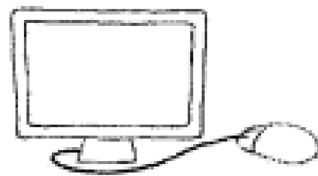
TURING



OUT OF THEIR MINDS

THE LIVES AND DISCOVERIES OF 15 GREAT COMPUTER SCIENTISTS

奇思妙想

15位计算机天才及其重大发现

【美】Dennis E. Shasha Cathy A. Lazere ◎著
向怡宁◎译

人民邮电出版社
北京

PDG

图书在版编目（CIP）数据

奇思妙想 ： 15位计算机天才及其重大发现 / （美）萨莎（Shasha, D. E.），（美）拉瑟（Lazere, C. A.）著 ；向怡宁译. -- 北京 ： 人民邮电出版社，2012. 3
书名原文：Out of Their Minds:The Lives and Discoveries of 15 Great Computer Scientists
ISBN 978-7-115-26881-5

Ⅰ. ①奇… Ⅱ. ①萨… ②拉… ③向… Ⅲ. ①计算机科学－科学家－生平事迹－世界 Ⅳ. ①K816.16

中国版本图书馆CIP数据核字(2011)第248002号

内 容 提 要

本书立足于现场访谈的第一手记录，介绍了15位当代最伟大的计算机科学家，描述了他们的生活历程以及工作成果。在书中，他们解释了自己对科学产生兴趣的缘由，回顾了其成长环境和其他科学家对他们的影响，阐述了各自进行基础探索和发现的途径，同时也分享了对未来的看法和主张。

本书既适合所有程序员阅读，也适合所有对计算机行业和软件开发感兴趣的人阅读。

奇思妙想：15位计算机天才及其重大发现

◆ 著 [美] Dennis E. Shasha Cathy A. Lazere
译 向怡宁
责任编辑 王军花

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京鑫正大印刷有限公司印刷

◆ 开本：800×1000 1/16
印张：13.5
字数：319千字 2012年3月第1版
印数：1－5 000册 2012年3月北京第1次印刷

著作权合同登记号 图字：01-2011-6293号

ISBN 978-7-115-26881-5

定价：32.00元

读者服务热线：(010)51095186转604 印装质量热线：(010)67129223
反盗版热线：(010) 67171154

版 权 声 明

Translation from the English language edition: *Out of Their Minds: The Lives and Discoveries of 15 Great Computer Scientists* by Dennis E. Shasha and Cathy A. Lazere.

Copyright © 1998 Dennis E. Shasha and Cathy A. Lazere. Springer is a part of Springer Science+ Business Media.

All Right Reserved.

本书简体中文版由 Springer-Verlag 授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书的任何部分。

版权所有，侵权必究。

致我们的家人——
他们对本书的完成功不可没。

凯茜·雷泽瑞一家：门罗、穆里尔、艾瑞克、玛利亚、布莱恩和基斯。

丹尼斯·萨沙一家：阿尔弗雷德、哈尼那、卡罗尔、乔、罗伯特、埃伦、卡伦、科洛、泰勒、杰夫、艾瑞阿娜、尼克、乔丹和卡莉。

译 者 序

21 世纪到今天早已进入第二个十年，计算机与我们的生活已经完全无法分割。我还记得上中学时用奔腾 386 玩打字游戏的情景，那时候不管干什么都需要一张或几张 5.25 英寸的 DOS 软盘才行，笨重的显示器屏幕是一个只有 256 色的球面。而现在别说软盘，就连 CD 都已经淘汰过时，计算机上默认的盘符直接以 C 开头，运行速度、显示效果与早期相比更是有天壤之别。流行的终端机型已经变成了便携的笔记本，方寸大小的手机堪比功能强大的个人电脑，其他日新月异的掌上触摸设备更是层出不穷。除了成为消费品外，计算机强大的能力近可帮助我们处理日常工作、记录存档，远能探索太空海洋、破解医学难题。这颗星球上每一秒钟都会产生无数的信息，而它们都在计算机构成的网络中以光的速度穿梭——世界变得更加透明，距离的概念正在消失。

这一切在以前都是无法想象的，而它们又都在短短 20 年中就完成了进化。更妙的是，未来的计算机又会发展成什么样，我们同样无法想象。

那么如果我们追根溯源，又会发现什么？是什么样的人为我们人类作出了如此卓越的贡献？他们是如何与古老的加法器和乘法器进行对话的？我们都知道 C 语言（无论你是否掌握），它的始祖又是谁？互联网的雏形是什么？用电脑模拟人脑这一想法是如何付诸实践的？有心人也许能从历史的倒影中一窥时代变迁的可能方向，更何况这些风流人物的轶事就像一杯杯淡雅的香茗，令人读来不忍释卷，品之齿颊留香。

这些风流人物正是计算机发展史上的拓荒者和奠基人，他们是真正意义上的世界大师，高举科技的火炬和明灯，为广大计算机从业者披荆斩棘、引领方向，留下了无数的里程碑。如今走上神坛的苹果教主史蒂夫 · 乔布斯[①]何尝没有受到过艾伦 · 凯发明的 Dynabook 的影响？倘若阿帕网（ARPANET）未曾诞生，马克 · 扎克伯格又怎能在互联网独领风骚？

《南渡北归》一书的作者岳南曾高叹“大师远去再无大师”[②]。所幸在计算机领域中尚有无数天骄儿女竞相追随前浪。比尔 · 盖茨、乔布斯、扎克伯格、谢尔盖 · 布林和拉里 · 佩奇以及更多的幕后英雄都在将前辈的薪火传递。也许十几年后，类似本书中所谈到的下一代大师将会出现。

---

① 本序写于 9 月初。一个月后惊闻乔布斯去世的消息，不胜唏嘘。特此向编辑提出补加注释。愿这位产品大师和精神领袖在天堂安息。

② 《南渡北归》一书与本书亦有些渊源。书中曾谈到著名哲学家、逻辑学家金岳霖，他的得意弟子王浩又是本书中史提芬 · 古克在哈佛的导师。

翻译这本书对我其实是一次朝圣之旅。在翻译过程中随这些大师一起喜笑怒骂，体验他们在研究生涯中的酸甜苦辣。这些传奇人物对课题孜孜求索的态度、坚韧不拔的精神，他们甘于冒险、勇于面对嘲笑的不羁和洒脱都给我留下了深刻的印象。恰值我32岁生日的今天，胡乱挥就此文，是为序。[1]

向怡宁

2011年9月6日晚于湖北武汉

（编者说明：为便于读者理解有关背景知识，译者不辞辛劳，为译文增加了许多译注。书中脚注除特别说明外，均为译者注。）

① 就在本书即将送印之时，又闻噩耗，人工智能之父，Lisp语言发明人麦卡锡于2011年10月24日与世长辞，享年84岁。特再次补加注释。

前　　言

在许多科学领域中，开创先河的大师们都生活在遥远的过去。要想揭示他们真正的丰功伟绩及其前因后果，我们只能深入历史的角落，在尘埃中寻求真相，从传说中分辨事实。牛顿是否真的被苹果砸中？阿基米德是否高呼着Eureka[1]从浴缸跳出跑到大街上？欧几里得是否曾经从古埃及祭司那里窃取了几何的秘密[2]？

计算机科学则与此不同。那些最先研究当前计算方法（computation）的数学家们，例如阿兰·图灵[3]、埃米尔·波斯特[4]和阿隆佐·邱奇[5]，都起步于20世纪30年代或40年代。我们今天的计算机的概念和他们的早期构想相差无几：一个计算引擎（calculating engine）和一个用于存储指令和数据的存储器（memory）。他们提出了该领域的基本理论问题：在有限的时间里能够计算什么？对于这些数学家来说，“computer”这个词既可能指机器，也可能是指人类本身——因为在他们开始建立相关理论时，世上还没有能够存储指令的计算机器。

到了20世纪40年代末，当计算机成为工程上的现实之后，该领域的重要问题开始转变到实用主义方向。仅仅知道一个问题最终能被解决是不够的，更重要的是知道如何高效且优雅地（如果可能的话）解决这个问题。这需要我们为真实的机器编写指令，需要我们努力寻求更有效的解决方案，需要构建更好的计算机，以便解决更复杂的问题，此外，有时还需要让计算机也参与到创造过程中来。本书分为四个部分，它们分别对应了过去数十年来计算机科学家们为之奋斗的4个基本问题。

(1) 语言大师：应该怎样与机器交流？

(2) 算法大师：如何能让计算机更快地解决问题？

---

① Eureka来自古希腊语，意为“我想到了”。据传，阿基米德在浸浴时看到浴缸溢出的水与自身的体积相等，茅塞顿开，从而证明了皇帝的金冠被掺入了杂质，并由此推导出著名的浮力定律。

② 由于洪水经常淹没土地，古埃及人总是需要重新丈量土地以便计算地界和租税，由此发展出了简单的度量几何学。欧几里得在埃及居住时，将各种几何定理、命题和求证按照逻辑进行排列组合，形成完整的体系，并以清晰简练的方式表述出来，编成《几何原本》（*Euclid's Elements*）一书。

③ 阿兰·图灵（Alan Turing，1912—1954）是英国数学家、逻辑学家，被视为计算机科学之父。他对人工智能的发展有诸多贡献，此外还是一位世界级的长跑运动员。1954年因食用浸过氰化物溶液的苹果死亡。

④ 埃米尔·波斯特（Emil Leon Post，1897—1954）是美国数学家、逻辑学家，制定了利用符号来表示逻辑的推理系统，同时也是可计算性理论的奠基者。

⑤ 阿隆佐·邱奇（Alonzo Church，1903—1995）是美国数学家，于1936年首次发表可计算函数的精确定义，对算法理论的系统发展作出了巨大贡献。

(3) 架构大师：能否构建更好的计算机？

(4) 机器智能的雕塑大师：能否编写程序让计算机自己找到解决问题的方法？

本书介绍了15位当代最伟大的计算机科学家，描述了他们的生活历程以及工作成果。他们都是第一流的创新大师，其中有8位是图灵奖[①]得主，其荣誉相当于计算机领域的诺贝尔奖。事实上，如果没有他们作出的贡献，现代计算机很难像今天这样深入平常百姓家。在本书中，他们解释了自己对科学产生兴趣的缘由，回顾了成长环境和其他科学家对他们产生的影响，阐述了各自进行基础探索和发现的途径，同时也分享了对未来的看法和主张。

在说明这些思想及其重要性时，本书尽量避开了科学术语，因此读者不需要任何专业知识，只要对计算机的发展演变，以及这一领域的科学家们的思维方式感兴趣即可。

设想回到1690年去拜访艾萨克·牛顿。你也许会问他对于惯性力的看法，而他也可能会讲述自己早年间在伍尔索普的农场生活[②]。能与计算机学科的艾萨克·牛顿们当面交流，是我们莫大的荣幸，这也鼓舞我们写出了本书。

关于写作风格与表现形式

本书立足于现场访谈的第一手记录，因为我们相信，这些科学家们的言传身教更能帮助读者透彻地了解他们的个性和方法，这比经过我们的转述和修饰要好得多。我们负责勾画出相应的历史背景，并解释相关的科学和数学原理，但关于这些科学家是如何提出自己的发现，当时他们都在想些什么，最好还是听他们自己来说。

考虑到行家里手通常会对技术细节感兴趣，我们把相关部分用补充内容的形式展现出来。也许读者把书读完一遍之后还会要回头再看看这些部分。我们也为刚刚接触这个领域的读者们提供了一份基本术语的词汇表。此外我们还在每部分开篇几页按时间顺序列出了本领域及诸位科学家生平的大事，帮助读者在时间上进行定位。本书的最后两章“后记”和“结语”，则探索了两个问题：伟大的计算机科学家在他们的人生和思想上是否存在着家族相似性[③]？25年后，类似本书这样的著作又将会探讨哪些新的发现？

① 图灵奖（Turing Award）由美国计算机协会（ACM）于1966年设立，专门奖励那些对计算机事业作出重要贡献的个人。其名称取自世界计算机科学的先驱阿兰·图灵。获奖者的贡献必须在计算机领域具有持久而重大的技术先进性。目前图灵奖由英特尔及谷歌公司赞助，奖金为25万美元。

② 牛顿出生于林肯郡伍尔索普（Woolsthorpe）的一个农村家庭。至于惯性力，在现实中其实并不存在，它只是为了使牛顿运动定律在非惯性参照系中成立而人为加入的一个假想概念。

③ “家族相似性”（family resemblance）一词由著名哲学家维特根斯坦（Paul Wittgenstein，1887—1961）提出。他认为同在一个“范畴”中的成员就像同处一个家族，每个成员都和其他一个或数个成员共有一项或数项特征，但几乎没有一项特征是全体成员共有的，于是以这样环环相扣的方式通过相似性而联系起来成为一类。

致　　谢

感谢书中所介绍的 15 位计算机科学家，他们慷慨地贡献了自己的时间和知识，耐心地接受冗长的访谈，并且仔细审核了本书的初稿。

感谢我们的同事和学生，尤其是布拉德·巴伯、欧尼·戴维斯、马丁·戴维斯、本·高德柏格、莫迪凯·哥林、安德里亚·科尼格、丽萨·M.李斯特、埃德·施恩伯格、杰克·施瓦兹，以及让手稿更加通顺的马修·斯莫斯那。此外，很多忠实的朋友也为我们提供了有价值的建议，例如阿尔伯特和直子·亚当斯夫妇、阿曼达·布劳曼、哈达斯·哈里尔和玛莎·帕拉布尼亚克。琳达·豪斯曼和赫伯特·斯托严为我们提供了珍贵的照片。而劳伦·辛尼一直都是作家的智慧源泉。

西尔维娅·沃伦是最杰出的文字编辑之一：精确、洞察深刻而且极富建设性。

作为出版商，杰瑞·里昂斯和以往一样讨人喜欢，他总是毫不显山露水地为我们建议新的探索方向。助理编辑莉莎·吉布森为我们提供了很多帮助和支持，生产监督编辑斯蒂文·皮萨诺和设计主管卡伦·菲利普斯也是如此。同时还要感谢校对员雅克·爱德华兹。

目　　录

第一部分　语言大师：如何与机器对话……1

1 约翰·巴科斯　不断进取的发明家……4
2 约翰·麦卡锡　不走寻常路的常识逻辑学家……16
3 艾伦·C.凯　清晰的浪漫主义梦想……28

第二部分　算法大师：如何快速地解决问题……39

4 艾兹赫尔·W.戴克斯彻　可怕的说明文和最短路径……42
5 迈克尔·O.拉宾　机会的可能性……51
6 高德纳（唐纳德·E.克努斯）　逐渐趣异一线牵……66
7 罗伯特·E.陶尔扬　寻找优秀的结构……76
8 莱斯利·兰伯特　时间、空间和计算……89
9 史提芬·古克和利奥尼德·列文　良解难觅……101

第三部分　架构大师：如何构建更好的机器……115

10 弗雷德里克·P.布鲁克斯　成功带来的愉悦……117
11 伯顿·J.史密斯　与光速赛跑……127
12 W.丹尼尔·希利斯　与生物学的连结……136

第四部分　机器智能的雕塑大师：如何让机器更聪明……147

13 爱德华·A.费根鲍姆　知识的力量……149
14 道格拉斯·B.莱纳特　一场二十年的豪赌……159

后记：下一个 25 年……173

结语：成功的秘密……176

术语词汇表……179

参考文献……189

第一部分

语言大师：如何与机器对话

假设现在第二次世界大战刚结束不久，而你是一位总工程师，负责一项计算机制造计划。你收集了一堆电线和开关，任务是构建出一台机器，以便预测飞机在空中的飞行模式。鉴于当时的技术水平，你只能使用一些不可靠的部件，而且构建的机器要便于装配。这台机器消耗的电力足以供应一个小型工厂。此外你还为它配备了一个足以容纳几千个字符的存储器、加法器、乘法器，以及一套用于计算的指令集。

指令是最不需要操心的问题，它们只需表达清楚，能够完成任务即可。

为了提高效率，你需要让各种指令尽可能与工程设计的细节相适应。这台机器的存储器具有不同的运行速度和能耗，因此你需要设定一个指令组用于在慢速存储器和快速存储器之间传递数据，设定另一个指令组用于在快速存储器中对数据进行四则算术运算或其他操作。这种分组方式无疑不利于编写预测飞行模式的程序，如果将存储器视为一种不分类的资源来进行指令分组，应该会容易不少，但你并不为此担心。在当时，一毛钱就能雇来一打学数学的编程人员，而相比之下，机器本身却非常精致、昂贵且罕见。

五年之后，情况发生了根本性的变化。你吃惊地发现，随着对程序的需求高涨以及硬件成本的降低，程序设计的费用要远高于硬件的购买和维护成本。（这种态势一直延续了下来。到了20世纪90年代中期，在大多数企业的软件开发环节中，程序设计的开支已经达到了硬件开支的50倍左右。）因此你为自己制定了两个新的目标：缩短程序第一个版本发布所需要的时间，以及让程序更易于改进，以适应客户需求的变化。

你决定修改编程人员与机器沟通所使用的语言。新的语言应当反映出所要解决问题的结构，而不再是反映存储器的快慢层级和算术运算。每一种行业都会根据自身的目的而发明某种语言，例如电影导演会用“开拍”（action）这个简单的术语来表示“摄像师开始拍摄，演员开始表演，其他人员各就各位”。由于在20世纪50年代中期，计算机主要应用于纯科学和数据处理，因此你更关注的是创造一种数学语言，包括运算公式、求和以及向量数组。为了开拓思路，你拜访了许多研究人员，也了解了许多项目，它们允许程序员输入类似$(X+Y)/Z$这样的数学表达式，然后计算机就能自行编译并且进行相应的运算。

到了1958年，有一门科学编程语言脱颖而出，它就是约翰·巴科斯[①]和他在IBM公司的同事合作设计的Fortran（取自formula translation的缩写，即公式翻译）。这门语言至今仍然是自然科学界使用最为广泛的编程语言。

与此同时，一个研究团队在1956年开辟了计算机的一个崭新领域：人工智能（artificial intelligence）。这门学科的一个目标是创造出具备人类推理能力的计算机。由于人类大部分的推理过程都需要进行抽象（例如积木的相对位置、句子的语法、不同洗衣皂的优缺点），偏重数学运算的Fortran语言对此并不适用。卡内基-梅隆大学的艾伦·纽厄尔[②]和赫伯特·西蒙[③]建议用符号表（list of symbols）来表现抽象概念，并且把推理过程视为一种"符号操作"（symbol manipulation）。

假设有一块红色积木、一块蓝色积木和一块黄色积木。一个5岁大的小孩也可以告诉我们，如果红色积木在蓝色积木之上，而黄色积木又在红色积木之上，那么黄色积木必然也在蓝色积木之上。将前两个事实用符号表来表示，我们可能会写：(above redblock blueblock)以及(above yellowblock blueblock)。之后引用符号操作规则，我们便能得出：如果(above x y)和(above z x)，则(above z y)。

年轻的约翰·麦卡锡[④]对这一想法印象深刻，但他认为纽厄尔和西蒙的设计过于复杂。他简化了设计，加入了两个强大的功能，也就是我们所知的递归（recursion）和求值（eval），并且创造了一门称为Lisp的语言（取自"list processing"的缩写，即表处理）。Lisp不仅是人工智能方面的首选语言，且自1958年诞生起，和Fortran一起不断为计算机语言的设计提供灵感。

这一年的夏天，一个国际计算机科学家小组在苏黎世成立，开始设计纳入Lisp元素的Fortran语言的继任者。他们的精诚合作产生了里程碑式的Algol 60，它直接催生了Pascal、C和Ada这些语言，同时也是从PostScript到Excel等各种语言的直系祖先。

然而到了20世纪70年代早期，一些研究人员逐渐发现由Fortran、Algol和Lisp所衍生出来的上百种语言都存在着一个共同的弱点：写出来的程序让人很难读懂。在实际工作中，人们发现如果想修改程序的行为，往往都需要整个重写。

拥有生物学背景的艾伦·凯[⑤]认为应该把程序构建成像活的生物那样：由独立自主的"细胞"构成，彼此通过讯息传递进行合作。他认为通过自主的单元来构建程序，将会使这些单元能够融入新的上下文情境。对于这种"细胞"，他使用了"对象"（object）一词来称呼，并且将这种方式称为"面向对象"（object-orientation）。对象的行为取决于它得到的讯息。

---

① 约翰·巴科斯（John Warner Backus，1924—2007）被称为Fortran语言之父，1977年获得图灵奖。本书第1章详细介绍了他的生平。

② 艾伦·纽厄尔（Allen Newell，1927—1992）专于计算机科学和认知信息学领域，是信息处理语言（IPL）的发明者之一。1975年他和赫伯特·西蒙因人工智能方面的基础贡献而一同被授予图灵奖。

③ 赫伯特·西蒙（Herbert Alexander Simon，中文名为司马贺，1916—2001）是现代一些重要学术领域，如人工智能、信息处理、决策制定、注意力经济、组织行为学、复杂系统等的创建人之一，并且获得了很多荣誉，如1975年的图灵奖、1978年的诺贝尔经济学奖以及1986年的美国国家科学奖章。

④ 约翰·麦卡锡（John McCarthy，1927—2011）是人工智能方面的鼻祖，1971年获得图灵奖。参见本书第2章。

⑤ 艾伦·C. 凯（Alan Curtis Kay，1940—）在面向对象编程和窗口式图形用户界面方面作出了先驱性的贡献。2003年获得图灵奖。参见本书第3章。

比如说，在与图片有关的程序中，可以用“旋转30度”、“开始打印”这样的讯息或命令来描述常用行为。而如果是操控登月机器人的程序，则可能是“收集岩石样本”或者“找到高度超过1米的石块”。20世纪70年代中期，艾伦·凯带领他在施乐公司的团队设计出了首个面向对象的语言Smalltalk。大批年轻人在数年间加入了它的用户群。由此开始，Smalltalk深刻地影响了C++等日趋成熟的编程语言，其自身也衍生了一批专业的追随者。

过去四十年间，世界上诞生了成百上千种编程语言，约有60种在今天仍然被广泛使用，其中包括文字处理、电子表格、图形设计和动画设计等领域的专用语言。就像工具帮助木匠解决问题一样，一门好的编程语言也能帮助我们解决计算机的问题。而且编程人员也正如木匠习惯于自己的工具那样习惯于自己钟爱的语言。

语言学家本杰明·沃尔夫[①]曾经说过：“语言塑造了我们思考的方式，决定了我们思考的内容。”他这番话是针对人类语言所言，而计算机语言则进一步印证了这一主张，尤其是那些灵感来源于巴科斯、麦卡锡和凯的成果的计算机语言。

① 本杰明·沃尔夫（Benjamin Lee Whorf，1897—1941）是美国人类语言学的重要人物。1931年于耶鲁大学师从人类语言学家萨丕尔，提出了“萨丕尔-沃尔夫假说”，创立了“语言决定论”和“语言相对论”，认为语言影响并制约着人类的思维模式。

1

约翰·巴科斯
不断进取的发明家

> 我们不知道自己想要什么，也不知道该怎样去做。一切都是顺其自然地发生。我们面临的第一个挑战就是不知道这种语言看上去应该是什么样子。然后就是怎样解析表达式——这是个大问题，而且在今天来看，我们当时的做法非常笨拙……
>
> ——约翰·巴科斯谈及Fortran的发明过程

常言道，需要乃发明之母。不过对有些人来说，发明的动力并非来自需要，而纯粹只是因为受不了事物的混乱和低效。约翰·巴科斯就是这样一位发明家。他开启了三个伟大的发明：第一门高级编程语言Fortran、为高级语言描述语法规则的巴科斯–诺尔范式，以及函数式编程语言FP。他的这三大贡献至今仍推动着世界上的科研和商务的发展。然而对巴科斯自己来说，这些发明之所以诞生，只是由于他对当时的概念工具感到不耐烦。

对效率低下的厌恶似乎来自于家庭的遗传。第一次世界大战前，巴科斯的父亲从阿特拉斯火药公司的一个小职员被提拔为首席化验师，这家公司主要生产用于炸药的硝化甘油。他的晋升有着充足的理由。

> 他们的工厂产量很低，而且经常会发生爆炸事故，却一直无法找出原因。产品的生产对作业温度极其敏感。我的父亲发现他们的高价德国温度计不准确。于是他前往德国，学习了温度计的制作技术，并且生产出了一批高质量的温度计。之后，工厂的爆炸事故明显少多了。

第一次世界大战期间，老巴科斯应征入伍成为一名军需官。战后他并未得到政府允诺的杜邦公司[①]的职位，于是转行做了一名证券经纪人。到1924年约翰·巴科斯在费城出生时，他的父亲已经在一战后的经济繁荣中略有发迹。巴科斯在特拉华州威明顿市度过了他的童年时光，之后在宾夕法尼亚州波茨敦市就读于著名的希尔中学[②]。

---

① 杜邦公司（DuPont）是世界排名第二大的美国化工公司，创办于1802年，早期以制造火药而出名。

② 希尔中学（The Hill School）是一所美国的大学预科学校，1851年建立，十校联盟成员。著名校友还包括美国前国务卿贝克。

我每年的考试都不及格，从来就没认真过。我讨厌学习，成天游手好闲。为此我不得不每年都参加新罕布什尔州的暑期补习班。我其实很高兴，因为在那里我可以驾驶帆船，玩得非常痛快。

1942年巴科斯好不容易从希尔中学毕业，依照父亲的意愿进入弗吉尼亚大学主修化学。他对理论知识还比较感兴趣，但痛恨实验室。大部分时间，巴科斯都忙于参加各种派对，坐等年龄一到就去服兵役。到第二学期结束为止，他每周定期参加的课程只有一门不计学分的音乐鉴赏。最后他终于引起了校方的注意，弗吉尼亚大学的生涯也随之结束。随后他于1943年入伍。

巴科斯成为了一名下士，在佐治亚州的斯图尔特堡带领一队防空兵。不久他在一次能力测试中表现优异，于是被指派前往匹兹堡大学攻读工程学预科。而之后的一次医学能力测试可以说是救了他的命。

战友们被装船参加阿登战役[①]，而我则跑到哈弗福特学院念医学预科。

作为医学院预科的一部分，巴科斯在亚特兰大一家医院的神经外科病房实习，治疗头部创伤。一次意外的巧合，巴科斯被诊断出患有骨肿瘤，在颅内安装了一块金属板。不久之后，他就读于花与第五大道医院的医学专科学校（现在的纽约医学院），但只坚持了九个月。

我受够了。医学专科学校的人不喜欢思考。他们只会死记硬背，而且也要求你这样做。在那里根本不允许你思考。

在此期间，巴科斯感觉颅内的那块金属板不太合适，于是到附近专门研制金属板的斯塔顿岛医院，希望能更换一个。由于对医院提出的设计方案不满意，他掌握了相关技术后自己设计了一款。在此之后，巴科斯退出了医学领域，在纽约找了一间小公寓住下来，每个月租金18美元。

当时我真的不知道这辈子到底想要干些什么。因为我喜欢音乐，所以想要搞一套高保真音响，但在当时并没有这种设备，于是我上了一所无线电技术学校。在那里我遇到了一位非常好的老师，也是我遇到的第一位好老师，他要我与他合作，帮某个杂志做些电路特性的计算工作。

我记得做了一些相对简单的运算，得出了放大器电路曲线上的几个关键点。工作非常艰苦、乏味，但是它引发了我对数学的兴趣。它能付诸真正的应用，正是这一点吸引了我。

巴科斯进入了哥伦比亚大学的基础教育学院，开始读一些数学课程。他不喜欢微积分，但对代数很感兴趣。到了1949年春，25岁的巴科斯还有几个月就将拿到数学学士学位，但仍然对自己的前途一无所知。

就在这个春天的一天，巴科斯参观了位于麦迪逊大街的IBM计算机中心。当时他参观了可选循序电子计算器（Selective Sequence Electronic Calculator，SSEC），这是一台IBM早期基于电子

① 阿登战役也称突出部战役（Battle of the Bulge），是第二次世界大战期间纳粹德国1944年西线最大的阵地反击战，双方投入近60个师，德军和盟军各伤亡8万余人，其中7.7万人为美军，被称为第二次世界大战期间美军最血腥的一次战役。

真空管制造的机器。

SSEC体积非常庞大，占据了很大一个房间，其上密布着仪表和线路。在参观中，巴科斯向带他参观的人提出自己想找一份工作，而她让他去找主管谈。

> 我说算了，我没法去。当时我看上去很邋遢，头发也很乱。但是她坚持让我试试，所以我还是去了。他们让我参加一个测验，我做得还可以。

巴科斯受聘在SSEC上工作。这台机器实际上并不是现代意义上的计算机。它没有储存软件的存储器，因此程序每次都得靠穿孔纸带"喂"进去。SSEC有着成千上万个电子机械部件，运行并不是很可靠。

> 在那台机器上工作很有乐趣。你可以独自使用整个机器。你必须时刻就位，因为那玩意儿每三分钟就会出错，然后停止运行。你必须想办法让它重新启动。

编程的情况也很原始。

> 一份操作说明和一份指令清单，这就是你能做的编程。要想完成一件事情，每个人都得自己想办法。而完成事情的方法有无数种，所以人们自然会用各种各样的方式来编程。

巴科斯在SSEC上工作了三年。他的首个大项目是月球历表的计算，即计算在任一给定时刻月球的位置。在当时，IBM有能力供养一个纯科学部门，他们曾与哥伦比亚大学合作一个项目，想办法将穿孔卡和纸带机用于科学研究。公司的收入来源主要是为企业和政府生产纸带机。夺人眼球的庞然大物SSEC只是为这种技术起到些宣传作用，而技术本身最终还是被IBM淘汰了。不管公司当初是出于什么动机，巴科斯毕竟从SSEC的工作中学到了很多东西。同时他也为科学计算作出了自己的第一次贡献：在小型机上做多位数运算的痛苦，迫使他开发了Speedcoding程序。

Speedcoding：用小字长表示大数值

在计算机领域，作为整体进行四则运算的一串数码单位称为"字"（word）。这些字的长度都是固定的，根据硬件不同，通常能包含8到32位数字。由于这种长度的限制，必须采用特别的手段才能表现出以埃（千万分之一毫米）或者光年为单位的数值。

著名数学家约翰·冯·诺依曼[①]是科学计算领域的先驱之一。早在匈牙利的童年时期，冯·诺依曼就被视为数学天才，高中尚未毕业就发表了自己的首篇数学论文。他的记忆力也同样惊人，曾一句不漏地复述出整部《剑桥世界古代史》，而这只是为了取乐。他一生的科研成就

① 约翰·冯·诺依曼（John Von Neumann，1903—1957），美籍匈牙利人，现代电子计算机创始人之一，被称为计算机之父。他在计算机科学、经济学、量子力学及几乎所有数学领域都作出过重大贡献。1931年以不到30岁的年纪成为普林斯顿大学的终身教授，1933年转入该校高等研究院，与爱因斯坦等人一起为最初的六位教授。1994年被追授美国国家基础科学奖。

包括古典数学和量子力学方面的基础理论，以及提出了博弈论。1933年冯·诺依曼移居美国，进入了普林斯顿大学高等研究院，并且在第二次世界大战期间的曼哈顿计划[①]中承担了领导作用。

冯·诺依曼意识到，在原子弹的设计过程中需要涉及非常大和非常小的数字的运算，而宾夕法尼亚大学莫尔电机工程学院（Moore School）的一个电子计算器项目引起了他的兴趣。1944年他加入了这个项目，最初作为观察员，之后便开始亲身参与，并且大力提倡在计算机中既存储指令又存储数据的方法。这种方法在今天早已应用于所有的计算设备[②]。

第二次世界大战结束后，冯·诺依曼研制了自己的计算机（名为“约翰尼克”），并且设计了一批早期程序，用于解答核物理中一些无法通过手工解决的问题。出于对科学计算的浓厚兴趣，他建议在计算机中使用一种“换算因子”（scaling factor）来存储和操作那些非常大或者非常小的数字。

原理其实很简单。假设一个计算机字只能存储3位数字，那么要想表示数字517，就需要向计算机字中写入517，并且指定其换算因子为0。要想表示51.7，则同样写入517，然后指定换算因子为-1。表示5170则指定换算因子为1，表示5 170 000则指定换算因子为4，依次类推。正值的换算因子等于跟在数字后面的0的个数，而负值的换算因子则等于小数点后的数字个数。

在进行运算时，编程人员也许并不知道每次计算的精确结果，但他们应该能知道它的换算因子是多少——至少冯·诺依曼认为是这样。没错，原理确实很简单，但前提是你碰巧还是一个优秀的数学家。巴科斯从程序员的角度出发，对此提出了自己的看法。

> 这样的结果是，你必须对问题了如指掌，知道其中各种换算因子，否则数字就会溢出，或者因为取整时误差太大而出现错误。在这样一种计算机条件下，编程工作变得非常复杂。

与IBM的同事哈伦·海尔里克（Harlan Herrick）一起，巴科斯开发了一款名为Speedcoding的程序，利用浮点数来支持运算。浮点数自身带有换算因子，从而为编程人员卸下了重担。巴科斯在Speedcoding积累的经验为他日后迎接更大的挑战奠定了基础。

> 当时每个人都知道编程的费用是多么昂贵。租赁机器需要数百万美元，而编程的成本与之相比只多不少。编程费用居高不下是因为你必须雇佣很多人力用“汇编”或者第二代语言写程序，这些语言与满是0和1的机器码或者说二进制系统相比，进步非常有限。汇编语言相当浪费时间，只为一台机器写命令就要费九牛二虎之力，然后还必须校正由此所产生的大批错误。在这种混乱拖沓中往往会丢失程序本来的目标。

---

① 曼哈顿计划（Manhattan Project）是美国陆军部自1942年起开始实施的核武器研究计划。该工程集中了当时西方国家（除纳粹德国外）最优秀的科学家，动员人数超过10万，历时3年、耗资20亿美元，于1945年成功地进行了世界上第一次核爆炸，并按计划制造出两颗实用的原子弹。整个工程取得圆满成功，但也引发了针对科学家道德的争论。

② 提倡并不等同于发明。冯·诺依曼将此归功于阿兰·图灵，其设想的“机器”也能存储指令。而电子存储器的真正构建则应归功于莫尔学院的约翰·艾克特（John Eckert），他利用水银延迟线原理，用充满水银的振动管来存储数据，从而首开先河。——原书注

Fortran：第一门高级计算机语言

1953年12月，巴科斯向当时他在IBM的老板卡斯伯特·赫德（Cuthbert Hurd）提交了一份备忘录，建议为IBM 704机设计一种编程语言（该机器当时已经具备浮点运算能力）。这个项目后来被称为“公式翻译”，也就是Fortran。它的目标非常明确。

> 这种语言只是为了大幅度缩短编程的时间。我并没有想过会在别的机器上使用它，当时也几乎没有什么别的机器。

但首先，巴科斯需要面对时任IBM公司顾问的冯·诺依曼的强烈反对。

> 他并不认为编程存在多大的问题。我认为他当时反对的主要理由之一是，在浮点运算中你很难知道自己会得到什么。而如果保持定点（fixed point），你起码在出现问题时知道问题出在哪里。但是他没有考虑到编程成本的问题。他认为 Fortran 简直就是不切实际、白费力气。

最终赫德还是批准了这项计划，冯·诺依曼也不再坚持。巴科斯招募的团队兼收并蓄、不拘一格，既有经验丰富的程序员，也有初出茅庐的数学系毕业生。1954年秋天，他的程序研究小组已经有了一个清晰的目标：为IBM 704打造一款能让编程更加容易的语言。

正如他们所见，设计语言并不是最困难的部分。真正的挑战在于如何翻译语言，让机器能够直接理解。进行这项翻译任务的程序被称为“编译器”（compiler）。今天的大学生可以在一学期之内就设计并实现出一个新的语言编译器，但在当时，巴科斯和他的团队缺少这种算法。尤其是对编译器的核心部分“语法分析器”（parser）的设计，他们一直止步不前。

计算机的语法分析和我们在初中学过的语句分析很类似（可能大部分人一毕业就都丢在脑后了）。在语句分析中，我们需要找出不同部分之间的关系，例如限定词、名词、动词、形容词和副词，然后用树形结构图表现出来（参见图1-1）。

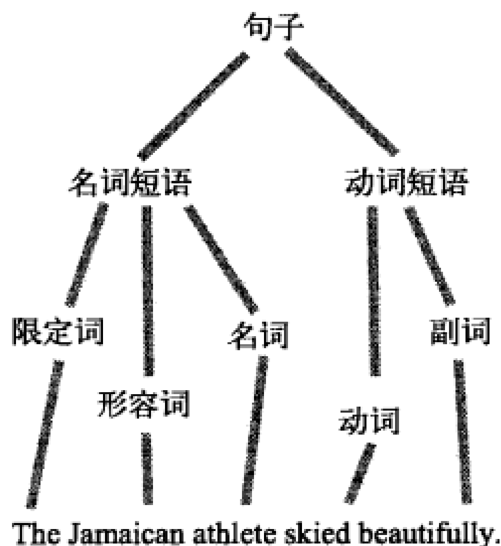


图1-1　分析一个英语句子

对于编译器而言，则是由语法分析器先画出树形结构图，再将“高级语言”（人类易于理解的语言）翻译成“机器语言”，以一系列指令的形式直接输入到计算机的电路中（不同模式的计算机往往支持不同的指令，这也是Windows系统的计算机上不能运行Mac程序的原因）。

机器语言程序的效率取决于设计人员使用那些存储速度高但存储时间短的寄存器（register）的效率。现代计算机最少有16个寄存器，有些则达到几千个。与寄存器相比，随机存取存储器RAM（Random Access Memory）可以储存数百万字节的海量信息，但由于其存储速度较慢，绝大部分计算行为仍然发生在寄存器中。

比如说，(A + B)/C这样一个数学表达式可能会被翻译成以下指令序列：

(1) 复制A的值到寄存器1

(2) 复制B的值到寄存器2

(3) 复制C的值到寄存器3

(4) 将寄存器1和寄存器2的值相加，并将结果放入寄存器1

(5) 用寄存器3除寄存器1的值，并将结果放入寄存器1

在这个例子中，算术运算需要将数据放入寄存器中（有些机器不需要将数据放入寄存器，但相应的运算也会慢很多）。如果在下一步运算中，除了变量A、B、C外还需要D和E，那么编译器就要决定是将D和E的值放入新的寄存器4和5，还是再次使用之前的1、2和3。这个决定又取决于A、B和C将何时被再次用到，以及其他很多因素。这个问题非常复杂，至今也尚未完全得到解决。巴科斯和他的团队属于首批寻求合理方案的人。

而最大的挑战在于Fortran的新功能：DO循环语句。循环语句能让程序员更加方便地执行重复性计算。比如说以下指令序列：

```
DO 13 J = 1, 100
C[J] = A[J] + B[J]
13 CONTINUE
```

该指令将A的第一个元素和B的第一个元素相加，得到C的第一个元素，然后将A的第二个元素和B的第二个元素相加，得到C的第二个元素，依次类推。

循环的贵族发明者

巴科斯和其他20世纪50年代的从业计算机科学家们可能不知道的是，循环这个概念早在100多年前就已经诞生了。1833年，诗人拜伦勋爵的女儿奥古斯塔·爱达[①]遇到了查尔斯·巴贝奇[②]，当时他正在设计一种名为“分析机”（Analytical Engine）的机械式计算机器。爱达在8岁

① 奥古斯塔·爱达·拜伦（Augusta Ada Byron，1815—1852）是英国诗人拜伦之女，母亲是数学家安娜·伊莎贝拉（Anne Isabella Milbanke，1792—1860）。婚后成为洛夫莱斯伯爵夫人。爱达翻译了查尔斯·巴贝奇早期的程序设计著作《分析机概论》（*analytical engine*），她被公认为是世界上第一位计算机程序员。1980年美国国防部开发的编程语言就以Ada为名，纪念这位先驱。

② 查尔斯·巴贝奇（Charles Babbage，1791—1871）是英国文学家、数学家、哲学家、机械工程师，计算机概念的第一奠基人。著有世界上第一部关于计算机程序的专著。他发明的分析机是现代电子计算机的雏型。

时就展现了极佳的数学天赋，是当时极少数能够理解巴贝奇高瞻远瞩思想的人之一。二人展开了维多利亚式的终身合作关系，这也让爱达成为世界上当之无愧的第一位计算机程序员。在为分析机设计程序的过程中，爱达根据需要提出了循环和子程序的概念。

爱达整理出了一篇描述分析机的笔记，但拒绝以自己的名字发表，因为她认为女性不应当撰写科学论文。在巴贝奇和她的丈夫洛夫莱斯伯爵（the Earl of Lovelace）的敦促下，她才同意用自己的姓名缩写A.A.L署名。爱达的一生以悲剧结束：她变成了一个赌徒、酒鬼、瘾君子，最后死于癌症，享年36岁。完全是拜伦笔下的那种不幸结局。

要想有效地编译DO语句，需要用到一种特殊的“变址寄存器”（index register）。Fortran最初是为了IBM 704设计的，而IBM 704上只有3个变址寄存器，因此属于非常宝贵的资源。

> 哈伦·海尔里克一发明 DO 语句，我们就知道会有问题——如何实现？我们只有 3 个变址存储器，但却需要面对许多下标（subscript）。（上例中的 J 就是一个下标，较复杂的程序可能会包含 20 个以上的下标。）
>
> 在程序里，我们很难去判断哪些信息应该使用哪一个变址存储器。如果只用常规方法，得到的代码会很糟糕。我们明白必须去分析代码出现的频率以及其他种种因素。

巴科斯对效率的关注带动了团队的所有成员。Fortran的设计者们很清楚，如果编译得到的机器语言还不如人们手写的效率高，编译就失去了价值，这门新的高级语言就会乏人问津。出于这种原因，他们把将近一半的工作重心都直接放在了如何生成高效的代码上。此举让Fortran一直因为良好的性能而广受赞誉。

IBM 704机总共只有大概80位用户，包括通用电气公司、联合飞机制造公司、洛克希德公司等一些飞机制造行业的厂商。1957年4月，西屋电气公司成为了Fortran的第一个商业用户。巴科斯的团队为他们送过去一套存有语言编译器的穿孔卡片。

> 他们意识到这就是全套 Fortran，没有看任何说明就让它运行起来。当时他们在研究流体力学——计算飞机机翼结构的承压性能等，用于新型飞机的设计。在此之前，他们只能通过大型计算器或者风洞来进行研究。

Fortran的第一次应用没有遇到任何问题。但不久之后西屋公司的科学家和其他一些人发现Fortran编译器中存在着大量错误。巴科斯团队在随后的6个月内修正了这些错误。

巴科斯领导他的精英团队工作了4年，进行了各种复杂的尝试，耗费了大量的精力，但他始终对自己在其中的贡献保持极度的谦逊。

> 让我独自获得如此多的荣誉，可以说对其他成员很不公平。还有罗伯特·纳尔逊、哈伦·海尔里克、洛伊丝·海贝特、罗伊·纳特、艾文·齐洛尔、谢尔登·贝斯特、大卫·赛尔、理查德·戈德堡、彼得·谢里登。是他们创造了大量的成果。管理这样一个团队并不困难，大家干得都很愉快，我的主要职责就是每天下午两点钟提醒大家结束午餐时的象棋比赛，让他们继续工作。

PDG

在发布40多年后，Fortran依然是科学计算的首选语言之一。这门语言至今仍在不断改进的事实就是最好的证明。例如在1992年，一个国际标准委员会给Fortran加入了一个新特性，程序员能够命令编译器允许多台计算机（如果它们有空的话）共同完成一个DO循环语句。不过这些故事说来话长，已经脱离了本书的范围。

20世纪50年代后期，巴科斯停止了自己在Fortran方面的工作。然而他对编程语言的贡献才刚刚开始。

1958年5月，一个由商界、学术界的杰出计算机科学家组成的国际委员会在苏黎世召开会议，目标是改进Fortran，并设计出一种单一的、标准化的计算机语言。他们的成果是一种国际化的代数算法语言（algebraic language），后来被称为Algol。

与Fortran相比，Algol有两个显著的优点。首先，这门新的语言引入了局部变量（local variable）的概念。每一个程序都会命名很多不同的数据元素，例如在前文Fortran的那个例子中，我们命名了元素A、B和C。通常一个程序包含的数据元素很多，因此程序员可能会由于疏忽而导致重复命名，也就是将一个已经存在的名称又指派给了一个新的元素，而这会引发一些恼人的错误。名字太常见的人一定能直观地感受到这种麻烦：账单寄错了地址，信用卡因为错误的原因而被拒绝，或者深夜接到电话结果找错了人，等等。在程序设计领域，在整个程序中一直保持含义不变的名称叫做“全局变量”（global），而弄混对象的情况则叫做“名称冲突”（name collision）。

避免名称冲突的方式之一是让名称的使用环境局部化。比如说，在约克城提到“公爵”一般指的是约克公爵[1]，而在新港爵士音乐节上提到“公爵”，则很可能指的是艾灵顿公爵[2]。与之相似，局部变量指的是名称只在某个有限环境下有效的计算机存储单元。在这个环境之外，同样的名称可以指派给其他的存储单元。

Fortran只允许全局命名，而Algol则可以局部命名。除了更加方便之外，局部命名也使得随后约翰·麦卡锡引入的“递归”这种编程形式成为可能。

计算机的递归函数是在一定程度上按照自身来定义的。举一个日常生活中递归定义的例子，假设一个女子对自己母系祖先的定义为：我的母亲是我的母系祖先，而我母亲的所有母系祖先都是我的母系祖先。这个定义初看起来好像是循环的，不过让我们仔细地揣摩一下。

假设安妮是芭芭拉的母亲，芭芭拉是卡罗尔的母亲，卡罗尔是多娜的母亲（我们也可以继续定义尤妮斯、弗洛伦斯，等等）。根据我们的定义，芭芭拉是卡罗尔的母亲，因此她就是卡罗尔的母系祖先。而安妮也是卡罗尔的母系祖先，因为她是芭芭拉的母系祖先（芭芭拉的母亲）。我们已经知道安妮是卡罗尔的母系祖先，那么她必然也是多娜的母系祖先。

通过递归，程序员能够将一个问题拆分成许多个同样的小问题，然后将各个小问题的解答整合到一起。比如说，要想整理一大堆文件，我们可以将它们分成两半，先整理一半，再整理另一半。最后再把它们放在一起。

① 约克公爵（Duke of York）是贵族头衔，通常被授给英国国王的第二个儿子，除非该头衔由一个前任君主的儿子所拥有。

② 艾灵顿公爵（Edward “Duke” Ellington，1899—1974）是美国著名作曲家、钢琴家，对于爵士音乐的发展影响深远。

从严格的理论意义上来说，递归和局部命名并没有让Algol变得比Fortran更强。但是它们启发了一种新的思考方式，从而在日后产生了深度优先搜索（depth-first search）等算法技巧。我们将会在后面章节中讨论这种算法。

巴科斯喜欢Algol蕴含的那些思想，但同时也意识到，要想清晰地表达它们将会非常困难。

> 他们只是坐在那里玩文字游戏：说明应该这样，而例子应该那样。这些 Algol 委员会（满是关于术语的徒劳辩论）让人非常烦恼，我发现得做点什么。人们需要学会如何准确地使用 Algol。

为了解决这一问题，巴科斯使用了一种叫做“上下文无关文法”（context-free languages）的形式体系。该体系刚刚由语言学家诺姆·乔姆斯基[1]发明（参见下页的补充内容“上下文无关文法与巴科斯–诺尔范式”）。乔姆斯基的发明则来源于埃米尔·波斯特在重写语法方面的成果。

至于巴科斯到底是如何想到这种综合方法的，这个问题一定会让历史学家们忙上一阵。

> 有一点我很困惑。我认为自己研究语法的想法来源于埃米尔·波斯特，因为我曾经在莱姆庄园（Lamb Estate，IBM 设在马萨诸塞州哈德逊的智囊机构）上过马丁·戴维斯[2]的课……所以我想如果需要描述什么东西，直接按波斯特说的那样做就行。但是马丁·戴维斯告诉我说他很久以后才教过这门课（根据戴维斯的记录是在 1960 年和 1961 年）。所以我不知道应该怎样解释。对于乔姆斯基我一无所知。我是一个孤陋寡闻的人[3]。

巴科斯的发明最终成为著名的巴科斯–诺尔范式（Backus-Naur Form，缩写为BNF，其中经历了一连串的偶然）。第一个偶然发生在1959年6月，当时联合国教科文组织将在巴黎召开一次关于Algol的会议，而巴科斯准备了一份有关精确语法的报告，准备在会上发言。

> 这份报告我完成得太晚了，结果未能被收录进大会的官方报告集。于是我只好自己带了一堆报告到会议上，因此分发的效果并不太理想。但是彼得·诺尔（Peter Naur）读到了它，所以一切都不一样了。

诺尔是一位来自丹麦的数学家。他改进了巴科斯的符号表示法，并用于描述整个Algol语言。当编程语言界开始试用Algol时，诺尔的参考手册被公认为是描述该语言语法的最好参考资料。

① 诺姆·乔姆斯基（Noam Chomsky，1928—）是麻省理工学院教授，为20世纪理论语言学研究作出了重要贡献。

② 马丁·戴维斯（Martin Davis，1928—）也是计算机科学发展史上的先驱人物，纽约大学名誉教授。

③ 据马丁·戴维斯推测，Fortran团队的成员、哈佛大学出身的逻辑学家理查德·戈德堡可能曾与巴科斯讨论过波斯特或乔姆斯基的研究成果。——原书注

上下文无关文法与巴科斯–诺尔范式

为了方便理解巴科斯–诺尔范式，我们可以参照以下一些英语短语的描述。其中“名词短语”和“动词短语”这两个术语借用自现代语言学。

句子 →名词短语 动词短语

名词短语 → 冠词 形容词 名词 | 冠词 名词

动词短语 → 动词 名词短语

形容词 → 红色的 | 蓝色的 | 黄色的 | 大的 | 小的 | 聪明的

名词 → 房子 | 女孩 | 男孩

动词 → 喜欢 | 打

限定词 → the | a

竖线符号“|”表示可选。比如说，限定词可以是“the”也可以是“a”。这种语法告诉我们，“the girl likes the boy”（女孩喜欢男孩）是一个正确的句子，因为“the girl”构成了一个名词短语，而“likes the boy”是一个动词短语。而“a smart house likes the boy”（聪明的房子喜欢男孩）这句话尽管在语义上不合常理，在语法上仍然是正确的。

在编程语言中，巴科斯–诺尔范式也具有这种特性。只要遵从其规则，我们就能得到语法上正确的程序，编译器就能成功地将它翻译成机器语言，并且保留其在高级语言程序中的原有含义。当然，原始的程序仍然可能没有意义。编译器只保证翻译的正确性，与原程序是否正确无关。

从自己的发明中解放程序设计

巴科斯发明了世界上最早和最流行的编程语言之一，并且发展了一种可以描述上千种语言的符号系统。很多人，甚至很多杰出的科学家都可能会满足于这些成就而止步不前。但巴科斯没有。他甚至并不确定自己是否喜欢这些成果。

> 当你写完一个 Fortran 程序后，其实并不知道程序到底会怎样运行。你只知道它读取了两个数，然后将它们相乘，然后把值储存起来，然后怎样怎样，最后进行判断，如此等等。但你很难弄清楚它实际上都计算了些什么。你也很难用其他方法进行计算，因为你基本上并不理解程序在做什么事情。

巴科斯的目标是让程序员只需表达出他们“要什么”，而无需知道“怎样做”。1977年他荣获图灵奖，并发表了名为“程序设计能否从冯·诺依曼形式中解放出来”的演讲，向整个计算机科学界提出了自己的观念。

在这里提到冯·诺依曼与他早年间曾反对Fortran的开发并无关系。巴科斯所针对的是冯·诺依曼对计算机的特征阐述，即处理器与存储器相连接，而程序和数据都储存在存储器中。在巴科斯看来，这种特征意味着这样一个基本循环过程：从存储器中调取数据，对其执行一些操作，然

后将结果返回存储器。而他认为凡是遵循这种范式的编程语言都必然缺乏透明度。以约翰·麦卡锡的Lisp（主要用于人工智能）和肯尼斯·艾佛森[①]的APL语言为基础，巴科斯提出了一种叫做FP的语言，其主要目的是用数学函数来构建程序。

“一般”语言（冯·诺依曼式的语言）和类似FP的“函数式”语言的主要差别在于，前者实际上是在直接修改计算机内存，而后者则主要依赖于“复合函数”（function composition）。巴科斯在他的图灵奖获奖演说中介绍了FP，并使用了求两个数组的内积作为例子。这是在物理中常用的一种运算。标准的冯·诺依曼式语言大体上会以如下形式来写这个运算：

```
c := 0
for i := 1 step 1 until n do
    c := c + a[i] * b[i]
```

巴科斯从几个方面批评了这一公式，尤其是以下两点。第一，c不断被修改，因此理解该程序的唯一办法就是理解这种修改其实是把一个新的乘积（product）加在一个运行中的总和（total）上。因此若想理解程序，人们必须能在头脑中执行一遍代码。第二，在公式里定义了动态数组（a和b）及其长度（n）。要想适用于一般的动态数组，这些都需要作为“参数”进行传递——在大多数语言中，这都是一个比较棘手的过程。而在FP中，内积运算可以定义为如下形式：

```
Def Innerproduct = (Insert +)(ApplyToAll *)(Transpose)
```

这个公式要从右往左理解。“Transpose”是将两个动态数组的对应元素配对。在上例中，a[1]应和b[1]配对，a[2]应和b[2]配对，依次类推。“(ApplyToAll *)”是用每一对的乘积来替换它的值，而“(Insert +)”则将这些乘积累加。

巴科斯认为这样做有三个主要的优点。其一，没有隐藏的状态（例如上面程序中的变量c）；其二，它对任意两个同样长度的动态数组都有效，因为它没有定义参数（也就避开了参数传递的问题）；其三，这当中没有涉及重复运算，因为在这个叫做“复合”的过程中，每一步操作，或者说每一个“函数”（Transpose、ApplyToAll和Insert）都只针对前一步的结果应用了一次。

具有讽刺意味的是，函数式语言，包括FP，并没有流行起来，而其原因正是Fortran流行的原因：函数式语言的程序很难编译成高效的形式。随着处理器和存储器运转速度的提高，权威人士预测人们对程序效率的关注可能会大大降低。不过这些预测至今尚未实现。

除此之外，FP语言也不适用于许多日常的编程任务，尤其是那些需要经常读取数据、修改数据然后将其返回数据库的任务，例如更新账目明细，等等。像这样的任务天生就适于使用冯·诺依曼范式。

设计一种函数语言然后将其与实际工作结合起来是很难的。每一个试图这样做的人都会遇到各种各样的问题。

就算巴科斯没有解决这个问题，他也漂亮地提出了这个问题。其他计算机科学家将接过他手中的火炬继续前进。1991年退休之后，巴科斯功成身退，离开了计算机科学界乃至整个科学界。

---

① 肯尼斯·艾佛森（Kenneth Iverson，1920—2004）1962年开发了APL（A Programming Language），这是一种强大、表达丰富而且简明的编程语言。1979年他因对数学表达式和编程语言理论的贡献而获得图灵奖。

他每天都练习冥想，并阅读克里希那穆提[1]和伊娃·皮拉卡斯[2]有关人类内省方面的著作。

大多数科学家之所以成为科学家，是因为他们畏惧生活。在科学中有所成就无疑非常诱人，因为在此过程中无需与人产生冲突，无需应付艰难的人际关系，可以完全按自己的方式生活。在这个近乎纯净的象牙塔里，你可以全力施展自己的才华，而没有任何痛苦。与生活中的困难相比，解决科学问题的困难简直就是微不足道的。

而自我反省则不是一种科学活动：它不可重复，也没有好的理论用以指导你如何去做或者追求什么。通过对自己的反省，你可以真正理解宇宙的奥秘，这是非常奇妙的一件事情。而这是通过任何物理定律都做不到的。

① 克里希那穆提（Jiddu Krishnamurti，1895—1986）被公认为20世纪最伟大的灵性导师。他一生走访全球70多个国家演讲，演讲被辑录成80多本书，并被译为50多种语言，被印度及当代佛家学者认为是龙树菩萨再世。

② 伊娃·皮拉卡斯（Eva Pierrakos）是小说家雅各布·韦士曼之女。她创建了自我转化体系Pathwork，目的是达到人性的纯净化。

2

约翰·麦卡锡
不走寻常路的常识逻辑学家

> 如果希望计算机具有一般的智能，那么其外在结构就必须基于一般的常识和推理。
>
> ——约翰·麦卡锡

一个5岁的小女孩在玩一辆塑料玩具卡车，把它推来推去，嘴里模仿着喇叭声。她知道不能在餐桌上玩它，也不能用它去打弟弟的头。去学校之前，她会把卡车放到弟弟够不着的地方。放学回家后，她也知道在原来的地方可以找到自己的玩具车。

引导她的行为和期望的推理非常简单，任何一个同龄的小孩都能理解。但是大多数计算机却不能。计算机的问题一部分在于它缺少一般5岁小孩能从父母那里学到的日常社会知识，例如不能损坏家具，不能伤到自己的弟弟；另一部分在于计算机没有我们的日常推理能力。人类使用的是一种基于经验的常识推测体系，它与常规逻辑不同，因此也与一般计算机程序员的思维不同。常规逻辑使用的是一种被称为“演绎”（deduction）的推理形式。演绎让我们能从“所有失业演员都当了服务生”和“托米是个失业演员”这两个陈述中推断出“托米是个服务生”这一新的陈述。其优点在于它的可靠性——如果前提成立，那么结论一定成立。同时演绎推理也是“单调的”（monotonic，数学术语，其基本含义为“不变的”）。如果你发现了新的事实，但并不与前提相矛盾，那么结论仍然成立。

然而，尽管我们大多数人都在学校学过演绎推理，却很少在实际生活中用到。5岁的小女孩相信自己的卡车还在原来的位置，是因为她把它放在了弟弟够不着的地方。但如果某天她在出门时看到弟弟学会了爬凳子，可能就不会这么有把握了。5岁小孩掌握的常识推理主要依靠基于经验的推测，而这可能会由于新事实的出现而不得不作出非单调的修改。而且并不是只有5岁小孩会如此。

就算是公认的演绎推理大师歇洛克·福尔摩斯，也并不经常用到演绎推理。在关于一匹受伤赛马的冒险故事《银色马》中，福尔摩斯运用其天赋的洞察力得出结论，看门狗没有叫是因为它认识罪犯。我们的侦探确实才智超群，而且推论看来合情合理，最后在故事中也证明是正确的，但他用的却不是演绎推理——狗可能是被麻醉了、戴了口套，或者当时正在野地里追兔子。

程序员知道如何让计算机进行演绎推理，因为计算机能够理解其中涉及的数学。但如果想让

计算机进行人类赖以生存的这种推测性的（而又常常是正确的）常识推理，就得发明一种全新的数理逻辑。而这正是约翰·麦卡锡为自己设立的目标之一。

麦卡锡的成名还有其他原因。他发明了人工智能领域的首要语言Lisp（list processing，表处理），而且自其诞生之日起，就为编程语言设计提供了丰饶的思想源泉。同时，作为一名教师和难题设计师，他在密码学和平面性检验等亚学科领域激发了众多计算机科学家的灵感。我们将在拉宾[①]和陶尔扬[②]的章节中再行描述。

约翰·麦卡锡1927年出生于波士顿一个共产党积极分子家庭，童年在四处奔波中度过。他的父亲是一名爱尔兰天主教徒，先后做过木匠、渔民和工会组织者，全家一直马不停蹄地奔波，从波士顿搬到纽约，然后又搬到洛杉矶。他的母亲是立陶宛犹太人，最初在联邦通讯社当新闻记者，后来就职于一家共产主义报刊，最后成为了一名社会工作者。麦卡锡早年对科学的兴趣与家庭的政治信仰密不可分。

> 当时普遍相信技术对人类必将有利无害。我记得还是小孩的时候，曾读过一本书叫做《十万个为什么》，是前苏联作家米·伊林[③]在 20 世纪 30 年代早期写的一套通俗科普书。在美国好像没有这样的书。有趣的是，十几年前我在报上看到过报道一个特别早熟的中国孩子，而他也读过《十万个为什么》。

麦卡锡认为自己的青少年时期平淡无奇，但事实证明并非如此。在上高三时，他得到了一份加州理工学院的课程目录，上面列出了该校一年级和二年级的微积分课本。他买了这些书，完成了所有的练习题目。这使得他最终在1944年进入加州理工后得以免修头两年的数学课程。

1948年，麦卡锡开始攻读数学系的硕士学位。同年9月他参加了加州理工主办的希克森脑行为机制研讨会，大数学家、计算机设计大师约翰·冯·诺依曼在会上演讲了一篇关于自复制自动机（self-replicating automata）的论文，这是一种可以对自身进行复制的机器。尽管当时的与会人员并没有明确地将机器智能与人类智能联系起来，但冯·诺依曼的讲话却激发了麦卡锡的好奇心。

1949年在普林斯顿大学数学系作博士论文时，麦卡锡首次开始尝试在机器上模拟人类智能。

> 我把有智能的东西看做是一个有限自动机，与同样是有限自动机的环境相连。我和约翰·冯·诺依曼见了面，他对此非常赞成，敦促我一定要把这篇论文写出来。但最后我并没有写出来，因为我认为它还不够成熟。

“自动机”模拟的是随着时间从一个状态转入另一个状态的机器。比如说，普通的手动变速箱汽车在驾驶员点火启动之后会从“熄火”状态转入“空挡但启动”状态。如果驾驶员挂挡前进

---

① 迈克尔·O.拉宾（Michael O. Rabin，1931—）是以色列计算机科学家，1953年获得希伯来大学理学硕士学位，1956年获普林斯顿大学博士学位。1976年获得图灵奖。参见本书第5章。

② 罗伯特·E.陶尔扬（Robert E. Tarjan，1948—）在数据结构和算法的设计与分析方面作出了众多创造性贡献，于1986年与约翰·霍普克罗夫特共同获得图灵奖。参见本书第7章。

③ 米·伊林（M. Ilin）是中国老一辈读者十分熟悉的前苏联作家。抛开政治因素不谈，他的作品脍炙人口，对我国的科普创作界产生了很大影响。

则转入“启动且挂一挡”状态。而“交互式自动机”（interacting automaton）则是根据其自身的状态以及它所观察到的其他自动机的状态决定从某个状态转入另一状态。有些自动机是智能的（可看做是自带驾驶员），但并不是必须智能。交互式自动机试图在这两种类型之间建立一种连续性的统一体。

麦卡锡放弃了自己对利用自动机模拟人类智能的首次尝试。但在十几年之后，当他从事情境演算（situational calculus）方面的工作时，关于状态和状态转换的思想将重新浮出水面。

在这段时间中，麦卡锡始终没有放弃制造一台像人类那样智能的机器这一想法。1952年夏，普林斯顿大学的一个研究生杰里·雷纳（Jerry Rayna）向麦卡锡建议，可以找一些对机器智能感兴趣的人去收集一些该领域的文章。麦卡锡找的第一批人就有克劳德·香农[1]，“信息论”亦即通信数学理论的发明者。香农的理论最初用于远程通信，后被广泛用于语言学、数学以及计算机科学等领域。

> 香农不喜欢华而不实的术语堆砌。他整理的卷宗为《自动机研究》（*Automata Studies*）。而其中收集到的文章让我很失望，里面有关智能的内容并不多。
>
> 所以在1955年开始筹备达特茅斯计划时，我希望开门见山，使用了“人工智能”这一术语，目的是让参与者们弄清楚我们是在干什么。

1956年在达特茅斯学院举办的夏季人工智能研讨会是计算机科学史上的一座里程碑。这项涉及10人、耗时2个月的雄心勃勃的研究计划，其目标是“基于‘我们能够精确、全面地描述人类智能中的学习等特征，并制造出机器模拟之’这一构想，继续阔步前进”（引自其提案）。

研讨会的四位组织者——麦卡锡、马文·明斯基[2]（当时还在哈佛大学）、纳撒尼尔·罗切斯特[3]（IBM的杰出计算机设计师）和香农——向洛克菲勒基金会申请了一笔资金支持，金额在今天看来几乎少得可怜：主要组织者每人1200美元，再加上“外地与会人员的火车票”，总共7500美元。

麦卡锡在提案中写到，他将研究语言和智能二者间的关系，希望通过程序使计算机能“进行棋类游戏并完成其他任务”。时隔40年后回忆起这次研讨会时，麦卡锡以他特有的直率形容了自己当时的愿景和期望。

> 我为这次会议设定的目标完全不切实际，以为经过一个夏天的讨论就能搞定整个项目。我之前从未参与过这种模式的会议，只是略有耳闻。实际上，它和那种以研究国防为名义的军事夏令营没什么区别。

创造一台真正智能的机器是一个极为困难的过程。尽管这次会议在实质上并未解决任何具体问题，但它确立了一些目标和技术方法，使人工智能获得了计算机科学界的承认，成为一个独立

---

① 克劳德·香农（Claude Shannon，1916—2001），美国数学家、信息论的创始人。曾是美国科学院院士，伦敦皇家科学院院士，1966年获得电子电气工程师协会荣誉奖章、美国国家科学奖章。

② 马文·明斯基（Marvin Minsky，1927—）是麻省理工学院人工智能实验室的创始人之一，著有多部人工智能和哲学方面的作品。他是美国工程院和美国科学院院士，1969年获得图灵奖，2001年获得本杰明·富兰克林奖章。

③ 纳撒尼尔·罗切斯特（Nathaniel Rochester，1919—2001）毕业于麻省理工，1948年起加盟IBM，是该公司第一位系统设计师，1984年计算机先驱奖获得者。

的而且最终充满着活力的新兴科研领域。虽然大多数与会者在会后并未继续从事该领域的研究，但另外那少数人中却产生了一批在该领域影响深远的成就。

来自卡内基梅隆大学的艾伦·纽厄尔、赫伯特·西蒙和J. C.肖（J. C. Shaw）描述了他们的第二代信息处理语言（Information Processing Language，IPL 2）。这三位科学家致力于构建一种名为“逻辑理论机”的程序，用于验证基本逻辑和博弈论中的定理。而为了做到这一点，他们设计出IPL 2这种程序语言以便于操作对象符号，例如象棋的棋子或者逻辑变量里的真值。由于这种操作与针对数字的算术运算非常不同，他们提议使用一种所谓的“表结构”。

让我们冒昧地借用《爱丽丝梦游仙境》来说明如何利用表来进行符号处理。假设柴郡猫告诉爱丽丝“不是我疯了，就是帽匠疯了”。我们用C、H、A代表三种观点，分别是柴郡猫疯了、帽匠疯了和爱丽丝疯了。猫之前的那句声明可以用表的形式表示为(or C H)。然后猫又告诉爱丽丝“不是你疯了，就是帽匠疯了”。聪明的爱丽丝会把这个声明和之前的那个一起表示为(and (or C H) (or A H))。最后猫又说“我们三个中只有一个疯了”。也就是说，至少有两个没有疯。爱丽丝可以将其表示为(and (or C H) (or A H) (or (and (not A) (not C)) (and (not A) (not H)) (and (not C) (not H))))。

把这些声明表示成表的形式之后，我们就可以定义一些表操作的规则，例如(and (or X Y) (or Z Y)) = (or (and X Z) Y)。也就是说，如果不是X成立就是Y成立，而且不是Z成立就是Y成立，那么不是X和Z都成立，就是Y成立。应用这样一些规则，我们就能得出结论(and H (not C) (not A))。那么，根据柴郡猫的说法，只有帽匠疯了。

通过表来进行逻辑推理的优点在于，在推理的过程中表可以扩展、收缩和重组。此外，我们可以用同一种形式表示规则和数据。在研讨会大多数与会者们看来，表操作无疑是这次的赢家。达特茅斯会议的另一项成就是马文·明斯基关于构建几何定理证明机的提案。明斯基在纸上试验了几个例子，认为证明几何定理可能是对纽厄尔和西蒙提出的基于规则的方法的一种很好的应用。IBM公司的赫伯特·格林特（Herbert Gelernter）和纳撒尼尔·罗切斯特决定去实现这个程序。格林特日后又开发了一种帮助有机化学家合成新化合物的工具，他的儿子大卫·格林特（David Gelernter）是并行程序设计和医学人工智能领域著名的研究者及设计者。麦卡锡身为这个定理证明项目的顾问，有机会为智能行为编写程序。

> 格林特和他的助手卡尔·格贝里希（Carl Gerberich）采纳了我的建议，以 Fortran 为蓝本设计了 FLPL——Fortran 表处理语言（Fortran List Processing Language）。其中也加入了一些他们自己的想法。

1956年，约翰·巴科斯和他在IBM的团队发布了首个高级编程语言Fortran，将从事数字运算的程序员从为每一台计算机写汇编语言中解放出来。直到今天，Fortran仍然是科学和工程计算中的通用语言。FLPL首次尝试了扩展Fortran的符号操作能力。1958年夏天在IBM工作时，麦卡锡试图用FLPL为自己在高中时常用的代数微分应用写一个表程序，但很快发现需要用到递归条件表达式[①]，而Fortran却不支持递归。

① 比如说，y^2的微分等于$2y$乘以y的微分，这就是一种递归，因为表达式的微分是依照其组成部分的微分进行定义的。——原书注

> 如果 Fortran 支持递归，我就能用 FLPL 做下去。我甚至也考虑了如何往 Fortran 中加入递归的问题，但是那样做过于复杂。

事实证明，IBM很快就失去了对人工智能的兴趣。一些客户认为智能机器可能会威胁到他们的工作岗位，因此20世纪60年代初期的IBM市场营销都把计算机说成是非智能的快速运算设备，百依百顺、只按要求行事。

麦卡锡不再纠缠于修补Fortran，而是转头发明了Lisp。纽厄尔、肖和西蒙后来把IPL形容为一种越变越复杂的语言，而麦卡锡则把他的Lisp形容为一种越变越简单的语言。

“Lisp”是“list processing language”（表处理语言）的缩写。确如其名，Lisp中所有的数据都用表来表示。这些表都被包含在圆括号中。比如说，(Robert taught Dennis) 可能就是表示“罗伯特教丹尼斯”这个句子的一个表。在这种情况下，顺序是很重要的，因为它指明了是谁在教谁。

而（apple tomato milk）则可能表示一个购物清单。在这种情况下，顺序就不重要了，因为这三种商品可以按任意顺序购买。上述这两个例子中，表都包含了“原子”（atom）作为元素。原子和表不同，是Lisp中最小的符号单位，不包含其他任何组成部分。而表还能够包含（而且一般都会包含）其他表作为组成部分。比如说，(Robert taught (Carol and Dennis)) 反映了句子的语法结构，其中的圆括号指明卡罗尔和丹尼斯都是动词“教”的对象。再比如，(times 6 (plus *x* *y*)) 表示$6\times(x+y)$。这里的顺序也很重要，而且圆括号表明x和y是一起的。

通过这种方式，表不仅能够表示构成科学和工程的标准数学结构，还能表示构成语言的语句结构。

从一开始，麦卡锡就拥有一个热情高涨的合作者团队。

> 当我在 1958 年秋天回到麻省理工的时候，我和明斯基有了一个大工作室、一台键控打孔机，此外还配备了一名秘书、两个程序员和六个数学专业的研究生。我们是在春天时向杰里·威斯纳（Jerry Wiesner）申请的这些，理由是为我们的人工智能项目做准备。
>
> 我们连书面提案都没准备，申请就得到了批准。很幸运，当时麻省理工的电子研究实验室刚刚与美国军方签署了一份无固定目标的双向合作协议，而相应的资源还没有到位。我想这种灵活的资源调配正是美国的人工智能研究起步领先于其他国家的原因之一。纽厄尔-西蒙的研究之所以能进行，也是由于美国空军在当时向兰德公司提供了弹性的支持。

随着工作的深入，麦卡锡希望改进这种语言的表达能力。1959年，为了展示Lisp可以明确地表达任何可计算函数，他加入了一个叫做“求值”（eval）的功能。

“求值”允许程序定义新的函数或者过程（procedure），然后将其作为程序的一部分执行。而大多数语言在执行新函数之前都会强制程序中止运行，并且“重新编译”。由于求值函数可以带动并执行任何函数，它扮演了一种“通用图灵机”[①]的角色，是其他计算机的通用模拟器。

求值概念具有非常实际的意义。比如说，由于国际金融市场时刻都在变化，股票交易所必须

① 图灵机又称确定型图灵机，是阿兰·图灵于1936年提出的一种抽象计算模型，在理论上可以计算任何直观可计算的函数。图灵机作为计算机的理论模型，在有关计算理论和计算复杂性的研究方面得到了广泛的应用。

每周7天、每天24小时不停地提供计算服务。如果有人写了一个程序，可以用新的方法分析路透社的股票数据，股票经纪人很可能希望马上就使用它，但又绝不想中断自己机器的使用。求值让这一切成为可能。

Lisp

Lisp的基本操作

典型的Lisp函数和过程要么是把一个表拆开，要么是把几个表合并成一个新表。比如说，"追加"（append）函数可以将两个表首尾相接，从而生成一个新表。如果我们打算用名词和动词短语造句，这可能会很有用，例如现在有两个人物鲍勃和爱丽丝，那么(append (Bob kissed) (Alice))将生成(Bob kissed Alice)这个新表。

