？"为此，我们把每样东西都抽象

---

① cheesesteak，费城牛肉奶酪三明治，这是费城及其附近地区的食品。用一种细长的意大利面包，里面有切得很薄的肋眼牛肉片加炒过的洋葱及蘑菇，然后淋上融化的奶酪酱，也可加上红椒或者是很辣的小尖椒。（摘自维基百科。）

出来。Memcached是抽象的，因为没有必要把它和LiveJournal绑在一起。随后我们还做了个类似GFS的文件系统，还有一个任务队列。为了提高可扩展性，我们不停地开发基础设施组件，它们能被应用于我们的各项产品之中，由于没有复杂的依赖关系，它们的维护也更方便了。虽然可能会增加一些工作量，但如果能减少依赖，那还是很值得的，所以我们开发了所有这些通用基础设施。

Seibel：*我对于你扩展LiveJournal的过程有些好奇，你是从什么地方开始的，一路上又是怎么学到你需要的东西的？*

Fitzpatrick：我们和其他客户共享一个Unix主机，差点把它搞挂了。

Seibel：*以CGI的方式运行的吗？*

Fitzpatrick：是的。我想从严格意义上来说那应该是个CGI，派生出所有的东西然后终止掉。ISP分配给我一个家伙。我的服务器当时总是死机，我对他说："我每月为这台服务器付10美元，它为什么不能工作？"他告诉我："哦，你该这么做。"很快我就学会了Unix，知道正在发生的事情。

我从CGI转到FastCGI，调整了Apache，关闭反向DNS查询功能。经过了这些步骤，性能有了好转。最终，我遇到了IO和CPU的瓶颈，我买了自己专用的服务器，但那也只是一台机器，它经常死机，而且我的硬盘快没有空间了。起先这台服务器只对我的朋友开放，我没有取消注册页面。他们邀请了他们的朋友，而这些朋友又邀请了其他朋友，但我并没想过把这个站点变成公开网站。它只是正好有个开放的注册页面。于是，我在LiveJournal的新闻页面里写了点东西："我们需要大家的帮助，我们需要购买服务器。"

我记得那次大概募集了六七千美金，我买了两台大的Dell服务器，托管在西雅图市区的Speakeasy[①]里。有人推荐了一些Dell服务器，这些6U的大家伙差不多有90磅[②]一台。数据库服务器和Web服务器从逻辑上是独立的。因为我运行了一个MySQL进程和一个Apache进程，所以我只知道这样去分离。

就这样过了一阵子。Web服务器直接面向大众，服务器上有两块网卡，通过交叉线缆（crossover cable）连接数据库服务器。后来Web服务器过载了，但那还比较好对付，那时我有了几台1U的服务器。之后我们有了3台Web服务器和1台数据库服务器。这时我先后用了三四个HTTP负载均衡器——

① 一家提供宽带接入、服务器托管等服务的公司。

② 1磅约等于0.454千克。

mod_bachhand、mod_proxy和Squid。这些我一个都不喜欢，我就是从这时开始讨厌HTTP负载均衡器的。

接下来过载的就是数据库，那时我抱怨道："哦，见鬼。"Web服务器可以很好地扩展，它们都是无状态的，只要投入更多的机器，分散负载就可以了。那段时间很难熬，"我可以优化查询来应付一下，"但那只能坚持一周，一周后它又会过载。那时我就开始思考一个独立的请求到底需要什么。

那时我认为自己是世界上第一个想到这个方法的人，我们可以切分数据库，将它分区存放。我画了几张图来做设计文档，简单说明我们的代码会是如何工作的。"我们的主数据库只存放全局相关的元数据，这些是低流量的。各个博客和评论相关的东西会分别存放到每个用户的数据库集群里。每个分区都有自己的用户ID。"现在看来，每个人都会这么做。但那时候在不间断服务的情况下迁移代码着实是件很费力的事。

Seibel：*在你迁移时有没有暂停服务？*

Fitzpatrick：没有。每个用户都一个标志位，标明他在哪个集群上。如果标志位是0，说明他在主数据库上；如果非零，则说明他已经被分区了。有个版本号是"你的账号正被锁定"。这时该账号处于锁定状态，尝试迁移数据，如果在这时你做了什么变更则需要重试。基本上在我们完成迁移前你在主库上不能进行写操作，迁移完成后声明："OK，现在你的账号可以使用了。"

迁移程序在后台运行了两个月。我们计算过，如果只是把数据导出来，写点程序拆分SQL文件随后重新加载，可能要花一周左右。摆在我们面前有两个选择——停机一周或者缓慢迁移两个月。我们先迁10%的用户，这时对其他用户而言，整站的可用性提高了，随后我们再慢慢提升迁移到集群的用户比例。

Seibel：*这就是memcached和Perlbal的前身了吧。*

Fitzpatrick：是的，那确实是Perlbal的前身。memcached是在那之后的事情了。在大学结束离开学校前，我都没想过要做memcached。站点越来越慢，一天我在浴室里洗澡的时候灵光乍现，突然意识到我们有这么多空闲的内存可以利用。那晚我写了个原型，服务端和客户端都是用Perl写的，服务端很快就崩溃了，因为对于一个Perl服务器而言CPU的使用率太高了。于是我们着手用C来重写它。

Seibel：*这样就省去了购买更多数据库服务器的开销了。*

Fitzpatrick：是的，数据库服务器又贵，迁移又慢。Web服务器非常便宜，把它们加上马上就能见效。如果你买一台新的数据库服务器，差不多要花一周来进行配置和验证：测试磁盘、配置和调优。

Seibel：*这么说来，你所开发的所有这些基础设施，比如memcached和Perlbal，都是为了响应LiveJournal的实际扩展需要？*

Fitzpatrick：是的。我们开发这些东西都是因为LiveJournal承受不了负载，我们挑灯夜战开发新的基础设施。我们甚至还买过NetApp[①]。我们问："这要多少钱？"他们回答："说说你们的商业模型。""我们有付费账户。""你们有多少客户？怎么收费？"你就能看到他们在那里做乘法。"价格是你们在不破产的情况下的所有可支配收入。"我们心说："去你的。"但我们确实需要它，所以还是买了一台。我们对它的I/O性能并不满意，它不仅很贵，还会形成单点故障。他们试图卖给我们一套高可用配置，我们想："去你的！我们不会再买这种东西了。"

后来我们开始写自己的文件系统。我不确定当时GFS的论文是不是发表了，我觉得我应该从谁那里听到过。那时我的内存很分散，取个散列的键，从分区里取值。为什么不能把这招也用在文件上呢？文件是永久性的。因为增加存储节点时配置会发生变化，所以应该记录下文件的实际位置。那用不了多少I/O，只需跟踪文件的位置就可以了，但如何保证高可用性呢？我们想出了一个解决方案，我提出一个计划："这是我们确定文件位置所需的全部的读和写。"我先写了主控的MySQL Schema和文件位置追踪器。随后，我发现："上帝啊！这部分用HTTP就能搞定了。它根本不难！"

我记得在想了整夜之后，我们就投入工作了。我们在公用办公楼下面有个会议室——一间昏暗的大会议室。"好了，各位，停下你们手头的事情。我们下楼去画图。"每次我有设计思路时差不多都会这么说，我们应该找块白板把它画出来。

我解释了整个设计，谁和谁交互，谁处理请求。然后大家就上楼了，我先预定了所有的硬件，因为差不多要两周左右它们才能到货。然后开始编码，我们希望能在机器到货前完成编码工作。一切总能挑出毛病，总有东西出问题，因此我们总在写新的基础设施组件。

Seibel：*有没有这样的情况，某人在开始时让你坐下，告诉你"你需要知道X、Y和Z"，你的生活会不会更容易些？*

---

① 知名高端存储和数据管理品牌。

Fitzpatrick：从一开始就把事情做好总是比迁移一个线上服务要容易很多。这一直都是最让人头痛的。我所说的每件事，你都能在一台机器上做到。开始时就这么设计，你就不需要假定能把这两个用户数据关联起来或是别的什么了。假设你想加载这20条数据——你的实现可以把它们从同一张表里加载上来，但在更高层次的代码里只需要说一句“我想要这20个对象”，就能实现从一组机器中收集数据的工作了。如果从一开始我就这么做，可以免去不少迁移的痛苦。

Seibel：这么说，基本上你学到的就是“要为数据无法容纳进一个数据库的那天做准备”。

Fitzpatrick：我认为这已是Web社区中的常识。人们总是假定他们的站点会变得很大，有些想过头了。但在那时，常识却是，Apache和MySQL就是你需要的一切。

Seibel：看起来你写这些组件是因为你需要它们，同时你也乐在其中。

Fitzpatrick：哦，当然！我确实是在找理由去使用各种东西，去学习它们。因为如果不实际用它写点什么，不和它生活在一起，你永远学不到东西。出于兴趣去学一门语言和学会它是两回事，如果不用它写些大的、复杂的系统，那你不能算是真的学会了。

Seibel：那你觉得对你而言，你和哪些语言生活在一起呢？

Fitzpatrick：Perl、C。以前的话，还有BASIC，不过我不确定BASIC算不算。我还写过很多Logo。在小学的Logo课上，大家都在提笔、落笔，而我不在图形模式里——有些键能跳出图形模式，我在写函数。老师会走过来说：“你在干什么？你做的不对，你应该在画房子。”“不，我在写Logo。你看。”“不，你弄错了。”在课程结束的时候，我做出了些东西，我写了个类，能以任意尺度和任意方向画出所有字母。有了它，我能在波状横幅上打印整条消息，并加入距离和填充。每个人都很吃惊：“这是什么？”我也不知道这个算不算。

现在在用很多的Perl和C，在大学里还为工作和Windows程序写了很多C++。后来我的C++忘得差不多了，或者说是退化了。现在我在Google，去年我写了很多C++、Python和Java。在Java刚出来的时候我也写过不少，但后来厌倦了。现在我又开始写Java，又开始有些不爽了。

Seibel：使用何种语言很重要吗？

Fitzpatrick：目前还没找到能让我满意的语言。我还不太清楚究竟什么才能让我完全满意。我讨厌在一个项目里一直切换语言。我想要的语言要在我需要时拥有静态类型，在编译时检查所有内容。Perl比较接近这个要求，我能用各种喜欢的方式来写代码。它在编译时没有足够的静态检查，但我可以在运行时对其进行补充。不过Perl还是不够好。

我想要可选的静态类型。在Perlbal里，除了核心（要复制字节），没理由让半数的东西都有这么高的性能。我希望在特定的部分代码和类型声明时能给运行时一些提示。但如果想使用惰性求值或模拟一些东西时，我可以采用另一种方式。

Seibel：*这么说来，你主要就是希望语言能有静态类型，这样编译器能更好地进行优化？*

Fitzpatrick：不是这样的。我还希望在编译时它能告诉我"你正在做傻事。"有时我并不关心编译时的情况，我希望它能在运行时有些强制措施，能做任何事。我并不想对Perl 6过于乐观，他们的确在讨论很多我希望看到的东西。但我并不认为它们最终能实现出来。

Seibel：*你喜欢C++吗？*

Fitzpatrick：我并不是很喜欢它。C++的语法很糟糕而且并不一致，它的错误消息也很可笑，至少GCC的错误消息是那样的，往往会因为漏了一个分号而显示40页的错误信息。但和别的东西一样，你很快能记住各种模式，之后甚至不用看那些信息就能知道"哦，我大概忘记在头文件里关闭名字空间了。"我认为新的C++规范，尽管加入了很多复杂性，但还是添加了不少能减少打字痛苦（击键次数）的东西。例如，自动变量和for循环，它的风格变得更像Python了。还有lambda表达式，尽管用的是C++，但这足以让我误以为自己在写Python了。

Seibel：*你是出于性能目的使用C++的吗？*

Fitzpatrick：是的，差不多吧。我在Google主要使用C++。各种有性能要求的东西都会用C++来写。我在Google还写了不少Java。

Seibel：*据我所知，Google有以C++为中心的文化，因为那是Google最早使用的语言，而且他们围绕它构建了整套软件基础设施。你不能改变历史，但在Google可能有很多C++代码对提升性能而言并不是必需的。*

Fitzpatrick：原因是这样的，随着时间的推移，Java越来越快了，JVM越来

越聪明了。关于Java，有件事让我很不爽，那就是每个人都很讨厌JNI。有时一个库是用C++写的，Python开发者（不管Google里的还是外面的）并不在乎，他们会说："哦，我们会用SWIG包装它。"他们继续自己的工作，做得很开心。Python能很快从用C++写的东西中获得支持，因为Python开发者并不介意它到底是用什么语言写的。

Java的人却会说："这必须得是纯Java的。我们不用JNI，否则若JVM崩溃了，我们根本不会知道是为什么。"这个问题最后会导致所有的东西都要写两次，一次为C++、Python和所有其他语言，另一次专为Java而写。如果他们能讨论出一个好的嵌入方案或者克服对JNI的恐惧，那我就没什么好说的了。

Seibel：*那你觉得内存管理和垃圾回收相比怎么样？人们还在争论这个问题，你有没有什么强烈的倾向呢？*

Fitzpatrick：没有。看见人们怀着那种强烈的倾向，尤其是通常他们背后并没有什么依据支持时，我还挺开心的。我个人并不觉得管理内存有什么讨厌的，至少在C++里用限定作用域的指针时是这样的。我能用C++连着写上几天代码，期间不用new或delete。看上去这什么都能做。

我在Google时重写了memcached，让它能同Google的基础设施协同工作，并将其加入到App Engine里。这里全部使用C++，因为我需要对内存进行独占式控制，这样可以减少碎片。我很欣慰C++能做到。

Seibel：*memcached最早是用C写的。你把它用C++重写了是因为C++在Google被普遍接受，还是有什么别的优势？*

Fitzpatrick：我开始时用现成的代码做迁移，但发现这样做工作量反而更大。memcached并没有太多代码，所以直接用C++重写能更快些。重写后代码差不多相当于以前的一半。

Seibel：*那你认为这是因为用了C++还是你比当时更聪明了呢？*

Fitzpatrick：有可能是我聪明了。记得我大约在11岁或12岁时，做了次环美旅行，我在一台TI-85计算器上写了个Mastermind游戏[①]。我在这个小屏幕上写了两百行程序来记住我在哪儿。最后我把这鬼东西删了两次，所以我一共写了三遍代码，但最后它变容易了。这是很有道理的，因为第二次做某件事

① 由以色列电讯专家Mordecai Meirowitz发明的经典益智游戏。

时，它就变容易了。

Seibel：你的很多工作都是用Perl完成的，这是一种相对高级的语言。你认为程序员应该了解多少底层的东西？程序员还需要了解汇编以及芯片是如何工作的吗？

Fitzpatrick：我不知道。我遇见过很聪明的人，我觉得他们是好程序员，但他们只懂Java。他们解决问题的思路被局限在他们的知识范围内。他们不会全面地思考问题。虽然你不用操作整个系统，但对其有所了解还是很有必要的。

我在做LiveJournal时，要考虑从JavaScript到如何与内核交互的所有内容。我读了Linux内核中关于epoll的代码，于是想："假如我们持有全部连接到这台负载均衡器上的长TCP连接，JavaScript在轮询它们时又会怎么样呢？"我试着计算每个结构体要占用多少内存。这还是比较高级的，我们还会思考些这样的问题：我们的以太网卡有太多的中断，每当收到数据包时网卡都会发送一个中断，它们加到一起就达到网卡的极限了，就算是一块千兆网卡也只能达到百兆的速度，要是我们在内核里切换到NAPI技术会怎么样呢？我们还收集数据，看看什么地方做切换比较好，能释放处理器资源。

我们从这些真正低级的东西上受益颇多。最近有人对我说："Java会考虑这些问题的，我们不需要处理它们。"我说："不，Java不会处理这些，因为我知道你正在使用的内核版本，这个版本不支持这个特性。你的虚拟机把这些都隐藏了，它给你一个抽象，让你觉得它很有效，但实际上只有当你使用特定版本的内核时它才会生效。"如果人们连整个系统的最表层都不明白，那我会感觉很失败的。

从实践角度来看，这些东西没一样是行之有效的。所有这些漂亮的抽象背后都一塌糊涂。那些看起来很美的库实现也很糟糕。所以说，如果你是那个对购买服务器的经费负责的人，或者对可用性负责的人（如果你在发生重要事件时要求随叫随到），那么去了解底层正在发生的事情，不轻信别人的库、代码和接口是非常有帮助的。

如果我是现在才开始编程的，我几乎不认为自己今天会变成一个程序员。这东西太丑陋了。这就是我对App Engine这类东西感到兴奋的原因。有人把Google App Engine说成是当代的BASIC，因为这个时代里一切都是联网的。在我写程序时，就只有一种语言，运行在我自己的机器上，部署是up回车或者RUN回车。现在的孩子不想在自己的机器上写些傻乎乎的东西，

比如益智类游戏的“bounce a ball”应用。他们想要一个可以交互的Web站点。

我还是会收到一些人的邮件：“嘿，我有个主意，我想整合Wikipedia和YouTube，整合……”每个人都想做一个Web站点，他们最喜欢的4个Web站点都不太顺眼，他们想做一个符合他们心意的。

事实上，App Engine给了你一个按钮，“把这个放到Web上”，你只需用一种语言来写程序，Python就很完美（相对比较易学，关于这点还有些争议）。这对编程而言是个很好的入门，我们有太多的分层，它摒弃了一些无用的层次。

Seibel：*前面那个人告诉你“Java会替你处理它”，你感到不爽。那这不一样吗？“App Engine会替你处理它的”。*

Fitzpatrick：我不清楚。也许因为这次我知道底层在发生的事情。实际上JVM也不是那么糟糕。我猜人们只是盲目相信他们的抽象，不清楚底层的东西。

Seibel：*你在进入大学学习计算机科学前就有很多编程经验。你觉得大学的课程怎么样？*

Fitzpatrick：我跳过了很多早期的C.S.课，因为它们真的很无聊。但我会去参加考试。后来开始上300-Level和400-Level[①]的课时就比较有意思了，可正当我有兴趣去学时，就毕业了。他们也不让我上研究生级别的课程，因为我不是研究生。

我记得在讲编译器的课上，最终的项目是给我们正在使用的语言添加一组特性，还包含一个自选特性作为加分部分。我选择了实现运行时数组边界校验。教授会拿他的测试集去运行我们编译出的二进制文件，有些测试失败了。他说：“对不起，你的分数是C，因为你的单元测试失败了。”我看了下他的测试，说：“你的测试里有个off-by-one错误[②]。”于是他重新给我打分，我得了个A，但我没有得到为语言添加新特性的附加分。在学校那会儿我对此很生气。

我还记得教我们数据库的老师是一个看起来完全没有实际经验的人。那时我用过Oracle、Microsoft Server和MySQL。我会问一些我很希望得到答案的、我们网站遇到的现实问题，但他只是给我一些书本中的答案。我只能说：

---

① 国外的课程在学科后跟上3位数字，100、200、300、400，数字越大表示难度越大。

② off-by-one错误，又称“差一错误”或“栏杆错误”，常见的关于边界条件的错误。详见http://en.wikipedia.org/wiki/Off-by-one_error。

“不，不。那样没用。”

Seibel：*你毕业于2002年。现在是否会对当时学校所教你的东西心存感激？*

Fitzpatrick：半数的课我都很喜欢，无论是学习那些当时还不知道的新东西，还是学习背景知识或一些术语，都是如此。在此之前，我虽然很懂编程，但却无法用合适的词汇去描述我正在做的事情。如果我用自己的语言来进行描述，其他人会觉得我根本不懂我自己在说什么。早期的C.S.教育让我能更好地与人进行交流。

Seibel：*关于在校期间经营自己的事业，你有没有感到后悔过？是不是应该从中选择一样来做？*

Fitzpatrick：不，我觉得这是最好的方式。我别的朋友在大学里只是完成课程，而我对这些已经很了解了，只读书的话会很无聊。有个朋友，同样很懂这些课程内容，但他的想法是在学校就该学习，而不是为了考高分，所以他在业余时间学习阿拉伯语、汉语和日语，还有很多疯狂的编程语言。每周他都会说：“我喜欢上了一门新语言，这周我只用OCaml来写东西。”他以这种方式让自己保持忙碌的状态。而我则选择了一种繁忙而又不怎么无趣的方式。

我还有些朋友在大学第一年就退学从事Web相关的工作。有对夫妻做着成人网站之类的东西，他们觉得“我们正在赚钱呢”。但他们只是在工作，在自己的地下室里忙碌着。大学校园里最棒的是认识人和参加聚会。如果我只是在做LiveJournal，我会把自己折腾死的。

Seibel：*你喜欢学习计算机科学专业吗？*

Fitzpatrick：不读这个专业我可能也干得挺好。我做了很多一般情况下不会去做的事，因此我想这个专业还不错。也许我还应该再学点别的，在学校多待一年，修个完全不相关的第二专业，多学点语言学。离开校园后我有些伤感，觉得自己才完成了一半的学业，因为有这么多东西都是之前就知道的。早期的C.S.课我基本都没去，等到后来课程变得有意思起来，他们却说：“OK，你毕业了。”

Seibel：*你有没有想过要读研？*

Fitzpatrick：有，读研可能很有意思，不过我太忙了。

Seibel：*你现在还会关注C.S.相关的文献吗？*

Fitzpatrick：我和我的朋友们还是会互相转发好的论文。不久前，我刚读了一篇关于运行时调整布隆过滤器（Bloom Filter）大小的新技术的文章，这看起来太棒了。我会去读那些来自存储会议、业界和学院的论文，那些关于不同系统的论文。Google有很多阅读小组——系统阅读小组或者存储阅读小组。我会看Reddit上的文章、朋友转发的论文，或者博客上的链接。

Seibel：你刚提到来自学院和业界的论文。你觉得如今这两者是否有交叉的地方？

Fitzpatrick：它们对我来说都差不多。很多时候读业界的论文会更有意思些，因为你知道他们用它来解决问题，他们的解决方案绝对不是“如果……，我们认为会很酷”。学院派里也会出些疯狂的东西，根本不能用，只是些疯狂的想法。也许今后能把它变成商业化的东西。

Seibel：你是如何设计软件的？

Fitzpatrick：我会从接口开始，比如常用方法、常用RPC或者常用查询。以存储为例，我会试着去思考，常用的查询是什么？我们需要什么索引？数据是如何存储在磁盘上的？然后给不同的部分写点试验模型，再慢慢充实它们。

Seibel：你在测试中是凭直觉写吗？这样可以随心所欲地进行测试。

Fitzpatrick：越来越常这么做了，我总是这样设计软件的，甚至是在测试之前。开始时，我只设计接口和存储，随后再开始真正实现。

Seibel：你的设计是什么形式的？伪代码？实际代码？还是白板草图？

Fitzpatrick：通常我会打开编辑器，写下关于Schema的伪代码。等它差不多了，就写真实的Schema，再复制粘贴一下，看看“create table”命令是不是好用。等一切都准备就绪，就开始动工实现它，我总是从spec.txt开始。

Seibel：在写了一大堆代码之后，有没有碰到过要重新考虑原始计划的情况？

Fitzpatrick：有时会这样。但我总是从最难或者最不确定的部分开始实现。我不会把任何难的或者让人吃惊的东西留到最后，我喜欢开始就做最难的部分。至于那些我没完成的项目，我的朋友会取笑我，不过我已经把难的部分做完了，我已经学到了想学的东西，那些无聊的工作就让它去吧。

Seibel：对于那些自学的程序员，你有什么建议吗？

Fitzpatrick：要试着做点更难的东西，超出能力范围的东西。要多读代码，我以前常听人这么说，不过那时一直没有静下心来去做。那么多年里我写了很多代码，可从来没读过别人的。后来我上了因特网，有那么多我可以贡献力量的开源代码，但我却被吓倒了，如果不是我自己的代码，脑子里没有对全局的认识，我根本无法深入理解它。

后来我开始给Gaim发补丁，这是GTK下的即时聊天工具，我通过阅读代码来了解整个设计。看了一部分之后，我明白了。我意识到看完别人的代码后，并不需要记住它们，我开始去了解模式。看着他们的代码，我会说："哦，OK。我明白代码的结构。"

慢慢地，我变得很喜欢读代码，因为当我遇到不理解的模式时，我会问："等一下，他们为什么要这么做？"随后再多看看就明白过来了："哇，这真是个好办法，很有效。"我应该早些开始，但我害怕，因为我觉得如果不是自己的代码，我就没办法理解它。

Seibel：*你是如何阅读他人的代码的？比如先说说，阅读代码是为了全面了解它是如何工作的，还是自己想要对代码做些修改？*

Fitzpatrick：通常我是想要修改一些东西。如果你崇拜某个程序员，也可以去读读他的代码。也许你会意识到他们也是凡人，不该成为你崇拜的对象。你也可能会从他们的代码中学到一些东西。

Seibel：*你刚说到你想做些修改，一般你是怎么做的？*

Fitzpatrick：第一步，找个原始的tar文件或者从SVN检出代码，试着把那该死的东西构建起来。你一定要跨过那道坎，那对多数人来说是最大的障碍——构建系统时的依赖或者开发者假定你已经安装了这个库。我希望这些大项目都能自带一个虚拟机，也就是它的构建环境。

Seibel：*你是说像VMware那样的虚拟机？*

Fitzpatrick：是的，如果你想立刻动手修改，这里有全部的依赖。人们的连接速度够快了，这是完全可以实现的。

不管怎么样，一旦你有了一个干净的、可工作的构建版本，干掉它，做一个修改。把标题栏改成"Brad says, 'Hello world.'"，改变一些东西，即使一切都很丑，只要动手开始去改变就好。

然后，把你的补丁发出去。我发现这是最好的开始对话的方法。如果你在邮件列表里说"嘿，我想添加特性*X*"，维护者可能会回答："哦，见鬼，

我很忙，一边呆着去，我讨厌特性X。”但如果你找到他们后这样说：“我想添加特性X。我正在考虑与附件里的补丁类似的东西。”——当然那是错的，你说：“但我认为这完全是错的，我想正确的方式也许是做X。”然后给出个更复杂的方法，通常他们会回答：“上帝啊，他们试过了，看看，完全做错了。”

这可能会让维护者很苦恼，他们会说：“哦，我不敢相信他们花了这么多精力来做这个。如果方法正确，做这个很简单。”或者是：“哦，哇，他们完全做错方向了。我希望他们不要再继续了。”随后他们就会回复了。

这是开始对话的最佳途径。甚至在Google，这也是我与很多不认识的团队开始对话的方法。当我修复了他们产品的一个bug后，第一件事就是用邮件寄给他们一个补丁，告诉他们：“你们怎么看这个？”或者在内部代码审查工具里写道：“这是审查结果。你怎么认为？”他们可能会说：“见鬼，这段修正的代码完全搞错了。”

Seibel：*你现在还会出于兴趣去读代码，而不是因为要用它才去读它？*

Fitzpatrick：有时会。我会毫无目的地签出Android和Chrome的源码。当它开源后，我会镜像一个它们的代码库，然后随便看看。我对Firefox和Open Office也做过同样的事。一些你一直在用的程序，突然有一天你能访问到它的代码了，你也会去看一看的。

Seibel：*那种程序的代码库非常大，当你出于兴趣去看东西的时候，会有多深入呢？*

Fitzpatrick：一般来说，我只是大致地看一下，试着去了解目录结构。然后如果有东西吸引我的注意，或者我对什么东西不太理解，我就随便选个文件，感受一下。随后漫无目的地看看周围的文件，直到我厌倦了，再挑个新文件去看。

很多时候，我会边构建边读代码，因为通常这是可以并行的任务，尤其是当构建很困难时。完成构建后，如果我愿意，就可以开始调整它了。

Seibel：*你读过好代码，它可能符合你已知的模式，或者你会发现新模式。但并非所有代码都那么好，坏代码最初的征兆是什么？*

Fitzpatrick：嗯，我现在特别傲慢，在Google工作，对所有语言都有非常严格的风格指导方针。针对使用最多的六七种语言，方针里写道：“我们是这样安排代码布局的。要这样来命名变量。要这样来使用空格和缩进，这是你

要使用的模式和惯例，你要这样声明静态域。”

我们也会把这些指导方针放到网上，以供我们项目的外部贡献者作参考。我们想要有一份成文的指南，这样就不用再说“我们不喜欢你的风格”了。

现在，当我用C语言做项目时，第一件事就是添加一份风格指南。当项目成熟后，有很多人会来修改它，他们会有一份风格指南。也不是总能有这么一份东西，但程序员应该尊重已有的代码风格。也许他们不喜欢大括号风格，但要知道，在一个文件、一个项目中保持一致的风格要比按自己喜欢的方式来做更重要。

Seibel：*你有没有做过结对编程？*

Fitzpatrick：我觉得那很有趣。结对编程对很多事都很有益处。有时你只是需要思考，想要一个人独处。我并不总是采用结对编程，但它的确很有趣。

我发起过很多项目。如果不完成它们会让我有负罪感，但我确实切换得过于频繁，精力过于分散了。这是我需要结对编程的原因——它强迫我坐定三小时，甚至是两小时或一个小时和别人一起做一件事，他们也会让我不那么无聊。如果我碰巧遇到一个无聊的补丁，他们会说“来吧，我们要搞定它”，然后我们就做成了。

我喜欢一个人工作，但在工作时会到处跑。在飞机上我总带块备用笔记本电池，笔记本上有整个开发环境和本地Web服务器，我能在浏览器里进行测试。但我还是会打开一个新标签页，键入reddit或lwn——我常上的网站。自动补全后按下回车，然后出现错误信息。一分钟里能重复好几次。见鬼！我在工作时就这么做吗？我甚至都没意识到自己经常看网页？这太吓人了。我有个朋友，他有些iptable的规则，当在一天里的特定几个小时去访问特定IP时，会重定向到一个页面，上面写着“你应该在工作”。我没有这个东西，但我想也许需要做一个类似的玩意儿。

Seibel：*你是怎么看待代码所有权的？是独立拥有代码重要，还是团队分享代码更好呢？*

Fitzpatrick：我不认为代码应该被谁拥有。我不认为谁会真的有那种想法。Google里的方式是这样的，有一棵巨大的代码树，一个根目录，一个适用于所有代码的统一构建系统。每个人都能动手修改任意的代码。但我们有代码审查，每个目录都有拥有者，考虑到可能有人会退出或休假，至少会有两个拥有者。

要签入你的代码需要符合三个条件：有人给你做代码审查，并予以肯定。

在语言方面得到认可——起码你得证明你知道这门语言的风格，这叫“可读性”。此外，你还需要得到那个目录的拥有者的批准。如果你已经是拥有者了，又能保证语言的可读性，那你只需要找个人说“耶，它看上去很棒。”这是个很好的体制，因为这里最少有两个，最多有二十到三十个拥有者。只要你在一个代码库上做了一段时间，就会有人把你加为拥有者。我认为这是个很好的体制。

Seibel：*让我们再往回倒一点，你是怎么开始LiveJournal的？*

Fitzpatrick：其实就是和朋友们闹着玩，我所要的和我们所想的都很有趣。LiveJournal上的评论就源于一个真实的玩笑。我在上课前登录了一下LiveJournal，当时我们刚加入好友页面，看到我朋友写的东西，真的写得很蠢，我想取笑一下他。“噢，但我不能回复。”然后我去上课了，整堂课都在想：“我该怎么添加回复的系统呢？”我考虑了现有的Schema以及如何呈现的问题。课间有两小时的休息时间，我用那段时间添加了评论系统，回复了些自作聪明又有点挖苦他的话，然后去上其他课了。第二节课回来后，他对我说：“我们现在居然可以发表评论了？”

LiveJournal上的一切都很搞笑。整个安全体系，例如好友帖和私人帖，都是因为一个朋友写了他去出席一个派对，第二天酒醒了发现自己掉在一条沟里。他父母读到了，说：“什么？你居然喝酒？”他对我说：“Brad，我们需要找个办法把这些内容锁起来。”我告诉他：“我这就去做！”我们已经有好友了，所以只要让一些帖子只能被好友看到，别和父母成为好友就行了。

Seibel：*LiveJournal的早期，你的生活看起来就是无休止的熬夜，睡得很晚，长时间工作。编程真的必须这么干吗？*

Fitzpatrick：这对我来说是压力最小的时候。白天总有各种事情会找上门来，比如要吃饭，要上课，或者要接电话，总会被打断。我没办法放松下来。如果在会议前工作两小时，那这两小时比起没那会议或者一早开会要低产得多。知道没事会来烦我，我会放松得多。

我会觉得晚上是我的时间，我正在偷时间，因为其他人都睡觉去了。没有噪音，没有干扰，我能做各种事情。我有时也会熬到很晚，忙着不同的事情，通常周末会这样。只是那样做会让我花上几天时间来调整睡眠。这么做主要是因为我还在大学里，有些项目要做，又要做LiveJournal。唯一可以用的时间就只有晚上了，而且服务器的维护也只能放在晚上。到了夏天，还有什么原因不这么做呢？早上不用起很早，不用去上课或做别的事情，所以晚

上可以做得晚一些。

Seibel：*那工作的长度和强度如何？你肯定一周工作80小时、100小时甚至120小时。这是必须的吗？在什么环境下这才是必不可少的，在什么时候这只是我们体现的一种男子气概？*

Fitzpatrick：在我看来，我不确定这是必需的还是男子气概。我就是觉得很有趣，这是我想要做的。有时会遇到麻烦，但就算没问题，我也会这么做，因为我正在做那些我希望实现的新特性。

Seibel：*你有没有遇到过不得不估算事情要花多少时间的情况呢？*

Fitzpatrick：刚迁移到Six Apart的时候遇到过一次。那是三年半以前，我想那是我第一次有这样的经历。我们已经开始做迁移了，一位客户问："你们能移动这个数据吗？"这么做要为代码添加支持，做测试，还要发布。这让我很惶恐。也许现在我还会为此类问题感到惶恐，因为我总是忘记某些因素，比如工作时常被打断，还有我从来没摆脱要维护一堆项目的局面。

我想现在我越做越好了，也幸好他们不再经常提这种要求了。现在当我真的遇到某件事有最后期限时，我会说"耶！有个最后期限！"我变得很兴奋，肾上腺素激增，开始干活，把那件事做完。在Google没有什么真的最后期限，我们总是说："你觉得这个东西什么时候可以发布？它看上去怎么样？"很少有真的最后期限。大多数时候，我们都认为它最好在某个时候发布，然后大家都很努力。但如果真的没有完成，你只是会让那些希望看到它准时上线的人感到失望，仅此而已。我所做的多数事情都很顺利，我总是说"船到桥头自然直"。

Seibel：*你以前为LiveJournal招聘程序员时，你还要管理他们吗？*

Fitzpatrick：我更喜欢假设他们都不需要管理，能像我一样自我驱动。HR的经验中有这么一条——一些人只会做让他们做的事，没有那种追求极致的热情。他们会说"做好了，接下来干什么？"他们甚至不告诉你，自己就上网玩去了。我有一些痛苦的经历，但一两年后，我发现原来人与人是不同的。

还有些人是纯粹主义者，他们总是抽象、抽象、再抽象，动作很慢，而且十分崇尚这种风格。他们会说"程序设计是门艺术"。而我则回答："你的代码跑不起来，效率太低，而且它和那些与之交互的代码格格不入。"

Seibel：*你找到让这种人充分发挥的方法了吗？*

Fitzpatrick：有一个人，我试过各种方法。我想他大概比我大10岁。我不知

道究竟大几岁，因为我从没问过，我害怕问那些法定的雇佣问题。但我有种感觉，他不想为年轻人工作，我当时22岁，他始终没想通。这是唯一一个我让他走人的人。

其他人，我最终都找到了激励他们的方法。有个家伙很擅长做原型。他写了sysadmin的Perl脚本。他把所有的东西整到一起，写shell脚本，虽然他的Perl和C写得真的很烂，不过他能让东西跑起来。随后我们惊奇万分："天哪，这些你都研究过了啊，是你让它们互相协作的？"

我们当时正在搭建LiveJournal上的语音系统，有了它你就能录音并发送给LiveJournal，其中涉及了很多要迁移的部分。我觉得那就和地狱一样，而他却很喜欢。他找出了所有的部分，让它们都运行了起来。然后我们重写了这些部分，我发现这就是和他工作的方式。他找出所有的接口，而我们修正它们。一旦我找准了他的定位，我们就能和睦相处了。

Seibel：*你以前为自己的公司招聘，现在我想你也参与Google的招聘工作，你怎么识别好的程序员呢？*

Fitzpatrick：我总是找这种类型的人，他们会做很多别人没要求他做的事，不仅是学校的项目或者前雇主要求他做的。他们对某些事情充满激情，有额外的项目。我问清楚他们如何维护它，对它有多上心，还是就草草调整随后放弃它们。

Seibel：*你有什么特别喜欢问的面试问题吗？*

Fitzpatrick：有一个问题我问了很多次，因为那是我在AP程序设计考试上做过的，题目是给定两个任意长度的十进制数字符串，将它们相乘。这个问题有很多种不同的答案。如果应聘者数学很好（不像我），他们能找出很多种聪明高效的方法。最坏的情况，搞一个类，不停地做加法。

我一开始就告诉他们："不用紧张，你不用给出很高效的解答，能做出来就可以了。"有些人会感到紧张，不知道从何下手。那可是个不好的信号。最坏的情况下，实现一个小学里才做的算法。

我真的在小学里写过程序做长除法和乘法，还展示整个结果，包括所有的步骤，要删掉的地方。以后我们再拿到这种题目时，例如一页10道题之类的，我就会把它输进电脑，再把结果草草地写下来。化学里找电子轨道时我也做过同样的事。我发现通过写程序去作弊也能得到学习，因为你需要深入学习之后才能写出那个程序。

Seibel：你认为这对别人有帮助吗？除了教孩子长除法，我们是不是应该再教他们如何编程，告诉他们“OK，现在你们的任务是写个乘法实现长除法过程”？在他们动手写程序前，他们会明白除法是怎么回事，还是说这种方法只对那些天生对此有偏好的才管用？

Fitzpatrick：这对我很管用。很多时候，别人教你东西时，你会说：“嗯，当然，我明白了。”你其实是在欺骗自己，可是一旦真的要动手去做，就需要了解所有的东西，这迫使你去学习。但我不清楚这是否对所有人都有用。

Seibel：Google和微软都以面试出难题而闻名。

Fitzpatrick：我觉得这些都是被禁止的，或者不被推荐的。可能有些人还是在用，但总的来说，我们不鼓励这样做。

Seibel：他们在面试你时问了什么？

Fitzpatrick：有一个问题是假设你有一组电脑，连在一台交换机上，开启整个机架，请给出一个算法，让机架上的每台机器都能知道其他机器的情况，是开着的还是关着的。基本上就是个存在性问题，条件就这些了。他们简单描述了一下Ethernet：你可以给所有人发广播，或者发给特定的MAC地址。我只是遍历了各种不同策略，降低带宽，减少发现某台机器宕机的延时。这道题目比较有趣。

Seibel：你遇到的最大的bug是什么？

Fitzpatrick：我尽力不去回想它们。我讨厌那些离自己的假设差距很大的东西。记得有一天（显然这不是最糟糕的情况），我花了90分钟调试一个问题，因为我把输出写到了一个文件，然后从一个同名文件中去读，但路径里却少了一项。我不停地返回这个巨大的MapReduce，检查输出，把它放进GDB，做单步调试。“到底怎么回事？什么都没变！”最后我看了下路径，天啊！我不知道为什么我在这个问题上花了90分钟，我当时一定是鬼迷心窍了，就是没想到回过头来检查一下命令行是不是正确。

还有很多类似的情况。我们总会在代码里用些Perl的语法糖，例如没有文法作用域的$_。但如果你在排序中用$_，你就会把别人的东西搞得乱七八糟。我们永远都摆脱不了这个bug，总是有东西弄错。最后我们把它解决了，我审查了所有的代码，定了条新策略：永远都不许这么干。

Seibel：你们的调试工具是什么？调试器？打印语句？还是别的东西？

Fitzpatrick：如果环境允许，我会选打印语句；要是所处的环境有好的调试

器，那就用调试器。Google把GDB维护得很好，当需要时，它真的是无可替代。我尽量不去用它，虽然我对GDB不是很熟，但通常只要看看，我就能找出问题所在。如果一定要深入其中，我有自己的办法。我喜欢strace，没它我都不知道该怎么活。当我不知道某个程序在做什么，或者我自己的程序在做什么，就用strace来跑一下，看看究竟发生了什么。要是只能选一个工具，我想就是strace了。还有Valgrind、Callgrind等工具都很好。

好多次有奇怪的问题发生时，我会说："OK，那个功能太大了，我们把它拆开，独立进行单元测试，看看我的假设哪里错了，而不是没头地打印。"

然后，在重构的过程中，我会更多地去考虑代码的细节，慢慢地就清楚了。这时，我可以回到那个庞大的、丑陋的函数，修复它，但我其实已经重构到一半了，我也可以为下一个维护者继续进行简化。

Seibel：你在代码里是怎么用不变式的？有些人把它扔在单独的断言中，有些人每步都会放上不变式，以便能证明程序中的形式化属性，在这两种极端情况之间还有很多做法。

Fitzpatrick：我尽量不走形式化路线。我的基本规则就是如果它来自终端用户，那就不算运行时崩溃。但如果是来自我的代码，那我就要尽力让它早崩溃，尽可能早地失败。

我尽量以前置条件的方式来思考问题，在构造器和函数的开始时进行检查。如果可能的话，调试检查，以此保证编译通过。有很多思维流派，也没人教过我怎么做才算合适。有些语言将所有这些内容都作为语言的形式化部分，基本上我用过的所有语言都是如此，选哪种就看你自己的了。

Seibel：你曾经写到过优化是编程中你比较喜欢的一件事，现在还是这样吗？

Fitzpatrick：因为优化不是必需的，所以它才有趣。如果你正在做优化，那么没什么比让东西跑起来更重要的了。你要么为了省钱而优化，要么就是因为它像Perl Golf比赛[①]——我能让它精简多少，或者快多少。我们会找出LiveJournal的热点，发起竞赛。"这里有些代码。这是性能基准。让它更快一些。"我给出负载均衡器的解析头。我们都埋头写疯狂的正则表达式，它们没有回溯，尽力用最有效的捕获组去捕获内容。大家都在互相竞争，让速度变得更快。第二天，一个哥们儿跑了过来，把所有的东西用C++的XS模块写了一遍，他说："我赢了。"

① 一个Perl的比赛，给定一个问题，寻找解决该问题的最短代码，参见http://perlgolf.sourceforge.net/。

Seibel：现在看那样做的负面影响是什么？

Fitzpatrick：程序员的时间更值钱，不该浪费在这里；究竟什么情况下这是对的呢？机器少的时候这是对的。一旦你有了很多机器，比起这个程序要部署的机器数量，程序员的时间一下子就不那么值钱了。因此，用C写代码，剖析出问题所在，修正编译器，再付钱找人用GCC让它编译得更快些。

Seibel：就算是Google也是使用C++而非汇编，是不是可以理解为有些地方并不值得去挖掘性能的极致。还是说好的C++编译器生成的代码比那些异想天开的稀有汇编程序员生成的要好？

Fitzpatrick：我们仍旧有些用汇编写的东西，但是非常非常地少。我们会对很多东西进行剖析，仔细确定是否需要将它从Perl改写到C，然后再由C改写为汇编。就算全是x86体系，还是存在很多x86的变体。你确定自己真的愿意为每种x86的变体写一段汇编吗？这个用SSE 2，那个用SSE 3.1。还是让编译器来处理这些吧。

Seibel：你还是个孩子的时候就从编程手册上学习编程。有没有什么书是你强烈推荐给新程序员的，或者是你觉得每个人都应该读一下的？

Fitzpatrick：我还在写Perl的时候——甚至是对那些很了解Perl的人——我都会推荐MJD（Mark Jason Dominus）的《高阶Perl》（*Higher-Order Perl*）。这本书真的很有意思，它先从简单的东西入手，你会觉得“嗯，我知道什么是闭包”，然后再慢慢被带入云里雾里。看完这本书后，你就会有极深的印象。尽管我知道所有这些内容，读这本书还是彻底改变了我的想法。我把它推荐给了很多朋友，都让他们对Perl有了新的认识。总的来说，每本书都会给人带来不同的想法。我想这是我能想到的最近的一本书了。

Seibel：我看你在那里放了本《计算机程序设计艺术》，它看上去还没翻旧，你看了多少了？

Fitzpatrick：哦，我拿到那本书还不到五年，也许正好五年。我大致翻了下，有兴趣了就读一读。看这本书之前，我已经从C.S.课中学到了书中的很多内容了。因此，也许这本书放在以前会更有价值，但我在联上因特网前并不知道这本书。

Seibel：你觉得程序员需要了解多少数学知识？读Knuth的书并要真正理解它，需要很好的数学功底，但作为程序员真的需要这些吗？

Fitzpatrick：你并不需要这么多数学知识。对大多数程序员而言，在日常工作中统计知识更重要一些。如果你是做图形相关的工作，数学会更重要些，但对于从事Java企业级应用或Web开发的人来说就不是了。逻辑和统计知识会更有帮助。

Seibel：你显然还很享受编程带来的乐趣。但我读了你在大学时LiveJournal上写的一些内容，看起来你承受了不少的压力而且讨厌电脑。

Fitzpatrick：哦，我其实一直讨厌电脑。在这么长的一段时间里，我并不认为我们真的取得了多少进步。电脑看上去比以前更慢、更容易崩溃而且问题更多。不过我是个乐天派，我还是相信它们会变得更好。看起来我十年前使用电脑的经历比现在要好得多，我十年前的电脑要更快些，工作得也更好。硬件变快的同时，软件却变慢了，而且漏洞百出。

Seibel：为什么你会这么想？

Fitzpatrick：我不知道。是门槛变低了？还是电脑太快了，让你不用再讲效率了？亦或是你无须知道自己在做什么了？我不清楚。也许是上述多种情况，也可能是存在太多的抽象让你不知道下面究竟发生了什么，因为电脑实在是太快了，它掩盖了人的愚蠢。

Seibel：按照现有电脑的速度，也许有些东西还未达到它应有的速度。但十年前，作为用户，人们根本没办法做到今天能用Google做到的事。

Fitzpatrick：是的。所以有人会写高效的代码并加以利用。我不玩游戏，但偶尔会看人玩，我会说："天哪，这怎么可能？"它征服了我。显然，有人做得很好。

我猜我是对我台式机的状态失望透了。在后端有那么多有趣的东西，但当我在用我的电脑时却对它越来越失望。我的Mac不应该总出现那个海滨球[1]。

Seibel：你有没有兴趣写更好的桌面软件？

Fitzpatrick：问题是没人会去用桌面软件。你想有人用你写的东西，就该做成Web应用。如果有一天我把笔记本电脑弄丢了，有人可能会问："噢，上帝啊，你有没有掉什么资料？"其实那上面根本没有我的文件，它只是个因特网终端，一个加密磁盘，我并不担心我的密码、Cookie或类似的东西。我不认为人们会去下载程序。

① Mac OS等待时的鼠标图案。

Seibel：*你的动力更多是源于拥有用户，还是出于编程的乐趣？*

Fitzpatrick：确实有些东西是为我写的，而且只为我一个人，我是唯一的用户，如果我有补丁或其他事要做时，对它的关注就会少些。但很多时候我愿意和别人合作。有用户是获得贡献者的关键之一。有更多的用户就能找到更多的问题和更多的用例。和他人合作起来也更有意思，特别是在做开源的东西时。

每当看到有人写信说“嘿，我们正在什么什么上用你的软件”时感觉都会很棒，这太酷了。当我看到使用memcached或负载均衡器或别的我写的东西的网站数量那么多时，我会说：“啊，那太酷了。”还有那些成人站点也告诉我他们正在使用我的文件系统。好吧，这也说明了一些东西，我在帮助建设成人站点，呵呵。在Craigslist上，每个请求通过的Web服务器可以算是memcached的一个前端。这太酷了！

Seibel：*你认为程序员过分热衷于新鲜事物吗？新的语言，新的工具，任何新的东西？*

Fitzpatrick：可能是这样吧。我不知道要是没什么可期待的是不是件让人失望的事，比如说我想要门新语言，让它实现我们要的所有功能。用户也这么想，他们总想要更高的版本，就算它可能会更糟。

我不知道从统计学角度来看程序员是否通常都有别于常人。新的东西必须比旧的好。尽管人们是这么希望的，但理想和现实往往有些距离。

我记得以前和我的牙医聊过一阵子，她总是不断地说着五年来牙科方面的进步，她对那些进步感到很激动。

Seibel：*想要成为一个当代程序员，就要找到正确的东西——你要用的东西，并理解透彻。这方面你是怎么做的？*

Fitzpatrick：CPAN[①]上有各种东西，那里光ID3解析器就有14个，选一个就好了。

Seibel：*从某方面来说，这也是当代程序员所面临的一个问题——有14个可供选择，你该选哪个呢？*

Fitzpatrick：用Google搜索，看看哪个排名高，哪个是人们更偏爱的。再去认识些人。我深深扎入开源社区，一切都是从参加所有这些会议开始的，因

---

① Comprehensive Perl Archive Network，http://www.cpan.org/。

为这样我能认识不少人，知道谁值得尊敬，谁比较酷。

然后再去看他们的代码。我记得一个家伙，他棒极了，既有趣，又友好，还很体贴，真的很关心自己的代码。当有人抱怨他的代码时，他会很激动。我会选这个软件，因为如果我发现任何问题，我知道他会积极地修正它们。那些性情乖戾的家伙则相反，他们也许能写出很好的代码，但脾气很坏，当你遇到问题或错误时不太好交流。因此要选一个你信任或敬佩的维护者。

Seibel： *有没有什么讨巧的方法能快速找到满足你需要的东西呢?*

Fitzpatrick： 刚开始时，我不会直接把它放进代码里，我会先写个测试程序试用几个我知道会用到的函数，确认它们能用。或者为那个库写一个单元测试，用将来要用的数据测一下。很多库甚至都没有自己的测试。即使有测试，也许你在读过文档后也无法确信它真的做了它承诺的事，或者是文档对它的行为描述得不够清楚。所以我亲自为那些我关心的东西写测试。反正都要通过写些什么来了解库的用法，因此我的第一个Hello World程序也可能是一个单元测试。

Seibel： *谈谈你正在使用的工具吧。你还是Emacs的用户，对吗?*

Fitzpatrick： 我依然是Emacs的用户。我希望自己能把Emacs用得更好些。我知道所有的按键，但很少做自定义。我会拿别人的自定义配置。当发现自己对什么东西不满时，我会说“我要写些Elisp来做键绑定”。但过一会儿就把这事给忘了。

Steve Yegge正在做的项目基本能用JavaScript来代替Elisp。我会等他的，这样就不用再学一门语言了。我就用JavaScript来做这事。我觉得JavaScript这门语言挺好的，有问题的是浏览器。在Google我用JavaScript写了不少东西，然后嵌入Java和C++中。我觉得JavaScript是一门很好的嵌入语言。

Seibel： *还有什么工具是你平时一直在用却十分痛恨的？除了你的台式机。*

Fitzpatrick： 是的，我讨厌整个台式机。我的台式机上有很多东西，那些浏览器总是会挂住、崩溃掉，还会占用大量内存。整个操作系统也会挂住。我的同事看到我用Emacs时，就试着说服我，Eclipse和IntelliJ能自动替我完成这些事。我每隔6个月会选择其中一个试用一下，Eclipse或IntelliJ。那该死的东西就停在那里，消耗内存，也许在我敲击键盘的时候会崩溃，不让我打字。来吧，突出显示后台语法，或在另一个线程中进行编译。为什么一定要打断我来做这些呢？OK，我过6个月再来试试吧。我很高兴不用被迫使用这

些工具。我真该把Emacs再用用好。

我的学习曲线是这样的，学东西很快，直到我学好它而且相当高产为止。随后我会维持在一个80%～90%的稳定水平，这时我效率很高，不用去查阅资料，我很开心。在那之后情况还会慢慢变得更好。只有到我觉得过度安稳了之后，我才会想："我要去仔细看看这门语言的文档（man页面），了解所有细枝末节的内容。"

Seibel：*这么做明智吗？有太多东西要学，你可以花一辈子学习怎么使用编辑器，那你又能写多少软件呢？*

Fitzpatrick：是的，但我发现——至少对编辑器而言——花掉的时间总是会有回报的。无论我学什么，总会有回报的，也许是一两周内就会有。当我在bin目录里写些很傻的shell脚本、Perl脚本，或者别的东西做自动化，它一定会有回报的。

Seibel：*那么说来你从没陷入无止境的工具开发中吗？*

Fitzpatrick：没有。我总是出于某种目的去开发工具的。我确实认识一些人，他们一直在开发自己的工具却从没拿出一个成品。我会朝着这个方向再更进一步，而且很安全。

Seibel：*你觉得对程序员来说什么才是最重要的技能？*

Fitzpatrick：像科学家那样思考，一次改变一样东西。有耐心，试着去了解问题的本质。尤其是在调试或者设计那些不太正常的东西时更应该这样。我看到过年轻程序员在那里抱怨"哦，见鬼，这个东西运行不了"，然后就把它彻底重写了。其实应该停下看看究竟发生了什么。要学会增量地开发，这样每一步你都能进行验证。

Seibel：*作为一个程序员，你会不会特意做点什么来提升自己的技能呢？*

Fitzpatrick：有时我会不走寻常路，虽然知道会花费更多的时间，但还是用一门我不太愿意用的语言来写东西，因为我知道最后这对我是有好处的。比如我刚到Google时，要写什么东西，我都会用Perl。然后我觉得"啊，不对，我应该用Python来写的"。现在我会用Python来写很多东西，这不再是我的弱项了，我甚至很少需要去查资料。Perlbal本来是用C#写的，我就是想学一学C#。

Seibel：*除了编程本身，还有什么技能是那些准程序员应该学习的？*

Fitzpatrick：沟通技巧，不过我不确定那是个能练习提高的东西。多和人在邮件列表里交流。书面沟通风格的养成需要很长时间。但这是一辈子的事，不是吗？有一项关于高中毕业生中成功人士的研究，是聪明孩子好，还是会交际的孩子好呢？结果证明，那些会交际的孩子一生都能赚钱，而不是那些成绩好的。我觉得这个结果很有意思。

Seibel：*看起来现在的情况和以前有些不一样了。以前的程序员更像是躲在办公室里的侏儒，而现在到处都是邮件列表，大家都在谈论协作。*

Fitzpatrick：在我工作的地方，无论是开源社区还是公司里，大家都相互依赖。我们的动机是"因为我知道过两周你需要这段代码或者我需要你的代码，所以我动手来写这个代码。"人与人就是这么相处的。

Seibel：*人们总说最好的程序员和最差的程序员在生产力方面有着天壤之别。根据你的经验来看是这样的吗？*

Fitzpatrick：是的，基本在所有的领域都是如此，这取决于你有多少经验。据我所知，通常的情况并不是两个人花同样的时间就能有同样的编程产出，时间差可能会有十倍之多。如果你不能时刻保持良好的状态，那你就可能感到疲倦，然后出局。

我觉得有些人只是在工作，而不是享受编程带来的乐趣，这没什么问题。但把他们和那些核心程序员做比较就不行了。当一个人花的时间多十倍，不停地考虑怎么写好这个程序，另一个人只是为了工作而工作，那两者生产力上的差距又岂止十倍呢？

Seibel：*你前面提到用科学家的方法来进行调试。你觉得自己是一个科学家、工程师、艺术家还是工匠呢？*

Fitzpatrick：科学家或者工程师吧，也许我更像个工程师，科学家排第二，但你必须要懂得科学的方法，一次改变一样东西，如何诊断问题。工程师是指设计方面的。我也有些朋友自称为艺术家或工匠的，但我从来没这么想过。

Seibel：*另一方面，软件领域里也有很多工程方面的问题。有这么一个笑话：如果人们用造软件的方法来盖摩天大楼，那第一只啄木鸟就能毁掉文明世界。你认为软件生产是一门非常成熟的工程学科吗？*

Fitzpatrick：不，我觉得没到那个地步。你不需要执照就能写代码。虽然我不想要那么多规矩，但如果能保证那些留下XSS隐患的PHP程序员不是写空中交通管制系统的人就太好了。我很希望对这些人能有个官方的界定。

我有个朋友是建筑工程师，他一直去学校充电，考各种工程学相关的认证。你站在桥上，想起造桥的人花了毕生的精力学这东西，参加了无数考试，一直在学习，这种感觉很好。

Seibel：那你能给程序员什么测试，让你相信他们能写出正常工作的软件呢？

Fitzpatrick：我不知道。这有点儿吓人。

Seibel：就算没有执照，你觉得程序员对社会有什么道德上的责任吗？程序员是一个职业，是职业就有自己的职业操守。

Fitzpatrick：你不能谋杀他人，比如你写的飞行控制软件出了问题，但那只是极少数的情况。我一般会要求大家写的信用卡表单能一致，比如能让我在其中插入空格或连字符，电脑很合适处理这种情况。又比如不要告诉我如何格式化我的数字。但这里没有什么道德标准，有的只是愚蠢的行为。

Seibel：你今年28岁了。有没有担心过编程是年轻人的游戏，当自己老了就会略逊一筹？

Fitzpatrick：没有。最坏的情况就是我停下来，自己找点乐子。我不觉得自己现在是在和谁竞争，我也不太关心别人是不是比我更好，因为我觉得已经有无数人比我好了。我发现我们总是处于中间位置，而我也很乐意保持在这个位置上。

Seibel：这么说来你是因为觉得有趣才编程的，即使你不再工作了？

Fitzpatrick：是的。我还是会继续做些小玩意儿。我的手机上有个无聊的棋类游戏，我对它有点厌烦了。我不太会正儿八经地去干什么，所以就写了个求解程序。其中试用了一些动态编程，处理了不同的棋盘大小，还有很多随机棋盘，针对不同的棋盘大小还给出了解决棋局所需步数的直方图。我把它发给了作者，因为游戏里的标准估算真的很糟糕。基本上你想在游戏中玩下去，必须要超过平均水准。邮件列表里的每个人都发现游戏玩到后面变得容易了，不知道这个标准他是怎么弄出来的。于是我把每种棋盘大小的直方图发给了他，我想他在新版本里会有所调整吧。这是件很有意思的事情，我想我可以退休后整天做这些事。

（编辑：谢灵芝）

3

Douglas Crockford

张龙　译

Douglas Crockford是Yahoo!的资深JavaScript架构师，他在上世纪70年代初求学期间就开始从事程序开发工作了，那时的他主修电视广播专业，但苦于无法进入演播室工作，转而学习了学校开设的Fortran课程。在其职业生涯中，Crockford曾先后供职于Atari、Lucasfilm和Electric Communities，以各种方式联姻计算机与传播媒介，现在他就职于Yahoo!。

Crockford生来就是一个至纯至简的人。深感于XML的复杂性，他发明了JSON这一广泛用于Ajax应用的数据交换格式。Crockford在最近出版的新书*JavaScript: The Good Parts*中谈到如果能避免使用某些特性的话，JavaScript实际上是一门相当优雅的语言。在接受采访的时候他强调了以子集方式来管理复杂度的重要性，同时介绍了他所使用的一种代码阅读方法：从清理代码开始。

在采访之际，Crockford就已经因极力反对将ECMAScript 4（ES4）纳入到ECMAScript（JavaScript）语言标准中而名声大噪，因为ECMAScript 4实

在是太复杂了。他倾向于更加简洁的ES3.1提案，最后Crockford与其他ES3.1拥护者大获全胜——ES3.1更名为ES5而ES4则被官方彻底抛弃。

在本次采访中，Crockford谈到了ES4提案的缺陷，作为团队行为的代码阅读的重要性，以及在现有系统当道的情况下如何推进Web向前发展。

Seibel：*你是如何走上编程之路的？*

Crockford：我在旧金山州立大学度过了大学时光。之所以选择这个学校是因为他们有个很棒的电视专业。大一的时候我无法进入到演播室工作但又想找点事儿来做，机缘巧合地学习了数学系开设的Fortran课程。结果我发现自己很擅长编程，于是又学习了第二学期的课程。

那个时候还是1971、72年，图书馆地下室的机器还在使用穿孔卡片，分时系统也刚刚走进校园。旧金山州立大学工程系不够强势，它并不是全校唯一拥有计算机的系，其他院系也都在使用计算机。自然科学系有计算机房，商学院有计算机房，教育学院有计算机房，人文学院也有计算机房。所有这些学科都在使用计算机，这很有意思。

我最先去的是自然科学系机房，后来又去了人文学院机房，因此有机会见到很多搞经济学、心理学和地理学的朋友，他们都非常有意思。在那里我有机会接触到他们所要解决的问题，这让我很早就从一般人的角度领悟到一些东西，知道他们在使用这些糟糕的机器时心里想的是什么，同时也开始思考如何才能让这些机器更好地服务于人。

最后，我进入了演播室并成为一名电视人，这项工作很有趣，但我最终还是选择了计算机之路。一直以来，我都在想如何才能将这二者结合起来。我之前对多媒体也就是现在的数字媒体寄予过很大希望，在职业生涯的不同时期，我也不停地穿梭在媒体与编程之间。

Seibel：*你最开始学的是Fortran，然后发现自己在编程方面很在行，那么除了“喔，我发现自己很擅长编程”这种理由之外，还有没有其他原因促使你走上编程之路呢？*

Crockford：其实这就是全部原因了。那是第一个学期，我需要选一门数学课，结果我就随便选了一门，碰巧是Fortran。我并非有意学习编程，应该算机缘巧合吧。

Seibel：*你还记得自己编写过的第一个有趣的程序是什么吗？*

Crockford：那是很早之前的事了。我曾经写过一个程序，可以反汇编我所使用的分时系统上的Fortran系统运行时。通过编写那个程序，我基本上理解了Fortran系统的工作原理，也在其模型基础上自学了大量的编程知识，这其中有些内容当时甚至并没有公开发布。

Seibel：*与那时相比，你觉得自己对编程的理解的最大变化是什么？*

Crockford：过去有一个时期，前后有大概十年左右吧，非常非常重视效率。我想那是在早期的微处理器时代，内存还非常小，CPU也非常慢。我们只能使用汇编语言来实现游戏和音乐等内容，以适应硬件条件并提升速度。现在这种境况已经一去不复返了，我们可以使用JavaScript编写大型应用并运行在浏览器中。与过去的环境相比，浏览器的运行效率非常低，但有摩尔定律的保证，这一切都不再是问题了。

Seibel：*在程序学习之路上有哪些令你后悔的事情？*

Crockford：我了解一些语言，但却一直没有机会使用。我花了不少时间学习APL并了解到其衰败的原因，但这门语言真的非常优雅，可我却没有花时间使用它，这太遗憾了。除此以外，我还了解其他一些语言，知道能用它们做什么，但实际上却并没有机会用这些语言思考。

Seibel：*实际上你获得的是广播专业的学位，毕业之后你又做什么了？*

Crockford：我后来读研了，专业是教育技术学。但我发现自己早已具备了该专业所要求的技能，在那儿读书也是浪费时间。大概一年后我离开了，成为门洛帕克市斯坦福研究院（SRI）的一名研究员。之后，我又跳到了一家名为Basic Four的公司，这家公司主营小型商用计算机，我在那儿呆了很久。在这期间，我为公司开发了一款字处理系统，并开始研究起便携式计算机和PC。我尝试着推动公司进入PC市场，我在公司买了第一台PC并把它放到桌子上一直开着，这样那些工程师就能随时看到机器了，看看IBM是怎么做的，但遗憾的是，我无法改变公司文化——他们这帮人绝对是鼠目寸光。

接下来的圣诞节，大概是1981年吧，我买了一台Atari 800。当我走进计算机商店时映入眼帘的是一台Apple II和一台800，后者看起来更时髦一点，于是我买了这台。我想我应该编写一个字处理软件或是为其开发一种编程语言。但6502什么也做不了。我花了2000美金只知道了一件事——这机器到底能做什么？显然，可以编写游戏。于是我开始编写计算机游戏，还卖给Atari一款，接下来他们邀请我到其位于森尼维耳市的研究实验室工作。这个实验

室是由阿伦·凯创立的，这也是离开PARC以后他做的第一件事。我接受了他们的邀请并奔赴那里，那里简直太棒了。我在那儿干了两年，亲眼目睹了公司的倒闭。但我在那里确实做了不少有趣的事情，周围的同事也非常好。

Seibel：*在这之前你是个游戏迷吗？*

Crockford：我在《太空人侵者》和《吃豆人》这两个游戏上花了不少时间。我喜欢游戏，但却不是一个游戏狂。游戏吸引我的地方在于它是电视与计算机的另一种联姻方式，公众也首次通过游戏参与到了这种交互当中，我觉得这非常有趣。

Seibel：*Atari倒闭后你去哪儿了？*

Crockford：我去Lucasfilm了，在那儿呆了8年。

Seibel：*你在Lucasfilm的时候Habitat项目启动了。*

Crockford：没错。我的朋友Chip Morningstar创立了这个项目。他发明了avatar[①]和图形虚拟世界，所有这些都是他首先实现的。该项目运行在Commodore 64s和off-peak x25网络上。真惊讶于这富于远见的设计——他做的这些都是正确无误的，真令人感到吃惊。整个过程我只充当一个旁观者，看着他们实现了这些并不断鼓励他们，但这些荣誉并不属于我。

Seibel：*接下来你跟他们基于这些想法而创立了Electric Communities？*

Crockford：是的。后来Morningstar和Randy Farmer离开Lucasfilm并创建了一家名为American Information Exchange的公司，把为社会服务的想法应用到在线市场。这是个杰出的想法，但有些太超前了。如果再晚一点，他们就成为eBay了。

后来我们想，可以用这个想法做一个通用平台，可以实现娱乐、社交、业务、商业等所有事情，该平台是面向全世界的。我们有一些点子能够实现完全分布式的平台，这样就不存在单一的服务器了——而是整个网络都是服务器。我们还提出了安全模型，可以实现完全分散化的平台。这是个非常宏大的想法，也是Electric Communities的根基。

Seibel：*这就是E的首个版本的缘起。*

Crockford：没错。我们需要一种安全的编程语言来开发平台和其中的应用。

---

① 此avatar并非电影名称，而是通过动画来表示的人物形象，具体解释参见http://www.fudco.com/chip/lessons.html。

首先我们尝试了Joule，它是由另一家公司Agorics开发的。Joule是一种Actor语言，在使用方式上非常怪异——它很棒，但却不符合人们的习惯。

我们对Joule怀有顾虑。能否说服人们来使用这门语言呢，它是不是太怪异了？后来我们想到了E，它抽取出了Joule核心的Actor概念并用Java重新实现。

Seibel：*除了发明者以外，还有没有其他人使用过E呢？*

Crockford：还没有其他人使用过最初的E。老式的E是个Java方言。Sun遇到的问题也是我们所遇到的问题。后来我们提出了E脚本语言，它更加轻量级，但却拥有相似的属性。现在所说的E指的就是该脚本语言。

我们在Electric Communities开发出了这门语言，但我觉得好像没人使用过它。我们曾一度认为不会再使用这门语言了，但它确实还不错，因此勉强用了一段时间，我很欣喜地看到它还是存活了下来。

我在Electric Communities的一个收获就是理解了闭包。在进行Web开发时，我发现JavaScript中很多地方都有闭包的影子。JavaScript中的很多东西都源于Scheme，但如果看过文档的话就会发现它根本就没有提到闭包的概念。我是偶然间才发现闭包的，"唔，这太棒了。"我深信这门有点蠢的语言实际上是可以登上大雅之堂的。

Seibel：*那来谈谈最近关于ECMAScript 4的争论吧。我听说你喜欢ES3版本JavaScript的简洁性。*

Crockford：嗯，最终无论怎么对语言进行修订，其要义都是希望促进语言的不断成功。语言越成功，修改的代价就越大。随着你的不断成熟，再教育的成本就会变得更大，同时还有潜在的破坏代价，而这些成本和代价也会变得难以接受。如果你确实非常成功，那就更要小心提防所做的任何变化了。反之，如果你尚未成功，那么就有更大的自由空间来改变了。

JavaScript成为世界上最流行的编程语言纯粹是偶然。目前世界上JavaScript处理器的数量要高于任何其他语言。得益于其安全模型带来的种种问题，JavaScript是唯一一门可在任何机器上编写并运行的语言。

这些还嫌不够的话，再看看那么多嵌入了JavaScript的应用吧。Adobe的大多数应用都嵌入了JavaScript，这样就可以在本地编写脚本控制这些应用了。还有其他很多应用，不胜枚举。这么一看，JavaScript已经变得非常流行了。

JavaScript这门语言的问题在于推向市场以及标准化的过程都过于匆忙了。其大多数缺陷都没有出现在目前的实现当中——只存在于规范中。标准

说照错的做，这听起来太吓人了，但这就是JavaScript的状态。它于1999年冻结了，接下来本应走向灭亡。但Ajax的横空出世改变了这一切，JavaScript变成了世界上最重要的编程语言。

于是，我们现在认为应该修复它。但这事应该是在2000年就开始做的，而那时并没有这么做，因为根本没人关注JavaScript。现在它已经长大了。

Web环境下的JavaScript还有一点非常怪异：如果编写服务器端应用、桌面应用或是嵌入式应用，你不仅需要选择语言，还要选择特定的编译器以及特定的运行时。但对JavaScript而言你别无选择，你必须在所有的环境下运行。

由于要在所有环境下运行，bug就没法修复了。如果某个浏览器厂商搞出个bug，他们会说"天啊，搞砸了"，下个月就会发布另一个版本，但我们却不能指望着所有用户都会升级。大多数人一旦在机器里装上IE就再也不会升级了，那些bug就会常年驻留在浏览器上。

Seibel：这就是目前的状况。你希望Web能成为更适合于应用开发的平台。除非所有浏览器都能修复这些bug，否则我们是没法修复的，如果这还不管用，那实在是没办法了。路在何方呢？

Crockford：这正是我努力争取的东西。我心中已经有一套理想的方案了，知道它要成为什么样子。我知道身处何方，也能看到前方的障碍。我正在思考如何才能前行。从某种意义上来说，我们已经身陷囹圄了，因为我们开发出了这些大型系统——我更关注经济系统、社会系统，还有技术系统——它们都依赖于这个并不完善的系统。

毫无疑问，JavaScript的鸡肋就是对全局对象的依赖。它没有链接器，无法在编译单元间隐藏信息。它将这些信息都丢到了一个普通的全局对象中。这样，所有组件都能访问一切内容，对DOM拥有相同的访问权限，对网络也拥有相同的访问权限。如果有人将脚本放到了你的页面上，它就可以访问服务器，看起来就像是你自己的脚本一样，而服务器则根本无法分辨。

这些脚本可以获取屏幕信息，可以访问用户信息，看起来就像是你自己的脚本一样，用户同样也无法分辨。在页面来自于你的服务器，而且无论脚本来自何处都拥有同样权限的情况下，用户所有新式的反钓鱼工具就都派不上用场了。

实际情况比这还要糟，因为脚本还可以通过其他方式进入你的页面。Web架构涉及几种语言——有HTTP、HTML，URL可以看作是一种语言，

有CSS，还有脚本语言。它们可以彼此嵌入并且具有不同的引号、转义和注释约定。所有浏览器对这些语言的实现都不一样。加之这些实现的细节并不是在哪里都可以找到，这样恶意用户就能够轻松将脚本放到URL中，放到样式中，放到HTML中，放到其他脚本中。

Seibel：*这就是典型的跨站点脚本攻击，利用了浏览器的bug。*

Crockford：是的，这很可怕。我们必须修复这个问题——不能再容忍这个问题继续存在下去了。

我们在此之上找到了混搭这种解决方案。混搭实现了我们在软件领域探索了20年而未得的一些东西：可以像乐高积木那样将一些有趣、可重用的组件组合起来并快速生成新的应用。我们用混搭的方式实现了一些应用，非常不错。你可以从Yahoo、Google、自己和其他人那里获取组件并组合起来生成应用，这简直太棒了。所有这一切都发生在浏览器上，就在你眼前，只不过这些组件依然拥有对同一目标的访问权限。现在我们可以有意发起XSS攻击了。浏览器的安全模型并没有料到这种好事，也不允许在彼此存疑的情况下进行合作。整个Web构建在一个接一个的错误之上。我们遇到太多的意外了。

Seibel：*这么说来人们在ES4上所付出的仅仅是机会成本，每个人都花时间来思考这个问题而不是寻找问题的解决办法？*

Crockford：没错。ES4的目标根本就不对，它想解决的是人们厌恶JavaScript这个问题。我很欣赏Brendan Eich对待这个问题的立场，他确实做得很棒，但却太着急了，没有管理好这个项目，放出了ES4这么差劲的东西。在过去的12年里，有很多人诋毁和诅咒他，觉得他是个蠢蛋，搞出来的语言也愚蠢至极，但我觉得事实并非如此。这个语言确实有闪亮的地方，而他其实是一个才华横溢的家伙。现在他在极力表明自己是无辜的，想要证明自己是个聪明人，他想要展示这门语言的优秀特性，将这些特性整合起来，这样它就能好用了。

我觉得这并非现在需要解决的问题，亟需解决的问题是：Web正变得分崩离析，我们需要对其进行修复。这要求我们清楚路在何方。我最反对Brendan的一个地方就是他的方案完全没抓住重点。

我越来越感觉到这是个问题。如果能够模块化，如果能够自己选择编程语言，我们本可以走得更远。但现在这一切都尚未实现，不过现实情况比这还好一些。现在有Caja和ADsafe这样的东西在使用目前的技术解决这些问题。时不我待啊。

ADsafe创建了JavaScript的一个安全子集，它不允许访问任何的全局变量和危险的东西。这么做的结果是这个子集依然可以构成一门有用的语言，因为有强大的lambda作为后盾，它可以完成很多事情。它是一门非传统的语言，它不支持我们今天使用原型的方式，但这个子集却是一个功能完整的lambda语言，非常强大。

Seibel：*先不考虑ES4目标的正确与否，仅仅从语言的视角来看，它有没有哪些特性吸引到你呢？*

Crockford：有一些bug修复我觉得很不错，值得保留。但ES4中有太多未经验证的东西了。我们使用ES3的经历表明，一旦规范中出现了错误就很难再把它剔除出去。我们并没有ES4的使用经验，没人用它开发过大型应用。

在真正应用到实际工作中之前需要对其进行标准化和部署，我觉得现在的节奏太快了。如果有多个参考实现，人们也开发出了一些有价值的应用，这才表明语言是没问题的，接下来才对其进行标准化并部署起来。但现在的做法却是反其道而行之。

Seibel：*Google的GWT会将Java编译为JavaScript。还有人尝试将其他语言编译为JavaScript。这是未来之路么？*

Crockford：看到JavaScript逐渐变为通用运行时还是蛮有趣的一件事，这一点倒是我们不曾想到的。

Seibel：*但正如你所说，JavaScript无处不在，它确实是通用运行时。*

Crockford：我认为这也进一步说明JavaScript确实该跑步前进。尤其是现在我们正在进入移动产品的时代，但摩尔定律并不适用于电池。JavaScript在解释上所花费的时间愈发显得重要，此外还有周期数。我认为这会更加督促我们来改进运行时的质量。

就GWT和其他转换工具而言，我的态度是实用至上。人们是很难融入到这个环境中的——如果能找到好的解决方案，那非常棒。我自己是害怕使用这些工具的，担心的就是抽象层泄漏（abstraction leakage）。如果你的Java代码、GWT或是其生成的代码存在问题，那可能就没法解决了。特别是在完全不了解JavaScript的情况下就贸然使用这种方法会更加危险，因为GWT对你完全隐藏了JavaScript。在这种情况下一旦出现了问题，那受伤的肯定是你自己。虽然我还没有听说发生过这种事，到目前为止这些工具还不错，但毕竟存在着风险。

Seibel：你希望JavaScript有哪些变化呢？

Crockford：我认为改进JavaScript最好的办法就是瘦身。如果我们能够取其精华，弃其糟粕，那JavaScript会变得更棒。我认为这个办法也适合于HTML、HTTP和CSS。我们应该仔细思考所用的各种标准，搞清楚需要哪些特性，遗漏了哪些特性并重新审视它们，绝不应该盲目地增加新特性。

Seibel：尽管小巧、璀璨的珍珠与散乱的泥球是截然不同的，因为一颗完美的小珍珠是很容易看清楚的，也没什么缺陷，但接下来你需要对其进行加工并增加更多的东西。这样每个人都重复实现同样的东西，导致不断地膨胀，最后出现与先前完全不同的丑陋结果。

Crockford：但这并非实际情况。我们有很多Ajax库的开发者，他们做的就是这件事，并且对语言的使用已经达到了炉火纯青的地步。后来社区中的一些人在上面草率地进行开发，能用就行。并非所有的应用开发者都有必要完全了解lambda并充分利用语言的lambda特性。我们已经这么干了，没必要因噎废食——这并非问题所在。

问题在于现在有太多的Ajax库了。造成这种结果的原因是JavaScript太强大了，人们都需要Ajax库，而这些库也很容易构建。有那么一段时间，每个人都在创建Ajax库。我倒是希望能经历一个动荡时期，但这个时期尚未到来。因此我们依然有太多的库可用了，这导致了另一个问题——由于有太多的选择，开发者都无从选起了。我认为最终一定会出现一个动荡期。

我们现在看到众多的Ajax库正汇集到一起。jQuery使用CSS选择器从DOM中获取对象列表并提供了批量操纵对象的方法。事实证明这个想法非常棒，也提升了JavaScript的效率。DOM接口让人感到恐怖，这也是低效之源，但jQuery将这一切都隐藏起来，真正简化了编程模型——太棒了。

现在所有人都在这么做——我们看到各种特性正不断汇集。结果就是用户社区很难抉择到底应该使用哪个库，因为这些库变得越来越相像了。但最后这些库会合并成几个，或许只有一个。我曾预言微软的Atlas框架将成为最终的胜利者之一，因为微软在各个领域都有获胜的先例，但实际情况是他们并没有引起人们的注意。开源框架似乎做得更好。因此我希望最终能有一两个开源框架脱颖而出。

Seibel：目前你在Yahoo！扮演着JavaScript架构师和布道者的角色，我想你的一部分工作是向Yahoo!的JavaScript开发者们传授“JavaScript应用之道”。那你的工作还涉及一般性的优秀设计实践与编码实践么？

Crockford：我一直在倡导良好的代码阅读方法。我认为这是社区中的开发者多花点儿时间阅读彼此的代码，这对每个人都非常有意义。现今的项目管理有这样一种趋势：让开发者们独立完成工作，接下来进行大规模的整合，如果没问题就发布出去，然后就大功告成了，接着就抛之脑后了。

这么做的一个后果就是一旦碰上差劲的开发者，到最后你才能发现问题，不过这时已经太晚了。项目会因此出现风险，在构建的时候才发现有些代码写得实在是糟糕，肯定会导致项目延期，而这是无法接受的。另外，项目中可能会有一些优秀的开发者，而他们却没有太多机会指导其他成员。代码阅读可以解决上面这两个问题。

Seibel：*能否详细谈谈如何进行代码阅读呢？*

Crockford：每次开会都让一些人阅读他们各自的代码，他们会引领我们查看其编写的所有内容，其他人则负责检查。对于团队的其他成员来说，这绝对是个学习的好机会，通过这个过程他们就可以知道自己的东西该如何与他人的相配合。

每个人都围坐在桌边，手里拿一叠纸，同时还把代码在屏幕上打出来，大家一起阅读。我们会在编写代码的过程中加上注释。有人会说“我看不懂这个注释”或是“这个注释与代码风马牛不相及”。大家的意见极具价值，因为作为开发者的你是不会阅读自己编写的注释的，你也根本没有意识到自己写的注释误导了读者。有这么多人帮助你编写整洁的代码是多么幸福的一件事啊——你会找到自己根本无法找到的缺陷。

我认为一小时的代码阅读抵得上两周的QA。这种剔除错误的手段真是很高效。如果你让能力很强的同事阅读代码，那么他们周围的新手们就会学到很多东西，而这一切是无法通过其他手段获得的；如果新手来阅读代码，那么他会得到很多极具价值的建议。

但这件事我们不能一直留到最后再做。回忆过去，我们会在项目完成之际安排代码阅读，但这个时候已经太迟了，只好取消。现在我深信代码阅读应该伴随着整个项目的生命周期。我花了很长时间才意识到这一点，这么做的好处不胜枚举。

首先，这么做有助于把控项目，我们能够真切地看到大家的进度，也能及早发现是不是有人已经偏离了轨道。

我曾经管理过一些项目，马上就到最后期限了，有人说“耶，马上就干完了”，然后我拿到了代码，发现里面什么都没有，有的也是一些垃圾，离

完成还远着呢。管理层最厌恶这种事情了，我觉得代码阅读能够有效避免这种窘境的发生。

Seibel：*你是说大家一起阅读我的代码。我把代码打印出来并打到屏幕上，那然后呢？我真要把它大声念出来？*

Crockford：对，一行一行地看，你可以对代码进行说明。我们这儿就是这么干的。如果有时间会逐行检查。

Seibel：*那你需要指导别人如何进行代码阅读么？可以想象，既不想让代码编写者感觉受到侵犯，又能给出颇具价值的意见是很难的。*

Crockford：没错，这需要给予团队成员充分的信任，要明确界定好边界。如果团队不和睦，那就别指望这么做了，这会导致团队分崩离析。如果还没有意识到团队的不和睦，那这么做很快就能发现。这个过程会让你学到很多，也会揭示出很多问题。起初会觉得不太自然，但一旦适应后就会觉得再自然不过了。

另外，我们要编写可读性好的代码。大家都知道，整洁很重要，而代码风格也同样如此。所有这一切会提升代码的质量并增强编程社区的能力。

Seibel：*如何编写可读性好的代码呢？*

Crockford：可读性有几个等级。最简单的一级是与表达保持一致，适当地保持缩进，在适当的地方使用空格。我有一个习惯来自于早年学习Fortran的时候，那就是，我往往会使用过多的单字母变量名，这个习惯可不好。我真的在很努力改掉这个坏习惯，但太难了——这么多年来还在与之斗争。

Seibel：*能有多难呢？难道你写完代码后还会回来检查“喔，看看那些单字符变量名吧”？*

Crockford：我惯用单字母思考问题。对于JavaScript来说，有这么一个关于效率的争论，需要考虑多余字符的下载代价，所以人们可以通过缩短变量名达到给程序瘦身的目的，但这个观点却站不住脚。

Seibel：*是不是可以通过工具解决这个问题？*

Crockford：压缩可以解决这个问题，因此在变量命名时根本不用考虑那么多。在回头检查以前编写的代码时，我发现变量名太短了，如果有时间我会修改的。类似于循环计数器之类的变量都是`i`，我倒不会修改这个地方的，但其他很多地方都是不可原谅的。

这是第一个级别，语法级别，类似于英文或其他语言的写作，要正确书写标点符号和大写字母，合理使用逗号。接下来看看下一个级别吧：如何组织句子、在何处分段。对于编程语言来说，这指的是如何将问题分解为功能或类的集合。

Seibel：*有什么具体的举措可以提升代码的可读性呢？*

Crockford：子集的想法非常重要，尤其对于JavaScript来说更是如此，因为这门语言包含了太多的糟粕，当然其他语言也一样。当还是个菜鸟时，我会翻阅语言规范并弄明白每个特性。我知道该如何使用这些特性并一直在用。但事实证明很多特性并非是深思熟虑的结果。

我现在想到的是Fortran，但其实所有语言都难逃这个宿命。有时语言设计者本身就错了。依我看来，C在设计上存在很多不妥之处。

Seibel：*比如呢？*

Crockford：拿switch语句来说，其默认的贯通就是个错误——不应该这么做。++存在严重的安全问题——它诱使你走向复杂，导致一行代码完成太多的事情，这会导致代码难以理解，还可能出现其他问题，比如缓冲区溢出错误。这些年来操作系统出现的大多数安全问题实际上都是++造成的恶果。

现在我的编程风格中已经不再有++了。我可以说什么时候使用++很好，什么时候就不行，但要在程序中去分辨哪里好哪里不好很难。

Seibel：*是不是可以这么说，++导致的安全问题实际上与++本身无关，而是由未检查的数组越界或原生指针造成的？在Java中就不会造成安全隐患，因为++导致的数组越界会抛出异常。*

Crockford：没错，Java中的风险要低很多，而JavaScript则根本没有这种风险，因为它并没有数组。但即便这样，我发现不使用++还是会提升代码质量，因为否则它会诱使我在一行中编写过多的代码，而这么做通常都不太好。

此外还有continue语句。我还没发现哪些代码离开了continue就没法实现。continue会简化某些复杂结构的编写。但我发现如果不使用continue代码结构会变得更好。作为自律，我不再使用continue。如果在代码中发现了continue，我会认为自己并没有经过深思熟虑。

Seibel：*你如何阅读别人编写的代码呢？*

Crockford：清理。我会把代码放到文本编辑器中并开始修复。首先，我会

统一标点符号，适当缩进，等等这类的事情。我有些程序可以完成这些事情，但从长远来看自己完成会更加高效，因为这有助于我加深对代码的理解。Morningstar曾教会我如何完成这些事情。他非常善于重构别人的代码，并且这也是他所使用的方法，很好。

Seibel：*你是否遇到过这种情况：看到代码写得一团糟，然后进行清理，但最后发现原来的代码其实写得很不错？*

Crockford：还没遇到过。我认为随随便便写出来的代码肯定不好。好的代码意味着可读性要好。在某种程度上，如果我搞不懂代码的意图，那么写得再好也没用，有可能代码在我不关心的方面表现得很好，比如很高效、很紧凑，等等。

代码的可读性是我的第一要义。它比速度还重要，可以与正确性一争高下，可读性是正确性的重要前提。如果可读性不好，那就不是好代码，代码的编写者可能做出了错误的权衡。

Seibel：*为了追求速度在内循环中嵌套内循环可以么？所有代码都要具备可读性？你是否遇到过为了效率而牺牲可读性的情况呢？*

Crockford：遇到过，但我会在两个循环末尾加上说明，解释这么做的原因所在，通常这会为人所忽视。我看到很多人苦心孤诣地想提高代码的速度，但很多时候根本就没必要。他们并没有意识到程序所花费的时间在哪里，只是优化那些根本无需优化的东西，程序的执行也根本不会经过这条路径，这种优化完全是无用功。所有的优化都会引入坏味道，这我见得多了。

Seibel：*在使用花括号的语言中，关于如何放置花括号这个问题总是争论不休，有人说这种风格可读性好，有人说那种风格可读性好。在“清理”代码时，你会不会将格式调整为自己喜欢的那种？*

Crockford：当然了，因为我相信只有我使用的风格才是正确的，其他人都不对！Thompson和Ritchie没有为C定义一种漂亮的展示风格，这是他们欠我们的。他们说“我是这么做的，但你可以不这么做”，这已经对人性造成了严重伤害，这种伤害还将持续下去。

Seibel：*你偏爱K&R风格？*

Crockford：是的，我认为他们是对的，其最初的风格没错，对于JavaScript更是如此。JavaScript会自动插入分号，这样如果将花括号放在左边而不是右边，那么程序的含义就会发生天大的变化，这种变化是很糟糕的。事实表

明K&R风格不会遇到这个问题，但flush风格会。

对于JavaScript来说，花括号是有着正确的摆放位置的；而其他C风格的语言则不是这样，哪种摆放位置都可以看作是正确的。有些人喜欢flush风格的花括号，我也看过有人就风格的正确与否争论了半天，但这些解释都毫无意义，因为他们真正争论的是自己在学校里用的是什么、在第一份工作中又用了什么风格，或者是影响过他的某人使用了哪种风格，那这种风格就是正确的，而其他则是错误的。

这就好比是关于应该靠左还是靠右行驶的争论，结果当然是无疾而终。如果住在孤岛上，靠哪边行驶都无所谓，但如果大家能达成统一确定走同一边，整个社区都会受益无穷。

Seibel：*假如你换工作了，新的公司在编写C或Java代码时所使用的风格并非如你所愿，你是会说“嗯，我习惯后就会喜欢这种风格了”还是愤然离去呢？*

Crockford：或许人们需要注意这一点——他们使用的是何种风格？我们在靠左还是靠右行驶呢？如果公司压根儿就是错误的也许就不在那儿干了。但是只有像Seuss博士[①]的书中的人物才会在意肚子上有没有颗小星星。最终，你只能接受这种风格并希望这些风格的制订者们能清楚他们的问题。他们可能认识不到，但这不重要。因为更重要的是每个人都能达成共识。

Seibel：*在阅读代码时你会先排版，那你会对代码进行多大程度上的重构呢？*

Crockford：我会重新编排代码以便所有东西都会在使用前声明和创建。有些语言提供了很大的灵活性，让你无需再这么做了。但这种灵活性我不需要。

Seibel：*你的意思是说不需要向前引用？*

Crockford：没错，如果存在向前引用的情况，我会显式说明。我可不想搞乱代码，除非是文学编程，在这种情况下，我会显式地根据表达顺序而非语言期望的顺序分离代码，我喜欢这么做。但话又说回来了，如果不使用文学工具，那还是别这么做。

Seibel：*你曾在之前的一次演讲中引用了《出埃及记》第23章第10节和第11节的内容——六年你要耕种田地，收藏土产，只是第七年你要叫地歇息，不耕不种。并建议每次第7个sprint都应该用来清理代码。那什么时候做比较好呢？*

---

① Theodor Seuss Geisel (1904.3.2～1991.9.24)，美国儿童文学作家和漫画家，笔名Dr.Seuss。——编者注

Crockford：每6个周期——不管周期间是什么都该如此。如果你是每月交付，那么我觉得每隔半年都应该跳过一个周期，专门用来清理代码。

Seibel：也就是说如果没有在第7个周期清理代码，那么将会面临大规模重写的境况。你怎么知道何时要进行这种重写呢？

Crockford：通常来说，团队应该知道什么时候合适，管理层很久之后才会发现。团队会定期遭遇问题，比如制造了过多的bug，代码太长了，速度太慢了，进度延后了。他们知道原因所在。这并非是愚蠢或懒惰造成的，而是代码与目标已经发生了偏离。

管理层很难看到这一点，尤其在管理者并非程序员的情况下更是如此。但即便是程序经理也会遇到这样的问题，因为你已经在这个问题上花费了太多的时间。重新开始意味着我们要追溯过去并重新讨论这个问题。同时，我们将无法继续前行，这是不可接受的事情，绝不，我们就把握好手头的资源继续前行。

谬误之处在于人们觉得只需再次花费相同的时间就能完成，但有鲜活的反例可以驳斥这一点。假设有些人已经取得了一些成就，给他一个白板，想做什么就做什么。通常他们会什么也做不出来，因为他们过于野心勃勃，没有想到限制，而你也不会从中得到任何东西。这就是第二系统问题。你必须非常自律，其实“这不是白板而是重新实现，实现我们知道的东西”。

编程之所以难，部分原因是我们大多数时间内都在做新的东西。如果之前曾经做过，那就可以重用了。我们所做的大部分东西都是之前未曾做过的。做之前没有做过的事情是很难的。虽然很有趣，但也困难重重。尤其是使用传统的方法对不太熟悉的系统进行分类时更是如此，在这种情况下很容易出错。

Seibel：“传统”指的是类么？

Crockford：没错。我感觉在原型世界中这不会导致什么问题，因为你的关注点在实例上。如果发现某个实例出现了问题，直接解决就行了，通常无需重构其他东西。但在传统的系统中就不行了——你总得从抽象关系上着手回到实例上，然后将继承关系抽取出来，要想保证准确无误很困难。最终，一旦明晰了问题所在，你不得不回过头来进行重构。但通常这种重构会对代码造成严重影响，尤其当代码规模变大后更是如此，因此你不再重构了而是在上面不断累加新的东西，希望能够修复问题，而这种问题实际上位于最初的继承结构中，这么做的后果将会使代码变得愈发复杂和糟糕。

Seibel：*但你认为如果每7个周期进行一次，那么重构是可以解决问题的，这样就无需大规模地重写了，对么？*

Crockford：是的。什么时候该抛弃旧的代码并重新开始呢？答案就是，在没有经过重构、重构得很糟糕、某些地方出错了或是代码基已经无法使用的情况下需要这么做。可以据此做出合理的判断：推倒重来要比修修补补更快。

Seibel：*如果想要重写某些代码，但却没有完全理解代码的意图，这会导致什么风险呢？因为任何代码段都会包含一些隐性知识——在你说"喔，我们可以重写这些代码"时可能并没有意识到代码中包含了一些难得的功能点。*

Crockford：这确实是个问题。我们身处困境的一个原因就是Web规范差劲至极。Web规范并不完整，很多地方还被曲解了，而这些曲解竟然还变成了标准的一部分。由于这些历史原因，很多系统变得非常复杂，实际上根本无需如此。我对此深表同情——代码基中充斥了太多未经证实的内容了。

微软的操作系统也面临着同样的问题，过去几年他们发布了太多糟糕的东西，现在只能尽力兼容它们了。这种限制对于下一代操作系统的设计是极具破坏力的，最后他们将寸步难行。

这种规范上的错误实在是太严重了，同样Ajax世界也难逃厄运。我们在Ajax世界中遇到的大多数问题都是由浏览器之间的差异造成的。跨浏览器的问题实在是太棘手了，事实本不该如此，造成这种恶果的原因是差劲的Web规范和千差万别的各种实现。

过去几年情况有些好转，尤其是Ajax库的出现更是在极大程度上缓解了这个问题。大多数库都非常棒——但并非十全十美——这让你能在库的基础上进行编程。我们无需直接处理浏览器问题了，可以在Ajax库提供的虚拟化应用层上面工作，这个层次非常有弹性，可移植性也不错。Yahoo!就有这样一个小组，主要负责解决浏览器导致的各种疑难杂症。这项工作可以极大减轻其他开发者的负担，非常不错。

Seibel：*从另一个角度来说，重写不一定总是可行的。你方才提到了第二系统效应（second-system effect），在以前的演讲中你说这"令人心碎"，那是什么时候呢？*

Crockford：那是在Electric Communities的时候。我们几个人组成了一个团队，这个团队是我迄今看到的最聪明的一个团队。我们手头资金充裕，打算重新实现Chip和Randy做过的东西，我们知道该如何做，但有些太自命不凡了。

Seibel：*基本上是重新实现Habitat。*

Crockford：是的，Habitat已经很流行了，但我们打算重新实现。结果发现很难。实际上我们已经构建好了，但发现实在是太痛苦了，这种事情我再也不想做了。

Seibel：*之前你建议只重新实现自己完全理解的东西，这能完全避免灾难的发生么？*

Crockford：我觉得这肯定会有帮助作用的，那时还没有仔细考虑过这个问题呢。我们并没有使用增量方法，假如使用的话，我会采取两个并行的手段。首先，实现一个安全的分布式平台，这个平台除了用于管理消息和对象的基础设施之外什么都不做。然后重新构建Habitat，使用我们熟悉的现代化语言重新构建。

我们会在第二个阶段将这两者整合起来。将一个搭建在另一个之上，如果没问题就发布。

假如采取了这种增量方法，我相信我们会取得成功。但我们却试图在一个阶段中完成所有这些事情，这太困难了。

Seibel：*是不是因为你了解系统的大部分内容才导致想在一个阶段中完成所有工作？*

Crockford：因为我们都很聪明，经验也很丰富，所以想将所有东西放在一个阶段实现，认为我们不会错的。程序员都是乐观派，我们当然也是，假如不是这样，我们也不会做这件事，这很好地解释了我们为何成为第二系统的牺牲品、为何没有规划项目进度、为何没有很好地实现出来。

Seibel：*编程是不是越来越容易了，门槛逐渐降低？*

Crockford：我对编程的兴趣在于帮助其他人编程、设计特定的语言或编程工具，这样会有越来越多的人能够从事编程工作——这也是Smalltalk的初衷。Smalltalk后来的发展方向出现了变化，但最初的方向确实吸引了我。我们如何设计一门面向儿童的语言，如何为那些并非程序员的人们设计一种语言？

Seibel：*你是不是认为每个人都应该学习编程，至少了解一些？*

Crockford：没错。当今世界快被计算机控制了，为了保护自己或是让自己更加全面，你应该了解这些东西的工作方式。

Seibel：*有些人认为通过编程可以学到一种重要的思考方式，比如阅读和数学就是不同的思考方式，但都非常重要。*

Crockford：我以前也这么想。在开始编程时我就有过这种想法：一切都是那么地井然有序，我看到了之前从未接触过的结构等东西。我在想“喔，这太神奇了。每个人都应该学习学习”，因为我突然之间感觉自己变聪明了。但不久之后，在与其他程序员交流的过程中发现，他们并没有开窍。程序员其实与常人也没什么区别，有时他们也会出现误解。当认识到这一点后我觉得很难过。

Seibel：*你现在还像以前那样热衷于编程么？*

Crockford：当然了。

Seibel：*那编程只是年轻人的专利么？*

Crockford：过去我是这么认为的。几年前我患有睡眠呼吸暂停的症状，但没有意识到。我想可能是太累了，年纪也有些大了吧，结果发现自己的注意力很难集中，甚至都没法编程了，因为我的大脑没法承载太多的东西。很多时候编程都需要先在脑子里想好，然后再写出来，但我却不行。

我丧失了这种能力，想当然地认为是年龄太大的缘故。幸好，病情得到了好转，于是我又开始编程了。现在的我编程水平可能比以前还要好，因为我知道如何不过多地依赖于记忆。现在的我更喜欢将代码文档化，因为我不敢保证下一周还能记得写这些代码的意图。事实上，有时我会检查自己的代码，但会惊讶于自己怎么会这么写代码：我压根就不记得自己曾经这么写过，这些代码有的非常丑陋，有的却非常优雅。我实在不知道怎么会这样。

Seibel：*你曾说过文学编程就像Donald Knuth所说的那样，是个非常棒的想法，那你使用过文学工具么？*

Crockford：没有。我一直在考虑这个问题，并为我所使用的一些语言设计过文学工具，但我现在并没有从事文学编程工作。

Seibel：*这仅仅是个工具链问题么？如果有现成的工具，你还会编写文学程序么？*

Crockford：当然会了。比如说，如果使用文学风格编写JSLint的话，那维护工作就轻松多了。我所说的文学风格是指在设计程序时要特别考虑到阅读问题，这么做会给程序带来巨大的价值。

Seibel：你觉得文学编程工具的主要特征是什么？

Crockford：Knuth的主要贡献在于提出了以各种顺序编写代码的想法。这样，如果我所关注的东西涉及很多地方的代码，那么我会将这些代码整合起来并加以说明，然后工具会在适当的地方解决细节问题。

他还帮助你摆脱了函数大小的困扰。理想情况下，一个函数不应该超出屏幕的范围，这样就能一览无余了。如果达不到这个要求，那可以将一个函数拆分成多个函数，如果函数对程序结构起不到什么作用，那么它也没有存在的价值。

Knuth建议抓住函数的各个方面，它们很可能是紧密相关的——具有一致性，但就是太大了；有时函数确实过于庞大——他建议使用具有描述性的标签来表示每个函数，"这个函数是："，然后列出这些标签。你当然可以使用函数达成目的，但这两者并不一样，如果使用函数，你必须要处理函数之间的通信，等等。这么做会引入更多与问题不相关的结构。

最后，我非常希望看到新的文学编程语言的出现。Knuth非常擅于将这些想法应用到Pascal和C当中，但我打心底想看到有人能真的设计出一种新的使用这种风格的语言。

Seibel：你读过Knuth的文学程序么？

Crockford：当然读过。

Seibel：如何读的？像小说一样？

Crockford：没错，就像小说一样。我像在读他的散文而非程序，喜欢他的展现方式，他写得确实好，偶尔还会搞个笑话出来。我很享受这种阅读体验。

Seibel：你从中得到了什么呢？你读过*TeX:The Program*，并一直读到最后。现在打算向TeX增加特性么，还是仅仅形成了这样一种认识：喔，Knuth是个有才华的家伙？

Crockford：问得好。我确实读过TeX，但并没打算修改它，我读TeX的主要目的只是想看看它是怎么写的。我对其处理换行的方式很感兴趣，因此饶有兴致地读了那部分代码，相比于代码的运作方式，我更关心算法部分，所以就没想过要修改或重用它。如果读TeX的目的是修改程序，我敢肯定阅读方式会发生很大的变化。

Seibel：你经常阅读代码么，文学代码还是其他代码，仅仅因为好玩？

Crockford：是的。其实质量足够好，可以从中获得乐趣的代码并不太多。Knuth写过一些，Fraser和Hanson写过一个C文学编译器，非常棒，但这样的例子并不太多，真让人感到羞愧。这或许表明文学编程彻底失败了，因为相关的示例太少。

Seibel：*Knuth的大部头《计算机程序设计艺术》如何？你是从头到尾读过这本书呢，还是将其作为参考随时翻阅，抑或是把它束之高阁碰也不碰呢？*

Crockford：除了你说的最后一种情形之外。在上大学时，有那么几个月我连房租都没交，就是为了买他的书。我读过这些书，从中得到了不少乐趣，比如在第一卷的索引有个关于拖车的笑话就很好玩。我到现在为止还没能把书上的内容全部搞懂。Knuth对某些地方的研究要比我深入得多，但我还是喜欢这些书并把它们当作参考资料。

Seibel：*你是从头到尾逐字阅读，跳过那些不理解的数学部分？*

Crockford：是的，我会很快略读过星号太多的部分。我试图将熟悉Knuth的书作为招聘标准，但结果却大失所望，根本没几个人读过他的书。依我看来，任何自称为专业程序员的人都应该读过Knuth的书，至少也应该买过他的书。

Seibel：*我觉得要想理解Knuth的书，必须得阅读书中的数学部分并理解它们。你觉得程序员需要具备何种程度的数学知识呢？*

Crockford：显然是没必要，因为很多人都没有那么深的数学知识。对于我所从事的应用开发来说，我们并没有使用Knuth的特定工具应用进行开发。如果编写操作系统或是运行时，那数学就非常重要了。但我们所做的是诸如表单验证和UI之类的工作，通常性能并非那么重要。我们大部分时间都花在了等待用户或是等待网络这种事情上了。

我很想坚持说，对大家而言，了解这些知识是非常必要的，但事实却并非如此，或许这就是为何Web编程的门槛很低，JavaScript能够运作的原因所在。这部分内容确实不难，导致它变得困难的大多数东西都是不必要的。如果我们能够清理一下平台，事情就变得简单多了。

Seibel：*也就是说Knuth讲述的是如何实现最根本的东西，然后才有全景。即使清理了平台，但使用理性的方式构建大型系统和设计也是非常困难的。请问你是如何设计代码的？*

Crockford：编写程序与对程序的生命周期进行迭代是不同的。通常，编写软件的原因在于我们知道将要修改它，而修改任何东西都不那么容易，因为很多时候修改意味着打破旧有的东西。

你不能期望使用这种方式完成所有事情，但还是应该尽力保证足够的灵活性，这样不管做什么都能适应。这就是我的观点。如何避免误入死胡同？如何保证灵活性？

这就是我喜欢JavaScript的原因之一。我发现JavaScript可以轻松实现重构，而重构一个继承层次很深的类实在太痛苦了。

比如说，自从我2000、2001年开始编写JSLint以来，它已经发生了很大的变化。它的目标也变化明显——现在它所能完成的已经远远超出了我当时的预期，这其中的主要原因在于JavaScript有足够的灵活性。我能够将JavaScript把玩于股掌之上，随着规模的增长程序再也不会变得一团糟了。

Seibel：*为什么会变得这么轻松呢？*

Crockford：我已经成为软对象的忠实粉丝了。在JavaScript中，你说什么是对象，什么就是对象。这对于那些以传统视角看待问题的人来说是不可思议的，因为没有类的存在该如何做呢？事实证明你已经拥有了所需的东西，这真的很棒。改变对象，直到它变得更加直接为止。

Seibel：*是不是可以这么认为，对于那些基于类的语言来说，问题在于它们过于静态了——你会得到一个很大的类层次结构，如果想要修改结构，只能先分解，修改完毕后再组合到一起。在JavaScript中，危险在于它过于动态了——东西都散落在各处，实际的程序结构是由运行期的很多因素共同决定的，没有什么静态的东西存在，你无法说："好吧，这就是程序，这就是程序的结构。"*

Crockford：这部分让人觉得有些提心吊胆，但小心点总归是好事，因为这就是真实的情况，我们需要一定的原则。在大多数传统语言当中，原则是由语言规定的，但在JavaScript中，你必须有自己的原则。

为了防止代码崩溃，我严格控制代码的组织方式，因为我清楚语言本身并没有强制要求什么。现在，如果没有JSLint的辅助，我是不会考虑使用像JSLint那么复杂的代码的。JavaScript本身的可伸缩性并不好，但借助于JSLint的帮助，我可以信心满满地说我能够做好一切工作。

Seibel：*这么看来，JavaScript对象的软性有时会造成危险，但如果没有充分*

利用语言本身的能力来增强对象，那么造成的后果就是依然使用编写Java类的那种方式来编写JavaScript代码。有没有什么好办法能够更好地组织JavaScript程序以充分利用语言本身的灵活性呢？

Crockford：我是经过了多年的尝试和犯错之后才找到解决办法的。在刚开始从事JavaScript开发工作之际，我并没有深入了解这门语言，就这么匆匆忙忙地开始了。我找到了一个写得非常差劲的示例程序，然后按照自己的想法修改了这个程序。就这样我开始了JavaScript之路，并没充分了解这门语言，不清楚它的工作方式，也不知道该如何思考。

我很清楚为什么这门语言会对很多人造成挫折。如果按照Java那种方式编写JavaScript代码，结果会很痛苦。我一开始就想搞清楚如何使用JavaScript编写出类似于Java类的那种代码，但在一些边缘地带这么做根本行不通。最后，我还是硬往上冲，结果受伤的还是我。

最后，我发现根本就不需要这些类，搞清楚这一点之后，我对语言的使用才走上正途。我不再与之对抗，这时语言也开始眷顾我，给予了我巨大的力量。

Selbel：*在设计软件时，你喜欢自顶向下、从下往上还是先中间后两边的方式？*

Crockford：都有。这些手段可以让你时刻把握系统的设计。最后需要分而治之，将系统分解为可管理的模块。我发现自己会遇到方方面面的问题，因此会同时使用这些技术。我会一直坚持下去，直到真正搞清楚系统的结构为止。一旦弄清楚了结构，其他东西都是自然而然的事情了。

Selbel：*设计与编码有什么关系呢？你会迅速开始编码并不断重构么，或者做一些与编码不相关的事情么？*

Crockford：过去这两者是独立的，但现在变得越来越相近了。我曾使用过设计语言或元语言——有点像英语、具有一定结构、更能清楚地描述你所要编写的代码的一种语言。但如果使用JavaScript，这种语言也就变成了JavaScript。

Seibel：*编码时你使用什么工具呢？*

Crockford：我会用到一些免费的文本编辑器，它们并不会完成任何棘手的工作，但这正是我所需要的。在JavaScript中，你并不会用到其他语言所需的那些正式工具。浏览器只需要一个源文件而已，这样你将这个源文件发给

它就好了，编译器是内建于浏览器当中的，因此你无需再做什么处理，无需链接器、无需编译器，什么都不需要。所有内容全部运行在浏览器中。

Seibel：*那你使用JSLint，对吧。*

Crockford：我确实使用JSLint，我经常使用它。每次在程序运行前都会使用JSLint，如果检查并修改了代码，我会在运行前首先使用JSLint运行一次。

Seibel：*也就是说在文本编辑器中写代码，使用JSLint运行程序，最后在浏览器中运行，那如何调试呢？*

Crockford：这取决于浏览器。如果是Firefox，我会使用Firebug；如果是IE，我会使用Visual Studio调试器。这两个工具都不错。这些浏览器都带有非常棒的调试器。

我曾经使用过一些框架，这些框架带有检测器，它们构建于DOM之上并能跟进到对象内部，展示对象的内部信息并检测其中的内容。但我不需要这些东西，仅仅一个调试器足矣。

Seibel：*当无法跟踪到某个bug时，你是否使用过单步调试？*

Crockford：我只在遇到某些确实难缠的问题时才这么做。单步调试是测试的一部分，但通常在遇到问题时才使用单步调试。

Seibel：*如何看待其他的调试技术呢，比如断言、证明等，你用过这些技术么，你会从不变量角度考虑吗？*

Crockford：我很喜欢这些技术，但感觉有些失望：Eiffel在面向对象语言竞赛中落败了，而C++则成为最终的获胜者。我认为Eiffel是个非常有趣的语言，我喜欢它的前置条件/后置条件契约。不管使用何种语言，我都希望能具备这个特性，但这只是个想法而已，尚未实现。

Seibel：*你曾经跟踪过最糟糕的bug是什么？*

Crockford：这并非是个实时的bug，它来自于一个视频游戏。我们在各个地方都会碰到干扰的爆音，同时也没有任何的内存管理手段，程序突然就会终止，原因也无从查起，这种问题最让人头疼了。通常也没有什么调试器能够解决这样的问题。

我们在Basic Four时开发了一个字处理终端，它基于Z80，能够实现整版显示，内存大小是64K，这样的内存对于显示来说并不太够。此外，它还可以通过本地的网络连接到服务器上并向其发送页数信息。

我们有时会遇到屏幕突然变成空白的问题。系统的架构是这样的：每行文本结尾处有个停止代码，然后紧跟下一行的地址，一个小型的DMA处理器会根据这些链接来查找文本行。有时链接突然就不见了——这表明系统中发生了竞争。

依我来看，从逻辑上说，所有链接都没问题，但我们并没有考虑到与DMA处理器的实时交互，这样可能就不会像我们一样同时查看内存了。我刚把问题琢磨出来。我记得那天我是在家工作，当时正跟团队通电话，突然眼前一亮，我一下找到了问题所在，赶紧告诉其他人，后来我们再也没有遇到过这类问题。

在我的印象中，最糟糕的bug就是实时bug了，对于这种情况，只能使用多线程模拟了。我的解决之道是避免这么做，我讨厌线程，线程是一种穷凶极恶的编程模型。偶尔需要使用到线程，但对于大多数情况来说根本没必要。

我喜欢浏览器模型的一点就是它只提供给我们一个线程。有些人对此怨声载道——如果锁住了线程，那浏览器也被锁住了，因此不能这么干。现在很多人都在呼吁把线程加到JavaScript中，我们一直在抵制。我很高兴我们抵制住了。

浏览器所提供的基于事件的模型确实很不错。唯一会导致崩溃的地方就是当某些进程需要花费很长时间的时候。我很欣赏Google在Gears中所采取的解决方法，他们使用了一个完全独立的进程，你可以把程序发给这个进程去执行。一旦执行完毕，它会把结果传回来，而结果的传回是通过事件实现的，这真是个漂亮的模型。

Seibel：你对形式化证明感兴趣吗？

Crockford：我曾在上世纪70年代研究过这方面内容，想看看是否能从中得到一些收获，结果却一无所获。软件这么复杂，错误之路千万条啊。

基本上，软件就是一套规范，指定了软件的工作方式。没有完整规范的指导，任何东西都无法准确表明软件的最终行为。因此，软件开发实际上是非常非常困难的。

Seibel：如何测试代码呢？你是像他们说的测试狂人吗？

Crockford：我正在变得更有针对性，这又是一个我要改变风格的地方，但尚未完全达成所愿。

Seibel：不是有JsUnit吗？

Crockford：没错。UI代码的测试非常困难，因为它依赖于很多东西，这样把它分解为各个单元往往会降低效率。此外，我发现由于自己编写JavaScript代码的风格所致，我并不会把代码分解成像类那样的各个单元，可以考虑独立测试类。

在JavaScript中，测试独立的函数并没有什么意义，因为函数需要某些状态才能有意义。我还没找到测试JavaScript单元的行之有效的方法。

Seibel：*对于拥有独立QA组的公司来说，开发组和QA组的人员如何才能通力合作呢？*

Crockford：我之前呆过的公司出现过开发组和测试组之间搞对抗的情况，这种现象实在是太不好了。有这样一个理论：将两个组独立开来，他们只会互相背弃。我觉得这种形式太可怕了。

如果将两个组融合在一起，让测试人员负责帮助开发人员提升程序质量而不是背道而驰，那结果就好很多了。这改变了他们报告的方式，也更具效率。此外，将开发人员也拉到测试当中，这样他们就不会彼此排斥了。

我认为最具效率的手段是把一些测试带给最终的用户：访问客户去。我在职业生涯的早期这么做过，非常棒，我和客户一起呆了一周，帮助他们安装新系统，解决新系统使用上的各种问题。

这么做使我深刻地认识到客户实际上会怎样使用系统，我该如何做才能满足客户的需求。话又说回来，没有这种经历的开发者都非常傲慢，这是绝对不能原谅的。缺少对客户的尊重是非常可怕的一件事，这基本上是因为他们没有见过客户所导致的。

Seibel：*你觉得自己是个科学家、工程师、艺术家、工匠还是什么？*

Crockford：我觉得自己是个作家。有时我使用英语写作，有时使用JavaScript。

归根结底，这完全取决于交流方式以及为了促进这种交流所采取的结构。人类语言与计算机语言在很多地方都是大相径庭的，我们需要阅读计算机语言，因此必须与之交流，我根据语言的这种交流能力判断计算机程序的优劣。在这个层次上，人类语言与计算机语言的差别不大。

Seibel：*如果某种计算机语言能与人和谐交流，你是不是就觉得这种人机交流真的会实现呢？*

Crockford：那只是人们的愿望。计算机是武断的，并没有那么聪明，要想

让计算机明白你的想法，必须做出特别的努力。做到这一点很难，因此很容易就忽略了其他方面，但我认为它也是非常重要的。

Seibel：*Dijkstra有一篇著名的论文“On the cruelty of really teaching computing science”（关于真正讲授计算科学的严酷性），说的是计算机编程是应用数学的一个分支，你怎么看？*

Crockford：数学对于编程来说很重要，但它只不过是众多重要方面中的一个。我觉得过分强调数学的作用会导致轻视其他更为重要的方面，比如读写能力。

此前曾提到过，我把是否读过Knuth的书作为招聘标准，但最后却无疾而终，因为根本没几个人能做到这一点。此外，我还要求应聘者真正精通读写能力，无论什么语言都算。我希望应聘者具备写作能力，因为我们经常都是通过书面文字进行沟通。我们需要写邮件、文档、计划、规范等。我要求团队中的每个人都具备这种能力，事实证明这的确很难。因此，我可能更希望那些从事编程工作的人来自于英语专业而非数学专业。

Seibel：*我觉得Dijkstra还说了另一句名言：“如果你都不能使用母语写作，我劝你还是放弃吧。”*

Crockford：我对此表示赞同。

Seibel：*编程对于你来说还有另外一层含义：虽然我们的身体是自由的，但却被历史上的诸多意外限制住了。你提出关于构造JavaScript子集和HTML5的众多提案的目的似乎都是力求修复这种历史问题。*

Crockford：没错，但某些提案似乎有空想的味道。我清楚自己希望实现的很多东西都是无法完成的，对这一点我早有认识，但有时还是能够取得成效的。就像XML提出时是作为一种数据交换格式，但我的第一感觉却是“天哪，这种方式也太复杂了吧。我们并不需要这么多东西，需要的仅仅是来回交换的数据。”因此，我提出了另一种方法，结果我赢了。JSON在目前的Ajax应用中是首选的数据传输方式，这个胜利也遍及到了其他众多的应用当中。JSON非常简单，它恢复了我对人类的信仰，我们终究能做出些正确选择。

但你不能让所有人都这么做，让每个人都构建自己的东西，这是不可行的，对谁都没有好处。一个人需要拿出一件东西，其他人会想想到底应该使用哪一个，JSON是另类的历史意外。

Seibel：总的来说，你认为软件产业是个杰出的创新引擎还是一场可怕的混乱呢？

Crockford：我想使用一种委婉的方式表达"可怕的混乱"。根据摩尔定律，虽然软件的发展速度不如硬件，但现在已经好很多了。软件的发展速度要比摩尔定律慢很多，花费了我们20年的时间才使软件开发速度翻了一番，但至少已经看到了改进。大多数改进体现在这样一个事实：我们不必让软件去削足适履，也不必盲目地提升软件速度，因此我们只要把它做好。但我认为我们在这上面花的时间还不够。

Seibel：不管这是不是称得上可怕的混乱，如果我们真的身处这种混乱之中该如何摆脱呢？

Crockford：这也是我苦苦追寻的东西，这其中很大一部分与我们制订标准的方式有关。软件之所以能运行，之所以能发展到现在，主要是因为网络的存在，可以通过可靠的整合获得各种优势。

但你不必死死抓住这个准则去寻找错误的源泉、寻找哪里才能做得更好。真正的困境在于如何才能合理修复这些问题。标准的修订是一种暴力行为，极具破坏性，它会导致软件无法正常运行、会导致无谓的代价，对人也有很大的伤害作用。因此，在修订标准时要特别小心，因为代价太大了。我们要保证为标准增加足够的价值以抵消这种代价。从目前的标准修订方式来看，并没有做到这一点。之所以要修订标准，仅仅是因为"我们想要修订"、"因为它应该整洁一点"，或是其他一些并不会创造很多价值的动机所造成的。我还在为之而努力，希望能真正做到这一点。

Seibel：看起来你倾向于少做规范。显然，这种方式可以避免事先制订好并标准化规范后，但后面可能又发生反悔的情况。但如果标准中制订的内容太少，人们就不得不做更多的事情，这样在解决问题的过程中就会产生很多事实上的标准。简化标准会解决这个问题么，如果其他地方出现了复杂性该如何应对呢？

Crockford：我们真正需要做的是能够比较好地预测出未来到底需要什么。在最终达成这一点之前或许我们得等待一段时间。同时，对所有可能方法进行的实验和分析都会产生积极的作用；因为标准化的正确方式就是找出哪个方法经过了深思熟虑、最易于维护、成长性最好，符合这些条件的就是我们所需要的。绝不能指望那些标准化委员会，他们仅仅是猜测哪个方法最好，

我们要从市场上找出最为行之有效的方法。

Seibel：*你觉得从整体上来看，我们还是取得了一些进步？*

Crockford：并非如此。有时我们在进步，但有时却是退步。迈入PC时代后我们失去了很多。在分时系统时代，我们有在线的社交系统，这时的分时系统可以看作是个市场，里面有社区，系统中的所有人都可以交换邮件、文件、聊天、玩游戏。他们乐此不疲，但所有这一切在PC时代全部消失殆尽了，我们又用了20多年才挽回这个局面。

我们在安全问题上也退步了。分时系统知道如何保护系统自身，如何保护系统中的每个用户。迈入PC时代后，你拥有了自己的机器，不管做什么，运行在机器上的所有东西都有同样的权限，事实表明，并非运行在机器上的所有软件都符合你的预期。到现在为止，我们还在为之而努力。我们看到PC操作系统已经进行了大量的改进，但遗憾的是，分时系统所具备的那些特性尚未归来。

Seibel：*你指的是什么呢？*

Crockford：MULTICS对多人协作的处理非常不错，它有多个地址空间，可以让用户彼此通信但却不能进入到对方的空间中。这是协作计算最基本的底线。我们现在正在考虑将其引入到浏览器中。MULTICS由来已久，我们现在正开始重拾它背后的思想。

Seibel：*我发现语言也与此类似——PC是用汇编语言编写的，因为即便是C也显得太高层，直到现在我们才开始重新回到Smalltalk和Lisp这类语言上，而他们在PC刚出来时就已经存在了。是程序员没能如愿了解这个领域中相对较短的这段历史，还是我们在不停地重新发明轮子？*

Crockford：太悲剧了，很多人并不知道这段历史，有时我会很失望，因为很多程序员并没有求知欲，他们并不想了解所用语言和工具的出处，想当然地认为是一些社区搞出来并提供给他们用的，他们自己只要能正确使用这些语言和工具就够了。

