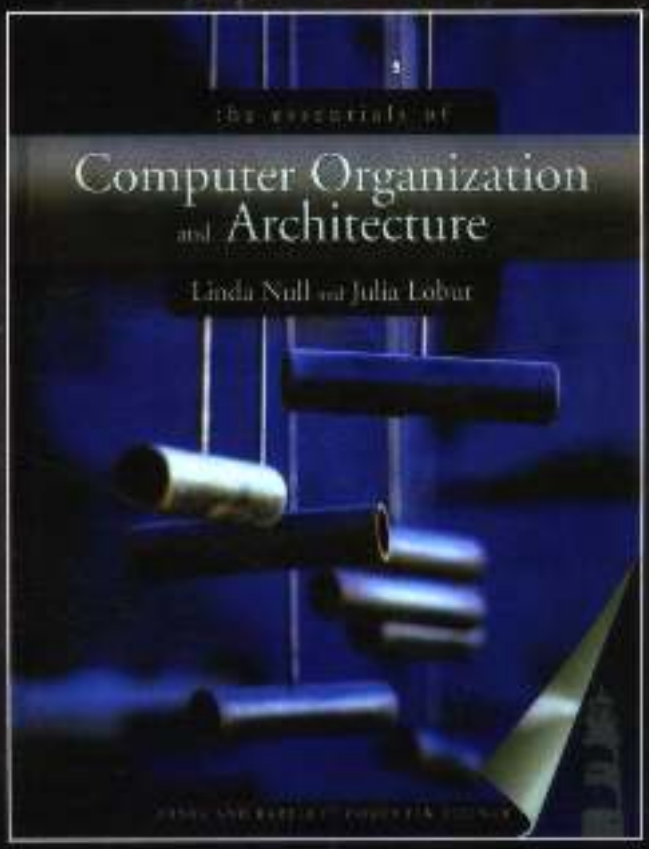
计 算 机 科 学 丛 书

计算机组成与体系结构

(美) Linda Null Julia Lobur 著 黄 河 等译



The Essentials of Computer Organization and Architecture

机械工业出版社
China Machine Press

计算机组成与体系结构

本书揭示现代计算机的内部工作方式，采用大量真实的例子，引导读者逐步由浅入深地学习计算机体系结构。本书内容经过精心组织，层次分明，内容翔实。对计算机硬件没有预备知识的要求，非常适合于大学本科的教学。本书还设计大量难度不同的习题，帮助读者更好地理解计算机工作原理。

为配合本书的教学，作者将大量资源和习题收录在 http://computerscience.jbpub.com/ECOA 网站上，供读者参考，也可登录华章网站下载。

作者简介

Linda Null 宾夕法尼亚州立大学计算机科学教授，美国爱荷华大学计算机科学博士，她长期教授操作系统、数据库、程序设计、计算机组成和体系结构课程，教龄近30年，目前，其研究领域包括面向对象数据库系统安全、操作系统安全和并发控制等。

Julia Lobur 宾夕法尼亚州立大学计算机科学兼职教授，她从事计算机工作超过20年，是一位计算机系统方面的专家，曾做过系统咨询师、程序员/分析员、系统和网络设计师、软件开发经理，并且具有丰富的教学经验。

ISBN 7-111-19048-3

9 787111 190486

HZ BOOKS 华章图书

上架指导：计算机/体系结构

华章网站 http://www.hzbook.com

网上购书：www.china-pub.com

投稿热线：(010) 88379604

购书热线：(010) 68995259, 68995264

读者信箱：hzjsj@hzbook.com

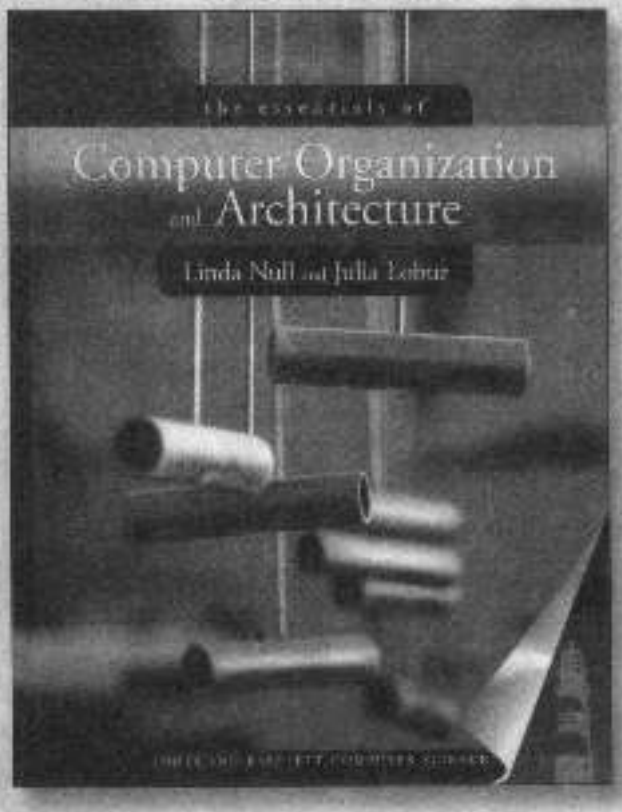
ISBN 7-111-19048-3

定价：55.00 元

计 算 机 科 学 丛 书

计算机组成与体系结构

（美）Linda Null Julia Lobur 著 黄河 等译



The Essentials of Computer Organization and Architecture

机械工业出版社
China Machine Press

本书系统介绍计算机组成与体系结构，主要内容包括：数字逻辑和数字系统、机器层次的数据表示方法、汇编层次的机器组织和结构、存储器的组成和结构、接口和通信、功能组织、多处理器和可供选择的其他结构、性能增强、网络结构和分布式计算机系统等。本书把计算机科学技术与实际问题相结合，用大量精致图片展示计算机内部结构，结构清晰，内容翔实，还包括大量补充材料和习题，方便教学。

本书可作为高等院校计算机及相关专业本科生的教材或参考书，也可供相关技术人员参考。

Linda Null, Julia Lobur: *The Essentials of Computer Organization and Architecture* (ISBN 0-7637-0444-X).

Copyright©2003 by Jones and Bartlett Publishers, Inc.

Original English language edition published by Jones and Bartlett Publishers, Inc., 40 Tall Pine Drive, Sudbury, MA 01776.

All rights reserved. No change may be made in the book including, without limitation, the text, solutions, and the title of the book without first obtaining the written consent of Jones and Bartlett Publishers, Inc. All proposals for such changes must be submitted to Jones and Bartlett Publishers, Inc. in English for his written approval.

Chinese simplified language edition published by China Machine Press.

Copyright©2005 by China Machine Press.

本书中文简体字版由 Jones and Bartlett Publishers, Inc. 授权机械工业出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

本书法律顾问　北京市展达律师事务所

本书版权登记号：图字：01-2004-5746

图书在版编目（CIP）数据

计算机组成与体系结构/（美）努尔（Null, L.），（美）劳伯（Lobur, J.）著；黄河等译.-北京：机械工业出版社，2006.7

（计算机科学丛书）

书名原文：The Essentials of Computer Organization and Architecture

ISBN 7-111-19048-3

Ⅰ.计…　Ⅱ.①努…　②劳…　③黄…　Ⅲ.计算机体系结构　Ⅳ.TP303

中国版本图书馆 CIP 数据核字（2006）第 041991 号

机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码　100037）

责任编辑：盛海燕

北京瑞德印刷有限公司印刷·新华书店北京发行所发行

2006 年 8 月第 1 版第 1 次印刷

184mm×260mm · 29 印张

定价：55.00 元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：（010）68326294

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭橥了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短、从业人员较少的现状下，美国等发达国家在其计算机科学发展的几十年间积淀的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章图文信息有限公司较早意识到“出版要为教育服务”。自1998年开始，华章公司就将工作重点放在了遴选、移译国外优秀教材上。经过几年的不懈努力，我们与Prentice Hall，Addison-Wesley，McGraw-Hill，Morgan Kaufmann等世界著名出版公司建立了良好的合作关系，从它们现有的数百种教材中甄选出Tanenbaum，Stroustrup，Kernighan，Jim Gray等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及庋藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专程为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍，为进一步推广与发展打下了坚实的基础。

随着学科建设的初步完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都步入一个新的阶段。为此，华章公司将加大引进教材的力度，在“华章教育”的总规划之下出版三个系列的计算机教材：除“计算机科学丛书”之外，对影印版的教材，则单独开辟出“经典原版书库”；同时，引进全美通行的教学辅导书“Schaum's Outlines”系列组成“全美经典学习指导系列”。为了保证这三套丛书的权威性，同时也为了更好地为学校和老师们服务，华章公司聘请了中国科学院、北京大学、清华大学、国防科技大学、复旦大学、上海交通大学、南京大学、浙江大学、中国科技大学、哈尔滨工业大学、西安交通大学、中国人民大学、北京航空航天大学、北京邮电大学、中山大学、解放军理工大学、郑州大学、湖北工学院、中国国家信息安全测评认证中心等国内重点大学和科研机构在计算机的各个领域的著名学者组成“专家指导委员会”，为我们提供选题意见和出版监督。

这三套丛书是响应教育部提出的使用外版教材的号召，为国内高校的计算机及相关专业

的教学度身订造的。其中许多教材均已为M. I. T., Stanford, U.C. Berkeley, C. M. U. 等世界名牌大学所采用。不仅涵盖了程序设计、数据结构、操作系统、计算机体系结构、数据库、编译原理、软件工程、图形学、通信与网络、离散数学等国内大学计算机专业普遍开设的核心课程，而且各具特色——有的出自语言设计者之手、有的历经三十年而不衰、有的已被全世界的几百所高校采用。在这些圆熟通博的名师大作的指引之下，读者必将在计算机科学的宫殿中由登堂而入室。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证，但我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。教材的出版只是我们的后续服务的起点。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下:

电子邮件: hzjsj@hzbook.com
联系电话: (010) 68995264
联系地址: 北京市西城区百万庄南街1号
邮政编码: 100037

专家指导委员会

（按姓氏笔画顺序）

尤晋元	王　珊	冯博琴	史忠植	史美林
石教英	吕　建	孙玉芳	吴世忠	吴时霖
张立昂	李伟琴	李师贤	李建中	杨冬青
邵维忠	陆丽娜	陆鑫达	陈向群	周伯生
周克定	周傲英	孟小峰	岳丽华	范　明
郑国梁	施伯乐	钟玉琢	唐世渭	袁崇义
高传善	梅　宏	程　旭	程时端	谢希仁
裘宗燕	戴　葵			

译 者 序

计算机的出现给人类社会的发展带来了深刻变革，计算机革命就如同自然规律一般，不可避免地将人类带入数字化时代。今天，计算机科学无处不在，与人们的生活、学习、工作和娱乐都息息相关。

本书是针对计算机科学和电子工程等专业的学生编写的内容适中、适用性强和可读性好的计算机组成原理和体系结构教材。主要内容包括：数字逻辑和数字系统、机器层次的数据表示方法、汇编层次的机器组织和结构、存储器的组成和结构、接口和通信、功能组织、多处理器和可供选择的其他结构、性能增强、网络结构和分布式计算机系统等。

本书对教材的内容进行了精心的组织，由浅入深，层次分明，对计算机硬件没有预备知识的要求，非常适合于大学本科教学。本书不仅汇集了相关领域的基本原理，而且给出了许多来自现实生活的例证，帮助读者很好地理解计算机科学和技术是如何与实际问题相结合的。从而，可以大大激发读者的学习兴趣。不容置疑，阅读本书过程中，读者将发现很多乐趣。

本书突破了计算机组成原理和体系结构的常规教学模式，将原本复杂抽象的计算机问题组合成一些特定的案例，集合起来讲述这些内容，而不是简单地把单个问题堆积在一起。本书将计算机硬件和软件的知识组合起来，给读者一幅完整的图像。读者通过一些特定环境下的学习，可以很方便地将这些基本概念移植到实际工作中。通过这种将问题的解释、举例说明、习题练习、专题讨论和模拟研究组合在一起的教学方法，在一个合适的层面上给学生一个整体学习的感受，使读者可以很轻松地了解现代计算机的内在工作方式。

与一般的计算机组成原理和体系结构的教材不同，本书专门为教学精心设计了一个教学模型MARIE（一个真正直观而简单的机器体系结构）作为教学的切入点。通过MARIE模型，读者很容易掌握计算机组成原理和体系结构的各种基本概念，而不至于纠缠于真实体系结构所存在的那些不必要的和复杂的细节之中。

本书还提供了丰富的补充材料和练习题，可以极大地增强教材本身的使用和学习效率。各种练习题可以强化对主要概念的理解，为读者提供即时的学习反馈，通过重复过程来加强理解。

在本书的翻译过程中，我们力求忠实原作。何旭曙和裴颂伟在本书初稿翻译中协助完成了部分工作，在此表示衷心感谢。限于译者水平，本书的翻译难免会存在纰漏，恳请广大读者批评指正。

黄 河

华南师范大学计算机学院

2005 年 12 月 10 日

前　言

给学生的话

本书是一本关于计算机组成与体系结构的教材，侧重于介绍信息数字处理的各种部件的功能和设计。本书按照不同的分层结构来展示计算机系统，从低层的硬件到高层的软件，包括汇编语言和操作系统。这些层次组成了一个虚拟机的分层结构。计算机组成原理主要讨论计算机的分层结构和如何划分这些分层，以及每个层次的实现。计算机体系结构主要讨论的是硬件和软件之间的接口，研究的重点是计算机系统的构造和行为特征。本书正文中的大部分内容涉及的都是计算机硬件、计算机组成原理和体系结构，以及它们与软件性能之间的相互关系。

学生们经常会问："假如我主修计算机科学专业，为什么必须学习计算机硬件？硬件不是计算机工程师的任务吗？为什么我们要关心计算机内部是什么样子呢？"作为计算机用户，我们也许不会太在意计算机的内部结构，就像我们驾驶汽车时不会太在意汽车引擎盖里面的构造一样。当然，人们可以编写高级语言程序，而无需了解这些程序的执行过程。也可以方便地使用各种应用程序包，而无需理解它们的工作原理。但是，如果当我们编写的程序需要运行得更快和更高效时，或者当我们使用的应用程序并不是真正想要的程序时，又会发生什么情况呢？作为计算机科学家，我们需要对计算机系统本身有一个基本的了解，才能解决上面的这些问题。