另一个有用的函数"倒置"（reverse）可以颠倒表中的元素顺序。例如(reverse (append (Bob kissed) (Alice)))将生成(reverse (Bob kissed Alice))，之后再生成(Alice kissed Bob)。而如果把两个函数互换一下，例如(append (reverse (Bob kissed)) (Alice))，则会生成(append (kissed Bob) (Alice))，之后再生成(kissed Bob Alice)。

不论鲍勃和爱丽丝之间的关系如何，我们能够看出Lisp程序可以由函数构成，而这些函数可以处理并产生新的表。这正是吸引巴科斯发明FP的东西。

Lisp中的递归和求值

麦卡锡把递归作为Lisp处理策略中的核心部分。递归是用操作本身对其进行定义的一种方法，因此程序员可以把问题变简单后再进行定义，从而绕过了循环定义的禁忌。

比如说，如果某个表只包含一个元素，我们可以定义这个表倒置后仍然为它本身。如果这个表包含多个元素，我们可以如下定义它的倒置：倒置除第一个元素以外的所有元素，再在其后追加上第一个元素。这句话看起来有点拗口，我们可以这样说明：把表L中的第一个元素称作是L的头部，表示为(head L)，而余下的元素称作L的尾部，表示为(tail L)。根据Lisp的理念，我们就可以写出这样一段程序（语法并不精确）：

```
define (reverse L) as
if L has one element then L
else (append (reverse (tail L)) (list (head L)))
```

我们可以套用一个具体的例子，而这次变成了三角关系：鲍勃、爱丽丝再加上卡罗尔。那么(reverse (Alice Bob Carol)) = (append ((reverse (Bob Carol)) (Alice))) = (append ((Carol Bob) (Alice))) = (Carol Bob Alice)。

由于函数和过程本身是定义为表的，因此我们也可以用其他函数和过程来构造它们。这样求值函数就可以带动并执行这种函数或过程了。

Lisp中蕴含的思想吸引了负责设计Algol语言（巴科斯和诺尔为其发明了巴科斯-诺尔范式）的国际委员会。1960年，在该委员会于巴黎召开的会议上，麦卡锡正式提出了递归和条件表达式

这两个概念。在标记方法上，委员会有了些争论，但最终仍然接受了他的思想。

Algol是第一个采用Lisp创新的语言，但绝对不是最后一个。Algol的后继语言如Pascal、C、Ada以及其他大多数现代编程语言都支持递归和条件表达式。但直到最近，主流语言都不支持求值，主要原因在于语言设计者们担心程序员往运行中的程序里添加新功能可能会很危险。不过如今的很多程序都必须每周7天、每天24小时地连续运行，人们对求值这种特性的需求越来越迫切，因此大多数实验性语言都包含了求值或类似求值的功能。

近50年来，Lisp一直是人工智能领域的标准语言。麦卡锡并未预料到它会有如此长的寿命，甚至曾建议将其修改成类似Algol那样。然而该领域的编程人员仍然喜欢Lisp最初的语法。麦卡锡和在他之前的巴科斯、之后的艾伦·C. 凯一样，最终已无法控制自己发明的语言的发展方向。

让常识合乎逻辑

幸运的是，麦卡锡一直都只是把Lisp看做一种手段，目的是达到他的主要目标（至今也依然如此）：制造一台像人那样有智慧的机器。

在他于1959年发表的论文《具有常识的程序》中，麦卡锡详细地阐述了这一目标。以这篇论文为起点，他开始了穷其一生的不懈求索，力求将数学的精确应用到所谓“常识”这一难以捉摸的推理形式之中。

为了具体说明这一目标，他将具有常识的程序定义为“对任意给定的事物（自己已知的或别人告知的），能够独立演绎推理出它将产生的一系列直接结果”。作为例子，他描述了一个人从书桌边离开并驱车前往机场这一事件涉及的推理过程[①]。

在论文介绍后的讨论环节中，著名的逻辑学家和语言学家耶霍舒亚·巴尔-希勒尔[②]将麦卡锡的方法形容为“半生不熟的”和“伪哲学的”。他认为麦卡锡所提倡的推理不能被称为是“演绎推理”，因为其结论并不总是成立。希勒尔指出：“叫辆出租车去机场不是更便宜吗？难道不能退订这趟航班，或者做点别的事情？”

麦卡锡回应了批评，他同意自己的论文基于“不明确的哲学假定之上。……每当我们设计程序让计算机学习经验时，就是在把某种认识论添加到程序中。”从那时起，解决认识论问题成了麦卡锡和其他少数人工智能理论家研究的重心所在。

在读者了解这些研究之前，也许想知道最开始为什么会有人想把常识纳入到计算机程序中来。这对科学或社会能带来什么好处？麦卡锡给出了答案。

> 所有的科学以及专业理论中都引入了常识。当你想要改进理论时，你总要回到常识

① 麦卡锡举的例子是这样的：假设我在家里坐在书桌边，而且我想去机场。我的车也停在家里。问题的解决方法是步行到汽车边、驾驶汽车、然后前往机场。他举这个例子的本意是想说明，推理程序需要有“正式声明的前提”才能得出相关结论。

② 耶霍舒亚·巴尔-希勒尔（Yehoshua Bar-Hillel，1915—1975）是犹太哲学家、逻辑学家、语言学家，20世纪50年代在美国麻省理工学院电子学实验室从事研究工作，后任希伯莱大学逻辑和科学哲学教授、国际科学史和科学哲学联合会主席。

PDG

推理，因为是常识推理主导着你的试验。

所以，如果有人想设计出更好的象棋程序，就需要对自己的成果进行试验，让它分析各种棋局。有关进行何种试验的所有推理都立足于常识框架之内。

我们从很多科学家的著作中都能找到类似的观点。例如理查德·费曼[①]在他著名的物理学讲义中讨论对称时，就提出了如下观点：

我们所有的物理思想，在应用中都需要一定的常识。它们并不只是纯粹的数学的或抽象的思想。当我们说把仪器移动到新的位置，得到的现象相同时，必须正确理解这句话的意思。它的意思是我们移动了所有我们认为相关的东西。如果得到的现象不同，可能是因为有些相关的东西被遗漏了而没有移动，我们就要把它们找出来。[②]

常识推理的主要优点之一在于其应变性。当环境中出现了新情况时，它能够很好地适应。麦卡锡给出了下面的例子。

假设一位旅客要从格拉斯哥经伦敦飞往莫斯科。再假定他知道自己得购买机票，也知道整个旅程的目的地顺序。

很多程序都能够进行如下推理：如果他从格拉斯哥飞到伦敦，再从伦敦飞到莫斯科，那么他就会在莫斯科。但如果他在伦敦把机票丢了怎么办？原计划将不再有效，但如果原计划中包含了再买一张机票的情况，就不会有问题。然而现有的实用程序都不能像这样细致地设置环境条件。

1964年，麦卡锡已是斯坦福大学的人工智能实验室主任。他提出了一种名为“情境演算”的逻辑理论，其中“情境”代表着世界的一个状态。当主体（agent）行动时，情境就会相应发生变化。主体下一步如何行动取决于他对情境的了解。

在麦卡锡的旅行例子中，没有意识到自己丢了机票的旅客会直接前往机场，而意识到自己丢了机票的旅客则可能会去旅行社，找（假设有智能的）代理人进行交涉。

情境演算和有限自动机（finite automata）理论中都有状态转换的概念。但情境演算中的推理不仅取决于情境，同时还取决于主体对情境的了解。主体知道或能够知道得越多，作出的决策就会越正确——而这正是他所希望的。情境演算理论吸引了众多研究人员，他们用各种方式应用这一理论或其变体，但它本身也引起了新的大问题。

在主体众多且互相联系的世界中，与一个主体相关的情境可能会随着其他主体的行为发生改变。当然，在我们的常识世界中，我们知道其他主体的大多数行为在实质上并不会影响我们自己的决策。

---

① 理查德·费曼（Richard Feynman，1918—1988）是著名的美国物理学家，1965年诺贝尔物理奖得主。他提出了费曼图、费曼规则和重正化的计算方法，均为研究量子电动力学和粒子物理学的重要工具。

② 出自*The Feynman Lectures on Physics*（纽约Addison-Wesley出版社，1977年版），理查德·费曼著，罗伯特·雷顿（Robert B. Leighton）、马修·辛特斯（Matthew Sands）整理。第一卷，第11-1页。——原书注

比如说，丢三落四的旅客（主体A）的行为不会因为美国总统（主体B）在华盛顿特区晨跑时买了一个小松饼而发生改变。逻辑不会告诉我们这两个行为无关，但是直觉会让我们得出这样的结论（除非你相信那些子虚乌有的阴谋论）。

在与爱丁堡大学的帕特里克·海耶斯[①]合著的一本书中，麦卡锡用“框架问题”（frame problem）来表示如何简洁地找出不受某特定行为影响的众多事实这一普遍问题。

简而言之，当特定的行为发生时，框架问题不必具体列出所有未变化的事物。

大多数成功的破案都依赖于对框架问题的洞察。英勇的侦探在找到破案线索时，实际上是辨别出了多数人可能会摒弃或忽视的行为和主体之间的联系。例如在《银色马》中，福尔摩斯从看门狗没有叫这一行为确认了狗的主人就是罪犯。而其他人，包括他可敬的伙伴华生医生在内都没有想到这一点。

福尔摩斯在他的调查开始之前作出了如下评论：“对这件案子，思维推理的艺术，应当用来仔细查明事实细节，而不是用来寻找新的证据。这件惨案极不平凡，如此费解，并且与那么多人有切身利害关系，使我们颇费推测、猜想和假设。困难在于，需要把那些确凿的事实——无可争辩的事实——与那些理论家、记者的虚构粉饰之词区别开来。”[②]

计算机所处的环境随时都可能出现新的、意外的事实。要想解决框架问题，它必须像歇洛克·福尔摩斯那样运用一套新的推理方法。1974年马文·明斯基出版了《表征知识的框架》（*A Framework for Representing Knowledge*，麻省理工学院出版社）一书，主张人工智能不应该使用逻辑，因为逻辑与生俱来就是保守的。

为了理解他的论点，我们假设你听说朋友有一只叫做班卓的鸟。你想象中的情境可能包括班卓会飞。但如果现在你的朋友告诉你，班卓是只企鹅，或者被剪过翅膀，或者腿被绑起来了，或者惧怕飞行，你就不会再认为班卓会飞了。虽然没有违背班卓是只鸟的事实，但新的信息却让你改变了最初的结论。

明斯基把这种推理形式称为是“非单调的”（nonmonotonic）。经典的演绎推理都是单调的——只要和旧的信息不矛盾，新的信息就不会改变原来的结论。日常的推理中，充满了非单调性。我们相信总统先生在华盛顿特区大吃小松饼不会影响到伦敦希斯罗机场的旅客，但如果我们发现当地著名的萨克斯音乐会就以他吃小松饼作为开场信号，我们就会改变主意了。

麦卡锡把非单调性看作是解决框架问题的一种途径。有智慧的主体在进行推理时，从直觉上会先假定遥远的行为不会产生影响，而一旦发现这些行为事实上可能有关，则需要重新推理。与明斯基不同，麦卡锡认为逻辑是解决这个问题唯一可靠的基础。为了做到这一点，逻辑需要一种用于推测的形式化基础。

麦卡锡提出了“限定推理”（circumscription）。这一规则允许主体进行如下推测：“我已经知道了全部可能具有性质P的对象。因此如果遇到新的对象，我会假设它不具有性质P。”

---

① 帕特里克·海耶斯（Patrick Hayes，1944—）是剑桥大学数学系学士、爱丁堡大学人工智能专业博士。在他年轻时与约翰·麦卡锡合著的《从人工智能角度看待的哲学问题》一书中，第一次明确声明了逻辑知识表征中人工智能领域的基础概念。

② 出自*The Complete Sherlock Holmes*，第335页。——原书注

例如对于班卓来说，你假定它会飞。也就是说你针对鸟类限定了“不会飞”的性质。而对于一只叫做强波的大象来说，你限定了“会飞”这一性质，即在没有相反证据的情况下，假定这只大象不会飞——因为它不具备会飞的性质[1]。因此，限定推理便于我们对事物进行基于经验的猜测。在了解到更多事实之后可能会证明它是错误的，但同时它也使我们能够作出一些有用的假设。

在一个给定的情境中，要想知道应该限定哪些性质，必须先了解世界上的事实。在班卓和强波的例子中，我们必须知道普通的鸟都会飞，而普通的大象都不会。人类几乎不假思索就能运用这些知识，但对计算机而言却是重大的挑战。

> 比如说，百科全书告诉你拿破仑死于1821年，而威灵顿公爵[2]死于1852年。拿破仑死时，是英国政府关押在圣赫勒拿岛的囚徒。
>
> 但是百科全书本身很可能不会告诉你威灵顿公爵是否听说了拿破仑的死讯。你必须用常识去分析百科的内容才能知道。

你也许会进行如下推理：威灵顿公爵很可能听说了拿破仑的死讯，因为他的戎马一生中有大半都为拿破仑所困扰。因此你可以限定“未听说自己关注的人的死讯”作为性质P。而更简单的问题，拿破仑是否听说了威灵顿公爵的死讯？你可以限定“听说比自己晚死的人的死讯”作为性质P。

最根本的问题在于，如何利用情境的上下文背景来作出自然的推测，例如将军普遍都会关注他们的对手。麦卡锡开始开发一种新的逻辑形式，使其能够处理和利用上下文背景。

> 其中的部分目标就是要理解人们在说什么。如果AI系统无法利用上下文背景，它就不能理解人们在对话中叙述的内容。

这个问题的难度极大。不仅是机器难以正确地应用上下文背景，就算是人类自身有时也会有困难。已故的前白宫发言人托马斯·奥尼尔[3]曾讲过一个欢迎新任总统罗纳德·里根[4]就职的故事。奥尼尔告诉里根，他有格罗弗·克利夫兰[5]用过的书桌。里根笑着说他曾经在电影里扮演

---

① 这一段话可能有些拗口。对于小鸟班卓来说，性质P指的是“不会飞”。套用上一段的推测方法，即“我已经知道了全部不会飞的鸟（比如企鹅、鸵鸟、被剪过翅膀或者怕飞的鸟），因此如果遇到新的鸟，我会假设它不是不会飞的鸟（即会飞）”。而对于大象强波，性质P则变成了“会飞”，即“我已经知道了全部会飞的大象（即不存在），因此如果遇到新的大象，我会假设它不会飞”。

② 威灵顿公爵一世（Arthur Wellesley，约1769—1852）是英国军人及政治家，19世纪上半叶最具影响力的军事及政治人物之一。正是他在滑铁卢战役中击败了拿破仑。

③ 托马斯·奥尼尔（Thomas O'Neill，1912—1994）是美国民主党政治家，从1977年至1987年长期担任美国众议院议长。因公开反对里根总统而出名。

④ 罗纳德·里根（Ronald Reagan，1911—2004）是美国第40任总统（1981年至1989年），共和党人。踏入政坛之前，他曾担任过比赛评论员、救生员、专栏作家和电影演员。

⑤ 格罗弗·克利夫兰（Stephen Grover Cleveland，1837—1908）是美国第22和24任总统，也是该国历史上唯一一名分开任两届的总统。民主党人。

过克利夫兰。里根说的是棒球运动员格罗弗·克利夫兰·亚历山大[1]，而奥尼尔说的则是美国的两任总统。

如此多的问题，如此少的答案

开启了常识推理的潘多拉魔盒[2]，促使麦卡锡开始研究一种能允许任意加入新的事实的形式体系。他将其称为“容变能力”(elaboration tolerance)。一旦允许变化，新的事实可能会使推理者在保留部分结论的同时改变之前作出的一些结论。要想得到所有正确和不正确的结论，可能需要用到基于某些预判的可能性得到的限定推测，例如鸟会飞，而大象不会。至于如何判断哪些推测可能成立，这需要知识，以及把知识系统化、整理成上下文背景或微理论（microtheory）的方法。推理者采取行动，知识也会相应增长，因而又会创造出新的事实。这些能力环环相扣、相互依赖。

在1959年发表的论文中，麦卡锡提出其目标是构建一台机器，使其能够“执行一些简单的、任何非低能的人都能进行的文字推理过程”。从字面上看，这一任务似乎并不算难事，但目标依然非常遥远。

读到这里也许有读者会问，麦卡锡试图为人工智能提供逻辑基础，他的众多尝试到底算是成功了，还是失败了？在他自己看来，自情境演算之后他的工作鲜有付诸实践的。医疗诊断或者预测股票价格等专业系统的开发者们，无不想方设法地避免在自己的系统中进行非单调推理或者一般上下文背景推理。麦卡锡和他的同事们揭示出来的困难，如今成了“不得靠近”的警示牌。

但在未来可能会完全不同。如果项目的覆盖面广阔，而又深不见底（就像道格拉斯·莱纳特的Cyc[3]，试图对亿万事实和规则进行编码），就必将面临许多难以解决的困难。任何这类使用逻辑的项目都必须建立在麦卡锡的工作成果之上（我们在莱纳特一章中将会提到，他的项目运用了R. V. 古哈[4]提出的“微理论”，而古哈则师从麦卡锡）。现有的逻辑工具能否够用？麦卡锡承认他并不知道。

---

① 格罗弗·克利夫兰·亚历山大（Grover Cleveland Alexander，1887—1950）是美国职棒大联盟MLB的投手，1938年入选棒球名人堂。里根在1952年的传记影片《胜利之队》中扮演了他。

② 出自希腊神话。宙斯将一个盒子交给潘多拉，但禁止她打开。潘多拉抑制不住好奇心打开了盒子，结果从盒子中跑出了大量的瘟疫和灾祸。潘多拉后悔莫及，赶紧盖上盒子，只有一点渺茫的希望留在盒中。据说这就是世人虽饱经忧患但仍怀抱希望的由来。

③ 道格拉斯·莱纳特（Douglas Lenat，1950—）是当今人工智能界的泰斗，美国人工智能协会创始人之一。Cyc是一个致力于将各领域的本体及常识知识综合地集成在一起，并在此基础上实现知识推理的人工智能项目。其目标是使人工智能的应用能够以类似人类推理的方式工作。这个项目是由道格拉斯·莱纳特在1984年设立的。参见本书第14章。

④ 罗摩占陀罗·V.古哈（Ramachandra V. Guha，1965—）毕业于印度理工学院马德拉斯分校，现任职于谷歌。他是Cyc项目早期的共同领导者之一。

利用逻辑表达世界中的事实的进展一直都很缓慢。亚里士多德没有发明形式体系。莱布尼茨[①]没有发明命题演算，尽管这种形式体系比他和牛顿同时发明的微积分更加简单。乔治·布尔[②]发明了命题演算，却没有发明谓词演算。戈特洛布·弗雷格[③]发明了谓词演算，但从未尝试过将非单调推理形式化。我想我们人类明白，要明确地表征我们思维过程中的各种事实，表面来看似乎简单，实际上是很困难的。

① 戈特弗里德·莱布尼茨（Gottfried Wilhelm von Leibniz，1646—1716）是德国数学家及哲学家，是历史上少见的通才，被誉为十七世纪的亚里士多德。他和牛顿先后独立发明了微积分，而且使用的数学符号要优于牛顿。他和笛卡儿、斯宾诺莎被称为17世纪三位最伟大的理性主义哲学家。

② 乔治·布尔（George Boole，1815—1864），爱尔兰数学家，哲学家。他深入钻研数理逻辑问题，成功地建立了第一个逻辑演算，并由此创立了布尔代数，成为现代信息技术的数学基础。由于他的特殊贡献，很多计算机语言中都将逻辑运算称为布尔运算。

③ 戈特洛布·弗雷格（Gottlob Frege，1848—1925），德国数学家、逻辑学家和哲学家。是数理逻辑和分析哲学的奠基人。

3

艾伦·C.凯
清晰的浪漫主义梦想

所有对事物的认识都始自于我们不愿盲目地接受这个世界。

——艾伦·C.凯

长久以来，科学家们一直梦想着机器能够改善人们生活的质量。早在17世纪，哲学家和数学家戈特弗里德·威廉·冯·莱布尼茨就曾设想过一种机器，能通过逻辑推理来平息所有的争论。三个世纪之后的1945年，计算机的雏型、微分分析仪（differential analyzer）的发明人范内瓦·布什[①]在《大西洋月刊》杂志上发表了一篇名为《诚如所思》[②]的文章。在这篇文章中，布什预测将来会出现一种叫做“Memex”的设备[③]，能够让普通人在资料库中以链接的形式查询、浏览各种微缩胶片，从多方面了解自己感兴趣的主题。科幻作家罗伯特·海因莱因[④]后来基于他的创意写了一则短篇故事，而这则故事让一个14岁的小男孩着了迷，他就是艾伦·凯。

从那时起，凯开始了他自己充满理想主义的设计发明生涯。他在FLEX机及其后续的Dynabook原型机、编程语言Smalltalk等方面作出的贡献，都极大地影响了现代个人计算机的设计。由他创造出的术语“面向对象”（object-orientation）则是20世纪90年代中期以来编程领域最基本的规范之一。而他与洛杉矶一所公立学校合作的项目，也反映出其自身对于教育体制的不满和反思。

1940年，艾伦·凯出生于美国麻萨诸塞州的春田市，一年后举家迁至他父亲的故乡澳大利亚。凯的父亲是一位设计假肢的生理学家，母亲则是一位艺术家和音乐家。

我的父亲是科学家，母亲是艺术家，所以在我童年的家庭氛围中充满了各式各样的

---

① 范内瓦·布什（Vannevar Bush，1890—1974）是美国著名科学家、教育家，拥有6个不同学位，被誉为“美国世纪的工程师”、“信息时代的教父”，与原子弹、硅谷和互联网等20世纪许多著名的事件都有着千丝万缕的联系。

② 《诚如所思》（*As We May Think*）1945年7月发表于《大西洋月刊》（*Atlantic Monthly*）第176期第1卷。这篇文章被信息界公认为信息科学领域的经典之作。

③ Memex取自“memory & index”的缩写，意为智能资料存储系统，也有人根据其读音译为“满觅思”。这一设备正是当今被广泛应用的超文本结构的原始思想。

④ 罗伯特·海因莱因（Robert Heinlein，1907—1988）是最有影响和争议的美国硬科幻小说作家之一，五次获得星云奖、七次获得雨果奖，与艾萨克·阿西莫夫和阿瑟·克拉克并称为科幻小说三巨头。

想法，以及各种各样表达它们的方式。我至今也从未把“艺术”和“科学”分开过。

我的外祖母是一名教师、学者和女权运动者，同时也是麻省大学阿默斯特分校的创始人之一。外祖父是克利夫顿·约翰逊，一位相当有名的插画家、摄影师和作家（出版了 100 多本书）。此外他还是一位音乐家，弹奏钢琴和管风琴。我出生于他去世的那一年，家里人都认为我是最像他的后代，不管是在兴趣上还是气质上。

1945年初，澳大利亚有可能受到日本的侵略，凯全家又重返美国。1945年到1949年，他们一直住在麻省海德莱市郊的约翰逊农场。凯3岁还在澳洲时就学会了阅读，因此很陶醉于农场的新环境——家里有近6000本书，以及大量的绘画作品。

我记得读过维利·雷[1]的一本书，名字叫《火箭、导弹和星际旅行》。给我触动很深的一点是，如果你从某个星球要去另一个星球，直着飞过去是不对的。你不能让飞船直接瞄准那颗行星，而应该瞄准它即将运行到的某个地方。

有关星际旅行的憧憬让凯兴奋不已，但学校生活对他来说就很乏味了。

到上学时，我已经读过好几百本书了。我从一年级开始就知道老师们在骗我们，因为我早就从书里知道了其他观点。学校里基本上只允许一种观点：老师的观点或者教科书的观点。他们不喜欢你持有不同的意见，所以就会出现斗争。当然了，我会用自己五岁的声音和他们争论。

凯从他的母亲那里接受了音乐的熏陶。他是学校唱诗班的男高音领唱，十几岁开始弹吉他，二十岁时甚至能以专业吉他手的身份挣钱谋生。他看到了音乐与计算之间的直接联系。

计算机程序有点类似格里高利圣歌[2]——单声部的旋律不断在乐章中来回变化。并行程序则更类似于复调[3]。

根据凯的类比，格里高利圣歌的一个乐章会针对某段主旋律进行多个变奏[4]。而在计算机程序的循环中，同一段指令序列也会被重复多遍，每一次都会以不同的值开始，直到某个“终止值”(stopping value) 判断循环结束，之后转入下一个循环，类似于格里高利圣歌的下一段旋律。而在复调音乐中，多个不同的主旋律会同时进行，就像在并行程序中会同时执行多段指令序列。

1949年，凯的父亲获得了纽约一家医院的工作，全家迁往长岛。凯进入了布鲁克林技术高中，

① 维利·雷（Willy Ley，1906—1969）是美籍德裔科普作家，为火箭和航天飞行技术的普及作出了大量贡献。月球背面的一个火山口即以他的姓氏命名。

② 格里高利圣歌是教皇格里高利一世采用的宗教颂歌，在中世纪音乐中极具代表性。音乐以简单、低沉的管风琴为伴奏（或完全无伴奏），压抑的男声连绵起伏，时而徘徊于幽沉的低音部，时而又突然攀升至难以想象的高音区，音域跨度极大。其他特点包括旋律简单、节奏自由、采用调式音阶。

③ 复调（polyphony）是一种“多声部音乐”，作品中含有两条或两条以上的独立旋律，通过技术性处理，和谐地结合在一起。

④ 变奏（variation）是一种乐曲的结构形式，即不断反复乐曲的主旋律，在反复的过程中加以变化。

纽约市最好的科技高中之一。由于有不服从学校的行为，他被勒令暂时休学，很快又患上了风湿热。他以为必须复读一年，结果发现自己已经拿到了足够的学分，完全可以直接毕业。问题是，下一步怎么办？

> 我去了西弗吉尼亚州的贝森尼学院念书，主修生物，辅修数学，但后来在 1961 年因为抗议学校关于犹太人的限额问题被赶了出来。于是我到了丹佛，教了一年的吉他课。然后我被召入伍。当时有这么一项条款，就是如果军队要招募你，你可以自选一个志愿兵种进行申请。
>
> 于是我到美国空军待了两年。我爱读书，于是读了军方的所有规章制度，发现我正在接受培训成为士官。而如果我退出了士官培训，就会变成普通士兵，相应的服役期也会从 4 年变成 2 年。所以我就退出了那个培训。

在美国空军时，凯通过了一项能力测试，成为了一名程序员，在IBM 1401上工作。这个型号的计算机在当时非常流行，市面上大约有15 000台。

> 他们需要编程人员。在那个年代，编程是一种低级职业，而且大部分程序员都是女性。我的老板就是女的。他们也招收语言学家……的确是个很有意思的群体。
>
> 当时我有个朋友，他负责的部分就是我们今天所说的操作系统（控制计算机的程序）。你要知道，1401 的存储器只有 8K（8000 个字符），所以它的操作系统必须小于 1K。结果他做到了，机器能够进行真正复杂的批处理，简直是个奇迹。我协助他工作，所以也了解了一些编程的思想。

新思路的萌芽

要想成为伟大的软件设计者，首先需要有能力把单纯的想法转化为正确、有效的程序，其次要能慧眼识珠，善于发现其他的优秀编程思想的价值。

1961年，凯的工作是为空军解决各个航空训练设备之间数据和过程（procedure）的传输问题。他发现不知哪个程序员想出了一个聪明的办法，就是把数据和相应的过程捆绑到一起发送。通过这种方式，新设备里的程序就可以直接使用过程，而无需了解其中数据文件的格式。如果需要数据方面的信息，可以让过程自己到数据文件里找。这种抛开数据、直接操作过程的想法对凯而言是一个极大的触动，为他日后形成“对象”的概念埋下了伏笔。

离开空军后，凯认为自己应该完成大学学业，于是考入了科罗拉多大学。

> 科罗拉多的数学系比贝森尼学院要好，但是生物系不行。大部分时间我都泡在博尔德分校的校剧院里写舞台音乐。不论什么专业，任何人都能参与剧院的活动，而一年大概能上演 25 部作品，很了不起。我根据《霍比特人历险记》[1]写了一部戏，也参与了其

① 《霍比特人历险记》（*The Hobbit*）是《指环王》作者英国作家托尔金（J. R. R. Tolkien）于1937年出版的小说，叙述了霍比特人比尔博·巴金斯为何拥有魔戒的故事，因此也被称为《指环王前传》。

他一些作品。

凯曾认真考虑过自己是否应该转行从事音乐，但最终还是拿到了数学和分子生物学的双学位，于1966年顺利从科罗拉多大学毕业。他又一次面临自己职业生涯的选择问题。

我考虑过医学，但感觉自己没有足够的责任心。现在我还是这么认为。

他还考虑过哲学，但最终放弃了。最后他决定去犹他州立大学念计算机科学系。

关于犹他大学，我只知道它的海拔超过 4000 英尺，而且有博士点。我很享受山里新鲜的空气。之前我曾想过去威斯康星大学念哲学，幸亏没去成。而科罗拉多的博尔德分校没有博士点。于是我身无分文跑去了犹他。对计算机我还比较感兴趣。

除他之外，计算机科学系还有6个研究生和3位教授，由戴夫·埃文斯（Dave Evans）带领，在此之前他曾经在加州大学伯克利分校负责计算机科学系。

进了犹他读博，在分到办公桌之前你先得读一大堆手稿。它们是关于 Sketchpad[1]的。基本上，如果你理解不了 Sketchpad，在犹他就别想挺胸做人。

他们还有一项传统，就是让新来的研究生干最脏最累的活。我的办公桌上摆着一堆磁带和一张纸条，上面写着："这是 UNIVAC 108 机上跑的 Algol 语言。如果它不能运行，就把它弄好。"那其实就是第一版的 Simula[2]。

伊凡·苏泽兰[3]于1962年提出的"画板"Sketchpad，是有史以来第一个交互式计算机图形系统，而且非常成熟。它提出了主图（master drawing）和例图（instance drawing）的概念。程序员可以在"主图"部分定义各种限制条件。这些条件可以很简单，例如"直线L的端点与某点P之间的距离应为1英寸"；也可以很复杂，例如"形状M的弯曲应遵从牛顿定律"。Sketchpad会检查各个例图，然后根据主图的这些限制条件对它们进行修改。

Simula语言是1965年由挪威人克利斯登·奈加特和奥利-约翰·达尔[4]共同开发的一种语言，它也支持类似主图与例图的概念，只是相关术语有所不同。在这两种语言中，程序员可以定义主图的行为，然后让每个例图都遵从这一行为。关于这些想法，凯进行了很多思考。他开始寻找某种基础构件，支持一种简单、有效的编程风格。

我的灵感就是把这些看做生物学上的细胞。我不确定这一灵感来自何处，但肯定不

① Sketchpad是伊凡·苏泽兰在麻省理工攻读博士时开发出的三维交互式图形系统，使用了早期的电子管显示器，以及当时才刚刚发明的光电笔，是之后众多交互式系统的蓝本。

② Simula是一种编译式的编程语言，它被认为是第一个面向对象程序设计的编程语言。

③ 伊凡·苏泽兰（Ivan Sutherland，1938—）是计算机图形学之父、虚拟现实之父，于1988年获得图灵奖。

④ 克利斯登·奈加特（Kristen Nygaard，1926—2002）和奥利-约翰·达尔（Ole-Johan Dahl，1931—2002）都是挪威计算机科学家，一同开发了最早的面向对象编程语言Simula-I和Simula-67。为此二人在2001年共同获得图灵奖，2002年共同获得冯·诺依曼奖。

是在我看到 Sketchpad 的时候。Simula 也没有给我太多启发。

与生物学的类比让凯总结出三大原则。第一，每个“例”细胞都遵从“主”细胞的某些基本行为；第二，每个细胞都能独立运作，它们之间由能透过细胞膜的化学信号进行通信。第三，细胞会分化——根据环境不同，同一个细胞可以变成鼻子的细胞，也可以变成眼睛或者脚趾甲的细胞。在日后凯将会把“主/例”区别、信息传递和分化原则运用到Smalltalk的设计中，但此刻它们还只是一些初步的想法，似乎重要，却尚无用武之地。

FLEX 和 Dynabook

犹他州立大学有许多项目的资金支持都来自于美国国防部的高级研究规划署（Advanced Research Projects Agency，ARPA），当时可谓是ARPA的黄金时代。1962年至1964年规划署的负责人是J. C. R. 利克里德[①]，他拥有心理学背景，而且对人机交互领域非常感兴趣。

从1962年到1970年，ARPA赞助了一系列大胆的项目，其中一些只是出于对项目负责人的信心，根本与军事无关。1970年的曼斯菲尔德修正案要求ARPA只赞助与军事相关的项目，规划署的名字也被改为国防高级研究规划署（Defense Advanced Research Projects Agency，DARPA）。1993年联邦政府又将名字改回了ARPA，再次回归亲民方针。

在ARPA的资助下，麻省理工学院林肯实验室的威斯利·克拉克[②]于1963年开发出首台个人计算机LINC[③]。这种计算机重达数百磅，与今天的掌上电脑相去甚远，但在当时这种项目只要存在就足以令人兴奋。类似的研究很快跟进，而戴夫·埃文斯对此也有所耳闻。

> 埃文斯曾经在本迪克斯（Bendix）公司当过副总裁，这是一家干实事的公司。他经常为手下带的学生找一些咨询方面的工作。我去了几个月之后，他也帮我找了一份当地的咨询工作。
>
> 当时是 1967 年，戴夫·埃文期认识的一个家伙正打算设计一种台式机，但他不懂软件。我懂软件，所以这看上去是个绝配。

这个“家伙”就是埃德·基德尔（Ed Cheadle）。这个项目就是FLEX机。

> 基德尔是个很棒的家伙，典型的得克萨斯大块头，天性乐观。我们俩相处得非常融洽，很快开始讨论应该如何设计这台机器。他希望把它设计成一种能为医生、律师、专家服务的工程工作站。
>
> 很快我们就遇上了个人计算机最主要的问题：无法预测用户的需求。一旦所有人都

① J. C. R. 利克里德（J. C. R. Licklider，1915—1990）教授于1962年从麻省理工加入ARPA。据估计，在他任职期间整个美国计算机科学领域有70%的研究都由ARPA赞助，使得ARPA不仅成为互联网的诞生地，同样也是电脑图形、并行过程、计算机模拟飞行等重要成果的诞生地。

② 威斯利·克拉克（Wesley Clark，1927—）开发出世界上最早的个人计算机，于1980年被授予计算机先驱奖。

③ LINC之名取自林肯实验室的头4个字母，同时也是“Laboratory Instrument Computer”的缩写，即实验室设备控制计算机。

有权使用，你就会出问题。所以我很早就对可扩展语言（用户能够根据自身领域的术语和操作进行修改的计算机语言）产生了兴趣。

兰德公司曾试验过 JOSS，这是一种为非计算机用户——这里也就是经济学家——设计的系统。我们的想法是把 FLEX 做成像 JOSS 那样。事实也确实如此，FLEX 有多个窗口，甚至还有类似图标那样的东西。

尽管FLEX拥有现代个人计算机的一些特征，凯仍然觉得不太满意，部分原因在于它笨重的个头有350磅。1968年夏，他在伊利诺伊大学进行了一次有关FLEX机的演讲。演讲受到了欢迎，而凯却对紧随其后的一次参观记忆犹新。

在那里我们看到了第一块平板显示器。从会议回来后我花了大半个晚上的时间，试图用摩尔定律计算出什么时候我才能把 FLEX 机装到一块小显示器的背面去。

1965年，物理学家戈登·摩尔[①]就预测，从基准年1959年开始，集成电路板上的密度会逐年翻番。也就是说电路的尺寸每年都会减半。依次计算，凯发现FLEX的20 000个电路元件在1980年左右可以放入显示器的背面——事实证明这一猜测颇为准确。

1968年秋季，凯拜访了麻省理工人工智能实验室的西蒙·派珀特博士[②]。派珀特曾与瑞士儿童心理学家让·皮亚杰[③]一同工作过，他开发了一种简单的编程语言LOGO，并且提倡教儿童使用这种语言自行编程。凯亲眼看到麻省莱克星顿公立学校的孩子们正在编写真正的程序。这一经历改变了他对个人计算机的看法。

我们曾把个人计算机比作私人汽车，而大型机则是公共铁路。

在看到派珀特所做的工作之后，我发现之前的比喻不再成立。孩子们自然不会去开车，我们需要关心的是他们如何识别媒介，掌握文明社会的符号系统。

拿到犹他州立大学的博士学位之后，凯到斯坦福大学人工智能实验室工作了一年（1969年至1970年），教授系统设计。

我对人工智能真的不太感兴趣，部分原因在于它太难了。我的生物学背景告诉我，为人工智能建立一套可行性标准绝不是件容易的事。每次我坐下来思考这个问题，总发现那些令其他人满意的成果无法令我满意。在我看来，他们所做的东西并不能称为是智能。

① 戈登·摩尔（Gordon Moore，1929—）是英特尔公司的创立人之一，摩尔定律的提出者。摩尔定律的内容是：约每隔18个月，集成电路上可容纳的晶体管数目便会增加一倍，性能也将提升一倍，或者说，每隔18个月，每1美元所能买到的电脑性能将翻两倍以上。这一定律揭示了信息技术进步的速度。

② 西蒙·派珀特（Seymour Papert，1928—），美国麻省理工学院终身教授，教育信息化奠基人，数学家、计算机科学家、心理学家、教育家，近代人工智能领域的先驱之一。他于1968年基于LISP创立了LOGO编程语言。LOGO是一种解释型语言，它内置一套海龟绘图（Turtle Graphics）系统，通过向海龟发送命令，用户可以直观地学习程序的运行过程，因此很适合于儿童以及数学教学。

③ 让·皮亚杰（Jean Piaget，1896—1980），法籍瑞士人，近代最有名的儿童心理学家。他的认知发展理论成为了这个学科的典范。

> 马文·明斯基是一个例外，他发展出了“心智社会”（society of minds）的想法。明斯基对生物学很感兴趣，而且他越想越觉得从生物学上得到了很多启示。

在斯坦福教书期间，凯开始构思另一种个人计算机——书本般大小，能将用户（特别是孩子们）与世界相连。

1970年7月，施乐公司的首席科学家杰克·高曼（Jack Goldman）说服高层在加州帕洛阿尔托市建立了一个长期科研部门——帕洛阿尔托研究中心（即PARC，Palo Alto Research Center），并邀请ARPA的梦想家鲍勃·泰勒[①]加盟管理。泰勒许诺凯以极大的自由，让他有机会“听从自己的本能行动”。于是凯加入了施乐PARC，开始设计一种叫做KiddiKomp的笔记本电脑，用一块小电视屏幕（14英寸）作为显示器，并配有可拆卸的键盘和鼠标。

在凯加入之后，泰勒从一家名叫伯克利计算机公司的濒临倒闭的小企业挖来了很多人才，其中包括巴特勒·兰普森[②]（1993年获得图灵奖）等名人。随后，从斯坦福研究院的增智研究中心（Augmentation Research Center）的道格·恩格尔巴特[③]带领的开拓性用户界面团队也来了许多研究人员，包括鼠标的合作发明人比尔·英格利什（Bill English）。英格利什鼓励凯申报预算、建立团队。

凯不情不愿地当起了领导者，组建了“学习研究工作组”（Learning Research Group，LRG），开始招募和他一样对笔记本电脑这一想法怀有热情的人。到了1971年夏天，凯开始设计一种新的语言，名字叫做Smalltalk。

> 这个名字非常不起眼，以至于它的任何成就都会让人感到惊喜。

“面向对象”的诞生

Smalltalk确实与生物学上的类比相吻合：相互独立的个体（细胞）通过发送讯息彼此交流。每一条讯息都包含了数据、发送者地址、接收者地址，以及有关接收者如何对数据实施操作的指令。凯希望这种简单的讯息机制能贯彻到整个语言中去。到了1972年9月，他已经完成了对基本想法的简化，甚至能把Smalltalk的整个定义写在一张纸上。这些想法组成了凯所谓“面向对象”的核心内容，它在20世纪90年代变成了软件设计的基本技巧。

如今，对“面向对象”这一术语过度的滥用几乎导致它失去了自己本来的意义。而它最基础的含义仍然来自类比的对象：独立、相互交流的生物细胞。细胞在特定环境下对特定化学信号产生特定蛋白质。与此类似，计算机对象也会对特定计算机讯息作出特定反应。比如说，一个代表

---

① 鲍勃·泰勒（Bob Taylor，1932—）于1970年至1983年担任施乐PARC主任。他的魅力在于能把性格迥异的科学家们捏合在一起，共同将各种有趣且有益的想法付诸实践，并形成有应用前景的成果。

② 巴特勒·兰普森（Butler Lampson，1943—）在硬件、软件、编程语言、计算机应用、网络等方面均有建树，1992年因分布式系统获得图灵奖，1996年获得计算机先驱奖。

③ 道格·恩格尔巴特（Douglas Engelbart，1925—）是美国发明家，鼠标的发明者，同时领导团队开发了超文本系统、远程会议系统等瞩目的成果。

PDG

视频游戏的计算机对象，会对敲击键盘、移动鼠标等讯息作出改变屏幕显示并同时调整得分的反应。而玩家们只需了解游戏外在的表现即可，无需弄清背后的工作原理。把内部活动的信息隐藏起来，这就是面向对象的精髓。

由于所有的交流都是通过讯息来实现的，因此只要对象收到的讯息和给出的反应能够适用于新的环境，那么对象就可以从一个上下文情境中提取出来，放入另一个上下文情境。这表示软件对象可以迎合设计师尚未设想到的用途，就像一个设计巧妙的车载收音机既可以装入已有的汽车，也可以装入那些尚未设计出来的汽车。

这样做有什么好处呢？原则上，它能让软件设计人员像土木工程师设计建筑那样构造程序：从工厂购买通用的元件，然后用定制化的软件当做"铆钉"将它们固定到一起。比如说，软件生产商可能会为图片对象定义行为，允许它们旋转、放大、投射、修改颜色、延展等。只要程序中需要操作图片，程序员就可以直接从生产商那里订购图片的"类"，然后根据自己的需要在其中添加行为。除此之外，就像预制活动房屋由水槽、马桶、书桌等元件构成一样，程序中新的行为也能由面向对象系统中的元件构建。因此，一个既提供图片又提供搜索功能的图片数据库，其实就是数据库"类"和图片"类"结合的结果。正是因为凯的发明，我们的世界不必再为每一个应用都设计一门新的语言，因为通过创建对象，我们能够模拟任何存在物。

Smalltalk：孩子就是设计师

1972年9月，凯完成了Smalltalk的第一版设计。几天之内，团队成员丹·英格斯（Dan Ingalls）就让语言成功运行。下一个问题是让孩子们学会并使用Smalltalk。凯对此雄心勃勃，他希望Smalltalk能够引发儿童教育方式的改革。在皮亚杰、杰罗姆·布鲁纳①以及鲁道夫·安海姆②的影响下，凯认为图形比单纯的文本更利于儿童的学习。很快地，泰德·克勒（Ted Kaehler）和黛安娜·梅莉（Diana Merry）用Smalltalk编写了基本的制图法，鲍勃·舒尔（Bob Shur）则搭建了动画系统（每秒2到3帧）。

凯招募了阿黛尔·戈德堡③，还有斯坦福大学的儿童工作者史蒂夫·文依（Steve Weyer）。1974年戈德堡设计出一种方法，用一个叫做"乔"（Joe）的小图形框来教Smalltalk。"Joe turn 30!"这个命令可以让"乔"旋转30度，"Joe grow 15!"命令可以让"乔"放大15%。图形框还可以被复制（例如把"乔"复制给"吉尔"）、连续翻转（"forever do"命令）等。

有些孩子学得很快。一个12岁的孩子设计了一个画图系统，和苹果的MacDraw（或者Windows的Draw程序）非常相似。还有个15岁的孩子捣鼓出了一个设计电路的程序。不过据凯所说，在戈德堡教的孩子里只有5%做到了这种超级成就。其他的孩子都很喜欢和"乔"一起工作，但上

① 杰罗姆·布鲁纳（Jerome Bruner，1915—），美国心理学家，曾任美国心理学会主席。他的贡献是教育心理学中的认知学习理论。

② 鲁道夫·安海姆（Rudolph Arnheim，1904—2007）德籍心理学家、美学家，曾任美国美学会主席、美国心理学会心理学与艺术分会主席。

③ 阿黛尔·戈德堡（Adele Goldberg，1945—）是构式语法研究的代表性人物之一，Smalltalk开发团队成员，计算机历史上著名的女性人物之一。

不了“档次”。无论如何，任何人都能很快上手，这一事实已经说明了设计的质量。

不过，施乐的管理层并不像孩子们这么热情。1976年，官僚主义的惰性占了上风，高层否决了对PARC在硬件和软件设计方面继续供给资源。作为反驳，凯和他的同事们争辩说公司应该敢于冒一点险。在一份备忘录中，凯（正确地）预测在20世纪90年代会出现数以百万计的个人计算机，而且会联结至全球化的公用信息设施（也就是互联网）。他指出，施乐拥有生产力资源、营销基础，麾下还有“绝大多数世界上最优秀的软件设计师”，足以称霸市场。遗憾的是这一点未能实现。

> 他们对这些成果的重大意义完全没有概念——因为没有先例可循。我们也没有为他们做好热身。
>
> 就算我们做好了热身，生意人也毕竟不是科学家，他们更像是工程师。一般而言，他们要的不是理解事物，而是想应用。在某些情况下，面对新现象时我们必须花时间去理解它，而不仅只是考虑把它拿来干什么。施乐本身也遇到过这种情况。20世纪50年代时，理特（咨询公司）[1]也曾预测说静电复印术不可能有市场，但是施乐坚持了下来。

施乐从未下力气追捧过PARC的技术，而苹果公司却正好相反。1979年，史蒂夫·乔布斯、杰夫·拉斯金[2]和苹果的其他技术专家拜访了凯的团队，参观一个演示程序。他们对所见印象深刻，但最刻骨铭心的莫过于当乔布斯抱怨屏幕显示算法时，丹·英格斯在一分钟之内就解决了问题。乔布斯试图找施乐买下这款软件，但是尽管在苹果小有投资，施乐仍然拒绝了这一建议。

之后的Smalltalk由施乐的一家分支子公司ParcPlace继续运营，依然因为其惊人的可塑性而赢得大批追随者。20世纪90年代，德州仪器（TI）公司有一个编程项目，但对是用当前最流行的C++还是Smalltalk犹豫不决。TI的Smalltalk团队提出进行一次“编码大赛”。他们打算出三个人，与任意数量的C++程序员展开比赛：早晨由中立方指定一个问题，两队同时开始编程，到中午由中立方再对之前的问题进行小范围的改动。比赛的目的在于判断哪一个团队的程序功能更强。在德州仪器的这次试验中，Smalltalk团队轻而易举地取得了胜利。

让校园天翻地覆

1984年凯加盟苹果之前，曾以驻场思想家的身份为雅达利公司[3]工作了一年。在雅达利，他监督了一些项目，其中的Vivarium是在他到雅达利后才开始构想的，后来成了麻省理工媒体实验室和个性化中心学校（洛杉矶的一所公立学校）的合作项目。

Vivarium让8岁左右的小学生用计算机记录真实数据，并与其他学校的孩子们比较结果。比如说，个性化中心的学生曾记录过学校街区的汽车交通数据，然后与全球范围内其他有相似条件的学校进行比较。通过媒体实验室提供的软件，孩子们可以自行模拟生态系统，并测试诸如“极

---

① 理特管理顾问公司（Arthur D.Little，ADL）是全球最早的管理咨询公司，创办于19世纪90年代。

② 杰夫·拉斯金（Jeff Raskin，1943—2005），美国人机交互专家，苹果公司Mac操作系统的发明者。

③ 雅达利（Atari）是1972年成立的美国电脑公司，街机、家用电子游戏机和家用电脑的早期拓荒者。

PDG

度饥饿的动物是否会试图吃掉原本捕食自己的动物”这样的问题。

表面上看，我们可以说，凯所构想的个人计算机原型Dynabook已经实现。毕竟笔记本电脑、调制解调器、手写识别系统、数据库存取和超文本链接在今天已经随处可见。但是凯仍然抱有隐忧，他认为计算机有可能会沦为“某种大众鸦片”。

> 人们想要的东西往往并不是他们真正需要的，因为两者是受不同的原因驱动的。技术应该让愿望和需求保持一致，并同时满足两者。这非常重要。
>
> 如果生产出计算机却没有正确的价值体系——这就像生产出钢琴却没有作曲一样——你仍然徘徊在门外，只会简单地弹一些现成的儿歌。我们应该让孩子们有自己的传播媒介。
>
> 苏格拉底曾经抱怨过写作。他认为著述会强迫读者接受意见，而不是参与讨论。而计算机远比书本更加生动。不管是公认的观点还是计算机模拟的结果，每一个孩子都应该有机会亲身进行测试、辨别再得出结论。问题是，我们的社会将会鼓励还是压制这种可能性呢？

第二部分

算法大师：如何快速地解决问题

如果读者曾看过食谱，就会理解算法是什么。食谱就是烹饪中的算法。像朱莉亚·蔡尔德[①]这种用烹饪算法谋生的人，必须能用通俗易懂的语言，精确地告诉烹饪业余爱好者怎样去做。不管是法式红酒炖鸡还是俄式奶油蛋糕，食谱必须满足厨师的目标：在合理的时间内做出美味佳肴。

和世界级的厨师们一样，计算机也可以根据各种各样的食谱解决层出不穷的数学问题，例如破译密码或者绘制卡通。而算法设计师的工作就是找到正确的食谱，从而提高解决问题的效率。

每一个计算机从业者都知道，一个好的算法可以缩短解决问题的时间，有时甚至能从数天缩短到数秒。这种大幅度的改进是完全可能的，比如说，在电话黄页中查询条目时（假设查询Egg Electric公司），我们会先打开电话簿的1/3处。如果该页的首字母在E之后，我们就会往前翻页；如果在E之前，我们就会往后翻页。如果翻到了E，则核对第二个字母，依次类推。我们会估计公司名所在的大致位置，如果超过了就往回翻，如果还没翻到就继续往后。最多来回约8次我们就能找到正确的页数。这种算法远比从第一页开始慢慢往下找要好得多。

本书这一部分讲述的科学家们的成名之处，就在于他们能够找到高质量的算法。在我们日常的网络、电子表格或者文字处理软件中，蕴含这些算法的程序每天都会运行数万亿次。除了发明基础算法之外，这些大师还把他们从简单性、正确性和有效性三个方面对算法的理解灌输给了整个计算机科学界：

- 算法需要强调的是那些为数不多，但优于前者的思路。
- 算法的验证应当考虑到以上各思路的相互影响。
- 算法的运算时间应当易于测定，即使问题的规模会任意增大。

艾兹赫尔·戴克斯彻[②]在第二次世界大战后不久就开始了他的学术生涯。当时的人们把计算机看做是工程学上的奇迹、物理学家们的高级工具，但普遍认为它并不适用于真正的数学运算。

---

① 朱莉亚·蔡尔德（Julia Child）是美国的知名厨师、作家和电视节目主持人，曾登上过《时代》杂志封面。

② 艾兹赫尔·戴克斯彻（Edsger W. Dijkstra，1930—2002），荷兰计算机科学家，早年钻研物理及数学，而后转为运算学。1972年获得图灵奖，1980年获得首届计算机先驱奖，1989年获得ACM SIGCSE计算机科学教育教学杰出贡献奖。参见本书第4章。

PDG

戴克斯彻耗费多年才终于发表了他的寻求最短路径算法，部分原因在于当时的数学家们正醉心于无穷大的数理难题，对于那些任何人花费有穷时间就能解决的问题，他们并不认为有多少改善的必要。

随后，戴克斯彻开始研究如何避免两个程序因共用数据而导致程序冲突、损坏的问题。这也是软件设计的根本性问题之一。为了强调这一问题的重要性，他直面工程学上的挑战，完善了键盘与计算机之间的交互机制。他晚年的研究涉及判断程序运行正确与否的“验证”（prove）技术，这一方法即使在今天也并不普及。作为一位永不趋附于学术潮流的科学家，戴克斯彻以其良好的品味和自信不断影响着其研究领域。

40余年来，迈克尔·O.拉宾一直在算法开发领域扮演着核心的角色。早年他与达纳·斯科特[1]的合作奠定了计算机语言处理的基础，后来开始研究随机算法（一种出错概率极低而效率却极高的算法），又推动了密码学和网络计算的迅猛发展。拉宾以这样的方式来概括随机算法技术：“抛一枚硬币，然后明智地运用结果。”

高德纳[2]则将数学的精确性带入到算法当中。举例来说，如果一个问题有5种解决方法，高德纳会向我们分别解释在哪种前提下哪一种方法最好。他孜孜不倦、持之以恒，著有篇幅浩繁的经典教材，囊括了计算机科学中所有重要的结果。高德纳的发明成果包括LR(k)分析算法，使现代的语言翻译器能够将高级编程语言转换为机器语言。

莱斯利·兰伯特[3]年轻时花了很长时间学习物理学，思考狭义相对论以及时间与空间中的事件排序等问题。这种深入的理解让他对当时刚刚兴起的分布式系统（由多个处理器联网构成的统一系统）抱有独特的观点。对于分布式系统中的事件，兰伯特建议只考虑它们的本地时间顺序，因为如果要强调全局时间，就需要保持时钟的完美同步，或者经常向时钟服务器发送讯息。随后他提出了时钟近似同步的算法。兰伯特建立的模型和算法为日后网络空间的基础性架构奠定了基础。

罗伯特·E.陶尔扬总是喜欢画一些圆点，然后在它们之间连上直线。在写写画画中，陶尔扬发明了多个算法，内容涉及平面路网的铺设、网络最佳流量的计算以及计算机的历史快照存储，同时也把对优美的追求带入了整个算法分析领域。他巧妙地借用了一些不相关的概念，例如投资领域的分摊以及经济学中的竞争力，为整个世界设立了一套精确判断算法质量的新标准。

荒野中迷路的旅行者在听到有人说“从这儿不能到那儿”时，都会停下来寻找方向。他可能会依靠手中的地图来解决问题，因为他凭直觉知道，总会有一条路通往自己的目的地。计算机科

---

① 达纳·斯科特（Dana Scott，1932—）的研究领域包括计算机科学、数学和哲学。他与拉宾合作提出的非确定自动机概念为他们二人赢得了1976年的图灵奖。

② 高德纳（Donald Ervin Knuth，1938—）是现代计算机科学的鼻祖之一，在计算机科学及数学领域甚至计算机排版方面广有建树。他的英文名直译为唐纳德·欧文·克努斯，“高德纳”是他的中文名字，取自1977年访问中国前夕。计算机科学界的杰作《计算机程序设计艺术》（*The Art of Computer Programming*）即为他所著，他也因此获得1974年的图灵奖（时年36岁）。参见本书第6章。

③ 莱斯利·兰伯特（Leslie Lamport，1941—）1960年毕业于麻省理工，后在布兰戴斯大学获得数学系硕士及博士学位。目前他任职于微软研究院。参见本书第8章。

学家们也常常处于类似的境地，但他们却发现自己不能像旅行者那样断然确定。史提芬·古克[1]和利奥尼德·列文[2]定义了一系列涵盖电路设计、逻辑、资源配置、日程安排等多方面的有趣问题。他们指出这些问题要么都很难解决，要么都很容易解决，但就是不知道到底是难还是容易。而且所有人都不知道。设想一下这样的情景：茱莉亚·蔡尔德正在描述一道她不知如何烹调的菜（甚至根本不确定能否烹调）。

① 史提芬·古克（Stephen Cook，1939—）是计算复杂性理论的重要研究者，他开辟了NP完全性的研究，并因此于1982年获得图灵奖。参见本书第9章。

② 利奥尼德·列文（Leonid Levin，1948—）是苏联计算机科学家，1978年移居美国。他与古克同期开始各自独立地研究计算复杂性理论。参见本书第9章。

4

艾兹赫尔·W.戴克斯彻 可怕的说明文和最短路径

> 我曾问我母亲（一位数学家）数学是不是一门很难的学科。她告诉我，你要做的就是学会所有的公式并且应用自如。另外要记住，如果你需要5行以上来证明某件事，那么你的方法肯定错了。
>
> ——艾兹赫尔·W.戴克斯彻

和时装设计一样，科学也有自己的潮流。只有少数人敢于忽视潮流，潜心钻研学科中的基础性问题。这些人其实下了很大的赌注，而其中的成功者往往能赢得批评的权利。回首从20世纪50年代开始的整个学术生涯，艾兹赫尔·W.戴克斯彻赌得很成功，批评得也很严厉。他关于最短路径算法和互斥的研究带有古代欧亚大陆特有的高雅和简洁，希望能和全世界分享。

戴克斯彻1930年出生于荷兰鹿特丹的一个科学家家庭，父亲是化学家，母亲是数学家。从小他就表现出对于科学的爱好与天赋。

> 我的姐姐有一套麦卡诺拼装模型[1]，就是那种有长长的钢条、上面带很多洞的玩具。我用它做出了很多机器。我记得搭建过两台很拉风的吊车，其中一台不管吊起什么东西，它的重心一直都会保持在支点正上方。另一台当吊臂中心和负载之间的距离改变时，负载能一直维持在同一高度。

1942年，12岁的戴克斯彻进入鹿特丹当地的精英学校伊拉斯米亚诺姆高中（Gymnasium Erasmianum），接受了传统的荷兰教育——学习古希腊文、拉丁文、法文、德文、英文、生物、数学、物理和化学。战争让荷兰百姓陷入了苦海，戴克斯彻和他的家人也不例外。德国纳粹的侵占即将结束时，食物严重短缺，他的家人把他送出了城外。

> 我父亲一位朋友的朋友带着我一起走。他的车当时还在。我们往乡下开。没有汽油，

---

① 麦卡诺（Meccano）是英国麦卡诺公司出品的一种儿童拼装玩具，在中国也被称为“积铁”。游戏者可以通过螺钉将小型钢条、钢板、滚轮等部件拼装到一起，搭建车辆或建筑。钢条和钢板上布满圆孔以便螺钉穿过，因此组装的自由度很大，利于培养儿童的创造力、空间感和动手能力。20世纪80年代以前出生的读者对此玩具应该有印象。

PDG

车只能烧甲烷……后来水箱也破了。天气寒冷刺骨。我当时只有 14 岁，而且非常虚弱——心跳每分钟不超过 40 下。

年轻的戴克斯彻在1945年7月与家人重聚。战争即将结束，政治理想主义空前高涨。戴克斯彻打算学习法律，以后到联合国为自己的祖国服务，但他的父亲劝阻了他。

我被说服不去念法律——毕业时我数理化三门课都考得非常好。

于是他进入了荷兰莱顿大学，只能在物理和数学间选择一下专业。

我决定，如果在大学里不学物理，以后可能就再没机会了。我觉得数学什么时候学都行。

戴克斯彻选择了理论物理学专业，他发现该领域有许多问题都需要大量复杂的计算，因此决定学习编程。由于第二次世界大战中积累的密码破译经验，英国在20世纪40年代至50年代的计算机发展领先于欧洲其他国家。戴克斯彻于1951年前往剑桥参加了一个编程暑期培训班。1952年3月他在阿姆斯特丹的数学中心（Mathematical Centre）找到了一份兼职，逐步踏入了计算机编程的世界。

数学中心驻扎在一所老学校里面。有台名为 ARMAC 的计算机占据了一整间教室。它用磁鼓作为存储器（一种可以旋转的铝制鼓筒，外表面涂有磁性材料，通过记录磁头存取数据）——完全领先于当时的业界标准。

20世纪50年代初，在Fortran和Lisp出现之前，由于计算机的设计彼此不同，编程人员写程序时不得不考虑到每台计算机的具体情况。一般程序员都会拿到一份对应机器能够执行的指令清单。如果硬件方面能够简化设计，就算将加大编程的复杂度，他们也会举手欢迎。不过，与戴克斯彻一同工作的硬件设计师却反其道而行。

他们往机器里添加任何东西都得经我允许。当时我在写功能规范，也就是机器的使用手册。他们称为“可怕的说明文”——它就像法律文件一样严格。

戴克斯彻尚未决心献身于编程事业。在当时的荷兰，基本上还没人听说过这一领域。

我尽快完成了莱顿大学的学业。物理是一门非常值得尊敬的、理性的学科。我向范·维恩加登[①]道出了我对做一名程序员的犹疑。我说我怀念物理学的基础和理性。他同意在当时编程的确还算不上是一门学科，但接着又告诉我，既然自动计算机已经出现，这正是一个崭新的开始，我为什么就不能用自己的努力让编程在未来成为一门备受尊敬的学科呢？

戴克斯彻的疑虑反映了当时人们对编程的普遍忽视。当他与未婚妻（他的同事，也是一位程序员）办理结婚登记时，死板的官员并不认可程序员是门职业，他只好郁闷地在职业一栏填上了“理论物理学家”。

① 范·维恩加登（Adriaan van Wijngaarden，1916—1987）是荷兰早期计算机先锋人物，阿姆斯特丹数学中心计算部主任，1986年计算机先驱奖获得者。他可以说是戴克斯彻的导师。

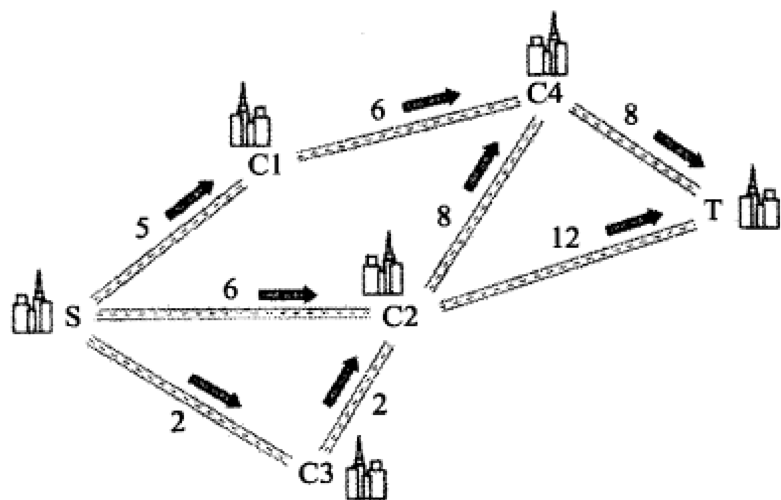
最短路径

戴克斯彻留在了数学中心。过了一段时间，他受命在即将召开的1956年国际数学会议上演示ARMAC的运算能力。以此为契机，他开始思索一个问题——如何在铁道路线图上判断两点间的最短路线。一个阳光明媚的星期六上午，戴克斯彻和妻子坐在一家咖啡馆的阳台上呒着咖啡聊天。突然间他陷入了沉默。

> 我进入了沉思状态。我的妻子对这种情况见怪不怪了……我突然发现这个问题是如此地简单，不需要纸笔就能得出答案。

图4-1表现了戴克斯彻思考的问题（用公路代替了铁路线）。任务是找出驾车从S市到T市的最快抵达路线——最短路径。在阅读图例文字或下一段之前，读者们不妨先试着自己来解决这个问题。

戴克斯彻的基本方法是：在出发地S市和目的地T市之间，建立一个途经城市不断增多的“核心集合”(core set)。在算法的任一给定步骤中，我们都能知道驾车前往核心集合中任意一个城市的最短时间。在一开始，核心集合只包含S市，驾车抵达所花的时间为0。在随后的每一个步骤中，我们都能在核心集合之外找到一个城市X，从S市到该地所花的时间要少于到其他核心集合之外的城市。由于任何路线驾车都至少等于0分钟，X必将与核心集合中的某个城市直接相连，我们称为Y。那么由S市驾车抵达X的时间就等于由S市到Y的最短时间（这一时间我们已知，因为Y在核心集合内）与由Y到X的时间之和。此时我们再将X添加进核心集合，并且把计算出的时间记录下来。如果X等于T市，则计算结束。图4-1显示了在算法的每一步中，核心集合是如何增长的。



最初核心集合只包含S。
之后，添加C3，总数为2。
之后，添加C2，总数为4。路线为S → C3 → C2。
之后，添加C1，总数为5。
因此当前核心集合包含S、C1、C2和C3。
之后，添加C4，总数为11。路线为S → C1 → C4。
最后添加T，总数为16。路线为S → C3 → C2 → T。

图4-1　戴克斯彻的最短路径算法

> 这是我第一次自己提出并解决的图形问题。最神奇的是我并未发表过它。因为这个问题在当时一点都不神奇。那时候，很少有人认为算法属于科学论题。
>
> 当时的数学界几乎全都在研究连续统（continuum）和无穷大问题。我这个不连续又有穷的小问题很难引起任何人的兴趣。在一个有限的图形上，从一点到另一点的路线数量显然是有限的。每一条路线的长度也是有限的。你要找的是一个有限集合的

最小值。任何一个有限集合都有最小值——这毋庸置疑。这个问题没有受到数学界的尊重。

我没有接受专业的数学教育，这点曾让我遗憾了很多年。不过我很高兴自己没有遭遇当时数学上的偏见。

后来，最短路径算法（也就是今天所称的戴克斯彻算法）被广泛应用于道路建设、通信网络路由、飞机航线规划等任何需要寻找最佳行进路线的问题。很快戴克斯彻又对算法进行了些微改动，用于解决另一个相关的实际问题。

硬件设计师卢普斯彻（Loopstra）和施赫恩（Scholten）当时是ARMAC的工程人员，他们正在构建下一个版本的计算机，需要让电流通过所有的基本电路，同时还得节俭，尽量少用昂贵的铜线。戴克斯彻解决了这一问题，出于技术原因，他将其方法称为最短子分支树算法（shortest sub-spanning tree algorithm）。

这样我就有了两个不错的图形算法。当时仍然没有主流报刊愿意发表，所以我最后只好发表在《数字数学》（*Numerische Mathematik*）的创刊号上。很明显这个算法连数字数学的边都没沾上。

这篇论文在当时非比寻常。首先，它以一种高效的方式解决了一个有穷问题。其次，它的结果经过了仔细的证明。

当时几乎所有的数学家都在从事教学工作，很难看到有工业数学。此外普遍认为数学证明主要针对的是理解力强的读者，所以可以有所保留，至少数学家们这么认为。只发表证明的梗概，是当时标准的证明发表方式。

很明显，在与早期计算机相伴的那几年中，我发展出了一套质量标准，一套完全不同于标准数学文化的价值体系。

它强调的是简洁性、完整性和正确性。从我那份可怕的说明文开始就是这样，使用手册应该面面俱到，应该完整且清晰。机器可不懂得什么体谅，它们执行程序不会加任何修饰，该怎样就怎样。说一套做一套的自由在这里是不被允许的。

临界区问题

戴克斯彻对互斥和协作顺序处理的创新始于20世纪60年代初，使用的是ARMAC的继任者，X1以及随后的X8。从硬件方面来看，它们是当时的典型设备。

X8 有一个巨大的磁芯存储器，指令执行周期达到了 10 微秒（比我们今天的 RAM 要慢 1000 倍）。它有一个红色的按钮，按下去机器就会停止运行。按绿色的那个按钮，就会再次启动。

相比之下，软件方面的某些趋势开始形成。戴克斯彻排列了计算机外连设备的彼此顺序，使它们在与计算机交换讯息时能够一步一步交替进行。在计算机术语中，这叫做通信顺序处理。他开始思考如何让这些处理过程彼此协作，或者说同步，“这样我就能厘清它们了”。

程序员经常会面临这种问题。假设有两个过程在同一时间读取了同一数据，那么其中一个过程就有可能会修改这段数据，导致另一个过程出错。要避免这种不良行为，就得在一个过程读取该数据时，禁止另一个过程读取。这就是戴克斯彻所说的同步。

戴克斯彻再次想到了火车——这次是铁路信号系统，也被称作为摇臂信号灯。假设在X市和Y市之间有两趟列车，一趟从X到Y，另一趟从Y到X。如果在路线的某个部分两条铁轨发生交汇，那么两个方向的列车就必须共用一段轨道。为了避免发生事故，工程师利用摇臂信号灯来确保在任何时刻都只能有一趟列车通过这段共用区域。信号灯每次只会对一个方向亮绿灯，并且在有车通过时不会中途改变颜色。通过这种方法，在任意时刻就只可能有一趟列车使用该段轨道。这种方法叫作“互斥”（mutual exclusion）。

戴克斯彻把互斥的概念用到了计算机和键盘之间的通信上。这两个设备通过存储器中一个叫做缓冲区（buffer）的通信区域交换信息。基本的原则是，两者中一次只能有一个读写缓冲区。

> 我意识到键入设备（键盘及其电路）和计算机之间的联结是完全对称的关系。在缓冲区满时我们让计算机进行等待，在缓冲区空时让键入设备等待。这样我们就得到了一个逻辑上的对称。我记得当时我们感觉这是一种解放，非常令人振奋。
>
> 这项技术要解决的问题是，参与协作的每个单元都有其自身的速度和时钟。我想做到的就是对协作进行排序，使其不受相关的速率比例的影响。这主要是出于安全考虑。

1961年，戴克斯彻想出了一种方法，用类似铁路信号灯的两种操作P和V来代表必要的协议。P是passeren的首字母，在荷兰语中表示“通过”，而V是vrijgeven的首字母，意思是“释放”。尽管计算机科学界主要使用英语，如今的计算机设计者们依然沿用了这些字母，足以证明这一理念的重要性。正是由于对清晰逻辑的执著追求，戴克斯彻才最终作出了如此优美的解答。

> 这项发明并不只是P、V操作那么单纯。更大的飞跃在于它忽略了相关的速度，从而让我们对系统的思考可以不受其约束。这可不是什么想当然的结果。我记得这一理念在当时受到了抵触。它太过超前，人们普遍认为不应忽略各设备的速度。

这种看法在今天早已不复存在。通过“测试置位”（test and set）或类似的指令，如今所有的现代处理器和绝大部分存储器板都从硬件上实现了支持P、V功能。这些指令要么锁定某个计算机资源（例如缓冲区）并返回成功，要么检测到资源已经被锁定，并返回失败。IBM的360构架[①]在1964年首批实现了测试置位，为树立该理念的正统性首开先河。

---

① IBM 360系统是美国IBM公司于1964年推出的大型机，整个系列的产品共用一个操作系统。IBM征召了6万名新员工、新建了5座工厂进行研发，是人类工程史上里程碑式的一项大型复杂软件系统开发。

哲学家的晚餐

戴克斯彻看待事物的角度不同于其他人，经常能抓住那些被其他人所忽视的本质性问题，他自谑这是“孤独带来的好处”。1965年，他的这一能力再次显露了出来。

这年秋季的一天傍晚，戴克斯彻对他在爱因霍芬科技大学的弟子们进行了一次在日后非常著名的试验，他将其称为“五人晚餐问题”。但这个问题很快以另一个名字为人所知，牛津大学的教授C. A. R. 霍尔①给它起了一个新名字：哲学家进餐问题。

假设有5位哲学家围坐在一张圆桌周围，每人面前有一碗米饭，碗的两边各有一根筷子。每位哲学家右手边的那根筷子同时也是相邻人左手边的筷子（参见图4-2）。现在，进餐的规则如下：

(1) 每位哲学家先思考一段时间，再进餐一段时间，然后等待一段时间；

(2) 哲学家要想进餐，必须同时使用他左手和右手的筷子；

(3) 哲学家之间只能依靠拿起或放下筷子进行交流（不得交谈或写字）。

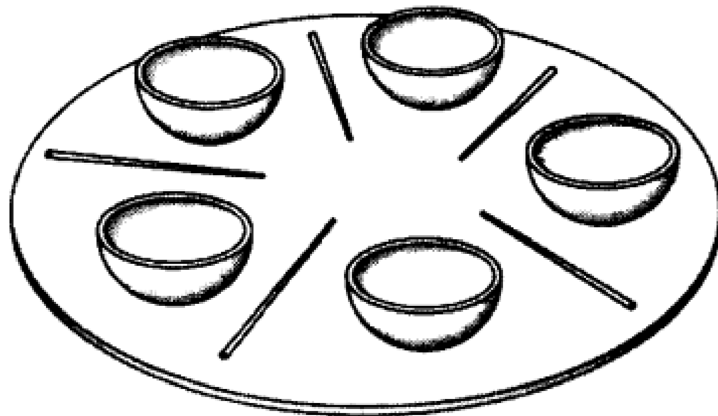


图4-2 哲学家进餐问题。每位哲学家都有一个碗，左手一根筷子，右手一根筷子。要想进餐，他必须同时拿起左手和右手的筷子，也就等于阻止了坐在自己两旁的人进餐。这里的问题是要找出一种方法，让所有的哲学家最终都能吃上东西

假设每位哲学家为了能吃上饭，都运用了如下算法。

(1) 在可能的时候拿起右手边的筷子（如果筷子在右边的人手中，则等待）；

(2) 在可能的时候拿起左手边的筷子（如果筷子在左边的人手中，则等待）；

(3) 进餐。

那么可能会发生以下情况。如果所有的哲学家同时决定开始进餐，他们就都会完成步骤（1），但在步骤（2）一直等下去。这种情况被称为“死锁”（deadlock）。

在看到其他伙伴都只能拿到一根筷子时，某位等在步骤（2）的哲学家也许会放下他右手边的筷子，沉默地坐一会儿，看着他右边的家伙大快朵颐。如果是这种做法，有些无私的哲学家就可能永远也吃不上。这种情况被称为“饥饿”（starvation）。

