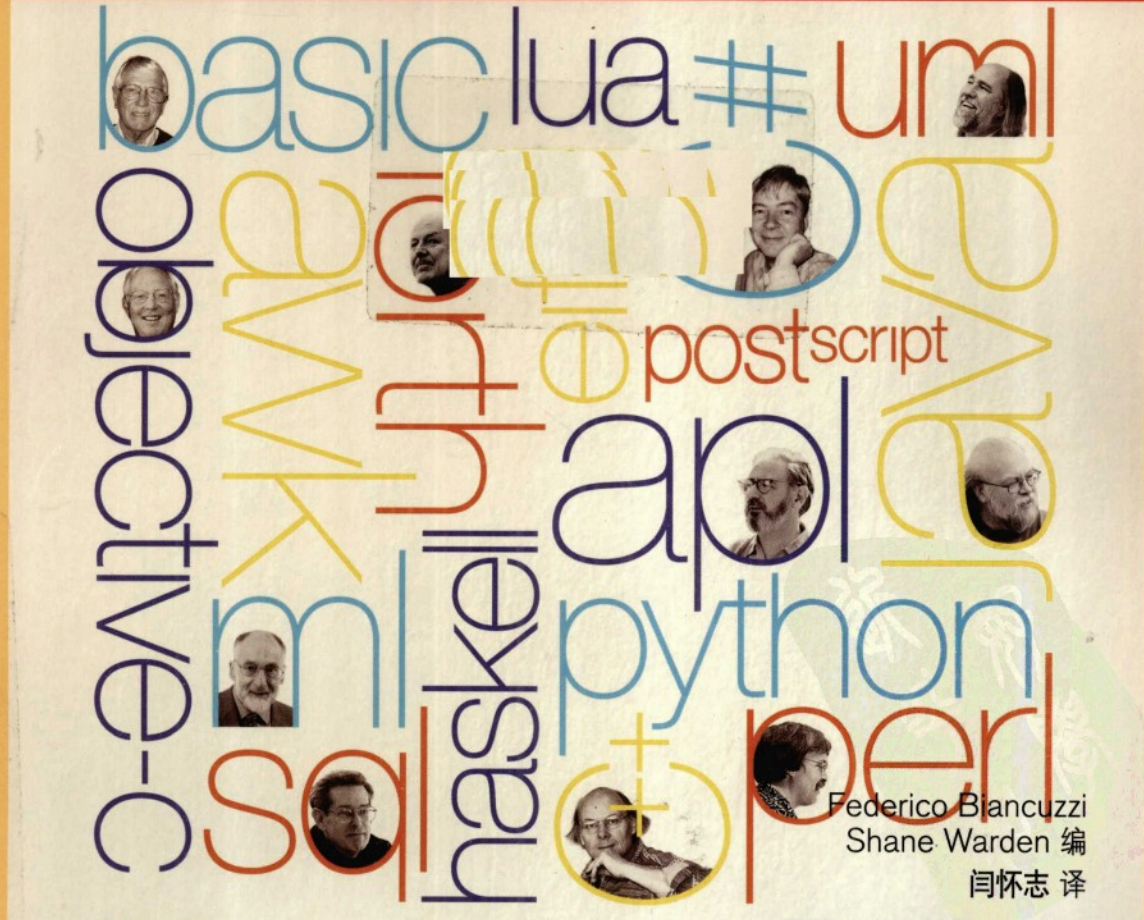
/THEORY/IN/PRACTICE

编程之魂

Masterminds of Programming

与27位编程语言创始人对话

Conversations with the Creators of Major Programming Languages



Federico Biancuzzi
Shane Warden 编
闫怀志 译

O'REILLY®

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

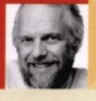
程序设计

编程之魂

《编程之魂》采访了数位极具影响力的编程语言创建者。在这本独一无二的采访集中，您会了解具体设计决策的过程，包括这些前辈必须作出的折中平衡，以及他们的经历对于今天编程的影响。受访嘉宾包括：





- Adin D. Falkoff: **APL**
- Thomas E. Kurtz: **BASIC**
- Charles H. Moore: **FORTH**
- Robin Milner: **ML**
- Donald D. Chamberlin: **SQL**
- Alfred Aho, Peter Weinberger和Brian Kernighan: **AWK**
- Charles Geschke和John Warnock: **PostScript**
- Bjarne Stroustrup: **C++**
- Bertrand Meyer: **Eiffel**
- Brad Cox and Tom Love: **Objective-C**
- Larry Wall: **Perl**
- Simon Peyton Jones, Paul Hudak, Philip Wadler和John Hughes: **Haskell**
- Guido van Rossum: **Python**
- Luiz Henrique de Figueiredo和Roberto Ierusalimschy: **Lua**
- James Gosling: **Java**
- Grady Booch, Ivar Jacobson和James Rumbaugh: **UML**
- Anders Hejlsberg: Delphi的发明者和C#的主要开发者

如果您对那些具有远见卓识并为计算机行业的发展殚精竭虑的人感兴趣，您会发现《编程之魂》有着无穷的魅力。

关于采访者

Federico Biancuzzi是位自由职业采访者（freelance interviewer），他的采访在ONLamp、NewsForge、TheRegister、ArsTechnica等很多网站上在线出版。

Shane Warden是位对编程语言设计和虚拟机感兴趣的自由软件开发者。他在业余时间经营独立出版商Onyx Neon Press的小说分部。他是《The Art of Agile Development》（O'Reilly）的合著者。

图书分类：软件工程

责任编辑：周筠

www.oreilly.com

O'REILLY®

Broadview®
WWW.BROADVIEW.COM.CN

www.phei.com.cn

ISBN 978-7-121-10498-5

9 787121 104985 >

定价：59.80元

O'REILLY®

编程之魂：与27位编程语言创始人对话

Masterminds of Programming

Conversations with the Creators of Major Programming Languages

【美】Federico Biancuzzi 编
Shane Warden
闫怀志 译

電子工業出版社
Publishing House of Electronics Industry
北京 • BEIJING

内容简介

本书将由27位杰出的设计师与你分享他们的智慧和经验。书中以深度访谈的形式告诉你他们为什么要创建某种编程语言，此语言在技术上如何开发，如何教授和学习，以及如何顺应时代发展等。让你了解到语言背后的故事，领悟语言设计的动机，并切实体会到大师们的个性和风采。阅读本书你会发现，构建成功编程语言所需的思想和步骤、它广受欢迎的原因，以及如何处理程序员常见的问题。如果你想深入学习设计成功编程语言的思想，本书会对你大有帮助。

版权贸易合同登记号 图字：01-2009-0866

图书在版编目（CIP）数据

编程之魂：与27位编程语言创始人对话/（美）比安库齐，（美）沃登编；闫怀志译.—北京：电子工业出版社，2010.4
书名原文：Masterminds of Programming:Conversations with the Creators of Major Programming Languages
ISBN 978-7-121-10498-5

Ⅰ.编… Ⅱ.①比…②沃…③闫… Ⅲ.程序语言 Ⅳ.TP312

中国版本图书馆CIP数据核字（2010）第040754号

策划编辑：徐定翔
责任编辑：周 筠
项目管理：梁 晶
封面设计：Monica Kamsvaag，张 健
印 刷：北京天宇星印刷厂
装 订：三河市鹏成印业有限公司
出版发行：电子工业出版社
北京市海淀区万寿路173信箱 邮编：100036
开 本：860×1092 1/16 **印张**：25 **字数**：500千字
印 次：2010年4月第1次印刷
定 价：59.80元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，
联系及邮购电话：（010）88254888。
质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。
服务热线：（010）88258888。

O'Reilly Media, Inc.介绍

为了满足读者对网络和软件技术知识的迫切需求，世界著名计算机图书出版机构 O'Reilly Media, Inc. 授权电子工业出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly Media, Inc. 是世界上在Unix、X、Internet和其他开放系统图书领域具有领导地位的出版公司，同时也是在线出版的先锋。

从最畅销的《The Whole Internet User's Guide & Catalog》(被纽约公共图书馆评为20世纪最重要的50本书之一）到GNN（最早的Internet 门户和商业网站），再到 WebSite（第一个桌面 PC的Web服务器软件），O'Reilly Media, Inc.一直处于Internet发展的最前沿。

许多书店的反馈表明，O'Reilly Media, Inc.是最稳定的计算机图书出版商——每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly Media, Inc. 具有深厚的计算机专业背景，这使得O'Reilly Media, Inc. 形成了一个非常不同于其他出版商的出版方针。O'Reilly Media, Inc. 所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly Media, Inc.还有许多固定的作者群体——他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O'Reilly Media, Inc.依靠他们及时地推出图书。因为 O'Reilly Media, Inc.紧密地与计算机业界联系着，所以O'Reilly Media, Inc. 知道市场上真正需要什么图书。

译序

编程语言是人们为了描述计算过程而设计的一种具有语法语义描述的记号。没有编程语言支持的计算机世界简直难以想象。由于编程语言具有如此重要的基础地位，从计算机问世至今的大半个世纪中，人们一直在为创建更好的编程语言而不懈努力。各种各样的编程语言及其方言层出不穷。而在成千上万种编程语言中，成功者是寥若星辰，其中广泛使用的语言甚至可以说是屈指可数。显然，每一种成功的语言背后都有其独到的设计思想和理念，契合了某一类特定的应用需求。

拿到这本书的英文版时，我又习惯性地看了一眼 TIOBE 的编程语言榜单：本书涉及的各种语言几乎每个月都赫然在列，其中很多语言更是在榜单上名列前茅。说出来你或许会对此感到惊讶，我每次都要看一看榜单上 BASIC 语言的排名。因为整整 20 年前，我第一次接触到 BASIC 语言，当时觉得它非常神奇，后来我用它来编写 DOS 程序，在使用过程中越发对它的设计者崇拜不已，心中常常在想，这是怎样的一位智者啊。后来，我又学习和接触到了更多的语言，几乎每种语言都有着鲜明的特色和独有的风格。每一种语言的背后，都闪耀着编程语言设计大师们智慧的光芒。我想，很多读者、IT 人士都和我一样，都希望有机会当面请教这些世界级的大师，直接聆听这些领域中最权威的声音。

很显然，并不是每个人都有机会跟大师对话，即便是大师已经走到了你的身边（虽然这样的机会微乎其微）。比如，C++之父 Bjarne Stroustrup 2002 年就曾造访中国，并在北京中关村进行了一系列公开活动，而中国的大多数对 C++感兴趣的 IT 人士，即便是身处中关村也很难有机会同他当面交流（非常遗憾，我也身处此列）。要同

17 种语言的创建者进行零距离对话，对绝大多数人来说，几乎没人敢有此奢望，因为这不啻于天方夜谭。

机会终于来了！不过，是以另外一种方式与大师面对面：《编程之魂》一书将 17 种语言的创建者请到了读者身边，让读者从对他们的深度访谈中，了解到语言背后的故事，领悟语言设计的动机，并切实体会到大师们的个性和风采。众所周知，大凡超凡之人，往往都极具个性，甚至会有一些偏执和怪异，毫无疑问，这本身就彰显出无穷的魅力。更重要的是，读了这些访谈原文之后，读者会对相应的语言有着更深入的理解。

在本书的翻译过程中，我经常会产生这样的期盼：在这个不同肤色、不同种族的编程语言创建者群体中，应该有我们黑头发、黄皮肤的中国人！由于历史和现实的原因，中国的程序员能够接触到的编程语言世界级大师少之又少，而中国人能够成为编程语言世界级大师尚需待以时日。我真心地希望中国 IT 界的朋友，特别是年轻的 IT 精英能够勇于承担此任。为了这一天的早日到来，我想，每一个抱有 IT 梦想的朋友，都应该好好研读一下这本书。或许你就是下一个 Bjarne Stroustrup！

本书翻译期间得到了电子工业出版社博文视点公司徐定翔、白爱萍等编辑的热心帮助。同时，我要感谢我的家人对我的支持和默默无闻的奉献。由于译者水平有限，加之时间较紧，虽已尽力避免错误，难免仍有疏漏，恳请广大读者将意见和建议发至：*bityhz001@sina.com*，不胜感激。

闫怀志

2010 年春于北京中关村

序

Foreword

编程语言设计是个魅力无穷的话题。很多程序员认为自己能设计出一种编程语言，可以比现在使用的更好；很多研究者也觉得自己能设计出这样的语言。这些人的自信本来无可厚非，不过他们的设计最终多被束之高阁。你不会在这本书里找到它们。

编程语言设计是件严肃认真的事情。语言设计中的小错误，可能会导致使用该语言编写的实际程序出现大问题。而且，即使是程序中的小错误，也可能会造成极其严重的后果。软件的使用范围很广，而其脆弱性使得恶意软件攻击频繁发生。这已经为全球带来了数十亿美元的经济损失。本书会反复讨论编程语言的防危性（safety）和安全性（security）这个主题。

设计编程语言是一种前途莫测的冒险之举。很多语言是为通用应用程序设计的，并且有众多组织的支持和倡导，却只能在受众很少的细分市场（niche market）内勉强立足。与此相反，设计用于有限用途或局部使用的那些语言，也可以赢得大批拥趸，有时候甚至会用于连设计者都从未想到的环境和应用中。本书集中讨论后一类语言。

这些成功的语言有一个共同的显著特征：它们都来自个人或志同道合的爱好者小团队的创意。它们的设计者是程序设计的宗师（mastermind，大师）级人物；他们具有足够的经验、远见、能力、耐力和绝对的天分，能够通过语言的最初实现、基于实战经验的演变，以及实际使用（事实上）和标准流程（法律上）这两方面的标准化，推动语言不断发展。

在本书中，读者将会和这些宗师们零距离对话。他们每个人都会接受深度访谈，讲述有关语言的故事，以及语言成功背后的原动力。他们坦陈成功取决于正确决策和好运气，二者缺一不可。最后，我们相信出版访谈原文，可以让您深入了解设计者的个性和动机，这和设计语言本身一样具有无穷的魅力。

—Sir Tony Hoare

Sir Tony Hoare，美国计算机学会图灵奖和京都奖（Kyoto Award）获得者，50 年来一直是计算算法和编程语言研究的领导者。他在 1969 年撰写的第一篇学术论文，系统地研究了证明程序正确性的思想，并提出编程语言设计的目标之一是更易于编写正确的程序。他非常高兴地看到该思想已在编程语言设计者中间逐渐传播开来。

前言

Preface

编写软件是件难事——最起码，编出来的软件如果要经得住测试、时间和不同环境的考验，确实很难。在过去的 50 多年里，为了让编写软件变得更容易一些，不仅软件工程领域为此在不懈努力，编程语言也被赋以重任。但是，真正的困难究竟是什么呢？

大多数书籍和论文在回答这个问题时，都将焦点集中在软件体系结构、需求之类的话题上。不过，如果困难在于编写程序本身，又会怎么样呢？换句话说，如果我们把自己当成是更具交流（语言）色彩的程序员，而不是更具工程色彩的程序员，又该如何呢？

小孩子两三岁学说话，五六岁学读书写字。我从来没见过哪个大作家是成年以后才学习读书和写字的。你也没见过晚年才开始学习编程的大牌程序员吧？

如果孩子比成人更容易学习外语，这对我们学习编程有什么启发呢？要知道，编程这事恰恰是和新语言密切相关的！

假如你不知道某件东西在新学的外语中怎么说，你会使用自己熟悉的词汇来描述它，并希望别人也能理解你的

意思。这不正是我们编写软件要做的事吗？我们使用编程语言来描述脑海中的对象，并希望编译器或解释器也可以理解。如果行不通，我们再回头检查脑海中的设计，找出被忽略或有偏差的部分。

带着这些问题，我决定开展一系列调查：为什么要创建某种编程语言、它在技术上如何开发、如何教授和学习，以及它如何顺应时代发展等。

我和 Shane 极为荣幸地邀请到了 27 位杰出的设计师与你分享他们的智慧和经验。

在这本《编程之魂》中，你会发现构建成功编程语言所需的思想和步骤，它广受欢迎的原因，以及如何处理程序员常见的问题。因此，如果你想深入学习设计成功编程语言的思想，本书会对你大有帮助。

如果你想找到有关软件和编程语言激动人心的想法，就需要一支荧光记号笔，甚至两支，因为我保证，你会发现这本书处处闪耀着智慧的光辉。

—Federico Biancuzzi

材料组织结构

在你浏览本书时，各章的排序方式会为提供各种充满争议的不同观点。请仔细品味这些采访内容，并时常回味、反思。

第 1 章，C++，采访 Bjarne Stroustrup。

第 2 章，Python，采访 Guido van Rossum。

第 3 章，APL，采访 Adin D. Falkoff。

第 4 章，Forth，采访 Charles H. Moore。

第 5 章，BASIC，采访 Thomas E. Kurtz。

第 6 章，AWK，采访 Alfred Aho、Peter Weinberger 和 Brian Kernighan。

第 7 章，Lua，采访 Luiz Henrique de Figueiredo 和 Roberto Ierusalimschy。

第 8 章，Haskell，采访 Simon Peyton Jones、Paul Hudak、Philip Wadler 和 John Hughes。

第 9 章，ML，采访 Robin Milner。

第 10 章，SQL，采访 Don Chamberlin。

第 11 章，Objective-C，采访 Tom Love 和 Brad Cox。

第 12 章，Java，采访 James Gosling。

第 13 章，C#，采访 Anders Hejlsberg。

第 14 章，UML，采访 Ivar Jacobson、James Rumbaugh 和 Grady Booch。

第 15 章，Perl，采访 Larry Wall。

第 16 章，PostScript，采访 Charles Geschke 和 John Warnock。

第 17 章，Eiffel，采访 Bertr 和 Meyer。

“受访嘉宾”一节列出了所有受访嘉宾的简历。

本书使用的约定

本书使用如下排版约定：

斜体字（*Italic*）

表示新术语、URL、文件名以及应用程序。

等宽字体（`Constant width`）

表示计算机文件目录和程序中的普通内容。

如何联系我们

我们已尽力核验本书所提供的信息，尽管如此，仍不能保证本书完全没有瑕疵，而网络世界的变化之快，也使得本书永不过时的保证成为不可能。如果读者发现本书内容上的错误，不管是赘字、错字、语意不清，甚至是技术错误，我们都竭诚虚心接受读者指教。如果您有任何问题，请按照以下的联系方式与我们联系。

奥莱理软件（北京）有限公司
北京市 西城区 西直门 南大街2号 成铭大厦C座807室
邮政编码：100080
网页：*http://www.oreilly.com.cn*
E-mail：*info@mail.oreilly.com.cn*

O'Reilly & Associates, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (in the United States or Canada)
(707) 829-0515 (international/local)
(707) 829-0104 (fax)

与本书有关的在线信息如下所示。

http://www.oreilly.com/catalog/9780596515171（原书）
http://www.oreilly.com.cn/book.php?bn=978-7-121-10498-5（中文版）

北京博文视点资讯有限公司（武汉分部）

湖北省 武汉市 洪山区 吴家湾 邮科院路特1号 湖北信息产业科技大厦1402室
邮政编码：430074
电话： (027)87690813　　传真：(027)87690813转817
读者服务网页：*http://bv.csdn.net*
E-mail：
reader@broadview.com.cn（读者信箱）
bvtougao@gmail.com（投稿信箱）

目录

Contents

第 1 章

C++

在诸多语言之中，C++的地位非常有趣：它构建在 C 语言的基础之上，融入了 Simula 面向对象的思想；由 ISO 实现了标准化；而且，它还遵循“你不会为没用的东西白花冤枉钱”和“支持用户自定义和内置类型同等重要”的设计理念。尽管在 20 世纪 80 年代和 90 年代，C++曾广泛用于 OO（面向对象）和 GUI（图形用户接口）编程，不过，它对软件的最大贡献还是其无处不在的泛型编程技术，它的标准模板库（Standard Template Library）就是例证。Java 和 C#等一些更新的语言都试图取代 C++，不过 C++即将发布一个修订版标准，其中就增加了期待已久的新特性。Bjarne Stroustrup 是 C++语言的创始人，而且他至今仍是 C++最坚定的倡导者之一。

1.1 设计决策

为什么您会选择对现有语言进行扩展，而不是创建一种新语言呢？

Bjarne Stroustrup：我在 1979 年刚开始设计语言时，目的就是帮助程序员构建系统。现在还是这样。一种语言如果想要真正有助于解决问题，而不仅仅用于学术，它就必须完全为应用程序领域服务。也就是说，一种非研究性的语言，它的使命就是要解决问题。我这里所说的问题，是与操作系统设计、组网以及仿真有关的那些问题。我和同事需要一种语言，它不仅能像 Simula 语言那样表示程序结构（人们大多愿意称它为面向对象编程），还可以像 C 语言那样编写出高效的底层代码。在 1979 年，还没有什么语言能够兼具这两种功能，否则我就会直接拿来用了。我并没有特别想去设计一种新的编程语言，我只不过想要帮助解决几个问题而已。

所以，在现有语言的基础上构建新语言就很有意义。你可以使用基础语言的基本句法、语义结构，还可以使用它的库，而且你自己也会融入这种文化。如果当时不是以 C 为基础，我也会在其他语言的基础上构建 C++。那我为什么选择了 C 呢？我在 Bell 实验室计算机科学研究中心（Computer Science Research Center）与 Dennis Ritchie、Brian Kernighan 和其他 Unix 大师比邻办公，因此，这个问题好像有些多余。不过，我在这个问题上是非常认真的。

特别是，C 的类型系统是非形式化和弱强制型的（正如 Dennis Ritchie 所言，“C 是一种强类型、弱检查的语言”）。“弱检查”曾经令我头疼不已，直到现在它仍为 C++程序员制造了很多麻烦。况且，目前 C 语言的应用并不广泛。在 C 的基础上构建 C++，这一方面表明了我对以 C 为基础的计算模型的信心（“强类型”部分）（译注1），另一方面也表明了我对同事们的信任。当时，系统编程使用的大多是更高级的编程语言，我是基于这些语言的知识（既是用户，又是实现者），做出了这样的选择。值得一提的是，当时大多数是“接近硬件”工作，而且汇编程序还对性能有着严格的要求。Unix 在很多方面都有重大突破，其中包括用 C 完成了最苛求系统的编程任务。

因此，我才选择了用 C 作为机器的基本模型，而不是选择“强检查”类型的系统。我最想用 Simula 的类作为程序框架，因此，我把它们映射成 C 的内存和计算模型。结果，它不仅极具表现力和灵活性，而且运行速度很快，甚至可以和没有大规模运行时系统支持的汇编程序相媲美。

为什么您会选择支持多种模式？

Bjarne：因为各种编程风格的组合通常会生成最好的代码，这里“最好”的意思是该代码能够最直接地表达设计思想、运行速度更快、可维护性最强等。当人们质疑这种说法时，通常他们要么是定义自己喜欢的编程风格，将每一个有用的结构（例如，“泛型编程只不过是 OO 的一种方式而已”）包括进去，要么就是限制应用范围（例如，“每个人都要使用 1GHz、1GB 的机器”）。

译注1：强类型（Strongly Typed），意思是说，对于程序里的每一个变量，一是在使用前必须声明类型；二是赋值的时候也必须与变量的类型相同。

Java 只关注面向对象编程。这是否会让 Java 代码在某些情况下变得更为复杂，从而 C++可以利用泛型编程取而代之吗？

Bjarne：噢，Java 设计者（很可能 Java 营销人员更是如此）把 OO 吹捧到荒谬可笑的地步。在号称纯正而且简单的 Java 刚出现时，我就曾经预言：如果 Java 能成功的话，它就会在规模和复杂性方面显著增长。结果不出所料，确实如此。

例如，在容器外获得一个值时（例如，`(Apple)c.get(i)`），会使用“造型”来转换 `Object`，之所以使用这种非常荒谬的方法，是因为无法假定该容器的对象为何种类型。它不仅繁琐冗长，而且效率低下。现在，Java 使用的是泛型，因此它只不过稍微有点慢而已。语言复杂性的增加（帮助程序员）还有其他例证，比如说枚举、反映和内部类等。

复杂性是不可避免的，这是一个简单的事实：如果它没有出现在语言定义中，就会出现在数以千计的应用程序和库中。同样，Java 会将每一个算法（运算）都放入类中，这种困扰会导致出现这样的荒唐事：仅仅是由静态函数组成而没有数据的类。这就是在数学里使用 `f(x)` 和 `f(x, y)`，而不是 `x.f()`、`x.f(y)` 或 `(x, y)f()` 的原因所在：后者试图表达两个参数“真正的面向对象方法”的理念，并试图避免 `x.f(y)` 固有的不对称性。

C++通过将数据抽象和泛型编程技术相结合，使用面向对象方法，解决了很多逻辑和符号的问题。向量<T>就是一个典型的例子，其中 T 是可以复制的任何类型：包括内置类型、OO 层次的指针，以及用户自定义的类型，比如字符串和复数等。完成这项工作，既没有增加运行时开销，也没有对数据规划附加限制，更没有对标准库组件使用特殊规则。还有一个例子是需要访问两个类的运算，它也不符合传统的单分派的 OO 层次模型，比如操作符`*`（`Matrix`（矩阵）、`Vector`（向量））等，该运算实质上不是这两个类的“方法”。

C++和 Java 的一个根本区别是指针实现的方式。在某种意义上，您可以说 Java 并没有真正的指针。这两种方式之间有什么区别呢？

Bjarne：噢，Java 当然有指针。事实上，Java 中的每个操作几乎都暗含一个指针。它们只是把这些“指针”称为引用而已。这种隐式指针有利有弊。使用真正的本地对象（像在 C++中一样）也各有优缺点。

C++选择支持堆栈分配（stack-allocated）的局部变量和各种类型的真正的成员变量，这样可以具有好的统一语义（uniform semantics），同时也支持值语义（value semantics）的概念，可以实现紧凑布局和最小的访问成本，而且它还是 C++支持一般资源管理（general resource management）的基础。这是主要的，而且 Java 到处使用隐式指针（aka 引用）关闭了通向这一切的所有大门。

考虑布局的折衷平衡：在 C++中，向量<complex>(10)表示为自由存储区中一个 10 个复数数组的句柄。它总共有 25 个词：3 个词用于向量，20 个词用于复数，再加上一个用于自由存储区（堆内存）数组的两个词头。在 Java 中，则应该是 56 个词（用于一个用户自定义类型对象的用户自定义的容器）：1 个词用于容器引用，3 个词用于容器，10 个词用于对象引用，20 个词用于对象，还有 24 个词用于 12 个独立分配对象的自由存储头。显然，这些数字是近似值，因为自由存储区（堆内存）的开销就是在两种语言中定义的实现。不过，结论是非常清楚的：通过无处不在的隐式引用，Java 可能已经简化了编程模型和垃圾收集器的实现，不过它大大地增加了内存开销，并成比例地增加了内存访问成本（需要更多的间接访问）和分配费用开销。

C 和 C++可能会通过指针运算误用指针，而 Java 并不具有这种能力，这对 Java 来说也是好事。不过如果 C++程序写得很好，也不会遇到这样的问题：人们使用更高级别的抽象，比如 iostream（译注2）、容器和算法，而不是乱用指针。本质上，所有的数组和大多数指针应深藏于大多数程序员并无必要看到的实现之中。不幸的是，大量写得很糟的和多余的底层 C++程序随处可见。

不过，有时候使用指针（和指针操作）也很方便，这有一个重要的例子：直接、高效的数据结构表达式。它没有 Java 引用：例如，你无法用 Java 表示交换运算。另一个例子是简单地使用指针来直接访问底层（实际）内存：对于每一个系统来说，一些语言不得不这样做，而且这种语言通常就是 C++。

拥有指针（和 C 风格的数组）的“负面影响”当然是潜在滥用的可能：缓冲区溢出、指向已删除内存的指针、未初始化的指针等。不过，在写得很好的 C++程序中，这不是主要问题。你只不过是没有碰到在抽象之内使用的指针和数组（比如向量、字符串和映射等）的那些问题。限定作用域的资源管理满足了大多数的需要；大多数遗留问题都可以使用智能指针和专门的句柄来处理。以前主要使用 C 或旧版 C++的人对此很难相信，不过基于限定作用域的资源管理这个工具功能非常强大，而且它还是用户自定义的，自身带有的合适的运算，可以使用比不安全的老式 hack 程序（译注3）更少的代码解决经典问题。例如，以下是典型的缓冲区溢出和安全性问题的最简单的形式：

```
Char buf[MAX_BUF];
gets (buf); //讨厌!
```

使用一个标准库字符串，问题会迎刃而解：

```
string s;
cin>> s; //读取使用空白字符分开的字符
```

这些显然是不起眼的小例子，不过使用合适的“字符串”和“容器”可能会基本满足所有需求，而且标准库还可以让你轻松上手。

5

您前面提到“值语义”和“一般资源管理”，这是什么意思？

Bjarne：“值语义”通常是指这样的类：其中的对象属性在复制时，你会得到两个独立的副本（具有相同的值）。

例如：

```
X x1 = a;
X x2 = x1;  //现在 x1==x2
x1 = b;     //修改 x1，而不是修改 x2
            //现在 x1!=x2 (假设 X (a) !=X (b))
```

当然，对于常见的数值类型，我们就是这样处理的，比如说 ints、doubles 和复数等数值类型，以及向量等数学抽象概念。这个定义的用处最大，在这里 C++支持内置类型和我们想要的任何用户自定义的类型。它和 Java 区别很大，Java 会支持 char 和 int 等内置类型，而不支持用户自定义的类型，而实际上也确实无法支持那些类型。正如在 Simula 中一样，Java 中用户自定义的所有类型都有引用语义。在 C++中，如果一种类型的语义

译注2：C++的标准输入输出库 iostream 是一个类库，又称为 I/O 流类库，它以类的形式组织，使用该库中的类要先引用命名空间。所谓流，就是数据或信息的流动。

译注3：hack 程序，是指一种辅助程序，完成对系统功能的扩展，即替换或增强原有的系统功能。这种偷梁换柱修改系统的工作方式，就像网络上的黑客侵入并接管别人的网络或者计算机一样，hack 程序由此得名。

需要，程序员也可以支持引用语义。在支持带有值语义的用户自定义类型方面，C#（不完全）遵从 C++。

“一般资源管理”是指让一个对象拥有一种资源（例如，文件句柄或者锁）的流行技术。如果该对象是一个限定作用域的变量，它的生命周期就是对持有资源时间的一个最大限制。典型的例子是，构造器获取资源和析构器释放资源。这通常被称为 RAII（Resource Acquisition Is Initialization，资源获取初始化），而且它会和使用异常的错误处理完美地集成在一起。显然，并不是每一种资源都可以用这种方式来处理，不过仍然有很多资源可以使用这种方式，而且对于那些资源来说，资源管理变得既隐蔽，又高效。

“接近硬件”看起来好像是 C++设计的一个指导原则。如果要与很多语言相比，自顶向下设计的 C++更像是“自底向上”的，之所以这么说，只是因为它们试图提供合理抽象的结构，并强制编译器让这些结构去适应可用的计算环境，这样说公平吗？

Bjarne：我认为使用自顶向下和自底向上来描述那些设计决策，这是一种错误的方式。在 C++和其他语言的语境中，“接近硬件”意味着计算模型是计算机的模型（内存中的对象顺序和定义在固定尺寸对象上的运算），而不是一些数学抽象概念。这对于 C++和 Java 来说是对的，而对于函数式语言来说则是不对的。C++和 Java 不同，C++的底层机器是真正的机器，而不是抽象的机器。

真正的问题在于，如何把人类对问题和解决方案的概念转换到机器的有限世界中去。你可以“不考虑”人类的关注，并最终以机器代码（或者是吹嘘成机器代码的糟糕的 C 代码）来实现它。你可以不考虑机器，并以惊人的成本和（或）缺乏理性的严密，提出一个无所不能的很好的抽象概念。C++试图让你在需要时直接访问硬件（例如，指针和数组等），同时还提供了扩展的抽象机制，以此来表示高级概念（例如，类层次和模板等）。

据说整个 C++及 C++库开发期间，一直持续关注运行时和空间性能。这种理念会以某种方式渗透到基本语言工具和抽象工具当中，而这种方式并不是所有语言都有的。

1.2 使用语言

您如何调试？您对 C++开发者有何建议？

Bjarne：通过自我领悟来调试。我花了很长时间来学习程序，并且多多少少系统地琢磨了这么长时间，因此，要为查找 bug 提供一个言之有据的推测，我对此有着充分的了解。

测试是另外一回事，将错误减到最少程度的设计也是另外一回事。我非常讨厌调试，并会主动采取措施来避免调试。如果我是一款软件的设计者，我会围绕接口和不变量来构建这款软件，因此就不会编写出极为糟糕的代码来错误地编译和运行。然后，我尽量会让它具有很好的可测试性。测试就是系统地搜索错误。结构很差的系统难以进行系统测试，因此，我再次建议要让代码结构保持清晰。测试可以实现自动化，并可以使用调试无法使用的方式来重复测试。让鸽群在屏幕上随意啄食，借以考察一个基于 GUI 的应用程序是否会崩溃，这并非是确保系统质量的方法。

你问我有何建议？要给出一般性的建议很难，因为最好的技术通常取决于对于特定开发环境中的特定系统来

说可行与否。不过，你可以找出能够进行系统测试的关键接口，并编写相应的测试脚本。最大程度地实现自动化，并经常运行那些自动测试。同时，要经常进行回归测试并运行它们。确认可以系统地测试系统的每一个输入和每一个输出。要使用高质量的组件来构建你的系统：不必使用难以理解和测试的单个大程序。

必须在什么级别上来改进软件的安全性呢？

Bjarne：首先，安全性是一个系统问题。局部的或者部分的补救办法很难独立奏效。请记住，即使你的所有代码都是完美的，如果我能偷到你的计算机或者保存备份信息的存储设备，仍然很有可能来访问你所存储的秘密信息。其次，安全性是一种成本/效益博弈：完美无缺的安全性，很可能超出了我们大多数人的承受范围，不过我很可能会这样有效地保护自己的系统："坏蛋们"会认为把他们的时间花在入侵他人的系统上会更好一些。实际上，我不喜欢在线保存重要秘密，并给专家们留下严重的安全问题。

不过，编程语言和编程技术会怎么样呢？现在有一种非常危险的趋势，那就是假定每一行代码都必须是"安全可靠的"（无论它的含义是什么），甚至假定有恶意企图的人会搞乱系统的其他某些部分。这点最为危险：为了预防混乱的假想威胁而进行的非系统化测试，将会把代码弄得乱七八糟。代码也因此变得难看、庞大和速度缓慢。"难看"为隐藏 bug 埋下了伏笔，"庞大"会导致不完全测试，而"速度缓慢"则会怂恿使用快捷键和不诚实手段，这些都位列最成熟的安全漏洞源之列。

我认为，安全性问题唯一的永久性解决方案是采用一个简单的安全性模型，这个模型由高质量的硬件和（或）软件在选定接口上系统化地应用。必须有一种安全屏障，可以简单、优雅而且高效地编写代码，而不用担心随机的代码段滥用其他代码的随机代码段。只有这样，我们才能把精力集中于正确性、质量和重要的性能上。像"任何人都能开发出不可信的回调、插件程序及覆盖程序等"这样的观点，显然是非常愚蠢的。我们必须把抵抗欺骗行为和防止意外事故这两种代码区分开。

我认为你不可能设计出这样一种编程语言：它不仅完全的安全可靠，而且又非常适用于实际系统。显然，这要取决于"安全可靠"和"系统"的含义。你有可能会实现某个特定领域的语言的安全性，不过，我主要关注的是系统编程领域（以一个非常宽泛的概念来看），其中包括嵌入式系统编程。我的确认为，类型安全可以、并且会在 C++提供的基础上得以改进，不过问题恰恰在于：类型安全并不等于安全性。编写 C++的人们，如果大量使用未封装的数组、造型和非结构化的新建和删除运算，简直就是自找麻烦。他们还坚持使用 20 世纪 80 年代的编程风格。为了更好地使用 C++，你必须采用这种风格：它会以简单、系统化的方式，将类型安全违例和管理资源（包括内存）减少到最低限度。

您会向在某些系统（比如说系统软件和嵌入式应用程序等）中不愿使用 C++的专业人员推荐 C++吗？

Bjarne：那当然，我会推荐 C++，况且并不非是每个人都不愿意使用 C++。事实上，除了某些历史悠久的组织本能地不愿意尝试新技术以外，我看并没有多少公司不愿意使用 C++。相反，我看到 C++的用户在稳定地显著增长。例如，我帮助美国洛克希德马丁公司（Lockheed Martin）的联合攻击战斗机计划（Joint Strike Fighter）编写任务关键软件编码指导方针。这是一个"完全使用 C++的飞机"。你可能对军用飞机并没有什么特别的兴趣，其实，C++的使用方式和军事应用并没有什么特殊的关联，至少我可以告诉你，在不到一年的时间之内，我的主页中 JSF++编码规则下载量已经超过 100 000 多次，而这些大多是非军用嵌入式系统开发者下载的。

自 1984 以来，C++就已经用到了嵌入式系统之中，很多有用的小工具都是用 C++编写的，而且它的使用仍然呈现快速增长的趋势。使用 Symbian（译注4）或 Motorola 的移动电话、iPod 和 GPS 系统就是例证。我特别喜欢在火星探测器上使用 C++：例如，视景分析和自主驾驶子系统、大部分地基通信系统及图像处理系统等。

信奉“C 语言必然要比 C++效率更高”的人们，应该读一读我的论文：“Learning Standard C++ as a New Language（把标准 C++作为一种新语言来学习）”[C/C++ Users Journal（C/C++用户杂志），1999 年 5 月]，这篇论文简要论述了一些设计哲学并给出了几种简单的实现结果。同时，ISO C++标准委员会也就性能问题发布了一个技术报告，其中阐述了许多关乎性能的、与使用 C++相关的问题和传说（你可以在线搜索“Technical Report on C++ Performance（C++性能技术报告）”找到该报告）（注 1）。特别是，该报告还阐述了嵌入式系统的相关问题。

有些内核，比如说 Linux 或者 BSD 的内核，仍然是用 C 写的。它们为什么还没有改用 C++呢？它在 OO 模式中很重要吗？

Bjarne：这主要是保守主义和惯性在作怪。另外，GCC 也在慢慢成熟。在 C 社区中，有些人对 C 有过十几年的经验，他们的维护行为看起来好象一个近乎固执的无知之举。几十年来，其他操作系统和大量的系统编程，甚至是难以解决的实时和安全关键的代码，也都已经采用 C++来编写了。看看这些实例吧：Symbian、IBM 的 OS/400 和 K42、BeOS，以及 Windows 的部分代码。通常来说，还会有很多用 C++写的开放源代码（例如 KDE）。

你好像把 C++的使用与 OO 混为一谈了。C++现在不是，也从来不只是一种面向对象编程语言。1995 年，我撰写了一篇论文，题目是“Why C++ is not just an Object-Oriented Programming Language（为什么 C++不只是一种面向对象编程语言）”；你可以在线找到这篇论文（注 2）。该观点自始至终支持多种编程风格（如果你愿意使用长词，那就是“模式”）及其组合。在高性能和接近硬件使用的环境中，最相关的其他模式是泛型编程（generic programming，有时简写为 GP）。ISO C++标准库，因其算法和容器（STL）框架，自身要比 OO 更具有 GP 风格。在同时要求抽象和性能的场合，严重依赖模板的典型 C++风格的泛型编程得到了广泛应用。

我从来没见过用 C 写的程序会比用 C++写的更好。我认为不存在这样的程序。如果没有更好的方法，你可以用接近于 C 的风格来编写 C++程序。没有什么会让你过于迷恋异常、类层次或模板。优秀的程序员会使用更先进的特性，它们有助于更直接地表达观点，而且也没有可避免的开销。

为什么程序员应该将代码从 C 移到 C++上呢？把 C++用做一种泛型编程语言有什么优点呢？

Bjarne：你好像假定首先选用 C 语言来编写代码，而且程序员也是从 C 开始入门的。对于很多（甚至是大多数）C++程序和 C++程序员来说，长期以来的事实证明根本不是这么回事。不幸的是，在许多课程中仍然遗

译注4：Symbian OS（中文译音“塞班系统”）由诺基亚、索尼爱立信、摩托罗拉、西门子等几家大型移动通讯设备商共同出资组建的一个合资公司，专门研发手机操作系统。iPod 是 APPLE 推出的一种大容量 MP3 播放器，它有完善的管理程序和创新的操作方式，外观也独具创意，是 APPLE 少数能横跨 PC 和 Mac 平台的硬件产品之一。除了 MP3 播放，iPod 还可以作为高速移动硬盘使用，可以显示联系人、日历和任务，以及阅读纯文本电子书和聆听有声电子书。

注 1：*http://www.open-std.org/JTC1/sc22/wg21/docs/TR18015.pdf*。

注 2：*http://www.research.att.com/~bs/oopsla.pdf*。

存有“C 语言优先”的建议，不过它不再是想当然的事情了。

可能有人会从 C 转向 C++，因为他们发现，C++对于 C 语言编程风格的支持通常要比 C 做得更好。C++的类型检查更为严格（你不会忘记声明一个函数或者它的参数类型），而且对于很多普通运算来说，还有类型安全的符号支持，比如对象创建（包括初始化）和常量等。我已经看到人们这样做，并且对它们的遗留问题也很满意。通常，这要与 C++库联合协力完成，这些库可能需要，也可能不需要被认为是面向对象的，比如说标准向量、GUI 库，或者是一些应用程序特定的库等。

如果只是使用一个简单的用户自定义类型（比如说向量、字符串或者复数等），并不需要模式转换。人们可以像内置类型那样使用（如果他们如此选择的话）它们。有人会使用 `std::vector` “采用 OO”吗？我会说没有。有人会使用 C++ GUI 而没有实际添加新功能的“采用 OO”吗？我倾向于说没错，因为要使用它们，通常会要求用户理解并使用继承。

将 C++用作“一种泛型编程语言”，可以为您提供开包即用的标准容器和算法（作为标准库的一部分）。在很多应用程序中，这是一种主流趋势，也是从 C 中抽象出来的一个主要步骤。除此之外，人们还开始受益于各种 C++库，比如说 Boost 等，而且开始体会到泛型编程中函数式编程技术固有的一些好处。

不过，我认为这个问题稍微有一点误导。我并不想把 C++描绘成“一种 OO 语言”或“一种 GP 语言”；它更是一种支持下列特性的语言：

- C 风格编程
- 数据抽象
- 面向对象编程
- 泛型编程

至关重要的是，它支持各种编程风格的组合（如果你必须使用“多模式编程”的话），而且是以偏好系统编程的方式来完成的。

1.3 OOP 和并发

OOP and Concurrency

软件的平均复杂度和规模（以代码行数计）看起来好像在逐年递增。OOP 能够很好地适应这种情况吗？还是仅仅使事情变得更为复杂？实现对象可重用会使事情变得更为复杂，我对此深有体会，最后，它使工作量成倍增加。首先，你必须设计一个可重用工具。随后，在需要修改时，你必须编写一些东西，它们确实能够弥合旧系统留下的缺陷，这也意味着对解决方案施加了限制。

Bjarne：这是一个重要问题，你问得很好。OO 是一个功能强大的技术集合，它很有用处，不过要想让它真正发挥作用，那就必须把它用好，而且是用它来解决相应技术擅长的问题。对于所有带有静态检查接口、依赖于继承的代码来说，更重要的问题是：要设计出优秀的基类（一个接口对应于多个类，也可能是未知的类），我们需要具有许多远见和丰富的经验。基类（无论你把它叫做抽象类，还是接口）的设计者怎么能知道它为未来来自于它的所有类指定了所需的所有东西？设计者怎么能知道未来来自于它的所有类能够合理地实现指定的内容？基类的设计者怎么能知道指定的内容不会严重地妨碍在未来来自于它的一些类所需要的东西？

一般来说，我们无法知道这些。在可以强制推行我们的设计的环境中，人们会顺应它，通常是采用编写讨厌的权宜之计的办法。如果一个没有组织来负责此事，就会出现很多不兼容的接口，实现实质上相同的功能。

而且，没有什么办法能够解决这些问题。不过，在 OO 方式无能为力的许多重要场合，泛型编程看起来好像能成为一种解决方案。简单容器就是一个值得注意的例子：我们既无法通过继承层次来很好地表示元素的概念，也无法通过它来很好地表示容器的概念。不过，我们能够使用泛型编程来提供有效的解决方案。STL（标准模板库，包含于 C++标准库中）就是一个例子。

这个问题只是针对于 C++，还是其他编程语言也受其困扰？

Bjarne：依靠静态检查接口来连接类层次的所有语言，都普遍存在这个问题。具体例子有 C++、Java 和 C#，动态类型的语言（比如说 Smalltalk 和 Python 等）则不在此列。C++通过泛型编程解决了这个问题，其中，C++标准库中的容器和算法就是很好的例证。这里的关键语言特性是模板，它提供一个新的类型检查模型，可以保证等效于动态类型语言的运行时时间的编译时间。最近，Java 和 C#也添加了“泛型”，这是仿效 C++的做法，而且常常（我认为这是错误的）宣称是在模板的基础上进行的改进。

在代码比初始接口设计的寿命更长时，“因式分解”就会大受欢迎，因为它试图以简单改造代码的暴力技术来解决上述问题。

如果这是 OO 的一个普遍问题，那我们又如何确信 OO 利大于弊呢？或许，难以实现优秀的 OO 设计正是其他所有问题的根源所在。

Bjarne：在某些情况下甚至是很多情况下，OO 会出现一些问题，这是一个事实，不过它并没有改变已经用这种语言编写出很多完美、高效而且可维护的系统的事实。面向对象设计是系统设计的基本方式之一，同时，静态检查的接口既有优点，也有问题。

软件开发根本没有什么“万恶之源”。在很多方面，设计都存在困难。人们大多会低估构建一个包含软件的大系统的理论和实践困难。它不是，也不会简化到一个简单的机械式“汇编行”程序。创建一个令人满意的大系统，需要创造力、工程原理及渐进性变革。

OO 模式和并发之间是否存在联系？目前对于提升并发能力的普遍需要，是否改变了设计的实现或者 OO 设计的本质？

Bjarne：面向对象编程和并发二者的联系由来已久。Simula 67 是第一种直接支持面向对象编程的编程语言，它还提供了表示并发活动的一种机制。

第一个 C++库就支持我们今天所谓的线程。1988 年，我们在 Bell 实验室的一台六处理器机器上运行 C++，而且，并不是只有我们在这样使用。在 20 世纪 90 年代，至少有几十种实验性 C++分支语言和 C++库着手解决与分布式和并行编程有关的问题。目前，我对多核技术非常兴奋，这并不是我第一次接触并发问题。事实上，我的博士课题就是分布式计算，而且从那时候起，我就一直在该领域工作。

不过，第一次接触并发、多核等问题的人们，由于简单地理解了不同处理器上运行一个活动的成本，通常会把自己搞得稀里糊涂。在另一个处理器（核）上启动一个活动，以及该活动访问“调用处理器”内存中的数

据（“远程”复制或者访问）的成本，可以是函数调用成本的 1 000 次以上（或者更高）。而且，只要你引入了并发，出现错误的可能性就会截然不同。为了充分利用硬件提供的并发性，我们需要重新考虑我们的软件结构。

幸运的是，除了困惑之外，我们还拥有数十年的研究经验，可以助我们一臂之力。从根本上来说，即便是这么丰富的研究经验，也几乎不可能去确定什么是真的，更不用说什么是最佳的。如果要追根溯源，还要从有关 Emerald 的 HOPL-III 论文开始。Emerald 是第一个从考虑成本的角度来研究语言问题和系统问题之间相互影响的语言。区分使用了数十年的用于科学计算的数据并行编程（主要是使用 FORTRAN 编程）和很多处理器上使用的“普通顺序代码”（例如，进程和线程等）通信单元，也是很重要的。我认为，一个编程系统，如果想要在这个拥有很多“核”和集群的伟大的新领域中里被广泛接受，就必须同时支持这两种并发方式，而且每种方式都很可能有若干个变体。实现这个目标一点也不容易，而且这个问题完全超出了传统编程语言问题的领域——我们最终会对语言、系统及应用程序问题做通盘考虑。

C++为并发做好准备了吗？很显然，我们能够创建 C++库来处理一切，不过需要时刻考虑严格检查语言和标准库的并发性吗？

Bjarne：差不多如此。C++0x 就是这样。为了为并发做好准备，一种语言首先必须有一个精确指定的内存模型，允许编译器作者使用先进的硬件（使用深管道、大容量高速缓存、分支预测缓冲区，以及静态和动态指令重新排序等）。然后，我们需要一些简单的语言扩展：线程本地存储和原子数据类型。然后，我们可以以库的形式添加对并发的支持。自然地，第一个新标准库会是一个允许跨系统（比如 Linux 和 Windows）移植编程的线程库。当然，很多年前我们就已经有了这样的库，但那不是标准库。

以“线程加上某些形式的锁定”来避免数据竞争，这几乎是直接利用并发最为糟糕的方式，不过，C++要用这种方式来支持现有的应用程序，并以此来维护它作为传统操作系统的系统编程语言的地位。由于这个库已经实际使用了很多年，所以它的原型会继续存在。

并发的一个关键问题是，你如何“包装好”一项任务，让它与其他任务并行执行。我想，在 C++中，答案应该是“作为一个函数对象”。该对象可以包含所需要的所有数据和根据需要传送的数据。C++98 对于指定的运算（我们实例化函数对象的具名类）处理得很好，而且该技术在泛型库（例如 STL）的参数化中无处不在。C++0x 提供了“lambda 函数”，这样更易于编写简单的“一次性”函数对象，而 lambda 函数则可以用表达式上下文（例如，作为函数参数）来编写，并正确地生成函数对象（“闭包”）（译注5）。

接下来的步骤更有趣。在 C++0x 发布后不久，该委员会随即计划制定一个有关库的技术报告。这几乎肯定会规定线程池和工作窃取（work stealing）的一些形式。也就是说，它会制订出一个标准机制，以便用户申请并行处理相对较小的工作单元（“任务”），而不必乱用创建、取消、锁定线程等，这很可能是使用函数对象作为任务来构建的。同样，在地理意义上的远程进程之间，也会使用套接字、iostream 库等通信方法，这与 `boost::networking` 有些类似。

译注5：闭包（closure）是由函数及其相关的引用环境组合而成的实体（即：闭包=函数+引用环境）。

依我看来，并发的诱人之处大多表现在支持逻辑独立并发模型的多库上面。

很多现代系统是组件化的，并且可以通过网络传播：Web 应用程序和内容聚合时代的到来可能会加速这种趋势。一种语言应该反映网络的那些方面吗？

Bjarne：并发有多种形式。一些形式的目的在于改进程序在单台计算机或者集群上的吞吐量或响应时间，一些形式的目的在于处理地理意义上的分布问题，还有一些形式要低于程序员通常考虑的层次（比如，流水线技术和缓存技术等）。

C++0x 会提供一组工具，并保证在机器架构师和编译器作者之间提供一种“契约”——“机器模型”，将程序员从最底层的细节中解放出来。它还会提供一种线程库，实现从代码到处理器的基本映射。在此基础上，该库还可以提供其他模型。我愿意看到一些 C++ 0x 标准库支持的更简单易用、层次更高的并发模型，不过，现在还不太可能出现。以后（希望是在 C++0x 之后很快），我们会拥有技术报告中规定的更多的库：线程池，未来还会有用于广域网（例如 TCP/IP）的 I/O 流库。这些库会存在，不过，并不是每个人都认为它们已足以指定为标准。

几年前，我希望 C++0x 能规定一种标准形式的编排（或序列化），以此来解决 C++长期存在的一些分布问题，不过我的希望并没有实现。因此，C++社区必须通过非标准库和（或）框架（例如，CORBA 或.NET），持续关注更高层次的分布式计算和分布式应用程序的构建问题。

最早的 C++库（真正最早使用类的 C 库）提供了一种轻量级形式的并发，而且几年之后，C++已经构建了数百种用于并发、并行和分布式计算的库和框架，不过该社区尚未就标准问题达成一致。我想，问题的部分原因在于，在这个圈子里做事花费不扉，而且大玩家更喜欢把钱花在他们自己的私有库、框架和语言上。这对于 C++社区总体来说还没有什么好处。

1.4 关于未来

我们最终会看到 C++ 2.0 吗？

Bjarne：这取决于你对“C++ 2.0”含义的理解。如果你的意思是或多或少地从头开始构建一种新语言，并汲取 C++的所有精华，去除 C++的所有糟粕（根据某些“精华”和“糟粕”的定义），我的答案是“我不知道”。我很愿意看到在 C++的传统上发展出新的主流语言，不过我并没有看到短期内会有任何新语言出现的迹象，因此我会把精力集中在昵称为 C++0x 的下一个 ISO C++标准上面。

在很多方面，它会成为一个“C++ 2.0”，因为它会提供新的语言特性和新的标准库，不过它几乎会 100%兼容 C++98。我们称它为 C++0x，并希望它成为 C++09。如果我们动作太慢（以致于 x 不得不变成十六进制的数），我（和其他人）都会感到非常难过和不安。

C++0x 几乎会 100%兼容 C++98。我们并不是特别想去破坏你的代码。最大的不兼容性问题出在几个新关键字身上，比如 `static_assert`、`constexpr` 和 `concept` 等。我们会选择未被大量使用的新关键字，设法将这种影响减到最低限度。主要的改进包括：

- 支持现代机器体系结构和并发：一个机器模型、一个线程库、线程本地存储器和原子运算，以及一个异步值返回机制（“未来”）。
- 更好地支持泛型编程：为模板定义和使用提供更好的检查，以及更好的模板重载的概念（用于类型、类型组合、类型和整数组合的一个类型系统）。基于初始化程序（auto）的类型推演、统一初始化程序列表、统一常量表达式（constexpr）、lambda 表达式等。
- 很多“微小的”语言扩展，比如静态断言、动态语义（move semantics）、改进枚举及空指针名称（nullptr）等。
- 支持正则表达式匹配、哈希表（例如 unordered_map）、“智能”指针等的新标准库。

具体细节，请访问“C++ Standards Committee（C++标准委员会）”的网站（注 3）。如欲概览，请访问我的在线 C++0x FAQ（C++0x 常见问题解答）（注 4）。

请注意：在谈到“不破坏代码”时，我指的是核心语言和标准库。旧代码如果使用来自一些编译器提供者或过时的非标准库的非标准扩展，当然就会被破坏。从我的经验来看，人们抱怨“破坏的代码”或“不稳定”时多是指私有特性和库。例如，如果你更换操作系统，而且没有使用一种可移植的 GUI 库，你就很可能要在用户接口代码上做一些工作。

是什么让您放弃了创建一种新的主流语言？

Bjarne：因为很快暴露出了一些关键问题：

- 新语言能解决什么问题？
- 它能为谁解决问题？
- 它究竟能新到什么程度（与各种已有的语言相比较）？
- 新语言能够获得实际应用吗（在这个拥有很多广受支持的语言的领域里）？
- 设计一种新语言是否只是一种愉快的经历，它能否将人们从构建更好的实际工具和系统的辛劳工作中解脱出来？

迄今为止，关于那些问题我还没有令自己满意的答案。

这并不意味着我认为 C++是一种完美的语言。它并不完美；我相信你能够设计出一种语言：它的规模是 C++（无论你采用哪一种方式来度量规模）的十分之一，而且能大致实现 C++的功能。不过，这种新语言的功能必须比现有语言更为强大，同时要稍微更好和更优雅一点。

您对现在和可预见的将来开发计算机系统的人们，就您这种语言的发明创新、进一步开发和采用问题方面，有什么经验和教训要说吗？

Bjarne：这是一个大问题：我们能向历史学习吗？如果能的话，怎么学习？我们可以学到什么样的经验和教训？在 C++开发早期，我明确提出了一套“经验法则”，你可以参考《The Design and Evolution of C++（C++的设计和发展）》[Addison-Wesley]，我在两篇 HOPL 论文中也有述及。确定无疑的是，任何严格的语言设计

注 3：*http://www.open-std.org/jtc1/sc22/wg21/*。
注 4：*http://www.research.att.com/~bs/C++0xFAQ.html*。

项目都需要一套设计原则，而且这些原则要尽可能明确地给出来。实际上，这是从C++设计中得到的经验：我没有尽早地明确提出C++的设计原则，也没有让大家足够广泛地理解那些原则。结果，就有很多人创造了他们自己的C++设计准则；其中一些是相当令人诧异的，结果也导致了许多混乱。直到现在，有些人还是将C++看成是没有设计出像Smalltalk这样的语言的失败后果（不对！我们并没有期望把C++设计成“Smalltalk”那样；它只是遵循OO的Simula模型），或者把它看成是为在C中编写C风格代码的某些缺陷的补救方法（不对！我们并没有期望C++只是对C进行修修补补）。

（非实验性）编程语言的目的是帮助构建“好”系统。它遵循系统设计与语言设计密切相关的理念。

在这里，我对“好”的定义是基本“正确、可维护并提供可接受的资源使用”。这个定义明显缺少了“易于编程”这个因素，不过对于我所考虑的大多数系统类型来说，那是次要的。“RAD开发”不是我的理想。能说清楚什么不是主要目的和什么是主要目的，这二者同样重要。例如，我并不反对快速开发（没有一个正常人想在项目上浪费不必要的时间），不过，我宁可不去限制应用程序领域和性能。对于C++来说，我的目标从来都是要直接表达思想，生成时间和空间高效的代码。

数十年来，C和C++一直都很稳定。这对工业用户来说非常重要。我有很多小程序，自从20世纪80年代早期以来一直没有实质性的修改。保持这种稳定性需要付出代价，但是，如果一种语言不具备这种稳定性，它就完全不适合需要长期运行的大型项目。企业语言和那些紧跟新潮的语言在这里一败涂地，总是为人们带来许多痛苦和烦恼。

这就引起了对于如何管理演进的思考。在多大程度上可以修改？修改的粒度如何？一种语言大约每年修改一次，就因为发布了一个产品的新版本，这样就太草率了，并会导致一系列事实上的子集、弃用库和语言特性和（或）大量的升级工作。而且，一年的酝酿期对于那些显著的特性来说显然不够充分，因此，这种干法会带来考虑欠周的解决方案并走进死胡同。另一方面，ISO语言（比如C和C++）标准化的10年周期时间太长，并会导致部分社区（包括部分委员会成员）陈旧僵化。

一种成功的语言会衍生出一个社区：该社区会共享技术、工具和库。企业语言有一个固有的优点：它们可以通过市场营销、会议和“免费”库来购买市场份额。这种投资可以在其他附加事项方面取得显著成功，使得这个社区规模更大并且更加活跃。Sun在Java上所做的工作，显示出了如何在非专业和资金短缺的情况下，做好创建一种（差不多称得上是）通用语言的每一项先期工作。美国国防部为把Ada创建成一种主流语言所付出的努力，与Java形成鲜明的对比，我和朋友们一起在没有资助的情况下创建出C++，这也是一种对比。

我不能说我赞成Java的一些策略，比如向非编程执行程序推销自顶向下的理念，不过它显示了可以干什么事情。非公司型成功包括Python和Perl社区。由于社区的规模所限，社区构建C++的成功经验太少，也太有限。ACCU会议是很棒，不过为什么自1986年前后以来，还没有一个连续的大型系列国际C++会议？Boost库是很棒，不过为什么自1986年前后以来，还没有一个C++库的中心仓库？现在使用的开放源代码C++库有好几千种。我甚至不知道一个全面的商业C++库清单。我不会回答那些问题，不过我要指出，任何新语言都必须以某种方式来管理大社区的离心力量，否则就会承受相当严重的后果。

一种通用语言需要若干群体的参与和认可，比如：行业程序员、教师、学术研究者、行业研究者，以及开放源代码社区等。这些群体不是相互割裂的，不过各个子群体通常会把自己看成是自信、独立的，它们认为自己知道什么是对的，而且，还会因“你没搞懂”这样的一些原因与其他群体发生冲突。这应该是一个很突出

15

16

的实际问题。例如，部分开放源代码社区反对使用 C++，理由是“这是 Microsoft 的语言”（其实不是）或“它是属于 AT&T 的”（其实也不是），然而一些主要的行业参与者也把 C++当成了一个问题，因为 C++并不是它们的。

这里真正至关重要的问题是，很多子群体正在推动有关“什么是真正的编程”和“真正需要什么”的狭隘观点：“如果每个人都以正确的方式编程，那就没有问题”。真正的问题在于平衡不同的需要，创建出一个更大的、更加多样化的群体。随着人数的增加和面临的新挑战，一种语言的通用性和灵活性开始变得很重要，而不仅仅是只为有限范围的问题提供最佳解决方案。

说到技术性问题，我仍然认为灵活的、可扩展的而且普通的静态类型系统非常好。我个人阅读 C++的经验强化了这种观点。我也非常渴望真正的用户自定义类型的局部变量：对于处理基于限定作用域的变量的一般资源来说，C++技术已经非常有效，与任何系统相比几乎都毫不逊色。通常和 RAII 一起使用的构造器和析构器，可以生产出非常优雅、高效的代码。

1.5 有关教学

您离开工业界，成为了一名学者。为什么要这样？

Bjarne：实际上，我还没有完全离开工业界，因为我作为一名 AT&T 会员，还维持着同 AT&T 实验室的联系，而且每年还要花很多时间和工业界人士在一起。我认为我同工业界的联系是绝对必要的，因为这样可以让我的工作紧密结合实际。

五年前，我到美国德克萨斯 A&M 大学当了一名教授，因为（在“实验室”工作将近 25 年之后）我觉得需要做一些变动，而且我也认为我可以为教育领域做一些贡献。我还有一些相当理想化的观点：在多年从事实际的研究和设计工作之后，我想多做一些基础性的研究工作。

很多计算机科学研究与日常问题相距甚远（即使是离未来想象中的日常问题，也相距太远），或者湮没在这种日常问题当中，它变得和技术转移别无二致。显然，我并不反对技术转移（我们迫切需要技术转移），不过，从工业实践到先进研究之间还应该有强大的反馈循环。行业内众多短期计划和学术出版/职位竞争的需要，共同促使了人们的注意力和努力从一些最关键的问题上转移开来。

在学术界的这些年间，您在初学者编程教学方面学到了什么？

Bjarne：我在学术界的这些年，最具体的成果（除了必须要写的学术论文之外）是出版了一本教授初学者编程的新教材《Programming: Principles and Practice Using C++（C++编程：原理和实践）》[Addison-Wesley]。

这是我出版的第一本面向初学者的著作。转向学术界之前，我完全不知道还有那么多的初学者需要这样一本图书。不过，我确实感觉到：对于完成本行业和其他领域的任务来说，有太多的软件开发者准备得极不充分。现在，我已经向 1 200 多名初学者讲授（以及帮助讲授）了编程课程，而且，我更加确信我在这个领域的观点可以调整。

一本启蒙书籍必须承担多种功能。最根本的是，它必须为进一步学习提供良好的基础（如果成功的话，它将

是终生努力的开始），并且能提供一些实际技能。同样地，编程（以及整体的软件开发）既不是一种纯粹的理论技能，也不是不用学习一些基本概念就可以做好的。不幸的是，教学始终总是难以在理论/原理和实践/技术之间维持一种平衡。因此，我们会遇到有人根本看不起编程（“只不过是编码而已”），他们还认为软件可以根据“第一原理”来开发，而不需要任何实际技能。相反地，我们也会遇到深信“‘优秀代码’就是一切”的人们，他们还认为先看两眼在线手册、再剪剪贴贴就可以获得“优秀代码”；我也遇到过认为 K&R（译注6）“太复杂而且太理论化”的程序员。我的观点是，这两种看法都过于极端，如此一来，不仅会导致结构化程度不高和效率低下，而且，即便是他们确实想要生成功能最简单的代码时，也会乱得一塌糊涂。

您对教材中的代码示例怎么看？它们应该包括错误/异常检查吗？它们应该是完整的程序以致于可以实际编译和运行吗？

Bjarne：我极为喜欢用尽可能少的代码阐释一个观点的例子。这样的程序段通常是不完整的，尽管我一再强调：如果把我的程序嵌入到合适的脚手架代码中，它就会正常编译和运行。基本上，我的代码展示风格都来自于 K&R。至于说我的新书，里面的代码示例全部采用可编译格式，全都可以使用。在正文中，我会经常在嵌入注释文本的小段和更长、更完整的代码段这两种方法之间来回切换。在关键的地方，我会对单个例子同时使用这两种技巧，这样一来，读者就会两次研读相关的关键语句。

一些例子应该包括错误检查，同时，所有的例子都应该反映可被检查的设计。除了有关错误及其处理的讨论遍布全书之外，错误处理和测试部分还分别独立成章。我极喜欢来自实际程序的例子。我确实很不喜欢矫揉造作的虚拟例子，比如动物的继承树和愚蠢的数学难题等。或许，我应该给我的书加上一个标签：“本书中的例子中没有滥用可爱的动物们”。

译注6：K&R，Brian W. Kernighan 和 Dennis M. Ritchie 编写的《The C programming Language》，简称 K&R，是最经典的 C 语言教程之一。

第 2 章

Python

Python 是 Guido van Rossum 开发的一种现代、通用的高级语言，这是他从事 ABC 编程语言工作的产物。Python 奉行实用主义哲学；它的用户常常会说起 Python 格言（Zen of Python，又称为 Python 之禅），极为推崇使用单一的、明显的方式来完成任何任务。虽然现在仍然存在着向 VM（比如 Microsoft 的 CLR 和 JVM）上移植的版本，不过，CPython 才是主流实现，它仍然是由 van Rossum 和其他志愿者开发的，他们刚刚发布了 Python 3.0，它是对该语言及其核心库部分向下不兼容的再思考。

2.1 Python 方式（译注1）

开发编程语言和开发“普通的”软件项目有什么区别？

Guido van Rossum：除了与大多数软件项目打交道以外，你最重要的用户还是程序员自己。这给语言项目提供了高级别的“元”内容。在软件项目的依存关系树中，编程语言差不多处于最底层：其他的所有东西都依赖于一种或多种语言。这也使得很难修改一种语言：不兼容的修改会影响很多“依存物”，以至于这样做通常并不可行。换句话说，一旦发布之后，所有的错误都会固定不变。C++很可能就是这种情况最极端的例子，它必须满足兼容性的需求，事实上，这种需求也许会要求 20 年前编写的代码仍然有效。

您如何调试一种语言呢？

Guido：你不用调试它。在语言设计领域中，敏捷开发方法学并没有什么意义：在该语言稳定之前，很少有人会用它，而且你在语言定义中找不着 bug，直到你已经有很多的用户为止，而此时你要修改已经为时太晚。

当然，实现中可能会有很多内容需要调试，比如所有老程序，不过语言设计自身大多要求提前仔细设计，因为 bug 的成本极高。

对于应该何时将某个特性扩展到库中，或者它何时需要核心语言的支持，您是如何做出决定的？

Guido：从历史来看，我对此已经有一个很好的答案。我很早就注意到，每个人都想把自己喜欢的特性添加到语言中，而且大多数人都对语言设计相对没有经验。每个人都在提议“把这个加到语言中”，“加一条实现某种功能的语句”。在很多情况下，答案是“好的，通过编写这样的两、三行代码，你就可以实现那种功能或者极为类似的功能，而且它也没那么难”。你可以使用词典，也可以将列表、元组数组和正则表达式组合起来，或者是编写一些元类——等等这一切事情。我对这个问题的最初答案甚至可能是来自 Linus，他好像也有类似的理念。

告诉人们你已经可以做到了，这就是第一道防线。第二件事是：“噢，这个很有用，而且，我们很可能会或者你也很可能会编写自己的模块、类或者封装抽象的特定位”。接下来，下一道防线是：“噢，这看起来非常有趣，也很有用，因此，我们会真正地接受它，并把它新加到标准库中，而且，它将是纯 Python 的”。最后，还有一些事情确实不适合采用纯 Python 来干，因此，我们会建议或者推荐如何把它们变成 C 扩展。C 扩展是在我们必须允许这么干之前的最后一道防线：“噢，是的，这非常有用，而且你对此确实也无能为力，因此，我们将不得不修改语言。”

到底是向语言中添加东西更有意义，还是向库中添加东西更有意义，要对这些做出决策，还需要其他标准，因为如果它必须使用命名空间语义或者诸如此类的东西，那么，就必须对该语言进行修改，此外别无他法。另一方面，扩展机制的功能非常强大，这样一来，使用 C 代码就足以完成非常多的工作，C 代码可以扩展库，甚至有可能添加新的内置功能，而不需要实际修改该语言。分析器没变，分析树没变，语言的文档也没变。所有工具仍然一切正常工作，而你已经给系统添加了新功能。

译注1：Python 方式（Pythonic Way），Pythonic 这个术语是指以 Python 的方式来编写代码、组织逻辑以及对象行为。

我想，你很可能已经考虑过这样一些特性：除了修改语言之外，这些特性无法在 Python 中实现，不过，你很可能会拒绝它们。你说这是 Python 方式，而那不是，你采用的是什么标准？

Guido：那样更难。在很多情况下，那很可能更是一种直觉。人们经常使用“Pythonic”这个词，也经常说“那是 Python 方式”，不过，到底什么是、什么不是“Python 方式”，没有人能给你一个无懈可击的定义。

你使用了“Python 格言”，但是除此之外还有什么呢？

Guido：就像每一本值得推崇的好书一样，它也会有很多解释。看到一个建议，我可以辨别出它是否优秀，不过，要编写一组规则，帮助别人区分语言修改建议的优劣，这个确实很难。

这听起来几乎就像是一个喜好的问题。

Guido：噢，首先就是要学会说“不”，并且，看看他们是否会就此罢休，或者是找个法子应付应付，而不用真的修改语言。值得注意的是它的频率如何。这不仅仅是“没有必要修改语言”这种操作性的定义。

如果你坚持不修改语言，人们仍然会想办法解决自己的需要。除此之外，通常还有来自的不同领域的用例问题，这些领域与特定应用程序无关。如果某些特性对于 Web 来说确实很棒，那么，对于加到语言中来说，就未必是优秀的特性了。如果它确实有利于编写更短的函数，或者是有利于编写可维护性更强的类，把它添加到语言中可能就是一件好事。一般来说，它确实需要超出应用领域的范围，并让事情变得更为简单或者是更为优雅。

如果要修改语言的话，每个人都会受到影响。你不可能把某种特性彻底隐藏起来，而大多数人却并不需要知道这种特性。人们迟早会遇到别人使用这种特性编写的代码，或者，他们会遇到一些隐蔽的边角情况，他们必须要学习这些边角情况，因为事情并不像他们预想的那样。

就优雅来说，通常也是仁者见仁，智者见智。我们最近在一个 Python 列表上讨论优雅问题，人们的争论非常激烈：使用 dollar 而不是 self-dot，会更加优雅。我认为，他们对于优雅的定义是击键的次数。

还有一种争议说，这会导致过度节俭，不过，它非常依赖于个人的喜好。

Guido：在很大程度上，无论是优雅、简单性，还是通用性，所有这些都取决于个人的喜好，因为，某些东西看起来好像能满足我的大部分要求，但它很可能满足不了别人的要求，反之亦然。

既然如此，Python 增强建议（PEP）处理又是如何出炉的呢？

Guido：这是历史上的一桩逸闻趣事。我认为，它主要是由核心开发者之一 Barry Warsaw 发起并支持的。1995 年，我和他开始在一起工作，而且，我想大概是在 2000 年左右，他建议我们需要更为正式的语言修改流程。

我认为，这些事情不必操之过急。我的意思是说，我并没有发现我们确实需要一个邮件列表。我没有发现邮件列表不便使用，我也没有发现我们需要一个新闻组。我没有提议我们需要一个网站。我也没有提议我们需要一个流程，用于讨论和实施语言修改，并确保避免出现这样的偶然错误：提出来一个建议之后，很快地接受它，而没有通盘考虑所有的后果。

在 1995 年和 2000 年之间，我和 Barry，还有其他几名核心开发者 Fred Drake、Ken Manheimer，一度都在 CNRI 工作，而 CNRI 的职责之一就是组织 IETF 会议。CNRI 最终把这个小部门变成了一个会议组织部门，而且它们的唯一用户就是 IETF。实际上，他们后来还一度举办过 Python 会议。因为，参加 IETF 会议相当容易，即

使他们不在本地也是如此。毫无疑问，我对 IETF 流程及其 RFC、会议小组与会议程序颇有心得，而且，Barry 对此也深有体会。因此，当他建议也组织一个类似的 Python 会议时，我们很容易达成了共识。我们特意决定，不会让它像当时的 IETF RFC 那样，变得那么强硬（heavy-handed），因为因特网标准，至少是其中的一些标准，它们会大大超出 Python 修改对于行业、人和软件的影响，不过，我们肯定会遵从那些标准。Barry 在起好名字方面是一个天才，因此我非常确信 PEP 是他的主意。

我们是当时第一批拥有这类东西的开放源代码项目之一，而且它也被相对广泛地进行了复制。Tcl/Tk 社区基本上是修改了标题，并使用完全相同的定义文档和程序，同时，其他项目也大都类似。

23

加上一小点形式体系会有助于 Python 增强设计决策的具体化，你发现确实是这样吗？

Guido：我认为随着社区越来越大，它变得很有必要，而且，我未必能判断出每一条建议本身的价值。对我来说，真正有用的是：让其他人就不同细节进行讨论，随后就会得出相对明确的结论。

他们会达成共识吗，在这种共识下，有人能要求你参加某个特殊的具体化期望和建议集？

Guido：是的。我最初赞成 PEP 时，我说："看来好像我们有些问题。我们来看看是否有人给出了正确的解决方案"，在某种程度上，它通常会采用上面这种方式。通常，他们会就问题应该如何解决给出一大堆明确结论，同时，也会提出大量的未解决的问题。有时，我的直觉可以帮助解决尚未解决的问题。在我感兴趣的领域，我就会对 PEP 过程非常积极：如果要必须添加一条新的循环控制语句，我就不想让别人来设计。而有些时候，我就不太关心，比如说数据库 API。

是什么引发了对新的大版本的需求？

Guido：这要取决于你对"大"的定义。在 Python 中，我们通常会把 2.4、2.5 和 2.6 版本当成"大"事，这种事每 18–24 个月就会出现一次。如果我们要引入新特性，这些事件就是唯一的机会。很久以前，每一次发布都源于开发者（特别是我）的心血来潮。不过，这十年的早些时候，用户就开始要求某种程度的可预测性——他们反对在"小"修改（例如，与 1.5.1 相比，1.5.2 添加的主要特性）时添加或者修改特性，而且，他们希望大版本维持某个确定的最短时间段（18 个月）。因此，现在我们发布的大多是基于时间的大版本：我们以发布管理程序可用性诸如此类的事情为基础，预先规划好大发布之前的一系列时间节点（例如，何时发布 α 版、β 版以及发行候选版等）（译注2），而且，我们还极力主张开发者在最终发布日期之前事先改好。

核心开发者们讨论了特性优点及其精确规范（有时是长时间讨论）之后，通常会就选择将哪些特性添加到版本之中达成一致意见。这就是 PEP 流程：Python 增强建议，作为一个基于文档的流程，它与 IETF 的 RFC 流程或者 Java 领域的 JSR 流程并无不同，除了我们不是非常正式之外，因为我们的开发者社区规模非常小。如果长期达不成一致（无论是就某个特性的优点，还是具体的细节），最后我就会采用平分决胜的方法；这种算法通常是靠直觉，因为，一旦要靠调用这种方法来解决问题，那就已经没有什么理性思考（rational argument）可言了。

大多数有争议的讨论通常都与用户可见的语言特性有关；添加库通常是很容易的（因为，它们并不会损害不关心的用户），而且，内部改进并不是真正被考虑的特性，尽管它们在 C API 级上受到相当严格的向下兼容限制。

译注2：发行候选版 (Release Candidate，RC)，指可能成为最终产品的版本，如果没有再出现问题就可以发布正式版本。在此阶段，产品是接近完整的，包含的所有功能也不会出现严重问题。和 Beta 版最大的差别在于 Beta 阶段会一直加入新的功能，但是到了 RC 版本，几乎就不会加入新的功能了，而主要着重于除错。

由于开发者是典型的最直言不讳的用户，我确实无法区分哪些特性是由用户或是由开发者建议的：开发者通常会基于认识的客户中提出的需求，从而提出某个特性的建议。如果某个用户提出一个新特性，它几乎不会成功，因为，如果用户对实现（通常还有语言设计和实现）没有全面的理解，他几乎不可能提出一个合理的新特性。我们希望让用户说出他们的问题，而不是给出具体的解决方案，然后开发者会给出解决方案建议，并与用户一起讨论不同方案的优点。

这里还有一个完全“大”或者“重大进展”版（akthrough version）（比如3.0）的概念。从历史上看，1.0版几乎都是由0.9版演进而来的，而2.0版也是对1.6版做了小幅改进。从现在开始，因为有了更大的用户群基础，这样的版本的确很少见了，而且，它为真正兼容前面的版本提供了唯一的机会。大版本都会向下兼容前面的大版本，同时，还建立了一种特殊的申诉机制，用于对准备删除某些特性提出反对意见。

您为何选择把数字作为任意精度整数（用于你所得到的极棒的所有优点）来处理，而不是像以前（以及极为常见的）那样把它交给硬件来处理？

Guido：我最初是从Python的前身ABC继承了这个观点。ABC使用了任意精度的有理数，不过，我没那么喜欢有理数，因此我把它换成了整数；对于实数，Python则使用了硬件支持的标准浮点表示法（而且，在某些力量的推动下，ABC也是这么干的）。

最初Python有两种类型的整数：常见的32位类型（“int”）和个别的任意精度类型（“long”）。很多语言都是这样，不过任意精度类型被转移到一个库中，比如Java和Perl中的Bignum，或者C的GNU MP。在Python中，这两种类型在核心语言中（几乎）一直是并驾齐驱，而且用户必须选择使用哪一种类型，如果是选用long类型的话，就要在数字后面附加一个“L”。这逐渐成为了一种烦恼；在Python 2.2中，如果对int的运算结果在数学上是正确的，如果无法表示成int，我们会把它自动转换成long（例如，2**100）。

以前，这会导致`OverflowError`异常。以前有一次，本来运算结果应该能悄悄地取整（不舍入），不过我对它进行了修改，在让别人使用语言之前，抛出一个异常。20世纪90年代初期，我花了一下午时间来调试我编的一个很短的演示程序，这个程序实现了一个非显而易见（non-obvious）使用超大整数的算法。这样的调试过程堪称一种创造性的经历。

不过，在某些特定情况下，这两种数字类型仍然会稍有不同；例如，以十六进制或者八进制格式打印一个*int*值，可能会产生无符号结果（例如，-1会被打印成FFFFFFFF），而以long来打印，虽然在数学上是相等的，但它会产生有符号结果（在这种情况下是-1）。在Python3.0中，我们采取了一种激进的做法：仅仅支持单一整数类型，我们称之为int，不过，它在很大程度上仍然是原来的long类型。

为什么您说是激进的做法（radical step）呢？

Guido：这主要是因为它在很大程度上偏离了Python目前的实践。在这些方面有很多讨论，而且，人们还提出了多种不同的备选方案，其中有两种（或者更多的）表示法可以在内部使用，但是它们全部或者大部分是对最终用户（除了C扩展的作者以外）隐藏的。这种做法的表现可能会稍微好一点，不过最终它的工作量非常庞大，而且在内部使用两种表示法，只是会增大使之正常运行的工作量，而且会使C代码及其接口更为麻烦。我们现在希望它对性能的负面影响要小一些，而且，我们还可以使用缓存等其他技术来改进性能。

您为何奉行“应该有一种，而且最好是只有一种明显的方式来完成任务”这种哲学？

Guido：最开始，这很可能是潜意识的。Tim Peters 在编写“Python 格言”（你就是引用了这个格言）时，制定了许多明确的规则，我在自己还没意识到的情况下，已经应用了这些规则。那就是说，这种特定的规则（虽然通常会经我同意违反这个规则）直接来自于对数学和计算机科学中优雅的普遍渴望。ABC 的作者也采用了这些规则，他们也渴望少量的正交类型或概念。正交性的概念是从数学中直接照搬过来的，它是指只有一种方式（或者一种真正的方式）表示某事的定义。比如，对于 3D 空间的任意一点来说，一旦你已经选定了一个原点和三个基本向量，那么，这个点的 XYZ 坐标就是唯一确定的。

我也喜欢认为：我也是在帮助大多数用户，因为不需要他们在类似的备选方案之间选择。你可以拿这个与 Java 做一下对比，在 Java 中，如果你需要一个链表类的数据结构，标准库会提供很多版本（一个连接链表，或者一个数组列表等）；或者，你也可以与 C 做一下对比，在 C 中，你必须决定如何去实现你自己的链表数据类型。

对于静态类型和动态类型，您持何种观点？

Guido：我希望我能给出“静态类型不好，动态类型好”这样的简单回答，但是，事情未必总是如此简单。动态类型有从 Lisp 到 Python 等多种不同的方式，同时，静态类型也有从 C++到 Haskell 等不同方式。像 C++和 Java 这样的语言，很可能会给静态类型带来恶名，因为它们会要求你反复告诉编译器同一件事情。不过，像 Haskell 和 ML 这样的语言，它们使用的是类型推理，这就有很大的不同，而且它还具有动态类型的某些优点，比如在代码中对概念的表达更加简洁。不过，就函数式范型本身而言，它看起来好像难以使用——像 I/O 或者 GUI 交互之类的功能并不能很好地适用于那种模式，而且，通常是借助一个更传统的语言（例如 C）的连接桥来解决。

在某些情况下，必须另外考虑 Java 的冗长性；它使得创建功能强大的代码浏览工具成为可能，这种工具可以回答像“这个变量有什么变化？”或者“谁调用了这个方法？”之类的问题。如果要使用动态语言，回答这样的问题就更为困难，因为通常难以搞清楚一种方法参数的类型，而不用分析整个代码库中的每一条路径。根据我有限的理解，我不确认像 Haskell 这样的函数式语言如何支持这种工具：要是你必须使用本质上与动态语言相同的技术就好了，因为那是类型推理肯定要做的事情！

我们正在往混合类型上转吗？

Guido：我想对于某些混合方式来说，它有很多说头儿。我已经注意到，使用静态类型语言编写的大多数大系统，实际上包含一个很大的、本质上是动态类型的子集。例如，用于 Java 的 GUI 窗口小部件集和数据库 API，它们给人的感觉通常是与静态类型步步为敌，将大多数正确性检查转到运行时上。

一种混合语言同时带有函数式和动态观点，可能会让人很感兴趣。我应该把它们添加进去，尽管 Python 支持一些像 `map()`和 `lambda` 这样的函数式工具，不过，Python 没有函数式语言子集：它没有类型推理，而且也没有机会实现并行化。

为什么您选择支持多种范型？

Guido：其实我并没有这样选择；Python 支持过程式编程，并在在某种程度上支持面向对象。这两者并非是截然不同的，而且 Python 的过程性风格仍然受到对象的强烈影响（因为基本数据类型都是对象）。Python 只是在很小的程度上支持函数式编程——不过，它与任何真正的函数式语言都不相似，而且也永远不会相似。函数式语言是尽可能在编译时间完成所有的工作——“函数式”的观点意味着，编译器可以在强力保证没有

副作用的情况下实现优化，除非是显式声明。Python 可能会拥有最简单、最蠢笨的编译器，而且，正式的运行时语义极力反对编译器的这种智能性，比如说并行化循环或者将递归变成循环等。

因为 Python 包含了 `lambda`、`map`、`filter` 和 `reduce` 等函数以支持函数式编程，它很可能是因此获得了声誉，不过在我看来，这些只不过是语法甜头（syntactic sugar）而已，而不是像它们在函数式语言中那样是基本构造块。Python 与 Lisp（也不是一种函数式语言!）二者共有的更基本的属性是：其中的函数是第一类对象，并且可以像任何其他对象那样传送。这与 state 函数的嵌套作用域和通用的类 Lisp 方式联合，可以很容易地实现那些简单模仿函数式语言的概念，比如 currying、map 和 reduce 等。Python 中内置了实现那些概念所必需的基本运算，在函数式语言中，那些概念就是基本运算。你可以用几行 Python 代码编写出 `reduce()`，如果是使用函数式语言可不行。

您创建语言时，考虑过可能会吸引到哪些程序员了吗？

Guido：是考虑过，不过我很可能还没有足够的想象。我考虑过 Unix 或者类 Unix 环境中的专业程序员。Python 教程的早期版本有一个口号，大意是“Python 架起了 C 和 shell 编程之间的桥梁”，因为那就是我本人以及我周围人的境况。直到人们开始打听 Python 为止，我从未想到它可能会是嵌入到应用程序之中的优秀语言。

它可以用于在中学或者大学教授编程基本原理，或者是用于自学，这个事实只不过是一种幸运的巧合而已，这得益于我在 Python 中保留的诸多 ABC 的特性：ABC 的目的很明确，就是向非程序员教授编程。

对新手来说，这种语言要易于学习；而对于富有经验的程序员来说，这种语言的功能应该足够强大。您如何平衡这些不同的需要？这是一种错误的二分法吗？

Guido：答案就在于平衡。要避开一些众所周知的陷阱，比如有益于新手而让专家烦恼的问题，或者是专家需要而令新手困惑的问题。要具有足够的空间让双方同时感到满意。另一种策略是给专家提供一些方法来解决初学者不会遇到的高级问题——例如，该语言支持元类，不过新手则不需要知道这些。

2.2 优秀的程序员

您如何发现优秀的程序员？

Guido：发现优秀的程序员需要时间。例如，一个小时的面试很难真正地区分程序员的好坏。你和其他程序员一起工作解决多种问题时，孰优孰劣自然会一目了然。我不愿意给出具体的标准：一般来说，我想优秀的程序员会显示出创造力，学习速度很快，而且很快开始编写出可用的代码，而且这些代码在检查之前不需要许多修改。请注意：每个人在擅长的编程方面都各有千秋：有些人擅长算法和数据结构，而其他人则擅长大规模集成、协议设计、测试、API 设计、用户接口，或者是编程的其他方面。

您会使用什么方法来聘用程序员？

Guido：根据过去的面试经验，我认为我并不擅长传统的聘用方式：无论是作为雇员还是雇主，我都几乎没有什么面试技巧！我想我应该使用某种学徒制度，和他们一起密切地工作一段时间，并最终对他们的优点和缺点形成一种印象。这在一定程度上有点像开放源代码项目的工作方式。

如果我们正在寻找很优秀的 Python 程序员，有什么基本的评价特征吗？

Guido：很抱歉，我想你提出这个问题，是从传统管理者的观点来看的，他们只想聘用一批 Python 程序员。我认为这个问题根本没有一种简单的答案，而且事实上我认为这很可能是一个错误的问题。你并不是想聘用 Python 程序员。你是想聘用聪明、有创造力而且自觉主动的人。

28

如果你看一下程序员职位的招聘广告，几乎全都有这么一行文字：能够与团队协作。究竟团队在编程中起何作用，您怎么看这个问题？对于那些无法与其他人合作的聪明的程序员，您认为他们仍然会有生存空间吗？

Guido：我非常支持工作职位广告在此方面提出的要求。无法进行团队协作的聪颖程序员，不应该让自己来应聘传统的编程职位：对于相关的方方面面来说，这都会是一场灾难，并且，无论是谁来继承他们的代码，都会是一场恶梦。实际上我认为，如果你无法与团队协作，这显然是不够明智的。现在，学习如何与其他人一起工作有很多途径，况且，如果你是真正的聪明，你应该能很容易地学会团队协作技能：实际上，如果你用心的话，它并没有像学习如何实现一个高效的快速傅立叶变换那么困难。

作为 Python 的设计者，与使用 Python 的其他熟练开发者相比，您使用自己的语言编码时看到了什么优点？

Guido：我不知道——此时，已经有这么多人接触了这种语言和 VM，我有时候会惊讶我自己具体是怎么干这些事的！如果说我比其他开发者有优势的话，这个优势很可能就是我比任何人使用这种语言的时间更长，而不是它是我本人编写的。经过这么长的时间之后，我有机会考虑哪些运算更快或更慢：例如，我可能会比大多数用户更明白局部要比全局速度更快（虽然别人过分爱用“全局”，我可不是这样！），或者是函数和方法调用代价昂贵（比在 C 或者 Java 中更昂贵），或者最快的数据类型是元组。

在使用标准库以及其他事情上，我常常感觉到别人会更有优势。例如，我每隔几年就会编写一个 Web 应用程序，每次使用的技术都在变化，结果每次都是以我使用新框架或者新方法编写“第一个”Web 应用程序而告终。而且，我现在仍然没有机会使用 Python 进行严重的 XML mangling（毁坏）。

看来，简明扼要是 Python 的特性之一。这个特性对代码可维护性有何影响呢？

Guido：我听说研究和逸闻都有证据表明，不管使用什么样的编程语言，特定行数的代码错误率都是相当一致的。因此，像 Python 这样的一种语言，用它编写的典型应用程序要比用 C++或者 Java 编写的同样功能的程序小很多，如此一来，该应用程序的可维护性就会更强。当然，这很可能意味着一名程序员要承担更多的功能。这是另外一个问题，不过它的结果仍然对 Python 有利：每个程序员的生产率越高，很可能是意味着一个团队需要的程序员越少，与此相对应，沟通成本也较小，如果我没记错的话，按照 Mythical Man-Month [Frederick P. Brooks; Addison-Wesley Professional]的说法，沟通成本将以团队规模的平方速度增长。

29

您认为，在 Python 提供的原型易用性和构建完整的应用程序所需的工作量之间，二者有什么联系吗？

Guido：我从来没有认为 Python 是一种原型语言。我认为，在原型和“产生式”语言之间不应该有明显的区别。有时候，编写一个原型的最佳方式应该是编写一个“用完即扔的一次性”C 语言小 hack 程序（译注3）。有时候，可能根本不会使用“编程”来创建一个原型——例如，使用一个电子表格或者一组 find 和 grep 命令。

在 Python 设计之初，我的目的非常简单：想让这种语言用于 C 显得大材小用、而 shell 脚本又过于麻烦的场

译注3：hack 程序，是指一种辅助程序，它完成对系统功能的扩展，即替换或者增强原有的系统功能。这种偷梁换柱修改系统的工作方式，就像网络上的黑客侵入并接管别人的网络或者计算机一样，hack 程序由此得名。

合。它涵盖了许多原型，但又涵盖了许多“业务逻辑”（这是它现在的叫法），它在计算资源方面并不是特别贪婪，但是要编写很多代码。我想说的是，大多数 Python 代码不是作为一个原型编写的，而仅仅是为了完成工作。在大多数情况下，Python 能完全胜任它的工作，而且实现最终的应用程序也不需要做太多修改。

常见的场景是，某个简单应用程序的功能逐渐增多，最终结果是复杂性增加了十倍，而且，从原型一直到最终的应用程序，中间从来没有一个精确的转折点。例如，我在 Google 开始写的代码检查应用程序 Mondrian，自从首次发布以来，它的代码规模很可能已经增长了十倍，而且它仍然是全部使用 Python 编写的。当然，也有 Python 最终被发展更快的语言所取代的例子——例如，最早的 Google 网络爬虫/网络索引器（主要）是用 Python 编写的——不过那些是例外情况，而不是普遍规则。

Python 的即时性会对设计过程有何影响呢？

Guido：这是我常用的工作方式，而且，至少对于我来说，它通常会非常成功！当然，我编写过许多已被我扔掉的代码，不过，这要比我用任何其他语言编写的代码更少，而且编码（即使是没有运行它）通常也会有助于我更深入地理解细节问题。考虑如何重新调整代码，以致于以一种最佳的方式来解决问题，这通常会对我考虑问题有很大的帮助。当然，这不能用作逃避使用白板来画出设计草图、体系结构、交互作用或者其他早期设计方法的一个借口。诀窍就是使用正确的工具：有时候，需要一支铅笔和一块餐巾——而其他时候则需要 Emacs 窗口和 shell 提示符。

您认为自底向上程序开发更适合 Python 吗？

Guido：我并没有把自底向上和自顶向下看成像 vi 和 Emacs 那样的严格对立。在任何软件开发过程中，有时候你会使用自底向上的方式，有时候又要使用自顶向下的方式。自顶向下，很可能意味着你所处理的事情需要在编码之前仔细检查和设计，而自底向上，则很可能是意味着在现有基础上构建新的抽象概念，例如，创建新的 API。我并不是暗示你应该事先不做设计就开始编写 API，不过新的 API 逻辑上通常是从可用的较低级 API 得出的，而且，你在实际编写代码时要进行设计工作。

您认为 Python 程序员什么时候才会意识到 Python 动态特性的好处呢？

Guido：当你在研究一个大问题或者解空间，而你还不知道该怎么做时，语言的动态特性通常是最有用的：你可以做大量的实验，每个实验都会吸取上一次实验的经验，而不需要将你禁锢在某种特定方法中的太多代码。它对你确有帮助，因为使用 Python 可以编写出很紧凑的代码：你可以一次编写 100 行 Python 代码来做实验；然后再重新开始；也可以是用 Java 编写 1 000 行框架用于实验，然后才发现情况不对。两相对比，显然是前者效率更高。

从安全性的观点来看，Python 给程序员提供了什么？

Guido：这取决于你是否担心发生攻击。Python 会自动分配内存，因此 Python 程序并不容易出现 C 和 C++ 代码常见的特定类型的 bug，比如缓冲区溢出或者利用解除分配内存等，这些都已经成为攻击 Microsoft 软件的重要途径。当然，Python 运行时自身也是用 C 编写的，这些年来确实也发现了一些脆弱性，另外还有故意逃避 Python 运行时限制的行为，比如可以调用任意 C 代码的 ctypes 模块等。

它的动态本质会有好处吗？或者是正好相反？

Guido：我并不认为动态本质有什么好处或者坏处。一个人可以很容易设计出一种具有许多弱点的动态语言，

或者是一种没有弱点的静态语言。不过，拥有一个运行时或者虚拟机（当前这些词很“时髦”），通过限制访问原始底层的机器，这样会好一些。很巧的是，这也是 Python 成为 Google 应用引擎（Google App Engine）支持的第一种语言的原因之一，目前，我正在参与这个项目。

Python 程序员怎么样检查和提高代码安全性呢？

Guido：我认为 Python 程序员不应该过分担心安全性，脑子里肯定不是没有一个具体的攻击模型。你要干的最重要的事情就是：怀疑你不信任的人提供的数据（对于 Web 服务器来说，就是接收到的 Web 请求的每一个字节，甚至是头），这在所有语言中都是一样。需要具体关注的就是正则表达式——编写在指数时间轴运行的正则表达式很容易，因此，使用正则表达式实现搜索最终用户在哪里输入的 Web 应用程序，应该有一些机制来限制运行时间。

要想熟练使用 Python 进行开发，您有什么基本概念（通用规则、观点、心态和原理）方面的建议吗？

Guido：我想说实用主义。如果你过分纠缠数据隐藏、访问控制、抽象或者规范这样的理论概念，你就不是一位真正的 Python 程序员，而且，你最后会将时间浪费在与这种语言为敌上面，而不是使用（并享受）它；你也很可能是低效地使用它。如果你是像我这样的一个“即时满足（instant gratification）”迷，Python 就非常不错。如果你喜爱极限编程这样的方式或者其他敏捷开发方法，它的表现就很出色，尽管我会建议凡事都要适度。

您说的“与这种语言为敌”是什么意思？

Guido：我说这话的意思，通常是指他们在使用一种不同的语言时，试图延续自己的习惯。

新近改用 Python 而仍未习惯的人们会提出许多建议，以某种方式去除显式 self（译注4）。对于他们来说，这已经变成了一种困扰。有时候，他们会提出一种语言修改方案；有时候，他们又提出一些超级复杂的元类，以某种方式把 self 变成隐式。这么干的效率通常极低，或是在多线程环境中也无法实际工作，或者无论什么其他边界情况，或者他们也困惑于必须键入那四个字符，以至于他们修改约定，把 self 改成“s”或者大写的“S”。人们会将每件事都变成一个类，并将每一个访问变成一个存取器方法，在 Python 中这么干实在不是什么明智之举；你只会得到更冗长的代码，它不仅更难调试，而且运行速度更慢。你听到过“你可以使用任何语言来编写 FORTRAN”这样的说法吗？你也可以用任何语言来编写 Java。

您花费了这么多的时间尝试创建（如果可能的话）一种明显的方式来完成任务。看起来好像您相信：以 Python 方式来做事，可以让你真正地利用 Python。

Guido：我不确认我真正花了很多时间来确认只有一种方式。“Python 格言”比 Python 语言更年轻，而且该语言的大多数定义特征在 Tim Peters 将它作为诗歌写下来之前，已经存在很久了。我认为他在写完时，并没有预料到它会如此广泛传播和并取得如此的成功。

这是一种诱人的说法。

Guido：Tim 在措辞方面很有一套。在大多数情况下，“这只有一种解决方式”实际上只是一个善意的谎言。

译注4：Python 中 class 下定义的函数，都默认要一个参数为 self。一个类通常会有多个方法，它们都以关键字 def 开头，并且第一个参数通常都是 self，Python 中的变量 self 相当于 C++中的关键字 this，其作用是传递一个对象的引用。

数据结构可以采用很多方式。你可以使用元组和链表。在很多情况下，你使用的是一个元组、链表，还是一个词典，这并没有什么关系。经验表明，如果你看起来确实很小心，解决方案肯定会更好，因为它会在多种情况下工作，而且，还有一、两种情况，在链表或者元组保持增长时，前者要比后者效果好很多。

其实，组件很分散更多地是来自于原始的 ABC 哲学。实际上，ABC 和 ALGOL-68 遵循同一种设计哲学，虽然它现在已经成为活力最差的语言之一，不过，当年 ABC 的影响非常大。毫无疑问，在 20 世纪 80 年代期间，它是非常有影响的，因为 Adriaan van Wijngaarden 是来自 ALGOL 68 的大师。我转到大学时，他仍然在教课。的确有一个或者两个学期，他只是在讲 ALGOL 68 的历史趣闻。他曾经是 CWI 的负责人。我进来时已经换人了。

有很多人曾经非常熟悉 ALGOL 68。我认为 ABC 的主要作者 Lambert Meertens，他也是 ALGOL 68 报告的主要编者之一，这很可能是意味着他做了许多排版工作，不过他也许偶尔也会进行很多思考和检查。他无疑是受到了 ALGOL 68 中提供结构的设计哲学的影响，这种结构可能会以多种不同方式组合起来，产生各种各样不同的数据结构或者构造程序的方式。

这绝对是他的影响，即："我们拥有列表或者数组，而且它们可以包含其他一切。它们可以包含数字或者字符串，不过它们还可以包含其他数组和元组。你可以将所有的这些组合在一起"。突然，你并不需要单独的多维数组的概念，因为一个数组的数组就可以解决任何维度的问题。抓住几件关键的事情，它们包含了灵活性的不同方面，并允许它们联合起来，这正是来自 ABC 的哲学思想。我全盘借用了这些思想，几乎没有对它进行认真的考虑。

虽然，看起来你在 Python 中可以以非常灵活的方式进行组合，只要你不在表达式之内嵌套语句，实际上，句法中仍然存在着大量的下列情况：有些情况下，逗号代表参数之间的分隔；而在其他情况下，逗号则代表一个列表；而且，在另一个情况下，它又代表着一个隐式元组。

句法中有一大堆变种，它们不允许使用特定的操作符，因为会与周围的一些句法冲突。这从来都不是一个真正的问题，因为你总可以在它不工作时额外加上一对圆括号。因为，至少从分析器作者的观点来看，那个句法已经成熟了许多。像列表理解（list comprehension，或称列表包含）和生成器表达式诸如此类的概念在句法上仍然没有完全统一。在 Python 3000 中，我相信它们可以统一。虽然，它们仍会有一些细微的语义区别，不过，至少句法是相同的。

2.3 多种 Python

在 Python 3000 中，分析器变得更简单了吗？

Guido：几乎没有。它没有变得更复杂，不过也没有真正地变得更简单。

我认为没变得更复杂就是一种胜利。

Guido：是的。

为什么想到的都是些最简单、最蠢笨的编译器？

Guido：最初这是一个很实际的目标，因为我不是"代码生成"科班出身。只有我一个人单干，而且，在我

能干其他有关语言的有趣工作之前，我必须得先有一个字节码生成器。

33 我仍然相信有一个很简单的分析器是一件好事；毕竟，它是将文本转换成一棵树，用以表示程序结构。如果句法含混不清，以致于确实需要使用先进技术才能把它搞清楚，那么人类读者很可能也需要相当长的时间。这样一来，确实很难编写另一个分析器。

Python 的分析简单得不可思议，至少在句法层面是这样。在词法层面，分析要相对精细一些，因为你必须读取带有少量堆栈操作的缩进，这些堆栈是嵌入在词法分析器中的，它是词法和语法分析分离理论的一个反例。不过，这是正确的解决方案。有趣的是，我喜欢自动生成的分析器，不过，我对于自动生成的词法分析并没有很强的信心。Python 一直有一个人工生成的扫描器和一个自动分析器。

人们已经为 Python 编写了很多不同的分析器。即使是 Python 移植到不同的虚拟机上，无论是 Jython、IronPython，还是 PyPy，也都有它自己的分析器，而且，这也不算什么大事，因为分析器从来都不是一个非常复杂的项目，因为它的语言结构，使用最基本的单个 token 预测递归下降分析器就可以非常容易地进行分析。

分析器变慢的原因，实际上就在于二义性，它只能在程序结束之前通过提前规划而消除。在自然语言中，有很多这样的例子：如果你还没有读完一条语句的最后一个单词和任意嵌套，那就不可能对它进行分析。或者也有些语句，你只有确实知道它们说的是什么，你才能够分析，但那是完全不同的情况。对于编程语言分析来说，我喜欢我这样的单 token 预测分析器。

它暗示我，Python 中可能从来不会使用宏，因为那时必须要执行另一个分析阶段！

Guido：将宏嵌入分析器中有多种方法，它们都能正常工作。不过，我根本不确信宏能解决 Python 语言面临的特别迫切的所有问题。另一方面，由于该语言易于分析，如果你提出某种适合在语言句法之内使用的健康的宏集，将微观评价实现为分析树操作可能会非常简单。我对那个领域并不是特别感兴趣。

为什么您在源代码中选择使用了严格的格式化？

Guido：在 Python 中选择分组缩进，这并不是一个新概念；我是从 ABC 中继承了这个概念，不过 occam 这种很老的语言也使用了这种概念。我不知道 ABC 作者的思想是来自于 occam，还是它的独自创造，或者是他们是都源于同一祖先。这种思想可能是出自 Don Knuth，他早在 1974 年就提出了这种思想。

34 当然，我当时也可以选择不去追随 ABC 的风格，就像我在其他领域那样（例如，ABC 的语言关键字和过程名称，使用的是大写字母，我没有照搬这个观点），不过我最终还是非常喜欢 ABC 的那些特性，因为它好像能平息当时 C 用户中间某个毫无意义的争论，也就是在哪里放置波形括号的问题。我也充分意识到可读的代码都是使用自动缩进来表示筛选，而且，我还偶然发现了代码中的细微 bug：缩进与使用波形括号的句法筛选并不一致——程序员和所有检查者都假定缩进与筛选相匹配，因此并未注意到该 bug。此外，调试过程时间很长，这也是一个颇有价值的教训。

严格的格式化应该会生成一个更整洁的代码，而且很可能会减少不同程序员的代码“布局”区别，不过这听起来好像是强制人去适应机器，而不是相反？

Guido：完全相反——它对人类读者更有帮助，而不是对机器；看看前面的例子。在维护另一个程序员编写的代码时，这种方式的优点很可能会更明显。

最初，新用户通常会因此望而却步，尽管我已经很少听说这些了；或许，教授 Python 的人们已经学会了预先想到了这些，并进行了有效的反驳。

我想请教您关于 Python 的多个实现问题。现在有四五个影响很大的实现，包括 Stackless 和 PyPy。

Guido：从技术上来看，Stackless 并不是一种独立的实现。Stackless 通常被列为一种独立的 Python 实现，因为它是 Python 的一个分支，使用不同的方式取代了很小部分的虚拟机。

基本上是字节码分派，对不对？

Guido：大多数字节码分派是非常类似的。我认为字节码是相同的，所有的对象当然也是相同的。你从一个 Python 过程调用另一个过程时，二者就会表现出不同：它们使用对象操作完成了调用，其中它们只是向栈帧压栈（译注5），而且和 C 代码的相同位仍然负责。C Python 对此的实现方式是，C 函数在那一点上被调用，最终会调用一个虚拟机的新实例。它不是真正的整个虚拟机，而是解释字节码的循环。在 stackless（译注6）的 C 堆栈中只有唯一的循环。在传统的 C Python 中，你可以在 C 堆栈进行多次相同的循环。这是唯一的区别。

PyPy、IronPython 和 Jython 是独立的实现。我不了解转到 JavaScript 上的事情，不过，如果有人在某些方面已经走得很远，我并不会感到惊讶。我已经听说转到 OCaml 和 Lisp 上的实验性工作，而且有些人知道这些。从前还有过转到 C 代码上面的事情。

在 20 世纪 90 年代后期，Mark Hammond 和 Greg Stein 曾致力于此，不过他们发现速度提高的程度相当有限。在最理想的环境下，它的速度应该能提高一倍；同时，生成的代码非常庞大，以致于你所拥有的库会变得相当庞大，而且，这也变成了一个问题。

启动时间也会让你苦恼不已。

Guido：我认为 PyPy 那些人的想法是对的。

听起来您好像通常会支持这些实现。

Guido：我一直是支持备选的实现。从 Jim Hugunin 涉足一定程度上的 JPython 完全实现那一天起，我就对此非常兴奋。在某种意义上来说，它可以算作语言设计的一种验证。它还意味着，人们可以在某个平台上使用自己喜欢的语言，否则他们就不会访问这个平台。我们仍然有一种方式，不过它肯定有助于我区分哪一个特性是我所关心的真正的语言特性，而且哪一个特性是特定实现的特性，我也赞同处理方式不同的其他实现。就是在这里，我们不幸地以垃圾收集的灾难性急剧下滑趋势而告终。

这种灾难性的急剧下滑趋势一直存在。

Guido：不过，这也是必然的。我无法相信我们能长期忍受纯引用计数和无法打破循环的状况。我一直把引用计数看作是收集垃圾的一种方式，而且还不是特别糟的方式。过去经常会因对引用计数和垃圾收集的观点不同而产生激烈争论，而这些在我看来是相当愚蠢的。

译注5：在当今流行的计算机体系架构中，大部分计算机的参数传递、局部变量的分配和释放都是通过操纵程序栈来实现的。栈用来传递函数参数，存储返回值信息，保存寄存器以供恢复调用前处理机状态。每次调用一个函数，都要为该次调用的函数实例分配栈空间。为单个函数分配的那部分栈空间就叫做栈帧（stack frame），也就是说，栈帧这个说法主要是为了描述函数调用关系的。

译注6：stackless Python 是 Python 的一个增强版本。stackless Python 修改了 Python 的代码，提供了对微线程的支持。微线程是轻量级的线程，它在多个线程间切换所需的时间更多，占用资源也更少。

再次考察这些实现，我认为 Python 是一个有趣的空间，因为它有一个相当好的规范。与其他语言相比，比如说 Tcl、Ruby 和 Perl 5 这样的语言相比，肯定是这样的。出现某些事情是因为你想要标准化语言及其行为，还是因为你在考虑多种实现，或者是别的什么东西？

Guido：它很可能更是关于 PEP 和多种实现的社区流程的一个负面效应。我最初编写第一个文档集时，曾经充满热情地创办了一本语言参考手册，并期望它成为一个足够精确的规范，即便是火星或者木星来客也能够实现这种语言，并正确地理解它的语义。不过，我从来没有在实现那个目标方面取得什么进展。

ALGOL 68 很可能是所有语言中最接近高度数学化规范的语言。其他语言，比如说 C++和 JavaScript 这样的语言，是靠标准化委员会的坚强毅力来管理的，特别是对 C++来说更是如此。这显然是一项不可思议的、令人印象深刻的工作。同时，它花费了这么多的人力来编写一个精确的规范，我希望能对 Python 也这么干，但是从未真正地实现过。

我们所拥有的是：对语言应该如何工作有着足够的理解；足够的单元测试；而且有足够的人们，能够在有限的时间内回答其他版本实现者的问题。例如，我知道，IronPython 那些人已经非常认真地尝试运行整个 Python 测试套件，并对每一个故障来判断测试套件是否真正地测试了 C Python 实现的特定行为，或者它们是否确实需要在实现中完成更多的工作。

PyPy 那些人干的也是同样的事，而且他们还更近一步。他们有些人比我更加聪明，而且有些人还提出了一种边界情况，这很可能是他们受到考虑如何在 JIT 环境中生成代码和分析代码的启发。实际上，在他们发现了一种特定的组合时，推动了相当多的测试、消除了很多歧义，并回答了很多问题，而这个组合没有人曾经真正在 JIT 环境中考虑过。那是非常有用的。一种语言有多种实现，这个过程对语言已形成无歧义的语言规范来说非常有帮助。

你能预测一下，C Python 可能什么时候不再是主流实现？

Guido：这很难预测。我的意思是说，有些人预测.NET 会统治世界；而其他人则预测 JVM 会统治世界。对于我来说，所有这一切看起来都像是主观愿望而已。但是，我也不知道会发生什么。也可能会出现一个巨大突破，即使我们所熟知的计算机实际上并没有什么改变，一种不同类型的平台突然变得更加流行，而且规则也随之改变。

或许它会远离冯诺依曼体系结构？

Guido：我还没有考虑过这个问题，不过，毫无疑问，那也是一种可能。我考虑更多的是：如果移动电话变成无处不在的计算设备时，该怎么办。不出几年，移动电话就会赶上普通便携式计算机的功能，这就暗示着，在几年之内，移动电话除了键盘和屏幕略小之外，它会具有足够的计算能力，以致于你再也不需要一台便携式计算机了。结果很可能是，无论是什么平台的移动电话，最后都会有一个 JVM，或者是其他的标准环境，其中，C Python 并不是最佳的方式，而其他一些 Python 实现的表现则会更好。

这里肯定还有另外一个问题：当我们在一个芯片上拥有 64 核时（甚至是在便携式计算机或者蜂窝电话上），我们应该怎么办。实际上，我并不知道，对于我们所干的大多数事情，是否应该修改编程范型。对于让你指定不可思议的细微并行进程的语言来说，它可能会有些用处，不过在大多数情况下，普通的程序员无论如何也无法编写出正确的线程安全代码。假定多核的增长以某种方式强制它们去实现那个目的，那是不现实的。我认为，多核肯定是有用的，不过它们会被用于粗粒度的并行，这样会更好一些，因为缓存命中总数和缓存不命中总数之间存在着巨大的成本差异，主存实际上已不再承担共享内存的功能。你应该将你的进程尽可能地相互隔离。

我们应该如何处理并发？应该在什么层面上处理这个问题，甚至更进一步，应该在什么层面上解决这个问题？ 37

Guido：我感觉编写单线程代码很难，而编写多线程代码更难；困难到大多数人（包括我本人）都认为它不会正常工作的地步。因此，我认为细粒度的同步原始类型和共享内存并不能解决问题——取而代之，我更愿意认为消息传递解决方案会取得成功。我相当确信：修改所有的编程语言来添加同步结构，这实在是一个馊主意。

我仍然认为试图从 CPython 中删除 GIL 不会起到什么作用。我认为支持多个进程的管理（相对于线程）确实是一个难题，而且，为此 Python 2.6 和 3.0 会有一个新的标准库模块，即多重处理，它提供了一个 API，类似于线程模块中具有同等功能的 API。作为一个附带的优点，它甚至支持在不同的主机上运行的进程！

2.4 权宜之计和经验

编写软件时您觉得缺少什么工具或者特性吗？

Guido：如果在计算机上画草图能像用铅笔和纸画那样容易，我就可能在为设计而苦思冥想时，画出更多的草图。我想这恐怕必须要等到鼠标全部被可在屏幕上画画的笔（或者手指）取代时为止。就我个人而言，我觉得所有的计算机绘图工具都非常弱智，虽然我已经非常擅长用铅笔和纸画图了——或许，我是从我父亲那儿继承了这一点，他是一位建筑师并且总是没完没了地画草图，因此，我从小就会画图。

从另一个角度来说，我想我甚至可能不知道对于探究大代码库来说，我缺少点什么。Java 程序员现在可以使用 IDE，对于“这个方法的调用者在哪？”或者“这个变量分配到何处？”这类问题，它能够很快给出答案。对于大型 Python 程序，这也是很有用的，不过由于 Python 本质上的动态性，因此，要进行必要的静态分析会更加困难。

你如何测试和调试代码呢？

Guido：我是怎么方便就怎么来。我在编写代码时进行了许多测试，不过测试方法会因项目而异。你在编写基本的纯算法代码时，单元测试通常很重要，不过在编写高度交互式的代码，或者是编写连接遗留 API 的接口时，我通常最后会借助于 shell 中的命令行历史或者浏览器中的页面重新载入，进行许多人工测试。举个（极端的）例子，如果一个脚本的唯一作用是关闭目前的机器，你就编不出很好的脚本单元测试来；当然，你可以模拟实际上关闭的部分，不过你还必须对其进行测试，否则你怎么能知道脚本是不是真的能起作用呢？

在不同的环境下，通常也很难实现测试自动化。对于大系统来说，BuildBot（译注7）非常好，不过使用它的开支也很庞大，因此，对于较小的系统来说，你通常只会采取一些人工的质量保证（QA）手段了事。在质量保证方面，我的直觉相当出色，不幸的是，它只可意会不可言传。 38

译注7：BuildBot 是用 Python 编写的一个自动构建工具。该 Python 程序只依赖 Python 环境和 Twisted（一个 Python 网络框架），可以在很多平台运行。每当代码有改变、服务器要求不同平台上的客户端立即进行代码构建和测试，收集并报告不同平台的构建和测试结果。

应该什么时候教他们调试呢？如何调试？

Guido：应该连续不断地调试。你在整个生命期间一直都在调试。我有一个六岁大的儿子，我正好“调试”过它的木火车的一个问题，那个木火车老是在轨道的某个特定点上出轨。调试通常是要降低一到两个抽象层次，而且，停下来仔细查看、思考和（有时）使用正确的工具，也会很有帮助。

我认为在某个具体点上，并没有什么单一的“合适的”调试方式可教，即使是针对一个非常特殊的目标，比如说调试程序 bug 等。导致程序出现 bug 的可能因素，其范围大得不可思议，包括简单的打字错误、“thinkos”、底层抽象的隐含约束，以及抽象或者其实现中的明显 bug。什么是正确的方法会因具体情况而异。当所要求的分析（“仔细查看”）单调乏味而且有很多重复时，这时候工具大多会大显身手。我注意到，因为搜索空间（被调试的程序）非常小，所以 Python 程序员通常只需要很少的工具。

您是如何重新开始编程的？

Guido：实际上，这是一个很有趣的问题。我记得并没有刻意考虑过我是如何做的，而我确实一直要面临这样的问题。我为此用过最多的工具很可能就是版本控制：在我回到一个项目时，我会区分工作区和仓库这二者，并告诉自己所处的状态。

如果有机会，在我知道将要被打断时，我会在未完成的代码中做一个 XXX 标记，告诉我具体的子任务。有时候，我也会使用 25 年前跟 Lambert Meertens 学到的办法：在当前源文件中的光标处，做一个特殊标志。为了纪念他，我使用的标志是“HIRO”。这是“here”一词的荷兰文口语，选中它的原因是，它和完成的代码中曾经出现的标志都不一样。☺

在 Google，我们还使用与 Perforce（译注8）集成在一起的一些工具，它们能在很早的阶段帮助我：当我开始工作时，可以执行一个命令，列出我的工作区中尚未完成的每一个工程，从而提醒我前一天在做哪个工程。我还坚持写日记，并会随手在日记中记下难记的特殊字符串（比如说 shell 命令或 URL），这些字符串有助于我执行手头项目的具体任务——例如，链接服务器状态页面的整个 URL，或者是用来重构我正在做的组件的 shell 命令。

对于接口或者 API 设计，您有什么建议吗？

Guido：这是另外一个领域，至于说什么是最佳流程，我还没有花多少心思来专门考虑这个问题，尽管我已经设计了大量的接口（或者 API）。我想在这儿借用一下 Josh Bloch 关于 API 的谈话；虽然他说的是 Java API 的设计，不过，他的谈话内容大部分都适用于所有语言。另外，他还有很多基本的建议，比如说选用清晰的名称（类用名词、方法用动词）、不用缩写、命名一致，还有在组合时提供灵活性最大的一个简单的小方法集等。他喜欢短小的参数列表：通常你最多可以使用两到三个参数，而不会造成次序混乱。最糟糕的是若干个连续参数都是使用同一类型；这样的话，如果偶然出现错位，很有可能会长期无人发现。

我个人有一些小毛病：首先，特别是对于动态语言而言，没有让某个参数值来决定对应方法的返回类型；否则，如果你不知道它们之间关系的话，可能很难理解返回的是什么类型：或许，确定类型的参数是从某个变量传入的，而这个变量的内容读代码时不容易读出来。

其次，我不喜欢设计用来彻底改变某种方法的行为的“标记”参数。使用这样的 API，在实际观测的参数列表中，该标记始终是一个常量，而且，如果 API 具有独立的方法：每个标记值都对应一种方法的话，它的名

译注8：Perforce 是美国 Perforce 软件公司生产的软件配置产品，可以实现跨平台版本控制。其特点是易用性强，速度快。

字可读性应该更强。

另一个毛病是回避这样的 API：会对它们是否返回一个新对象或者就地修改某个对象造成混乱。这就是在 Python 中列表方法 `sort()` 为什么不返回一个值的原因：这表明它就地修改了列表。内置的 `sorted()` 函数也是一种可选方案，它会返回一个新的分类列表。

应用程序的程序员应该信奉“简约成就非凡（less is more，简单就是美）”的哲学吗？他们应该如何简化用户接口，从而更便于学习？

Guido：谈到图形化用户接口时，看起来我的“简约成就非凡（less is more，简单就是美）”的观点最终得到了越来越多的支持。Mozilla 基金聘用了已故的 Jef Raskin（最初的 Macintosh UI 的共同设计者）的儿子 Aza Raskin 来设计 UI。Firefox 3 至少有这样一个 UI 的例子，它的功能非常强大，却不需要按钮、配置和属性等；地址栏监视我的输入，并把它与我以前浏览过的内容进行比较，最后会给出有用的建议。如果我对这个建议置之不理，它会设法把我输入的内容解释成为一个 URL，如果失败的话，则把它解释成为一个 Google 查询项。这就是智能！而且，它取代了三、四种功能，要不然的话就需要使用独立的按钮或者菜单选项。

这正好应验了 Jef 和 Aza 说了这么多年的一段话：键盘是一种功能强大的输入设备，我们应该以创新的方式使用它，而不是强制用户事事都靠鼠标，因为鼠标是最慢的输入设备。它的妙处在于不需要新的硬件，这一点不同于其他人建议的科幻解决方案，比如说虚拟现实头盔或者眼部运动传感器等，更不用说脑电波检测仪了。

当然，这还有许多工作要做—例如，Firefox 的 Preference（属性设置）对话框很难看，感觉一切都是脱胎于 Microsoft，至少有两层标签和很多模式对话框隐藏在毫不起眼的位置。为什么要我记住：为了关闭 JavaScript，必须转到 Content（内容）标签？Cookies 是在 Privacy（私密性）标签之下，还是 Security（安全性）标签之下？或许 Firefox 4 可以使用某种“智能”特性来取代 Preference（属性设置）对话框，这种特性允许你键入关键字，如果我键入“pass”，它就会直接转到配置密码（password）的那个部分。

对于今天和可预见的未来开发计算机系统的人们，您在创新、进一步开发和采用您的语言方面，有什么经验？

Guido：关于这一点，我有一两个小的想法。我不是哲学型的思维方式，因此这不是我喜欢的问题，或者是准备对此做出回应的问题，不过，我早就认识到我在 Python（以及 Python 的前身 ABC 没有做的、对其有害的事情）上做的对。系统应该由它的用户来扩展。而且，一个大系统应该在两个（或者更多的）层面上进行扩展。

自从 Python 第一次向公众发布以来，我陆续接到一些修改语言的请求，用来支持某种特定的用例。我对这种请求的第一个反应总是建议编写一些 Python 代码来满足他们的需要，并把它放入一个模块供他们自己使用。这是第一个层面的可扩展性：如果它的功能足够有用，就可以写入标准库。

第二个层面的可扩展性是使用 C 语言（或者 C++或者其他语言）编写一个扩展模块。扩展模块可以完成纯 Python 无法完成的特定功能（尽管这些年来，纯 Python 的功能已经有了很大提升）。我更想添加一个 C 语言级别的 API，这样一来，扩展模块就可以在 Python 中的内部数据结构中修改，而不需要修改语言自身，因为修改语言会坚持兼容性、质量和语义清晰度的最高标准。而且，人们通过在自己的解释器副本中修改语言实现“自给自足”时，可能会出现该语言的“分支”，他们也会把它分发给其他人。这样的分支会带来各种各样的问题，比如，随着核心语言的发展还得维护专用的修改，或者是融合其他用户可能需要组合的多种独立的分支版本等。扩展模块并没有这些问题；实际上 C API 已经提供了扩展所需的大多数功能，因此，很少需要为了某个特定的扩展而修改 C API。

另一个想法是去接受不会一次成功的观点。在开发的早期阶段，已经有少数的早期试用者成为了用户，此时只要你发现了一个问题，就必须彻底解决，而不必担心向下兼容性。我常常喜欢引用的一大奇闻轶事（而且已经当时的目击者证实）是：Unix v7 中“Make”的原始作者 Stuart Feldman，曾被要求修改 Makefile 句法与硬 tab 符的依存关系。他的回答思路是：他承认 tab 是一个问题，不过，因为有了十多位用户，要解决这些问题已经为时太晚了。

随着用户基数的增长，你需要更加稳健（保守）一些，而且在某些方面，无条件地向下兼容也是绝对必要的。因为，多年以来，这种语言已经积累了很多不再可用的不良特性。我在 Python 3.0 中采用的办法，堪称处理此类问题的良策：宣布某个特定版本不再提供向下兼容性，要利用这个机会尽可能多地解决这样的问题，并给用户社区提供足够的时间来实现过渡。

具体到 Python 语言上，我们打算支持 Python 2.6 与 Python 3.0 长期共存——这个时间要远远长于老版本惯常的支持周期。我们还提供了几种过渡策略：一个远非完美的源代码到源代码自动转换工具；与 2.6 版本中将在 3.0 中修改的功能用法的可选警告相结合（特别是如果转换工具无法正确地判断情况时）；以及，将特定的 3.0 特性选择性地向后移植到 2.6。同时，我们也不是完全重新编写或者重新设计 Pytho 3.0（不同于 Perl 6，或者 Python 世界中的 Zope 3），这样一来就可以将因意外而减弱基本功能的风险降到最低程度。

在过去的四五年中，我注意到这样一种趋势：许多更大的公司采用了动态语言。首选是 PHP，有些时候是用 Ruby，而有些时候则绝对是用 Python，特别是在 Google 中。这对我来说非常有趣。我想知道 20 年前，像 Tcl、Perl 以及稍晚一点的 Python 这样的语言在做所有这些有用的事情时，这些人都在哪里呢？你听到让这些语言更加企业友好的（无论它意味着什么）呼声了吗？

Guido：企业友好通常会是这样一种情景：真正聪明的人们失去了兴趣，同时，技术更一般的人们必须以某种方式自己谋生。我不知道对于普通人来说 Python 是不是更难使用。在某种意义上，你会认为在 Python 中你不会造成什么破坏，因为它全部是解释型的。另一方面，如果你编写很大的程序，而且你并没有进行足够的单元测试，它的性能到底如何，你可能并没有什么把握。

你已经制造了一种争议：如果 Python、Ruby、Perl、PHP 代码要写 1 行，那么 Java 代码可能要写 10 行。

Guido：通常是这样的。我认为企业界的采用水平，即使是在有有用的特定功能包的情况下，也很可能会顾虑非常谨慎的管理者。设想一下，一个人负责一家有 100 000 名员工的公司的 IT 资源，而 IT 并不是它的主要产品：它们可能在生产汽车，或者是从事保险业等，不过无论它们做什么都要用到计算机。负责基础设施的人必须非常谨慎。他们会追随大牌公司好像在使用的那些东西，比如说 Sun 或者 Microsoft，因为他们知道 Sun 和 Microsoft 一直在搅局，而这些公司不得不从一团乱麻中恢复过来并使之稳定，即便要花掉五年的时间。

传统上，开放源代码项目还不能让普通的 CIO（首席信息官）感到同样心安。我确实不知道这能不能、怎么样和什么时候会改变。如果 Microsoft 或者 Sun 突然在它们各自的 VM 上支持 Python，实际上，企业的程序员很可能会发现，使用更先进的语言，他们可以获得更高的生产率，而不会有任何不利的影响。

第 3 章

APL

20 世纪 50 年代后期，执教哈佛大学的 Kenneth Iverson 为精确描述算法而设计了一种拓展的数学符号集。之后，他与 Adin Falkoff 和 IBM 的其他研究者组成的团队，逐渐将该符号集转变成一种成熟的编程语言——APL。尽管 APL 语言须使用专用键盘录入，而且它的字符集有时候人们并不熟悉，但其底层的一致性却使它非常易于学习，而且 APL 无与伦比的数组处理能力使其功能非常强大。它的衍生语言 J & K 秉承了这种理念，继承了 APL 的简明性和代数运算功能强大的特点。

3.1 用纸和笔设计

我读过你和 Ken Iversona 合写的一篇论文："APL 设计（The Design of APL）"。这篇论文提到，在开发 APL 开始的七八年中，没有涉及任何有关计算机的工作。这使您在修改设计时不必担心任何遗留问题。APL 语言的第 1 版软件实现对其发展有何影响？

Adin Falkoff：没错，APL 刚开始起步的时候，它除了被称为"Iverson 的符号"之外，甚至都没有名字。那时，我们主要是关注纸笔式的书面数学应用程序、分析数字系统和教学。在很大程度上，我们认为编程是数学的一个分支，它主要关心的是发现和设计算法，这个观念也被符号表示法所支持。不久，这些符号作为一种通用的编程语言的吸引力变得非常明显，这也因不同的人们（特别是 IBM 的 Herb Hellerman）的努力而向前发展，这些人利用机器实现对该符号集的重要元素进行了实验，包括基本函数和数组运算。不过，在那段时期我们确实是完全自由地设计语言，而不用去考虑任何"遗留"问题。

APL 在早期最重要的发展分为两大步骤。第一步是编写和出版《System 360 的形式化描述（The Formal Description of System 360）》[IBM 系统杂志（IBM Systems Journal），1964]。为了形式化地描述这个新设计出的计算系统的某些行为，在 Iverson 的《一种编程语言（A Programming Language）》[Wiley]一书中提到的对该符号集的增改也是非常必要的。第二步是为基于 Selectric（字球式电动打字机，原商标名）的终端的录入设备设计，我们预期利用这样的设备在机器上使用这种语言。这样一来，就强加了源于线性打字方式的重要限制，也带来了基于 Selectric 的机械装置的强制需求。我相信，在你所说的"APL 设计（The Design of APL）[IBM 研发杂志（IBM Journal of Research and Development），1974]"一文中，非常详细地谈到了这两种因素对于该语言发展演进的影响。

当然，APL 语言的第一个综合性实现是 APL\360。它引入了编写定义函数（也就是程序）的一些必要的工具，这些工具在使用纸笔方式时是必备的，同时也是为了控制程序将要执行的环境。然后，引入了包括工作区、库系统、作用域范围名称规则，以及如何使用共享变量与其他系统通信等概念，这些概念至今都没有什么显著的变化。为 APL\360 编写的程序，不经任何修改，就可以在现在的 APL 系统中运行。

公平地说，通过严格施行"新概念必须向下包含"的原则，对"该语言如何应用于新的和不同的应用程序"进行持续不断的必要检查，这个原则的出现影响了该语言的进一步发展。

在定义句法时，您是如何想象典型的 APL 程序员的呢？

Adin：我们在考虑句法问题时，没有针对程序员，而是想到这种语言会成为人与人之间、人与机器之间的交互平台。我们确实意识到：如果拥有像代数一样的符号语言，用户不但会感觉到更加适应，而且会喜欢这种符号表示法，因为它更便于表达式的形式化操作，从而能够更加有效地分析和合成算法。特别是，我们确实相信，它并不需要很多的数学知识与经验，事实上，使用 APL 系统在小学、高中层次进行教学都是相当成功的。

随着时间的推移，我们发现一些专业的、富有经验的程序员被 APL 所吸引，并使用 APL，而且对 APL 的发展做出了许多贡献。

句法的复杂性会限制 APL 的传播吗？

Adin：尽管我不同意说它是“复杂的”，但 APL 的句法规则及其对被接受程度的影响是很值得讨论的。APL 是以数学符号和代数表达式为基础，通过去除不规则格式和推广可接受的符号实现了规则化。例如，很显然，像加法和乘法等二元函数符号应该写在两个参数中间，而一元函数都应该把函数符号写在参数之前，而没有传统的数学符号中的那些例外。因此，APL 中的绝对值符号为参数之前有一道竖线，而不是两侧都有竖线；还有，APL 的阶乘符号是在参数之前而不是在参数之后。在这些方面，APL 句法要比它的历史渊源（也就是传统的数学语法）更简单。

APL 的句法比其他代数符号和其他编程语言更简单的另一个重要方面是，APL 中表达式赋值的优先级规则非常简单，所有的函数都具有相同的优先级，这样一来，用户不用再去记求幂运算是否应该在乘法之前，或者定义的函数属于哪个优先层次。APL 的优先级规则很简单：最右边的子表达式将被第一个赋值。

因此，我认为 APL 的句法并不会限制 APL 的传播。反而是字符集里面使用了很多在标准键盘上难以找到的非字母符号，很可能会对其有一定的限制。

您为何决定使用一个特殊的字符集？这个字符集是如何发展起来的？

Adin：这个字符集是使用传统的数学符号来定义的，还加入了一些希腊字母和一些像空铅（quad）之类的直观符号。

直线式打字机的局限也带来了一定的实际影响，它导致创造出了一些通过加粗产生的新字符。后来，随着终端和输入设备的功能变得越来越强大，这些复合字符凭借自身的资格都变成了基本符号，同时，为了适应新设备，APL 还少量引入了一些新符号，比如用做语句分隔符的菱形符号。

有过“让有限的时间资源发挥更大的效率”这种刻意的决定吗？

Adin：当时，字符集的确受到了优化使用可用的有限资源的影响，不过，又发展并维持了简单的符号化格式，因为确信它有利于分析和表达式的形式化操作。而且，比起其他的语言，一旦努力用简明的 APL 表示法来读取 APL 程序，这个程序的逻辑流就更便于人们理解。

我认为人们需要许多培训来学习这门语言，特别是字符集部分。有没有一种自然选择的过程，它意味着 APL 程序员熟练掌握该语言？他们生产率更高吗？他们能写出 bug 更少的高质量代码吗？

Adin：为达到像 FORTRAN 那种水平的编程能力而学习 APL，这并不难，而且也不会花很长时间。由于规则简单，拥有像排序这样的数据操作等基本函数、矩阵转置等数学函数，因此，使用 APL 编程生产率更高。这些因素使得 APL 编程十分简明扼要，并因此而更易于分析和调试。APL 实现使用了带有所有有用属性的工作区以及基于交互式终端的解释型系统，它也为生产率做出了很大贡献。

对于带有小屏幕的设备（比如 PDA 或者智能手机）来说，超级简明的表达式形式的用处十分巨大。考虑到 APL 是第一个在像 IBM System/360 这样的大型计算机主机上编码的语言，它能够扩展用于处理需要管理网络连接和多媒体数据的现代项目吗？

Adin：APL 在掌上设备的一种实现至少可以提供强大的手动计算功能；我认为在网络和多媒体方面没有问题，因为在 APL 系统中管理这些应用已经有很长时间了。现代的 APL 上已经广泛地使用了 GUI 的管理工具。

在 APL 系统发展的早期阶段，引入了许多管理 APL 功能内部的主机操作系统和硬件的工具，这些工具被 APL 系统程序员用来管理 APL 自身的性能。而且，为经济生存能力而依赖网络的商业 APL 分时系统也使用 APL 来管理它们的网络。

47

第一个可用的商业 APL 是在大型机上编码的，而证明了 APL 系统可用性的最早实现是在相对小一些的机器上完成的，比如 IBM 1620、IBM 1130 系列，还包括 IBM 1500，它在教育领域有着重要的应用。甚至在前期的实验性台式机上也有一种实现，这种机器绰号“LC”取低成本（low cost）之意，它只有几个字节的内存和一块低容量磁盘。APL 实现的发展演进的具体细节，详见论文《APL 系统的 IBM 系列（The IBM Family of APL Systems）》[IBM 系统杂志（IBM Systems Journal），1991]。

3.2 基本原理

Operating Principles

您在追求标准化，这是一个慎重的决策吗？

Adin：毫无疑问，我们的标准化工作起步相当早；我曾经为它撰写了一篇论文，而且开始成为 ISO 的一部分。我们一直想实现标准化，为此花了相当大的精力。我们劝阻人们不要在语言的基本结构上浪费时间，不要随意添加使句法复杂化的元素，或者是违背我们试图维护的基本法则。

您认为标准化的主要需求是什么？是兼容性还是概念纯洁性？

Adin：标准化的需求是一个经济问题。我们想要 APL 在经济上可行，而且由于很多不同的人在实现并使用它，有一个标准确实很必要。

一些不同的厂商拥有不同的 APL 编译器。如果没有强大的标准化，APL 扩展在某一个系统可用，而在另一个系统上不可用，会出现什么问题？

Adin：那是 APL 标准化委员会要小心处理的工作，而且他们正努力在可扩展性和纯洁性之间进行折衷处理。

您希望人们能解决未预料到的问题，但你并不想要他们摒弃 APL 系统的基本性质。40 年之后，APL 系统将会变成什么样子？您选择的设计原则还会依旧适用吗？

Adin：我想是这样的。我确实没看到有任何问题。

为什么？是因为你花了很多时间进行小心谨慎的设计，还是因为现在的 APL 有非常强大的代数理论背景？

Adin：我认为我们这几个人是相当睿智的，我们秉承着简单、实用的理念，而且不愿意就这种理想进行妥协。

我发现学习和记忆其他语言的所有规则十分困难，因此尽量保持它的简明性，以便于使用。

48

我们的一些想法在论文中有所展示，特别是我和 Iverson 合写的那些论文。我后来自己撰写了一篇论文：《模式匹配：空数组如何匹配？（A Note on Pattern Matching: Where do you find the match to an empty array?）》[APL Quote Quad，1979]，这篇论文使用了一些很好的推理方法（包括小程序和代数原理），用来获取报导的结果，它被证实是一致的，而且是有用的。该论文还考虑了不同的可能性，而且发现最简单的表示效果最好。

我发现，利用小的原理集来构建一种语言，以及在这些原理上发现新想法，确实是魅力无穷。这看起来好像是对数学的很好的描述。数学在计算机科学和编程中发挥什么作用呢？

Adin：我相信计算机科学是数学的一个分支。

数学计算编程显然是数学的一部分，特别是维护离散数字运算和理论连续性分析之间的兼容性的数值分析。

我想：推动力来自于只有通过扩展计算才能解决的数学问题，扩展计算提出了速度要求；数学所需的逻辑思维能力训练可以延续到所有的编程当中；算法的概念，它是一种典型的数学工具；而不同的专门数学分支，比如拓扑学，则有助于分析计算问题。

我阅读过其他的一些讨论，其中，你和别人一起建议的最有趣的应用之一是在中小学中使用 APL 进行编程和数学教学。

Adin：确实是的，虽然这还处于起步阶段，但我们对此十分感兴趣。

那时，我们只有一些打字机终端，并提供给一些本地私立学校使用。特别是，还要想好应该教给学生们什么东西。而且，要让他们在打字机上练习，并不受拘束。

有趣的是，我们发现有些原本被认为反感学习的学生，却在学校为此多花好几个小时，因此他们能学得更多。他们使用与我们的分时系统相连的打字机终端。

因此，他们从此突然爱上这种（以后甚至是必须的）学习？

Adin：是的。

您使用 APL 向非程序员教授“编程思想”。APL 为什么会吸引非程序员呢？

Adin：在早期阶段，有一个好处就是你花费的开销少。在你对两个数进行加法运算之前，不必声明。如果你想要对 7 和 5 进行加法运算，只需写下 7 + 5 即可；而不用说这里有一个数是 7，一个数是 5，它们都是数字，再说清楚是不是浮点数，而且运算结果也是一个数，需要把结果存储在这里。因此，使用 APL 来满足你需求的门槛较低。

人们在学习程序时，最开始要做的事是很少的。基本上只需要将想要做的事写下来，而不必花费时间用编译器编译它，是这样吗？

Adin：没错。

这样很容易学习，而且也易于使用。这种技术会让更多的人变成程序员，或者是增加他们的编程知识吗？

Adin：易访问的特点使它易于实验，而且如果你可以实验并测试不同的事情，你就会学到东西，而且我认为这有利于提高编程技术。

您为 APL 选择的符号为什么不同于传统的代数符号？

Adin：哦，差别并不是那么大。只是优先级规则不同而已。它们非常简单，只需从右到左进行计算。

您认为它更易于教授吗？

Adin：没错，因为它只有唯一的规则。而且你也不必说明这是一个已定义的函数，你就可以开始了，而且如果它是求幂运算，它的优先级要高于乘法或者其他的运算。你只需要说，“看看指令行，从右到左进行就行了”。

这种深思熟虑的设计决策是为了与熟悉的符号和优先级相决裂，而最终实现更大的简单性吗？

Adin：没错。更简单和更通用。

我认为是 Iverson 主要负责那部分的工作。他在代数方面非常优秀，而且他对教学非常感兴趣。他喜欢使用的一个例子是多项式的表示，它使用 APL 来运算时极其简单。

我第一次见到那个运算符号时，尽管我对它并不熟悉，但我也觉得它十分简单。您如何看待设计或者实现中的简单性？您是靠良好的风格或者经验，还是通过严格的步骤来寻找最佳简单性的？

Adin：我认为在某种程度上它必定是主观的，因为它多少有点取决于你的经验和背景。我想说，通常是规则越少就越简单。

您是从一个小公理集开始，并且从那里开始构建系统的，不过，如果你理解了这个小公理集，您可以获得更多的复杂性吗？

Adin：哦，让我们以优先级为例说明。我认为，基于简单的从右到左的优先级方式，要比一个需要利用优先级表的方式简单，而这个优先级表得说明各个函数运算的先后关系。我认为，这个问题是一个规则和一个几乎是无限规则集的对比。

你知道，你可以在任何特定的应用程序中建立自己的变量集和函数集，而对于一个特定的应用程序来说，你可能会发现编写一些新规则更为简单，不过如果对 APL 这样的通用语言，你肯定会想以最少的规则开始学习。

您会给使用这种语言来设计系统的人们更多的发展机会吗？

Adin：事实上，构建应用系统的人就在构建语言；从根本上说，编程就是跟发展适合于特定应用程序的语言有关。

你只是在使用编程语言描述特定领域的问题而已。

Adin：不过，那些目标，尤其是名词、动词、对象和函数，它们必须使用某种方式来明确定义，例如使用 APL 这样的通用语言。

因此，你使用 APL 来定义这些，然后建立运算来推动你想要在应用程序中做的事情。

您关心建立构建块吗，人们可用它来表达自己？

Adin：我所关心的是，如果你愿意，可以给用户提供基本的构建块，以及建立构建块的基本工具，这些工具便于他们满足自己的需求。

这似乎是所有语言设计者共有的想法：我想起了发明 Forth 语言的 Chuck Moore、实现 Lisp 语言的 John McCarthy，还有 20 世纪 70 年代早期的 Smalltalkin。

Adin：我想也是。

我知道 McCarthy 是一个理论家，他主要关注研发改进 lambda（λ）演算效率的系统，不过我认为，对于大多数应用来讲，lambda（λ）演算并没有简单的一般代数那样方便，而 APL 就是起源于这种代数运算。

假设我想要设计一种新的编程语言。您能给我的最佳建议是什么？

Adin：我想我可以说的就是做你喜欢的事情、那些使你愉快地工作的事情，还有那些能帮助你实现目标的事情。

我们总是在方法上非常个人化，而且在听到人们的想法时，大多数设计者都是这样的。他们就是从自己喜欢干的事开始，结果证明这种方法通常是很有用的。

您在设计 APL 时，能否在某些时候认识到“我们现在的方向不对；我们需要减少这种复杂性”或者“我们有几种不同的解决方案；我们能把它统一到更简单的方案上来”？

Adin：基本上是这样的，但通常会有一个问题：“这是一种已经包括了已有的东西的归纳概括吗？而且它能让我们做更多的事而遇到更少的难题，这种可能性有多大？”

我们非常注意边界条件，例如，你从 6 到 5 到 4 再到 0，就会出现边界问题。这样，在减法中，假如说你在使用一个向量加法函数，而且如果你对一个有 n 个元素的向量进行加法运算，然后 n 次减去一个元素等，当你最终没有元素了会出现什么情况呢？和是什么？结果只能是 0，因为它是一个单位元素（译注1）。

还有乘法的例子，对于空向量的乘法运算，我们将返回该函数的单位元素 1。

您提到几种不同的解决方案，并尝试进行归纳概括，并问自己一些问题，比如，当接近 0 时会出现什么情况。如果您已经知道在做减法时，如果 n 变成 0 时，你需要把结果赋为单位元素。你可以查看这些例子，并且说“我们这里有一些争论：在这种情况下是 0。那种情况下是 1，因为它是单位元素”。

Adin：没错。这就是我们使用的方法之一。

特殊情况下会出现什么情况是非常重要的，而且你高效地使用 APL 时，保持将那些准则应用到更复杂的函数上，此时你可能是在开发一个特定的应用。这个通常会带来意外的却又令人满意的简化。

要向可能会使用该语言编程的开发人员通告您在创建这种语言时使用的设计技术吗？

Adin：是的，因为如我此前所言，编程是一个设计语言的过程。我认为这是一个非常根本的问题，据我所知，这一点在文献中很少提及。

译注1：单位元素（identity element，幺元），是指与其他数字进行运算后而此数字不被改变的数字元素。例如，0 在加法运算中是任何实数的单位元素，因为假设 a 是任意实数，a 加上 0 等于 0 加上 a 还是等于 a。同样，1 在乘法运算中是任意实数的单位元素，因为 $a \times 1 = 1 \times a = a$。

Lisp 程序员可能会认为如此，不过对许多后来的语言，特别是 Algol 和 C 语言系列，人们好像并不这样想。在是否内置于语言之中，而其他所有的一切都是第二类的，这二者会有区别吗？

Adin：噢，所谓第二类是什么意思？在 API 中，所谓第二类和第一类遵从相同的规则，而且没有任何问题。

你在 Lisp、Scheme 或者 Smalltalk 这些语言中都会使用同样的参数，不过，C 语言在操作符、函数、用户定义的函数之间有一个严格的区分。对这些实体进行严格的区分，这是设计的错误吗？

Adin：我不知道是否应该把它叫做一个错误，不过我认为，无论是不是基本类型，都适用同样的规则，这样会简单一些。

您在设计和编程中所犯的最大错误是什么？您从中吸取了什么教训呢？

52

Adin：在设计 APL 的起步阶段，我们有意识地避免做出迎合计算机环境的设计决定。例如，如果机器能够在数据对象输入或者生成时，很容易地根据对象自身来确定数据对象的大小和类型，我们就会避开使用声明，把它们的使用看作加在用户身上的不必要负担。不过，随着时间的推移，总有一天 APL 的应用会越来越广泛，而且会产生越来越多的既得利益，硬件的因素变得无法避免。

我个人所犯的最大错误可能是低估了硬件的发展速度，在系统设计中显得过于保守。例如，在早期考虑在 PC 上实现 APL 时，曾经主张不考虑对普通数组和复数的扩展，因为要提供令人满意的性能，这样会给现有的硬件带来很大的压力。幸运的是，我的方案被否决了，而且不久之后，PC 内存和处理器的发展速度就突飞猛进，使得这种强大的功能扩展完全可行。

很难想象出编程中所犯的大错误，因为一个人在编写具有合理复杂性的程序时通常都会犯错。它取决于错误的产生方式、它的发现时间以及解决这个错误问题而必须重做的工作等。模块化和惯用的复用代码段，比如 APL 所支持的函数式编程风格，往往会去限制错误的产生和传播，因此，这些错误就不会变成大错误。

至于说 APL 自身的设计错误，我们的开发方法包括使用设计者和实现者协商一致的结果作为最终决策因素，以及来自大量的应用用户反馈使用经验，还有在设计“固定”之前，我们自己对语言的使用，这些都有助于避免严重的错误。

不过，一个人的经验可能是另一个人眼中的错误，甚至会经过很长时间，也不可能根据经验来解决这些分歧。我们想到了两种解决思路。

一个是字符集。最早期阶段，选用保留字而不是抽象符号来表示基本类型函数，面临着相当大的压力。我们的立场是，我们是真正地处理数学的扩展，而且数学标记的发展方向很明显就是使用符号，这样有利于表达式的形式化操作。随后，长期致力于数学教育的 Ken Iverson，选择在未来的工作中把字符集限定在 ASCII 上，并且选定了 J 语言，这样，学生们和其他人就可以方便地使用 J 系统，而不需要专门的硬件。我自己还是倾向于符号方式并坚持使用，因为这更符合历史习惯，最终也便于阅读。时间会证明这个方向的正确与否，或者根本就无关紧要。

我们想到的第二个事就是，可能在从未明确的方向导致明显错误就是处理普通数组，也就是，该数组的标量元素可能自己在语言内部有一个可访问的结构。在 APL\360 被确定为 IBM 的产品（1966 年或者 1967 年，IBM 对软件和硬件分类计价时很早的一个），我们开始考虑更多的普通数组扩展，并开始深入研究和讨论其理论基础。最终，使用标量元素和句法重要性的竞争方式构建起了 APL 系统。随着并行编程在商业上变得更加重要，人们也对此更感兴趣，看一看这是如何发展演变的，也是非常有趣的事情。

53

3.3 并行

Parallelism

把数据看成一个集合而不是单独的个体，这（对于应用程序设计来说）意味着什么呢？

Adin：这是一个相当大的话题，正如在 FORTRAN 这样的语言中推广“数组语言”和引入数组基本类型所表明的那样，不过在考虑集合时，我认为这里有两个重要的方面。

当然，首先是我们希望思考过程的简单化，不要陷于处理个别问题的常规事务细节的泥淖之中。它非常接近我们考虑问题的自然方式，例如，这个集合中多少数等于零，只需编写一个简单的表达式就能产生想要的结果，而不是开始考虑使用任何形式的循环。

第二是在编程中使用并行方法的可能性更为明显，并行直接作用于集合上，因而可以更加高效地利用先进的硬件。

在现代编程语言方面有一些讨论，有些人希望向 C++或者 Java 之类的语言中添加高阶特性，在这些语言中，你要花很长时间反反复复地编写 `for()`循环语句。例如，我有一个事务集合，而我希望是一个一个来干。不过，APL 早在 40~45 年之前就已经解决了这个问题!