其实有很多饶有趣味的故事谈到了它们来自于何处、谁受谁的影响、谁做的这件事、什么地方是错误的、什么地方本来是错误的但却未被看作是错误。我觉得自己有时像是个软件技术的考古学家，过去几年我了解到很多未受到正确评价的技术，其中有些技术我觉得非常非常棒，甚至超越了我们目前的状态。我一直希望有朝一日能重新探索这些技术，定会受益无穷，但进

展还是不太乐观。很多人还是满足于能用就行，这种观念已经根深蒂固了，很难改变。

Seibel：*你指的是哪些技术呢？*

Crockford：就是你方才提到的Lisp和Smalltalk。它们真的非常棒，我最终看到这些语言背后的想法融入到了现代语言中，我们现在主要从事JavaScript方面的工作，希望能够实现更加现代化的JavaScript。事实上，JavaScript已经从中汲取了很多养料，它提供了常量作用范围和一等的函数，这好极了。现在我们在考虑如何将Smalltalk和Scheme中更多的精华部分融入到JavaScript中，同时还不破坏语言本身。你可能会说我们不如直接抛弃现在正在做的东西，回归Smalltalk和Scheme，说不定比现在状态还好，但现在还不能这么干。

现在混搭风越来越流行，我们希望能从各个地方找到需要的代码，这些经过测试和未经测试的代码实际上能够实现混搭这个目标。这是一种全新的编程方式，之前我们从未接触过。我觉得这是编程的未来，我们首先在JavaScript中进行了试验，结果没问题，虽然JavaScript有很多糟粕，但这一次语言本身促成了此事。

看看编程技术的历史吧，首先使用机器码，然后是符号化的汇编语言，接下来是高级语言，后面是结构化编程，现在又来到了面向对象编程的时代。每次前进都需要一代人的时间。

我们是要算下一个时代的——已经在对象时代呆了一阵子。你可能会说是Smalltalk-80吧，其实还可以再往前说一点，可是这些想法由来已久了。

虽然具体的名字无从得知，但我觉得下一个时代是关于混搭的，那时我们可以随时随地将程序组装在一起并得到新的程序。数十年来我们一直在讨论能否出现这样一种编程模型，可以像乐高积木那样将不同程序混合起来并得到新的程序，遗憾的是，时至今日这种编程模型也尚未问世。但我觉得现在是时候开始了，就在JavaScript中，在这个最不可能出现的地方发生。

Seibel：*在招聘程序员时，如何识别优秀人才？*

Crockford：我采取的手段就是代码阅读。我会让应聘者带来他们写过的优秀代码并带领我们阅读这些代码。

Seibel：*你想考察什么呢？*

Crockford：我想考察这些应聘者的展示能力，我想看看他们引以为豪的东

西，我想知道这些代码是不是他们自己写的。我觉得这么做要比出智力题和问那些细节问题好得多，那些完全没有意义，但应聘者的沟通能力也是我的招聘条件。

Seibel：*你对自学的程序员有什么建议呢？*

Crockford：两个字：多读。现在有不少好书，去找些好书来看吧。如果从事Web开发工作，请找一些优秀的站点，看看他们的代码。话虽如此，其实我不太想这么建议。大多数Web开发者都是从“查看源代码”开始走上Web开发之路的，但直到现在，大多数源代码的质量都是非常低劣的。因此有一代程序员都被那些低劣的示例误导了，他们写的代码质量也不高。现在的情况已经有所改观，但依然有很多低质量的代码游离于你我之间，所以我是不太想给出这个建议的。

Seibel：*对于那些获得了计算机科学学位并有志成为程序员的朋友有何建议呢？*

Crockford：我还是关注于交流。好好学习写作和阅读吧。

我对所有人的建议都是一样的：多读，多写。通常在招聘时，我不会特别看重具体的技能。直到最近，我也招不到优秀的JavaScript程序员，因为太少了。现在情况已经好多了，但这也只是最近才有所好转。所以在这之前，我在招聘的时候只关注素质。你是个优秀的Java程序员么？是个优秀的C程序员么？我不在意你到底是什么。我只想知道你是否理解算法、数据结构，是否知道该如何撰写文档。如果满足这些要求，那你一定能够搞定JavaScript。

Seibel：*这么做遇到过问题么？有时，让熟练掌握某种语言的人放弃习惯的方式是很困难的，即便在新的语言中他们这些习惯完全派不上用场也是如此。*

Crockford：遇到过这样的Windows程序员。Windows有大量复杂的API，甚至要花费数年时间才能明白这些API的工作方式。你所要做的就是熟悉这些API。你可以编写一个Windows句柄，但其他的东西就无能为力了。我并不喜欢这种过于专业化的开发人员，除非有特定的职位要求我这样。一般来说，我更愿意选择多面手。我想找的是有能力学习任何API的开发人员而不一定非要精通其中某一种。

Seibel：*之前你曾说过之所以踏入计算机领域是因为他们会让世界变得更美好。*

Crockford：这是我的希望。

Seibel：*结果如何呢？*

Crockford：大多数时候都很好。我觉得虽然这个世界并非总是在前进，但确实是在进步着。看看过去十年的国际政局吧，开放的网络并没有对大媒体集团的合并以及由此产生的贪污腐化起到压制作用，这确实太令人遗憾了。

这直接的后果就是导致无数人失去了自己宝贵的生命，太让人悲愤了。我希望网络能够变得更好一些，这种事情别再发生了。现在还不清楚为了达到这个目标需要对网络进行何种修订。或许网络本身并没有错，但我对此还是持悲观态度。我觉得我们需要了解下一个时代，这样才能解决现在无法解决的问题。

Seibel：*不是有很多博主说："嘿，我们的博文会披露所有内幕，而主流媒体却在那儿蝇营狗苟。"*

Crockford：这太棒了，但我们还是用错了地方。我们已经可以通过网络联在一起共享信息，但有时却不行，外界干扰太多了。

Seibel：*你觉得这个问题的解决方案是技术性的么？程序员或系统设计者对架构做何调整才能满足要求呢？或者说这干脆就是个社会问题？*

Crockford：新的社交系统是在这种新的网络基础设施上不断演变而来的，现在还不成熟，因此无法正常使用。或许它解决了自身的问题，我倒是希望这样，但我觉得还有很多地方可以改进。当今的网络在身份识别、安全等方面实在是太弱了，我认为这些东西是构建健壮的社交系统所必不可少的组成部分。Web在这方面还是有缺陷的，或许这就能说明现在出现的各种问题吧。

（编辑：李莉萍）

4

Brendan Eich

吴珂 译

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。

Eich拥有坚实的理论基础和较强的工程实践能力，早期在Silicon Graphics和MicroUnity公司从事网络和系统内核开发。离开MicroUnity之后，Eich去了网景（Netscape）。在开发网景浏览器的巨大时间压力下，他创造了JavaScript。

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。

另外，我们还谈到了JavaScript和Java看起来有几分相似的原因；为什么ECMAScript 4的失败并不会阻碍JavaScript成长为一门真正的语言；以及JavaScript对类似于静态代码分析方法多样化的需求。

Seibel：*你是从什么时候开始学着写程序的？*

Eich：70年代末80年代初的时候，我在圣克拉拉（Santa Clara）大学念物理本科。那时我和同学常常去斯坦福大学研究LOTS-A和LOTS-B，这是两个庞大的DEC TOPS-20分时系统。圣克拉拉当时也有一台TOPS-20，这机器使用DEC的36位处理器，性能相当好，还有不错的操作系统和非常强大的宏指令汇编语言。虽然C语言现在算得上“可移植汇编语言”，但是它的宏指令功能仍然非常弱，然而那时却有许多宏指令要做汇编处理，如果你训练有素就能写出相当结构化的程序。当时没有类型系统，不过C语言至今在这方面也乏善可陈。系统调用、服务、内存映射的I/O等等一些机制最早也都没有出现Unix中。

Eich：我之前是学物理的，但却越来越喜欢写程序，喜欢上数学和计算机方面的课，喜欢研究和讨论自动机理论和形式语言。有一段时间大家在比赛研究如何做出最好的自底向上的分析程序生成器（parser generator），大家都想超过yacc[①]实现的功能。当时编译器构造的前端部分，很容易看到形式语言可以被完美地解释为源代码，后端的情况却很糟，大家都还在探索和学习。但是我还是挺喜欢形式语言和正规语言理论的。

Seibel：*当时你是在什么语言环境下写程序的呢？——如果是学物理的话，我猜是Fortran吧？*

Eich：这个说起来有趣。我当时学的是理论物理，所以没有选工程类的课，免得让我拿着一堆卡片失手洒了再去使用卡片排序机。实际上，我并没有使用过Fortran，而Pascal才是用得比较多的，而且我们也开始接触C和汇编语言。那时我做过一些底层的程序，像用汇编写的散列表之类的。这些活还挺不错，可以让我学到如何更好地平衡几个不同的需求。实际上人们很容易辨别那些从0和1开始写程序的程序员和那些从开始到最后都被高级语言保护起来的程序员。

同时我还对C和Unix非常感兴趣，然而我们只有那台DEC的老机器能够

---

① Stephen C. Johnson和AT&T共同开发的Unix下的分析程序生成器。

用来试试手。那时候还有Portable C编译器，不过是基于yacc的，不太好用，只能够用来生成一些代码或移植一些Unix功能。由于物理并不能帮我找到一份暑期工，我又老在鼓捣代码，做实验室助理，于是我在大四的时候转专业到了数学和计算机科学，并且拿到了这个专业的本科学位。

Seibel：你记得你写的第一个有意思的程序吗？

Eich：这个就不太好意思说了……当时DEC推出了一个非常不好用的图形终端，它可能是VT100的改进版，能够进行转义序列的操作；但是它的色彩深度一般，分辨率也是80年代的水平。我就在这种机器上模拟开发了一些游戏，像《吃豆人》（Pac-Man），《大金刚》（Donkey Kong）什么的；游戏主要是用Pascal写的，而且都会发出转义序列。写这些程序还只是业余爱好，但是程序却越写越复杂。那可能是我第一次印象比较深的编程经历，当时我都考虑到了模块化和自我保护机制。

做这些事情的时候我好像还是在物理专业，上大三。大四我转到数学和计算机科学专业后才开始学形式语言，并写一些分析程序生成器。也就是说，我最早主要编写的是游戏和生成器。后来也开始想想编译器的问题并翻写了一些宏指令处理器，例如m4和CPP。我还记得那时我们搞到几个版本的Unix源码，也开始读一些出奇复杂的C代码，像John Reiser的C预处理器——可能是最原始的版本——简直就是一团乱麻，但它的效率还是蛮高的，使用了全局缓冲器；随处可见的指针让人理不清头绪，应该是想防止别人拷贝才这样做的。但我总觉得还应该有更好的方法去实现这些东西。

这可能是我最终放弃学物理，开始学习计算机和编程的原因，在这之前我其实没有真正在编程，我只不过是解决数学和科研问题。我当时想买一台Apple II但是我爸妈没同意，我试了一把，不是求他们，而是说："我可以用它来学习外语。"这当然是个烟幕弹，我爸妈一语道破："你可能会把时间都浪费在玩游戏上。"他们说的没错，是他们挽救了我。

Seibel：除了可能帮你找份暑期工，编程还有什么地方吸引你呢？

Eich：理论和实践的结合吧，特别是在编译器构造过程的前端，这种感受更明显。数值方法之类的东西就没有那么有意思了，我对这一块不是很感兴趣，因为数值计算说一千道一万，不过是在权衡如何用有限精度的浮点数来表示实数，对我来说这件事就像噩梦一般……这一部分在JavaScript中也是个弊端，因为我们采用的是80年代的硬件标准，因此难免会有一些局限性。

Seibel：*就像对西班牙宗教裁判所一样，没人真的喜欢浮点数。*

Eich：没人能料到会出现怎样的舍入错误——5次方的运算就开始有问题了，在二进制下的舍入问题是非常糟糕的。计算美元美分以及和或差的时候，在JavaScript中很可能返回一串很长的0后面跟着一个9的结果。我记得有篇博客上抱怨过Mac和Safari不会算数——但是实际上，问题在于IEEE的double类型标准，Java和C中也存在同样的问题。

物理学给我的另一种不好的感觉是它有点迷失了方向，人们都觉得自己掌握了伟大的理论，于是挖空心思去发明什么暗物质之类的东西，而这些东西又无法证伪，这让我觉得很不舒服。我喜欢那些有实际价值同时又以数学和逻辑作为理论基础的东西。

我后来还是去伊利诺伊大学香槟分校念了个硕士。本来是觉得可以顺利念完的，可后来被忽悠进了一个IBM的项目并卡在了里面。他们有个很怪异的68020机器，是从康涅狄格州丹伯里（Danbury）的一家公司买来的，IBM把 Xenix移植到了这台机器上。这玩意程序里的"臭虫"太多了，于是IBM将我们的研究项目和他们的合并在一起，结果反倒是把我们变成了质量保证团队。每周一都会有那种穿着西装的人过来向我们发表一通鼓舞士气的讲话，而我们的教授却对此无动于衷。那时，我或许还接触过其他一些人，但我听了Jim Clark（Silicon Graphics创始人）的校园讲座之后，就下定了决心要去Silicon Graphics工作了。

Seibel：*那你在SGI做什么工作？*

Eich：大多是网络或是系统内核编程。在这里我的编程水平见长，最后我们自己写了网络管理和数据包嗅探层，而我开发了域和数据包配对的表达式语言。我开发的翻译程序把配对过程精简到只需用过滤程序对数据包的前36个字节做很少几次遮盖和配对操作。

最后我开始写一个新的语言实现——只要提供协议描述就能生成C代码的编译器。有人希望在我们的数据包嗅探器中加入支持AppleTalk的功能。AppleTalk是一个庞大复杂的协议句法混合物，涉及不同大小的序列、域和各种相关类型的数列之类。做这种开发很有挑战性，也很好玩。后来我用了Aho和Ullman写的那本古老的龙书[①]中的关于编译器的技巧。仅此而已。我想我可能是仿制了一个unifdef命令。Dave Yost也做过类似的东西，不过他做

① 即*Compilers: Principles, Techniques, and Tools*。

的不能处理#if表达式，也不能根据物件是否通过#定义过还是未定义过来最小化表达式，但是我实现了这些功能。这个编译器至今还在，我想它已经被Linux吸收了。

从1985年到1992年我都在SGI工作，直到1992年我认识的一个人去了MicroUnity，而且我也越来越厌倦SGI无度的扩张、收购和日益滋长的官僚作风，于是也跳槽到MicroUnity。George Gilder在《福布斯ASAP》杂志上写到过它，觉得这公司有前途，但是不久就被人遗忘，这家北森尼维尔市（North Sunnyvale）[1]的公司最终亏空两亿美元。但我在那里学到不少东西。我做了一些关于GCC的工作，让我有了不少编译器语言的知识。我也为MPEG2开发了一个小型语言编辑器，这个编辑器可以用来编写复杂的伪规范语言（pseudospec language），如ISO或者IES规范，并且可以成功生成有正确句法的测试数据流。

Seibel：*从MicroUnity离开之后你就去了网景，然后开始创造历史。现在回头想想，关于学习编程，你有什么想做而没有做到的事情吗？*

Eich：在转到数学和计算机科学专业之前我一直在学物理，所以我的数学底子很好，自己开始编程并自学了不少。后来上课的时候我会坐在后面自顾自地看书或者开小差。我那时对自己要求不严，可能错过了不少东西。

我也跟我念博士的朋友们谈过，他们在某些领域上肯定有更深入的研究，让我觉得我其实应该也找个机会读一读这些课程的，可惜我现在也不能回去读书了。当然互联网能让你学到任何知识，但不可能再有那么多时间，有好的导师为你布置作业——其实已经没有机会去认真学了。但是我也没有太过后悔。

就像我提到过，随着编程事业的发展，我开始做一些底层的代码。我并不是一个热衷面向对象和设计模式的人，从来没有买过Gamma的书[2]。网景公司中有些人买，Jamie Zawinski的人或者网景收购的公司来的那些冤家对头会买，他们把这书当成《圣经》一样显摆，真有点让人受不了，他们根本就不是最好的程序员。

我做了很多比设想中要更底层的工作。我在Mozilla和Firefox学到的测试驱动的开发模式是非常有用的，另外一个很有用的就是我们常用的模糊测试。我们有很多源语言和渲染通道，其中有些渲染通道的进化版会带来很多

---

① 加州硅谷中的一个主要城市。

② 指《设计模式》一书（*Design Patterns: Elements of Reusable Object-Oriented Software*）。

内存安全错误，但是模糊测试在这方面就比其他的测试要更有效率。

我也在公司促进了静态分析方面的投入，现在这些投入已经开始带来回报。虽然静态分析看起来比较高深，但我们也请到了技术大牛来用这些技术。

Seibel：什么样的静态分析？

Eich：C++中的那种，要做到却是很难。通常，在静态分析的过程中你要站在全局的高度看整个程序，并且要做一些内存验证的事情。你得对内存中的信息作消除二义处理，找到那些可能引起内存错误的别名，这个工作量会呈指数级增长，因此通常在大型的程序中你都无法完全做到这一点。我们取得的突破就是不必担心内存错误，如果你成功地建立起来一套控制流程图，让所有的虚拟方法都正确地与实现联系上，就可以在不运行完整程序的情况下做部分代码的测试，那些无效代码、冗余测试单元和那些被遗忘的空测试就能很容易地发现。

等达到一个更高的程度，即可以在脑中对自己写的程序有一套完整的验证系统的时候，你往往能写出更好的东西来。但问题是，我们无法在通用语言中用类型系统来实现脑中的验证系统。柯里-霍华德同构（Curry-Howard correspondence）理论说在类型和逻辑之间存在对应关系。其中，类型是实体项，程序则是验证过程，而你应该能够写下这些你试图使用的高层模型。比如说吧，一个数组，在刚刚被实例化的时候需要有长度上的限制，但是在内存中存在一段时间后，这个数组所需要的限制或许会改变，或许根本没有再去限制的必要。你就必须小心翼翼地注意这些不同需求下的不同情况。还有时候在自己抽象的"防火墙"内，你会为了效率改动不变量，但你自己清楚在做什么，而且从外部来看，程序仍然是安全的。像这些情况下就很难完整地用类型检验机制来实现。

用Haskell写程序时，你就必须在知道自己要写什么之前，就决定用什么验证机制；而动态语言之所以流行，就是因为人们可以在很短的时间内建立起一种原型，并对这种潜在的类型系统时时留意，说不准就会碰到有种语言可以支持这种类型系统，抑或是通过静态语言重新实现的时候，就可以写出这些类型。这也是为什么在JavaScript中，我们一直对保留可选类型有浓厚的兴趣，虽然这种类型在委员会中有很大的争议。当然，以后我们很可能在JavaScript的更高版本中加入混合类型系统也说不定。

于是我们希望找到一种注解C++的方式，让传统的静态语言分析可以处理这种注释方式，从而让程序不会陷入效率问题的黑洞，不用等太长时间就

能完成指数级运算。这种注解能保证诸如垃圾回收器的安全性，或是保证让控制和脚本互相作用的函数的分离性，或是重新实质化解释器栈时的安全判断机制。总之就是能给我们带来可靠的安全的东西。有不少特性都是顶层的性质，还不是内存安全。所以我们还要在这条路上继续探索。

Seibel：*这些都是宏观地看编程，那你认为程序员对底层的原理应该了解到什么程度呢？比如有个人主要用JavaScript写应用，他需要非常了解汇编语言这种东西吗？*

Eich：我认识许多非常聪明的JavaScript程序员，其中一些人深知怎么处理效率问题。他们时常做些基准测试，能写出非常紧凑的JavaScript代码，但这并不需要他们了解语言和机器指令的映射关系。

这些程序员在听说我们正在开发的实时可追踪虚拟机的时候都表露出很大兴趣，并且越来越多的人开始用JavaScript进行图形处理。如果JavaScript能有更高的编程性能和更强大的图形处理能力，我认为JavaScript程序员们会使用它来做一些底层应用的。——机器效率和虚拟机效率，这两种说法有区别么？我觉得无所谓，可能虚拟机效率听起来更舒服点。

抽象固然很强大，但我对90年代出现的CORBA、COM、DCOM、面向对象之类的乱七八糟的概念很反感。几乎所有这些东西最初都会调用20万个方法来写个"hello world"。这其实很滑稽，真的程序员其实不喜欢这种东西。在SGI的时候，系统内核是每个程序员都不敢掉以轻心的地方，大家都会尽量不出错。`Malloc`核心那时还是个新玩意，而且它用的还是固定大小的表，导致我们在往表里写数据的时候总会担心出错。

了解底层原理是我的习惯，可以帮我更直接地面对一些问题，避免一些错误。但是，现在随着时间的发展，更快更好的硬件不断涌现，抽象过程也不断地优胜劣汰，我认为人们也不一定非要达到这么底层的程度，不用了解汇编语言，也能写出好程序。

Seibel：*那反过来说，原来那些能用汇编语言写异常复杂程序的程序员，在今天能否做好顶层的开发呢？或者这种两种工作其实需要不同的知识？*

Eich：某种程度上，这两种东西是有一定对应关系的。但是原生指针和年轻的JavaScript之间还是有所区别的。对于这种区别的理解是区分"骨灰级"程序员和其他人的关键。

了解并记住这些不同是很重要的。每个人的记忆力不同，有人可能轻易记住在内存安全构架中的顶层变量并且不用担心指针的问题。不过长此以往

那种无法触及到底层的感觉还是会让我很不舒服。有人就可以做到，反正由编译器来生成代码。但是写编译器的人就需要不断改进他们的作品。

Seibel：因此这种编程方式总会占有一席之地的。但是有没有一类程序员只能在现如今的条件下成功，而不适合只能写底层代码的时代？或者说是否有群人天生就能编程，只不过现在被分成了专业写底层程序和专业写顶层程序的两个群体？

Eich：我许久没有接触系统内核了，所以我想我会同意人们可以触类旁通。现在需要编程的地方太多了，而当下的抽象机制能助你解决那些之前无法解决的问题。

Seibel：接下来回忆一下你最初实现JavaScript的那十天吧。我知道有次你被介绍给了Abelson和Sussman，而你本来的想法是要将Scheme放在浏览器中的。

Eich：网景最直接的想法是要把这种语言做得跟Java很像。有人开发过跟Algol句法很像的Lisp，但是我没有时间去读Scheme的内核，所以我直接就开始写新的了。这样做可能会让我重蹈其他人犯过的错误。

我没有用完整的动态作用域，Stallman则坚持说动态作用域对于Emacs很重要并在Elisp里面大量使用。JavaScript的大部分的词法作用域有点奇怪——有些循环挺动态的，如全局对象、`with`语句、`eval`等。但它又不像Perl中的$变量，或者Tcl中的`my`、`upvar`和`uplevel`。90年代似乎有种这方面的趋势，语言里全是类似的东西。

因为时间匆忙，我没有执着于Scheme。我没有足够的时间思考一些东西会导致怎么样的结果，而是集中精力去限制需要在浏览器中实现的对象。于是我让window对象成为全局对象，结果window对象则引出了未知的新名称绑定问题，这也让对自由变量做静态判断变得不可能。这个缺陷令我感到很遗憾。Doug Crockford和其他一些“对象”发烧友则对从全局对象中得到的未知来源授权感到非常失望。其实这些只是同一个东西的不同说法罢了。实际上JavaScript是有内存安全引用的，我们基本实现了想要实现的目标，但是也发生了一些重大的错误，留下了不少漏洞。

那些处于顶层的变量最后其实变成了对象的可变属性，你可以在不为人知的情况下给它们取别名并弄得一团糟——这种情况真的让人很郁闷。这些东西其实应该是词法绑定的。如果你从这些变量一直向下追寻到函数和嵌套函数，那这种风格就接近于Scheme。你并没有富表单绑定、灵活的`let`，等等，却有不少的集合。但是你为本地变量初始化的绑定则是个词法变量。

Seibel：人们现在就是用建立个顶层函数这种方式来获得命名空间的。

Eich：对。人们建立一个函数然后马上就能调用。这些函数有一个安全的环境去绑定私有变量。Doug是这种做法的拥护者。这种模式对于Lisp和Scheme的程序员并不是完全陌生的，但是很多JavaScript程序员却不得不从头学习这种方式。Doug等人很好地传播了这种概念。尽管这种做法不是想让人成为好的Scheme程序员，然而某种程度上他们还是成功了，现在不少人已经在模式上更多地了解了函数式编程概念，虽说并不深入。

Seibel：JavaScript已经问世10年多了。最近由于Ajax技术的出现，它迎来了一次复兴。人们纷纷说："哦，我们确实应该对JavaScript刮目相看了。"你最近也研究了ECMAScript 4的提案和出发点不同的ECMAScript 3.1方案，似乎它们最终"和谐"共处了？你觉得ES4是一个向大家正名的好机会吗？——"瞧，还是我聪明吧，JavaScript也真的是个好语言呢！"

Eich：没，没这样想过。Doug可能会这样想，而他并不是真的特别了解我。我没有刻意地追求别人的重视，特别没指望来自Java小团体或是落在后面的那帮人的重视。

Seibel：ES4是你一个人的主意吗？它是否体现了你根据你的了解对JavaScript的一切期望？

Eich：ES4毫无疑问地是一个群体的智慧结晶，同时在某些方面也是一种妥协。因为我们在和Adobe合作，他们有种衍生语言叫做ActionScript（AS）。AS的第三个版本对ES4影响很大。其实ES4最开始是由Waldemar Horwat在90年代对JavaScript 2/ECMAScript 4的研究中有了雏形的。然而2003年的时候Netscape退出舞台Mozilla崭露头角，他的那些工作也就停滞了。

Waldemar其实做得还挺好，1997年底我和Jamie去筹建Mozilla的时候把关键的地方都交给他了，而且他也非常聪明——似乎在1987年得过普特南数学竞赛奖，是MIT的博士。他努力确保了语言的动态性，却在试图为语言增加更适合编程的功能——比如命名空间——的时候遇到了问题。

另外一方面，还有种相反的论点更注意小细节。这种人会说"我们应该减少原始数据类型的数量让文档看起来更简单，用lambda函数解决一切问题。所有人都应该这样写程序，因为我就是这样想的。"或是说："这是最好的思路。"实际上这种思路太简约化了，并不适合所有人。或许你的思维习惯的验证系统是减少东西，精简语言。精简固然是很强大的，但如果每个人

都用这种精简模式来写程序，最后是行不通的。

Seibel：*在针对ES4的一些讨论中你引用过Guy Steele论文"培养程序语言"。作为Lisp程序员，我认为他那篇论文的主要观点就是往程序语言中加入宏指令系统，然后那些看起来花哨的东西就会自动消失。*

Eich：其实主要问题是两个。C的句法系统比起符号表示法（s-expression）更让你头疼——你必须定义自己的抽象句法树（AST）并且将其标准化，但是标准化的过程很痛苦。另外一个问题是hygiene的概念还没有被接受。最近看到我们中的Dave Herman正在弄这方面的论文，是关于用逻辑来验证良好的hygiene。我希望他能得到点有用的东西，毕竟我们最终是要采用宏指令的。

几年前受邀在雅虎演讲时，我把这些想法告诉过Doug Crockford。当我谈到一些我十分着迷的好点子的时候，他打断我说："不如我们先做个宏指令系统吧。"我则反驳道："不行，做个那玩意会花掉我们9年时间。"而那时候还有个公司间的纠葛带来的风险：微软并不想合作。优哉游哉了好长一段时间，他们又掺和到ECMA里来了。从印度海得拉巴（Hyderabad）来的一个新人很激动地说："没错，我们会把CLR植入IE8，并会使用JScript.net作为JavaScript的实现语言。"但是我觉得他激动得有点过分了，给出的回应是"不可能"。这样才导致了委员会中的震动和分裂。

于是我们开始担心如果开始做宏指令，就等于开始做研究，而做这种研究性质很强的东西，我们就没法把微软扯进来，从而无法给他们施加压力。因此大家不得不把宏指令先搁置起来。其实我觉得这样做是可以的，只要我们能处理好自动语法检测，保证我们现在做的语言特性在将来能很好地转化为宏指令，但是目前还不用让用户对这些好东西太过于渴望，这样可以让他们保持冷静少犯错误。

Seibel：*1995年的时候，JavaScript的设计受到了那些语言的影响？*

Eich：因为Dave Ungar的一些论文，Self语言在当时很流行。但实际上我没有真正用过Self，只是受到了它的启发。我也喜欢Smalltalk，有人借鉴了Smalltalk中以原型为基础的委托思想——跟JavaScript略有不同的是，Smalltalk是多原型的——尽心尽力地在维护它。这两种语言都能给我动力和灵感，因为它们既是好的编译器，都在虚拟机的层面上下了功夫，语言设计得又非常好。

因为就像Crock等人一样，我想人们还是愿意简化程序语言的。我喜欢

那些尽量减少原始类型而发挥语言效力的语言发明者。在JavaScript中也可能存在着斯德哥尔摩综合征[①]，有人会想："这语言微软不想改进了，也就只能就这样了，我们还需要更好的句法干什么？其实用lambda函数编码不也挺好？"。但抛开这种荒唐想法和微软阻碍互联网发展的事实不谈，语言设计还是能从一两个核心思想的推广中受益的。

Seibel：*那你当时听过NewtonScript吗？*

Eich：是别人告诉我我才知道的。"嘿！原来他们的父链中也有和我们的作用域链类似的东西，还与我们用了同样的原型。"我觉得NewtonScript是Self的趋同进化，还有受HyperTalk和Atkinson发明的HyperCard影响的DOM事件处理程序。我不仅注意到了Self和Scheme，还从HyperTalk中学到以onFoo为名的事件处理函数，后来我把它用到了JavaScript的DOM中，并改名为onClick，类似的情况还有很多。

还有一个受到的正面影响，不过这一个说起来有点不好意思，是关于awk语言的。我是个早期的Unix的拥趸，但是当Perl都显得过时的时候，我还在用awk做一些事情。我本来可以给那些一类函数起个别的名字的，但是由于受awk的影响，我还是用了"function"这个叫法。8个字母是有点多，但也就只能这样了。

Seibel：*至少不是lambda，你要是叫它们lambda，估计JavaScript早就玩完了。有没有哪些程序语言在JavaScript发明的过程中有过负面影响呢？就是那种"不，我可不想那样做"的感觉。*

Eich：由于时间太赶，我没有空去担心"它是否能融进Ada或者Common Lisp"这种问题。但是Java的确带来过不好的影响。首先我得让JavaScript看起来像Java，但同时我又不能把那些糟糕的东西带到JavaScript中，像原始类型和对象的严格区分这种事情我是不干的；另外，我也不喜欢类。所以我回过头去研究Self并采用了原型机制。

Seibel：*那你有没有考虑过让语言更接近Java呢？比如说对Java做一些必要的简化，去掉原始类型和其他无用的复杂部件。*

Eich：一方面是来自管理层的压力想让我把句法做得像Java，一方面我又要保持语言的精悍，毕竟，Java才是人们真正编程时的选择，JavaScript只是它

---

① 又称人质情结、人质综合征，指被害者对犯罪者产生情感，甚至反过来帮助犯罪者。
——编者注

的一个无名的小兄弟而已。

Seibel：*也就说你想让JavaScript像Java，但又不是特别像。*

Eich：对，只有一点像。但是如果我加入了类的设计，那就麻烦大了。这不仅仅是时间很紧的问题，从一开始我就没这想法。

Seibel：*再来说说现在的情况吧。ES4提案已经被正式放弃了，所有人都在为ES“和谐版”而努力，这个“和谐版”会在ES3.1的基础上融入ES4的理念吗？你认为这个决定最终将被证明是明智的吗？*

Eich：Doug当初在博客上用胜利的姿态宣布“我们胜利了，坏人被赶跑了”，而去年有一次在伦敦的演讲上我有一张幻灯片开玩笑地把Doug比作凯萨督姆石桥上的甘道夫，不同的是他面对的不是《魔戒》中的炎魔，而是ES4这个大怪物。他很喜欢那张幻灯片，那是我第一次同他开玩笑，因为他有时候比较严肃，但说到这件事情时他很开心。也许他是甘道夫，但ES4未必是炎魔。

现在想想，ES4有点过于庞大了，但问题是我们需要一个实用的标准，而不是只给你一堆lambda函数，说Alonzo Church证明过lambda是万能的，所以我们就不再提供其他功能了。这种极简的做法简直是把所有人都当专家，而事实是有一大批被Java惯坏的程序员会不知所措。或许有一天JavaScript会衰落，但是我们还是会不断改进它，而只有我们不去为了它的纯洁性而放弃那些好用的功能，JavaScript才能既有理论的优势又很实用。

JavaScript需要进一步的改进以解决程序员们的需求，有时候他们会自己写些抽象库来弥补功能缺陷，但是没有扩展的支持，写抽象库也会很困难——他们甚至连getter和setter函数都写不了，无法让对象本地化，不能将属性变成代码，等等。安全性也不能用隐式或自动的方法解决。

Seibel：*那总体来说你觉得各种程序语言有进步吗？*

Eich：有。我们或许正在步入程序语言的第二个黄金时代，很多程序语言都越来越有意思，同时还有很多新的语言出现。编程就像唱歌和写作一样，需要反复练习。但是跟唱歌一样，没有好的音调系统，练也白搭。语言很重要。所以我们要不断改进程序语言，而不能让它们停滞不前。可能由于互联网需要更多的兼容性，所以JavaScript的发展不得不尽量放慢步调。但是我们不能以此为借口而不发展了，就算改进后的版本根本无法取代现在的这一版，或是太过超前，我们还是应该努力去把它变得更好。

像受Ada和Smalltalk影响很大的Ruby其实就很好，这种折中的语言我也

不反对。虽然它有点被捧过头了，但是本身它并不坏，只是有时候它的粉丝们会捧得它看起来是王者归来能解决一切问题，但实际上不可能。我们需要新的程序语言，但是不能头脑过热。就像先前发生在C++身上的一样，还有那种“设计模式是救世主”的想法，它们不过是对80年代的Unix C保守派发出的反击。

但我们终究需要更好的语言，也许为了好的证明系统，也许是为了在代码里的那些声明能有自动验证功能，因为没什么能一次带给我们所有的好功能。对了，还有像Valgrind这种动态工具及其竞争检测器也不错。如Brooks所说，没有灵丹妙药[1]，但我们却能选择更好的语言。

Seibel：*设计一种语言的时候，需要在防止程序员们出错方面花多少力气？*

Eich：像Java这种“IT民工语言”是不应该有复杂的泛型系统的，因为IT民工根本不会明白协变和逆变类型的限制在语法中到底是什么意思。当然在C和C++里面我也头疼过。编程有一些跟工程学相似，而工程学中要考虑一些很重要的安全属性，设计浏览器时这很重要，设计Therac-25[2]的时候更是如此。 如果没记错的话，最主要的问题在于线程调度，但是即便如此，还是在找更好的语言编写并发程序和有效利用硬件平行性。我们不应该全用同步阻塞，也不应该用互斥量或自旋锁，所以你在设计语言时就得权衡：“为安全起见，我得牺牲点儿表达性。”。

所以在设计JavaScript时我们就很有分寸，没有按照那些“骨灰级”程序员的要求弄一个X86的lambda语言，也没有加入call/cc函数，因为没有必要。除了给实现者增加负担外——就算这不是问题吧——人们还肯定会因为那些东西误入歧途。就算不是所有人，我想大多数人都是想成为顶尖程序员的，在程序员中也有个金字塔，所有人都在往上爬，但是就算在最顶尖位置的人也会犯错。

其实在JavaScript中犯错的机会有很多，那些一类函数啦，原型啦之类的，因为它不具有那种典型的面向对象特点，所以很容易让人困惑。

这些东西基本上都够用了，我也不是那种极简主义者，会说那种“好，只能做这么多了”的话。微软就很喜欢说这种话，让我很愤慨，因为我总看到人们在浪费时间，却还是有很多bug——就算你用lambda，还是会有些很难找的bug。

---

① *No Silver Bullet: Essence and Accidents of Software Engineering*。

② 一种化疗仪器。上世纪80年代曾因软件缺陷导致严重医疗事故。

Doug已经教会很多人不同的设计模式，但是我更同意Peter Norvig的话，他说模式会体现程序语言中的不足之处。模式不是天上掉下来的馅饼，是有代价的，所以我们应该更多地关注语言本身的发展。说不定哪天就有新的类型可以用了，甚至他们可能会变得像PLT协议。

Seibel：*你手头上的事很多啊，C++式的静态分析，实时追踪，还有JavaScript的新功能，看起来你似乎在进行很多计算机科学的尖端研究工作。*

Eich：我们在打一场漂亮的仗，同时也会保持头脑冷静。我们进行一些很前沿的研究是因为关于学术类的研究一直存在一些问题，其实自从我上学开始就有这方面的感觉，而现在这些问题还没有解决。

于是我们想修正这些问题，所以我们的研究人员都会讲究课题的实用性，这很好。我们的资金不是特别充分，所以我们会利用其他的方式，比如说请人讲座或是促进网上协作等等，来推动研究。

如果所有研究人员都为了每年的NSF（美国国家科学基金）资助而奔走，问题就大了。另外一方面，大家看到动态语言在兴起。很有些疯狂和无知的言论说动态语言马上要取代Java和其他静态语言，这当然是胡说八道。但是学术界却有人还深信静态语言才是终极语言，这些人还在研究一些特定的静态系统，像ML和Hindley-Milner类型推断之类的，而这些东西完全跟业界脱节。

Seibel：*怎么会发生这样的情况呢？是因为这些东西完全没有用，还是因为它们只解决了部分问题？*

Eich：我们和新泽西SML组织合作过，合作项目是自主支持JavaScript第4版中引用的实现，这项目现在已经停止了。我们当时试图开发一种可定义的解释程序，但是我们甚至没有用Hindley-Milner类型推断算法。为了不在类型不统一的时候抛出匪夷所思、臭名昭著的错误消息，同时随便找一段代码来作“替罪羊”，我们会给类型和参数加上注解。这可能牵扯到具体实现过程中的质量问题，也可能涉及到类型的理论性问题，因为当统一过程出错的时候，很难找到真正的错误源。

所以你可以去做这方面的研究，找出程序员易犯的认识错误的顶层模型，让程序员能更快地找到出错地点。可能这只是一个小细节，但是看上去是个大问题。

学术界往往不会去引导大家得到更好的模型，实际上我觉得学术界有点被大家遗弃的意思，或许这不是他们的错，环境所致而已。我们确实在向着

大规模并行计算这个热门方向前进，却没有人能很好地解决这个领域里存在的问题。虽然大家热衷于事务性存储，但也不能解决所有问题，毕竟几个嵌套事务在数个处理器里跳转、竞争资源并不是个好事。不光是效率问题，其实在某些情况下这种模式根本就不适用。你根本无法将你的并发任务和平行算法跟这些东西映射起来，连试都不该试。

Joe Armstrong等人将无分享构架（shared nothing architecture）弄得有声有色，在不同的浏览器中都能见到它的身影，比如Google Chrome等；我们也在用JavaScript实现这个架构。但是我觉得这个概念在学术界中根本就没有得到过响应，特别是对那些研究计算机体系结构的人来说，事务性存储更有意思一些，因为这个东西能够让他们找到更好的指令和硬件支持。但问题是这个玩意不能解决现阶段存在的问题。

我始终觉得程序语言会有长足的进步，这也是我为什么说第二个黄金时代会来临的原因。只不过在语言用户、开发者和研究程序语言的学者之间还存在一定的隔阂。

Seibel：你拿了一个硕士学位，但是没有读博士。你觉得程序员都需要去读个计算机科学博士呢，还是只有一部分人需要博士学位？

Eich：我觉得只有一部分人需要博士学位。博士学位很费功夫，有时候你甚至会想最终授予你学位只是看在你已经受尽了折磨的份儿上。但是当你的名片上印着博士两个字的时候，你得到的机会就会多些。但是我在硅谷的经验告诉我，在这个20年来一直在通货膨胀的时代（或许行将结束），花那么多精力去读个博士不一定划得来。至少我不后悔没有这个头衔。

有能力去系统地、放松地学习一些东西是很不错的，现在到处都在说抢占市场、驾驭摩尔定律、与快速产品周期搏斗，甚至淘汰掉现有软件，如果所有人都只关注这些东西，那才悲哀。所以那些想读博士学位的人要有一种责任，因为他们有能力、也应该有做研究的兴趣。Mozilla也在努力加强学术研究和产业实践的联系，包括编译器、虚拟机、调试器甚至类似Valgrind的性能分析工具。虽说有些投资不足，对研究人员也不太有吸引力，不够新潮，工程的味道浓厚了一些，但还是有可以突破的空间。我们跟Andreas Gal合作，没想到他的论文因为太实用而被驳回。

当然，我们需要这样专业的研究人员，也需有研究能力的程序员。编程学科不应该只是IT民工知道的事情，不应该被隔绝在象牙塔之外。

Seibel：你对验证机制怎么看的呢？

Eich：很难。大多数人都太懒，但Larry Wall说得好，懒惰是一种美德，这就是为什么我喜欢自动完成功能。验证这种东西就是程序员不喜欢但是学术界非常喜欢研究的东西。写断言可能有点帮助，尽管有写得不好的断言，但在Mozilla我们还是不断会写出好的断言来。由此我们得到一些启迪，应该在理想的类型系统中表示什么样的不变量。

我认为把断言当作验证点是有用的，但是不能强求一个完整的验证，学术论文里有漏洞的验证比比皆是。

Seibel：换个话题，你处理过的最烦人的程序bug是什么？

Eich：这个问题啊……最头疼的bug是关于多线程的。我在SGI的工作是关于Unix内核的，这个内核一开始跟那时候其他的Unix内核一样，是一个巨大的监控程序，当你通过系统调用进入内核后就能运行到底，除非有异常中断，否则就会很顺利地完成，也不会发生数据结构被锁死的事情。这个过程很好，简单明了。

但是后来HP的人带来了点新东西，他们把对称多任务处理业务卖给了SGI，而且这些人真的给老的系统内核团队带来了冲击。他们有些还是新人，就登堂入室，像在棒球场上一样跃跃欲试，不断地挥棒，一下子把球打出了界外。我觉得他们本可以用C啊、信号量啊、自旋锁啊，或者像是控制器啊、条件变量啊，等等，但是他们却用手写全部的代码，于是留下了无数的bug，真是一场噩梦。

我在博客上提到过我去澳洲和新西兰的一次免费旅行，其实是调试那个程序bug去了。想要找到这个bug就很困难，因为我们将一部分单线程内核代码放到对称多处理器多线程内核中，而且那时我们还没部署竞争校验条件。一开始我们就得写一个测试用例来找到bug，这就已经很难了；后来时间压力很大，客户迫切要求趁我们在现场的时候把问题解决掉，我们必须要尽快把它解决了。

之所以诊断出这个bug很困难，原因是时间要求很紧。我们推测问题肯定跟终端通过集线器对机器的过度使用有关，因为有好几个虚拟终端（PTY）接到一个真实终端。客户是一个数据挖掘软件公司，在澳大利亚的布里斯班（Brisbane），实验室的学生和其他人挤在满是70年代显示器的机房里，房间尽头用玻璃隔开了一堆机器，其中就有SGI的双处理器机。最后解决掉那个bug的时候我们都很高兴。

这种bug不会存在很长时间，只是很难找到出错的位置，而且需要你没

日没夜、茶不思饭不想地去分析它到底是怎么回事。最终用的解决方法却很简单，跟解决别的bug大同小异。最后调试用的是二分法，我们俗称“圈狼”，根据执行结果和内存状态，界定bug的范围、控制流和数据的内存地址。如果是指针的问题，你可能需要一些很难用的工具的帮助，像Valgrind和Purify，这些工具直到最近出现了GHz级别的处理器时才变得非常重要。

弄清一个内存的等级结构也很困难。Robert O'Callahan，一个在新西兰的非常聪明的工程师，用Valgrind的框架开发了一个调试工具。这个工具可以记录所有的指令，所以他可以从任何一个地方开始重新写程序。这不仅仅是个基于时间的调试工具，它还是一个可以让你看到数据结构的完整数据库，在发现某个域的值被篡改的时候，你可以知道最后的操作是谁做的，也可以得到整个栈数据。你可以用它从表面现象追溯到原因——其实所谓的调试过程就是这个样子的。很慢，比想象中还慢，但是问题总有被解决的希望。

另一个选择是弄一台速度很快的记录虚拟机，它们只记录系统的调用指令和输入输出边界，也可以在边界上把毁坏的程序状态重写，只不过对边界之间的东西使用这种方法就会遇到困难。但是用这个东西你几乎可以快到实时定位bug，基本定位之后，再用Rob的Chronomancer工具将运行速度降下来，这样就能从所有的状态中找到问题所在。

可惜的是，关于程序调试方面的研究已经停滞不前了。这是业界和学术界脱节的又一个明证：学术界差不多都在做验证机制，或者手工写，或者托POPLmark竞赛的福越来越多地用机器完成。但是实际上我们现在用的已经是GDB之类的调试工具了，而他们从上世纪70年代以来就什么也没有干。

Seibel： *而且在业界也存在两类人，有人用符号化调试工具，有人则会用print语句。*

Eich： 确实。我会用GDB，其实GDB也不错，至少在Mac平台上它有不错的观察点功能，所以我能监视一个内存地址上的字节的变化，了解这些变化是对是错。这功能很好，否则我就要用`printf`来一点点找了。当我接近问题所在的时候，我也会在GDB里面做一些调试，或者用一点命令行脚本，但是后一种方法作用不大，因为脚本语言本身很弱。我知道Van Jacobson带来了循环功能，但不知道在FSF（Free Software Foundation）的监控下，GDB是不是真的把这个功能加进去了。

调试还能为我们做更多的事情，而Chronomancer和Replay这些工具就在做些有益的尝试，它们对我影响挺大的。但是关于多线程调试的发展我也不

太了解，好像有个叫Helgrind[1]的工具，我也在用另外一些动态竞争探测器。这些工具有时候也会作出错误判断，需要剔除掉，我们像是在慢慢地训练这些工具为我们所用，或者通过修复代码来避免触发它们。总之，是非曲直还没有定论。

那些关于多线程的东西，说实话，在我结婚有孩子之前几乎占据了我所有时间，所以我都有点阴影了。并不是所有人都会思考并发性的问题，也不会想很多先后次序的问题，哪怕情景很简单。一旦代码需要与别人的代码整合的时候，局面就失控了。你不可能在头脑里模拟出个状态空间。于是我可能显得跟那些在Slashdot[2]遭人侧目的博客作者一样，在抱怨"多线程很糟糕"的时候被人鄙视，他们说："看啊，这家伙啥也不懂，他不是个真人"。其实这些人才是笨蛋。的确，我去新西兰和澳大利亚出了趟差，拿了些出差补助，但是解决那些bug也确实太花时间，调试起来太痛苦。

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。

---

① 一种线程错误探测器。

② 一个著名的技术站点。

——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。*

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

① 指*Joel on software*。

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：*你有没有试过文学化程序设计，像Knuth那样？*

Eich：我按照书上那些最原始的东西做了，很简单明了，我很喜欢，有单词检索功能。Knuth给出的散列字典数据结构就都是文学化程序设计出的。后来Doug McIlroy用常规编程完成了它。

现在的程序都有大量的注释，但是缺乏从中提取可用信息并转换为可读文字人工或自动检查的方法。Python社区里有人在做一些有意思的东西。我一般都只给出大量注释，有时候还会把代码翻出来，维护其中的注释——那其实是很痛苦的过程，后来我不去维护了，结果又很后悔，因为害得别人做了。

我比较喜欢McIlroy的回答，他并不是反驳文学化编程，但是有点儿像是。编程时人们不喜欢多写字，不管是注释还是代码。其实就小范围的代码来说，它们本身应该让人一看就能理解。大型的函数或者模式的话，文档更好些，或者是文档式的注释。Python有个很好的功能是嵌在注释里的测试，我觉得不错。

文学化编程也有整合测试和文档字符串的含义。我希望看到更多的语言都能支持诸如此类的特性。我们曾试图在ES4中的一类元数据或者反射挂钩

中加入对文档注释的支持，但是没法让所有人都同意这个想法。

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。

Seibel：当有一大段代码要读的时候，你会从哪里着手呢？

Eich：我通常会从上往下读。但是如果程序太大，函数和控制流程会变得不清晰，我就会用调试工具来辅助阅读。我也会根据我了解的框架结构从底层向顶层分析。如果是一个语言处理程序，或者是一个我能弄明白的关于系统

调用的东西，我会从它对原始类型的操作入手。我会问自己，这个东西是怎么被更高层的系统调用的——这样问一下对我了解整个程序帮助很大。但更重要的是程序背后的一整套结构，这需要你从多角度去读这个代码，去用调试工具运行，一步一步地分析，当然，这个过程是很乏味的。

如果你能弄明白底层的一些东西，可以搞清楚那些指针啊，条件语句单元啊等等，这种无聊的过程就会显得有点意义了。对我来说，这些东西跟读代码这件事一样重要。读代码是一个很漫长的过程，你可能会被卡住，觉得无聊，甚至恍惚中觉得自己弄懂了，其实并没有。

我在设计JavaScript的正则表达式的时候参考了Perl 4。当时我用了调试工具去一步步地运行它，同时也去读了它的代码。这些过程给了我一些灵感，最后我实现正则表达式的方法跟Perl就很相似。后来我发现它的递归回溯设计有点不现实，于是我只好自己去弄一套。实际上，调试简单的正则表达式还是很有用的，你能对执行过程一目了然。也有其他的程序员说过一样的话：要深入代码，要了解程序的动态状态，看程序要像看航拍照片一样，要从全局的角度去把握它们。这些说法我完全同意。

Seibel：*你也会这样读你自己的代码吗？甚至不是在调试查错的时候。*

Eich：那是必需的，健全性检查嘛。我写了很多断言，如果其中有一点点错误，我就必须拿起调试工具去战斗了。但是有时候写代码你会有些类似“指导手册”之类的东西在旁边，于是写出来的东西在测试的时候好像是对的，但如果拿到调试工具里一摆弄，问题就来了。而有些问题只有当一些特定的条件被满足的时候才会出现，可能用设置些条件断点啊，观察点啊，数据断点之类的方法可以发现问题，但是当你做检查的时候，却发现所有代码都一个萝卜一个坑地放得好好的，就是测试起来不对。这个时候你应该去调试工具里找原因，即便你的代码看着很完整。我觉得这一点很重要，所以我一直在坚持这种做法。

Seibel：*检验代码的时候，你是不是在脑子里先有个大概的设想，然后发现设想的结果没有出现，于是就确定某个地方有问题了？*

Eich：预想结果没有出现，或者我意识到自己太乐观。虽然随着年龄增大我变得更加小心，也做得更好，但是有时候还是会太乐观了。所以我总记着Jiminy Cricket的话：“你一定是忘记了什么东西，从而会导致bug的出现。”而这种情况确实时有发生。

而有些时候我肯定知道我有什么地方做错了，我总是会有这种第六感

——其实也算不上第六感……但是我不能确定哪里错了，但是能肯定有些很细小的错误。总之，我有时候就有这种感觉。我知道有些东西需要特别留意，然后调试工具则能帮我找出这些问题所在，或者让我发现测试虽然在一定程度上检验了代码，却并未涵盖所有情况的组合，因为出错的空间太大了。有时候你只需要改动一个小小的变量，这错误就会跳出来。

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。

Seibel：也有不少书是教具体怎么编程、怎么写代码的，没有讲多少理论知识。

Eich：这种书我更喜欢。我们刚刚说到过Knuth的文学化编程的著述，其实有很多关于编程实践的领域我都喜欢，像Smalltalk的书。现在想想其实这系列的书是很有影响力的。还有Adele Goldberg的书，以及比它还早的那期*Byte*杂志。

Seibel：就是封面上有热气球的那期？

Eich：对，它对我影响很大，这个杂志很有名，好像是上个世纪80年代的。那个时候我很少写代码，只是会看这个杂志，然后本科时在玩DEC的老机器。Smalltalk环境很纯粹，而且是高度辅助程序化的，给我留下的印象相当深刻，对我最终进入这个行当起了很大作用，让我对程序语言和虚拟机产生了浓厚的兴趣，接触Unix后我又对硬件和操作系统方面的东西有了兴趣，那时才开始有了真正的行动。但是那时还是读书读得多，像Springer出的一本书里汇编了很多论文，那个时候人们很喜欢讨论关于全局对象文件格式和Java的字节码，有些超前。是的，Smalltalk还是很火的，但是我直到在伊利诺伊大学念硕士的时候才使用Smalltalk，当时他们发布了一个在Sun的机器上运行的版本，但是速度很慢。

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。

我在念本科的时候深受Pirsig的那本《摩托车维修的禅与艺术》（*Zen and the Art of Motorcycle*）的影响，我也读过柏拉图和其他早期哲学家的理论。于是，那时的我有点哲学意味上的理想主义。比如我会觉得小字节序要比大字节序更好，因为在小字节序中最低有效位在内存地址的最低位，这样不仅

看起来更顺眼，还有种几何的美感。然而如果试着去读十六进制的内存，就会发现，实用性最重要，细节最重要。在名画《雅典学院》中，亚里斯多德手指向下而柏拉图手指向上，我想现在我也变成手指向下那一派的了。随着年龄增长，经验慢慢丰富，我变得越来越怀疑一切，变得更注重实用的东西。

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。

Seibel：*有没有不如意的地方呢？*

Eich：不知道……C++？C++的很多功能我们都能用到，但问题就是功能太多了。可能C++的类型设计比Java要好一点，但是我们用的还是上世纪70年代的调试工具和链接程序，这很荒诞，搞不懂为什么大家还能忍受。

没耐心和对原始工具的讨厌，这两种态度是驱动我成长的动力。值得注意的是，现在大多数人的代码充斥着晦涩难懂的断言，这样的代码真要命。我们很注意这个问题。但是断言有时候确实会帮大忙，就是上面我说到的代码中遇到不可能面面俱到而必须做出90/10权衡的时候，这个时候断言往往就是那个笨拙但有效的方法。它能提醒你问题在哪儿。

我也发现了自己的一些缺点，比如有时候会过度“优化”。我有时候过

---

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。——编者注

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。——编者注

于乐观，而忘记了其他重要的问题。这里其实有个矛盾，程序员就应该保持乐观，但同时我们又得是偏执狂、神经质，就像伍迪·艾伦在电影中扮演的那些角色一样，总有无尽的担心。但是，如果程序员太过偏执，就会有太多的问题没法解决了。

Seibel：*你是否认为编程应该是年轻人的游戏？*

Eich：我认为年轻人有巨大的优势，脑袋瓜更灵光。他们缺的是经验和智慧。我们这些老人们有时候会变得顽固、跟不上节奏，但是有亲自走过一些弯路并从中得到的宝贵经验，可以传授给年轻人。不过年轻人经常不听，非要自己再去走一遍弯路。对这种情况我只能扼腕叹息。

但如果撇开这一点不谈，如果能博学多识并与时俱进，也不一定非要他们完成太大工作量。写大量代码固然很重要，但我很感兴趣的是——这一点我以前在网景跟他们探讨如何成长为一名首席工程师时也说过——有的人并不在管理层，但也领导能力和影响力，能让手下的程序员们像他们一样地写代码，不是自己亲自去做，因为你时间有限，也没有三头六臂。

有能力把自己的想法和编程感悟在某个小团体中传播，从而产出比自己编的更多的代码，这与自己熬夜写出大量代码同样会让我满意。

现在的我还是会辛苦工作，而且我还有了几个孩子。我妻子虽然心大量宽，但我觉得她还是不喜欢我老出差。但是我有时候还是会到处跑，虽然这些工作不是编程，但是也很重要。比如说JavaScript吧，我们要推动这个语言发展，就不光得去宣传，还得启发大家思考这个语言还需要什么，应该怎样发展，向何处发展。还需要面对不同的声音。

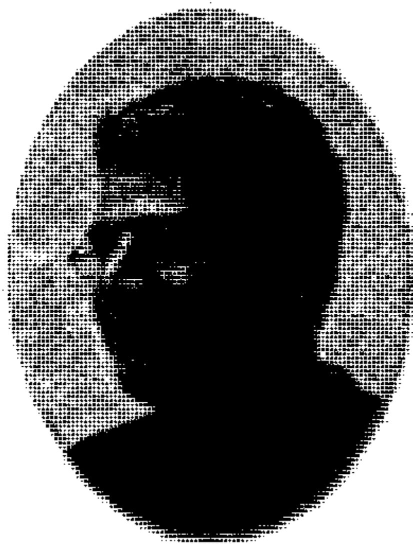
不是所有程序员都会提意见的，很多程序员都有点孤僻，喜欢单干。但是我在网景得到的一个经验是，我喜欢跟使用我设计的程序的人互动，如果我把自己孤立起来，就没有这种体验了。我希望跟人接触。虽然我知道我自己埋头单干也能做出不错的东西，但理智也告诉我，这种情况下做出来的东西只能给我自己用，而不能适合所有人。就像Hillel说的："如果我只考虑自己，我又能走多远？"

我不代表JavaScript。早些时候，JavaScript匆匆忙忙推出，一堆bug。Zawinski给我看了Usenet论坛上的一个帖子："他们说你生了个丑孩子。"现在我真的有小孩了，所以我也不担心了。

（编辑：李松峰）

5

Joshua Bloch



郝培强 译

Joshua Bloch现任Google公司首席Java架构师。之前他在Sun公司工作，曾获杰出工程师称号，领导并实现了Java 2中的Java Collection Framework，还参与了Java 5发行版中几项语言附加特性的设计。Bloch在哥伦比亚大学获得学士学位，在卡内基梅隆大学获得博士学位。读博期间他参与设计了Camelot分布式交易处理系统，这个系统后来演变为Transarc公司的产品Encina，而他则成为Transarc的资深系统设计师。他编写的*Effective Java*一书获得了2001年Jolt大奖，他还与人合著了《Java解惑》[①]和《Java并发编程实践》。

没错，Bloch是Java的忠实拥护者，他的工作就是推动Java在Google公司的广泛使用。最近风行于世的并发解决方案，如软件事务内存或者Erlang的消息传递机制等，Bloch都不太看重，他认为Java是并发运算的“最佳语言”，并预言随着越来越多程序员需要开发应用于多核CPU的程序，Java将会再次

① 该书中英文版均已由人民邮电出版社出版。——编者注

风行起来。

Bloch也很推崇把编程看作API设计，我们探讨了这对他自己的设计流程有何影响，同时还讨论了Java是不是太复杂了，以及为什么说选择编程语言就像选择酒吧。

Seibel：*你是怎么开始编程的？*

Bloch：我想这是受益于我的家庭影响。我父亲是布鲁克墨文国家实验室的化学家。当我上小学四年级的时候，他参加了一个程序设计培训班。当然在那个时候，电脑都是放在玻璃窗背后的大型机，你只能把写好的程序卡片交给操作员。虽然没法儿亲自动手，但我还是被电子计算机可以帮助你做事儿这一点震撼了。所以，我在父亲上课的那段时间，跟他学了一点儿Fortran。

Seibel：*那大概是哪一年？*

Bloch：我想是1971年。直到很多年以后我才真的对程序产生了强烈的兴趣。让我产生兴趣的当然是分时系统。长岛有一台DECsystem-10电脑，供Suffolk县内所有的学校使用。Nassau县也有一台。很神奇的是，很多著名人物的职业生涯都是从这两台DECsystem-10电脑开始的。

你和程序一旦有交互，就被迷住了。大概是从1973年到1976年，那时候我跟其他人一样，在写BASIC程序。我就是从那时开始正式写程序的。你知道吗，我还保存着当年写的程序，是印在电信打印纸上的，这种纸真耐用。如今回头再看这些程序的时候，我发现我代码风格中的某些部分从那个时候起就一直没变过。

Seibel：*你还记得你写的第一个有趣的程序是什么吗？*

Bloch：噢，我记得那是1977年7月4日，我为经典的二十问游戏写了一个程序，叫“猜动物”。这个程序包含一个二叉树，是非题位于它的内部节点，动物位于它的叶节点上。如果用户所提的动物是叶节点上没有的，它会猜测，会向用户提出是非题，通过区别新动物和它猜出的错误动物之间的差异来了解新动物。二叉树保存在硬盘上，这样程序可以越来越“聪明”。

我当时想：“天啊，真酷，程序真的能学习。”这是我一生难忘的瞬间。我还记得另一件事。当时我在高中，应该是10年级吧。是关于DECsystem-10的。当时不允许我们编写现在叫做即时消息软件的东西，因为它们对系统资源的消耗实在太大了。

Seibel：*其实现在也这样。*

Bloch：千万别让我打开这个话匣子。IM毁了我的生活。不，是E-mail毁掉了我的生活，IM仅是个娱乐而已。算了，不谈这个了。当时还是小屁孩儿的我参加了长岛数学展的一个项目。这个项目我把它叫做“工作协同通信程序”。它还为我赢了个奖呢。

Seibel：*那么你真写了这些程序？*

Bloch：是的，我写了。只是其中一个程序是我的朋友Thomas De Bellis写的。Tom程序的独到之处在于，它全部是用BASIC编写的。这是一个面向行的、使用文件通信的程序。虽然不够快，不够高效，但是它可以运行！我写了两个，一个是面向行的，一个是面向字符的。我是用PDP-10电脑的汇编语言MACRO-10编写的。该程序使用了一种叫做“高段”（high segment）的共享内存进行通信。

那时候我对并发编程一无所知，甚至还不太理解互斥量（mutex）。程序有通信缓存，而独立代理试图并发访问对方。所以程序里面有竞争条件，时不时地丢失一两个字符。作为一个高中生，我没能找出原因。

Seibel：*你说你在早期程序中还可以看到现在风格的轮廓。那么哪些部分是你现在仍旧保留的呢？*

Bloch：我一直努力让程序的可读性更好。正如Knuth所说，编程本质上是一种文学艺术。不知道为什么，那时候我就意识到程序的可读性必须更好。这种态度至今未变。

Seibel：*那什么变了呢？*

Bloch：嗯，当你的变量名只能用一个字符的时候，让程序易读是非常困难的。所以，我现在更关心变量的命名。显然，编程语言的新特性改变了很多事情。很多你只是模模糊糊理解的东西多年以后会变得司空见惯。

例如，不要重复自己。那时候我复制粘贴程序时没有任何压力。现在则不同。现在我真的试图不复制任何程序。这么说有点儿夸张，但并不为过。一般来说，如果我发现自己在复制粘贴，我就想：“我的设计有什么问题？该怎么解决呢？”这些东西需要点儿时间才能搞定。基本上，这些年来我对自己越来越严格，但是这有利于写出好程序。你不能纵容自己的坏习惯。

Seibel：*如果时光能够倒流，可以一切从头来过，有什么东西是你真的希望改变的？BASIC对你来说太简单了？其他还有什么？*

Bloch：我没什么遗憾的，实际上BASIC很有趣。我觉得Dijkstra对BASIC的看法是完全错误的。原谅我这么评价已故的人，愿他在天堂安息。我知道很多非常好的程序员，他们是从BASIC编程开始的，因为那是他们当时能找到的唯一的语言。

然而我觉得使用多种语言是件好事。上大学的时候，我用很多语言编程。每门课都可以用一门语言。在数学课或者理科课上，应该用Fortran。那时候的编程课学的都是Pascal、SAIL、Simula之类的东西。在人工智能课上，用LISP。

不过也许我应该学更多的语言。有意思的是，一开始我对面向对象并不感冒，直到很晚，我才真正对面向对象有了感觉。严格来说，Java才是我真正使用的第一种面向对象语言，某种程度上是因为我不太想用C++。

Seibel：*那是什么时候？*

Bloch：那是我1996年加入Sun公司时。我觉得要是我能更早学习这些概念就好了。我不认为这些概念都是好的。面向对象很有意思，它有两层含义。第一，它意味着模块化。模块化是非常好的。但是我不认为这是创造面向对象的人们的专利。你可以去看以前的文献，例如，Parnas关于信息隐藏的论述，就会发现这种概念可以看作面向对象编程中类概念的一种抽象的原型。第二，它意味着继承，我认为继承有利也有弊，这跟如今很多人的使用感受一致。

另外，我应该进入更多的领域，计算机科学领域内外都应涉猎。你学的东西越多，开始得越早，就越有好处。我一直没有真正做过的就是GUI编程，在某种程度上说我应该强迫自己做做看。但是由于种种原因，如这些年来开发库代码，构建他人可以使用的代码块，这些事情已经占据了我的大部分时间。这样算来，我做数据结构和算法等方面的工作已经有几十年了。

Seibel：*有什么书是所有程序员都应该看看的？*

Bloch：《设计模式》无疑是一本，虽然我对它的感情有点复杂，但是我还是认为每个人都应该读一下。书中列出了通用的词汇，也提出了很多好的创意。另一方面，这本书有点儿像风格和语言的大杂烩，内容也有些过时了。但是我认为它绝对值得一读。

另外一本书是*Elements of Style*，它甚至不是一本编程书。为什么要看这本书呢？理由有两条。首先，每个软件工程师工作中很大一部分是写文档。如果你无法写出精确、统一、易读的说明书，那么没人会去用你的产品。所

以说可以改善你写作风格的东西都值得借鉴。其次，该书里面的大部分思想都适用于编程。

我的荒岛读书单有点古怪。例如，最重要的书是Herry Warren写的*Hacker's Delight*。

Seibel：这是一本位操作（bit-twiddling）的书？

Bloch：是的。我爱位操作，这跟我的工作有关系。如果你写库、编译器、底层图形代码或者加密代码，这本书是不可或缺的。Warren把曾经口口相传的东西放在一起，用严谨的数学去验证。这本书出版的时候，我被震惊了。

当然还有Knuth的《计算机程序设计艺术》。事实上，我从来没有读完这一套书，没有从头到尾看过。但当我研究某个具体算法的时候，我就去看他会怎么说。往往可以得到我想要的东西，这套书太全面了。

但是我没有能力、也没有时间去读完整套书，所以如果我告诉你我读完了，那么我就是在说谎。我觉得还有一本非常好的老书，是Kernighan和Plauger写的*The Elements of Programming Style*。书里面的例子都是用Fortran IV和PL/I写的，所以有点过时。不过，虽然这本书这么老，但里面的思想却从未过时。

另外一本老书是Frederick Brooks的《人月神话》。这本书都出版40年了，里面的思想仍同甫出版时一样有影响力。阅读它是一种快乐，每个人都应该读一下。这本书的主要信息是“给一个延期的项目加人，会让它延期得更加厉害”，今天这一点仍旧是正确的。里面还有其他很多重要的观点。有些细节虽然过时了，但仍值得一读。

现在每个人都必须要学习并发编程。所以应该看看《Java并发编程实践》这本书。虽然标题中有Java，但是很多内容并不限于任何具体的编程语言。

Seibel：这就是你和Brian Goetz合著的那本书？

Bloch：我的名字是印在封面上的，但是我提到它的原因恰恰是因为这不是我写的书。第一作者是Brian，第二作者是Tim Peierls，其他几个作者都是制定Java并发标准JSR-166的人。把我的名字印在了封面上仅仅是出于礼貌，我贡献了些材料，但是没有正式参与编写此书。

噢，还有一本：《韦氏学院词典（第11版）》。我去哪里都带着它。这倒不是你实际上要读的东西，但是我说过，写程序的时候，必须能命名好变量。你的文笔必须好。没有好的字典，我就会觉得少了点儿什么。

Seibel：*除了命名好变量，尽量少地复制粘贴以外，你经验丰富后，还有哪些写程序的习惯改变了？*

Bloch：随着年龄的增长，我逐渐意识到编程不仅仅是让程序运行而已，编程是创造一个易于理解的、可以维护的、高效的作品。一般来说，我发现干净整洁的代码，往往运行起来更快。这与流行观点正好相反。而且即使它们不快，也可以很容易地让它们变快。正如人们所说的，优化正确的代码比改正优化过的代码容易多了。

我的一些改变是跟具体语言相关的。每种语言都提供了一个工具集。你要使用正确的工具，在这种语言中正确的工具在另外一种语言中可能不是最好的。举一个简单的例子，如果你用Java 5，使用枚举来代替整数常量或者布尔常量可以大大简化程序，让它更安全，更可靠。

Seibel：*说到这儿，你能否谈谈如何能快速熟悉一种新语言？*

Bloch：这跟人类的语言很类似。一种方法是学会很多语言。如果你已经熟悉意大利语和西班牙语，那么学葡萄牙语就不需要花太多时间了。你知道得越多，你能吸收得就越多。

学习一种新语言的时候，要利用以前所学的语言的功底，但是也要保持开放的心态。有些人执着于一种理念："这就是写所有程序必须遵循的方法。"我不说是哪种语言，但是某些语言，出于某种原因，令人执着于这样的理念。当开始学习新语言的时候，他们会批评这种语言跟真正的神的语言的所有不同之处。当他们使用新语言时，他们极力使用真正的神的语言的方法去写。这样，你就会错过这个新语言真正的独特之处。

这就像你本来只有一个榔头，有人给了你一个螺丝刀，你说："唉，这不是一把好榔头，但是我应该可以倒着抓住螺丝刀，用螺丝刀把来砸东西。"你得到了一个很烂的榔头，但事实上它是一把很不错的螺丝刀。所以你应该对所有事物保持开放和积极的心态。当然，最重要的是代码！代码！代码！要多用语言，这样才能学得更快。

Seibel：*为什么人们对所选的计算机语言那么虔诚？*

Bloch：我不知道。但是选择一种语言时，所考虑的不仅仅是一系列技术上的权衡，而是在选择一个社群。这就像选择一个酒吧。没错，你希望去一个提供美酒的酒吧，但是美酒不是最重要的，主要是那个酒吧里都有什么样的人，他们在谈论些什么。选择计算机语言也是这样的。时间一长，就这门语言也形成了一个社群，社群里不仅仅有人，还有他们的软件成果，如工具、

库等。这就是有些理论上看起来更好的语言无法成功的原因，他们无法在周围构建成功的社群。

Seibel：*要是这么理解的话，Java就非常有趣，它有两个社群。一个是实现者和系统开发者，也就是在Javasoft、Weblogic或者类似地方工作的人们。另一个是所有用Java、应用服务器、构建好的框架来构建商业应用的人们。这两个酒吧差别非常大。*

Bloch：与Java或者其他语言关联的社群很多。如果语言周围没有形成社群，这通常表明，要么这个语言无人问津，要么就还很不成熟。语言繁荣发展，就会自然地表现为出现越来越多各式各样的社群。同时，如果对一门语言的投入总额增长了，那么它的价值也会相应地增长。

这就像梅特卡夫[①]定律：网络的价值与用户数量的平方成正比。这对于编程语言也适用。所有使用这种语言的用户构成社群，然后突然间出现了Eclipse，出现了FindBugs，出现了Guice。即使Java对你来说不是最好的语言，但是使用它有这么多附加的好处，所以你还是会创建社群，解决如何在Java中进行数学编程，或者你需要的其他类型的编程方法。

Seibel：*你现在还像小时候那样觉得编程是一种享受吗？*

Bloch：是的，虽然不是完全一样的方式。像其他孩子一样，那时候我觉得某种程度上编程是遇到生活中处理不了的问题时候的避风港。另外，年轻的时候，你有无尽的精力，你可以不停地钻研。

后来岁数大了，有了家庭和孩子，有了责任，有了其他重要的事情。但是，写个程序，看着代码飘落，错落有致，写出几句优美的、易懂的、高效的代码，做你想做的事情，还是那么令人兴奋。

Seibel：*你有没有过这样的感觉：你越来越清楚那程序就是搞不定，到处都是这样那样的问题，几乎令人绝望？*

Bloch：当然有过。写书也是这样的。每次开始一件事情的时候，我总想逃避。开始是最艰难的，有时候我会鼓励自己："加油Josh！这行你都干了三十多年了，你知道怎么能做到跟别人一样好，不必瞻前顾后，动手吧。"或者对自己说："你瞧，以前的一切你都做得挺好的，这次也错不了。"

Seibel：*你刚才提到你的生活体验变宽广了，这可能会影响编程，但是有没有什么东西，是编程以外的体验，但是帮助你成为了一个更好的程序员呢？*

① 罗伯特·梅特卡夫是3COM公司创始人、计算机网络先驱。——编者注

Bloch：当然有。我认为你能做好的每件事情都有这个作用。思想是没有学科限制的。我想到了一个例子。我写论文的时候，要对一种分布式的数据结构RSM（Replicated Sparse Memory，复制型稀疏内存）做一个分析。而做这个分析的基础思想来自于我所上的化学课。那是一个动态平衡公式：如果系统里有一个动态的平衡关系，就可以写出一个等式，“事物进入一个特定状态的速度，和他们离开这个状态的速度相等。”这样就得到了三个变量的三个等式，求解这三个等式，计算出来的结果和观察到的这种复杂的分布式的数据结构的行为恰好匹配。这就是我直接从化学里偷来用于计算机科学的思想。

你在生活中看到的很多东西，不管是架构上的，即建筑物构建的方法，还是在语言上的，即人们进行沟通的方法，很多思想都是可以借用的。当然包括数学。数学和编程相当类似。所以要睁大双眼，积极地吸收重组各种思想，这样做绝对错不了。

Seibel：*你认识有什么伟大的程序员不会数学或者没有接受过良好的数学教育的吗？要成为一个程序员，学习微积分、离散数学和其他的数学知识真的那么重要？还是做程序员只需要一种思想方式，即使没有受过这些数字训练，也能拥有？*

Bloch：我觉得是思想方式，学不学数学都能拥有这种思想。但是学一下确实有好处。我曾有个同事叫madbot，Mike McCloskey。他很懂数学，但是没有学过数论。他重写了BigInteger的实现。原来的实现是C语言函数包的封装，他发誓用Java重写，要达到基于C语言版本的速度。后来他做到了。为此他学了大量的数论知识。如果他的数学不行，他肯定搞不定这个项目，而如果他本来就精通数论，就无需费力去学习了。

Seibel：*但是，这本来就是个数学问题啊。*

Bloch：对，这个例子不恰当。但是，我相信即使是跟数学无关的问题，学习数学培养出的思维方式对编程来说也是必不可少的。例如，归纳证明法和递归编程的关系非常紧密，你不理解其中一个，就不可能真正理解另外一个。你可能不知道基础条件和归纳假设这两个术语，但是如果你不能理解这些概念，你就没有办法写出正确的递归程序。所以，即使是在与数学无关的领域内，不理解这些数学概念的程序员也会遇到很多困难。

你刚才提到了微积分，我觉得它不那么重要。可笑的是这么多年来似乎已经成为了一种思维定势了，只要你受过大学教育，那么人们就认为你应该

懂微积分。微积分中有很多美妙的思想，可以让人展开无穷的想象。

但是，你可以以连续或者离散这两种不同的方式思维。我觉得对程序员来说，精通离散思维更为重要。例如我刚提到的归纳证明法。你可以证明一种假设对所有整数都成立。证明过程就像施魔法一样。首先证明它对一个整数成立，然后证明针对这个整数成立意味着针对下一个整数也成立，这样就能证明它适用于全部整数。我认为对程序员来说这比理解极限的概念要重要得多。

好在我们无需选择。大学课程里这两样都教得不少。所以即使你用微积分用得没离散数学那么多，学校里还是应该教授微积分的。但是我认为离散的东西比连续的东西更重要。

Seibel：*前面你提到写程序和写文章有许多相似之处。尽管数学和计算机、编程的联系一直很紧密，但是是不是可以认为，写Web框架或者基于Web框架的Web应用程序所需要的技能跟写作的关系更为紧密呢？*

Bloch：是啊。前面你提到Java程序员有两个不同的社群。编写库、编译器和底层框架的社群，更需要数学知识。而如果你是在底层框架之上编写Web应用程序，那么必须了解如何进行沟通，言语上的、视觉上的沟通都需要了解。遇到那些令我操作失误的网站我就很恼火。显然有些人完全没有考虑过用户怎么使用他们的产品。所以实质上，编程能力是一系列不同技能的结合。你擅长哪些技能，决定了你擅长编写什么样的程序。但是，即使是库、编译器以及底层框架也需要代码可读、可维护。如果你不擅长写作，你就很难达到目标。

Seibel：*你设计软件的流程是什么样的？打开Emacs就开始写代码，然后改来改去直到程序写好？还是坐到沙发里拿着一叠纸先列个提纲？*

Bloch：很多年前，我在OOPSLA[①]上做了一个演讲，题目是“如何设计一个好的API，以及这为什么很重要”。网上可以找到这个演讲的几个版本。这个演讲很好地解释了我的设计流程。

最重要的是了解你到底要设计什么，也就是你要解决的是什么问题。需求分析的重要性怎么强调也不过分。有人认为：“噢，需求分析呀。跑到顾客那边问问他需要什么。得到客户的答案不就成了嘛。”

---

① 面向对象编程、系统、语言和应用国际研讨会。

事实绝非如此。这不仅是一个协商的过程，而且是一个理解的过程。许多顾客不会告诉你问题，而会告诉你一个解决方案。例如，顾客可能会说："我需要你给这个系统加上以下17个特性。"那么你必须问："为什么？你想用这个系统做什么？你期望它怎么发展？"你要来来回回好几次，直到弄明白顾客真正需要软件去做的所有事情。这些就是用例（use case）。

这个阶段最重要的事情就是提出好的用例。一旦有了用例，你就有了用来比较所有备选解决方案优劣的基准。你可以花大量的时间去改进用例，因为一旦用例错了，你就彻底失败了，所有后续的流程都会徒劳无功。

我见过这样的事。有人找来一帮聪明人，还没搞清到底要做个什么样的系统，就开工了。辛苦地工作了6个月，写出来247页的系统规范文件。这是最糟糕的情况。因为6个月后他们精确制定出来的系统可能毫无用处。他们往往会说："我们已经投资了那么多，制定出来了规范文件，我们必须把这个系统做出来。"所以他们创造了一个没有任何用处的系统，这个系统也从未投入使用过。多恐怖啊。如果没有用例就做好了软件，那么当你试图做点非常简单的操作时就会发现："哦，我的天，像选择一个XML文档并打印这么简单的事情，需要这么多的代码啊。"这是很恐怖的。

所以先获取这些用例，然后编写骨架API。骨架API应该很短很短，也就一页纸的内容吧，一般正好是一页。无需非常精确。你要声明包、类和方法，如果还不清楚他们应该什么样的话，可以放一句话的描述。不过这不是产品发布要求的那种质量文档。

中心思想就是在这个阶段保持敏捷，逐步完善API，使其满足用例，为原始的API添加代码，看是否可以满足需求。真是不可思议，很多事情事后看真是太浅显了，但设计API的时候，甚至是构思用例时，你还是会犯各种错误。用代码实现用例时你会说："哦，我的天，全都错了。类太多了。这些可以合并，这些需要拆开。"或者类似这样的话。好在API文档只有一页长，改起来也很容易。

你对API越来越有信心，代码也就越写越长。但是，核心原则是，先写使用API的代码，然后再写实现它们的代码。因为，如果实现代码被废弃，之前的工作就都白做了。事实上，应该在给出设计规范前写API的代码，否则你可能把时间浪费在给最后完全不需要的东西设计规范上。这就是我设计软件的方法。

Seibel：设计Java集合类这样的一个具体的自包含的API，设计规范需要有多具体？

Bloch：我敢说比你想的要粗略多了。任何复杂的编程都需要API设计，因为大程序都需要模块化，你必须设计模块之间的接口。

优秀的程序员把问题分解开，一块块地去看待问题。这样做的理由有几条。比如，你可能会在无意中创造出好用的、可重用的模块。如果你写一个单一的系统，它越来越大，等你想分块的时候，已经无法找到清晰的边界，最后系统就变成了一个无法维护的垃圾。所以我断言，无论你是否把自己看成API设计者，把问题分块都是最好的编程方法。

当然，编程的世界非常广阔。如果对你来说编程就是写HTML代码，那么这也许不是最好的编程方法。但是，我认为对于大多数编程来说，这就是最好的方法。

Seibel：*所以你希望系统由不同的模块松散地耦合在一起。要达到这样的目标，现在有两种不同的看法。一种是坐下来实现设计模块间的API，像你前面提到的那样。另外一种是，"构建可运行的最简系统，然后毫不留情地重构"。*

Bloch：我不认为这两种方法是完全不相容的。某种程度上，我谈的就是测试先行编程，以及对API的重构。如何测试你的API呢？在实现API之前编写它的测试用例。虽然我还不能运行用例，但我在进行测试先行的编程：实现用例后看API是否能完成任务，我用这样的方法测试API的质量。

Seibel：*也就是说你写好使用API的用户代码，然后评审代码："这就是我要的代码吗？"*

Bloch：对！有时候你都不用走到评审用户代码的这个阶段。写代码的时候可能就会有感悟："写不出来，我忘了这部分API的功能了。"或者："这代码写起来太乏味了，一定是哪里出错了。"

这跟你多么优秀无关。不用API写代码，就不可能看出API有什么问题。设计了一个东西，使用了才知道："哦，错得这么离谱。"如果是在你浪费大量时间基于这个API写了无数代码之前的话，那么这就是一个重大的胜利。所以，我谈的更多的是测试先行编程和对API的重构，而不是重构API的实现代码。

说到能够运行的最简程序，我完全赞同这种提法。API设计有一条基本原则：疑则不用。它必须是完全满足你关心的所有用例的最简系统，而不是说"把乱七八糟的代码堆在一起"。有很多格言警句说明了这点。我最喜欢的一条是："简单没那么容易做到。"坊间认为就是Thelonious Monk说的，

实际不是，是误传。

没人喜欢烂软件。人们提倡“构建可运行的最简系统，然后毫不留情地重构”，而不提倡“写垃圾代码”，更不会说“不要做前期设计”。我曾跟Martin Fowler讨论过这个问题。他坚信，只有仔细推敲要做的东西，系统才会有合理的形状和结构。他说过：“不要在写代码前先写下247页的设计规范。”我很赞同。

我不赞同Martin的一点是：我认为测试远不能用来取代文档。只要你写了别人编程时可以利用的代码，你就需要做出精确的说明，而测试确保这些代码符合你给出的说明。

所以两大阵营确实有些不同意见，但是我认为他们之间的鸿沟没有某些人想象的那么大。

Seibel：*既然你提到了Fowler，咱们就聊聊他。他写了很多关于UML的书，你把UML当设计工具用过吗？*

Bloch：没有。我觉得用UML做些图表让其他人理解起来可能更容易。但是说实话，我根本记不住那些组件应该是方的还是圆的。

Seibel：*你尝试过Knuth的文学编程方法吗？*

Bloch：没有。原则上我并不反对。我只是没有机会尝试。另外就是……我不知道该怎样说才能不失礼。这么说吧。我不笃信宗教，不笃信任何宗教。不管是面向对象编程，还是函数式编程，不论是基督教还是犹太教，我会吸取其中有用的东西，但是不会全盘接受。文学编程中有很多好的思想，作为体验去尝试一下也许不错，但这不是我最钟爱的那种酒吧：在里面徜徉的程序员太少了。

反之，我很乐意花几个小时来选择标识符名字、变量名、方法名，等等，力图让我的代码可读性更好。如果你发现表达式里用了这样的标识符，读起来就像英语的句子一样，那么你的程序很有可能就是对了，而且易于维护。我想有些人会说“这是浪费时间，不过是变量的名字而已”，他们根本没想明白。有这样的想法，你就写不出来可以维护的程序。

Seibel：*编程和写作有区别的一点，至少是跟非实验文学有区别的一点，是代码是没有阅读顺序的。那你是如何阅读那些别人写的大型代码的呢？*

Bloch：问得好。我真希望那些程序是精心写出来的。我知道有几个人，他们可以处理任意大的、写得很烂的代码，把它们封装起来，然后琢磨出整个

框架的设计思路。这真是非常有用的技艺，不过我从来没学会过。

我希望能够孤立地处理小的模块，读代码，理解含义。如果我试图理解一个耦合紧密的系统，我就必须阅读所有的东西才能够理解其中的一个部分，这简直是一场噩梦。如果真要这么做，我必须事先做好充分的精神准备，我需要同时阅读所有的代码。我通常会把所有代码都打印出来，然后坐在这堆纸中间，在上面记笔记。

如果读的是书写风格良好的代码，我就试图找到一个全局的视角：通常会有人写整个系统结构的描述。如果能找到，我就可以知道哪些模块最重要，先阅读这最重要的，偶尔查看一些更底层的模块来增强理解。

另外，虽然代码是一行行顺序排列的，但代码的执行却不是线性的。如果幸好遇到一段可以按从头到尾的顺序阅读的代码，那当然好。如果不行的话，那么就必须能用工具迅速定位被调用的方法、继承的类等。这样才能理解代码的关键运行路径。

Seibel：*你用单步执行代码的方式来理解代码？*

Bloch：当然！这是我调试程序的首选方法。尤其是并发执行的代码，状态太多了，我完全无法穷举。我盯住代码，在脑子里面单步执行，考虑哪些不变量在什么时候必须保留。我放弃了许多精美的调试工具，因为没什么东西比在调试器或者大脑中单步执行一个程序功能更强大。我用这种方法发现了很多bug，我把它当作编写代码流程的一部分。

写代码的时候，我问自己，这段代码中哪些东西必须是对的？把这些断言放在代码中很重要，这样才能保证后续运行中断言的正确性。如果你用的程序语言在语法上支持断言，那么使用这个语法；如果不支持，就把断言放在注释中。不管怎么样，这些信息都万分重要，千万不能丢了。六个月后，当你想回头理解自己的程序时，需要知道这些信息。你的同事也可能随时需要理解你的程序，他们也需要这些信息。

Seibel：*你觉得程序员必须理解不变量和如何使用断言吗？*

Bloch：不。你也许知道断言是我加入到Java语言中的第一个结构，但是我知道它也许永远不会成为Java文化的一部分。只有少数Java程序员使用断言。我不太清楚到底是为什么。至于不变量，它应该属于数学领域。

Seibel：*但是不需要太多数学知识就可以理解不变量啊。*

Bloch：没错。不过让我来扮演一下魔鬼的代言人吧。思想在某种程序上的

精确性需要用数学来验证。我曾做过四五年级学生数学奥林匹克队的教练。那正是孩子在某种程度上开始理解证明的概念的年龄，也就是可以明确地证明一个命题，毫不含糊地说它为真，而不是说："我想它是真的，因为有几个例子可以证明这一点。"

要理解不变量的概念，你需要理解证明的概念。但很多成年人都不理解。这是一种在数学课上教授的思维方式。

Seibel：*你有没有想过也许教授这类思维方式的好方法是用编程来教呢？如果你来教授编程中的不变量，会怎么教？*

Bloch：某种程度上，我同意你的观点。但是沿着这个方向你可以进行更深入的探讨，然后就回到了Dijkstra的思想上去了。你应该读过"On the Cruelty of Really Teaching Computing Science"这篇文章，我认为这篇文章错得太离谱。Dijkstra说，整个一个学期内，学生们都不必关心符号的真实含义，只管学习操纵符号，在精通之前不允许他们碰计算机。输入指令，然后看着计算机运行，这是一种快乐。我不想剥夺学生的这种快乐。而且，我觉得我也无法禁止学生碰计算机，现在到处都有计算机。10岁的小孩也会写程序。

Seibel：*作为Google公司里面的Java程序员，你有没有想过Google是否可以多用点Java？如果不考虑现实因素，假如轻挥一下魔棒就可以把Google所有的C++代码用Java代替，这样行吗？*

Bloch：某种程度上是可以的。系统的大部分都可以用Java编写，而且现状也是逐渐往这个方向发展的。但是对系统的绝对核心，例如索引服务器的内循环来说，性能上的一丁儿点提升都有巨大的价值。当这段代码运行在很多机器上的时候，你让它稍微快一点，那么无论是从经济考虑，还是从环保角度看，都会获得很大的收益。所以有些代码你恨不得用汇编来写，C语言不就是种美化了的汇编语言吗？

我不是对某件事物特别虔诚的那种人。能用就好。我写了20年的C语言代码。从消耗程序员多少时间的角度来看，使用现代的编程语言更有效率，而且现代化的编程语言更安全、更便利，表达能力更强。在大多数情况下，程序员的时间比计算机的时间更宝贵。但是当你的程序运行在成千上万台机器上的时候，就完全不同了。所以我们写的有些程序，使用那些可能不那么安全的语言，榨出每一点值得榨出的性能。现在程序员们使用的现代语言效率都差不多，如果有人说他们的语言效率高十倍，那么多半是在骗你。

但是从工程师写程序耗时的角度去看，差异很大。首先，更现代的语言

已经排除了大量的错误实践。其次，它们包含了大量的工具，可以提高工程师的工作效率。可以说这是一种文化，是人们在学校学的语言。但是它也是工作中的基础工程问题。例如，假如一种语言有宏处理器，那么就很难给它写出好的工具。解析C++比解析Java要难多了。

现在，Google用Java写的代码比以前多多了。我不知道具体的数量，但就算还没有达到临界点，估计也快了。不过，各种语言都有多少行代码和在各种语言下执行多少个CPU周期是有很大区别的。试图把索引服务器的内循环用Java改写很愚蠢，不值得称道。如果你是一个初创公司要做类似的事情，可以用Java或者其他现代的安全的语言来写大部分代码，但是在不需要它们的时候，不要用它们。我们有自己的工程基础设施。代码库、监控工具等所有的东西都维系着它。就算Java最终不能获得同等的地位，也会在这些系统中有很多用处，这就不错。我刚到Google的时候，还不是这样的。

公司通常早就确定了DNA，这能够让它们获得巨大的成功，但是也令他们很难换掉那些早期应用良好、后来不再适用的技术。我记得1982年左右，我在约克镇高地的IBM研究中心实习的时候，那里的主流还是批处理系统。甚至当他们已经开始做分时系统的时候，他们还用虚拟读卡机（编程卡片）、虚拟打孔器这样的术语交流。什么东西都用80列的记录。而后来DEC一直将思维禁锢在分时系统上。我估计微软也面对这样的问题，就是他们的思维能否超越桌面PC系统。

Seibel：*20年后，人们将会谈论Google为何只能死守着在互联网上卖广告。*

Bloch：没错。毕竟，在Google还有一部分人认为Java太慢而且不可靠。有这种看法的原因很显然，那就是1999年左右的发布的用于Linux的Blackdown Java[①]，它确实又慢又不可靠。既有的看法总是很顽固的，很难改变。事实上Google在很多核心功能上使用Java，甚至包括广告。

所以某种程度上，他们知道Java既快又可靠。但是在实际的搜索流程中，对机器CPU周期最敏感的领域，所有的东西都基于C++，这么做很明显的一个原因就是公司的DNA。这将在很长一段时间里影响着我们。

Seibel：*你实际编程中用哪些工具？*

Bloch：我就知道你迟早要问这个问题。我是老帮菜了，提这个都觉得丢人。Emacs的键盘快捷方式在我的脑子里面已经根深蒂固了。而且我喜欢写小的

① 一个非官方移植的虚拟机。

程序，代码库之类的。所以，我写代码的时候几乎不用现代的工具。但是我知道，很多现代的工具可以提高效率。

写大程序的时候我确实使用IntelliJ，因为我们整个团队都在用，但是我不是这方面的专家。这个工具给我留下了深刻印象，我喜欢这些工具对代码做的静态分析。我找用Eclipse、NetBean以及FindBug的人来帮我审阅《Java解惑》，书中的很多错误陷阱都可以被这些工具自动检测到，太了不起了。

Seibel：你相信如果花一个月真的学通了IntelliJ，你会更有效率吗？

Bloch：我相信。现代的IDE很擅长做大规模的重构。正如Brian Goetz指出的，现在很多人写的代码更干净，因为他们做了很多重构，以前是想不到这么做的。有了这些工具，就可以保证代码的改动不会改变它们的行为。

Seibel：其他工具呢？

Bloch：我不擅长使用编程工具。我也希望自己能精通这些，但编译和代码控制工具都变得太快了，我跟不上。所以每次我建立新工作环境的时候，都麻烦更擅长工具的同事帮忙。我会问：“你们现在是怎么做的？”他们只要瞄一眼，就能帮我搞定。我会一直使用这个环境，直到它们不能再工作为止。

我这么说一点儿也没有得意的意思。工程师也是有所长，有所短。有些人就不承认这一点，他们以为工程师是标准件，可以互换，每个人可以也应该成为完全的多面手。但事实上，非常擅长某事的人可能对其他事情并不在行。非要逼迫大家样样通的话，出来的产品指定不怎么样。

我要重点说说一类人，用Kevin Bourrillion的话来说，他们“缺乏感同身受的基因”。如果你不能把自己想象成使用你的API、你的语言的普通程序员，那么你就没有办法做一个好的API或者语言设计者。例如，有些人从技术上讲是好的API和语言设计者，他们可能会说：“这样做就不符合LALR(1)了，你应该这样改一下。”这固然是非常有用的技巧，但是这仍然是“缺乏感同身受的基因”，没有意识到自己设计了一个非常难用的可怕的语言。

我知道有些人非常擅长榨取最后百分之一的性能。你应该把他们放在人尽其才的位置上。他们也会很高兴，而且可以给公司做出贡献。你应该找出工程师们的特长，让他们人尽其才。这就是我对自己不擅长使用工具的辩解了。这些理由站不住脚，我有自知之明。

Seibel：我们聊聊调试吧。你找到的最糟糕的bug是什么？

Bloch：提起bug我立马就想到了一个，这个bug很严重，而且很搞笑。那是

90年代初，我在匹兹堡的Transarc公司工作时。我在很紧的工期下提交了一个事务共享内存的实现。我在限期内完成了设计和实现，甚至还在过程中做出了几个可重用的组件。但是这么匆忙地写了很多新代码，我还是挺担心的。

为了测试这些代码，我写了一个叫做“乱撞”的很长的程序出来。它运行了大量的事务，每个事务又包含了嵌套的事务，嵌套到可以嵌套的最大深度。每个嵌套事务都可能会加锁，以递增的顺序读取共享数组里面的几个元素，对每个元素都加入点东西，保持数组中所有元素的和为0，还是不变量。这些事务要么提交，要么取消，大约是90%的提交，10%的取消。多个线程同步运行于这些事务之上，长时间地访问数组。因为我测试的是一个共享内存机制，所以我同时运行好些个有多个线程的乱撞程序，每个都有自己的进程。

在一般的并发级别下，乱撞轻松过关。但是当我真正调高并发级别时，我发现乱撞偶尔，仅仅是偶尔，无法通过一致性检查。我不知道这是怎么搞的。这只能是我的错，因为新代码都是我一个人写的。

我花了大约一个星期，痛苦地为每个组件写了彻底的单元测试，所有的单元测试都通过了。然后我为每个内部数据结构写了详细的一致性检查，这样我就可以在每次变化后调用这些一致性检查，直到测试失败为止。最后，我终于发现一个底层的一致性检查失败了，这个问题无法重现，但是某种程度上可以帮助我分析问题出在哪里。最后，我得出了确实的结论，我的锁根本不工作。两个事务锁定、读写同一个值的时候，产生了并发的读-修改-写回操作，而后一次写入毁掉了第一次的写入。

我编写了自己的锁管理器，所以我怀疑是它出了问题。但是锁管理器轻松地通过了测试。最后，我觉得问题不在锁管理器，而是它依赖的互斥体的实现！那时候操作系统还不支持多线程，我们需要写自己的多线程包。原来负责互斥体代码的工程师，不小心把我们的Solaris的线程实现中的lock和try-lock的汇编代码的标签弄混了。所以，每次你以为你在调用lock的时候，其实调用的是try-lock，反之亦然。也就是说当真的有争用发生的时候——在当年其实是很罕见的——第二个线程直接就进入了第一个线程的临界区，因为第一个线程也没有锁住。搞笑的是，这也就是说，整个公司几个星期都在运行没有互斥体的程序，而且谁都不知道。

Knuth有句关于测试的名言，Bentley和McIlroy的精彩论文“Engineering a Sort Function”中曾经引用过，大概意思是说，做测试时，**要不惮以最大的恶意来推测所要测试的代码的错误**。做这些测试的时候，我就是这么做的。

但是这样会把所有东西纠结在一起，更难找到bug。首先，并发的时候很难这么做，往往完全无法复现场景。其次，到最后可能会发现你的核心假设是错的。喜欢喊“耶，这语言出错了”或者“系统出问题了”是新手干的事儿。但是在这里，我依靠的基石——互斥体，确实出问题了。

Seibel：*也就是说bug不在你的代码中，但是同时你只能对你的代码进行彻底的单元测试，因为你没有别的办法，只能去检查自己的代码。你觉得这些测试是不是可以或者说应该让互斥体代码的作者来写，这样你就不用浪费一个星期，节省了一半的调试量，而且也可以找到这个bug。*

Bloch：给互斥体代码加一个好的自动化单元测试肯定可以避免让我遭受的那些痛苦，不过注意那可是90年代初。我想都没想过要抱怨那个工程师没写个好的单元测试。即使是今天，为并发工具写单元测试还堪称一种艺术。

Seibel：*前面我们谈了单步调试代码，那么你调试的时候使用什么工具？*

Bloch：说到这里就搞得我有点儿像尼安德特人[①]了。对我来说最重要的工具仍旧是我的眼睛和大脑。调试的时候，我把所有相关的代码打印出来仔细阅读。

调试器当然不错，有几次我可以用print命令的时候用了breakpoint命令。所以，我偶尔也用调试器，不过没有调试器我也无所谓。只要我还能在代码中放print语句，能完全读懂它们，我一般就能找到bug。

我前面说过，我用断言来保证复杂的不变量的安全性。如果不变量被破坏，我希望立刻知道，我希望知道什么样的动作会造成这种破坏。

这让我想起了我遇到过的另外一个非常难找到的bug。我记不太清具体时间了，也许是我在Transarc工作期间，也许是我从CMU本科毕业后，总之是研发Camelot分布交易系统时。bug不是我找到的，但是给我留下了很深刻的印象。

我们有一个trace包允许代码发出调试信息。每个跟踪事件上带有其所属线程的ID作为标记。偶尔日志里面会出现错误的线程ID，我们不知道为什么，只好决定暂时容忍这个bug。好像它也没多大的害处。

结果后来我们发现这个bug根本不在trace包内，问题比我们想的更严重。为了找到线程ID，trace包调用threading包。为了得到线程ID，threading包使用了一个当时相当常见的小技巧：查看栈变量地址字节的高几位。也就是说，

① 智人的亚种，已经灭绝，详见参考http://zh.wikipedia.org/wiki/尼安德特人。

取一个栈变量的指针，移几位，然后就得到了线程ID。这个技巧好用的原因是，每个线程都有固定尺寸的栈，栈的尺寸是2的一个已知的乘方。

看起来是个靠谱的做法吧？除非有少不经事的人在栈上构建用当时的标准看起来非常大的对象。100个元素的数组，每个4KB——这样你就往栈里塞入了400KB的数据。你超越了栈的边界，进入了下一个线程栈。这样取线程ID的方法就搞错了线程。更糟的是，当线程访问一个线程本地变量的时候，它得到的是另外一个线程的值，因为线程ID是用来访问线程本地变量的键。

所以我们在跟踪系统里面看作小瑕疵的东西，实际上是存在一个非常严重bug的征兆。当一个事件被标记为thread-43而不是thread-42时，是因为thread-42无意中冒充了thread-43，可能带来非常严重的后果。

这个例子讲的就是为什么需要安全的语言。这不是任何人都能处理的问题。最近在某个大学，有人跟我说，他们学校想先教授C和C++，然后教授Java，因为学校认为程序员必须了解系统底层。他问我怎么看这个问题。

我觉得前提是对的，但结论是错的。是的，学生应该学习底层语言。事实上，他们应该学习汇编语言，更应该学习芯片结构。虽说芯片已经变得异常复杂了，但其实连芯片的性能模型也不怎么好，因为它们实际上是非常复杂的状态机。但是如果学生能理解系统底层工作原理，他们就可以成为更优秀的高级语言的程序员。

所以，我认为学习所有这些东西是很重要的。你是说我的意思是应该从C这样的底层语言学起？不！学生不应该刚接触编程就不得不去处理缓冲区溢出、手动内存分配等。

James Gosling曾经跟我讨论过Java的诞生，“只在极个别的情况下才需要重启应用程序，这个估计是Java最了不起的地方。”通常情况下，新生事物都必须跟几十年前的老东西保持兼容，偶尔才不需要考虑兼容问题，那个时候真是太幸福了。但是很遗憾，十年后必定会遇到问题，你看看Java就会明白。

Seibel：既然你说到这里，Java是否逃脱了灭亡的命运了？它变复杂的速度是不是比变好的速度更快呢？

Bloch：这个问题不好回答。具体说来，Java 5加入了比我们设想的更多的复杂度。将泛型特别是通配符加到语言中到底有多复杂我也说不好。我得为有功劳的人说句话，Graham Hamilton真是了不起，那时候他就想明白了一切，而我不明白。

有趣的是，他抗争多年，希望阻止泛型进入Java语言中。但是在泛型被成功地阻挡在Java外的这些年里，变体的概念，也就是通配符的隐含意义，流行了起来。如果它们来得更早，没有变体，也许我们现在可以有一个更简单的、更容易跟踪的语言。

不过引入通配符有实际的好处。子类化和泛型之间根本就是阻抗失配的，通配符尽力在弥合这种不匹配。但是这么做又显著地增加了复杂度。有些人认为在声明地点，而不是用户地点，可变性是更好的解决方案，但我不太相信这一点。

这仍旧悬而未决，因为它们还没有在真实环境下被海量的程序员测试过呢。经常一些语言只在小范围内获得成功，人们会说："噢，这些语言很不错，只是可惜没有成为世界范围成功的语言。"但是这往往是有原因的。希望Scala或者C# 4.0使用这样的声明地点可变性的语言可以彻底解决这一疑问。

Seibel：*那么是什么推动Java引入泛型呢？*

Bloch：就像所有那些听上去很棒的东西一样，推动力就是相信了新闻宣传。我的思维模式是，"嗨，集合多半都应该是同构的——一组字符串，一个从字符串到数字的映射，等等。而现在默认情况下集合是异构的：它们都是对象的集合，取出时都需要类型转换。这简直是胡闹。"如果我可以告诉系统，这是一个从字符串到数字的映射，它会帮我做类型转换，而且会在编译期间帮我盯着，防止我做错什么，那不是挺好的吗？它可以抓到更多的错误——它可以包含高层的类型信息，看起来是件好事儿。

我认为泛型和其他加入到Java 5的语言特性一样，我们只是让语言去做以前我们要手工去做的事情而已。某些情况下我坚信：for-each就是好。它所做的就是对你隐藏迭代器和索引变量带来的复杂性。代码更短，概念也不复杂。从某种意义上说，它的概念更简单，因为我们为数组和其他的集合创建了这种伪多态机制，你可以遍历一个ArrayList或者一个数组，而无需关心你遍历的是什么类型。

这种思想不能适用于泛型的主要原因是，它是对已经很复杂的类型系统的大扩展。类型系统是很微妙的，修改它们可能对语言带来深远的、难以预期的影响。

我认为得到的教训是，当你改进一个成熟语言的时候，你必须更加仔细地考虑能力和复杂度之间的平衡。而且，实际上，复杂度跟语言的功能数量

间至少是平方级关系。为一门老语言加上了一个新的功能，通常就意味着为它加入了一大堆复杂度。当一种语言已经达到或接近程序员理解能力的极限时，你加入任何复杂性进来都会加剧理解的难度。

语言更复杂后就会消失吗？不会。我认为C++早已超越了它的复杂度极限，但是还是有很多人用它编程。可这实际上是逼人们只使用其中一个子集。所以我认识的每个用C++的公司都说："对，我们用C++，但是没有用多实现继承，也不用操作符重载。"有很多功能你完全不用，因为使用它们会造成代码太复杂。到了这么做的地步，我认为实在没什么好处。不像过去，程序员能读懂别人的代码，方便移植，那多好啊，现在则不一样了。

Seibel：*如果你去掉泛型，现在Java会变得更好用吗？*

Bloch：我不知道。我还是喜欢泛型。泛型能帮我找到代码中的bug。泛型可以让编译器强制做一些限制，之前这些限制我只能放在注释中。另一方面来说，当我看到那些疯狂的参数类型相关的错误信息，当我看到像`class Enum<E extends Enum<E>>`这样的泛型类型声明时，我就会想，显然泛型的设计还没成熟到可以放到Java中的水平。

我们总是太乐观，然后搬起石头砸自己的脚。所以我们说："耶！我们当然可以把泛型放到Java中。在CLU的时候我们就知道泛型了。这技术25年前就有了。"最近我听到关于闭包的类似言论，不过那是50年前的技术了。"噢，闭包很简单，不会给语言加入任何新的复杂性。"

嗯，没错。但是我觉得我们从泛型这件事儿得到了教训。在你懂得这个改动会对概念层面带来什么影响之前，在你可以确保软件行业从业人员可以高效地使用新特性，而且这一新特性会让他们活得更好之前，你不应该给语言加入这一特性。

如果早知道程序员们对泛型是这个反应，我们肯定不会把它加到Java里。这是不是说我们就完全不会搞泛型？不，我不这么认为。我认为泛型确实很好。主要是因为大多数集合是同构的，而不是异构的，同构的集合处理起来是比较方便的。多数情况下类型转换都不合适。转换可能会失败，而且让你的程序不再优雅。我想你知道这是什么集合，它应该自动符合你的这些需求。但是，是不是这就意味着你应该承受我们现在承受的这种复杂度？不，我想我们只是没有处理好泛型。

Seibel：*关于泛型有来自于用户的压力吗？有人抱怨没有泛型就写不了程序了吗？*

Bloch：有没有工程师痛斥缺乏泛型？不，很遗憾没有，没有人抱怨过。只是因为泛型简洁就把它们加进来，我有些内疚。当时看上去这么做是对的。

话说回来，很多工程实践是凭感觉。有人要求我们加入foreach了吗？没有。也没有人要求我这么做，但是我就是知道这是应该做的。我对了——每个人都喜欢它。但是我觉得我们行业内的一大问题就是，在工程实践中做一个东西，往往仅仅因为它简洁，或仅仅因为它是一个好的工程，等等。如果你不能解决真实用户——在这里就是Java程序员——的真实问题，那么你就不应该加入新的特性。

James Gosling曾做过一个非常了不起的演讲——"Java的感觉"。他说，给Java加入任何东西之前，都需要三个真实的用户。不应该因为一个东西简洁就把它放进来。

但是人们就是想把什么东西都放进去。工程师是做什么的？他们就是写代码的。而当他们写一个库，或者一个语言的时候，他们就是想放自己的各种东西进去。你需要他人的参与，需要指导的声音，需要这些东西来帮助你完成产品，帮你在放与不放之间做出最好的权衡。因为你可以放进去的东西总比你应该放进去的东西多。那么是不是说所有的这些东西都不好呢？那也不是。只是你需要做出决定，某些东西是不应该放进去的。

Seibel：*我阅读《Java解惑》和《Effective Java》的时候，才猛然发现，Java这么一个力图简单的语言还是有不少诡异的地方。*

Bloch：是的，没错，不过这才是生活嘛，所有语言都有的。你还没看过《C语言疑惑》。为什么不看看呢？

Seibel：*因为里面都是迷题。*

Bloch：是啊。这书值得买。在Java中，这些迷题值得收藏的原因恰好是因为你觉得Java是一个简单的语言。每个语言都有自己的特殊情况，Java也有，但很少，还不至于让人觉得无趣乏味。

Seibel：*使用Java并思考Java的设计，是否让你学到了什么跟编程有关系的东西？*

Bloch：我学到的东西太多了。比如我知道了即使是想把一个很小的程序写对也是非常难的。我把这个想法发表在了博客里，题目是"几乎所有的二分搜索和归并排序都是错的"。认为自己程序没有bug就是在愚弄自己。程序肯定有bug。多数情况下，程序里的bug足够少，足以使其完成任务。

我知道，既然写正确的程序那么难，我们就应该尽力获取帮助。所以能减少bug的所有东西都是好的。这就是我是静态类型和静态分析的信徒的原因，任何可以减少某个特定类别bug的东西都是非常好的，任何可以让程序员的工作更轻松的东西都是好的。

我更加确信有好的API文档是很重要的。Javadoc对这个平台的成功也起了作用，只是人们较少提及。好的API文档一直都是Java文化的一部分，也许是因为Javadoc从一开始就存在吧。

我一直信奉"简单就是美"这句话，现在更是如此。我不断看到后加的复杂的东西最终被证实是有害的，只是有的很久才发现，有的立即就显现。我设计的时候，会仔细看着我的"复杂度计量表"，一旦复杂度要到红线了，就需要重新设计了。

偶尔我会遇到不相信这些的人们，他们会说："Josh你太傻了，你怎么就是不明白。这才是应该做的，可惜你就是搞不懂。"我就是不信这些。我觉得事情一旦复杂起来，那么一定有什么地方错了，也许到了寻找更简单的方法的时候了。

Tony Hoare的图灵奖获奖感言中有一句充满了大智慧的话，讲的是设计一个系统的两种方式："一种是尽量简单，这样显然不会有什么问题；另外一种是，尽量复杂，这样没什么显然的问题。"

后面说的话同样饱含智慧，但是知道的人不多："第一种方法其实更难。它需要从复杂的自然现象中发现简单物理规律的那种技能、投入、洞察力，甚至是那种灵感，同时还需要你能接受你的目标受限于物理、逻辑和科技的约束，以及在目标间有冲突的时候可以妥协。委员会不会这么做，除非已经完全来不及了，不得不做。"

Seibel：*你是否想过在职业生涯中再次更换你的主要语言，还是准备退休前一直做Java？*

Bloch：我自己也不知道。我从C语言转向Java有点突然。从研究生毕业时起，到1996年，我主要使用C语言编程，然后一直使用Java直到现在。我已经预见到我可能要更换到其他语言了。但是我不知道是什么语言。也许它还不存在。我觉得产生一个新编程语言的时机已经成熟，但是同时我又觉得平台的惯性也比以前更大了。现代的平台不仅仅是一个语言和一些库，它包括很多工具，是一个虚拟机，一个庞然大物。创建一个完整的新平台的前景比以前更不乐观。

我不知道将出现什么。但是我认为如果改变我的主要语言是对的，那么我就会这么做。我想尽力保持开放的心态。我想尝试更多的语言。我最近没时间做，但是以后还是会做的。

Seibel：列出几个你想尝试的语言吧？

Bloch：我想试试Scala，虽然我怀疑它是否能成为未来的新宠。我很崇敬Martin Odersky。我觉得他写的语言中有很多精妙的想法。但是我同时也认为他加入了太多复杂的东西，太学术化了，所以很难取得世界范围内的大成功。当然我还没权利去评价，因为我还没学习过。

我还想用用Python。Scheme不是新生物，不过我也想试试。我想花几个月，跟我儿子一起过一遍《计算机程序的结构和解释》一定很有意思。每个人都说这是一本伟大的书。我已经买了，算是开了个头。看完它需要一些时间。我想这就是我目前想学的。

Seibel：现在很多人在讨论我们写程序的时候，如何能把未来的多核CPU的优势利用起来。Java显然是第一个内建多线程机制的主流语言。你觉得Java的方法在多核的世界是否仍然可用？

Bloch：我想说得更深入一些。我认为Java是现有语言中最好的。但有趣的是，现在很流行谈Java是否即将死去。我觉得这基本上是扯淡。我认为现在最好的多线程构件就在Java里。我认为Java将迎来复兴。我不是说它是未来20年内最先进的，也不是说它是处理多核的最好方式。但是我认为从现有的东西来看，我们是足以傲视同侪的。

Seibel：你认为谁是Java的竞争者？

Bloch：噢，我想是C++和C#。

Seibel：像Erlang或者软件事务内存（STM）这样的东西呢？

Bloch：据我所知，STM在主流语言中还不存在一个具体的形式。如果STM已经成熟了的话，我想Java里面会有，不会晚于在其他的语言里出现。

Erlang处理并发的方法是actor，如果actor很成功，那么也可以被很多其他语言实现。是的，Odersky和他的公司已经在Scala里面实现了actor。我不确定actor是不是多核机制的最佳拍档，但如果它们是，我想很快会有人在Java里面实现的。

Seibel：所以，正如你所说的，Java提供了基础构件，使用它们你可以可移

植地访问OS提供的线程支持，还有一些在java.util.concurrent API包内的高层构件。但是它们比起Erlang或STM还是有点儿底层，不是吗？

Bloch：我说不好。有些Java构件比较底层，例如AtomicInteger；有些是中层的，如CyclicBarrier；有些是高层的，如ConcurrentHashMap和ThreadPool-Executor。我相信只要人们愿意去找，总可以在Java的“并发构件”中找到和STM以及actor一样舒服的方法。

某种类型的事务内存未来可能会变得重要起来，也许是用作并发库设计者的构件。但是我不认为STM可以成为一个工具，让应用程序员不再担心锁，从而活在无需担心线程冲突的美丽世界里。这是不太可能会发生的。

理由很多。有一个是我开发事务系统时明白的。当你试图在字节级的读写操作之上做自动锁定和乐观并发管理时，你最终会遇到线程间的“假争用”：你会遇到跟逻辑冲突无关的物理冲突。如果你被迫去想出要获得什么锁，除了按要求解决逻辑冲突之外，尽量不获取任何锁。

所以可以打个比方，如果你有两个线程，都要对一个计数器加一，它们应该被允许并发执行。它们可能访问同一块内存，但是从逻辑上看它们互相并不冲突。如果你有一个读取计数器的线程，还有另外一个增加计数器的线程，这两个线程就产生了冲突。但是你可以有任意多读取者，或者任意多增加者并发执行。我见过的系统没一个可以搞清它们的。这个例子可能有点假，但是在物理争用被限制得比逻辑争用更严的时候并不罕见。

STM的另外一个问题是，在一个事务内任何类型的操作都不能发生。I/O是个典型的例子。还有第三个问题，某些STM机制允许“死亡事务”在不一致的情况下查看内存，这样可能带来非常严重的后果。而且，在我们构建通用分布式事务系统的时候，还有大量的问题需要解决。虽然有解决方案，但是我知道所有解决方案都会增加复杂性，降低性能。

所以，我认为，STM仍处在研究阶段。研究这个真的很伟大。我只是不相信在并发领域存在什么灵丹妙药，至少在可见的未来内是没有的。

Seibel：好，换个话题：你喜欢以什么样的方式同其他程序员合作？

Bloch：我这个人很好相处，哪种方式方便就用哪种。我喜欢“结对编程”，你和某人一起工作，不过不用同一个键盘。你们分别写系统的不同部分，来回交换代码。你们甚至不需要在同一个半球。Doug Lea和我就这么工作了好几年。一个人写了一个接口，然后另外一个人可能会说：“不错，不过这部分写得有点儿烂，我来重写一下吧。”

最后得出一个我们俩都满意的接口，然后我实现非并发版本，他实现并发版本。这时会找出所有我们做错的地方，重新修改接口。我们阅读对方的代码，他会说："噢，这么做可以让代码运行速度更快。"我回应："你是对的，Doug，就照你的改。"他非常善于让代码速度飞快，几乎可以和虚拟机媲美。这是我喜欢的一种合作风格。我们可以远程协同工作。

我喜欢和人坐在同一个终端前一起写代码，但是我没有用这个方法从头写过多少程序。基本上只发生在代码复查的时候。我要做代码复查，肯定会有许多代码要改，我就会说："为什么我们不坐在一台电脑前把它处理完？"这么做有不少好处。我觉得这是很好的教授方式，可以把你知道的东西传授给别人。

我不喜欢完全与世隔绝地工作。写程序的时候，如果遇到一个棘手的设计决策，我就会找别人一起讨论。我工作的每个地方，都有几个同事可以一起讨论。这对我非常重要，我需要这样的反馈。

Seibel：*讨论是为了得到你需要的反馈，还是只是需要一个机会把它聊清楚？*

Bloch：都是。我们做的东西里面细节太多了，经常遇到就是没有最佳解决方案的情况，或者有，但是没用过就是想不到。你不得不出来找一个不同观点的人聊聊，讨论非常有用。

我知道有些人不这么看，他们喜欢在真空的环境里写程序。我想这对他们没好处。你可以更早发现自己的bug，在设计期间就发现bug真的好过在写出来以后才发现。所以当你在不同方法和特性间挣扎的时候，当你不知道是该支持这个还是那个的时候，你应该跟别人讨论一下。当然，你也不能把每个人的话都当作福音，因为你会得到相矛盾的意见，最终，做决定的那个人是你。

Seibel：*这就引发了另外一个老问题，Weinberg在他70年代出版的《计算机程序设计心理学》一书中提过，现在的极限编程信仰者们也一直谈论：代码应该被一个人所有，只有所有者可以接触它？还是项目中的每个人都可以共同拥有这些代码，所以每个人都可以修改？*

Bloch：我认为代码的所有权是不能否认的。某种程度上像母子关系：你生下了那些你写的代码，特别是它很大、很复杂而且是原创的，它就是你的。如果你工作在别人的代码下，那么在修改之前跟他们谈一下。尤其是你觉得哪里有错误的时候，因为你也可能错了。如果你破坏了别人的代码，那可不行。

当然，如果在一个组织内一段代码属于具体的某个人这也不合适，因为如果这人离开了组织，这些代码只好被束之高阁。所以有几个人同时明白每段代码，可以修改他们，是很重要的。但是我认为期望每个人都拥有全部代码是不切实际的。

这也涉及了我们刚才讨论过的专业领域的问题。没几个人真的会写复杂的位操作的代码，如果你发现某些代码内部做了位操作，而你不是会位操作的那几个人之一的话，那么你最好跟他们谈谈。做这些东西的人们喜欢整天坐在那里琢磨如何从操作语句里面去掉一句，或者放一个标记来加速计算。但是弄坏它们也很容易。而且很容易写出那种2^{32}个输入中有（$2^{32}-1$）个工作正常的代码。单元测试不一定能测试出你的新解决方案中无法正常工作的那一条。而且如果不是你弄坏的，你也会成为替罪羊。

Seibel： *谈到写复杂的代码，我注意到有些某种意义上说太聪明人写出的代码会很烂。因为他们确实可以把整件事情都丢到脑子里，所以他们可以写出来一大堆乱成一团的代码。*

Bloch： 我同意你的说法，这些人聪明到了可以处理非常复杂问题的程度，而同时缺乏对我们这些无法理解他们代码的人的同情。他们认为："我能看懂而且使用这些代码，所以他们一定是好的。"

Seibel： *编程中是不是有什么固有东西，这种东西会一直牵制着程序员的想法？*

Bloch： 绝对有。我们喜欢有挑战性的工作。但是我们必须把这种激情用在解决实际顾客的实际问题上。如果不这么做，就会被市场淘汰。我想我工作过的第一个公司之所以失败，部分原因就是当时我们不懂实际上我们要做的不仅仅是纯工程。

我们当时真的不认为最重要的事情就是解决实际顾客的实际问题。从你忽视顾客那一刻起，你就失败了。但是我知道这跟有些潜心编程、喜欢做挑战性工作的程序员的想法是有冲突的。但是我觉得你可以鱼与熊掌兼得之。设计API的时候要尊重顾客，然后到了考虑怎么让程序跑得飞快的时候，你就可以全身心地投入解决难题的战斗中了。

当设计并优化算法和数据结构的时候，你还有大把的机会解决难题，尤其是在并发领域。你必须把你的问题想到数学级的精确程度，这非常复杂，为了达到预期效果，你还需要找到组合各种因素的新点子。

但是你必须知道哪里可以而且应该使用这样的思维方式，哪里使用这种

方法会造就一个既无法维护更无法使用的系统。

Seibel：*抛弃这种编程方法的时机是否已经到来？很多底层部件在你使用的VM或并发库中已经实现了。所以对很多人来说，编程就是把一堆代码粘起来。*

Bloch：我完全同意你的观点。相对来说用这种编程方法的人越来越少了。从百分比数量上看，现在必须用这种编程方法的程序员比原来少多了。回到你买台电脑、上面连操作系统都没有的年代，更不消说程序语言还是写好的应用了，反正每个人都必须用这种编程方法。