在计算机系统中，计算机硬件和计算机编程以及软件部分的各个方面都存在一种基本关系。要编写好的软件，对计算机有一个整体的认识是十分重要的。对计算机硬件的理解，有助于解释某些时候程序中出现的一些神秘错误，例如大家熟悉的分段出错或总线错误。对于高级程序设计人员来说，需要掌握的有关计算机组成原理和体系结构的知识的程度，取决于他们需要完成的任务。

例如，如果要编写一个编译程序，必须了解要编译的特定硬件。硬件中使用的某些思想（比如说流水线作业）可以采纳到编译技术中来，这样可以使编译器更快更高效地运行。要对大规模的复杂的实时系统建模，必须了解浮点算术应该和能够完成的任务，及其工作原理（这些内容不是同一件事情）。要编写有关视频、磁盘或其他输入/输出接口，必须很好地理解 I/O 接口和计算机体系结构的基本知识。对于嵌入式系统来说，由于它们通常在资源上受到很大限制，所以我们必须理解时间、空间和价格三者之间的平衡。若要从事硬件系统、网络或特殊算法等问题的研究调查和推荐工作，必须非常熟悉计算机的性能基准，并且能够恰当地展示系统的性能结果。在购买计算机之前，应该了解计算机的性能指标，以及某些商家为证明自己的系统优于其他系统而操控性能测试结果的方式。不管我们在哪个专业领域工作，作为一位计算机科学家，熟悉了解硬件与软件之间的相互作用是必不可少的。

读者也许会奇怪，为什么本书的标题上有一个很大的单词"essentials"（基本的）。这里有两个理由。首先，计算机组成原理的内容广泛，并且每天都在发展。第二，很难定义在信息的浩渺大海中，哪些话题真正是基本的以及哪些知识只是需要了解的。编写本书时，我们的目的是希望为大家提供一本符合美国计算机学会（ACM）和电气和电子工程师协会（IEEE）出版的计算机体系结构课程指导纲要的简明教材。计算机专家一致认为，这个课程指导纲要中的主题内容构成了有关计算机组成原理和体系结构知识的"基本"核心体。

我们对于我们认为有用的某些主题内容，但对于学生继续从事计算机科学研究和未来专业领域的进一步发展并不是基本的一些 ACM/IEEE 推荐的课程也进行了强调。这里，我们觉得对大家继续从事计算机科学的研究工作有帮助的内容应该包括操作系统、编译器、数据库管理和数据通信等。本书还包括其他一些内容，这些内容将会有助于读者理解现实生活中实际计算机系统的工作原理和过程。

我们希望阅读本书会是一种愉快的经历，并且希望读者花些时间对我们介绍的某些资料进行更深入的钻研。我们的目的是希望在正式课程的学习完成后的较长的一段时间内，本书都可以作为一本有用的参考资料。虽然我们希望尽力为读者提供大量充实的信息资料，但这些内容在读者以后的学习和事业过程中仅仅是构建一个基础。成功的计算机专业人士，将会对理解计算机的工作原理不断增添新的知识。欢迎开始计算机王国的旅行。

写给教师的话

关于本书

本书起源于宾夕法尼亚大学 Harrisburg 分校的两个计算机科学组成原理和体系结构班的授课内容。随着计算机科学课程的不断发展，我们发现不仅课程中所讲授的材料需要进行修改，而且课程的内容也需要从两个学期的教学浓缩成一学期 3 个学分的课程。许多其他学院也意识到需要对原来的内容进行压缩，以便给一些新兴的主题内容留出更多的空间。这门新课程，与本教材一样，最初是为主修计算机科学的学生准备的。其目的是讲授一些计算机科学专业的学生必须熟悉的有关计算机组成原理和体系结构方面的内容。本书不仅汇集了这些领域的基本原理，而且只要是对学生扩展专业知识和从事计算机科学的深入研究有益的主题内容都会加以介绍，激发学生的学习兴趣。

我们编著本书的主要目的是希望改变计算机组成原理和体系结构的常规教学模式。计算机科学专业的学生在完成计算机组成原理和体系结构的课程学习之后，不但要掌握有关数字计算机的主要基本概念，而且要理解这些基本概念是如何应用于现实世界中。这些概念的内容应该远远超过计算机销售商所用的专用术语和构思。事实上，学生们应该能够掌握特定的环境下给出的这些概念，并且可以将这些概念移植到实际工作中，反之亦然。另外，学生也必须为该领域的进一步学习打下坚实的基础。

我们希望借助本教材传授一些每位计算机科学专业的学生都应该了解、熟悉或掌握的知识。我们并没有要求使用本教材的学生要完全掌握本书介绍的全部内容。但是，我们非常希望本书中有些内容是学生必须掌握的，有些内容是学生必须清楚熟悉的，而对于仅做了简要介绍的内容学生只需要了解就足够了。

有些概念具有一定的深度，仅靠单独学习基本原理是很难理解的。因此，我们将问题的解决方案集合起来讲述这些内容，而不是简单地把单个问题堆积在一起。本书将问题的解释、举例说明、习题练习、专题学习和模拟研究组合在一起，在一个合适的层面上给学生一个整体的学习感受，了解现代数字计算机的内在工作方式。

我们采用一种非正规的形式编写本教材，省略了一些不必要的技术术语。在写作上，尽量简明扼要，避免一些不必要的抽象概念，希望增加学生们的兴趣。我们还扩充了内容范围，增加了一般计算机体系结构的初级教材中没有的系统软件、操作系统简介、性能测评、可选的其他计算机体系结构，以及网络导论等一些与计算机硬件密切相关的主题内容。像大多数的著作一样，我们也挑选了一种结构模型作为主线，但是这个模型是为了简化问题而精心设计的。

本教材与计算机课程 2001 的关系

2001 年 12 月，ACM 和 IEEE 联合工作小组公布了 2001 年计算机课程指南（CC-2001）。这个指南是对非常流行的计算机课程 1991 的第一次主要修改。CC-2001 与 CC-1991 相比，存在几个重要的改变，我们这里只关注有关计算机组成原理和体系结构方面的改变。CC-1991 推荐计算机体系结构的教学学时约为 59 小时（课程定为组成原理和体系结构，课程代号是 AR）。主要包括以下内容：数字逻辑、数字系统、机器层次的数据表示方法、汇编层次的计算机组织结构、存储器系统的组成和结构、接口和通信，以及可供选择的其他计算机体系结构。最新公布的 CC-2001（可在网址 www. computer. org/education/cc2001 查阅）将计算机结构的课程内容缩减为 36 个核心学时。其中包括数字逻辑和数字系统（3 学时），机器层次的数据表示方法（3 学时），汇编层次的计算机组成原理和

体系结构（9 学时），存储器组织和结构（5 学时），接口和通信（3 学时），功能组织（7 学时），以及多处理器和可供选择的其他体系结构（3 学时）。另外，CC-2001 还建议将性能增强和有关网络的结构，以及分布式计算机系统作为 CC-2001 中计算机组成原理和体系结构教学模块的一部分。在我们对课程进行全面修改并编写本教材后，让人高兴的是这本新编有关计算机组成原理和体系结构的教材与 ACM/IEEE 2001 课程指南直接相关的内容有如下的几个部分：

AR1. 数字逻辑和数字系统（核心课程）：第 1、3 章
AR2. 机器层次的数据表示方法（核心课程）：第 2 章
AR3. 汇编层次的机器组织和结构（核心课程）：第 4、5、6 章
AR4. 存储器的组成和结构（核心课程）：第 6 章
AR5. 接口和通信（核心课程）：第 7 章
AR6. 功能组织（核心课程）：第 4、5 章
AR7. 多处理器和可供选择的其他结构（核心课程）：第 9 章
AR8. 性能增强（选修）：第 9、10 章
AR9. 网络结构和分布式计算机系统（选修）：第 11 章

为什么编写本教材

毋容置疑，当今的市场上已经充斥了大量有关计算机组成原理和体系结构的教材。在我们 25 年多该课程的教学经历中，已经使用过许多好教材。但是，每次讲授课程时，课程的内容都发生了进化演变。最后，我们总是发现需要编写更多的课程笔记来弥补教材中的材料和现在课程中需要讲授的材料之间的缺口间隙。我们常常发现我们对于计算机组成原理和体系结构的课程正在从计算机工程的方法转向对于这些主题的计算机科学的方法。在决定将计算机组成原理和体系结构的课程整合为一门课程时，我们却不能找到一本可以覆盖整个主题所需要的材料，一本真正从计算机科学角度编写，没有专用计算机术语，并且可以在介绍这些主题之前激发学生兴趣而设计的教材。

本教材希望传授现代计算机系统发展过程中所采用的设计思想，以及这种设计思想对计算机科学专业的学生会产生什么影响。但是，学生在学习和鉴赏计算机设计中那些不可触摸的方方面面的内容之前，必须首先牢固掌握一些基本概念。大多数关于计算机组成原理和体系结构的教材针对这些基本概念方面的内容的有关技术信息都进行了类似的介绍。但是，我们的教材特别注重这些信息应该覆盖的程度，即特别注重书中的内容与计算机科学专业学生的相关程度。例如，在本书的全部内容中，如果需要具体的论证，我们提供如个人计算机，企业级的计算机系统，以及大型主机等，这类在实践中最可能遇到的系统类型。我们避免其他类似书籍中流行的“PC 偏置（PC bias）”这类情况，希望同学们可以了解这些结构的差异和相似性，以及在当今自动化结构体系中扮演的角色。对许多人来说，常常忘记了教材的目的，也是唯一最重要的关键是学习。为了这个目的，我们加入了许多实际例证，并且努力把理论与实践相结合并做到平衡。

本书的主要特色

本教材采用了许多方法来强化计算机组成原理和体系结构的各种概念，以便更方便学生的学习。其中包括如下一些特色内容：

- 插入段：这些插入段包含一些相关的信息，这是一些课本章节的主题之外的内容。读者可以通过这些插入段对课程内容进行深入学习和研究。
- 实际例证：本书给出了许多来自现实生活的例证，可以帮助学生更好地理解计算机科学和技术是如何与实际问题相结合的。
- 本章小结：对每章的内容做一个简明扼要的重点总结。
- 深入阅读：这部分内容为那些想要详细研究该章主要课题的读者列出了一些额外的资源。其中的

参考文献包括一些与主题有关的权威性文章和书籍。

- 复习题：每章都设计了一组复习问答题，以帮助读者牢固掌握该章的主要内容。
- 练习题：每章都挑选了大量的练习题，以加强对所学概念的理解和记忆。一些较难的练习题采用星号标记。
- 部分练习题答案和提示：为了使学生顺利完成练习题，本书还对每章一些有代表性的问题给出答案。本教材后面部分给出答案的问题采用菱形符号来标记。
- 专题内容：这部分为希望详细讲述某些专题的教师提供额外的资料，例如卡诺图和输入/输出等。这部分内容也附有练习题。
- 附录：书中的附录简要介绍或复习数据结构，包括堆栈、链接列表和树结构等。
- 术语表：内容广泛，包括各章节所有关键术语的简要定义。

关于本书的作者

本书的作者不但有超过 25 年的相关领域的教学经验，而且拥有 20 年行业实践的经历。由于这些背景，本书更加强调的是计算机组成原理和体系结构的基本原理，及其与现实世界的相互关系。通过一些现实生活的例证帮助学生更好地理解这些基本概念是如何应用于计算机世界的。

Linda Null：于 1991 年在美国爱荷华州立大学获得计算机科学博士学位，1989 年获得爱荷华州立大学计算机科学硕士学位，1983 年获得西南密苏里州立大学计算机科学教育硕士学位，1980 年获得西南密苏里州立大学数学教育硕士学位，1977 年获得西南密苏里州立大学数学和英语教育学士学位。她教授数学和计算机科学课程已经超过 25 年，目前担任宾夕法尼亚州立大学 Harrisburg 校区计算机科学研究生课程的协调管理员。她 1995 年成为宾夕法尼亚州立大学 Harrisburg 校区的一名教师。她的主要研究领域包括计算机组成原理和体系结构、操作系统和计算机安全等。

Julia Lobur：在计算机行业从业 20 余年。她是一位计算机系统的专家顾问，专业程序设计员/分析师，计算机系统和网络设计师，此外还担任软件开发经理。另外，她还兼职一些教学任务。

课程需要的预备知识

通常，本课程要求学生具有一年以上使用高级程序语言编程的经验。还需要学习一年以上的大学数学课程（微积分或离散数学），本教材使用了这些相关的数学概念。本书不要求具备计算机硬件知识。

计算机组成原理和体系结构通常作为进一步学习有关计算机高级课程的必修课程，例如操作系统（学生需要掌握有关存储器分层结构、并发、异常和中断等概念），编译器（学生要求了解有关指令系统、存储器寻址和链接等概念），网络（在学习将各元件组合成计算机网络之前必须了解计算机系统的各部分的硬件），当然，也是一些高级计算机体系结构课程的预备课程。

本书的组织结构和主要内容

本书希望全面简要地介绍计算机科学专业学生所必须掌握的基本概念。我们不认为要做到这一点，最好的方法是将内容划分为不同的主题模块。因此，本书采用了一种结构化的，但也是完整综合性的方式，围绕有关完整的计算机系统的各个层面来组织内容。

与许多流行的教材一样，我们采用自下而上（bottom-up）的方法来安排课程内容。首先从数字逻辑层次开始，构建一个学生在学习本课程前就应该熟悉的计算机应用层面。教材的内容经过了精心的组织，以便使读者在进入下一个层次之前，能够完全理解本层次的内容。当读者到达应用层面时，有关计算机组成原理和体系结构的基本概念都已经讲述。我们的目的是让学生可以将计算机硬件知识与他们所掌握的有关编程的基本概念联系起来，获得有关计算机硬件和软件组合的一个完整视图。从根本上来说，对计算机硬件的了解程度会对计算机软件的设计和性能有重大影响。如果学生们能够打下一个计算机硬件知识的坚实基础，无疑会对他们将来成为一位出色的计算机科学家大有帮助。