Adin：是的，我不知道具体是在多少年前，不过它有两个阶段。第一个阶段是使用数组作为基本类型，而第二阶段则是引入了称为 each 的操作符，它基本上可以将任意的函数应用到任意的集合项上。不过，始终存在一些问题，比如说，“我们真的要将这个基本类型放入特定的循环吗？”我们觉得我们不想那样做，因为它把句法搞得太过复杂了，而且使用标准的方式就足以容易地写出所需的几个循环。

将句法复杂化是针对实现还是针对用户？

Adin：二者都有，人们必须阅读它，机器也必须阅读它，无论句法简单与否。

你想要放入新的语句，而且这无疑是一个难题。现在的问题是：“这种代价值得吗？”这时候就须做设计决策了。而且我们总是不想使用新的句法来处理循环，因为对原来的方法已经轻车熟路了。

您说过 APL 有一个真正的优点就是并行编程。我也了解到使用数组作为该语言的数据结构。您还提到了共享变量的使用。这些是如何工作的？

Adin：APL 的共享变量是可以一次访问多个处理器的变量。这些共享的处理器既可以是 APL 处理器，也可以是其他类型的处理器。例如，你可以使用一个变量，我们称之为 X，而且对 APL 来说，读写 X 与读写普通变量并无不同。但是，可能有其他的处理器，比方说一个文件处理器，它也要访问 X，它就成为了一个共享变量，而且，无论 APL 为 X 赋什么值，文件处理器总是根据它自己的解释来使用这个值。同理，如果它给 X 赋了一个值，然后 APL 再来读取，那么，APL 同样也是根据自己的知识来使用它，无论它如何选择解释该值。这个 X 就是一个共享变量。

在 APL 系统中，比如说 IBM 公司的 APL2，我们有一些协议，用于管理这种变量的访问，由此你不会遇到各种竞争条件的麻烦。

这是您所说的编译器可以自行决定的并行化吗？假设我想让两个数组相乘，并把该值添加到一个数组中的每个元素中。这使用 APL 很容易表达，但编译器可以执行这种隐式并行化吗？

Adin：使用 APL 的定义，意味着你使用什么顺序对数组元素进行操作都无关紧要，因此，编译器、解释器或者任何实现都可以同步地或者随意地执行。

除了提供语言层面上的简单性之外，它还可以利用新的硬件优势，为实现者改变实现方式提供了极大的灵活性，或者是为你提供了一种开发机制，比如说开发自动并行化机制等。

Adin：对，因为根据 APL 定义，当然是使用处理器时发生什么的定义，按照什么顺序来完成工作对它并不重要。这是一个深思熟虑的决策。

这是在当时的语言的历史情况下做出的独一无二的决定吗？

Adin：我不熟悉语言的历史，但由于我们基本上是唯一专业的面向数组语言，所以，它可能是独一无二的。

谈论集合和大型数据集非常有趣，这些显然是现代程序员密切关注的问题。APL 早在关系型数据库发明之前就已经出现了。现在，我们在关系数据库以及大型非结构化集合（例如网页结构等）中放入了很多数据，这些数据都使用包含不同数据类型的结构。APL 能把它们处理好吗？它有没有提供模型，可供使用 SQL、PHP、Ruby 以及 Java 等比较流行的语言的人们借鉴呢？

Adin：APL 数组可以包含没有内部结构的标量元素，也可以包含任意复杂的非标量元素。非标量元素是其他数组的递归结构。因此，“非结构化”集合，比如说网页，可以方便地使用 APL 数组来表示，并通过 APL 基本函数来操作。

对于非常大的数组，APL 可以将其视为外部对象来处理。一旦在工作区中的一个名称和一个外部文件之间建立关联关系，就可以使用 APL 表达式将操作应用到文件上。在用户看来，这个文件就好像在工作区内一样，尽管它实际上可能要比工作区大好多倍。

很难具体说其他语言的设计者能从 APL 中学到什么，而且我也不好贸然评论你提到的其他语言的细节，因为我不是那些语言的专家。不过，我通过文献了解到，总的来说，APL 语言的设计指导原则（例如，我们在 1973 年发表的《APL 设计（The Design of APL）》一文中介绍过）已经在后来的其他语言设计中得到了传承。

在简单和实用这两项最重要的原则中，后者似乎表现很好，而简单则是较难实现的目标，因为对复杂性没有实际限制。我们通过仔细地定义它所允许的基本类型操作的范围，维护 APL 对象的抽象本质，并拒绝将其他系统的运算表示的特殊案例包括进来，努力争取保持 APL 的简单性。

这方面的一个例子是，在 APL 中没有出现“文件”的概念。我们可以将数组当作文件来处理，应用程序也可以称之为“文件”，但是，没有专门设计用于文件操作的基本函数。不过，早期对于文件管理效率的实际需要，已经培育了共享变量范例的发展，这种范例自身也是在大量应用程序中有用的通用概念，在这些程序中，APL 程序需要调用另一个（APL 或者非 APL）辅助处理器。

随后，使用命名空间的通用概念，设计了一种附加设备，它可以让 APL 程序直接操作工作区以外的对象，包括访问 Java 字段和方法、超大数据集、使用其他语言编写的汇编程序等。连接共享变量和命名空间设备的用户接口都严格遵从 APL 的句法和语义，从而保持了它的简单性。

因此，不用细述，可以合情合理地说，新的语言能够通过以下方式获益：维护一个严格遵守它们自己的基本

概念，尽可能地在它们关注的应用程序环境中逐一进行通用定义。

至于说 APL 作为一种模型的具体特点，APL 已经证明声明是不必要的，尽管它们在某些情况下可能有助于提高执行效率，而且不同数据类型的数目可能相当小，新语言可能会从瞄准这些目标中获益，而不是想当然地认为用户必须提供这种与实现有关的信息来帮助计算机。

另外，在 APL 中指针的概念也不是一个基本类型，而且也从未提及。当然，语言中可用的基本运算应该定义在使用抽象的内部结构的数据集合之上，比如正规数组、树等。

APL 先于关系数据库出现，这一点你说对了。20 世纪 60 年代，E. F. (Ted) Codd 博士和 APL 小组都在 IBM TJ Watson 研究中心工作，当时他正在开发关系数据库的概念，而且我相信我们对这项工作产生了非常强烈的影响。我清晰地记得，一天下午，我们进行了热烈的讨论，我们证明了：是简单的矩阵而不是复杂的标量指针系统可以用于表示数据实体之间的关系。

56

3.4 遗留

我知道 Perl 的设计受到了 APL 的很多影响。有人说 Perl 的一些奥秘来自于 APL。我不知道这是否是一种恭维。

Adin：让我给你举一个这种恭维的例子。特别是在 IBM 这样的地方，它是一家企业，编程语言的设计和使用受到很多约束。在不同的时期，人们常是进行竞争性实验来看一看 APL 是否比 PL1 或 FORTRAN 之类的语言更好。结果总是被歪曲，因为是对方当裁判，不过，我永远记得来自负责人一个评论，他说，APL 不可能太好，因为他认识的两个最聪明的人 Iverson 和 Falkoff，不可能让人们相信这一点。

关于创新、进一步的开发和采用你们的语言，您有什么经验和教训想要告诉那些现在和不久的将来开发计算机系统的人们吗？

Adin：系统设计决策不是纯粹的技术或科学。在语言和系统的设计中，经济和政治因素对它有着很强的影响力，尤其是基础性技术细节的潜在灵活性上。

在 20 世纪 60 年代中期，APL 作为 IBM 内部使用的一个重要工具，这时候有了把它做成产品的想法，我们不得不应付 IBM 的“语言沙皇”，他下命令公司未来只支持 PL/1，当然，FORTRAN 和 COBOL 例外，因为它们已经在行业内站稳了脚跟，而且不可能被完全抛弃。

历史表明，公司采取这种立场是不切实际的，而且注定要失败，但这在当时并不是那么明显，因为 IBM 在计算产业中占有主导地位，而且特定派别在公司占据了强势地位。

我们必须为 APL 的生存争取必要的政策支持。这项工作通过几条主线全面铺开：作为 IBM 研究部的成员，我们利用尽可能多的机会，提供专业讲座、研讨会和正式的课程，把 APL 的特色深植到当时的技术思维当中；我们得到（不惜代价要找到）公司内有影响力的人物的支持，以对抗管理阶层的权力，我们支持推广和支持内部使用我们的 APL\360 系统进行开发和制造；我们利用重要客户对 APL 的兴趣，加强来自公司外部的 APL 可用性证明，至少是在实验的基础上；与营销部门联盟。最后，我们取得了成功，在 20 世纪 60 年代末 IBM 公司计划对软件和硬件分类计价时，APL\360 位列第一批上市的 IBM 程序产品之中。

57

在美国航空航天局戈达德航天中心（NASA Goddard Space Center）技术讲座及展示所产生的效果是一个非常重要的里程碑。 1966 年该中心请求访问我们的内部 APL 系统，进行系统使用实验。他们是一个非常重要的客户，IBM 的营销人员敦促我们同意他们的请求。不过，我们表示反对，坚决要求首先要在戈达德航天中心现场进行为期一周的教学课程，我们才能同意这样做。

他们同意了我们的要求，但我们随后就遇到了实现的困难：当时，像 APL\360 这样的分时系统需要使用终端，而这种终端是通过与专用电话“数传机”联用的声频调制解调器连接到中央系统的，在另一端也使用了这些专用电话机，这些电话机连接到中央计算机上，一时间电话机奇货可居，供不应求。在所有的行政管理部门都同意该项目继续往下干的时候，我们发现，无论是在纽约和华盛顿特区电话公司都无法提供航天中心项目课程所需的设备。

虽然只使用自己的设备是华盛顿特区电话公司的一贯做法，但是他们还是同意了安装我们能提供的数传机。IBM 通信经理也做了同样多的工作，尝试说服纽约电话公司找一些数传机，他们不能提供任何电话机，尽管他们表达了这样的观点：如果我们碰巧已经以他们无法正式批准的方式在我们的系统内使用了他们的设备，他们将会寻求其他途径。

因此，我们决定切断进入我们中央计算机一半的线路，把那些数传机拆掉，放在航天中心 IBM 工作站的小推车上。然后，通过本地电话公司偷偷地安装了数传机，我们的课程才得以继续，第一个由非 IBM 实体创建的备用 APL\360 系统，就这样新鲜出炉了，尽管这是在只支持 PL/1 的政策背景下。

关于 APL 语言，你最遗憾的是什么呢？

Adin：我们尽了最大的努力来设计 APL 系统，并尽力在政治领域内使之被接受和广泛使用。在当时的条件下，我感觉对于这种语言来说没有任何的遗憾。后来想来，可能有一个遗憾是我们没有更早一点开始，而且没有在开发一个高效的编译器方面付出更大的努力，但是考虑到现有的资源限制，我们无法知道折衷平衡的成本。此外，我们有理由相信，目前对并行编程和在 FORTRAN 这样的传统编译语言采用类 APL 数组运算的兴趣，最终将会适时地带来同样的问题。

在您的工作方面，您是如何定义成功的呢？

58

Adin：事实表明，在 IBM 的业务发展的许多方面，APL 都是非常有用的工具。它提供了一个非常简单的计算机使用方法：允许研究人员和产品开发人员更有效地运用自己的知识解决工作中的实际问题，其范围可从理论物理一直到平面显示器的研发。它还可用于为组装生产线和仓库等主要业务系统制作原型，使它们能够快速启动，而且，还要在实现中“冻结”之前使用其他编程系统实施测试。

我们成功地将 APL 列入 IBM 的整体产品线，并引领其他计算机公司提供他们自己的符合国际标准的 APL 系统。

APL 作为一种工具和一个学科，在学术机构也得到了大量的应用，这也实现了它的开发主要目的之一：用于教育领域。

在将数组作为原始类型数据对象以及使用共享变量来管理同步问题等方面，APL 当之无愧是这种编程语言和系统的先驱，同样地，毫无疑问，它对包括并行编程在内的进一步发展会产生很大的影响。我非常欣慰地看到，在过去几个月中，在这个领域又成立了三家独立的计算机产业联盟。

第 4 章

Forth

Forth 是 Chuck Moore 在 20 世纪 60 年代设计的一种基于堆栈的连接合成语言（concatenative）。其主要特点是使用堆栈来保存在堆栈上操作的数据、字，弹出参数并压入结果。语言本身很小，足以运行在从嵌入式设备到超级计算机的任何平台上，而且表现力很强，用几百字就足以构建出一个有用的程序。它的后续语言包括 Chuck Moore 自己的 colorForth 及 Factor 编程语言。

60

4.1 Forth 语言与语言设计

您是如何定义 Forth 的呢？

Chuck：Forth 是一种使用最小句法的计算机语言。它具有显式参数堆栈，允许高效的子程序调用。这导致须采用后缀表达式（操作符后接参数），并鼓励使用多个短程序在堆栈上共享参数的高度分解化的编程风格。

我听说 Forth 这个名字代表的是第四代软件的意思。您能给我们详细解释一下吗？

Chuck：Forth 的名字来自于“第四代”，这暗指“第四代计算机语言”。我记得我跳过了一代。Fortran 语言/COBOL 是第一代语言；Algol/Lisp 是第二代。这些语言都强调句法。句法越复杂，越可能做更多的错误检查。然而，大多数错误都发生在句法方面。我决定尽量减少句法以有利于语义。事实上，Forth 的字均具有某种意义。

您把 Forth 看成是一个语言工具包。我可以理解这种观点，因为相对于其他语言，它具有相对简单的句法，并能够以较少的字来构建一个字典。我还漏掉了什么吗？

Chuck Moore：没有，基本上它是非常容易分解的。Forth 程序由许多小的字组成，而 C 程序则是由更少一些的大字组成。

我所说的小字是指典型的一行长定义的一个字。该语言可以通过定义一个相对于以前的字来说的新字来构建，而且你可以建立层次结构，直到你有上千个字为止。我们面临的挑战是：1）决定哪些字是有用的；和 2）记住这些字。目前我在做的应用程序有上千个字。而且，我使用了字搜索工具，不过，即使你还记得它的存在而且知道它是如何拼写的，你也只能搜索一个字。

现在，这导致了不同的编程风格，程序员要习惯于这种方法得需要一段时间。我看到过很多 Forth 程序，看起来非常像 C 程序硬套到 Forth 程序中的，这不是 Forth 的目的。Forth 的目的是有一个全新的开始。有关这种工具集的有趣之事是，与核中预定义的字相比，你使用这种方式定义的字，每一位都非常高效或是更有意义。这样做没有什么坏处。

包含来自 Forth 实现的许多小字的结构，是否是外部可见的？

Chuck：这是我们非常高效的子程序调用序列的结果。因为语言是基于堆栈的，所以没有参数传递。这只是一个子程序的调用和返回。堆栈是暴露的。编译的是该机器语言。与一个子程序的切换从字面上看是一个子程序调用指令和一个返回指令。另外，你可以下移到类似于汇编语言的层次。你可以定义一个字，它将执行子程序调用，而不是实际的机器指令，所以您可以实现与任何其他语言相同的效率，甚至可能会更高效一些。

61

您没有 C 调用的开销。

Chuck：对。这使得程序员具有极大的灵活性。如果你提出了一个巧妙的因式分解问题，您不仅可以让它非常高效，你还可以让它具有很强的可读性。

另一方面，如果你做得不好，那么最终的结果就是没人能读懂你的代码，即使项目经理能够明白一切，但就是读不懂你的代码。您还可以创建一个真正的混乱局面。因此，它是一把双刃剑。你既可以做得很好，你也可以做得非常糟糕。

您所说的（或者是您所显示的代码），会让使用另一种编程语言的开发者对 Forth 感兴趣吗？

Chuck：很难让一位有经验的程序员对 Forth 感兴趣。那是因为他在学习他的语言/操作系统的工具方面花费了很多精力，而且已经建立了适合于它的应用程序库。告诉他 Forth 更小、更快而且更容易，这与重新编写所有的代码相比，根本没有什么说服力。新入门的程序员或工程师无须编写代码，而且也不会面对那些障碍，所以更容易接受 Forth，有经验的程序员开始做一个带有新的约束条件的新项目时可能会这样，使用我的多核芯片时也可能会这样。

您刚才提到，您已经看到很多 Forth 程序项目看起来像是 C 程序。您如何设计一个更好的 Forth 程序呢？

Chuck：自下而上。

首先，你很可能会有一些必须生成的 I/O 信号，所以你就生成了它们。然后，你编写一些代码来控制这些信号的产生。然后，你的工作会逐步上升，最后直到最高级别的字，你把它叫做 go，然后你键入 go，然后就一切 OK 了。

我很不信任自上而下的系统分析师。他们确定了问题所在，然后以非常难以实现的方式来分解它们。

领域驱动设计建议描述了客户字典方面的业务逻辑。在构建一个字的字典和使用来自问题领域的专门术语之间有联系吗？

Chuck：在程序员开始编程之前，非常希望他了解这个领域。我会同客户对话。我会听一听他使用的字，我会用尝试使用这些字，这样一来，程序员就会明白程序在做些什么。Forth 具备这样的可读性，因为它使用了一种后缀符号。

如果我在做财务应用程序，我想可能有一个字叫“百分比”。你可以说“2.03%”之类的东西。而且，参数就是 2.03%，而且可以自然而然地工作和阅读。

为什么一个以穿孔卡片开始的程序在因特网时代的现代计算机上仍然有用呢？Forth 是在 1968 年为 IBM 1130 设计的。在 2007 年仍然选择这种语言进行并行处理，这无疑是非常令人惊叹的。

Chuck：它也在与时俱进。但是，Forth 可能是最简单的计算机语言。它对程序员没有任何限制。程序员可以定义那些字，它们会以简洁的层次化方式来描述问题。

您在设计程序时，将英文可读性作为一个目标了吗？

Chuck：在最高层次上是这样的，但对于描述或功能来说，英文并不是一种很好的语言。我们的设计目标不是这个，但在定义新字方面，英文确实和 Forth 一样具有相同的特征。

您通常是通过解释它们在以前定义的字中是用什么来定义新字的。在自然语言中，这可能会产生问题。如果你去查字典，你会发现很多定义是循环的，你并没有得到任何内容。

专注于文字而不是您可能在 C 中会遇到的括号和方括号句法，这种能力会使得 Forth 程序更易于实现高品味吗？

Chuck：我希望如此。这会让 Forth 程序员关注事物的外观，而不是仅仅关心其功能。如果你能将一系列字集合在一起，这是一个很好的感觉。实际上，这就是为什么我开发 colorForth。我对 Forth 中仍然存在的句法非常恼火。例如，您可以通过一个左括号和右括号来限制注释。

我看着那些标点符号说："嘿，也许还有更好的办法"。更好的方法是相当昂贵的，源代码中的每一个字都必须附加标签，但一旦我能容忍这些开销，那就会非常愉快了：这些有趣的小符号都不见了，它们全被这些字的色彩取代了，对我来说，这是一个非常优雅的标记方式。

我受到了色盲们的无休止的批评。我试图把他们排除在程序员队伍之外，他们对此非常生气，但最终有人使用字符集区分，而不是一种颜色区分，这也是一个令人非常愉快的方式。

关键是每个字的 4 位标记，它给你提供了 16 种选择，而且编译器可以立即判定它们是什么目的，而不必从上下文中推断。

第二代和第三代语言采用的是最低方式，例如使用元循环的逐步实现。就语言概念和所需的硬件数量支持来说，Forth 是一个很好的例子。这是那个时代的特征，还是随着时间的推移您开发的呢？

Chuck：不，有一个尽可能小的内核，这是一个精心设计的目标。以尽可能少的字来预定义，然后让程序员添加他认为合适的字。

这么做的最大原因是为了实现可移植性。当时，有几十台小型计算机，后来变成了几十台微机。而且我本人不得不在这些机器上面运行 Forth。

我想让它尽可能地容易些。实际上，可能会有 100 字左右的内核，或者这只是足以产生一个我称之为操作系统的东西，但它不太完整，它还有另外一两百个字。然后你就可以准备做一个应用程序了。

我会提供前两个阶段，然后让应用程序的程序员完成第三个阶段，我通常也是程序员。我定义我所知道的必要的字。第一个百字可能会在机器语言或汇编语言当中，或者是直接同特定平台打交道。第二百或第三百个字是高级字，以尽量减少机器对以前定义的低层字的依赖。然后，这个应用程序几乎是完全独立于程序的，它很容易从一台小型计算机移植到另一台小型计算机上。

您能很容易地在第二阶段移植吗？

Chuck：当然。例如，我有一个用来编辑源代码的文本编辑器。它移植时通常不需要做任何变化。

这就是那个传闻的来源吧，他们说每当您运行一台新机器时，您就立刻开始把 Forth 移植上去？

Chuck：是的。事实上，这是了解机器如何工作的最简单的途径，这种特殊的功能是基于实现 Forth 字的标准封装的难易程度。

您是怎样发明间接线程代码的？

Chuck：间接线程代码是一个有些微妙的概念。每个 Forth 字在字典中都有一个入口。在直接线程代码中，遇到这个词时，每个入口都指向待执行代码。间接线程代码指向包含该代码地址的一个位置。这使得除了地址之外还可以访问其他信息，例如，一个变量值。

这也许是字最紧凑的表示方法。它已被证明相当于直接线程和子程序线程代码。当然，这些概念和术语在 1970 年是不为人所知的。但在我看来，这是最自然的实现多种类型字的方式。

Forth 会对未来的计算机系统有何影响呢？

Chuck：这已经发生了。我已经在为 Forth 优化微处理器方面工作了 25 年，最近在做一种多核心芯片，其核

心是 Forth 计算机。

Forth 能提供什么呢？作为一种简单的语言，它允许使用一个简单的计算机：256 个本地内存字，2 个下堆栈（后进先出栈），32 条指令，异步操作，易与邻居沟通。它非常小，功耗也很低。

Forth 鼓励高度可分解的的程序。这非常适合多核心芯片所要求的并行处理。许多简单的程序，鼓励每个有想法的设计。而且也许只要另外编写 1%的代码。 64

每当我听到人们吹嘘数百万行代码，我知道他们严重地误解了他们的问题。现在没有什么问题需要数百万行的代码。问题出在草率的程序员、不合格的经理或不可能的兼容性要求身上。

使用 Forth 来为许多小的计算机编程是一个很好的策略。其他语言恰恰不具有模块化和灵活性。随着计算机变得越来越小，而且计算机网络都在共同运转（智能微尘？）（译注1），这将是未来的环境。

这听起来像是 Unix 的一个主要观点：有多个程序，每个程序在做一件事，它们相互作用。现在它仍然是最好的设计吗？不是一台计算机上运行多个程序，我们有可能通过网络运行多个程序吗？

Chuck：比如 Unix 和其他操作系统实现多线程代码的概念，就是并行处理的一个先驱。但是它们也有重要的区别。

大型计算机能够负担多线程通常需要相当大的开销。毕竟，已经有了一个庞大的操作系统。但是对于并行处理来说，几乎总是计算机越多越好。

使用固定的资源，计算机越多意味着计算机越小。而且，小型计算机无法负担大型计算机所需的开销。

小型计算机将在芯片、芯片之间和跨射频连接之间实现联网。小型计算机的内存也小。没有可供操作系统使用的空间。该计算机必须是自主的，具有独立的沟通能力。因此，沟通必须是简单的，没有复杂的协议。软件必须是紧凑而高效率的。这是 Forth 的一种理想应用。

需要数百万行代码的那些系统将变得无关紧要。它们是大型中央计算机的产物。分布式计算需要一种不同的方法。

旨在支持庞大而且符合句法的代码鼓励程序员编写大型程序。他们倾向于因此而获得青睐并得到奖赏。他们没有追求紧凑的压力。

虽然句法语言生成的代码可能会比较小，但通常它不是这样。为了实现句法所隐含的通用性，这常常会难用低效的对象代码。这不适合小型计算机。在设计良好的语言中，源代码和目标代码之间存在着一对一的对应关系。对于程序员来说，他的源代码会产生什么样的目标代码，这是显而易见的。这种自我实现是高效率的，同时也降低了对于文档的需求。

在一定程度上，Forth 的设计同时追求源代码和二进制输出的紧凑性，而且它也因此深受嵌入式开发者的欢迎，不过，许多其他领域的程序员有理由选择其他语言。这种语言的设计是否有只添加源或输出开销的方面？ 65

Chuck：Forth 确实是非常紧凑。原因之一是它几乎没有句法。

译注1：智能微尘（Smart Dust）实际上是一个比米粒还小的计算机。尽管一个一立方毫米甚至更小的智能微尘在尺寸上与沙子没多少区别，但是它却拥有传感器、处理器等组件。

其他语言似乎拥有谨慎添加的句法，它为句法检查进而进行错误检测提供了机会。

Forth 提供错误检测，因为它缺乏的冗余几乎没有机会。这有助于更紧凑的源代码。

我使用其他语言得出的经验是，大部分错误源于句法。设计师似乎是为程序员创造可由编译器检测到的错误的机会。它的效率似乎不高。它只是为编写正确的代码增加了麻烦。

类型检查就是一个例子。为不同的数分配类型使得错误可以被检查出来。一个意想不到的后果是，程序员必须转换类型，而且有时为了做他们想要的东西，还得去逃避类型检查。

句法的另一个后果是，它必须满足所有的预定申请。这使得它变得更为复杂。Forth 是一种可扩展的语言。程序员可以创建同编译器提供的一样高效的结构。因此，不必预期和提供所有的功能。

Forth 的一个特点是使用后缀操作符。这简化了编译器，并提供源代码到目标代码的一对一转换。程序员增强了对源代码的理解，而且编译的代码也由此变得更加紧凑。

最近许多编程语言的支持者（尤其是 Python 和 Ruby）把可读性作为一个重要的特点。Forth 容易学习和维护这些吗？在可读性方面，Forth 可以教给其他编程语言什么呢？

Chuck：所有的计算机语言都声称是可读的。其实他们不是。这些语言看起来对所有人都是这样，但新手总是感到困惑。

问题在于晦涩难懂的、任意的、神秘的句法。它包括所有的括号、连字符等。您试图了解它为什么存在，并最终得出结论：它们并没有很好的理由。但是你仍然必须遵循这些规则。

而且，你不能读出这种语言。你必须像 Victor Borgia 那样念出标点符号。

Forth 通过最小化句法缓解了这个问题。其中的@和！这两种神秘符号发音为“fetch”和“store”。因为它们出现得如此频繁，所以它们被符号化。

程序员被鼓励使用自然语言的文字。这些串在一起，没有标点符号。随着选好的字，你可以构造合理的语句。事实上，诗歌也可以用 Forth 来写。

另一个优势是后缀符号。像“6 英寸”这样的短语可以以很自然的方式来对参数 6 应用操作符“英寸”。可读性很强。

另一方面，程序员的工作是创建一个描述该问题的字典。这个字典的规模可以相当大。读者必须知道它才能发现该程序的可读性。而且程序员必须定义有用的字。

总之，阅读程序需要花些功夫。使用任何语言编写的程序都是如此。

在工作方面，您是如何定义成功的？

Chuck：一个完美的解决方案。

一个人不使用 Forth 编写程序。Forth 就是一种程序。一个人可以添加字来构造一个描述问题的字典。显然，在定义了正确的字之后，那你就可以通过交互方式解决问题，无论问题在哪一方面是相关的。

例如，我可以定义描述一个电路的字。我想为该电路添加芯片，显示布局，验证设计规则，运行模拟。这些字通过应用程序来完成这些工作。如果它们是精心挑选出来的，并提供了一种紧凑、高效的工具集，那么我就成功了。

您在哪里学习编写编译器的？当时每个人都必须做这些东西吗？

Chuck：嗯，我是在 20 世纪 60 年代去的美国斯坦福大学，并有一组研究生在为 Burroughs 5500 编写一个 ALGOL 编译器。我想，其中只有三四个人给我留下了深刻印象，三四个家伙可以坐下来写一个编译器。

我说："好吧，如果他们能做到这一点，我也可以做到"，然后我就做了。这并不很困难。在当时，编译器还有一些神秘色彩。

现在仍然如此。

Chuck：是的，但是少多了。当你得到不断涌现出来的这些新语言时，我不知道他们是如何被解释或编译的，但是，黑客之类的人肯定愿意这样做。

操作系统是令人好奇的另外一个概念。操作系统是极为复杂的和完全不必要的。这是一个光辉的事情，Bill Gates（比尔盖茨）向全世界兜售操作系统的概念。这也许是世界上前所未有的最大骗局。

对你来说，操作系统绝对是什么事也没有做。只要你现在有一些东西：一个称为磁盘驱动程序的子程序，一个称为通信支持的子程序，它不会做别的事。事实上，Windows 在重叠、磁盘管理这样一些不相干的事情上花费了大量的时间。你有数 MB 的磁盘，你有数 MB 的存储器。世界已经在以"操作系统不再必要"的方式发生着变化。

设备支持怎么样？

Chuck：每个设备你都有一个子程序。这是一个库，而不是操作系统。在你需要的时候调用或加载那些子程序。

您如何在短时间中断内恢复编程？

Chuck：我根本没有发现麻烦的短编码中断。我把注意力集中在这个问题上，并且整夜都在想这个问题。我认为这是 Forth 的一个特点：在很短的时间内（天数）全力以赴来解决问题。它有助于 Forth 应用程序自然地分解为子项目。大多数 Forth 代码很简单，而且也易于重读。当我在做这些很麻烦的事情时，我对它们的评论很高。好的评论有助于重新输入一个问题，但它总是有必要阅读和理解代码。

就设计或编程来说，您的最大失误是什么？您从中学到了什么？

Chuck：大约 20 年前，我想开发一个工具来设计 VLSI 芯片。我没有在新 PC 上使用 Forth，所以我想我会尝试不同的办法：机器语言。不是汇编语言，而是实际输入十六进制指令。

我像使用 Forth 那样构建代码，使用很多在层次上相互交互的字。它运行得很好。我用了 10 年。但它难以维护和编写文档。最后，我使用 Forth 对它进行重新编码，它变得更小了，也更简单了。

我的结论是，Forth 的效率要高于机器语言。部分是因为它的交互性，部分是因为它的句法。Forth 代码的一个好的方面是，数字可以通过用于计算它们的表达式来记载。

4.2 硬件

人们应该如何看待他们的硬件开发平台：作为一种资源，抑或是一种限制？如果认为硬件是一种资源，您

可能需要优化代码并利用每一个硬件功能，如果您认为这是限制它，您可能会去编写代码，同时想象您的代码将在更强大的新版本硬件上运行得更好，这不是一个问题，因为硬件的发展速度非常迅速。

Chuck：软件一定要瞄准硬件，这是一个非常敏锐的观察。个人计算机软件肯定会先于更快的计算机的发展，而且对它也很宽容。

68 但是对于嵌入式系统来说，软件期望该系统在项目周期中是稳定的。而且，并没有很多的软件从一个项目移植到另一个项目上。因此，这里的硬件是一种约束，但不是限制。不过，对于个人计算机来说，硬件资源还会增加。

采用并行处理应该会改变这种状态。无法利用多台计算机的应用程序将会受到限制，因为单台计算机的发展速度变慢了。改写遗留软件来优化并行处理是不切实际的。而且，寄希望于聪明的编译器节省时间，那只是一厢情愿的想法。

并发问题的根源是什么？

Chuck：并发问题的根源在于速度。计算机必须在应用程序中做许多事情。这些可以在多任务处理器中完成。或者，它们可以同时使用多个处理器来完成。

后者的速度要快得多，而且现代软件需要这种速度。

这种解决方案是硬件、软件还是两者兼有？

Chuck：将多个处理器粘在一块并不难。因此，硬件方案就有了。如果编写软件来利用这一优势，问题就可以得到解决。但是，如果该软件可以重新编程，使其高效，就不再需要多个处理器了。问题是使用多个处理器而不用修改遗留软件，这是从来没有实现的智能编译器方法。

我很惊讶，20 世纪 70 年代编写的软件没有/不能被改写。原因之一可能是，在那些日子里软件令人兴奋，事情都是第一次完成；程序员因为快乐而每天工作 18 个小时。现在，编程是一个朝 9 晚 5 的工作，作为团队工作的一个时间表，没有多少乐趣。

因此，他们又增加另一层软件来避免重新编写旧软件。至少这要比重新编写一个愚蠢的文字处理器更有趣。

我们可以在普通计算机上实现很强的计算能力，但是这些系统又在做多少实际计算（computing，calculating）呢？它们又有多少只是在移动和格式化数据呢？

Chuck：你说得对。大多数计算机活动是在移动数据，而不是计算。不只是移动数据，还有压缩、加密且置乱（Scrambling）。如果要实现高数据速率，就必须使用电路，所以有人很疑惑为什么还需要一台计算机。

我们可以从这些问题中借鉴到什么？我们是否应该以不同的方式来构建硬件？
Don Knuth 发起了挑战：检查在 1 秒的时间里计算机内部发生了什么。他说，我们的发现可以改变很多事情。

Chuck：我的计算机芯片通过一个简单的低速乘法运算认识到了这一点。它的使用并不是很频繁。在内核和访问内存之间传递数据是重要的特征。

69 **一方面你有一种语言，它真正让人们能够开发自己的字典，而不必考虑硬件的存在。另一方面，你有一个非常依赖于该硬件的非常小的内核。有趣的是，Forth 如何能够弥合两者之间的差距。在其中一些机器上，**

除了您的 Forth 内核之外，没有其他的操作系统吗？

Chuck：没有，Forth 是真正的独立。所需的一切都在内核之中。

但是，对于使用 Forth 编写程序的人来说，它从硬件当中抽象出来了。

Chuck：对。

Lisp 机器（Lisp Machine）做了一些类似的工作，但它从来没有真正流行起来。Forth 悄悄地做了这项工作。

Chuck：嗯，Lisp 没有解决 I/O 问题。事实上，C 也没有解决 I/O 问题，因为它需要一个操作系统。Forth 从一开始就很关注 I/O。我不相信最常见的共同点。我认为，如果你使用一台新机器，唯一的原因是新机器在某些方面不同，而且你想利用这些差异。所以，你想在输入输出级做工作，您可以做到这一点。

Kernighan 和 Ritchie 支持 C 说，他们想要通过最小的共同因素来使得连接端口更容易。然而，你会发现如果你这么做，连接端口会更容易。

Chuck：我会使用标准方式来做。我会使用一个字，我认为可能会是 fetchp，这将会从一个端口取 8 位。它的定义因不同的计算机而异，但它会在栈上具有相同的功能。

那么，从某种意义上说，Forth 相当于 C 加上标准 I/O 库。

Chuck：是的，但我早期在标准 FORTRAN 库（Standard FORTRAN Library）上工作过，这太可怕了。它只有错误字。这是非常昂贵和庞大的。很容易定义半打指令来执行 I/O 操作，而你并不需要一个预先定义的协议开销。

您发现自己在那方面做了很多工作？

Chuck：在 FORTRAN 方面，是这样的。您在处理 Windows 时，没有什么可以做。它们不会让你访问 I/O。我非常谨慎地同 Windows 保持距离，但是即使没有 Windows，在 Pentium（奔腾）机器上使用 Forth 也是最困难的。

它有太多的指令。而且，它有太多的硬件特性，比如说实际上你无法忽略的后备缓冲区和缓存。你必须渡过难关，而且让 Forth 运行所必需的代码初始化工作是最困难和最繁琐的。

即使它只需要执行一次，我花了大部分时间来试图发现它是如何正确工作的。我们让 Forth 独立运行在 Pentium（奔腾）机器上，所以有点麻烦也都是值得的。

这个过程可能要超过 10 年，部分是在追赶 Intel（英特尔）硬件的变化。

您提到 Forth 真正支持异步操作。您所说的异步操作是什么含义？

Chuck：嗯，它有好几层含义。Forth 一直有一个多道程序设计能力，一个称为 Cooperative 的多线程能力。

我们有一个字叫 `pause`（暂停）。如果你有一个任务，它会来到一个位置，若在那里没有什么立即可干，它便会说 `pause`。循环调度会指派计算机执行循环中的下一个任务。

如果你没有说 pause，你可以独占垄断计算机，但绝不会出现这样的情况，因为这是一台专用计算机。它在运行一个应用程序，所有的任务都很友好。

我想，所有任务都很友好，那只是在过去。这是一种可以运行这些任务的异步方式，它们在做自己的事情，根本不需要同步。再者，Forth 还有一个特点是，pause 这个字可以嵌入到较低级别里。每当你尝试读取或写入磁盘时，将会为你执行 pause 这个字，因为磁盘组知道这将不得不等待操作完成。

在我正在开发的新芯片，新的多核芯片中，我们采用了同样的设计理念。每台计算机都在独立运行，而且如果你在计算机上运行一个任务，在相邻的机器上运行另外一个任务，它们都在同时运行，不过它们可以互相沟通。使用线程的计算机无法完成这样的任务。

Forth 只是很好地把它们分解成这些独立的任务。事实上，在多核计算机的情况下，我可以使用不尽相同的程序，但我可以以同样的方式来分解程序，并使它们并行运行。

您在使用多线程协作时，每一个执行线程有它自己的堆栈吗，您会在它们之间切换吗？

Chuck：当你进行任务切换时，根据计算机的不同，有时你需要做的是在堆栈顶部保存这个字，然后再切换堆栈指针。有时，你实际上必须复制出栈内容并加载新东西，但在这种情况下，我将让它指向一个很浅的堆栈。

您会刻意限制堆栈的深度吗？

Chuck：是的。最初，堆栈的长度是任意的。我设计的第一个芯片堆栈深度为 256，因为我认为这是很小的。我设计的芯片其中有一个堆栈深度为 4。我现在认为 8 或 10 左右是一个很好的堆栈深度，所以我的最小化随着时间的推移变得更加严格。

我已经预料它会走另外一条道路。

Chuck：嗯，我在超大规模集成电路（VLSI）设计应用程序中有一个案例，我递归跟踪整个芯片，在这种情况下，必须将堆栈深度设置为 4000 左右。要做到这一点可能需要不同类型的堆栈，一种软件实现的堆栈。但事实上，在 Pentium 处理器上它可以是一个硬件堆栈。

4.3 应用程序设计

Application Design

您提到过这种观点：对于许多联网的小型计算机来说，例如智能微尘，Forth 就是一种理想的语言。对于这种小型计算机来说，您认为哪种应用是最合适的？

Chuck：当然是通信。但是我刚开始学习独立的计算机如何合作完成更大的任务。

我们所拥有的多核计算机非常非常小。他们使用的是 64 字的内存。那么，换一种方式，它们有 128 字内存：64 个是 RAM，64 个是 ROM。每个字可以容纳多达 4 条指令。在一个给定的计算机中，你最终可能会有 512 条指令，因此任务必须是相当简单的。现在，你怎么能针对像 TCP/IP 协议栈这样的任务，并把它分解给若

千个不同的计算机，其中你不需要使用任何超过 512 条指令的计算机来操作呢？这是一个很美妙的设计问题，而且我已经开始着手研究这个问题了。

我认为对于几乎所有的应用程序来说都是这样的。这很容易做一个应用程序，如果把它分解成独立的程序段，而且它试图在单一处理器上串行处理这些问题。我认为视频生成也是这样。当然，我认为压缩和解压图像也是如此。不过，我只是学习如何去做到这一点。我们公司还有其他人也在学习这个，而且也花费了很长时间。

它不适合哪些领域呢？

Chuck：当然是遗留软件。我真的很担心遗留软件，但只要你愿意重新考虑一个问题，我觉得你就会很自然地采用这种方式。我认为，它更类似于我们认为大脑是 Minsky 的独立智能代理方式。对我来说，一个智能代理就是一个小核心。这可能是在它们通信之间出现的想法，而不是在其中任何一个操作中产生的。

遗留软件是一个尚未引起重视，但非常严重的问题。它只会变得更糟，不仅是在银行业，还包括航空业和其他技术行业。问题是有数百万行代码。这些项目将会使用成千上万行的 Forth 重新编码。这样一来机器翻译就没什么意义了，它只会使代码的规模更大。但是没有验证代码的办法，成本和风险将是非常可怕的。遗留代码可能会是我们文明的衰落标志。

听起来你像在打赌：在未来 10 至 20 年，我们将看到越来越多的软件产生自松散连接的众多小组件。

Chuck：哦，是的。我敢肯定是这种情况。射频通信很好。它们考虑的是你的机构内的微智能代理，而且会固定问题并进行感知，这些代理可能只能通过射频或者是音频来进行通信。

它们做不到那么好。它们的数量很少。因此，这就变成了世界的走向问题。这是我们人类社会的组织方式。我们已经有 6 亿 5 千万个独立代理在连续运转。

如果字选择不当的话，可能会导致设计出粗糙、不好维护的应用软件。通过几十个或几百个小字来构建大型程序，会弄得乱七八糟吗？如何避免呢？

Chuck：嗯，你真的无法避免。我发现自己的字选得不好。如果你这样做，你会把自己搞糊涂。我知道，在一个应用程序中，我使用了一个字，我忘了它现在叫什么，但是我已经定义了它并对其进行修改，最终的结果是它与原来的意思恰好相反。

这就好比是你用了“右”这个字，而它的功能却是向“左”去。这样就会造成一团混乱。我为此考虑了一段时间，并最终重命名了这个字，因为无法理解含有这个字的程序，它为你的理解带来了很多噪音干扰。我喜欢用英文单词，而不是缩写。我喜欢把它们拼出来。另一方面，我喜欢让它们短一些。一段时间后，你用完了那些有意义的英文短单词，你不得不去想别的办法。我非常讨厌前缀，这是一种粗暴的试图创建命名空间的方法，你可以反复使用同一个旧词。对我来说，它就像是一种投机取巧的逃避方法。这是一种简单的区分方法，但是你应该更加聪明一些。

Forth 应用程序往往会有你可以重复使用的截然不同的字典。在这种情况下，这个字是这种功能；在那种背景下，它完成的是其他功能。在我的 VLSI 设计经历中，所有的这些理想主义无一成功。我至少需要一千个字，而且它们不是英文字，他们是信号名称或其他什么，而且，我很快就重新考虑定义、怪里怪气的字和前缀等。所有的这些可读性都非常差。但另一方面，它里面满是各种各样的 nand、nor，以及 xor 门。如有可能，我会使用这些字。

现在，我看到其他人在编写 Forth 程序；我不想假装成唯一的 Forth 程序员。其中有些人提出的名字也非常好，而另外一些人则非常糟糕。有人提出了一个可读性很强的句法，而其他人则认为这并不重要。有些人提出了很短的字定义，有些人的字则是一页长。这里没有什么规则可言，它只是风格习俗而已。

此外，Forth、C、Prolog、ALGOL 和 FORTRAN 之间的关键区别就是，传统语言试图事先去把所有可能的结构和句法都定义好，并把它内置到语言里。这就促生了一些非常笨拙的语言。我认为 C 是一种笨拙的语言，它使用括号、大括号、冒号及分号等之类的东西。Forth 就不存在这些问题。

73 我不必去解决普遍问题。我只是提供一个工具，别人可以利用它来解决他们遇到的任何问题。有能力去做任何事情，而不是有能力去做每一件事情。

微处理器是否应该包括源代码，以便它们甚至可以在几十年后仍保持稳定？

Chuck：你说得对，将源代码包含在微型计算机当中将会把情况很好地记录下来。Forth 是紧凑的，会对这个有帮助。但下一步应该包括编译器和编辑器，以至于微机代码能够在不涉及可能已丢失的计算机/操作系统的情况下进行检查并作出修改。colorForth 就是我在此方面的一种尝试。这仅仅需要几千字节的源代码和目标代码。它可很容易在闪存上存储，并在遥远的将来使用。

在一种语言的设计和使用该语言编写的软件设计之间有什么联系？

Chuck：语言决定了它的用途。人类的自然语言就是这样。看一看罗曼语系（法语和意大利语）、西方语系（英文、德语和俄语）和东方语系（阿拉伯文和中文）之间的区别就知道了。它们影响到了他们的文化和世界观。它们影响到了说什么和怎么说。其中，英文特别简洁，并且越来越受欢迎。

因此，人-计算机语言也是如此。最初的语言（COBOL、Fortran）过于冗长。后来的语言（Algol、C）有过多的句法。这些语言必然会导致庞大的笨拙的算法描述。可以用它们来进行表述，但是表述得很差。

Forth 解决了这些问题。它是相对无句法的。它鼓励紧凑、高效的描述。它对注释的要求达到了最小化，这往往是不准确的，而且也分散了对代码本身的关注。

Forth 也有一个简单、高效的子程序调用。在 C 语言里，子程序调用需要代价高昂的设置和恢复。这样就限制了它的应用。而且还鼓励分担调用代价的复杂参数集，并会导致大量的复杂子程序。

高效率使得 Forth 应用程序能够高度地分解成许多小子程序。而且它们通常是这么做的。我的个人风格是单行定义：几百个子程序。在这种情况下，为代码分配的名称变得非常重要，无论是作为助记手段还是作为实现可读性的一种方式。可读性强的代码对文档的要求较低。

缺少句法使得 Forth 相应地缺乏约束。这对我来说，会激发个人的创造力，并编写出一些非常好的代码。其他一些不利因素主要是担心管理失控和缺乏标准化。我认为这是一种语言故障管理的失败。

74 **您说过："大多数错误都是源于句法"。如何避免在 Forth 程序中的其他错误，比如逻辑错误、可维护性错误，以及做出不良风格的决定呢？**

Chuck：噢，Forth 的主要错误与堆栈管理有关。通常，您会无意中在堆栈中留下了一些东西，而且它稍后可能会去访问。我们有与字相关的堆栈注释，这是很重要的。它告诉你堆栈进入时是什么东西，以及在堆栈退出时是什么东西。但是，这只是一个注释。你不能相信它。

有些人在实际执行它们，并利用它们来进行核查和堆栈操作。

基本上，解决办法就是分解。如果你有一个字，其定义是一行长，你可以通过阅读它来思考堆栈的行为并最终得出它是正确的结论。你可以测试它，看看它是否按照你所想象的方式工作，但即使如此，你会遇到堆栈错误。`Dup` 和 `drop` 这些字无处不在，而且要正确使用。从上下文中提取字，并把它们放入到输入参数之中，并查看一下它们的输出参数，这具有十分重要的意义。同样，当你自下而上的工作时，你知道你已经定义的所有字都能正常工作，因为你已经测试过了它们。

此外，Forth 里面的条件语句很少。它有一个 `if-else-then` 结构，还有一个 `begin-while` 结构。我的哲学，我经常拿它去教育别人，就是在你的程序中将条件语句的数目最小化。不是要用一个字来测试，并且不是用它做这个或用它做那个，你有两个词：一个做这个，一个做那个，你要正确使用它们。

现在，C 不是这样工作的，因为调用序列非常昂贵，它们往往会使用参数，这样一来，根据不同的调用方式，同一个程序就可以完成不同的功能。这就是导致遗留软件的 bug，以及并发难题的原因。

试图解决实现方面的缺陷了吗？

Chuck：是的。循环是不可避免的。循环可以是非常、非常好的。但是 Forth 循环，至少可以说 colorForth 循环，是具有单一入口和单一出口的非常简单的循环。

为了让新手更加愉快、更加有效地编程，您有什么建议呢？

Chuck：嗯，肯定不会出你所料，我会说，你应该学会编写 Forth 代码。即使你不准备去专业编写代码，接触一下它也会教你学到其中的一些教训，让你对你所使用的任何一种语言都有更好的理解。如果我是编写 C 程序，我几乎写不出什么来，但我能以 Forth 的风格写出很多简单的子程序来。即使这里面涉及了一个成本问题，我想在可维护性方面依然是值得的。

另一件事是保持简单。设计飞机或编写应用程序（即使是文字处理器）的必然趋势是，没完没了地增加新功能，直到成本变得无法忍受为止。最好是有五六个文字处理器侧重于不同的市场。使用 Word 来写电子邮件是很傻的，因为 99%的功能是不必要的。你应该使用电子邮件编辑器。过去曾经是这样，但目前的趋势似乎是与其背道而驰。我不清楚这是为什么。

保持简单。如果你遇到一个应用程序，如果你是设计团队的一分子，试着说服其他人保持简单。不要去预测预期。不要去解决你认为未来可能发生的问题。解决你目前的问题。预测是效率很低的。你可以预见到 10 件事，但其中也许只有一件会发生，所以你浪费了大量的精力。

您如何识别出简单性呢？

Chuck：我认为崭露头角的复杂性科学，其宗旨之一是如何衡量复杂性。我所喜欢的描述（我不知道是否还有其他描述）是最短的描述，或者是，如果你有两个概念，描述较短的那个就是简单的。如果你能给出一个较短的定义，你就是给出了一个简单的定义。

但是，这有时候也会失败，因为任何描述都要视情况而定。如果你可以用 C 编写出很短的子程序，你可以说这是很简单的，但是你要依赖于 C 编译器、操作系统和要执行它的计算机等东西。因此，实际上并没有简单的事，当你考虑的范围更广时，事情会变得非常复杂。

我想它跟美丽一样。你不能定义它，但是你一看就知道，简单就是小。

团队协作对编程有何影响？

Chuck：团队协作，被过于高估了。一个团队的首要工作就是要分割成相对独立的部分。每个部分再分配到个人。团队负责人负责考虑将这些部分聚合到一起。

有时两个人可以在一起工作。问题越辩越明。但过分的沟通成为了沟通本身的目的。集体思维不利于创新。而且当几个人在一起工作时，必然有一个人会做这项工作。

这对于各类项目都是有效的吗？如果您编写像 OpenOffice.org ...这样的功能丰富的东西，这听起来会相当复杂，不是吗？

Chuck：像 OpenOffice.org 这样的东西会分解成子项目，每个人在充分沟通的基础上来编程一个子项目，以确保兼容性。

您如何识别一个好的程序员呢？

Chuck：好的程序员能够很快地编写出优秀的代码。优秀的代码是正确的、紧凑的，而且可读性也很强。“很快”是指几个小时或几天。

糟糕的程序员会谈论那些问题，将时间浪费在规划而不是编程上，并会煞有其事地编写和调试代码。

您对编译器怎么看？您认为它们会掩盖程序员的真功夫吗？

Chuck：编译器很可能是有史以来编写得最糟糕的代码。它们是由此前从来没有写过编译器的人写的，而且之后再也没写过编译器。

语言越复杂，编译器就会越复杂、bug 丛生，也越不能使用。但是简单语言的简单编译器是一个不可或缺的工具，如果只是为了文档的话。

比编译器更重要的是编辑器。各种各样的编辑器可供每个程序员自行选择，这对于协作来说危害极大。这促进了小作坊之间的相互转换。

编译器作者的另一处败笔是强迫使用键盘上的每一个特殊字符。因此，键盘就不能变得更小、更简单了。同时，源代码也变得难以理解。

但是，程序员的技能是独立于这些工具的。他能迅速地克服自己的缺点，并编写出优秀的代码。

应该如何把软件记录下来？

Chuck：我对于注释的评价要远低于其他人。主要有以下几个原因：

- 如果注释很简洁，那它们往往含糊不清。然后，你不得不猜测它们的意思。
- 如果注释很繁琐，就会在它们试图去解释的代码中喧宾夺主。很难找到代码并将它与注释关联起来。
- 注释往往写得很差。程序员的文字技能并不出众，尤其是英文不是他们的母语时更是如此。术语和语法错误往往会让这些注释可读性极差。
- 最重要的是，注释往往是不准确的。代码可能会在没有更新注释的情况下进行修改。虽然代码可能会进行严格审查，但很少会检查注释。不准确的注释会比没有注释带来更多的麻烦。读者必须判断注释或代码是否正确。

注释往往是错误的。他们应该解释代码的作用，而不是代码本身。为代码释义是无益的。如果它不准确，那就是彻头彻尾的误导。注释应该解释代码为什么存在，它想要完成什么功能，以及在完成功能的过程中使用了什么技巧。

colorForth 将注释分解为阴影块。这样就把它们从代码本身中除去了，使该代码更具可读性。不过，它们可以立刻读取或更新。它也将注释大小限制在了代码的大小中。

注释并不能代替正确地编写文档。文档必须平铺直叙地解释代码模块。它应该充分地扩展注释，并将重点集中在通俗易懂和完整的解释之上。

当然，这是很少见的，往往代价不菲，而且也很容易丢失，因为它是与代码相分离的。

引自 http://www.colorforth.com/HOPL.html：

"最后讨论 Forth 的专利问题。但由于软件专利是具有争议的，而且很可能会牵涉到美国最高法院，美国国家射电天文台放弃追求这个目标。于是，恢复了我的权利。我认为不应该为想法授予专利。事后表明这是 Forth 在公共领域发展的唯一机会。在那里它才能蓬勃发展。"（译注2）

直到今天，软件专利仍有争议。您对专利还持与原来同样的观点吗？

Chuck：我从来没有支持过软件专利。这太像是给一种想法申请专利了。而且，语言/协议的专利问题特别令人不安。一种语言只有使用才会成功。任何不鼓励使用的事情都是愚蠢的。

您认为专利技术是否会阻止或限制它的传播呢？

Chuck：销售软件是很难的，这是因为它很容易复制。各家公司都不遗余力地去保护它们的产品，在这个过程中，有时还会让它无法使用。我对这个问题的回答是销售硬件，放弃软件。硬件是很难复制的，加上为它开发的软件，就更有价值了。

专利是解决这些问题的一种方式。它们已被证明是创新的美妙福音。但是，其中有一个微妙的平衡，必须减少不重要的专利，并保持与现有技术/专利的一致性。同时，专利授权和实施等相关事务还需要大批费用。最近提出的专利法改革似乎要排斥个人发明者以支持大公司。这将是一场悲剧。

译注2：1968 年，Charles Moore 在美国国家射电天文台（NRAO）工作期间，和 Elizabeth Rather 共同创建了 Forth 语言。

第 5 章 79

BASIC

1963 年，Thomas Kurtz 和 John Kemeny 创建了 BASIC 语言，这种通用语言的设计初衷是用于教初学者编程，同时，富有经验的用户也可以用它来编写有用的程序。他们最初的设计目标是将编程从硬件细节中抽象出来。20 世纪 70 年代引入微型计算机之后，这种语言得以广泛传播；很多个人计算机都装有其定制的变体。尽管在 Microsoft 的 Visual Basic 和 Kurtz 的 BASIC 之后，这种语言已经不再需要使用代码行号和 GOTO 语句，一代又一代的程序员还是从鼓励尝试和求知欲可得到回报的 BASIC 语言中体会到了编程的乐趣。

5.1 BASIC 背后的目标

The Ideas Behind BASIC

学习程序的最佳方式是什么？

Tom Kurtz：新人门的程序员应该不需要啃完整本手册。大多数手册过于冗长，难以持续吸引初学者的注意力。所以，它需要简单的编码设定和便于访问的易用实现，同时还要有很多例子。

某些老师更喜欢讲授这样的语言：在使用这种语言之前，程序员需要拥有很多经验。与此相反，您选择创建的这种语言，任何水平的程序员都可以很快上手，他们还可以根据经验完善自己的知识。

Tom：是的。一旦你已经学过编程，就很容易学习新的计算机语言。万事开头难。除非一种语言特别晦涩难懂，否则，你有过学习语言的基础，学会新语言就只有一步之遥。这类似于传统语言中的口语（这个更难学），一旦你学会了第一门罗曼斯（Romance）语，第二门就更简单。首先，它们的语法类似，其中有很多单词是一样的，而且句法也相当简单（即动词是否像英文那样位于中间，或者位于结尾）。

第一种语言越简单，普通学生就越容易学习。

是这种发展方式让您决定创建 BASIC 的吗？

Tom：我们（我和 John Kemeny 两人在 1962 年左右）决定开发 BASIC 时，我考虑过尝试开发 FORTRAN 或 Algol 的简化子集。结果行不通。大多数编程语言都有很难懂的语法规则，它们会成为初学者的拦路虎。我们设法把 BASIC 中诸如此类的东西全部删掉了。

BASIC 设计考虑了以下若干因素：

- 一行代码，一条语句。

  我们不能像 JOSS 那样，使用句号来结束一条语句（我认为）。而且，Algol 惯用的分号对我们来说也没有意义，就像在 FORTRAN Continuation（C）中那样。

- 代码行号就是 GOTO 的目标。

  我们不得不使用代码行号，因为这要远早于 WYSIWYG（所见即所得）时代。在我们看来，创造一个“语句标号（statement labe）”的新概念，并不是什么好主意。（后来，在创建和编辑程序变得更容易时，只要不使用 GOTO 语句，我们允许用户不使用代码行号；从那时起，BASIC 就实现了完全结构化。）

- 所有的运算都使用浮点。

  对于初学者来说，最难的概念之一就是整数类型和浮点类型之间的区别。当时，几乎所有的编程语言都屈从于大多数计算机硬件的体系结构，其中包括用于工程计算的浮点类型和用于提高效率的整数类型。

  使用浮点方式来处理所有的运算时，我们可以使用户免于对数值进行分类。如果运算需要整数值（比如一个数组下标），而用户提供的是非整数值时（如 3.1 所示），我们就必须在内部进行一些复杂的处理。在这种情况下，我们取整即可。

  由于二进制小数和十进制小数之间存在区别，我们也会遇到类似的问题。比如，在下列语句中：

```
FOR I = 1 TO 2 STEP 0.1
```

十进制小数 0.1 是一个无限循环二进制小数，我们必须使用一个模糊因子来判断循环是否完成。

（最初的 BASIC 中，并没有考虑二进制/十进制的转换问题，不过，最新的 True BASIC 已经考虑了这些问题。）

- 一个数就是一个数（是一个数）。

  在代码或数据语句中输入一个数时，没有格式要求。而且，PRINT 语句会以默认的格式应答。FORMAT 语句（或者其他语言中的类似语句）很难学习。新用户可能会感到困惑：我为什么必须学习它？我只不过想要一个简单的答案而已！

- 合理的默认。

如果有任何适用于“更高级”用户的复杂特性，它们不应该对初学者可见。诚然，最初的 BASIC 中并没有多少“高级”特性，不过这种思想是（过去是，现在也是）非常重要的。

以下事实证实了我们这种方式的正确性：教会新手编写简单的 BASIC 程序，只需要一小时左右的时间。一小时一次的培训课，我们最开始要讲四次，后来减到三次，两次，到最后减成了几盘录像带。

我曾经得出结论：计算机科学入门课程可以使用 BASIC 版本来讲授（它并不是最原始的一种语言，但它是包括结构化编程模型的一种语言）。你唯一无法做到的就是向学生介绍指针和存储分配的思想！

还有一点：早期运行一个程序需要若干步骤，即编译、连接、加载、执行。我们决定在 BASIC 中将这些运行步骤全部组合起来，以致于用户甚至不会意识到这些步骤的存在。

从计算历史来看，当时大多数语言都需要一个多遍编译器，它可能会消耗过多的昂贵的计算机时间。这样一来，我们就可以一次编译，多次执行。但是初学者的小程序只能编译和执行一次。它要求我们开发单遍编译器，而且如果编译阶段没有错误，就可以直接执行。

在向学生报告错误方面，我们通报了五个错误之后就会停止。我能够回想起 FORTRAN 的报错打印输出长达数页，它详细描述了一个程序中的所有句法错误，通常会从开头就遗漏的一个关键标点开始。

1964 年，我读过一本 BASIC 手册。它的副标题是“设计用于达特茅斯（Dartmouth）分时系统的初等代数语言”。那么，什么是代数语言呢？

Tom：噢，由于我们都是数学家，因此，这种语言中的一些地方就自然而然地看起来像数学上的，例如数字求幂等。而且，我们添加的函数也是数学上的，比如正弦函数和余弦函数等，因为我们想让学生使用 BASIC 程序来实现计算方法。因此，和当时开发的 COBOL 等其他语言相比，BASIC 语言显然更适合数值计算，这些语言的关注焦点不同。

当时，我们考查了 FORTRAN 语言。访问任何一台 IBM 大型计算机上的 FORTRAN，都使用 80 列穿孔卡作为记录媒介。我们通过电传打字机将一台计算机引入了校园，把电传打字机用作计算机的输入，因为它们与电话线兼容，而且我们想用电话线将校园内不同位置的终端连接到中心计算机上。所有这些工作都是使用原本设计用于通信的机器完成的，比如电传打字通信、存储并转发消息等。这样一来，我们就废除了穿孔卡。

我们想做的第二件事是，放弃原先强制用户使用的穿孔卡，这种卡上必须打上特定的列，我们想在形式上多少自由一些，比如能够使用电传打字机键盘输入。顺便说一句，它正好是一个标准打字机键盘，不过上面只有大写字母。

这种语言的格式就是这样出现的，也就是易于输入，实际上，最初它是空格无关的。你是否输入空格并没有

任何区别，因为该语言的最初设计是无论你输入什么，计算机都会正确解释，不管它有没有空格。这是因为某些人（特别是大学教员）不能熟练地输入。

由于空格的不敏感性，空格由此进入了一些早期的 BASIC 个人计算机版本，并导致了对输入内容非常有趣的异常解释。

在达特茅斯中，这样做根本没有什么歧义。多年以后，随着语言的发展，才明确提出需要空格；一个变量名的结尾必须是一个空格或一个符号。

BASIC 的一位批评者说，它是设计用于教学的一种语言；只要你开始编写大程序，它们就会变成一团乱麻。您对此怎么看呢？

Tom：持这种说法的人，肯定有很多年没有关注 BASIC 的发展了。BASIC 并不是尚处在襁褓中的语言。我个人就使用 True BASIC 编写了 10 000 行和 20 000 行程序，同时，它的扩展性非常好，我能够编写出 30 000 或 40 000 行程序，不会有任何问题，而且它也不会导致运行时效率低下。

该语言的实现是与语言的设计相分离的。

语言的设计是用户为了完成自己的工作而必须输入的东西。一旦你提供了库，你就可以做你想做的任何事情。至于说它是否支持无限大规模的程序，这是语言实现的问题，而且 True BASIC 可以支持这种程序。

这与其他版本的 BASIC 不同。例如 Microsoft BASIC 和 Visual Basic 有一些限制，这些语言都是基于 BASIC 的。已经广为流传的其他版本的 BASIC 有其他限制，不过那些都是实现的限制，而不是语言设计的限制。

要编写大程序，您可以利用 True BASIC 的哪一个特性呢？

Tom：只有一个特性，那就是封装或模块。我们把封装结构称为模块。

信不信由你，实际上，在解散之前，BASIC 委员会已经对 BASIC 进行了标准化。这事儿发生在 True BASIC 的早期阶段。这种特性被添加到语言标准中，大概是在 1990 年和 1991 年左右。

现代计算机内存很大，芯片速度很快，因此实现这些特性没有问题。

即使您现在回到编译器的两遍编译时期。

Tom：连接器也是使用 True BASIC 编写的。实际上，它是 True BASIC 的版本，或者是 True BASIC 的简化版本。它被编译成这个 B 代码，就像 Pascal P 代码那样。

为了真正地实现连接，你执行那些 B 代码指令，执行 B 代码指令的解释器速度也非常快。True BASIC 就像最初的 BASIC 那样，也被编译。最初的 BASIC 被一步编译成直接的机器指令。在 True BASIC 中，我们编译成 B 代码，而且 B 代码非常简单，因此，就像现在那样，通过 C 编写的一个非常快的循环执行 B 代码，它最初是为 DOS 平台编写的。

它的速度很快。虽然它的速度还赶不上为速度而设计的语言，不过它也是相当快的。如我所言，B 代码中有二地址指令，因此它的速度很快。

在早期，解释并没有降低速度，因为我们必须用软件处理浮点。我们一直坚持 True BASIC 和最初的 Dartmouth BASIC 能处理双位置（double-position）数字，以致于 99%的用户不须要担心精度问题。当然，我们现在使用的是 IEEE 标准，所有芯片都自动支持这个标准。

您认为在设计用于教学和用于构建专业软件的语言之间，其唯一区别是前者更易于学习吗？

Tom：不，那只不过是开发方式而已。C 语言生逢其时，而且能够访问硬件。现在，无处不在的面向对象语言，它们教的是什么和专业程序员要干什么，都是环境的衍生物。那些语言非常难学。

它意味着：使用这些衍生语言的人们、受过专业训练的人们，以及编程团队成员可以创建更加复杂的应用程序，这些程序经常处理电影、声音及诸如此类的事情。使用 Objective-C 这样的面向对象语言更易于完成这些工作，不过，如果你的目标并非如此，你只想编写一个大应用程序，可以使用脱胎于 Dartmouth BASIC 的 True BASIC。

84

让一种编程语言更容易使用的最终目标是什么？我们能否构建出一种简单的语言，让每一台计算机用户都能够自己编写程序？

Tom：不，我们基于 Dartmouth BASIC 的许多功能，现在可以由电子表格等其他应用程序来处理。你可以使用电子表格进行非常复杂的计算。此外，我们考虑的一些数学应用程序，现在可以使用专业团体提供的程序库来完成。

编程语言的细节实际上并不重要，因为你可以在一天之内学习一种新语言。如果有合适的文档，学习一种新语言就会非常容易。我并没有看到任何声称完美的新语言。如果没有瞄准某个特定领域，你就不会有好语言；这是一种自相矛盾的观点！这就像在问什么是全世界最好的口语和书面语？是意大利语？英语？还是别的什么语言？你能定义一个吗？不能，因为所有的书面语和口语来源于语言使用地的生活，因此并没有什么完美的语言。同理，也不会有什么完美的编程语言。

您的意思是人们会先编写一百个很小的程序，然后就称自己为程序员？

Tom：那就是我们的目的，不过奇怪的是，随着语言的发展，没有变得太复杂，它变得可能编写 10 000 行程序。这是因为我们秉承了简单的理念。我们的整个想法是，分时的诀窍是周转时间非常快，你不必担心优化程序，这一点你也看到了。你担心优化人的时间。

在我们创建 BASIC 的数年以前，我就有过为美国麻省理工学院的计算机编写程序的经验。当时使用的是 IBM 704 的符号汇编程序 SAP。我设法编写这个程序，并完成每一件有意义的事情。我使用传感指示灯来优化它，因此不会重复不必要的计算。我做了其中最重要的事。噢，又出了讨厌的问题，我花了一个月的时间来查找它的故障原因，因为我每两周检查修改一次。周转时间是两周。

我不知道在这个过程中究竟花掉了多少计算机时间。在下一年推出 FORTRAN 时，我转用 FORTRAN 编写程序，我觉得总共花了五分钟的计算机时间。

将优化和编码放在一起是绝对错误的。你可别这么干。你只需要优化必须优化的部分，而且可以稍后来做。更高级的语言会自动优化计算机时间，因为你出错很少。

这一点我很少听到。

85

Tom：计算机科学家在这方面是有点迟钝。我们是计算机程序员时，会把精力集中在编程的错综复杂而又魅力无穷的小细节上，并没有从更广泛的工程观点来试图优化整个系统。你总是在设法优化位和字节。

无论如何，那只是一个编辑的评价。我无法确认能支持它。

硬件的发展会影响这种语言的发展吗？

Tom：不会影响，因为我们认为这种语言会阻止人们去了解硬件。我们将 BASIC 设计成硬件无关的；后来在语言或特性中添加的东西，不会对硬件产生影响。

对于某些早期的个人计算机版本的 BASIC 来说，并不是这样，因为那些 BASIC 是较为松散地以 Dartmouth BASIC 为基础的。例如，在一种个人计算机版本的 BASIC 中，它们有一种方式来设置或询问一个特定存储单元的内容。在自己的 Dartmouth BASIC 中，我们从来没这么干过。那些个人计算机 BASIC 当然会极其依赖硬件的性能，而且，其语言的设计会影响到它可用的硬件。

如果你同 Microsoft BASIC 的设计人员聊这个问题，他们会说，“没错，语言的特性会受到硬件的影响，不过，最初的 Dartmouth BASIC 并不是这样的”。

您选择使用浮点来执行所有的运算，这样对用户来说会更容易一些。对于现代编程语言处理数字的方式，您持何种观点？我们应该改用任意精度的精确数字表示形式（您将它们看作一种“数字数组”）吗？

Tom：数字表示有很多方式。当时的大多数语言，还有现代语言，反映了数字表示类型的可用性，现在的硬件已经支持了这些数字表示方法。

例如，如果现在你使用 C 语言编程，就会有与硬件可用的数值表示相对应的数字类型，比如单精度浮点、双精度浮点、单精度整数和双精度整数等。那些是 C 语言的不同方面，因为它被设计与硬件打交道，因此，无论计算机采用什么数字表示法，它们都必须提供访问。

现在，在计算机中可以表示哪些数字？噢，在一个固定长度的二进制数字中，二进制位（大多数计算机使用的是二进制位）至少是一个有限位（finite number）十进制数字，你可以表示的类型和数字会受到一些限制，而且大家都知道，这样会导致特定类型的舍入误差。

有些语言可以存取无限精度的数字，例如 300 个小数位，不过它们的办法是使用软件，将非常大的数字表示成潜在无限位的数字数组，不过，这都是通过软件完成的，因此速度非常慢。

在 BASIC 中，我们的方式可以简单地说成：一个数就是一个数，“3”是一个数，“1.5”也是一个数。这些区别并不会让我们的学生感到困扰；无论给的是什么数，我们都会尽量用计算机提供的浮点硬件来表示这个数字。

这里要说的是，首先考虑的是用什么计算机（当然，我们在 1964 年最终用的是 GE 计算机），我们坚持要用带有浮点硬件的计算机，因为我们不想在软件运算上浪费时间，因此，这就是我们要那样表示数字的原因。当然，这样一来，肯定会有某种不精确性，不过，那恰恰是你必须接受的代价。

GOTO 和 GOSUB 语句只是在当时给定的硬件条件下的一种选择吗？现代编程语言也应该支持这些语句吗？

Tom：我认为这不是硬件的问题；它与此无关。

某些结构化语言需要它，不过那是在过去，20 年或 30 年前，我的确认为这不是一个问题。

当时它之所以重要，是因为那是人们用机器语言和汇编语言编写计算机程序的方式。我们设计 BASIC 时，还没有形成结构化编程的思想；同时，我们的 BASIC 模仿了 FORTRAN，而 FORTRAN 也有 GOTO 语句。

在 BASIC 的演进期间，您在考虑给这种语言添加新特性时，使用的是什么准则？

Tom：噢，当时是根据需要添加的——没什么理论可言。

例如，BASIC 在 1964 年的早期问世之后，我们做的工作之一就是增加处理非数值信息、字符串信息的功能。我们允许使用字符串，因此，人们在编写玩游戏之类的程序时，可以键入“yes”或“no”，而不是“1”或“0”。最初的 BASIC 中，“1”代表“yes”，“0”代表“no”，不过，我们很快就增加了处理字符串的功能。这么做正是因为确有需要。

5.2 编译器设计

您在编写 BASIC 第 1 版时，能够编写出一个单遍编译器，而其他所有人用的都是多遍编译器。您是怎么做到的？

Tom：如果一种语言的设计相对简单，那它就会非常简单。在这方面，很多语言都是很简单的。每件事都很清楚，而且对于我们所说的“遍”和不完全的“遍”，我们必须要做的唯一一件事就是把它填写好（filling in），用于正向传递。那就是真正妨碍一个完全的单遍编译器的唯一因素。

在一个程序的前一百行，您使用 GOTO 转到前一千行的某一个地方。这是连接阶段。

Tom：那就是我们要做的。它等同于连接链表。现在，我们实际上在计算机汇编语言中已经不再使用连接链表结构，不过当时可是缺它不可的。它可能已经建立了一些表，后来填入了地址。

那么，您能同时分析和生成代码吗？

Tom：我能。为了第一次能运行下来，该语言的其他部分故意做得很简单，以致可以完成单遍分析。换句话说，是对变量名做了很多限制。一个字母或一个字母后面接着一个数字，还有数组名称，一维和二维数组一直是单个字母后面接着一个左圆括号。分析也没什么意义。在那里没有查表。此外，我们采用的是简单策略：单个字母，或者单个字母后面接着一个数字，也就是为你提供了 26×11 个变量名。我们预先分配了空间，如果变量可以有值并被赋值的话，这些固定空间就用于存储那些变量值。

我们甚至没有使用符号表。

您需要变量声明吗？

Tom：不用，绝对不用。事实上，数组一直是单个字母后面接着左圆括号，这就是声明。让我看一下是否能正确地记住这个。假设你使用一个数组，如 `a(3)`，那么它自动成为一个数组。哦，我们来看一下，我认为它是从 0 到 10。换句话说，它是自动默认的声明，而且从 0 开始。作为数学家，在表示一个多项式的系数时，第一个下标就会是 0。

您觉得它易于实现吗？

Tom：实现并没什么意义。事实上，编写编译器的很多工作根本不难。即使是后来广为流传的更先进的 BASIC 版本，它使用了一个符号表查找，不过它也没那么难。

它对最优化有负面影响。

Tom：我们并不担心最优化问题，因为，当时大学老师和学生编写的程序，99%都是极小的、毫不起眼的小程序。对它来说，优化没有任何意义。

您说过多态性意味着运行时解释。

Tom：我相信那是对的，不过没人就这句话向我发出诘难，因为我还没有讨论它。多态性意味着你编写一个特定的程序，而且它的行为会因运算的数据而异。现在，如果你没有把它作为一个源程序拉进来，那么执行时这段程序就不知道在干什么，直到它实际上开始执行为止，这就是运行时解释。我这么做，错了吗？

考虑一下 Smalltalk，其中有争议的是您有源代码可用。如果您确实决定要后期绑定，这算作运行时绑定吗？

Tom：这就是一个难题。绑定分为三种：前期绑定、后期绑定和运行时绑定。它确实是个难题，而且，我想你可以找到避开这个难题的方法。

例如，假设你编写一个排序程序。如果你对数字排序，比较哪个数小，这是很明显的。如果你对字符串排序，那么它并不是很明显，因为你不知道是要进行 ASCII 排序、词典排序，还是其他排序。

如果你在编写一个排序程序，知道自己想要哪一种排序，那么你就清楚如何进行比较。如果你在编写一个通用的排序程序，那么你必须调用一个子程序或采取类似的措施，来判定项目 A 是否小于项目 B，无论这些项目是什么。如果你试图对记录键之类的东西排序，那么无论你对什么排序，都必须知道它的顺序。它们可能是不同类型的东西，有时，它们可能会是字符串，不过还要考虑这种可能性有多大。你在编写排序算法时，根本不知道要对什么排序，这意味着必须在以后再放进去。当然，如果要在运行时完成这项工作，那么它就会在运行时解释。

这样做效率会很高，不要误会我的意思，因为你可以有一个小程序，一个子程序，然后大家都必须在子程序中这么做，为要排序的元素编写排序规则。不过，这不是自动完成的。

您不会凭空获得多态性。您必须编写多态变量。

Tom：有些人不得不担心这个问题。

使用面向对象的人们还经常会谈到继承。如果你要在语言中键入数据，那是最重要的。我读过很多有关“面向对象”图书的介绍，它们都谈到了一个编写循环程序的家伙。别人可能是为了其他目的而使用它，不过这非常罕见。关于那一类东西，我总感觉问题在于，如果你想要编写一个足够通用的程序，让其他人也可以使用，那么你必须去掉文档，并使它可用。我想那肯定会有一大堆要考虑的问题，你差不多是在编写一个带有文档的完整应用程序。

对于我编的那种程序，人们评价得过高了。我不了解行业里的情况。那是另外一件事。

您会把“很方便的低成本代码重用”的思想称为“过早通用化”吗？

Tom：这种思想可能与编程专业有关，不过，它和更多的可能编写程序的非专业人员没有什么关系。事实上，大多数人现在并不编写程序。以前的很多事，常常要编写程序才能干，现在都可以买一个应用程序来完成，你可以把它放到一个电子表格中，或者采用其他方式。有了非专业程序员之后，其他领域的人们大都不再编写程序了。

在美国的中学进行的编程启蒙教育，其中有一门《计算机科学》，是大学预修课程。对此，我的困惑之一就是它实在太复杂了。我不知道现在教的是什么语言，我还没有看过。

我曾经考虑过如何使用 BASIC 来开设计算机科学方面的大学入门课程。除了处理指针和分配存储器之外，它几乎能够做入门级计算机科学课程要做的所有事情。这有点复杂。如果你使用 Pascal 语言，可能会不得不陷入指针和分配存储器的泥淖之中，而此时人们甚至还不知道什么是计算机程序，不过那与本文无关。我从未强人所难地推销我的观点，我是单枪匹马地以寡敌众。

人们开始认识到，除非你是在编写虚拟机，否则，就不必再去处理分配内存和指针。编写编译器的人要这么干，不过这就是我们的工作。

Tom：把它交给编译器干吧；你没有必要干这些。

True BASIC 为我们提供了可移植性。这是几个年轻人设计的，他们确实很有才华。当时，我正好在这家公司工作，做应用程序编程。他们以 Pascal 的 P 代码的方式，设计了一个中间语言。它并不是两个地址，而是三个地址，因为实际上 BASIC 的所有指令都是三个地址，LET A=3，这已经被证实了。总共有三件事：一个操作码和两个地址。然后，他们使用 BASIC 自身，构建了一种编译器，并为真正编译该编译器提供了非常简单的支持。编译器自身是用 True BASIC 编写的，而且，它可以运行在任何带有 True BASIC 引擎的机器上，我们把这种引擎称为解释器。

语言是在执行级（execution level）解释的，而不是在扫描级（scanning level）上。因此，程序执行有三个阶段。首先是编译器阶段，然后是连接/加载阶段，最后是执行阶段。不过，用户不会知道这些。用户只需要输入“运行”、按下“运行”键或什么东西，然后砰地一敲，就万事大吉了。

编译代码也是机器无关的。它可以实现跨边界传输。

语言环境确实相当复杂。我们使用了多种平台，目前就有四五种不同的平台，不过，大多数平台的生存时间都很短，很快就被淘汰了。现在只剩下两三种主要平台：Unix、Microsoft 和 Apple，对于我们来说，Apple 是一个有趣的平台，因为达特茅斯（Dartmouth）一直是一所 Apple 支持的学校。

事实表明，平台移植非常令人讨厌。支持窗口、小技巧及按钮等诸如此类的东西，平台对此的实现方式各不相同，你必须开始认真考虑这些实现的细节。有时候，它们是在非常底层的平台上做这些事情，因此，一切都必须由你自己来构建。

最初的老版 Mac 有一个 Mac 工具箱。我们一度使用过一个分层软件——来自美国科罗拉多博尔德（Boulder）的 XVT，据称它是专门针对 Windows 和经典的 Mac 的软件。除此之外，我们还能得到另外一些好处。在公司解散之前，程序员推出了一个 Windows 版本；它可以直接用在 Windows 应用程序环境中。

麻烦在于，我们只有一个程序员，事无巨细都要他来负责，这需要一些时间，而且，新版操作系统推出以后，要带着新 bug 运行，你还得捕获到这些 bug。对于我们这样的小团队来说，这几乎是不可能的。我们一度有过三位程序员，后来减为两位，再后变成一位。对于这一位程序员来说，要处理的事情实在是太多了。

底层的代码，主要是 C 代码，里面包含了大量的`#ifdefs`。

5.3 语言和编程实践

Language and Programming Practice

在一种语言的设计和使用该语言编写的软件设计之间有什么联系？

Tom：二者联系非常紧密。大多数语言设计用于具体类型的软件。APT 就是一个很好的例子，它是控制自动编程工具（Automatic Programmed Tool）的一种语言。

在早期阶段，您添加了用于注释的 REM 语句。这么多年过去了，您对注释和软件文档的观点改变了吗？

Tom：没有改变，这是一种自防护机制。我用 True BASIC 编写程序时，会添加注释，提醒我在编写代码时是如何考虑的。我认为注释很有用，而且它的作用会因编写程序的类型而异——你是在一个团队内工作，还是没有其他人会读到你的代码。我只支持它们在必要的范围内进行注释。

对于采用团队方式编写软件的人们，您有何建议？

Tom：没有什么建议，因为我们从来没这么干过。我们在自己环境中做的所有软件都是单枪匹马干出来的。在 True BASIC 中，我们可能会有两三个人在编写代码，不过他们实际上做的是完全不同的几块。我恰恰没有一点儿团队工作经验。

您有一台分时机器，您建议：用户应该事先计划好他们在电传打字机上的会话内容。有句格言是：输入并不能代替思考。这在今天还成立吗？

Tom：我认为还是要思考。一家大公司在开发一款新软件产品时，事先要进行许多思考，因此我认为它依然成立。

就我自己而言，我在做事之前不会考虑太多，而是先开始编写程序再说。随后，发现它并不是令人很满意，我会完全推倒重来。这就等于是思考。通常，我开始编码只是想看看哪里存在问题，然后再扔掉那个版本。

想清楚你正在干什么是重要的，而且是非常重要的。我并不确认，不过我认为 Richard Hamming 说过“输入并不能代替思考”。那是在计算的早期阶段，几乎没有人知道应该怎么做，因此会有很多建议在坊间广为流传。

学习新的编程语言的最佳方式是什么？

Tom：一个人一旦了解了如何编程及相关概念（也就是如何分配存储器），如果又读了参考手册和良好实现（也就是编译器），学习新语言就会非常简单。我经常会这么干。

参加学习班简直就是在浪费时间。

每一位称职的程序员在职业生涯内都会学习很多语言。（我很可能已经用过 20 多种语言。）学习新语言的方式是阅读手册。除了少数语言之外，大多数编程语言在结构和运行方式方面非常类似，如果有比较好的手册可用，那么，这种新语言就会非常容易学习。

一旦你克服了行业术语的障碍（多态性意味着什么？），事情其实就会变得相当简单。

现在的编程风格有一个问题：它们没有手册，仅有一个接口构建工具而已。按照它们的这种设计，程序员就不必逐个字母键入很多指令，就像工程师的 CAD 和 CAM 工具那样。对于像我这样的老程序员来说，这一点非常讨厌——我想逐个字母键入所有的代码。

过去也曾试图通过提供宏（比如关键字 LET 的单个按键）来简化键入操作（方便笨拙的打字员或学生），不过它们从未流行起来。

现在，我在尝试学习一种想像中的“面向对象”语言，它没有参考手册，至少是我没有看到。它所提供的手册，开发了一些看起来毫不起眼的小例子，并用了 90%左右的篇幅来强调 OOP 是多么优秀的“流派”。我的朋友参加了一门 C++培训课，从教学的角度来看，这门课简直就是一场灾难。我的观点是，对于社区来讲，OOP 就是一个最大的谎言。所有的语言最初都是设计用于特定的用户群，比如，FORTRAN 用于扩展的数值计算等。OOP 的设计，竟会使它的客户声称“内部”是其精华所在。实际上，OOP 最重要的一个特性是：子程序和数据封装，这是几十年前就已经有了的一种方式。其余的都不值一提。

5.4 语言设计

Language Design

您认为 Microsoft 的 Visual Basic 目前是一个成熟的面向对象语言吗？如果是，您对它这方面满意（假设不考虑面向对象模式）吗？

Tom：我不知道。通过几个简单的试验，我发现 Visual Basic 相对容易使用。我想，Microsoft 之外的任何人都会将 VB 定义为一个 OOL。事实上，True BASIC 在面向对象方面和 VB 一样，或许还有过之而无不及。True BASIC 包括了模块，它是子程序和数据的集合；它们提供了单个最重要的 OOP 特性，也就是数据封装。（True BASIC 没有继承类型，因为它没有用户自定义的类型，除了数组维数以外。事实上，几乎没有任何语言具备多态性，它意味着运行时解释）。

您提到 Visual Basic 与 True BASIC 相比，有很大的局限性。您的意思是 Visual Basic 与您的模块化系统相比缺少点什么吗？

Tom：我不知道。我只是编写了一些范例程序——没错，我没有使用 Visual Basic 编写任何程序，我只是证明自己能够使用它。和我曾经使用的那些语言相比，它的用户接口相当简单。Visual Basic 是老款 Microsoft BASIC，而且号称是面向对象的，但是，实际上并非如此，它只不过是添加了前端接口开发工具而已。

对于一些构建用于视频和音频的更大的系统，您还做了一个有趣的评价。您说它更易于创建一个复杂的应用程序，比如用 Objective-C 这样的面向对象语言创建的应用程序。

Tom：是的，很可能是因为语言环境足以促成这样的结果。

我试图马上学会 Objective-C，结果没有成功，不过至少已经建立起了环境。如果知道在做什么，你可以用某合理方式访问该平台上的一切，包括视频和音频灯。我还没有试过，因此我不知道它有多难，不过它包含在语言开发环境之中。

它并不是这种语言自身必需的特性；它是这种语言的一种环境特性。

Tom：实际上，它和这种语言本身并没有什么密切的关联，不过，它和语言环境密切关联。如果目前有很多人在使用某种语言环境，那么，或许有上百名程序员在背后保障它的正常运行。

对于现在和可预见的将来开发计算机系统的人们，在发明创新、进一步开发，以及采用这种语言方面，您有什么经验和教训要说吗？

Tom：没有。

从 20 世纪 60 年代早期起，我就知道了 Burroughs 5500 计算机。它的硬件设计原则是，后进先出堆栈应用程序（比如 Algol 编译器）效率应该更高。

今天，设计趋势看起来有了变化，正向着 RISC 机器（精简指令集计算机）方向发展。

对于大多数编程语言来说，没有什么特殊需要。部分原因在于，计算机速度非常快，而且是越来越快，而且特定语言的编译和处理时间也不是问题。

反过来可能是对的。例如，为了处理大数组，你可以构建一台数组处理计算机，它可能需要你开发一种合适的编程语言。

如果今天您必须创建一种全新的教学编程语言，它会和 BASIC 有多大程度的类似呢？

Tom：它们会非常类似，因为我们遵从的原理仍然是正确、有效的。例如，我们试图让一种语言易于记忆，即便一个人好久不用这种语言了，仍然能够坐下来就用。

我们尽量不让一种语言的需求深奥难懂。例如，在 FORTRAN 中，如果想打印一个数，你必须使用一种格式语句，而且要明确地说明如何打印。这就是一个深奥难懂的问题，特别是人们不经常使用该语言时，需要专门学习，在 BASIC 中，我们就只以我们认为的最佳方式来打印数字。事实上，如果是一个整数数字，即使使用内部浮点，也能将它作为没有小数点的整数值打印出来。如果想键入一个数，你不必须担心格式；你尽管放心编写自己喜爱的公式，而且它的运行没有问题——你在输入数据时，不会受限于一些特定格式。

大多数情况下，这是由电子制表软件来完成的；它们对此非常擅长。如果你想以特定列显示的定点数字输出，你就可以指定，或者你并不在乎使用通用格式。我认为，这差不多就是我们所要的 BASIC 风格。我们允许字符输入，而且是以一种非常简单的方式输入——你只需键入即可，不必拘泥于什么规则。

在 20 世纪 70 年代~80 年代间，我们在 Dartmouth（达特茅斯）开发了结构化编程版本的 BASIC，而且还添加了具有面向对象编程功能的元素。我们不会什么事都与众不同。

关于面向对象，您喜欢封装吗？

Tom：绝对是这样。我经常说，封装占了对象编程的 70%，不过，我认为应该把它改成 90%。把程序及其数据融合在一起，这就是主要工作。它确实非常重要。我现在已经不再写程序了，不过在为 True BASIC 等项目工作时，我把自己编写的所有东西都封装起来了。

我们有一种方式，可以将子程序群封装在模块中。它和前面的道理一样，只不过将子程序组合在了一起，如此一来，除了通过它们的调用序列以外，它们就可以与编程的其他方面隔离开来。事实上，它们也有自己的私有数据。这样像隔离各种功能等诸如此类的事情会非常方便。

我曾经问过很多语言和系统设计者一个问题：对于数学形式主义这种概念，他们究竟喜欢到什么程度。采用 Scheme（模式），它可以非常有效地表示 lambda 计算方法。您有六种基本类型，所有的一切都是完美地构建在这个基础之上的。它看起来好像是数学家的方式。

94

Tom：是的，这非常有趣。这是一个有趣的数学问题，不过，如果重新设计一种计算机语言，你并不需要这样做，因为我见过的每一种计算机语言都比它更简单，甚至 FORTRAN 也是这样。Algol 很简单，它使用递归定义，不过相当简单易懂。

我从未研究过编程语言的理论，因此无法做出更多的评价。

您认为谁会使用这种语言，他们必须解决的最大问题是什么？

Tom：好的。对受众来说，最大的问题是要每周都记得用这种语言，不过他们每两周才编写一个程序。我们还想要一种用几个小时左右就能讲清楚的编程语言和系统环境，这样，你就不必再上课学习了。

我和许多同辈就是这样学习编程的。我们在 20 世纪 80 年代初期的 PC（Commodore 64 和 Apple II）上运行 Microsoft BASIC，它们是带有子程序的行 BASIC（line BASIC），而不是其他的什么东西。

Tom：现在是怪事满天飞。实际上，它们还有一些技巧来应对这些问题。例如，我使用 Apple Soft BASIC，我不知道这是 Microsoft 还是其他人设计的，不过这些都是从 Dartmouth BASIC 启动复制而来的。它们引入了多字符变量名的概念，但是没有正确地分析它。如果你碰巧在多字符变量名中嵌入了一个关键字，它应该能把这个关键字去掉。

这是因为对空白字符不敏感吗？

Tom：不是，因为它们声称有多个字符变量名，其实并没有。在这个问题上，它们说了谎。假设有一个多字符变量名，比如 TOT，它们会把 TO 识别为一个关键字。这是一种营销手段。那些语言和小特性是为市场营销而设计的，它们认为采用多字符变量名应该是很好的手段。使用该语言的人们应该设法少用多字符，从而回避这个问题。

这不是通过阅读手册就可以发现的。

Tom：没错，其中的错误没法通过阅读手册来发现。

空白字符的不敏感性又是如何出炉的呢？

Tom：对于这个问题，我唯一知道的就是它已经发布了，采用空格不敏感性的部分原因在于，John Kemeny 是一个蹩脚的打字员。我不知道这是不是真正的原因。我们共同设计了这种语言，不过这个特性是他设计的。

变量名是唯一的，看起来也不像关键字，所以不需要空格。你可以自发地添加或删除空格。

95

一切都在发展变化之中。现在我编程时，仍然有多种广为流传的 True BASIC 版本，它们运行在不同的机器之上。那是我使用的唯一一种语言，而且我使用空格来提高清晰度。

没有避开计算机的限制，对您来说，这完全是一个人为因素吗？

Tom：没错。如果你在编写一个真正的程序，而且以后还要用到它，最重要的就是要确认：你先把它扔掉半年，再回头看的时候是否仍然能够读懂它。选择变量名非常重要，变量名暗示了它们表示的数量，构建结构也很重要；我主要考虑的是：使用单用途子程序，然后给它们起一个可以望文生义的名称。

使用任意长度的多字符变量名。现在所有的计算机语言都是这么做的，即使子程序名称有 20 个字母，它仍然可以表明功能，有人能在半年后回忆起来，并且不会理解错误。

Chuck Moore 说，您在 Forth 中构建了一个词汇表，如果选对词的话，您就可以用领域语言来编写程序。为什么这种思想会出现这么多次？这非常有趣。

Tom：像 BASIC 这样的计算机语言，是供非专业程序员使用的。如果处于某些领域，而且你决定为该领域编写一个应用程序，与那些专业人士相比，你会更喜欢使用一种更简单的编程语言。我在仔细考虑这些广为流传的面向对象语言。

我对它们其中的一种语言有点经验，而且它极为复杂。这是唯一一种我还没有掌握的计算机语言，尽管我已经用过 30 种左右的计算机语言编写程序。

为什么 WYSIWYG 不再需要代码行号了呢？您暗示过，它们仅仅是作为 GOTO 语句的目标而存在的。程序员编辑文件时，还需要代码行号来指定某些行吗？

Tom：绝对不需要（对于最后一个问题）。行数字编辑早就不用了。

WYSIWYG（所见即所得）会如何改变编程方式呢？

Tom：根本不会有任何改变。现在，WYSIWYG 编辑器已经非常成熟，而且，它和所服务的语言紧密相关（在缩进、色彩使用等方面）。

我们今天的计算机科学教育缺失了什么呢？比如，有些人批评没有关注工程等。

Tom：哦，这个我不知道，因为我不知道现在计算机科学都教些什么。我在 15 年前就退休了，而且除了统计学和计算机科学之外，我没有讲过其他课程，因此对该领域如何发展并没有什么想法。

应该如何讲授调试呢？

Tom：噢，未雨绸缪是最佳方案，而且，如果在做之前能提前想好，就更好了。

我们以前的一个学生成为了 Apple 会员，并为 Apple 做了非常重要的一些工作，后来他退休了。在此之前，他在达特茅斯计算机中心工作。他编写了一个 PL/1 编译器，这是一个大项目，他反复检查、仔细考虑，不过直到全部工作完成之前，他从来没有测试过它，也没有运行过它。你要知道，它有 20 000 行或 30 000 行代码，他所做的唯一测试就是读代码。随后，他运行了这个编译器，一次成功！

在整个计算机科学历史上，这是一个奇迹！如果有人编写一个 20 000 行或 30 000 行的程序，而且一次成功，那就是天方夜谭，对不对？但是，他做到了，而且是以这种方式做到的。他孤军奋战，没有参加任何团队。独自工作时的效率总是更高。他对程序的各个部分如何协作非常关注，同时对代码读得很细，这意味着读代码时，真正在干的事情就是模拟计算机将要干的事情，因此逐步检查：这里没有问题，那里也没有问题等。

因此，在他按下“运行”按钮时，程序就能正常工作，那是不切实际的，没有人会做到这一点，不过这也算一种想法。你首先要把 bug 拒之门外，这样才能减少 bug。

许多商业软件现在是 bug 遍布，因为它不是由优秀的程序员编写的，而是由团队编写的，而且功能设计是由市场部门决定的。它必须使用特定的程序、具有特定的功能，而且必须在特定的时间内出炉，因此它必然会充满 bug。对于软件公司来说，大多数用户只是使用计算机的浅显特性，并不会被很多 bug 所困扰。

融入语言或添加到库中，二者孰优孰劣，您能画一条界线吗？

Tom：好的，我们对此也非常关注。对于那些只有少数用户想要知道的有些深奥的东西，我们就会把它放到库中。这些年来，这种语言就是这么开发的。因此，我们开发了一个库，它有很多功能；在现代版 BASIC（也就是 True BASIC）中，我们使用一个子程序库来访问面向对象编程的对象，比如按钮、对话框之类的对象等。我们使用一个库程序，语言本身无法访问这类东西，你必须调用一个子程序来做这些事情，而且子程序会包含在许多库中。我们从一开始就认为做这些事会非常困难。

团队编写软件时，通常会构建公共库供大家使用。对于在 True BASIC 中构建这样的库，您有何建议？

Tom：噢，我本人也写过库，不过在设法保持简单之外，我并没有什么特殊的建议。尽人皆知的技巧是尽量简单并设法避免引入 bug。

例如，单用途子程序很重要。不要因为它听起来像一个好主意，就让子程序兼职干别的事，它可能会带来的灾难性的副作用。编写库有许多方式，使用这些库可以减少，或者消除未来的 bug 或错误。要减少错误，可以使用很多著名的技术，不过，我不知道业内究竟有多少人会使用它们，因为业内主要遵照市场营销部门的旨意来编写程序。

5.5 工作目标

在工作方面，您是如何定义成功的呢？

Tom：多年以来，由于我们的工作（因为它非常开放，我们并不限制参与项目的学生访问），Dartmouth（达特茅斯）成为全世界声誉最佳的计算机学院之一。参访人员来自俄罗斯、日本和其他地方，他们都是慕名而来的。这是在个人计算机普及之前。现在是人手一台个人计算机，因此它不再是一个问题。不过，在那个年代，它是一个非常令人感兴趣的问题。只要学生愿意，我们就会允许任何学生在任何时间进行任何计算，而不需要事先批准。这在当时别树一帜。它给 Dartmouth（达特茅斯）带来了持续 10 年或 15 年的声誉，并让 Dartmouth 在筹款、吸引学生及招募员工方面占据优势地位。

我们的成功还包括来到达特茅斯学习计算的很多学生，由于他们知道如何去做计算，从而成就了非凡的业绩。这些人不在公司的计算中心工作，而是工作于其他部门。有相当多的 Dartmouth 毕业生成为了百万富翁，仅仅因为他们知道如何使用计算机！

那就是我们成功的主要标志。

年轻人应该从您的经验中学习什么呢？

Tom：他们应该关注软件的最终用户，也就是将要使用他们开发的软件的人。我们现在使用的很多应用程序确实很难用。

我知道，很多年前，Microsoft 的人通过“Bob”程序或类似的程序引入了用户方便性的概念，不过他们并不理解它；他们认为“用户友好”意味着“施恩于人”，好像你同一个孩子谈话一样。那不是用户友好的含义。

86 我担心整个行业都不知道用户友好是什么意思。我不知道现在从事计算机科学的人们能否理解这个词的含义。

我的建议是去关注将要使用你的软件的用户。

我们应该构建一个像 BASIC 那样易于学习的接口吗？

Tom：没错，易于学习，便于在各类手册中描述，而且在使用时或多或少地符合你的期望，没有出乎意料的事情发生。

您说独自工作时效率更高，我想知道您说的效率是什么意思。

Tom：我的意思非常简单。我认为这有很多证据支持。我平生从没有在一个编程团队中工作过。我一直在说：“好的，这件事值得一干，我得为它编写一个程序”。作为单干的程序员，我编写的所有东西都能自己一个人掌控。

我用过别人编写的程序，但是我从未加入一个开发团队。哦，我已经编写了不知道多少行程序，不过我仍然使用 10 000 行长的程序干成了很多事。它们易于编写、易于调试。如果发现一个并不喜欢的特性，它很容易修改——如果是你自己编写的，也不必写备忘录。如果它有文档，我也不介意写文档。我的东西大多数没有文档，因为是供我自己使用的，不过，我对 Fred Brooks 曾经在他的《Mythical Man-Month》一书中就编程所写的内容深信不疑。

我很喜欢一个程序员说编写一个程序估计要花多长时间。一名程序员每天编写三行有文档的代码或类似的代码，发现自己的应用程序太长了，却无法判断错在哪里。他们发现原因在于程序员每周只工作 20 小时。事实上，他们在公司工作了 40 小时，不过有 20 小时花在了辅助性的事情上，以及开员工会议等。

我最讨厌的就是员工会议。有时候，它们是绝对必要的。我还记得最初为 GE225、GE235 计算机开发 BASIC 的情景。学生程序员应该一周开一个小时左右的会议。John Kemeny 会主持会议，而且就像他喜欢说的那样，他会做出所有并不重要的决定，比如赞成谁做调度算法等，不过学生们会做出所有重要的决定，比如说哪一位用于什么功能等。

那时候，我们有一个双机环境，学生们是一人一台机器。他们是二年级学生，必须要在一起工作、一起沟通。他们基本上都在做自己的工作。无论我们编写一个编译器或一个编辑器，都是一个人干。

您是在说一个学生时作出这个评论的，这名学生编写了 PL/1 编译器，而且第一次运行就能正常工作。

99

Tom：你说的是 Phil Koch，他还是一名 Apple 会员。现在，他已经从 Apple 退休了，住在美国缅因州（Maine）。他是一位令人惊讶的程序员，在那个编译器上花了很长时间，而且非常认真地阅读了代码。

如果您愿意向人们介绍从这些年来丰富阅历中得到的一条经验，那会是什么呢？

Tom：让用户很容易使用你的软件。

如果你愿意，可以说是用户友好，不过依我看来，部分问题是行业已经把用户友好定义为屈尊相从（condescending）。用户方便性的真正问题是，无论你做什么应用，都会有合理的默认处理，因此，新上手的人不需要学习所有的变种和可能的自由度。人们可以坐下来并开始工作，然后，如果他们想要有些新意，也可以相对容易地实现。

为了做到这些，你必须对用户群的基础情况有一些了解。

我经常使用 Microsoft Word，不过按照我的标准，它根本谈不上用户友好。后来，大约 10 年前，Microsoft 向市场发布了 Bob（译注1），那就是错误的想法。他们并没有理解用户友好的真正含义。

我认为，一些应用程序是用户友好的，不过现在的主要问题是网站设计。设计网站的人，有时候干得不错，有时候则干得不怎么样。如果你访问一个网站，却无法知道如何找到更多的信息，那就是一个糟糕的设计。

有些问题很难讲。

美国马里兰大学计算机科学人因专家 Ben Shniederman（注 1），实际上做过一些研究，结果表明：从用户方便性的角度来看，我们在 BASIC 中为 DO、LOOP 和 IF 选择的结构，比正在使用的其他语言更容易，比如 Algol 或 Pascal 会使用分号来结束一条语句。

人们通常不会使用分号来结束语句，因此你必须专门学习。例如，我记得在 FORTRAN 中，有些地方需要逗号，而有些地方又不需要。结果，美国佛罗里达州（Florida）空间站程序中的 bug 出现故障，因为丢了一个逗号而损毁了一枚火箭。我认为 Ed Tufte 实际上有文档。尽量远离那些有可能含混不清的东西。

我一直对外界说，由于没有让别人的计算机实现用户友好，我和 Kemeny 确实失败了，不过，我们仍然为自己的学生做了一件好事，因为近 20 年来，他们出去之后很容易在行业内找到工作，因为他们知道如何做事。

100

这也是一种很好的成功。

Tom：如果你是一位老师，其实这才是正事。

译注1：Bob 是微软于 1995 年推出的一款产品，并被定为用在 Windows 3.1 中的下一代界面。它是微软首次尝试开发互动性更强、更自然的用户界面。Microsoft Bob 的推出主要是为了满足初级计算机用户的需求，虽然有着很好的创意，但是过于简单，只讲解了如何使用计算机，结果在没有市场的情况下被淘汰了。

注 1：Shneiderman，B“从小学编程：前提、概念和结果”，计算教师，第 5 卷：14–17（1985）。

第 6 章 101

AWK

有许多体现 Unix 设计哲学的小工具，它们的组合功能非常强大，显然，AWK 编程语言就是这样一种语言。它的创始人（Al Aho、Peter Weinberger 和 Brian Kernighan）把它描述成一种语法驱动的模式匹配语言。它的句法简单易懂，并且明智地选择了有用的特性，这样一来，它就很容易分割处理文本，而无须懂得分析器、语法和有限自动机。尽管很多通用语言也汲取了它的灵感，比如 Perl 等，不过，所有的现代 Unix 系统中仍然都默认安装了 AWK 语言，并且它们能够安静、有效地连续工作。

6.1 算法生命周期

您是如何定义 AWK 的？

Al Aho：我会说 AWK 是一种易学易用的脚本语言，擅长于日常数据处理的应用程序。

在 AWK 的开发过程中，您发挥了什么作用？

Al：20 世纪 70 年代，我一直在研究高效的分析和字符串模式匹配算法。我和 Brian Kernighan 一直在讨论 grep 的通用化，对于我们所知的许多数据处理应用，我们期望它能够处理更普遍的模式匹配和文本处理。Peter Weinberger 对这个项目很感兴趣并且加盟进来，因此，我们很快就在 1977 年实现了 AWK 第 1 版。

随后几年，这种语言获得了很大的发展，我们的很多同事开始用它来完成各种各样的数据处理任务，其中有很多任务是连我们都未曾预料到的。

AWK 最适合在什么样的背景中应用呢？

Al：我认为，对于简单的常规数据处理应用程序来说，AWK 仍然是无可匹敌的。我们的 AWK 书中给出了很多实例：一行或者两行 AWK 程序就可以完成几十行甚至数百行 C 代码或者 Java 代码才能完成的功能。

人们在设计用 AWK 编写的软件时，应该注意些什么？

Al：AWK 是一种脚本语言，它设计用于为普通数据处理应用编写短程序。我们并不准备用它编写大的应用程序，不过，我们发现人们常常会这么干，因为这种语言非常易于使用。对于大的应用程序来说，我想推荐常见的良好软件工程实践：良好的模块化、良好的变量名以及良好的注释等。这些实践对于短程序来说也是很适用的。

硬件资源的可用性会对程序员的心态有什么影响？

Al：毫无疑问，高速硬件、大容量内存以及优秀的 IDE（集成开发环境）确实会让编程过程变得更加令人愉悦。同时，程序也可能会应用在比以前更大的数据集上。现在，我运行的 AWK 程序要比过去大好几个数量级，这已经是再平常不过了，因此，高速硬件让我成为了生产力更高的用户。

不过，这也是一种折衷平衡：硬件的改进会导致软件系统的规模和复杂度的激增。随着硬件的改进，软件也变得更有用，不过，它也会变得更复杂——我不知道谁会胜出。

使用 AWK 开发算法时，您估计代码会在多大规模的数据上运行？

Al：只要有可能，不管是在最坏的条件下，还是平均条件下，我们实现的算法都会与时间线性相关，通过这种方式，AWK 就可以从容地处理规模越来越大的输入。

我们在不同规模的数据集上测试 AWK，想看一看随着输入规模的增大，AWK 的性能表现会有何变化。我们竭尽所能让实现效率更高，并使用真实数据来测试我们干得到底怎么样。

您考虑过将来数据规模会如何增长吗？

AI：我们设计 AWK 的时候，我认为 MB（2^{20} 字节）级的数据集就已经很庞大了。如果考虑现在因特网的数据规模已经达到了 EB（2^{30} 字节）级，当时我们认为的所谓大数据集与它相比，相差了好几个数量级。当然，即使是线性时间扫描 1TB（2^{40} 字节）的数据也实在是太慢了，因此，必须采用一种全新的方式来处理因特网上的相关数据。

我曾经听说 AWK 被描述为“适合简单数据处理任务的模式匹配语言”。AWK 是在 30 年前创建的，从那时到现在，您认为模式匹配发生了什么样的变化？

AI：过去 30 年间，模式匹配的规模和多样性出现了爆炸式的增长。这些问题的参数显著地增大了：模式变得更加复杂，而且数据集的规模也极大地增长了。今天，我们通常会使用搜索引擎在因特网的所有网页上搜索文本模式。我们还对数据挖掘感兴趣：在庞大的数字图书馆中搜索各种模式，比如基因组数据库和科技档案等。公平地说，字符串模式匹配是计算机科学中最基本的应用之一。

现在，对于这些需求是否有更好的模式匹配算法和实现呢？

AI：在 AWK 中，模式匹配是使用一种快速、紧凑、惰性状态转换结构的算法来实现的，该算法是基于正则表达式，需要用确定型有限自动机来做模式匹配。该算法收录在 Red Dragon 一书中（注 1）。该算法的运行时间，基本上与正则表达式的长度和输入文本的规模成线性关系。这是尽人皆知的正则表达式预期运行时间。对于只有单一关键字或者有限关键字集合的特殊情况，我们可以实现一个 Boyer-Moore 算法或者一个 Aho-Corasick 算法。我们并没有这样做，因为不知道人们在 AWK 程序中使用的正则表达式的特征。

我可能提到过，在软件系统中使用复杂算法也有不利的一面。该算法可能难以令其他人理解（以至于在较长时间之后，连作者自己也可能理解不了）。我曾经在 AWK 中引入了一些复杂的正则表达式模式匹配技术。尽管在 Red Dragon 一书做了有关注解，Brian Kernighan 有一次看了一下我写的那个模式匹配模块，他只在那个模块上用斜体加了个注释：“入此门者，莫存侥幸（abandon all hope, ye who enter here）”（注 2）。结果，无论是 Kernighan 还是 Weinberger 都不愿意碰那部分代码。为那个模块修改 bug 的任务，只能落在我一个人头上了！

6.2 语言设计

您对编程语言的设计者有何建议？

AI：始终牢记你的用户。要让别人觉得用你的工具来解决问题，确实是物有所值。还应满足别人以你的工作为基础创建更加强大的工具的需求。

注 1：Aho, Alfred V 等著。Compilers: Principles, Techniques, and Tools（编译器：原理、技术和工具）（Addison-Wesley, 1986）。

注 2：“入此门者，莫存侥幸。(Lasciate ogne speranza, voi ch'intrate)”是 14 世纪意大利诗人但丁所著《神曲》之地狱门上的题词。

Kernighan 和 Weinberger 对语言设计是如何考虑的？

AI：如果必须选择一个词来描述我们在语言设计中的核心关注点的话，我会说 Kernighan 强调容易学习；Weinberger 强调实现的完美性；而我则强调它的效用。我认为 AWK 兼具了这三种属性。

在考虑效用的情况下，您是如何做出设计决定的？它对您所考虑的设计方式产生了什么样的影响？

AI：我不知道这是有意识的，还是无意识的，不过，毫无疑问，生存下来的都是那些有用的东西。它印证了达尔文的学说。你所创建概念和语言风格，对于解决你感兴趣的问题是有效的，不过，如果它们在解决其他人感兴趣的问题方面表现不佳，它们就会枯萎凋零。正是适者生存创造了应用。我们并不需要保留没有用的语言。

除非我们是艺术史学家，否则，在程序“好看”还是“有用”这两方面就会顾此失彼。

AI：您不能二者兼顾吗？

人们好像是想给它们划条界线，但问题是，无论是作为艺术品，还是工艺品，编程是否是一种创造性劳动呢？

AI：Knuth 作为一位艺术家，当然对编程非常感兴趣。他认为程序应该是“好看”的。我所认识的程序员几乎都一致认为编写的程序应该是优雅的。

一位木匠可能会说，“这是一把椅子。你可以上面坐着，也可以在上面站着，还可以把电话本堆在上面，不过，来看看它优雅的设计，看看它奇妙的连接，看看它出色的雕刻”。它具有很高的艺术性，尽管这是一种实用的工具。

AI：但是，也有简约之美，因此，我们并不需要为了好看而使用各种各样的装饰品或者洛可可式（过分修饰的）架构。

怎样才能成为一名较好的程序员呢？

AI：我的第一个建议是在你编程之前多加思考。其次，我提倡多写代码，交给专家评判，读别人写的好代码，并参与代码检查。如果你有足够的勇气，你应该教你的学生来编写好的代码。

我发现，如果要学习一个课题，教会别人就是再好不过的学习方法。在教别人的过程中，你必须组织材料并且展示出来，通过你的展示，别人会对这个课题有一个清晰的理解。你在教室上课时，学生会问你问题，这时候你的思考方式就与最初不同。你对它的见解深化了，远远超出了此前对它的理解。

对于编程来说，这是千真万确的。如果你教的是编程，学生会问，“我们不能用这种方式来解决吗，我们不能用那种方式来解决吗？”然后你会意识到，“没错，使用程序就会有很多方式来解决那个问题”。你意识到人们的想法是千差万别的，而且，由于他们的想法各异，就会用不同的方式来解决问题，这样一来，对于这个问题的不同解决方式，你就会有一个更好的整体把握。

我确实发现了，在我写的每一本书里，程序都变得越来越高效，也越来越简洁。在我们写 AWK 这本书的这些年里，书里的很多程序都缩短了 50%。这是因为，我们学会了如何比我们最初的设想更加有效地使用 AWK 中的抽象。

105

编写这本书时，您发现 AWK 的设计缺陷了吗？

AI：人们开始用 AWK 来完成许多我们最初没有想到的任务，这时候，这种语言在某些方面暴露出了一些不足，而我们原本就没打算把它设计成一种通用的编程语言。我不想把这些叫做“缺陷”，不过，这也表明了 AWK 是一种专用语言，而不是让人们用它来处理其他应用问题的。

您能解决其中的一些问题吗，或者，您还是强烈反对让 AWK 更加通用？

AI：从创建初始版本到现在，AWK 语言已经经历了 10 年的演化，其中添加了新的结构和新的操作符，而且，它还是一种 pattern-action（模式-动作）型语言，一种意在解决数据处理方面问题的语言。我们并不打算偏离这个领域。

您是如何让用户能够理解语法驱动变换思想的？这些用户可能对有限状态机和叠加（下推）自动机知之甚少或者是一无所知。

AI：毫无疑问，作为一名 AWK 用户，你并不需要了解这些概念。从另外的角度来说，如果你对语言设计和实现很感兴趣，就必须理解有限状态机和上下文无关语法的知识。

lex 或者 yacc 的用户应该懂得上下文无关的语法吗，尽管他们编写程序并不需要这样？

AI：大多数 lex 用户可以不用知道什么是有限状态机。yacc 用户实际上是在写上下文无关语法，因此，从这个观点来看，yacc 用户确实得懂“语法”，不过，这些用户大可不必为了使用 yacc 而成为形式语言专家。

否则，你将不得不承受整页整页的移位/归约冲突错误带来的痛苦。

AI：yacc 的一个有用方面是：因为它能通过语法自动构造一个确定的分析器，并通知编程语言设计者，在他们的语言有一些含混不清或者难以分析的结构。如果不是借助这种工具，他们就不可能注意到这些不当的结构。有了 yacc，语言设计者通常会说，“噢，我没有意识到还有两种方法能解释这个语法结构！”然后，他们会删除或修改那些有问题的结构。优先权和结合中的不明确性很容易用一个简单机制来解决：“对于语言中的这个操作符，我愿意使用这种优先顺序和这种结合顺序”。

您是如何构建一种调试友好（debugging-friendly）型语言的？设计一种语言时，您是如何考虑需要用增加或者删除来辅助调试阶段的特性的？

AI：编程语言的设计趋势是创造一种能够提高软件可靠性和程序员生产率的语言。我们应该完全按照软件工程实践来开发语言，因此，开发可信程序的任务要贯穿于软件的全生命周期当中，特别是系统设计的早期阶段。

假如要人们编写百万行代码而不能犯任何错误，那样，系统就无法开发了，况且调试自身也不是开发可信系统的高效方式。依靠语法和语义规则来消除偶然错误，才是一种很好的方式。

6.3 Unix 及其文化

早期的 Unix 文化看起来好像是在倡导这种观点：你有问题时，就要编写一个小编译器或者一个小语言来

解决它。创建一种新语言来解决一个具体问题，而不是用其他语言编一个程序来解决那个问题，您觉得在何种程度上这是一种正确的方式吗？

AI：今天，世界上已经出现了成千上万种编程语言，人们会问，为什么会出现这么多的语言。事实上，几乎所有的人类活动领域都有自己的行业术语。音乐家使用专门符号来创作音乐；律师使用行业术语来讨论法律；化学家使用专门图表来描述原子和分子及其结合方式。如果人们说“我们要创建一种用这些符号表示的语言，用来解决特定领域出现的问题”，这其实再自然不过了。

你可以使用一种通用编程语言来表示所有的算法，不过从另外的角度来看，使用特定语言来解决某类特定问题，通常会更方便、更经济，甚至可能会更有启发性。何时创建一种新语言成了一种主观判断，不过，如果某个领域备受关注并有适合自动化处理的专门用词，自然就会有一种编程语言出现，成为某一特定领域的解决方案。

程序员的投入和硬件效率二者相比，哪个更为经济？

AI：至少在早期，语言的产生是因为人们认识到有一类特定的重要问题需要解决，随后，他们设计出“硬件高效（hardware-efficient）”的编程语言，用它编写程序来解决那些领域的问题。随着硬件变得成本更低、速度更快，语言倾向于变得越来越高级，而硬件效率的相关性则是越来较小。

您认为 AWK 在自己的小众市场中是否足够强大？

AI：AWK 中内嵌的 pattern-action 模式，用于解决日常出现的大量数据处理类问题，这是非常自然的。改变这种模式将会累及语言本身，也无法达到我们解决那类问题的目的。对于 Unix 命令行编程来说，这种语言也是很有用的。

这听起来像是 Unix 的设计哲学：将在各自领域内表现出色的众多小工具组合起来。

AI：我认为这是一个非常恰当的描述。

我见过的使用 AWK 的场合大都是命令行或者是 shell 脚本。

AI：用命令行组合或者结合 Unix 命令编写 shell 脚本来解决问题，这样的应用程序就是非常流行的 AWK 程序。这种解决问题的方式在早期的 Unix AWK 应用程序中非常常见，即便在今天，也还有很多这样的 Unix 应用程序。

在 Unix 中，“所有的东西都是文件”。您设想过因特网也是“文件”这种情形吗？

AI：文件是一种很好的、简单的抽象概念，它们应该在恰当的地方使用。不过，今天的因特网，数据类型已经变得更加丰富，同时，程序也常常需要处理并发的交互式多媒体数据流。目前，使用明确定义的标准化 API 来处理数据，看起来好像是最佳的解决方案，而且，安全程序需要关注如何对结构欠佳的数据做出恰当反应。

您认为在命令行工具和图形化接口工具方面各有哪些限制？

AI：在程序输出格式转换方面，AWK 是很有用的，由此，这个程序的输出就可以用作另一个程序的输入。如果一个图形接口已经把这类数据转换预编成“点鼠标”，这样显然会更加方便。如果没有这样做的话，那

么，可能就很难在内部获得这样的格式来做所需的数据转换。

对于某些特定领域，通过管道从命令行把两个程序连接起来的想法和编写一个小语言的想法，二者是否有联系？

108

AI：我认为有联系。毫无疑问，在 Unix 的早期阶段，管道技术推动了通过命令行实现的函数组合。你能够在一个输入上面做一些转换，然后通过管道输出给另一个程序。这为通过简单的程序组合快速创建新功能提供了一个非常强大的方法。人们开始思考如何通过这些命令行来解决问题。Larry Wall 编写的 Perl 语言，就是把很多的这类程序组合融入到一种独立的语言当中，我认为这种语言是从 AWK 和其他 Unix 工具中派生出来的。

当您说到“函数组合”的时候，我的脑海里出现的是一种“函数组合”的数学方法。

AI：我也正是此意。

在发明管道技术的时候，脑子里已经有了数学形式主义了呢，还是后来有人觉得它们的作用原理相同才把它加进去的？

AI：我认为一开始就有。至少在我的书里，我认为 Doug McIlroy 有功于管道技术的提出。他像数学家那样思考，而且，我认为他一开始就认识到了这种联系。我认为 Unix 命令行是一种典型的函数式语言。。

对语言思想和语义的形式化在多大程度上是有用的？AWK 中是否有一个底层的形式体系？

AI：AWK 是按照语法制导翻译（syntax-directed translation）模式设计的。我对编译器和编译原理很感兴趣，因此当我们创建 AWK 时，是作为语法制导翻译实现的。在 AWK 中，我们有一个上下文无关文法的形式语法，而且从源语言到目标语言的翻译是在那个形式语法的基础上按照语义行为来进行的。这一点促进了 AWK 的开发和发展。我们拥有自己能够支配的新创建的编译器构造工具 lex 和 yacc，它们对 AWK 语言的开发和实验提供了极大的帮助。

来自 Haskell 的 Simon Peyton Jones 说，80%到 85%的语言都有形式体系，不过，除此之外，因为回报的减少，剩下的语言不值得他们浪费时间来进行形式化了。

AI：由于安全的原因，专一性已经变成了语言和系统设计中的一个更加重要的问题。黑客通常利用系统不常见的或者未详细说明的部分来危害它的安全性。

库设计的问题也在其中，而且，问题突然变得更大。“我已经详细说明了我的语言的形式体系，不过，现在我需要一个库同因特网交互，我是否为那个库的形式体系进行了量化？他们是否符合那个形式体系？它们是否与语言的形式体系和保证相冲突？”

109

AI：在电信行业工作时，我注意到几乎所有的贝尔实验室所造设备的接口规范都符合国际标准。话虽如此，很多标准都是用英文编写的，而且它们通常是含混不清、尚不完善以及相互矛盾的。不过，尽管存在这些困难，国际电信网络和因特网之间的交互和协作都进行得很好，主要得益于系统之间定义明确的接口。

第三方通常为其他厂商的操作系统创建设备驱动器和应用程序。如果一个设备驱动器或者应用程序满是缺陷，系统厂商就会为糟糕的软件质量而承担骂名，虽然这并不是他们的过错。最近，在使用模型检查和其他功能强大的验证技术构建软件验证工具方面，研究社区取得了很大的进步，这些验证工具可以检查以保证那

些设备驱动厂商和其他应用开发人员正确地使用了系统 API。这些新的软件验证工具对于提升软件质量有着明显的好处。

语言是否能从这种形式体系受益呢？

AI：现在，几乎每一种语言都采用了某些形式语法描述。主要问题在于，我们可以在多大程度上，能够或者愿意使用目前描述编程语言语义的形式体系来描述语言的语义。语义形式体系远不像使用上下文无关的文法构造分析器那样机械化。即使描述语义是单调乏味的，在实现一种语言之前，做好它的语义轮廓规划、描述和概括也将会大有裨益，我对此深信不疑。

我想到了两个故事：它们都是关于“Make”命令的经典故事。Stuart Feldman 决定不删除 tab，因为他已经拥有了 12 位用户；而 Dick Gabriel 在《Worse Is Better（更坏就是更好）》一文中（注 3）（译注1），描述了新泽西方式和美国麻省理工学院方式的异同。结果是 Unix 和 C 和新泽西方式更胜一筹。

AI：我一直像达尔文学说的成功那样表达我的观点。我相信成功的语言会在程序员的实际运用中发展和演化。语言设计初期需要一个谨慎、持重的委员会把关，而一般来说，绝大多数程序员都忽略了这一点。除非它们被授权使用，否则，它们似乎不太可能幸存下来。

也许，流行语言需要警惕的一点，就是它在持续不断地变得越来越大。我们不知道如何从已有的语言中抽取它们的特性。现在的主流语言，比如说 C++和 Java，已经比创建之初大了很多很多，而且，似乎也看不到这些语言在未来有缩减的趋势。没有任何一个人能够真正地通晓所有的语言，而且这些语言的编译器代码动辄就是数百万行。

在系统研究上，这看起来好像是一个尚未解决的问题：您是如何不用修改语言本身，而使你的语言具有可扩展性，从而解决了初始领域之外的问题的？您有没有一个扩展机制？

AI：库是解决这个问题的一个由来已久的办法。

即使是 C++和 Java 也即将对语言做一些修改。

AI：是的。即使是核心语言也在发展，不过，还有一种向心力使得核心语言与过去的语言相兼容，因此，并不会破坏已有的程序。这就阻碍了这些语言彻头彻尾的演化。

这事本身是不是好事呢？

AI：毫无疑问，能够运行过去的程序是一件好事。我曾经为 Science Magazine（科学杂志）写了一篇文章，题目叫“Software and the Future of Programming Languages（软件和编程语言的未来）”。在这篇文章中，我想评估一下有多少软件会在日常事务中使用，并考虑到了世界各地不同的组织和个人使用的各种不同的软件系统。

注 3：*http://www.dreamsongs.com/WorseIsBetter.html*。

译注1：更坏就是更好（Worse Is Better），这是 Lisp 编程语言特别是 Common Lisp 领域的著名专家 Richard P. Gabriel 根据自己的亲身经历得出的著名论断。后来发展为一种软件设计风格或理念。Gabriel 强调软件和界面的简洁。这同美国麻省理工学院方式（MIT approach，or the right thing）的作法形成鲜明对比，后者强调事前完美的设计。由于“更坏就是更好”可快速推出软件应用，并持续完善，故而易于推广。