绝大多数程序员都必须这么做的情况越来越少了，可能已经不存在了。但是从绝对数的角度看，还是有不少这样的人。我们想鱼与熊掌得兼，既想获得安全语言的优点，又想拥有汇编代码的效率，所以我们需要人来写虚拟机、垃圾收集器，来设计可以运行这些软件的芯片，虽然都是用硬件实现的。

我相信还是有很多工作需要喜欢这种编程方式的程序员，但是在确定这样的程序员时必须十分谨慎。如果找到了纯粹的迷题爱好者，你必须做好管理工作，确保他们能将技能用于帮助公司获取最大利益。

这就是问题，也就是，编程是这么一个脑力精英的活动，而这些人往往是公司里最聪明的人，所以他们认为自己有权做任何决定。但是他们是公司里最聪明的人这个事实，不意味着他们应该做所有的决定，因为智慧不是标量，而是矢量。如果你情商不够，就不应该来设计API、GUI或者语言。

我们做的东西是一种美学追求。需要一些数学技巧，一些人际关系技巧，一些写作技巧，这些都不属于工程领域，但是缺了这些你不可能成为一个真正优秀的工程师。所以我觉得这些东西是我们必须牢记的。尽管如此，我仍然认为编程是这个星球上最有趣的工作。我觉得我们很幸运长大在这个时代，可以利用这些技巧做出这样的工作。我不知道在过去的时代我们这样的人应该去做什么工作。

（编辑：朱　巍）

6

Joe Armstrong

米全喜 译

Joe Armstrong最广为人知的是他发明了Erlang编程语言，并且创建了用于构建Erlang应用程序的框架——开放电信平台（Open Telecom Platform, OTP）。

在现代语言的版图中，Erlang有点另类。同很多流行的语言相比，它既老又新。早在1986年，也就是Perl出现的前一年，Armstrong就已经开始了Erlang的工作，当时它只作为商用产品出售并且主要在爱立信公司内部使用。这种情况直到1998年Erlang作为开源项目发布后才发生变化，那时Java和Ruby已经问世3年了。Erlang并非起源于Algol系列中的某个成员，而是源于逻辑编程语言Prolog。当时Erlang设计初衷也相当明确，它针对的软件是类似于电话交换机那样的高可用性、高可靠性系统。

但是几乎在不经意间，Erlang适合于构建电话交换机的那些特征也让它非常适合于编写并发软件，当程序员们开始努力应对多核系统未来的发展趋势时，并发性引起了他们的注意。

Armstrong本人也有点另类。他起初是一名物理工作者，在攻读物理学博士学位时因为用完积蓄而转向了计算机科学，找到一份研究员的工作，为英国人工智能领域奠基人之一的Donald Michie工作。在Michie的实验室，Armstrong接触了人工智能领域各个方面的杰作，成为英国机器人学会的创始成员并撰写了一些有关机器人视觉的论文。

由于Lighthill所做的那份非常有名的调查报告[①]，人工智能的资金来源枯竭，Armstrong又回到了物理学领域，从事了5年多与物理学编程相关的工作。开始时他在欧洲非相干散射科学协会（EISCAT）工作，后来又到了瑞典空间研究中心，最后加入了爱立信计算机科学实验室，Erlang就是在那里发明的。

我们在他斯德哥尔摩家中的餐桌上交谈了几天，谈论的话题很多，包括Erlang对并发的处理机制，为什么需要更好、更简单的方法来连接程序，以及打开黑盒的重要性。

Seibel：*你是如何开始学习编程的？是从什么时候开始的？*

Armstrong：我是从中学时开始的。我出生于1950年，上中学那会儿还没有几台计算机。到了中学最后一年，那年我应当是17岁，我们当地的议会得到一台大型计算机，好像是IBM的。我们可以在上面写Fortran程序。通常，我们在编码纸上写好程序，然后发出去。一个星期后，等编码纸和穿孔卡拿回来的时候还必须确认一下。但是制作穿孔卡的人总会出点错，所以可能要反复一两次才能弄好。最后这些穿孔卡就可以送到计算机中心了。

卡片进入计算机中心后会再拿回来，因为Fortran编译器会在程序中出现第一个句法错误的地方停下来，后面的程序就都不处理了。你的第一个程序似乎需要3个月才能跑通。我认识到，不能每次只送一个程序，应当并行地开发多个单一子例程并且一次都送去。我记得写过一个显示国际象棋棋盘的小程序，用打印机绘制出来。但是因为中间等待的时间太烦人了，我不得不把所有的子例程都当做并行的任务一次写完。

Seibel：*所以你每写一个子例程，基本上都会有一个单元测试，这样你就能看到这些程序是可以运行的，是这样吗？*

① 1973年英国发表了James Lighthill爵士的报告，该报告认为人工智能的研究即使不是骗局，至少也是庸人自扰，从而使得英国政府取消对该研究的资助。——编者注

Armstrong：是的。接着必须把程序都整合在一起。我不知道这算不算是编程学习。我大学读的是伦敦大学物理系。我们好像从第一年就开始编程了。运行程序的周转时间变成了大约3小时。但是我同样发现最好是每次运行四、五个程序，这样能够相当快地拿回程序了。

Seibel：*在高中的时候，计算机是学校规定的课程吗？*

Armstrong：那门课是利用业余时间上的，类似于计算机俱乐部。我记得我们去参观了那台计算机。很多年纪比我们大的人穿着白大褂走来走去，口袋里插着笔，表情严肃，那就像一座教堂。那台计算机很昂贵。

Seibel：*你学的是物理学，是从什么时候开始转向编程的？*

Armstrong：嗯，有一些本科生的课程需要编写程序，而我又特别喜欢编程。我还非常善于调试程序。如果别人程序出了问题，我就会去调试别人的程序。标准调试的开价是一杯啤酒。也可能提价，还有两杯啤酒、三杯啤酒的问题。

Seibel：*在给他们调试程序时，是以他们必须给你买多少杯啤酒而论的，对吗？*

Armstrong：对，等我修复了程序时他们要给我买啤酒。我在读程序的时候总是在想："他们为什么要这样写程序呢，太复杂了。"我会重新编写并简化程序。看到人们编写复杂的代码我感到很吃惊。我发现有些问题用几行代码就能解决，但是他们要写上几十行。我有点好奇，他们为什么看不到简单的方法呢。我就颇为擅长采取简单的方法。

我真正开始编程是我拿到第一个学位并打算读博士学位的时候。我开始读高能物理博士学位并加入了那里的气泡室[①]小组，他们有一台计算机。那是一台DDP-516，是Honeywell公司的。我可以独自一人使用它。它是穿孔卡式的，但是可以在上面直接运行程序，只要把穿孔卡放进去，按一下按钮，答案刷地一下就出来了。我特别喜欢那台计算机。我在上面编写了一个小象棋程序。

那时的实际磁心存储器是由妇女编织而成的，能够看到磁心，能够看到一块一块的小磁铁和穿进穿出的线路。价格高得惊人。它有一个大约10MB的磁盘驱动器，上面有20个小底板，大约15公斤重。它还配了一个电传文本的界面，可以在上面输入程序。

① 气泡室（bubble chamber）是探测高能带电粒子径迹的一种有效的手段，1952年由美国人D.A.格拉泽发明，曾经为高能物理学创造了许多重大发现的机会。

后来又出现了“玻璃电传打字终端”，那是最早的视频显示器设备，可以在上面输入并编辑程序。我觉得这太神奇了。再也不需要穿孔卡了。我记得当时和计算机管理员说：“要我说，将来有一天人人都会有这样一套机器。”他说道：“我看你疯了，Joe，你真是疯了！”“为什么不可能呢？”“这些东西贵得离谱。”

正是从那时我真正开始学习编程了。当时我的导师对我说：“你不应该再读物理学博士了，改行吧。你热爱计算机，你应当搞计算机。”我说道：“不，不，不。我不能半途而废。”但实际上他的话是对的。

Seibel：那你拿到博士学位了吗？

Armstrong：没有。我没钱了，所以没有读完。我后来去了爱丁堡大学。此前在读物理的时候我们常常到物理系图书馆去学习。在图书馆的角落里有一些计算机科学书籍。有一些棕色封底的杂志叫做《机器智能》，一共有4期，是爱丁堡大学的机器智能系编辑出版的。我学的是物理学，但我却渴望阅读这些杂志，并且在想：“真是太有趣了。”Donald Michie那时担任爱丁堡大学机器智能系的主任，我给他写了封信，说我对这种东西非常感兴趣，问他那里有没有工作可做。他给我回了信，说目前还没有，不过无论如何，他很想和我见一面，看看我是什么样的人。

几个月后我接到一个电话，也可能是一封信，是Michie的。他说：“我周二去伦敦，我们见一面如何？我要乘火车回爱丁堡，你能来车站吗？”我去了车站，见到Michie，他说：“嗯，不能在这儿面试——我们去找个酒吧。”于是我们到了酒吧，我和Michie聊了聊。过了没多久又收到他的一封信，说：“在爱丁堡大学有一份研究工作，你申请一下吧。”于是我成了Donald Michie的研究助理并去了爱丁堡大学。我就是这样从物理学转到了计算机。

Michie在二战期间曾经和图灵在布莱切利公园（Bletchley Park）[①]一起工作过，拿到了图灵所有的论文。我在图灵图书馆有一张书桌，周围也全都是图灵的论文。我在爱丁堡大学待了一年。此后由于数学家James Lighthill的原因，爱丁堡大学都有点维持不下去了。Lighthill受雇于政府，前往爱丁堡大学调查人工智能。他回去后说道：“那个地方什么有商业价值的东西也弄不出来。”

说得就像一个巨大的儿童游戏区。我是英国机器人学会的创始成员，我

① 二战爆发前，英国在距离伦敦不远的布莱切利公园设置了国家密码破译机构，许多破译员在那里工作，破解德国电报密码。——编者注

们都认为这个工作意义重大。但是拨款机构却说：机器人！这是什么东西！我们不打算在这上面投入资金了。我记得那是1972年前后，所有的资金来源都枯竭了，大家都说："嗯，在这里度过的时光非常美好，但现在最好还是找点别的事情做吧。"

于是我又回去从事物理学工作了。我到了瑞典，在EISCAT科学协会得到一份物理学程序员的工作。我的上司来自IBM，年纪比我大，他想要上头给出一份规格说明书，这样他可以拿去实行了。我们曾经讨论过这个问题。他说："如果没有任务说明，也没有规格说明，这样的工作太糟糕了。"我说："嗯，如果没有任务说明，那才是一个好任务。因为你可以按照自己喜欢的方式来完成。"一年后我的上司离职了，我接替了他的工作，担任首席设计师。

我为他们设计了一个系统，可以称为应用操作系统，那是一个在普通操作系统上运行的系统。那个时候计算机的价格已经比较合理了。我们有一些NORD-10计算机，是挪威制造的——我觉得他们这种型号的计算机想要进入PDP-11的市场。

我在那里工作了将近4年。接着在瑞典空间研究中心得到一份工作，构建了另外一个应用操作系统，用于控制瑞典发射的名为海盗的第一颗卫星。那是一个有趣的项目，不过我忘了那台计算机的名字了，只记得它克隆的是Amdahl公司的计算机。那上面还只有行编辑器，没有全屏幕编辑器。所有的程序都只能放到一个目录下面。文件名是10个字母，扩展名是3个字母。还有一个Fortran编译器或汇编语言编译器，全部东西就是这些了。

有趣的是，现在回头想想，我不认为当今这些小玩意会让你的生产率更高。比如说分层文件系统，它怎么可能让生产率更高呢？很多程序开发方式是在脑海中形成的。我认为在那些简单的系统上工作可以强制你规范地进行思考。如果没有目录系统，就只能把所有的文件都放到一个目录下面，你只能变得相当规范。如果没有修订控制系统，你也只能变得相当规范。如果自己做的事情能够规范起来，那我觉得分层文件系统和修订控制系统也就没什么可取之处了。它们解决的问题并不能从本质上解决你的问题。如果多人一起工作，它们可能会让事情容易一些。但对个人来说，我看不出来有什么差别。

另外，我觉得我们现在因为选择过多而不堪重负。我的意思是，那时候我只能使用Fortran。甚至连Shell脚本也没有。只有可以运行程序的批处理文件，编译器，还有就是Fortran。如果确实需要的话，还可能有汇编语言编译器。不需要痛苦地做出选择。今天年轻的程序员肯定会感到很不舒服，面对

20种编程语言和几十种框架，该如何选择，真是无所适从。我们那时没有这些难以选择的地方。只要开始做就行了，因为使用什么语言、什么工具都已经是定下来的。不需要考虑该做些什么，只管做就行了。

Seibel：*另外一个差别是现在无法再彻底了解整个系统了。也就是说不仅仅是要做出很多选择，而且在选择要使用哪些黑盒的时候，还不一定完全理解黑盒的工作方式。*

Armstrong：是啊，如果这些大黑盒不能正常工作，必须做出修改，我觉得自己把所有的内容都从头开始编写一次会更容易些。做不到软件复用，真是太糟糕了。

Seibel：*你不仅仅是Erlang语言的架构师，还是开放电信平台这个框架的架构师。开放电信平台能够复用吗？*

Armstrong：在某种程度上是可以复用的。但是也面临着同样的问题。如果那个框架刚好能够解决你的问题，比如说如果某个程序员对OTP的设计标准一无所知，花了几年的时间了解了OTP并且说："太好了，正是我所需要的。"那没问题，你可以实现这种程度的复用。但是如果情况不是这样，那就有问题了。

前不久我看到有人这样说："这太不自然了，我们要扭曲代码来适应这个OTP框架。"我对他们说："嗯，可以把OTP框架重写一下。"他们觉得自己修改不了框架。但框架不过是另外一个程序而已，修改起来相当容易。我动手做了起来，实现了他们的目的。他们看了以后说："是的，果真很容易。"他们也觉得容易修改。但是又说："项目管理人员不希望我们把时间浪费在框架上。"这好办，把这件事情换个说法就行了。

Seibel：*但是如果把所有这些黑盒都打开，看看里面有什么，看看它们的工作方式，再确定如何对它们做一点改造来满足自己的需要。你觉得这样做确实是可行的吗？*

Armstrong：这些年我犯了一些人们常犯的错误，那就是没有打开黑盒。有时候想一想，觉得这个黑盒无法理解，难度太大，所以不想打开它。我曾经打开过一、两个黑盒。有一次我需要做一个窗口系统，为Erlang做一个图形系统，我在想："嗯，就在X Windows上运行吧。" X Windows是什么呢？它是一个套接字，上面跑着协议。只要打开套接字，往里面注入这些消息就可以了。为什么要用库呢？Erlang是基于消息的。整体指导思想是向其他东

西发出消息，让它们执行操作。嗯，X Windows中的指导思想则是，有一个窗口，向窗口发送消息，再由窗口执行操作。如果在窗口中执行操作，它会把消息回送给你。这非常像Erlang。但是X Windows的编程方式是运用回调库——如果出现了这个情况就调用这个函数。这不是Erlang的思考方式。Erlang的思考方式是，给某个东西发送消息，让它做一些事情。所以，等一下，把其中的库去掉吧，直接和套接字对话。

猜猜结果会怎么样？非常简单。X协议收到了一些消息，我不知道具体是多少条，也许是100条、80条，大致就是这么多。但实际上只需要其中的20条就能完成有用的工作了。把这20条消息映射到Erlang术语上，变个小魔术，然后可以向窗口直接发送消息，它们就开始执行动作了。这样做的效率也很高。但界面不是很好，因为我没有把太多的精力用到图形和艺术标准上。如果为了让界面再美观一些，要做的工作还很多。但是不管怎么说，实际上并不难。

另外一个例子是我做的排版系统，我打开的抽象边界是PostScript。到了边界的地方你会想："我不想越过这个边界。"因为你会认为边界里面的东西极其复杂。但是我再次发现，它实际上是很简单的。那是一种编程语言，一种不错的编程语言。抽象边界很容易穿越，而一旦穿越，会有很多收益。

在出版我那本Erlang编程书时，出版社说："我们有画图工具。"但是画图工具真的很难精确地对准箭头，所以我不喜欢那些工具。而且画图的时候手也很难受。我想："编写一个生成PostScript的程序，然后在'这里画个圆圈、那里画个箭头'，让程序正常运转起来，这样一比，编程花的时间并不长。"编写程序需要几个小时。以所见即所得的方式画图也需要这么长时间。只是自己编写程序还有两个好处。你的手不会难受，而且即使把图形放大1万倍，看到的箭头也是对得整整齐齐的。

我并不是说刚入行的程序员应当把所有这些抽象的东西都打开。我的意思是，一定要考虑是否可以打开它们。不要完全放弃这个想法。看看直接到达的途径是不是比包装后的途径要快一些，这是值得一看的。一般来说，如果购买软件或是使用其他人的软件，一定要充分考虑还需要花很长时间来加工这套软件，因为它和你想要的不完全一样。软件的执行方式有微妙的差别，而这个差别可能需要很长时间来解决。

Seibel：你开始的时候说软件的复用情况"太糟糕了"，但是打开每个黑盒并摆弄一番，这似乎很难算作是向软件复用的方向前进。

Armstrong：我认为缺少复用是面向对象编程语言造成的，而不是函数式语言造成的。因为面向对象编程语言的问题在于总是得到语言运行环境的所有隐含信息。你要的是香蕉，但看到的却是香蕉拿在大猩猩手里，并且后面还有整个丛林。

如果代码具备引用透明性，如果是纯函数，即所有的数据都来自输入参数，所有的东西在离开时都不会留下任何痕迹，那么会达到惊人的复用效果。你可以在这里复用、那里复用，在哪个地方都可以复用。如果想在其他项目中使用，只要把代码剪切、粘贴到新项目中就可以了。

程序员们被哄骗着使用所有这些不同的编程语言，他们被哄骗着不要使用简单的方法把程序连接在一起。Unix的管道机制（A通过管道和B连接，B再通过管道和C连接）很容易把事物连接在一起。程序员们在连接不同的事物时使用的是这种方式吗？不是。他们使用的是API，把不同的事物都链接到同一个内存区，这样做难度极大而且不是跨语言的。如果语言是同一系列的还好，比如说都是命令式语言，是可以这样做的。但是假设一种语言是Prolog，另一种语言是C。这两种语言看待世界的观点、处理内存的方式完全不同。所以没有办法用那种方式把它们链接在一起。无法复用那些东西。对某些人来说肯定存在巨大的经济利益，他们不想让不同的内容很好地在一起工作。这种状况为咨询师创造出大量的工作岗位。还有大量的工具来解决本不应存在的问题。这些问题在多年前就已解决了。

几乎没有几种语言能描述事物之间的交互，我觉得这真是太奇怪了。我一直都在考虑将事物连接在一起的方式和描述协议的方式。我们没有用来描述事物之间的协议的方式：假设我发送给你一个事物，你再发送给我另一个事物，而这就需要一种协议。我们有各种方式来描述包和包的类型，但是用来描述协议的方式却非常有限。

编程和我们在现实世界中构造事物的方式有很大差别。假设你是一名汽车制造商，需要从分包商那里购买零部件。从Lucas公司购买电池，从其他地方购买发电机。要把不同的零部件装配在一起，采取的方式仅仅是把它们一个挨一个放在一起。盖房子的时候，只需把砖头一块块地垒在一起，再把门放在那里。制造芯片也是这种方式。印刷电路板基本上都能提供这种连接。但是你可以把制作电子元件的过程看作是先购买所有这些芯片，然后用线把芯片的脚连接在一起。这就是制造硬件的方式。但软件并不是这样开发出来的。我们应当采取那种方式开发软件，但却没有那样做。

我们没有那样做的原因是因为并发性。你看到了，在把芯片挨个放在那

里的时候，它们都是并行地执行的。它们会发送消息。它们依据的是编程中的消息传递机制，这种机制正是我所信赖的。但我们把软件编写在一起的方式并不是那样的。我认为Erlang可以采取的一个方向，或者说我愿意采取的一个方向，就是这种组件的方向。我还没有实现，但是我想做一些图形化的前端程序来开发这些组件，而生产软件的方式就是把这些组件连接在一起。数据流编程完全是说明性的，不存在顺序状态这样的概念，不存在程序计数器翻转，就只有那些东西。那是一个说明性的模型，很容易理解。在大多数编程语言中都看不到这些。

这并不是说黑盒内部不复杂。比如说grep。假设有一个小方框，我们从外面来看，输入的是数据流、文件。输入cat foo | grep，grep得到一些参数，这是一个需要与之匹配的正则表达式。所有输入就是这些。在grep外面是所有与那个正则表达式匹配的行。从感性的层次来看，非常容易理解grep做了什么。它的输入是一个文件，或者它的输入是一个正则表达式。它的输出是与正则表达式相匹配的一组行或一连串的行。但这并不是说黑盒内部的算法简单，实际上可能极为复杂。

黑盒中发生的事情极为复杂。但是把这些复杂的组件中的东西组合在一起，这个过程并不一定复杂。grep的使用一点儿都不复杂。但是在系统架构中看到的情况却是，在把这些东西组合在一起的过程与黑盒内容的复杂性之间，没有一个清晰的界限。

通过编程语言API进行连接时，我们并未触及黑盒的抽象。我们把它们放到同一个内存区。如果grep这个模块暴露出其API中的子例程，你向它传递一个char*指针，必须对它做malloc，需要深复制这个字符串，这时你能够创建并行过程来完成这件事情吗？如果这样做，会变得很难理解。我不明白人们为什么会使用这种复杂的连接方式。他们应当使用简单的连接方式。

Seibel：*同刚开始编程时相比，你在看待该如何编程的问题上最大的变化是什么？*

Armstrong：我认为编程方式中的最大变化与硬件无关。显然，现在的计算机速度要快得多，功能要强大得多，但是人的大脑比最好的软件工具还要强大一百万倍。我在编写程序的时候，几天之后会突然说："程序中有一个错误——如果这样、那样、那样、这样的话，程序就要崩溃了。"然后我去看了代码，确实如此。此前一点征兆也没有。你能告诉我哪一个开发系统能够做到这一点吗？作为一个程序员，我所发生的变化是内心思想的变化。

我认为在经过多年的编程之后会有两个变化。一个变化是，在年轻的时候，我会不停地写程序，直到完成。当程序完成后，我就不再管它了。程序写好了，完工了。然后我会突然领悟："啊！搞错了！真是笨蛋！"我会重新编写程序，后来再次发现："噢，程序是错的。"于是又重新编写。

我记得当时有这样一个想法："先不要动手写代码，把这些东西都想好，这样做不是很好吗？"如果我不写代码就能获得那番领悟，不是很好吗？我认为现在可以做到这一点了。那20年可以算作是学习如何编程的时期。现在知道该如何编程了。我以前通过实验来学习编程。现在我知道该如何编程了，不需要再做实验了。

偶尔，我也得做一些很小的实验，比如编写一些非常小的程序来回答某个问题。我会把事情想清楚，等到开始编程的时候，这些程序就可以或多或少地像我预计的那样运行起来了，因为之前我已经想清楚了。这也意味着要花很长时间。编写程序、有所醒悟、重新编写。这样可能需要花上一年的时间来写程序。所以我现在可能不这样做，而是先思考上一年。我不会再做那种简单的输入工作。

这是第一个变化。出现的第二个变化是直觉。在年轻的时候，我会通宵地写程序，干到凌晨4点钟，精疲力尽，那是男子汉气概的编程，一个小时接着一个小时，不停地编写代码。即使情况不好我也坚持不懈，总要让代码能够跑起来。即使没有直觉，我也要继续编程。

我得到的教训是，在疲惫的时候编写的程序都是垃圾，第二天就要把它们都扔掉了。20年前，就算强烈地感到事情不对劲、代码中有错误时，我也会继续编程。这些年来我注意到，真正好的代码是我完全进入状态的时候编写的，时间不知不觉地过了，而我甚至没有在考虑程序，只是很放松地坐在那里，输入这些东西，看着自己输入的东西出现在屏幕上。这样的代码会很不错。如果你不能集中注意力，弄出来的东西会说："不行，不行，这儿错了，那儿也错了。"可我在多年前并没有注意到这一点。写出来的代码都被扔掉了。现在，如果觉得不行，我就不再编程了。"不能再写了。"这是我根据经验得到的，停下来，不要再写代码了。不要再处理这个问题了。干点别的。

我在上学的时候很擅长数学之类的课程，所以在想："噢，我是一个按照逻辑思考的人。"但是我参加心理测试时，在直觉上得了高分，而逻辑思考方面的分数却有点低。不是很低，我还是可以做数学这类的题目，我相当擅长。但正是因为我擅长数学，所以我过去认为科学是关于逻辑和数学的。

我现在就不会这样说了。我要说科学也有很多直觉，根据直觉能够知道什么是正确的。

Seibel：*你现在在编码之前会花更长的时间思考，那么在思考阶段会做些什么呢？*

Armstrong：噢，我会记些笔记，我不仅仅是在思考。在纸上随便写点什么。我可能不会写很多代码。如果你密切注意我的活动，会发现我大部分时间都在思考，偶尔写点什么。另外一件对解决问题非常重要的事情是问问我的同事："你将如何解决这个问题？"你找到他们，说："我不知道应当采取这种方式还是那种方式。必须在A和B之间做出选择。"然后你向他们描述A和B，等讲到一半的时候，你会说："啊，是B。谢谢你们。非常感谢。"这样的事情发生过很多次。

你需要这样一块智能白板，如果你只是独自一人在一块白板上写写画画，是得不到反馈的。但是如果面对的是人，你会在白板上向他们解释替代方案，他们也会加入讨论，提出一点建议。然后突然间你就知道答案是什么了。对我来说没有涉及到代码编写。但是和处理同样问题的同事进行交谈是非常有价值的。

Seibel：*你觉得是因为他们给出了一点反馈或者提出了一点问题？还是说你只要给别人解释就够了？*

Armstrong：我想是因为必须把想法从解决问题的那部分大脑转移到进行表达的那部分大脑，这两部分大脑是不同的。我认为是因为必须强制进行这种转移。我从来没有做过一个人在空房间说话的实验。

Seibel：*我听说有这样一个计算机科学系，在教师的办公室中放着一个毛线玩具，那里的规矩是在打扰教师之前必须先把问题向毛线玩具解释一番。"好吧，大熊先生，我的问题是这样的，我的方法是——啊哈！我知道答案是什么了。"*

Armstrong：是吗？那我可要试试。

Seibel：*可以跟你的那几只猫交谈。*

Armstrong：和猫交谈——好主意！有一个和我一起工作的人，年纪比我稍微大一点，非常聪明。我每次到他办公室去问他问题时，他对每个问题都会说："程序是个黑盒。有输入、有输出。在输入和输出之间有函数上的联系。

问题的输入是什么？问题的输出是什么？在输入输出之间的函数联系是什么？”在这样的谈话进行到某个时刻，我会说：“你太聪明了！”然后就冲出房门。他吃惊地摇着头说：“我还不知道他要问什么问题呢，他一直都没有说。”他就像是玩具熊一样，听我解释问题。

Seibel：*你说的随手写一些东西，是编写一些小代码段呢，还是说随手画一些图形呢？*

Armstrong：更多是一些气泡图和箭头。你在白板上向人们解释一些东西的时候，会画一些气泡图、箭头、等式、注释。没有代码。只有一些代码片段，很少的一点点代码，因为这是表达想法比较简洁的方式。现在还处于思考阶段。我不知道执行这些操作需要花多长时间，所以偶尔会写一点代码，做做实验。我会写上十来行代码，看看执行需要多长时间。

Seibel：*你的意思是看看在计算机上执行那些代码需要花多长时间？*

Armstrong：是的。需要一毫秒还是一微秒，我不知道。我可以猜测一下，但是需要验证猜测。我只会看我没有真正理解的地方。但是我编写Erlang程序的经验极为丰富，所以很清楚是怎么回事。解决问题的方法和几年前是一样的。也就是，识别难点，写一些小的原型，识别不确定的地方，写很少的一点代码。我现在做的基本上也是这些事情。但是如果是Erlang的话，就没有必要再去做这些小实验了。如果是Ruby或Java，我不知道会出现什么情况，所以就不得不采取原来的方法，做很多实验。

Seibel：*在思考过程中的某个时刻，你就知道该如何写代码了？*

Armstrong：是的，然后把零散的部分合在一起。但是也许没有办法向其他人解释。我只是强烈地感觉到，现在可以开始写程序了。我并不知道解决方案是什么。就像母鸡做好准备要下蛋了，我现在也准备好下蛋了。

Seibel：*在那个时刻，你需要进入状态，不希望被打断。*

Armstrong：没错，是这样。

Seibel：*到了代码级别，仍旧有很多细节的东西需要弄清楚，需要全神贯注。*

Armstrong：噢，是的。但是分两种情况。真正需要全神贯注的是那些不能够自动出来的东西——你必须要思考。假设说遇到很难对付的垃圾回收，涉及需要哪些标记、在哪里标记，这一定要好好考虑一下。你的思维已经进入这个问题，知道自己肯定会找到解决方案。知道方案就在那个小黑盒中。

米开朗基罗在西斯廷教堂等地方作画的时候，有一群画家在帮助他。他首先画出大图。一片片巨大的区域必须涂成蓝色和绿色的。与写程序非常类似。第一个草图是一个很大的草图，每个东西都要放到合适的位置。部分区域将用单一的颜色填充，填充的速度相当快，因为这不需要思考。

然后开始画眼睛的细节，不过那种地方不好处理。你知道可以开始画了。因为草图已经画好，眼睛的位置也定下来了。只要画出眼睛和细节就可以了。这并不是说容易画，这实际上是个难点。在画眼睛的时候需要全神贯注。额头、脸颊基本上是千篇一律的，在画的时候不需要全神贯注。这里要画点胡须茬，要集中一半的注意力。

输入代码，遇到句法错误，运行几个小测试，确保代码可以工作。这些都相当轻松。遇到小的编译错误时修复一下。等到对某种语言有了经验后，甚至连诊断信息也不用看了。只要知道行号就可以了，不需要知道具体说些什么。是这一行？是的，这行错了，重新输一下。

我在芝加哥讲授过一次Erlang的课程。我在教室中四处走了走，注意到学生们有一些不对的地方。噢，这里少了逗号，在这个操作之前执行那个操作会崩溃掉，你没有链接起来。我妻子很善于校对，她说错误会从页面中跳到你的眼前。漏掉一个逗号或是拼写错误，这些问题都从页面中跳到她的眼前。如果四处走走，看看别人的代码，编程错误也会从页面中跳出来。感觉不是有意去思考这些问题，而是一种全面思考的结果。在屏幕上看到所有的东西，发现那里有一个错误，`bumpf`。只需要修正这些表面的错误就可以了。

一个很难处理的问题是变量名中细微的拼写错误。所以我会特意选择一些差别很大的变量名，这样就不会出现这种错误了。如果一个长变量名为`personName`，在后面添加一个`s`，得到`personNames`，用来表示一组人，对于这种命名方式，我的眼睛很容易把它们混淆。所以我会用`personName`表示一个人，用`listOfPeople`表示一组人。我是特意这样做的，这样就容易看出我写得对不对。但是标点符号是能看出来的，能够看到逗号和括号出错。当然，Emacs会把每个符号都标出颜色并自动缩进，括号的颜色也是不同的。所以这个相当简单。

Seibel：*在开始编码的时候，你是自顶向下、自底向上还是从中间开始？*

Armstrong：自底向上。我写一点就测试一点，再写一点就再测试一点。我现在的做法变成了先写测试用例。单元测试。写好测试用例，再写代码。我可以很自信地说这种方式是可行的。

Seibel：再回头看看你过去的经历，你在离开瑞典空间研究中心之后就进入到了爱立信研究实验室？

Armstrong：是的。那个时候非常、非常幸运，我记得那是1984年，我是在实验室成立两年后进去的。我们那时非常乐观。我们的世界观是，解决问题、把它们用于实际项目，提高爱立信的生产率。那时这种世界观还没有与现实世界接触，没有受到现实世界的一点影响。所以我们认为很容易就能发现一些新奇有用的东西，而且认为在发现这些新奇有用的东西后，世界会张开双臂欢迎我们。但是后来认识到，发现一些新东西并不是件容易的事情。而且让人们使用新的、更好的东西会非常难。

Seibel：Erlang就是一种你希望他们使用的新的、有用的东西？

Armstrong：是的，那是肯定的。情况是这样的，开始我们只有Prolog。我做了一个小语言，人们开始使用它。Robert Virding过来说："嗨，这个语言看上去很有意思。"他读了我的Prolog程序后说道："我能改点东西吗？"这样很危险，因为Robert虽然说是改一点东西，在程序开始的地方也只写着一句注释："Joe想到了这个东西，我稍微修改了一下。"但最后你会发现程序被改得面目全非了。我和Robert反反复复地重新编写了程序，我们有很多争论："啊，我没有办法读你的代码，所有的逗号后面都有空格。"

后来我们发现爱立信内部有些人想得到一种新的编程语言，或是想要一种更好的方式来编写电话程序。我们和这些人每周碰一次头，一直持续了6到9个月的样子，我记不太清楚了。我们的想法是我们教他们如何编程，他们教我们电话方面的知识，从而告诉我们要解决的问题是什么。我记得那段经历既让人沮丧，又让人兴奋。那段经历改变了这个语言，因为有人在真正地使用它。那也促成了一项研究，因为他们会想："这种语言不错，但是速度太慢了。"他们测量了程序的性能并说："速度应当再快70倍。"我们说道："这个阶段的工作到此为止。我们将把速度提高70倍，他们继续使用它编程，我们必须在两年左右的时间把速度提高70倍。"

我们开始的时候犯了几个错误。我们有过几次让人非常尴尬的时刻。其中一个应该避免的严重错误是：还未实现就告诉别人程序会有多么快。不过到最后我们还是搞清楚了如何实现。我用Prolog写了一个编译器，Rob做了库和其他东西。时间已经过了快两年了。我想可以用C语言来实现这个抽象机了，于是开始编写了我的第一个C语言程序。Mike Williams过来看到我的C程序说："这是我这辈子见过的最糟糕的C程序。真是太糟糕了。"我觉得

并没有那么糟糕，但是Mike不喜欢。于是由Mike用C语言编写虚拟机，我用Prolog编写编译器。然后编译器编译了自身，产生字节代码，放到虚拟机中，我们又修改了语法和句法，用这个编译器编译了自身，随后便产生了可以引导的镜像，我们取得了飞跃。我们做的东西不再基于Prolog，而是变成了一种语言。

Seibel：*你们是否发现某些东西很难放到Erlang模型中？*

Armstrong：是的。我们完全没有考虑内存。将JPEG图像转换为位图数据要精确地依赖于数据的布局，在Erlang中做得并不好。依赖于破坏性升级状态的算法，在Erlang中做得不好。

Seibel：*这样，如果要编写一个大型的图像处理工作流系统时，你会用其他语言来编写实际图像转换程序吗？*

Armstrong：我会用C、汇编或其他语言来写。我也可能用一种Erlang的方言来写，然后将Erlang程序交叉编译为C。做一种方言，就是做一种与问题领域相关的方言。我也可能编写一个产生C程序的Erlang程序，而不是手工编写C程序。但是目标语言将是C、汇编或其他语言。至于是手工编写还是生成它们，这倒是个有趣的问题。自动生成C会容易些，所以我倾向于自动生成C而不是手工编写。

但是我会使用Erlang的结构。我找到一些程序来处理我们家的照片和其他东西。我使用了ImageMagik和一些shell脚本。但所有程序都是从Erlang进行控制的。我只是写了一个封装器并调用os:command，然后调用ImageMagik命令。把程序都封装进去是相当不错的。我不想在Erlang中做实际的图像处理。如果在Erlang中写这些代码就太笨了。C要好得多。

Seibel：*而且，ImageMagik是已经写好了的。*

Armstrong：这对我的影响倒是不大。我想如果我是做OCaml语言开发的，我就会用OCaml写一个程序，因为OCaml的效率很高。但是Erlang不行。所以如果我是OCaml程序员，我会说："我该怎么办了？把ImageMagik重新实现一次？好，马上开始。"

Seibel：*只是为了乐趣？*

Armstrong：我喜欢编程。为什么会不喜欢呢？虽然我总是说Erlang不善于图像处理，但实际上我从来没有试过。我觉得Erlang不善于，但这种感觉可能是错的。应当试一试。嗯，会很有意思。你就别再怂恿我了。

真正优秀的程序员会在编程上投入大量时间。我从来没有看到过哪一个优秀的程序员不愿意在编程上投入大量时间的。如果两、三天不编程，我就会受不了。你会越来越熟练，编程速度会越来越快。编写那些代码的另外一个作用是，在遇到一些普通问题的时候能够非常迅速地解决。

Seibel：*你做过什么特别的事情来提高自己的编程技能吗?*

Armstrong：没有。我认为没有必要。我会学习一些新的编程语言，但不是为了成为更好的程序员。我的目的也许是为了成为更好的语言设计师。

我喜欢分析事物的工作原理。一个很好的验证方式就是由你自己去实现它们。对我来说，编程并不仅仅是往机器里输入代码。编程的目的在于理解。我喜欢理解事物。我为什么要实现刚才说到过的JPEG呢？因为我想理解小波变换。编程是理解小波变换的一个工具。我又为什么要试着给X Windows做一个接口呢？因为我想理解X协议的工作原理。

这是一种实现某个事情的推动力，我强烈推荐。如果想理解C，就写一个C编译器。如果想了解Lisp，就写一个Lisp编译器或Lisp解释器。有人听到我的话后说："啊，写个编译器真是太难了。"其实不难，相当简单。有很多小东西需要掌握，但是都不难。需要知道数据结构。需要知道散列表。需要知道语法分析。需要知道代码生成。需要知道解释技术。这些知识的每一样都不是很难。我想初学者会觉得编译器很大、很复杂，所以不想动手做。你不去做的事情都很难，做过的事情都很简单。但是人们连试都不去试一下。我觉得这种做法是不对的。

Seibel：*和我交谈过的几个人建议学习几种不同的编程语言，因为这样可以让你站在几个不同的角度去解决问题。*

Armstrong：要学就学能够做不同事情的语言。学习很多做同样事情的语言是没有意义的。当然，我编写过很多JavaScript程序、很多Tcl程序、很多C程序和很多Prolog程序（实际上是大量的Prolog程序），还有大量的Fortran程序和大量的Erlang程序。还有一点Ruby程序，一点Haskell程序。所有的语言我基本都能够读懂，但并不是对所有这些语言的编程都擅长。当然我能够使用的编程语言也很多。

Seibel：*不包括C++?*

Armstrong：不包括。说起C++，我基本上看不懂也不会写。我不喜欢C++，它让人觉得很别扭。太复杂了。我喜欢小的、简单的语言。C++让人觉得既

不小、也不简单。

Seibel：*哪些语言影响了Erlang的设计？*

Armstrong：Prolog。显然，Erlang是从Prolog上发展起来的。

Seibel：*现在在Erlang中已经看不到太多Prolog的痕迹了。*

Armstrong：嗯，合一（unification），也就是模式匹配，是直接从Prolog中拿过来的。还有数据结构一类的。元组和列表与Prolog中的句法稍有差别，但也是来自Prolog。还受到了Tony Hoare的CSP（Communicating Sequential Processes，通信顺序进程）的影响。我还读了Dijkstra的保护命令。这也是我为什么总是要求模式匹配的原因，不应当有默认情况，需要明确地要求某些分支必须匹配。我认为主要影响是这些。

Seibel：*函数式这方面的内容是从哪里得到的呢？*

Armstrong：在把并发性加到Prolog中以后，就一定要确保做完某件事情后，它不会再回溯。在Prolog中，可以调用一些东西，在对解决方案进行回溯后，基本上可以恢复到调用前的状态。这样在看到这句话的时候就要注意了："发射导弹。"导弹一下子就发射出去了，不能让导弹原路返回并逆转这个过程。纯Prolog语言是可以逆转的。但是当与现实世界打交道时，所做的事情都是单向的。假如说要发射导弹，导弹就发射出去了。假如说："把交通信号灯从红灯切换到绿灯，"他们就把红灯切换到绿灯了，你此时不能说："这个决定可不好，取消这个决定。"

我们得到的是一门并发语言和一些并行的进程，在这些进程中我们使用了完整的Prolog，它可以回溯等。这样，Prolog就变得必须要全面裁剪一下，以避免回溯。

Seibel：*在裁剪的地方，那些不可逆转的事项要把消息发送给其他进程？*

Armstrong：是的。但那只是一个函数调用，也许不是发射火箭的函数，只是调用其他函数，那个函数再调用其他函数，要把这两个世界完全分开，是一个很痛苦的尝试。所以在过程内写的代码变得越来越函数化了，有点像是Prolog的方言，是一个函数化的子集。虽说是一个函数化的子集，不如让它完全函数化。

Seibel：*不过Erlang是动态类型的，和当今大多数函数式语言都有一定的差异。你觉得它是不是可以算作函数式语言社区的一分子？*

Armstrong：噢，是的。当我们去参加函数式编程大会的时候，我估计我们会就其中的差异进行争论。我们争论及早求值和惰性求值。我们争论动态类型系统和静态类型系统。但是尽管存在这些争论，函数式编程的核心却是非可变状态（nonmutable state）——x不是内存中的位置名，它是一个值。是不能改变的。比如说x等于3，之后就不能再改变了。所有这些不同的社区都说这一点对于理解程序、让程序并行、调试程序是非常有好处的。后来又出现了像Erlang这样动态类型系统的函数式语言，还有静态类型系统的函数式语言，这两种类型都有各自的优缺点。

能在Erlang中获得静态类型的优点是非常好的事情。也许在某些地方可以对程序进行批注，让类型更明显，这样编译器就可以导出类型并生成更好的代码了。

那些搞静态类型的人会说："嗯，我们在封送（marshal）数据结构时非常喜欢动态类型的好处。"我们需要知道类型，所以不能把一个任意的程序发出去，到另一端再重新构造。我们有一个Cardelli称为永久不一致的系统。还有一些一直在增长、变化的系统，系统中各部分可能出现短暂的不一致。而且当修改系统中的代码时，它不再是原子的了。有些节点变了，有些不变。它们相互交谈，等到了某个时刻它们就一致了。而在其他时候，比如在检查交流边界的时候，我们一定要相信边界是正确的吗？边界可能是虚假的。所以需要检查一下。

Seibel：*你很早以前就通过调试别人的程序来换啤酒。你为什么认为自己擅长调试呢？*

Armstrong：嗯，我喜欢调试。在程序的某个时刻把变量和其他东西打印出来，看看进展情况，它们都在按照你的预想进展。在这个地方程序是对的，而到了后面某个地方它又错了。因此要在中间阶段看看问题，它或者是对的，或者是错的，只需要采取这样的区间二分法就可以了。假设你会重现错误时是可以这样做的。那些不可重现的错误相当难调试。但是他们没有给我那样的缺陷。他们给我的都是会重现的错误。所以只要采用二分法就可以了，直到找到为止。最后肯定能找到。

Seibel：*在这方面擅长，你认为是因为自己的观点更系统吗？*

Armstrong：是的，他们放弃了。我不知道是什么原因。我真是不理解他们为什么不去调试程序呢。我的意思是，你认为调试很难吗？我不这样认为。只需要停下来，放慢一点就可以了。我指的是批处理的Fortran。

我再来说说实时系统和垃圾收集器的调试。我记得有一次Erlang崩溃了，那是早期的事情了，Erlang一启动就崩溃了。我只是输入了一些东西。它在shell中内置了某种Emacsy命令。我输入erl，启动它，进到“读取—求值—打印”循环中。接着，我输入了四、五个字符，出现了拼写错误。我把光标回退了几次，改正了拼写错误，结果系统因为一个垃圾回收错误而崩溃了。我知道这是一个很严重、很严重的错误。我在想：“我能准确回忆起刚才输入的什么字符吗？”一共也就是大约12个字符。我重新启动了Erlang，输入了一些字符，可系统没有崩溃。然后我坐在那里弄了大概一个半小时，试了估计有一百来次不同的内容。系统终于再次崩溃了！我把它记了下来。总算可以调试了。

Seibel：你使用的技术是什么？是打印语句吗？

Armstrong：是打印语句。编程之神说过：“应当在程序中你认为可能出错的地方放上一些`printf`语句，重新编译并运行。”

还有“Joe调试定律”，不过我不知道这是我从其他地方看来的还是我自己发明的。我这个定律说的是，所有的错误都出现在上次程序改动的地方前后3条语句中。我在瑞典空间研究中心工作的时候，我的上司是搞硬件的。当时我们在Esrange航天中心工作，那是一个位于北部的火箭发射场和卫星跟踪站。有一次他焦头烂额地在调试硬件中的一个缺陷，插上示波器，把零件换来换去的。我说道：“噢，我能帮上忙吗？”他说道：“不行，Joe，你帮不上忙，这是硬件问题。”我说道：“是的，但是问题肯定和软件类似。这个缺陷一定靠近你上次改动的地方。”他说道：“我换了一个电容。你真是天才！”原来他换了一个大一点的电容。他取下了这个大电容，把原来那个电容又了换回去，问题这下解决了。这个道理在哪儿都行得通。你修了车，然后车就出现故障，这正是你上次改动所造成的。你改过一些东西，所以你必须记住改的是什么地方。万事万物都是这个理。

Seibel：你曾经证实过自己程序的正确性吗？你对那种形式主义感兴趣吗？

Armstrong：可以说证实过，也可以说没有证实过。我用代数的方式操作过程序，为的是证明它们是相等的。但是还没有真正进行那样的定理证明。我采取过指称语义等类似的方法。不过我记得自己最后放弃了。我们的练习是这样的：`let x=3 in let y = 4 in x plus y`，等式foo采用的是及早求值模式，等式bar采用的是惰性求值模式，请证明这两种模式求值的结果都是7。

引理、论点一共写了14页，我后来想："还是算了吧——x等于3，y等于4，x加上y，肯定是7。"那时我正在编写Erlang编译器。为了证明3加上4等于7都要写上很多页，如果要证明我的编译器在各方面都正确，那还不得写上几千几万页？

Seibel：你喜欢独自工作还是团队工作？

Armstrong：我喜欢团队的工作场所，你懂我的意思吧。我不反对社交。但是我喜欢独自编程。当然，就讨论问题而言，我喜欢和人们一起合作。利用工作时喝咖啡的那段休息时间，你在上班路上想的所有问题都变得清晰起来，我觉得这种方式非常有意义。可以获得很多深刻的理解。在众人面前把你的想法好好讨论一下，这种做法很好。采取这种方式，你让自己必须解释自己的想法，对我来说，是把想法从大脑的一部分转移到另一部分。常常地，在解释的过程中这种理解也加深了。

Seibel：你做过结对编程吗——和另外一个人一起坐在一台计算机前面生成代码？

Armstrong：做过。和Robert Virding。当我们两个人都感到有点像是在黑暗中挣扎时，一般会做结对编程。因为我们就不知道自己在做些什么。所以，当你不知道自己在做什么的时候，我觉得这时和另外一个处于同样状态的人一起工作是很有帮助的。如果其中一个程序员比另外一个出色，对于能力稍差的或是经验较少的那个程序员来说，观察别人如何工作可能是有好处的。他们能够从这个过程中有所收获。但是如果两个人的差距过大，就学不到什么东西了，两个人坐在那里都会觉得很无趣。如果和一个跟我水平差不多的人做结对编程，并且我们都不知道自己在做些什么，这个时候做结对编程就比较有意思了。

还会遇到一些在我看来比较特殊的问题。如果感冒了或是觉得身体不舒服，我就不会去尝试解决这些问题了。如果我知道这需要3天的时间来写代码，我就会抽出一天的时间来规划规划，不阅读电子邮件，埋头行动，一干就是整整4个小时。我会在家里做，这样就不会被打扰了。我就是想完成它，想让自己处于这种全神贯注的状态。我认为此时结对编程不会有什么帮助。反而是破坏性的。

Seibel：能举一个这类特殊问题的例子吗？

Armstrong：比如说分析垃圾收集器——那是命令式编程，因而一定要记得

标记所有那些寄存器。再比如说在编译器中实现lambda提升，相当难。这要对所有变量重新标记，然后会得到4、5层的抽象数据类型，看上去都乱糟糟的，以及里面充满各种东西的帧，你会想：“我要完全理解，要深入思考。”此时需要全神贯注。

我会根据自己的状态来安排不同的任务。有时候灵感匮乏，我会对自己说：“嗯，现在应当去骚扰谁？”或者我会阅读一些电子邮件。另外一些时候，我感觉可以做一些有难度的编码工作，那是因为我正在状态。你必须状态合适才能去编码。所以这时候两个人在一起怎么工作呢？其中一个人不在状态，只想看看电子邮件或是做点其他事情。

Seibel：*当你每次和Robert Virding来回传递代码并重新编写的时候，你们所做的工作可以看成是串行结对编程。*

Armstrong：是的。一次一个程序。我一般会花上两三个星期写程序，然后说：“Robert，我已经弄好了，现在看你的了。”然后他就接手做了。我们每次结对编程的时候，拿回来的程序都有点让人认不出来了。他会做很多更改，拿给我的时候，我又会做很多更改。

Seibel：*都是些富有成效的更改吧？*

Armstrong：噢，当然是啦。如果他能找到一些更好的方式，我会很高兴的。我们两个人合作得很融洽。他常常高度概括。我记得我曾经发现这样一个变量。我在45个子例程中一个一个地找，最后发现，这个变量根本就没有使用。他只是在45个不同的函数中把这个变量传进传出。我说道：“这个变量是干什么用的？并没有用到啊。”他说：“我知道。是留作以后扩展时使用的。”我于是把那个变量删掉了。

我会写一个特定算法，删除所有在这个程序中不会用到的东西。每次拿到程序时，程序都会越改越小、越改越具体。而每次Robert修改我的程序，都会改大，会增加通用性。我相信Unix哲学，程序应只做它应当做的，不应做其他事情。Robert的观点是它应当是通用的程序，而程序本身应当是这种通用程序的特定案例。所以他会增加通用性，然后再把它具体化。

Seibel：*看上去像是很大的哲学分歧。把程序按照这两个极端修改一次，有什么好处吗？*

Armstrong：是的。每一轮下来都有改进。我想正是因为这个原因代码质量好多了。可能比我们两人单独弄自己的都要好。

Seibel：*你能说说你是如何设计软件的？能以OTP这样的程序作为例子讲讲吗？*

Armstrong：OTP是由我、Martin Björklund和Magnus Fröberg设计的。最初的设计者只有我们3个人。我们每天早晨在喝咖啡的时候见面并长时间地交谈，大概是一两个小时。在白板上写满了东西。我会记很多笔记。然后我会马上编制所有的文档，他们写所有的代码。有时候我也会写一点代码。当我写文档的时候我可能会发现没有办法描述，必须修改内容。或者他们在遇到我的时候会说："不行，这样是行不通的，我们早上想到的这个主意行不通，因为这样、这样、那样、那样的原因。"到了一天结束的时候，我们或者是得到了所有的文档和所有的代码，或者是得到足够使用的代码和文档。这样我们才觉得这一天没有白过。

有些日子取得不了什么进展，我们会说："明天再弄吧。"一天之内没有足够的时间来做第二遍了。但是一天过一遍是可以的。因为我们在上午可以有大约两个小时讨论，两个小时写文档或是编码。如果能够花上4个小时认真地思考，这一天的工作就已经不错了。那种方式非常、非常好。我不知道我们这种工作方式持续了多长时间。大概是10个或12个星期。我们得到了基本框架，然后招了更多的人。我们定下了架构，现在可以让它成长了。我们又招了3、4个程序员。

Seibel：*你们是如何为新加入的人分割工作的？*

Armstrong：嗯，我们知道什么是原型，什么是最终的版本。我考虑的总是系统设计，首先解决最复杂的问题，识别并解决复杂的问题。然后是容易的问题，你知道容易的问题自然会得到解决的。把问题归为简单的还是复杂的，这是需要一点经验的。我知道IP故障转移或类似的工作会相当难，但分析配置文件的语法相当容易。在原型中可能只是一个需要读取的配置文件。不需要检查它的句法——没有语法检查程序。在产品版本中你可以使用XML，并有一整套的语法检查程序验证它。但你知道做这些事情的步骤是很机械的。需要花费一名称职的程序员几个星期甚至更长的时间。不过这样做是可行的，花费的时间是可以预测的，不会遇到让人恼火的意外事情。但是让通信协议正确，当发生事故的时候能够让它们正常工作，这些事情我只会安排一小部分人来做。

Seibel：*在这种情况中你是先写文档，还是至少在写代码的同时写文档。你一向都是这样做的吗？*

Armstrong：这取决于问题的复杂程序。如果问题很复杂，我通常是先写文档。问题越复杂，我就越可能先写文档。

我喜欢文档编制。我认为除非写出来的文档也过得去，否则程序就不算完成。我也很喜欢规格说明书。有些人会说："要它干什么？看代码就够了。"我觉得这种说法不专业。代码只是告诉你它做了些什么，没有告诉你它应当做些什么。代码是对问题的解答，如果你没有规范或是文档，你就只能猜测代码要解决的是什么问题，而你有可能猜错。我需要知道要解决的到底是什么问题。

Seibel：*你在这个阶段写的是供其他程序员阅读的内部文档，还是用户使用的文档？*

Armstrong：是为用户写的指南。它在某种程度上让我转换到了另外一种思维方式。为了写文档，首先创建一个目录，把文件都放到那里，改一下名字，梳理一下文档结构。我考虑过这方面的问题。我敢保证Knuth会说："嗯，所有的程序都是文学编程。"你不是先写代码，再写文档。你是两个一起写，所以就成了文学编程。我不太赞成那种说法。我不认为是那样。我不知道他是不是因为要出版程序，所以才这么说。

不知道这是左脑右脑之间的转换还是其他什么问题，反正在写文档的时候，对程序的思考方式与写代码时的思考方式是不一样的。所以我估计编写文学程序的时候，可以强制你进行那种转换。这可能是非常有成效的。我也做过一些Erlang的文学编程，但那是很久以前的事情了。那是一个有趣的想法，也许我应当重新做做，用文学化Erlang写一些程序。我不反对这种观点，只是我没有那么大的耐心，想写的是代码而不是文档。但是如果你想真正理解的话，我认为写文档是一个必不可少的基本步骤。

如果使用Haskell语言编程，我就不得不相当早地考虑类型并用文档记录下来。如果使用Lisp或Erlang编程，则不用考虑数据类型就可以开始编写程序了。可以这么说，编写文档是思考数据类型的一种方式。我想会以"……是……"开始。你会说："旋律是由一系列音符组成的。"是的。旋律是一系列的和音，每个和音由时长相等的平行组合音符组成。在文档中定义术语，也就是说什么东西是什么东西，这也是在做某种类型分析，也是在用说明性的方式思考数据结构是什么。

Seibel：*你觉得编程语言整体变得更好了吗？我们是否已经走上正轨，是否从过去学到了足够的经验并提出了足够新的想法？*

Armstrong：是的，新出现的语言很好。像Haskell这样的语言。还有Erlang。有一些有趣的语言，真是应该用起来。Prolog是一种美丽的语言，但是没有得到广泛使用。它有点尖端。Kowalski说这个解决方案还在等待着问题的出现。

Seibel：*Dan Ingalls曾说，Prolog所体现的观点就是，既然摩尔定律已经存在了几十年，那我们真的有必要故地重游。*

Armstrong：Prolog不同于其他任何一种语言。它的思考方式令人称奇。它不适合于所有的问题，但适合于特别大的问题集。它没有得到广泛使用。因为用Prolog编写的程序都短得让人不敢相信，这会让人感到莫大的耻辱。我在编写第一个Prolog程序时感到很吃惊。那种体会让人震惊。你四处看看，程序在哪里呢，我没有写程序。只是告诉系统几个事实，关于问题的事实。然后它就解决问题了。太神奇了。我应该返回去使用Prolog，放弃Erlang。

Seibel：*有没有其他和编程没有直接关系的技巧让你觉得提高了自己的编程水平，或者说对于程序员来说是有价值的？*

Armstrong：写作。有一位计算机科学家说过："噢，如果你不擅长英语，就永远都不可能成为一个优秀的程序员。"

Seibel：*我想是Dijkstra说的。*

Armstrong：偶尔有些大学教授会向我征求意见，问我计算机科学课程的教学大纲该如何设置，问我在这个行业是如何工作的，业界需要什么样的人？我说："让他们成为能够写作的人，成为在辩论时能切中要害的人。"大多数毕业生走出校门后，能拿到计算机科学的学位，可写作不是他们的强项。

写作是带有个人性质的，所以我觉得很难传授。得要有人一手拿着你的文章，一手拿着红笔，向你解释你什么地方写得不好。这很花时间。你读过Hamming对年轻的研究员的建议吗？

Seibel：*你说的是"你和你的研究"（You and Your Research）那篇文章吗？*

Armstrong：他说过这样的话："做好的东西。"他说："如果不在好的领域中做好的东西，那你做什么都没有意义。"Hamming还说："我每星期都要花上1天学习新东西。这意味着我比同事要多花20%的时间学习新东西。20%的复利意味着4年半之后我掌握的知识就是他们的两倍。而且因为这种复利的存在，每星期一天，这个额外的20%，那么5年之后，我掌握的知识就是

他们的3倍。”反正就是差不多这样一个比例。我认为这个观点非常正确。因为我做研究的时候不是用20%的时间思考新东西，我是用40%的时间思考新东西。而我这样做了30年。所以我发现自己掌握了很多东西。如果有人找我来解决问题，我就可以告诉他们，应该这样做，应该那样做。你刚才问如何才能成为更好的程序员？花上20%的时间学习新东西，因为知识是有复利的。读一读Hamming的文章。那篇文章不错。非常好。

Seibel：*你发现过代码之美吗？*

Armstrong：是的。不过我不知道为什么会发现。有意思的是，如果你给两个程序员同样的问题，当然这取决于问题类型，但本质上是偏重数学的问题，他们最后编写的代码常常是相同的。如果只是格式化问题、变量和函数名的问题，它们是同形的，算法是完全一样的。我们是创造了这些东西，还是说只是擦去尘封发现了它们？就好像是雕塑已经存在，我们只是拨开了上面的蜘蛛网，揭示出早已存在的算法。我们是发明了一个新的算法还是发明了一个已经存在的结构？有些算法感觉是这样的。我认为这更多是数学算法。我在实现电话协议等类似的东西时并没有这种感觉。那种情况不是雕塑已经存在，只要拨开了上面的蜘蛛网就可以了。

Seibel：*这和数学之美是相似的，因为那是自然的一部分。在其他层次上代码也是有一定美学的。*

Armstrong：是的。有点像是风水。我喜欢最小化的代码，那是姿势优美、井井有条的代码。假设你已开始删减，当删到不能再删的时候，也就是说多删掉一点东西就再不能工作了，这个时候的代码就美了。当你能够想到的任何修改都只会让算法变差，那么这个时候的代码就叫美。

Seibel：*你刚才说在你和Robert Virding之间传递代码的时候，你们是如何更改格式上的低级细节，那些东西是程序员们争论不休的地方。*

Armstrong：那些更改并不影响算法之美。

Seibel：*但那也算审美。那是人们的欣赏品味。*

Armstrong：是的。但我不会这么说：“逗号后面有一个空格，所以这个代码很丑。”丑陋是因为某种错误的原因，在本该用二分法的地方用了线性搜索。或者是本该用对数完成的事情用了线性方法。当然，如果搜索的是10个元素，肯定是用线性搜索的，谁会在意呢？但如果是一个很大的数据结构，那肯定是要用二分法的。这时如果采用线性搜索，那就真算不上漂亮。

数学算法，就像是柏拉图式的美丽，更像是一座建筑物。你会欣赏一座优秀的建筑物——这不是数学对象。不是立体图形、球形或棱柱形——是一座摩天大楼。它看上去很好。

Seibel：*是什么造就了优秀的程序员？如果招聘程序员，你要找什么样的人？*

Armstrong：我想这要看解决什么问题了。你是问题驱动还是解决方案驱动的？我一般喜欢说这种话的人："我发现一个非常有趣的问题。"然后你会问他："你写过的最有趣的项目是什么，让我看看这个项目的代码。你准备如何解决这个问题？"我不会总是问别人他们对这种或那种语言懂多少。根据我对程序员的了解，他们要么擅长所有语言，要么一样也不行。擅长C语言的程序员也会擅长Erlang，这是一个非常准确的预言。虽然我看到过例外的情况，但是擅长一种语言的思维能力似乎也能转换到其他语言上。

Seibel：*有些公司因为在面试期间使用逻辑谜题而闻名。在面试的时候你也会问那种问题吗？*

Armstrong：不问。有些非常出色的程序员对那些逻辑谜题的反应有点慢。曾经有一个参加了Erlang语言工作的人，他获得了数学博士学位。我对他的唯一形容就是，他像是能在花岗岩上钻孔的金刚钻。我记得他有一次感冒了，所以就把Erlang程序清单带回家了。等他回来的时候，他在Erlang程序中写了一个原子，说："这个原子将把虚拟器置于死循环中。"他发现那个原子的初始散列值等于0，于是我们把一个值对另外一个值取模，得到的值也是0。他就那样通过很少见的异常情况破解了散列算法。他甚至都没有执行一下程序，看看是否可以正常工作，他只是阅读程序。但是他做得并不快。程序读得相当慢。我不知道如果让他来做那些快速反应的谜题，能做成什么样。

Seibel：*优秀的程序员还有其他特征吗？*

Armstrong：我在其他地方读到过，说要想成为一个好的程序员，记性要好。我认为这是对的。

Seibel：*比尔·盖茨曾经说过，他还能够走到黑板前面，写出一大段他当时为Altair写的BASIC代码，他说这些话的时候离他当时写代码已经有10多年了。你觉得你也能回忆起以前编写的代码吗？*

Armstrong：是的。嗯，我可以重新构造一些东西。有时候我的一些老代码全都弄丢了，但我一点儿也不担心。我没有程序清单或其他资料，只是重新输入一遍。在逻辑上是相同的。有些变量的名称会改变，文件中函数的顺序

会改变，函数名称会改变。但基本上是同形的。输入的新代码也可能会比以前有所改进，因为我的脑子已经钻研一番了。

比如说我10年前编写的编译器中的模式匹配，我现在还可以坐下来输到计算机中。这和原来的版本会不同，但是如果根据自己的记忆来写，会得到一个改进的版本。这是因为此时你并不是在创作，所以这个过程就像是在完善。但是结构可能非常相似。

我不担心丢掉什么代码。代码的模式已经记在脑子里了。嗯，我不能说是记着代码，而是说可能再写一次。实际上是记不住这么多东西的。我说你能够完全记着程序，实际上并不是真的记得，而是可以重新写一遍。如果比尔能够记着那些代码内容的话，我是做不到的。但我肯定可以长时间记着代码的结构。

Seibel：*Erlang风格的消息传递机制是不是解决并发编程问题的银弹？*

Armstrong：噢，不是银弹。这种机制是一种改进。比共享内存编程强得多。我想在这点上Erlang成功了，它实际上已经证明了这一点。当我们首次做Erlang时，我们会去参加大会。会上，我们说："应当复制所有数据。"我想他们会接受容错的论调，因为之所以复制所有数据，为的是让系统容错。他们说："如果这样做的话，效率实在是太低了。"我们说："是的，效率会低，但却是容错的。"

令人吃惊的是Erlang在某些情况下效率会更高。后来我们发现为了容错所采取的措施在很多情况下和共享编程的效率一样，甚至更高。

我们就会问了："为什么会这样呢？"是因为它增加了并发性。如果是共享内存的方式，在存取数据时必须要为数据上锁。而你忘记了上锁的成本。也许你复制的数据量不是很大。如果复制的数据量相当小，而你却要做大量的更新和存取并且需要大量的锁，这时你会突然发现复制所有的数据也不是件坏事。在多核系统上，如果采用的是老式的共享模型，锁会把所有的核都停掉。你有一个千核CPU，当一个程序上了全局锁，这一千个核都要停下来了。

我对隐式并行也非常怀疑。你的编程语言可以有并行构造，但是如果它不用映射到并行的硬件上，如果它只是由你的程序系统所模拟，那这么做就不会有什么好处。一共存在3种硬件并行。

第一种是管道并行。在芯片中做一个深一些的管道，这样就可以并行地工作了。但这是在设计芯片时就一次完成了的，以后不能改。普通的程序员是不可能做这种指令级并行的。

另外一种是数据并行，这不是真正的并行，但与缓存行为有关。如果要让一个C程序效率更高，如果*p在16字节的边界，如果存取*p，那么对*(p + 1)的存取基本上是没有成本的，因为它已经在缓存栈中了。你需要关心的是缓存线有多宽，在一个缓存转换器中放了多少个字节？这就是数据并行，程序员在使用的时候对于它们的结构要非常小心，要准确地知道在内存中是如何放置的。多麻烦啊，你肯定不想用。

还有一种芯片级的真正的并发，也就是多核系统。在2010年的时候将会有32核，到了2019年的时候说不定就有100万核。所以必须在程序中利用并发性的颗粒并把它们映射到计算机的内核中。当然这是一项繁重的作业。在另一个核上开始一次计算并把答案取回来，这个操作本身是要花时间的。所以如果只是把两个数字加在一起，是不值得这样做的——你把数据从一个核移到另外一个核上，然后再把答案取回来，这比在原地执行它要费劲得多。

Erlang在这方面相当适合。因为程序员只用说："我要一个进程，我需要另外一个进程，我还需要再开一个进程。"然后我们只需要把它们放到核上就可以了。也许我们现在应当考虑它们在核上的实际物理位置。一个进程在生成另外一个进程后可能还会与它对话。所以如果我们把它放在物理上接近的核上，是一个不错的位置，不要放在很远的核上。也许当我们知道它们之间的对话不会很多，说不定可以把它放得远一点。也许做I/O的进程可以靠近芯片的边缘，就是那些与I/O进程对话的芯片。随着芯片越来越大，我们要考虑到，把数据放到芯片中央比放到芯片边缘的成本会高。也许你有两三个服务器和一个数据库，还要把它映射到核上，这样我们可以把数据库放到芯片中央，那些需要和客户端打交道的数据放在靠近芯片边缘的地方。我不确定，这还需要研究。

Seibel：*你很关注Erlang做并行计算的方法。你会不会更关注这种想法，传递消息、没有共享的并行？或者说更关注Erlang这种语言？*

Armstrong：想法？当然有啦。人们总是问我："Erlang今后会怎么样？会成为流行的语言吗？"我不知道。我想Erlang已经很有影响力。它最后可能会像Smalltalk。我认为Smalltalk的影响力非常非常大，受到一些热心人的喜爱，但是它从来没有得到真正广泛的使用。我想Erlang可能也是那样。为了取得市场的成功，可能需要微软花点心思，需要这里添点、那里加点大括号，然后放到公共语言运行时中。

（编辑：李莉萍）

7

Simon Peyton Jones

米全喜 译

作为发起人之一，Simon Peyton Jones于1987年参与的那个项目最终促成了编程语言Haskell的定义。他现在是英国微软剑桥研究院的高级研究员。他编辑的《Haskell 98修订报告》是这种语言当前最稳定的定义。他是格拉斯哥Haskell编译器（GHC）的架构师和首席开发人员，按照haskell.org的说法，GHC是"事实上的标准编译器"。他所讲的一句话"不惜一切代价避免成功"被作为Haskell的非官方格言广为引用。

Peyton Jones从未拿过博士学位，曾是大学教授，做起研究来干劲十足，他认为理论与实践中都能发现美感。他开始学习编程用的机器没有永久存储器，只有100个存储单元，读大学后，他用学校里昂贵的大型计算机编写高级编译器，并用学生收入购置的零部件组装了自己的简陋计算机。某次看到一位教授演示了不使用变量修改（mutation）如何构建双向链表，并领略到惰性求值思想的优美，他就被函数式编程深深吸引了。Peyton Jones把函数式编程看做是"对整个编程体系的一次彻底而优雅的攻击"：这种方法"建

造了一面新墙”而不“仅仅是在墙上添一块砖”。2004年，美国计算机协会（ACM）选举他为会士，以表彰他“对函数式编程语言的贡献”。

这次访谈的话题包括：为什么他认为函数式编程越来越有可能改变软件的编写方式；在编写并发软件时，为什么软件事务内存（Software Transactional Memory，STM）在编写并发软件时要远优于锁和条件变量；为什么很难真正地研究并弄清楚不同的编程语言是否会提高或降低程序员的效率，即使在微软研究院也很难做到。

Seibel：*你是什么时候开始学习编程的？*

Peyton Jones：在我上中学的时候。英特尔刚刚生产出4004——世界上第一款微处理器。我们那时没有4004之类的处理器——那种芯片在当时可不是业余爱好者能够得到的。我们那时唯一能够使用的是一台IBM校园计算机，这台机器很奇怪，是用大型机的一些边角料制造出来的。它没有永久存储器，所以每次运行程序时都要手工输入程序。

那台机器一共有100个存储单元，我记得每个存储单元能够存储8位十进制数。程序和数据都存储在里面。这样在编程的时候，最重要的是能够把程序放到这100个存储单元中。我不大记得我的第一个程序是如何编写的了。我和学校里另外一个爱好者在那台校园计算机上投入了很多时间。那是1973年、1974年前后——差不多就是那个年代，我大约15岁。

在那台机器上编了一点程序之后，我们发现斯温顿市的技术学院有一台计算机。然后我们每星期都腾出一个下午，坐上一辆很慢的公交车，花上一个小时到斯温顿市，那里有一台很大的机器Elliot 803。这台机器在房间里独自占了6个白色的、像电冰箱那么大的柜子，房间里还有一个穿白大褂的操作人员。

没过多久，那个穿白大褂的操作人员就知道了我们会用这台机器，所以在我们摆弄那台巨大的机器时她就走开了。机器使用的是纸带和电传打字机，程序放在纸带上。我们用Algol编写程序，那是我使用的第一种高级语言。程序是在纸带上编写的，也是在纸带上编辑的。如果要修改的话，必须通过电传打字机来操作纸带，从电传打字机中打出一条新纸带，在需要修改的地方停下来，输入新字符。这种编程方式非常费力，是一种物理介质上的行编辑器。那就是我第一次编程的经历，它极大地激发了我的兴趣。

Seibel：*但那不是学校的课程。*

Peyton Jones：对，不是！学校里根本没有讲过计算机课程。

Seibel：*当时的情况肯定是——“嗨，孩子们，这儿有一台计算机，你们自己来学学用用吧。”*

Peyton Jones：没错。计算机锁在一个大柜子中，可以借来钥匙，计算机还配了一个屏幕，固定显示寄存器中的内容以及存储单元中的那些十进制数。可以设置好程序，按一下“执行”键。可以单步执行。那就是我们使用的机器。因为根本没有ASCII字符，所以连汇编语言也算不上。基本上就是机器码，只能显示十进制数字，连十六进制都无法显示。

Seibel：*但是配了屏幕？*

Peyton Jones：是有一个电视屏幕。那是唯一的输出介质。

Seibel：*那输入是什么呢？*

Peyton Jones：是一种触摸式键盘。只要触碰那些按钮，按钮就能够感觉指头的压力。相当先进——没有机械按键。那是用电容做的——触摸按键，东西就出来了。一共大约是20个按钮。

Seibel：*只有数字按钮吗？*

Peyton Jones：有数字，还有“执行”、“单步执行”按钮。还有一个“显示内存位置”按钮。非常简陋。不过这让人觉得更加兴奋。

Seibel：*我想，你需要先部署好你的程序，这可能要考虑到很细节的东西，然后才会走到机器那里开始输入程序。*

Peyton Jones：首先要画流程图。然后分解为指令。接着再把指令编码成这种奇怪的数字格式。然后输入数字。基本上要输入800位数字，那就是我们的程序。然后按一下“执行”键。如果幸运的话，这800个数字都没有输错，那就太好了。所以我们会花大量的时间检查程序，一个人盯着屏幕检查，另外一个人在旁边说：“进入下一个单元。”

等我后来去剑桥大学读书的时候，微处理器刚刚开始流行起来。学校里有一个大学计算俱乐部。有一台类似大学计算用的大型机，叫做Phoenix，带有一套特别复杂的记账系统。

你在什么时间使用计算机，这点很重要。给你一定数量的计算机代用币，程序使用的内存越多，你用的代用币就越多；程序运行时间越长，用的代用币也越多。而负载越低，消耗的代用币就越少。我们这些本科生分到的代用

币不是特别多，所以只能在晚上去那里，从晚上9点开始，因为那个时间段运行程序的成本相当低。

从晚上9点到凌晨3点我们待在那里，编写自己的程序。主要使用什么语言呢？我记得是BCPL。和以前一样，编程完全是业余爱好。我那时在攻读数学学位，没有接受过一点计算机科学的正规教育。

那时也还没有一整套本科学位课程。那是1976年～1979年之间。最后一年才上计算机课程，然后你就可以以计算机科学专业毕业。但是你不能学习整整3年的计算机科学，此前必须学习一些像数学或自然科学那样的课程。我开始学的就是数学，到了最后一年改学了电子科学。主要是因为我觉得计算机只是我的业余爱好——如果把业余爱好当做学位课程就有点投机取巧了，学位课程应当是很难的。

但是后来发现数学有点太难了，剑桥有太多顶尖的数学家，所以我就转学电子科学了。

Seibel：电子科学——就是美国所说的电子工程吧？

Peyton Jones：是的。那个时候，我读中学时的那个朋友Thomas Clarke也在剑桥读书。我和Thomas组装了各种各样的计算机。我们自己购买了一个微处理器和很多7400系列的TTL，并把它们用线连起来。我记得我们最大的问题是打印机。打印机和屏幕，这两样是最难弄的。

Seibel：因为那些东西太贵了。

Peyton Jones：非常贵，是的。你能够得到电子零部件，这些是学生买得起的。但是当时的打印机通常都是那种很大、像冰箱大小的行式打印机。里面有很多机械装置，即价格是我们根本负担不起的。打印机以及存储设备——任何一种永久存储设备都很难得到。所以我们的计算机一般会有键盘、屏幕，除此之外就没有什么了。还有一些原始的磁带设备。

Seibel：你们在1976年到1979年就从零开始组装计算机了。Altair[①]也是在那个时候出现的吧？

Peyton Jones：是的。供业余爱好者使用的计算机开始出现了。但是我们认为那些计算机是骗人的。

自己组装机器，软件是个问题。我为那台机器编写的最先进的程序是Conway的“生命游戏”（Game of Life）。那个程序运行得相当好。但是如果

① Altair，是1975年美国推出的一台原始机型。——编者注

要编写一些重要的程序，比如编写编程语言，工作量就太大了，因为计算机中的永久存储介质太有限了。而且全都是用十六进制输入，因为没有汇编程序。

Seibel：*更像是原始机器码。*

Peyton Jones：当然剑桥大学的大型计算机是能够理解BCPL语言的，所以我们编写了很多BCPL程序。我们还为自己发明的一种编程语言编写了一个编译器。那个编译器一直没有完成——太复杂了。那是两个完全分离的世界，一边在大型计算机上用高级语言编写编译器，一边摆弄自己的硬件。

Seibel：*你记得自己编写的第一个有趣的程序是什么？*

Peyton Jones：是在中学的计算机上编写的一个得到24位平方根的程序，程序要放到99个内存单元中。

Seibel：*这样只剩下一个单元了！*

Peyton Jones：是的。采用的是某种牛顿-拉夫申近似法来计算平方根。那个程序让我感到非常自豪。接下来的程序是什么呢？在大型机上第一个有趣的程序肯定就是那个我们一直都没有完成的编译器了。那是用BCPL编写的，极为复杂。但我们雄心万丈。没有类型系统，我们就打印了很多图片、结构图和连接各个结构的箭头。

Seibel：*你的意思是在BCPL中没有类型系统。*

Peyton Jones：是的。所以我们基本上是在大纸上画出类型和箭头，通过这种方式写出类型。那就是我们的类型系统。那是一个相当大的程序——实际上我们过于野心勃勃了，结果一直没有弄完。

Seibel：*你认为自己从那次失败中得到了什么教训吗？*

Peyton Jones：那可能是我第一次认识到，编写一个真正大型的程序时，最后可能会遇到规模的问题——在脑子里无法同时记住足够多的东西。此前我可以毫无困难地记住所有编写的东西。那可能是我第一次认真地尝试着编写长期使用的文档。

Seibel：*在那种情况下，即使有了文档，问题也仍旧存在？*

Peyton Jones：嗯，我们还有很多其他事情要做，像攻读学位等。那些事情都是利用晚上9点到凌晨3点的时间来完成的。

Seibel：*你希望自己当时会采取一种不同的方式来学习编程吗？*

Peyton Jones：嗯，没有人教我该如何编程。我不知道是否也因此而感到有所缺失。今天我觉得自己在编程方面主要的空白点是对面向对象编程缺乏深刻、全面的理解。当然，我知道如何编写面向对象程序和所有那些东西。但是在做具有一定规模的事情时，是有一些差别的。比如构建长期使用的大型程序，要以复杂的方式使用类层次结构，或者要构建框架，这时就需要我所说的深刻、全面的理解。这些知识不是从书本上能马上学到的。

因为有所欠缺，所以对于面向对象编程能够做什么、不能做什么，我感到自己在这方面不够权威。我总是谨慎发言，特别是因为命令式编程非常尖端，有丰富的编程范型，所以我对此不想做出负面的评价。但是大概因为我自己的生活经历，我从来没有真正地花上几年时间来编写大型的C++程序。那才是产生深刻、全面的感觉的途径，但我却从来没有。

Seibel：*我认为那种感觉一般都会是厌恶。*

Peyton Jones：是的，但是那种厌恶是经过深思熟虑的，不是草率地说些“噢，那个东西太糟糕了”之类的厌恶。

Seibel：*你在剑桥完成了3年的课程，然后呢？*

Peyton Jones：我接着在想：“嗯，最好是在计算机方面做点工作。”于是我花了一年时间攻读计算机科学的研究生学位——那是我在计算机科学方面唯一的一段正式教育。

Seibel：*类似于硕士学位吗？*

Peyton Jones：有点像硕士学位。那是非常美好的一年。我觉得那个学位类似于计算机科学专业中的本科荣誉学位。但那是为从来没有学过其他计算机科学课程的学生设置的。

Seibel：*回到研究领域之前，你在工业界度过了几年。那时你在做些什么呢？*

Peyton Jones：那是一家非常小的过程控制和监控公司。我们构建的硬件和软件放到使用微处理器的计算机中，计算机又放到传输带的测重控制器上。我构建的程序用来监视运煤传输带上的测压元件。程序控制传输带的速度，侦听测压元件的反应，把传输带调整到合理的流速。那是一个小型的实时操作系统，我是用一种叫做PL/Z的语言编写的，它有点像Algol。程序是在Z80机器上编写的，那种机器运行的是一种简化的Unix系统，叫做Chromix。

那是一个很小的公司——好像也就是六七个人，最多的时候达到过15人。但是公司很小，一切都不太稳定。有时候我们有很多钱，有时候又一分不剩了。在经过两年之后，我认定那种创业生活并不适合我。我对小公司认识的最重要的一点是：作为企业家，即使在面临金钱窘境时，仍需要保持精力旺盛，而我的精力却恰恰被这种窘境给消磨殆尽了。我的老板是公司的执行总裁。事情越糟糕，他的精力就越旺盛。他经常和人们谈论软件，提出很多新的技术想法。他像蜜蜂一样快乐。我认识到这种态度是必需的，因为如果这些事情让你的精力消磨殆尽的话，你就会一直是劳而无功的。

我认定那份工作对我来说太难了，我四处找工作，最后在伦敦大学学院找到一份讲师的工作。我去那里的时候还没有博士学位，也没有接受过研究方面的培训。于是系主任给我腾出时间，让我做研究。他给我的教学工作量很少，这样我可以开始研究工作了。但是应当干些什么，我一点主意也没有。我坐在办公室里，拿着一张白纸和一支削尖了的铅笔，等待灵光乍现。一片宁静，我四处张望，期待着好想法的出现。但是什么也没有发生。

John Washbrook是系里的一名资深学者，一直很关照我，他对我说过一句非常重要的话。他说："动手去做，不管这件事有多么不起眼。"这句话针对的并不是编程，而是针对研究工作而言的。不管事情看起来多么不起眼、多么老套、多么不重要，都要去做，在纸上写下来。因此我就这样去做了。事实证明这的确是至理名言。

此后我对我的每个研究生都会讲这样的话。因为这就是你开始的方式。一旦开始转动起来，就会发现计算机科学不管看上去差别有多大，都有相似之处——几乎每样东西都非常有趣，因为这门学科一直走在你的前面。它并不是一些定型了的东西，等着你去发现。它的范围在不断地扩张。

Seibel：这么说你回到了学术界，但一直都没有拿到博士学位。这怎么可能呢？

Peyton Jones：现在如果没有博士学位而想得到教师的职位会非常难。那个时候是1982年、1983年。当时我妹妹正在伦敦大学学院读计算机科学专业，她说："噢，伦敦大学学院有几个讲师职位，你何不试试？"于是我就向伦敦大学学院提出了申请。让人吃惊的是，我居然被录用了。我只能这么推测，那个时候肯定非常缺人，任何一个人，只要在计算机方面能够证明一下自己，都可以被聘用。如果不是这种情况，他们怎么会聘用一个没有博士学位的人呢？

在伦敦大学学院待了7年之后，我觉得自己也许应当拿个博士学位。不过写学位论文是一件很麻烦的事情。但是，后来发现在剑桥可以依据特殊条例来取得博士学位，只要提交已经出版的作品就可以了，如果运气好的话，他们会说："你这个人不错，应当得到博士学位。"我正准备这样做的时候刚好被任命为格拉斯哥大学的教授，是正教授。那时别人已经开始称呼我为"教授"了，他们不关心我是否有博士学位，于是我放弃了拿博士学位的想法。Robin Milner也没有博士学位，可这个群体依然相当了得[①]。此后我一直就这种状态了。

Seibel：*现在拿博士学位还有价值吗？有人曾经跟我说，博士学位实际上是个就业学位——如果想成为教授，你就得拿个博士学位；如果不想当教授，这个学位就没有意义。你认为这样的分析适用于计算机科学吗？*

Peyton Jones：这种说法当然有一定道理。博士学位不能保证你成为研究员，但是如果你想继续从事研究工作并以此为职业——在学术界也好，在像微软研究院或Google研究实验室这种严肃的业界研究实验室也好，博士学位都是必不可少的。在这种情况下你肯定需要博士学位这块敲门砖。

如果你不想以研究员为职业，那么攻读博士学位就成为一种自觉自愿的事情了。如果你从事的是自己非常热爱的工作，生产率会提高5倍。如果你发现自己是这么想的："我很喜欢，愿意花点时间再深入研究一下。"那么在英国，攻读博士学位是一个非常难得的学习机会，花上三年时间专心学习一些东西，在美国时间还要更长一些。这段时间你非常自由，有点像社会的寄生虫。如果你知道自己不想做全职的研究工作，那么你只能凭借热情、求知欲、兴趣来攻读博士学位了。但不管怎么说，博士学位相当奇怪。你不得不独自工作，写一些大多数人都不会阅读的很长的学位论文，但他们会阅读你的其他学术论文。所以说这是一种很不寻常的研究模式。

在拿到博士学位后，你开始更多地与其他很多人一起协作，做一些小的、非常琐碎的工作。从某种程度上讲，我认为博士学位是一种奇怪的准备工作，即使你准备从事研究工作，这也显得很奇怪。在英国会更奇怪是因为时间表被压缩得很紧。我认为在美国，在你完全集中精力做自己的研究课题之前，你肯定会有一段时间与他人有更多的协作。

---

① Robin Milner（1934—2010）是函数式编程语言ML的发明者，1991年图灵奖得主。他没有获得过博士学位，在大学毕业后做过中学教师和程序员，于1963年进入伦敦城市大学教书并先后在几所大学任职。

Seibel：说起研究和学术，函数式编程在研究领域相当流行，但是我觉得在这个领域之外的很多人在看待函数式编程时都有一种观点，认为函数式语言虽然简洁，但是过于数学化了，和日常编程脱节了。这种说法有道理吗？

Peyton Jones：我认为这种描述有一定的道理。我归纳的函数式编程的特征——我说的是纯函数式编程，先不考虑其他副效应——是对整个编程体系彻底、优雅的攻击。我所说的彻底是名副其实的彻底，不是只考虑事物所处状况的发展。

我们当今“所处的状况”是大公司在生态系统、编辑器、剖析工具、各类工具、程序员、技能等诸如此类的东西上投入了大量资源。所谓的主流，就是以实效为主导。而函数式编程这种彻底、优雅的能力算不上是那种根本的、基础结构上的支持。但与此同时，也没有必要为了追求这种支持而改变函数式编程。因为，归根到底，如果事情做得不够彻底，不够精良，那么改进也只是局部的、增量的优化，且无法达到另一个层次。

所以我认为在学术研究领域最好的就是，教授做着他们感到着迷的事情，没有人会问你这对公司利润有什么贡献。事后会证明有些事情非常重要，而有些则不然。但此前是无法判断的！所以从全局来看，有些人与我一样在纯函数式编程上付出那么多时间是值得的，函数式编程的前景非常美好。我不想说人人在今后都要采取这种方式写程序，但是函数式编程的确有着美好的前景。实际上我应当说是它有着越来越好的前景。我认为，当命令式编程的石灰岩磨尽的时候，就会看到函数式编程的花岗岩了。

话虽如此，初期的纯函数式编程过于偏向技术，太学术化、数学化。我这20年来一直从事函数式语言的工作，在这20年间它正逐步地贴近实际，不再只关注抽象概念，而且还关注如何克服现实生活中程序员在使用函数式编程语言构建实际应用程序时所遇到的一个个障碍。开发Haskell语言本身就是这样的例子。

如今一个很好的现象是，有很多人（可能稍有点言过其实）正走向主流，在纯函数式领域中学到的观点和想法可能会给主流领域带来信息、带来启发。我们已经看到这种现象了。函数式编程就像实验室，很多关于类型系统和泛型的东西最初都是在函数式编程语言的环境中开发出来的。还有生成器和惰性流。Python有语法级的列表解析，还有很多比较特别的东西。通常它们被改头换面了，有时候为了适应主流环境，它们要做很多修改。我不想声称它是一种排外的专有体系，但是我确实认为函数式编程中的很多思想还是渗透进来了。所以说是有用的。

Seibel：*对你来说，研究与实际编程之间的关系是什么？*

Peyton Jones：噢，相互之间有着很大的影响。我的研究领域是编程语言。编程语言最终的目的是什么？是为了让编程更容易。它们实际上是编程的用户界面。所以说编程和编程语言的研究是密切关联的。布丁做得好不好，吃的时候就知道了，所以我们也应当观察一下程序员是怎么"吃的"，但我们在这方面做得并不好。我的意思是，应当对程序员的编程工作做一些适当的、正式的研究，看看他们是怎么做的。但这样做的成本很高，也很"不精确"。要想得到一个明确而不含糊的结果就更难了。

编程语言领域的文化更多是"证明你的类型系统是正确的、完整的"。我们也许都回避了一个更重要也更难回答的问题：在现实中，编程语言是否能够让人们的工作效率更高。但是这些问题确实很难给出令人信服的答案。对同样一件事情，分别编写函数式程序和面向对象程序，哪一种工作效率更高呢？就算你可以花上很多钱来认真地实验，我也不敢保证你能得出大家会实际买账的结果。

Seibel：*你们做过实验吗，哪怕是很小的实验？你们在为微软工作，微软有很多钱，你们为什么不找一个经验丰富的Haskell团队和一个经验丰富的C#团队，交给他们同样的任务，看看会发生什么情况？你们需要做些这样的实验，对吧？*

Peyton Jones：是的，是这么回事。部分是钱的原因，但又不仅仅是钱的原因。还有时间的问题，还要考虑人们是不是关心这个问题。要做这种实验，你的整个方法都是不同的。在文化上也需要转变。而且，在外人看来，微软似乎有很多钱，但实际上我们大体就是一个研究员加上一个工作站。不是每件事情都可以花钱的。如果可以的话那就好了。在雷德蒙德市有一些大型的可用性实验室，这些实验室贴近实际工作环境，人们在那里做一些重要产品的实验。新版的Visual Studio就在那里经过了广泛的可用性测试。

Seibel：*他们考虑的大概更多是用户的总体交互，而不是编程语言问题。*

Peyton Jones：嗯，他们也会做一些有趣的API测试工作。Steven Clarke和他在雷德蒙德的同事们做过一些系统的尝试，观察程序员们的工作方式，交给程序员一个新API，谈谈准备如何使用这个API。他们还让设计API的人坐在玻璃屏幕的后面，观察程序员的工作。

坐在玻璃屏幕后面的那些人经常会叫起来："不，不，不要那么做！那

种方式不对！”但是玻璃屏幕是隔音的。这种方式通常都证明是有很大指导作用的。设计API的人离开后就去修改API了。说实在的，编程语言的研究在这方面做得很不好。但部分是因为这些是很难回答的问题，而且我们也没有适应这种文化。我认为这是不足之处。但是我个人觉得没有谁特别适合处理这些问题。

Seibel：*对于如何改善编程方法，如果研究人员提出了有趣的想法，那么那些最好的想法是不是能迅速从研究实验室和大学渗透到实践中？*

Peyton Jones：唔，够快吗？我不知道。人们开发的产品是客户需要而且也准备花钱购买的，每次和这些开发人员交谈的时候，我都清楚地知道，很多困扰我的事情根本都不在他们的考虑范围之内。

他们在这个星期内必须做出一些客户感到有价值的东西，他们根本没有时间应付一些可能有效，或者说即使在某些方面可能有效但还没有完全成熟的东西。

这种说法有点前后不一致——像是鸡生蛋、蛋生鸡的难题。有些时候，为了让一些在研究领域提出的想法能够直接产生实际效用，我们确实需要在与非基础研究领域交界的地方做很多工作。

我的意思不是说基层开发人员反应迟钝，不采用那些对他们有益的想法。他们按照自己的方式行事是有相当充分的理由的。有时候，在研究原型和你在实际中能够构建的东西之间有一定差距。我认为微软实际上在这方面做得相当好，微软研究院确实弥补了其中的一些差距，并且有一定的机制——孵化小组及类似的机制，他们的目标是让研究人员和开发人员相互更紧密地接触，也许还能对跨界合作提供额外的帮助。我认为，在跨界合作这方面，微软研究院已经做得不错了。

这就像洋葱一样是一层一层构成的。对于充满Java的主流开发场所，函数式编程的方式不仅与他们编程思考方式有着天壤之别，而且相互操作方面存在很多问题。你有足够的书可以参考了吗？有足够的库可以用了吗？所以编程应该有一整套生态系统，包括人、技能、库、框架、工具，等等。

如果这种阻碍多到一定程度，你就会有些困惑了。所以我认为在编程语言中不同的研究技术就像是处于光谱中不同的点上。有些新技术更多是对当前状况的演进。你可以说：“把它放到你现有的框架中，它可以在标准的、未做修改的Java上工作，这是一种静态分析，能够指向代码bug，非常好！”这比“这是一种全新的编程思考方式”要容易接受得多。

话虽如此，如果具体到函数式编程，我认为人们的态度已经发生了质变。听说过函数式编程的人比以前多得多。突然间，不用再总是解释Haskell是什么了，有时候人们会说："噢，我听说过那种语言。我上个星期还在Slashdot上读过这方面的内容，我觉得它相当酷。"这种事情在几年前是不可能发生的。

但是深层的原因是什么呢？它就是一个随机的流行事物吗？还是也许部分是因为越来越多的学生曾经在大学里学过函数式编程，而他们现在已身居要职。也许是这样吧。但原因也可能是：为了处理不定的副效应所带来的坏结果，软件规模也相应地增加了，以及因为我们要处理更多的验证和并行问题，因而所有这些问题都变得更加紧迫了。我想这是激起人们更大兴趣的原因。我想这将逐步打破成本和收益的平衡。

Seibel：你是什么时候开始接触函数式编程的？

Peyton Jones：在剑桥大学的最后一年我听了Arthur Norman讲的一门很短的课程，我才听说函数式编程。Arthur Norman是系里非常优秀、稍有点古怪的讲师，一个很了不起的人，对符号代数很感兴趣，很热衷于Lisp。他开了一门很短的函数式编程课程，在课上向我们展示了不使用任何副效应就可以构造双向链表的过程。直到现在我还清楚地记着这件事情，那是我第一次知道还可以做一些不一样的事情——你可能会认为，为了构造双向链表，必须分配单元，必须填充单元，让单元相互指向对方。反正不管怎么样，都只能使用副效应。

但是他向我们展示了，通过纯函数式语言，可以不使用任何副效应就能编写双向链表。当时我对函数式编程基本是一无所知，他的展示让我看到了函数式编程作为一种手段，不仅仅能写一些玩具程序，还能够编写一些相当有趣的程序。

Seibel：估计很多人在看到那个演示的时候都会说："噢，也没什么意思。"然后又回去搞BCPL的编码了。你职业生涯的大部分时间都在告诉人们如何才能真正地使用函数式编程，你认为自己为什么能够走得这么远？

Peyton Jones：当时还有另外一个因素，那就是David Turner关于S-K组合子的论文。S-K组合子是转换并执行lambda演算的一种方法。我当时学过一点lambda演算，也许是受到了潜移默化的影响。Turner的论文说明了如何把lambda演算转换为3个组合子：S、K和I。S、K和I是闭包lambda演算中的项。

实际上是说："这些复杂多变的lambda项都可以转换为这3个组合子。"实际上，因为I等于SKK，所以也可以去掉I。

其中有一个奇怪的编译步骤，把一个你能够理解一部分的lambda项转换为一堆根本不理解的S和K。但是在把参数代入的时候，奇迹般地，它计算出的结果和原始lambda计算出的一样。有的方法非常巧妙，在那个时候让我觉得不可思议，就如这个例子，但是你会看到它总是可行的。

我不是很清楚是什么原因让我对它那么感兴趣。我发现它非常鼓舞人心。我估计部分是因为我对硬件感兴趣，我感觉它是一种能够实现lambda演算的方法。因为lambda演算看上去根本不像实现机制。它更像是数学的思考方式，和机器离得相当远。这个S-K看上去是可以运行的，而实际上也是可以运行的。

Seibel：这么说，你认为可以构建一个和S与K密切相关的机器，然后把程序编译成一系列的S和K操作符。

Peyton Jones：实际上那正是我的几位朋友所做的事情。William Stoye、Thomas Clarke和另外几个人，构造了这台机器，SKIM（SKI Machine），直接执行S和K。由于某种原因，我没有直接参与那个项目。但我的想法是在那个时候形成的。John Backus的题为"程序设计能从冯·诺依曼模式中解放出来吗？"的论文在当时影响非常大。那是他在图灵奖颁奖仪式上的演讲，这位Fortran语言的发明者说，实际上"函数式编程是未来的发展方向"。

而且，他还说："也许我们应当开发一种新的计算机架构来运行这种程序。"既然对这个研究领域有了如此之高的认可，我们也就都在狂热地引用那篇论文。SKIM就是这样一个例子。我们认为这种程序执行方式，哪怕只是程序思考方式都非同寻常，说不准会由此诞生一种完全不同的计算机架构。那个阶段大约是从20世纪80年代到90年代——产生的是函数式编程的基本架构。现在看那种做法稍有点偏离正轨，但的确让人振奋。

另外，惰性求值也对我产生了巨大影响。我现在才认识到惰性求值的美妙，而在那时只是隐隐觉得它可能会比较重要。惰性求值就是函数不立即对它们的变量求值（而是等到变量取用的时候再求值）。当然，仍然是那些优雅的、美丽的、不寻常的、根本的特质打动了我。

这些优点激发了我们的想象：它似乎可以发展成为一种全新的编程思考方式。我们能够做的不是在墙上添砖，而是建造一面新墙。这让人很激动。我立刻充满动力。那只是奇技淫巧吗？在某种程度上我认为奇技淫巧非常重

要。惰性求值是如此巧妙，你可以做一些你觉得无法实现、非同寻常的事情。

Seibel：比如说？

Peyton Jones：我记得我的朋友John Hughes为我写过一个程序。在一个项目中，我要实现两个lambda演算并比较它们的性能，John给了我一些测试程序。其中一个程序是计算e的任意小位数的值。这就是惰性程序——它产生了e的所有小数位，是一个相当漂亮的程序。

Seibel：最后能够得到所有位数？

Peyton Jones：是的。但是取决于使用者。事先不用说明需要得到多少位小数。给你一个列表，你不停地从里面取出元素，直到程序经历了足够的计算周期，它才会给出另一个数字。如果编写的是C程序，这种事情就不是轻而易举能做的。话又说回来，如果方法足够巧妙，也是可以做到的，但那不是C语言固有的编程范型。而John的程序只有四五行，令人惊奇。

Seibel：后来其他语言实现了那种计算的特例，比如说Python的生成器或者其他一些方法，从中可以得到值。有没有什么东西让你会说“啊哈，很多东西都可以看做是无穷级数计算，从中可以抽取答案，直到我们不再需要为止”？还是说也可能是相反的说法：“噢，那种技术对于某些问题来说是有趣的，但并不适合于每个事物。”

Peyton Jones：我在那个时候还没有那么多感悟。我只是觉得太酷了，而且很有意思。我认为重要的是做一些你觉得激励人心的、有趣的事情，并一直做下去。我只是觉得它鼓舞人心。至于为什么应当采取这种编程方法，我没有认真思考过深层次的、理论上的原因。我只是觉得那是一种相当好的编程方法。比如我喜欢滑雪。嗯，为什么喜欢滑雪呢？不是因为我觉得滑雪可以改变世界，只是觉得滑雪好玩。

我现在觉得惰性最重要的一点是保持纯粹性。你可能在我的几个讲座中看到过这个观点。但实际上我真的喜欢惰性。如果只让我选一项，我会选择惰性语言。我认为惰性对于所有编程来说都有很大帮助。你肯定读过John Hughes的论文“函数式编程为什么重要”（Why Functional Programming Matters）。那篇文章可能最早解释了惰性为什么非常重要，而不仅仅是一种巧妙的编程方式。他的主要观点是惰性可以帮助你编写模块化的程序。

惰性求值可以让你编写生成器——他的例子是在象棋游戏中产生所有可能的移动步数——与使用者无关，程序遍历树并做alpha-beta极大极小值或类

似的算法。或者如果生成答案的所有近似值的序列，那么要有一个使用者说明何时停止。实际上，将生成器和使用者分隔开可以模块化地分解程序。但是，如果生成器必须和使用者一起产生，由使用者决定何时停止，那么程序的模块化程度就会低得多了。模块化是将不同的想法分隔在几个不同的、可以组合在一起的地方。John的论文给出了一些很巧妙的例子，通过这些方法，可以更改使用者，也可以更改生成器，相互独立；通过这些方法也可以把新程序放到一起，而如果修改的是紧密交织在一起的程序，就很困难了。

这些都说明了为什么惰性是一种非常好的机制。对程序的局部也非常有帮助。你一般会发现Haskell程序员们会写下一些函数的定义，带有一些局部定义。他们会说“$f(x)$等于什么、什么、什么，其中……”在后面一句中他们会写下一些定义，但这些定义并不是在任何情况下全都需要的。尽管如此，还是把它们都写下来了。需要使用的那些定义会进行求值，不需要使用的定义就不用求值。这样你就不用想了：“噢，好家伙，所有这些子表达式都要进行求值，但是如果被0除，程序会崩溃的，所以不能对它求值，必须把定义挪到条件语句适当的分支中。”

这些都不需要。你一般只是写下可能用到的辅助定义和需要求值的定义。所以这是要考虑编程是否方便的问题。那是一种非常、非常方便的机制。

但是回到全局上，如果有一个惰性求值器，更难精确地预测表达式在什么时候求值。这也意味着，如果想在屏幕上打印一些东西，在每一种按值调用的语言中，求值顺序是非常明确的，实现打印的方式是使用一个不纯的“函数”——这个函数是带引号的，因为那根本不是函数——还有一个类型，类似于String与Unit的关系。调用这个函数，然后作为一个副效应，函数在屏幕上打印一些内容。那是Lisp中的方式，ML中也是这样。在每一种按值调用的语言中基本上都是这样。

现在说说纯语言的实现方式，如果有一个的函数，你是根本不需要调用它的，因为你知道它是会给出答案单元的。函数所做的事情就是给你答案。你知道答案是什么。但是当然，它是有副效应的，很重要的一点是你要调用它。在惰性语言中，困难之处在于，如果你说“`f(print "hello")`”，那么`f`是否对其第一个参数求值，对于函数的调用者来说并不明确。这与函数内部的构造有关。如果向它传递两个参数，`f(print "hello")`和`f(print "goodbye")`，那么可能打印出一个，也可能以任意顺序把两个都打印出来，也可能一个也不打印。所以不管怎么说，使用惰性求值，通过副效应来进行输入/输出是不可行的。通过这种方式无法写出有意义、可靠、可预测的程

序。所以，必须容忍这一点。这确实是有点尴尬，因为你不能做任何的输入/输出。所以长时间以来我们基本上只有把字符串转换为字符串的程序。那是整个程序做的事情。输入字符串是输入，结果字符串是输出，程序所做的就是这些。

也可以巧妙一点，在输出的字符串包含一些输出命令，由一些外层的解释器解释这些命令。比如输出字符串可以说："在屏幕上打印这个，把那个存在磁盘上。"解释器是可以完成这些操作的。你会认为函数式程序很好、很纯，只是那种邪恶的解释器解释了命令字符串。但是接着，当然，如果读取一个文件，如何把输入回送给程序呢？嗯，那不是问题，因为可以输出一串命令，由邪恶的解释器解释，使用惰性求值，将结果转储回程序的输入中。这样程序现在可以取得一系列请求的一系列响应。一系列请求进入和世界打交道的邪恶解释器中。每个请求产生一个响应，然后回送到输入中。因为求值是惰性的，程序及时释放出响应，来到循环中，作为输入来使用。但是容易出问题，因为如果过早地使用响应，就会得到某种死锁。因为在寻求答案的时候，答案所对应的问题还没有从后台传过来。

惰性的这个特点让我们陷入了困境，我们必须围绕着I/O问题进行思考。我认为这是极其重要的。惰性最严重的一个问题是让我们陷入了那样的困境。但是它开始的时候不是这样。开始的时候，惰性是非常酷的，多么好的一种编程方式。

Seibel：*自从开始编程后，你认为编程的方式有什么改变吗？*

Peyton Jones：我认为编程方式一个大的改变可能与单子（monad）和类型系统有关。20世纪80年代早期，我们思考的是纯函数式编程，类型系统相对简单，与那时相比，我现在考虑的是将纯函数式语言、命令式语言、由单子协调的并发编程混合起来。类型变得比以前先进多了，允许表达的程序范围比我那时能够想象的要宽广得多。我认为这两种改变都可以看做是某种程度的演进。

Seibel：*比如说，你第一次编写编译器时没有成功，你在此后编写了很多编译器，肯定学到了很多让你现在能够成功编写编译器的经验。*

Peyton Jones：是的，嗯，是很多东西。当然那是一个命令式语言的编译器，是用命令式语言编写的。现在是用函数式语言为函数式语言编写编译器。但是GHC，也就是我们为Haskell语言编写的编译器，一个很大的特征是它使用的中间语言本身是带类型的。

Seibel：*中间表示法中的类型仍旧是源代码中的那些类型吗？*

Peyton Jones：是的，但是更加明确。在源代码中，有很多的类型推断，源语言要仔细地构造，这样才可能实现类型推断。在中间语言中，类型系统要广泛得多，因为它更明确，所以表达力也好得多：每个函数参数都由它的类型修饰。没有类型推断，中间语言中只有类型检查。中间语言是一种显式的类型语言，而源语言是隐式的类型语言。

类型推断基于一套精心挑选的规则，确保处于类型推断引擎能够分析的范围之内。如果使用源到源的转换来转换程序，也许就跨出边界了。类型推断无法再处理它。这对于优化来说很不好。你不希望优化担心是否可能已经跨出了类型推断的边界。

Seibel：*这也说明了那些程序需要是正确的，因为假设源到源的转换是合法的，如果这个转换是手工编写的，编译器就会说："抱歉，我无法知道类型。"*

Peyton Jones：是的。这是静态类型系统的本质，也说明了动态语言为什么仍旧引人关注并且重要。你可以写出这样的程序，它不能由特定的类型系统识别类型，但是在运行时又不会"出错"，这就是黄金标准——不会出现段错误，不会把整数和字符相加。这样就可以了。

Seibel：*当动态类型的拥护者和静态类型的拥护者争吵的时候，动态类型的支持者会说："嗯，有很多那样的程序——静态类型妨碍了我写程序的方式。"静态类型的支持者会说："不，虽然存在这种情况，但实际上并不是问题。"你怎么看呢？*

Peyton Jones：部分原因是看你对它熟不熟悉了。这很像我说的在编写C++程序时一样，我没有发自内心的感觉。或者，你从来没有使用过惰性求值，所以不会因为不使用它而感到遗憾，但我用了很多，如果不使用它我就感到不便。也许动态类型有点像那种情形。我的感觉是——不管这种感觉对不对吧，可能我有文化偏见——大量的程序可以都很好地静态类型化，特别是在那些类型很丰富的系统中。而且在能够使用的地方，它是非常有价值的，其中的原因已经说过很多次了。

但是一个不太常提到的原因是可维护性。如果你有一大块3年前编写的代码，需要进行系统化的更改——不是对一个过程进行微调，而是对会产生很大影响的代码进行修改——我发现类型系统的帮助是巨大的。

这种事情也出现在了我们自己的编译器中。我可以对GHC进行修改，对遍布于编译器各处的数据表示进行更改，并且可以很自信地说我已经找到了

所有用到这些数据的地方。在动态的语言中，我会非常担心。我会担心漏掉一个数据，在交付的编译器中，某些人输入一些我从来没有处理过的数据，跑到了一些我没有统一更改的东西上。

对我来说，静态类型也部分地解释了程序所做的事情。那是一种小语言，在这种小语言中，可以说明程序做了些什么，但说得不是太多。人们常常问："函数式语言中，和UML图对等的是什么？"我能够得出的最好的答案是：类型系统。面向对象的程序员可能画一些图，而我则是坐在那里写类型签名。说实在的，它们不是图形化的，但因为是正式的语言，所以成为程序文本的永久部分，可以对编写的代码做静态检查。所以它们也有各种各样的好属性。它几乎是程序所做的部分事情的架构描述。

Seibel：*你是否写过这样的程序，你知道是正确的，但是不知为什么落到了类型检查程序的边界之外？*

Peyton Jones：在进行泛型编程的时候会遇到这种情况，比如说在泛型编程中，需要编写的函数接受任何类型的数据，遍历并序列化数据。那个时候类型可能有点尴尬，而无类型语言会特别简单明了。在无类型语言中编写序列化器是一件再容易不过的事情了。

现在有一小部分业界的人描述编写泛型程序的聪明的类型方式。我觉得这样的事情很吸引人。但是不知道为什么，它不像使用动态类型语言那样容易写。我正试着说服John Hughes为*Journal of Functional Programming*杂志写一篇文章，介绍一下静态类型为什么糟糕。John是一位主流的、强类型的、很资深的函数式程序员，现在在做很多无类型Erlang方面的工作。让他来写一篇文章，介绍一下为什么静态类型很糟糕，我认为这会很有意思。我认为他会写一篇引人思考、有趣的文章。我不是很清楚最后会怎么样。

我想我还是会说："在能够使用静态类型的地方就要使用静态类型，因为它会让维护工作很轻省。"它帮助你思考程序，帮助你写程序，还有类似的好处。但是我们还在不停地产生越来越复杂的类型系统，这种现象说明了我们正试图扩展我们的边界，向世界展示更多，也要涉及更多的程序。所以说故事还没有结束。

依赖于类型的编程人员会说："类型系统最终将能够表达任何事物。"但是类型是一些有趣的事情——类型就像是非常紧凑的规则说明语言。它们描述了一些有关函数的事情，但不是很多，不至于让你一次记不下来。所以关于类型一个重要的特征是简明扼要。如果长达两页，就无法再传递它应当传

递的信息了。

我认为我想看到的事物的发展方向是仍旧简洁和紧凑的类型，带有一点弱类型，目的是它们可以简明扼要，有不变量，也许可以用一种比可推断的类型系统更丰富的语言来描述，但仍旧经得住静态检查。我在另外一个项目中做的某些事情是试着为前置条件、后置条件和数据类型不变量做静态验证。

Seibel：与Eiffel中的契约式设计（Design by Contract）是类似的？

Peyton Jones：是的。你将能够为一个这样的函数来写契约："你给我一个大于零的参数，我给你一个小于零的结果。"

Seibel：你是怎么做软件设计的？

Peyton Jones：我想说的是，在考虑编程的时候——考虑编写GHC的一些新内容的时候，我考虑的不是如何用代码表达想法。更多要考虑的是，要表达的想法是什么？

举个例子，我们在迁移GHC后台，也就是代码生成部分的过程中，采取了一种新的重构方式。那时在编译器中有一个步骤，所做的事情基本上是把函数式语言转换为命令式语言C--。那是很大的一个步骤。之所以叫C--，是因为它是C的一个子集。但实际上它本来是用作一种可移植的汇编语言的。它不是用ASCII码打印的——它只是一种内部数据类型。在编译器中这个步骤是，一个函数把表示函数式程序的数据结构转换为一个表示命令式程序的数据结构。这一步应该怎么做？

嗯，那个时候我用了一些相当复杂的代码来实现。但是几天前我意识到那个步骤可以分成两部分：首先将它转换为C--的一种方言，允许过程调用——在过程内部可以调用一个过程。然后把它转换为没有调用的子语言——只有尾部调用。

这样问题的实质就很清楚了：数据类型是什么？这种C--是什么？它是一种代表命令式程序的数据结构。在做第2步的时候，遍历程序。看看每一小段，一次一段。这样注意力将转到控制流，也许可以通过控制流回到前面。一个表示它的好的结构叫做"拉链"，这是一种非常有用的纯函数式数据结构，可以把注意力放到纯函数式的数据结构上。

哈佛大学的Norman Ramsey找到一种方法，使用它来遍历表示命令式控制流图的数据结构。他、John Dias和我花了一些时间，重新构造了GHC后台，采用的基本上就是这种重构的技术。这样做会让后台更具普遍性，它就可以

用作其他语言的后台。

我们的很多讨论基本上都在类型层次上。Norman会给出类型签名，说："这是API。"我会说："看上去很复杂，为什么会那样呢？"他会解释原因，我会说："采取这种方式不是会简单一些吗？"于是我们会花很多时间，在类型描述这个层面上反复讨论。

但很多时候，很多内容都不是关于编程的——它是关于，要实现的想法是什么？我们使用这个数据流分析究竟是想做什么？你试图清楚地说明程序的这个步骤的目的是什么。所以我们花了相当多的时间搞清楚输入输出是什么，并且思考输入输出的数据类型。只要能够让数据类型正确，你就已经描述了很多程序要做的事情了。实际上，非常大量的内容都描述到了。

Seibel：*对类型的思考与实际地坐下来编程有什么联系？当你画出类型后，你能坐下来编写代码吗？还是说编写代码的过程又有助于你理解类型？*

Peyton Jones：噢，更多是后一种情况，是的。我会立即开始在文件中写类型签名。实际上我可能会开始写一些代码来操作这些类型的值。然后我再返回去更改数据类型。这不是一个两阶段的过程，我们不会说："类型写完了，可以写代码了。"

如果说这种说法有什么不合规则的地方，那就是这样的经验并不是来自大型团队队员。你在写代码的时候可以这样做，而其他人仍旧能够理解你做的事情，这样的情况也许不会出现在一个大型的团队中。

Seibel：*你说过，在GHC的最后一次代码剧变中，代码变得更具普遍性了。GHC是一个很大的程序，随着时间的推移而演进，你有可能从普遍性中受益，也有可能为过度的普遍性而付出代价。你是否学到一些经验，如何在过度普遍和不够普遍之间找到平衡？*

Peyton Jones：我默认的做法是在开始的时候不要写很普遍的程序。我会试着让我的程序尽可能地*美*，但不是尽可能地*普遍*。这是有差别的。我在编写代码来解决手边的任务时，会尽量使代码清晰、易懂。只有当我发现自己多次编写基本上相同的代码时，我才会想："噢，只要做一次，把两个代码不一样的地方参数化并向它传递另外的参数。"

Seibel：*你实际的编程环境是什么？你使用的是什么工具？*

Peyton Jones：噢，非常原始的工具。我就是坐在那里使用Emacs编辑并用GHC编译。就是这些。我们的编译器有一些剖析工具，人们经常使用这些

工具剖析Haskell程序。我们也用那个工具剖析编译器本身。GHC转储很多中间输出，这样就能看到发生了什么事情。

对我来说，调试是经常的事情，编译器无法产生好的代码，我会盯着内部状态看。或者，找出这一小段代码，编译一下，看看情况。我的调试就是这样。我很少单步调试程序——那种方式更多是查看编译结果中不同部分的值。

我甚至没有Emacs中那些高级的、花里胡哨的东西。有些人喜欢使用。还有很多人习惯使用Visual Studio和Eclipse之类的IDE。我认为之所以妨碍别人采用函数式编程语言，一部分文化因素上的原因在于我们没有解决IDE环境的问题。这就像鸡生蛋、蛋生鸡的问题。现在鸡变得越来越忙碌了———函数式编程中还有更多有趣的东西。我希望那会激发我们对蛋的工作。为Haskell开发IDE有很大的工程量。即使使用Visual Studio或Eclipse作为外壳，要想做出真正平滑、正确无误的插件，也要做很多工作。

Seibel：GHC有一个“读取-求值-打印”循环，GHCI。你一般是交互式地编写Haskell程序吗？

Peyton Jones：实际上，大多数情况下我采取的方式是编辑并编译。但是其他人完全依赖于GHCI。

Seibel：说到测试，我认为函数式语言最好的一点是，当你在程序中调试一些小程序时，只需要指出输入是什么形式的就可以了。

Peyton Jones：嗯，对我来说，如果输入数据很简单，是可以那样做的，对我的程序来说可能不是问题。我的程序的问题在于，当GHC编译一些很大的输入程序时，得到的答案是错误的。

我认为测试对于编写属性是极为重要的，用QuickCheck检查属性是非常有用的——QuickCheck是Haskell中的一个库，可根据函数的类型生成随机的测试。但是我在想我为什么不多使用一下QuickCheck——那是一个很好的工具。我想这是因为给我带来麻烦的是那些很难生成测试数据的环境。不管带来多大麻烦，还是有很多人开发的程序让GHC以这种或那种方式产生奇怪的应答。那也是为什么要使用GHC的bug跟踪工具的原因。

所以一般情况下，我在开始的时候已经有一些出错的东西。也许编译器完全崩溃了，或是拒绝了一个不该拒绝的程序，或者产生了不够优化的代码。如果它产生的是糟糕的代码，我会查看编译管道中各阶段的代码，说：“这个地方看起来是好的，这个地方看起来也是好的。哼，这里出错了。出了什么错？”

Seibel：你是怎么去查看的？

Peyton Jones：GHC的标志可以让你采用一种很像是批量转储的方式，你会说："只要把各种东西都打印出来就行了。"

Seibel：是内置的打印语句调试？

Peyton Jones：是的。还有一点就是，它的结构像大多数的编译器：有顶层的管道之类的东西。如果在传递的过程中出现错误，可能会有点难以处理。但是我一般会用一些不太复杂的调试技术。只要给我看看这次传递之前的程序和传递之后的程序就可以了。啊，我知道哪里出错了。有时候我看不出来问题出在哪里，我会在各处放上几个不安全的`printf`语句，看看实际发生的情况。

Haskell有各种调试环境——有一个暑期实习生叫做Pepe Iborra，今年早些时候开发了一个很好的调试环境，现在放到GHC中了，有点像某种交互式的调试器。这个环境我现在用得还不多。部分原因是我们长时间以来一直都没有使用调试器，因为对于如何单步调试函数式程序，它不那么明显。

长期以来，如何调试函数式程序成为了一种有趣的研究。我们无法以简单明了的方式处理那个问题，这让人觉得有点儿尴尬，但也成为了一个有趣的研究问题。

很长一段时间我一般都只会使用非常原始的调试技术，用不安全的`printf`语句。我对此并不是很自豪。但是很长一段时间，我们都没有其他工具。至少对GHC来说，我制定了各种方法，那对我来说是完成工作最快捷的路径。

Seibel：听起来是个常见的故事。如果这么多人都使用打印语句来调试，会让你觉得是否有必要写一个更好的调试器。

Peyton Jones：但这取决于文化。在.NET平台上，人们愿意花上几十、几百人年来开发调试器，我认为这是不同性质的体验。为了让调试器更好地工作，确实可能需要更多的开发周期。但是如果把调试器放到.NET中，确实会得到一些非常有帮助的东西。

也许和你交谈过的大多数人更多是学术软件类型的。也许是在较少使用复杂的调试环境长大的。我不想得出一般的经验。我当然不希望轻视或是贬低一个好的调试环境的重要性。特别是在这些相当复杂的生态环境中，其中有很多很多层的软件。一套完整的.NET环境有多层的DOM和UML，我不知道还有什么乱七八糟的东西，相比之下，GHC是一个非常简单的系统。现

在世界是如此不确定，所以更多技巧上的支持可能真的是很重要。

Seibel：*另外一种得到正确软件的方法是使用形式证明。你认为形式证明（formal proof）的前景怎么样，有用吗？*

Peyton Jones：假设你宣布，对每件事情的目标都是通过机器检查证明其正确性。我们甚至不清楚这句话是什么意思。以机器的方式证明什么？证明的是一些规格说明书。那么你是如何编写规格说明书的？这就意味着规格说明书要包括程序所做的每一件事情。否则你无法证明程序做了每一件应当做的事情。所以对每件应当完成的事情都需要有一个正式的规格说明书。那么好啦——你准备如何编写规格说明书？你可能要用函数式语言编写。如果是这样，规格说明书可能就是你的程序。

我在这里说得有点快、有点松散，你可以在规格语言中描述一些在程序中不能表达的事情，比如："函数的结果是y，y的平方等于x。"对平方根函数来说，这是一个很好的规格说明书，但是可执行性不强。然而，我认为如果想要把程序应当执行的所有东西都说明清楚，那么得到的规格说明书本身会很复杂，甚至让你无法再确信规格说明书所表达的内容是你的本意。

我认为在实际生活中，写下一些希望程序具备的属性，这样的效率会高得多。你会说："这个阀门和那个阀门不能同时关闭。树应当总是平衡的。函数总是应当返回一个大于零的结果。"这些都只是很小的部分规格说明书，不是完整的。它们只是一些你希望是真实的东西。

应当如何记录这些内容呢？嗯，函数式语言对此相当擅长。实际上在写QuickCheck规格说明书的时候情况就是这样的。把属性当做Haskell函数写下来。比如说我们要检查那个`reverse`是自身的反转——嗯，可以写一个`checkreverse`，A的类型列表是布尔型的。所以`checkreverse xs`就是`xs`的反转的反转，等于`xs`。所以这个函数总是应当返回真值。这就是属性函数。但它是用同样的语言写的——很不错。

现在你可能希望对它做些静态检查。可能难，也可能简单。但是即使是把属性都以形式化的方式写下来也是有很大帮助的。可以生成测试数据来进行测试，这实际上也正是QuickCheck所做的。

所以，不要试图把程序做的所有事情都写成规格说明书，只写下部分规格说明书，我认为这样的生产率会高得多。也许是多个部分的规格说明书。然后通过测试、动态检查或静态检查来检查它们。你并没有证明程序是对的。你只是增加了信心，认为它是对的。我认为每个人所做的都是这些。

Seibel：你会定义很多属性，涵盖了你所关心的内容。然后根据实际情况，可以选择地确认那些特性是静态还是动态的。因为我们也许不知道如何静态地检查所有内容？

Peyton Jones：是的。但是在函数式环境中，你有更好的机会。但我们对于证明有点回避。不过，第一步是首先把这些属性都写下来。

但是我认为重要的事情是避免这种要么就包含全部要么就什么都没有的庞杂的规格说明书，可以在部分规格说明书上做一些有用的静态或动态测试。这些将增强你对程序正确性的信心，那可能是你所希望的。