① C. A. R. 霍尔（C. Antony. R. Hoare，1934—）是英国计算机科学家，开发了快速排序算法和验证程序正确性的霍尔逻辑。1980年获得图灵奖，2000年由英女王授予爵士爵位。

就算是所有的哲学家都吃上了饭，有些人吃的次数可能会比其他人要多。这种情况被称为“缺乏公平”（lack of fairness），正如我们的生活。

计算机网络中频繁出现各种类型的哲学家进餐问题。比如说，局域网中的计算机往往共享一条网线或者广播信道，每次只能有一条讯息被发送。如果所有计算机都在同一时间发送，它们就会全部失败。如果它们立刻再次发送，就会再次失败。这就相当于死锁。如果某台计算机总是有优先权，那么就可能会有另一台计算机饥饿，也可能协议是缺乏公平的。

不管是哲学家还是网络，有一种解决方法就是随机选择，迈克尔·拉宾会在下一章向我们展示。如果某个哲学家或计算机不能获取所需的资源，就等待一段时间，等待时间的长短由某个随机进程来决定（比如闪烁计时器），然后再次尝试。这种机制仍然可能导致饥饿，因为某个进程在随机状态下可能一直都不走运，但这种可能性很小。

在戴克斯彻提出哲学家进餐问题的数年后，他惊奇地发现当时最成熟的计算机系统之一、麻省理工的MULTIX[①]的设计者们居然根本就没有想到死锁问题，导致系统偶尔会突发性死机——就像所有的哲学家们都拿着一根筷子干瞪眼一样。带着些微的讽刺，戴克斯彻沉思道：

> 麻省理工没有注意到某个荷兰小镇上的某位不知名计算机科学家，这也无可厚非。

戴克斯彻在美利坚

在受聘成为美国布劳斯公司[②]的研究员之后，戴克斯彻利用职务之便，开始推行编程的可验证性。但是他投身的学科显然不太受欢迎，有时是出于文化原因，有时则是出于经济原因。

> 应该是在1970年，我第一次在国外进行演讲，题目是设计能够完全控制、证明正确性的程序。我是在巴黎举办的讲座，而且非常成功。在回家途中我到布鲁塞尔一家公司举办了同样的讲座，却彻底搞砸了。我发现管理层根本就不喜欢我的理念。这家公司靠提供维护服务来赢利。程序员更不喜欢我的理念，他们喜欢程序中有一些不确定性，把这当做一种智力游戏，而我的想法剥夺了他们的这一乐趣。他们乐意程序中存在漏洞，因为找出漏洞是个有趣的挑战。

有了戴克斯彻的推动，加上对高质量软件的需求，软件行业的规范化程度开始显著提高。他有一句名言在程序员中流传甚广：“GO TO语句是有害的。”GO TO语句会让程序从处理一件事情突然转到处理另一件完全不同的事情上，而且没有任何掉头回去的计划。带有过多GO TO命令的

---

① MULTIX全名Multiplexed Information and Computing System（多路复用信息与计算系统），由贝尔实验室、麻省理工学院及美国通用电气公司于1964年共同研发，是一套安装在大型主机上的多人多任务操作系统。

② 布劳斯公司（Burroughs Corporation）是美国主流的商务设备生产商，创立于1886年。该公司的发展轨迹与计算机发展史相平行，成立之初生产算术加法机，之后成为大型计算机的主要生产商，同时出品打字机、打印机等相关设备。1986年与斯派里公司（Sperry Corporation）合并，并改称优利系统（Unisys）公司。

程序就像马克思兄弟电影中的法律合同[1]一样容易把人绕晕。

然而对于许多程序员来说，灵感激荡、不遵守教条的自由开辟才是他们认为最理想的方式。戴克斯彻将其视为一种病态[2]。

> 人们总是痴迷于自己不幸的根源——这也是许多婚姻能稳固的根本原因。

深入数学的核心

20世纪80年代初，戴克斯彻举家搬迁至得克萨斯的奥斯汀，他在那里接受了得克萨斯大学计算机科学系的斯伦贝谢百年纪念奖。由于孩子已经长大，夫妻俩喜欢开着自己的大众野营车四处旅行，还为汽车起了个名字叫做“巡回机器”。而在戴克斯彻的思维旅程中，他又回到了数学中继续追求着严格和精确。

> 当时我正在试图改进数学论证：让论证更简单、更清晰。这需要把从编程中得到的经验运用到更广阔的数学领域中。
>
> 众所周知，要想把某件事做大，就必须对其进行分解——类似于某些模块。我们必须能够独立开各个部分……编程的人都知道，这绝不只是劳动分工的问题，因为如果接口选择错误或者不恰当，工作量就会成10倍地攀升——这可不是加法那么简单。
>
> 举个例子。假设有四位住在不同城市的作曲家，决定共同谱写一首弦乐四重奏。一种分工方式是你写第一乐章、我写慢板乐章、他写终曲。另一种方式是你写第一小提琴，我写大提琴，他写中提琴。如果是后一种划分，作曲家们就需要进行大量的沟通。这个例子很好地说明了实用与不实用的劳动分工。程序员必须考虑到这一点。一个优秀的数学理论一定具有实用性劳动分工的所有特征。缺乏经验的理论家的典型表现就是陶醉在自己提出的复杂论证中。

戴克斯彻很反感那种给读者添麻烦的定义。他曾经中止阅读温斯顿·丘吉尔的《英语国家史略》[3]，因为它“不必要地复杂，他提到同一个人却总喜欢用不同的名字”。在戴克斯彻看来，好的定义、精细的论证和想法本身同样重要。

---

① 该典故出自于美国知名喜剧组合马克思兄弟1935年出品的影片《歌声俪影》（*a night at the opera*）中的著名桥段。剧中的骗子想尽办法欺骗一名贵妇投资歌剧院，其中涉及一份虚假合同，里面满是“第一部分之当事人在本合同中特指第一部分之当事人”等绕口的滥竽充数之词，意在讽刺法律文书中冷漠死板、高深莫测的专业用语。

② 这种习惯导致软件开发成本的提高。世界上约有40%的软件项目因此被取消，其余中的70%则因此拖延了开发进度。——原书注

③ 温斯顿·丘吉尔爵士（Sir Winston L. S. Churchill，1874—1965）是20世纪最重要的政治领袖之一，曾两度出任英国首相，在文学上也有很高的成就，曾于1953年获诺贝尔文学奖。《英语国家史略》（*A History of the English Speaking Peoples*）一书的时间跨度从英语民族的诞生直至20世纪初期维多利亚女王逝世，内容以英国历史为主，同时也展现了其他英语国家的诞生和发展。

> 只要是开发新的东西，就要承担相应的义务。这实际上是在开创新的论题。我们必须创建合适的语言，以便讨论这个论题。很多人都未能充分地意识到这一附属的责任。

晚年的戴克斯彻努力为计算机科学和数学建立形式体系，并诞生了一本著作：《程序与证明的形式开发》[①]。书中认为，编写清晰证明的重要性等同于编写程序本身。有关这一论点及相关话题的研究在两次夏季会议中进行了专门的讨论，一次在德国慕尼黑附近，一次在英国格拉斯哥。而这两次会议美国人的缺席都令人注目。

> 美国人对于规范化操作总有一种病态的恐惧，似乎美国的“反数学化”已经存在有100年了。这当然是一个悲剧，因为就在这100年中诞生了数学计算机，对于数学来说正是一项重要的挑战。但是莫名其妙地，这一挑战的数学性质被他们忽略了，就像某些令人不快的政治因素一样。

作为一个纳博科夫[②]式的人物，戴克斯彻就像一位身处牛仔国度的欧洲绅士。也正因为如此，人们对他充满了争议——年轻的计算机科学家们在敬畏他的成就的同时，又不免怀疑他从事的研究是否还有意义。出于同样的原因，戴克斯彻对他身边同僚的研究也存在着疑问。例如，当被问到人工智能应如何划归到计算机科学中时，他挖苦地回答：“没门。”在某种程度上，他把人工智能视作一种美国式的天真幼稚。

> 欧洲人倾向于保持人与机器之间的距离，且他们对两者都不抱有太大期望。

艾兹赫尔·W.戴克斯彻存在着两面。一方面，我们看到的是一位极富创造力的科学家，他对于优秀的问题和解决方法十分痴迷，为科学与计算机的实际应用作出了卓绝的贡献；另一方面，他又是一位绝不忍耐人类愚行的批判者，那支辛辣尖刻的笔（万宝龙牌的）让很多同僚都与他渐行渐远。戴克斯彻坚持认为他是一个很容易理解的人。

> 我永远秉承自己的看法和判断，毫不妥协。

对于希望学习他研究方法的年轻科学家们，他提供了三条黄金法则。

(1) 不要与同事竞争。

(2) 尝试做能力范围内最难的事情。

(3) 选择科学上健康、有意义的研究主题。科学操守上绝不妥协。

---

① *Formal Development of Programs and Proofs*，戴克斯彻著，Addison-Wesley出版社1990年出版。

② 弗拉基米尔·纳博科夫（Vladimir Nabokov，1899—1977）是一名俄裔美国作家，同时也是20世纪杰出的文体家、批评家、翻译家、诗人以及鳞翅目昆虫学家。他童年时在圣彼得堡度过，后辗转于克里米亚、英国、德国、匈牙利、法国，于1940年赴美。1955年他用英语出版了名著《洛丽塔》（*Lolita*）。

PDG

5

迈克尔·O.拉宾
机会的可能性

我们应当放弃寻找绝对正确的结果和答案。

——迈克尔·O.拉宾

计算机都很听话，我们说什么它就做什么，不多也不少。编程语言通常都是在表达命令，例如“do”、“assign”和“begin”，从没有出现过“请”或者“谢谢”。这其中的潜台词是：机器就是你的奴隶，下达命令吧。在这样的环境下，很少有人想到还能让计算机自己进行猜想，甚至随机地行事。

迈克尔·拉宾还能想到更加奇特的事。在对计算机行为进行理论化的过程中，他为计算机科学创立了一整套分支学科。他的研究成果促使数学家和计算机科学家们不得不接受这样一个概念，即有意将某些程序设计成偶尔会出现错误，例如将一个本来不是素数[①]的对象指定为素数。令人惊奇的是，其他计算机界人士都对此表示赞同，而且在密码学、遥控技术和通信系统等应用中，这类程序每天都会运行数百亿次。

1931年，迈克尔·拉宾出生于德国的布雷斯劳（第二次世界大战后划归波兰，改名弗罗茨瓦夫），是一个历史悠久的拉比[②]家族的后代。他的父亲来自俄罗斯，是一位拉比，也是一位博士，在当时著名的布雷斯劳神学院教授犹太史和哲学，并担任院长。拉宾的母亲在17岁时就开始从事儿童文学的创作，拥有文学博士学位。1935年，由于预期到可能来临的危机，他们举家迁至巴勒斯坦。

> 我的父亲在俄罗斯生活过，他见识过反犹太主义，也知道即将到来的极端危险。
>
> 他是一名犹太复国主义者。当希特勒刚上台时他就意识到未来渺茫了。我父亲找到神学院的董事会，提议把学院搬到耶路撒冷，但是那些德国的爱国分子说祖国处于困难时期，他们不能做有悖于国家的事情。

① 素数也被称为质数，定义为：大于1且除了1和自身之外无法被其他自然数整除的自然数。

② 拉比（rabbi）是犹太人中的一个特殊阶层，主要包括有学问的学者或教士，是一群观察生活、思考生活从而获得智慧的人。

抵达以色列港口城市海法之后，拉宾的父亲成为了当地一所高中的校长。小迈克尔则在一所宗教小学念书。

> 有一次，比我大 5 岁的姐姐带回来一本保罗·德·克鲁伊夫的《微生物猎手》[①]。这本书和其他微生物方面的先锋书籍激发了我的想象力，因此从 8 岁到 12 岁时我一直认为自己将来要当一名微生物学家。
>
> 后来有一天，我在教室外面罚站。机缘巧合，当时正好有两个九年级的学生正坐在走廊里解欧氏几何题，我站在一旁观看。他们有个问题解不出来，就以此挑衅我，结果我做了出来。几何纯粹依靠思考、经过证明来创建出关于线与圆的真理，这个过程所具有的美感震撼了我，我完全着了迷。

拉宾努力劝说父亲把他送到莱利学院读书。虽然那是一所模仿德国风格的学校，但专门从事科学方面的教学。他的父亲原本希望他继续念宗教高中，但拗不过儿子。

> 他正确地预计到科学的魅力将会让我远离宗教。我向他保证我不会这样，但事实上正是如此。

20世纪50年代初，拉宾从莱利学院毕业，进入了希伯来大学。当时第一批有关计算机的文章刚刚在以色列发表。

> 我看了 S. C. 克莱尼[②]的一本书，叫做《元数学》[③]。其中有一章讲述了阿兰·图灵的研究成果——主要是可计算性和图灵机的概念。这对我产生了莫大的吸引力。
>
> 阿兰·图灵为可计算性给出了一种“预估性的”定义。所以计算机并不是一门全新的技术，而是以数学和逻辑思维为基础的。
>
> 甚至在当时我就知道自己将来会对逻辑特别是可计算性产生兴趣。但为了丰富经验，我的硕士论文做的是代数，解决了埃米·诺特[④]提出的一个关于交换环[⑤]的问题。

拉宾在日后将对图灵的研究成果进行扩充。

---

① 保罗·德·克鲁伊夫（Paul de Kruif，1890—1971）是荷兰裔美国微生物学家和作家。出版于1926年的《微生物猎手》（*Microbe Hunters*）是他最著名的作品，不仅被无数次再版，而且被各类科普读物推荐列表提到，启迪了众多医学家、生物学家和科学家。

② 斯蒂芬·科尔·克莱尼（Stephen Cole Kleene，1909—1994）是美国数学家、逻辑学家，不动点定理与正则表示法的发明者。他的递归论研究奠定了理论计算机科学的部分基础。

③ 元数学（metamathematics）是一种将数学作为人类意识和文化客体的科学思维或知识，或者说是一种用来研究数学和数学哲学的数学。

④ 埃米·诺特（Emmy Noether，1882—1935）是德国犹太裔数学家，对数学和理论物理都作出了非常重要的贡献。

⑤ 交换环（commutative ring）属于环论，是抽象代数的分支理论。一个交换环指的是乘法运算满足交换律的环。对交换环的研究称为交换代数学。

图灵和可计算性

英国数学家阿兰·图灵（1912—1954）在1935年为"计算"定义了逻辑基础，当时计算机还未出现。他使用了"computer"一词，但在当时这一术语指的是公司雇用的负责执行计算的人（通常还是女性）。

图灵想象了一下这样一位"计算者"将会如何执行任务。计算者会有一支铅笔和一卷无限长的窄纸，上面用竖线分成了一个个的方框。图灵将这卷纸称为"纸带"（tape），将上面的每一个方框称为"方格"（square）。他写道："在任一时刻，计算者的行为由他观察到的符号，以及他当时的'思维状态'所决定。"[1]基于这种观察和思维状态，计算者会在以下三件事中选择其一：移到邻近方格、改动当前方格、进入新的思维状态（有关思维状态我们将在稍后举例说明）。图灵猜测计算者会使用一套数量有限的字母系统，因为他们不可能区分无限多个字母。他还猜测计算者的思维状态的数量也是有限的。

图灵认为，一个计算机程序应当包含一套精确的指令，交付给进行观察的人。比如说"如果当前方格中是0，就将其改为1并向右移动一个方格"。他认为，所谓"可计算性"其实指的就是一个计算者通过这卷纸带能够做的事情。就目前我们所看到的计算机而言，他是对的。

图灵希望解决大卫·希尔伯特[2]在1920年提出的"判定性问题"：希尔伯特想找到一种系统化的方法，能够证明或证伪任何用一阶逻辑这种数学语言表述的命题[3]。图灵通过自己的机器证明了这种系统化的方法是不可能存在的，也就是说，他证明了有一些问题是无法通过计算解决的。

为了寻找灵感，图灵借助于奥地利数学家库尔特·哥德尔[4]的研究成果。此人在1931年（25岁）证明，在算术和欧氏几何中一定存在着一些命题，它们既无法被证明也无法被证伪。

哥德尔的想法是把数字置入到逻辑语句中考虑。例如，逻辑语句S说："S是不可证明的。"对此他进行了如下推导：假设语句S可以被证明，那么S本身就是错误的，因为它的意思与自己已被证明的事实相悖。因此，S是不可证明的，所以它是正确的，但又不可被证明。（这种运用自指句的方法最早可以追溯至克里特人埃庇米尼得斯[5]提出的说谎者悖论。他说："所有克里特人都说谎。"那么，埃庇米尼得斯是不是在说谎呢？）

图灵将这一悖论应用到了他的"人类机器"所使用的程序中。他首先证明了必然有无数个问

---

① 语出自"On computable numbers with an application to the Entscheidungsproblem"，*Proceedings of the London Mathematical Society*，(2)42，1936～1937，第230至267页。——原书注

② 大卫·希尔伯特（David Hilbert，1862—1943）是伟大的数学家、科学家。他发明和发展了大量的思想观念，例如不变量理论、公理化几何、希尔伯特空间等，此外还是证明论、数理逻辑等理论的奠基人之一。

③ 在可计算性理论与计算复杂性理论中，所谓的判定性问题（Entscheidungsproblem，英语decision problem）是一个在某些形式系统回答是或否的问题。例如"给定两个数字X与Y，X是否可以整除Y？"便是判定性问题，此问题可回答是或否，且依据X与Y的值而定。

④ 库尔特·哥德尔（Kurt Gödel，1906—1978）是数学家、逻辑学家和哲学家。其最杰出的贡献是哥德尔不完备定理和连续统假设的相对协调性证明。

⑤ 埃庇米尼得斯（Epimenides）是公元前6世纪的古希腊预言家、哲学家、诗人，传说他在洞穴中沉睡57年，醒来之后获得了预言的能力。

题是无法计算的，也就是无法通过计算机编程来解决。事实上，他证明了数学函数的数量是不可数无穷大，而程序的数量只是可数无穷大。这是什么意思呢？意思是必然还有无穷多个函数是没有程序可以对应的。

图灵接着展示了一个无法由计算机解决的问题，他称之为"停机问题"。问题是这样的：能否为计算机X编程，使其在有限的时间内判断出计算机Y（装有另一个程序以及一些初始数据）是否会停止运行？运用和说谎者悖论类似的推理过程，图灵证明了计算机X不可能用程序来解决这类问题[①]。正如拉宾所指出的，这个问题非常有实际意义。

> 假设一个IBM的经理找到一位程序员说："玛丽，公司需要一种自动化的方式来检测程序是否会停止，我希望你能找到一种算法——一种可计算的方式来解决这个问题。"唔，你看图灵已经证明了这是不可能的。

拉宾在20世纪50年代对这些问题感兴趣时，以色列还没有计算机，就连从事计算工作的人都是凤毛麟角。于是拉宾搬到了美国，先是在宾夕法尼亚大学念数学专业，然后又在普林斯顿大学攻读逻辑学的博士学位。

拉宾成了阿隆佐·邱奇的门生："邱奇是个非常严格的人，让我真正理解了数理逻辑的方方面面。"邱奇同样也是可计算性理论的支持者，他的观点与图灵不同，但有着同样的影响力。

拉宾的博士论文讨论了代数群（algebraic groups）的可计算性问题。"群"是一种基础的数学结构，应用于许多科学领域，尤其是理论物理学。拉宾指出许多涉及群的问题都无法通过计算机解决。

> 图灵定义了计算上可解决与不可解决两者间明确的界限。我的论文与之一脉相承，证明了某些关于群的问题也是不可计算的。
>
> 我们通过特定的方式得到群，希望确定它是否具有一定的属性，例如是否满足交换律。（假设某运算表示为 *，如果对于群中的任意元素A和B，有A * B = B * A，那么该运算就满足交换律。）
>
> 这个问题中，群本身有其描述，因此可以看做是一段程序。这个问题是计算机无法解答的。其他的例子还有很多。

能够进行猜想的计算机

1957年，拉宾还在完成他的论文，IBM研究院为他和另一位年轻的逻辑学家达纳·斯科特提供了一份暑期工作。公司让他们自由从事任何感兴趣的项目，而二人合作成就了计算机科学领域中的一项基本定理。作为工作的一部分，他们提出了一个概念：能够"猜测"答案的计算机。

---

① 如果计算机Y停止运行，将调用一个程序通知计算机X。而此时因为运行了这一程序，计算机Y其实又开始了运行。显然这是一个不可解的问题。

拉宾和斯科特首先考虑的是图灵设想的计算机中的一种受限形式：不允许往纸带上写的计算机，也被称为“有限状态机”（finite state machine）。这种计算机记录从存储器中读到的内容，而存储器的状态集在第一次固定后就无法再改动。

有限状态机在日常生活中的一个例子就是密码锁。如果锁需要按顺序拨动3个数字才能打开，那么它的状态应该分别对应未拨动数字、正确拨动第一个数字、正确拨动第二个数字、正确拨动所有的数字。只有最后的状态才会打开密码锁。在这个过程中任何一次失误都会回到未拨动数字的状态。

有限状态机在计算机应用中用途广泛。比如说，它可以判断哪些字符可以满足单词Br*e C*n，其中的“*”对应任何非空格的字符组合。满足这一条件的单词包括Bryce Canyon以及Bruce Chatwin。[①]

图灵设想的模式带有局限性，这引起了拉宾和斯科特的兴趣：在给定指令集和输入的情况下，机器将总是按照同一方式运行。它的行为是“确定的”（deterministic）。

按图灵的定义，一个“确定的”人类计算者在得到同样的一组输入时，每次都会经历同样的“思维状态”。这种确定性也正是图灵研究工作的基本信条。为了解释 “非确定性”行为，拉宾拿晚餐作为类比。

> 我们假设当计算机处于某个特定状态、读取某个特定输入时，会出现一个选单，其中列出了一系列新的状态可供选择。这样一来，将出现多条路径，针对某个输入的计算也不再单一。
>
> 这就像法式餐厅里的菜单。上面有各种点心、汤和鱼类料理可以作为前菜，然后还有各种肉类作为主菜，如此等等，任君选择。我们选择菜品时，就处于初始状态。先选择点心，然后选择汤，等等。这不是一个随机过程，我并不是在抛硬币决定。但这是非确定的，因为我也可以选择自己想进入的另一个状态（也就是说，选择其他菜品）。
>
> 菜品的选择就是各种状态。每一种选择顺序都代表一种可能的算法，而结果就是满意，或者对餐点不满意。或者接受，或者拒绝。

两位逻辑学家意识到这种选择可能会导致徘徊不决，因此他们指定，针对输入只要有任何一种运算能达到“可接受的状态”，“非确定性”的机器就会“接受”并运行下去。借用拉宾的类比来说，非确定性的顾客会随意选择一些“可接受”的菜品，如果这种选择让他满意，那么他就会说这家餐厅好，反之就会说这家餐厅很差劲。从这方面来看，一个进行猜测的或者说非确定性的计算机似乎只是个智力玩具而已，但有两点除外。

---

① 但这也有限制。麻省理工的诺姆・乔姆斯基在1957年已经证明，有限状态机难以模拟完整的英语语法。他指出，英语中的“if-then”句式可以被任意嵌套，例如“if X is true, then Y is true also”。这里的X和Y也可以是包含有任意“if”和“then”的句子。因此，他表示任何英语语法规则都应识别包含有“if then”、“if if then then”、“if if if then then then”等元素的句式，而有限状态机无法处理这样的系列嵌套。这让他提出了巴科斯曾经用过的上下文无关文法。乔姆斯基同时也指出B. F. 斯金纳的条件性刺激学习模式也不足以教授英语语法，因为从严格意义上说，这套模式和有限状态机的能力相当。——原书注

第一，拉宾和斯科特证明，对于有限状态机来说，任何能够由非确定性机器解决的问题也都可以在确定性机器上解决，但是非确定性机器用到的状态更少。他们还提出了让机器在非确定性和确定性之间转换的方法。

第二，非确定性有限状态机被证明是在语言翻译、文献检索和文字处理程序中表达模式搜索（pattern search）的一种绝佳方式。事实上，我们每次用计算机进行模式搜索时，都在使用拉宾和斯科特的步骤或其变体在文件中寻找适配的文字。拉宾和斯科特用极快的速度完成了这一理论。

> 这是一次非凡的合作……我们俩先有一个人会提出一个问题，然后分别回到自己的角落，另一个人很快就会提出解决方案，也许只需要一晚上的时间。大概在三周之内我们就解决了所有的问题，包括论文中由于篇幅所限未提及的很多结论。这些结论在日后又被其他人重新发现。

有限状态自动机

有限状态机可以处理的模式并不一定得是英文单词，可以是字母、数字或其他符号的组合。由于计算机是一种二进制数字系统，我们用只包含两个符号“a”和“b”的字母系统来说明。

简单的有限状态自动机

图5-1中的自动机接受“ab”这样的输入，但会拒绝其他任何输入。

自动机在一开始处于初始状态。如果输入的第一个字符是“a”，它将会前进到状态2，如果是“b”则前进到状态3。在状态2，如果输入的第二个字符是“b”，自动机就前进到最终状态，如果是“a”则前进到状态3。（两个圆圈表示最终状态，这也是惯例。）如果继续收到输入，自动机就会离开最终状态。在状态3是没有出口的。因此，这个自动机能够接受的输入就只有“ab”。如果自动机接收到输入的最后一个符号，并且停留在最终状态，那么整个输入就是被接受的。除此之外，全部被拒绝。

这种自动机是确定性的，因为在每个状态中，针对输入“a”只有一种转换结果，针对“b”也只有一种。

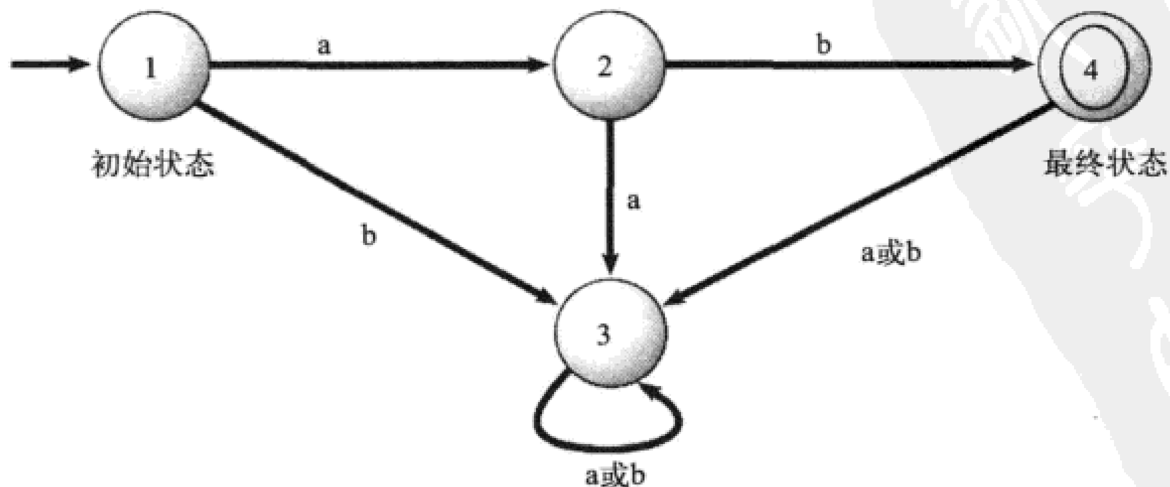


图5-1 接受“ab”输入的有限状态自动机

任意长度的字符串

前面的那个简单的例子只是一次快速的热身。有限状态机也能接受任意长度的字符串。图5-2的有限状态机就可以接受“ab”、“abab”、“ababab”、“abababab”这种形式的字符串。任何其他字符串都会被拒绝。

正如读者所见，这个自动机和之前那个没有太大不同，只不过在最终状态时，针对输入“a”会再次前进到状态2。由状态2到状态4然后再回到状态2的这一路径允许自动机接受任意长度的字符串。比如说，对于“abab”，自动机将由状态1开始，第一个符号“a”时前进到状态2，第二个符号“b”时前进到最终状态，第三个符号“a”时回到状态2，然后在第四个符号“b”时回到最终状态。

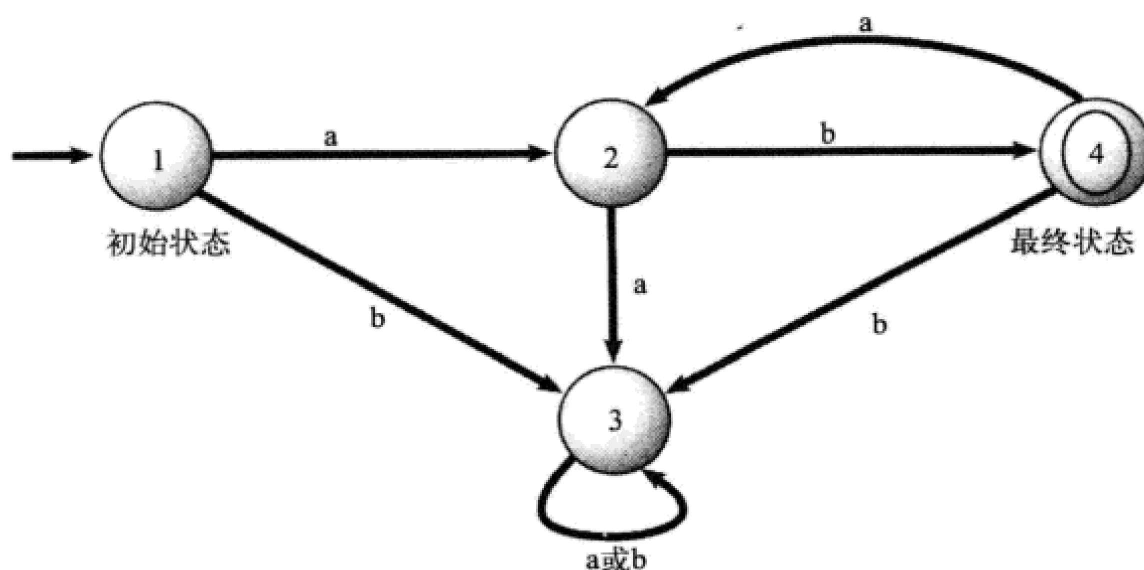


图5-2 接受“ab”、“abab”、“ababab”等输入的有限状态自动机

ab、abab……或abba、abbaba……形式的字符串

现在我们再考虑识别“ab”、“abab”、“ababab”、“abababab”……或者“abba”、“abbaba”、“abbababa”……两种形式的问题。难度在于这两种无限长的形式都始自“ab”，因此自动机必须选择其中一种才能继续运行。如果用非确定性有限状态机就很容易表达这种选择。

在图5-3中，非确定性的选择发生在状态1，也就是初始状态。针对输入“a”，自动机可以选择前进到状态2或者状态3。按照拉宾和斯科特的定义，如果从初始状态能够抵达最终状态，而且第一次转换对应输入的第一个字符，第二次转换对应第二个字符，如此等等，那么这个输入就是被接受的。比如说，“ababab”就是被接受的：路线由初始状态前进到状态2，然后状态4，然后2，然后又到4，再到2，最后再次抵达最终状态4。

再举个例子，“abba”也是被接受的，因为路线由初始状态前进到状态3，然后5，再到6，最后抵达最终状态7。另一方面，“abaa”则是不被接受的，因为这种输入不能从初始状态到达最终状态。

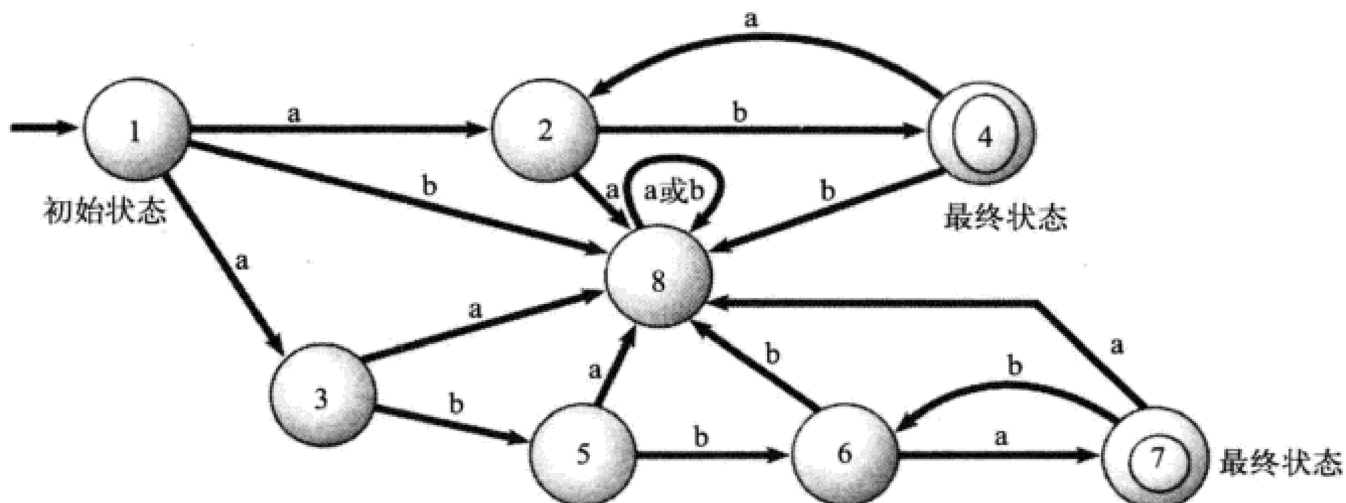


图5-3　能够接受ab、abab、ababab、abababab……形式或者abba、abbaba、abbababa……形式的字符串，但不会接受其他字符串的非确定性有限状态机

非确定性机器转换为确定性机器

非确定性有限状态机很容易写，但怎样对其编程呢？拉宾和斯科特基于集合论(set theory)总结出了一套系统化的步骤。图5-4显示了如何写出一个确定性有限状态机，它能接受"ab"、"abab"、"ababab"、"abababab"……形式或者"abba"、"abbaba"、"abbababa"……形式的字符串，同时拒绝其他字符串。

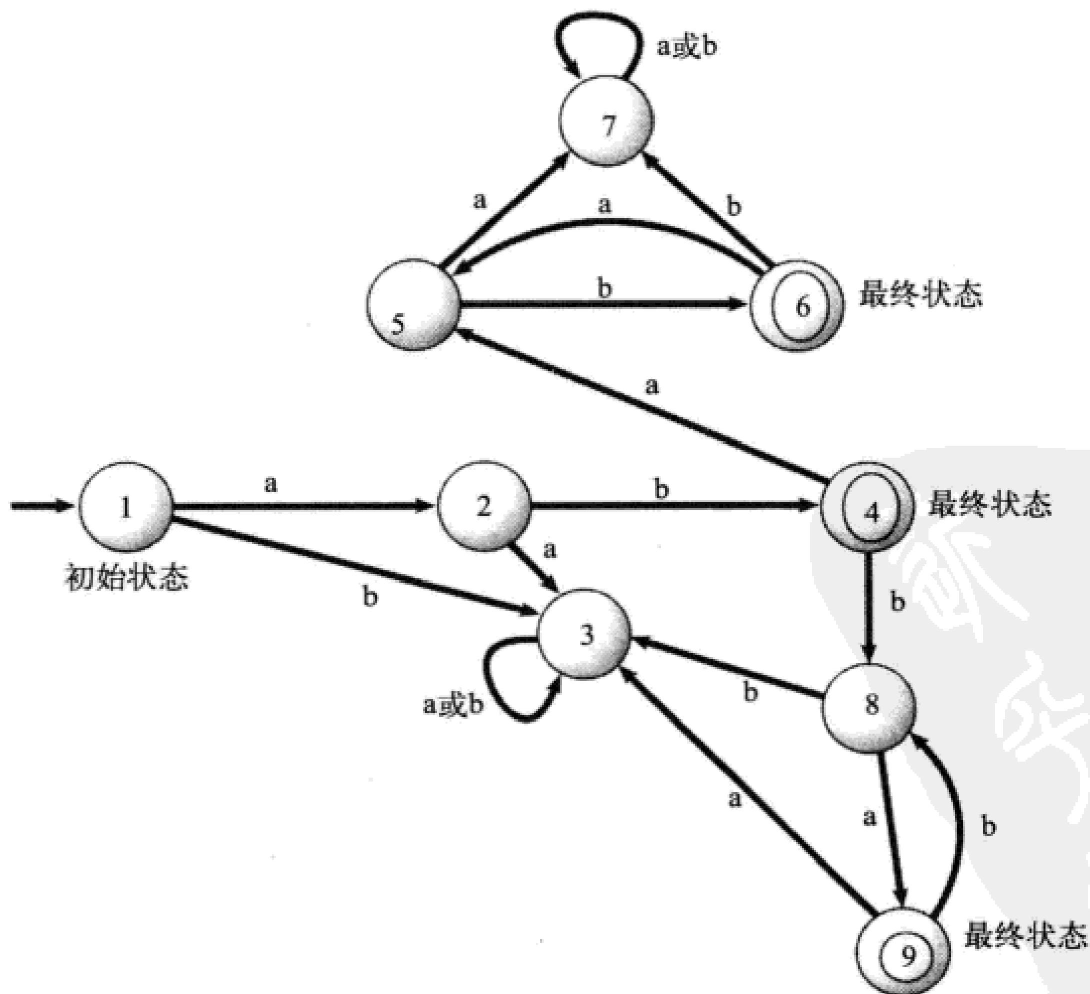


图5-4　能够接受ab、abab、ababab、abababab……形式或者abba、abbaba、abbababa……形式的字符串，但不会接受其他字符串的确定性有限状态机

他们的论文直到1959年才出现在技术文献中。在随后的10年里，理论学家对这一发现进行了多种扩展，但事实证明非确定性这一概念的吸引力远不止此。拉宾自己也很惊讶。

> 必须承认，当时我们并没有完全意识到这一理论所蕴含的意义。对于我们来说，非确定性的概念只是一种数学上的发明而已。在那之后，我在业界参与过许多咨询顾问工作。我会坐在那里听系统工程师，例如操作系统专家们讨论某个操作系统的设计，会有人说："我希望你来判定这个到底是确定性的还是非确定性的。"我只能暗自偷笑，因为这又回到早年间那些非确定性机器的数学抽象中去了。

计算的固有难度

1958年夏天，拉宾回到了莱姆庄园，IBM公司设在哈德逊的智囊机构。约翰·麦卡锡当时正在那里努力解决Fortran语言中的表处理问题，并给拉宾出了一道难题：

> 有两个国家处于战争状态，分别向对方国家派送间谍。间谍完成任务后要返回自己的国家，但是当他们穿越边境时，会有被己方守卫枪击的危险。
>
> 所以需要有一套口令机制。假设间谍的素质都很高，能保守秘密。但是边境的守卫经常去当地酒吧喝酒聊天，因此你告诉他们什么，就等于告诉了敌方。
>
> 问题是如何设计一套方案，既能让己方的间谍安全返回，又避免敌方利用从守卫那里得到的信息把他们的间谍混进来。

拉宾用一种叫做"单向函数"(one-way function)的算法得出了解决方案。单向函数可以转化为计算过程。一个方向的计算很容易，但反过来却很难。我们熟悉的函数都不是这样。例如，使一个数变成三倍大就是一个双向的算法，因为很容易从y得到y的3倍是多少，也很容易从z得到三分之一的z是多少。拉宾使用了由数学家约翰·冯·诺依曼提出的一个单向函数。

> 假设数x有100位数字。它的平方值用计算机很容易计算，而且会包含200位数字。在这200个数字中我们取中间的100个，称其为y。如果我给你这个x，你能计算出y。但如果我给你y让你算出x，你就得尝试算出所有可能的x，可能会有很多结果。所以"平方取中"就是一个容易从正面计算却很难逆行的函数。我们通过单向函数就能解决间谍口令问题。

读者是否明白了这其中的思路？（如果你喜欢挑战谜题，可以在继续阅读之前先思考一下。）

拉宾的方案是这样的：假设守卫知道如何计算平方取中，而且他们有一份y值的列表，每一个y值对应一个间谍。当间谍来到边境时，他会给出他自己的那个x值和姓名。守卫会核对这个x的平方取中是否等于y（同时核对该间谍没有进入过任何其他地方）。就算守卫们在酒吧里把手中所有的y值都泄露出去了，冒名顶替者也不可能找到合适的x。

拉宾的解决方案引出了一个更一般的问题，那就是如何去"验证"某个函数（例如平方取中）

是很难逆向的。换句话说，无论采取何种方式，都需要大量的操作。为了证明这一点，拉宾继续把问题一般化，研究任意给定计算任务所需的最小工作量，亦即任务的固有难度。正如拉宾所指出的，这一概念在物理学中非常直观。

> 假设在桌上有一本书，我想把它弄到天花板上去。有很多办法可以做到这一点，例如我可以用手把书拿起来，站到椅子上，然后把它举到天花板；或者设计一个滑轮装置，用线把它固定在天花板上，然后把书拉上去；甚至我还可以做一个电动马达。但不管使用什么方法，必然存在一个完成任务所需的最小工作量，也就是书的重量乘以天花板的高度。我所做的就是为计算任务定义这种固有难度。

为了解释如何应用这一想法，让我们参考一个游戏：一个人心中默想1到1000内的任意一个数字，另一个人提出是非问题来判断这个数字是什么。有一种著名的策略就是先问这个数字是否大于500。如果是，就接着问是否大于750？如果否，就问是否小于250？依次类推。每一个问题都会把可能的集合减少一半。这种策略叫做“二分查找”（binary search）。

二分查找法需要提出10个问题，就能得出确切的数字。你也许会想有没有更快的策略。假设我们选择了一种不确保能将可能的集合减少一半的策略。例如，你可能在一开始就问这个数是否小于100。如果对方答“是”，你就能用少于10个问题找出正确的数字。但另一方面，你也可能不走运，对方给出了否定答案，即这个数字是在100和1000之间。这样我们就能证明，最有保证的策略还是每次把可能的集合减少一半。因此也就证明了最少需要10个是非题才能保证得出正确答案，那么这就是必需的最小工作量。

> 我证明了，无论是什么衡量（难度的）方式，而且无论怎样衡量，总有一些可计算函数是非常难以计算的，需要大量的工作。因此说明，固有难度是一个非常有意义的概念，因为函数或计算的固有难度表现出很大的区分度。

与耶鲁大学的迈克尔·费彻[①]一起，拉宾运用这一理论检视了“匹斯伯格算术系统”，这是一个只涉及加法的数学问题。20世纪30年代初，波兰数学家M.匹斯伯格[②]提出，在只包括自然数相加运算的数学系统中，对状态的真伪判断是可用图灵机计算的。匹斯伯格对运行这些计算所需要的操作数量并不感兴趣，他只关心是否能在有限的时间内得出正确的结果。

拉宾与费彻证明了，在匹斯伯格算术系统中判断状态真伪其实极其困难，几乎无法实现。就算系统里只有100个符号，即使拿1万亿台每秒运行1万亿次的计算机，运行1万亿年也远远不能得出结果。

事有凑巧，正当拉宾思考这一问题时，人工智能界有不少科学家还执着于编写程序验证匹斯伯格系统的原理。

---

① 迈克尔·费彻（Michael Fischer，1942—）是美国计算机科学家，研究领域包括分布式计算、并行计算、密码学、算法、数据结构及计算复杂性等。曾就职于卡内基梅隆大学、麻省理工学院、华盛顿大学和耶鲁大学。

② 匹斯伯格（M. Presburger，1904—1943）是波兰犹太数学家、逻辑学家和哲学家。实际上他是在1929年提出的匹斯伯格算术系统。后死于纳粹集中营。

我受邀赴斯德哥尔摩参加国际信息处理协会并做演讲。我的演讲题目是“人工智能的理论障碍”。

我的演讲当天正逢尼克松宣布辞职（1974年）[①]。我原以为不会有听众出席。在我的演讲开始前，大厅基本空无一人。好吧，尼克松。结果演讲一开始突然所有人都涌进了会场，真是蔚为壮观。

于是我阐述了匹斯伯格算术系统的棘手性——一些人工智能从业者正在编写程序验证这一理论，而我的研究结果是，一句话，绝对无望。

从事匹斯伯格系统研究的人非常担忧。他们对我说他们担心这会影响到各机构对他们提供的资助。

在演讲结束后，人们在麦克风前排成长队向我提问。他们只是想听我说这并不是世界末日。

在拉宾演讲的同一时间，史提芬·古克、利奥尼德·列文和理查德·卡普[②]分别指出了一大批难度稍小，却依然棘手的问题，并称其为NP完全问题[③]（参见后面介绍古克与列文的那一章）。业界弥漫着一种悲观的气氛，似乎计算机科学领域到处都潜伏着那些表面上简单，实际上却很难用确定性方法解决的问题。

拉宾对问题固有难度的研究，为计算机界系了一个“戈尔迪之结”[④]。下一步，他将提出自己的思路，将这个死结劈开，为许多问题做出可行的解答。

抛硬币做决定的计算机

在斯德哥尔摩的演讲中，我提出了一个问题：在固有难度这样一种前提下，我们能做什么？

我的提议是，应当放弃寻找绝对正确的结果和答案。我们应当适当使用随机性以便

---

① 理查德·M.尼克松（Richard Milhous Nixon，1913—1994）时任第37届美国总统，于1974年8月9日因水门事件引咎辞职。

② 理查德·卡普（Richard Karp，1935—）是一位计算机科学家及计算理论家。加州大学伯克利分校教授，美国科学院、美国工程院、美国艺术与科学院、欧洲科学院院士。他在算法理论方面作出了卓越的贡献，1985年获得图灵奖，还曾获得过冯·诺依曼奖、美国国家科学勋章、哈佛大学百年奖章、富兰克林奖章等。

③ NP完全问题中的NP指非确定性多项式时间（Nondeterministic polynomial），它是计算复杂度理论中的一类问题，而NP完全问题则是NP问题中最难的一类问题。如下问题就是一个NP完全问题：“对一个有限数量的整数集合，找出任何一个此集合的非空子集且该子集内整数的和为零。”这类问题如果给出答案，将非常容易验证，但没有任何一个够快的方法可以在合理的时间（即多项式时间）内找到答案，只能逐一将它的子集取出来检验。本书第9章将会谈到这一问题。

④ 戈尔迪是古希腊神话中的一位国王，他在一辆牛车上打了个分辨不出头尾的复杂绳结，并把它放在宙斯的神庙里。神示说解开此结的人将统治亚洲。这就是著名的“戈尔迪之结”。无数智者面对这个死结都无可奈何，直到亚历山大大帝远征波斯时，经过一番尝试无果后，果断挥剑将其劈成两半，戈尔迪之结就此破解。因此戈尔迪之结常被喻作缠绕不已、难以理清的问题。

更快地得到结果，即使以小小的错误率为代价。在当时（1974年），我已经掌握了一些说明这一方法的例子，但仍有些人为和做作的痕迹。

1975年，我趁休假前往麻省理工，得知了加里·米勒[①]的研究发现。他指出，利用一种未验证的假设，也就是"黎曼猜想"[②]，就可以用一般确定性算法来判断很大的数字是否为素数。

所谓"素数"（prime）就是只能被它自己和1整除的自然数，例如2、3、5、7都是素数。数字15可以被它自己、1、3和5整除，因此它不是素数，而是"合数"（composite）。米勒的发现对于数学家们非常重要，因为在此之前，就算是那些公认最好的判断素数的方法，也需要耗费大量的时间。这些方法要想判断x是否为素数，需要检验从1到x的平方根之间的很大一批数字[③]。

拉宾采用了黎曼猜想，并证实了其中的一种概率性状态。这一结果是迄今为止寻找素数的最快方法。

拉宾的素数检验基于一种"见证者"（witness）的概念。例如数字143，如果我们知道有13可以整除143（143等于11乘13），那么可知143为合数。因此13就是一个证明143为合数的"见证者"，另一个因子11也是一样。问题是有很多类似143这样的数字只有很少量的见证者，只凭运气来发现它们是不可取的。

我又设计了另一类型的见证者：与数字n存在某种关系，从而证明n是合数的那些数字[④]。对于合数n来说，我们可以证明在1和n之间至少有3/4的数字都属于这种新意义上的见证者，它们见证了n是合数。

除了运用了米勒的发现之外，拉宾这一新见证者的想法还来源于数论中有关素数密度的理论。最后产生了一种非常简单的算法。

对n是否是素数的随机性检验过程如下。在1和n之间随机抽取，比方说，150个数字。然后逐一核验这些数字是否是新意义上的见证者。如果其中有任何一个数字属于见证者，我们就知道n不是素数。这种方法的新意在于，如果这150个数字都不是见证者，那么我们就宣称n是一个素数。

---

① 加里·米勒（Gary Miller）凭借论文"黎曼猜想和素数检验"于加州大学伯克利分校获得博士学位。2003年获得ACM Paris Kanellakis理论实践奖。

② 黎曼猜想由德国数学家波恩哈德·黎曼（G. F. Bernhard Riemann，1826—1866）于1859年提出。它是数学中一个重要而又著名的未解决的问题。

③ 这些比较好的方法包括：遍历2以上到x的平方根以下的每一个整数，是不是能整除x；如果不能整除，同时也就排除了该整数的倍数，将其从除数队列中删除。如果已知这段区间中的所有素数，那么该方法还可以改进为：遍历2以上到x的平方根以下的每一个素数，是不是能整除x。这些方法看似能够满足日常需求，但对于实际的大素数判断还是无能为力，例如$x=2^{127}-1$是一个38位数，要验证它是否为素数，假设计算机每秒钟能运算1亿次除法，那么方法一要用4136年，方法二要用93年。

④ 例如，数字28的见证者包括2、4、7、14。而拉宾所谓的新意义上的见证者还包括与2、4、7等有乘积关系的数字，即与28不互素的数，例如6、8、10、12、16、18、20、21、22、24、26等。所有这些数字都是见证者，而且数字n越大，见证者越多。

有没有可能 n 其实是一个合数，而我们错误地将其宣称为素数呢？的确有可能。但是要想出现这种情况，就必须在 150 次内每次都避免抽中见证者。但是在 1 和 n 之间见证者的数字最少也要占 3/4，因此这种可能性微乎其微（小于 1 万亿万亿万亿万亿万亿万亿分之 1）。

拉宾认为这一方法需要进行试验，于是他向麻省理工的同事沃恩·普拉特[①]展示了自己的算法，普拉特自告奋勇为其编程。

他编写了程序，我们尝试了一些已知的很大的素数和非素数，返回的结果都是正确的。于是我们开始检验那些迄今为止人力未能达到的数字。

沃恩工作非常努力，经常在他的终端机上熬到很晚。当时是 1975 年冬天，有一天我在家过犹太光明节，家里有很多客人，还有土豆饼什么的。快到午夜时我接到了一个电话。

"迈克尔，我是沃恩。我得到了试验的一些结果。拿纸笔记下这个。"他已经算到了 $2^{400}-593$ 是一个素数。如果用 k 表示所有小于 300 的素数 p 的乘积，那么 $k\times388+821$ 和 $k\times388+823$ 是一对孪生素数（彼此相差 2 的素数）。这两个数是当时已知的最大的孪生素数。我的头发都竖起来了。不可思议，简直是不可思议。

也许只有数学家才会为极大的素数感到兴奋，不过非数学家应该也能够领会到素数所具有的那种普遍品质。它们的存在是如此自然，发现的年代都已无法考证。今天我们的"旅行者号"宇宙飞船[②]早已离开了太阳系，探寻着地外文明，而素数在它们承载的信息中扮演着核心的角色。为什么会有素数？为什么数字越大，其中的素数却越少？为什么很难找出它们分布的规律？除此之外还有无数的问题，每一位数学家都渴望着作出与素数有关的基础性发现。普拉特和拉宾已经作出了表率。而他们提出的随机化方法是否还能发挥更多的作用呢？

在计算机科学的文化中，只适用一次的想法被视为弄巧，适用两次的想法叫做窍门，只有经常适用而且具有普遍性的想法才能被称为技术。随机化方法属于其中的哪一种呢？发现这一方法后不久，拉宾就在卡内基梅隆大学发表了演讲，提出了自己的发现：

结束完演讲，有一些听众围住了我。他们说演讲很不错，但一致认为（只有一个人例外）过于偏颇。他们认为我利用了素数的特殊性质（见证者理论）和几何问题的特殊性质来研究这两个问题的解决方法。因此他们认为，这种方法并不具备可推广性。

唯一与他们持相反意见的人是乔·特劳布（Joe Traub，当时卡内基梅隆大学计算机科学系的主任，计算学理论家）。他说利用随机化理论并允许出错可能是一项创举。

---

① 沃恩·普拉特（Vaughan Pratt，1944—）也是计算机科学的先驱，为搜索算法、排序算法、素性测试等领域作出了基础性的贡献。他毕业于悉尼大学，后在斯坦福大学师从高德纳，在20个月的时间内获得博士学位。

② "旅行者号"（Voyager）宇宙飞船共2艘，由美国航天总署（NASA）于1977年发射。其中"旅行者1号"飞船在2010年已远离太阳超过170亿公里。两艘飞船各自携带了一张镀金碟片，包含了来自地球的图片和声音，有符号和图示说明如何操作这张碟片，并且详细指示了地球所在的位置。信息被组合成一个时间囊，取得这张金碟片的任何一个星际文明，甚至未来的人类，都能还原旅行者计划的信息。

特劳布是对的。如今随机化理论已经在很多领域付诸应用，包括计算几何学（与遥控技术和制造业联系紧密的算法领域）、分布式计算、信息检索、密码学、通信学，甚至是计算机黑客行为。拉宾的一个学生罗伯特·泰潘·莫里斯[①]在哈佛读本科期间，就利用随机化方法在1998年11月在互联网上散播病毒。

在良性科学应用方面，拉宾在哈佛的同事莱斯利·瓦利安特[②]发现了一种随机化算法，在大量并行计算机相连的专用网络中发挥了效果。使用一个网络的最简单方式就是将讯息从源头直接发送至目的地。瓦利安特则证实，让每条讯息先从源头发送到某个随机站点，然后再从该站点发送到目的地，能明显降低线路争夺（堵塞）的几率。这种看似疯狂的机制恰恰证明了随机化方法的威力。

随机化方法最令人激动（同时也最具争议）的应用是在密码学方面，尤其是被称为“公钥加密法”（public-key cryptography）的加密形式。传统的加密形式是私钥加密：发送者和接收者双方知道同一段密钥，发送者用该密钥对信息进行编码，而接收者用同一段密钥进行解码。私钥系统的问题在于发送者必须事先通过某种手段将密钥发送给接收者，而传递者在途中可能会被俘虏、贿赂或敲诈，甚至是迷路。

公钥加密法基于单向函数，就像一扇单面开的门，在效果上类似拉宾早年间用于解决麦卡锡间谍谜题的函数。利用公钥加密法，某人X可以四处传播公钥K1，允许任何人向X发送以K1编码的信息。只有X掌握的密钥K2能解码这些信息。由于K1是基于极大素数的单向函数创建的，因此除X之外无人能通过K1找到K2（我们这样认为）。这种方法不再需要传递者，而且通信的形式远较私钥自然。

美国国家安全局的工作是保护美国政府的机密和破解其他国家的机密。他们只批准了一种加密机制：美国国家安全局与IBM公司合作开发的私钥密码构建方式，被称为“数据加密标准”（Data Encryption Standard）。安全局的参与导致批评家们怀疑他们有能力破解任何符合该标准的加密。

1993年，美国国家安全局推行了一种叫做“Clipper”的新的加密芯片，在设计中留有一个“后门”。每一块芯片都有一个“委任密钥”在政府备案。如果政府想检查某人的邮件，他们就会申请得到这个密钥。政府对此的解释是，他们有权以国家安全和法律执行为名获取私人信息。（1994年，时任AT&T贝尔实验室研究员的马修·布雷兹[③]指出，Clipper编码技术可能会被用于编写执法机关无法破解的密码。布雷兹的发现可能导致Clipper系统被另一个政府提议的系统所顶替。）

批评家们倾向于一种被称为RSA算法[④]的公钥加密法。这种方法用到了拉宾提出的随机化理论。

---

① 罗伯特·泰潘·莫里斯（Robert Tappan Morris，1965—）是历史上第一个在互联网散布电脑病毒的作者，被誉为“病毒之母”。他在1988年散布了“莫里斯蠕虫”，造成约6000个系统瘫痪，给这些用户总共带来约1000万到1亿美元的损失。他自辩编写病毒的起因并不是想造成破坏，而是想测量互联网的规模。由于违反了1986年的《计算机欺诈及滥用法案》，最终被判3年缓刑、400小时社区服务及10 000美元罚金。如今他在麻省理工任副教授。

② 莱斯利·瓦利安特（Leslie Valiant，1949—），英国计算机科学家，2010年图灵奖获得者。1974年获得英国华威大学计算机科学博士学位。1982年成为美国哈佛大学工程和应用科学院教授。

③ 马修·布雷兹（Matthew Blaze）如今是宾夕法尼亚大学计算机与信息科学教授。

④ RSA加密算法是1977年由罗纳德·李维斯特（Ron Rivest）、阿迪·萨莫尔（Adi Shamir）和伦纳德·阿德曼（Leonard Adleman）三位麻省理工同事共同提出的，名称来自三人姓氏的开头字母。它是一种非对称加密算法，在公钥加密标准和电子商业中被广泛使用。

首先，素数检验在这些公钥加密系统中起到了重要作用。第一步需要产生非常大的素数，两个这种素数的乘积就组成了公钥。

RSA 算法产生了两个极大素数 P 和 Q，然后创建 P 乘 Q 的积，称之为 N。N 和其他一些数字组成公钥。如果有人找到一种算法能够快速分解极大数字（也就是根据 N 得到 P 和 Q），那么 RSA 算法就会被破解。但我们不知道这种算法是否存在。所以说这些加密方法的可行性依赖于（未证实的）因式分解的难度极大。

因式分解是否会被破解（还是已经被破解了）？拉宾对此付诸一笑，不愿就这一问题表明态度。

随机性的追求

如今拉宾身为美国哈佛大学计算机科学系教授，同时也是耶路撒冷希伯来大学数学和计算机科学系教授。他每年都在这两所大学之间奔波，而家人则定居耶路撒冷。

他的妻子是以色列司法部国际分部的负责人。一个女儿是律师，另一个女儿是计算机科学家，研究分布式系统和密码学中的随机化理论。

拉宾自己近期的工作是将随机化方法用来（以很高的概率）确保大型并行计算机的可靠性。他的团队包括印度计算机科学家克里希纳·帕莱姆（Krishna Palem）和帕尔塔·达斯古普塔（Partha Dasgupta）以及以色列科学家尧纳坦·奥曼（Yonatan Aumann）和泽维·柯德（Zvi Kedem）。拉宾的研究生涯，无疑已经证明了计算机可以在如此不同的领域内解决如此之多的问题，没人知道他还能带给我们多少惊喜。

我认为，我们对于复杂任务还缺乏充分的理解。比如说，我们对人的记忆如何工作就缺乏充分的理解。如果我和你谈起贝多芬，你马上就能知道我说的是一位作曲家。人们能够构建一种记忆组织方法，或者计算机程序，在一个有限的领域内对姓名分类。

但是，我们的记忆比这要复杂得多。你走在路上遇到一个邋遢的人，于是突然想起上学时那位脏兮兮的同桌。

就算是这种例子也仍然过于简单。我们通过结构来记忆事物。你看到一个满怀心事的人，却联想到以前听过的一个完全不相干的笑话。我们无时不刻不在进行这种思维的跳跃。

我们看到一个人的背影，看到他走路的方式，然后说这个人就是杰瑞。我们几乎从不会出错。至少我自己几乎从不会出错。做到这一点不需要进行太多思考。而这是如何做到的，我们完全无法理解。

我认为，这并不说明人类的思维能力和计算机的能力之间的差别。只不过是我们不知道如何编写计算机程序来做到这一点罢了。

6

高德纳（唐纳德·E.克努斯）逐新趣异一线牵

计算机编程是一种艺术形式，就如同人们谱写诗歌和音乐。

——高德纳

高德纳真的只是一个人么？①他一生发表了150余篇论文，提出了领域内最重要的三个算法。他的经典巨著（如今在写第四卷）《计算机程序设计艺术》对整个行业进行了综述，同时也提出了原创的研究见解。这本书的前几卷已经诞生了中文、日语、俄语和匈牙利语等多个版本。在三十多年的研究生涯中，他还找时间创造出功能强大的排版印刷软件系统，以便进行多样化的写作，主题包括古巴比伦算法和《圣经》诗篇，甚至还有一部小说。在“业余时间”里，他喜欢弹奏自己设计的管风琴。

在其职业生涯中，高德纳受到了公众广泛的赞誉和褒奖，包括1974年获得计算机科学界的最高荣誉图灵奖，1979年获得吉米·卡特总统颁发的美国国家科学奖章。然而高德纳以一种超然的态度面对这些荣誉。当年的图灵奖杯如今已被他当做了盛放水果的果盘。

从艾佛瑞·E.纽曼②到冯·诺依曼

高德纳于1938年出生于密尔沃基。他的父亲是克努斯家族的第一位大学生，最开始是小学教师，后来在一所路德教会高中教簿记学。每周日他还在教堂弹奏管风琴。高德纳继承了他父亲对音乐和教育的理解，尤其是语言风格。

我最感兴趣的东西恰恰是老师们擅长的。我们在语句分析图方面进行了扎实的训练。下课后总有一堆人凑在一起分析诗歌中的语句，其乐融融。

① 这句话出自高德纳自传的开篇。

② 艾佛瑞·E.纽曼（Alfred E. Newman）是美国著名漫画杂志*MAD*的虚构封面人物。他是个长着招风耳、掉了一颗门牙、双眼一大一小的男孩，除了恶作剧之外对世界一无所知。

作为校报的编辑，高德纳发明了填字游戏[1]。他至今仍记得当年在词海中寻找合适题目的乐趣。从小时候起，高德纳就开始赢得荣誉。8年级时，一家糖果生产商赞助了一次比赛，要求选手用其品牌“Ziegler's Giant Bar”（齐格勒巨型棒棒糖）中的字母组成新的单词，组成单词数量最多者获胜。小高德纳决定尝试一把。

> 我找出了大概 4500 个单词，还没有用撇号。如果用撇号我会找到更多。比赛裁判给出的“官方”单词表上只有 2500 个。

他赢得了自己的第一份奖励：一台电视机（在当时属于奢侈品）以及足够整个学校享用的齐格勒棒棒糖。上高中之后，高德纳又获得了西屋科学天才奖的提名，理由是他的“普茨比度量衡体系”。高德纳以他日后研究生涯中标志性的细心和准确，定义了一系列基本单位，例如长度单位普茨比[2]（第26期*MAD*杂志的厚度）、“麻烦程度”单位MAD（即48件事的麻烦程度）以及能量的基本单位whatmeworry[3]。1957年6月，*MAD*漫画杂志花25美金买下了这篇作品，使之成为了高德纳等身著作中的第一篇出版物。不过，他在高中时期最关注的既不是写作也不是科普，而是音乐。

> 当时我认为上大学时应该会主修音乐专业。一开始我吹萨克斯，后来乐队的低音号手出了事故，我又开始吹低音号。我为乐队编了一首曲子，把当时热门电视剧的主题曲都加了进去——《法网恢恢》[4]、胡迪·都迪秀[5]还有百利发乳[6]的广告音乐。我那时可不懂什么版权保护法。

他投身音乐的计划遭遇了改变，因为凯斯理工学院（后来的凯斯西储大学）向他提供了物理奖学金。

> 这个体系向任何有科学天赋的人敞开大门，引导他们进入物理领域。当时是在第二次世界大战战后时期，物理领域中有很多令人激动的发现。

高中时高德纳对数学还没什么感觉，但是在凯斯理工学院，一年级的微积分老师保罗·冈瑟（Paul Guenther）劝说他从物理专业转到了数学。冈瑟在这一阶段成为了高德纳的良师益友。

> 在冈瑟之前我从未见过数学家。他很有幽默感，但无论你对他说什么，他都无动于衷。

---

① 填字游戏是一种经常出现在报纸上的益智游戏，玩家根据题目提供的信息，在纵横方格里填写字母组成单词。填字游戏是否真由高德纳发明，尚待考证。

② 普茨比（Potrzebie）是*MAD*漫画杂志故意错用的一个波兰文词汇，意思是“肆无忌惮地恶作剧”。该词有一种东欧风味，正好适合于该杂志的纽约犹太风格。

③ 语出自*MAD*杂志封面人物艾佛瑞·E.纽曼著名的狗屁不通的口头禅“What, me worry?”。

④《法网恢恢》（Dragnet）是美国NBC电视台1952年推出的犯罪剧集，以纪实手法表现罪案及侦破过程，一度成为同类型作品的范本。

⑤“胡迪·都迪秀”（It's Howdy Doody Time!）是NBC电视台1947年推出的一档儿童电视节目，胡迪·都迪是其中的主角，一个满脸雀斑的提线木偶。

⑥ 百利发乳（Brylcreem）是一种英国男士护发品，1928年推出第一款润发油产品。它的电视广告歌曲风靡一时。

1956年，高德纳有生以来第一次接触到了计算机，那是一台IBM 650，Fortran之前的机器。他通宵不眠地钻研使用手册，并且自学了基本编程。

> 我们从 IBM 得到的手册上面有一些编程的案例，而我发现自己的方法比上面介绍的好得多。所以我猜自己可能有些天赋。其实我不知道的是几乎所有人都能对那些程序加以改进——当时的相关书籍都很差劲。我开始学习计算机时，正是巴科斯忙于 Fortran 的时候。

高德纳的第一个程序是把数分解为素数的乘积，另一个程序则是教计算机玩井字棋[①]。不过这些都只是小打小闹而已。1958年他为凯斯校篮球队编写了一个程序，根据命中率、抢断、失误等数据为每位球员评分。球队教练非常欣赏这套程序，宣称球队在联赛夺冠也有它的一份功劳。《新闻周刊》专门为该程序写了一篇报道，IBM也在公司宣传册上刊登了高德纳与650机的合影。

高德纳被计算机的无所不能深深地吸引住了。事实证明计算机甚至也符合他对音乐的兴趣。

> 数学是一种模式的科学。音乐也是模式。计算机科学在进行抽象、建立模式等方面同样有很多作为。我认为，计算机科学与其他领域最大的区别就在于它不断的跃变——从微观角度上升到宏观视角。
>
> 有许多职业的产生是有明确的需要，人们要找到解决重要问题的方法，比如医学方面的职业。而像计算机科学这样的职业之所以存在，是我们从小形成的思维结构所决定的。
>
> 如果你碰巧属于某 2%的人，那么你就会自然而然地被计算机吸引而产生共鸣。正是思维方式将我们与其他人区分开来。最终，我意识到自己是一位计算机科学家。

1960年，高德纳以最优异成绩从凯斯理工学院毕业，而且校方打破惯例，投票决定同时授予他数学硕士学位。随后他前往加州理工学院，3年后获得数学博士学位。他写了一篇关于组合几何学的论文“有限半视场及射影平面”。

1963年毕业后，高德纳以数学系助理教授的身份加入了加州理工学院教研组，仍然继续追随着对计算机的兴趣。自1960年起他就担任了布劳斯公司的顾问。布劳斯公司（后合并为优利系统）是当时计算机产业的领跑者，与艾兹赫尔·戴克斯彻等杰出人物保持着密切的联系。

高德纳在布劳斯的工作内容包括硬件和软件设计，尤其是对新发明的Algol 60编程语言提供支持。这份工作让他有机会与戴克斯彻当面认识，分享彼此对编译的共同爱好。戴克斯彻和J. A. 祖内维德[②]在1960年8月已经实现了第一个Algol 60编译器。

> 我们见了面，并一直保持书信来往。他最大的长处在于永不妥协的审美品味，我呢，总是意志不坚，摇摆不定。如果他对我说他喜欢我做的某件事，那么他就是真的喜欢；如果他说不喜欢，那么就是真的不喜欢。所以我视他为难得的诤友。

---

① 井字棋是一种纸笔游戏，两个玩家轮流在3×3的格上打自己的符号，以横、直、斜最先连成一线则为胜。

② J. A. 祖内维德（Jaap A. Zonneveld，1924—）是荷兰计算机的先驱人物之一。他与戴克斯彻1960年共同开发Algol 60编译器，发誓项目一天不完成，一天不刮胡子。祖内维德最终如愿刮掉了胡子，戴克斯彻则一直保留了下来。

PDG

在那个时侯，数学和计算机科学之间存在着巨大的鸿沟。在写程序时，你必须来回调整，直到自己认为它能顺利运行。利用数学对程序进行证明——这在当时是一个非常激进的概念，所有人都认为不可能。戴克斯彻的确是计算机程序验证方面最伟大的先驱之一。

计算机程序设计艺术

1962年1月，Addison-Wesley出版社[①]找到当时还是学生的高德纳，邀请他为编译器这一新生领域写一本书。同年夏他启动了这项计划，到1963年秋，他开始让自己在加州理工的学生试读初稿，以测试效果。

到了1966年，我已经手写了3000页的草稿，开始用打字机把它们打出来。我比较了手书和打印页面的文字大小，估计这个3000页会减少到700页。但是出版社说我错了，比率应该是1比1。经过疯狂的会议讨论，我们定下了一个计划：出一个七卷册的系列丛书。

在1966年，举一人之力就可以了解整个计算机科学领域。但是它一直在不断成长壮大。我已经尽全力跟上它的步伐。而现在我发现第4卷（关于组合算法）一卷马上就要超过2000页——第4A、4B和4C卷，估计能在2003年完成。[②]

一个新科博士，挑战如此包罗万象的典籍，本就是一件惊人的事，但它得到的反应却更加惊人：《计算机程序设计艺术》的前三卷成了20世纪70年代初教科书的首选，至今仍频频被用于参考书。它们之所以一直受到追捧，正是因为高德纳对待课题一丝不苟的态度。对提及的每一个理论，书中都会巨细无遗地讨论所有细节。在解释某个算法后，高德纳还会再给出一个程序实例——目的是确保读者不会产生误解。书中严谨与机智并举，而且尽力展现每一条理念下蕴藏的美感。正如纽约大学的编译器设计师埃德·施恩伯格[③]所言："戴克斯彻教我们分辨是非，高德纳教我们分辨好坏。"

编译器

在著书的过程中，高德纳开始了对编译器的研究。"编译器"（compiler）指的是一段程序，

① Addison-Wesley出版社1942年成立于美国新泽西州，出版范围覆盖化学、计算机科学、电子、经济、健康、生命科学、数学、物理、统计学共9大类图书。公司网址见：http://www.aw-bc.com/。现属于英国培生教育集团。

② 截止2009年，《计算机程序设计艺术》已出版的卷册包括：第1卷"基础算法"、第2卷"半数值算法"、第3卷"排序与搜寻"、第4卷"组合算法"。其中第四卷计划包括4A卷（已出版）、4B卷和4C卷及4D卷。按高德纳的大纲，后续3卷应为：第5卷"造句算法"、第6卷"上下文无关语言理论"、第7卷"编译器技术"（第5卷预计2015年完成）。

③ 埃德·施恩伯格（Ed Schonberg，1942—）现任纽约大学计算机科学教授，AdaCore软件公司的联合创办人及副总裁。

能够将一种语言（源）翻译成另一种语言（目标）。源语言可以是任何一种高级语言——包括Fortran、Cobol、C这样的现代计算机语言，甚至也包括文字处理、绘图及电子表格语言。目标语言则是计算机能够直接理解的0和1的序列。在20世纪60年代早期，几乎没人清楚如何进行这种翻译。巴科斯在IBM的Fortran团队历尽艰辛，花了4年时间才写出最早的大型编译器。编译被认为是最困难的学科，编译器编写通常都被排为高年级研究生课程。

> 我对编译器感兴趣，是因为我觉得使用计算机的最高境界莫过于让它们自己写程序。当“计算能被应用于计算”，计算机科学才算真正达到圆满。
>
> 20世纪50年代的程序员用代数计数法在卡片上打孔，然后“喂”到机器里面去。之后各种灯开始忽明忽灭，机器砰然作响——计算机指令就这样被打出来！真是让人叫绝。我无法相信这一切就这样发生了，所以必须弄明白其中的原理。而当我弄明白之后，我发现可能还有更好的方法。

事实上，现代软件工具已经让编译器编写变得越来越容易，因此如今的很多大学都会向本科生提供同样的课程。高德纳在这些工具的开发过程中起到了关键作用。

> 在编译器方面，我最知名的成果就是LR(k)分析法[①]。它是我在完成了第10章的初稿后突然想到的（别忘了，我一开始以为只会出一本书）。当时我刚刚研究完已知的成果，所以很自然就形成了LR(k)的基本理念。

高德纳发现的是一种文法分析的通用方法，即按结构翻译。分析程序的工作是截取一段字串（单词序列）并判断该字串符合哪一种文法规则，以便进行翻译。出于效率考虑，实际中的分析程序是一次通过的。也就是说，它们不会回头修改已经作出的决定。

但决定有时候也很难作出。比如说，在两个英文句子“Flying planes is fun”和“Flying planes can crash”中，前一个句子“开飞机很有趣”里的“flying”是动名词，意为“开飞机的动作”，而后一个句子“飞行中的飞机可能撞毁”里的“flying”是现在分词当形容词，表示“空中的飞机”。这些解释是由“flying”和“planes”不同的语法分类决定的。