我估计这些软件的规模在五千亿到一万亿行源代码之间。假定开发一个软件成品需要 10 到 100 美元，因此，我得出结论：大部分遗留系统我们无法改编，因为我们根本承担不起相关费用。这就意味着，现有的语言和系统将会陪伴我们很长的时间。在很多方面，硬件的可移植性比软件更好，因为我们总是想要开发速度更快的硬件，用于旧程序的运行。

使用 Unix，您突然就有了一个可移植性非常强的操作系统。这是因为它们本身就可以移植，还是因为当它们变化时，需要将已有的软件移植到不同的硬件平台上？

AI：随着 Unix 的发展，计算机的发展速度甚至更快。Unix 最大的发展之一就是，Dennis Ritchie 创建了 C 语言并用它编写了 Unix 第 3 版。这增强了 Unix 的可移植性。我在贝尔实验室工作的近几年里，我们让 Unix 在不同的平台上运行，从小型计算机一直到世界最大的超级计算机，这是因为它是用 C 语言编写的，而且我们还拥有可移植编译器技术，它可以使 C 编译器很快地适应新机器。

Unix 的一个重要关注点是建立一个适合软件开发的系统，程序员会喜欢这个系统，并愿意用它来开发新程序。我认为这是极为成功的。

大多数最佳工具和最佳软件都是为了实现这个目标。

AI：这是一个有趣的问题：最好的工具制造者是工匠，还是刀具工？对此，我认为并没有什么明确的答案，不过毫无疑问的是，在 Unix 的早期，很多最有用的工具是由程序员创建的，他们设计这些创新性的工具来解决他们感兴趣的那些问题。这正是 AWK 产生的原因之一。Brian、Peter 和我已经想好了要编写的应用程序类型，不过我们想要通过真正的短程序来编写它们。

是不是这些工具的出现和快速的实践反馈促使人们研究更好的工具和算法呢？

AI：如果回顾一下 Unix 的早期历史和我的早期研究生涯的话，你会看到，我从 Knuth 名言中受到了极大的激励：最佳的理论是靠实践来推动的，而且，最佳的实践也是靠理论来推动的。我撰写了很多论文，探讨如何让分析更加高效，并且以便捷、高效的方式来分析实际编程语言中出现的结构。在研究这个理论和开发 yacc 时，我和 Steve Johnson 以及 Jeff Ullman 合作得非常紧密，因此，yacc 是理论和实践的完美结合。

6.4 文档的作用

The Role of Documentation

我在为自己编写的软件撰写文档、教程或者论文时，经常发现某些设计很难或者不好解释，为此我会修改程序。您有过类似的经验吗？

AI：不错，正是这样。我在 AWK 方面的经验对我在哥伦比亚讲授编程语言和编译器课程有着深远的影响。这门课程里有一个一学期长的项目设计，学生在这个项目中 5 人一组编写他们自己的小语言，并为它构造编译器。

在我 20 多年的编译器课程教学生涯中，还没有哪个团队在学期结束时无法提交一个可运行的编译器。这种成功并非偶然，而是基于我关于 AWK 的工作经验，在贝尔实验室我了解了软件的开发流程，认识到轻量级软件工程过程对于这种项目的重要性，而且，我还学会了听取学生的意见。

要在 15 周内成功创建一种新语言并为它编写编译器，对于这个编译器项目来说，遵照软件工程流程至关重要。学生有两周时间来决定是否要参加这个课程。两周后，学生们组成一个团队。再过两周，他们必须编写出一个简要的白皮书（按照 Java 白皮书的模式），阐述他们想要创建的语言。实际上，白皮书就是有关他们的语言的价值主张，要说明为什么要编写这种语言，以及这种语言应该具备什么属性。对于白皮书来说，最重要的是促使学生尽快决定想要创建什么样的语言。

一个月后，学生编写出一本语言教程和一本语言参考手册。该教程参照了 Kernighan 和 Ritchie 的《The C Programming Language（C 编程语言）》[Prentice-Hall]的第 1 章，参考手册则是参照这本书的附录 A。我对这些教程和语言参考手册的评阅得非常仔细，因为这时候，学生并不知道创建一个编译器是多么的困难，即便是为一种小语言而创建。

我要求学生说清楚：他们最低限度会实现哪些特性，如果时间富余的话，他们还能额外实现什么特性（从来没有学生额外实现过任何特性），这样做的目的是确定这个项目的范围，以便能在一个学期内完成，并且同其他项目的工作量相当。

一旦形成团队，学生们就会为这个团队选出项目经理、系统架构师、系统集成师、核查员以及语言顾问等。在创建和交付可用编译器的过程中，这些人都会发挥关键作用。

项目经理负责建立和实施项目交付时间表；系统架构师负责创建编译器结构图；系统集成师负责定义工具和开发环境，用于创建编译器系统。一旦编写好了语言参考手册，核查员就会为整个语言制定一个测试计划和测试套件；语言顾问负责确认白皮书中定义的语言新属性确实得到了实现。

我们为 AWK 创建了一个回归测试套件，或许是稍微有点晚了。我们的测试套件简直是无价之宝。我们开发语言时，在向主目录提交 delta 版之前，一直在进行回归测试。这样一来，我们在任何时候都会有一个可运行的编译器版本。在给语言添加新功能之前，我们要为给它创建一个新的功能测试，并添加到回归测试套件之中。

前面我提到，还没有一个团队会在学期结束时无法提交一个可运行的编译器。回归测试套件是实现这个目标的关键所在：学期结束时，学生会提交他们所做的工作。不过，可运行的编译器需要实现语言参考手册中提到的语言特性。

系统架构师为编译器创建结构图，详细定义接口规范，并指定谁在什么时间完成哪个组件。每一个团队成员都必须为该项目编写 500 行以上的源代码，并且每个人（包括项目经理）都要做一些实现工作。对于学生来说，编写程序时要同别人的代码进行交互，这是非常有益的（并且富有挑战性）。

系统集成师必须指定这个编译器的构建平台，以及需要使用哪些工具，比如说 lex、yacc 和 ANTLR 等诸如此类的工具。他还必须学会如何使用工具，并教会团队的其他成员如何正确使用，因此，每个团队都会安排负责工具的人员。

语言顾问的工作最为有趣。他负责语言知识的完整性，因此，语言白皮书中提及的那些属性能够得以实现。如果团队对语言设计做了修改，他就需要修改基线设计和编码，而且，这些修改会被记录下来，并且通告整个团队和回归测试套件。

通过这个项目，学生学到了三种重要的技能：项目管理、团队协作，还有口头和书面交流。课程结束时，我会问学生在课程学习期间学到的最重要的东西是什么。他们通常会说学到了这些技能当中的某一个。文档编制工作也推动了这个项目，而且，学生还获得了大量的有关软件编写和讨论的实践锻炼。学生们必须就他们的语言做一个课内推介演示，主要目的是要说服他们的同学：世界上每个人都应该使用他们的语言。我还和

第一个团队排练了一下如何做一次成功的演示。后来的团队一直设法要超越第一个团队，因为学生对于自己的语言热情极高。他们创建的语言范围很广，从模拟量子计算机到作曲、制作漫画、模拟文明、进行快速矩阵计算，一直到生成图形等。

课程结束时，学生必须提交项目最终报告，包含以下几章内容：语言白皮书；语言教程；语言参考手册；关于如何管理项目的一章，由项目经理编写；关于结构图和接口规范的一章，由系统架构师编写；关于描述开发平台和工具的一章，由系统集成师编写；关于测试计划和测试套件的一章，由核查员编写；关于语言基线过程（language-baselining process）的一章，由语言顾问编写；最后一章标题为“经验与教训”，回答了以下问题：“作为团队，你学到了什么？作为个人，你学到了什么？如果下一年还有这个课程，你建议哪些应该保留，哪些应该修改？”；附录包含代码清单，每个模块有作者自己的签名。

如果你长期以来一直精益求精，结果通常会令人相当满意。我经常听取学生给我的建议，而且就在几年前，我还因为这门课程获得了美国哥伦比亚大学研究生会评选的最佳教师奖。

很多面试过参加这门课的学生的招聘人员，他们都表示希望能以这样的流程来开发自己的软件系统。

课上的这些学生是哪个年级的？

AI：他们大多是四年级本科生和一年级研究生，不过这门课需要很多先修课程：高级编程、计算机科学理论以及数据结构和算法。令我印象深刻的是，这些学生最后都在做分布式软件开发，因此，他们经常使用 wiki（译注2）和高级集成开发环境（IDE）。还有许多学生在业内实习。

还有一点我要重点强调：学生们应根据语言的发展，及时更新回归测试套件。回归测试套件会使得学生的生产率更高，因为他们是在找自己的 bug，而不是团队其他成员的 bug。

调试应该在什么时候讲授，又怎么来讲授呢？

AI：我认为调试应该和编程一块儿讲授。在 Brian 的各种书籍中，他都给出了完善而且实用的调试建议。不过，对于调试来说，我还从来没见过任何通用的好理论。编译器调试技术与那些常用的数值分析程序调试技术，二者区别很大，因此，最佳的方式或许就是强调单元测试实例、系统的测试流程以及使用调试工具，把它们作为每一门编程课程的一部分。我还认为：在写程序前，让学生先就他们程序的实现内容编写技术说明，将会大有裨益。

对于 AWK，我们所犯的错误之一就是没有从一开始就进行严格的测试。在项目开始之后，我们确实进行了严格的测试，不过事后想想，如果从一开始就创建和完善了严格的测试套件，那我们的生产率还会更高。

对于代码库的发展，开发者应该关注哪些因素，并且从哪方面来关注？

AI：实现的正确性是最重要的关注点，不过对于正确性来说，没有什么实现的捷径。它涉及到多项任务，比如要考虑不变量、测试和代码审查等。应该实现最优化，但要等到时机成熟。保持代码、文档和注释的一致也很重要，不过这些很容易被人忽视。带有优秀的软件开发工具的现代集成开发环境（IDE）是绝对必要的。

译注2：wiki 是一种提供“共同创作（collaborative）”环境的网站，也就是说，每个人都可以任意修改网站上的页面数据。（wiki-wiki 是夏威夷语里“快（quick）”的意思。）

如果有好几天甚至好几个月没有接触代码，您会如何接着编程呢？

AI：当人们在编写一种编程系统（或者一本书之类的）时，需要把整个系统的情况记在心里。出现中断会打破一个人的思考过程，不过，如果中断是短暂的，那么回顾一下系统的一些代码就能想起来。如果中断长达几个月或者几年，我通常就会查看记录、书籍或者我做的算法标记，以此来唤起我对以前所编代码的回忆。

我想我要说的是，优秀的注释和文档，不仅是对于系统的设计者非常有用，同样，对于那些需要长期维护这些代码的程序员来说，也是很有益处的。Brian 维护了我们设计语言时所作重要决策的日志记录。我发现这个日志简直是无价之宝。

6.5 计算机科学

Computer Science

计算机科学研究些什么？

AI：这个问题问得很好，然而，这个问题并没有一个明确的答案。我认为计算机科学的研究范围已经极大地扩展了。在计算机科学方面，我们仍然有一些尚未解决的、深奥的经典问题：我们如何证明因式分解或 NP 完全问题，其实这个问题非常难；我们如何为像人体细胞或人类大脑这样复杂的系统进行建模；我们如何构造可扩展的可信系统；程序员如何随意构建可信软件；我们如何可以使软件具有情感或者智能这样“人类友好”的特性；摩尔定律还能走多远？

今天，计算机的规模和范围出现了爆炸式的增长。我们试图组织和访问世界上所有的信息，而且，计算机和计算也在对人们日常生活的方方面面产生影响。结果是，在计算与科学和人类活动的其他领域相结合的跨学科应用中，全新的计算机科学研究领域渐现端倪。这些新领域包括计算机生物学、机器人以及计算物理系统等。我们不知道在教育和健康领域如何更好地利用计算机。保密性和安全性变得比以前更加重要了。我相信计算机科学在任何时候都是一个令人兴奋的研究领域。

数学在计算机科学和编程中起到什么作用？

AI：我认为最佳的工程应该建立在坚实的科学基础之上。我们设计的 AWK 语言就是基于许多源自计算机科学理论的“优雅的”抽象概念，例如，正则表达式和关联数组等。大部分脚本语言都采用了这些结构，比如说 Perl、JavaScript、Python 和 Ruby 等。我们还使用了基于有限自动机的高效算法来实现字符串匹配基本类型。总而言之，我认为 AWK 是优秀理论和良好工程实践的完美组合。

您的研究领域是自动机理论及其在编程语言中的应用。当您开始把自己的研究成果付诸实现时，最让您吃惊的是什么呢？

AI：或许最让我吃惊的就是它具有极强的适用性。让我把自动机理论解释成形式语言以及识别它们的自动机。自动机理论提供了有用的符号，特别是正则表达式和上下文无关语法，那些符号用于描述编程语言重要的句法特性。识别这些形式语言的自动机，比如说有限状态机和叠加（下推）自动机，它们能够担当编译器扫描和分析程序的算法模型。自动机理论对于编译来说，最大的好处或许就在于能够创建编译器构造工具，比如 lex 和 yacc 等，这些工具自动构建了基于那些自动机的高效扫描器和分析器。

是什么妨碍了我们构建一个可以识别所有潜在 bug 的编译器（与/或一种语言）呢？有些 bug 是与程序设计错误相关的，而有些 bug 则是如果语言更具前瞻性就可以发现或者预防的，这两类 bug 的分界线在哪里呢？

Al：不可判定性决定了不可能设计出能够发现程序中所有 bug 的编译器。不过，使用功能强大的技术（比如说模型检测）来创建有用的软件验证工具，以查找程序中重要的 bug 类，我们已经在这些方面取得了很大进步。我认为，未来的软件开发环境会带有很多验证工具，利用它们，程序员可以更加精确地发现程序中的许多常见错误。

我的长期目标是：通过使用更强的语言、功能更强大的验证工具以及更好的软件工程实践，软件的可靠性和质量将会得到进一步提升。

我们怎样才能设计出利用多核硬件并发性的模式匹配算法呢？

Al：目前，这是一个活跃的研究领域。很多研究者在研究一些模式匹配算法的并行硬件和软件实现，例如 Aho-Corasick 算法和有限状态算法等。基因组分析和入侵检测系统是这些研究强有力的推动因素。

是什么促使您和 Corasick 开发 Aho-Corasick 算法的？

Al：这背后还有一个非常有趣的故事。在 20 世纪 70 年代早期，我和 John Hopcroft 还有 Jeffrey Ullman 正在编写《The Design and Analysis of Computer Algorithms（计算机算法设计和分析 ）》[Addison-Wesley]这本书。我在贝尔实验室就算法设计技术做了一次演讲。来自贝尔实验室技术信息库的 Margaret Corasick 听了我的演讲。在我的演讲结束后，她过来告诉我，她为关键字和短语的布尔函数编写了一个文献检索程序。不过，在一些复杂的检索中，运行限制程序可能会超出 600 美元大限。

她最初实现的检索程序使用了一种简单模式匹配算法。我建议她使用有限自动机来并行检索关键字，这是在线性时间内针对任意关键字集高效构造模式匹配自动机的一种方法。

几周后，她又来到了我的办公室，告诉我说，“你还记得要花 600 美元的检索程序吗？我已经实现了您建议的那个算法。现在，这种检索只需花 25 美元。事实上，现在每一个检索系统都是 25 美元；这些钱花在了读磁带上”。这就是 Aho-Corasick 算法的起源。

我的实验室主任 Sam Morgan 知道了这些事后，对我说，“你为什么不继续研究算法呢？我认为它们未来会大有用处”。这就是当时贝尔实验室的魅力所在：有些人满脑子都是问题，而有些人则会用非传统的方式来考虑这些问题。当你把这些人聚在一块儿的时候，就会不断地涌现出令人惊叹的创新思想。

116

6.6 培育小语言

Breeding Little Languages

是什么让您迷上编程的呢？

Brian Kernighan：我真的不记得有什么特别的事情让我迷上了编程。大三以前，我都不知道我的第一台计算机是个什么样子，而且，当时我没有真正地学习过编程（使用 FORTRAN），直到大约一年之后。我认为我最

有趣的编程经历是在 1966 年的夏天，当时我在美国麻省理工学院找了一份暑期工作，为新的 GE 645 编写一份作业磁带，它是 Multics 的雏形。我使用的是 MAD 语言，它要比我此前用的 FORTRAN 和 COBOL 更加友好，同时，我还用到了 CTSS，它是第一个分时系统，用起来要比穿孔卡（译注3）方便多了，而且也让人觉得更舒服。这就使编程从解决难题变成了真正的乐趣，因为机械的那些东西几乎不再是什么障碍了。

您是如何学习一种新语言的？

Brian：我发现，通过精选的例程来学习新语言是最容易的，这些例程的功能跟我想要的很类似。我复制一个例程，按照自己的需要修改，然后再通过这个具体的应用来扩充自己的知识。我涉猎过很多种语言，不久我对它们的印象就变得模糊了，当我从一种语言转移到另一种语言上时，我需要一段时间来适应它，特别是它们跟我很久以前学过的 C 语言不一样时。如果一个编译器能够甄别出可疑结构和非法结构，肯定是再好不过了；像 C++和 Java 这样强类型的语言，在这种情况下就非常有用，同时，强制要求严格遵守相关标准的语言的表现也相当出色。

一般来说，最好是自己多编一些代码，甚至是别人也会用的好代码。其次是，尽管自己不常写，也要多读一些好代码，看看别人是怎么写的。最后，丰富的经验也很有用：每一个新问题、新语言、新工具、新系统都会对你有帮助，并且会和你已经掌握的知识融会贯通。

对于一种新的编程语言来说，应该如何组织使用手册的内容？

Brian：这种手册应做到易于查找。也就是说，它必须有一个很好的索引，其中的表结构，比如说操作符和库函数等，必须简单明了，而且非常完备（并且易于查找），同时，手册里面的例程应该短小精干，并且清晰易懂。

这和教程不同，教程肯定和手册不是一码事。我认为，对于教程来说，最佳的方式可以说是一种“螺旋式的上升”，教程里只需要讲一点有用的基础内容，足以编写出完整、有用的程序即可。这个螺旋的下一圈应该涵盖另一个层次的细节，或者是采用另外一些方式来阐述同一概念，同时，例程也应该仍然是有用的，不过规模要稍大一些。最后还要附一个参考指南。

例程（甚至是入门级例程）中应该包括错误处理代码吗？

Brian：我对此也是难以取舍。错误处理代码通常很庞大，它不仅枯燥无趣，而且也没有什么意义。因此，它通常会妨碍你学习和理解语言的基本结构。另一方面，的确有必要提醒程序员：程序错误在所难免，而且他们的程序应该能够妥善处理这些错误。

我个人倾向于：在教程的最开始部分，除了提及错误可能发生之外，应该忽略错误处理。这就像在参考指南中一样，除了专门讲错误的那些章节以外，大部分例程都应该忽略错误处理。但是，这可能会强化人们潜意识中“忽略错误是安全的”这种观点，而这种观点肯定是不对的。

对于 Unix 手册中描述 bug 的章节，您持何种观点？现在这种做法还适用吗？

Brian：我很喜欢有关描述 BUG 的章节，不过，前提条件是程序很小并且很简单，而且也不可能确定出单个

译注3：穿孔卡，一种用于将数据输入计算机的介质，主要是一张上面穿有代表字母及数字的洞孔或 V 形孔的卡片，或者是一张上面穿有表示相关数据的小孔图形的卡片。

bug。BUGs 通常来自尚未提供的特性或者来自不当实现，而不是通常意义上的数组越界之类的 bug。我认为，对于真正的现代大系统中（至少不是在参考手册中）存在的大多数错误来说，这并不可行。在线 bug 仓库是一个很好的软件开发管理工具，不过，它对于普通用户帮助不大。

您认为目前的程序员需要了解您在《The Elements of Programming Style（编程风格要素）》[Computing McGraw-Hill]一书中提到的关于编程风格的经验教训吗？ 118

Brian：没错！良好的编程风格是编写清晰而简单的程序的根本所在，现在，这种风格的基本概念仍然同 35 年前我和 Bill Plauger 首次提出时一样重要。编程风格的细节大同小异，在某种程度上取决于不同语言的特性，但是它的基本概念还是和 35 年前一样。简单易懂的代码更容易正常运行，而且很少会出现问题。因此，随着程序越来越大、越来越复杂，保持清晰、简单的代码风格，甚至变得更加重要。

您编写文本的方式会对编写软件的方式产生影响吗？

Brian：可能会有影响。无论是文本还是程序，我总是对材料进行反复推敲，直到感觉它没有错为止。当然，写散文更是如此，不过它们的要求都是一样的，就是让语言或者代码尽可能地简洁明了。

了解软件要为用户解决什么问题，这会有助于开发者编写出更好的软件来吗？

Brian：除非开发者确切地知道软件的用途，否则，极有可能是适得其反。

在幸运的情况下，开发者能够理解用户，那是因为开发者也是一个用户。早期的 Unix 表现这么出色，非常适合程序员的需求，原因之一就在于，它的创始人 Ken Thompson 和 Dennis Ritchie 想要创建一个满足他们自己软件开发需求的系统；结果，Unix 非常适合于程序员编写新的程序。对于 C 语言来说也是如此。

如果开发者对应用程序所知不多、理解不深，那么，尽可能多地获取用户的输入和体验就是至关重要的。关注你的软件的新用户是非常有意义的：一个典型的新手会马上尝试做一些事情或者做出一些你从来没有考虑过的假设，而且，你的程序使他们的日子更加难过。但是，你的用户第一次使用软件时，如果你不监控他们，那么你就看不到他们的问题所在。如果以后再去看的话，他们可能已经适应了你的糟糕的设计。

程序员如何才能提升自己的编程能力呢？

Brian：编写更多的代码！然后，多琢磨你编写的代码，并想办法把它重新编写得更好。如果可能的话，让别人也来读一读，无论这是你工作的一部分还是开放源代码项目的一部分。它还有助于编写不同类型的代码，以及使用不同的语言编写代码，因为它拓宽了你的技术领域，并为你解决某些编程问题提供了更多的方式。例如，阅读别人的代码并设法添加特性或者修复 bug；这样你就会了解别人解决问题的方法。最后，没有比教别人编程更有助于你改进自己的代码的了。

尽人皆知，调试软件要比编写软件困难多了，那么，应该如何教授调试呢？ 119

Brian：我敢肯定调试不是能教出来的，不过，我们确实可以告诉别人如何进行系统地调试。这在《The Practice of Programming（编程实践）》[Addison-Wesley]一书中占了整整一章，在这一章中，我和 Rob Pike 设法讲解如何进行更加有效的调试。

调试是一门艺术，但它肯定能够提升你的调试技能。新程序员常会犯一些粗心的错误，比如说，数据越界、函数调用时类型不匹配，以及（在 C 语言中）使用 printf 和 scanf 时错用转换字符等。幸运的是，这些错误

非常容易发现，因为它们非常明显。更好的是，它们很容易消除，你可以首先进行边界检查，这等于是在编写程序的时候就考虑哪里会出错。Bug 通常会出现在你最近编写或者准备测试的那些代码中，因此，你应该在这方面多加注意。

随着 bug 越来越复杂或者越来越隐蔽，处理 bug 就要花费更多的精力。有效的方式之一就是“分而治之”，它试图移除部分代码或者部分数据，将 bug 定位在越来越小的范围之内。另外，还有常见的 bug 模式：错误输入和输出的“数字命理学”通常也是出错的很大一部分原因。

最难对付的 bug 是你分析问题的心智模式（译注4）不对，因此，你根本没有意识到有问题。对于此，我宁愿休息一会儿，读一些文章，把问题解释给别人听，使用调试器。所有的这些都将帮助我以一个不同的角度来看问题，并且这通常可以把问题解决了。但是，不幸的是，调试一直是困难的。避免调试的最佳的方式是在开始的时候就很小心的编写程序。

硬件资源会对程序员的心态产生什么样的影响？

Brian：有更多的硬件资源总是好事——它意味着，举例来说，程序员无需对内存管理过于担心，而在二三十年前，内存管理则是让人头疼不已的错误之源（那时候我们恰好在编写 AWK）。它还意味着可以使用低效的代码，特别是通用库，因为运行时绝不像二三十年前那样存在很多问题。例如，我认为现在在 10MB 或者 100MB 的文件上运行 AWK 是小事一桩，而这在以前则是不太可能。随着处理器速度越来越快、内存容量愈来愈大，现在很容易进行快速实验，甚至用 AWK 这样的解释型语言也能写出高效的代码，而这在几十年前是不可能的。所有这一切都是一种伟大的胜利。

另一方面，随时可用的硬件资源通常会导致系统的设计和实现臃肿不堪，如果稍加限制，系统就能变得速度更快，而且更容易使用。毫无疑问，现代操作系统会有这个问题：我的机器启动时间好像越来越长，尽管如此，硬件速度也因摩尔定律比以前快了许多。因此，是软件让我的机器速度慢了下来。

您对“领域特定语言（DSL）”是怎么看的？

Brian：我用过很多现在所谓的领域特定语言，尽管我通常把它们叫做“小语言”，别人则是叫“应用特定语言”。它的思想是，通过把语言限定在特定领域之内，你可以使它的语法契合这个特定的领域，因此就容易编写代码来解决这个领域的问题。这有很多例子——SQL 就是一个例子，当然 AWK 自身也是一个很好的例子，它是一种简洁、易用的特定文件处理语言。

“小语言”们都想变大，这是一个大问题。如果它们很有用，人们通过提升原始语言的运行能力，以便更为广泛地应用它们。这通常意味着会向语言中添加更多的特性。例如，一种语言最初可能是纯粹声明式的语言（没有 if 条件测试，没有循环），而且，它也可能没有变量或者算术表达式。然而，上面的这些都是有用的，因此，它们也有可能被添加到语言之中。一旦添加进去以后，这种语言就“变大”了（不再是那么小了），同时，它也就逐渐地开始像任何其他的通用语言一样了，只不过所用的语法和语义不同，而且有时候实现也较弱罢了。

我编写的几种小语言都是专门用于文档编制的。第一种语言是和 Lorinda Cherry 一起编的 EQN 语言，这种语言是用于数学公式排版的。EQN 语言非常成功，而且随着我们的排版设备功能变得更强大，我还编写过一种用来画图表的语言 PIC。PIC 开始只能画图，不过，很快就发现：它需要算术表达式来处理坐标之类的计算；

译注4：心智模式（mental model），又叫心智模型，是指深植我们心中关于我们自己、别人、组织及周围世界每个层面的假设、形象和故事，并深受习惯思维、定势思维、已有知识的局限。

需要变量来存储结果；需要循环来创建重复的结构等。所有的这些功能都被加到了 PIC 语言之中，但是，每一种功能都笨拙难用，而且也不稳定。最后，PIC 功能变得非常强大，变成了一种图灵完备（译注5）的语言，不过，再也没有人愿意用它来编写代码了。

在工作方面，您是如何定义成功的？

Brian：最大的回报是有人说他们使用了你的语言或者工具，并且发现它有助于把工作做得更好。我对此非常满足。当然，有时候也会被告知存在问题或者缺少某些特性，不过，即使是那些也是很有价值的。

对于哪些应用，AWK 仍然是功能强大而且有用的？

Brian：对于快速数据分析和脏数据分析，AWK 仍然是最佳的：找出具有某些属性的所有数据行，或者概括数据的某些方面，或者对它做一些简单转换。我通常用两三行 AWK 代码，就可以比别人使用五到十行其他语言（比如 Perl 或者 Python）的代码做更多的事情，而且经验表明，我的代码几乎都运行得很快。

我收集了一些小的 AWK 脚本，可以完成很多功能，比如：对所有行中的所有字段进行求和，或者计算所有字段的范围（一种快速查看数据集的方法）。我有一个 AWK 程序，它可以把任意的文本行填充到至多 70 个字符的一行中，我用它来清理邮件或者干类似的事情，每天都要用上近百次。所有这些功能都可以用其他脚本语言来编写，但是用 AWK 更加简单。

编写 AWK 程序时，人们应该注意些什么？

Brian：AWK 语言最初是用来编写一两行长度的程序的。如果你想编写大程序，使用 AWK 或许并不合适，因为它没有支持大程序编程的机制，而且一些设计决策也让查找 bug 变得很困难。例如，对于单行程序来说，变量不用声明并会自动初始化，这是非常方便的，不过，这意味着大程序的拼写和排版错误是无法检测的。

6.7 设计一种新语言

Designing a New Language

如果要创建一种新的编程语言，您会从何入手？

Brian：假定你有一些任务和一些应用领域，在那里，你认为新的编程语言会比任何现有的语言更好。想一想人们会说什么。这种编程语言将要用于解决什么问题和应用场景？你为什么想要用那种语言来解决问题？描述这些问题最自然的方式是什么？人们使用它的最重要的例子和最简单的例子是什么？想办法把这些问题弄得尽可能简单易懂。

最根本的，它的思路就是在创建语言之前，先写写关于这种语言的东西。你会怎么样来描述呢？我认为这对于 AWK 来说很好，因为 AWK 语言的所有设计都是使它易于编写有用的程序，而无须解释很多。这意味着我们无须声明，部分是因为我们没有类型；意味着我们没有显式输入语句，因为输入是完全隐式的，它只是发生了；意味着我们没有将输入行自动分割为字段的语句，因为那是自动发生的。这种语言的所有属性都来自于这个目标：使它真正地易于描述真正简单的事情。

译注5：图灵完备(Turing-complete)，即如果一个计算系统能够计算任何图灵可计算问题(Turing-computable function)，就称之为图灵完备。

我们在最初写的 AWK 论文中使用的标准例子，在参考手册等处用的也是这些例子，所有这些标准例子基本都只有一行代码。我想要打印所有长度超过 80 个字符的行，只要编写"length>80"，就可以完成任务了。在这个特定语言中，我们想要做什么，目标很清晰，当然，随后你会发现落掉的所有真正需要的功能，比如通过名称读取特定输入文件的功能，因此，我们必须把这种功能添加进来。如果程序不止是几行，比如说函数，就需要使用结构，所以，后来就把它添加进来了。我和 Lorinda Cherry 编写的 EQN 语言是一个完全不同的例子。EQN 语言用来描述数学表达式，由此，这些表达式才可以被打印出来。

122 这种语言的设计目标是尽可能地接近人们描述数学的方式。如果我在电话里向你描述数学公式，我会怎么说？或者在课堂的黑板上写一个公式，写式子的同时我会说些什么？或者，拿我为盲人课本录音来说，我是怎么读数学表达式的，以便那些看不见的人也能理解？EQN 的关注点完全在使数学公式像说它们一样的简单的书写，并且它不太担心输出的质量。相比之下，TeX 有很多语法，并不容易输入，不过，这种语言的功能非常强大，它能让你更好地控制输出，而代价就是它更难使用。

您在设计语言时，对于实现您考虑了多少？

Brian：我对实现考虑了很多，因为我一直参与设计和实现；如果我不知道怎么实现，我就不会参与设计了。

对于我编写的几乎所有语言来说，无论它是简单的语言，简单到足以通过一个专门的分析器分析，还是它有一个更丰富的语法结构，我都能使用 yacc 为它详细定义语法。

我认为，如果我必须编写 EQN 或者 AWK 这样的语言，而没有 yacc 帮助的话，它们就会胎死腹中，因为很难手工为其编写分析器。并不是说你无法实现它，而是说它很让人厌烦。借助于 yacc 这样的工具来编写，可能更容易做一些有趣的、冒险的事情，而且，如果有错的话，还可以很快地修改设计，因为你只需要重写一点语法；你不必大量修改代码，或者添加一些新的特性。使用 yacc 这样的工具很容易实现这个目的，如果使用传统的递归下降分析器就非常困难了。

语言设计者应该优先采用某种风格来避免一些经常性错误吗？例如：Python 的源代码格式化，或者 Java 的缺乏指针运算。

Brian：我对这个问题是毁誉参半，尽管，一旦你习惯以后，这些强制性的原则大多是很有用的。一开始，我对 python 的缩进排版规则很是头疼。不过一旦我适应以后，这就不是一个问题了。

语言设计者应该设计一些结构良好的语言，以便人们容易使用，并且不会含糊不清或者有太多的不同实现方式。无论如何，人们总会找到最自然的表现方式。因此，如果 Java 忽略了指针，这是不同于 C 或者 C++的一个主要变化，不过，它提供了引用，在很多情况下，这是一个合理的备选方案。Java 没有提供 `goto` 语句。

123 我从来没觉得这是一个问题。C 语言提供了一个 `goto` 语句，我并不常用，只是偶尔使用，这是有意义的。因此，我对于这样的决定是很认同的。

我曾经提到过用于绘图的 PIC 语言。用它画箭头、方框图和流程图等，非常方便。但是人们想要通过常规的结构来画图片。为了这个目的，我极不情愿地增加了 `while` 和 `for` 循环，甚至是 `if` 语句。当然，这些都是马后炮，增加了并不适合 PIC 语言的不规则语法，但它们又不同于任何其他传统语言。结果是既可用，又难用。

似乎语言一开始都是非常简单的，随后开始增扩，逐渐添加，直到具备任何成熟的通用语言的所有特性，比如变量、表达式、if 语句、`while` 循环以及函数等。但是，它的结构混乱不堪，语法很不规则，或者至少可以说是不同的，这种机制的表现可能会让人无所适从，并让人感觉所有的东西都有问题。

出现这种结果，是不是因为它们起源于小语言，并没有考虑到将它们演化成通用语言呢？

Brian：没错，我认为问题就在于此。这仅仅代表我的个人观点，在我心里，小语言就应该很小、很简单，没想用它们来编一些大程序，也没想让它们发展为通用编程语言。不过，如果它有用的话，人们便开始解除它的限制并想增加更多的特性。通常，他们想要的东西是通用编程语言所具有的特性，使它们可编程而不仅仅是声明。他们想要有方法来进行重复，想要避免一遍又一遍的重复，这些会导致循环、宏或者函数的出现。

我们怎样才能设计出一种人人适用的语言呢？您曾经提到过专注于一个特定目标的小语言，但是我也记得您推崇开发者编写一种语言来满足自己的需求，一旦您做出了一些东西，您会怎么发展它，使它更加适合别人呢？

Brian：根本不可能有一种人人满意、适合于所有应用的语言，或者是很大一部分人满意、适合于很多应用的语言。

现在，我们拥有很多优秀的通用语言。对于某些任务来说，C 是合适的；C++、Java、Python，每一种语言都有自己适用的领域，并且几乎都可以扩展到其他领域。但是，我想我不会用 Python 来编写操作系统，而且，我也不想再用 C 来编写文本处理的代码了。

您是如何识别一种语言在哪个领域特别有用或者特别强势呢？例如，您说 Python 并不擅长于编写操作系统。您是特指语言还是实现呢？

Brian：我认为两者都是。实现了也可能意味着系统速度很慢。但是，如果我是编写一个玩具或者演示版操作系统，Python 可能会非常适合。它在某些方面可能会表现更好。但是我想，我不会用 Python 来写一个支持 Google 基础架构的操作系统。

现实中的程序员有时候根本没有选择的余地。他们必须满足局部环境的要求。因此，如果我是华尔街一家大金融公司的程序员，只能使用指定的语言或者一小部分语言编程，或许是 C++和 Java。我没有权利说："我只想用 C 编程"，我也没有权利说"哦，对于这个项目，我认为 Python 会更合适。"我知道有一家公司，它使用的语言是：C++、Java 和 Python。有时候，Ruby 或许会更合适，但是你没有权利使用 Ruby 编写程序。许多人并没有自由选择编程语言的权利。

另一方面，如果他们必须完成一个特定的任务，可能要在 C++、Java 和 Python 中间自由选择。然后在决定使用哪种语言时，会考虑技术人员的想法。

或许每个人都应该有自己的个人语言。

Brian：一个真正适合自己的，别人不会使用的语言？

在这种语言中每个人都有自己的个人语法，它将被转换成普通字节码，然后就通用了。

Brian：这样一来，合作开发就会变得很困难。

在你构建第一个原型之后，人们应该怎么做？

Brian：首先用这种推荐语言自己编写代码。用它来编写你自己要写的程序，或者你身边的人可能要写的程序，感觉如何呢？例如：对于 EQN，它就非常清晰。数学家们作何评价呢？我不是一名数学家，不过我有一个相

当好的主意，因为我学过很多数学课程。你自己想尽可能快地使用它，并且希望别的用户也尝试一下，不过，你希望这些用户确实是优秀的评论者，也就是说，人们要尝试使用、挖潜，并且告诉你他们发现了什么。

在 20 世纪 70 年代和 80 年代，贝尔实验室最值得称道的事情之一是：Unix 组内外有一大帮人，他们非常擅长对别人的工作进行批评性评估，其擅长程度简直令人难以置信。那些批评往往非常直率，但是对于工作的高下优劣，这种反馈既很中肯、又很及时。

125 我们都受益于批评，批评有助于磨掉棱角，保持系统在文化上的兼容性，并且去除其中的糟粕。

我认为目前在某些方面很难做到这样，不过，因为有了因特网，你可以通过更广泛的人群收集批评信息，不过，你可能无法得到真正有天赋的那群人的集中批评意见：那些人离你很近，你们可以在大厅里讨论一个小时，你还可以走进他们的办公室，跟他们一起讨论。

您是如何管理您所有的想法和实现，同时还要构建一个独一无二的稳定系统的呢？

Brian：这个并不难，因为 AWK 非常小。第 1 版大概只有 3000 行代码左右。我认为第 1 版是 Peter Weinberger 编写的。语法是通过 yacc 完成的，它非常简单。词法部分是通过 lex 完成的。并且从这一点上来说，语义非常规则。我们有几种不同版本的分析器，不过它们都不是很大。实际上，它们现在依然很小，即使是 30 年后的今天，我发布的版本也没有超过 6000 行代码。

你们每个人都必须为自己想要添加的新特性编写测试模块，这是真的吗？

Brian：不是真的。绝对不是这样。那本书是 1988 年出版的，从那时起，我们就开始更系统地收集测试用例。可能是若干年后，我才开始更加有序地收集测试，差不多就是这段时间，我决定：如果要添加新功能，我会增加一些测试，确保这个功能正常工作。我已经有很长时间没有添加新功能了，不过，测试集还是在不断地增大，因为一旦有人发现一个 bug，我就会添加一两个测试进去，以便能更早发现 bug，并确保它不会再次发生。

我认为这是很好的实践。我真希望我们能更早一些开始细心地、系统地做这些事情了。现在的测试套件基本上包括了 AWK 一书第 1 章和第 2 章的所有程序。不过，这些工作显然要在出书之后才能做，而出书又要在该语言问世之后。

过去 40 年在计算机科学和编程语言方面做了很多研究工作。在工具改进的背后，您有没有看到语言设计方面有所改进呢？

Brian：我不确认我所知道的是否足以对这个问题做出合理的解答。对于一些语言，就说脚本语言吧，这种语言的设计过程仍然是非常特殊的，它基于设计者的喜好、兴趣和信念。因此，我们有几十种脚本语言，我认为这些并不是直接得益于型态理论之类的研究。

126 **在 20 世纪 70 年代，每一种型态的语言都是可用的：C，Smalltalk—这是两种型态差异很大的语言。今天，我们拥有各种各样的语言，不过我们仍然在用 C、C++和 Smalltalk。对于语言的设计以及与计算机交互方面，你是否期望更多的创新和改进呢？**

Brian：我想我对整个领域了解得还不够。我认为在让机器为我们做更多的工作方面可能会变得越来越好。这意味着语言可能会变得更高级和更易声明，以致于我们并不需要详细解释这么多的细节。人们希望语言会变得更安全，因此，编写一个不能工作的程序会变得很困难。或许程序会很容易地转换成一种非常有效的可运行的形式，但是除此之外，我就真的不知道了。

我们能否拥有一种语言设计的科学？我们能否用科学的方法进行语言设计，这样我们就能从先前的发现或发明中学到更多，并且不断进步？它总是包括设计者的个人喜好在内吗？

Brian：我认为在语言设计中将会有很大一部分是靠个人喜好和直觉。实际上，几乎所有的语言都是两三个人的作品。你几乎很难说出任何语言是团队努力的结果，因此仅仅可以说可能是个人的。

同时，，对于编程语言几乎所有的方面，我们都认为很好，并且可能会更好。这暗示着，新语言应该基于合理的原则，并且它们的特性应该很容易理解。从这个意义上说，语言设计将比 10 年前或者 20 年前、确定无疑的是 30 年之前有更科学的基础，在那些年代，语言设计并没有太多的基础。但我猜测，很多语言设计还是由设计者个人的喜好决定的。

人们喜欢追捧那些吸引他们的事情，同时，另外一些事情一样也会吸引许多其他人。但是，语言还必须拥有越来越多的东西，因为它们的存在被认为是理所当然的。因此，举例来说：现在，任何一门重要的语言几乎肯定会有对象机制，并且从一开始设计就把这种机制考虑进去了，而不是后来才加进去的。另一个重要领域是并发性，因为我们有了多处理器的机器，而且语言也需要处理自身的并发性，而不仅仅是增加一些库。

随着我们更好地了解了情况，我们会尝试构建更大的系统，因此，在某种程度上，我们变得越来越强大。但是我们常常尝试着把目标设置得更高。我们一直在建设更大的团队。

Brian：我认为你的观点基本正确。我们总是想做更大的事情，因此总是疲于应付。随着我们有了更多的硬件、更了解了怎么写程序、有了更好的编程语言，我们开始着手干更大的事情。在 20 世纪 70 年代，可能需要几百号人的团队花一到两年才能完成的一件事，现在只要一个研究生花几个星期内就能搞定。因为现在有这么多的支持、这么多的基础架构、机器的功能这么强，并且还有这么多供你构建的已有软件。因此我认为，在某种程度上，我们总会遇到那样的情况。

我们是否会有像微软开发 Vista 那样的好几千人的大团队呢？或许会有，不过，我们非常清楚需要找到一种把大程序分解成很多小项目的方式，这些小项目通过安全而且组织良好的方式相互合作。

这里面部分需要改进语言的实现，另一部分需要更好的机制把组件整合起来，不管它们是用什么语言编写的，并且在信息通过接口时对其进行封装。

先前，你说过现在的编程语言肯定需要支持面向对象编程（OOP）。面向对象编程真的有这么好吗？为了简化构建大系统的流程，我们还能做些别的什么，或者是发明或者增加些什么吗？

Brian：在某些环境中，面向对象是很有用的。如果你编写 Java 程序，你就别无选择了；如果你编写 Python 或者 C++程序，你可以用它，也可以不用。我认为这很可能是正确的模型：你可以使用它，或者不使用它，这取决于具体的应用程序。随着语言的发展，肯定会有其他封装计算单元和管理程序的新机制。

如果你看一看 Microsoft 的对象模型组件 COM，这就是基于面向对象编程，不过，不仅仅只是面向对象，因为一个组件有很多的对象，并不仅仅只有一个。你如何使用比 COM 更加有序的方式来处理呢，而这不止是这些事情如何关联的概念？

我们需要一些机制来处理大量的对象。随着我们要处理的程序越来越大，或者是程序各部分来自很多地方，我们要处理非常复杂的对象结构。

Unix 是用不支持面向对象（OO）的 C 语言开发的，然后，你可以把组件（对象）构建成易于组合的小工具，从而实现复杂的功能。我们不是把对象的概念加到语言之中，或许我们需要把对象或组件这些小工具构建成独立的程序。如果你想编写一个电子制表软件，一般会认为它是使用对象构建的大程序，也许它会支持插件或者附件，不过，这个观点在于对象是在语言内部、集成和管理的。

Brian：Excel 是一个很好的例子，因为它包含了大量的对象及其相关方法和属性。你可以编写控制 Excel 的代码，因此，实际上，Excel 变成了一个大型的子程序，或者仅仅是另一个计算单元。这种管道不会像 Unix 管道那样整洁，但它们可能会非常接近，而且，让 Excel 成为某个管道的一部分也不会很困难。

Mashup（译注6）就有一些这样的风格：有可以以专门的方式聚合在一起的大的构建块。这并不像 Unix 管道那样的容易，但它构成大的，自我包含的大系统的想法是一样的。

Yahoo! Pipe 就是一个很好的例子。它非常有趣，“我们如何处理相当复杂的操作，并把它们结合在一起呢？”在所有底层的前面，他们放置了一个非常漂亮的图形前端。但是，你可以想象使用基于文本的机制来做同样的事情，从而有一个系统能够让你把任意的计算集合放在一起，只需要通过再编写一个基于文本的程序即可。搞清楚如何做好这一点肯定是值得的。我们怎么样才能通过已有的组件构建一个系统？我们怎么样才能让编程语言帮助我们做到这些？

在命令行的时代，你必须通过书面语言来同计算机交互：键入文本作为输入，读取文本作为输出。今天，我们通过键盘和鼠标进行交互，而且，我们的输出可以采用部分图形和部分文本方式。语言还是与计算机交互的最好方式吗？因为使用了语言，使用命令行同计算机的交互，在某种程度上会不会更好呢？

Brian：对于那些不熟练的用户，某些系统的新用户，或者你不经常使用的应用程序或者本身就是图形化的应用程序来说，图形化接口是非常好的。但是不久之后，你会发现自己在一遍又一遍地重复着同样的事情。计算机非常擅长于重复性的操作。如果我们让计算机“一遍又一遍地重复做”，这不是更好吗？

现在有这种机制了，比如说 Word 和 Excel 中的宏。但是，我们又看到了 Google、Yahoo!、Amazon 或者 Facebook 等系统使用的可编程 API。你可以进行以前要用键盘和鼠标实现的任何操作，而且，你也让它们实现“机械化”。同时，你也不需要像 10 年前那样紧盯着屏幕，并解析 HTML 了。

实质上，这又退回到了命令行时代，在这里，纯粹基于文本的操作是最合适的。你可能不知道自己想要做什么，直到你用基于鼠标和键盘的操作做了一些之后，你才知道你要做什么，但是，一旦你看到重复的操作，你就可以用命令行界面和这些 API 进行机械化处理，而在循环中不需要人工干涉。

在设计语言的时候，您是否考虑过功能的可调试性？AWK 遭受的一个批评是，变量是无需声明而自动初始化的。虽然这样很方便，但是，如果你出现了拼写或者排版错误，就很难发现问题。

Brian：这是一种折衷平衡。每一种语言都需要折衷平衡，而且在 AWK 中我们做出这种折衷的目的是为了使它非常非常容易使用。我们的目标是使用单行程序，因为我们认为大多数程序应该是只有一行或者两行代码。变量无需声明和自动初始化赋值与这个目标相一致，因为如果你必须声明并且初始化它的话，程序长度将会增长三倍。对于小程序来说，变量的无需声明和自动初始化是非常好的，而对于大程序来说则是不利的。因此问题是，你想用它来做什么？

译注6：Mashup（糅合），是指新出现的一种网络现象，它将两种以上使用公共或者私有数据库的 web 应用加在一起，形成一个整合应用。一般使用源应用的 API 接口，或者是一些 RSS 输出作为内容源，合并的 web 应用什么技术，则没有什么限制。

Perl 有一种模式，它会警告你："当我做蠢事时，请告诉我"。而 Python 则需要你更加小心。在 Python 中，你必须初始化变量，但你经常会未加声明就使用了。或者你手头有独立的 AWK 程序的 lint 工具（译注7），提示你："你有两个变量的名字非常相似，你真的要这么做吗？"

AWK 还有一个饱受争议的设计：它把使用邻接来表示串联，而且没有显式操作符；邻接值序列只是串联在一起。如果再没进行变量声明，这二者凑在一块儿，那么，你写出来的所有东西几乎都是合法的 AWK 程序。这就太容易出错了。

我认为这就是愚蠢设计的一个例子。它并没有为我们节省什么——我们应该已经使用了操作符。自动初始化变量是一种有意识的折衷，在小事情方面它做得很好而且不打折扣。

6.8 遗留文化

假设我要新编一种小语言，用于只有 2M 内存的手机或嵌入式设备。像这种实现问题，会在多大程度上影响接口层？用户使用我的程序时，会理解我的这种设计选择吗？或者说，现在我们为什么要摆脱这种类型限制？

Brian：我认为这跟以前有很大不同。如果你回顾一下早期 Unix 程序的历史，当然 AWK 也算在内，你可以看到：很多时候，语言或操作系统各个部分的内存都显得非常紧张。

例如，多年以来 AWK 都有内部限制：你只能打开有限多个文件，只能把有限多的元素存入到联合数组当中等。这些措施都是为了应付内存确实非常紧张和处理速度根本没那么快的问题。这些约束已经逐渐消失了。我在实现中已经取消了固定限制。固定限制是指资源限制，并且对于最终用户来说是可见的。

AWK 试图去保存变量的状态，因此，如果当一个变量被用作数字，并被转换为字符串来打印，AWK 就可以知道这个变量的数值和字符串值都是当前可用的，因此就不需要再做强制转换了。在比过去快 1000 倍的现代机器上运行，你根本不须要做转换。当需要的时候，你只须把值进行强制转换就可以了。

即便是最初，这也很可能是一件蠢事，因为有许多错综复杂，精确平衡和可能并非总是正确的代码管理着这种状态。如果是在今天来做的话，我根本不会去考虑它。我肯定我会使用 Perl 或者 Python，它们根本不需要当心这个。

Perl 5 仍然在使用那些非常奇怪的技巧。

Brian：Perl 的第 1 版是在 AWK 之后的不到 10 年内编写的，它里面仍然有大量的资源约束条件。总之，这些例子都是在资源紧张的情况下不得已而为之的，回过头来，你很可能不会再这么做。

如果我没有记错的话，当我们涉足 Unix 的时候，也是从只有 64K 字节的机器开始的。

译注7：lint 是一种软件质量保证工具，许多大型专业软件公司，如微软公司，都把它作为程序检查工具，在程序合入正式版本或交付测试之前一定要保证通过了 lint 检查，他们要求软件工程师在使用 lint 时要打开所有的编译开关，如果一定要关闭某些开关，就要给出关闭这些开关的正当理由。

Peter Weinberger 说过，在 Unix 早期阶段，那时总有一个感觉：明年可以再重写一个程序，所以，它无须完美，因为它既不大，也不复杂。你总是可以重写它。这是您的经验吗？

Brian：那时确实重写了相当多的程序。我不知道它们是不是彻底重写。就个人经验而言，我认为，我并没有把曾经做过的所有东西都扔掉，并从头开始重写。我是循序渐进地修改，但是里面包含了许多的重新思考，同时要看一看是否有办法使程序更小，这肯定也是这种修改的题中应有之义。

他给我造成这样一种印象：这已经成为了一种文化。设计程序时，从来没有考虑过它会生存 10 年、20 年或 40 年。您有没有看到这种从短期到长期的思考转变？

Brian：我不知道现在是否还有人会考虑软件的长期运行，不过以前有人曾经考虑过。有些人考虑是因为他们必须这样。例如：如果我在电话公司做交换软件，在过去那段好时光中，代码要持续存在很长一段时间，而且这些代码必须要与更早的代码兼容。你做事情时必须更加谨慎。或许，在无法重写代码的这个事实上，我们只是更加现实而已。因为没有足够的时间。

至少在我的记忆中，对于 20 世纪 70 年代的 Unix 来说，还有那么多有趣的新事情可以做：人们刚进来就在不停地修改程序。我想没有任何人会认为是为那个时代而编写程序。如果你在 1978 年告诉 AL、Peter 或者是我，我们会在 30 年后有一场关于 AWK 的对话，没有人会相信。

Unix 内核确实也在发展演变。可能很多人会有别的想法，但 C 语言仍然是 AWK 和内核这样的软件的最佳选择之一。为什么像 C 这样的语言会生存下来，而其他语言早已灰飞烟灭了呢？

Brian：它们生存下来，部分原因在于它们在自己的领域表现真的很出色。C 语言在系统实现方面很有前景。它有着难以置信的表现力，但另一方面，它既不复杂，也不大，而且还高效，这在一定程度上对它的生存确有帮助。它是一种好用的语言，因为，无论你想要说些什么，它没有太多的不同方式可供选择。我会一边浏览你的代码，一边说，“我看到你在干什么了”。对于像 Perl 和 C++这样的语言，我认为并不是这样的。我会一边浏览你的 Perl 代码，一边说：“嗯？”，因为没有一种方式可以编写它。

C++语言不仅很大，而且错综复杂，同时还有多种不同的表达方式。如果咱俩编写 C++程序，就可能会使用不同的方式来表示一个庞大的计算。C 语言就没有这种情况。C 语言之所以能生存下来，是因为它找到了表现力和效率之间的平衡，而且对于核心的应用程序，它仍然是最佳工具。

这就是为什么我们从未替换 Unix 上的 X Window 的原因。几乎所有的程序都在使用 Xlib（译注8），或者是使用 Xlib 的东西。Xlib 或许是千奇百怪的，它简直是无处不在。

Brian：确实如此。它完成了这种功能，而且做得很好。从头重做的工作量实在太大了。

现在，当你回顾 C++的时候，最初设计的目标之一就是向后兼容 C，不管结果是好是坏。该理论认为，如果你想替换 X，那么我们显然就需要能够运行 X 程序字符串。在很多地方，C++中并没有取代 C，尽管这是它名义上的目标。

Brian：为了让 C++尽可能与 C 兼容，Bjarne 费了九牛二虎之力。C++取得了其他语言没有取得的成功，原因

译注8：Xlib 是一种用 C 语言编写的 X Window System 协议的客户端。其功能是与 X server 沟通。程序人员编写程序时，无须了解其协议的细节。但应用程序很少会直接使用 Xlib；通常是通过其他的库来调用 Xlib 用以提供部件工具箱。

之一就是它的兼容性非常好，不管是在源代码还是对象方面。而且，这意味着你无须为了在 C 环境下使用 C++而改变原来的业务逻辑。

我确信，Bjarne 在兼容性方面的决策来自他饱受困扰的某些方面，因为人们会说："哦，这太可怕了，因为……"他是在经过了深思熟虑以后，才非常谨慎地做出了这些决策，因为，兼容现有系统非常重要，并且从长远来看，更有可能成功。

有人认为它最大的过错是太接近于 C 了。

Brian：或许吧，不过，它离 C 越远，就越不可能成功。这是一种很困难的平衡，我认为他做的已经相当好了。

在引进一个革命性的新东西时，你会在多大程度上考虑向后兼容呢？

Brian：这在任何领域绝对都是一个进退两难的问题，我也没有办法解决。

你提到过很多小语言开始添加新特性，变成了图灵完备型语言，同时失去了他们概念上的纯粹性。如果你在设计小语言，想让它更加通用而不失去其自身特点，这有没有设计原则可以适用？

Brian：我想应该有。我记得我在多种场合下说过，而且，我常常想知道这在多大程度上是一种狭隘的观点。也就是说，我所接触的所有语言都有这种属性，而且，或许其他语言不是这样。也许我只是看到了我自己的问题。事后看来，在大多数情况下，我自然更满意于新特性在语法上和已有语言兼容，因为人们无需再学习一种全新的语法了。

小语言是否会再次复兴？

Brian：我不确定使用"复兴"这个词是否恰当，不过，小语言还是会继续发展的。

在某种程度上，推动小语言发展的力量是 API 向 WEB 服务的扩散。每个人都有一个 API，这会使你从一个程序驱动他们的 web 服务，而不是从你的指尖驱动。这里，大部分 API 都封装成 JavaScrip API，但是，我可以想到更加方便的方法，它从 Unix 或者 Windows 的命令行运行，而不是编写浏览器内嵌的 JavaScript，在浏览器中你还需要点一下它才开始。

总的说来，您好像是在谈论在因特网上运行的 Unix 命令行的复兴。

Brian：这是表达观点的一种绝妙方式。这样不是很好吗？

6.9 变革性技术

Transformative Technologies

你提到 yacc 使得语言的语法实验变得更容易，因为你可以更新语法并再次运行，而不用修改"hand-rolled"直接下降分析器。yacc 是不是一种变革性技术？

Brian：毫无疑问，对于语言开发来说，yacc 的影响非常巨大。就我个人而言，如果没有它，我就无法顺利开始做这种语言工作。因为，不管是什么原因，我并不擅长编写递归下降分析器。我经常会遇到关于优先级和结合性的麻烦。

使用 yacc，你就不必考虑那些问题。你可以写下有意义的语法，然后说“这是优先权和结合性，以及如何处理棘手的情况，比如说一元操作符和二元操作符等”。所有的事情都是那么简单。正是这种工具的存在，使得人们有可能从语言的角度来考虑问题，否则就可能会很困难。

毫无疑问，yacc 对于 EQN 是非常合适的。它的语法并不很复杂，不过，它的结构却有点古怪。其中的一些结构在以前的编程语言环境中并没有被考虑过。而且，实际上 EQN 是声明性语言，而不是过程性语言。甚至有人还就此在 CACM（译注9）上展开了讨论：把上标和下标放在同一实体上，这个棘手的问题，使用 yacc 可以处理得很好，而使用其他方法则很难奏效。

从理论的角度来说，yacc 是一个令人惊叹的杰作，也就是说，采取这种语言技术，理解如何分析并把它转换成一种程序——但是，它也有非常好的设计，在当时堪称最佳。很长一段时间内，没有任何设计能够接近 yacc 所能达到的水平。

133

lex 有一些相同的属性，但不知何故，它没有像 yacc 那样广泛应用，可能是因为它更易于运行你自己的词法分析器。在 AWK 中，我们原本有一个 lex 词法分析器，但是，随着时间的推移，我发现它很难支持不同的环境，所以，我用手工 C 词法分析器把它替换掉了。这成了随后多年所有程序 bug 的来源。

除了 lex 和 yacc 以外，还有没有其他技术使得语言或者程序开发变得更简单、更容易，或者是功能更加强大？

Brian：此后，随着 Unix 操作系统的兴起，意味着所有的计算任务都变得容易了。创建 shell 脚本的能力，运行程序并捕获输出的能力，想象一下，或许当机器缓慢的时候，可以编辑并制成不同的东西——这使得它相当不同。总的来说，有了这些工具，特别是像 sort、grep 和 diff 这样的核心工具，使得人们能够看到你在做什么，并保持跟踪一部分代码。

如果没有 Make，我无法想象怎么去编译程序，不过，当然我也无法设想一个没有 patch 命令的世界，而且那大概是 1986 年或 1987 年。

Brian：在我开始教学之前，我从来都没有用过 patch，因为我根本没有编写过这么大的程序，在这样的大程序中，patch 就更有意义，而不仅仅是拥有所有的代码。几年前，我决定要让我课上的学生知道 patch，因为有很多代码都安装了，特别是在 Linux 领域。这门课的作业之一就是要求学生从网上下载我编写的 AWK，添加特定的功能，比如说重复，使用 shell 脚本创造一些测试和运行它们，然后再给我们发送 patch 文件。这样，他们就会获得对某些开源程序的完全体验，并进行小幅修改，最后再把它发回来。我从来没想过要用 patch。我通常会让他们给我发送源文件。

使用 patch 这种形式更容易审查吗？

Brian：我想这是另外一回事。patch 文件更为紧凑，而且，你可以更快地看到它们干了些什么。

您提到了测试。现在，为了便于单元测试，您会以不同的方式来编写代码吗？

Brian：这种程序，我都写了好多年了，对它来说，单元测试并没有多大意义，因为程序本身太小，并且它们也是自包含的。单元测试的想法，就是在模仿的 main 函数中，加入一些“调用这个函数，看看它干了些什

译注9：CACM，即美国计算机学会通讯。美国计算机协会（ACM）每年都出版大量计算机科学的专门期刊，CACM 是其旗舰出版物，报道计算机技术，包括信息技术、计算技术、软硬件工程及其应用的最新进展。

么”之类的小东西用于测试，它对于这些程序来说没有意义，因此，我确实不在这个级别上进行单元测试。我在课堂上试过几次，结果都惨遭失败。

对于小程序来说，我更喜欢做端到端的黑盒测试。通常是以一种专门的小语言的方式，组织一些测试用例，然后再编写一个程序。这个程序应该能自动运行测试用例，并报告错误。它对于那些小的 AWK 程序来说很合适，对于正则表达式会更为适用。它也适用于 Base64 编码器和解码器，有时我会让学生这么干。对于这些来说，我进行的是外部测试，而不是内部测试。我不会在程序里放些什么东西来进行测试。

另一方面，还有一件事，如果是现在我就不那样干了，这就是通过断言和完整性检查功能，让内部一致性检查变得更容易，而且，或许还有更多的测试点或者途径来获知内部状态，而不必为此太伤脑筋，有点像搞硬件的那样使用内置自我测试。 134

这听起来好像是：测试代码，同时也是在调试代码。或许，这两者并没有很大的区别。

Brian：对我来说，断言的概念是：你可以在很大程度上确认在某一个特定点上是对的，但你尚不能绝对肯定，所以你使用了一个降落伞，以确保如果系统崩溃，你还可以安全着陆。这是一种糟糕的混合隐喻。断言和完整性检查是非常有用的，因为如果确有问题，你的调试将会容易得多，因为你知道从哪里下手找出错误。它还会告诉你，你可能应该使用哪种测试，而目前还没有使用。

我曾经尝试让学生去建立一个联合数组类，其概念基本上和 AWK 中的联合数组相同。他们是用 C 语言编写的，这意味着字符串处理通常会出现问题。我自己在编写时，写了一个独立的完整性检查函数来检查数据结构，并确保：通过对数据结构内部计算得到的元素数，与你从外部推算的元素数相同。

我想它就像各种版本的 `malloc` 一样，在每次事务之前和之后都检查。检查的理由是：“如果会出错的话，错就会出在这里，让我来确认一下吧”。我会做得更多。

是不是有一部分原因在于你是一个成熟的开发者，所以你能知道会产生这种，或者是因为这样做成本较低？

Brian：我并不敢号称自己是一个成熟的开发者。绝大多数情况下，我写的代码都比我愿意写的要少，而且我写出来的代码，质量往往很差，尽管我说得很好听。这更像是：“依其言而行事，勿观其行而仿之。”

我们的编辑曾经在某个会议上听到您力捧 Tcl 和 Visual Basic。现在，您对这两种语言怎么看呢？

Brian：在 20 世纪 90 年代早期，我便写了大量的 Tcl/Tk 程序。我对它的了解确实非常透彻，而且，我还编写了一些系统，至少在贝尔实验室有一些简单的应用。我能够做到接口速度很快。对于构建用户接口来说，Tcl/Tk 是一个很好的环境，而且，对它所有的后继者来说都是一个很大的改进。

作为一种独立的语言，Tcl 确实是风格迥异。它在自己的领域做得非常出色，但这通常是远远不够的，我认为很多人在使用它的时候会遇到很多问题，而且，如果没有 Tk 的话，Tcl 早就已经消失了。Tk 非常适合用于构建接口。

早期的 Visual Basic 是一种很好的 Windows 应用程序编写语言和环境。曾几何时，VB 是最流行的编程语言之一。用它来构建图形化接口非常容易，因此，VB 在 Windows 世界中的功能，就跟 Tk 在 X11 Unix 中一样：它是一种快速构建图形接口的方式。微软在慢慢地扼杀 VB，有鉴于此，我不会再用 VB 来构建任何新的应用程序了。C#自然就成为了一种选择。 135

当您要放弃某个特性或者某种想法，并且要求人们升级到新版本时，您是怎么想的？

Brian：不幸的是，这些问题没有什么正确答案：无论如何，都有人会不高兴。如果这是我写的程序，我就想让别人追随于我；如果是其他人写的程序，那么我想让他们继续维护我正在使用的那些特殊的结构。这两种情况我都遇到过。这么多年，我一直深感痛楚：贝尔实验室有多个不同的 AWK 版本。我、AL 和 Peter 有一个版本，还有一个来自其他组的、叫做 NAWK 的变体。他们想把语言往不同的方向发展，因此我们必须同时面对两个不太兼容的版本。

有一个一致的观点：那就是“什么让你的生活更加方便？”如果清除掉那些难以维护或者难以解释的特性，使得长期维护某个程序更加容易，这无疑是一个方面。如果为了将某个程序升级到新版本，你就得重写大量代码的话，这是另一种烦恼。

Brian：在某些情况下，这是可以控制的。例如，微软有一个从 VB 6 转换成 VB.NET 的转换向导。该向导的早期版本不能真正地胜任这项工作，但是新版本就好多了，因此，从这个意义上说，它就不仅仅是“可用的操作”那么简单了。

在何种程度上，设计师应该把优雅的接口作为实现的主要目标？这是不是您一直最关心的问题，或者说，它要取决于您的其他目标？

Brian：如果这是一种编程语言，你必须考虑人们是如何编写程序的。他们会编写什么样的程序？在你将它固定（freeze）以前，你应该尝试自己编写很多例子。如果它是一个 API，你就需要认真地思考一下人们会如何使用它，以及它如何处理那些棘手的问题，比如“什么资源归谁所有”等。

2007 年 5 月，Michi Henning 在 ACM Queue 杂志（译注10）上发表了一篇关于 API 设计的文章，这篇文章写得非常好，我在准备课堂讲授 API 之前，又把它重新读了一遍。他在文章中提出了一个观点：API 现在非常重要，因为 API 用得越来越多了，而且它们还要处理更复杂的功能。

Web 服务 API 就是一个例子。例如，Google Maps 的 API 现在非常庞大。我记得三年前跟它打交道时，它还没有这么大；它似乎还要扩大。依我来看，它做得相当好。其他接口就不太好用了。把它们都做好确实很难。当然，如果你改变了想法，又该怎么办呢？

136

您在升级所有的服务器时，可以设置一个标志日。

Brian：或者您改一些名称，它就会向上兼容。

那是您能设计的事吗？Stuart Feldman 不是说过：“我不能修改 Make 中的 tab，因为我已经有了 12 个用户”？

Brian：是的。那是 Make 的棘手问题之一，而且我确信，Stu 现在和过去一样对此很不爽。一旦你真正有了用户，就很难修改了。Joshua Bloch 就 API 设计说过，“API 是永久的”。一旦你完成了它，就难以修改了。有时候你可以做些转换器。我们讨论一下 VB 的转换器吧。很久以前，Mike Lesk 修改了 TBL。在过去，表是用列来实现的，他觉得用行比较好，所以他编写了一个转换器。它并没有那么完美，但足以读取旧表并把它映射成新表。有些时候，这个方法是有用的。还有一个从 AWK 到 Perl 的转换器，它的功能虽然相当有限，不过它足以让你顺利起步了。

译注10：ACM Queue，美国计算机学会《队列》杂志，它是一份应用性的出版物，描述和定义了计算机软硬件发展中面临的技术问题和挑战，帮助读者具备敏锐的思想，并追踪创新性的解决方案。

您有这么多年的丰富阅历，您觉得可以从中得到什么经验和教训呢？

Brian：要认识到你想要做的事确实很难，但另一方面，你要继续工作、继续尝试并且不断修改，直到满意为止。不要指望一次就能成功。

对于某些系统，你会感觉到他们发布的是第一个版本。根据发布过程你就知道，它肯定是有问题的。试想一下，连贝多芬这样的天才，他的手稿也是改得乱七八糟。莫扎特很可能是唯一一位能够一气呵成创作出完美音乐的作曲家。

千年一遇的天才和普通人在交错工作方面的表现是截然不同的吗？

Brian：Isaac Asimov 在自传中说，他仅仅是把文字写下来然后就出版了，而且，他的作品确实是相当棒的。他说他从来没有重写过，而且对他来说这样很好，但是我认为不能把这当成模板。

大学某个房间的墙上有 Paul Muldoon 写的一篇诗，它使我想起了贝多芬的手稿。在一张纸上，没完没了地勾来划去，无数次地推倒重来，一遍又一遍地修改；有人把它装裱起来挂在墙上，提醒人们第一次就把事情做好是多么困难。程序也是一样，不要指望一次就能写好。

6.10 改变世界的“位”

为了给您的数据库项目添加一个扩展语言的分析器，你和 AL Aho 展开了讨论，而 AWK 就是起始于此，这是真的吗？

Peter Weinberger：我记得并非如此，尽管记忆难免会有错。我曾在一个做数据处理（在 Univac 计算机上）的部门工作，而 Al 和 Brian 对在 Unix 命令行中加一些数据库的东西很感兴趣。很可能是他们已经开始了更为雄伟的计划，但我的记忆是，我们最早确定的是：最有效的方式是从扫描数据开始。

您为什么会对从文件中抽取信息的工具感兴趣？您为什么避开了插入数据这个特性，举个例子？

Peter：Unix 命令行工具的统一特性之一是，它们都是处理由行组成的文件（而且，那时使用 ASCII 码）。这样通过编辑器就能插入数据，不然的话，更新文件常常意味着通过修改内容来创建新文件。也可以使用其他工具，也能完成任务，但它们不是主流。

我听说您聚焦于读取数据，因为您不想在编写程序时处理并发问题。

Peter：噢，不完全是这样，那不是事情的真相。

您现在还会做出同样的决策吗？

Peter：不会，我认为如果我们现在再来编写它，并且要记住不要有太大的野心，我认为其中不会有任何用户可见的并发性问题，但我相信它将被构建出来，发挥本地多核或并行机制的优势。我确信它会给我们带来一些麻烦，但是我们应该可以克服它。这儿有一个有趣的问题，或者是可能有趣的问题，那就是：“这在多大

程度上改变了语言设计呢？”

我不知道；你必须考虑这个问题。如果你认为你有空闲的 CPU、大量空闲的 CPU，那么你就可以做好几件事。其中之一就是你可以说，好的，我们不打算用它；我们只是把它放在这儿，不管正在运行别的什么东西。在 AWK 中的情况，或者说跟 AWK 很像的情况，也是一种不错的选择，因为如果你通常认为它设计用于管道，那么管道里的其他东西也需要 CPU 处理时间。

另一方面，如果你认为它会用于相对复杂的文件转换，你也许要拿出可以使用几个处理器同时在上面运行的东西，当然这个我们无法做到，因为当时机器的运作方式不是这样。

138

你认为在什么情况下 AWK 会比 SQL 更合适，举个例子？

Peter：好的，其实，它们本质上是不可比的。AWK 没有显式类型，而 SQL 却是强类型的。也就是说，AWK 读写的都是字符串，不过必要时，它会把字符串预处理成数字。SQL 使用 join（连接），但要让 AWK 做同样的事，人们会在它前面运行一个程序，很可能是“join”。SQL 做排序和聚合，但在 Unix 环境下，这些是由 sort（排序）来做的。然后再通过 AWK 或者是其他 Unix 命令进行管道传递。简而言之，AWK 计划用做通过管道联系在一起的命令序列的一部分。SQL 计划应该以用户知道的某种模式，与这些隐藏在不透明结构中的数据一起使用。最后，SQL 还有多年的优化查询工作来支持，而在 AWK 中，则是采用所见即所得的方式。

以文本文件的形式存储（Unix）日志并且用 AWK 来操作它们，这有什么优势？

Peter：文本文件是一个大赢家。查看它们并不需要特殊的工具，而且，所有的 Unix 命令都可以使用。如果这还不够的话，它们还易于转换并且加载到其他程序中。各类软件都有一个通用的输入类型。此外，它们与 CPU 字节顺序相互独立。即使是像压缩它们这么小的一个优化，也暗示着人们记住使用哪个压缩命令，而且，这通常会有好几种选择。至于说使用 AWK 操作它们，如果命令行管道做了它需要的操作，那就很好。否则，用 Perl 和 Python 这样的脚本语言来读取文本文件就足够了。最后，也可以使用 C 和 java。

用文本文件来记录日志非常好。在过去，反对它们的争议是它们必须分析，而且数字必须要转换成二进制等。但是后来，这种操作占用的 CPU 时间无足轻重，而且，与 XML 相比，文本行的分析也变得无关紧要。另一方面，固定规模的二进制 struct 不需要分析，但这很不寻常，而且这是使它与众不同的一个罕见情况。

AWK 是 Unix 关于“许多小程序一起工作”这个概念的一个成功的早期证明。这些程序大多是面向文本的。这种概念是怎么应用到非文本数据和多媒体上的呢？

Peter：这对于弄清楚什么是“Unix 概念”非常有用。这是一种程序风格：很多程序只有一个输入和输出；支持命令行语法；系统支持所有输入和输出的统一（读写系统调用，无论是何种设备）；而且，系统支持（管道）不必命名和分配临时文件。代码转换和压缩是非常适合这种理念的好例子，即使是音频数据或者是视频数据。不过，即便是对于文本，也有很多应用程序没有采用那种工作方式，特别是人们想要同它们交互时。例如，spell（拼写）命令产生了一个字的列表，尽管有误拼，但它不会相互影响；用户必须返回并且编辑他们的文档。

139

因此，你的问题本质上可能是，“如果我们只有命令行，应该使用什么命令来处理数据或多媒体呢？”但是，这与事实相悖。现在，我们有了同计算交互的其他方式，而且，对于划分任务来说，我们也有了更多的选择。新的方式并没有必要比以前好或者坏，仅仅是不同而已。举例来说，TeX 与 Word 之类的程序比较，一个就比另一个好吗？我想它们有很多的共同点。

您认为命令行工具和图形接口各有什么限制呢？

Peter：这个话题是老生常谈了，而且，它们的边界也变得有点模糊了。或许这需要一篇深思熟虑的论文来解释。这里只给出一个粗浅的回答。如果我需要把大量的程序组合起来，那么一个调用命令行工具的 shell 脚本就足够了。这也是确保各种组件的选项和属性保持一致的一种办法。不过，如果要考察数量适中的备选方案并做出选择时，使用图形接口可能会更好一些，而且，更有可能会让所有的信息都保持井然有序。

很多受访嘉宾都强调学习数学对于成为一名更好的程序员的重要性。我想知道，我们可以在需要的时候学习需要的东西，要达到什么样的程度。例如，通过因特网，您可以很快找到要学的东西，并开始学习，是不是？

Peter：既是又不是。不幸的是，对于学习某些东西来说，你不仅要思考它们，还必须有一些实践，因此有一些东西——你可以求助于因特网，你会边读边说："对，这么干没错"。但是，还有一些东西，则非要数年的努力积累不可。因此，你正在做一些项目，而且确定需要了解线性编程来解决问题；你从因特网中得到的信息可能并没有什么用处，而且如果你需要在一周之内解决这个问题，就不太可能选择一种需要学习很多新东西的方法——即使这种方法会更好，除非你已经了解它了。

数学在计算机科学，特别是编程方面，会起到什么作用？

Peter：数学是我的专业，因此，我愿意相信数学是基础。但是，计算机科学的很多领域和很多编程，没有任何数学基础也可以很成功。数学到底有用还是没用，这也是分层次的。对统计和随机过程没有概念的人们通常会被现实世界的数据一次又一次地误导。图形中有数学问题，机器学习中也有很多数学的东西（我想统计学家会认为这是一种回归），而且密码学里面也有很多数论问题。没有某些数学知识，人们就会无法理解计算机科学的很大一部分内容。

您认为研究理论和构建实现的区别是什么？

Peter：在最高层次上，当你证明了一个定理时，你会了解到关于这个领域的一些东西，而这些东西你以前一直怀疑是不是真的。它是无条件的知识。当你编写程序时，你可能做到以前无法做到的事情。

从某种意义来讲，你已经改变了那个领域。大多数改变是很小很小的。数学和编程很不一样。或许，解释这个观点的最简单方式就是，对数学论文和理论证明以及定理证明者编写的程序进行比较。论文很短，而且通常会有很深刻的见解。机器证明则是既不简短，也不深刻。编写程序有一些机器生成证明的特征，程序中的所有细节都必须正确无误，这对于程序员的理解和测试技巧都是一个很大的负担。

构建的实现是不是让您学到了更多的知识？

Peter：当然了。特别是你学会了：应该先把它扔到一边，再重新来实现它。任何项目都有几十个或者更多的设计决策，其中的大部分在当时似乎是中立的，或者是基于直觉做出了方案选择。几乎无一例外，当代码真正运行时，很显然我们当时应该做出更好的选择。然后，随着时间的推移，代码可能会用于意想不到的情况，这时就会发现有更多的决策看上去非常糟糕。

函数式编程会有帮助吗？

Peter：如果问题是，数学性更强的函数式程序的表示结果是否要比普通程序更好，我看二者并没有很大的区别。任何一种单赋值的语言都更容易分析，但这并不会使程序更容易编写，也没有确凿的证据表明程序更容易

编写。事实上，关于语言、编码技术、开发方法和软件工程的大多数比较性问题，一般来说都是极不科学的。

以下引自 R. Bausell 所著的《Snake Oil Science（蛇油科学）》[Oxford University Press（牛津大学出版社）]：

> 证据表明，在判别什么可行、什么不可行方面，涉及数值型数据的严格实验（比如随机实验对照实验等），要比依赖我们尊崇的专家意见、经验、直觉或者教诲更加可靠。

软件仍然是一种手工艺品，它很像是制作家具：既有 Chippendale 的大作（译注11），也有普通工匠的手艺，还有少数老艺人的绝活。我扯得有点远了。

为了成为一名更好的程序员，您有什么建议吗？

Peter：“学习数学”怎么样？噢，好的，可能会有其他更好的建议。“理解浮点”怎么样？或许这个建议也不行。其实这是仁者见仁，智者见智的事情。

我认为学习新技术和新算法很重要，否则，人们很快会变得过于专业和狭隘。另外，现在人们应该精通于编写安全可靠而且健壮的代码。现在有很多针对用户和系统的攻击，因此，你应该保证你的代码没那么脆弱。这对于网站来说是一个特别棘手的难题。

应该什么时候教授调试，又该如何教授呢？

Peter：讨论调试应该是所有编程课程不可或缺的部分（也是所有的语言设计不可或缺的部分）。编写在独立的机器上运行的正确的顺序程序是很困难的。编写多线程代码甚至更难，而且调试工具也不是非常令人满意。设计需要考虑的一件事就是，它是否能使调试更容易。毫不夸张地说，像我这样的程序员，要么是在决定下一步干什么，要么就是在调试。几乎没有时间来考虑其他的问题。

关于设计或编程，您认为您犯过的最大错误是什么？您从中吸取了什么教训？

Peter：我不知道某个具体的最大错误。人们会经常出错。你从错误中学到了（或许没有清晰无误地表述出来）一组通用的设计原则。然后，你开始广泛地运用，然后它们又失效了，然后又可以向其中添加新的经验教训，或者你的代码可能会一直带有你逐渐过时的设计规则的痕迹。我发现没有在错误消息中放入足够的有用解释，而且结果通常是重新返工并添加细节。这是一种典型的冲突：如果出现了错误，你想要全部的有用信息。如果出现错误，你又得键入许多东西，并会占去屏幕的很大空间。这是一种平衡。

对于 AWK 您的最大遗憾是什么？

Peter：我认为，“使用空白符作为字符串连接符”的妙招并没有像我期望的那样奏效。显式操作符应该能让事情变得更清晰。想要更短的命令行和允许使用大程序，它们的冲突也常常使句法饱受其苦。最初我们并没有考虑后者，所以我们的一些选择缺乏创见。

您对什么受欢迎（或者有用）感到很惊奇？

Peter：整个语言都比我们看到的或者我希望的更受欢迎。设计指导思想之一是让那些熟悉 Unix 类语言（特别是 C 和 grep）的人们容易学习 AWK。这不会使大批秘书（他们曾经被叫做书记）或者牧羊人成为 AWK 的使用者。不过，在 20 世纪 90 年代早期的一个婚礼上，我碰到一个牧羊人用 Unix 来保存他的记录，他还

译注11：Chippendale（齐本德尔），18 世纪的一位英国木工，以其制作的新古典主义风格的精美家具，尤其是椅子而著称。这种风格的家具外廓优美，并常有华丽的装饰。对同时代的工匠有广泛影响。

是 AWK 的超级粉丝。我怀疑他现在还是那样。

你如何激发软件开发团队的创造力？

Peter：编写高质量软件的最佳途径是一帮有才能的专家一起合作，他们都非常清楚到底想要做出什么。还有其他方式，不过他们需要更多的工作。如果缺少有才能的程序员，我不知道如何才能创建出优秀的软件，即使也是有可能的。

作为一个团队，你们是怎么开发一种语言的？

Peter：我们全都是一起讨论句法和语义，然后是人人都编写代码。随后，我们任何人都能修改代码。对于大多数情况来说，它不是哪一个人的代码，尽管这些年来一直是 Brian 在维护它。我们还制定了有限目标。我认为目标机也帮了我们大忙，虽然它只有 128k 字节的内存。

为了设计，我们围坐在会议桌旁，一边讨论，一边写写画画。然后在编码时，可能会发现我们遗漏了一些重要的东西。这又需要进行非正式的讨论。

如果在代码库中发现了一个反复出现的问题，到底是局部就事论事，还是全局通盘考虑，您如何判断哪一种是最佳解决方案？

Peter：有两种软件项目：失败的项目，和那些遗留下来的烂摊子。避免第二种情况的唯一方式应该是根据环境变化重新编写代码。麻烦在于它的成本极高，大多数项目负担不起，因此，现实压力迫使人们局部就事论事处理。随着局部修改达到一定程度，代码变得非常僵化，实际上已经难以维护。如果没有最初的开发者，或者是非常好的说明书，那也很难重新编写代码。生存下来可能会很困难。

如果您有一个建议，读者最应该从您的经验中学习什么？

Peter：我引用（也可能是错误的引用）爱因斯坦的话："尽可能简单，而不是更简单"。

诀窍就在于不是放任自流，它很容易变成那样。如果人们确实提了一些要求，那么你也可能把它放进去了。就需要判断：是要变得简单，而不是比必需的更简单，无论是谁提出的要求。

最简单的或许也能工作？我想那是 Kent Beck。对于简单性，以及反对添加那些当时你并不需要的东西，您是如何认识到的？

Peter：它取决于你跟谁在一起。对于很多人来说，"你能给父母解释它吗？"就是一个很好的测试。有时那或许是不可能的，不过作为起点，它对我来说看起来非常合理。更常见的测试是如果考虑你希望使用它的人们，"你能给中间用户解释它吗？"而不是"最聪明的用户能解决它吗？"

6.11 理论和实践

您在加盟贝尔实验室之前是教授数学课程的。我们应该以教授数学那样的方式来教授计算机科学吗？

Peter：我们教授数学有几种不同的原因。其中之一是培养未来的数学家，在某种程度上，这是我在教数学时

要干的事。另外一种原因是因为数学太有用了。还有，我认为“数学是什么”要比“计算机科学是什么”要清晰一些。

143 在计算机科学中，有各种不同类型的编程，并且很难知道关于那些应该考虑些什么。那里会出现所有数据结构，各种各样的算法和复杂因素。计算机科学的用户需要什么和计算机科学是什么，这二者之间的差别不是很清晰，至少要搞清楚人们在考虑些什么，潜在的数学用户需要些什么。因此，当你教数学的时候，你知道工程师需要什么；现在，我假定你了解搞统计学、经济学或其他所需的人们，但我认为了解他们的需要对数学家来说相对更简单一些。

另一方面，我认为计算机科学家应该多了解数学，所以从我还是数学家的时候，就有了很多积累。

因此，与我们笼统地称之为计算机科学系相比，至少在这个国家，这个问题在吸引主修学生方面已经有了一些困难，至少在过去的几年中是这样；它的原因并不清楚，不过成功吸引到更多主修学生的那些系，他们的课程也改了许多。因此，计算机科学应该教授的内容是在不断变化的。

从您前面的回答，我得出一个印象：你建议编程的最好的结合点介于纯理论的方法和纯实用的方法之间。纯理论的方法可能离现实生活的需要太远；纯实用的方法可以把各种来源的代码聚集起来解决问题。这么理解对吗？

Peter：对，没错，不过我认为，更大的问题是很难知道应该在哪里画线。它取决于你为代码设定的目标。如果你期望人们长期使用它，那么代码的 bug 就应该很容易修复。

其他的困难是你很快就会有太多的用户，这意味着很难修改任何设计的问题。如果我只是为了自己的需要而编写它，那么，如果我不喜欢，我就可以去修复或者修改它。如果你是为小用户群编写的，那么，当你做一些不兼容的修改时，人们就得抱怨一会儿，因为他们知道这是实验。但是，如果你是为大用户群编写的，你想要进行不兼容的修改就会变得非常困难，因此，无论你以前做出的是什么决策，你都得继续坚持。

这可能是遗留软件的问题之一，当人们使用不同来源的代码块时，这些代码块中的问题会继续存在几十年。

Peter：是的，我认为许多代码是很久以前编写的，它们的作者根本就没有想到这些代码会存在这么长时间。

AWK 继续存在的一个因素是：这么多用户使用别人编写的脚本，并修改它们来做其他的事。

Peter：没错，那是对的，而且，实际上这就是一个设计目标。这就是我们认为它应该被用的方式，我们认为它应该有很多应用。人们愿意使用几乎能满足自己所需的代码，只不过是要稍微改改。

144 **例子中的这种编程思想是否适合于更大的项目？**

Peter：我认为它适合不太大的项目，因为，例子应该小到足以让人容易理解。在你可以做到的程度上，如果它只有几行代码，那么它就是最容易的。你很可能弄了满屏的代码，而且期望人们也跟着这么干。它需要的是足够的简单，以便你只要看一下代码，就知道哪儿需要修改；或者至少足够了解哪儿需要修改，以便让你运行实验，看看你是否改对了。

编写短小的“一次性”脚本，这个主意听起来非常诱人。您在大代码库和其他编程语言方面的经验，是否会告诉您何时修改代码库，以及何时重新启动？

Peter：在实践中，很难重新从头开始。如果你的用户群体很小，你可以跟他们对话。另外，如果你的代码有

一个明确定义的接口，那么也有可能重新开始。如果接口没有明确定义，而且用户群体很大，看起来就很难避免修改代码。不幸的是，关于比较温和的（less-drastic）升级也是如此。因此，这实际上是好消息，也就是说，对于用户来说，既然连任何实质性的升级都需要修改代码，那么，动作更大一点的重新实现看来也没那么糟糕。几年以后，新代码几乎肯定要全部重写。用户会以开发者未曾想到的方式来使用它，而且很多实现上的决策被证明并不是最理想的，特别是对于新的硬件来说，更是如此。

AWK 的经历稍微有一些不同。我们确实重写了好几次，但后来我们就宣布它“定型”了。它本来是可以升级的，但是我们的所有想法似乎都与基本原则不符。我认为那几乎是完全正确的决策。我们都去干其他事了，而没有去扩展系统。我认为，它在现代的小众市场中，唯一缺失的就是没有使用 UTF-8 作为输入。

Brian 说您是一位速度很快的实现者。您有什么秘诀吗？

Peter：我认为没有什么秘诀。人们各有不同。例如，我不确认是否是现在必须做，我就愿意尽快做完。我认为部分原因在于乐观的忽略。这就是你相信你能把它写下来，而且这就足够了。另一部分原因是觉得工具和语言怎么样，它们是否可用。有些人觉得工具很合适，而有些人却不这么认为。这就像为水彩画配色，有些人觉得很容易，有些人觉得很难。

我认为，如果你要是为了生计而专业编写代码，你就会发现编程相当容易；否则，你始终都是在挣扎。这就像编写短故事一样：如果你觉得在某个层次上编写短故事不容易的话，我认为你会发现一切都很难做到，尽管我不知道，因为我不能做这样的创作。要让事情最终搞定，需要大量的工作。

您是不是先编写原型，然后再修改代码，最终使它达到专业级的质量水准？还是先实现您的想法，然后完全重写来完成工作？

Peter：我认为这个无法预先判别。当你开始编写原型时，有时你可以知道要做什么样的妥协。有时，这些妥协处理意味着，原型会与“通过简单修改就能生成程序”相矛盾。无论如何，也许你所做的一切事情都能使它变得困难，并且在这种情况下，你只好不得不重新编写它，不过，如果你够幸运的话，也许你可以逐渐地修改它。你应该预计到可能必须把它丢掉并且重新编写。

首先，你未必会做出足够正确的决定。你编写它、开始做试验，还会做一些修改。不久之后，除非你很幸运，否则，代码就开始变得很糟糕。实际上，它至少需要重构，但很可能仅仅是重写。那很可能正是我所期望的：你最终确实是重写了它。毫无疑问，AWK 的第 1 版实现纯粹是一个概念验证性的东西，因为它生成的是 C 代码；当然，这与你想要的使用方式完全不一致。

BASIC 的创建者 Tom Kurtz 说，编写代码使您理解了这个问题中您未曾想到的那些方面。

Peter：没错，你不可能聪明到还没遇到问题的时候你就开始思考它。我认为你在雇人时，其实是在求证：对他们来说，编码是不是一种自然的表达形式。这是他们表达自己算法思想的方式吗？

编写软件和创建语言有什么区别？

Peter：在某些方面，编写语言要比一般的写软件更容易，不过，我不确定这是不是对的。我认为，它集中了你的选择，因为它需要相互适应，而且做一件事的方法相对较少。一旦你已经决定了语言的主要特性，你就已经在脑海里形成了许多框架：函数会怎么工作，你是否准备做垃圾回收，等等。什么是语言的原语？实现是以层的形式呈现的。我认为这比较容易。当然，如果这样的事情稍微有点晚，你已经发现你做了一些非常糟糕的选择，那就必须全部推翻。

实现会影响语言的设计吗？

Peter：噢，肯定会的。毫无疑问，我想你可以理解。例如，我认为，很长一段时间垃圾回收是一个比较特殊的问题。Lisp 的那帮家伙在实现它，某些函数式编程的家伙也在实现它，其他许多人都在等待，因为不清楚它会怎样以类似于 C 语言那样的方式去工作。然后，例如，Java 的那帮家伙会说这就是我们要做的事情。通过对语言的特色做出相对小的修改，这是很不一样的折衷权衡，我并不是说这事发生在 Java 身上，而是通过放弃可供程序员访问的实际内存地址，你可以决定是否想尝试垃圾收集或者压缩垃圾器等。

146

我认为，现在的语言没有真正经受多少考验（即便是垃圾收集），它们离完善还差得很远。反对内存分配也会令人烦恼不已，所以它从来都不只是一些无关紧要的琐碎之事。但是，关于如何实现语言，现在又有了更多的了解，而且对于你想做的事情，特别是你想要做一些轻量级的实现时，你就有了更多的选择。

如果你的语言想要一个难以实现的特性，它是否值得就值得商榷，因为它很难实现，并且你在尝试一些轻量级的实现。如果你在尝试编写一种语言，它有可能解决困难问题，那你可能就必须干一大堆事，而且你不得不忍受任何困难的折磨。

语言会对程序员的生产率有多大影响？程序员的能力会有多大差异？

Peter：小伙子，我希望我知道这个问题的答案。我曾经认为我知道这个问题的答案。很显然，程序员的能力差异很大。实际上，他们的能力会相差 10 倍，或许是 10 倍以上。这在某种程度上算是软件工程，因此，对此没有任何实验证据，我的观点是语言不应该对此产生影响。也就是说，给定一组人和一些项目，用什么语言确实是无关紧要。但是，对于单个程序员来说，我想不同的语言是会有影响的。我认为，就个体来说，或者先学什么，或者是在其他方面，总之，他们都会找到某些语言会比其他语言更易适应。在这儿你就能听到这些有趣的辩论。

例如，很明显，Lisp 应用程序难以真正实现与 C 相同的功能，同样，C 应用程序也难以真正实现与 Lisp 同样的功能。要不是有很多程序可以使用多种语言，我不确定，在实践中所有的程序员使用各种语言都同样很舒适，我也不知道为什么。不过，成为语言专家当然是需要时间的。另一方面，对于一些人来说，学习某些语言要比学习其他语言花费的时间要少。

人们对语言有很大的争议，许多很好的特性，它们实现了哪些，没有实现哪些，哪些是糟糕的，人们在诸如此类的事情上争论不休。但是，目前尚不清楚它是否真的这么重要。更大的争议是：你可以使用任何语言来编写 Mars Lander（火星着陆器）软件；每种语言都有自己的属性，而且它更加依赖编写它们的人，更加依赖他们如何组织它们，而不是更加依赖于语言本身。每个人都在极力鼓吹自己的选择，不过我不敢相信那些。

C 语言不支持对象，但另一方面，你可以构建 Unix 系统的小工具、组件，它们组合起来能够构建复杂的特性。在语言内构建对象作为大程序的一部分，和构建组件作为系统的一部分，前者在多大程度上要比后者更好？

Peter：这唤起了我脑海中的两个问题。一个是绑定的问题或者说是模块化的问题。这个问题在于，你放进语言中的东西同你想要构建的工具恰恰是相反的。你将得到一个更紧密，当然也更复杂的组件关系，如果它们都在一种语言内部的话。其中的一些仅仅是计算效率问题，不过，我认为也有一些是概念一致性问题。

147

另一个问题与“面向对象”作为总体思路有关。我认为面向对象各种各样的成功可能是太过热情了。撇开这个尚有争议的问题不说，如果你粗略浏览一下语言，很多人会声称“我们是面向对象”的语言，当你进一步仔细考察的时候，你会发现他们都是在做全然不同的事情。“面向对象”这个术语究竟是什么意思尚不明确。

事实上，人们陷入了这种极为混乱的讨论之中，因为，这里有一个天然的诱惑：你相信你在语言中关于对象所做的一切，正是面向对象的真正含义吗。我认为，这个术语基本上没有一个简单、相对浅显易懂并被广泛接收的定义。

编程语言的选择会对代码安全性有何影响？

Peter：毫无疑问，你需要有一个工具，帮你管理那些与安全性相关的各种事务。我想，大致说来，程序中有两类容易出错的事情与安全有关。其中之一是一类逻辑错误，因此，你可以告诉程序一些东西，而且它出现了错误，给了你优先权，或者做了一些它根本不该做的事。另一个是缓冲区溢出，这是可利用的各类实现错误，这是人们没有考虑到的实际 bug。而且，我认为大多数错误都不应该出现。由于各种各样的粗心编程，低级语言助长了缓冲区溢出的出现，因此，要想编好程序并不容易。

我注意到，曾经有一段时间盛传微软的 Vista 就是把所有的 C 和 C++程序重新用 C#编写了一遍，因此，你也不用再处理缓冲区溢出的问题了，因为根本不会有缓冲区溢出。不过，这当然没有奏效。取而代之的是，他们使用可执行机器语言来做这些极为庞大的复杂工作，试图使缓冲区溢出之类的东西难以利用。

还有另外一类安全性问题，发生在程序的“接缝”之间，因为许多接口并没有明确地定义，或者，从某种意义上说，根本就没有定义，除了很不正式的（比如 HTTP 和 XML 的跨站点脚本之类的东西）以外。我们需要采取一些措施来保证安全性，但是，我真的不知道应该做些什么。

如果语言的设计几乎可以避免某些特定的问题，那么这种做法又能对安全性有多大的帮助呢？

Peter：噢，尽其所能吧，不过，正如我们前面讨论的那样，还不清楚这到底能有多大的帮助。有一段时间，我在编写很多 Python 程序，而且我也遇到了一些这样有趣的 bug，当然，这些 bug 都源于考虑不周和不良风格。但是，Python 中的缩进问题是，一旦循环变得太长（我使用了一个嵌套循环）而无法到达循环末尾，为了退出循环并做些别的工作，就需要退回两个 tab（制表符），而且，我没有给它两个 tab；我只给了它一个 tab，因为我想这足够了，因为在屏幕上看起来是足够了，而且，这当然意味着我每次通过外部循环来做这些都要花费很大的开销，这是非常愚蠢的。尽管程序仍然没错，但是它速度很慢。

我想不管语言设计得有多好，程序员总有可能会犯愚蠢的错误。而且，你是否能以相对科学的方式，使它的可能性变得更小或者更大，我并不知道这个问题的答案。软件工程在很多方面的表现都差强人意，因为它很多都来自趣闻逸事，并是基于人们的判断，甚至是人们的美学判断。我不清楚人们是以什么标准来讨论语言的，这些标准是否与编写正确的、可维护的或易于修改的程序直接相关。

一般来说，研究会有助于实现，不过设计方面通常反映了设计者的个人喜好。

Peter：没错，事实上我认为，真正成功的语言里面都有很多小的细节，这些细节并不是那些出现在文献中的直接例子。人们决定某些事做起来会很有趣。对于思考、讨论语言来说，编程语言中的所有东西都是有用的，但是，你想要用语言做什么或者怎么使用语言、编写代码、使编程流程更好以及维护更加容易等，这些都是不清楚的。对此，每个人都有自己坚定的看法，但是对我来说，我并不清楚为什么我们要相信什么。在这方面，目前似乎还没有任何科学性可言。

难以使用一种科学的方法来设计语言，部分原因在于，我们没有科学的方法来衡量语言的好坏。

Peter：是的，我认为说得很对，或者总的来说，编程的好坏也是一样，而不是仅仅是语言。有许多人认为他们知道解决方案，但是，我并不清楚为什么应该相信他们，因为很明显，开发程序有很多不同的成功方法。

您如何为语言选择正确的句法？你更关注边角情况，还是一般的用户体验呢？

Peter：答案毫不令人惊讶：两个同样关注。它应该像在我们人类的智商范围能做到的那样合情合理，但是也应该清楚在边角情况下语义是什么。成功的语言会被很多人使用，大多数人并不会与设计者具有同样的观点或者美学判断，而且，如果他们不被奇怪的特性或者边角情况无故地误导，这就是最理想的情况。而且，人们会编写程序，用新语言生成程序，而且对于实现来说这是一个惊喜。

设计语言时，当您评估潜在特性的时候，会考虑调试吗？

Peter：这是一个棘手的问题。你所希望的是获得来自开发环境的大量帮助，使得开发容易实现。我们能做的就是，猜测什么时候能完成，告诉你参数是什么，还可以告诉你其他的好参考资料，以及查找定义。当然，如果你只知道它的功能，而不知道其名称，要查找它是比较困难的。我的意思是你能确认：在众多函数中，有一个以机器可读的形式格式化数字的函数，并且添加一个逗号或者类似的东西，对不对？但是，你是如何记得名称的呢？你又如何查找这些函数的名称呢？而且，当人们编写这些库的时候，他们尝试使用命名约定、非正式命名约定和类似的方法。但是像这样的东西很难扩展。

您对良好的错误消息感觉如何？

Peter：嗯，那个应该很好用。我常常抱怨错误消息，因为它们读起来像是程序给它自己写的便条，而不是建议用户应该如何修改，并且有些情况下，它们甚至更糟糕。

错误消息需要多详细的信息呢？

Peter：噢，他们应该尽可能地有帮助，但这并没有真正地回答问题。编程语言中有些特定类型的错误会比其他类型更难理解，尽管你肯定懂得启发式方法。因此，在类似于 C 的语言中，分隔符和括号方面的错误常常会困扰编译器，而且它们很难解释。所以，当你在类定义和下一个函数之间漏掉了分号时，人们就学会识别你的编译器在表达什么意思了。你会意识到这种愚蠢的错误消息与实际出现的错误实在是风马牛不相及。在编译器发现搞错了之前，它早已经处理到下一个函数了。当你忘了右大括号时，你得到的也是同样莫名其妙的错误消息；你需要识别它们是什么形状的括号。这个能够做得更好，但是它需要做很多工作，而且，是否值得这么做也尚未可知。

这个问题的另一种问法是：你喜欢愚蠢的错误消息吗，这些消息给你提供了程序认为错在何处的提示？或者是，你喜欢有帮助的错误消息吗，那些让你想起微软 Word 中过去常用的消息提示，它们给你提供的是从来不会很有用而且似乎常常错误的帮助？

我认为如果你想提供优良的错误消息，你就必须努力了解它们，以至于它们大部分是正确的；不过，部分原因是我们已经习惯了平庸的错误消息，我们可以从图中弄清楚发生了什么事情。

6.12 等待突破

为了支持大程序，你会如何改进 AWK？

Peter：假如又要编写大程序，又要应用 AWK，问题是，我们应该使用 Perl 呢，还是应该使用其他的语言呢？

哦，我认为，全部都用 Perl 来实现的这种想法并不正确，但是，如果你看看 AWK 的精髓，如果你认为它应该用来编写大程序，那么全部都用 Perl 来实现是可能的。

我认为还有另外一种答案，也就是我们该停就停，因为它看起来好像适合停止。我不记得是否给你讲过这个故事——在我的脑子里，这事似乎刚过去几个月，不过回想起来，它是在 AWK 内部发行之后，似乎已经有好几年了。我接到了计算机中心某位打来的电话，他在使用 AWK 时遇到了一些问题，我过去看了看他的程序，我想 AWK 应该像我们想的那样只是一两行的小程序，对吧？而他用 AWK 为一些深奥的硬件写的汇编程序，足有 55 页代码。我们当时就惊呆了。事实上，这种干法并不奇怪；人们确实会用只有很少结构的语言编写很长的程序，但是，这对我们来说太不可思议了。

150

Brian 说过，实际上，每当你开始设计一种小语言时，人们开始使用它，然后就会提出想要加入循环或者其他什么，因此，每次你都不得不停下来，否则的话，语言就会变得……

Peter：变得越来越大，而且，你必须决定是否想要那么做。

如果你想要构建一种通用语言，那么一开始就在心里牢记这个目标，而不是从小语言开始，并在它成功后把它变得大而全。

Peter：我认为这话说的也对。我可能已经说过，这是另外一回事。“大”是人们编写程序、语言、分析器或者编译器的一种倾向。他们把人看成是输入者，但是你会发现，人们会出于各种原因编写程序来产生输入。我认为第一个典型例子就是编译器，因为，在类 C 语言中，此前没人见过一条有 80 000 个 case switch 语句的。他们认为人类不可能敲出来这样的语句。代码编写者或许从来没见过一条这么多 case switch 的语句。各种五花八门的事情都会发生，即使是对于通用语言来说也不例外。

用户可以修改的可扩展语言怎么样呢？

Peter：这个嘛，我认为那得看它确切的意思是什么了。我的看法是基本上可以，不过也有许多限制，除非你是指 Lisp 一类的语言，在这类语言中，人们可以使用宏把这些东西加到语言当中。人们要往语言中添加东西，会有很多原因：或者是为了可表达性，或者是因为你需要把用其他语言编写的库包含进来，而这些库可以完成一些复杂的功能。

谢谢你，这是另外一个问题，当然在 AWK 中根本不会碰到这个问题。在你的语言中，纳入一个子程序，或者是纳入用其他语言编写的语言包，这个有多难？这个问题的答案很多，众说纷纭。

这看起来好像是在区分数学家和非数学家。数学和软件开发之间有区别吗？C 赢了；Scheme 没有赢。C 赢了；Lisp 没有赢。这是“更坏就是更好”方式的再次应验。

Peter：我认为，数学家和非数学家设计的语言可能会有区别，不过，Scheme 和 C 的区别，要比是否想要有一个简单的基础的那些语言之间的区别更大。在这里没有什么绝对的事情；我们不知道将来会发生什么。

毫无疑问，到底是什么因素促成了语言的成功，现在还没有什么一致的意见，不过，确实存在着许多因素。每个人都有自己喜欢的多个方面，而且根本没有人知道如何去把它们聚合在一起。Lisp 最初非常纯洁，它有一个非常简单的模型，这个模型很清楚它要做什么，而且模型自身也非常有效。可能有人会问，为什么他们不得不把语言弄得这么复杂？什么东西需要额外的复杂性，而他们却没有做？毫无疑问，所有的中级 Lisp，然后是 Scheme，最后还有 Common Lisp，它们都需要额外的复杂性。

151

很可能会考虑到两方面的事情。这个可能无法解决，因为我还没有完全想明白，我只是向前推进而已。一方面是程序员的方便。另一方面是程序的性能。我们最终会选择其他语言，不过，我认为，像 Lisp 这类语言可以解决其中的许多问题。

我相信，最初的 Lisp 和 Common Lisp 之间的一大区别是存在额外的数据类型、哈希表等。那些对于性能来说非常必要。另一方面——我对此几乎没有什么经验；实际上，我对 Lisp 了解并不多——在某些时候，Lisp 程序员开始把宏添加进来。用这种方式，宏是一个非常自然的事情。对于它们来说，仅仅是一种不同的评价环境而已，不过——这也是程序员的方便问题——还没有什么宏能完成的功能，编写它们的程序员却无法完成。

它们是一种力量倍增器。

Peter：那是人们的期望。不过，你得到的也是混乱的力量倍增器，而且被广泛地使用；这意味着别人根本读不懂你的代码。你很难描述清楚你的语言做了些什么，即使是非正式地描述一下。还有许多可笑的边角情况。即便是在这个相对单纯的环境中，你也可以看到在所谓的纯数学和完成讨厌的工作之间那种剑拔弩张的局面。

所有的 Lisp 语言和所有的其他语言，很多都具有一样的形式语义定义、具有相同的问题。有人至少参与了程序生命周期的两个阶段，外加一台计算机。为了使程序令计算机满意，对于人们编写的东西是什么意思，你需要有一个非常精确的定义。人们不需要那些难写的东西。一旦程序已经生存下来并被用了一段时间，人们就很可能需要修改程序，维护者必须能够读懂代码并进行修改。

依我来看（正如我们所说的，“虽然简陋，但是好用”），有些语言的很多特性虽然便于编写程序，但是很难维护。如果只是为了获得更大的冲击力，我们几乎可以全部选择面向对象语言。这种实用语言，只有编译器及其编写者清楚它们的精确语义，对于新代码的编写者来说，它是非常有价值的。有一些语言允许你解释你的意图，而且是通过一种除了代码之外没有其他指导的那些人能搞懂的方式来解释，这种语言也很好。我很难想到任何例子。当然，尽管我没有使用过所有的语言，但是就我所熟悉的语言来讲，在这一点上，它们的表现可能有的稍好，有的稍差。

152

您说的是两种不同的坐标系。

Peter：是的，没错。不过，我们编写代码是在很久以前，那是“巨人大行其道”的时候。矮人也在地球上活动，不过是屈居笼中。没有人认为他们的代码能持续 30 年。如果你说“Unix 会仍然存在”，或者“FORTRAN 会仍然存在”，我们所有人的回答自然都是“是的，但我们会重新编写它”。这就是我们的生存之道。你创造了它，并重写了它。每一次，它都会稍微有一点不兼容，并会更好一些，直到你成为“第二系统效应”（译注12）的牺牲品为止。在这种情况下，系统的兼容性变得很差，系统越来越糟。我们稍微抽象地理解了这个概念：只有两类软件项目，即失败的项目和遗留下来的烂摊子。

我们无法理解：你们不能年年在编写软件又年年在重写。但它合情合理，而且，你要么把时间花在重新编写旧软件上，要么就对它放任自流。你无法二者兼顾。在某种程度上，维护问题越来越突出，尽管我这么说是因为我花了大量时间为 Google 做维护，这个问题对我来说非常突出。

译注12：第二系统效应（Second System Effect），又称第二版效应。系统设计师设计第一个系统（或系统的第一版）时，往往自信心较弱，会量力而行尽量剪裁要实现的功能。设计第二个系统（或第二版）时，设计师因为有了经验，很容易信心膨胀、过高估计自己、过于追求产品的完美，结果往往会过度设计，从而很可能导致一个臃肿而缺乏概念完整性的第二版系统。

先把这个没有解决的问题搁在一边——关于我对数学式语言和非数学式语言划分方法的早期评论——另外，还有数学家设计的语言，或者前数学家或者具有数学家思维的人设计的语言，以及其他人设计的语言。我希望前者几乎可以完全列出来，即使不是很正式。你应该真实地写下来：在你能想到的所有情况下它应该做什么。实际上，你应该把词法的事情写下来，而不是在做提示。你应该尝试写下其余的部分，你可能不会成功。

Simon Peyton Jones 说他们设法完成了 Haskell 第 1 版 85%的详细定义，但是，剩下的部分不值得他们浪费时间了。

Peter：有时你可能太仔细了。其中之一就是，`short int` 或 `long` 没有它们在 C 中表现得那么好。你有两种选择。

你们还可以考虑有符号类型（signedness）。

Peter：就此而言，我们甚至不会使用有符号类型或者常量。你可能会说，“我们会使用 `int8`、`int16`、`int24`、`int36`、`int64`，等等这些都是，而且，语言要么给出确切的承诺，或者除了取整之外，我们会尽力而为”。

这些都是在回顾过去。我不能确认从头开始干是不是好一些。或者，你会说，“听着，我们有了 `short int` 和 `long`，而且，我们会告诉你它们是什么，随后我们就走开不管了。其他事情，你必须自己设法处理”。对于想向用户提供尽可能高的效率的编译器作者来说，这并不会让他们感到惊讶。你可以在 GCC 中看到，它们优雅地为你提供了大量的类型，比如 unsigned 和 pointer-sized 诸如此类的其他类型，甚至多到考虑不过来了。

字符串也是一个有趣的例子。在你已经走到那一步之后，你会发现，如果允许它们使用任意字符，而不是使用 UTF-8 字符串或者 ASCII 字符串的话，你就不能打印它们。虽然说“是的，我的语言和程序所支持的所有结构都是二进制的，除了那些明确定义为可打印格式的以外”，听起来并不是太糟糕，事实上却是讨厌到了极点。

我们需要一些真正的概念性突破。它需要很长时间，而且我对此比较悲观。

您已经让我对此感到悲观了。

Peter：对不起。令人惊异的是实际上有相当多的函数充斥了这些东西，而且你实际上可以信赖它。它并没有提供什么保证，但是你可以充分信赖它。你知道，你的车如果没有计算机的话，早就会失去所有的运行功能，而且在你的车上有许多代码。它们几乎随时都在工作。我知道这没有保证，而且我听说过很多这样的故事：当你行驶在高速公路上或者其他地方时，车载计算机需要自己重新启动，但是从根本上你还得依靠它。从某种程度上来说，我只是抱怨它效率低下，而不是抱怨它的基本缺陷——但它们会让人大伤脑筋。

要开始解决这些问题，我们需要哪些突破？

Peter：我认为不仅仅是“不!”，而应该是“当然不!”，不过让我做一些观察。一方面，在我们编写软件的机器上，我们拥有大量的高性能计算能力。大多数的计算能力仅仅是做一些无用循环。极大一部分计算能力用来运行用户接口。然后，例如你是在编译 C，它的很大一部分被用来把数据读入内存，然后写到不同版本的中间文件中，以便它们能够再次被读入内存。

对吧？对于一种结构完整性适度的语言，在此你可以真正讲述混淆现象和其他许多东西，你应该认为编译器可以把工作做得更好。这意味着程序员需要一些方式来解释他们意图，这些意图我们目前还没有领会。事实上，它变得越来越糟。把线程和面向对象描述成互不相关（正交）的，这应该是极为慷慨大度的。你更接近于把它们描述成错综复杂的东西。我们已经得到了所有这些东西。有些东西会让你很难说清除程序在做什么，你应该把这些东西都清理掉。这在多核世界中将变得离奇地困难。

或许很多聪明人在那儿做着非常有趣的工作。他们或许会开发出一些东西来。我们唯一的方法就是要求电脑帮助我们的程序做到更安全和更清晰，而且在我们设计它们时，使用的就是一种较好的语言。很难弄清楚是哪一种语言；它就像一个巨大的纱球（译注13），而且里面乱成一团，你很难理出什么头绪。要找纱线末端？我不知道末端在哪儿。我不会无限地乐观。

另一方面，它也并非一无是处，只不过是有点讨厌而已。

154

在您的工作方面，您是如何定义成功的？

Peter：当我们在编写 AWK 或者做诸如此类的事情时，似乎（其实可能不是这样）我们会有些想法并做适量的工作，而且如果你的想法不错，同时你的实现也不错，你就会对计算产生较大的影响。所以，设定一个标准，再让自己来满足这个标准的要求，我认为是很难的。以前在一些重要的计算领域有所作为是比较容易的，但是我想现在已经没那么容易了。

我认为，产生较大影响的小组相对来说是比较少的。要产生很大的影响很难。成功几乎是不可能的，很多人在使用你的语言，人们都认为不错，但是我认为你不可能产生同样的影响。这个很难理解：Unix 就是一个极端的例子，一个相对较小的核心小组开发了这个系统，并且给计算机界带来了很大的影响。那么，或许你的哪位读者会解释说有很多例子，我在这 5 年或 10 年内错过了很多。但是我不这么认为。我认为，现在是小组规模更大，而且成功会更加困难。

对于你的问题，我的回答是：我们是幸运的，虽然工作干得不多，但影响却比较大。对于成功来说，这是一个多么伟大的时刻，一个多么好的基准！现在，我认为那些比我们聪明的人，他们是很难获得那样的成功的。当然，它在很多方面都是很棒的。这意味着一种巨大的社会进步，同时也意味着个人要懂得知足。

6.13 通过实例来编程

Programming by Example

你说过，AWK 这种编程语言之所以能生存下来，是因为它是通过实例来编程。

Peter：它是一种深思熟虑的设计决策。它包括了很多内容，其中既有一些好东西，也有一些不太理想。AWK 有一个有趣的集合（这是一种很礼貌的表达方式）：句法选择，我认为其中只有个别真正的错误。它的大部分思想都类似于 C 语言，因此，不需要我们向合作者专门解释。

那么，现在的问题是什么？我们的观点是：既然所有的 AWK 程序都只有一行或者最多几行，那么，编写 AWK 程序的方式就是去寻找一些例子，它们的功能与你要做的事情类似。如果你想要更复杂一点的程序，你就必须得加点东西进去，随后就可以一切搞定了。我们在编写 AWK 的时候，同时 Xerox PARC 也有一个项目，不幸的是，我已经把它的名字忘掉了，它的功能也大致与 AWK 一样。它是设计用于处理文件系统的。Xerox PARC 系统不允许你的文件有什么痕迹，它是不透明的。它打算由秘书来使用。供你写程序的页面有两栏。你在左边那一栏编写程序，右边那一栏是运行过的实例。编译器会检查你的程序的运行结果，看它是否完成了相应实例的对应功能。

译注13：ball of yarn，原意为纱球。在英文中，可以引申为看起来乱七八糟，而里面隐藏着秘密的东西。

这很聪明。

155

Peter：它是聪明的。此外，他们力争让语法对秘书来说是友好的。当然，它失败了。它没有获得广泛的成功，而 AWK 则获得了成功，这有很多原因。Unix 推广开来，而其他编写出来的系统则没有推广开，等等。我们的那个版本是这种想法：你找到了一些程序，它们的功能与你想要的功能大致相同。AWK 始终是专门为程序员而准备的。

当然，这不是它的发展方式，但是，它相对简单，并且还有例子供你参考，同时，你还可以看 AWK 的书并查看里面的例子，我认为这些事实很有帮助。

在复制、粘贴和修改一个程序时，你充其量是通过潜移默化学习了语言的语义。

Peter：Except AWK 书的思想是在非正式的介绍和全部是例子之间，它们被期望对于“语言是什么和语言做什么”，有一个很完整的描述。我认为这是它的发展方式。

人们会阅读它吗？

Peter：有些人会读，而有些人则不会。让我使它略有不同，好吗？不管你是否读它，你都可以根据 AWK 中的例子来做，对不对？这是一个经验事实。Ada 语言怎么样呢？

我从来没试过。

Peter：我想仅仅靠例子，你很难编写 Ada 程序。想通过例子学会 C++程序可能也是很难的。对于某些简单的内容，你可以通过某些方式来学习它。

人们也不会运行只有一行的 C 程序。

Peter：当然，这是不同的。除了与屏幕有关的问题以外，很难编写出只有一行的程序。虽然那些例子很小，但本质上却是个大问题。

这是您尝试解决的问题的范围。

Peter：是的。它们是通用的，而 AWK 却不是。早期用 AWK 编写的一个程序是某些附属处理器的汇编程序。我对此感到很震惊。它没有解释清楚为什么要这么做，但是有一点很明确：用解释型语言来做这件事就容易多了，而 shell 功能还不够强大。

我很希望看到，编程对人们来说都很容易，但是我更希望程序变得更加可靠，更容易组装成更大的元程序。这些问题很难解决。

Peter：某些可组合性可以借助于语言设计和风格来实现，而可靠性却有些困难。简约设计也是很困难的。

您是如何认识到简约设计的？

156

Peter：这是后设话题（metaissue）。我曾经对我的判断比现在更有信心。你看看它，并尝试着编写一些小例子。你会考虑人们对它怎么看。人们在课本中使用那些简单明了的例子完全是不现实的，这一点永远不会变。那里可能会有一个类，看起来好像是面向对象编程中介绍的那个类，但我不相信它。他们常常有很多很多的

成员，这些成员能够响应很多消息。你可以往程序里加入很多的有用对象、它们还可以理解你的程序是何状态。我认为，如果回报足以令你满意的话，你可以接受一定的复杂性。

或者是可以感受到的回报。

Peter：没错。这就是你的全部所得。这就是软件工程。那里没有定量的东西。你所感受到的回报和实际存在的回报是一样的，因为我们无法衡量实际的回报，或者至少还没有出现能够衡量实际回报的可能。

我们甚至无法衡量生产率，这样一来，衡量更好或更坏变得很难。

Peter：确实如此，不过我认为这并不是问题的关键所在。假如你干的是一个真正的工程，你就会衡量它的效果。比如说，你在建桥。这座桥要花多少钱？建造它有多困难？它能不能建立起来？

对于程序来说，你可以衡量构建它有多么困难。要花多少钱来构建它，但是，其他的一切都完全是个谜。它能像想象的那样好吗？我们怎么能知道？你如何描述对它的期望？

对于软件来说，它还不是一种实质性的科学。

Peter：我有这样的期望：当做硬件的人们最终筋疲力尽时，软件就不仅仅只是一种工程原理了。一切都在快速地变化。这是一个含糊的类比，不过，如果水泥或者钢材的性能每年以10%的速率变化的话，结构工程看起来就会大不相同。

这是一种推测，因为它没有任何理由看起来不同。只有模型才能说是2007、2008或者2009。既然我们在软件方面从来没有任何模型，我想我们无法那样做。

我们讨论的是软件。我们既没有原子，也没有物理属性。

Peter：是的。它并不完全像数学。它并非完全存在于人们的大脑中。它几乎完全是靠人力构造的，而且，它的约束都是一些数学组合，而这些组合都与可计算性和算法复杂性有关，而无论硬件给我们提供的是什么。这部分变化特别快。

157

您提到的另一点是计算机程序员和理论之间的真正区别。您可以证明一条定理，随后您就了解了一些知识，而通过编写计算机程序，您突然就可以做到以前做不到的事情。

Peter：我仍然认为这是正确的。当然，更新一些的理论版本有时会伴随着算法出现，不用惊讶，因为计算变得如此有用，以至于这条边界已经变得有些模糊。

在计算机时代之前，人们在常常思考计算会有多难。他们要计算一些东西。科技文献有大量这样的例子：你查看一下某些人的笔记本，他们手工进行这种惊人的计算，并且算出了答案。我认为这种区别并不是完全绝对的。从我开始成为一名数学家时，就一直这么认为。

我们开始把组件当成理论来思考时，会出现计算革命吗？

Peter：不会出现，直到我们学会如何去描述它们为止。已经广泛使用的那种描述方式纯粹是功能性的：也就是描述输入如何转换成输出。它根本没有或者很少论及需要花多长时间，它也几乎没有说明大概需要多少内存。至于它要在什么环境下运行，这一点也有些模糊。公平地说，那些理论要比机器理论更人性化，但是，它们总是伴随着假设和结论。它只能适度解释，但是使用软件，则需要你解释更多。

还有很多令人讨厌的例子。我给出的那份名单甚至遗漏了一些东西，比如需要真正详细的输入规范。当从 16 位机移到 32 位机上时，有些程序就会失败，因为它们中间有些地方本质上是 16 位的，而你对此并不知晓。这些东西很难说全。这并不公平，不过，令人喜欢的例子是那些虽有 bug 却仍被证明正确的程序。不幸的是，它有一些东西模拟的并不是现实情况，而人们却从未注意到这一点。

类型也有这个问题。他们想要实现一个“类型归纳保证”，它会使系统的强度极弱。在这个多维空间中，另一个极端是 C++模板，它可以在编译时间内以你可以预期的速度计算任何东西，不过，既然你什么都可以计算，不管怎样，它都会变慢。

对于“究竟是什么在困扰着计算机科学”这个问题，这就是一个更有趣的回答。计算机并没有变得越来越快，而是它的应用范围变得越来越宽了。

Peter：利好（goodness，优点）的指数增长是一个很好的生存环境。

我最近认识到，自 20 世纪 70 年代以来，数据量的增长使得处理器速度的增长相形见绌。以当时设计的 SQL 为例，在这个数据规模大爆炸的时代，它仍然可以应付自如。其他语言的情况就没那么好了。

Peter：我要说的是，从那以后，CPU 的速度已经增长了上千倍。这不是数据量增长的数字。以我在计算方面的经验，我认为，随着时间的推移，CPU 的速度提高了 10^n 倍。硬件速度提高了大约 $10^{n/2}$ 倍，而算法速度也提高了 $10^{n/2}$ 倍。我认为这些数字是对的。

另外，在查询优化器以及数据库设计方面，也做了很多工作，使得我们可以使用 TB 级的数据库。

硬件资源会对软件程序员的心态有什么影响呢？

Peter：程序员千差万别，不能一概而论。你必须要注意约束条件。例如，光速是固定的，它就无法提高。很多东西在本地工作良好，而对于远程使用来说则是一场灾难。所有的那些抽象层和有用的库几乎都会允许我们让程序快速运行，不过，它们对速度和健壮性会有负面影响。

你是首先确定正确的算法，然后让它们运行得更快，还是从一开始就聚焦于速度？

Peter：如果你已经对某个问题有了一些理解，你就会以保证所需之处性能良好的方式来构建算法，还可以根据需要来修改这个算法。通常来说，使算法能运行更加重要，而且在多次修改之后，你很难重新整理实现。不过，以下情况并不鲜见：你会发现一些东西比预期更大，或者比预期使用更加频繁，如此一来，连平方算法都无法忍受，甚至是把太多时间浪费在了复制和排序上面。很多程序不需要很多工作就能充分地运行。现代计算机速度非常快，而且经常可以感觉到网络或者 I/O 的延迟。

您是如何搜索软件中的问题的？

Peter：诀窍在于找到你能解决的问题。软件里面通常有许多问题很难解决，但是也有许多问题并不难，它们的数量是那些难题的上百倍乃至上千倍，而且，老一套解决方案通常效果没那么好。

一旦你已经使某些东西能基本运行了，你会做什么？

Peter：如果它是供我自己使用，我就会到此为止，希望我会记得足够的上下文来升级它。如果是专业软件，应该真正地为它写个文档，并设法强化它以应对不同的恶劣条件，同时，为代码添加足够的注释以便别人容

易维护（或者像我那样容易维护）。最后这一件事很难。很少见到注释很好的代码。

159

有一种 Unix 设计哲学说，“如果你不知道怎么才能做好，那就干脆别做”。这种艺术化方法的传播超出 Unix 范围了吗？

Peter：这是个更大的问题，而不仅仅是软件的问题。我们生命中的大部分时间，都需要做一些并不知道如何才能做好的事情。我们参与了 Unix，而且 Unix 最初的开发人员把很多事情都规划好了，这真是一种极大的幸运。

探讨一下这种精明而又实际的假设可能会很有趣：如果企业没有采用这种方法，那就是他们也没干什么。不过，我怀疑数据会更有说服力，无论是在这方面还是那个方面。微软（Microsoft）和苹果（Apple）之间存在着自然的（而且毫无疑问是很肤浅的）比较，但你怎么衡量它们呢，是来自评论界的赞誉，还是累积的利润？

第 7 章 161

Lua

Lua 是一种很小的自包含动态语言，由 Roberto Ierusalimschy、Luiz Henrique de Figueiredo 和 Waldemar Celes 在 1993 年创建。Lua 具有一小组强大的特性和易用的 C API，这使得该语言易于嵌入并扩展，用以表示特定领域的概念。Lua 在专有软件界声名显赫，例如，Blizzard（暴雪）公司的《魔兽世界》和 Crytek GmbH 公司的《孤岛危机》等游戏以及 Adobe 的 Photoshop Lightroom 等，都使用它来作为脚本和 UI。它的前身是 Lisp 和 Scheme，也许还有 AWK；它的设计类似于 JavaScript、Icon 和 Tcl。

7.1 脚本的功能

您是如何定义 Lua 的？

Luiz Henrique de Figueiredo：一种可嵌入、速度快、功能强的轻量级脚本语言。

Roberto Ierusalimschy：不幸的是，越来越多的人们使用“脚本语言”作为“动态语言”的代名词。现在，甚至是 Erlang 或者 Scheme 都被称为脚本语言。这一点非常糟糕，因为我们失去了描述一类特定动态语言的精确性。在最初的含义解释中，Lua 是一种脚本语言，这种语言通常控制用其他语言编写的其他组件。

人们在使用 Lua 设计软件时，应该记住什么呢？

Luiz：答案很可能是 Lue 的工作方式。不推荐尝试效仿来自其他语言的所有实践。你必须真正地使用你的语言特性，尽管我想这对任何语言来说都是对的。就 Lua 来讲，在大多数情况下，这些特性堪称所有事情的法典和优雅解决方案的元方法。另外还有协程。（译注1）

Lua 的用户应该是哪些人呢？

Roberto：我认为大多数没有脚本功能的应用程序的编写者都能从 Lua 中受益。

Luiz：问题在于，大多数设计者直到很久以后都没有意识到这种需求，当时已经有了很多用 C 或 C++这样的语言编写的代码，而且他们也觉得，现在用 Lua 为时太晚了。应用程序设计者应该从一开始就考虑脚本。这会给它们带来更多的灵活性。通过迫使这些应用程序设计者思考应用程序的哪些地方需要保留原有性能，哪些地方根本就无关紧要，因此，可以降低性能要求，从而缩短脚本的开发周期，这样一来，也为考虑性能时提供了一个更好的视角。

从安全性的观点来看，Lua 为程序员提供了什么呢？

Roberto：Lua 解释器的核被构建为一个“独立的应用程序”。这个术语来自 ISO C，大意是程序不使用来自环境（没有 `stdio`、`malloc` 等）的任何东西。所有那些功能都由外部库来提供。使用这种体系结构，只要有限地访问外部资源，就能够非常容易地创建程序。例如，我们能够在 Lua 自身创建沙盒（译注2），只要从它的环境中去除那些我们认为危险的内容（例如 `fileopen`）即可。

Luiz：Lua 还提供了用户自定义的调试钩子，用它可以监视 Lua 程序的执行，因此，例如，如果它耗时过长或者过度占用内存，可以将它异常中止。

Lua 的限制是什么？

Roberto：我认为，Lua 的主要限制也就是我所认为的任何动态语言的限制。首先，即使是使用最先进的 JIT（Just-In-Time Compiler，即时编译器）技术（而且，Lua 拥有动态语言中最好的 JIT 之一），你无法获得优秀的静态语言的性能。其次，若干复杂的程序真正地可以受益于静态分析（主要是静态类型）。

译注1：协程（coroutine）技术是一种程序控制机制，早在上世纪 60 年代就已提出，用它可以很方便地实现协作式多任务。现在，很多动态脚本语言（Python、Perl）都提供了协程或与之相似的机制，其中最突出的便是 Lua。

译注2：沙盒（sandbox）是软件开发过程中的一种虚拟测试环境，它用于将未经测试的代码变化与实际的产品环境隔离开来。

您为什么决定使用垃圾收集器？

163

Roberto：Lua 从第一天开始，就一直使用垃圾收集器。我想说，对于一种解释型语言来讲，垃圾收集器可以比引用计数更加紧凑和健壮，更不用说它没有把垃圾丢得到处都是。考虑到解释型语言通常已经有自描述数据（通过标签值之类的东西）了，一个简单的标记并清除（mark-and-sweep）收集器可能确实会很简单，而且几乎对其余的解释器不会产生什么影响。

而且，对于无类型语言(untyped language)，引用计数可能会非常多。没有静态类型，每个单一赋值都可能会改变计数，因此，需要对变量的新值和旧值进行动态检查。后来的经验表明，在 Lua 中引过引用计数根本没有改进性能。

您对 Lua 处理数字的方式满意吗？

Roberto：从我的经验来看，计算机中的数字老是会给我们带来一些惊喜（因为它们也在计算机外部!）。至于说 Lua 使用 double 作为唯一的数字类型，我认为这是一种合理的折衷。我们已经考虑了很多其他可选方案，不过对于 Lua 来说，大多数方案不仅极慢，而且还很复杂，或者太耗内存。对于嵌入式系统，甚至是使用 double 也不是一种合理的选择，因此，我们可以使用一个备选的数值类型（比如 long）来编译解释器。

您在 Lua 中为什么选择表作为统一数据构造器？

Roberto：从我的角度，灵感来自于 VDM（虚拟驱动器，一个主要用于软件规范的形式方法），当我们开始创建 Lua 时，有一些东西吸引了我的兴趣。VDM 提供三种集合形式：set、sequence 和 map。不过，set 和 sequence 都很容易用图来解释，因此我有了用 map 作为统一结构的想法。Luiz 也有他自己的原因。

Luiz：没错，我非常喜欢 AWK，特别是它的联合数组。

程序员从 Lua 的第一类函数（first-class function）中学到了什么？

Roberto：50 多年来，虽然名称各异：从子程序一直到方法，“函数”已经成为编程语言的主要部分，因此，对函数的良好支持是所有语言都有的一种资源。Lua 支持程序员使用函数式编程领域中的一些功能强大的技术，比如，把数据表示成函数。例如，一种形状可能用函数来表示，给定 x 和 y，可以判断这个点是否在这个形状内。这种表示需要一些简单的操作，比如联合和交集等。

Lua 还会以某种非常规方式来使用函数，而且事实上，它们作为最棒的方式，极大地简化了使用。例如，每一块代码（我们提供给解释器的所有代码段）都可像函数体那样来编译，因此，在 Lua 中，所有的常规函数定义都内嵌在一个外部函数中。这意味着，甚至是毫不起眼的 Lua 程序也需要第一类函数。

你为什么要实现闭包？

164

Roberto：我们一直想在 Lua 中实现闭包这种结构：它不仅简单、通用，而且功能很强大。自第 1 版以来，Lua 中的函数就是一类值（first-class value），而且实践证明，它们确实很有用，甚至对于“常规的”没有函数式编程经验的程序员来说，也是如此。不过，如果没有闭包的话，第一类函数的使用就会受到一定的限制。顺便说一句，“闭包”这个术语指的是一种实现技术，而不是指特性自身，这是“带有词法作用域的第一类函数”，不过闭包确实更短。

您打算如何处理并发问题？

Roberto：我们不赞成多线程技术，也就是抢占式共享内存。在 HOPL 论文中（注 1），我们写道："我们仍然认为没有人能够使用这样的语言编写出正确的程序：在这种语言中，a=a+1 是不确定的"。我们可以通过去除"抢占或共享内存"来避免这个问题，而且，Lua 对这两种方式都支持。

通过协程，我们可以不使用抢占方式来共享内存，不过，这对多核机器来说并没有用处。不过，多个"进程"能够非常有效地考察那些机器。我这里所说的"进程"，是指一个它自己的 Lua 声明的 C 线程，因此，在 Lua 层面没有内存共享。我在《Programming in Lua（Lua 语言编程）》[Lua.org] 第 2 版中，已经给出了这种实现的一个原型，而且，最近我们也已经看到有些库支持这种方式（例如，Lua Lanes 和 luaproc）。

你并不支持并发，但你却为多任务实现了一个有趣的解决方案——也就是非对称式协程。它们是如何工作的呢？

Roberto：在 Modula 2 方面，我有一些经验（我的妻子在她的硕士论文工作中为 M-code 编写了一个完整的解释器），而且我一直喜欢这种观点：把协程作为协作并发和其他控制结构的基础。不过，对称式协程，比如通过 Modula 2 提供的那些协程，将无法在 Lua 上运行。

Luiz：我们在 HOPL 论文中，对那些设计决策全部做了极为详细的解释说明。

Roberto：我们最终选择了非对称式模型。它的基本思想非常简单。通过显式调用 coroutine.create 函数来创建一个协程，把一个函数作为协程主体来执行。当我们恢复协程时，它开始运行函数体并且直到结束或者中断（yield）；一个协程只有通过显式调用 yield 函数才会中断。以后，我们可以再次恢复它，而且它会从它停止的地方继续执行。

它的基本思想非常类似于 Python 的生成器，但有一个关键区别：Lua 协程可以在嵌套调用中中断，而在 Python 中，生成器只能从它的主函数中中断。考虑到实现，它意味着一个协程必须有一个独立的堆栈，就像一个线程一样。令人不可思议的是，较之于"平的（flat）"生成器，这些"栈满的（stackful）"协程是多么强大。例如，我们可以在它们的基础上实现一次性延续（one-shot continuation）。（译注3）

7.2 经验

在您的工作方面，您如何定义成功？

Luiz：一种语言的成功，取决于使用该语言的程序员数量以及使用它的应用程序的成功。其实，到底有多少人在使用 Lua 编程，我们并没有确切的答案，不过毫无疑问的是，有很多成功使用 Lua 的应用程序，其中包括一些非常成功的游戏。同样地，使用 Lua 的应用程序的范围，从桌面图像处理到机器人嵌入式控制，表明 Lua 具有一个非常明确的小众市场。最后，Lua 是唯一一种由发展中国家创建并在全球获得广泛应用的语言。它也是 ACM HOPL 曾经重点推介的唯一语言。

注 1：R.Ierusalimschy、L.H.de Figueiredo 和 W.Celes 主编，《The evolution of Lua（Lua 演进之路）》，ACM HOPL（美国计算机学会编程语言历史）论文集 III（2007）。

译注3：延续（continuation）是一种函数式编程技术，它能记录线程当前的运行状态，并允许在随后中断和恢复。

Roberto：这很难定义。我曾经在多个领域工作过，我觉得各个领域的成功定义都各不相同。总之，我想说这些定义大多都有一个共同之处："非常著名"。被介绍给某些人，或者去找某些人并得到认可，这些都让人非常开心。

对于这种语言，你还有什么遗憾吗？

Luiz：我没有任何遗憾。不过，事后回想起来，如果像现在一样知道如何去做那些事的话，我们本可以早点儿开始！

Roberto：我不知道我有什么具体的遗憾，不过语言设计会牵涉到很多困难的决策。对我来说，最困难的决策是在易用性方面。Lua 的目标之一是让非专业程序员易于使用。我没有契合这种定位。因此，从我作为一名用户的观点来看，有关 Lua 语言的某些决策并不理想。Lua 的语法就是一个典型的例子：Lua 的很多应用都得益于其详细的语法，不过，就我自己的感觉而言，我更愿意使用更加紧凑的符号。

你在设计或实现时犯过错吗？

Luiz：我认为我们在设计或实现 Lua 时，并没有犯什么大错。我们只是学着如何发展一种语言，远非只是定义它的语法和语义并将其实现。这里也有很多重要的社会问题，比如说通过手册、书籍、网站、邮件列表以及聊天室等方式，创建并支持一个群体。毫无疑问，我们认识到了支持一个群体的价值，而且还学会了为此必须努力工作，设计和编码也需要这样的努力工作。

Roberto：我们很幸运，没有犯什么大错，不过我认为，我们在这个过程中还是犯了很多小错。但是我们有机会在 Lua 的发展中改正它们。当然，这会让一些用户感到烦恼，因为各个版本并不兼容，好在 Lua 现在已经非常稳定。