你知道，即使是那些所谓完全的规格说明书也会有遗漏，必须在0.1秒内完成。或者是必须适合于10KB的内存。资源方面的内容常常没有包含在内。或者是运行耗时的问题。有无数多小的东西，意味着即使程序满足了正式的需求规格，可能也不像预期的那样运行。所以我觉得，如果说我们证明了所有一切都是完全正确的，那只是在开玩笑。最好就是认识到这一点并说我们的信心增强了——那正是我们所做的。开始的时候不用做太多——只投入5%的工作量，信心就提升了75%。这样不错。

Seibel：我们谈一点并发吧。Guy Steele让我问你一下："STM能够拯救世界吗？"

Peyton Jones：噢，不能。STM只靠自己的力量是无法拯救世界的。并发，更广泛地说是并行编程，是个多面兽，不是一颗子弹就能把它驯服的。说到并发，我认为它是多元化的。

人们很想说："使用一种编程范型来编写并发程序，很好地实现它，这样就够了。"只要学着如何使用那种范型来编写并发程序就可以了。但是我不相信这种说法。我认为对于某些编程风格，可能需要使用消息传递。对于另外一些编程风格可能需要使用STM。还有一些编程风格使用数据并行要好得多。在编程的时候，程序员需要处理的方式不止一种。

但如果你问的是，STM比锁和条件变量好吗？这两者才具有可比性。是的。我认为STM完全胜过锁和条件变量。所以还是忘掉锁和条件变量吧。对于多个程序计数器、多个线程、在共享多核的共享内存上快速移动：使用STM。但这是编写并发程序唯一的方式吗？绝对不是。

Seibel：我听到的对STM的一个批评是，当真正静下心来做它的时候，乐观并发并不像预期的那样，允许那么多并发。我认为人们在无法取得实质进步的时候，是很容易得出那种结论的。

Peyton Jones：你一定要当心饥饿（starvation）的问题。我最喜欢举的一个例子是，因为一个小的事务先到达那里并先落实了，导致后面的大事务总是无法落实。比如说一个在图书馆中重新摆放图书的图书管理员。他们开始乐观地重新摆放图书。他们已经完成了三分之二的工作，这时一名大学生过来借了一本书。嗯，因为图书馆的重新摆放还没有落实，所以这个学生的事务成功地落实了。图书管理员最后发现，啊，在重新摆放图书的过程中，图书馆又发生了变化，看到的布局和记忆中的不一样，所以要倒回去重新开始。

Seibel：*在锁和条件变量程序中，可能会出现另外一种情况——图书管理员会把图书馆锁上，在重新规划完成之前，谁也不能借书。你也许看到这个问题马上就会说："在把图书重新摆放好之前，不能把图书馆一直锁起来。"因为不允许借出图书，所以必须提出一种更好的锁机制。*

Peyton Jones：对。可以设置一个图书分馆或类似的设施——把常借的书放到那里，这样在你锁上图书馆主馆并重新规划或做其他事情的时候，大学生们还是可以借书的。现在必须考虑与应用程序相关的策略，并以某种方式表达出来。嗯，在两种情况下都会产生同样的问题——需要一个和应用程序相关的策略，这样在重新规划图书馆的时候不必将各种借阅活动挡在外面。在努力思考过需要做什么之后，就要将其表达出来了。表达的方法是什么？显然STM是一个上佳之选。在表达并行程序的时候，STM比锁和条件变量要好得多。

Seibel：*即使有人来查找最热门的第21本书的时候，我也不想把他们挡在外面，如果是这种情况，该怎么办？在现实世界中，可以想象一下，当某人借出一本书后，我们会补充一本后备书作为替代，然后图书管理员继续重新规划图书馆，不管那本书在什么时候还回来，我们都会把它放到那本后备书当前所在的位置。但是如果在STM的世界中，对图书馆的改动会让图书管理员重试他的整个事务。*

Peyton Jones：但是有些东西并没有改变——不管怎么说，那本书的关键字可以确保是没有改变的，对吧？有很多方法可以做到这一点。一种方法可以是：在用一本后备书替换它的时候，你根本没有修改图书馆——图书馆没有改变。改变的是图书。你没有修改书的关键字段——只是修改了它的值字段，也就是当前所在的位置。当书在其他地方时，索引是可以重新规划的。太酷了——可以很自然地表达这种思想。

通过STM，到了最后图书管理员可以查看他读取过的所有内存位置，看

看是否还包含同样的值。他读取的书的位置将包括图书的关键字段，因为这个字段决定了书放在哪里。但是他没有读取书的内容。他会说："啊，这本书——这个关键字段包含的值仍旧是73吗？噢，是的。"

但是我并不是说饥饿的问题无关紧要，因为它会带来一点潜在的问题。需要一个好的剖析工具，帮你找到无法提交的事务，因为它们不断地被拒绝而不是默默地什么也不做，所以能够得到它的一些反馈。基于锁的程序也是同样的情形。在出现这些沙漏的时候很让人讨厌。

Seibel：*我认为在基于锁的程序中，我们只是学会了如何尽可能短地保持锁，因为那样可以让竞争最小。*

Peyton Jones：对。但是，当然啦，也更难编程了。如果锁很细，就很难让它保持正确。我想这是STM的一个巨大优势，STM会给你非常细的锁，还有非常简单的推理原则。

STM会提供一种锁根本无法提供的推理原则。我会建立顶级不变量——有一些银行账户，这些银行账户中的金额之和是N。钱在不同的银行账户之间移动——就是这些。这是不变量。任何一个事务在开始的时候都要认识到这个不变量并在结束时复原这个不变量。如何推断这个操作呢？我们查看任何一个事务，比如说："从这个账户里取出3元，放到那个账户中。"好，不变量得以保持。这个推理是如何完成的呢？是纯顺序的推理。在描述了一些顶层不变量后，就可以完全独立地顺序推断每个事务了。

Seibel：*因为事务是隔离的。*

Peyton Jones：因为它们都隔离开了。所以说那是一个很强大的推理原则。因为尽管程序是并发的，仍旧可以使用有关命令式代码的顺序推理。需要建立类似那些顶层不变量的东西，但也是有好处的。因为这样就可以知道需要维护的内容是什么了。如果在事务中间得到一个异常，这种方式也很酷——因为在抛弃事务的时候没有副效应，所以不会破坏不变量。我觉得这真是太好了。关于性能问题的推理在另外一个层次上——你已经确保了某种正确性，现在要确保的是性能上不存在任何漏洞。那些事情更难实现——目前除了剖析工具和特定的反馈工具外，我不知道还有什么更好的方法。

Seibel：*我突然想到，虽然乐观并发在持久性数据库中时不时地得到使用，但是与基于锁的并发相比，它从未获得很牢固的基础。*

Peyton Jones：当然，STM是可以用各种方式实现的——乐观并发只是其中

的一种。你可以使用锁，这更像是悲观并发模型。

Seibel：但是也有一个原因，锁管理器是数据库最难的一个部分。

Peyton Jones：对。所以对于STM，你要确保一个人或一个团队实现了STM，其他人都使用它。你可以付给他们很多钱，找一间小黑屋子，把他们关上一年，让他们出色地完成工作。

但是接下来，通过一个简单的接口，大家就都可以使用那项成果。这正是我认为好的地方。我要避免的是让每个人的脑子都把它想上一遍。我在昨天的讲座中举的一个例子（来自Maurice Herlihy）是一个双端队列：插入、删除元素。

双端队列的顺序实现是本科生一年级的编程问题。通过在每个节点上加一个锁的方法来实现并发，这是研究论文所关注的问题。这个步骤太大了。有些事情会难到这种程度，太荒唐了。有了事务内存，就又变成本科生那个水平的问题了。只要在插入和删除操作上包裹一个"原子的"操作就可以了——任务就完成了。我觉得这真是令人称奇。这是质的差别。人们在实现STM时，必须确保将一系列的更改作为一个原子提交到内存中。如果只使用比较和交换的方法，是不容易完成这个事情的。能够完成，但是要特别小心。

如果有与饥饿相关的性能问题，那么可能需要做一些应用程序级别的思考，看看如何避免这个问题。但是接下来仍旧需要表达应用程序级别思考的结果，使用的还是STM。我认为这对于那种程序确实是一种飞跃。

我还要提另外一件事情——又要说到函数式编程了。当然，STM和函数式编程是没有任何直接关系的。STM是关于变换共享状态的——听起来并没有太多函数式的东西。

但实际上，我听过一次由Tim Harris介绍的Java中的STM的讲座。此前我从未听说过STM，只是碰巧去听了他的讲座。他讲述的STM中，有"原子性"，但是其他东西就不多了。你可以实现这些原子的事务。

我说："喔，看上去非常简洁。啊，这样的话需要在内存中记录每一个副效应。每个装入和存储指令。天哪，Java中有很多这样的东西。"但是在Haskell中基本上是没有的，因为它们发生在这个单子式的环境中。在Haskell中，装入和存储是相当明确的——程序员们认为这一点很重要。

于是我在想："噢，这将是一个非常酷的特性，我们可以试着在Haskell中复制这种原子内存。"我们就那样做了——我去找Tim谈了谈，问他该如何实现。没过多久，因为我们框架的类型特定——很纯、相当简朴的框架——我们发明了`retry`和`orElse`。在这种机制中，`retry`允许你在事务内执行分

块操作，orElse允许你在事务内部进行选择。这两种机制在Tim和他的同事们开发Java的事务内存时都没有出现，因为其他的环境都太复杂了。

他们对分块考虑得不多。也许他们认为你做分块的方式就是说："在这个谓词正确的时候只运行这个事务。"但那是很难组合的——假设你从一个账户里取出一些钱放到另外一个账户中，那么，在什么样的条件下可以运行事务呢？答案是：如果在一个银行账户里有足够的钱，在第二个账户里有足够的空间——也就是说两端都是有限的。于是条件就相当复杂了，很难弄清楚。如果还涉及第3个银行账户，情况就更复杂了。很难组合——必须查看方法内部，把它们的前置条件放到前面。

那就是他所面临的情况，对小程序还不错，但是显然，让人也不是很满意。于是我们在Haskell环境中提出了retry、orElse，后来这些机制又移植回到主流的命令式环境中，他们现在也忙着做retry和orElse了。太好了。

Seibel：*这么说，这个概念的产生并不因为Haskell中一些与生俱来的东西？只是因为你们想到了它？*

Peyton Jones：是这样的。大致说来，Haskell中糟糕的东西不多，所以能够产生一些很好的想法。在不失去抽象的情况下没有办法执行分块，真是让人讨厌。这让我们发明了retry和orElse。我认为函数式编程的最佳定义或角色可以被比作检查动物的一个实验室。想法还可以反馈回去。这个STM就是一个特别明显的例子，因为在两个方向都是一种转变。一个环路闭合了，我觉得相当好。

Seibel：*你认为程序员必须要读哪些书呢？*

Peyton Jones：嗯，Jon Bentley的《编程珠玑》是一定要读的。说到珠玑，Brian Hayes为《代码之美》一书写了一篇很好的文章，题目是"为'The Book'写代码"，这里所说的'The Book'，我觉得他的意思说程序要有一种永恒的美。给出两个点，然后再给出第3个点，需要找出第3点在那两个点连线的哪一侧。几个解决方案都不是太好。但是后来又提出了一个很简单的方案，正确地解决了问题。

当然，还有Don Knuth的《计算机程序设计艺术》系列。我觉得这不是能够一口气读下去的，不是那种书。在某个阶段我多次推荐过这套书。Chris Okasaki的*Purely Functional Data Structures*非常好。它像Arthur Norman的课程，只是整本书讲的都是这些。它讲的是如何可以不使用副效应，只要有好

的复杂性边界就可以做出队列、查找表和堆。非常、非常好的一本书，每个人都应当读一下。书不厚，很容易找到。还有《计算机程序的构造和解释》，Abelson和Sussman写的。我喜爱那本书。还有Andrew Appel 的*Compiling with Continuations*一书，讲的是如何使用连续传递风格编译函数式程序。也是非常好。

有些书对我非常重要，但是我很久没去读了：Dijkstra的*A Discipline of Programming*。Dijkstra在编写美的程序时非常小心。那些程序完全是命令式的，但是具备“Hoare特性”，也就是明显没有bug，而不是没有明显错误。而且它有一个非常合理的推理原则去推断那种做法。这本书第一次让我知道了以相当严密的方式推断程序。另外一本给我留下深刻印象的书是Per Brinch Hansen写的关于并发操作系统的书。我读了很多遍。

Seibel：*你现在还常编程吗？*

Peyton Jones：噢，是的。我每天都要写一些代码。实际上也不是每天，但我每天都会念叨着编码。有的人因为擅长某项工作而得到晋升，或者说变得越来越重要，可是随着地位的提升，他们就不再做自己擅长的工作了，我觉得这种情况真是太危险了。我之所以喜欢在这里工作、在研究领域工作，原因之一是我还能够从事我自1990年开始一直在做的编译器方面的工作。有很多代码，我熟悉这些代码的大部分。

我编写多少代码？有些天我会花上一整天写代码，实际上只是盯着代码看。有些天我什么都不做。所以，平均下来，一天是几个小时吧。编程真是太有趣了。怎么可能会不去编程呢？而且编程会让你保持诚实——可以好好地接受现实的检查，使用你自己的编译器并使用你提倡使用的语言。

Seibel：*你还和刚开始的时候一样喜欢编程吗？*

Peyton Jones：噢，是的，是的。那是最有意思的事情了。我想大多数程序员都是有一种“肯定有一种好方法”的感觉。在研究领域工作，很好的一点是不会有哪个经理站在我的身后对我说：“必须在本周完成这件事——一定要做完。”我可以坐在那里看着某个东西说：“肯定有一种正确的方法来做这件事。”

所以为了让程序正确，我会花上很多时间来重构、移动接口、编写新的类型，甚至把整个内容都重写一遍。GHC相当大——按照业界标准来看并不算大，但是按照函数式语言的标准来看是很大的——大约有80 000行Haskell代码，也许比这个数量还要多一点。生存期也很长——已经有15年了。大家

还在积极地开发它，这种情况也表明很多地方都重新写过了。没有哪个地方是不能动的。看到某个地方并想“做这件事的正确方法是什么？”这时候我感到既充满挑战又非常有趣。我常常在某件事上花了好几个星期都想不出来有什么好的方法。但是那件事情一直让人惦记着。因为肯定存在一种好方法。

Seibel：*在那几个星期，都发生了什么事情？*

Peyton Jones：噢，我在潜意识里思考着这件事。有时候必须试一试——有点像是沿着山坡往上跑。然后我就想起来它为什么这样复杂，常常会无意识地做一些其他事情。有时候我会往山坡上跑几次。有时候我会在背地里思考一下。有时候我在想：“嗯，时间到了——还是去做点事情吧。”也许并没有达到它应该达到的完美。

Seibel：*这是那种你在早晨醒来并说“啊，我找到了”的事情吗？还是你打算再跑一轮，然后就到达山顶了？*

Peyton Jones：更像是后一种情况。我很少会在早晨醒来的时候突然对一个不能理解的东西有所顿悟。作为研究员，你还可以有机会反思你所做的事情并把它记下来。所以常常地，在发生一些有趣的事情时，我会拿张纸把它记下来。举个例子，有一篇论文叫做“GHC内联函数的秘密”（The Secrets of the GHC Inliner），那是一篇完全针对实现而写的文章，描述的是我们对GHC内部结构的特定部分给出的一些实现技术，我们认为这些内部结构可能是其他人可以复用的。做学术研究可以有机会从代码中进行抽象，做上4次，最后得到有一个感觉不错的东西，然后写下来，这样其他人就可以复用同样的技术了。

Seibel：*编程对你来说意味着什么？你认为自己是一个科学家、工程师还是手艺人？还是说完全是别的什么？*

Peyton Jones：不知道你是否看过Fred Brooks写的一篇这方面的论文，叫做“让计算机科学家成为工匠”（The Computer Scientists as Toolsmith）。我最近重新读了一遍。非常好的文章。它让我们不会忘记我们做的工作是与构建相关的。我觉得正因如此，编程才有趣。

与此同时，我非常热衷于抽取出永恒价值的原理。我写过一篇文章，说的是如何编写好的文章或是做一次好的讲座，重要的一点是不要描述工作产品。工作产品的一个思想是实现。什么是思想呢，是思考出来的可以复用并

试图传递给听众的东西吗？有些东西对他们是有用的。从具体的工作产品中抽象出可复用的思想，我认为是学术上的事情。说到如何发现问题，有什么规律，现在还算不上是一门科学。但它是把真实生活中可以复用的想法抽象出来，我认为是非常重要的。

Seibel：*工程和手艺相比怎么样呢？我们希望像建造桥梁的人那样工作吗，最主要是不要让桥塌下来？还是说我们更像是制作陶器的人——只是我们的陶器特别复杂——你所能做的就是跟着师傅，向他们学习如何制作陶器？*

Peyton Jones：这种泾渭分明的想法是不对的。这并不是一件非此即彼的事情。有一件事情很难，即使对专业的软件工程师和开发人员来说也很难，那就是真正地理解我们所做的工作产品的大小。就像是从一尺见方的小孔中去窥视帝国大厦，很难真正地感受到那座建筑有多么雄伟，内部构造是什么。

GHC有多大呢？我对它的大小的感知和我对这座大厦的大小的感知是不一样的。所以我认为我们根本还达不到建筑桥梁的工程师那样。他们的设计模式现在表明了他们相当确定地认为桥梁不会倒塌。我们的软件还远远没有达到那个程度。但是这也不是我们可以一点儿都不关注它的理由。

实际上函数式编程在这个地方还是大有可为的。因为它从根本上允许你构建更为健壮的结构，构建易于理解、测试和推断的框架。我认为这里也正是函数式程序员们落后的地方：我们谈到了推断函数式程序，但是做的不是很多。我希望看到更多的工具，这些工具理解Haskell程序，从形式上推理程序，向你保证类型之外的东西。我们站在一个更高的平台上，我们应当能够走得更远。

刚才说的这些都是说材料应当变得更结实。你的材料越结实，你的注意力就越有可能放到大规模的架构上而不是细节的地方。但是这也会让我们更加雄心勃勃，构建一个更大的结构，直到达到极限，达到这个结构几乎要坍塌而又不倒的程度。

我认为这在某种程度上是一种不变量。你刚完成，但很快发现自己就到了临界点，再往前一步都是不可能的。我估计很难真正地看到这个点在哪里，是这里还是那里？总是存在一个非常微妙的因素，因为我们的雄心会膨胀。在工程结构中，对于你能够走多远，有一个物理上的限制。在最近的将来，没有人能够横跨大西洋建造一座桥梁。如果真建造一座，可能会倒塌。但这不是人们不去建造的理由——只是因为建造的费用太高了。然而今天，在软件方面，如果你在英吉利海峡上很快就建了一座桥，成本也不高，那么好了，

那成为既成事实了，我们想着既然成本不高，那么试试大西洋吧。如果是这样，就又行不通了。

Seibel：Guy Steele说摩尔定律在他整个职业生涯中都是正确的，他怀疑在他儿子的整个职业生涯中，这个定律不会再适用了。他还探讨了一些摩尔定律是否适用于编程的问题。我想我们最终是不是还得说："如果能够在英吉利海峡上建造一座桥梁，那么也能在大西洋上建造一座桥梁。"

Peyton Jones：不，不。我想软件是不同的。因为如果写一个10倍大的程序并不意味着要在一个10倍快的计算机上运行。程序计数器把时间花在一小部分代码上。程序90%的时间花在10%的代码上。所以说程序中性能关键的部分也许只是相对很小的一部分。

一般情况下你会无意识地一层一层地堆积抽象，在屏幕上按一下按钮，会接连发生大量的操作，最后总算可以让一些寄存器移动了。

也许我们必须使用尖端的编译器转换工具来破解那些多层的抽象，这样就不会发生那么庞大的连锁操作。抽象边界对人们也许有用，但是机器并不关心。我不认为因为我们也许会到达机器的边界，所以软件就一定要立马停止前进，不能变得越来越复杂。因为到那时机器的速度需要变得相当快。我认为软件最主要的限制不是计算机的速度，而是我们理解自己应当做些什么的能力。

Seibel：你喜欢的是编程的哪个方面？

Peyton Jones：对我来说，编程之所以有趣，部分原因是能够试着写一些能够真实完整地体现出智力水平的程序。你可以随意地写一个程序，这样的结果是，写出来的程序尽管在很长一段时间内差不多都能运行，但并不令人感到十分满意。所以我认为出色的程序员一个优秀的特征是试着找出一个漂亮的解决方案。不是每个人都能够放任自己，因为想不出来一个漂亮的解决方法，就可以不完成今天的工作。

但是因为它是如此灵活，所以我真的认为那是一个有趣的技巧。凭借这种技巧你真的可以做任何事情。但这样也意味着你既可能做出丑陋的东西，也可能做出漂亮的东西，也可能做出完全不可维护、无法持久的东西。我有时候对于商业世界感到有点恐惧，一方面，因为客户需要在下周拿到软件，所以必须要完成，而另一方面，我们构建的软件只有广度、没有深度。

系统中遍布着各种东西——为了构建一个ASP.NET网页服务之类的东西，需要了解这个API、那个工具，需要使用3种不同的语言来写，需要知

道Silverlight和LINQ，这种类似的事情可以做个没完没了。每一种工具都有一本厚厚的书来讲述内容。

这是一种我不知道该如何解决的紧张局面。这些都是有用的系统——不是随随便便设计的。每个系统都有存在的道理，每个系统都有一个聪明人，努力地思考着这个系统的架构是什么样的。但是，每个系统都各自有一个宽泛的接口。可能很深，也可能不深，但是肯定都是很宽泛的。脑子里要装很多的东西，就好像学习一门语言——我说的是人类的语言——有很大的词汇量。

对我来说，这不是什么乐趣。我从来没有学过乘法表，我每次都是利用基本原理导出乘法结果，我已经锻炼出足够好的技巧，能够快速算出。如果要计算7乘以9，我的算法是，7乘以9就是7乘以10再减去9，等于63。但是其他人都学过乘法表。这实际上是小事一桩。我讨厌的是必须学习那些大东西。我本能地避开那些乱七八糟的东西。不过同时我也承认那些东西在实践中是有用的、重要的。我的想法是，如果你在设计这些东西的时候能够多花点时间，是不是可以把它们设计得小一点，不要那么复杂，不要随意设置那么多接口？

Seibel：*有时候完全是因为这些东西的每一部分都有一些聪明人在做，每个聪明人都要一点自己发挥的空间，所以事情就变得非常复杂了。*

Peyton Jones：我相信是有这方面的因素。但是如果愿意做一个更积极的架构，你会发现这是一个很大的、复杂的世界，有很多事情要做。如果你有奥林匹亚山神那样宽广的视野——如果有一个很大的大脑和广阔的思路，那么做事情的时候才可能重合的地方会更少，整体才更具一致性，但这是不现实的。

实际上我们必须把这些问题拆分成小块。这些小块的每一个都有人照管，他们会受到以前做过的事情和他们传统做法的影响。也许他们在那个小块内设计的一些东西不像它所应当达到的那样好——他们的时间太紧张了。因而当你看到所有这些小块组合到一起的时候，你会发现它们比可能达到的完美程度要低得多。在你知道这种情况之前，你已经为遗留问题所困——这是事物没有达到它们能够达到的完美程度的另外一个原因。

我们周围有非常大量的遗留问题。这是Haskell好的一个地方。有一次在回顾Haskell的时候，我记得那是在POPL 2004上，我的幻灯片中有一页说我们在Haskell中学到的经验之一是“不惜一切代价避免成功”。这显然成为了

一句格言，因为人们记着那个句子，又当着我的面引用了它。

这句话中有一点真理，意思是不要过度成功、不能成功得太早，只有这样，我们才能在Haskell的生命过程中不断地对它做出很多改动。可是现在我感到有些忙乱，因为Haskell已经变得过于成功，所以我收到越来越多的缺陷报告、特性请求。越来越多的人说："请不要破坏我的程序。"这些事情在以前可是很少出现的。

Seibel：*你几次提到了编写美的代码。美的代码有什么特征吗？*

Peyton Jones：Tony Hoare说过一句很精彩的话，他说代码应当明显没有bug，而不是没有明显的bug。所以我认为美的代码是明显正确的代码。在某种程度上应当是透明的。

Seibel：*那些珠玑代码怎么样？你几乎要费很大力气才能看出它们的工作方式，但是一旦看出来，发现很让人称奇。那些代码也是美的吗？*

Peyton Jones：有些时候，说代码明显是正确的，并不意味着你不需要经过一番认真思考就能看出它的正确性。也许需要深入理解才能分析出代码为什么是正确的。看一下AVL树的代码，如果不知道代码想要实现的是什么，你真的是一点头绪都没有，不知道为什么要做那些旋转。但是一旦知道了它所维持的不变量，你就会发现，啊，如果保持那个不变量，就可以记录查找时间了。那么再去看每一行代码的时候，你会说："啊，是的，代码保持了那个不变量。"正是这个不变量给了你洞察力，看到它你才会说："噢，代码显然是正确的。"

只看代码是不够的，这点我完全同意。我认为这不是美的代码的特点，你无法只看代码就知道它为什么是正确的。你也许需要被告知为什么。但是当你知道原因后，有了这样的观点，有了那个不变量，理解了发生的事情，这时你才会看到，噢，是的，那是对的。

Seibel：*这会给程序设置一个上限吗？软件可以大到什么程度仍旧是美的？*

Peyton Jones：我不知道程序的大小有没有边界。为了让自己确信软件是正确的，或者至少是接近正确的，所需要的理解分析与你对程序正确性的自信是相关的。任何一个非常非常大的软件肯定会有缺点，或是你所知道的所有事情都是有问题的。但是此刻全部修复是不经济的。GHC就是这种情况，微软的软件肯定也是这样的。

但是为了让大软件是可管理的，那么，对于它应当做的事情或是应当为

真的事情，就应该有一些全局的不变量或全局语句。比如GHC，有这样一个不变量，每个中间程序都应当很好地定义类型。实际上，如果你愿意，在运行时是可以检查的。对于发生的事情，那是一个相当强大的不变量。所以我不是很确定，是否一定与大小有关系。

当然，程序内的相互联系最终会让大程序因为自己的庞大而不堪重负。有些时候，做研究工作的一个好处是可以拿上一段代码，按照自己新的理解来重新编写一次，把代码写成你试图实现的东西以及试图实现的方式。我们讨论过重构GHC后台的事情。如果在一个更加商业化的环境中，我可能没有办法做重构了。但是长期来看，我希望GHC可以变得更容易维护、更容易理解。

大小有没有一个上限？我不知道。我认为只要我们能够在好的抽象上进行构建，就可以在大西洋上不断地把桥建下去。我们有可以使用的软件——不是很完美，但是考虑软件的大小，做成那样也让人称奇了。

Seibel：问题也就是说，能不能建造一个又大又可行，而且还很美的庞然大物？

Peyton Jones：很难保持它的美。刚开始建造的时候常常很美，或者至少是不丑的、可以接受的。但是随着时间的推移，可维护性是很难得到保持的。对于使用期很长的程序来说，这是最糟糕的事情……它们慢慢就变丑了。所以不是某个时刻变形，而是在一定时间后就变得很糟糕了。

Seibel：唯一能够采取的方式是："嗯，这个软件存在的时间够长了，重新编写吧。"

Peyton Jones：我认为这个软件的大部分最终是需要重新编写的。如果你能够在进展的过程中一点一点重新编写，那么可以这样做，但如果你在10年间都没有做任何东西，那么结果就令人望而生畏了，你会想："我只能扔掉它，重新编写。"如果你能够在进展的过程中一点一点重新编写，那就太好了，就像人脑细胞再生一样——我希望GHC就能出现这种情况。

我认为作为程序员，生活中最糟糕的事情是，面对别人的一堆代码不敢动手去改，或者更糟糕的是，面对自己编写的代码也不敢动手去改。真是太让人泄气了。

（编辑：陈兴璐）

8

Peter Norvig

戴玮 译

Peter Norvig是一位渊博的思想家，但在骨子里，他也是一名黑客。他写过一个程序，能从Google的搜索日志中找到同一用户连续3次搜索输入的关键字，而且这3个关键字还恰好能连成一首俳句[1]。这其中，我印象最深的俳句是："Java ECC / Java elliptical curve / playboy FAQ"。

他的个人网站上，页面链接有些平淡无奇，比如他所著的书和论文、演讲时用的幻灯片，还有他写的各种小程序；但也有不少链接很有意思，比如他在*McSweeney's Quarterly Concern*[2]发表的短文[3]，他对如何编写世界上最长回文[4]生成器的精妙解读，以及他为讽刺微软的PowerPoint软件而作的"葛

---

① 俳句是日本的一种古典短诗，由十七字音组成。"Java ECC/Java elliptical curve/playboy FAQ"正好有17个音节。——编者注

② *McSweeney's Quarterly Concern*是一本在美国颇受欢迎的文学期刊。——编者注

③ 可在http://www.norvig.com的Humor一栏里找到。——编者注

④ 回文也叫回环，即把相同的词汇或句子在下文中调换位置或颠倒过来，产生首尾回环的情趣。——编者注

底斯堡PowerPoint演示”——Edward Tufte曾引用过这份文档，而且如果你在Google搜索“PowerPoint”，它还会出现在搜索结果的第一页中。

如今他担任Google的研究总监，在这之前也曾担任Google的搜索质量总监。加入Google前，他是NASA Ames研究中心的计算科学部主任。再往前追溯，他还是上世纪90年代末诞生的互联网创业公司——Junglee的创始员工。2001年，他获得了NASA杰出成就奖。他还是美国人工智能协会（American Association for Artificial Intelligence，AAAI）和计算机协会（Association for Computing Machinery，ACM）的成员。

从Junglee到NASA再到Google，Norvig体会到了“黑客”和“工程师”开发软件的不同方式。在这次访谈中，他指出这两种方式各自的优劣所在。曾经是计算机教授，如今任职于全球最大的软件企业之一，因此他对计算机科学学术研究和行业实践之间的关系的看法也生动有趣。

我们在访谈中涉及的话题还包括：近几年编程发展得如何；为什么设计技术无法弥补需求分析不充分带来的缺陷；假如NASA采用可靠性较低却更廉价的软件，其处境为什么较之当下或许会有所改善。

Seibel：你什么时候开始编程的？

Norvig：在我上高中的时候。当时我们学校有台计算机，我记得好像是PDP-8。我参加了一个学习班，开始用BASIC编程。就这样，我开始了编程之路。

Seibel：那是哪年的事？

Norvig：我74年高中毕业，所以那年应该是72或73年。回想那段时光，有两件事让我记忆犹新。我记得老师教我们写洗牌程序，她的算法是：用随机数产生器选两张牌，将它们互换，并用位向量来记录换过哪些牌。重复这一过程，直到每张牌都换过一次。我当时的反应是：“太傻了，这一定是世界上最傻的算法。它可能会运行个没完没了，因为或许有些牌怎么选都选不着。”我当时掌握的算法知识还不足以让我看出，这算法的复杂度本来能达到$O(n)$，却被她给弄成了$O(n^2)$。不过我看得出，它就是不太对劲。于是我琢磨出一个和Knuth算法一样、复杂度为$O(n)$的算法：先从0到52交换，再从0到51交换，以此类推。我记得老师在听过我的算法后，还是认为她的算法更好。从此我觉得“或许我天生就擅长编程”，也明白了“或许老师也不是无所不

知的”。

Seibel：她刚教你们那算法，你就马上回应“呀！这不对”，还是你琢磨一阵才说“啊，是不是应该还有更简便的方法”？

Norvig：我觉得自己当时是马上发现问题的。现在很难回想当时我到底是怎么想的了，但我确实马上就注意到，这种算法可能会无法终止。我当时大概还不清楚像运行时间的期望值这样的概念。

我还记得，我在阁楼上找到几本我爸以前买的《科学美国人》(*Scientific American*) 杂志，并把它们全部翻了一遍。其中有一篇Christopher Strachey写的关于软件工程的文章，宣称高阶语言 (higher-order language) 即将流行。他还发明了一种没有任何编译器支持的高阶语言（学术性语言），然后说："我要用它写个国际跳棋 (checker) 程序。"我读过那程序。在学校，老师只教过我们洗牌之类的，因此它成了我见识到的第一个重量级程序。最近我又读了一遍，但一读就发现其中有个bug。这真是不得了。要知道那可是Christopher Strachey啊，他写这样的程序自然是驾轻就熟，而且《科学美国人》也会有编辑之类的人把关，他们都有机会消灭那个bug，却都没有。bug是这样的：文中有个make-move函数，以棋盘位置为参数执行一次移动；可在文章对应的代码中，该函数除棋盘位置外，还带了另一参数。这显然是因为他们先写文章再编程，但编程时发现搜索深度不能无限增长，所以又加上个控制深度上限的参数，保证递归下降过程可在某层终止。不过他们忘了回头检查一下文章，把文中的参数也改过来。

Seibel：这么说，它是你读过的第一个有意思的程序。那你写过的第一个有意思的程序是什么呢？

Norvig：我想应该是"生命游戏" (the Game of Life)。它其实是我们上课时留的作业，我很快就完成了。那时候，我们连一台像样的显示器都没有。我有的只是一台纸张发黄的电传打印机，而不是30英寸的显示器。我琢磨着："这么大一张纸上打印这么小一块地方（差不多10×10），再一代接一代地打印出来，这太浪费了。"于是我又想："干脆一口气并排打印出5代得了。"我知道BASIC没有三维数组，而且我甚至不能用太多二维数组，那样会耗尽内存，所以我得想个办法用5到6个二维数组解决问题，这时我想到了位域。

Seibel：看来就算内存受限，你还是能玩转这么多数据。你是在某人指点下领悟到位数组的使用方法，还是翻阅手册时突然发现"啊！快看，这里居然有PEEK和POKE语句"，还是有其他原因？

Norvig：是这样的，我在位域的每一位上存储0或者1时，还要找地方存储其他内容。于是我想："哎，把其他数据也存进去算了。"实际上，我甚至记不清自己是否真正实现了位存储。我可能用的是十进制而不是二进制，因为我觉得二进制看着不太顺眼。我还要在程序里添加一些循环，确定一次循环要包含几代，因为仅保存之前一代完成不了这游戏。

Seibel：*当你作为一名程序员崭露头角时，你有没有刻意去做一些事情来提高自己的编程水平？还是说你只是通过不断编程来提高？*

Norvig：我觉得我就是不断编程。当然，如果是有意思的事情，我也会去做。特别是读研那会儿，如果课程不太紧的时候，我喜欢说："哈，这问题很有趣，我倒要看看能不能解决它。"这并不是说它对我写论文有帮助，而只因为它有趣。

Seibel：*虽然你上大学时研究过计算机，但计算机并不是你的专业，对吗？*

Norvig：我刚入学时，计算机课是由应用数学系开的。我毕业时，学校设了计算机科学系，但我还是一直把数学作为自己的专业。那时计算机专业学的那些必修课就像在主修IBM专业[①]，你得学他们的汇编语言，还得学他们的360操作系统，等等。我对那种事情兴趣不大。我会选修一些我喜欢的计算机课程，但我可不想从头到尾学上一遍。

毕业后，我为剑桥的一家软件公司工作过两年。两年后我想："学校生活四年才让我感到厌倦，但工作两年我就厌倦了。也许我喜欢学校是喜欢工作的两倍那么多吧。"

Seibel：*你在那家软件公司都做了哪些工作？*

Norvig：他们的主要产品是一套软件设计工具集，同时也开展不同行业的软件咨询。其创始人来自剑桥的Draper实验室，参与过阿波罗计划和其他一些类似的项目。他们是空军的关系户，也经常接政府的单子。他们提出过一个改进软件设计的设想。我从不相信那个设想真的管用，不过它非常有趣。

我记得我在这家公司做过一个项目，内容是编写一个绘制流程图的软件。其思路是先分析某人给出的程序，再生成该程序的流程图。这思路相当靠谱，因为大家都这么用流程图。其实按理说，编程之前就应该把流程图画出来，但实际上根本没人这么做，都是完事以后才画。这个软件的高明之处，

① 主修IBM是在某些大学开设的一系列课程，由IBM赞助。课程讲授的内容以IBM所需技能为主。后文还会提到Norvig对主修IBM的看法。

在于它有一种不完全语法机制，能转换语法不完全正确的程序，忽略无法分析的代码。本来它还应该懂得如何分析if语句，因为if语句会产生代码块分支什么的，但项目组有人反对说："不管if语句有什么，都塞进一个代码块里算了。"我们签下了这个软件的合同，不过他们明确表示想在Unix系统上运行，于是我们向MIT借了台计算机，还用上了所有的Unix工具，比如yacc什么的，来编写这个软件的编译器。然而，就在最后关头，他们改口说："不对，我们要把它安装到VMS系统上。"真是个晴天霹雳，一下把yacc给劈没了。但我们想："也还好吧，我们现在用不着它了。我们只用它生成表，这事早就办妥了。"

Seibel：只要你们不再改变语法，就不会出什么问题。

Norvig：没错。我们就这样交付了软件，他们也很高兴。然后，很自然地，语法改变了。而且，我们手头再也没有任何Unix系统的计算机。于是我们落得这种下场：边凭借对表的理解给语法打补丁，边说着"这跳转不太对，我要引入一个新状态，把这跳转到达的状态转移到这个新状态上"。

Seibel：这真的是最合适的解决方法吗？你们想没想过只要重写一个分析器就行？

Norvig：我当时很可能那么想过，但你知道，那只是个小改动。

Seibel：他们每3周就要求改动一次语法，你没陷入这个泥潭中？

Norvig：嗯，后来我离开公司去读研了。他们的确遇到了这样的问题，但我不清楚后来发生了什么。

Seibel：你是彻底解脱了。后来你获得了博士学位。回想当初学编程时，有什么让你后悔的事情吗？

Norvig：我最终还是进入产业界安身立命，所以也许应该更早地深入接触计算机产业。我的确尝试这么做过，但我在学校和研究生院里待的时间太长了。不过学生生活一样乐趣无穷，所以没什么可后悔的。

Seibel：对于工业编程（industrial programming），你觉得哪些东西是你必须学到的？

Norvig：你得有一个进度表，还要哄团队成员、客户以及经理们高兴。当你还是研究生时，你用不着做这些，你要做的只是偶尔见见导师。

我觉得最大的变化是你从自己编程变成了与团队协作，你必须学会如何与人打交道。通常你在学校是学不到这些的。可能有些学校已经着手把这些内容更多地纳入到课程体系之中，但在我上学那会儿，团队合作是作弊的代名词。

Seibel：*对于将要进入产业界的人来说，除了编写代码的能力之外，还应该发展哪些技能？*

Norvig：人际交往是最重要的。你要充分理解客户的需求，这包括两方面内容：了解他们想要的是什么，了解你们现有的是否适合他们。你还要做到与团队成员协调一致、合作愉快。而且对内你要做到与领导交流无碍，对外还要做到与客户沟通顺畅。社会关系错综复杂，处理它们的技巧也各不相同。

Seibel：*和以前相比，现在的编程更像是一种社会活动了吗？*

Norvig：是的。以前的计算机几乎与世隔绝。那时候，计算机通常用来做批处理，所以界面极其简单，也使得瀑布式设计成为可能。你可以这样设计："输入是一组卡片，输出是由几列数字组成的报告，其中这列数字代表了什么什么，那列数字又代表了什么什么。"

不过，这种进行了明确规定的设计可不是什么好方式。你应该从一开始就和客户展开大量的互动，而不是采用这种看上去与客户隔阂很深的设计。现如今，没有什么是一成不变的，而且什么都讲究相互配合，所以这句话也就更合情合理了："与其一下子全设计好，不如请客户过来一起探讨。"

Seibel：*当你认识到团队合作与单打独斗之间的区别时，有没有一种茅塞顿开的感觉？*

Norvig：我没觉得有多茅塞顿开，但我的确领悟到一点：在团队中，你不能什么事都亲力亲为。比如说，你尽可以去了解各种各样的程序设计方法，但你还是做不了设计，至少我当初是这样的。你还得指望专门负责设计的人提取出合适的抽象，然后你再去用现成的。这时候，你应该开始考虑这样的问题："最合适的抽象是什么样的？"而不是想："我知道这抽象是什么样的，因为是我以前做的。"如果让你负责设计，你会怎么做？你当然希望设计人员得到的抽象和你设想的一模一样，但如果不一样的话，你也要弄清楚原因是什么，然后想想如何运用他们的设计方法。

Seibel：*以这样的方式进行团队合作，就算团队成员来自不同的年龄层，你也能有足够的精力去处理那些最重要的事情吧？*

Norvig：你说得没错，我正是在那些不断成长的年轻程序员身上发现了这点。和以前相比，现在的编程还有另一个不同之处：以前是从零开始，什么都得自己动手写，而现在，你只要把各种模块组合起来就行。现在的学生在完成编程作业时会说："好，我要做个网站。我可以用Ruby on Rails实现这些功能，还可以用Drupal完成那部分。我用Python编写脚本，还下载了能统计网站数据的函数。"利用脚本语言，我们可以把这些模块组合起来，而不是什么都自己写。因此我觉得，理解模块的接口和组合模块的方法，比完全理解它们的内部细节更重要。

Seibel：*在你看来，现在能获得成功的程序员是否跟以前不一样了呢？*

Norvig：我认为真正获得成功的程序员还是和以前一样，至少在我周围是这样。不过我得承认，现在想在最短时间内快速了解所有必备知识的人比以前多了，想彻底理解这些知识的人少了。有些人喜欢虚张声势，他们大声宣布"我要开始写这程序了"，而且敢对别人说："我不知道这程序究竟该如何运行，但我通过查阅文档找到了这3样东西。我测试了一下，它们还挺好用。有了它们，我就能继续。"这样做能带你到达某个境界，但称不上是真正优秀的程序员。仅靠这样还不够。你要多了解一些东西，还要不断自问："我写的这部分程序安全吗？有没有在这种情况下失败的先例？我确实用过它一次，而且还挺管用，可它能一直管用吗？我怎么用测试用例证明这一点，并且更深入地理解它？我完成这程序后，由于我是通过某种方式把这些模块组合在一起的，因此，我能不能把我所做的这部分程序提取出来，再把它作为人人都能使用的新工具发布？"

Seibel：*当你还是一名程序员时，你觉得团队合作是怎样一种感觉？把问题化整为零，让每个成员负责其中一部分，这是更好的方法吗？你喜欢什么都结对编程以及每个成员都共同拥有全部代码的极限编程模式吗？*

Norvig：我觉得那样把问题搞复杂了。Steve Yegge写了篇文章"好敏捷，坏敏捷"。我觉得他的观点基本上是正确的。十分之一的时候大家坐下来一起讨论，的确不赖。这样做可以充分地交流彼此对事物的理解。但我认为绝大部分时间都这么做未必效率很高。

如果你同时拥有两名优秀程序员，最好先让他们各自独立工作，然后再互相给对方排错。免得有人说："我们多花了一个人的钱，但却没多干一个人的活。"

如果你们还在思索下一步的任务，包括将要解决哪些问题或者软件具备

哪些功能，而且希望大家畅所欲言各抒己见，那就确实有聚在一起讨论的必要。在真正开始编程之前，你们甚至不可能知道它最后会成为一个什么样的产品，这正是你们要聚在一起讨论的问题。这问题解决后，你还会说："好，我们现在知道要做些什么了，可我们该如何分工呢？"这也是你们聚在一起要讨论的问题。等到你们经过反复讨论，提炼出一个相当不错的想法，我觉得最好各自分开，用剩下的大部分时间来实现。你们希望得到一些反馈，所以要把每段代码都提交给别人审阅。但大可不必实时进行，不必一边写代码就同时请别人审阅。

我想起IBM所倡导的编程大师思想了，它是我所说过的最愚蠢的事。居然有人心甘情愿地让资深程序员呼来唤去？

Seibel：*我很惊讶，你会觉得以编程大师为典范的思想如此愚蠢。你在"用十年时间自学编程"（Teach Yourself Programming in Ten Years）一文中，详细阐述了编程何以成为一门技艺。在你看来，编程就像其他技艺一样，要经历十年左右时间磨练，才能成长为真正的大师。很多行业都存在某种等级制度，把从业者划分为大师、工匠和学徒。尽管可能不会有人心甘情愿地想当学徒，但如果有些人已在这行业里摸爬滚打了10年之久，他们与刚毕业的菜鸟分工不同也是顺理成章的事。*

Norvig：做学徒最棒的一点，就是能观察到大师的所作所为。我希望编程行业更多地出现这一现象。我想，结对编程用在这方面也许效果不错。我真觉得这是好现象，如果你经验不足，那就看看那些经验丰富的人是怎么做的，特别是那种课堂上没怎么教过的技术，比如调试技术。在学校，人人都会学到算法什么的，但没人正经学过调试也少有机会看别人怎么做，然后惊叹道："哇！我从没想过居然还能这么做。"这真是太有用了。

不过，之所以会有大师和学徒这样的关系存在，我认为一个重要的原因是材料稀缺。当金匠打造金器时，金子总共就那么多，或者换个说法，当外科医生做手术时，手术台上的病人就只有一颗心脏，所以主刀的肯定是经验最丰富的医生，其他医生只能给他打打下手。但编程不同，你有大量的终端，也有大量的键盘，不需要搞什么定量配给。

Seibel：*你刚才提到，有些知识在课堂上没怎么教过。你既做过学术研究，也做过技术，你觉得计算机学科的教学与实践之间是否存在脱节的现象？*

Norvig：这问题太关键了。我认为现在的计算机学科里，称得上鸡肋的课程不多，几乎每门课程都能让你大长见识。不过话说回来，虽然这些课程都很

有用，但仅仅凭借它们，还不足以让你在产业界取得成功，也不足以让你能构建一个完整的系统。我认为大多数学校更新课程的速度都太慢不适合产业发展需要。比如产业界里，团队合作的精神早已深入人心，学校却仍然不怎么传授这方面的知识。同样，“搭积木”的思想也不怎么教，好在我们小时候都玩过积木，所以不教问题可能也不大。在Google公司，我们关心的是大规模云计算、并行计算之类的，可学校教得也不多，尽管他们对此也很有兴趣。总的来说，我认为，计算机学科的教学有些滞后，但还是有用的。

Seibel：*那么学术界是否在某些领域上远远领先于产业界？产业界可能会忽视一些关于应该如何构建软件的优秀思想。*

Norvig：确实存在一些这样的情况。模型校验（Modeling Checking）也许就是最合适的例子。英特尔公司刚开始没怎么注意到模型校验的重要性，结果他们的芯片出现了一个乘法功能的bug，对问题芯片所采取的大规模召回行动让他们损失惨重。后来，英特尔开始关注这方面。他们找到一位学术界专家，问他：“你能帮我们改善这一问题吗？”专家真的帮了他们大忙。如今，模型校验作为英特尔工艺流程的重要组成部分，发挥着不可或缺的作用。所以对我们谈的这个话题来说，这是个非常合适的例子。在编程语言方面也有些类似的情况，但可能不明显，学术界在这方面做了很多工作，但对新编程语言没有起到多大的实际影响。操作系统方面有一点儿。过去几年，我们一直在资助伯克利的可靠性自适应分布式系统实验室（RAD Lab），那个实验室里有Dave Patterson等人。他们想出了一些关于如何生产可靠系统的好方法。不过在产业界里，人们面对的是规模更大、更为艰巨的问题。他们也许无法解决所有的问题，但与处于大学环境中相比，产业界会解决得好一些。

Seibel：*所以在你看来，产业界其实并没有因为抗拒某种形式的变化而不接受学术界的好想法？但据说那些自学成材的PHP程序员永远不会对Haskell感兴趣，尽管它也许更有利于编程。*

Norvig：我只是比较怀疑。如果真那么有好处，他们会接受这些新事物的。我虽然觉得并不存在一个完全信息市场，无法立即利用各种因素来实现最优解，但现在的情况也差不太远了。学术界也许没有完全看清产业界正在面对的问题。这里有一部分属于教育问题，如果你手下有一大帮程序员既不理解“单子”（monad）到底是什么，也没有学过范畴论，那就确实存在一个差距。

还有一部分是遗留系统的原因。我们有很多遗留系统，但又不能就这么一股脑地把它们扔进垃圾堆，因此系统转换应运而生。有这样一种关于系统

转换的观点，“我们应该规划一下十年之后的系统会变成什么样。当然，我们现在不能直接把系统转换成十年后的样子，但我们仍然要具有前瞻性。到那时，系统会与现在大不相同，那我们又该采取怎样的方法，把现在的系统转换过去呢？”我认为，我们一定要重视这种观点，产业界应该更有远见。

不过有些人想在具有重大影响力的领域取得一些进展。但我认为很多情况下，程序语言考虑的问题可能都太底层了，无法达到设计者当初所预想的影响力。因此，如果有人宣称，“看，用我全新设计的语言编程，原来6行代码实现的功能现在两行就能完成。”这当然很妙，也许会让你的编程效率更高，还会让程序更易于调试和维护。不过不要忘记，在整个生产系统中，你所编写的程序也许仅仅是一个小环节。真正让你头疼的事，或者是每天更新数据，或者是采集网上数据并纳入系统，或者是以正确的格式存储数据。因此你必须牢记，当前你所面对的问题只是全部问题的一个非常小的部分，这同时也意味着进行改变存在很大风险。

Seibel：*让我们暂且把编程语言的话题放一放。你是否认为计算机教育已经有了长足进步？你说你那时学计算机就像“主修IBM”。*

Norvig：是的。现在已经有了很棒的课程，但令人郁闷的是，许多学生现在都不学计算机了。学生人数在不断减少。毫无疑问，有些人发自内心地热爱计算机或者计算机设计，他们决心为之奋斗终身。我们必须牢牢抓住这一群体。然而现在大量最优秀和最出色的人才要学物理、生物什么的，因为它们是最热门的领域。此外，还有很多人会说：“嗯，我对计算机有点兴趣，不过这行业前景黯淡，什么工作都外包给印度了。所以为了就业，我还是进法学院读个预科什么的好。”这太可惜了，不实消息对他们造成了误导。

Seibel：*你是因为很多人本来能从程序员这份职业中获得乐趣，还是我们这个行业需要他们？*

Norvig：都是。有很多人能从不同的事物当中获得乐趣。如果他们对两种职业有一样的乐趣，那我不会强求他们非得从事计算机。但我认为不是这样，有些人喜欢计算机却没有学这一行。我们需要更多的优秀人才，他们会对这个世界产生深远影响。如果想做到这点，平心而论，我们理应得到比现在更多的顶尖人才。

Seibel：*在Dijkstra的一篇论文里，他谈到计算机何以成为数学的一个分支，以及为什么计算机系学生不该在学习计算机的最初几年中直接接触计算机，*

而是应该学习如何操作形式化符号系统。在你看来，如果想成为一名合格的程序员，学习多少数学知识是必要的呢？

Norvig：用不着有Dijkstra那么高的水平。而且他关注的数学比较特别，是一种离散的、研究逻辑证明的数学。我想现在这个领域中，逻辑的重要性不如过去了，概率开始变得重要。我几乎找不到任何一个能证明其正确性的程序。

Google正确吗？嗯，输入这些关键字，你得到了10页结果。如果Google崩溃，那肯定是不正确的。但如果Google给你这10个链接，而不是那10个，谁又知道到底哪个是正确结果？你可以主观判断哪些链接是你想要的，但也就这样了。在我看来，逻辑证明与人脑处理问题的过程大不相同。如果让你去解决搜索问题，或者解决无人车导航的问题，让它开在街上撞不到人，那你肯定会把逻辑证明远远甩在脑后。

Seibel：*那么对优秀程序员来说，有哪些基本技能是必须具备的呢？虽然显而易见的是，编程面向的领域不同，需求也就不同，但从根本上说，不同领域在编程方面是否存在一些共性？*

Norvig：你必须有所进展，还得能加以改进。这些就是你生活所需的全部能力。首先，你必须对某个方向有一定了解，然后决定，"这就是我前进的方向，"再然后，你还得说，"现在我要开始改进它。"改进的结果可能是，"我理解得不太对，有些情况我没有处理，"还可能是，"既然我对它的理解更深了，那我要写个工具让它变得更抽象，这样下次我再编写类似系统时，就会比较轻松。"内省到如此程度之后，你会自问，"我下一步要去哪儿？通过何种方法到达那里？有更好的方法吗？"

Seibel：*先做出来，再调试它，然后不断重复这一过程——你认为这种技能是许多人甚至是不以程序员为最终职业的人应该学习的思维方式吗？如果由你制定小学、初中和高中课程，你希望所有学生接触这种编程思想吗？还是说，它只是一门编程行业的专业技能？*

Norvig：我觉得它是一门专业技能。而且我还觉得，它只是这种思维方式中的一个例子。你要能再说出一些比如机械方面的例子，我也很爱听。"现有一堆零部件，我怎么利用它们把水从这边运到那边，然后倒进杯子？"你操作的不是一行行代码，而是各种各样的零部件。你要把它们组合起来，再看看它们如何运转。

Seibel：还有，程序员应该了解计算机底层知识到何种程度？你在“用十年时间自学编程”中，曾讨论过如何获取执行一条指令以及读取磁盘数据等操作的时间。我们是否仍有必要学习汇编语言？

Norvig：我不知道。Knuth曾说过，要用汇编语言做任何事，只因C语言效率太低。我不同意这一观点。你可能想充分了解哪些指令的执行效率较低，但这已不仅仅是指令层的事。它不仅是“这不是二指令序列而是三指令序列”，而且还是“你是否遇到了‘换页错误’或者‘未命中缓存’？”我们需要熟悉的不是汇编语言，而是体系结构。你应该理解汇编语言到底是什么，还应该理解计算机具有分级存储体系，如果在某级未能命中，而去下一级存取，性能损失会非常大。不过我觉得它在抽象层更好理解。

Seibel：有哪些书是所有程序员都应该读的？

Norvig：有很多选择，路不止一条。首先，你肯定得找本算法书好好看看，不能只是东拼西凑地学些算法。你可以看Knuth的书，也可以看Cormen、Leiserson和Rivest的书。别的书也行，比如Sally Goldman，她的新书从更切合实际的角度看待算法，我觉得挺有趣。总之你得从这些书里挑一本看看。另外，你还必须看一些关于抽象思想的书。这方面我比较喜欢Abelson和Sussman的书，你也可以找其他人的书来看。

你要十分熟悉自己使用的编程语言。你要看看参考手册，再看看有关语言机制以及整个调试和测试流程的书，比如《代码大全》什么的。不过我知道，不同的路还有很多，我可不想给你们列个必读书单出来。

Seibel：虽然你现在的工作没必要编太多程序，但为了支持你网站上短文的观点，你还是编了不少。编这些小程序时，你是怎么着手准备的？

Norvig：我觉得最重要的，是把每一件事牢记于心。如果能做到这点，你成功的可能性就会非常大。它能让你比较轻松地完成小程序，但对于规模较大的程序来说，你还需要其他工具来帮助记忆。

了解你当前要解决的问题也很重要。我在编程求解“数独”问题时，有些博客评论说：“看看差距吧——这是Norvig编的程序，那是另一个家伙编的程序。”我不记得那人的名字，只知道他是个“测试驱动设计”专家。他开始编这程序的时候说：“嗯，我要解决数独问题，就必须有个数独类。我得先写出一系列针对这个类的测试。”然而，他一直在原地踏步。他发了5篇博文，每一篇的内容都比上一篇多那么一点点。他在博文里写了很多测试，但最后一点可执行程序都没做出来，因为他根本就不知道怎么解决这问题。

从人工智能的角度来看，我熟知这一问题。在人工智能里，有一项技术叫“约束传播”（constraint propagation），还有一项技术叫“递归搜索”，我很了解这两项技术的原理。而且我从一开始就知道，只要把这两项技术组合起来，数独问题就能解决。他不了解这些，所以即使他有一大堆测试用例，而且这些代码也都“运转正常”，他却仍然如同赶夜路一般，一点方向都摸不清。

这些博客后来还反复讨论这件事到底有什么意义。我觉得它没太大意义，测试驱动设计还是很棒的思想。如今我编程比过去多用了很多测试。不过，你虽然能写出任何你想要的测试，但你要是没搞清楚从什么地方开始解决问题，那你还是无法得到解决方案。

Seibel：这么说，关键在于他应该如何了解问题，对吗？他是不是应该去拿个博士学位，而且还是人工智能专业的博士？不过，你不可能对每一个算法都了如指掌。现在虽然有了Google，但搜索解决某个问题的正确方法还是比搜索某个Web框架困难得多。

Norvig：你是不是问如何知道自己不了解的是哪些东西？

Seibel：一点没错。

Norvig：我想它可能包括两部分。其一是了解是否可能存在现成的解决方法。在了解清楚这点之后，你也许会说：“好，没人知道怎么解决它，所以随便怎么研究都好。”这是一种可能。还有一种可能是：“嗯，很可能已经有人知道解决它的方法了，我只是不知道相关的资料有哪些，所以我要找出来。”从某种程度上，直觉会告诉你：“这问题似乎和人工智能某方面的知识相关。”然后，你得想清楚怎么找到这方面的知识。假如他这么做，那他很可能搜索数独问题，并且找到答案。但也许他觉得这样是在作弊。谁知道呢。

Seibel：假设这的确是在作弊，假设你第一个尝试并且最终解决了数独游戏，你的解决方法流传开来，等着别人来用。

Norvig：假设我想解决生物领域的某个问题。我不知道基因测序或类似工作的最佳算法是什么，但我有个好想法，那就是这种最佳算法已经存在。这样，我就可以着手寻找它。另一方面，有些知识相当基础，比如说，如果你不知道动态规划是什么，那你可就吃大亏了，它会在各种问题中一再出现。如果你不了解各种搜索思想，你可以权衡利弊选择一种，并在搜索到一定程度后回溯。这些思想都发端于60年代，发现它们可用于编程还是近几年的事。似

乎这类知识是每个人都应该了解的。而那些去年才刚刚发现的知识，并不是每个人都应该了解的。

Seibel：*那就是说，程序员应该回头读一下所有的老论文？*

Norvig：不，因为很多论文一开始就走错了。而且后来的合并也会很多，因为两个不同领域可能会独立发展出两套完全不同的应用技术和术语，最后却发现它们干的其实是同一件事。你最好获取一些现代的观点，而不是完全追随历史的脚步。但你手头也应该有一份旧材料。我不太了解这方面有哪些书比较好，我自己是零零散散积累起来的，很辛苦。

Seibel：*让我们回到软件设计的话题。当你编写大规模软件时，你无法记住所有代码组合在一起的形式，在这种情况下，你会怎么设计？*

Norvig：我觉得应该有各层次系统设计的完备文档。我们应该做什么，我们又是怎么做的？记录每一个方法往往过于繁琐。大多数情况下，文档只是简单重复你从函数名和参数名获得的信息。但是对于工作的整体设计，确实是应该首先写明白的。这种设计每个人都得能理解，还得是正确的选择。成功完成项目最重要的因素之一是员工具有丰富的经验，能构建恰如其分的软件。此外，如果你从来没构建过类似软件，不知道怎么去做，这时你能做到的次优选择就是把软件构架做得尽量灵活一些。这样一来，即使在软件的构建过程中出错，你还可以调整。

Seibel：*假如你之前没有构建过某个东西，只是坐在那里空想，那你能在多大程度上想出它的工作方式？为了真正理解问题的关键所在，你是否有必要着手编写代码？*

Norvig：思考它的方法之一是回退。你想最终得到一些好结果，对某些问题来说，大体上得到一种好结果。而有一些问题却有成千上万条路径可走，它们殊途同归。所以我认为，这取决于你所面对问题的类型。

你还想知道哪些选择比较困难，哪些比较简单。如果你选择了一个错误的体系结构——或者是超过系统限制，或者只是构建了错误的东西——那让你感到头疼的又会是什么呢？在Google，我遇到过各种类型的问题。可扩展性问题总是一再出现。如果你针对现在的情况，说我们构建一个处理能力10倍于现在的程序，那两年之后我们的需求就会超越这个程序的处理能力，这样一来就必须将其完全舍弃，一切从头做起。不过你希望至少对运行条件作出正确选择——你要应付10亿到100亿个页面。所以要考虑，如何把网站分

布式部署在多台机器？你的网站流量会在什么范围？你必须在这一层面给出令人信服的理由。你可以做些粗略计算，可以做些统计模拟，还必须对未来有所预见。

Seibel：*对这类问题来说，你似乎更可能通过粗略计算或者统计模拟，而不是编写代码，来得到更准确的答案。*

Norvig：是的，你说得没错。通过计算更容易解决那些问题。然后，可能会出现这样的问题：一些厂商声称他们明年会发布可处理10倍于现有流量的交换机。你会按照这一标准设计程序吗？你相信他们说的话吗？还是你仍然按照现有标准设计呢？这里有很多需要权衡的地方。

然后还有用户界面的问题。只有把界面完全构建好之后，你才知道它会出现哪些问题。你认为这种交互形式非常完美，不过当你把它展示在用户面前时，一半的用户都不明白它如何操作。在这种情况下，你不得不回头重新提出一个解决方案。

Seibel：*先不说用户界面的问题，什么时候应该做原型设计？还是只用想想程序是如何运行的？*

Norvig：我认为设想一个解决方案是有用的，感觉一下它是否可行。找找感觉是有用的。你需要一套工具，它能帮你构建当前必须构建的东西，也能随着时间推移帮你改善系统。如果你采用了原型设计，然后突然间却觉得笨拙不堪，那你可能是得到一组错误的原语。越早发现问题越好。

Seibel：*用测试来驱动设计的想法怎么样？*

Norvig：我更多地是从纠错而不是设计的角度来理解测试的。一个夸张的例子是先说："嗯，你先写个测试，它能表明我最终得到了一个正确答案。"然而，当你运行这个测试时，却发现它失败了，然后你茫然道："我接下来该怎么做？"在我看来，这不像是设计事物的正确途径。

似乎只有在解决方案一目了然的简单情况下，测试才有意义。你从最开始就要考虑这种情况。你不得不自问："这些部分是做什么的？在我理解它们到底做什么之前，怎么给它们编写测试？"然后，一旦你理解了它们，最好逐个给它们编写测试，完全理解它们如何与其他部分交互，边界情形如何等等。这些内容都应该测试，但我认为不能单凭"这测试失败了"一句话就能驱动整个设计。

我不喜欢的另一件事是，我们在Google遇到的很多问题都不符合测试的

简单布尔模型。你看看这些测试套件，它们包括assertEqual、assertNotEqual、assertTrue等。这很有用，但我们还需要assertAsFastAsPossible，来验证一个大数据库里所有可能出现的查询，我们从数据库得到了某个准确率和某个召回率，想对它们进行优化。它们没有你能优化的那种统计值和连续值，只有一个布尔值："它是对的还是错的？"

Seibel：但最终，所有这些都能转化为布尔值，你可以运行一堆查询，获取所有的查询值，并且看看它们是否处于你可接受的范围内。

Norvig：是可以那么做。但从那些测试套件所提供的方法中，你能看出它们本来不是那么用的，他们没考虑那样做的可能性。我很惊讶Google已深入接受了这种方式。我以前在Junglee的时候，可是必须要教会QA人员这些的。那时在做一个购物搜索的时候，我们要求："我们想要一个针对这个查询的测试，看看是否有八成以上结果是正确的。"然后他们说："没问题！一个错误的查询结果就相当于一个bug，对不对？"我们回答："不，一个错误结果不碍事，只要别太多就行。"接下来，他们又问："也就是说，错误结果并不是bug了？"好像这里只存在两种选择一样，非此即彼，没有任何中间路线可走。

Seibel：不过即便如此，你仍然主张单元测试。你觉得程序员应该如何看待测试呢？

Norvig：他们应该编写大量测试，还应该考虑各种不同的条件。对他们来说，复杂的回归测试和单元测试是多多益善。此外，失效模式也是应该考虑到的。我记得有一回在希思罗机场，突然停电，所有计算机都无法工作，但我的飞机却仍然没有晚点。这件事的确深深影响了我。

也许他们事先已经把所有航班都打印了出来。我不知道他们在哪儿打的，不过这计算机肯定存在。我也不知道他们是当天早上现打的，还是头天晚上就打好了的。他们把这些打印好的航班信息运送到机场，如果没停电，这些纸就会直接扔进废纸堆。但不管怎样，他们有这些信息。登机口的工作人员可以看看这纸，而不一定操作什么计算机系统。

我想，这是可供软件设计借鉴的典范。我觉得大多数程序员都不会考虑："停电的时候，我的程序是否还能正常工作呢？"

Seibel：停电时Google是如何工作的呢？

Norvig：Google在停电时做得可不怎么样。不过我们有后备电源，还有多个

数据中心。我们是这么考虑的："当我们连接的服务器崩溃，或是发生其他失效状况，那我做的这部分工作还能正常运行吗？"或者："我写的程序运行在上千台计算机上，如果其中一台崩溃，又会发生些什么呢？那台计算机上的工作能顺利交接给其他计算机吗？"

Seibel：*Knuth写过一篇关于TeX开发的短文，里面谈到他从排斥QA，转变成一个非常重视QA以致具有破坏力的程序员。他费尽心思去破坏自己写的程序。你觉得大多数程序员都擅长这个吗？*

Norvig：不。有这么个例子，是在我写拼写检查器时发生的。我给这检查器写了个评测代码质量的工具，但里面却有个bug。我对检查器做了些小改动，然后对它运行评测工具，结果得到了比改动前高得多的分数。我相信这是真的！但如果得分比以前低得多，我却不会说："啊，改动这函数让它的性能更差了。"我情愿相信我做的改动能让分数变得更高，而不会去质疑说："不对，这改动不可能让性能提高这么多，一定有什么地方出错了。"

Seibel：*你是如何避免过度泛化（over-generalization）和构建多余的工具，从而浪费过多资源的呢？*

Norvig：这就像一场战斗，我们要在各方面反复斗争。不过我一向偏爱一流的而非实用的解决方案，所以我大概不是回答这问题的合适人选。面对这问题时，我必须和自己的内心作斗争，告诫自己："我是在为公司工作，要对公司负责，不要浪费时间和精力去考虑那些不实用的东西。"因此，我只能这样考虑："我们现在只要给出一个可行方案就好。如果再多考虑考虑，也许我们能够找到一个完美的解决方案，但那样的话，也许我们就无法如期交工了。"于是我们不得不放弃追求完美的想法："看来，我们要把当前问题中最重要的那个解决掉。"我必须不断向自己以及同事们灌输这一理念。德国人有这样一句谚语：完美是美好之敌。我忘记这句谚语到底出自何处了，不过，每一位务实的工程师都应该牢记它的含义。

Seibel：*我们为什么都想去解决那些没有必要彻底解决的问题呢？*

Norvig：人们都希望自己足够聪明，能彻底解决这些问题；人们也想在了结它们之后，继续研究其他问题。在我看来，人类只能集中处理一定数量的事情，所以人们会这样说："我已经完全解决了这个问题，现在我可以把它以及所有与之相关的材料统统抛到脑后了。这样一来，我就有精力研究其他问题了。"不过在解决问题前，你最好计算一下完全解决它能给你带来怎样的

回报。这回报率总是呈S型曲线。所以当你解决一个问题到差不多八、九成的时候，再往后得到的回报会越来越少，而你本可以用那些时间去解决另外100个问题，那些问题尚处于曲线底部，你能从中得到可观的回报。也就是说，你在解决问题到某一程度之后会说："好了，就这样吧。让我们停下来，做些别的有更多回报的事情。"

Seibel：*那程序员怎么才能更清楚地认识到自己正处于曲线的什么位置呢?*

Norvig：我认为首先要有一个合适的环境，以结果为导向的。我还认为，人们要有意识地培养自己这方面的能力。你想着在某些方面有所改进，但你改进的只是自己是否感觉舒适，那并不是你真正应该改进的。真正应该改进的应该是公司的投入产出比，还有客户的满意程度。你必须考虑到，如果你把某个特征从95%完善到100%，客户会从中获得多少利益，如果你从零开始构建另外10个特征，客户又会如何受益。

在Google公司，我觉得这是件很简单的事情，因为我们的哲学理念是"尽早发布，不断更新"。公司这样做是出于两个原因：其一，我们的大多数产品都不需要用户花钱，所以很容易下决定："好，把它发布出去吧，看看他们会如何抱怨。"其二，我们不用给CD打上标记，再把它们放进包装盒，所以就算今天有什么没彻底完成，甚至存在一个bug，那也不是什么灾难。我们的大多数软件都放在服务器上，所以只要明天一修复好这bug，所有用户都能立即得到更新。这样，我们就摆脱了只有通过安装才能更新的梦魇。因此对我们来说，这样想会更轻松自在："我们就是要先发布些什么，再从用户那里得到些反馈，然后根据反馈，有针对性地修复那些必须修复的地方，而不是盲目地修复那些暂时没必要修复的地方。"

Seibel：*如果你正设计一个大型系统，你会使用哪些工具？你是拿着一沓坐标纸坐下来写写画画呢，还是使用某种UML绘图工具呢?*

Norvig：不论什么样的UML工具我都不会喜欢。我一直认为："如果不能用语言本身的特征来解决某个问题，那就说明这语言存在某些不足。"我认为人们很多时候都是在较高的层次上考虑如何解决问题，而在Google，我们更需要考虑如何把问题分解，以及如何并行化。我们有必要把程序运行在多台服务器上，因为我们拥有太多用户，而且很多应用都要处理海量数据。我们要从这些角度考虑。所以说，我们较多地从机器层次以及多个服务器角度考虑问题，较少从函数以及函数间的交互考虑。只有当你完全弄清楚这些，你才能进一步思考独立的函数和方法。

Seibel：在那种层次描述问题的话，仅用文字就够了吗？

Norvig：是的，差不多够了。偶尔他们也会画些图。他们会说："我们部署在这里的一台服务器要处理这几类请求，然后它会连接到另一台服务器上，我们会使用各种各样的工具，用于存储、大型分布式散列表以及其他用途。我们会选择这3种现成的工具，然后我们还会讨论一下，是否有必要构建新工具。因为我们还不知道，是现有工具中的某一个就能搞定它们呢，还是需要一些新的工具。"

Seibel：那你们如何评价这类设计是否合理呢？

Norvig：我们会把设计给那些解决过相似问题的人看，他们会给你一些建议："啊，看起来这里需要一些缓存。因为这里可能会产生很多次的重复请求，所以速度会变得非常慢。如果在这里设置这么大的缓存，应该会对提高速度有很大帮助。"会有人审查你的设计，他们会告诉你这样设计是否合理，然后你才能开始后面的构建和测试工作。

Seibel：你们有正规的设计审查制度吗？你在NASA工作过，那里设计审查可是非常正规的。

Norvig：没有比NASA还正规的地方了。我们所冒的风险较低，因为我们出现故障和修复错误的代价都小得多。而在NASA，一次故障就足以致命，所以他们多小心都不为过。至于我们，实在不必太担心这些。在我看来，我们更像是咨询而不是审查。

在我们这儿，会有些人很正式地阅读设计文档，并且给出他们的意见。你的设计必须在经过这一步之后，才能获得大家的一致认可。但NASA所做的还是远远比这些要正规得多。他们在项目的启动阶段就开始审查。在项目的整个流程之中，他们会定期审查，但他们不是真正查看代码。他们更多情况下会问你："你是怎么考虑这个问题的？你会在预定日期之前还是之后完成？你会遇到什么大麻烦吗？"诸如此类。

他们的发布阶段是所有阶段中最正规的。发布前他们会列个清单，提出很多逻辑严密的安全性问题，比如：如果我们现在发布它，会有谁能侵入它吗？会有人通过跨站脚本控制它吗？这个清单上列出的问题都是十分严格的。

Seibel：你曾经对我说过，Guido van Rossum和Ken Thompson来你们公司，也一样要检查他们写的Python和C代码，以确保这些代码都能满足明确而详尽的编码规范。你们有没有和编码规范同样明确详尽的设计规范？

Norvig：没有。有些编码规范可能会导致设计上的问题，但你也会因此得到很多自由设计的空间。然而基本原则肯定还是有的，你在贡献代码之前，必须先证明自己胜任。你提交的每一段代码都要由别人审查，并确认里面没有错误。

Seibel：这么说，每一段提交给Perforce版本库的代码，入库前都要接受审查？

Norvig：你可以自己鼓捣一些试验性代码，而且也存在一个例外机制，在此机制下，它们可以进入版本库，而且以后才会审查。但你应该尽量降低发生这种情况的可能。

Seibel：也就是说，这基本上和传统的人工检查没什么区别："这是我写的代码，别人会先阅读一遍，然后说道：'嗯，这代码没什么问题。'"

Norvig：对。实际上，Guido的第一个项目就是关于这个的。我们使用标准的diff工具检查代码，但它做这项工作有点困难。于是Guido写了个分布式系统，具有漂亮的界面和颜色标记功能。使用这系统，我们能更方便地审查别人提交的代码。

Seibel：很多公司都说自己应该有审查制度，但极少有公司贯彻始终。从某种程度上说，你必须手把手地教会他们如何审查。

Norvig：我觉得人们一直在实践它，它也由此而被广泛接受。嗯，也许我不该那么肯定。有些人只需要较短的时间就能习惯它。不过有个典型的失败案例，是在那些新员工身上发生的。他们刚刚进入公司，还没养成这些习惯，所以他们会建立一个试验性分支，把所有代码都提交进去。你要是不断地提醒他们："嘿，你不能这么提交。"他们会回答你："好的，好的，好的，我把它们清理掉，明天我再提交。"一周又一周过去了，他们终于积攒了一大堆提交上来的代码，然后会出现这样的问题：这些代码经历的时间实在是太长了，不仅在短时间内很难审查完毕，而且有些用来对比的代码也和原先完全不同了。他们会发现这样做后患无穷，以后就会尽量避免这种情况。

Seibel：这的确是程序员应该注意的。那么审查者应该培养哪些技能呢？

Norvig：确实有这样一些人，大家都公认他们是出类拔萃的审查者。在你提交审查报告时，需要认真考虑一个问题：你是想让别人给你反馈许多有益意见，还是只要他简单地说上一句"好的"就行？

Seibel：是什么让那些审查者比别人更出色？

Norvig：嗯，因为他们比别人掌握了更多的技能。其中有些技能可能是雕虫小技，比如改正缩进空格数的错误之类的，但也有些是“如果你把这部分挪到那里，这设计会更简洁”这样的经验之谈。所以有些人喜欢审查工作，有些人却不胜其烦。

Seibel：还有一个与此相关的问题：是不是每个优秀程序员慢慢成长之后，都能成为优秀架构师？还是说，可能有这样一些程序员，他们编写代码时才思敏捷、技艺精湛，但水平却总在原地踏步，因此永远设计不了更大规模的程序？

Norvig：我认为不同的人拥有不同的技能。我们公司有个最优秀的搜索技术人才，如果从编码风格来看，他和最优秀的程序员相去千里。但如果你问他：“我们在这里发现了一个新因子，你看，人们在完成某行为后，会点击n次这个页面，我们怎么利用它来改进搜索结果页面呢？”他会告诉你：“看，代码的第427行有个变量α，你应该用那个新因子的平方乘以1.5，再把结果加到变量α上。”在他说了这番话之后，你花了两个月时间，反复验证那个新因子的各种值，最后发现他说得几乎完全正确，除了应该乘以1.3而不是1.5之外。

Seibel：这个例子告诉我们，他构造了一个关于该软件运行方式的行之有效的心智模型。

Norvig：他完全理解了软件代码。其他人的代码可以比他写得更好，但只有他明白代码字里行间的意义，比如软件的某个功能在代码的什么地方实现。

Seibel：你认为写代码和理解代码之间有联系吗？似乎总有这种事：最差劲的面条式代码往往出自那些对代码考虑最周全的人。不过也正因为他们理解得太多，所以代码写出来只能是那种形式。

Norvig：是的，我觉得它们也许有联系。

Seibel：这样看来，这审查的确远不如NASA正规。如果从“软件工程”和“黑客精神”这两个词的内在含义来看，这两种信仰还存在哪些不同之处？

Norvig：组织结构和对软件的理解方式都有很大不同。Google是一家软件公司，他们外聘了一位毕业于加州大学伯克利分校、拥有计算机科学博士学位的CEO，还聘用了一位具有计算机工程背景的销售副总裁。在Google，这种情况比比皆是。而NASA全是科学界精英，没有专业的编程人员。他们念叨着：“编程的确很烦人，但谁也逃避不了。如果代码是线性执行的，那我还

能明白一点；如果代码里出现循环，我就开始犯晕了；如果循环里还有分支，啊啊啊啊啊，这大大超出了我的能力范围，我擅长的领域可是控制论里的微分方程求解。”所以说，他们不太信任软件。

Seibel：*可他们就应该那样！*

Norvig：是的，他们就应该那样。他们不信任一切新事物。所以他们会先说："看看我手头的这个绝妙的新原型吧。"然后说："它简直太棒了。不过，虽然我很想在自己的下一个任务中使用它，但我必须先在另外两个任务中试验一下，证明它完全没问题才行。"在NASA，每个人都会对你说类似这样的话。

在Don Goldin作为负责人来到NASA之后，他提出："我们必须做得更好、更快，花更低的成本。太空任务开销巨大，我们最好同时进行更多的任务。这样的话，虽然其中有些任务会失败，但总的来看，我们花费差不多的钱，却完成了更多的任务。"毫无疑问，这观点十分正确。然而，它在道理上是不正确的。航天飞行器一旦出了事故就很难应付。老百姓其实并不知道价值1亿美元和10亿美元的航天器有什么区别，他们只知道NASA损失了一个航天器。这跟10个1亿美元等于1个10亿美元完全是两码事。所以说，他说的那话肯定是不对的。

Seibel：*在你不得不去彻底追查的那些bug中，情况最糟的是哪一个？*

Norvig：我想，在我参与调查过的那些造成严重后果的bug中，绝大多数都不是由我全权负责的。不过，发生在98年的火星探测计划中的一系列事故，却让我在事后不得不为其收拾残局。其中有一个bug是英尺磅和牛顿之间的单位换算错误。另一个bug，我们认为，但无法百分之百肯定，是由于软件问题而导致发动机提前关闭。

Seibel：*我看过一份火星气候探测者号的事故调查报告，就是刚才你提到的那个存在英尺磅和牛顿的单位错误的火星探测卫星，你是那个调查小组里唯一的计算机科学家。你在调查过程中和那些软件开发人员交流过，有没有帮助他们发现问题所在？*

Norvig：这非常简单。从事故发生的结果来看，只能推断出唯一一种原因，因为他们了解那种故障模式。知道原因之后，他们本可以取消任务，这花不了太长时间。是不是有点事后诸葛亮？我认为那次事故是由很多原因综合导致的，外包是其中一个原因。那次任务由帕萨迪那的喷气推进实验室（JPL）和科罗拉多的洛克希德·马丁公司合作完成，结果两个团队的成员甚至都

不肯坐到一起吃顿午饭。我相信，如果他们能在一起一边吃饭，一边好好沟通一下，这问题是可以解决的。但他们没这么做，而是由一位成员发了封电子邮件，邮件里说："这些度量单位不太对劲，我们似乎有些偏差。偏差不算很大，情况可能还好，但是——"

Seibel：*这就是那次任务的事故原因真相吗？*

Norvig：是的。在那次任务中，他们有好多次机会可以把错误揪出来。他们知道有什么东西不对，还发了电子邮件，但却没把这错误放进bug跟踪系统。假如他们这样做的话，NASA对于bug跟踪有着非常出色的控制，肯定会有人在其后的某个时间修复这个bug。但他们只是发了一封非正式的、不会有人理会的电子邮件，JPL的人看见了说："哈，我想洛克希德·马丁公司肯定已经解决了这问题。"洛克希德的人也说："哈，JPL没继续追问，他们一定觉得这不算什么问题。"

所以说，这首先是一个沟通问题。同时，它也是一个软件复用问题。他们对任务中的关键部分进行了极其细致的检查，可在以前的任务中，英尺磅并不是关键部分，它只用在了日志记录中，而日志记录没有用于导航，因此它给分到了非关键部分那类。在那次任务中，他们复用了以前任务的大部分内容，但修改了导航。原先仅仅作为日志记录文件的那部分，现在变成了导航的输入。

Seibel：*也就是说，问题实际上是这样的：你们一边生成以英尺磅为单位的数据文件，一边用这文件来计算以牛顿为单位的导航输入。*

Norvig：没错。所以从本质上说，还存在另一个根本原因：太空中的太阳粒子实在是太多了。这颗探测卫星上面有几块太阳能电池板，因此并不对称。这样一来，太阳粒子会让卫星稍微有些旋转，你得通过发射火箭来恢复卫星的角度。在洛克希德公司的新雇员和火箭制造商打交道时，所有规格说明都是以英尺磅为单位的，所以他就说："我需要这种单位。"制造商记录下这要求，却不知道NASA想要的是公制单位。

Seibel：*在看过那份报告后，NASA的这种态度深深打动了我："嗯，虽然这问题的原因在于软件bug，但我们有大把机会注意到这卫星没在预定位置上出现。我们本该注意到这点。由于软件中存在的一些愚蠢的小错误，导致我们处理的数据是完全虚假的。不过，尽管现在修复错误已于事无补，我们还是应该认真修复它们。"我觉得这是一种值得钦佩的态度。*

Norvig：是的，他们检查了整个流程。

Seibel：软件里的数值误差bug其实很普遍，对吗？因为现在我们几乎所有事务都是在线处理的，所以才没怎么听说过这种bug。

Norvig：是的，的确如此。检查一下你计算机的所有软件bug吧，这种bug有数以百万计之多。但在大多数情况下，它们都不会导致系统崩溃。

Seibel：不过，你听没听说他们在给航天飞机写软件时，每行代码都要花费1500美元，因为他们必须加倍地小心谨慎，而且据说要保证一个bug都没有。这是真的吗？

Norvig：这很可能是真的。但我不清楚这样做是不是最优。在我看来，如果使用带有一些bug的软件，他们也许会更好过些。

Seibel：使用更便宜的软件，但采用更好的运营管理方式？

Norvig：是的，因为宇航员必须得到充足训练，以保证有能力处理软件所无法处理的事情。他们要让宇航员进入一个模拟装置，该装置可模拟所有可能发生的情况，故障出现时，你会看到屏幕上出现很多滚动信息，你无法让信息停止滚动，无法退回到原先的无故障状态下，也无法让系统告诉自己下一步该做些什么。宇航员不得不通过训练逐渐懂得："当我看到这种情况出现时，背后的真正原因是什么什么。"上百条不断显示的信息告诉他们："这台带电的大家伙出错了。"他们在训练之后会知道："好，这种情况一定是因为最开始只有一个程序出错，然后这错误迅速波及到其他程序，导致所有程序的错误报告都显示到屏幕上。"那么，为什么他们宁愿用软件完成这些，而不愿去训练宇航员呢？因为他们不想让过多的训练打乱原先的管理流程，所以压根就不去尝试这种方式。

Seibel：让我们换个话题。你喜欢哪些调试技术和工具？是打印语句，形式证明，还是符号调试器？

Norvig：我觉得是多种方法的组合，而且取决于所处环境。有时我用IDE，它具有不错的跟踪功能；有时我就用Emacs，它没有这方面的任何功能，当然，还要用跟踪和打印的功能。还有就是认真思考。先写一些较小的测试用例，然后观察它们如何运行，再通过功能分解找到测试用例在哪儿出错。不得不承认，我往往会以重写代码收场。有时，我用这些方法找不到bug。我到达程序中的某处，感觉bug离我已经很近了，因为这部分代码让我看着感觉心里十分不爽。它简直就是一团糟。它真不该以这种方式写出来。这种情

况下，我不乐意一点点改它，而是干脆把好几百行代码直接删掉，再从头开始重写，这样一来，bug往往就会烟消云散。

有时我不免对这种方式抱有几分愧疚。那部分代码真出错了吗？我既没弄明白那bug到底是什么，也没找到那bug到底在哪儿。我只不过往旧房子里扔了颗炸弹，把所有bug统统炸光，然后盖一所新房子。某种意义上说，bug从我手中溜走了。但如果能形成可行的解决方案，那你这样做也无可厚非，毕竟和定位bug相比，你重写一遍代码会更快。

Seibel：*你如何看待断言和不变式这类技术？你在编程时会形式化地考虑使用这类技术吗？*

Norvig：我想我更喜欢非形式化。我只用类型声明语言，从没用过有大量形式化机制的语言。我总觉得像循环不变式这样的技术，它们带来的麻烦比好处更多。我偶尔会碰到死循环，但大多数情况下不会有这种问题，而且我确实感觉到形式化会让自己变得更慢。如果你的程序里真有死循环，那么调试器会告诉你陷入了哪个循环中。我想，如果你当前正在写的软件需要高可靠性，将构成一个非常重要且不允许故障的大型系统，那你的确应该严格证明软件的每一部分。但如果仅仅想把程序的第一版做出来，先看一下运行和调试的效果，那我宁愿尽快实现程序，而不去考虑之后可能需要多完善的形式化文档。

Seibel：*你们有没有透明、公开地做过什么尝试，为了能从已发生的bug中吸取经验教训？*

Norvig：是的，我们做过。我觉得很有趣，希望以后能多多尝试。实际上，目前我正和别人探讨由我在公司范围内进行一项实验的可能性，而且这项实验也许还能逐渐扩展到世界范围。通过这项实验，我可以更深入地理解一些与bug有关的问题：你怎么对bug分类的？其中哪几类可作为影响生产率的因素？你通过什么了解到这些的？每个人都能归为某种类型吗？对于某一类人，又有哪些可提高他们生产率的因素？我更感兴趣的是那些能让人在工作时表现得更加出色的可控因素。如果给员工提供更大的显示器能够提高多少个百分点的生产率，那你也许真应该给他们一台。

Seibel：*不过如果你发现其实微型显示器能让他们的生产率更高，他们就该记恨你了。*

Norvig：倒也是。而且，如果保持环境安静是关键因素，那你应该尽量少说

话，但另一方面，如果团队成员之间的沟通也是关键因素，那你又应该和别人多沟通，你要如何权衡它们呢？

我最近一直在考虑如何正确实施这项实验。我们要如何安排它？如何跟踪实验结果？如果把某一类问卷添加到实验中，我们能否得到有效数据？我们非进行这项实验不可吗？

Seibel：*常有人宣称，不同程序员之间的生产率差距可能达到几个数量级之多。然而，我也在什么地方看到过一个对这种言论的批评意见，说有多项研究表明这种差距早已时过境迁，随着时代的发展，很多证明这差距的编程知识和技术都已经发生了变化。比如某项研究中，有些程序员还在使用批处理技术，有些程序员都已经用上分时编程环境了。*

Norvig：我觉得这还不是全部，因为同样的工具在同一组织中使用时，情况还略有不同。我还记得对发现统计相关性的某些研究结论的批评，就是因为他们不清楚在一对相关因素中，哪个是起因，哪个是后果。如果你发现在豪华办公室里工作的程序员效率都很高，那是因为优秀程序员都奖得了豪华办公室呢，还是豪华办公室真能激励程序员努力工作呢？你很难得出一个正确结论。

Seibel：*你还是像刚接触编程时一样沉浸于编程的乐趣之中吗？*

Norvig：是的，但如果写着写着卡壳了，我就会感觉心烦意乱。和以前相比，我现在编程没那么多了，所以有些东西也忘记了，加上各种新鲜的编程技术也在不断涌现。看看我的个人网站，我早就该把它重新设计一下了，应该在客户端用到一些JavaScript，还应该用些PHP什么的，可我就是提不起精神去学习和使用它们。

Seibel：*你觉得编程这行业更适合年轻人吗？*

Norvig：我觉得年轻是一种优势，不过的确各级别、各年龄层都有顶尖程序员。在我看来，年轻人的优势是能够全面地把握程序和问题，能够集中精力去思考。年轻人的大脑比较善于处理这类事情。当然，也可能年轻时没什么分散精力的事情，等到成了家，有了小孩什么的，就很难像以前那样连续好几个小时全身心投入到编程上了。所以这是年轻人有优势的一面。然而从另一方面来看，年龄的增长还带来了丰富的经验，让你在面对问题时明白该如何下手，因而比年轻人解决问题的速度更快，这也在一定程度上弥补了我刚才提到的差距。

Seibel：现代编程风格的特点之一，正如你所说，是程序员必须迅速消化和吸收自己面前的东西。如果你面前有一大堆以前从未见过的代码，那你要如何去读懂它们呢？

Norvig：我想应该从静态和动态两方面去做。刚开始读代码时，你要先弄清楚它是干什么的；然后，你对在什么地方调用什么东西进行跟踪，这是最花工夫的一步；最后，你总结出全部代码的工作流程。这些都做完之后，你就可以去尝试解决更难的问题，比如说："我要稍微改改这代码。"或者到问题库（issues database）里看看然后说："我要解决掉这个问题。"你必须深入理解某个问题，然后再去解决它，但你仅仅需要理解一小部分问题。你可以先解决掉这一小部分，再去理解后面的其他部分。

Seibel：你像Knuth那样用过文学编程方法吗？

Norvig：我没直接使用过他的那些工具。我的确写过宏什么的。而且我也用过Java文档以及类似的工具。从许多方面看，用Lisp语言编程其实是帮助你逐步构造一套你自己特有体系的过程，所以最终它也会以一种文学的形式展现出来。你在给自己特有的应用编程时，会得到一组特有的宏，其中一部分是文档，一部分是数据。从这方面来说，我肯定是用过文学编程方法的。还有最近，我用任何一种语言编程——不管是Java、Python，还是别的什么语言，都会特别小心地编写测试用例，以及与之相关的文档。

你可以看看Knuth写的《文学编程》一书，他一直试图解决这样一个问题："我写一本书时，什么样的顺序最合适？"假如有人通读全书的话，肯定希望那书具有某种逻辑顺序。可这样做的人越来越少。人们不想去读一本书，而是想要一份那书的索引，然后说："在这本书里，我必须了解的基本内容有多少页？我只想读一下需要我去了解的那3段内容。读完它们，我就可以继续前进。"我觉得这是一次本质上的转变。

Seibel：我想知道到底有没有用现代风格进行文学编程的方法。使用Knuth的那些工具确实能产生索引，而且也能产生漂亮的交叉引用文档。不过我想知道的是，如果采用现代的文学编程方法，他那本书的组织方式是不是也会完全不同，因为你既可以从宏观上从头到尾地理解整个程序，也可以从微观上理解程序的各个组成部分。

Norvig：我不知道。我觉得他当时想解决的问题，现在都不算什么问题了。我这么说，一部分原因在于他想用线性顺序组织程序，而不考虑网状顺序等

便于搜索的顺序；另一部分原因，我想应该是编程语言的局限性。我知道他当初用的是Pascal语言，这语言相当严格，它规定变量必须先声明再使用，而你又不一定想用这样的顺序编程。现代语言在顺序上更加自由，所以我觉得，如今这几乎已不是什么问题了。

Seibel：前面你提到，在看《科学美国人》时，你读过Strachey写的国际跳棋程序的代码。而且在“用十年时间自学编程”一文中，你也谈到了阅读代码的重要性。你还读过哪些代码？

Norvig：我读过许多Symbolics公司的代码，因为我在Berkeley的时候比较容易得到它们。

Seibel：仅仅是因为比较容易得到你又感兴趣？还是你阅读它们的时候，还试着去理解一些程序中反复观察到的行为？

Norvig：二者都有。有时我就想琢磨一下它的运行方式，有时我还想利用它解决其他问题。

Seibel：也就是说，有时候你阅读代码只是为了陶冶情操。你是怎么到达这种境界的？

Norbig：我想这大概是兴趣所致。“嘿，这文件系统使用和本地计算机相同的协议来读取网络中的文件，我想知道它到底怎么做到这点的。”你说，“也许秘密就在这个open函数里。”你看看那函数，然后发现：“啊，它调用了另一个函数。”你又去看它调用的那个函数，终于发现：“哈，答案就在这里。”

Seibel：你在看Knuth的书时，有没有读过他写的文学程序？

Norvig：我肯定大致浏览过他那些书。可以这么说，我看过那些文学程序，但我没深入研究过它们。

Seibel：你觉得《计算机程序设计艺术》这套书怎么样？我在访谈中问过一些人，他们中有些真的从头读到了尾，有些把它放在书架上，在需要时作为参考书查阅，还有些就只是摆在书架上，连碰都不去碰它。

Norvig：有段时间我拿它当我的显示器底座，因为它是我最大部头的成套书之一，而且高度恰好合适。我感觉这样很舒服，因为它总在那儿陪着我，而且因为它就在我面前，所以我找参考书的时候就更容易去顺手翻翻它。

Seibel：但你每次想查阅它，还得先把显示器抬起来吗？

Norvig：不用，我那套书是盒装的，你只要使劲抽书就行，也可以只抽其中一本。不过我现在不怎么查阅参考书了，一般都是搜索。

Seibel：*因为搜索更方便？*

Norvig：的确方便。也因为我可能比较以目标为导向。如果你说"我想知道关于这个题材的一切内容"，那Knuth的书确实不错。不过我一般会说"我想知道A是不是比B更好"或者"我想知道它的渐进复杂度，我一旦知道了这点，就不必再了解其他所有相关的细节了"。

Seibel：*作为一名程序员，你觉得自己更像是科学家、工程师、艺术家，还是手工艺者呢？*

Norvig：哎，在比较时下的各种书名的时候，我始终认为"手工艺"才是正确答案。我觉得用"艺术"形容编程有那么点做作，因为艺术的目的是产生美，或者说，要有情感上的交流与冲击，我没觉得自己编出来的程序是这样的。当然，我也希望程序具有一定的美感，有时我会感觉自己好像在美化程序上浪费了太多时间。我现在所处的职位给了我不小的空间，因此我可以说："快，我还有时间，我可以再稍微美化一下这程序。"发表文章也一样，如果你真想提高自己的专业技能，那就得在写文章上花费更多心思。

但我没把它看成艺术。我觉得对编程来说，"手工艺"是最贴切的词。就好比你会造椅子，你造的椅子看上去很美观，但更重要的是坐上去不会散架，说到底，它是把椅子。

Seibel：*你们怎么判断一个人是不是优秀程序员，特别是在招聘程序员的时候？你们招聘过很多程序员，而且显而易见的是，你们想招那些真正优秀的编程高手。怎么做到这一点？*

Norvig：我们现在还不太清楚怎么能做到这一点。

Seibel：*Google在问面试智力题方面也算小有名气。你认为那是好方法吗？*

Norvig：我认为能不能解答出智力题是无关紧要的。我反感那些暗藏机锋、花样百出的智力题。在我看来，面试的关键是为面试者设置一个技术场景，考察一下他怎么处理这种局面，而不是和他瞎扯一通之后，看看他是否给你留下了一个不错的印象。虽然你和招聘到的人能否和谐相处也很重要，但你真正要了解的，是他们的技术能力是否和他们声称的相一致。你可以通过很多方法看出这一点。很多情况下，你可以通过他们的简历判断出来。如果他以前曾和我们公司的某位职员共事过，而且那职员对他的评价不错，我觉得

这也很能说明问题。尽管如此，我还是希望在面试过程中逐渐了解他。你想去体会他思考问题的方法，还有他和别人一起合作的方式，也就是说，他是否懂得那些从事技术工作的基本概念。他们也许会说："哦，要解决这问题，我必须知道A、B和C。"于是他们着手把A、B和C拼凑在一起。我想，你在解答不出智力题的时候，也可以这样展示你的思维过程。你可以说："嗯，这是解答这道智力题的关键所在。我先认真思考了这点，然后又思考这个、那个，可是老天啊，我就是不太理解这一部分内容。"有些人能突然明白那一小部分内容，有些人不能。如果不能的话也没关系，只要你在思维方面展现出一定的能力，而且比较有条理，那也能顺利过关。如果你招聘的是编程人员，那你一定要让他们在白板上写些代码，因为他们要是忘掉或者不熟悉哪一方面，你很快就能看得出来。

Seibel：这仅仅是一个负面信号吗？如果他们写出来的代码不合理，那确实是个不好的信号，但如果他们没被难倒，你也很难断定他们能在更大规模的环境下写出令人满意的代码。

Norvig：没错。你只能在某一水平上判断这点，但到了另一个水平就又判断不出来了。我们非常认真地研究过这问题，因为我们在这方面的实际应用很多。我们会在两个水平上考察他们：首先，我们从手头的个人简历中，能不能找到一组合适的面试者？其次，我们从已经完成的面试中，能不能找到一组合适的录用者？

Seibel：那你们怎么评分呢？你们不了解那些没交谈过的人，也不了解那些还没录用的人。

Norvig：是的，所以说这问题很难。在这两个水平上，你都只能用到一半样本，所以这是个有偏问题。但我想当务之急还是要了解："在我们面试时表现出色的那些人，他们的简历是什么样的？"我们还尝试了解更多这类问题。有多年工作经验是否重要？为开源项目贡献过代码是否重要？是这些更重要，还是在编程比赛上获过奖更重要？

Seibel：你们真这么做了？把它们都塞进数据库？

Norvig：是的，我们真这么做了。我们在招聘时，能在短时间内得到分数："简历预测器告诉我们的分数是多少多少，面试预测器告诉我们的分数又是多少多少。"我们没把这些分数当作绝对的衡量标准，而只是把它们看成与其他反馈信息并存的输入。

Seibel：主持面试的人会事先知道这些分数吗？

Norvig：不会，我们在收集反馈信息的时候，他们还在招聘委员会里，只有我们能看到所有的反馈信息。在尝试预测员工受聘一到两年后会有怎样的工作表现时，我们发现了一件有趣的事：在多次面试中，你得到的最差分数是个很好的预测信号。我们把面试者按1到4评分，如果你在某次面试中得到1分，那说明你不太可能成功。

Seibel：为了得到录用一定得有些过人之处吧？

Norvig：没错，就是这样。在某次面试中得到1分的人，百分之九十九都没录用。剩下的人要想被录用，公司里必须有人站出来，一边情绪激动地敲着桌子一边对我们说："我必须录用他，他身上有些特点十分突出，说他不行的人看走眼了。我一定全力支持他，我愿用我的名誉担保。"

Seibel：看来Google是人才济济，顶尖程序员数不胜数。当今社会，计算机和软件无处不在。你觉得，为了跟上世界的发展脚步以及更好地认识人类居住的这个世界，普通人有没有必要学习一点编程知识？

Norvig：你可能想让受过良好教育的人去理解软件是如何写出来的，就好像理解小轿车是如何生产制造的一样。还有一件有趣的事情是，有多少有学问的人不得不去做一名程序员。当然，如今普通人也能做些文字处理工作，其中很多人还会用电子表格软件。这样一来，如果你以前用过电子表格软件，那你就可以成为一名程序员了。

在终端用户编程和从编程方面进行的多种尝试都不算太成功。我不知道它的难度有多大，也不知道是否存在这样的思维方式：所有容易教的人都已经教过了，剩下的都是难教的；或者我们只是错过了某种简单的模型，如果我们创造出这种模型，它就能取代很多程序员。

Seibel：我写这本书的过程中访谈过许多人，他们之所以从事计算机行业，一方面是因为他们发自内心地热爱计算机，另一方面是因为他们觉得计算机能改变世界。其中有些人做过不少研究，但最近一段时间却意志消沉，因为他们觉得世界并没有因为计算机而改变多少。你有这种感觉吗？

Norvig：没，我找准了自己的位置。我们有数以亿计的用户，我们会让他们感觉到日新月异的变化，也会为他们不断地更新服务。我觉得这种感觉很棒，因为我无法想象自己做别的事情也能为人们带来如此巨大的影响。

（编辑：陈彦辛）

9

Guy Steele

叶淮光 译

Guy Steele是个真正的程序语言多面手。当我问他曾经认真地使用过哪些语言的时候，他列出了下面这一长串：COBOL、Fortran、IBM 1130汇编、PDP-10机器语言、APL、C、C++、Bliss、GNAL、Common Lisp、Scheme、Maclisp、S-1 Lisp、*Lisp、C*、Java、JavaScript、Tcl、Haskell FOCAL、BASIC、TECO以及TeX。他还说："这些是其中主要的语言。"

他参与了现存两种主要的通用Lisp方言——Common Lisp和Scheme的创建。他也在Common Lisp、Fortran、C、ECMAScript和Scheme的标准化组织中工作，Bill Joy还邀请他帮助制定Java的官方语言规范。现在他正致力于Fortress的设计，这是一种用于高性能科学计算的新语言。

Steele在哈佛大学获得文学学士学位，在MIT获得科学硕士学位和博士学位。在MIT期间，他和Gerald Sussman合作编写了一系列著名的论文，现在被称作"The Lambda Papers"，其中包括了Scheme程序语言的初始定义。他还曾经是一名黑客文化编年史的作者，JargonFile的最初编撰者之一，以

及《黑客词典》一书的编者（该书后来由Eric S. Raymond更新修订为《新黑客词典》）。他还在Emacs的诞生中扮演了重要角色，同时也是最早移植Donald Knuth的程序TeX的程序员之一。

Steele是ACM（美国计算机协会）会员和美国艺术与科学院院士，也是美国国家工程院院士。1988年他获得ACM Grace Murray Hopper奖，2005年获得Dr. Dobb's程序设计杰出奖。

在这篇访谈中，他讨论了软件设计，以及写作和编程的关系，他还给出了我所听过的关于正确性的形式证明的价值及其局限的最佳解释。

Seibel：*你是怎样接触编程的？*

Steele：嗯，当我还是个小学生时，我就已经深深迷恋科学和数学了，我读了很多这方面的书，比如Irving Adler的Magic House of Numbers，它是我的最爱。我也喜欢儿童科幻小说，比如Danng Dunn系列，等等。总的来说，我对科学和数学有着广泛的兴趣。所有我能找到的关于科学和数学的东西，我都读了，同时我也读到了一点关于即将到来的新奇的计算机的介绍。

Seibel：*这些都是什么时候的事情？*

Steele：1960～1966年，在我读小学期间。不过我想转折点发生在我开始就读于波士顿拉丁学校[①]——大约相当于9年级。一个朋友问我："你听说了那些放在地下室的新计算机吗？"我以为那不过是个恶搞[②]，之前有人说第四层有个游泳池，而学校教学楼一共只有三层。不过他说："不，这是真的，它就摆在那里呢。"

后来证实是文森特·利尔森先生[③]赠送给波士顿拉丁学校的IBM 1130小型计算机。他是校友，并且显然是一个慷慨的人。我朋友向我展示了一段5行的Fortran程序，我马上就着迷了。

我找到我们的数学老师，问他是不是有些什么书可以让我来学习。他给了我几本，并且以为至少可以让我忙活一个月了，不过实际上，仅仅一个周末我就搞定了它们。在1968年感恩节的那个周末——是的，那是个长

---

① 拉丁学校，以拉丁语和希腊语为主科的文科中学，作为大学的预料。13世纪始于欧洲。——编者注

② 原文为story，双关语，既可以解释为"恶搞"、"故事"，也可解释为"楼层"。这一句颇有递归的意思，因为Gug steele的背景是Lisp。

③ IBM前CEO。

周末[1]——我自学了Fortran。从此我就被编程套牢了。

拉丁学校的朋友和我因为IBM 1130而迷上了IBM，我们每隔个月就去一趟市中心的IBM办公室，和那里的人们交谈，偶尔用我们仅有一点钱买一些刊物。

市中心也有一个书店，那里卖一些新奇的语言（比如PL/I）的书，我们偶尔也在那里买几本。就这样，在拉丁学校我们熟悉了IBM的设备。我们只有1130，很眼馋360系统。我们在书本上见过它，不过没有机会接触真正的。

1969年的春天，在高中学习项目上，我有机会进入到了MIT。这非常棒——星期六早上去，那些大学生会教你一些很酷的玩意。我学习了群论和计算机程序设计，其他的我已经忘记了。我就深陷其中，并且对MIT的氛围熟悉了起来。通过高中学习项目，我们有机会接触到IBM 1130和DEC PDP-10[2]。这样我们也开始了解DEC公司产品线了。

我们这些高中生很快熟悉了中央广场的DEC公司办公室，那里本来是接待MIT的学生的。他们不介意高中生进办公室来要一些参考手册，这可真是太好了。我还是拉丁学校的三年级还是四年级的学生的时候，就和一个朋友一起向DEC公司提交了一份在PDP-8上实现APL语言的计划书。他们认真评估了这份计划书。大约一个星期后，他们告诉我说："这样的，我们不赞成这个点子，但是很感谢你做的计划书。"

Seibel：*你编写的第一段有趣的程序是什么？*

Steele：嗯，我首先学习了Fortran语言，不过在我开始学习IBM 1130汇编语言之后，事情才变得真正有趣。我能想起来的最早的有趣的程序是一段能产生上下文关键字索引的东西。IBM为他们的用户手册提供一个被称作是快速索引的东西：给定一个关键字，你可以从一个按字母排序的索引中查找，关键字的前后是这个关键字的上下文的一些单词。

Seibel：*关键字出现处的上下文？*

Steele：在原始文档中关键字出现处的上下文。每一栏中间是按照字母排序的关键字，左右两栏是大量的上下文。我想我在1130上面解决了这个问题。考虑到130只有4k字的内存，很明显我得把记录保存在磁盘上。因此我学到了很有效率的out-of-core排序算法[3]。这段程序有趣的地方不仅在于产生了上

---

① 长周末指赶上节日的周末，可以多休一两天。——编者注

② DEC公司生产的小型机系列的代号。PDP是Programmed Dute Processor（程序数据处理器）的首字母缩写。——编者注

③ 也就是基于外存的排序算法。

下文关键字的索引，还实现了多路out-of-core合并排序。它相当有效。不过内存中的排序用的是冒泡算法，因为我还没有那么老到。在内存中我本应该也使用合并排序，不过那时我并没有意识到。

Seibel：*从你意识到地下室真的有一台计算机到你编写这段程序之间，过了多久呢？一个月？还是一个星期？*

Steele：应该是在最初的两年当中，我不记得是不是在第一年。1968年秋天我学习了Fortran语言。APL语言是我学的第三种语言，所以我应该是在圣诞节期间或者刚过完圣诞节时学习了汇编语言。我清楚记得1969年的春天我学习了APL，因为那时春季计算机联交会在波士顿举行。

IBM的展位上展示了所有的IBM产品，特别是APL 360，而我就在他们展位附近处闲逛。交易会结束后他们准备扔掉一大堆电动打字机终端的演示纸。就在那时我走上前去问："您准备扔掉这些纸吗？"那位女士困惑地看了看我，说："那，给你吧。"好像她给我的是一个圣诞节大礼包。不过对我来说这的确算是圣诞大礼包。

Seibel：*那些纸上是什么呢？*

Steele：这些从电动打字机终端上出来的、有着扇形褶皱的纸，是他们过去两天中展示APL用的。那上面刚好有他们随意键入的一些程序小例子。从这些程序示例还有交易会上那些小册子中，我自学了APL。

Seibel：*你在MIT很自在，但最终还是去了哈佛读书，而同时又在MIT打工，这是怎么回事呢？*

Steele：我申请大学的时候，申请了三所学校，MIT、哈佛还有普林斯顿。我最想去的是MIT。三所学校同时都录取了我。波士顿拉丁学校的校长Wilfred L. O'Leary是个老派的学者。老先生人非常好，打电话给我父母说："你们知道令郎拿着哈佛的通知书实际上却考虑去MIT吗？"他就这样向我父母施压，我父母转而对我施压，最终我决定去哈佛了。

我父母继续找我的麻烦，让我去打一份夏季工，而不是在家呆着——你知道，做父母的都会这样。我很清楚自己的兴趣是计算机，我可不想去快餐店摆弄汉堡包。我面试了打孔工的工作，并且自以为是完全能够胜任的。但是没有人愿意雇用我，部分原因是我还不满18岁，可找到后才明白。他们听了我的叙述后说："不要打电话给我们，我们会打给你的。"然后就了无音讯了。

大约7月初我听说MIT的Bill Martin正在寻找Lisp程序员。我想："啊哈，机会来了，我了解Lisp啊。"我过去经常出没于MIT的时候，从AI实验室搞到了一些Lisp文档的副本，我也曾偷偷溜进实验室摆弄过计算机。那些日子里大门是敞开的，反越战抗议发生后门才被锁上。我在高中四年级时在IBM 1130计算机上实现过我自己的Lisp程序。

于是，我这个不知道哪里冒出来的小瘦猴儿，跑到Bill Martin的办公室，从门口探进头说："我听说你在招Lisp程序员。"他并没有嘲笑我，只是打量了我一下，然后说："你得先做做我出的Lisp考题。""没问题，现在考怎么样？"我就坐了下来，花了两个小时来答题。完成后我把试卷递给他，他用了十分钟浏览了一遍，然后对我说："你被录取了。"

Seibel：*是不是你在高中学习项目当中学习过Lisp呢？*

Steele：一点点，更多的是Fortran和其他的语言。

Seibel：*在你起步时有没有遇到对你很重要的导师呢？*

Steele：在拉丁学校期间我的数学老师对我的适当鼓励刺激很重要。9年级的Ralph Wellings，就是在那个感恩节周末借我书的那位老师，和我做了一个交易。他说："我注意到你在所有数学测验中都得到了100分。"我可以让你在每周的前4天数学课都呆在计算机室，不过在第5天数学课的测试上你必须得到100分，否则，交易就自动终止。"看，这就是激励。在那年余下的时间中我变成了测试高手——我特别刻苦地学习数学，因为这能让我接触到计算机。更好的是，第二年我的数学老师没有与我做同样的交易，这正好，因为我对那一年的数学了解不多。他们做出了恰当的评估。我的老师都是非常好的老师，我要学什么他们总是为我大行方便。

Seibel：*在那之后，随着你更深入地学习计算机，有没有特别的人在这领域帮助你呢？*

Steele：有，当然就是雇用我Bill Martin。还有Joel Moses，他领导着Macsyma项目，我受雇于MIT期间就在这个项目组里。

Seibel：*在整个大学期间，你一直在做这个项目吗？*

Steele：是的，我在哈佛读书的时候就一直是MIT的一名雇员。在暑假时是一份全职工作，开学后它就变成了一份下午的兼职工作。我尽可能地把哈佛的课程安排到早上，这样我就可以搭乘地铁去MIT，用两三个小时来编程，然后再回去。

Seibel：一直在用Lisp做Macsyma项目吗？

Steele：是的。具体说就是当Maclisp解释器的维护人员。JonL White原本同时负责解释器和编译器的工作。他后来成为了一位相当厉害的编译器大师，而我则负责解释器，这个分工不错。就这样，JonL White成了我的导师。Macsyma项目组里所有的人都很关照我。我也得以结识一些AI实验室的人，所以当我申请读MIT的研究生的时候，很容易就被录取了，因为他们已经了解我，并且知道我在干什么。

Seibel：你得到了计算机科学的学士学位？

Steele：是的，我本来打算主修纯数学，并且都安排好了我的课程。后来发现我对什么无穷维巴拿赫空间完全没有感觉，简直要害死我了。幸好，我已经在业余时间学习了足够多的计算机课程，这让我可以在专业的时候很主动。确切地说，我转去修应用数学专业，而计算机科学是应用数学的一个分支，在哈佛应用数学又属于工程学的一部分。

Seibel：在哈佛你使用哪种机器？

Steele：DEC PDP-10。在学校里有一台PDP-10，不过我想它主要是给研究生用的。本科生只能通过电传打字机终端接入商业系统，那是哈佛租来的。

Seibel：如果有可能让你重新学习编程，你会有什么不同吗？有什么事情你希望更早一点完成的吗？

Steele：并不是一开始在我的脑海里就有特定的目标。我对我选择的这条路也不后悔。回首往事，我想我是一个幸运儿，受惠于一系列有趣的巧合，或者说，恩赐。

现在我意识到，实际上同时在MIT和哈佛的经历是很不寻常的。我可以跑来跑去，然后说："这条河[①]那一边的教授是这样说的。"而这一边的教授就会说："哦，别信他，你应该这么想。"这很快让我的视野更开阔。

作为高中生就能进入MIT是另一个相当不寻常的经历。我15岁时就可以摆弄那些价值数百万美元的机器，在那时1百万可真是一笔相当大的钱。所以，我当然没有抱怨，没有后悔，也不会有任何得陇望蜀的想法。我本性也是个随遇而安的，既来之，则安之。

Seibel：与那时相比，对于编程的思维方式，有哪些大的改变？除了认识到

① 哈佛大学与MIT只有一河之隔，即查尔斯河。——编者注

冒泡排序不是最好的排序算法之外。

Steele：我想对我来说，最大的变化是认识到你不可能了解运行在你计算机上的所有一切。有些事情绝对超出了你的控制，因为不可能了解所有软件的一切细节。而在上世纪70年代计算机仅仅有4k字的内存，你完全可以做一个内存转贮，然后一个字一个字地去检查是不是你期望的。阅读操作系统的源码，了解它是如何运行的自然也正常。我也确实这样干过——我研究过磁盘管理程序和卡片阅读机程序，然后实现了我自己的版本。我觉得我自己了解整个IBM 1130是如何运作的，或者至少可以说我了解我自己想了解的所有事情。不过现在你再也不能这样干了。

Seibel：*在你学习编程的时候，有没有什么书对你特别重要？*

Steele：在70年代的是当然是Knuth的TAOCP《计算机程序设计艺术》。

Seibel：*你从头到尾地读过吗？*

Steele：差不多每页都读过，差不多。我做了几乎所有我能做的习题。一些被称作是高等数学之类的东西我不太明白，我做了些注释或跳过了那些我不懂的。不过头两卷和第三卷的大部分我都很认真地读过。还有Aho、Hopcroft和Ullman编著的那本算法书[①]——我想我是从这里面真正学到如何排序的。我得从我的书库里查找一下，看看能不能还记得有什么别的书。我是个收集狂——这类书我都有。不过这两本是我首先想到的。还有Lisp的书。Berkeley和Bobrow编辑的III Lisp[②]：是主题各异的论文集，不过我从中学到了很多有意思的东西。然后我开始读《SIGPLAN公报》和《ACM通讯》。那些日子里的ACM通讯可是有很多真正的技术内容，非常值得一读。

我要提及两件事。第一件，当我在拉丁学校开始对科学感兴趣的时候，我决定从事计算机科学相关的事情。有一天有个导师问我："你考虑过成为ACM的学生会员吗？"我不知道他的名字。不过我从那时起就非常感谢他，这给了我很大的鼓励。

而我上哈佛以后，每当早晨有点空闲时间，我就会去Lamont图书馆做两件事：按照我的方式，从后往前阅读《科学美国人》，或者从前往后阅读《ACM通讯》。对《科学美国人》，我特别注意Martin Gardner的所有数学游戏专栏。

---

① 是指*The Desing and Analysis of Computer Alogorithms*（《算法分析与设计》）一书。——编者注

② III指Intformation International，Inc.。——编者注

对《ACM通讯》则阅读所有我感兴趣的文章。在1972年这本期刊还只有15年的历史，所以不难把它们全部都过一遍。

Seibel：*阅读所有文章在那时比在今天要容易得多，一个人还是希望了解这整个领域的。*

Steele：是的，你有希望了解这个领域。有很多只有一页长短的文章，让你知道："这儿有一项新的散打技术。"我读了很多这类的文章。

Seibel：*旧的文章我个人常常觉得不太容易理解，因为它们和一些旧硬件或者旧语言联系得比较紧密。*

Steele：是这样的，需要是创新之母——一个想法的出现是因为在一个特定的环境下需要它。过了一段时间，大众认识到这个想法很重要。然后你需要摆脱环境的局限，展现出核心思想本身凸现出来，这样就可以流行好几年了。"这个神奇的技巧可以按位逆转字。"他们给出了7090汇编语言的一些东西。有些很有趣的数学思想在里头，不过他们还不能从中抽象出来。

Seibel：*我猜那是Knuth的工作，对不？*

Steele：Knuth和像他一样的人，当然了。

Seibel：*的确很多人从学校里开始，在指导下学习计算机科学知识。但是还有很多程序员是没有正规学历背景的，只是边干边学。对这个你有什么建议吗？你怎么开始阅读那些技术论文，如何抓住要点并理解它呢？应该从ACM最初读起，一直读到现在吗？*

Steele：嗯，首先，我得说通读《ACM通讯》并不是我刻意博览群书成为一名伟大的计算机科学家的计划。我阅读是因为我有兴趣，有内在的动力去学习那些资料。所以我觉得这里有两个因素：第一是有内在的动力，想要阅读它们，因为你有兴趣或者说你觉得能提高你的技能。

另一个问题是你怎样才能发现好的东西？当然，对"好"的认识也是三十年河东，三十年河西。今年你觉得是真正好的十年后说不定过时了。我觉得你可以拜访一位曾经经验丰富的前辈，问他觉得什么才是好的东西。对我而言就是Knuth，就是Aho，Hopcroft和Ullman。还有Gerald Weinberg《程序设计心理学》，那本书今天还是非常值得读一读的。Fred Brook的《人月神话》也给我一些启示。

那时候我流连于MIT书店的计算机科学书架，下决心每个月到那里去一次，去翻翻那些书架。今天你再去一个书店，它的计算机书架规模可能是那

时的10倍了，不过其中大部分图书都是如何使用C或者Java的。但还是会有一部分理论背景、算法这类书。

Seibel：*另一种阅读——我知道你认为很重要的——是代码阅读。你是怎么样以你的方式切入不是你编写的一大段代码的呢？*

Steele：如果那个软件我知道如何使用，但不了解内部的工作机制，我通常会选择一个特定的命令或者交互行为然后追踪下去。

Seibel：*执行路径吗？*

Steele：是的。如果我要开始阅读Emacs源代码的话，我会说："让我们看看'向前移动一个字符'的那部分代码吧。"即使我不能完全理解，我至少会知道它使用的一些数据结构以及缓冲区是怎么表示的。如果我足够幸运，我能找到缓冲区增加一个的地方。一旦我理解了之后，我接下来会尝试"后退一个字符"、"删除一行"。通过我的方式就了解越来越多的使用方法或者交互，直到我觉得我能够按照这种方式追踪代码的其他更重要的部分。

Seibel：*"追踪"是指查看源代码在心里执行它呢，还是要在调试器中启动它，然后单步执行进去呢？*

Steele：两种方式我都会做，我会用单步调试器对付那些70年代或者80年代的小一点的程序。今天的问题是从启动程序到它真正可以做点什么，这中间有一段很长的初始化过程。所以更好的办法是找到主命令循环或者中央控制子程序，从那里开始追踪。

Seibel：*当你找到了那些以后，你是设置一个断点然后单步跟进去呢，还是仅仅在脑海中想像它们的执行过程？*

Steele：我更愿意做桌面检查——就是阅读代码想象它会做什么。如果我确实需要理解整段代码，我会坐下来试图按照我的方式来通读代码。不过你不能一上来就这样做，应该先在脑子中有了事情的组织框架。现在，如果你足够幸运，程序员会留下一些文档或者规则的命名，或者合理组织文件的顺序，方便你快速阅读他们的代码。

Seibel：*什么是合理组织文件的顺序？*

Steele：很好的问题。使我想起了诸如Pascal这样的程序语言的一个问题，Pascal是为只过一遍的编译器设计的，源文件中的过程倾向按照自底向上的方式组织，因为在使用过程之前你必须定义过它们。也就是说，阅读Pascal

程序最好的方法实际上是从后面读起，因为这样你就会看到程序自顶向下的结构。现在（编译器）形式如此多样，你也就不能指望什么了，除非程序员有一颗很强的责任心，将一切事情安排得井井有条，有助理解。不过，第三点，我们现在也有很棒的IDE来帮助你查看交叉引用，也许程序的线性组织也不再是那么重要了。

第四点，我个人非常不喜欢IDE的一个原因是，你看完了所有内容后，还是很难理解。在图形迷宫里乱蹿，很难知道所有地方都走到了。但如果你得到的是线性顺序，就会确保所有事情都会梳理到。