为了解决这类句子的分析问题，高德纳运用了一种已有的“前瞻”（lookahead）技巧。通过前瞻，分析程序在面对“Flying planes”时会继续检视字串后面的部分，之后再决定应用何种文法解释。与以前的方法相比，高德纳的算法可以处理更多的语言。

属性文法

在研究了如何分析程序语言的陈述之后，高德纳继续深入，试图找到一种通用的方法来辨明程序的含义。他想找到的是一种优美的公式化表述，就像巴科斯和诺尔所做的语法分析那样。

---

① LR(k)分析法是一种上下文无关的语法分析方法。LR意指由左（Left）至右（Right）处理输入字串，同时再向右前瞻k个符号，以确定应用何种语法解释。LR(k)分析法的识别效率高，从左至右扫描就能发现其中的语法错误，并能准确地指出出错位置，因此被广泛应用。

> 我对巴科斯的成果非常感兴趣。他意识到计算机语言有着美妙的结构，其语法能做到正规兼顾优美。我对此颇为着迷，因为这也是计算机领域中唯一与数学相似之处。
>
> 我希望能找到一个好办法定义语义学，从而与巴科斯-诺尔在语法上的范式相称。可以说属性文法是一种自然产物，因为它吻合句子的直观意思。

在高德纳开始这项工作之前，要想描述编程语言的含义，最简练的方法只有依靠翻译程序，其结果最少也有数千行。这与巴科斯-诺尔对语法优美、简洁的描述形成了鲜明的对比。高德纳想出了一种办法，将解释的规则（他称之为“属性规则”）与文法规则结合起来。

比如说，在$x+y/z$这样的代数表达式中，属性文法在分析树的y/z节点调用了除法指令，然后在$x+y/z$节点调用了加法指令（参见图6-1）。因此，一段计算机程序的含义（本例中也就是将x与y/z的商相加）其实是由它的元部件所组成的，或者说“合成”的。程序语言理论家奈德·爱恩斯（Ned Irons）在1960年曾针对简单语言提出过这一想法，但是高德纳深知只有合成属性仍然是不充分的：复杂语言中的程序处处承载着上下文环境的信息，只有全面地理解这些才能对程序中各个属性名的含义作出精确解释。1967年的一次会谈中，彼得·韦格纳[①]建议从分析树的顶端往下传递信息，同时由下往上进行合成。高德纳最初认为这个想法荒谬可笑，但后来找到了应用它的方法，并随之产生了“继承”属性。现代编译器的很多技术都直接源于这些深刻的见解。

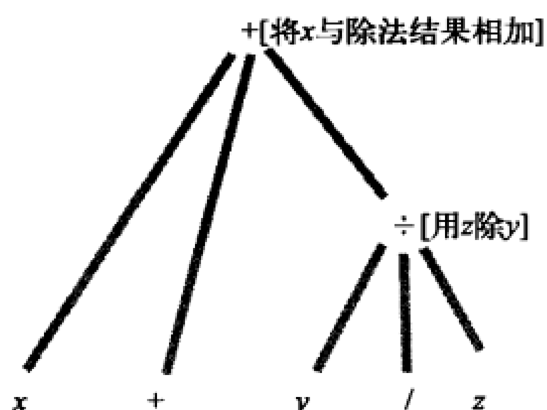


图6-1 $x+y/z$的分析树。合成属性（见方括号）将表达式的含义由分析树的底端传递到顶端

精确分析

与编译方面的工作同步，高德纳已经开始了算法的研究。所谓算法就是计算机完成常见任务的有效方法。在最基础的算法中，有一些是对列表中的数字或名称排序，还有些是在列表中搜寻与某个名称相对应的地址。高德纳在他的《计算机程序设计艺术》中花了整整一卷来讨论这些任务。而他在该领域最主要的贡献，来自于他的“精确分析”（exact analysis）算法。

对于很多计算机科学家来说，能够指出某个算法花费的时间与输入的平方成正比，就已经满

① 彼得·韦格纳（Peter Wegner，1932—）是美国计算机科学家，在20世纪80年代为面向对象编程理论作出了卓越的贡献。

足了；而高德纳则能够明确地证明它花费的时间是输入的平方的3.65倍。

> 我想，我分析算法的方式与业界绝大部分研究者不同。我的研究的独特之处在于能够指出某事比另一件事要好10%到15%。这是品味和性情的问题。我善于观察细节，这是从小接受的训练决定的；而有些人则善于通观全局。
>
> 有些科学家就像探险家，喜欢走出去，在新领土上竖起旗帜；另一些则喜欢在已有的土地上灌溉和施肥，规范结构、立定法规。

不过我们的立法者高德纳同样也进行了不少重要的“探险”，甚至超过了他在编译器方面的成就。例如，高德纳与他在加州理工学院带的本科生彼得·本迪克斯（Peter Bendix）合作发明了一种算法，对数学定理的推论进行了探索。

> 数学家们在抽象结构——群——上建立定理，然后他们证明这些定理的诸多推论。某天我无意中发现计算机可以系统地找出这些证明。你以群的3条定理为开始，导出7个推论，那么你就可以证明这10个结论是“完备的”，即任何由最初的3个定理引申出的恒等式都可以由这10个结论经过一个非常简单的步骤导出。

这个算法的基本理念是从等式形式的定理开始，并将它们视为“约简”（reduction）。比方说，定理是$a \times (b \times c) = (a \times b) \times c$以及$a \times 1 = a$。这二者产生了约简$a \times (b \times c) \rightarrow (a \times b) \times c$以及$a \times 1 \rightarrow a$。一个典型的结论是：$a \times (1 \times b) = a \times (b \times 1)$。基本的想法就是通过证明表达式x和y都能约简到某个“最简的”表达式z，从而证明x和y相等。算法首先对x进行任何顺序的约简，直到无法再约简为止，由此得到z，之后对y也是如此。在上例中，$a \times (1 \times b) \rightarrow (a \times 1) \times b \rightarrow a \times b$，而相似地，$a \times (b \times 1) \rightarrow a \times b$，由此得到结论$a \times (1 \times b) = a \times (b \times 1)$。如果最初的集合是不完备的，克努斯-本迪克斯算法就会生成新的约简。比方说，如果最初再增加一个定理$a \times a' = 1$，那么算法就会自动再生成$(a \times b)' \rightarrow b' \times a'$这样的约简。

从物理学家到社会科学家，科学家们一直都在发明着各种定理系统。克努斯-本迪克斯算法有利于探索各种定理的含义，还能够验证通过这些定理得出的结论——其精确度往往比那些科学家自己的方法要高得多。

1968年，高德纳离开加州理工学院来到了斯坦福大学，该校当时已成为世界三大顶级计算机科研部门之一（另外两者是麻省理工学院和卡内基梅隆大学）。他和研究生沃恩·普拉特一起，发现了一种简单却极端有效的在文本中搜索字符串的方法。大概在同一时间，詹姆斯·莫里斯[①]也发现了相似的方法，因此人们统一称其为克努斯-莫里斯-普拉特算法。

假设要在一长段文本中查找“init”字符串。最明显的方法是逐一检查文本中的字符，如果我们找到i，便检查下一个字符，如果是*n*，则检查再下一个字符，依次类推。这里面有一个棘手的问题：如果找到部分字母匹配，紧接着却发现了下一个字母不匹配，此时应该如何处理。比如

---

① 詹姆斯·莫里斯（James H. Morris，1941—）时任卡内基梅隆大学计算机科学系主任，后任卡内基梅隆硅谷分校院长。

说，如果这段文本是“isininity…”，我们会发现文本的第一个字母匹配，但马上会发现第二个字母不匹配。然后我们会从第三个字母重新开始匹配，发现第四、第五个字母也都匹配，但第六个却不匹配。在此时，直接从文本的第七个字母开始是不对的，因为后面的是“ity…”，我们就会得到一个错误的结论，以为这串字符里面没有“init”。要想解决这个问题，我们得回到第四个字母重新开始，这是因为第三个字母开始的匹配失败了。问题是，这样一来我们就不得不对第四、五、六每一个字母都检查两遍。这种做法的效率很低，尤其是对较长的文本来说，因为我们可能会重复检查很多字符。

高德纳、莫里斯和普拉特想到了一种既正确又高效的算法。他们的灵感来自于罗伯特·波伊尔和G.斯特罗瑟·摩尔的算法理念[①]，以及史提芬·古克提倡的自动机理论。

> 我们的论文事实上准备叫做“自动机理论也很有用”，因为这种算法并不是一个程序员通常能够想到的。对自动机理论的理解给了我们一些重要线索。（论文最后的题目则更为直接：字符串快速模式匹配。）

该算法建立了一个表格，模仿了有限状态自动机的简化版本（参见有关拉宾的章节）。这是一种含有状态和转换的理论化自动机。有些状态叫做开始状态（机器由此开始处理），还有些状态叫做接受状态（机器在此宣称自己找到了某个样本）。在这个简化的版本中，状态对应于字符串中已经匹配的字符数量。因此，对于四个字符的样本“init”，我们从状态0开始，在状态4就宣称有了一个匹配。转换描述了一个状态根据下一个输入字符向另一个状态的行进过程。表6-1显示了自动机在字符串“isininity”中查找样本“init”的行为。

表 6-1

字符	结果状态	字符	结果状态
i	1	n	2
s	0	i	3
i	1	t	4，匹配
n	2	y	0
i	3		

由于第一个字符是“i”，自动机前进到状态1；但由于第二个字符是“s”，于是又回到状态0。大多数算法都能做到这一点。克努斯-莫里斯-普拉特算法的关键之处在于，当找到第一个“ini”、之后却不是“t”时，无需回头再考虑之前的字符。这种避免了重复检查字符的算法在理论方面和实际应用方面都具有着重大的优势。

① 罗伯特·波伊尔（Robert Boyer）和G.斯特罗瑟·摩尔（G. Strother Moore）于1977年公布了他们的波伊尔-摩尔字符串查找算法。

字体

终其一生，高德纳一直都保持着对印刷和绘图技术的兴趣。早在20世纪40年代，当他还是威斯康星州夏令营里的小男孩时，就已经开始用复写纸编写植物指南、绘制花卉样本了，这是当时普遍使用的“印刷”方式。在大学时，他也曾聚精会神地欣赏数学课本上所用的铅字字体。不过他乐于将设计和排版方面的工作留给专业人士。

> 我从未想过自己会参与印刷工艺。印刷是排字工人干的事——滚烫的铅，据说还有毒。但在 1977 年，我听说新的印刷设备是由 0 和 1 组成的——只有比特，没有铅。突然间，印刷变成了计算机科学的问题。我完全无法抵御这种挑战带来的诱惑：我要用这种新技术开发一些计算机工具，用它们来写我以后的书。

高德纳暂停了手上的其他项目，一埋头就是9年，在此期间他为数字印刷设计并实现了两套计算机语言。第一种称为TEX[①]，可以在页面上为字母和其他符号排版；第二种称为METAFONT[②]，可以定义文字本身的形状。如今这些程序在全世界都可以免费通用，有一百多万的拥趸，包括本书的出版社[③]。

高德纳以他一贯的一丝不苟精神探索着印刷领域。例如，他曾写了一篇名为“字母S”的论文，从数学角度详细分析了这个字母形状的变迁历史，还花了几天时间提出一个方程式，声称这样生成的轮廓外形最能让人满意。

编译器、算法分析、字体研究——在高德纳如此广泛的成就之中，我们能否找出一条贯穿始终的主线？他认为答案是肯定的。

> 我们常说需要乃发明之母，这句话并不确切。一个人还得拥有该领域的背景知识。我并不是随走随看，研究自己看到的每一个问题。我解决的那些问题，都是因为我正好有独特的背景知识，也许会帮助我解决它——这是我的命运，我的责任。

他独特的背景知识包括对语言和语法的热爱、在数学方面广泛而又条理清晰的学问、强烈的视觉审美能力、理解事物的意愿以及对编程的热衷。对高德纳来说，最后的两条有着紧密的联系。

> 总的来说，不管你想研究的是什么，只要你能想象自己要对计算机来解释它，你就能发现这一主题中有哪些你还不太了解的地方。这能帮助你提出正确的问题，也是对你所知的终极考验。
>
> 比如说，人们提出音乐理论是为了对什么好听、什么不好听作出客观而非主观的解

---

① TEX是一个功能强大的排版工具，尤其善于处理复杂的数学公式。因此它在学术界十分流行，特别是数学、物理学和计算机科学界。

② METAFONT是一种用于定义矢量字体的编程语言，其中所有的字体都是用几何方程定义的。

③ 即本书原版的出版社，德国施普林格（Springer-Verlag）出版社。

答。我们知道莫扎特的音乐很好听，因为曲目和谐等。但当你要为计算机写出一个程序，让它创造真正优秀的音乐时，你就会知道已有的那些规则是多么苍白。

身为斯坦福大学的名誉教授，高德纳如今把大部分时间用来撰写《计算机程序设计艺术》的组合算法卷。但他也面临着其他的诱惑——管风琴一直在召唤他进行更多的音乐创作。而他对文学的兴趣也可能让他着手开始下一部小说。（他在1974年出版的《超现实数》（*Surreal Numbers*）描写了两个大学辍学生开发出了一套数学体系。）世界范围内的会议和咨询项目也在分散着他的注意力。除了天赋、激情和传奇般的工作习惯之外，高德纳的伟大成就中再无其他秘密。

我一次只做一件事，这就是计算机科学家们所说的批处理——与之相反的就是交换进出。我从不交换进出。

传奇是如何诞生的

民间故事中一直把高德纳视作为有史以来最伟大的计算机程序员。我们不妨参考以下艾伦·凯所讲的轶事。

当我在斯坦福大学从事AI项目时（20世纪60年代末），每个感恩节我们都会与在湾区做研究项目的人们进行一次编程竞赛。奖品是一只火鸡。

麦卡锡为竞赛出题。高德纳参加的那一年，他一举拿下了两个奖项：程序调试所用的时间最少、算法执行效率最高。而且他用的是所有参赛者中最烂的系统，叫做Wilbur系统，只能远程批处理。可以说他把所有人都打得屁滚尿流。

然后他们问他："你怎么这么牛？"他回答说："我学编程的时候，一天能摸5分钟计算机就不错了。想让程序跑起来，就必须写得没有错误。所以编程就像在石头上雕刻一样，必须小心翼翼。我就是这样学编程的。"

PDG

7

罗伯特·E.陶尔扬
寻找优秀的结构

我喜欢想象结构、线图和数据结构。这似乎比其他很多事情要来得轻松。

——罗伯特·E.陶尔扬

在计算机科学界，一个优秀的想法通常具备三个特征：常被用于实践、对年轻计算机科学家具有启蒙价值、能为科研指明方向。就这些标准而言，罗伯特·恩卓·陶尔扬提出的许多想法都堪称优秀。

他在斯坦福大学读博士时与约翰·霍普克洛夫特[①]一同进行了有关平面图测试的课题研究，其成果催生了大批应用，包括更有效的芯片布线和更好的地图编排设计。这些算法凸显了一种名为“深度优先搜索”（depth-first search）的强大编程技巧，现在是每位大学生的必学功课。深度优先搜索同时也被广泛应用于计算机游戏、电子表格和图形程序中，每天都会运行数十亿次。陶尔扬随后与丹·史利特（Dan Sleator）、安德鲁·戈德堡（Andrew Goldberg）针对最大网络流问题进行了研究，其成果帮助设计师规划网络中各条线路的容量，使石油开采、电话呼叫等管道网络提高了传输效率。他还研究了向上树（up-tree）和伸展树（splay-tree）这一对简单的数据结构，为衡量算法效率引入了新的技术。

此外，陶尔扬还与同事一起提出了一种数据结构，能够高效地同时保存当前和过去的信息。如今，这种“持久性”的数据结构在遥控学及数据库系统等方面获得了越来越多的应用。

陶尔扬为计算机科学界的学术文化带来了本质上的改变：先找到合适的“伟大构想”，然后再创造最有效的数据结构对其进行支撑。这一策略引领陶尔扬等人对戴克斯彻的最短路径算法和其他基础算法又作出了重大改进。

纵观陶尔扬的研究成果，不难发现其特点是极简、优美，以及对普遍性的含蓄追求。

优秀的想法总是有办法进行简化，而且能解决原有目的之外的问题。

---

① 约翰·霍普克洛夫特（John Hopcroft，1939—）1961年在西雅图大学获得学士学位，1962年在斯坦福大学获得电子工程硕士学位，1964年获得博士学位。他的研究方向是理论计算机科学。1986年他与陶尔扬携手获得图灵奖。

罗伯特·恩卓·陶尔扬于1948年出生于加州的波莫纳市。从幼年开始，他就对科学十分感兴趣。

> 大概是七年级时，马丁·加德纳[①]在《科学美国人》(*Scientific American*) 上的专栏让我对数学产生了兴趣——玩数学游戏、解智力谜题。在此之前，我感兴趣的是天文学，憧憬着成为登陆火星的第一人。

陶尔扬的父亲是一位儿童精神病学家，专攻儿童智力发育不良，并经营着一所州立医院。还在初中时，陶尔扬就在医院里干点活，对象是IBM打孔卡片校对机——他称之为“前计算机”。1964年，他参加了高中三年级的暑期科学夏令营，第一次与真正的计算机打交道。

> 夏令营的主题是观察一颗小行星，并计算它的运行轨道。我们有机会使用 UCLA（加州大学洛杉矶分校）的一台计算机，于是我学会了 Fortran。
>
> 与此同时，州立医院也引进了一台小型计算机。所以我平时在加州理工念书，暑期就用这台计算机写程序——对病人群体进行统计分析。

在加州理工学院，陶尔扬主修的是数学专业，但他同时也选修了所有的研究生计算机课程。

> 我希望从事计算机科学，因为它是数学的一种更实用的形式。我对人工智能很感兴趣，主要是出于逻辑和定理证明的角度。但是当我到斯坦福大学正式学习人工智能方面的课程时，又感觉这门学科其实相当模糊。

斯坦福大学拥有着一些计算机科学领域最优秀的人才，包括高德纳以及约翰·麦卡锡。

> 高德纳给了我很多启迪。启迪的地方在于他一直专注于具体的分析。他一直对数学的那种精确感兴趣——总是希望得到确切的结果。

另一个启迪来自约翰·霍普克洛夫特，当时他从康奈尔大学获得了一年的学术假期，于是来到了斯坦福做研究。他到达的时候，陶尔扬刚结束研究生院的第一学年，正在放暑假。霍普克洛夫特搬入了与他毗邻的一间办公室，二人很快变成了合作伙伴，最终在1986年携手共获图灵奖。

> 我曾上过麦卡锡的符号处理课程，基本上讲的就是 Lisp 语言。他在课堂上建议学生们去编程判断一个图（graph）是否是平面图。
>
> 当时我对自己并没有什么期望——我只是个努力解决有趣问题的研究生而已。

---

① 马丁·加德纳（Martin Gardner，1914—2010）是美国声名显赫的业余数学大师、魔术师、怀疑论者，他在《科学美国人》杂志上开设了一个长达20多年的数学游戏专栏。加德纳著作颇丰，虽然自己并没有数学博士学位，但是他的作品却能让广大普通读者和数学家都为之着迷。

能够平放的图

图是由节点（node）和节点间的边（edge）构成的集合（在图7-1的图中，小圆圈称为节点，而连接节点的线称为边）。图被用来表现真实世界中的许多不同现象。例如，我们可以用节点表示分子、用边表示分子键，也可以用节点表示人、用边表示交往关系，或者用节点表示路口、用边表示街道。

对某些应用来说，需要保证边与边之间互不交叠。比如说，电路布局中的边（金属电线）如果交叠就可能会导致短路。那么随之而来的问题是，一个图能否做到边互不交叠，同时又维持节点间的连接关系？这样的图叫做平面图（planar）。在图7-1中，图A产生了交叠，但我们能重新布置让它们不再交叠，就像图B那样。

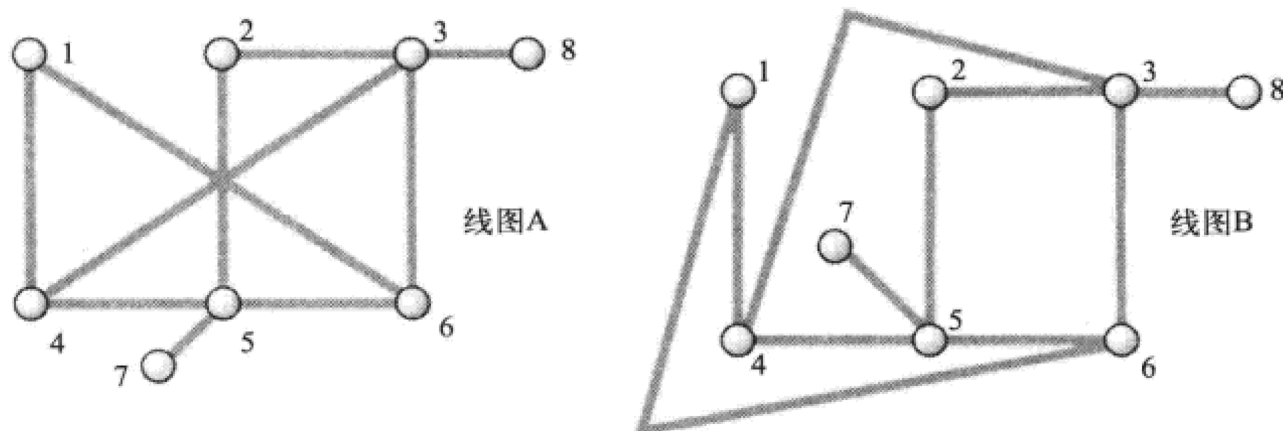


图7-1 一个平面图。我们可以对图A 重新绘制，使它的边互不交叠，同时维持节点间的连接关系，如图B所示

判断图是否为平面图的测试，最早可以追溯至18世纪的瑞士数学家欧拉[①]。他证明，如果节点数为N，且N至少等于3，那么具有$3N-6$条以上的边的图都不可能是平面的。

1930年，波兰数学家库拉托夫斯基[②]证明，任何非平面图都必定包含有图7-2中的某个图这样的连接关系。麦卡锡建议他的学生在程序中运用库拉托夫斯基的条件。陶尔扬很快就发现，以这种方式得到的算法非常低效。库拉托夫斯基测试的确可以转化为算法，但是执行该算法所需的时间与节点总数的6次方成正比。也就是说，判断一个含有100个节点的图，需要执行1万亿个步骤。

> 所以，当时我已经着手考虑平面图测试了。当霍普克洛夫特来了之后，我们开始讨论算法和效率问题。
>
> 霍普克洛夫特提出了一个查找双连接组件的算法，算是不错的开始。但这其实就是一种深度优先搜索。我经过仔细的考虑，决定改进他的算法中的原理，使其更加严密和完善。

① 莱昂哈德·欧拉（Leonhard Euler，1707—1783）是瑞士数学家和物理学家，近代数学先驱之一。他在微积分和图论等多个数学领域都作出过重大发现。此外他还在力学、光学和天文学等领域有突出的贡献。

② 卡齐米日·库拉托夫斯基（Kazimierz Kuratowski，1896—1980）主要研究点集拓扑学和集合论。在平面图方面提出了库拉托夫斯基定理。

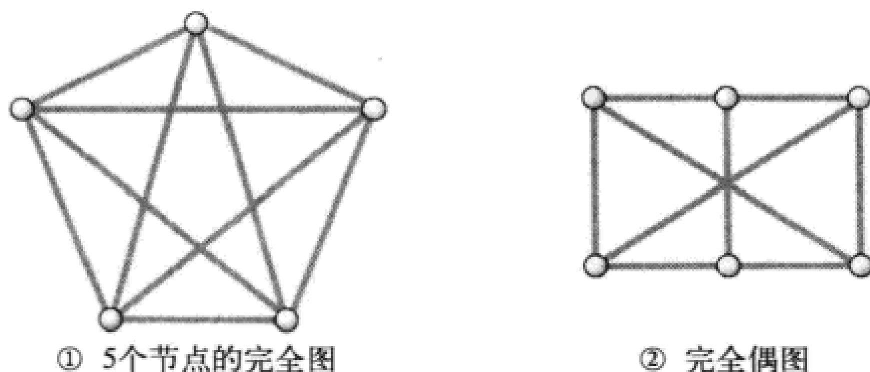


图7-2 任何非平面图都必定包含如上所示的一种结构（子图）

深度优先搜索的应用

深度优先搜索（depth-first search）一直被广泛用于解决人工智能方面的问题，在该领域它被称为“回溯法”（backtracking）。回溯法为解决某个问题探索不同的处理方法，而不会出现重复。在AI领域，它被用于象棋对弈中，根据对手棋子的移动在对应的步法序列中进行系统化的搜索。陶尔扬和霍普克洛夫特利用深度优先搜索对图进行系统化探索。在深度优先搜索中，每一个处理方法都会伴随着它的结论，然后再尝试其他方法。而另一种策略，也就是广度优先搜索（breadth-first），则会同时尝试许多处理方法。采用何种方法取决于问题本身。

以图的某个节点n开始，深度优先搜索会选择一条从n出发的边。这条边通向一个新的节点。一般来说，程序会从刚刚访问过的节点上选择一条之前未曾探索过的边，一直向某个分枝纵深推进，过程中确保每条边都只用过一次。

1961年，L. 奥斯兰德（L. Auslander）和S. V. 帕特（S. V. Parter）提出了一种策略，并由A. J. 葛斯汀（A. J. Goldstein）在1963年加以改进。霍普瓦洛夫和陶尔扬用深度优先搜索来实现这一策略。与库拉托夫斯基的非平面图测试不同，这三人提出的算法尝试将图直接平铺到平面上。

霍氏和陶尔扬的算法直接建立于葛斯汀的算法之上。它的基本步骤如下。

(1) 测试图中的边数有没有超过欧拉条件所允许的数量（如果超过，那么它就不是平面图）。

(2) 将图划分为双连接组件。图的“双连接组件”指的是一系列节点的集合，集合内任何两个节点间都存在一条路径相连通（可能包含多条边），且即使移走集合内的任何一个节点，该两点间仍然存在有路径相连通。如果读者感觉这过于抽象，不妨这样来想：如果城市里任何一个路口施工都不会影响我们开车去常去的餐厅，那么我们的城市就是双连接的。

(3) 利用深度优先搜索在图中找到一个圈。“圈”就是图中由某个节点出发然后又回到该节点的一段路径。

(4) 移走该圈将会把图分成一些互不相连的部分，称为“桥”（bridge）。分别检测每个桥和之前的圈一起是否组成一个平面图。可以对每个桥都执行相同的算法。（这一“递归”过程不会无限重复，因为桥比最初的图要小。）

(5) 如果圈被放置到平面上，每一个桥都必须要么完全在其内部，要么完全在其外部。某些桥可能会两两互相干扰，必须被分别放置在圈的相对面中。

这一算法之所以有很高的效率，关键在于只用了一次深度优先搜索就完成了整个计算，包括对其内部图（正式名称应为子图）的平面性测试。其中涉及的细节极具专业性，就连他们发表的论文，“平面图测试的高效算法”，在排版极为稠密的《美国计算机协会学报》上都占据了整整20页之多。不过读者可以参考图7-3了解其基本的结构。图7-3显示了针对一个平面图的深度优先搜索，所用的图是图7-1中图A的轻微简化版本。实线箭头表示了在图中探索节点的一种可能方法，而虚线箭头则表示那些实线箭头尚未涉及的边。比如说，从4到1、从5到6的边存在于图中，但并没有直接出现在图7-3右侧的树中，所以显示为虚线。这种特殊的边被称作“回边”（back edge），指向树中的祖先节点，而这正是陶尔扬与霍氏的深度优先搜索的众多优点之一。

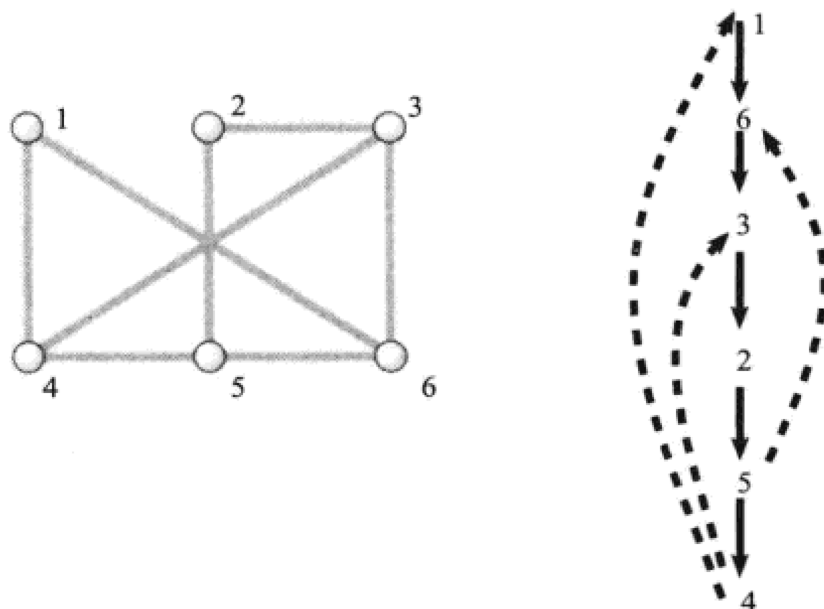


图7-3 深度优先搜索的平面图测试。对图中左侧的图进行深度优先搜索，会生成很多不同的树。右侧图解中以实线边表现的树只是其中的一种可能。左侧图中存在但却不属于这一生成树的那些边则以虚线边表示

从图7-3中的图来看，算法第3步中找到的圈是1 → 6 → 3 → 2 → 5 → 4 → 1。第5步（校验步骤）则观察到还有两个虚线边，4 ⇒ 3和5 ⇒ 6，可以分别放置在圈的内侧和外侧。这表示该图是平面图。

运用陶尔扬与霍普克洛夫特的策略，判断整个图的平面性问题被转化为判断生成树加上回边的平面性问题。解决问题的效率得到了很大提高。

与之前的各种判断方法不同，霍普克洛夫特-陶尔扬算法的运行时间是线性的。也就是说，运行时间与图的大小成正比关系，图的大小翻一倍，解决问题所需的时间也只翻一倍。而反观库拉托夫斯基的判断标准，图大小翻番后所需的时间会增加60倍以上。

> 平面图测试引起了很大的轰动，因为当时每个人都认为这是一个相当困难的问题，所以必然会非常耗时。而我们的测试算法执行起来却快得惊人。这说明只要使用成熟的算法，解决问题的速度就会提高很多。

在20世纪70年代早期，支持这一观点的人还仅占少数。从数学上研究计算的耗时在当时才只经过了十来年的发展。这一领域始自迈克尔·拉宾在1959年的研究，随后经过了高德纳、尤里

斯·哈特马尼斯[1]、史提芬·古克和利奥尼德·列文等人的锤炼。但是，这种根据解决问题需要的操作数量来判断成本的先进理念，还尚未被当时的实际工作者所接受。

相反，许多从业者和学者满足于在自己的计算机上运行自己的算法，然后与那些已发布的算法在老式计算机上运行的速度相比较。这意味着哪怕算法很糟糕，但只要计算机够快，看上去也会比一个在较慢的计算机上运行的优秀算法要好！

霍普克洛夫特与陶尔扬在他们的平面性测试论文中，对同时期的这种野蛮行为表明了自己的看法："对算法运行时间的分析应本着严格、精确的态度，然而人们在这方面的付出少得可怜。与以前发布的算法相比，如今的算法明显低劣许多，而且这种状态一直持续。"

陶尔扬与霍氏提倡的是，对算法效率的衡量应当取决于它需要进行多少基本操作（加法操作、比较操作、边的遍历操作等）。平面性测试算法很快在实际应用方面获得了成功，使其在计算机科学的理论和实践领域都被奉为经典。霍普克洛夫特与贝尔实验室的阿尔·阿霍[2]、普林斯顿大学的杰夫·乌尔曼[3]在1974年合著了一本广为流传的算法书[4]，其中就用这种方式来衡量算法的效率。

平面性测试的另一影响在于，它使深度优先搜索方法获得了业界的认可，并且得到了广泛的应用。在霍普克洛夫特与陶尔扬荣获图灵奖的典礼上，该年度最佳国际象棋程序的设计者就提到他的程序在比赛中执行了4千万次以上的深度优先搜索。

1971年获得博士学位之后，陶尔扬来到康奈尔大学任助理教授。他依然对算法效率保持着浓厚的兴趣。

并查和分摊

到达康奈尔之后，陶尔扬很快开始着手研究并查（union-find）问题。这个问题看似很简单，其实不然。如果能高效地实现，它将会提高很多问题的解决效率。

在很多图形算法中，节点会被分离成不同的集合，称为"区间"（partition）。在算法执行的过程中，不同的区间可能会彼此合并，形成一些更大的区间。

比如说，若指定某个区间内的所有节点都必须连通，那么在该区间内任意两个节点间都必须存在一条路径。

如果算法在某一步骤发现，有一条边连接了A区间的某节点与B区间的某节点，那么这两个区间就必须合并成一个大的区间，其成员是A区间与B区间成员的并集。

---

① 尤里斯·哈特马尼斯（Juris Hartmanis，1928—）是拉脱维亚裔计算机科学家和计算理论学家。他和理查德·斯特恩斯（Richard E. Stearns，1936—）同为计算复杂性理论的奠基人，于1993年共获图灵奖。

② 阿尔·阿霍（Al Aho，1941—）是加拿大计算机科学家，冯·诺依曼奖获得者。他于1967到1991年在贝尔实验室工作，后任计算机科学研究中心副主席。

③ 杰夫·乌尔曼（Jeffrey Ullman，1942—）是美国计算机协会资深会员，高德纳奖获得者。他曾在贝尔实验室工作，后任普林斯顿大学和斯坦福大学教授。

④ 这本书就是《计算机算法的设计与分析》（*Design and Analysis of Computer Algorithms*）。它是数十年来最为广泛引用的计算机科学书籍之一，而且使算法和数据结构被划归为计算机科学学科的重要课程。

图7-4显示了一个例子。最初每个元素都是彼此独立的（参见图7-4a），随后集合{1, 2, 4}成为A区间，集合{3, 6}成为B区间。而最终的区间则是{1, 2, 4, 3, 6}。

每当两个区间合并时，就会有箭头从其中一个区间的"标准节点"（canonical node）指向另一个区间的"标准节点"，通常是从小区间指向大区间（参见图7-4b，从6到2的虚线边）。"标准节点"就是区间中那个没有边发出的节点。这就是并查算法中的"合并"操作。

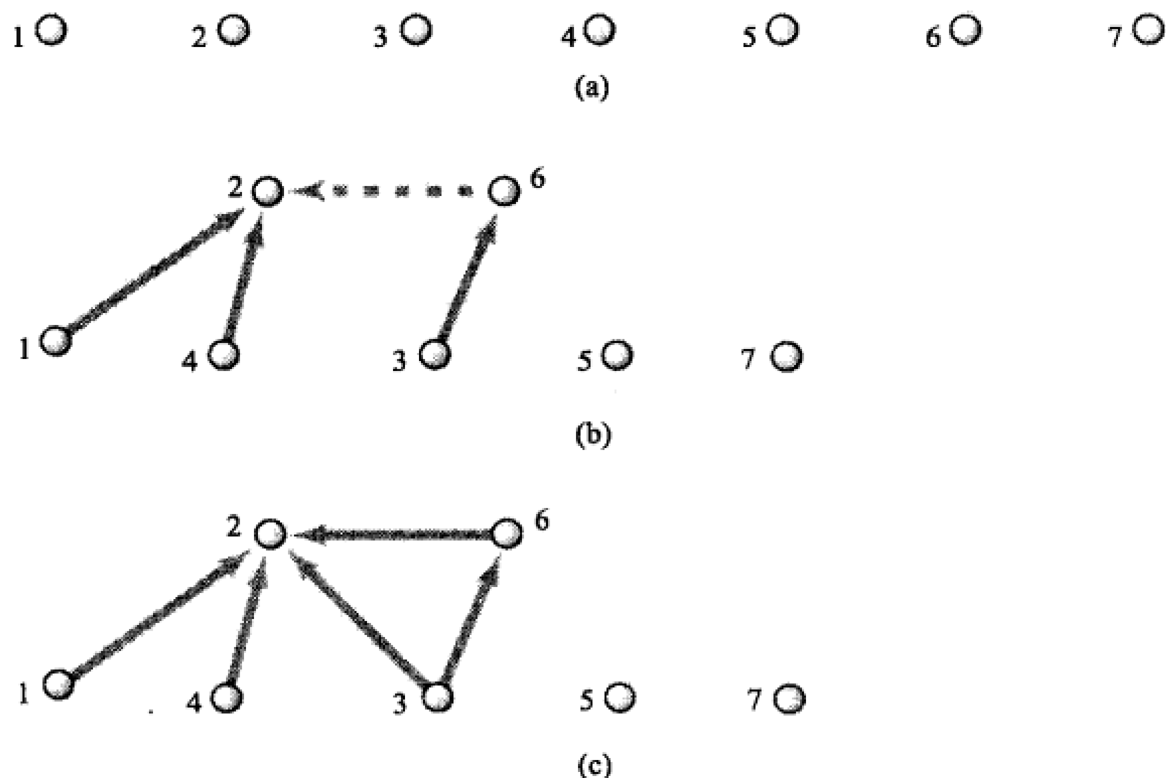


图7-4　并查问题

由于区间的成员会随着时间发生变化，好的数据结构要能允许算法更容易地查验在某一时刻两个节点是否属于同一个区间。

图7-4b就是一个"向上树"（up-tree）的例子。这个术语是由伯纳德·加勒[①]和迈克尔·费彻在1964年提出的，在这一结构中他们把区间中没有边发出的那个节点视为该区间的标识物。判断任意两个节点是否属于同一个区间，只要循边而下，顺藤摸瓜找到该区间的标识物即可。换句话说，要想知道两个节点是否属于同一集合，我们只需检查它们是否共有同一个标准节点即可。这就是并查算法中的"查找"（find）操作。根据这种规则，find(1) = 2，同时find(3) = 2，那么1和3就属于同一集合。而由于find(5) = 5（因为该节点没有从它出发的边），可知5与6、3分属于不同的集合。

每一次查找操作都会尽可能缩短节点到区间标识物之间的路径。这种额外的行为叫做"路径压缩"（path compression），可以为后续的合并及查找操作提高效率。图7-4c显示了霍普克洛夫特-乌尔曼算法是如何利用路径压缩来缩短从3到2之间的路径的。通过这种方法，后续对3的查找操作只需要一步就行了。

① 伯纳德·加勒（Bernard A. Galler，1928—2006）是密歇根大学数学和计算机科学教授。他在大型操作系统和编程语言（如MAD）等方面都作出过贡献。

在许多算法中，并查就好比高楼里的电梯。我们在建楼时可以不配电梯，但居民肯定不会高兴。从这个角度来说，找到最快的并查算法是十分必要的。不过，这需要非常准确的耗时分析。陶尔扬的重要贡献就在于他做到了这种分析。

在此之前，霍氏和乌尔曼曾半推测性地证明了并查算法存在一个线性的时间上限。和平面性测试类似，线性的时间上限表示当问题的大小翻倍后，运行时间也只会跟着翻倍。但是在1973年IBM公司举办的一次算法研讨会上，耶鲁的迈克尔·费彻指出霍普克洛夫特-乌尔曼论证中存在有漏洞。这让陶尔扬开始思考这个问题，他琢磨是否能构造一些反面例子，证明这一时间上限实际上是超线性的，也就是当问题大小翻倍时，所需要的时间要远远超过原来的两倍。随后他提出了一个双重递归构造，给出了一个反阿克曼函数[①]作为时间下限，也就意味着所有的并查算法都不可能是线性的。不久，他又通过一个算法演示证明了这个时间下限同时也是它的时间上限。

> 本来一个非常简单的数据结构，却不得不靠这个非常深奥的函数来分析。这正是这个问题的惊人之处，没人能预料到这种情况。绝大多数人都认为该算法的运行时间是线性的。

其实这种看法和事实非常接近。阿克曼函数之所以提出，原本只是为了说明数学中的一个高级分支，即递归理论。这个函数的增长极为快速。这也表示反阿克曼函数的增长极为缓慢——慢得在实际应用中几乎永远都不能超过5。

不过，和平面性测试一样，结果令世人吃惊——没人能想到阿克曼函数居然还有实际层面的用途。这让研究者们开始把目光投向陶尔扬发明的新的分析技巧。

并查算法的分析非常困难，因为路径被压缩后（见图7-4c），查找操作只需要遍历一个指针即可，但如果路径没有被压缩，则可能需要遍历很多节点。

单单一次查找操作可能就会需要很多步骤。利用已有的分析技巧（也包括霍氏和陶尔扬在平面性测试中运用的技巧），当时的研究人员衡量算法难度的主要依据是操作数量与可能的最长时间的乘积。虽然这种公认的保守方法能够满足通常的要求，但陶尔扬发现在并查算法上它会误导性地得出过长的耗时。

尽管单次的查找操作可能会消耗很长的时间，但是这次查找能对路径进行压缩，从而为后续的查找节省时间。数年后陶尔扬和他的学生丹·史利特在术语上借鉴了会计领域的“分摊”概念，即一次查找的工作量将会被“分摊”到因其而获益的后续其他查找操作上。

> 重点在于，对于一个数据结构的操作肯定不止一到两个。我们不能只考虑单个操作的耗时，而应该考虑这整个一系列操作中的平均耗时。

就像深度优先搜索已成为了一种标准的算法技巧，分摊也已成为了一种标准的分析技巧。很多算法开始有意让自己的某一个操作进行“无私”的奉献，从而让其他的“伙伴”操作从中获益。

① 威廉·阿克曼（Wilhelm Ackermann，1896—1962）是德国数学家。他于1928年提出的阿克曼函数是一个非原始递归函数。该函数需要两个自然数作为输入值，输出一个自然数。其输出值的增长速度非常高，仅是(4, 3)的输出已大得不能准确计算。

最大网络流

1973年，陶尔扬受不了康奈尔大学当地再次的严冬，于是接受了西海岸加州伯克利大学的聘书，在那里度过了两年。他在1975年回到斯坦福，丹·史利特成为了他的研究生。两人在研究所谓的最大网络流问题时再次运用了分摊的概念。

> 假设有一个有向图，它的每条边都有特定的容量，此外还有一个源点（source）和一个汇点（sink）。把各条边都当做抽取液体的管道，我们要算出从源点到汇点间液体的最大流量。

图7-5中的图A代表了一个输送石油的管道网络，其中每一段管道（图的边）都有特定的容量（有数字标识），也就是该段管道所能承受的最大流量。图B显示了在给定的管道流量下整个网络的最大流量，从s到t的总量是115。所谓网络流问题，也就是要找出给定网络的最大值。

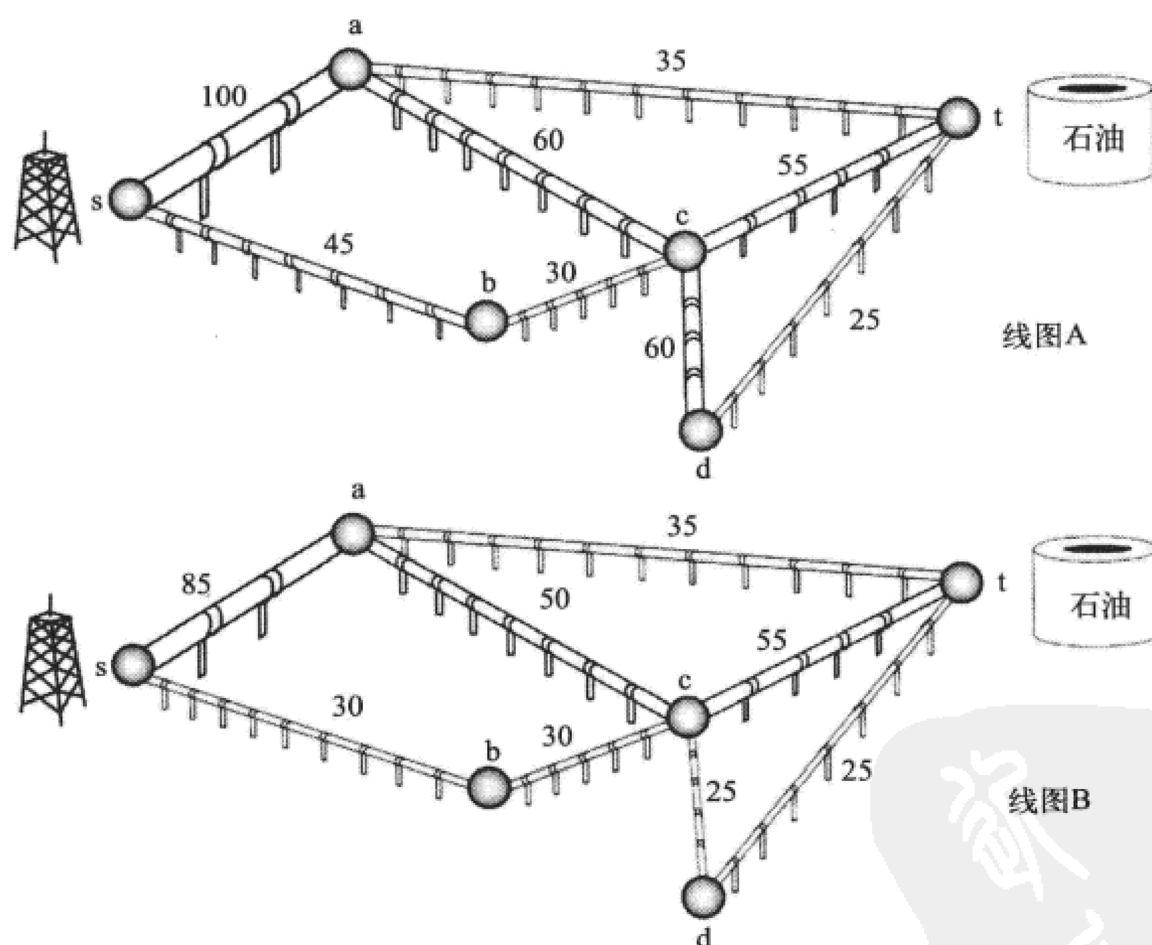


图7-5 网络的最大流量

1956年，L. R. 福特[①]和D. R. 富克逊[②]共同发表了解决该问题的第一个算法。

① L. R. 福特（Lester Randolph Ford, Jr.，1927—）的父亲大福特（L. R. Ford, Sr.）也是一位知名数学家，曾任美国数学协会主席。小福特专门研究网络流问题，与富克逊共同提出了著名的福特-富克逊算法。

② D. R. 富克逊（Delbert Ray Fulkerson，1924—1976）1951年于威斯康星大学麦迪逊分校获得博士学位，后在兰德公司数学部门任职，1971年任康奈尔大学工程系教授。离散数学界的富克逊奖即以他为名。

(1) 起始流的值为0，称其为当前流。很显然这是满足容量限制的（亦即没有任何边会过饱和）。

(2) 尝试找到一条连通源点和终点的路径，其中的每条边都能容纳额外的流量，称为"增广路径"（augmenting path）。尽量增大该路径的流量，但不得让任何边过饱和。将通过该增广路径的流加入到当前流。重复步骤2直到无法再找到新的增广路径。

正如福特和富克逊自己所指出的，在某些情况下这个算法的效率很低。如果容量不合常理甚至根本无法得出答案。10多年后，J. 埃德蒙德[1]和理查德·卡普于1969年提出了该算法的改进版，解决了这一问题并提高了效率：每次都选择边最少的那条增广路径。这让算法运行的时间能够正比于节点数乘以边数的平方。

当时远在苏联的数学家E. A. 迪尼克[2]在1970年独立获得了与埃德蒙德和卡普同样的发现。不过他又更进一步，想办法一次性找出所有具有相同容量的增广路径。由印度、以色列、前苏联和美国等地的研究人员陆续提出的算法一直在为这一思想添砖加瓦。而陶尔扬和史利特则以一种新的数据结构，实现了更快的算法。

> 我们发现，问题的关键在于要找到一种有效的数据结构，以便从整体上解决问题（使边饱和）。这种数据结构后来叫做动态树（dynamic tree），史利特的博士论文即以此为基础。

1980年，陶尔扬和史利特加入了新泽西莫雷山的贝尔实验室。他们在那里对动态树继续进行研究。

> 最终我们发现，如果不去关注最坏情况，而只是考虑平均情况（分摊），就可以大大简化这一数据结构。
>
> 我们开始以更系统化的方式来考虑分摊概念。我们提出了自调整（self-adjusting）数据结构的想法，这种方法在最坏情况下并不高效，但在分摊情况下效率却很高。我们创建了一种自调整数据结构，叫做伸展树（splay tree）[3]，性能极佳。

和路径压缩一样，对树进行伸展也是一种为了让后续操作获益而执行的操作。尽管"伸展"操作本身可能非常耗时，但是陶尔扬与史利特运用分摊技巧证明，这种做法得到的好处要大于付出的成本。

---

① J.埃德蒙德（Jack Edmonds，1934—）在组合最优化领域作出了重要贡献，于1985年获得冯·诺依曼奖。他与卡普共同提出的埃德蒙德-卡普算法又称为最短增广路径算法。

② E. A.迪尼克（E. A. Dinic）出生于前苏联，1948年以色列建国后成为以色列人。他提出的迪尼克算法又称为阻塞流算法（blocking flow algorithm）。

③ 伸展树是一种二叉排序树，其中的一般操作都基于伸展操作。它的优势在于不需要记录用于平衡树的冗余信息。在对一个二叉查找树执行一系列查找操作时，为了缩短整个查找时间，被查频率高的那些节点就应当经常处于靠近树根的位置。陶尔扬想到了一个简单的方法，即在每次查找之后对树进行重构，将被查找的节点"搬移"到离树根近一些的地方。

竞争力

不管是平面性测试、网络流还是其他问题，陶尔扬在20世纪80年代初以前研究的所有问题都能得出明确的答案。一个图要么是平面的，要么就不是；某段管道的流量要么达到了最大，要么没有。相比之下，有些问题的是非则是相对的，解决的方法也没有明确的对错之分。

打个比方，我们认为一位投资顾问很不错，是因为他能推荐好的股票。一种可能的衡量方式是，让一位投资者听从这位顾问的建议去做，另一位投资者听从一位能透视未来的女巫的建议去做，最后比较结果。尽管并不存在这样一位女巫，但从比较的角度来说，这却是一种定义"最佳"的标准。陶尔扬和史利特针对计算机的缓存管理就提出了这样的一种比较标准。

计算机的快速存储器叫做RAM（Random Access Memory，随机存取存储器），而硬盘（disk）则是另一种存储器，比RAM慢1000倍。通常情况下，RAM无法储存用户感兴趣的所有数据，所以缓存管理的目标就是要在RAM缓冲区里保存那些可能很快要用到的数据。由于算法无法预测未来，要想做到这一点就只能依靠之前使用的情况进行猜测。这其中最流行的当属"最近使用"（least recently used）算法。

> 它持续追踪计算机最近读取过的每一个数据页面，当缓冲区装不下时，就扔掉那些最久以前的页面。

"最近使用"算法假设，程序最近读取过的页面最可能马上会被再次读取，就像镜面的反射一样。所谓以史为镜，观往知来。

> 假如你能预知未来，知道程序需要读取哪些数据页面，以及它们的读取顺序，那么你的最佳策略是什么？早在20世纪60年代，IBM的L. A. 贝莱迪[①]就已经做到了这一点。他的算法是扔掉那些在很远的将来才会读取的页面。但是这种（女巫式）最佳策略无法实施，因为我们不能预知未来。那么问题是，如何才能将一种简单、可操作的策略与不可操作的最佳策略进行量化比较呢？
>
> 史利特创造了一个词"竞争力"（competitive）。如果一个联机算法的执行效率保持在最佳离线算法的一定比率内，那么它就是有竞争力的。

陶尔扬和史利特证明，"最近使用"算法在这种标准下表现良好。对于S大小的缓冲区，"最近使用"算法扔掉的页面数量最多两倍于女巫式算法在S/2大小的缓冲区中扔掉的页面数量。他们对可操作算法效率的这种衡量标准不仅在自身领域显得举足轻重，同时也被广泛应用于时序安排、资源分配以及其他以理性为基础的相关问题之中。

持久性数据结构

20世纪80年代初期，陶尔扬在贝尔实验室工作的同时，还以兼职教授的身份在纽约大学任教。

---

① L.A.贝莱迪（Laszlo A. Belady，1928—）是匈牙利计算机科学家，他在IBM研究院工作时提出了理论化的内存调整算法。

在纽约大学研究生丹·史利特和尼尔·萨纳克（Neil Sarnak）、卡内基梅隆学生詹姆斯·迪斯克尔（James Driscoll）的协助下，他开始研究一种能长期保存信息的数据结构。

> 问题是要建立一种数据结构，在其中既能追踪到当前最近的版本，又能追踪到过去的版本。而且为了提高效率，在这个过程中我们不能复制整个数据结构。

“数据结构”是一种储存在计算机存储器中的图表结构，以便于更快地存取数据。通常的数据结构都是树，能让程序快速得到需要的数据。例如数据库系统中应用最广的B树，是由在波音工作的两位计算机科学家，德国人鲁道夫·拜尔①与美国人E. M. 麦格雷特②在20世纪70年代初发明的，能够在0.01秒之内从1千亿个节点中找到需要的节点。这比遍历所有节点要快大概10 000倍。陶尔扬和同事们将这种能长期保存信息的数据结构称为“持久性数据结构”。持久性数据结构的速度几乎和当时常见的数据结构一样快，而且还能让程序（也就等于用户）访问过去的数据状态，只需付出额外的一点代价。

图7-6显示了持久性数据结构的一个例子。如果图中节点X代表的数据需要更新，最有效的方法是用新的值X′代替X。这种方法存在的问题是无法再知道数据库在时刻1的状态。运用陶尔扬-萨纳克方法，程序既可以从时刻1的根节点开始获得时刻1的状态，又可以从时刻2的根节点开始获得时刻2的状态。由于在时刻1和时刻2之间大部分数据都没有变动，因此时刻2的数据结构只需保留变化的部分即可。程序用新的节点来保存X′以及X′的所有祖先节点，从而避免了用X′直接取代X而导致丢失历史数据。对于绝大多数结构来说，这并不会增加过多的工作量。

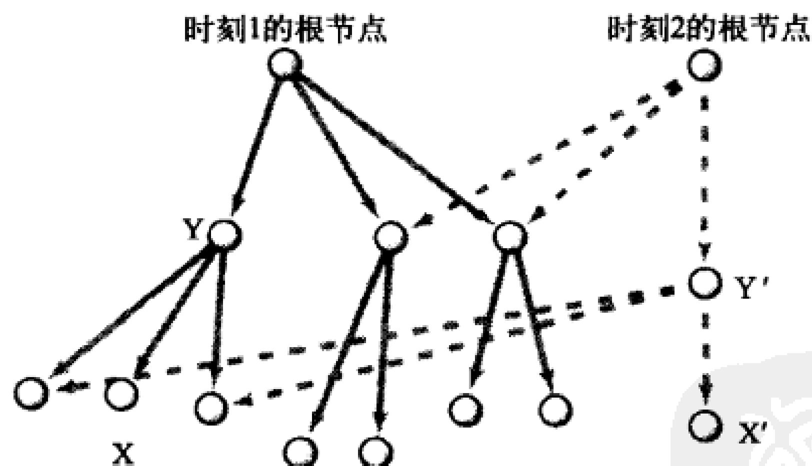


图7-6　陶尔扬与萨纳克的持久性数据结构

这一想法在计算几何学、并行处理等方面已经有了很多应用，但它更主要的长期应用在于“时间数据库”（temporal database）。这种数据库能够快速、高效地重现过去的历史快照。

① 鲁道夫·拜尔（Rudolf Bayer，1939—）自1972年以来一直是慕尼黑工业大学信息技术系的名誉教授。除了B树之外，他还发明了UB树和红黑树等数据排序结构。通常认为“B树”的“B”意指平衡（balanced），也有人称“B”代表拜尔自己（Bayer），或者是波音（Boeing），因为他和麦格雷特当时都在波音科学研究实验室工作。

② E. M.麦格雷特（Edward M. McCreight）除了合作发明B树之外，还参与设计了施乐公司的阿尔托工作站。他也曾为Adobe公司工作。

工作的模式

陶尔扬在他选择的众多算法问题中看到了一种模式。

成功的关键在于问题的选择，而不是找出解决的方法。只要你能找到正确的方向，只要你能提出正确的问题，你就已经开始迈向成功，问题的解决就在眼前。

要从实际应用中发现问题。人们很容易钻进理论的牛角尖，埋首于细枝末节，却失去了和真实世界间的联系。我研究的往往是那些有一定实际意义的问题，我感觉自己能够为它们找到更加实用的算法。

另一方面，世事毕竟无法预料。在这些与世隔绝的领域中产生的想法说不定什么时候就会与其他事物发生联系。数学和理论计算机科学的魔力就在于此。你正想寻找某种普遍原理，突然就出现了某种神秘的联系。没有人知道这是为什么。

陶尔扬也从他的成功中看到了一种社会模式。

人与人之间的交往是非常重要的，而且它具有合作精神。身为教授，最大的乐趣之一就在于能带领一批研究生。它能让你在研究中拥有巨大的优势，而且能调动极大的积极性，因为你身边的人都有着创新、开放的思维，而且求知若渴。

不过，研究工作很大程度上毕竟还在于个人的努力。即使是陶尔扬，也会遇到一筹莫展的日子。

成功需要什么？需要头脑，也需要坚忍不拔的精神。要解决一个问题，可能会有许多次失败的尝试，但最后总会有一次尝试让你看到奇迹的出现。

8

莱斯利·兰伯特
时间、空间和计算

> 如果在一个系统中有一台你从没见过的计算机出了问题，却能连累你自己的计算机无法使用，那么它就是一个分布式系统。
>
> ——莱斯利·兰伯特

伟大的概念革新有时候源自于对常识性假设的再思考。爱因斯坦正是由于对光在任何参照系中都保持同样速度的假设进行了再思考，才得出了他有关质量和能量的著名结论，也就是狭义相对论。他在此过程中证明，没有事件能够真正“同时”发生：在一个参照系中看似同时发生的两个事件，在另一个参照系中则可能并不同时。爱因斯坦藉此端正了物理学中原本错误的常识概念，指出事件发生的相对顺序取决于我们如何去观测。

而在计算机科学中，莱斯利·兰伯特也让分布式计算系统（通过网络相连的计算机集合）的设计师们开始重新思考他们的假设。受到物理学中爱因斯坦的启发，兰伯特证明在分布式系统中事件的相对顺序同样依赖于观测者，它取决于局部相关事件的顺序和讯息收发的顺序。

1941年，莱斯利·兰伯特出生于纽约市的一个东欧移民家庭。大萧条让兰伯特父亲没能实现当医生的梦想，迫于生计只得经营一家干洗店度日。

> 也许是家庭的移民背景让我充满了斗志，习惯于自力更生——我的早期研究工作很少有合作者参与，差不多是与世隔绝。

对数学强烈的求知欲望让兰伯特很早就开始挑战老师们的能力极限。一次，某位初中老师在课堂上教同学们如何分解多项式。

> 我问道：“你怎么知道多项式只有一种分解方式？”你可能以为，如果一个孩子向数学老师提出这种问题，老师肯定是两眼放光，滔滔不绝地讲起唯一因式分解以及相关的问题。但这位老师的回答是：“因为它就是这样。”他不能向学生给出证明。

在布朗克斯科技高中（纽约市专门培养科技、数学尖子的公立精英高中）就读时，兰伯特对教师的印象有了改善，不过仍然很难满意。

布朗克斯的微积分课很不错，提供了严格的定义和证明。但是我记得有一件事没有给出证明——对 x 的自然对数求导得到 $1/x$。当时我对此很不满意。

即使在那些喜欢数学的高中生里，也很少有人如此热衷于证明。而兰伯特不仅希望看到证明过程，还学会了对已有的证明也并不轻信。

我曾经对非欧几何很感兴趣，经常泡在 42 街的纽约公立图书馆。有一次我读到了一本书，里面故意进行了一些谬证，得出诸如“所有三角形都等腰”这样的结论。我想这本书在我心中播下了种子，让我对任何证明过程都心存怀疑。

1965年是兰伯特高中生活的第三个年头，他在纽约的IBM大楼里第一次看到了计算机。正是那一年巴科斯发表了第一门高级编程语言Fortran（参见巴科斯的章节）。

我记得当时去参观纽约的 IBM 公司时，得到了一些他们计算机上废弃的电子管，勉强还能用。于是我和一位朋友装配了一些触发器（一种基本的存储电路），捣鼓出了一个一位数计数器。

我和计算机的真正接触其实发生在高中毕业那年的暑假。当时纽约市政府为成绩拔尖的学生安排了一个实习项目，于是我去了爱迪生联合电力公司。我在那里的工作非常无聊，主要是管理设计图纸。如果有人要挖开街道，就必须找到对应的设计图，避免对电线造成破坏。

但是在爱迪生公司里有计算机。我整天在它们周围晃，最后想尽办法调到了计算机部门。

兰伯特阅读了IBM 705的操作手册，开始每天趁其他人午休的时间在计算机上编程。他写出了一个雄心勃勃的程序，计算出数学常数e的头256位数字[①]。e在微积分领域有着核心的作用，兰伯特早期就曾希望看到有关它的证明过程。

我记得有一次午休时我坐在那里，有人进来以大人的口吻说：“看他那个开心样，把计算机当玩具了。”他说的没错，这就是个绝妙的玩具。而这些大人根本没认识到它的绝妙之处。

我做研究从来都不是坐在那里干想：“X这个问题很重要，所以我应该解决X。”我想也没人会这样做。人们做研究是因为觉得某事很好玩，才会有兴趣去做。

从布朗克斯毕业后，兰伯特来到了麻省理工学院，理想是毕业后成为一名工程师。

我一到麻省理工，就发现工程学并不适合自己。我还不太了解工程学是什么，但显然它对我毫无吸引力。我的父母对我转系的决定不太有所谓，因为数学家的收入似乎没有工程师来得高。不过我有机会当大学教授，他们也觉得很体面。

当时数学的情况很糟糕。似乎每个人都不愿意去碰应用数学，就好像会脏了手。这

① 数学常数e是自然对数函数的底数，与圆周率π及虚数单位i一样属于数学中最重要的常数之一。它的数值约为2.718 281 828 459 045 235 36（小数点后20位）。数学家冯·诺依曼在1949年已经算出了它的头2010位数字，2007年e的已知位数则达到了1000亿。

> 种纯数学要高于应用数学的观点可以追溯到戈弗雷·哈代[1]，它已经渗透到了当时的教育体系之中。如果你能找到一种更抽象、更概括的方式去做某件事，你就会受到褒奖，因为你离肮脏的尘世又远了一步。

对于兰伯特遭遇到的、对数学的这种柏拉图式的追求，我们可以猜测一下背后的原因。也许只是因为那些偏爱理论探索的数学家们感觉计算工作过于繁重和艰苦。第二个原因（据我们推测，而非兰伯特推测），也许是数学家们像部分物理学家那样，把曼哈顿计划视为一种道德上的寓言：将实际计算带入数理结论有可能会导致可怕的新式武器出现。我们已亲眼目睹过这方面的证据：纽约大学柯朗数学研究所的同仁们对星体熔合的研究，却意外导致了氢弹引爆技术的改进。

不管怎么说，"理论胜于计算"的学术偏见并不只是美国独有。前苏联数学家在纯数学方面有许多杰出的成果，但在应用数学领域则乏善可陈。而日本数学家则正相反，他们向来都对计算情有独钟。

在麻省理工获得学士学位之后，兰伯特前往布兰迪斯大学攻读数学博士学位。迪克·帕莱斯（Dick Palais）在那里教第一年的分析课程。

> 我发现，原来真的可以进行如此严密的分析。我天生就对他的分析方式有共鸣。它完全符合我对任何事都希望看到证明的观念。

尽管帕莱斯的课让兰伯特留下了积极的印象，却仍然无法阻止他对学校和数学提出质疑。第二年即1962年，兰伯特认定数学过于脱离现实，毫无意义可言，毅然抽身而去。他开始练习钢琴并尝试作曲，并考虑是否回到研究生院去学习音乐。不过最后他还是决定到马尔波罗学院（佛蒙特州一所小规模的艺术学校）去教数学。

> 我什么都教——我就是整个数学系。那里有一些学生相当聪明，但应付不来传统学校的那些狗屁规矩。

此时的兰伯特依然痴迷于证明。

> 我努力教学生们如何作数学证明，但没有成功。如今我意识到，失败的原因在于当时我有一种错觉，总以为应该有一套严密、精确的数学论证方法，但实际上数学证明只是一种书面形式，也并没有那么严密。现在我已经转变了看法。[2]

教书开始变得无趣，研究生院的生活更是如此，尤其是当他开始研究物理学（主要是相对论）之后。

> 我的动力主要来自于渴望理解时间。人们总是说时间只是时空四维世界中的另一个

---

① 戈弗雷·哈罗德·哈代（Godfrey Harold Hardy，1877—1947）是英国数学家，在数学分析与解析数论上有重要贡献，同时也是群体遗传学中哈代-温伯格定律的发现者之一。

② 兰伯特已经察觉到自己与众不同的风格。在后来的一篇论文中他写道："证明已经细化到了令人发指的地步……读者可能会认为我们矫枉过正，对那些明显的断言也给出又臭又长的证明。然而，他们没看到还有很多同样明显的断言，正是因为我们试着给出又臭又长的证明，才发现了其中的谬误。"——原书注

维度而已。但它并不只是另一个维度，它非常特殊。在寻常的三维空间中，原点(0, 0, 0)与点(x, y, z)之间的距离是$\sqrt{x^2+y^2+z^2}$。但在四维时空中，(0, 0, 0, 0)与(x, y, z, t)之间的距离却是$\sqrt{x^2+y^2+z^2-t^2}$。

正是这个减号让时间的方向不同于另三个空间的方向。我希望理解这个减号从何而来。我希望理解时间与空间的不同。

麻省理工的一位教授劝阻了兰伯特继续思考相对论，但鼓励他攻读数学专业的研究生。为了勤工俭学，兰伯特在麻省计算机协会（Massachusetts Computer Associates，COMPASS）找了一份兼职，当时COMPASS正负责为ILLIAC IV计算机编写Fortran编译器。

ILLIAC计算机至少有64个处理器，这在20世纪70年代是一个非常大的数字，其目标是通过多个处理器同时执行指令，能够运行庞大的数值计算。问题是Fortran程序都是循序运行的，一次只能执行一个指令，如何将其编译为一次执行多个指令呢？兰伯特奋勇投身于该项目中。

这在当时（1972 年）是计算机的最前沿领域。我用了一点点简单的数学运算来弄明白其中的算法。我认为这个编译器其实很简单，只是涉及了一些线性代数。COMPASS把我视为从山上下来的摩西，手持晦涩的神圣文本让他们学习[①]。他们最终通过一个算法弄清了它的意思。

1972年获得博士学位之后，兰伯特继续在COMPASS设立于帕洛阿尔托的办公室为ILLIAC工作。

我只身去了加利福尼亚，远距离为 COMPASS 工作，但却不怎么和 COMPASS 里面的人打交道。我认为脱离了学术研究圈子对我的研究工作起了关键作用。

置身于学术界之外，这让我免于受到当时业界风气的影响，不会随波逐流。

面包店算法

兰伯特开始订阅专业计算机科学刊物，包括由美国计算机协会（ACM，Association for Computer Machinery）出版的《ACM通讯》。在该刊物上，有关戴克斯彻解决互斥问题的讨论一直余波未息（到当时已经持续了11年）。

如果有多台计算机同时接入存储器的一个共用资源，它们之间就必须协调访问的顺序，以便每次只有一台计算机操作该资源。这种协调被称为互斥：当一台计算机使用资源时，它就应排斥其他计算机使用该资源。

读者们可能还记得，戴克斯彻利用火车轨道的类比给出了这一问题的解决方案。如果两趟列车方向相对，却必须共用一段轨道，那么为了避免发生事故，每次只能有一趟列车可以驶过这段轨道。工程师们建立了信号灯系统，以确保在任何时刻都只有一趟列车通过共用区域。戴克斯彻

① 据《圣经》记载，摩西带领族人逃出埃及，途中在西奈山下祈祷，请求耶和华为他的族人指一条道路。上帝之手在西奈山的峭壁上刻出十条戒律，即著名的"摩西十诫"。

针对互斥问题提出的方案之一就使用了这种程序机制，称为“信号灯”。

信号灯机制让某台计算机排他性地使用某个资源（例如打印机、计算机屏幕的某个部分，或者某个内部数据结构）。如今的计算机硬件都直接支持信号灯机制。

戴克斯彻的另一种解决方案（1965年首次发表于《ACM通讯》）则不需要信号灯机制，因此降低了硬件的负担。但它有两个缺陷。首先，某台运气不好的计算机可能每次都会在竞争中落败。用戴克斯彻引入的术语来说，就是处于“饥饿”状态。1966年高德纳独辟蹊径，改进了戴克斯彻的算法，一举解决了饥饿问题。

但是戴克斯彻和高德纳的算法还有另一个缺陷。他们假设对计算机存储器的读、写操作是“原子级的”（atomic）。计算机科学中的“原子”和这个词最初的希腊语含义[①]相同：原子级的读或写操作都是不可分割的。如果一个原子级的读操作和一个原子级的写操作在同一时刻发生，那么要么读操作整个完成后写操作才开始，要么反之。要想实现这一效果，硬件设计就必须确保在多个读写操作同时抵达某个存储单元时，能以某种方式进行排序。而兰伯特提出的解决方案——“面包店”算法却能让硬件免于这种负担。

> 我思考了互斥问题，想到如果每个人都“排号”会是什么情况。我把这种方法称为面包店算法是，因为在我小时候，当地的面包店是唯一一家用排号方式决定接下来谁会获得服务的店铺。当然，在计算机里没人会给大家发号，所以每个人都必须自己选择一个号码。
>
> 如果两个人同时开始选号，当一个人进入了临界区而另一个人还在选时会怎样？第二个人也许会选到一个更靠前的号，于是也进入临界区。
>
> 避免这个问题的办法是让进程 *X* 在选号时“举旗”，此时如果 *Y* 也想排队选号，就必须先等待 *X* 把旗降下来。这个算法被证明是正确的。

兰伯特用这一简单的类比勾勒出了解决方案，但具体的实现细节却非常复杂。为了将其付诸实践，研究者们数年来一直在仔细检验该算法及其证明过程。兰伯特自己也是其中之一。

> 接下来的几年里我所做的所有事情都是在研究面包店算法。我发明过许多算法，但感觉只有面包店算法是真正由自己发现的。

不少产品都运用了兰伯特的面包店算法，但它最主要的贡献在于解决了一个更为普遍的问题，即如何设计一种即使在出现故障时也能正确运转的系统。

> 我想当时我在寻找的是一种容错机制。其他算法（包括基于信号灯机制的算法）的所有进程都写入同一个存储单元，如果这个存储单元出了故障，那么就全军覆没。
>
> 而在面包店算法中，每一个进程都写入不同的存储单元，被视为该单元的“所有者”。如果某个进程出了问题，只会导致它专属的那个存储单元出故障，给出默认的“失败”

---

① 原子说是公元前四世纪的古希腊哲学家德谟克利特提出的一种关于物质结构的朴素唯物主义学说。他认为万物皆由大量不可分割的微小物质粒子组成，这种粒子称为原子。各种原子没有质的区别，只有大小、形状和位置的差异；原子遵循必然的规律在“虚空”中不断运动；它们集合时形成物体，分离时则物体消失；物体“投射”出来的形象与感官接触就引起色、声等感觉。后进一步发展成为现代的物质结构理论。

值。因此，即使一些进程出现故障，也不会影响到其他进程继续运转。

该算法的另一个重要特性就是它不需要依赖比信号灯更基础的假设。这对兰伯特来说更像是一种惊喜。

> 在我为面包店算法写出正确性证明之后，我才意识到它的正确性根本不依赖于那些与写操作同时发生的读操作返回什么样的值。这表示（硬件强制的）对共享变量的访问互斥在我的算法中并不是必需的。我所发现的其实是一个无需基本互斥假设的互斥算法。

兰伯特对理论的贡献是极其彻底的。它既不依赖于所选用的编程语言，也不依赖于任何未实现的假设，它只依赖于算法和自然法则。这让他有信心将该结论推广到被称为“原子级寄存器”（atomic register）的技术问题上（参见补充内容“兰伯特谈原子级寄存器”）。

> 我的动机来自于物理现实。似乎我考虑得更多的是并发机制的物理问题，而非计算机问题。这对别人来说无疑是不可思议的。

兰伯特谈原子级寄存器

面包店算法让我想到了一个问题：什么才是进程间相互通信的最简单、最基础的途径？从物理层面来考虑，通信都是由一个个此进程写、彼进程读的字节所组成，因此无需假设当读和写同时发生时会出现什么情况——只不过是读操作获得一个0或者1而已。

之后我提出可以实现一种“原子级寄存器”，它的运行方式就和传统算法中的存储器一样。我已经知道如何利用这样的寄存器来解决经典的并发性问题，所以这种问题也可以用这些最原始的比特来解决。