有关计算机组成原理和体系结构的各种概念都涉及到计算机专业人员每天从事的日常工作的方方面面。由于计算机专业人员必须接受的教育领域众多，所以我们采取从高层次视野的角度来讲述计算机的体系结构，只有在理解某个特定的概念时才涉及低层次的具体内容。例如，在讨论指令系统（ISA）时，会在各种不同的实例研究的内容中引入与硬件有关的问题，以便区分和强化与 ISA 设计有关各种问题。

本书的正文分为 11 个章节和一个附录：

- 第 1 章：简要回顾了有关计算机的历史展，阐述了计算系统发展过程中的许多重大事件，读者可以比较直观地了解计算机发展是如何达到今天的成就的。本章介绍了一些计算机专业基本术语，计算机系统的各个组成部件，计算机系统的不同逻辑层次，以及冯·诺伊曼计算机模型。本章给读者一个有关计算机的高层次的视野，以及进一步深入学习计算机知识的动机和必要的基本概念。
- 第 2 章：全面介绍了数字和字符信息的各种表示方法。读者可以学习到有关基数和几种常用的数字表示技术，例如反码（1 补）、补码（2 补）和 BCD 编码，还讨论了加、减、乘、除等运算法则。此外，还介绍了 EBCDIC、ASCII 和统一字符编码。定点表示法和浮点表示法也在本章作了介绍。最后，简单讨论了数据记录的编码方式，错误检测和错误校正问题。
- 第 3 章：传统数字逻辑内容介绍。本章详细讨论了组合逻辑和时序逻辑电路。通过本章的学习读者可以理解比较复杂的中规模集成电路（MSI），例如译码器。本章还介绍了一些更加复杂的电路，如总线和存储器。在本章最后的专题内容中，介绍了数字逻辑电路的优化和卡诺图。
- 第 4 章：详细说明了基本的计算机组成原理，并且引入了许多基本概念，包括取指-译码-执行周期、数据通路、时钟和总线、寄存器传输表示法，当然还有 CPU。本章展示了一个简单的体系结构，MARIE 机器，以及相应的指令集系统（ISA）。通过这个模型机，读者可以对涉及程序执行的基本计算机体系结构有一个全面的了解。MARIE 是一个经典的冯·诺伊曼体系结构的设计，其中包括一个程序计数器、一个累加器、一个指令寄存器、4096 字节的存储器，以及两种寻址方式。这里引入了汇编语言编程原理，以加深读者对前面介绍的有关指令格式、指令模式、数据格式和控制等概念的理解。由于本书不是一本汇编语言的教材，所以并没有打算提供汇编语言编程的实用课程。本书介绍汇编语言的最初目的是为了理解并加深一般的计算机体系结构。但是，由于本章介绍了一个 MARIE 的仿真器，可以编写汇编语言程序，并利用仿真器汇编，而且可以在 MARIE 模型机上运行。本章介绍了两种控制方法：硬连线和微编程，并对这两种方法进行了比较研究。最后，对 Intel 和 MIPS 体系结构的计算机系统进行了比较研究，以强化本章学习的基本概念。
- 第 5 章：详细讨论了计算机指令系统。包括指令格式、指令类型和寻址方式。介绍了指令级的流水线。最后，通过学习真实的指令系统（ISA），包括 Intel、MIPS 和 Java 指令系统，来加强对本章的基本概念的理解。
- 第 6 章：介绍了基本的存储器概念，比如 RAM 和不同的存储器件。接下来，还介绍了一些先进的存储器分层结构概念，包括高速缓存存储器和虚拟存储器。本章还详细讨论了高速缓存的几种映射技术：直接映射、关联映射和组关联映射技术。另外，本章还详细介绍了覆盖、分页和分段技术、TLB，以及与高速缓存有关的算法和设备。在本书的网站中，还可以找到一个本章的指南和仿真器。
- 第 7 章：详尽描述了基本输入输出（I/O）、总线通信和协议等问题，以及常用的外存储器设备，如磁盘和光盘，以及它们的数据存储格式。还讨论了 DMA，可编程的 I/O 接口，以及中断等。此外，还介绍了不同设备之间交换信息的各种技术。详细讨论了 RAID 体系结构和各种数据压缩技术。
- 第 8 章：主要讨论现行的几种编程工具（如编译器和汇编器），以及它们与运行这些程序的机器的体系结构之间的相互关系。本章的目的是加强程序员的计算机系统的观点与构成机器的硬件和体系结构之间的联系。此外，还介绍了操作系统，但仅涵盖了系统的体系结构和组成原理的细节部分，例如资源的使用和保护、陷阱和中断，以及各种其他服务。
- 第 9 章：对最近几年出现的其他可选择的体系结构做了一般性介绍。主要内容包括 RISC、Flynn

分类法、并行处理机、指令级并行处理、多处理器系统、互联网、共享存储器系统、高速缓存一致性问题、存储器模型、超标量计算机、神经网络、生物计算机、数据流计算机，以及分布式计算机系统等。本章的主要目的是告诉读者，计算机并不局限于冯·诺伊曼体系结构，并且促使读者考虑有关系统的性能问题，顺利过渡到下一章的学习。

- 第 10 章，主要介绍计算机的性能分析和管理问题。首先，引入了必要的数学预备知识，接着讨论了 MIPS、FLOPS、基准，以及计算机科学家应该熟悉的各种性能优化技术，包括分支预测、推测执行和循环优化。
- 第 11 章，主要介绍网络的组织和结构，包括网络元件和网络协议。其中，OSI 模型和 TCP/IP 协议组合将在因特网的内容中介绍。本章不打算包罗万象，主要目的是讨论与互联网结构相关的计算机体系结构的内容。

附录的内容是关于数据结构的。主要考虑是学生可能需要简要介绍或复习有关堆栈、队列和链接列表（链表）等主题内容。

教师可以按照书中编排的顺序教学，但是如果需要也可以改变各章节教学的前后顺序。图 P-1 给出了不同章节之间存在的相互依赖关系。

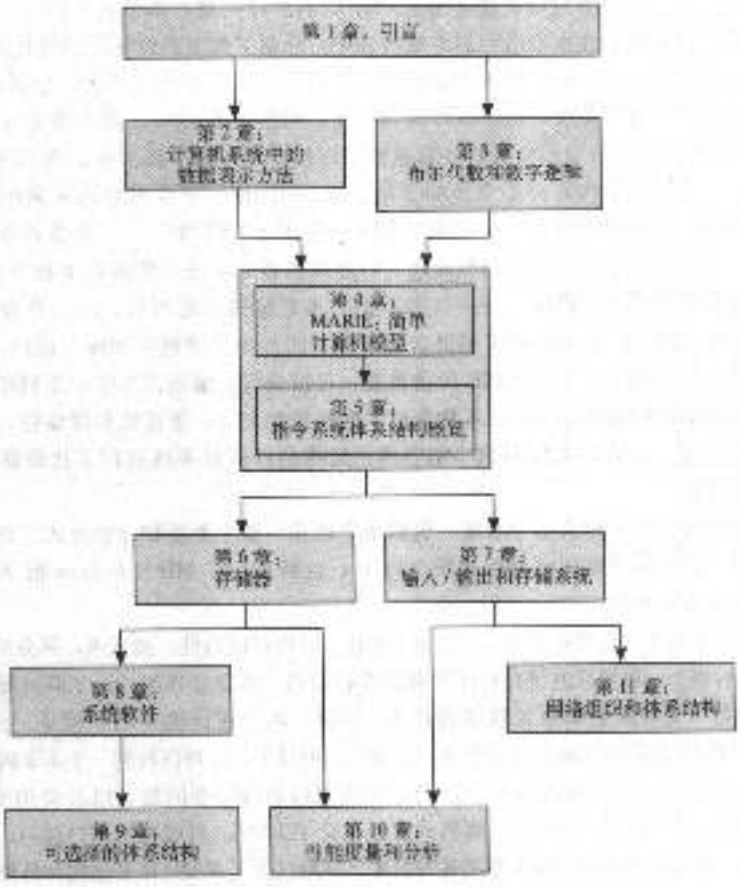


图 P-1　本书各章之间的前后依赖关系（必要条件）

面向的读者

本书最初是为主修计算机科学的大学本科学生编写的。显然如此，但本书也可以作为主修 IS 和 IT

专业的学生的教材。

本书所包含的内容远远超过了一个学期的教学内容（14 周，42 个学时）。但是，要让普通学生在一个学期的教学中掌握本书的所有内容难度较大。如果教师想详细讲述本书的全部主题内容，最好采用连续两个学期的时间安排。教师可以根据自己的教学经验和学生的实际需求，选择不同的教学安排，对主要的主题内容做不同程度的介绍。表 P-1 为教师提供了一个各章节的学时安排上的建议，以及学生们预期可以达到的相应程度。

表 P-1　学时安排建议

章　节	一学期（42 学时）		二学期（84 学时）	
	学时数	预期水平	学时数	预期水平
1	3	掌握	3	掌握
2	6	掌握	6	掌握
3	6	掌握	6	掌握
4	6	熟悉	10	掌握
5	3	熟悉	8	掌握
6	5	熟悉	9	掌握
7	2	熟悉	6	掌握
8	2	了解	7	掌握
9	3	熟悉	9	掌握
10	3	了解	9	掌握
11	3	了解	11	掌握

我们希望在正式课程结束后，本书仍然可以作为一本有用的参考书。

支持材料

教材是学习中使用的基本工具，而丰富的补充材料和练习题可以极大地增强教材本身的使用和学习效率。各种练习题可以强化对主要概念的理解，为读者提供即时的学习反馈，通过重复过程来加强理解。为此，本书准备了以下附加材料：

- *教师手册*（Instructor's Manual）：这本手册包含练习题答案和模拟考试题。此外，本手册还提供了一些有关概念的教学建议和学生通常可能会遇到的一些难点问题。
- 教学幻灯片（Lecture Slides）：这些幻灯片包含的资料可供计算机组成与体系结构一个学期的授课。
- 图表（Figures and Tables）：希望自己准备讲稿材料的教师可以下载各种图表。
- *存储器指南和仿真器*（Memory Tutorial and Simulator）：利用这套工具，学生们可以运用他们所学的有关 Cache 存储器和虚拟存储器的概念。
- MARIE *仿真器*（MARIE Simulator）：利用这个仿真器可以汇编和运行 MARIE 程序。
- *自学软件*（Tutorial Software）：提供本书中各种概念的其他一些自学软件。
- *帮助网址*（The Companion website）：所有的软件、幻灯片和有关的材料都可以从本书的网址下载：http：//computerscience. jbpub. com/ECOA

上面提供的练习题、考试样题和答案经过了多个教学班级的测试。使用本书的教师可以获得本书提供的教师手册，其中包括各个章节的教学建议和练习题答案，还有一些推荐的编程作业和模拟考试题。

教学指导模型：MARIE

在计算机组成与体系结构的教学中，结构模型的选择对教师和学生都有很大的影响。如果选择的模型过于复杂，教师和学生都可能会深陷于模型的细节中，而忽视了课堂所主要讲述的基本概念。对于实际的计算机体系结构，尽管会令人感兴趣，但是通常都具有许多独特的设计，这样使得结构无法

在入门级的课程中使用。令事情更加困难的是，实际的体系结构每天都在发生变化。另外，一般情况下很难找到一本带有适合于特定教学环境的计算机平台的模型，实际上计算机的教学环境也在年复一年地不断发生变化。

为了解决这些问题，我们实现了一个自己的简单模型，MARIE，专门设计为教学使用。通过MARIE（一个真正直观而简单的机器体系结构）模型，同学们可以方便地学习计算机组成与体系结构的基本概念，包括汇编语言，而不至于陷入真实体系结构所存在的不必要的和复杂的细节之中。尽管模型很简单，但它模拟了一个计算机功能系统。MARIE 机器仿真器，MarieSim，具有很好的用户友好的 GUI 界面，利用这个仿真器学生们可以：（1）生成和编辑源代码；（2）汇编源代码变成为机器目标代码；（3）运行机器代码；（4）调试程序。

特别地，MARIE 具有如下特点：

- 支持第 4 章引入的 MARIE 汇编语言
- 具有一个综合文本编辑器，可以用于程序生成和修改
- 十六进制的机器语言目标代码
- 具有一个综合调试器，可以实现单步执行、断点、暂停、重启，以及寄存器和存储器的跟踪记忆等模式
- 具有一个图形存储器监视器，可以显示 MARIE 中的 4096 个地址
- 可以实现 MARIE 寄存器的图形显示
- 在程序执行过程中，会高亮显示正在执行的指令
- 具有用户可控制的执行速度
- 显示状态信息
- 用户可见的符号表
- 一个交互型的汇编器，用户可以纠正各种错误，并且自动重新进行汇编，而无需改变汇编环境
- 具有在线帮助
- 具有可选的存储器清除功能，用户可以指定存储器的范围
- 用户可以修改页帧的大小
- 具有一个小的学习模仿曲线功能，学生可以很快学会系统的原理和操作

MarieSim 采用 Java™语言编写，这个系统可以移植到任何一个可运行 Java™虚拟机（JVM）的工作平台上。使用 Java 语言的学生可能希望可以阅读仿真器的源代码，并且甚至对模型的简单功能进行某种改进或添增。

图 P-2 为 MarieSim 的图形界面，即一个 MARIE 机器仿真器的图形界面。显示屏由 4 部分组成：菜单条、中间的显示区域、存储器监视器和信息显示区域。

选择菜单让用户可以控制 MARIE 机器仿真器的各种动作和行为。这些选择菜单包括对用 MARIE 汇编语言书写的程序执行装入、开始、停止、设置断点和暂定等操作。

MARIE 仿真器可以详细展示程序的汇编、装入和执行过程，所有这些操作都在一个简单的环境下完成。用户可以从程序中直接看到汇编语言的语句，同时可以看到对应的等效机器代码（十六进制形式）。同时也显示出这些指令的地址，用户可以在任意时刻都观察到存储器的任何部分的内容。高亮度部分用来显示程序最初装入的地址，以及程序运行时正在执行的各条指令。寄存器和存储器采用图形显示，学生们可以观察到指令是如何导致寄存器和存储器中的数值发生改变。