Seibel：*那么在你写代码的这些日子里，你是不是尽量按照自顶向下来组织代码呢？高层函数出现在它们依赖的低层函数之前？*

Steele：我尽量表现高层的想法。最好的表现方法可能是展示一个中心的命令，控制的过程，以及向下面分发的事情。或者，重要的事情可能是首先要展现数据结构，或者说较重要的数据结构。重点是要像讲故事那样去表现想法，而不是只罗列一堆代码在那里。

在MIT工作时，一件很棒的事情是可以随意访问到没有加密的代码，它们全是非常聪明的黑客们的作品。于是我读过了ITS操作系统，也读过了TECO的实现和Lisp的实现。还有第一个相当漂亮的Lisp格式打印程序，是Bill Gosper编写的。事实上，我在读高中的时候就读过它们了，还试图复制一些到我个人的1130实现当中。

如果没有接触到在其他机型上的现有Lisp实现，我一定不能为1130实现Lisp的，我根本就不知道怎么做。这是我个人教育的一个很重要的部分。时至今日，我们面对的部分问题是软件已经变得很有价值了，大部分软件都是商用的，也就是说我们不再有可作为免费例子的优秀代码来参考了。开源运动在某种程度上扭转了这一点。如果你愿意，你可以深入阅读Linux的代码。在我那个时候阅读TeX的代码是一项很有意义的练习，因为它是一大块良好组织的、易于调试的代码。

Seibel：*通常我只是在需要了解事情是怎么运行时，才会去阅读代码。在阅读像TeX这样的程序时你是一种什么心态呢？*

Steele：有时我是带着特定的目的的，因为我想解决某个问题。我记得有两次，我不能通过阅读The TeXbook来解决我写的TeX宏中的bug，你必须向前，阅读TeX源程序来找到哪一个特性是如何工作的。两种情况下，我都在15分钟之内找到了我的答案，因为TeX源程序的文档是一份非常优秀，有着丰富

的交叉引用。它本身就令你眼界大开——程序可以如此完善的组织索引，你可以很快找到你要的东西。

我从中学到的另一件事是，大师级的程序员是如何组织数据结构的，如何组织代码使它更易于阅读。Knuth巧妙构思了TeX程序，你甚至可以把它当作一部小说去阅读。你可以从头至尾顺序阅读它，也可以在遇到问题时跳跃地的阅读。当然，这对作者来说是很巨大的工作量，因此只有非常少的程序是这样写的。

Seibel：*当你做完这些之后，你从中学会了什么？*

Steele：我对它是如何组织的有了一个很好的认识，我可能从中学到了一些想法来如何更好地组织自己的代码。就像我不认为我能像福克纳或者海明威一样去写作，我也不认为我自己能按照Knuth那样的风格去写代码。不管怎么说，阅读这些作家的小说多少影响了我对英文风格的认识。也许因为种种原因我明智地决定不是像海明威那样去写作。这毕竟是个宝贵的经验，更何况还有阅读很棒的小说或者精妙代码片断所带来的乐趣呢。

Seibel：*你写过文学程序吗？*

Steele：不是严格遵循Knuth提倡的方式。我想它在这些点上影响我的个人风格——在写一段子过程前我常常先写一段散文，不过我不是完全遵循这种方式。有时我也会好奇Knuth是怎么做的，在准备好发布之前，他的编程之旅是怎样的。我不能想象他是如何做到的。

Seibel：*你尝试过文学编程，但是它没有让你的编程变得更有效率或者更有乐趣？*

Steele：部分的原因是我没有想为自己写很多工具。他所创造的工具是Pascal和C编写的。Pascal我能理解，但是我很清楚C的缺陷，我不确信使用文学编程工具就可以克服这些缺陷。如果他为Common Lisp准备了文学编程工具，我可能会飞快地投身其中。

Seibel：*暂且把文学编程放在一边，让我们再说说代码阅读吧。你觉得自己可以从头到尾通读写得很好的程序，还是说它们像是超文本的组合，你必须找到自己的路径呢？*

Steele：我没有必要反对超文本，不过我认为如果一个程序写得够好，一定有什么东西是和结构相关的，会引导我按一种合理的顺序到达各个部分。你知道，它不仅仅关于程序的功能，而是要说出一个故事。这个故事将告诉你

程序是如何组织的，还将讲述程序执行时的场景。有人希望知道关于程序的事情，无论它是每个过程之前的一大块注释，还是一份单独的概观文档，亦或仅仅是变量名的选择，所有这些都在向你讲诉着这些故事。人们认为优秀的程序员，一个真正的优秀程序员，除了告诉你这个程序真正干了什么的故事之外，也会传达故事内容。

Seibel：*最近你读过什么很有趣的代码吗？*

Steele：很难找到值得一读的代码。我们没有收集一些大家认可的程序，说："这是一份卓越的代码，每个人都应该读一读它。"所以，往往只是一页的代码片断出现在论文中，而不是现有程序的大量代码片断。也许我最近阅读的代码是我自己团队编写的，作为Fortress实现的一部分代码。还有一些Java的库文件。

也许最近我纯粹因兴趣而阅读的一段非常精妙的代码是由George Hart完成的。他是一名数学家，多面体专家。他写了一段非常有意思的代码，使用VRML在浏览器中产生和显示复杂多面体。他写了这段巨大的JavaScript代码，来产生VRML代码然后把它们输入到VRML显示程序。

我决定在几个方面增强它，所以我深入其中，通读了他的代码。然后开始尝试不同的增强效果，并且试图理解到底发生了什么：做几个变形的多面体等。我也故意制造了一些错误——测试其中的一个松弛算法，试图展开多面体的顶点来使它显示得更方便和更漂亮，偶尔我也引入数学上的不稳定性，导致怪异的图案产生。无与伦比的乐趣，单纯是陶冶自己。这大约在六七年前吧。

Seibel：*阅读和修改在多大程度上是交织在一起的？在桌子旁读打印出来的代码还是在计算机上做点小修改但是并不去执行它看看会怎样？*

Steele：实际上我的确会把代码打印出来。我会坐下来慢慢阅读代码，常常还会做些标记和注释，问自己问题，等等。然后我回到计算机旁开始键入一些东西，看看执行的结果，并且追踪它们。

Seibel：*这种情况下你是想要修改这些代码，于是你那样做了。不过你有没有只是从阅读代码中得到一些教益或者乐趣呢？打印出来，阅读代码，也许随手记些问题，最后把它记下来？*

Steele：也会这样，如果我停在这个点上，那它就只是一次阅读代码的有益练习。我学会了一点VRML、一点JavaScript，那就是它们不像我喜欢的那样

有那么多的抽象。动态类型对我个人来说形式上有点太自由了——在一个面向对象的语言里。

Seibel：接下来我们谈点软件设计。你不再像过去的日子里那样写那么多的代码了，不过你是怎么开始设计一个新软件的呢？你是坐在计算机旁开始写代码，还是先带一大堆图纸坐下来，还是什么别的？

Steele：在这儿我得特别小心，因为记忆太容易修正了，我会说："早在上世纪70年代我好像就是这样，所以我一定是这样做的。"我试图回忆起我到底是怎么做的。

有时我会画画流程图——我确实有一套IBM的流程图模板和一堆这样的图纸。并且我是在结构化编程时代之前开始学习编程的，所以我的设计有些是结构化的有些不是。当我开始了解结构化编程之后，我意识到其中有些很不错的思想，所以我想在70年代写的汇编代码自然而然有结构多了，我也开始用if/then/else或者循环之类的逻辑去显式地思考，我也对汇编代码的结构越来越关注。

我常常为程序的输入做一个列表，填入我能够想到的值，然后写下我想要的是哪一种输出。有时我也会写几个简短的例子，最近就发现了一个我最早写的一段APL程序，那时大约十五六岁吧。这是一小段APL代码——在我真正测试它之前把它记在纸上，这是第一个。连在一起的是另一张纸，上面看起来好像是输入/输出之间的相互关系的例子。这上面有bug并且和代码不符合，但是毕竟我尝试制造了几个我认为应该这样使用程序的例子。它正是我认为应该在一个打印输出终端上的打印结果。我想这就是可由程序引起的一系列的过程。

自从我到Maclisp项目工作后，项目就提供了一个结构。几乎我所做的所有的事情都是在一个现存的、巨大的函数集合中添加新函数。已经有很多的例子纪录了函数文档看起来应该怎样，所有要做的只是添砖加瓦。

Seibel：你说过你从JonL哪里接管了解释器，JonL原来是同时负责解释器和编译器的。

Steele：我们在设计问题上相互合作。我是一个新组员，因此他会说："这儿有一个我们应该有的函数，它是这么工作的。你为什么不实现它呢。"或者更常见的是我们从Macsyma实现者那里拿到需求："我们需要有东西这么做。"然后JonL和我就把头凑在一起然后说："好的，为什么我们不设计一个看起来一样的接口呢。"然后我就起身离开去实现它了。

Steele：必须在解释器和编译器上实现的新语言特性就是这样添加到Maclisp中的吗？

Steele：是的，语言特性。其中许多是面向系统的——它们需要控制资源或者分配内存页。我实现了一个新的数据结构hunk，这可能是在我们语言设计当中添加的一个最大的灾难。这个数据结构本质上通过两个以上指针连接的单元，它移动起来真是非常要命，因为我们耗尽了在PDP-10上的地址空间。要知道PDP-10只有18位的地址空间，在一个链表里，50%的指针都用于维护这个链表的结构，然而如果使用线性化的一大块，可能只需要八分之一的指针就可管理，也就是你可以用一种更好的方式使用内存。

Seibel：*因此你是按照一种增量的方式得到新特性的需求，那你怎么维护其中的一致性？如果你只是按照一种最明显的方式添加东西，最终你只会得到一个包罗万象的拼凑物。*

Steele：曾经有过一两次重新组织。我想最值得一提的是重新设计和重新实现语言中所有的输入/输出操作。这被称为新I/O设计，我参与了其中，大约是在1975年或是1976年。目标是让旧I/O系统只允许一条输入流和一条输出流，加上能和控制台交互。我们意识到如果用真正的Lisp对象表示I/O通道，就可以获得更大的灵活性，最多可以打开15个I/O通道。

另一个推动因素是Maclisp被移植到了其他操作系统上面，每一个站点都有PDP-10系统的变种。当察看所有客户时，我们发现有六个需要支持的操作系统：TENEX、TWENEX、ITS、TOPS-10、WAITS以及CMU的变种。

于是在一个夏日我和JonL坐下来开始设计一套新的API，那时还没有称之为API，而是一些函数描述，比如可用于创建、打开、关闭文件对象，系统化地实现删除或者重命名，得到目录列表。

我得到了一个列在纸上的、全新的Maclisp特性列表，返回到我父母暑假住处待了一个星期，带着6本操作系统参考手册和那个列表，每天花6个小时修改代码。

对每个特性（诸如“重命名”函数怎么完成，我必须心中有数。因为在这6个操作系统中，如何和操作系统交互来重命名一个文件的细节是不同的，不过可将它们分成3类——TOPS-20变种、TOPS-10变种以及ITS。

我度过了充实的一周，请注意，设计和实现工作都不是坐在计算机终端前实现的，都是案头工作。如此一周后我回到MIT，接下来的一个月我键入所有代码，调试并测试代码。

Seibel：你为什么要这么干呢？

Steele：我这么做的原因是每一个我要实现的函数，都被充分地研究过。就像我说的，我必须阅读6个操作系统的说明，我得花一个小时这么做然后只写30行代码，很可能要重复三次。所以我若坐在终端前却不能这样做，也就没什么意义。我又不用Google什么或者访问在线文档，而且花在打字上的时间也不多。所以最好我还是好好利用面前的桌面空间摆放纸质文档吧。

Seibel：时至今日你还是认为有时候关掉计算机，清理你的桌面是正确的做法吗？

Steele：是的，我还是这么做的。事实上，我发现有时真必须得关掉计算机，因为如果风扇在我后面呼呼作响，似乎在诱导我"检查你的电子邮件，检查你的电子邮件"。所以我把它关掉，或至少是把它转入休眠状态，走到房间另一边的桌子旁，摊开我的文件开始思考，或者在白板上工作，等等。

Seibel：我曾读到过你引用Fred Brook对流程图和表格的论述，这么说的吧："给我看你的接口，我不需要你的代码，因为它们是多余或无关的。"当你工作在像Java这样的语言上时，你是不是从接口开始设计的？

Steele：是的，和过去相比，我变得越来越爱用接口了。描述方法的输入、输出和动作，但是不写代码。我喜欢写这样的东西。我也喜欢亲自写代码，但是现在已经没有以前写得那么多了。当然有过怎么做的经验很重要，这样你就不会设计不可能的规范。当你设计接口时你应该对实现的样子有个认识。至少有一个实现的想法。如果有人提供了一个更好的想法，自然更好。

Seibel：除了可能实现的方式外，你怎么判断你的接口是好是坏？

Steele：我通常考虑通用性和正交性，要符合做事情的通常方式。举例来说，如果没有特别好的理由，你不会把除数放在被除数之前，因为在数学中我们通常采取另一种做法。因此，要考虑做事情的惯例。

我已经做过足够多的设计了，因此可以考虑我以前做事情的方式以及评判这种方式是好是坏，而且有些类似的东西我以前也设计过了。举个例子，在设计Java的数值计算函数之前，我已经为Common Lisp实现过数值计算函数了，我还为C语言的数值计算函数写了文档。我知道这方面一些实现上的缺陷和规范上的缺陷。我把大量的时间花在处理边界情况上。这是我从Trenchard More的APL数组理论中学到的，他的论点是如果你处理好了边界情况，那么在中间的部分通常会不辩自明。只是他并没有这么说过，我猜我

是从他那里得到这个结论的。

反过来，用这种方式设计中间部分的规范，它在边界上也是自然而然正确的方式去设计规范，而不是把边界当做特殊情况处理。

Seibel：当你在MIT时，某种程度上你也参与了Emacs的诞生。不过Emacs的早期历史有些模糊，众说纷纭。这个故事你的版本是什么？

Steele：我在其中充当了一个标准员的角色。最先发生的，是有一个显示模式可以把TECO变成一个所见即所得的编辑器。在我们24×80的屏幕上，21行显示的是缓存中的内容，最底下的3行则是TECO的命令行，你在这里键入TECO的命令，只有敲击Alt模式，这些命令才会被执行。随后就是实时编辑模式，一条TECO命令就把你带到另一个模式下，它意味着TECO不必等待进入Alt模式就会执行一个字符的命令。键入一个字符，马上执行一个命令。键入另一个字符，再执行一个命令。大多数的打印字符可以自行插入，然后有控制字符用于向前、向后、向上、向下移动。它非常非常初级——看上去就是Emacs的非常原始的版本。

然后突破就来了。灵感就是我们有了这个主意：取一个字符，在表中查找它，然后执行TECO命令。为什么不把它用于实时编辑模式呢？于是每一个你可以键入的字符都出现在这张表中。然后默认表规定：打印字符可以自行插入，由控制字符来做这些事情。不过让我们把它变得可编程并看看会发生什么。接下来马上在MIT就有四五个聪明人有了各自的想法。几个月后，TECO大约就有了5个完全不兼容的GUI接口。

Seibel：因此本质上来说，他们只是在定制键绑定？

Steele：是的。他们每人都很明白做得越多的事情就应当简洁，其他情况则可以承受。例如一个伙计关心的是键入Lisp代码，于是开始想办法去对应括号表达式。另一个伙计对文本更关心一点，于是他对诸如移动单词、大小写转换还有大写所有字母的命令更有兴趣。这就是Emacs里那些命令来历。

不同的人对于键绑定应该怎样组织都有不同的想法。我是做Lisp系统支持的，我常常被人叫到他们的终端前，帮助他们解决问题。我很快发现不能坐下来在他们的TECO前帮助他们修改代码，因为我得面对他们的一套绑定键，而我对其用法完全摸不着头脑。

Seibel：这些人里包括Richard Stalllman吗？

Steele：不过，Stallman是TECO的实现者和支持者。他提供了内置的实时编

辑模式，尽管我认为是Carl Mikkelsen做的早期版本。他提供的键绑定特性使这成为可能。

不管怎样吧，似乎有四个宏包并且它们相互不兼容，我决定充当标准员，或者说社区协调人。我看到有些东西在我们的社区中丢失了，那就是能够在终端前容易地帮助其他人的能力。我说："那好吧，我们已经做过一些尝试，我们已经看过很多想法。我们能不能同意一套博取众长的通用的键绑定做法？"

我特意带了一些纸张，然后跑遍大楼，和这些家伙交谈，多次分别拜访他们，试图达成某种共识。我努力让大家对内容取得共识，然后我利用他们的设计，试图组织起合适的键绑定，目的在于使它们更规则一点，也更容易记忆一些。由于我不是一个人体工学专家，我完全没有考虑盲打上的方便，我主要的考量是容易记忆。这也是为什么Meta-C、Meta-L、Meta-U分别代表了全大写、小写和大写的原因。

Seibel：颇有点讽刺意味的是，有些命令从你的头脑中涌出，然后就出现在你的手指上了。我肯定你也有这样的经验：有人问你一个每天重复上千次的键绑定是什么的时候，你却说不出来。

Steele：确实我的妻子有过这样的经历。而我自己不太能感受到这点，一个原因可能是我并不是特别棒的盲打专家。她不用Emacs已经有20年了，我为她的Macintosh准备了一个，她坐下来，键入一些东西，然后说："我怎么保存？我忘了怎么保存文件了。"随后她意识到她的手指已经完成了，但是她不知道她键入了什么。于是她重复了一遍，盯着她的手指然后说："哦，原来是Control-X Control-s。"但是她实现上想不起语义上的命令是什么了。

Seibel：因此你制定了这一套键绑定的标准。大家觉得如何？对它满意吗？

Steele：嗯，大家已经这样做了。然后我坐下来，开始打算实现它们。同时我们也有另一个主意是，如果去除空格和删除所有注释，你可以让TECO的宏执行起来快速很多。TECO解释器的工作方式是一次解释一个字符，当碰到一个注释的时候它需要花费时间来跳过这些注释。所以我们有了这个主意：一个很初级的TECO编译器可以去除大部分的空格和注释，可以在一个格式中做几件小事情，这会使运行速度快一些。

因此我开始尝试搭建这个宏压缩器的一个版本，我想这实际上是基于Moon的早期想法，我不认为是我本人发明了这个主意。我开始考虑如何组织初始分派，如何组织最早从现在其他宏包的实现当中借鉴过来的几个过

程，我试图整合它们。大约就在那个时候Stalllman出现了，他说："你在干什么？看上去很有点意思。"他立马投身其中，可能是因为他了解TECO的内部机制，他实现起来速度可以比我快数十倍。

因此我认真地在Emacs的实现上只工作了4周或者6周的时间。很明显Stallman了解那个程序是什么。我想转回去继续研究生的工作，因此Stallman做了99.999%的工作。我只是扮演了一个催化剂的角色并且开启了实现的工作。

Seibel：*换个话题说说，现在的理论计算机科学是很数学化的。对于工作的程序员来说，能够理解Knuth的数学有多重要呢？还是这样的说法？"我需要进行排序。我要去翻翻Knuth，找到他说'这是最佳算法'的地方，然后依葫芦画瓢地实现之。"*

Steele：我不清楚。Knuth的有些部分我也不能理解，因为我没有接受过数学训练。特别是涉及高等数学或者连续数学的时候，我在这些方面有点薄弱。我想我的长处在于组合数学、排列、群论，等等，我总在反复用到这些，它们才是我手中的金刚钻。我并不认为每个程序员都需要这样，但是我认为数学把很多概念做了形式化处理，这些概念程序员在日常工作中也是需要的。

我给你一个例子，关于我最近的并行编程语言工作Frotress项目。假设你想把一堆数加起来，你可以这么做，把一个寄存器置为0，然后一次加上一个数字——经典的串行算法。

注意，这取决于加法具有不变性。你需要从0开始，细微的观察，但是的确是这样。

还有另一种策略——你可以把所有的数排成一排然后两两相加，得到一堆和数，然后继续两两相加，直到最后得到一个数为止。当然如果中间得到奇数个数字的时候，你需要留出一个放到下一轮。这样也行得通。如果你使用的是浮点数，这实际上得到精确度更高，尽管这种簿记开销有时候并不值得。

注意，这与从0开始每次加一个会得到相同的结果，至少当你操作整数的时候是如此。这是因为加法具有结合性。换句话说，你如何分组数字并没有影响。

接下来是第三种策略。假设你有一堆处理器，你可以把成对相加数打包分给每个处理器。那个"从0开始每次加一个"的算法很难并行化，不过使用打包成对相加数，你就很有效率地生成了一棵树，然后你就可以把这棵树

的不同部分分给处理器，然后处理器就可以单独处理自己的部分，到最后它们才需要交互，然后你就得到了结果。

啊哈，这很酷吧。这儿还有一个并行策略更像第一个：选择某个寄存器，把它初始化为0，然后由处理器竞争抓取处理数，在那个公共的地方做加法运算。这涉及同步的问题，不过你也能得到相同的结果。这取决于加法运算同时具有结合性和交换性。也就是说，不仅怎么分组操作数不会影响结果，如何处理相加数的顺序也没有影响。

数学家使用宏大的、吓人的名词，诸如“等价性”、“结合性”以及“交换性”来描述这个问题——这是他们的速记法。但是程序员需要了解这样的想法，即相加的顺序没有影响。还需要了解这样的想法，即重新结合也是可以的。在这个层面上我认可数学思维对程序员是重要的。

Seibel：*显然这是个很好的例子，因为任何理解算术的人都能理解它。不过你发现就编程而言这还是有点高级概念的意思吗？*

Steele：现在假设我要生成一份报告，典型的做法是你使用一堆打印语句，按照你调用它们的顺序，就会按打印出来你要的结果。不过在多核世界里我可能更愿意把生成报告的过程分开来，打包到不同的处理器上。那么好了，怎样拼接字符串呢？我可以用像加数一样的技术吗？记住这里是可结合但是不可交换的，这准确地告诉我哪些技巧可以在字符串上使用，而哪些不能。作为一个关心并行编程语言设计的设计者，我认为这些概念及其用语是很有用的。

Seibel：*说到作为一个语言设计者，这些年你对语言设计的看法有什么改变吗？*

Steele：我个人最大的变更是大约10年前，1998年我在OOPSLA上作了一个演讲“生成一门语言”。回到70年代人们在发明语言的时候会做一个完整的设计然后着手实现，然后你就得到了一门语言。即使你没有完成这门语言，但是你的思想在那儿，它是一个完整的东西。

因此Pascal是一项发明。设计中的取舍有各式各样的原因，不过它是一个完整的设计。如果说结果不那么完美——如果最后字符处理不是那么棒，那就很糟糕：Wirth已经设计了这门语言。还有PL/1已经设计过了，Ada设计过了。Ada和C++是那一代语言的尾声。也许C++没有那么过分，过去的岁月中它还在取得某些发展。

我意识到随着语言越来越复杂，已经无法一下子设计出语言的全部，语

言有必要立即进行演化，因为一下子全部设计或者全部实现都太庞大了。这使得我对编程语言设计的途径和思考的方式开始发生改变。

Seibel：你认为Java不是那样去设计的？

Steele：我想也许Java不是，但是它已经在这样做了。Java通过JCP获得演化，它更强调API而不是语言核心问题。过去的12到13年当中，许多特性已经被加入到语言中去了，而我想在20世纪90年代早期设计Java的团队认为他们当时是为特定目的设计了一门完整的语言。你知道，他们瞄准的是机顶盒。

Seibel：是的。

Steele：他们那时没有想到要为万维网编程，也没有想到后来能有那么大的用户群体。我认为他们设计了一个相当小的、自包含的核心语言，在此之上你就可以创建一套API去做各种事情。这就是它的模型和实现方式。“生成一门语言”中的部分想法就来源于观察这个过程，Java最终有点太小了，人们一直希望更多的特性来完成其他的事情。

比如有人要求加入for循环的，可以在枚举结构中迭代，这个特性后来被加入到语言中。还有来自高性能科学计算社区的压力，要求加入更多支持浮点运算等特性。这基本上被JCP拒绝了，我认为这不是技术原因而是社会原因。

所以总会面临给语言增加特性的需要，又总会有社会力量以不同方式为此设置门槛。因此我开始认为也许一门真正成功的编程语言，对于社会过程的设计和计划，你需要考虑得和技术特性一样多，并且你还要考虑两者之间的交互。这方面Fortress是我们的第一个实验器，或者说至少是我个人的第一个实验。它还处于早期阶段——只是一个半成品。

Seibel：你不认为你参与的Common Lisp树立了一门语言该怎么做的标杆吗？

Steele：是的，可以这么说，这正是和Java之类相反的一些早期的例子，促使我考虑“生成一门语言”中的几个问题。我当然很熟悉Lisp的历史，知道宏功能使得Lisp易于演化，也容易让人们做出自己的贡献。

Seibel：最近，似乎你在某种程度上参与的三门语言已经或者正在进行痛苦的重新设计。Scheme刚刚完成了R6RS。JavaScript——ECMAScript——正在经历ES4还是ES3.1的争辩。Java也在挣扎要不要加入以及怎样加入闭包。

Steele：你举的这几个例子，是的。

Seibel：这些语言是否就是因为没有足够的内置技术或社会机制，不能很容易地成长，从而导致了痛苦的成长过程？这样痛苦的经历是否总在发生呢？

Steele：是这样的，如果一门语言没有死亡，那它就会继续成长。总会有演化的压力，因为需要改变，与5年前相比，人们也需要改进工具以满足今天的需要。我不去推测一门语言是否需要成长，而是要估量在一门语言的早期设计的技术选择如何给后来的成长预备某些路径。我认为由于技术上的差异，也部分因为社会环境的差异，一些语言比其他语言要更容易成长一点。

Seibel：那么这些容易成长的例子有哪些？

Steele：我想由于其宏机制提供的灵活性，Lisp是容易成长的一个例子，在某种程度上还因为创建Lisp的群体的社会态度。

相反，Scheme有着一条痛苦得多的成长道路。部分的原因就在于Scheme社团早期形成的文化是，除非所有人同意，或者说接近于每一个人，否则不会添加任何东西到语言当中。因此这更多是一种否定的文化。至于塑造Common Lisp的社团，多数人同意就足够了。他们为了更远大的目的有意愿去接受不那么令他们狂热的新事物。

Seibel：语言的选择是多大的事情呢？有什么很好的理由选择一门语言而不是另一门吗？或者这只是口味的原因？

Steele：为什么口味不是一个好理由呢？

Seibel：比如说，我可能喜欢香草冰淇凌而你喜欢巧克力口味的，但是我们不会因此而打架。而大家会因编程语言而争执。

Steele：这是希望成为胜利者的人类社会现象。并且我真的不认为值得为此打架，但是对于给定一个任务，什么才是最有效的工具，持有自己的观点是合理的。

我有理由确信，认为一门语言比其他语言都能更好地，或者说一样好地解决所有问题是一种错误的认识。我相信对于不同的应用领域，都有特定的语言更适合。

在做算法设计的时候，使用多种语言的结合，我感到运用自如。在我需要和自己交流的时候，我总是走到白板前，写下Java、Fortran还有混杂着APL的片段。只要后来我能看出我写过什么，这丝毫不会困扰我。对于一段特定的算法，如果我觉得别的语言不太好使，这种标记法就能助我一臂之力。

问题在于，如果你使用这种标记法记录了一小段很好的想法，接下来还

是要把它放在完整的编程语言环境中去，你必须围绕它做些其他的工作，使其圆满无缺。如果你不把所有事情做好，最终得到的可能是门倾斜的语言，在一个想法上处理得很好而其他方面却笨拙无比。

另一方面，一种语言很难在所有事情上都做得很好，这部分是由于只有这么多的简洁的标记法可用。这是哈夫曼编码问题，如果这里简洁，其他的一定会要繁琐一点。因此在设计语言中，你要考虑的一件事是：什么事情我希望能够很容易地表达并且能够很容易得到正确的结果？但是要知道，为此选择的字符或符号一定会使其他什么东西要更难表达一些。

Seibel：*一种方法是Lisp的方法——把所有事情都一致变成半简洁的。这种一致性让用户能很容易添加自己的一致的、简洁的、一级语法扩展。也有很多人坚持s表达式语法。Lisp死党有个自鸣得意的观点："有些人就是没有理解它。如果他们理解了，他们就会看到完美的解决方法。"你是这种Lisp死党吗？认为如果你真的理解了Lisp，你就不会反感括号？*

Steele：不是，我想自鸣得意但还不够格。由于我研究过如此多的语言，我想要说我比很多人要理解得更好一点，就是不同的语言可以提供不同的东西。有充分的理由在多种语言中选择，而不是抓住一种语言说："这才是赢家。"

肯定有这样的项目，除了Lisp我不想用其他语言来完成，因为我对Lisp提供给我的工具很感兴趣。比如，现成的输入/输出——如果我愿意遵循Lisp的语法，我就已经有了设置好的读入程序与打印程序。这自然使你有能力实现某些快速原型。另一方面，如果按照现有的特定格式定制I/O是重要的，那么Lisp可能就不是一个好的工具。要不然我也会用某些语言，Lisp或者其他什么，实现某种转换器将其导入到Lisp世界。

Seibel：*你认真地使用过哪些语言？这应该是个很长的列表吧。*

Steele：在COBOL编程上我赚到了第一笔钱。那时我还是高中生，有人在做另一个学校的报告卡生成系统，把一部分转给了我，这里没有任何利益冲突。我使用过Fortran、IBM 1130汇编语言、PDP-10机器语言和APL。我想我不能说我正儿八经地使用过SNOBOL。当然还用过C、C++、Bliss，后者是从卡内基·梅隆大学流传出来的DEC系统实现语言。基于Red的GNAL，我曾相当认真地使用过。

几种Lisp的变种，包括Common Lisp、Scheme、Maclisp。Dick Gabriel和我用在S-1上的Lisp，S-1 Lisp，这也是最终合并到Common Lisp里的四五

种Lisp之一。我开发过Connection Machine Lisp，但是我说不好是否算认真地使用它写过代码，我想那是用于实现*Lisp的时候。*Lisp不应该和Connection Machine Lisp弄混，它们是两种截然不同的语言。

我很认真地用C*写过代码，这是另一种我们为Connection Machine开发的语言。当然还有Java，还有一些脚本语言，我曾经用JavaScript和Tcl做过很多工作。

我还曾经很认真地用Haskell编过程。这里“认真”的意思是我用这种语言工作过一个月以上，而且用它们写过一些大的代码。哦，FOCAL，一种DEC计算机上的早期交互语言，类似于……有点像BASIC，也有点像JOSS。我用BASIC也写过很大的代码，既然提到了它。还有TECO，文本编辑和修改器，当然是用于编写Emacs的第一个版本，在这种意义上我认为它也是一种编程语言。我写了大量的TECO代码。还有TeX，也可认为是一种编程语言。我想上面这些就是主要的吧。

Seibel：*刚才我问你：“你最喜欢的编程语言是什么？”现在我想这个问题的答案应该是“嗯”吧？*

Steele：我有三个孩子，你也可能会问哪一个才是我最喜欢的。他们都很棒——他们有不同的能力、不同的个性。

Seibel：*有哪些编程语言是你不喜欢的呢？*

Steele：我从每一种语言那里都得到了一些乐趣。当然有些语言我用起来有点挫折感。以前我享受TECO，但现在我不想回头再用。它有很多难点——比如一个月后你回头去阅读你写的代码时就很困难。

我不确定我写的Perl代码是否够多，可以正经地充当一个贬低者，但是我从没有被这门语言吸引。我也没有被C++吸引过，我也写过一些C++代码，任何我想选择用C++实现的东西我想现在我都可以用Java完成，而且更容易，除非效率是主要的考量。

但是我不想被看作是在诋毁Bjarne Stroustrup。他为自己设定了一个特别的目标，那就是让面向对象语言完全向后兼容C。这对他来说是一个艰巨的任务。在这个约束下，我想他提供了一个令人钦佩的设计，一切也做得很好。但是对于我在编程时追求的目标，我认为决定向后兼容C是一个致命的缺陷。这是不可能克服的困难。C基本上是一个损坏的类型系统，它可以很好地帮助你避免一些困难，但是它不是无懈可击的，你也不能依赖它。

Seibel：你认为语言变得更好了吗？你不停地在设计语言，因此但愿你认为这是一个有价值的追求。由于我们的进步现在写软件要更容易了吗？

Steele：可以这么说，现在编写像30年前那样的程序是要容易很多。不过我认为人们也变得雄心万丈，因此我以为和30年前比，现在的编程是一项更困难的活动。

Seibel：那么什么事情使它变得更困难了呢？

Steele：我想现在的人和30年前一样聪明，另外和30年前一样，他们也被推到了他们能力的极致 ——我选择30年前作为判断的基准是因为那时我刚刚毕业。但是困难在于——就像我前面强调过的——你不可能理解所有事情了，甚至连想都不要这么想。因此我认为今天的程序员要应付一个困难得多的环境——仍然是施展等量的才智，但是在一个更难理解的环境中。因此我们制造了更精巧的语言来帮助他们应对这些不确定的环境。

Seibel：你很有意思地提到了"更精巧的语言"。有一种思想流派——你肯定知道，被称为Sheme学派，他们认为，管理复杂性的唯一方法是保持事情简单，包括编程语言。

Steele：我想语言很重要的一点，是能够描述程序员想告诉计算机什么，想要记录什么，想考虑什么。关于要记录什么，不同的程序员有不同的风格和不同的想法。我认为要说明什么是要记录的，我们要先说一大堆数据结构、一大堆不变量。我们在Javadoc描述的东西也是应该要告诉编译器的。如果说有什么东西值得告诉其他程序员，那它也值得告诉编译器，我想。

Seibel：难道大部分的Javadoc内容，除了那些人可以阅读的部分，不都是从代码中提取出来的吗？

Steele：有些是，但也有些不是。参数间的关系很难在Java代码中描述。比如，这是个数组，这是个整数，这个整数应该是这个数组的索引。这种事情你不太容易用Java描述。但这是很重要的概念，在Fortress里你就能描述。

Seibel：那它们是编译成运行时断言还是静态检查呢？

Steele：两种都可以。具体到Fortress我们试图能描述这种关系。刚才我们谈过了代数关系，一些操作是可结合的。我们希望在Fortress里能够把这些说得很清楚。我不会期望每一个应用程序员会停下来想："你知道，我刚写的这个子程序是可结合的。"

但是库函数程序员真的得对此格外关心。或许因为如果他们打算使用算法的智能实现，那么这些性质对于算法的正确性来说至关重要。因此在这些至关重要的性质上面，我们要用一种编译器也能理解的方式来讨论它们。为了在语言中描述编程的重要性质，我猜想这是一条重要途径。

Seibel：*在争取不犯错误方面，语言的角色是什么？有些人会说："如果把语言限制得足够好，就不太可能写出差的代码。"另一些人则会说："别折腾了。这个做法必定要失败，我们不如对所有事情都保持开放，让程序员作些漂亮的事情吧。"你如何平衡它们？*

Steele：重要的是意识到你所做的是一种权衡。你不可能指望完全杜绝差的代码。你能期望的是用类似问"妈妈，我可以吗？"这样的代码避免一些错误。为了做一些困难的事情，你需要多点努力这样说："是的，这正是我的意思。"或者你可以故意让一些事情很困难或者不太可能，比如，破坏类型系统。有消有长——很难用完全类型安全的语言编写裸设备驱动程序，只是因为这种层次的抽象对于裸设备来说是错误的。或者你可以尝试加入这样的东西，你可以说："这个变量就是位于绝对地址XXXX的设备寄存器。"这本身就是一种不安全的特性。

Seibel：*有什么新语言带来了什么有趣的亮点吗？*

Steele：Python在组织方式上做得很好。Guido一开始时就决定不使用垃圾收集器，我不同意他的做法。我想他后来已经明确否定——我早就说过他们最终可能还是需要一个的。Python做了一些很有意思的语法上的选择，包括使用缩进，在某些语句后使用分号，这些都很漂亮。Python支持对象和闭包的方式也很有意思。

Seibel：*大多数搞Lisp的人会认为那种形式的闭包是有些不足的，lambda也很受限制。*

Steele：没错。你要知道，他是基于实现性和可解释性等做出的一系列的妥协。这些选择有趣，但不是我希望的，但是他服务于一个特定的用户社团，试图解决一些特定的问题，我很理解他做出这些选择的方式。Haskell是一门美丽的语言，我喜欢Haskell，不过我用得并不多。

Seibel：*你正在设计一门语言，并且你喜欢Haskell，但是为什么Fortress不是一门纯函数式语言？*

Steele：一方面，Haskell发现了monads，还拖进了I/O monad，现在则是事务

内存monad。理论上说它还是函数式的，但是这不会给你额外的帮助。另一方面，Haskell越来越像命令式。我不禁想起《镜中奇缘》[①]里的白骑士——“我曾经想过这样，把一个人的胡子染绿，并且一直使用一把巨大的扇子使他们看不见。”在我看来，monads在某种程度上就像那把扇子，你拖进I/O却试图把它再次隐藏起来——副作用真的有吗，还是真的没有呢？

尽管我还是每个月会这么说一次，但我觉得在设计Fortress的时候，我们应该从Haskell开始，然后逐步向Fortran和Java靠近，而不是从Fortran和Java开始，然后向Haskell靠近。在尝试创建更有效的并行数据结构而碰到困难时，我们发现自己正在采用越来越像函数式的方法来设计Fortress库。

Seibel：*显然你写作了很多东西，也很在意写作的技巧。你发现写散文和写代码是类似的脑力练习吗？*

Steele：呃，我觉得这二者很不一样，大多数英文散文的读者有不一样的处理器。因此，比如我就不能用几乎一样的方式使用递归，对有经验的读者我可以使用一点点。不过读者如何处理并理解文本总是知道的。

在写作时我比较担心的是英语表达的模糊性，而在使用计算机时我就不太担心。我经常担心的是读者可能会误解我写的东西。因此我花费了很多时间有意识地锤炼自己的行文风格，尽量减少误解。

我喜欢看《周末夜现场》——那个闹哄哄的场面和疯狂的嘉宾，但我喜欢它的一个幽默短剧，Ed Asner在其中扮演了一个核电厂的经理，计划外出休假两周。他走出门，回头说道：“再见了，伙计们，我要出发了。记住，你们不可以向核反应堆里添加太多冷却剂。”而其他人在接下来的三分钟之内一直在讨论他到底是什么意思。

Seibel：*因此当你用英文写作的时候，你显然是写给人类读者的，不过似乎你把它和为计算机写软件对立起来。但是很多人——比如Knuth——的一个很重要的观点是，当你写代码的时候，你同样是写给人类读者看的。*

Steele：是的，是这样的。

Seibel：*那么为人类读者写英文文章的经验对你编写代码有帮助吗？*

Steele：那是肯定的。当我写代码时，我脑子中想的最重要的事是，计算机会按照我的想法那样去工作吗？这也就是说：“我所做的它可以用哪怕只有

① 又译作《爱丽丝镜中世界奇遇记》，是英国作家/数学家刘易斯·卡洛尔的童话《爱丽丝漫游奇境证》的续篇。——编者注

一种方式去理解吗？”而不是完全不理解。接下来一个问题就是通常是有多种方式正确编码，那时我就会考虑人类读者，而且也考虑效率问题。

这通常需要权衡。如果效率最重要，我通常采用一个小花招。然后我意识到这会让人误解，就不得不加些注释来标示它，让它变得更可读一点。不过，经常是变量名的选择、代码的格式等，要多考虑人类读者，你要考虑怎样安排代码格式的种种细节为人类读者提供必要的信号，又对计算机没有影响。

Seibel：*随着我们的语言越来越好，或者说至少对程序员更加友好，和那个在打孔卡上写汇编语言的旧社会相比，似乎更容易编写出正确的程序了——你从编译器那里得到很多帮助，它能为你标出错误。是否允许把可读性放在第一位呢，比正确性稍前那么一点点？毕竟，Haskell的拥趸喜欢说：“如果你的Haskell程序通过类型检查了，它就不会出错。”*

Steele：我认为这是个很糟糕的陷阱。一个可以编译的程序可能会有太多错误，时时刻刻你都要关注正确性。否则你不仅会误导了计算机，也会误导了你的人类读者。

我确信编程是一项高度不自然的活动，需要很认真地学习。人们习惯于由听众自己根据理解去弥合差距。我假定我们可以依靠编译器做这些小事——“我需要一个名为foo的变量”，而你不必关心具体是什么寄存器，等等。但是我认为大多数人还不习惯在交流中异常准确和严格。当我们描述将要执行的程序的时候，小细节也很重要，因为一个小细节的变化可能会影响最终的输出。

我认为人们习惯于以一种有限的方式使用递归——我想Noam Chomsky[①]证明了这一点。但是实践中很少有人走到三层深——通常还是以一种尾递归的方式。理解递归的训练真的是很难学习的艺术。一旦你掌握了，并且把你的头脑用递归武装起来，它就变成我们编程工具库里一件最强大的工具。因此我确实认为你不能也不敢把你的眼球从正确性上移开。

Seibel：*然而有很多人出这样的语言或是编程系统，允许“非程序员”编程。我想你会认为这是一项注定要失败的尝试——编程的问题不在于我们没有发现正确的语法，而在于人们必须学习这种不自然的活动。*

Steele：是的。我想另一个问题是人们喜欢关注脑海中的主要问题，不太关

① 乔姆斯基（1928— ），MIT的著名语言学家。——编者注

心边界情况，或是混乱的情况，或是不太可能发生的情况。然而正是这些情况别人很可能不同意你的做法是正确的。

有时我问学生："这种情况下会发生什么？"他回答："显然是这样的。"马上就有其他人跳出来说："不不，应该是这样的。"正是这些情况你需要在编程规范里确定下来一个流程。

我想我们通常用魔法来比喻编程，这并不是偶然的。我们说到计算魔法师，我们认为事情是由魔法产生或者自动发生的。我想那是因为让机器做你想要的事情最堪比作我们在技术领域使少年的梦想成真了。

都读过童话故事，童话里的人会希望有这样的能力：在脑海中想象一下，挥动双手，然后就得到了。当然童话故事中也有人忘记了边界情况，结果坏的事情就会发生。

Seibel：*比如无端幻想再递归推导的风险。*

Steele：幻想和递归，是的。或者说："我希望我是这个国家最富有的人。"——那么，也可以通过使所有其他人赤贫，而你和以前没两样来做到。童话故事里有这种故事，是因为人们忘记了做事不止一种方法。如果你只考虑自己的主要愿望而不考虑细节，会留下很多漏洞。

Seibel：*从童话故事里得到的教训是，要想成为这个世界上的甘道夫[①]必须通过辛苦劳动，学习咒语才能达到，而且其中没有捷径？*

Steele：是的，我给你另一个实例——假设我要告诉我聪明的电脑："好吧，我得到了这个地址簿，我希望它永远保持有序。"然后它除了保留第一个条目让它会扔掉所有其他条目。现在这个地址簿是被整理过了，但这显然不是你想要的。可见要说明这样一件简单的事，即需要"一个已经排序的列表，并且没有丢失任何数据，也没有任何条目重复"，写出来也是相当复杂的。

Seibel：*那么有什么语言特性使程序员——那些已经掌握了这种不自然行为的人——更有效率？你现在正在设计语言，因此你肯定会有自己的看法。*

Steele：刚才我说过人们绝不要忽视正确性。另一方面，我想我们可以设计工具来更容易达到正确性。我们不能做到轻而易举，但是我们可以让人们更容易避免几类错误。一个很好的例子是对算法做溢出检测，或者指定大数而不只是看着32位整数溢出。现在实现这些代价更高，但是我相信提供功能齐备的大数对某些编程来说，是一种更少出错的做法。

---

① 甘道夫是《魔戒》中的一个神通广大的巫师。——编者注

我发现系统程序员和操作系统算法设计者经常会掉入这样的陷阱："好吧，我们有些阶段需要同步，因此我们可以采用取一个数的策略。每次进入一个新的运行阶段，我们就给某个变量增1，得到一个新的数，那么不同的参与者在特定操作发生前，就会确定他们工作在相同的阶段上面。"在实际应用中这工作得很好，但是如果你采用32位整数技术，那么它不一会儿就会计数到40亿那么大。如果整数溢出了会发生什么呢？还能置之不理吗？不少文献中的很多算法都有这样的隐藏bug。如果有些线程在第2次～32次迭代中处于停屯会发生什么呢？在实际应用中这是不太可能发生的，但还是有这种可能性。要么让正确性问题不让人担心，要么通过计算显示说，是的，这不太可能发生，我可以不用关心这个问题了。或者可能你愿意每天处理一个小故障。重点在于你做过了分析而不是笼统地忽略这个问题。实际上，那个计数溢出是一个很隐蔽的陷阱，它不会伤害大多数程序员，但是会在某些人的算法里布下陷阱。

Seibel：*说到故障，你不得不查找的、最糟糕的bug是什么？*

Steele：我不确定我是否能搞定最糟糕的bug，不过我还是可以讲几个故事。当然，与并行进程打交道会产生最难解决的bug。

当我还是少年时，在IBM 1130上编程，有一次在梦中解决了一个bug也许是在刚醒来的时候，还真是一辈子少有的事情。我被这个bug困扰了好几天，始终不能解决它。某个夜晚我突然从床上坐起来，意识到问题出在哪儿。那是因为我忽略了接口规范里的一些东西。

它与并发进程有关。我编写了一个反汇编器，这样我就可以通过反汇编来学习IBM磁盘操作系统。它会从磁盘上拿出二进制数据，用不同的格式把它打印出来，包括指令、字符形式的代码、数字，等等。为了转换字符，我把数据喂给几个字符转换处理例程，其中一个要在从读卡器读入卡代码之后使用。我忽略了在规范中的一条小小的脚注："在调用这个过程之前，用于读入卡数据的缓存，其低序位均应已经清空。"或者也许是把它们置位了。

不管怎么说，从卡上代码栏过来的12位将会成为16位字的高12位，而低位在这里则被用于耍了一个聪明的小花招。你可以异步地调用读卡过程，异步地装载缓存，后面执行转换过程用低位去判断下一个卡栏有没有读过。如果已经读过了就开始转换。因此，一旦开始读卡，很快这个会话过程就会结束——卡转换的时间和读卡的时间重叠在一起了。不过 我使用的是原始的二进制数据，并没有遵循上面的约定。我只是忽略掉了这一点——我认为这

不过是另一个卡转换过程，但是结果它的接口有点特殊——它依赖于低位，通常你不会想到。在解释缓存中的内容时会说："哦，数据还没有从读卡器过来呢。"从道理上我知道这一点，不过我当时没想起来。然后，就像我说的，在我睡觉的时候想起它了。这真是非常离奇。

我可以想起来的另一个有趣的故事是，当我作为Maclisp系统的维护者的时候，Maclisp支持大数——任意精度的整数。使用了很多年，被认为经过了很好的调试和修正，Macsyma中到处在用，其用户也无时不刻都在使用它们。然而Bill Gosper的报告却说："这两个整数的商是错误的。"他知道这个错误是因为这个商值应该很接近于π的十进制倍数。

这两个数每个都有上百位长，由于除法过程相当复杂，而且这些是大数，手工追踪显然很不可行。我紧盯着代码但是没有看到明显的错误。不过有件事引起了我的注意，有个条件分支我不是很理解。

这些过程基于Knuth的算法，所以我把Knuth的书从架子上拿下来开始阅读说明，把Knuth算法的步骤处理成对应的汇编代码。Knuth算法里的书一条注释说这一步很少发生——概率大约是2的字长幂分之一。因此我们断定每四十亿次才会发生一次。

从这里我开始推理："这些过程被认为经过了很好的敲打，这一定是一个很罕见的bug，因此问题很可能发生在几乎不执行的代码中间。"这些信息已经足够让我关注那些代码，后来发现是一个数据结构没有被正确复制。结果在这一行中引起了副作用，在事情变得糟糕的地方，产生了严重的问题。然后我修复了它，传入数据，得到了正确的结果，Gosper似乎对此很满意。

不过一周后，他又带着两个更大的数回来了，说："它们的除数也不对。"这一次，由于已经有充分的准备，我又回到了那一小段大约十条指令代码的地方，再次发现了类似的第二个bug。因此我通览了代码，以确定所有东西都被正确复制了，自那以后就再也没有错误报上来了。

Seibel：*总是这样的——同样的错误绝不止一个。*

Steele：我想可以有这样几个教训——我应该知道有不止一个bug，在第一次我应该检查得更仔细一点。第二个教训是，如果一个bug很罕见，检查很少被执行的路径应该会很有成果。第三个教训是，应该对算法要做什么有一份好的文档，比如去查阅Knuth的书，是非常管用的。

Seibel：*除了你在半夜里醒来意识到问题出在哪儿的之外，碰到问题时你偏好的调试技巧是什么？你使用符号调试器、打印语句、断言还是形式证明，还是所有这些？*

Steele：我承认我有点懒——首先我会尝试加入打印语句，看看是否管用，当然我知道对复杂的bug而言，这可能是最没有效率的做法。不过要纠出简单bug，打印语句做得很好，所以还是值得一试。另一方面，我对编程认识的提高的一个极好契机是，我从事了一个Haskell项目。因为它是一门纯函数式语言，我不能随便加入打印语句。

这促使我100%地强化单元测试。我为每一个子程序都创建了单元测试。这最终变成了一种很好的习惯。

这也影响了Fortress的设计，Fortress试图包含那些鼓励创建单元测试的特性。把它们记录在程序文件中，而不是单独的文件中。我们在一定程度上借鉴了Eiffel的按合约设计的想法，也就是可以在过程中放入前置断言和后置断言。在你申明测试数据和单元测试过程的地方，当你请求执行的时候，测试框架会接管运行。

Seibel：*既然你刚提到了按合约设计，你在自己的代码里使用断言吗？*

Steele：我倾向加入断言，特别是在过程的开始处和一些重要点上面。当我试图——这里用“证明”也许过重——试图验证一些代码的正确性，我通常考虑不变量，然后证明不变量没有变。我认为这是一条很有成果的考虑方法。

Seibel：*在调试器中单步调试呢？所有其他的方法都不管用的时候你是否会这么做？*

Steele：这取决于程序的长度，当然有工具可以帮助你跳过那些不需要单步调试的地方，因为你确信那些地方是没有问题的。当然Common Lisp有个很棒的STEP函数，非常有用，我曾经单步调试过大量的Common Lisp代码。能够跳过特定的、你了解底细的子过程，当然会省你很多力气。能够设置俘获条件也很有用，比如说“等这个循环执行到第17次，我再来关注它”。对于PDP-10还有硬件工具的调试支持，这也很棒，至少在MIT是如此，那些日子里他们往往要修改他们的机器，增加特性。说起用多种方法观察真正的代码执行过程，真是三言两语说不完。

Seibel：*你尝试过形式化证明你代码的正确性吗？*

Steele：这取决于代码。如果我写的代码里面包含某种复杂的数学不变量，我会去证明。我不能想象在没有设置一些不变量并证明它之前就去编写一个排序过程。

Seibel：*Peter van der Linden在他的《C专家编程》那本书里，关于证明有一*

章是持否定态度的，其中他举了一个证明什么东西的例子，但是，哈哈，这个证明本身就有一个bug。

Steele：是的，的确，证明也会有bug。

Seibel：比起要证明的代码，至少它们出现bug的可能性要小一点？

Steele：我认为是这样，因为你采用的是不同的方法，使用了不同的工具。使用证明的理由和使用数据类型的理由一样，或者说和登山者使用登山索的理由一样。如果说什么都没问题，你不需要它们。如果真的出现问题了，这些工具方法可以增加抓住问题暴露的机会。

Seibel：我猜真正糟糕的状况是你的程序中有bug，然后在你的证明里出现了一个抵消它的bug。希望这不是个常见的状况。

Steele：这会发生的，甚至我都不也说这不是个常见的状况，因为你自然会构建符合你代码结构的证明。或者相反地，如果你脑子中的证明和你写代码时的一样，这自然会引导你的程序结构。所以以概率论的看法，你真的不能说代码和证明完全独立。不过你可以用不同的工具、不同的思考模式来克服这个问题。

特别是你在编程细节上倾向于采取局部的观点，而确定不变量时倾向于采取全局的观点，做证明就是要让这两件事情互动起来。你检查程序的每一步是怎么影响你努力维护的全局不变量的。

我职业生涯当中最有趣的一次经历是一次有人要求我去评审一篇David Gries投给*CACM*的论文，是关于证明垃圾收集器算法的正确性的，一个并行的垃圾收集器。Susan Owicki是Gries的学生，她开发一些用于证明并行程序正确性的工具，Gries决定应用这些技术来证明一种由Dijkstra提出来的并行垃圾收集器。全部的代码我记得是半页纸，论文剩下的部分都是正确性证明。

我开始考虑证明的事，试图由我自己去验证每一步。这里很复杂的是，因为是并行程序，实际上程序里的每一个语句都有潜在的可能去破坏不变量。因此Owicki的技术涉及在所有点上的交叉检查。我花了大约25个小时仔细检查它们，在这个过程中我发现了有几步不能正确完成。因此我报告了这些问题，这意味着那个算法确实存在bug。

Seibel：所以说尽管证明的结果是证毕，表明结果成立，但是算法中的bug在证明中还是被忽略掉了。

Steele：是的，证明是一个有错误的证明。因为忽视了某些地方。是公式

变换的细节——公式几乎是对的，但不是完全对的。好像是要改变一下顺序——两条语句的顺序等。

Seibel：因此你花了25个小时去分析这个证明。如果你只有代码，你能在25个小时内发现代码的错误吗？

Steele：我甚至怀疑我是否能意识到有bug。那个算法相当复杂，可能我只是盯着代码："是的，看起来有道理。"而不会发现其中非常隐晦的相互作用。那是一个十分必要的多步序列，不太可能出现交互。

Seibel：证明的过程基本上抽象化了这种相互作用，于是你不必自己提出这样的场景，假如这个发生了会如何，然后是这个、这个，最后才意识到出了问题。

Steele：确实如此。实际上证明采取了一种全局的观点，并且包含了所有的可能性，把这些总结成非常深奥的公式，然后你就不得不去做公式推导。因此作者重新提交了论文，虽然我完成了整个过程，它还是发回来做重新评审。我又花了25个小时重新验证证明。这一次似乎是没问题了。

我提交了结果，文章发表了，从那以后没有人再发现bug了。但是真的就没有bug了吗？我不知道，不过我想，我已经推导了那个证明，因此有信心认为算法现在是没有问题的。我希望我不是唯一的真正完成了全部证明推导的评审人。

Seibel：Dijkstra有句名言是，通过测试你不能证明一个程序是没有bug的，你只能证明你不能发现任何bug。对证明来说也一样——你不能证明一个程序是没有bug的，你只能证明，就你自己的理解而言，它没有推导出任何bug。

Steele：是这样的。这也是为什么存在机械证明验证这样一个特别专门技术的原因。希望在于把问题归结为去证明那个证明验证器是正确的。也就是——如果你可以写个足够小的。验证器——实际上验证一个大得多的程序的证明是个容易得多的问题。

Seibel：然后用这个手工证明的机械验证器去做你花了25个小时验证的其他代码的证明？

Steele：是的，非常正确。

Seibel：还有什么事情你愿意谈谈吗？

Steele：那好吧，我们还没有怎么谈论程序中的美感，我不想不说一点什么

就这样算了。我读过一些程序确实打动我了，因为它们蕴含着美感。TeX是一个例子，TeX的源代码。METAFONT要稍差一点，我不知道是因为我较少使用它，这是因为代码的组织或程序的设计与我喜欢的样子有些很微妙的区别。我真的不能判断。

一些算法也因其精彩性打动了我。我见过一些短小的程序，真是代码压缩的奇迹。要知道在那些日子，这样的事情是很重要的——你只有1M内存，是使用40个字还是30个字真的是一个问题，大家想法设法地压缩程序的大小。Bill Gosper写过一些只有4行的程序，当你把一个放大器连接到一个处理数位的累加器的低位比特时，就会发现这些程序的作用令人叹为观止。

似乎看起来是我在浪费时间和精力，不过我个人职业生涯中最得意的时刻是，我发现了一个方法可以从Gosper写的11字的程序中移除一个字。它只需要额外的一点执行时间，大约是一个机器循环的几分之一，但是我真的发现了一种方法可以缩短他的代码一个字，这“仅仅”花了我20年的时间。

Seibel：*所以20年后你说：“嗨，Bill，看看吧？”*

Steele：我并没有花20年时间去做这件事，而是过了20年后我回过头来再一次检查代码，突然就灵光闪现：我意识到通过改变一个操作码，它将变成一个非常接近我想要的结果的浮点常数，因此我既可以把这条指令当作指令，也可以把它当作一个浮点常数。

Seibel：*这就是“Mel的故事，一个真正的程序员”。*①

Steele：的确是的，就是这种事情。不过我其实没有想过在现实生活中这样做，不过那是唯一一次我成功地精简Gosper的代码。感觉上就像一次真正的胜利，那是一段美丽的代码，用来计算正弦和余弦的一个递归的子过程。

这是那时我们比较关心的一类问题。当我为IBM 1130编程时，有个启动卡片的概念，就是一大叠卡片中最前面的一张卡片。你按下计算机的启动按钮，然后硬件会自动地读第一张卡片，把其上的内容放入内存的前80个位置，然后从指定位置上开始执行。而那张卡片作用就在于让读卡程序来阅读其余卡片。这就是启动的过程。

在IBM 1130上这很困难的原因在于，卡片只有12行，而计算机是一个16位字的机器。因此12位会扩展成16位的指令，这就意味着一些指令不能在卡

① “梅尔的故事”由Ed Nather于1983年5月21日贴在USENET上，此后广为流传，是程序员圈中很流行的传奇故事。——编者注

片上表示。所以不能表示的指令就必须用其他能在卡片上表示的指令表示。因此这里你就有一个很困难的权衡——“我可以使用哪些指令，如果用了这条指令，我还需要其他几条卡片上的指令来表示它”——这会带来巨大的压力，你只有80个字来写出你的过程，因此你就倾向于这样的事情，重用指令为数据，用一段数据来做不止一件事情。如果你计划把这个子过程放入内存，那么它的地址也可以用作数据常数。这就是它看起来的样子——像日本折纸和俳句，所有这些都是一种编程风格。我在上面花了好几年。

Seibel：*在现在的环境下，你认为经历过这种训练的程序员好还是不好呢？*

Steele：他们得到了应付资源受限的经验以及精确估算的经验。

Seibel：*学着精确估算是件好事情。但是这样会让你养成模棱两可的、在今天已经不适用的编程习惯。*

Steele：很容易就走偏到优化上去，只是因为你能做到，即使你不需要这么做。确实有这样的问题，我很高兴我的儿子在高中时，有在TI计算器上编程的经验。因为那也是个内存极为受限的环境。因此他也必须学习如何用压缩形式去呈现数据，以符合计算器的环境。我不希望他在整个职业生涯中都按照这种方式去编程，但是我认为这是个很有用的经验。

Seibel：*回到代码的美感上来——那种俳句、日本折纸式的编程之所以美，是因为任何小巧的东西都是美的。*

Steele：是的，但是我要强调Gosper的代码之所以美不仅仅是因为你可以用那种方式去压缩代码——它可以这么短小的一个原因是因为它是基于一个美丽的数学公式、正弦函数的三角公式。并且递归在这种特定的结构中可以表达得很简洁，因为这个结构被设计成支持这种形式的递归，而同时代的其他机器却不能。因此是几种不同的美学融合到一起，组成了一个例程。

Seibel：*你也提到了Knuth的TeX，显然是个大程序，是什么让这个程序变得美感十足呢？*

Steele：他把一段有着很多特殊条件的非常复杂的程序归结成一个单一的、非常简单的范式：把许多盒子放在一起再黏合起来。这是一个极其重要的突破。不仅仅让文字排版变得灵活，而且让在一页纸上做的二维可视化摆入的事情都变得灵活。我希望更多的GUI接口展示按钮时也是类似地摆放盒子并胶粘式的。

Seibel：因此一旦你理解了盒子和胶水的所指，就会开始欣赏其中的美感了。你会说："是的，这是一个深邃的、正确的思想，我欣赏其中的美感，了解它将怎样作用于这个程序的外部。"你是通过一点点地阅读源代码，并且逐渐了解到主旨是如何表达的，逐渐从中得到——并且只能得到——更进一步的美学感受吗？还是说更有可能是在你读完了整个代码，末了方说："喔，所有的一切都是基于这个简洁但是不简单的主意呢。"？

Steele：两者的结合吧。Knuth很善于讲代码的故事。当你按照你的方式阅读《计算机程序设计艺术》，按照你的方式阅读算法，他把这个解释给你听，然后向你展示一些应用，然后给你些练习题去尝试，仿佛有人引导你经过了一次宝贵的旅程。一路上你看到了很有意思的风光。浏览TeX代码的时候我的感觉正是这样，我学会了编程的许多方面，其中有一些是平凡琐碎的，有些我就会说："喔，我没有想到那样去组织。"所以说两者兼而有之。

Seibel：从代码美感的相反方向看，软件里到处都有我们想要清除掉的、讨厌的历史遗留问题，比如不同的行结束习惯用法。

Steele：是的。在Common Lisp的标准委员会上我们花了很多时间讨论如何处理行结束，想要同时支持只使用新行的Unix和使用CRLF的PDP-10系统。要搞出一个新行的定义，让它适用于两种操作系统，真是场噩梦。

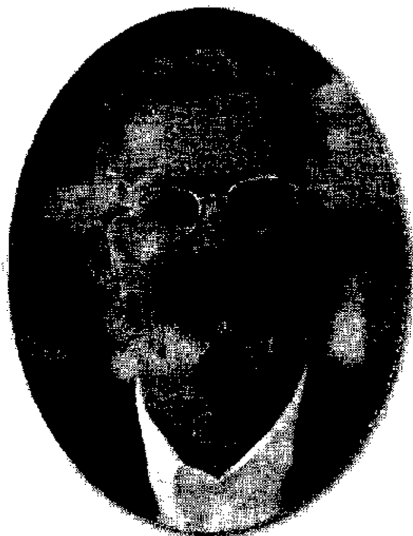
Seibel：对本书的读者或者是准备在将来去写点软件的人来说，有什么办法可以避免这种问题？我们可以采取什么聪明点的方法吗？还是说这就是设计演化的本质？

Steele：是的，我们都不知道将来。如果我可以改变一件事——听上去有点蠢——如果时光倒流到从前，如果我可以改变一件事，我想劝远古的人们不要使用手指头来数数。及早确立一个新标准，可以让现代生活变得更容易些。否则，你知道，在十进制和二进制不兼容的战斗中，我们吃够了苦头。

（编辑：陈兴璐）

10

Dan Ingalls



李琳骁 译

如果说Alan Kay是Smalltalk之父，那Dan Ingalls就是Smalltalk之母，因为Smalltalk或许肇始于Alan Kay尚未成形的想法，但却是Ingalls的辛勤劳动才将其带到这个世界。从Smalltalk第一版实现开始（该版本是在Kay一页备忘录的基础上用BASIC写就的），Ingalls一直参与七代Smalltalk的实现，从第一个原型版本，直到现在的开源实现Squeak。

Ingalls最初是个物理学家，一开始使用Fortran编程，开了家公司，推销他读研时开发的性能分析工具（profiler），最后去了施乐帕洛阿尔托研究中心（Xerox PARC），加入Kay的学习研究小组（Learning Research Group），该小组创建了Smalltalk，主要探索计算机在儿童教育中的应用。

在PARC期间，Ingalls还发明了用于位图的BitBlt运算，并通过PARC Alto计算机的微程序实现了该运算。它支持高性能位图，使得用户界面创新成为可能，比如我们今天习以为常的弹出菜单。（在Ingalls一次Smalltalk系统内部演示中，弹出菜单曾让本书下一章的采访对象L Peter Deutsch惊叫着跳起来，

“你们做的竟然和我想的一模一样啊？”）

目前，Ingalls是Sun公司[①]的杰出工程师，从事Lively Kernel项目研究。这是一个类似Smalltalk的编程环境，整个运行在浏览器里，使用JavaScript和浏览器提供的图形开发。Ingalls凭借在Smalltalk上的工作成就获得1984年ACM Grace Murray Hopper奖，1987年ACM Software System奖。2002年，他获颁Dr. Dobb's程序设计杰出奖。

在这次交谈中，我们讨论了交互式编程环境的重要性，为什么他没学过Lisp是件幸事，以及，相比构建静态系统而后试图添加动态特性，构建灵活的动态系统然后锁定下来为什么更好。

Seibel：*开门见山，你是怎么开始编程的？*

Ingalls：大致是这样的，我从小就在自家地下室捣鼓些小发明，能够想到和发明家关系最近的是物理，于是上大学时选择主修物理。我去了哈佛大学，期间上了门Fortran编程课。

Seibel：*讲讲那些年的经历？*

Ingalls：1962年到1966年，我一直在哈佛上学。我当时的两次编程经历其实就是那门Fortran课程，另外，有栋大楼的地下室里有个很棒的实验室，我在那里上了一门模拟计算机课。这门课会彻底改变你的思维方式，你手头只有一大块插接板，还有一堆电路元件，包括积分器、微分器等，你可以将这些电路元件连接在一起，实时解决各种各样的问题。不过我先接触的是Fortran，一开始就喜欢上了它。我总是设法一探究竟，看看自己能把程序写得多简短，看看自己能不能做其他类似的事情。

结果，我发现自己可能对电机工程更感兴趣。于是，我转读斯坦福大学电机工程学系，在那里上了一些计算机科学课程，欣喜不已。我在电机工程学上花的时间反倒没那么多。我还上了Don Knuth的课，他开了一门程序测量方面的研究生课程，我为之着迷。期间我开发了一个程序，能够分析其他程序，并在退学后以此为基础开了家公司。第二年我取得硕士学位，然后才从斯坦福退学。没记错的话，那是1968年。

Seibel：*这么说来，你是攻读博士学位中途退学的？*

---

① Sun Microsystems，Oracle已于2010年1月完成对Sun公司的收购。

Ingalls：没错，斯坦福大学无线电科学系，电机工程学系的一部分。

Seibel：*那么你藉以开展业务的是什么程序？*

Ingalls：嗯，那个程序始于我上的Knuth开的研究生课程。课程的主题是测量程序并观察其动态行为。

Seibel：*你指的是性能分析？*

Ingalls：对。有个程序会接管Fortran程序，在每个分支点插入计数器。我做了一个更好的版本，还有定时器中断处理，跟踪整个程序各个不同部分实际占用了多少时间。

Seibel：*基本上可以看作是采样性能分析工具？*

Ingalls：没错。值得注意的是，在那之前，性能分析通常是针对存储单元做的，但那需要量子力学才能搞懂分析结果的含义，而这个工具是针对源代码做的，结果一目了然，"哦，原来时间都耗在这儿。"对用户来说，这立马就能派上用场。我意识到，"哇哦！人们会用到这个工具。"

Seibel：*这么说来，你自己开公司运营了一段时间，直到去施乐PARC工作为止？*

Ingalls：差不多。实际上，那正是我最终去PARC的原因。我非常忙碌，大把时间都耗在当地的服务局[1]。当时CDC和IBM各有一家。另外我还会带着自己的程序四处奔走，确保程序能在这些客户特定的计算机上运行。

Seibel：*那个程序仍是测量Fortran代码的性能？*

Ingalls：是的。不过我发现一件很有意思的事。谁是Fortran的主要用户？他们都是大型科学计算用户。他们为谁工作呢？政府。他们会在乎程序的效率有多高？大都不会。他们真正想做的是要表明计算机负荷过高，他们需要更新的计算机，更多经费。我在几家公司展示了自己的程序，他们提议："哎呀，要是有针对COBOL的版本就好了。"

Seibel：*因为没人会给他们更多钱为COBOL购买大型机。*

Ingalls：完全正确。于是我写了一个COBOL版本的性能分析工具。那是我第一次使用COBOL。我记得当时写的完成例程会从定时器中断收集统计数据。完成例程需要使用待测量程序用的语言编写，这样两者才能加载在一起。

---

① service bureaus，主要为客户提供数据处理业务。

除我之外，可能再无第二个人用COBOL写过散列表。

总之，性能分析工具卖得很不错。我还记得几次业务拜访的经历。我会到客户那里，然后针对他们的程序跑一下性能分析工具，以演示效果，在演示过程中，我会向他们指出这套程序是多么物超所值，利用它省下来的钱完全超过购买它的成本。

在走访这些服务局的过程中，我在斯坦福工业园的CDC服务局忙碌了一阵，通常你得等到深更半夜才干活，那时花费会少一些，当时那里还有个家伙用Fortran程序做语音识别。他手里有各种各样的语音样本，他的程序会分析频谱，按音位等分组。我跟他搭话，问他："嘿，伙计，要不要用我的工具分析一下你的程序？"于是我们对程序做了性能分析，之后便分道扬镳了。

几个星期后，他突然打电话给我，"施乐雇我去做语音识别项目，我得找个人帮忙做具体实现，要不你来和我一起做？"于是我找他商量了一下。那人就是George White，长期做语音识别方面的研究。这就是我进施乐的来龙去脉，后来与Alan Kay共事则是因为我的办公位置和Alan的只有一厅之隔，我经常听到比语音识别更吸引自己的谈话。

Seibel：*是因为语音识别这个领域没那么吸引人，还是和其中涉及的编程有关？*

Ingalls：哦，语音识别很有趣，也很迷人。我在Sigma 3迷你计算机上逐步搭建了一个完整的个人计算环境。它使用卡片组，而Fortran是我开发用到的主要语言。我用这些搭建了交互式环境。我用Fortran写了个文本编辑器，以及另外一个程序，这样我们就可以从终端远程提交东西。它最终成了一个小巧的计算环境，只是实现的方式比较奇特。

Seibel：*在你的职业生涯中，对交互式环境的渴求是多次出现的主题。例如，你用BASIC写了第一版Smalltalk，因为当时那正是你手边的一种交互式环境。解决问题时，你会先想到要弄个交互式编程环境，这种想法是怎么形成的？*

Ingalls：好问题。我认为任何东西，只要你能从中获得即时满足，就会进入良性循环。

Seibel：*那么你最早是从哪里获得那种即时满足的？*

Ingalls：这种经历有过两次。一次是我有机会从事半交互式PL/I开发。另外一次，我有个朋友在IBM工作，他们有个交互式APL环境。我记不清哪个在

先。那个APL环境我记得很清楚。我在很多方面都受其影响，APL环境注重交互即时性（immediacy of interaction），也就是立即看到返回结果，以及表达式求值（expression evaluation），这和Fortran等面向语句的编程截然不同。

现在你仍能见到面向语句的编程，着实令人惊讶，整个C/C++/Java传统，尽管朝着面向对象的方向演变，但仍旧是面向语句的。不过，方便编写表达式的话，体验则会全然不同。对我而言，这会让数学变得更鲜活。总之这是两次体验中的一次。我到施乐工作时，除了那些Lisp程序员的环境，交互式的并不多。我刚好又没学Lisp。要是学了Lisp，想必一切都会大不一样。

Seibel：*为什么这么说？*

Ingalls：我觉得自己学了Lisp的话，就会完全朝着那个方向前进。正是因为没有选择那个方向，我才有机会以不同的方式做那类事情。我认为自己同Alan开发的东西和Lisp一样，拥有漂亮、生动、表达的感觉，只不过我们做的东西引入对象和消息等概念更为自然。

要是安于诸如Lisp之类的系统，我也许就不用自寻烦恼。我可能会尝试在已有系统的基础上想办法引入对象，但我认为一开始就从对象等概念着手构建，然后打造漂亮、交互式和方便的系统，才算得上是贡献。

Seibel：*Alan Kay说过Lisp和Smalltalk都存在同一个问题，就是它们太过出色，以致会吃掉自己的孩子。你要是学了Lisp，那Smalltalk可能就是第一个被吃掉的孩子。*

Ingalls：也许吧。

Seibel：*看来这是个活生生的例子，表明一定的无知也自有优势，它为创造力留了一些空间。但有时这行业让人觉得无知泛滥，人们不知道已有的东西，不断重蹈覆辙。*

Ingalls：没错。

Seibel：*我们应该设法修正吗？抑或这是我们必须付出的代价，这样人们才能保有创造力的空间？或者，要是人们多去了解其他领域的进展，我们的境况是否会更好？*

Ingalls：我推崇多样化。我认为，刚才讨论的例子实际上是提倡“不，放手让人去做，想怎么做就怎么做。”无知固然会造成浪费，不过孰优孰劣自有自然选择决定。这些不经意的芽变会将你带向未来。

我可以举出很多例子，那些领域设法标准化，试图全都朝一个方向前进，结果却压制了创新。我所在的公司靠Java支持，所以我不打算就此展开太多，但Java刚推出时情况就是这样，我采用的最简单的衡量方法就是观察OOPSLA[①]上发生的种种情况，总而言之，Java推出时，当运行其他面向对象的编程语言时，不但速度会慢下来，甚至还会停下来，甚至连动态编程语言也未躲过。我认为这是一大失败。

还好这种情况并没有一直持续下去。人们最终意识到："哦，等等，Java拥有这些强大的优势，所以我们使用Java做这做那，但我们本可以用动态编程语言得到其他好处，是时候回到动态编程语言上了。"不过这是个我容易发现的例子，因为我就身在其中。

从更大的意义上来说，我看不惯计算机科学系自认为其角色是帮助人们做好到该行业工作的准备，行业什么走向，我们就教学生选择那个方向。这么做完完全全错了。学校应当教导学生开拓思路，打破常规，规划我们应当追寻的其他航程。

这个问题并不简单，因为有这样一个完整的标准放在那儿，确实非常有用。数以千计的程序员已经开发出数以千计的例程，它们非常可靠，你可以利用它们完成工作。

服务于生产和服务于推动知识发展的计算机科学还是稍有不同的。Lively Kernel我一直做得很开心，从某种角度看，它微不足道。它其实没什么新东西，我用的都是之前就有的东西。Lively Kernel建立在JavaScript和浏览器支持的图形基础之上。不过它真的很有意思，因为这是另一个类似Squeak的核心。由于JavaScript和图形全都由浏览器支撑，因此这个核心的工作量很小。全部工作就是怎么设法显示图形，如何逐步搭建一个小型计算环境。

只要你做的东西像Lively Kernel那样，足够小，那么任何人都能理解。如果暂时撇开语言和图形等若干因素，那么问题就变成：核心是什么？我觉得这个问题很有意思。

我希望这个，并非特指我做的东西，我是说这类调查研究也许会重新启发计算机科学做一些相关研究，比如怎么制作核心，我们还能构建其他什么更简单更统一的核心。

---

① 全称Object-Oriented Programming, Systems, Languages & Applications，ACM计算机协会的一个年度会议，主要在美国举办。ECOOP是OOPSLA在欧洲的姐妹会议。OOPSLA的主题包括面向对象编程相关的系统、语言和应用。

这有点像数学。数学发现通过符号表示法人们可以简化许多东西。正因如此，你才能开始思考更大的概念。这就是我的希望。

Seibel：你提到核心时，讲的都是编程核心。Lively Kernel的核心是什么？

Ingalls：我所指的核心，通常来说，是你把足够多的东西组合在一起，以致在一定程度上，它能构建自身或构建其他有用的东西。Squeak就是真正能构建自身的核心。Lively Kernel假定已经存在JavaScript和若干图形，但最终它能够编辑图形，这样你就能制作新的图形元件，还能编辑程序，从而创建新的程序。总之，它足以构建你可能想要在浏览器里构建的所有应用。

我觉得在这场比赛中你必须能隐藏自己想隐藏的层。问题在于哪里才是你的比赛场地。在Squeak中，整个语言都是核心的一部分，因此它有自己的编译器和字节码解释器。这样，它才有自己完整的图形系统，拥有BitBlt及其周边的所有相关部件。

这些好像是任意核心的重要组成部分，不过你可以将它们抽取出来。你可以说："假定我们有一门动态语言，假定我们有图形。"按以前的想法，我可能会认为，"嗯，这下全都齐了，没别的东西。"但这么想并不对。剩下还有，怎样把图形组合在一起，创建一个吸引人的用户接口环境？怎么做才能把程序和脚本提升到你可以修改它们的层级？

我试图在浏览器里实现一个版本，这样不用安装就能运行，不过，只能是浏览器有什么我用什么。浏览器里有什么？好，我们有JavaScript，有图形环境。这是个机会，暂时放下这一切，然后说："嗯，是的，语言核心有了，图形核心也有了，之后就有了这另一种自主的用户界面环境核心。"

Seibel：在Lively Kernel和Squeak中，两者我都稍稍玩过一阵，这些核心除去语言或图形那部分，主要在于UI总是可编程的概念，它有这些小句柄等，这样你就可以用它来做编程处理。

Ingalls：是的。

Seibel：我发现这很让人困惑。我到底是在编程还是在使用这个应用程序？有时我真希望它们之间的区别大一些。

Ingalls：是的。这是另一把双刃剑。我觉得这没有简单直白的答案。从最底层来看，我们制造了完全支持变化的计算机，奇妙无比。全都是随机访问存储器，一切都是可编程的。对我而言，保持其活力、可锻性和可变性非常重要。如果你有个系统是动态和可变的，那么很容易你就可以划定界线圈一块

出来，限定“你不能改变这里面的东西”，相比之下，一开始从不是动态和可变的东西着手，然后想办法让它变得动态和可变，这要难得多。

不妨看看现在的Web编程，它一开始是以文本标记语言为基础，之后JavaScript才成为其中一部分，试图让它变得动态。要是一开始就以当时众所周知的动态图形为基础，然后再在需要时让东西固定下来并变成可打印的内容，一切都要容易得多。

Seibel：嗯，对所有人来说都更为容易，除了那些只想在Web上显示一些文本的人。

Ingalls：我想你说的没错。不过，对某些人来说，在那之上再加一层如HTML，也会更容易。我认为底层系统做得越动态越好。然后你可以增加语法或类型限定，或这或那，以及其他能将它固定下来的东西。肯定会存在这种情形，人们只是使用系统，因而不需要灵活可变的东西，就希望它们是固定下来的。的确，人们要是意识到其灵活性，就会被吓坏。就现状而言，Lively Kernel根本不是最终用户想要的东西。谁也不愿意看到自己的窗口突然倾斜20度。

Seibel：也不愿意检查他们准备点击的按钮的代码。

Ingalls：是的，没错。其实这只是个演示，试图给愿意选择那个方向的人一些启发。另外，它也非常简单，人们可以增加一层东西，让它可用，并且不以各种怪异的方式变化。但是，又想将东西做得灵活和通用，又想让它有已编好的程序，能像食谱一样，总是做到你所期望，两者间肯定会有权衡利弊的问题。

Seibel：你是否真的认为当前的Lively Kernel，或其短期演变，将成为人们构建应用的一种方式，或者这只是Sun Labs推出的思想实验，向人们展示一种思考方式？

Ingalls：嗯，这的确是个思想实验。不过它提供了几个最佳点，在某种意义上，这几点有可能真会转化为实际产品。它能快速实现一些功能，比如，如果你想制作一颗红心，在其中添加一条消息，并让它跳动起来，然后将其存为一个网页，你完全可以利用它完成这些工作，不用安装任何软件。也就是，登录Lively Kernel，在其中写几行脚本，构建这个会动的小玩意，然后通过WebDAV协议发布，创建和存储一个新网页。

这种做法既简单又管用，如果脚本也同样简单，就像eToys中的tile scripting，我觉得许多人都能从中找到乐趣。因此这有点像要花招。不过，

如果更进两层，你就会深入到真正有教育意义的东西，可以搭建与之交互的简单动态模型。它和Flash非常相像，但更简单，与编程结合更紧密。

从那个角度来说，我只是把它看作一个漂亮的环境，用来嵌入大量小巧动态、有教育意义的实例。大概一二十年前，有个软件叫HyperCard，许多教师都会这个软件，并用它做些有用的东西。但这些经验并没有很自然地被Web吸收，真的很奇怪。我觉得Lively Kernel仍有待完善的功能，可以用像HyperCard简单、如Web立即可用的工具加以补充。如果当时能朝那个方向前进，它肯定会异常出彩。

Seibel：*你一直参与了五代还是七代，总之很多个版本的Smalltalk实现，非常有名。我们先从你用BASIC写的第一版Smalltalk谈起。你拿到Alan Kay写的两页备忘录，必须据此实现Smalltalk。你都做了些什么？*

Ingalls：我直接开始写代码。我认为第一要务是验证操作模式。只需要几个基本结构，相当于栈帧结构。于是我采用BASIC的数组实现该功能，并将足够多的栈帧组合在一起，直至能执行一段代码为止。

一般来说，碰到这种事，映入脑海的第一个想法是做块"实验板"，你需要做的，只是将结构放在自己觉得想要解释的地方，然后设法让它工作起来。我记得我们准备运行的第一个计算是六阶乘。这其实是个很简单的例子，不过也涉及动态查找和新建栈帧等过程。一旦让它工作起来，你就会理解具体是怎么回事，并找出难点。

最后你算出时间都耗在什么地方，这样就可以完善各个部分。就这个例子而言，一旦运转起来，问题就来了，要在它上面添加一层，它本质上是个解析器，你可以键入文本，从而将它装入你实验的那个结构。接着，你就得到了一个小环境，开始学习各种东西。

然后你会说，"好，我知道这是怎么回事，我准备用汇编代码编写"，或者其他什么。突然间你会意识到："哦，对了，我们需要自动存储管理。那么我们该怎么实现？"问题接踵而至。

Seibel：*那么是否存在这样的情况，那种所谓的即时开发不起作用，或者你知道它不起作用，不得不采用某种不同的方式进行设计？*

Ingalls：嗯，你总是做自己能够做的，当遇到困难时，你总会转变方向，并进行反省。

在各种各样的实现人员当中，我通常就是努力把东西做出来，尽管这样做可能会犯错。很大程度上这是因为做出东西能让我激动不已，即使一开始

是错的，那也不要紧。关键在于，一旦它做出来了，就会开始告诉你那是什么。

然后你会发现，对了，也许本可以采用完全不同的方式实现存储管理，不过你学到的真正重要的东西又与之无关。我做的第一版Smalltalk垃圾回收采用引用计数；要是使用其他机制的话，效果可能会更好。有一段时间，引用计数相关的麻烦接二连三。但是这并无关大碍，关键在于系统起来了，运转起来，这样我们可以学习其他重要的东西，比如怎样用对象把各种东西组合在一起，以面向对象风格做数值计算又会怎样，以及其他所有实实在在的进展。

Seibel：*我不知道你在这群人里面如此特立独行，至少在本书目前访谈过的人中间是第一个。尽管在真正键入代码之前，Donald Knuth的确先用铅笔在记事本上花六个月写TeX，可他说这么做节省时间，因为不用费劲地针对自己开发的所有代码编写测试框架，他直接把整个程序写好了。*

Ingalls：我相信有这事。有些人的处理方式完全不同。但具体到特定某个人，我认为那么处理必有自己的理由。我知道自己或多或少浪费了些时间。不过这也有另一面，它可以说是探索性编程的典型表现，那就是如果它能更快将你带到可从中学到东西的环境，你也许会发现自己最初的部分目标根本无关紧要。更为重要的是还能有其他的收获。那会变成全新的焦点。

回过头来说这事，我需要好好想一想，有一两次我这么做过。我想到的例子是BitBlt。当我决定做这件事的时候，也就是后来的BitBlt，就有难题横在我面前，我不得不静下心坐下来，研究了一两个晚上。那个难题是，如何高效地搬移那些跨两个字的所有比特位？当时我还找不到什么有效的替代方案。于是我思前想后，终于想出一个简单的模型。那不是其他人提出的规格，不过我查看了我们写过的所有涉及画线、文本显示和滚屏的代码，因此对需要做什么我心里已经有谱。

Seibel：*或许你可以解释一下BitBlt旨在解决的基本问题。*

Ingalls：人们想把整个显示看作一个1000×1000像素的屏幕，但实际情况却是内存是逐字排列的，两者之间就不相通。如果你想把这四个比特位取出来放到另一个地方，结果它们可能会出现在目标字的另一个部分。实际上它们还可能横跨两个字。在屏幕上，如果你试图将某个东西搬移到另一个地方，有可能你得从源位置的两个字里分别取出比特位，再将它们放到目标位置。放好之后，你必须存储整个字。这样你就必须将那些比特位插入早已存在的

目标字中，然后还要将它罩起来，加以掩饰。总之一团乱麻。

还有就是屏幕光栅，屏幕一行一行地显示，形成了两个区域。源和目标的每扫描行字数可能不同，BitBlt就处理这种情况。

这是一个挑战，难处在于针对需要做什么已有明确的规范，而且你也想为此尽力做出个非常通用的核心，因为如果你把这做对了，它不仅能把东西从一部分移到另一部分，而且还能让你做重叠滚动。另外，它还支持像素混合。现在也正好是通用化的大好时机。

我做了测试，确保它先能在Smalltalk里运行，然后用汇编，最后再把它写成针对Alto机器的微程序。最后完成时，我们基本上做到了以内存的全速执行这些操作，讨厌的掩码和移位也不会造成什么延迟，因为这些都可以在内存时钟周期内搞定。

周围有支持微编程的计算机是个美妙的激励，因为很明显，如果有个小核心能满足需要，那么就可能付诸微代码，整个东西会跑得很快。所以我总是愿意主动这么做。

实际上，一想到这个，最先映入我脑海的不是别的，正是一幅图像，它就像一个车轮。不妨想象一下源、目的和字边界，它就像有一个轮子，从这里挑取完整的字，然后把它们放到另一个地方，其间只需一次移位，那就是映入我脑海的画面。余下的就是怎么用代码实现。

因此，BitBlt操作的中心本质上是个长移位器，从源地址选取字，然后把它们扔到目的地址。这就是我要坐下来思考的问题。不过，一旦可以这样存储常数，你就可以做文本的拆分，从字体里提取字形，并将它们放到任意像素位置。

Seibel：回到Smalltalk的BASIC实现版本。那是个比较原始的Smalltalk，甚至在Smalltalk-72之前？

Ingalls：是的。那个版本可用之后，我立即开始编写新版本，并实现了全部由汇编语言编写的版本，因为那台Nova机器就提供汇编支持，这个版本相当完整。我们就用这个版本调试大量东西，与此同时，Alto计算机的设计制造也在齐头并进。Alto一造好，我们立刻切换到那上面，开始在Alto上运行。这个版本后来成为Smalltalk-72。

Seibel：这么说来，Smalltalk-72是用汇编写的，那它什么时候变成自主执行(self-hosting)？我们经常听到Smalltalk的一大优势是它很大一部分都能由它自身实现。

Ingalls：那是很久之后的事情。Smalltalk-72还包含了一堆汇编代码。Smalltalk-76也不例外。从Smalltalk-72到Smalltalk-76的主要变化是我提出了Smalltalk的字节码引擎，拥有关键字语法，而且是可编译的。另外，类，甚至栈帧都成了真正的对象，正接近你所说的自主描述（self-description）。

Seibel：*你是怎么想到要写一个字节码解释器？*

Ingalls：那是种机制。我尽力解决的主要问题是Smalltalk-72即时解析，原因有二，至少，我们需要能够编译具备那几种语义的东西，但并不要求即时解析所有内容。

于是我提出了Smalltalk-76语法，与Smalltalk-80语法非常接近。紧随其后的问题是，你把那个编译到什么里头才能有效地运行？它带来的唯一麻烦就是，当执行我们所谓的远程求值（remote evaluation）时，你明明在这个地方声明变量，它却要到另一个地方才会求值。这最终形成了Smalltalk里的块（block），和其他系统里的闭包类似。

Seibel：*为什么不直接编译成机器码？*

Ingalls：我们还是非常在意空间占用，与其他实现方式相比，字节码解释器非常紧凑。而且也需要那么紧凑，因为我们仍然试图在内存只有96K的Alto上运行。后来他们推出了内存大一点的机器，有128K。总之，致密性很重要。

Seibel：*这是否意味着生成的代码会比较小，因为字节码比机器固有指令更丰富？*

Ingalls：是的。另外，我本人酷爱这个想法，主要受Peter Deutsch开发的Lisp字节码引擎的启发。此外更为这个想法的协同增效作用而激励，这是又一个适合做成微程序的核心。从一开始，我就预想它能写到Alto的微程序里。

Seibel：*微程序是用RAM实现的，因此可以将Smalltalk核心放在那里头，然后切换至Lisp，将Lisp字节码解释器放在里头。*

Ingalls：是的。

Seibel：*那么下一代是什么？*

Ingalls：Smalltalk-76继承了所有相同的图形包袱，针对画线、文本显示等的大量专门代码。不过那时我已经完成了BitBlt，于是我重写了这个核心，其中所有图形只用到BitBlt和Smalltalk，这样一来，整个核心要小许多。结果

就是Smalltalk-78，那是我们在微处理器上运行的第一个版本，当时是在8086上。

但那仍旧不是用Smalltalk实现的Smalltalk。用Smalltalk实现的Smalltalk直到Squeak才成为现实。Smalltalk-80有一套虚拟机规范，发表于蓝皮书[1]，不过所有实现都是用C或汇编代码写的。

Seibel：*那么编译器呢？*

Ingalls：编译器是用Smalltalk写的。实际上，在我们编撰Smalltalk-80系列图书期间，Dave Robson和我，主要是Dave的功劳，写了一个字节码解释器的Smalltalk仿真。作为Smalltalk-80发布的一部分，我们想帮助人们打造他们自己的虚拟机。我们发现，对人们用处最大的帮助，是当你首次启动系统时，能准确跟踪到哪些字节码以何种顺序被执行。

于是Dave用Smalltalk写了一个仿真器，那时我们的Smalltalk速度已经够快，这么做也水到渠成，它会生成能帮助人们调试的所有跟踪。

Seibel：*你们想帮助人们编写Smalltalk虚拟机，Smalltalk-80的出发点就是扮演救生舱的角色，这样即使PARC决定不再继续投入，Smalltalk也能对外发布，是这样吗？*

Ingalls：没错。随后，我离开了这一行业，回来时，我想用Smalltalk做个新项目。那时机器运行速度已经很快，"等等，为什么我们不试试运行虚拟机的Smalltalk实现版本，看看会怎样？"但令人高兴的是，将Smalltalk版本机械地转换成C实现不应该太难，这样它就会跟其他引擎一样快。如果打算修改虚拟机，你可以用Smalltalk来修改，也就是先用Smalltalk试验，然后按个按钮，它就会即刻转成解释器的产品版本。

Seibel：*于是你就得到这个用Smalltalk子集写就的Smalltalk解释器，而且也不管是什么子集，然后编写知道如何将那个子集编译成C的专门编译器？*

Ingalls：这个C翻译程序只是一个Smalltalk的子集编译器，只是我们必须用C打印输出分析树（parse tree）。实际上我们之前在施乐已经做过类似的东西，Ted Kaehler用Smalltalk写过虚拟内存，随后我们使用同一技巧将它转成BCPL。同一样东西。

Seibel：*Smalltalk-80对外发布时，已经出现了几家Smalltalk公司，对象正是*

① *Smalltalk-80: The Language and its Implementation*。

热门话题，Byte杂志做了一期Smalltalk专刊[1]。人们期望对象会成为可重用组件，程序员只需找家对象老铺，掏钱买些对象，把它们插入自己的程序即可。这一期望实现了吗?

Ingalls：我想既可以说实现也可以说没有。

Seibel：那么从哪方面可以说它实现了呢?

Ingalls：看看Java世界吧，它就是这么一个世界。因为有了那些共同的接口，几个庞大无比的软件就可以很好地协同工作。我认为这是一大真正的进步。由Smalltalk而生的几样东西多多少少影响了整个世界。其中一个是面向对象的设计和接口。另一个是动态语言和用户界面。但它并未占上风，你可以看看，在许多不起眼的历史节点上，有些事情如果以不同的方式来处理，可能会有更好的机会。不过我并不觉得这有什么大不了的。世界在缓慢地向前发展。总会有其他的事情做得很好。自然选择自会关照这一切。

Seibel：但是自然选择也可能选出一些非常怪诞的结果。

Ingalls：哦，是啊，比如Beta和VHS录像带格式战[2]就是如此。不过到最后，是金子总会发光的。

Seibel：Smalltalk的另一方面，也是Alan Kay近些年特别强调的，不在于什么对象，而在于消息传递（message passing）。C++和Java都没提供与Smalltalk做法接近的消息传递。这个概念为何如此重要?

Ingalls：因为它支持实打实的分离。Alan最近的言论，我觉得很对，一路下来它都应该像Internet。我们担心程序里哪些地方要设置完全措施和各种安全机制，以及随之而来的各种问题。但因特网式的分离才是不折不扣的真正的分层。

那么为什么消息传递如此重要?原因在于它能将内部和外部完全分离。至少，它做得对。另外有些系统在这方面走得更远，我认为我们会在这个领域看到更多进展。

Seibel：是金子总会发光的。有没有来自Smalltalk或其他地方的想法，是你希望能被主流世界采用的?

Ingalls：实际上，我对主流世界不抱什么希望，我有自己想做的或想让它变

---

① 参见http://c2.com/cgi/wiki?SmalltalkIssueOfByte。

② 参见http://zh.wikipedia.org/wiki/录像带格式战。

得容易的事情。我对计算机科学主流世界的一个希望是，人们能更注重智力空间里如何充分利用计算相关的基本原则。

对于我们熟悉的编程系统和语言，我们用得轻车熟路，非常拿手。要是我们同样善于使用逻辑编程，又会怎样？已经整合得很好了吗？我认为在以人为本的空间里，我们有更多事情可做。它确实朝着人工智能的方向前进。你必须知道，在某个时刻，我们终会跨过那道门槛，将来计算机会比我们更擅长思考。

有时我怀疑我们是不是有意无意地在拖延这个过程。到1980年为止，那个领域已经取得了许多进展。目前，计算机更快更强大，已经提升了多个数量级。在我用过的最新计算机上，如果我用Smalltalk运行Smalltalk音乐合成，它足以计算一个无线电台的无线电信号。对于曾经历过计算机只能求解简单算术的人而言，这真是难以想象。

于是你接受这一点，并以它作对比，在逻辑编程、基于规则的系统和人工智能中寻求各种可能性，你必须清楚这些领域有大量进展有待取得。我想看到那种导致Lively Kernel的思维，除了语言和用户界面之外，什么是核心？还有其他什么核心？如果你围绕逻辑编程建立一个核心，情况如何？你又可以做哪些事情？我认为人们对这块东西的捣鼓和修补还不够深入。因为，天哪，有了现在的机器，只要有一个小突破，你就能做出令人难以置信的事情。

Seibel：*Smalltalk原先被设想为一个用于教育的平台，对吗？*

Ingalls：它原本设想成孩子们的语言，按照Alan的原话是所有年龄段的孩子。在我看来，对整个项目而言，当然，它是个长期项目，有利的一点就是，我们并不是试图打造世上最好的编程环境。我们准备构建教育软件，因此更多的是关注简单和模拟现实世界。

正是心怀这个教育目标，这才不断激励我们保证较低层的东西尽可能地简单。因此，我们做的许多东西运行速度并不如你想象的那么快。第一个Smalltalk-72系统确实很慢，第二个修订版的运行速度大概快20到25倍。但我们让它跑起来了，谢天谢地，这样我们就能和孩子们一起使用Smalltalk，在尝试第二个版本之前就学到了很多东西。

我们将大量精力用于支持一些好看的图形、位图图形、音乐等，并用相当简单的语言组合起来。正是从这些学到的东西才真正帮我们成就了非常好的语言。因此在Smalltalk-72之后，我们开发了Smalltalk-76，实质上就是Smalltalk-80。这时我发现Smalltalk可以成为用于计算机行业的严肃的编程环

境。我和Alan之间有些分歧，因为他不想把精力分散在那个方向。

不久之后Alan离开施乐，于是我们开始沿着不同的道路探索。不过那是因为我们已经发现了某些东西。例如，系统中发生变化的周转时间很快就从开始的秒级提高到亚秒级。它完全是鲜活的。那是我和其他不少人热衷去做的。让我们构建一个系统，就像那样生动。这就是Smalltalk要做的。因此，这随即成为新的目标，由此催生了Smalltalk-80。Squeak是对这个的回归，不过还做到了自行搞定一切。

Seibel：于是，就像你说的，你和Kay走上了不同的道路。你是否对最初设想的Smalltalk不再抱有幻想？

Ingalls：不，绝对不能这么说。我前面谈到自己作为一个物理学家所接受的训练，我觉得自己的天性就是观察世界，提出相关问题，并实际接触这些现象，看看它是如何工作的，力是怎么回事，行星如何运动，风怎么吹等诸如此类的东西。在物理世界中，起码这么做轻而易举。你可以集中全部注意力来观察，最终掌握其工作机理。

我觉得在计算机中也是同样的道理。在计算环境中，你应该能够专注于音乐、音乐合成和声音，进而理解整个东西是怎么工作的。它应该是容易接近和理解的。图形也一样。它组合在一起的方式非常相近。你有一些原子的东西，包括各种图形特效，然后你拿到结构化的东西，并将它们组合在一起。数字计算的情况也一样。

给新人介绍Smalltalk时，我会问："你对什么感兴趣？拆散文本？捣鼓数字？欣赏图形？抑或玩音乐？"然后我们就挑选一个，进而深入其中。直到现在，我仍然热衷于此，我敢肯定Alan也一样，带领人们深入探究自己热衷的方向，这样他们才能最终得到Alan所谓的有巨大影响力的想法。这个伟大想法会带你发现，这个惊人的变化实际上就是工作中几个微不足道的、常见的东西。

在音乐里可以看到这点。在图形中也能看到。在数字和文本操作中都能看到这一点。对我来说，把这些做出来并容易使用，真是令人激动。

Squeak环境其实是面向计算机科学家的。eToys环境则是面向孩子的，但并不及想象的那般全面，我仍然觉得有一点我们还没做到，它本应该让你根据自己的直觉深入浅出，真正理解那些有巨大影响力的想法。

对此我仍充满热情。我现在为什么会做这个东西，也就是JavaScript和这个浏览器？原因在于，我们很快就能够把类似Squeak的材料放在网页上，

到时候你可以用任意浏览器浏览，并以某种漂亮、自显的方式进行交互。那只是整个构想的一部分。我敢肯定会有所改变。浏览器会改变。我们还会得到JavaScript之外的其他语言，我仍与Alan和他的小组保持着全方位的联系，他们正在做另一种尝试，更加深入，试图更为认真地解决其他一些事情。但毫无疑问，我们持有相同的愿景。

Seibel：*你提到了四个学科：音乐、图形、数学和文本。这些几乎与人类文明一样悠久。显然，影响巨大的思想独立于计算机而存在，计算机只是提供了一条途径，没有计算机，要探索这些思想可能会很困难。会不会计算机本身也有一些有趣且影响巨大的思想呢？编程或计算机科学是否是另一门深奥的学科，第五个领域，而且它是不是我们自打有了计算机后唯一能做的事情？*

Ingalls：是的，我觉得这正是我要说的意思。我一直设想这样一套课程，你从其中一门学科入手，也许会有动力深入其中一个领域，可随后又转到另一个不那么熟悉的领域，不过做的事情却类似。从中学到的经验是，你只要掌握实现那些更简单更深入的结构的办法，这些结构会生成那整个领域，而这个实现方式在所有情况下都是类似的。

这里有图形代数。它是原始对象，叠加，平移，旋转。或者音乐。它是音符、时间序列及和弦。这都是一回事。而我认为这会回过头去探究风怎么吹，行星如何运行。这是在邀请你走下去，找出事情工作方式，并学习构成这种代数的各种要素，即过程和原始的东西。所以，是的，如你所说，这第五个领域，正是所有这些事情的共通之处。

Seibel：*你会期望有人玩过上述三四个领域之后，最终来学习如何编码吗？或者说，那只是可能出现的结果之一，如果他们的兴趣刚好深入某个特定的领域，他们是不是有可能最终也来学习如何编程？*

Ingalls：我认为这只是可能发生的事情之一。希望这么做能锻炼他们的思维能力。一来通过向他们介绍这种东西，二来通过这样或那样的方式让他们感到兴奋。不过，有人会喜欢编程，也有人不会喜欢编程。比如说，我有个儿子12岁，他唯一想做的就是练习滑雪板的540度旋转动作，所以说，做任何事情都是有可能的。

Seibel：*再问一些细节问题：你怎么测试自己的软件？*

Ingalls：这取决于我在做什么。我总是先把软件搭起来，尽快获得即时满

足。所以，每当尝试做新东西时，我只是考虑哪部分最容易首先取得成功。每次都不一样。如果我有一个更正常的生活，身在更正常的开发团队，那我可能就会完全融入现在的团队编程方式。但对我来说，更多的是自我诱导，甚至到了我的注意力持续时间的问题上，也不例外。如果我觉得这个周末就能让它跑起来，那么这就是我选择做的那块，这就是我全力以赴的事情，其他所有事情一概忽略。这个很难概括，只是说，有一个目标你一直想实现，你选择了其中一块，这一部分会令你满意，也会表明你上路了，并且你能一直持续做这一块，直到你下次要暂停手头上的工作，在家或在单位做这做那的时候。

Seibel：*你选择做的那部分最后应该要取得令人满意的结果，那其实就是你的第一个测试，验收测试：它有没有在屏幕或其他上面画个窗口？那么细粒度测试呢？*

Ingalls：如果这是你第一次有可能挑选某样东西，把它拖过来，又放下，那么你需要有一个框架在那里工作。是不是一定要把框架这个因素包括进去，这样，跟随而来的他人才会意识到，这些就是测试？这事我通常不会做。这可能只是我这一代的奢侈享受，现在没人躲得了这些。但我是个老家伙，他们不打算逼着我去做。但我认为内在的感觉仍然是相同的。Squeak代码过去全都是可执行文件的注释和待检查的东西。例如，在大量BitBlt测试中，有这些琐碎的事情会从屏幕上一个地方挑选某个东西，对它做一些操作，然后把它放回去，如果你在屏幕上看到什么变化，就表明它工作异常，并附有对应的注释。这是个相当直接的测试。

Seibel：*我们接着谈谈与其他人的合作。PARC的学习研究小组好像组织非常紧密。你们是如何就代码本身开展合作的？*

Ingalls：就是通过保持密切联系。偶尔也会出现混乱。学习研究小组规模一直不大，我们每个人都有自己负责的领域。期间形成了大量团队编程技能，不过我并不精于此道。现在，在Lively Kernel项目中，核心部分一直只由我和另一个人Krzysztof Palacz负责。在某种程度上，我们都负责不同的模块。实际上，我们现在的确使用了代码库，团队其他成员主要开发应用部分，偶尔会做一点核心开发。我发现有个可用的共享代码库很好，非常棒。下一步是把Lively Kernel与仓库集成在一起，你可以修改Lively Kernel里的代码，但仅在当前运行版本中有效。它不会被推到仓库中，变成新版本的一部分。这是我们下一步需要做的。

Seibel：你做过结对编程吗？

Ingalls：我要想想看。通常我自己一个人搞定，或者独立做一部分。我也有许多项目是和其他人协作完成的，其中包括许多紧张的结对调试过程。

Seibel：有什么技巧能用来管理那类合作？当每个人都自顾自做自己那块工作时，各部分无法很好整合在一起的情况总是有可能出现的。

Ingalls：要么就某些接口达成一致，或者通常我会搭建一个还不完整的框架，只能跑一个例子，然后其他人就会明白他们的东西应该放在哪里。或者，他们这么做之后，我也很容易就知道自己那块该放在什么地方。基本上我们都采用这种具体的做法，而不是拘泥于什么规范，因为我们通常开发的东西都是没人写下来的。全都看当时的需要而定。

Seibel：你的工作涵盖了多个层级，从底层的BitBlt实现，编写微代码，上至相当上层的Smalltalk相关开发。程序员对于自己打交道的各级软件和硬件，到底需要了解多少？

Ingalls：这是个好问题。为了跳出固定思维，你自己也得稍稍跳出固定模式。如果有什么东西要被充分利用，但在平时语言使用过程中又没见过，那你必须对其有一定的理解，对它之外的东西也有一定直觉，并且有能力在控制这东西的系统里工作。

在语言设计方面，也许你准备打交道的处理器已经存在，也许你并不需要对他们了解很多，除非是要实现很好的性能，想知道缓存是怎么工作的，诸如此类的东西。我认为你必须后退一步，并自问："这要穿越哪些界限？"

Seibel：先不管最终需要知道多少东西，谈到学习编程，有些人认为应该从高级语言开始，学习某些普遍的概念。其他人则认为需从汇编入手，逐步往上学习高级语言，从而真正理解是怎么回事。你赞同这个说法吗？

Ingalls：不，我并不这么看。那是我学习编程的过程，并且我过去对它也很着迷。我认为总有人会迷恋这一层级或另一层级。但是我并不认为只有一种方式，就像从事艺术绝不是只有一条道。

我认为现在有其他同样激动人心且可能更合适的事情有待探索。天啊，四分之一个世纪之前，我们还在考虑怎么做人工智能。现在，机器运行速度已经不知道快了多少倍，但我们在那个领域几乎毫无建树，我们还在做和Fortran差不多的东西。Prolog出现时日已久，而且各种各样的问题都可以用逻辑编程解决。如果你应该了解汇编，并弄清楚它是怎么回事，那你应该跳

出固有框框用心钻研，做真正有可能属于未来的事情。

所以我不是说“不学汇编语言”。我想说的是，你应该学习若干其他强有力的技术，这样当你考虑如何继续前进时，就可以充分利用这些技术。至于从哪里开始，对我而言，即时满足一直起着非常大的作用。在教别人Smalltalk时，我通常会先进行简短的对话。“你对什么最感兴趣？你对怎么折腾文字感兴趣吗？对数字、音乐、图形等所能做的事情有兴趣吗？”我会从上述任意一个问题开始。

你可以做各种有趣的事情，比如把文字拆开，重新组合在一起。你可以对数字、不同的底数、浮点、定点做各种有趣的事情。对音乐也是一样，你可以从音符开始，将其组合成旋律和和弦。对图形则可以做叠加和旋转等操作。其中任何一项都值得好好探索一番。我真的认为不同的人对这种方式的看法也不尽相同。同样，如果你要教别人计算机编程，你也许对表达式求值在行，也许擅长逻辑编程。也许你从事用户界面相关开发。人们都会在一个领域内放射光彩，而那里就是他们应该深入的地方。

Seibel：*据我了解，Smalltalk最初的目的是要教一种编程基本技能。那是每个人都应该具备的吗，就像我们期望每个人都能阅读和书写，都会一点数学？是否每个人都应该具备一定的编程能力，只因为那是种有用的思考方式？*

Ingalls：我很难说谁应该做什么，因为我遇到过许多这方面或那方面都胜过我的人，但他们对编程一无所知。就基本技能而言，编程所包含的就是逻辑和数学，没错，人们应当学会逻辑思考。但我从不说别人应该知道如何编程，我不这么看。我们在日常生活中做的许多事情本身就与编程相仿。你需要了解过程的各个步骤，诸如此类的东西。

计算机融合了一些强大的想法，可以实现一些美妙的主意。计算机的强大之处在于它们给予数学以生命。因此它们在那方面可以用作一种强大的工具。现在，我觉得强大的想法是创造美好生活必不可少的，但不清楚有多少属于这块。

Seibel：*Seymour Papert在Mindstorms一书里写到，调试是智力工具包的重要组成部分，他认为问题的实质不是得到正确的答案，而是找出答案，然后调试。*

Ingalls：哦，当然！人们应该学会清晰地思考，学会提问。这对我来说是非常基本的。这与个人成长的家庭环境有莫大的关系。一种情况是，当橱柜门

关不紧时，有人会打开柜门，看看铰链有无问题，发现有螺丝松动，柜门这才一直晃悠。相比之下，另一种情况是，发现橱柜门关不紧，他们会说："哦，这门关不上了，找个人修一下。"这两者之间的差别显而易见。在我看来，撇开计算机，你平时就会有这种经验，你所看到的是不对的，你该怎么办？询问。观察。那么，如果你发现了问题，你又如何解决它？我认为这是非常基本的，也是人之常情，许多都是父母给孩子的影响。

计算机无疑是这么做的一种媒介。不过，计算机只是计算机。这种方式很大一部分是能传递的，但对我来说，这是真正重要、基础和人性的，因此我们不大可能只教他们计算机就能启迪整个世界。

Seibel：你还记得自己写过的第一个有趣的程序吗？

Ingalls：我得想想。在每次编程经历中，总有些东西会跳出既有框框。当我接触到VisiCalc时，我在VisiCalc里写了一个电子表格，把英语转换成儿童黑话[1]。那对我来说很有意思，因为它把电子表格隐喻用作一种并行编程方法。用那种方法解析文本既有趣，又予人启迪。

Seibel：这么说来VisiCalc直接提供拆散字符串的原语？

Ingalls：是的，你可以把字符串拆散。我用的可能是Lotus 1-2-3而非VisiCalc，我无法确定VisiCalc支持字符串原语。我拿到一台小巧的Poqet PC[2]，它是最早的手持式PC，名副其实。Poqet PC由两节AA电池供电，我在上面装了1-2-3，那是在一次横跨美国的飞行中，我想着这个时候自己能做点什么。

Seibel：这肯定是在你学会编程很久之后，因为你开始学习编程那会儿压根就没有Poqet PC。

Ingalls：对，那是后来的事。我用Fortran也做过非常有意思的事，有一次，我拿到Val Schorre关于META II的论文，META II是个奇妙的、非常简单的编译器的编译器，我用Fortran实现了一个版本。真是出人意料，这意味着在只有Fortran的环境中也可以有其他语言。这是我用Fortran做过的最有意思的事情，因为它是借Fortran之力逃离Fortran世界。

---

① 儿童黑话（Pig Latin）是一种英语语言游戏，形式是在英语上加上一点规则使发音改变。据说是由在德国的英国战俘发明来瞒混德军守卫的。儿童黑话多半被儿童用来瞒着大人秘密沟通，有时则只是说着好玩。——编者注

② Poqet PC是种非常小巧，可移动的IBM PC兼容机，由Poqet计算机公司于1989年推出，详见http://en.wikipedia.org/wiki/Poqet_PC。

Seibel：*这看似有点主题的味道：先是儿童黑话电子表，然后是刚才说的这个，以及用COBOL为性能分析器写的散列表。是不是你有时喜欢违反常规？*

Ingalls：我不觉得这是违反常规，但不论何时，只要有计算环境可以摆弄，我都喜欢在里头尝试新东西。那也是我开发Smalltalk系统时乐趣多多的原因所在。你基本上是从零开始，你的任务是确定怎么组装，以及先要搞定那些能帮你取得进展的东西，并以此为基础不断构建。

在这些情况中，重要的是跳出条条框框。只有跳出来，你才能确定自己已经掌握某种工具，是不是能用它做到一些你原本以为自己做不到的事情。

Seibel：*现在你对编程的看法有什么大的改变？*

Ingalls：这个问题问得好。其中一点是我们已经有那么多计算机时钟周期可用。因此我现在自在不已，正如人们常说的那句带贬义的话，浪费时钟周期吧，只要能把事情做得漂亮些。但对我来说，基本内容没什么改变，仍是设法弄清楚我必须处理的核心是什么，我要尽力实现的目标又是什么。

它也稍稍变了一些，因为我不再是自己工作小组里出力干活的。我更多是在更上层，那意味着我在目标和政治上花的时间比代码上的更多。我更多是去创造环境。在这之前，我很幸运，一直身处现成的环境中，不用我费心创造。但现在每隔一段时间，我还是会坐下来认真写代码。

Seibel：*我原来看你上世纪70年代写的一篇论文，讲的是你的Fortran性能分析器。在序言里，你非常热情地讲述了那项工具是如何改变自己的编程方式的。过去你是先确定自己打算写什么，然后编写代码，再调试代码，后来你就先确定自己打算写什么，再编写一个非常简单的版本，进行性能分析，然后优化代码。你现在仍按那种方式编程吗？*

Ingalls：我当然先要获得效果，看到屏幕显示些东西，不管具体是什么，因为这非常激励人心，找机会先一瞥进展如何，对于原本想做的东西，你往往会有新的体会。

然后，如果需要性能分析，那就照做。或者，结果发现可能你没做对，那不是你真正想做的东西，于是你修正目标，或者调整做法。不过，事关性能时，我仍会按照同一种方式操作。我们在Smalltalk里做了一个很棒的性能分析器，Squeak也是如此，它能给予很好的反馈。部分与性能相关，但部分只与结构和架构有关。你可能会发现有些东西极少用到，换种方法来做的话，也许你可以直接去掉它们。这只是对同一事物的不同视角。

Seibel：Knuth的《计算机程序设计艺术》几乎程序员人手一套。有的人买来只是搁在书架上，装点门面。有的人买来用作参考书。有的人买来则是一页一页用心研读。你曾在斯坦福大学和Knuth一起做研究，那套书你读过多少？

Ingalls：我很喜欢与Don共事，在斯坦福大学，他的MIX[①]课程我教了一个学季，对我也很有启发。我自觉与Don截然不同，他极富数学头脑，同时还热衷深入实现细节，事物的实用部分，这是他吸引我的地方。我也喜欢深入事物的实用部分，但没有Don那么严格。

我受过物理学的训练，对我来说，我处理的问题或处理问题的方式，实际上更多是客观存在的事情。当我谈及程序的其他观点时，我内心真正的看法是，这是件物体，你可以触摸并感受其振动。

你可以看他开发TeX的方式，就会发现它完全是跟数学有关的，而且非常优美和优雅。相比之下，比如最早的Smalltalk引擎，它们就显得非常临时。我只是把需要的合成一体。几个回合之后，我可能开始得到某种数学上的图景，或者我们确实得到了，但就这方面来说非常不同。

其实，他书中基本数据结构相关的内容，我读了不少，但我并不是特别喜欢读书，我更多是个实干家。我有不足的话，那就是我往往会以自己的方式来实现甲乙丙丁，而不是去阅读文献，了解这些东西。我觉得这么做通常对我自己很有利，但谁知道呢？

Seibel：你认为程序员需要掌握多少数学？Dijkstra声称计算机科学只是数学的分支。理解《计算机程序设计艺术》需要相当扎实的数学功底。

Ingalls：你必须具备逻辑头脑。不过我在学习计算机期间，在弗吉尼亚的乡间待了很长时间。我总是认为，如果想在弗吉尼亚的深山里开办一家计算机公司，我需要掌握技术细节。除了相当深奥的特定部分，数学不及逻辑和直觉那般重要。

我认为大部分更多是架构上的：图形与模型配合的方式，以及数据需要更新或缓存的方式。那不是深奥的数学。也就是说，我的确把计算机科学看作是数学领域的一部分。对我而言，真正激动人心的是，有了计算机，数学才可能变成一门综合性的，而非只是分析型的艺术。我每天乐此不疲，做的是有几分像数学的活儿，但它是创造性的、有产出的、综合性的。

Seibel：你提到自己不是特别喜欢阅读。不过有什么书可以推荐的吗？

---

① Donald Knuth那套书中的假想计算机，http://en.wikipedia.org/wiki/MIX。

Ingalls：没有。我相信自己这样的确属少数。我从小就不是特别喜欢读书。偶尔，我会阅读些资料，十分投入，不读完不罢休。有些论文绝对值得一读，我记得还有几本书也是。Val Schorre关于META II[1]的论文当属其中之一。还有那本《LISP 1.5程序员手册》。当然还有APL，但我并不认为Iverson的书是学习APL的良方。数学家则有可能会觉得是。我甚至记不清自己是从什么书里了解到APL的。不过我还是很喜欢APL。因此我觉得花点时间在那门语言上，就像是阅读一本书。Smalltalk也是如此。

Seibel：*那你是否仍一如既往，像最初那样享受编程呢？*

Ingalls：当然，仍享受编程本身。过去几年很有意思，因为我得脱离自己非常熟悉的环境，Smalltalk和随后的Squeak，那里的工具实在是棒极了。我必须稍稍后退几步，与浏览器里的JavaScript和传统开发环境为伍。有时我得比以往花更多时间去调试，但想出办法并实现的基本过程，我仍然很享受。

Seibel：*你觉不觉得编程是年轻人的游戏？*

Ingalls：不，不完全是。这确实需要能够全面研究正在进展的事物，也需要有无限精力，而我也不像从前那样有用不完的精力。但是，我仍喜欢接受难题，坐下来，认真钻研，直到搞定为止。正好这里有个类比：我开始尝试学弹钢琴时年纪已经不小。人们都说："哦，你应该年轻的时候学啊。年纪轻学起来快得多。"尽管我在这方面造诣还不是很深，不过也得出一个结论，并不是年轻人学起来快很多，只是因为他们有更多时间。只要我付出时间，我就能有所进步。

我感觉编程差不多也是同一回事。回顾过去，早些年，我的时间任由自己支配。我埋头工作再工作。而现在，生活中还有其他事要做，我还要负起其他职责，不再只有编程。因此全力投入编程越来越难。

Seibel：*撇开必须付出的时间多少不谈，编程不是还要求相当的注意力和专注？人们总在谈论流畅状态，总说你要是每15分钟被打断一次，你绝对干不成什么事，因为重新理清头绪就得用上15分钟。*

Ingalls：这倒让我想起自己还在PARC时对某人说过的话。那时，除了开发Smalltalk，我开始承担其他职责，不过，在将Smalltalk变成真正的生产系统方面，我们也在不断取得很大进展。我开玩笑说，我是跑着改进Smalltalk环

---

[1] 参见http://en.wikipedia.org/wiki/META_II。

境，这样我才能在越来越短的时间仍然完成适当工作。于是，我会找个自己能待上15分钟的地方，坐下来真正做点有用的东西。

另外一点是你要与其他人共事。我和较年轻的人一同工作，感觉很棒，好像我会多花些时间思考目标、政治层面的东西，安排规划，他们则弥补我的不足，可以钻研得很深很深，而我没那么多时间这么做。

Seibel：你在领导项目时，口碑很好，大家都乐意为你效力。你是怎么领导团队，让他们富有成效，保持好心情？

Ingalls：我喜欢干这行。与其他人分享很有意思。这些活动可深可广，没有限制，因此很容易找到适合各种人做的事情。一直以来我就喜欢与其他人一起做项目。有时做得更好，有时则未必，这类工作有非常不同的状态。有的时候，你能发现所有需要做的事情，只要找到人做就行了。另外，有的时候，你真的不知道需要做什么，你得设法找出来。这些真是很不同的状态。

Seibel：对于如何成为一名优秀的技术主管，你有什么诀窍？

Ingalls：首要任务是弄清楚自己准备做什么，确定清晰的目标。如果具备一定的阅历，你便能真正明白自己准备如何实现目标，这样才能安排好不同成员都能做些什么，他们的工作如何组合在一起的。

有好多次，我在做项目时都能看清一切。这感觉非常管用，因为只要有人碰到问题做不下去，我可以立即告诉他们下一步该怎么做，或者怎么绕开问题。大家也会感觉到，如果你知道领向何方，他们就能立即意识到目标就在那儿，因为“他已经知道了”。这也增强整个团队的能力。

Seibel：对你自己想要的东西知道得太过清晰是否会让他人丧失动力，因为你已经掌控一切，他们没什么有趣的事可做？

Ingalls：其实，你可以任由他们做他们自己负责的那块，也许只要在有需要的地方介入一下，进行微观管理。结果往往都会很好。我有幸与一群很棒的组员一起共事了很长时间，我信任他们。信任是一部分，相信和你共事的人。另一点就是信心。目标清晰明确，就容易对它抱有信心。我认为焦虑和不安全感容易滋生糟糕的微观管理，以致事无巨细，你都想敲定下来。

Seibel：你还是普通员工时，有没有遇到过非常棒的团队领导？

Ingalls：Alan Kay是我生命中遇到的最好的上司。在施乐，我在他手下干了好些年，我们的组合很有意思，因为他知道自己想要什么，但对于我应该怎

么做，他很少会说。不过，他在技术上通晓一切，因此他是很好的批评家。我和一起合作的同事效率真的很高，因此我觉得他能感到进展顺利，不用过多介入。他给我们撑起了一把伞，他确确实实对自己想做什么心里有谱。

Seibel：*当人们组成小组协同工作时，每个程序员独自拥有一部分系统是否更好？“这是我的代码，其他人勿近。”还是整个团队拥有代码，任何人都能接近所有代码。*

Ingalls：我不得而知。我们现在开展的Lively Kernel项目的做法是不同人负责不同的领域，但相互之间不存在藩篱。更多是和专长、精力或目标有关。我正尽力回想这些时期的真正成功，具体是怎么做的。我从未在很大的团队里工作过，因此总的来说，人们都会各自单独负责一块代码。

Seibel：*换个话题，聊聊调试：你追踪过的最棘手的缺陷是什么？*

Ingalls：那是垃圾回收方面的缺陷。垃圾回收很难对付，因为造成问题的原因出现继而消失很久之后，问题才会显现。要想成功追踪像这样的模糊的缺陷，我只能想到断开代码。我父亲过去在战略情报局工作，他们以团队的方式工作，而他们所做的许多工作无非就是搜集情报，努力把有用的东西汇聚起来。于是，他们在报纸上看到的某些新闻片断可能就是一个密码，而他们就要将这些密码加以整理。

追踪这个缺陷也是同样的道理。我能做的，就是凭直觉发现是什么原因导致了这些情况下会发生这些缺陷。这个特别的缺陷，我深入挖掘了至少一天。最终搞定的时候，我欣喜不已，我儿子，记得那时只有四岁，给我颁了个“最有毅力调试能手奖”。

Seibel：*我猜这个缺陷出自Smalltalk。你有没有可用的符号调试器，或者你是直接分析内存的十六进制转储？*

Ingalls：这比那个很棒的Smalltalk调试器更低一层。我没法告诉你相关细节，大体来说，就是某个地方出错，实际上它会把你带到低层调试器。所以你会看到一堆八进制地址的内存。然后你发现诸如一个对象指向另一个对象等问题，这些原本不应该出现。于是你绞尽脑汁，“这怎么可能？”你一步一步找到导致问题发生的所有那些蛛丝马迹，然后设法把原因找出来。

Seibel：*这么说来，那个缺陷出自非常低层的位置。当你在很棒的Smalltalk环境中开发时，我想你使用的是符号调试器。你有没有采用打印语句？*

Ingalls：如果可以选择使用很好的调试器，我不知道有谁会用打印语句。原

因很简单，打印语句该放在哪里？你在那里插入打印语句。那好，你不会直接到那个地方，查看所有代码，难道只是加个打印语句？现在我大量使用打印语句这种调试方法，因为通常我找不到一个足够好的JavaScript调试器。

Seibel：*Smalltalk调试器缘何这么棒？*

Ingalls：嗯，你可以停在程序的任意位置，可以真正查看所有变量的全部绑定。你可以执行片段，直接在上下文中间评估表达式。

Seibel：*停在栈帧里任意位置？*

Ingalls：是的，你可以做出重大更改，然后继续进行。你可能会收到一个错误，把它显示在屏幕上，保存系统的整个状态，交给使用Windows机器而非Mac的人，然后他们就可以启动同一个映像，找到出错的地方，修正之后继续进行。总之在不同机器表示之间仍能保留完整的状态。

Seibel：*不变条件（invariant）是另一种调试工具。有些人非常热衷给所有方法和类不变条件设定形式前置条件和后置条件，有些人考虑问题的方式则更即兴。你怎么看待他们？*

Ingalls：我可能属于不那么形式的阵营。主要是出于最初的感觉，也就是尽可能让事物绝对简单。对于类型，我有同样的感觉。类型本质上是关于程序的断言。而我认为越是保持绝对简单越好，甚至不提及什么类型。在我看来，随着你对待系统的态度越来越认真，能够加入所有东西当然很好，不过，这有点鱼和熊掌想兼得的意思，比如说既要推断出类型，又不用看到它们，除非你想看到。

与类型归为同一范畴的还包括单元及可以对其施加的各种其他断言。这块迷人的领域我们必须利用这种综合的数学知识加以探索。我越发觉得，我们可以利用计算的各个方面，更多用鲜活文档、实作编程文档来编制文档。这样一来，就能形成对你真正有帮助但通常你看不到的断言，不过，一旦碰到问题，你又可以着手引入这些断言，并针对问题做各种测试。

Seibel：*你对形式化方法证明程序正确性有什么看法？*

Ingalls：我从来没这么做过。我倾向于把重点放在架构上，可以更容易地对事情作出断言。总之，如果你被允许在程序中做各种危险的事情，那么当你坐下来进行形式化证明，这会非常困难，因为每走一步你都得说："哦，这可能发生，那可能发生，那也可能发生。"如果架构很清爽，那么形式化证明几乎只通过阅读代码就能完成。你会说："哦，这只可能从那里来。我们

安全了。”

Seibel：你用过C++吗?