我无法让任何人对此感兴趣，所以也从未整理过这些结论。大概是1985年，我读到一篇讨论集成电路问题的论文，讨论的方向和我研究的原子级寄存器相同。由于我的假设是基于物理层面的，所以看到它被应用于超大规模集成电路也并未感到吃惊。因此我还是整理并发表了自己的结论。

这引发了一个新的小规模产业，理论学家们扩展了我的结论，开始实现多读多写寄存器——这是一个我曾花时间考虑过的问题，却没有成功。

时间与分布式系统：兰伯特时钟

兰伯特发现，物理学确实有助于以新视角来看分布计算中的基础性问题。于是他开始回顾自己热衷于狭义相对论时的心得，将其用于研究分布式系统中的时间概念。

一个“分布式系统”由空间中相互分散的许多元素组成，它们通过网络相互连接，彼此间的讯息传递需要花费时间。不管是办公室网络、国际通信网络还是邮政系统都是如此。

在这些系统中，“时间”是如何确定的呢？就拿美国的计算机网络来说，如果每一台计算机都有自己的时钟，那么这些时钟可能无法完全同步——要知道在计算机系统中，即使是几个纳秒（一秒钟的十亿分之一）的差别也是不可忽视的。如果每台计算机都参考同一个时钟，例如以俄

亥俄州的代顿市为准，那么它们在接收时钟时也会出现延迟，而且延迟的长短也不固定。从加州西部的圣何塞发送讯息到时钟，在系统空闲时可能需要1.4毫秒，而系统拥堵时则可能需要30毫秒。那么对于分布式系统来说，全局时钟这一概念的意义究竟何在呢？

兰伯特认为它没有任何意义——应当被抛弃。他从狭义相对论中获得了灵感。

> 如果在时空中使用这种时间流逝的标准，你会如何测量呢？也许你会在两端都放上爆竹，并安排它们同时点燃爆炸，然后就能看到烟雾出现在什么地方。问题在于“同时”这一概念本身。有相对运动的两位观测者就会有不同的看法。所谓“现在”的概念是相对而言的。
>
> 在分布式系统中，和相对论一样，不同的观测者对两件事发生的先后顺序可能会有不同的结论。

在兰伯特发表的《时间、时钟与分布式系统中的事件顺序》论文中，他解释了如何面对一个没有全局时钟的世界。据他所言，最关键的问题是如何找出那些必须按顺序发生才有意义的事件。

利用相对论的类比，兰伯特提出，根据分布式系统的物理特性，存在着3种决定顺序的方法。

(1) 如果某个处理单元完成了A事件之后才开始B事件，那么所有处理单元都会认为A在B之前发生。

(2) 如果A事件发送一条讯息，而B事件接收同一条讯息，那么所有处理单元都会认为A在B之前发生。

(3) 如果所有处理单元都认为A在B之前发生，而且所有处理单元都认为B在C之前发生，那么所有处理单元都会认为A在C之前发生。

如果在第三条规则中加入非自反性（即任何事件不会发生于自己之前），就会得到数学中所称的“偏序”(partial ordering)。偏序相当于我们认为的可能因果关系路径：如果信息从A流动到B，那么A就在B之前发生。图8-1中显示了一个真实世界的例子，用图解法表示了鲍勃给爱丽丝发送传真、爱丽丝给哈利发送传真的状态，两者都成功收到了传真。

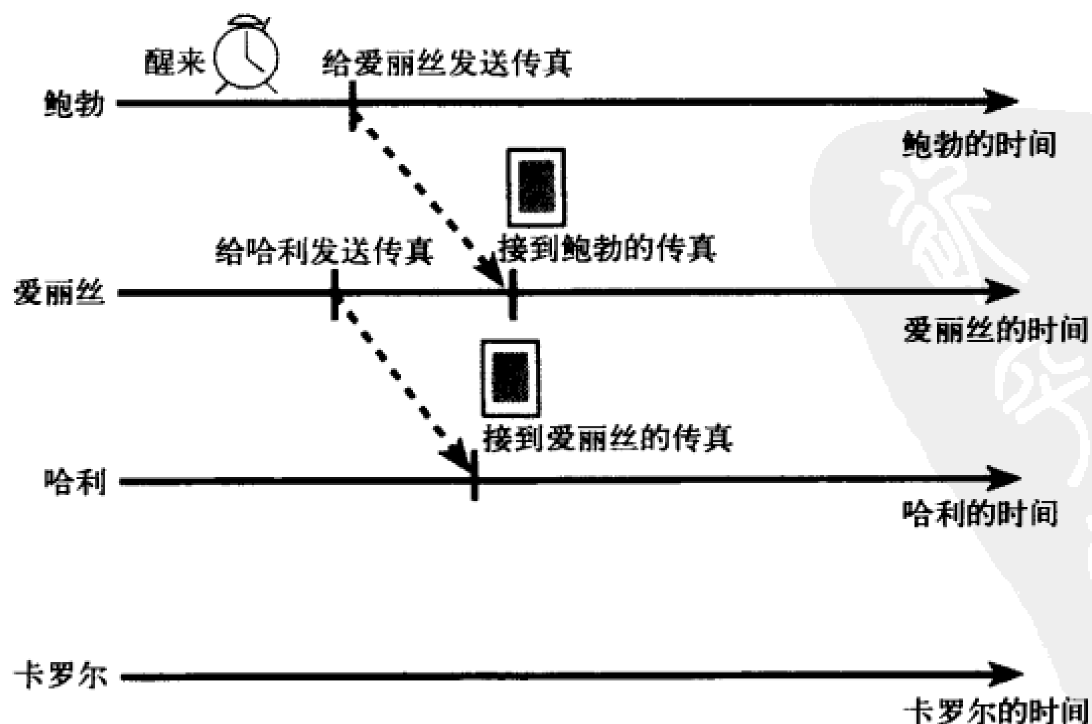


图8-1　可能因果关系路径

所有的观测者（例如卡罗尔）都会认同如下观点：鲍勃先醒来然后给爱丽丝发传真，爱丽丝先给哈利发传真然后接收到鲍勃的传真，以及每一份传真都是先发送然后才能收到。请注意有一些顺序信息是任何参与者都无法知道的，例如爱丽丝发传真是否在鲍勃发传真之前，甚至是在鲍勃醒来之前。由于并不是所有事件都需要按顺序发生，所以时间概念天生就是“偏序的”(partial)。分布式系统的任何算法都必须将这一因素考虑在内。

兰伯特的论文还证明，如果对这种偏序持续扩展，就能构建出一个全局性的排序。也就是说，对每一个事件都指配一个“时间戳记”，以使每一个事件发生的时间都与其他事件不同。如今这种全局性排序被称为“兰伯特时钟”。

> 这种时钟是一系列时钟的集合。每个进程都有一个时钟，根据如下规则进行同步。
>
> (a) 同一个进程的两个不同事件，发生在该进程时钟的不同时间。
>
> (b) 如果 A 向 B 发送一条讯息，那么发送讯息的 A 时钟时间将先于接收讯息的 B 时钟时间。
>
> 我意识到，若要建立分布式系统，不同的进程对事件发生的时间需要有一致的认识。特别是任意两个事件的顺序，所有进程都必须统一认识。如果两个事件是因果关系，那么就必须认同因果顺序。如果事件没有因果关系，那么它们选择什么顺序并不重要——但必须全部选择同一个顺序。如果能够建立起这种协议，那么（原则上）就能轻而易举地让分布式系统做我们想做的任何事情。

兰伯特时钟所涉及的全局统一排序绝不仅仅是抛出一个同时事件概念那么简单，因为这种排序甚至有可能与完美同步（根据单一观测者）时钟产生的排序不一致。也就是说，一个外部观测者可能看到A事件12点01分发生在一台计算机上，而B事件12点02分发生在另一台计算机上，但是兰伯特时钟却可能给B事件一个更早的时间。根据其定义，兰伯特时钟会确保这种违背不会产生影响：由于B的兰伯特时钟值小于A，所以不会有任何信息从A流向B。

无故障的证明

1977年，兰伯特离开了麻省计算机协会，加入了加州帕洛阿尔托的一个智囊机构——斯坦福国际研究所。该所有一个美国航天局赞助的项目，即“软件实施的容错多处理器”（Software-Implemented Fault Tolerant Multiprocessors，SIFT），目标是设计一种自动处理故障的飞行控制系统，要能应付哪怕是非常少见的故障。任何一个严重错误，即使只有百万分之一的可能性，每年都可能导致许多架飞机坠毁。要想杜绝这种可能，就需要可以“容忍”错误的硬件和软件。即使真有故障发生，计算机系统的其余部分也必须能够正确运行。容错机制最简单的形式就是将硬件备份，让两个完全相同的处理器运行同一个程序。如果其中一个停止工作，另一个就能接手继续完成任务。

但是，故障处理器可能会在软件出错时做出无法预测的行为。如果考虑到这种可能性，那么容错机制就会复杂得多。在最坏的情况下，这种不可预测的处理器还可能会做出“背叛”行为。

这种背叛比处理器“死去”更糟糕，因为它可能会发送矛盾的信息，导致正常处理器的工作发生紊乱。贝内迪克特·阿诺德[1]之所以让殖民地人民深恶痛绝，就是因为他是一个活着的叛变者，而不是死去的爱国者。

在兰伯特加入之前，SIFT团队（罗伯特·舒斯塔克、迈克尔·米拉史密斯和马歇尔·皮斯）已经找到了处理这种问题的方法，但要付出相当大的代价。事实上他们已经发现，即使提供三倍冗余的处理器（三倍于可能出现故障的处理器数量），也仍然不够！

兰伯特鼓励他们整理出自己的结论。他还给这个问题起了一个名字，以表现其错综复杂的特性：拜占庭将军问题。

> 我认为这个问题需要一些形象化的表述，于是有了叛变将军的比喻。一开始我打算叫它阿尔巴尼亚将军——那时候阿尔巴尼亚对西方世界就像个黑洞[2]，所以我觉得该国也不会有什么人对此提出异议。后来我还是改成了拜占庭将军。

兰伯特的另一个贡献是引入了数字签名，从而找到了一种新的、更简单的解决方法。和手书签名一样，数字签名也能够识别来自特定源的讯息（参见图8-2）。数字签名甚至更加安全——没人知道如何伪造，也许除了美国国家安全局之外。

图8-2 叛变故障问题的简化例子。B根据收到的讯息，无法辨别是C叛变（场景1）还是A叛变（场景2）。但如果所有讯息都有签名，B就会知道A是否篡改了C的讯息

① 贝内迪克特·阿诺德（Benedict Arnold，1741—1801）是美国历史上最臭名昭著的叛徒之一。美国独立战争期间，他本是美国陆军的将军，但为了6000英镑的悬赏，将纽约西点堡垒拱手让给了英国，后加入英军成为了一名陆军准将。

② 阿尔巴尼亚位于欧洲东南部，曾是欧洲最不发达和低收入的国家之一。1944年恩维尔·霍查领导阿尔巴尼亚劳动党人取得政权，陆续与南斯拉夫、苏联、中国建立友好关系（后又断交）。也许这就是兰伯特所谓“黑洞”之语的由来。

利用数字签名，发送矛盾信息的“叛徒”处理器就会被揭穿，因为其他正常工作的处理器会交换信息。正常处理器会发现这种前后矛盾的信息，于是便忽略掉故障处理器。数字签名如今已成为容错设计中的一种标准技术。

兰伯特将自己的发现归功于天时地利。

> 这个例子很好地证明了科学有时候也依赖于小道消息和偶发事件。在那个年代，数字签名还是一个很深奥的概念。除了研究密码学的计算机科学家之外几乎没人听说过，而研究密码学的计算机科学家也屈指可数。我知道这个概念是因为维特菲德·迪菲（Whitfield Diffie）是我的朋友。[①]

容错只是安全系统的特性之一。另一个是故障预防。SIFT项目的目标是在硬件和软件设计中消除哪怕是极端罕见的故障。对复杂程序的测试无法穷尽所有可能的情形，所以许多罕见的故障并不能检测出来。但这不表示它们在现实中就不会发生。

一个实例是，尽管进行了大量的测试，第一次航天飞机发射仍然因为一个程序错误而被推迟。测试只有1/67的机会发现这个错误——它只在计算机时钟走到某个相关值时才会发生。因此该错误在预计发射时间的数小时前计算机未能正常启动时才被发现。

据苏格兰斯特林大学从事可靠性研究的彼得·拉金（Peter Ladkin）回忆，这个问题导致了一系列更为彻底的试验，其中一些复杂得令人烦不胜烦。

> 航天飞机发射中止之后，机组人员模拟了起飞后的全过程，其中包括倾泻燃油，并坠落于西班牙。模拟中的飞行控制处理器发生了中断，问题出在一个计算 go-to 语句上（程序将控制权转向到某个存储单元）。在运算中，这个 go-to 语句会将控制权转到计算机存储器之外的某条指令，导致系统瘫痪。除了这个问题，程序员们还发现了其他 17 个问题，其中一些可能会导致严重的后果。当时是 1981 年。在那之后，数字飞行控制系统的编码量和复杂程度大大增加，也就更容易产生这样的错误。

航天飞机航行软件中的550 000条指令是当时世界上最顶尖的成果。软件从未在航天飞行任务中造成过任何真正的事故（与模拟中发生的事故相反）[②]，也从未影响过任何航天飞行任务的目标。航天飞机数据系统的负责人鲍勃·辛森（Bob Hinson）称，到目前为止主要的测试手段包括设计走查（多人连续审查程序以确定其运行状况）和极其仔细的测试。辛森估计每5000行代码中存在的错误不超过1个，或者说飞行软件中遗留的所有错误不超过100个。这个错误率比民用工业的要小得多。民用工业无法达到这一等级的部分原因在于成本。如果考虑到设计、编码和测试，修改一行代码所需要的成本大概是2000美元。

---

① 维特菲德·迪菲和马蒂·赫尔曼（Marty Hellman）发明了许多公钥加密和数字签名的技术。——原书注

② 1986年的“挑战者号”航天飞机灾难与2003年的“哥伦比亚号”航天飞机灾难不在此列。“挑战者号”是因其右侧固体火箭助推器的O型环密封圈失效，导致高压高热气体泄漏而解体。“哥伦比亚号”是因为其外储箱的绝缘泡棉脱落，导致隔热系统受损、左翼和起落架舱熔化而失事。

验证还是不验证

兰伯特和拉金等人深信，仅仅靠测试是远远不够的。他们认为人们应当去“证明”（形式化地验证）硬件和软件是正确的，尤其是软件。兰伯特是计算机科学领域中形式化验证最为积极的倡导者之一。

> 在对 SIFT 进行验证时我们发现了一个漏洞。这种漏洞只有形式化验证才能发现，其他任何方法都发现不了。这种漏洞每两年就会导致一架飞机由于偶发的宇宙射线而坠毁。
>
> 对那些性命攸关的计算机系统而言，验证绝对是必不可少的重要环节，但现状看来却并非如此。人们仅仅依赖于测试。测试当然不能忽略，但它绝不能替代形式化的数学验证。
>
> 程序和算法都是抽象的数学实体。它们都可以应用数学方法。所以看在上帝的份上，让我们做吧。

兰伯特的这一观点存在着争议。绝大多数从业者都避开了验证，可能是因为难度过大。这也是学术界的普遍立场。

> 1979 年，理查德·德米洛[①]、理查德·李普顿[②]和艾伦·佩利[③]发表了一篇论文《证明、定理和程序的社会历程》，文章基本上对程序验证抱有一种敌视态度。他们提出了一些有力的观点，指出数学家们进行的数学证明在一开始并不十分可靠。数学证明的可靠性需要经过一段社会历程，其中需要许多数学家对其进行核验，之后业界才会认可它是正确的。而在程序验证中缺少这种社会历程。
>
> 之后他们列举了一些不相干的论点，得出结论说程序验证是不切实际的。实际上他们证明的是，唯一对程序有效的验证方法是机械验证。人们不会去检验证明，但机器必须做到。我想对于关乎性命的系统而言确实如此。

验证还是不验证的问题不仅仅是一个学术问题。生活中每天都有日益复杂的计算机系统出现，我们的社会越来越依赖于它们的正确运转。彼得·拉金给出了一个现代飞机的例子。

> 1993 年 9 月，德国汉莎航空的一架空中客车 A320 在华沙机场坠毁。飞机在恶劣天气下着陆后，在长达 9 秒的时间里，制动装置的逻辑电路（着地轮制动器、扰流器和推力反向器的控制程序）一直未能启动刹车系统。飞机撞在路堤上，导致 2 人死亡、多人受伤。按照设计逻辑，飞机在这段时间内仍然处于“飞行”状态，尽管飞行员已经将其

---

① 理查德·德米洛（Richard DeMillo，1947—）是美国计算机科学家和教育家，2000年任惠普公司副总裁，如今是乔治亚理工学院的特聘教授。至今已发表过100余篇文章、著作和专利，研究领域涉及数学、计算机安全和软件工程。

② 理查德·李普顿（Richard Lipton，1946—）的研究领域涉及计算机理论、密码学和DNA计算。他如今在乔治亚理工学院计算机学院任副院长和教授。

③ 艾伦·佩利（Alan Perlis，1922—1990）是首届图灵奖的获得者，编程语言领域的先驱，1971年开始任耶鲁大学计算机系主任。

降落在跑道上，但也只能无助地等待刹车系统启动。

针对这种隐藏漏洞的问题，兰伯特提出了一种“层级结构证明”的新方法。

兰伯特这一方法的原理是：在最高层级作出该定理的基础陈述，然后在下一层级给出导出该定理陈述的一系列断言，再下一层则是导出上层断言的一系列子断言——依次类推，直到看上去合理为止。这种方法使得证明的检验过程不再像之前那么痛苦。

兰伯特一生对证明过程的痴迷可能防止了飞机坠毁和电厂爆炸。到目前为止，他认为层级证明方法尚只能解决部分的问题。

> 我的这种手写结构化证明的方法只能让手写证明更加可靠而已。它们永远都不会像机器检验的证明那样可靠，所以可能并不适合那些关乎性命的系统。但是与无结构化的手写证明相比，它们仍然要可靠得多。我认为，作为一种更为经济的手段，它们可以用来在不那么生死攸关的系统或部件中代替耗时的机器检验。
>
> 比如说，对于控制飞机飞行的那部分系统，它的正确性也许要用机器检验来证明。而计算最省油的飞行航线剖面图的那部分则可以通过手写来验证。

侧记

兰伯特仍然记得计算领域的有趣一面。事实上，他最广为人知的其实是一项业余爱好——名为LATEX的排版系统。

> 20世纪80年代早期，我开始写一本关于并发性的书。1982年前后高德纳发表了他的“新版”TEX，我想我得为自己的书写一套新的宏指令集。我决定要额外多付出一些精力，让除我之外的其他人也能用上它。
>
> 结果就有了LATEX。这是一个多重双关语，“La”既代表法语的“the”又代表我的姓“Lamport”，而“latex”又有“乳胶”的意思。不用说，“额外的一点精力”变成了额外的大量精力，我想我大概花了10个月在这上面。
>
> LATEX成了科学界通用的文档处理语言——我估计有将近75%的计算机科学论文是用LATEX写的。我最出名的身份其实是LATEX的作者，至于真正的工作成果反而没那么多人知道。在得知我并未将一生都奉献给LATEX时，人们都很惊讶。

为分布式系统重新定义时间、驯服“背叛”进程、验证关乎人类生死的代码、创建文档处理语言——兰伯特在所有这些领域中都扮演着关键性的角色。为什么是他？为什么是这些问题？他的解释谦虚而又全面。

> 回顾自己的工作成果，大多数都是纯属好运——我只是在合适的时间遇到了合适的问题，而又恰好具有合适的知识背景而已。

PDG

9

史提芬·古克和利奥尼德·列文

良解难觅

某些问题无法依靠算法解决，这是不可改变的基本事实。而正是这一想法吸引了我。

——史提芬·古克

有时候，有些事情不能做也是件好事。有很多事情我都不希望别人对我做。

——利奥尼德·列文

一个15岁的普通人也能看懂毕达哥拉斯定理[①]的证明过程，但当古希腊几何学家发现这条定理时，他们居然向神明献祭膜拜。成千上万的人都能哼唱贝多芬的第九交响曲，却几乎没人敢奢望有他那样的音乐天赋。所有优秀的思想都存在着一个基本的不对称——认识它们很容易，无中生有地发现却很难。在计算理论中，这种不对称同样也是不争的事实，至少看起来是这样。

例如著名的巡回推销员问题。假设我们的推销员威利先生有一条推销路线、一份旅行预算和一张航班票价表。他需要从波士顿出发，走遍图9-1中的其他9个城市然后返回波士顿，并保证不超出预算。而你需要帮他找出合适的路线。如果他的预算很充裕，要完成任务只需举手之劳。但如果预算很紧张，那么事情就会非常棘手，甚至有可能无论如何都会超支。你必须考虑每一种可能的旅行顺序——就这么几个城市而言，大概就有100 000种可能的路线！

如果有3个城市A、B、C，相互之间都有往返航班，那么就有6种可能的顺序（或者说走遍每个城市的次序）：ABC、ACB、BAC、BCA、CAB和CBA。如果有4个城市，那么就会有24种顺序：在A、B、C的6种可能顺序中，第四个城市D对其中的每一种顺序都能以4种方式插入。例如，对于顺序BCA来说，可以有如下4种顺序：DBCA、BDCA、BCDA和BCAD。

---

① 毕达哥拉斯定理即勾股定理：直角三角形两直角边（即“勾”、“股”）边长的平方和等于斜边（即“弦”）边长的平方。据说毕达哥拉斯证明了这个定理后，斩了百头牛作庆祝，因此又称百牛定理。中国典籍记载该定理的公式与证明是在商代由商高发现，故又称之为商高定理。

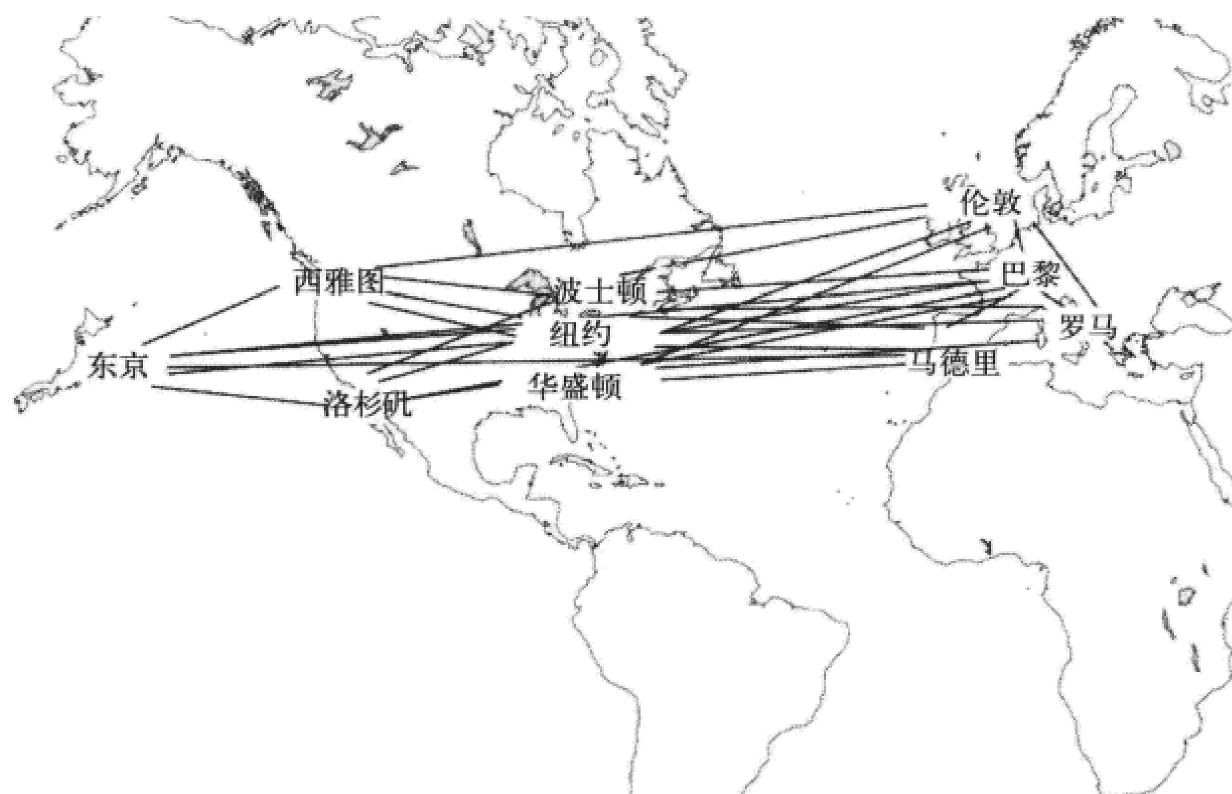


图9-1　巡回推销员问题。任务是确定推销员威利是否能在一定预算内从波士顿出发，走遍其他所有城市然后返回波士顿

顺序的数量也被称为阶乘（用符号“!”表示）。阶乘的增长非常迅速，4的阶乘4! = 4 × 3 × 2 × 1 = 24，5! = 5 × 4! 也就是 5 × 24 = 120，6! = 720，依次类推。固然，计算机很擅长处理大量的可能性，但可能性数量过于巨大依然会超出任何计算资源的能力。正如史提芬·古克所指出的：

> 如果有 100 个城市，那么你就得估算 100 的阶乘那么多种路线。没有任何计算机能穷举 100! 种路线。做一点简单的计算你就能意识到它的难度：如果让太阳系中所有的电子以自旋的频率来计算这个问题，即便算到太阳衰竭也到不了头。你需要明白，有些事情在现实中就是行不通的。

巡回推销员问题的不对称性在于：如果有人向威利推荐某条路线，他很容易就能判断其是否符合预算限制。要他自己找一条路线很难，但验证却很容易。

在20世纪50到60年代，设计、运筹和人工智能领域的许多问题似乎都有这种“难解易验”的特点。计算机科学界的很多人猜测这些问题可能都属于某个共同的数学范畴。

20世纪70年代早期，有两位科学家——美国的史提芬·古克和前苏联的利奥尼德·列文在同一时间彼此独立地给出了这类问题的描述。如今它们被称为NP完全问题（理由将在下文中提到），数量极多而且还在不断增长。对于计算机科学家们来说，证明某个问题属于这个范畴就等于宣判无法为其找到任何精确的解答——顾客将不得不接受一个近似的答案，别无他法。

在西方的计算传统中，计算困难度的概念起源于逻辑和图灵关于“不可计算性”的结论。迈克尔·拉宾于1959年首次将可计算问题的固有难度这一概念形式化。史提芬·古克遵循了这一传

统，于1971年定义了NP完全问题。

20世纪50年代末，前苏联的运筹学界开始非正式地将某些优化问题形容为需要“perebor”，其俄语意为“蛮力”，或者说穷举。他们将那些需要搜索全部或几乎全部的可能性之后才能确定最佳答案的问题统称为“蛮力搜索”。例如，即使是一个中等规模的巡回推销员问题或类似问题，要想尝试所有的可能性也是行不通的。

然而，perebor问题背后的理论并不像西方传统中那样来源于逻辑，而是起源于俄国数学家安德雷·科尔莫戈罗夫[①]提出的字符序列的随机性及其描述的困难性两者之间的关系。他直接利用了美国人克劳德·香农在20世纪40年代末提出的信息理论。从那时以后，前苏联和西方国家在数学方面的交流日益减少。[②]

在1971年于前苏联各大学进行的演讲和1973年发表的一篇小论文中，利奥尼德·列文指出了perebor问题与科尔莫戈罗夫理论之间的关系。其中的一个结论就是对NP完全问题的特征性描述。

但这只是讨论的开始。不管是古克、列文还是后来的科学家们，没人真正证明了这一范畴的问题是极端困难的。他们只是证明了如果该范畴内的某一个问题很困难，那么所有的问题都很困难。因此他们提出了当今理论计算机科学领域中的一个关键的开放性问题：NP完全问题是否需要穷举搜索？换句话来说，如果有哪个算法只需探索一小部分可能的路线就能解决巡回推销员这样的问题，那么它就将改写计算机科学的历史。

史提芬·古克：逻辑和西方式传统

1939年，史提芬·古克出生于纽约州的布法罗市。古克的父亲是联合碳化物公司（Union Carbide）的化学家，同时也在布法罗大学教书。他一直都梦想着能享受乡村的恬静生活，于是在史提芬10岁时举家搬到了纽约州克拉伦斯的一个奶牛场。他们把土地租给了一位农夫，自己留下了一头奶牛（史提芬每天为它挤奶），还养了些其他动物。

> 我对棋类游戏有一定的兴趣——没什么特别的。我在学校的数学成绩不错，但那也只是一所普通的乡村高中而已。我从未想过要当数学家。我母亲的叔叔亚瑟在威奇托市当数学教授，他是我知道的亲戚中唯一一位数学家。

纽约州的克拉伦斯也是威尔森·格瑞特巴赫[③]的故乡，他是可植入式心脏起搏器的发明者。正是格瑞特巴赫让少年的古克立志成为一名电气工程师。

---

① 安德雷·科尔莫戈罗夫（Andrey Kolmogorov，1903—1987）是20世纪杰出的数学家，在概率论、拓扑学、算法信息论、直观逻辑、计算复杂性等领域都有着卓越的贡献。

② 在这种相互隔离的状态下却同时作出类似发现的程度令人惊讶。本章中描述了列文和古克相互独立的发现，但很快就能想到另外两项独立的成果。1959年拉宾提出复杂性的基础理论时，戈雷果日·萨穆伊洛维奇·塞廷（Gregorii Samuilovich Tseitin）在前苏联也进行了同样的研究。1969年，格雷戈里·蔡廷（Gregory Chaitin）在纽约定义了与科尔莫戈罗夫相似的任意性测量，但较之略晚几年。——原书注

③ 威尔森·格瑞特巴赫（Wilson Greatbatch，1919—）是一位工程师，普遍误认为是他发明了心脏起搏器，但他其实只是推动了可植入式起搏器的发展。

> 我在暑假时到他开的小店（其实是个谷仓）的阁楼上帮他干活。那时候是20世纪50年代，晶体管问世不久，他正在试验晶体管电路。而我则帮助他焊接电路，感觉非常有意思。
>
> 在进入密歇根大学后，我的专业被称为科学工程，但我的兴趣其实是电气工程。入学第一年（1957年）我选修了伯纳德·加勒的一门一学分的编程课程。

古克和他的一位朋友编写了一个程序来测试著名的哥德巴赫猜想，即每一个大于3的偶数都可以表示成两个素数之和。在程序能够运算的范围内该猜想一直都能成立。（这一猜想至今依然是未解的，因为要证明素数的普遍性质很难，而另一方面又没人能举出反例。）

古克拿到了数学学位，但他也学到了足够多的计算理论知识，了解了类似图灵“停机问题”这样的固有不可能问题（参见拉宾的章节）。之后他转入哈佛大学攻读数学博士学位，本意是研究代数。但是对他影响最大的老师是应用科学领域的王浩[1]。王浩在数理逻辑和哲学方面求学多年，后转而研究自动定理证明，即让计算机自己去发现证明方法。

对他的另一影响来自于复杂性理论，迈克尔·拉宾刚刚在1959年发表论文为其奠定了数学基础。许多复杂性理论的开创性人物，包括拉宾、尤里斯·哈特马尼斯和理查德·斯特恩斯，都为古克等求知若渴的学生进行过讲座或报告。

> 这似乎是一个非常自然而又基本的问题。显然，有些问题在原则上可以通过算法解决，但实际中却不行，因为等到太阳系毁灭也不会算完。所以考察问题的固有难度是一个很自然的事情。
>
> 在计算机出现之前，人们只能靠手写来执行算法。这个过程极为漫长乏味，所以复杂性问题很难引起人们的兴趣。现在有了强大的计算机来帮助我们，每秒执行数千次操作，自然而然就会探究到底有哪些类型的问题可以真正得到解决。

古克的导师王浩为这个问题加入了逻辑学的观点。

> 我很了解王浩的思想和方法。我对NP完全问题的结论与他非常类似。图灵和王浩说的是谓词演算（predicate calculus），我说的是命题演算（propositional calculus）。

谓词演算和命题演算是数理逻辑中的两种语言。王浩研究的是谓词演算的“可满足性问题”的复杂性。古克后来开始对命题演算的可满足性有了兴趣。为了理解二者间的区别，读者可以参考下面的例子。

谓词演算作出的陈述是有关一群个体的。例如，陈述“所有的奥运会选手都很健康”可以描述为：对于所有x，如果是奥运会选手（x），则健康（x）。

谓词演算允许我们用特定个体来代替x。比如说，如果我们知道阿喀琉斯[2]是一位奥运会选手，

[1] 王浩（1921—1995）是华裔美籍哲学家、数理逻辑学家。1943年西南联大数学系毕业，1945年清华大学哲学系毕业，曾师从著名逻辑学家金岳霖。1948年哈佛大学逻辑学博士毕业，同年成为哈佛的副教授。

[2] 阿喀琉斯（Achilles）是古希腊神话中的英雄人物，色萨利国王佩琉斯与海洋女神忒提斯的儿子。他是所有英雄之中最耀眼的一位，参与了特洛伊战争，被称为“希腊第一勇士”。

也就是说“奥运会选手（阿喀琉斯）”成立，那么上述公式导出的结论“健康（阿喀琉斯）”也成立。

而古克研究的命题演算则是一种更为简单的语言，它只允许对个体作出陈述，例如“米奇是一只老鼠”。命题演算允许我们从已有的命题推导出新的命题。例如，如果命题“要么米奇是一只老鼠，要么唐纳是一只鸭子”和“唐纳不是一只鸭子”都成立，那么根据规则我们就能推导出“米奇是一只老鼠”成立。

可满足性

上述两种逻辑语言都包含了一个重要的问题：是否存在一个真假值的赋值，使得给定的公式成立？如果存在，那么这个公式就是“可满足的”（satisfiable），否则就是“不可满足的”（unsatisfiable）。比如说，“x且非y”是可满足的，因为只要x赋真值、y赋假值这个断言就成立。

另一方面，“x且非y且非x”是不可满足的，因为无论怎样赋值，x和非x中都会有一个不成立，导致整个断言不成立。

阿隆佐·邱奇和阿兰·图灵已经证明，从计算上无法判断某个谓词演算公式是否是可满足的，即使用再多的时间也不行。而古克则证明，判断命题演算的可满足性，可能需要尝试很多种可能的赋值。

那么会有多少种可能的赋值呢？如果有1个命题变量（例如x），那么就会有2种可能的赋值：x为真或者x为假。如果有2个变量（x和y），那么就会有4种可能的赋值：x真且y真、x真且y假、x假且y真、x假且y假。一般来说，如果有n个变量，就会有2^n种可能的赋值。按这样计算，10个变量会有1000余种可能的赋值，20个变量就有100万种，30个变量有10亿种，40个变量有1万亿种。每增加10个变量，可能的赋值数量就会增长1000倍。

可能性的数量会随变量n以指数增长，因为n在上式中是指数。如果一个算法需要分别检验每一种可能性，那么执行该算法的时间同样也会以指数增长。相比之下，在多项式时间算法中，需要的时间则是按照n的某个固定次方的表达式增长的，这里的n表示问题的大小。

比如说，多项式时间表达式n^3在n为10时的值是1000。当$n=20$时，它的值也只有8000，$n=30$时值为27 000，$n=40$时值为64 000。而在指数增长中40个变量就有1万亿种可能的赋值！

定义 NP 完全性

在完成了哈佛的博士论文之后，古克在加州大学伯克利分校待了很短一段时间，然后去了多伦多大学。

1971年，他在计算机协会（ACM）第三次年度研讨会上发表了一篇论文[①]，阐明了自己关于可满足性的思想。古克在论文中指出，有些问题可以在多项式时间内检验出一个可能的解答（称之为“候选者”）是否正确。因为程序并不一定总能找出最好的那个候选者，所以有时候它只能

① 这篇论文名为“定理证明过程的复杂性”（The Complexity of Theorem Proving Procedures），文中库克首次明确提出了NP完全问题，并奠定了NP完全性理论的基础。

“猜”一个出来。出于这种原因，古克将这些问题称为非确定性多项式（nondeterministic polynomial）问题，简称NP问题。猜测的部分是非确定的，而检验的部分是多项式的。

巡回推销员问题和可满足性问题都具有这种性质，因为我们能很快检验出某个旅行计划是否满足推销员的预算，或者某个赋值是否让公式成立。而重要的问题在于，是否能在多项式时间内找出最合适的那个候选者。

古克证明可满足性问题是NP中最难的问题之一。确切来说，他证明如果可满足性问题可以有多项式时间算法，那么任何NP问题都可以有。这使得可满足性问题成为了一个NP完全问题。因此，在实际中证明一个问题是NP完全问题，就表示它很难解决，尽管在找到某个解答后我们能很快检验出它是否正确。如果有人受到命运眷顾，找到了能解决某个NP完全问题的有效算法，那么这个算法很可能对所有的NP完全问题都适用。

就在古克的论文问世后不久，加州大学伯克利分校的理查德·卡普证明有另外21个问题也属于NP完全问题，其中有一个和巡回推销员问题紧密相关。他用一种叫做“可约简性”（reducibility）的方法进行了论证：如果这些问题中的任何一个能被快速解决，那么可满足性问题也可以。因此，这些问题至少和可满足性问题一样困难，反之亦然。不幸的是，任何科学家都无法证明这些问题真的很难。

利用基因寻找最短路径

20世纪90年代末，南加州大学的伦纳德·阿德曼[①]发现了一种用DNA链解决巡回推销员问题的方法。DNA由两条被称为核苷酸的化学成分链构成。核苷酸共有4种碱基，传统上标记为A、C、T和G。它们成对相连（确切来说是A与T、C与G相连）然后组合在一起，使DNA形成了其独特的双链形态。

阿德曼将每个城市视为一条链上的某个序列。如果$X^{in}X^{out}$是城市X的链、$Y^{in}Y^{out}$是城市Y的链，那么从X到Y的航线则表示为$x^{out}dy^{in}$。这里的x^{out}与X^{out}相连（一条链中每个A的位置与另一条链中每个T的位置相同，C与G也与此类似），y^{in}与Y^{in}相连。d的长度则与从X到Y所花的费用成正比。

他运用成熟的生物实验技术，将城市链和行程链混合在一起，找到了最短的包含所有城市的双链。这就是推销员应该选择的路线。

这个极具独创性的方法让我们可以用小型、低功率的计算设备来解决这一问题。那么这种方法是否为所有的NP完全问题提供了通用的解答呢？很不幸，没有。要解决1000个城市的巡回推销员问题，需要的分子数量比已知宇宙的所有原子数量还要多。

一个（近乎）永恒的讨论

读者可能会问，如果无法真正证明某个问题很难，那么这些有什么用呢？让我们来打个比方。

① 伦纳德·阿德曼（Leonard Adleman，1945—）是美国理论计算机科学家，南加州大学计算机科学和分子生物学教授。他的杰出贡献在于DNA计算和密码系统，后者的应用包括https。

如果某个专利申请夸口说能实现永动机，专利局立刻就会驳回这一申请。专利审查机构使用了归约论证：如果某台机器真能如此运行，那么能量守恒假设就会无法成立。而审查者们对这一假设是深信不疑的。

永动机类似于一个NP完全问题。如果任何一个这样的问题能被快速（即在多项式时间内）解决，则所有的问题都可以，那么NP完全问题需要穷举搜索这种现行的假设就无法成立。而计算机科学家们对NP完全问题确实很难这一假设同样深信不疑。因此，如果某位计算机科学家遇到了一个不能给出快速算法的问题，他可能就会试着证明它是一个NP完全问题，含蓄地指出这种问题真的很难。古克对此进行了总结。

> NP 完全问题可能没有多项式时间的解决方法。我们已知所有的 NP 问题都可在指数时间内解决。它们也可能可以在多项式时间内解决，但目前尚无定论。如果一个类似可满足性这样的 NP 完全问题能在多项式时间内解决，那么所有问题都可以。它们都是相互可约简的——这正是其重要之处。

在卡普之后，全世界的研究人员已经证明了其他数千个问题也属于NP完全问题。典型的例子包括电话网络的几何布局优化、跳棋等游戏的最佳走法等等。古克对NP完全问题的数量感到吃惊。

> 最初我只认为 NP 完全性是一个有趣的想法——我根本没意识到它具有如此巨大的潜在影响。

利奥尼德·列文：科尔莫戈罗夫式传统

20世纪60年代古克在哈佛大学苦思复杂性理论时，苏联有一位年轻的高中生正在学习perebor问题和科尔莫戈罗夫复杂性。

利奥尼德·列文1948年出生于第聂伯罗彼得罗夫斯克，位于乌克兰腹地的一座工业城市。他的父亲最初在高中执教俄语和文学，后来拿到了教育学博士学位，开始在大学任教。利奥尼德的母亲则是一位设计桥梁的工业建筑师。很小开始，列文就对科学和数学产生了兴趣。

> 我在学习门捷列夫表（化学元素周期表的雏形）时遇到了问题。它并不如我希望的那样有规律。我花了很多精力进行研究，一遍又一遍地重绘，最终的成果和我日后在美国看到的几乎一样。

列文还在家中浴室里混合各种试剂做化学实验。他的父亲为他买了一套前苏联科普作家雅科夫·别莱利曼[①]的作品，并鼓励他参加奥林匹克竞赛。奥林匹克竞赛是前苏联政府举办的竞赛，旨在选拔数学和科学方面有特殊天分的学生。地方的优胜者将进入地区级比赛，然后被选入专攻数理的寄宿学校深造。

① 雅科夫·别莱利曼（Yakov Perelman，1882—1942）是俄罗斯和前苏联科普作家，著作包括《趣味物理学》和《趣味数学》（均被翻译为英文）。

> 在 20 世纪 50 年代末到 60 年代初，苏联举国上下都对科学抱有极大的热忱。奥林匹克竞赛非常热门，全国绝大多数青少年都会参加。如果你获胜，同时又不是犹太人，那么还有机会参加国际奥林匹克竞赛。

列文在基辅市物理奥赛中拔得头筹，被保送到基辅大学的物理数学寄宿学校。一天，前苏联科学界的权威科尔莫戈罗夫来访，这在学校可算是一次空前的盛典。当时列文刚满15岁。

> 科尔莫戈罗夫的来访是一件大事。他与所有领导进行了会面，然后接见了孩子们。那是一个巨大的房间。他讲了话，然后提出了一些问题，让我们举手回答。很快这就变成了我和另一个男孩之间的比赛。

科尔莫戈罗夫向小列文提出的那些问题，为他日后在计算机科学中面临的难题做了很好的铺垫。补充内容中给出了列文对其中一个问题的解答。

列文对科尔莫戈罗夫谜题的解答

问题

假设所有的词汇（所有的字母序列）都分属于两种类别：一类可以用于出版物（合理词汇），另一类不行（不合理词汇）。那么对于任意无限长的字母序列，是否都可以拆分成有限长的序列，且这些序列都属于同一类别（第一个序列可以除外）？

解答

可以。如果某个字母序列任意长度的前缀都是合理的，那么我们就称这个序列“前缀合理”。假设一个无限序列$A = a^1, a^2, \cdots$存在某个上限n，使得只要取长度超过n的前缀，就会导致剩下的序列“非前缀合理”（即至少有一个前缀是不合理的）。因此，将前$n+1$个字母截取出来。剩下的（非前缀合理）序列一定有不合理的前缀，将其取出来，无限重复这一过程，最后就能将整个序列拆分为无数个有限长的前缀，其中每一个（可能除了第一个）都是不合理的。可以用反证法证明能够实现这一拆分：假设不能，取出第k个不合理前缀后，剩下序列的所有前缀都是合理的，那么可以把前面的所有字母作为一个前缀取出，剩下一个前缀合理的序列（而这与假设矛盾）。

如果不存在这个上限n，那么取出任何一个前缀，剩下的序列都是“前缀合理”的。我们可以无限重复这一过程，不会出现例外。因为假设有例外，那么把之前的所有前缀长度加起来就是一个上限n，而我们已假设n不存在。当然，这并不表示我们可以随意截取，因为这样就可能会留下一个非前缀合理的序列。但肯定每次都可以截取出适当的前缀以使剩下的序列是前缀合理的（因为n不存在）。那么A序列就被拆分为无数个有限长的前缀，其中每一个（可能除了第一个）都是合理的。

科尔莫戈罗夫对列文解答问题的能力印象深刻，于是后来邀请列文到莫斯科大学他自己的学校里就读。

他教的孩子不多，大概20多个。他甚至会为我们安排音乐会——他非常喜欢音乐。此外他还为我们分析诗作。他了解很多东西，而且大部分都达到了专业水准。

科尔莫戈罗夫最初是一名历史学家。他好像曾经提到自己证明过一些历史猜想，而当他介绍自己的成果时，别人说“这很好，但我们还需要更多的证明”。之后他决定成为一名数学家，因为在数学里一个证明就足够了。

列文沉浸在数学之中，几乎排斥其他的一切。他偶尔也下下象棋，但拒绝学习钢琴。

我确实很喜欢俄罗斯小说。我的记忆力超强，出于好玩就可以记住大篇的诗歌。后来我失去了这种记忆力，因为我担心它会限制我的想象力。

尽管前苏联的体系培养了大批数学和物理方面的年轻科学家，对于计算机科学却曾经抱有敌意。

在20世纪50年代初的苏联，计算机科学是一种违法的话题，往往显得不合时宜。

我想一些前苏联哲学家可能是被诺伯特·维纳[①]给激怒了——他年轻时是一位伟大的数学家，晚年时却写了一堆有关计算机科学的废话。他晚期的“通俗科学”思想之一就是控制论，这个时髦字眼在前苏联却用于形容计算机科学。虽然这些废话似乎并无害处，但前苏联的哲学家们认为它与马克思主义背道而驰。

然而到了20世纪60年代，由于计算机科学显而易见的广泛应用，前苏联军方坚持学校必须开设这门学科。他们甚至开始培训年轻学生学习使用计算机。列文第一次接触计算机，就是在本科生必须参加的准军事单位服役期间。

我们……用一些非常古老的计算机工作，(使用的是) 穿孔纸带、各种指示灯还有操作面板。它们给人的印象极其深刻。此外我们还学了一些概率方面的东西。假设一只火箭正在飞行，它撞上这个或那个的概率分别是多大？

自然，我痛恨军事和武力。但是我们在那里学的数学比大学里的常规计算机课程要有趣得多——学校那门课程除了Algol语言啥也不教。

科尔莫戈罗夫复杂性

在1971年的博士论文中，列文对科尔莫戈罗夫复杂性进行了阐述。科氏是他的导师，他和列文的答辩委员会都通过了这篇论文，答辩委员会中还包括其他一些杰出的前苏联数学家。然而，列文的博士学位却被校方拒绝授予。

① 诺伯特·维纳（Norbert Wiener,1894—1964）是美国应用数学家，在电子工程方面贡献良多。他是随机过程和噪声过程的先驱，提出了“控制论”（cybernetics）一词，即“反馈”的一种形式化概念，是一门研究机器、生命、通信等各类系统的调节和控制规律，即动态系统改变的环境条件下如何保持平衡状态或稳定状态的科学。

在当时，几乎所有的苏联青年都属于共青团——这是一个类似共产党但属于年轻人的组织。它通过各种活动教导我们要热爱政府。而我则做了一些离经叛道的行为。

对于我所做的事情，本应有相应的惩罚，不过这些事对委员会来说也不太好看：它可能会让上层觉察到不对劲。于是他们试着假装什么也没发生，而我也能躲过这一劫。

但我没有注意到捷克斯洛伐克之后，时代已发生了变化[①]——出于某种原因我不愿意认识到这一点。

聒噪不已而又盛气凌人，我正好是当时大学当局亟需的一只最好的替罪羊。

最大的冲击并不是（博士学位）被拒授本身，而是在该决定的书面陈述中使用了罕见的、极为直接的政治词汇。这种措辞让我再也不可能获得博士学位，最终导致了我的移民。

在移民到美国之前，列文于1973年在一份前苏联刊物《信息传输问题》上发表了一篇关于通用顺序搜索问题的文章。他在文中勾勒出了perebor问题与科尔莫戈罗夫信息理论之间的形式化联系。科氏的信息理论探索了字符序列（例如0和1）的“随机性”与描述该序列所需的字符数之间的关系。比如说，很容易用几个字来描述由100万个连续的1组成的字符串（我们在这句话里已经做到了）。类似地，我们也可以用简短的方式来描述任何重复的模式，例如00111重复100万次。

用科尔莫戈罗夫的术语来说，可以用简短语句描述的长序列并不是随机的。他的研究有一个基本结论，即随机的程度并不依赖于所用的数学语言。无论如何定义，大多数序列都是“随机的”——它们需要的描述至少和序列本身一样长。

（读者可以通过一个计算来证明。n的二进制数有多少个？2^n个。多少个能用$m<n$位二进制数编码的程序描述？只有2^m个。假设$m=n-10$，那么2^m就只有2^n大小的1/1000，因此99.9%的长度为n的数都需要$n-10$个以上的二进制数字来描述。）

列文研究的问题是，编写能够描述长字符串的程序有多难？

这个问题相当于编写能简单、快速生成给定字符串的程序的固有难度。我想到了一个办法，把平铺（一个空间压缩问题）进行简化，但是失败了。不过，我成功地证明了它与另一个类似问题是同等难度的：为部分布尔函数查找小二度电路（这是一个电路设计问题）。我还证明了它与其他 5 个问题也是同等难度的，包括可满足性、集合覆盖、地图映射和嵌入。

由于前苏联的闭塞，列文并不知道远在大洋彼岸的古克和卡普已经证明大部分这类问题都是同等难度的——或者按他们的说法，这些问题都是“NP完全”的。

在好几年时间里，古克和卡普的论文在前苏联根本无人知晓，因为任何前苏联图书馆或机构都没有收录刊载这些文章的文献。而旅行和流通等方面的限制又阻挡了这一成

① 这里所说的应指“布拉格之春”改革运动。第二次世界大战后捷克斯洛伐克在前苏联的帮助下获得解放，成为社会主义国家。1968年，当时的捷共领导人杜布切克发动了名为“布拉格之春”的经济和政治改革运动，为当时对外推行扩张政策的前苏联所不容，于是联合其他华约国家人侵捷克斯洛伐克，改革运动戛然而止。前苏联的行为受到世界各国的广泛批评。

果通过私人渠道进入苏联。当我终于看到他们的成果时，对卡普的研究感到非常惊讶，因为我从未想到居然有如此之多的奇妙问题都是NP完全的。

和古克的研究获得的反响相比，列文的论文在前苏联引起了一些正面的反响，但似乎受到了某种抑制。

它引起了一些人的兴趣。我不知道他们是真的对研究感兴趣，还是对我政治上的麻烦感到同情：有时候人们对我的友善有些过分。当时我的研究还没有引起广泛的关注。直到20世纪70年代后期，在得知卡普的研究之后，苏联人才开始变得兴奋起来。

古克与列文——NP完全问题之后

20世纪70年代中期，当列文还在和前苏联当局纠缠政治问题时，古克开始着眼于一个新的问题：如何在时间和内存之间进行折衷？

处在杂乱的办公室或房间的人都知道，在狭小的空间里找东西或者整理文件是多么麻烦的一件事情。古克和多伦多大学的艾伦·鲍罗廷[①]一起，研究了计算机上的一个类似问题。

试想国税局要整理1亿份纳税申报单。所有已知的快速方法都需要占用大量的计算机内存，而慢速方法则不需要那么多内存。我们证明，既快速又不占内存的方法是不存在的。

（擅长数学的读者可能想知道：所需时间和所用内存空间的乘积最少应与$(n^2)/(\lg n)$成正比，其中的n是需要整理的税单数量。）

对于外行人来说，如果科学家宣称某个问题不可能解决，那就等于说它已毫无希望。古克一提到他年轻时的导师、发明家威尔森·格瑞特巴赫，一位虔诚的长老会信徒，就经常取笑他所选择的领域。

由于我的名气主要来自于对不可能性的证明，他对此总是持一种怀疑态度。他会问我："最近又证明出什么是不可能的了吗？"（其实）我并不是在故意找茬，存心去证明人们要解决那些问题是白费心机。总有很多委曲求全的办法。人们不是非要一个永动机不可，他们可以退一步，给它一个动力源就可以了。

对于NP完全问题，人们可以有运用启发式方法、寻找捷径或者求近似值等各种退路。人们已经研究过这些东西。我对NP完全问题的证明只是为了引导人们把精力放在解决那些能够解决的问题上面。我想这是积极的、有建设性意义的。我想强调的重点是尽管技术在飞速进步，仍然存在着一些根本的原则是不变的，仍然有很多我们做不到的事。

古克还在继续他对命题逻辑算法的基础性研究。他目前感兴趣的领域是为命题演算公式寻找简短的（多项式长度的）证明。

---

① 艾伦·鲍罗廷（Allan Borodin，1941—）1969年在多伦多大学任教，1977年成为教授，后来出任系主任。他在计算复杂性理论和算法领域作出了许多贡献，1991年被选为加拿大皇家学院院士。

列文如今是波士顿大学的计算机科学教授，在那里他与芝加哥大学的马里奥·茨格第（Mario Szegedy）、拉兹洛·巴柏（Laszlo Babai）和兰斯·弗德诺（Lance Fortnow）合作开发了一种“透明”证明的理论。

这几位数学家开发出来一套写证明的方法，具有如下的奇异特性：如果以这种方法写出的证明中存在某个错误，那么对证明里任何一小部分进行数学检验，都会有一个相同的概率能发现这个错误。为了讨论的方便，我们假设每次发现一个错误的概率是1/2。假设我们以这种方法写出了一个证明，如果分别对它不同的部分进行了40次数学检验，都没有发现错误，那么这个证明错误的可能性就小于1万亿分之一。

> 这有点像全息摄影。这种照片的每一个部分都包含有其他所有部分的信息。这种证明也是如此。

更大的问题

20世纪90年代中期，普林斯顿大学的安德鲁·怀尔斯[①]提出了一种有望解决费马大定理[②]的方法。这位17世纪的法国数学家某天在自己稿纸的空白处随手写了一个难题，结果这个问题从那时开始就一直困扰着许多学者。证明NP＝P是否也会这样呢？换句话说，一个能在多项式时间内“检验”其解答的问题，是否能在多项式时间内被“解决”呢？（参见图9-2。）

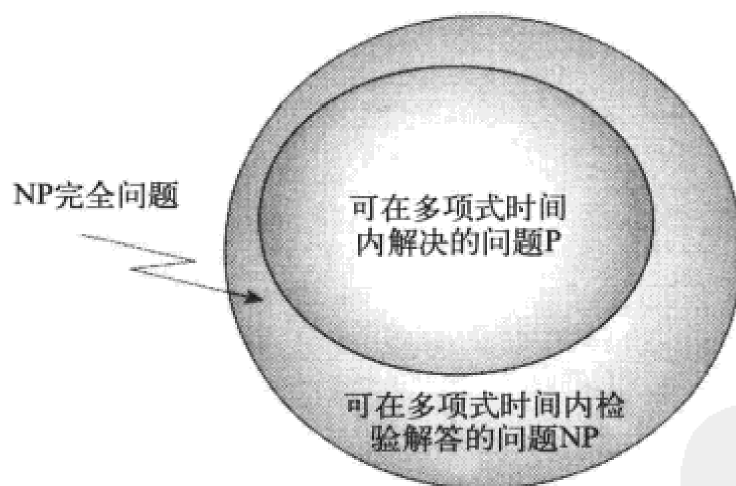


图9-2　如果一个问题能在一定时间内解决，那么就也能在同样的时间内检验它的解答是否正确。因此P包含于NP内。计算机科学理论中最重要的一个尚无定论的问题就是：这么多NP完全问题是否实际上可以在多项式时间内解决？还是真的需要指数时间？亦即，P是否等于NP

① 安德鲁·怀尔斯（Andrew Wiles，1953—）是英国数学家，居于美国。他于1979年在剑桥大学获得博士学位。1994年他证明出困扰数学家三百多年的费马大定理，是数学上的重大突破。

② 皮埃尔·德·费马（Pierre de Fermat，1601—1665）是法国律师和业余数学家。他在数学上的成就很高，对数论、光学、微积分、解析几何、概率论等多个领域均有所贡献。费马大定理也称费马最后定理，称对任何大于2的正整数n，以下不定方程没有正整数解：$x^n+y^n=z^n$。

古克：我们无法证明这些东西，这也是数学的这个领域里令人沮丧的地方。我们也无法证明P不等于NP。因此可能会有人发明某种聪明的并行算法，可以在多项式时间内解决某个NP完全问题。

在这种情况下，很难去定义进展。我们还没有接近成功，不过也有了部分的成绩。人们正在从多个不同的领域试图攻克这一难题。顺便说一句，P=NP是可能的。

列文：这么多世纪以来，几乎所有著名的数学猜想都得到了解决，这一事实就是一个有力的证据，说明NP完全问题可以在多项式时间内解决，而不需要指数时间。数学家们往往认为历史证据表明NP需要指数时间，我认为他们完全搞反了方向。[①]

① P/NP问题已于2000年被列为七大千禧年大奖难题之一。美国克雷数学研究所宣布了规则：所有难题的解答必须发表在数学期刊上并经过各方验证，只要通过两年验证期，每解破一题的解答者将颁发奖金100万美元。这七大难题是呼应1900年德国数学家大卫·希尔伯特在巴黎提出的23个历史性数学难题，一百年后许多难题都已获得解答。七大难题还包括霍奇猜想、庞加莱猜想及黎曼假设等。其中庞加莱猜想已于2006年被证明。

2010年8月，惠普实验室的科学家维纳·迪奥拉利卡（Vinay Deolalikar）声称已经解决了P/NP问题，有些遗憾的是，结论是P≠NP，并公开了长达100页的论文，目前业界对此的看法似乎已经趋于认同。

第三部分

架构大师：如何构建更好的机器

主导一个庞大的项目也许是件令人头疼的事。你必须时刻鼓舞大家的信心，哪怕自己心存疑问。你必须勇于尝试各种想法，即使它们尚未得到验证。如果你太保守，就会惨遭市场淘汰。如果你太激进，又可能一无所获。但如果项目能够圆满完成，成功的甜美将令你终生难忘。

而对于计算机系统设计师来说（这里的“系统”指的是组成计算机的一个或多个处理器、网络系统及其运行的软件），即使是成功，也可能转瞬即逝。总会出现更先进的产品或技术，人们总能对系统组织产生新的想法、在计算机构架中找到可改进之处。行业进步像文学作品一样依赖于大众的传播和改进，但每当改进后的思想以新产品的形式问世时，最初的奠基人却常常会被遗忘。本书这一部分介绍了三位计算机设计师，正是他们的思想构成了当今以及未来计算机架构哲学的中枢。

弗雷德里克·布鲁克斯[①]在20世纪50年代后期获得了哈佛博士学位，随后加入了IBM公司。当时IBM在计算机产业中的地位尚未巩固。他的第一项工作是协助设计实验用的Stretch计算机。布鲁克斯提出了可屏蔽中断的概念，这是一种控制处理器与键盘、触摸板、声音传感器、硬盘等外围设备之间交互的方法。如今不管是掌上电脑还是巨型计算机，都借鉴了布鲁克斯的思想。

20世纪60年代初，布鲁克斯领导了IBM 360系统的软硬件设计，催生了一系列带有可交换软件的设备，让IBM在随后25年中一直盘踞计算机行业的统治地位。1965年他离开了IBM，在北卡罗来纳大学创建并领导了全国首批计算机科学系之一。在此之后，他和同事们为当今所谓的虚拟现实领域进行了大量基础性的设计研究。

伯顿·史密斯[②]是道路设计师和化学家的后代，也出生于北卡罗来纳，但在新墨西哥州长大。20世纪50年代初他还是一名聪明但不爱学习的学生，在科学课上设计弹射纸团的装置，在家制作

---

① 弗雷德里克·布鲁克斯（Frederick P. Brooks, Jr.，1931—）年仅29岁就主导了被称为人类从原子能时代进入信息时代标志的IBM 360系列计算机的开发工作，取得辉煌成功。著有经典文集《人月神话》（*The Mythical Man-Month: Essays on Software Engineering*）。1985年获得美国国家技术勋章，1999年获得图灵奖。参见本书第10章。

② 伯顿·史密斯（Burton J. Smith，1941—）1967年新墨西哥大学电子工程学士，1972年麻省理工科学博士。他目前是微软技术院士（微软内部在技术方面设立的最高级别称谓，目前全球只有20位左右）。参见本书第11章。

带夜光按钮的半导体收音机。平淡的学业进行到一半，他就辍学加入了美国海军。当他离开校园后才意识到自己愿意设计电子设备。1972年于麻省理工研究生院毕业后，他经历了一段短暂的学术生涯，但很快就转而设计自己的第一台超级计算机。他的设计抛弃了传统思维——处理器无需本地存储器，相互之间每隔几纳秒便交换一次任务，而且他设计的网络非常简单，速度却快得惊人。1988年，史密斯开始设计每秒执行数万亿条指令的机器。无论成功与否，他的思想已经被世人所接受。

W.丹尼尔·希利斯[①]还是麻省理工的一名研究生时，就开始思考一个简单的问题：人类大脑的神经元细胞每秒最多能执行数千次计算，何以能在众多任务中表现远胜每秒执行数万亿次电路切换的计算机呢？他认为答案在于大规模并行：大脑中的“处理器”更多，而且它们能够很好地彼此协作，完成面孔识别、抽象类比等工作。他设计了一种带有数百万个微型处理器的计算机，希望把大脑的功能集成到硅片上。计算机科学界的权威们认为这一想法不可能取得成功，但像CBS总裁威廉·佩利[②]这样的风险投资商们仍然为他提供了资金，而且诺贝尔奖得主、物理学家理查德·费曼也曾参与了希利斯的“连结机器”（Connection Machine）的第一个网络的设计工作。经过数个模型之后，希利斯的机器便得到了广泛的应用，包括从石油勘探到股票价格预测等许多方面。他已经证明，只要问题足够大、网络运行足够快速，那么大规模并行就是有效的。如今希利斯的研究重新回到了生物学之上，他试图用自己的机器模拟生命的代代相传，从而揭示进化的本源机制。

计算机的架构和房屋建筑学颇为类似。思想和观念从争论中产生，寻求认同，而一旦被接受就抛弃了自己的奠基者，然后再与更先进的想法和技术重新结合。在民用建筑中，每隔几个世纪才会出现新的、重要的观念，而它们的起源早已迷失在神话之中。在计算机架构中，这种新的观念每5年就会出现。下面让我们来听听这些观念的奠基者们（或者至少是其中的3位）讲述他们自己的故事。

① W. 丹尼尔·希利斯（William Daniel Hillis，1956—），美国发明家、企业家、作家。他在美国拥有34项专利，同时也是思考机器公司（Thinking Machines Corporation）的首席科学家与联合创始人、并行超级计算机“连结机器”的设计者。参见本书第12章。

② 威廉·佩利（William Paley，1901—1990）是哥伦比亚广播公司（CBS）董事长和创办人，大企业家。CBS正是在他手中崛起并成长为美国著名电视网之一，因而也被称为CBS之父。

10

弗雷德里克·P.布鲁克斯 成功带来的愉悦

> 最好是研究与他人提出的问题相关的问题，因为它们会让你脚踏实地。
>
> ——弗雷德里克·P.布鲁克斯

P.布鲁克斯

科学家们在自己的实验室里就像是神一样。他们决定游戏的规则——提出哪些问题、使用何种方法，而唯一的义务就是向同行们报告自己的进度和结果。大多数科学家，包括许多杰出人物，都宁愿让他人把结果从理想的实验室带入到嘈杂的外界去。但弗雷德里克·布鲁克斯并不是这样。

1961年，弗雷德里克·布鲁克斯希望为IBM公司的一个部门设计并建造一条计算机专业产品线，但在争论中走进了死胡同。公司的管理层想要的是贯穿所有部门的单一产品线。令布鲁克斯惊讶的是，公司要求他来负责这个项目，称为IBM 360系统。他的团队成功了，让IBM在随后30年中一直处于计算机领域的领袖地位。

离开IBM之后，布鲁克斯来到北卡罗来纳大学教堂山分校，创建并领导了计算机科学系。他同时也领导了一个图形化的研究项目，为如今被称为虚拟现实的行业领域奠定了基础。在布鲁克斯这两段职业经历中，他的学术宗旨一直未曾变化，那就是解决他人的问题，并让自己的解答为他人所用。

> 如果有人对你说："你做的东西很不错，但它对我一点帮助都没有。"这其实是在提醒你要脚踏实地。

从布鲁克斯的言谈中，我们不难听出他来自北卡罗来纳州的温和乡音。实际上，他成年后的大部分时光都在老家附近度过。布鲁克斯1931年出生于北卡罗来纳的达汉姆，在附近一个小镇格林维尔长大。他是一名医生的儿子，很早就表现出对科学和计算机的兴趣。

> 我13岁时（1944年）在杂志上看到了哈佛大学马克一号[1]的报道，从此就迷上了计

---

[1] 马克一号（Mark I）是美国第一台大型自动数字电脑，被认为是第一部通用型计算机。它的全名是全自动化循序控制计算器（Automatic Sequence Controlled Calculator，ASCC），由于使用了3000多个继电器，又称继电器计算机。马克一号是它的用户哈佛大学为它起的名字。其后IBM制造了另一台大型计算器SSEC（参见本书第1章）。

算机。我孜孜不倦地阅读一切能找到的资料，而且开始在破产拍卖会上收集老式的商用机器。

马克一号是IBM和哈佛大学在战时合作的产物，能够根据战舰炮弹的重量、发射角度、风向和速度等因素计算弹道轨迹。这台庞然大物（50×10×8英尺）开动时，电子计数器会发出巨大的噪声，但就算是如此热烈的运转，它的速度还是比现在的个人计算机要慢上大约1亿倍（它的频率是3赫兹）。布鲁克斯在高中时正好赶上了战后大众对于电子产品的痴迷。

> 我们的高中有一个无线电俱乐部和一个电子工程俱乐部。高中的科学老师罗宾逊先生非常好，他鼓励我们参加各种各样的课外活动。我们为校剧院做音效，为舞会配乐。
>
> 在暑假，我靠砸铁皮挣生活费——为烘焙烟叶的谷仓做烟道排气管。格林维尔就在烟草产地的中心。

在杜克大学，布鲁克斯主修物理、辅修数学。他的毕业设计是用真空管制作一个闭路电视系统。虽然他很喜欢物理，但一直被另一个更加年轻的领域所吸引。

> 我向我的物理教授哈罗德·刘易斯（Harold Lewis）解释自己的最大兴趣是计算机。他说："你不能从一楼进入这个领域——现在已经太迟了。但你还来得及从二楼阳台赶上它。"

二楼阳台就位于哈佛大学，还有霍华德·艾肯[①]，正是他设计了布鲁克斯数年前读到过的马克一号。在马克一号之后艾肯与IBM分道扬镳，因为他想采用电子真空管，而IBM却坚持继续使用机电继电器。艾肯给年轻的布鲁克斯留下了深刻的印象。

> 他是个强悍的人，有6英尺3英寸高（1米90），双眉倒竖，还有斯波克[②]那样的尖耳朵。
>
> 当他生了气、居高临下地盯着你时，你会感觉自己面对的是魔鬼的化身。平时他总是催着我们前进。在我们写论文期间他每天都会到办公室来，要求读我们前一天的成果。这就是我和其他一些同学能在三年内就完成博士课题的原因之一。

这些同学里面包括肯尼斯·艾佛森，简明有力的编程语言APL的发明者。为此他于1979年荣获图灵奖。[③]

1954年，布鲁克斯准备开始做他的论文时，艾肯的实验室已经完成了最新的计算机，为美国空军设计的马克四号。和它的前辈一样，马克四号也被用于科学计算和工程。

艾肯深信商用计算需要一种完全不同的设计，并要求布鲁克斯进行这方面的尝试。布鲁克斯

---

① 霍华德·艾肯（Howard H. Aiken，1900—1973）于1944年成功研制了马克一号。离开IBM后他又开发了马克二号（一号的加强版）、三号（使用电子真空管）和四号（全部采用电子元件）。

② 斯波克（Spock）是美国影视系列《星际迷航》（Star Trek）的主角之一，也是美国热衷的星际迷航文化中的代表性人物。他在进取号星舰上担任科学官及大副，是一个外星混血儿，耳朵、眉毛和发型都很有特色。

③ 艾佛森和布鲁克斯合作了一本书《自动数据处理》（*Automatic Data Processing*）。在约翰威立出版公司的建议下，他们总共合作了两本书，第二本《APL语言》（*A Programming Language*）的作者署名只有艾佛森，不过在前言中对布鲁克斯进行了致谢。——原书注

的创新设计让他在1956年获得了IBM的垂青，参与了雄心勃勃的Stretch计算机计划。

> 从 1961 年 Stretch 首次交付开始，它就是世界上最快的超级计算机，直到 1965 年 CDC 6600[1]出现为止。他们一共造了 9 台 Stretch，每台耗资大约 1 千万美元。它被设计为纯科研用超级计算机，第一台送到了洛斯阿拉莫斯科学实验室（高能物理和武器实验室）。
>
> 我们为国家安全局制造了一台字符处理外设（扫描仪），叫做 Harvest（收获），大概有 Stretch 的三倍大。这是一台令人印象深刻的机器。

Stretch开创了现代计算机架构的许多新思路。首先就是吉恩·阿姆达尔[2]发明、约翰·科克[3]和赫伍德·考尔斯基（Harwood Kolsky）协助开发的指令流水线。

计算机中的"流水线"类似于生产装配线。在汽车装配线上，不同的汽车组件在各自的装配阶段同时加工着。与之相似，流水线计算机中也有一系列四则运算流水作业。这样做的净效应就是用一个步骤的时间就能完成整个计算，花的时间比从头开始计算出整个结果要少得多。

科克日后又领导了精简指令集计算机（Reduced Instruction Set Computer，RISC）的开发，为日后的PowerPC芯片[4]奠定了基础。而布鲁克斯和杜拉·斯维尼（Dura Sweeney）则发明了可屏蔽中断的概念。

"中断"就像电话来电一样。假设你是个一心往上爬的年轻经理，在小隔间的座位上有几部不同号码的电话。为了便于往上爬，你根据不同人士在公司中的地位（也就是对你晋升的重要性）告诉他们不同的号码。自然，给总裁的是电话1的号码，副总裁是电话2的号码，依次类推。如果总裁来电，你就会忽略或"屏蔽"其他电话的铃声。如果你手头上有重要的事，甚至不想让总裁打扰你，那么就会屏蔽所有的电话。

事实上今天所有计算机都采用了这种方案。埃德加·科德[5]在Stretch上采用中断子系统，构建了第一套交互操作系统。它允许用户在键盘上打字，同时在屏幕上看到输入的字符。这一功能在如今司空见惯，但其实敲键需要中断计算机的运行，同时又不能造成电子混乱。这就是保持计算机正常运转的屏蔽。

---

① CDC 6600是控制数据公司（Control Data Corporation）生产的大型计算机，由著名的计算机设计师西摩·克雷（Seymour Cray，1925—1996）设计。它被视为第一代成功的超级计算机，运行速度比前辈Stretch快3倍。

② 吉恩·阿姆达尔（Gene Amdahl，1922—）被认为是有史以来最伟大的计算机设计师之一，他一手缔造了IBM 360系统的辉煌，在离开IBM后又创建了自己的阿姆达尔计算机公司向IBM挑战，开拓了IBM大型机兼容机的全新市场。他的每一件作品都被同时代的人誉为一流杰作，被《计算机世界》列为"改变世界的25人"之一。

③ 约翰·科克（John Cocke，1925—2002）在计算机架构、编译器优化及精简指令集（RISC）计算机开发等方面贡献卓著，1987年获得图灵奖，1991年获得美国国家技术勋章，1994年获得美国国家科学奖，2000年获得本杰明·富兰克林奖章。

④ PowerPC是一种RISC架构的中央处理器（CPU），设计源自IBM的POWER（Performance Optimized With Enhanced RISC，增强RISC性能优化）架构。

⑤ 埃德加·科德（Edgar F. Codd，1923—2003）是一位英国计算机科学家。他在IBM工作期间首创了关系模型理论。1981年因在关系型数据库理论方面的贡献而获得图灵奖。

Stretch项目还在计算机辞典中留下了永恒的一笔。首席架构师维纳·巴克霍尔兹（Weiner Buchholz）创造了词汇"字节"（byte），表示能够容纳一个字符的0和1的序列（他建议长度为8）。一个8比特（bit）的字节能储存256个可能的值，足以表示绝大多数欧洲语言字母表中的所有字符（字母、数字和标点符号）。如今我们购买900千兆字节容量的硬盘时（用销售商的话说也就是900G），所用的"字节"一词和Stretch的设计者们是一个意思。

20世纪50年代Stretch项目的步调放缓，IBM由于成本低廉的7090硬件和高级编程语言Fortran（参见第1章）在科研计算机产业居于领先地位。但是在占市场65%的商业数据处理领域，竞争仍然激烈。美国无线电公司、通用电气、尤尼瓦克（Univac）等对手都在虎视眈眈。