如果你发现错误

我们力图使本书在技术上尽可能正确。尽管本书的手稿经过了反复校对，但错误是难免的。我们非常感谢读者对本书任何需要纠正的错误的反馈。欢迎读者将您的意见和建议发送至电子邮件 ECOA@jbpub.com。

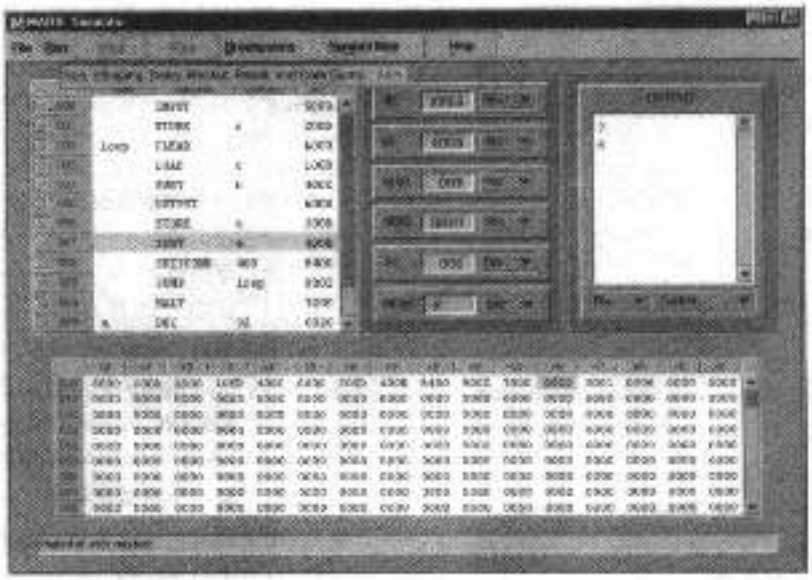


图 P.2 MarieSim 的图形界面

致谢

很少有书籍可以由一两个人单独完成，本书也不例外。现在，我们意识到编写教材是一项十分艰巨的任务，并且需要许多人的共同努力。这里，很难对帮助本书完成的每个人一一致谢。因此，如果由于我们的疏忽在下面的致谢有遗漏的话，在此深表歉意。

多年来选修计算机组成与体系结构课程的全体学生，都对本书的各个主题内容提供了非常有价值的反馈意见。我们特别感谢使用本教材初级版本的同学，感谢他们在发现错误时表现出来的宽容和细心。

许多人都仔细阅读了本书的手稿，并且提出了宝贵的意见。这里，要特别感谢 Merry Creel 和 Hans Royer。同样非常感谢花费时间和精力审阅本书的评论家，他们除了提出了许多很好的建议外，还保证了本书内容的质量，他们是：Victor Clincy（Kennesaw 州立大学）、Robert Franks（Centrel 学院）、Karam Mossaad（Texas 大学 Austin 学院）、Michael Schulte（Missouri 大学，St. Louis）、Peter Smith（CSU Northridge）、Xiaobo Zhou（Wayne 州立大学）。

这里，还要特别感谢 Karishma Rao，她为设计一个高质量的存储器软件模块付出了艰苦努力。

Jones 和 Bartlett 出版社的工作人员为本书出版做了出色工作，他们中的每一个人都应该得到特别的感谢，包括 Amy Rose、Theresa DiDonato、Nathan Schultz 和 J. Michael Stranz。

我个人，Linda Null，在此要感谢我的丈夫 Tim Wahls，这期间，他的日常生活中成为了“书的丈夫”，感谢他的非凡耐心，聆听有关本书的各种评论并且率先发表对书籍内容的各种意见，感谢他在我编写本书期间，承担了全部做饭之类等生活琐事。我个人认为我嫁了一个非常好的男人。这里还要衷心感谢我的导师 Merry McDonald，他教给我学习和教学的价值和快乐，并且全身心地投入到学习和教学工作中。最后，我要向 Julia Lobur 致以最衷心的感谢。没有她的参与，本书和本书使用的各种软件都不可能实现。

我个人，Julia Lobur，非常感谢我的合作伙伴 Marla Cattermole，她的耐心和真诚使得我可以完成本书的工作，她精湛的厨艺使我大饱口福，而她丰富的智慧更使我受益匪浅。她在事业上勤奋努力，争取更高的学位。她的行为不断鞭策我，不得有丝毫的松懈。我同样要向 Linda Null 表达我最深致的谢意，首先敬佩她在计算机科学教育领域无与伦比的热忱和投入，以及她对学生们的无私奉献。然后，感谢她给了我这样一个机会，与她一起分享创作本书那种难以言表的经历。

目　录

出版者的话
专家指导委员会
译者序
前言

第 1 章　引言 …… 1
1.1　概述 …… 1
1.2　计算机的主要组成部分 …… 2
1.3　计算机系统示例 …… 3
1.4　国际标准化组织 …… 7
1.5　计算机的发展史 …… 8
1.5.1　第零代:机械计算器 …… 8
1.5.2　第一代:真空管计算机 …… 9
1.5.3　第二代:晶体管计算机 …… 13
1.5.4　第三代:集成电路计算机 …… 15
1.5.5　第四代:超大规模集成电路计算机 …… 15
1.5.6　摩尔定律 …… 17
1.6　计算机的分层组织结构 …… 17
1.7　冯·诺伊曼模型 …… 19
1.8　非冯·诺伊曼模型 …… 21
本章小结 …… 21
深入阅读 …… 22
参考文献 …… 22
基本概念和术语复习 …… 23
练习题 …… 24
第 2 章　计算机系统中的数据表示方法 … 25
2.1　概述 …… 25
2.2　位置编码系统 …… 25
2.3　十进制数和二进制数之间的转换 … 26
2.3.1　无符号整数的转换 …… 26
2.3.2　分数转换 …… 27
2.3.3　以 2 的指数幂为基数的数制之间的转换 …… 29
2.4　带符号整数的表示方法 …… 29
2.4.1　符号幅值表示法 …… 29
2.4.2　补码体系 …… 32
2.5　浮点表示法 …… 36
2.5.1　一个简单的模型 …… 36
2.5.2　浮点算法 …… 38
2.5.3　浮点误差 …… 38
2.5.4　IEEE-745 浮点标准 …… 40
2.6　字符编码 …… 40
2.6.1　二进制编码的十进制数 …… 40
2.6.2　EBCDIC …… 41
2.6.3　ASCII …… 41
2.6.4　统一字符编码标准 …… 42
2.7　用于数据记录和传递的编码方式 … 44
2.7.1　不归零编码 …… 44
2.7.2　反转不归零编码 …… 45
2.7.3　相位调制编码 …… 45
2.7.4　频率调制编码 …… 46
2.7.5　运行长度限制编码 …… 46
2.8　错误检测与校正 …… 47
2.8.1　循环冗余码校验 …… 47
2.8.2　海明编码 …… 49
2.8.3　里德-所罗门编码 …… 53
本章小结 …… 54
深入阅读 …… 54
参考文献 …… 55
基本概念和术语复习 …… 55
练习题 …… 56
第 3 章　布尔代数和数字逻辑 …… 61
3.1　概述 …… 61
3.2　布尔代数 …… 61
3.2.1　布尔表达式 …… 62
3.2.2　布尔恒等式 …… 63
3.2.3　布尔表达式的化简 …… 63
3.2.4　反码 …… 64
3.2.5　布尔函数的表示方法 …… 65
3.3　逻辑门 …… 66
3.3.1　逻辑门的表示符号 …… 66
3.3.2　通用门电路 …… 67
3.3.3　多输入的门电路 …… 67
3.4　数字电路元件 …… 68

3.4.1 数字电路及其与布尔代数的相互关系 …… 68
3.4.2 集成电路 …… 69
3.5 组合逻辑电路 …… 69
3.5.1 基本概念 …… 69
3.5.2 典型的组合逻辑电路 …… 69
3.6 时序电路 …… 74
3.6.1 基本概念 …… 74
3.6.2 时钟信号 …… 75
3.6.3 触发器 …… 75
3.6.4 典型的时序逻辑电路 …… 77
3.7 电路设计 …… 79
本章小结 …… 79
深入阅读 …… 80
参考文献 …… 80
基本概念和术语复习 …… 81
练习题 …… 81
卡诺图专题 …… 87
3A.1 概述 …… 87
3A.2 卡诺图的描述和基本术语 …… 87
3A.3 利用卡诺图化简二变量函数 …… 88
3A.4 利用卡诺图化简三变量函数 …… 89
3A.5 利用卡诺图化简四变量函数 …… 91
3A.6 无关条件 …… 93
3A.7 小结 …… 94
练习题 …… 94
第4章 MARIE:简单计算机模型 …… 96
4.1 概述 …… 96
4.1.1 CPU的基本知识和组成原理 …… 96
4.1.2 总线 …… 97
4.1.3 时钟 …… 100
4.1.4 输入/输出子系统 …… 101
4.1.5 存储器组成和寻址方式 …… 102
4.1.6 中断 …… 104
4.2 MARIE …… 104
4.2.1 体系结构 …… 104
4.2.2 寄存器和总线 …… 105
4.2.3 指令系统体系结构 …… 106
4.2.4 寄存器传输表示法 …… 109
4.3 指令的执行过程 …… 111
4.3.1 取指-译码-执行周期 …… 111
4.3.2 中断和输入/输出 …… 111
4.4 一个简单的程序 …… 113
4.5 有关编译程序的讨论 …… 114
4.5.1 编译程序的功能 …… 115
4.5.2 为什么使用汇编语言 …… 116
4.6 MARIE指令集的扩充 …… 117
4.7 有关译码的讨论:硬件译码和微程序控制译码 …… 120
4.8 实际的计算机体系结构 …… 122
4.8.1 Intel体系结构 …… 123
4.8.2 MIPS体系结构 …… 126
本章小结 …… 127
深入阅读 …… 128
参考文献 …… 129
基本概念和术语复习 …… 130
练习题 …… 131
第5章 指令系统体系结构概览 …… 135
5.1 概述 …… 135
5.2 指令格式 …… 135
5.2.1 指令系统的设计 …… 135
5.2.2 小端和大端的位序问题 …… 136
5.2.3 CPU的内部存储机制:堆栈和寄存器 …… 138
5.2.4 操作数的数目和指令的长度 …… 138
5.2.5 扩展操作码 …… 141
5.3 指令类型 …… 142
5.4 寻址 …… 143
5.4.1 数据类型 …… 143
5.4.2 寻址方式 …… 144
5.5 指令流水线 …… 146
5.6 ISA体系结构的真实案例 …… 149
5.6.1 Intel体系结构 …… 149
5.6.2 MIPS体系结构 …… 150
5.6.3 Java虚拟机 …… 151
本章小结 …… 154
深入阅读 …… 154
参考文献 …… 155
基本概念和术语复习 …… 156
练习题 …… 156
第6章 存储器 …… 160
6.1 概述 …… 160
6.2 存储器的类型 …… 160
6.3 存储器的层次结构 …… 161
6.4 高速缓存存储器 …… 163
6.4.1 高速缓存的映射模式 …… 165

6.4.2 置换策略 …… 170
6.4.3 有效存取时间和命中几率 …… 171
6.4.4 何时高速缓存的方法会失效 …… 171
6.4.5 高速缓存的写策略 …… 172
6.5 虚拟存储器 …… 172
6.5.1 分页 …… 173
6.5.2 使用分页的有效存取时间 …… 177
6.5.3 综合举例:同时使用高速缓存、TLB 和分页 …… 179
6.5.4 分页和虚拟存储器的优缺点 … 180
6.5.5 分段 …… 181
6.5.6 分页和分段的组合方式 …… 181
6.6 存储器管理实例 …… 182
本章小结 …… 183
深入阅读 …… 183
参考文献 …… 184
基本概念和术语复习 …… 184
练习题 …… 185
第 7 章 输入/输出和存储系统 …… 189
7.1 概述 …… 189
7.2 AMDAHL 定律 …… 189
7.3 输入/输出体系结构 …… 190
7.3.1 I/O 的控制方法 …… 191
7.3.2 I/O 总线操作 …… 194
7.3.3 深入讨论中断控制的 I/O …… 196
7.4 磁盘技术 …… 198
7.4.1 硬盘驱动器 …… 199
7.4.2 软盘 …… 201
7.5 光盘 …… 202
7.5.1 CD-ROM …… 203
7.5.2 DVD …… 205
7.5.3 光盘记录方法 …… 205
7.6 磁带 …… 206
7.7 独立磁盘冗余阵列 …… 208
7.7.1 RAID Level 0 …… 209
7.7.2 RAID Level 1 …… 209
7.7.3 RAID Level 2 …… 210
7.7.4 RAID Level 3 …… 210
7.7.5 RAID Level 4 …… 211
7.7.6 RAID Level 5 …… 212
7.7.7 RAID Level 6 …… 212
7.7.8 混合 RAID 系统 …… 213
7.8 数据压缩 …… 214
7.8.1 统计编码 …… 215
7.8.2 Ziv-Lempel(LZ)字典系统 …… 221
7.8.3 GIF 和 PNG 压缩 …… 223
7.8.4 JPEG 压缩 …… 224
本章小结 …… 228
深入阅读 …… 228
参考文献 …… 229
基本概念和术语复习 …… 229
练习题 …… 230
选择磁盘存储器的实现专题 …… 234
7A.1 概述 …… 234
7A.2 数据传输模式 …… 234
7A.3 SCSI …… 235
7A.4 存储器的区域网络 …… 244
7A.5 其他的 I/O 连接 …… 245
7A.6 小结 …… 247
练习题 …… 248
第 8 章 系统软件 …… 249
8.1 概述 …… 249
8.2 操作系统 …… 249
8.2.1 操作系统的发展史 …… 250
8.2.2 操作系统设计 …… 254
8.2.3 操作系统服务 …… 255
8.3 保护环境 …… 258
8.3.1 虚拟机 …… 258
8.3.2 子系统和分区 …… 260
8.3.3 保护环境和计算机系统体系结构的发展进程 …… 261
8.4 编程工具 …… 263
8.4.1 汇编程序和汇编 …… 263
8.4.2 链接编辑器 …… 265
8.4.3 动态链接库 …… 265
8.4.4 编译器 …… 266
8.4.5 解释器 …… 269
8.5 Java:一种综合语言 …… 269
8.6 数据库软件 …… 274
8.7 事务管理器 …… 278
本章小结 …… 279
深入阅读 …… 280
参考文献 …… 281
基本概念和术语复习 …… 281
练习题 …… 282
第 9 章 可选择的体系结构 …… 284