Ingalls：没有。C也没用过。

Seibel：不过你用BCPL和汇编做过开发，因此你也不是没用过低级语言。

Ingalls：没错。实际上，为调试Squeak生成的东西我也写过一些C代码。不过，我记得我们开发Squeak时，部分目的就是打造一个你能真正掌控的系统，除了Squeak，其他什么都不用知道。因此我打定主意不学C。John Maloney做了个Squeak到C的翻译器，这样我们就有切合实际的实现版本。实际上，我可能会去看一看那些C代码，但我一定会保证你不需要去折腾C。

Seibel：C++出来时，你肯定做过一番了解，毕竟，或许除了Simula那班人，你是最有资格宣称发明了面向对象编程的人之一。

Ingalls：我了解得没那么深入。相对C而言，C++从许多方面似乎都进了一步，但好像并没兑现最初的诺言，而我们在Smalltalk中已经体验到了。如果我不得不再次自底向上实现一把，也许我一开始就会选择C++，而不是使用机器码。我认识好几个人都是C++专家，我喜欢看他们做事的方式，因为在我看来，他们并不依赖C++去做C++根本不擅长做的东西，只是完全把它当作一门元编程语言来用。

Seibel：下面聊聊代码阅读。你是怎么上手一块新代码的?

Ingalls：泛泛而谈的话，这个问题有点难答。你一开始先要知道代码做什么或该做什么。我觉得自己大多会采用自顶而下的做法，我设法理解这些代码是什么，怎么组合在一起。看看都定义了哪些类和方法，它们都起什么作用。接下来就要看我们是出于什么原因去读代码的。有可能它是新东西，我们想了解是怎么回事。还有可能是它性能很差，于是对它进行性能分析，仔细查看。

Seibel：前面我们聊过Knuth，他的另一项爱好是文学编程。你写过或读过文学代码吗?

Ingalls：每当有充裕的时间去做，我喜欢用那种方式处理事情。我一开始写东西时，不加注释说明。一俟它能运转起来，我会写些注释。如果我很喜欢自己做好的东西或做出来的东西看似难以理解，我就会多写些注释。但我并不主张到处加上注释。在我看来，语言越好，所需注释越少。你会使用合理的变量名。这正是我喜欢Smalltalk中的关键词参数的原因所在。它确实能大

幅提升代码可读性。在JavaScript中，你可以在许多地方使用这项美妙的小技巧。这么做代价有点昂贵，不过JavaScript拥有这种花括号对象表示法，因此你可以使用关键词，它们真的就像Smalltalk关键词，同样以冒号结尾，这样就可以在花括号表达式里使用多个参数。它的确能使程序好看许多。

Seibel：嗯。那真是美丽与丑陋并存。

Ingalls：是的，没错。

Seibel：你说服过其他人采用那种风格吗？

Ingalls：事实是在我自己想到之前，我发现已经有人在那么做了。

Seibel：你认为自己是科学家、工程师、艺术家，还是工匠？

Ingalls：实际上，是上述这些角色的综合体。我觉得自己所受的物理学专业的课程教育使我受益匪浅。很多方面无非就是教我们如何考虑问题，如物理问题，设法分解物体上的各种力。当你要查看系统里正在使用的都是什么样的方式，它们是如何影响整个系统的，这时，你同样要用到物理学里的分析方法。对于那些同样也实实在在与空间有关的事物，比如，各种东西如何协同合作，不同的东西如何能变成相同，怎么做才能实现更好的结构，我感觉它们都是同客观物质一样有形。

我还记得早期所做的一次有关Smalltalk的讲座。我说："我们这一组所采用的像是科学方法，即，先观察，提出理论解释它，然后做试验验证。"我们对后续几代Smalltalk采取的做法也是如此。对如何让东西工作起来，我们先提出理论。然后建立按那种方式工作的系统。用过一段时间后，我们发现："哦，如果我们采用不同方式做这个，还有这块，结果可能会很不错。"于是我们建立一个新系统。总之我们就按这个循环不断前进，就像科学研究和发展。

我觉得自己工作时像个艺术家，因为我头脑中先有想法，想着把它变为现实。我觉得雕塑家也会有同样的感觉，给作品以生命。

就我们谈论的背景而言，我认为工程师和工匠几乎毫无差别。工程师只不过是技术领域中的工匠。有些时候我肯定也会有这种感觉，不过这些经历也会各不相同，尤其是当我做某些低层工作时。在我过去的经历中，我曾做过BitBlt最深层的部分，或者说Smalltalk字节码引擎。这些工作非常像工匠干的事情。而我又难得地做过好几次这样的活儿，让它们恢复正常，这就是手艺活儿。

Seibel：在我看来，工程师和工匠之间的区别在于，认为自己是工程师的会说："我们应该像造桥的那些家伙一样。桥不会倒塌。他们有一套可重复的施工过程。"工匠则会说："这更像是木工活。木材每次都是独一无二的，有些经验法则，但没办法保证特定的结果。"

Ingalls：因此，在这方面，我可能不怎么像工程师。我认为我强调系统的角度有所不同。我知道，有些人在做严肃的企业编程系统。这不是我关注或酷爱的。在你提到的四个角色中，就我而言，工程师的成分最少，其次是工匠，最像是艺术家和科学家之间的有趣组合。

Seibel：你提到自己曾离开业界一段时间，然后又回来。你是厌倦了计算机，还是因为生活中其他事情？

Ingalls：是因为生活中的其他事情。另外，这也是一次不错的休整，而且我挑选的时机正好，当我回来时，事情看似变化还不是那么大，只不过所有机器的速度都快了一百倍。

Seibel：对那些想成为程序员的人，你有什么忠告？

Ingalls：嗯，我认为一门不错的计算机课程应该有很大帮助。我的做法是安排自己学几门不同类型的语言，而且要各有优点。Smalltalk有很多优点，但也没法包打天下。我们还有逻辑程序设计，还有函数式程序设计。其实，Smalltalk也支持函数式风格的编程，而且涵盖诸多函数式特性。不过，就像我先前谈到的Lotus 1-2-3以及将英语转换成儿童黑话，我认为值得采用几种不同的计算环境，然后挑选问题，并用它们来解决，这样可以展示语言的优点，或者鼓励你以某种方式避开它。

Seibel：你认为编程有变化吗？程序员的成功历程有变化吗？仅作比较上层的事情，不学汇编或C，甚至不学Knuth书里的算法，这样做的程序员能成功吗？毕竟现如今你可以直接使用一些高级语言，库里已经包含大量算法。

Ingalls：要想对自己所做的工作有一种成就感，其实不同的人需要达到的水平也不尽相同。所以我认为，哪怕一个人用的就是集合类库，也就是说，他自己根本就没有进行编程，他一样可以感到自信十足。这么做只是意味着他是在另一个层面上进行操作。谁都知道，我们不会期望想使用图形来工作的人一定得来写BitBlt。你不必非得做与非门，因为你可以使用汇编语言。不论在哪个层面上工作，我认为都可以。如有更大的挑战在召唤你，你一定会深入进去，因为你不得不这么做，而如果你的兴趣被激发，你也会深入进去，

因为你愿意这么做。

Seibel：*那么你认为今天大多数程序员，也许他们工作的层面相当高，要是换个环境的话，他们是不是早就学习汇编和微程序了？或者你是否认为成为成功的程序员，人们需要的才能已经不同以往？*

Ingalls：可以说是，也可以说不是。从某种程度上来说，不论在何种层面上工作，情况都是如此，而且这种现象有望愈演愈烈。但是，眼下不少领域，我觉得更多就是根据公式进行的组合，还有其他领域则是处理更为原始的东西。

我过去研究物理学，不过那时我也有搞数学的朋友，我觉得自己的大脑和他们的并不同属一类。但是我们做的工作都很不错。在我看来，计算机行业也是如此。从事程序证明器研究的人自然与从事图形系统开发的不同。因此人们会寻找自己的长处和自己愿意从事的领域，也会发现自己工作起来不自在的领域。我觉得既有后天因素也有先天特质，而且情况一直都会如此。

或许有些系统有足够的层级和组成部分，足以让某人找到一块领域，做得既自在又多产，不过我觉得这些都是相通的。既有逻辑思维，也有构造思维。还有人的因素和创造力。人总是先天和后天的混合体，对我来说，这一点没有太大改变。人们好像正在设法做更大、更好的东西，但在我看来，它们还是大同小异。

Seibel：*与此相关的，随着越来越多的领域依赖计算，而且方式越来越特定，有些人想寻找一种方法，让“非程序员”也来编程。你认为这能实现吗，或者说，某个领域的专家，比如生物学家，总是得与程序员合作，构建定制软件以解决他们的问题吗？*

Ingalls：我认为将来会出现这种合作，因为生物学家对编程不感兴趣。他感兴趣的是找出这样那样的结果。而另外有人知道怎么用计算机处理这些材料，并且能帮他做这件事。我觉得让非程序员也能编程的东西就是应用程序。

Seibel：*我做过一个项目，设法为生物学家提供编程环境，依据的理论是他们需要的软件始终是特制的，应需而变。你不可能构建出一个应用，然后就算做好了，因为不到时候，生物学家不会知道自己真正需要什么，比如他们直到获得了一些生物学数据，才会发现自己真正想知道的是X，而从那些数据中提取X的唯一途径，除了编写程序别无他法。*

Ingalls：是的，如果我们能有个计算环境，把所有你要的信息都装里头，这

样你就能设法只借助其自显（self-revealing）特质来确定怎么解决问题，那应该会非常不错。但我觉得有人会对那个感兴趣，也有人不一定感兴趣。

Seibel：有哪些问题是我还没问但你认为我可能会问的？

Ingalls：阅读名人传记时，通常我感兴趣的那一面是，他们怎么安排他们的生活？对于那些不是他们酷爱的事情，他们是怎么处理的，如何与家人相处，怎么处理财务，以及如何平衡这几者间的关系。或者，他们只是避而不见，“不相干的事都见鬼去”，手头的工作不做好，天塌下来都不管？

Seibel：你有没有发现，在自己的生命中，有些时候你对编程的爱好太过强烈，以致损害到生活的其他部分？

Ingalls：是的，有些时候对别人太过苛刻，因为我得专注，需要保持专注。任何人只要爱上自己所做的事情，都会存在风险。我认为，要么学着稍加节制，要么就与人沟通，让身边的人都知道你正在处理这件事，你可能会在一周内搞定，但在那之前，他们不要招惹老爸。

Seibel：然后你赢得了“最有毅力调试能手老爸”称号？

Ingalls：正是如此，很对。另一点是，尽可能将从进展中获得的满足感反馈给那段时间与你相处的所有人，至少他们会意识到老爸在做的事情还不错，这样完成时就会皆大欢喜。

（编辑：李莉萍）

11

L Peter Deutsch

吴玎　译

“神童”L Peter Deutsch在20世纪50年代末就开始写程序了，那时他只有11岁。他爸爸带回家一份备忘录，上面写着为实现哈佛那台剑桥电子加速器的计算功能程序的资料，激发了他的兴趣。很快，他就出没于MIT，在PDP-1上实现Lisp，修改并改进MIT程序员的程序，而这些程序员的年纪几乎大他一倍。

在加州伯克利大学上二年级的时候，他加入了Genie计划，这是最早的一个基于微型计算机的分时系统，Deutsch写了大部分的操作系统内核。(第12章要谈到的Unix之父Ken Thompson也参加了这个项目，那时他是研究生，这个项目对他后来发明Unix产生了影响。) Genie计划商业化失败之后，Deutsch加入了施乐公司帕洛阿尔托研究中心（PARC），在那里，他从事Interlisp系统和Smalltalk的虚拟机的研究，并协助发明了即时编译的技术。

他曾是PARC衍生公司ParcPlace的首席科学家，Sun公司的院士，在Sun公司他发表了题为“分布式计算的七个谬论”的著名论文。他还是Ghostscript的开发者，Ghostscript是PostScript的解释程序。1992年，Deutsch所在的小组因为Interlisp获得了ACM软件系统奖。1994年他当选ACM会士。

2002年，Deutsch退出了Ghostscript的开发工作，转而学习作曲。现在，他更愿意研究作曲而不是程序。但他常常抑制不住编程的欲望，主要开发他自己设计的乐谱编辑软件。

访谈中，我们谈及了他觉察到的计算机程序语言存在的深层问题，包括对指针和引用的看法，为什么软件应该被当成资产而不是支出，以及他不做职业程序员的原因。

Seibel：你怎么开始写起程序的？

Deutsch：在我11岁开始的，说来有点偶然。我爸爸带回来一些关于剑桥电子加速器的备忘录，这个机器当时正在建造中，有一个小组负责设计它的计算功能，但一些设计备忘不知道怎么就到了我爸爸手里。我在他的办公室看到了这些文件，上面有些代码，一下子就引起了我的兴趣。

这个备忘录实际上只是另一个备忘录的附件，于是我问我爸能否搞到原始的备忘录。当他把那份备忘录带回家给我看时，我说："天哪！这东西真是太有趣了。"我好像问了爸爸我能否见见写这些东西的人。结果我爸就带我去了。更多的我就记不得了，那可是50年前的事情了。总之，后来我就写了一些剑桥电子加速器的计算功能的代码。我就是这么开始的。

Seibel：这是你11岁时的事情。到十四五岁的时候，就开始在MIT玩PDP-1了[①]，你爸爸那时是MIT的教授。

Deutsch：我14岁开始摆弄TX-0，后来就用PDP-1。我还记得，当时不知怎么就拿到了一本Lisp 1.5程序员手册，这手册相当老，是用滚筒油印机印的，还是那种紫色油墨。就这样，Lisp又激发了我的兴趣。我一直都很喜欢数学，所以Lisp在我看来就比较酷。我一直想自己试一试，但是我接触不到位于26号楼的大型机，于是就在PDP-1上写了Lisp的实现。

Seibel：还记得是怎么设计的吗？

Deutsch：想起来好笑，因为程序很小。你看过那段代码吗？只有几百行的汇编程序。

Seibel：看了，但没想着去理解。所有的工作就是将1.5程序员手册里的内容写成汇编程序吗？

Deutsch：不，不是。1.5版手册里主要是解释程序，我需要写的是阅读程序和标记程序，而且我还要设计数据结构之类的东西。我记得其实我是先从数据结构开始的，这是我一贯的做法。我小时的直觉总是能让我走对方向，虽然不是百发百中，但也八九不离十，这个方法还真不错。

而这几年我发觉自己变得有些迟钝，直觉也没那么准了。这几年我断断续续地做着一个大项目，想开发一个开源的乐谱编辑软件。我为此拉了几年小提琴，以前直觉可以指引我设计好数据结构然后水到渠成地写出程序来，现在这种方法已经行不通了。

---

① PDP是DEC公司的一个计算机产品系列。

Seibel：到底是你的直觉不准了呢，还是精力不足呢？也许之前也有直觉不准的时候，但你有精力去解决问题。

Deutsch：我觉得可能两方面的原因都有吧，但更多的是前者。所谓直觉，我认为就是从大量数据中提炼出解决方法的潜意识能力。我越来越少去沉浸到软件世界里了，能让我提炼的信息也越来越少。

我记得有人说过，要成为某个方面的大师，必须经过20000件这方面的事例的历练。现在我的情况是，45年的从业经验让我经历了这么多锻炼，但是现在这些东西却离我越来越远，记性不好了就是这样。我认为这就是我的现状。

Seibel：你觉得写程序最吸引你的是什么？

Deutsch：50年的从业经验让我明白，我最感兴趣的是标记符号系统，即语言。不仅仅是指正式场合用的书面语言（人类的语言），也包括那些讲话能产生影响的口头语言。程序设计语言无疑也在此列。

同样的，我开始学习编曲也是由于这个原因。音乐是一种语言，或者说是一套语言。而且你用这种语言进行交流的时候，不仅仅是表示出来，而且也会影响到别人。音乐比较有意思，它从形式上来讲，介于人类语言和计算机语言之间。它比人类语言更正式更有规范，但还比不上计算机语言的结构严谨。这也可能是我不选诗歌而选音乐的原因。诗歌就没有那么严谨。

一句话说，我就是立刻被这种东西吸引住了。

Seibel：你还记得你写的第一个有意思的程序吗？

Deutsch：第一个让我对内容感兴趣的程序，说起来应该是我写的第二个程序。我参与的首个程序是剑桥电子加速器用的计算程序，第二个则是一个浮点输出格式化的程序。

Seibel：那应该很复杂啊。

Deutsch：嗯……要是在二进制机上就复杂了，但在十进制机器上并没有那么复杂，我就用过十进制机器。只需要将字符串过一遍然后看小数点放在哪一位就好了，或者看看使用E还是用F的格式。但是在那个时候做什么都会困难很多，我那时用的是汇编语言和一个批处理机器，所以这个问题不那么容易。它不难，但是也不那么简单。这是我第一个自己想写的程序。

Seibel：你在高中的时候就经常出入MIT，但是大学却去了伯克利，你是想逃离东海岸吗？

Deutsch：有一点儿那个意思吧。我当时想离我父母远一点儿可能有好处。我认真考虑过的学校有罗切斯特大学、芝加哥大学和加州大学伯克利分校。后来挑起来也不困难，只有一个学校的天气总会那么好，所以我去了伯克利。这是我一生中最美妙的一件事。

去那里不久我就开始在Genie项目组里工作，然后我就去了……啊，其实在Genie之后还有伯克利电脑公司，然后才是施乐。

Seibel：*然后在伯克利你开始做一些比DPD-1 Lisp更大的项目。*

Deutsch：是的。Genie项目规模要大得多。最开始我几乎是一个人写了整个操作系统内核，大概有1万行的代码。

Seibel：*项目规模有数量级的扩大，这对你的设计过程有什么影响呢？*

Deutsch：让我想想那个内核里有些什么东西。现在看来那个程序并不是很大，我一个人就能完成。里面有几个很明显的功能模块。我在心里其实很清楚哪个模块应该跟哪个关键数据结构互动。实际上，数据结构并不是太多。有一个进程表，还有几个就绪列表。另外就是I/O缓存和若干用来跟踪虚拟内存的数据结构。然后，每个进程都有一个打开文件表。但是所有系统数据结构的描述用C语言的struct定义只需两页纸就能写完。所以这并不是一个复杂的系统。

Seibel：*那最大的系统是什么？你还记得是怎么设计的？*

Deutsch：我主持过三个大型系统的设计。第一个是Ghostscript，不算设备驱动程序的话——大部分驱动代码都不是我写的——大概有5万到10万行C代码。

然后是ParcPlace的Smalltalk虚拟机，我只做JIT编译器这部分，大概占整个工作的20%，有几千行代码吧，3000到5000行的样子。

然后就是Interlisp了，我做的部分有几千行的微程序（microcode），另外还有5000行的Lisp代码——这是我现在猜的。所以GhostScript大概是我参与过的最大的项目。

Seibel：*而且除了设备驱动程序以外，你几乎是独自一人写的所有东西。*

Deutsch：1999年年底之前，基本上每行代码都是我写的。一开始由我确定若干架构设计决策。首先第一条就是将语言解释程序和图形处理完全分开。

Seibel：*是用的PostScript语言？*

Deutsch：对。这么一来，语言解释程序不用关心图形处理相关的数据结构，只需调用图形库的API。

第二条是使用驱动程序接口来组织图形库的结构。这样图形库就知道怎么完成像素处理、曲线渲染、文字渲染等，但是它并不需要过分详细地知道像素是如何经过编码去适应不同设备的，也不需要过多地了解像素在特定的设备上是怎么传输的。

第三条，就是驱动程序需要实现基本的绘图命令功能，最初这个设计只是包括了绘制像素图命令`draw-pixmap`和填充矩形命令`fill-rectangle`。

渲染库将矩形和像素组传递给驱动程序，驱动程序则可以随意将这些东西渲染成一组可以显示的图片，或是为了显示把这些东西直接传送到Xlib或者GDI或者其他类似的东西里。上面说到的就是我做出的关于架构的三个重要设计决策，最后它们实现起来的效果也不错，这三个原则也是最基础的设计原则。重要的是我觉得如果某些操作是跨域的，而这些域又没有很多内在的耦合关系，相互之间也没有太多的互动，那么这些操作和这些域就应该在设计软件的时候被隔离开。

语言解释和图形这两者之间并没有内在的互动。图形渲染和位图的表示互动得多一些，在这里明确设置隔离比较好。

实际上我先完成了PostScript第一层解释程序，然后才开始写图形方面的代码。如果你打开说明手册，看看那些没有做任何跟图形相关的引用的所有操作符，这些实现都是在我设计图形之前做的。当然，我得设计分词器，还得先决定PostScript的数据类型的呈现方式和其他PostScript手册上要求解释器提供的东西。而且，在做第二层解释器的时候，有很多东西又需要返工。但是我就是这样开始的。

然后，我就凭借着我对语言解释器的设计经验，开始设计解释器的数据结构。大概用了3周，我就实现了输入`3 4 add equals`（3加4等于）能得到7的功能。并不是很难。哦对了，我那时是在MS-DOS环境下做的，MS-DOS外加一个简化版的Emacs和某个C编译器，忘了是谁写的了。

Seibel：*你已经为很多语言设计过解释器了。每次你都是直接开始写C代码吗？还是会先在小本本上设计一下数据结构图？*

Deutsch：这种事情对我来说已经很简单了，所以我都没想过要画图。印象中我会先研究PostScript的手册很长时间，可能会在纸上记些东西，但很可能就直接开始写C的头文件。我说过，我喜欢从数据入手开始设计。

然后我才开始考虑要有个解释器主循环的文件，需要做些初始化，需要分词器，还需要内存管理程序，还得管理PostScript的文件记法，还要实现各个PostScript操作符等，然后把这些东西大体按功能分成不同的文件。

后来我注册Ghostscript的版权时，我需要给他们一份完整的最早期代码实现清单。那时距最早开发大约过了10年，其间又发布了两个主要版本，我因此又回顾了一遍最初的代码、结构、各种命名等，有趣的是，大概有70%～80%的结构和命名约定没有变化。

这就是我所做的事了。数据结构先行，然后再初步划分成不同模块。我坚信把数据结构和一些不变的东西搞清楚了，代码就水到渠成了。

Seibel：*所以当你写头文件的时候，你是先写函数签名还是先写结构？还是两个同时进行？*

Deutsch：是结构。那还是1988年呢，ANSI C都还没出来，所以也没有函数签名这个东西。在ANSI C编译器差不多成为标准之后，我花了大概两个月，为Ghostscript的所有函数加上了函数签名。

Seibel：*你觉得你对编程的理解还有你编程的方式，从最开始到现在都经历了哪些变化？*

Deutsch：变化很大，因为那些我感兴趣的程序都在经历巨大的变化。我现在可以说这个话：最开始那一段时间我写的代码不过是学前班级别的。

原来我总在思考这样的几个问题：如何才能让程序功能更加丰富，如何才能让它更加有趣，如何组织代码等，以及怎样看待一门语言，你在用它表达时还要解决如功能性、可靠性、高效率和透明性等问题。

现在我对如何判断程序软件设计好坏有了更多的标准。这些标准适用于更大型、更复杂的程序，在这些程序中，最难的部分是系统级的架构问题。并不是说具体的算法等问题不够难，只是这些问题在很长一段时间以来已经不是我最感兴趣的问题了。

Seibel：*你认为所有的程序员都要达到这个程度吗？*

Deutsch：不是。我刚好获悉PARC研究中心的一位老友Leo Guibas刚刚获得了相关领域内的一个大奖。他从来都不像我那样在系统层面做工作。他是研究算法的，非常厉害。他能为很多难题提供一类分析或者优化的算法，由此产生了可以解决这些问题的一些新工具。这些工作都很棒。我们也需要大家成长为像Leo那样的程序员。

架构的基本准则和Leo等人解决艰深的优化和分析问题所用的算法设计的基本准则之间有不少类似之处。二者的区别在于，解决算法类的问题有长达5千到1万年历史的数学做基础。而在编程方面，我们则没有现成的基础来依靠。这也是为什么有的软件设计得那么糟糕：我们很多人还弄不清自己是干什么的呢。

Seibel：*有些人没有能力从系统层面去思考问题，那么是不是可以给这些人分配一些小活儿做呢？程序员和构架师是两种职业吗？或者说，因为系统级软件是“分形”的，所以每个从业者都应该具备那种系统层面的思维？*

Deutsch：我觉得不能说软件是“分形”的。虽然能分形更好，但是我觉得它不是，因为我们没有好工具去处理慢慢变大的系统。系统成为大规模的软件系统后，会发生质变，这与它从小系统变为中等规模的系统是截然不同的。

但是关于谁才能够胜任软件开发工作这个问题，我没有很好的简单答案。我觉得软件越深入底层，就越需要好的人去做。这个观点有些精英论，但我就是这么认为的。

如今的问题是，软件和非软件的界线变得模糊。比如说网站构建，如果网站带一些略微复杂的跟用户互动或者是状态追踪的行为，你就需要专门的开发工具去构建这样的网站。使用这些工具的过程有点儿像编程，但具体方法又不太像是写程序，当然这只是我的理解，我并没有用过这些工具。

所以，对你的问题可以有这样一个答案：今后，越来越多原来认为需要编程的工作，变得不再是“编程”了，差不多任何人都能“编程”了，并且都能干得不错。

你听说过关于电话和电话接线员的老故事吗？是这样讲的。在电话普及的早期，普及速度很快，非常多的人被聘为电话接线员，因为那时我们还没有拨号电话。有人根据这个增长率估算了一下，然后惊呼道：“不出二三十年，所有人都会成为电话接线员了！”这是以前发生过的事，我觉得类似事情也可能发生在一些大的编程领域。

Seibel：*程序员就这样被取代了吗？*

Deutsch：这要看你想要编什么样的程。我最近这五年多时不时思考的一个问题就是：为什么编程会如此困难？

编程有算法方面的问题，这方面跟数学联系紧密。你可以用数学作为基本模型来分析问题，可以使用数学方法和数学思维。这样做并没有让编程变得简单，因为没有人认为数学很容易。因此如下三个方面的难度还是很匹配

的：一个是工作本身的难度，一个是你对这个工作的理解，还有一个是你对完成工作所需的技能的理解。

我觉得编程还有一方面的问题是，几乎所有的编程语言所描绘的世界都和我们的感觉、头脑和社会所认识的物质世界格格不入，所以要想让人干好编程，这个想法本身就很怪异。一个真正好的程序员必然是“有些错乱”的。可能“有些错乱”这种形容有点儿过，但是实际上，成为一个正常的人才的品质和成为一个出色的程序员的品质，二者有一部分重叠，但是也有相当大的区别。当然，我这里指的是非常出色的程序员。

冯·诺依曼机和Algol类语言的世界跟现实世界如此迥异，我很惊讶我们竟然还能建立起这么庞大的系统，即便这些系统的功能十分糟糕。

可能喷气式飞机能在天上飞这件事更让人吃惊，但喷气式飞机毕竟是真实世界里的产物，有数千年的机械工程理论做基础。软件工程却是一个这样奇怪的世界，建立在奇怪的基础准则之上。真实世界则是建立在亚原子物理上的，以这样的层次上升，亚原子，原子，然后是化学。在这方面我们总结了无数的新属性，我们也有一切必要的工具来让事物运转自如。

现实世界中没有长得像内存地址或者是指针一样的东西。我们知道实实在在的“物体”（Object），但是我们不知道到底什么东西是计算机专家说的“对象”（Object），这个词其实是被他们误用了。

Seibel：*更别提范围上的差别了，2的64次方是一个巨大的数字，一秒钟要发生几十亿次的事情也未免太快了。*

Deutsch：但是在真实世界中我们并不关心这种事情。你知道阿伏伽德罗常数吧？10的23次方？虽然我们的世界里有很多细小的事情聚在一起同时发生，但是我们并不关心这些，因为你不需要从亚原子的角度去理解一张桌子的存在。

百分之九十九点九的情况下，你总可以从整体的角度来理解一个物质的物理属性，你想了解的任何事情总能通过从和一个物体打交道来了解。但是在软件世界里，很大程度上来说不是这样的。

人们总是试着做软件的模块化结构，而且这种手段越来越先进。但是在我看来，这种做法与我们观察周遭事物的悠然自得还是相去甚远，实际上，就算这个东西是由10的23次方个原子组成的，也丝毫不会让我们忐忑不安。

软件最重视细节，细节是软件设计里最重要最基本的东西。除非我们能以全新的方式提炼和组织软件，让人们不再去操心软件内部的小零件是如何

相互作用的，否则就于事无补。而实际上我们现在还差着十万八千里。

Seibel：*这是技术上的原因，还是软件本质上的问题？如果是前者我们还可以改进。*

Deutsch：必须彻底从头来过。因为在现实世界中没有指针这个东西，就必须抛掉所有带有指针概念的程序语言。你必须深刻认识到信息占据着空间，在一段时间内存在，并位于某个特定的地方。

Seibel：*当你慢慢地从写小块代码成长到开发大型的系统，你是仍然用写小型程序的方式去给大型系统一点点添加新内容呢，还是用一种完全不同的方式去开发大型系统呢？*

Deutsch：我用了完全不同的方式。我的第一个大型程序是为哈佛大学的UNIVAC写的，第二个则是MIT的PDP-1。在60年代早期，我读高中的时候，大概考虑过三种完全不同的系统或是程序。

其中有我为PDP-1开发的Lisp解释器，还为Jack Dennis的改装PDP-1做了操作系统方面的东西，另外还有这个机器的文本编辑器。

这三个我一直在做的东西基本上都是庞大的。为UNIVAC开发的东西和我以前做的东西的不同之处在于，我需要开始做数据结构设计。这是我编程过程中的第一次质变。

我开始会写所谓的功能块了，但是我没有意识到它有什么特别的意义。我知道可以只写一个程序的某几个部分而不用操心其他部分，但是接口问题除外，随着程序的规模变大，接口问题十分突出，可我那时好像没想过。

这种转变在我做下一个大项目的时候开始发生。那个项目是本科的时候在伯克利做的，就是上面说到的Genie 940分时系统和一个QED文本编辑器。我写了个汇编语言调试器，现在我已经不太记得了。

最具有“系统”味的项目就是操作系统了。我并没有写出整个操作系统，但基本上写出了全部的内核，用汇编语言写的。我们现在谈论的是有点儿大的程序了，大约是1万行汇编语言代码。这个系统包含进程调度器，有虚拟内存，有文件系统，实际上有好几个文件系统。

另外在数据结构设计方面还有更困难的问题。我记得的一个是跟活动进程表有关的——问题是怎么设计它，如何让系统判断进程在什么时候可以运行，什么时候不可以，诸如此类的事。在数据结构里有关于追踪虚拟内存的设计，但是后来接口方面的问题开始出现——问题并不在操作系统内部，因为操作系统太小，只设计了单片独立的内核。

接口的问题是在另外两个很重要的方面出现的。第一个是用户程序和内核之间的接口。系统调用如何实现？参数如何布置？我知道940 TSS的早期版本里面，读写文件的基本操作和Unix的read和write命令一样，给出基本地址和计数就好。这样做没什么不好，但是有时候这种设计不是你想要的，你需要的其实是一个数据流接口。但那时我们并没有听过这个概念，我们不知道可以将操作系统里面的一个组件打包，再以用户级别的代码封装并提供一个不错的接口，在read和write之上构建getc和putc方法。所以我们只能在后面几个版本里面加上跟getc和putc等效的操作系统调用方法。

第二个接口的问题也是在MULTICS模式下出现的，一开始我们就将系统内核和今天所说的shell完全分开。在开发的早期阶段我们根本不知道可以在没有任何特别权限的情况下构建一个shell。shell是个用户模式的程序，它有许多很特别的权限。但是内核可以给shell哪些设施，即哪些事情shell可以直接做，哪些应该由内核调用，这方面还有些问题。

在开发过程中我们目睹了接口问题慢慢浮现。也就是那个时候我模模糊糊地感觉到，实体间的接口应该拿出来单独设计，而且这是一个很重要的问题。

所以你问我设计大系统的方法是否有别于设计小系统，我的回答是：的确不同。我做的系统越大，我在坐下来写代码时就越会反复自问：这些东西间的接口应该是怎么样的？哪些数据被传入，哪些数据被输出？有哪些应该在接口的哪一面实现？我越来越多地在处理这些问题。这些问题确实影响到我写一块块局部代码的方式。

Seibel：*这是在大型系统中工作过后的一个很自然的结果——那些系统太大了，所以你不得不去把它们拆开变成小块。*

Deutsch：就是这个意思。从这个角度来看，我们可以说软件是分形的，在很多层次我们都需要对它进行解构。但是我认为这种解构在高层和底层上有质的区别。在底层做解构时，你不需要去考虑什么资源分配，但是在高层时，这个必须要考虑。

Seibel：*你在工作中是否碰到一些很好的程序员，但他们却只能在特定范围内发挥作用？比如说一些人只能掌控一定规模的程序，超过某个规模，他就想不到要把系统分解为小部分。*

Deutsch：我遇到过一些很有才华的程序员，但是没有做过大规模的程序设计。在做Ghostscript的时候，我跟两个工程师有很大的分歧。他们是在

Ghostscript转到我公司的时候招进团队的。这两人有能力，也很勤奋，经验丰富。我觉得他们是很好的程序员、设计师，但缺乏系统级思维。他们不知道去思考某些变动会引出的连锁反应，他们甚至没有意识到这是个根本性问题。对我来说，区别就在于，一拨人知道在做大规模设计时要问什么问题，而另一拨人则莫名其妙地对这些问题熟视无睹。

Seibel：*但是你认为这些人如果不做整个系统的架构，还是可以很好地完成工作的？*

Deutsch：是的，那两个工程师就为公司做出了很多贡献。其中一个做着一件对公司经营来说很重要但却费力不讨好的工作，另一个重写了大量我以前写的图形处理的代码，而且他重写的代码运行起来效果更漂亮。所以说他们都是很好的、有才华的、有经验的人，只是他们看不到大局——至少我这么认为。

Seibel：*是不是有什么特殊的技能让你成长为一个好的程序员？*

Deutsch：我可能会给你一个很“前卫”的答案。虽然我也有留长发的日子，但其实我不是一个很前卫的人。在我鼎盛时期，我有特别可靠的直觉。我凭直觉做事，出来的结果往往都是不错的。其中有一部分可能是运气好，另外一部分，我就觉得是我与生俱来的，有些解决问题的方法往往是在我潜意识里的。我就是有这样的天赋。我知道这个答案可能不能让你满意，但是我确信就是有些与生俱来的东西让我发挥所长。

Seibel：*你年少的时候就出入MIT，是个超常的神童。你会不会碰到机会这么说别人：“哇，这家伙真聪明，但他搞不定这个问题，我却可以。”*

Deutsch：其实没有。我只是记得当我开始重写Dennis的PDP-1文本编辑器的时候，还只有十五六岁。原来的编辑器代码是一两个来自“技术模型铁路俱乐部”[①]的人写的，那是帮绝顶聪明的人，不过我当时觉得代码中的不少地方一团糟。

我不是说同我一起工作的人跟我有什么差别，我只是说我想象中的代码和我真正看到的代码是不一样的。我可不想由此评判那些人如何如何。

我会自得其乐地沉浸在我所称的符号世界里面。符号及其模式就是我的精神食粮。但是很多人并不是这样的，甚至我和我现在的搭档之间都有这样

---

① Tech Model Railroad Club（TMRC）是hack文化发源地。

的差异。我们都是做音乐的，都作曲，都搞声乐，但是我是从一种符号的角度来看待音乐的。我作曲的时候多半只是用笔和纸，我写下音符，而不是用钢琴去一个个尝试，我能听到它们然后就有了主意。

但是我的搭档会靠吉他作曲，他用吉他试弹，琢磨，或许还会在钢琴上再过一遍，但是他从来都不会写下来什么。如果催他写他可能会记下一些和弦，甚至有时我猜他只是做文字描述。他从来不会从符号的角度去看待作曲。

所以有的人会这么做，有的人不会。如果让我总结一下，我会略带优越感地说，真正要编程的人都是能生活在符号世界里面的人。如果你对这个世界不适应，那编程就不是适合你的行当。

Seibel：*有没有对你重要的导师？*

Deutsch：有两个。其中一个是Calvin Mooers，已经过世了，他是信息系统领域的先驱。我想他是“信息检索”（information retrieval）一词的创造者。他本来是学图书馆学的，我好像是在高中或者大学的时候遇到他的。那时他试图设计一种可以直接被人们使用的编程语言，但是他对编程语言一点也不了解，而我有相关的知识，因为我做了一个Lisp的系统，并且学习了其他的一些语言。

所以我们碰在了一起，最后他弄出来的那个程序语言我想应该算是我和他一起设计的，语言的名字叫做TRAC。那个时候他给了我全力支持。

另外一个人是Danny Bobrow，我时常会认为他是我的导师。其实我们做了很长时间的朋友，但是我又把他当做我事业上的导师。

如果说到教我如何编程，如何设计软件，MIT里没有一个人可以算是我的导师，伯克利也没有。在PARC研究中心，只有一个人影响到了我编程的方式。他叫Jerry Elkind，本身不是程序员，在PARC做过计算机科学实验室的经理。

他对我的帮助在于，教会了我“度量”的重要性。很多时候，可能超出你的想象，我们坚信的东西，或者直觉，其实是不正确的，所以我们需要度量，甚至要度量那些你觉得没有必要度量的东西。这一点对我产生了深刻影响。

当我做的东西需要大量计算和数据的时候，我总会做一些度量。这个习惯我保持了35年，就是在PARC中心养成的。

Seibel：*在我为这本书联系采访对象的时候，你是唯一一个对“编码者”（coder）一词有强烈反应的人。那么你觉得怎么称呼自己才对？*

Deutsch：我得说现在的我对“程序员”（programmer）一词也有轻微的反感。在创造一个可以正常运转的软件的过程中，有许许多多不同的角色、步骤和技能是不可或缺的。有些人自称为程序员，但是这没有准确说明他们为这个过程带来了何种技能。

但是，程序员这个称呼至少包括了很多内涵，也早已为大家接受。编码者这个词大概就只概括了所有环节中最细微的一个环节。我认为，在创造一个可用的软件的过程中，编码者所起的作用只比盖楼时的垒砖者高。

做一个编码者本身并没问题，做垒砖者本身也没问题。只是，构造软件需要非常多的角色，而编码者这种叫法只涵盖了一小部分而已。

Seibel：*那你觉得什么词能完整地表达出这个意思呢？软件开发者？或者是计算机科学家？*

Deutsch：对计算机科学这种说法，我也不太满意，而且我还能很容易地证明科学这个词不应该用于计算机。现在所谓的计算机科学其实是工程学和应用数学的结合体，它跟科学研究中的“科学”大相径庭，因为科研的主要任务是把自然现象描述得更加完善。

如果非让我挑一个词，我会选软件开发者（software developer），因为这个词基本可以概括从架构到编码的所有职位。虽然这个词也没有完全包括软件开发过程中的所有工种，但是它基本上概括了我做过的所有工作。

Seibel：*那些没有被包括的是些什么呢？*

Deutsch：还没有包括理解问题域的过程，以及引出并理解需求的过程。也没有包括，至少没有全面包括类似反馈回路的过程，如从测试开始到软件发布后的一切活动。软件开发者这个词只是界定了在公司里开发软件的这群人，但是很少涉及软件企业和客户或外围世界的联系，而正是后者才是决定软件开发最关键的因素。

Seibel：*你觉得这些方面在改进吗？现在已经有人提倡要在开发早期就跟客户和用户沟通，并把这种沟通作为软件开发的一个环节。*

Deutsch：有的，像极限编程（XP）就在这样做。两个事情让我不是很热衷于XP。XP提倡在开发过程中跟客户保持紧密的联系，我想它是基于两个考虑。第一个是以为这样做就能更充分理解和满足客户需求。或许可以做到，我没有一手资料，但我有些好奇，因为客户往往并不能了解他们自己的真正需求是什么。

另外，之所以它要保证跟客户的紧密接触，是因为想要避免过早的泛化或过度设计。这其实是把双刃剑，因为我就见识过这一开发过程走入两个极端，过早泛化和过早特化都会发生。

所以我对XP持保留态度。在项目完成之后怎么办？它的可维护性如何？被支持得怎么样？是可以演进的吗？如果原开发者离职了怎么办？由于XP不要求文档，所以我对这些问题非常担心。

我曾和很多人争论过这个问题，他们热衷于快速设计原型，总之没有把软件开发当做工程问题来对待。我很难想象如果不以工程学的视角来开发软件，软件又能持续多久。

Seibel：*能不能具体说说“泛化”和“特化”的问题？*

Deutsch：在我事业的顶峰期，我很擅长做的一件事——我不敢说是用完全系统化的方法做的——就是在适当程度上进行泛化，从而正确地预留出今后几年的演进，而当时的演进方向还十分模糊。

回想过早特化的一个例子，我记得在Ghostscript做一个架构决策时，我们显示彩图时使用像素而不是平面做基础，以像素为基础，就可以让位图的显示适应机器宽度。

不用平面而用粗重的点表示，导致点状的色彩显示会很奇怪。特别是打印机对一些特殊任务要求的颜色不是标准的CMYK色时，比如要求准确匹配银色、金色或者其他淡色彩时。

用像素图，大体上有两种在内存中表示的方式。一种是用数组将RGB标准或CMYK标准的像素数据存储起来，通常显示器控制器就是这样工作的。

另外一个存储方式在印刷行业比较常用，是用三个数组分别将所有像素点的红、绿、蓝三种色素的数量存放起来。你若是逐个像素作处理，这个方式就不够方便。而且这种做法不会预先限定显示图像的墨水种类和色块类型。

Seibel：*所以打印机需要打印金色的时候，只用把这个颜色加上就好了。*

Deutsch：对。当然这种需求在办公室和消费者级别的打印中不常见，但是在胶版印刷行业是经常要有特殊色层的，所以这是一个不充分泛化的领域。

这个例子表明，即便我考虑充分，技巧纯属，却仍然错失良机。但我的观点还是没有充分表达。我讲了一个谨慎的预测却导致了不充分泛化的例子，这个例子其实削弱了我上面的观点。但是我能告诉你为什么会出现这样不同的预测，因为Ghostscript是由一个很聪明的程序员弄出来的，但这个程

序员对印刷行业一无所知。

Seibel：这个程序员就是你。

Deutsch：是。Ghostscript最开始是作为PostScript文件的预览器来设计的，因为那个时候没有这种产品，PDF还没出来。如果要对这个故事做个总结，我会说需求总是在变化的，而且总是在或想要往你预料不到的方向变化。

应对这种情况一般会有两种认识。第一种和XP的看法类似，认为需求总是在变，故而软件是不能持久的，需求一旦改变，就应该开发新软件。我认为这种说法有一定的道理。

老话说得好："及时、便宜和好质量，三者中只能满足俩。"如果你有快速便宜地生产软件的方法，那么得到的产品一定不怎么好。但是这种认识还说，不要奢求能用很久的软件。

在这种想法背后，应该是关于软件的不同理解：到底把软件当做公司开销，还是把软件当成是公司资产。我赞成后者。我在ParcPlace工作的时候，Adele Goldberg在四处宣传面向对象的设计。一方面是单纯地宣传对象的概念，一方面也是借面向对象语言与设计试图说服客户和潜在客户："瞧，应该把软件视作一种资产。"

因为资产是需要持续投资和维护的，所以在维护一个日渐丰富的软件库上的开销也是理所当然的。这就让财务变得更复杂，你不能把开发软件的成本只记在当时创立软件的项目或客户身上。你要像看待资产一样去看待软件开发。

Seibel：就像投产一个新厂房。

Deutsch：正是。在推销"对象"的时候我们总会说设计良好的对象是可以重用的，所以对设计的投资会在以后很轻松地得到回报。

我至今仍相信这一点，但不像以前那么坚信了。现在我看到的可重用对象都是要么很大要么很小。而之前我们在推销"对象"这个概念的时候，说的是类这种规模。而现在除非在类中映射现实领域知识，其他的类重用情况太少见了。

我现在见到的重用，有像图标或者单个网页模版这种小东西，也有像带扩展层架构的整个语言或者大应用之类的大东西——比如说Apache或者Mozilla。

Seibel：所以现在你对最初的对象重用这个说法没有那么大的信心了。那问题是出在这个理论本身呢，还是因为现阶段就只能这样了？

Deutsch：我不再把自己当做“计算机科学家”的部分原因是，我目睹了50年的软件开发实践，而最近的30年都没有看到任何巨大发展。

如果把目光放到编程语言上，我可以肯定地说，最近的40年，编程语言并没有质的飞跃。现在流行的语言并没有比Simula-67有本质改进。我知道这听起来有点奇怪，但我是认真的。Java并不比Simula-67强多少。

Seibel：那Smalltalk呢？

Deutsch：比Simula-67要好点。但是现在的Smalltalk基本上就是1976年就有的东西。我并不是说现在的语言没有30多年前的语言好，事实上我现在编程都用Python，我觉得它比30年前的任何语言都要好很多。相比Smalltalk，我更喜欢Python。

所以我刚刚谨慎地用到了“本质”这个字眼。现在我知道的在广泛使用的语言，都有指针的概念。我不知道在指针的使用上，编程语言能有怎样质的改变。

Seibel：所以你觉得Python和Java的引用其实就是指针。

Deutsch：是的，肯定是的。用Python和Java在小范围内也会出现C和C++除内存泄露外的所有问题。

关键问题是，对于系统中信息的共享和访问模式，我们并没有一种语言上的机制来理解、描述、控制或者是推导它。指针的传递和存储是一种本地操作，但是这些操作的结果其实隐含地勾画了这个画面。且不说多线程应用，即便是在单线程应用中，数据也在程序不同的部位中间游走，各种引用指向程序的不同部分。即便是设计得最好的程序，里面也会有类似的三四个复杂模式在交替进行，你无法描述、推理或者总结出程序的较大单元的运作，从而制约小单元的行为。人们在试图解决这个问题，但是我觉得到目前为止没有任何突破，也还没有一个令人满意并被广泛使用的解决方案。

Seibel：函数式语言可能是一个方案？虽然它没有被那么广泛地应用。

Deutsch：可能，虽然函数式语言有其他的问题，但是它们的确是解决上述问题的一个方向。

我总会有想要设计一种程序语言的冲动，就是没有付诸行动。如果我真的着手设计，会本着这样一个原则：把功能部分和其他部分很清晰地分开。前者专门处理数据，但是不会有指针的概念；后者会负责共享、引用和控制这几个东西的模式。

作为一个跟编译器和解释程序打交道的人，我能想出多种方法实现这么一种语言，不会把一个体积巨大的数组复制过来复制过去的。函数式语言已经在这方面领先了，有很多聪明人在研究Haskell和其他一些类似的语言了。

Seibel：*Haskell那伙人会不会这样说："这东西是我们的monad，本质上的区别是在数据类型系统里面。"*

Deutsch：我一直都不理解Haskell monad。其实从ML开始，我就没有特别关注函数式语言了。

有种不为太多人所知的语言叫做E，这种语言有很强的能力符号系统，它跟Hewitt的角色语言和基于能力的操作系统都有点关系。这个语言用端口（或者叫通信频道）来连接两个对象，而且被连接的两端彼此完全陌生，这种设计和指针就很不一样。指针是单向的，带指针的实体非常清楚另一端的实体，透明度极高。

我对程序语言有个模糊的高要求：可以进行函数式计算，但是又不需要共享对象。在这个语言里，有一套序列化的端口，当你要通过引用与对方通信时，语言本身的特性会让你知道对方是什么，这需要由多源通信来处理，所以应该经过序列化或者仲裁之类的处理。这里没有属性访问的概念，也没有存储进属性的概念。

有些语言的API写得并不清楚，所以实现起来会有很多不定因素，这些语言往往对其更高层面的通信模式语言不详。比如说，一个常见模式是，有一个对象，你把它传递到第三方，让第三方做些事情，然后还需要让对象返回。这就是一个分享的模式。你作为调用者，不会放弃指向你传送出去的对象的所有指针，但是你知道在第三方完成任务之前，你是不会去引用的。

这是个简单的例子，如果人们可以用自然语言很好地描述这个模式，写出的代码就会满足意图。

我一直都没有动手去设计这个语言，最大的原因是我还不能从一定的高度去描述共享模式和通信模式，还不能很好地组织语言将它描述好。这也是我认为30年来软件开发进步甚微的原因。

我的博士论文是关于程序正确性的验证——而我现在已经不用这个词了。我要说的是，一个开发环境，应该尽可能让你对自己的代码有足够的信心，让你相信代码能完成你想完成的功能。

最初关于程序正确性的概念，是说以某种方式写一些断言来保证程序能够达到预期目标，同时这种方式还能自动地被代码检查。但是这个过程其实问题不断。我现在觉得，想要保证程序达到我们的预期，断言、归纳断言什

么的都不是最好的方法，相反我们应该改进语言本身，让它有更好、更强大和更深入的声明式标记法。

Jim Morris会总结一些关于计算机的警句，我很喜欢他。他说过类型匹配检查是种原始的验证方法。而我觉得语言上的突破就应该出现在这个方面，有更好的方式去声明程序要怎样组织，应完成什么任务。

Seibel：*你的意思是，能在程序里直接表达意图，可以直接说："我现在把这个对象的引用传递给第三方子系统，在第三方完成工作之前，我不需要这个对象也不会收回它。"*

Deutsch：对。Sun公司曾做过些实验性的研究，那是上个世纪90年代早期我还在Sun的时候，他们开发了一个有类似概念的程序语言。在MIT，Dave Gifford也做过相关的研究，他有种语言叫FX，这种语言试图将计算模式中功能和非功能的部分分开，同时也让指针的游走路径更清楚。

但是我觉得这些相关的研究解决问题的层次都太低。这方面需要有突破性进展，让Windows Vista这种残次品毫无必要也不可能产生，我们需要的是崭新的思考方式，思考程序究竟是什么，怎样将其融合在一起。

Seibel：*所以，尽管Python在本质上并不比Smalltalk好，但你还是更喜欢它。*

Deutsch：嗯。有几个原因。Python很好地回答了几个关键问题：一个真正的程序应该是怎么样的，运行一个程序意味着什么，以及成为一个程序的一部分意味着什么。Python有一个模块的概念，模块主要声明从其他模块得到什么信息。同时，还可以创建一个模块或一组模块来供第三方使用，并且能让第三方一目了然地知道这些模块依赖什么，其有效范围有多大。

如果要在Smalltalk里做同样的事情，就会很笨拙。就算是在可视化的模式下使用Smalltalk，也没有类似的程序实体供人使用。VisualWorks是ParcPlace版的Smalltalk，用了三四种概念来充实一个实体，好让这个实体比一个单独的类要丰富。但是这些概念在不断地变化，没有被开发工具很好地支持，至少是没有特别好的图形化支持。也没有一个很好的机制将相互依赖关系搞清楚，让机器可以处理。所以当你用图形化的方式来做开发的时候，共享信息变得很困难，你只能与人共享一个整体内容。

如果用文字的形式来写程序，即所谓归档（file out），则绝对没办法在中断点之后再让程序回到原来的轨道上，因为图像状态已经改变，无法复原，也不能靠再读入源代码来重建。可能你在程序中做了一些随意的动作，你的一些静态变量的值可能变了，这些东西你都没法知道，你没法对每个环节都

把握准确。

我参与了VisualWorks的开发，开发过程中我看到上述问题在不断地重复，而这些问题在没有图像概念的语言里面是不会发生的。图像概念与快速原型和快速开发的世界里的很多东西一样，它在完全由个人掌控的小项目里面扮演了很好的角色，但是当需要把软件当成资产并与他人共享的时候，这个概念就一塌糊涂。我想这是Smalltalk的开发过程中一个很严重的弱点。

我喜欢Python的第二个原因是，我不需要像以前那样在写程序的同时去记很多东西。当然，这里面也有我脑子变得不好使了的原因。我需要把东西都放在眼前，而Smalltalk一屏只能显示一个方法，这个局限性让我有点崩溃。对我而言，用Emacs写Python程序有其先天的优势：我一次至少能看到10行代码。

我跟还在VisualWorks工作的几个老朋友聊过，他们正准备把它的对象引擎和实时代码生成器开源——虽然是我以前自己开发的这些东西，但是我还是认为它比市面上其他类似的产品要好。那么，现在我们有Smalltalk，它有完善的代码生成机制，它也相当成熟，有20年的历史，已经特别可靠，它相对而言比较简洁，是一个高效的、可重新定向的实时代码生成器，在不需声明类型的语言中能非常好地工作；另外，我们还有Python，一种很不错的语言，有很好的库和蜗牛一样慢的实现。我觉得把两者结合起来，会有惊喜。

Seibel：这是不是就是pycore这个项目背后的创意？用Smalltalk来二次实现Python。

Deutsch：本来是的。但是我慢慢发现需要做的东西比想象中的要多。Python中的对象模型和Smalltalk中的对象模型并不对应，没有办法用简单的方式将它们一一映射起来，映射需要额外多层的方法去调用杂七杂八的东西。

即便是这样，即便是用Smalltalk的实时代码生成器来生成Python的代码，跟用C做的解释程序里出来的东西也是同等水平的。所以我还有个想法就是看能不能把Smalltalk的代码生成器开源，调整它的代码生成器，让它适应Python的对象模型，从而使得Python的数据呈现并不是那么困难。

然而这条路也走不通。Eliot Miranda大概是VisualWorks项目组我认识的人当中最棒的一个，他尝试过开源，但是Cincom却说：“不行，这个是我们的战略性资产，我们不能将它开源。”

Seibel：你刚刚也说过应该把软件当做资产的。

Deutsch：但这不意味着不管什么时候都不能让别人用。

Seibel：*除了使用Smalltalk，你原先还是一个Lisp程序员，但是你现在也不用它了。*

Deutsch：我的博士论文就是一个长达600页的Lisp程序。我曾经是Lisp的狂热使用者，从PDP-1开始就是，并且囊括了Lisp、Byte Lisp和Interlisp。我不再使用Lisp的原因是，我受不了它的语法——是的，语法很重要。

一套程序语言就像一个三脚架，三只脚分别是语言本身、库和工具。判断一个语言的好坏的关键就在于这三者之间的协调。Python在三方面都做得很好。

Seibel：*"工具"包括了语言的具体实现？*

Deutsch：是的，也包括。Lisp有首屈一指的可伸缩性，但是它对于用户的可读性却很差。我不知道现在的Common Lisp库做得怎么样，但我要强调的是，语法很重要。

Seibel：*有些人很喜欢Lisp的语法，有些人却无法忍受它，你觉得这后面的原因是什么呢？*

Deutsch：嗯……别人怎么想的我不清楚，但是我能说说我不喜欢Lisp语法的原因。有两点。第一点其实我早些时候提到过，就是我现在年纪大了，所以对我来说重要的是，在固定大小的屏幕上能够呈现的信息密度越大越好。中缀式语言（infix language）的信息密度就比Lisp的要大。

Seibel：*但是几乎所有语言都是前缀式的啊，除了语言里的计算符号之外。*

Deutsch：并不全是，像Python里的列表和有序集合（tuple）还有字典结构都不是。它们是用括号的。但是字符串格式用的就是中缀。

Seibel：*就像用了FORMAT的Common Lisp。*

Deutsch：可以这样说。但不仅仅是上面那些东西。比如说更常用的，像循环语句，就不是用前缀，而是关键字和被循环的对象配合实现的。虽然从这一点上来说，它比Lisp要冗长一些，但是说到这点，就不能不说我不喜欢Lisp的另一个原因：Lisp在词汇上太单调陈旧了。

Seibel：*我记得Larry Wall（Perl的创造者）这样形容过它：表面漂浮着指甲壳的燕麦粥。*

Deutsch：其实我还说过Perl像块狗屎。我觉得Larry Wall谈起语言设计的时

候经常神经兮兮的——Perl语言其实令人作呕。这个问题就此打住。

读Lisp代码的时候，为了明白它的意思你必须要做两件事，而在Python里面就没这个必要。

首先你要弄清楚那些烦人的括号。这不是个智力劳动，只是我们的大脑要进行多层次的理解，而第一步就是辨认出那些括号。先要辨认所有的括号，然后再把它们一层层捋清楚。因此你要形成个大脑符号辨识机制来做这件多余的事情。

到了现在，也许Lisp里面那些算术符号已经可以用通用的名字写出来，像加号乘号什么的直接写即可。

Seibel：的确。

Deutsch：好。那么接下来我要说的第二件事其实就不用做了。就是用比标记识别能力而不是符号识别能力去弄清楚那些标记到底是怎么回事。这个也是大脑中的高级活动。

还有件看起来很小的事，就是在中缀式语言里，操作符的两端是操作对象，而前缀语言里则不是——我认为这并不是一件小事。因为这样一来，看后面的那个操作对象就更费力。

其实这三件事听起来都没什么，但是我觉得关系到信息密度的事情就很重要。

Seibel：但实际上Lisp的基本语法，即词汇语法，跟程序的抽象语法树十分相近，因此语言是可以支持宏的。而且宏命令可以用来创建抽象语法，这可是压缩内容的最好方法啊。

Deutsch：是。

Seibel：在我写的Lisp书里面有一章是讲二进制文件的语法分析的，用了MP3文件里的ID3标签作为例子。Lisp的编程风格可以让你把ID3文档中的细节拿出来用括弧括起来，然后将这一段直接作为代码使用。

Deutsch：也没错。

Seibel：所以在对ID3头信息的分析进行表述时，我用的标记数正好就是文档中所标记出来的细节的数目。

Deutsch：但是，Python也能做到这一点。有一次我也需要分析一个相当复杂的文件格式——是一种复杂的音乐文件格式。我用Python写了一组类，分别用来分析和打印，效果很好。

类结构和方法名的对应都在一个通用父类里面实现，全是面向对象的，甚至都不需要宏功能。可能看起来没有其他语言的实现方法好看，但是得到的结果却有非常好的可读性，比较起来应该和Lisp宏指令相当。有些事情的确可以用Lisp来实现得更清晰更通用，我不否认这一点。

转过来看看Ghostscript，Ghostscript是用C写的，但是其实它还用了成百个预处理程序的宏命令。结果就是，为Ghostscript写代码，你要学的不仅仅是C，事实上学了一门扩展语言。你可以在需要的时候这样扩展C，也可以在需要的时候扩展其他语言。

我因此自己也扩展了Python，不是语法上的扩充，而是类——Mixin类，用来增加语言的语义。在Python中有一组专用工具来做这些事情，在Lisp中也有。有人喜欢前者，有人喜欢后者。

Seibel：*是什么让你从编程转行到作曲的？*

Deutsch：我在Ghostscript上耗得差不多了。从1986年开始，我对技术的主要兴趣就集中在它上面了，到了92年还是93年，Ghostscript就是我唯一的主打项目了。这样一直到1998年，我就感到耗不起了，一直以来，只有我一个人，不仅做技术，还加上客户支持和管理工作。基本上是一个人的事业，消耗太大。然后我雇了一个人，成立了公司，让他去招其他的工程师。

又过了两年，才找到可以代替我的人，其后再两年，我才把手里的事情全部都交出去。到了2002年，我觉得够了，不想再插手Ghostscript的任何事情。

然后我告诉自己，用六个月来放松，去找下一个目标。那时我55岁，觉得自己还不特别老，我觉得如果我还想继续，我还能集中精力再做一个项目，所以我开始想做什么好。

后来我在施乐认识的老朋友J. Strother Moore II的一个项目引起了我的兴趣，他是——或者曾是——得克萨斯大学奥斯汀分校的计算机系主任。他最大的成就就是和SRI的一个叫Bob Boyer的人合作了一个非常好的定理证明程序。用这个程序建立了一系列的软件，包括一个巨大的定理和引理的库，囊括了很多实用领域。

这样一个充满活力的小组一直在做定理证明器，我对这一块一直都有兴趣，这正好也是我的博士研究课题。他们把程序做进了一块AMD CPU的算术单元。我觉得这一群人很不错，也在做着我感兴趣的东西，小组的领导是一个我很了解并且感觉还可以的人，而且他们做的东西是以Lisp为基础的。我觉得这个地方很适合我。

于是我就去他们那里做了个报告，关于定理证明能否加强Ghostscript的可靠性。那时我还有Ghostscript臭虫追踪系统的历史记录，于是我随机挑了20个程序臭虫，一个一个看，看要让定理证明程序对解决这些问题有帮助，还需要做些什么，添点什么。

然后我得出来的结论是，定理证明的技术并不能起多大作用。有些地方虽然是可以利用定理证明程序来改进，但是都需要费相当大的力气才能让这些技术派上用场。

所以我觉得，定理证明技术在增加软件可靠性的方面，实用性并不高，因为那些属性的形式化实在是太困难了。

于是我做完报告，反响还不错。我跟一些研究生谈了谈，还有J.，然后我就走了。我就在想："方方面面似乎都不错，但我就是对此不再有激情了。"

然后我就有点迷惘的感觉，在一个合唱团里表演了几年。2003年的暑假，合唱团去了意大利的几个老教堂，举办了6场表演。我的搭档和我都去了，我们决定在演出结束后在欧洲多逗留两三个星期。

我们去了维也纳旅游。霍夫堡宫的一部分被分成了10个专门的博物馆，都很小。我在旅游指南上发现了一个古乐器博物馆。

我去了之后发现只沿着一个长长的大厅有一些展室，天花板很高。最老的乐器我不能确定是否可以追溯到新石器时代，但是的确都很古老，然后依次陈列出来，大多数都是最近几百年西欧的产物。我没有逛完。我最喜欢最后一两个展室，我看到了莫扎特用过的钢琴，勃拉姆斯用来练习的钢琴，还有海顿曾经放在家里的钢琴。

那一刻我顿悟了。我一直苦苦寻觅下一个可以让我投入的项目，真是徒寻烦恼，问题不是找什么项目，而是我对软件开发已经热情不在了。你现在可能觉得我的想法很离谱。我投身软件行业很大的动力是，我觉得软件可以让世界更美好，但我现在已经不这么想了。真的。至少方式上发生了变化。

于是，当时我就灵光一闪，想到了一种全新的方式，不是让世界更美好的方式，而是为世界贡献一些更久远的东西的方式，就是从事音乐创作。就是在那一刻，我毅然决定离开我已经坚守了50年的事业。

Seibel：*但是你还是会做一些编程。*

Deutsch：忍不住，忍不住用程序来做我想做的事，至少这还是个有趣的解决问题的途径。我断断续续做了一些小的软件项目，其中有两个是持续了几

年的。

一个是我的邮件服务器的垃圾邮件过滤器。虽然不是特别好玩，但我还是有一点兴趣。我时不时会看看服务器日志，这个过滤器确实在起作用，挑出了百分之八十到九十的垃圾邮件——过滤的准确性取决于过滤器和邮件哪个更聪明。

另外一个我一直在研究的程序就是乐谱软件了。因为我对市面上的类似软件做过调查，我还在朋友家用过几次Finale——很不好用，是那种没法用语言形容的烂。我有一份Sibelius，为此还专门买了一台Mac笔记本。它的用户界面比较烂，如果键盘上没有数字锁定键（Num Lock），整个界面几乎就是废的，而Mac的笔记本就没这个键。用户界面还有些其他令我不爽的。所以我决定自己开发一个。

我一共做了四个架构设计，最后挑了一个我觉得最好的。这个过程是个很有趣的学习过程。是个交互式程序，很复杂很庞大，系统接口总是出问题。

经历四个架构后，我挑中的那个方案是用方程式编程来解决程序渲染问题的，这是其中最难的部分。用方程式定义变量值，然后用具体实现去计算它们，然后我发现用Python来实现这个功能不是很难，我知道有人用Python做过两次类似的东西。我喜欢用Python来解决是因为它的样板文件数量最少。

所以，是的，我还在做一些必要的编程工作，也觉得它还有点意思。但我只是为自己做，而且有时一连中断几周都没事儿。当我是职业程序员时，我总希望处于项目的核心位置，但是现在的我只喜欢待在几首曲子中间。

Seibel：*你之前提到想用软件改善这个世界，你是怎么想的呢？*

Deutsch：其中有一部分跟软件无关，只是看到别人做的事情总没做好，让我上火，我觉得我可以做得更好。小孩子心性罢了。现在想来像一场梦。

从我开始编程，一直到80年代，电脑技术总是跟大公司有关，然而我自己的个人信条是有点“反大公司”的。现在看我以前做的工作，一般叫做个人计算、交互式计算。我想我的一部分动力是，如果我可以让计算机的能力被更多的个人掌握，那么对那些大公司文化会是一种打击。

我也从来没有想过因特网的发展如此迅猛。我也没想过因特网的力量可以对大公司产生如此巨大的影响，我想过因特网从本质上是不可被操控

的——但是我现在也不这么想了，还是有人可以有效控制它的。

我觉得如果微软打对了牌，他们是有机会统治因特网的。我想他们也很乐意这么做，我也觉得他们足够聪明，能够找到正确的路径，最终控制因特网上的所有软件。

所以我对计算技术的未来不是很乐观。实话说，这也是我能轻松地全身而退的原因之一。我看到的是一个被不道德的垄断者统治的世界，在那里我找不到自己的位置。

（编辑：武卫东）

12

Ken Thompson

董金乾　译

Ken Thompson属于传统意义上的Unix黑客，长须连鬓，痴于编程。纵观他的职业生涯，他从事过各式各样自己感兴趣的工作，从模拟计算电路、系统软件，到正则表达式，再到国际象棋软件，涉猎很广。

最初，他以一名研究员的身份受聘于贝尔实验室，加入到MULTICS操作系统项目中。在贝尔实验室撤出该项目后，Thompson仍坚持研发，终于与Dennis Ritchie一同创造了Unix。这期间他曾担心会因不务正业而被解雇。在开发Unix的过程中，他还发明了B语言，这可是Dennis Ritchie C语言的前身。

后来，他着迷于用计算机下国际象棋，造出了世界上第一台专用的国际象棋计算机Belle，它是当时最厉害的电脑棋手。他也顺便收集齐了全部4子和5子残局，补充进国际象棋残局库。

在贝尔实验室研发Plan 9操作系统期间，他又设计出了目前广泛使用的UTF-8 Unicode编码。

1983年，Thompson和Ritchie凭借“对通用操作系统理论的发展的贡献，尤其是实现了Unix操作系统”而荣获图灵奖。同样由于在Unix开发中的贡献，他还获得过美国国家技术奖章和美国电气与电子工程师学会（IEEE）颁发的金井务奖（Tsutomu Kanai奖）。

在这次访谈中，Thompson谈及了自己早年对电子技术的热衷，还有身为学生即已开始授课的传奇学术经历，以及他缘何不齿当下某些编程方式。

Seibel：*你最初是如何学会编程的呢？*

Thompson：从小到大，我一直痴迷于逻辑运算。也因为这份痴迷，还在上小学的时候，我就已经开始琢磨起二进制算术之类的问题了。

Seibel：*你那时就取得成就了？*

Thompson：哪里哪里，没有这么简单的！不过，小学那会儿，我还真想出了一个实现不同进制加法的算法，并从中明白了什么是进位，以及数字的每一位都表示什么等一系列问题。按照那个算法，我做了个十进制的小计算器，有点儿像个十进制的算盘。我并没用1和2来表示数字，而是采用了一枚可以表示0到9的滑块，还有表示从高位借位的减1和由低位进位的加1。我还得借助根铁丝来表示二进制的高位，比如4，当需要进位或借位时，就把铁丝支在相应的地方。最终，我先根据以上这些完成了二进制计算器，并随后扩展到了其他进制。

Seibel：*你怎么会想起要做二进制算术呢？*

Thompson：我刚开始设计计算器那会儿，课堂里正好教了二进制。

Seibel：*你算是那场“新式数学”教学运动的受害者？*[①]

Thompson：谈不上，我只算得上是个糟糕数学基础的受害者。小的时候，我们家差不多每年都得搬次家，结果我呆过很多学校，有的糟糕透顶，有的还算不错。这种情况下，我实际上是学一年休一年，别人两年的课程我必须用一年时间学完。这样三天打鱼两天晒网，害得我的小学数学教育十分糟糕。但巧的是，我上过的这节课恰好就讲到了二进制算术。我学了二进制，再扩展到任意进制，自得其乐地琢磨这其中的奥妙。我就算这么起

---

① 20世纪60年代，前苏联率先发射了卫星，美国朝野深受震动，随即在中小学开展了数学教育改革，力图培养更多理工人才。——编者注

步的吧。

Seibel：这都是你还在读小学时的事？

Thompson：没错，七年级。后来，好像是高中最后一年的时候，我又对电子技术着了迷，组装些收音机、扩音器、振荡器和铁耳明式电子乐器。我沉醉于模拟计算电路的世界而流连忘返，那真的是太神奇了。那段时间我所有的激情全都倾注在了电子技术上。再后来，我进入加州大学伯克利分校主修电气工程，在那里，我生平第一次见识到了真正的数字计算机。

Seibel：我想知道这是哪一年的事？

Thompson：我入学早，那时的初级课程一共是3个学期。我是1960年9月入学的，推算起来应该是在1962年的春天或秋天。学校当时有一台模拟计算机，我曾经在上面玩得很开心。另外还有一台G15磁鼓计算机。因为学校设有相关的实验课，所以机器就开放了。任何人都可以操作，但实际上没人去动，结果反倒闲置了。这样一来，整个机器实际上成了我的专用机。我在那机器上一个人摸索着写程序，一点一点扩展那台模拟计算机。模拟计算本身几乎都是在做扩展的工作。

Seibel：能在哪些方面扩展呢？

Thompson：时域上和幅度上。简单地说，你用它完成某种功能，这就是扩展。你先给一些输入，然后得到函数的输出，再根据反馈不断去钻研。整个过程中的每一步你都不能让幅值太高，否则就会搞砸。

此外，还可以在时域上扩展。你可以将许多地方的频率减半或加倍，这么做时，一系列线性扩展也随之变化。因此，如果你的工作很简单，根本用不着扩展的话，模拟信号已经足够了。可一旦需要扩展，一切将会变得异常复杂。所以，我改用数字计算机程序来扩展模拟计算机，这样一来不是计算实际波形，而是要计算出每一点波形上的幅值和频率。无论你在进行什么操作，你都能随时知道值域是否超出了允许的范围。

Seibel：这些数字计算机的程序是用什么语言编写的，汇编还是Fortran？

Thompson：主要是汇编语言。当时也有一种解释型语言，只不过太慢了，于是不得不用汇编语言，这反倒使我更清楚计算机的本质了。

Seibel：加载编写好的程序再按下运行按钮，在等待结果的间隙，你还可以去干点儿别的。是用穿孔卡片来输入的吗？

Thompson：不是，用的是电传打字机，就像是使用纸带的Teletype打字机。代码就记录在那些纸带上，我们用电传打字机来输入计算机。

Seibel：实验室里有人教汇编语言吗？

Thompson：这倒没有。

Seibel：这之后，你再次接触到编程时，又是怎样的？

Thompson：G15机器上有个叫Intercom 501的解释器，电气工程课要求大家用它来编程。我的一个研究生朋友，在全校用的那台IBM大型机上写了Intercom的解释器。我那时正好有这个解释器的源代码，在一个节假日期间，忘了是圣诞节还是别的什么节日，我一边阅读代码一边琢磨。起初虽不知道它是用什么语言写的（后来才知道是NELIAC语言），却已经对它啧啧称奇了。我从中学习编程，学习NELIAC、Intercom，以及如何用解释的方式来写程序。整个假期，我几乎都在研读。一个星期的假期结束后，我向他提出了一大堆问题，反复叨叨程序中的种种小错误等等。之后，我明白了如何编程，而且还干得相当不错。后来，我干上了编程的工作。

基本上，我在学校就一直从事着一些工作，我参与了助学项目，后来也接些零活。我当过研究助理，协助研究生完成论文中涉及的编程任务。我还做过助教，在计算机中心编程。在计算机中心时，我经常就是坐在一张小小的办公桌前，等着不断有人进来说："我只不过改了一处代码，结果……"我往往答道："是吗，那咱们一起来看看这处代码，查查到底出了什么事。"

Seibel：你是怎么看待这些经历的，是它们锻炼出了你的调试能力，还是说这只不过是一些愚蠢的琐事？

Thompson：还是锻炼了某一方面的调试能力，在经过这类工作之后，你必定会深刻理解常见的错误。经常遇到有人花很多天搞不定的程序，我一眼便能发现症结："喏，就错在那儿！"

Seibel：你毕业拿的是电气工程学位，对吧？那时学校有计算机科学这一学位吗？

Thompson：没有，在当时的美国，计算机科学刚刚萌芽，大致有两个方向：要么在理论上，通过数学发展出来；要么在实践上，通过电气工程发展起来。当时的伯克利，计算机科学几乎完全从属于电气工程系。数学系虽也有些尝试，但是他们还不够精，斗不过电气工程系那些头发花白的老头儿们。

Seibel：显而易见，伯克利在理论方面的贡献，远不如它在系统建造方面声名遐迩，比如伯克利系统实验室。

Thompson：是的，的确是这样的。那时正是计算机科学崛起的时代，要么产生一个理论型的计算机科学系，如在康奈尔大学；要么诞生一个伯克利式的计算机科学系。这是由学校的土壤决定的。所以我花了一年时间在那里的研究生院学习，倒不是因为我有什么雄心抱负，只不过当时我没有别的事情可做，并且我在那里过得很愉快。

Seibel：大学毕业后直接上的研究生？

Thompson：是的。实话实说，我先是留在大学工作，根本没去申请研究生，实际上是一位教授帮我申请的，然后告诉我被录取了。

Seibel：仍然是电气工程专业？

Thompson：是的。大四和研究生时，我过得都特别爽。不用做任何自己不愿做的事，也没什么要求，一点儿都没有。为了毕业，有个夏季小学期我选修了美国历史之类的课，只为了拿到学位。除此之外，大四和研究生阶段，我还在我选修的一半课程里当授课老师。

基础的计算理论应运而生。希尔排序出现了，却没有人能弄清楚为什么它会比N^2排序快。每个人都做测试，试图找到答案——这种排序初看极其简单，但是没有人知道为什么它如此之快。他们采取渐近的方式去弄清楚为什么它是n的1.3次方之类的东西。它绝不是一个自然数。从探讨希尔排序、希尔排序的智慧魅力以及它速度奇快的原因，产生了计算速度分级，还产生了最早的$n \log n$算法和分而治之算法等，彻底改变了人们的观念。那真是一个令人兴奋不已的年代。

我有一大群朋友，很多都是年轻教授。有一位数学系教授和我关系相当好，一位电气工程系教授和我也不错，还有那个我替他编程的研究生，还有其他人。他们会为我开设一门课程，然后我就教这门课程。

Seibel：你正式教这门课，还是仅仅挂名？

Thompson：不，不，绝不仅仅是挂名。课程编号是EE199，表示是个人研究课程或集体研究课程等。他们会开设一门课程并且起个名字，然后就交给了我。一般会有三四个学生来听。

Seibel：你就正式名列教员之一。

Thompson：是的。

Seibel：你喜欢教学吗?

Thompson：从某种程度上说是的。我曾回去教过两次书：1975年～1976年我休了一年假回伯克利教书；1998年又在悉尼教书。很有意思，我真的非常喜欢。我那时实验室做研究，回伯克利讲课，也算从头学习这些我教的课程，因为我从未接受过计算机科学教育。通常一个访问老师只教一门课，而我却教五门课程。有些课我教过两次，这样是最理想的，因为第一遍我算是学习；当我第二次教时，便知道要讲些什么，就可以提前准备，有条理地组织授课内容。第三遍却会让人厌倦。我曾教过某一门课三遍，结果真的很差劲。所以，我是当不了老师的，因为那要求你最终要一遍一遍又一遍地重复。我绝对受不了的。不过，我倒是挺喜欢这么讲课的：在第一遍需要努力备课，第二遍时享受乐趣。但，第三遍就简直是苦难了。

Seibel：你写的第一个有趣的程序是什么?

Thompson：我写的第一个长点的程序是用于求解“伤脑筋十二块”(pentaminos) 问题的。你听说过“伤脑筋十二块”吗?

Seibel：是种骨牌排列游戏，对不对?

Thompson：对，就是骨牌排列游戏，一共有十二块不同形状的骨牌，每块均由五个全等的正方格组成。 我曾在物理系的一台IBM 1620上运行这款游戏。因为我很清楚学校里所有那些鲜为人知的电脑都在哪，晚上正好用它们来干我的事。另外，我在计算机中心的不同主机上还有差不多20来个账户。

Seibel：有些类似俄罗斯方块。

Thompson：差不多，但“伤脑筋十二块”中的每一块都是由五个方格组成的，所以又称五格骨牌。当把所有这些骨牌都拿出来放在板上时，会有两种布局非常吸引人，不知道这么说过不过份。一种是10 × 6的；另一种是8 × 8，中间有个2 × 2的孔。我就想方设法求解骨牌在这两种板上所有可能的布局。当时大致是这样做的，先挑一种板的布局方式，再考虑相应的骨牌布局，这样只要保证骨牌能匹配板的布局就可以了。根本不必考虑是不是在解“伤脑筋十二块”。

Seibel：这差不多算是穷举搜索?

Thompson：是穷举搜索。

Seibel：*那它也是用汇编写的吗？*

Thompson：我想想看。嗯，很可能是汇编。不过，记不太清了。

Seibel：*在这期间，想必你也学了Fortran语言。*

Thompson：是的，是这样的，我要在计算机中心教Fortran语言并调试Fortran程序。可我从来没用Fortran编过程。我为早期的Unix写过一个Fortran编译器，B语言可以算做Fortran编译器的一个尝试，不过后来放弃了。

Seibel：*我还以为，B语言是你对BCPL的翻译。*

Thompson：这么说也有那么点意思。我最初想做成什么样子我也说不上来，语义上，它就和BCPL是一样的。当我开始时，是想做成Fortran的。但后来，我用BCPL完成了最初描述。我喜欢上了它清晰的语义，也是在那个时候我放弃了Fortran，它最后具有了C的语法和BCPL的语义。

Seibel：*从你学会编程那天开始直到现在，对于如何认识编程与如何实践编程，你觉得有什么大的不同？你是否觉得你编程功力从某种程度上已成熟，或是你已非常擅长编程，或是你后来学会的东西会让你回头忍不住感叹："哎呀，天哪，真不知道那时都在做些什么？"*

Thompson：不，并非如此。有时候，我回头看曾经做过的东西时会说："哇，当时的我可比现在聪明多了！"从我花一个星期阅读的那个程序到大概我30岁、35岁左右，我能牢牢记住我曾经写过的每行代码。我会在白天写程序，晚上坐下来逐行检查寻找漏洞。第二天我会再去看一下，还是会发现有错误。

Seibel：*你认为，你35岁时还能记得10年前写的东西？*

Thompson：是的。不过再往后，我就是有选择地记忆了。

Seibel：*在学习编程上，有什么你想要做得不同以往？关于走过的道路你有什么遗憾，或有什么事你希望应该早点儿做？*

Thompson：哦，当然，当然有。我希望高中时就能学好打字。直到今天，我还苦于打字不好，但谁想得到呢。我以前并没计划着要做任何事，也什么都没做。我这个人没什么计划。我只做我下一步能想到的事，一直都是如此。如果我能有先见之明，或者计划什么的，像打字这些事情，再给我机会我一定会去做的。我也会去深入学习一些数学，因为我就经常遇到需要数学才能帮得上忙的事情。所以，确有这样的事情想要弥补。但是，如果真让我回到

从前再活一遍的话，我敢保证不会有什么不一样的。基本上我还是不做计划，还是只做下一件事情。就算给我机会重来，我也只是照例如此。

Seibel：毕业后你受聘直接去了贝尔实验室，这是怎么回事？听起来你那时不像是一个典型的学术研究者。

Thompson：我只是随波逐流，那时的状况一言难尽。我当然不是诚心实意地要呆在学校里。从表面上说，是的，我确是身在学校。有一个教授，也是我一个非常好的朋友，唆使贝尔实验室招聘人员来找我。但我当时没在找工作。事实上，我一点没有雄心壮志，一点都没有。他约我好几次去他的小招聘展台见面，我要么是睡过点了，要么是告诉他我不感兴趣，反正是没去。他就一直约我。有一回他打电话给我，说想过来看看我。于是，他来到我的公寓，并希望我去贝尔实验室面试。我告诉他不去。他说："就当是一次免费旅行。你去了干什么都行。"我说："嗯，直率地说吧，我对工作不感兴趣。不过有一次免费的旅行，我倒很高兴，因为我在东海岸有不少朋友。可以去拜访一下。"他说："好的。"于是就有了我的面试。我去贝尔实验室呆了两天，然后租了一辆车，在东海岸四处乱跑，访问我高中的朋友，他们到处都是。

Seibel：显然贝尔实验室的人看到你有某种特质，说："必须得把这小子弄进我们的实验室。"

Thompson：他们是怎么看我的，我不清楚。从我这边看，这些人就是我当时上的课或教的课中所学论文的作者，他们的名声如雷贯耳，仍然在做很有趣的事情。对我来说，工作就是工作，然而这些人不是在工作。他们过得很愉快，就像在学校里似的。

Seibel：那么，刚加入的时候你都做了些什么？

Thompson：贝尔实验室正在做MULTICS项目，我是被招聘去做MULTICS项目的。我听从了安排。我摆弄那些机器，启动MULTICS，从事我所负责的那一小部分。后来，贝尔实验室发觉MULTICS并不适合他们，就退出了这一项目。

但他们拥有不少MULTICS机器，这都是专用的机器，因此只能闲置在那里，等着有人来用车拉走。大约有一年时间，我都还可以继续使用这种巨大的机器。我们中可能只有两三个人还在使用这种机器。于是我开始做操作系统方面的东西，试图让一个微型操作系统启动和运行。

困难是不可想象的，因为它是一个真正复杂的计算机。但是我做到可以让它在楼里的50台Teletype电传打字机上显示Hello。之后它们被搬走了。我又四处找出其他一些没人用的机器，最终在这些非常非常小的PDP机上发明了Unix。

Seibel：你有时间干这些吗？是因为你的老板知道你在做，并说这是个很好的研究项目，还是因为你正好在项目的间隙？

Thompson：没有，坦率地说，我有些积习难改。我觉得我最终会被解雇，但也没觉得怎么样。我们应该做基础研究，但有一些我们可以做，有一些却是不该做的。由于MULTICS失败的阴霾，操作系统恰恰成了当时不该做的基础研究项目。因为我们尝试过，结果却以失败告终，彻底的失败，代价沉重，只能放弃了。所以，我总感觉，因为我还在做操作系统，最终可能会被解雇。但结果却没有。

Seibel：你如何设计软件？你是在方格纸上画草图，还是启动一个UML工具，或者就是直接开始编码？

Thompson：这得看程序的规模。在大多数情况下，想法会在我的脑子中形成，根本不必写在纸上。我会集中精力思考困难的那一部分，简单的部分会自行消失，我会写下来，当你想好了的时候，它们会直接从你的指尖流淌出来。困难的那部分，我会坐下来思考一段时间，让它逐渐萌芽，也许要一个月时间。时机一到，点点滴滴开始汇集在底部，然后自底向上构建出金字塔来。当金字塔在我的脑海里建得足够高的时候，我便从底部开始写起。

Seibel：但你不单单只构建枝叶，你知道它们的整体结构。

Thompson：假设有人给我形容说“这里有一台电脑和它的操作码”，我可以根据操作码勾画出程序的结构，判断程序效率的高低，因为我能看出程序的底层，想象出其层次结构。对于程序，我也可以看出类似的东西。如果有人给我看库例程或基本的底层东西，我就可以看出可以怎样构建出不同的程序以及还缺什么——尽管这些程序此刻还很难写出来。因此，我可以想象出那个金字塔，问题只是要尝试并去分解它，得出底层部件。

现代编程在许多方面使我恐慌，人们只是构建一层一层又一层，除了翻译什么也不做。使我困惑的是，你必须自上而下阅读一个程序。它说“做某事”，于是你去找这个“某事”；找到后它又说“做某某事”，你再去找这个“某某事”；而后它又说“做某某某事”，接着却可能返回到最开始。其实什

么也没做，只是将问题降至越来越深的层次。我无法在脑中构想这些，我无法理解这种方式。

Seibel：*那么为什么不仍自下而上阅读呢？那些枝叶先不管它。*

Thompson：是这样的，你不知道哪些是叶子，哪些不是。如果文档描述得很好，你可以阅读英文就能理解，根本不用去读代码。但如果实际上只给你大量的代码，告诉你说："阅读它，努力改善它或让它做些别的事情。"这时，我通常自上而下阅读。

Seibel：*你会先写下什么，然后再开始编写代码吗？*

Thompson：我通常先写下数据结构，再写代码。我不写算法，没有流程图之类的东西。但会写几乎每行代码都得参考的数据结构。

Seibel：*如果你正在编写一个C程序，这是否意味着，你会用C代码来定义这些数据结构？*

Thompson：不会，就用带箭头和其他东西的图框。

Seibel：*这样，你有了这个蓝图，金字塔。一旦开始编码，你会多大程度上遵照该计划？*

Thompson：对于代码编写，我并不是一成不变的。如果我中途发现了一种不同的程序结构划分，我会研究并采用的。我知道很多人，写下代码后终生不改，除非有错误。特别是如果他们编写一个带API的例程，却只是草草地把API写在个信封上或API列表中——那就这样吧，从来没人会去修改的，不管这有多么糟糕。但我会一直心甘情愿地琢磨代码，一旦发现另一种更好的方法，或者不同的代码组织方式，我都会去修改它。对现有的代码，我从不迷恋。本质上，代码是易腐烂的，注定是要被修改的。即使没有任何修改，总会因为什么，而最终腐烂掉的。

Seibel：*你是怎么来决定何时扔掉腐烂的代码？*

Thompson：当它难于修改时。我比大多数人扔得快。当我想向代码中增加东西时，一旦觉得很难就立即扔掉。我会把它扔掉重做，使用不同的程序结构来让它能容易做任何我想做的事。让我扔掉旧代码真的很容易。

Seibel：*这要是别人的代码，你也会这样处理吗？*

Thompson：这得看我有没有权利。如果我有这个权利，当然会的，我没有

什么顾忌。如果我无权，由于这是别人的代码，我就得受苦了。或者干脆就别做。

Seibel：*在你继承别人代码的情况下，改写就会有这样的危险：也许你忽略了某些微妙之处，或忽视了某些功能，都将导致其无法工作。这些问题，你可曾遭受过？*

Thompson：嗯，肯定会的，但这只不过是调试代码时的必经之路。如果你忘记了某些功能，或根本就没做，当你意识到时就得去做。这正是调试的一部分。你第一次编写程序时肯定是不完整的，你得扩展它。

Seibel：*一旦完成整个系统，你会回过头以某种方式写文档吗？*

Thompson：这取决于写文档的目的。如果是我自己看，我一定不会写的；如果怕以后忘记参数，我会加上一行使用说明。我还会在文件头加上注释，概述整个程序的用途，但一定会非常非常简短。如果它是系统或函数库的一部分，亦或是要发布的，我才会花时间写文档。除此之外，根本不会去写。

同编程一样，写文档也是门艺术。我认可的好文档，极难一见。通常的文档都远远超出所需的细度，其中又包含太多不相干的内容，还有数不胜数却无从查阅的引用。要想写出好的文档，是非常非常难的，同时也非常耗时。要想写出正确的文档，就更得像编程一样：首先划分，再漂亮地重新组合在一起，发现有错的话还必须得重写。没人愿意这么做的。

此外，我更喜欢自下而上组织的文档，不过，文档通常可不是这样写出来的。如果某一程序依赖其他程序、文件或数据结构，我希望能明确看到对于它们的引用，这样，我可以跳转到那里，不过，那里用不着再引用回来的。

Seibel：*所以你希望能自下向上来理解代码，就如同编写代码那样的方式？*

Thompson：是的。这就是我的大脑处理它并能记得住的方式。否则，可能刚读过时还记着，但过后马上就忘了。如果我能理解它的结构，那么它便成为我记忆的一部分，然后我就能理解整个程序。

Seibel：*在图灵奖获奖感言中，你提到如果Dan Bobrow被迫使用PDP-11，而不是更强大的PDP-10的话，那天站在那里领奖的就是他，而不是你和Dennis Ritchie。*

Thompson：我只是想说获奖非常偶然。

Seibel：*你认为你们得益于功能不够强大的机器的制约？*

Thompson：当然它也有好处，因为它小且有效。但我认为我们更多地得益于自己所编写的代码。不能不提的是，事实上，那时恰恰处于小型机革命的风口浪尖。我母校计算机中心运行的大型主机便是PDP-10。而我们真正的意外发现却是，计算技术正从集中走向自治。这一切都体现在PDP-11上。

Seibel：难道Unix没有受益于是用C语言编写的，而其他的操作系统，如TENEX和ITS，都是用汇编写的，难以像Unix那样非常容易地移植到别的硬件平台之上？

Thompson：当时也有优秀的系统编程语言可以写出操作系统。

Seibel：比如……？

Thompson：NELIAC是Algol 58的用于系统编程的版本。

Seibel：Bliss语言也是那个时代发明的？

Thompson：我记得Bliss语言是在此之后发明的，并且它更多的是关注优化编译。我认为从一开始就必须非常清楚，你不应该为了优化编译而自断后路。你应该做得不错，但不必真的非常优秀。为了从良好到优秀而费尽周折之际，摩尔定律早已把你甩在了身后。你可以优化出10个百分点，但是正当你为此奋斗之时，计算机已经变得比之前快了两倍，可能还出现一些其他因素比起优化更为关键，比如缓存。我认为一开始就想做得特别好根本就是在浪费时间。因为这真的很难，并且你一边修复bug一边还会引入新的bug。应该见好就收，而不是再多用百分之一百的时间去完成那百分之十的工作。

Seibel：你大概听说过Richard Gabriel写的《更糟的即是更好的》这篇文章。

Thompson：还真没听说过。

Seibel：他对比了两种风格——他所谓的MIT风格和新泽西（贝尔实验室所在地）风格。MIT风格认为正确性是压倒一切的王牌，而新泽西风格则高度重视实现上的简洁性。他主张新泽西风格，又被称为“更糟的即是更好的”，使人们可以先做出东西来运行，并在之后逐渐改善。

Thompson：我认为MIT一直有Unix自卑感。我在那里做过Unix的演讲，我记得我是经Michael Dertouzos介绍的。他详细阐述了为什么Unix没能在MIT编写出来，以及为什么本应能在MIT做成的。为什么他们有机会，有人，有

所有必备的条件，但却没能做出来。我才恍然大悟，原来他们头脑中一直在想着竞争。而我脑子中没有这一点。我们做出了Unix，而他们做了MULTICS，却是个怪物。这明显就是第二系统综合征。

Seibel：你说的是，MULTICS是MIT继CTS系统之后的是第二个系统？

Thompson：是的。因此，过度设计，过度建造，什么都过度，结果近乎不可用。他们还称这是一个巨大的成功，但很明显根本不是。

Seibel：我的理解是，一帮MIT的黑客也是这么看待MULTICS的。他们偏爱ITS和用Lisp语言编写的系统。在MULTICS之后似乎还真有这么一个分水岭。一边是Unix诞生了，正如你所深知的，另一边是MIT这些爱好Lisp的家伙，在PDP-10上编写了基于Lisp的操作系统，最终产生了Lisp的机器。

Thompson：是的，是的。所有那些家伙我都认识。我当时就认为这是一项疯狂的工作。我认为Lisp不是一门太独特的语言，没必要再做Lisp机器。我认为事实证明我是正确的。我一直都对他们说："你们疯了。"PDP-11已经是一个伟大的Lisp机器，PDP-10也是一个伟大的Lisp机器。没必要再设计一个不能更快的Lisp机器。根本没有任何理由。这真是愚蠢的做法。

Seibel：是否有你喜欢的MULTICS的功能，但却没能在Unix上实现？

Thompson：我足够喜欢并从MULTICS中汲取过来的，是分层的文件系统和shell（一个单独的进程），你可以用其他的进程替换掉。在此之前，所有系统都有某种"执行语言"（executive）——我们一般这么叫，是一种内置的处理语言。按进程执行。每次，当你输入到shell，它会创建一个新的进程，运行你键入的任何东西，当这个进程死后你再回到shell来，避免与刚才运行的内容有亲密的接触。

Seibel：这些就是你汲取的东西，没落下什么让你现在后悔的吗？

Thompson：没有。

Seibel：从我了解到的Unix的历史来看，好像你就用了前面所描述的设计过程。你思考了一段时间，后来你的妻子和孩子离开了一个月，你说："哦，太好了，现在我可以写代码了。"

Thompson：是啊……我们一群人坐下来讨论一个文件系统，大约有三四个人，其中唯一大家并不熟知的人是Rudd Canady。当年，贝尔实验室的设施

可是一流的，甚至可以拨打电话留下文字记录。你想想看，打电话留言，说你想写下它，那么电话记录会在第二天出现在你的信箱里。当我们在黑板前谈论了一小会儿文件系统后，Canady就拿起电话，拨通一个号码，将黑板上的内容读入电话。

等黑板上的内容打印成稿时，我们发现它已几乎等同于设计文档了，除了一些稀奇古怪的同音字以外。之后，我开始实现那个文件系统，严格地说只是在PDP-7机上。一段时间过后，我觉得要做些测试，就写了一些用来生成负载的程序。但在编写程序来驱动文件系统时才遇上了麻烦，交互很不方便。

Seibel：*你只想编写文件系统？那时你并没想着要写个操作系统吗？*

Thompson：没有，就只是想做个文件系统。

Seibel：*所以你就写了个操作系统，以便有个好点的环境来测试那个文件系统。*

Thompson：是的。开发了一半的时候，我才意识到这真的算得上是个分时操作系统。一开始我写了个shell来驱动文件系统，然后又写了几个其他程序来驱动文件系统。回头一看，我说："我只想要个编辑器，结果竟做出了个操作系统。"

Seibel：*你曾跟踪调试过的最糟糕的bug是什么样的？*

Thompson：基本上都是内存破坏类的bug。现在不再发生了，我都不知道是为什么。但在早期，我们一直使用各种实验型的硬件设备，反倒有好多硬件bug。

Seibel：*内存破坏，是由硬件故障而引起的，而非人们常常提及的"野指针"？*

Thompson：可能是因为"野指针"，也可能是因为硬件，或二者兼而有之。我现在想起的一个bug，一个最糟糕的例子，发生在PDP-11上。那台PDP-11不支持乘法，虽然可以买个乘法器装进去，但这样一来，乘法器就成了I/O外设。你在上面存储分子和分母，然后开始运行。经过繁重的循环，最终得出了答案：商和余。这个程序起先是针对一个没有存储管理的PDP-11构建的，然而我们得到的第一个实验设备却是个有内存管理的PDP-11，显然这乘法程序无法在其上正常运行。

所以你得存储数据，然后做忙测试。在忙测试的某一时刻，乘法器将发出一个物理地址，而不是虚拟地址，一些内存会突然因你想除的分子出问题。

要查出它可要非常非常长的时间，而且即使查出，位置也变了。这是到目前为止我遇到的最难查的bug。

Seibel：*你是如何跟踪调试这个问题的？*

Thompson：我编过一个程序，用它来计算常数e，试图打破e的位数的世界纪录。之前的世界记录不是受限于计算，即CPU的主频，而是受限于I/O。我想出了一个新算法，它是计算密集型的，I/O变得微不足道了。它大量使用乘法和除法运算。我们注意到，只要一运行程序，整台机器就崩溃了。就这样，我们看到其中的关联。

Seibel：*也就是说，这提示你问题可能出在乘法器上？那么你最终查出根本原因了吗？*

Thompson：在某一时刻，我们发现问题出在存储乘数的地方，但重新去取的时候，却拿不到正确的数据。我们将其报告给DEC，可是DEC也没能找到原因，他们也不愿意处理这个问题。他们那帮人都不愿意处理这种复杂的设备。在那个年代，你是能得到机器的电路图的，最终我们发现这个电路图确实有个bug，就又打电话给DEC说："连接一条线和另一条线就解决了。"

Seibel：*哦，谢天谢地，当今的硬件设备基本不会像那样瘫痪的。*

Thompson：确实是的。所以说这种问题现在非常少见。而且，现在各模块彼此隔离，除非你极不守规矩才会出错。另外，以前是用汇编语言编程，极易在子程序调用过程中导致寄存器内容错误。现在使用所有参数均需严格匹配的高级语言，这些事情就变得越来越罕见。

在早期，你会发现在汇编语言里，以上这些问题很多。如果是软件，而不是软硬件组合，这种问题通常会发生在一个地方，总是这个地方被破坏。bug一定与某些东西相关。你可以坐下来，在操作系统中植入一个监控程序。隔一段时间就检查一下是否有错误发生，一旦有就尽可能快地阻止它，看看其他地方还会发生什么，一路追踪过去。这样，你就可以解决问题了。

而这一个却不好解决。直到我写这一密集的乘除法程序，才让bug频率不断提高。以前是每隔几天崩溃一次，现在每隔几分钟就崩溃一次。只有再现导致你机器崩溃的场景，你才有继续排查bug的机会。

Seibel：*现如今，会有人说："嗯，当然，汇编语言先天极易通过软件bug破*

坏内存，而C相比其他语言而言，也更易导致软件bug。”你可以让指针访问敏感的内存，或者越界访问数组，你觉得没有一点儿问题？

Thompson：没有，你可以利用语言的习语规避这些问题。有些人编写脆弱的代码，有些人编写结构良好的可靠代码，每个人的情况都不一样。我认为几乎任何语言都可能会编写出脆弱的代码。我这样定义脆弱的代码：假设你想添加一个功能，那么良好的代码，只要在某处添加了就正好合适；而脆弱的代码，你必须还要去改动10个地方。

Seibel：如果有一个安全漏洞，被证明是由于缓冲区溢出，你怎么面对有人批评说C和C++都负有一定的责任？他们说如果使用的语言检查数组边界或者进行垃圾收集，就可以避免很多这类问题。

Thompson：bug就是bug，你编写的代码中有bug，那是你造成的。如果它是一个运行时安全的语言，操作系统会直接崩溃，而不是因为缓冲区溢出而被利用来攻击。死亡侦测攻击（Ping of Death）的是操作系统中的IP协议栈。在我看来，这样的后果是会有更多的死亡侦测攻击。不会再有“接管机器成为超级用户”，而会变成死亡侦测攻击。

Seibel：但是，拒绝服务攻击与截获根权限进而控制电脑为所欲为的漏洞利用，还是有区别的。

Thompson：有两种方法得到根权限，一种是缓冲区溢出，另一种是让程序执行它不应该做的事情。大多数问题都是缘于后者，并非缓冲区溢出。你可以在没有任何缓冲区溢出的情况下获得根权限。因此，你的观点我不接受。所有你要做的就是说服超级用户给你一个shell，而没有发生任何的运行时错误。

Seibel：好。姑且不论这会导致崩溃还是被利用还是其他，有一类bug由于同样的原因发生在C和C++中，但是在Java中就不会发生。因此对于某些应用，你允许这类bug出现得到的好处难道会大于它们引起的痛苦？

Thompson：我认为这类bug实际上是少数的问题。确实，每一次我写完一个非比较子程序调用，如`strcpy`之类的东西，我就知道我写了一个bug。我会采取一个合算的决定，看是否值得为这个bug写额外的参数。现在，通常我会写出来。但是有一个语义问题，如果你截断一个字符串，并且使用了这个被截断的字符串，你就会遇到另一个问题。bug仍然在，只是没有溢出缓冲区。

Seibel：你通常使用什么工具来调试？

Thompson：大多数时候我就只打印值。我在编写程序时，会打印出大量信息，等我删掉或注释掉那些问题语句后，打印出的程序都是相当可靠的。我几乎用不着再往回查。

Seibel：你一般都打印些什么东西？

Thompson：只要是我需要的，只要是正在琢磨的。一般是不变式。但是大多数我只在进行编码的时候打印，这就是我的调试方式。我绝不会从头编写程序。我通常拿来一个程序并修改它。即使是一个很大的程序，我也会说："main函数，左，右，打印，你好。"哦，"你好"并不是我想让程序打印出来的。我首先要什么，我就将编写并且调试那一部分。我在开发程序时会每小时运行程序20次。

Seibel：你打印不变式，也使用断言检查不变式吗？

Thompson：很少。我确认自己是正确的，然后，要么注释掉打印语句，要么删掉。

Seibel：那么，为什么对你来说，将不变式打印出来要比用assert自动检查更简单呢？

Thompson：因为打印时，你可以真切看到它是什么，而不仅仅是个特定值，而且需要打印很多不是不变式的信息。那只是我的方式，我不建议将其作为一个范例。这只是我自己在一直做的。

Seibel：刚才我们谈到了如何设计软件，你描述的是自下而上的过程。你是分开建立这些自下而上的片段吗？

Thompson：有时候是。

Seibel：你编写辅助的测试程序来测试底层函数吗？

Thompson：是的，很多时候我就这么做的。这实际上取决于所编写的程序。如果该程序是从A语言到B语言的解释器，我会把A和B对应地放在一起。运行所有A，和B一一进行回归和比较。编译器或者解释器或者正则表达式搜索，都大体如此。但是还有一些程序不是这样的。这种程序我测试得不多，有些让我不知所措。我会投入一些检查，但是往往不会持久，因为它们太难维护了。多数时候只用回归测试。

Seibel：很难测试，你指的是设备驱动程序或网络协议吗？

Thompson：嗯，只要运行一个操作系统，它们就一直在运行。

Seibel：所以你认为，可以通过这种方式排查出bug？

Thompson：噢，确实。我的意思是，用操作系统来测试总好过手工测试吧？

Seibel：编程的另一个阶段是优化。有些人从一开始就优化，有些人喜欢先编写可工作的代码，然后再考虑优化。对此，你有何高见？

Thompson：一开始做时，我会尽量保持简单，一般情况下这就足够了。为一个不运行的东西构建了一个非常复杂的算法，这才是相当愚蠢的。只是在浪费时间，还会产生许多bug，甚至导致代码根本无法再维护，因为需要用50页的数学计算才能告诉别人你到底在做什么。

99%的情况下，简单的穷举就可以做得很好。如果你真想构建一个经常使用的工具并且它具有小二次的特性，那么只好精雕细琢。但是，通常不会，越简单越好。

Seibel：一些人喜欢将代码写得珠圆玉滑，完美无缺，只是想追求唯美。

Thompson：嗯，我也如此，但有些时候不得不以牺牲算法来换取代码的完美。我是说，通常复杂的算法需要复杂的代码实现。相比庞大的算法和代码，我倒是更愿意有简单的代码和简单的算法。如果非要用什么来形容一下我的代码，那就是简单、详实、短小。没有什么花哨活儿，任何人都能看懂。

Seibel：是否仍然有一些任务，为了提高性能，人们还不得不手工调优汇编代码？

Thompson：这样做的太罕见了。真的是极为罕见的，除非你真正可以提高一个数量级，其实你不能。如果你真的努力不懈，可以让大程序的一小部分运行速度提高一倍，再过一两年后就可以让整个程序运行速度提高一倍。如果你在编写编译器，肯定百分之九十九的代码只运行一两次，而其中一些代码将用于操作系统，每天要运行24小时，还有一些将是在操作系统极深的内部循环使用。因此，也许你对编译器进行的优化工作只有百分之零点一的可能会对你的用户产生影响。但是它也会产生深远的影响，也许有人还是想做的。

Seibel：但是，这更多的是通过编译器来生成更好代码时，而不必非用汇编语言编写编译器。

Thompson：噢，是的，确实如此。

Seibel：现在直接使用汇编语言写程序已经没有必要了，想必也是因为编译器已经足够好了。

Thompson：不，我认为这主要是因为机器变得更快了。编译器可真是糟糕透。你看看GCC编译器生成的那些糟糕的代码，真的很差劲，而且实在很慢。唉，天哪！我的意思是，编译器本身居然要过20多遍，实在是超慢无比。但自从GCC出现以来，计算机的速度提高了1000倍。GCC看起来更快了，只是因为它变慢的速度赶不上计算机变快的速度而已。

Seibel：说个与此有些关联的问题，你怎么看待垃圾收集？Java已最终将垃圾收集技术融入主流语言。Dennis Ritchie曾说过，C非常敌视垃圾收集。人们转向支持垃圾收集语言，这究竟是好是坏？这种技术是否应成为主流？

Thompson：我不知道。但对于这个话题，我有完全相反的两个答案。如果你是在编写操作系统，或者C语言的编译器，或者是会被很多很多人使用的东西，我认为垃圾收集多半是一个错误的选择。用它是自欺欺人，因为你完全可以自己手动去做，而且效果要好很多。用它无异于毁了你的任务、你的工作，让你的用户用起来更慢。因此我认为对于操作系统而言那是个错误。那根本不适合操作系统。但是如果你在编写一个简单程序，完成一个小任务，有了答案便将此程序扔掉，那是很棒的。它剔除了一层你并不想深究的东西，这种代价你可以接受，因为计算机是如此之快，使用它可达成一个几方共赢的局面。因此，对于此问题我真的有两种截然不同的答案。

还有个问题是有不同的垃圾收集算法，并且它们具有不同的属性，绝对迥异的属性。你若是编写一些非常通用的程序，像操作系统，如果用带有垃圾收集的语言编写它，则对于此操作系统而言，你根本没有了算法上的选择。假设你不能忍受巨大的实时差距，但有一个垃圾收集器，它运行时达到某个阈值时就得去做标记和清除。你兵马未动就已经输了三分。

因此，如果要做一些通用的任务，还不知道谁是你真正的用户，你就千万不要这样做。另外，垃圾收集严重损害缓存的一致性。而且也没有适合所有机器的垃圾收集算法。通过对缓存进行特殊处理，有的机器的速度可以提高5倍以上。算法应该随“机”应变，而不应当是现在这个样子。现在这些算法和机器没有任何关系，但是实际上缓存的一致性对于垃圾收集算法是非常重要的。

Seibel：你认为自己是一个科学家、工程师、艺术家、手艺人，还是其他？

Thompson：我不知道。我不想用科学家这个词，因为我觉得科学家是精英，并暗示具有博士学位。当你完成理科的课程时，并不会发给你一个上写“科学家”的证书。所以我不喜欢这个词，也不使用这个词。工程师，我确实取得了一个“工程师”的学位，所以我可以使用“工程师”这个词。我填写职位的时候，一般写工程师或者程序员，因为我有充分的理由。不过，大多数时候我并不考虑这些。

Seibel：*那么，暂不说你怎样称呼自己，你觉得跟哪种人最接近？是物理学家，桥梁建筑者，画家，还是木匠？*

Thompson：我做的是一种较低层的事情，我觉得是手艺人吧，但还有一些艺术技巧的活力。

Seibel：*你如何发现有天赋的程序员？*

Thompson：只看他们的激情。你问他们做过的最有趣的程序是什么，然后让他描述该程序和它的算法，等等。如果他们经不住我的盘问，那么他们就不是好的程序员。如果我挑他们的毛病，或者是发现他们的算法和解决方案有问题，而他们不能有效解释，不能比我做得更投入，那么他们也不是好的程序员。与此同时，你可以感觉到他们是否有热情。这不用直接去问，但是在谈话的过程中你会感受到这种热情有多少，这对我有极大的帮助。这就是我如何面试别人的，有人曾告诉我，这样做对来面试的人极具杀伤力。

Seibel：*能想象得到，这有点像口试。是否想过那些无法应付这种面试方式的人，有的人编程能力其实很强，只是个性问题使然？*

Thompson：没有，我认为这不是与编程无关的。我又不是在问他们经典计算机科学之类的问题，我要求他们描述他们花费心血所做的东西。我从来没有遇到过花费心血做了事情的人不能热情洋溢地讲述自己做了什么，怎么做的，为什么要这么做。我让他们自己挑选题目，不是我来挑。因此在这个题目上，我是业余的，而他们是专家。如果他们不能经受一个门外汉问他们专长的问题，那么他们本身就不专业。

Seibel：*你目前在Google做什么工作？*

Thompson：基础设施。操作系统类的东西。碎片间的黏合剂。我有权利要任何东西。目前的挑战是使许多不可靠的机器，能够像可靠的多处理机一样工作。我想这是我最想做的事情。

Seibel：*那不就是Google著名的MapReduce机制，即用无共享的信息传递，而不是共享内存？*

Thompson：是的，这是一个有着众所周知的语义和无反馈回路的过程。如果你有一个可靠的结构，你就可以用这个结构解决很多的问题。

Seibel：*你就是在这种框架下开展工作吗？*

Thompson：不，它只是试图让可靠性不依赖于每个程序员。这是一个真正严峻的挑战。这里所有的软件都有数不清的层级关系，如果这里出了问题，该如何如何，如果那里出了问题，又该如何如何。如果我出了问题，该怎样处理，该由谁杀了我，谁又启动，谁又该做什么。我猜想，50%以上的代码都是这种条件代码。

Seibel：*所以你的目标是使这一半的代码消失？*

Thompson：是的，是让它们隐藏到某个地方。它将以一种系统的方式作用于其他代码。希望能如此。这个真的很难。

Seibel：*你喜欢在Google工作嘛？*

Thompson：有一部分我非常喜欢。不过，还有一部分不好说，因为修复bug的时候还要考虑金钱，还有很多工作都涉及金钱。规模大得不可想象。好比第一天，你鼓捣一些东西，把它弄瘸了，第二天你就面临着200万的用户，你根本难以想象这样的事情。

Seibel：*你实际上是在产品部门。不像在Google实验室，那儿可能更接近于你过去在贝尔实验室的环境。*

Thompson：但是我并没有在实际的产品部门。我是在会变成产品的项目组工作。它们一旦成为产品我就不负责像保姆一样照看它们。我的岗位职责大概可以这样描述——我是否照此工作暂且不论，这是另一个问题——我的职责是想办法让生活变得更好。或者是找一些新想法取代旧事物。总之要努力使其变得更好。不论它怎么不好，是错误的，浪费时间的，还是导致bug的。如果说你发现有任何的事情在Google的架构内还可以做得更好，那么就努力去做得更好。

Seibel：*我知道Google有一个政策，每一个新员工必须先通过语言能力测试，然后才允许提交代码。这就意味着你必须通过C语言能力测试。*

Thompson：是的，不过我没有。

Seibel：你没有！那就不允许你提交代码？

Thompson：我还不能提交代码，是的。

Seibel：你只是还没有去测试，还是你不接受Google编码标准的哲学？

Thompson：我只是还没有做。我至今没有发现有做的必要。

Seibel：那么只在自己的沙箱中工作？你主要用C语言来干活吗？

Thompson：我主要是用C来编写程序，我所有正式和非正式的工作都是用C完成的。而Google采用的是C++，严格的C++。用C++编程没什么难的，但我不喜欢。我拒绝用C++。

Seibel：你和Bjarne Stroustrup在AT&T是同事。你是否对C++开发有所贡献？

Thompson：要说这个我就要惹麻烦了。

Seibel：没关系。

Thompson：C++在开发时，我曾试用了这种语言，并且给它提些意见。那里的工作氛围就是这样的。有时我们编写了一些东西，结果第二天它不能工作了，因为语言在频繁改变。有很长一段时间语言非常不稳定。终于我对自己说，不，再也不要这样了。

有一次接受采访时我就这么说的，我没有使用它，是因为它不能连续稳定两天。Stroustrup读到这个采访后，尖叫着冲进我的屋子说我的话很严重，认为我贬损他，认为我说它是一种糟糕的语言。其实我从来没有说过那是一种糟糕的语言，诸如此类的话语，从那以后我尽量避免粘上这件事。

Seibel：现在你能说说它到底是一个好的还是糟糕的语言？

Thompson：它当然有它的好处。但总的来说，我认为它是一种糟糕的语言，很多事情都是半途而废，最后成了互不相干的想法汇成的一个垃圾堆。我所认识的所有人，不论是个人还是公司，都只会各自选择一个子集使用，并且这些子集是各不相同的，所以从展现算法方面来讲，它不是一种好的语言，你不能说："我编写了个算法，你拿去用吧。"过于庞大，过于复杂，而且很显然，它是由一个委员会建立的。

为了使得这种语言被采用，Stroustrup年复一年地从事各种活动，这已经远远超出了他对语言所做的技术贡献。他在所有标准委员会里谋取一官半职，他不对任何人说不。他将每一种存在的功能都加入到这种语言中。那不是一种清澈的设计，只是把所有东西拼凑在一起。我认为该语言由此遭受了

重创。

Seibel：你认为这仅仅是因为他喜欢所有这些想法，还是说，因为要讨好每个人，提供他们想要的特性，才能让C++语言获得认可？

Thompson：我觉得后者的因素更多一些。

Seibel：看起来有很多人嚷嚷说："天哪，C++太糟糕了！"然而，大家却都还在用它编程。例如，它是Google的四大官方语言之一。如果它真那么糟糕，为什么还继续用它呢？

Thompson：我也说不清。不过，我认为在Google使用C++的人却是越来越少了。现在，不喜欢C++的人要远多于喜欢它的。

Seibel：这些人都改用Java了？

Thompson：我可说不准。好像几乎没有可替代它的。人们抱怨，但是却没有换用其他语言，被Google雇用的刚毕业的研究生都知道这个。目前很难再做其他事情，这就是它还在继续的原因：它省去了大量的教育和再教育，它可以让人们很快具备生产力。

Seibel：你是否还喜欢用其他语言，或者说，曾经喜欢用其他语言编程？

Thompson：所有有趣的语言我都曾试过。像用于解方程的Maple和Macsyma，处理字符串的SNOBOL。总之，我玩过数十种语言，只要是有趣的，我都玩过。

Seibel：是否有你中意的开发工具？

Thompson：我钟爱Yacc。我就是爱Yacc。它正好做出你想要的。与它相反的是Lex，十分可怕。它做不出任何你想要的。

Seibel：那你是用它，还是动手编写你的语法分析器？

Thompson：我自己动手编写语法分析器，简单很多。

Seibel：你尝试过文学式编程吗，像Donald Knuth倡导的那种？

Thompson：没试过，文学式编程听起来很棒，但想用在实际的编程中，却不太可能。

Seibel：为什么？

Thompson：这是用两种方式表达同一段程序，常常互不协调，甚至互相冲

突。人们对此还一筹莫展。如果能用程序语言编写得很好，就有不错的可读性，那就足够了。不必要求注释与代码亦步亦趋。注释可能只说明算法，如果代码诡异可能会以警告等形式出现。我不是一个喜欢写冗长恼人的注释的人。文学式编程只是个传说。

Seibel：*我采访Knuth时，他说，技术写作的关键，在于通过两种互补的表述方式同时解释一件事。所以，我觉得他把这当作文学式编程的一个特性，而不是bug。*

Thompson：那么如果你有两种方法，其中只有一种是真实的，即机器执行什么。另一种就不是了。只有当一种表述方式比另一种简明许多，才更值得使用。如果效果不相上下，你只需看有用的这种。如果一种更简洁，却不够精确，但可以得出你需要的东西，那也不错。但是通常情况下，你无法得到你所需的东西，你真需要了解细节，这时才要去看另一种表述。根据你想要什么决定是看前一种还是后一种。但是想细致入微地描述算法，既用编程语言还要用英语，也许Knuth可以做到，但是我做不到。

Seibel：*你有没有看过他的文学式程序？*

Thompson：只看过他早期的论文。新的没有。

Seibel：*有没有一些你认为特别重要的书籍？包括对你很重要的，也包括你会推荐其他人去阅读的。*

Thompson：我不看初级编程的书籍，所以推荐对我来说有些困难。如果我不得不学习一种新的语言什么的，我会设法找本这样的书。我更偏爱信息量大的书籍，只介绍语法和语义，而不是聊天式的，还告诉我什么是好的风格，什么是不好的。

我以前教书的时候，为了选择一本教材，我会阅读此领域的所有教材，然后做出选择。因此我这辈子有两次，我了解了这些课程的基本文献。除此之外，我不会去读书。

Seibel：*当年创造Unix时，你的计划是通过完成那四个模块来组装一个完整的操作系统。当时你的妻子和孩子不在家，留给你一个月时间集中精力编写软件。我猜想在那个月里，你有时会一下子干很长时间。为什么要这样做呢？有必要吗？还是，只是由于它很有趣？*

Thompson：这么做只是因为被这个项目驱动着，无法想象我不这么做。此

外，老婆孩子在身边时，你的生活也是遵循一天24小时的。他们走了，就没有24小时的概念了。也没有什么让我和太阳同步了。于是我一般每27～28小时才睡一次，每次睡6个小时。我只是顺其自然。每天睡到自然醒，工作状态非常好，比被孩子哭闹吵醒要强多了。

Seibel：*那是你受项目驱使，一睁眼，就只想着赶紧奔到电脑边写代码。不过人们经常长时间工作的原因，是想着让产品早点发布，为此就要求每个人每周工作80甚至100个小时。*

Thompson：这会把大家整得精疲力尽。激情洋溢地编程，我从来都不会感到压力。我也遇到过你说的那种情况，交付期限特别是对外的交付期限，会让人备感压力。这可一点都不好玩，我不喜欢那样。

Seibel：*要是人这样最终都累倒了，肯定不好。但是从短时间内完成工作的角度看，有什么不好吗？*

Thompson：通常来说，你是在连续不断地这么做。一旦这件事完成了，另一件事情又来了。如果你总是这样赶期限，下一次你就不会有那么大热情了，然后你就不会那么干了。反正我是不会那么干的。

Seibel：*与赶期限相关的是，你需要估计完成时间。你能够估计出完成一段代码需要多长时间吗？*

Thompson：这得看是写给自己呢还是当做产品来卖了。如果只是我自己用，我能估计出来。有瑕疵我也能用，我不必完成剩下的10%，我可以避免那些我自己知道的问题。如此种种。我可以完成它，如果有空的话，再补补这些漏洞，没空的话就先用着。也许这是“完成”的另一种定义。但是如果你在做产品，这肯定会有其他人参与，需要协作，这种情况我就没法估计了。

Seibel：*在1999年的一次采访中你说到，你对Linux嗤之以鼻，引起Linux社区群情激奋。十年之后的今天，它几乎已经遍布全世界，你又怎么看呢？*

Thompson：毋庸置疑，它比以前可靠多了。我还偶尔看看它的代码，现在不像以前那么常看了。我以前开发Plan 9时经常会看看Linux的代码，Linux开发者们总是能超前于我们，他们明显有比我们多得多的资源来处理硬件。因此当我们遇到某一款硬件时，我会去看Linux针对这款硬件的驱动程序，然后再写Plan 9的驱动程序。现在用不着看了。我也在用Linux。偶尔也看看代码，但是很少，所以我就无法说出，软件质量是否比以前更好了。但是很

肯定的是，可靠性好多了。

Seibel：你是否会仅仅出于好玩而读代码?