对于成为一名优秀的程序员，您有什么建议？

Luiz：永远不要害怕重新开始，这当然是说到容易做到难。永远不要低估需要注意的细节。不要增加那些你认为未来才会有用的功能：如果现在增加，就可能会妨碍你以后在真正需要的时候添加更好的特性。最后，让解决方案更简单应该是永恒的追求目标。正如爱因斯坦所言，是尽可能简单，而不是更简单。

Roberto：学习新的编程语言，但只能从好书那里学习！Haskell 是所有程序员都应该学会的一种语言。学习计算机科学：新算法、新形式体系（如果你不知道 lambda 演算的话，应该知道 pi 演算、CSP 等）。设法持续改进你自己的代码。

计算机科学的最大问题是什么？我们又如何教授呢？

Roberto：我想还没有什么能像"计算机科学"那样，它的知识语料库是如此的广为人知。这并不是说计算机科学不是科学，而是说"什么是计算机科学和什么不是（以及什么重要和什么不重要）"仍然是太难界定。计算机科学界的很多人都没有一个正规的计算机科学背景。

Luiz：我把自己当成是一名对"计算机在数学中起何作用"感兴趣的数学家，不过，我当然非常喜欢计算机。☺

Roberto：即使是那些有正规计算机科学背景的人，也没有达成共识，我们缺乏一个共同的立场。很多人认为是 Java 创建了监视器、虚拟机以及接口（而不是类）等。

许多计算机科学项目是光彩夺目的职业培训项目吗？

Roberto：是的。而且，很多程序员甚至没有计算机学位。

Luiz：我并不这么认为，但我不是作为一名程序员被雇用的。从另外一方面来说，我认为，要求程序员具有计算机科学学位证书或者诸如此类的东西应该是错误的。计算机科学学位并不能代表很好的编程能力，而且，很多优秀的程序员并没有计算机科学学位（在我开始编程时或许这是对的；现在我可能是太老了）。我的观点是，一个人拥有计算机科学学位，并不能保证他能很好地编程。

Roberto：要求大多数专业人士都拥有学位，这是不对的，但我的意思是这个领域的“文化”太弱。这个领域很少有什么东西需要人们必须知道。当然，雇主可能会有自己的要求，不过，不应该对学位作出规定。

数学在计算机科学，特别是编程方面，起什么作用？

Luiz：噢，我是一位数学家。我明白数学无处不在。我之所以被编程所吸引，很可能是因为它绝对具有数学的品质：精确、抽象和优雅。程序是复杂理论的证明，你可以持续不断地精炼和改进，而且它确实会有所作为！

当然，我在编程时根本没想这些，不过我认为，一般来说学习数学对于编程是非常重要的。它有助于带你进入一种特定的心境当中。如果你习惯于考虑具有它们自己规则的抽象事情，就更易于编程。

Roberto：按照 Christos H. Papadimitriou 的说法，“计算机科学是新数学”。一名程序员如果没有数学功底，就很难有大的作为。从更广的观点来看，数学和编程都具有一些共同的思想原则：抽象。它们还共享一个关键工具：形式逻辑。优秀的程序员始终会使用“数学”，利用它来创建代码不变量、接口模型等。

很多编程语言都是数学家创建的——或许这就是编程困难的原因所在！

Roberto：我会把这个问题留给我们的数学家☺

Luiz：好的，此前我已经说过，编程绝对具有数学品质：精确、抽象、优雅。对我来说，设计编程语言就像是构建一种数学理论：你提供了功能强大的工具，其他人可以使用它来做很出色的工作。我一直被那些规模小而功能强的编程语言所吸引。强大的原语和结构之美就像强大的定义和基本理论之美一样。

您是如何识别出优秀的程序员的呢？

Luiz：你也知道。现在，我觉得糟糕的程序员更容易识别——不是因为他们的程序很糟糕（尽管那些程序通常非常复杂又混乱不堪），而是因为你可以感觉到，编程对他们来说并不愉悦，好像他们的程序对自己来说就是一种负担，犹如谜团一般。

应该如何教授调试呢？

Luiz：我认为调试无法教授，至少不能正式地教授，不过当你跟别人（或许是比你经验更丰富的人）一起调试的时候，你可以通过具体调试来学习它。你可以从他们那里学习调试策略：如何去缩小问题范围，如何去做出预测和评估结果，判断什么没有用只是添乱等。

Roberto：调试是为了解决问题。它是一个活动，在这个活动中，你可能必须使用你曾经学过的所有智能工具。当然，也有一些有用的技巧（例如，如果可能的话不要用调试器，如果用 C 这样的底层语言编程不要使用内存检查工具），不过，这些技巧只是调试的一小部分。你在学习编程的时候，应该学习调试。

你如何测试和调试你的代码呢？

Luiz：我主要是一点一点地构建和测试。我很少使用调试器。我要使用调试器，一般是调试 C 代码，我从不用它来调试 Lua 代码。对于 Lua 来说，有几条得体的打印语句通常就可以胜任了。

Roberto：我差不多也是这样。当我使用调试器时，通常只是用来查找代码在哪里崩溃了。对于 C 代码，像 Valgrind 或者 Purify 这样的工具就足够了。

168

源代码中的注释有什么用？

Roberto：用处不大。我通常认为，如果有什么需要注释的，那就是程序写得不好。对于我来说，一个注释几乎是像“以后我应该设法重新编写这段代码”这样一个便条。我认为清晰的代码要比注释的代码可读性更强。

Luiz：我同意。我一直坚持：注释应该用来表达代码没有清晰表达的那些东西。

一个项目应该如何文档化呢？

Roberto：强制执行。没有什么工具可以代替一个井井有条、深思熟虑的文档。

Luiz：但是，就一个项目的发展写出好文档，唯一的可能就是从一开始就在头脑中计划好。Lua 并不是这样；我们从来没想到 Lua 能发展这么快，并在今天获得这么广泛的应用。我们在撰写 HOPL 论文（这花了将近两年时间！）时，发现已经很难记起当时是怎么做出一些设计决策的了。从另外一个角度来说，如果早期我们有正式的会议记录，可能就会失去一些自发性，并错失很多乐趣。

在代码库的发展过程中，您会考虑什么因素？

Luiz：我会说“实现的简单性”。这样做，随之而来的是实现的速度和正确性。同时，灵活性也是重要的一点，以至于如果需要的话，你可以修改实现。

可用的硬件资源会如何影响程序员的心态呢？

Luiz：我是个老家伙了。☺我是在一台 IBM 370 上学习的编程。从卡片穿孔、提交给队列再到打印输出，通常要花好几个小时。我见过各种各样的慢机器。我认为程序员应该体验一下这些机器，因为并不是世界上人人都有最快的机器。编写应用程序是给大众用的，因此，程序员应该在慢机子上运行它，获得更广泛的用户体验。当然，也可以使用最好的机器来开发：花很长的时间来等待完成编译，这并不是什么很愉快的事。在现在的全球因特网中，Web 开发者应该尝试慢速连接，而不是它们用的超快连接。以平均水平的平台为目标，会让你的产品速度更快、更简单，而且质量更好。

就 Lua 来说，“硬件”是指 C 编译器。我们在实现 Lua 的过程中学会的一条经验就是：以可移植性为目标确实值得。几乎从一开始，我们就是用非常严格的 ANSI/ISO C（C89）来实现 Lua 的。这样一来，Lua 就可以在专用硬件上运行，比如机器人、打印机固件和网络路由器等，这些没有一个是我们当初的实际目标。

Roberto：你应该始终认为硬件资源是有限的，这是一条金科玉律。当然，它们总是有限的。“自然界里是没有真空的”；任何程序都有扩展的趋势，直到它用完了所有的可用资源。此外，随着那种资源在固有平台上变得更便宜，又会出现带有严重限制的新平台。微型计算机是这样；移动电话是这样；一切都是这样。如果你想做成市场第一，你最好要时刻关注你的程序需要什么资源。

169

就创新、进一步开发和采用您的语言方面，您对于现在或者不久的将来开发计算机系统的人，有什么经验和教训要提醒他们吗？

Luiz：我认为，程序员应该始终记住：并非所有的应用程序都是运行在功能强大的台式机或者便携式电脑的。很多应用程序要运行在受限的设备上，比如说手机，甚至是更小的设备等。设计和实现软件工具的人们应该特别关注这个问题，因为没有人会告诉你，你的工具会在什么地方如何使用。因此，就应该为使用最小的资源而设计，而且，你可能会惊奇地发现：很多环境使用了你的工具，而你并没有把这些环境作为主要的应用目标，你甚至都不知道它们的存在。Lua 就碰到过这种事！而且这很自然；我们内部有一个笑话，这其实不是一个真正的笑话：我们讨论在 Lua 中加入一个特性时，我们会问自己，“好的，不过，它会不会在微波炉上运行呢？”

7.3 语言设计

Language Design

Lua 易于嵌入使用，而且要求的资源也非常少。您是如何设计的，使得它在硬件、内存和软件资源都很有限的情况下运行得很好？

Roberto：开始时，我们并没有把这些目标搞得很明确。我们只是为了提交项目才不得已而为之。随着我们的发展，这些目标对我们来说变得更为清晰。现在，我想各方面的主要问题都始终是经济问题。例如，无论什么时候，有人建议一些新的特性，第一个问题就是需要多大的成本。

您有没有因为特性成本太高而拒绝添加它们呢？

Roberto：相对于它们带给语言的东西来说，几乎所有的特性都是“成本太高”了。举一个例子，甚至一个简单的 continue 语句都不符合我们的标准。

添加一个特性需要带来多大的收益才是值得的呢？

Roberto：没有一定之规，不过优秀的规则是：该特性是否能让我们感到“惊喜”；也就是说，除了它的初始动机以外，它对于事情是有用的。它让我想起了另一条经验法则：多少用户会从该特性中受益。某些特性只对一小部分用户是有用的，而其他特性对于几乎所有人都是有用的。

添加的特性对于大多数人来说是有用的，您有这样的例子吗？

Roberto：比如说 for 循环。我们甚至反对过这个特性，不过当它出现时，它改变了书中所有的例子！弱表也是出奇地有用。使用它们的人并不多，不过他们应该用一用。

在 1.0 版本之后，你一直等了 7 年才添加了 for 循环。这种循环的加与不加之间，各有什么原因呢？

Roberto：我们没有添加它，是因为我们没有发现一种通用而且简单的 for 循环格式。当我们发现了一个好的格式，就使用生成器函数添加了循环。实际上，闭包是使生成器容易和更通用的一个重要组成部分，因为通过闭包，生成器函数自身能够在循环中保持内部状态。

通过更新代码来使用新特性和最新发现的最佳实践，这是另一个高成本领域吗？

Roberto：人们不必使用新的特性。

人们选择 Lua 的一个版本，并在整个项目生命周期中自始至终都使用这个版本，并且从不升级吗？

Roberto：我认为，在游戏领域大多数人确实是这样做的，不过在其他领域，我认为有一些项目推动了他们所用 Lua 版本的更新。作为一个反例，魔兽世界使得 Lua 从 5.0 更新成了 5.1！不过，请记住： Lua 现在要比以前更加稳定。

你们在开发过程中是如何分工的，特别是在编写代码方面？

Luiz：Lua 第 1 版是由 Waldemar 在 1993 年编码的。自 1995 年左右以来，Roberto 编写和维护了大部分代码。我负责一小部分代码：字节码 dump/undump 模块和独立编译器 luac。我们一直在修改代码，并通过电子邮件向其他人发送代码修改建议，而且，我们曾就新特性及其实现开了很长时间的会议。

您从用户那里得到了很多有关语言或者实现的反馈吗？对于在语言中加入用户反馈及其修改，您有一个正式的机制吗？

Roberto：我们开玩笑说：你要是忘了什么，那它肯定不重要。Lua 讨论列表非常活跃，不过一些人将开放软件和社区项目等同视之。有一次，我向 Lua 列表发送了以下消息，总结了我们的方法：

> Lua 是一款开放软件，不过它从未进行过开放式开发。这并不意味着我们没有听取其他人的意见。实际上，我们几乎阅读了邮件列表中的每一条消息。Lua 里面的若干重要特性就起源或发展自外部的贡献（例如，元表、协程以及闭包的实现，仅举几个重要的为例），不过，一切都要由我们来最终决定。我们没有实行开放式开发，是因为我们认为我们的判断要比其他人更好。这仅仅是因为我们想让 Lua 成为我们想要的语言，而不是世界上最流行的语言。
>
> 由于采用了这种开发风格，我们不愿意为 Lua 建一个公共知识库。我们不希望每一项代码修改都必须解释。我们不想老是更新文档。我们想要自由遵从奇怪的想法，如果放弃，也不必处处解释。

为什么您喜欢获得建议和想法，而不是代码？我在想，或许您自己写代码能够让您学到关于问题（解决方案）的更多知识。

Roberto：差不多可以这么说。我们喜欢彻底搞清楚在 Lua 中发生了什么，因此，一段代码贡献不大。一段代码并不能解释为什么采用这种方式，但是，一旦我们理解了它的基本思想，编写代码就成了我们不想错过的乐事。

Luiz：我认为我们还需关注包含第三方代码，关于这部分我们不能够保证所有权。我们肯定不想溺死在要别人把代码授权给我们的合法化的过程中。

Lua 会不会达到这种状态：你已经添加了所有想要添加的特性，唯一需要的就是改进实现（例如，LuaJIT）？

Roberto：我觉得现在就处于这种状态。我们已经添加的特性，如果不是全部的话，也是我们想要添加的绝大部分。

您是如何处理冒烟测试和回归测试的？使用开放知识库的一大好处是，你可以让人们对几乎每一个修改进行自动测试。（译注4）（译注5）

Luiz：Lua 版本更新并没有那么频繁，因此，发布一个版本时，已经进行过很多的测试。当这个版本已经相当可靠时我们才发布可运行的版本（pre-alpha 版），因此，人们能够看到有什么新特性。（译注6）

Roberto：我们确实进行了强大的回归测试。这么做的出发点是，因为我们的代码是用 ANSIC 编写的，基本上没有什么可移植性问题。我们没有必要在若干不同的机器上进行测试。一旦修改了代码，我就会执行所有的回归测试，不过这一切都是自动进行的；我必须做的就是完成所有的形式检验。

如果发现了一个反复出现的问题，到底是局部就事论事，还是全局通盘考虑，您如何判断哪一种是最佳解决方案？

Luiz：我们一直尽量做到一发现 bug 就修复它。不过，因为我们并不经常发布新的 Lua 版本，所以我们都是等到有足够的修复量才发布一个小版本。我们把所有的改进都留给大版本来解决，这些改进不是 bug 修复。如果问题比较复杂（并且经常出现），我们在小版本是局部就事论事权宜解决，而在大版本中则是全局通盘考虑修复。

Roberto：通常，局部的权宜修复很快就可以完成。只有在确实不可能进行全局修复时，我们才会进行局部的权宜修复：例如，如果某个全局修复需要一个新的不兼容接口时。

从开始到现在，已经过去了这么多年，您仍然会为有限的资源而设计吗？

Roberto：当然会的，我们一直致力于此。我们甚至考虑过 C 结构内的字段顺序，以节省几个字节。☺

Luiz：相比于以前，现在有更多的人们把 Lua 语言运用到更小的设备上面。

从用户的角度来看，对简单性的这种需求对语言设计有何影响？我想起了支持 Lua 的类，这使我想起了 C 语言中许多方面的面向对象特性（但没有那么让人讨厌）。

Roberto：目前，我们有一个准则叫“机制而不是策略”。它可以保证语言的简单性，不过就像你说的，用户必须提供它自己的策略。这是使用类的情况。它们有很多种实现方式。一些用户会喜欢这种方式，而其他用户则可能恰恰相反。

Luiz：这个确实给 Lua 提供了一种“自己动手的”的风格。

Tcl 也有一种类似的方法，不过它会导致很多存储碎片，因为每一个库或者工作室都有它自己的方法。根据 Lua 的预期目的，是否有存储碎片并不是一个太大的问题？

译注4：开发人员修复测试中发现的 Bug，如想知道这次修复是否成功，或者是否会对其它模块造成影响，就需要进行所谓的冒烟测试（smoke testing）。它设计用于确认代码中的更改会按预期运行，且不会破坏整个版本的稳定性。这一术语源自硬件行业。对一个硬件或硬件组件进行更改或修复后，直接给冒烟测试设备加电，如果开始通电的第一时间里没有冒烟，则该组件就通过了测试。

译注5：回归测试是指在发生修改之后重新测试先前的测试以保证修改的正确性。理论上，软件产生新版本，都要进行回归测试，验证以前发现和修复的错误是否在新软件版本上再次出现。

译注6：有时候软件会在 alpha 版本前先发布 pre-alpha 版本（准预览版本）。Alpha 版本仍然需要测试，其功能亦未完善，因为它是整个软件释出周期中的第一个阶段，所以它的名称是“Alpha”，希腊字母中的第一个字母。相对于 Alpha 版本，Pre-alpha 版本是一个功能不完整的版本。

Roberto：是的。有时它是一个问题，不过对于大部分用户来说（例如游戏），这并不是一个问题。Lua 主要用于嵌入到其他一些应用程序中，因此，为了统一编程风格，应用程序提供了一个坚固的框架。你可以使用 Lua/Lightroom、Lua/WoW 或者 Lua/Wireshark，每一个都有它自己的内部文化。

您认为 Lua 的“我们提供了机制”这种可扩展性风格，在当时是一种极大的好处吗？

Roberto：这么说并不确切。对于大多数其他事情来说，它是一种折衷处理。有时候，准备好了使用策略非常有用。“我们提供了机制”是非常灵活的，不过，它需要更多的工作，而且也存在着“存储碎片”风格的问题。它也是非常经济的。

Luiz：另一方面，有时候这很难向用户解释。我的意思是，让它们理解是什么机制以及这种机制的基本原理。

这种工作会影响在项目之间共享代码吗？

Roberto：没错，通常是这样的。它也阻碍了独立库的发展。例如，WoW 拥有大量的库（它们甚至还使用遗传编程实现了货郎担问题），不过在 WoW 之外却没人使用。

您担心 Lua 会因此分裂成 WoW/Lua，Lightroom/Lua 等语言分支吗？

Luiz：我们并不担心：语言还保持一样，只是可用的函数不同而已。我认为这些应用程序会在某些方面受益于这种不同。

认真的 Lua 用户会在 Lua 基础上编写他们自己的语言分支吗？

Roberto：很有可能。至少我们没有宏。我认为你可以使用宏来创建一种真正的新语言分支。

Luiz：是的，它本质上不是一种语言分支，而是使用函数实现的特定领域语言的一种分支。这就是 Lua 的目标。Lua 仅供数据文件使用，它看起来可以像是一种语言分支，不过，当然他们仅仅是 Lua 表（Lua table）而已。或多或少有一些项目在做宏。例如，我想起了 metalua。这是 Lisp 的一个问题。（译注7）

为什么您选择提供可扩充语义学？

Roberto：它开始是作为提供面向对象（OO）特性的一种方式。我们不想在 Lua 中添加面向对象机制，不过用户想要它们，因此，我们提出这种观点：提供足够的机制，让用户来实现他们自己的面向对象机制。我们仍然认为这是很好的决策。尽管使得初学者使用 Lua 进行面向对象编程更为困难，它也给语言带来了很多灵活性。特别是，我们将 Lua 与其他语言混合使用时（这是 Lua 的一个特色），这种灵活性允许程序员将 Lua 的对象模型和外部语言的对象模型相适应。

目前的硬件、软件、服务和网络环境同您最初设计时的系统环境有何不同？这些变化对您的系统有何影响？它们对进一步的修改有何要求？

Roberto：因为 Lua 是以极高的可移植性为目标，我认为目前的“环境”同以前的环境并没有什么不同。例如，我们开始开发 Lua 时，DOS/Windows 3 是 16 位机器；一些老机器仍然是 8 位。目前我们没有 16 位的台式机，不过，若干使用 Lua 的平台（嵌入式系统）仍然是 16 位或者甚至是 8 位的。

译注7：metalua 是一种扩展 Lua 语法的方式，比如想在循环里使用 continue，但 Lua 里没有 continue 关键字，而通过 metalua 的扩展就可以做到。

最大的变化是 C 语言。我们是在 1993 年开始编写 Lua 的，ISO（ANSI）C 还没有像今天这么成熟。很多平台仍然使用 K&R C，而且，很多应用程序还有一些通过 K&R C 和 ANSI C 编译的复杂宏模式，主要的区别在函数头的声明。当时，坚持使用 ANSI C 是一个冒险的决定。

Luiz：而且，我们仍未感觉到需要转移到 C99 上面。Lua 是用 C89 实现的。如果过渡到 64 位机器上时出现了故障，或许我们必须使用 C99 的一部分（特别是具有特定规模 size-specific 的新类型），不过我并不期望出现任何故障。

如果能全部重新构建 Lua 的 VM 的话，您仍然会坚持使用 ANSI C 吗，或者您希望有一个更好的语言用于跨平台的底层开发吗？

Roberto：没有。ANSI C 是我（目前）知道的可移植性最好的语言。

Luiz：有些杰出的 ANSI C 编译器，不过，即使是使用它们的扩展，也不会给我们带来很多改进的性能。

Roberto：改进 ANSI C 和保持它的可移植性和性能并不容易。

顺便问一句，这是 C89/90 吗？

Roberto：是的。C99 尚未构建好。

Luiz：再者，我不确定 C99 能给我们带来很多额外的特性。我还特别想到了 gcc 中使用的 goto 标签语句作为 switch 的一种替代方案（虚拟机执行的主要 switch）。

Roberto：在很多机器中，这样做可以改进性能。

Luiz：我们早期就对它进行了测试，并且别人最近也对它进行了测试，效果并不引人入胜。

Roberto：部分原因在于我们基于寄存器的体系结构。它支持使用较少的操作码，而每个操作码会承担更多的工作。这样就减少了调度程序的影响。

您为什么要构建一个基于寄存器的 VM 呢？

Roberto：为了避免所有的 getlocal/setlocal 指令。我们也想要发挥想象。我们认为，如果它运行得不好，至少我们还能写一些有关它的论文。而最后，它运行得非常好，而我们仅仅写了一篇论文。☺

在 VM 上运行对调试有没有帮助？

Roberto：它不是“帮助”；它改变了整个调试的概念。曾经有过调试编译型和解释型语言（例如，C 和 Java）经验的所有人，都知道它们相差甚远。好的 VM 会让语言变得更安全，在某种意义上，该错误就语言本身而言总是可以理解的，而不是就底层机器而言（例如分段错误）。

如果语言是平台无关的，这对调试有何影响？

Roberto：通常它有利于调试，因为一种语言越是和平台无关，它就越需要可靠的抽象描述和行为。

考虑到我们是人（而不是机器），而且我们知道编写软件时在所难免要出错，您是否曾经考虑过：为了有助于调试，您需要向语言添加哪种特性或者从中删除哪种特性？

Roberto：当然了。辅助调试的第一步就是良好的错误消息。

Luiz：从初期版本开始，Lua 中的错误消息就在一直改进。我们已经从可怕的“调用表达式而不是一个函数”的错误消息（这种错误消息一直用到 Lua 3.2）中变成了更好的错误消息，比如“试图调用全局‘f’（一个 nil 值）”。从 Lua 5.0 开始，我们使用字节码符号执行来尝试提供有用的错误消息。

Roberto：在语言自身的设计中，我们一直设法避免通过复杂的解释来构建。如果它很难理解，就会更难调试。

在一种语言的设计与使用这种语言编写的程序设计之间有什么联系？

Roberto：至少对我来说，设计一种语言时主要是从用户的角度出发，也就是说，考虑事项是关于用户应该如何使用语言的每一个特性及各种特性的组合。当然，程序员总会找到使用一种语言的新方式，而且，优秀的语言应该允许那些意想不到的使用，不过，语言的“正常”用法应该遵从设计者创建语言的初衷。

语言的实现会在多大程度上影响语言的设计？

Roberto：这是一条“双行线”。实现会对语言设计产生巨大的影响：我们不应该设计无法高效实现的东西。一些人忘记了它，不过在设计任何软件时，效率一直是一个（或者是惟一的）主要约束条件。不过，设计也可能会对实现产生较大的影响。乍一看，Lua 的几个特色之处都来自于它的实现（规模小、与 C 有很好的 API 以及可移植性等），而 Lua 的设计则在其中起到了关键作用，使得实现成为可能。

我读过您的一篇论文，《Lua uses a handwritten scanner and a handwritten recursive descent parser（Lua 使用一个手写扫描程序和一个手写的递归下降分析器）”》。你是如何开始考虑手工构建一个分析器的？是不是从一开始就很清楚，这样做要比 yacc 生成的分析器要好？

Roberto：Lua 第 1 版使用了 lex 和 yacc，不过，Lua 最初的主要目标之一是作为一种数据描述语言，而不像 XML。

Luiz：但是时间要更早一些。

Roberto：很快人们开始在数 MB 数据的文件中使用 Lua，而 lex 生成的扫描程序则很快地变成了一个瓶颈。通过手写编写一个优秀的扫描程序非常容易，并且，就是这么一改使 Lua 性能提高了大约 30%。

从 yacc 改成手工编写分析器，这个决定是很久以后才做出的，而且它并不容易。它起始于大多数 yacc/bison 实现使用的骨架代码问题。

当时，它们的可移植性很差（例如，有一些使用了非 ANSI C 头文件 `malloc.h`），而且，我们确实没有能很好地控制它们的整体质量（例如，它们如何处理堆栈溢出或者内存分配错误等），而且它们也不是可重入的（从分析过程中调用分析器的意义上来说）。同样地，如果你想要像 Lua 那样在运行时生成代码，自底向上分析器也不像自顶向下的那么好，因为它难以处理“继承属性”。在我们做了一些修改以后，我们看到那个手写的分析器要比 yacc 生成的分析器更快更小，不过这不是促成我们修改的主要原因。

Luiz：自顶向下分析器也会提供更好的错误消息。

Roberto：不过，对于任何语言来说，我从不推荐在没有成熟句法的情况下手工编写分析器。而且，可以肯定 LR（1）（或者 LALR 或者甚至 SRL）会比 LL（1）功能更加强大。甚至对于 Lua 这样的简单句法来说，我们也必须使用一些技巧来构建一个像样的分析器。例如，二进制表达式程序根本没有遵从最初的语法，而我们使用基于优先级的智能递归方法方式来取而代之。我在编译器课上一直向学生推荐 yacc。

176

您的教学生涯中有什么趣闻轶事吗？

Roberto：我开始教授编程时，供我们的学生使用的主要计算机设备是一台大型机主机。有一次，一个非常优秀的团队提交的一个程序作业甚至在编译阶段就失败了。我训了他们一顿，而且他们保证会使用若干测试用例认真测试那个程序，最后它运行得很好。当然，他们和我使用的是完全相同的环境：大型机主机。这个谜团一直到几周以后才真相大白，因为我了解到 Pascal 编译器已经升级过了。这次升级恰巧就发生在他们完成任务和我开始改正之间。他们的程序里有一个大型机主机，而老编译器却没有检测到（如果我没记错的话，是一个多余的分号）！

Haskell

Haskell 是一种纯函数式惰性语言，最初设计用作现代函数式语言的开放标准。Haskell Report（Haskell 报告）首次出现在 1990 年，并在 1998 年被采用为“标准”。不过，近几年这种语言发展很快，特别是它的类型系统方面更是如此，在这方面出现了很多新特性。Haskell 最近变得非常流行，它拥有大量的独立存在的库、很多实际的应用程序，并在实现方面有着显著的改进（最著名的就是杰出的 Glasgow Haskell 编译器[GHC]），而且还有一个蓬勃发展的用户社区的支持。Haskell 对于研究特定领域的语言、并发以及状态的严格控制特别感兴趣。它为解决问题所提供高级抽象是无可匹敌的——至少，一旦你理解了 Haskell 的软件设计方式是这样的。

编注 1：这个访谈主要是以与 Paul Hudak、John Hughes、Simon Peyton Jones 以及 Philip Wadler 的电子邮件交流为基础，其中还融入了对 Simon Peyton Jones 的电话访谈内容。

8.1 功能性团队

在团队中你们是如何开发一种语言的？

Simon Peyton Jones：幸运的是，我们有一个共同目标（这就是开发一种通用的惰性函数式编程语言），而且，还就技术议程达成了广泛共识。我们在关于 Haskell 历史的论文中（注 1）介绍了使用过的各种不同的方法（当面会议、电子邮件，还有一个 Editor and Syntax Tzar）。我们还不必受困于在让现有用户向下兼容这种附加的需要。而且，这里面也没有公司牵涉进来，因此，也不必应付（不兼容的）公司目标。

John Hughes：我们都有一个共同的理想。对函数式编程充满热情——当时，大家对这个领域都极为兴奋，而且也都想尽自己所能，让函数式编程的梦想变为现实。不仅如此，我们互相也非常尊重。我认为，对于处理那些不可避免的棘手决定来说，热情和尊重这二者都是至关重要的。

Paul Hudak：大家一开始就有共同的理想。没有这种理想，我想不会走得太远。最初的 Haskell 委员会有一个蔚为壮观的共同理想。此外，Haskell 委员会成员的精力极为充沛，甚至都充沛到了不可思议的地步。这些成员简直就是一群野兽。

你还需要谦逊。像人月神话（mythical man-month）一样，并不是人越多事就干得越快，因为，虽然大家有着共同的理想，但毕竟还是有差异的。我们有很多不同，但也足够谦逊并易于达成妥协。最后，你还需要有领导才能。很幸运，我们能够共享领导才能——我想这是非常罕见的。总有一个人在推动工作，我们都知道那个人是谁，而且也相信他能把工作干好。

你们是如何把各自的想法融为一体的？

Simon：我们通过电子邮件进行了很多辩论。撰写了技术性的论证意见来支持自己的观点，然后再相互传阅。我们也乐于妥协，因为获得一种语言最为重要。而且，还有一个原因，就是我们意识到妥协的另一面也是有效的论证。

John：有时，我们会使用两种交叠的方式，例如，Haskell 既支持方程式风格，也支持表达式风格。不过，大多数时候会就竞争的观点进行很长时间的技术讨论，反复推敲最终达成共识。我认为语义在这里扮演了一个重要角色——尽管我们从来没有为整个 Haskell 生成一个完全的形式语义，我们通常会将设计的各个组成部分形式化，而且“丑陋的”语义通常是我们反对任何一项提议的有力证据。密切关注形式语义有助于指导我们实现清晰的设计。

Paul：通过辩论——主要是技术层面的辩论，孰“对”孰“错”通常是显而易见的，而且在一个主观的/美学的/有时是非常个人化的层面，在这里没有对错。这些辩论看起来是没完没了（有一些在今天也仍然是争论得很厉害），但是不管怎么说，我们通过它推动了设计工作的发展。在看似无关紧要的问题上，我们往往会依靠我们的领导能力来做出最后的决定。例如，我们使用“syntax czar”，它会对语法细节做出最后的决定。

你们如何识别最好的想法，又如何“处理”那些并不喜欢的特性呢？

Paul：最好的想法非常明显——最糟的想法也是如此！更难的问题在于那些并没有明确的最佳解决方案的想法。

注 1：“通过类提供惰性：Haskell 的历史（Being Lazy with Class: the history of Haskell）”，第三届美国计算机学会编程语言历史会议（HOPL III）论文集，*http://research.microsoft.com/~simonpj/papers/history-of-haskell/ index.htm*。

Simon：对于不喜欢的特性，我们只是不赞成它们。如果有足够的人反对，那么这个想法就很难有什么进展。不过，事实上我并不记得有过这样的情况：某个人或者某个小团队强力推行某种想法，但最终还是投票失败了。或许，这就是对我们参与项目的共同技术背景的考量。

简而言之，在大部分情况下，意见不同并不是问题。问题在于很难找到志愿者来做细节上的事情。语言都会有许许多多的细节。在这样或那样的不明朗情况下会出现什么问题？库也有大量的细节。这些东西并不浪漫，不过它非常重要。

John：通过不经投票而口头表态的方式，你会发现最好的想法！类系统就是其中之一：当我们看到它时，大家都是赞叹不已。然而，这并不是说，在处理完默认实例和与模块交互的细节之后就无须大量的努力工作了。

至于说我们不喜欢的特性，实际上对它们的讨论几乎是最为深入的。用户始终在抱怨它们，而且每一次语言修订，无论 X 是什么东西，总是有人会说，“我们最后不能去掉 X 吗？”，而且这个特性最终会被再次讨论。这意味着，至少我们会非常清楚为什么会拥有并不喜欢的特性，而且还无法去除它们。

有些时候，我们一开始就走对了路子——例如，从未流行的“单一同态限定”从一开始到现在都仍然保持着相同的形式，道理非常简单：因为它解决了一个实际问题，而且没人能找到一个更好的方式。在其他的例子中，我们根据来自该领域的经验修改了决定。对于最初设计中有一些明显过于严格的处理，后经证实有碍于程序发展，我们对此均进行了修改——这在语义优雅性方面只需要花费很少的成本，我们很少会遇到这样的情况。在证明编程新手会对重载列表理解（译注1）感到很困惑之后，我们就把它删掉了。我们已经能够根据经验回溯并修复错误，甚至对语言做出不兼容的修改，这样就最终改进了语言的设计。

团队开发有什么优势吗？这会不会迫使您做出妥协？

Simon：形成一个团队最为重要!我们每个人都有自己的语言，而且有一个共同的语言能够避免重复劳动，而且有助于让用户信任我们，因为我们全部都支持一种语言。这种希望已经获得了丰厚的回报——不管以什么方式来评价，Haskell 都是一个巨大的成功，而且它也非常符合我们最初的期望。

Paul：它有一个非常明显的优势。尽管（再次）我们拥有共同的梦想，每一个人各有所长。彼此信任，相互学习了很多。这是一群精力充沛、踏实肯干的聪明人之间的精彩互动。单凭个人是不可能设计出 Haskell 的。

John：折衷妥协可能会是件好事!

团队作战绝对是一大优势。我们具有互补的经验和技能，而且我认为队员组成较广的团队设计出来的语言，结果肯定会比我们任何个人独自实现的语言的应用更为广泛。个人设计出来的语言可能会更小、更简单，甚至会更优雅——但我并不认为它会更有用。

同样地，每一个棘手的设计决策都应该从各种可能的角度来检查，并且是反复检查，事实上也是如此。三个臭皮匠，赛过诸葛亮。对于某人来说非常合理的一个决定，对于别人来说，或许就是明显的漏洞百出——如果出现了诸如此类的严重缺陷，我们会一次又一次地反复拒绝某些想法，只是为了最终找到更好的想法。我认为我们的细心会在设计质量上有着很明显的反映。这听起来非常辩证，不是吗：论点+反论点=合题。

译注1：列表理解（list comprehension，或称为列表包含），这是 Python 中很具特色的一个特性。它实际上是对列表进行映射处理，即对列表中每个元素进行操作，然后再生成一个新的列表。一般用它可以将一个列表经过变换成为新的列表。

8.2 函数式编程之路

是什么使得函数式编程语言不同于其他语言？

Simon：噢，这很容易回答：控制副作用。

John：嗯，很显然是对副作用的严格控制。第一类函数（尽管它们在命令式语言中也获得了越来越多的应用），用于纯函数式运算的简明符号——从创建一种数据结构一直到列表理解等所有纯函数式运算。我认为轻量级类型的系统也是非常重要的——无论它们是 Scheme 和 Erlang 这种纯动态类型系统，还是 Haskell 和 ML 这种基于多态推理的系统。从这些类型并没有妨碍你来看，这两类都是轻量级类型系统，即使你频繁地使用高阶函数时也是如此——而这正是函数式编程的核心。

我认为惰性评价也是重要的，不过当然它不是在每一个函数式语言中都有的。

Paul：抽象、抽象，还是抽象——这对于我来说，包括高阶函数、monad（一种抽象的控制）、不同的类型抽象等。

使用没有副作用的语言来编写程序有什么优点？

Simon：你只需要关心值，而不需要关心状态。如果你给一个函数相同的输入，它就会给你相同的输出。这对推理、编译、并行都有影响。

正如（来自 Amgen，Inc.的）David Balaban 所言，“函数式编程缩短了从想法到代码 brain-to-code 的距离，这比什么都重要”。

John：噢，事实上现在并没有这样的语言。如果 Haskell 程序使用正确的类型，或者使用“不安全的”运算，就可能出现副作用。ML 和 Erlang 程序可能会出现副作用。副作用不是作为所有编程的基础，而是这些语言的异常情况；不仅不能鼓励这些副作用，而且还要对它们进行审慎的控制。因此我要重新解释你的问题：没有副作用给编程带来了多大的优势？

现在，很多人都开始谈论推理了，我也要讨论一下，不过，是从非常实际的观点来看的。我们以测试命令式语言的一个函数为例。测试代码时，要提供各种不同的输入，并检查它们的输出是否与规范定义相一致。但是，在有副作用的情况下，那些输入不仅包括函数的参数，而且还包括函数读取的那些全局状态的部分。同样地，它们的输出不仅包括函数的结果，而且还包括它所修改的状态的所有部分。为了有效地测试函数，你要把测试输入放在它要读的那些状态中，并且读它修改的那部分状态。但是你甚至没有直接地访问过那些部分，因此最终不得不通过其他函数调用序列来间接构建你需要的测试状态，并且在测试函数之后，要通过更多的调用来观察该函数的影响，吸取希望改变的那些信息。你甚至不能确切地知道读取和改写了哪部分的状态。而且，通常来说，为了检查函数的后置条件，你要同时访问它运行前的状态和运行后的状态！因此要在测试之前复制它的状态，还要在运行之后访问所有的相关信息。

将这个与测试纯函数相比，后者只取决于它的参数，而且它的唯一作用就是提交结果。这非常简单。即使对于那些肯定会有许多副作用的程序来说，使用一个瘦副作用效应包装器，把尽可能多的功能分解为高度可测试的无副作用的代码，这样做也是有意义的。Don Stewart 在最近发布的博客中，对这个应用于 XMonad window

管理的方法给了一个非常有趣的描述（注 2）。

把一个函数所依赖的所有东西都当做参数来传递，也有利于理清它们的依赖关系。在 Haskell 中，你甚至可以编写一个操作很大“状态”的程序，可以把它作为参数传递到所有函数中，并且由所有修改过它的函数作为结果返回。但是，你并不倾向于这么做：你仅仅需要传入函数所需的信息，并让它仅仅返回函数自己生成的信息。这使得依赖比在命令式代码中更为清晰，在命令式代码中，任何函数基本上都要依赖状态的任意部分。而且，忽视这种依赖恰恰会导致那些最为棘手的缺陷！

最后，只要你开始带有副作用编程，评估顺序就变得非常重要。例如，你必须在做任何文件操作之前打开这个文件句柄，你必须记住每次都需要关闭它，并在关闭它之后不能对它进行任何操作。每一个有状态对象在它的顺序上都要增加一些约束，你也许会用它的 API，并且这些约束随后会被大量的代码段所继承——例如，这样的一个函数必须在它的日志文件被关闭之前调用，因为它有时会编写一个日志入口。如果你忘记了一些限制，并以错误的顺序调用了函数，那么就会“砰”的一声提示失败!你的程序失败了。这个是一种重要的缺陷源。例如，微软的静态驱动验证（Static Driver Verifier）本质上是检查那些你关心的 windows 内核中有状态对象强加的约束。如果程序没有副作用，那么你就无须担心这个问题。

我最近处理了一个最棘手的缺陷，它要追溯到拥有有状态 API 的 ERlang 库，我在代码中以一个非常复杂和动态确定的执行顺序，来使用这个库。最后，能使我的代码运行的唯一方式是在标准的 API 的基础上构建无副作用的 API。我害怕我不够聪明到足以让带有副作用的代码运行的地步！（等一下，或许命令式编程会更容易，而不需要有多年的函数式编程经验……）☺

噢，我是不是提到了易于并行化呢？

Paul：哈，你神智很清醒，为解决这个难题而享受一下吧!☺

John 提到了并行化。在计算机领域，是否存在其他的变化，使得函数式编程要比以前更令人满意和更加需要？

Simon：我相信，从长期趋势来看，控制副作用对所有的编程语言来说都会变得越来越重要。下面这个 5 分钟的视频中解释了我的意思：

http：//channel9.msdn.com/ShowPost.aspx？PostID=326762

随着我们转到更高级的语言上面，这些语言带有高阶顺序控制结构和循环，而不是仅仅只有 jump（跳转）和 goto 语句，结构化编程的自然演化是否要求避免这种副作用？没有副作用的编程会不会成为超越这种状况的下一阶段呢？

Simon：这可能是一组透镜，通过这组透镜，你就能看出端倪。在这一点上我有一些谨慎，这是因为人们在说“结构化编程”的时候，通常是各有所指。我一直认为，如果你想使用坦率、准确的词汇让你的采访原声片段言简意赅，你就必须非常小心。

如果你还记得 Dijkstra（译注2）那封信中的经典名句：“Goto 被认为是有害的，为了让你的程序更易于理解

注 2：*http://cgi.cse.unsw.edu.au/~dons/blog/2007/06/02#xmonad-0.2*。

译注2：Dijkstra，荷兰计算机科学家，荷兰皇家科学院的院士、美国国家科学院的院士，1972 年图灵奖得主。他是废除“goto 语句”的领导者。在 1968 年的一封叫做 Go To Statement Considered Harmful 的信中，Dijkstra 认为应当在高级语言中废止不加限制地使用 GOTO 语句，因为它使分析和验证程序正确性（特别是涉及循环）的任务变得复杂。

和编译等，请弃用 Goto 语句”。然后，你可能就会把纯洁性看成去掉某些赋值的方式，使你的程序可读性更强。不过我认为，把函数式编程看成是唯一符合美学的实践，这是一种错误的观点。（“我们将取消这些罪恶的坏东西，同时却让你过着无聊而且艰难的生活”）。

我们并非只是说，“我们不会让那些事情再去烦你”，而是说，“我们不会让某些事情再去烦你，不过，作为交换，我们会为你提供惰性计算（译注3）、高阶函数和极为丰富的类型系统，最后还有 monad”。这种改变会迫使你从一个全新的角度来考虑编程，因此这不是一个没有痛苦的过渡，不过它的回报也非常丰厚。

在函数式编程中错误处理有什么改变？

Simon：你可以用一种新方式来考虑错误，相比于“异常传播”，这更像是“误差值”。误差值就像在浮点数中的 NaN。这样进行错误处理，它提供的就是一种更多的面向值而更少面向控制流的观点，这是一个非常好的事情。结果是，函数的类型更可能是用来表示它的错误行为。例如：

```
item lookup( key )/* 如果没有发现的话可能会抛出异常 */
```

而我们有：

```
Lookup :: Map ->Key->Maybe Item
```

其中，Maybe 数据类型意味着在扫描值的过程中可能会有失败。

在函数式编程中，调试有怎么样的变化？

Paul：好的，首先，我一直觉得“执行跟踪”的调试方法在命令式语言中是中断的，甚至对于命令式程序来说也是如此！实际上，一些著名的命令式程序员会避开这种方法，他们赞成使用更严格的基于测试或者验证的方法。

现在，对于函数式语言，特别是惰性函数式语言来说，好在它没有任何特别有用的“执行跟踪”的概念，因此那种调试方法并不是一个很好的选择。GHC 有一个跟踪图形归约机的设备，这个设备是以它的赋值机制为基础的，不过依我看来，这样做会过多地揭示它的赋值过程。取而代之，人们设计了调试器，比如基于“数据依赖”的 Buddha，这更加符合函数式编程的支持性原则。但是，可能是非常奇怪，在我这么多年的 Haskell 编程过程中，事实上，我从未使用过 Buddha，或者 GHC 的调试器，或者任何诸如此类的调试器。我发现测试工作得很好；使用 QuickCheck 或者类似的工具来测试小代码段使得事情变得更加严格，然后，这是关键的一步，仅仅简单地学习代码，看看它们为什么不按我所希望的那样运行。我怀疑很多人的编程方法都与此大同小异，否则，在 Haskell 调试器方面应该有非常多的研究，这是一个昙花一现的话题，现在已经不再流行了。调查人们实际上是怎么样调试 Haskell 程序的，这是一件非常有趣的事情。

这就是说，有另一种调试获得了更多的关注，也就是，描绘了时间和空间，尤其是空间。空间泄露是惰性函数式编程的一种潜在的灾难，而对于空间描述则是消除这些潜在灾难的重要工具。

译注3：惰性计算（Lazy Evaluation），又称懒惰计算、懒汉计算，是一个计算机编程中的概念，它的目的是要最小化计算机要做的工作。它有两个相关而又有区别的含义，可以表示为“延迟计算”和“最小化计算”，本条目专注前者，后者请参见最小化计算条目。除可以得到性能的提升外，惰性计算的最重要的好处是它可以构造一个无限的数据类型。惰性计算的相反是热情计算，也叫做严格计算。这是一个大多数编程语言所拥有的普通计算方式：惰性计算。

如果我们没有多年使用命令式语言的经验，是不是更容易学习函数式编程语言？

Simon：我说不好。一般来说，学习函数式编程的能力看来好像与成为一名优秀的程序员密切相关。毫无疑问的是，学习函数式编程要改变一下思维方式，不过，优秀的程序员完全可以做到这些。如果要把函数式编程的小众市场状态归咎于大多数程序员最初都受过命令式技术方面的培训这个事实，我认为这是强词夺理。

更大的原因在于这是一种极大的锁定效应。很多人在使用 C++，因此，编译器、工具和程序员池等都会很好地支持 C++。不过，甚至那也算不上什么非常充分的理由：看一看 Python 或者 Ruby 的快速成功吧！

John：不，这是一个神话。大量的经验都会变得很有用——无论它是否理解抽象、算法知识和数据结构在编程中的重要性，或者仅仅是理解编程语言是形式语言。总体来说，我教那些 C/C++迷们用 Haskell 来做，要比那些没有经验的初学者强得多。他们能够理解什么是“句法错误”；他们可能并不理解类型系统，但是他们知道什么是类型错误；他们知道给变量起一个提示性的名称并无助于计算机“理解”他们的程序并修复其中的缺陷！

我认为之所以出现这种情况，是因为命令式程序员发现函数式编程比他们预想的更加困难。富有经验的程序员很容易就能学会一种新语言，因为他们可以直接地迁移变量、赋值和循环等基本概念。不过，这对于函数式语言来说是行不通的：即使是富有经验的程序员也会发现他们需要学习一些从未见过的新概念。因此，他们认为函数式编程很“难”——另一方面，他们也会比完全的初学者更快也更容易地学习并使用这种语言。

Paul：我常说积习难改，不过现在我不再这么确定了。我认为最好、最聪明、最富有经验的程序员（任何类型的）会发现 Haskell 很容易学习，并会喜欢上 Haskell。他们的经验有助于充分意识到抽象、对影响的严格控制、强大的类型系统等的好处。而没有什么经验的程序员通常是感觉不到这些的。

为什么您认为没有函数式编程语言会进入主流？

John：糟糕的市场营销！

我指的并不是宣传，我们已经做了大量宣传。我指的是仔细选择一个小众目标市场来统治，接着通过坚定不移的努力，使得函数式编程成为迄今为止最适合这个小众市场的方式。在 20 世纪 80 年代的那些快乐时光中，我们认为函数式编程是样样都行——不过，把一种新技术叫做“样样都行”就等同于叫它“样样稀松”。人们希望用这种语言做什么呢？John Launchbury 在它的 ICFP 邀请访谈中已经把这个问题说得很清楚了。Galois 连接（Galois Connections）在打出“用函数式语言编写的软件”这张招牌时，几乎是一败涂地，而他们聚焦于“高可信软件”后就变得越来越强了。

很多人对于技术革新是如何发生的并没有什么概念，而且认为好技术会轻轻松松地自己占据统治地位（酒好不怕巷子深），但世界并非如此简单。

一些图书，比如说 Moore 的《Crossing Chasm》[HarperBusiness]和 Christensen 的《Innovator'Dilemma》[Collins Business]，对我在这方面的思考有着极大的影响。如果说 20 世纪 80 年代存在一个小众目标市场的话，那就是并行编程——不过，结果证明这在当时根本不重要，直到最近，由于计算机架构师们巨大的独创力，这种状况（随着多核的出现）才有所改观。我认为这比技术性问题更重要，固然技术也是很重要的，比如说低性能等。

Paul：因为它同传统的编程截然不同。这种差异使它难以被人们接受、学习，而且也难以获得库、实现等的支持。

那种情况改变了吗？

Simon：函数式编程是一种长期赌注。它是以一种完全不同的方式来考虑整个编程。这样一来，人们就很难学习它；而且，即便是学会了，也很难采用它，因为它是一种革命而非改良。

函数式编程最终是否会变成主流，前景依然并不明朗。已经明朗的是：函数式编程影响了主流编程语言，而且这种影响正在日益增强。这些例子包括：垃圾收集、多态类型（“泛型”）、迭代器、LINQ、匿名函数等。

函数式编程变得影响更大，这有两种原因。首先，随着程序规模的增大，而且人们越来越关心正确性，无限制的副作用带来的成本和更多函数式的风格带来的好处变得更加明显。其次（尽管可能是更短期的影响），多核和并行已经重新唤起了人们对于纯计算的兴趣，至少是对于那些精心控制副作用的计算很感兴趣。最近的一个例子是软件事务内存（STM）。

所有这些，在后来的 Haskell 语言社区中都有显著的增长，而且一些面熟的函数式语言不是不可能成为主流。（不过，据我猜想，即使它成为了主流，它也会被叫做 Java 3，并在语法构成上看来像是一种面向对象语言。）

John：当然了。来看一看 Erlang——这种专注于特定的小众市场：电信系统所需的健壮的分布式系统，其中，每一个电信相关的任务都有一个巨大的库集合，而且非常幸运的是，因特网服务器从根本上也需要相同的特征。即使在电信系统中，Erlang 可能也不是主流，不过它有很多很多用户，而且用户规模还在呈指数方式增长。为电信应用程序选择 Erlang，在今天是一个毫无争议的选择——这是一项成熟技术。

Haskell 没有 Erlang 那么火，不过受关注程度上升得比较快，而且，还出现了各种各样没有预想到的应用。OCaml 也是这样。

多核为函数式编程提供了一个极为难得的机会——这里有一个广泛认识，那就是我们不知道如何为它们编程，许许多多的人开始考虑并行系统编程的替代方法，其中包括使用函数式编程。有趣的是，你还会听到遗留代码自动并行化被描述成一个“短期”的解决方案，而函数式编程则被描述成为一个有吸引力的“长期”方式。不过事实是，如果你今天开始开发一个产品，在一年的发布时期中，它必须能充分利用八核，那么编写顺序结构 C 程序并希望它能自动地并行化解决你的问题，这是一种风险非常高的策略。选择 Concurrent Haskell 或者 SMP Erlang 没有风险，因为现在这种技术已经过关了。

市场上已经有双核的 Erlang 产品了，因为它利用了额外的内核，所以速度会是其他产品的两倍。从现在开始后的短短几年内，易于并行化将成为一种决定性的优势，而函数式语言将会有机会从中脱颖而出，并且会有出色表现。

Paul：是的，潜在的采用环境正在发生变化，具体原因如下：

- 其他一些语言采用了其中一些好主意，因此它不再是一个彻底的改变。
- 过去 15 年中进入这个行业的程序员，有更多的机会接触现代编程语言思想、数学和形式方法——从而再次使思想不太极端。
- 现在有更多的库、实现和有关的工具，这样一来，就使得使用语言更容易、更实用。
- 现在有一个重要的用 Haskell 语言（或者其他函数式语言）编写的应用程序主体，这使人们对于它的可用性充满了信心。

50 年后，函数式编程仍然有用，这个事实是否会在计算状态方面对我们有一些什么启发呢？

Simon：我认为它告诉了我们一些关于函数式编程的东西。我喜欢函数式编程，因为它既有很强的理论性（得以立足的根本基础），又有很强的实践性。

说它们具有“很强的理论性”，我的意思是指语言及其实现（特别是像 Haskell 这样的纯语言）非常紧密地以异常简单的数学为基础，它不同于功能强大但更加专门化的 Python 或者 Java 之类的语言。这意味着函数式编程不会过时——函数式编程代表的是考虑计算的一种基本方式，因此，它根本不是一个时髦的应景之物。

而说它们具有“很强的实践性”，我的意思是现在函数式编程比 10 年前有了更多的应用，因为实现和库都有了极大的改进。这使得更多的受众可以受益于理论性方法带来的好处。

随着人们更加关注：

- 安全性
- 并行性
- 由于副作用而带来的缺陷

函数式编程你会用得越来越多，而且使用也日益广泛。如果你愿意的话，计算可以达到这样的状态：函数式编程的成本没有以前那么重要，而它们的好处则更有价值。

8.3 Haskell 语言

The Haskell Language

回到 John 此前的回答上，设计类系统时，是什么让您兴奋不已呢？

Simon：我们知道我们在如何使任意类型等价、如何显示和打印任意类型以及如何处理数字类型等方面存在着大量的问题。我们想使用整数、浮点数、双精度数字和任意位整数。我们不希望程序员写程序时必须要加上 `int`、`float` 和任意精度的整数。

我们想要这样一种方式：你只须编写一个 A+B，就可得到正确的结果。对于这个问题，ML 解决方案允许你编写一个 A+B，不过加法的类型需要本地解决。如果你编写 `f(x, y)=x+y+1`，系统会说，“啊!你需要告诉我更多的东西。你需要给 `f()` 一个类型签名，这样我才能知道这个加号是两个整数、浮点数还是双精度数相加”。

这样一来，你就需要在程序的很多地方加上类型信息。

Simon:比这个还要糟糕。在浮点型、整型或者双精度类型上调用这个特殊函数可能是有用的。而把它固定在一种类型上则是一件痛苦的事情。

你丢掉了泛型。取而代之，你必须编写三个函数：`float`、`double f` 和 `int f`，所有的函数都有相同的函数体，只不过类型签名不同而已。然后，在你调用它们的时候，这些函数中的哪一个会被编译器调用？你又回到了加上 `int` 这个问题上了，只不过是高了一个层次。

我们对此感到很苦恼。这让我们觉得有些不对劲。这就是类系统所要解决的问题，因为，对于所有的类型来说，你只需要编写 `f(x, y)=x+y+1` 一次即可。它具有类型 `Num a => a->a ->a`，而且它会适用于任何数值类型，包括你根本没有想到的那些类型!

188

它们必须是 `Num` 的实例，不过最妙的是你以后可以创建一种类型——在确定 Haskell 标准、定义 `Num` 类而且编写出这个 `f` 函数 10 年之后。你可以实例化一个 `Num`，而且以前编写的函数可以很好地同它一起运行。

这就是让人兴奋之处。我们似乎解决了一个非常棘手的问题。最初的工作是由 Philip Wadler 和 Stephen Blott 做的。

它也解决了相等问题。对于相等问题，ML 有一种不同的解决方案。如果你通过类型成员`:: [a] ->Bool` 定义一个成员，它要问你这个值是否是 a 列表的成员，该运算符要求你为相等比较类型 a 的值。让所有的值都支持相等，这是一种可能的解决方案，但是我们不喜欢这样做。你无法合理地为相等比较函数。

ML 会说：我们会给你一个特殊的类型变量`'a`。成员有类型成员`:: a->['a] ->Bool`。这个'a'被称为相等类型变量。它的范围只包括承认相等的类型。因此，现在你可以应用成员来表示整数或者字符，不过，你无法用成员来表示函数，因为，`'a` 随后会被一个函数实例化；所以用成员来表示函数是不合法的。

ML 与重载数值的语言的解决方案不同，不过它也是无处不有的类型系统，所以这些`'a` 会在描述中无处不在。它以一种完全不同的方式，解决了一个关于相等的非常特殊的校验问题，不过，它对于排序问题没有帮助。如果你想要对 a 列表进行排序又会怎么样呢？现在你没有相等问题，只有排序问题。类型类（Type class）也解决了这个问题。在 Haskell 中你可以编写

```
member :: Eq a => a -> [a] -> Bool
sort :: Ord a => [a] -> [a]
```

由此精确地指明类型 a 必须有哪些属性（分别为相等或者排序）。

这就是我们为什么如此兴奋的原因，因为它依靠单一的强大的类型系统级机制，为我们解决了很多问题，否则我们就只有专用的和不同的解决方案可用。一把锤子能敲破一大堆坚果，而一个坚果也没能敲好，即便是只有一个坚果也敲不好。

Philip Wadler：类型类会影响 Java 的泛型工作方式，这对类型类来说是一件好事。下面的这个 Java 方法：

```
public static <T extends Comparable<T>> T min (T x, T y) {
      if (x.compare(y) < 0)
    x;
      else
    y;
}
```

它与 Haskell 的方法非常类似：

```
min :: Ord a => a -> a -> a
min x y = if x < y then x else y
```

189

后者更短。一般来说，在 Java 中，如果说某个类型变量扩展了一个接口（这通常被参数化成相同类型的变量），这与在 Haskell 中说某个类型变量属于某个类型类是一样的意思。

我非常确定它会有直接的影响，因为我（与 Martin Odersky、Gilad Bracha 和其他很多人一起)受邀参与了 Java 泛型的设计。我认为在 C#中泛型受了这种设计的影响，不过，我没有参与 C#的这项工作，因此我无法确认。C++中的“概念”这种新观点也是非常类似的，而且为了进行比较，他们在论文中引用了 Haskell 中的类型类。

您会觉得 Haskell 程序员什么时候才会意识到这种强类型的重要性呢？

Simon：Haskell 的类型系统十分丰富，足以表达多种设计。

类型检查不仅仅只是避免 5+true 这样的低级错误。它为你提供了一种描述和讨论程序设计和结构整体水平的抽象，因为在这里，面向对象那些人会画出 UML 图表，Haskell 那些人会编写类型定义（ML 那些人则会编写模块签名）。这就好多了，因为它会更加精确，而且是机器可检查的。

Philip：这里有一个古老的奇闻趣事（注 3）。Software AG 销售一款叫做 Natural Expert 的商业数据库产品，在这个产品中，数据查询和操作都是用他们自己的函数语言，类似于 Haskell。他们要进行为期一周的课程培训。在培训课程刚开始时，开发人员经常抱怨连天：类型检查给他们带来了很多可怕的类型错误。在培训课程结束时，他们发现：他们编写的大部分程序，只要通过了类型检查，都能很完美地运行。因此，类型为他们提供了所需的所有调试。简而言之，在课程开始时，他们认为类型检查是自己的敌人，但是到课程结束的时候，他们觉得类型检查是自己的朋友。

我并不是试图说你写的所有程序只要通过了类型检查就能很好地运行。但是，类型检查确实发现了大量的错误，并且使调试容易多了。

随着使用更加复杂的特性，类型似乎变得特别重要。例如，当你为了保持井然有序而使用类型时，使用高阶函数就会非常容易。多态函数揭示了关于它们类型的大量信息。例如，如果你知道某些东西使用了以下类型：

```
l :: (Int -> Int) -> [Int] -> [Int]
```

（使用一个函数把整数转换成布尔型和一个整数列表，并且返回一个整数列表），它几乎无所不能，但是如果它使用以下类型：

```
m :: forall a b. (a -> b) -> [a] -> [b]
```

（对于任何类型的 a 和 b，使用一个函数把 a 转换成 b 和一个 a 的列表，并返回一个 b 的列表）然后，你会对它了解得更多。事实上，这种类型自身有一种理论（注 4），函数符合这种理论，并且通过这个类型，你可以证明：

```
m f xs = map f (m id xs) = m id (map f xs)
```

其中，为了获得一个新的列表，map 对列表的每个元素都使用了函数，并且 id 是这个函数的标识符。最有可能的是 m 自己就是一个 map，因此（m id）就会是标识符。不过，m 也可能重新安排元素——例如，它可能会反转输入列表然后再应用这个函数，或者是先应用这个函数然后再从结果中逐一取出其他元素。不过，这就是它的全部所能了。类型保证了它必须对输入列表的元素应用该函数以获得输出列表的元素，同时还保证了它无法查看每个元素值以决定作何处理，该操作只能在列表中完成。

对我来说，类型系统最不可思议的事情就是它们与逻辑联系非常紧密。它们有一个高深的好属性，叫做："作为类型的命题（propositions-as-types）"，或者叫做 Curry-Howard 同构，它声称每一个程序都是一个命题证明，并且这个程序的类型就像这个程序证明的命题，并且评估一个程序就像是简化这个证明。构造数据最基本的方式（记录、变量和函数）恰好与逻辑中最基本的三个结构（合取、析取、蕴含）精确对应（注 5）。

注 3：Hutchison, Nigel 等 "Natural Expert：一种商业函数式编程环境（Natural Expert: a commercial functional programming environment）"，函数式编程学报（Journal of Functional Programming）7(2)，1997 年 3 月。

注 4：Wadler, Philip. "Theorems for Free"，第 4 届函数式编程与计算机体系结构国际会议（4th International Conference on Functional Programming and Computer Architecture），英国伦敦，1989 年。

注 5：Wadler, Philip. "新语言，旧逻辑（New Languages, Old Logic）" Dr. Dobb's Journal，2000 年 12 月。Dr.Dobb's Journa 是全球著名的程序设计方面的英文杂志。

它被证明适合于所有类型的逻辑系统和程序，因此，对于设计类型编程语言来说，这不是短暂的巧合，而是一种有价值的深刻原则。事实上，它为你提供了一种设计技巧：考虑一个类型，为了给这个类型赋值向语言中添加构造函数，并为了从这个类型中取值向语言中添加析构函数，同时你需要遵守一些法则：如果你构建了一些东西并且把它拆开了，你将会获得跟开始一样的东西（这被称为 beta 法则），如果你拆开了某些东西并且重新构建它，你也将获得跟开始一样的东西（这被称为 eta 法则）。这非常的强大，也非常美妙。大多数情况下，当你设计某些东西的时候，感觉可以随心所欲，比如你可以使用 5 种不同的方式而且不清楚哪一种才是最好的。但是这告诉我们，对于函数式语言有一个核心，而这根本不是任意的。

现在，我们与计算机科学家达成了一致观点，他们把证明输进电脑，因此电脑可以检查它们是否正确，而且由于程序和证明之间、类型和命题之间的这种深入的联系，这个过程和函数式语言一样基于相同的原则和类型系统。因此，我们开始看到事情正在融合，并且类型允许你越来越多地描述你的程序如何运行，并且编译器可以保证你的程序有越来越多的属性，而且它会慢慢地变得更加平常，当你编写程序的时候，它会证明程序的属性。有时候，美国政府会坚持要求证明军用软件的安全属性。

我们将会看到这一趋势仍在继续。现在，操作系统并不能提供非常强大的安全性保证，不过我认为我们会看到改变，并且类型系统会成为其中非常重要的一个部分。

惰性是否适用于其他语言，或者是因为 Haskell 的其他功能而使它更适合于 Haskell？

jhon：惰性结果来自不可预测的复杂控制流。在 Haskell 中，这不是一个问题，因为赋值秩序不会影响结果——你可以想让控制流变得多复杂就能变得多复杂，而且它不会影响你的代码工作的难易。惰性能够而且已被添加到其他语言当中，而且它也不是那么难以模拟。但是，当惰性和副作用搅在一起的时候，就会乱成一团。事实上，让那样的代码正常运行是不可能的，因为你不要奢望理解副作用为什么会以那种方式出现。我已经从 Erlang 中得到经验，我是在代码中使用了带有辅助接口的一个库来模拟惰性。最后，能够使代码符合我的要求的唯一方式是在辅助接口之上去构建一个纯函数式接口，这样的话，我的 lazy 代码就可以是无副作用的了。

因此，我认为应该这样回答你的问题：没错，惰性可以导入到其他语言之中——但使用它的程序员必须要避免在他们的代码中出现副作用。当然，LINQ 是一个优秀的例子。

Haskell 是否有其他的特性可以被其他语言借鉴，使它们更加有用的或者安全？

Philip：来自 Haskell 的若干特性正被纳入或已被纳入许多的主流语言。

许多语言都出现了函数式闭包（lambda 表达式），包括 Perl、JavaScript、Python、C#、Visual Basic 以及 Scala 等。内部类作为一个模拟闭包的方式被引入到 Java 之中，而且就如何向 Java 中添加适当的（类似于 Scala 中的那些）函数式闭包，还存在着广泛的争论。对函数式闭包的影响不仅仅来自 Haskell，而且还来自所有的函数式语言，包括 Scheme 和 ML 家族等。

列表理解出现在 Python、C#和 Visual Basic（都是通过 LINQ 连接）中，还有 Scala，而且也计划加入 Perl 和 JavaScript 中。Haskell 没有引入列表理解，不过为列表理解的普及做了非常大的贡献。C#、Visual Basic 和 Scala 中的理解除了列表以外，还适用于结构体，因此它们更类似于 monad 理解或者“do”符号，这二者在 Haskell 中都引入了。

Java 中的泛型类型还受到了 Haskell 中的多态类型和类型类的强烈影响；我曾帮助设计了 Java 中的泛型，而且还与人在这方面合著了一本图书《The features in Java in turn inspired those in C# and Visual Basic》，由 O'Reilly 出版（注 6）。Scala 也使用了类型类。现在，C++正着眼于吸收被称为“概念”的一个特性，它和类型类也是非常类似。Haskell 还影响了许多使用并不广泛的语言，包括 Cayenne、Clean、Mercury、Curry、Escher、Hal 和 Isabelle 等。

John：除此之外，还有 C#的匿名委托 anonymous delegate 和 Python 的列表理解。函数式编程思想突然变得无处不在。

Paul：我看到很多人在学习 Haskell，却很少在实际编程工作中使用，不过，他们声称它改变（好转）了他们使用命令式语言的思维和编程方式。而 Haskell 对于主流语言以及最近的新语言影响巨大。因此，我们必须正确做事，而且我们似乎已经在影响主流，即使我们现在还没有成为主流。

一种语言的设计和用该语言编写软件的设计之间有什么联系？

Simon：你所使用的语言对使用该语言进行程序设计有着深远的影响。例如，在面向对象领域，很多人使用 UML 来描述设计。在 Haskell 或 ML 中，则使用类型签名来代替。在函数式程序的初步设计阶段，大部分是由类型定义构成的。但是，与 UML 不同，所有这些设计都被列入最终产品之中，而且自始至终都是由机器在进行检查。

类型定义也为写下常量类型提供了一个很好的地方，例如，“这个列表一直不空”。目前，这些声明并没有实现机器检查，不过，我相信它们最终会逐渐实现。

鲁棒类型改变了程序维护的方式。您可以更改数据类型，也知道编译器将指向随后会修改的所有地方。对我来说，这是使用具有表现力的类型的一个最大的原因；我无法想象：要对它进行重大修改，修改成一种几乎具有同等自信的大型动态类型程序。

正如 John 先前所述，使用一种函数式语言会大大地改变测试的方式。

使用函数式语言有力推动了人们使用纯函数式数据结构，而不是使用内存中突变的数据结构。这会对程序设计产生深刻的影响。你可以使用 Haskell 编写命令式程序，但它们看起来很笨拙，并在可能的情况下以纯洁性来引导程序员使用纯函数式数据结构。

Paul：我喜欢 Simon 的回答，虽然他的重点基本上集中在 Haskell（或其他函数式语言）如何影响软件设计上面。这个两面性的问题是：软件应用程序是如何影响语言设计的？当然，Haskell 和大多数其他函数式语言有意要成为通用语言，但用 Haskell 编写的应用程序近年来很酷的事情之一就是它们中有多少是基于 “嵌入”在 Haskell 中的领域特定语言（DSL）（我们常称这些为“特定领域嵌入式语言”）。这方面有很多例子，图形、动画、计算机音乐、信号处理、分析、印刷、金融合同、机器人，还有更多和大量的图书馆设计也基于这一理念。

对于房地产代理商来说，“位置，位置，还是位置”是房地产的三个最重要的因素，与此类似，我认为“抽象，抽象，还是抽象”则是程序设计中三个最重要的概念。而且对我来说，一种精心设计的特定领域语言就是一个域的最终抽象，它正好不多也不少地捕获了正确的信息。Haskell 之所以如此之棒，原因在于它提供了用于创建这些 DSL 的方便和有效的框架。这套方法并不完美，但它是相当不错的。

注 6：Naftalin，Maurice 和 Philip Wadler。Java 泛型和收集（O'Reilly，2000）。

Philip：函数式语言可以很容易地在内部扩展。Lisp 和 Scheme 就是典型的例子；请看一下 Paul Graham 的这本书（注 7）：Lisp 是如何成为构建最早的网络应用程序（后来成为了雅虎产品）之一的秘密武器，特别是 Lisp 的宏如何成为构建此软件的关键所在。Haskell 还提供了许多特性，这样更容易扩展语言的功能，包括 lambda 表达式、惰性、monad 符号，以及（in GHC）Haskell 元编程模板等。

Paul 已经提到，正是这些使得 Haskell 成为了领域特定嵌入式语言界最受欢迎的语言。它在较小规模的层次上也显示出了很强的功能，比如在为解析器组合子（parser combinator）或者精美打印构建小型库时。如果想要真正了解函数式编程的功能，从这两个例子开始就再好不过了。

Haskell 中的惰性还对如何编写程序有着深远的影响，因为它允许你分解以其他方式难以实现的问题。还有一个我喜欢考虑的地方是惰性允许以时间换空间。例如，我可以返回一个包含所有值（空间）的列表，来替代考虑怎么传递序列参数值（时间），事实上，惰性确保了列表里的值将根据需要进行逐一计算。

考虑空间往往要比考虑时间容易：空间可以直接形象化，而时间形象化则需要动画。来做一个对比：浏览当天的日程表和观看当天的事件录像！因此，利用惰性可以深刻地改变你解决问题的方法。前面提到的解析器组合子就是一个例子，它返回所有可能的解析清单；惰性则确保这个列表按需进行计算。特别地，如果你乐意接受第一个解析，那么其他的就都不会产生了。

8.4 传播（函数式）教育

在教授大学生编程的时候你悟出了什么道理？

Paul：多年以来，可能现在也是如此，函数式语言主要是导论课上介绍，因为它们非常易于学习，而且从很多命令式计算的细节中抽象出来了。现在，从长期发展来看，这可能是个错误!其原因在于学生们会很快给出定论，认为函数式语言是一种“玩具性”语言，因为毕竟它们只在他们导论课上介绍，而且大多使用“玩具性”示例。而且，一旦他们发现副作用的“威力”，他们中的大部分已经不能回头了，这是多么悲哀的一件事啊!

对我来说，如果初学者通常意识不到函数式语言的好处，那就再好不过了。只有在你编了一段时间的程序后，好处才开始逐渐显现。

在耶鲁大学，我们有一门函数式编程的课程，选这门课的大部分都是高年级学生。我没有为了展示函数式语言的真正实力，而给他们出一些难题和大题，甚至是高等数学这类的问题。更重要的是，我可以说：“把这些放到你的命令管道中并且消化它。”并且我们经常拿 C 代码和 Haskell 代码作比较——这是非常有启发性的，而对于第一门语言学的是 Haskell 的那些学生来说，你就不能这么干。

计算机科学出了什么问题，我们应该怎样来教授它？您是如何解决它的？

Paul：我想要写一写我的个人培养目标，我希望其他人能够从中发现一些有趣的东西，甚至是带有挑战性的东西。

按照字面意义理解，似乎有成百上千种教授如何编程的图书，或者如何用某种语言编写程序的图书。这些书

注 7：Graham，Paul 著。黑客 & 画家（Hackers & Painters）（O'Reilly，2004）。

里的例子来源通常是五花八门，而例子却往往相当蹩脚，都是从斐波纳契数列和阶乘，到字符串到文本处理，再到简单的问题和游戏。我想要知道的是，能否编一本书，其主要议题不是编程，而是用一种编程语言作为主要工具来教授主要概念。

我想你会说："那些有关操作系统、网络、图形或者编译器的书，如果它们使用语言来广泛地解释那些材料的话，那就是你想要的书！"，但是，我所感兴趣的是离核心的计算机科学相距甚远的议题。因此，我在考虑特定科学的一些事情，包括物理、化学、天文学，甚至是社会科学、经济学等。而且，我还想知道能不能更进一步教授艺术学科的一些知识，特别是音乐知识。

我认为函数式语言，特别是 Haskell 这样一种对特定领域嵌入语言提供高度支持的语言，应该是一种教授除了编程以外的概念的杰出媒介。编程的一大优点是强制你精确，而函数式编程的的一大优点则是让你非常精确。对于我提到的很多学科方面的教学都会从这二者当中受益。

Simon：这个略微涉及我在英国特别是在学校接触到的一些东西。在学校我是一名管理者；每个学校都有一个理事会。当时，在学校教授"计算"的方式是比较无趣的。那时根本没有计算机科学。它基本上就是信息技术。

我所说的信息技术，是指电子表格和数据库。这就像是在说："这是一辆车，你应该怎样驾驶它"。那就是关于怎么使用电子表格的。"既然你可以开车了，那我们就来讨论一下，你要开车去哪儿。你要去伯明翰吗？如果你要去伯明翰的话，这就是我的建议：你应该怎么计划你的路线，还有你想要带上谁。"

如果你对进入项目规划、需求分析和系统集成诸如此类的东西产生了兴趣。此时在学校，你无须学习车罩下面是什么。在某种程度上，这是一种防护手段，因为在某种意义上，每个人都应该学会开车，对不对？此外，你还要知道你想要开车去哪儿，以及如何避免撞到行人。

并不是每个人都对汽车工作原理感兴趣。这很合情合理，因为大部分人只是要驾驶它们，但是有一些人应该知道它到底是怎么工作的。这就是在学校里，计算机科学或者计算应该教授的学科，至少对于那些感兴趣的人来说是应该教授的。而目前的情况则是，他们被告知"计算机是干什么用的"，但是因为枯燥乏味之极，实际上是无人理会。

我参与的工作组总部位于英国，该工作组正尝试支持教师在学校这个层面来教授计算或者计算机科学。当然，英国中学生的年龄是 11 到 16 岁。在 A 级上，有一个 A 级计算，它还牵涉到一些真正的计算机科学，这个年龄段是 16 到 18 岁，不过到这时，它们实际上也无人理会了。

中学生学习计算机科学的人数正在减少，而且大学生学习计算机科学的人数更少，这与在美国一样。部分是因为现在每个孩子自己都有电脑了，因此他们已经知道了很多关于 IT 的东西。当教授他们这些的时候，而且还是在学校的不同场合反复教授，他们会想，"这些东西相当的无聊，我为什么要对这个感兴趣呢？"

我认为这主要是错在学校教授计算的方式上。对于那些并不是真正对技术感兴趣的大多数人来说，教他们如何开车就足够了。这应该是适度的并且应该同其他科目相结合。这个工具很有用，然后你就去学习吧。这没有什么大不了的。但是对于某些孩子来说，我们教他们物理，最终只有极少数人会对这门课感兴趣。而其他大部分同学都不会去四处了解有关物理的扩展知识。同样，我认为计算学科应该要有一些概念性的东西，而人们会被这些概念所激发，因为它是如此令人激动。

196

8.5 形式体系和发展

Formalism and Evolution

在给一种语言定义形式语义中，您看到了什么价值？

Simon：形式语义为我们在 Haskell 上做的所有工作提供了保证。例如，如果你看看我出的书，你会发现其中大部分内容都会包含一些形式方法，这些形式方法尝试解释到底发生了什么。即使像事务内存这样的一些命令式的东西，相关论文里也有关于这个事务是什么意思的形式语义。

形式语义对于处理一个想法、尝试确定一些细节，以及把一些棘手的边角问题公开化来说，堪称是一种非常神奇的方式。不过，如果是在真实语言中，当所有的东西都凑到一起了，想要真正地把语言中的所有东西都形式化，这也是非常麻烦的一件事。我对标准 ML 定义（Definition of Standard ML）表示钦佩，因为我认为这是一个壮举。这几乎是唯一一种形式化描述比较完整的语言。

我想应该在何种程度上问它的好处是什么。要把描述语言方方面面的最后 10%那部分形式碎片集合转换成一个完整的语言形式描述，它的成本显然很高。这需要很多工作。它可能是 70%的工作。你从这最后的 70%的工作中获得了多少收益？或许只有 20%左右。我不知道它是怎么算出来的。对我来说，只要你是将整个语言而非该语言的一部分形式化，成本/效益比似乎就增加得相当快。即使是第一次也是正确的。

然后你试图说，“但是如果语言发展了呢？”我们一直在改进 Haskell。如果我必须将这种改变的每一个方面都形式化，这对于这种语言来说，就是一个相当大的制约，而且，ML 中还真的发生过这种事。想要精确地改变 ML 真的很困难，因为它是一种形式化的描述。

形式体系可能是对创新的一种制约。它是对创新的一种刺激，因为它有助于你理解什么是创新，但是如果某种环境规定这种语言必须完全形式化，那么，它对创新来说也是一种限制。

是否有中间立场，或许是一个半形式体系，在这个体系中你可以运动服配穿牛仔裤？

Simon：我认为这就是 Haskell 所占据的那个位置。语言定义几乎完全用英文的，不过如果你查一下相关的研究论文，你会发现许多形式体系只是针对该语言的一部分。因此，它没有在报告中系统化整理，肯定不是作为一个完整的描述。那些没有形式化描述的语言，比起 C++来，你会发现更多的形式化材料，这是非常不正式的，尽管在这个非正式的描述上面花费了巨大的心血。

- 这是一种有趣的平衡。我确确实实认为形式体系在保持 Haskell 简约方面功莫大焉。我们从来没有不加考虑地加入一些东西，根据某些原则，所有的东西都必须是合适的。它为你提供了一个极好的方式，你可以说“这看起来很乱。你确定它必须得像这样吗？”如果它看起来很乱，机会在于，它将会难以实现而且程序员难以找出它们到底实现了什么。

197

Philip：最初的一些有关类型类的论文是我和 Stephen Blott 一起撰写的，并且编入了 1989 年的编程语言原则研讨会论文集（Symposium on Principles of Programming Languages）（注 8）。它形式化了类型类的核心，并且我们尽可能使它保持既小又简单。后来，Cordy Hall、Kevin Hammond、Simon 和我本人又设法编写了一个更完整的模型（注 9）。它发表在 1994 年的 ESOP 上，由此，你可以看到我们花费了 5 年的时间来完成它！

注 8：Wadler、Philip 和 Stephen Blott.《如何让减少专用多态性》第 16 届编程语言原则研讨会，Austin, Texas: ACM Press (1989 年 1 月)。

注 9：Hall, Cordelia 等《Haskell 中的类型类》欧洲编程研讨会：LNCS 788, Springer Verlag: 241–256 (1994 年 4 月)。

我们并没有对 Haskell 进行全部形式化，但是我们尝试了形式化类型类的所有细节。因此，存在着适用于不同的目的的不同级别的模型。

ESOP 论文是作为用 GHC 实现的直接模型，尤其是把高阶 lambda 演算用作一种中间语言，它现在是 GHC 的核心。对于形式化来说这是一件好事。但是形式化需要花费很多的工作，不过，一旦你完成了它，它就能对实现提供很好的指导。一般来说，它通常看起来似乎很难实现，不过一旦你努力去对它形式化，那么实现就变得简单多了。

形式化的另一个例子是 Featherweight Java，这是我和 Atsushi Igarashi 还有 Benjamin Piercewhich 一起开发的，并于 1999 年在 OOpSLA 上发布（在 2001 年又在 TOPLAS 上重新发布了）（注 10）。当时，很多人都开始发布 Java 的形式模型，并且尽可能使它们完整。对于 Featherweight Java，我们的目标是使它尽可能的简单——我们用尽量简单的语法去完成每一件事，这种语法规则只有一页纸。最后发现这是一个好主意，因为这个模型非常简单，当人们想要添加一些新特性并建立新模型时，该模型就是一个很好的基础。因此，这篇论文后来被大量引用。

从另一个角度来说，结果证明与赋值和数组有关的泛型初始设计有一个 bug，而且我们并没有捕获到它，因为我们在 Featherweight Java 中没有包括赋值或者数组。因此，一个简单的模型可以让你熟悉情况，更完整的模型可以帮助你捕获更多的错误，这二者之间存在着一种折衷平衡。

我还参与了 XQuery 定义形式化的部分工作，XQuery 是一种针对于 XML 的查询语言，也是一个 W3C 标准（注 11）。当然，你可以从标准化委员会那里听到很多理由；在我们这个例子中，许多人说："这些形式化是什么东西？我们应该如何解读它？"他们不想使用形式化的规范标准；他们想要用英文来撰写规范，认为这样会便于开发人员阅读。不过类型系统的一部分很容易用形式化描述，而用英文来描述却很难，因此决定对于这部分内容采用规范的形式规约（译注4）。

曾经，有人提议过修改设计。而且还有一件有趣的事，尽管存在这些抱怨，标准化委员会还是要求我们这个做形式化的团队来形式化这种修改。因此，我们对这种修改进行了形式化，而且发现：即使用英文写的这个修改提议已经相当精确了，我们还是有 10 处不知道如何形式化，因为它的描述可以有多种解释方式。因此，我们解决了这些问题，并且提出了一种形式规约。在下一次会议上，我们提出了形式化体系，这种修改得到了大家的一致赞同——这在标准化会议上是史无前例的。因此在这个例子中，形式化体系的使用确实是一个巨大的成功。

关于 Haskell，正如 Simon 所言，把所有的东西都形式化通常是物无所值。因此，对于 XQuery，我们只把其中 80%形式化了，而另外的 20%固然也很重要，但要把它形式化工作量非常大，因此并没有将其形式化。

也就是说，我认为我们所做的形式化工作物有所值。

除了这个故事之外，形式化成了 Galax 的核心，它是由我的同事 Mary Fernandez 和 Jerome Simeon 实现的，现在已经成为了 XQuery 的一个关键实现。因此，这又是一个形式化可让实现更容易的例子。

我所认识的所有数学家都在说，如果数学看起来不美，那它很可能就是错误的。

Simon：是这样的。举个例子：我们忙于为 Haskell 添加类型级函数，而且努力为此找出形式化体系。我们查

注 10：Igarashi, Atsushi 等. "Featherweight Java:用于 Java 与 GJ 的最小化核心演算" TOPLAS, 23(3):396– 450 (2001 年 5 月)
注 11：Simeon、Jerome 和 Philip Wadler. "XML 的本质," 预备版本: POPL 2003, New Orleans (2003 年 1 月)
译注4：形式规约（Formal Specification）是对软件系统所要解决问题的完备、精确的描述。

到了 ICFP 关于形式化体系的一篇论文，但是我对它并不是完全满意。因此我们去掉了它。这会对实现有直接影响。我们只能东拼西凑出一个实现，然后说："就是它了，试试看吧"，然后我们就得到了一个很好的机会，第二天他们会找回来并且说："嗯，这里有一个程序，我想它会检查类型，但是实际上它却没有检查，因此，它应该这样吗？"然后我们会说"嗯，你要知道，这个实现并不进行类型检查，因此可能不会检查。不过，你有权利提出要求。"

我们从来没有将整个语言形式化，对此并没有感到不快。不过，这并不是说完全形式化一无是处，最后那 70% 的努力确实会产生一些好处。或许，成本/效益比并不是很高，但是确实有些效益。或许语言的特性之间有一些你并不了解的相互影响。你形式化了一些方面，但是你不知道如果有一个巧妙的计划 A 和一个棘手的特性 B，他们二者会相互破坏。对此，我们显得有些脆弱。

如果你在某些方面有一个很大的语言社区，人们最终会聚集在那里，并对 bug 进行归档整理。

Simon：没错，然后你可能会尴尬地说，"哦，噢！没错"，也可能会说"如果我进一步地形式化该语言的更大一部分子集，我们就可以有更好的结果"。这非常非常重要。但是，除此之外，我们并不知道 ML 做了些什么。

出现这样情况时，您有处理向下兼容问题的技术吗？

Simon：我想我们仍然在发展一种技术，不过在过去，我们或多或少地忽略过它。现在情况不同了。大约十年前，我们创建了 Haskell 98 这种语言，并把它作为一种稳定的子集语言。我们非常确定不会修改改变这种语言。Haskell 编译器默认接受 Haskell 98。除了 Haskell 98 以外，如果你想要编译任何东西，你必须给它们一些标签，上面标明：在其他扩展之后接受此语言。

一个标签用于记载：打开每一件事，而且现在它已经分为 30 多个独立扩展了。老的单个标签只能扩展其中的 15 个。如果你看一看源模块，通常你可以知道它在实际使用哪个语言扩展。事实上，对于邀请程序员确定他们使用什么语言这方面，我们已经变得更加谨慎了。

这些约束往往倾向于你不应该打破旧程序，虽然并非完全如此。其中有一些扩展打开了附加的关键词，比如说 `forall` 等。在 Haskell 98 中，你可以使用 `forall` 作为一个类型的类型变量——但是当你打开了高级类型时，forall 就变成了一个关键词，而且你不能将某个类型变量命名为 `forall`。

不管怎么说，人们很少这么做。对于大部分语言而言，扩展是向上兼容的。但是，如我所言，当你打开太多扩展的时候，肯定有一些用 Haskell 98 编写的程序会遭到破坏。

在未来是否会有 Haskell 2009 或者 2010，将所有的这些写进一个新的标准呢？

Simon：有可能。在这方面，有一个很超前的流程叫做 Haskell Prime，"Prime"这个词是用作可变名称的记号，主要是指我们还没有最终决定叫它什么。我们最初的设想是一群人进行公开辩论，并催生一种新语言和新标准，我们对其进行培育并且说"这就是 Haskell 2010"，就像我们编写 Haskell 98 时那样。事实上，现在已经很难有人愿意奉献足够的精力来干这种事了。

我想这是因为它有点像一种成功的灾难。作为使用最广泛的 Haskell 编辑器，GHC 已经成了事实上的标准。这也就是说，在实际过程中，人们不会因为不同语言编译器之间的不兼容性而遇到太多的困难。我认为这最终不会对语言有什么好处，而它挫伤了人们的激情和动力，使人们难以奉献出最宝贵的东西——时间——来完成这种语言的标准化。

是否会有一些非常准确地符合 GHC 语言标准的竞争性实现呢？

Simon：目前已经有了竞争性实现，这些实现往往都是针对某些领域的专门实现。事实上，最近，就在几周之前的 ICFP 函数式编程会议上，我们改弦更张了。我们不是尝试生成单个庞然大物 Haskell Prime，取而代之的是，我们去系统化整理语言的扩展。我们不是让 GHC 来定义它们，而是邀请人们提议他们认为应该整理哪些语言扩展，并对它们进行一些讨论，然后让一个人或者一个小组写下报告附录，这个报告说“关于这种语言的扩展应该有哪些功能，这儿有一个独立的描述”。

然后，我们还能说，Haskell 2010 是相互协调的扩展集。像它以前一样，我们还可以继续干下去，首先是通过系统化整理并和命名扩展，然后是把它们分给命名组，就像 Glasgow 扩展一样，不过它们会更为协调。

我们希望，要有一点像开源社区中发布 GNOME、Linux 之类的新版本那样来做，一直到这种语言设计被人关注。背后有很多工作在做，但是总有人用“胶带”对它进行一些包装，并且说：“各个部分都在一起工作，而它们的这种特殊的集合被叫做 GNOME2.9”。

200

在大家都极为认同的共同思想指导下，一些流程松散地集聚在一起。

Simon：是的，而且还承诺它们之间会相互兼容。这就是我们在语言方面所做的工作。语言几乎都是通过实现来定义的，因此在这个流程中间有了很多的顺序。如果非要找个理由的话，推动力不足的原因就是因为它太井井有条了。

而在库方面，正好完全相反。很多人都在开发库。你知道 Hackage 吗？那里每天都会出现一个新库。我们现在已经有 700 多个库了。这意味着什么很难说，“这个库能运行吗？它与那个库兼容吗？”如果你仅仅是一个尝试使用 Haskell 的普通用户，这就是很严重的问题。

此时，作为一个编译器作者，我的目标是脱离库的设计和维护。相反，有另外一群人们正要按照我们描述的那样在做，只不过是针对库而已。他们把它叫做 Haskell 平台（Haskell Platform）。本质上，它会系统地整理大量的库。这是非常传统的。Haskell 平台会成为一个元库，这个元库要取决于大量特定版本的其他库。它会说，“如果你使用 Haskell 平台，你就会获得大量的库，所有的这些库都带质量控制风筝标识（英国标准协会商标），而且，所有的库都在某种程度上相互兼容。”

如果它们不兼容，原因之一可能是两种库都基于同一个通用基础库的不同版本。如果你把它们混为一谈，你就会获得基础库的两份副本，而你很可能并不是想要两份。如果基础库定义了一种类型，库的两种不同副本可能会创建出同一类型的不同版本，而这些版本是互不兼容的。你可能无法实现你所期望的功能。它们不只是一个类型错误，还会给出一个令人困惑的类型错误；它会说第 8 版 P1 包 M 模块的 T 和第 9 版 P1 包 M 模块的 T 不相匹配。

回答你向下兼容的问题需要花很长的时间。我们开始更加认真地对待它。但仍然有问题，在发布 GHC 新版本时，编译器还是与库的基础包耦合得相当紧密。

201

每个人都依赖于 Prelude（译注5）。

Simon：没错，不过这是因为 Prelude 很有用。它定义了许多有用的函数。当我说“紧耦合”的时候，我的意思是编译器知道精确的实现图，并且知道它的名字和它在哪里定义的。有一些库是 GHC 深深地基于此的。

译注5：Prelude 是一个类似 C 中标准库的小函数集合。

这是不是为了在编译阶段进行欺骗呢？

Simon：是的。在某种程度上，如果编译器中有部分代码调用这个库函数，它必须知道这些函数存在并且必须知道它们是什么类型。这些知识被嵌入到编译器中。这就是考虑它的一种方式。

我们的意思是，如果修改了基础包的接口，就像版本更新中的情况一样，在未来发布的版本中，里面包含了某种对新的基础包的适配接口，它提供了和旧的基础包相同的 API，因此你可以隔离这种修改，如果你想这么做的话。所有这些都是这种费力的向下兼容的东西，我们以前都不必做这些。不过，这就是不遵守“不要为了成功不惜一切代价”这条箴言所带来的问题。

受欢迎也有它自己的一堆问题。

Simon：是的，不过在某种程度上它们是一些很好的问题。

关于创新、进一步的开发和采用你们的语言，您有什么经验和教训想要告诉那些现在和不久的将来开发计算机系统的人们吗？

Simon：函数式编程像一条实验室，人们在这里探索很多很多的令人感兴趣的想法。因为基本设置更简单，对于那些有前途的想法，我们可以近一步深入探究（例如：类型系统、泛型编程、被动编程等）。因此，如果你走进函数式编程实验室，在它周围，会发现许多有趣的东西。这些东西对你来说可能并不会马上有用，不过，这个实验室正在孕育着未来的潮流。

Paul：我总结的最有趣的一条经验就是：坚持一个理想可能会花费很高的代价，Haskell 设计中的纯洁性就是一例。找到一个正确的解决方案可能需要很长时间，看到回报可能会需要更长时间，不过最后必将成功。但是，如果你在这个过程中妥协了你的原则，你最终将会失败。

第 9 章 203

ML

ML 是一种通用的函数式语言，它是由英国爱丁堡大学的 Robin Milner 及其团队在 20 世纪 70 年代开发的。它是由设计用于描述数学证明的元语言项目发展起来的。ML 对语言设计的最有价值的贡献可能是 Hindley-Milner 类型推理算法，这个算法应用于很多的静态、隐式类型系统。这种语言引发了 Standard ML、Caml、Haskell、F#及其他语言的出现。

204

9.1 可靠性定理

你创建了 LCF，它是自动理论证明的第一批工具之一，你还创建了运行证明的编程语言 ML。它是如何工作的？

Robin Milner：在 20 世纪 70 年代，在机器辅助证明中还有一些其他的尝试。它们处于两个极端：要么是全新的发明创造（例如，使用著名的 Robinson 归结原理）用于搜索证明，要么是完全非创造性的，从某种意义上讲，它们只能检查人类执行的每一小步都是逻辑合法的（例如，Bruijn 的 Automath 系统）。顺便说一句，这两种方式对于今天的证明技术都有很大的贡献。

我曾经寻找过介于这二者之间的解决方案，在这种解决方案中，人们可以设计一个小策略（或者是小策略构建而成的战略），并把它和要被证明的东西一起提交给机器。它们应该是相互作用的：如果一个策略失败了或者部分失败了，那么机器也会说失败了，而且人们还可以建议使用另外的策略。关键问题是，策略是有风险的，只有找到真正的证明时，机器才能声称它成功了。事实上，在一个成功的例子中，机器能够骄傲地输出那个被发现的证明，以致于另外一个独立编写的程序（该程序是完全非创造性类型的）可以对它进行检查。

作为元语言的 ML（聪明的用户能够使用这种语言编写他们想到的策略）能够工作的关键是它有一个类型系统（有一些新颖性，但不是完全创新），该类型系统会让该语言绝不声称成功，除非它能够提供这个证明的每一个细节信息，而该证明在策略中只是简单描述了一下。因此，ML 是满怀希望的人类和一丝不苟的机器之间的一个合作工具。

LCF 有什么限制？

Robin：我没有看到 LCF 有什么明显的限制。现在，人们常常把非常冒险的策略写进 HOL、COQ 和 Isabelle 这样的系统之中，而且要解决的问题也慢慢变得越来越难。HOL 实际上获得了一个证明：ML 的类型系统是健全、完好的，这就像证明你父母的生殖系统很正常一样。不过，捕捉到数学家的灵感并把它作为策略，这要花费我们很长的时间。我认为更大的进展将主要来自在更复杂的理论上构建简单的定理之塔，这正是大多数数学理论的构建方式。

至于说证明那个程序可以运行，在用户能够用“每次执行都能达到这个点，在程序变量之间将维持这些关系”这种断言来注解他的待证明程序（该方法由 Floyd 和 Hoare 在 20 世纪 70 年代早期所开创）时，这是很有可能的。

205

可以用这种方法来分析程序源代码中是否含有 bug 吗？

Robin：可以!它能做很多事情，特别是对于小的关键程序，这些程序嵌在真正的产品中，比如像在汽车里的刹车盘一样。最大的问题出现在人们不能（或者拒绝）使用严格的形式来对所需的属性进行形式化。

用严格的形式来定义这些属性要付出多大的努力？

Robin：对于处理可以表达属性的逻辑的那些人来说，这确实是一个问题。ML 的角色不是要成为这样的逻辑，而是用这些逻辑能够表达证明的一种工具，同时也是找到这些证明的启发式算法的工具。因此，ML 是这种逻辑的宿主。最初以 ML 为宿主的逻辑是 LCF，它是由 Dana Scott 创建的一种可计算函数的逻辑。在这一逻

辑中，我们使用 ML（及其前身）发现并（或）检查一些理论；我可以很高兴地说，这些理论的其中之一，就是（几乎是完备的）证明从非常简单的源语言到非常简单的目标语言的编译器的正确性。

这能移植到其他编程语言上吗？

Robin：由于已经解释了 ML 的角色（作为逻辑宿主），我想对于这个问题，我能回答的最接近的问题是：对于证明来说，其他语言能像 ML 那样好地作为逻辑宿主吗？我确信，如果它们有一个丰富、灵活的类型系统，而且还能处理高阶函数和命令式功能，那它们就可以做到。ML 很幸运地被一些成功的逻辑开发者选为宿主，而且这意味着人们会继续选择它。

对于想要成为 logics of proof 的好的宿主语言来说，为什么必须要有高阶函数？

Robin：你从定理到定理，用函数的形式实现了推理规则。因此，这是一阶类型。你的定理基本上都是命题，因此推理规则基本上都是从字符串到字符串的转换函数。我们所创造的这些东西叫做策略——你表达的是目标、命题，那个你想要证明的命题，而且你得到的是带有函数的子目标集，这些函数给出了这些子目标的证明，它们一起为这个目标提供了证明。因此，策略是二阶函数

有一些这样的策略。我们把它们编写成程序，然后我们想要把它们联系在一起以获得更大的策略。这样，我们就有了三阶函数，我们把它叫做策略，它会使用策略，并产生更大的策略。这样的一个策略或许可以描述成："首先，去除所有的蕴涵，并把它们放进了假定列表中，然后应用归纳规则，最后再应用简化规则。"这是一个相当复杂的策略，我们把它叫做战略，而且这会推翻一些定理。

我在斯坦福大学为 John McCarthy 工作时，我说，"你看，我已经做了一些很好的东西，我理解了这个战略和这个策略，而且这儿还有它所证明的字符串属性。我仅仅是表达了这个目标，这是关于字符串的事实，然后，我使用了我的策略，同时它产生了作为定理的断言。"随后，他跟我说："你的策略通用性如何？它还能应用于什么场合？我有一个想法：对于我想证明的这个特定的东西，它的表现如何？"他提供了另一个例证。背地里，我已经用相同的策略证明了第二个例子，但是我并没有告诉他，因此，我们对它应用了策略，并且可以很确定地说它运行得很好。对此，他什么也没有说，因为这就是他表示赞同的方式。我能够展示我们有多态策略，它们能够做的不仅仅只有一件事。

您还提出了一种理论框架，用于分析并行系统、通信系统演算（Calculus of Communicating Systems，CCS），还有它的后继者 pi 演算。这会有助于我们研究和改进如何处理现代硬件和软件中的并行问题吗？

Robin：1971 年到 1972 年间，我在斯坦福大学的 McCarthy 人工智能实验室工作时，我开始考虑通信系统。它让我感到十分困扰，在这个领域几乎没有已有的语言能够很好地处理它们。我主要是寻求一种数学理论，语言能够把这种理论当作语义来用——这就需要对某些东西进行模块化；你应该能把小规模的通信系统组合成一个（并行）通信系统。

当时，Carl Adam Petri 已经创建了一个很好的模型：Petri 网，它能能够很好地处理因果关系（因果性）；Carl Hewitt 还创建了 Actor 模型。Petri 网并不是模块化的，而且我想要的是比 Hewitt 的模型更好的关于自动机的并行理论；同时，我还想把同步通信（握手）的概念作为基础。另外还有自动机理论，把它的语义作为形式语言（符号字符串集合），但它并没有处理好非确定性和相互作用。因此，CCS 是我的一次尝试。

最让我兴奋的是处理代数—首先是用于静态，然后是动态。为了把这个做得更好，我们花了很多年的时间，包括 David Park 推动的一大步：引入基于最大固定小数点的互模拟性概念。最开始，我想要模拟能够重新配

置状态的系统；例如，A 和 B 可能无法通信，直到有了连接着 A 和 B 的 C，C 把 B 的地址发送给 A。一开始，在与 Mogen Nieslen 讨论时，我们（在数学上）失败了；然后 Mogens 加入了一些正确的东西，这些东西我们以前并没有恰当的考虑；这就促成 Joachim Parrow 和 David Walker 共同开发出了 pi 演算。

令人兴奋的是，pi 不仅能处理重新配置，而且还能表示数据类型，而不需要任何额外条件。因此，它看起来好像是移动并行系统的基本演算，非常像顺序系统中的 lambda（λ）演算方法。

Pi 似乎用得非常多，甚至可以用于生物系统。不过，更重要的是，它提出了一种全新的演算方式，能够更直接地模拟分布式系统，而且具有处理灵活性和随机行为的功能。我们不是草草地完成并行处理理论，而是好像解决了一个非常有用的尚未解决的复杂问题。

207

您在设计和构建某个系统之前，是否科学地了解了这个系统？

Robin：对于这个问题我想了很多，它关系到后来的普适计算，但我认为它是普遍的。你必须有一些系统的工作模型。最起码你得有冯诺依曼计算机，这是一个科学的或者形式的或者严格的模型，事实上，正是它促成了 FORTRAN 和一系列顺序语言的诞生。它是一个科学的模型。这个模型非常简单，这正是它的妙处所在。

你需要一个模型，编程语言就是从模型抽取出来或者实现定义的。对于普通系统来说，这个模型可能与冯·诺依曼计算机有很大的不同；一般来说，它必须处理一系列相互作用和四处活动的智能代理和传感器群体等。

这听起来好像是您在谈论一系列表达语义的元语言。

Robin：我不喜欢使用“语言”这个词，直到我们拥有一个模型为止。当然，这与一直以来的实际情况完全相反；这些语言被定义，而且使用元语言来定义，而且通常是在有一个很好的模型之前来定义，当然，除了这种元语言事实上提供了这个模型以外。或许在这种情况下，元语言就是这个模型的同义词。我们在定义 Standard ML 语言时就使用了元语言。我们使用元语言，就是指明了应该接受什么指令以及它们应该做什么样的归纳推理。我想通用模型就应该是这样。我同意你的观点：我是在谈论一系列或者一种不出名的元语言。每一种元语言都专门用于特定的系统，我们可以把它叫做程序。

在这个意义上，计算是不是模型在若干不同层面上的定义和形式化，它允许您在更高的层面上来构建模型？

Robin：是的。我对普适计算非常关注，因为有相当多的概念，你想通过一个特定系统的行为来反映这些概念，但是你不能直接将这些概念全都包含在一个模型当中。我已经说过模型之塔，在这个塔的底部或许只有一些相当简单的机器。随着你不断的往上走，你会发现更多有趣的或者更人性化的或者更加深奥的概念，比如说失效管理、自我意识、信任以及安全等。不管怎样，人们都想要以层次的方式来构建模型，因此，你在每个模型中讨论的都是一个很好处理的概念集，然后你在一个相对低级的模型上面实现它们。

Lisp 和 Forth 经常讨论从可重用的内涵概念中提取和构建系统。在某种意义上来说，你是开发了一个“富”语言来解决你的问题。

Robin：因为我考虑的是一大堆模型，在较低级别上，你会有一些被称为程序的东西。在较高级别上，你会有详细的规范或者描述：什么可以做、什么不可以做，或者什么应该发生、什么不应该发生。它们的形式可以多种多样，可以是逻辑形式，甚至可以是自然语言形式。当你处于较低级别时，你会得到称为程序的那些熟悉的东西，而且它们只被看成是特定的模型而已。

它们是对“计算模型本质上应该是程序”这种观点的一种安慰吗？

Robin：是的。我想，当你接触到更动态的显式模型时，它们就是程序。你可以有一种规范模型。它可能由逻辑谓词组成；这是一个模型，它不是一个非常动态的模型，不过你可以使用谓词公式来表示前置条件和后置条件，你可以通过是否可以逻辑验证来评估实现的可靠性。你从一个并不是明显的动态规范或者模型，转到了更加动态的规范或者模型。这非常有趣。我认为，对于从动态的低层到描述性的高层的转变，我并不是非常了解，但它确实发生了。

作为备选方案，比如说在抽象解释这个方法中，你仍然有一个高级别的动态模型，不过它只可用于某些数据抽象。它并不是真正的程序，不过是动态的。这就是法国人在校验欧洲空中客车嵌入式软件所使用的方法。至于说这个模型什么时候是动态的，什么时候又仅仅是描述性的，这是一个有趣而复杂的问题。

或许，这种转换出现在我们必须承认物理定律时，比如，与非（NAND）门的行为。我们都了解这些物理过程，但是应该存在某个“点”，我们在这个点上建立的模型应该能包含这个级别。

Robin：是的。每台电脑都有很多电路图，这些电路图上画的都是电子器件，然后，在它们之上，你使用的是汇编代码，而且也不再谈论电子器件了。但是随着你先上移动，你在程序中仍然保留了动态元素，就好像它被转换成电路图中的动态元件一样。实际上，你似乎可以穿越不同的动态概念，同时仍然保持着动态性，但是随着你继续往上，它们本质上已经变得很不相同了。

通常，逻辑模型也有动态元素。例如，所谓的模态逻辑是就可能的世界而言来定义的，而且从一个世界转移到了另一个世界。你有一个动态元素，但是它已经被轻微地掩盖了。

我可以想象到人们反对在底层上的错误或者不可判定性元素，它们或许会影响上层的可计算性。

Robin：这看起来好像是无可争辩的事实。你可能无法降低底层的不可判定性，但是在高层次上可以用类型检查来完成。类型是一个抽象的模型，而且在类型上你或许可以具有可判定性，因为它是一种弱抽象，而且你不必用那些导致不可判定的组成成分。当然，它们仅仅是谈论了程序的某一个方面，因此，随着你以细节为代价逐渐向上时，你仅仅是获得了可判定性。

我从来没有那样想过。

Robin：我也没有想过很多。在类型检查方面，你确实有类型系统，该类型系统可判定；程序是否类型良好，和它有什么类型，然后你对它做了小幅修改，结果它就变得不可判定了。你只是为你的类型系统添加了很少一点细节。我们在 ML 中使用的类型系统出现过这种情况；该类型系统基本上是可判定的，但是如果你加进了所谓的连接类型之后，它就变得不可判定了。

什么是有用的和有什么是完全可控的，这二者之间存在一种平衡。很多时候，即使你不能经常检查某些东西来反对连接类型系统，通过一个足够智能的定理证明程序，你也可以非常成功。

通过我所说的模型塔的概念，可以把所有的一切都表达清楚。随着你向上层走，你会失去更多的信息。你或许会获得一些分析能力，而且它可能是颇有价值的，因为你正在分析程序属性，即使你不了解整个故事情况，知道这些属性对你来说也是很有用的。

我听说你可以用其他方式来做。在上层的一个模式的表达式意味着，当你可以证明某个特定的条件不会发生时，可以移除绝大部分来自底层的不可判定的东西。

Robin：噢，是这样的。较低级别由一个基本上不可判定的模型组成，但是，在对你所考虑的元素的某种约束下，它就会变得可以判定了。

不可判定性是不是不像听起来那么普遍？

Robin：是的，为了了解模型为你做了什么和它们是如何影响不可判定性的，这是一个很有趣的话题。我认为它是一个很好的话题。

作为信息专家、计算机科学家或者有经验的程序员，我们应该如何向那些仅仅是想要完成任务的人，教授包括定理、可证明性和类型的概念呢？

Robin：如果太早在学位课程中就教授这些，就可能是致命的错误。我们反对这么干，在数学领域也是这样。你所做的这些事情，以后会用一种更加抽象的方式来完成它，不过你如果过早用一种更加具体的方式来做，那么人们就只能模糊地理解它们。你学会了欧几里得几何学，而对其他几乎所有的几何学都一概不知。后来，或许是在大学二年级，你才开始了解其他的几何学，这种级别的抽象对于大多数 17 岁的学生来说是不可行的。在任何假设的情况下开展工作是很不明智的，其中有相当多的假设会认为它是值得教授的。

我知道我在学位课上也犯过错误：我试图教他们一些东西，而这些东西对于大学四年级学生来说也是太抽象了。甚至可以说在这个级别上，很多计算理论也还是太抽象了。我们必须学会适应和接受某些东西。麻烦在于，为了恰当地理解这个学科，而不用通过这些抽象，你必须要有一种层次性的理解概念。有些人永远不愿意谈论这些抽象。而其他人则可能会喜欢它们，而你们所需要做的就是确保他们能聚在一起谈论某些事情。

这会使程序员的适用性受到限制吗？我们能希望有 20%的人会对理论感兴趣吗？

Robin：他们不必必须理解理论，这是合情合理的。语言是一种工具，而且存在各种各样的工具。模型检查是人们为避免了解过多细节而采用的一种工具。如果有些人确实理解并知道模型检查是一种好的工具，那么这个办法就很棒。基本上，我们似乎在学科方面有了很多工具，这些工具仅仅是为了让人们不必去理解某些特定的东西，因为他们有更好的事情要做。他们有更大、更急的事情要做，而这恰恰就是高级编程语言萌芽之处。我喜欢某些理论的原因实际上是你可以从中提炼出一种编程语言。

我正在为普适计算建立一种图形模型。这是一种描述机制，可能对于大多数人来说，它是很难理解的，但是你可以从中提炼出一种语言来，我认为这很容易理解。当你提炼出一种语言时，你是在使用某种隐喻：有时他们是特殊的隐喻，有时他们是结构限制等。因此，从抽象模型到编程语言，这一步是一种非常舒服的步骤，它保护了人们免受一些事情的困扰。例如，类型系统保护了你不必知道大部分情况下你不想知道的那些事情。这不正是问题的本质吗：我们顺着模型塔往上，会获得越来越多的抽象，而且每个人都准备向上（或者向下）移动一定的距离而不再变动？

随着你顺着模型塔往上爬，你不必获得更多的抽象，但是你可能会受到更多的限制。消息序列图模型就是一个很好的例子：这个图描述了消息传递并发行为的有限片段，还有什么可以发生、什么不可以发生。对我来说，它就好像是一种限制性模型，它已经准备好被转换成一种更复杂的模型，它会处理递归循环和那些你不愿去想的各种讨厌的事情，比如说竞争条件等。消息序列图模型的一个好处就是你的执行者可以理解它，因此当你顺着模型塔往上时，不仅是你的抽象可以使得它更容易从理论上理解，而且你还可以在某种程度上限

定该模型更易于让不太专业的人来访问。

在某些方面，它更为通用；而在另外一些方面，它却更加专门化。

Robin：没错，确实如此。我认为这是一个难题。你可能想要把更加专门化的东西放在更底层而非高层，不过，我曾经把它放在高层。主要问题在于，它不仅很难，而且它还起到以通用性的代价，使更多的人们能够访问你的东西。这件事好像值得一做。

如果某个模型是定理的集合，而这些定理则是在更基本的原理上构建起来的，它会对你使用这个特定模型来表达想法有何影响？

Robin：我有一个例子，希望它不是太牵强附会，但是它确实是发生在我工作的模型上。你使用一个移动系统的模型，系统中有信息的移动，传感器和执行器的移动，这些东西在普适计算中屡见不鲜。你可以建立一个模型，因此你就可以对它有很大的发言权。你可以表达不变量，因此就会不出现这样的状况：同一个房间里挤了 15 个人以上，或者诸如此类的事情。但是在模型的某个版本中，你不能追踪特定的某个人，并且说："我的房间里从来没有这个人。"

这个例子看起来好像很不靠谱，不过它却非常简单、真实：在那个模型中，没用任何东西能够通过各种事件和重新配置来追踪个人的身份标志。你甚至都说不清楚这个问题："这个人曾经在这个房间出现过吗？"因为你不知道该如何说："这个"人："这个"暗示着任何时刻都是一种标志，特别是它与动词的某种时态连用在一起时。

这个模型的例子，其中有些事情你无法表达，而我对此却非常感兴趣，因为对于某种用途来说，这好像是一种优势。把这种模型应用于生物系统有很大的优势，在生物系统中人们讨论的是数百万的分子，而且并不关注一个一个的分子；仅仅关注 15 分钟之内会有多少分子等等。这种模型不需要表达特定分子的标识符，这对生物学来说非常重要。

标识符不像随机描述那样重要吗？

Robin：在这个特定的例子中，它不需要用于多种目的。我已经对生物学和普适系统做了一个类比，在其中，人们或者某种类型的智能代理会在城市或者可控的环境中移动。在后面的这种情况下，你更可能想要谈论一下特定个体的身份标识符。你可能会说：他从来没有在这个作为犯罪现场的酒吧出现过，因此不论何时你都知道"他"指的是什么。

再说说普适计算。在模型中，你也许会以完全离散的方式来谈论空间，因此就不会说任何关于距离的东西了。你只会谈论那些彼此邻接或者内部相互嵌套的实体。可能不希望用模型模拟空间的连续性，因此，把这些全都忽略了。它似乎有很多功能，模型可能很愿意去实现这些功能，而没有什么特别的目的，而且它还需要为了其他目的来完善自己。

假设我编写了一个 API。我的设计选择得越好，就越容易表达和理解模型。同样，我们也会更好地利用这个系统而不是滥用。

Robin：这个很可能发生，无论你考虑的是什么样的系统族。你可能会说到安全性系统，而且在某个特定模型中，你可能无法表达安全性的某些方面，但是你可以讨论其他方面。或者模型可能既不能表示安全性属性，也不能够表达任何有关保密性的东西。比如说真实性、保密性、安全性和防危性等这些属性都有细微的不同。

大部分模型只能表达其中的一些属性。

212

再次借用生物学来比喻，我无须理解细胞中 ATP 的转换过程，我只要知道怎么去抱起我的小猫并且宠爱它，但是这些细胞不知道任何关于猫的事情。直到它们被关注为止，它们只是另外一群细胞而已。

Robin：在这种方式上，我们与自然科学并没有什么明显的区别。你可以对运动物体进行牛顿力学分析，但是你得忽略摩擦力。

我们是否处于一个可证明的复兴时期？

Robin：我认为，只要人们通过显式语义继续设计语言，它就会继续下去。这种语义不会有任何用处，除非你可以用某些理论来证明它，比如说“没有程序会做这种蠢事”，或者是更正面一些的事情，比如说“程序将坚持使用多词学名的内存。”

这可能是人们设计产生那些证明的新语言的一个动力。

Robin：希望如此。人们在语言设计时，通常没有特别地把这放在心上。我想改变这种状况，不过这就是设计语言的方式，而且通常是由混合的委员会来设计的。其中，有一些语言设计很优秀，但是他们脑子里通常没有证明的概念。你会遇到像 Pascal 中关于变量记录那样的缺陷，它们违反了类型原则。像这样没有理论的语言是很常见的。

我认为我们很可能正在进步，因为我们确实看到了语言证明，而且并没有花什么大力气就实现了，因为证明系统的效率很高，因此这会提供从证明人员到设计者的反馈。这样就太好了。

9.2 意义理论（译注1）

The Theory of Meaning

在设计语言和用这种语言设计软件之间有什么样的联系？

Robin：这个问题非常大。很久以前，我认为在 20 世纪 60 年代到 70 年代，那时希望有一个面向通用计算机的语言（UNiversal Computer Oriented Language，UNCOL），这从来没有出现过。总的想法是：对于任何东西你都可以用这种 UNCOL 语言。如果存在这样一种语言的话，那么在语言设计和程序设计之间就不会有什么有意义的联系了。

现在（以及在此之前）我们已经看到了语言的进步，几乎所有的语言都倾向于某些特定的应用程序。Prolog 倾向的应用程序，其中的行为可以方便地使用逻辑公式来描述，因此，Prolog 语言的设计是用逻辑术语来构建的。ML 和 Haskell 使用富类型结构，因此 ML 和 Haskell 程序的设计通常是与类型结构紧密连接在一起的。这样的例子还有很多。每一项任务都可以用很多不同的语言来编写，而它们的结构在程序员脑子中可能是一样的，不过，每种语言都会擅长于明确表达结构的某些部分，而其他部分则并不擅长；至于说具体擅长于明确表达哪些部分，则会因各种语言而异。

译注1：意义理论（theory of meaning）是语言哲学的中心问题之一，字面上它与语义学（semantics）相似，但习惯上把对意义的哲学研究称作意义理论，把对意义的语言学研究称作语义学。

我想，在面临一个特定的任务时，程序员会选择那些能够明确地描述他认为最重要的任务的语言。不过，一些语言能够做得更好：它们实际上可以影响程序员思考任务的方式。面向对象语言在这一点上就表现得非常好，因为对象的概念有助于人们理清不同的应用思路。

213

您认为除了面向对象以外的范例会影响程序员设计和思考的方式吗？

Robin：是的，我认为逻辑编程和函数式编程会有这种影响力。我希望处理演算的范例有这种影响力。Lotos语言肯定就有这种影响，它是一种规范语言。而且，我认为通过 ALT 命令的概念，Ada 语言也是如此，等等。

每个程序员都会使用他们自己的编程语言而不是选择一种语言来完成各项任务？我们会集聚到少数语言系中吗？

Robin：如果程序员自己的的语言不受公认的理论约束的话，他们使用这种语言就会造成混乱。毕竟，不用公认的理论术语，他的语言意思如何定义呢？一旦存在一种理论，那么程序员就可以创造用该理论解释的句法短语了。因此，他就可以使用“自己的”句法，但是得用那个理论来解释意思，而他描述他的语言时，也要明确提及这一理论。这没有什么错误。不过，因为理论是隐含在这些语言之后，人们希望它们会有很多共同点。

您如何来定义设计编程语言的观点？它是一种表达想法的工具，还是表达目标的工具呢？

Robin：如果你考虑一下函数式编程还有逻辑编程的好例子，那就已经有了一个理论：对于函数式编程来说是函数理论、类型和值理论；而对于逻辑编程来说，则是成熟的一阶逻辑理论。在语言出现之前这个理论就存在了，而且这个语言多多少少是基于这个理论的，因此还是有理论先于语言的例子的。同时，我认为我们需要更多这样的例子；我不知道我们到底需要多少不同的例子。

我们可能会说，想要实现的目标就是设计一种语言的基本原则。

Robin：你可能需要用一种不同的语言或者理论工具来表达目标或者程序行为属性。例如，你可能想要使用某类逻辑来编写规范，而编程语言则应该是某种代数性更强的语言，不过代数和逻辑这二者应该是已经令人满意地联系在一起了，甚至是在你把它们的某些部分设计成编程语言之前，它们就联系在一起了。

我认为，你用来表达目标和所需属性的那些工具，不一定非要与表达程序的工具相同，但是它们应该在某种理论上联系着，这种理论的存在或许并非仅仅是用来产生程序的，甚至是用来理解自然现象的，比如说我所提到的生物学例子。它看起来好像是如果我们能够理解信息，我们就可以从信息的角度去理解自然系统，而自然科学家们就是这么做的。但是或许我们可以使用相同的形式体系、相同的数学结构和属性来定义语言，由此产生了并非是自然现象的人工制品。因此，我看不出来为什么自然系统的信息描述要与程序系统或者软件系统的信息描述分离开来。

214

假设今天您发现了一个系统 bug，而这个系统是在 5 年前做的。你也有与实现同步的规范文档。如果在语言设计上有 bug 呢？如果是一个缺陷导致了某种特定的错误呢？

Robin：我可以很高兴地说，没有出现过这种事情，而且我不知道该怎么做。我想我们很有可能会说你已经接受并适应它了。我们会发布一些东西来澄清：“是的，这里出了点问题，不过，如果你能这么做、那么做，你就不必担心。”这些定义都是相当敏感的。我的意思是，有些人在努力降低定义机制的敏感性、提高它的

模块化，我认为这确实非常困难。我不知道这是怎么完成的。现在，我倾向于不去改变它，而是简单地告诉人们那里有一个问题。这只是一种实用的办法，而不至于使你抓狂。

假定一种语言带有富类型系统，比如说 ML 或者 Haskell，这种类型系统有什么样的想法能使得用这些语言编写和设计的程序变得明确？

Robin：如果他们的系统通过了编译器，也就是说通过了类型检查，然后，就不会出现某类特定的事情了。他们肯定知道，肯定不会再出现某种类型的运行时错误了。他们不会知道你无法获得数组溢出，而且他们也不会知道可能会出现其他令人讨厌的事情，比如说无限循环等，不过，他们知道的已经相当多了。

对于我们的应用程序，数学定理证明如果能够说："如果你认为你证明了 ML 中的一个定理，而且你认为你在 ML 语言中的推理规则的逻辑描述已经圆满完成了，而且 ML 程序有了一种定理证明，那么这个定理肯定就是被证明了的"，这是很了不起的。这得益于抽象类型机制，该机制允许你把定理类型解释为只能用推理规则来操作的某种东西。无论你想为可能的推理搜索多么聪明的技巧，只要你搜索到的可能推理的一个序列成功了，那么你就必须实际完成这个推理，而且你必须在这种定理的类型上实现推理。你知道你唯一可以做的就是这个合法的推理。或许，你永远搜索不着某种可能的推理序列，但是如果成功了，你就得实现这个推理，或者系统已经为你完成了这项工作。在验证实现和语言设计的可靠性时，你会知道这确实是一个定理。如果使用的是类型语言，就不必如此了。

您仅仅有一个操作的集合。

Robin：系统会说，"我已经有了一种理论"，而你会说，"我如何确信它呢？"这才是真正严重的问题。我记得在斯坦福大学的早期时光，那时我们在设计第 1 版，实际上，甚至还不是 ML。当时，我们正在研究自动推理系统，而且相信能使它正确地自动化，而且唯一能被推理的就是可用这种推理规则的东西。记得有一个午夜我正在思考，我脑海中出现了一个东西，还有一个定理从我脑中一划而过，并在说："我是一个定理"，我不必担心，因为我相信类型、相信实现。我相信它到了这种程度：即使我在终端做着疯狂的事情，都不会影响到系统的健壮性。

我认为，这真是一个非常强大的功能，而且 Isabel 和 HOL 这样的系统，还有现有的所有系统一直以来都有这种功能。在这方面，这是一种令人惊讶的真正解放，因为在此你无须担心。

然后的问题是：我们如何说服计算机来告诉我们程序是什么意思？

Robin：某个特定的程序可能会这样：如果你对我做了这个，那么就会发生这个；如果你对我做了那个，那么就会发生那个。类型允许你对此做出严格的声明。这就是计算机可以帮你的地方，编译器通过类型检查器来帮你。当然，这不必是可判定的类型检查器，而应该是这样的：如果它断定程序是类型良好的，那么它肯定就是类型良好的，但是它不能断定某些程序的任何东西。你可以有不可判定性的富类型系统，但是你仍然有所谓的积极安全性：如果该定理仍然成功的话，那么这就是一个定理。

您对于那些想成为更好的软件设计者有什么建议？

Robin：这得取决于他们是想赚钱，还是想要做科学研究。你不能建议别人应该做什么，不过，有很多种方式可以赚钱，而且不需要做科学研究，反之亦然。

如果我对做科学研究的那些人提些建议，那么我想告诉那些做设计的人：不要闭门造车地设计那些看起来很

漂亮的理论，而是要确保设计跟实际有更多关联。

您把千年虫问题描述成以下情况的一个好例子：我们不知道将会面对什么样的问题。在设计阶段，我们应该如何避免这样的结构性问题呢？

Robin：我不知道。市场对软件产品非常渴求，如果你花点时间来分析你要出售的产品，那么就会有人要来签合同。这听起来很令人怀疑，不过我认为这确实是事实。如果你面对现实世界，如果你尝试使用分析工具，实际上你不会取得成功，即使确实存在那样的分析工具。当然，更通常的情况是，根本没有那样的分析工具。

在面对千年虫的问题上，如果我们以一种合适的方式来编写程序的话，我们拥有所有需要的理论来避免千年虫问题；所有要做的就是小心使用已经存在了 20 年的类型理论。当然，对于为什么忽略了这个理论，人们有很多猜测，但我认为主要的原因在于市场的力量。

216

或许文档会有帮助，开发者应该如何编写文档呢？

Robin：嗯，他们应该在代码中加上注释，不过这也得要有某种严格的基础。我相信，适当地标记注释的难度会随着代码的规模呈现非线性增长，如果你面对的是一百万行代码，那么只用几千行的代码标记注释是远远不够的。事情变得更加困难了；不同部分之间交互复杂性的增加要远远超出线性方式，因此，对于你所谓的真正的程序最好是有一个严格的规范说明。

顺便说一句，我使用推理规则的形式为 ML 写下了整个形式规范。我们放弃了语言的形式定义。我们没有写下实现和形式规范之间的相关程度，但是因为我们有了形式规范，因此非常清楚知道在尝试实现什么。首先，我们有一个非常清晰的规范，其次，也没有想过要去修改它，至少是极其缓慢地对它进行修改。

现在，在现实的程序中，它们必须能够修改或者删除某些部分、调整某些部分、引入其他部分，因为文档规范将会修改。因此，对于在现实生活中的真正程序，处理规范和实现之间的关系时要格外小心，因为你是要修改规范，而且你还想要确切地知道这种修改在实现中意味着什么。

在开发 ML 语言的过程中您有什么逸闻趣事吗？

Robin：有，其中一件趣事就是我们花了更多的时间来讨论句法而不是语义。在很大程度上，我们在语言的功能理解方面都能达成一致，但是很多情况下，比如说要在句法中使用哪个词等之类的事情，我们肯定会争论一番，因为要做出这种决定并没有什么科学的基础。

还有一件趣事：我们为 ML 提供了一个非常完善的类型系统，与已经存在的类型理论相比，这个系统是最合适的。我们使用 ML 的目的是用来使用数学逻辑来做形式证明，而且，我们想做的一件事情就是有效地实现所谓的“简化规则”。当你把复杂的表达式转换成为某种特定形式时，有时叫做正规形式或者正则形式，进行这种转换需要遵从很多规则。为了很快地做这些转换，你必须非常聪明地实现它们，因此你可以检查可能应用的所有规则，并以某种方式对它们进行同步匹配。

我们在 ML 中实现了同步匹配，而且发现有些东西效率不高，结果证明是因为类型系统有一些限制。对于这个小一点的特定实现（实际上，这是一种分析工具和一个定理证明器的实现），我们决定停止在 ML 中使用这个规则，而且使得它效率更高。事实上，它也并没有那么糟，因为我们使用过的更普遍的类型系统虽然也很容易理解，但是它更为复杂；我们想让 ML 有一个简单的类型系统，而且为了一些要高效完成的工作，需要稍微普遍、但也稍微复杂的类型系统。

217

如果您今天重新开始设计 ML，计算方面的进步或者你的理解的提高会显著地改变设计吗，或者是它的结果还和以前差不多？

Robin：我们设计它是为了理论证明。结果表明，定理证明是一种很有需求的任务，这使得它成为了一种通用语言。我不禁要问："现在我应该设计它来做什么呢？"如果我还是设计它来做理论证明，那么就会有相同的问题。你想要情况会有所改变的一些东西。你不想要纯函数语言，因为你想要经常改变它的状态；你想要控制推理树、目标树和子目标树，你想要控制你所产生的一切。你想要改变它们。

从那块天地里走出来处理更加明确的动态系统，比如说普适计算，如果我使用的是函数式语言，我就会感觉像是迷失了方向。如果我是为定理证明设计语言，那么或许是 Haskell 中用来处理序列的工具 monad，或者一个更好的主意，不过我不确定。我得非常清楚我是为了什么目的而设计它的。

令我感到困惑的是：人们没有优先的应用领域，也没有容易做的具体东西，他们是如何设计语言的。对此，Java 有个很好的主意，而且结果证实它是一种优秀的语言。不过，现在可能的应用领域空间相当庞大。这就是你会得到许多不同的语言的原因所在，这些语言用于不同的目的。其实，目的是一个缺失的参数。如果我为了同样的目的去设计它，我就很有可能得到相同的语言。

我已经看过您的工作了，而且它看起来好像是您使用了这种方式："在这个问题上，我想创建一个可重用的原语集（我的定理），然后在这个基础上构建一些其他的定理"。

Robin：我认为你可以使用 ML，即使你头脑里没有多少定理，不过，或许你说的是一个设计者而非使用者。毫无疑问，当我们设计东西而且使用我们做的运算结构和语义时，我们想的是要证明关于整个语言的某种定理，例如：这将不会有虚引用（译注2）。我们希望有很多事情是对的，事实上，这后来都被自动或者半自动证明系统所证明。这让我倍感轻松！

我们非正式地知道不会出现任何虚引用，不过为了确保你不会犯这种低级错误，有一个正式的事实证明是很好的。另一方面，我们确实在引用类型、赋值变量类型等问题上遇到了麻烦。有几个人向我们展示了在类型系统中这些情况确实是存在的。如果我们在语言中强加进某种约束，那么我们就不会有这样的麻烦了。一项调查表明，只有 3%的应用程序会受到这个约束的影响！如果我们对另外的 97%满意的话，我们就可以避免这种麻烦。我们修订了语言，强加了这个约束，这促使我们在 1997 年进行修订，而不是 1990 年的修订，那时候我们的第 1 版语义才刚刚出炉。

在您修订语言时，正式修订是不是保持实现和规范文档一致的唯一方法？

Robin：我想我们已经保持了它们的同步。我们能够证明它是向上兼容的。换句话说，只要旧的实现稍微遵循严格的程序形式，那么它就没有问题。在修订中，向上兼容是一个真正的问题。

事实上，我更希望我们没有做这次修订，不过，把它改对也是一件非常吸引人的事情，而且也有人建议让我们做一些更加简单的事情，这同样也很诱人。出现这种情况，是因为我们在这次修订过程中，在其他方面也做了很多改进。修订工作比预想的更加费力。大体上，我们并无须做这次修订。但是总的说来，我很高兴做了这次修订，因为类型系统所暴露出来的问题是很有价值的，而且我们能够把有些事情做得更简单。

译注2：虚引用（dangling reference），是一种指针，它指向存储器中不再存在的对象。虽然这个对象可能已被正确删除，但引用的另一副本还存在。如果曾用该副本访问过该对象，那么在原引用释放的空间中所创造的某些其他对象可能会遭到破坏。

9.3 超越信息学

在今天的计算机科学中，主要的问题是什么？

Robin：最近，我在研究模型结构的概念。如果你工作使用的是高级编程语言，那么这就要用低级语言中的实体来表达了，然后用汇编代码来表达，而汇编代码的行为则是用逻辑图来表达的，因此，你会有一个模型，这个模型不再是软件模型，而是一个电子对象模型。随后，解释这个人工制品的东西是计算机，它最终会去运行你的程序，这就是模型层次结构中大概的四个层次。故事到这里并没有结束，因为你可以走向更高的层次，从编程语言走到规约语言（译注3），在某种程度上这是更高级别的模型，因此，你已经有了五个层次。

想一想普适计算，这种系统将会管理你的家庭购物计划，并为你装满冰箱，或者通过随附在个体身体甚至是在他们身体中“旅行”来监测个人的健康状况。为了理解这些系统，你需要很多层次的模型，因为人们谈论的软件智能代理，这些代理会与其他代理相互协商、相互请求资源、相互信任并反映到它们自己的行为当中。换句话说，展示了各种半人类的属性。在这些系统中，有些行为会用一种非常高级别的逻辑来解释，这种逻辑需要处理信任、知识、信念等。因此，你需要一种关于这些逻辑的理论，用更基本的行为来阐述如何详述程序。这些规范文档很正式，它们与程序的运行行为有关。但是，在高级别上，你会问这样的问题：“这个程序信任那个程序，这是真的吗？”；或者“你是如何实现计算智能代理之间互相信任的？”；或者“一个智能代理可以以某种方式被告知：应该相信其他代理希望要做的事，或者是相信其他代理对它自己的目标来说是一种威胁。你是如何理解这种方式的？”

这些问题的存在要比正常的普通程序规范高三个层次。无论这些模型是什么，它们塑造了软件，或者塑造了其他能够解释更低层次的软件的模型。除此之外，如果您想建立欧洲空中客车之类的东西，您就得将软件模型与飞机工作时的机电模型结合起来，甚至还要结合飞机飞行时须要满足的大气条件和天气模型。因此，我们必须迎接把模型结合起来的这一挑战，有些模型来自自然科学，比如气象模型、机电工程模型和软件模型等，而且这些模型是相互影响的。在这种组合模型的层面上，你才能预测空中客车的性能表现如何。

我喜欢自然和人工相结合的想法，我们也有适用于人工和自然建模的相同理念。只有在人工的情况下，模型是先于人工制品的。而在自然情况下，自然现象总是在建模完成之前发生的。这是信息学和自然科学之间的一种整体性。

您在设计软件时，可以进行物理性的测试。对于软件来说，有时候会在没有任何用户之前的软件实施过程中，发现软件设计的失败。我们该如何把设计、实施和实际使用中的不同步骤结合起来呢？

Robin：自然科学建立的模型，最终都是通过观察来验证的，通过观察现实世界是否按照模型预测的方式来运行。这意味着不能充分验证那些模型，通过发现观察到的和模型预测的有时候不一样，才能发现这些模型的描述是错误的。我们不可能观察到一切现象都如同预测的一样，因为工作是无穷无尽的。

译注3：规约语言（specification language，或称规范语言、描述语言），是在计算机科学领域的使用的一种形式语言。编程语言是用于系统实现的、可以直接运行的形式语言。与之不同，规约语言主要用于系统分析和设计的过程中。

我们有一种更好的情形，举个例子，根据低级语言实现了高级的编程语言。我们对每个行为都有一个形式描述，因此能够验证从高级程序到低级程序的翻译实现，这是通过观察对高层次和低层次行为的科学或理论的解释实现，高层次的行为同低层次的行为实际上是一致的。所以，我们有机会验证一种模型由另一种较低层次的模型实现的方式。只有我们开始实现较低层次的程序时，比如说实物，我们不能做这样的数学证明。但是在更高的层次上，只要模型得到了良好的表达，而且每个层次实体的含义都是这个模型在这个层次上的一部分时，我们就有机会进行数学证明。在每个层次上，你可以有实体，就能解释它们的行为。此时，它就是您通过较低层次的模型实现来验证较高层次的模型实现的媒介。

220

这就是我一直在试图说服人们思考的方式。例如，我最近的讲座之一叫做“普适计算：我们应该理解吗？”。我所说的“理解”是精确地指定这些系统之一的行为，比如说监测你身体行为的东西，我们也想了解智能代理是如何实际实现这些系统行为规范的。

我认为说服别人相信这个并不容易，他们往往会说这将是不可能的，因为系统会变得非常之大，大到你不可能这样做。事实上，我看欧洲报告说，没有人能够分析普适计算系统的行为。在我看来，这么说是完全错误的。这取决于我们设计的系统能否被分析，所以我们应该创建可供分析的系统。

您觉得工程和信息学之间有什么联系？

Robin：哦，工程往往会得到首先出现的自然科学的支持。自从化学理论经过了现实的设计和测试以后，化学工程的时代已经到来。化学工程的兴起源于科学化学发展的结果，科学化学研究的是自然现象。因此，你就能使用从自然科学得来的理解来进行工程设计。

我认为物理学也是这样，但是，对于软件来说，似乎有所不同。据我所见，不存在天然的软件。如果硬要说软件存在我们的脑子里或者什么，这更多的是一种夸张。我们还没有成熟的科学足以作为软件工程的基础。因此，我觉得工程和软件之间的联系，与其说是联系，还不如说是对比。我们并没有以被普遍认可的科学为基础的软件工程，而其他形式的工程大多是以被普遍认可的科学为基础的。

数学在计算机科学中起到什么作用？

Robin：数学的很多分支都得到了使用。我们使用逻辑、代数，还有概率论等。在混合系统中，离散行为的混合是很常见的现象，我们使用了微分学。因此，越来越多的不同数学分支似乎都可以发挥一些作用。我们所不清楚的是如何选出一部分。你选择它是因为你喜欢概率论或者随机论？或者是因为你预见到它能对计算机系统或者信息系统做出解释呢？

221

作为一名数学家，你可以选择喜欢学习的东西。对于我们来说，我认为你必须看看实际的事件系统，无论是自然的或人为的，然后问我需要用什么来解释它？

最近，我不得不搞懂随机分析，它是持续时间、占用时间的概率等。如果利用我们的模型来解释一些生物现象的话，随机分析是绝对必要的。为了理解这种理论的工作原理，我转而学习了以前从来没有学过的理论。我还学习了数学中比较抽象的分支，比如代数、范畴论等。通常你会发现，你只是使用其中的一部分，所以你不用去研究复杂的数学家们才研究的纯理论，你只需要四处找一些理论；这部分理论可能你已经很好地理解了，或者是所知甚少，因为它们不够完美，因此你可能最终会对纯理论做出贡献，即使您的目的是解释那些实实在在的事。