1960年，在公司研究中心待了一小段时间之后，布鲁克斯被提拔为纽约州波基普西市的系统架构主任。IBM在这里设计了（而且依然在设计）他们最巨型的机器。系统架构师决定了机器的总体设计，包括需要的存储器数量、可处理的数据类型、执行指令的细节以及如何管理指令流水线等等。布鲁克斯的团队提出了一种以Stretch为基础的设计方案。

> 高层认为公司不需要为单独一个部门建立产品线，他们认为新的产品线应当贯穿多个部门。我的上级就在当晚被换掉，由鲍勃·埃文斯[①]继任。

布鲁克斯极力反对这一决定，但以失败告终。

> 斗争最终在1961年6月尘埃落定。鲍勃·埃文斯把波基普西实验室的所有经理人员召回到了萨拉托加斯普林斯，并根据最终确定的项目为每个人重新指派了新的任务。我去那里是为了保证我的手下不受伤害。我以为自己会被踢回研究中心。让我惊愕万分的是，他们要我来负责这条新的产品线——不是当架构经理，而是项目经理，全面负责工程、市场、程序和架构等各方面。
>
> 在这次斗争中，鲍勃·埃文斯和我之间的矛盾的确很激烈。有一天他打电话给我说："我希望你知道你获得了晋升。"我说："那么，谢谢你。"然后他说："我希望你知道这并非我推荐的。"

布鲁克斯组建了他能找到的最好的团队。不仅包括部门内最好的架构师，例如杰瑞特·布劳（Gerrit Blaauw），还包括吉恩·阿姆达尔（他日后创办了自己的公司，阿姆达尔计算机公司，和IBM竞争）、雅各·约翰逊（Jacob Johnson）和伊莱恩·勃姆（Elaine Boehm），他们都曾在IBM研究中心设计过一种叫做ABC的机器。同年12月，他们完成了高阶设计阶段。一个月后，公司管理层批准了这一开发计划，也就是后来的IBM 360计算机系列。

核心概念非常简单：如果用户为360系列中的某台机器编写了程序，那么该系列中所有更大的机器（以及大多数更小的机器）都能运行这段程序。这意味着一个发展中的公司可以先购置一台较小的机器，为其编写软件，然后在发展需要时把同样的软件用于更大的机器上。（对于经济

① 鲍勃·埃文斯（Bob O. Evans，1927—2004）是IBM大型机的先驱人物。1951年加入IBM公司，后领导团队研发出了IBM 360系统。1985年获得美国国家技术勋章。

大繁荣的20世纪60年代，这一特点很有吸引力。）为了保持这种系列共同性，所有机器都必须以完全相同的方式处理同样的指令，而且所有的支持软件都必须按照标准设计。

在今天，这种思想不足为奇。不止IBM，英特尔、苹果和绝大多数其他计算机公司都支持这种标准化指令的思想。但在20世纪60年代早期，没有公司愿意采取这种姿态。美国无线电公司的工程师就曾断然拒绝了设计一系列互操作机器的建议。他们不耐烦地向自己的经理解释说，不同的机器“不能”运行同样的指令集。（与之相反，吉恩·阿姆达接受埃文斯的邀请参与设计指令架构的前提就是，他要为所有360机器统一设计一套指令集。）

很明显，这一次IBM对市场的偏好战胜了偏好技术的美国无线电公司和其他竞争者。硬件设计一帆风顺，IBM已准备好在1964年初发布360系列的7种机型。至少他们以为如此。但是操作系统的开发工作却令人绝望地止步不前。

> 我找到埃文斯说：“让我到另一个部门去，因为那里的漏洞很大。”他说：“祝你好运。”于是第二年我便去了程序系统部负责操作系统的设计。
>
> 我做的第一件事就是让团队整个推倒重来，提出一个新的操作系统计划。

在新的设计中，最基本的技术革新就是为系列中的所有机器都配置至少一个磁盘，而不是只有磁带。任何快进过录音机磁带的人都知道，磁带是一种“顺序”设备。在磁带上从x点移动到y点需要缠绕过所有的中间点。而磁盘则是一种“任意访问”的设备，从x点移动到y点只需将读写磁头移动到合适的轨道即可。这种操作更快，而且不依赖于x和y点间的距离。埃德加·科德在Stretch上就使用了磁盘，布鲁克斯和他的团队受此启发，意识到通过磁盘可以运行比物理存储器大得多的程序（该系列机器中最大的存储器也不到1兆字节）。

1965年4月，360操作系统首次交付使用。它运行于7种不同的机器，最小的机器主存储器只有32K字节，较大的机器主存储器有500K字节（1/2兆字节）。即使是其中最大的机器，速度和存储器都不到今天个人电脑的1/1000，但当时的各种企业都依靠它们来帮助经营。这些机器销量极佳，很多IBM的经销商在刚到货数小时内就把一年的配额都销售一空。

7种机器的硬件设计总成本大约是2亿美元。由于它们都运行同一套核心指令集，7种机器分摊了3.5亿美元的软件开发成本。设计者们弹冠相庆，而且预感到将来还会有更加美妙的事情发生。

> 1965 年我们就在论文中预测，这种架构将会持续多年，实施于多代机型，而且必然会达到 32 位地址。

这种架构实际上已经持续了40多年。20世纪80年代中期，32位地址（能从2GB内存中获取数据）就已经成为了标准，而64位地址的机器如今也正在成为主流。

至于360操作系统，它的直系后裔，多任务虚拟存取器（Multitasking Virtual Storage，MVS），至今仍在多数IBM大型机上运行。

在这个项目的管理过程中，布鲁克斯学到了软件开发的一些教训，并将其写入了他的经典文

集《人月神话》之中。布鲁克斯在书中列举了奥维德、杜鲁门等各种管理者，谴责了当时企业软件管理中的浪费行为，尤其是为了加速完成任务而粗暴添加人数的策略[①]。他写道："不管安排多少女人，也得要9个月才能生下孩子。"同时他也发展了一套技术项目领导的理论。

> 我认为，在项目经理之外专设一个系统架构师是很重要的。在架构的实施过程中，有一个首席设计师以个人智慧掌控整体设计也是同等重要的。这是让设计获得竞争力的唯一途径。
>
> 在《人月神话》中我曾说过要制造软件，然后适时扔掉。但我不会再做出这种言论。现在我会说，先构建一个最小限度的版本——将其投放到市场、开始收集反馈，然后再逐步增加功能。先制定规范、然后再构建和测试的瀑布式模型对于软件设计来说显然是错误的。与用户之间的交流对于规范的制定是极为关键的。我们必须在构建和测试的过程中不断地完善规范。

瀑布式模型的流程是从抽象的规范开始，然后构建程序，最后再进行测试。它仍然是艾兹赫尔·戴克斯彻、莱斯利·兰伯特等计算机科学家所选择的方法，目的是从可证明的正确规范中衍生出正确的程序。对于安全第一的软件来说，这确实合乎情理——谁也不能等着先看飞机失事，然后再修改飞行控制系统。但对于研究软件或者办公软件来说，布鲁克斯等人都主张尽快从最终用户那里获取设计反馈。

无论如何，布鲁克斯在《人月神话》中对编程这一创造性行为表达了崇敬。

"正如孩童喜欢玩泥巴做馅饼一样，成年人也喜欢创造事物，尤其是自己设计出的事物。我认为这种快乐正如上帝造物时的快乐，体现在每一片叶子、每一片雪花的独特与新奇当中。"（语出《人月神话》第7页。）

1964年，布鲁克斯决定离开IBM，前往北卡罗来纳大学教堂山分校建立计算机科学系。他这一举动的背后有着深刻的个人原因。

> 最根本的原因在于，我是一个基督徒，主要我们去哪我们就去哪。我感到有一个清晰的召唤让我接受这份工作。要想帮助别人成长，在大学里比在他们就业后要容易得多。这是召唤的其中一部分。一部分是为了教学，另一部分则是我渴望更加接近技术工作。

在当时，美国只有一个真正意义上的计算机科学系，就在普度大学。其他学校的计算机科学系都被划归于数学系或电子工程系。布鲁克斯很快就体会到学术独立带来的好处。

> 我们可以自由雇人，可以根据计算机科学的标准来选择研究方向，而不是按照其他学科的标准来选择。

---

① 《人月神话》是布鲁克斯的代表作，全名《人月神话：软件项目管理之道》（*The Mythical Man-Month: Essays on Software Engineering*），全书讲解软件工程、项目管理相关课题，被誉为软件领域的圣经，内容主要源于布鲁克斯在IBM 360系统中的项目管理经验。该书于1975年首次发行。这一句中提到的奥维德（Publius Ovidius Naso，公元前43—14）是古罗马诗人，与贺拉斯、卡图鲁斯和维吉尔齐名。代表作有《变形记》等。他曾当过警官和法官。杜鲁门（Harry S. Truman，1884—1972）即美国第33任总统（任期1945～1953）。

同时我们也决定，规模较小的系应该遵循斯坦福工程学院院长弗雷德里克·特曼[①]所提出的“优势峰值策略”——针对少数研究领域，建立足够数量的教师队伍，放弃其他的子学科。

我曾考虑过软件工程。但是在我看来，过去和当今软件工程中最重要的问题在于规模：不在于如何编写单个的小程序，而在于如何掌控大型软件的复杂性。而大学并不是进行大型软件项目的好地方。

于是我们决定选择两个方向：计算机图形软件和自然语言处理。

他的这一决定来自于Sketchpad的发明人、伊凡·苏泽兰吹响的革命号角。

1965年，我参加了IFIPS[②]三周年会议，苏泽兰在会上公开提出了他的伟大构想。直到如今，人们依然在追求他的这一目标。

他说：“把屏幕当做进入虚拟世界的窗口。计算机图形研究的任务就是让这一窗口中的画面显得真实，不管是在视觉上、听觉上、交互上还是感觉上。”我回答道：“这正是我们的目的。”从那时开始，我就投身于这一挑战。总有一天我们会成功的。

苏泽兰所描述的便是如今我们所称的虚拟现实。但在1965年，硬件的限制让他的梦想看起来遥不可及。

要想得到连续运动的视觉效果，计算机必须达到电影的速度，每秒至少显示24帧画面。只要“观察者”在世界中移动，那么每一帧就都需要重新计算。

布鲁克斯的同事亨利·福克斯（Henry Fuchs）、约翰·普尔顿（John Poulton）在20世纪90年代初开发了一种当时最高级的计算机，叫做**“像素飞机”**（Pixel-Plane）。它有成千上万个处理器，每秒钟能够绘制出数百万个彩色立体三角形。就算是这种机器也只能提供中等大小的分辨率。而它绘图的速度比1965年苏泽兰那时候的机器要快100万倍。

所以在布鲁克斯组建他的系时，全运动视频是不可能的。但他认为只要以三条准则为基础，团队就可以向这个目标迈进。

首先，要从适合于当时技术水平的图形问题入手。其次，他坚持计算机图形不仅要模拟现实，还得实现智能增强[③]的目的。

1969年，我找到院长查尔斯·J.莫罗教授（Charles J. Morrow）说：“目前我们有条件而且有必要建立一种智能增强系统，让人类思维和机器合作处理一些复杂的问题。我们的教师队伍中有哪些人的智力值得增强？”

然后我解释说我需要一类含有大量几何图形内容、计算内容和模式识别的问题。

---

① 弗雷德里克·特曼（Frederick Terman，1900—1982）是美国学者，正是他成立了斯坦福研究园区，最初被称为“果树林”，后来逐渐发展成今日高科技企业云集的硅谷。因此他也被称为“硅谷之父”。

② 国际信息处理联合会，International Federation for Information Processing Societies。

③ 智能增强（intelligence amplification）也称为认知增强（cognitive augmentation）或机器增强智能（machine augmented intelligence），指运用信息技术对人类智能进行增强，完成一些复杂问题的探索。

他说："之前没人问过我这个问题——让我回家想一想。"第二天他向我列举了一长串可能的合作者。

有操作驾驶模拟器的高速公路安全人员，设计低成本住宅、需要实时联机评估的城市规划者，比如如果需要拆迁，对项目成本会有多大影响。还有关注星系结构的宇航员，关心地下水储量的地质学家等等。

布鲁克斯在筛选时，为自己的研究订立了第三条准则：计算机图形不仅需要实现智能增强，还应当解决影响人民生活的实际问题。很快他就找到了一个符合这三条准则的问题。

莫罗提到的人当中有一位生物化学系的蛋白质化学家，杨·赫尔曼（Jan Hermans）。他对蛋白质折叠问题有一些自己的想法。

蛋白质折叠问题至今尚未解决，它是关于如何通过氨基酸序列得到蛋白质的三维空间形状的问题[①]。

我们就以这个问题开始了研究。北卡后来的教授威廉·莱特（William V. Wright）据此作了一篇论文，描述了如何在 IBM 2250 图形终端上实现可视化的分子三维模型。

与赫尔曼的合作项目完成之后，布鲁克斯的团队很快又找到了下一位合作者，杜克大学的晶体学家金荪和（Sun Ho Kim）。金想利用X射线衍射计算出核酸的详细三维分子结构。

在此之前，人们主要利用黄铜模型、镜子和带有电子密度剪切图的塑料片来进行研究——这种装置叫做理查德箱（Richards box）。

化学家们通常先根据假设构建出一个结构比例模型，然后不断调整，直到模型的电子分布与实验结果一致。我们很难想象还有比这更令人沮丧的工作。

这些玩意立在那里大概有一立方米的样子，比例是 2 厘米比 1 埃[②]。如果你有 3000 个原子，在摆放的过程中稍有不慎就会把其中的某些碰歪。就算完成了这一步，你还得用铅锤和米尺小心翼翼地测量每一个原子的三维坐标。

因此我们认为应该可以构建一个图形化的理查德箱，这会方便得多。这个系统在 1974 年成型，它帮助金找到了转移核糖核酸（tRNA）分子的原子坐标，无需任何黄铜模型。

金荪和以合伙人的身份与布鲁克斯的计算机团队合作。他的热心参与证实了布鲁克斯的观念：在开发过程中应当让最终用户参与。

---

① 蛋白质的基本单位为氨基酸，其一级结构就是氨基酸序列。蛋白质会因为所含氨基酸残基间的相互作用而折叠成一个立体的三级结构，而且只有一种三级结构。如果能从蛋白质的一级结构得到它的立体结构，那么就能推测它所对应的生物学功能，但这一过程很难。因此研究蛋白质折叠的问题，就是破译"第二遗传密码"——折叠密码（folding code）的问题。

② 埃（angstrom）是光谱线波长单位，即10^{-10}米。

金有时候会过来，每晚花8个小时来回移动他的分子模型，以图找到一种连贯的总体模式，在排布上能够满足他的电子密度测量结果。

后来国家卫生研究院的一个参观团来访时，他站起来宣布说，之前以季度为单位的数据拟合工作，如今只花一周就能完成。

1974年这次成功的试验也让布鲁克斯更加确信智能增强相对于人工智能的价值。

人工智能是以取代人类智能为目标，而我们则一直将人类智能放在系统的中心位置考虑。

如今的人工智能界在25年后意识到了这一思想的重要性。但这和他们的初衷并不一致。他们曾经说："我们肯定能够解决这些问题，并不需要人类的思维。"但实际上的确需要人类的思维。

布鲁克斯深信，人类在三种智能行为上将永远胜于计算机。

第一种是模式识别。1个月大的婴儿可以从之前从未尝试过的视角或光线条件下辨认出自己的妈妈。

第二种是判断评估。人们买车时会在各种车辆中来回浏览，根据复杂的多重标准进行选择——这些事情计算机很难胜任。

第三种是上下文背景搜索——利用表面上无关的事实。当然这就是所谓的天赋。

在这方面有一个很好的故事，是小托马斯·沃森[①]告诉我的。有一次他访问科罗拉多州夏延斯普林斯的北美防空指挥所，遭遇了警报。结果发现这是一场虚惊，原因是雷达采集了月亮升起的讯号。

当晚的指挥官是一位加拿大人。他说："噢，他们（苏联人）不可能今晚发动攻击。"于是有人问："为什么，长官？"他说："因为赫鲁晓夫现在正在纽约。"

我们可以为计算机系统精心构建各种规则，但它们仍然不会利用这些表面上无关的事实。

计算机是否真的永远不能完成这样的任务？本书提到的五位人工智能研究者对于机器的局限性都有着不同的看法。爱德华·A.费根鲍姆[②]本质上同意布鲁克斯提出的人与机器之间的共生伙伴关系，并由此定义了他的专家系统的研究领域。道格拉斯·莱纳特在他的Cyc项目中试图把足够多的日常生活知识集成到程序中，以此来解决布鲁克斯提到的上下文背景问题。约翰·麦卡锡研究从逻辑层面判断如何把不同的事实联系起来——也就是布鲁克斯所称的判断评估。而丹

① 小托马斯·沃森（Thomas Watson, Jr.，1914—1993）是IBM公司第2任首席执行官（1952—1971）。他的父亲大托马斯·沃森是IBM第1任首席执行官（1914—1952）。他是20世纪100位最有影响力的人物之一，也被称为是"历史上最伟大的资本家"。

② 爱德华·A.费根鲍姆（Edward Albert Feigenbaum，1936—）是大规模人工智能系统的设计和实现的先驱，被称为"专家系统之父"。他和拉吉·瑞迪（Raj Reddy）共同获得1994年图灵奖。参见本书第13章。

尼尔·希利斯的连结机器及其后继者则可能能够完成人类轻而易举的模式识别行为。然而这些勤奋的科学家们都尚未成功。

布鲁克斯的研究团队也开发了其他一些智能增强设备。化学家们利用头戴式设备和遥控式手臂可以在分子间遨游和操作（基因科技公司[1]很快就运用了这一技术），建筑设计师在尚未破土动工时就可以在建筑中行走，放射科医生能够提前规划照射肿瘤的射线方向。不过，他的团队对娱乐领域一直都冷眼相待。许多公司和学术研究中心都瞄准了这一有着巨大利润的市场，甚至包括“模拟色情”的成人服务……布鲁克斯坚决不涉足这一领域。

> 虚拟现实如果成为电视领域的扩展，我认为这对于个人和社会的发展是极为有害的。我会与其划清界限。让其他人做这些东西吧。

布鲁克斯的这番声明让他显得比其他同行更具道德意识，这可能是由于他虔诚的宗教信仰所致。

> 科学界有一些普适的伦理道德标准，而基督教则对此更加严格。它指引我们去做那些我们认为更加重要的事情。

① 基因科技公司（Genentech）是全球第一家生物工程技术公司，1976年创立，是生物技术产业的巨擘。

11

伯顿·J.史密斯与光速赛跑

几乎任何事物的速度都是可变的。任何计算机都能以某种速度模仿其他任何事物。

——伯顿·J.史密斯

一栋建筑如果经受住了人们审美和时间的考验，大部分荣誉都会归于它的设计师。对于设计优良的计算机来说也是如此。和建筑设计师一样，计算机的架构设计师必须考虑成本因素，在设计中权衡用户的需求。现代民用建筑一直在进行高度竞赛，而计算机架构则是速度竞赛——运行速度越来越快，内存也越来越大。

衡量速度最简单的方法就是检测计算机每秒能执行多少次操作，通常是指算术运算操作。现在的目标是每秒能执行1万亿次操作——比最快的个人计算机还要快大概1000倍。

即使是光，每纳秒（十亿分之一秒）也只能行进1英尺（30厘米左右），而电路中的电流速度甚至更慢，因此单个处理器的速度是存在着物理限制的。具体的限制是什么尚存争议，但从经济上说开销是显而易见的，速度每一次提高10倍都比上一次的提高要昂贵得多。

如果对于某项应用来说，单个处理器的速度过慢，那么要想提高速度，唯一的办法就是连接多台机器“并行”运行。也就是说，让一个处理器处理任务的一个部分，同时让另一个处理器处理另一个部分。但是任何学过组织行为的学生都知道，有些任务是不能按这种方法分解的：正如必须先铺设地基，才能在楼上安装电梯设施。

对于组织行为来说，任务只有在能被分解为独立的子任务时才是“可并行的”——子任务彼此之间依赖性不强。比如说，贷款申请堆积过多的银行可以雇用更多的贷款员来加快处理。对于计算也是一样。大多数科学计算都是可并行的，因为它们的子任务都只涉及局部现象，例如气流只会通过飞机机翼的一小部分。编程人员可以为问题的每一个部分指派一个处理器，然后再指定各个处理器之间如何通信。

但即使任务可以并行处理，也仍然存在着问题：有些是技术层面，但更多是逻辑层面。解决这些问题是当今计算机架构师们所面临的主要挑战。很少有个人具备这方面的全部工程和数学技能，而拥有一定设计想象力的人则更是凤毛麟角。伯顿·史密斯就是这少数人之一。他设计了两

种创新型的超级计算机：异质元素处理器（Heterogeneous Element Processor，HEP）和Tera机（“Tera”取自单词teraflop，表示每秒执行1万亿次浮点数或算术运算）。虽然这些机器短期内不会出现在你家附近的计算机店铺中，但它们所蕴含的思想却一直在高性能计算中扮演着重要的角色。

散乱的轨迹

伯顿·史密斯1941年出生于北卡罗来纳州的教堂山。他的家史显示出家族对工程和科学的偏好。

史密斯的祖父是一位土木工程师，曾设计过通往总统山[1]的道路。雕刻那些巨大总统头像的雕塑家格曾·博格勒姆[2]也是他们家的朋友。伯顿的叔叔也是一位土木工程师，他的父亲则是化学教授。1945年伯顿刚满4岁时，他的父亲被提拔为新墨西哥大学的化学系主任，于是举家搬到了新墨西哥的阿尔伯克基。在史密斯成长的年代，无线电广播依然是美国年轻人的潮流，他也未能免俗。

> 我勉强能做些修补工作。我用放大器做一些半导体和晶体管收音机。房间里所有的电灯开关都被我涂上了夜光涂料，方便晚上能找到——还有工具也是。天知道为什么我会在半夜去找螺丝刀，但我就喜欢它们都发出磷光的样子。

史密斯在无线电上花着大把的时间，而学院科学却无法打动他。

> 那时候的学校就没怎么好好教过科学。我还记得在初中时，我们几个后来从事科学的人整堂课都在琢磨怎么用塑料尺弹射纸团。我的弹射轨迹总是很散乱。

史密斯的父亲发现他对学校学习没有兴趣，于是把他送到了加利福尼亚的一所私立学校。史密斯在一次全州高中化学考试中获得了第三名，但仍然没有学习的动力。1958年他进入了波莫纳学院，但学得很差。1959年他转入新墨西哥大学，到了第二学期他甚至比高中时还要不安份了。

> 我当时学的是物理，结果发现自己对它越来越不感兴趣。我实在无法想像一个物理学家的生活会是什么样子。后来我逐渐发现实验物理还有点意思。我想学一些更令人兴奋的东西。这可能是我不爱学习的原因之一——以往的任何机遇都不能让我兴奋。我还没做好心理准备投入一项事业。

史密斯辍学加入了海军。他在潜艇上服役了4年，大多数时候是当通信技师。

> 我学到了不少东西，而且对重返校园感到非常兴奋。我觉得设计这些东西会很有趣味。

① 美国总统山即拉什莫尔山（Mount Rushmore），是一座美国总统纪念公园。园内有四座高达18米的美国著名总统头像，被认为代表了美国建国150年来最杰出的几任总统，分别是华盛顿、杰斐逊、老罗斯福和林肯。

② 格曾·博格勒姆（Gutzon Borglum，1867—1941）率领约四百名工人花了整整14年时间（1927年10月～1941年10月）雕刻完成这个杰作。他曾说：“美利坚将在这条天际线上延伸。”

史密斯回到了新墨西哥大学并获得电子工程学位，之后前往麻省理工攻读硕士。与此同时，他从1968年开始为亨缀克斯（Hendrix）电子公司做顾问，这是一家位于新罕布什尔州的计算机终端小型生产商，大部分员工都是麻省理工的研究生。在亨缀克斯，他的第一个硬件设计遭到了查克·赛茨（Chuck Seitz）的批评。赛茨后来是加州理工的计算机科学系主任，而且是少数可以和史密斯比肩的开拓性计算机架构师之一。

1972年史密斯获得了麻省理工的博士学位，之后回到了西部，在科罗拉多大学丹佛分校教授电子工程。很快，他就在德尼尔科（Denelcor）公司获得了一份顾问的工作。这是一家小型企业（年收入100万美元），但构想却很宏大，想制造通用的并行超级计算机。

装配线、向量和数据流

实现并行计算主要有两种途径：流水线并行方式（时间上并行）和多处理器并行方式（空间上并行）。

在弗雷德里克·布鲁克斯一章中我们曾提到，流水线就相当于计算机上进行数学运算的装配线（参见图11-1）。每个电路部件在一个时钟周期里执行一部分运算（例如加、乘等），然后将结果传递到下一个部件，依次类推直到运算完成。在任一时刻都有许多运算处于完成中的不同阶段，就像是汽车装配线上尚未完全装配好的汽车一样。

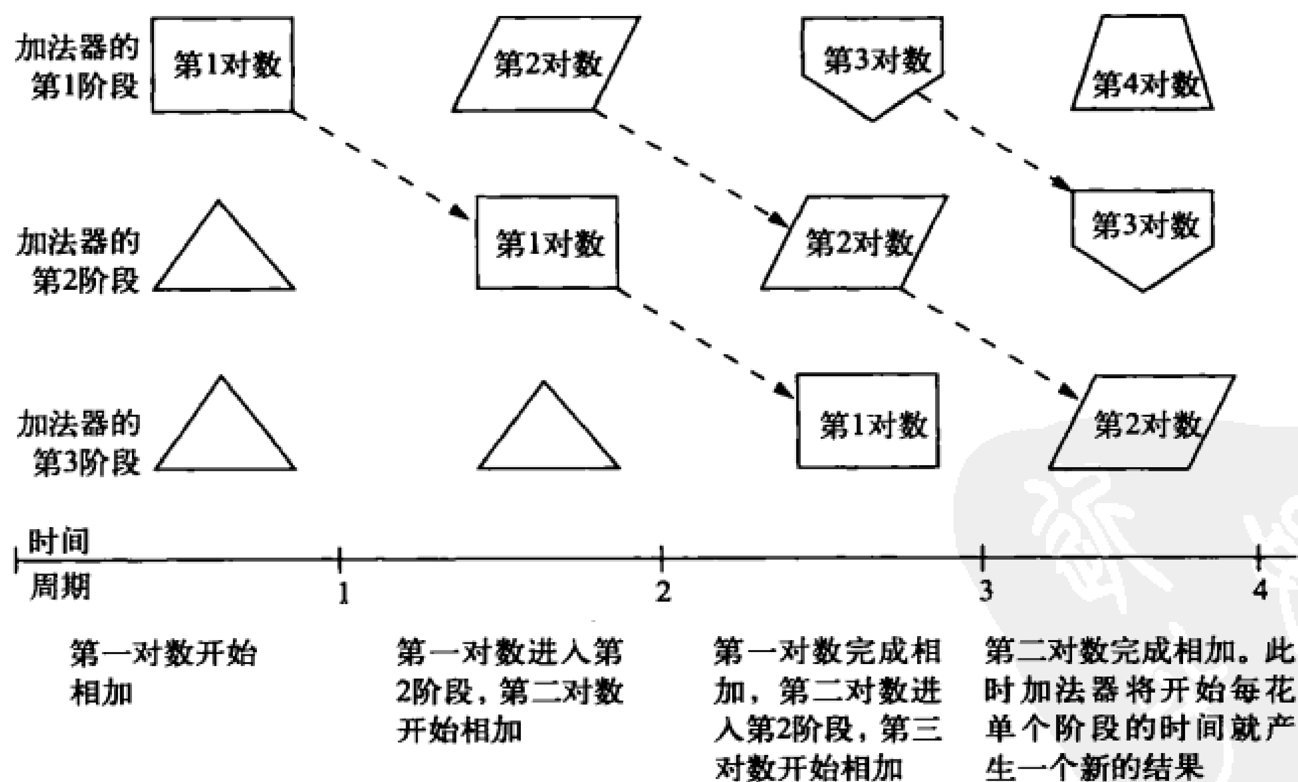


图11-1 计算机流水线。如果有多对数需要相加，加法器实际上只需单个阶段的时间就能得出一组运算的结果

比如说，在以前的Cray YMP计算机上，一个加法运算被分解为9个周期，每个周期时长为6纳秒。因此从头到尾执行一次加法操作需要54纳秒。然而，如果这9个阶段中每一个都在进行着

不同的加法，那么就会每6纳秒产生一个运算结果。这就好比从头到尾生产一辆汽车需要15个小时，但在装配线上每分钟就能产出一辆汽车一样。和装配线一样，流水线计算机适用于那些需要进行大批量相似运算的问题。这种并行方式正是向量处理器的基础。向量处理器又称数组处理器，由克雷研究公司[①]、控制数据公司和德州仪器公司于20世纪70年代设计。

向量就是一列数。因此将两个向量相加就意味着将两个长列中的数依次按对相加。这是流水线处理器的一个完美应用，只要向量足够长。（批处理的量必须很大，否则多阶段流水线的成本就会得不偿失——这也是向量未被用于商业数据处理的原因。）

对于气流建模、天气预测等需要大量向量计算的应用来说，向量机很受欢迎。用户都是科研机构的大鳄，包括美国海军研究实验室、美国地球物理流体动力学实验室以及洛斯阿拉莫斯国家实验室等。这些机构不像IBM在20世纪60年代和70年代迎合的那些柔弱的企业。如果有什么地方出现了问题，这些实验室的科学家能够自己想办法修好它。

比如说，克雷研究公司曾经将一台早期机器租给了洛斯阿拉莫斯实验室。这台机器没有操作系统也没有编译器，也就是说根本没有可支持的软件。洛斯阿拉莫斯的科学家将一个已有的编译器进行了修改并装到了这台机器上，而且非常满意最后的成果。但是当向量机在克雷和其他地方正风靡时，德尼尔科公司却想攻克一个由于向量过短而不适合使用流水线的问题：求解非线性常微分方程。这些方程常出现在控制应用中，例如在高速下航空航天的稳定性问题。这些方程内在的并行性依赖于特定的数据输入，因此程序员很难预见在哪里可以使用并行来解决问题。德尼尔科公司的梦想是让机器为程序员寻找并行。

史密斯和德尼尔科的同事们希望设计出的机器在输入刚刚完成时就能执行运算。这种方法被称为"数据流架构"（dataflow architecture），最早由麻省理工的杰克·丹尼斯（Jack Dennis）及其同事开发。在数据流架构中，何时进行何种运算是由数据的可用性决定的，而非固定顺序的指令。

数据流计算机类似急诊病房。不同症状的病人在不可预测的时间前来就诊，而大夫们则当机立断采取应对措施。在数据流架构中，数据和运算码进入后，某一个处理单元就会执行适当的操作（参见图11-2）。

分工不同的处理单元将并行执行运算，有一些进行乘除运算，有一些则进行加减运算。由于每个处理单元都进行完整的运算，因此这种结果被称为"空间上并行"，与流水线的"时间上并行"相对应。1979年，史密斯离开了科罗拉多大学，全职为德尼尔科公司工作。

> 促使我离开科罗拉多大学并全职为德尼尔科工作的原因，是我对存储器有了新的理解。能够容忍加法和乘法延迟（数据流思想）和不同步的机制同样也能容忍存储器的延迟。
>
> 不过在此之前我先获得了（大学的）终身教职，这样他们就不会说"我们知道你为什么离开"了。

① 克雷研究公司（Cray Research）正是Cray YMP计算机的设计厂商。它是由"超级计算机之父"西摩·克雷（Seymour Cray，1925—1996）于1972年创办的上市公司。

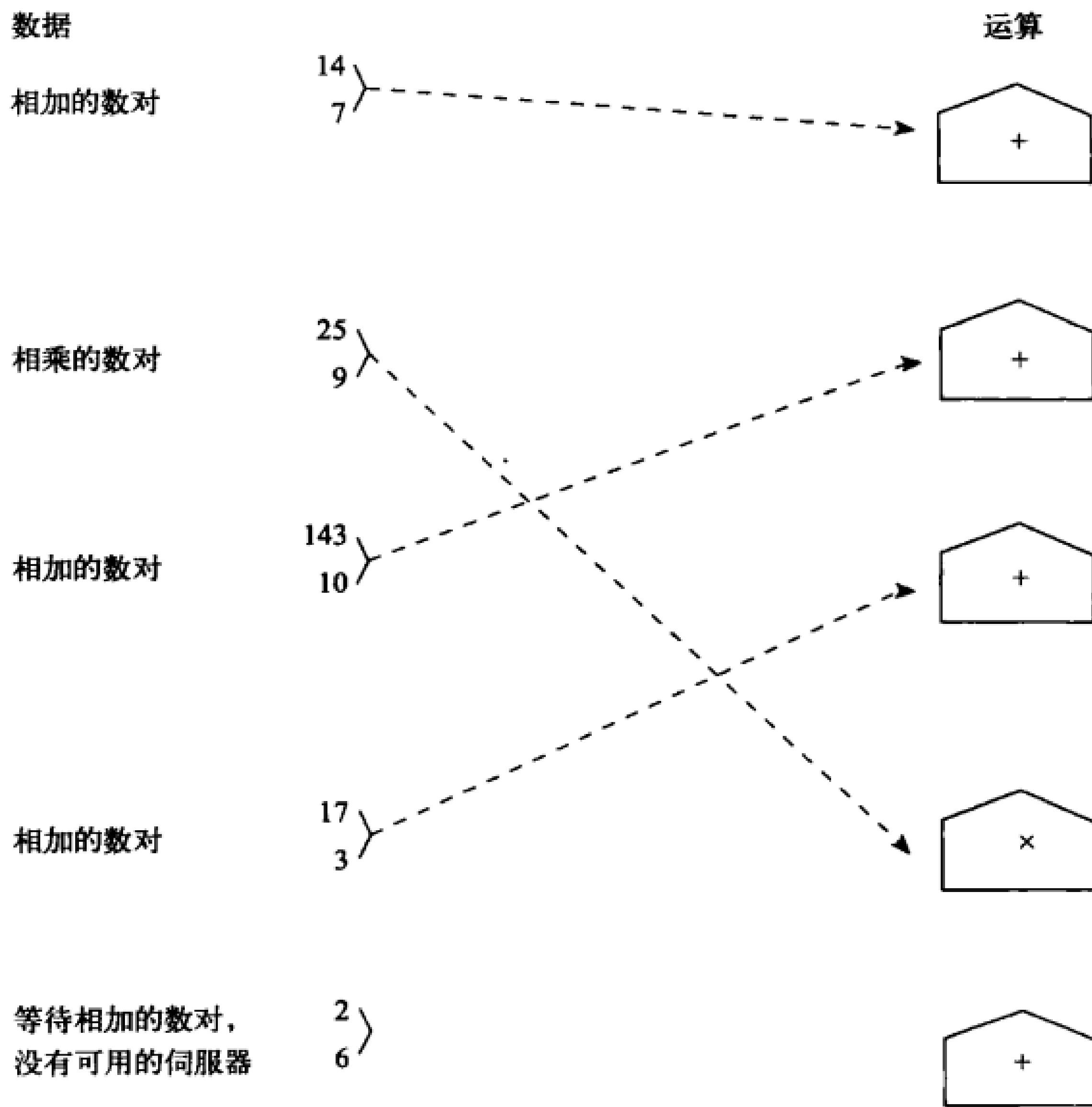


图11-2 在数据流系统中，多个电子伺服器都能执行运算（图中为算术运算）。
当数据元素就绪时，就被传递到某个当时可用的伺服器（只要有）

慢存储器和快处理器

“存储器延迟”（memory latency）是指处理器访问存储器芯片所花的时间。[①]这段时间约为100纳秒（千万分之一秒），看起来似乎不长，但它已是处理器芯片执行单条指令所需时长的20到50倍。由于许多操作都需要处理器访问存储器芯片，所以只有处理速度已经不足以确保运算的速度。

解决存储器延迟最常见的办法就是使用高速缓冲存储器（cache）。高速缓存由IBM于20世纪60年代发明，是一种高速但昂贵的存储器，访问时间只有执行单条指令所需时长的1到2倍。由于造价不菲，它的容量往往只有主存储器的1‰～1%。

① 随机存取存储器（RAM，Random Access Memory）芯片是一种没有活动部件的存储器设备。当笔记本电脑中的处理器寻找数据时，它会先搜索主存储器中的RAM芯片，然后再搜索硬盘。1998年能存储64M位信息的RAM芯片就已成为主流。——原书注

高速缓存保存着主存储器中数据项的副本。看起来这么小的存储器似乎帮不上什么大忙，但一个好的算法可以给最近读写过的数据项分配最高的优先级，从而把最有用的数据项都保存在高速缓存当中。当有新的数据项进入高速缓存时，算法就会把那些优先级最低的，也就是最近最不常使用的数据项驱逐出去。

这个简单的机制被称为“最近使用”（least recently used）策略，在单处理器上非常有效。哪怕高速缓存的容量只有主存储器的1%，就已经能满足日常90%的存储访问需求。

基于高速缓存的多处理器中，每一个处理器都有自己专有的高速缓存，这就会引起问题。由于每一个高速缓存都保留了主存储器中数据项的副本，不同的高速缓存可能（而且经常）会持有同一个数据项的多个副本。当一个处理器修改数据项X时，就必须通知所有处理器修改或抛弃X的副本，以免对过期的X值进行操作。

日常生活中也有类似的事情。当许多人一起撰写一个文档时，每个人都可能在自己的计算机上保留一份副本。如果一位作者修改了某个部分，就必须通知其他作者，以免他们对该部分进行不一致的修改。日常生活中的这种交流量很大，可能会出现错误。在多处理器计算机中，电路既昂贵，又容易出错。

在认识到高速缓存方案可能并不适用之后，史密斯推断真正的问题并不在于存储器的速度慢，而在于处理器在等待存储器返回数据时没有做有用的工作。因此他设计了异质元素处理器HEP，让每个处理器执行多项任务（最多可达64项）。

在一个时钟周期内，一个任务会执行一条指令，并发出下一条指令所需数据的请求。而在下一个时钟周期内，另一个任务可能会做同样的事情。因此等到前一个任务发出它的下一条指令时，也许是在64个周期之后，它的数据可能已经从存储器中返回了。史密斯理所应当地为这一想法充满了自豪感。

> 我找到了一种方法来构建机器，想要它多大就能有多大，而且程序员不必担心存储器中如何存放数据的问题。但是如何构建网络的问题依旧存在。

“网络”指的是处理器和存储器之间的连结。和连结城市的公路网一样，每一个交叉点都需要有交通路由逻辑。关键的挑战在于，要找到一种好的算法将每一个处理器发出的信息按路线发送到相应的存储器中，避免可能出现的电子拥堵问题，否则交换器就会饱和。传统网络使用了一种“储存转发”（store-and-forward）方案，交换器会将无法发送至目的地的信息储存起来，这些储存的信息被称为“队列”（queue）。但史密斯认为队列本身仍然存在问题。

> 如果队列满了怎么办？在我看来这样太混乱了。

战胜存储器延迟

在多数商用多处理器使用的高速缓存方案中，人们希望能将一次运算安排成处理器A的数据在高速缓存A中、处理器B的数据在高速缓存B中、处理器C的数据在高速缓存C中（参见图11-3）。当实际情况不是这样时，处理器就可能得等待100到1000个时钟周期。此外，内连网络

还必须让所有高速缓存的数据都保持一致。这种网络非常昂贵。

伯顿·史密斯的设计没有用到高速缓存，而是让处理器在等待远程存储器返回数据时执行其他任务。每个处理器都可能一次执行上百个任务。关键在于设计可以从一个任务迅速切换到另一个任务的处理器。

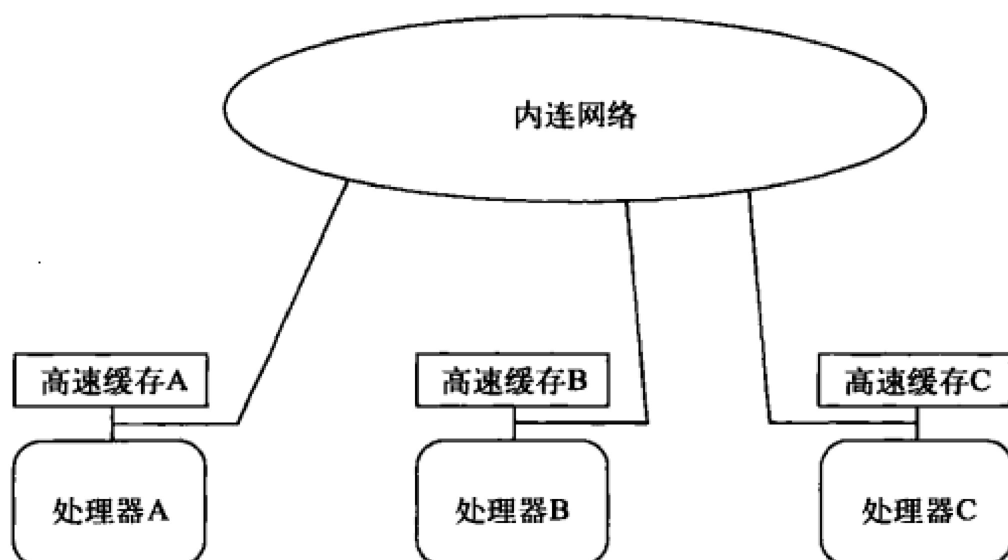


图11-3　基于高速缓存的机器的内连网络

热土豆路由

1977年春，史密斯产生了一个想法：如果到达交叉点的信息无法送达目的地，交换器就应该将其发送至某个地方（必要时甚至可以远离其目的地），从而避免产生电子拥堵。

兰德公司的保罗·巴兰[①]曾提出过一种用于电话的类似方案，称为“热土豆路由”[②]。在热土豆路由中，信息会不断运动，直到抵达目的地为止。这种方法不需要交换器队列，但允许出现错误路由：信息可以暂时远离目的地。史密斯提出了一种更为可靠的方案。

> 我在想，如果不停地将信息来回转发，就可以记载每条信息路由错误的次数。当次数到达最大值时，路由算法就发生变化，不与其他信息发生冲突，而是在网络中进行欧拉环游。

“欧拉”环游取名自18世纪伟大的瑞士数学家莱昂哈德·欧拉，这种路由从起点开始，恰好经过每一条边一次最后回到该起点。这种方法能保证每条信息都能抵达目的地。史密斯将其比拟

① 保罗·巴兰（Paul Baran，1926—2011）是波兰裔美国人，著名工程师，对电脑网络的建立作出了极大的贡献。他提出了分组交换的概念，曾参与过ARPA（高级研究规划署）网的建立。

② “热土豆”的英文是“hot potato”，引申义为烫手山芋，形容急于脱手之物。国内一般都称呼这一方式为热土豆路由，故译文从之。

为物理微粒的布朗运动[①]，并证明了这种比拟的正确性：

> 我考虑的是物体如何流动，而不是三维的抽象。

尽管HEP I型机有16个处理器，而且设计颇为巧妙，但它的速度仍然不足以打开市场——希望向量性能优越的人不需要并行数据流机器，而想要有并行机的人又觉得它不够快，因为它的处理器组件太慢了。即使HEP II型机提高了速度，但HEP I的推广还是非常困难。1985年德尼尔科公司终于破产。

尽管德尼尔科离去了，但通用并行计算机的思想却并未随之消亡。史密斯随后在美国国防分析研究所的超级计算机研究中心为Horizon机工作了三年。这种机器可以说是HEP的高速版本，每个周期可以执行多次运算。

Tera 机

1998年，史密斯加入华盛顿州西雅图的泰拉（Tera）计算机公司。据他回忆，他成为了公司的“第2号雇员”。

泰拉公司由德尼尔科的前任首席财务官吉姆·罗特索克（Jim Rottsolk）创立，宗旨仍然是创造通用超级计算机——具有每秒执行1万亿次操作的能力（相当于人脑的计算能力）。HEP只有16个处理器，而Tera机则是它的16倍：256个处理器排列在三维网格立方体中——16 × 16 × 16。它有512个存储器单元，每一个都有多个内存条，同时还有4096个内连节点。它的时钟周期（执行一次运算所需的时间）少于3纳秒（一秒钟的十亿分之三）。

拥有大量的处理器意味着在同一时间会有更多的讯息通行于网络中，导致更多的冲突以及更多不顺利的路由。同时由于处理器速度远超于存储器速度，这也就意味着有效存储器延迟可能会比50倍还要高。

有一种解决方案是将每个处理器执行的任务数量增加到64个以上。然而把一项巨大的工作分解成16 000个子任务（每个处理器64个任务 × 256个处理器）无疑是非常困难的。

在一次前往加拿大亚伯达省艾德蒙顿市的飞行途中，史密斯边吮着威士忌边思考抽象流水线。他突然得到了启示。

> 假设你想分配 8 件事，例如 5 只山羊和 3 只绵羊（动物表示向存储器发出的相互独立的请求），其实你真正要做的只是确定整个羊群是否都分配完毕就行了。我在飞机上恍若醍醐灌顶。我从座位上跳起来，向邻座的一位女士解释我的想法，而她根本不知道我在说什么。如果以群组的方式来考虑，Tera 机可以在一个指令流中发布多个存储器引用——多达 8 个。

① 1827年英国植物学家罗伯特·布朗（Robert Brown，1773—1858）利用显微镜观察花粉悬浮于水中所迸裂出的微粒时，发现微粒会呈现不规则状的运动，因而称其为布朗运动。这是一种正态分布的独立增量连续随机过程。它是随机分析中的一个基本概念。

史密斯对此的洞见是，对于一项任务，执行运算的顺序有时可以作出轻微的调整。因此对将来运算的存储器请求可以与对下一个运算的请求同时发生。相比之下，在HEP中一个任务每次只能发送一条请求，如果该请求被延迟，那么整个任务也会被延迟。

史密斯深信，这个简单的想法将使我们不再需要进一步细分任务。

> 我根本不需要测试。只有一些细节需要解决。这是这一想法的本质决定的。

微型杀手的进攻

史密斯追求的目标之一就是设计真正的通用大型并行式计算机。与计算机科学界那些单纯用于研究的设备不同，他的目标依赖于市场的起伏变化。

到本书完成时为止，这一进展仍然非常艰难。功能强大的微型计算机，劳伦斯利福摩尔国家实验室[①]的尤金·布鲁克斯（Eugene Brooks）所称的“微型杀手”，正让并行计算机显得越来越昂贵。然而，史密斯仍然保持乐观，他相信并行计算机将会像今天的个人计算机一样普及开来。[②]

> 从现在（1998年）开始，10年后我们的办公桌上就会出现并行式计算机。15年后我们就可以随身携带这种机器——到时候它会更加轻巧。人们可以随时随地访问并行式计算机，不管它们放在何处。也许我们会将“通用”重新定义为并行。也许所有需要快速运行的设备都将成为并行式。

在工作之外，并行同样是史密斯的兴趣所在。他是一名经验丰富的业余男中音，在合唱团演唱，对帕莱斯特里纳[③]、汤姆金斯[④]和穆迪[⑤]等16世纪文艺复兴时期作曲家创作的复调音乐有着特殊的兴趣。

> 这种音乐有些晦涩，但的确是我所知的最值得做的音乐形式。演唱无伴奏的复调音乐非常困难——许多旋律都同时进行。没有其他音乐与之类似。

史密斯身材魁梧，就像一个退役的橄榄球运动员。他就像四分卫一样带领着Tera团队的40个成员——指挥运筹，同时又放手让每一个成员自己寻找解决问题的方法。

> 在涉及如此多人的设计中，你无法通晓所有事情的状态。但你必须明确自己想要什么。最重要的一点可能是：用自己的想象力来考虑事情，然后让大家来共同完成。

---

① 劳伦斯利福摩尔国家实验室（Lawrence Livermore National Laboratory）是美国能源部所属的国家研究机构，与洛斯阿拉莫斯国家实验室同属于美国为设计核武器而建立的部门。

② 史密斯的信仰已经在一个领域中获得实现：在线数据库交易处理（用于航班预订系统、银行以及远程通信）。Teradata和Tandem两家供应商已经为这些应用制造了并行计算机，并且获得了极为可观的利润。到了20世纪90年代中期，即使是克雷研究所也开始制造用于数据库处理的计算机。——原书注

③ 乔瓦尼·帕莱斯特里纳（Giovanni Pierluigi da Palestrina，1525—1594），意大利作曲家，被广泛认为是文艺复兴时期最杰出的作曲家之一。

④ 托马斯·汤姆金斯（Thomas Tomkins，1572—1656）是英国作曲家，擅长编写情歌、风琴音乐、圣歌和礼仪音乐，也是英国维金纳琴学校的最后一名成员。

⑤ 约翰·穆迪（John Mundy，约1554—1630）是英国作曲家，文艺复兴时期的维金纳琴和风琴音乐家。他的父亲威廉·穆迪（William Mundy，约1529—1591）也是一名杰出的作曲家。

12

W.丹尼尔·希利斯与生物学的连结

显而易见，大脑的组织原理是并行的，而且是大规模并行。信息就在许多非常简单的并行单元的连结中间传递。如果我们按照这种组织系统来制造一台计算机，就很可能完成大脑所做的事情。

——W.丹尼尔·希利斯

很难想象两种科学文化会像计算机科学和生物学这样水火不容。计算机科学源自数学、物理学和工程学，运用简化的基本原则，产生了最优秀的设计和最敏锐的洞察。现代生物学则产生于对自然的观察和改造，其复杂性时常让人惊叹，试图对其进行简化的努力往往徒劳无功。在日常工作中，计算机科学家在恒温盒体中对硅和金属发出指令，而生物学家则在培养皿中为生物细胞提供营养。计算机科学家认为偏差就是错误，而生物学家则认为偏差正是大自然的奇迹所在。

W.丹尼尔·希利斯对这两门学科都进行了深入的研究，而且经常从一门学科中获得借鉴，促进另一门学科的发展。他设想并制造了以人类大脑为模型的计算机，拥有成千上万个处理器，在其他人并不看好的情况下依然勇于前进。他后来的设计需要16 000个以上的处理器，并广泛应用于地震勘测、医疗诊断等多个方面。希利斯将他的“连结机器”计算机建立于生物学理论之上，并且提出了只有最大胆的生物学家才敢设想的进化机制。

在边远地带

希利斯1956年生于马里兰州巴尔的摩。他的父亲是美国空军的一名传染病专家，因此一家人经常会沿着肝炎爆发的地区四处迁徙。

我小时候去过很多地方。我在中非许多不同的国家生活过——卢旺达、布隆迪、扎伊尔还有肯尼亚。我们家总是在丛林中出没，所以只能在家念书。

希利斯的母亲承担了大部分教育工作，而且特别喜欢教他数学。而他的父亲则对他在生物学

方面的爱好鼓励有加。

> 当实验室丢弃试验设备时，我就去找一些玻璃器皿拿回我自己的实验室。我做的最好的一次生物实验是对一颗青蛙心脏进行组织培养，让它在试管里继续生长跳动。这些同质化的细胞居然能够结合在一起协调运动，让我感到非常惊奇。
>
> 我想我童年时受到的是生物学的熏陶。但其实我天生对工程学感兴趣。还是个孩子时我就经常制造东西。

希利斯小时候喜欢玩模型和积木玩具，但很快就开始研究引擎和机器人。他对一本名叫《迈克·马力甘和他的蒸汽挖土机》[①]的书特别感兴趣，书中讲述了一辆小蒸汽挖土机建造政府大楼的故事。

> 我对他们将蒸汽挖土机改造为大楼底部供暖器的画面印象非常深刻。
>
> 后来我也知道了计算机。我有本书叫做《神奇世界解密：机器人》[②]，里面讲的其实就是计算机。书里有很多图片，现在我知道大部分都是远程操控器。我也有个玩具机器人，在它的脑袋里面坐着一个小人，肯定是在控制它。我记得自己曾对脑袋里有个小人的概念着了迷。

在9岁时，希利斯用一个电唱机和两块圆盘制作了他的第一台“计算机”。一块圆盘上有一些问题，另一块则写满了各种可能的答案。电唱机旋转时，一个连接器会挂住两块圆盘，并将问题和答案连成一行。

20世纪60年代末，希利斯一家搬到了加尔各答，他第一次开始对数字计算机的基本理论有所认识。

> 当我在印度时，身边根本没有技术可言。甚至连英文版的技术书都很难找到。英国领事馆有一个图书馆，里面有一本乔治·布尔写的《思维规律的研究》[③]，书中他提出了如今人们熟知的布尔代数。我非常喜欢这个书名。

这本书对我来说过于高深，但我理解了与、或、非这些基本的思想。

对于初出茅庐的计算机科学家来说，这本1854年出版的书很适合阅读。布尔在书中描述了一种代数，可以将逻辑命题转化为某种算术运算（参见补充内容“布尔代数101”）。

---

① 《迈克·马力甘和他的蒸汽挖土机》（*Mike Mulligan and His Steam Shovel*）曾入选美国纽约公共图书馆“每个人都应该知道的100种图画书”，首版于1939年，全球销量累计已超过2千万册。它的作者维吉尼亚·李·伯顿（Virginia Lee Burton，1909—1968）是美国儿童绘本作家，最著名的作品包括《小房子》（*The Little House*），1943年获得凯迪克金奖。

② 《神奇世界解密》（*How and Why Wonder Books*）系列图书是美国20世纪60年代和70年代出版的、面向青少年的系列科普绘本。

③ 本书全名为《思维规律的研究，作为逻辑与概率的数学理论的基础》（*An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities*），是乔治·布尔最著名的著作。

布尔代数101

假设有如下三个命题，每个都可能是真或假。

命题R："天正在下雨。"

命题T："今天星期二。"

命题U："我带了伞。"

假设有人给出了由命题R、T和U复合产生的句子：

"天没有下雨或我没带伞，且要么我带了伞且今天星期二，要么天正在下雨且今天不是星期二，要么我没带伞且天正在下雨。"

如果我们知道今天星期二、正在下雨而且我没有带伞，那么很快就能判断这个句子是真还是假，但是过程比较绕。

布尔的方法是将真值指定为1、假值指定为0，然后找到与逻辑关系与、或、非相对应的运算。乘法对应于"与"，具有其常规意义，但限于1和0。加法对应于"或"，也具有其常规意义，但$1+1=1$。减法符号对应于"非"，对1和0进行相互转换，因此$-1=0$、$-0=1$。

通过这种定义，我们就能把上面的句子转化为非常简单的公式$(-R+-U)\times((U\times T)+(R\times -T)+(-U\times R))$。因为我们知道今天星期二、正在下雨而且我没有带伞，所以$T=1$、$R=1$、而$U=0$。

代入这些值，那么$(-R+-U)$就等于$(0+1)=1$。$(U\times T)+(R\times -T)+(-U\times R)$就等于$(0\times 1)+(1\times 0)+(1\times 1)=1$。所以整个句子对应于$1\times 1=1$，为真。

由于计算机的基础电路执行的正是与、或、非运算，布尔代数就成了最合适的知识工具，用来设计加法器、位移器和解码器这些机器的内脏。当然，前提是我们能够得到这些部件。

> 一开始我想的是用身边的用品来制造一台计算机——可我身边可以沾得上边的只有手电筒。所以我能弄到电线、灯泡和手电筒电池，但我弄不到开关。不过我用纱门和钉子自己做了一个开关。所以我就靠把钉子钉到一个纱门上或另一个纱门上来扳动开关。钉子上缠着电线。所以我想你也可以用这些玩意造一台计算机——用纱门造一台井字游戏机出来。

之后一家人离开印度搬到了马里兰州的陶森，丹尼尔终于可以进入正规初中就读，他的母亲也得以攻读生物统计学的博士。学校生物学的标准实验室练习无法满足希利斯，他培养了一些在健身房附近发现的细菌，结果发现它们对人体有害。校方拒绝相信这个年轻人的发现，幸运的是没人被感染。到了高中时，希利斯开始寻求科学和工程学的挑战。约翰霍普金斯大学化学系丢失了一台计算机的使用手册，于是请希利斯把这台计算机连接到一台质谱仪上。他为其编写了程序，能够计算实验结果并求平均数。这似乎比用纱门和钉子要简单得多。

1974年希利斯进入了麻省理工学院。他决定研究大脑的工作原理，于是计划主修神经生理学。但未能持续很长时间。

我到麻省理工的第一个晚上就遇到了杰罗姆·莱特文[1]。他正在写一篇名为《蛙眼告诉蛙脑什么》的论文——我对这个想法非常感兴趣。他问我打算学什么，我回答说我准备学神经生理学。他说："我猜你连这方面有什么好论文都说不上来。"

莱特文对我说，如果我真的对智慧产生的原理感兴趣，就应该去研究神经元。而且他还说服我去找马文·明斯基。

我当然听说过人工智能实验室和明斯基的大名。在当时明斯基可不是一个轻易能接触到的人物，所以我只能经常去他们的实验室四处闲逛，等待机会。

明斯基和约翰·麦卡锡在1958年创办了麻省理工人工智能实验室，至今仍然是世界领先的人工智能实验室之一。希利斯认为，要想打动明斯基，最好的办法就是帮助实验室完成赞助机构规定的项目。当时实验室有部分资助来自于美国国家科学基金会的一个计算机教育项目，希利斯找到了项目提案，想看看自己还能为实验室做些什么。

提案中赞助方提到了一件他们希望研究但尚未完成的事，那就是制造出给还不会读书写字的孩子们使用的计算机终端。

所以我坐下来想了一会儿，找到了一种方法然后走进了实验室。我有把握能凭此得到一份工作。果然我如愿以偿，开始在实验室和LOGO团队一起工作。

拉迪亚·珀尔曼[2]和我一起开发了一种可以四处移动图标的终端。那还是在图形用户界面出现之前，所以它看起来很棒。

LOGO主要由西蒙·派珀特博士[3]发明。派珀特曾希望把计算机用于教育，因此LOGO是一种非常简单的编程语言——简单得足以让小孩制作他们自己的卡通片（这个项目早期还给了艾伦·凯设计Dynabook的灵感）。希利斯还是很难见到明斯基，但他发现明斯基当时正在地下室造一台计算机。

于是希利斯去了地下室，了解了明斯基的计划，并对他的计算机进行了一些修改。明斯基很欢迎这位帮手，把他视作自己的弟子，甚至还把自己家的地下室借给希利斯居住。每天开车上下班的途中两人有充分的时间交流。

明斯基最大的贡献在于，他指出大脑实际上非常复杂，有多个目标在相互影响，冲突和妥协。因此从某种意义上来说，明斯基所做的其实比弗洛伊德更进了一步。

弗洛伊德指出在我们心中可能会同时思考两到三件事。明斯基则提出对于思维的崭新看法，即在我们心中会同时思考上千件事。

---

① 杰罗姆·莱特文（Jerome Lettvin，1920—2011）是美国认知科学家，麻省理工学院的生物工程系名誉教授。1959年他发表了著名论文《蛙眼告诉蛙脑什么》（*What the Frog's Eye Tells the Frog's Brain*），是科学文献索引中被引用最多的文章之一。

② 拉迪亚·珀尔曼（Radia Perlman，1951—）是一名软件设计师和网络工程师，她有时也被人称为"互联网之母"。她最著名的发明是生成树协议（Spanning Tree Protocol，又称扩展树协议），用于确保一个无循环的局域网络环境。

③ 西蒙·派珀特和LOGO参见本书第3章注。

成千上万个相互影响的分散主体居然能产生外在统一的思维，这种想法初看起来似乎无法想象，但它遵循了一种进化的原理，即“倏忽进化”（emergence）。

在有机体中，基因突变很少会造成物理特性的改变。突变导致的物理变化通常都是有害的。然而，各种物种正是由于突变和自然选择才得以进化，变得更加美丽、更加适应环境。

倏忽进化的原理是，相互影响的个体会通过某种选择过程来适应生存机制。

这一概念吸引了希利斯。他猜测大脑中也存在相似的过程，于是决定设计一台计算机来检验这种猜测，他称之为“连结机器”（Connection Machine）。他的想法很简单：利用数千个小处理器（最初的设计需要1百万个），每一个都是自带存储器的计算机，全部连接并编程使其交互，看是否有类似大脑那样的智能出现。

> 连结机器的产生是因为我意识到了以下矛盾：人脑的运行比最快的计算机还要快，但其实人脑的转换需要好几微秒的时间，而计算机只需要几纳秒。
>
> 所以很明显，大脑的组织原理是并行的，而且是大规模并行。信息就在许多非常简单的并行单元的连结中间传递。如果我们按照这种组织系统来制造一台计算机，就很可能完成大脑所做的事情。连结机器的设计动机就这么简单。

无论简单与否，大规模并行却存在着争议。著名的单处理器计算机设计师吉恩·阿姆达尔（布鲁克斯在IBM的同事，后来创建了以自己名字命名的公司与IBM竞争）就已经发现，并行在很多情况下有着其固有的限制。

我们可以通过一个类比来理解他的观点。假设有一家公司希望雇用足够多的工人，目的是在一小时内生产10 000个配件。工人每次只能生产1个配件，需要1个小时才能完成。在生产之前，工人需要从库房管理员那里领取必要的零件。管理员只需要36秒就能为一个工人配备好零件。

公司观察到，10个工人每小时能生产10个配件，50个工人每小时能生产50个配件。于是公司雇用了10 000个工人，结果却发现一小时只生产了区区100个零件。原因是无论库房管理员怎样努力，他在一小时（3600秒）内最多也只能给100个人提供零件。

阿姆达尔法则对这一现象进行了总结：如果一个并行任务包含有需要顺序进行的部分（本例中即为从管理员处领取零件），该部分需要的时间比例为f（本例中为1/100），那么即使是大规模并行，也不可能让生产（或计算）的速度快过$1/f$（本例中为100）。

阿姆达尔法则使得20世纪80年代初与希利斯同时代的人们不得不相信，大规模并行是行不通的。但是希利斯对大脑的理解让他仍旧对此保持乐观。

> 有很多类似阿姆达尔法则的证明都认为，大规模并行计算机是不可能实现的。虽然我并未发现这些证明中有什么错误，但我知道大脑确实是按这种方式进行组织的，而且大脑确实能有效工作。
>
> 所以，不管这种制造方式在大范围上是否有价值，但对于让计算机进行常识性思考等活动来说，它一定是有价值的——例如图像识别、记忆提取、推理等大脑能做的所有事情。这就是我们制造连结机器背后的动机。

PDG

1983年，连结机器的模拟实验获得了正面的结果。受此鼓舞，时年27岁的希利斯（在明斯基的建议和鼓励下）于麻省剑桥创建了思维机器公司（Thinking Machines）。资金来自于公私两个渠道——哥伦比亚广播公司总裁威廉·佩利是主要投资人，而国防高级研究规划署（DARPA）则同意购买第一台设备。

在公司启动前几个月，希利斯和诺贝尔奖得主、物理学家理查德·费曼讨论了自己的想法。费曼评价这个项目很“愚蠢”，但随即又同意从夏季开始为公司工作。加入后费曼要求“做一些真正的事情”，于是协助分析了第一台设备的内连网络，确定了网络中每个交换器所需要的短时存储器数量。除了为思维机器公司所做的实际贡献之外，费曼还给了希利斯许多精神上的支持。

> 我成长的环境中有一种学术偏见，即科学比工程学要好。费曼让我相信事实并非如此，创造各种东西同样有价值。

费曼还根据自己在洛斯阿拉莫斯的曼哈顿计划中积累的经验，提出了如何组织团队力量的建议。他建议公司在软件、组装等制造的关键领域挑选专家，让他们当子团队的负责人。（比尔·盖茨在微软也遵循了类似的哲学，他坚持管理人员应当比手下的程序员具备更多专业知识。而大多数高科技公司很少做到这一点。）

费曼的另一项创举是鼓励科学家为连结机器提供新的应用。这促成了思维机器公司早期和神经网络先驱、加州理工的约翰·霍普费尔德[①]的一系列合作。

正如霍普费尔德在1982年所定义的，神经网络是一系列“神经元”的集合，每一个神经元都可由一个计算机程序模拟。神经元程序接收来自于其他神经元或外界刺激的输入，然后执行计算，再输出给其他神经元。这就是大脑中神经元的大致运作方式。

霍普费尔德指出，利用一系列有已知正确响应的刺激，便可以对神经网络进行“训练”。网络接收这些刺激并计算其响应。如果响应正确，那么不作任何改变，否则网络就调整一个或多个神经元的计算，使其更加接近正确的响应。

霍普费尔德还指出网络可以识别图像的模式。在此之后出现了更多的应用，包括股票市场分析和外汇交易。甚至好莱坞都知道了神经网络：影片《星际迷航》（*Star Trek*）里的角色Data就是一个神经网络实体化后的机械人。

霍普费尔德的方法立刻引起了连结机器设计师们的共鸣。每个模拟神经元可以映射为一个处理器，而神经元中传递的信号可以在计算机网络中进行传输。这正是这种机器的一个完美应用。

不久之后，永不满足的费曼又将注意力转向了量子色动力学计算。这是量子物理中的一个分支领域，将夸克、胶子和质子、电子联系起来。运用这种理论进行的计算是一项庞大的工作，涉及巨大的矩阵（数字表格）。费曼证明连结机器比加州理工专门为此目的建造的计算机还要快。

---

① 约翰·霍普费尔德（John Hopfield，1933—）1978年受聘于贝尔实验室，专职以物理学观点观察生物体内的现象，用逻辑计算模拟生物神经元的运作。他于1982年提出了“神经网络”（neural network）概念，如今被称为霍普费尔德网络。

第一台连结机器于1985年开始运行，从那以后程序员和科学家们将其用到了数据库搜索、地球物理建模、蛋白质折叠、气候模拟甚至保险单处理等各个方面。

例如，石油开采业利用连结机器来进行声波反射分析。在可能有石油的区域进行地面引爆，会产生声波冲击，而地下的反射将产生上亿个数字。连结机器接收这些数字并生成一幅精确的地质构造图像。石油公司根据地质构造分析的结果，只在有开采前途的地方进行挖掘。毫无疑问，大规模并行处理是非常有用的。

> 并不是说只要拿一堆慢速处理器同时运行，就什么问题都能解决。但事实是大规模并行不能解决的问题确实很难找到。这是我们在更大范围内应用连结机器时所发现的事实之一。很难找到这样的问题：它有大量数据，而并行的可能性却并不随着数据量的增多而成比例增长。只有那些数据量少且固定的问题才不能在并行计算机上良好运行。
>
> 比如说，如果要模拟太阳系九大行星的运动，你用 18 个数字就能体现。如果你非要用一大堆并行来解决区区 9 个（哪怕是 100 个）行星的问题，这反而很难。我不是说这不可能——我只是说很难（这也表示这个问题值得研究）。

受到大脑固有的并行机制的启发，希利斯永远地改变了计算机技术。到了20世纪90年代，已有20余个大规模并行项目进入了商业化的不同阶段。为之付出过努力的科学家们，包括Tera机的首席设计师伯顿·史密斯，都以极大的兴趣关注着连结机器的进展。

希利斯从思维机器公司的商业化发展中得出了几个结论。首先，使用相对较少的快速处理器要比使用一百万个慢速处理器更好。否则用于将处理器连结起来的成本就会过于昂贵。连结机器的第5代模型CM5最多只包含16 000个微处理器，但每个微处理器每秒可执行1.2亿次运算，总体处理速度每秒可超过2万亿次。

其次，使用市场上已有的高容量微处理器要比费力制造专用的微处理器（就像伯顿·史密斯所做的）更好。希利斯选择了符合业界标准的Sparc芯片，但连结机器的设计可以随着处理器的更新换代而进行改动。

第三，网络仍然极为重要，因为许多指令必须访问远程的存储器芯片。这一观点让伯顿·史密斯放弃了本地存储器，并且让每个处理器同时进行成百上千个任务。与之不同的是，希利斯在最新一代连结机器的网络中，使用了麻省理工学院的查尔斯·莱塞森教授[①]发明的“肥胖树”（fat tree）。我们可以用一个类比来说明莱塞森这一发明的高超之处。

正常的通信树就好比公司里常见的、有着严格汇报关系的层级结构。在同一个部门内，成员间的通信（在理想情况下）总是方便而快捷的。然而从一个部门到另一个部门间的通信却可能会有问题。它往往需要沿着共同上级的管理线进行上传下递。如果部门间的通信量很大，那么每一个上层管理人员都需要传递大量的信息，导致部门间通信极为缓慢。

肥胖树类似于在每一个管理层级都设立一个同等且自主的管理团队，层级越高团队人数就越

① 查尔斯·莱瑟森（Charles Leiserson，1953—）专门从事并行计算和分布式计算等领域，基于C语言开发了专为并行运算而设计的泛用式编程语言Cilk，他是基础算法领域的权威著作《算法导论》的作者之一。他发明的“肥胖树”内连网络通用于各种硬件，被用于多种超级计算机。

多。这样能够避免传统设计中的通信瓶颈问题，因为管理团队中的任何成员都能将信息传递给上一层或下一层的所有人（参见图12-1）。最新的连结机器就运用了肥胖树。

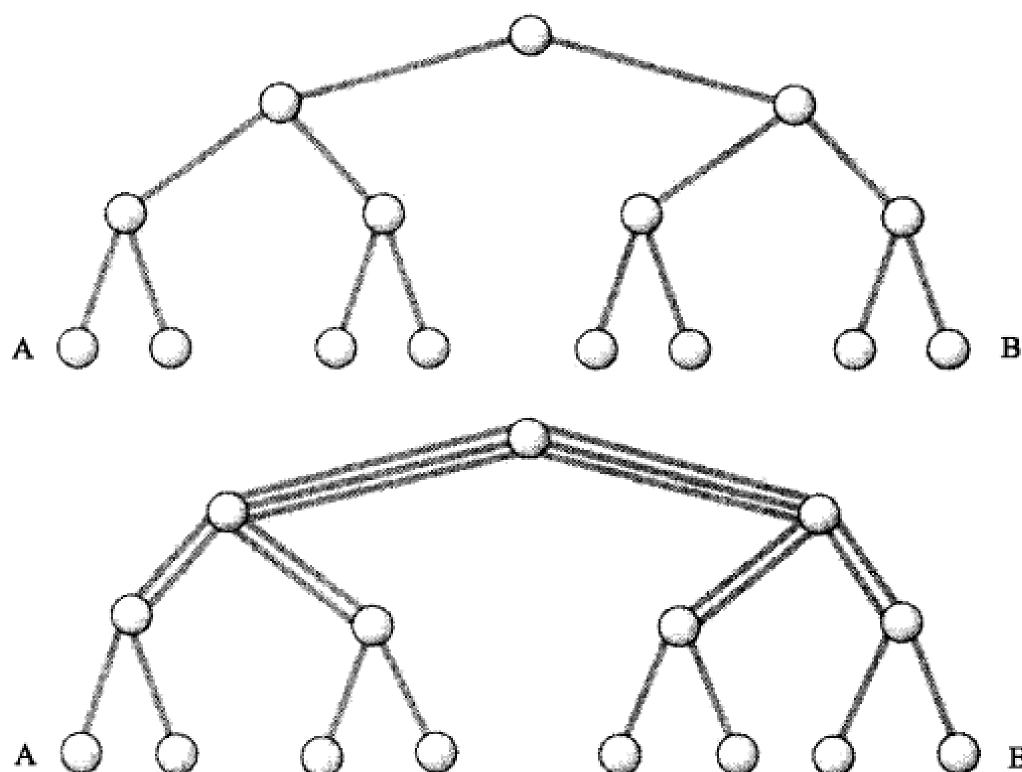


图12-1　公司汇报关系层级树的两种模型。在上方的标准模型中，从A部门到B部门的所有指示都必须上传至最高层领导，然后再下达到最底层。在下方的肥胖树模型中，从A部门到B部门则可以有多种路径

硅片中的进化：创造人工生命

生物学启发希利斯建立了成功的计算机设计哲学，如今他用手中的工作证明，计算机科学同样也能有所回报。当然，生物学家早已开始利用计算机进行文字处理、数据存储、统计分析和模式识别等工作，但他们主要的工作仍然是进行病毒、细胞和高级有机体的试验，并从这些试验中得出各种模型。

物理学家（从某种程度上来说，还有化学家）却采取几乎完全相反的方法。他们往往先从抽象的模型开始，然后再通过实验对其进行检验。而且，物理学家即使在不能得到完美实验证明的情况下，也更倾向于相信优美、抽象的理论。

例如，诺贝尔奖得主、物理学家史蒂文·温伯格[①]曾在他的《终极理论之梦》[②]一书中评论道，用于检验爱因斯坦广义相对论的第一次实验与该理论的预测结果并不完全吻合。而物理学界对此

① 史蒂文·温伯格（Steven Weinberg，1933—）和谢尔登·格拉肖（Sheldon Glashow，1932—）、阿卜杜勒·萨拉姆（Abdus Salam，1926—1996）于1979年携手获得诺贝尔物理学奖，他们在基本粒子间弱相互作用和电磁相互作用的统一理论以及对弱中性流的预言等方面做出了卓越的贡献。

② 《终极理论之梦》全名《终极理论之梦：对自然基础法则的探究》（*Dreams of a Final Theory: The Search for the Fundamental Laws of Nature*），于1993年首次发行。

的反应是断定试验设备存在缺陷！（事实上物理学界的看法是正确的，但是他们对该理论的信念主要基于美学观点，而非根据实际观测。）而据希利斯所称，生物学家绝不会这样。

> 生物学家相信，简单的数学理论通常都是错误的，因为生物系统成因众多，彼此也没有清晰的界限——基本上可以说杂乱无章。生物系统确实也有美存在，但这是一种复杂、丰富之美，而不是物理学那种简约、优雅之美。