Thompson：过去是，现在很少了。当我刚来这儿工作时，我凭兴趣读读代码，就是想感觉一下这里的工作环境。你应该这么干。虽然公司没这么说，但是你应该知道这么做。

Seibel：你是否会顺手拿一个程序并完全理解，还是仅仅看看它们的思路?

Thompson：两者兼而有之。我肯定会首先选那些比较大的库。我会看看其中的主程序。Google的编程风格是如此奇怪。他们采取子程序调用，将其包装成RPC，静态地存储起来。这意味着任何人在任何时间以任何理由，都可以调用。他们调用通用的侦听类的代码，另一个地方的其他人就能得到这个消息，开始工作并且找到那个子程序，进行调用。

Seibel：所以这是一种用于分布式计算的机制。

Thompson：是的。这地方就只干这个。因此代码非常难以阅读。所以，你开始工作，先读绑定代码，然后是工作代码，然后是通用的IPC。照这种方式，你才可以真正开始阅读和理解代码。在此之前，你什么都理解不了。

Seibel：当你在团队工作时，你喜欢什么样的组织方式?

Thompson：很简单，只要是工作优秀，并且合得来的人就行。

Seibel：当你与他们工作时，你是偏好很强的个人代码所有权："我写了这段代码，它属于我，我会为它负责。"还是喜欢代码共有权："我们共同拥有这段代码，任何人都可以修改它。"

Thompson：我一直都是二者兼而有之。如果你发现某人的个人所有代码有问题，可以发邮件告诉他，而修复是他的本职工作。但有时候找不到人，他们不想要了，不去修复，也没有反应，这时你修改就可以了。常言道："最后一个碰到它的是你。"你拥有它。所以说，二者皆有。你不会有一大帮人，一起编写和修改代码，搞得一塌糊涂。这会由所有者来审核。但是所有者却很容易改变。

Seibel：当前有人倡导结对编程，说的是，2个人一起编写代码，你尝试过这种方式吗?

Thompson：一些小的程序可以这么做。大多数时候，我负责打字，但如果

另一个人明显比我打得快，他会坐下来打字，我来说。我这么做过，但仅限于几分钟到几小时的编程，我们两个人一起完成本来可以分头做的事情。

Seibel：那你有没有发现结果更好了或是工作更快了呢？

Thompson：结果不是更好。也许调试速度是快了些，因为你在打字时，别人从你肩膀上望过去，就能发现bug。从这个意义上说，我觉得结对编程产生的bug要少一些。但我并不觉得这可以作为一种编程哲学来使用——而现在却正是如此。

Seibel：你还喜欢编程吗？

Thompson：是的。我喜欢小点儿的程序。小，意味着你可以在一个月内完成。如果是要一年才能完成的庞大的软件，我无法坚持那么长的时间。

Seibel：一直都这样吗？还是你对长期项目失去了热情？

Thompson：我不知道。这取决于实际的事情。许多需要多年才能完成的大项目，像操作系统，你可以把它划分成许多有趣的部分，从而变成许多小项目而不是一个大项目。但是还有很多项目，只能是一个大项目，我一直都认为那些项目太难了。我需要满足感，以及反馈。如果得一直坐在那儿，天天干，月月干，除了一大堆代码外什么也看不到，那么我会有问题的。

Seibel：你主要是在做研究工作，看起来你有很大的自由度来做自己喜欢做的事情，但是，当它成为一份工作后，你还认为它有趣吗？

Thompson：是的，编程一直都很有趣，主要是因为我选择了自己喜欢的。就算它是一份工作，早在大学时期，就有非常非常多的工作机会。在我看来，许多人都在做着形形色色的工作，但无论他们做的是什么工作，他们总是需要一些人帮他们做一些编程的事情。这些对我来说太合适了。这些小工作我能胜任，并且我可以几天就完成，还可以随意选择我想做的。

我还记得我的第一份工作是一位人文教授给我的，编目荷马的著作。他收藏有《伊利亚特》和《奥德赛》的卡片，希望统计词语频率和数目，基本上就是对这两本著作进行统计分析。这很有趣。它属于文本处理，只是那时还不是用电脑处理。那是我第一份零工。

Seibel：在1999年接受采访时，你谈起曾告诉你的儿子，让他去学生物学而不是计算机，因为你认为计算机已经不流行了。这是10年前的事了。你现在怎么看呢？

Thompson：我还是同感。人们曾经预测到的都已经实现了，在计算机领域也没什么新的理论出现了。最后的重大事件，我认为是因特网，而它在1999年已然出现。一切都在发展之中，个人电脑的处理速度始终在以指数级增长，但这又有什么特别之处呢？

Seibel：回首Unix的历史，看起来你们这帮人创造了一个操作系统，就是因为你们想找个与电脑打交道的方式。所以，为了做一些今天看来也许很简单的事情，比如说在电脑上写个游戏什么的，在当时，你却不得不写出整个操作系统。你需要写编译器，并构建大量的基础设施才能做一些事情。我相信这本身都很有意思。但是我想知道，是否刚才讨论的现代编程的复杂度，有那么多层嵌在一起，不过是相当于以前我们说："那么，第一步就是得构建你自己的操作系统。"至少现在你再也不用这么干了。

Thompson：但是比那更糟糕。如今操作系统不仅是现成的，还是强制性的。如果你面试那些刚从计算机科学专业毕业的学生，会发现他们根本不懂底层计算。他们对什么是计算机以及计算理论本身的理解都很抽象，抽象得令人恐怖。他们根本就不懂。

Seibel：我在想，你为什么要建议自己的儿子去学习生物学而不是计算机科学。编程可以带给人们通过计算机这种神奇的机器来定义处理流程的智力乐趣，无论你是在很接近硬件的层面，还是非常抽象的层面，其中的乐趣不都是相似的吗？

Thompson：这很容易上瘾，但你决不会想去让自己的孩子也深陷其中。而且我认为编程已有些异化。可能只是缘于我年纪慢慢大了，不过你只是在一层之上再建立一层，然后再建立一层。例如，你真的无法从编写一个有限状态自动机（DFA）中得到好处。我觉得本质上，算法尤其是新的算法必定随着时间的推移会变得越来越复杂。一种新的算法差不多得基于其他50个小算法。回想我还是个小孩时，做这些小算法会觉得它们很有趣。你能够理解它们，而不必像会计似地把它划分为各种情况，这种情况由这种算法解决，这种算法你读了却并不真正了解，等等等等。因此时代不同了。我真的觉得大不相同了，主要是因为随着时间的推移，一切都在分层，而我们就在处理这些层。这可能是因为我慢慢变成个倔老头而不想去理解这些层。

（编辑：武卫东）

13

Fran Allen

董金乾 译

最初，Fran Allen只是打算做一名数学老师，可为了还清助学贷款不得不先临时找份工作，结果自1957年加入IBM研究院之后，竟阴差阳错地开始了自己的程序员生涯。有趣的是，她最初的工作还是教书，只不过不是教数学，而是教那些对高级语言有着严重抵触情节的IBM科学家们使用刚刚发明的Fortran语言编程。

后来，Allen没有再回到学校教书，反而在IBM一干就是45个春秋。这期间，她参与了一系列编译器项目的研发，诸如STRETCH-HARVEST机的编译器、雄心勃勃却未能实现的ACS-1超级计算机的编译器，以及她自己创立的并行翻译项目（PTRAN）。并行翻译项目中，开创了针对Fortran程序的自动并行化技术和静态单一赋值中间表示法（Static Single Assignment intermediate representation），后者目前广泛使用在静态编译器和即时（JIT）编译器中。

2002年，Allen凭借“在优化编译器技术的理论和实践领域中的开拓性

贡献”荣获图灵奖，成为40年来第一位获此殊荣的女性，同时她也是IBM第一位女院士。此外，她还是电气和电子工程师协会（IEEE）院士、美国计算机协会院士、美国国家工程院院士、美国艺术与科学院院士，以及美国哲学协会会士。

回顾自己的职业生涯，Allen见证了女性在计算科学领域的地位变迁。计算机行业发展之初，尽管程序员作为新兴职业定位还不明确，IBM等大公司却多招募女性做程序员；几十年过后，结果竟是男性在这一领域占绝对主导地位。

在我们的谈话中，她谈及计算科学领域中女性地位的变迁，还有增加计算机领域多样性的重要性，以及为什么C语言令人惋惜地伤害了计算机科学的研究。

Seibel：*你的编程生涯是如何开始的？我听说你最初想成为数学老师，后来为了偿还助学贷款才去IBM工作的。*

Allen：在纽约要获得合法的教师资质，起码得有硕士学位。当年，我已经拥有数学本科学位并辅修过物理学课程，在教过两年书之后我进入密歇根大学攻读数学硕士学位。而在密歇根大学，要想获得硕士学位，必须选修两门专业之外的课程，我当时选择了计算技术。其实在1957年的时候，还没有计算机科学这一学科。严格说来，计算机科学是在10年之后才开始兴起的。不过，在当时的工程院校中已经有一些相关课程了。

Seibel：*那些课程都教些什么呢？*

Allen：学校有台IBM 650 大型机，它可跟我们今天使用的计算机大不一样。当时我们主要学习的是IBM 650 大型机程序设计。我们不仅要学习机器本身的细枝末节以便用汇编语言编程，还必须在这台机器上运行编写好的程序。不过，这对于动手实践倒是非常有益的。

Seibel：*是不是你得先在卡片上打孔，再将这些卡片输入到机器中？*

Allen：是这样的！此外，还得运行并修正出现的bug。IBM 650是个磁鼓计算机，指令就存储在那个一直旋转的磁鼓上。要想提高机器的运行速度，就必须去研究如何将指令均匀分布在磁鼓上，以便随着磁鼓旋转下一个指令正好处于合适的位置。

Seibel：*在此期间，IBM 来你们学校招聘。是什么吸引你去IBM工作的呢？*

Allen：说来也很简单，我当时正急着找工作。由于我欠着助学贷款，而这时恰好有招聘人员来到学校，再加上这份工作又恰好在纽约。因此我就去随便填了份申请表，甚至当时根本就不清楚是谁在面试我，后来才知道是IBM研究院。说真的，刚开始的时候，确实一无所知。

几个星期后，当我正在参加伊利诺伊州南部的一所师范学院的教师招聘面试时，突然接到一个电话。那段时间，我都快崩溃了，差不多除了我以外，其他人大都已找好了工作。接电话的那一刻，我正在赶路。当时也没多想，二话没说就径直接受了，甚至都忘了问对方是哪家公司。等收到录用通知时，才明白是IBM的波基普西实验室。

之后，我去那儿做了一名程序员。当时，IBM公司正值快速向计算机行业扩张之际，由于当时还不存在计算机科学类课程，因此他们会聘用能发现的一切相关的人。

Seibel：*进公司后，有什么培训吗？*

Allen：基本上是边做边学。公司倒是有自己的安排，可我却不记得有任何编程类的培训，现在想想都觉得奇怪。我原以为公司会根据个人背景安排相应的培训，并且这些培训一定都非常正式。

鉴于我之前当过一段时期的数学老师，因此分配给我的第一份工作任务，就是去教IBM的科学家和其他程序员学习Fortran语言。我是1957年7月加入IBM的，而Fortran语言是在同年的4月15日才刚刚发布的。当时我所在的IBM研究院宣布，9月之后的所有程序都必须使用Fortran语言来编写。IBM就是以这种方式说服自己员工的，与他们试图说服外面人的方式没什么两样。

Seibel：*在那个年代，IBM就已经有独立进行科学计算的科学家了？*

Allen：是这样的。他们使用704大型机来计算，Fortran 最初就是针对这款机器设计并优化的。

那些科学家们更习惯于用汇编语言编程，然后在这款机器上运行自己亲手编写出的程序。确切地说，要先预约安排时间，轮到自己之后才能运行的。这和我在密歇根大学时的工作方式没什么两样。对于同样的工作，他们不相信使用任何高级语言可以比直接用汇编语言操纵机器编程做得更好。

Seibel：*这算得上是科学家们最后一次使用新的语言了吧？因为他们直到现在仍在使用Fortran，对不对？*

Allen：的确如此。遗憾的是，Fortran入门的过程并不那么令人愉快。可是，

随着对Fortan理解的逐渐深入，我们所有人都忍不住对它啧啧称奇：Fortran本身就是一门编程语言，此外它还提供一个极为先进的编译器，这个编译器已成为现代编译器结构的基础。

Seibel：*据我所知，你接下来参与了Stretch编译器这一大型项目。不知道在你加入Stretch编译器项目之前，是否还参与过其他什么项目？*

Allen：在此之前，我还曾参与过两个项目。其中一个是使用汇编语言针对704大型机开发的受控式自动化调试系统（Monitored Automatic Debugging System）。在开发该系统的过程中，不仅乐在其中，同时也获益颇丰。

说起来，那个系统算得上是操作系统的雏形。我和另外两位同事加入该系统的研发后，首先给那台704机加装了不少自制的按钮，这在那个年代可算是司空见惯的事了。在这些按钮中，有一个是应急按钮，当怀疑程序因死循环无法正常工作时，就可以按下这个按钮来结束程序。接着，我们就开始编写调试器。还记得当时我负责将汇编语言写的源程序转换成竖排二进制输出。当使用横排二进制格式的读卡设备时，每行所包含的各个位正好对应着源程序的指令；而使用磁带的话，读取的方式就不同了，当时为了方便采用的是竖排二进制格式。直到今天，我还珍藏着那段程序。

现在仍能回想起来，当时最令我陶醉的事就是阅读源代码。在我记忆中，那些代码真是优雅无比，我时常沉醉其中，感叹设计的精巧。心想，这代码写得真漂亮，不愧是老手Roy Nutt的杰作。

Seibel：*在你看来，什么样的程序才称得上漂亮？*

Allen：要么能简单而又直接地解决问题，要么能通过显而易见的程序结构展现隐晦的问题。从那以后，我就养成了通过研究已有的代码实例来学习编程方式或新的编程语言的习惯。

Seibel：*你是如何阅读代码的呢？比方说，当你要学习一门全新的语言，并且也已经找到了使用该语言编写的程序样例，接下来你会如何开始分析呢？*

Allen：我还是举个例子来说更好些。我有个同事曾编写过一款语法分析器，这个语法分析器后来经过逐渐演化，最终成了我主持研发的并行翻译项目PTRAN的一部分。一开始，我想弄清楚的是这位同事在语法分析器里使用的方法。因为在我眼里，它算是世界上最好的语法分析器了，更令人欣喜的是它现在已经开源了。这款语法分析器的杰出之处，就在于它有很强的即时纠错能力。

为了弄清楚它的工作原理，我找来源码开始阅读。我认识该语法分析器的作者Philippe Charles，知道他十分讲求优雅的代码美。对我而言，学习一门新的语言，或是针对某个难题寻找新的解法，最简单的办法就是直接阅读源代码，而且这些源码最好就出自自己身边杰出程序员之手。

Seibel：*你是如何深入分析代码的呢？通过执行来一步步跟踪？还是自顶向下阅读，建立起软件结构的蓝图？代码阅读，可是一个十分棘手的问题啊！*

Allen：没错，的确十分棘手，但通常，对于该程序的结构，要么先有些直觉，要么先了解一下该领域，然后直接从中间部分入手探寻程序的核心。这绝对是个好方法，无论是想学习算法，还是想学习如何使用该语言写出优雅的程序。

Seibel：*在调试方面，有什么特别的经历吗？*

Allen：确实有一些。我记得，有一次负责MAD系统的操作员在深更半夜给我打电话，说有个已提交的程序本应通宵达旦地连续运行，但现在却怎么也运行不起来了。考虑到那台机器本身并不具备多少错误检查能力，更不用谈什么错误纠正了，我们就采用了一种特别的算法来完成自动校验，以确保数据正确无误。

一开始仅凭电话中的只言片语，我根本就没法定位到底是哪里出了问题。不过讨论了一段时间后，突然间，我意识到自己编写的计算校验和程序不能处理一种特殊情况。就算那个程序是正确的，它也没法越过障碍，就因为我计算校验和的方式。我连忙给那个操作员回电话，告诉他有种方法可以暂时克服那个问题。

再说说另一件事，我之所以还记得这一件事完全是因为这一次我对自己的表现十分满意。那还是在Stretch项目中，有位同事喜欢通宵工作。他这个人身材魁梧，不苟言笑，有时甚至还有些令人望而生畏。有一天早上他走进办公室后，径直将一份调试代码清单甩在了我的办公桌上。我看出那是程序的崩溃转储，打印出来后又大又厚。他指着那张转储清单里的某一特定位，质问我说："你为什么要设置这一位？"原来他一整夜都盯着这个犯愁。奇怪的是，我当时竟一下子就知道了原因。那并不是hug所在，仅仅只是因为他并不知道那一位有什么用，就主观臆断认为那就是错误所在。

Seibel：*这说的是后来那个项目吧，按你刚说的，MAD和Stretch之间应该还有另一个项目？*

Allen：你记得没错。那个项目是辅助研究院里的一位科学家设计硬件布线图，与在芯片上如何布线有关。我们在实现一个基于数学的解决方案，由于基板面的尺寸导致限制条件很多。我是该项目中的一名程序员，在我的印象里，那个项目一共也就两三个人，而且全都是女性。

Seibel：*接下来该说说Stretch项目了，这可是大项目。*

Allen：由于在Fortran方面的经验和对Fortran编译器的深入了解，我被从研究院中选了出来，参与到IBM下一个大型项目——Stretch超级计算机的研发中。这个项目始于1955年，如果我没记错的话，大概是在1956年的时候正式命名为Stretch，它的目标是要造出比当时世界上最快的机器还要快100倍的机器，这样的计算机绝对会令人刮目相看的。

当时大家一致认为，编译器的好坏将决定这个项目的成败。因为要想达到上面说的那么高的性能，最大的挑战来自于内存访问，而编译器对这一环节有着重大的影响。

Seibel：*是因为处理内存延迟要比程序员们手工编写汇编程序更为复杂吗？*

Allen：没错。而且，因为内存延迟的问题正被大量硬件内置的复杂并发所解决，再加上存储设备的组织本身就是多路交错的，所以数据将以不可预知的顺序进入计算单元执行。同一时间很可能同时执行六次访问。而计算单元本身又具有流水线，完全可以在同一时间执行多条指令。机器中最复杂的单元要算先行缓冲区，因为精确中断是作为体系结构的一部分来设计的，因此要求该缓冲区不但可以跟踪所有并发，而且能在中断出现时取消缓冲。

那可是有史以来我见过的最复杂的计算机，在上面编程一定极为美好。怎样充分利用这台机器高超的性能，编译器面临着很大的挑战。这真是个极具挑战性的项目。

我们几个就这样从研究院中被选了出来，参与到编译器和操作系统的研发之中。与这款梦幻般的机器一样，它的编译器本身也不禁令人啧啧称奇。鉴于我之前在Fortran优化器开发中的经验，我最终参与了针对Stretch计算机（后来正式命名为Stretch Harvest机）的优化器研发。虽然编译器的概要设计是由另一个团队完成的，不过我们之中有四人参与了详细设计，包括编译器的内部接口以及详细的规格说明书，并负责各个不同的模块。我负责优化器，一人负责语法解析器，另一人负责寄存器分配模块，还有人负责汇编接口部分。

Seibel：*项目的人员结构是什么样的呢？*

Allen：嗯，大概有三个人负责编译器的总体架构，他们决定要有一个语法分析器模块，要有这样一个模块或是那样一个模块，以及如何组织各个模块等。在这些人之上还有不少人。这是一项产品，因此一定会有不少人负责决策和管理。

此外，还需要为每个规模较大的组件配备项目监督员。因此，我们按要求成立了一个小组一起负责接口设计。值得一提的是，这个小组一共四个人，而其中三个都是女性。

Seibel：*那么，是否还有别的程序员一起参与具体的实现呢？*

Allen：当然有。我们组一共17个人，全都负责具体的编码工作。

Seibel：*设计阶段和编码阶段之间是什么关系？你们是四个人一起讨论，整理各模块之间的接口。这个接口设计工作是安排在这17位程序员开始编写代码之前，还是在编码阶段再反馈到你们的设计中来？*

Allen：设计是在编码中逐渐完善的。给我们设定规则的人正是我们向他汇报工作的人。不同模块的负责人，比如我，要统一向名叫George Grover的人汇报，他负责从技术上制定出更大的方案。而这又主要是由客户方面的限制来决定的。在那个时候，需要团队协作很广泛，且灵活性很大。部分原因在于，我们的工作还算得上有些开创性，只不过有很大的交付期限的压力。这样，也就没有了那么多的管理层级，更多的是依赖团队合作了。

Seibel：*你管理的人中间，有没有人在编写代码的过程中，导致需要修改上层设计中已经确定的各模块如何配合的实现方案？*

Allen：有这样的时候，如此一来，这些接口就没法再工作了。而这正是跟踪工作进度的一部分。此时，我们四个人会召开团队会议。但除此以外，我们大部分时间都花在编写由我们负责的组件上，对这部分我们有很多的自由。

软件工程是后来才兴起的。一开始并没有软件工程，也并没有严格的项目流程。在IBM之后的项目中，由Fred Brooks负责的360项目最终演变成了一场巨大的危机，我当时没参与这一项目。在1963年左右的时候，360项目进展得相当顺利。由于人手不够，一大批对于软件开发一无所知的硬件工程师加入到项目中来。这样一来，最终搅得整个项目简直是一团糟。

在360发布之后，有人，我也不知道他有没有参加360项目，给IBM的上级主管们写了封信，提议了一个称之为Cleanroom的软件工程规范。他声称，如果能遵循他列举的一系列流程进行开发，完美的软件就可以开发出来。因为有过如此混乱不堪的项目管理——这可是我的一面之词——管理层竟完全采纳了这一建议。

Seibel：*就因为360项目让你们吃尽了苦头？*

Allen：我想是的。此后，IBM的产品开发强制引入Cleanroom流程，这可是一整套完整的流程规范。一方面，会有一些人对各项具体事物进行规划，另一方面，会由另一个团队来做设计。设计人员将特别注意细节的设计，以便程序员对此设计进行编程。团队之间互不协商，大家只需完全按照流程做事，完美的软件就会应运而生。

Seibel：*在360项目中，Brooks同时负责软件和硬件开发，对不对？*

Allen：对，我想是的，他全权负责那个项目。不过，他更换掉一些软件模块的负责人，而由具有硬件经验的人来负责这些模块。这算是做对了，因为做硬件的人已经具有一种极好的设计硬件的原则：硬件的芯片设计、测试过程等。这是一种古老但更为严谨的表达设计的方式。我们软件开发人员，只管开发就好了。

Seibel：*所以在你看来，至少在这个项目中，他们在软件开发流程中引入了一些好的方式从而拯救了这个项目？*

Allen：必须得这么做，但这样一来却深深伤害了软件开发人员，因为让这些对软件一无所知的人加入项目，就是要强加他们自己的设计观点、设计规格等，诸如此类。

Seibel：*这样做确实有助于挽救该项目。这是否也使大家更有信心进一步实施Cleanroom过程，某种程度上说，这一步走得更远了一些？*

Allen：是的，我觉得完全可以这么说。事实上也确实如此。Cleanroom过程、瀑布过程，由于管理层的大力提倡而开始盛行。

Seibel：*这主要是那些从事硬件开发的人提倡的？*

Allen：不，这是从事软件开发的人提倡的。在我看来，他根本没参加过360项目，不可能有那种竭尽所能试图拯救项目于水火的深刻体验。但很多了解软件结构的人，也被他当时的发言震住了。有时为了推销自己的观点，还真

得大胆些。

Seibel：你曾经在采用了那种流程的项目中工作过吗？

Allen：哦，工作过。我发现这种流程真的很令人沮丧，因为在这种项目中，架构师和程序员在软件开发的初期阶段没法沟通和互动。当年有过这样的问题，现在差不多还是这样，那就是软件的开发周期都很长。如果软件规模比较大，需要开发好几个月，甚至好几年时间，在开发的过程中，环境在变化，需求也在变化。可对于软件需求，握有最终发言权的是客户。

Seibel：然后，你是否在开发流程的全过程中推动需求变化？还是，人们开始缩短流程，直接对程序员说"好的，我们分析需求，发现客户需要某某功能"？

Allen：没有人能写出那种经过了多年的开发后，在详细程度上仍然是恰当和有用的软件规格书。这是一个问题。现在，我们当然还有另一种流程，差不多可以这么描述：凑合能用，之后就扔掉。

Seibel：对了，Brooks在他的名著《人月神话》中说过："构建一个然后扔掉，因为你就是打算这么开发软件的。"

Allen：是的。事实上，真是这样的，我非常相信这一点。但在我看来，很多时候，在你开始构建之前根本就没法思考。

我一直希望有一幅略图，确切地说，应该是一个模型，通常是流程图和一些接口定义。在当时，因为不能经常使用机器，所以理所当然得更多地使用流程图。这是一个相当好的模型，使用这一模型，可以方便思考系统各组件之间如何交互，将做什么，将在哪里做什么，以及各组件的功能。我不知道现在类似的是什么。

Seibel：即使都是流程图，有的是按文件规定做出的正式流程图，有的是画在黑板上帮助理解的流程图，你更多是采用哪一种？

Allen：在某些情况下，我会采用正式的流程图。软件核心部分的逻辑通常会非常复杂，这时我们会画出正式的流程图。除此之外，都采用非正式的流程图，就当是解决问题的一种工作方式。在黑板上画下的流程图，常常会保留一个月或更长。

Seibel：你领导大型项目，也就是PTRAN编译器项目时，也是第一次研究如何实现显式并发，与之相对，隐式并发是通过CPU流水线等其他方式实现的。

当你开始这么做时，对于你和IBM来说，都算是一个全新的项目吧。

Allen：对IBM来说这一项目的确是全新的，但实际上我们是属于很晚才涉足这一领域的。这项伟大的工作最初起源于伊利诺伊大学在1969年至1970年间开展的一个实际的语言实用学项目。

Seibel：PTRAN编译器能编译什么语言？就是没有添加任何并发结构的Fortran语言吗？

Allen：是的，我们就那么开始的。我想做的是，就像优化一样：用户从应用角度使用语言写出自然的顺序型代码，再通过编译器优化映射到机器语言，并利用并发的优势来最终执行。

在PTRAN项目中，我们仍基于一个被我们内部称为dust decks的代码库实现，并自动利用硬件的并行组件。

Seibel：所以，基本上，这是针对我们今天所说的对称式多处理器计算机的？

Allen：是的，可以这么说。并行有很多模式，这也是难点之一。我认为这可以大大简化。但是多核并行是并行模式中极其有趣的一种，至少对我来说是这样。其实有很多种并行模式。

事实上，我们从现有的基础开始工作，特别是Dave Kuck的工作。还有一些来自纽约大学的工作。我们从这些地方聘请大量刚刚毕业的博士，他们已经建立了大量的专业知识。我们同时在实践和理论方面研究，都有相当多的重大成果。我坚信，要从实践角度考虑可识别算法、理论以及解决问题的方法，并要将算法用于实践，展现它们真正的价值，研究如何应用。我认为，这对于我们的研究领域是最好的做法，在同一项目的两方面同时奏效。

Seibel：在PTRAN项目，你领导着一个团队。那时，你还编码吗？

Allen：我不再做具体的编码，但没有远离编码。举个例子说吧，当静态单一赋值的理论完成时，我还不知道该如何在合理的时间内实现。我的意思是说，这是一个非常好的算法，但我没有看到有在时间和空间限制下的具体真实的实现。因此，有了这么一个挑战。我必须得看代码。我需要它，必须实现它。它不能只是一篇显示图形和复杂性界限的论文，虽然它非常棒，很著名。

如果我们不能在实际系统上实现这一算法，这种挑战就会一直存在。它不会像我预想的那么有用。最后，有个组员实现了一种编码。我阅读代码，一块都不放过，也看了使用的数据结构。这个实现真是很让人惊喜。我说：

"就是它了。它能用。"

Seibel：所以你会审阅将合并进系统的所有代码的片段？

Allen：是的，是的，是的。

Seibel：你也管理所有这些人？或者是你只负责技术架构，另有人负责管理团队？

Allen：不，我就是这个组的研发经理。组里核心人员大约有10个到12个，我们分配好工作，以便让每个人全权负责一部分。

Seibel：至少从Gerald Weinberg写出《计算机程序设计心理学》一书后，人们一直在辩论，是让人"拥有"代码以便让他们能对代码负起责任好，还是让更多的人一起协作以避免代码只有一个人理解好。听起来，你认为分配代码的所有权好些？

Allen：我们协同工作，协作的是有关整个系统和软件实现的进展。有些人很擅长实现，于是他们可以负责一块，优化器或程序内部分析正好适合一两个人来做。但是，大多数人都是在做理论工作，撰写大量的论文和算法。我觉得，只有这两类特殊的人互补协作，才能切实增强整个研发小组的能力。

这是一个正在分析和转换为并行的时期。因此，我试图让每一个从事理论研究的人能写一些代码，通过作为整体系统一部分的代码来表达理论。而让负责编码实现的另一部分人，能尽量写得详细些，以便更多人能够理解。

Seibel：很多的程序员都会尽一切努力避免成为管理者。而你同时也很喜欢管理类的工作？

Allen：这么说吧，早些时候，研究院并不区分管理和开发工作，从事管理工作并不意味着晋升，更不会涨工资。只不过，总得有人来做管理工作，当有人问你"嗯，难道你不愿意做管理？"，或者说"很明显，你是负责这项工作的最佳人选"，你很难有理由拒绝的。而且，这只是技术管理，不会涉及太多的人事管理。但是在研究院里，人人都是研究员，从他们进入研究院的那天起，一直到他们职业生涯结束。我所有的同事，我们都是一样的。因此，做不做管理无关荣誉。

Seibel：这么说来，被选来做管理的人都是真正擅长管理的。那么你又是如何学会那些管理技能的呢？

Allen：嗯，我被送到管理学校参加培训。每个人都如此，只不过我是那个时候第一个被送去培训的。但我认为这可能和我的成长经历有关。因为我的父母共有6个孩子，负担相当重。我是家里的长女，下面还有5个弟妹，因此从小以来，我习以为然去承担些责任。

Seibel：技术管理的困难之一是，在关于解决某问题时强加自己的技术观点和给人们足够的空间表达他们自己的想法这二者之间，你需要寻找平衡。

Allen：我想自己在Stretch项目中有过这样沉痛的教训。记得有一天，这个项目的几个人对我说：我们应该使用列表和散列函数。好吧，我们知道列表程序，但不了解散列。所以，当很多人表示希望在符号表中使用散列时，我说："不，我们不能这样做。我们不知道怎么做。"如何，如何，我说了一大堆。接下来的星期一，我来时发现他们已经做完了。他们拆掉系统，重新用散列构建。它不仅能工作，而且还快得多。所以这对我而言，这是一个很大的教训。我本应对一些新的想法更加开放才对。

Seibel：所以，有时候，甚至经常，你的小组成员实际上知道他们在谈论什么，这时候，你不应该干涉太多，因为你可能会扼杀了一个好主意。当你是正确的，而他们的主意又确实有点缺陷，但你并不希望打击他们太多的时候，这是比较棘手的事情。

Allen：是有些麻烦。往往是某人知道某一领域的一些知识，并希望将这些知识应用到正在进行的项目中，而并没有将其融入项目足够长时间来让大家足够了解，这往往延误了项目的交付期限。

我深刻理解这一点是在做一个分包项目时。我领导着一个开发小组忙于构建一个出色的优化器，这种工作我们在研究院曾经针对PL/I做过，PL/I是一种大而不一样的语言。但是，一名参与分包工作的员工刚刚了解面向对象编程，并决定在项目中完全应用它。我阻止不了他，尽管我是项目监督员，后来项目毁了。归根结底，这是因为PL/I有很多指针，我们则要一直跟踪指针；而使用指针，我们要用两个指令才能追踪到值，总得先找到指针的值。

Seibel：你指的是生成的代码。

Allen：生成的代码和编译器本身，因为它引导着编译器。每当完成了一步，你必须检查它的有效性。检查，复查，再复查。今天仍然这样做。其中一些我们没能好好吸取教训。我想我没能好好处理这个问题，因为我本应该提出，

在那种情况下，采用面向对象技术的成本会有多高。结果编译器无可救药地慢，最终整个项目被取消了。

Seibel：你什么时候开始直接为IBM公司构建产品，而且这个产品必须满足交付期限？

Allen：当然，Stretch项目就是以这样的方式进行的。我曾经进行了两三次的产品开发，在那种情况下一周接一周地进行代码评审，直到交付期限。我非常尊重这些流程，对于最终的结果和执行这一流程的小组来说，它们是如此地重要。每个星期五大家坐下来阅读代码，听人解释他们为什么要这么做和做了什么，并要查找别人的错误，这是多么痛苦的事情啊！

Seibel：非常痛苦，但是，不值得吗？

Allen：绝对值得。在PTRAN项目临近结束时，我们要将部分研究成果转化为产品，每周要花半天时间专门解释我们代码中的错误，他们代码中的错误，或者其他什么，每周都如此。这一直持续了10个月，我称之为星期五下午奉献日。

Seibel：当你在这些环境中工作时，你是否觉得流程要求你必须按某种精确程度，估计完成一部分软件需要多长开发时间？

Allen：的确，他们是有这方面要求的。产品开发人员确实这么做。所有都得跟踪，我想现在恐怕仍然如此。一定程序上是为了从统计角度掌控代码质量。本周又出现了多少bug，嗯。我喜欢产品研发实验室这样的工作环境，因为在这里所有事物都变得很真实。

Seibel：你招聘程序员时，会看重些什么呢？

Allen：嗯，我和很多大学都有联系。纽约大学有一些撰写编译器的超棒的教员，那帮人写编译器代码真是训练有素。

Seibel：因此，你可以雇用你认识和信任的教授推荐的人。当前来面试的人并不是你所信任的教授推荐的面试者时，你如何通过几个小时的面试判断出他是否会是一名合格的程序员？

Allen：对于来IBM研究院面试的人，一开始，我总是试图找出最令他们兴奋的东西是什么。在我看来，这是最基本的门槛。它们是否与编程或计算机相关倒不重要。如果他对什么事都缺乏热情，那么在任何开发团队里他都不可能激情四溢地工作。

人都有冒险的时候。有一次，我冒险招了一个人，他的论文指导老师说他有诵读障碍。他工作得并不理想，因为在某些方面他不适合，但他自己开了家公司，现在我有时还给他提建议，是关于技术实现方面的。他相当通晓世事，所以这也算不得失败。虽然从项目上说是个失败，但从他和我们其他人之间的关系上来说，并不失败。

Seibel：*不久前，你一直在参与指导，在IBM有个导师奖还是用你的名字命名的。对于新程序员如何成长为优秀的程序员，你有什么看法？*

Allen：指导的事情，近些年我已不怎么做了。不过，一般情况下，我会鼓励年轻人一开始先不要直接做管理，虽然这对有管理天赋的人特别有诱惑力。先在技术方面赢得声誉，无论是一门科学、算法或是编写优秀的代码，总而言之，不管它是什么，建立良好信誉是第一位的。如果你的确想做项目管理类工作，这一定会对你大有裨益，因为你已经从中学会技术工作的工作方式和准则，以及如何按技术规律做事。

Seibel：*不精通技术却擅长团队协调的人，能成为好的管理者吗？*

Allen：这不是不可能，不过他得首先承认自己并不精通技术，其次还要能分辨团队中谁擅长技术，谁不擅长技术。

Seibel：*这可算是最难的了。对你来说，如何区分出有真才实学的卓越程序员？*

Allen：嗯，我总是乐于去发现那种可以让我眼前一亮的人。这对我至关重要，因为我要花大量的时间思考系统，所以我希望有人，至少在研究院，能向我展示一些新鲜有趣的点子，或者是观察问题的全新角度，又或是解决问题的新方法。

此外，我会参考其他人的想法。当发觉自己对某人的评价高于团队里其他人对他的评价时，我发现自己常常是错的，而这正是真实学习的机会。如果你身处一个好团队，通过大家互相评价是辨别个人工作好坏的好方法。

Seibel：*你还记得什么时候最后一次编程吗？*

Allen：哦，这可有段时日了。当C语言盛行时，我差不多就很少编程了。这是一个很大的打击。我们当时在语言优化和转换方面进展良好。我们漂亮地解决了一个又一个问题。当C语言流行开来，在一次SIGPLAN编译器会议上，有一场贝尔实验室的Steve Johnson和我们研究院的Bill Harrison之间的辩论，Steve支持C语言，而Bill则在我当时领导的 一个自动化编译项目中工作。

辩论的焦点是Steve关于不再需要构建优化编译器的论题，他辩称这是因为程序员会自己考虑优化。优化是程序员需要解决的问题。设计C语言的动机是为了解决三个不能用高级语言解决的问题：其一是中断处理；其二是资源调度、接管机器、调度队列中的一个进程；其三是内存分配。这些工作不能用高级语言做，而这正是C语言产生的理由。

Seibel：*在你看来，如果限制C语言只用来编写操作系统内核，C语言是合适的选择吗？*

Allen：哦，是的。真那样就好了。事实上，你需要这样的语言，用这种语言，专家可以真正微调而没有大瓶颈，因为那些才是需要解决的关键问题。

到1960年，我们拥有一长串令人惊叹的语言：Lisp、APL、Fortran、COBOL、Algol 60。这些都是比C高级的语言。自从C语言诞生后，我们严重倒退了。C语言摧毁了我们推动最先进的自动优化、自动并行化以及高级语言和机器语言自动映射技术的能力。这也是编译器基本上不再在学院和大学里教授的一个原因。

Seibel：*可现在确实还有关于如何构建编译器的课程啊？*

Allen：很多学校都没有了，这真是令人震惊。当然还有相关会议在继续召开，还有人在研究如何改进算法，这是好事，但在我看来，这些工作所能弥补的微忽其微。由于C之类的语言对问题的求解方法彻头彻脑地过度指手划脚，这类语言就是破坏计算机科学作为学术研究的元凶。

Seibel：*但近来流行的大多数较新的语言都比C语言高级，如Java、C#、Python和Ruby。*

Allen：它们还是过度细致。最核心的一点，在于它们都指定了数据的存储位置。如果看过其他几种语言，你就会发现这些语言中决不会有指定数据的存储位置的语句，也决不会有如何移动数据的语句，更不会有选择在机器的什么位置存放的语句。归根结底，它只关注任一时刻数据的值。

Seibel：*但除了C和C++，几乎再没有语言有指针了。Java有垃圾回收，数据自动搬移。你认为这还是过于指手划脚吗？*

Allen：是的。我相信，就像我们曾在计算并行方面取得的突破那样，我们也很有机会在数据优化方面做出成绩。我们目前对数据管理得并不怎样好。我们没有数据自动管理的好的解决方案——建立数据局部性，与指令局部性

一起来提高并行性。

让人欣喜的是，这方面现在有很多研究方向。但我认为其中还是缺少更大型、更大胆的概念。这些研究方向中，有许多都受限于已经存在的事物或目前的思想。总而言之，这不可能在一夜之间改变，因为已经有数百万行的代码在那里。但是，我们必须开始尝试打破边界，“这将在这里进行，那会在那里进行”。

Seibel：你职业生涯的大部分时期都是在做高性能计算。我们假设到2019年，或者不必深究具体是哪一年，可能会有1000核的笔记本电脑，这是否意味着高性能计算和日常计算将融合？还是说，高性能计算依然会以截然不同的方式发挥作用？

Allen：一定程度上这取决于它自身的规模。必须能进行每秒千万亿次浮点运算，这是我们现阶段高性能计算的目标，但是我还不知道如何实现它。当然，性能的比拼已渐渐偏向多核，因为多核立足于减少能耗等多方面好处，并且是使用基础物理学来解决一些问题的。

有个比较有竞争力的因素将会推动它的发展，是利用多核将问题从硬件转移到了软件。据我所知，这方面我们还没有做好取得任何进展的准备。要想好好把握这些多核，我认为应在新的语言层面有所突破。我们应该从头到尾仔细研究，但是这需要一些非常新的思考。

我思考这些是在50年前，也许是60年前，ENIAC[①]是在1943～1944年开始研制的，我们建立的不仅是一个美好、令人惊叹的遗产——真的极其惊人，而且是我们需要摆脱的人工制品。取代这些需要很长的时间，而且我认为这种取代将如何进展也有点难以预测。但是如果我们能在正确的方向上萌生一些新的思想，它会进展得非常快。对于很多东西，我们知道如何做计算。但我们不知道如何向机器的运算部件提供数据。

Seibel：关于向机器的运算部件提供数据，与现在的做法做个对比，能否举个简单的例子说明一下你指的是什么？

Allen：对我来说，这意味着接手数据管理。简单说，我们现在是通过引用，也就是由硬件，或者底层操作系统或支持系统来提供数据。通常，引用是在元素级上进行的。

Seibel：你的意思就是说，在结构体或数组的中间放入指针。

① 世界上第一台现代电子计算机，1946年完成研制。

Allen：是的，放置在其中的一个元素之中。依赖于硬件及体系结构本身的协议，这将在计算过程中给使用值的地方带来值。

另一个达到上述目的的办法就是根据相对位置组织数据，以此作为优化目标。另一方面，一种方法对于一类计算好通常对于另一类计算就很差。一种数据组织方式，即使像矩阵这样简单的，当使用另一种不同的方式进行访问时，就不好了。因此，它是一种访问级别和位置的组合。它可能需要一些其他的体系结构工作、硬件工作，但我认为，如果我们将一些引用、寻址功能通过硬件本身实现，就可以做到。在众多的机器之中，有的机器能在数据载入内存的那一刻进行大量的转换。映射也可以在这一转换过程中进行。

在高性能计算中，运算速度是主要的测量指标，所以我们想法设法提高机器的运算速度。如何能尽量让计算的执行单位满负荷运转，是我们面临的一个重大问题，但遗憾的是我们却从未把它列为首要问题来解决，而是把问题抛给了硬件。

Seibel：*你在图灵奖获奖感言中大概这么说过："我们正处在一个十字路口，极有可能迷失方向。我们可能会走错路，这样就不得不再多花些时间折返回来。"*

Allen：是的。

Seibel：*那么在你看来，正确的道路是再次回到自动并行化之类的工作上？*

Allen：是的，但它必须是基于比目前使用的语言更高级的语言展开研究。

Seibel：*错误的道路是寻找能明确表述并行的更好的方法？*

Allen：嗯，在我看来，我们最终会认识到，我们已经创造了远多于已有的一大堆废物。但是，我们又确实需要更高级的语言，并且这些语言必然是领域特定语言（DSL），开发方式必然多种多样，这真可谓是个美妙的方案。

但是，我们必须愿意尝试并利用以上优势，而且还应利用系统集成的优势，并认识到处处皆有数据这一事实。它不再只是封装在程序或者代码之中。我认为，目前有大量可访问到的数据。它们是数字数据或各种信息数据，存储在遍及全球的每个角落，尤其当你是在生物信息行业工作时，感觉会更为明显。因此，我们必须有能力建立一个平台，尽可能多融合些其他知识，这样的计算能力可能迥异于当前。此外，我们迟早还需要考虑系统的可用性和完整性问题。

Seibel：*可用性，是站在程序员的角度说，还是站在使用系统的最终用户角度来说？*

Allen：这是站在系统的最终用户角度来说的。这种可用性是一种资源，一种巨大的资源，也是系统的正确性和完整性。好几年前，我在从事美国国家安全局（NSA）的项目风险管理的时候才恍然大悟，在大多数情况下（高性能计算也不例外），不必苛求计算的准确性。我们并非只有在获取所有数据的情况下才能取得进展。所以，在数据方面我们做了很多漂亮的工作，能得到足够好的答案。在我看来，多核是一个难得的机遇，我们可以回过头来，或者说换个角度思考计算技术的很多方面。

Seibel：*你认为自己是科学家、工程师、艺术家，还是工匠呢？*

Allen：我认为自己是一名计算机科学家。我在自己参与的研究领域内寻求创新和突破。计算机科学出现之时的研究颇为有趣，因为有一系列的问题需要解答，比如，这是一门科学吗？因为凡是名称中带有科学的一般并不是科学。当然，我那时也并不清楚科学的含义。

但是，编译器是一个非常古老的领域，甚至比操作系统还要古老。有时间的话，我倒是愿意查查它到底有多古老。"编译器"这个词实际上源于"嵌入小段指令执行"的意思。比如加法运算就必须表述成计算机能够执行的一些基本的机器指令。这样当进行加法运算时，机器才能跳转到对应的加法运算库并执行。

汇编也是使用符号的语言。我不知道这么说是否准确，但我过去曾以为，第一个使用符号作为变量名的人是Nat Rochester，还是在一台非常早期的IBM机器——1951年的701。他当年负责测试701。在编写测试程序的过程中，他领导的团队引入了符号变量。到现在，我已经看到过一些其他的事情，这使我相信，很早之前就已经有通过符号表示信息的方法。根据我个人的推测，这可能是在20世纪50年代初出现的，甚至有可能追溯到20世纪40年代。举个例子，可以研究ENIAC中的表示方式来追溯早期是如何表达事物的。

Seibel：*所以在成长中的某一刻，你意识到自己已成为一名计算机科学家，建立了编译器优化的理论。但是一开始，你只是一名程序员，是被聘请来编写代码的。到PTRAN项目的时候，你负责管理具体编写代码的程序员。你为什么要作出这样的转变呢？*

Allen：嗯，说来有个原因。其一，我并不是一个非常优秀的程序员。我经

常会犯些错误，并不像当时传统认为的，女人能做优秀的程序员是因为她们关注细节。我不属于这一类。因此，我对理解所有细节并正确实现并不感兴趣，相比之下，我对整个系统如何工作更感兴趣。

我对数学有着外人难以理解的兴趣。如果当初我有足够的钱继续攻读博士学位，我会成为一名几何学家。我喜欢演算过程的严密与精确。这是我发自内心最为热衷的，琢磨各种各样的系统问题，但不必像工程师那样深入细节。深入细节可是另一个完全不同的领域，这我并不喜欢。

Seibel：说起你在技术上对PTRAN项目的贡献，听起来像是你一方面对整个项目的体系结构有大局观，另一方面又可以指出一些模糊的细节的工作原理。

Allen：是的。

Seibel：你怎么认为，这方面的能力是你与生俱来的，还是后天慢慢培养出来的?

Allen：我认为这一定程度上可归因于我在农场长大。如果你仔细看看在我们这个领域发生的许多有趣的工程方面的事情——这个时代或稍早一点，就会发现很大一批人都来自农场。从几个和我一起工作在国家工程院的人中我偶然发现了这一点，他们有很多是来自中西部农场的。他们中有人参与了火箭的设计，有些人则从事实际的工程和系统工作。我觉得这正与农场和自然有关，而我也对这些很感兴趣：如何修理东西，事物是如何工作的。

Seibel：农场是一个有着众多输入和输出的大系统。

Allen：没错。而且，由于它非常接近大自然，所以有自己的周期，自己的系统，你根本无法控制。因此，每个人能做到的就是找到一个适合的位置。

Seibel：你之前提到，当还你在Stretch编译器项目中工作时，引领编译器开发的四个人员中有三名都是女性。你能详细说说为什么会这样吗?

Allen：没记错的话，那是1959年的事了。在那个年代的程序员群体里，女性发挥着巨大的作用。仅就这一点来说，IBM一直可谓是个超级伟大的公司。通过最近阅读的一些历史，我才知道IBM的这一多样性政策可以追溯到1899年，这么多年来竟一直坚持着。这一政策很明确，虽然并没有多少人关注过。

Seibel：在你看来，Stretch编译器项目中女程序员很多，是否缘于公司管理

上的明确政策，直白点说，那些政策规定IBM应该多聘用女性？

Allen：我可不认为他们说过“我们必须多聘用女性”。实际上，他们是在聘用合格的人，不限于性别。那时，整个社会环境对非洲裔美国人来说真挺难的，IBM着实向前迈进了一大步。很多人都不知道，当时在波基普西甚至还有黑人居住区。IBM开始改变了这一切。

Seibel：在一次采访时，你说过一件参加会议的事。不少人看到你后惊讶地说：“你就是Allen？”

Allen：“你是女的！”

Seibel：“我们已经安排Gene Amdahl和你住在一起了。”

Allen：哦，是，是有这么回事。那次是在IBM公司的会议上，当时我们正准备将System Y项目移交至西海岸进行产品开发，并把该项目重命名为ACS。我们在有河贯穿的哈里曼庄园（Harriman Estate）举行了大型会议。除了一两个人，参加会议的都是IBM的员工。会议的组织者来自西海岸，根本不认识我们中的任何一个人。他仅仅按字母顺序安排了房间。

Seibel：后来他们东查西找，最终给你找了间房？

Allen：是的，阁楼上有个女佣的房间。

Seibel：发表论文的时候，你是以Fran还是 Frances署名呢？

Allen：F.E Allen。我不大记得为什么会这么做。但这也再平常不过了，那个时候，大家差不多都是用姓名的首字母缩写。

Seibel：前面你提到，你编程的那个时期，人们认为女性能成为优秀的程序员是因为女性注重细节。最近又认为，正是因为男人有种关注事物的奇异能力，尤其是事物不好的一面，使得目前大多数程序员都是男性。

Allen：是的。

Seibel：针对这些言论，这么多年来，你一定见证过很多次的观念转变。

Allen：是的，今天人们不再说伟大的程序员，而是说因为他们在团队中乐于合作所以伟大。说起女性，她们通常合作得相当好。这几乎是对“女性注重细节”这一观念的当代诠释。

Seibel：尽管在Stretch编译器开发组中女性较多，但在你整个职业生涯中，一定有不少次与你一起工作的全都是男性。在一个满是女性的团队中工作是

一种迥异的经历吧？

Allen：是的。我认为确实是不同的经历，但不仅仅在于有许多女性，还因为她们中许多和我是真正意义上的同龄人，因为大多都是和我一起被大规模招聘来的。基本上，我们年龄相仿，背景相似。所以这完全是一个非常学院式的团队。此外，整个领域也是全新的，有太多未知的东西。我们也不知道我们不知道什么，不过似乎也没有多少人在这一领域有着多年经验或者知道很多。

Seibel：那接下来呢？该领域现在已经没有多少女性了。什么时候开始改变的？

Allen：我花了好几年才想出原因。差不多在六十年代后期，至少是我还在研究院的时候。我后来离开研究院前往加利福尼亚加入到ACS项目中。8年后，当我再次回到研究院时，发现环境与我离开之前截然不同。

多了一个显眼的玻璃天花板。工作上有一定的流程，不少管理规范。管理结构也发生了变化，决策也变得更加正规，尤其是要做什么项目和如何完成既定项目。女性的人数和职位都有显著改变，不过并不是向好的方向。因此我很不开心，原因不言自明。

在70年代初，我十八九岁时进入这一行业时，这个领域充满了乐趣和机会。我虽从未见过自己的进步，但我觉得我可以自由地去做自己认为是正确的事，或是选择喜欢的角色做自己感兴趣的事。可当我回来后，发现一切物是人非。

Seibel：你不觉得，玻璃天花板可能事实上以前就有，只是你并没留意？还是真有什么变化？

Allen：以前还真是没有。最近才我意识到可能的根本原因：计算机科学是在1960年和1970年间逐渐兴起的。它主要源自于工科学校，还有点数学血缘。

而这一时期在工科学校学习的大多是男性。IBM招聘必须符合一定的条件：有一定的学位，并修过计算机科学中的某些课程。因而，满足条件的基本上都是男性，这也渐渐成了条规则。另一件看来也已发生的事是：软件开发已成为一种职业——有很多的流程和保障流程的管理活动，这也使得一切稳步运行。结果研究院成了一个完全不同的地方。

Seibel：我敢肯定，五六十年代时，性别歧视在社会上还相当盛行。然而，

在这期间你却常在有很多女性的团队工作。这些团队为什么会对女性如此开放呢？

Allen：软件是当时最新的东西。时至今日，它依然被认为是“软”科学的部分。这恰恰是女性被其吸引的原因所在。早年，女性便已是ENIAC和布莱切利园（Bletchley Park）里的程序员。女性就是计算者——这是她们的名字。但在工程学和物理学以及“硬”科学，古老的科学领域却没有多少女性。早些时候，就是这样划分的。

之后，工科学校开始有了女毕业生。现在，工科院校中女大学生的比例基本为20%。卡内基梅隆大学甚至远高于这一比例，校方可说是为此作出了特别的努力。但在计算机科学，实际上却只占8%。在数量上，对于现在的女性，没有哪个学科比计算机科学更糟了。“糟”是个错误的词，“低”也许更准确些。

Seibel：说句抬杠的话，难道实现像Anita Borg所设定的目标——“到2020年，50/50 ”（意思是到2020年时要有50%的女性从事计算机科学），就这么重要？难道这一学科成为大众领域就这么重要？

Allen：计算机科学对整个社会而言是如此有变革能力的一个领域。没有多样化的人群参与，我们所做的结果将不可能对社会各阶层的人都具吸引力、都有用。我们所面临的部分挑战，在于如何使计算技术及其所衍生的各领域对所有人皆可用。这是一个理想。不过真的是计算技术发展的方向，像MIT正努力实现的百元电脑计划，我们正努力通过计算技术在不发达国家的边远地区尝试实现初级水平的商务。

Seibel：可不可以这么说，越接近最终用户，越容易理解：有着不同经验的人，在讨论用户可能喜欢如何与计算机交互方面，能带来不同的创意。再说句抬杠的话，如果有人说：“当谈论的是如何设计应用程序时，这当然很好。但当设计编译器优化时，有谁会在乎意见的多样性呢？”你会怎么回答？当你的工作涉及软件深层技术，像优化编译器，拥有多样化的团队人员是否仍然有价值？

Allen：是的。实际上，这是PTRAN组的一个关键要素。很多妇女都被吸引到我们组，部分原因是已经有其他女性在组里。这也使我们成为一个非常融洽的团队。正是因为混合——不是因为有女性，而是由于混合。组里的这些人中有的来自其他组织，还有的具有其他教育背景。

比如，来自纽约大学柯朗研究所的同事们就具有关于如何做事的独特思维方式，因为他们毕业于同一所研究生院。同时还有很多来自MIT的同事，其中有一位女性特别突出，理论深厚、思维独到。来自伊利诺伊州的同事也有某些不同的特征。因此，即使不考虑性别差异和其他文化分歧，仅仅来自不同地方这一事实，就足以保证我们是一个极为强大的团队。

Seibel：*我觉得，假如达到大学本科计算机学科中男女比例是50/50这一目标时，某种程度上其实失去了经验的多样性，如果每个人都得获得同样的计算机科学学位。*

Allen：是什么吸引刚毕业的大学生去IBM之类的公司？我认为，是他们不限于一种学科。他们经常从一个学科转到另一个学科。并且，这些学科虽然常常是十分深奥的技术学科，但却极其多样化。这通常是刻意如此——有人决定公司就是要关联些大领域。我也曾和有着类似思想的人探讨过，他们认为语言学和计算机科学之间有着某种联系。就这一点说，他们很适合来IBM工作。

Seibel：*那么，你觉得"到2020年，50/50"可能实现吗？*

Allen：不抱多大希望。

Seibel：*在你看来，要想达到这一目标，要采取些什么步骤？是不是得改变初中数学的教学方式？在我看来，大多数女孩子正是从初中开始放弃数学和其他理科的，而之前，她们一般都很喜欢数学。*

Allen：很多人都持这种观点，但我并不这么认为。看看西屋数学竞赛（Westinghouse Competition），差不多都是女孩获奖。再说，中学时还是有很多女孩子学习理工科，研究那些艰深的自然科学和数学。我是在纽约克罗顿上的中学，五分之一的人都能拿到西屋竞赛全国奖。学校有一个很好的科学项目，今年高年级组里，七分之六都是在做独立科学研究的女性。

发生在这些女性身上的是，她们正在进入社会相关领域。虽然计算机科学也可以和社会极其相关，但她们做的更多的是地球科学、生物科学、医药。医学会很快达到50/50。很多学科也相信这一理论，但我们并不大认同。

Seibel：*是什么原因导致计算机科学这么没吸引力？*

Allen：很多人认为这个行当就是整日坐在电脑前，了无生趣。社会化网络带来的影响应该会让它变得有趣。不过我也不敢断定。但我觉得这是我们要解决的问题。这并不是要求教育者改变他们的培养方式，我们在这个领域必须做得让它更具吸引力。

我们不得不给这个领域一种身份，远高于它目前所拥有的身份——更人性化的身份。我们没有阐明为什么我们喜欢这一领域，这个领域目前以及未来最令人激情澎湃的是什么，为什么它是值得投身其中的伟大领域。

Seibel：*那么，你为什么喜欢它？*

Allen：部分原因是每天都能产生新想法。总能看到某一想法后，说："哦，这是新的。"整个领域常常都让人有新鲜感。去思考所有这些潜力及其所具有的影响，真是非常令人兴奋。

Issac Asimov曾这么描述计算机行业的未来——我不知道是否如此——他们将要做的一切就是使我们有更多的创造性。计算将启动创造力的时代。人们已看到了这些正在发生，尤其是在多媒体。孩子们正在做他们之前做不了的事情：制作电影、创建图片。我们倾向于将创造力看作是一件个人拥有的特殊礼物，就像欧洲中世纪黑暗时代能阅读和写字就已是特别的礼物一样（在那个时代只有少数人能够这样做）。我觉得，"计算机是创造性的触发器"这一看法非常鼓舞人心。

Seibel：*你已经在许多类别荣列第一女性，如图灵奖第一位女得主，第一位IBM女院士。你觉得有没有别的女性在你之前被忽略了？*

Allen：哦，有，一定有。

Seibel：*所以当你赢得了图灵后，你有没有想过："哎呀，是不是有一个女人应该更早赢得这奖？"*

Allen：嗯，获奖后，我首先想到是，这太美妙了。然后，我开始想到那些做了很多工作但没有得到认可的女性。在许多情况下，她们的工作被剽窃了。我想到了那些做了非常令人称奇的事情却没有得到认可的女性，甚至得不到他们同伴的认可。当我接近她们并告诉她们："你需要加入一些专业机构，我愿意为你写推荐信。"她们羞涩地逃避了。

Seibel：*那么，你认为她们没有得到认可，部分原因在于她们没将自己置于一个易于为人承认的位置。*

Allen：是的。

Seibel：*你现在是否能说出，有哪个人应该获得一些认可？*

Allen：嗯，比如Edith Schonberg，她可是一个伟大的计算机科学家。就技术工作而言，她是第一个写某方面论文的人。然而，她的研究成果被盗取了，

明目张胆地被盗取了。她写了篇有关并行程序调试的论文，这是个非常难的问题。这篇论文在某次会议上并未被录用，但会议组委会里有人据此发表了三篇论文。这确有其事，经常发生在我们的领域，而我们却没有处理此类问题的好办法。

Seibel：这类事情更多地发生在女性身上？

Allen：是的，我想是这样的。人们通常认为，女性往往不愿抗争。她们更为孤立，也没有拥护者会帮其对付有名的剽窃者。他是一个出了名的剽窃者，很多人都知道，但没有人敢管。研发Stretch时，就有过很多此类事件。有一位女研究员实际上是多道程序设计的发明人，但有人抢走了这一殊荣并最终成为图灵奖得主。

Seibel：你是否更愿意获得图灵奖，而不太愿意成为第一个获得图灵奖的女性？很多报纸都报导说“女人赢得图灵奖”，我想这可能是有点讨厌。如果有另外一名女性早在十年前就获奖，是第一位获得图灵奖的女性，然后你获奖，你会更喜欢这样吗？

Allen：嗯，说不上喜欢不喜欢。我能获得图灵奖，也是有不少原因的。这么久才获奖，一定程度上也表明，很多人并不清楚我都在做些什么。一直以来，我都和团队成员一起工作，其中不乏牛人、名人。因此，工作成绩也常常归功于他人，比如John Cocke[①]，他到处传播自己想法。许多人都曾获得表扬和奖励，因为都曾得到他的赏识，大家都一样。

不过，我很高兴能获奖，部分原因在于对于女性来说这个奖来得有些晚。我觉得这是我们这个领域的尴尬，差不多40年来有50位男性获得过图灵奖。因此，我认为女性获得这一奖项绝对算得上是逾期了，而我非常荣幸成为获得该项奖的第一位女性。我不想刻意渲染这件事，而会更注重自己的职业生涯整个历程。

Seibel：你在IBM研究院度过了自己的整个职业生涯，有什么感想？

Allen：在IBM研究院工作是我今生最幸运的事情之一，因为IBM研究院处于工业界和学术界之间。我感觉这就像立足于一面石墙上，可以方便地观察任何一边，从两边寻找有趣的问题和机遇。

Seibel：从“石墙”这样有利的位置来看，你觉得学术界和工业界之间的合

① 编译器优化的奠基人。

作与交流充分吗？

Allen：嗯，几年前国家科学基金会发表了一份美妙的报告，其中有张图给出了资产达10亿美元的几个行业，如图像、互联网、高性能计算、晶体管。这些10亿美元的行业排在Y轴，X轴是时间轴，表示该行业开始的时间、工业界的作用（实验室）及学术界的作用。

一些行业开始于工业界，一些行业开始于学术界。同时，这两类实体在建立这些10亿级产业中的作用相当。我认为真正重要的是保护相互作用，以便有大量跨领域的思想流动、技术交流、方法融合、投资互补来保证其持续发挥作用。

眼下，重点是保持美国创新，它很重要，就交互、携手合作、共同解决问题而言，我认为我们做得相当不错。解决可以把人类从解决问题的桎梏中解放出来的问题——知识产权就是其中之一。

Seibel：IBM并不是完全无可指责。

Allen：完全不是。

Seibel：你一定拥有自己的专利吧？

Allen：没有，这个真没有。部分原因在于软件并不授予专利权。还有就是，我经常工作在事物的前沿，使其进入IBM公司的最好方式是发表它，这样其他公司也将选择它。比起获得专利，我对将其转化为产品更感兴趣。

Seibel：是不是这比说服IBM的人以你的研究为基础构建产品更为容易？

Allen：我们现在有更好的做事方法。但是有时从研究中的好想法到产品之间还有很长的路要走。

Seibel：获得图灵奖促使你对自己整个职业生涯有所反思，你觉得是什么把这一切紧紧地联系在一起？

Allen：想想自己的职业生涯和做事的方式，真要用一个词来总结的话，我认为“探索”最为合适。我喜欢探索一切事物的前沿，创意、项目以至万事万物，包括人类本身，我觉得这一切都如此令人兴奋。

但也有一点，我总是启动者，而非终结者。我总是被新事物所吸引。编译器领域就是一个了不起的领域，因为计算机行业不断有新的挑战，就连那些解决中的问题也越来越具挑战性。

（编辑：傅志红）

14

Bernie Cosell

李琳骁　译

1969年，后来成为Internet核心的ARPANET最早的两个节点上线。当时，在速率为50kbit/s的专线上传输的所有数据包都会流经两台专用计算机，即所谓的IMP（Interface Message Processor，接口消息处理机）。这两台IMP由BBN公司（Bolt Beranek & Newman）设计和制造，其上运行的软件则由三名程序员组成的小组编写[①]，其中一位就是Bernie Cosell，三年之前，刚念大三的他离开MIT，加入了BBN。

最初，Cosell是作为应用程序开发人员进入公司的，参与的项目是构建最早的分时系统。很快，Cosell转做项目的系统编程，没过多久，他就完成了操作系统代码，并让整个系统运转起来，因此赢得"PDP-1分时系统沙皇"的称号。

在BBN公司26年多的职业生涯里，Cosell几乎无所不做，涉及所有产品，赢得了调试和修复大师的美名，即便被派去处理处境维艰的项目，也能让软件运行起来。另外，他也会出于好玩编写些程序：为了磨练自己的Lisp技能，他根据Joseph Weizenbaum的一篇杂志文章，写成了ELIZA的Doctor版本。Cosell的Doctor用BBN-LISP编写，随TENEX操作系统[②]一起流传于ARPANET网络，影响范围比Weizenbaum的原始版本还要广，由此造就了一大批新的实现和相关程序。

1991年，Cosell离开BBN，在弗吉尼亚买下一个牧场，与妻子Lynn以及三条狗、数量众多的猫和羊一同生活在农场里。他间或为当地ISP（网络服务提供商）写些程序，搞点自己的项目，讲授几门编程和计算机安全方面的

① 整个项目的人员组成参见David C. Walden的*Looking back at ARPANET effort, 34 years later*。

② TOPS-20操作系统的前身，由BBN开发。

课程，对自己不再当全职程序员感到心满意足。带有讽刺意味的是，搬到乡下之后，作为Internet奠基人之一的Cosell眼下在家只能拨号上网。

在这次访谈中，我们谈到他是怎么赢得调试大师的美名，编写清晰代码的重要性，以及他如何说服IMP项目的其他程序员不要再给二进制程序打补丁。

Seibel：*你最早是什么时候开始编程的？*

Cosell：高中吧。当时传闻我们学校是全国第一所真正拥有计算机的高中，是否属实不得而知。IBM向我们学校捐了一台1620[①]。我是1959年上的高中，那台1620应该是我进高中那年或前一年捐的。

Seibel：*你在哪所高中就读？*

Cosell：纽约布朗克斯科学高中。我记得前一届学生用的是哥伦比亚大学的650。拥有自己的计算机，我们数学系主任很满意。实际上，他还写了一本编程方面的书，那时这方面的书屈指可数。书里的例子我全都调试过一遍。除了学习编程，我想不起来高中还做过其他什么了。

Seibel：*当时你们怎么编程？用打孔卡写汇编吗？*

Cosell：没错。是的，用打孔卡，另外1620还有控制台。它附带一台IBM Selectric打字机，用作输入/输出控制台，可以用来输入程序。那个时代完全不同于现在，他们没给机器加上算术运算单元。那台机器利用查表法做算术运算：机器里有块内存区域；做加法时，一个数字是行数，一个数字是列数，行列交叉处即为两数之和。所有程序都要加载那块包含加法和乘法表的内存区域。

实际上也可以用打字机键入，不过我们主要还是先给卡打孔，然后装到机器里。它支持Fortran，不过我很少用。我主要用1620汇编器编程。

另外，我在高中还学了怎么连接插接板。从某个渠道，我们弄到一台旧的403计算打印机，我学会了怎么连接插接板。这在当时也算是门原始艺术，后来还真派上了用场。在BBN工作期间，大概是高中毕业10年后，我们真的需要有人连接插接板，我自告奋勇说："哈，把说明书给我。"我读了说明书，改造了一台旧的独立式会计机打印机，让它能执行简单协议，用作PDP-1上的行式打印机（line printer）。

---

① IBM于1959年推出的低端"科学计算机"。

Seibel：高中毕业后，到BBN之前，你在MIT？

Cosell：1963年，我高中毕业，进入MIT求学。在MIT，我学的是纯数学专业，还上了门少见的计算机课程。除电机工程系外，其他系只是偶尔开计算机课，当时还没有计算机科学专业。刚开始，大家在709或7094上构建最早的分时系统，计算机中心有什么机器就用什么，而我还忙着念数学。

我选了几门电机工程和逻辑课程，去听听计算机课，觉得还不错。那时我还少不更事，不知道怎样才算真正优秀的程序员。不过我自觉能够编程。

我偶然加入了技术模型铁路俱乐部[①]。这个社团非常棒。继电器逻辑正是我所熟悉的。俱乐部有个铁道沙盘，完全由继电器逻辑和步进开关控制。我借机认识了几个RLE（电子研究实验中心，Research Lab of Electronics）的人。那时，我们整天都泡在26号楼的地下室，用键控打孔机制作打孔卡，之后交给机器管理员，第二天他会把代码清单返还给我们。随后，我开始经常去Project MAC晃荡[②]。简单来说，我本该上许多数学课，却发现自己开溜去玩计算机的时间越来越多。

在RLE之后，我转而去了Tech Square，在那里遇到了Richard Greenblatt和Bill Gosper等人。不过我只是过客，认为自己做的编程不够多。我还记得自己当时参加Project MAC，其实是被PDP-1上的星球大战游戏所吸引。我并不是以黑客或程序员的身份去接近它的，不是想说"让我看看源代码，你们是怎么做到的？"只是觉得那个游戏实在太棒了。那时，我不过是个游戏玩家，而非程序员，听说Project MAC的那些家伙写了个超强版星球大战游戏，他们的控制台很棒，还有一台闲置的PDP，于是我就老去那儿晃荡。我还碰见了Peter Samson，他雄心勃勃，试图解决纽约城地铁系统问题，凭一张票用最短时间走完整个地铁线路，尽管最后还是失败了。

当时我好像大二，看着这些人都已是行家里手，他们清楚地知道自己在做什么，我却还在忙着对付大二躲不掉的各种事情。我还在写小程序解谜题：青蛙从一个莲叶跳到另一个莲叶，最后跳离池塘。我记得自己写的就是那个程序，另外帮着室友搞定他们写的程序。那就是我当时的状态。我不知道在我完成学业后又会怎么。

回头看来，我觉得那个时候，自己是在学习编程技艺。一定程度上，我可以让计算机做到我想做的事。希望还在。我还没有把握编程技巧，我真的

① Tech Model Railroad Club，黑客文化的起源地。

② 实为一个实验室，具体请参看维基百科相关词条。

不知道发生了什么。它对我而言神秘而陌生。我就这样虚度着大学时光。到BBN工作后，我才逐渐成长为真正的程序员。

我上大学时结识的一位朋友，毕业后到了BBN工作。他对我说："来我们这儿吧。"一天半夜，他带我去了BBN，那地方真是不可思议，一周7天，每天24小时，都有人上班。BBN有点像是MIT实验室的进修部。公司并不限定大家几点到几点走。他上晚班，所以我们是晚上去的。一切都太神秘，太奇妙了，让人看不透，对于他展示给我的东西，我全然不知。不久之后，他把我推荐给公司。接着公司约我面谈，面试，最后雇了我。

Seibel：*你在MIT的三年就是这么度过的？*

Cosell：没错。我大三那年9月去BBN做兼职，记得一直做到10月。之后我休学，到BBN做全职。

回想起来，我并不算太出色。我见过PDP-1却从未想过要怎样写个程序。我对分时一窍不通。当然，这一点也不奇怪，毕竟当时全世界真正懂分时的大概也就50个人。

不过，当时BBN正在跟麻省综合医院[①]合作开展项目，尝试实现医院自动化，我也参与到那个项目中。我本想当名应用程序员，因为我只擅长应用开发。我记得自己大概当了三周的应用程序员，没过多久，就转做系统程序员，开发他们正在使用的库。此后不久，我得到两位系统专家的眷顾，被指定接手他们的工作，他们编写了PDP-1分时系统的大量代码。那年冬天，他们俩离开BBN回学校读研究生。到一月份，我成了PDP-1分时系统的"沙皇"，全权负责整个系统。

也就在那一小段时间里，心中的明灯一盏盏点亮。突然间，我就悟到了分时的真谛，理解了实时系统。理解之后，我就掌握了分时系统，一切都变得易如反掌。

在那个时代，这个项目可谓雄心勃勃。总体构想是每个病房装一台Model 33电传打字机。这种打字机又笨又吵，还只支持大写字母。还要让每间医生办公室也要有一台Model 33电传打字机，药房有一台Model 33电传打字机，住院处好像也得有一台。我们的小型分时系统就负责协调所有电传打字机。

病人入院时，医院会指定一张床位。医生安排病人去实验室化验。到了某个时间，护士的电传打字机会打印："取走这些标本。贴上编号。"实验室则会收到"做这些化验"的消息。如果医生开了药方，药房也会收到消息，

① Massachusetts General Hospital，哈佛医学院教学医院，美国五大顶级医院之一。

准备好药品。

在病房里装这些又吵又笨的玩意儿，真是离谱。让那些医务人员天天面对这些笨拙的机器，确实非常令人不快，因此阻力很大。不过，上述种种问题好像与我绝缘，我满脑子想的都是整个项目的系统部分。

我坚信系统不宕机至关重要。我记不清他们有没有跟我提过，总之我认为我们必须证明，至少我得证明，分时系统能行。证明分时系统够好够牢靠，你才会考虑用它来运营医院。我考虑过各种情况，如果病人需要用药，而系统却崩溃了怎么办？或者更糟，系统弄丢了处方，就再也不给病人服药了？或者系统搞错处方，而护士业已开始依赖这个系统，又该怎么办？因此我开始认为系统不能崩溃。这个系统应当像Unix一样，运行30年也没问题。

但是，当时根本没法实时调试。系统崩溃时，基本上只是运行指示灯熄灭，仅此而已。另外，可以通过控制面板开关读写内存。调试整个系统的唯一途径是要知道“系统崩溃时都在做什么”，你没法运行程序，只能查看记录当时系统动作的表格。于是我去查看内存，并在坐标纸上记录系统当时正在做什么。最终，我对系统调试越来越拿手。

回想起来，我当时特别精于此道，以致他们给我配了个呼机。那个年代有个呼机很酷，只有医生才配。呼机又大又难看，只会哔哔响。只能单向接收，不支持消息。而且，这个呼机只能在波士顿地区使用，因为发射塔就架在普天寿中心（Prudential Center）大楼的楼顶。不过只要我待在波士顿周边50英里内，呼机就能使用。

简单来说，我就像受过训练的小机器人：一听到呼机哔哔响，就回电话查明出了什么问题。令人匪夷所思的是，我在停车场里，没有坐标纸，用投币电话打个电话，让他们先检查八进制地址，修改八进制地址，然后告诉他们：“好，输入这个地址，确认并运行。”那个系统竟然恢复正常了。我都不知道自己到底是怎么做到的。不过，我的确能做好这类事情。我维护这个分时系统大概有两三年。

Seibel：那个时候，虽然已经有现成的系统，你大概还是写了许多代码吧？

Cosell：是的。我拿到那个操作系统程序的时候，整个操作系统还有很多bug，Steve Weiss和Bob Morgan离开公司去念研究生时，还有几块没有完工。最终我做到了他们没有办到的：让系统跑起来，这是让我在BBN小有名气的一件事。

我坚信计算机是可控的，你可以掌握它们应该做什么，而且它没有理由不工作、不正常运行。现在回想起来，我当时特别擅长保证系统运转正常，

添加新代码，同时还不会破坏整个系统。

那是我头一次意外地有了点名气。我知道，老板或许还有同事说过我是个调试能手。这种说法并不全对，里面有些水分。

实际上，当时我是个非常细心的程序员，也有些自大，认为真正难对付的计算机程序屈指可数。我会找些看似无法工作的代码，并设法读懂它们。要是能够读懂的话，我一般都能找出问题所在，并解决这些问题。不过有时我也会拿到一些代码，通常是其他人搞不定的代码，这时我会说："这代码写得太复杂了。"

面对这种代码，我会先弄清楚它要实现什么功能，然后摒弃不用，自己从头开始重写一遍。不过，有些与我共事过的非常棒的程序员，如Will Crowther[①]，并不接受我的做法。他们觉得这么做可能在修正2个bug的同时引入27个新bug。但实际情况是，这种做法我非常拿手。总之我会彻底重写代码，代码结构和原来的程序员写的大不一样，因为我思考问题的角度不同。通常我重写后的代码较之前的更为简洁，或者至少在我看来更简洁，并能正常工作。

因此我赢得了这名声——我能修复其他人搞不定的难缠的bug。幸好他们从不问我具体是什么bug。不然，要是他们真的问"你怎么搞定的啊"，那我就只好说："我不能很好地理解这段代码都做些什么，所以我重写了一遍。"

开发PDP-1分时系统期间，这种事我干过许多次。有很多代码，读过之后，我觉得它们没有很好地实现这部分程序应该实现的功能，还有些代码则看起来就很怪。于是我就重写这些代码。当时，以这种态度，我还能一直留在那里工作的唯一原因，就是我有良好的工作成绩。有些事就是这样，你要是不拿手，就会弄得一团糟。但是一旦你做好了，所有人都认为你能搞定不可能的任务，我这种做法就属于这一类。

Seibel：做出休学离开MIT的决定是不是很难？

Cosell：不是。其实回想起来，当时做这个决定特别轻松。我讨厌上学，上学快把我逼疯了。再说MIT给人压力太大。相比之下，BBN就像是块乐土，奇妙无比。那里的人是在"玩"计算机，整个公司的氛围也很轻松。那里其实比Project MAC还要Project MAC。那个时候人们常常把宠物狗带到办公室。各种宠物在走廊里上蹿下跳，人们则不分昼夜地工作。

---

①文字冒险游戏Colossal Cave Adventure（洞窟历险）的作者。

我一开始是做兼职，实际上我在MIT读书期间一直有兼职工作。一到BBN，马上就有种回家的感觉，真是令人难以置信。我在MIT过得糟糕透顶，于是我休学了，开始到BBN做全职。在BBN稳定下来之后，我也变得更成熟，状态大有改观。第二年秋季，我本该读大四，我重新回到MIT，继续我的学业。我又回到了学生时代。所以这一切对我来说都不错。

Seibel：*你觉得自己在MIT的教育是对工作经验的很好补充吗?*

Cosell：我在MIT念本科时上过的那些编程课，确实让我具备了比较好的抽象能力，但是除此之外，我学到的东西并不多。实际上，绝大部分都是我在BBN工作的过程中学到的。虽然除了Steve Weiss，其他人都没真正指导过我，但是我从每个人身上都学到了自己必需的知识。

Seibel：*在那个年代，计算机方面的图书显然不像现在这么丰富，有没有你觉得特别有用的书，或是你认为程序员必读的?*

Cosell：让我建议程序员现在应该读什么，着实有点困难。其实我很难记起那个时候有什么书直接教人怎么编程，最相关的应该是Knuth写的《计算机程序设计艺术》，那本书我从头到尾读过一遍。不过我并不推荐将这本书用作教材。

Seibel：*你把Knuth的书通读了一遍?*

Cosell：是，那本书很热门，我当时状态奇佳。每新出一卷，我们都会用心去读，吃透每一页。

Seibel：*读这套书要求具备相当扎实的数学功底。你认为大多数程序员都需要像你们那样读懂每一页吗?*

Cosell：我只是拿Knuth的书举个例子。我不会向学生讲授Knuth的书，原因有二。首先，书中那些数学相关的内容并不只是为了呈现算法，主要是推演算法的优劣。我觉得你不需要明白个中细节。我理解其中一部分，但不确定自己有理解这些的必要。倒是对什么算法快、什么算法慢、什么时候算法快或慢的认识的确很重要，即使你不知道到底有多快或有多慢。

其次，一旦学生了解了这些东西，他们会自作聪明。他们开始优化程序里很不起眼的部分，仅仅是因为他们认为："这部分很适合用AB不平衡2-3倍反转回历指针立方之类的数据结构或算法[①]，而且我总是想试试这个算

---

①这是Cosell杜撰的"复杂数据结构"。

法。”于是，对于这种不起眼、根本不需要大动干戈的地方，他们会用上一两个星期去调试，结果除了使程序变得臃肿之外，看不到丝毫改进。因此学生需要用心理解都有哪些算法，它们是怎么工作的，以及怎样运用这些算法。实际上，更重要的是学会如何选择合适的算法完成相应的工作，而不是要记住这种算法的复杂度是n的3次方加3，那种算法的复杂度是n的平方乘4。

他们要是真的对算法感兴趣，知道有Knuth就可以了，但普通人不需要知道。他们还需要懂得其中的智慧。他们要懂得数据结构。他们看到我用Perl构建链表时不应该被吓到。知道所有数据结构之后，你就能挑选合适的算法。不一定要挑选最快的，也不一定要挑选实现最精巧的。对替代做法心知肚明，你才能真正挑选到最适合你的数据的算法。别跟Don提他在那书里为减少组合数学而做的大量可怕的数值计算让我学得很累，但却没有多大用。但是我得声明，我的确学到许多数据结构相关知识，那是好东西。

Seibel：*对众多自学的程序员，你有什么建议？*

Cosell：要编写大量的程序。这对我的确很管用。回想我上过的各种课程，只有编写程序才真正得其要领。不是只为消磨时光而编程，而是有明确目标：“这方面我应该学点什么，为什么我不试着写个小程序做这个呢？”那确实很管用。

只有自己动手实践之后，你才能明白这些东西是怎么工作的，相互之间是如何作用的。自己花了两个星期调试的东西，优秀程序员只用五分钟就能搞定，除非对此有亲身经历，否则你不会知道哪些编程实践危险重重。我认为这些光靠听课是学不到的。听课只能给你大量知识，而编程归根结底是门技艺，只有不断锤炼才能日臻完美。

要是走运的话，你可以在工作中不断锤炼编程技艺。但即使在工作环境中，在具体的工作中学习，我认为要真正变得优秀，你必须学得够快，在工作要求你学之前就学会。除了工作要求你做的，你还得再多学点。如果工作要求你用Tcl做个东西，只学习够制作界面的那点Tcl远远不够。正确的做法是，那个周末就开始写些Tcl程序，这样到了周一早上，用起Tcl来就会游刃有余。

Seibel：*就你而言，出于好玩和有意识地学习特定技术而做的编程各占多少？*

Cosell：通常我把计算机编程看作是做好事情的一种手段，我学习怎么编程是为了做好事情。有些事情在我看来有问题，我能修复它。我认为做些Lisp编程很好玩，不是因为我想学Lisp，而是因为河对岸的几个朋友都是Lisp高

手，而且对我来说，Lisp有些神秘。于是我写了几个程序，这对我来说是再自然不过的事了，我不会坐在Dan Murphy一旁，让他教我CONS、CDR和CAR[①]。

Seibel：对于打算以程序员为职业目标的人而言，你认为计算机科学的哪些领域特别有用？

Cosell：那就多了。我知道许多学校都做得很糟，但我认为面向对象编程的课要上好，重在其抽象的形式。我和当地大学的一些人争吵，焦点之一是应否用C++教面向对象编程。我反问他们，怎么能确定他们的学生真的理解面向对象编程的哲学概念与用C++实现它的套路和古怪方式之间的差别。

我认为学校能做的另一件事是教授Knuth书里的东西。我碰到好多人都觉得链表不可思议。他们对几十种不同类型的树一无所知，也不知道某些类型为何比其他类型更优。他们不懂垃圾回收，也不懂数据结构及相关内容。

然后是下一卷：排序与查找。如果程序设计语言没有提供排序函数，面对不同类型的排序，他们会一头雾水，不知道如何查找目标，什么时候应该建立索引，当下使用的数据库以B-tree存储数据又是什么意思。我认为一门好的课程不是教学生怎么用C写链表，那纯粹是种技艺，而是要让他们从抽象层面上理解链表的作用和操作。

Seibel：ARPANET成形之初，你、Will Crowther和Dave Walden为最早的ARPANET IMP开发软件，这大概是你从事过的最著名的项目。讲讲来龙去脉？

Cosell：在Frank Heart的小组，我们的部门，Frank就靠他手下的这些程序员做所有的事情。他负责挑选和决定怎么在各个项目之间分配人手。我做的项目结束后，Frank会决定我接下来该做什么。那些真正的顾问工程师会开始飞赴华盛顿，撰写建议，而我不用那么做。不知何故，Frank决定让我加入IMP项目，我是第三个加入这个项目的。

1968年秋天，Dave和Will等人已经开始做这个项目，我当时正在做另一个项目。我记得合同已经签了，不过要到来年1月才开始。我加入这个项目时，进展不大。我记得他们已经捣鼓了一部分代码，但还没真正跑起来。我加入项目时Dave和Will已经开始拟订整个系统该怎么组织，并选了几个模块开始编写代码。我直接参与进去，给自己挑了一两个模块。我们各有专长，

① Lisp语言中，car和cdr是对cons cell组成的链表的基本操作。

都知道每行代码做什么，因为整个程序并不大。复杂，但并不大。

我加入之后，才真正明白他们为什么完成的并不多，因为他们还在脱机写汇编。他们要拿着纸带到Honeywell机房里，那里有台516，运行一遍纸带，用上一整盒纸带，生成汇编清单，然后将穿孔后的纸带拿到另一台机器上，因为Honeywell的机器上没有行式打印机，无法打印汇编清单。这么做软件管理实在太麻烦。我为这个项目做的第一件具体的事情就是针对我们的PDP-1写了一个交叉汇编器。

之后我们就可以在PDP-1上编辑文件，汇编这些文件，制作这些文件的汇编清单，并运行TECO宏。最终只需针对二进制可执行程序穿孔相对少量的纸带，然后输入Honeywell机器。

Seibel：*编写IMP软件最大的挑战是让它跑得更快吗？*

Cosell：哦，那很有意思。好，让我们看看。我们不是太在意它的大小，反正系统必须具备大量空间用于缓冲。代码也不会大到放不下。假定代码缩减得不够，比最小尺寸大百分之十，那也只意味着缓冲少了些。因此我们不是太担心总共要占用多少指令。

Seibel：*相对于它会占用的空间大小而言？*

Cosell：没错，空间大小。但是我们非常关心速度，关心我们能否跟上带宽。你该怎么组织整个系统，以便它能优雅地降级，特别是以它能自救的方式降级，而不是直接崩溃，无法使用？

其次是怎么让系统运转起来。有很多东西都没试过、没测过。协议行不行得通？Will提出了一些路由算法的点子，是否可行？还有大量底层问题有待解决。其中一个问题与拥塞控制有关。我们是不是敢打包票，假如世界上所有人都向同一个可怜虫发送数据包，我们真的能按正确的顺序拒绝这些数据包，救他于水火之中？

Seibel：*这么说来主要是因为此前从来没有人解决过这个问题。*

Cosell：说得很对。那个时候，这还是个研究项目，理论居多。许多人都写了论文。很多人都自认为知道怎么回事。那个时候，只有真正用起来才能见分晓。我们必须亲眼见证排队理论是否能用，路由算法会不会出现路由振荡。

第三个挑战就是怎么调试代码。你突然无法与俄亥俄州辛辛那提连线了。哪里出问题了？你该怎么解决？你打电话到俄亥俄州辛辛那提，凌晨3

点叫醒睡眼惺忪的守夜人，走过去看角落里那个一闪一闪的小盒子。他都看到了什么？你怎么处理？即使你将系统恢复正常，到底是哪里出错？你是怎么修复的？要知道，我一直秉持系统不崩溃、永不停顿的理念。

我记得有件事令Will印象深刻，有个bug他们找不出来，不过我找到了。结果发现这个bug出在调制解调器某个协议的处理过程中，它在错误的时间发送了错误的数据包。我打了一组补丁，以便在数据包里加上记号，发现那个特定数据包时，程序就会给系统打上补丁，这样可以查找这类异常事情，一看到那个数据包，就会立即停止系统。一旦整个系统停止了，我们就可以用调试器弄清楚是怎么回事。按上述方式打好补丁后，我大概花了两分钟找到那个bug，因为那个有问题的数据包仍在内存中，一直未被改写。

确切是什么问题我记不起来了，不过那不是个致命问题。有个坏指针破坏了内存，这不会引起任何故障，但是执行成千上万条机器指令之后，整个程序就会因为某个数据结构被破坏而崩溃。但分析表明这个数据结构一直都在使用，因此我们没法加上一段代码，让该数据结构一有变化就停止程序执行。我仔细考虑了一阵子，最后添加了这组补丁。当坏指针破坏了内存时，它会激活另一个补丁来检查其他代码。之后，它会激活另一个补丁，放入另一段代码。然后当它注意到有错误时，它会冻结整个系统。我设法推测出怎样在适当的时候延迟执行，诀窍是动态打补丁，代码的一条路径会被动态打补丁成另一段代码。幸运的是，我猜对了，我们很快就找到了问题。

Seibel：*怎么做才能拥有这种直觉？*

Cosell：对于我非常在行的系统，比如刚才提到的IMP系统，又或者像PDP-1分时系统这种多道程序、多层、中断驱动的系统，一旦掌握了整个系统，我就能在脑海里演绎整个系统的所有动态变化。我知道各种事情应该以什么顺序发生，也知道什么时候不应该发生什么。这样一来，针对“这种事情如何可能发生的”，我就能建立起一个模型。

另外还有些问题涉及两台机器，那也需要一些奇招才能定位。也就是说，我的机器出了点问题，但现象在你的机器上才显现出来。我不知道什么时候停下来，等到你的机器出了问题并反馈“收到一个假数据包”消息时，我的机器已经又处理了6000个数据包。那么接下来该怎么办？我们三个人会一起努力，想办法找出那些问题，并予以修复，从而把系统打造得更加稳固。

Seibel：*你们会嵌入调试代码吗？*

Cosell：不会。

Seibel：*那么碰到许多不同的难缠的bug，每个都得用不同的方法才能追查下去吗？*

Cosell：就我记忆所及，我们没有嵌入任何调试代码。我的意思是，现在，我总是指出写程序时必须让它可测试。让程序可测试的唯一途径是在你动手写第一行代码之前，务必仔细考虑。对那些有效工作的阻塞点、断言点和测试点，你不可能再去改造，如果你急等着程序工作，一开始就把事情做对。

不过，我确定我们当时并没有考虑那么多。我们只是力图写出这个极其复杂的实时系统，而且速度还得快。光这个就已经够难了。我们没做任何真正的一致性检查，谁会愿意把时间浪费在那上面？因此查找问题时打的补丁都是临时的。跳转到一块空闲内存，运行一些专门编写的代码，检查这个或那个，然后跳回去，继续执行原来的代码。

事实上，这种方式还相当固定。其中的一个补丁更新程序，我记得很清楚那是我写的，可以利用它向系统提交补丁，它会从循环链表里抽出一个缓冲，用来保存代码，连接到那段代码，然后再连接回原来的代码。我们过去常常这么做，不过那都是临时应急的。我们会找到某个bug，然后绞尽脑汁，设法弄清楚来龙去脉。

很多时候，只要理解了bug具体是什么，就能定位到相关代码。接着以更为挑剔的眼光阅读那段代码，找出问题并予以修正。另外有些时候，你需要收集更多数据。还有些时候，你也要绞尽脑汁，奋力一搏，设法捕获能给人启发的蛛丝马迹。我们会用到各种方式。

别忘了，我们运行的机器没有控制台，什么也没有。一般来说，补丁都会存储一些数据，然后停止机器。之后我们可能会使用前面板。我不记得我们有能从终端运行，同时还不会损坏机器的调试器。因此我们会通过前面板，仔细查看相应的内存区域，做各种检查，静下心来弄清楚是怎么回事。

Seibel：*也就是说只有一排指示灯？*

Cosell：对，一排指示灯。每个灯表示一比特。

Seibel：*用拨动开关输入地址？*

Cosell：没错。实际上这种机器还好些。PDP-1带拨动开关。这台机器我记得有按钮开关。

Seibel：*你们三个人如何协作？*

Cosell：从我还记得的事情来看，我们风格各有特点。Will是个直觉非常敏锐的程序员。碰到那些大多数人根本不知道怎么处理的难题，他都能从容应付，找到解决办法。

比如，他用Fortran写的游戏Adventure的AI引擎，另外还有IMP系统路由算法及动态变化的各种组件，这些都是Will搞定的。实时系统很重要的一点，是一切都必须设置超时。你不可能永远等待某个事件，实时系统中根本没有永远这一说。

程序各个位置的超时越来越多。我设法弄懂这些超时，耗费了大量时间和精力。我在自己修改的一版源代码中，就试着确定所有超时之间的代数关系。例如，收到消息应答的总超时应该是单个网络包穿过整个网络及其他操作的超时的8倍。或者，消息穿过网络的总超时等于这个网络的最大直径乘以该数据包完成一跳的最长用时。

我试图找出Will在集成系统时心里想的那些基本常量。当两个超时相等时，它们是预设成一样的，还是碰巧相同？谁知道？当你打算修改其中一个常量时，还必须同时修改几处？如果你不时发现等待某件事发生的时间还不够长，而它却超时了，你就知道自己不能单单修改一处超时，因为这些常量都是相互关联的。

于是我写了一大堆井字符定义（即#define），主要是想尽可能找出独立常量的一个最小集合。我之所以还记得这事是因为整个过程真的很可怕。那是我涉足过的一个地方，其中细节几乎没人理解，因为许多常量完全是Will凭直觉设定的，为此这些常量必须一一加以微调，系统才能跑起来。超时不够长的话，我们就把它变长一点，我们这么做不是根据什么第一原理或代数，只是不断微调，直到它起作用为止。

Seibel：*这期间你发现过bug吗？或者你只是想打造更坚实的基础，这么一来需要改动时，修改起来就会相对容易，不用没完没了地重新微调？*

Cosell：我不记得发现过bug。不过，毫无疑问，有些地方的定时器设定的值与之前的有所不同，不过操作上并无重大区别，只是出于防御而做了修改。总之，常量更少，如果必须做修改，你就能比较容易地修改。确实如此，这样一来，程序变得更易理解。程序里四处散布着200个随机选择的独立常量，而且这些常量还与网络的核心有关系，我对此非常反感。我认为修改简化了部分代码，同时让人更容易理解具体是怎么回事。另外我们也藉此用了更多的符号常量。这样就会理解，8倍的直径加上脉冲时间之类的东西。

Will是一个标新立异的人。我记得自己曾为此向Frank Heart抱怨过，我认为他比较适合去做些需要创造性的工作，因为BBN当时在做大量非常前沿的工作，而他非常擅长搞定之前做不好的事情。

他并不擅长编写要彻底敲定的代码。他真正拿手的是编写出能在大部分时间都工作的75%或80%的好代码。我记得当时Will已经转到TIP项目上，Dave和我则继续从事IMP系统的开发，期间我重写了路由算法，因为算法里头有些很奇怪的常量，我理解不了。Will的路由算法仍保留不变，不过我用自己的方式重写了一遍。我觉得自己的版本更加稳固些。至少我理解它是否会振荡，为什么会振荡，因为是我让它振荡的。

我和Will Crowther截然不同的一处是，他相信重组程序时引入的bug比修正的还要多——就算我花了很多时间干活，他还是疑虑重重。因此他惯于保留一页页全是补丁的笔记本。在不得不重组程序之前，他会尽可能先对现有系统打补丁。那些补丁都是补丁之上的补丁，非常复杂，以至于他的不祥的预言总是自然应验。这么打过补丁之后，很难再将程序搞对，结果程序真的出现了补丁要修正的bug。

Seibel：*你手头有最初的源代码清单，可以输进汇编器？*

Cosell：对，还有正在运行的二进制映像。然后我们会用纸带，有时直接手工完成，在这个地方植入跳转，转移到一块小的区域，其中三行代码会被另外五行代码替代，然后又跳回去接着执行后面的指令，因此当你执行这块代码时，它会转到补丁上，执行若干指令，然后再返回。

Seibel：*这么说来，纸带上保存的是补丁的二进制版本？*

Cosell：是的。后来，当我构建小型交互式调试器时，这个调试器具备我非常喜欢的检查和保存功能，实际上我们可以构建一个小的文本纸带，比如，“转到地址12785，值，值，值，值。空行。转到12832，值，值，值，值，值。”如果必须从头装载程序，那么你就可以装载程序，装载完成后再装载补丁纸带。

Seibel：*也就是说，那个时候，你其实没有源代码，可以汇编成当前状态的二进制文件？*

Cosell：千真万确。我们碰到的一个麻烦是我们有多个不同的清单副本。其中一个清单会在代码某个位置用墨水标记，删去两行代码，用旁边的代码替代。那么现在是不是所有清单副本都做了这个修改？Will一清二楚，因为他

有笔记本在手，最终不是哪个清单而是他的笔记本说了算。这就是他的做事风格。

我的做法是系统始终应该能够立即运行。我不愿意在汇编清单上做记号。当我一开始参与这个项目时，整合Will的所有补丁困难重重。白天我们工作一整天，而我要通宵编辑和重组系统，这样第二天早上就能得到另一个干净的磁带，然后我们再以此为基础开始一天的工作。结果就是，经过通宵干活虽然只改了两三个地方，而且位置很明确，但是因为这个修改你可以阅读代码了，代码也变得有意义了。当然，这马上就稳定了下来。

因此我们几乎没再碰到修正一个bug的同时带来一个新bug的问题，除非那个补丁本身就有问题。不过Will和我在这一点上的看法并不一致，他真的非常喜欢打补丁，只要可能就避开汇编器。部分原因是用汇编器非常耗时，而他可以打补丁，就这样继续下去；还有部分原因是编辑起来太可怕了，导致他不相信整个周期。

Seibel：*你是否认为从事的IMP开发是自己的一项重要技术成就？*

Cosell：说来也怪，其实这算不上。这个程序的确有意思，很难对付，不过我之前已经写过Doctor，做过Lisp开发，另外还是医院计算机系统的“沙皇”。当然，那个时候我干过的最漂亮的活儿是弄明白了这个前沿分时系统的每一行代码。而这不过是个小型的独立通信处理器。它不像PDP-1那样拥有那么多中断通道。当只有32个交换时段（swapping slot）又有40个人登录系统时，你又该怎么做，而它对于这种情况是不用处理的。

我们三个人相处得非常融洽，可以说既有趣又有挑战性。IMP系统的调试和实现都很有难度，但我并不认为这是我职业生涯的顶点。它只是又一个程序。另一点令IMP系统少人问津的原因是它的传播范围有限。总的来说，PDP-1也很难。它是个分时系统，必须不断演变。

IMP系统令人称奇之处，在于我们是以这样一种科学的方式来开发系统的。他们从一月份正式启动项目，我二月份加入，最终于九月完工。谈不上真正“完工”，之后我们仍在从事该系统的开发，修正bug和完善功能，只不过它于九月发布，此后也并未停止开发。不久之后，Will转到下一个项目，Dave和我继续做这个项目，另外还有新人加入。

有个人我必须好好感谢一下，就是Frank Heart。我不知道他是怎么想到那种管理方式的，基本上就是任我们所为。我很难想起有过软件评审会，也很难想起曾为文档编写等事烦心过，那时我们三个人满脑子想的都是那个程

序，不能有太多那种干扰。他信任并相信我们仨能够做好这件事，因此基本上不管我们，听其自然。回想起来，作为项目经理，那么做实在叫人吃惊，真是匪夷所思。不用开每周内部例会，也不用绘制PERT图。当然，Will记录了需要完成哪些，我们发现的bug，等等，不过令人印象最深刻的是这些都没人监督。把我们凑在一起，径直告诉我们做这个，我认为这堪称管理上的勇敢创举。

Frank做的另一件事是针对其他项目做的设计评审。他主持的设计评审最让人提心吊胆，实际上之后我也一直实践这一理念。在他的设计评审会上，人们会不寒而栗。这有点像参加自己的论文答辩。他会精心挑选一拨人作为听众，而你则必须介绍自己的设计。他挑选的人都很和善。他的设计评审之所以吓人，是因为他一眼就能看穿你是不是在糊弄。

我相信你一定参加过这样的设计评审：其中一部分设计你做得并不到位，于是想快速掠过。你认为自己会把这块做好，但又没有好好做过分析，因此你并不是很清楚具体怎么回事。他有种直觉，或许也得益于现场那些优秀的参会人员，一旦发现你开始糊弄，或者觉察到你没有仔细考虑过，他都能把你抓个现形。

你做得很好的部分很少有人提及。我们都会说“哦”一语带过。但你最没把握的那部分，我们会全力关注。我知道有些人对此心存恐惧。问题在于，如果你是个没有安全感的程序员，你会把这看作是攻击，觉得大家认为你不称职，你的生活会一团糟。

偶尔，我也会想到好的一面，实际情况并不像你想的那样。设计评审的目的在于帮助你彻底搞对程序。对于你已经做对的部分，我们没什么好帮的，而现在你得到的是，BBN公司最聪明的四个人在帮你修正欠仔细考虑的那部分。告诉我们你为什么考虑得不够仔细。告诉我们你是怎么想的。哪里做错了？我们有15分钟时间，我们能帮到你。

作为工程师，你需要对自身技能有十足的自信，才会说：“好，太好了。我的问题出在这儿。我想不出该怎么做，真希望你们这些家伙不会注意到，这样设计评审时你们就会让我通过。”隐含的回答是：“当然你会通过设计评审，因为看上去都对。既然所有厉害的角色都在，让我们一起搞定那个问题，省得你再折腾上一两个礼拜。”

开展设计评审，目的在于复查他认为做对而且确实做对的那几部分，另外还可能针对他没做对的部分给他一些启发。我领会到这点，那时我只有二十一二岁，觉得那无疑是对的，充分利用高级人才来做评审。

当然，面向客户的设计评审截然不同。面向客户的设计评审只能是："我们全都搞清楚了。我们会做到一切让你满意为止。"内部设计评审则是一次良机，我一直都很奇怪，为什么这多人对设计评审避之唯恐不及。这些人都很优秀，不过他们总是找借口，说什么"我的设计会被弄得七零八落"。很难说服他们，只要设计有可取之处，绝不会被弄得七零八落，另外评审人员也不会心存报复。他们只是想方设法要延续BBN把事情做好的优良传统。

另外很难直接告诉他们，在你的职业生涯中，你再也找不到这么一拨人乐意花上一个小时，帮你彻底弄清楚自己的设计。错过这种机会，你就得全靠自己了，那本应是非常美妙的体验。

Seibel：*设计评审多久做一次？是在项目开始阶段，还是贯穿各个阶段？*

Cosell：一个项目不会做多次技术评审。基本上，只有在认为设计完成时才做一次设计评审。

Seibel：*这么说来，设计是在真正开始编码之前完成的？*

Cosell：是的，没错。那时可能已完成部分编码工作，因为许多人，包括我在内，已经开始堆砌一些代码，确认自己的想法是否切实可行。不过一般来说，我们都有固定的周期，我们必须先提出一些目标，然后寻求资金支持以实现目标。因此我们的任务是向客户提出目标："这是我们打算做的。"你需要彻底弄清楚，因为这时客户会给你那么多时间和金钱，并期望这能行。基本上就是在那个时候，我们得定妥整个提案，我们得从技术上描述打算做什么。接着我们会坐下来开展设计评审，确认我们都弄明白了。合约一旦开始履行，我不记得Frank是否介入过。至少在我从事的项目中，我不记得Frank是否参与过项目评审。

Seibel：*你刚才提到了Doctor，那是什么？*

Cosell：在我开发PDP-1分时系统期间，Dan Murphy和他几个朋友正在PDP-1上开发这个Lisp系统。于是我觉得自己有必要学习Lisp。那年春天，Joe Weizenbaum在《ACM通讯》上发表了一篇关于ELIZA的文章。我觉得那实在太棒了。那时我相信，现在也是如此，只要是我能理解的东西，我就能让计算机去做。他介绍了ELIZA的工作原理，我对自己说："我相信自己可以写个程序实现这功能。"于是我开始在公司内部Dan Murphy的PDP-1系统上编写Lisp程序。我的PDP-1机房里有台Model 33电传打字机，与Dan Murphy的PDP-1相连，因此我可以从自己的机房登录到他的计算机上，假装正在自

己的系统上工作。我写好那个程序，并让它跑起来。折腾这东西完全是在BBN内部展开的。人们会给我留言："要是做到这个就更好了。"或者："我试过这个，不过不起作用。"其实，那是在帮Weizenbaum传播他的想法。一开始它采用PDP-1 Lisp编写。不过那时他们正在PDP-6上构建Lisp，也可能是PDP-10。这个Lisp系统在ARPANET上传播开来，结果Doctor也随之流传开来。

Danny Bobrow写了篇题为"A Turing Test Passed"的文章，里面提到我的Doctor，这让我有了一点点名气。那是我第一次真正被注意，还是因为自己笨拙的举动：我忘了关Doctor。BBN的一位副总来到PDP-1机房，误以为Danny Bobrow拨号登录在计算机上，而他自己是在和Danny交谈。对于我们这些玩过ELIZA的人来说，我们都熟悉那些在机器中预先加载的回答，不会觉得它们是人类的自然语言。不过，对于不怎么熟悉ELIZA的人，那些回答似乎非常得体。尽管那些回答令人不快，不过这个副总真的以为对方是Danny Bobrow。"不过，再跟我多讲点……""你先前说过你想去客户那边。"诸如此类的回答从上下文来看都讲得通，直到最后他键入一些话，但忘了敲继续键，结果程序没回答他。他以为Danny断线了，于是他打电话给还在家里的Danny，对Danny大声吼叫。而Danny彻底懵了，根本不知道怎么回事。只有Danny知道我的终端。他赶到机房，撕掉机器上输出的对话记录纸条，保管起来。

相比于Weizenbaum的ELIZA，我的版本更灵活。我们稍稍改进了脚本。几代的黑客都曾致力于这个程序的开发。我说过，它在ARPANET上传播开来。我猜现在有个版本是用Emacs宏写的。不过那是我成为真正的Lisp程序员的一次实战练习。

Seibel： *据我观察，通常编写最复杂最令人抓狂的代码的程序员都能在脑海里保存大量细节，对此我十分好奇。很显然，你不仅能在脑海里保存大量细节，而且还特别用心将代码写得简单清晰。*

Cosell： 我得承认我两样都做得不错。我会尽量让事情整体上保持简单。不过说到程序应当简单，并不等于这个程序功能的某几部分一定很简单。我也可能会写些非常复杂的代码做些适合的事，那些代码人们唯恐避之不及，碰都不愿碰。但这些代码一般都会集中在一个地方。

我碰到的大部分差劲的程序，也就是那些我摒弃不用，自己重写的程序，复杂代码都不是集中一处以方便理解和修正；相反，复杂代码散布在程序各

个位置。

我总是想方设法让人们铭记两条准则，通常是针对那些刚从大学毕业的学生，他们相信自己已经掌握精通编程所需的一切。第一条准则是要相信没什么真正难的程序。如果你正在读一段代码，觉得它很难懂，假定你无法搞懂这段代码本该实现什么功能，那么基本上就表明这段代码考虑得不够仔细。这时，你不用捋起袖子，试着修改那段代码；不妨后退一步，再仔细考虑一下。当你确认自己考虑得足够全面细致，就会发现这很简单。

最近我们刚好在工作中实践了这条准则。当时他们正在开发某个设计非常复杂的项目，而且越变越复杂。于是我们开了一次会，着手分析整个项目。我说道："那个好像太复杂。"出乎意料，我们于是就有了一幅整个工作机理的功能框图。所有人都被震惊了，因为大家一下子就弄清楚了各个模块是怎么实现自身功能的。我们并没有做那些无趣的活儿，比如必须把所有东西写下来，他们确实明白了接口都很清晰而且可以继续开展工作。这个项目我跟踪了很久，知道其中有些问题很难，但为数不多。有一点总是如此，他们努力用心地考虑，它就变得简单了，于是突然之间，进行正确的编程也就轻而易举了。

另一条准则是要意识到程序是写给人看的。尽管我对自己早期写的好几页TECO宏心存愧疚，大概是在我从事PDP-1分时系统开发，开始理解分时系统的复杂性的时候，不过很快我就转变了观念，坚信计算机程序源代码是为人而不是为计算机写的。计算机才不在乎代码什么样。我认为Perl既有"if"又有"unless"这一点令人称道。因为当你有种直觉，某件事应该做什么的时候，会说"如果（if）某个条件不成立"，其言外之意并不是"除非（unless）条件成立"。

计算机要的是二进制比特流，而我要的是文本文件。我会找人参与我的项目，那些人聪明，非常优秀，刚从大学毕业，而且在他们班上排名前几名。他们对编程了如指掌，我会把项目的某一块交给他们做。我们在项目评审会上就会开始交锋。他们会说："为什么你总是抱怨种种做法，比如我在这里使用全局变量，我没做这个啦，你不喜欢子例程的设计方式？不管怎样，这个程序能用！"

听到我的回答，他们会目瞪口呆。我告诉他们："我并不在意这个程序能用。你在这儿干活本身就意味着我认为你有能力编写能用的程序。编写能用的程序只是门熟练技能而已，这是你本该擅长的。而现在，你必须学会怎么编程。"其中有些家伙是非常优秀的程序员，但是别人写的代码他们从未

读过一行。事实上，有些人甚至从未读过自己写的代码，因此他们永远也体会不到六个月后代码读不懂之苦。

有些人会抗拒。有些人则更加坚信他们自己是出色的程序员，而我只是个已过壮年的老家伙，并不了解他们在做什么。我知道自己会重复不久前说过的话。“这个程序能用。哪里有问题？程序能用并不会给你增色。我们要更进一步。能用的程序是基本要求。”他们会说：“哦。”然后他们同其他人交谈，发现总的来说那是BBN标准。如果你的技艺不够好，没法让计算机实现你心里的想法，你就没办法探索新想法。

如何看待全局变量，怎么组织子例程，我有自己的偏好。有一次，我和一个家伙争执了好几天，他说：“瞧，它跑得好好的。”他是个很出色的程序员，我不愿利用职权来压他。我认为重要的是，让他明白我并不专横，我之所以要他换一种做法完全是有原因的。那时他还意识不到，要理解只包含一个有42页长的C子例程的程序有多难。

Seibel：呀！

Cosell：为此我和他争辩了一番，我非常推崇只调用一次的子例程，它的唯一作用就是抽取上级子例程的某一部分代码。当你阅读上级子例程时（这是我的编程方式），读到这块代码时，你会因这个关系繁杂的部分细节而注意力分散，这时我会抽离那整块代码。之后你就只会看到一个函数调用，“对这个表进行排序，并找出最佳路径”，即使只有这个地方调用该子例程。优化代码的人会说：“这不应该写成子例程。直接把那些代码写在这里。”但这个小的子例程，我是用来隔离代码的。输入也很明确。你可以查看算法，而且只与那个算法有关。我经常会说：“你的例程太过复杂，牵涉整个设计的几大模块。”他总是嗤之以鼻，回答说：“那很好啊，我用一个例程就能搞定它。”

他一开始抗拒但最终接受了我的做法。那时，他的下一个任务是从某位程序员写的早期项目里提取大段代码，整合到我们的系统中。他大概做了一个星期。他十分讨厌那个人的程序，找我的上司抱怨说，部门里的编程标准不够严格。那个人按他自己的想法编程，但和我们不是一路。因此，他切身体会到非常高产非常优秀的程序员没给程序分段会出现什么问题。你会得到一个非常长的程序，并不是说程序全是面条式代码，而是这程序写成这么一长串，层数太多，太过复杂。他差点把我惹毛了，我提到过，他直接越级找我上司，要求部门必须制定标准以杜绝类似问题。

Seibel：*他没有意识到自己之前的代码有可能与同样的标准相抵触？*

Cosell：不。他意识到了。他转变了做法。这就好比有人戒了烟，最痛苦的莫过于别人还在抽烟。他成了我项目中最厉害的一位成员。每当我不够仔细，妥协让步，他总会唠叨我。我的项目是他做过的第一个这种类型的项目。对他来说，通信、实时等等都是全新的。不过，他很聪明，经过这次小小的顿悟，成了我期望的那种程序员。上次我还听说他做得很不错。这在他身上起作用了。有些人不喜欢和我共事，觉得我太蛮横。我不知道为什么。

Seibel：*对于注释的多少，你有没有特定的规则？*

Cosell：我不会在代码里添加太多注释，在我看来，你编写的代码就应该是可读性强，能清晰反映你的算法和想法。我会添加注释说明这个例程应该实现这项功能，通常还会简单描述该例程怎么调用，出现异常时该怎么办，参数以什么顺序传递，等等。代码本身应该清晰表达你做了什么。

只有当直觉告诉我这段代码能用，但没有清晰表明我打算实现的功能时，我才倾向于在代码里添加注释。如果有段代码写得与标准表排序代码有所不同，因为这样一来，我可以利用某种技术，那么我就会在代码里添加注释："这段代码对这个表进行排序。"

我从来就不是结构化程序设计的拥趸，每个子例程开头非得有18行注释，传递的参数必须按正确顺序排好。我切分程序的方式并不总是一成不变的。我写的子例程有的复杂，有的简单。不过，我确实很在意代码排版等事项，我会为大括号的使用而喋喋不休。

其中一个原因是，我阅读代码是为了理解代码做什么，而不是探究每一小块代码的细节。例如看到if语句时，我会在意判断条件。接着，我要考虑那个条件成立或不成立的情况；若想跳过那个if语句，自然喜欢程序组织合理，好让双眼能直接定位到if语句末尾，不用去理会中间的诸多语法。总之，我属于守旧派，喜欢左括号右括号对齐排好。

如果去掉第五列之后的内容，我的代码大致就会变成"操作符，左括号，右括号；操作符，左括号，右括号"，这样我就能看清操作符序列。另一部分则跟我刚才提到的有关。如果左括号和右括号离得太远，那往往表明其中代码过多。碰到这种情况，我可能会把这些代码拎出来。有时，即使代码不多，我还是会把它拎出来，因为要是有太多无用的代码堆在一起，我就没法理解那个小的分支做了什么。

我会尽全力隐去无用的代码，将其移至某个地方，这样我就能跟踪代码

流程，在头脑中建立代码实现功能的全景图。有些编程风格，我读起来很不自在，因为试图理解代码块结构时很费力。有意思的是，Python语言的设计者显然也持类似的想法。他彻底消除了语法上的论战，因为Python根本就不这么用左括号和右括号。当你看到if语句时，总是隐式地伴有左大括号和右大括号，需要查找下一块代码时，只要找到与这个if对齐的那行代码即可。我使用的C和Perl编辑器，只要点击按钮即可收缩整块代码，只查看外层结构，想必Python编辑器也支持同样的功能。

我不会因为一种风格难看而反对这些风格。我相信我反对这个风格是因为它干扰了我对代码的理解。我向来都很擅长理解代码。除非你能说服我你比我更擅长理解代码，否则很难说服我你的做法更好。

Seibel： *毫无疑问，理解陌生的新代码并进行调试，这项技能不是所有优秀程序员都具备的，你好像就具备这项技能。*

Cosell： 是的。这包括两个方面。另外有个叫Steve Butterfield的家伙，也是擅长修复代码，但和我完全是两种人。Steve不用知道程序如何运转的任何线索就能修复程序，他大概是我见过的最擅长此道的人。他能够快速进入一个程序，修改程序内部几段别扭的代码，使之做些不同的事。即使面对庞大复杂的程序，Steve也能迅速上手，修复部分功能，在我看来，代码在功能上更好了，不过情况更糟了。

我总是尽力去改善整个程序，即便是个小程序，我也会设法理解整个程序。我会尽量通过自上而下的推理和查找来找出问题，而不是直接说："哦，这程序出问题了，这里得做个手术。"因此，即使有更直接的做法能修复程序，但为了弄清来龙去脉，有些事情我也会花费大量时间。

但是，通常Steve离开项目之后，就很难修改那些代码以实现某些功能。而我则尽力保证程序良好，但这也意味着，如果程序真的非常庞大棘手，在我觉得可以自如进入程序之前，我可能会花费不少时间死干活。不过还好，在我调试时这种情况不是经常发生，我排查错误并不是全靠调试。

正如前面我提到的，有许多bug真正出在什么位置，我根本无从知晓。有时候，我只是觉得："这段代码应该实现这项功能。但它看似又没实现。我的意思是，怎么会有人写这么复杂的代码去实现这么简单的功能？"于是我删掉那段代码，用自己写的例程取而代之，实现我认为那段代码应该做的简单功能。之后程序居然还能工作。回头看来，之所以出现这种情况，是因为程序不断演变，这个小例程一直改来改去。在此期间，开发人员并不是去

替换这个例程，而是不停给它打补丁，添加各种不同的功能，结果忘了初衷。

我从不调试这种代码。我会花上一两天时间修改代码，按照自己的想法实现一遍，没人知道我在干嘛，而程序最终会得以修复。我是这么调试程序的！这么做非常危险，说实在的，Will的格言也很在理——重写一百行代码，你在修复一个bug的同时很可能引入六个bug。对于那一个bug，至少你知道查找的是什么问题；而现在，你得开始查找六个新的bug。我真是够幸运的，因为这么多年来我一直记录良好，编写的代码绝大部分都能正常工作。

Seibel：*那么你阅读代码时肯定讲究一些策略。即便不存在bug，也要处理大量代码，你是怎么应付的？*

Cosell：其实这方面我做得不太好。我之所以倾向于重写而不是修复大块代码，原因之一是经过一番努力之后我还是理解不了。我不会像读书那样阅读代码。我会设法确定这个程序是做什么用的，然后自顶而下阅读代码找出相关线索。

阅读程序代码的同时，我会想着自己会如何解决这个问题。这意味着我正在查找某些特定代码，这样我就能说："哦，程序就是在这里实现这功能的。"然后，我就可以自信地断定，写这代码的家伙弄错了。或者，至少现在我理解了他们采用另一种方式来实现这个功能。

