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

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。

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

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。

已经搬到加州大学伯克利分校求学，我正好可以随她而去。

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

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。

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]？

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。

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。

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

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

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]

系统不断膨胀而无法实现，连带第一个系统也被荒废。

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。

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。

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

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》。

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？

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）