您认为自己是一名计算机科学家呢，还是研究者？

Robin：我不喜欢“计算机科学家”这个词，因为它过分地强调了计算机。我认为计算机只是信息行为的实例，因此我会说自己是“信息科学家”。当然，这要取决于你所说的信息学的含义。我个人倾向认为信息学是计算和通信行为，通信是非常重要的。

作为一名信息科学家，您的作用是什么？

Robin：我的作用是试图建立一个能分析的概念框架。要做到这一点，你必须考虑软件中实际发生的事情，比如说普适系统，但是你想用某种方式将它们抽象出来。这确实很难，你会犯错误，你会创建错误的概念，在某种意义上，它们不会被人接纳，它们不会扩展。您在寻找可以扩展的基本观念，以致于它们实际上可以用来解释现有的或拟建立的大型软件系统。

我觉得智能代理之间的通信是非常重要的，因为这是在计算理论中第一个被隔离出来的概念，连逻辑学家都没有研究过。互动式智能代理的结构化群体的想法：结构应该是什么？它们之间是怎么相互联系的？哪些可以跟其他的代理通信？它们能否创造出新结构？它们能否根据邻居的请求和行为而改变它们自己的行为？你遇到了一大堆问题，这只有你有智能代理群体时才能问出这样的问题，而不是在编程的早期阶段，那时只有支持单个任务的单个程序。

因特网能帮我们找到解决方法吗？

Robin：我认为因特网可能会带来一些问题。首先要理解它自身也是一个问题。如果我们创建概念性工具来分析因特网的行为，它可能会有助于我们，它们可能就是我们需要用来理解其他智能代理群体的工具，比如监测你身体的智能代理，或者控制高速公路交通的智能代理。

因特网上也出现了很多出色的设计。这是一种很好的可供学习的例子，因为它在实践中用得非常好，因此，我认为它既是解决方案的一部分，也是问题的一部分。

您如何看待今天的计算机科学研究领域？

Robin：我知道它传播得很广泛。那些构建大系统的人不会使用严格的方式来详述它们。也有人在较低级别或者在更为形式的级别上工作，如果你喜欢，他们有时候会吸收进纯数学，他们可能并不喜欢向现实妥协，因此你会看到各个群体的分歧。事实上，这里有一个范围，当你从应用程序的最前沿一直到背后的理论，你会发现人们沿着一条很长的路一路走来，他们发现很难理解人们是要靠左走还是靠右走。

我觉得它没有很好地连接起来。我们在英国设计了一个非常好的挑战（Grand Challenge），叫做“普适计算：实验、设计和科学（Ubiquitous computing: Experience, Design and Science）”。所谓“实验”，我们指的是人们通过某些环境的仪器所做的实验。他们可能用仪器监测一栋建筑物，每个房间放一台计算机，这台计算机可能会在来人时认出他们，或者报告他们的活动。因此你可以让人们做对比设计实验；你就会发现那些在给定场景中后台根据良好的工程原理进行实现的人；然后，在他们当中，你又会发现将更多的抽象模型与工程工作关联在一起的科学家，而工程师们则是使用他们构建的工具在现实世界进行实验。这三个层次——实验、设计与科学——尽量去弥合我说的那种隔阂。

我开始跟那些我通常不愿意讲话的人讲话，这些人考虑普适系统对社会的影响，而且他们确实把系统看成不是人在使用系统，而是人们本身就是系统的一个组成部分。你是在非常具体的层面上来了解这些系统的，你必须从工程原理来进行反驳，下至那些可能用作分析整个事情的基础的某些概念。

您认为现在的研究方式和 20 世纪 60、70 年代有什么区别吗？

Robin：噢，其实我对于普适系统更感兴趣，它们会更深地嵌入我们的环境当中，而且这是一个很大的区别。尤其是在机动车或其他关键机器中的实时关键软件。为了验证实时软件而采用的分析方法很不相同；计算和实时之间的关系通常很少有人关注。当然现在，工程师在构建空中客车或者任何普适系统时，非常关注实时发生的状况，就像是在物理中你说那些东西有多长一样。

223

普适计算能为人工智能（AI）带来改进或者重大突破吗？

Robin：是的，不过我认为我们应该间接地跟人工智能打交道。在 20 世纪 60 年代或 70 年代把重点放在人工智能上，我对此从来没有感到满意。在我看来，对它的某些希望言过其实了。

随着我们开始在理解智能代理群体时使用“信念”和“知识”等这些词，我们就开始把人工智能不是看成那些存在或不存在的东西，而是看成你逐渐去接触的那些东西。你的系统变得越来越智能，而且变成所谓的“反应式的”，也就是它们可以报告自己的活动并分析它们的行为，因此，随着我们开发更多的这类系统，被视为人工智能一部分的那些概念已经逐渐地从少量使用，到甚至是大量使用。

因此，我不能肯定人工智能中那些完成的工作是不是有用。我认为，在我们设计大型系统时，使用了更人性化的术语（比如说“信念”）来描述事件，那些工作将会更加有用。然后，智能和非智能之间的区别就变得很弱了，因为你有了度的概念。

您从那些并未应用的研究领域中学到了什么吗？

Robin：大多数编程语言在设计时没有事先考虑“意义”将以什么理论为基础。因此，经常会出现这样的情况：人们设计并实现了一种语言，而它想要干什么、当所有程序运行时它会发生什么都没有进行必要的预测。当然，它在某些情况下却预测得很好，例如，在 ALGOL60 中；1960 年的 ALGOL60 报告非常准确，以至于人们可以根据它来找出将要发生什么。但这只是个例。甚至是在优秀的语言中，规范的标准也不是在语言之前就设计好了的，因此，人们要做的就是后来改进语言的“意义理论”，而且这或许意味着设计不能够利用理论理解。

改进分析的其他例子是大型软件系统。在英国有很多例子；它们引起了一个严重的延时，有时甚至是一场灾难。大系统会从严格的规范和一些科学分析上获益良多。

您说语言有一种“理论意思”，您指的是什么？

Robin：这是一种“实现会干什么”的理论。ML 有一种意义理论，因为我能够从它正在使用的运算语义证明它没有虚引用。近来，人们已经表示所有关于 C 的语义，并创建了 C 的意义理论。你有了 C 语义，然后你使用这种语义证明了关于所有可能的 C 程序的理论。近来的确是有很多成功的例子。我认为，事情进展得非常顺利。

作为一种编程语言，你是指语言自身的规范和设计。这可以防止用户的一些语言错误吗？

224

Robin：可以，它是指可以检查用户错误是否违反了规范，然后，可以在程序运行或者实际使用之前发现一个错误匹配。

我现在从事交互智能代理群体行为学理论研究。它可以用来描述人们和机器在构建好的交流环境中如何相互沟通。我希望这个理论也能有助于理解生物系统，例如，一个细胞是如何产生新细胞，作为它表面的一种保护泡。

通用的信息科学可能不会倚赖于特定的应用程序。在你设计出一种编程语言之前，你希望能有一种理论来指导你进行设计。我想在一种语言稳定之前创建这种理论。

第 10 章 225

SQL

给定一个大量的结构化数据集合，在你不知道人们需要何种运算时，你如何能提供一种高效方式来收集、检索和更新信息呢？这就是 E. F. (Ted) Codd 创造的在关系模型之后的基本概念。SQL 是关系模型最明显的实现，这是一种用来描述想要什么而不是如何去做的声明式语言。Donald 以 Codd 的思想为基础，Chamberlin 和 Raymond Boyce 开发了 SQL。

226

10.1 一篇开创性的有重大影响的根本性的论文

SQL 是如何设计的？

Don Chamberlin：在 20 世纪 70 年代早期，集成数据库系统刚刚开始广泛使用。技术和经济上的趋势，使得企业第一次将它们的数据视为一种公共资源，在很多应用程序中间共享。这种考察数据的新观点为开发新一代数据管理技术提供了一个机会。

20 世纪 70 年代，IBM 的主流数据库产品是 IMS，不过除了 IMS 开发组之外，各地的若干 IBM 研究小组开始研究数据库问题。E. F. (Ted) Codd 博士是其中一个小组的领导者，这个小组位于美国加利福尼亚州圣何塞市 IBM 研究实验室。我和 Ray Boyce 是另一个研究小组的成员，我们小组位于纽约 Yorktown Heights 的 IBM Watson 研究中心。我和 Ray 研究数据库查询语言，试图找到改进当时通常使用的语言的方式。

1970 年 6 月，Ted Codd 发表了一篇开创性的有重大影响的根本性的论文（注 1），介绍了关系数据模型，并描述了它在数据独立性和应用程序开发方面的优点。Codd 的论文在 IBM 公司内外吸引了极大的关注。

1972 年，我和 Ray Boyce 参加了一个关系数据模型讨论会，这个讨论会由 Watson 研究中心的 Codd 组织。对 Ray 和我本人来说，这个讨论会让我们“转变了思路”。以关系形式存储数据的优雅和简单性，给我们留下了深刻印象，而且我们能够看到很多查询方式都易于以关系形式表示。在讨论会后，我和 Ray 参加了一个“查询比赛”，我们在比赛中互相挑战，看谁设计的语言灵活性足以支持多种方式的查询。

1973 年，Codd 的思想变得很突出，因此 IBM 决定在圣何塞市 Codd 处整合数据库研究力量，并开发一个工业强度级原型 System R，作为关系概念的一种证明。我和 Ray，连同来自 Yorktown 和剑桥的其他 IBM 研究者一起，转战加利福尼亚，加入 System R 团队。由于我和 Ray 对语言感兴趣，我们的第一个任务是去设计一种查询语言，用作 System R 的用户接口。我们研究过 Codd 和其他人提出的关系语言，并给自己设定了下列目标：

- 我们要设计的语言是基于普通的英文关键字，而且易于通过键盘输入。我们想要这种语言基于熟悉的概念，比如带行和列的表。像 Codd 最初的语言方案建议的那样，想要我们的语言是声明式语言而不是过程性语言。我们想要获得关系方式的能力，同时要避开一些数学概念和术语，比如通用数量词和关系除法操作符，这些都在 Codd 的早期论文中出现过。我们还想包括一些高级查询概念（比如分组），我们觉得这些概念用其他关系语言不容易表示。

227

- 除了查询之外，我们还希望这种语言能提供其他功能。最明显的扩展是包括插入、删除和更新数据操作。我们还想解决传统上由数据库管理员处理的任务，比如创建新表和视图、控制访问数据，以及定义约束条件和触发器，以维护数据库的完整性。我们想要在一个统一的句法框架下完成所有这些任务。想让授权用户能够执行运行管理的任务，比如定义新的数据视图而不会停止系统和调用特殊的工具。
- 换句话说，我们把查询、更新和数据库管理看作是一种语言的不同方面。在这些方面，我们有一个唯一的机会，因为我们的用户是从头开始创建关系数据库，而且不受向下兼容性的限制。

我们想要这种语言既能用作一个独立的决策支持查询语言，还可以用作更复杂的应用程序的开发语言。后一个目标需要我们发现将新语言与流行的各种应用程序编程语言连接起来的方式。部分地基于我们在“查询比赛”中的及更早一些的经验，我和 Ray 开发了一个关系查询语言初始方案，称为 SEQUEL（“结构化英文查

注 1：Codd, E. F. “A Relational Model of Data for Large Shared Data Banks（大型共享数据库数据的关系模型）” *Communications of the ACM*（ACM 通讯），1970 年 6 月。

询语言”的首字母缩写）（注 2），而且在 1974 年 5 月的美国计算机学会年会 SIGFIDET（SIGMOD 的前身）上发表了一篇 16 页的方案论文（该次会议还以 Ted Codd 和 Charles Bachman 之间的著名争论而著称）。在最初的论文发表后不久，Ray Boyce 突然因大脑动脉瘤黯然辞世。

在最初的 SEQUEL 方案发表后不久，该语言经历了验证和改善改进提炼精炼 menta 阶段，时间大约从 1974 年一直持续到 1979 年。在此期间，SEQUEL 一直作为 IBM 圣何塞研究实验室 System R 数据库项目实验的一部分。System R 研究了数据库管理的若干方面，包括 B 树索引、加入方法、基于成本的优化，以及事务一致性等。这个实现经验反馈到该语言的演进设计当中。SEQUEL 还受到三位 IBM 用户的反馈影响，他们安装了 System R 原型，并在实验基础上使用。System R 团队同用户团队一季度会面一次，讨论这种语言及其实现方式的改进。

在 System R 项目实施期间，SEQUEL 语言的发展非常迅速。该语言的名称从 SEQUEL 简化为 SQL，以避免商标侵权。还添加了最初的提案中缺少的一种通用连接设备。分组设备进行了改进，还将 HAVING 子句加进了过滤组。为了解决缺少信息空缺（missing information），语言中还添加了 null 值和三值逻辑。增加了一些谓词新品种，其中包括用于部分匹配的“Like”的谓词和用于测试非空子查询的“Exists”谓词。另外还发表了论文来记录语言的演变（注3）（注4）。在语言设计的这个阶段，决策往往是根据我们的实施经验和我们的实验用户的需求，建立在务实的基础之上的。

1979 年，随着 System R 项目的完成，IBM 的 SQL 研究阶段也宣告结束。此时，语言转由开发团队负责，他们要把 System R 原型转换成各种 IBM 平台上的商业软件产品。总之，1979 年发布了第一款基于 SQL 的商业产品，它不是由 IBM 公司发布，而是由一家叫做 Relational Software 的小公司发布的。产品名称为 Oracle，后来被采用为公司名，这时候它已经不再是一家小公司了。在 Oracle 的产品出现不久之后，IBM 就实现了 SQL，所有主流数据库厂商最终也都实现了 SQL。SQL 现在是世界上使用最广泛的数据库查询语言。

为了促进应用程序的在各个 SQL 实现之间的可移植性，美国国家标准学会（ANSI）开展了一个项目来制定一个 SQL 标准规范。结果，所谓的 Database Language SQL 在 1986 年作为 ANSI 标准颁布（注5），1987 年又成为了一个 ISO 标准（注6）。SQL 标准已经成为了持续发展的 SQL 的重点，而新的功能则被添加到语言当中，以满足不断变化的需求。国际标准化组织（ISO）分别在 1989 年、1992 年、1999 年、2003 年和 2006 年颁布了 SQL 标准的新版本。

Adin Falkoff 与 Ken Iverson 一起开发了 APL。他的工作类似于你在 SQL 上所做的工作；二者都是由严格定义的模型的表达式发展起来的。像 Iverson 的符号或者 Ted Codd 的关系模型这样的形式体系，会有助于创建一种成功的编程语言吗？

Don：我认为关系数据模型是 SQL 设计的根基。我认为，计算确定性结果的任何一种编程语言都需要一种明

注 2：Chamberlin，Don 和 Ray Boyce。“SEQUEL: A Structured English Query Language,” Proceedings of ACM SIGFIDET（SEQUEL：一种结构化英文查询语言）”，美国计算机学会 SIGFIDET（SIGMOD 前身）会议论文集，美国密歇根州安阿伯市，1974 年 5 月。

注3：Chamberlin，Don 等人。“SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control《SEQUEL 2：一种数据定义、操作和控制的统一方式）”。IBM 研究和开发杂志，1976 年 11 月。

注4：Chamberlin，Don。“SQL 数据子语言用户经验概要总结（(A Summary of User Experience with the SQL Data Sublanguag”。数据库国际会议论文集，苏格兰阿伯丁，1989 年 7 月。

注5：美国国家标准协会。“数据库语言 SQL（Database Language SQL）”，标准号 X3.135（1986 年和后来更新）。

注6：国际标准化组织“信息技术--数据库语言 SQL（Information Technology—Database Language SQL）”，标准号 ISO/IEC 9075（1987 年颁布，以后又有更新）。

确定义的对象和操作符集，你可以称之为形式数据模型。我认为这是确定性编程的基础。

即使像 Python 这样的松散类型的语言，会发现它们也有一个明确定义的数据模型作为基础。它比关系模型更为灵活，不过它必须明确定义以用作语言语义学的基础。

如果我想自己创建一种新语言，你会推荐我从严格的数据模型开始，还是在某种东西的基础上随着它的发展将它改造成一种语言呢？

Don：理论上，我认为你可以做任何一种方式，不过，你并不总是能够设计一种新语言时又灵活地定义一个新的数据模型。例如，XQuery 的设计者没能发明 XML——它们必须与 XML Schema（译注 1）和其他 W3C 标准定义的数据模型一起工作。

10.2 语言

为什么您开始对查询语言感兴趣？

Don：我一直对语言感兴趣。

您还说其他语言吗（除了英文以外）？

Don：不，我不会说任何其他的人类语言，但我喜欢读书写字，我觉得语言这个话题魅力无穷。我很幸运我的职业生涯生逢其时：当时，Ted Codd 在关系数据模型方面有了突破性的想法。我遇到了一个一生一次的难得机会：参与了一个对我们早期的关系数据库研究有影响的项目。我对语言已有的兴趣帮助我找到了这个项目的生存空间，我非常感谢有这个机会。

在早期的设计决策中，你想要 SQL 是说明性语言，而不是过程性语言。您做出这种选择的重要标准是什么？

Don：这有好几个原因。首先，我们希望这个语言是可优化的。如果用户告诉系统用什么算法来处理查询的详细步骤，那么优化器就无法灵活地进行改变，比如选择一个替代访问路径，或者是选择一个更好的连接顺序。说明性语言要比低级的程序语言更加优化器友好（optimizer-friendly）。

第二个原因是我们对数据独立性很感兴趣，这意味着系统管理员可以自由添加或删除索引、修改数据的结构，并创建新的数据视图。您应该能够以这样一种方式来编写应用程序：他们并没有依赖于物理存储层提供的数据的物理结构和访问路径。因此，数据的独立性是我们希望它是说明性语言的第二个重要原因。

第三个原因是必须考虑用户的生产率。我们认为，这更易于用户在高级别上使用熟悉的词汇来表达意图，而不是在他们不熟悉的低级别上来表示他们的查询。

因此，我们认为说明性语言在最优化、数据独立性以及用户生产率方面具有显著的优点。

译注 1：XML Schema 如同文档类型定义（DTD）一样是负责定义和描述 XML 文档的结构和内容模式的。它可以定义 XML 文档中存在哪些元素和元素之间的关系，并且可以定义元素和属性的数据类型。它本身是一个 XML 文档，符合 XML 语法结构，可以用通用的 XML 解析器解析。

当时在您的小组之内，大家都普遍持有这些观点吗？

Don：我认为，说明性语言总的优点是非常好理解，但我认为，像 SQL 这样复杂的说明性语言实现是否能够达到商业应用程序的性能要求，存在着很大的不确定性。

视图将磁盘上存储的数据的物理结构抽象出来。用户可能会使用数据视图而不是直接通过表交互，这是当时的目标之一吗？

Don：我们认为，视图应该广泛用于数据查询，因为不同的应用程序需要以不同的方式来访问数据。例如，不同的应用程序可能会在不同的聚合层次来查看数据，也可能会被授权查看数据的不同部分。视图为实现不同的数据访问方式方面提供了非常自然的机制。

另一方面，为了更新，情况要困难得多，因为当你通过一个视图来更新，系统需要将这种更新映射到底层存储的数据上。在某些情况下可以做，但在另外一些情况下，并不能得到唯一的映射。例如，查询一个汇总了按部门计算的平均工资视图，这是很棒的，但如果您尝试更新这个视图，我不知道这对修改一个部门的平均工资来说意味着什么。因此，我们发现，与在更新应用程序方面的应用相比，视图在查询应用程序方面的应用要广泛得多。

SQL 是必须处理并发访问共享数据最早的语言之一。这个问题对 SQL 的设计有什么影响？

Don：在同步更新的环境中维持数据库的一致性，这是 IBM Research 的 System R 项目最为关注的一个问题。这项工作的最终结果是对电子事务 ACID（Atomic、Consistent、Isolated、Durable，即原子的、一致的、隔离的和耐久的）属性的严格定义，Jim Gray 藉此于 1999 年获得美国计算机学会图灵奖。System R（和其他关系系统）通过在很大程度上对 SQL 用户透明的锁和日志，支持这些事务属性。

SQL 主要是通过事务和隔离程度（隔离是 ACID 属性中的“I”）的概念来反应并发访问共享数据的。隔离程度允许应用程序开发人员在控制用户相互保护与可以并发支持的用户数量最大化之间的折衷平衡。例如，正在执行一项统计调查的应用程序，可以指定一个较低的隔离程度，以避免锁定大部分数据库。另一方面，一个银行事务可能会指定较高的隔离程度，以确保可能会影响某一帐户的所有事务实现序列化。

同步更新也可以以潜在的死锁形式对 SQL 程序员可视。在某些情况下，两个并发的 SQL 事务可能会遇到死锁，在这种情况下，其中一个事务会接收到返回代码，以显示它的作用已被回滚。

我曾经听说过一个有趣的故事，那是 Pat Selinger 和 Morton Astrahan 在为 System R 工作时的一个“万圣节故事”。

Don：我相信，在 1975 年的万圣节假期，Pat Selinger 和 Morton Astrahan 正在做第一个 SQL 实现的优化器，在进行大批量更新（例如给那些报酬过低的员工加薪）时，它必须选用某个访问路径。Pat 和 Morton 最初认为，通过薪金属性索引来查找收入低于一定的薪水的雇员会非常有效。因此，优化器为了给他们加薪，使用了一个索引在雇员名单中扫描，然后再更新工资。但是，我们观察到，当一名雇员的薪金变化时，他会在索引中转移到一个新位置，而随着扫描的继续，就可能会再次遇到同一位雇员，并再给他加薪一次。在早期的试验中，这个问题导致了错误的而且不可预知的结果。

Pat 和 Morton 在万圣节前夕这个美国假期的一个周五下午发现了这个问题。Pat 来到我的办公室问我：“这个问题我们该怎么办呢？”，我说：“Pat，现在是周五，今天我们无法解决这个问题，我们把它记做万圣节问题，下周再想办法”。

不知怎的，这个词成为无法使用已被更改或修改的属性索引来访问数据的代名词。由于这是所有的关系型数据库优化器都必须解决的一个问题，这个词自然而然就在业内广为人知了。

对于今天和可预见的未来开发计算机系统的人们，您在创新、进一步开发和采用您的语言方面，有什么经验？

Don：我认为 SQL 的历史说明了使用特定的一套原则来指导语言设计过程的重要性。我做了一个列表，上面列出了我（事后）认为对计算机语言设计非常重要的原则。我在这里并不是说，在 SQL 设计时要严格遵循这些原则，尽管很多人事实上遵循了这些原则，其中一些原则代表了原始 SQL 设计中的缺陷领域。在相当大的程度上，这些早期缺陷的影响已经随着语言的演进而得到缓解。

以下是我的设计原则清单。它们中很多看起来似乎是常识，但在实践中坚持并不容易。

闭包

语言应该通过一组带有明确定义属性的对象组成的数据模型来定义。语言中的每个操作符应当通过操作数及其作为对象模型的结果来定义。每个操作符的语义应该指明操作符对于参与对象的所有属性的影响。

完备性

数据模型中的每类对象应该让构建并解构它的操作符成为更原始的部分（如果有的话），并将它与其他类似对象做对比。

正交性

语言的概念应独立定义，而不应受到其使用的特定规则的限制。例如，如果标量值是语言中的一个概念，那么，任何返回标量值的表达式都应该在预期使用标量值的上下文中可用。

一致性

公共任务，例如从结构化对象中提取组件，应该使用与在语言的其他部分中出现的一致方式来处理。

简单性

一种语言应该使用少量的相对简单的概念来定义。设计师应避免加入单一用途功能的诱惑。如果一种语言获得了成功，保持它的简单性需要纪律和决心来驳回很多“改进”的要求。如果语言拥有良好的可扩展性机制（见下项），这种持续不断的斗争将会更容易一些。

可扩展性

语言应该有一个明确的通用机制，籍此可以完美地添加新功能而不会对语言的语法造成很大的影响，甚至应该是没有影响。例如，为了添加使用单独的图灵完备编程语言编写的用户自定义函数，数据库查询语言可能会为此提供一种机制。

抽象

一种语言不应受具体实现的影响，或者依赖于具体实现。例如，取消一组值的复制，应该由“主键”这样的抽象概念而不是由“唯一索引”这样的具体策略来指定（这是在最早版本的 SQL 的缺陷）。在数据库领域中，这个概念有时被称为数据独立性。

最优化

语言不应为执行它的表达式设置不必要的限制。例如，语言的定义应该允许一些灵活性以便进行谓词运算。如果可能的话，语言的语义规范应该是说明性的而不是程序性的，以便提供自动优化的机会。在某些情况下，容忍一些不确定性是有帮助的（例如，在处理一个给定的查询时，是否会导致错误出现可能会也可能不会取决于谓词运算的顺序）。

适应性

不是所有的程序都是正确的。应以这样一种方式来设计语言：许多编程错误可以在“编译”时（即在没有实际输入数据的情况下）被检测出来，或者是清晰地识别出来。此外，语言应该提供一种机制，使编程人员能够在运行时处理异常情况。

10.3 反馈和演进

Ted Codd 最初的关系数据模型论文在公开的文献资料上发表，并在 IBM 之外产生了影响，比如美国加州大学伯克利分校 Larry Ellison 和 Mike Stonebraker 的研究小组。这一过程与“开放源代码”模型类似吗？外部可见性会对 SQL 开发产生何种影响？

Don：在 20 世纪 70 年代，关系数据模型是一种新的想法。这是一个先进的研究和原型课题，而且不是商业可用的。SQL 是 System R 这个试验性研究项目的一部分，它独立于 IBM 的产品开发过程。IBM 研究部门通常会公开出版探索性研究成果，对于 SQL 语言和系统的其他部分，我们都是这么做的。

我们没有公布 SQL 实现的源代码，所以这与今天的开放源码软件的模式有很大的不同。我们没有公开任何软件，但我们介绍了在 SQL 实验性实现中使用的一些界面和技术。例如，我们在公开文献中介绍了一些优化技术，而且如你所知，其中一些论文对开发类似软件的业内人士产生了影响。

这种分享思路的过程不是单向的。在关系数据库研究的初期，IBM、加州大学伯克利分校以及其他组织可以在互惠互利的基础上自由地分享思路。

为什么 SQL 会受到欢迎呢？

Don：我认为，SQL 之所以受欢迎，主要原因来自于 Ted Codd 的关系数据模型的强大功能和简单性。Codd 在数据库管理的概念上实现了革命性突破，SQL 只是以一种简单易行的格式对 Codd 的思路进行了封装。相对于其他现有技术，关系数据库在提高用于创建和维护数据库应用方面的用户的生产率方面做出了重大贡献。

当然，在 20 世纪 70 年代，SQL 并不是基于 Codd 思想而实现的唯一一种语言。我认为 SQL 成功的具体原因包括以下方面：

- SQL 支持一整套数据库管理任务，这个事实对于这种语言被接受来说是非常重要的。使用 SQL，任何授权用户都可以通过简单的命令来创建或删除表、视图以及索引。传统上，这些任务需要数据库管理员的服务，这会关闭数据库并招致大量的开销和延误。SQL 将最终用户从数据库管理当中解放出来，允许他们自由地使用其他的数据库设计进行实验。
- SQL 非常容易学习。只要几个小时就能学会一个足以完成简单任务的 SQL 子集。然后，用户就可以根据需要来学习这种语言更复杂、更强大的功能。
- 至少有两家供应商(IBM 和 Oracle)能够提供强健的多用户的 SQL 实现，而且，SQL 还可以在包括 OS/370 和 UNIX 等在内的多种平台上运行。随着该语言的流行，还在不断地涌现出具体实现，从而创造了一个滚雪球效应。
- SQL 还提供对流行的编程语言的接口支持。连同这些宿主语言一起，SQL 足以扩展到支持各种复杂的应用程序。

- ANSI 和 ISO 在 SQL 标准化方面的前期工作，为用户将 SQL 应用程序从一个实现移植到另一个实现方面提供了信心。这种信心还因为美国国家标准与技术研究院（NIST）创建的 SQL 一致性测试套件而得到加强。美国政府的一些机构要求其数据库采购要符合基于 SQL 的联邦信息处理标准（FIPS - 127）。
- SQL 的开发生逢其时，当时，许多企业正在开发或者将关键应用程序转为使用集成的企业数据库。应用程序开发人员和数据库管理员都供不应求。使用 SQL 的机构带来的生产率提高，能够有效地处理他们积压的应用程序开发任务。

为什么 SQL 能一直很受欢迎呢？

Don：许多 25 年前流行的语言，实际上已经销声匿迹了，包括一些得到大公司支持的语言。SQL 之所以仍在广泛使用，我认为原因包括以下方面：

- ISO 的 SQL 标准提供了一个途径，让语言能以可控的方式演变，以满足不断变化的用户需求。该标准由一个委员会来维护，其中包括用户和供应商，而且供应商为保持其实现符合不断发展的标准贡献了资源。多年以来，SQL 标准已经纠正了原来的语言设计缺陷，并添加了重要的新功能，比如外部联接、递归查询、存储过程、面向关系的功能以及 OLAP（在线分析处理）等。SQL 标准还起到了聚集行业的注意力和资源的作用，它提供了一个通用框架，使个人和公司可以开发工具、写书、教课，并提供咨询服务等。
- SQL 可以管理生命周期很长的持久性数据。在 SQL 数据库上面投资的企业倾向于在那个投资的基础上做事，而不是使用不同的方法从头开始。
- SQL 非常强健，足以解决实际问题。它的适用范围很广，从业务智能到事务处理。它支持多种平台和许多处理环境。尽管有人批评它不够典雅，SQL 已被许多组织成功地用来开发关键的实际应用程序。我认为，这一成功证明了从实验原型发展而来的这种语言，这种原型能够对早期的实际用户需求做出迅速的反应。它也证明了在整个语言的历史中做出务实决策的正确性，因为它已经能够满足不断变化的需求。

235

今天用 C 编写的系统规模，也许要比 20 世纪 70 年代的系统超出好几个数量级，但今天的数据集是非常、非常庞大的。行数不多的 SQL 代码仍然可以运行在一个已经增长很多的数据集上；它似乎能很好地适应数据规模的增长。这是否属实？如果是这样，为什么？

Don：也许这又得归功于说明性语言的额外好处，因为它们比过程性语言更容易进行并行处理。如果您正在大数据集上执行操作，并以一种非过程性的方法来描述，该系统就会有更多的机会，将工作分解在多个处理器上进行。关系数据模型及其所支持的高级抽象，对其可缩放方面很有帮助。而 SQL 作为一种说明性语言，为编译器提供了利用隐式并行的机会。

这些年来，您从基于您研究的产品用户那里获得了反馈吗？

Don：自从 20 世纪 80 年代 IBM 的主流数据库产品开始支持 SQL 以来，IBM 公司进行定期审查，也就是所谓的“客户咨询理事会”，我们收集有关 SQL 和我们的数据库产品其他方面的用户反馈。还有一个叫 IDUG（国际 DB2 用户组）的独立用户群，每年都要在美国、欧洲和亚洲召开一次年度会议，DB2 用户和 IBM 在这个会议进行着很多的信息交流。其中很多意见又会反馈给 IBM 的研发团队，并用它来帮助我们计划今后改进产品。比如对象关系扩展和 OLAP 扩展等许多新功能都是起源于此。

另一种思想来源是维护 SQL 语言标准的 ANSI 和 ISO 委员会，里面包括了来自语言用户和实现者的代表。多年以来，在帮助语言进一步演变以满足用户社区不断变化的需求方面，这些反馈来源已经发挥了很大作用。

您还与心理学家 Phyllis Reisner 一起对 SQUARE 和 SEQUEL 这两种语言进行了一些可用性测试。您在那些测试当中学到了什么？

Don：没错，Phyllis Reisner 是一位实验心理学家，她与 System R 小组一起就我们的语言思路对大学生进行测试。SQUARE 是在基于数学符号的关系数据库语言方面进行的早期尝试，而 SEQUEL 则是使用了英文关键字符号的一种类似语言。

Phyllis 做了一个实验，她向大学生教授了这些语言，想搞清楚哪种方法更容易学习，而且在使用中出错最少。我认为，结果表明英文关键字符号总体来说要比数学符号更容易学习和使用。

有趣的是，虽然大学生所犯的大多数的错误跟语言结构并没有多少关联。这些错误多与数据字符串是否带引号、数据是否大写等这样的事情有关，你可能认为这些都是微不足道的小错误，而不是真正跟语言或数据的结构有关。尽管如此，用户还很难搞对这些细节问题。

236

今天，Web 服务面临着很多 SQL 注入攻击问题，这些服务将输入放入数据库查询中，事前并没有正确地对它们进行过滤。您有什么想法？

Don：SQL 注入攻击是一个很好的例子，它表明了我们在早期没有考虑到的一些问题。我们没有预料到用户会从网页浏览器中输入查询。我想它给出的教训是，软件在进行处理之前，始终应该对用户的输入进行仔细考察。

SQL 是按照最初设计的方式发展的吗？

Don：SQL 的本意是成为一种说明性而非过程性的语言，而且至今仍保留了这个特征。但是这些年来，该语言的发展已经比我们最初的设想复杂多了。它被用于我们初期从来没有想过的许多地方。该语言中已经添加了数据立方体（Data Cube）和 OLAP 分析一类的特性。它变成了一种对象关系语言，使用用户自定义的类型和方法。我们没有预料到这些新的应用。与我们在早期的预期相比，今天的 SQL 用户不得不处理更大更复杂的应用，它也需要更大的技术复杂程度。

我和 Ray Boyce 希望的 SQL 能对数据库行业产生影响，但其影响并没有遵循我们预期的方式。我和 Ray 是要开发一种查询语言，主要是由临时用户（casual user）用来进行专门的决策支持应用系统查询。我们试图让那些不是训练有素的计算机科学家的新用户阶层也能使用数据库。我们期望看到金融分析师、城市规划者以及其他专业人士来直接使用 SQL，他们需要访问数据，但并不需要编写计算机程序。事实证明，这些期望过于乐观。

从一开始，SQL 主要都由受过训练的计算机程序员使用的。事实上，多年以来，人们已经使用自动生成工具生成了大量的 SQL 代码，我们在早期并没有预见到这种开发方式。我和 Ray 曾经认为，直接使用 SQL 的非专业编程人士更有可能使用应用程序支持的基于窗体的接口，通过后台访问 SQL 数据库。临时用户直接访问数据必须要等待电子表格和搜索引擎开发出来才行。

237

你曾经做过关系系统，也做过将用户与文档物理表示相隔离的文档处理工具 Quill。像 Excel 这样的电子表格也能以非常直观的面向用户的方式来表示数据。这些系统是否有共同之处？这种数据独立性能否扩大到整个 Web 网络上来呢？

Don：Quill 和 Excel 都支持我们所谓的直接操作用户接口，使用户可以以可视化的形式来操作具有底层逻辑结构的数据。这已被证明是一个非常强大的隐喻。它类似于关系数据库：用户在较高的抽象层上对数据进行操作，这种抽象独立于底层数据结构。直接操作用户接口易于学习和使用，但在一定程度上，他们仍然需要由专业的底层数据结构支持。一些优化编译器或解释器也需要将用户的意图映射进入底层数据。

至于 Web 总体而言，Web 就是 Web，现在我们没有机会来重新设计 Web，但所有流行的搜索引擎都将用户与信息的处理请求相隔离。我相信，搜索引擎将继续发展，以支持更高层次的抽象，并能发现和利用 Web 上的信息语义。

您试图开发一个有益于普通用户的工具，但在大多数情况下只有程序员才使用它。

Don：我认为，事实表明，我们在语言的最初设计目标上具有一点朴素的乐观情绪。我与 Ted Codd 在关系数据模型的初期一起工作，当时，Ted 在做一个名叫 Rendezvous 的项目，它是一个基于关系模型的自然语言问答系统。我认为当时在自然语言中这么干是不可行的，但我希望我们能够设计一个人机界面，它非常容易理解，人们只需要简单培训就可以使用它。

我认为在很大程度上这种情况并未出现。SQL 的复杂性迅速发展到了编程语言级别，而且也需要同编程语言相当的培训，因此，它主要是由专业人士来使用。

我对最近基于 Web 的应用程序深表钦佩，比如说 Google，它可以用于检索有用的信息而不需要任何培训。早在 20 世纪 70 年代，我们只是没有足够的技术支持。

解释 SQL 的工作原理，或者说明人们可能没有思想准备的关系模型的思想，有没有难度？

Don：我认为这二者都是导致需要 SQL 用户具有一定技术专长的原因。另一个原因与精确查询和不精确查询有关。

当你向 Google 中输入几个搜索词时，你愿意接受一个不精确的结果。换言之，Google 尽了最大努力去寻找与你的搜索词最为相关的文档。这是一个不确定的过程，而且在大多数情况下是非常有用的。

我们正在使用 SQL 研究一个不同的问题，它具有确定的答案，而且，为了获得精确的答案，你需要一个精确程度较高的查询语言。例如，您需要非常明确地了解“与”和“或”之间的区别。与使用 SQL 的结构化查询领域相比，在 Google 当中可以提供稍微模糊一点的查询语义。

犯错误的成本或从网络搜索中获得不确切答案的成本要远远低于你为公司员工付错薪水的成本。

Don：没错，如果你拼错了，或者不能确切记得表里的连接列，您的查询可能在 SQL 根本无法运行，而较少确定性稍微差一些的接口，比如说 Google，对这样的小错误要宽容得多。

您相信确定性的重要性。当我写一行代码时，我要依靠对它要做什么的理解。

Don：噢，在有些应用程序中，确定性是重要的，而另外还有一些应用程序，确定性并不重要。传统上，在所谓的数据库和所谓的信息检索之间存在着一条分界线。当然，这二者都是蓬勃发展领域，而且各有所用。

10.4 XQuery 和 XML

未来 XML 会影响我们使用搜索引擎的方式吗？

Don：我认为这是可能的。搜索引擎已经充分利用包含在 HTML 标记中的元数据，比如超级链接。如你所知，

XML 是一种比 HTML 更具扩展性的标记语言。当我们开始看到更多的基于 XML 的标准，用于标记特定的文件，例如医疗和商业文件时，我想搜索引擎会学会利用该标记的语义信息。

现在，您在为访问 XML 数据的一种新语言 XQuery 而工作。XML 不同于关系数据，因为它包括了元数据。您在设计 XML 查询语言时，遇到了什么挑战？

Don：XML 的最大优点之一是 XML 文档是自描述的。这样就允许 XML 文件有不同的结构，而且允许这些差异是可以通过阅读以 XML 标签的形式存在的元数据来观察的，这些标签本身包含在文件之中。这使 XML 表示信息具有非常丰富的而且灵活的格式。当今，在交换结构并非完全相似的文件的商业应用系统中，包含在 XML 格式中的内部元数据是非常重要的。XQuery 语言的主要目的之一就是去充分利用这种灵活性，以致于查询可以同时操作数据和元数据。

我们在设计 XQuery 时面临的挑战之一就是有很多需要使用语言的不同环境。有一些应用程序，类型在其中是非常重要的。在这些应用中，你想有一个强类型语言，它要做大量的类型检查，而且如果事实证明一个对象不符合预期的类型则会出现错误。但也有其他的环境，有时也被称为模式混乱（Schema Chaos）的环境，此时数据类型就不太重要了。在这些环境中，您可能愿意接受未知类型或异构类型的数据，你可能需要语言非常灵活和合作，而且可以处理多种不同类型的数据。

很难设计出一种语言，其使用范围跨越强类型到松散类型。另外，XML Schema 的类型系统也要比关系数据模型的类型系统更加复杂，而且设计使用这样一种复杂的类型系统的语言是一种巨大的挑战。

结果是出来一种比 SQL 更复杂的语言。我认为，XQuery 要比 SQL 难学，不过，处理这一复杂问题的回报是能够处理 XML 提供的更丰富、更灵活的数据格式。

您曾经参与了 SQL 和 XQuery 这两种查询语言的标准化工作。在这些标准化过程中，您学到了哪些经验？

Don：首先，我了解到，标准在以下方面具有巨大的价值：提供一种形式化语言定义；关注用户反馈；可控制的语言演进机制以满足不断变化的需求。标准过程将具有不同观点和不同专业知识的人聚集在一起。由此产生的合作效率很低，但我相信它将会产生一个相对强健的语言定义。

根据我的经验，下列做法对于一个语言标准化委员会的效率是非常重要的：

- 该委员会应在语言发展的任何时候都维护一个参考分析器（reference parser），应当利用这个参考分析器来验证语言规范和有关文件中使用的所有例子和用例。通过这个简单的过程能够暴露出数量惊人的错误。维护参考分析器可以暴露出可用性和实现问题，并确保没有将任何二义性或其他异常引入到语言的语法之中。
- 该委员会应该维持一个表明语言用途的官方用例组。这些用例有益于在设计过程中探索各种可用的方法，并最终能够成为语言应用的“最佳实践”例子。
- 语言的定义应该由一套一致性测试套件支持，而且至少需要一个参考实现，在采用该标准之前证明其一致性。这种做法往往会揭露“边缘”案例，并确保语言的语义描述是完整的和明确的。要有一个标准，如果没有一个客观的一致性衡量标准，往往没有什么价值。

设计 XQuery 与发明 SQL 的感觉有什么不同？

Don：我的确注意到了一些区别。我们在设计 XQuery 时要比设计 QL 拥有更多的约束条件，这有好几方面的原因。

239

240

其一是，很多人从一开始就对 XQuery 很感兴趣。我们设计这种语言，有这么一个背景：来自大约 25 家公司的代表组成的一个国际标准组织，这些公司对语言应该使用什么符号具有先入之见。我们将工作草案公布在网上，在众目睽睽之下做这项工作。因此，我们得到了很多反馈，其中大部分被证明是有益的。

设计 SQL 是与此非常不同的经历。我们在一个很小的群体里设计了这种语言，IBM 之外没人对此感兴趣，而且 IBM 内部也没有多少人感兴趣，所以我们有更多的灵活性来做出自主决策，而不必向那些很有主意的人解释并证明它。

我发现，低调一些可以是一种解放。

Don：没错，它有很多优点！

团队的规模是否会影响结果？

Don：没错，我发现对于我来说，一个团队的理想规模是 8 到 10 人。这个规模的很多人可以完成很多事，但它足够小，每个人都可以了解其他人正在做什么，而且信息可以很容易地传播而没有太多阻力和开销。正是这种规模的 System R 团队构建了第一个实验性 SQL 实现。

激励研发团队的最佳方式是什么呢？

Don：我认为，激励研发团队的最佳方式，是给他们提供机会让他们的工作产生影响。如果人们可以看到自己的工作改变了世界，他们就会深受鼓舞并努力工作。我认为，这是小型创业公司的一个优势：他们经常做一些革命性的工作，而且也不会对他们的工作产生后续的限制。

在大公司，这种机会非常罕见，但它们的确存在。我个人觉得参与关系数据库技术的早期开发很受鼓舞。我们能在其中看到潜在的革命性影响。参与一个有潜力的项目会鼓舞人们尽力工作。

在您的领域中，您是如何定义成功的？

Don：这个问题问得很好。我会将研究中的成功定义为能够产生持久的技术影响力。如果我们能开发出广泛应用的理论、接口或者方法，并且经得起时间的考验，我认为我们就可以说我们的研究具有一定的价值。

Ted Codd 的工作就是最好的一个例子。Ted 提出的想法很简单，简单到足以让每个人都能理解；他的想法又非常强大，强大到将近 40 年之后仍然占据信息管理行业的主导地位。我们当中没有几个人可以渴望达到那种水平的成功，不过这就是我所定义的研究项目的理想结果。

第 11 章 241

Objective-C

Objective-C 是一款 C 和 Smalltalk 相结合并加上了 Smalltalk 对象的编程语言。Tom Love 和 Brad Cox 在 20 世纪 80 年代开发了这个系统。在 1988 年伴随着 Steve Jobs 的 NEXT 系统的发展，它也得到了普及。目前在 Apple 的 Mac OS X 上它仍然非常流行。与当时其他面向对象的系统相比，Objective-C 使用一个非常小的运行时库取代了虚拟机。 Objective-C 也对 Java 编程语言产生了影响，而且 Objective-C 2.0 在 Apple 公司的 Mac OS X 和 iPhone 应用程序中很受欢迎。

242

11.1 Objective-C 工程

为什么您会选择对现有语言进行扩展，而不是创建一种新语言呢？

Tom Love：这是因为在大型系统中兼容性的需求是非常重要的。在早期，决定一个 C 程序可以在 Objective-C 编译器上运行而且不需要做任何修改，这是非常重要的。你在 C 中能完成的功能不会被禁止；同样，你在 Objective-C 中完成的功能也会与 C 没有任何的不兼容。这是一个很大的约束条件，同时这种约束又是非常重要的。这也使得它可以很容易地混合和协调。

您为什么会选择 C 呢？

Tom：可能是因为我们从一开始就在研究环境中使用 Unix 系统，而且是用 C 来编程，同时我们也一直尝试做一些用 C 难以实现的事情。1981 年 8 月，Byte 杂志创刊了，用于向 Smalltalk 的初学者介绍它的主要功能。Brad 说，“我认为 Smalltalk 的大多数功能，可以添加到 C 上”。

我们是 ITT 公司的一个研究小组，主要是构建分布式编程环境以帮助软件工程师建立通信系统。因此，我们正在寻找合适的工具来构建一个类似当前 CAD 的工具，但它不仅仅是 CAD 工具。

今天看来，Objective-C 在某些方面要比 Smalltalk 好吗？

Tom：与在 1984 年或 1983 年秋天时的第 1 版相比，今天的 Objective-C 的类库已经有了非常大的变化。刚才我们说过的一些应用，对于一种语言来说是非常合适的，但对另外一种语言则不尽然。Smalltalk 绝对是一个学习面向对象编程的好语言。但是它在理论环境中并没有被广泛的应用，我对此也感到非常惊讶，因为它的确是学习基本概念非常好的途径。相比之下，如果我负责编写一个新的操作系统，我不会使用 Smalltalk 来编写。如果我需要开发一些研究模型或是原型等诸如此类的事情，Smalltalk 是一个很好的解决方案。我认为，对于任何一种语言来说，都有其最适用的解决方案范围，而且这二者之间是有交集的。

Objective-C 和 C++都是以 C 为鼻祖的，不过它们沿着两种完全不同的方向发展。现在你更喜欢哪一种方式？

Tom：有好的方向，就会有方法，比如说 Bjarne 的 C++。在某些情况下，一种很小的、简单的，甚至我敢说是优雅的语言是非常简练而且明确定义的。而在其他情况下，它又很难看，很复杂，因为它有一些令人讨厌的特性。我认为这就是两者之间的区别。

243

C++在某些方面过于复杂吗？

Tom：噢，绝对如此。

它仍在发展，并仍然在添加新的东西。

Tom：嗯，随它去吧。我的确希望我的语言要简单一些。APL 是一种很出色的编程语言，因为它极为简单，但对某种特定的应用来说它功能极强。如果我编写一个统计软件包，那么 APL 语言就是绝对的首选，因为它是我所知道的最好的矩阵代数运算语言。不过，你知道，这仅仅是其中的一个例子而已。

为什么您认为 C++比 Objective-C 更为常用？

Tom：因为它背后有 AT&T 的支持。

仅仅是这个原因吗？

Tom：我想是这样的。

您如何看待现在的 Objective-C？

Tom：它仍然存在。对吧？

Objective-C 2.0 添加了许多有趣的特性—Apple 毫无疑问在保持它的活力。

Tom：我昨晚刚与一个 iPhone 的程序员交谈过。他说他下载了一个 iPhone 的开发工具包（Developers Kit），它从头到尾都是用 Objective-C 实现的。这说明它还存在。

您在设计初期有没有想过这种语言可能会应用到移动设备和小设备上？

Tom：我和 Brad 第一次打交道，是我聘请他加入电话业 ITT 公司的先进的技术研究小组。我们的工作是要在当时前瞻 10 年。这么做就证明了我们对此并不在行，特别是在软件方面更不在行。经验表明我们对于预测硬件技术 10 年要变成什么样非常在行，而对于软件来说，能预测 10 年对我们来说是太过乐观了。我这么说的意思是我们预测在 1990 年会发生的事直到 10 年后也没有发生。

即使在 20 世纪 90 年代晚期，人们仍然怀疑此时 Lisp 其他语言已经被成功地发明和使用了 30 年或 40 年。

Tom：对。当然，程序员是出了名的乐天派。另一个则是人们不断地推陈出新，我们发明了 PC，发明了 PDA，发明了程控电话，而且编程开发这些设备的人也是各不相同。好像人们不会始终使用同一种技术。很久以前的传统是，使用大型主机的人、使用小型计算机的人、使用 PC 的人、使用工作站的人都各不相同。他们各自都必须接受一些同样的教训。我们仍然是这样。

听一听参加 iPhone 开发会议的那群人怎么说。他们看起来根本不同于参加大型机会议甚至是 Windows 开发会议的那些人。.NET 程序员不仅是不同的一类人，而且还是新生代的程序员。

您看到硬件方面也是这样吗？

Tom：我并不确定会截然不同。

为什么我们可以预见 10 年内硬件的发展，而不能预见软件的发展？

Tom：硬件拥有良好的可量化数据。而在软件方面，我们没有这样的量化数据。可能你也知道，我在计算方面有些名气。30 年来，我一直有一点嗜好，就是试图搞清楚程序员写一个类到底要用多长时间，而且要花多长时间来测试，每个程序员需要配多少测试人员，每个类需要多少测试用例，一整箱的纸上有多少行代码：100 000 行。

11.2 培育一种语言

您相信一种语言是在不断发展和演进的吗？

Tom：很慢。一个有趣并且复杂的问题跟专有语言、公共领域的语言和开源语言有关，这是一个相当难以解决的问题。如果你有权负责修改语言，这些修改就会有条不紊地慢慢进行，这可能是一件好事。实际上，有些人就是不喜欢使用付费的编译器，或者不喜欢为一种不经常变动的编译器支付年度维护费用。多年以来，我们一直受这个问题的困扰。这是我们尝试设计和部署一种编程语言时遇到的问题，当然，操作系统也有同样的问题。

对于是否将一种特性添加到语言中，您如何做出决定？

Tom：你需要用尽可能少的特性来提供尽可能多的功能和灵活性。

您说过面向对象语言的适用性有些限制。有方法来降低这个限制吗？

Tom：你所选择的任何一种语言都有它合适的应用范围，也有超出该范围的系统。还有一些人在为某种专用应用系统编写紧凑的汇编代码，因为它们需要最大的运行时效率。我认为不会再出现这种情况了，而仅仅需要注意物理空间的限制。我认为不管某一种语言的应用范围有多大，总有它不适合解决的问题。因此，我并不是说面向对象语言的应用范围特别小。比如说，在只有少量的处理器和应用系统的飞机模型上构建一套机载航电系统，这和在 Boeing Dreamliner 上设计一套软件是完全不同的两个问题。

我能够理解您为什么选择了 Smalltalk；显而易见，它是最佳的选择。即使到现在这也是一种不错的选择。

Tom：这是一种优雅的语言。我知道有很早的语言，比如说 APL。APL 也像 Smalltalk 一样，有一个真正庞大的简化原则用来设计和构建它。这是极为重要的。Objective-C 在开始时就被定位为一种混合语言，并且我们严格遵循一条原则，那就是绝不删除 C 的任何功能，相反我们会添加一些功能。因此我们并不是创建一个 C 衍生语言，而是基于 C 的混合性语言。

事实上，一些早期的决定确实非常重要，因为现在还是这样。我经常说我要为 Objective-C 中方括号的用法负责，我曾经和 Brad 有过一次长谈，内容就是：我们应该使用 C 中一贯的语法呢，还是我们创建一种混合语言，我把它描述为“方括号是我们进入面向对象世界的标志”？我们的看法是，如果你有一个混合的语言，你可以构建一个基础类的集合，在某些情况下，大部分工作其实可以在内部完成。对一个典型的应用系统程序员来说，就已经隐藏了很多细节。

方括号是在 Objective-C 中发送消息的信号。最初的想法是一旦你构建了一组类库，你就会将主要精力花在方括号的内部运算上。因此，你实际上是使用的是在混合语言中开发的对象的底层架构，在做面向对象编程。随着程序库 libraries of functionality 的构建，使用过程式语言的需求越来越小，你就可以只考虑内部操作问题了。设计具有两种层次的语言是经过深思熟虑的：一旦具有了足够的功能，就可以从较高的层次上进行操作了。我认为这是原因之一。如果我们已经选择一个非常类似于 C 的语法，我不确认人人都知道这种语言的名字，而且它也不可能到处再用了。

当时另外两个目标是简单和优雅。早在那些日子里，业界人士可能用 20 种不同的程序语言编写了程序。就

我自己来说，我发现，当您尝试使用 APL 来做一些认真的事情时，你开始了解它真正的功能，而且会看到，它对于这些应用程序来说是非常棒的程序。

我的第一台家用计算机是 IBM 5100，它实际上是一台 APL 机。而我认为看看你是否能够使用 APL 将全屏文本编辑器做出来是很有趣的，结果表明这是一个真正的难题。

你必须把整个屏幕当做一个字符矩阵……这确实很棘手。

246

Tom：对。这是很难匹配的。对我们这一代的很多程序员来说，从一开始起就在语言上花了很多时间，至少包括字符串处理语言、Lisp 处理语言，矩阵操作语言和面向对象语言等。对于我来说，我感觉都学到了重要的基本概念，我又多了一种语言技能。

我能够明白为何需求一种良好的通用语言了。您的动机是从 C 开始逐步添加 Smalltalk 特性吗？

Tom：我们试图弄清楚，对于大型国际团队搭建电话交换系统构建编程环境来说，哪一种是正确的编程语言。因为当时的任何语言都无法令我们完全满意。

1981 年 8 月 BYTE 杂志上介绍了 Smalltalk，我们一页又一页地来回翻看了好多次。有一天 Brad 走进我的办公室，对我说："我能把这台计算机带回家一个星期吗？我认为，一周后我再回来，可以向你证明我能构建一个类似于 Smalltalk 的 C 语言扩展"。

我让他把计算机带回家，这在当时绝对是极为罕见的事情。当时那台计算机大约有一双牛仔靴的盒子那么大。这是一台 Onyx 计算机，对于大多数人来说，这家计算机公司早已消失在计算机历史当中了。

我这里有关于 Smalltalk 的 80 本从 83 年到现在的图书。读完它们可能需要一周多的时间，但编译器本身并不复杂。

Tom：是的，它并不复杂。语言的基础本身并不复杂。相反你可以想像，而一个 C++编译器是一个十分可怕的东西，因为它不是一个简洁的语言。它拥有不完全一致的特殊结构，这是它的一个问题。

其他一些受访嘉宾说您最开始其实是想设计一个非常小的核心思想集，然后在顶部再构建其他的应用。这是您的经验还是直觉呢？

Tom：我认为这是一种合理的做法。我应该以真正的纯语言为例开始，而 Smalltalk 和 APL 就是两个显而易见的候选语言。您可能会想到其他语言。Lisp 也是另外一种候选语言。

相反，你可能想要去考虑一种真正不太好的语言。我会给你一些候选语言，它要比自己想象到的更重要的。那就是 MUMPS 编程语言。实际上它非常简单。它有一点丑陋、整洁，也不是非常传统，但是结果表明，它很适合构建电子医疗记录系统之类的系统。

实际上，这是一种编程环境，而不仅仅是一种编程语言。这个环境确实有助于构建高性能电子医疗记录系统。现有的最大系统是美国退伍军人事务部（Department of Veteran Affairs.）构建的全面 MUMPS。

在 108 个应用中，有 100 个左右是用 MUMPS 写的。它大约有 1100 万行代码，这是很棘手的。它跟你此前见过的都不一样。由于现在电子医疗记录对于这个国家和世界来说都很重要，而已知的最大的产品系统就是用这种语言编写的，实际上它比大多数人想的还要重要。

247

我的一位同事一直在说电子制表软件是世界上最流行的编程语言。这种事为什么会让我们感到惊讶？

Tom：我告诉你一个你可能还没有听说过的编程语言的异常例子。原来我曾经在一家名叫 Morgan Stanley（摩根士丹利）的小公司工作了一段时间，他们一直专注于做交易系统，这可能并不让你感到惊奇。

有一个人说，在世界所有的编程语言当中，没有一个是做真正的高性能交易系统的理想选择。他决定自己设计编程语言。他实际上采用了很多 APL 的理念来设计，但他非常坚定地认为，任何一个称职的语言编译器可能有 10 页或更少的代码页。

随着越来越多的功能添加了到语言和环境当中，他一直试图将其缩短为 10 页的限制。这是，他开始缩短变量名，可以在一行中放几个字符。经过大约 15 年之后，你看看那些 10 页的代码，它看起来就像一个核心 core dump。这是 20 世纪 90 年代的事了。摩根士丹利的每一个债券交易系统都是用这种称为 A +的语言来编写的。

我开始关注一些他们一直在使用的编程语言。它们是什么时候提出来的，什么时候完成的，我都把它们列在一个表中。 在我的这个表中已经有 32 种语言了。它们既不是查询语言，也不是命令语言，它们全部都是编程语言。我和小组的人说，“你们认为我们能把它们减少到 16 种吗？”

我开始了精简运动，把所有 32 种语言贴在一墙上，每种语言都印在一个卡片上。这个主意是，在卡片的一面写上语言名字，再用斜线把它划掉，这就表明我们放弃了这种编程语言。有一次，一个家伙告诉我说：“好的，Tom。把你的电话设成免提，并走到这面墙前面来，我们要进行一个小仪式。我们从今天起正式淘汰下面的编程语言。”我真的不记得是哪一个。我看看墙上的名单，并回答说，“哦，不。”他说，“你为什么这么说呢？”我说，“这不是在墙上的问题。我们又回到了 32 种。好消息，我们只是淘汰了一种不必要的编程语言，但坏消息是我们仍然有 32 种语言”。

这是一个幽默的故事，但想象一下，250 人使用 32 个不同的编程语言带来的问题，考虑一下人力资本的问题和资源分配问题。如果你有 15 人在工作，但在下一个项目中准备使用不止 4 种编程语言，这将是一个令人难以置信的昂贵的组织方式。

在这方面，不是越多越好的。

如果您现在设计一种新语言，那会是什么样子的呢？

Tom：记住，我不喜欢加入不同的技术，除非是绝对必要。我会问：“我真的，真的，真的需要它吗？”，我们还没有提到的语言是 Ruby。这是一个干净的语言，可以合理有效地，充分有效地做很多事情。它拥有一个很好的结构，而且纯净。

我不觉得这是一种新的语言的需要。我花费了大量的时间关注一个问题，那就是在你写下第一行代码的时候将会发生什么。

这个词实在是太不精确。你需要找一个更准确的词。在我曾经参加的一个大项目中，我们已经强加了一个 requirements tax 税。如果有人单独使用“需求”这个词，他们必须给娱乐基金增加 2.00 美元。如果他们想谈谈用例，或者他们想谈谈绩效指标，或者他们想谈谈业务用例或业务过程模型，这些都是可以接受的术语。他们不会引起 tax，因为如果你说，“我需要用例或功能规范，或者需要开发的应用程序模拟版本。”这才是一个精确的要求。

我发现这个项目陷入了麻烦，因为他们没有正确地理解、编写代码，这看起来并不是困难的部分。最难的是要搞清楚什么代码做什么用的。

您认为我们已经达到的生产率水平，使得语言、工具、平台和库对于成功来说没有 20 年前我们认为的那么重要了吗？

Tom：我认为你说的很对。我花了很长时间才意识到当你使用一个面向对象编程语言来写一个类时，实际上你是在扩展这种编程语言。

有些语言比其他语言更加明显一些。例如，Smalltalk 就非常明显，而 C++则没有那么明显。

由于我们基本上有能力领域开发可用框架或类库来创建专门的语言，问题就转移到开发周期的其他部分。我既要强调开发前期“我们要做什么”，也要强调后期的测试过程。

249

我还曾经和另外一些人讨论过一个星期，他们打算构建一个日访问量在 40 000 的应用程序。我说：“请告诉我压力测试的情况，如果 40 000 个用户同时按下返回键，你们如何保证系统能够适应这种情况？”

也许是因为从事过电话行业，所以我把它看做是一个系统工程问题。在医学界，我和一伙人讨论从全国的中心数据库传送电子医疗记录数据库的亚秒级响应时间问题时，我问：“你们知道一天的电子医疗记录信息量有多么庞大吗？5GB。”

当你谈到如何在全国范围内实现在亚秒级内传输 5GB 的数据时，成本是非常高的。事实上你可以做到，但是代价不菲。而且这几乎没有必要。

11.3 教育和培训

Education and Training

您对掌握复杂的技术性的概念有何建议？

Tom：我认为我们会以昔日的欧洲业务为例。一个人应该在这个行业发展的一开始便规划需要做什么，如何写测试用例，如何开发项目的功能规格说明，再到更多的技术性问题，比如如何实际设计解决方案，如何实施这些解决方案，还有一些更复杂的事情，比如说系统压力测试以及如何实际部署大规模系统。我认为我们有一种倾向，就是让一些事实上并不能胜任的人来完成这些任务，然后对他们干不好这些工作感到惊讶。

比起意大利来，我碰巧对德国的了解更多一些。但我的理解是，在德国，要成为一名注册建筑师必须要在各种建筑业务上花很长时间，我想是 6 个月。我想在你成为合法的建筑师之前，你必须了解一些上下水管道系统和电路系统的工作原理，以及实际建造房屋需要掌握的很多知识。在软件业，我们缺少让人们进行适当的学术培训和实践锻炼的过程，即使是在学术培训和适当的在职培训之后。

实际经验有多重要？

Tom：我也对航空业做个类比：在航空业，在你最终坐在载有 300 名乘客横跨大西洋飞行的 757 驾驶舱内之前，你必须经过一个非常明确的有条不紊的训练过程，从小飞机到稍大一点的小飞机，再到稍大稍快一点的飞机。当然，航空飞行的这些业务规则是以很多的生命代价换回来的。因此，在要求航空管制之前并没有出现这些规定。

250

学生应当学习更多什么呢？

Tom：我刚刚在上周的一个会上遇到了软件工程界的几位老前辈，我们问他们，美国最好的软件工程学校是

哪些，他们实力如何，国家科研基金对他们有何资助。实际上，答案并不能让人满意。我认为软件工程教育和计算机科学教育有着明确的区分。这里说的不是如何设计编译器或如何编写操作系统，而是如何策划和管理项目，如何扮演好在项目实施过程中的不同角色。

我对欧洲的软件工程教育体制也不是很熟悉。但我认为在美国现在是很弱的。最近有一位瑞典朋友告诉我说，“我要去管理一个软件组织。要做好这项工作，我应该读哪些有关软件工程的书籍呢？”我给他介绍了5本书，然后他说：“我去哪儿找这些书呢？”我说，“你可以通过亚马逊或其他途径来买，但实际上我明天要去去一所著名大学的书店，如果你想要的话，我可以帮你买好寄给你。”

我去了耶鲁大学，带着我的书单来到了书店，结果书店里一本也没有，这实际上说明了很多问题。虽然耶鲁大学的计算机科学学院在常春藤盟校中不是最负盛名的，更不用说在美国，但在很长一段时间里，它是非常受人尊重的。然而，他们甚至没有为学生提供这些书籍，更不用说学生会在课堂上学到比书本更多的东西了。我昨天看到一些有趣的事情。我看到一个新产品的广告，广告上说这个产品完全是用Objective-C写的。

您会如何培训一名软件开发者呢？

Tom：我想先把他们放入测试组，然后教他们如何测试代码和阅读代码。软件行业是为数不多的先教写再教读的领域之一。实际上这是一个错误。它根本不像是拿起一段糟糕的代码，并试图去读懂它。我还鼓励他们熟悉现有的设计得很好的具有良好架构的软件，这样他们就会从内部而不是外部获得经验。

下面是我们公司的一个做得很好的产品的例子。看一看经过精心设计并且做得很好的产品，然后和你现在正在进行的工作进行一下比较。我开始在很短的时间内给他们分派越来越多的活，这样就有机会来判断他们的进步和成绩，并在他们需要的时候为他们提供帮助。

您如何聘用优秀的程序员呢？ 251

Tom：这是一个很大的话题。你可能知道也可能不知道，我在20世纪70年代做的学位论文研究的是成功的程序员的心理特征。

其中一些研究成果现在仍然是有效的。

Tom：是的，其实有一些仍然有效。这是相当惊人的。你应该考虑一些特定的认知心理特征，比如记忆能力、注重细节等。我还期待像沟通技巧、书写和表达能力等。在一个团队工作，能够有效地与团队其他成员沟通是很重要的，然后当你开始部署产品的时候，作为一个领导者，与客户、专家人员或修理组的人员的交流也是很重要的。这并不是在提要求，而是我希望得到这种相关性，比如真正优秀的程序员，如果我想聘请他为项目首席设计师和架构师，我还会注意他们的业余爱好。如果他们精通音乐，那就是非常好的一件事，精通是指他们学过古典音乐，并可以凭着记忆演奏一首钢琴奏鸣曲。这是对他们的记忆能力和注重细节的完美测试，而且它听起来也应该非常不错！

11.4 项目管理和遗留软件

Project Management and Legacy Software

您说过一个程序员得维护半箱的纸质文档。

Tom：没错。这些年来，我参与了联邦政府的很多项目。这一点已被证明会有惊人的帮助作用。10万行代码

打印出来就是一整箱。开发它要花 300 万美元。这需要两个人来维护。测试用例的代码量通常是被测试代码量的 2 到 3 倍。

这与语言无关吗？

Tom：几乎没什么关系。至少它适用于面向对象的语言。实际上，使用一个中等规模的语言，测试人员要多于开发人员，这是因为面向对象语言的功能已经足够强大。我正在做一个 75 万行代码的项目，其中一半以上的代码是外包的。实际上，这根本不是什么先进的技术。如果你说“我打算为每一个程序员配备一个测试人员，”从这个角度来看，你绝对低估了真正的工作量。比如说在一个医疗应用程序中，因为程序员实际上会从外部引进大量未经测试的代码，而你不得不对它们进行彻底的测试。

在这种情况下，你不得不为每个开发人员配备 4 到 6 名测试人员。例如在 20 世纪 80 年代初的 C 语言开发环境下，你可能就不得不为每个开发人员配备 6 个测试员。

由于 C 语言才提高了抽象或代码重用的水平吗？

Tom：这是库的问题。在 OBJECT-C 的初期我做了大量的分析。如果你有一个大 C 程序并使用 OBJECT-C 来重写，最后会有多少行代码呢？这个数是 5 左右，也就是大概只有五分之一的代码行数。

这是一个很好的压缩比。

Tom：这是一个大数目。我们约四年前做了一个项目，是混合使用 COBOL 和 C + +做的。它有 1100 万行代码，最后被压缩成了 50 万行 Java 代码。年度维护费用降到了 1/20。最后到了这样吸引人的数字并没有花很长时间。

重新实现系统时学到的经验教训。

Tom：它总是正确的。事情总会按照这样的方式进行：与系统的第一个版本相比，人们总是比较易于构建系统的第二个版本。其中一个原因是你不必再花时间去问是否有可能。你知道它的存在是不可能的。

你可能知道俄罗斯获得 B-29 轰炸机的故事，就是这种飞机投下了原子弹。他们做出了完美的克隆品，甚至在模型上漂亮而精确地复制了飞机的每一个小补丁。并且他们在两年之内就有了第一架 B-29，而美国则花费了 30 亿美元和 5 年的时间。它比曼哈顿计划（译注1）还要昂贵（注 1）！

实际上，到第二次时就快很多了。

当然，您说的不仅仅是源代码行数的数量级差异。不论是什么语言，程序员都可以维持同样的代码行数吗？

Tom：我曾就该主题与优秀的程序员有过很多对话。这个数字的差异很大。我认为平均值比较好确立。另一方面，我认为它的范围也相当好。这会使得代码编写会很工整、规范，一个人也可以维护 20 万行的代码量。这是非常罕见的，但却有可能。

同样，就像有一只老鼠在窝里做一个人工作得到的馒头，试图保持万行的代码工作。它比前者有一个更大的观察趋势。我们常常会觉得很有趣：如果你的架构确实做得很好，而且设计了一些新系统，然后你把它交付

译注1：曼哈顿计划（Manhattan Project）是美国于 1942 年 6 月开始的原子弹研制计划，历时 3 年，耗资 20 亿美元。

注 1：*http://www.rb-29.net/HTML/03RelatedStories/03.03short stories/03.03.10contss.htm*

资助你的组织，你就会感到很惊讶：应用程序怎么这么快就变得乱七八糟了。人们不知道他们在做什么并开始尝试弄清楚，但他们的行为会在很短的时间内给程序造成许多危害。

253

在开发 Objective-C 时，您对组织原则有何思考？

Tom：我们确实有所考虑。原来在 ITT 公司的研究小组，负责为一个大型国际电话公司构建一个分布式、面向对象的数字电话交换系统。因此就必须有一个分布式的开发团队。

我们都专注于解决这些问题。我从通用电气公司（General Electric）来到 ITT，在那里我从事类似的工作。在 GE 我把它叫做软件心理学研究小组。我们期望一种编程语言，它不仅能让人们更容易阅读、理解和维护，而且对开发团队的组织结构来说，大规模的软件发布也相对简单。

如果政府提出软件开发者应对安全问题负责，软件业将会发生巨大的变化。

Tom：是的，绝对是这样的。而且在我找人参加项目时，我的原则之一是他们参加的新项目与此前已经参加过的项目之间的差异不要超过 20%。

这意味着在你成为一个架构师之前，需要从一个大项目中汲取大量的经验。

Tom：噢，它也有这一特点，不是吗？另一种是做短期的项目。但是，这也有制约因素。有些项目会夸大说做了 3 年，那么对于一个工作了 40 年的人来说，他的项目经验并不多。在航空业有同样的问题。问题的解决办法是模拟现实，这也是我同项目经理一直在争论的问题。事实上，一个人一生能否做 100 个项目还是一个问题，但是如果你能模拟一些决定和精力，你就可以在模拟项目而不是实际项目的基础上来重新构建一套系统，这将会是解决问题的另一种方式。

生产率会更多地依赖程序员的素质还是会更多地依赖编程语言的特点呢？

Tom：个体差异的影响将远远超过任何编程语言的影响。20 世纪 70 年代的研究显示具有相同的教育背景和同样工作经验的程序员，这种差异是 26:1。我不相信任何人声称说，他们的编程语言的效率会是其他语言的 26 倍。

你说过你现在是一个遗留软件再工程专家，而且还要求理解三个词：敏捷、遗留和再工程。这是什么意思？

Tom：让我们倒着来看这 3 个词，首先来看“再工程”。我所说的“再工程”，是使用那些现代设计技巧和技术的功能来进行非常类似的替换。我把再工程项目和现代化项目之间的区别说得清楚一些。现代化项目长期受困于 Fred Brooks 的《The Mythical Man-Month》一书中所描述的第二系统效应。试想这样的一个反复过程：

254

在第一次反复中，我们已经做了一半了还没有弄清楚应该怎么做。这些项目通常都会陷入困境。我一直在找一个真正成功的美国政府的现代化项目。找一个不成功的项目是非常容易的，但是我到现在还没有找到一个真正成功的例子。

在项目上，我一直把“再工程”作为一个主要的限制条件。我并不是说我们要去看看旧系统，想想新系统，然后再从一张白纸起家，完全重新开始。如果我们能够重用工作流程、屏幕定义、数据模型或数据元素，至少是我们能够重用测试用例、文档或是培训课程，那么就会节省大量的时间和精力，也会为项目避免很多风险。

再工程系统的巨大优势是，你有一个工作系统，它能够让你随时从中找到问题的答案，否则你就会一筹莫展。这是三个词中的第一个。

“遗留”当然是指现有的系统，通常是一个企业级系统。它们并不一定是构建在 20 年前古老的编程语言或是更糟的语言基础之上的。例如，我听说过需要用 Java 来重做和再工程的遗留 Smalltalk 应用系统。遗留系统的特征在于，它是一个已经部署和使用的现有系统，它对于某个组织来说非常重要，而且由于某些原因确实必须进行更换。这些原因可能是操作系统太过古老已经淘汰，不再支持系统的运行，或者是很难再找到相关编程语言的程序员等诸如此类的原因。也有可能是由于商业上的原因，由于需要原系统的全部功能而不得不进行重新打包，或者是由于需求完全不同而不能对系统进行再工程。这是一个新的应用程序开发项目。

第三个词是“敏捷”，这是一个已被证明反复出现的过程。因此，具有“风险规避（risk-adverse）”意识的项目经理应该认真地把它视为一种业务方式。

如何才能防止在开发的软件中遗留问题呢？

Tom：我认为你无法防止它的发生。任何产品都是有使用期限的。在软件行业，如果一个系统有很好的组织结构、完善的文档，而且经过了完整的测试，那么它的使用期限通常会超过最初设计期限几十年。最近我在从事一个政府项目。它有 1100 万行代码，没用测试用例，而且其中一部分代码是在 25 年前写的。它没有系统级的文档说明，几年前甚至没有代码注释，完全不受配置文件的控制。从 1996 年起，几乎每个月都要为此系统打 50 个补丁。这的确是一个问题。

你可以把所有这些事情拿来进行逆向分析，并且说，这样做不行，那样做不行，这样做才是对的。

模块化设计怎么样？

Tom：系统设计得越好，它的模块化程度就越高。系统的对象模型越好，它的使用期限就完全有可能更长。当然，一个设计良好的系统，如果业务发生变化，那么系统也许会也许不会超出变化的范围。

根据您的经验，一个团队应该使用多少种编程语言？

Tom：在项目管理方面我有一条规则，但它不一定很准确。一个项目经理必须了解在项目中所用到的每一种语言的相关知识。顺便说一下，这种情况很少发生。我想这也是很多项目陷入困境的根本原因。

有一个项目经理来找我，说，“我们应该在这个项目中使用 6 种语言。你真的不能奢望我能全部读懂这 6 种语言的代码”。我说，“不，不，这不是解决问题的唯一方法。还有一种办法是减少使用语言的种类。”

他最终意识到我是认真的。我参加过很多类似的会议，程序员和项目经理在会上讨论写一个类会花费多少代价，而项目经理根本不了解现代编程语言中的类是个什么东西。

您还使泡沫塑料球（Styrofoam ball）代表类来为您的系统建模吗？

Tom：实际上我们是这样做的。我们也做了它的三维动画版本，但发现它远不如泡沫塑料球那么有用。在项目的发展过程中，泡沫塑料球不断地更新，不仅提供了你所构建的系统结构，还能够使系统的结构一目了然，而且还能显示出每一个类的当前状态。

我们已经用它做了 19 个项目。其中一个项目有 1 856 个类。因为它是一个大型的商业项目，因此类比较多。

类是否仍然代表系统过程的基本单位呢？

Tom：我发现这算是最稳定的事了。你必须定义好你所编写的每个类的属性。如果你只是在写一个类的最初

原型，一个人需要花一周的时间，而软件产品中的一个类则需要一个人花一个月的时间。一个高度可重用的类则需要一个人花 2~4 个月的时间。

256

它包括测试和文档吗？

Tom：加起来有九码（译注2）长。

读懂和理解一个类大约需要一天的时间。这就是很多项目现在遇到麻烦的原因，因为你要使用 Swing 这个类库，或者是用你喜欢的东西，并没有人正在坐下来认真地读懂他打算使用的类库，而且会说，要是每个程序员都需要弄懂 365 个类，那么在编写第一行代码之前要忙上 365 天。

相反，如果您忘记了一开始理解代码的时间，那么就会空出很多的时间。

比如，你可以把这部分时间用在调试过程中。

Tom：噢，或许。你必须在某些地方有一些开销。这是一个大数字。如果你在做一个六个月的项目，那么你可能会延迟两年才能开始。

这么做值得吗？

Tom：可能吧，不过还可以使用其他方法。首先是你可以聘用那些对类了解的人员。另一种方法是你可以把它们划分开，没有必要让每个程序员什么都了解，这个主意总是很不错。你需要有这些想法。你也不必对此感到惊讶。

看看在 Java 世界中的这些现代项目。他们能够轻松地从其所依赖的 2000 个类开始进行开发。一年你有 200 个工作日，那么在时间表上最起码要推后 10 年。

你说过，现在写代码的时间多年来仍然保持一致，但您也提过，其他因素使我们更有效率了。有些生产力的提高需要花时间和精力去学习。

Tom：让一个人花一天时间理解一个类不是比让一个人花费一个月重新编写它更好吗？它是昂贵的，但你可以从大量的功能开始，而在过去，我们不得不从头开始写这些功能。

如果一个月只有二十天，那么你的效率可能会提高 20 倍。这是很好的折中平衡。

Tom：这是很庞大的。让我们只挑选一些我认为现在可能是正常的数字。想象一下，在现代世界的一些重要应用程序中，你需要了解 500 个类。你开始启动项目时，不是在每次启动一个新的项目类。你可能不需要在一个步骤当中从 0 个用到 500 个类。你可能至少需要 5 步来用这么多类，甚至更少。

257

您如何识别出简单性呢？

Tom：在过去，为了描述语言，要使用 BNF 描述的页数来度量，这种度量方法不错，因为它最终让您能够区分简单的语言和复杂的语言。

你知道，当你站在那里，用正确的方式来看待 APL 程序或 Smalltalk 程序时，它只不过是一种语言。就像是一种语言消失了，这是一件好事。

语言的参考手册有多大？它包含了多少知识？Objective-C 编程语言并不是很复杂，类库有点复杂是因为它是

译注2：约为 8.23 米。

经过许多年的积累才形成的。描述这一切细节很难，而且内容繁多并容易出错。它难以测试，而且也很难编写文档。

即使像 C 这样简单的语言都有复杂的语义，那么特殊库函数的内存管理是由谁来负责的呢？

Tom：确实如此。你认为微软的应用程序运行越来越慢是由于内存管理不合理造成的几率是多少呢？你是否遇到过一个使用了 3 年的微软操作系统？我有一个运行微软操作系统的电脑笔记本。我惊奇地发现，我的电脑比同一间办公室的其他电脑工作效率高得多。当完成我的工作并关闭电脑时，其他人刚开始他们的第一个 Excel 表格。

根据您的经验，您认为哪些是最重要的？

Tom：我可以用五个字来形容它：注意新错误。

我们并不需要回味历史，但要感激它。25 年来，有多少计算机科学程序是用 APL 语言编写的？

我不知道到底有多少，但我认为是几乎没有。然而，这是一个重要的经验，因为几乎可以肯定，某类特定的应用程序可能会使用一种特定的语言，不过我认为，熟练的 APL 程序员编写的统计程序会把其他的语言打得一败涂地。

我与软件工程研究所（Software Engineering Institute）的一些人探讨过如何培训软件工程师的问题，我们认为即使有一些研究生课程，也是很少的研究生课程会真正地认真培养人们的系统构建经验而不是算法设计。我认为我们在这方面做得很不好。

这样不是很好吗？就像我现在做的一样：如果你要聘用一名项目经理，他有自己的项目管理证明书，而且有人去搞清楚每个项目的细节，这里的每一个项目，还有可供我查看的数量度量，比如说有多少行代码，多少个测试用例，进度执行情况等诸如此类的东西。

我碰巧当过通用航空的飞行员，我在这上面花了一点时间。从失败的经历中获得的规章制度几乎都是以生命为代价换回来的，在这个国家和全世界都是如此。

在美国，某一天的任何时间点，都有 56 000 架飞机在空中飞行，而且最近连续几年没有一起商业客机事故。这其实是一个非常惊人的纪录。

他们实际上是找到了一些办法，比如说当飞机起飞时飞行员应该清醒。在驾驶特定飞机时，应该在领航员的陪同下试飞一到两次。当这个行业成熟时，我们最终也会秉承同样的精神理念。企业很少考虑开发新的应用程序或是改造已有的应用程序的原因之一就是他们太害怕出现人员死亡。他们会认为他们要去做的任何项目都有可能失败，而且他们不愿意看到在他们手里出现这样的失败。

如果他们能够相信他们已经有 90%或 95%的机会，上帝保佑，或者是 99%的机会成功，而且他们能够在开始之前知道它的成本，那么，这样这个行业就会得到蓬勃的发展。

您所说的“成功”是指的什么？

Tom：关于成功，你肯定听说过各种不同的定义。假设一个项目最终有一百万行或更多的代码，在美国这类项目成功的比例很低，连 50%都不到。这一点尚有争议。人们不喜欢宣传这些信息，所以我也不知道他们是如何得到这些可靠的数据的。也有很多有能力的人一直在试图搞清楚这些信息。但我要说的是得到这些数据非常困难，而不是不可能，只是非常困难而已。

11.5 Objective-C 和其他语言

为什么您是在一种现有的语言基础上进行扩展，而不是创建一种新语言呢？

Brad Cox：除了众所周知的局限性之外，我对 C 语言非常满意。重复发明一种面向对象的程序设计语言肯定是在浪费时间。

为什么您选择了 C 语言呢？

Brad：是我们只能选择它。首先 Ada 是不必考虑的，而 Pascal（被认为）是研究人员的玩具。剩下的 COBOL、FORTRAN，噢，对了，还有 Chill（一种电话通讯语言）。唯一能和 C 相提并论的就剩下 Smalltalk，可是 Xerox 还不愿意卖掉它。

我们的目标是让 OOP 走出实验室，进入实际应用。而 C 是唯一可靠的选择。

259

为什么模仿 Smalltalk 呢？

Brad：因为 C 中没有封装的概念，当我看到 Smalltalk 时，我感觉这正是我需要的。

它让我在 15 分钟之内顿悟了它的真谛。它使我非常震撼。在尝试用 C 构建大项目的过程中，最让我苦恼的就是没有封装并把数据和程序包装成对我来说看起来是方法的东西，就是这样。

您第一次见到 C++是什么时候，对它怎么看？

Brad：Bjarne 听说我的工作以后，在我们都没有完成各自的语言之前，邀请我去贝尔实验室（Bell Labs）。他是完全致力于使用静态绑定和升级 C 的。而我则偏重于以我可控的最简单、破坏性最小的方式向普通的旧 C 中加入动态绑定。Bjarne 的目标是做出一种雄心勃勃的语言：以门级制造为重点的一个复杂的软件生产线。我的目标要简单得多：能够使用普通的 C 装配软件 IC 的软件“烙铁”。

您如何解释这两种语言具有不同的传播速度？

Brad：Objective-C 是一个小公司的最初产品，它除了编译器和它的支持库之外没有其他收入来源。AT&T 构建 C++不仅仅是为了收入，还为分发提供了很大的支持。免费软件几乎每次都会胜过付费软件。

您参与了 Apple 宣布的 Objective-C 2.0 项目吗？

Brad：除了我喜欢 Apple 的产品以外，我同他们没什么关系。

您对垃圾收集器怎么看？

Brad：我认为它们非常棒。一直如此。我不得不和销售商们斗争，他们认为这是语言的一种“特性”，它可以轻松地添加到 C 当中并且不会影响 C 的执行效率。

为什么 Objective-C 禁止多继承呢？

Brad：历史原因是，Objective-C 的是 Smalltalk 的直系后裔，而 Smalltalk 就不支持继承。如果我今天再次决

定，我可能不惜一切代价消除单一继承。继承不那么重要。封装才是面向对象的持久的贡献。

为什么 Objective-C 不支持命名空间呢？

Brad：我直接参与时，我的目标是复制 Smalltalk，而它是没有命名空间的概念。

如你所知，今天的 Objective-C 更像是 Apple 的产品而不是我的产品。目前，我的工作重心是在 XML 和 Java 上。

协议的概念是 Objective-C 独有的吗？

Brad：我希望我能因这一点而获得好评。它是添加在 Objective-C 骨架之上的一种东西，它是我能从 Smalltalk 得到的最小集。Smalltalk 在当年并没有协议之类的东西，这是由 Steve Naroff 添加的，现在他在 Apple 负责 Objective-C。如果我没有记错的话，我认为他是从 SAIL（Stanford Artificial Intelligence Language，斯坦福人工智能语言）那里得到的启发。

看起来 Java 也是受您的设计影响，因为 Java 引入了单继承。单继承也可以从 Java 中删除吗？

Brad：也许是吧。但它不会也不应该被删除。它在那里运行，完成既定的功能。它只是像其他语言特性一样，也可以滥用，而且它也没有封装那么重要。

最初，我做了大量的实验去寻找它的适用条件。后来我发现封装就是面向对象编程的真正贡献。它几乎可以实现继承的所有功能，而且还使得程序结构更加清晰明了。

从那时起，我的研究重点开始向对象的高粒度级别（面向对象编程和 JBI/SCA）转移，这个级别一点也不支持对象继承。

你如何决定一个项目中的某个功能呢？例如，一个垃圾收集器可能会减慢一些 C 应用程序的运行速度，但同时它会带来其他很多的优势。

Brad：绝对如此。事实上，我们在 Stepstone 上开发了一套和 Apple 类似的软件。也可以说是一个 Objective-C 的编译器。但是销售人员会不自觉地去和 Smalltalk 进行比较，而不是和类 C 的产品进行比较。

这样会有默认配置和配置的限制吗？

Brad：可以有，事实上我们也是这么做的。我所反对的就是试图将 C 绑定在市场产品中。

少数的程序语言设计者是先从一个很小的形式化核开始，然后在此基础上进行设计。在 Objective C 中是这么做的呢，还是决定借鉴 Smalltalk 和 C 呢？

Brad：是的，我肯定不是从一个形式化的基础开始的。我所考虑的是硅。我们向一个大型的硅代工厂做了大量的咨询并且参观了他们的工厂。我一直在思考他们是怎么做的，而我们又是怎么做的。我发现他们做的每一件事情都是围绕着组件的重用。他们思考的所有时间和重点都是在组件上，而不是他们的烙铁。当然，对于我们来说，该语言就是烙铁。

我发现每个人关注的都是语言或是烙铁，没人考虑过组件。在我看来这正好相反。这对于芯片制造商来说是非常重要的。因为硅组件是由原子硅构成的，而且还存在买卖它们的商业模式。而在软件业，这种软件组件

的商业模式是很短命的。

261

软件本身是短命的。

Brad：的确是这样。这也是为什么我们更关注语言而不是组件。本质上来说我们没有任何组件的概念。这和建造房子类似。回想穴居时代我们造房子的方式就和搜集一堆集成电路之类的东西一样，只不过我们找的是火山灰而已。我担心这会导致我们走向 Java 的类装入器的路子。现在软件开发的方式是：从一大堆材料开始，然后去除不需要的部分，这就是类加载模型。我现在主要在想另外一种方式，它从一开始什么都没有，然后不断地加入所需要的功能。这也是我们今天盖房子的方式。

我们用英文描述生物学和化学。但问题在于，我们使用的编程语言功能不如英文强大。

Brad：如果计算机像人类一样聪明，我想我对这种方法很有信心。但是我们现在讨论的是像一块砖头一样不会说话的计算机，自从 20 世纪 70 年代我使用计算机开始，这一点就没有改变过。

另一方面，也会有一些出色的新语言出现，比如函数式语言，它们对于多核计算机来说是很有帮助的。我听到很多人都这么说，我没有理由怀疑它们。

并发会对面向对象机制有何影响呢？有必要改变面向对象的方法吗？

Brad：虽然面向对象编程有 Simula 继承机制，但是我认为一种编程语言应该提供足够的线程，以有限制并且可控的方式支持构建展示并发的高级组件。比如，Unix 就是用普通 C 实现并以可控的方式来支持并发的。我已经花费了很长时间在 Objective-C 上实现类似的功能：利用一个叫 TaskMaster 的多任务处理库来实现，它是基于 `setjmp()` 机制的。

另外一个例子就是美国国防建模和仿真办公室的 HLA，它们主要用于军事仿真。它是用多种语言实现的，其中有 C++和 Java。据我所知，它支持事件驱动的并发模型，而不是像其他语言那样主要依赖线程来支持并发。

最后一个例子是我现在参与的一个项目。SOA 支持大粒度的网络驻留对象，因为这些对象本质上是并发的，它们通常都驻留在不同的主机上。Sun 公司的 JBI 和 OASIS 的 SCA 使用细粒度的对象/组件增强了这种模型，这些对象/组件是用来组装构建 SOA 对象的。这是来源于硬件工程学的多粒度方法，它第一次应用到软件设计当中：由低粒度的对象（门电路）构成中粒度的对象（芯片，比如著名的 Software-IC），中粒度的对象最终构成高粒度对象（卡）……依次类推。这与其他语言最大的不同之处在于：在实际的系统当中，其他语言对域的管理是比较复杂，不规则的，而我们能够将它们分成 3 个级别。现在，大家都有这样的一个共识，就是应用软件需要更紧密地综合在一起。这并不是我当初要解决的问题。我这么做的部分原因就是我对高度并发系统的无能为力，并且我相信别人也是这样。

262

应用程序的复杂性和规模看起来好像还在继续增长。面向对象编程会带来帮助吗？从我的经验来看，可重用对象将会增加系统的复杂性和两倍的工作量。首先，你编写可重用对象。然后你必须对它进行修改，并为它留下的空白填补不同的东西。

Brad：你说得没错，如果你所说的面向对象编程是指 Objective-C/Java 风格的封装，这种封装在我的第二本书《Superdistribution（超级分发）》[Addison-Wesley Professional]中称之为芯片级别的集成。除非你像在硬件工程中那样，把芯片级集成只看成是多级集成工具套件中的一级，同时，门级对象在门级、芯片级、卡级以及更高级别的封装选项的不规则碎片（fractal）世界中完美、和谐地共存。

这正是我现在致力于 SOA（机箱级，chassis-level）和 JBI（总线级，bus-level）的原因。这些和传统的面向对象编程一样支持封装。它甚至比传统的面向对象编程更好，因为它封装的不仅仅是数据和程序，甚至还包括对功能强大的整个线程的控制。

最重要的是，多层次集成不费什么气力，而且众所周知，这种方法在其他行业可应用于任意规模。唯一的问题在于，很难被单层面向对象编程的倡导者所理解。在那里，就这样做：过去是面向对象编程对面向过程编程，现在是 SOA 对 JBI 对 Java。

每个人都沉迷于新技术抛弃面向对象编程等老技术的神话之中。这种情况从没发生过，将来也不会发生。每一次新技术的产生都是建立在老技术的基础之上的。

我们似乎进入了一个全新的语言实验阶段，程序员们愿意使用他们以前并不熟悉的语言，比如 Rails 和函数式语言。当语言设计者改进或是发明一种新语言时，以你对 Objective-C 的经验，会有哪些建议呢？

Brad：我曾经试图改进一些语法语言，比如 Haskell，但是没有成功，至少我认为还不够好。我也曾认真研究过 XQuery，它是一种函数式语言，我认为它比 XSLT 读起来更易于理解。

我想我可以使用新方式，但它不是针对 Haskell 的，我还是绕不过它的语法结构。在 Lisp 上我面临着同样的问题。一个海军的项目给我留下了很深的印象，它使用可通过检查来证明的 Haskell 规则来表达复杂的授权和认证策略。

对我来说，今后不会一直去做那种已经做了几十年的工作：编写程序代码。这不是说这一层次的工作并不重要。所有的高级组件都是建立在它的基础之上的，这一点永远也不会改变。我只是想说是编写过程式代码的新方法并不是创新的重点。

令我感到兴奋的是引入全新的概念来做全新的事情，也就是通过已有的组件库来构建高级系统。BPEL 就是一个具体的例子，它是两个层次的集成应用：用于大粒度对象的 SOA 和 JBI（Sun）或是 SCA（OASYS）。具体的例子比如说 NetBeans 中的 BPEL 编辑器。OMG 在驱动模型架构中发挥了巨大的作用。

263

11.6 组件、沙子和砖

Components, Sand, and Bricks

关于创新、进一步的开发和采用你们的语言，您有什么经验和教训想要告诉那些现在和不久的将来开发计算机系统的人们吗？

Brad：我能想起来的就是我一直对多级集成（又名封装）感兴趣：只要任务能够继续划分，就把它们分配给相应的专家，然后利用他们的工作成果，仅需要打破一点或完全不用打破封装，就可以了解内部过程，以便成功地使用它。

在我写的一些东西里面，常常会以木制铅笔举例。当我问人们是数字笔（比如 Word）还是木制铅笔更简单一些，人们都认为木制的更简单。没有人认识到木制铅笔的复杂性，直到我说 Word 是由 8 个程序员完成的，而木制制笔则是经过采伐木材、开采石墨矿、冶炼金属、生产油漆、生产油菜籽油等上千人的工作才完成。铅笔制造工艺非常复杂，只不过用户看不到罢了。

除了一些封装好的函数之外，我对 C 语言（以及类似的语言）不满的地方是它将程序的复杂性完全暴露给了程序员。实际的封装边界就是整个过程空间，Unix 通过 shell 脚本以及“管道和过滤器”的概念实现了两个级别的封装，其有效高级封装的数量是非常庞大的。

为了更好地解释面向对象编程的优势，我经常使用“Software-IC（软件 IC）”来表示一种新级别的集成。它的级别比 C 语言的函数级集成高，比 Unix 的程序级集成低。开发 Objective-C 的主要动机是实现一个简单的软件“烙铁”，它能够从可重用的软件集成单元组合完成大的模块功能。相比之下，C++从另外一个角度实现了这个功能，它是一个能够自动地将有联系的门电路级单元集成在一起的巨大工厂。如果将程序比作硬件的话，那么芯片级的集成主要发生在链接时间，门电路级的集成则主要发生在编译时间。

当时（20 世纪 80 年代中期），轻量级线程并不广为人知，况且也没有什么比 Unix（“重量级线程”）更好的方法。我曾经写了一个叫 Taskmaster 的 Objective-C 库，将它作为卡级集成的基础来支持轻量级线程。RISC 的发现终止了这一计划。因为这种低级别的堆栈帧的修补技术使得它的移植性受到很大的限制。

264 随着网络的不断普及，出现了一种比 Unix 的过程空间更高级别的集成。比如说面向服务的体系结构中，因特网就像是连接 HiFi 音响系统各部件之间的线缆，不同服务器上的服务（程序）功能就像是各部件本身。发现这一点时我非常高兴，因为这近似于是将日常生活中人们关注的不同东西进行第一级的集成。当服务程序远程运行在别人的机器上时，我们并没有什么动力或需要去查看所用服务的源代码。

值得注意的是，也正是第一级的集成，使得它能够借助软件 IC 的概念解决软件的致命缺陷：激励他人提供组件的能力。铅笔（和用于制造它的许多子组件一样）被生产出来，因为针对有形物品的大量劳动是守恒的。在数字产品方面一直没有可比性，直到 SOA 的出现，它使得部署有用的服务变得可行，并在使用它时收费。

在最近的两年中，我已经认识到纯 SOA 中的基本问题所在，最近我的主要精力大部分集中在这个问题上面。在大规模的面向服务结构（比如美国国防信息系统局（DISA）的网络中心事业服务（NCES）项目和美国陆军的火控系统（FCS））的部署中，在协调陆军、海军和空军的运输机制使用完全均等的面向服务系统时，在满足大多数需求方面将会受到极大的阻力，更不用说什么机密性、完整性和不可否认性等。如果这个“计划”扩展到美国国防部（DOD）的巨大系统中，盟军会怎么样？与其他政府机构的兼容性会怎么样？各州又会怎么样？第一响应者会怎么样？无论如何定义这个计划，你总会留下一些需求，而这些需求肯定需要和其他系统进行沟通。让每一个服务的开发人员负责全部是不现实的。

因此，我目前期望在 Sun 的 JBI（Java 业务集成）和它的多语言后继者 SCA（软件组件架构）继承基础之上，能够支持较低级别的集成，这些集成可以组合起来构建普通的面向服务架构。

我们需要硬件协助来创建这样一个系统吗？

Brad：毫无疑问，硬件能够为我们提供帮助。自工业革命 200 年来，计算机硬件就是工程学的杰出代表之一。人们非常善于搭建一个硬件系统，所以我认为在模型建立上可以从硬件方面学习到很多。但是有些东西不是按下一个按钮就能改进的。

最近，我用另外一个例子来说明我所说的软件的基本地位。以盖房子为例。人们已经盖了几千年的房子了。事实上，在某些地方人们还是在用水泥把砖头沾在一起这种古老的方法来盖房子。

265 虽然最近在软件方面有一些进展，但总体上来说，在软件系统方面并没有什么变化。比如说，建立一个 SOA 服务，每一个开发团队都是从 java.net 信息库中得到的原始材料开始，来满足 SOA 服务的基本需求的。

考虑到安全性，以房子为例，墙壁选用什么材料就是基本问题。国防部对安全性有着极为严格的政策要求。

在安全标准方面投入了大量的资金，例如 Web 安全服务标准。最近他们又推出了认证认可（C&A），要求每一个 Web 服务都必须通过认证。认证是一个费时费力的过程，每一个 Web 服务都需要独立地保证它是安全的，并足以安全地安装在敏感的网络上。

这些 Web 服务就像土砖一样。政策和标准要求每一个开发者只能做他自己的东西。但是这些东西并不会被别人认可，因为它的质量还是要取决于开发者的水平。他们真的遵循标准了吗？他们做的对吗？虽然这些土砖是免费的，但是没有人会信任它们，因为只有它们的生产者才了解其质量。

建筑行业很早以前就有了相对复杂的生产形式，我们今天看来是理所当然的。建筑工人也不再做砖了，因为没人会信任他们。政策和标准仍旧存在，认证过程也是如此。然而，我们很少有人能看到这些。我们只能依靠这些看不到的过程来保证市场上的砖块不会风化，还能够支撑房顶。

在土砖到现在的砖的演化过程，技术的进步并不是主要的，主要的是它建立了信任。我们慢慢了解了有关砖的独立的标准规范，知道了砖应该是多长，如何防止风化等。这些我们都可以暗地里依赖独立的测试实验室。大多数人不知道有砖检测实验室，也没有必要知道，我们只需要相信砖的质量来干我们要干的事情。

我们绕了一大圈，就是为了说明经济系统为何重要。我们能够相信砖的质量，是因为有一套激励机制：买砖要付钱。除了制砖专家，其他人只需要完成房子的其他部分，完全不用考虑制砖的复杂性。

去 *http://bradjcox.blog.pot.com/*可以看到更多的土砖和现在的砖之间的结构区别。

最后说一个乐观的吧，这种情况已经开始改善了。一些公司已经开始构建 SOA 安全组件，并且开始努力争取美国国防部将其视为可信组件。我所知道的最成熟的一个例子就是 OpenSSL，而且 Sun、Boeing（波音）等其他一些公司最近也提出了类似的构建 SOA 安全性的倡议。

是封装和分离的概念驱动了软件设计吗？

Brad：我想是这样。它主要取决于其他行业处理的复杂性。这似乎是人为地利用封装将复杂性隐藏起来。

SOA 和你的倡议试图将软件组件化？

Brad：绝对如此。这正是我整个职业生涯的目标。

是您一直在思考这些事呢，还是您认为现在流行的软件开发方法并不能满足现实需要？

Brad：我是从后者开始的，开始查看其他人如何处理这个问题。我是从软件被打破这种认识开始的。

在实践中，软件的大组件的方式与严重依赖数学的计算机科学观点是截然不同的吗？

Brad：也许是吧。我不想过多的评论计算机科学。我还没有看到计算机能够像人一样解决问题，这来源于观点的特征差异。计算机科学家们认为存在一种软件科学，它的目的就是记录它并教会它做什么。我并不这么认为。既有房屋建筑学，也有硅工程学，而我们的工作是先了解几千年来的经验教训，然后再开始创建软件科学。这中间的具体过程对我来说并不重要，但是对于他们来说还是非常值得关注的。

您说过好几次，你认为软件的经济模型已经失效了。

Brad：它并不是失效了。那句话的意思是曾经有过一个模型，而它现在正在分解。从来没有一个模型。你无法让它顺利开始，因为软件的传播就是水蒸气一样。人类还没有发明一种在其上面构建经济学的方法。这已

经是无法想象的，因此，使用 SOA 服务，很容易想象一个服务的经济模型，因为该软件是建立那边的一些服务器之上的，而且它们是收费访问。

这些对象有很小的粒度：我常常希望的是有一个方式来处理细粒度的东西，计算的沙子和砾石对象，java.net 对象，不过人类给出一种解决方案的复杂性就被掩盖了。在中间立场上，那些具有中间尺度的中级前景为我们提供了一个核心。因此在 SOA 层，出现了一些希望。而且在下一层，也可能有点希望。这个词仍然适用。我觉得真正的小粒度对象没有什么希望。

框架的级别是处于中间级别吗？

Brad：当你从非常大的东西开始并着手处理功能问题时，这些都是挖洞（cave-dwelling）对象和单个大组件。我想它应该有一个模型。例如，JBoss 就支持它。基本上，你会放弃具体的软件，而完全是依靠信任。这是一个非常复杂的事情。

你说的能够信任一个软件是什么意思？

Brad：嗯，美国国防部正身临其境，他们会知道答案。这既费时又费力，在你能想到的所有方面都不能令你满意，但是对此他们有他们的答案。

国防部太过专注于 SOA，他们没有意识到单独的面向服务并不能够满足他们的需求。还需要可重用的小粒度的组件来解决面向服务所不能够解决的一些重复出现的问题。用国防部的术语来说就是安全性和互操作性。我现在的工作就是提供一些面向服务的子组件，它能够在面向服务结构中解决重复出现的一些问题。我希望能够将这些土砖做一些改进，将来能够通过国防部的 C&A 认证，使它们能够被大家所信任。

安全性是国防部 SOA 应用需求中的一个例子，它是政策上的一种要求。SOA 应用的构建者唯一必须遵从的就是标准和规范。国防部并不是墙砖供应商，如果没有很好地了解 SOA 标准、安全策略，以及使用 java.net 组件满足国防部安全需求的细节问题，他们可没有可信任的组件供你使用（真实的砖，如果你愿意的话）或者是购买，并把它们组合在一起。

如何从安全性的观点来看待多层抽象？

Brad：我不确定我理解了你的问题。这就像问："在汽车工业中分工的专业化是如何影响行车安全的？"我想，在这种专业分工下，每一件产品都要经过很多人的手，他们其中一些人技术可能不是很好。但是他们每一级的产品都必须能通过高一级的检测，这说明负面因素对大的整体结构影响不大。

请注意，多级集成并不是多级"抽象"。实际上，它是多级的"具体化"：高级组件是由可重用的、不用定制的、预先测试过的组件构成的。例如，安全 SOA 服务是由 JBI 的一个子组件库组成的，这个库用 JBI 组件实现了 SOA 的核心功能，每一个功能都满足它的安全属性：身份验证、授权、机密性、不可否认性以及完整性等。其结果是更加安全，只是因为你现在能够做得更好。

其他人有没有找到令人满意的答案？

Brad：我想是的。看看土砖和现在的砖的区别，为什么我们不用土砖造房子？因为我们不信任它。软件也是一样。土砖的质量主要取决于制造者及其技术水平。抛开技术层面的差别不说，它们之间最根本的不同就在于，现在的砖经过了认证实验室的高温测试。要想成为砖供应商，你不得不通过这一关。这基本上就是国防部的信任模型。

认证和授权基本上来说是一些很费力的测试程序，也有人称为通用标准。现在我想我们能够停止数千年来对上砖的抱怨了。你知道，“我们”意味着整个行业，因为有众多的软件安全性问题，但又没有解决的办法，最后人们就不再相信软件了，也不会再相信了。最终就会归结为应用到软件提供者之上的一种信任模型。

看看 SUN，它基本上是试图创建一个全新的商业模式。你注意他们最近求助于一种开源办法。他们非常符合我刚才提到的这个比喻，旧的商业模式销售具体的软件，而新的商业模式则是在销售信任。他们确实对这一想法产生了共鸣。这从根本上说明了他们在做什么：现在具体的东西免费。如果你想要的话，您几乎可以下载 SUN 现在提供的所有东西，并可以使用它。他们曾经打赌，如果给出选择的话，人们会选择购买具体的东西，同时还提供保证和支持，还有一些东西我在这次采访当中没有挖掘出来。时间将会证明它是否有效，但我认为有一个似乎合理的机会。

你可能已经听说在加利福尼亚州用 COBOL 语言开发系统时所遇到的问题了。用这些“土砖”构建的系统能够帮助我们在将来避免遗产软件的问题吗？

Brad：我非常相信组件，这不是在吹嘘组件，它并不能解决所有的问题。组件能够帮助人们适度地解决上述问题。这就是我们区别于黑猩猩的事情之一。我们发明了一种简单的解决问题的方法，就是把它变成别人的问题。这就是劳动分工，就这么简单。这就是我们和黑猩猩的区别：他们从来没有发明这个。他们知道如何制造工具，他们也有语言，因此，对于大多数显而易见的事情来说，人和猩猩并没有什么区别。我们通过一个经济系统，让别人来处理某个问题，从而解决了这个问题。

如果我们发现了答案，那么它是建立在以前的基础上呢，还是一个全新的呢？

Brad：进化是一种缓慢运动的革命。它们在二进制代码上并没有什么区别。先前你提到了 COBOL 的文章，它提到了最近 COBOL 的标准也支持面向对象。我最近没有继续跟踪 COBOL，但是在 20 世纪 80 年代，我很提倡这么做：将面向对象引入到 COBOL 中，就像我在 C 语言上做的那样。

这就是为什么我能预见事情发生的例子。他们并没有抛弃 COBOL，并用新语言来取代它。他们一直在不断地加入它缺少的功能。

您还认为超级分发是一种可行的方式吗？Web 应用怎么样？

Brad：超级分发（我使用的术语）适用于细粒度对象。粒度越粗，方法就越简单，比如说 SOA。毫无疑问，我认为这需要强有力的激励机制。现在可以从比面向对象编程的对象更容易的方式开说，特别是 SOA。这些将不再存在（很多夸大的“瘦客户端”除外）。

问题是，这就好像是说学会融洽相处才是解决巴以冲突的方法。在美国和南非这种方法被证明是正确的。但是，这数字产品的用户和所有者之间的热战毫不相关。没有一方会妥协并学会融洽相处。抛出超级分发作为这种冲突的“解决方案”，只会使你成为双方炮轰的靶子。

在您的比喻中，无处不在的网络可用性是建设整个城市的先决条件吗？

Brad：如果房子是 SOA 的话，那么网络肯定是这么做的先决条件。没有网络就没有 SOA。但现在的实际情况是没有网络什么也干不了。

机器语言学者 Robin Milner 希望使许多很小的、功能简单的机器能够并行工作，这和您的目标类似吗？

Brad：这是一个有趣的想法。我曾经在军事仿真上花了很多时间，这种想法对于解决这类问题非常有吸引力。在具体应用上可能还有一些我没想到的问题。

11.7 作为经济现象的质量

Quality As an Economic Phenomenon

我们怎么样来改进软件质量呢？

Brad：一种方法是将软件放在服务器上，也就是我们说的软件即服务（SaaS）。这种做法有希望成为一种经济的方案，尽管这种方式同时也有明显的折衷平衡（比如保密性，安全性与性能等）。

我花了几年的时间试图找出一种相反的解决办法，使用在最终用户机器上本地运行的组建来创建一套经济系统，但是我对这种方法很悲观，因为这种方法最终会涉及数字版权管理的问题，而且这个问题可能会一直存在。我还没有看到一个能够协调好生产者和所有者以及使用者之间问题的办法。没有 physical conservation laws 的支持意味着，那么就会有一堆的法律、法院和律师等问题在等着你，最终会逐步升级为极权国家（police-state）策略。想象一下，银行决定夜里打开保险柜，把钱放在大街上，然后再去起诉是谁偷了钱。后果不堪想象。

另外一种经济的做法就是现在广泛使用的手法：广告。此时，用户既不是渔人也不是鱼，他们是渔船。虽然 Google 在这方面取得了成功，但我并不认为最后会有什么好的结果。我们在广播和电视上已经看到了同样的模式，同样的事情迟早也会发生在因特网上。

270

最后一种方法就是不同版本和改进的开放源代码模式，从免费软件到共享软件再到收费软件等。这是我现在工作的主要模式。这不是因为这种模式最好，而是因为只剩下这一种模式了。而且它能够解决国防部自己造成的许多问题：所有权锁定。

我们使用“质量”相同的实际对象创建软件对象时，失掉了什么？

Brad：值得改进的经济系统。

就是这样？

Brad：这就是驱动船的引擎，但是一旦能量（经济）系统发生了创新，其他方面也需要创新。在生物学中就可以发现很多这样的例子。例如，Java 已经有了封装的基本概念，类似于生物学上线粒体、细胞、组织以及器官等概念。在 Java 中，粒度最大的封装体是一个完整 Java 虚拟机，粒度最小的封装体是一个 Java 类。它们之间唯一的级别是 Java 包（.jar）。可以对类加载器使用各种技巧，以支持其他的一些类（仅举一例：用于 SOA 服务的 Servlet）。

最近，我被 OSGI 所吸引，它是在 Java 类和 Java 虚拟机之间创建一个成熟的封装级的第一次尝试。比如，我的 SOA 安全子组件和互操作组件被打包成 OSGI 束。

您认为我们需要更好的鼓励措施来开发更好的软件吗？

Brad：如果你看看改进袜子、毛衣等产品的驱动力，那么，Twinkies，驱动该系统的引擎就是经济系统。

药物公司会在研究上花费数十亿美元，因为它从出售研究而得的药品上可以获得数千万甚至数千亿美元的回报。这种科学中有货币化因素，即使其中有很多是公共研究。

Brad：嗯，这是一种明确你的软件产权的方法。有了砖块和硅片产品，你还需要有自然法则来保护经济系统。这种方式依赖于人类的法律。换句话说，去找一些律师，打个官司，就能获得专利和一些商业秘密，这在将来很有可能。如果是这样就太令人不爽了。

想象一下，银行不把金子放在保险柜里锁好，而是放在马路上，然后去起诉谁偷了钱。这种画面太不人道了。现在我们就有一个绝好的机会避免这类事情发生，那就是数字千年版权法案（DMCA）。数字千年版权法案和美国唱片协会都这么做过，我不喜欢与它有关联。

开源模式的开放访问方式能改善目前的情况吗？

Brad：嗯，开放源码是目前最好的经济模式。我参与的大部分工作都是开放源码；因为在 SOA 成熟并真正得到应用之前，它是最好的模式。

不过，当我在提到土砖的时候，通常我实际上是在谈开放源代码。制砖的材料是免费的，不过结果是，使用这些土砖的人不得不忍受开放源代码糟糕的文档问题。

用土砖做比喻的原因是，在没有其他选择的情况下它是合适的。土砖总比什么都没有要强。不过，这是我们唯一的选择并且它也是很基本的。

在软件设计中，因特网和组网扮演了什么样的角色？

Brad：现在，没有因特网就没法完成软件；它甚至是不可想像的。

但是，从另一方面说，如果因特网或软件背后没有经济模式，我们得到的质量可能没有有形市场的质量好。

在服务和组件方面，在比语言更高的层次上来考虑软件，是否会改变编写软件的经济性？通过出售信任，你会因为销售一点软件产品而破坏销售市场吗？

Brad：我的水晶球（译注3）并不好。我不知道。建筑工业的发展花了好几千年，为什么我们认为我们可以更快做到这一点？我认为它也需要几千年才能结束，但我到时候已经去世了，而且被埋葬了。很难预测那么远。

也许有人会反对我们说，我们没有找到物理规律来指引我们的道路。

Brad：是的，这应该是个优点。它看起来好像弊大于利。它仅仅为初始者削减了经济模型。而且这是个相当大的破坏。

如果你现在创建 Objective-C，它会是一个开放源代码项目吗？

Brad：那里有太多的假设需要回应。当时，没有考虑开放源代码，而且我们不得不支付抵押。如果当时我有一份安稳的工作，我可能会把闲暇时间用在开放源代码上。但是最终，收益就像很弱的万有引力一样，容易被更强大的力量（比如自我实现）所制服。但是收益也是一个最长期的标准，因此，从长远来看这是避免不了的。

译注3：算命者预测未来的道具。

272

您的意思是说，开源项目如果没有财力支持，就不能很好地代替商业软件？

Brad：不，我的意思刚才已经说过了。收益有多种形式。对于我们很多人来说，获得名声同样也满足了这种需要。

网络应用现象是好事吗？

Brad：SOA 将软件放在服务器上不受终端用户控制。这可能会在未来导致一个 SaaS 的经济系统。

这种方式几乎没有让我们看到任何结果，因为 SaaS 的研究仍处于起步阶段，但我可以看到这种做法如何能够培育一个经济系统，它可能会以改进有形事物的方式来改进软件。

11.8 教育

您拥有有机化学和数学学士学位、数学生物学博士学位。您是如何转到创建编程语言上来的呢？

Brad：在读完博士后之后，我仔细考察了一番，意识到相对于当时的空缺职位来说，我对计算机更感兴趣。

您的大学背景是否会影响您的软件设计理念？

Brad：当然了。经常会这样。

如果你检查生态系统，你会发现除了物理守恒定律……在质量或能量上不守恒之外，软件符合生态系统的所有方面。我们的劳动产品由比特组成，它们很容易被复制，所以很难买卖并拥有它们。经济系统、生态学，都失效了。

如果猎豹可以像我们复制软件那样复制食物，那么猎豹和它的猎物都不会进化。

这是有关软件的一个常见问题，如何拥有并补偿你的产品和付出。

计算机科学研究是否有从学术界转移到工业界的议程？

Brad：我从来没有把自己看成是一名“计算机科学研究者”或者一名学者（尽管我确实在这方面花了一些时间）。我的所有其他工作都是在工业环境中，所以我会自然而然地带有偏见。

不过，我从来没有对学术界的研究实力有深刻印象，直到最近，我已经注意到越来越多的学者参与了标准制订机构，比如说 W3C、Oasis 等。依我看来，那是非常令人兴奋的。

273

我喜欢 virtualschool.edu 的评论：

> 计算机科学不是“死（dead）”了。计算机科学从来没有存在过。
>
> 我在自己的新书里试图引进的新范例的核心概念是：引起我们称为软件危机的症状的疾病，是由一种病毒引起的，而这种病毒就是我们处理的由比特而非原子组成的物质，它是由人类生产出来的，而非自然存在的。
>
> 但是，由于这种物质不遵守质量守恒定律，以物质来激励人们在一起工作生产铅笔或装卸行李传送

带的商业机制完全失效了。没有商业，先进的社会秩序就不能发展，所以我们就会一直处于原始状态，这样每一个书呆子都会根据第一原理（译注4）来编造一切。

因此，一切都是独一无二的，所以不存在“位”这个层次以上的东西是符合实验研究的。这就是没有计算机科学这样的东西的原因所在。

10 年后，计算机科学仍然是“死”的吗？

Brad：在你引用的这句话之外，我没有什么更多的可以补充。毕竟，这一问题取决于如何界定“计算机科学”和“死”的概念……以及一个人对抗一个群体或其他人的愿望的多少，相当多还是没有，我完全没有这种愿望。

唯一我可以补充的是，只有软件是独一无二的，它不是由物理定律来支配的（但要注意标准越来越多地使用了这种说法，这是很多年以前编写的），这使它成为一种社会科学，它与物理科学区别很大。

您如何看待 OOP、超级分发和 SOA 的变化呢？

Brad：你似乎在寻找的线索是，我的兴趣根本不是在编程语言上，而是在于：与人们很容易处理的其他事情相比，软件为什么这么难。像为数以百万计的纽约人提供午餐，遵从摩尔定律，破坏星球（眼前的这一个）以保持汽车的生产增长等。实际上，这是一个非常显著的现象，其他物种都没有掌握。我一直试图理解它是如何让软件运行的。

事实上，管理这类复杂性的能力是非常普遍的，我认为它是天生的，它不是尚未发现它的狩猎社会的遗存，每个人都是一样的……每个人培育自己的花园，捕杀他们自己的肉食，建造他们自己的房屋。但即使如此，仍然可以发现劳动封装和分工的痕迹：妻子做饭，男人打猎，孩子们帮忙，老人出主意。

但是在现代生活中，专业化泛滥成灾：把我的问题分解成小块，并把它交给别人来干，这是人类特有的能力。我会去做午饭，如果你开商店，别人拉来日用品，别人种小麦，别人做化肥，等等不一而足，直到开山采矿。这太明显了，以至于我们都认为这是理所当然的，这表现在词汇的缺乏上，甚至是谈论很多层面的产品时也是这样，这些产品就像是饭桌上的午饭那样普通。请注意两个部分：1）分解问题的能力，也就是把我的问题定义成可分解的小块；2）把它变成别人的问题的能力。我一会儿再回来谈这个问题。

在面向对象编程之前，界定一个问题的能力非常欠缺。一个问题，参与的人越多，难以控制的问题也就越多：一堆一堆莫明其妙的文件名，外部变量冲突，几乎没有任何封装性可言等。这与门级电路设计者在设计大芯片时遇到的问题一样。他们的解决方案是什么？将一串门电路封装在一个芯片里。我来负责芯片设计，你来负责把它们焊接在一起。Objective-C 就是基于这样一种思路。

当然，这仅仅是开始，而不是结束（这就是有时我会对以语言为中心的思想失去耐心的原因）。语言仅仅是工具，用不用这种语言要根据遇到的问题不同而定。并且同一问题会在每一个级别上都重复出现。例如，20 年后的今天，Java 的类库出现了同样的问题，人们很难完全理解并且充分地利用它（我给你说的 J2EE）。

回到超级分发上来，它来自于让别人来解决我的问题，我们谈到了由“位”组成的产品（译注5）时，这些

译注4：第一原理（first principle），一个系统研究中的基本原理、规则或法则；该系统或体系的其他原理、规则或法则都是从它那里推导出来或从它那里得到的解释，而它本身却不是从那个体系或系统中的任何其他原理或规则推导出来或得到解释的。数学公理和逻辑原理被认为具有第一原理的资格。

译注5：即软件。

产品不适用自古以来补偿就遵从的守恒定律。为什么你能保证解决我的问题？我用你的产品来满足我自己的需求，你会怎么办？在我的关于超级分发的书中，在小粒度面向对象编程的背景下给出了一个答案，它是基于软件的使用度量，而不是基于我们现在这样的购买软件。

但是，使用度量也会因为易受到不良用户的影响而变得非常困难和枯燥。相反，SOA 就不存在这样的问题。它在服务器上运行，而不是在用户的主机上，怎样就可以监测它的使用，而不必过于担心用户耍花招。你依然会面临为所有级别的产品结构提供公平交换收益的问题，不过这只是一个会计问题，而不是对某个对象加强防护以防篡改的问题。

有趣的是，当你开始从 OOP 到 JBI/SCA 再到 SOA 逐层提升时，所有的那些计算机科学“老”问题仍然存在。唯一的区别是，使用标准数据表示（XML 模式），每个表示就不再需要自定义解析器。这使得代码生成不再需要由日益图形化的语言（比如 UML、DODAF）驱动的 XML 分析器所创建的 DOM 树。至少与无休止地争论哪一个面向对象语言“最好”相比，这些东西很令人振奋。

在那里，使用（C、Objective-C、Perl、Pascal、Java 与 Ruby 等）来做。IT 变得很无聊，所以我转向了有趣的问题。

我们的软件开发教学中缺了些什么呢？

Brad：在我的教学当中，设法让人们意识到在软件质量中强大的经济力量。通常来说，在计算机科学和软件工程的课程中丢掉和忽略了经济问题，根本就不会引起注意。

为什么计算机科学不是一门真正的科学？

Brad：每次你遇到一个新软件，你遇到的一切都是全新的和唯一的。你说说哪一门科学中每件事都是唯一的？

如果你天天都在研究金子或者是铅，你可以通过测量属性和采用科学的方法来研究它们。但软件没有这种方法。

第 12 章 277

Java

Java 语言源于一个运行在小型设备上的开发项目。随着网页浏览器和 applet（译注1）的普及，它获得了越来越多的应用。通过自动内存管理、虚拟机、一组绑定库和某种程度的“仅须编写一次，不同平台运行”，它日益成为一种通用的编程语言。尽管 Sun Microsystems 已经以自由软件的形式发布了其源代码，但该公司仍通过 Java 社区保留对语言的演变和库管理的一定程度上的控制。

译注1：applet 是指小型应用程序，它仅需要很小的内存资源，通常可以在操作系统间移动的一种应用程序。

278

12.1 功能或者简单性

您说过简单性和功能是讨厌的孪生兄弟。您能解释一下吗？

James Gosling：通常系统想更强大就往往变得很复杂。比如说 Java EE，它拥有事务和持久性等非常强大的功能。但在 Java 的早期，这些是根本不存在的。

该系统确实非常简单，人们也很容易理解。如果只要求自己学习 Java 语言和基本的 API，它仍然是非常简单的。但是，当你开始使用像 Swing 和 Java EE，以及其余一些更强大的子系统时，你会感觉到眼花缭乱、应接不暇。看看 OpenGL 库，里面有好多功能真的很强大。不过，我的老天爷，它可不是那么简单的。

尤其是使用 OpenGL，你需要拥有理想化图形硬件的知识。

James：对。我记得爱因斯坦曾经说过，实际的系统应尽可能地更加简单，而不是更简单。

简单性和复杂性在系统中是一成不变的吗？Larry Wall 谈到了一种关于复杂性的水床理论，如果你在语言的某一部分降低复杂性，它就会重现在其他地方。虽然 Java 核心语言本身很简单，但复杂性是否会出现在例如库等其他地方呢？

James：这句话我喜欢用“打鼹鼠”来解释。通常人们说，“哦，我解决了这个问题”。但如果你看一下周围，你会发现他们并没有真正解决问题，只是将问题转移了。

其中一个问题在于语言是供大家使用的。如果你的语言支持专门的事务，使用该事务的人可能会感觉十分便捷，但不使用人的就会感到十分繁琐。

这要付出无谓的概念性开销。

James：噢，他们肯定要付出一些开销，这取决于具体怎样做，他们可能还要付出一些别的开销。

Java 是一个已经广泛使用了十几年的成熟平台，它还有可能被重新设计为一个简单的系统吗？

James：我不知道，我想或许有可能吧。

事实上有一大堆代码使用了这些东西。人们通常想甩掉那些复杂性，可问题是，如果你仔细调查一下周围的应用程序，你会发现他们还是倾向于大量使用复杂性。

279

你曾想用 Swing 代替 AWT，不过 AWT 至今仍然存在。

James：是的，仍有很多人在使用 AWT，这很让人惊讶。部分原因在于手机上还在用它。

但是还有部分原因在于，尽管和新闻报道相反，Java 令人难以置信地在桌面应用程序中有大量应用，主要是用于构建企业级应用程序。有成千上万的应用程序在使用 beta 库，并且它们一直运转得很好，所以几乎没有什么理由放弃它们。

答案是肯定的，原因在于，在一定程度上，如果你有一个带有合理抽象的面向对象系统，那么在世界改变时，你就可以找到更好的方式，而且能使用那种方式来做事，并且这种新方式不会和旧的发生冲突。

你使用了命名空间、抽象，还有封装的概念。

James：是的。把 JavaEE 变为 JavaEE5，我们对 JavaEE 进行了重大的革命性简化。如果你看今天的 JavaEE，仅仅是翻翻 JavaEE5 手册，它在复杂性方面的确做得不错。但是如果你翻翻旧版的 Java 手册，再尝试把这两版都搞清楚，那就会很烦人了。我们很合理地推进了 JavaEE 的发展，除非你不得不脚踏两只船。生活并非都很愉悦。

向下兼容总是很难。是否允许使用者在最新的 JVM 中运行老的 Java1.1 版本程序，这是 Sun 公司工程师们慎重讨论的问题吗？

James：很有趣的一点是，JVM 自身实际上从来没有被讨论过，因为 JVM 是很稳定的。所有的痛苦和烦恼都来自库。在核心虚拟机中，库却是非常容易管理的。它们被设计成模块化，可以自由进库、出库。其实你可以构建一个类加载器，用它来分开命名空间，从而使你可以同时拥有两个版本的 Java AWT。已经有很多类似的工具存在了。

一旦你准备着手处理虚拟机本身，你的麻烦就来了。但是虚拟机里也没有想象中那么多的问题。

嗯，有人说，在 Java 中实际上有两个编译器。一个是 Java 字节码的编译器，一个是 JIT（运行时编译），JIT 基本上对所有东西都专门进行再次编译。而所有那些令人望而生畏的优化工作都在 JIT 中进行。

James：没错。这些年我们击败了不少确实很好的 C 和 C++编译器。当你用动态编译器时，你就会体会到两个好处。一个就是你能确切地知道在哪种芯片上运行。很多时候人们在编译一段 C 代码时，他们不得不把代码编译成能够运行在通用 x86 架构上。而你几乎找不到什么库能特别适合它们。你可以下载 Mozilla 的最新版本，而且它能在几乎所有的英特尔的 CPU 架构上良好运行。还有一个相当好的 Linux 库。它非常通用，使用 GCC 来编译，但 GCC 并不是一个很好的 C 编译器。

当 HotSpot 运行时，它确切地知道是自己在什么芯片组上运行，也确切地知道缓存如何工作，知道内存的层次结构如何工作，知道所有的管道互锁如何在 CPU 中工作，它知道该芯片应得到什么指令集扩展，而且会根据你的机器进行精确的优化。另一方面，它可以在应用程序运行时对其有一个清楚地了解。它还能获得统计数据，从而知道哪些东西是重要的。它能够进行任何 C 编译器都做不到的内联。Java 的内联功能是相当惊人的。另外你还可以加上使用现代垃圾收集器的存储管理方式等优点。使用现代垃圾收集器，存储分配会变得非常快。

您是在说移动分配指针。

James：这只是字面上的移动分配指针。在 Java 中 `new` 的开销约等于 C 中的 `malloc()`。有些时候移动分配指针的基准是比 `malloc()` 好 10 倍。涉及大量的小对象时，`malloc()` 的表现通常会相当糟糕。

说到 C，您如何设计一个系统编程语言？当一名设计师准备构建一种系统语言时，他需要考虑些什么问题呢？

James：我倾向于不过多考虑语言和功能。在我设计语言的那个年代，非常可悲的是，语言基本上都是为了解决特定的问题。运行的背景是什么？人们用它来做什么？它的领域有何不同？使用 Java 而有不同的就是网络。无处不在的网络会使你在思考的时候有些不同，因为它有很多派生的信息。甚至有可能是你祖母起居室的计算机上正在进行着的某项计算。

或者是看起来不像计算机的一种手持设备？

James：在长长的不断变化的事物清单上，你永远不希望看到蓝屏死机。你不想有复杂的安装特性。因此，Java 最终实现了真正强大的故障隔离机制。大多数人并不这样认为，但这种隔离机制确实存在。像内存指针、垃圾收集、异常处理等都与故障隔离有关。其最终目的是确保系统在有小错误的时候也可以正常运行。就比如你的汽车沿着马路行驶，而门把手松了，你的车仍然可以继续行驶。大多数 C 程序的问题在于，你似乎在做一些完全无害的事情，结果变成了一个讨厌的空指针引用，而且还很有欺骗性。它就像榴霰弹一样到处都是。

281

您无法预言什么时候会出现系统损坏，以及在哪里或何时会崩溃。

James：是的。我屡遭打击，对此我完全不可接受。在 C 出现的时候，在 SUN 的早期，性能高于一切。数组边界检查之类的行为或者问一下“这到底是什么”都是完全不可接受的。当 Java 规范出现时，就成了“数组下标检查无法关闭”。没有“无下标检查之类的事”。一方面，它在一定程度上与 C 大相径庭，因为 C 根本就没有下标检查。另一方面，它也是语言规范的内在部分。

如果您要这样考虑的话，它几乎就没有数组的概念。

James：是的。它增加了一些东西，还有有关的语法。现代编译器的法宝之一，是它们能“定理化地证明”一种方式潜在地所有下标检查。虽然你可能也这么看（你知道，禁止关闭指针检查是一件非常糟糕的事），但事实上，这并不是一件坏事。它没有任何负面的性能影响。在循环外部你也许还要做一些检查，但在循环内部，你不得不为它的出色表现而惊叹。

因为 C 语言的字符串是以空字符结尾的（null-terminated），从而无法计算出它的长度，如果仅仅修复了这个问题，我们可能 40 年前就会有更快的 C 语言。而这恰恰是我最近使用 C 时遇到的最糟糕的问题。

James：是的。我喜欢 C。我已经当了很多很多年的专业 C 程序员了。在很多程序员开始用 C 前，我就已经转到 C 上很多年了。第一个 C 编译器运行在 32K 的 RAM 上，而且对于运行在 32K 的 RAM 上的程序来说，最初的 C 编译器是令人惊奇的，但现在你已很难找到一只只有 32K RAM 的手表。

12.2 品味的问题

无处不在的互联网连通性是如何改变编程语言的概念的呢？

James：关于网络会如何影响编程语言设计，这是一个非常大的问题。只要你连上了网络，你就不得不处理多样性，你就不得不处理交互性，你就不得不考虑故障会对其他部分造成怎样的影响，你不得不去更多地考虑系统可靠性。

尤其是，你不得不担心如何构建一个非常健壮的系统，这个系统可以在面临部分失效的时候仍然继续运转，因为人们所构建的大多数系统总是关心损坏。

282

对于软件的传统观念就是要么全部成功要么全部失败；它会正常工作或根本无法工作。它对 Java 异常机制、强类型系统、垃圾收集器及虚拟机等都非常关注。我的意思是说，网络在 Java 设计、语言、虚拟机上确实都有着深远的影响。

在您设计和编程时，什么对您影响最大？我在看您设计或开发的一个系统时，您指着某个东西并说这就是出自 Gosling 之手，您有什么方法可以做到这些？

James：好的。我希望现实就是那么简单。

我遇到过一些架构师，他们说“我有一个商标设计”，反映了他们趋向于这样对待事物。我更加趋向于没有商标。如果我有一个商标，当你跟已经看过我写的代码的那些人交谈时，我会让他们发狂，他们不得不维护我写的代码，因为我对性能的要求非常之高。

我不会使用积极内联机制，当简单的工作确实难以奏效的时候，还是倾向于使用复杂的算法。我会竭尽全力，会比大多数人使用更多的缓存操作。我会出于条件反射中抛出缓存，因为如果没有缓存，我会很紧张。

那使我想起有人问我“为什么使用数组线性搜索？快速排序（译注2）速度更快。”我的回答是：“可能最多只有 7 个元素需要排序，写一个快速排序代价太高了”。

James：我肯定不会为了 7 个元素而构造一个复杂的数据结构。

很多程序员从来不考虑这些事情的实际后果。

James：让我发狂的一件事是，人们会为他们的开发工作而构建系统，有时做一个简单的线性排序查找就够了。他们知道当这个系统部署的时候，有可能会遇到 10 万个元素，而我只测试过 10 个元素的情况，所以我可能也做一个线性搜索，而且他们总是说“我会考虑性能改进的。”而这就像 Robert Frost 说的一样“但我知道路径延绵无尽头，恐怕我难以再回返。”许多代码就是这样。我有一种感觉：在构建系统以后再返回来修改并不常见，所以这个世界上到处都是慢得难以想象的系统。

是因为时间的限制还是懒惰呢，抑或是为非程序员而使编程变得越来越简单造成的？

James：跟它们每一个都稍微有点关系。这也归功于现代机器越来越快，每台机器里都有三台千兆赫时钟，现在你可以在有限的时间内完成一个无限循环。☺

你们为什么选择让 Java 运行在虚拟机上？

James：它对可移植性帮助极大。这样有助于便携性、可靠性，而且说来也怪，它还非常有助于提高性能。使用适时编译的时候，非常容易获得高性能。让 Java 运行在虚拟机上有许多好处。它对调试有极大的帮助。

除了虚拟机设计之外，你打算做一些别的事情吗，或者说你对 JVM 的工作满意吗？

James：实际上，我对它相当满意。JVM 的设计可能是整个系统架构中最稳定的一部分。如果我要离开时却发现某个问题需要解决，我是不会指责它的，因为它的状态非常好，它没有任何事情令你烦恼。还有一大帮非常聪明的人已经在虚拟机方面工作十几年了，这些人基本都是博士编译专家。

译注2：快速排序（Quicksort）由 C. A. R. Hoare 在 1962 年提出，它是对冒泡排序的一种改进。它的基本思想是：通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此让整个数据变成有序序列。

您会参照其他流行语言来修改 JVM 吗？

James：噢，可能会在某些方面有一些小的调整。现在，我们正在围绕一系列的语言设计问题进行仔细研究。有些东西对其他语言确实有帮助，因此你也想在虚拟机中做一些这样的修改，而事实上你要想找到这样的东西是极为困难的。

我们在虚拟机上遇到的重大问题是那些有点哲学色彩的难题。例如，因为没有裸指针，我们很难在虚拟机上实现类似于 C 或 C++这样的语言。

允许裸指针必然会带来很大的可靠性问题，所以我们决定绝不引入裸指针。我是说这会既涉及可靠性又涉及安全性，这实在是太可怕了。因此，要我把 C 和 C++引到虚拟机上来吗？肯定不会。

你举过一个例子，如果你开车时车载收音机坏了，汽车还是会走的，那我们是否需要为软件提供新的构件块，以避免当一部分出问题时整个系统都停止运转这个基本问题？

James：很多原因在于在这些语言系统当中建立的低级管道系统。C 语言最大的问题之一是“一切都会尖叫着停止”，因为它们使用的是指针方式。

一旦你因为使用指针而出现了任何内存错误，系统核心就会崩溃，你就完了。但是在 Java 中，一方面你不太可能会遇到指针 bug，另一方面出现故障时，你实际上还能控制它。

异常系统确实很好地避免了更多的意外带来的破坏，就好比当你的车载收音机坏了时，你只要简单地断开收音机就可以了。

实际上，用 Java 来构建大型企业系统的人会花一定的时间来保障系统各个部分之间相互地合理隔离，如此一来，出错的部分可以保持相对独立。

如果你看一看 Java 企业标准之类的东西时，你会发现很多带有容错机制的框架运行良好。

这样的话还能保证面向对象方法的完整性和正确性吗？

James：现在，面向对象的程序运行得很好。人们对于优势产品的讨论五花八门，他们对所有语言理论的批评性言论极有兴趣，这些批评往往是针对语言中的旁枝末节的，事实上面向对象编程的基本理念却是极其成功的，几乎没有显现出任何大的缺陷。

有时开发者使用对象技术好像是倍加困难。首先，他们必须设计一个可复用的组件，如果之后要做一些改变，他们就要写一些东西来精确地弥合旧系统留下来的缺口。基本上，我发现在“使用对象实现优秀的设计”和“对象会让事情变得更复杂之间”只有一线之隔。

James：噢，面向对象设计肯定需要某种较高的品味。人们会在这个领域中迷失方向、失去控制，这样的例子比比皆是，事情变得有些疯狂，但是实际上情况还好。接口和对象的使用非常成功，它迫使你去仔细思考每一个子系统之间如何相互关联，而这种理性的应用是非常重要的。

面向对象可以使你把一个系统分解开来，系统的每一部分都可以被拆分，这会对革新、调试、容错及其他一大堆的事情极有帮助。

是的，正确地使用面向对象需要某种较高的品味，但它实际上也不难。而且面向对象是非常、非常有价值的，

比某些人热衷的面条式代码（译注3）有价值多了：面条式代码互相之间的直接联系非常紧密，如果你试图更改一个部分，其余所有部分都会随之改变。这一点非常可怕。

如果你不使用面向对象的方法，那将会面临一个非常非常糟糕的世界。当你的系统开始变得很大，当你有一个很大的团队，或者当你的系统随着时间的变化不断演进时，面向对象就开始展现出它的威力了。