因此我会自顶而下开始阅读。不过，我认识的几个家伙特别擅长自底向上的做法。他们会从阅读小的子例程开始，最后找到他们需要的那个子例程。碰到这种活儿，我主要采用自顶而下的做法。也即，阅读程序代码时，我总是会设法弄清其他程序员应该做的事情。有时我也会借此修正一些bug，尽管我并不知道具体是什么bug。我读到一段代码，会想："按我现在对这个程序的理解，这段代码应该做这个。"然后，要么我看的那段代码没实现那个功能，要么这代码太过复杂，好像还做了其他六件事，我觉得那样没什么意义。

不管哪种情况，那个时候我通常的反应就是修正那段代码，好让它符合我的想法。你会发现我的做法有多么大的危险，因为正确组织程序的方式不止一种，如果这个程序组织非常合理但采用的方式和我预想的不符，我就会毙掉该程序，接下来就有大量东西要修正。还好，在这方面我非常幸运。通常我说"这看上去不对，我准备修正它"，我还真的就把它修正好了。即使在我编程的早期也是如此。

PDP-1分时系统是我从事开发的第一个大型程序，之前我只是个新手程

序员，只做过大学本科生水平的编程，通过那个医院项目，我很快从应用开发转入系统程序员行列。尽管我当时只做了六个月的专业程序员，我还是断定那个小型远程进程交换程序还不够完善，我要重写这个程序。

Seibel：*除了存在引入新bug的危险之外，另一个风险是你有可能误解程序本该实现的功能。*

Cosell：没错。我选择的道路，它并不适合内心脆弱的人。那个时候我只有19岁，那似乎是做事情的唯一方法。我有两个信念，对我帮助很大：程序应该能被理解，另外真正非常难的问题少之又少。看似很难或棘手的代码多半是因为程序员没有充分理解自己需要做什么，结果他们就靠反复试验，直至写出看似正确的代码。

我不知道自己怎么会有这两个信念。我进入BBN时身无一技之长，不过不知什么原因，我心中已经有了这些原则。我认为自己应该什么都能理解，什么事都不会那么难。我发现，即使对于诸如分时系统和IMP这个级别的程序，也不例外。通常只要我正确理解了程序应该做什么，一切问题就能迎刃而解。不协调的代码会非常显眼，就像拼图游戏中颜色不对的图块那样。

另一个原则是我总想让代码保持干净。我就想把事情做对。当你必须修正程序里的bug时，千万不要头痛医头，哪里找到bug就只修正哪里。我的准则是："如果你那个时候就保有现在自己对这段代码有问题的想法，那你会怎么组织这段例程？"你当初什么地方考虑有误？那就在那里修正代码，保证代码不再出问题。在你修正好例程时，我想看到的是修正好的例程就像是当初写的，只不过这次写对了。我不想看到任何事后想法的迹象，也不想看到事情出错，后面跟着某段代码修正这个错误，或是有段神秘的代码，在一旁附注："这个例程偶尔返回错误值，为此我必须加以修正。"我不想看到这类注释或说明。我想看到代码天衣无缝，你一开始就把它做得正确无误。

然后我会辅之以另一个小技巧。这是我在从事美国国防部项目开发时学会的。他们从不资助新项目。BBN和政府在当前项目里已经投入太多资金，即使那时它还有诸多糟糕的限制，亟待修正。最普遍的问题是程序启动伊始还是正确的东西，随着程序的使用或需求或其他内容的不断演变，现在居然出错了。你要做的是扔掉那部分程序并予以修正。而他们会问："那会有什么改善？"你回答："这不会有任何改善，不过下个星期这个程序会变得更好。"他们不会允许你那么做的。

我采取的是偷偷摸摸的做法，我一直非常受用，在许多程序上屡试不爽。

我会设计程序的下个版本。根据我现有的了解，这就是程序该有的样子，不过是在程序层面，而不是在子例程的层面。现在，当修正bug时，如果可以选择怎么修正，一定要让它向更好的模型靠拢。修正时不要一味走捷径。不要只按它原来的方式加以修正，而要向其他模型靠拢，这样数个月后，整个程序也不至于深陷补丁的泥潭，忙于修正那些顽固不化的错误，程序所有关键部分突然间焕然一新。通常，最终采用老一套做法的代码所剩无几，你可以悄悄进去，修正剩下的那些代码，因为现在你不会破坏整个程序。

因此，当他们问道："完成这次修改要用时多久？"你有三种答案。其一，自然是那条捷径，只修改那一行代码。其二，采用我的简单规则重写子例程，同时还不会犯那种错误，要用多长时间。其三，如果你是在更好版本的程序里编写这个子例程，并修正那个bug，又要用多长时间。于是你会估算大概需要后两者用时的某个中间值，这样每次拿到任务，都会有一些额外时间允许你把程序写得更好。在我看来，这会起到难以置信的作用。程序会以干净利索的方式不断演变。真是太妙了，程序仍算是版本一，不过它就像是乔治·华盛顿的斧子。这下程序整修一新，因为所有关键部分已得到修正，不用劳烦项目经理亲自指派你摒弃问题代码并修正bug。

Seibel：*你听说过重构吗？*

Cosell：没有，那是什么？

Seibel：*就是你刚才描述的。我觉得现在重构的接受度更高一些，甚至项目经理层也是如此。*

Cosell：哦，那真是不错，而我过去可是需要借bug之名——过去修改代码经常需要找个理由，做你刚才说的重构，如果重写只是为了让代码更清晰，那可能绝不会获得准许。因此我必须等到出现bug或有人提出改进要求，才能修改那部分代码，但到了那个时候我只会按着要求来做。我猜所谓的重构是你要花时间思考何为正确的目标，不然的话，不同的人瞄准不同的方向，或者目标根本就不对，那就不可能做好重构。

我从未给这取过名字。这好像是我达成以下两点的唯一途径：管理复杂度，以及得到不用摒弃再重写代码的程序。这是PDP-1教给我的。它运行了很多年，而且工程浩大。最初两个版本是由三四名程序员写成，不可能摒弃不用，但又必须变得更好。

Seibel：*你是怎么聘用程序员的？如何识别人才？*

Cosell：我从未参加过标准的面试环节。人们经常谈到要给面试人员出些小问题，并叫他们现场解决，我听说微软在这方面很有一套。我好像更多是凭直觉去做。我会快速过一遍面试人员的简历，大体感觉他们和我是不是属于同一类人。通常简历没什么用处，毕竟他们只是即将毕业的大学高年级学生。你必须看得透字里行间，有些看似很炫的项目实际上只是某门课程的课堂作业。不过，我常常会跟他们交谈，亲自观察，看看他们是否具备我想要的素质，我总是希望身边的人能够刨根问底、勤学好问、一丝不苟。

他们有什么其他兴趣，业余爱好？他们有无表现出快速学习和好学求知的天分？我这些做法有些粗枝大叶。我心里对符合BBN要求的人有个理想化的形象，大致是有天分、勤学好问、学习快速、爱好广泛。我过去常会跟踪确认自己的印象是否对路，比如这个人是不是真正符合BBN的要求。

Seibel：*你刚才提到，微软在考智力题上很有一套。你也喜欢智力题。用这个来衡量一个人的潜力，你怎么看？*

Cosell：我认为，精心挑选的智力题有可能测出一个人的潜力。不是因为这个人解出了智力题，而要看它能否反映出他们如何组织安排最终解决问题的过程。我从没用过智力题。当然我也不会给人出个拼图智力题，观察他们怎么把它重新组合在一起。问题在于大量这类智力题的解决方法可能迥然不同，你要么知道要么不知道。这种考法不够稳妥，因为我不想招到那种人，他们貌似对拼图游戏很在行，实际上只是碰巧知道一些解决拼图智力题的窍门。

BBN在探索未知领域方面投入了大量时间，有些事情可能从未了结，有些事情我们也不知道怎么做。而这既要有一定的勇气，因为这很不简单，还需要具备一定的技能，以免失败。那才是我一直寻找的人，不是寻找技术娴熟、能解决几个智力题的人，而是看他们面对这么复杂的事情，能不能想办法搞定。

这里有个很好的例子，与当时出来的三阶魔方（Rubik's Cubes）有关。我们刚好听说这个奇妙的智力游戏，有位同事恰好在英格兰出公差，捎回一包魔方。没有参考书，没有文档，当时在美国还不流行。那只是个奇特的群论小游戏。我们开始玩起来。我们几个人解题的方式各不相同，但有意思的是我们都能够搞定这个游戏。那时候BBN的人确实脑子好使。那就是我过去要找的人。

我不太了解微软的小测验，我还听说Google也有能力测试或类似测验，

我不知道这些是否能给你提示，表明这个人具备相应的才智。但那是我过去常常寻找的。这个人是否为成为符合BBN要求的职员做好了准备？通常我得到的是否定的答案。他们都非常出色，都是很棒的工程师，但是在我们交谈过程中，我没有发现闪光点。我的做法是寻找闪光点，至于怎么做到的，我也不得而知。

Seibel：你认为编程是年轻人的行业吗？

Cosell：我觉得这也许是事实。回想我在BBN工作的最后几年间所从事的项目，我甚至发现安排其他人干的活，我自己倒未必能搞定。我有个下属提议用Tcl实现部分界面会是不错的选择，于是他花了一天半时间学会了够用的Tcl，并用Tcl实现了这部分界面，并且让它跑了起来，我觉得自己不一定做得到。有时我会自我解嘲："哎呀，以前我做这种事情根本不成问题。"

我认为，真正的产品代码，就是运转正常、合乎逻辑的良好代码，在理解新事物方面要求做到充满激情、头脑敏捷，我发现自己现在很难做到。从另一方面来看，对于各种问题，你积累了一定的智慧，这些你年轻时显然还不具备。现在我更清楚怎么做事情。综合来看，我觉得自己更胜任指导积极主动的年轻人。我认为，总的来说，刚才谈及的这类编程与那句数学相关的老话很相似，即大部分数学家的杰出工作都是在30岁之前完成的。从事真正前沿的数学工作需要的那种激情和专注，也许和我年轻时疯狂编程所需的差不多。

Seibel：一部分激情就是耗在长时间的工作上。长时间工作是必不可少的吗？抑或这只是我们热爱编程的后果？

Cosell：我认为那完全是个性使然。能否把事情搁置一阵回头再做，抑或是否要求自己坚持做一件事直到完成为止，这种问题更多是由个性决定的。当然有许多人，我在BBN认识的那些出类拔萃的人，在正常工作时间内就能完成所有工作，并没兴趣周末到公司加班。当然，还有其他人则近乎疯狂，有一阵子，我就睡在机房里，因为开车回公寓要花很长时间。我会在机房里打个盹，我不清楚在别人看来自己这么做有多疯狂。但我并不认为非得这么做。我觉得，我们做的那些事情真是激动人心，这只是个附带的结果，尤其是当它开始逐渐成形的时候。

在BBN，有个很厉害的家伙上下班非常准时，他对自己近乎苛刻，最终完成了博士论文。他用周六写论文，晚上则处理杂务。我想，他能完成论文，部分是靠合理的安排。如果你能一直如此地做一件事，那么做起来就会轻松

得多，你不必考虑安排是否仔细，也不用打断一阵再重新捡起来做，因为一俟事情做完，便可抛诸脑后。我最近一直在学习怎么合理安排，现在我的生活更为正常。我会抽空编程，事情搁置一阵，尔后再重新捡起来。我发现自己只要两三个礼拜不做那件事，要想重新捡起来就变得异常困难。通常，当我在某些小的个人编程事务上滞后时，我真的很想搞定它，就会试着安慰自己："好吧，我打算像运动锻炼的人那样，每天早上花几个小时在这上面。"这种做法对我并不奏效。结果往往是到某个时候发现它还没做完，我开始厌烦，就会用上一两天把它搞定。

我隔三差五还能像以往那样集中精力，但还是没法跟壮年时期相比。因此我认为那种特殊、费神的编程，也就是真正的黑客所为，更多是年轻人的行当。我必须承认，我认识的几乎所有年轻时有过杰出成绩的人，那时都是在高强度下工作的。我很难想象真正有所成就的人一天只用几个小时就能搞定工作。我认识的几乎每个人都是集中精力、大强度地工作，到了近乎疯狂的地步，直至搞定为止。但现在要像那样集中精力很难。这会让人筋疲力尽。当然以前这也会让我筋疲力尽。

Seibel：*你认为自己是科学家、工程师、艺术家、工匠，还是别的什么？*

Cosell：显然是上述各种角色的混合体。我不觉得自己是个科学家，我深谙科学家都做些什么。我更愿意把自己看作是艺术家和工匠的混合体。我做工程时采用的方式就结合了艺术和手艺。

Seibel：*那我先来问问工程部分。显然，Watts Humphrey和软件工程研究所（SEI）那些人认为编程应当属于工程学科，就像建造桥梁那样。人们能够建造桥梁，还能估算工期有多长，而且绝大部分桥梁都不会倒塌。*

Cosell：完全正确。这个类比非常恰当，只是这中间的推论不合逻辑。按照这个逻辑，桥梁不会倒塌全靠设计人员，而非捆扎缆索、检查缆索确保钢架正常、浇注混凝土或是做其他工种的人。

从那个角度来看，编程也算是工程学科。你必须知道怎么做。你也必须清楚自己的能力。就我工作的内容来讲，我必须能够预估各个模块是怎么组合在一起的。我得具备一定直觉，知道哪些快，哪些慢，哪些东西很难构建，哪些东西容易构建，还必须从工程层面上提出一个我认为可行的模型。

艺术家部分决定整个设计应当优雅。这些都是相辅相成的，因为对计算机程序而言，艺术性会影响其生命力。我所谓的计算机程序的艺术性部分，是指将来人们能够修改这个程序同时还能保证程序正常运转的难易程度。这

与其功能的构建没什么关系，而是影响到构建完成后的生命力。

Seibel：*那么在你看来，代码之美和人们必须修改代码这一事实紧密相关。*

Cosell：我写的程序只有几个是黑盒性质的，计算机运转期间必须一直运行，但绝大部分代码好几代程序员都能修改且多数仍能保持正常。当我谈及程序的艺术性和美感时，实际上想说的是，编写程序时你必须考虑到方方面面。怎么组织例程，怎样布局例程，哪里该添加注释，怎么命名变量，你是否希望子例程都有统一的调用顺序，或者根据情景不同编写相应的代码。

因此，之后某个时候你必须以新程序员的角度重新审视一遍整个程序。这个程序的结构怎么样？你在做什么？你是怎么做的？你为什么这么做？艺术性是指下一个开发人员阅读了整个程序就能理解这个子例程的功能，意识到不能打乱这个子例程，程序结构应保持不变。

Seibel：*怎么看待清晰和效率之间的关系？有时，最简单易读的代码并不是最快的。*

Cosell：程序员是世界上最差劲的优化者。他们总是优化最吸引眼球的那部分代码，而真正需要优化的代码则几乎从不触及。因此你优化得到的这些异常难懂的小段代码没什么价值，根本不得要领。我总是告诉和自己共事的人："想方设法把代码写得清晰、易读，如水晶般透彻。采用简单直接的做法。然后，如果真的需要提高速度，我们稍后会处理。这么做了之后，再加个小框注释一下。"

很久以前，某个Emacs版本的一页源代码里，有段注释里画了个大大的骷髅头，另外还写道："下面这段代码一团乱麻。"这是搜索代码或类似功能代码的最底层部分，他们已做过彻底的优化。我看过那部分代码，的确十分晦涩难懂。那地方还有一个大黑框，里头写道："不要迷失于此，除非你知道自己在干什么。"

但是，相比我在PDP-1上开发时的优雅风格，我们现在编写的程序越来越庞大、笨拙，速度也更慢。这样也行，问题不大。眼下，从事视频合成以及CGI动画制作的那些家伙，编程显然没法这么随意了。这些工作要求非常精心的编程。这活我可能再也干不了了，我已经落伍了。不过我以前能做。我能理解做这活的家伙。不过我们编写的大多数程序只需循规蹈矩即可。

有些大学会开一门两学期的课程，从9月持续至第二年5月，一开始你得开发难度相当大的程序。他们事先并不提醒你，他们准备在4月份让你重写那个程序，现在真正磨练你的是其他事情。他们的打算会让你惊讶得目瞪口

呆，因为要想起仅仅六个月前你自认为已经精通的东西相当难。

Seibel：这么说来，所有那些到期之前最后一刻出的活早晚还会是个问题。

Cosell：是的。我认为这个课程安排真的非常巧妙。这是在给他们上一堂只有在现实生活中才能体会的课。

Seibel：与Ken Thompson交谈时，我问他C语言是否天生就存在问题，会导致安全问题，他直截了当地回答那不是问题。你教的是计算机安全方面的课程，怎么看待这一点？

Cosell：我不想与他交锋，不过我在自己的计算机安全课上会直言，现代计算机面临的最大安全问题就是C。C一开始被设计为系统编程语言，而且用起来非常舒服，于是所有高手都开始使用这门语言。我们用它来构建操作系统，用它来构建实时系统。

我还记得Pascal时代的两个论战，一是计算机应当帮助你，二是C这门语言太危险了。我记得喊得最凶的两个人是Wirth和Dijkstra。另一方是我认识的所有系统程序员，包括我自己在内。我用C编写所有程序。总之，那个时候C几乎以不可阻挡之势将无数语言甩在后面。

那时政府试图强行推广Ada，他们签订的所有项目合同都只能使用Ada。C还是越过了这些障碍，实在是令人吃惊。不过现在重新审视，我几乎每时每刻都仍惊讶不已，用C编写真正复杂的程序并且不存在安全问题几乎是不可能的。程序员必须投入大量精力，才不会在显式确认操作没有溢出缓冲区之前读取缓冲区，才不致在错误的时间释放一块内存导致程序里其他地方的指针失效，才不致存储错误大小的东西，覆盖掉邻近的值，那些问题查找起来可能异常困难。

系统编程一直受益良多。用汇编器编写系统以及用Pascal编写全部应用程序的想法，令我不寒而栗。我并不认为这是正确的选择。我不得不说，用C编写系统和应用程序，已被证明并不是很有效。想要有效实在太难了。

这和我们碰到的中断bug的问题有点类似。你可能会争辩，编写一连串中止或中断的程序没什么神秘的，也不存在真正的问题，只要多点理解，多加小心就行。不过据我所知，即使是非常出色的程序员，掌握了所有相关知识，照样会在其程序里出现那些bug。像我这样的程序员就必须上阵修正那类bug，我得学学Niklaus Wirth，发明不会产生中断bug的计算机语言。

我为IMP系统写了一组复杂的汇编宏，这样你就可以声明自己在做什么。当你进入某个中断时，就可以写一句声明，“我处于调制解调器输入中”

或“我处在高优先级时钟或低优先级时钟中”。然后，在汇编你的程序时，实际上它会给每个指令附加点东西，表明该指令运行于哪一级中断，此外有个后处理器，好像是用TECO宏写的，真的，它就会处理附加的东西，查找分时问题。它会查找一个变量，发现两级不同的中断都访问该变量，然后就会提示："出现中断冲突。"这下出人意料，分时bug就都不见了。其他程序员可以理解，如果他们添加相应的声明，这些宏就能让他们避免犯时序上的错误。我去匈牙利做了一次演讲，指导那些并不真正理解实时问题的程序员，让他们利用这项技术隔离冲突问题，从而编写可靠的实时程序。

我对C的看法与这有点类似。我相信好的程序员，也许可以包括我自己，能够写出好的C程序。但它比本该的要难。在现代环境中，它变得愈加困难，因为现在的环境更难对付，C的缺点可能被利用或忽视的地方更多，需要花费更多精力。这也是我用Perl写程序非常自在的原因之一。Perl比较慢。我相信Perl是速度较慢的一种语言，但它从根本上修补了用C编程存在的所有安全问题。用Perl编程时访问数组越界时会发生什么？它会扩展这个数组，以容纳更多元素。

Perl知道指针指向什么位置，因此你绝不会引用错指针，因为你只需说从头到尾查一遍，它会告诉你去什么位置。因此，我用Perl构建考究安全的应用程序要自在舒服得多，因为有大量Perl开发人员在锤炼其核心，这么多年来一直保持稳定。我不认为我们会找到太多内存分配或指针bug，何况这些bug无论以什么方式都很难利用随机Perl代码。我也不用依赖身边的程序员检查每个指针并保证正确。

那时，我们看到过不少有问题的程序，比如那个经典程序，有人写了个Web页面，可以在数据库表里查找某人，结果有个黑客输入了“Joe;drop all tables”（删除所有表）之类的内容。那种问题仍然存在。那显然不是C的过错，但确实表明了程序员防不胜防。他们无法查看所有地方。而C犯错的地方可能太多。对我来说这太可怕了，老实说，我猜自己用C编程的时间只比Ken少五年。我们虽不属同一级别，但我使用C编程也有很长一段时间，知道它有多难，我认为C是这个问题的主体。

随着这些应用越来越复杂，构建于越来越复杂的运行库之上，没人会试图去了解这些库的安全漏洞，因为这些库都异常复杂，或许我们将不得不转到更可靠的应用开发语言上。处理器的速度快得令人乍舌，内存已经便宜得不能再便宜了。难以想象明天的语言会是什么样。我并不认为C或其衍生的C++等语言是高负荷程序应用乃至系统开发的正确之选。

Java给我的感觉并不好。我又要老调重弹了。我觉得Java过于专制。这也正是我觉得Perl不错的原因之一，因为它有足够的安全和检查，但它提供如此多维度，让我的艺术家部分有如此广阔的自由空间来清晰地表达事情，考虑做事情的正确方式。我有一定的自由。

当我初次捣鼓Java时，当然那时Java还是新生语言，我的反应是："哦，又一门通过限制程序员所为来帮助他们步入正轨的语言。"不过，也许我们是时候该这么做了。也许世界已变得如此危险，你没法再靠一门优秀灵活的语言，让百分之一二的程序员用它来创造伟大的艺术，因为世界上有七千五百万普通程序员，在构建着这些异常复杂的应用，他们需要更多帮助。从这个角度看，Java也许做对了。我不得而知。

Seibel：*Fran Allen在IBM从事Fortran编译器研究，我同她交谈时，她对C相当失望，不过视角完全不同，认为用C不可能编写真正高度优化的编译器，因为C太底层化了。*

Cosell：现在，她来自不同的阵营。她从事编译器研究，把C看作可怕、拙劣的退步，你对它无可奈何。而我们原先使用的是操作比特位的汇编器，C令我们耳目一新，如沐春风。因此，显然那个时候绝大部分顶尖程序员并不写BASIC程序，也不写Fortran程序执行计算。真正的大人物都是写汇编代码的。于是我们都投奔C，因为C让人如沐春风。如果你认为C的数组检测有问题，不妨在汇编器里自己写个数组循环试试。因此从这方面来看，C语言是一大福音。

我不想说C已经失去其效用，但我认为太多优秀程序员使用C，以致现在不够出色的程序员也开始用它来构建应用程序，归根结底就是他们不够优秀，无法驾驭C。也许对于真正优秀的系统程序员而言，C语言是绝佳选择，但不幸的是，不那么出色的系统和应用程序员也在使用C，他们本不该用C的。

Seibel：*鉴于我们无法再全面掌握它是如何工作的，你是否认为编程的本质已发生改变？*

Cosell：哦，是的。这又显得我很落伍了。一切都建立在之前发生的事上。我记得在旧的PDP-11、第7版Unix上，我们要做些动画和图形。那是一个大难题，很难编程。显示做起来不顺手，也没有现成的库可用。

每一代程序员都在离底层的东西渐行渐远，开发用的工具也越来越花哨。好处是他们做事可以更灵活。这个基线不错，让新的技术争奇斗艳，然

后又形成新的基线，两年后更是异彩纷呈。问题是这些基线越来越复杂。与当前涌现的这些东西相比，PDP-1指令集真是太简单了。

我不想成为微软员工，他们构建的这些操作系统必须能跑在四核多处理器上。显卡也已经带有几十几百兆的内存，上面还有完整的流水线并行处理器，能够实时完成数组和向量运算。总之现在你可以将显卡用作非常棒的数据处理器。我老在想，这些编程做起来得有多难。

我们有台IMLAC机器，那是台真正带有不错的集成向量显示器的早期机器，做法同老的PDP-1一样，不过它是台迷你计算机。那台机器有个程序，你会坐在一辆小车里，身处三维显示的迷宫。你会看到墙壁从旁掠过。你可以仔细查看各个角落。我非常着迷，因为它采用了隐线抑制。那个时候，人们更多是在《ACM通讯》上发表文章讨论算法。我有本书，全书讨论怎么使用对称坐标和某个算法，计算出两条线的交点，这样你就能知道一条线穿过一个平面的具体位置，进而知道一到那个位置就不要再画线，因为现在那条线已经隐掉不见了。

那时处理隐线还是个大难题，IMLAC上的那个程序却做到了。那个程序真叫人吃惊。那代码真是不简单，独一无二。而现在，据我所知，显卡支持三维坐标，本身就能实现隐线抑制。八九年前，诸如纹理映射和光线跟踪等还是难题，很难靠编写代码实现。编程实现球体反光得花上数小时。

现在，显卡就能做光线跟踪。一方面，在NVIDIA工作的那些人必须实现异常复杂的东西；另一方面，现代程序员不再满足于写些只有线描墙壁的程序，他必须掌握这个不可思议的、构建于越来越复杂的运行库之上的3D视频环境。使用这些库比你自己动手写代码实现来得容易，不过我还是没能弄明白现在人们怎么可能掌握这一切。对我而言，这太难了。

我只是在用Tk进行开发时才碰到了这个问题。我一直想写个简单的Tk程序，看到Tk那么复杂，一下子手足无措，Tk居然有那么多钩子函数，要让按钮变大变小，移这移那，都得用到这些钩子函数。要掌握这些可是个大难题。相比之下，理解PDP-1分时系统真是简单。

因此我并不羡慕现代程序员，而且情况会变得更糟。简单的东西都已经包装成库，剩下都是难以处理的。事情正在变得如此复杂，而人们的期望值却高得惊人。他们给我展示的一项功能着实叫人吃惊。有人向我展示Google Maps如何为用户规划路线。你可以用鼠标把一条路线的一头拖拽到某个位置，另一头拽到另一个位置，给Google指定你要去的地方。然后Google会重新绘制那条路线，它会经过你刚才拖拽时指定的位置。现在我明白那是怎么

回事了，底层有大量JavaScript代码来跟踪鼠标。当你移动鼠标时，它会向服务器系统发送一条Ajax XML请求，表明用户在路线上放置了一个点。随后这条路线就会在原有基础上更新。系统会重新计算路线。真是难以想象，他们怎么能把代码写得那么好。人们会抱怨计算得到的路线穿过人家的后院等类似的问题，但是最优路径问题一直是计算机科学的一个经典难题。怎样才能分析这个形状不定的图，并通过它找出最短路径？真是令人惊讶不已。

从一个角度看，我在想“能做到这个真是太酷了”。从另一个角度看，我内心的程序员部分会说：“天哪，真庆幸自己是程序员的时候没这些玩意儿。”我可能绞尽脑汁也写不出能够实现上述功能的代码。这些家伙是怎么做到的？这代程序员肯定比我那时的程序员厉害得多。我真庆幸，自己曾经是一名好的程序员，混了点名气，现在再也不用真正去证明自己，因为我觉得现在自己恐怕已无力证明。

对于已过巅峰期的程序员，现在退休是个好时机，因为至少你还有些优势，曾经干过编程，但这个世界是如此奇妙，你完全可以好好利用这一点，也许还能借此捞点美名，不用非得自己做得到。不过，你要是身在大学，专攻计算机科学，那你就得开始行动，想方设法掌握这堆东西。我终于解脱了。

（编辑：傅志红）

15

Donald Knuth

戴玮　译

本书所有主人公中，Donald Knuth或许最为闻名遐迩。他笔耕逾四十载的多卷本巨著《计算机程序设计艺术》[①]，如今已成为基础算法与数据结构领域的传世经典。《美国科学家》[②]杂志将其评为20世纪最重要的12部自然科学专著之一，与罗素和怀特海、爱因斯坦、狄拉克、费曼、冯·诺依曼等人的著作齐名[③]。他推广了算法分析中的渐近符号（即大*O*符号），发明了

① 这部巨著原计划出版7卷，但就译者看来，Knuth在其个人网站上的言辞似乎让人感觉完成前5卷是更现实的目标。他打算在写完第5卷之后，重新对第1～3卷进行修订，并写一本精华集，包含前5卷中最重要的内容。

② American Scieutist是1913年由科学研究社团Sigma Xi创办的一本科技杂志，双月刊。虽为科普杂志，但面向的读者却是科学家，其刊名也反映了这一点。每一期杂志都由杰出科学家或工程师介绍一些科学界的重要工作，内容权威且艰深，需要一定的专业水准方能看懂。

③ 除了文中提到的这几位科学巨匠外，其他几本书的作者也是鼎鼎大名，包括分形几何之父曼德勃罗（Mandelbrot）、控制论鼻祖维纳（Wiener）、量子化学与结构生物学先驱鲍林（Pauling）等。感兴趣的读者可参考这个网页：http://www.americanscientist.org/bookshelf/pub/100-or-so-books-that-shaped-a-century-of-science。

LR语法分析，还反驳了Dijkstra对goto语句的批判[1]。

Knuth不只是理论家。1976年写完《计算机程序设计艺术》第三卷后，他本打算花一年时间写个排版系统，包括TeX和METAFONT，好让自己的著作看起来更加赏心悦目。结果十年时光过去，他才完成了这款软件。同时，他还发明了一种称为"文学编程"（literate programming）的编程风格，以及一种用于排版、至今仍很先进的文章段落断行算法。

他一生获奖无数，包括首届美国计算机协会Grace Murray Hopper奖（1971年）、图灵奖（1974年）和美国国家科学奖（1979年）。自1990年起，他不再使用电子邮件[2]。据他自己解释，这样做是因为他追求的不是"高屋建瓴"而是"寻根究底"，他必须深刻理解与领悟计算机科学的广袤领域，从而在书中加以解释。

在这篇访谈中，我们谈到了Knuth对文学编程的由衷热爱，对黑箱的矛盾心态，以及对过分注重软件复用的遗憾之情。

Seibel：*你是什么时候开始学习编程的？*

Knuth：是1956年秋天，我在凯斯理工学院[3]读大一时。在那个学季[4]还是学

---

① 关于goto语句是否有害的论战曾在编程史上风行一时。Dijkstra批评goto语句破坏了程序结构，而Knuth较为客观地从结构与效率两方面分析goto语句，主张在必须提高效率的情况下，可以有限制地使用goto语句。此类论战可谓见仁见智，看问题角度不同，见解也就不同。但以译者拙见，现今的各种高级语言往往通过包装goto语句来使其改头换面，以达到提高效率的目的。如我们熟悉的异常处理、break、switch语句等，都相当于Knuth所说的有条件限制的goto语句。这似乎也在一定程度上证明了Knuth的观点更贴近于实际。

② Knuth专门为此写了一篇文章，解释他看待电子邮件的态度。他暗示，电子邮件是一种效率低下的联系方式，那些事事操心的人才会需要它，而他现在除了专心写书之外，什么都不用操心了。他也提到，自己从1975年开始一直到1990年，总共用了15年电子邮件，已经够意思了。有趣的是，他还建议大家去掉e-mail拼写中的短横线，改成email，同样，non-zero、soft-ware等单词也应如此，理由是多打这么个字符浪费了人生中的很多宝贵时间。感兴趣的读者可参考这个网页：http://www-cs-faculty.stanford.edu/~uno/email.html。

③ Case Institute of Technology，坐落于美国俄亥俄州的克利夫兰市，由慈善家伦纳德·凯斯（Leonard Case Jr）创立于1880年，1967年与西方储备大学（Western Reserve University）合并为凯斯西储大学（Case Western Reserve University）。Knuth在这里度过了大学的四年时光（1956年—1960年），并且由于他学业表现异常出色，毕业时被同时授予学士和硕士两个学位。

④ 美国大学通行三种学制：学期制、学季制、三学期制。学季制每一学年分为四个学季，每学季大约10～12周。

期里，学校弄到了一台计算机。

Seibel：IBM 650？

Knuth：没错，就是它。它是首台产量过百的计算机，前后大约生产了数千台，但可能没到一万。不过即使没到一万，它也是第一台大规模生产的计算机。所以，就连凯斯都能弄到一台。

那阵子我在统计实验室打工，负责整理卡片上的数据。我为统计员绘制数据表，以补贴奖学金的不足。机房就在这实验室的一楼，有个窗户。你能透过窗户看到那台计算机，还能看到它的指示灯在忽明忽暗地闪烁着，简直迷死人了。

一天下午，实验室一哥们走到黑板前，给我们三个大一新生讲解那台计算机是干什么用的。我还找到一本手册，上面有一些小程序示例，不过在我看来，它们有些糟糕——程序根本没多长，却仍有改进余地。

后来，学校居然批准我们可以在晚间操作那台机器。这种事并不常见，当时敢让本科生这么做的学校，我想大概只有达特茅斯[①]和凯斯。如果在其他学校，你得先把几组卡片交给专门的计算机管理人员，第二天再去拿结果。可在凯斯不一样，什么都得你自己动手，他们只会对你说，“嘿，这么做可要当心”，“你不应该那么做”，“这会把机子弄得一团糟”。就这样，我们真得到了不少摆弄它的好机会。

话说回来，我对手册上的程序做了些改进，我得看看它们是否有效。结果确实有效。于是我感叹道：“老天，这也太夸张了，我才上大一，就能比这手册做得更棒，没准我是天才。”好吧，我是天才，这没错，不过和我当时所想的天才却不太一样，因为是个人都能写得比那手册里的程序更好。

我在中学阶段简单接触过一些二进制计算，不过这台计算机采用的是十进制，因此我不必深入研究诡异的二进制计算。不知怎么的，它由于采用十进制而显得更有人情味儿，或者说让人感觉更舒服。我现在仍然记得它的机器码，其中65代表“先复位再加到低位”[②]，这可以帮我编密码什么的。

---

① 达特茅斯学院（Dartmouth College）建于1769年，是美国历史第九悠久的大学。它位于美国东北部新罕布什尔州汉诺瓦市，是一所私立大学，也是常春藤盟校之一。其在计算机发展早期曾经地位显赫，首个成功实现的大规模分时系统——达特茅斯分时系统（DTSS）以及著名的Basic语言均出自该校。不过如今的达特茅斯以商学院享誉全球，计算机科学却日渐没落。

② reset-add-lower是IBM 650的一条指令，机器码为65。该指令的作用是先把累加器清零，然后把指定内存里的内容加到累加器的低10位上（IBM 650的累加器共有20位数字位和1位符号位），其作用似乎与现在汇编语言里的MOV指令相当。

Seibel：坏了，你把自己的秘密说出来了。

Knuth：嗨，管它呢。后来，我决定写个100行左右的小程序计算质因数。我在夜深人静、机器闲置的时候调试程序，结果就这一百来行，我找出了上百个bug。两星期后，我排掉了所有bug，写完了这个程序。它能算出你用控制开关输入的任意一个十位数的质因数。

我就是这么学编程的。说白了，就是先鼓捣出一个程序，然后坐在机器前，花上几周时间来一点一滴地改进它。

我写的第二个程序可执行二进制和十进制之间的数字转换。不过我在写了第三个程序之后，才真正成为一名程序员——那是个会下“井字棋”①的程序。

我必须给这程序用一些数据结构。我前后一共做了三个版本，其中一个版本具有自学习能力。一开始，它对下“井字棋”一无所知，然后自学习能力让它明白，自己输棋时走的棋挺臭，而对手走得还行，于是它会给一些格子加分，给另一些格子减分。这么着，在下过400盘后，它就能下得有模有样了。

Seibel：*我访谈过的人里，似乎有很多都是由直接操作机器开始学习计算机的。不过Dijkstra有篇论文你一定很熟悉，他在那篇论文里主张不该让计算机系学生在学习阶段头几年直接接触到机器，而是应该让他们在符号操作上多下工夫。*

Knuth：他当初可没那么学。他提出过许多真正伟大且耐人寻味的观点，然而人非圣贤，孰能无过。当然，我也不是圣贤，但在这件事上，我的看法与他不同。打个比方说吧，某领域有位科学家，随着年纪渐长，他的学识和阅历也日渐精深。然后有一天他宣称：“哦，对了，我学过的知识中，有些的确回报很大，但也有些从来就没用过。我不会再让我的学生为这些没用的知识浪费时间，也不会再讲授任何底层的基础知识。了解那些强大的理论就够了，忘掉我是怎么得到它们的吧。”

我认为，各个领域的科学家都会犯这种根本性错误，他们不明白，学一门知识必须全方位、多层次了解，浮沙之上筑不起高台。然而，我们的大脑在完全理解了基础知识后，会把它们都塞进一个不起眼的角落里，所以，那些老学究还真以为自己用不着它们。

---

① 井字棋（tic-tac-toe）的规则是在九宫格内由双方轮流落子，先连成直线或斜线者胜。由于其局面较为简单，因此广泛应用于人工智能等领域的研究中。

Seibel：为了写这本书，我访谈过不少人。我问过他们同样一个问题："《计算机程序设计艺术》这套书你读过多少？"其中大多数人回答只是把它们作为参考书查阅，仅有少数人回答从头到尾通读过。是不是每个程序员都应该看懂它们呢？里面的数学内容实在是太高深了。

Knuth：我偶尔也会纳闷，就连我自己是不是都看不懂它们了。我在讨论书中的某个主题时，会尝试围绕这个主题组织大量知识。这些知识的来源各不相同，所以看上去有些支离破碎。我把它们汇总到一起，捏合成一个可由他人继续补充发展的统一实体。我还澄清了历史的谬误，修正了原始材料中存在的缺陷与晦涩之处。

一直以来，我都从数学期刊上获取素材，由这些素材开始动笔。我无法想象数学期刊上的术语在编程圈子里也同样让人耳熟能详，因此我得尽量把它们变得通俗易懂，至少要达到我自己不费什么劲就能理解的程度。我试着提炼它们的核心思想，尽力简化它们。不过如果我真的做到了，那我在书里随便挑5页就够别人学一辈子的了。

也就是说，计算机领域的知识实在是太丰富了，值得你毕生钻研的知识无论如何也还是比我书中任何5页多得多，我不可能把它们完全归纳为一组简单知识的集合。假如计算机科学其实简单无比，那你只需列出50个知识点，然后去潜心研究它们就行。那样的话，我会大声宣布："没错，世界上每一个人都应该了解、而且要彻底了解这50个知识点。"

这终究不过是白日做梦。现实中，我把自己积攒的几千页资料和习题都写进了书里，这样它们就不会占据我有限的大脑空间。不过我把它们写进书里时，还得再重温一遍。此外，我还在书里提供了习题答案，因为再过十年，我肯定会忘记这些麻烦问题的解法，到那时，我又得花大把时间重新解题，所以我最好还是给自己预留一些基本线索。

对我的书，人们时常持有两种不同评价，这让我感到左右为难。一种评价是："哎，这部分内容太复杂了，你还是别提它比较好。"另一种评价是："你这书上的知识都太微不足道了，没什么意思。"对前一种评价我愿意回答："我干脆什么都别提算了。"对后一种评价我想说："鄙人实在是太才疏学浅了。"

说到底，对我来说至关重要的是，所有能在半页纸篇幅之内解释清楚的精妙知识，我这书里必须得有，所有精彩得让我过目难忘的材料，我也决不会错过。于是我发现，在我最近写的二叉决策树一章中，居然塞进了260多道习题。因为层出不穷的素材似乎在提醒着我，关注这章的读者绝非等闲之辈。不过，我的意思可不是所有读者都会对这260多道习题感兴趣。我只是

相信，很多读者会因为这里的每一道习题，而对我的书赞不绝口。

居然真有人从头到尾读过我的书，这事儿太让我惊讶了。多数情况下，人们只挑选自己喜欢的章节看看。可他们如果继续读下去，会发现书中的内容其实只用到了一小部分术语，而并非充斥着各种各样的符号和术语。如果我不写这些书的话，别人查找类似资料的过程可能会困难得多。这正是我不断写作的动力。

另外要提到的一点是，我宁愿以一种最接近编程实践的方式来探索计算机科学领域，也不愿借着发表一些虽具理论意义、却对实际编程毫无用处的东西，去博取学术上的功名。

我有意没有放入书中的，是诸如那种仅当n大于2的一百万次方时才可以节省$\log(\log n)$因子复杂度的数据结构。现在仍有不计其数的论文在研究它们。这些论文其实在故弄玄虚，因为从理论上说，假如计算机跟上帝一样，有能力处理任意数量的数据，那我们的确能获得越来越快的算法。然而，事实并非如此。如果我自己写的程序要用到树，我甚至都不会考虑平衡树或AVL树，除非我事先估计这棵树的规模会变得很大。

Seibel：那你用什么呢？

Knuth：我就用普通的二分查找树。但往树里放东西的时候，我会使个小技巧把它随机化[1]。

Seibel：说到实践，你在写作《计算机程序设计艺术》期间，中断了10年时间之久，转而编写排版系统TeX。我听说你在写TeX的第一版时，连计算机都没用着。

Knuth：1977和1978年，刚开始写TeX那会儿，我当然还没发明文学编程，用的还是结构化编程。我用铅笔工工整整地把程序记在了一个大本子上[2]。

六个月后，我写完了TeX的第一版，于是我着手把它输入计算机。我从1977年10月开始写这程序，到1978年3月，我就开始调试程序了。如今，这些代码存放于斯坦福大学[3]档案馆，都是用铅笔写的。当然，如果我发现其

---

① 随机化是算法中经常用到的思想，其思想是为了避免最坏情况，对数据结构中的元素进行随机排列或选择，以此达到平均情况下的性能。

② 不仅写程序，Knuth在写书时也会用铅笔写初稿，然后再输入到计算机中。

③ 斯坦福大学（Stanford University）由铁路大亨、前加州州长利兰·斯坦福（LeLand Stanford）和他的妻子简·斯坦福（Jane Stanford）于1884年为纪念他们夭折的独子而建立。硅谷的繁荣与斯坦福大学密不可分，很多著名公司比如谷歌、雅虎、思科、惠普、NVIDIA的创始人都是斯坦福校友，1951年创建斯坦福研究院区并鼓励学生和教师创业的副校长弗雷德里克·特曼也被誉为“硅谷之父”。从这一意义上说，斯坦福堪称IT产业的发源地。

中有什么问题，我会跑回去修改那些存在问题的子程序。

那时候，计算机还属于第一代系统，充斥着各种五花八门、千奇百怪的架构。无奈之处在于，你只有使用某种架构一段时间后，才明白它并不是你想要的，才不得不把它放弃。写这软件还存在一个“鸡生蛋、蛋生鸡”的问题——要先有字体才能排版，但也要先排出版，才知道到底需要什么样的字体。

不过结构化编程让我认识了不变式[1]，还让我明白了怎么写一个我能理解的黑箱。因为有了这些工具，所以到了我最后调试代码的时候，我对它能正常运行有十足把握。先花上六个月时间把代码全部写完，然后再去测试，这样做可为我节省大量时间。我充分相信这些代码不会出什么大问题。

Seibel：*也就是说，你没花时间对那些还不存在的代码构建测试用的脚手架和桩[2]，因而节省出大量时间？*

Knuth：是的。

Seibel：*要是没把那十年时间花在TeX上，除了版面会更加美观之外，你的书会和现在大不一样吗？*

Knuth：问得好。以一种非纯学术的方式使用结构化编程的经历——也就是说，我不仅思考玩具程序中的不变式，还思考现实程序中的不变式——很可能充分影响了我现在在新书中描述算法的角度。或者说，哪怕现在还没影响，以后也肯定会影响。

如果我一直以学术专著的路子写这些书，我本来无需了解高速缓存或是计算机发展趋势之类的。我写《计算机程序设计艺术》前三卷的时候，编的都是些玩具程序，没编过像TeX这样具有很多大型编程实践特征的程序，这也让我把更多精力放在了数学问题上。

令我大为惊讶的是，当一个人写书的时候，有些什么东西在潜移默化地影响着、改变着他的表达方式，我甚至不明白它们是怎么做到这一点的。TeX

---

① 不变式是指这样的断言：执行一系列操作前，该断言为真，而在执行它们后，该断言仍然为真。虽然程序由各种变量组成，但对于程序设计来说，认清哪些条件保持不变也十分重要。这有助于我们理解程序中的逻辑。不变式，就是规定不变条件、对其验证以保持程序逻辑正确的断言，广泛应用于编译优化、契约式设计、形式化验证程序正确性等领域中。

② 在软件测试中，脚手架（scaffolding）是指一套用来支持测试的代码。这和建筑行业里的脚手架类似。它们构建出来只是为了方便下一步的构建。其代码仅在测试期间运行，不会带入到最终产品中。测试中的脚手架包括三大组成部分，其一是驱动，其二是桩（stub），其三是环境模拟。驱动调用真正的测试，桩模拟函数、方法和对象，环境模拟对程序实际所处环境进行了模拟。

给我带来的最重要的影响就是这样——它改变了我从精神层面看待事物的角度，也由此改变了我的写作风格。我会更自由洒脱，会更全面地看问题，也会对写出来的东西更有把握。

Seibel：你觉得通过编写TeX，自己的编程水平突飞猛进了吗？

Knuth：嗯，是的。这全拜文学编程所赐。

Seibel：虽然你掌握了更好的编程方法，但在编程技巧方面，你是否有所提高呢？

Knuth：通过实际编程，我学到了很多很多。我明白了编程到底会占用我多大一部分精力。事实证明，编程比我想象的要困难得多。我可以兼顾教课和写书两项全职工作，但却无法兼顾教课和编程。编程需要太多精力去关注细节，那会占据我大脑的绝大部分空间，让我无暇顾及其他事情。所以我现在特崇拜那些从事大型软件项目的人——若非亲身实践过，我决不会这么想。

Seibel：也就是说，编程比写书更困难。我记得在哪儿看到过你说的一句话：不可能预测写一本书要花多长时间。这是否也就意味着，更不可能预测编一个程序要花多长时间呢？

Knuth：对，没错。你做了一个非常漂亮的推论。

今年我差不多写了3个大程序和大约150个小程序。3个大程序总共有100页代码，是文学代码，每页大小为8.5×11，不过其中2个是相互关联的，所以说是两个半大程序也可以。总的来说，我今年应该比去年写得多。今年我不仅小程序写得多，而且甚至有两个小程序花了我一个月时间。

Seibel：你事先想到过它们会花你一个月时间吗？

Knuth：嗯，我想到了其中一个。我知道编写它们不会有多简单，但我不知道它们到底要具备多少功能。我不断地把我必须要用到的功能加进去，结果功能变得越来越多。我觉得这句话是条不变的真理：管理者不该认为编程时间是可预测的。

Seibel：除了《计算机程序设计艺术》和TeX之外，你还是文学编程的发明者和倡导者。通过文学编程，我们能写出别人更易读懂的代码。WEB和CWEB也是你写的，它们分别基于Pascal和C语言实现了文学编程语言。

Knuth：你刚才说到倡导者，我的确挺擅长夸这夸那的。但我也觉得布道——也就是试图改变别人的信仰——是件挺别扭的事。我觉得编程和宗教

很像，每个人心中都有自己的信仰。有些人喜欢强加信仰于他人。有些人不会强加，而是可能会说：虽然我无法证明这东西是最棒的，但我觉得它确实好用，所以我想让别人也尝试一下，并且在试过之后也觉得它不错。不过我不喜欢那种在外面四处转悠、告诉别人应该信仰什么的方式。

Seibel：好吧，也许你能为我们解释一下你为什么如此热衷于文学编程？还有，文学编程和非文学编程的区别到底在哪儿？

Knuth：写作的第一要旨是去了解你的读者，自然，对读者了解越深，写出作品来也就越出色。其次，对于科技写作来说，为了加深读者对你作品所表达思想的印象，达到一种前后呼应的效果，你最好能以某种互补的形式，把所有内容说上两遍。

因此在科技写作中，翻来覆去地解释是种很常见的情况，同一件事会用形式化和非形式化两种方式表达出来。如果不这样做，比方说，如果你先给出一个定义，然后再基于此定义说："所以，什么什么是正确的。"那么，别人只有先理解你给的定义，才能弄明白后面说的到底是什么。

或者，你还可以换一种方式说："我们定义a等于什么什么，将其作为所有主元的集合。"这样一来，"所有主元的集合"这一非形式化术语，就能与我们构造集合a的数学描述前后呼应，从而起到相辅相成的作用。

因此，文学编程基于这样的想法：最合理的表达方法，是以相互关联的形式化与非形式化两种方式来描述同一事物。实际上，文学编程只不过提供了一种能在自然语言和形式化语言之间自由切换，并能把它们结合在一起的天然框架，比方说，在英语和C或Lisp之间。所以对我来说，用这种方式写文档显然更有优势。

另一方面，在写程序时，我可以用读者最容易理解，而不是编译器最想要的形式表达出来。

有些人自底向上编写程序，他们不断构造子程序，让程序规模越来越大，而他们的自信也随之逐渐建立起来。还有些人自顶向下[1]编写程序，他们一

---

① 这里的自底向上和自顶向下指的是两种程序设计方法。自底向上的设计是从具体到抽象，从问题域中已确定的具体操作开始，逐渐向上构造更高层次、更抽象的操作；自顶向下的设计是从抽象到具体，从最高层次的、整体性的问题描述开始，逐渐向下构造出各种较低层次的具体操作。前者对分析和设计的要求较低，可迅速从局部开始编码，但最后可能会得到一些与整体需求不符的模块；后者在分析和设计阶段对问题进行了良好的定义与验证，具有较好的通用性，但最后可能无法完全匹配底层操作。实际设计中，往往把这两种方法结合在一起使用。从更为广义的层面上说，它们代表的是两种不同的思维方式和方法论，可以应用于程序设计之外的各个领域中。

开始就在考虑："嗯，这是我要去解决的问题。想解决它的话，我必须先完成这个，再完成那个。"

在编写文学程序时，我可以根据自己的喜好，两者任选其一。我最终写出来的程序几乎总是和我实际思考问题的顺序相一致。我可以在开始的时候先这么想："这是我要解决的问题，所以我得先解决它，再解决它。"

随后，我还可以换一种思路："现在，让我们自底向上地构建一些工具。"我们心中时刻牢记着最终要达到的目标，但我们可以时而自底向上地构建几个工具，时而自顶向下地解决几个问题。实际上，我们编写文学程序的顺序是：先去解决当前必须解决的问题，再在后续章节里解决下一步要解决的问题。

我开始处理当前最让我头疼但已经下决心要解决的事情。如果我下定决心去做一件事，就会干净利落地把它完成，而不是拖延下去直到火烧眉毛。然而，这种顺序既不同于自顶向下，也不同于自底向上，它是一种心理上的顺序："接下来完成这件事最让我感到心满意足，可我是否已经做好准备、下定决心去完成它了呢？"对于这种顺序，我们有着比较清楚的了解与认识，因此，能够自由地按照人类易于理解的顺序组织程序，这点对我而言十分重要。

既然这方法这么棒，那为什么到现在还没风靡全球呢？为什么不是每个人都用它来编程？有人曾一针见血地指出这一问题的关键所在，我记不清是谁了，好像是Jon Bentley[①]，大意是这样的：世上只有百分之二的人天生就是程序员，也只有百分之二的人天生就是作家，而Knuth呢，他希望所有人又是天才程序员，又是天才作家。

我觉得程序员的数量肯定超不过百分之二——我说的程序员是指那些真正与计算机产生共鸣的人，他们为编程而生，程序就是他们的氧气。但是如今人们都热衷于写博客，我注意到普通民众表达自我的能力已经有了很大提高。因此，前面那项论据的第二部分已不再那么有说服力。

对这样的说法，我只是在斯坦福大学进行过一次小规模的验证。和我合

---

① Jon Bentley是在编程技术和算法设计等领域颇有成就的计算机科学家。他写的《编程珠玑》一书堪称经典之作，该书书名和文章均取自于他在《ACM通讯》上的同名专栏。Knuth曾于1986年5月在该专栏和Bentley联名发表过一篇介绍文学编程的文章。值得一提的是，Bentley在卡内基梅隆大学当助理教授时教过的学生之中，后来也有很多成为知名科学家，"Java之父"James Gosling、Java Collections框架的实现者以及多本Java畅销书作者Joshua Bloch（本书第5篇主人公，在本篇访谈后面还会提到他）、Tcl语言的发明者John Ousterhout、《算法导论》的合著者之一Charles Leiserson均堪称其中翘楚。

作的是一群本科生，他们要完成暑期项目的程序，于是，我向他们介绍了文学编程的思想。只有七位同学整个夏天一直与我合作，其中六人非常喜爱文学编程，甚至直到现在仍在用它，另外一人讨厌文学编程，他觉得文学编程就是先写个普通的程序，然后把它封装起来[1]，最后号称“这是个模块”。当然，斯坦福的学生写作水平都还不错，所以这次试验并不算是随机取样。

Seibel：*我很难想象，人的意识总能最合理地组织程序。为了更清楚地表达思想，你有没有重新组织过自己以前写的文学程序，使其顺序发生重大改变？*

Knuth：没有发生过这种事。我不会回顾自己写过的文学程序，然后改动其中的章节顺序。下一步该做些什么？选项似乎总是唯一的。我无法把这种现象解释得很透彻，但这就像是顺理成章一样，一件事很自然地接着另一件事。

Seibel：*你会不会写些这样的代码：只为自己今后重温，而不是给别人看？*

Knuth：当然会。这正是文学编程的伟大之处——具有自我沟通的能力。我在读自己一年前写的程序时，仍能准确领会当时的意图。

Seibel：*每回都管用吗？*

Knuth：嗯，过了一年再去理解，往往要比当时就理解困难得多。不过我们可以把文学编程和现有的非文学编程做个比较。文学编程虽不能化繁为简，但绝对比我已知的其他方法都更优秀。

我刚刚打印出一大堆子程序中的一小部分看了看，它们都是用C语言写的，并且几乎代表了布尔决策图[2]（BDD）操作的最高水平。这些程序的写法和CWEB背道而驰，但在开发软件的时候，几乎人人都这么写。程序中的注释颇有章法，编程圈子里的人都能看懂。代码本身也不难理解，因为它已从逻辑角度分解为一个个子程序，而且在头文件里，你还能看到数据结构的定义，以及解释数据结构各部分作用的注释。因此，这种风格虽与文学编程

① 文学编程中，一般采用“宏”来封装程序。

② 布尔决策图（Boolean Decision Diagram），也称作二元决策图（binary decision diagram）或分支式程序（branching program），是一种用来表示布尔函数的数据结构。它以有根有向无环图的形式表现出来，由若干代表布尔变量的判定结点和两个代表二元最终结果0和1的子结点构成，其中每一个判定结点都有两个子结点，代表布尔变量两种不同的选择0和1。使用布尔决策图，我们可以轻松完成各种布尔函数的操作，并且能够压缩布尔函数的空间。在归约和有序的条件下，变量顺序固定的布尔函数具有唯一的布尔决策图，因此布尔决策图也可看作布尔函数的规范形式。实际应用中，布尔决策图常用于电子电路的逻辑合成以及计算机软、硬件的形式验证等方面。

截然不同，但却行之有效。

然而，我内心的感觉还是不断提醒着我，文学编程能做的要比这多得多。只不过像编程这种事情，很多都难以言表。对我而言最具说服力的是，在我写过的程序中，有些不用文学编程根本写不出来。比方说，假如我不得不用传统风格写MMIX[①]模拟器，那将会是一个极大的智力挑战，一个我承认自己无法完成的挑战。仅靠分解为子程序的方法，并不足以将其简化到人力可控的地步。

这模拟器要模拟一台包含所有通用技术规范（general specification）的计算机，需要考虑的问题非常多：有什么样的执行单元？每次能执行多少指令？总线如何工作？输出如何工作？分支预测如何设计？流水线如何运转？

你能想象一台具有6个除法单元和多级流水线的计算机吗？它会帮你模拟出来。你想不想用这台计算机来更迅速地算出质数？它甚至用不着你动手制造。

我可没说过划分子程序做不出这样的程序，但我绝对不会那么做。用文学编程来写的话，整个模拟器程序只有170页长，更重要的是，别人也能看懂它——这程序不是只给我一个人看的。

Seibel：*我读了你用文学编程重新实现的游戏《冒险》[②]。我注意到，它看上去有几分代码与文字水乳交融的感觉。代码很有文学味道，你还可以往里添加字句，摆脱了传统的划分子程序的方式。*

Knuth：没错。它更像是一个个内联子程序连接在一起，而不是去调用子程

① MMIX是由Knuth设计的一种64位精简指令集（RISC）处理器，也是为了取代过时的MIX而重写的机器实现。按Knuth本人的话说，MMIX的机器语言简单、优雅、易学。对于《计算机程序设计艺术》这部书来说，MMIX在三个方面扮演着重要角色：其一，它是一种先进、高效且易于模拟的计算机体系结构；其二，它作为数学模型，保证了算法描述的清晰与严谨，有利于算法分析；其三，用它的机器语言和汇编语言写出来的程序，很容易为读者所理解。想进一步了解MMIX的读者可参考网页：http://www-cs-faculty.stanford.edu/~knuth/mmix.html和http://www-cs-faculty.stanford.edu/~knuth/mmix-news.html。

② 《冒险》（Adventure），全名《深洞大冒险》（Colossal Cave Adventure），是电脑冒险类游戏的鼻祖。它是1976年由Will Crowther用Fortran语言编写的、运行于PDP-10计算机上的纯文字游戏，其界面和操作方式类似于90年代末风靡中国互联网的MUD游戏（侠客行之类的）。最初版本只有大约700行代码、79个可移动的地点，后由Don Woods扩充发展为约3000行代码、1800行数据、140个可移动地点的游戏程序，而且这个规模更大的程序，所需内存反而比原来的程序要小。从Woods的这个版本开始，《冒险》游戏逐渐开始流行，后来也被移植到不同的操作系统，比如Unix和DOS上。

序。这里我提到了子程序的概念，但在最终的文学程序里，它们并非以子程序的形式表现出来，从某些方面看，它们更像是宏。其实叫什么并不重要，关键在于从概念层面上说，如果你拿手的语言还有其他实现方式，那也不一定非要有调用子程序的机制不可。

我觉得自己其实并没有去掉Don Woods写的Fortran版《冒险》游戏中的子程序，我只不过把他的Fortran程序拿过来，再分别放进了英语和C语言中。不过当你阅读我写的TeX代码时，你在子程序栈上进人的调用深度大概只有4或5层，但如果别人来写TeX，不用文学编程的话，那估计要50或100层才行。

我尝试用与我大脑思维相一致的方式，而不是逻辑学家幻想的形式化系统中的方式来构建程序。我的程序理应服从我自己的直觉，而不是服从他人死板的框架。

当然，程序最终还是要进入计算机，而计算机是死板的，有它自己的一套理解事物的准则。但在我看来，正确的程序必须尽可能接近我的思维方式，而不是尽可能接近机器的思维方式。为了做到这点，我不得不寻找一种方法，把我的思维方式转换为机器的思维方式，不过不管怎样，我源程序里的一字一句还是要尽量贴近我的心，而不是机器的心。

我还相信，文学编程是一种对写文档十分有利的风格，而文档可用来在不同群体间交流思想。我知道许多深刻理解TeX代码的编程爱好者，他们甚至能构建出让TeX崩溃的各种场景。我觉得理解TeX的人会比理解其他任何同等规模程序的人都多。

Seibel：*有没有发生过这种事，就算文学代码那么好理解，但还是有人在读过你写的代码之后向你发问，而他们的问题会让你觉得："啊，他们是真的没看见这部分内容吗？"*

Knuth：当然有。这种事总会发生，不过错误是出在我的解释方法上。我给你举个简单的例子。在《计算机程序设计艺术》一书中，我曾谈到面向位操作的早期历史。书中有这样的句子："和EDSAC大致同时制造的Manchester Mark I[①]计算机不仅有按位与操作，还有按位或和按位异或操作。1950年，

① EDSAC和Manchester Mark 1都是英国早期的计算机。前者由剑桥大学的莫里斯·威尔克斯（Maurice Wilkes）教授和他的团队受冯·诺依曼关于EDVAC计算机的报告启发，于1946年开始设计制造，1949年5月6日正式运行，这是一台电子计算机，被誉为世界上第一台实际应用的存储程序式（即冯·诺依曼架构）电子计算机，威尔克斯教授也因此项成就而获颁1967年图灵奖。后者由曼彻斯特大学制造，从稍早研发的实验机型"小规模实验机"（Small-Scale Experimental Machine）发展而来，1949年4月首次运行。图灵曾写过一本关于它的编程手册，还与别人合作，设计了它的输入输出系统。

阿兰·图灵[1]写它的第一本编程手册时评论道：‘按位非可通过按位异或和位串1共同计算获得[2]。’”

这里我说的“阿兰·图灵写它的第一本编程手册”，是指给Manchester Mark I计算机写的第一本编程手册。但是有四五位读者分别指出，我的本意肯定是“他的”，于是这句子成了：“1950年，阿兰·图灵写他的第一本编程手册时。”

唉，实际上，他之前还写过别的编程手册，所以我说的没错，但别人却会错了意。因此，我后来把这句子改成：“1950年，阿兰·图灵给Mark I写第一本编程手册时……”

也会有读者理解错一些数学内容，那样的话我会想，嗯，我的讲解是没错的，但我也知道，我还得不断改进写法，把它们写得更清楚些。

Seibel：*当你发布一个文学程序时，一般来说，它已经是程序的最终形式了。人们常常会提到你说的那句格言：“过早优化是万恶之源。”然而程序在到达最终形式时已经不算早了，你也许已经十分巧妙地优化过了某些部分，但这不会使程序难以阅读吗？*

Knuth：不会的。良好的文学程序会展现出自己的历史发展过程，良好的文学程序也会提醒你：“既然这里有一条光明大道，那我们何不沿着它往下走呢？”

当你把一些精妙而晦涩的技巧用在程序里时，文学编程就开始发挥威力了，因为你不仅将得到代码，还将得到文档。你得说：“我在这儿耍了个花招，它管用是因为——”然后小心翼翼地阐述使用它的原因和设想。

我会在两种情况下耍些小花招：其一，它们的确可提高性能，而我的程序将应用在性能要求较高的场合中；其二，我偶尔会说：“今天我情不自禁

---

① 阿兰·图灵（Alan Turing），英国数学家，最著名的计算机科学家之一，也是人工智能的开创者之一。他1936年发表的论文《论可计算数及其在判定问题中的应用》（On Computable Numbers With an Application to the Entscheidungs Problem）提出了现在被人们称作“图灵机”的抽象模型，为计算机理论和模型奠定了基础。二战期间，图灵设计了一种能够破译德国密码机Enigma的机器，为国家立下大功，战后被光荣授予“不列颠帝国勋章”。1950年10月，图灵发表了他的另一篇经典论文《计算机器和智能》（Computing Machinery and Intelligence），文中阐明了他认为计算机可以具有智能的思想，并提出被后人称为“图灵测试”的机器智能测试方法。1954年6月8日，他由于受到同性恋问题的精神困扰，吃下在自己提取的氰化物中浸泡过的苹果，于家中自杀。美国计算机协会在他去世12年后的1966年，设立了以他的名字命名的奖项——图灵奖，现今已成为计算机领域的最高荣誉。

② 任何二进制数串对全1数串进行异或运算，相当于对其本身求非。

耍了个小花招，因为它实在太吸引我了。”纯属自娱自乐。不过任何情况下我都会写文档，而不是写完程序搁在那儿就算了。

Seibel：文档会在程序的文字部分更多地出现吗？

Knuth：它就在文字部分里。我没让你看到我去掉的那些代码，不过如果想看的话，你尽可以去看。

Seibel：CWEB是否可以包含那些没有在实际应用中用到的代码？这样一来，你就可以说：“这里有一个该函数的最简版本。”而不是只在文档中用文本的形式把这个版本的代码写出来。

Knuth：如果你有这么一段代码，但你从不使用，那么文档里会写着这段代码从未使用。

Seibel：那它只是一个你永远不会引用并且和其他代码没什么关系的代码片断？

Knuth：是的。不过既然代码在那儿，我就可以通过调试器来调用它。我可以下令：“使用某某参数调用某某函数。”程序本身实际上从不会调用子程序，子程序存在于文档之中，因此，我可以先中断程序，再调用这个子程序，看看它是怎么运行的，或者它已经完成了些什么。

Seibel：所以说，用同样的符号你可以这么写：“第一节是该算法的简单实现；第二节是第一节中实现的加强版；第三节是我们实际用到的实现，但如果没读过前两节，你就理解不了本节内容。”

Knuth：完全正确。我写过几个解15数字推盘游戏[1]的程序，并且把它们都放在了我的网站上。程序有3个不同的版本。我声明说：“先读版本1，否则你理解不了版本2；先读版本2，否则你理解不了版本3。”

我写了各种各样的不同类型的程序。有时我对效率毫不在意——我只想得到答案，这时我会使用穷举搜索算法。我在用它的时候什么都不用想——因为这算法一点技巧都用不上，所以我也没必要绞尽脑汁。在这种情况下，我不会做任何“过早的优化”。

---

① 15数字推盘游戏是在4×4方格中，有15个数字和1个空格，玩家要不断移动这些数字，把初始状态下的无序数字排列变为有序排列。此类游戏中较为常见的还有8数字推盘游戏，中国的传统智力游戏“华容道”也与此类似。马丁·加德纳（Martin Gardner）曾在《科学美国人》杂志上征求8数字推盘游戏的最快解答方法。人工智能中的A*算法也常用数字推盘游戏作为例子。

然后，我可以用些别的算法，看看得到的结果是否和穷举算法一致，我还可以增大程序规模、加大运算量。对大多数程序来说，我也就做到这种程度了，因为你不会执行它上万亿次。就如同我为《计算机程序设计艺术》绘制插图时，我可能会对插图进行数次改动，翻译我书的人也许不得不为此而重新编写绘图程序，但我自己绘制插图的方法很慢也无所谓，因为我只需生成一次图像文件，然后把它交给出版商，再印刷到书上。

不过眼下，我正在写有关组合算法的章节，显然，这类问题的规模极其庞大。因此，为了让我书中的例子不至于太枯燥，我必须把解题程序写出来，而且这些程序能让读者感觉到："啊，我明白了，单凭那些简单方法解决不了这问题，我必须懂些编程技巧才行，否则，用穷举算法解它得花100年时间。"

组合算法非常迷人。对这类算法来说，一个优秀的想法能节省10阶大小的运行时间。但我也不会因此而小看那些只能节省20%时间的想法，因为就算一次循环只节省100纳秒，循环一万亿次也能省下一天时间。如果程序重复使用很多次，那它真能给你带来不小的回报。所以说，你必须学会使用一些精妙而隐晦的技巧。

大约一年前，我在《计算机评论》[①]上看到过一篇文章，有个家伙评论了一本书，书名好像是《编程技巧》之类的。这篇评论的重点在于，他宣称："如果任何一个为我工作的程序员，被我逮到他用了任何一个这书上的编程技巧，我会让他马上卷铺盖走人。"既然他这话都说出来了，我当然要把这书找来看看，因为我觉得："这书肯定很有意思，我得从里面吸取点教训。"不幸的是，那书里的技巧真的不怎么好用。

Seibel：它们真能砸掉程序员的饭碗？

Knuth：是的，它们的确很烂。我没有系统地分析它们，但我觉得这显而易见。那是对编程截然不同的另一种理解方式。不过那个声称要炒人鱿鱼的家伙，他的目的是想让程序以一种低效的方式编写出来，因为编程理应符合他所设想的井然有序的状态。他不关心程序是好是坏——我指的是速度和性能上的好坏——他关心程序是否满足其他条件，比如说，随便找谁都能维护它。好，人们又有了许多有趣的思想。

人们会有这样一种奇怪的思想：他们想写自成一体的程序，别人只需设

---

① 《计算机评论》（*Computing Reviews*）是由美国计算机协会（ACM）出版发行的一本专门对计算机领域作品进行评论的学术期刊。

置几个参数，程序就会做他们想要的工作。所以，会有一小部分程序员编写程序库，也会有人给库写用户手册，还有人使用这些库。

问题是，如果只是在调用库里的东西，如果不能自己写库，那编程还有什么乐趣可言呢？如果编程这项工作就是找到正确的参数组合，而且找它们又没什么技术含量，那又有谁愿意把编程作为自己的职业呢？

可复用软件过于强调了这句话："决不能打开盒子去看看里面有什么。"程序里有黑盒是好的，但通常来说，如果能看到盒子里的东西，弄清楚那东西到底是什么，那我们就可以改进它，让它运行得更出色。但现在的世道是，什么都被严格封装起来，仅仅让程序员看到一个封闭的单元，而不让程序员去剖析研究它。程序员唯一能做的，就是把各部分组装起来。所以你的脑子里全都是：调用这个子程序时，参数是x0、y0、x1、y1，但调用那个子程序时，参数变成了x0、x1、y0、y1。你要把它们一一填写正确，这就是你的工作。

Seibel：会有很多人同意你的说法。是的，自己写代码更有乐趣，但除了乐趣之外……

Knuth：不仅仅是乐趣而已。数学家的工作是给出证明，但在你求解数学问题时，几乎找不到某个定理的假设和你求解该问题所需的假设恰好一模一样。通常情况下，你已经得到了一个类似于书上定理的东西，你要做的就是看看它的证明，然后说："嗯，为了证明我手头现有的假设，我要这样改动一下这个证明。"所以说，虽然数学书里总是塞满了定理，但你永远找不到严丝合缝的那个。实际上你想看到的还是那个需要改动的证明，因为正好找到你想要的定理的概率只有百分之一。我认为这和软件的情况恰好吻合。

Seibel：不过，软件不是人类制造过的最复杂的东西吗？我记得你自己曾经这么说过。

Knuth：我记得是Dijkstra先说的，不过我确实也说过。这是因为把事物拼凑在一起的复杂性太不一致了。纯数学倾向于用少量普适原则，3到4条公理来描述一个系统，而计算机程序有很多组成部分，第一步和第二步不同，第二步又和第三步不同。所有这些综合起来，就不得不用非常复杂的方式制造它。

Seibel：所以说，在这么复杂的情况下，有些时候我们似乎不得不一边用着黑盒，一边说着："好，我们熟悉它的工作方式，我们用得上它。"假如我们必须深入探究每个黑盒的内部构造，那我们什么软件都做不了。

Knuth：我的意思不是说黑盒没用。我是说如果不允许我打开黑盒——如果什么程序都只能用库或类似的玩意来写，那我会写出慢得多也烂得多的程序。

Seibel：开发得更慢还是运行得更慢？

Knuth：都更慢。嗯，这么说吧，用这些程序库我可以急匆匆地赶出我所需要的程序，所以我不可能说它没用。只是由于反复查阅多本参考手册，再从中找到合适的内容，这一过程花费了我大把时间，因此我觉得这不是个编程问题，而是个查找问题。

Seibel：不久前，写标准Java Collections库的那个家伙①写了篇文章，内容是说在他写的二分查找算法②的实现里，怎么会发现一个已经存在了9年之久的bug。简单说来，这算法的实现就是先取最大的索引值，再取最小的索引值，然后相加取和除2。不过显然，如果加在一起的时候溢出，那就出现bug了。不过，尽管标准库存在bug是件很糟糕的事，但他们最终还是发现并且修正了它。假如每个人都自己写二分查找，那写错的概率可能会相当高。

Knuth：没错，而且这样的例子数不胜数。在我们70年代开设的编程课上，二分查找就是个典型的例子。那是在上课的第一天，班上每个人都写了二分查找程序。我们把这些程序汇集起来，由助教检查了一遍，结果发现只有不到10%的程序正确无误。这些程序共有4或6种不同的bug，但不包括你刚提到的溢出bug。我们以前没注意过这种bug——没想到两数相加也有出错的可能。

但是我为什么会如此厌恶黑盒思想呢？刚才我们说到了计算问题，那么假如说，现在你有一个矩阵相乘程序，程序里有个黑盒，执行矩阵乘法运算。然后，你把数据类型从实数变为复数，于是你又有了一个复矩阵乘法的盒子，而且运算从原先的n^3步变成了$4n^3$步。也就是说，如果实数情况的运行时间是t，那么复数情况的运行时间就是$4t$。但是，如果允许你打开盒子，那么由于可以利用同一性来改写复矩阵相乘的公式③，因此这里只需3次实矩阵乘法运算，而不是4次。这只是一个简单的例子，类似例子还有很多。

---

① 就是444页译注中提到过的，Jon Bentley的学生Joshua Bloch，也是本书第5篇的主人公。

② 按Knuth本人的说法，二分查找是“思想何其简单，但实现何其困难”的算法。文中提到的这个溢出bug，只需把mid = (low + high) / 2改为mid = low + (high − low) / 2即可避免。此bug极其普遍，就连Jon Bentley的《编程珠玑》和Brian Kernighan的《程序设计实践》中的例子也未能幸免。

③ 复数乘可以简单写为 $(a+bi)(c+di)=ac-bd+i(ad+bc)$，因此较之于实数乘需要4倍的运算次数。但注意到该式还可化为 $ac-bd+i[(a+b)(c+d)-ac-bd]$，即只需3倍即可。复矩阵乘亦是如此。

用优先队列？还是用某种堆结构？不管用什么，我都能写出一个二分查找的源代码实现。不过每次用到它时，它和我想要的还不完全一样，所以还得再稍做修改。这样做更让我感到轻松自在。我知道，我站到了很多人的对立面上，他们认为自己的任务就是编写别人可复用的代码，这样一来，如果代码出现任何bug，那就非得要他们去修复，并且一旦他们修复完毕，所有程序都会焕然一新。说真的，这样的世界很让我不爽，我还就想看看他们的代码到底写了些什么。

当我写《计算机程序设计艺术》第1卷的时候，人们还没有意识到他们的程序可以使用链表，也没有意识到可以使用数据结构的指针。

如果你手头有个问题超出了数组的解决范围，那你只能求助于别人的软件包，或者改用像IPL-V[①]或LISP[②]这样的解释型语言。也能找到一些Fortran版本的子程序，你可以学习如何使用它们，并用它们来写程序。在当时，如果有人教普通程序员如何给程序写个链表，那别人一定觉得他彻底疯了。那是一个所有程序都应该用现成的子程序堆砌出来的时代。

但通用软件包为了照顾一小部分用户的需求，不得不使用大量额外的处理机制。所以我的书就像擦亮了群众的双眼："哦，我的天哪，我能弄明白这个，我还能在修改它之后马上得到两个元素列表。我居然能改变它的数据结构了。"这一思想取代了仅仅把代码封装在软件包里的思想，而成为时代的主流。

对了，在写BDD结构的章节时，我又发现了同样的事。现在有三四个实现了BDD功能的子程序包，但我《计算机程序设计艺术》正在写的这部分内容，其主旨在于让你也学会如何给大多数应用写个简单版本的BDD程序，而且这程序以后还能不断扩展。你可以把它们用在不同类型的问题中，而不必使用那些软件包提供的华而不实的特征。你用它们写的程序会很容易理解，也会很容易移植到你自己写的程序里。

---

① IPL全称是信息处理语言（Information Processing Language），大约于1956年由兰德公司研究员艾伦·纽厄尔（Allen Newell）、克利夫·肖（Cliff Shaw）和卡内基梅隆大学的赫伯特·西蒙（Herbert Simon）共同设计开发。它是第一个符号操作和表处理语言，首次提出了列表操作、高阶函数、符号计算、虚拟机等编程概念。IPL-V是该系列语言的第五个版本，发布于1958年，是应用最为广泛的一个版本。

② LISP全称是表处理语言（LISt Processor），由约翰·麦卡锡（John McCarthy）发明于1960年左右，是一种基于λ演算的函数式语言。在LISP发展早期，存在着很多方言，各种方言的实现并不完全一样。到了80年代，Guy Steele（本书第9篇主人公）编写了Common LISP，建立了一种LISP的通用标准。Steele同时也是另一种广泛应用的LISP方言——Scheme的开发者之一。

今年头几个月，我写完了关于按位运算的诀窍与技巧的章节，这些内容作为“妖术”在黑客圈内流传已久。我觉得是时候告诉你们真相了，它们背后自有一套理论，而你们完全能够理解这套理论及其组织结构。你能自信满满地独立应用它们；你也能以此为基础，做出一些令人叹服的程序，而就在一年前，你还对怎么做出这种程序一无所知。既然这方面知识至今仍不为人知，不妨就由我把这个本应作为常识了解的知识介绍给大家吧。

我写了很多程序。我不敢妄言它们都很典型，但我敢断定它们足以应对很多种不同情况。假如说，让我把所有时间都花在学习使用别人的子程序上，那我会觉得更难过。先了解一些基本概念，再通过修改过去可用的代码来复用，这对我来说容易得多。

Seibel：*编程从最初发展至今，你认为发生了怎样的变化？*

Knuth：我们前面谈过文学编程，它与传统编程完全不同。我视自己为解说员，向人们讲解文学编程的理念，而不仅仅是把合适的指令归纳出来。Dijkstra提出的发展方向与文学编程相同。结果呢，他写的程序甚至比我的更富有文学气息。当然，我的意思是，那些程序只有人类理解，机器根本读不懂。

他也是为结构化编程做出了极大贡献的人之一。我们知道，结构化编程里的模式既能用于扩大程序规模，也能保持程序员的思路清晰。你写的程序规模扩大了10倍，可是花费的精力并没增加10倍，因为你掌握了一些工具，能让你在大型系统中可靠地把各部分组合起来。在编程领域中，这是个决定性的改变。

所以说，结构化编程的模式是个重点，其关键在于我们必须充分理解抽象的思想。抽象赋予我们处理大型系统的能力，并且让我们有信心掌控系统中错综复杂的局面，看清当前正在执行的任务。尽管摆在我们面前的问题也许极端复杂。

还有许多变化看似重要，对我却没太大影响。它们都流于表面，比如不同形式的语法糖（syntactic sugar）[①]，以及编程语言中存在的各种方言。不过，正所谓仁者乐山，智者乐水，有些人比我的思维更具逻辑性，他们喜欢在程序里使用大量括号，让代码前后匹配，就像代码在说：“我现在开始了。”结束时再说：“我现在完成它了。”我对这种东西不太感冒。那不是我考虑

---

① 这是由英国计算机科学家Peter Landin首次提出的一个术语，指在计算机语言中添加某种语法，这种语法对语言本身的功能并无实质影响，但却对程序员编程提供了便利。比如在C语言中，用a[i]表示*(a+i)、用for表示while，高级语言支持面向对象特性的语法等。

问题的方式，但却可能是别人考虑问题的方式。就思考方式来说，这里不存在优劣之分。

在我看来，C语言对指针的运用是编程语言中最重要的变革之一。当你使用比较复杂的数据结构时，常常需要让结构的一部分指向另一部分，因此，人们琢磨出各种方法，将这种机制引入高级语言。比如Tony Hoare[①]，他实现了一套相当简洁且漂亮的系统。然而C语言所采用的指针却有些与众不同：令x表示指针，则$x+1$表示的不是紧挨着x的1个字节，而是x之后的1个结点。指针向后跳过的字节数取决于x指向的内容。如果指向一个大结点，$x+1$就会往后跳很多；如果指向一个小玩意，$x+1$就只移动一点点。起初，我认为这样实现指针是个严重错误，但渐渐地，我发现我爱上它了。它堪称最令我喜出望外的助记符创新。

Seibel：*和它的前辈相比，它的确神通广大。不过，也正是从那时起，很多人认为使用裸指针（raw pointer）指向内存是相当危险的，他们更倾向于使用行为类似指针却没指针那么危险的引用。*

Knuth：如今的指针已是风光不再，我也对它颇有微词。因为我发现在我的64位计算机上，如果真在乎性能，那还是别用指针的好。因为这台计算机的寄存器有64位，内存却只有2GB，所以32位指针就足够了[②]。然而现在我每次使用指针，都要用掉64位的内存空间，数据结构的大小也会增加一倍，更糟的是，它还进入缓存，我的一半缓存就这么没了，兜里的钞票也这么没了——缓存贵啊。

所以，如果我想在性能上有所突破，那只能用数组取代指针。我写了一些复杂的宏，这样看上去像在用指针，但实际上没用。从某种意义上说，指针只是个小玩意，并且已经落后于时代。但对我来说，它是面向底层助记符的重要思想。我在工作和调试的时候，心中仍然充满了对Thompson和Ritchie[③]的感激之情。我想不出还有谁曾提出过如此出类拔萃的思想。

---

① 托尼·霍尔，英国计算机科学家，1980年图灵奖得主。他最著名的成就是在1960年26岁时设计了快速排序（quick sort）算法，此外他还参与了ALGOL 60语言的研究与实现，开发了用来验证程序正确性的形式化逻辑系统——霍尔逻辑（Hoare logic）。2009年，霍尔为自己在1965年设计ALGOL W语言的类型系统时引入了空指针而道歉，因为空指针这个概念从此得到广泛应用，也因此而造成了巨大损失，他把这个错误称为“十亿元的错误”。

② 2GB内存需要21位地址映射，就算可能存在虚拟内存和其他映射，32位指针也足够了。

③ Ken Thompson和Dennis Ritchie，著名程序员，一起开发Unix操作系统，一起设计C语言，一起获得1980年图灵奖。Thompson也是本书第12篇的主人公。

Seibel：你的编程工具箱里还有什么宝贝吗？

Knuth："变动文件"（change file），它是我实现文学编程时做的东西。我不知道其他程序员的工具箱里有没有类似工具，所以我得先解释一下它的概念。

在我完成TeX和Metafont以后，大家都来管我要。他们用的编程语言、操作系统和计算机可谓五花八门，组合起来有200到300种之多，而我又想让程序能轻松适应任何一种组合。于是，我提出一个解决方案：写一个主程序，运行在斯坦福的服务器上，然后用一个称作"变动文件"的插件，根据每个人不同的系统需求定制软件。

"变动文件"由许多小变动组成，它非常简单。其中的每个变动都用几行代码开头，你要找到主文件里和你某个变动的第一行匹配的那行，从该变动第一行直到第一部分结束的这部分代码，应该和主文件里刚才匹配的那行后面的几行匹配。接下来，是该变动的第二部分，它告诉我们："用这部分替换主文件里匹配上的那几行。"

该变动或许会说："匹配上的话就用这12行替换那6行，否则就什么都不改变。只要你找到主文件里的一个匹配，你就把匹配上的那部分更换成你想要改的这12行，然后继续寻找下一个匹配。"你必须正确编写这些变动——它不会进行任何智能匹配，而是只会说："找到下一个变动，使它的第一行与主文件中的某行相同。"

这个系统只花1小时就能编出来，然而它足以达到我的目的。所有文学编程工具，包括"织出"和"混出"[①]程序，都要使用主文件和变动文件。

有时候，我不得不发布新的主程序。全世界几百号人都有自己的变动文件，可能在这些变动文件中，有些原来能匹配我主程序的某6行，但现在不再匹配了，所以他们也许也得做些修改，但不用改太多。每当我修正bug，这一修正都会自动生效——bug修正也应用到了他们的程序中。变动文件可以简单有效地解决问题。任何人都能把它琢磨透，而且可以自己实现一个。

把TeX改成支持Unicode的版本是个极端的例子。他们用的变动文件可能10倍于主程序那么长，换句话说，他们是把程序从8位改成了16位。但他们

---

① "织出"（weave）和"混出"（tangle）是用于从文学程序源代码中获取不同表达方式的两种工具，其中"织出"可产生供人类阅读的文档，"混出"可产生供计算机编译的代码。这两种工具的英文名称"weave"和"tangle"均出自于苏格兰历史小说家、诗人沃尔特·司各特爵士（Sir Walter Scott）笔下的史诗《Marmion》中的名句"啊，从我们第一次撒谎，就在编织（weave）一张错综复杂（tangled）的网。"

是在变动文件里写全部的代码初稿，而不是从头到尾重新实现主程序。他们把这个变动文件称作Omega。相对于TeX的2万行代码来说，该变动文件有上百万行之多。所以说，这是个极端的例子。

现在我还一直用着变动文件，因为在我写书时，我会写很多程序——我想解决的问题很多，而我又想在这些问题上多试验几个不同版本的程序。比如说昨天吧，我想求出n位二进制数的乘法运算需要多大规模的布尔电路[①]。我有个程序，它接受任意布尔函数作为输入，然后输出该布尔函数的BDD大小。也就是说，我现在已经有了一个用任意布尔函数计算BDD的程序。

我原来的程序里，你要在线输入函数的真值表——它会告诉你："给我一张真值表。"然后，我输入一个十六进制数，因为我在书中用作例子的小函数很多。但只有我在真值表里给某个小函数设了值，它才能计算这个小函数的BDD。

然后，我得到了一个大函数，形如"把每一对8位二进制数相乘"。这个函数有16个变量——来自x的8位和来自y的8位。所以我写了个很小的变动文件，可以用自动生成乘法真值表的程序来替换交互式对话框。

然后，我一边改着文件，一边念叨着："让我们从右往左而不是从左往右读，这样的话，我会得到一个不同的BDD。"或者："让我对所有6变量布尔函数做个试验，我想看看它们的结果，找出其中哪个生成了最大的BDD。"所有这些程序，都是对我原来那个程序的定制。

我会给原来那个程序实现出大约15种变形，它们都很容易理解。这是文学编程给我带来的意外惊喜，用分发主文件的方式来适应人们的各种系统。但我现在却用一种完全不同的方式使用着它。

Seibel：*我好像有点明白，它为什么对你手头的工作有用了。你想针对不同的主题做出很多变形。*

Knuth：是啊，我手头正在写书。

Seibel：*你认为这种机制会得到更加广泛的应用吗？*

---

① 布尔电路是用于研究计算复杂度的数学模型，它实际上是一个有向无环图。若用它描述一个具有n个输入位和1个输出位的布尔函数，则图中包含n个代表输入位的输入结点和若干代表与、或、非门的门结点，门结点中还包含1个代表输出位的输出结点。布尔电路的大小和深度是非常重要的计算复杂度度量，其中大小是指电路中与门和或门的总数，深度是指从输入到输出的最短路径。过去很多研究者曾认为，通过证明任一NP完全问题的电路大小的下界，可以解决NP与P的关系问题。但1994年，Razborov和Rudich在其著名论文《自然性证明》（Natural Proof）中指出，在单向函数存在的前提下，不可能分离NP和P。他们也因为这篇论文而获得了2007年的哥德尔奖。

Knuth：我不知道。如果我身处50人的团队之中，我不敢确定它会如何发挥作用。不过我希望作为每一个程序员个体来说，因为求知欲而编程的想法不会慢慢消逝。

Seibel：*在你编程初期，你写的是机器码；然后，你发现了结构化编程，顾名思义，它给你提供了一种组织程序的结构；再然后，你发明了文学编程，它给了你另一种规划程序的方式。自从文学编程发明以来，你认为编程方式还有什么别的重大发展吗？*

Knuth：我做了一个更好的调试工具，是给文学编程使用的。基本上就是这样。

Seibel：*好，那我们来聊聊调试。你所说的更好的调试工具是什么？*

Knuth：GNU调试器（GDB）的发明者竟然意识到，可以用预处理器[①]编写程序。这样一来，你就能在完全不同的语言里，把低级语言和高级语言的代码关联到一起。所以，我虽然用CWEB写程序，但我不必考虑那些用低级代码写的东西，因为在我单步调试程序时，CWEB程序的行为会充分显现出来。

Seibel：*也就是说，CWEB利用了GDB内建的机制？*

Knuth：因为CWEB是用C语言构建的，也有#line指令，所以它也可以利用GDB的机制。我们必须用#line指令工作，不过它干得很漂亮。计算机只能默默地接受二进制指令，可GDB却知道指令来自于WEB源文件的某处，尽管WEB要比C语言晚问世10到20年。所以说，GDB非常了不起，它具有一些高瞻远瞩的设计，使它有能力调试晚于它出现的语言。

Seibel：*这正是你使用GDB的原因所在。你还用过哪些调试技巧？*

Knuth：我会加入很多代码，来查看我程序里的数据结构在带有数据冗余的情况下是否能够正常工作。如果开启完好性检查[②]，那么整体性能有可能会下降为原来的百分之一。

---

① C语言的预处理器一般作为一个独立程序由编译器调用，其功能是在开始编译之前，预先处理源代码中的预处理指令，如文件包含（#include）、条件编译（#if、#elif、#else、#ifdef、#ifndef、#endif）、宏定义（#define、#undef）等。后面马上将要提到的#line也是其中之一，它的作用是改变当前行号。

② 完好性检查（sanity check），也称作完好性测试（sanity test），是一种快速验证系统功能的基本测试方法。它是一种较为简略的测试，关注的只是一部分功能，是回归测试的一个子集。在实际应用中，一般采用可快速执行的简单方法，筛掉那些不满足某项条件的结果，以备后面进一步测试或其他需求使用。

比如说，我有个包含引用计数的复杂数据结构。所以，当我写一些复杂的程序时，确保这些引用正确计数是件头疼的事。有时，我要让引用计数增加，有时又要减少。可如果指针在寄存器中，或作为子程序的参数，它是否要作为数据结构的引用参加计数呢？因此，我写了个完好性检查，它遍历上百万引用计数中的每一个，看看其中到底有多少引用是真正创建出来的，引用计数的数字是否正确。然后，我会做一些计算，检查整个程序。这样一来，错误在因为崩溃而浮现出来之前，已经经历了数十亿次检测。

我手头有个乘法程序，是用以前没测试过的新算法写出来的，所以，我对它进行了详尽的测试。我生成256个数字，把每对数字两两相乘，不过每次乘完之后，我都做完好性检查。我用3乘以2——失败！因此，我修正了程序，然后再继续测试其他数字。最后，我终于使程序在所有的256×256个数字配对上都能正常运行，而且可以取得正确的结果。

所以对我而言，这是个重要的调试技巧。也许10%的程序代码只有在调试时会用到。此外，完好性检查的代码也会记录数据结构的验证结果。

我还会写程序把数据结构用美观的符号形式展现出来，这样我就不用解码一长串二进制的代码。如果有必要的话，我还可以用适当的格式把数据结构打印出来，或者转存到文件后再写个程序去分析它，看看里面到底存在什么问题。

Seibel：*关于不变式和各种断言，以Dijkstra为代表的人会争论说，我们必须为程序的每一步设置形式化断言，用来证明我们的程序正确无误。我曾读过你的一篇文章，你想证明自己写的程序“非形式化地正确”。你是否认为应该更进一步，形式化地证明程序正确呢？*

Knuth：一方面，你不可能证明程序的正确性。某些人声称他们验证了一些程序，说程序只有在符合验证程序的特定规范时才可被验证。但是，验证程序里也许也有bug，那些规范里也许也有bug，所以，你永远无法知道程序是否正确。你可以找出它正确的大把理由，但找理由这过程永无止境。因此从理论上说，这类证明不可能。

最早的有关形式化证明的论文是Tony Hoare写的，论文的名字是《程序证明：FIND》（Proof of a Program: FIND）。这篇论文的意义非常重大，极大推动了该领域的研究水平。但他论文中的证明有两三个错误。他们没遇到过这种情况：必须保证数组下标不会越界，以及其他与此类似的事情。只要沟在那儿，就总有掉进沟里的可能。不过和当时其他人相比，他所做的验证工

作还是多得多。

我昨天刚编了个程序，但我不知道怎么在程序里写哪怕一个断言。如果非要我写断言，那我宁愿不编程，因为我对断言的内容和程序的内容一样没信心。

要不就拿TeX来说吧，它的形式化就很糟糕。它本来就是给人用而不是给计算机用的。对TeX是否正确所下的定义会让人摸不着头脑。某些形式化语义方法实在是太复杂了，以至于没人能够理解正确性的定义到底是什么。

Seibel：*当你编写TeX的时候，测试它让你感到既烦躁又痛苦。*

Knuth：对。

Seibel：*你是怎么把测试纳入你的思维框架的？因为程序员们通常倾向于保护他们的劳动结晶，所以他们往往不会尽其所能地去测试。*

Knuth：嗯，我得承认，我一直是个喜欢吹毛求疵的人。因此，我必须忘掉我是这程序的作者，然后我就可以充分享受寻找错误给我带来的乐趣。我试着想象别的什么人是作者。除了乐趣之外，这样做还很容易让我进入批评别人的模式。我也不知道这是为什么。

举个例子，我为巴勒斯公司[①]做过的最棒的工作，是给他们的硬件设计排错。他们的工程师会把计算机的规范说明书给我看。我盯着计算机，心里琢磨着怎么找个例子，能让它产生越界之类的错误。尽管B-5000系列计算机已通过模拟器测试，但在它正式投产前，我还是找出了200多个错误。

Seibel：*从本质上说，你做了一个编程语言的语义正确，但计算机执行不正确的程序？*

Knuth：是的。如果它们的浮点运算无法算出两数的正确乘积，我就要试着找出计算错误的浮点数的例子。他们还有在硬件中实现栈的情况，但寄存器有的为空，有的不在栈顶，于是我要去发现他们逻辑混乱之处。

---

① 巴勒斯公司（Burroughs Corporation）是美国主要的办公设备制造商，1886年由William Seward Burroughs I创立。创立之初主要生产和销售他自己发明的机械式加法器，还兼营打字机等办公设备，它很快成为了全美最大的机械式加法器公司。1953年，巴勒斯公司的业务开始转向大型计算机，此外，还向客户提供包括打印机、磁盘驱动器、磁带驱动器、打印纸等在内的一系列计算机相关产品，名列美国八大计算机公司之一。值得一提的是，自1961年起，该公司在其“面向语言设计”的理念指导下，生产了以B-5000为首的一系列堆栈式计算机，可支持ALGOL、COBOL、Fortran等各种高级编程语言，并且由于该系列计算机在技术上有所突破创新，因此学术界也对其评价颇高。1986年，巴勒斯公司与斯佩里公司（Sperry Corporation）合并，成立了优利公司（Unisys）。

Seibel：你做这些有没有一套系统的方法？你是怎么发现它们的？

Knuth：我不知道自己是不是一个难以相处的家伙。但我会觉得，证明某样东西是错的更容易一些，比如数学定理之类。我通常只用说"好，我找到了一个反例"就行。我在发现定理中的漏洞，或解释定理无效的原因时，会感觉非常兴奋。然后，当我再也找不到更多的漏洞时，才会去看看这定理的证明。

我认为我的性格特点就是热衷于抨击和挑错。我喜欢跟别人对着干，那种感觉对我来说好像一针强心剂。我不会就那么坐着，附和着说："啊，是这么回事。那它为什么好使呢？"

Seibel：这很奇怪，批评别人和给别人挑错给了你不断前进的动力，但你毕生最重大的成就却是解释事物。批评和挑错是否在某些方面有助于你解释事物？

Knuth：唯一可以确定的是，在解释事物的时候，我努力遵循大脑的自然认知过程，同时从正反两方面认识事物。这样，我就可以更深入地理解事物。我认为关键在于把视野放宽，从各个角度全面地认知，而不是仅仅从一个角度片面地认知。我不知道这对我的批评业务有怎样的影响。

但是，当你批评别人的时候——只要你与人打交道，就必须敢于挑战——你体内的激素之类的东西会被唤醒。这种激素可以激发大脑的活力，让你容易找到更多的批评方法。一个优秀的解释也是这样，能通过某种途径综合各种不同的观点。

Seibel：还有一件事，也来自于你为TeX所做的工作。你在《TeX的错误》一文中，描述了你把自己在程序里发现的每一个错误都记录到了日志里。软件工程研究所[①]的人会说，作为成熟的软件流程的环节之一，我们应该跟踪每一个bug，并且知道如何去避免将来发生类似的错误。但你却说过，维护这样的日志对你预防错误没什么帮助。

Knuth：是的。不过有这么个日志也不会让我的程序更烂。

① 1984年在美国国防部资助下，软件工程研究所（SEI）在卡内基梅隆大学成立，主要任务是研究如何分析和评估软件厂商的软件开发流程，从而提高其开发能力。能力成熟度模型（Capability Maturity Model，CMM）就是该协会研发的一套以软件流程评估和软件能力评价为基础的模型，可协助组织改善软件开发流程，提高软件开发的组织管理能力。实际应用中，CMM既可作为评价承包方执行能力的参考标准，也可作为软件企业改进流程的参考模型。

Seibel：但你不会有这样的想法："啊，既然我已经知道了这个错误，我就不会让它再发生。"

Knuth：我必须认清我的原罪（sins）。人们不断回想过去发生过的事情，目的是解罪（absolution）。这是个宗教术语。

Seibel：你发现自己的程序出现了和以前一样的bug，然后你会说："啊，我又犯了同样的错误。"

Knuth：是的。

Seibel：为什么会这样呢？犯错误这种事就是有让人们难以吸取教训的本质吗？假如可以从中吸取教训，我们本可以避免再次发生同样的错误。

Knuth：我觉得这很可能因为我乐于尝试更困难的事情。我总在挑战自己的极限。如果让我再写以前写过的那种比较容易的程序，那我不会出这么多错。但是，既然我的知识比以前丰富了不少，我就要试着写些更难的东西。所以说，我会出错是因为我总是游走于极限边缘。如果仅仅一直安于现状而不思进取，那也未免太无聊了。

Seibel：假如说，就让你在余生之中一直编写排版系统呢？

Knuth：那好，我会干得很漂亮。不过，人们的标准和要求在不断提高，总有一天我们会被绊倒。我们要去解决的问题——正像前面提到过的那样——恰好位于人类可解决问题的边界上，而且比我们先前解决的问题更复杂。

我们不会把自己局限在简单问题上。因为人类总是渴望超越边界，到达新的未知领域，所以局限于简单问题会让我们心有不甘。但是，一旦到达了新的领域，我们势必还想继续超越下去。永无止境。

所以不可避免地，我们要出些错误，除非我们下决心永远不写任何挑战自身能力极限的程序。可是这样的话，我们又如何进步呢？每三年就会出现一个新的流行词，号称可解决所有问题且卓有成效。极限编程就是过去两三年间出现的流行词，在它之前，还有其他词也是如此。人们提出一个又一个看似有效的"银弹"，也有很多人成为它们的忠实拥趸，不过他们后来发现："啊，还是那么难。"

Seibel：随着时代发展，可成为优秀程序员的人有什么变化吗？

Knuth：以我多年调查实践的经历看，优秀程序员的比例几乎稳定在百分之二。也就是说，每次我调查时，如果从某个主修专业为非计算机的群体中抽

取100个人，那么其中会有2个人，在与计算机产生真正共鸣的意义上是程序员。阿拉斯加的瓦西拉市[1]有1万人，所以差不多有200名程序员。

Seibel：那么，编程是否已经发生了足够大的变化，使得那百分之二的人也有所变化呢？还是说，编程仍旧一如往昔？

Knuth：我不知道——你用的“编程”这个词有两种不同的含义。我们总是在制作工具，这工具由人与人之间的分工合作，共同配合完成计算机程序。我说的主要是当计算机超出其能力时真正运行的方式，而不是仅仅得出答案时。

我们现在已经有了非常强大的计算机，强大到足以让那些以我内行人的眼光看不擅长编程的人也能得到答案的程度，而这些答案在老式计算机上需要很专业的人才能得到。但是，真正的程序员会用新式计算机去解决那些用老式计算机无法解决的问题。

所以，存在着这样一种改变，而这种改变正是我所担忧的：如今很多编程方式都索然无味，因为你要做的只是插入一些魔咒——和别人写的程序组合起来，然后运行。这用不着多少创造力。我担心编程会变得越来越枯燥，因为你得不到什么创新的机会。你的快感来自于计算机在你眼前产生一些有趣的结果，那种快感和我通过不断创新获得的快感是两码事。现在的所谓快感，就是在完成一系列枯燥乏味的工作之后，突然就能在显示器上显示个图片什么的。以前的工作可没这么枯燥。

Seibel：但是，你仍然找到了让自己做起来感到有趣的那种编程？

Knuth：谢天谢地，是的。我的内心已经获得了对编程的迫切需求感。早晨，我带着文学程序的句子醒来，吃早饭之前——我相信诗人一定有这样的感觉——我必须坐在计算机前，把这段代码敲进去，然后我才能吃得舒坦。我必须承认，这是一种强迫症。

好吧，给你看看我昨天写的程序。我把比世界上最大的数还要大的数乘起来——它们是一种特殊的整数，你可以压缩它们的表示形式，所以就算不能用常规符号表示，我还是能处理它们。我已经在这些大得难以置信的整数上做了乘法和平方运算，并且看到了它们平方后的样子。实际的计算结果很让我大惑不解，但也很激动。

① 瓦西拉市（Wasilla）是美国阿拉斯加州南部的一座小城市，离阿拉斯加最大城市安克雷奇大约72公里。据2007年统计，该市人口约为9780人。

Seibel：你是个科研人员，但你也开发过大型系统，在产业界颇有建树。你怎么看学术化的计算机科学和生产实践之间的关系？

Knuth：此一时彼一时。60年代，学术界远远领先于产业界。大学里的每一个人，都觉得产业界打造出来的程序滑稽可笑，可能只有飞机订票系统[①]是个例外。

到了1980年，情况几乎颠倒过来了。大学里的人写出来的程序被产业界的人嘲笑，因为大学编程已陷入了一种神学模式[②]之中，绝不允许使用一点goto语句。我对这一情况的描述有些夸张，但从根本上说，大学编程的这个不能、那个不许束缚了人们，而产业界的人用不着考虑这些。

但此后，大学里的人又提出一些先进思想，涉及网络和大块数据处理等内容，从而再次取得领先。所以二者可说是你追我赶，交替领先。不过，算法和数据结构社区提出的很多发展方向都不太合我口味，因为其中有很多数据结构非常地……巴洛克[③]，这是我唯一想到的词。它们复杂而精巧，你不得不钦佩它们的技术水平，可我却觉得它们空洞无物。它们与现实生活毫无联系。它们运行在另一个世界中。那是一个还算凑合的世界，有自己的组织结构，居民也都友好且体面。但从个人角度来说，那个世界对我没有吸引力，它的的确确与实践无关。

不知道为什么，事物是否与实践有关对我来说这么重要。有些数学家从不考虑任何有限事物，而且甚至都不怎么涉及可数无限[④]——他们发表的绝妙论文仅仅讨论一类极其复杂的无限，并且他们能把这种无限深入浅出地讲解清楚，这给了他们极大的满足感。算法也存在类似事情。不过对我而言，如果能把想法用自己的计算机实现出来，那我会兴奋得多。

Seibel：你在1974年说过，到1984年的时候，我们会拥有“Utopia 84”（乌托邦84）——一种完美的编程语言，它将取代COBOL和Fortran。你那时说，有迹象表明，这种语言的成型将极为缓慢。如今，距离1984年也已经过去20年了，看起来它还是没有成型。

---

① Sabre公司1964年发布的系统，可实时响应遍及全球的代理业务。

② 文中所说的神学，指的是把编程视为一种宗教思想或者信仰。

③ 巴洛克指的是17世纪初至18世纪上半叶流行于欧洲的主要艺术风格，追求一种繁复夸张、富丽堂皇、气势宏大、富于动感的艺术境界。文中用以比喻数据结构的华而不实。

④ 可数无限集指的是可与自然数（或整数）建立一一对应映射的集合，也称作可列集。按这一定义，整数、代数数、有理数都是可数无限或可列的，而实数是不可数无限或不可列的。此外，还可以对不可数无限集再作划分，构造出比实数集无限程度更高的集合。涉及无限的概念是反直觉的，也是数学中较为抽象的概念。

Knuth：是的，还没有。

Seibel：当时下的结论仅仅是由于年轻人的盲目乐观吗？

Knuth：我在写那篇文章的时候，肯定正在思考Simula和面向对象编程的发展趋势。我认为每次新语言的出现，都会清除我们对旧语言的理解，然后再加入一些新的、实验性的特征等，而且，没有人会停止理解一门新语言的脚步，他们一直都想理解得更深入。

也许有一天有个人会说："不，我不要革新，我只要简单明了，我会坚持这么做。" Pascal以这种哲学起家，但他并没有将其发扬光大。也许我们会来到另一个时代，有人说："让我们把眼光放低些，试着做个保持稳定的东西。" 这可能是个好主意。

Seibel：语言存在错误特征还不算什么问题，更严重的还有遗漏特征。如果遗漏了一个特征，那你必须拼凑出个东西来填补漏洞。

Knuth：是的，你说的没错。语言必须能以某种方式扩展。Java的可扩展方式就不太好。

Seibel：你独立设计过一些语言，这其中应用最为广泛的，可能就是TeX。

Knuth：对，TeX是一种编程语言，但我还必须加入其他一些刺激和令人兴奋的特征。Guy Steele[①]、Terry Winograd[②]、Leslie Lamport[③]等人，在用TeX

---

① Guy Steele，美国计算机科学家，在多种编程语言——包括Common Lisp和Scheme两种主要的Lisp方言——的设计、标准制定和文档编写等工作上做出了卓越贡献。他也是本书第9篇的主人公。

② Terry Winograd，美国斯坦福大学计算机科学系教授，主要研究领域是人机交互和软件设计，早期在人工智能领域也颇有成就，曾在1968～1970年开发过一个理解自然语言的程序"SHRDLU"，可在一个具有一些块状物体的场景下和人类完成简单对话。值得一提的是，他还是Google创始人之一——拉里·佩奇（Larry Page）在斯坦福大学的博士生导师，但拉里·佩奇1998年中断学业，和谢尔盖·布林（Sergey Brin）一起创建了Google公司。不过他们之间仍保持着良好关系，拉里·佩奇曾多次在演讲中感谢导师给自己提供了一些非常好的建议，帮助自己开拓了视野，Terry Winograd也在2002年作为访问学者，参与了Google有关人机交互理论与实践的研究。

③ Leslie Lamport，美国计算机科学家，在分布式系统领域有着杰出成就。他在著名论文《分布式系统中的时间、时钟以及事件排序》（Time, Clocks and the Ordering of Events in a Distributed System）中，提出"逻辑时钟"（logical clock）的概念，通过在进程间同步逻辑时钟，来判断事件发生的先后顺序。在另一篇论文《如何制造一台正确执行多进程程序的多处理器计算机》（How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Program）中，他提出"顺序一致性"（sequential consistency）模型，要求各处理器的进程按某种顺序执行所有指令，且所有进程看到的顺序均相同，这样一来，就可确保各处理器并行执行的正确性。此外，他还编写了LaTeX宏包，可以把版面内容和文档内容分开处理，使TeX变得更加简便易用。

作为他们所做程序的前端时，都要用到这些特征。我知道当时Winograd正在写一本关于自然语言语法的书，所以他写了一些威力强大的宏，想让这些宏来生成他书中的图表。刚开始的那段时间里，这些特征极大地推动了TeX向着一门编程语言的方向发展。

Seibel：*你有没有后悔自己没把它更多地作为一门语言来设计？*

Knuth：我不知道，也许后悔过。我对那些通用的语言心怀不满，因为它们采用不同的方式“通用”。这有点像Unix，有30种正则表达式定义同时存在——当你使用Unix的不同工具时，你需要遵循的正则表达式定义也会略有差别。如果每一个你手头的工具都包含了图灵机，你会纳闷这到底是什么路子。我一直觉得TeX自身的编程能力越强，执行排版任务的机会也就越少。

当我把质数计算的功能添加到TeX的手册中时，我觉得TeX不应该用来计算质数。我那时是这么想的：“嘿，顺便提一句，看看这个：狗能用后腿站，质数能用TeX算。”

Seibel：*但人们正是利用TeX是图灵完备[①]的编程语言这一事实来做排版相关的计算。如果它不是图灵完备的，那他们就做不了那些计算了。*

Knuth：是的，你说的没错。60年代，我为仿真写了一种编程语言，结果我落到不停kill[②]用户的境地，因为它的用户实在太多了。后来更强大的Simula问世，于是我对人们说，别再用我的SOL语言了。我几乎想都没想过我在设计语言方面有出众的才能。

通过TeX，我与前人数百年来不断积累的经验联系起来。我可不想对图书设计师们潜心钻研好几个世纪得到的宝贵财富视而不见，不会一切从头开始并且说着：“嗯，忘记那些家伙吧，我们现在要的是逻辑性。”这种情况的关键在于，要解决的问题极为复杂，必须找到一个较小的原语子集来辅助。我用了差不多100个原语，而不是可能出现的1000个。当然，若是出于数学形式简洁的目的，我应该把原语数量减少到50个或者10个，不过真要那么少，我想TeX就没法运行了。图书排版问题是世界上最难的问题，对其进一步简化的难度很大。

---

① 图灵完备指在可计算理论中，编程语言、指令集或细胞自动机等计算系统具有等价于通用图灵机的计算能力。也就是说，如果某种语言图灵完备，则它可以和通用图灵机互相模拟。

② Unix和类Unix操作系统中的kill命令是给某个进程发送信号，其默认操作一般是中止进程。文中提到的kill是指连接到主机的进程太多，他不得不中止一些进程。

Seibel：*我没有正式地调查过，但现在好像绝大多数的数学论文和科技论文都在用TeX排版。当你看到用TeX排版的东西时，一定会想："哈哈，我的程序帮上忙了。"*

Knuth：嗯，费马大定理的证明就是其中之一，它也是最著名的数学论文。常常有作者对我说："要不是有TeX，我根本就不会动手写这本书。"排版的这种情况，和黑盒又有些相似之处。

搁在过去，你得先用打字机打出稿子，然后把稿子交给排字工人，他处理完了，再把长条校样[①]交还给你，等等等等。和你打交道的各色人等都不是数学家，你得经受他们的考验，然后你的作品才会最终出版问世。所以，任何一件让那个行业中的人感到迷惑或者为难的事，你都要躲得远远的。[②]

但是，如果你能亲眼看到它排版出来会是什么样子，而且你还能自己创造出别人样式表里没有的符号，因为这符号恰好只适用于你眼前的问题，那么，你就有勇气干得更好。

因此当我了解到，人们在使用TeX之后，不仅排版工作变得更加省时省力，而且他们的创造性思维还能更直接地传达给读者时，我心里一直有极大的满足感。

Seibel：*你希望程序员和计算机科学家充分了解我们这个领域的历史吗？尤其是这段历史本来就谈不上有多悠久。*

Knuth：我们没有多少学识渊博的学者。我1963年开始写书时，甚至觉得人们不会知道1959年发生的事情。我上个星期在《美国科学家》杂志上看到，有人重新发明了一个Boyer和Moore[③]在1980年提出的算法。我们对自己所拥有的辉煌历史知之甚少，这种事情总在发生。对于许多年轻的程序员来说，想去了解一两件70年代的事会被人当作异类。

在这样一个复杂的领域中，人们难免忘东忘西。好在有维基百科这样的东西，成就不会像过去那样被人淡忘。不过我还是希望，我对阅读原始材料的热爱之情能感染更多的人。你不仅要知道某项成就归功于谁，还要回头看

---

① 长条校样（galley proof）用于校对，一般可根据页面行宽和行距排成与版面不符的长条，以便于改动。

② 暗指公式、符号什么的很容易把他们弄晕，不知道如何排版。

③ Bob Boyer和J.Strother Moore都是美国计算机科学家，同为得克萨斯大学奥斯汀分校教授。1977年，他们共同研发了一种快速高效的字符串搜索算法，其时间复杂度在最好情况下是$O(n/m)$，最坏情况下是$O(n)$，其中n是字符串长度，m是模式串长度。相比于时间复杂度为$O(n)$的KMP算法，其运行速度又有不小提高，尤其是在模式串较长的情况下。

看他是怎么用他自己的话表述出来的。我认为，这是提高自身能力的极好方法。

能够深入到别人的思维方式之中，并对他们使用的词汇和符号进行解码，这是非常重要的。如果你可以理解别人思考事物的方式和发现事物的途径，那你就可以借助它们做出自己的发现。我常常阅读一些原始资料，上面记载着历史上曾经出现的天才们说的，有关他们思考和发现事物的方法。按今天的习惯看，他们的表达方式有些特别，但不管困难有多大，我为了理解他们的符号、领悟他们的思想所付出的努力都是值得的。

举个例子，我花了大量时间查看古巴比伦人的手稿，想了解4000年前，他们是如何描述算法的？描述过程中，考虑的事情都有哪些？他们有while循环或类似机制吗？如果有循环，又是怎么描述它的？这对我理解大脑的工作方式，以及他们发现事物的方式都有极高价值。

20年前，我找到一份13世纪关于组合数学的古梵文文档。和作者同时代的人，几乎都理解不了他笔下的内容。但我找到一篇这份文档的译文，它向我诉说了一切。在我刚开始编程那会儿，我就已经有了类似的思想。所以说，阅读原始资料极大地充实了我的生活，增强了我的创造力。

我无法把这种思想传授给我的任何一位学生。如今，计算机科学领域中还有善于阅读原始资料的人在世，但已经不多了。不过，如果单算像我一样热爱阅读原始资料的人，那简直是屈指可数。

我手头搜集了很多成套的源代码。我有一套编译器，19世纪60年代的Digitek编译器，它们的编写方式十分有趣。它们自己有语言，使用30个字母长但描述性很强的标识符。当时，Digitek编译器在竞争中远远胜过其他编译器——这家公司代表了1963年、1964年编译器的最高水平。

我这里还有Dijkstra为THE操作系统写的源代码。我没仔细读过，只是浏览了一遍。不过我敢肯定，阅读它的经历一定会非常有趣。正因为这样，我才会把它珍藏起来。

有一回，我摔断了胳膊——从自行车上摔下来。随后的一个月时间，我都不能干什么活，所以我利用这段时间阅读了一份源代码。我听别人说其中有一些精妙的思想，但它们并没有形成文档。我觉得所有这些，对我来说都是极其重要的经历。

Seibel：我们都知道，阅读源代码很让人头疼，就算用自己惯用语言写的也不例外。你是怎么解决这个难题的呢？

Knuth：很让人头疼，不过为了培养这种能力，头疼也值了。我是怎么做的

呢？以前有台计算机，名叫Bunker Ramo 300，别人告诉我，它用的Fortran编译器速度出奇地快，可谁都不明白其中的原因。我弄到一份这编译器的源代码副本。我连这台计算机的手册都没有，所以，我甚至无法确定它用的机器语言是什么。

但是，我把它看成是一个有趣的挑战。我能琢磨出BEGIN在哪儿，有了BEGIN，就可以开始破译。有些操作码的助记符是两字母的，所以我渐渐明白："这里可能是个装载（load）指令，那里可能是个分支（branch）。"而且我知道，它是个Fortran编译器，所以它有时会查看卡片的第七列，来分辨该行是不是注释。

三个小时过去后，我有点明白这台计算机了。我还发现了一些庞大的分支表。所以说，这编译器就像个谜题，我不断地画着简单的图表，就好像我在国家安全局上班，正在辛辛苦苦地破译密码。但我知道它的速度的确很快，我还知道，它是个Fortran编译器。从有意隐藏信息的角度来说，它没有加密，只不过我手头没有这台计算机的手册，所以它用机器码写出来就如同密码一样。

最后，我终于明白这个编译器为什么如此之快了。不幸的是，这并非因为它的算法出类拔萃，而是因为它用的是非结构化编程，并且代码被人工优化到了极致。

这就是你解决这一类未知谜题的基本方法——先画出表格和示意图，然后通过图表获得一些信息，再做出一个假设。一般来说，在我阅读技术文献时，必然会迎接这种挑战。我试着模仿作者的思路，试着理解文中概念的意义。你阅读别人的材料越多，未来开拓创新的能力就越强，至少在我看来是这样。

我们理应公布代码。《莱昂氏UNIX源代码分析》（*The Lions Book*）是我们可善加利用的书。多亏了苹果公司，Bill Atkinson的程序现在也可以公开获得，我们不久之后就将能看到它。它有完善的文档，还有很多先进的图形算法。

Seibel：开源之后，可供阅读的代码肯定比以前要多得多了。

Knuth：对，你说的没错。不过，多种不同形式的编程记法还是有用的——不要只阅读那些编程习惯跟自己一样的家伙的代码。

（编辑：朱　巍）

参考书目

The Art of Computer Programming, Donald Knuth (Addison-Wesley, 1997)

Beautiful Code: Leading Programmers Explain How They Think, Andy Oram, Greg Wilson (eds.) (O'Reilly, 2007)

Byte, Vol. 6, No. 8, "Smalltalk issue," August 1981

Code Complete, Steve McConnell (Microsoft Press, 1993)

Compiling with Continuations, Andrew W. Appel (Cambridge University Press, 1992)

The Design and Analysis of Computer Algorithms, Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman (Addison-Wesley, 1974)

Design Patterns: Elements of Reusable Object-Oriented Software, Eric Gamma, Richard Helf, Ralph Johnson, and John M. Vlissides (AddisonWesley Professional, 1994)

A Discipline of Programming, Edsger W. Dijkstra (Prentice Hall, Inc., 1976)

Effective Java, Joshua Bloch (Prentice Hall, 2008)

The Elements of Programming Style, Brian Kernighan and P.J. Plauger (Computing McGraw-Hill, 1978)

Elements of Style, William Strunk and E.B. White (Longman, 1999)

Expert C Programming, Peter van der Linden (Prentice Hall PTR, 1994)

Founders at Work, Jessica Livingston (Apress, 2007)

Hacker's Delight, Hank Warren (Addison-Wesley, 2002)

Higher-Order Perl, Mark Jason Dominus (Morgan Kaufmann, 2005)

Java Concurrency in Practice, Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea (Addison-Wesley, 2006)

Java Puzzlers: Traps, Pitfalls, and Corner Cases, Joshua Bloch and Neil Gafter (Addison-Wesley, 2005)

The Lisp 1.5 Programmer's Manual, John McCarthy (MIT Press, 1962)

Literate Programming, Donald Knuth (Center for the Study of Language and Information, 1992)

Machine Intelligence 1, N.L. Collins and Donald Michie (eds.) (Oliver and Boyd, 1967)

Machine Intelligence 2, Ella Dale and Donald Michie (eds.) (Oliver and Boyd, 1968)

Machine Intelligence 3, Donald Michie (ed.) (Edinburgh University Press, 1968)

Machine Intelligence 4, Bernard Meltzer and Donald Michie (eds.) (Edinburgh University Press, 1969)

Magic House of Numbers, Irving Adler (HarperCollins, 1974)

"META II a Syntax-Oriented Compiler Writing Language," D.V. Schorre in *Proceedings of the 1964 19th ACM national conference,* (ACM, 1964)

Mindstorms: Children, Computers, and Powerful Ideas, Seymour A. Papert (Basic Books, 1993)

The Mythical Man-Month: Essays on Software Engineering, Frederick P. Brooks (Addison-Wesley Professional, 1995)

Principles of Compiler Design, Alfred Aho and Jeffrey Ullman (AddisonWesley, 1977)

"Proof of a Program: FIND", C.A.R. Hoare in *Communications of the ACM,* Vol. 14, Issue 1 (ACM, 1971)

Programming Pearls, Jon Bentley (ACM Press, 1999)

Purely Functional Data Structures, Chris Okasaki (Cambridge UniversityPress, 2008)

A Retargetable C Compiler: Design and Implementation, David Hanson and Christopher Fraser (Addison-Wesley Professional, 1995)

Smalltalk-80: The Interactive Programming Environment, Adele Goldberg (AddisonWesley, 1983)

Smalltalk-80: The Language & Its Implementation, David Robson and Adele Goldberg (Addison-Wesley, 1983)

Structure and Interpretation of Computer Programs, Harold Abelson and Gerald Jay Sussman (MIT Press, 1996)

TeX: The Program, Donald Knuth (Addison-Wesley, 1986)

The Programming Language LISP: Its Operation and Applications, Edmund Berkeley and Daniel Bobrow, eds. (MIT Press, 1966)

The Psychology of Computer Programming: Silver Anniversary Edition, Gerald Weinberg (Dorset House, 1998)

The TeXbook, Donald Knuth (Addison-Wesley Professional, 1986)

Writers at Work: The Paris Review Interviews, Malcolm Cowley (Penguin, 1977)

Zen and the Art of Motorcycle Maintenance: An Inquiry into Values, Robert Pirsig (Bantam, 1984)