9.1 概述 …… 284
9.2 RISC 计算机 …… 285
9.3 FLYNN 分类方法 …… 288
9.4 并行和多处理器体系结构 …… 290
9.4.1 超标量和 VLIW 体系结构 …… 291
9.4.2 矢量处理器 …… 293
9.4.3 互连网络 …… 293
9.4.4 共享存储器的多处理器 …… 296
9.4.5 分布式计算 …… 298
9.5 新的并行处理方法 …… 299
9.5.1 数据流计算 …… 299
9.5.2 神经网络 …… 301
9.5.3 脉动阵列 …… 303
本章小结 …… 304
深入阅读 …… 304
参考文献 …… 304
基本概念和术语复习 …… 306
练习题 …… 307
第 10 章 性能度量和分析 …… 310
10.1 概述 …… 310
10.2 基本的计算机性能方程式 …… 310
10.3 数学预备知识 …… 311
10.3.1 平均数的意义 …… 311
10.3.2 统计学和语义学 …… 315
10.4 基准 …… 316
10.4.1 时钟速率、MIPS 和 FLOPS … 316
10.4.2 综合基准：Whetstone、Linpack 和 Dhrystone …… 318
10.4.3 标准性能评估公司基准 …… 319
10.4.4 事务性能委员会基准 …… 322
10.4.5 系统仿真 …… 325
10.5 CPU 性能优化 …… 326
10.5.1 分支优化 …… 326
10.5.2 使用好的算法和简单的代码 … 328
10.6 磁盘性能 …… 331
10.6.1 性能问题 …… 331
10.6.2 物理性能 …… 332
10.6.3 逻辑性能 …… 332
本章小结 …… 335
深入阅读 …… 336
参考文献 …… 337
基本概念和术语复习 …… 338
练习题 …… 338
第 11 章 网络组织和体系结构 …… 341
11.1 概述 …… 341
11.2 早期的商业计算机网络 …… 341
11.3 早期的学术和科学网络：因特网的起源和体系结构 …… 342
11.4 网络协议 1：ISO/OSI 协议 …… 344
11.4.1 一个比喻 …… 344
11.4.2 OSI 参考模型 …… 345
11.5 网络协议 2：TCP/IP 网络结构 …… 348
11.5.1 IPv4 网际协议层 …… 348
11.5.2 IPv4 遇到的困难 …… 350
11.5.3 TCP …… 354
11.5.4 TCP 的工作原理 …… 354
11.5.5 IPv6 …… 357
11.6 网络组织结构 …… 360
11.6.1 物理传输介质 …… 360
11.6.2 网络接口卡 …… 364
11.6.3 转发器 …… 364
11.6.4 集线器 …… 365
11.6.5 交换机 …… 365
11.6.6 网桥和网关 …… 366
11.6.7 路由器和路由 …… 367
11.7 高容量数字链路 …… 373
11.7.1 数字分层体系 …… 373
11.7.2 ISDN …… 376
11.7.3 异步传输模式 …… 378
11.8 因特网的概况 …… 380
11.8.1 走进因特网 …… 380
11.8.2 遨游因特网 …… 385
本章小结 …… 385
深入阅读 …… 386
参考文献 …… 387
基本概念和术语复习 …… 387
练习题 …… 388
附录 A 数据结构和计算机 …… 391
术语表 …… 405
部分练习题答案和提示 …… 433

计算不再只是计算机的事情，它已经渗透到人类生活的方方面面……不难看到，计算机已经从巨大的空调房间移到了我们的眼前，又搬到我们的办公桌上，现在又置于我们的膝头，并装进我们的口袋里。不仅如此，所有这些变化还远远没有结束……就像自然规律一样，数字时代的进步是无法抗拒或者停止的……信息高速公路在今天可能是最令人眼花瞭乱的词语，但对于未来它可能只是一种保守的描述，未来信息技术的发展可能会远超出人们现在最大胆的预测……我们所期待的任何发明创造正发生在我们的身边和眼前。这些发明创造从本质上推动着我们的社会朝着更加数字化的方向发展。

Nicholas Negroponte 教授，美国麻省理工学院（MIT）媒体技术

第1章 引 言

1.1 概述

Negroponte 博士是众多将计算机革命视作自然力量的科学家之一。这种自然的力量可以将人类带入数字化时代，使人们可以征服许多已经困扰了人类多个世纪的难题，以及在解决这些原始难题时又出现的各种新问题。计算机已经把我们从许多繁琐乏味的日常工作中解放出来，这使得我们的集体创造潜能得以解放，从而又可以建造出更大、更好的计算机。

目睹计算机给人类社会所带来的科学和社会方面的深刻变化，人们最初很容易被计算机的复杂性所完全折服。而计算机的这种复杂性却来源于一些非常简单的思想。正是这些简单的思想带来了今天计算机的成就，并且它们也是未来计算机的基础。至于这些思想会在未来的计算机发展过程中起多大作用，目前还只是各种各样的猜测。可是在今天，正是这些简单的思想构成了当代计算机科学的全部基础。

相对计算机硬件设计而言，计算机科学家常常会更关注编写一些复杂的程序算法。当然，如果要使这些算法起作用，最终还是需要一台计算机来执行。有些算法过于复杂，以至于现在的计算机系统运行它们需要太长的时间。这类算法被认为在计算上是不可实行的（computationally infeasible）。当然，以现在计算机技术的创新速度来看，某些现在不可行的事件在将来是可能行的。但是，无论将来的计算机系统变得多么巨大或多么快，人们构想出的问题，总会超出这种机器所能解决的合理范围。

要了解一个算法为什么不可行，或者要了解一个可行的算法在执行时为什么会运行得太慢，必须从计算机的角度来看待这个程序。在试图优化要运行的程序之前，必须了解计算机系统是由什么构成的。如果一开始就不了解计算机的构造，却要对计算机系统进行性能优化，那就如同往汽车油箱里灌注万金油来调整汽车。如果这样处理后，汽车还能行驶，那真算是走运了。

程序优化和系统调优可能是人们学习计算机工作原理的最重要的动机。当然，还有许多其他的理由。例如，如果要编写一些编译程序，那么必须了解这些编译程序运行时所需的硬件环境。最好的编译程序要利用特定的硬件特性（例如流水线），以获取更快的运行速度和更高的工作效率。

如果要对一些大型复杂的真实世界的系统进行建模，那么就需要了解浮点算术如何工作及其实际工作方式。而如果要设计计算机外围设备，或者是外围设备的驱动程序，则必须知道某台特定计算机如何处理其输入/输出（I/O）的每个细节。如果工作内容涉及嵌入式系统，那么还要知道这些系统通常都是资源约束的（resource-constrained）。对时间、空间和价格权衡，以及 I/O 结构的理解，对我们的职业发展是至关重要的。

所有计算机专业人员都应该熟悉有关计算机基准测试（benchmarking）的一些基本概念，并能够解释和表示这些基准测试系统所产生的结果。从事硬件系统、网络或程序算法研究的专家们会发现，基准测试技术对于日常工作至关重要。负责采购硬件的技术经理们也会使用基准测试技术来帮助他们在给定的经费内购买到最好的系统，他们深知利用性能基准测试这种方法可以发现有利于特定系统的结果。

前面所列举的例证说明了这样一种思想，即在计算机硬件和计算机系统的软件部分及计算机编程

的各个方面之间存在着基本的联系。因此，作为计算机科学家，不管其技术领域如何，了解硬件和软件之间的相互作用是十分必要的。同时，也应该熟悉怎样把不同的电路和元部件组合在一起，来创建有效的计算机系统。为此，需要学习计算机组成（computer organization）。计算机组成所强调的是有关控制信号（即怎样控制计算机）、信号传递方式，以及存储器类型等问题。它涵盖了有关计算机系统的物理构成的各个方面。这部分的内容将有助于回答这样一个问题：计算机怎样工作？

另一方面，计算机体系结构（computer architecture）则集中讨论计算机系统的结构和行为，主要涉及的是程序员所熟悉的系统实现的逻辑方面的内容。计算机体系结构包括许多基本要素，如指令集和指令格式、操作码、数据类型、寄存器的数目和类型、寻址方式、主存储器的访问方式和各种输入/输出机制等。系统的体系结构会直接影响到各种程序的逻辑执行过程。学习计算机体系结构将有助于回答另外一个问题：怎样设计一台计算机？

对于特定的计算机来说，计算机体系结构就是其各硬件部分的组合，再加上其指令集体系结构（instruction set architecture，ISA）。ISA 是在机器上运行的所有软件和执行这些软件的硬件之间的协定接口。ISA 实现了人机对话。

计算机组成和计算机体系结构之间的区别很难清晰地加以界定。对于哪些概念属于计算机组成，哪些概念属于计算机体系结构，计算机科学和计算机工程领域的人员持有不同的观点。事实上，无论是计算机组成，还是计算机体系结构，都不是孤立的。它们之间相互关联、相互依赖。只有在全面了解这两部分内容之后，才能真正理解它们。对计算机组成和体系结构的理解，最终将引导人们对计算机和计算有更深入的了解，计算机和计算才是计算机科学的精髓。

1.2 计算机的主要组成部分

虽然我们很难区分哪些概念属于计算机组成，哪些概念属于计算机体系结构，但我们要说出计算机硬件问题会在哪里结束，软件问题从哪里开始是几乎不可能的。通常，计算机科学家们会使用某种计算机语言（比如 Java 或 C 语言）来编写一些计算机程序，以实现各种设计算法。也许大家会问，是什么使得这些算法程序运行呢？答案是，当然是另外一种算法！并且，又会有其他一种算法来执行和实现该算法。依此类推，一直向下到机器层，这种机器层可以看作是由电子设备实现的一种算法。因此，现代的计算机实际上就是执行其他算法的算法实现的过程。这种嵌套的算法链可以导出下面的原理：

> **硬件和软件等效原理：**任何可以利用软件实现的事情可以利用硬件来实现。反之，任何可以利用硬件来实现的事件也同样可以利用软件来实现[⊖]。

通常，人们可以设计一台专用计算机来执行的各种工作任务，比如文字处理，预算分析，或者是用来玩一个友好的 Tetris 游戏。同样，也可以编写一些程序来实现专用计算机的各种功能，如实现放置在汽车内或微波炉中的嵌入式系统的功能。有时，人们常常会看到这样的情形，一个简单的嵌入式系统会比一个复杂的计算机程序表现出好得多的性能。而在另外一些场合，人们又会发现，计算机程序却是首选方法。硬件和软件的等效原理说明，可以用不同的选择来实现相同的计算机功能。学习计算机组成和结构的知识将有助于做出最佳选择。

下面先从构成一个计算系统所必需的基本部件开始，讨论计算机的硬件问题。在最基本的层次上，计算机通常由下面三部分组成：

1. 用来解释和执行程序的处理器。
2. 用来存储数据和程序的存储器。
3. 与外界进行数据传输的机制。

本书将在随后的章节中详细讨论这三部分与计算机硬件有关的内容。

⊖ 这一原理没有强调等效任务执行的速度问题。一般来说，硬件实现的速度总是快得多。

如果能够从各分立部件的角度来理解计算机，那么就应该可以了解系统在任意时刻的行为，并且还可以知道怎样根据需要改变系统的行为。有时，似乎很难想像人和计算机系统有什么共同之处。但下面的比喻也许不算太牵强附会。设想一下，怎样利用一个坐在教室里的学生来展示计算机的三大部件：该学生的大脑对应处理器，所记的笔记代表存储器，而用来记笔记的笔就是输入/输出（I/O）机制。需要记住的是，人的能力要远远超过当今世界现有的、或在可见的将来能够建造的任何计算机系统。

1.3 计算机系统示例

本书将引入一些计算机的专用词汇。有些术语可能易于混淆，不够严密，或有些牵强附会。不过，经加解释之后，相信读者就可以明白这些概念。

作为讨论，下面来看一个计算机广告（见图 1-1）。这是大量计算机广告中的一个典型例子，它使用了“64 MB SDRAM”、“64-bit PCI sound card”和“32 KB L1 cache”等诸如此类的广告词语来对读者进行宣传推销。如果对此类术语没有一定的了解，很难判断购买广告中所推销的商品是否是明智之举，甚至也不太清楚该系统能否满足我们的需求。随着本书的介绍，你将会学习到那些隐藏在这类专业术语后面的基本概念。