如果生物学家是这样一群怀疑论者，为什么他们会凭借信心，从低等动物推演至高等动物呢？进化论最好的依据是，所有已知生命都拥有共同的祖先，这一点已有化石记录为证。与之相反，数学或计算模型所构筑的生命形式却没有任何实际的依据。人们永远不会知道某个模型中隐含的假设是否影响了对预期现象的分析。

希利斯在20世纪80年代深信大脑的组织结构可能会有助于解决困难的计算机科学问题，如今他同样深信，有着强大的计算机支持的简单数学模型也可能会带来新的生物学洞见。他引用了经典的、由贝尔实验室和麻省理工学院在20世纪30年代所提出的反馈控制论思想，以此说明数学模型也能够增强我们对生物学的认知。反馈（feedback）是一种利用输出中的错误来调整输入的思想。在淋浴时感到水温过热，反馈就会促使我们找到水龙头，把水的温度调低。

> 大脑并不是由运算放大器构成的。但如果能理解放大器电路中的反馈原理，对于理解某些生物系统的运作方式会是很有帮助的。反馈已经成为生物学中内在化的东西，而且已经根植于我们对生物体运作的认识当中。我们并不认为反馈出自控制论。

在基于计算机的进化研究中，希利斯较之传统的生物学家有两方面的优势。第一，自然界化石记录的不完整性导致难于研究自然进化的细节。第二，在实验室中模拟的进化，即使是最喜滥交的果蝇，也很难得到数百代以上的结果。而在希利斯的连结机器上，可以轻松地模拟100 000代进化结果，而且能获得完整的“化石”记录。

希利斯所用的基本技术是构建一系列仿真生物体，每个生物体都封装一个简单的程序。然后选择某种适应能力标准，例如程序将数从小到大快速排列的能力。对于能够“适应”的程序会给予奖励，让它们有更高的几率繁殖下一代，尽管也可能会出现突变。

希利斯早期的一个模型显示，对于某些适应能力标准来说，有性生殖（两个“父母”程序混合）要优于无性生殖，但其他标准则没有这一现象。随后他又引入了带有一系列未排序数的“寄生虫”，如果程序不能很快将这些数排序，就会被寄生虫“消灭”。

寄生虫也会进化——它们的适应能力由生成复杂数列的能力所决定。令人惊奇的是，存活下来的排序程序进化得更加迅速了。

到20世纪90年代末期，在这场排序竞赛中，人类依旧占据着微弱的上风。即使是希利斯进化得最好的排序网络，也仍然比研究人员米尔顿·格林（Milton Green）所发现的人类最好的网络要慢2%。

牛津大学的生物学家威廉·汉密尔顿[1]早已指出，寄生虫可能有助于进化。希利斯的模型似乎证实了这一猜测。喜欢批评的生物学家可能会说希利斯所做的其实毫无新意，只是设计了一个人工的实验而已。但希利斯模型的优势在于，它以非黑即白的程序代码形式揭示了这一假设。希利斯的代码显示，只要保证三个前提，加速进化就会产生：寄生虫、突变能力以及与某个目标（例如快速排序）紧密相连的适应能力标准。那么在真实世界中，这些前提是否成立呢？希利斯坦承他也没有答案。

> 我也不确定进化能解决问题这一看法是否正确。进化并不是解决现有问题。进化是发明一个问题，同时解决它。

希利斯只是"人工生命"——研究计算机生成的生物体的繁殖和进化——的一位倡导者。他还有和他志趣相投的同事们（主要是在圣达菲学院）受到了生物学家的冷眼相待。正如《自然》杂志主编约翰·马杜克斯[2]所指出的："你不能指望那些经年累月培养生物的人会给只在计算机上花几个小时做点什么的人太多尊敬。"有些生物学家甚至斥责人工生命根本就和生物学毫不相干。希利斯礼貌地进行了反驳。

> 我曾问过（诺贝尔奖得主，生物学家）戴维·巴尔的摩[3]，为什么他认为这种东西不是生命。他对我说："生命必须进化、繁殖、新陈代谢而且由碳构成。"所以根据他的定义，在计算机上运行的任何东西都不可能成为生命。
>
> 现在我更感兴趣的是它们能繁殖和进化的事实，尽管它们不是由碳分子构成的。我们之前并不需要区分这两种事物，因为人们从未发现过不是由碳分子构成、却又能繁殖和进化的东西。我并不在乎最后是否能用"生命"这个词来形容这种新事物。不管它们应该怎样称呼，我所感兴趣的是，它们与碳基生命体存在一些共性，但又在某些方面不同。它本身并不能证明任何有关碳基生命体的问题，但仍然具有一定的指导意义。

希利斯对连结机器的研究结合了他对生物学和计算机科学这两方面的热爱。在工作之外，他是一名业余的地质学家，喜欢在古老的矿井里勘探。他甚至在新墨西哥买了一块地，在思维机器公司由于资金问题而放弃硬件业务后[4]，曾打算去那里专门从事勘探工作。他还设计玩具——在大学时他就曾为米尔顿·布拉德利公司[5]设计过玩具，他的办公室里堆满了自己的作品。

---

① 威廉·汉密尔顿（William Hamilton，1936—2000）是英国进化生物学家，被视为20世纪最伟大的进化理论家之一。

② 约翰·马杜克斯（John Maddox，1925—2009）两度担任《自然》（*Nature*）杂志主编，达22年之久。《自然》杂志创刊于1869年，是世界上被引用最多的跨学科科学刊物。

③ 戴维·巴尔的摩（David Baltimore，1938—）是加州理工学院生物学教授，并曾在1997年到2006年期间担任该校校长。1975年诺贝尔生理学或医学奖获得者之一。

④ 思维机器公司于1994年8月申请破产，硬件部门被太阳微系统公司收购，之后继续作为一家纯数据挖掘技术公司存留，于1999年被甲骨文公司收购。

⑤ 米尔顿·布拉德利公司（Milton Bradley）创建于1860年，是美国最大的玩具厂家之一。

明斯基和费曼都教过我如何思考，而他们思考的方式大相径庭。但他们共有的一点是乐于质疑一切。在我们的现实生活中有许多看似毫无疑问的假设，而他们两人都善于把这些假设挖掘出来。而且明斯基和费曼都以自己的工作为乐。我想这一点很重要，而且我自己也是如此。

希利斯还有一个异想天开的构思，他希望建造一座巨钟，像埃及金字塔那样宏伟。这座钟每一年动一次指针，每一百年敲响一次，每一千年钟里的布谷鸟才出来报时一次。

这座钟可能会让人们畅想，在布谷鸟报了几次时之后，世界又会变成什么样子呢？

第四部分

机器智能的雕塑大师：如何让机器更聪明

本书第四部分讨论的是计算机科学中那些最具创新精神（有时甚至乐观过度）的研究者所从事的领域。他们的梦想是让计算机变得和人类一样聪明，甚至更胜一筹。人工智能（Artificial Intelligence，简称AI）领域的研究人员希望计算机能以人类的方式思考和行动：它们应当能识别图像、积累经验并学习，而且能像我们常做的那样进行违反逻辑却又符合人之常情的推理。这些创举所产生的产品可以是打扫房间或修理太空舱的机器人，也可以是进行医疗诊断的程序。值得注意的是，对人类来说最艰巨的任务（例如专业的医疗诊断）计算机却能轻松掌握，而打扫房间这一挑战却比太空行走要艰巨得多。决定哪些物体应该放进字纸篓要求机器人把物体从背景中区别开来，识别它们，并且运用常识来判断如何处理。这些看似“简单”的技能，当前的技术水平依然很难实现。

事实上，如果我们以人类智能的标准来衡量现代计算机的能力，结论只能是难分伯仲：计算机更擅长运算；有超强的记忆力，能更好地组织各种琐碎细节；在棋盘上击败绝大多数人类；精通晦涩难懂的专业知识；能绘制出精确的图像。但是它们却不能理解语言、不能识别面部，也不能进行5岁小孩就能做的日常推理。我们将要讨论的正是那些希望计算机在此类失败之处有所建树的人。

约翰·麦卡锡（由于他在编程语言方面的重要性，本书在第一部分已经介绍了他的生平）在1958年发明了编程语言Lisp之后，便开始了从根本上去追求向计算机灌输常识。这一目标似乎很简单，因为正如他所说，常识“能被任何非低能的人所掌握”。然而他的每一次努力都会揭示出新的问题。首先，常识需要基于经验的猜测。我们对这些猜测有多少信任呢？这些猜测有可能其实是错误的。任何一个外界事件都可能迫使你进行修正。而对于计算机而言，确定哪些事件相关是一个极为困难的问题，尽管对人类来说易如反掌。麦卡锡大胆地提出了一种逻辑框架来解决这类问题，但这个逻辑框架直到20世纪90年代末才开始受到重视和检验。

爱德华·A.费根鲍姆则采取了一种更为务实的方法。他和斯坦福大学的同仁一起，基于医学和光谱学的庞大知识体系，构造出了第一批能给出专家级建议的程序（Mycin和Dendral）。这些程序展示了如何构造专家系统，并且开创了一个新的行业，应用范围从航班飞行计划管制到贷款

决策，不一而足。

道格拉斯·莱纳特以一款名为“自动数学家”（Automated Mathematician，后来被简称为AM）的程序开始了他的计算机科学生涯。AM最初就包含了数学集合的概念，能够计算集合的元素个数，并且能执行交集、并集和差集等操作，随后能够导出加法、乘法、素数和指数。它甚至重新发现了哥德巴赫猜想：所有大于3的偶数都是两个素数之和。尽管这款程序成功地重建了算术和初等数论的基础，但是莱纳特发现它在努力进行更多探索时会显得毫无目的。他认识到，如果想让计算机发现新的事实或思想，就必须先让它掌握相关的事实和概念。他花了10年时间尝试将当代北美洲的所有知识编码进一个叫做Cyc的计算机程序中。Cyc之名取自“encyclopedia”（百科全书）一词，这是迄今为止规模最大的AI试验。它可能会产生出极端智能的主体，也可能会落得个惨败的结局。

当我们与大多数计算机科学家谈起人工智能领域时，经常会得到“这玩意没戏”的轻蔑回答。真的就绝对没戏吗？毕竟，这个世界上有超过70亿人在这方面都干得不赖。

13

爱德华·A.费根鲍姆
知识的力量

要构建一个基于知识库的系统，有三件事非常重要：知识，知识，知识。要衡量一个系统的能力，主要得看它知道多少，而不是它的推理能力如何。

——爱德华·A.费根鲍姆

专家系统早已成为我们日常生活中不可分割的一部分。它们帮助我们操作股票、设计车辆、安排航班路线，甚至策划战争。许多研究人员和从业者都从事专家系统方面的工作，但有一位科学家却主导了该领域的发展方向，影响远超其他人。

1936年，爱德华·费根鲍姆出生于新泽西的威霍肯。他的父亲是波兰移民，但在他周岁之前就去世了，所以爱德华对他没有任何印象。他的母亲后来又和一个面包店的会计兼办公室经理结了婚，正是这位继父激发了爱德华对科学的兴趣。

纽约市的海登天文馆每个月更换一次展品，我的继父每次都带我去看。看完之后，我们就去自然历史博物馆参观一小部分。这样，几个月之后，我们看完了所有展厅。这就是我对科学事物的最初记忆。

费根鲍姆如饥似渴地阅读，他在学校功课很好，科学方面表现尤其优异。他对继父在面包店里用的笨重的机电式门罗计算器[①]非常着迷。

这种台面式计算器重约 20 磅，里面有很多轮子和马达，还有一个很大的键盘。你得用力敲击数字、按下写着“加”和“乘”的巨大按钮。之后轮子就会开始转动，发出叮叮当当的声音。

我对门罗计算器能做的事情感到好奇——它可以进行各种复杂的计算，如果手算则非常费力。有一次我想向朋友们炫耀它的功能，就带着它上了高中的校车。这玩意太重了，搬动它需要花不少力气。而他们对它却根本就不感兴趣，所以我又把它搬回了家。

① 门罗计算器（Monroe Calculator）由门罗计算器公司生产。该公司由杰伊·门罗（Jay R. Monroe）于1912年创立。

费根鲍姆对科学事业极为憧憬，但在父母的敦促下他还是选择了机电工程专业。

> 我年少时深知生活的艰难，得有足够的钱维持日常生活。工程学看起来比科学更为令人满意——它更实用，挣的钱也更多。
>
> 机电工程处在科学和数学的交界处。它涉及的知识相对更加抽象，而机械工程或民用工程则更多面向具体对象。我从来都不是一个关注“具体对象”的人。我是一个关注“思想”的人。

卡内基的岁月

费根鲍姆1952年进入卡内基理工学院（如今的卡内基梅隆大学）。在当时，机电工程师主要还是和发电机、无线电及早期电视打交道，本科生是不会接触到计算机的。一位教授鼓励费根鲍姆把眼光放长远一些，于是他开始选修机电工程之外的课程，甚至跑到了学校新成立的工业管理研究生院听课。那里的一位教授詹姆斯·马奇[①]向费根鲍姆介绍了匈牙利数学家约翰·冯·诺依曼在10年前提出的博弈论思想。

> 博弈论很有魅力——对我来说绝对有魅力。人们可以通过它将细致的分析模型应用于社会现象。
>
> 对一个尚未毕业的工程师来说，课堂学到的东西都是早已成型的老生常谈。牛顿定律已经出现很长时间了。热力学定律也已经出现很长时间了。我们学到的微分方程，其他工科学生也早就解过无数回了。可突然间，我接触到了这样一位眼界开阔、语出惊人的思想家——他的各种思想都很独特，又很了不起。

在卡内基，费根鲍姆在培训中还遇到了赫伯特·西蒙，马歇尔计划的前任主管，同时也是一位对政治感兴趣的科学家。在对庞大官僚机构的研究中，西蒙观察到个人并不会作出合理决策，而是根据隐性或显性的规则来做出行为。他还观察到组织内的部门往往无视顾客的抱怨，只关注自身的目标——例如运输部门总希望收取最高的运输费用。（西蒙后来因为揭示了组织中理性的局限性而荣获诺贝尔经济学奖。）

20世纪50年代初，西蒙和艾伦·纽厄尔开始讨论利用计算机模拟人类思维的试验。纽厄尔曾在兰德公司某防空项目工作时学过计算机编程。他们一致同意程序应围绕着西蒙观察到的模型来设计：一个由子目标各不相同、按自有规则各行其是的元素组成的集合。

1955年，在费根鲍姆攻读西蒙的社会科学的数学模型高级课程时，纽厄尔和西蒙已经找到了一种策略。

> 我们度过了圣诞节假期后，西蒙开设了这堂课，并说他和纽厄尔已经发明了一台会

① 詹姆斯·马奇（James March，1928—）是管理大师心目中的大师，主要研究组织学及组织决策。他从1953年开始在卡内基理工学院任教，1970年加入斯坦福大学。

PDG

思考的机器，叫做“逻辑理论家”（Logic Theorist）。我们在课堂上问：“你说什么？会思考的机器？”——这个概念对我们来说太不可思议了。

西蒙告诉我们他所谓的机器是什么，并且分发了IBM 701计算机的使用手册。

我把手册拿回了家，读了一整个晚上。当第二天破晓时，我获得了新生——当时还没有“计算机科学家”这种称谓，但不管怎样，我认识到了自己想做的是什么。那么下一个问题就是：应该怎么做？

“逻辑理论家”程序试图发现罗素[①]和怀特黑德[②]的经典著作《数学原理》中命题演算的证明。在此之前，还没人编写过能自己做出发现的程序。

纽厄尔和西蒙首先对启发式进行了形式化。“启发式”（heuristic）这个术语借鉴自数学家乔治·波利亚[③]，是一种表达明确的解题技巧。比方说，如果想证明Y，且已知X必然包含Y，那么可以用证明X来代替证明Y。任何学过中学几何的人对这种启发式都很熟悉：当试图证明一个几何图形中的两条边相等时，可以看看包含这两条边的两个三角形是否全等。

然后他们制定了搜索策略，从一个全局命题列表中移除一个命题，然后对它应用所有可能的启发式。每一个启发式都可能会引入一个新的需要验证的命题，之后被加入到列表中，或者取代列表中的某个命题。

比方说，如果需要证明的命题是“苏格拉底终有一死”，同时给出公理“所有人都终有一死”，那么我们就能应用启发式，通过证明“苏格拉底是人”来取代证明“苏格拉底终有一死”。这种启发式和搜索的结合将在10年后为费根鲍姆阐述专家系统的思想起到主导作用。

费根鲍姆继续留在卡内基，在西蒙的工业管理研究生院攻读博士。他决定自己也应该学习编程。1956年夏他前往纽约城，开始为IBM工作。

我们这些年轻人被安排在一栋褐石建筑的一楼，就在IBM公司主楼的拐角处。他们给我们分配了办公桌和其他用品，我们就开始不停地工作。时不时有IBM的人给我们作讲座。

那个夏天的一天，一个人从我们这栋楼的四楼下来，告诉我们这些学生有一件了不起的事情正在发生。“你们一直坐在这里写着‘清除再加’、‘储存’之类的代码吧？你们再也不用干这些苦力活了。我们正在做的东西能让你们直接写公式，它就叫做‘公式翻译器’，简称Fortran。”到了秋天这个东西就诞生了。那个人就是约翰·巴科斯。

---

① 伯特兰·罗素（Bertrand Russell，1872—1970）是著名的英国哲学家、数学家和逻辑学家，并致力于哲学的大众化、普及化。1921年他曾来中国讲学，对中国学术界有相当影响。1950年获得诺贝尔文学奖。

② 阿尔弗雷德·怀特黑德（Alfred Whitehead，1861—1947）是英国数学家、哲学家。曾任教于剑桥大学和哈佛大学。他与罗素合著的《数学原理》（*Principia Mathematica*）标志着人类逻辑思维的巨大进步，是永久性的伟大学术著作之一。同时也创立了20世纪最庞大的形而上学体系。

③ 乔治·波利亚（George Polya，1887—1985）是著名的匈牙利裔美国犹太数学家和数学教育家，历任布朗大学和斯坦福大学教授。他在函数论、变分学、概率论、数论、组合数学以及计算数学和应用数学领域中都颇有建树。其重要数学著作包括《如何解题》、《数学发现》、《数学与猜想》等。

到了这一刻，费根鲍姆已经确定他的研究方向将会是计算机。他还不太确定该怎样做，但有些梦想已经开始成形。

> 在当时，20世纪50年代，我根本就没有考虑过实际应用。我只是被高智能甚至超智能的人造物这一愿景激起了兴趣。这就是我有时候自称“魔像建造者”的原因。建造魔像的不是布拉格的拉比吗？说不定我就是布拉格那位拉比的后裔。[①]

不过，费根鲍姆的博士论文却是关于心理学的。纽厄尔和西蒙在“逻辑理论家”程序中模拟了人类如何解决问题。也就是说，他们试图让计算机自己解决问题，目的是观察它们能否学到人类解决问题的方式。费根鲍姆在写博士论文时或多或少带有相同的目标，他模拟了一种有限的记忆能力。

> 如何在结构体系中进行机械式学习，是心理学实验室中长期研究的一个问题。它包括系列机械式学习和关联机械式学习。在实验室环境下，用程序足够细致地模拟人类处理信息的方式，就会产生出与人类相同的结果。

费根鲍姆把他的程序命名为基本识别和存储设备系统（Elementary Perceiver and Memorizer，简称EPAM）。直到今天，卡内基梅隆大学依然在使用这个程序。程序中包含了一个模型，模拟人类在刺激-反应环境下如何记忆成对的无意义单词，例如XUM-JUR和FAX-VUM。在刺激-反应环境下，实验者先展示一系列成对的刺激-反应单词，让被试者尝试记住它们。这称为“训练部分”。之后，实验者给出某个单词对中的一个单词，让被试者说出另一个。

这种学习过程让心理学家得以观察短时记忆的方式和能力，以及可能出现哪些记忆错误。例如，被试者在刺激单词彼此相似的情况下很可能会把反应单词弄混，而在插入其他学习任务时便常常记不住之前的成对单词。

费根鲍姆的目标是建立一种记忆组织的计算机模型，从而解释被试者的各种正确以及不正确的行为。他把这种记忆模型称为“鉴别网络”（discrimination net）。在训练的过程中，这种网络会努力持有足够的信息，以便区分目前所接收到的各种刺激。

在我们学习外语或某个新领域的术语时，也可能会用到这种方法。可能你常常听说“ROM”这个词，但只知道它和计算机的某种存储器有关。之后你又听说“RAM”也与计算机的存储器有关，而你不确定“RAM”和“ROM”是不是一样。可见，只有在你都听说了这两个单词之后才会试图去了解其中的差别。鉴别网络的工作原理也是这样。事实上，它还能正确地预判人类会犯的错误。

费根鲍姆获得博士学位后来到了伯克利任教。在其后的6年间，他继续与西蒙合作进行EPAM的扩展以及相关的项目。他们在各种心理学期刊上发表了很多文章。但费根鲍姆依然对建造一台超智能机器的梦想念念不忘。

---

① 故事中的拉比用巫术灌注黏土创造出了一个人偶，一开始把它当做仆人，最终却被其控制。

费根鲍姆是在伯克利的工商管理学院任教。当时伯克利还没有计算机科学系，而且学术政治也决定了近期不可能有。而与伯克利一水相邻，湾区对面的斯坦福大学由于管理层的支持，已经开始成为全美计算机科学的中心之一。乔治·福赛德[1]受聘组织计算机科学系，随后他就邀请了约翰·麦卡锡领导人工智能实验室。1965年麦卡锡说服费根鲍姆前往斯坦福加入了他的团队。

第一个专家系统

> 我认定自己并不想成为一个心理学家，用计算机来研究人类行为。我要自己来建造计算机。于是我从伯克利搬到了斯坦福，加入了这个新成立的计算机系。

早在1962年，费根鲍姆就曾与胡里安·费尔德曼[2]合作主编了一本人工智能的论文集，名为《计算机和思维》(*Computers and Thought*)。这本书是该领域早期最重要的著作之一。在前言中，费根鲍姆提议用计算机来考察归纳过程。

> 一直以来，人工智能都被认为与演绎过程有关——例如定理的证明、象棋对弈等等。而实证归纳（empirical induction）过程则是从广泛的数据中进行概括或假设。为什么人工智能不能进行实证归纳呢？

1878年，哲学家查尔斯·桑德斯·皮尔士[3]以一袋豆子为例解释了演绎和归纳之间的差别。如果我们知道袋子里的豆子都是白色的，那么就能通过“演绎”推断从袋子里拿出的下一颗豆子将会是白色的。而如果我们对袋子一无所知，但发现从里面拿出的豆子都是白色的，那么就可能“归纳”出袋子里的豆子全部是白色的。

因此，归纳是一种根据已有信息进行的猜测，而且会根据新出现的证据而发生变化。（归纳类似于麦卡锡提出的非单调推理。）

> 于是我开始思考。纽厄尔和西蒙曾教过我，要想在这些困难且不明确的问题上取得进展，最好的办法就是选择一些特殊的问题进行研究，就像他们研究象棋程序、证明命题演算的定理时一样。
>
> 我花了很长时间思考对于实证归纳来说，什么问题才算合适。两年多过去了，我还是没有确定下来。

费根鲍姆考虑了很多可能性——他甚至想去测试计算机能否根据一场棒球赛中的事件来归

---

① 乔治·福赛德（George Forsythe，1917—1972）作为斯坦福大学计算机科学系的创始人，从1965年开始担任该系主任及教授直到逝世。他也曾担任过美国计算机协会的主席，一生与人合著了4本计算机科学领域的书籍，并主编了75本该领域的其他书。

② 胡里安·费尔德曼（Julian Feldman）是加州大学信息与计算机科学系的名誉教授。

③ 查尔斯·桑德斯·皮尔士（Charles Sanders Peirce，1839—1914）是美国哲学家、逻辑学家、数学家和科学家，是数学、研究方法论、科学哲学、知识论和形而上学领域中的改革者。

纳出棒球的赛事规则。但这些想法看上去都不太合适。

到了1964年，在斯坦福召开的行为科学高级研究中心会议上，费根鲍姆认识了该校的遗传学系主任、诺贝尔奖获得者乔舒亚·莱德伯格[1]。莱德伯格提到他正在研究外空生物学——一门探索地外生命的学科。莱德伯格当时正在做一个火星探测器方面的工作，该设备将能降落到火星表面并四处检测是否有生命或前体分子存在。

> 这个探测器上会有一个关键仪器，就是质谱仪。莱德伯格的实验室当时正在用质谱分析法测量生命前体分子，尤其是氨基酸。
>
> 当他向我提到这件事时，我便想到这正是一个完美的实证归纳问题，而且正是大众所认为的科学家应做的那种事情：面对一大堆数据，科学家们拿出一台仪器然后用各种分析工具苦苦思索，最后得出一个假设，能够解释所有这些不同的数据。

但在与莱德伯格合作了几个月之后，费根鲍姆和他的团队发现他们缺少必要的化学知识。

> 我们所使用的人工智能已经颇为成熟了，没必要进一步开发——它们已经是很容易理解的搜索过程。我们需要的是化学知识，让这些过程能够产生正确的答案。

莱德伯格是世界级的遗传学家，但不是化学家。费根鲍姆和莱德伯格把斯坦福的另一位同事卡尔·杰拉西[2]拉入了项目。杰拉西以对质谱分析法的研究著称，同时也是口服避孕药的发明人之一。他为这个后来被称为Dendral的项目提供了必要的知识基础。

到了1965年，大功告成的Dendral项目成为了世界上第一个真正的专家系统（参见补充内容）。它能够确定分子的化学结构，有时甚至比杰拉西的学生做的还好，尽管还没有登上过火星。

Dendral及其后代

Dendral发展成了一系列项目。其起源是莱德伯格在1964年提出的算法，能根据已知的一组构成原子生成所有可能的无环结构（无环结构即不包括环路的结构，莱德伯格以这种结构开始，是因为它比较简单）。

1965年费根鲍姆和莱德伯格合作了启发式Dendral。它提供了一种框架，运用质谱分析数据和专家级化学规则来约束这一巨大的结构集合。

一条典型的规则可能会将某个分子结构的质量与其光谱峰值结合起来，以推导出某种特定结构的存在（例如酮基）。应用知识和启发式会极大地缩减可能的结构数量，通常会减少到不足200分之一。

该程序由三部分组成。“计划”阶段标识出最终结构所必须包含以及不可能包含的分子片段。“生成”阶段会创建出一系列与计划阶段的约束相吻合的可能结构。“检验”阶段则通过模

---

① 乔舒亚·莱德伯格（Joshua Lederberg，1925—2008）是美国分子生物学家，主要研究方向为遗传学、人工智能和太空探索。因发现细菌遗传物质及基因重组现象而获得1958年诺贝尔生理学或医学奖。

② 卡尔·杰拉西（Carl Djerassi，中文名为翟若适，1923—）是奥地利籍美国化学家、小说家和剧作家，美国国家科学院院士、美国科学与艺术学院院士。

拟该结构在质谱仪上的读数，对这一系列可能的结构进行排序。

变体Dendral

下一个重要的进展发生在1970年，该程序被称为变体Dendral（Meta-Dendral）。它的目标是推断出新的质谱分析规则，添加进Dendral之中。根据若干对已知的原子结构及其对应的质谱数据，变体Dendral就能提出一些规则，用来从质谱仪的数据中推断出结构。这些规则中有一些被证明是全新的，因此改进了原有Dendral的性能。[①]

各种不同形式的Dendral不仅解决了特定的问题，而且还提出了专家系统所通用的框架。每个专家系统都包含一个数据集合（在Dendral中即为原子构成和质谱仪测量数据）、一个假设集合（可能的结构），以及一个用来筛选假设的规则集合（将质谱与结构约束进行联系）。

这一框架是否适用于其他领域还有待检验。费根鲍姆相信它可以——只要他能找到合适的专家。

20世纪70年代初，在斯坦利·科恩[②]、布鲁斯·布坎南[③]和爱德华·肖特里夫[④]的合作下，Mycin诞生了，这是一个帮助医生诊断传染病并提供治疗建议的专家系统。科恩后来开发了基因重组的基础性技术，肖特里夫和布坎南则是Mycin的主要实现者。

Mycin的一项重大革新是它能够衡量概率，并以此计算某些结论的可能性。某条规则很可能会把以下事实联系起来：病征是细菌性感染、繁殖环境是无菌的、感染由肠道开始。那么Mycin就会有70%的把握得出结论：这种微生物是拟杆菌属（bacteroides）。有了这种可信度系数，内科医师就能记录多种假设，并将它们从最可能到最不可能进行排序。

> 医学很棒的一点在于，它是一个极好的演练沙盘——启发式无处不在。医学就是这样。这里面几乎没有生搬硬套，也从不像白纸黑字那样绝对。医学是一门艺术，需要经验和知识，也有规矩可循，能进行准确的猜测和良好的判断。所有这些综合在一起，再加上推理过程，当然还有真实的数据，医生会测量病患的各种体征、做各种化验，然后努力对检测结果进行判断。
>
> 人工智能方法就是在满是不成形的问题的领域中进行选择的方法。一般领域内的许多可计算问题，不管是物理运算还是工资计算，都是成形的问题，有对应的算法。解决这种问题我们不需要做任何究查。

① 更多有关Dendral的信息请参阅《人工智能手册》（*The Handbook of Artificial Intelligence*）一书第二卷。阿福隆·贝尔（Avron Bear）与爱德华·费根鲍姆合著，（加州洛斯拉图斯）威廉·考夫曼出版社，1982年自费出版，106ff。

② 斯坦利·科恩（Stanley Cohen，1922—）是美国生物化学家。1986年获得诺贝尔生理学或医学奖及美国国家科学奖章，1987年获得富兰克林奖章。

③ 布鲁斯·布坎南（Bruce Buchanan），美国信息科学家、医学家。

④ 爱德华·肖特里夫（Edward H. Shortliffe，1947—）是加拿大籍美国生物医学信息专家和计算机科学家，人工智能在医学领域的先驱之一。

知识原则

不成形的问题到处都是，但费根鲍姆相信其中有一些比其他的要好。他认为最关键的是要有获得一致认同的知识体系。

> 在有些领域里专家能够明确说出自己选择和判断的依据，这些领域正适于我们在早期阶段进行研究。
>
> 比如医学和工程领域就很合适，因为我们能明确地表达自己是如何思考问题的。相反，在经济学中就没有真正意义上的明确的知识体系可以用来作出经济预测。
>
> 我想，随着这些年来技术的发展，人们已经获得了更多的技能，应用起来也更有信心。这些技术已经被应用到我在20世纪70年代根本无法想象的问题中，像经济预测和外汇交易。如今人们愿意利用一些模型来工作，有人可能会称其为“授权模型”。也就是说，如果没有一个广泛认同的专家体系，但某个个人也可能会有一套表达清晰的知识体系（其他人可能不一定同意），然后提出这些知识，并说这是代表某种知识的专家系统。现在这种事情大行其道。

在写作本书时，许多航空公司都在利用专家系统来决定如何在多个机场间部署航班，以便在恶劣天气、机械故障或机场拥堵等情况下满足需求。以美国航空公司（American Airlines）为例，旗下的550架飞机中每架飞机连续飞同一航线的时间平均只有3天。还有一些专家系统则会帮助避免飞机相撞事故。

欧洲一些繁忙的火车站，例如巴黎里昂火车站，也使用专家系统安排列车进入不同的站台，以便减少列车切换轨道。

计算机硬件生产商利用专家系统来制定各种类型的处理器、存储器、硬盘、网络、打印机和显示器之间的组合，从而在最大程度上迎合顾客。对于惠普这样的厂家来说，这意味着要在超过2万种可能中作出选择。

医院则利用专家系统来确定重病特别护理中病患的存活几率，所有病患视情况分为1到100个等级。它还能建议哪些病人应获得医院的多少资源。并非只有医生才能扮演上帝。

专家系统还能帮助战场上的指挥官们作出决策。

> 这里有一个例子：在伊拉克入侵科威特之后，ARPA（Advanced Research Project Agency，美国国防部高级研究计划署，为各类国防研究提供资金）要求一些合同承包商将其订购的一个用于生产调度的计算机程序改为物流调度程序，以便后勤官员把人员和物资从美国和欧洲调度到沙特阿拉伯。而他们成功地完成了这一程序。
>
> 该署的负责人说，这一个程序便收回了ARPA在人工智能领域的所有投资。

在费根鲍姆身上，我们找到了学术和商业的一种独特结合。作为智能遗传公司（IntelliGenetics）的创立者、技术知识公司（Teknowledge）和智能公司（IntelliCorp）的董事，他引领了生物遗传学、工程学和常规商业应用等专家系统的开发。费根鲍姆在斯坦福大学创立的研

究中心，即斯坦福启发式编程项目，已经培养了大批学子，他们将自己从中积累的经验带到了世界各地，费根鲍姆的专家系统概念获得了广泛的传播。

> 一门新的技术不会自己走进现实生活。它只能依靠个人或组织对这种技术的拥护才能融入这个世界。技术不会推销自己。

有一件事他的学生们无疑都听说过，就是费根鲍姆坚信要以充分、精确的知识来装备专家系统。他称之为“知识原则”。

> 假设我们现在是在乔治敦大学，那里有一所优秀的医学院，还有一个极为杰出的数学系。那么我们坐在这里，突然有一个人病倒了——心脏病之类的。我们会赶紧把他送到医院去，因为那里的人都接受过医学训练。我们不会把他送到满是推理大师的数学系去。对吧？因为推理在此不能解决任何问题。你需要懂医学的人，而不是推理。

在成为美国空军的首席科学家之前，费根鲍姆是斯坦福大学知识系统实验室的联合主任之一。他利用这个职位作为讲台来倡导专家系统的广泛使用，有时候也用来劝诫自己的同事。

> 我非常高兴这一技术已被广泛采用，它能带来很多好处。但是我很困惑，为什么这样强大的技术却没有获得巨大的成功。

费根鲍姆相信，许多专家系统在市场开拓及自我推介方面都存在着问题。

> 这个领域到处都在出售没有知识的软件外壳（推理软件），他们并没有真正出售知识。第二个原因是我们无法突破知识获取的瓶颈。获取知识的难度太大。与专家一起工作、从他们那里获取知识、让他们清楚地表达该领域的模型，这是一个困难而且旷日持久的问题，而且我们没有为此提供任何工具。
>
> 所谓知识原则，指的是专家系统的性能取决于知识库，而非逻辑。所以如果知识过于陈旧，或者没有经常更新，那么系统的性能就会随着时间流逝而削弱。

在将专家系统和常规的数据库系统（以记录的格式储存数据，但不会对其进行推理）进行比较时，费根鲍姆再次指出知识正是造成这种公认差别的根本原因。

> 在工业应用的整个第一代基于知识库的系统中，没人对知识库进行过再利用。比方说，没人用解决钢铁工业问题的知识解决其他问题。没人这样做过。他们总是为每一个新的问题重新构造一个知识库。而在数据库系统中却不会出现这种情况。你搭建起一个数据库，不同的人可以将其用于不同目的。

道格拉斯·莱纳特的Cyc项目（参见下一章）在逻辑层面体现了费根鲍姆的知识原则。Cyc试图将一个生活在20世纪末的北美成年人的所有日常知识编码进一个计算机程序中。其主要动机之一是创建一个可反复利用的基本常识库，便于人们从中构建起专家系统。这些知识可以避免系统接受明显荒谬的数据（例如2微秒飞行3000英里），或者得出类似的荒谬结论。

费根鲍姆以知识原则代言人的身份四处周游。在商业行程的间隙，他沉溺于对热带岛屿的热爱，尤其是巴厘岛和新几内亚。在加利福尼亚，他在斯坦福大学合唱队担任男中音。他与不同的作者合作（其中包括他的妻子，同样是一位计算机科学家）出版了数本关于专家系统和商业领域的著作。

费根鲍姆深信专家系统必将获得其应有的地位，成为一种智能主体与人类并肩合作，解决这个世界上最富挑战的问题。

> 人类极其擅长解决问题和学习，也极其擅长协调感官和运动机能，并融入自己解决问题的能力。然而，我们这些非凡的生物却没有从进化中获得计算机程序所拥有的那种智能环境。
>
> 我们必须认清，人类是一个智能主体，而计算机程序则是另一个智能主体，二者的能力可以互为补充。我们必须设计出自己的系统，以便二者能够协同工作，从而获得更好的结果。
>
> 从收割机的问世到正式投入农业使用，这之间相隔 25 年。从电话的发明到收费电话出现，同样经过了 25 年。我们还有很长的路要走。

14

道格拉斯·B.莱纳特
一场二十年的豪赌

有多少人一生中能有2%到10%的机会对这个世界产生重大的影响？如果真有这样的机会，你应该抓住它。

——道格拉斯·B.莱纳特

和艺术家一样，科学家常常需要作出权衡，要么甘愿遭到嘲笑，要么选择默默无闻。“人们会喜欢这项成果吗？”这个问题决定了你的论文是否会被业界接受，也决定了人们对你是尊重、嘲笑还是根本就不把你当回事。对于道格拉斯·莱纳特来说有一件事是肯定的：最后这种可能绝不存在。

当莱纳特还是一位年轻的研究者时，就已经开始着手构建程序解决人工智能的中心问题：让机器学习，并为其注入知识和常识。他以纯粹探险家的精神，手拿一张简陋的地图就开始向前跋涉。在某些批评家看来，他步了传统AI界的后尘，走进了一个再也无法回头的死胡同。另一些批评家则既怀疑又愤懑，他们认为在理论完全成熟之前，任何人都不应尝试去解决AI的中心问题。甚至连本书的作者都感受到了这些攻击的猛烈。“为什么写莱纳特？”很多人都问过这个问题。读者很快就能知晓其中的原因。

远在杂碎之上

莱纳特1950年出生于费城，在那里和特拉华州的威明顿度过了童年时光。他家做苏打水装瓶生意。6年级时，莱纳特在学校图书馆发现了艾萨克·阿西莫夫[①]写的物理和生物科普读物。科学满足了他对世界如何运转的好奇心。到12岁半时，他的父亲突然病故。

---

① 艾萨克·阿西莫夫（Isaac Asimov，1920—1992）出生于俄罗斯，美国犹太人作家与生物化学教授、门萨学会会员。他是公认的科幻大师，与儒勒·凡尔纳、H. G.威尔斯并称为科幻历史上的三巨头，同时还与罗伯特·海因莱因（见第3章注）、亚瑟·克拉克并列为科幻小说的三巨头。他创作力丰沛，产量惊人，一生创作和主编过的书籍超过500册，范围涵盖科幻、科普和历史等，除了哲学类以外，涉及了整个“杜威十进制图书分类法”。

PDG

在12岁半之前，我一直都信仰宗教（犹太教）。就在我成年礼[①]的前几个月，我的父亲去世了。这种事情会让你认识到这个世界天生就是不公平的，而且不管上帝的客观真理是什么，当不幸的事情发生时，信奉这些教规戒律根本没有用。

在父亲去世后，莱纳特一家频繁搬迁，他发现自己总得在新的学校从头开始。

我总是被安排到差班里面，也就是所谓的“杂碎”班级。因为前一年我不在这里，所以他们就不让我进好班。你必须不断去证明自己，而不能破罐破摔。好班的学生被认为肯定能学好，所以他们总是养尊处优。我的同班同学则被认为肯定学不好，所以必须分外刻苦才能克服这些恶劣条件。

在这段时间里，年轻的莱纳特将科学作为自己的慰藉。他的天赋显露无遗。1967年他描述了第n个素数的闭型定义，以此获得了某次国际科学博览会的参展邀请。莱纳特和另一位优胜者一起前往底特律，一切费用都由主办方承担。彼时的莱纳特对举办地点还颇为失望，因为上一届展会是在东京举办的。不过展会还有其他的好处，参展的选手将有机会获得领域内科学家、研究人员和工程师的评价。

在此之前，我所认识的最接近科学家的人是我的高中科学老师。

1968年，莱纳特进入宾夕法尼亚大学就读。越战正如火如荼，而莱纳特抽到的征兵号数值很低，有可能不得不开赴战场。这些不确定性促使他抓紧时间，奋发图强。

刚进入大学时，莱纳特的兴趣是物理和数学，但最终他改变了初衷。

我对数学的研究足以让我意识到自己不会成为一名伟大的数学家，对物理的研究则让我意识到从某种角度来说它的基础并不牢固。人们花一辈子时间来求解某个数学难题，或者是像爱因斯坦的天体物理学方程这种不知道对物理或现实是否有意义的问题。人们怀里揣着越来越多的各种基本粒子的笔记，但我们只是发现了它们，却并不真正理解。与物理现实相脱节的事情越来越多。

1971年，约翰·卡尔（John W. Carr III）教授的一门课向莱纳特敞开了人工智能的大门。计算机为人工智能创造了技术条件，但莱纳特认为该领域的研究水平依然处于早期阶段。

这有点像望远镜发明之后的天文学研究。这个领域本身就很吸引人，但它还有两个非常有趣的特点。

一是它可以提供积极的支撑——你可以构建像智能增强器这样的东西，它们能让你更聪明，因此能做更多的事情，而且做得更好。

第二个有趣的地方是这个领域的研究人员很明显还不知道自己到底正在干什么。

---

① 犹太男童到十三岁、女童十二岁时便会举行成年礼，象征他们在身体、心智及道德上踏入新的阶段，有义务履行犹太律例。通常会在孩子生日后的安息日举行。

爱德华·费根鲍姆和约翰·麦卡锡可能会对这一点提出反对，但的确还没人能够构建出真正解决许多基础问题的程序。

在莱纳特读书时，人工智能领域最成功的应用是基于规则的系统，例如麻省理工学院用于解决微积分问题的Macsyma，和费根鲍姆用于模拟专家的Dendral。这两者都在20世纪60年代中期完成，但都只表现了极为有限的智能形式，例如没有考虑学习和日常推理（约翰·麦卡锡对后者有所研究）。正如当时坊间所流传的一句话："模拟一位地质学家要比模拟5岁小孩容易得多。"

但莱纳特并未气馁，因为智能增强的希望让这一切看上去都颇为值得。

> 19世纪的物理学和直到当今的工程学都对人类的"身体"机能提供了"增强"。我们能到达的地方比步行的距离更远，速度也更快。我们彼此间能交流的距离也完全超过了大声叫喊的范围。
>
> 人工智能的目标就是一种"智能增强"，这样我们就能更聪明、更具创造力，可以更快地解决更难的问题，同时减少遗忘、增强记忆。

1972年莱纳特从宾夕法尼亚大学毕业，获得了数学及物理双学士学位、应用数学的硕士学位（当时美国在越南的战事逐渐偃旗息鼓，而莱纳特的征兵号幸运地没有被抽中）。他谢绝了斯坦福大学的聘请，前往加州理工学院攻读博士。但读了几个月的技术文献之后，莱纳特认为还是斯坦福和麻省理工的人工智能更为活跃。他给麦卡锡打了个电话，询问是否还能加入斯坦福。麦卡锡同意了，所以莱纳特打点行装，开车从帕萨迪纳去了旧金山湾区。

结果麦卡锡不久便在休假时去了麻省理工。莱纳特在计算机科学系一个刚任教一年的助理教授考德尔·格林（Cordell Green）手下开始了学习。格林在读博士时运用麦卡锡的情境演算完成了自动编程的理论研究（参见补充内容"自动编程"），在到斯坦福之后便转向了实验研究法，而且像许多虔诚的皈依者那样把这种转变强加到了莱纳特身上。

自动编程

自动编程（automatic programming）是一种减少编程工作量的尝试。通过自动编程，用户向计算机陈述某个问题，然后让计算机来提出解决方法。例如，用户可以把一系列税务法规提交给计算机，要求计算机生成程序来减少自己的纳税额。

尽管这一目标尚未获得成功，但该领域的某些思想却启发了新的编程语言，例如Prolog和一系列"约束逻辑编程语言"。程序员利用这些语言向计算机提出某个问题的正式陈述及其约束条件，而计算机则尝试寻求解答。计算机生成的程序通常比用最好的算法编写的程序要慢得多，但世界各地的研究人员都在尽力缩减其中的差距。

有意栽花花不成，无心插柳柳成荫，这是人工智能项目的典型命运。人工智能研究者们提出的问题和技巧影响了一些明显无关的学科，例如数据库管理、机器人技术、符号编程语言和图形学等。例如，机器人最早是在人工智能实验室中诞生的，而运动规划、传感/驱动器协同等应用仍旧在使用着人工智能提出的算法。20世纪90年代的绝大多数数据库管理研究都涉及如何有效地将专家系统型规则融入到数据库系统中。这些研究与诞生它们的人工智能之间唯一的不同，就是对效率和可证明正确性的关注。

> 在去斯坦福之前，我认为自己是一个形式主义者。考德尔（以及我日后的导师费根鲍姆、布坎南）让我认识到了实际经验的价值——观察数据、做实验、利用计算机来检验可证伪假说——哪怕是在人工智能这样的领域。

“可证伪假说”（falsifiable hypotheses）系由哲学家卡尔·波普尔[①]提出，本意是用于解释自然科学理论，但却在人工智能领域扮演了重要的角色。按照波普尔自己的解释，他是在尝试解决科学理论的验证时发展出这一概念的。一个与理论相符的实验并不足以构成证明，因为下一次实验有可能就会推翻这一理论。

在仔细考虑了这一问题之后，波普尔得出了结论：要证明一个科学理论实际上是不可能的。这一洞见很快导致了一个问题，那就是如何将科学理论与非科学理论（例如宗教理论）区分开来。

根据波普尔的说法，如果一个断言能够被观察所检验，那么它就是一个好的科学理论。当神发怒时就会下雨这个断言是不可检验的，因为谁也不可能直接观察到神发怒。换句话说，不存在实验能够产生反驳这一断言的结果。相比之下，下雨前的天空应该有云这一理论就是可以验证的：一旦你发现下雨时天空无云，就能证明该理论是错误的。

于是，波普尔认为，如果一个理论是“可证伪的”，也就是可以有实验证明它是错误的，那么该理论就是科学的。

波普尔的观点对于物理学来说非常适用，因为该领域的理论都是对自然的推测，因此也是可证伪的。例如，当迈克耳孙－莫雷实验[②]证明在运动的火车上光并不遵循同声音或抛出的球一样的规则时，牛顿力学就被推翻了。用实验来证伪对于数学并不适用，因为这个领域都是一些同义反复：最终可约简为A的语句就是A。比如说，假设有断言“奇数乘以3得到一个奇数”以及相应的证明，有谁能设计一种合理的方法来检验它的可证伪性？

计算机科学处于一个奇怪的中间地带。理论计算机科学本质上就是数学。典型的情况是，人们证明某个算法只需多少步就能解决某个类型的问题。这种证明可约简为数学上的同义反复，因此是不可检验的。

相比之下，人工智能程序直接与外部世界相互作用，因此是可被检验的。一个典型的（也是不言而喻的）可证伪预测就是“这个医学专家系统拯救的病人将会比它伤害的多”。但一些研究人员依然不经过严格的检验便随意创造理论。

例如，一位人工智能科学家可能会提出一条理论，然后选择一个很小的范围进行检验，以表示他的程序可以处理所有可能的结果。用这种方法编写程序，其理论自然毫无意义。

人工智能的方法论仍旧是一个弱项，存在着很多批评的声音。莱纳特认为这种缺陷与计算机

---

① 卡尔·雷蒙德·波普尔爵士（Sir Karl Raimund Popper，1902—1994），奥地利裔犹太人，20世纪最著名的学术理论家、哲学家之一，在社会学上亦有建树。

② 迈克耳孙－莫雷实验是为了观测“以太”是否存在而进行的实验，是1887年由阿尔伯特·迈克耳孙（Albert Michelson，1852—1931，1907年诺贝尔物理学奖获得者）与爱德华·莫雷（Edward Morley，1838—1923）合作进行的。

科学短暂的发展历史有关，还需要时间来改善。

> 计算机科学其实很像 1740 年的物理学。人们对任何结果都很欢迎，甚至不太在乎这些结果是否可再现或复制。它尚处于发现阶段，远未进入开拓阶段。

AM 程序：寻找合适的启发式

莱纳特的第一个人工智能实验是用Lisp语言编程，以模拟数学概念的形成。用他自己的话来说，它能够“生成一些数字，然后跟它们玩耍”。

> 一开始（它的名字）是指“自动数学家”（Automated Mathematician），但在 1974 到 1975 年时我开始谦虚起来，只用缩写 AM 来称呼它。

莱纳特用AM程序写就了博士论文，这也是最早的人工智能程序之一。正如纽约大学的人工智能研究员厄尼·戴维斯（Ernie Davis）所说：“它和我所知的任何东西都不同：一个没有输入、没有交互的程序，永远在那里思考数学概念，并提出各种假设。”

AM在科学上最主要的贡献是它体现了启发式学习的概念。读者可能还记得费根鲍姆一章的内容，计算机科学的启发式学习法是一种解决问题的方法，它是一种基于经验的猜测，其解决方案并不能保证是最好的，但往往也还不错。人工智能经常使用启发式，因为人工智能程序所得到的信息往往不够完整。在专家系统中，启发式体现的是针对具体情况的专业性知识。AM则利用启发式来发现新的知识。有些启发式甚至能应用于数学之外。以下是一些典型的例子。

(1) 如果一个操作作用于同一类型的两个对象，那么就将其作用于同一个对象两次。

AM利用这一点从乘法发现了平方的概念。乘法作用于两个数——而“平方”则是对同一个数进行两次的乘法。与之类似，“加倍”则来自于加法。在数学中有很多专用词汇就是用于表现这样得出的概念，这也表明了这种启发式在数学中的价值。

即使是在数学之外，体现这种启发式的概念也有各自专用的词汇。例如，“自杀”来自于杀死某人，“自我分析”来自于分析，“内战”则来自于战争。

(2) 寻找极端情况。

AM利用“求因数[①]”这一操作来寻找零因数的数（没有）、只有一个因数的数（数1）和只有两个因数的数（素数）。

极端值在数学之外的领域也有专用词汇。例如政治思想中的无政府主义和极权主义，物理中的绝对零度，等等。

(3) 如果两个分别独立得到的概念其实是相同的，则继续深入考察。

AM利用这一点对1这个既是只有一个因数的数，又是乘法单位元的数进行了深入的考察。很

---

① 因数又称因子、约数，是一个常见的数学名词。假如整数n除以m，结果是无余数的整数，那么我们就称m是n的因数。

多人用这种启发式来决定看哪场电影，或者买哪辆车——有媒体的好评当然很好，但媒体的好评再加上一个朋友的好评无疑会更好。

> 我们的想法是给它一堆定义和概念，再给它一堆用于判断各种有利价值的启发式，然后坐等看它能够发现什么。

AM基于的是115套理论概念和243个启发式规则。以此为基础，AM能够“发现”300个数学概念。每个发现都涉及20到50个启发式，包括上述三种以及各种类推概念。除了绝大多数小学六年级程度的算术之外，AM还作了一些猜想，例如“任何比3大的偶数都是两个素数之和”，也就是已知的哥德巴赫猜想。能作出这种猜想对于计算机程序来说是一项令人印象深刻的成就。但另一方面，程序并未发现一些莱纳特认为应该发现的东西。

> 我们事先安排了一些情境，想看看它能否做出合理的反应。但是我们设想的情境并未出现。我们以为它会发现各种类型的无穷大和康托集[①]，但它没有做到。

AM激起了斯坦福大学许多知名数学家和计算机科学家的好奇心，包括乔治·波利亚。波利亚曾写过一些书，主张最好的数学教育方法就是教孩子们自己去发现。

波利亚是从AM的一个结果开始了解它的。AM曾考察过包含多个因数的最小数。例如，包含6个因数的最小数是12——1、2、3、4、6和12本身。AM试图找到包含7个、8个以及更多因数的最小数。莱纳特怀疑是否有数学家曾考虑过这一问题。波利亚似乎是唯一知道这个问题的人。

> 他说：“这看起来很像我朋友的一个学生曾做过的事情。”波利亚当时大概92岁高龄。后来我发现他的这个朋友就是戈弗雷·哈代，这个学生就是斯里尼瓦沙·拉马努金[②]。在20世纪早期，拉马努金就提出了关于高度合成数的一些问题，与AM发现的模式之一非常类似。

在斯坦福还有其他一些大师在关注着AM的进展，其中也包括高德纳。

> 高德纳曾经把AM发现的那些结论当做古巴比伦或阿兹特克卷轴或象形文字那样仔细研究。他寻找的是我和系统并未意识到，但却非常有价值的东西。
>
> 他不断地发现一些AM提出但未意识到其重要性（我也未意识到）的东西，然后他会画上圈。这简直太棒了。

---

① 康托集（Cantor set）是一个数学集合概念，系由德国数学家格奥尔格·康托（Georg Ferdinand Ludwig Philipp Cantor，1845—1918）在1883年引入，它是位于一条线段上的一些点的集合，具有许多显著和深刻的性质。最常见的构造是康托三分点集，由不断去掉一条线段的中间三分之一得出。

② 斯里尼瓦沙·拉马努金（Srinivasa Ramanujan，1887—1920）是印度数学家，没受过正规的高等数学教育，但沉迷数论，尤爱牵涉π、质数等数学常数的求和公式，以及整数分拆。

AM获得了成功，但仅限于数学中一个极为有限的领域。当该程序摆脱初等集合理论继续深入时，就开始了“逆行”（莱纳特用这个词来形容它无效的路径搜索对时间的浪费），启发式也失去了威力。

1976年在斯坦福完成了博士论文之后，莱纳特前往卡内基梅隆大学担任助理教授，同时决定将自己工作的重心放在启发式的系统性研究之上。在卡内基梅隆以及两年后重返斯坦福这段时间里，他开发了自己在人工智能方面的第二个重要的实验性程序，称为Eurisko。

Eurisko的目标是发现新的启发式，而不是仅仅利用给定的那些。同时它还要创造达到学术期刊水平的数学成果。

不过，Eurisko主要的成功在于完成了一些创新的电路设计，同时还在一个叫做“旅行者”的海战游戏比赛中构建了一支舰队拔得头筹。程序运用自己的启发式找到了少许被游戏设计师所忽略的漏洞。例如，舰队可以自我沉没受损的船只以增强机动性。在1981和1982两年夺魁后，比赛主办方禁止莱纳特和他的程序参赛。

但这个程序仍然很难扩展到其他领域。根据在AM和Eurisko中积累的经验，莱纳特在1983年得出结论：即使是最聪明的启发式也是不够的。

> 学习只在已知事物的边缘发生，所以人们只可能学到与自己已知相似的新东西。如果你试图学习的东西与你已知的东西距离不远，那么你就能学会。这个边缘的范围越大（你已知的东西越多），就越有可能发现新的事物。

Cyc：用常识来武装计算机

莱纳特由此推论，如果学习依赖于已知事物，那么如果计算机程序想发现真正有趣的东西，它本身就必须知道得像人类一样多。

> 我的想法是先暂停10年，先把足够多的知识填进计算机，然后在20世纪90年代中期再回过头来让机器开始学习。

莱纳特希望能将一个见多识广的北美成年人所拥有的知识灌输到计算机程序中。他认为有了这些知识作为基础，程序就能进行一些有趣的学习。这个程序的名字叫做Cyc，来自“encyclopedia”（百科全书）一词。

> 艾伦·凯、马文·明斯基和我一起做了一些粗略的计算，看看需要输入多少知识，以及大概需要多少时间。结果我们发现需要10年，100万帧。

明斯基在计算机中用“帧”这个字眼来描述现实世界的物体。例如，一个关于汽车的帧可能会说明汽车是一个实际物体，高度大约4英尺、长度在6到40英尺之间，有轮子、引擎、座位，以及用于转向、加速、刹车的装置。

完全依靠手工输入100万帧知识无疑是一项巨大的工作，三位科学家很快得出结论，这样规

模的项目相应也需要巨大的资源支撑。这可不是一般机构能够提供的。

早在1975年，莱纳特还是个研究生时，曾在格鲁吉亚首府第比利斯（那时候还属于前苏联）出席了国际人工智能联合会议（International Joint Conference of Artificial Intelligence，IJCAI-75）。这件事情有两个地方颇值得纪念。那是莱纳特第一次公开做演讲，他还记得那台前苏联产的幻灯机上的大功率风扇将他的幻灯片吹散在讲台上时的惨状。而第二件事对莱纳特的研究生涯更有意义：他结识了伍迪·布莱索[①]。两位科学家在会议期间针对莱纳特当时的研究进行了积极的探讨。8年后的1983年，布莱索成为了微电子及计算机技术公司（MCC）的人工智能部主任。该公司是一家创立于得克萨斯奥斯汀的新兴企业。他安排莱纳特与公司的负责人、海军上将波比·雷·英曼[②]见了面。

在进入计算机行业之前，英曼长期在政府情报部门工作，度过了一段辉煌生涯。他曾为多个海军情报职能部门服务，后成为美国国家安全局主管，最终官至中央情报局副局长。1994年他有望出任美国国防部长，但由于不愿忍受媒体的攻击而中途退出。

> 英曼是我见过的最具魅力的人。他待人极为友善，而且非常真诚直率。彼此咬牙切齿的人去找了他之后都会满意地走出他的办公室，各自觉得自己才是胜利者。
>
> 他真的具有这种魔法般的能力，MCC 的每个人都喜欢他。我们每隔一两个月就会和 MCC 召开会议，但每次的话题都不可避免地转到国际事务上去，因为所有人都特别想了解他对世界局势的看法。那简直就像是圆桌武士会议。

和其他科学领域一样，资金决定了计算机科学的命运。该领域中许多伟大的进展都是在大型企业的研发部门诞生的。巴科斯在IBM研究中心开发了Fortran，而UNIX则来自于贝尔实验室。学术研究人员的资金支持主要来自于政府科研部门、商业机构或军方。

> 英曼对我说："我们这里（MCC）是唯一的地方，也是史上唯一的机会，让你去完成这个项目、一个成功率只有 2%的项目。你不可能获得来自政府或学术界的资金。一个独立的公司，哪怕是有几十亿美金的公司，也不可能在这上面投入超过 1 千万。"

莱纳特明白，不仅是MCC和它的股东需要投入大量的财力，他自己也要面临将大部分工作重心投入到一个很可能不会成功的项目中去。

> 爱德华·费根鲍姆对我的影响是，让我明白作为研究人员，我们其实就是在拿自己生命中的三十年进行赌博。所以，我们最好让每一年都有所作为。

如果莱纳特真能把一个成年人的知识赋予一台计算机，那么对这种程序的应用所带来的益处

---

① 伍迪·布莱索（Woody Bledsoe，1921—1995）是美国数学家、计算机科学家和杰出的教育家，人工智能、模式识别和自动定理证明的先驱人物。

② 波比·雷·英曼（Bobby Ray Inman，1931—）曾是美国海军上将，在情报界具有极大影响力。从海军退役后他先后成为微电子及计算机技术公司（MCC）、韦斯特马克系统公司（Westmark Systems, Inc.）等多家公司的董事会主席及CEO。

将是极为可观的。

> 这可以从本质上让自然语言前端和机器学习前端融入到程序中。它能实现应用软件间的知识共享，例如不同的专家系统彼此间可以共享规则。很显然这将是计算方式的一次真正的革命（如果能够成功的话）。但我们成功的几率确实小到难以置信。
>
> 它存在着各种缺陷——我们如何表示时间、空间、因果关系、物质、设备、食物、意向等概念？这些概念在过去30年中一直在困扰着人工智能的研究人员，在过去30个世纪中也一直在困扰着所有的哲学家。他们把毕生都献给了对这些细微差别的研究。

在开始数年中，莱纳特一直在试验各种表示知识的方法。后来他认识到无法用一种表示方法来应对所有的知识集合，因为知识极其依赖于上下文背景。幸运的是，一位年轻而且才华卓越的博士生R. V. 古哈（麦卡锡和费根鲍姆的门生）提出了一种“微理论”的模式。

> 购物或驾车等人类行为都有其对应的微理论，我们需要对不同的微理论作出不同的假设和决策。
>
> 最重要的是，我们必须放弃全局一致的理想，接受局部一致。每一个理论、每一个上下文背景自身必定是一致的，但理论和理论之间并不一致。
>
> 例如，我问：“德拉库拉[①]是谁？”你会说“一个吸血鬼。”然后我问：“吸血鬼存在吗？”你会说：“不存在。”这之中的矛盾根本就不会对你造成困扰。回答第一个问题时的上下文背景是布拉姆·斯托克的小说，而回答第二个问题时的上下文背景则是我们科学的、无神论的日常世界。尽管这些看起来完全相互矛盾，但你毫不费力就能区分它们。

Cyc运用微理论作出的断言，在特定的上下文背景中是成立的。例如，运用日常生活中“朴素物理”的微理论，Cyc断言没有支撑的物体会下落。而更为复杂的微理论可能会在某些情况下与这一规则相抵触，例如存在着磁浮力场，或者该物体是个氢气球。莱纳特相信微理论对于理解“比喻”是极为重要的，而比喻在我们的日常生活中也是极为重要的。

> 如果你不能理解比喻，就无法真正理解语言，甚至连《纽约时报》上一家公司收购另一家的豆腐块新闻都读不懂：那里面到处都是比喻，例如捕食、掠夺、拉锯、战争等等。在中学我们就能看到像“我们的祖国母亲胸怀博大”这样的作文，如果没有丰富、广泛的知识根本无法理解这些语句。
>
> 绝大多数希望包罗万象的系统都以失败告终，这其中的关键在于它们都试图保持全局的一致性：把所有的断言统一到一个背景下，把所有上下文背景中的各种假设全部写出来。这一点是无法实现的。你可以把每一种背景中的假设都写出来，但总有一些会被遗漏。

---

① 大名鼎鼎的德拉库拉（Dracula）是19世纪爱尔兰作家布拉姆·斯托克（Bram Stoker，1847—1912）1897年的小说《德拉库拉》中的角色，是一个靠鲜血为生的怪物，原型来自中世纪的瓦拉几亚大公弗拉德三世。

在莱纳特的主导下，MCC有30个人负责为计算机输入知识——包括购物、足球、看医生等成千上万种日常行为。莱纳特的团队需要知道每一个知识条目的上下文背景。但他们同时还必须找到一种方法来表现事实和观念之间的关系。这涉及微妙的逻辑概念，例如“模态算子”(modal operator)[①]。

> 假设我说比尔·克林顿不知道我的年龄是42，而我的年龄是42。通常，我们可以用与某事物相等的事物来代替它，但若这样，这里的陈述就会变成比尔·克林顿不知道42是42。一旦我们使用类似“知道”、“相信”、“打算”或“渴望”这样的模态时，就不能让相等的事物彼此替代。它们必须特殊对待。

有了模态算子，莱纳特和古哈极大地扩展了机器推理的复杂性。传统的问题解决程序都局限于一个或少数几个推理机[②]，而Cyc系统中则有多达30个不同的推理机。

> 大量推理机的运用让我们的程序能够处理复杂、难以理解的概念。例如，某家公司拥有一栋建筑，那么它也就拥有该建筑中的每一片砖瓦。所以程序就应当包含以下规则：“所有权”“可引申至”“物理部件”。

使用大量不同的推理方法可能会导致不一致，但莱纳特认为别无他法。

> 我们尽量站在工程学而不是科学的角度来面对它，从而避免陷入无底洞。比方说，我们不会因为要体现时代感且面面俱到，而强迫自己只追求唯一一个完美的方法。我们寻求的是一系列解决方法，即使它们加在一起也只能应对常见情况。
>
> 对此我的观念非常现实：只要某个方法有效，就用它。为了实现智能增强这一目标，我们愿意做任何事情。……
>
> 这就好像有30个木匠在争论应该使用哪种工具。他们每个人都有自己的工具，有的是锤子，有的是螺丝刀。
>
> 而答案是他们都错了，也都对了。如果我们把他们聚到一起，就能造出一栋房子来。这正是我们的行事方法。

Cyc 和批评家们

不管是什么项目，只要持续上10年，它的目标和方法都会或多或少发生一些改变。莱纳特认为这很正常，但批评家们却以此为证据，指责这是由他的“肮脏”方法论的基本弱点造成的，根本就无法与“整洁的”理论化形式相比。

---

① 模态逻辑是处理用“可能”、“或许”、“可以”、“一定”、“必然”等模态进行限定的句子的逻辑，其中利用模态判决算子来表示模态。

② 推理机（inference engine）是实施问题求解的核心执行机构，常见于专家系统。它是对知识进行解释的程序，根据知识的语义，对按一定策略找到的知识进行解释执行。

的确，莱纳特有那种名人般招徕争议的能力。有关其研究的辩论经常出现在各类学术出版物和专家会议上。主要的批评集中为两类。

(1) Cyc忽视了许多在表现不完全知识和观念等方面的理论性问题。而且在项目启动之后出现了一些新兴的、可能很有前途的形式体系，但Cyc并未加以利用。

(2) 如果Cyc失败了，很难说能从中吸取什么教训。设计者进行了很多选择，但大多数都没有明确的理论基础，所以很难说是什么导致了失败。而且，如果Cyc失败了，对于人工智能领域将会非常糟糕。人们会说人工智能只能应付鸡毛蒜皮的问题，或者范围狭窄的专业技术问题。

莱纳特回应了第一种批评，他说忽视理论性问题是因为自己不可能"缩在角落里一直等到理论家们结束辩论"，而且他也知道人工智能的这些进展，但认为那些所谓的理论都是白费功夫，毫无价值可言。

对于第二种批评，莱纳特指出，这样的指责对任何像曼哈顿计划或太空计划这样的大项目都可以这么说。他还说他从没想过这个项目会失败。

有时候批评家也有同情心。一位评论者提出，这样大规模的研究需要敢于冒险，甚至是作出错误的决定。其有利的一面在于，在Cyc研究中被迫发展出的许多技术对任何同等规模的研究都会有帮助。

我们不知道莱纳特对有关自己的学术批评是否关心，至少他从未表现出来过。他所关注的是为项目拉到足够的赞助，使之得以继续下去。一些大型企业向Cyc伸出了援手，每年捐赠50万美元，其中包括贝尔通信研究所（Bellcore）、苹果、柯达、数据设备公司（DEC）、美国电话电报公司、微软和区间研究公司（Interval Research）等。据莱纳特说，这些公司希望将Cyc用于一些非常实际的应用上，例如购车向导或智能表格。

> 我们可以对 Cyc 解释电子表格中行与列的内容的大概含义，例如年龄、某天购买的物品清单、年收入等等。对我们所知的所有东西，Cyc 也有同样的理解水平。
>
> 一旦表格的格式被建立起来，那么 Cyc 的数百万条规则就都能适用了。
>
> 比方说，电子表格中的某个单元格表示年收入，而我们给艾奥瓦州一位老师年收入的赋值是 25。不可能是 25 美元——25 000 美元还差不多。我们可以对彼此相关的数据库做类似这样的数据清理工作。例如一个人的记录显示她是另外两个人的母亲，但这两个人的出生日期记录却显示他们相隔一周出生。很明显 Cyc 不会接受这样的情况。

莱纳特还为Cyc设想了其他商业应用，例如照片存档搜索。

> 假设某人正在为《花花女郎》杂志写一篇文章，需要一些"身材健美且赤膊的年轻男子"的照片。其中一张候选图片的标题是"帕布罗·莫拉莱斯在 1992 年奥运会勇夺 100 米蝶泳金牌"。一般人是怎样从这个需求推导到这个图片标题的呢？你必须要知道游泳赛事的参与者一般都会待在水中，而且会穿着泳装，而如果你是男性，穿的又是现代西方式泳装，那么你就是赤膊的。而奥运会选手的身材都是健美的。这个推理过程并不烦琐。

PDG

但我们无论用什么词典都不能从“赤膊”查到“100 米蝶泳”。你要么知道这些，要么不知道。如果你知道，那么这个问题就微不足道。如果你不知道，这个问题就是不可能完成的任务。

《花花女郎》一例中的推理过程

莱纳特向我们展示了一份带有注释的文字记录，描述了Cyc是如何针对“找出一张身材健美且赤膊的年轻男子照片”的查询条件，找到标题为“帕布罗·莫拉莱斯在1992年奥运会勇夺100米蝶泳金牌”的图片的。

以下是该文字记录中的一些关键步骤，不包括Lisp代码。

(1) 一条现有的公理是，如果X人物在Y赛事中夺冠，那么X就参加了Y赛事。Cyc由此得出结论，帕布罗·莫拉莱斯参加了此次赛事。

(2) 另一条公理是，所有奥运会男子赛事都是男子参加的运动竞赛。

(3) 另一条公理是，运动竞赛的参与者都很年轻，而且有着运动员的健美体格。这是一条默认规则，和Cyc中几乎所有的公理一样。比这一条更有力（但适用的范围也更专一）的另一条公理是在奥运会赛事中尤其如此。Cyc由此得出结论，即帕布罗·莫拉莱斯是一个年轻人，而且其体格是运动员型的。

(4) 另一条公理是，游泳的男性会穿泳装短裤。Cyc由此得出结论，即在游泳比赛中，帕布罗·莫拉莱斯会穿泳装短裤。请注意这一点仍未表示他是赤膊的，这需要从下一条公理中得出。

(5) 另一条公理是，如果Cyc无法推断（猜测）某人穿了X，那么就假设他没有穿X（Cyc在此使用的是麦卡锡有关界限的概念）。Cyc由此得出，在整个上下文背景下，帕布罗·莫拉莱斯是一位年轻人、男性、身材健美而且没有穿上衣。

莱纳特还相信在未来20年Cyc的推理对于医学领域也会提供帮助。它可能会帮助老年痴呆症患者恢复以前的记忆。莱纳特的许多有关Cyc和人工智能未来的推测听起来就像是充满了他的成年人幻想的科幻小说。

我们需要知道，有很大一部分科学幻想都是对未知的空想——并非有意为之，但事实的确如此。还有一部分幻想则出于种种原因无法实现。思考它们为什么错得这么彻底，其实是很有用的。通常这都会让我思考（在人工智能领域）我们应该怎样去做一些不同的事情。

比如在很多故事中，远在机器智能出现之前，机器就能理解自然语言了。故事中机器理解语言命令要比拥有常识和智慧容易得多。如果我们思考为什么这是错误的，就会认识到智慧和常识可能是理解自然语言的必备条件。然后你会找到很多证实这一点的例子。只有你真正对这个世界有了极为充分的了解，才有可能做到正确地把 A 翻译成 B，以及消除句子中词汇的各种歧义。

市面上有很多科幻的角色扮演游戏，例如“龙与地下城”系统[1]。如果你翻阅这些游戏系统的参考手册，就能得到大量有关常识的洞见。他们必须为每一件事都建立规则。

莱纳特也从数年来的多位导师那里获得了灵感。其中一位是诺贝尔奖得主、物理学家理查德·费曼，他毫不矫揉造作的一颗童心已成为一代科学家的榜样。

我依然认为，费曼物理学讲义是非小说体写作的一座丰碑。从认识的第一天起，我们之间的相处就极为融洽。他对人非常随性，而且极富幽默感。如果他认为某人做某事是为了显示权力或权威，就会走过去捏他的鼻子。

我是 1983 年在思维机器公司认识他的。我们一直保持联系，但只见过几次面。我们会为一些稀奇古怪的事情打赌——如果你种了一棵橡树，能否仅通过调整灌溉量将其变成一株橡木灌木？他对我的研究真的很感兴趣，而我也很喜欢他做的事情。

在很大程度上，莱纳特也受到了麻省理工的马文·明斯基的影响，反过来也是如此。（在明斯基与科幻作家哈里·哈里森[2]合著的一本未来主义小说《图灵选择》[3]中，Cyc已经成为了现实。）

尽管我和明斯基的研究方向不同，但我们都对彼此所做的事情很感兴趣。我俩都有点担心对方比我们所想像的更正确一些。这也就表示我们自己的那部分工作是错误的。

和明斯基一样，莱纳特深信随着人工智能的发展，人类将从根本上扭转对自身的认知。

直到现在，人类的生命都是线性有限的。我们在一个地方存在一段时间，生命的结束就是死亡。而这一切可能会发生改变。想像一下每个人在网络空间中都有多个智能主体可以作为替身，可以去做我们喜欢的任何事情，而真实的意识则停留在一个肉体中。只要通信带宽足够大，在任何时候我们都可以将自己的精神迁徙到这些另外的主体中去。一个人死亡后，他或她的某些主体还会继续存在。弗瑞德死了，但还有一些弗瑞德在我们身边。

如果他憧憬的这种网络空间天堂成为了现实，那么一个莱纳特的实体可能会在大堡礁玩潜水

---

① 龙与地下城（Dungeons & Dragons，简称D&D）于1974年发明，是世界上第一款商业化的、同时也是最成熟、最具影响力的桌上角色扮演游戏（Tabletop Role Playing Game，简称TRPG）。玩家在游戏中可以做几乎任何事情、进行任何选择，但其利害需要根据游戏规则指南进行判定。这就要求游戏的系统设定极为庞大，不仅得涵盖战斗、进食、睡眠等所有行为，还得为地形、生物、气候、历史、政治、因果关系等作详细的描述，可谓是包罗万象。

② 哈里·哈里森（Harry Harrison，1925—）是美国著名科幻作家，世界科幻小说协会首任主席。著名作品有《死亡世界》系列、《不锈钢老鼠》系列、《伊甸园三部曲》等。