你的模块化工作做得越好，越需要将不同程序员的任务进行隔离。当其他地方修改的时候，它们之间的边界也必须修改，这是非常有用的。

12.3 并发性

人们时常谈论摩尔定律的结局：系统一定会变得越来越大，但并不一定会越来越快。你是否同意这个观点？

James：是的，我同意。摩尔定律是关于门数的。很容易就能看出，遵从摩尔定律，数年以来门数获得了极大的发展。但是如何转换为时钟频率？长久以来，它被转换为时钟频率的机会可以说是微乎其微。

为了获得良好的并发性，是否需要修改现有的设计？

James：噢，是的，它在很大程度上改变了设计，尽管从一个问题域转到另一个问题域会存在很多可变性。

所以像大多数数学软件一样，为了让一些数学软件适用于多线程领域，实际上你必须大幅度地修改算法。虽然在许多企业级软件中，软件通常在框架之内运行。

就像在 Java 中，有应用程序服务器框架 Java EE，而且在 EE 框架中，程序几乎无须意识到它们运行在多线程中；实际上是容器（应用程序服务器）来全权处理多线程问题，同时也是它来处理集群和多进程等所有问题。

这些问题已经被抽象出来，你也不必担心它们，而这些工作对于企业级软件来说是非常好的。企业级软件的特点是由许多彼此互不相关的小事务组成的，它们只是各自独立出现！

数学软件中有很多事务是相关的，它们共享公共数据，等等，而且事情会变得更为复杂。而且也没有什么明确的答案。

我们是否需要通过新的语言工具或库来描述它？是否需要所有程序员都用一种完全不同的思路来理解？

James：哦。这得视情况而定。我认为它和特定领域极为相关。对于大多数基于事务的企业及软件来说，你可以选择 EE 框架，这种框架完全可以直接处理多线程。而如果你使用的是一台 128 核的 Sun 机器，实际上你也很难能找到这样的机器，开发者根本不用去管它，它会充分调动自己的多核去完美地运行，一切都非常轻松。

译注3：面条式代码（Spaghetti code）是软件工程中反面模式的一种，是指一个代码的控制结构复杂、混乱而难以理解，尤其是用了很多 GOTO、异常、线程或其他无组织的分歧架构。得名于程序的流向像面条一样扭曲纠结。

这对程序员的技能要求不高，只要有一个适当的程序员，即使他对多线程并不熟悉，其他的交给 Java 就可以了。

James：没错。这些框架可以帮助人们把多线程问题抽象出来。你要知道，如果你在做仿真之类的事情，当你陷入更多的数值问题时，事情会变得非常困难。要把每一种图形算法和数值计算分解到多条线程上，这个事情从本质上来说就是很复杂的。

286

其中一些原因是因为你要进行数据结构访问和锁定等。通常，从数据结构和算法正确性方面来说，它在本质上非常困难。旅行商（或称货郎担）问题是一个特别棘手的问题。也有些问题比较容易，比如光线跟踪图像渲染，但它是一个领域特定的观测问题，也就是你能够处理单个像素，而且它们是完全独立的。

如果你有那样的硬件，可以在低至像素级的水平上来实现并行化。

James：说得对。实际上，它们表现得非常不错。大多数优良的光线跟踪图像渲染都是这么做的。如计算流体力学，它是天气预报及计算出飞机是否会坠落等这些事情的基础算法，因为流体各部分之间的交流非常多，所以那些问题变得更为复杂。分解一个共享地址空间系统非常容易，但在一个没有共享地址空间的集群之内分解确实是相当困难的。你知道，对于计算流体力学算法，它们很快就会因集群节点之间的通信成本而停止运转。他们在多核状态下表现更好，但往往深受算法特定之苦。

我之所以喜欢 Scala 这样的函数式语言，原因之一是：如果你用 Scala 编写了一个数值算法，那么编译器就可以进一步推理出算法的目的。它可以更好地实现从算法到多线程、多核分布式系统的映射。

这是因为 Scala 是一种纯函数式语言吗？

James：它并不是纯函数式语言。原因之一就是，对于多数人来说它具备双重性质。你也可以把它当成 Java，也可以把它当成纯函数式语言来编程。

有没有这样的问题域，其中共享内存多线程的性能要比函数式更好？

James：实际上，对于企业应用程序来说，这种基于框架的方式在多核分布系统上运行确实非常非常好。我认为 Scala 之类的系统并没有很大的优势。如果你是在编写旅行商之类的算法确实就更加有趣了。

看来来自 Green Project 和 Oak Project 的一个深思熟虑的决策好像是，在无处不在的多线程网络世界中，设计适用于这种网络的编程语言意味着需要用于同步的基本类型，而核心库也必须是线程安全的。

James：我们有大量的线程安全机制。实际上，这是一种非常可怕的情况，因为这通常意味着你的系统上只有一个 CPU 在运行，你肯定要付出一些代价。不过在这种特殊的情况下，我的意思是说，抽象化肯定会于事有补，因为有更多的抽象 API 和接口存在，Java 虚拟机之类的东西就会比真实的机器具有更好的抽象性。底层机制可以做很多改进。

287

在多线程情况下，原来的 HotSpot 虚拟机（译注4）会了解到这与单核的机器是不同的。

译注4：HotSpot 虚拟机是 Java SE 平台的核心组件。它实现了 Java 虚拟机规范，在 Java 运行环境中作为一个共享库提供。

使用 JIT（译注5）技术时，你可以说："我确实不用担心这部分的同步问题，因为我知道从来没有死锁；我们从未为这个特定的内存段进行线程竞争"。

James：是的，而且它的出现既不可思议，又显而易见。没人意识到这点。这是与 64 位指针类似的问题。

使用 C 语言的人们深受困扰，因为他们必须让应用程序运行在 64 位系统上，而在 Java 应用程序中，绝对没有你必须要做的事。

12.4 设计一种语言

要不是为了 Java，您还会选择 Scala 语言吗？

James：是的，很可能会。

所有这些新语言不仅仅是研究项目，而是真打实干要在 JVM 上构建一个真正强大的语言，您对此怎么看？

James：我认为它们相当酷。

您难道没感觉到威胁吗，好像它们汲取了你的全部设计精华并避开了你？

James：不，Java 的所有重要之处都在 VM 里。那是互操作性工作的基础。从某种意义来讲，Java 和 ASCII 的句法设计应该能让 C 和 C++程序员感觉舒服些。它确实做得很棒。大多数 C 和 C++程序员都能看一下 Java 代码，并且可以说，"噢，我能看懂。"

即使我不知道 API 的细节，从战术上讲我也明白它在做什么。

James：这确实就是重要的设计目标之一。在某些抽象世界中"什么有可能是世界上最好的编程语言"，这实际上并不是一个目标。我个人认为 Scala 是相当有趣的。Scala 的问题是，它是一种函数式编程语言，而要想用好它，大多数人都得颇费一番周折。

如果让我设计一种仅供我自己使用的编程语言，那很可能大多数人都得为之抓狂。

会不会是 Lisp 呢？

James：它不可能是 Lisp，不过在很多细节上可能与 Lisp 相似。

Bill Joy 曾经说您的目标让 C++程序员放弃并反对 Common Lisp。

James：如果你看一看什么是 JVM，那么从某种意义来说那就是事实。

译注5：JIT（just in time）即时编译技术。它允许实时地将 Java 解释型程序自动编译成本机机器语言，以使程序执行的速度更快。有些 JVM 包含 JIT 编译器。

288 **对于程序员来说，VM 不应当很慢，或者无处不在的垃圾收集这种想法是非常高效的。**

James：人们确实没有意识到一件事就是，垃圾收集对保证可靠性和安全性非常有益。来看一看 bug 频发的典型大系统：始终都是内存管理 bug。我对一段时期以来发现的所有 bug 进行了统计，特别是查找它们花费的时间。内存破坏 bug 的问题之一就是要花费过多的追查时间。我发誓不想再浪费一个小时再查一次。

我编写了 JVM 的前两个垃圾收集器。垃圾收集器的调试过程非常讨厌，不过一旦完成调试以后，它们就会正常工作。现在我们已经有了一些非常“火箭科学（Rocket Science）”（译注6）。

您最初编写的垃圾收集器是哪一种？

James：我需要一个能在很小的空间中工作的收集器。它在概念上是一个基本标记（mark），带有压缩（compaction）的清扫（sweep），而且还具有一定的异步运行能力。它没有更多的空间来实现更现代化的设计了。如果有人使用句柄，那么空间的使用会进一步压缩。

一个附加的间接指针可以帮助你完成复制吗？

James：因为我尝试使用 C 语言库，但是最初的版本是 semi-exact。在堆中的所有指针都是 exact 类型，但是在栈中就是 inexact 类型，因为它实际上会先扫描 C 语言栈来寻找是否存在看上去类似于指针的东西。

这是我见过的能处理好 C 语言栈的唯一方法，除非你根本不想使用 C 语言栈，而 C 语言栈既有优势也有劣势。

James：对。实际上，这就是 JVM 现在正在做的。它没有使用 C 堆栈。我的意思是它有自己的堆栈机制。对 C 代码，必须用一个独立的堆栈。关于 JNI 的一件棘手的事情就是在各领域之间的转换。

考虑到 C#中很多的灵感来自 Java，您认为 Java 还有哪些特性可供其他编程语言借鉴？

James：噢，我的意思是，C#基本上吸取了 Java 所有的特点，尽管它们非常奇怪地决定添加那些不安全的指针，从而使得安全性和可靠性大打折扣，我感觉这样做是非常傻的，不过人们还是几乎利用了 Java 的大部分特性。

您写了垃圾收集器，这样就不用在调试内存管理 bug 上浪费时间了。您如何评价 C++的指针和 Java 的引用？

James：C++中的指针是一场灾难。它们就是错误之源。它不仅只是直接地实现了指针，而且必须人工地管理回收，最重要的是你必须在指针和整数之间进行类型转换（cast）；而且你还必须建立许多 API。

289 **您在 Java 中设计引用的目的是为了解决所有这些问题，同时它依然可提供 C++指针的优点？**

James：没错，你可以完成在 C++中使用指针能够完成的所有重要任务。

有些问题是可以通过在语言里实现一个通用解决方案而避免的，您经常遇到过这样的问题么？

James：这种问题在 Java 语言中随处可见。异常机制就是一个例子。在 C++中，人们完全无视他们获得的种

译注6：火箭科学（rocket science），意思是很难做、很难懂的事情。

种错误代码。而 Java 可以使您轻松地处理出现的错误。

适当的默认选择能否帮助程序员不借助外部库或附加软件来编写出更好的代码？

James：多年以来，大多数语言都已经解决了大部分的这类问题。比如，C++中多年存在的多线程问题。Java 代码中多线程是被非常严格设计的，结果就是，Java 可以与多核机器合作得非常、非常好。

语言设计和用它这种语言编写的软件设计之间有何联系？

James：哦，这其中联系非常微妙，而且无处不在。我的意思是，一个面向对象的语言确实鼓励你构建模块化系统。你知道，当你拥有了一个强大的异常处理系统时，它可以鼓励你建立强健的（robust）系统。基本上，你能想到的每一个语言特性，都会对软件设计有着微妙的推动作用。

您希望在标准语言中还具有什么其他特性？自动代码检查怎么样？

James：噢，我们在编程工具里有很多类似的实现。你要知道，如果你看看 makeme(?)的功能，它基本上就像一个实时 LINT（译注7）等所有的那些高级工具。现在，你无法仅单独考虑一种语言，你应当把它们和工具放在一起来看待。

在设计一种语言时，您会考虑人们是如何调试它的吗？

James：要想实现一款高可靠性的软件，需要考虑很多重要的事情，但这不仅仅是调试本身这么简单。在调试方面我们也制定了大量相关的标准，例如系统如何与调试器进行交互等。

实际上，你在语言中做了很多工作，避免陷入你必须调试的困境；这就是内存管理器、强类型系统、线程模型遍布 Java 的原因，这些东西甚至会在你调试之前已经产生了积极的影响。

如果一种语言是平台无关的，会不会影响调试？

James：噢，你知道，从开发者的观点来看，调试是完全无缝的，它可以运行在 Mac 上，调试 Linux 服务器上的某些东西。它干得非常棒。

您如何调试 Java 代码？

James：我只使用 NetBeans。

您对 Java 程序员有什么建议吗？

James：使用 NetBeans 非常小心地构建任务，还有很受欢迎的 JUnit，把它们组合在一起效果会更好。

计算机科学系的大学生缺乏什么？

James：多数大学注重事情的技术层面，许多软件工程只是："好的，给你一段代码，找出一个 bug 来"。而常见的大学作业可能是"写一段可以工作的软件"，这样你在开始时拥有的只是一张空表，因而你可以做任何事情。

译注7：LINT 是一个历史悠久，功能异常强大的静态代码检测工具。

而且，许多软件工程都是团队合作的社会动力学成果，但大学里根本就没教过这个。

您如何看待软件文档？

James：当然是越多越好。

对于 Java API，还有名叫 Javadoc 的工具，它可以从源代码中提取文档。人们往往发现软件文档与实际的 API 经常不匹配。因此，通过自动提取的很多样板文件，同步问题有了很大改进。实际上，Javadoc 生成的 API 文档是可信的。

因此，对于代码文档化来说，最重要的一件事就是使用 Javadoc 工具，然后是在你的代码中添加合适的注释，这样就可以人人受益了。

您认为在构建项目之前需要形式化的或完全的详细说明吗？

James：我对形式规范的看法非常复杂。我认为它们在理论上非常重要；但在实践中，它们看起来并不好用。对于一些小问题来说，它们通常表现尚可，不过问题越大，形式规范的用处就越小，只是因为它们不能很好地扩展。

更重要的是，形式规范通常并不会去解决实际问题，只是把问题转移了。它把你软件中的问题转化成了在你的规格说明中发现的 bug。然而，在规格说明中去发现 bug 往往很难。

即使你不做形式规范，也会做一些需求分析——许多组织都遵循这种瀑布式开发流程，在这种流程中，会有一个小组提出需求文档，然后把它分发给实际构建系统的人。

需求文档通常问题重重，但是只有确实存在很紧的反馈循环时，我们才会发现需求中的这些 bug；你不会在规格说明中发现 bug。因此，虽然我通常对需求和规格说明之类的文档乐此不疲，但我也不会太把它们当回事。我肯定不会指望这些文档能解决大问题。

我也曾经跟使用 UML 和其他语言的人聊过，我感觉听起来很有趣的一种想法是，我们使用高级设计语言来构建软件背后的逻辑。他们提到有可能在编码之前就把模型中的逻辑错误找出来。

James：是的，在 Java 领域中，已经有了许多基于模型的高级工具。你会发现，在 UI 框架、UML 框架的 Web 应用程序框架中，许多高级建模工具的功能是非常强大的。如果你有 NetBeans 软件，你会发现这是相当复杂的 UML 建模系统。你可以在一开始使用 UML 模型来详细描述一款软件，然后让它自动生成软件，或者你可以把 UML 建模工具用作一种“考古”工具来探究一款软件，它们会有帮助，不过仍然有一些问题。

12.5 反馈循环

关于 Java 语言自身而不是实现，您获得了多少反馈？

James：小伙子，我们获得了关于语言的很多反馈。

您是怎么处理这些反馈的呢？

James：如果某些特性只有一个或两个人想要，我们就倾向于忽略它。

因为在语言中，你必须非常清楚你改变的东西。在 API 中修改会稍微容易一些，但是通常来讲，要是没有特别强烈的需求，我们就不会做任何改动。

所以如果很多人都有同样的需求，我们就可能会说“噢，好的，那可能值得一做”；而对于百万分之一的开发者要求的功能，我们则可能会说：“把它加进去可能会弊大于利”。

向 Java 开放源代码后您获得了什么经验？

James：哦，我们有很多良性互动。我的意思是，我们在 1995 年开始提供 Java 源代码，人们已经大量地下载和使用，从学位论文项目到安全审计，现在，它变成了一种非常强大的语言。

它会使语言的实现和合作更加精益求精吗？

James：开源之后，会有很多人为其做贡献，这当然是件好事。它最后会成为一场对话。

您能以一种民主的方式来设计语言吗？

James：这是非常非常好的一条界线，如果过于民主，你得到的只会是一堆垃圾；而过于中央集权的话，就可能没人会使用，因为它只是一个人的观点。

所以说，和大众交流，还要有一个适度严密的决策过程，这确实是非常重要的两件事。

您认为开发一种新语言是有益的吗？或者如果有新目标，并且您可以做点什么来实现这个目标，您会去写一种新语言吗？

James：我想两者可能都有一些吧。我想培育一种语言不是问题，不过我认为门槛要求还是相当高的。当被证实有其价值时，培育语言是很不错。如果价值确实非常大，这种随意修改句法之类的事情没什么意义。如果你某天早上起来判定括号很讨厌，我们可能会对此无动于衷。但是对于某些可以真正影响人们构建软件能力的事情来说，我们就会去改变，几年前，我们做了大量工作将范式加入到 Java 中，这就是一件相当好的工作。

设计 API 时您使用什么准则？

James：我的第一个原则是尽可能让它小。而且，我想我的第二个原则是基于用例来进行设计。人们在设计一个 API 时总是有这样的失败模式：他们就像在真空当中设计 API 一样，坐在那里说：“噢，有人可能想要这么做，”或“有人可能想要那么做”，这只会使 API 无限制扩展，变得比实际需要复杂得多。

但是当你实际做这些的时候，你会问“人们想用这个来干什么呢”？去看看那些人们试图去拖曳菜单或是连接网络的系统。你知道其他地方运行的是什么吗？与 API 所写的功能相比，人们实际上又在做什么别的事情呢？

只要对其他语言编写的软件进行统计分析，你就会在 API 设计和语言设计中做很多有趣的事情。

在您设计 Java 的过程中，有什么经验和教训要告诉别人吗？

James：设计语言并不是很难。设计一种语言时，最重要的不是设计语言，而是要搞清楚为什么设计语言。

它的背景是什么，人们会用它来做什么？

真正令 Java 与众不同的是它在网络中的应用。网络会对编程语言设计有什么潜在影响？结果表明，它具有深远的影响，在某种意义上，一旦你考虑到网络的潜在影响，语言设计的选择上就会非常简单。

Java 是否影响了人们对平台无关性的看法？

James：我认为普通用户不会关心平台无关性。我想这一点很有趣，我从一名工程师的角度出发构建出了这些大规模的系统，但是公众根本没有意识这一点。

当你在 ATM 机或收银机上使用 Visa 信用卡时，后面肯定有一大堆 Java 代码正运行在 SUN、IBM、Del 或 HP 机器上，也可能在 x86 架构或是 PowerPC 架构等上，而你在读卡机上刷卡时根本没有留意这些。

而且，这些机制都隐藏在屏幕之后。你乘坐地铁的时候，如果使用类似的卡，比如说在伦敦地铁公司（London Underground）使用 Oyster 卡的时候，那肯定就是一个基于 Java 的系统。

你确实使用了与各种平台无关的特性，而且，如果要强迫消费者意识到构建系统所用的编程语言，那么这个系统就真的是非常失败。我们的目标之一就是做到完全透明，让人们从中解脱出来。当然，这让销售人员心有不悦。他们宁可让你每次乘坐伦敦地铁时都有一个 Java 标识在你面前晃动，不过，这也太荒唐了。

第 13 章 295

C#

当微软刚刚处理完来自 Sun Microsystems 公司关于 Java 编程语言修改的诉讼官司，他们就转而求助于经验丰富的语言设计者 Anders Hejlsberg，请他设计一种有强大的虚拟机支持的面向对象的新语言。结果就产生了 C#，用以替代 Microsoft 生态系下的 Visual C++和 Visual Basic。虽然仍然难免要与 Java 在语法、实现和语义方面比较一番，但是这种语言本身的发展已经超越了其根源，同时吸收了来自 Haskell 和 ML 之类的函数式语言的特性。

296

13.1 语言和设计

您已经创建了并维护着好几种语言。您是从 Turbo Pascal 的实现者起步的，从实现者到设计者是不是一个自然的过程呢？

Anders Hejlsberg：我认为这是一个非常自然的过程。我编写的第一个编译器是为了 Pascal 的一个子集，然后，Turbo Pascal 就是 Pascal 的第一个近乎完全的实现。不过，Pascal 一直是作为一种教学语言，而且它缺乏大量的编写真实世界的应用程序所必需的相当普遍的特性。为了实现商业的可行性，我们不得不立即尝试用多种方式来扩展研究范围。

令人惊叹的是一种教学语言能在教学和商业之间牵线搭桥方面如此成功。

Anders：有很多不同的教学语言。如果你看一看 Niklaus Wirth 的历史—Niklaus Wirth 设计出了 Pascal 语言，随后又设计出了 Modula 和 Oberon——他一贯重视简单性。教学语言可以用来进行语言教学，因为它们擅长于教授一个特定的概念，但它们并不是真实存在的、真正能够作为你编程基础的成熟语言。这一直都是 Pascal 的作用所在。

学校也似乎分为两种流派。有些学校是以 Scheme 开始的，例如麻省理工学院。而其他学校则似乎侧重于“实际”。有一阵子，他们教的 C++。现在教的是 Java，有些时候也用 C#。您怎么看？

Anders：毫无疑问，我一直是“实践”派。我是一名科学家，但我更是一位工程师，如果你愿意这么分的话。我的理念是，如果你教人东西，那就教他们一些以后实际用得着的东西。

一如既往，这种回答也不走极端。答案就在二者之间。在编程语言不断的实践中，在行业的编程语言实现中，我们借鉴了学术界的观点。现在，我们看到了得益于函数式编程思想的巨大收获，而这些思想到底已经在学术界存在了多久，也许只有上帝知道。我认为这里的神奇之处在于你必须二者兼顾。

您的语言设计哲学是出自自己的观点并付诸实践的吗？

Anders：噢，在某种意义上是这样的。我认为你可能会从某些指导原则开始。简易性始终是一个很好的指导原则。另外，我是一个十足的进化论迷，而反对凡事从零开始。

你可能会痴迷于一个特定的想法，于是，为了实现它，你去创造一个全新的语言，而这种语言的妙处就在于它是一个新东西。然后，顺便说一句，90%语言都有某些差劲的东西。如果你还能够不断改进现有的语言，其中有许多类似的东西，例如，最近我们确实对 C# 朝着函数式编程方向做了很多改进。我觉得，这些都是意外所得。你有一个庞大的用户基础，它们恰好非常熟悉这些东西。这样会有一点复杂，但是毫无疑问，它要比为了追求某种特定的编程风格而必须学习全新的语言和执行环境要好得多。

297

要在语言本身及其生态系统之间划出一条界线，这是很难的。

Anders：嗯，是的，最近无疑是越来越是如此。如果回到二三十年前，你可以说语言支配着你的经验曲线。学习一种编程环境几乎就是学习这种语言的一切。然后，这种语言有一个小的运行时库。如果你能学到操作系统那里的话，操作系统可能还有一些内容。现在你看看我们所拥有的这些庞大的框架，比如说.NET 或 Java，

这些编程环境是由规模宏大的框架 API 所统治，以至于语言本身几乎成了事后才考虑的东西。这不完全正确，不过可以肯定的是，它更与环境而不是语言及其句法相关。

这会让库设计者的工作更加重要吗？

Anders：平台设计人员的工作变得非常重要，因为，你确实能最大程度地利用的就是你能否确保该平台的寿命以及在该平台上实现多种语言的能力，而这些则是我们一直看好的。.NET 是作为多语言平台开始设计的，而且，你看它现在驻留了各种不同语言：静态语言、动态语言、函数式语言，还有 XAML 这样的说明性语言等。然而，它们下面都是相同的框架、相同的 API，而这种影响是非常巨大的。如果这些都自成一体的话，你就会在互操作和资源消耗中慢慢耗死。

一般来说，您赞成使用多种语言的虚拟机吗？

Anders：我认为必须这样。依我看来，它就像你回到了 8 位系统那段美好的旧时光，你使用的是 64KB 内存。这些 64KB 内存会被填满，而且是很快就被填满。这不可能让你去构建可供多年使用的系统。

64KB 内存，您可以先实现一两个月；也许 6 个月，你就要用到 640KB。现在，它基本上是一个无底洞。用户的需求越来越多，而且我们也没办法全部重写。这就是要创建并利用互操作性。否则，你就永远只是做这种枯燥无味的事，只是在做最基础的工作。

如果你能给它们提供一个共同的基础，并在共享系统服务之外获得更高程度的互操作性和效率，那么，这就是一种方向。以托管代码与非托管代码的互操作性为例。它会遇到各种各样的挑战。但是，这样远比在每一个不同的编程环境来努力解决它要好得多。最具挑战性的是构建混合应用程序，它一半是托管的，另一半是非托管的，而且您需要在这一半进行碎片收集，而另一半则根本不需要。

在 JVM 的设计目标中，似乎从来没有准备要破坏向下兼容早期的字节码版本。这就限制了他们的设计决策。他们可以在语言级别进行决策设计，但在实际的泛型实现中，举个例子，他们不得不进行类型擦除。

Anders：你知道吗？我认为，他们的设计目标不仅是向下兼容。您可以添加新的字节码，并且仍然向下兼容。他们的设计目标是根本不动任何字节码和虚拟机。这有很大的不同。实际上，设计目标没有变化。这就把你完全限制住了。在.NET 中，我们有向下兼容的设计目标，所以增加了新功能、新的元数据信息。.NET 2.0 会有一些新指令、新库等，但.NET 1.0 的每一个 API 都能在.NET 2.0 继续运行。

我一直很困惑他们为什么选择了这条道路。我可以理解它为什么让你这样做，但是如果你从本行业的历史来看，一切都在发展变化。一旦你停下了发展的脚步，就等于是给自己判了死刑。这只是一个时间问题。

我们选择的是具体化泛型而不是擦除，对于这种选择，我极为满意，而且获益良多。我认为，我们用 LINQ 完成的所有工作离开具体化泛型几乎是不可能的。我们用 ASP.NET 做的所有的动态的东西，在几乎每一个实际产品中生成的所有动态代码之所以能够如此深入，都是受益于这个事实：泛型是真正地在运行时表示，而且在编译时间和运行时环境之间存在着对称关系。这就是如此重要。

对 Delphi 的批评之一是，它强烈不愿意中断通知语言决定的代码。

Anders：让我们再退一步。当你说到中断（打破）代码时，你必须首先谈谈它的发展。你在谈论它的 N+1 版。你应当承认，有时中断（打破）代码是件好事，但总体上概括来说，我就不能为中断找到什么托辞了。因为他们并没有很好的理由，我所听到的支持中断的唯一理由是：“这是更整洁的方式”、“它在结构上很完

美”，或者是“它会给我们的未来作好准备，我们的未来”等。那么我说，你知道，也许平台只能生存 10 到 15 年，随后它们就会由于这样或者那样的原因而不堪来自自己的重负。

他们早晚会成为遗留系统，或许需要 20 年。到那时，它们周围就会出现足够多的新东西，还有很多的新东西不需要任何开销。如果你想要去中断它，那就好好去中断吧。中断一切。要接触最前沿的技术。而不要四处分散精力，那样毫无意义。

这听起来好像是一种需要 5 年或 10 年轮换一次的跳背游戏。

Anders：你要么玩跳背游戏（译注1），要么就好好地把握向下兼容性，并且每次都要带上你的整个团队。

托管代码只能在一定程度上起作用。你可以在这个过程中使用已有的组件。

Anders：当然，从.NET 一开始，我们就已经在每一次发布中都保持向下兼容。我们修正了一些造成代码中断的 bug，但我的意思是必须有一些定义，通过这些定义中断人们的代码是不错的。

不错，有时在安全或者修改程序行为的名义下，我们会中断代码，不过，这种情况非常少见，而且一般来说，它揭示了在用户程序中的设计错误，或者是他们本来准备修复的那些错误，因为以前尚未意识到这里存在问题。如果是这样，当然很好，不过，如果是要以让代码更完美的名义来进行不必要的中断，我认为是一个错误。我在早年做过很多这样的事情，后来才知道这对于客户来说是行不通的。

至于说什么是好风格确实很难判断。

Anders：是啊。对我来说是好风格，而对你来说就未必是什么好风格。

如果回顾一下您涉猎过的语言，从 Turbo Pascal，到 Delphi、J++、Cool，再到 C#，那您的工作有没有什么主题呢？我可以听早期的莫扎特乐曲，然后再听他的安魂曲，我会说：“这些显然都是莫扎特的曲子”。

Anders：事事都是你所处时代的写照。我是伴随着面向对象之类的东西长大的。可以确定的是，从 Turbo Pascal 的中期到目前为止，我的工作核心一直是面向对象语言。那里发生的很多演变现在都已经发扬光大了。在 Delphi 中，我们在面向组件的模型上面做了大量的工作，比如说属性和事件等等。

有关这一点，我也把它用在了 C# 上面，这肯定是公认的。我试图一直把握社区的最新动态，力争与时俱进。没错，Turbo Pascal 是创新性的研发环境，Delphi 是可视化编程，而 RAD、C# 和.NET 则都与可控执行环境、类型安全等有关。你从身边的东西中学习，让它进入你的生态系统或竞争性生态系统。你确实想取其精华，去其糟粕。在这个行业里，我们都是站在巨人的肩膀上的。它的魅力在于，相对于我们所见的硬件发展，编程语言的发展是何其缓慢。结果令人大吃一惊。

从 Smalltalk- 80 开始，我们已经经历了 15 或 20 代的硬件！

Anders：实际上，硬件是每大约 18 个月更新一代。然而，我们现在使用的编程语言和 30 年前的构想之间并没有巨大的差别。

译注1：跳背游戏：一个游戏者跪下或弯腰，队列中紧接着的人从其身上跃过。

他们仍在喋喋不休地争论那些旧概念，比如说 Java 中的高阶函数。这种争论可能会持续 10 年。

Anders：这很不幸，因为我觉得他们应该在这个问题上进展得更快一些。我认为问题并不在于它是否有价值，而在于 Java 社区完成它是否有太多的流程和开销。

如果要在语言级别采用 CPS（译注2）和揭露“Call/cc 调用”，就会给你提供一个巨大的优势，你会怎么做呢，即便只有 10%的程序员可能会理解它？

Anders：如果假设是的，那么这也是一个很大胆的假设。我认为并非如此，不过要看看我们使用 LINQ（译注3）做了些什么。我确实相信，这将有益于绝大多数的 C#程序员。能够跨不同的数据域，编写出更具声明式风格的查询，拥有一种合适的通用查询语言，这是它的最大价值所在。在某些方面，这就像 Holy Grail 语言和数据库集成。我们可能并没有解决整个问题，但是我认为我们取得了足够的进展，它证明了必须进行额外学习，同时，也有方法可供人们使用，而不必掌握来自第一原理（译注4）的 lambda 演算。

我认为这是函数式编程的一个很好的实际应用例子。你可以愉快地使用它，甚至不需要知道正在做函数式编程，或者有函数式编程原理在背后提供支持。我对这种结局非常满意。

您用了“实际”这个词。您是如何决定要添加哪些功能，拒绝哪些功能？在添加和拒绝之间，您采用什么标准呢？

Anders：我不知道。随着时间的推移，你就会有这种判断能力：相当多的用户是否会从它所创建的概念化的过时观念中获益，对不对？相信我，我们的用户群给我们提供了大量的建议：“哦，如果我们只能做到这一点，”或者是“我确实喜欢做这些”，不过，有些时候，它太过于狭隘地关注解决一个具体的问题，而且作为一种抽象概念，它并没有带来很大的增值。

毫无疑问，最好的语言通常都是由小团体或者个人设计出来的。

语言设计和库设计之间是否存在区别呢？

Anders：确实有很大的区别。API 显然要比语言更具有领域特定的特点，而且如果你愿意这样说的话，语言确实要比 API 高一个抽象级别。语言制订了 API 设计框架，如果你愿意的话，甚至可以到夸克、原子和分子等细节。它们指示你把 API 整合在一起，而没有指示你 API 做什么。

在这个意义上，我认为二者区别很大。实际上，这又回到了我以前想要谈的内容。每当我们考虑为语言添加一个新特性时，我总是尽量让它适用于多个领域。优秀语言特性的标志就是，你可以以不止一种方式来使用它。

这里，我再次举 LINQ 为例。如果你把使用 LINQ 做的工作分解一下，它实际上是大约六七种语言特性，包括扩展方法、lambda 演算和类型推理等。然后，可以把它们放在一起，创造一种新的 API。特别是，如果你

译注2：所谓延续（continuation），是指函数式编程中的一种函数调用机制。CPS（continuation passing style）是指延续传送风格。call/cc 调用（call with current continuation）是指取得当前的延续，传递给要调用的这个函数，这个函数可以选择在适当的时候直接返回到当前延续。

译注3：LINQ，即语言集成查询，它是一组技术的名称，这些技术建立在将查询功能直接集成到 C#语言（以及 Visual Basic 和可能的任何其他.NET 语言）的基础上。借助于 LINQ，查询现在已是高级语言构造，就如同类、方法、事件等等。对于编写查询的开发人员来说，LINQ 最明显的“语言集成”部分是查询表达式。

译注4：第一原理（first principle），一个系统研究中的基本原理、规则或法则；该系统或体系的其他原理、规则或法则都是从它那里推导出来或从它那里得到解释，而它本身却不是从那个体系或系统中的任何其他原理或规则推导出来或得到解释的。数学公理和逻辑原理被认为具有第一原理的资格。

愿意的话，可以把创建的这些查询引擎作为 API 来实现，但语言特性本身对其他方面也是非常有用的。人们使用扩展方法来处理其他有用的东西。局部变量类型推理等特性也非常好用。

如果我们说："让我们只是将 SQL 塞进那里或者完全是 SQL 服务器特定的东西，而且只是谈论数据库，然后想要使用 LINQ 这类的东西"，我们可能会更快一些安装 LINQ 之类的工具，但是它还没有那么通用，尚不足以成为通用编程语言的一部分。你很快就会成为一种领域特定的编程语言，而且你会同那个领域生死攸关。

你把自己很好的第三代编程语言变成第四代编程语言（译注5），这是一种通用的消亡之路。

Anders：是的。我对此非常清楚。现在，我们正在考虑的一件大事就是并发。每个人都在考虑并发，因为他们不得不考虑这个问题。这不是想不想考虑的问题，而是必须考虑的问题。此外，在并发领域，我们能让语言指定一种特定的并发模型，但是，这是一种错误的做法。我们必须从整体的角度来审视，找出语言缺乏什么能力，而这种能力会使得人们能够实现用于并发的大库和大的语言模型。我们需要这种语言要有一些处理方式，能为我们提供更好的状态隔离。我们需要功能的纯洁性。我们需要核心概念的不变性。如果你可以把那些作为核心概念添加进去，那么我们可以把它交给操作系统和框架设计者进行不同的并发模型实验，因为他们都需要这些东西。然后，我们不必猜测到底谁是赢家。相反，我们可以相当轻松地知道结果：一方会大为光火，由此证明另一方更为成功。

我们仍然与此有关的。

这听起来好像是你想给人们提供工具来构建伟大的事情，而不是您指定他们要构建什么。

Anders：我是想这么做。那样你就可以更好地利用社区创新。

您是在 C#社区的什么地方看到的？大家会带给您代码吗？您去拜访客户吗？您会让您的 MVP（译注6）在新闻组和用户组中闲逛吗？

Anders：它是以上几种方式的混合，再加上一些别的方式。我们有 Codeplex 这样的代码共享 XX。社区是多种多样的，还有商业性社区。既有开源的，而且还有很多的.NET 开源代码。它们的来源五花八门。可以这么说，我认为并没有单点的汇聚。那边有一个多样化的复杂生态系统。

无论你走到哪里，你总会偶然发现一些东西，"哇，他们是想到这些的？"或者是，"这真让人吃惊"。你会意识到这会有多大的工作量。它可能在商业上并不可行，但是，它是一件很好的工作。

302

我的确在尽力密切关注与 C#和 LINQ 相关的很多博客。

这些是我最喜欢的一些关键字，当我去博客闲逛时，只是去那儿看看发生了什么。它为你提供了很好的洞察力，看看人们是否已经熟悉了你的工作，无论你是不是通过正确的方式完成的。它教会了你面向未来的一些东西。

译注5：在计算机行业中，通常用几代来表示编程语言的发展。它是按照语言的抽象程度（即和自然语言的接近程度）划代的。第三代编程语言（3GL）是一种"高级"编程语言，例如 PASCAL、C 都属于这一类。而第四代编程语言（4GL）则包括 VB、C++等。访问数据库的语言通常都是第四代。

译注6：MVP（最有价值专家）是指具备一种或多种技术专业知识，并且积极参与在线或离线的社区活动，经常与其他专业人士分享知识和专业技能，受人尊敬、信任，而且平易近人的专家。

13.2 培育一种语言

Growing a Language

您是如何看待简单性的呢？

Anders：存在真正的简单性，而且，还有我称之为 simplexity 的特性，这种特性我见过很多。当你第一次建立超级复杂的东西时，就会说，“哇，人们永远无法实现它。这实在是太复杂了，不得不为此竭尽全力。但我们必须要有所有这些功能。让我们来试试另外构建一个简单的系统吧。让我们来试试能否仅在一个简单的界面里就把所有东西都包含进去”。

然后，你必须做的事情并不完全是系统被设计成什么样，梆梆！你陷入底层复杂性的大泥潭中，因为你所考虑的全部只是非常复杂的事物的浅层表面，而不是真正简单的东西。我不知道这是否会对你有启发，但我倾向于认为它就是这样的。简单性往往意味着你事半功倍。虽然只是花费很少的精力，但它的功效却能同其他做法一样或者是超过其他做法。这是事半功倍的精要。这不是用更多东西完成更多的事情，而是使用一个简单的表层。

如果您现在要创建一种新的编程语言，您会遵循这个原则吗？

Anders：噢，肯定会的。到目前为止，我已经创建了多种编程语言，当然还肯定会有许多实现。在着手创建新语言之前，你必须非常非常清楚你为什么要这么做以及你要解决什么问题，我认为这是非常重要的。

通常，在新编程语言方面，人们爱犯的错误是他们执迷于某个特定的问题。或许这种编程语言可以正确地解决那个问题，因此他们着手解决这个问题的一部分，也许他们会干得非常出色。那么，每一种编程语言，我的意思是每一种编程语言，都是由 10%的新东西和 90%的编程支撑要素组成的，而这些支撑要素又是必需的。在这些新编程语言中，我们所看到的这些创新的解决方案都是这 10%的东西，但另一方面，为了让你真正能编写程序，每种语言都必须有的那 90%又非常糟糕，因此，他们失败了。

每一种编程语言都必须有许多枯燥的标准要素，理解这一点是非常、非常重要的。如果你没有正确地理解它，你就会惨遭失败。相反这意味着，如果不是创建一种新的编程语言，你可以在现有的编程语言上进行发展演进，然后这种“综合的产品”看起来很不同，因为你已经包含进去了 90%的内容。事实上，你已经包含进去了 100%的内容。你只不过想往里面添加新东西而已。

就像 C++一样。

Anders：就像 C++一样，它就是 C 或者是我们已经完成的 C#不同版本等演变而来的一个很好的例子。我是非常相信发展演变的人。当然，有时候，你就不能往那里添加更多的东西，在你添加的新东西和语言中无法移除的旧方式之间需要大量的平衡。创建一种新语言确实是规则的例外多过规则自身。

您会创建一种通用语言或领域特定的语言吗？

Anders：我认为，真正的答案是“都不会”。我非常想通过创建一种通用编程语言来解决问题，而这种语言擅长于创建领域特定的语言。再者，使用这种领域特定的语言，我们面临的难题是：它们在领域内表现很出色，但将它们作为通用语言则非常勉强。各种领域特定的语言确实都需要一些通用的特性。除非这种领域特定语言只是一种纯粹用于描述数据的数据定义语言，那样的话，依我看还不如干脆使用 XML。

如果你是拥有逻辑、谓词、规则等的真正的编程语言，你就必须有各种表达式，表达式还必须有运算符，或许你还必须有标准函数，而且你的客户想要完成你从来没想过的事情。还有你需要的一大堆标准要素。如果你能在通用编程语言的基础上创建领域特定语言，那么我想这比你每次都从头开始要好得多。

现在，通用编程语言面临的一个难题是它们在创建内部 DSL 方面变得越来越好，你可以看看 LINQ 这个例子，但是它们目前并不擅长获取这些内部 DSL 的正确使用模式。在某些方面，当你创建内部 DSL 时，实际上你想要限制可以使用通用编程语言来做的事情。您希望能去除语言的通用性，而且你只想让 DSL 具备这种功能。现在，通用编程语言在这方面并不擅长。这也许是值得考虑的事情。

Brian Kernighan 说，如果你想创建一种通用语言，你就应该从铭记这一目标开始。否则，如果您创建一个小的语言，一旦人们开始使用它，他们就会要求添加新特性。培育一个 DSL 效果通常不会很好。

Anders：噢，是的。我想起来 Gosling 说过，每一个配置文件最终都会成为它自己的编程语言。这话说的很对，你应该认真对待。

你说过，在某些方面平台比语言更重要。那我们能生成可重用组件吗？

Anders：哦，我说这话的原因是，如果你回顾一下语言、工具和框架在过去的 25 年、30 年中的发展，那你就会对编程语言只有很小的变化感到诧异。同样让人感到诧异的是，我们的框架和运行次数是如此之大。较之 25 年、30 年前，它们可能是增大了 3 个数量级。我最初做 Turbo Pascal 时，运行时库或许只有 100 到 150 个标准函数，当时情况就是这样。而现在，.NET 框架有 10 000 个类型和 100 000 个成员。显然，利用好这些资源变得日益重要。因为它重塑了我们思考问题的方式，而框架变得日益重要，则是因为正是它改变了我们的编程方式。

现在，“利用”就是一切。从编程的观点来看，你的计算机基本上就是一个无底洞。你从现在开始写代码，就是一直写到死，你都不能填满它。它的容量很大，而最终用户的期望会变得越来越高。你真正能获得成功的唯一一条路就是寻找一些利用现有资源的灵巧方法。如果你回到 25 年或 30 年前就不是这种情况了。你只有 64k 内存可用，好的，加油，你只需要一两个月时间就能把它填满。

语言会对程序员的生产率造成多大的影响呢？程序员的能力又能造成多大的区别呢？

Anders：我认为这两种因素是同时起作用的。我认为语言会影响我们的思考方式。如果你愿意的话，程序员的工作就是去实现这一想法。那就是原始素材、原始动力在推动这个过程。是语言在重塑你的思考：它的作用实际上是帮助你以一种高效的方式思考。例如，支持面向对象的语言如何促使你以某种特定的方式来思考问题。函数式语言促使你以另一种方式来思考问题。动态语言可能促使你以第三种方式思考问题。不同的语言可以促使你进行不同的思考。有时尝试从不同角度来处理问题是很有益的。

你会更喜欢添加一种语言特性让每个人都提高一点效率，还是喜欢让少数开发者变得更加高效呢？

Anders：对于一种通用编程语言来说，只为帮助少数人而添加特性并不是一个好主意，因为你最后会成为一个装满奇怪东西的杂货包。所有优秀语言特性的标志是，它有很多优秀的应用，而不只是一个。如果你回顾一下我们在 C# 3.0 中添加的所有东西，它们整体构成了所谓的语言集成查询（即 LINQ）的概念，实际上，它可以分解为六七个互不关联的语言特性，而且每一个都有很多优秀的应用。它们并不仅仅有益于某个特定的程序员。它们处于一个更加抽象的层次上。对于每一项优秀的语言特性，你必须能够展示它能让你在某些

场景下受益，否则它就不适用于该语言。或许让它成为一个 API 特性会更好一些。

你认为添加或者删除哪些特性会使调试更容易？在语言的设计过程中你考虑过调试的体验吗？

305

Anders：噢，绝对是这样的。如果你看一看整个 C#的基础，这种语言是一种类型安全的语言，它意味着没有数组溢出或者野指针（译注7）的现象。一切都有定义好的行为。在 C#中就没有未定义行为之类的事情。错误处理是通过抛出异常来完成的，而不是返回你可以忽略的代码。因此，像类型安全、内存安全和异常处理这样的每一种支撑都非常有助于消除整个 bug 类，或者是更易于发现整个 bug 类。那是我们一直都在思考的事情。

您是如何在不限制开发者的前提下预防重复性问题的？在开发者的安全性和自由度之间，您是如何选择的呢？

Anders：我认为，如果你愿意的话，每一种语言都会在某一范围内具有影响力和生产力。C#肯定是一种比 C++更安全和更受保护的语言环境，比起你正在编写的汇编代码，C#将会更加安全和更加高效。纵观整个编程语言史，编程语言的普遍趋势是：保持抽象层的较高水平以及促使编程环境更加安全，或者是把程序员必须要做的越来越多的事务性工作交给机器来处理，让程序员主要关注过程的创造性部分，这样才是真正地提升他们的价值。程序员们非常讨厌把内存管理作为一种规则。他们同样非常讨厌类型安全分析；因此，我们就有了 bug。

在某种程度上，我们可以把它交给机器来完成，而让程序员放手进行创造性的思考，我认为，这是一种很好的折中平衡。它的代价只是一小点的性能损失，不过小伙子，它并没有损失太多。今天，在一个典型的.NET 应用程序中，如果你观察一下程序的执行概况，并看看程序的时间花在了什么地方，甚至很少会看到垃圾收集。然而，你的程序是安全的，而且也没有内存泄漏。这是一种很妙的折中平衡。比起在 C++或者 C 中这些人工内存管理系统来说，这一点真是棒极了。

我们能够使用一种科学的方式来设计和改进语言吗？我能够看到实现的研究结果给出的改进，不过语言设计听起来好像与设计者的个人喜好有关。

Anders：我认为编程语言设计是艺术和科学的有趣组合。很显然，这里面有大量的科学，比如用于分析、语义、类型系统以及代码生成等的符号数学形式体系。这里面融合了大量的科学和工程知识。

而且，它还是一项艺术。这种语言感觉像什么呢？当你用这种或者那种语言编程，脑子里想些什么呢？它们又有怎样的不同呢？对于人们来说，什么更容易理解，什么很难理解呢？我认为我们并不能度量这些东西。

它从来不是很科学的。它总是从纯艺术的角度来看待语言的设计。就像绘画也有好坏之分一样，你只能从某种程度的科学上说："当然，这种组合不一定合适。或许他没有用对颜料"。但这最终是仁者见仁，智者见智。这只是一些你无法形式化的东西而已。

306

译注7：野指针（stray pointer），也就是指向不可用内存区域的指针。通常对这种指针进行操作的话，将会使程序发生不可预知的错误。一般有两个原因：1. 指针变量没有被初始化。2. 指针被释放或者删除之后，没有置为 NULL，让人误以为它是合法的指针。

您至少会说两种语言，您认为这在某些方面会有帮助吗？有时意大利语只用一个单词就可表达某个概念，而英语却得用一句话，显然反之亦然。

Anders：我不知道。这个问题问得很好。我从未想过这个问题。或许是吧。我当然会认为成为一名优秀的语言设计者必须要懂得多种编程语言，这一点是毫无疑问的。至于说它是否有助于理解多种口语，我真的不知道。这两者应该联系得很紧密。在设计团队中，我们可以肯定有人可以说很多语言，并擅长于音乐。他们好像存在着某种联系，不过，我并不确定它们是如何联系的。

13.3 C#

C#未来还能存在多长时间呢？你已经说过是 10 年左右。

Anders：C#项目始于 1998 年的 12 月份，因此，我们正在讨论 10 周年纪念。这并不是说已经在行业中存在了 10 年，而是从我们内部开始做的时候算起来是 10 年。我认为我们至少还会使用 10 年，不过这要视情况而定。我说过，我已经不再对这个行业遥远的未来进行预测，因为从来没有人能预测准确。不过，我的确看到了 C#繁荣而又健康的未来。我们并没有创新，但我们仍有大量的工作可做。

从应用领域立场来回顾 C#的发展，我看到有这样的需求：C#很有希望取代 C++语言成为一种系统编程语言。

Anders：它的确可以应用在那方面，不过，在更适合使用.NET 或 Java 等语言的可控执行环境中，C#也有很多应用。

我拿 C#与 Java 做了一下对比，结果发现，C#的发展动力似乎更为强大。Java 那些人看起来好像想要一个基准，在这条基准上每个人的代码都差不多是相同的。无论是你已经编了 10 年 Java 程序，还是在之前从未编过程，或者只是刚上了六个月的 Java 培训班，你们的代码都统统差不多。C#看起来从 Haskell 或者 F#中获得了一些新观点。是否要添加这样一种特性，即使是上完 6 个月 C#课程的人们也无法马上理解？

Anders：我这么说并不是表明是要设计下一个 COBOL；我们只是借用这种表达方式而已。

我们看到因特网革命和电子技术革命的动力何在呢？动力在于我们在不断发展的这个事实。让我来回顾一下这种进步。一旦你停止了发展，我不知道你还能增添什么价值。再者，这是从极端的观点来看的。当然，平台的稳定性很有价值，不过，我认为你是通过确保向下兼容来提供的这种价值的。在 C#1.0 中，你可以随便下车但不能走开。对于那些真正想要更高效率或者想要构建更新的（比如说 SOA 一类的）应用程序的人们，或者更具动态编程风格（比如可改编程序）或者更具声明式编程风格（比如说我们正在使用的 LINQ）的人们来说，要么你就改进这种语言，要么你就放弃这种方式，否则就会被别的东西取代。

你收到过关于 C#语言的反馈吗，而不仅仅是实现？

Anders：我们每天会获得关于这种语言的不同方式的反馈。可能是人们给我发邮件。我阅读人们的博客，浏览人们讨论技术性问题的论坛，我还会参加会议，这样，每天会通过各种各样的方式获得有关该语言优劣的反馈。我们会把这些反馈给设计团队，而且我们还维护关于这些疯狂想法的长长的详细清单。其中有一部分从未添加到语言中，不过还是把它们维护在列表中，因为或许有一天会从其中获得一些好主意。我们知道虽然做得并不到位，但还是愿意做事的。

随后，我们逐渐地找到了问题的解决方案。其中一些只是人们讨论的简单问题，那恰好也是我们要解决的。另一些确实是人们从未谈论过的大问题，比如说 LINQ 等。这不像有人曾经问过我们的："我们喜欢把查询嵌入到语言里"，因为你并没有真正考虑这些你能考虑的概念。

我不愿意说我们从某种特定方式获得反馈。这是一个有机的过程，我们可以从很多不同的地方得到它们。毫无疑问，如果没有这些反馈，我们就无法设计出这种语言，因此，这一切都是以倾听产品使用者的意见为基础的。

您是如何管理设计团队的？你如何做出决策？

Anders：首先，你从用户获得反馈时，通常用户会告诉你，"如果你能够添加这个特定特性，我们会非常喜欢"。随着你的深入了解，结果是，哦，他们试图又想做这个还想做那个，而且典型地，人们还会告诉你他们的解决方案。现在要做的是发现他们的实际问题，然后设法把它们融入到更大的语言框架中。从某种意义上来说，获得反馈的第一阶段是去侦探问题所在，并理解隐藏在客户要求的解决方案下的真正诉求。他们的真正问题是什么？

然后，我才考虑决定以何种方式来解决它。在改进一种语言时，你必须始终认真对待强加到语言中的一大堆功能，因为你添加的功能越多，你的语言就老化得越快。最终，语言会因不堪自己的重负而被淘汰。那里东西太多了，而且很多东西是相互冲突的。

你必须非常非常明智地对待添加的东西，因为实际上，你并不想只是出于历史原因而使用三种方法来完成同一个功能。

因此，有很多次我们会说，"是的，如果我们能够重新开始，我们肯定会立刻加上人们要求的这些特性"。既然我们无法重新开始，我们就不能这么做，因为这是非常基础的东西，我们不能通过接连出击而从根本上改变它的属性。我们唯一能做的是使它变为双头兽，而这并不是我们想要的结果。

就设计过程自身而言，我们拥有一个非常强大的 C#设计团队，它由 6~8 名定期开会的成员组成。在过去的 10 年里，每周一、周三和周五下午的 1:00 到 3:00 都要定期开会。其中有一些会议取消了，不过在 10 年里，我们都在日程表里给它留出了时间，而且这种情况还会继续下去。在这个过程中间，人员也有变动。我自始至终参与了这项工作。Scott Wiltamuth 也差不多是这样。其他人进进出出，变化不定，不过这个过程已经存在了很长时间。

我们把这个当作我们的设计职责。这就是我们一直在进行设计的原因。为了保持产品的连续性，把设计当成一个持续的过程是非常重要的。通常人们会突然拼命努力去干，"噢，该做下一版了。我们开个会讨论一下该怎么干吧"。然后，就没完没了地开会，然后人们纷纷离开，最后在这一年里没有做任何的设计。一年过去了，又到了开发下一版的时间了，你连最初的那拨人都凑不齐了。最终只能发布一款令人抓狂的产品，而每一版发布都会让人感到面貌各异。如果你继续进行设计，你发布的产品几乎就像是一个个性化的产品。

灵感乍现可没有什么时间表可言。它们只会偶然迸发。如果你并没有一个获取灵感的过程，如果你没有马上进行设计，那么或许这种灵感就会消失。我们总是不断地对将要开发的下一版进行持续的探讨。我认为这么做的效果确实很好。

C#有一个 ECMA 标准化流程（译注8），这在语言中是很罕见的。这样做的动机是什么呢？

Anders：对于很多人来说，标准化是采用技术的一种要求。毫无疑问，在很多地方（不仅仅是企业），如果你看一看政府部门，标准化确实是一种需求。学术界也是如此。实际上，对于微软的标准化来说，它有很多令人瞩目的好处。无论何时，只要是我们构建像.NET 这样的技术，它总是能够在第三方提供技术的其他平台上实现，而且，你可以选择让它们随意地复制你已经创建的东西，并发现故障，然后获得一些糟糕的经验。这对那些客户也意味着是糟糕的经验，这些客户基本上都依赖于你的那些东西，不过要在他们所有的遗留硬件上进行另外的实现。

当你把这些都汇总起来，实际上即使是从商业的观点来看，做这些也都是有意义的。你所构建的东西本质上有很多优点，它也是作为在这些方面非常精确的优秀强制性函数而存在。事实上，我们标准化 C#就意味着我们必须编写一个非常简明、非常精确的语言规范，仅仅是从内部的立场来看，这项工作的投资已经给我们带来了许多倍的回报。

就来自质量保证部门的更好的测试框架而言，以及就针对于实现新语言特性的更好的研究手段而言，对于原型编译器，它们所期望或要求的东西是非常清晰的。针对该语言的可教性，非常简明的规范意味着人们总能把它当作参考资料来查阅，否则就只能猜想了。

它有助于确保代码的反向兼容性。因此不管怎么说，可能会有很多你无法立即想到的好处，不过实际上它们确实存在。通过标准化过程，你可以让见多识广的社区来关注你的产品。我们已经获得了参与标准化过程的其他公司和个人的大量反馈，而且，这样也使得 C#变得更好。那也是很有价值的。我不能确认如果不是因为我们的标准化工作，这些组织和个人是否会对它感兴趣。

然而，标准化会滞后于语言的发展。

Anders：是的。在某种程度上，标准化会减慢你的速度。这还得看你怎么说了。有些标准可以说成，“你必须把这个实现了，而不用实现别的，而且如果要扩展我们指定的东西，那就违反了标准”。我从来不相信那些。标准的目的或者是被要求建立一个通用基准，而且毫无争议的是，还要有一种方式确保你在遵从这条基准而没有逾越它。但是，标准肯定应该为创新留出自由空间，因为你要通过那些创新来制订 V2 版本的标准。你不能宣告它是非法的。

对于 C#，它有一个标准，不过这个标准并没有使我们远离发展。一个创新的过程不会发生在标准过程之外，因为你不会跳出标准社区而获得认可。那不是它的目的。不管你工作在什么框架上都必须允许创新随时出现。

在语言设计的形式方面您持何种观点呢？一些人建议您应该首先制订书面的形式规范，然后再编写代码。一些人恰恰完全忽略了形式规范。

Anders：答案是尽量不走极端。我认为，如果根本没有形式规范，那么该语言通常就会乱成一团。如果首先使用形式化方法指定一切，然后再事后实现编译器，这样的语言也会很难使用。我们开发 C# 的方式是同步编写编译器和语言规范，而且二者还相互深入地影响渗透。我们在编写编译器时可能会偶然遇到一些问题，这时候必须回过头来在规范中解决。或者，在编写规范时尝试严格分析所有的可能，“吁!! 或许我们应该在编译器中试试别的办法，因为有很多我们没想到的其他情况”。

译注8：ECMA 是“European Computer Manufactures Association”的缩写，中文称欧洲计算机制造联合会。它是 1961 年成立的旨在建立统一的电脑操作格式标准——包括程序语言和输入输出的组织。ECMA 标准是 C#语言所有功能的官方说明。

我认为这两点都很重要。我们做了标准化工作，我对此非常高兴，因为它促使我们更清楚地把握了语言是什么和它如何工作这两个问题。它还促使我们拥有了一个形式化规范说明，而这一点，正像你所说的，恰恰是一些语言没有的东西；那并不是一件好事。如果源代码就是“规范说明”的话，那就意味着，一旦你想要搞清楚这个特定的程序将会发生什么，你就必须去看一看要编译的源代码。并没有多少人能做到这一点。唯一的替代方案就是靠猜测，或者是编写测试用例来看看结果，并寄希望于找到所有的边界条件。我认为这不是正确的方式。

顺便问一句，您如何调试 C#代码呢？

Anders：我主要使用 Console.Writeline 这个调试工具。说句老实话，我认为很多程序员都是这么干的。对于更复杂的情况，我会使用一个调试器，因为我要看一看堆栈踪迹或者是本地出了什么问题等。不过通常情况下，你只要简单地检查一下，就能够很快地摸清底细。

设计 API 时，您遵循了什么原则吗？

Anders：噢，首先我会说要保持它们的简单性，不过这将意味着什么呢？我的意思是说这听起来很愚蠢，对吗？我极为推崇方法和类都尽可能少的那些 API。有些人相信越多就会越好。我不是这么想的。我想，看看你认为人们会怎么样来使用你的 API 是非常重要的。找到五种典型的使用情景，然后确认这些 API 会使事情变得尽可能地简单。理想的情况是它只不过是一个 API 调用。你不应该为了典型的情景而必须编写很多行与 API 无关的代码。在这一点上，它并没有处在一种恰当的抽象层次上。

不过，我还认为在 API 中忽略一些东西是重要的。你想要从简单地使用 API 开始，然后无缝地移动到你需要的更高级的应用中。许多 API 有这样的渐进功能。没错，你只能调用一些简单的方法，不过，当你想完成稍微高级一点的功能时，突然，你却一败涂地。现在，你必须为未来高级一点的功能去学习一些与你无关紧要的东西。我非常推崇这种循序渐进的方式。

文档怎么样呢？

Anders：总的来说，软件的文档状态是很糟糕的。我一直力劝程序员并且也在内部一直呼吁，但应者寥寥，不过我告诉程序员，对于你交付给用户的 API 来说，一半的功劳要归于优秀的文档。如果没有文档来解释 API 是什么、应该如何使用它，那么再出色的 API 也毫无价值可言。这是一个难题。许多公司喜欢让程序员编写代码，而让其他人来编写文档，而且他们两者之间没有什么沟通。最后，你的文档上面只是写着“ MoveWidget，移动窗口小部件（widget）”，或者对一件显而易见的事情却啰里啰唆地说上半天。这真是让人啼笑皆非。我认为，比起他们编写的代码来说，程序员应该编写更多的文档。

您喜欢在代码内添加注释，或者你考虑过使用外部文档吗？

Anders：我一直倡导在代码中添加 XML 文档注释。如果你把它放入代码，使用它的程序员就有机会注意到文档注释说得不对的地方。或许他会把它修改好。如果你把它放在别的文件里，那么程序员绝不会再看它，因此它永远也不会改正。

大家都试图使它们尽可能地接近。尽管没有绝对的完美，但是我们在一直朝着这个目标努力。

对于成为一名优秀的C#程序员，您有什么建议？

Anders：这是很困难的。在C#编程方面有很多好书，而我只是鼓励人们从中选出一本来看。我在这里不会点名，不过，有很多好书会帮你成为一名优秀的C#程序员，并且帮助你更好地理解.NET框架。在网络上也会获得很多有用的东西，比如说Codeplex等。你能获得大量的开放源代码项目供研究和学习。

一般来说，对我成为一名优秀的程序员有帮助的是了解不同的编程风格和不同类型的编程语言。在过去5到10年里，我通过研究函数式编程学到了很多东西，这是一种截然不同的编程方式，不过它能教会你很多知识。显然，对于编程来说，这是一种不同的角度，而且它会让你以一种不同的观点来看待问题，我发现这是非常、非常有用的。

13.4 计算机科学的未来

The Future of Computer Science

您认为计算机科学的突出问题是什么呢？

Anders：如果你是在一个元层次上来看待这个问题，语言发展中一如既往的趋势总是抽象层次的不断提升。如果你回顾一下从外部指令（plugboard）、机器代码、符号汇编程序、C、C++一直到现在的可控执行环境等的发展，我们迈出的每一步都关乎抽象级别的提升。最重要的挑战总是去寻找下一个抽象级别。

现在，最大的挑战好像是有若干个竞争者。一个是我们已经说过的并发性：创建可用于并行的有意义的编程模型，并能够被各个层次的编程者所理解，而不是只为少数权威（high priest）而提供的平行性。我们现在所处的世界正是这样。现在，甚至权威们也会为他们自己的代码感到吃惊。这是我们面临的一个很大的挑战。

最近在社区里也有大量关于领域特定语言和元编程的讨论。在我看来是说得多干得少。我认为我们并不知道答案。你把事情看成是面向方面编程和意图编程（intentional programming），不过我们必须真正彻底了解它。

人们可能会说“噢，没有领域特定的语言”或者是“噢，领域特定的语言无处不在”，这要取决于你问谁。在某种意义上来说，我们甚至无法就特定领域的语言是什么达成共识，但是很显然，它为更清晰无误地表达你自己提供了更多的方式。在某些方面，我们已经用尽了命令式编程风格所提供的所有方式。我们并不想用我们的命令式编程语言实现更多功能。它不可能像我们添加新语句那样，使我们的效率突然提高10倍。

我认为当今的大多数编程语言都是这样的，它们会强制你过于详细地描述问题的解决方案。你得编写嵌套的for循环和if语句等等，而你只不过是想将两段数据连接起来。不过它不允许你这么干。你不得不委曲求全，开始做哈希表和字典，等等。

问题是我们如何转移到更具说明性风格的编程上来。当然，你的层次提升得越高，你最终拥有的概念就越多，因为你的领域特定性更强。对于领域特定的语言的这种梦想早已是老生常谈，然而我们还没有找到合适的方式来实现它们。我的感觉就是如此。因此，这仍然是一个挑战。

现在，我们看到了动态编程语言的复苏。实际上，我感觉确实是这样，尽管是语言的动态性越来越强，因为它们拥有强大的元编程能力。正如你看到的Ruby on Rails那样，Ruby的元编程能力让它的功能变得更加强大，而不仅是只有动态性。恰好，动态语言中的eval函数和元编程要比在静态语言中容易得多。

另一方面，放弃你的陈述式完成（statement completion）和你的编译时错误检查等须要付出很大的代价。

我已经看见过很多人在争论动态语言时说的是 Smalltalk 的浏览器。

Anders：我不确认我会买这个帐。当你的问题足够小时，它是对的，在 Smalltalk 刚出现时它一向只是一个足够小的问题。由于现在框架的规模的增大，即使人们想去了解，也难以想象他们会真正了解特定对象中的所有的 API 。由编译时元数据和静态类型所驱动的陈述式完成、智能提示（IntelliSense）或重构等工具是颇有价值的。它们的价值会越来越大，因为世界将要变得更加复杂。现在，我们看到大量的动态编程语言层出不穷，不过我认为它的兴盛大多是因为 1）是以元编程的角度看待的；2）在很多方面，这只不过是对于复杂的 J2EE 环境的一种反应。

我的确看到很多 Java 程序员转而使用 Ruby，而仅仅是因为他们深受框架、Struts、Spring 和 Hibernate 等的困惑。除非你是一个技术奇才，否则你将不可能凭借一己之力将这些东西融合到一起。

对于那些不是也不想成为“魔法大师”的人来说，编程会更容易吗？

Anders：我想是这样的。这完全取决于你通过编程所要表达的意思。因为在某种意义上来说，你是在使用一个电子表格编程吗？如果你可以让人们在编程时甚至没有意识到他们正在编程，那么，哦，我的上帝，那就太棒了。对于教会全球的用户能像程序员那样在我们今天使用的编程环境中编写程序，我对此不抱任何希望。毫无疑问，是编程，没错，不过应该在一个更高的级别当中。

我们现在和在未来五年里将会面临什么问题呢？

Anders：并发是目前很大的一个问题。问题已经是迫在眉睫，而我们也已经开始去寻找解决方案。，在可预见的将来，我的最大挑战之一就是让我们的团队着手解决这个问题。

再者，我们愿意以一种发展的方式做这件事情，但是，你如何在没有打破现有代码限制的情况下来处理共享状态问题和副作用呢？我们也不知道答案，不过并发性是所有新语言和新框架都要具有的一个足够大的模式变化。尽管我认为我们还没有达到这个层次。

我认为，让人们有可能编写本质上并行的 API，而且是由真正理解某个特定领域的人来编写，无论它是进行转换、数值处理、信号处理、位图或者图像操作，我们在这些方面都可以大有作为。不过，把 API 加入到它上面，从外部看来很像是同步方式，在某种意义上来说，它与内部的 API 的并行性就隔离开来了。

为了使我们能够正确地做到这一点，目前的编程语言中还需要一些东西。我们已经做到的就是将代码作为参数来传递。由于 API 的功能变得越来越复杂，你不能只是给 API 传递简单的值或者数据结构。你应该能够将代码段传递给 API，然后进行编排和实现。

你需要高级函数和抽象，比如 map、fold 和 reduce。

Anders：高级函数。确实如此。为了能够做到这一点，你需要用到一些东西，比如说 lambdas 和闭包等。对你来说，为了能够在并行环境中做到这一点，你还需要保证这些 lambdas 是否纯粹，或者它们是否具有副作用。我能自动地并行运行它吗？或者，它们会产生副作用吗？我怎么能知道这些呢？现在，我们的语言当中还没有这些东西，不过，我们肯定会考虑把它们加进来。当然，添加它们的诀窍在于，它对你不会有太多限制，而且也不会过多地破坏你的现有代码。这是一个巨大的挑战。

我们团队每天都在思考这些问题。

314

有必要为并发性而改变语言实现或语言的设计吗？

Anders：噢，它确实改变了语言的设计。很多人抱有这样的希望：一个人只能够拥有一个编译器的并行交换，而且你只是想说，“并行地编译它”，然后它会运行得更快，并且是自动并行化。从来不会发生这种情况。人们已经尝试过，这对于我们在主流语言中（比如 C++、C#和 Java 等）使用的命令式编程风格是不会起作用的。这些语言很难自动并行化，因为人们在程序中过于依赖它的副作用。

你要做一些事情。首先，要为并行性构建现代 API，它们的层次要比线程、锁和监视器以及我们现处的层次更高。

接着，你要从语言中得到一些东西，从而使编程更容易而且更安全，比如保证对象的不变性、你所知道的没有副作用的纯函数、分析孤立的对象图，这样你会知道一幅对象图的特定引用是否已经被其他人使用过。如果它没有被使用过，那么你可以安全地 mutate 它，如果它已经被使用过，就可能会有副作用。诸如此类；在这种情况下，编译器可以做出一些分析并有助于保证安全性，就像我们今天拥有的类型安全和内存安全等一样。

为了能够更好在这些并行系统上编程，我认为在未来的 5 到 10 年里将需要这些情况。

本质上你是在告诉计算机去做什么。

Anders：这其中的一个大问题是，我们现在使用的命令式编程风格实际上是描述过于详细（overspecified）。而且这就是为什么难以实现自动化并行的原因。

未来，我们有可能让框架来完成处理并发的工作吗？

Anders：噢，我想可能。并发有很多不同类型，不过如果你是说数据并行方式的并发，它要在大型数据集上进行运算（比如图像处理、语音识别或数据处理之类的运算）时，我认为拥有一种可视为 API 的模型是非常可能或是非常适合我们的。如果你能对 API 说：“这是我想要使用的数据和操作。你去完成它，并在给定可用 CPU 数量的情况下尽可能快地做这些事情”时，你已经拥有了高阶 API。

这非常有趣，因为现在你可以相当轻松地说“数据在这儿”。对于一些大数组或者一些对象等，你可以只给出引用。指定运算通常须要给出代码段的引用，如果你使用委托（delegate）或者 lambda 演算，而且如果编译器能够分析并保证这些 lambda 演算没有副作用，而且如果有副作用就会警告你，这无疑是很好的。那是我所说到的一部分，不过，那只是并发的一种类型。在异步性更强的分布式系统中，还有其他类型的并发，
315 在编程语言中支持这种不同的并发，我们也会从中获益。如果你看看 Erlang 这样的语言，它应用于高度可扩展的分布式系统中。它们有一个非常非常不同的编程模型，它更具函数式特征，而且是基于异步智能代理和消息传递等机制。我认为我们的语言也可以把那些令人感兴趣的东西吸取进来。

面向对象范式会带来什么问题吗？

Anders：这取决于你如何对面向对象范式归类。多态性、封装和继承本身不再是什么问题，尽管函数式语言在如何使用它们的代数数据类型来处理多态性方面，典型地具有一种不同的观点。除此之外，我认为，面向对象编程的最大问题是：人们要以一种极具“命令式”风格的方式来进行面向对象编程，其中，对象将可变状态封装起来，而且你调用某些对象的方法或者向那些对象发送消息：引起它们修改自己，而引用这些对象的其他人则不知道。现在，你最终只能对你无法分析这个副作用感到惊讶。

从那个意义上来说，面向对象编程是一个问题，不过你可以使用不可变对象（亦称常量对象）进行面向对象编程。然后，你就不会再遇到像那种函数式编程语言才有的问题。

就您对函数式编程的兴趣而言，学计算机科学的学生应该为了函数式编程而学习更多的数学和做更多的实验吗？

Anders：噢，毫无疑问，我认为在任何计算机科学课程中加入函数式编程内容是非常重要的。你是否要从它开始，那要视情况而定。我无法确定你最开始的编程导论是否应该是函数式编程，不过我毫不置疑地认为它应该是课程的一部分。

人们应该从你的经历中学到什么经验和教训呢？

Anders：噢，如果你看一看我最初的作品 Turbo Pascal，它就反映出我不喜欢采用传统方式来做事。别害怕。那只是因为人们说你无法做到，并不是你真的无法做到。只是他们无法做到而已。我认为，利用创造性思维来考虑问题，并设法为现有的问题找到新的解决方案，这总是非常令人开心的。

我认为简单性一直是一个赢家。如果你能找到一个更简单的解决方案——毫无疑问，这是我一贯的指导原则。我一直设法使它更简单。

另外，我还认为要想真正在某些方面有些特长，你必须对其充满热情。那是你学不来的东西。而你已经有了那种东西。我对编程感兴趣不是因为我想赚很多钱，或者是因为有人告诉我应该这样做。我对它感兴趣完全是因为它深深地吸引了我。你无法让我止步。我必须编写程序。它是我想要做的唯一一件事。我对它非常非常地狂热。如果你想要真正地擅长某些事情，就必须对此充满热情，因为它需要你投入大量的时间，而大量时间的投入正是真正的关键所在。你需要完成很多的工作。

第 14 章 317

UML

你如何就软件设计与其他人交流思想？建筑领域使用的是蓝图。UML（Unified Modeling Language，统一建模语言）是一种设计用于表示软件项目产品的图形化语言。结果，将 James Rumbaugh 的面向对象分析、Grady Booch 的面向对象设计，以及 Ivar Jacobson 的面向对象软件工程组合在一起，允许开发者和分析师通过特殊类型的图表来为它们的软件建立模型。虽然语言有很多一以贯之的标准，你很可能已经在快速白板绘图时使用了它的一些概念。

318

14.1 学习和教学

Learning and Teaching

我听说您开始在爱立信公司工作时，几乎对编程一无所知。您是怎么学习的？

Ivar Jacobson：我开始在爱立信公司工作时，对电信一无所知。这是一种颇有价值的经历。即使我在一个开发硬件交换机的分部工作，我也能提炼出构建大系统的整体观点。我在那儿工作了将近四年，而且我学会了如何从整体上来考虑系统。那种知识是独一无二的，因为开发软件的人们在构建大系统方面毫无经验。

我是一名电子工程师——很可能是唯一一位拥有工程学位的工程师。那里大多数人没有学位。我在大学里学会了如何去攻克问题，而且我也从基本上能够解决每一个实践问题中获得了很多自信。

你过去常常把汇编代码带回家，在晚上研究它，并为开发者准备了一些问题。

Ivar：我们几乎没有文档，它是由对软件了解不多的人们完成的。他们编写了需求，并以某种流程图的方式将这些需求记录下来，不过这些需求经常是矛盾的和不完全的。

我们也有流程图，我们的人开发了它并在使用，不过它们不是组件化的，因此图表变得非常庞大。我们给每一行汇编代码添加了描述，不过人们主要是通过在一起工作、交谈及读取代码来学习它。我会就晚上读过的代码问人们，基本上都是："你在这里究竟想要干什么呢？"等。

我在理解代码之前，很可能必须要就同一段代码读上三五遍，不过我是非常固执的，因此我会在晚上对很多代码反复研究。白天我通常是忙于管理项目。我的角色是一名项目经理，并不真正地参与技术问题，而是要理解项目，而且至少能走过去问一下人们干到哪里了。我更像是一名项目管理员。我讨厌那个角色，而且我学得越多参与得就越多。我只花三个月的时间就能得出结论：我们在做的事情能否变成一个产品。

当时，我们的项目组有 75 人，同时我们要完成的任务对爱立信公司来说绝对是非常关键的。你可以想象一下项目经理走过去跟他的老板说，"这会/不会变成一个产品"。

319

您是如何提出用例概念的？

Ivar：非常自然。在电信行业，人们经常会用话务案例（traffic cases）这个词。话务案例跟用例差不多，不过只用于电话呼叫。我们在交换机的其他特性中并没有任何用例或话务案例，即使那些特性实际上占据了超过80%的代码，比如运算和维护等。对于这个软件我们仅仅是讨论"特性"。整个特性清单很长，而且很难搞清楚它们之间的相互关系。

话务案例和特性这是两个不同的概念。它们已经在电信行业使用了至少 50 年，不过它们不易于组合。我认为，难以找到一种统一的概念，能够描述系统中各种各样的相互作用。我从外部来看这个系统，就好像它是一个黑盒一样。我设法识别出看起来对用户有用的所有情景。这个概念在瑞典语中叫做使用案例（usage case），不过我没翻译好，因此它就变成了用例（use case）——而且我对此非常满意。

直到 1986 年 4 月，我的用例需要依靠它们自己，因此我把它们做成类似于"类"的东西。只要在用户和系统之间出现事务，用例可能就会被看成是对象。它们也可能与其他用户互相作用，就像电话呼叫一样。我有一个重要目标，就是能够重用用例，因此我需要一个比面向对象中的抽象类更加抽象的用例系统。使用对象和类进行类比有助于我找到一个统一概念（用例），能够同时用于描述话务案例和特性。

在这个概念之前花了一些时间，不过到了 1992 年我已经对用例的重要因素了然于心了。在我编写了《面向对象软件工程（Object-Oriented Software Engineering）》[Addison-Wesley]之后，我不知道还有谁从此以后在本质上做出过真正的贡献。其他人用更好的方式解释了它们，比如说 Kurt Bittner 和 Ian Spence 编写的《用例建模（Use Case Modeling）》[Addison-Wesley Professional]一书。这两个人现在都在我的公司工作。对于介绍和详细讲解相关概念来说，他们的书写得相当不错。

面向方面是一种新发明，当然，用例也是真正的好“方面”。这就导致 2005 年出版了《使用用例的面向方面软件开发（Aspect-Oriented Software Development with Use Cases）》（和 Pan Wei Ng 合著，他也在我的新公司工作）[Addison-Wesley Professional]一书。

当您给爱立信公司的开发者提出这种概念时，他们有什么反应？

Ivar：最初的反应来自爱立信公司我的朋友和资深人士，当时他们联想到方法学说“这里确实没有什么新东西”。我能够告诉他们的只是他们并没有搞懂它。我马上明白了用例也是测试用例（test case），因此如果你能预先指定你的用例，你就已经有了很多测试用例。在 1986 年，那就是真正的新东西。我们能够使用用例驱动的开发，意味着每一个用例都描述了某些情景集，在这些情景集你描述它们是如何通过类或组件之间的合作来实现的。

我们怎么样来分享您在软件领域的经验呢？

Ivar：这就是我花了过去五年时间来解决的一个非常特殊的问题。你需要理解你所拥有的知识，因此你可以描述它。对其他人来说，学习专门的观点是很难的。即使你已经拥有了该领域最好的流程，你仍然需要将那些知识转移给其他人。你需要系统化的观点，然后通过转移知识系统来转移你的知识。做这些，好与不好的方式都有。

15 年前，我们就有了 Objectory（译注1）。它最后发展成了统一软件开发过程，当然其中还增加了很多新知识，不过获取知识的整体技术仍然只是差强人意。这在当时就是我们能够做到的最好。它是独一无二的，因为在此之前它从未做到这样的程度。

现在我们推动“基于实践的”知识转移。与转移你在软件开发时需要知道的所有知识不同，你当时只是转移一个实践，而且是在其他人最需要时。那些实践很小而且很友好，而且在实践之内的一切都是很恰当地逻辑组合在一起的，因此它易于学习，而且在一个过程中你已经拥有了需要一直记住的所有东西。在过去，你可以说一个过程就是一组观点的集成。现在我们是用一个实践包构成的过程将其取而代之。

我们应该如何看待计算机科学教育呢？

Ivar：我们的问题在于大多数大学教授实际上对工程知之甚少。具体到软件领域，他们中很少有人曾经自己开发过任何有用的软件。或许他们曾经开发过一些编译器，不过这些编译器大多都是非常学术性的。或许他们曾经开发过一些培训软件。我们确实无法期望他们能够教授软件工程。

译注1：Objectory 是一个全面的、商品化的面向对象开发过程，它由 Objectory 公司的创始人 Ivar Jacobson 博士创建。统一软件开发过程（Rational Unified Process，RUP），又称为统一软件过程，它是 Rational 公司（现归属 IBM 公司）推出的一种软件过程产品。从软件过程模式角度看，RUP 又是一种典型的软件过程模式，它以迭代增量式、架构为中心、用例驱动的软件开发方法为主要特征，其中用例驱动乃是贯穿软件开发始终的方法。

如果你及时回顾一下的话，你可以在大学层面学习他们可能已经教授的 Java 编程或任何其他语言，不过如果你想要真正地理解软件，你就需要具备其他很多不同领域的能力，比如说需求、架构、测试、单元测试、集成测试、系统测试、性能测试，更不用说配置管理、版本控制、构建框架和构建应用程序之间的区别、构建可重用软件、面向服务的架构，以及产品线架构等。

在大学你学不到真正困难的知识。

学生需要更多的实践经验，比如说参与一个开放源代码项目，或者是在一家大公司实习？

Ivar：大学里的训练主要是通过教育和构建简单的东西来完成的。

当然，我对此并没有一个公正的整体印象，而是与其他工程学科的情况做的比较——例如，建筑业。建筑业有建筑学的教育，它与建造什么建筑物是分开的。若干不同的学科融合在一起。如果你要把人们培养成建筑师，他们仍然得去建造一些建筑物，否则他们不是有用的建筑师。你可以一直充满梦想，不过如果你无法实现你的梦想，它们就没什么用处了。

实际上，我们并没有教育人们“工程”。构建软件要比艺术更加“工程”。很多人愿意看到它的不同，不过很少有专业程序员会把时间花在艺术上。他们大多数都是工程师。这并不意味着他们没有创造力。人们应该相信受过机械专业教育的人会没有创造力吗？比如说建造不同类型的机器？如果我建造轮船，我没有创造力？或者是建造房屋？当然我有创造力。建筑师也是非常有创造力的。

这样一来，我们需要认识到：软件开发是一个工程而不是一种艺术。你需要在工程方面来培养工程师。不过，美国和欧洲的很多大学都有一个悠久的传统：让他们的教授只是搞纯学术研究。按照我的理解，根本问题在于我们确实没有一种软件工程理论。对于大多数人来说，软件工程仅仅是各种专门观点的集大成者。这是要解决的最重要问题之一。

请分析一下为什么你认为这个问题是如此重要。

Ivar：在应该如何开发软件方面，我们的观点看起来好像是每两三年都要大变一次，比时尚的更新更为频繁。全世界的大公司会草率地放弃耗资巨大的流程和工具投资，几乎连试都不试。他们不是从经验中学习，而是随意地开始他们认为的根本性的新东西。实际上那只有很小的一点变化。在时尚界，差不多都是小题大做。在服装这种不起眼的小事上尚可接受，不过对于我们的软件投资规模来说，这简直就是既浪费、又昂贵，而且还非常荒唐。

最新的趋势是“敏捷化”（以 Scrum 为例）。“敏捷”运动让我们想起人们在开发软件时只关注最重要的东西。这并不是什么新东西——这个主题每十年左右就会卷土重来一次，而朴素的管理者试图将在创造性解决问题方面的基本职责“机械化（mechanize）”和“商品化（commoditize）”。重要的是，我们没有忘记如何在团队中工作，如何去合作，如何把我们的工作用文档记录下来，而且如何按照每天、每周及每月的时间尺度来计划我们的工作。不过，如果让这些事情重回焦点，很多都使用了新名词而难觅其踪，从而造成了全新的假象。

这么做的结果就是浪费了很多精力：过去的事实又被重新发现出来，不过在外观上是披了一件新衣服。在一直渴求“新闻”的媒体的大肆宣传下，更年轻和经验不足的合作者会推动新趋势，追捧新权威。与实际开发脱节的管理者发现他们自己处于一种绝望的境地：拒绝最新的方式，而且他们给自己打上了落伍的烙印。为了证明新方法的优点，又强制运行了一些试验项目，不过有目的的开发者可以在很小的规模上来做试验。结果是，新方法战胜了旧方法，而且，使用旧方法完成的工作连同那些不是采用旧方法完成的工作都一起扔掉了。等他们发现新方法自身也有一些地方不能与采用新方法的那部分一起工作时，已经为时太晚了。

这个问题的根源在于对软件开发本质的深层次误解。研究者已经尝试使用新理论来攻克这个问题，比如，利用形式体系来证明程序的正确性，或者是使用在学术界之外从未使用过的形式语言。工业界已经花了很多年的功夫来标准化极难理解的膨胀的元模型。

大学和工业学院教给我们一种特定的工作方式。每一个项目都是采用一种特定的方法，在我们可以开始工作之前，我们必须首先学习和掌握这种方法。我们每换一次工作，都必须在开始做手头的实际工作之前学习一种新方法。这样效率很低；我们无法从经验中学习，因为我们永远都是重新开始。

我们不要继续永远在追随时尚和不费力的解决方案了，它们总是令我们失望。但我们要怎么做呢？这个问题我至少已经考虑 10 年了，而且，对于如何才能做到，现在我有一个具体的想法。

您的解决方案是什么？

Ivar：我们需要一个关于“软件开发实际上是什么的”基本理论。依我来看，这个理论就在我们面前。我们只是要抓住它。以所有这些方法、过程及实践开始，找到软件开发的“真相”。例如，我们能够做我们在我的公司已经做的事，以及现在已被全世界数百家公司使用的那些东西。

首先，我们需要发现构建软件时一直拥有或一直在做的事情的核心。例如，我们一直在编写代码，我们一直在测试它（尽管有时并没有用文档记录好我们是如何测试它的），我们一直在考虑需求（有文档或没有文档），我们一直有一个积压工作清单（显式的或隐式的），而且我们一直有一个书面计划或是脑子里的计划。你可能会借用一个用滥了的比喻，说我们必须找到软件开发的 DNA。

我和同事们一起，通过研究 50 来种方法，找到了 20 个以上的这种元素（包括 XP 和 Scrum）。表面上，我们使用的这些方法和方式可能会出现很大的区别。作为一个例子，你可以使用特性或用例来获取需求。不过，这两种方法有一个公共的基础，即这是在内核元素中获取到的。

然后，我们利用这个内核元素来描述广泛使用的成熟方法和实践：架构、Scrum、组件、迭代等。现在已经开发了大约 15 种这样的实践。由于与任何具体的实践相比，内核是不可知的，因此我们可以简单地找出不同实践之间的实际区别，它不是仅仅是表面上的，而且是深层次的。这就减少了盲目信仰的因素，其中每一种方法都是嵌入进去的。教育会变得更有逻辑性，因为它更关注于单独的观点，而不是构成每一种方法、过程或方法学的特定的观点集成。我相信学生会喜欢它。

我们的工业学院或大学如果能培养学生软件工程的基础知识，并利用在此基础之上的一组优秀实践来训练学生，那将是很棒的。那里也有很多相关的研究空间。

记得 Kurt Lewin 曾经说过：“没有什么能比优秀理论的实践性更强了”。优秀的理论使它易于学习和开发你的知识而不必过度严格。

您游历颇丰。您注意到在世界的不同地方编程或设计的方式有什么不同吗？

Ivar：当然会有不同，不过现在美国在干什么，世界的其他地方也会干什么。或许美国会比其他国家在尝试新东西方面稍微快一点，不过他们也会抛出自己的东西。很多美国公司在新东西出现后更易于放弃他们必须运行的东西，而在欧洲，人们则会三思而后行。

在东亚，他们在新技术方面落后几年，不过另一方面，他们可能不必去犯同样的错误。

我已经看到中国有一个清晰无比的趋势。他们想要追随印度，因此采用 CMMI（译注2）变得非常流行，大约在五年前达到巅峰。现在，他们已经看到了 CMMI 只能处理问题的过程改进部分。不过，在你改进一个过程之前，须要让它名符其实，因此，他们现在发现需要优秀的实践来帮助自己，以较低的成本快速地开发出优秀的软件。

文化会对我们设计软件的方式产生多大的影响？

Ivar：我不知道。我通常看到，芬兰人在某些方面会比其他斯堪的纳维亚人（译注3）更具牛仔情怀。他们更为实际，并因此成功。芬兰语里有一个特殊的单词“sisu”，意思是“永不放弃”，他们非常尊崇这个理念，因此他们并不做不必要的事情。很多人很可能会说芬兰人天性近乎敏捷（Agile），而这总是一件好事。

其余的斯堪的纳维亚人在软件开发方面也非常优秀。你可以拿瑞典爱立信公司作为例子，不过我认为不应该夸大这个话题，因为我并没有足够的证据来详细描述。

14.2 人们的角色

The Role of the People

我们怎么能知道某人是否适合担任一个软件项目的架构师呢？

Ivar：这一点我非常清楚。我认为架构是非常重要的，不过，出于多种原因，我对称某人为架构师非常谨慎。我已经见过很多拥有一个架构师团队的公司，他们被派到其他组织去干项目。如果他们在一个特定的项目内工作，那就很好，不过，很多公司（比如大银行）通常有一群企业架构师坐在那里设计架构方案。

然后，他们会把这些整个推给开发者。开发者只好对自己说：“这是什么？什么用也没有”。在很多公司，企业架构师坐在象牙塔里，什么有用的事也没干。

我从不认为应该把架构师当作特殊的一类人，因为软件是依靠团队开发的，而不是依靠烟囱式的组织来开发的。

很多公司试图将软件开发组织为很多部门、分部或小组。一个小组进行需求分析，一个小组进行架构设计，一组编码，一组测试，等等。然后，他们把各种项目都抛给这个组织，因此，你必须有一个项目经理来同这些不同的小组打交道。需求分析的责任在需求小组的领导者手中。测试则由测试小组领导来负责。这些不是团队，只是小组而已，因此，你并不会真正地了解项目。项目经理仅仅是一个管理员，而不是一个可以给出发展方向的经理。结果肯定是以极慢的速度、极高的代价开发出了糟糕的软件，因为需求小组编写的需求，其他人很难理解。

相反，我们要同包含胜任管理需要或胜任软件设计等的人才之团队合作。团队由一名经理或教练领导，而且团队也是自组织型的。它就像一个足球队一样：你有前锋、后卫，还有守门员，不过他们也要根据需要转换角色。有时是前锋防守，或者是后卫进球。这就是我们在软件中需要的模型。

我们需要有一个团结奋战的团队，团队成员互相帮助，而且编写需求的人能体会到开发人员的难处。做需求分析的人也能够随后确认需求是可测试的，而不会为应付差事而编写需求文档。

译注2：CMMI（Capability Maturity Model Integration，能力成熟度模型集成）本质是指软件管理工程的一个部分，软件过程改善是当前软件管理工程的核心问题。

译注3：斯堪的纳维亚（Scandinavia）是指北欧一地区，包括挪威、瑞典、丹麦、芬兰、冰岛以及法罗群岛等。

我们有一个新模型：团队模型，而不是组织模型。

您如何定义“社会工程”这个术语？

Ivar：社会工程是指让人们在一起工作。它牵涉到组织一个团队。它牵涉到按天、按周、按月等来组织你的时间，它并未牵涉到技术；它牵涉到如何推动和激励你的队员奋发工作，以及如何取得成果。

我们在这方面一直有很多管理的书籍，不过在软件方面，它是一个新领域。当 CMMI 和 RUP 之类的方法学让组织两手空空时，敏捷运动就应运而生了，它主要用来解决刚才的问题。

我从未认为，人们在尝试做什么东西时会按照 RUP 的说法去做，因为你不得不更多地把 RUP 当作一个知识库、思想库，然后根据什么对人们有意义再去做。我一直都在这么说。不幸的是，RUP 被看成是一种约定俗成的方法学，就像烹调食品一样。我们这些曾经开发过软件的人，甚至没人会梦想能真正按照检查清单一步一步地做下去。

为什么我们改进编程方法和过程的过程如此缓慢呢？

Ivar：这是实际问题。从我的观点来看，这个行业是很不成熟的。虽然它比 20 年前更成熟一点，但我们今天要构建的系统更为复杂。20 年前，我们是从一种编程语言和一个操作系统开始。现在，我们已经拥有了各种各样的框架。

软件行业是我所知道的最具流行意识的行业。人们每两到三年就需要有一个新话题；否则他们不会看到任何进步。我们接受新观点的方式不仅仅是剔出我们需要取代的不好的或旧的观点，而且基本上是全盘否定并全部重新开始。我们并没有因系统地改变已拥有的东西及添加新东西而前进，因此，我们在某种程度上仍然是原地踏步。我们并没有真正感觉到进步。

现在流行的新方法学与 20 年前或者 30 年前有很大的不同，不过它们有一种新的表达方式和新的谈论方法。我们也看到了已经非常成功的大过程的逆作用，比如 CMMI 和 RUP。逆作用的意思是说，属于这些或其他类似阵营的一切都很糟糕，而且现在我们需要一些有活力的新东西——但它实际上不是有活力的新东西。这些新方法学实际上并不新，它们只是我们已经拥有的那些方法的变体而已。

实际上，敏捷的确包含了一些新东西：更加突出地强调了人和社会工程。即使过去成功开发过软件的大多数人对此都非常熟悉。在软件开发中，人是最重要的资产。拥有有能力、有动力的人是低成本快速开发优秀软件最为重要的前提条件。有时我们会忘记这一点。

我认为我们的另一个问题是大学毕业生是用最新的银弹培养出来的，实际上，他们并不知道如何去处理他们从过去的实践中继承而来的商业软件。他们年轻、充满活力而且精力充沛，我们无法让他们从他们认为的过时的老一套开始。他们只能无法完成工作，特别是在这样的好时代当中。这些年轻、经验不足而又受过良好教育的人开始在组织中占据统治优势，结果就是我们无法前进。

我们应该如何处理遗留软件的问题？

Ivar：传统上，软件都是由那些从未有过任何显式的方法学的人们开发出来的。他们无法描述他们在干什么。他们也不写文档。你是后来加入的，你就很难理解系统结构，因此，你并未理解系统之后的架构或者想法。让新人接管这类系统肯定是非常非常困难。

如果一个企业的所有员工同时离开，这个企业必死无疑。即使你已经花钱招聘他人，他们也不知道要做什么。软件领域也毫不例外。这是商业的本性。如果你已经有一个可以理解的系统，而且你有办法培训人们熟悉系统，那就很有用，不过这里不会出现奇迹！

326 我们需要具有良好架构和良好模型的可理解软件。我们知道，如果代码没有一个可见的架构，几乎无法管理。

对于大多数大公司来说，一个主要的挑战是去修改遗留系统及开发和（或）扩展的方式。那里有很多与这些系统相连的固有实践，它们随着时间的推移而发展，而且那些实践很多不是 Agile 的，或者是兼容 Agile 的。为新系统或新产品而改变开发方法远非一个挑战。使用的方式应该为遗留系统而优化。我的观点可以用这句话来表达：产品开发是一个改变管理的过程，是从有到多的改变。新的开发仅仅是一个特殊情况，是从无到有的改变。这种观点应该渗透到你做的每件事和开发软件时部署的实践当中。

基本上，有两种方式可以用来管理遗留系统和改进它们。

第一种只是部署并不真正地改变产品但会改进你工作方式的实践，比如迭代式开发、连续集成、测试驱动的开发、用例驱动的开发、用户故事、成对编程及横切（cross-cutting）团队等。引入这种实践的成本和风险微乎其微，不过对于大公司来说仍然非常重要。

第二种方式更重要：通过实践改变实际产品，比如说架构（在简单级别上）、企业架构、产品线架构、组件等。你需要花很大代价进行设计。成本和风险更大，但投资回报也极高。

使用正确的方法，能否避免没有可见架构的系统管理问题？

Ivar：不，不可避免，不过可以减少问题。为你的软件编写文档可能不会很管用，因为人们并不去读文档。即便因此，集中关注本质问题的健全文档也是有用的，因为它使得系统更加友好。例如，能够描述你的架构意味着你实际上有一个架构!

你仍然无法期望人们可以离开后，其他人能接管好。你需要有一个过渡，让新人熟悉他们的工作环境。无论你教给他们多少知识，如果没有可见的架构，要想转移该系统的知识也没有那么容易。

327 **转移知识的最佳方式是什么呢？**

Ivar：一般来说，人们使用软件不需要阅读书籍或手册。如果要说在哪儿会读的话，那就只有在大学里人们才会真正地阅读。如果说人们在工作时使用书籍和手册，那仅仅是一个神话。

我曾经写过几本书，而且我非常高兴有人去买这些书，不过说到其他的书，他们是不会去读的。人们既不会读过程书籍也不会读语言书籍，这是一个自然规律。

不是要学习“重要的”方法学或者像 UML 或者 Java 这样的“重要”语言，要关注实践。实践更具可管理性。你可以成为一名实践专家，而不必成为一名完全方法学专家。我的那些编写方法学图书的同事们，大多数在超出方法学——实践方面不是真正的专家。

不是要研究“重要的”方法学或一种语言，而应关注每一次的工作实践。没有哪一个人能够掌握所有优秀的、有用的实践，不过你有可能把这些实践组合到你的工作方式当中。在过去的五年中，我一直致力使实践更加简单，同时还要保持它们相互独立，不过是采用这样一种方式：其他人可以用它们来组成更大的过程（或者工作方式）。

我还听说过使用卡片的故事。

Ivar：每一种方法学都是从一些有趣的新观点开始的，从别的地方借用了一些有趣的观点，把所有这些观点都集成在一起，并把这种集成叫做一种方法学、过程、方式，或者其他什么东西。

能够做到这些，并且是相容、完备和正确的，这是相当了不起的。已经有人成功地做到了这些。一些人已经成为了公认的权威。

不过，这些只是问题简单的一面。真正的困难在于其他人愿意采用。另一个问题是在出现新观点时，你能够改变已经拥有的东西。

因此总的来说，在大量“生产”方法上，我们并没有成功。

我的公司已经有人（特别是 Brian Kerr 和 Ian Spence）提出一些重要的创新方法来解决这个问题。这些创新之一就是使用卡片来描述你在做的东西，或者你在开发软件时产生的一些东西的本质。

使用卡片是描述实践的一种敏捷方式。它们的本质相同，剩下的你可以照葫芦画瓢自己弄了。

14.3 UML

您如何定义 UML 呢？

Ivar：UML 是一种蓝图语言，用于软件的说明、架构、设计、测试及使用。

它与不同的软件工程方法如何互相影响呢？

Ivar：OMG 在 20 世纪 90 年代早期确认的所有软件工程方法学（我可以记得是 26 种方法学）都有自己的符号，不过现在它们大多数都已经采用了 UML。

您是将三位设计者的设计优点集合起来，还是把您的观点强加给他们？

Ivar：我们的讨论非常热烈，不过这些讨论有助于我们设计一种更好的语言，它要胜过我们任何单个人的一己之力。如果没有 David Harel、Jim Odell、Cris Kobryn、Martin Griss、Gunnar Overgaard、Steve Cook、Bran Selic 与 Guus Ramacker 等人的贡献，我们就不可能做好。

您在未来会改变什么？ UML 可能会有什么变化？

Ivar：最重要的是：

- 语言太复杂了。我们需要改变它们。所有的应用程序 80%都可能使用小于 20%的 UML 来设计。在我的公司里，我们定义了一个 UML 纯子集，它变成了精炼版统一建模语言（Essential Unified Modeling Language）。我们也使用一种很不同的方式来描述 UML，它对普通用户更有吸引力。传统的 UML 是为方法学家或工具厂商而设计的。
- 我会喜欢将 UML 重新构造为一组领域特定的语言（DSL）。我愿意用类似于我公司重新设计的统一过程（Unified Process，UP）的方式来做。ADSL 是建模语言（UML 是它的一个例子）的一个方面。你以“根据很多横切关注构成一个软件系统”的方式，将建模语言创建为很多那种 DSL（方面）的组合。我声明这种语言不是为用户而是为方法学家和工具厂商设计的，我声明即便是对于后者它也不怎么好。UML 的语义定义得很差。UML（特别是 UML 2.0）包括了很多来自不同方法学阵营的结构，以至于它不可能去清晰无误地定义它的语义。与很多其他语言一样，正如 John Backus 形容 Ada 那样，UML 变得“臃肿不堪”。

  焦点聚集在具体的句法（图标上），在某种程度上还聚焦于静态语义之上，不过我们没有定义操作语义

（operational semantics）。我认为我们应该会招致这种批评，因为在当时标准语言设计实践中使用的是标志语义这样的技术。我们没有这么干。我们只是一页一页地编写我们知道的非常难以理解的东西。我们能使用定义 SDL 所用的相同的实践（早在 1984 年就使用 VDM 定义了电信建模标准）。

329 SDL 成为了一种具有明确定义语义的建模语言。即使 UML 采用了 SDL 的主要部分，我们也不会采用 15 年以前使用的语言设计实践。悲哀!

也就是说，即使 UML 不是形式化定义的，它的设计也要远远好于大多数其他受欢迎的面向对象建模语言。在 UML 变得可用时，基本上所有的竞争语言都被抛弃了。如果使用得当，UML 可以真正地帮助开发者实现成功。UML 的朋友不用担心，这是一个美好的未来，不过 UML 应该提供一个更好的结构，而且它也需要形式定义。

您如何算出哪一个元素可能会从 UML 中删除？您会用什么过程来简化语言？

Ivar：我会从语言的基础开始。我不会从整个语言开始并删除个别片断。我知道哪些语言结构是真正有用的，以及哪些不是。有一些语言结构我甚至不愿意多看一眼。我并不想具体点名，不过我们已经确定了 20%，至少粗略地说是这么多。

我们教授 UML 时，通常会教授 Essential UML，这是基于我们的经验。我们使用与描述过程或实践元素同样的观点来描述语言元素：我们使用卡片，而且每一张卡片都代表一种语言结构，比如说一个组件、一个接口等。我们是在说教育学。我们不是在谈论什么新东西，或者任何新的语言结构。我们只是知道人们的确不会阅读和并不喜欢厚厚的语言规范，因此我们需要找到一种更加友好的方式学习。通过对象、接口、类、组件等来学习对象。

您如何将“UML 重新构造为一组领域特定的语言”？

Ivar：我们的 UML 有一个基本的通用适用核心。我会确定该核心的各个方面，而且通过不断地添加方面来描述 UML。UML 的这些方面就是我们所说的实践，而且 UML 这些类实践的事情为领域特定的语言。

望文生义，一种领域特定的语言会支持一个特定的领域，比如说一个垂直行业（企业级系统、电信系统、卫生保健系统等）或者一个学科（需求、设计、实时、测试等）。小而言之，UML 会构成一种领域特定的语言。按照这种方式，你会从不同的领域用特定的语言组成 UML。这些领域特定的语言需要有一个公共核心和公共语义，否则将会出现一个非常困难的问题，那就是无法在不同的领域之间转换。

有没有实践可以用于设计 SDL，能够改进 UML？

Ivar：在 15 到 20 年前我们设计 SDL 时，使用维也纳开发方法（Vienna Development Method，VDM），它是在 20 世纪 60 年代或 20 世纪 70 年代由 IBM 的员工开发的。它是可以用数学方式描述概念的一种语言，比如说一种语言、一个操作系统或任何其他系统等。它依靠离散数学：集合论、映射等。实际上，以这种方式，你就可以利用数学方式来定义每一种语言结构的含义。

330 我们首先确定了抽象句法，并且是使用离散数学描述了抽象句法。然后，我们使用它来定义元素域。我们通过描述在该域中何种条件元素为真和假，定义了静态语义。接下来，通过描述特定语句的含义，描述了操作语义。这是一种描述语言的数学方式。最后，我们将图形化符号映射成抽象句法。

我非常深入地参与了 SDL 的相关事项，不过我无法说服 UML 同事在这上面为 UML 做任何事情。他们觉得它只是学术。根据我在 SDL 上的工作经验，我对此并不同意，因为只要你想要构建工具，你就要知道严格的

语义。否则人们只能去猜了。

当来自 IBM 的 Steve Cook 和来自 Objectime（后来被 Rational 并购）的 Bran Selic 加入团队时，他们说，“这是非专业的。在没有更正式地定义语言之前，我们是不会加盟的”，因此，我为了折衷提出了一种变体。我说，“让我们用数学方法定义抽象句法和静态语义，不过我们要用英文来描述操作语义”。UML 2.0 比 UML 1.0 要好，不过如果你确实想要理解每一个细节的话，它仍然不够。

您怎么看使用 UML 来生成实现代码呢？

Ivar：根本不需要两类语言。假定你的设计是实现的一个抽象，为什么只是让一种语言来表达你的设计，而让另一种语言来描述实现呢？这就是现状，而且它导致了重叠。

我们要使用这两类语言有若干原因。或许最重要的就是因为我们没能让一般的计算机科学家看到建模语言的价值，他们觉得一种编程语言就足够了。现实是，代码是为机器（编译器等）而设计的语言，它不会使用人脑的所有功能。

在某些方面，我认为我们能清晰无误地展示视觉建模的价值，并说服计算机科学家在这个领域开展研究。很多研究都是针对 UML 的，因此，我们会使用两类语言并没有什么根本原因——但我们现在还不是这样。

这仅仅是说服人们关注它那么简单吗？

Ivar：这是让大学里面的人们懂得并非每件事都是用代码来完美表达的。其中很多人已经理解了，但这还不够。我们需要展示更多的成功。

从根本上说，UML 要比我们此前所有的一切都更好。SDL 在电信行业非常有用，不过 UML 是一种更为通用的语言（包括 SDL 中不可用的重要语言结构）。UML 是在 20 世纪 90 年代末期创建的，当时根本没有什么更好的基础语言，因此，你可以期望它在 20 年到 30 之内不会被取代。不过，到那时，我们可以改进教授 UML 的方式。

我相信随着时间的流逝，UML 的价值会得到证明。我们需要一些像 UML 这样的工具来帮助人们“伸缩”软件开发。或许更多具有软件开发实际经验的人会参与到该项研究中来。或许他们会展示那些我们现在教授给新学生不可伸缩的东西伸缩起来。

软件项目是否存在着一个规模限制，超出此规模使用 UML 可能会增加更大的复杂性和做无用功？

Ivar：从项目的成本来说，如果你在 UML 中增加培训和教育，而且在工具中培训和教育使用和支持 UML，这对于激励采用 UML 来说就太过昂贵了。不过，如果人们启动一个新项目，强制要懂得 UML 和如何使用至少一种支持 UML 的工具，那么，情况就大不相同了。

如果你想要在正常工作时间向人们教授基本的软件工程，它可能会很难激励他们，特别是在一个现有的小项目中。对于大项目来说，动机就会迥然不同，因为没有正确地建模风险会非常之高。

假设我很谨慎地使用 UML。您怎么能说服我它会有助于我的团队呢？

Ivar：这个问题的答案取决于你是谁。

如果你对软件一无所知，那就很简单了：你需要一种绘图语言，因为编写代码对于人类来说并不是什么好差事。代码对机器解释来说是很好的，但对于人类来说就不是这样了。

如果你是一个富有经验的程序员，我会问你如何描述你的系统、组件，以及它们是如何互相作用的。你如何从一个用户的观点来描述某个特定的情景？它会通过在你的组件或对象之间的相互作用而实现吗？没有编程语言能够以一种合理的方式做到这一点，因此它就是你可以使用 UML 的一个例子。还有大量的类似例子。

有些人我永远无法说服，因为他们已经使用很多很多年代码了。不过，如果他们必须使用一种完全不熟悉的语言（比如说 Prolog），或者一个新的语言类（比如声明式语言），或者是函数式编程语言（比如 Scheme 或者 Lisp），你问问他们感觉如何，他们很可能会觉得使用图形化语言也会获得很多帮助。

一旦他们理解了在构建系统的需要，我在说服他们使用 UML 方面就不会有任何真正的问题。

14.4 知识

有多少软件工程知识是与一种特定编程语言相连的呢？

Ivar：非常少。大学里教授的是编程语言，因此人们相信语言就是中心点。一般来说，真正的问题是理解软件。

你如何获取需求？你如何知道你是不是在构建正确的系统？你如何测试你已经构建的系统是否正确？你如何进行配置管理和版本控制？你如何完成你在学校里并没有学到的 30 或 40 个实践？

人们在学校里学习的事情都很容易。这就是学校里教授它们的原因。编程语言相对地易于教授和学习。我在美国麻省理工学院时，选修了 6001 个课程，其中我们使用 Lisp 的一个变体 Scheme 来描述计算机科学领域的若干现象。人们直接从高中选修该课程——他们在课程期间编写代码——而且它是我曾经上过的最棒的课程之一。你使用一种语言来描述编译、执行、解释等现象，以及计算机科学领域中很多有趣的现象。你还学会了编程的基本思想，因此编程实际上变得很简单。

现在我们有了框架，不过学习框架更难。这些事情仍然相对容易，它们只是成为优秀的软件开发者须要知道的若干事情之一。我们必须提高我们的软件工程能力水平。

我们应该找到一种方式，在我们需要时传递知识而不是提前传递。

Ivar：没错，而且你不应该扔掉你有的那些东西。从你自己那里开始。现在开发软件的每个人都有过一些不好的实践，不过那仍然是有用的。我们不应该设法立刻改变每件事，而应改进我们最需要改进的地方。

他们可能知道如何编程或进行配置管理，不过或许他们确实不知道如何去进行良好的需求分析和测试。这也有一些实践。他们可以保留今天所做的事情，并改变他们需要改变的东西，而不用抛弃已经纳入新事物的每件事。这是一个自然的发展过程。

我听说你预见未来智能代理将会与我们一起进行成对编程。为什么？

Ivar：开发软件不是火箭科学。来看一看 500 万到 1000 万的称自己为软件开发者的人们。他们其中极少数人能真正地做一些有创造力的或者原始性的创新。不幸的是，外界往往认为程序员是富有创造力、才华横溢的聪明人，而现实远非如此。

科学证据表明，软件开发者每天 80%的工作——不同的步骤和小微步——不是脑力劳动。他们每天做的事，此前已经重复了成百上千次。他们只是将一种模式应用到一种新的环境中。

当然也有创造性的工作，不过大多数人并不是这样。20%是脑力劳动。它仍然不是火箭科学；他们可能不得不以不需要提前考虑的方式来思考。

80%的工作是基于规则的。给定一个特定的背景，你可以应用一个接一个的模式来开发软件。这些模式不必定义，所以你实际上可能会使用错误的模式，并由此开发出很糟糕的软件。人们并不是一直都使用相同的模式，因此他们的一些软件很好，而另一些软件则很糟糕。 333

通过工具有一种方式来描述和应用这些规则。这是智能代理背后的观点。智能代理理解你需要应用的背景和活动，而且它们会应用该活动。它们自己可能会做很多事情，因为它们知道这些是不起眼的规则，或者它们可能会向与代理一起工作的开发者征求一些建议。

我创建的公司 Ivar Jacobson International 开发了支持软件开发的智能代理，并获得了极为成功的显著效果。Tata Consulting Services 使用一个相当小的规则集降低了 20%的成本。他们提升了程序员和开发者的质量并缩短了培训时间。他们能够让新员工很快地进入角色。

依我看来，毫无疑问这是技术在起作用。问题在于，人们仍然想要使用这么多种平台和不同类型的工具。如果你真正地想要开发这种软件，你必须对它进行改写以适应多种工具和不同类型的平台，因此，对于一家小公司来说，开发这些代理是非常难的。如果你达到了 TCS 这样的大公司规模，它就变得切实可行了。

潜在地，使用智能代理这样的技术有可能削减 80%的成本。例如，我们使用智能代理来指定用例、设计用例及测试用例等。那只是刚刚开始。我并没有怀疑：有技术，有问题，也有钱。

每个人都能够与互相计算机交互并问它要做什么，或者会一直在程序员和用户之间有很大的区别，这是最终目标吗？

Ivar：我认为越来越多的工作会由用户社区而不是程序员来完成。其中一种实现方法就是使用基于规则的编程。使用基于规则的编程，你并不需要实际理解执行，你只须编写你的规则。一个规则引擎会解释它们。这些东西人工智能社区都教了 40 年了，因此它不是根本性的新东西。对象技术有助于我们更好地理解如何去构建建模工具。20 年到 30 年前，基于规则的系统是庞然大物，而且很难修改。现在使用代理你就有了一种面向对象的专家系统，而且它更易于修改。

您是如何认识简单性的呢？

Ivar：简单性是变聪明、做聪明之事或总称为“聪明”背后的核心思想。爱因斯坦这样说过：“应该尽可能简单，而不是更简单”。我完全同意。这就是我所说的聪明。

如果你很聪明，你就会让事情尽可能简单，而不是更简单。每件事你都应该以聪明的方式来完成。当你设计架构时，你应该尽可能地少建模，不过要满足你的需要。如果你并不建模，你会花费很多精力来试图描述你在干什么，而且你也不可能有一个必要的概览。 334

例如，提前进行需求分析，而且试图在你开始构建之前确定所有的需求并不聪明。为了确定关键用例，或者关键特性并开始实现它们，因此你会获得一些反馈，这样才是聪明的。我已经确定了大约 10~15 个这样的聪明用例。

在我们工作和开发软件时，需要变得聪明一些。聪明是敏捷的扩展。敏捷主要是社会工程，尽管人们现在已经丰富了它的含义。你并不需要为了敏捷而聪明，但为了聪明你必须敏捷。我现在的演讲都是关于如何变得更聪明的。

14.5 作好变革准备

Be Ready for Change

您拥有麻省理工学院物理学士学位、加州理工学院天文学硕士学位和麻省理工学院计算机科学博士学位。您的大学教育背景对考虑软件设计的方式有什么影响呢？

James Rumbaugh：我认为，我的这种不同的背景赋予了我超出计算机科学课程本身的洞察力和综合能力。在物理里面，对称性是一个根本的概念，实际上它是现代物理的核心。我尝试应用这个概念来建模。例如，相对于大多数编程语言中使用的更传统的指针方式，关联为状态提供了一个更为对称的观点。在我的麻省理工学院计算机科学研究生涯中，我在 Jack Dennis 教授的计算结构小组工作，这是最先进行基础计算模型研究的小组之一。带着理性严谨的活跃性思维，这种激发环境直到现在仍然深深地影响着我的思路。

学生们应该更多地学习哪一个主题呢？

James：我对目前的学术计划并不是非常熟悉，不过在我的印象中，很多大学狭隘地专注于计算科学，强调具体的编程语言和系统，而不是理解重要的底层计算原理。例如，我很少见到程序员能够理解计算复杂性原理，并把它们付诸实践。他们反倒是过分讲究那些毫无意义的“小处聪明，大处糊涂”的局部最优化。

我认为在计算中最重要的技能（比如在物理和其他创造性领域）是抽象能力。不幸的是，经验告诉我，只有不到 50%的程序员可以正确的抽象。一位同事甚至说这个数字实际上是不到 10%。可能他是对的。不幸的是，软件领域的很多人可能并不具备干好工作所需的基本技能。

在软件领域共享知识的最佳方式是什么？我敢肯定人们不会去真正阅读上千页的手册。

James：如果你需要手头备有上千页的手册，肯定是你所使用的系统出了问题。它没有划分好。不幸的是，这个领域的很多人都对复杂性顶礼膜拜。IBM 就利用复杂性制造了一种崇拜。当然，这也有助于销售顾问。

工程师在接受培训期间，学习了很多技能，首先是在大学课堂上，然后是实际项目的在职培训。最重要的是去学习基本原理。在工程领域，它包括物理定律和某个特定学科的工程原理。在计算领域，它包括算法、数据结构、复杂性理论及软件工程原理等计算机科学原理。无论在哪个领域，找到如何做事的感觉都是很重要的。如果软件应用程序按照期望的标准进行一致性设计，熟练的开发者通常可以依靠直觉知道一个新系统的结构和行为，而不用去查阅一大堆手册。

提供系统工作指南也是很也重要的。仅仅列出组成部分并假定别人能够搞懂把它们放在一起应该如何工作，这是不够的。如果你尝试学习一个复杂的应用程序，比如说 Photoshop，使用一本显示如何创建基本命令来完成最常用任务的教程，就是最佳的开始方式。你可以一直使用复杂的命令清单来检查细节，不过这种学习系统的方式非常糟糕。不过，有多少系统开发者认为他们已经完成了工作，实际上他们仅仅是提供了构成该系统的一个复杂的命令或过程清单呢？这样无助于人们理解系统的工作方式。因此传送系统知识的最大缺点是关注过度地静态分解信息而不是使用模式。Pattern Movement（模式运动）聚焦于使用方面的思路很对，尽管它们有时候过于狭隘地从模式的观点来看问题。

您如何判别某人是否适合担任一个软件项目的架构师呢？

James：这是一种棘手的平衡。优秀的架构师必须能够平衡理论和实践，他们需要平衡优雅和效率；他们需

要平衡经验和远见。架构师的工作是保证系统整体结构正确，并做出具有全局影响的决策。这包括分解模块、主要的数据结构、通信机制及待优化的目标等。如果架构师在编码细节上纠缠不休，那他就很可能会在大方向上出现失误。

一位架构师必须能够有效地沟通，以致于开发者和程序员能够在一起工作。最糟糕的是你有一位架构师，他是一个天才，但是他无法把事情向普通人解释清楚。政治技巧肯定会加分，架构师的部分工作就是让各个竞争派系平稳地在一起工作。

一位架构师需要有大系统的工作经验。这是你在大学课堂和书本里面学不到的，你第一次担当重任之前需要实干经验。

我们如何在软件领域传播经验呢？

James：我过去常说，与其他创新性领域相比，软件的问题是没有程序博物馆。如果你是一位画家，你通过历代的著名艺术家来学习绘画，可以借助图书，也可以在博物馆观摩原始绘画作品。如果你是一位建筑师，你可以访问很多不同类型的建筑物。而在编程领域，程序员只能依靠他们自己。

Pattern Movement 提供了有用技术的目录，可能会适合多种不同的情况。这就是从顶级程序员那里获取最佳实践的很好方式，以至于其他所有人都可能会受益。

不过，人们还需要在一个完整的应用程序中有一切是如何相互协调的大型例子。最近，开放源代码运动提供了任何人都可以研究的大程序的例子。不过，并非是系统中的一切都是那么好的，而且新手需要别人引领他们通过。我们需要的是带注释的案例研究，这样的话，软件开发者就可以区分那些系统中的优劣之处，比如对国际象棋或一个公司对一家商业学校的评论。这些应该能够展现优秀实践的例子，而且也能指出可能干得不是很好的地方。就像学习任何技能一样，看看不良实践的例子想法来避免它，这是很重要的。

有多少软件工程知识是与一种特定编程语言相连的呢？

James：不幸的是，有太多的精力投入在了考虑具体的编程语言方面。设计一个程序的大部分精力可能会以独立于编程语言的方式来完成。当然，你无法忽略编程语言，而且在策略层次，你必须注意语言的基本属性，比如说它如何处理存储器、并发等。但是大多数设计都牵涉到数据结构、计算复杂性及分解成独立的控制线程等问题，这些问题超出了一种特定编程语言的范围。

它就像自然语言一样。你可以概括一篇新闻文章，而没有对语言留下太深印象。如果你在写诗，那么语言从一开始就至关重要。如果你是编写像诗一样的程序，那么你是太放任自流了。但是，当你坐下来写出实际的词或代码时，你并不是将它们从概要提纲（outline）中翻译过来，你使用你的语言知识来选择优秀的表达式。

我们一直会看到在程序员和所谓的用户之间的区别，或者每个人都能告诉计算机要做什么吗？

James：我已经注意到一些人能够清晰无误地以自然语言表达自己，而另一些则不能，因此，即使你能对计算机说你的母语，一些人也很难让人理解，因为他们不能清晰无误地思考。因此，人们之间总会有区别：有些人可以清晰无误地考虑并表达自己，而有些人则不能。

在话题受限时，还有一些更加简明的表达思想的方式。音乐符号是精彩、简洁的音乐表达方式，象棋符号对于象棋对弈来说也是非常棒的。为了确保构建你想要的建筑，绘制蓝图就是更好的方式，而不是试图完全用自然语言与木工对话。因此，你需要可以清晰、准确地思考并使用专门的语言表达自己的那些人。

我不会屏息以待：我们在可预见的将来某个时候，可以用自然语言与计算机对话。请记住：COBOL（通用商业语言）曾被期望成为一种使用英文与计算机交流的方式！因此，长期以来，人们对于自然语言交流太过乐观了。

关于创新、进一步的开发和采用你们的语言，您有什么经验和教训想要告诉那些现在和不久的将来开发计算机系统的人们吗？

James：首先，要想成功，你需要一些运气。我是在正确的时间处于正确的位置上。我们开发了 OMT，这是最早的面向对象方法之一，而且我们运气很好，写了一本书，它以极为简单的方式解释了面向对象。后来，方法可能也已经同样出色了，不过它们错过了时机。我还很幸运，在 GE Research 工作期间，正好赶上 GE 还没真正的软件业务。我不知道为什么他们让我们在这上面工作了这么长的时间，不过因为不必兜售大量的公司产品，我们确实能够心无旁骛地致力于此，与大多数其他方法相比，它为我们提供了很强的可信性。

我在 Rational Software 的经历更是五花八门。将三位顶尖的面向对象方法创始人聚在一起，这给我们提供了巨大的力量来打造 UML 并使其广泛被人接受。并不是 UML 比很多现有的方法（尽管它使得做事磨掉了独立方法的一些棱角）要好很多，而是它能让大多数人用同一种方式来沟通，且不是在争论不同符号的优点以及各种晦涩难解的区别。不幸的是，Rational 无法通过足够快地构建高效、易用的工具，来进一步继续方法学的成功。我认为，高层管理者或者大多数开发者并不会真正相信建模——他们仍然相信 Heroic Programming（个人冒进的编程方式）——而且它也是用工具来展示的。为什么你应该从自己也不会用的人们那里购买一种工具呢？当看法有所改变时，已经为时太晚了。另一个教训是，你必须信任你做的东西，否则它就不会成功。

OMG（Object Management Group，对象管理组）（译注4）就是政治干涉戕害所有好主意的一个案例。UML 第 1 版足够简单，因为人们没有时间来添加很多乱七八糟的东西。它的主要缺点是有一些不一致的观点——一些方面相当高级，而其他方面则又与特定的编程语言紧密相连，这就是第 2 版应该清理的。不幸的是，很多人妒嫉我们最初成功的人参与了第 2 版的工作。

他们觉得自己能够做到我们那样。(而事实证明，他们无法做到。) OMG 过程允许将各类特殊兴趣都塞进 UML 2.0，而且由于该过程主要是基于协商一致的，所以几乎不可能剔除糟糕的主意。因此，UML 2.0 变成了一个膨胀的庞然大物，里面充斥着过多有问题的内容，而且仍然没有一致性的观点，也没办法去定义一个。它就是像一个塞满了各种好东西的糟糕的预算账单。它显示了通过“投票”（committee）试图进行创造性活动的局限性。

整个过程证明了 Brooks 的第二系统效应。如果你从未读过 Fred Brooks 编写的《人月神话（Mythical Man-Month)》[Addison-Wesley Professional]，那么今天就去拿过来读读。它绝对是迄今为止有关软件工程的最棒的一本书。令人沮丧的是，这本 30 年前的旧书里引用的大多数问题今天仍然在发生。正如 Brooks 所言，管理者试图为延迟的项目添加人力，结果却导致它们延迟的时间更长。

或许这是描述面对计算领域主要问题的很好方式：大多数专业人士并不了解计算历史，因此，正如 Toynbee 形容世界历史那样，他们被诘难重复犯同样的错误。与在过去的发现基础上构建的科学家和工程师不同，太

译注4：对象管理组（OMG，Object Management Group）是一个国际性开放会员的非赢利的计算机组织，成立于 1989 年。任何组织都可以加入 OMG 并且参与标准制定的过程。OMG 的 OOOV（One-Organization-One-Vote）原则，保证每个组织无论大小，都拥有有影响力的发言权。拥有约 300 家机构的国际联盟，它开发了对象管理体系结构（OMA），制定了 UML 标准。

多的计算专业人士把每一个系统或语言都当作新东西，而未意识到类似的东西早已经做完了。在 1986 年的首次 OOPSLA 会议上，亮点是展示 Ivan Sutherland 在 1963 年创造的 Sketchpad 系统。在创造出面向对象很早以前，它就融入了一些面向对象的最初思想，而且它比现在大多数面向对象系统做得更好。它在 1986 年看起来很新鲜，20 多年后仍然如此。为什么我们现在仍然还有很多不如 Sketchpad 的图形化工具呢？

为什么我们今天的操作系统中仍然还有缓冲区溢出错误（一个丰富的恶意软件漏洞源）呢？为什么我们仍然使用像 C 和 C++这样的语言，它们并未理解有界数组的概念并因此容易诱发缓冲区溢出错误？当然，C++可以定义有界数组，不过程序员仍然在滥用裸指针。它全部要归咎于开发者的无知、懒惰，或者是自负。诚然，计算是很困难的，而且复杂系统中的逻辑错误在所难免，不过，过去了这么多年，仍然在犯这种低级错误，实在不应该有任何借口。

那么，开发一个新系统要搞清楚哪些主要问题呢？首先，要搞清楚它用于什么和为谁服务。不要一开始就雄心勃勃——最好是先拿出一些有用的东西，然后再迅速添加上去，而不要试图将你可能需要的东西全部都考虑清楚。这就是 Agile Development（敏捷开发）的优秀原理。你无法包打天下，因此要准备好做出某些困难的抉择，而且还要懂得如果一个系统成功了，它可能会按照你无法预测的方式发展，因此，要为不可预期的变化做到未雨绸缪。

14.6 使用 UML

使用 UML 生成实现代码，你对此怎么看？

James：我认为这是一种可怕的想法。我知道我与其他很多 UML 专家意见不一致，但 UML 没有魔法。如果你可以从一种模型生成代码，那么它是一种编程语言。而 UML 不是一种设计良好的编程语言。

最重要的原因是它缺乏一个明确定义的观点，其中，部分原因是由于它自身的目的，部分原因是因为 OMG 标准化过程的专横，它试图为每个人都规定好一切。它没有关于内存、存储器、并发等几乎所有东西的一个明确定义的底层假设集。你怎么能使用这样一种语言来编程呢？

事实是 UML 和其他建模语言并不意味着是可执行的。模型的问题是他们并不精确，而且含混不清。这令很多理论家抓狂，因此他们设法让 UML“精确”，而模型不精确有一个原因：我们不考虑影响很小的事情，因此我们可以把精力集中在具有显著效果或整体效果的事情上。这就是物理模型的工作方式：你对“大效果”（比如太阳万有引力）建模，然后再处理对基本模型（比如行星相互作用）有微小扰动作用的“小效果”。如果你试图直接在细节层面解决整个方程组，你就什么都干不成。

我认为，很多新近在 UML 上做的工作都被误导了。它从来没打算要成为一种编程语言。使用它的目的是获得正确的策略，并以一种合适的编程语言来编写最终的程序。

不幸的是，我并没见到任何真正好的编程语言。它们全部都有极易诱发错误的很多缺陷。整个 C 家族（C、C++与 Java 等）严重不足（句法几乎是不可分析的），但我们坚持使用它们，无论我们喜欢与否。很多新的时髦语言在它们的开发者方面显示了对于任何严肃的语言理论的无知。另一方面，很多学术性更强的语言对于它们自己的好特性和瞧不起的重要特性过于优雅，比如需要多个团队在同一个系统上分别工作。

一种语言可供多个开发者团队使用，它都需要些什么呢？

James：让我回到早期的编程语言 Algol-60，它是我使用过的第一批语言之一（很可能是在本采访的很多读者出生之前）。它引入了很多重要的概念，比如 BNF 句法符号、递归子程序及结构化控制结构等。在很多方面，它比 FORTRAN 语言更为清晰。不过它有 4 个主要缺陷，使它实际上并不可用：它没有内置的输入/输出结构；它不支持双精度和矩阵运算；它不支持子程序单独编译；它也没有连接机器语言和 FORTRAN 子程序的标准接口。在理论上，所有这些都是一些很小的问题，不过却是主要的软件工程障碍。

340 很多学术语言犯了同样的错误：它们解决了有趣的数学问题，不过忽视了实用问题：一种语言如何用于环境之中，因为实用问题在理论上并不会令人感兴趣。正是这些小事决定了一种语言的可用性。

首先，开发者需要能够在系统的隔离部分上工作，而不需要其余系统的声明或代码。然后，他们需要一种方式将各个部分组合起来，并确认它们可以作为一个系统工作；我认为这需要一些声明类型的概念。你需要适应不同类型的通信机制，因为系统现在是高度并行的。大多数语言并没有很好的方式来描述或声明动态行为。我认为你的调试工具要更好地和语言集成在一起，不过在运行时应可以打开或者关闭它。现在，编码和测试之间的分离太过份了。

UML 仅仅是供大型开发团队协调工作的一种工具吗？

James：对于引导和组织个体开发者来说，它是第一种也是最重要的一种工具。你需要在不同的抽象层次上工作；代码是一个特定的层次，但不是理解如何系统工作最有用的层次。你需要工作在一个更高级别的层次上，这意味着要远离所有的细节代码，转到更高级别的重要事情上来。这就是 UML 可执行的错误原因之所在；它会破坏抽象的核心问题。

说到架构师，您强调了良好沟通的重要性。UML 会有助于解决这个问题吗？

James：它提供了一种通用的概念和符号集。这有助于沟通。如果你没有共享相同的词汇表，你就无法进行沟通，或者反过来说，你认为你是在沟通，但不同的人却有着不同的理解，这甚至比没有沟通更糟糕。

架构师能够使用 UML 更好地沟通吗？

James：噢，这就是它的核心问题，不是吗？

我在 GE 内部开始推动面向对象时，我访问了 GE Aircraft Engines。在说服程序员面向对象是一个好主意时，我们遇到了极大的困难——他们坚持现有的概念（比如 FORTRAN 编程），而且也不懂我们在说什么。不过，一些飞机工程师，很精确地理解了我们说的东西，并对面向对象的观点感到非常兴奋。在他们的工作中，他们过去习惯于制作模型和抽象高级概念，比如“发动机性能曲线”或 “失速速度对攻角”。他们过去习惯于创建心理对象来表示物理概念。程序员只见树木不见森林——他们沉浸于代码之中，而没有意识到代码的目的是表示更高级别的概念。现在还有很多人没有意识到这一点。

341 **ML 的创始人 Robin Milner 举例说明了模型层次的观点，他把一切都连接在一起，从高级设计语言（比如 UML）到低级汇编代码，到硬件之后的物理模型，再到作为使用了硬件的环境的一部分额外模型。他举了一个飞机的例子，其中，你使用了延续至硬件的高级代码，硬件自己带有模型，然后按照空气动力学、物理学和天气模型设计整个飞机“硬件”！当飞行员按下按钮时，所有这些模型都参与进来了。**

我们应该减少层次数（对一个通用模型/语言来说），或者提升抽象（和层次数）？

James：你说到点子上了。来自物理学（更确切地说，一般科学）的主要概念之一是，涌现观点的多层次思想。每个观点都是构建在低于它的基础之上的，一旦构建之后，都是前后一致的而且能自我表明含义。因此，我们拥有很多层次，比如说量子物理、化学、微生物学、生物组织、种群、生态系、环境等。另一个等级塔是计算：材料物理、半导体、电路、数字系统、计算机、固件、操作系统、应用程序框架、应用程序及网络等。没有哪一个层次是“对的”或“真的”，或者是“基础的”层次。每一个就自己来说都是有意义的，而且可以通过下一个较低的层次来定义。不过，这并不是意味着它会被较低的层次所理解。每个层次的含义都是唯一的，而且只有在那个层次上才可以理解。这就是一个涌现系统：每个层次的含义都是来自于一个简单的较低层次，但必须是可以以它自己的语汇来理解的。因此，为了理解任何复杂的系统（当然，宇宙是终极的例子），我们都需要同时在各个层次上工作，没有哪一个可以说是主要的。

emergent levels 之塔，建模语言并没有做得很好。我们需要一种方式来在多个层次上同时对系统进行建模。我不是在说 OMG 4 级元模型。之所以提到它是因为有些人犯了与 Bertr 和 Russell 同样的错误：假定你无法在自身方面为某些东西建模。当然，你可以；读一读 Douglas Hofstadter 的著述吧，比如《I Am a Strange Loop》[Basic Books]。还有瞧不起建模的代码编写者失败的例子。他们认为代码是唯一重要的。就像是说电路是唯一重要的，或者说半导体物理是唯一重要的一样。所有的层次都很重要，而且为了一个特定的目的，你需要在正确的层次上工作。我认为，对于理解一个对人类有益的复杂大系统来说，在代码层次上出错是非常糟糕的。

单个通用语言没法工作。我们需要的是一个框架，它允许我们工作在多种抽象层次上。这就是 UML 2.0 应该干的，而它却没干。不是它添加的细节是错误的，不过他们让事情变得更复杂；成本效益分析很差。主要的问题在于缺少一个整洁的方式来为构建涌现模型的各层保持不同的层次相互独立。

例如，UML 包含更适合于编程语言的相当低级的概念，比如说权限或指针，它也有高级概念，不过它并没有一种很好的方式来区分低级概念和高级概念。Profile 是一种尝试，不过它们并没有真正地获得成功。在构建 UML 期间，编程语言概念和高级逻辑概念之间的紧张关系引发了很多争吵，甚至是达到精神分裂的地步。

其他主要的问题是不同的人们开发的不同部分之间的风格差异问题。例如，消息顺序图对 UML 贡献很大，非常有用，不过它与活动图等东西的风格差异很大。

您会使用什么过程来简化语言呢？

James：我怀疑它可能会通过标准过程来简化，比如产生 UML 2.0 的 OMG 过程。有太多的竞争利益攸关方都试图将他们自己的想法加到里面去。问题在于，标准化过程很少强调一致性、简单性及统一的风格。取而代之的是，他们过于强调过多的内容。事实上，我不是标准化机构的拥趸；他们趋向产生塞得满满的产品，而缺少优雅性和可用性。首先，我不愿意跟他们在一起掺和，我对其负面效应的担心已经应验了。

对于一个人或更多的人们来说，最好的办法就是做出他们自己的删减版 UML，并让公众通过应用来决定。这种产物不必再叫做“UML”，因为这个术语可能具有特定的法律和感情之累。任何语言开发者都能够清晰无误地表明他们版本的目的，而不是试图包打天下，这是非常重要的。

您如何认识简单性呢？

James：这需要自愿做得更少而不是更多一些。请记住，建模语言不是编程语言，如果该语言缺少某些功能，建模者总是会往上添加东西以弥补不足。如果你不得不随身携带一个大号登记卡来记住语言的所有特性，那它就不够简单。如果一个简单易懂的场景建模面临着四五种可选方案时，那它也不够简单。

342

假设我怀疑 UML。您如何说服我：它会帮助我呢？

James：我自己也对它有很多怀疑。我认为它已经太过膨胀了，OMG 这个厨房给它放了太多的饭菜。人们也试图让它成为一切、一切的答案。整个计算领域有一种趋势：超出理性地大肆宣扬所有的新进展。还有另外一种趋势：寻找一个放之四海而皆准的单个解决方案。对于简单的解决方案来说，生命和计算实在太复杂了。

UML 这个工具，对于数据结构设计来说非常有用，对于将系统分解成分层模块有一定用处，而对于动态的事情它处理不好，因此用处不大。它是有益的，不过，它无法解决你的所有问题。你还需要很多其他技能和工具。

软件项目是否存在着一个规模限制，超出此规模使用 UML 可能会增大的复杂性和做无用功？

James：没有限制，不过，这并不是说：对于很小的项目和很大的项目，你都要以同样的方式使用 UML。在小项目中，在使用工具、模型和软件过程等方面，你会有很多很多的“形式”。对于小项目来说，类和数据结构的类图仍然是有用的，不过，或许不会有这么多的往返设计。因此，UML 提供了一种概略描述初始设计的方法，不过，最终你会使用一种编程语言，并保持不变。

在大项目中，有一半或更多的开发过程是交流，而不仅仅是设计。如果是那样的话，必须有分解系统、控制访问模型和代码及跟踪程序的工具和过程；否则，人们总是会相互捣乱。我知道很多程序员抱怨必须让它们自己遵守这种纪律。在体育运动、建筑、新闻写作、火箭宇宙飞船设计，或者几乎任何其他类型的大型合作冒险行动中，这些抱怨者将会遗憾地被团队剔除。如果我们想要认真对待它，现在就是我们在软件中采用那种观点的时候了。