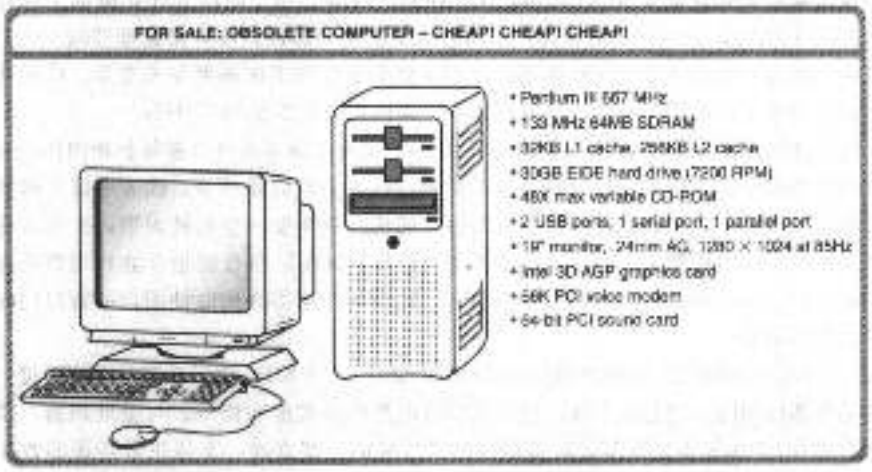


图 1-1 一个典型的计算机广告

在解释这个广告之前，我们还需要讨论一些更基本的概念：这就是我们在学习计算机知识的过程中将会遇到的有关测量方面的专门术语。

可以说，在科学技术的各个领域都有自己测量事物的方法，计算机领域当然也不例外。计算机行业的人们相互之间可以讨论某些事情有多大，或某些事情有多快。为此，他们必须使用相同的测量单位。在谈到一些计算机事物的大小时，经常会使用千、百万、十亿或万亿字符数来描述。图 1-2 中左侧给出了这些测量术语的前缀和它们所代表的意义。我们会看到，计算体系中以 2 为底的幂指数经常会比以 10 为底的幂指数更加重要。然而，对多数人来说，10 的幂指数更容易理解。因此，这些测量的前缀所表示的大小，在图中同时用 10 的幂指数和 2 的幂指数给出。因为 1000 与 2^{10}（1024）的值接近，可以近似地用 2 的 10 次方表示。在计算机测量的体系中使用的前缀通常采用以 2 为基的系统，而不是以 10 为基的系统。例如，一个千字节（KB）的存储器是一个典型的 1024 字节存储器，而不是 1000 字节的存储器。然而，1GB 的磁盘驱动器实际上可能是 10 亿字节，而不是 2^{30}（差很 10.7 亿）。因此，我们应该仔细阅读制造商印刷的参数说明来确定 1K，1KB 或 1GB 所表示的准确数值。

在讨论某些事情的快慢时，通常利用秒钟的分数来描述，如千分之一秒、百万分之一秒、十亿分之一秒，或万亿分之一秒。图 1-2 中右侧给出了这类有关快慢测量的前缀。注意：图中右侧的分数前缀与左侧的大小前缀的互为倒指数。因此，如果有人说一个操作需要百万分之一秒完成，那也就表示

在 1 秒内可以有一百万个该种操作发生。当谈及 1 秒内有多少次事件发生时，需要使用前缀 mega-。而在讨论操作执行快慢时，要采用前缀 micro-。

Kilo-(k)	(1 thousand=$10^3\approx2^{10}$)	Milli-(m)	(1 thousandth=$10^{-3}\approx2^{-10}$)
Mega-(M)	(1 million=$10^6\approx2^{20}$)	Micro-(μ)	(1 millionth=$10^{-6}\approx2^{-20}$)
Giga-(G)	(1 billion=$10^9\approx2^{30}$)	Nano-(n)	(1 billionth=$10^{-9}\approx2^{-30}$)
Tera-(T)	(1 trillion=$10^{12}\approx2^{40}$)	Pico-(p)	(1 trillionth=$10^{-12}\approx2^{-40}$)
Peta-(P)	(1 quadrillion=$10^{15}\approx2^{50}$)	Femto-(f)	(1 quadrillionth=$10^{-15}\approx2^{-50}$)

图 1-2 与计算机组成与体系结构有关的通用前缀

现在可以对上述广告加以解释说明：微处理器是计算机中实际执行程序指令的一个部件，是系统的大脑。广告中的微处理器为 Pentium III（奔腾 III），工作频率为 667MHz。每个计算机系统中都包含一个时钟，用来保证整个系统的同步。系统时钟同时向计算机的各个主要部件送出电脉冲信号，以确保数据和指令准确地按时到达指定的位置。系统时钟在每秒钟内发射的脉冲数目就是时钟的频率，时钟频率的度量单位为每秒周数，或称为赫兹（hertz）。因为计算机系统的时钟每秒产生几百万次的脉冲，也就是说计算机工作在兆赫兹（megahertz，MHz）范围。当今的许多计算机都工作在千兆赫兹（gigahertz，GHz）范围，即系统的时钟要产生每秒数十亿次的脉冲。没有微处理器的参与，计算机系统将无法工作。因此，微处理器的时钟频率的大小对整个系统的工作速度至关重要。广告中的微处理器的工作频率是每秒 677 兆周，所以销售商声称该计算机系统工作在 677 MHz。

然而，微处理器工作在 677 MHz 这个事实，并不一定表示该系统可以在每秒钟内执行 677 兆条指令，换句话说每条指令执行所需要的时间为 1.5 纳秒。在本书的后续章节，读者可以了解到每条计算机指令的执行需要若干个固定的时钟周期。有些指令的执行只需要一个时钟周期，但是大多数指令所需要时间都多于一个时钟周期。很明显，一个微处理器每秒钟实际执行的指令数目与微处理器的系统时钟的速度成正比（proportionate）。执行一条特定的机器指令所需要的时钟周期的数目同时也是机器组成与体系结构的函数。

接下来，广告中的参数是“133 MHz 64MB SDRAM”。133 MHz 表示系统的总线速度，系统总线是计算机内部各部件间的一组电学连线，用来在系统内部传递数据和指令。与微处理器一样，系统的总线速度也采用 MHz 来量度。许多计算机都有一条特殊的局部总线，支持非常高速的数据传递（例如视频信号传送所需要的高速数据传递），这种局部总线是直接连接处理器和存储器的高速通道。系统的总线速度最终成为制约系统的信息传载能力的上限。

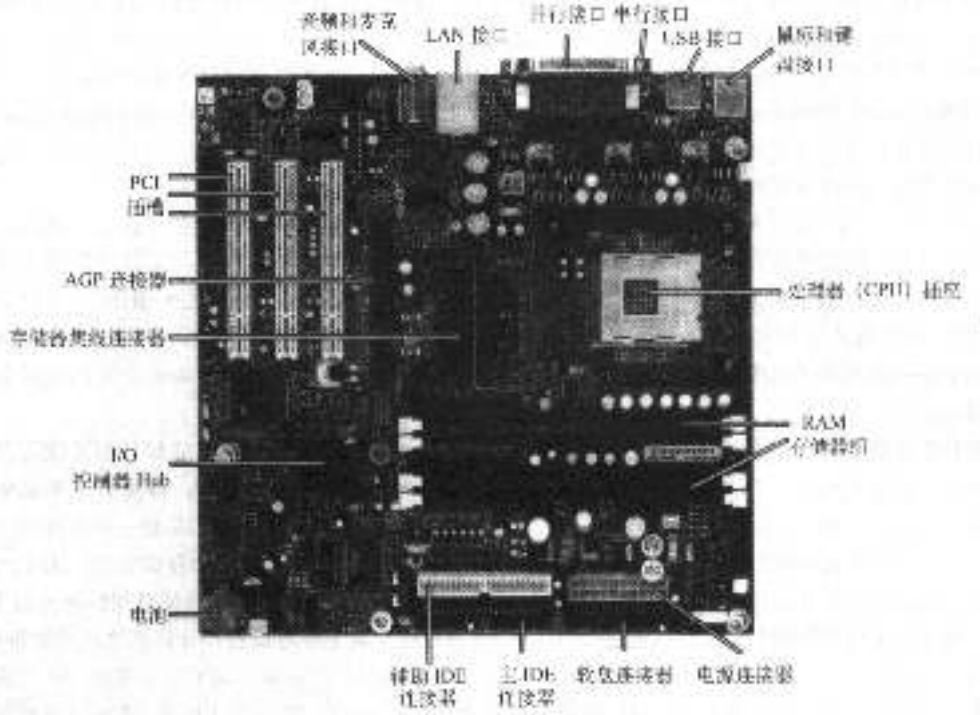
广告中的宣传还夸耀说，系统有 64 兆字节（MB）的存储容量，或者说存储 64 兆字符的容量。存储器的容量不但决定了可以运行的程序大小，而且也决定了可以同时运行且不会使计算机陷入停顿的程序数目。计算机的应用程序，或者操作系统制造商通常会对其产品的运行推荐所需要的内存大小（有时，这些推荐说明可能会过于保守，这就需要你仔细思考所看到的数据）。

除了介绍存储器的大小外，广告推销的系统还说明了存储器的类型，SDRAM，同步动态随机存储器（synchronous dynamic random access memory）。SDRAM 比传统（异步）的存储器要快得多，原因在于这类存储器可以与微处理器的总线同步运行。在这类写入操作中，SDRAM 总线只能与运行在 200MHz 和 200MHz 以下频率的总线实现同步。新的存储器技术，比如 RDRAM（Rambus DRAM）和 SLDRAM（SyncLink DRAM），则要求系统的运行速度要比总线速度快。

计算机内部结构一览

你想知道计算机内部到底是什么样子吗？下面举例给出了现代个人计算机（PC）的组成部分。然而，当打开计算机内部，试图寻找和辨认各个不同的计算机部件时，却是一件很让人头痛的事情，即使你已经熟悉了各个部件及其功能。

打开计算机的机箱，首先会看到一个装有风扇的金属盒，这是电源。我们还可以看到各种驱动器，



图片经过 Intel 公司的许可

包括硬盘驱动器，也许还有软盘驱动器和 CD ROM 或 DVD 驱动器。计算机内部包含许多集成电路，一些长着多条腿的黑色的小方块的电子元件。还会看到系统中有一些电器，或者说总线。一块较大的计算机电路主板安装在卧式 PC 机的底部，或者是立式 PC 机的侧面。机箱中还有其他一些印刷电路板（称为扩展板）插在主板的上面的各个插槽中。由印刷电路组成的主板通过总线把计算机中的所有部件连接在一起，包括 CPU，RAM 和 ROM 存储器，以及其他各种基本部件。对初学者来说，主板上的部件也最难辨认的。示例图为一个 Intel D850 主板的照片，其中的各主要部件都作了标注。

主板顶部的输入/输出（I/O）端口用来沟通外部世界与计算机之间的通信联系。通过 I/O 控制器的集线连接器，可以将计算机的多个部件连接起来，进行协调工作而不产生冲突。PCI 插槽可以插入不同 PCI 设备的扩展板，而 AGP 连接器则是为 AGP 图形卡的接插而设计的。主板上有两个 RAM 存储器排列区和一个存储器的集线连接器。照片中的主板上并没有接插处理器芯片，但有一个处理器（CPU）的插座。所有的计算机都有一个内置电池，如照片的左下角所示。主板上还有两个 IDE 连接器的插槽和软盘控制器接口。电源直接连接到主板的电源插座上。

在对计算机内部进行细致观察研究时，要注意一些警告。打开计算机箱盖之前，必须考虑安全因素，这对你和机器都很重要。可以采取下面的一些措施避免可能出现的各种危险。首先，也是最重要的，要确保打开计算机之前，机器的电源已经关闭。建议把电源插头留在插座上，这样可以提供一条释放静电的通路。另外，在打开计算机箱盖和接触里面的任何元件之前，应该先确认操作者的身体是否已经适当进行了接地处理，以保证身体的静电不会损害计算机内部的任何元件。机器的许多边缘部分，如箱盖和主板边缘，都很尖锐，操作时应该加以小心。在插入新的扩展卡，或是移走原有的扩展卡又重新安装时，都需要小心地将各种扩展卡插到所对应的插槽中，以避免损坏扩展卡和主板插槽。■

接下来的一行广告词，"32 KB L1 cache，256 KB L2" 所描述也是一种类型的存储器。在第 5 章

中，读者将会了解到无论总线的速度有多快，都需要一定的等待时间从内存中读取数据到微处理器。为了使读取数据的速度更快，许多计算机都设计了一种特殊的存储器，称为*高速缓存*（cache）。本书广告中的系统有两种类型的高速缓存：一级高速缓存（L1）是一个微处理器内置的较小的快速存储器，用来提高对经常使用的数据存取速度。二级高速缓存（L2）是一组位于主存储器和微处理器之间的快速存储器。注意，这里的高速缓存的存储容量是 KB 数量级，远小于主存储器。第 6 章将详细介绍高速缓存的工作方式，读者会看到大容量的高速缓存并不总是最好的。

另一方面，对于硬盘来说，却是希望容量越大越好。广告中的系统有 30GB 的硬盘，还是很有吸引力的。但是，硬盘的存储容量并不是唯一要考虑的因素。如果对于主机来说，硬盘的访问速度太慢的话，大容量的硬盘也无助于系统速度的提高。这里的硬盘驱动器的转速是 7200 RPM（单位：每分钟转数）。对于有这类知识的读者来说，该数据指出（虽然不十分明显），这是一个比较快的硬盘驱动器。除了利用转速来表示硬盘的快慢外，通常还可以采用访问磁盘上的数据平均所需要的毫秒数来描述硬盘的速度。

硬盘的转速是决定硬盘综合性能的关键因素之一。然而，硬盘与系统的其他部分的连接方式，或称*接口*，也是很重要的。广告中的系统硬盘采用的一种称为 EIDE 的硬盘接口，即*增强型集成驱动电子接口*（enhanced integrated drive electronics）。对于高密度存储设备而言，EIDE 是一种高性价比的硬件接口。EIDE 包含特殊的电子线路，可以用来增强计算机的可连接性、速度和存储容量。由于大多数的 EIDE 系统与微处理器和存储器共享系统总线，因此硬盘上的数据存取也与系统总线的速度有关。

系统总线负责计算机内部的数据传递，而*端口*（port）则承载了外部设备与计算机之间的数据传递。上面的广告描述了三种不同类型端口，“2 USB ports，1 serial port，1 parallel port”。大多数台式计算机都有两种数据端口：*串行端口*（serial port）和*并行端口*（parallel port）。串行端口通过 1 条或 2 条数据线发送电脉冲序列来传递数据。而并行端口至少有 8 条数据线，同时进行并行数据传输。该系统还配备了一种称为 USB，*通用串行总线*（universal serial bus）端口的特殊串行连接端口。USB 是一种流行的外部总线，它支持*即插即用*（plug-and-play）功能（即具备自动配置设备的能力）和*热插拔*（hot plugging）功能（可以在计算机运行的过程中增加或移除设备）。