③《图灵选择》（*The Turing Option*）于1993年出版。这部作品属于赛博朋克（cyberpunk）类型。赛博朋克是科幻小说的一个分支，以计算机或信息技术为主题，小说中通常有社会秩序受破坏的情节。

或滑翔伞（真实的行为），而另一个莱纳特则在翻阅一本艾萨克·阿西莫夫的书，查阅盖娅行星[1]的重要数据。另外几个莱纳特可能在玩着桌上游戏。

而对于触手可及的将来，莱纳特相信类似Cyc这样的人工智能项目会成为一种“知识公共设施”。普通人可以在家中、办公室或学校通过个人计算机访问Cyc这样的程序。智能家庭购物程序会帮助消费者甄别促销卖场或者挑选电影，而这些都基于一个不断进化的计算机资料库。（“有关信息的信息”是一个公认有利可图的产业。例如《电视指南》[2]杂志就是一个盈利极为可观的信息网。）了解了消费者的偏好，公司也就多了一种渠道来监控反常的信用卡消费，例如信用卡盗用等。如果这听起来有些令人不安，莱纳特却并不害怕。

> 现在是时候开始在Cyc的基础上构建机器学习了。它将和人类一样富有持续的创造性。我在这片荒野中近10年的跋涉也将走到尽头。我期待多花一些时间在机器学习上。这也是我中断10年的原因。如果它成功了，大概再过10年，我将再次回到通用智能增强的研究上来。而到那时候，我说不定会伴随着智能增强再次进入物理或数学领域。最终，到65岁时，我会再重返21岁时搁置的计划上面。

Cyc会成功吗？最可能的情况是它会成功，但只是部分地。Cyc可以充当数据库的清洁工，消除那些明显荒唐的错误。它还能帮助消费者过滤那些无用的广告：如果爱丽丝刚摔伤了腿，她可能不会有兴致购买跑鞋。但Cyc可能永远无法进行有说服力的交谈。

长远来看，Cyc的机制（微理论、可交互的推理机、发现新启发式的机制，以及对人工智能需要大量知识的认知）将会永远地改变人工智能领域的研究。如果Cyc成功了，即使只是部分成功，莱纳特就将证明人工智能不仅需要绝佳的头脑，还需要勤勉的汗水。相对于知识表示的一条新理论而言，成千上万有关我们这个世界的细微事实更加重要。

为什么写莱纳特？因为智能研究亟需像他这样大胆的探险家。

---

① 盖娅行星（Gaia）在阿西莫夫的小说《基地边缘》中登场。星球上所有的动植物和矿石都分享着一个整体意识，构成一个超级心灵，合力为大我奉献。阿西莫夫企图藉由盖娅行星，探索集体意识的可能性。

② 《电视指南》（*TV Guide*）是全美销量前列的杂志之一，创刊于1953年。主要内容包括电视节目表、影视动态、名人访谈、八卦流言和电影评论等。

后　　记
下一个25年[1]

这一领域仍旧处于胚胎阶段。它还没有2000年的历史，这很好。非常非常重要的成果就在我们眼前发生着。

——迈克尔·O.拉宾

无论你是一位计算机科学家还是计算机用户，本书中所谈及的这些大师已经改变了你的生活。他们的成就推动着这一领域中最重大的发展。

50年前，最优秀的工程师和物理学家所面临的难题仅仅是如何让硬件设备能够运转起来。

40年前，本书中的部分科学家开始了他们的职业生涯，面临的挑战变成了如何让编程更加容易。巴科斯和麦卡锡等研究者们设计的语言不仅适用于科学家，而且也适用于商业应用及人工智能。

30年前，布鲁克斯、戴克斯彻等操作系统设计者向人们展示了一台计算机可以同时执行多个任务，而程序员亦无需了解这些任务之间的相互影响。高德纳又通过编译器进一步简化了编程语言的设计。费根鲍姆的专家系统则证明了人工智能的实用价值。

20年前，高德纳、拉宾、陶尔扬、古克和列文向人们展示了怎样精确地分析算法。凯让程序员在设计软件时考虑使用“对象”，在很大程度上消除了为每种新的应用去发明新的专用语言的需求。兰伯特则提供了一种有关分布式系统的思考框架。

10年前，人工智能引起了全球性的关注，莱纳特开始了Cyc项目。微处理器的广泛适用性促使伯顿·史密斯和希利斯开始研究两种完全不同的大规模并行设计方法。如今布鲁克斯等人利用并行计算机来解决各种问题，从模拟分子到创建虚拟厨房。

回首往事，我们不难发现研究重心由机器转向问题本身的趋势。如今的挑战已不再是设计处理器，而是设计能将各个处理器有效连接到一起的网络。而更大的挑战则是通过软件设计，让并行处理器能作为一个既快又稳定的整体运行。尽管人工智能还远未实现其宏伟的目标，但已获得了部分的成功，并衍生出了一批新的产业：图形学、可视化、虚拟现实等方面的新算法，正在帮

[1] 本书出版于1998年。众所周知，计算机领域的发展极为迅速，因此在2010年后的今天翻译本书的过程中，译者在一些细节方面都进行了相应的修改。但这篇后记有意保留了作者在当时的观点，读者不难发现其中的某些预测如今已经成为了事实。

计算机为人类实现智能增强。随着计算机越来越不可或缺，它们也越来越趋于“无形”：汽车生产厂家已经将计算机网络直接植入车辆内部，很快洗衣机就将能和个人计算机进行对话。

在未来25年，当计算无处不在时，计算机科学家们又会扮演何种角色呢？在此我们大胆作出一些预测。

- 计算机要想创新，算法就是食谱。因此算法依然是这个领域的重中之重。

  古克和列文所提出的P与NP问题将会得到解决。如果P＝NP，那么公钥加密法就会遭到重创，但许多有用的组合论问题将会用算法得到解答。如果P与NP未被解决，那么NP大于P的假设将会产生出新的启发式算法，也许会基于希利斯的“进化”计算模式。

  无论P与NP的问题如何解决，如果随机化方法能够解决因式分解问题，那么就将出现新的密码协议。

  跨学科的算法将会出现，广泛应用于光通信、基因设计、虚拟世界、机器人手术等大量新的领域。

- 新的编程语言就像照片中的面孔一样，总是会引起人们的注意。但和精彩的摄影作品一样，创新且有影响力的语言并不多见。

  Smalltalk、C++和Eiffel这样的面向对象的编程语言将会超过单纯的函数式语言，因为面向对象的范式模拟了产品的自然演化：如果你用组件来构造产品，那么产品的下一版本就可以利用一些现有的组件，并替换一些选择性的组件。另一方面，由巴科斯的FP所衍生出的函数式语言也会在需要高可靠性的领域中得到应用。

  有些专用语言可以融合各种以不同语言编写的组件，这样的专用语言也会流行起来。

- 在计算机科学和机器人学中，设计到制造之间的周期将会由数月缩减为数天。从电子设计到金属和塑料成品，这之间的转换将会像发送传真一样简单。
  19世纪中叶，牛顿力学的原理得到了广泛的理解与运用，工程师和企业家们每年都会带来许多创新。我们发现如今的计算机技术也处于同样的阶段。

  所有的计算机都会连接到一起，可以进行大范围的信息搜索，但同时知识产权的侵权行为也会更加猖獗。要在鼓励自由通信的同时确保隐私和数据的安全，我们对计算机设计（和法律）的认识也要不断更新。

  由于软件可以用于屏蔽硬件故障，可靠性将成为一个极为突出的问题。这也使得兰伯特、戴克斯彻、列文和古克在后期对形式化验证的研究显得至关重要。

  正如在飞机机翼设计中计算机很大程度上取代了风洞一样，计算机也将取代自然科学甚至社会科学中的许多试验。这里面存在的风险是，试验将不再反映真实数据，而只是编程人员的假设。

  能自我繁殖的太空用机器人工厂将会设计出来，甚至在地球上进行测试。

虚拟现实将会帮助设计师测试他们的想法，但主要市场依然会是娱乐及医疗领域。

基因序列方面的计算将会得到应用，尤其是在食品工业。

- 人工智能将继续不断衍生有价值的应用以及各种各样的麻烦，因为人工智能研究人员愿意去冒更大的风险。

  数据库系统将会吸收更多来自莱纳特的Cyc及其继任者的想法，以及它们的知识库。

  人工智能将会在系统中融入虚拟现实和神经生理学，人们可以直接以脑电波来控制周遭的环境。

教育将会反映出不断增长的计算智商。教师将不再操劳于教授计算、记忆等技能，计算机在这些方面比人类做得好的多。在最好的学校里，学生们将会利用网络上提供的信息库和专家系统，以团队的形式解决问题。阅读、逻辑推理和团队合作技能将成为这种自由式教育的基础。至于那些尚未被编码的专业化知识，少数有快速领悟力的人会去学习某个领域，然后将自己掌握的专业知识有偿地输入进专家系统，然后继续学习下一个领域。

我们不禁幻想，在25年后这样一本书中又会描述什么样的英雄人物。他们的肉体也许仍将由碳构成，但他们的思维却会插上计算机的翅膀。他们会利用计算机来增强自己的智能，同时发挥人类独有的创造天赋。祝他们一路顺风！

结　　语
成功的秘密

什么样的性格和智力造就了伟大的计算机科学家？回看书中访谈的这15位大师，我们得出的结论是，他们彼此之间的不同点要比相似之处多得多。

例如，他们的天赋都是在学校里展现出来的吗？有些人是。列文十多岁就在基辅市物理奥林匹克竞赛中夺冠。但巴科斯回忆他高中时每年都不及格，父母把他送到暑期补习班，结果他却把时间都花在帆船上。

艾伦·凯在学校的悲惨遭遇反而给了他很多灵感。他成年后花了很多时间琢磨如何用计算机来帮助孩子们学习。伯顿·史密斯，Tera机的设计者，在初中科学课上设计的装置居然是用来向同学弹射纸团。12岁的迈克尔·拉宾由于恶作剧被赶出课堂，却由此开始了他的数学生涯："两个九年级的学生正坐在走廊里解欧氏几何题……他们有个问题解不出来，就以此挑衅我，结果我做了出来。"

这些科学家中有三位——凯、高德纳和兰伯特——都差点成了职业音乐家。其余大多数对音乐也很热衷。能让数百台计算机一起工作的伯顿·史密斯在合唱团里演唱文艺复兴时期的复调歌曲。但同是计算机架构师的丹尼尔·希利斯却更喜欢诗歌和韵律，而不是音乐。拉宾则声称音乐太难了——不像数学和物理那么简单。

我们都知道，数学家往往年轻时有丰饶的创造力，而到中年后便会停滞。但计算机科学家似乎并不是这样。拉宾在40岁后发明了随机化算法，麦卡锡在50岁后发明了非单调逻辑，巴科斯年过60才开始研究函数式语言，戴克斯彻也是在同样年龄提出了数学验证的新方法。

另一方面，早年的激情通常也会反映在后期的成就上。丹尼尔·希利斯童年时在试管中培育了一颗跳动的青蛙心脏，十几岁时又对神经解剖学入了迷。这些兴趣很自然地引发了他设计大量并行连结机器以及后期研究人工生命，让计算机去模拟生物的进化。

有些计算机科学家很注重培养自己的灵性与精神。弗雷德里克·布鲁克斯毅然放弃了在IBM的大好前程（他曾是大获成功的360系统的项目经理），只身前往北卡罗来纳大学创建计算机科学系，并将此归因于他虔诚的基督教信仰。随后他又带头将虚拟现实带入化学、建筑学和医学领域，而且从道德上反对将虚拟现实主要用于娱乐。约翰·巴科斯在退休后喜欢阅读伊娃·皮拉卡斯的神秘著作。他说："通过对自己的反省，你可以真正理解宇宙的奥秘……这是通过任何物理法则

都做不到的。"

其他的大多数科学家都并不信奉自觉与灵性，但他们也有顿悟的时刻。戴克斯彻是在一个周六早晨和妻子坐在咖啡馆里的时候发现最短路径算法的。他突然之间安静下来，意识到自己找到了解决办法。（他也提到妻子对他这种突发性沉默的反应："我的妻子对这种情况见怪不怪了。"）伯顿·史密斯则是在飞机上呒着威士忌时想到了一种并行的硬件设计方法。

中世纪的科学家向国王和王子们寻求资助，而计算机科学家则依赖于企业、政府和大学的支持。IBM赞助了巴科斯和拉宾，施乐和苹果对凯提供了支持,莱纳特的研究资金则来自于工业财团MCC。而美国国防部ARPA（高级研究规划署）和美国国家科学基金会则赞助了他们所有人。

赞助的类型几乎和金额一样重要。麦卡锡和明斯基在走廊上与实验室主任杰里·威斯纳（Jerry Wiesner）随便聊了几句，结果得以设立11人的人工智能实验室。麦卡锡说道："我想这种灵活的资源调配正是美国的人工智能研究起步领先于其他国家的原因之一。"在随后的一年半时间里，这个团队开发出了Lisp，人工智能领域编程语言的佼佼者，所有的现代语言都借鉴了它的思想。"我们资助的是人而非项目"，这种开放式的拨款使得像Lisp、连结机器、Smalltalk和随机化算法这些存在争议的想法有了实现的机会。艾伦·凯回忆道，施乐公司成立帕洛阿尔托研究中心的目的非常不明确："我记得的唯一一句话是：如果有人能让办公室不再依赖纸张，那肯定是我们。"

很少有计算机科学家埋头单干。首先，他们都在世界知名的学府待过，要么是去进修，要么是当教授。几乎所有北美计算机科学家的传记中都出现了麻省理工、斯坦福、卡内基梅隆、哈佛、普林斯顿这些字眼。

其次，计算机科学家往往会以小团队的形式合作：罗伯特·陶尔扬和约翰·霍普克洛夫特，迈克尔·拉宾和达纳·斯科特，高德纳和艾兹赫尔·戴克斯彻，艾伦·凯、丹·英格斯和阿黛尔·戈德堡。拉宾与斯科特的合作方式可能是许多计算机科学家所向往的："我们俩先有一个人会提出一个问题，然后分别回到自己的角落，另一个人很快就会提出解决方案，也许只需要一晚上的时间。"但也有少数人就是喜欢独善其身。戴克斯彻一生潜心钻研许多基础性问题，无视当时业界那些华而不实的潮流，而他对此形容为"我孤僻，但我快乐"。兰伯特认为自己如果在学术界随波逐流，很可能就不会取得分布式计算的成果。"我完全不需要担心职位的问题。我有着充分的自由去研究自己感兴趣的东西，而不是其他人感兴趣的东西。"但孤立也会带来不利之处。由于铁幕的遮挡，利奥尼德·列文的赫赫声名在西方却默默无闻，而且他连古克和卡普已经作出了类似的发现也不知道。

这些计算机科学家们多数都曾对物理有所涉猎。兰伯特关于分布式系统的研究直接受到了他痴迷10年的狭义相对论的启发。有4位科学家都与诺贝尔奖获得者、物理学家理查德·费曼私交甚笃。莱纳特回忆道："我们会为一些稀奇古怪的事情打赌——如果你种了一棵橡树，能否仅通过调整灌溉量将其变成一株橡木灌木？"费曼认为丹尼尔·希利斯将100万个处理器进行连结、设计出类似人脑的机器的想法非常"愚蠢"，但仍然加盟思维机器公司去协助他的工作。费曼天性幽默、极富创造力，而且在任何时候都乐于尝试任何创意，而这些都是计算机科学家们希望具备的性格。

这些科学家们对生物学的普遍兴趣令人惊异。这两个领域的气质看上去迥然不同。生物学家的试验对象是湿性的有机材料，而且不相信精心设计的理论；而计算机科学家则使用防腐的硅片、金属/塑料的雕刻品，他们的研究依赖于大量的数学及语言学理论。但希利斯和凯对生物学的研究都达到了大学水准。即使是列文和拉宾这样在物理和数学中成长起来的理论计算机科学家，也认为计算机智能的下一个重大进展将来自于生物学。正如列文所指出的："从我们还生活在树上时开始，遗传学的历史其实非常短暂。不过我们开发用于摘香蕉的算法也同样有利于利用核能。"

如果讨论这些科学家为什么能如此杰出，最后总会归结到运气因素。这些人中谁也没有中过彩票，但他们确实都获得了幸运之神的垂青，在适当的时间研究了适当的问题。巴科斯在25岁结束学业、对未来一筹莫展时，参观了IBM公司总部并碰巧得到了一份工作。在机器语言编程上的不顺利迫使他和几位同事思考一种更好的办法。麦卡锡因为用Fortran解决人工智能问题时遇到了困难，于是不得不发明了Lisp。费根鲍姆则是受到了艾伦·纽厄尔和赫伯特·西蒙的熏陶。而艾伦·凯刚当上研究生，第一项任务便是让Simula语言运行。

这些科学家对自己为什么成功的解释各有不同，但也有一些共同点。首先，你选择的问题决定了你的解决方法的重要程度。弗雷德里克·布鲁克斯寻求的是其他学科的人们提出的问题，例如化学家、医生和建筑师。他称其为"有生命力的问题"。正如他所言："如果有人对你说：'你做的东西很不错，但它对我一点帮助都没有。'这其实是在提醒你要脚踏实地。"找到合适的问题可能需要时间。爱德华·费根鲍姆花了3年时间考虑了归纳中的很多问题，包括棒球比赛规则，直到最后才决定为生物化学建立自己的第一个专家系统。

其次，有时候你也无法判断出哪些问题是真正重要的。古克也未料到NP完全问题的广泛适用性。拉宾和斯科特略去了有关非确定性问题的一些结论，因为他们不希望论文太长。据陶尔扬自称，刚开始研究平面性检验时他还只是一个摸不着头脑的研究生。巴科斯曾以为Fortran只可能适用于IBM的一款机型。

第三，你必须是解决该问题的合适人选。正如高德纳所说："我们常说需要乃发明之母，这句话并不确切。一个人还得拥有该领域的背景知识。我并不是随走随看，研究自己看到的每一个问题。我解决的那些问题，都是因为我正好有独特的背景知识，也许会帮助我解决它——这是我的命运，我的责任。"

第四，重大的成就需要甘于冒险，甚至勇于面对嘲笑。麦卡锡第一篇严肃的人工智能论文被一位著名数学家贬斥为"半生不熟"。拉宾的大多数同事都认为他的随机化方法是只适用于极少数问题的侥幸成功。巴科斯的简化编程的提议不得不面对来自知名物理学家、数学家和计算机设计者约翰·冯·诺依曼的反对，后者认为机器语言编程已经足够简单了。巴科斯在1978年提出了以"函数式"编程语言替代Fortran类语言的建议，似乎从未被人接受过。莱纳特的Cyc项目同样如此，他说："作为研究人员，我们其实就是在拿自己生命中的三十年进行赌博。"

最后，计算机科学家是怎样看待他们自己的学科的？它和其他学科有何不同？艾伦·凯这样说道："在自然科学中，是大自然给出一个世界，而我们去探索其中的法则。对于计算机来说，却是我们自己来构建法则，创造一个世界。"

术语词汇表

注：楷体的词另有单独词条予以解释。

访问（access） 对计算机存储器进行*读*或*写*操作。

算法（algorithm） 说明某问题应如何解决的一系列指令。例如，排序算法必须能处理大量的名称，并按字母顺序将它们排序。算法通常用英语进行描述，使用数学符号，而且非常精确，有经验的程序员能将其翻译成计算机*程序*。

分摊（amortization） 罗伯特·陶尔扬和丹·史利特发明的一种效率分析方法，用于衡量计算“在最坏情况下”的平均成本。假设现在有一个104人的团体，而你希望知道其中有多少人的姓以A开头、多少人以B开头，一直到Z。你请人们在听到他们的字母时举手。根据举手人数来衡量成本。类似S这样的字母成本会比较高，而类似Q这样的字母成本则较低。但是，每个字母分摊的成本是104/26＝4，因为总成本是104，然后除以字母总数26。

ARPA 高级研究规划署（Advanced Research Projects Agency）。美国国防部下属的研究经费拨款部门。本书提及的多数大型项目都接受过来自ARPA或DARPA（国防高级研究规划署，ARPA在20世纪70年代到90年代初的名称）的资助。ARPA也资助了阿帕网（ARPANet）的研发，它就是我们今天的互联网的雏形。

人工智能（artificial intelligence，AI） 计算机科学的分支学科，植根于哲学、语言学和心理学。其目标是让计算机模拟人类的智能活动，例如运用专家知识、汲取经验或像棋手那样进行推理。

人工生命（artificial life） 涉及物理学、生物学和计算机科学的一门交叉学科，其目标是利用计算机模拟考察进化、竞争等生物过程，以及生命复杂性的本质。一些研究表明人工生命有可能导致新的计算范式：提出问题，创造人工生命细胞，奖励那些提出更好解决办法的细胞，并让细胞去寻找好的解答。

汇编语言（assembly language） 用符号表现的*机器语言*。例如，要在某些英特尔处理器中把寄存器AL的内容复制到寄存器CH，那么用汇编语言就可以输入“mov AL, CH”。如果用机器语言来表示则是令人生畏的10001010 11000101。

原子性（atomicity） 对于操作O来说，如果任何其他操作只能在O开始前或结束后才能读取寄存器X和Y的值，那么操作O就是原子级的（即不可分割的）。在日常生活中也需要这一特性。例如，鲍勃和爱丽丝同时把钱存入他们的共同账户，以下的事件发生顺序将会导致其中一人的存

款丢失：鲍勃的程序读取余额、爱丽丝的程序读取余额、爱丽丝的程序将存款值加入余额、鲍勃的程序将他的存款值加入该程序之前读取的余额。这会导致爱丽丝的存款记录丢失。然而，如果每一次存款操作都是原子级的，那么鲍勃的程序就会在爱丽丝的程序开始之前执行完毕，或者爱丽丝的程序在鲍勃的之前执行完毕。原子性通常都是通过*互斥*来实现。

原子寄存器（atomic register）针对该*寄存器*的任何两个操作都是一次只能发生一个。

自动编程（automatic programming）将编程工作量减少到最小的一种技巧：只需把问题提交给计算机，然后让计算机来完成余下的工作。这一领域在20世纪60年代颇为流行，它为编程语言，尤其是现代的规范语言提供了许多新的思想。

公理（axiom）数学理论中无需证明即可给出的声明。例如欧氏几何中有一条公理称，两点之间最短的路径是直线。

批处理（batch processing）让计算机从头到尾执行一个程序，不受其他程序干扰。与之相对，在多道编程或多任务系统中，计算机先执行某个程序的一部分，然后再执行另一个程序的一部分，依此类推。批处理是计算机最早使用的一种方法，如今在很大程度上已被*多道编程*所替代。

位（bit）取值为0或1的数字。8位构成一个*字节*。

布尔代数（Boolean algebra）由英国逻辑学家乔治·布尔于19世纪50年代发明的一种代数形式，可以通过计算进行命题逻辑的推理。真表示为1、假表示为0，逻辑“与”表示为乘法，逻辑“或”表示为加法（在布尔代数中$1+1=1$）。如果我们知道“X为真且Y为假”，那么“X且Y”就为假，因为“真且假”为假。这在布尔代数中表示为$1\times 0=0$。近一个世纪之后，信息论的发明人克劳德·香农证明，布尔代数是分析数字开关电路的极佳工具。

广度优先搜索（breadth-first search）搜索图问题所用的一种技巧，类似我们日常生活中在书里查找内容时所用到的这样一种方法：先查看目录中所有章的标题；如果没有找到想要的内容，再查看所有节的标题；如果还是没有找到，则查看正文。与之相对的是*深度优先搜索*。

缓冲区（buffer）处理器中内存的一个部分，用于在多个速度不同的进程之间进行通信。在日常生活中，邮箱就类似于缓冲区。我们在不同时间都可以把信投入邮箱，邮递员再把它们集中拿走。如果没有邮箱作为缓冲区，那么我们就必须站在那里等邮递员，然后当面把信交给他。用计算机科学的术语来说，把数据放入缓冲区的人被称为生产者，将数据取走的人被称为消费者。

字节（byte）按一定顺序排列的8个0和1。大体上，一个字节对应于文本文件中的一个字母。一个“兆字节”（MB）表示100万字节。

高速缓冲存储器（cache）容量较小、速度较快的存储器（通常能在两亿分之一秒内存取100 000字节），用于提高计算机的性能。最近被*访问*的数据会被从较大的主存储器复制到高速缓存中，替换掉最近最久未被访问的数据。它有些类似人类的短期记忆。

高速缓存一致性（cache coherency）在多处理器计算机中，多个高速缓存可能会持有同一个数据的多个副本。如果某个处理器修改了数据，其他所有高速缓存都必须立刻更新数据，以确保彼此相等，也就是一致性。

组合数学（combinatorics）涉及有限集合及其关系的数学分支。给定从克利夫兰到芝加哥各条路线的行车时间，找出两地之间最快路线的问题就是一个组合数学问题。根据已给出的棋局，

找出跳棋的获胜策略也是组合数学问题。

编译器/翻译器（compiler/interpreter）将Fortran、C++、SQL、Excel、Photoshop、WordPerfect等高级语言翻译成机器语言（处理器语言）的程序。编译器和翻译器让我们能够用人类易于理解的语言编写程序，而不需要背后的处理器也理解这种语言。编译器生成代码比翻译器快，因为它在编程的同时就开始进行翻译了。然而翻译器更加灵活，因为它们是在*执行*的时候进行翻译。

并发（concurrency）不同行为在同一时间进行重叠的状态。在家读小说和写报告有可能是并发的。也就是说，某个行为开始和结束之间的时间区间与另一个行为发生的时间相重叠。

合取（conjunction） 由逻辑“与”连接的一组断言。例如：玛丽有一件雨衣且玛丽有一把雨伞。

消费者（consumer） 见*缓冲区*。

上下文无关文法（context-free grammar） 最早由语言学家诺姆·乔姆斯基提出的一种文法类型。文法中的规则如“句子由名词短语及跟随其后的动词短语构成”可以表示为“句子 → 名词短语 动词短语”（S → NP VP）。另一条规则可能是“名词短语由冠词及跟随其后的零到多个形容词及跟随其后的名词构成”。

临界区（critical section） 程序中的一段，访问由多个进程共享的数据。*互斥*算法必须确保每次只有一个*进程*操作其临界区。

密码技术（cryptography）字面上的意思是隐藏或秘密的文字，该领域关注的是只有发送方和接收方才可以辨认的信息传递。典型的方式是由发送方以某种编码加密信息，然后接收方以相关方式对信息解码。在私钥密码技术中，接收方必须知道发送方的加密编码，这导致接收方有可能会乔装为发送者。在公钥密码技术中，接收方不知道发送方的加密编码，因此也无法乔装。目前使用的公钥密码技术依赖于一个事实，即将两个素数相乘比求一个很大数的素数因子要容易得多。如果有人能找到求一个很大数的素数因子的方法，那么这种密码技术就会失效。

DARPA 国防高级研究规划署（Defense Advanced Research Projects Agency）。参见*APRA*。

数据库（data base或database） 用于支持某种人类活动的大量数据集合。例如，银行有账户记录数据库，音像店有CD唱片或磁带的数据库。直到20世纪80年代末，大型数据库仍存储在大型机上，但基于微型计算机的并行数据库应该会在日后占据主流。

数据清理（data cleaning） 纠正大型*数据库*中的错误。

数据流体系架构（dataflow architecture）一种计算机设计方法，支持局部并行的计算形式。大多数程序都严格按照程序文本规定的顺序进行操作，而在数据流程序中，当某个操作所需的数据被计算出来时，该操作就可以进行。常规的程序类似我们拼装飞机模型时读的那种参考说明：第一步、摆放好各个部件，第二步、粘连机身，等等。这种说明往往要求严格地按部就班完成，但其实有时不必要。数据流程序则类似同时烹调多道菜式的专业厨师所遵循的步骤：等水开后就放入蔬菜，等奶油搅拌好后就淋到糕点上，等等。

数据结构（data structure）为了便于快速访问而设计的信息存储方式。字典与此有些类似：按字母进行排序就是为了便于快速查找单词。持久性数据结构还可以像检索当前数据一样快速检索过往数据。例如，你不仅能查询自己当前知道的每一个姓名和地址，还能查询5年前的那些。

演绎（deduction） 一种内部一致的推理形式，通常都是有效的。哲学家查尔斯·桑德斯·皮尔士给出了如下例子来说明演绎：如果我知道这个袋子中所有的豆子都是白色的，而有人从中拿出来一颗豆子，那么我就知道它是白色的。与之相对的是*归纳*。

深度优先搜索（depth-first search）搜索图问题所用的一种技巧，类似我们日常生活中在书里查找内容时所用到的这样一种方法：先查看第1章第一节的内容，然后查看第1章第二节，依此类推直到找到想要的内容。与之相对的是*广度优先搜索*。

确定性（determinism） 根据计算过往的状态以及当前访问的数据，能精确地预计计算的下一个状态（state）。与之相反，非确定性计算在同等的情形下可能会进入多个状态中的任何一个。在时间T之内非确定性地“解决”一个问题，意味着在T步骤后可能的状态之一是正确的状态。迈克尔·拉宾和达纳·斯科特证明，任何非确定性程序都可以转换为确定性程序，但相比之下非确定性程序的编写要容易得多。

哲学家进餐问题（dining philosophers problem） 由艾兹赫尔·戴克斯彻提出、安东尼·霍尔命名的问题，用于说明计算机系统中访问共享资源的原则和面临的挑战。一群哲学家围坐在圆桌周围，每人面前有一碟意大利面，相邻的每对碟子之间有一根筷子。因此每位哲学家左手边的筷子也就是坐在他左边的人右手边的筷子。每位哲学家思考一段时间，进餐一段时间，然后继续思考，如此往复。要进餐就必须使用两根筷子。问题是：哲学家们应该遵循何种程序才能让每一个人都吃到东西？这个简单的问题及其上百种变体在多道程序系统中用于解释*互斥*技巧。

消除歧义（disambiguate） 解决同一语句多种解释的问题。例如，如果你看到报纸上的标题“Stolen Painting Found by Tree”（失窃画作在树边找到），就会联想到画被找到时旁边有一棵树。但如果标题是“Stolen Painting Found by Detective”（失窃画作被侦探找到），你就会联想到机警的侦探角色。消除句子的歧义需要常识和对世界的认知。

析取（disjunction） 由逻辑“或”连接的一组断言。例如：玛丽有一件雨衣或玛丽有一把雨伞。

分布式算法（distributed algorithm） 在由网络连接的一系列处理器上运行的一种*算法*。

DO语句（DO statements） 在Fortran语言中用于执行*循环*的语句。

欧拉环游（Euler tour）在一个图中由某个节点出发，经过所有边然后再回到出发节点的路线。

执行（execution） 某段程序的运行。如果把程序比作菜谱，那么执行程序就相当于按菜谱烹饪。

专家系统（expert system）模拟人类专业知识的计算机程序。这里的专业知识定义很广泛。例如专家系统可以针对某种病征给出医疗建议，也可以给出如何将钻头从钻洞中取出的建议。

指数时间（exponential time） 一段过长的时间。精确的描述是，如果解决大小为N的问题时，某个*算法*需要的时间与2^N成正比，那么该算法就需要指数时间。如果针对某问题可能的最佳算法需要指数时间，那么该问题的大小只是增加1时，所需的时间就会翻一倍，甚至更多。例如要列举N个字母的所有可能组合，当N为1时，可能的组合方式有26种。如果N为2，可能组合则有26×26种，依此类推。一旦N超过10，那么就算是我们能想象出的最快的计算机对此也束手无策。

与之相对的是*多项式时间算法*。

因式分解（factoring）找到一个数的素数因子（素数是只能被它本身和1整除的数）。例如221的素数因子是13和17。到目前为止，公众认为（某些安全机构可能知道得更多）因式分解需要指数时间，与被分解的数所包含的数字位数有关（221是3位十进制数，或8位二进制数）。如果因式分解很容易，那么就算是最优秀的公钥*密码技术*也会很容易遭到破解。

容错（failure tolerance） 对于计算机网络等机制来说，这是一种当某些组件出错后仍然能够正常运行的能力。

费马大定理（Fermat's last theorem） 由费马写于一份手稿边缘空白处的一个猜想：如果$n \geq 3$，那么方程$a^n + b^n = c^n$无解。

有限状态自动机（finite sate automaton，复数为automata） 一种数学模型，用于描述包含有限个“状态”（state）及状态转换的机器。以字母标记每一种状态转换。除了必须有一个初始态和一个以上的接受态之外，状态可以是任意的。如果一个字母序列输入到有限状态自动机中后，自动机停止在接受态，那么可称该自动机接受了输入。有限状态机有一个很好的例子就是密码锁，不过是拨动数字或按下按钮，而非输入数字序列。接受态即对应于“开锁”。在确定性有限状态自动机中，对应每一个字母只有一种状态转换。而在非确定性有限状态自动机中，对应每一个字母可以有多种状态转换。非确定性有限状态自动机易于编程，而且可以转换为确定性有限状态自动机。参见*确定性*。

定点（fixed point） 计算机中带有固定长度的数，目前通常相当于16位十进制数字。20世纪50年代初的定点计算机用这16个数字来表示所有的数。程序员必须记住在某个位置的25表示2.5，另一个位置的25则表示25 000 000 000 000 000。除了约翰·冯·诺依曼，几乎没有人会觉得这很直观。参见*浮点*。

浮点（floating point） 为了解决*定点*计算的问题，硬件设计者开始在计算机中储存指数。因此我们可以用25 E -1来表现2.5，用25 E 5来表现2 500 000。约翰·巴科斯对于计算机科学的第一大贡献就是在定点计算机上设计了模拟浮点运算的软件。

Fortran 巴科斯领导的IBM团队在20世纪50年代中期开发的计算机语言。它是第一款被广泛应用的计算机语言，至今依然是许多科学应用首选的计算机语言之一。

函数式编程（functional programming） 一种编程类型，其结果系由函数计算得出，并没有明显的修改存储器中数据的概念。电子表格有些类似函数式程序，其中的每个单元格要么是一个数，要么是一个涉及其他单元格的函数。约翰·巴科斯设计了一款函数式编程语言，并称之为FP。

文法（grammar） 对一种语言的语法或能被接受的形式的描述。语言学家诺姆·乔姆斯基提出，英文句子“Colorless green ideas sleep furiously”（无色的绿色思想狂暴地睡觉）在语法上是正确的，但含义却完全莫名其妙。计算机语言通常都由一种叫做巴科斯-诺尔范式的语法所描述，其在本质上与诺姆·乔姆斯基的*上下文无关文法*概念相同。这类文法过于羸弱，无法用于描述人类语言。

图（graph） 在组合数学和计算机科学中，特指一组节点（也称结点或顶点）和连接各节点的边（也称弧或链）。图可以表现无限种网络，包括由道路连接的城市网络，或者连接石油钻探

区和储油区的管道网络。

停机问题（halting problem）判断计算机X是否能确定另一台针对数据D运行程序P的计算机Y是否会停机的问题。阿兰·图灵在描述了通用的计算设备即图灵机之后，证明即使是他的设备，在任何情况下都不能保证解决停机问题。也就是说，必定存在Y、P和D这样的实例，而X无法确定其正确的答案。事实上，这只是无限多个“不可决定的”问题中的一个，而到目前为止，这类“不可决定的”问题只发现了数百个。

硬件（hardware） 构成计算机系统的处理器、硬盘、打印机、显示器等。如果你用力踩它们，它们就会损坏。

启发式（heuristic） 由数学家乔治·波利亚提出的术语，指一种解决问题的技巧，并不保证在所有情况下都正确，但通常都是可行的。例如在日常生活中的启发式说：如果红绿灯说可以通行，那么不需要看路就可以走。这条规则在伦敦很管用，但在纽约可能就不好说了。

归纳（induction） 由特殊到一般的推理。借用哲学家查尔斯·桑德斯·皮尔士的装豆袋子的例子：由于你每次从袋子里取出的豆子都是白色的，那么归纳得出袋子里所有的豆子都是白色的。参见*演绎*。

智能增强（intelligence amplification）在计算机的帮助下扩大人的能力。特定的专家系统可以做到这一点，它能协助普通的从业人员利用最新的医疗研究成果来诊断各种疑难杂症。虚拟现实系统也可以做到这一点，它能让建筑设计师“走”进自己刚刚设计出来的作品中观察揣摩。

中断（interrupt）来自处理器之外的设备传给处理器的信号。例如，个人计算机键盘上的每一次击键都会“中断”计算机内的处理器。中断有点类似响铃的电话。屏蔽中断（弗雷德里克·布鲁克斯的发明之一）可以让处理器在一段时间内切断这些干扰，从而避免出现电子混乱。

Lisp 由约翰·麦卡锡于20世纪50年代开发的一种符号运算语言。Lisp现在依然是人工智能的主要编程语言，而且自被发明以来，它实际上已经影响了每一种编程语言。

循环（loop）在程序中允许多次重复执行某组指令的结构。大多数工作都可以用循环进行描述，例如高速公路收费员的基本循环大概是这样的：

```
重复直到换班（repeat until your next break）
    与司机打招呼（greet driver）
    收钱（collect money）
    找钱（return change）
    如果司机需要则指路（give directions if requested）
结束重复（end repeat）
```

机器语言（machine language） 处理器用于理解某些指令集的电路。这些指令以1和0的形式编码为所谓的二进制机器语言。例如，在某些英特尔芯片中，若要将数据从寄存器AL复制到CH，就可以用机器语言输入10001010 11000101。编译器可以将Fortran、Lisp或Smalltalk等高级语言翻译成机器语言。参见*编译器/翻译器*。

大型机（mainframe）大型企业使用的价值数百万美元的计算机系统。IBM在1990年之前一直占据这一市场，随后通过网络连接的大量微处理器开始盛行。

存储器（memory）计算机中储存数据的地方。存储器分为多种层级：寄存器速度最快，但容量最小（访问时间为几纳秒，但只能储存1000字节，甚至更少）；高速缓冲存储器速度相近，但容量较大（约数十万字节）；主存储器，也称随机存取存储器（RAM）容量更大，但速度也更慢（访问时间100纳秒，可储存数百万字节）；磁盘存储器更大，当然也更慢（访问时间15微秒，可储存10亿字节）；磁带或光盘存储器容量最大，同时也最慢（访问时间按秒计，可储存1万亿字节以上）。这些数字可能会随着时间发生变化，但容量与速度之间的基本关系将保持一致。

存储器延迟（memory latency）处理器向主存储器发出请求并得到返回数据之间所花的时间。就目前技术而言，处理器在等待的这段时间内可以执行100次操作。

微处理器（microprocessor）在单个芯片上面积约为4平方英寸或26平方厘米的处理器。如今随处可见价值数美元到数百美元的微处理器，它们的速度和10年前价值数百万美元的机器差不多。

微秒（microsecond）一百万分之一秒。

毫秒（millisecond）千分之一秒。

多处理器（multiprocessor）多个处理器通过快速网络相连、共享存储器的计算机。

多道编程（multiprogramming）单个处理器同时执行多个程序、每个程序每次只执行几毫秒。这种思想的好处之一是任何个别程序都不会对其他程序造成过度延迟。好处之二是当一个程序等待从磁盘或屏幕返回数据时，其他程序仍然可以继续运行。与之相对的是*批处理*。

多任务（multitasking）与*多道编程*相同。

互斥（mutual exclusion）确保两个事件不会*并发*发生。交通红绿灯是为了确保交叉方向行驶的车辆不会在同一时刻穿过十字路口。

纳秒（nanosecond）十亿分之一秒。

非确定性（nondeterminism）不是*确定性*的。

非确定性多项式问题，NP问题（nondeterministic polynomial problem，**也称NP problem**）任何可能的解答都可以快速（用术语来说就是在多项式时间内）得到检验的问题。显而易见，任何能快速解决的问题（称为P问题）也具有这种性质，因此P问题都是NP问题。理论计算机科学中一个尚待解决的重要问题即：NP问题是否也是P问题。

NP完全问题（NP-complete problem）如果某问题可以证明和NP问题一样难，则该问题就是NP完全问题。这表示如果任何NP完全问题能被快速解决，那么所有NP问题都能被快速解决。巡回推销员问题就是一个典型的NP完全问题：给定一组城市以及两两之间的旅途费用额度，那么是否存在一条路线可以让推销员走遍所有城市然后回到起点，并且保证总费用低于一定额度？不管正确与否，NP完全问题的任何候选解答都很容易验证，但没人知道到底能不能很快得到好的解答。这表示NP完全问题有可能需要*指数时间*。

面向对象（object orientation）一种编程类型，将各个软件组件当做黑箱：其外部操作对于用户是可见的，但内部的实施则被隐藏了。例如日常生活中的收音机，我们可以控制音量、选择频段，但从小就被警告说不要打开它看内部器件，以防触电。通过面向对象，我们可以用可靠而且可重复使用的组件来构造极为庞大的程序，而无需关心其内部原理。

并行计算（parallel computation） 由多个处理器处理任务的各个部分，从而更快地完成任务。例如，某个计算机图形程序可能会将显示屏的各个部分分配给不同的处理器。

可并行任务（parallelizable task） 可以被分解为多个彼此几乎相互独立的子任务的任务。如果某任务是可并行的，那么用多个处理器来处理它会比用单个处理器要快。例如在日常生活中，由烧水煮茶和烘焙糕点组成的任务是可并行的，分娩则是不可并行的。

语法分析（parsing） 对文法的分析。在英语等自然语言中，分析一个句子表示用树形结构图表现各个词性（冠词、形容词、名词、动词等）以判断它在语法上是否正确。对于计算机语言，语法分析包括判断它在语法上是否正确，以及如何翻译程序。例如Fortran的语法类似普通算术，那么x + y在语法上就是正确的，而x + ÷ y则是错误的。

perebor 俄语词汇，表示“蛮力”。蛮力搜索是指那种需要穷举全部或几乎全部可能性才能得到解答的搜索。普遍认为NP完全问题需要蛮力搜索。

持久性数据结构（persistent data structure） 参见*数据结构*。

流水线并行（pipeline parallelism） 并行的一种形式，处理器同时执行处于不同完成阶段的多个操作。日常生活中的汽车装配线就是流水线并行。装配线一次装配多辆汽车，而每辆汽车都处于不同的安装阶段。因此流水线或装配线的生产率将按执行最长的步骤所需的时间而定。

平面图测试（planarity testing） 判断一个图是否可以通过重画，使各边只在节点处相交。

多项式问题，P问题（polynomial problem，也称P problem） 能够快速找到解答，而且是*确定性*的解答的问题。对于计算机科学家来说，“快速”是指解决问题的时间和问题的大小是多项式函数的关系，而不是*指数函数*的关系。具体来说，如果解决某个大小为N的问题所需的时间与N^K成正比（K与N无关），那么该问题就是多项式问题（或者说P问题）。排序、标准流问题和加密都是多项式时间算法的例子。

素数（prime number） 类似13、17、101这样只能被自身和1整除的数。素数在*密码技术*中得到了广泛应用。

私钥密码技术（private-key cryptography） 参见*密码技术*。

谓词演算（predicate calculus） 逻辑的一种形式系统，用于表示关于群体及个体的事实，1879年由时年31岁的德国数学家戈特洛布·弗雷格提出。谓词演算允许存在这样的陈述：“所有的奥运会选手都很健康。”如果你相信这一点，同时又给出陈述“玛丽是一位奥运会选手”，那么就可以推断玛丽很健康。因此谓词演算的功能比*命题演算*更强，因为后者只能表示个体。

进程（process） 正在运行的程序。如果把处理器比作厨师，程序比作菜谱，那么进程就是正在按菜谱做菜的厨师。

处理器（processor） 一个计算元件，可以执行加、减、乘、除四则运算和基本逻辑操作。一台计算机可以有多个处理器。

生产者（producer） 参见*缓冲区*。

程序（program） 通过特定的编程语言发出的，让计算机执行任务的指令。程序就是按人类语言写就、包含了所有细节的菜谱。

程序验证（program verification） 也称形式化验证（formal verification），计算机科学的子学

科，目的是尝试用数学方法证明程序将会完全按照指定的方式运行。一些研究者试图验证那些与安全性关系极大的软件，例如飞机中使用的软件。对于常用领域的软件，人们主要是进行一系列的测试。如果没有什么错误，软件就会被交付使用。

编程语言（programming language） 向计算机发出指令的手段之一。目前有数百种编程语言，其中一些主要由科学家及工程师使用，例如Fortran和Lisp，另外一些则被我们这样的大众所使用，例如文字处理语言等。一个好的编程语言应贴合特定的应用，就如好鞋会贴合特定用途一样（远足、长跑、跳舞或沙滩漫步）。*编译器*可以把高级编程语言翻译成*机器语言*。

命题演算（propositional calculus） 逻辑的一种形式化系统，用于表示关于个体的事实。通过命题演算的规则可以推断出新的命题。比如说，如果命题“要么米奇是一只老鼠，要么唐纳是一只鸭子”和“唐纳不是一只鸭子”都成立，那么我们就能推导出“米奇是一只老鼠”成立。与之相对的是*谓词演算*。

公钥密码技术（public-key cryptography） 参见*密码技术*。

随机存取存储器（random access memory，RAM） 也称为主存储器，访问时间约为100纳秒。大多数程序在主存储器中的速度比在磁盘上要快。

随机化算法（randomized algorithm） 迈克尔·拉宾提出的一种算法设计技巧，利用随化机进程（例如抛硬币）来寻找答案。这种算法有时候可能会得出错误的结果（例如将某个不是素数的数指认为素数），但算法的设计者可以设置这种概率的大小。通常的概率大概是$1/10^{24}$。

读（read） 一个计算机指令，指把计算机存储器中某个寄存器的值取出给处理器，同时不会影响存储器中的这个值。

只读存储器（read-only memory，ROM） 单个或一小组芯片，包含由生产商植入或“烧入”的无法更改的指令。一般情况下ROM包含了计算机启动时需要执行的基本程序指令。ROM的作用类似于人体中控制反射的神经系统。

递归（recursion） 根据函数本身定义函数的一种编程形式。如果是在家谱中，递归就会将词汇“祖先”定义为父母或祖先的父母。

可约简性（reducibility） 通过证明解决问题X必将会解决问题Y（用术语表示就是Y可约简为X）来证明X很难的一种技巧。如果已知Y很难解决，那么X必定也很难解决。

寄存器（registers） 处理器中一种非常快的存储器元件。通常一台计算机中会有16到10 000个寄存器。

分解规则（resolution rule） 约翰·阿兰·罗宾逊发明的一种证明逻辑定理的有效方法。

语义（semantics） 对自然语言或计算机语言的表达含义的一种描述。在自然语言中，即使某个短语有*语法*错误，但它在语义上也可能是有意义的（例如“走你”）。而在计算机语言的表达中（也就是程序），如果语法出现错误就不会被翻译为*机器语言*。而如果它们的语法正确，但语义是错误的（例如用加替换了减），那么程序在使用时就会出错。

信号灯（semaphore） 防止火车相撞的一种信号机制。艾兹赫尔·戴克斯彻将这一想法引入到计算机科学中，成为了一种确保*互斥*的机制。

最短路径问题（shortest-path problem） 给出一个图，它的每一条边都有权值，最短路径问

题就是要找出从起点到终点且总权值最小的路线。

模拟（simulate）以类比的方式运作。例如，可以编写计算机程序来模拟石油穿过沙石的流动。这种程序可以帮助地质学家选择钻取石油的最佳地点。

软件（software）一个或多个程序。

结构化证明（structured proofs）一种层级化的证明方法：主要断言在最高层级，支持最高层级断言的理由在第二层级，支持第二层级断言的理由在第三层级，依此类推。

语法（syntax）参见*文法*。

巡回推销员问题（traveling salesman problem）给定一个通过航线相连的城市网络，巡回推销员问题就是要找出让推销员访问所有城市并返回，且旅行费用最少的路线。这是一个*NP完全问题*。

晶体管（transistor）由硅或锗等半导体材料制成的电子开关。一个现代计算机的芯片中大约有10亿个晶体管。这些器件可以存储信息或执行*布尔运算*。

图灵测试（Turing test）由阿兰·图灵于1950年提出的一种测试，用于判断计算机是否能展现出人类智能的水准。大体上如下：假设一个人通过终端键入文字，与一个隐藏的对话者交换信息，这个对话者可能是一台计算机，也可能是一个人。如果测试者无法判断对方是人还是计算机，那么计算机就展现出了人类智能的行为。这一测试是人工智能尚待解决的挑战之一。

不可决定问题（undecidable problem）任何计算机程序都无法保证能解决的问题。实例参见*停机问题*。

向量（vector）一组有序的数，其解释根据应用的不同而不同。在制图学中，我们以两个元素的向量来表示地球表面某点的经度和纬度。伊斯兰教圣地麦加在(22, 40)，而(40, 22)则在希腊首都雅典附近。

向量处理（vector processing）针对（通常很大的）向量的数学操作。科研中常常以很大的向量表示物理现象，向量各元素表示特定时间内各个不同地点的测量值。要对现象的演变进行预测就必须对向量进行计算。这些计算往往比较规则，可以进行*流水线并行*处理。

虚拟现实（virtual reality）由弗雷德里克·布鲁克斯、伊凡·苏泽兰和其他一些先行者开创的领域，利用计算机创造出一个虚幻的世界，用户可以在其中触摸或控制并不真实存在的对象。在布鲁克斯的研究中，用户可以穿越分子键，也可以挪动原子的位置。

写（write）一个计算机指令，可以改变计算机存储器中某个寄存器的值。

参考文献

我们请书中的每位科学家选择了一些他们最钟爱的自己所著的论文或图书。下面首先按各章顺序列出了他们推荐的著作，然后给出了本书四个部分中所涉及的一般参考资料。

第一部分　语言大师

Chomsky, Noam. 1957. *Syntactic Structures*. The Hague: Mouton.

Gelernter, David, and Suresh Jagannathan. 1990. *Programming Linguistics*. Cambridge, MA: MIT Press.

Goldstine, H. 1972. *The Computer from Pascal to von Neumann*. Princeton, NJ: Princeton University Press.

Sethi, Ravi. 1990. *Programming Languages: Concepts and Constructs*, Reading, MA: Addison-Wesley.

1. 约翰·巴科斯

Backus, John W. 1981. "The History of Fortran I, II, and III." *History of Programming Languages*. New York: Academic Press.

Backus, John W. 1978. "Can Programming Be Liberated from the von Neumann Style? A Functional Styles and Its Algebra of Programs." *Communications of the ACM* 21(8):613-641.

Backus, John W., R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. 1957. "The Fortran Automatic Coding System. " 1957 *Western Joint Computer Conference*, pp. 188-198.

2. 约翰·麦卡锡

McCarthy, John. 1993. "Notes on Formalizing Context. " *International Joint Conference on Artificial Intelligence*-93, pp. 555-560.

McCarthy, John. 1986. "Applications of Circumscription to Formalizing Common Sense Knowledge. " *Artificial Intelligence*, April 1986, pp. 89-116.

McCarthy, John. 1960. "Recursive Functions of Symbolic Expressions and their Computation by Machine. " *Communications of the ACM*, April 1960, pp. 184-194.

McCarthy, John. 1959. "Programs with Common Sense. " *Proceedings of the Teddington Conference on the Mechanization of Thought Processes*. London: Her Majesty's Stationery Office.

McCarthy, John, and P. J. Hayes. 1969. "Some Philosophical Problems from the Standpoint of Artificial Intelligence. " In D. Michie (ed,) *Machine Intelligence 4*. New York: Elsevier.

3. 艾伦・C.凯

Kay, Alan. 1993. "The Early History of Smalltalk, " *ACM SIGPLAN*, March 1993, pp. 69-96.

Kay, Alan. 1991. "Computers, Networks and Education. " *Scientific American*, September 1991, pp. 138-148.

Kay, Alan. 1984. "Computer Software. " *Scientific American*, September 1984, pp. 52-59.

Kay, Alan. 1977. "Microelectronics and the Personal Computer. " *Scientific American*, September 1977, pp. 230-239.

Kay, Alan. 1972. "A Dynamic Medium for Creative Thought. " NCTE Conference, November 1972.

Kay, Alan. 1972. "A Personal Computer for Children of All Ages." ACM National Conference, August 1972.

Kay, Alan, and Adele Goldberg. 1977. "Dynamic Personal Media." *IEEE Computer*, March 1977, pp. 31-42.

Kay, Alan, and Adele Goldberg. 1976. Smalltalk-72 Instruction Manual. Xerox PARC, March 1976.

第二部分 算法大师

Aho, A. V., J. E. Hopcroft, and J. D. Ullman. 1971. *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley.

Cormen, T. H., C. E. Leiserson, and R. L. Rivest. 1991. *Introduction. to Algorithms*. New York: McGraw-Hill.

Davis, Martin. 1987. "Mathematical Logic and the Origin of Modern Computers." *Studies in the History of Mathematics*. Washington, D.C.: The Mathematical Association of America.

Garey, Michael R., and David. S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: W. H. Freeman.

Herken, Rolf (ed). 1988. *The Universal Turing Machine: A Half Century Survey*. Oxford: Oxford University Press.

Hodges, A. 1983. *Alan Turing: The Enigma*. New York: Simon and Schuster.

Kolmogorov, A. N., and V. A. Uspenskii. "Algorithms and Randomness." *Theoria Veroyatnostey i ee Primeneniya* (Theory of Probability and Its Applications) 3(32):389-412.

Trahktenbrot, B. A. 1984. "A Survey of Russian Approaches to Perebor (Brute-Force Search) Algorithms. " *Annals of the History of Computing* 6:384-400.

4. 艾兹赫尔・W.戴克斯彻

Dijkstra, Edsger W. 1974. "Self-Stabilizing Systems in Spite of Distributed Control. " *Communications of the ACM* 17:453-455.

Dijkstra, Edsger W. 1960. "Recursive Programming" *Numerische Mathematatik* 2:312-318.

Dijkstra, Edsger W., and Carel A. Scholten. 1990. "Predicate Calculus and Program Semantics. " In *Texts and Monographs rn Computer Science*. New York: Springer-Verlag.

Dijkstra, Edsger W., and Carel A. Scholten. 1980. "Termination Detection for Diffusing Computations. " *Information Processing Letters* 11(1):1-4.

5. 迈克尔·O.拉宾

Rabin, Michael. 1989. "Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance." *Journal of ACM* 38:335-348.

Rabin, Michael. 1979. "Digital Signatures and Public-Key Functions Are as Intractable as Factorization." *MIT Laboratory for Computer Scrence Technical Report no*. 212.

Rabin, Michael. 1976. "Probabilistic Algorithms." In J. F. Traub (ed.). *Algorithms and Complexity: New Directions and Recent Trends*. New York: Academic Press, 1976.

Rabin, Michael. 1969. "Decidability of Second-Order Theories and Automata on Infinite Trees. " *Transactions of the American Mathematical Society* 141:1-35.

Rabin, Michael. 1963. "Probabilistic Automata. " *Information and Control* 6:230-245.

Rabin, Michael. 1960. "Degree of Difficulty of Computing a Function and a Partial Ordering of Recursive Sets." *Office of Naval Research Contracts Technical Report*. Jerusalem: Hebrew University.

Rabin, Michael. 1958. "Recursive Unsolvability of Group Theoretic Problems. *Annals of Mathematics* 67:172-194.

Rabin, Michael, Y. Aumann, Z. M. Kedem, and K. V. Palem. 1993. "Highly Efficient Asynchronous Execution of Large Grained Parallel Programs. " *34th Symposium on the Foundations of Computer Science*, 1993, pp. 271-280.

Rabin, M., and M. Fischer. 1974. "Super Exponential Complexity of Presburger Arithmetic." *SIAM-AMS Proceedings of Symposium on Complexity of Computations* 7:27-41.

Rabin, M., and R. Karp. 1987. "Efficient Randomized Pattern-NJatching Algorithms. " *IBM Journal of Research and Devopment* 31:249-260.

Rabin, Michael, and Dana Scott. 1959. "Finite Automata and Their Decision Problems." *I.B.M. Journal of Research and Devolpment* 3:114-125.

6. 高德纳

Graham, Ronald L., Donald E. Knuth, and Oren Patashnik. 1989. *Concrete Mathematics*. Reading, MA: Addison-Wesley.

Knuth, Donald E. 1991. *Literate Programming*. Stanford, CA: Center for the Study of Language and Information.

Knuth, Donald E. 1986. *Computers and Typesetting*. (series.) Reading, MA: Addison-Wesley, 1986.

Knuth, Donald E. 1968. "Semantics of Context-Free Languages. " *Mathematical Systems Theory* 1968:127-145; 1971:95-96.

Knuth, Donald E. 1968. *The Art of Computer Programming*. (series.) Reading, MA: Addison-Wesley.

Knuth, Donald E. 1965. "On the Translation of Languages from Left to Right. " *Information and Control* 1965:607-639.

Knuth, Donald E., and Peter B. Bendix. 1970. "Simple Word Problems in Universal Algebras. " in J. Leech (ed.) *Computational Problems in Abstract Algebra*. New York: Pergamon.

Knuth, Donald E., Svante Janson, Tomasz Luczak, and Boris Pittel. 1993. "The Birth of the Giant Component. " *Random Structures and Algorithms* 1993:233-358.

7. 罗伯特·E.陶尔扬

Goldberg, Andrew V., and Robert E. Tarjan. 1988. "A New Approach to the Maximum-Flow Problem." *Jourrnal of the ACM* 35:921-940.

Hopcroft, John, and Robert E. Tarjan. 1974. "Efficient Planarity Testing." *Journal of the ACM* 21(4):549-568.

Lipton, Richard J., and Robert E. Tarjan. 1979. "A Separator Theorem for Planar Graphs." *SIAM Journal of Applied Mahematics* 36:177-189.

Sarnak, Neil, and Robert E. Tarjan. 1986. "Planar Point Location Using Persistent Search Trees." *Communications of the ACM* 29:669-679

Tarjan, Robert E. 1987. "Algorithm Design." *Communications of the ACM* 30:205-212.

Tarjan, Robert E. 1985. "Amortized Computational Complexity." *SIAM Journal of Algorithms and Discrete Methods* 6:306-318.

Tarjan, Robert E. 1983. *Data Structures and Network Algorithms*. Philadelphia: Society for Industrial and Applied Mathematics.

Tarjan, Roben E. 1972. "Depth-First Search and Linear Graph Algorithms." *SIAM Journal on Computing* 1972:146-160.

8. 莱斯利·兰伯特

Lamport, Leslie. 1987. "A Fast Mutual Exclusion Algorithm." *Transactions on Computer Systems* 5(71):1-11.

Lamport, Leslie. 1979. "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. " *IEEE Transactions on Computers* C-28(9):690-691.

Lamport, Leslie. 1978. "Time, Clocks, and the Ordering of Events in a Distributed System. " *Communications of the ACM* 21(7):558-565.

Lamport, Leslie. 19 77. "Concurrent Reading and Writing. " Communications of the ACM 20(11): 806-811.

Lamport, Leslie. 1974. "A New Solution of Dijkstra's Concurrent Programming Problem. " *Communications of the ACM* 17(8):453-455.

9. 史提芬・古克

Aanderaa, S. O., and Stephen Cook. 1969. "On the Minimum Computation Time of Functions. " *Transactions of the American Mathematical Society* 142:291-314.

Borodin, A., and Stephen Cook. 1982. "A Time-Space Tradeoff for Sorting on a General Sequential Model of Computation. " *Siam Journal on Computing* 11(2):287-297.

Cook, Stephen 1975. "Feasibly Constructive Proofs and the Propositional Calculus. " *Proceedings of the Seventh Annural ACM Symposium on the Theory of Computing*, May 1975, pp. 83-97.

Cook, Stephen. 1971. "The Complexity of Theorem Proving Procedures. " *Proceedings of The Third Annual ACM Symposium on the Theory of Computing*, May 1971; pp. 151-158.

Cook, S., and R. Reckhow. 1979. "The Relative Efficiency of Propositional Proof Systems. " *Journal of Symbolrc Logrc* 44(1):36-50.

9. 利奥尼德・列文

Babai, L., L. Fortn.ow, L. Levin, and M. Szegedy. 1991. "Checking Computations in Polylogarithmic Time. " ACM *Symposium on the Theory of Computing*, 1991, pp. 21-31.

Goldreich, Oded, and Leonid Levin. 1989. "A Hard-Core Predicate for All One-Way Functions. " *ACM Symposium on the Theory of Comptitrng*, 1989, pp. 25-32.

Levin, Leonid. 1964. "On Storage Capacity for Algorithms." *Doklady Akademii Nauk SSSR* 14(5):1464-1466.

Levin, Leonid. 1984. "Randomness Conservation Inequalities. *Information and Control* 61(1):15-3 7.

Venkatesan, R., and Leonid Levin. 1984. "Random Instances of a Graph Coloring Problem Are Hard. " *ACM Symposium on The Theory of Computing*, 1984, pp. 217-222.

第三部分　架构大师

Flynn, M. J. 1972, "Some Computer Organizations and their Effectiveness." *IEEE Transactions on Computers* C-21(9):948-960.

Friedman, D. P., and D. S. Wise. 1978. "Aspects of Applicative Programming for Parallel Processing." *IEEE Transactions on. Computers* C-27(4):289-296.

Jefferson, D. R. 1985. "Virtual Time. " *ACM Transactions on Programming Languages and Systems* 7(3): 404-425.

Ladner, R. E., and M. J. Fischer. 1980. "Parallel Prefix Computation. " *Journal of the ACM* 27(4): 831-838.

Schwartz, J. T. 1980. "Ultracomputers." *ACM Transactions on Programming Languages and Systems* 2(4):484-521.

10. 弗雷德里克・布鲁克斯

Airey, John, John Rohlf, and Frederick P. Brooks, Jr. 1990. "Towards Image Realism with Interactive Updates in Complex Virtual Building Environments. Proceedings of 1990 Symposium on Interactive 3D Graphics, *Computer Graphics* 24(2):41-50.

Amdahl, G. M., G. A. Blaauw, and F. P. Brooks, Jr. 1964. "Architecture of the IBM System/360." *IBM Journal of Research and Development* 8:87-101.

Bergman, Lawrence D., Jane S. Richardson, David C. Richardson, and F. P. Brooks, Jr. 1993. "VIEW—An Exploratory Molecular Visualization System with User-Definable Interaction Sequences." *Computer Graphics: Proceedrngs* of SIGGRAPH 93, 27(4):117-126.

Blauww, G. A., and Brooks, F. P., Jr. *Computer Arcbitecture* (2 vols.). Reading, MA: Addison-Wesley, in preparation.

Brooks, F. P., Jr. 1988. "Grasping Reality through Illusion: Interactive Graphics Serving Science." In D. Frye and S. Sheppard (eds.), *Computer Human. Interaction '88 Proceedings*. Reading, MA: Addison-Wesley, pp. 1-11,

Brooks, F. P., Jr. 1987. "No Silver Bullet-Essence and Accidents of Software Engineering." *IEEE Computer* 20(4):10-19.

Brooks, F. P., Jr. 1977. "The Computer 'Scientist' as Toolsmith: Studies in Interactive Computer Graphics." In B. Gilchrist (ed.) *Information Processing 77*. Amsterdam: North-Holland.

Brooks, F. P, Jr. 1975. *The Mythical Man-Month: Essays on Software Engineering*. Reading, MA: Addison-Wesley.

Brooks, Frederick P., Jr., Ming Ouh-Young, James J. Batter, and P. Jerome Kilpatrick. 1990. "Project GROPE: Haptic Displays for Scientific Visualization." *Computer Graphics: Proceedrngs of SIGGRAPH 90*, 24(4): 177-185.

Taylor, Russell M. II, Warren Robinett, Vernon L. Chi, Frederick P. Brooks, Jr., William V. Wright, R. Stanley Williams, and Erik J. Snyder. 1993. "The Nanomanipulator: A Virtual-Reality Interface for a Scanning Tunneling Microscope. *Computer Graphics: Proceedings of SIG GRAPH 93*, 21(4): 127-134.

11. 伯顿・J.史密斯

Alverson, Robert, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. 1990. "The Tera Computer System." *Proceedings 1990 International Conference on SuperComputing*, Amsterdam, June 1990, pp. 1-6.

Cathey, W. T., and Burton Smith. 1979. "High Concurrency Data Bus Using Arrays of Optical Emitters and Detectors." *Applied Optics* 18:1687.

Smith, Burton. 1990. "The End of Architecture." Keynote address presented at the 17th International Conference on Computer Architecture Seattle, Washington, December 1990. *Architecture News* 18(4):10.

Smith, Burton. 1987. "Shared Memory, Vectors, Message Passing, and Scalability." *Proceedings 1987 DFVLR Seminar on Parallel Computing in Scrence and Engineering, Lecture Notes in Computer Science* 295. New York: Springer-Verlag, pp. 29-34.

Smith, Burton. 1985. "The Architecture of HEP. " In J.S. Kowalik (ed.). *Parallel MIMD Computation: The HEP System and Its Applications*. Cambridge, MA: MIT Press, pp. 41-58.

Smith, Burton, and David Callahan. 1990. "A Future-Based Parallel Language for a General-Purpose Highly Parallel Computer. " In D. Gelernter, A. Nicolau, and D. Padua (eds.). *Languages and Compilers for Parallel Computing*, Research Monographs in Parallel and Distributed Computing. Cambridge, MA: Pitman and MIT Press, pp. 95-113.

12. W.丹尼尔·希利斯

Hillis, W. D. 1993. "Why Physicists Like Models and Wrhy Biologists Should. " *Current Biology* 3(2).

Hillis, W. D. 1988. "Intelligence As an Emergent Behavior: Or, The Songs of Eden. " *Daedalus* 117(1):175-189.

Hillis, W. D. 1985. *The Connection Macbine*. Cambridge: The MIT Press.

Hillis, W. D., and B. M. Boghosian. 1993. "Parallel Scientific Computation. " *Science* 13:856-863.

Hillis, W. D., and Guy L. Steele. 1986. "Data Parallel Algorithms. " *Communications of the ACM 29*(12):1170-1173.

Hillis, W. D., and Lewis W. Tucker. 1993. "The CM-5 Connection Machine: A Scalable Supercomputer. " *Communications of the ACM* 36(11):30-40.

第四部分　机器智能的雕塑大师

Crevier, Daniel. 1993. *AI: The Tumultuous History of the Search for Artificial Intelligence*. New York: Basic Books.

Dreyfus, Hubert. 1979. *What Computers Can't Do*. New York: Harper & Row.

Levy, Steve. 1992. *Artificial Life: A Report from the Frontier Where Computers Meet Biology*. New York: Vintage.

McCorduck, Pamela. 1979. *Machines Who Think*. San Francisco: W. H. Freeman.

Michakski, Ryszard (ed.). 1983. *MacP ine Learning-An AI Approacb*. Palo Alto, CA: Tioga Publishing.

Minsky, Marvin. 1985. The Society of Mind. New York: Simon and Schuster.

Moravec, Hans. 1988. *Mind Children: The Future of Robot and Human Intelligences*. Cambridge, MA: Harvard University Press.

Schank, Roger C. 1987. *The Cognitive Computer: On Language, Learning, and Artificial Intelligence*. New York: Walker.

Yazdani, M., and A. Narayanan (eds.) 1986. *Artificial Intelligence—Human Effects*. Chichester, England: E. Horwood.

13. 爱德华 • A.费根鲍姆

Feigenbaum, E. A. 1977. "The Art of Artificial Intelligence: Themes and Case Studies in Knowledge Engineering. " *International Joint Conference on Artificial Intelligence* 1977.

Feigenbaum, E. A., and J. Feldman (eds.). 1963. *Computers and Thought*. New York: McGraw-Hill.

Feigenbaum, E. A., B. Buchanan, and J. Lederberg. 1971."On Generality and Problem Solving: A Case Study Using the DENDRAL Program. " In D. Michie (ed.), *Machine Intelligence* 6. New York: Elsevier, 165-190.

14. 道格拉斯 • B.莱纳特

Lenat, Douglas. 1984. "Computer Software for Intelligent Systems: An UnderView of AI. " *Scientific American, September* 1984, pp. 204-213.

Lenat, Douglas. 1983. "Three Case Studies in Learning. " In R. S. Carbonell, J. G. Michalski, and T. M. Mitchell (eds.) *Machine Learning*. Palo Alto, CA: Tioga Press.

Lenat, Douglas. 1977. "The Ubiquity of Discovery." *The Journal of Artificial Intelligence*, December 1977, pp. 257-285.

Lenat, Douglas. 1975. "BEINGS: Knowledge as Interacting Experts." *Proceedings of the Fourth Internatronal Jornt Conference on Artificial Intelligence 75*. Tbilisi, U.S.S.R, September 1975.

Lenat, D., D. Borning, D. McDonald, S. Taylor, and S. Weyer. 1983. "Knowsphere: Design of an Expert System with an Encyclopedic Knowledge Base." *Proceedings of the Eighth International Jornt Conference on Artificial Intelligence 83*. Karlsruhe, Germany, August 1983.

Lenat, Douglas, and John Seely Brown. 1984. "The Nature of Heuristics IV: Why AM and Eurisko Appear to Work. " *Journal of Artrficial Intelligence*, July 1984.

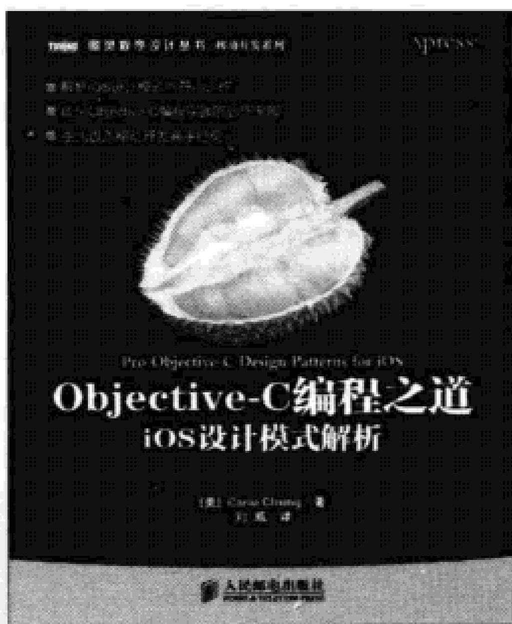
L.enat, Douglas, and Edward Feigenbaum. 1991. "On the Thresholds of Knowledge." *Artificial Intelligence* 47:185-250.

Lenat, Douglas, and R. V. Guha. 1994. "Enabling Agents to Work Together. " *Communications of ACM*, July 1994, pp. 126-142.

Lenat, Douglas, and R. V. Guha. 1990. *Building Large Knowledge-Based Systems: Representation. and Inference in the CYC Project*. Reading, MA: Addison-Wesley.

TURING 图灵教育

图灵最新重点图书



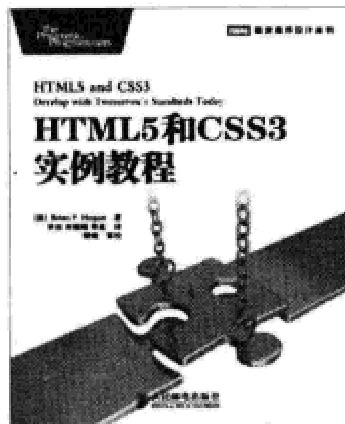
- 解析 iOS 设计模式的开山之作
- 优化 Objective-C 编程实践的必修宝典
- 由此迈入移动开发高手行列

本书是基于 iOS 的软件开发指南。书中应用 GoF 的经典设计模式，介绍了如何在代码中应用创建型模式、结构型模式和行为模式，如何设计模式以巩固应用程序，并通过设计模式实例介绍 MVC 在 Cocoa Touch 框架中的工作方式。

本书适用于那些已经具备 Objective-C 基础、想利用设计模式来提高软件开发效率的中高级 iOS 开发人员。

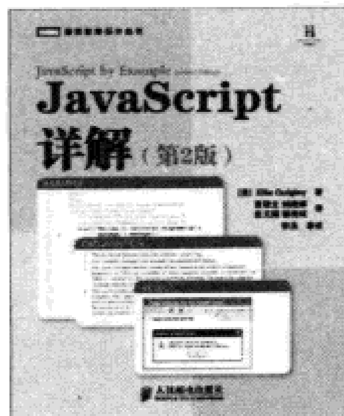
书　名：Objective-C 编程之道：iOS 设计模式解析
书　号：978-7-115-26586-9
作　者：[美] Carlo Chung
译　者：刘威
定　价：59.00 元

布道之道：引领团队拥抱技术创新
书号：978-7-115-26727-6
定价：29.00 元

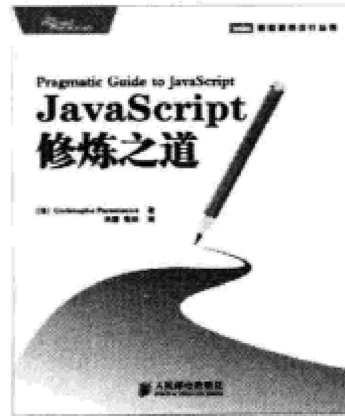


HTML5 和 CSS3 实例教程
书号：978-7-115-26724-5
定价：39.00 元

精通 Android 3
书号：978-7-115-26602-6
定价：128.00 元



JavaScript 详解（第 2 版）
书号：978-7-115-26291-2
定价：95.00 元



JavaScript 修炼之道
书号：978-7-115-26556-2
定价：29.00 元

Oracle SQL 高级编程
书号：978-7-115-26614-9
定价：89.00 元