14.7 层和语言

Layers and Languages

在您第一个答案中，你说过你认为模式运动在做一个伟大的事情，不过它们的观点太狭隘于模式。您能展开谈一谈吗？

James：在我曾经参加的一个研讨会中，包括 Hillside Group 的一些人，几乎是一种准宗教的态度来看模式，而且他们实际上并不想扩展它，结果证明他们这个观点很狭隘。他们非常维护关于模式的官方观点，而且对架构师 Alexander 有点儿顶礼膜拜的意思。他们把模式看成是个很具体的东西，而且认为模式可以应用于各种不同的等级。

并不是每个人都认同这个小到中等规模的模式的具体观点，这些模式出现在四人小组（Gang of Four）（注 1）编写的《Design Patterns（设计模式）》一书和 Hillside Group 的东西等之中。实际上，即使是在模式运动内部也有一些分歧，不过我认为这个概念变得流行起来了，那才是主要的。

人们所说的模式与他们在其他很多学科里所说的一样，也就是我们应该从非常熟练的人们那里搜集经验，并按照分类把它们编写出来，供更多的普通人来仿照遵从。像工程、绘画和建筑之类的几乎所有的创造性领域都有这样的事。在计算领域，人们要接受“从过去的经验中学习”这个理念稍微会有点慢，现在仍然如此。那

注 1：这个术语是指 Erich Gamma、Richard Helm、Ralph Johnson 以及 John Vlissides，《Design Patterns: Elements of Reusable Object-Oriented Software》（Addison-Wesley Professional）一书的作者。

就是我的抱怨之一。在计算领域中，很多人自我感觉良好，看起来好像忘掉了几事都有过去的根基。很多人还在重新做那些已经被发现了的事情。

我认为模式观点会反击该问题说，“看，有很多事情我们可以获取、理解，并使它们为领域中更多的人们可用，他们可能无法创建这些事情，不过如果它们被正确地描述，他们就可以使用”。

在任何领域这都是对的。在任何领域中，能做出重大突破的人都是很少的。在最初的突破之后，其他人就接受了这种思想并对其进行扩展。也就是说一旦公开了这种思想，创建者就失去了对它的控制。你无法让一个内部小组来控制这个思想的含义。新人会发现创始者没想到的额外含义。

您刚才谈到过您的物理学背景，在物理学中有很多层次，我想要说一些类似的事情。在某一层适用的东西可能在其他层上并不适用，不过你也无法否认其他层存在的现实。

James：这就是整个涌现系统的概念。那就是关于科学的一切，那就是关于语言的一切。这就是来自复杂性理论的一个概念。你感觉到了涌现性，而它没有基本层。毫无疑问，人们早就在计算中理解了这些。你拥有多个层，而且这些层里没有一个是真正的层。

当你说模式运动已经把自己的目的限定得很窄了时，你认为在它的地址应该是整个堆栈时它应坚持在一个层上吗？

James：让我来公正地评价模式运动，人们在不同的层上工作。例如，架构模式方面有好几本书。或许 Hillside Group 最初对什么构成了模式有一个非常详细的看法，并把它和整个 Alexander 模式语言的概念绑定在一起。但我认为你需要以一个更宽泛的方式来使用这个词。

《Design Patterns》一书招致的一个普遍批评是，在除了 C++或 Java 之外的语言里，这些模式实际上并不是有效的。

James：模式的要点之一是他们必须具体针对你的工作。你不能把什么都拿过来，并把它软化成一个非常通用的方式。如果你编写编程语言模式，其中很多都是具体针对于一种特定的编程语言，而且有一些还可能是通用的，并可以跨多种编程语言工作。在工程中也有这种情况：一些东西是用于钢铁而不是用于木材，反之亦然，而有一些东西则是通用的。

或许这就是我对建模的抱怨。UML做不好也得区分在哪里使用。可以将建模应用于具体的编程语言，也可以将建模用于逻辑性更强的事情上。UML 二者都有，而且是它把它们乱七八糟堆在一起的。我会对此负部分责任。在第 1 版中，我们将很多不同的东西混在了一起。这是发生在第 1 版中——你并没有把如何分开那些东西的经验。

我曾经希望在第 2 版中我们把它整理好，因此你就可以说，“这里的这些特性是应用于 C 或 C++的”，因为那里有些特性有点像 C、C++及类似的语言。把它们放在那里没有问题，不过它们不像跨多语言的特性那么通用，最好弄清楚它们之间的区别。

建模特性既有高层次使用，也有低层次使用。Profiling 机制设计用于允许从某个特定的角度来定义特性，但不幸的是，profiling 机制也是一个笨蛋。它不允许你适当地分层。它说“这是一个领域”，但这在某种意义上来说就是一级的事情，没有方式能将它们整洁地组织到建模等级之中。

假设您能够创建 UML 3.0 并中断向下兼容性。您会怎么做而绝对不会打扰每个人？

James：根据你的前提假设，你恰恰保证了会打扰每个人。当然，这是 Microsoft 和 Apple 一直面临的麻烦。你会像 Microsoft 那样维护很多代产品的兼容性，或者你会像 Apple 那样偶尔中断它吗？这二者各有优劣。

你无法永远维护兼容性。那是不可能的。在任何领域，你最终都不得不说“对不起，我们不再那么干了”和“对不起，你去买些新的来”。来看一看模拟电视机吧。美国在 2009 年 6 月以后，他们就不再销售天线。直到改变发生时为止，很多人是不会发现的。大多数人读到这类文章时都会意识到有这样的事，不过，要中断兼容性没有容易或者无痛的方式。它只不过像是遗留系统的问题而已。

人们总想找到解决遗留问题的神奇解决方案，而事实是根本就没有容易的解决方案。这是一个很棘手的大问题。你必须得卷起袖子往深里挖。它不像是一个具体的问题。你试图做两种互不兼容的东西，而且它总是要带来一些痛苦，总是要在让人们不高兴时做出一些决策。

你更喜欢大刀阔斧一次到位，还是小打小闹连续不断？

James：对于任何新系统来说，你总会发现错误。你可以暂时做一些局部修改，不过最终你会发现你的一些基本假设和架构决定不再成立，而且你已经开始要做一些重大的修改并完全重新开始，因为否则它就无法继续发展。我称之为“大地震”。第一次你可以每个系统做一下。第二次可以说难以实现它，因为你有很多内置的承诺。最终，你不得不无情地以你无法再做重大修改的方式而告终。我认为很多系统都是这样的——在计算机系统当中，在我们周围世界中的其他不同的系统，其中所有东西都被锁定了。最后，一些人拿着亚历山大之剑（Alexandrian sword）通过，并找到了一个取代旧方法的新的解决方案。它不是在旧东西上修修补补。一些人带来了一些新东西，并使那些旧东西失去了意义。

我认为那肯定是计算的真谛。这个领域是有困难，而你实际上恰恰是要来解决这个难题。我不知道有什么可发愁的。事实是我们不必总是坚持同样的事情，因为人们确实带来了新东西。如果你看一看 19 世纪的运输问题，你会说：“我们预计以后到处都是马匹，大街上到处都是马粪，而且我们也没有那么多马厩！”到 20 世纪又讨论每个人最后都得变成电话接线员。之所以如此，是因为人们的思维无法超出它们所处的时代。

随后发生的是，一些人带来了做事情的新方式，这是此前人们从未有过的想法，它使得那些旧问题变得毫无意义。我认为这一点非常令人振奋。对于那些一生对某种做事方式从一而终的人们来说，这可能是令人沮丧的。总之，过去的 50 年已经告诉我们：你无法选择一个职业而坚持不变；一生呆在一家公司或期望在你所学的专业领域干到退休，这个时代已经结束了。你必须为变化做好准备。

Heinlein 说过，“蜕化是为昆虫而存在的”。

James：昆虫繁殖很快，而且会大批死亡。这种方式对人类来说并不美好。令我沮丧的是，很多公司看起来好像要走那条路。他们想要聘用“昆虫”，他们聘用专于狭窄系统的人们，他们想让这些人可以马上上手，他们想在这些人干完活后过河拆桥。我认为这是一种令人沮丧的趋势。我认为从长远来看这种方式并不可取。我们需要的是可以思考、改变并在必要时进行学习的人们。我认为去学校学习一种特定的编程语言并没有多大价值。你去学校学到的只是一种编程语言的概念。学习一种新语言是很容易的。我们需要能够改变的人们。人们需要观察什么在发生变化。有一种说法是，现在每个人都需要继续教育他们自己，这话说得很对。

你对 UML 的一些批评有可能解决吗？

James：肯定有可能。我认为这整个标准化的概念更是一种营销手段，而不是别的什么。你为什么需要一种

标准化模型语言？你需要将实际上执行的事情标准化。我认为这整个标准化的概念大大吹嘘过度了。你建模并不需要标准。如果 UML 过于膨胀，人们会划出他们需要使用的部分。每个人都使用类模型，而很多人使用顺序图和用例之类的东西。有一部分 UML 几乎没人使用。这就显示出了将太多聪明人聚在一起而没有办法强制决策的危险。他们会提出很多看起来有用的想法，而且也没有办法说："这是一个好主意，但它的用处尚不足以放入要为很多人服务的大锅当中来"。

人们会使用他们想要的东西。实际上，它确实与使用 Photo-shop 之类的东西不同。我能够使用 Photoshop，但我不是专家。我并不记得如何去使用它的大多数功能——如果需要的话我可以去查它，不过我知道如何去调整级别、做出选择，以及干其他一直要干的事。如果我必须做别的事，我可以去查它。专业图形设计师会知道得更多。这就是人们使用大多数大型应用程序或设备（比如蜂窝电话）的方式。他们不知道如何使用每一个特性，因为大多数特性被放进去只是准备作为销售卖点的。

很难评估建模语言的可用性。我可以请人使用一种编程语言来解决实际问题，并向我反馈它表现如何。

James：如果你来看一看那些程序语言，我就有理由说它们中间很多没有遵循该范式。在很多情况下，开发者把自认为聪明的想法放到语言当中，而没有经过很好的可用性测试，从而引发了各种各样的问题。

语言设计者提出这些他们觉得必须解决的假想实验。有时你应该说："好的，我们并不需要解决那个问题；人们可以采取另外一个方式来解决它。它还没有重要到足以成为语言的一部分"。

在建模语言、编程语言或应用程序系统之类的系统中，你想要添加一些特性，如果这些特性被证明确有需要，人们可以记得它们的工作方式，然后你需要进行测试以确认它们能实际工作。如果你往里塞了太多的东西，那就没有人能够记得如何使用它，因此，它实际上就变成了一种负担。它很可能会给系统带来问题，因为开发者要测试的东西越多，错就会出得越多。你不能只是问什么东西是否有用，你必须问如果它足够有用，与其他的比起来，是否值得付出"记得如何去使用它"这样的成本。

对人们来说，能够发展 UML 的一种方式就是使用它的子集，毫无疑问这种子集一直都有。我怀疑 OMG 过程肯定是用来解决所有问题的，因为它不是决定性的。我认为 OMG 做的 UML 3.0 并不会获得成功，因为里面有那么多的竞争利益。有太多的人想往里塞东西，因此你无法使它保持简单。

如果你让我加入我的愚蠢特性，我就会让你加入你的愚蠢特性。

James：确实如此。里面的折衷交易实在太多了。还有另外一种方式，那就是，一些人提出了一些新东西，它是基于 UML 的，不过换了个新名，并和基本的设计决策有所不同。当然，最终的结果就是里面什么东西都有。否则，人们就会决定不在这儿玩了，转移到其他地方去了。

14.8 一点可复用性

看起来软件的平均复杂性和规模逐年增长。面向对象编程能很好地适应这种情况吗，或者它会让事情变得更复杂吗？

James：首先，系统规模的增长速度是不清楚的。你无法在一个程序中度量字节数。如果你生成代码，那么源代码行数就很重要，而不是形成码行数、字节或其他什么东西。如果你使用高级程序（higher-level

procedure），那么复杂性就取决于调用的数量，而不是执行代码的行数。随着在更高级别上工作，我们获得了更大的系统，不过它们的固有复杂性可能不会随之增大到那种程度。

即使你不买它（而且我也不是全部买下来——系统看起来确实好像变得更复杂），面向对象系统也是值得继续发展的优秀方式。不过，你需要将面向对象与主要关注可复用性区别开来。我知道，最初的 Smalltalk 开发者将构建可复用部分当作面向对象的中心，但我认为这是一个错误。你可以使用面向对象结构，而不必过分聚焦于试图构建一个可复用的组件库，此组件库中包含了你在应用程序中使用的每一个类。可复用性是很不错，但它实际上并不是大多数系统的主要目标。构建一个优秀的可复用库其实是很难的——大多数程序员都干不好，也不应该鼓励尝试。这是一种不同于系统构建的任务。设计系统时，使用面向对象结构来构建由类来组成的一个清晰的应用程序，这些类在需要的情况下易于修改，而不是坚持它们可以立即被其他人复用。如果发现你最后多次使用了一个类的变种，那么你就可以费神让它真正地可复用了。

你还必须知道什么时候停止。我已经看见过大量的初学者在让每一行代码都成为可复用对象方面焦头烂额。当你可以使用自然语言表达一个简单易懂的算法并编写代码时，那么就不会被切成更小的代码段而烦恼——只要编写代码就可以了。面向对象是要在复杂情况下提供更高级别的结构，而不是为了应付这些小东西。

我们怎么能确信面向对象的优点要比缺点更有价值呢？

James：如我此前所言，你可以一直使用面向对象结构。问题是知道如何“低”地用它。真正的信徒一直想要推动它继续发展。在计算领域，除了面向对象结构之外，还有很多其他的问题，比如说优秀的算法、优秀的数据结构、可接受的计算复杂性及理解能力等。并不是所有的都和面向对象有关；事实上，面向对象只是整体的一小部分。面向对象为组织设计和编程提供了一种有用的框架。这一点很重要，否则你就会被问题湮没，乱得一塌糊涂。不过，任何设计的本质内容根本不是面向对象——而是我所提到的其他的所有事情。

你提到可复用性不是面向对象的焦点。

James：我认为它并不应该成为焦点。可复用性从一开始就被大大地吹嘘过度了。它是一个卖点，可以说服很多项目经理来买它。可复用性非常难，让产品真正地可复用需要很高的技能，远远超出了大多数人的能力。有人说，我认为是 Brooks 或 Parnas 说的，形成产品要比实验室原型难上三倍，而要让它在可复用的基础上工作还得再难上三倍。当你以最快的方式做某件事情时，如果要坚持让它可复用，实际上大多数时候你是在浪费时间和精力。

不过，你可以采取一些预防措施，让未来的修改更容易一些。首先，如果你能以更通用一点的方式来做，那你就不要使用一种很特殊的方式。如果不必要的话，你就不要让自己陷入困境。如果你能看到把它概括出来而没有增添太多的麻烦，那你就干吧。设计时要照顾一下以后不得不修改这个事实。这并不是说你在编写第 1 版时必须完成所有的概括化。你留了一扇门，你为修改插入了钩子，你把那些编写成可以替换的方法，但你并不需要马上编写那些过于具体的方法，直到你知道你需要它们为止。问题在于，很难猜测未来你需要什么，而且猜测通常都是错误的。如果你在概括一个领域上花费很多时间，你很可能会发现那个领域并不需要修改。

它只不过像是最优化的。实际上，我认为太多的程序员担心在错误的地方运行得很快。这个问题已经困扰了这个领域很多年。即使计算机变得更快，那也没关系。他们仍然会被我所谓的最优化困扰。他们不懂得复杂性理论，他们不懂得“数量级”，但他们担心很小、极小的速度改进。

我编写了一个子程序包，然后我去把它描述了一番。你不能总是在猜哪里需要优化，你已经有过考虑，然后

你再去解决那些问题。一个子程序要花费30%的时间。我解决了那个子程序。

人们过优化（overoptimize）了。它很可能会使程序失败，而且它可能不是一个重要的位置。我认为这是人们并不了解的一个领域。人们说“效率很重要。你知道，把它嵌进去”。废话！一切都在飞速变化。进行过度的最优化，最后导致bug遍布，付出这种代价并不值得。我并不理解人们为什么会在做他们能做的简单事情上出现失败。由于很多开发者的思维倾向是这些事情仍在发生，所以一些事情错得一塌糊涂。我认为他们有些人喜欢采用困难的方式来做，就像不用绳索攀岩一样。不过，如果他们使用他们编程的方式来攀岩，那就必死无疑。

如果复用不是面向对象的目标，那还有什么目标吗？

350

James：复用是由面向对象推动的，不过我认为面向对象的主要目标是使用，而不是复用。如果你以这种方式来构建，它会更易于一次成功，而且它也会具有可修改性。你可能会获得一些复用，不过那只是一个附带的优点。

你知道，你必须修改应用程序，不过你不知道从何处下手，因此以面向对象方式来构建有助于未来进行修改，因为它产生的是易于修改的结构。要点是使它可修改，而不是总是去构建可复用库。你开始构建一个应用程序时，并没有期望公司里的其他人都要在你的应用程序中使用每一个类。你公司里的其他人，可能是你也可能是别人，最终必须要修改你编写的那个程序。你知道那肯定是要发生的。面向对象使未来的修改变得很容易。我认为这是它最重要的价值所在。第一次编程时，只要能简单地使用所有的方法就行。在第2版时，面向对象才会大显神威。

因为你已经把它们封装起来了？

James：没错，实际上，你已经得到了一个更整洁的设计，里面只有很少的纠结。这种构建方式可以与它自身打结保持同样多的功能。这是人们很难处理的真正问题：很复杂的功能。在某种意义上来说，你是在同一应用程序的下一个版本中复用它。如果你愿意的话可以称之为可复用性。在很多不同的项目中，只有很少一点儿会被复用。

你在一个较宽的基础上开始复用之前，实际上你需要用它三次。第一次总是一个特定的情况，第二次可能是一个巧合，第三次，好的，现在你开始理解一些事情的模式了。现在或许值得费心把其中一部分拿出来并使它们真正地健壮。你并不需要复用一切，你会找到最有用的东西并复用它们。

这使我想起了SOA，它的假设看起来好像是你可以定义跨整个企业可复用的服务。

James：在我看来，SOA更是一种营销工具，而不是别的什么。我从来没看到它里面有什么更深层次的东西。它是相当肤浅的。

这些东西很多都是营销工具。当然，你不得不使用这些东西。我从一个早期的项目中学到，找到一个优秀的名称值得一年不干这个项目。在我的职业生涯中，我提出过一个很棒的名称，而且它确实助益良多。人们会受到这些东西的影响。使用OMT和UML这两个名字——麻烦在于，我们找不到更棒的名称。我并不喜欢首字母缩略词，如果有可能的话我就会避免使用它们，不过有时候它们是你的最佳选择。

在面向对象模式与对并发的新关注之间，您认为有什么联系吗？

351

James：作为数据结构和行为的自包含包，对象的概念对于并发来说是理想的。在现实世界里，一切都是自包含的，一切都是并行的，因此在建模中对象的观点对于并发来说是很完美的。

不过，这并非是编程语言本身就有的。人们学习的几乎所有的编程语言本质上都是顺序式的。可能有几种学术性语言是固有的并行，不过大多数人并不会去学习那些语言来编程。你可以为 Smalltalk、C++或者 Java 这样的语言添加并发特性，不过底层的计算模型和观念模式仍然是固有的顺序式。

因此，问题不在于是面向对象模式，而在于编程语言和系统。当然，你可以设计并行语言。我在 1975 年做博士论文时设计了一种并行语言，20 世纪 70 年代早期，我的一些研究生同学在美国麻省理工学院 Jack Dennis 计算结构小组也是做这个的。要拼凑出一种新语言来并不难（尽管难以拼凑出范围很宽，且实际问题都容易使用的一种新语言）。难的是让它们获得应用。

创造语言时并没有钱——一种流行语言的本质是广泛使用，而且人们并不想要使用一种私有语言。对于一家公司来说，如果他们必须放弃的话，很难捐出所需的资源（否则如果他们放弃的话，你可能会怀疑他们的动机）。不要认为创造一个更好的捕鼠器毫无意义——它会让采用某些东西彻底市场化。因此，我并不是非常希望优秀的并行编程语言会被广泛采用——我并没有看到激励它出现的迹象。

我们怎么样来设计一个“本质的”并行编程语言呢？

James：我在美国麻省理工学院做博士论文时，与 Jack Dennis 教授和他的学生一起在做数据流语言和计算机方面的工作。这些语言都是非常“本质的”并行语言。它们是非常创新的，而且多年以来产生了大量的后续工作。不幸的是，我有一些问题没能解决，而且别人也未能解决。但是，这是一个很有前途的想法，不过从长远来看它最终没有实现。我把那些想法拉到了 UML 里，不过在大多数情况下，数据流架构看起来好像不会取代冯诺依曼体系结构。因此，我做了一下尝试，不过并没有取得什么成功。

还有细胞自动机。我认为我的研究生同学一半以上都是尝试在它们的基础上构建一个高度并行的计算机。那肯定是正确的方式，因为宇宙就是这样构建的。（或者不是。现代物理比小说还要奇怪。最新的推测说空间和时间起源于某些更基本的东西。）但是，细胞自动机看起来只适用于特定的几何问题，当然是很重要的问题，而不是普通情况下的问题。人们还没有搞清楚如何为普通情况编写细胞自动机。或许根本就没有普通情况。

关键问题似乎是控制流与数据结构之间的接口。高度并行控制流无法与相当多的数据相啮合，而且它不清楚如何让数据并行，它仍然在运行我们习惯的那种计算。很可能我们需要运行新类型的计算。大脑是一个并不运行冯诺依曼算法的高度并行的计算机，不过我们完全不知道像大脑那样来组织编程。很可能你无法为它编程：有效的可编程序性和极端并发的概念可能是相互排斥的。或许这是一种更高级别的的海森堡测不准原理（Heisenberg uncertainty）（再次提到了物理背景）。

你所谈论的为并发而改写的编程语言，它看起来跟函数式语言有明显的联系。函数式语言的缺点是什么？为什么它们没有提供易于并发的线索？

James：函数式语言非常适合描述一个非常并行，以至于你并不需要讨论并发的领域。问题在于，在现实世界中，你通常需要中间立场，你需要在其中显式地讨论某种并发性。我猜想这是那些被破坏的对称性的另一面。不过，函数式语言具有很多优点，而且能够在更具命令式风格的语言内选择性地使用它们。

14.9 对称关系

您说过相对于大多数编程语言中使用的更传统的指针方式，关联为状态提供了一个更为对称的观点。您能

详细描述一下吗？

James：在现实世界中，关系就是关系。它们通常不会被封装为只能单向作用。它们偶然也会那样，不过大部分情况是如果 A 和 B 有关，那么 B 就和 A 有关。

它是一个双向的关系吗？

James：好的，这是一种关系。关系本质是双向的。数学上的关系是双向的。甚至说双向意味着一个人正在考虑指针。关系就是关系。世界是关联的，哪一方也没有优先权，或许这要看怎么去说它。双向并不意味着对称，当然了：人咬狗不等于狗咬人。

以关系的角度来考虑你的数据结构和你的系统结构，这是很好的开始方式，而不是从以指针的角度考虑你会如何封装它们开始的。

的确，在代码行层次，你使用的是具体的编程语言。此时你应该考虑你的编程语言提供了什么。

我曾经做了一种内置了关系的编程语言，而且它表现很好。你不需要做出方向性的承诺-——你可以很快地去往任何一个方向。这个成本并不高。我很奇怪很多语言并没有提供它。

这是一种数据流语言吗？

James：实际上，它是一个数据结构包。我称它为 DSM，数据结构管理器。它本质上是一组数据结构过程，不过我把它组装起来了，因此你会很容易从关系的角度来考虑，而且双向都可行。如果你想要从一个关系中删除元组，你只是删除一个元组。我对它进行优化，因此你可以从每一端开始，使用散列法使它实现线性复杂性，而且你并不会对主要性能造成损害。

所有这些技术都非常著名，不过普通人无法当场使用它们来编程。你需要把它们从库中取出来或者内置到语言中。我怀疑大多数程序员现在会知道如何去使用散列法。这不是一种大语言，不过，我们在其中包括了一种方法，即为集合、关系和可扩展数组提供散列法。如果你把数组填满了，它的规模就成倍增加。它以一些内存为代价保持成本的线性增加。再次，我们没有重新编写整个语言，我们有点儿像是在其他东西上铺了一层。它使得很多事情更容易。我们从来没有过缓冲区溢出问题。这种方式的功能可以非常强大。

我很惊讶人们为什么没有做得更多。我知道我的一些以前的同事在 C++中构建了泛型模板类。我不确认他们使用了这么多。或许整个机制在 C++中很难使用，或者仅仅是程序员们太懒，觉得学习它们太麻烦。

我有一个抱怨，在计算领域，编程语言与数据库之间的分歧非常大。你会遇到从事编程语言工作的人们，他们是基于指针的，并热衷于效率等一切，而且，他们使用程序时会有各种各样的问题，那些程序之中有很多讨厌的错误。

你会遇到从事数据库工作的人们，他们是基于关系的。他们懂得关系的概念。那就是我最初获得关系概念的地方。他们并没有那么痴迷于效率。从编程的观点来看，数据库在某种意义上效率是极低的，不过这非常值得，因为他们想要它具有健壮性。他们想要数据是安全可靠的，他们并不想要它突然崩溃（crash）。似乎没有人处在中间。这或许像我们现在的政治制度，你让人们都处在两极，而且我们已经把中间的人排挤掉了。我试图接近数据库人们，而他们似乎没有意识到编程语言里面的东西的价值，反之亦然。

如果这双方有分歧，我确实认为它会帮助我们消除分歧，至少让我们不会有这么大的分歧。如果我们拥有语言，它使得你可以更容易去做跟这些关系数据库有关的事情。你可以更容易去做过程类型的事情，并且很容易在这两种方式之间来回切换，而且大多数时间能以安全的方式使用内置的那些东西来做事情。它会为你提

供计算复杂性效率，普通人不可能拿来就能（out of the box）编程，而且不迷恋于这种自己动手编程（program-it-yourself）的微位推进。我认为人们时常会觉得自己的效率越来越高。事实上，他们的效率不高，因为他们是自夸高效率，只见树木不见森林。他们最终实现了微优化（micro-optimizing），不过它在更高级别上效率更低。

354 **我想 Kernighan 是在《The Practice of Programming（编程实践）》[Addison-Wesley Professional]一书中说过，在过去的 Unix 工具中，他们使用的是一种简单的线性扫描。**

James：确实如此。为什么我们仍然使用 Unix 呢？今年是哪一年，2009 年？那 Unix 又是哪一年发明的呢？Unix 已经发明四十年了，而其中仍然使用了一些同样的假设，比如说字符使用的字节等。你无法在一个字节中对一个字符编码。我们已经超越了那个阶段，而且我们仍然把字符串当成是字节数组来考虑。现在早不是那样了。

当然，他们已经更新了一些，不过 Unix 最初是构建在 PDP-7 的概念之上，使用一个 18 位字的 64KB 内存（而不是小小的 16 位的那些东西），在其中，你会经常交换出整个内存。我不知道还有多少那样的概念潜伏在那里等待惹是生非。

再次，它的相同问题是：一些人试图在做新的操作系统，不过，要让它们获得应用很难。我们在这里使用的是这些旧事情。Windows 使用的代码要回溯到 20 世纪 80 年代。我想它只有 20 年。哇，我已经变老了。你说它只有 20 年，我想这表明我早就是老手了。

如果你回忆一下历史，现在出来的很多东西看起来就不再那么新颖了，因为你看见它已经出现过三四次了。人们都在做相同的重大发现并认为它是新的。我以前已经看见它出现过很多次了。为了引用 Elrond，“我的记忆可以追溯到远古时代（Elder days），而且我已经看见过很多失败以及很多无益的成功”。

我在此提出一种不同的观点。我知道很多人热衷于说这是一种软件危机。在此基础上，很多建模东西已经被接受了。我本人对此感到内疚。另一方面，我可以求助于美国超大型电器连锁商场 Fry's Electronics，并且每年都去购买越来越复杂的设备和软件。每年它都可以降低成本。人们一直在开发新的应用程序。或许根本就没有软件危机。或许为了轰动效应把它夸大了。

你也可以辩论安全性之类的事情。如果人们确定它足够重要，我们将获得安全性。他们说它重要，不过有迹象表明人们并不认为它重要到足以值得投入花销。如果它足够重要，厂商就会注意它。如果人们说不愿意为它付出额外花销的话，它就不重要，这些花销包括现金、循环和内存等诸如此类的事情。

如果安全性是一个重要的考虑事项，放弃使用带有危险指针的 C 或者 C++来编写程序，这样做值得吗？

James：当然值得了。人们会因它们的付出获得回报。比如说数据库——把你的数据弄丢或者数据遭到破坏的数据库肯定不会在业内呆很长时间。开发者在编写数据库管理程序时，他们花了很多精力以确保不丢失数据。机器崩溃是一方面，不过如果它丢失或者破坏了你的数据，那就更糟了。

355 在项目中，人们通常会在发布之前讨论对 bug 的排序。一个会让应用程序崩溃的致命 bug，通常会被认为是最需要修复的。我认为这是一种被误导的方法。有些问题会破坏你的数据，但它不会让你的系统崩溃，它要远远严重于仅会让你系统崩溃的问题，因为你并不意识到你存在一个问题而且你丢失了数据。事实上，一些让人感觉不方便的东西往往可能会比一个致命 bug 更为严重。

我构建了一个早期的建模工具。我做了一个优先级列表。结果表明列表上级别较低的很难对付，而且你不得不

一直疲于应付它们。一般来说，这种小事情在实践中是如此令人讨厌，以至于我把它移到了列表第一位。让你一直感到烦恼不堪的一些事情可能会比可令系统崩溃的 bug 更为严重，因为如果你把系统搞崩溃了，你可以重新启动它，而且它会继续运行。而如果它在你每次点击屏幕时都出来干扰你，你就根本无法使用这个工具了。

就我已经修复的 bug 来说，我所收到的赞扬大多是因为那种 bug。

James：对。你是指真正令你烦恼的那些。我不知道你是否读过 Edward Tufte 写的书。他谈到过“chartjunk”，这是一种 Excel 产生的图表——你画得满篇都是，而提供的信息却很少。他的概念是你要以最少的笔墨来在屏幕上描述信息。

你可以将同样的概念应用于用户接口。我一直认为，最佳的应用程序就是让你点击最少的那个。如果你必须到处点击，如果你有时候必须点击两次，这就是一种糟糕的设计。这些烦人的事情就是阻止你使用某些东西的原因之一。

如果设计者和开发者应该从您的经历中学到的一条经验，那会是什么呢？

James：我的一条经验是一切都在变化。这是我一生当中最重要的经验。一切都在变化。

你需要为变化而构建。你编写应用程序时，需要确信它会在下一个中修改。你教育自己时，需要确信你在大学里学的东西不会是你到退休时还在使用的唯一的东西，而且你一生中可能会经历不同的职业生涯。

工商界的需求也会变化。今天的问题没有必要非得解决，而它们与新的问题相比可能会变得毫无意义。变化无处不在。你需要期待它，拥抱它，并学会适应它，因此你会成功。那些可以适应变化的人们既会拥有成功，也会拥有美好的人生。如果你不能适应变化，你就会困难重重。你是不是在计算领域，或者是否在别的什么领域，这并不重要，因为就现在来说，那是放之四海而皆准的真理。

14.10 UML

您如何定义 UML 呢？

Grady Booch：统一建模语言是一种图形化语言，用于可视化、详细说明、推理、建立文档以及构建软件密集型系统的产品（artifacts）。根据我对 UML 的使用经验，UML 不只是一种编程语言，而像是一种带有深奥语义的语言，它超出了传统的编程语言范围，而且允许人在等于和（或）高于代码的一个抽象级别上工作。

UML 在开发的哪一个阶段最有效？

Grady：UML 适合于软件密集型系统的全生命周期，从产生到消亡再到再生。我的特定偏见是，UML 在系统架构推理方面特别有用。我还偏向于 Kruchten 的 4+1 模型观点，而且在我的著作《Handbook of Software Architecture（软件架构手册）》中，我还没有发现在哪个单个系统中，UML 无法足以让我获得有价值的设计决策。

它与不同的软件工程方法如何互相影响呢？

Grady：我和 While Jim、Ivar 都有特定的方法学观点，我们确实没在 UML 中放入与任何合理的软件工程过程绑定在一起的什么东西。

假设我对 UML 很怀疑。您怎么能说服我它是有用的呢？

Grady：两件事：使用它来为你编写的程序编写文档，并看看它是否有助于在代码之外进行沟通；同样地，在 Web 上搜索它的使用实例，并研究别人是如何使用它的（从 MediaWiki 到嵌入式系统，以及类似二者的很多例子）。

对于使用 UML 来生成实现代码，您的观点是什么？

Grady：UML 设计用于——而且仍然适用于——软件密集型系统的可视化、详细说明、构建以及建立文档。为此，模型驱动的开发已经证明把 UML 当作编程语言来对待是有用的，通过它可以生成可执行程序。同时，我是使用 UML 实现成熟的或者发展中的系统设计可视化（与模型驱动相反）的一个超级粉丝。

我听说医院的房间都只有两张或者四张床，因为在三个人中间通常会变成二对一，而那一个人总是他。UML 设计三人小组出现过这种事吗？

Grady：不过，我记得只有一年左右是两个人（我和 Jim），随后一年是三个人（我和 Jim、Ivar），但在此之后，都是几十人然后是几百人（随着 UML 逐渐变成了一个公共标准）。同样地，参与各方的动力来源也有很大的不同。无论如何，有趣的并不是这么多的妥协本身，而是我们确实是在很多充满热情的人们面前实现了聚合，他们都是出于各种各样的想法而提出这些问题的。正是这种多样性和后来的公开曝光使得 UML 被如此广泛地接受。

357

如果您要将 UML 修订为 UML 3.0，您会如何处理这个过程？

Grady：我可以明确地告诉你，我已经考虑过了。首先，我要从用例集开始，因为我想将它应用于 UML 3.0。在开发 1.x 和 2.0 过程中，我感到痛惜的一件事是：UML 中的很多东西现在是从头做起，而不是靠原型驱动，就像这么说："这里有问题。这里让人头疼。我们如何解决呢？这就是我们要在目前的 UML 中要做的。这就是我们如何改进它"。

开始时，我在行业内就我们要对什么建模进行了大量的调查研究。在此之后，我会努力设计一个 3.0 版本来支持那些用例，并刻意分解元模型使其简化。

您会改变什么？

Grady：UML 仍然需要更简单，但这始终是最难的事情之一，因为为了支持某个具体问题而希望添加元素进来，这样会永无止境。另外，我看到 UML 正向更适合作为系统语言的方向发展。

人们都只有使用 UML 的 20%那部分吗？

Grady：我确实这么说过。在手册中，我进一步把它归纳成了经典的 80/20 定律。我工作要用到的很可能只有大约 20%的 UML。剩下的 80%涵盖了很多边界条件和细节，不过它对于我的目的来说，实际上这就是核心部分。

请记住，人们使用 UML 有很多种不同的方式。如果我使用 UML 驱动来创建代码，那么我就需要它们的很多细节。另一方面，如果我只是使用它来推理一个系统、实现可视化，以及做出心中有数的决定，那么我就并不需要那些细节。这是一种少数派的暴政（tyranny of minority）。我会那么说。因为有一些 UML 应用，特别是来自 DD 的，给 2.0 版带来了很多复杂性，而且让它在完成其他功能方面也变得复杂了。

我见过的大多数是很多专门的白板讨论。这是我的类模型。也是我的实体模型。这是它们之间的关系。

Grady：我开始开发 Booch 方法（译注5）时，我的确从未想过让它变成一种编程语言。如果你沿着那条路走下去，并做出一种可视化编程语言，你就需要创造很多新东西。例如，真正地集中精力做符号和元模型语义。它并不存在，而且我很惊讶从来没有为此付出努力。

如果我有一个盒子，它意味着什么呢？老实说，并没有什么形式规范能说出这个盒子是什么。虽然现在关注 UML 元模型的形式语义，但并没有在符号与 UML 元模型耦合之间做多少工作。

有可能为利用分布式开发和团队协作而做出改变吗？

Grady：我没有考虑过。我已经为可用于分布式系统的多种设计模式进行了分类编目，而且就现有的概念来说，还没有什么东西我无法把它放入 UML 中的。至于说团队协作，我在协作开发环境的问题上花费了很多的时间，包括虚拟世界里的一些特殊技巧，比如说第二人生（Second Life）（译注6）；UML 对此很有帮助，而且它并没有真正地改变语言自身，因为，在临时的和地理上分布式开发中是社会动力学解决了技术性问题。

开发者应该从 UML 的发明、发展和采用中学到什么呢？

Grady：我常说的一句话是，整个软件工程的历史都是以抽象等级的提升为特征的。我们会在我们的工具、我们的方法、我们的语言以及我们的框架当中看到它的身影。在这种背景下，UML 是随着发展自然而然产生出来的。在最早的一代计算中，是硬件平台处于支配地位；而在下一代计算中，则是语言的选择处于支配地位；目前的这一代，软件平台 的问题占据统治地位（主要表现在操作系统之战中，特别是从较广阔的角度来看，并将 Web 看成是一个对等的平台）。这并非是说这些早期的关切一点也不存在。某种程度上，它们仍然在现代系统中扮演着一个非常重要的角色。再回到 UML 上来，它的出现适逢软件系统的复杂性处在这样一种发展轨道上：一个项目的失败或者成功对于语言和编程的问题要求不高，而是更多地依赖于架构和合作。

我说过的另一句话是软件开发过去、现在和未来从根本上来说都是很困难的。正如 Brooks 博士说的那么到位，软件永远都不可能摆脱本质的复杂性。那么我们知道，无论我们的洞察力有多么深邃，所有的未来进步仍然都要依靠待编写的软件（仍然是构建在目前的软之上）。过去对我们来说没什么意义的软件密集型系统，现在它们已经在实践中成为中流砥柱—正像现在对我们来说颇具挑战意义的现代系统，要与未来的系统相比，很可能也会黯然失色。正是这些力量推动我们不断改进软件工程实践和专业。

14.11 语言设计

Language Design

在设计语言和用这种语言设计软件之间有什么样的联系？

Grady：你这个问题是老生常谈了，尽管换了一种新说法：语言学家和认知科学家已经对这个问题考虑了好

译注5：Booch 方法（Booch Method）是 Grady 开发的面向对象的分析设计方法，他还开发了可重用的、灵活的 Booch 组件。

译注6：第二人生（Second Life）是总部位于旧金山的 Linden Lab 提供的 3D 虚拟世界，它完全是由用户拥有并实时创造，用户可以在其中经历一次与现实世界完全不同的人生体验，在虚拟世界中得到了一次重生的机会，完成一直梦想的生活或者理想。

几十年了，他们就所谓的萨丕尔-沃夫假说（译注7）辩论不休。Edward Tufte 也类似地指出过：正确的表示法可以化解复杂性，使得它可能以一个抽象的方式来颇有意义地推理复杂的信息。

359 进一步解释一下，萨丕尔-沃夫假说（来自语言学家 Edward Sapir 和 Benjamin Whorf）假设在语言和想法之间有一个连接：口语的句法和语义元素会影响一个人感知和推理世界的方式（而且是反之亦然）。现代语言学家，比如说 George Lakoff（《Women, Fire, and Dangerous Things》[University of Chicago Press]的作者）就对此表示赞同。Tufte 的著作《The Visual Display of Quantitative Information》（Graphics Press）聚焦于复杂数据的可视化，而且通过大量的例子表明有效的图形可以使易懂和晦涩二者泾渭分明。

你要问，我是怎么知道这些问题的？好的，构建简洁的抽象是面向对象开发的一个根本原理，抽象主要是一个分类的问题，而且像 Chomsky 和 Lakoff 这样的语言学家都对我考虑分类产生过影响。

总之，要把你的问题改为“在语言和软件设计之间有联系吗？”——如果我们这里所说的语言是指典型的文本编程语言（比如说 Java）和图形化编程语言（比如说 UML）——我会回答“很可能有联系”。

鼓励算法分解（比如 FORTRAN 和 C）的语言将会导致特定的程序组织风格与鼓励面向对象分解（比如 Java）的语言甚至是面向函数式语言截然不同。从我的经验来看，很多其他因素会在更大程度上影响设计：开发团队的文化、历史背景，以及在任何给定的时刻系统所承受的特定压力等。有人可能会主张语言是一种“第一推动力（First Cause）”，不过我不这样看，我认为是其他力量在起作用。

开发编程语言和开发“普通”软件项目之间有什么区别？

Grady：它很可能与“起草公共政策方面的法律和让这些法律明白无误”之间的区别相关，不过仍然存在着很大的区别。无论是人类语言还是编程语言，都没有无限的自由度，相反地，它们会受到技术、商业、社会、历史以及实践等各种力量的约束。特别是对于编程语言来说，它在句法和语义上必须是精确的，因为我们最终要使用这些语言来创建实用产品。我不知道“普通”软件项目是不是这样，不过其中肯定存在很多二义性。一种语言一旦定义之后，就会保持相对稳定；一个软件项目，如果它本质上是有趣的，就会处于不断前进的环境之中。因此，当语言和项目本质上都是要解决的工程问题，这个项目的推动力就会更为多样化并且更具活力。

360 **要从实现一种语言核心的某种程度的形式规范开始并在此基础上发展，您认为这个重要吗？**

Grady：绝对重要。事实上，在我们开始尝试真正地解决 UML 前身的语义并将 OMT 和 Booch 方法统一的时候，我和 Jim 就是这么开始的。我们是从使用 UML 自身来编写元模型开始的。而且挑战在于一个人的“形式”在另一个人看来就是“非形式”。对于我们来说，问题是“对于我们要做的事情来说，它足够形式化吗？”而且答案是“是的”。

您是如何判别“足够”的呢？

Grady：有一个首席法官的故事经常被人提起，他曾经被问过一个问题：你如何判断什么是色情的呢。他的回答是——你很可能已经听说过这个故事——“我一看就知道”。就是这样。当你开始谈论含义的含义时，你就陷入非常形而上学的讨论当中，因为什么才是真正的形式化并不清楚。因为你把自己带入了深沟，还在

译注7：萨丕尔-沃夫假说(Sapir–Whorf hypothesis)是一个关于人类语言的假说，由语言学家兼人类学家萨丕尔(Edward Sapir）及其学生沃夫（Benjamin Whorf）所提出，这项学说认为，人类的思考模式受到其使用语言的影响，因而对同一事物可能会有不同的看法。

说："含义是什么含义呢？"即使是在形式体系中也不得不到此为止了。

很多人可能会说，"我们可以依靠图灵完备性或者 Lambda 演算方法，而且除此之外，我还知道如何在参数上使用函数"。

Grady：我非常高兴世界上有那么多的人关心那些事，不过设想一下，如果我们对 Java 或者 Vista 同样如此严格，实际上有多少东西是用那些没有形式体系的语言和平台编写的呢。还有操作语义。我们运行这些东西，并且知道它们是如何工作的。这对于我们现在开发的大部分软件来说已经相当不错了。这并不是说不需要形式体系。需要！而且我要非常谨慎地使用这些词——肯定是非常窄的行业角落非常需要深奥的形式体系。

把可以用来构建系统的语言和可以得到某种程度的形式体系的语言区分开来，这么做值得吗？

Grady：当然值得。但是 Linux 有形式语义吗？ C++有形式语义吗？当然有。我猜想人们已经在这上面做了一些工作，但它并没有阻止人们干实事。

那它也有很实用的关切。

Grady：绝对如此。我是一名实用主义者。如果它能工作，那它就是好的，我就会使用它，这就是我使用了大约 20%的 UML 的原因。对我来说这已经足够了。

您的 20%可能跟我的 20%不一样。

Grady：我想那很可能有相当程度的重叠。或许你的 19% 和我的 19%能匹配上。

老实说，这是一个相当不错的匹配。

Grady：是的。的确如此。

为了那 19%，您会从哪儿开始寻找用例呢？

Grady：我会研究一下人们使用 UML 的方式。我会找到尝试使用 UML 的实际项目，并问它们："给我提供一些普通的用例。给我提供一些中心用例，而且让我们确信易于使用 UML 来做简单的事情。然后我们再开始考虑那些边界条件用例"。我会从真正的使用开始，而不是从预期的使用开始。

该到整合与简化的时候了。这种语言到了第 3 版，而且有这么多地方在用，它不仅有创新价值，而且还有分解价值。我说这话有点刺耳：在某种程度上，UML 2.0 受到了第二系统效应的一些拖累，其中有很大的机会因素，也有特殊利益集团牵涉其中（如果你愿意这么说的话）叫嚷着要把特定的特性添加到臃肿的 UML 2.0 当中去。现在该后退、分解和简化了。在任何有意义的系统中，你确实会看到这种增长和崩溃。现在该简化了。

是偶、奇、偶、奇模式吗？

Grady：我边想边自言自语是 Microsoft 操作系统版本。当我看到 Windows 7 时，是的，可能就达到那种效果了。

或者 Star Trek 电影特效。

Grady：现在这个肯定值得进行很多的形式化研究。

在多大程度上 UML 2.0 的向下兼容性可以成为一种考虑因素？

Grady：要记住在 UML 2.0 中，如果我们能不断前进和完成的话，如你所言，有很多东西能够完全向下兼容，它的规模要持续增长。很可能只会在极少的情况下我愿意中断它，并说“哇，这个东西增加了很多复杂性。那它就是不值得的。对不起，我要冒犯一些人了。这是一些权宜之计”。

如果你来看看使用那 20%的 UML 的核心内容，我们不会中断那些东西。让它保持向下兼容性—当然，那是非常重要的。

几乎每一个编程语言设计者都会试图那么做，而且有很多不同的方式。Lisp 几乎是在说：“我们不会为你提供对象系统；你可以构建一个。我们不会为你提供特定的控制结构；你可以构建它们”。然后每个人都去这么做了，而且他们开始共享代码。然后，Common Lisp 设法确定每个人应该使用什么。

Grady：这是一个用来描述它的优秀模型。

在某种程度上，这也是一种演进或者一种模拟退火算法，其中，公众搞清楚是什么看起来不错，同时设计者再把它放回核心之中。

Grady：绝对如此。这就是为什么我特意从行业自身寻找用例的原因。人们实际上如何使用这个呢？从企业这方面来看看它吧，人们在企业里考虑过一些真正的大型系统。让我们再来看一看一些小系统。

通过研究这些，您对标准化过程怎么看？

Grady：我曾经参与了很多的标准化过程，它是一个精彩的、有趣的过程，我想众多社会学家接下来会非常高兴了。就标准过程而言，它很难概括。有一些标准，而我不想在此点名，是由具体的行业推动的，它们成立了具体的公司，最后变成了事实上的标准。还有另外一些标准确实是集体的功劳。还有一些显然是由政治推动并强迫他人接受的。因此，标准形成的过程各种各样。

真正令人高兴的是，不管人们如何抱怨标准制度，它确实具有它的价值，而且在发挥作用。因此，我非常感激 OMG 等组织，它们愿意投入资源来带领大家做这些事，并为它们的发展创建一个论坛。如果没有这样的标准，Web 就不复存在。

这是一个痛苦的过程，很多人参与其中，他们满怀热情，他们对这个领域有着自己特定的观点，他们知道有关领域的这个观点是对的；这就是人类经验的本质。

每个人离开时都稍微有一点失望，但感觉其他人也都稍微有一点失望，而标准竟然就这样搞出来了。您的经验是这样吗？

Grady：我不确定它肯定是这么糟糕，不过标准一直都是一定程度的折衷妥协。你能想到的每一个标准都是这样。总会有人对它失望。事实上，仍然有人对美国总统奥巴马当选感到失望，他们属意布什执政。不过想想看吧。很多人喜欢美国当红歌手小甜甜布兰妮斯皮尔斯（Britney Spears）的音乐——人各有好。再说，还有人类经验不同，让我们保留一点多样性吧。

在 Linux 内核方面，除了 POSIX（可移植操作系统接口）之外，还没有真正的标准。这是 Linux 之父 Linus Torvalds 和他的副手们的嗜好。

Grady：噢，我认为这个类比很不错。有一段时间 Linus 作为领导者处于最前沿。就 UML 而言，我和 Jim Rumbaugh 远离了标准过程，并对它放手不管。它没有那么大的推动力，而且我认为还是有一些区别的。来看一看 C++；Bjarne 在 C++的发展仍然起到很重要的作用。坚定的支持确实有助于一定程度的理性的完整性和一致性。

他们也正在完成新版本的标准化过程。

Grady：对。但是，像 Linux 的情况，你会听到有人说他具有很丰富的经验，而且说这话的人有着很大的期望，这个人已经证明了自己。虽然他们不是唯一的声音，他们是一支强大而又清晰的声音。

363

也许这是两个问题。一种想法追求标准化的价值？你在多大程度上需要一个具有很强洞察力的强势领导，来帮助您取得成功？

Grady：关于后者，你可以向所有做出过努力的人来提问。看看这个世界上发生的变化。我没有将我们的努力与这些人进行比较，但看一看印度圣雄 Gandhi（甘地）做了些什么，或美国黑人民权领袖 Martin Luther King（马丁·路德·金）做了些什么。在这个世界上，一个人的力量带来的变化是非常巨大的。

从技术的角度来看，你可以想一想做出过贡献的技术领袖。看看 Google 的 Larry 和 Serge，他们从美国斯坦福大学带来的想法，现在变成了一个完整的帝国。一个强大的远见卓识的出现，已经在许多领域证明了自身与众不同、有所作为。

我不想把这些作为竞争的观点提出来。

Grady：这只是一个例子，证明一个人或一小部分人的能量可以为很多地方带来根本性的变化。

假如我在创建一种编程语言。我应该期望从追求标准化中得到什么样的价值呢？

Grady：在创建这样的新东西时，最终是由市场来决定它是否能够成为标准。很多脚本语言不是通过标准组织开始的，而是发源于草根阶层，因为人们认识到里面有很多关键的东西："哦，我们需要更精确地将这些东西标准化，因为我们需要互操作性"。这里的关键是，你必须考虑：在这些事情中间，你能提供什么价值，而且市场对此有何反响。这些标准的过程可以帮助您找到使它成真的背后的一些关键的东西，但是这最终要由市场来决定。

所有新语言的挑战都是，我可能会想出最完美的技术设计，但是还有无数的其他的东西，主要是社会方面的，会影响该语言是否成功。我能创建一个实践社区吗？它会解除这个特定社区当时切实存在的痛楚吗？在它处于远未成熟的状态下，组织中的个人是否有足够的兴趣来促进它？它还没有明显的商业模式，但它似乎是正确的。

很多的这类早期发展始于信仰，如果你愿意这样说的话，它们是在正确的时间处于正确的位置上。毫无疑问，Booch 方法就是这样的，也许如果你去问 Jim Rumbaugh，Objectory 和 OMT 肯定也是与此如出一辙，而 UML 正是由它们发展而来的。我们是在正确的时间正确的位置解除了市场的一个痛楚。

现在，这个世界需要另一种语言吗？你可能会想到 M，Microsoft 给市场带来了什么呢？M 会成功吗？这在技术上很有意思，但是否成功要取决于是否在市场上具有吸引力。即使它可能被标准化，即使它可能会因为 Microsoft 的行动而称为一种事实上的标准，最终还是要由市场来决定。

364

除了远见卓识之外，它似乎还有很多务实的东西。

Grady：噢，绝对是这样的。我相信这是真的，如果无法为实际项目解决实际问题，即使是拥有最好的理念、最可靠的技术、完全的形式化、周到的考虑、完善的文档，也终将一败涂地。

下面这些话人们可能不喜欢听：它似乎要成为一个真正的技术乌托邦，它仍然相信技术最好的项目肯定会成功。

Grady：我非常高兴，有些人因为他们的乐观主义而继续存在。他们的乐观主义鼓舞着我们。但是我更是一个解决问题的工程师，我处理的是实际问题。我不是一个计算机科学家，我更是一个工程师。这是令人愉快的，我们都具有领域的观点（world-views），因为这种跳跃，这种紧张，使得双方越来越坦诚。我被我的务实所推动，但在另一方面，纯粹的计算机科学家促使我变得纯粹和更加形式化，这不是坏事。同样，我也促使他采取务实的态度。

在两极之间有没有一种创造性的紧张关系？

Grady：当然。我认为应该有。在我心中，创造性问题的一部分就在于紧张关系的存在，因为紧张关系会将我们的创造力聚焦于解决实际问题。有一个计人喜欢的网站名为 Gaping Void（译注8）。它的创建者基本上是一个公关型的人物，他的声名鹊起源于他在商务名片背面进行的艺术创作。但他的这种已经非常流行的即兴创作是关于如何实现创造性的。你可以把它推荐给读者（注 2）。这非常有趣，因为它实际上表明的是一种务实的紧张关系的重要性。

14.12 培训开发者

Training Developers

为什么我们改进编程方法和过程如此缓慢呢？

Grady：哦，我不赞成你的问题的前提。或许它看起来慢，是因为你是从行业内部来看的，我们知道可以做得更多，并且做得更好。不过，考虑到我们的行业确实从根本上改变了世界整整一代人的生活。对我来说，那是很快的，而不是很慢。

我们如何在软件领域传播经验呢？

Grady：在中世纪，同业公会担当了传播部落记忆的主要任务；今天，我们在软件方面缺乏这种实习制度。不过，大量的经验是通过网站（例如 Slashdot（译注9））、图书、博客以及技术性会议来传播的。这也是这种情况下，原始的、可运行的、裸源代码成了来自过去的一个知识来源，这是我曾在计算机历史博物馆（Computer History Museum）工作为后代保存经典的软件代码的原因之一。

365

译注8：Gaping Void，网址是 www.gapingvoid.com，由 HughMacLeod 创建，拥有大量粉丝。HughMacLeod 在收到的名片背后乱画种种尖酸刻薄甚至有点超现实的卡通，并把这些画贴到网上。

注 2：*www.gapingvoid.com*

译注9：Slashdot 网站（slashdot.org）创办于 1997 年，是一个著名的科技新闻网站。它的稿件都是由读者投稿，编辑审核后发表。

现在的学生应该更多地学习什么呢？

Grady：我会从两个方面回答这个问题。从软件的视角来看，所有优秀的研究课程都会教给你编程和设计的基本技能。不过，我建议三件事：学会如何抽象，学会如何参与团队工作，并研究别人的代码。从更广阔的视角来看，我鼓励学生尽情追随自己的爱好，但永远不要忘记成长为一个完整的人的价值。

在您的领域研究之外（世界远比软件更丰富），发展不断学习的能力（因为这个领域在不断变化），并且强化你的好奇心和冒险精神（因为这些是创新之源）。

根据您的经验，您能给刚入门的程序员提供的一条最好的建议是什么？

Grady：这个问题正好是我在美国南加州大学（USC）时学生提的问题。几个星期前，我在加州州立理工大学（Cal-Poly）南加州大学了做了一些讲座。有一些问题是我从这些学生那里得到的后续问题，所以我会给出当时回答他们一样的答案。

首先是遵从你的爱好，并确保你从中获得乐趣。在追求事业和追寻生活的过程肯定会有价值，但最终的发展和我们做的所有事情也有价值，这是人类的经验，而且你在这个过程中会成为一个完整的人。因此，充分地享受生活，丰富人生阅历，所以请您这样做。遵从你的爱好，因为这样很容易发现确实没有意义的工作，你会在其中发现你讨厌什么东西。请不要去做。这就是我的忠告。

其次，我鼓励你要获得一些经验。让自己参与一些开源项目。找到一些你感兴趣的东西，并开始去做。不要害怕尝试新事物，让自己的接触多样化，这样你就可以接触到不同领域中的新思路。坦率地说，无论你从事哪个领域，这都会对你很有帮助。

音乐的灵感会不断地涌现。艺术的创作思想，尤其是写作，也是如此。

Grady：说到写作，我经常问学者们一个问题："你们有多少人学过软件课程？"只有两个人说学过。如果你是英语文学专业，你应该读过大师的作品。如果你想成为一名民用空间（civil space）建筑师，那么你就得仔细研究 Vitruvius（维特鲁威）、Frank Lloyd Wright（弗兰克·劳埃德·赖特）、Christopher Rennin（克里斯托弗瑞宁）和 Frank Gehry（弗兰克盖里）等这些世界著名建筑师和其他人的建筑作品。而我们在软件上却不是这样做的，我们没有研究过大师的作品。我鼓励人们看看其他人的作品并向其学习。

如果我们有一种文献资料体系模式，能够告诉你："漂亮的 Pascal 程序应该是这个样子"，这种模式将会非常好。

Grady：这就是 Andy Oram（安迪·奥拉姆）和 Greg Wilson（格雷格·威尔逊）的《Beautiful Code（代码之美）》[O'Reilly]尝试做的事情。新西兰的一个人出了一本书，里面给出的读书清单也很好。他是我遇到过的实际这样做的两个人之一。

我们没有软件文学批评的知识体系。我认为，Knuth（高德纳）在文学编程所做的工作是在此方面的早期尝试之一。大多数代码实际上写得很不好。如果你想表述的话，它就像三年级的句子结构水平等。但是，漂亮的代码美丽、优雅，充满了戏剧效果，它的写得很好。我有机会读过 Mac Paint 的源代码，大约有 10 000 行 Object Pascal 代码，它写得非常漂亮。在公开的代码中，我们还没有发现这样的例子。

接下来面临的挑战是，如果我有一个系统，比如说总共有 10 万行代码的 Linux 内核，我不是不想去读它的。这就像是一遍又一遍地反复看《战争与和平（War and Peace）》。我怎么样才能展现它的美丽和优雅呢？今年

1 月份有一个由 Peter Denning（彼得丹宁）和 Alan Kay（艾伦凯）组织的 Rebooting Computing 峰会，还有其他人参加，我也会去参会。我们讨论的就是这些问题之一。怎样才能找到那些非常隐蔽复杂的智慧，并把它们的美丽展现给世界呢？

我在计算机历史博物馆就遇到了一个类似的问题。我是该馆的理事会成员。我们已经创建了一个软件收集委员会（Software Collections Committee），拥有所有精彩的作品。你如何展现 Vista 当中的美呢？你如何让人们读到里面有什么呢？在民用建筑领域，你可以向他们展示建筑物。你可以向他们展示一幅油画。他们可以听一首乐曲。我们如何展现软件当中的音乐呢？

或许我们需要文档算法和数据结构。

Grady：我认为这个层次太低了（太底层了）。我的观点是你需要将遍布系统本身的模式用文档方式记录下来。

14.13 创新、改进和模式

Creativity, Refinement, and Patterns

我们如何解决旧软件的问题？

Grady：我常说的一句话是，虽然该代码是真的，但它不是全部：从执行的视角来看，它有一个信息损失。我的经验是，对于旧软件你有 9 件事可干：放弃、分发、忽略、维持、重写、获取、包装、改造或保护它。这些事情都有技术和社会因素。从技术角度看，还有一个正在进行的有趣研究，从代码中去获取模式；从社会角度来看，口头历史技术能够有助于问题的解决。

367

您从哪里找到了设计灵感呢？

Grady：我是在复杂的事物的优雅中找到了灵感的。这些东西可能是软件（手册（Handbook）的目标之一就是能系统编纂整理建筑模式，并解释它们的美）；有些是有机系统（实际上，任何有机系统都已经演进了数百万年，它显然有一些东西值得我们学习）；艺术（有很多艺术家通过各种媒介展现了伟大的美）；音乐、生产工程、量子物理……对我来说，这个清单简直是举不胜举。

您认为编程在什么程度上需要创新性？

Grady：很多编程并不需要很多的创新性，因为你在解决已知的问题，而您试图以有趣的方式来完成它。这跟有人把房子延伸到我家里来并不一样。他们肯定被限制在一定的环境，而且必须遵循特定的最佳实践。我并不完全是要聘请马上就能上手的人，而是想聘请能够遵循最佳实践的人。他需要一直创新。这就像是说："哦，哇，这并不需要完全一致，所以我需要想出一些创新的方法来做这个事"，这就是小的个体开发者的创新来源。

我们在软件开发方面做的很多工作并不需要那么多的创新性。我们知道构建系统的技术基础。因此，我们必须以全新的方式来应用它。必须塑造它，磨光它的棱角，并四处插入新东西。做这种难题的部分乐趣就在于此。这一点毫无疑问。

当然，还有很多地方需要大胆创新。我们不懂得如何正确地构建大型的全球规模的搜索系统，Serge 和 Larry 做了，他们做出了原型，Google 就此诞生了。我们不懂得如何正确地查看所有的数据，它们来自数以万计（如

果不是数百万的话）的摄像机，这些摄像机遍布伦敦、纽约和北京等世界各个角落。要这样做的正确的架构是什么呢？我们并没有很好的模式，因此，肆无忌惮的、大胆的创新很有必要。

但是，一旦你开始收缩到合适的架构上，那就会将问题空间限定下来，并开始小幅度的创新。

您再次使用了“创新”和“限定”这个词。对于语言演变来说，这看起来像是两条直接的平行线。

Grady：当然。一个画家、作家，我本人、或者是创新领域中的任何人，最讨厌的事情就是完全是一张白纸，因为它没有任何限制。没有什么可供参考。这时候你就开始给它添加限制，在某种程度上这是一个奇怪的事情。你就变得无拘无束，因为你现在就可以在这些限制因素内开始工作，并运用你所有的创新技巧来解决这些限制。

如果我是一个音乐家，我会说：“哇，我可以随心所欲”。那么，不受约束就好像是：“天哪，我应该弹钢琴，还是演奏别的什么呢？我应该做什么？”

它实际上是一种富人的烦恼，焦点并没有集中在你身上。我可以说，“天哪，我可以制造自己的钢琴”。或者像《The Art of Computer Programming（计算机程序设计艺术）》的作者、一代编程宗师 Don Knuth 所说：“我写了这本书，我不喜欢它的排版方式，所以我要停下来几年，写一种能够实际排版图书的语言文本（language text）”。我们不能对 Don 妄加指责，因为他在这方面做了许多很棒的工作，但在面对没有任何限制时，确实很难将自己的技能集中在某一方面。

您认为我们在尝试建立新系统时应该首先确定的限制因素，然后再将它奉为信条吗？

Grady：我不认为你总是可以先做到这一点。你生活在那种环境中，你可以选择一些限制因素；我在做每一个决定时，接着就决定不做别的事，这并不是一件坏事。因此，在这些项目当中，一个人作出的最早的决定是通过一种信仰的飞跃来建立一些限制因素。你无法事先决定这些限制都是些什么东西，你必须想办法尝试新东西，然后再把它扔掉，最后再遵循它。举一个很简单的例子，我可以说：“哎呀，我要设计一种新的图形化编程语言”。因此我就开始了。嗯，我突然会做出这样的决定：“哇，这也许像一个 2D 语言，而不是 3D 语言”。也许我作出决定，颜色对我来说很重要，而且我突然意识到：“哇，我刚才忽视了整个色盲程序员群体”。每一个这样的事情都变成了我要解决的限制，我必须处理这些限制的后果。

这就证明迭代过程很有必要。

Grady：当然。生命的一切都是迭代。这又回到了我刚才说的那一点上，你先前所知不多，甚至不足以提出正确的问题。当然，在面对不完全信息时，必须借助于信仰的飞跃并籍此前进。

在未来 10 多年里，我们是否有可能看到爆发出一种可视化编程语言或系统？

Grady：哦，已经有了。这就是美国国家仪器有限公司（National Instruments，NI）的 Lab View。这绝对是我见过的最酷的可视化编程语言。我不知道你是否熟悉。这真是太神奇了。你可以画代表虚拟仪器和虚拟电子设备的方框和线条。我认为，如果需要的话，你可以深入钻研在其中使用 C 和 C ++。基本上，你可以构建所有的虚拟仪器，并与现实世界的设备相连接，它实在是太酷了。已经有了这样的东西。

什么类型的仪器？

Grady：示波器、监测仪器等。可以想象一下，您的个人计算机上的 USB 端口上有一些数/模转换器和并行串

行转换器之类的东西，因此我可以与现实世界的设备对话。现在，我可以简单地把那些东西画在一起，我就可以在我的计算机上构建虚拟仪器。也许我想要一个带状记录纸，或者是示波器或做一些数据转换处理（munging），并在各种仪表上以有趣的方式来展示。它实在是太酷了。

369 请你看一看 Charles Simonyi（查尔斯西蒙尼）尝试领导 Intentional 的进一步的发展方向。Charles 的想法是，在他的实例中聚合一些原型：如果我是一名电气工程师，那我会使用图表，并且把它加到我的系统当中。我还没有见过他的其他东西，不过这就是一个经典的例子。我选择与特定领域密切关联的可视化。

您的标准业务应用程序是怎么样的？您有一个后台数据库。你有业务对象。您有一个表示层？

Grady：大多数的企业系统构架很乏味，因为它们在这方面比较简单。它们是一个规定。你必须做出一系列设计决策，而且我们也知道这些决策是什么。它之所以如此困难，是因为存在着巨大的技术流失，它是富人的苦恼。我还可以使用很多方法来继续构建我的东西。

我的意思是，如果我今天说："哎呀，我要从头开始建立银行体系"。那么，可能我首先担心的是一组标准，但是我可以从数量有限的方式中选择出一种来解决这个问题，不过这些方式也实在是太多了。我想很多的挑战和混乱正是由此而来。考虑当前业务应用最不稳定的这一块。它就是表示层。我如何向人们传递这个信息呢？10 余年来，人们都满足于仅仅做这些纸印刷品，但现在我们已经通过网络和移动设备有了这些精彩的开放的方法，你可以看到这个领域很多创新正在"流失"，因为我们并没有会聚在正确的融合模型中，在其中人们希望与这些系统一起合作。这就是你会看到流失的原因。

另一块最不稳定的是业务规则领域。因为我们过去受机器所限，当时的挑战原因在于我们将业务规则扔在一边，将过程存储起来，并把它放到浏览器中，但它们到处都是。我们意识到这有一点问题，当不稳定的业务规则的变化非常快时，因为它使我们难以对改变做出反应。

你会看到许多企业系统的分解，业务规则就是从其中抽取出来的。我们找到了企业系统查看这个业务规则库并对此做出反应的方法。不过同样，我们无法知道……我们建立了第一个系统，我们无法知道哪里最容易发生变化，而且，在那个空间里现在要发生哪些变化。

它对可视化是开放的吗？当然。我们看到能够找到新语言来表示业务规则的机会。我个人的意见是 UML 已经足够了，我们看到发生了一些有趣的政治事件，导致了 BPEL（Business Process Execution Language，业务流程执行语言）的出现，它已经与 UML 有很大的不同了。

Microsoft 可能会认为 M 能做到这一点。

Grady：当然，这又回到了刚才我提到的那一点上：M 会成功吗？这要取决于市场。

370 **人们可能还没有意识到他们需要一个单独的业务规则层。**

Grady：当然。IBM 的研究人员撰写了一篇题为"The Diary of a Datum（数据日记）"的可爱的论文（注 3）。我们不断地增加这些层，它也许会在一定程度上帮助我们对这些系统进行人类推理和可视化，但在我们的系统可执行力方面，我们最终加入了不可思议的层。他们在这篇论文中指出，我们查看这些简单地数据段，该数据转换、流入、流出、缓存，直到实际完成某些功能，这些方法的数量非常惊人。我们向系统中添加抽象层需要成本。

注 3：Mitchell 和 Nick 等。"The Diary of a Datum: Modeling Runtime Complexity in Framework-Based Applications（数据日记：基于框架的应用系统运行时复杂性建模）"，IBM Research（2007）。

在我看来，看起来软件的平均复杂性和规模逐年增长。面向对象编程对此有帮助吗？

Grady：如果你说的“面向对象”是指特定类别的语言，人们可以说语言的表现力的确比其他的要丰富，而且它本身代表着更高层次的抽象（因而表示时需要更少的代码）。如果你说的“面向对象”仅仅是指哲学意义上的分解（与算法或者函数式抽象相比），那么你就会回到了 Aristotle（亚里士多德）的一个文本范畴（Categories）问题上了：对于表达复杂的东西来说，多种形式的分解确有必要。

我观察代码的复杂性并不与代码的规模/行同构。还有另外一种复杂，也许用我所谓的语义密度（要表达的意义和事物之间的语义关系度量之间的比率）来度量最好。语义密度是在增大吗？是的，我想是这样，但我认为这是与我们用来表达这种语义的表现手段正交的（无论面向对象与否）。

现在，软件并发是一个很大的话题。

Grady：这个问题已经存在很长时间了。在单处理器上的模拟并发（多任务）是一个很古老的概念，当世界上只有一台计算机时，人们就开始思考如何使不同任务同时工作。今天，我们有计算之岛，但是更多的时候，人们面对的都是松散耦合的分布式并发（例如网络）或者紧密的并发（大规模并行计算机使用了多核技术）。总之，这一直是一个“很大的话题”，坦率地说，这是一个很困难的问题。普通的开发者不知道如何去构建分布式、并发以及安全的系统，因为这些属性需要系统性的解决方案。

并发的发展如何？请问我们能否通过增加人力来缩短开发时间？

Grady：同样，我怀疑你的问题的前提。

“缩短开发时间”可能会有好处，但其他可行的是“提高质量”、“增加功能”以及“降低复杂性”等，这些都是增加人力所希望产生的结果。无可否认，增加人力增大了可用的劳动周期，但同时也增加了噪音、通信开销以及项目内存的成本。现实情况是，大多数经济性最好的软件密集型系统都只需要很少的人，而且真正有趣的是需要数百人，如果没有更多的利益攸关者的话。因此，在某种意义上说，你问了一个很无聊的问题。☺

在软件开发过程中，什么会限制合作的成效？

Grady：我的经验表明，开发者日常生活中有一些摩擦点会单独地或者共同地对团队效率产生影响：

- 启动和正在进行的工作区组织的成本
- 低效率的工作成果合作
- 维持有效的小组沟通，包括知识、经验、项目状态和项目档案
- 跨多个任务的时间等待
- 利益相关者协商
- 无法正常运行的东西

我把这些都写在了 CDE 论文中，你可以访问：*http://www.booch.com/architecture/blog.jsp?part=Papers*。

你如何确定系统中的简单性呢？

Grady：Dave Parness 问过我这个问题。最近，我为我的 IEEE 软件专栏写了一篇文章，它是关于处理架构复杂性的，我开始说：“看看岩巨石。它非常大，但很简单。再看看 DNA 链。它非常小，但很复杂”。Dave 写

信告诉我说，他要挑战这种说法，我回复他，引用了 Herbert Simon 在《The Sciences of the Artificial（人工科学）》[MIT Press]里面的一句话："捍卫你为什么认为的想法"。Simon 指出，如果你考察一下我们能理解的复杂系统，它们往往有很多共同的特征，往往是多层次的。它们以这样或者那样的方式实现了层次化。因此它们内部可能存在着大量的重复。

Simon 在他的那个时代看到了结构上的重复。我要更新他的观点，这不仅仅是结构上的重复，这也是我们在系统内看到的设计模式的重现。我曾经有机会为你能想象到的几乎每个领域的很多项目工作过，我已经看到一些真正的"粘粘乎乎的"系统。我也看到过一些漂亮的系统。那些美丽而简单的系统往往都具有那些共同特征，它们是通过一组超出该系统范围的设计模式来实现的，这些设计模式贯穿了许多单独的组件，并给它们提供了极大的简单性。

事实上，如果你看一看研究 DNA 的人们，他们试图去寻找这些共同的东西。开始我们认为 DNA 发现的所有一切都是垃圾编码，而且认为："哦，这是来自演化时代的废物，它毫无意义"。而当我们进一步深入时，我们才认识到："哇，这种废弃物不一定是无用的垃圾"。

我们之所以把它称为垃圾，是因为我们不懂得它的价值。我认为这是"填补空白的上帝"问题（译注10），但我不想讨论这形而上学的话题。正如人们在这个空间内开始解压缩这些系统美丽的，激烈的高雅，还有一些正在出现的惊人的模式。这就是我觉得的简单和优雅。

您所说的"超出该系统范围的设计模式"指的是什么？

Grady：我会提到一个华尔街系统（Wall Street system）。我不会提起它的名字。不久前，我和一些家伙在做考古式的挖掘，我们试图去挖掘出他们的一些决策。这是一个庞大的系统。它有数百万行代码，使用了你能想到的所有语言：从汇编语言，一直到今天的现代语言。如果你从系统的总体架构来看，它有一些非常普遍的指导原则。这个系统存在一些安全问题。你确实不希望人们入系统，也不希望有人知道系统正在做这些，它要抹去一个交易的十万分之一。如果你每天做的是 1 万亿美元的交易额，没有人会去注意它。

如何防止这种事呢？非常简单的业务规则！所有的状态变化的事物，都必须处在数据库中存储的程序中，这样一来就没有办法做了，因为这是通过走查和大量形式化方法核查过的，如果不这么做，你没有办法注入使用状态改变方式的代码。这是一个非常简洁而又美观的原则，超越了那里所有的代码。事情就是这样的。

设计模式最终是以一种和谐的方式在一起工作的类的集合体。仅凭查看单行代码，你根本看不到那些东西。顺便说一句，这就是架构挖掘面临的艰巨任务，因为代码并不是全部的真相，也不能找到那些东西并发现那些模式。它通常被锁定在个人的头脑里。

这几乎意味着，你无法总是去查看系统，理解它的简单性或者是弄明白它的基本结构，直到你搞清楚这些限制、规则或者设计决策为止。

Grady：当然。我们也必须要通过推理来学习。你查看着这一大堆事情，模式就开始显现出来了。你不能仅看一个实例就说："哇，就是这种模式"，因为这不是模式自身的本性。这是该领域之一，我更倾向于使用 UML 的可视化作为构建系统架构的很多东西。毫无疑问，有时候是"绿场"开发，但总的来说，世界上大部分的软件，用 Chris Winter 的话来说是"棕场"。这听起来非常恶心。但在某种程度上，我们构建的系统大

译注10：旧的上帝观把上帝看作是填补空白的上帝（God of the Gaps）。意思是说，我们之所以有上帝的观念是因为在用科学理论解释不了的地方，要用上帝来解释。上帝存在于科学无能为力的地方，存在于科学不能涵盖的间隙之中，也就是说上帝要为人类知识的不完整性填补空白（stop-gap）。

部分是对现有系统的改造或者是在其基础上添加新东西。

也许我们应该接受“棕场开发”这个术语（译注11）。

Grady：我认为这是一件好事，根本不是一件坏事。对我来说，UML 开始发挥作用，因为它使我看到了仅仅通过代码本身看不到的东西。

这是一个抽象层次，在此之上，这些模式变得显而易见。我可以想象，如果你为现有的生产系统绘制图表，或者是生成 UML 产品，当你将这些概念统一起来，你会有一种非常满足的感觉。

Grady：当然。

你（和很多其他受访嘉宾）引用面向对象开发作为正确设计的主要元素。它的基础性有多大呢？

Grady：对我来说，这听起来很奇怪，但恕我直言，面向对象是一种结果，而不是第一推动力。我曾经有机会检查许多研究领域的大量的复杂的系统，我发现最好的一些（最有用、最漂亮、最精巧、最……随便你怎么形容）在其开发过程中都有若干共同特点，那就是：简明扼要的抽象、良好的关注分离以及均衡的责任分布。抽象在很大程度上是一个分类问题，面向对象的机制特别适用于分类。

在大约 20 年前的 OOPSLA（译注12）会议上，Ward Cunningham 和 Kent Beck 在 Christopher Alexander 的工作基础上首次提出了软件设计模式。这是一种成功，一种温和的成功吗？

Grady：我个人的观点是，那并不是因为任何人的存在或者去世，或者是我的公司或任何人尚未诞生的原因，而是因为 Alexander 的工作很有趣，它流入软件空间给人们提供了很多灵感。我认为是小社区会从中受益，但大社区则没有。模式语言也是如此，我发现在许多地方，它可能没有像它能够或者应该的那样占据优势地位。

您所说的模式语言，是指类似于 Alexander 说的：“让我们创建一个词汇表，我们可以通过它来设计理念的再现”吗？

Grady：完全正确。在我曾经从事的诸多行业中，四人小组模式（Gang of Four）的概念可能还是一个书本上的概念，而不是一个用来构建自己的系统或其架构的实际应用。它在有些地方发挥了巨大作用，但它没有我希望看到的那样渗入主流的根源。它不是具体的细节问题。它是有关思考问题的方式问题。

这个问题如何解决呢？

Grady：我参与项目时，倾向于帮助它们开发一种模式语言。他们需要去理解。他们需要懂得“模式是什么”。然后，我尝试去鼓励他们寻找并命名他们在自己的系统中开发的模式，并开始为这些模式编写文档，真正使这些模式变成他们自己的东西。

译注11：棕场（brown field），此术语出自建筑行业，未开发的（尤其是未污染的）土地称之为绿场，而以前开发过的（通常是被污染的和被废弃的）土地称之为棕场。

译注12：OOPSLA（Object-Oriented Programming, Systems, Languages & Applications）是美国计算机学会（ACM）的一个年度性会议，主题为面向对象编程系统、语言以及应用程序，主要在美国举办。

374

哪些模式可能反映了 business 规则和约束条件。

Grady：它取决于该领域的性质。他们解决问题的增值方式、创新的方式是什么？是设计模式？

此时我是为一个人造卫星项目工作。它只有大约 50，000 行 Ada 代码。有一些令人惊异的潜在模式，存在于这些开发者的脑海里。而且挑战就变成命名那些东西，因此开发者就可以与这些潜在的模式很好地沟通。这些架构师非常聪明，但是只要你无法给一个东西命名，那么就很难讨论它、称呼它、操作它以及与其他人沟通。

让我们以 APL 为例。它的特定数据流可能没有必要直接地转换成其他语言。

Grady：对。Ada 在很多方面都是非常棒的语言。考虑我们看到的进入 Java、C++和其他语言中的特性。并发的问题内置到语言、异常机制、泛型机制、抽象数据类型当中——Ada 走在了它所在时代的前面。

第 15 章 375

Perl

Perl 爱好者们称它为“病态折衷垃圾列表器”和“瑞士军刀”，并对 Perl 的格言“做事不只一种方法！”扬扬得意。创始人 Larry Wall 有时把它描述成一种胶水语言（译注1），最初的意图是能够使 Unix shell 和 C 一起帮助人们来完成事情。它融入了 Unix 系统的语言学原理和设计决策（并炫耀说 CPAN（译注2）可能是所有语言中最大的收藏库）。很多程序员都在焦急漫长地等待着 Perl 6 的修订版，这种语言设计寿命至少为 20 年。

译注1：胶水语言（glue language）是用来连接软件组件的程序设计语言（通常是脚本语言）。

译注2：CPAN（Comprehensive Perl Archive Network）是一个巨大的 Perl 软件收藏库，收集了大量有用的 Perl 模块及其相关文件。网址为：*www.cpan.org*。

15.1 革命性的语言

您是如何定义 Perl 的呢？

Larry Wall：Perl 一直在对如何更好地将某些自然语言原理纳入计算机语言之中在更加务实的层面上进行着持续不断的实验，而不是像 COBOL 一样处于肤浅的句法层面上。人类语言的一些基本原则是——噢，我还是给你列一个清单吧：

- 语言表现力比 learnability 更加重要。
- 如果你碰巧是个小孩子，那么使用“儿语”同你交流就再合适不过了。
- 即便是在完全学会该语言之前，它也是很有用的。
- 大概说明同一件事，通常会有若干种恰当的方式。
- 每一种语言表达都可以从多种上下文中立刻获知它的意思。
- 你的语言对你当前应该优化的上下文是不知情的。
- 你的语言并不会强制执行任何特定范例而排斥其他范例。
- 高效交流需要一种特定规模的语言复杂性。
- 语义网络通常不能很好地映射到正交空间。
- 富于快捷性；公用表达式应该比非公用表达式更短。
- 并不是每件事都易于表达；如果能将一些很困难的事情表达出来是不错的。
- 语言当然会有动词、名词、形容词及副词等。
- 当句法结构很明显时，人类擅长于消除句法的二义性。
- 语言会自然地根据暂停、语调、重音和步调等添加标点。
- 当谈话主题很明显时，语言会充分利用代词。
- 语言应该理想地表示出解决方案，而不是讨论它们自己的结构。
- 对于一种语言的成功来说，健康的文化要比具体的技术更加重要。
- 语言的首要目的是去和人们进行交流。
- 只要别人能够理解，说话时要尽量使用一种口音。
- 亚文化有一些特殊的问题，而且通常会导致产生有用的方言或次语言。
- 当面对不同的方言或重音时，人们会学会“框架转换”。
- 当你能与某种次语言轻松地交流时，框架转换会更加高效。
- 对于现存的任何一种语言来说，在漫长的时间里发生演变不可避免。
- 对于大多数的交流来说，“更坏即是更好”也很不错，不过有时“精益求精”当然更好。
- 把书面文档放到历史的背景中进行评价特别重要。

多年以来，上述原则已经对 Perl 的设计产生了深远的影响。毋庸置疑，其中每一条原则都可以扩展为一段、一章，甚至是一篇学位论文。即使不考虑计算机语言，语言学也是巨大的多学科研究领域。

另一方面，这些原理大多已经在很大程度上被其他计算机语言忽略了。由于不同的历史原因，比起作为尽最大努力来进行跨文化交流的工具，很多语言设计者更趋向于把计算机编程看成是一更类似于公理数学证明的活动。

当然，另一方面——专注于这些语言原理，也使我有时忽略了来自计算机科学的一些重要观点。现在，我们正致力于修复这些瑕疵。

这些观点都有哪些呢？

Larry：其中最主要的一个观点是我们在 Perl 早期设计中摔跟头的地方——或许我应该坦诚地说“我曾经摔过跟头”——是正确的作用域的概念。是的，Perl 5 确实具有变量作用域这个词，不过，在很多地方它的确不能非常准确地给出作用域范围。而且，它仍然有许多全局变量。

Perl 5 也还有许多超距作用。与 Ruby 语言目前弄了很多瞎搅和的类型一样，Perl 犯了同样的错误，也就是说，插手事情的内部结构并瞎搅和一气，结果导致了怪异的超距作用。

我们已经知道，很多东西都有不同的作用域。比较典型的是，我们可以把信息依附到一个对象、一个词法作用域或者一个动态作用域上，当然，我们还有文件作用域、进程作用域、线程、类型、元类、角色、原型、事件、句法及事务等，此处不一一列举。你甚至还可以把优先权级之类的东西看成是一种另类的作用域。空间或时间里的每一个位置都有跟它自然而然地相关或者附属于它的东西，如果它们依附到别的位置上面，那显然就是放错了地方。我是随着时间的推移才慢慢学到了这些知识的。有些人会说那也太慢了。

另一个原则是懂得你的数据结构可变和不可变的重要性。未来几年，它们会因并行而变得更为重要；除非你非常清楚哪些东西随时可变，否则，你实际上无法做好那些牵涉到并行的事情。这是 Perl 历史上一直在遮遮掩掩的东西。我们把一切都看成是可变的。

可变性有时甚至是需要你留意的。

Larry：是的。对于小程序和朴素的（na.ve）用户来说，这是一种很好的心态：如果它们不是非常复杂，那就不会违背新手们的期望。

另一方面，它使大规模编程变得更加困难，因为你必须更加注意可变和不可变、公共和私有，当你被允许或不允许改变事情时，你应该知道二者的差别。在 Perl 特别是 Perl 6 的设计中，这些事情已经变得日益重要。

同时，我们想要设法保持相同的感觉，同时如果有可能，把一些自大的概念隐藏起来，以至于新用户几乎可以忽略它们。不过，因为它们仅仅是隐藏，这并不意味着它们不存在；如果 Perl 的确在底层工作得很好，那么，我们至少有希望在出现错误的时候把它们探测出来，而且有希望知道何时通过多核来扩展是合理的，且不用过于担心事情的进展不妙。为了做到这一点，你必须追踪数据依赖。这实际上意味着你很清楚一个指针什么时候被视为一个值，什么时候必须被视为一个对象。

Perl 是以文本处理和简化系统管理的工具集面世的。现在情况如何呢？

Larry：Perl 现在实际上包括两件事。首先，在 Perl 5 结构中，它自始至终都是一个非常稳定的例子：真正擅长文本处理的一种 API 胶水语言（以一种反常的有用文化倾心打造的大量扩展而实现扩张）。这就是 Web 主

要以 Perl 为原型的原因所在——因为 HTML 是文本，而且人们想要编写包括来自数据库等各种出处的数据在内的混合 HTML。扩展要做的是已经存在的事，或者是易于编写的那些事。

不过，Perl 还有一个 Perl 6 版本，在这个版本里，我们试图修复 Perl 5 中的每一个错误，而不用动其他的地方。我们认识到这是不可能的，不过无论如何我们都要做这件事。我们彻底重新设计了这种语言，同时仍然遵循与以前相同的基本设计原则。

即便是以目前的部分实现形式，在很多人看来，Perl 6 都已经是一种相当酷的语言了，而且在它完成后，很有希望大量使用衍生语法使之成为同时拥有自描述和自分析功能的语言，这样一来，经过优化，它就可以顺利发展为 20 年内我们想要的任何类型的语言。它同时还带有调节器，用来调整它的方方面面，包括将那些你目前并不感兴趣的所有隐藏起来的能力，这取决于吸引你的是用何种模式来解决手头的问题。

没错，无论如何，这就是我们的梦想……

你是如何从编写一种处理文本和简化系统管理的工具转变到一种完备的编程语言的呢？这是深思熟虑的一步，还是循序渐进的转变呢？

Larry：嗯，那些并不是互不相容的，那是个好事，因为我能把这个过程描述为一个深思熟虑的、循序渐进的转变。语言是一个奇妙的场所，而且从一开始我就很清楚：我应该能够与时俱进地持续改进它以满足“当前”之需。不过，不管如何深思熟虑，这肯定是一个循序渐进的过程，如果没有其他原因，你就必须等待到“当前”来修改，直到你真正知道接下来想要什么为止。

话虽如此，我认识到 Perl 不仅可以使事情变得更容易，而且还可以使完成困难的事情变为可能。Perl 2 只能处理文本数据，因此我告诉自己：“Perl 只是一种文本处理语言；如果我教 Perl 如何去处理二进制数据，有谁会知道它止于何处呢。”然后，我意识到世界上的大部分问题都是文本问题，不过也需要处理少量的二进制数据；增加该问题空间的解决方案应该能大大地提升 Perl 的适用性，即使只是能粗浅地处理二进制数据。因此，Perl 3 就增加了二进制数据处理功能，而又有谁会知道它止于何处呢？

而且，我开始把所有这一切都看成对 Perl 的早期概念的一种延续，那时我认为 Perl 不会有什么任意限制，比如困扰早期 Unix 工具的那些限制等。截断一个字符串仅仅是因为它包含了一个偶然的 null 字符，这大概和你截断它一样糟糕，因为你的缓冲区实在太短了。你甚至可能会说，推广一种语言的过程只不过是在一个或另一个层次上去除各种任意限制。

您更喜欢自由还是更喜欢有条理？您更喜欢按一种方式来做事，还是喜欢使用不同的方式来达到相同的目标？

Larry：这不是一个特别有意义的问题，除非你通过“方式”或“达到”来定义你的意思、你允许什么样的最优化，以及如何允许不同方案的排列和组合来实现增加。自然语言就像以前的老城一样，很少有街道成直角交叉，而且你想去什么地方都有几条不错的道路。显然，如果你统计一下到某处的所有可能的道路，而不仅仅是最好的一条，那可能会是个天文数字（古戈尔派勒斯）。

即使在一个按照严格的矩形块构建的完全正交的城市里，到达目的地也会有数不清的方式，除非你将解决方案限定于给定的维度，就像数学里的向量给出了某个三维空间的位置，并说明你想要一个最小路径。不过作为一个人，你在城市里的行动可不像定义一个向量那么简单。当处在一座城市或一种语言中，你可以对很多不同类型的外部事物进行优化。它既可能有、也可能没有一个单独的最佳解决方案，或者一种解决方案在一天的某个时间可能是合适的，而换个时间可能就不再合适。你可以对访问沿途所有的公园进行优化，或者你可能只是想要置身于河流之外，或者就在河中。

因此，你的问题答案实际是，任何极端都不是最佳的。我怀疑自然语言的分形维数不止一个，但却远小于 1000。如果你真正只有一种方式来说明这个问题，那么你就会被一个机器人程序员取代了。如果你真正有 1000 种看起来等效的方式来做这个事，你会很快地追赶上某种类型的剃须刀，在你选的每一个点上都可以把选择削减到一个好处理的数量上。

15.2 语言

Language

很多人都称赞 Perl 非常非常擅长文本处理。这和你创建这种语言时脑子里关注语言学有关吗？

Larry：哦，这个问题问得好。有人想说是对的，而别人则可能会说是错的。

一个能说明问题的证据是，尽管 Perl 设计试图以与自然语言同样的方式来解决特定层次的问题，但它的确不擅长分析 Perl 代码（通常是使用 yacc）。我很怀疑，如果我回答你“是”，那么 Perl 1 应该已经把目标更多地放在语言分析上，而这正是 Perl 6 现在的目标。不过它不是这样的。如果你愿意，可以把它称之为分割，不过 Perl 1 的任务是尝试解决文本处理的一个更有限的形式，但不是我们在考虑自然语言时大脑所做的那些工作。

我回顾了一下几年前 Perl 1 的测试组件，尽管那些代码是可辨别的 Perl 语言，甚至考虑了很多 Perl 语言的概念，比如上下文已经随着时间的推移而发展演变了。最一开始，您的脑海里有多少固有的 Perl 概念呢？

Larry：毫无疑问，从一开始上下文的概念就很重要，而且在 Perl 4 中上下文的概念甚至还有相当成熟的发展。不过我认为，我是逐渐、逐渐地才明白上下文的重要性。作为一名语言学家，我早就了解了上下文；在我特别喜欢的语言理论之一的法位学中，多级上下文是极为重要的。单词的词法分类是不同于它的使用方式的。你知道，“动词（verb）”是一个名词，即便后边加上“ing”还是如此。因此，我脑海里一直浮现着这个概念：你可以以不同的方式应用一个特定的结构，这些方式不仅可以通过建立它自己的结构和类型来进行驱动，而且也可以通过它的语义和文化上下文进行驱动。我认为，在早期的 Perl 上下文设计中显示出的并不是像标量和列表一样的低层次句法，不过更多的是当你试图完成一项工作，或者是当你在一定的上下文里编程时试图完成在外部程序时的概念。因此，在一种语言中有多种表达方法非常有用，以至于你可以为外部事物来进行优化。

换句话说，你的程序应该表示问题的上下文，而不仅仅是试图用语言的上下文来表示问题。

Larry：没错，谁是主要的，这才是问题的根本所在。一个特定的问题通常会有多种不同的表示方式，特别是你考虑不同的编程模式时更是如此。它们中的一些会更加自然地映入某些问题空间，而且其他的则会更加自然地映入其他问题空间。如果你把问题看作是某种数学证明，那么像函数式编程的东西就会更好，更具声明式编程风格的那些，你知道，它们会有从来不变的通用含义，会有许多不可变的状态嵌入到直接接触的概念当中。不过，如果你做的是仿真之类的事情，那么你要更多地从随时间变化的对象方面来考虑问题。对于你试图解决的一个特定的问题，可能会有一些同构的观点，不过，不同的编程模式会强制你把可变状态放到一个地方或另外一个地方。该状态处于对象的什么位置是很明显的；那就是对象为了什么目的。函数式编程中的状态并没有如此明显——该状态隐藏在堆栈中，并且各种 monad 和函数调用的方式都已经安排好了。

那么，作为一名程序员，您是如何看待上下文格式这类问题的。

这就是萨丕尔-沃夫假说（译注3）。我并不相信大牛的说法……

您事先就决定不相信大牛。

Larry：是的。我选择了预先决定。或者这可能只不过是我脑子里一开始就没有语言学这根弦，因此我做了许多非语言学的思考。我并不认为语言已经控制了我的思想。尽管如此，更弱形式的假说更适合我：你选择的语言的确会对你如何选择产生一些影响。因此，如果你确实想让某种语言能与很多不同类型的问题更好地阻抗匹配，你需要的就是一种不会强迫你以任何一种特定方式思考的语言。

有几种精巧的上下文：有 Unix 单行程序，你可以用它来表示一种有用的运行 Perl 程序；有 shell 脚本上下文，它看起来就像是带有 Perl 5 的所有功能的 shell 脚本；还有独立单页 CGI 程序上下文。在某种程度上，它是一种用于“尽管去做”（译注4）的不同上下文。你可以认为它既是明显的 Perl，也是明显的那种“尽管去做”的风格。这是一种 Perl 单行程序，或者是一种用 Perl 编写的 shell 脚本。

Larry：在语言理论中，我们称之为“语用论”，比语义学更偏向社会学一些。一些语言学家趋向侧重于低级的音韵学或句法。在很多计算机语言设计中你可以看到许多相同的狭隘视野，一些设计者还没有深入思考话语如何用于实际问题。如果你从单行程序和 shell 脚本进行讨论，基本上你会在自然语言中看到同样事情的发生。你可以在公共汽车站跟别人说一句话用于交流。英文里满都是简练的话语。在另一个极端，你可能把各种各样的文字风格同编程模式进行类比。在写文章时，你可以受到各种层次的训练。这些风格拥有不同的规则，有时你可以打破这些规则，但是有时打破它们是很愚蠢的，不过语言自身并不试图强制执行任何一种特定的体系。

自然语言在那点上是中立的。语言是诗人的仆人——它是艺术家以一种艺术的方式试图做“别的东西”。这些别的东西推动着整个过程，而且应该正当地推动整个过程。

在某种意义上，自然语言是非常简陋的。它们并不告诉你必须用什么方式交谈。你的语文老师告诉过你交谈的方式，不过基本上，人们忽略了这一点，而这是一件好事。

然而，对于计算机语言来说，也有类似于语文老师那样的规则，而且对某些话语，你应该遵从这些规则，除非你知道为什么要打破它们。话虽如此，计算机语言也必须能被计算机所理解。这就强加了附加约束条件。特别是，我们恰恰无法使用一种自然语言做到这一点，因为，在大多数情况下使用自然语言交流时，我们假定倾听方是一个极为聪明的人，而倾听方也会假定说话方也是一个聪明的人。如果你期望一台计算机会如此智能，那么你会非常失望，因为我们不知道如何编程才能让计算机做到这一点。

即使 Perl 是第一个后现代计算机语言，实际上计算机也不能很好地理解讽刺的话语。

Larry：确实如此。当它们应该应答反馈时，它们并没有真正地理解。这是因为它们没有理解它们什么时候没有把握。留心你自己，因为有很多人也是这样，不过总有个大概齐。计算机很快就会被提升到无法胜任的级别。

译注3：萨丕尔-沃夫假说（Sapir–Whorf hypothesis）是一个关于人类语言的假说，由语言学家兼人类学家萨丕尔（Edward Sapir）及其学生沃夫（Benjamin Whorf）所提出，这项学说认为，人类的思考模式受到其使用语言的影响，因而对同一事物时可能会有不同的看法。

译注4：尽管去做（getting things done，GTD），又称为“完成每一件事”，它是一种行为管理的方法，也是著名时间管理人 David Allen 写的一本书的书名。其主要原则在于一个人需要通过记录的方式把头脑中的各种任务移出来。通过这样的方式，头脑可以不用塞满各种需要完成的事情，而把精力集中在正在完成的事情上。

您经常说，比起数学家来，计算机语言设计者应该更加注意语言学家，因为实际上语言学家更知道如何去同人们交流。

Larry：这样来说吧，他们懂得现实的人们是如何交流的。

数学家知道如何相互交流，但这并不是说你是否想把数学家当成现实的人们；我确信数学家能够找到某些集合论形式的方式来肯定他们是现实的。不过，我认为，比起数学家来，语言学家应该更加注意心理学和语用学。因此在那些事情上或许语言学家可以有助于计算机变得更加智能。

你相信这种观点吗，一个严格的而且可证实的小编程语言模型可能在人们必须使用它的现实世界里并不起作用？

Larry：总有这样的一小拨人，他们愿意通过很小的“漏斗”来倒出他们的想法。在这个意义上，你可以找到愿意去这样做的人们，给定语言在预期的问题空间上也会是成功的，除非遇到其他灾难。

反过来不一定也是真的。一种足够大的语言可能会因为多种原因而无法获得成功。它可能是太难实现，可能是太难让人们学习一个有用的子集。我认为让人接受一种更大的语言存在两个主要的困难。

不过，我认为没有一种计算机语言已经创造出接近于自然语言的复杂性。而且人们的确非常愿意——是的，或许不是美国人——比如说大多数不讲英文的人的确非常愿意学习多种语言。

我已经尝试学习了几种自然语言，而且它们确实很难懂。词法复杂性、因语言而异的奇怪语法规则，语言让你以不同顺序来考虑问题，某些语言让你这么说而其他语言则不会让你这么说，反之亦然，等等。学习一种自然语言是很难的。问题是，有人会创建出一种足够丰富的计算机语言，让人们愿意为之越过学习障碍吗？这种语言能够使人们有效地学习它的洋泾浜语子集吗？当人们有充分的动机以某种方式发现一种公用语言时，我们就会看到脱胎于自然语言中的洋泾浜语和克里奥尔语（译注5）应运而生。

我们能利用这种动态性吗？我们能设计出不会比实际需要更复杂的语言吗？这些问题确实很好，而每一位语言设计者的回答则各有不同。Perl 使用层次化的命名空间，不过一些计算机语言只有一个平面名称空间，并会把一切都塞进一个字典里。证据表明，英文就是如此。特别当你的名字是 Webster 或 Johnson 时。

英文的确支持行业术语和隐式交流方式的概念。

Larry：它肯定是这样的。它提出了词法定界（译注6）的概念，它对控制 Perl 6 语言的多样性是非常重要的。这对于我们在词法作用域中的每个点确切地知道我们说哪种语言也是非常重要的。不是说这对人类非常重要，人们非常聪明，最后总能推测出来。编译器很可能没那么智能，因此对编译器来说，了解正在分析哪一种语言是至关重要的。

译注5：洋泾浜语（pidgin）和克里奥尔语（creole）是混合语的典型，是一种跨文化的语言。它们的形成和发展反映了语言融合和发展的过程及共性。克里奥尔语指已经成为某一群体母语的洋泾浜语。洋泾浜语是操不同语言的人们进行贸易或其他目的的交际语言。洋泾浜语是为了应付有限的交际而形成的简单语言，一旦它发展到一定阶段，被某一集团当作母语并作为媒介语成为儿童的本族语（母语）时，它就发生了克里奥尔化（creolization），成为克里奥尔语。

译注6：词法定界（lexical scoping，也叫静态域）是许多编程语言约定使用的，变量只能在这套范围（按功能排列）内被一些已经定义了的代码段调用（引用）。在被编译后，这些范围将确定下来。变量定义用这种格式的时候叫做私有变量。与之相对的是动态域（dynamic scoping）。动态域产生可以在定义变量的代码段外调用的变量。这样定义的变量也叫公共变量。

如果我们可以防止编译器犯糊涂，我想我们也可以防止人们犯糊涂。不同的词法作用域，文献中不同的段落（语言心理学中称之为不同的框架）使用不同的语言。人们一直进行框架转换。他们知道何时去闲谈，也知道何时正式地讲述。人们在没有意识到它的情况下总在不断调整。这只是自然语言工具箱的一部分。

384

你如何知道你试图解决的问题是否需要先解决工具或语言？

Larry：这个分界线确实很模糊；单一工具的概念和工具集的概念混杂在一起，而且我认为你可以将一种语言视为一组工具，因此这并没有一定之规。一把瑞士军刀是一个工具还是多个工具呢？

或许更重要的区别是如何把工具集很好地组织在一起。瑞士军刀可能是一套很方便的工具集，不过同时使用它的多个功能也是颇有难度的。

我想语言与工具或工具集的区别在于它们的通用性如何（尽管当然有专用语言和通用语言之分）。当一种语言被看成只不过是一种工具时，语言确实擅长以意想不到的方式凝聚各种思想。如果你的问题需要这样的组合性，而且不会因为语言的线性本质过于被动，那么，为了解决你的具体问题（或者至少是你的问题的一部分），定义一种语言就是一种很好的选择。

除了第一个问题之外，你的语言随后可以给你提供一个机械工厂，它可以使你更容易使用其他工具，而且在某种程度上它确实如此，你已经一劳永逸地改善了你的生活。有时，预先知道了这些，你会不辞辛劳地创造一种语言，即使你知道它不可能是解决第一个问题的最快途径。不过，如果你确实很懒，你会弄清楚如何去把额外的工作分摊到该语言的所有最终使用中。

如果一种语言开始偏离某个专门领域，它会有怎样的变化呢？这样说是很公平的：把 Perl 的早期版本描述成洁净的 Unix 分支语言，设计用于将各种语言混合在一起的 API。当一种语言从专用变成更通用时，它会如何变化呢？

Larry：从我的经验来看，当你已有一个这样的特定领域语言时，你觉得它们有不同的结构是很自然的，但从某种意义来讲，它们是以一种专门的方式来定义的。这比它们觉得大概是这样更为理性。

人脑正试图与这些事情联系起来吗？

Larry：没错，当这些发生时人们会贸然给出不必遵从的结论。你最后会以一个很大的常见问题集合而告终。如果你来看一看 Perl 的常见问题集，它们内容很少，这可能会被解释成所谓错误归纳过程的证据。当你把一种语言发展成为一种更通用的语言时，你回过头来看一看所有的地方，你会说“人们为什么会那样概括呢？这种语言应该支持这些吗？对它实际上在底层如何工作，如何做他们所期望的而不是他们不期望的工作，我们能做哪些最小的修改呢？

如你所知，Perl 6 设计过程是最大的设计过程。在过去的几年中，我们一直在实现这个过程。

385

您能举个例子吗？

Larry：在 Perl 5 中，`$var` 是一个标量，`@var` 是一个数组，而且`%var` 是一个联合数组。新用户通常认为`@var{$index}`和`%foo{$key}`是你编写索引的形式，不过由于历史原因 Perl 5 并不是那样做的。这些文档都要经历一番周折来解释为什么事与愿违。到了 Perl 6，我们决定应该更好地解决语言问题而不是用户问题。

在 Perl 5 中，如果没有提供参数的话，不同的函数默认是一个`$_`的参数（当前主题），而且你基本上必须记住那些函数列表。如果没有记住，很可能会错误地总结所有函数的功能。在 Perl 6 中，我们选择不用任何函数

默认方式，因此将不再有任何错误总结的后顾之忧。取而代之的是一些轻量级而显式的句法，用于调用当前主题的方法。

只要你的语言强迫你记住一个主观的列表，正是这个列表的主观性表明了：有人会认为某些函数应该在此列表之中而实际没有，反之亦然。这些事情趋向于逐渐渗透到你的设计中。Unix 文化在首先创造正则表达式句法时，其中只有非常少的元字符，因此它们易于记忆。随着人们为模式匹配添加了更多的特性，他们也使用了更多的 ASCII 符号作为元字符，或者为了维持向下兼容性，他们使用了以前不合法的更长的序列。毫无意外的是，结果肯定是乱成一团。你应该能够在很多程序看到错误的归纳：慌里慌张的用户可能会在 regex（译注7）中将一切都反斜杠化，因为他们记不得哪个字符才是真正的元字符。在 Perl 6 中，随着对模式匹配句法的重构，我们认识到大多数 ASCII 符号已经成为元字符了，因此预留了所有的文字数字作为元字符，以减轻程序员的认知负担。将不再有元字符列表，而且句法更加整洁。

我用于描述 Perl 设计的一个单词是“融合”。你可以从其他地方汲取很多好东西，并把它们组合成一个连贯的整体。你如何平衡融合这些通用性的观点和在概念及特性之间的连贯性？

Larry：糟糕的是，一些人会这样说。

这是语言设计者令人遗憾的的困境，他们只能通过直觉来把握平衡。

过去的实验效果并不是很好。

Larry：我同意这种说法，至少总的来说是这样。它已在 Perl 空间有很少的特权 rare privilege，实际上有一个被称为 Perl 5 的成功实现，它会允许我们尝试一种称为 Perl 6 的不同实验。

基于已经学到的东西，我们正积极地考虑打破一种不同的平衡。毫无疑问，出现了很多不好的猜测，如果不是以特性集的形式，至少也是以为了给定的特性默认行为应该如何的形式。我们此刻想要犯不同的错误。

看它是否起作用会非常有趣。我们向人们教授 Perl 的早期版本必须做很多解释，为什么有些事情必须是以它们已有的方式进行。现在，我们必须回过头来说：“噢，我们那时的想法是错误的。这意味着你现在想的是错误的”。这变成一个有趣的文化问题，你是否并在多大程度上可以带领人们走出你曾经带领他们进入的地方。使用圣经的一个隐喻，你已经把他们带到埃及，现在你正试图把他们带往福地。

一些人会追随你，而且一些人会渴望得到韭菜和洋葱。

15.3 社区

Community

您一直鼓励社区参与设计和实现。这对完成任务来说是必要的吗？它是不是您的工作风格或美学意识的反映呢？

Larry：我确信它必定是各种因素的组合。

毫无疑问，构建一个社区的早期动机只不过是让人们给你提供正面和负面的反馈。正面反馈是最好不过的了，

译注7：regex 是一款正则表达式辅助程序，它通过高亮匹配及替换内容使整个流程可视化，帮助用户更好地理解和使用正则表达式这一强力工具。

不过负面反馈也很有用。

从语言学的观点来看，只有很小的社区说这种语言，它的表现不会太好。在某种程度上，社区对于一个项目活力的帮助是相当明显的。

不仅仅是这些，我只是想让很多人使用我的作品，因为我喜欢帮助他们。随着事情的进一步发展，项目会变得非常大，以至于我也需要帮助了。大约在 Perl 4 结束时，我认识到事情正向各个方向分化，而且人们在编译不同的 Perl 可执行版本。很明显，Perl 这时候需要一种模块化扩展机制，而且不同的模块应该由不同的人来负责。

在 Perl 5 发布后不久，当时就已经非常清楚：Perl 自身变得太大，在所有的扩展期，任何个人都无法胜任集成管理者（更不用说设计者），而这还没有到达极限。工作需要推动。基本上，在 Perl 5 的早期阶段我就很清楚须要学会把工作委派给别人。但难题在于，唉，在我脑海里还没有一个管理框架。

我不知道如何去委派别人，因此我甚至为了委派而委派，这看起来好像解决得还不错。我并没有让人们做什么事，不过其他人逐步提高，在正确的方向上发挥管理能力，准备了 to-do list，并与其他人协调。很有趣的是，尽管我没有微观管理的能力，或者是因为我没有能力做这些，因此，社区在某些方面更健康。

387

我可能确实有一种管理才能：我从未犹豫告诉人们他们应该做什么，尽管通常是使用如此抽象的语言，他们还没有领会我说得含糊不清的想法。

您说过对成为一名管理者并不感兴趣，而且您认为您并不具有那样的能力。但是您仍然领导着 Perl 社区。这是您最重要的工作吗，或者您只是想让人们前进而且您愿意做这些必要的工作？

Larry：比起 Rodney King，我可能是想让人们参与得更深一点。通常我认为，我的工作是提醒他们事情有一点儿错误苗头，并施加一点儿压力以防止错误继续扩大。我很少会让它发展到必须采取严厉措施的地步，我只有一次给 Perl 社区的人打电话并大声斥责他们。

顺便说一句，这样很有效。

毫无疑问，如果我的管理水平更高一些，我应该能够将事实上在 Perl 文化内涌动的有害之流减少到最低程度。在 Perl Town 的一些地方你不应该摸黑前行，即使没有很亲切的、无所不知和无所不能的规则可以告诉他们每个人是否应该始终在做什么。

毫无争议的是，不是上帝为我们尝试做这些。正如他在 Time Bandits 中所言："我认为在自由的愿望下需要做一些事情"。

Perl 5 最大的成功之一就是 CPAN，而且对我来说，这似乎将成为主要的可扩展性形式。是您做出具体的设计决策以鼓励这样一件事的发生，还是一个偶然发现的意外历史事件呢？

Larry：嗯，跟往常的这种历史问题一样，答案是"成功了"，这是我做出的决策；"失败了"，这不是我做出的决策。

模块化系统的设计相当"隐蔽"，很多人都会提出模块并发布它们，甚至还会有这些模块的收藏库。其他语言会有各种类型的可重用软件的收藏库。当然，我无法预料它的规模及人们会将 Perl 连接到什么怪异的东西上。

从一开始起，我就一直关注着设法让 Perl 尽可能与很多不同的 API 对话，无论是 shell、环境变量、操作系统，还是终端。我是第一个使用 XML 分析器来编写小工具的人。对我来说，在不同的阶段有一点一直都很重要：

Perl 不是一种试图把什么都融入自身的语言，而是以尽可能多的方式连接外部世界。这正是胶水语言的本质所在。与 Icon 这样的语言相比，它设法以一个保守的方式在内部定义每件事。

现在，我并不想要过分简单化；在一个更大或更小的程度上，所有的语言至少部分要从头再来。

解决方案总有这样一种压力，要求 100%利用 Perl 语言，或是 100%利用 Java 语言，或是 100%利用其他的什么语言。这样可以简化配置和测试等这些简单的事情。这使得分发更加理想。它会让聘用程序员更容易。另一方面，如果你的语言完全是这种方式而且你的文化也完全是这种方式，这实际上是一种恃物自傲的有害方式。

这需要一种平衡。Perl 总是在犯使用太多外部 API 的错误，而不是太少。不过，最好是同时支持这两种方式。

我采用了这些连通性和语用学的哲学思想。当其在万维网出现时，我的确没有预料到它们的规模。实际上，Perl 以我从未料到的方式飞速发展，不过从某种意义上来说它也是隐式的。

您慎重地采用适当的机制，可以有意外的幸运发现或者在它发生时对它进行利用。

Larry：对于一些深思熟虑的定义，它们可能是或者可能不是左脑型的、右脑型的、后脑型的、前脑型的或者中脑型的。通常都是没脑型的。从一开始，其他人的想法就对 Perl 做出了很大的贡献，从这个意义上来说，毫无疑问，很多都是“外脑型”的。

很多语言都有收藏库，不过 CPAN 有一个优点：实现者（Jarkko、Andreas 等人）只是构建足够的基础设施以鼓励开发，而没有对它进行限制。您所倡导的语言或社区特性哪一个可能做到这一点呢？

Larry：我无法保证 CPAN 实现者能做出优秀选择（或者不好的选择），不过，我认为 CPAN 在史特金定律（任何事物，其中 90%都是垃圾）的应用方面击中了要害。特别是你在创造一些新东西时，易于超裕度设计它，并设法去掉大多数 90%的代码，结果是你去掉了大约 10%的代码。毋庸置疑，我们已经看见过一些讨厌的无用代码后来变成有一定价值的代码，因此，通常需要一些耐性，并采用“更坏即是更好”的方法。而且一些人也是大器晚成。我们也是在边干边学。

至于说是什么让它在语言中变得可能，那很可能是 Perl 5 最重要的设计目标：允许任何人通过模块来扩展语言。从后来的 Perl 6 的观点看，我用不同的方式修复了 Perl 5 模块的设计，不过它还是不够好，所以 CPAN 才应运而生。我想我应不时地鼓励社区做出努力，尽管只是以最普通的形式。不过，大体上来说，它实际上意味着这样一个事实：大多数 Perl 开发者都对这个目标做出了极其重要的贡献。我轻舞大旗，把他们凝聚在一起，不过人们参加一个团队，大多只是因为他们找到一些志趣相投的人。而且，我很高兴地看到合作精神也传播到了其他社区。

社区的贡献出人意料吗？

Larry：我不知道实际上是还是不是。我想，或许整个强制性的严格和警告这种文化，对我来说有点意外：与默认的规则相比，人们要求给予更多的行为准则。我们决定把它融入 Perl 6，这在文化上已经可以接受了。这只是一个小意外。

对我来说最大的意外是我们开始重新设计 Perl 6 时。我们以 RFC（Request For Change）的形式征求建议，我期望征集 20 条左右。实际上，我们得到了 361 条。有多少惊喜就有多少烦恼：它几乎超出我预期的 15 倍之多。祸不单行的是，有很多问题人们认为可以孤立地解决，而实际上并非如此。因此，在过去的 20 年里，

我最大的惊喜就是来自社区对系统化重新设计的需求。我对某些文化传播的成功也感到很意外（欣喜）。Perl 3 中最初的双重许可证小工具已经使得 Perl 大行其道，无论是在对企业文化拥护者有疑虑的小工具作者社区，还是在对小工具作者社区有疑虑的企业文化拥护者中间都是如此。

这两个群体都发现使用这种方法可以消除疑虑：实际上我并没有强迫任何人来决定他们是否使用真正的 GPL（通用公共许可证）或 Artistic License（艺术许可证）的原则。

我几乎从来没有见过有谁会真正选择其中之一。

Larry：是的。它是许可证数量的叠加，而人们根本不会遵从这些。我想我对这个事实感到非常遗憾：我们创建了双重许可证这个术语，而且这开始意味着强制用户二选一。你从来都不能创建一个术语并使它表示你想要表示的意思。它总是在你认为可以得到的和实际上得到的之间徘徊。

15.4 改进和革命

在软件设计和开发方面，您采用哪种方式：改进抑或革命？

Larry：在某些方面，我是一个脑筋笨笨熊（bear-of-very-little-brain），因此，我个人采取一种改进的编程方式。当开发一个 Perl 程序时，我通常会做一些修改，接着运行一把，然后再做另一个修改，或许循环时间只有 30 秒。我没有花很多时间来调试，因为最后的一件事干的是对是错，通常会相当明显。我会不时地重做，不过这也都是倾向于改进方式，在做出修改和确认没有真正修改之间不断切换。

至于说语言设计，我的基本方式一贯如此：采取一个改进的方式，不过会“加快”突变速度，因此，如果你两次“快照”间隔足够长的话，看起来就会像一种革命性的改变。

对于 Unix 爱好者来说，Perl 1 看起来就像是一个彻头彻尾的 awk、sed 和 shell 变体，不过实际上是为了让人们能够接受它，才把大量的 Unix 元素加入到 Perl 当中。任何新语言都必须考虑移植问题，因此新语言都会大量借鉴现有的语言。（我们对这些借鉴来的文化有一些遗憾。特别是，regex 句法这些年来已经变得不合适，我们希望 Perl 6 能对此有所补救。）。

对于很多人来说，Perl 5 看起来像是 Perl 4 革命性的改变，不过事实上，这种实现是以外部不可见的各种中间形式实现改进的。某些操作码曾一度采用旧的“满栈”解释器来解释，而其他操作码则是通过新的“无栈”来解释的。Perl 5 在向下兼容性上还是很保守的；事实上，Perl 5 仍然可以正确地运行大多数 Perl 1 脚本。

在 Perl 6 中，我们最终破坏了它的主要兼容性，在某种程度上可以说是“抛弃原型”，并且迅速地改进句法和语义设计，同时努力保持基本的“感觉”，使它就像原来一样。尽管这次，我已经成功地让社区越来越多地参与到了重新设计的过程当中。

我们第一次发布 Perl 6 的成果时，得到了上面提到的 361 条 RFC，而且它们大多都是假定 Perl 6 只是 Perl 5 的一种渐进式变革，而没有任何根本变化。在某种意义上来说，Perl 6 设计只不过是概括、简化、统一及合理化那些增加的建议的结果，不过，对于没有参与设计过程的任何人来说，毫无疑问会感到 Perl 5 到 Perl 6 是飞跃是一场革命。然而，大多数 Perl 6 程序会非常类似于用 Perl 5 编写的程序，因为基本的处理思想是类似的。同时，Perl 6 也会更易于分化成函数式或面向对象模式的思想观点，如果你想这么干的话。一些人会认为这是革命性的变化。

对我来说，革命多半是人们假装他们没有经历过所有的中间步骤。Perl 的设计目的是帮助人们尽可能迅速跨越中间步骤，因此他们可以假装成革命者，这是非常有趣的。

这是什么样的革命呢？

Larry：我在这里说到私有革命。不会 Perl 的人告诉一个 Perl 程序员“我想做这个，但我不知道怎么做”，这时候就会出现革命。Perl 程序员说，“噢，那很简单。这么做”，并给他编写了一点儿既浅显又足够快的程序，完成了工作。不会 Perl 的人一看，“噢，真酷”。当有人说“噢，真酷”的时候，它就算得上一种“小”革命了。

在某种意义上来说，这是一种从改进到革命的连续统一体，它只不过大声对你说，“噢，真酷”——或者如果你碰到的是一个精英，他会说“噢，垃圾”。真正优秀的革命会使更多的人们说“噢，真酷”而不是“噢，垃圾”。

我想我的确相信那是一个优秀的革命。或许那又回到了神学信仰，至少在个人层面上是这样。我相信，只要给一点适当的提示，人们就能够在短期内使自己适应。

无论他们想不想这样。

Larry：是的。它就像现代科学家和希腊哲学家之间的区别一样，后者试图根据“第一原理”来解决一切问题。他们弥补某一方面知识的不足，但没有经过实践经验，你是不会得到科学发现的意外之喜的，科学经常会有意外发现，让你醍醐灌顶，并会完全颠覆你原本认为很好的思维方式。

例如，在 Web 上应用 Perl 5。

Larry：在 Web 上应用 Perl 5。而且毫无争议，当遭到小行星撞击时，或者许多的其他意外事故强加到我们的远祖头上时，有很多中间类型到陆地爬行或者爬进洞里。这里的普遍原理是：你应该既能适应稳定的小改进，也能适应大变革。

这些大变革具有可归纳性吗？这让我想起了 lambda 演算，你以四个或五个分离原理开始，然后推理和推断到达图灵完备性世界的方式。

Larry：是的，不过现实经常会以我们意想不到的程度打击我们。在一个非常温和的气候里，我可能以维持目前的体温为目的进行优化，不过有时小行星会以超大规模突然撞来。归纳会以逐渐显现的方式帮助你。它不必帮你预料在大量的前提失效时将要发生的变化。这就是你想要有一个大基因池的原因所在。基因就是你工具箱里的工具，而且你想要许多工具，特别是在小行星的背景（上下文）中。

你想要给人们提供可用的工具，供他们在那种情况下适应新的环境。

Larry：这又是整个可变的和不可变的问题。归纳构建在你的前提不可变的概念上。

是归纳还是演绎？

Larry：二者都有。我认为它们是同一枚硬币不同的两面，而且硬币有时会立在边缘。概率论假定立在边缘的概率为零。但事实并非如此；我自己已经做到了这些。这是一件非常令人惊讶的事情。我小时候和邻居一块儿踢足球，我们在抛硬币玩。他说“快猜是哪面”。我说“是边缘”。结果落下来立在了中间，因为它落在了硬草叶之间。有时候背景（上下文）会战胜概率论。

我想我的生活故事已叫“边缘”，成为了那段时光的一部分。

在潜在地多种竞争或协作实现 Perl 6 的概念背后，你的理由是什么呢？

Larry：理由分为好几个部分。我们已经提到你想要一个大基因池。一个健康的基因池要求大量基因的交换，这可能会是很有趣的。另一个原因是不同的实现会趋向保持互相诚实。不同的人们自然会以不同的方式来考虑这些规范，如果该规范有任何模糊性，他们很可能会发现。然后它就变成了一个在不同的实现中间就规范真正应该意味着什么进行协商的过程。

392

这听起来就像一个教育策略："你先把蛋糕切成两半，然后让你弟弟挑"。

Larry：是的。这是一种把某些设计工作强加给实现者的方式，它对于像我这样讨厌的设计者来说是必需的。

多种实现策略的另一方面是人们兴趣各异。他们首先会想要建立实现不同部分的原型。有时候，一个项目写了一半，然后才发现你做的假设会使另一半非常难以实现，因为没人曾经真正地设法完成它；与其这样，还不如让人们分别考虑不同的设计方面的原型，然后再互相分担这些危险点（*danger point*）。

我们恰好在上周或上上周看到了这个例子，SMOP 那些人其实对实现了解不多，但是他们在以下方面考虑到了很多细节：对于 list、capture、signature 及所有的这些至关重要的概念，它们是如何在一个较低的层次上一起工作的；惰性、迭代器和数组实际上应该在语义上如何使用等。

有一个科学的简化是：我们只研究这个特定的方面，而忽略其余所有的问题。我认为，强制我们在还没有思考的地方思考设计得出结论，这是很有价值的。

这是因为他们想要完全实现或因为有人为特定部分建立了原型吗？

Larry：我几乎从不关心。如果有人在为一部分实现建立原型，这实际上要取决于他们想要干到什么程度——他们有多大能力，他们还可以招募多少人来干等。那是我委派给其他人的另一个例子。现在要开发一个新大陆，人们需要在各个方向开拓。

由于不倡导潜在的实现，实际上你是在倡导实验。

Larry：是的，我们正在倡导一种扩散算法（flooding algorithm），如果对于一个公司来说，你想要的东西对它来说并不一定是必要的。取而代之的是，你设计项目时心里只有一个主要的目标，而且你已经有了一个特定的速率，而且它必须在某某特定的日期里完成。不过事实是，我们已经得到这个很好的并行处理器，称为开放源代码社区，而且扩散算法好像在并行硬件上比在串行硬件上表现得更好。不过这不是一个愚蠢的扩散算法；它更像一个焦急寻找食物的蚁群。因此实际上，我们是在优化将要运行的引擎。它不是一个对称多处理器，而是更像一个小工具作者贝奥武夫（Beowulf）集群（译注8）。而且这些小工具作者并没有完全相同的体系结构。

共享内存也是一个问题，特别是当状态不可变时。

393

Larry：这不仅仅是内存的问题，开放源代码社区是一个"一切非一致体系结构（非一致 Cache 体系结构）"，不过我认为，我们的方式是去加强它而不是弱化它。而且，对于蚁群来说，共享内存看起来肯定不像是一个大问题。

译注8：贝奥武夫集群（Beowulf cluster），又称贝奥伍尔夫集群，是一种高性能的并行计算机集群结构，特点是使用廉价的个人电脑硬件组装以达到最优的性能/价格比。

Perl 6 中的任何特定实现会限制发生在更高的语义级别上的事情吗？

Larry：哦，即使你设法创建一种新话（译注9），也没有什么语言能够绝对控制一切，而实现只不过是一种语言无法绝对控制的事情之一。在一定程度上我们设法练习控制，语言的定义涉及我们应在测试套件中包括什么内容，或者把它排除在外，而且我认为那些决定应该主要是不同的实现者协商的结果，也可以偶尔加一些语言设计者提的意见。所以，我们认为拥有多种实现非常重要，因为它们趋向于相互包容。

您会特别关注资源的可用性吗？

Larry：我认为，在有效地扩大规模方面，不同的成功范例都趋向于自我约束。当编程团队多于 6 人或 12 人时，他们最终会把自己分到不同的子项目。这实际上恰恰是某种程度的资源耗费，人们正在重新“发明车轮”而没有考虑其他的相关工作。

在某种程度上，他们正在重新“发明车轮”，嗯，这就是扩展基因池方面一个有趣的实验。毫无疑问，在你采用的任何方法中都有一些无效率的东西。无论是在程序员中同步分摊，还是这些东西后来又困扰你，因为你不可能什么事都自己首先做，无论你采用什么方式来完成那个项目都会有无效率的东西。

有证据表明，在领域里的小工具作者中把这些无效率的东西分散开，实际上可能会比你设法把它集中串起来解决得更快。假定你有足够的小工具作者来解决这个问题，而且这个问题是可并行处理的，而且你可以让志愿者相互沟通。

您无法告诉志愿者该做什么。您可以说，但没人听。

Larry：那是另一件重要的事情。他们会做自己想做的东西。因此正如格言所说：如果你无法改变事实，干脆让它顺其自然。

我很喜欢来自你的观点：“他们说更坏即是更好，不过我们只是希望在 Perl 6 中有一个精益求精的循环”。

Larry：对。这就是我目前的“边缘”。我希望你能感觉到它的锋利。

译注9：新话（Newspeak）是乔治·奥威尔小说《一九八四》中设想的新人工语言，是大洋国的官方语言，被形容为“世界唯一会逐年减少词汇的语言”。

第 16 章 395

PostScript

PostScript 是一种拼接式（concatenative）编程语言，通常用于描述桌面出版和电子出版系统文档。John Warnock 和 Charles Geschke 在 1982 年创建了 Adobe Systems 公司之后，发明了这种语言。Apple 公司的 LaserWriter 在 1985 年把 PostScript 推向市场，使得桌面出版成为可能。PostScript 很快成为文档交换事实上的标准。从此以后，PostScript 的后继者 PDF 已经在该领域取而代之。

396

16.1 为永恒而设计

Designed to Last

您是如何定义 PostScript 的呢？

Charles Geschke：PostScript 是一种编程语言，它的主要目的是提供以一种与设备无关的表示方式来对（打印的）页面内容进行高级描述的方法。

John Warnock：PostScript 是一种解释型编程语言，它仿效一个面向堆栈的简单虚拟机。除了创建大多数编程语言中常见的标准操作符之外，PostScript 还有一个非常丰富的图像、图形及字体渲染（rendering）操作符集合。使用 PostScript，应用程序可以以一种与分辨率无关的方式发出 PostScript 命令，它会定义打印页的外观（或者显示）。

我认为 PostScript 很成功，因为它非常灵活，而且还有一个定义明确的底层成像模型。其他的打印机时间协议试图使用数据结构来静态地定义页面。这些协议在直观地描述一些简单页面时，一直都不成功。

您为什么创建了 PostScript 语言，而不是创建一种数据格式？一台打印机会来解释它吗？一种语言和一个数据格式的实际区别是什么呢？

John：我们在 Xerox PARC（施乐公司帕洛阿尔托研究中心）开始做这个时，当时有一种语言叫做 JaM。那时候我们正在研究图像，而且想要有一种解释型语言，它能很快地运行实验，而且还能接入 Alto 硬件及 Alto 编程接口，而且不是必须经过巨大的编译循环和许多程序尝试、编译、汇编、加载和再尝试来完成实践。我们使用了一种解释型语言，结果表明，它在测试验证大量的新观点方面非常有效。

Charles：随着时间的推移，你能做的就是：如果你决定要在语言中添加一些关键的特性，而它们几乎无法高效地以解释型的方法来运行时，你可以把它们系统整理到这种语言的一组扩展中，并通过引入新的操作符，在一个较低的级别上实现它们，以至于它们可以更加高效率地运行。

事实上，关于这种语言的整体想法是因为我们不知道：未来要控制和描述一个打印页的外观，会面临何种类型的设备、何种环境，而且在某种程度上还有什么新机会。这给了我们一个根本的灵活性，而你肯定不能从数据结构中得到这种灵活性。

它与图灵其他的完备的语言具有相同的运算和控制方法，当然，Church-Turing 的论文发现它无法在数学上证明。

John：在我们的行业中，有很多命令协议的例子不是完整的编程语言。它是我们当时的判断，因为一个协议的使用方式是一直不受限制的和未知的，一个完整的编程语言应该能允许使用我们忘记的或没有预料到的方式来编程。这种“完备性”允许 PostScript 有一个我们无法预期的长期寿命。

397

拼接式语言的好处之一是如果需要系统整理一个新特性，你可以在某个特定平台上以硬件方式来实现它，而且你还可以在旧的平台上以软件的形式来模拟它。如果你定义一个新词，你可以在语言中以其他旧词的形式来定义，这样它就可以在旧机器上运行，不过如果你需要解释器支持这个词，你可以把它添加到更新的版本中。是这样吗？

Charles：是的。

John：我们事实上是这么做的，这样一来，甚至基本的操作符也能被重新定义。在 PostScript 中，你可以重

新定义添加的指令来完成你想要做的事情。实际上，这种灵活性使得 PDF 成为可能，因为我们定义的基本图形操作符可以用来获取它们的操作数堆栈，并作为一个静态的数据结构来发布它，这与保持 PostScript 的可编程性质是截然不同的。

这在一定程度上会对 ROM 中出现 bug 的可能性起作用吗？

John：绝对如此。

Charles：随着它的发布，LaserWriter 拥有了最大数量的已在 ROM 中系统整理烧录的软件。

有 M 字节的一半吗？

John：是的。

这个标准在当时实践过吗？

John：这是把大部分软件烧录到一块掩模 ROM 中（译注1）。如果你有一些 bug，你为自己留了一些更好的安全出口。如果一种编程语言能确实让你带着错误编程，那么它就是非常有用的。

Charles：对于带有这种混合的任何项目，你必须认识到：它以后会有 bug，或者现在就已经有 bug 了。你不要企望幸运之神眷顾并希望这不会真的发生。

我们基本上都是采用这种机制来允许我们修复 bug，因为如果你有数以万计或数以千计的外部打印机，你就不堪忍受每月都弄一套新 ROM。

John：当时，如果全靠掩模 ROM，你根本无法忍受那么多新 ROM。

除了必须要用一些硬件之外，您把硬件当作一个考虑因素了吗？

John：没有。最初的语言出现在一个叫做 Evans & Sutherland 的公司里（译注2）。我们构建这些庞大的图形投影模拟器，而且当时根本没有关于硬件的投影规范，因此我们还必须为此构建数据库。我们必须给自己留出许多后期绑定的空间。后期绑定属性非常重要，因为你并不知道它以后会如何发展，也不知道你的目标机器是什么。

Charles：John，你可以告诉他一点儿有关数据库的东西。它是整个纽约港。

John：它是曼哈顿的所有建筑——不是在曼哈顿的所有建筑，而是曼哈顿、自由女神像的轮廓引导人们把油轮驶入纽约港。它是一个神奇的项目，因为它用了一年左右的时间来完成。它拥有许多数据。

John：它拥有许多数据，特别是必须使用 PDP 15 计算机来处理它们时。相对来说，它们是非常非常非常小的机器。你只有 32K 字节的内存可用。

译注1：掩模 ROM（masked ROM）是制造商为了要大量生产，事先制作一颗有原始数据的 ROM 或 EPROM 当作样本，然后再大量生产与样本一样的 ROM，这一种做为大量生产的 ROM 样本就是掩模 ROM，而烧录在其中的资料永远无法修改。

译注2：PostScript 语言的思想起源于 John Warnock 1976 年在著名的计算机图形公司“Evans & Sutherland”时的想法。当时 John Gaffney 正在开发一个解释纽约港大型三维图形数据库的解释器。Gaffney 设计了非常类似于 Forth 编程语言的 Design System 语言来处理图形。

升级到一台激光打印机看起来好像很奢侈。

John：甚至连激光打印机都是当时 Apple 公司曾经构建的最大处理器。对于它们来说，这就是一个巨大的处理设备。那时，它们已经有了 Mac 图形接口。

我们实际上使用了 PostScript 编程语言构建了一个很好的支持接口，从某种意义来说，它可以接入它们的图形接口，以便我们能够接受它们的数据结构，用来完成与 PostScript 同样的功能，而那些就是 PostScript 程序。

那时，Mac 只有 256KB 或 512KB，当时使用 Mac Draw 和 Write 指令确实有很小的灵活性。LaserWriter 实际上下载了整个程序集来解释它们的命令。

那时 Mac 推出 PostScript 了吗？

John：它正在推出 PostScript，不过那只是 PostScript 的 quick-draw 版本。

Charles：是的；它正在推出 PostScript，不过它通过一组 PostScript 子程序吸收了 Quick Draw，并号称是 PostScript。那些宏或子程序实际上运行在打印机上。

当支持更多的设备时，你们修改 PostScript 了吗？

John：我们遇到了一个困难，那就是我们必须得使用激光打印机。我们最终确实把 Post-Script 移植到到了点阵打印机上。

Charles：这不是最令人满意的项目。

John：没错，它不是。

用二维方式（通过图表）作为考虑语言的基础，那会是什么样子呢？

Charles：人们必须从机制上来设计以支持二维转换，这样就允许程序员以自己的坐标系来编程，但最终要将该空间转换成实际设备的坐标系。

399

John：我认为以二维方式来考虑很容易，当你通过一个子程序把每个二维结构设想为正在绘制（或者图像化）。PostScript 的成功之一是每个对象或对象集都可以包装在自己的坐标系中。这样一来，对象就可以在不同的尺寸和方位的页面实现实例化，而不用考虑内部细节。这种简单的想法使得考虑如何制作一个页面或部分页面变得非常简单。

您是训练有素的数学家。在设计 PostScript 时，这种背景对您有何帮助？

Charles：从对成像模型逻辑转换的理解来看，这一点对我和 John 是明显的：这正是你想要运行它的方式，因为你想要只是固有地构建在成像模型基础上的线性转换机制。这种事跟语言无关。

我不知道它有多大影响，这只不过是一种巧合而已，至少就我的背景来说，不仅有数学基础，而且对整个印刷流程也非常熟悉。我的祖父和父亲都是凸版印刷图像雕版师。我对印刷机的工作原理非常了解，特别在制作网点印版等方面。拥有这样一些背景，会对 PostScript 有所帮助，但我不想说数学是一个主要的促成因素。当你使用抽象定义时，把这些放入你的工具包中，总是非常有帮助的。

您是如何控制自己的思想的呢？您是否有过意见分歧？您是如何发现这些通用解决方案的呢？

Charles：我在 1978 年雇佣了 John，因此我们在一起工作了 30 年。在此期间，我和他从未因彼此生气而造成公司分裂。我们一直对彼此的想法给予充分的尊重，如果我们想法不同，我们会立即设法搞清楚问题所在，弄明白哪一个想法更好，或者想办法将它们合二为一。

在我的记忆中，几乎还从未到过意见分歧的地步。这是一种建立在相互尊重基础之上的独特合作关系。很少有人会在工作生涯中有过这样的经历。很多人在友谊、婚姻或其他地方有过这种经历，不过在工作环境中这是相当罕见的，你可以获得彼此尊重，而且有能力很快地统一你们的想法。

很多人会拿 PostScript 和 Forth 比较，因为它们都是基于堆栈的语言。这有影响吗？

John：没有影响；实际上，当我们在 E&S 公司完成这种语言时，我们是在讨论 Forth 的一些东西，Forth 与此非常类似，不过它们在许多细节方面是非常不同的。因此，实际上没有多大影响。这完全是一种巧合。

Charles：我完全同意。

我一直假定有一些关联，不过现在听起来就像是类似的需求导致了类似的设计。

John：PostScript 酷在你只需要使用很小的程序来实现它，因为它模仿了硬件环境。它非常易于构建基本的体系，然后在你需要它们时再添加操作符即可。

Charles：PostScript 的核心解释器只是很少的几 KB。

John：相当小。

Charles：非常小。

基于堆栈的设计能解决什么问题呢？如果是更直接的描述性语言，PostScript 还能承受吗？

John：PostScript 采用基于堆栈的设计，因此很容易实现，而且也能很快地解释和执行。在引入 PostScript 的机器时代（Motorola 68000s），这种简单和高效是非常重要的。

我认为 PostScript 定义中最为重要的设计决策有以下几点：

- PostScript 是一种拥有变量、条件、循环等的完整的编程语言。
- PostScript 操作符可以在语言本身中被重新定义。这种能力允许我们重新解释现有的 PostScript 文件并生成 Acrobat（PDF）文件。这也允许我们修复烧入只读内存的实现 bug。
- 成像模型允许隔离和图形化地操作可被纳入更大元素之中的子结构。例如，PostScript 允许描述一个页面、按比例缩小它，并使它成为另一个页面的一个组件。任何其他打印机协议中都没有这种灵活性和易用性。
- PostScript 允许用户处理任何图形化元素之类的类型：它能够像任何其他图形化元素一样被伸缩、旋转或转换。这种能力被引入了 PostScript 中。