有些系统的总线配备了专用 I/O 总线。*外围设备互连*（Peripheral Component Interconnect，PCI）就是这样一种专用的 I/O 总线，它支持多个外围设备的连接。由 Intel 公司开发的 PCI，可以高速连接外设，同时支持即插即用功能。该广告介绍了两种 PCI 设备：PCI modem（调制解调器）可以连接计算机和因特网（本书将在第 11 章详细讨论 modem）。另一种 PCI 设备是声卡，它包含了系统的立体声扬声器工作所需要的一些电路部件。第 7 章将会介绍更多不同类型的输入/输出接口（I/O）、I/O 总线和磁盘存储器。

在介绍完端口之后，广告还提供了有关显示器的特性，“19" monitor，.24mm AG，1280 × 1024 at 85 Hz”。虽然显示器对计算机系统的运行速度和工作效率作用很小，但是它对计算机使用者的舒适度却有很大影响。该显示器支持 85Hz 的*刷新速率*（refresh rate），这意味着显示器上显示的图像在一秒钟内刷新 85 次。如果刷新速率太慢，显示屏上的图像会出现令人不适的跳动或波纹现象。起伏不稳定的显示图像很容易造成操作人员的眼睛和身体的疲劳。有些人在长时间使用闪烁不定的显示屏后，甚至可能会产生头痛。造成眼疲劳的另一个原因是显示器的分辨率太低。高分辨率的显示器有着较清晰的视觉和更精细的图形。分辨率是按照显示器的像素点的间距来定义的，即两个最邻近的同色点（像素）之间的距离。像素点越小，图像就越清晰。广告中的显示器为*栅格式*（aperture grill，AG）结构，可以达到 0.24 毫米间距的像点。栅格式显示器的工作原理是直接引导电子束打击显示屏玻璃内侧涂敷的磷光粉，来绘制显示图像。AG 显示器可以产生比传统老式的荫罩技术更清晰的图像。显示器最后由一个 AGP（*加速图形端口*，accelerated graphics port）图形卡提供接口支持。这种 Intel 公司设计的图形接口特别适合于 3 D 图形显示。

按照前面的讨论，读者可能会有些奇怪：为什么显示器的像素点的间距不能制造成任意小的结构，

以获得理想的图像分辨率呢？原因在于显示器刷新速率与像素点的间距有关。例如，刷新 100 个像点与刷新 50 个像点相比需要更多的时间。对于较小的像点间距，同样显示屏就需要有更多的像素点。需要刷新的点越多，意味着每个刷新周期的时间就越长。专家们推荐刷新速率不小于 75Hz。本广告宣传的显示器的刷新速率为 85Hz，比专家推荐的最小值高 10Hz（大约 13%）。

尽管我们不可能对各种现有的特定品牌的计算机部件都做深入的研究，但在完成本书的学习后，读者可以了解到大多数计算机系统的工作原理。这些知识无论对于偶尔使用计算机的人士还是有经验的程序设计员来说，都是很重要的。作为用户，我们需要了解所使用的计算机系统的优点和局限，这样可以对计算机的应用做出一些明智的决定，可以更高效地使用计算机。而作为程序员，更需要准确地了解系统硬件的功能，才能更有效地编写各种功能程序。例如，即使对于计算机硬件中的一些简单的算法设计，例如主存储器与高速缓冲存储器之间地址映射的算法和存储器交叉编址的方法，在决定采用按以行为主，还是以列为主的次序访问数组元素时，硬件的结构都会对程序的设计产生巨大影响。

本书内容既涉及大型计算机，也研究了小型计算机。在大型机中包括企业级的服务器主机和超级计算机。而小型机则包含个人计算机、工作站和一些便携式计算机设备。大家将会看到无论是从事日常的琐碎工作，还是进行复杂的科学任务，这些计算机系统的主要部件都是非常相似的。本书也会介绍一些主流计算机系统之外的计算机结构，并衷心希望本书介绍的知识可以作为读者在计算机组成与体系结构这一振奋人心的广阔领域中继续学习的一个跳板。

1.4 国际标准化组织

假设你想购买一台精巧的 0.24 点距的 AG 显示器。你可以多访问几家商店，以便买到一个好价格。设想一下，先打几个电话查询一下价格，然后上网浏览几家网上商店，或者开车在城里转一转，一定可以找到称心如意的产品。以你的经验来看，在任何地方都可以买到非常适合于你的系统的显示器。之所以可以做出这样的假设，是因为计算机设备制造商们都遵守政府和工业组织所建立的设备互连和操作规范。

某些标准制定组织是一些特殊的行业协会，或者是一些由行业领头企业组成的公会。制造商们都很清楚，通过建立特定类型的设备的共同准则，可以比他们单独提出的设备制造标准有更大的市场，各自分立制定的技术规范很可能是互不兼容的。

有些标准化组织有自己正式的规章，并且在计算机和电子技术的某些领域是国际公认的权威性组织。继续学习计算机组成与体系结构的知识，无疑会遇到这些国际标准化组织和团体制定的各种技术规范，所以有必要对这些组织做一些介绍。

电子与电气工程师协会（Institute of Electrical and Electronic Engineers，IEEE）是致力于电子工程和计算机工程专业进步的组织。IEEE 积极推动全球范围内工程领域的合作，出版了大量的技术文献。IEEE 也为各种计算机部件，信号传送协议和数据表示方法制定标准，并对组织所涉及的几个领域进行各种技术命名。对于新标准的创建，IEEE 有一套民主的，却有些繁琐的组织程序。IEEE 所发表的有关标准的最终文件一般都受到广泛认同。并且，在文件更新再版之前，这些标准会持续使用许多年。

国际电信联合会（International Telecommunication Union，ITU）设立在瑞士的日内瓦，其前身也称为电话电报国际咨询委员会（简称 CCITT）。ITU 组织关注通信系统的协同性，包括电话、电报和数据通信系统。ITU 已经建立了许多通信方面的标准，以后的学习中将会遇到这类的一些标准，它们都是以 ITU-T，或者是机构的前身 CCITT 为前缀开头的。

世界上的许多国家，包括欧共体，也建立了一些自己主导的组织，来代表他们在国际上不同团体中的利益。代表美国的组织是美国国家标准学会（American National Standards Institute，ANSI），而英国有自己的英国标准协会（British Standards Institution，BSI），并且，在欧洲标准化委员会（Comite Europeen de Normal，CEN）中也有自己一席之地。

国际标准化组织（International Organization for Standardization，ISO）则是一个协调世界范围内

各种标准化发展的机构，包括协调 ANSI 和 BSI 在其他组织中的活动。ISO 不是其组织名称中各词语的首写字母的缩写，而是源于希腊词，“isos” 意思是 “相等的事物”。ISO 由 2800 个技术委员会组成，它们各自负责一些全球性的标准化问题，涉及的范围覆盖了从照相胶片的性能到螺纹的间距，直到计算机工程的复杂领域。ISO 促进了全球贸易的迅速扩展，实际上已经触及到人们生活的各个方面。

本书会在需要之处提到各个正式标准的名称。至于大部分标准的权威细节可以查阅相关组织的标准化网站。很多标准中还包含大量与标准有关的参考资料，提供一些与该标准相关的背景知识，读者将会得到额外收获。

1.5 计算机的发展史

在过去的 50 年，计算机的发展给现代人类生活带来了极大的方便。生活的记忆也许会让人回忆起过去那些使用速记、复写纸和油印机的年代。有时想起来，计算机这种魔术般的计算机器似乎是突然按照现在人们所熟知的形式就出现了。然而，计算机的发展过程充满了偶然性的发现、商业利益的驱动和各种千奇百怪的念头。有时，计算机却是通过纯粹的工程实践而获得改进。尽管在计算机的发展史上经历了各种各样的扭曲、反复和技术上的绝境，但是计算机还是在以令人难以置信的步伐不断地向前发展。只有在了解了计算机的发展史后，我们才有可能全面地评价今天计算机所取得的成就。

下面，我们按照制造计算机的技术把计算机的发展划分成若干个阶段（或称为代，generation）。这里，我们只给出各个发展阶段的近似年代时间，仅供读者参考。事实上，对于每个技术时代起始的准确时间，即使是专家们也很难达成一致的看法。

历史上的每一项发明都带有当时的时代气息。如果计算机是发明在 90 年代的话，人们也许会奇怪，它是否还会被称为计算机。事实上，我们到底发现那些放在桌上或桌边的神秘的盒子做了多少计算工作呢？在不久以前，计算机还只是用来为人们解决令人头疼的数学运算问题。今天，计算机已经不再只是为那些穿着白大褂的科学家所独用。现在的计算机能够帮助人们书写文件，帮助相隔千里的恋人进行通信，以及帮助人们完成购物等家务事。现代的商用计算机也只是花费极少的一部分时间来进行有关财务账目的数学计算。这些机器的主要目的是为用户提供大量商业策略方面的信息，以取得竞争上的优势。那么计算机这个词现在是不是变成了用词不当？或者说是时代的错误呢？如果不叫计算机，又怎样称呼它呢？

我们不可能用几页纸就完整地陈述计算机的发展史。关于这个话题已经出版了好几本书。尽管如此，读者们还是希望了解某些更细节的内容。如果感兴趣的话，我们建议读者阅读列在本章后面的参考书目录中的一些书籍。

1.5.1 第零代：机械计算器

在 16 世纪以前，欧洲的商人通常使用算盘进行各种计算，而用罗马数字来记录计算的结果。后来，十进制的计数方法最终取代了罗马数字。历史上曾经有许多人发明了各式各样的计算工具，目的就是要使十进制数的计算工作变得更加快速精确。Wilhelm Schickard（1592—1635）由于第一个发明了一种机械式的计算器（1642—1945），称为计算钟（Calculating Clock，具体发明日期不详），而一直倍受人们的赞许。这种计算钟能够进行多达 6 位数字的加法和减法运算。法国数学家帕斯卡（Blaise Pascal，1623—1662）也研制了一个称为 Pascaline 的机械式计算器，用来帮助他的父亲进行税收工作。Pascaline 可以进行带进位的加法和减法运算，它可能是第一个有实用目的的机械式加法机器。事实上，Pascaline 的构思非常巧妙，一直到了 20 世纪初期，其基本设计思想还在被人们所使用。这里的例证就是在 1908 年面世的闪电便携式加法器（Lightning Portable Adder）和 1920 年设计的加法仪（Addometer）。著名的德国数学家莱布尼兹（Gottfried Wilhelm von Leibniz，1646—1716）发明了一个被称为步进式计算器（Stepped Reckoner）的计算工具，可以进行加、减、乘、除四种运算。但是，所有的这些计算设备和工具都不能进行编程计算，也没有存储器。在整个计算过程中的每一步，它们都需要人的

手工参与才能进行。

尽管像 Pascaline 这类的计算机器一直使用到 20 世纪，但在 19 世纪时就已经开始出现了新型计算器的设计。在这些新型的设计中，最引人注目的是巴贝奇（Charles Babbage，1791—1871）设计的差分机（Difference Engine）。直到现在，还有人称 Babbage 为“计算之父”。据说 Babbage 是一个行为古怪的天才。当时他还给人们带来了他的其他发明：万能钥匙（skeleton key）和捕牛器（cow catcher），一种用来将牛和一些可移动的障碍物推出机车轨道的装置。

1822 年，Babbage 制造了差分机。差分机之所以得名是因为它使用了一种被称为差分方法（method of difference）的计算技术。设计这种机器的目的是用来对多项式函数进行机械式的求解。实质上，差分机本身是一个计算器，而不是计算机。1833 年，Babbage 还设计了一种多用途的机器，称为分析机（Analytical Engine）。尽管 Babbage 生前未能建造出他所设计的分析机，但是如果从设计构造上来说，分析机的功能要比早期的差分机更加通用。分析机已经具备了执行任意类型的数学运算的能力。分析机还包含现代计算机的许多部件：一个算术处理部件进行计算工作（Babbage 称这部分为运算逻辑部件，mill），一个存储器（store），以及输入和输出设备。Babbage 的设计中还包含一个条件转移操作，即要执行的下一条指令由前一个操作的结果来决定。当时，著名诗人拜伦（Lord Byron）的女儿，Lovelace 伯爵夫人艾达（Ada）曾建议 Babbage 编写一个分析机进行数学计算的原理方案。这个方案被公认为是第一个计算机程序，而 Ada 则被视为是第一个计算机的程序员。不仅如此，据说 Ada 还提出了采用二进制的数字体系来进行数据的存储，而不是十进制的数字体系。

长期困扰着早期的计算机设计者的一个难题是怎样把数据输入到计算机器中。Babbage 设计的分析机是采用一种穿孔的卡片来进行数据输入和编程。使用卡片来控制机器的思想并不是 Babbage 原创的，而是来自他的一位朋友 Joseph-Marie Jacquard（1752—1834）。1801 年，Jacquard 发明了一种可编程的织布机器，可以在布料上生成各种复杂的图案花样。Jacquard 曾经把一张用他自己的织布机织成的挂毯送给了 Babbage，完成挂毯的编织共使用了超过 10000 张的穿孔卡片。对于 Babbage 来说，看到一台织布机可以由卡片来控制，那么想到也可以用卡片来控制他的分析机则是一件十分自然的事情。Ada 也对这种想法感到很高兴，她曾经写到：“Babbage 的分析机可以求解各种各样的代数方程，就如同 Jacqard 的织布机可以编织出姿态万千的花卉图案一样”。

后来的事实证明了穿孔卡片是应用时间最长的计算机输入方法。直到后来，构建计算机器的方法发生了根本性的变化，才出现了键盘输入方式。19 世纪后半期，大部分计算机器都是采用转轮式机制来输入数据。这种转轮式机制很难与早期的打字键盘整合在一起，其原因是老式的机械键盘都采用杠杆式结构。有趣的是，杠杆式结构的设备非常适合于在卡片上穿孔，而转轮式的设备又可以很方便地阅读这些卡片上的穿孔。当时，人们发明了各种各样的设备，用来对穿孔卡片数据进行编码，然后“制表”（tabulate）。19 世纪后期最重要的穿孔卡片制表机是由美国人霍尔瑞斯（Herman Hollerith，1860—1929）发明的。霍尔瑞斯的制表机曾用来对 1880 年美国的人口普查数据进行编码和编辑处理。这次人口普查的完成时间创下记录，为霍尔瑞斯带来高额利润，他的发明也因此名声大振。霍尔瑞斯后来创建了一家公司，这就是著名的 IBM 公司的前身。他发明的 80 行穿孔卡片，即霍尔瑞斯卡片，作为当时的自动数据处理的主要产品流行了 50 多年。