尽管 PostScript 打印机最初只有黑白色，但 PostScript 的设计预先考虑了使用彩色。运行时堆栈使用有限制吗？

John：我认为执行堆栈被限制在 256 个等级，并且我认为所有那些堆栈都受限于字节长度。

Charles：正在内部运行 LaserWriter 的机器只有一——我只能说在当时它是非常多的，不过它有 1.5M 字节的 RAM，其中 1M 字节用于帧缓冲区。所有的执行都运行在 0.5M 字节的 RAM 上。

401

你已经有 0.5M 字节的 ROM 了吗？

Charles：是的。

John：是的。

如果必要的话，你会把 ROM 映射到 RAM 中，然后再映射到执行代码段中吗？

John：不，你应该做的只是用下载的代码把一个操作符添加到 RAM 中，并取代内置操作符，如我所言，你可以重新定义操作符，或者取代它，也可以添加新的功能。

形式语义怎么样？一些设计者确定了这种语言的语义，然后再验证一个小的核心特性集。

John：我们有一个用来查询名称和符号的词典设备。我们拥有数组和与数字有关的一切。如果你想要更进一步，我们还有非常简单的堆栈机能够接受 256 位操作符，不过这是一种非常简单机器，而且一旦你使用了基本的解释器，它确实非常容易调试。

我认为我们被它的健壮性所折服。我认为我们并没有经历任何形式化的东西。

您打算为人们亲手编写 PostScript 吗？

John：不；很多人都亲手编写了它。你不久之后会习惯于它。我现在更喜欢 Java-Script。

Charles：我们用来展示 PostScript 能力的所有早期的小册子，都是由必须学习程序的设计者来完成的。

我认为一种语言能够健康长寿的其他原因之一要归功于注释。关于解释器的规模，我做得更早一些。它确实很小。难以想象，人们无法度过一个长的咖啡周末，并把它移植到你想要去的地方。

John：这可能要花超过一个周末的时间。

Charles：当你在图形领域处理很复杂的事情时，就会变得稍微有一点复杂。

John：关于 PostScript，还有一些其他事情，而且使它成功的就是它的语言部分，不过另一方面我们还要解决实际问题。字体外形的缩放比例尤其重要。它具有很好的外观，没有人能够做到这一点。

一些人认为那是不可能的。

John：我们也无法确定它的可能性。

402

Charles：是的；这种事就是打赌。我们决定必须这么做，因为另外一种办法就是聘用一大帮人来手工调整位图。这并不符合 PostScript 任意线性转换机制的整体思路，因为只要你稍微旋转一下你的字符，你就不得不做一个全新的位图。

John：要实现这个目标，有几种关键的思路。我们是第一个这么做的。

Charles：我认为，应用 PostScript 非常成功的其他领域是最先进的高质量半色调（half-toning）色彩处理技术。人们在这个领域已经做了大量的工作，但我认为到 20 世纪 80 年代末期为止，人们觉得使用 Adobe 的实现显然比过去使用的机电系统要好。

您会把它归功于您的图像研究背景吗？

Charles：噢，这不仅仅是我们两个。

John：不是的，这不仅仅是我们两个。我们实际上通过模仿真正的机械式半色调完成了第一个半色调，不过后来我们聘请了一些数学家——一个名叫 Steve Schiller 的家伙真正地为半色调做了许多工作，并真正地在一个更基础的层次上来理解它。

Charles：我和 John 曾经都受过有关印刷的很多熏陶。我的父亲和祖父曾经都是图像雕版师。当我把一些早期的东西带回家，我的父亲看了看说，“哦，这个并不是很好”。

后来 Schiller 和其他一些人也参与进来。最终我们得到了我父亲的赞许。那真叫一个酷啊。

您想过位图字体的旋转吗？

Charles：任意的缩放比例和分辨率。随着你达到非常高的分辨率和相对标准的点大小时，这变得不再关键，不过对于激光打印机或一个屏幕来说（但愿上帝保佑不要这样），它就不再适用。

John：这一切的根本想法是人们都设法采用一个字符的典型外形并设法计算出要旋转哪些位。我们采用的不是这种方式。

我们注意到光栅图像的频率，而且我们变换外形图像以恰当地排列位图，然后是旋转明显的位图。这样一来，所有的枝干（stem）都一样了，所有的衬线（serif）也都一样了。黑体也能正常实现，这个点子非常非常简单，不过从来没有人想到过。

Charles：有时候比其他的粗一点或者细一点，你是可以容忍的，除了元素在字符之内是重复的而你的眼睛又能分辨出来之外。

John：很小的区别。

您如何处理字距调整和连字符呢？

John：您从在栅格边界上开始，然后挑选最接近的。如果是字距和复杂的字距，你还是这样做。你沿着光栅边界对齐字符，然后挑选到下一个字符的最近的像素间距。这样它就跟以往任何时候都一样了。连字符只是不同的字符设计而已。

Charles：然后，所有的中文和日文汉字字符也都照此处理。

John：有一个名叫 Bill Paxton 的家伙为我们工作。当我们开始处理中文字符时，不仅笔画之间的空间非常重要，并且小洞和小矩形也很重要——它们不会消失或者过扫。他构建了一个非常复杂的规则集，把一个字符转换为光栅频率，以至于它可以保留字符的所有关键部分。

您必须识别每个字符的关键部分吗？

John：是的，差不多如此。

Charles：但你只是做一次。实际上，设计中文字体的 PostScript 描述显然要比 Roman 更长，只是因为有这么多的字符，不过这样可以一劳永逸。

您能跨字体共享信息吗？

John：你构建一种策略，并且可以使许多东西自动化，并说“就是这种情况。就这样来处理它吧”。

在创建 TrueType 时，它使用了相同的策略，但它对每一个字符都是具体处理的。我们在 PostScript 中使用的策略几乎对整个字母表都是通用的。换句话说，你可以用一个小写字母 h 来区别左边的词干（left stem）和右边的词干(left stem)。现在，左边的词干和右边的词干要靠小 n 来区分，这是一条普遍规则。字符的 x-height（译注3）对整个字体来说都是常量。

我们应该能识别那些属性，然后这些算法应该能处理它。如果你改变了 n 的设计，实际上你并不必做这些。构建 PostScript 字体要比构建 TrueType 字体更容易。

现代打印机可以解释 PostScript 和 PDF，而无须中间转换。这是 PostScript 设计中灵活性和优雅性的好处吗？

John：有一点值得牢记，当 PostScript 第一次被实现时，机器的可用内存还很小。最初的 LaserWriter 拥有 1.5M 字节的内存和 0.5M 字节的掩模只读内存。1M 字节的 RAM 保留用于页面缓冲区，因此我们仅有 0.5M 字节可供所有的工作存储器使用。掩模 ROM 用于 PostScript 的实现。

PostScript 的设计保留的状态非常少。在大多数情况下，没有足够的内存用来保存整个 PostScript 程序。这意味着，打印机读取 PostScript 程序时，程序是按照页面来处理和打印的。使用这种策略，我们就可以完成非常复杂的打印工作而不需要很多内存。

Acrobat 与此不同。每个页的位置都可以在文件的末尾被找到，这意味着可以使用 Acrobat 的打印机必须在读取文件之前启动。对于当今的 GB 内存容量来说，这不是什么问题。

前面已经说过，PostScript 和 PDF 的打印成像模型相同，因此它们是密切相关的。

PostScript 问题域是否存在某些特性，使它难以编写出清晰的 PostScript 程序？

John：在图形领域它已经非常不错了，因为在你处理图形时，堆栈、嵌套转换的方式和递归调用的方式都非常好。你可以画一个图像，然后把它约束起来，并围绕它做转换，它会处理好所有的内部问题，一切都运转正常。

Charles：我禁不住要讲一段来自 Xerox PARC 的小插曲。对非常结构化的 Mesa 语言、Lisp 语言和相当基本的 BCPL（译注4）之间，人们经常会有争论。

我们的一位编程人员提议，人们使用这三种语言为相同的问题编程，由此展开竞争，而我们要根据他的判断来看谁的解决方案最短、最快而且最优雅。

结果表明，我们唯一能搞清楚的是，这完全取决于最聪明的程序员 Bob Sproull 碰巧选用哪种语言，他用 BCPL

译注3：x-height 就是小写 x 的高度。在英文字体的设计中（主要指小写），一个字体的高度体包含三部份，以基准线（baseline）为中央，以上称之为上部（ascender area），基准线内称之为 x 的高度，基准线以下称之为下部（descender area）。所有小写字母中的核心部件都位于在这一黄金地带中，正是 x-height 成为了英文排版中行气贯通的核心，很容易引导视线的流动。

译注4：Basic Combined Programming Language（BCPL）是一种早期的高级语言。1967 年由剑桥大学的 Matin Richards 在同样由剑桥大学开发的 CPL 语言上改进而来。BCPL 最早被用做牛津大学的 OS6 操作系统上面的开发工具。后来通过美国贝尔实验室的改进和推广成为了 Unix 上的常用开发语言。

完成了一切，而不需要任何其他东西。到了最后，程序员的影响可能要比选用特定语言的影响高出很多个数量级。

PostScript Level II 添加了一些特性，比如说垃圾收集等。对于这种语言您可以预见更多的发展吗？

John：在 JavaScript 中，仍然没有能让我打印一个页面的图形接口。但是如果我想要设计类型，我总是要在 PostScript 中编写这些东西，并通过 Distiller 生成 PDF 文件来运行它。是的，没有它我的生命中肯定缺少了另一半。

Charles：如果你是具体地说到 PostScript，我想它肯定无法加速发展。现在要做的是把 PostScript 的图形成像的模型融入到其他环境中，比如说 Flash。

John：是的，对于 Flash 来说能够实现传统的 Adobe 文本引擎的全部文本处理功能。所有的这些东西最终都会归结到语音上来。

405

Apple 使用 PDF 来实现 Mac OS X 桌面的图形化描述。我看，这本质上是使用 PostScript 来完成的。

Charles：最初在 1983 年与 Apple 合作时，我们给了它们许可权利，把 Display PostScript（屏幕显示系统）作为协议的一部分。Steve Jobs 确实想要把它作为合同的一部分。Steve 离开 Apple 后，Apple 决定走自己的发展之路并放弃那种思想，不过 Steve 很清楚他想要的是在显示和打印页上具有相同的成像模型，因此在它们之间绝不会出现中断。当他转向 NeXT 时，我们和他达成了一项交易，Display PostScript 成为 NeXT 计算环境的图形成像模型。

我可以理解，从桌面出版系统的观点来看它是如何工作的，你想要在显示设备之间尽可能地准确。

John：实际上从通用系统的观点来看，一旦我们能够以屏幕显示分辨率调整字体，而且那也是 Bill Paxton 的工作，它就会真正地成为一个功能非常强大的、非常一致的图形模型。

Charles：设想一下：使用 PostScript 成像模型来做 HTML，而不像你现在这样傻乎乎地必须模拟好多东西，那样的 Web 将会何其有趣！

John：这是很有趣的。现在 Adobe-Flash 越来越强，以至于它可以处理字体。这确实是很有趣的。它们在字体上从未如此强大。它们在图形引擎上从未如此强大。Flash 在那些领域正发展得越来越强大。

Charles：实际上，它将会成为我们想象中的 Web Display PostScript。

您认为它会移植到打印机中吗？

John：不会。

Charles：不会。

和打印机有关系吗？

John：几乎没有什么关系。

Charles：我们已经看到现在许多打印机实际上是 PDF 打印机，而不是 PostScript 打印机，这正好说明了又回到计算机中来完成解释工作。

John：但是计算机的规模与以往相比有了不同。

Charles：你可以在计算机上以一分钟 20 页的速度驱动一台喷墨打印机，这个事实能说明很多问题。

16.2 研究和教育

406

Research and Education

自 20 世纪 70 年代以来，在软件和硬件的发展方面，有什么使你感到震惊吗？很多至今仍在使用的观点就是 PARC 在 20 世纪 70 年代提出来的。

Charles：我认为理解 PARC 的来龙去脉是非常重要的。这个故事得从美国第 34 任总统艾森豪威尔（Eisenhower）到第 35 任总统肯尼迪（Kennedy）两届政府政权交接时期说起。

艾森豪威尔把肯尼迪叫到一旁说，美国国防行业一些智者已经建议他，为了继续扩大美国在全球的军事存在，必须从模拟通信转换到数字通信。我并不认为艾森豪威尔总统或当选总统肯尼迪可能会理解这番话的深意，不过肯尼迪对此非常认真。

他把最具技术背景的内阁成员 McNamara 叫到一旁说，艾森豪威尔已经告诉我这些了。我想要你考虑考虑这个问题。你拥有最大的预算。我想要你用足够的钱来启动它，但同时钱数又不要太大，以免国会在我们这个问题上纠缠不休而浪费时间，因为我想迅速、高效地启动它。

随后，McNamara 又选出来自美国麻省理工学院的一个人。他的名字叫 J.C.R.Licklider，曾在美国麻省理工学院的研究实验室呆过，在那里他们开始研究计算技术，不是去计算而是去通信。他看到麻省理工学院的研究者已经同全美其他学术实验室和一些工业研究实验室建立了一些联系，他逐一走访了他们，发现他们中异常聪明的人。他们都呆在美国加州理工学院、加州洛杉矶分校、斯坦福、伯克利、犹他及密歇根这样的地方，当然还有大量位于东海岸的实验室，比如说麻省理工学院、Carnegie Tech，以及其他一些地方等。

他决定花费数百万美元，并以小包的形式把它分配给这十几所大学和几个 Bolt Beranek、Newman 和兰德公司（RAND Corporation）这样的研究实验室，而且他说“我不会微观管理你们。我只是把钱给你们，你可以用它来启动和安排未来几年的研究计划，而且我想要你做的是，如果国会真的过来询问，我们确实能说出来钱用到哪儿了。不过更加重要地是，特别是学术机构，我想要你培训出一批精通这个领域的骨干专家”。

如果你仔细看一下我和 John 的背景，你会发现，我们都是 ARPA（Advanced Research Project Agency of the Defense Department，美国国防部高级研究计划署）的学生。事实上，如果你看一看硅谷的族谱，这个领域的所有公司创始人和资深研究者几乎都曾在某个时期受过 ARPA 的教育，如我所言，你并没有事必躬亲地细致管理这项研究。

这就是人们来到 PARC 的最重要原因，因为 Xerox 所做的事就是受接替 J.C.R.Licklider 的家伙所聘，这个家伙叫 Robert W.Taylor，而且他知道我们去过的所有学校。他把所有这些人招募到了 PARC。PARC 是第一个行业机构，它的成员在先前的十年左右里都受过 ARPA 的教育。将所有这些人聚集在 PARC 这个组织中，他们对这个领域产生了巨大的影响。

407

Charles，**您在 Xerox PARC 建成了成像科学实验室，并在那里指导研究活动。您对如何管理研究小组有何建议？**

Charles：最绝对、最重要的事情就是聘用你可以找到的最聪明的人。我这一生所做的最佳的聘用决策很可能

就是聘用 John 来那个实验室工作，加上该实验室已有的几位也非常有才能的研究者。把这些人作为研究活动的核心基础意味着他们趋向吸引其他高级人才，特别是年轻人拥入，就研究来说，典型的毕业学位。我们能够用这些人构建一个非常强大的团队。

从一开始，我们很多研究就不是局限在实验室内成立的一个组来完成，而是扩展到了 Xerox 的其他部门，在某种程度上甚至是学术研究社区。这种融合也是做研究的一个颇有价值的方式，因为它带来了很多多样化的观点。

您如何判别一名优秀的研究者呢？

Charles：这个没法测试。主要看在这个领域呆过的人对这个领域有何影响。自从 John 从学校毕业后我就听说了他的大名，不过，直到和他会面之前，我从未和他在一起工作过或正式地碰过面。我从他在 Evans and Sutherland 干的工作得知他是一个拥有创造性见解的人，而且还是一个善后者（finisher）（译注5）。

他不是提出建议并把活派给别人，而是亲自实现自己的设想。我发现做研究最可贵的是，提出自己的观点并身体力行来真正地为此开发一个高质量的实现。

或许在纯理论研究和应用研究之间存在区别。

Charles：我并不认为它们之间存在区别。我认为它们遵循相同的准则。在我的职业生涯中我已经发现，是否成为研究者或工程师，除了他们所表现出来的智力技能之外，如果他们能完成自己着手做的事情，肯定会更加高效，而且会产生更大的影响。

您是如何识别什么项目更有前途的呢？

Charles：有一些是同行评议——它是否能吸引同事们的兴趣，并使他们想要加入到活动中来。我记得当我们着手为 Xerox 干一个名叫 InterPress 的项目时，也就是开发 Post-Script 的前身，我们吸引了 Xerox 以外很多人的兴趣，他们都想参与那个计划。

我们有一位来自斯坦福的教授，一位来自卡耐基梅隆的教授，还有一位正在 East Coast 工作的 Xerox 研究人员。确实，这非常有趣。除了 John 和我本人之外，直到我们完成了这个项目为止，为这个项目工作的六个人从来没有在同一个地方工作过。在整个项目设计期间，我们都是使用电子邮件和 ARPAnet 来交换信息的。

我的策略一直是去确定一个大致的方向。人们对实验室的特殊关注有一个理解，然后要靠他们的创造力来决定具体干什么。作为他们的管理者，我的工作是帮助他们做出决定并描述清楚他们想要干什么，然后再给他们提供资源以完成任务。

在你主要开发的业务中，你通常要更多地关注最终要投放市场的东西，而且我们设法制作一组准则，以使得特定的项目能在市场中获得成功。

我问一下有关研究实验室的管理问题，因为我知道您在构建来自 InterPress 的产品时会遇到一些问题。

Charles：研究管理是对的。问题在于没有尝试真正地搞清楚如何从研究中提取思想并在公司的开发部门有效地实现它们，这就是二者之间的真正失误。比起只做研究，这是一个更高层面的管理失败。

译注5：善后者（finisher），指人做事情很勤奋努力，并且很有秩序；为人处世都很认真，对待事情力求完美。

尽管公平地说，我认为我们在研究中如果提出一些伟大的想法，开发人员就会蜂拥而至并实现它们。这又回到了我更早时的评论上。一个真正伟大的研究者——我在年轻时不是这样——必须接受这种思想并想尽一切办法最终实现，如果它们真想获得成功的话。

领导者和管理者之间有什么区别呢？

Charles：领导者需要知道方向，并对如何完成目标有好的想法，而且还必须拥有吸引和激发其他人一起工作完成某一目标的能力。那就是领导者应该做的。

管理者主要关注确保提供各种支撑，比如说预算和沟通模式，以及项目组成员之间发生的其他事情，不过他可能对他们想要干什么没有或几乎没有什么远见。

任何复杂的组织都需要这两类人物，不过如果你混淆了他们之间的区别，通常就会带来灾难性的后果，因为某些领导者也可以是管理者。如果你认为你聘用某人担任领导者而他们实际上主要具备的是管理者的才能，你很可能会对他们的所作所为感到失望。相反地，如果你需要管理一个大组织而你选了一位领导者，他（她）可能会花大多数时间来考虑关于未来的妙策，他们很可能不是非常优秀的管理者。

这是两种不同的能力，二者同样至关重要，而且如果想让某人做某事，你不能将这二者混淆。他们要么是具备领导者的能力，要么是具备管理者的能力，也可能是二者兼备，不过这种情况非常罕见。

您如何判别一名优秀的程序员呢？

Charles：主要是通过和他们一起工作的经历，或在他们旁边，或为他们工作——通过拥有一个积极的工作关系。我不知道如何抽象地来区分。

我已经拥有同很多人在一起工作的宝贵财富。除了 John 之外，还有一个人真正令我印象深刻，你可能从未听说过这个家伙，他的名字叫 Ed Taft。我最初聘用 Ed 是在 1973 年，他为我在 Xerox PARC 工作。Ed 是我有机会共事过的最佳的一位程序员。他会很仔细地描述我们试图实现的问题，而且他非常坚决地要在某个层次上实现它，这样一来，如果你要改变你真正想要如何工作，他就具有了很大的灵活性。

从某种意义来讲，他就像我们谈 PostScript 一样延迟了绑定，不过他是一个善后者。当他拥有一段代码时，他会说干完了，它坚如盘石、非常可靠，他是一个伟大的善后者。从概念层次到最后完成，他从始至终全包下来。如果我能够克隆 Ed，我会这样做的。这应该是一个构建伟大组织的极好机会。

关于计算机科学，大学生应该关注某个特定主题吗？

Charles：说到计算机科学的大学本科教育，我是一个相当保守的人。我认为，你要是一名本科生，你就应该尽可能多地学习物理和数学，并在硕士和博士阶段学习计算机科学，但这只是我的看法。显然，大学正按照学生要求的方向发展，所以我理解他们为什么会授予计算机科学本科学位。

除非你已经具备了强大的数学和科学背景，显然，如果想要进入硬件领域，你需要具备那个背景及化学和物理的一些综合知识。这只是我的特殊偏见。

至于说你的通识教育（general education），我非常强烈地认同基础教育（liberal arts）。如果你无法同他们有效地沟通并说服别人追随你的方向，成为一位拥有伟大思想的科学家又有什么好处呢？你的书写和演讲能力对你的成功绝对是至关重要的。如果你并不具备这些能力，你就实在算不上受过真正的全面教育，而且我认为你会非常低效。

我强烈地鼓励基础教育尽可能将科学和数学连在一起，使之成为本科学位教学大纲的一部分。

别人应该从您的 PostScript 经历中学到什么经验和教训呢？

Charles：PostScript 背后的一个很好的主意就是，你尽可能长地延迟你和显式事物的绑定。换句话说，你要在一种合理的抽象层次上估算、计算并完成你的大部分工作。不到最后一刻，你不要决定在这个光栅图像中打开哪些位，或者是不要决定把你自己绑定到具体的实现算法上。 410

通过处于一个更高的抽象级别上，你能够构建一种你想要生成图像的合理的高级描述，对于整个系列的设备来说，这本来就是非常重要的。它具有与成像模型相同的理念，这使得我们有机会做一些我们早先谈论过的事情，这将会产生相同类型的成像模型，它不仅运行在你个人的计算机上，也会运行在 Web 上，还可以运行在你所拥有的所有数字成像设备上，比如说从电视到电话，再到不同的 Web 设备上等。

这就是 PostScript 的美妙之所在。从你如何描述你想要产生的事物和只在最后一刻把它绑定到具体设备方面来说，你可以在更高级别上运行。

16.3 长寿命接口

设计者如何考虑一种通用编程语言的寿命呢？可以采取什么具体步骤使之长寿吗？

John：许多语言追求解决某个具体问题。记住 Atkinson 开发的 HyperCard（译注6）。他犯的错误，我认为是人们最常犯的错误，而且那不是创建一门完全的编程语言。你必须拥有控制结构，你必须拥有分支结构，你必须拥有循环结构，你必须拥有所有的数学和构成一门完全编程语言所需的一切，否则未来你就会在某些地方碰壁。

人们会来看一看我们并说，“你为什么放入了所有的三角函数？你想要使用它们做什么呢？”——而且它们全都用上了。语言设计中很重要的就是一开始就认识到它会变得很完备。你已经能够访问文件系统。你已经拥有了所有使它完备的东西。我认为这很重要。

PostScript 机器出现了 25 年，今天它仍然在运行，一切依旧，PostScript 仍然在运行。它的功能增强了很多，但基本程序仍在运行。

Charles：我一直在用一个第二代 LaserWriter，因为它拥有我见过最好的手动送进机构。它现在仍然在用。但是 Canon 认为它只能打印 100 000 次，然后打印机就应被抛弃。很显然，他们是超标准设计了它。

给人用的和给机器用的语言在设计上有什么不同？给人用的产品和给机器用的产品呢？因为没人会坐下来并一直重复这个循环，你就认为这么做已经够了，此时你设计它，会想到什么因素？ 411

John：PostScript 的一大缺点是调试非常困难。那是因为你写这些代码时避开了它。当你完成以后，6 个月后再回来，这就有点难办了。而标准中缀符语言，其中并没有很多状态，所以它更易于阅读，也更容易调试。

译注6：HyperCard 是一个优秀的脚本语言解释环境，它所支持的脚本语言称为 HyperTalk。HyperTalk 由 Bill Atkinson 在 Apple 公司工作期间开发，是最接近自然语言的脚本语言之一。

您不必考虑整个堆栈的状态。

John：对。

您在设计 PostScript 时是否强调了二分法？

John：因为很容易增加新的操作符，你会倾向于编写非常非常短的子程序，并在编程实践中尽可能地隔离不同的功能，但是，句法对人类编程并无帮助。

您提到过让 PostScript 程序设计者编写小册子。在 20 世纪 80 年代，你还让行政助理手写 LaTeX。如何教编写程序的设计人员来编写小册子？

John：这是一个有趣的问题。鲜为人知的是，Adobe 公司一直没有告诉任何人，我们的每一款应用程序都有 JavaScript 的基本接口。您可以为 InDesign 编写脚本。您可以为 Photoshop 编写脚本。你可以使用 JavaScript 为 Illustrator 编写脚本。

我一直在写 JavaScript 程序来驱动 Photoshop。正如我所说，这是一个非常鲜为人知的事实，但脚本接口非常完整。在 InDesign 中，如果有人想使用对象模型的话，它们就会给你提供真正的访问机会。

Charles：这是勇敢者的游戏。

有时候甚至使用 JavaScript 为 HTML 文档模型编写脚本也是勇敢者的游戏。

John：不，并不是“有时候”。我一直这样做，而且这一直是勇敢者的游戏，尤其是因为字符集不同，同时这两种环境中的一切几乎都不相同。

但无论如何，人们这么做了。如果你想自动生成文件，最好的办法是进入 JavaScript，并在 InDesign 里面为不可思议的排版引擎编写脚本，或者在 Photoshop 里面为不可思议的成像引擎编写脚本。你可以以相对简单易懂的自动方式做很多事情。

412

这听起来又像是有关长寿的争论。使它更通用化，让人们能够完成所有这些操作和可能性，但允许他们控制编写循环和控制流。

John：这确实是真的。这里有很多项目，我就做了几个。我有一个这样的 90 000 页代码的网站。如果不是自动生成该网站，你根本就不能做出来。它的 HTML 页面太多了。

设计者要求这样做吗，或者有人告诉他们如何做？

John：在维护和扩展 Photoshop 及扩展 InDesign 等方面，例如，大多数人不知道 Bridge（译注7），这是一个程序，用于处理文件结构和 Illustrator、Photo-shop 和 InDesign 之间的图像视图，这一切都用 JavaScript 编写的。

你只是来回转换对象模型。

John：是的。它用到很多 JavaScript，但事实上，它安装在一起是很惊人的。它完全是可移植的。

译注7：Photoshop 的文件浏览器已经被完全重新改造并命名为 Adobe Bridge。Adobe Bridge 是一个能够单独运行的完全独立的应用程序，并且成为了 CS 套装中新的一分子。

从你的观点来看，开发者应该如何看待硬件？是软件在引领创新吗？

Charles：我认为这是一个阴阳平衡的问题。回到过去，当 PARC 出现这种“创新”时，它是在很小的机器上完成的，磁盘存储相对也很小，只有中等水平的网络性能等，而且在这些有限的环境中，在软件方面使用了很多创新，这让使用那些计算机的人们大吃一惊，因为它们从来没有见过这些。

今天，我们所处的环境发生了巨大变化，硬件在不断地迅猛发展，我们可以以低廉的成本获得 GB 级的存储容量，而且我们的处理器速度简直是快如闪电。

我认为很有趣的是，当您摆脱了相对缓慢的性能和相对中等的内存规模限制，从假定它们使用这些方面来说，人们容易变得有点惰性。然后，他们突然遇到一个非常复杂的问题，硬件再次成为限制因素，这时候就要发挥创造力并真正发展软件，以便能解决特定的问题。

在硬件和软件环境之间总有一种平衡。就那些应用程序来说，我认为 Vista 就是一个很好的例子，它的开发就有些滞后，而且它还假定这些机器能够克服固有的低效和某些特定的完成方式。而 Vista 的这些希望最终落空了。

现在，人们又反过头来，并在某种意义上限制了他们对于硬件功能的期望，他们也许会带来一个更好版本的 Windows，当然，是要比 Vista 更好。

现在让一种语言流行会更容易一些吗？

Charles：不，并非如此。我认为，当人们第一次学习如何开发和编程时，他们往往会把软件开发放在以前熟悉的那些环境中来考虑。对他们来说，很难克服经验和新工具之间的这种关联。如果想让一种新的编程语言真正受欢迎，你必须找到这样一种环境：很多人早就开始使用它，而且自身也参与其中。在一定程度上，它更多的是来自教育环境，而不是来自试图将一种新语言推向市场的一个独立组织。

例如，我们现在看到云计算概念的演变，其中计算和访问信息分布非常广泛，它们分布在桌面计算机、互联网、服务器等各式各样的平台之上。

上述云计算的场景我完全可以想象，虽然我在这个领域并没有做任何工作，但是，云计算为开创一种语言提供了机会，使用这种语言，可能会比其他现有语言更能直接处理多种环境。我这样说或许是因为我并不知道这是否是一个语言问题。当然，如果你看一下 Google 和 Microsoft，甚至在一定程度上，Adobe 公司，它们确实是把重点更多地放在了为客户提供该环境的无缝体验上面，我们可能会发现，随着时间的推移，这些工具成为这样做的一种限制。无论是编程语言本身，还是语言的附加物，往往随之而来的一种编程语言——一种编程语言驻留的环境，也需要在今天可用的工具基础上进一步增强。

现在的问题是，没有很多的自然环境。目前已经没有过去那样的 Bell 实验室，或者是 IBM 研究中心，或者是 Xerox 帕洛阿尔托研究中心等。我们不是处在任何类型的这种研发“产业实验室”中。确实也有很多高水平的学术环境，但其中大多数人现在正在进行目的性很强的研究，这些研究项目主要是由美国政府通过 NSF 或 DARPA 资助的。

在伯克利 Unix 的开发环境中，或者是在我做论文时与 William Wulf 一起工作、为完成 BLISS 系统编程而开发的高级语言的那种环境中。当时，依靠 ARPA 的管理方式，我们可以得到基金资助。而在今天，这在研究环境中并不容易做到。对我来说，很难预测这种发展的结局如何。

对于一家公司来说，进行这种投资是非常困难的，因为需要产生收入和利润。除非有一个实体，可以作为一个独立研究机构进行投资。今天的大多数公司，特别是在软件和互联网环境中，在公司内部根本没有这样的实体。

414

或许是一个开放源代码项目？

Charles：也许，但是从我的角度来看，开放源码的问题，如果已经有了一个相当完善的概念、结构也非常完整，而且也很容易理解，那么开放源代码可以做很多贡献。如果只拿一张白纸过来，把它说成是开源并以此开始，我觉得这很难胜利实现目标。

你会建议在任何情况下让它成为开放式标准吗？

Charles：在这个时代，你必须这样做。人们需要将其公开，坦率地说，你需要有一种方式让人们添加他们自己的工具。你可能不知道的是，我只能说 Adobe，因为它对此最熟悉，但 Adobe 的产品每一个都有 JavaScript 风格的开放接口，这样一来，第三方可以为我们的所有产品建立非常复杂的附加功能，包括从 InDesign 到 Photoshop 再到 Acrobat 等，而且是以与平台无关、独立且以脚本化的方式来实现这些功能的。许多公司、个人和团体一直都是采用这种方式。

从为 Photoshop 的内部结构提供 C 代码这个意义上来讲，这不是开放源代码，但它是一种保持核心组成部分完整性的方式，然后给第三方提供了很大的自由度来进行试验和增值。

16.4 标准愿望

在计算机编程或计算机科学领域，要解决的下一个突出问题是什么？

John：嗯，到目前为止，我的书架大概有三四十本跟 Web 有关的手册。它们都非常厚，而且相互矛盾。我很想看到清理掉解决成像模型、清理掉编程环境、清理掉构成网络的所有东西，因为今天实在没有理由要这些东西。

不同的浏览器以及由于使用不同的 HTML 实现带来的可怕场景一去不复返了。

我们正在努力增强 Flash 的功能，并使它足够强健，起码可以让你得到一个与平台无关的语言，而且切换平台再也无须每次都在不同的语义之间转换。

Charles：我完全同意。它是如此令人沮丧，许多年以后了我们仍然处于这么一种环境：有人说，如果你真要想工作，你必须使用 Firefox。我们从现在就想让它成为过去！Web 的普遍性问题将不再有那些区别，但我们仍然需要面对它们。

看看特定的历史场景消亡要花多长时间，这总是令人着迷。您把它们放入永久固定下来的浏览器中，这是非常愚蠢的。

415

您把这个看成是我们今天标准过程的一种失败吗？

John：嗯，标准过程在解决问题方面的作用显然要逊于在系统整理历史方面的作用。

Charles：我个人的态度是，很多标准活动就是在既成事实上再撒一点圣水。正如 John 所言，它实际上不是创建什么。这只是在编制标准及其历史产物。

除非有一个活跃的、充满活力的组织，它拥有标准的所有权，执行或是让这种标准实现得非常容易，进而没

有人愿意自己来做自己的东西，否则你就不会有一个标准。这是我们早期处理 PostScript 遇到的问题。如果有翻版已经从我们这里夺走了 PostScript 的控制权，我们就不会再做 PostScript 3 了。那样它就会变成一套不相容的东西，这会使它的整个前提变得毫无意义。

John：PDF 也是如此。我们最终让美国档案馆采用了 PDF 格式，它是目前这些东西的一个子集，但它至少是一个规范。

当我们做 Acrobat 时，我们说确实需要它，以便这些文件能够真正留存下来，我们签约使他们完全向后兼容，这样一来，很旧很旧的 Acrobat 文件将依旧可供读者阅读。这是一项很重要的工作。Acrobat 的代码非常庞大，但它是用于 Web 的，我甚至不能想象如果没有它，Web 将会怎样。

这些标准应该是由一个主要的实现来推动，还是也可以来自一致性共识？

John：PostScript 是我们的实现，它真正地说明了标准的定义，Acrobat 也是如此。

目前的难题是，如果你使用 Netscape，然后你又使用 Microsoft，Microsoft 无论如何也会倾向于让它们兼容。我认为这是一种悲哀。

Charles：Java 也是如此。

如果我们使用的是 PostScript 而不是 HTML 和 JavaScript，能不能有一个更好的 Web 呢？

Charles：嗯，我们一直在做用于 Web 的新平台，目前的名称叫 Adobe AIR。Adobe Internet Runtime 是一个办法，使得 Web 图像的复杂等级提升到你在我们的应用程序和 PostScript 图像模型的内核中看到的相同水平，将它用于 Web，以至于您可以构建应用程序，无缝地弥合桌面程序和 Web 应用程序之间的差别。我们相信，这是一种从 Web 中获取类 PostScript 的图形和图像的方式，而这种方式 HTML 确实不能有效地支持。

HTML 有两个问题。第一，这基本上是一种面向位图的信息表示。第二，它不是一个标准。我说这不是一个标准是指，如果你把最流行的 Web 浏览器指向一个特定的 HTML 页面，它们都会产生不同的结果。对我来说这是不能接受的，因为这意味着，如果你真的想建立一个复杂的网站，你必须进行浏览器特定的编程，以此来保证无论您使用哪种浏览器都可以得到相同的网页外观。这又回到了过去。这就跟从前一样糟糕。

之所以出现这种情况，是因为 HTML 是一种“开放标准”。我相信，这是一个本质的矛盾。你可以有开放的标准实现，但标准本身需要非常精心地设计和考虑，以至于你不会看到这种差异。

我记得 Sun 首次推出 Java 时的情景。他们最终也与 Microsoft 达成了合作，而 Microsoft 做的第一件事就是去修改 Java。各种实现之间没有统一。很多文件都没有顾及一个标准的整体概念。如果你想有一个标准，它就必须确实是一个标准，每个人都必须遵守它。通常，这意味着要有一个权威组织来维护标准而不是实现。

我们认为有一些真正的机会，使你可以在 Web 上达到一个全新的高水准。我们一直在做这方面的工作。我们已经有了很多第三方利用 AIR 平台做的有趣的应用程序，我们将对此持续关注。它将使你运行本地桌面电脑或 PC 的操作系统和平台更加无关。它并不存在问题。

你可以理解，为什么 Apple 和 Microsoft 认为这在某种程度上是一个挑战，因为他们宁愿你购买他们做的 Web 无缝集成的实现。我们要说这真的没有什么关系。云应该可以供任何计算机访问，而且还能不以 Web 形式提供信息。

第 17 章 417

Eiffel

Eiffel 是一种主要由 Bertrand Meyer 在 1985 年设计的面向对象编程语言，现在由标准组织 Ecma International 管理，该组织于 2006 年制订了 ISO 标准并正式颁布。它提供了很多现在认为很先进并被广泛使用的特性：垃圾收集、泛型编程和类型安全等。它最重要的贡献可能是契约设计（Design by Contract）的理念，通过这种方式，语言可以强制执行接口的前置条件、后置条件和不变量，从而提高了组件的可靠性和可复用性。Eiffel 显然对许多语言都有重大影响，比如 Java、Ruby 和 C # 等。

418

17.1 一个充满灵感的下午

为什么您选择创建一种编程语言呢？

Bertrand Meyer：很少有人是为了创建一种语言而创造。Eiffel 的产生是出于需要。我是为了编写软件才创建一种编程语言的，因为没有我满意的语言工具。

您需要一种工具来进行契约设计吗？

Bertrand：这是显而易见的，但总的来说，我需要一种面向对象的语言。让我给你讲述一下当时的背景。我们于 1985 年创建了互动软件工程公司，现在被称为 Eiffel 软件。我们实际上是要建立软件工程工具。我们由一家日本公司资助来建立一个程序编辑器或面向语法的（syntax-directed，或称语法制导的）编辑器，我们实现了目标并获得了一定的成功。

这是一家很小的公司。当时我仍在美国加州大学圣芭芭拉分校（Santa Barbara）教书，因此这是一份兼职工作。那是 1985 年，我已经使用面向对象方式编写了几乎十年的程序。我很幸运在 20 世纪 70 年代碰上了 Simula 67，它立刻就吸引了我。我知道这是未来编程的方向。

在我们考虑的那类机器上，还没有 Simula 的编译器，我非常喜欢 Simula。正如 Tony Hoare 说 Algol 的那样，这是在很多后继者基础之上的改进。不过，Simula 既没有多重继承，也没有泛型，但那时候我知道同时需要继承和泛型。我在首届 OOPSLA 会议上发表的《泛型与继承（Genericity versus Inheritance）》这篇文章中解释了原因。因此，我们就在考虑有什么可以使用。当时有 C++，所以我打开书本，但很快就合上了，这确实不是我心里想要的那种东西，让它有一点面向对象的风格让 C 程序员认同，这种想法很有意思，但这肯定只能是通向更一致目标的临时跳板。当时也有 Objective-C，但它是很具有面向 Smalltalk 风格的，跟我们所感兴趣的软件工程原理没有多少关系。Smalltalk 本身也是这样。Smalltalk 的发展确实非常棒，但它跟我们的关注没有多大关系。Eiffel 的诞生，是过去十年发展的面向对象技术和软件工程原则的一种结合。Smalltalk 具有很强的实验性语言风格，我们觉得这一点不适合于我们要做的事，例如，没有静态类型已经可怕极了。因此，虽然它有许多令人兴奋的想法，但它跟我们真正想要的没有多大关系。

我当时写了一份报告。这是为加州大学圣芭芭拉分校写的报告，实际上描述的不是语言，而是数据结构和算法库，因为我对复用很感兴趣，并希望有一个标准库涵盖计算机科学的基本数据结构，我有时称之为 Knuthware。后来叫它 EiffelBase。当时被称为 Data Structures Library（数据结构库）。因此我写了这篇论文，以说明数组、链表、栈、队列等；它使用特定的符号。我说我们将要实现它。这就是 Eiffel 的诞生。我认为实现它将需要三个星期。我们一直在做它。不过，这时候已经是 Eiffel 了。

419

该语言本身并不是焦点。焦点是可复用组件，我认识到，要拥有优秀的可复用组件，你就需要类、需要泛型、需要多重继承，需要一个泛型和多重继承精心组合，就像我在 OOPSLA 论文中证明的一样。你需要延迟类.你需要契约，当然，这对于我来说是最简单的事情。我至今仍然不明白为什么人们在编程中不用契约。当然还需要是一个很好的流媒体或序列化的机制，所以这是我们所做的第一件事。事实上，我学习过 SAIL（Stanford Artificial Intelligence Language，斯坦福人工智能语言），这是很好的设计，虽然它不是面向对象的，但它非常有趣，我曾于 10 年前在斯坦福大学使用过。当然，垃圾收集显然也是非常需要的。

您是如何提出这一理念的？是您作为一名实际的程序员的经验足以确定如何改进软件开发的吗？

Bertrand：当然，部分是来自于阅读文献。1973 年，我在斯坦福大学当学生时，阅读了 Dahl、Dijkstra 和 Hoare 编写的《Structured Programming（结构化编程）》[Academic Press]这本书。这本书实际上分三个专题。第一个专题是 Dijkstra 编写的著名的结构化编程。第二个专题是 Hoare 编写的数据结构，它也写得很棒，还有第三个专题。我学到的重要教训之一就是人们只阅读书籍的开始部分，这也给写书的人提了个醒，你必须很小心地写好前 50 页，因为为 90%的人只会读到第 50 页，然后就不往下读了，即使这本书非常好。大多数人阅读了 Dijkstra 写的第一部分：结构化编程。有些人读了 Hoare 的第二部分。我认为，很少有人真正读到底，阅读了 Ole-Johan Dahl 在 Tony Hoare 的帮助下撰写的第三部分：层次化编程结构。它实际上是对 Simula 和面向对象编程的介绍。我是一个勤奋的学生：我被告知要读这本书，而我从就从头到尾读完了它。我喜欢第一部分和第二部分，同时发现最后一部分也很有启发性。

这也是为什么在几年后，当面向对象编程粉墨登场时，很多人认为它是结构化编程的后继，而我却不这么认为的原因。它从一开始就是结构化方法的一部分。结构化编程用于小规模编程（programming- in-the-small）方面，而面向对象编程则是用于大规模编程方面（programming-in-the-large）（译注 1）但这两者之间没有任何差距。在 20 世纪 70 年代中期，当我进入这个行业时，幸运的是我的老板让我购买了昂贵的 Simula 编译器，这个编译器非常好。我用它来开发很多非常有趣的软件。对我来说，很明显没有其他合理的方法来编程，在当时大多数人认为我完全疯了，因为在当时，面向对象编程还没有定型。

在 20 世纪 70 年代中期，刚出校门，我就和朋友 Claude Baudoinhad 一起用法语编写了 “Méthodes de Programmation（编程方法）[Eyrolles]” 这本书，它是一本知识纲要，里面有我在斯坦福和在其他地方学习到的所有知识。这本书非常成功。实际上，它现在还在重印，对于 1978 年首版的图书来说真是太了不起了。我想我可以毫不夸张地说，它用于好几代的法国软件工程师和俄罗斯软件工程师的培训，因为当时它在前苏联被翻译成俄语。它在俄罗斯也非常成功；我去俄罗斯时，仍然会有人告诉我他们用这本书学习过编程。它使用伪代码来解释编程技术、算法以及数据结构。

我把这本书拿给 Tony Hoare 看，他说有兴趣把它翻译成英文，加入他的著名的 Prentice Hall 计算机科学国际系列丛书，我说，“当然可以”，然后，他说：“你能说一些英语，你为什么不自己翻译呢？”我很愚蠢地同意了，而没有坚持找别人来翻译它。这是我在一生当中干过的最大的蠢事：随着我在翻译这本书，我重写了它，因为这是最初出版后三四年了。我已经成熟了，我有了更多的想法。我把这本书叫做《Applied Programming Methodology（应用编程方法学）》，它并没有出版，因为我从来没有写完。我重写了每个句子。这是徒劳的，不过我改进了第 1 版里使用的伪代码。特别是，我觉得如果没有契约的概念，我真的无法正确地表达程序或者算法。Eiffel 的契约概念就出自这里。

另一件事是非常重要的。我是在工业界工作，但我去年在美国加州大学圣芭芭拉分校进行学术休假，因此作为访问者我教授了一些没人愿意教的课程。这是一个连续的课程:130A 和 130B，数据结构和算法，它是一个很有趣的角色，因为它实际上有三个目的。正式的目的是教授数据结构和算法。但有两个没正式说明的目的，那才是真正重要的。一个是使它足够难，以至于很多学生都不能学好，而那些能学好的学生就堪称是计算机科学的学生了。第二个秘密目标是，它应该教给学生 C 语言，因为其他课程中会用到 C。

译注 1：“大规模编程”（programming-in-the-large）和“小规模编程”（programming-in-the-small），这两个术语分别用来描述程序的“大规模”特性和“小规模”特性。“小规模编程”主要研究程序中“代码只有几页长”的组件，例如一个类。而“大规模编程”则是研究如何将“小规模”的组件组合成一个完整的程序——用面向对象的术语来说，就是研究类之间的关系。

这完全是荒谬可笑的，因为尽管 C 是很好的，但它不适合表达算法知识，更不用说教授给学生了。这是一个可怕的经历，因为不是我想在课上教些什么，我基本上是在帮助学生调试他们 C 程序中的野指针等。这教会了我两件事：第一，我再也不想为人教授 C 语言了。C 是一个相当不错的生成编译器的语言，但人类使用它来编程的理念是完全荒谬的。第二，我学到了介绍基本数据结构和算法的唯一途径是使用循环不变量、循环变量、前置和后置条件等。因此，在一定程度上，在年底，我不得不为我们在公司刚刚开始的工作设计一个概念，我使用了曾经在 UCSB 使用过的语言。更一般地说，这是我大量阅读的结果，我沉浸于 Dahl、Dijkstra、Hoare、Wirth、Harlan Mills、David Gries、Barbara Liskov、John Guttag、Jim Horning 等人在软件工程方面多年的先进工作成果之中，而且基本遵循了编程语言的演变过程。对我来说，这是显而易见要做的事情。我认为我可以在某个下午说说 Eiffel 的设计，但实际只用了 15 分钟。

421

是 Eiffel 语言让您产生了契约设计（Design by Contract）的想法吗？

Bertrand：不是，正好反过来。也就是说，这些概念以前早就存在了。语言只是为了反映这一点。对我来说，这是一个没有意义的问题，或者说，这个问题你应该去问那些不使用契约设计的人，应该问问他们为什么。我只是不明白为什么人们会编写软件元素，而没有考虑这种元素如何表达的麻烦问题。这个问题要问 Gosling、Stroustrup、Alan Kay 或者 Hejlsberg。他们怎么能编写软件或者设计一种编程语言而没有提供这种机制呢？我感觉任何两行代码都要用到这个概念。若有人问为何要用到契约设计，就像在问为什么要用阿拉伯数字一样。是那些使用罗马数字进行乘法运算的人应该去自圆其说。

我听说面向对象的语言设计契约的实现要利用里氏代换原则（Liskov Substitution Principle），您认为这是真的吗？

Bertrand：我认为只有你具备完整的可代换性时才是对的。例如，您不能限制继承类型比父类完成的还少。基本上，你必须执行与父类相同的契约。

Bertrand：嗯，我认为这就是在 1985 年引进 Eiffel 的原因，这种思想是在重新定义一个程序时弱化前置条件和强化后置条件。因此，如果里氏代换原则这么说，我想答案是肯定的。但是 Eiffel 是不会等待 Barbara Liskov 的。

我喜欢融合发现的这些概念。

Bertrand：我们部分依赖于 Barbara Liskov 的工作是抽象数据类型，这是从 1974 年开始的开创性工作。当然，这些对在 MIT 中进行的 CLU 的语言工作，也有一定的影响力。不过对我来说，里氏代换原则并没有什么创新性。

422

契约设计对开发团队能有何帮助？

Bertrand：它可以使团队的各个部分了解合作伙伴在做什么，而不必知道他们在如何做。这使您能够获得所有团队的产品“快照”，仅仅是依赖于说明书，而不必受表述的特定选择约束。这对于管理者来说也是很好的。

是否有解决方案过于详细的风险？

Bertrand：其实不会。倒是始终存在着描述不够的风险。人们很少过于详细地描述契约。而过于详细地描述的风险产生于人们过早致力于实现，而没有停留在规范的层次上，但那也没有出现契约，因为他们描述的是意图，而不是实现。基于契约的规范问题与此相反：人们不会说够了，因为它很难把所有的事情都说明白。

我听说，使用契约时，代码不能尝试验证契约的条件。整个思想是代码很难失败。你能解释一下这是为什么吗？

Bertrand：很多人有可能会声称使用契约设计原则，而他们却不够大胆应用此规则。道理很简单。它应用前置条件。如果你对一个程序使用一个前置条件"这是我想要的条件"，那么，程序代码本身不应该检查该契约，也就是说，运行时检查契约的所有责任，假设你怀疑客户端可能会有很多 bug，而且不能保证前提条件的实现，责任在于别人。它在于测试和调试期间的一些自动机制。但是，如果你具有前提条件，并在代码中测试了同样的前提条件，发现有些是错误的，你就是一件事做了两遍，这意味着你不能决定条件、约束是否是客户或者供应商的责任。这是人们是否会采用契约设计而不是一些保守编程的一种真正的考验：他们愿意去除检查吗？其实很少人有胆量这样做。

契约设计有一个非常明确的规则，就是前提是在客户端、调用者身上施加的限制，所以如果有一个不满足前置条件，它就是客户的错。这不是程序的责任。对于后置条件，它就是供应商、程序的责任。如果你有一个前置条件，这意味着调用者的责任，但是随后程序自身会检查它，那么你不必决定，而且实际上你会有有大量的无用代码。当然，这是非常危险的，特别是因为在开发过程中，这个代码通常不会在测试和调试期间运行。这确实是一个有关规范的严重问题。

规范和执行之间的区别有多重要？

Bertrand：这是一个非常好的问题。这种区别是非常重要的，但它是一个相对的区别。这就是说，绝对不可能说：规范中的东西是绝对的，或者说实现当中的什么东西是绝对的。软件的特征之一是，你看到的任何软件元素都是规范要更具体一些，而实现要更抽象一些。

你拿一个听起来绝对像实现的结构，比如说赋值= := A+1 或者 A := B。多数人会认为这是纯粹的实现。但事实上，如果你是一个编译器作者，对你来说，这是一个要扩展成半打机器语言或 C 指令的一些规范。这种区别是很重要的，但实际上创建软件的困难在于足够大的规模，在某种程度上，技术，用来编写实现的技术与用来写规范的技术非常相似。

例如，对编写形式规范的人来说，有一个引人注目的现象：当你的正式规格写得足够大时，你最终做的事情和要问的问题和你编写实际项目时相当接近。因此，区别总是相对的。原因在于在软件中我们不和具体的物理东西打交道。我们从来不处理具体的、有形的、物质性的元素。我们所处理的这一切都是抽象的，因此实现和规范之间的根本区别在一种抽象层次上。因此，要说实现或规范中的某件事情通常是没有意义的。你可以说，X 是 Y 的规范，换句话说，Y 是 X 的一个实现，这是一个有意义的语句，是可证伪的。但"X 是一个规范"或"X 是一个实现"这些语句是不可证伪的。它们没有一个确切的是/否的答案。

编程语言和用它编写出的软件有什么关系？

Bertrand：Eiffel 的一个真正的独特方面，~一个其他人不认为显而易见的甚至真实的是明显的软件特性是，概念和实现之间的链是完全连续的明显特征之一。这就是我们所说的无缝开发，它可能是 Eiffel 最重要的方面。

一切都支持它。例如，正是这种想法，设计和实现之间没有真正的区别。套用一句名言，实现就是使用其他方式进行的设计。仅仅是粒度和抽象级别不同而已。特别是，与实现语言相比，Eiffel 的设计同样也是一种分析、设计语言。总之，它基本上是一种方法，而不是一种语言，但就作为语言的那部分来说，它既用于实现，也用于分析和设计。

很多使用 Eiffel 的人通常不用 UML 之类的工具，这对于 Eiffel 设计者来说在很大程度上是一种噪音，它与什么对软件有用关联很少。Eiffel 是一个旨在帮助您的工具，你在谈论设计，但我认为一个人要从规范和分析的层面开始，然后是设计，最后是实现，以在这个过程中支持你。但是从我的角度或者是 Eiffel 开发者的角度来看，各个任务之间通常没有根本性的差距。

另一点要提到的是语言应尽可能地不引人注目。现在有很多语言，我会称之为“大主教语言”，里面充斥着很多奇怪的符号和约定，你只有了解了这些才能进入内部的核心。例如，今天的很多主流语言基本上到可以回溯到 C 上。它们是在 C 的基础上加加减减的结果，你要掌握它们还得需要了解很多多余的东西。

424 我不能说 Eiffel 完全没有任何累赘，不过确实是很少。Eiffel 的想法是，当你在设计时，你考虑的是设计而不是语言。我曾经听到的最好的溢美之词是，他们在使用该语言时，只需要关注于他们的问题本身。这是语言可以设计产生的最好的影响。

除了语言级处理对象之外，你有没有考虑过其他解决方案？可能是构建了 Unix 系统的小工具之类的组件，并且通过管道来共同组成复杂的功能。毕竟，当你给人写邮件或发信息时，你想要描述什么东西时，你不想使用法语、意大利语、英语等的对象概念。

Bertrand：当然，Unix 的机制非常优雅，但相对于我们要做的建设大型软件的粒度来说，它的粒度太细了。根据我的经验，对象是唯一被证明可扩展用于大系统的可行方法。唯一的其他办法，我考虑过函数式编程，但我认为它的效果不好。这是一个颇具吸引力的主意，非常优雅，有很多可以从中吸取的经验，但在最高层次上，它败给了对象。对象，我应该说类，在描述大规模系统结构来说要有效得多。

这个比喻不是人类的语言，而是数学。它更是类而不是对象。类只不过是对结构概念的编程转换，这些结构概念在数学中表现良好：群、域、环等。也就是说，数学描述的对象和自然对象有很大的不同，比如说，数字与函数，而且，在这两种情况下，你都有相同的结构，由具有相同属性的操作来定义。然后，你将它抽象为一个单一的概念，例如，群、半群或域等。在过去的 200 年里，这个概念在数学里用得非常好。就这方面来说，类或对象不是一个新概念。它们只是数学结构标准概念的直接转换。

很多现代系统都已经实现了组件化，并通过互联网进行传递，您认为语言反映了网络的哪些方面？

Bertrand：这是一个理想的属性，你可以在 Eiffel 中做到这一点，但我也不认为这是 Eiffel 最大的亮点所在。未来几个月内，将会有许多的动态更新和并发特性出现，但现在这些还没有真正出现。是的，我认为这是重要的。我们可以争论它是否应在语言中或在实现中，但一些语言支持是必要的。

您如何看待面向对象语言中的范式和并发呢？

Bertrand：我认为并发是非常重要的。关于这一点，我已经写了很多东西，特别是关于我们开发的并行面向对象编程的所谓 SCOOP 模型。基本上，朴素的方法不起作用。有一种趋势说，“哦，是的，并发，对象，这
425 是同一种思想，它们必须工作得非常漂亮，对象是自然地并行的”，并假定一切到位，这简直是行不通的。如果您尝试将面向对象思想与并发的观点在一个基本层次上结合起来，那基本上是不可能的。

我只想简单谈谈 SCOOP。基本的认识是，标准契约的概念不能在并发上下文中，像在连续上下文中具有同样解释。SCOOP 是一种顺序面向对象编程模型，并以最简单的支持并发的方式来扩展它。这种想法与别人截然不同。例如，如果你对所有的过程进行演算，它们会采取完全相反的方法：问什么是最佳的并发模式，然后再在上面添加其余的程序。这与一般的编程方法完全不同。SCOOP 的想法是，人们使用并行的方式来解决麻烦问题，但也可以使用连续的方式更加高效地来进行，因此 SCOOP 在实现、在模型中隐藏了大部分的并行复杂性。然后，它可以让程序员以并行的方式编程，不过它非常接近于顺序编程，并允许他们保留其惯常的思维模式。

我们应该在哪一层处理并发呢？例如，JVM 几乎透明地管理事物。

Bertrand：显然 Java 线程对于很多应用程序很有用，但那与面向对象概念的联系并不紧密。在 Dijkstra 看来这是一种标志信号。实际上有一个叫做 EiffelThreads 的 Eiffel 库，这或多或少与此相同；我觉得很显然对所有人来说这种解决办法在短期内是好的，但是无法扩展。它仍然会产生很多数据竞争和死锁的可能性。我们的目标应该是自动地防止程序员出现这些问题，这需要在一个更高层次上的表达。正如你指出的那样，这意味着越来越多的工作将通过实现来完成。

17.2 可复用性和泛型

Reusability and Genericity

Eiffel 是如何处理程序的变化和演进的？

Bertrand：利用可复用性、可扩展性自一开始就是一个主要目标。在一定程度上，这就是为什么 Eiffel 是连续的，因为，正如我所说，我们最初设计它是用作内部使用的工具，而不是用来销售给别人的。当开发者开始说，它与以前我们使用语言的很大不同是他们能够更容易地改变主意，而不必为自己的犹豫不决而受到惩罚，实际上，正是这件事让我们重新考虑，并让 Eiffel 的应用范围更加广泛。

我认为这几个方面是至关重要的。首先，在 Eiffel 中，信息隐藏是做得非常认真的，用来隔离不同的模块，并互相展示自己的变化。对于后继者来说没有信息隐藏，因为它没有任何意义，但也有对客户隐藏的信息。举例来说，实在令人震惊的是，在最近的面向对象的语言中，你仍然可以直接指定一个属性，直接指定一个对象域。而你在 Eiffel 中不能这么做，因为它违反了 Eiffel 信息隐藏原则。对于软件来说，这是一个灾难。

那么，继承机制是非常灵活的，它使得你可以打破现有模式来编写软件。泛型也给予语言额外的灵活性。

关于类在语言机制下的缺失使得有可能以非常灵活的方式来组合类。契约还对此有帮助，因为在你修改软件时，重要的是要知道你在修改什么，特别是是否要修改规范的内容，还是只需要修改实现方面的问题。当您更改软件时，必须决定这是否纯粹的内部修改，而不会影响契约，在这种情况下，客户完全不会受到影响，他们是否修改了契约；当然，如果修改了的话，你要看看结果如何。这给你提供了一个控制修改范围的粒度等级。我认为这些都是支持扩展性方面最为根本的一些机制。

开发者应该如何考虑可复用性呢？我之所以提这个，是因为有些受访者说，即使你正在构建类时，也会忘记可复用的部分，因为它需要大量的工作，而且只有当你发现在不同类型的背景中多次使用一个特定的类，你才应该考虑在可复用性上。是否值得花更多的时间使之可复用。

Bertrand：我认为只有两种情况，要不你不善于复用，要不你是一个新手。如果你没有任何复用的经验，当

你试图让软件比当前需求更具一般性时，你会花很多时间，而且可能不会成功。但我会极力主张说，一旦你善于复用，当你有了长期使用其他人的组件经验，以及拥有了自己做的可复用软件时，那你就走对路子了。

我认为很多人失败的原因在于它们不知道可复用性有两方面，其中一个必须要优先于另外一个。一方面是消费者，另一方面是生产者。作为消费者的复用，你只是为自己的应用程序复用现有的软件，许多人这样做主要是为了节省时间。作为生产者的复用，则是让自己的软件更具有可复用性。如果你从一开始就想成为复用生产者，你就会失败，因为生产可复用的软件确实需要专门技术。你要做的就是要花很多时间让你的东西更为通用，但你随后必须猜测应该在什么方向进行“通用化”，通常你会猜错，因为这很难。

但是，如果您具有更谦逊一些的态度，并作为一名消费者开始研究高品质的可复用库，它们是如何产生的，它们是如何设计的，其 API 是什么，那么你可以应用从自己的软件中学到的风格。这也是 Eiffel 世界最常用的方式。人们通过研究 EiffelBase 或 EiffelVision 图形标准库来学习 Eiffel 编程；这些库作为良好的软件模型，具有相当高的质量。如果你研究这些，那么你就能够在自己的软件里运用同样的原则，使其更好，特别是更具可复用性。这条路就是：从一个消费者开始，从您的经验中学习成为一名生产者。根据我的经验，如果你喜欢这样，那么它确实就可以这样。这就是我的《可复用软件（Reusable Software）》[Prentice-Hall]一书潜在的观点。

通过这种方法可以让你的软件实现复用。敏捷编程或极端编程的观点是，你不应该担心复用，复用可是节省时间。我认为只有不擅长复用编程的那些人才担心复用，因为他们还没有遇上麻烦：应该学习如何通过查看良好的模型去产生优秀的可复用软件。

或许这也是他们使用的编程语言中的问题。

Bertrand：当然是，你不必在这一点上再逼我了。Eiffel 旨在从根本上实现三件事。第一的当然是正确性，更普遍地可以说是可靠性。第二是可扩展性，修改软件的容易性。第三就是可复用性。因此，自始至终都有复用。举例来说，实际上很突出的就是我们从一开始就有了泛型类。这绝对是至关重要的复用，但多年以来遭到人们一次又一次的嘲笑。在 1986 年的首届 OOPSLA 上，我们公司有一个展位，上面标明了相关的信息，人们纷纷来到我们的展位取笑“泛型”这个词，他们说，这甚至连一个英文单词都不是。没有人知道它究竟是什么。

然后，几年之后推出了 C++模板。当 Java 于 1995 年出现时还没有泛型，人们都说那没有必要，它会使面向对象编程语言变得复杂。果然，10 年后引入的泛型很复杂，在我看来这并不是完全令人满意的方式，这更多地不是因为这是一个糟糕的设计，而是因为兼容性的限制。然后，当我看到没有泛型的 C#出现时，我简直不敢相信自己的眼睛，但与 Java 一样，7 年后 C#还是增加了泛型。

Eiffel 从一开始就有可复用性动机。在继承机制的具体细节、重命名的特定混合、重定义、取消定义，Eiffel 当中都出现了，重复继承机制，这一切都是由可复用性来证明或者推动的。当然，契约对于可复用性来说是不可或缺的。我刚才说过，我不明白为什么没有人使用契约，但我更不明白：在没有对假定要复用的元素做出明确说明的情况下，人们如何来推测可复用组件。

渐渐地，人们了解这一点。您可能还没有看到它，但是，NET 4.0 已经宣布将包含契约库 Code Contracts 以及所有的基础库；Mscorelib 将会使用契约来重新编写文档和重新设计。它正像是 23 年后的 Eiffel。人们终于理解了没有契约就无法复用。这要花许多时间。当然，在此期间，Eiffel 继续引进新理念，以保持领先地位。人们终于理解了不能没有复用契约。当然，在此期间，Eiffel 将继续推行新概念以保持领先地位。

您何时发现泛型与类一样重要，是如何发现的呢？

428

Bertrand：我认为，这是特定的点或者是特定的实现，更多的是来自学术领域，而不是直接来自工业需要。在我的职业生涯中，大多在工业界，不过我在学术界也做过很多事。有几次，在 1984 年和 1985 年，我在圣芭芭拉的 USCSB 教授“编程语言中的先进概念（Advanced Concepts in Programming Languages）”课程。这是非常 free-ranging 的。

我想看看当时名列前茅的编程语言。于是找到了 Ada 和 Simula，Ada 当时正炙手可热，而 Simula 则是默默无闻，但我知道后者因为其面向对象的概念，必然会成为未来之星。

这个问题变得更加难以避免是我在讲授这门课程时，因为我想用一个星期谈一谈泛型，下一个星期谈谈继承。很自然地就出现了如何比较这两个东西的问题。我也不记得是哪一个学生提出的问题，尽管可能是学生提出来的。不过，我记得问过自己：“这周会是泛型博士，下周会是继承先生吗？”

这使我问自己，“我能使用这个做另一个不能做的是什么呢？”。当然，这是许多讨论和反思以前的结果，不过，编程语言社区分裂成两部分，一部分人认为 Ada 是编程语言的灵活性的要义所在，另一小部分则是发现了面向对象编程和继承的人。

在工作组中这样的讨论很正常，“用我的语言更好。”“不，不，用我的好。”至于我可以说的就是，没有人真正地深入进去，尝试去精确地理解其中的关系，并对这两种机制的表达能力进行过精确的比较。

在我的课程中，我对两者进行了比较分析。然后，OOPSLA 就发来了征稿通知，因此很自然地我就向 OOPSLA 投稿了。

我写下了我所理解的概念，那就是首届 OOPSLA 论文集中的《泛型与继承》这篇文章。

在大多数面向对象的语言都没有意识到这是一个问题之前，您发表了这篇论文。即使 Smalltalk 中没有考虑这种方式。

Bertrand：Smalltalk 从不关心，因为 Smalltalk 使用动态类型，并不需要这些。其实这是非常奇怪的。我的意思是，按照叔本华的话来说，这就像他们首先笑话你，然后他们……

他们首先忽略你，然后他们嘲笑你。是这样的。

Bertrand：其实，在字面意义上是这样的。首届 OOPSLA，因为我的论文署名 USCSB，所以它是一篇真正的学术论文，公司也有一个展台，不过是一种权宜之计，因为我们根本没有什么钱。这个展台的标牌是手工制作的，部分是手写的。

429

人们会来到我们的展台，基本上取笑我们，然后他们会带朋友分享笑。正如我前面提到的，他们的事情之一是泛型。有来自 HP 的人对我说：“你怎么念呢？这必须是法国人或其他的。”他们大声地假装尝试各种发音：“它必须得念成 generisissyty”等。

这确实是那个时代的精神。当然，20 年后，我看到一些参考文章说，泛型是由 Java 发明的。这是生活乐趣的一部分。

17.3 校对语言

我听说你至少会三种自然语言：英文、德语，当然还有法语。使用或者是通晓多种语言，是否会对你作为一种语言设计者产生影响？

Bertrand：简而言之，是这样的。实际上，我的德语并不好。法语是我的母语。英文我说的最好。我的俄语也很流利。实际上我有一个俄罗斯硕士学位，我也能说意大利语。实际上我用俄语演讲没有太大问题。我能用意大利语演讲 15 分钟，然后我的思维会变得非常兴奋。哦，我说得太细了，不过，对于你的问题，答案绝对是肯定的。

我从事计算机科学，部分原因是我对语言的兴趣。我们知道一件事可以有好几种说法，它们并不是一一对应的，你可以采取不同的方法，有时一个名词是正确的解决方案，有时候用一个动词来表达你想要表达的微妙之处，这无疑对我有很大的影响和帮助。我也认为，如果你会至少一门外语，你说自己的母语会更好。

此外，许多技术领域的事情，特别是在编程领域，不仅是编写程序，还要写英文或其他自然语言。稍微练习一下写作技巧，并学习一门或者几门外语都会对你很有帮助。

你是从数学角度还是从语言角度来看待编程的，或者是从两者组合的角度？

Bertrand：我希望能更多地从数学角度来看待编程。我相信 50 年后，编程就会成为数学的一个分支。

有些人长期以来在推广以数学的方式来看待编程。它没有真正流行起来，除了少数特定的可选领域之外，在那些领域中，人们在构建相对小的生命关键系统时确实没有选择。最后，编程是可操作的数学，数学可以用机器来解释。我认为，未来编程将更加是“数学式的”。

至于我自己的方法，我认为是你所说的语言方式、更具自发性、创造性和散漫的论述方式的混合，并努力成为更严格更具数学性的方式。当然，Eiffel 受数学影响比其他现有语言大得多，除了函数式语言，比如 Haskell 之外。

我曾经就“从一个严格的小核心语言开始，并在此基础上构建，比如说 lambda 演算”这个问题，问过很多设计者。一旦你有了函数应用，就可以构建任何可计算系统。您如何看待这种想法？

Bertrand：我不认为它是那么的有帮助。编程问题是科学和工程二者的结合。编程的本质之一是科学的，正如我所说，从根本上是数学。但是，另一边则是工程问题。如果你看看现在的一些程序，它们比人类以前构建的产品都更为复杂。大型操作系统，比如说 Linux 发行版、Vista 系统、Solaris 等，都具有数千万行的代码，通常会超过 5 千万行，这些都是极其复杂的工程结构。它们必须要解决的许多问题主要都是工程问题。

简单来说，科学和工程之间的区别就是在科学中你需要一些非常聪明的想法，在工程中则是需要考虑大量的细节，其中大部分并不是很复杂，但是它们非常之多。这是很少的难题和很多并不是特别困难的问题的并列。有趣的是，在编程过程中这二者都需要。这与我刚才说的几十年后编程将基本属于数学问题，似乎是自相矛盾的，但我认为二者之间并没有矛盾。让我来解释一下。

从根本上说编程是数学范围的事，但重要的是“根本上”这个词。在实践中，编程还涉及许多工程问题，你必须要考虑到这些问题。你在编写一个操作系统时，必须处理成千上万个由朴素的程序员编写的设备驱动程序的问题，并确保它们不会让你的操作系统崩溃。你必须要考虑人们使用的所有的人类语言和对话。您需要

有一个非常复杂的用户界面机制，即使这些基本概念可能很简单，但实际的细节问题会有很多。

编程的困难是双重的：科学的困难和工程的困难。拥有一个非常强大的数学基础，例如，lambda 演算，将帮助你很好地理解前半部分，而不是第二部分。lambda 演算对于 Pascal 或 Lisp 级别的编程语言核心部分建模是非常好的，但更多的现代编程语言的雄心要远远超过了它。

最后，我们必须将问题归结为简单的数学原理，但数学原理本身并不足以应对今天的大规模编程所面临的挑战。

这是学术编程语言和工业编程语言之间的区别吗？

431

Bertrand：当然。在设计 Ada 时，人们批评它是太大、太复杂，而且它的设计者 Jean Ichbiah 在一次接受采访时说的："小语种解决小问题"。这有一定的道理。我认为他基本上回答了人们的批评，就像是 Wirth 那样，他非常痴迷于"小就是美"，但他在很大程度上是对的。

结构化编程和面向对象编程的区别是什么？您刚才提到，您认为结构化编程是编写小程序的一个很好的方法，而面向对象则是一个解决大项目的好办法。它们之间是否存在明确的范围区别？

Bertrand：不，我不会使用面向对象之外的任何其他的编程方法。我基本上同时学的这两种方法，不过，我看不出有任何理由使用任何非面向对象的技术，除了用过就扔的小型脚本程序。"面向对象"只是运用结构的数学概念来编程，对此并没有什么很好的反对之声。

你既有大项目也有小项目。

Bertrand：没有明确的理由不使用类。我不知道 Dijkstra 是如何想的。他从来就没有大力支持面向对象编程，但我也没有听到他批评它，如果不喜欢的话，他就会非常直率地大声说出来。

您提到过，您真的需要为 Eiffel 提供一个流序列化机制。我能问一下为什么需要吗？

Bertrand：我提到过，我们构建的第一个应用程序是 Smart Editor，它的商业化版本是 ArchiText。你使用编辑器需要做的是对一个小的数据结构进行处理，它处在内存之中，然后进行存储。一种方法是每次解析或反解析文本，这不是一种好办法。假设你正在编辑文本，你已经为该文本构建了一个小的结构，而且你有一个抽象的语法树或其他有效的内部表示，你不想将它反解析为文本，然后下次再重新分析它。您想要一直使用抽象结构。当你需要存储它时，你只要按一个按钮，这就是流机制为你提供的功能。

这是第一个应用程序，但以后的应用程序也都像这样。如果您正在编写编译器，也是同样的道理。假设你在编译器中要经历许多遍，每遍都从上一遍中获得数据结构，装饰它，再做一点修改，然后将结果储存到磁盘之上。你不想每次都必须使用一种特定的方式来编写。你只想按一个按钮就能解决。很多应用程序都需要这种东西。

你可以在中间增加更多的阶段。

Bertrand：对。你不必受限于一种特定的处理结构。

您提到了一些所谓的"无缝开发"，并说这是 Eiffel 的根本观点。什么是无缝开发？

432

Bertrand：这是统一软件构建过程的观点：一套问题，一套问题解决方案，然后用一个符号来表达结果，这就是 Eiffel 要做的。

这完全违背了行业在过去 20 年来发展的本质，在这方面我并不赞同。这种倾向已经出现分离，因为这对企业有利，因为人们不得不购买分析工具、设计工具和 IDE（集成开发环境），并在每个层次上都使用顾问。

此外，我想这是因为人们不知道，例如，规格基本上是相同的。从历史上看，编程语言处于非常低的水平，所以你能想到的编程语言都是荒谬的，但我们今天没有任何理由仍然保留这些差距。

人们有逃避麻烦的心态，我们在打卡的时代就已经有了这种心态：您将程序作为批处理作业提交，第二天再回来，如果你有一个编译错误，那你就是遇到了麻烦。你不得不坐下来，非常非常认真地事先思考。

Bertrand：对。事先认真思考没有什么错，但这并不意味着你应该完全使用不同的思考模式，并在不同阶段使用不同的工具、语言等。

人们几乎在道义上看到这种情况。这里有一个隐含的观点是：分析是高尚而伟大的，而实现则是肮脏而卑劣的。在一定程度上，当你使用汇编语言或者是 Fortran 这样的语言编程时，这是对的。Fortran 语言在当时是一个了不起的成就，但不是大多数人都愿意这么想。因此，这种想法认为工作的高尚部分是早期的思考，然后某个时刻有人，不一定是同一个人，来在总体上实现，卷起衣袖，来做这个卑微的工作，就像打开汽车的引擎盖，并把手弄脏。

这在某一点上是对的，但具体到现代编程语言，特别是 Eiffel，并不必是这种方式。不是试图使我们的分析和设计方法更面向实现，我们从另一端开始。我们从编程终端开始，并使得编程语言如此富有表现力，如此优雅，如此接近生产的思维模式，使用它我们可以完成所有的工作。程序的第 1 版是抽象的和描述性的，后来的版本将更具操作性，并将实际运行。它要求过程的不同阶段之间没有任何缝隙。

例如，这跟模型驱动的开发相反，你使用了模型，而后你的程序完全不同。

在这种情况下，您的模型无论是不是可视的，它都类似于源代码，因为它是你生成的，而且源代码是一个事后的东西。

Bertrand：这是好事，例如，只要你能够绝对保证，没有人会接触到源代码，或者是调试或者是修改它。这始终是人们最初说的，“当然，我们只是在模型上工作，没有人会接触到源代码”，但实践往往是完全不同的。

源代码是设计的产物。

Bertrand：问题是你怎么调试？如果你真的调试模型，那么就没什么好批评的了。当然，这意味着你所说的模型只是一个程序。也许这是一个非常高级的程序，但它是一个程序。你已经开发出了一种非常高级的编程语言，然后你要构建一个完整的开发环境。

另一方面，如果你调试的是生成的程序，那么你就会遇到困难。谁负责模型驱动开发是一个关键问题：应该调试哪个版本？对于客户昨天提出的新功能需求，你应该修改哪个版本？

一个人可以证明程序，或者契约只是为了测试吗？

从 Eiffel 开始提出这个契约机制，一直有一个观点，认为从长期来看，契约将用于证明程序，而在短期内它们将用来测试程序。你可以运行时监测契约，然后当违反契约时就会出现一个异常。

下一步，实际上花了很长时间，就是说，让我们以此作为完全自动化测试的基础。这就是我的小组过去几年在 ETH 所做的工作，它现在被集成到了工具集当中。

基本的问题是：测试的困难是什么？

第一，我们要实现测试过程自动化。JUnit 和很多工具已经解决了这些问题，人们现在使用这些工具来实现该过程的自动化。

有两个以上的东西要实现自动化，基本上没有其他的实现了自动化，但它们是最困难的。一个是测试用例生成，因为你需要创建所有的测试用例，可能是数千、数万或者数十万个。第三点也是最后一点，即使你没有前两个问题，你仍然有一个问题，因为你要运行数千例测试，而且还得有人去逐一确定测试成功与否，因此也必须得有自动化测试圣言。我们使用契约可以做的是让测试圣言通过简单地说：如果满足后置条件或者不变量，那么测试就是成功的；如果不满足后置条件或者不变量，那么测试就是失败的。这是自动的。

剩下的是生成测试用例，为此，我们使用的方法似乎是最愚蠢的，而事实上，工作效率却十分出色：随机或准随机的生成。

该工具创建的对象几乎是随机的，然后它调用所有的程序、所有相应类的方法，其中大部分是随机的。然后，我们只需要等待。我们称之为“午饭时测试（Test While You Lunch）”。我们启动按钮开始测试，一小时后吃完午餐再回来，这时候就会看到不满足后置条件。

434

这样方式工作得非常好，因为基本上你没有什么事做。你只是等待自动生成机制来运行你的软件。但是，你只能对内置契约的语言这样做，否则人们将不得不补充契约和契约检查机制。如果你支持契约，它实际上是一个相当出色的测试你软件的方式。

你怎么看待计划的可证明性的想法？它有用吗？它始终是白日做梦吗？

Bertrand：这越来越现实。作为我的学术工作的一部分，它占用了我大量的精力。这是非常令人沮丧的，因为基本的想法已经存在了将近 40 年，主要是自 1969 年 Tony Hoare 的公理化语义论文发表以来。实际的实现非常缓慢，但不是没有进展，这不是白日做梦。

当然，在过去 5 到 10 年中间，已经有了相当大的进展。Microsoft Research 在 Spec#方面的工作是非常有趣的。还有我们在 ETH 对 Eiffel 做的工作，它一直没有注册，主要是因为我们决心很大，因此在我们能够真正让世界刮目相看之前，我们要解决很多问题。但我认为是有希望的。

还有 SPARK 方面的工作。这是一个非常有趣的开发。它们实际上是能够生产被证明的程序的。现在让人为难的是没人愿意使用这种语言来编程。他们称之为一个 Ada 子集。实际上，这是一个带有模块的 Pascal 子集。付出的代价是，你必须放弃所有的人生乐趣。没有类，没有动态对象创建，没有泛型，没有继承，没有指针。使用这种简化的语言，他们能够创建有效的证明工具。这是一个真正的成就，因为它使我们有可能建立重要的系统，通常是军事或航空航天领域的系统，并证明其正确性。现在，你不会想用这种语言来编程。我不会这样，世界上 99%的程序员也不会这样。不过，这确实是一个重大成就。

我们在 ETH 使用我们的证明试图去做的就是类似的工作，不过是为了提供人们想要使用的编程语言。困难在于将我们了解和喜爱的所有编程语言机制都包含进来。当然，举个例子，只要你有指针，就会出现混淆现象，这马上会变成相当复杂的问题。我们面临的挑战不再是证明程序。它是去证明用现代可行的编程语言编写的程序。这将会发生。

人们必须认识到，我们本质上是在处理不可判定的问题。最后，总会有部分程序我们不能证明，而且这也是为什么我们正在做的另一面是测试的原因。开发变得越来越快。这当然是近两三年来我们做得最令人兴奋的事情，而且它现在完全与环境相集成。所以，使用契约的好处之一基本就是你可以完全自动测试。我们有 Eiffel

435

测试框架（Eiffel Testing Framework），现在已经完全与 Eiffel 工作室（Eiffel Studio）集成在了一起，它基本上是按键测试。你不必编写测试用例。你不必编写测试圣言。你只须调用测试框架，创建类的实例，并调用这些类的方法，然后等待契约失败。测试框架的其他酷的方面是测试综合部分：当一个执行失败之后，该工具就会自动从中创建一个测试，您可以重新运行它来协助解决 bug，并作为你的回归测试套件的一部分。

将这个和你的问题联系起来，有很多证明和测试，这两个方面永远都是必要的。但是，证明确实正在成为可能。

证明和契约看起来好像是处于按比例计算的不同点上。

Bertrand：正如我所说，我们从契约动态评估开始，但是一直有一个观点认为，最终的目的是证明类满足它们的契约。人们只是没有为证明做好准备。

我们能预期在随后这几年看到 Eiffel 这样吗？

Bertrand：当然可以。这仍然处于部分研究中，因此很难给出一个确切的期限。测试的研究工作大约在 4 年前开始启动，现在的最初结果是环境的一部分。为了证明，我认为我们会在同样的时间，亦即从现在开始三年后看到初步结果。

好的，您的目标肯定是一个产品吗？

Bertrand：当然了。

您提到了代码的不可证明区域。Haskell 使用了 monad 的概念来区分纯代码和不纯代码，不纯代码是指具有副作用。可能会有类似的机制来隔离无法证明的代码吗？

Bertrand：我们必须要将可以证明的部分与不能证明的部分分开，但我认为我们并不会使用 monad。monad 是一个非常有趣的概念。它可以用于证明其他东西的环境当中：他们通过定义一种支持可证明性的基本语言，然后加入更多先进的语言结构，比如说异常或者其他更渐进的方式，使得一种渐进的方式成为可能。

17.4 管理成长和演进

436

您说过您只用了一个下午就设计出来了 Eiffel，不过实现这种思想用了 20 年的时间。

Bertrand：正如他们所说，关键的想法真的很简单，其余的一切都是评注。在一定程度上，这就是我们做的工作。在过去 20 年里扩大了基本概念。

你使用了类、继承，特别是多继承。你使用了泛型、契约，然后是大量的语言原则，例如，Eiffel 中非常重要的东西，那就是你要提供一种很好的方式来做任何事情。此外，还有高信噪比的概念：即语言不应追求小体积，而是应该在每个特性添加多少功能并带来多少并发难题的基础上进行选择。你有了几十个这样的想法，其中一些是关于语言的想法，其他的则是关于语言设计的元思想，基本上就是这样。但是，把这些转化成编写应用程序有用的东西，比如说模拟美国弹道导弹防御系统或者管理数十亿美元的应用系统，你需要工程部分，这将占用大量的时间。

任何人的资源都是有限的，它跟公司大小无关，因为每一个创新设计都来自于一个小群体。唯一例外的基本上是工程项目。你把一个人发送到月亮上去；噢，这需要数千人干好几年。或者，如人类基因组，你知道怎么做，你只需要我所说的工程的实现。但这是例外。

如果你真的有创新性的思想，我从来没有见过重大突破的软件产品是由 10 个人以上创建的，而一般是介于两个和五个之间。任何人的资源都是有限的，因此，你做出的关键决定就是你什么可以做、什么不可以做。

当然，一直以来，我们也犯过一些错误。例如，我们投资了一个 OS/2 版本，这完全是浪费精力，我们应该投入精力把基本版本做得更好。这些都是你日常必须做的决策，其中一些是错误的。

要用 20 年才达到你对该语言的满意，那么实现、完善和适用性满足您的原始设计目标了吗？

Bertrand：我不会那么说，因为如果没有一定的魄力，你就不会从事这样的事情。在某种程度上说，第一个实现已经准备好了的话，那么都应该已经使用了。人们也可能会更加谦虚，并说我们还没有。

我们正在为此每天努力工作，改进实现，并做一些我们认为是绝对不可缺少的事情。它永远不会是完美的，肯定不会在我的有生之年达到完美，但问题是在任何的特定时间点你在干什么，你的决定是重要的，而且你的决定是一个附件。一个人可以重新选择，而且还怀疑重点正确与否。

毫无疑问，实现的有些方面一直招致人们的批评，它们通常是正确的。另一方面，我们也有已经使用 Eiffel 10 到 15 年的用户，其中一些用户是从它出现起就开始使用了，而且看起来他们对它一直都很满意。

因此一方面，我是从未对实现感到满意；另一方面，我认为，Eiffel 的存在自始至终都为想要使用它的人提供了一个远远领先的杰出的解决方案。

你必须大胆地相信你可以在一个项目上花费 20 年时间，不过你也还必须致力于一个你可以喜欢 20 年的项目。

Bertrand：对。要做什么和不做什么的决策是非常困难的，因为举例来说，我们是第一家推出了 Linux 版本的商业公司。当时，这听起来似乎是一个完全疯狂的决策。以 Linux 与 OS/2 为例。另一方面，有一次有人告诉我，我记得是 1993 年，确实是相当早了，“我们正在使用这个所谓的 Linux。你能提供一个 Linux 版本吗？”公司里面没人想要去做它。我告诉他们，Linux 声称自己是 Unix 的变体——我们的产品覆盖了许多的 Unix 变体，因为当时商业 Unix 有很多变体，我们非常努力地开发一个高度可移植的技术—因此只要尝试在 Linux 下重新编译那些东西，并看看是否可行。而且，如果它需要 1 个月的工作量，那它就不值得。如果只需要一天的工作量，那它可能就是值得的。实际上它根本没有什么工作量。整个 Linux 环境编译就是按一下按钮。要求我实现 Linux 版本的那个家伙感到非常高兴，然后各种请求开始纷至沓来。

从传统的常识角度来看，Linux 是很蠢的，而 OS/2 是聪明的，而事实上恰好相反。拥有 Linux 的早期版本，对我们帮助极大。很难作出这些决策，当然，您拥有的信息很有限，而给你提供建议的人往往都搞不清楚自己在说些什么。

这个教训是，你必须依靠大家的意见，但是最终你必须知道你在做什么，只能依靠你对自己的判断。

您使用什么标准来分析那些决策？

Bertrand：一致性。它是否与该组织的观点一致？是否要引领我们偏离我们的核心观点、能力和愿望，因为你必须要乐于自己的所作所为？或者，这将是一个新的经验，教会我们一些新知识并丰富了我们已经知道的

知识?

今天，您如何选择向 Eiffel 添加哪一个特性呢？您如何发展一种语言呢？

Bertrand：直到 1998 年为止，差不多是 2000 年–2001 年，基本上由我负责该语言的发展演进，后来变成欧洲计算机制造商委员会（Ecma）Eiffel 标准委员会，他们在 2005、2006 年分别制定了一个 Ecma 标准和一个 ISO 标准。要行政式地回答这个问题，语言的修改都要由该委员会来批准。现在的回答更是技术性的，我们一直非常关注 Eiffel 的发展，我们作为委员会和社区做的一些工作是相当有创造性的。

438

首先，我们在语言中有一个原则：做任何事情都应该有一个很好的方式。这是我们必须抵制“蠕变特性（creeping featurism）”最好的方法。这并不是说我们不希望新特性，但我们不希望在语言中增加与现有机制相冗余的新的语言特性。该标准是，如果一个程序员想要做什么，它就需要有一个很好的方法来做它。作为一个反例，有些语言有动态绑定和函数指针数组。作为一个程序员，特别是作为一个面向对象编程新手，如果你根据一个特定的对象类型调用一个不同的函数时遇到了问题，你不知道如何使用这些技术。Eiffel 几乎不会招致这样的困境。原则上不可能达到 100%，但我们非常接近这个数字。

另一个指导原则是使人们所谓的信号信噪比最大化。拿这个与 Niklaus Wirth 对待语言的方法做比较，这是很有趣的。Wirth 确实很喜欢小语言，他有膨胀恐惧症（phobia of bloat）。我认为，对他来说，Eiffel 的很多机制似乎使语言变得很大。我非常欣赏这种观点，但 Eiffel 略有不同。一种语言不一定要为了规模的因素而变小，它更应该是信号的信噪比很高，这意味着噪音很小。我所说的“噪音”是指并不十分有用的语言特性，它只是让语言复杂化了，而没有带来许多表现力，“信号”则代表“表现力”。作为一个例子，我们在大约 12 年前增加“智能代理”的概念，它已被证明非常成功。这是一个闭包的形式，完善的 lambda 表达式，而且有人担心这将与现有的机制冗余，但这并没有发生。这是已被证明深受用户欢迎的重大改进的例子，而且它使得以前我们无法优雅地完成的事情成为可能。所有的都是有用的信号，只有很少的噪声。

第三条原则是确保我们所做的一切都符合 Eiffel 的目标和精神，特别是提高语言的可靠性和降低程序员出错的可能性。在过去的两三年里，的确有一些这样的重大改进。我认为，Eiffel 是第一个使用安全性算法（void-safe）的商业语言。这意味着不再有空指针引用了。这在最新版本 6.4 中全面实现了，同时它的库也进行了全新修改。无效安全（Void safety）是面向对象语言的标准问题，实际上即使是在 C 或 Pascal 中，x.f 也可能崩溃，因为 x 是空的，或者在 Eiffel 术语中是空的。对于 Eiffel 程序员来说，这种风险将不复存在。我认为这是一个重大的成就，因为它消除了面向对象开发仍然存在的主要的潜在运行时问题。这正是我们提升软件开发的可靠性要做的事情。

439

还有更多的原则，我仅举一个为例，那就是标准委员会相当大胆。我们并不是羞于改变语言。特别是，如果我们觉得有更好的方法，我们会毫不犹豫地删除相应的语言机制。当然，我们这样做非常小心，因为这里有一个固定的基础，而且，如果客户拥有数百万行代码，我们不能打破代码，因此通常是旧的机制仍然支撑运行了很多年。我们提供了移植工具和各种辅助工具，但如果在某一时刻，我们得出结论：现在使用某种特定的方式 A，已经找到了一种效果相同的更好的方法 B（在某种意义上是更简单、更安全、扩展性更强的方式，等等），那么我们就会把它删除，并使用我们认为更好的机制来取代它。

您如何处理向上和向下的兼容问题？

Bertrand：这是我们的关注焦点。我想说，如果你在公司中做了一段时间的产品，这很快成为占主导地位的考虑因素之一，它花去了我们不可思议的大量的时间。

如果你不是市场的主导者，这是一个极难回答的问题。如果你是市场的主导者，你就可以按你所好随心所欲。所有的业内大牌偶尔都会这样做。他们基本上会在一夜之间做出改变，客户别无选择，只能追随。

Eiffel 社区在这方面有一点特殊，因为就创新来讲，它比大多数其他社区尤其是语言社区更加开放。人们会接受必须改变的事物，而使用 Eiffel 的人往往是具有前瞻性的，他们对优雅的、创造性的真正优秀的解决方案很感兴趣，他们会接受改变，即使他们有数百万行的代码要管理。他们不喜欢，其他人也没人喜欢的是有人拿枪顶在他们脑袋上，并说："要么现在就改，要么去死"。如果你采用这种改变方式，你可能不会很受您的用户的欢迎。

我们在欧洲计算机制造商委员会当中的策略是设法搞清楚当时情况的所有细节，并考虑到每一个问题。如果我们决定某些东西必须改变，那么我们就会去改变它。我们不会以某种方式已经使用了多年作为借口。如果我们必须改变，那我们就会去改变，而这种改变必须经过非常仔细地计划。我在做简化，因为语言和库变化了，工具也变化了，每一种情况的策略都不必相同，但这些是基本的规则：

- 做好准备工作，并完全相信这是正确的改变。
- 做一个计划。
- 你必须解释，为什么你要进行改变。这是非常重要的。你必须假设正在同聪明人对话。如果你的确做好了准备工作，并仔细地思考了改变的理由，并能向你身边的同事解释并说服他们，那么你也能说服其他的聪明人。
- 给人们一些时间。语言的重大修改过程几乎总是有两个步骤，有时是三个步骤。有一个发布版，其中新机制是可选的，而旧机制则仍然是默认的，但你可以尝试选择一个新机制，通常是在类到类（class-by-class）的基础上，以便你可以在你的部分系统上进行尝试。然后，有一个版本会取消默认机制。

  我们很少会将一些东西完全删除。即使有些遭到了反对，它仍然可以作为一种选择。遗憾的是，不可能总是能做到这一点，因为有时候旧机制和新机制之间会有不兼容现象出现。
- 如果可能的话，应提供转换帮助：我们可以帮助人们找到从旧机制转换到新机制上来的工具、库等。

这种转换已经经历了很多次，尤其是在过去的五六年里。如果你看一看 Eiffel 语言的历史，它有两个重大的变化。第 1 版是 1985~1986 年。第 2 版是 1988 年，但它基本上是往上添加东西，因此并没有造成任何不兼容的问题。然后是 1990 年到 1993 年，我们转移到 Eiffel3 上，这无疑是一个重大变化，但它的好处非常大，而没有引起太大的麻烦。

直到 2001 年开始标准进程，该语言也没有出现很大的变化。标准在 2005 年出版，并在 2006 年成为 ISO 标准。它引入了一些实质性的语言修改，它的实现历时数年，而且该实现已接近完成。目前，我们可能正面临着最困难的问题，这是非常有价值的。它是附加类型机制，解决早期提出的无效安全问题：如果 x 为空则不会执行 x.f 调用的保证。编译器会捕捉这种情况，如果可能导致无效调用的话（一个空指针解引用（dereference）），则会拒绝该程序。但这一机制会引起与现有的代码的不兼容，只要是出于现有代码的好理由，就有可能出现无效调用的情况。该机制基本上是为 6.2 版实现的，而最后的修改被放到了 6.3 版中，整个库转换成了无效安全类型，事实证明这是一个很大的工作量，是用于 6.4 版的。（我们使用了"时钟周期"计划，一年发布两个版本，春季一个，秋季一个。）

现有的代码转换已被证明是一个微妙的问题。我们负担不起弃用现有的代码的代价。我们所能够做的就是告诉用户：如果你想利用这种即将可用的新机制，这就是你必须要做的，我们自己都做过了，因此我们知道它是值得的，而且我们也知道它有多大的工作量。我们将为你提供我们所有的有益经验和工具来帮助你。

人们应该学习你们的什么经验？

Bertrznd：不要盲目跟风时尚并选择正确的解决方案。

后记

441

只有一个词能形容我从这次采访中获得的感受——狂热。

每一位受访嘉宾都会给出你所期望的回报——深层次知识、历史性发现及实践洞察力——不过正是他们对于语言设计、实现与发展的狂热方才显示出了巨大的感染力。

例如，Anders Hejlsberg 和 James Gosling 再次唤起了我对 C#和 Java 的兴趣。Chuck Moore 和 Adin Falkoff 说服我去研究 Forth 和 APL，而这两种语言在我出生前就已经发明出来了。Al Aho 通过描述他的编译器类来诱惑我。我们采访的每个人都给我提供了很多想法，我真希望有时间来研究它们!

承蒙各位帮助，我对此感激不尽，不仅仅是因为你们给予了我和 Federico 时间来采访，还因为你们开辟了许多丰富多彩的创新领域。我从这次经历中获得的最佳经验是：

- 永远不要低估设计或实现简单性的价值。人们可以一直增加复杂性。而大师会力图消除复杂性。
- 充满热情来努力满足你的求知欲。很多最佳的发明创新和发现都是在正确的时间正确的位置追求正确的答案时完成的。

442

- 了解一个领域的过去和现在。每一位受访嘉宾都是和其他聪明的、努力工作的人们一起工作的。

我们的领域取决于这种信息共享。语言可能会持续不断地修改，不过这些宗师们面临的问题仍然会困扰我们——而他们的答案仍然适用。诸如如何维护软件？如何找到一个问题的最佳解决方案？如何令用户惊奇并赞赏？如何在要处理不可避免的修改要求而又不能中断必须继续工作的情况下获得解决方案？

这次采访对这些问题提供了很好的答案。我希望本书在你自己寻找灵感时会对你有所帮助。

—Shane Warden

受访嘉宾

443

444

Alfred V. Aho 是美国哥伦比亚大学 Lawrence Gussman 计算机科学教授。他在 1995 年到 1997 年，并在 2003 年春天担任系主任。

Aho 教授先后获得加拿大多伦多大学工程物理应用科学学士（B.A.S）和美国普林斯顿大学电子工程/计算机科学博士学位。

Aho 教授获得过哥伦比亚大学研究生会 2003 年名师大奖。

Aho 教授是 IEEE 约翰·冯·诺依曼奖章（John von Neumann Medal）得主，而且还是美国国家工程院和美国艺术和科学研究院（American Academy of Arts and Sciences）院士。他获得芬兰赫尔辛基大学和加拿大滑铁卢大学荣誉博士学位，并且是美国科学促进会（American Association for the Advancement of Science）、ACM、Bell 实验室和 IEEE 会员。

Aho 教授因其在算法和数据结构、编程语言、编译器及计算机科学基础等方面的很多论文和图书而著名。他的图书合著者包括 John Hopcroft、Brian Kernighan、Monica Lam、Ravi Sethi、Jeff Ullman 和 Peter Weinberger。

Aho 教授是 AWK 语言中的“A”，AWK 是一种广泛使用的模式匹配语言；“W”是指 Peter Weinberger；“K”是指 Brian Kernighan。很多文献搜索和基因组分析程序使用了 Aho-Corasick 字符串匹配算法。他还编写了 egrep 和 fgrep 的初始版本，它们是最早在 Unix 上出现的字符串模式匹配程序。

Aho 教授目前的研究兴趣包括编程语言、编译器、算法、软件工程和量子计算机等。Aho 教授担任美国计算机学会（ACM）算法和可计算性理论的专门兴趣小组主席，以及国家科学基金的计算机和信息科学及工程理事会的顾问委员会主席。目前，他是 ACM 通讯投稿文章的共同主编。

就任目前的哥伦比亚大学职位之前，Aho 教授是 Bell 实验室计算科学研究中心的副主任，该实验室创建了 UNIX、C 和 C++。他还是技术教员、部门领导及该中心的主管。Aho 教授还是 Bellcore（现在是在 Telcordia）信息科学和技术研究实验室的主任。

Grady Booch 因其在软件体系结构、软件工程和合作开发环境方面的创新工作而蜚声全球。他专心于改进软件开发的艺术性和科学性。自 1981 年 Rational Software Corporation 创建以来，直到 2003 年被 IBM 收购，Grady 一直担任该公司的首席科学家。他现在供职于 IBM Thomas J. Watson 研究中心，担任软件工程首席科学家，在此，他继续致力于编写软件体系结构手册，并且还领导若干超出直接产品范围限制的软件工程项目。Grady 继续致力于解决用户的实际问题，并努力构建与学术界和全球其他研究机构的深层次关系。Grady 是

统一建模语言（UML）的创建者之一，同时也是某些 Rational 产品的最初开发者之一。Grady 还为全球几乎遍布每一个领域的许多复杂软件密集型系统担任架构师和架构顾问。

Grady 是 6 本最佳销售图书的作者，包括《UML Users Guide》和产生重大影响的《Object-Oriented Analysis and Design with Applications》（均由 Addison-Wesley Professional 出版）。他为 IEEE Software 撰写了体系结构专栏。Grady 已经在软件工程方面发表了数百篇文章，包括 20 世纪 80 年代早期发表的提出面向对象设计（OOD）术语和实践的论文，还有 21 世纪初期提出合作开发环境（CDE）术语和实践的论文。

Grady 是美国计算机学会（ACM）成员、American Association for Advancement ofScience（AAAS）会员、社会责任计算机专业人士（CPSR），而且还是电气和电子工程师学会（IEEE）高级会员。他是 IBM 会员、ACM 会员、世界技术网络会员（World Technology Network Fellow）、软件开发论坛远见卓识者（Software Development Forum Visionary），他还是 Dobb 博士杰出编程大奖（Dr. Dobb's Excellence in Programming award）以及三次 Jolt 大奖的获奖者。Grady 是 Hillside Group Agile Alliance 和全球软件架构研究所董事会创始成员，现在还任职于国际软件体系结构协会顾问委员会。另外，Grady 还供职于神学和计算机历史博物馆 Iliff 学校董事会。他还是 IEEE 软件编辑委员会成员。Grady 帮助筹建了计算机历史博物馆，该馆收藏了经典软件和若干杰出人物，如 John Backus、Fred Brooks 以及 Linus Torvalds 等的口述历史。

1977 年，Grady 获得美国空军军官学校学士学位，1979 年，获得加利福尼亚大学圣芭芭拉分校（Santa Barbara）电子工程硕士学位。

Don Chamberlin 和 Ray Boyce 一起共同创立了世界上应用最为广泛的数据库查询语言 SQL。他还是 System R 的管理者之一，这个研究项目产生了第一个 SQL 实现，并开发了 IBM 数据库产品系列的很多底层技术。

Don 也是“Quilt”提案的合著者，现在这个提案已经变成了 XQuery 语言的基础。他在 XQuery 开发期间担任 W3C XML 查询工作组的 IBM 代表，并担任 XQuery 语言规范的编辑。

Don 目前是加利福尼亚大学 Santa Cruz 计算机科学的副教授。他还是 IBM（荣誉）会员、IBM Almaden 研究中心的成员，他已在此工作多年。在过去的 11 年中，他还担任美国计算机学会国际大学编程年度竞赛的评委和命题人。

Don 拥有 Harvey Mudd 大学的工程学士学位，以及斯坦福大学电子工程博士学位。他是 ACM 会员和美国国家工程院院士。因为对关系数据库系统的设计与实现做出了贡献，他还获得了 ACM 软件系统大奖。

Brad Cox 博士目前是 Accenture 公司的首席架构师，擅长为政府和行业客户提供 SOA 安全性、互操作性、标准，以及基于组件的工程等相关服务。

他任职于 George Mason 社会和组织学习计划（PSOL），这是一个聚焦于随着企业向全球化信息密集型经济转变克服改革、开发和学习等障碍的跨学科部门。他的兴趣在于应用因特网、电视和群组技术来促进经验和知识交流。课程包括 Taming the Electronic Frontier、Internet Literacy，以及 Advanced Object Technology 等。

445

他参与编著了《Object-Oriented Programming: An Evolutionary Approach》（Addison-Wesley）一书，通常认为人们现在热衷于对象技术和基于组件的工程要归功于此书。他的第二本书《Superdistribution: Objects As Property on the Electronic Frontier》（Addison-Wesley），提出了一种技术社会解决方案，用于购买、销售和拥有由“数位（bit）”组成的资产，这与自古以来由原子构成物品的概念截然不同。

446

他是 Stepstone Corporatio 的共同创始人，他创建了 Objective-C 编程语言和 Software-IC 库。

在 Schlumberger-Doll Research，他将人工智能、面向对象、Unix 及工作站技术应用到油田管线服务上。

在 ITT 编程技术中心，他应用 Unix 和面向对象技术，支持开发了一个大型的高度分布式电话交换系统：System 1240。

他从芝加哥大学获得博士学位，其间主要在神经网络领域做神经生理学的研究理论和实验工作。他的研究生实验研究是在美国国家健康研究所和伍兹霍尔海洋生物实验室完成的。

Adin D. Falkoff 1941 年获纽约城市大学学士学位；1963 年获耶鲁大学数学硕士学位，在第二次世界大战期间供职于美国海军之前，研究精密光学仪器的高质量制造的材料和方法开发。随后，他为军用飞机设计飞机用天线，1955 年加盟 IBM 之前，在 IBM 研究部门形成期间，他是研究出版的管理者。20 世纪 50 年代后期，他开始致力于计算机科学的不同方面。1960 年，他在 IBM 驻校奖学金项目的资助下进入耶鲁大学，他集中研究计算机科学，包括 APL。他多年担任 IBM 系统研究研究所的客座研究人员，而且还是耶鲁大学计算机科学访问讲师。从 1970 年到 1974 年，Falkoff 先生创建并管理着 IBM 费城科学中心，并且，从 1977 年到 1987 年，他还担任 Thomas J. Watson 研究中心 APL 设计小组的管理者。他因开发 APL 和开发 APL\360 获得 IBM 杰出贡献大奖，而且是因对 APL 的贡献而获得美国计算机学会 Iverson 大奖的第一位获奖者。他的著作或者与人合著的作品包括：《Algorithms for Parallel Search Memories》，《A Formal Description of System 360》，《The Design of APL》，《A Note on Pattern Matching: Where do you find the Empty Vector》，《A Pictorial Format Function》，《Semicolonbracket notation: A hidden resource in APL》，《The IBM Family of APL Systems》等。Falkoff 先生拥有精密光学仪器制造材料和方法以及计算机系统设计方面的多种专利。

Luiz Henrique de Figueiredo 拥有巴西里约热内卢纯粹和应用数学国家研究所 IMPA 数学博士学位，他是该研究所的副研究员和视觉及图形实验室的成员。他还担任 PUC-Rio 计算机图形技术小组 Tecgraf 的几何建模和软件工具顾问，在那里，他帮助创建了 Lua 语言。

除了致力于 Lua 之外，目前他的研究兴趣还包括计算机图形中的计算几何、几何建模及区间算法，特别是仿射计算的应用。

他曾在加拿大滑铁卢大学和巴西科学计算国家实验室做博士后研究工作。他是通用计算机科学杂志编辑委员会成员。

James Gosling 1977 年获得加拿大卡尔加里大学计算机科学学士学位。1983 年，他获得卡耐基梅隆大学计算机科学博士学位。他的论文题目是《约束条件的代数运算（The Algebraic Manipulation of Constraints）》。目前他是 Sun Microsystems 的副总裁兼研究员。他构建了人造卫星数据获取系统、一个多处理器版本的 Unix、若干编译器、邮件系统及窗口管理者。他还构建了一个 WYSIWYG 文本编辑器，一个基于约束条件的绘图编辑器和一个用于 Unix 系统的文本编辑器 Emacs。在 Sun 公司，他的早期工作是担任 NeWS 窗口系统的首席工程师。他对 Java 编程语言进行了最初设计，并实现了它的最初的编译器和虚拟机。他是 Java 实时规范（Real-Time Specification for Java）的撰稿人，而且还是 Sun 实验室的一名研究人员，他的主要兴趣是软件开发工具。他是 Sun 公司 Developer Products Group 的首席技术官，现在是 Sun 公司 Client Software Group 的首席技术官。

Charles (Chuck) Geschke 在 1982 年与别人共同创立了 Adobe Systems Incorporated 公司。在软件行业担任领导超过 35 年之后，2000 年从 Adobe 总裁的职位上退休，并继续与 Adobe 的共同创始人 John Warnock 共同担任董事会主席。

Geschke 积极参与了一些教育性机构、非盈利性组织、技术公司，以及艺术机构的董事会工作。1995 年，他当选美国国家工程院院士。2008 年，他当选为美国艺术和科学研究院院士。他最近完成了旧金山大学理事会主席的任期。他是旧金山交响乐团主管委员会和加利福尼亚协会俱乐部委员会成员。他还供职于卡耐基梅隆

大学计算机科学顾问委员会、Egan Maritime 基金委员会、全国教会管理领袖圆桌会议理事会、Tableau 软件董事会，以及 Nantucket Boys and Girls Club 委员会。

在共同创立 Adobe Systems 公司之前，1980 年 Geschke 在 Xerox 帕洛阿尔托研究中心（PARC）成立了成像科学实验室，在那里他领导计算机科学、图形、图像处理及光学等领域的研究活动。从 1972 年到 1980 年，他担任 Xerox PARC 计算机科学实验室首席科学家和研究员。在 1968 年开始全职研究生学习之前，他是美国俄亥俄州克利夫兰 John Carroll 大学数学系教员。

448

工业和商业机构，包括美国计算机学会（ACM）、电气和电子工程师学会（IEEE）、卡耐基梅隆大学、美国国家计算机图形学会及罗彻斯特技术研究所，都对 Geschke 的技术和管理成就给予很高的评价。他获得地区企业家 1991 年度大奖和国家企业家 2003 年度大奖。2002 年，他当选为计算机历史博物馆会员，并在 2005 年获得硅谷 NCCJ（美国全国基督教徒和犹太教徒联合会）颁发的模范社区领袖大奖。Geschke 获得 2006 年美国电子器件电子学会（AeA）成就奖章。他和 John Warnock 是获得这个大奖的第一位软件领袖。2007 年，他获得 John W. Gardner 领袖大奖。2000 年，Geschke 位列 Graphic Exchange 杂志评出的过去 1000 年 7 位最有影响的图形人之一。

Geschke 拥有美国卡耐基梅隆大学计算机科学博士学位、Xavier 大学数学硕士以及拉丁语学士学位。

Anders Hejlsberg 是 Microsoft 服务器和工具业务部门的技术会员。Anders 被认为是对开发工具和编程语言有影响的创始人之一。他是 C#编程语言的首席设计师和 Microsoft .NET 框架开发的核心参与者。C#编程语言自 2000 年初始发布以来，已经获得了广泛应用，现在已经由 ECMA 和 ISO 实现了标准化。

在 1996 年加盟 Microsoft 之前，Anders 是 Borland International Inc.的第一批雇员。作为首席工程师，他是一种革命性的集成开发环境 Turbo Pascal 的最初作者，并且是其后继者 Delphi 的首席架构师。

Anders 参与编著了由 Addison-Wesley 出版的《The C# Programming Language》一书，而且获得了多项软件专利。2001 年，Anders 是颇具声望的 Dobb 博士杰出编程大奖的获奖者， 2007 年他和他的团队获得 Microsoft 杰出技术成就卓识大奖。Anders 在丹麦技术大学学习过工程。

Paul Hudak 是耶鲁大学计算机科学系教授。他自己 1982 年以来一直供职耶鲁，并在 1999 年到 2005 年间担任系主任。1973 年，他获得 Vanderbilt 大学电子工程学士学位，1974 年获得美国麻省理工学院电子工程和计算机科学硕士学位，并在 1982 年获得犹他大学计算机科学博士学位。

Hudak 教授的研究兴趣集中于编程语言设计、理论和实现。他促进组织和支持 Haskell 委员会，该委员会于 1988 年发布了 Haskell 第 1 版，这是一种纯粹的函数式非严格编程语言。Hudak 是第一个 Haskell 报告的共同编写者，而且还编写了颇受欢迎的语言辅导材料和教程。他的早期工作还包括并行函数式编程、抽象解释及状态声明方式等。

449

最近，Hudak 教授参与了多种应用领域的领域特定语言设计，包括移动仿人机器人、图形和动画片、音乐和音响合成、图形化用户接口以及实时系统等。他还开发了嵌入 Haskell 等语言的技术，包括使用抽象计算模型，比如 monad 和 arrow 模型等。他最近大多关注在用于教育和研究的计算机音乐和音响合成领域中使用 Haskell。

Hudak 教授已经发表了 100 多篇论文，并出版了一本书。他是函数式编程杂志的主编，而且还是 IFIP 函数式编程工作组 2.8 的创始成员。Hudak 教授拥有众多头衔：ACM 会员，IBM Faculty 开发大奖获奖者和美国国家科学基金委总统青年科学家奖获奖者等。

John Hughes 1958 年出生于 North Wales，他作为后来的牛津大学 Christopher Strachey 研究小组的一名程序

员，在学校和大学之间花了一年时间（1974–1975）。在面试期间他帮助 Strachey 安装了一个调制解调器，就是在此 John 被引入了函数式编程领域，并激发了现在还有的巨大热情。在剑桥学习数学期间，他与人合作共同为 GEDANKEN 开发了可能是历史上第一个编译器，完成了 John Reynolds 关于编程语言设计的假想实验。1980 年，他返回牛津大学，开始他的博士研究工作，并在 1983 年完成了函数式语言实现技术的论文。在那里，他结识了自己的妻子 Mary Sheeran，她是同一研究组的一位女研究生。

从 1984 年到 1985 年，John 在瑞典 Gothenburg 的 Chalmers 大学进行了一年的博士后研究，并开展了编译 Haskell 之类惰性语言的开创性工作。他既喜欢这个研究环境，也欣赏瑞典西部的美景。一年结束后，John 暂时返回牛津大学，担任了一名讲师，随后在 1986 年，担任苏格兰格拉斯哥大学教授。

当时，Glasgow 大学扩展势头很强，John 建立了 Glasgow 函数式编程小组，最后此小组成长为全球最佳的小组之一——成员包括 Phil Wadler 和 Simon Peyton Jones。年度研究小组研讨会变得非常著名，而且最终发展成了函数式编程趋势研讨会，并一直持续到现在。

但是，在 1992 年，Chalmers 为 John 提供了一个教授职位，从此他有机会重返瑞典。在那里他继续函数式编程方面的工作，并在 1999 年以后，致力于使用自动工具 QuickCheck 进行软件测试。2006 年他创建了一家销售和开发 QuickCheck 的新兴公司，现在他的一半时间要花在这个公司上。

John 现在是瑞典公民，正在努力学习瑞典语和滑雪——后者是从前途渺茫的零点开始的！他有两个儿子，其中一位双目失明并患有孤僻症。

Roberto Ierusalimschy 是 PUC-Rio（巴西里约热内卢宗座天主教大学）计算机科学副教授，从事编程语言设计和实现。他是 Lua 编程语言的首席架构师，也是《Programming in Lua》一书（Lua.org，现在是第 2 版，并被翻译成中文、韩文和德文）的作者。

Roberto 拥有 PUC-Rio 计算机科学的硕士学位（1986 年）和博士学位（1990 年）。他是加拿大滑铁卢大学（加拿大，1991 年）、美国国际计算机研究中心（美国加利福尼亚，1994 年）、GMD（德国，1997 年）及伊利诺伊大学香槟分校（美国伊利诺伊，2001/2002 年）的访问研究者。作为 PUC-Rio 的一名教授，Roberto 是很多学生的指导老师，后来这些学生成为有影响的 Lua 社区成员。最近，他开发了一个新颖的 Lua 模式匹配包 LPEG。

DR.Ivar Jacobson 1939 年 9 月 2 日生于瑞典 Ystad。（他的全名是 Ivar Hjalmar Jacobson，不过他从未用过中间名）。Jacobson 博士 1962 年在 Gothenburg Chalmers 技术研究所获得电子工程硕士学位。1985 年他在瑞典斯德哥尔摩皇家技术学院以大实时系统的语言结构方面的论文获得博士学位。1983 年至 1984 年，他是美国麻省理工学院函数式编程与数据流体系结构小组的访问科学家。2003 年 5 月 3 日，他被 Chalmers Alumni Association 授予 Gustaf Dalén 奖章。

Ivar 在瑞典建立了 Objectory AB 公司，1995 年，该公司并入 Rational 公司。他使 withRational 存在期间实现了杰出增长，直到它在 2003 年被 IBM 并购。然后他作为雇员，仍然担任了一年多公司执行技术顾问，直到 2004 年 5 月离开 Rational 公司。

除了为 Rational 工作以外，他还有其他有趣的计划。其中之一就是参与 Jaczone AB 公司的工作，Jaczone AB 是他和女儿 Agneta Jacobson 在 2000 年 4 月创建的一家公司。Jaczone 在实现一个古老的梦想——让软件处理流程更主动而不是更被动。主动的处理流程可以实现并辅助开发者完成他们的项目。

Ivar 还认识到软件开发社区在应用软件开发能力方面亟需提高。2004 年，他创建了 Ivar Jacobson International，其目标是促进和帮助项目团队跨全球应用良好的软件开发实践。Ivar Jacobson International 现在已经通过独立

的公司在英国、美国、瑞典、中国、澳大利亚和新加坡等六个国家运营。2007 年，他的新公司并购了 Jaczone，使两家公司合二为一了。

Simon Peyton Jones，M.A.，MBCS，CEng，1980 年毕业于剑桥大学三一学院。在工业界工作两年之后，他在伦敦大学当了七年讲师，并在 1998 年转到 Microsoft Research（剑桥）之前，他在格拉斯哥大学当了 9 年教授。

451 他的主要研究兴趣是函数式编程语言及其实现和应用程序。他领导了一个持续的研究项目，主要关注可同时用于单处理器和并行机的产品级品质的函数式语言系统设计和实现。他对现已成为标准的函数式语言 Haskell 设计做出了关键贡献，而且是广泛使用的 Glasgow Haskell Compiler（GHC）的主要设计者。他编写了两本有关函数式语言实现的教材。

而且，他对语言设计、富类型系统、软件组件体系结构、编译器技术、代码生成、运行时系统、虚拟机以及垃圾收集感兴趣。他特别受到了将基础理论直接用于实用语言设计和实现的激励——这就是他如此喜欢函数式编程的原因之一。

1964 年，***Brian Kernighan*** 获得多伦多大学应用科学学士学位；1969 年获得普林斯顿大学电子工程博士学位。直到 2000 年为止，他供职于 Bell 实验室的计算科学研究中心，目前在普林斯顿大学计算机科学系工作。

他编写了八本图书和一些技术性论文，并拥有 4 项专利。2002 年，他当选为美国国家工程院院士。他的研究领域包括编程语言、工具以及接口，这些使得计算机更容易使用，通常用于非专家用户。他还对为非技术性受众进行技术教育很感兴趣。

1928 年 2 月 22 日，***Thomas E. Kurtz*** 出生于美国伊利诺伊州芝加哥市近郊。他曾就读于伊利诺伊的 Knox 大学，并于 1950 年毕业。随后，他进入普林斯顿大学，并在 1956 年获得数学博士学位。从 1956 年开始，直到 1993 年退休为止，Kurtz 一直供职于达特茅斯大学，讲授统计学、数值分析，最终还讲授计算机科学。1963 年到 1964 年间，他和 John Kemeny（后来成为达特茅斯大学校长）设计了 BASIC 编程语言。借助于分时和个人计算机革命（Time Sharing and Personal Computer Revolutions），BASIC 成为几十年来全球使用最广泛的编程语言。他在 1966 年到 1975 年间，担任达特茅斯 Kiewit 计算中心主任。

他曾任职于多家管理委员会和委员会，并和 Kemeny 博士一起编写了数本编程图书。他从达特茅斯退休后，积极参与了 True BASIC, Incorporated 的工作，开发了 BASIC 计算机语言和其他个人计算机教育软件产品，并进行市场推广。

Tom Love 在华盛顿大学获得了认知科学博士学位，在那里，他研究了成功的计算机程序员的认知特征。Tom 毕业后第一份工作是受聘于通用电气（General Electric）公司，为一款私有的文本搜索引擎（Google 的前身）设计用户接口。几个月后，海军研究办公室联系到他，问他是否对继续他的博士研究感兴趣，由此，他在 GE 成立了软件心理学小组。

452 Tom 被 GE 在 ITT 创建领先的软件研究者小组所吸引。正是在这个小组，Brad Cox 提出并开发了第一个面向对象的 C 语言扩展。ITT 小组还在 1982 年研究社区软件、分布式计算以及交互式开发环境！由于这种经历，Tom 随后在 1982 年成为 Smalltalk 的第一个商业用户。

1983 年，Tom 和 Brad Cox 创建了第一家面向对象的产品公司 Step-stone。在 Step-stone，他们推动对象技术，创造了 Software-IC 的概念，并且将第一个独立的可重用类集 IC-pak 201 推向了市场。还取得了很多其他成就：他们说服 Steve Jobs 使用 Objective-C 作为 NeXT 计算机的系统编程语言（后来成为 Apple 的 OS X 操作系统的基础）。Tom 还出主意并组织创建了 ACM 的 OOPSLA 会议的最初的志愿者小组。

在当了五年的一人顾问之后，Tom 加入了 IBM 咨询（IBM Consulting）项目并创建了 Object Technology Practice（对象技术实践）小组，它是为 IBM 大客户开发大型应用程序项目的一个应用程序开发组织。基于这项成果，他被摩根士丹利（Morgan Stanley）所吸纳，在那里他在 Barings 灾难之前两天提交了一个重新设计的公司信贷风险管理系统。

在 1997 年，Tom 与 John Wooten 博士合作创建了 ShouldersCorp 公司。在 ShouldersCorp，他领导了十几项成功的百日项目，包括最著名的 2001 年完成的敏捷开发（Agile Development）项目。1993 年，他在剑桥大学出版社出版的《Object Lessons》一书中记录了很多使用对象技术的经验。

Bertrand Meyer 是瑞士苏黎世联邦理工学院（ETH Zurich）软件工程教授，同时还是以加利福尼亚圣芭芭拉为基地的 Eiffel Software 的首席架构师。他从事过多种职业，包括软件项目管理者（监督工具和库开发，代码总量达数百万行）、软件架构师、教师、研究者、图书作者和顾问等。

他写作出版了 10 本图书，其中包括若干种畅销图书，比如《Object-Oriented Software Construction（Prentice Hall，Jolt 大奖 1998）》，《Eiffel: The Language, Object Success, and Introduction to the Theory of Programming Languages（Prentice-HALL PTR）》。他的最新图书，是一本使用全部对象技术和契约的入门编程教材《Touch of Class: An Introduction to Programming Well》，由 Springer-Verlag 于 2009 年 3 月出版，它是他 6 年中在 ETH 教授入门编程课程的结晶。

作为一名研究者，他发表过 200 多篇关于软件的论文；他的主要贡献在于软件体系结构和设计（根据契约设计）、编程语言（Eiffel，现为 ISO 标准）、测试和形式方法等领域。目前，他与 ETH 的团队成员合作，主要研究领域是并行和多核体系结构（SCOOP）的安全简易编程、自动测试（AutoTest），程序证明、教学工具（Trucstudio）、计算机科学教育学、开发环境（EiffelStudio，Origo）、重用和基于的组件开发、软件流程以及对象持久性等。

他是 ACM 软件系统大奖（2006）获奖者，而且是第一个 Dahl-Nygaard 对象技术大奖（2005）获得者，他还是 ACM 会员和法国技术科学院成员。

Robin Milner 于 1958 年毕业于剑桥大学。在很短的邮件沟通之后，他于 1973 年加入爱丁堡大学，并于 1986 年与人合作创建了基础计算机科学实验室。1988 年，他当选为英国皇家学会会员，并在 1991 年获得了 ACM 的 AM 图灵奖。1995 年，他重新加入剑桥大学，担任了四年的计算机实验室领导，并于 2001 年退休。他的研究成就（通常是合作成果）包括：LCF 系统，这是后来很多交互式推理系统的基础模型；Standard ML，一种工业级并具有严格基础的编程语言；通信系统演算（CCS）；以及 pi 演算方法等。

目前，他致力于一个用于移动交互式系统的地形模型 Bigraphs。这个模型融合了 Pi 演算方法和移动灰箱（Cardelli 和 Gordon）的强大功能，Pi 演算方法强调移动代理如何能够修改它们的联系，而移动灰箱（译注1）则强调如何将它们移入一个嵌套空间之内。这两个特性的组合把它们视为相互独立："你在哪里并不影响你同谁对话"。这就产生了一种泛型模型，它不仅包含了很多进程演算，而且还有一个目标，那就是为将会在 21 世纪统治计算的普适计算系统设计提供一个严格的平台。

Charles H. Moore 出生于 1938 年，在美国密歇根州长大；获得美国麻省理工学院物理学学士学位；与 Winifred Bellis 结婚，并育有一子 Eric。目前，他居住在漂亮的塔霍湖畔斜坡村（Incline Village）；驾驶一辆斯巴鲁翼豹 WRX；远足环太浩湖（Tahoe Rim Trail）和太平洋山脊径（Pacific Crest Trail）；阅读广泛。他乐于找到简单的解决方案，如果必要的话还会改变问题。

译注1：移动灰箱（Mobile Ambients），是一种描述环境分布和移动计算的形式化模型，即移动环境演算模型。

在 20 世纪 60 年代，他是一名自由职业程序员，直到在 1968 年创建出 Forth 为止。（Forth 是一种简单、高效和多功能的计算机语言，他对此非常骄傲。）他使用它为 NRAO 的望远镜编程。而且，在 1971 年，他与人共同创立了 Forth, Inc，并用它编写其他实时应用程序。

1983 年，饱受硬件限制之苦的他，与人共同创立了 Novix，Inc，并且开始设计 NC4000 微处理器芯片。这个芯片最终装备了 Harris RTX2000，它是空间适用的并在卡西尼号宇宙飞船（译注2）上围绕土星轨道旋转。

作为计算机牛仔（Computer Cowboys），他使用自定义软件来设计 ShBoom、Mup20、F21 和 i21，以及所有的 Forth 架构微处理器。他同样为这些尺寸小、速度快、功耗低的芯片感到骄傲。

在本世纪，他与人共同创立了 IntellaSys 公司，并创建了 colorForth 作为多核芯片的编程设计工具。到 2008 年为止，Intellasys 公司已经生产并向市场推出了 40 核的芯片。目前，他正在将他的设计工具移植到这个令人惊叹的芯片之上。

James Rumbaugh 先后获得美国麻省理工学院物理学学士学位、加州理工学院天文学硕士学位以及麻省理工学院计算机科学博士学位。他的博士学位工作完成于麻省理工学院 Jack Dennis 教授的计算结构小组，该小组在计算的基础模型研究领域领先。他的论文提出了一种数据流计算机的语言和硬件体系结构，这是一个最大的并行计算机体系结构。

他在美国纽约州斯卡奈塔第市的通用电气（General Electric）研发中心工作了 25 年，参与了各种各样的研究项目，包括第一个多处理器操作系统之一、一种 X 光断层摄影术图像重现算法、一个 VLSI 设计系统、一个图形化接口的早期框架以及一种面向对象语言。他和 GE 的同事合作，开发了对象建模技术（OMT），并编写了《Object-Oriented Modeling and Design》（Prentice Hall）一书，推广普及 OMT。他还撰写了六年的广受欢迎的 Journal of Object-Oriented Programming（JOOP）的月度专栏。

1994 年，他加盟位于美国加利福尼亚州库珀蒂诺市的 Rational 软件公司，在此，他和 Grady Booch 结合他们的建模方法，创建了统一建模语言（UML），随后又有 Ivar Jacobson 和来自对象建模小组（OMG）的合作者的投入。OMG 对 UML 进行了标准化，从此，它作为领先的软件建模语言被广泛采用。Rumbaugh、Booch 和 Jacobson 编写图书，向公众推介了 UML。他指导了 UML 的进一步开发，并大力宣传在软件开发中使用良好的工程原理。在 IBM 收购 Rational 之后，他于 2006 年退休。

James 是一位专家级滑雪者、水平一般的高尔夫球手，他还是每周一次的徒步旅行者。他爱好歌剧、戏剧、芭蕾舞以及参观艺术博物馆。他喜欢美食、旅游、摄影、园艺以及外语等。他读书涉猎广泛，包括宇宙学、进化论、认知科学、史诗、神话学、科幻小说、历史以及公共事务等。他和妻子居住在美国加利福尼亚州萨拉托加市。他们有两个儿子在上大学。

Bjarne Stroustrup 设计并实现了 C++。在过去的十年间，C++通过使抽象技术对于主流项目来说既便宜又具有可管理性，已经变成使用最为广泛的支持面向对象编程的语言。使用 C++作为他的工具，Stroustrup 率先在效率优先的应用领域使用面向对象和泛型编程技术；例子包括通用系统编程、交换、仿真、图形、用户接口、嵌入式系统以及科学计算等。C++的影响和它所推广的理念无疑已经远远超出了 C++社区本身。包括 C、C#、Java 以及 Fortran99 等在内的许多语言都提供了为 C++主流使用而准备的特性，就像 COM 和 CORBA 等系统那样。

他编著的《The C++ Programming Language》（Addison-Wesley，1985 年第 1 版，1991 年第 2 版，1997 年第 3

译注2：Cassini（卡西尼号宇宙飞船），是 1997 年发射的土星、土卫六探索飞船，2004 年 7 月进入土星轨道，由轨道飞行器和探测器组成。

版，2000年“特别”版），在同类书中最受欢迎，并被翻译成至少19种语言。后来出版的《The Design and Evolution of C++》（Addison-Wesley，1994年）一书，开启了描述方式的新天地，新书《Programming Principles and Practice Using C++》可以作为介绍编程和C++的入门书。除了六本书之外，Stroustrup还发表了100多篇广受欢迎的学术论文。他积极参与创建了C++的ANSI/ISO标准，并继续进行维护和修订该标准的工作。因为一种编程语言会受到理念、理想、问题以及实践约束条件的影响。

Bjarne出生于丹麦的奥尔胡斯，并在奥尔胡斯大学获得数学和计算机科学硕士学位。他的博士学位工作是在英国剑桥大学进行分布式计算研究。从1979年到2002年，他在美国新泽西州Bell实验室和AT&T实验室担任研究员，后来担任管理者。目前他是美国得克萨斯A&M大学工程学院的计算机科学首席教授。他是美国国家工程院院士、美国计算机学会会员和IEEE会员。他获得过许多专业大奖。

Guido van Rossum 是Python编程语言的创始人，这种语言是Web内外的主要编程语言之一。Python社区指定他作为BDFL（自封的开放源码大佬），这可能是从Monty Python喜剧中取的一个名字（不过实际不是那样）。

Guido在荷兰长大，长期在荷兰阿姆斯特丹的CWI工作，Python即创始于此。1995年，他移居美国、居住在弗吉尼亚州北部，并在此娶妻生子。2003年，Guido全家迁至加利福尼亚州，现在Guido供职于Google公司，他把50%的时间花在了Python开放源代码项目上，其余时间用于Google内部项目的Python事务上。

Philip Wadler 喜欢将理论引入实践，也喜欢将实践引入理论。从理论到实践有两个例子：GJ，它是Sun带有泛型的新版Java的基础，来自于二阶逻辑的量词。他在XQuery上的工作标志着第一次使用数学来制定一个工业标准。从实践到理论的一个例子是：Featherweight Java以不足一页的规则详述了Java的核心。他是Haskell编程语言的主要设计者，主要的创新性贡献有二：类型类和monad。

Wadler是爱丁堡大学理论计算机科学教授。他是皇家社会沃尔夫森研究会会员，是爱丁堡皇家学会会员以及ACM会员。以前，他曾经工作或者学习于Avaya实验室、Bell实验室、格拉斯哥、Chalmers、牛津、卡内基梅隆、Xerox Parc以及斯坦福等大学或研究机构，并作为客座讲授在巴黎、悉尼、以及哥本哈根等地讲学。他位列Citeseers计算机科学最多引用作者名单的第70位，是POPL最有影响论文大奖获得者，担任Journal of Functional Programming（函数式编程杂志）主编，而且还供职于美国计算机学会编程语言专门兴趣小组执行委员会。他的论文包括《Listlessness is better than laziness》，《How to replace failure by a list of successes》以及《Theorems for free》，而且他还是《XQuery from the Experts（Addison-Wesley，2004）和Java Generics and Collections（O’Reilly，2006）》的合著者。他还受邀在从Aizu到苏黎世的世界各地发表演讲。

Larry Wall 曾在不同地方接受过教育，包括康沃尔音乐学校、西雅图青年交响乐团、西雅图太平洋大学、Multnomah圣经学校、SIL国际，加利福尼亚大学伯克利分校，还有加利福尼亚大学洛杉矶分校。尽管他接受的主要是音乐、化学和语言方面的教育，但Larry已经在计算机领域工作了35年以上。

他在编写_rn_、_patch_和Perl编程语言方面最为著名，不过他更喜欢把自己看成是一个文化黑客，其人生使命就是就是给程序员的枯燥生活带来一点乐趣。根据对“工作”的不同定义，Larry还为Seattle Pacific、MusiComedy Northwest、System Development Corporation、Burroughs、Unisys、the NSA、Telos、ConTel、GTE、JPL、NetLabs、Seagate、Tim O’Reilly、Perl Foundation和他自己工作过。目前，Larry受聘于位于美国加利福尼亚州Mountain View的NetLogic Microsystems。为了去工作，他要路过计算机历史博物馆和Googleplex，这也许有什么目的。也可能只是巧合。

John E. Warnock 是Adobe Systems, Inc.董事会的共同主席，这家公司是由他和Charles Geschke在1982年共

同创建的。最初两年，Warnock 博士担任 Adobe 公司董事长，随后 16 年担任 Adobe 公司的主席和 CEO。Warnock 开创性地开发了世界闻名的图形、出版、Web 以及电子文档技术，这些都是出版和可视通信的革命性领域。Warnock 博士拥有六项专利。

Warnock 作为企业家的成功，已被一些国家的最有影响的商业和计算机行业出版物记录下来，而且他已经获得许多技术和管理成就大奖。部分大奖包括：Ernst & Young、Merrill Lynch、Inc. Magazine 的年度企业家；犹他大学杰出男校友大奖；美国计算机学会（ACM）软件系统大奖；美国国家图形学会卓越技术大奖；以及首个美国罗得岛州设计学校艺术与设计杰出服务国际大奖（Rhode Island School of Design Distinguished Service to Art and Design International Award）。

Warnock 博士还获得了美国光学学会 Edwin H. Land 大奖、牛津大学的牛津大学图书馆奖章以及英国计算机学会 Lovelace 奖章。Warnock 是美国国家工程院的著名院士，还是美国艺术和科学研究院院士。他还获得了犹他大学和美国电影研究所的荣誉学位。

Warnock 是 Adobe Systems Inc.、Knight-Ridder、Octavo Corporation、Ebrary Inc.、Mongonet Inc.、Netscape Communications 以及 Salon Media Group 的董事会成员。他是圣何塞创新技术博物馆前主席。他还供职于美国电影研究所理事会和圣丹斯协会董事会。

在共同创立 Adobe Systems 之前，Warnock 是 Xerox Palo Alto 研究中心（PARC）的首席科学家。在加盟 Xerox 之前，Warnock 分别在 Evans & Sutherland Computer Corporation、Computer Sciences Corporation、IBM 和犹他大学担任关键职位。

Warnock 拥有美国犹他大学数学学士学位和硕士学位以及电子工程博士学位。

457

Peter Weinberger 自 2003 年中以来，一直就职于美国纽约 Google 公司，从事处理或者存储大量数据的各种不同项目。

在此之前（从 AT&T 和 Lucent 分拆开始），Peter 在 Renaissance Technologies 工作，这是一家神话般成功的对冲基金公司（他根本未因此而获得好评），他在此开始担任技术领导，负责计算、软件以及信息安全。大约是在此工作的最后一年，他退出了所有这些工作，并致力于一个交易系统（用于抵押担保证券）。

直到 AT&T 和 Lucent 分拆为止，他都在 Murray Hill 的 Bell 实验室从事计算机科学研究。在从事管理之前，Peter 曾经研究过数据库、AWK、网络文件系统、编译、性能和造型，而且很可能还有其他一些 Unix 的东西。随后他悄然进入管理层，他的倒数第二个头衔是信息科学研究副总裁（这是一个只有大公司才会设置的头衔）。Peter 管理着大约三分之一的研究工作，包括数学和统计学、计算机科学以及演讲等。在 AT&T 的最后一年，他把精力放在了 Consumer Long Distance 上面，试图展望未来。

在加入 Bell 实验室工作之前，Peter 在美国安阿伯的密歇根大学讲授数学，发表了大量论文，最后一篇论文写于 2002 年，写这些论文完全是职务需要。

Peter 在美国宾夕法尼亚州斯沃斯莫尔学院获得学士学位，并在加利福尼亚大学伯克利分校获得数学（数论）博士学位。

A

B

C

D

E

F

G

H

J

K

L

M

N

O

P

T

U

V

W

X

Y

Z

关于采访者

ABOUT THE INTERVIEWERS

Federico Biancuzzi 是一位自由职业采访者（freelance interviewer）。他的采访在 ONLamp.com、LinuxDevCenter.com、SecurityFocus.com、NewsForge.com、Linux.com、TheRegister.co.uk 和 ArsTechnica.com 等网站上在线出版，而且还刊登在用波兰语出版的 BSD 杂志及用意大利语出版的 Linux&C 杂志上。

Shane Warden 拥有十余年的免费软件开发经验，其中包括对 Perl 5 核、Perl 6 的设计，以及 Parrot 虚拟机的贡献。他在业余时间经营独立出版商 Onyx Neon Press 的小说分部。他是《The Art of Agile Development（O'Reilly）》的合著者。

[General Information]
书名=编程之魂　与27位编程语言创始人对话
SS号=12572183