1.5.2 第一代：真空管计算机

尽管 Babbage 常常被称为“计算之父”，但他所设计的计算机器是机械式的，而不是电动的或者电子的。在 20 世纪 30 年代，德国人 Konrad Zuse（1910—1995）重新拾起 Babbage 的机器，并在 Babbage 的设计中加入了电学技术和其他一些改进，制造出一台电动计算机。Zuse 的计算机称为 Z1，采用电机械式继电器，而不是 Babbage 机器中的手动曲柄齿轮。Z1 是可编程的，并且配有一个存储器，一个算术部件和一个控制部件。当时德国正处在战争时期，资金和资源都十分匮乏。Zuse 只能用废弃的电影胶片来替代穿孔卡片作为 Z1 的输入方式。Zuse 的机器设计采用的是真空管，但是 Zuse 在独自建造这

些机器时，却买不起真空管。因此，尽管 Z1 中没有使用真空管，但它确实属于第一代电子计算机。

Zuse 是在柏林他父母的起居室里建造了 Z1 系统，而此时，德国正在与欧洲的大部分国家开战。值得庆幸的是，Zuse 没能说服纳粹政府购买他的机器，因为纳粹当局还并没有意识到这种计算设备有可能给他们带来战术上的优势。盟军的轰炸摧毁了 Zuse 计算机器的首批全部三台系统：Z1、Z2 和 Z3。Zuse 的机器的确令人印象深刻。但是，一直到战争结束，Zuse 都没有机会重建或进一步改进他的计算机。战后，Zuse 的设计却由于走进了计算机历史上的“另一个进化过程中的死胡同”而终止了。

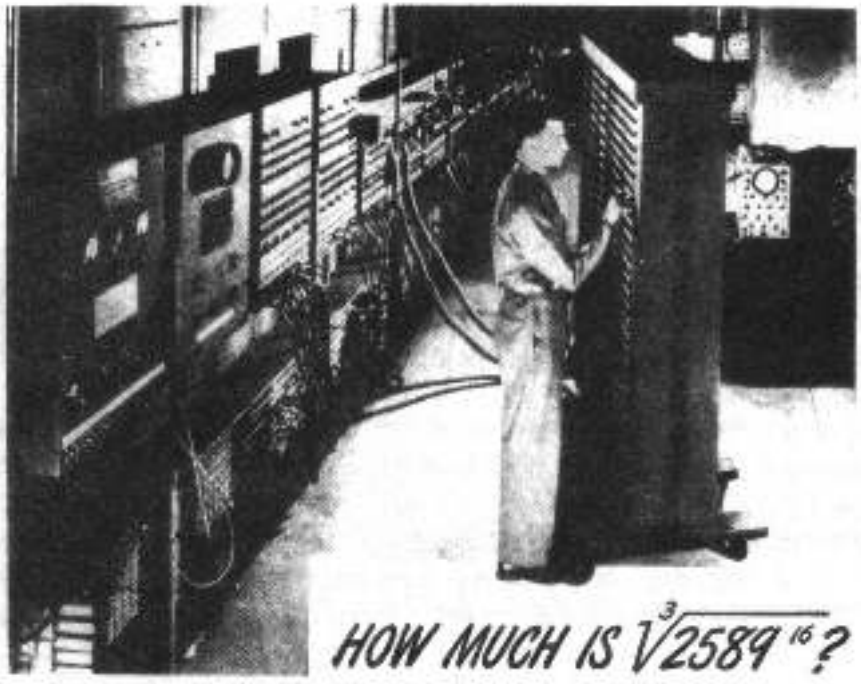
今天人们所熟知的数字计算机，是许多科学家在 20 世纪 30 年代到 40 年代时期的工作成果。这期间，不少人设计了基于 Pasical 基本原理的机械式计算器，并对模型进行了大量修改。与此同时，科学家们在现代电子计算机的研究方面也做了大量工作。人们对于谁才是现代计算机的第一人，一直存在着各种争议。但是，在现代计算机的发明者中，有三个人的地位是不容置疑的。他们是：John Atanasoff、John Mauchly 和 Presper Eckert。

John Atanasoff（1904—1995）建造了第一台完整的电子计算机系统，并得到了大家的公认。Atanasoff Berry 计算机（简称 ABC）是一台用真空管建造的二进制机器。由于这台机器是为了求解线性方程系统而专门建造的，所以它不能称为通用计算机。然而，ABC 计算机的某些特性与几年后建造的通用计算机 ENIAC（又称为电子数字积分器和计算机，Electronic Numerical Integrator and Computer）是相同的。由于这两种机器所具有的这些共同特性，在对于电子数字计算机的发明权和专利权应该授予谁的问题上，曾经引起极大的争论。对于这种争议，有兴趣的读者可以阅读 Mollenhoff［1988］的著作，其中详细描述了一桩有关 Atanasoff 和 ABC 计算机的马拉松式的长期法律诉讼。

John Mauchly（1907—1980）和 J. Presper Eckert（1929—1995）是在 1946 年公布于众的 ENIAC 计算机的两位主要发明人。ENAIC 被公认为世界上第一台全电子通用数字计算机。这台机器共使用了 17 468 个真空管，占地面积 1 800 平方英尺，重 30 吨，耗电达 174 千瓦。ENIAC 有一个存储容量为 1000 信息比特（可存储大约 20 个 10 位的十进制数字）的存储器，并且采用穿孔卡片来存储数据。

John Mauchly 毕生都对利用数学方法来预测天气有极大的兴趣，这种兴趣导致他产生了想建造电子计算机器的念头。当他在费城（Philadelphia）附近的 Ursinus 学院担任物理学教授时，Mauchly 就使用几十台机械式的加法机器并雇佣学生操作员来处理成堆的数据，他一直相信这些数据可以揭示各种天气类型背后所隐藏的数学关系。他当时觉得如果可以再增加一些计算能力的话，就可以实现他的目标。表面看来，这一目标似乎就在他现在所掌握的那些东西的后面。本着结合进行研究工作和学习电子计算的目的，Mauchly 自愿到宾西法尼亚大学的 Moore 学院承教一门电气工程方面的课程。在完成这个教学任务后，Mauchly 接受了 Moore 学院的一个教学职位。在这里，他教授了一位非常有才华的学生，J. Presper Eckert。Mauchly 和 Eckert 发现他们在建造电子计算机方面有着共同的兴趣。为了确保建造他们设计的计算机所需的资金支持，他们书写了一份正式的建议书送到学校去参加评审。在建议书中，他们对所设计的机器做了非常保守的描述，把它说成是一台“自动计算器”。当时，他们或许已经知道采用二进制的计数系统可以使计算机能够最有效地工作。但是，为了与其他正在建造中的大型电子加法器保持形式上的一致，Mauchly 和 Eckert 最终还是把他们的机器设计成为一个使用以 10 为基数的数字体系。宾西法尼亚大学最后拒绝了 Mauchly 和 Eckert 的建议书。非常幸运的是，美国的军方却对他们的计划更感兴趣。

在第二次世界大战期间，美国的军队非常迫切地需要对他们设计的新型火炮的弹道进行计算。当时，军方雇佣成百上千的人力“计算机”。许多人都在使用由时钟转杆组成的机械式的计算机器，对火炮的射击表进行各种各样算术计算。当得知电子计算机可以将弹道表的计算时间从几天缩短为几分钟时，军方决定资助 ENIAC 计划。事实上，ENIAC 的确把计算一个弹道表的时间从 20 小时缩短为 30 秒钟。但不幸的是，直到战争结束，这台机器都未能建成。然而，ENIAC 已经证明了真空管计算机的运行速度很快，并且切实可行。在接下来的 10 多年中，人们对真空管的计算机系统不断改进完善，并且在商业上取得了成功。



HOW MUCH IS $\sqrt[3]{2589^{16}}$?

The Army's ENIAC can give you the answer in a fraction of a second!

Think that's a stumper? You should see some of the ENIAC's problems! Brain twisters that if put to paper would run off this page and feet beyond . . . addition, subtraction, multiplication, division—square root, cube root, any root. Solved by an incredibly complex system of circuits operating 18,000 electronic tubes and tipping the scales at 30 tons!

The ENIAC is symbolic of many amazing Army devices with a brilliant future for you! The new Regular Army needs men with aptitude for scientific work, and as one of the first trained in the post-war era, you stand to get in on the ground floor of important jobs which have never before existed. You'll find that an Army career pays off.

The most attractive fields are filling quickly. Get into the swim while the getting's good! 1½, 2 and 3 year enlistments are open in the Regular Army to ambitious young men 18 to 34 (17 with parents' consent) who are otherwise qualified. If you enlist for 3 years, you may choose your own branch of the service, of those still open. Get full details at your nearest Army Recruiting Station.

YOUR REGULAR ARMY SERVES THE NATION AND MANKIND IN WAR AND PEACE

A GOOD JOB FOR YOU

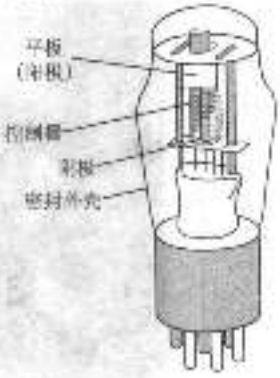
U. S. Army

CHOOSE THIS FINE PROFESSION NOW!

什么是真空管

今天看起来结构错综复杂，导线纵横交错的电子计算机系统，最初源于一个被美国人称为真空管（vacuum tube）的简单的电子器件的发明。更确切地说，英国人称其为阀门（valve）。真空管在当时被称之为阀门是非常合理的，因为真空管可以控制电子系统中电子流动，其工作方式类似于水管装置中的阀门控制水流。事实上，20 世纪中叶的许多电子管产品根本就不是真空产品。为了满足电学特性的需要，人们通常在这些电子器件里面充满了一些导电的气体（例如汞蒸汽）。

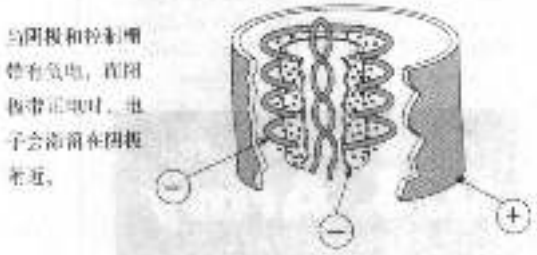
与电子管工作原理有关的电学现象是在1883年由美国的发明家爱迪生（Thomas A. Edison）发现的。当时，爱迪生正在试图寻找某种方法，解决他发明的电灯泡所遇到的一个难题，即灯泡的灯丝在通电几分钟后就会被烧断（或者说被氧化）。爱迪生成功找到了一种避免灯丝氧化的方法，这就是把灯丝放入真空中。当时，爱迪生并没有马上意识到空气不但可以助燃，而且是一种很好的绝缘材料。当他给一个新装的用钨材料制成的灯丝电极通电时，灯丝很快就开始发热，并和以往其他的灯丝一样被烧断。然而这一次，爱迪生注意到灯泡中的电流还在继续地从加热的负极向没有加热的正极流动。1911年，英国物理学家理查森（Owen Willans Richardson）对这种现象进行了分析，并得出了结论：当一个带负电的电灯丝被加热时，电子就会像水分子沸腾一样，产生电子蒸汽。他把这种现象恰当地命名为热电子发射（thermionic emission）。

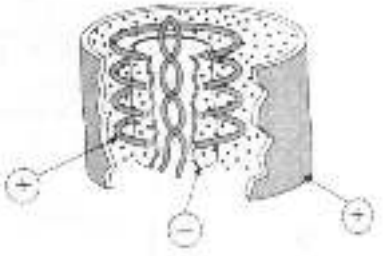


对于爱迪生发现的热电子发射现象，许多人都认为这只不过是一个电学上的奇怪现象而已。然而，爱迪生的前助手，英国科学家弗莱明（John A. Fleming）却从爱迪生的发现中看到了更多的东西。他注意到这种热电子发射产生的电子流是单方向的，也就是说电子是从带负电的阴极（cathode）流向带正电的阳极（anode，或者 plate）。他立即意识到可以利用这种特性对交流电进行检波（rectify），即将交流电转变成直流电。在当时，直流电对于电报设备的工作是必不可少的。于是，弗莱明将他的想法付诸实践，发明了一种后来称为二极管（diode），或者称为整流器（rectifier）的电子器件。

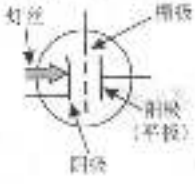


这种二极管在当时很适合用来把交流电转换成直流电。然而，这并不是电子真空管的最大功能。1907年，美国人 Lee DeForest 在这种二极管中加入了一个被称为控制栅（control grid）第三极。如果控制栅带有负电时，可以减少和阻止电子从二极管的阴极流向阳极。





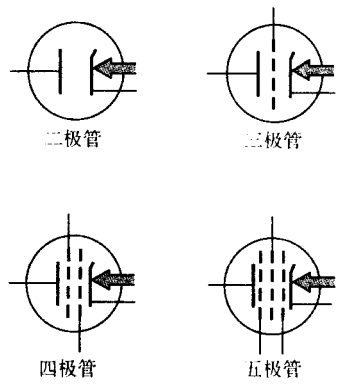
当 DeForest 在为他的发明申请专利时，将其称为三极电子管（audion tube），后来称为三极真空管（triode）。三极真空管的原理图如右所示。



这种三极真空管既可以用作一个开关，又可以用作一个放大器。在控制栅上电荷的一个微小的变化，就会引起阴极和阳极之间电流的一个非常大的变化。因此，只要在控制栅上施加一个微弱的电信号，就可以在阳极上产生一个很强的信号输出。如果在控制栅上施加足够多的负电荷，就能够阻止所有的电子离开阴极。

后来，人们又在三极真空管中再次加入了另外一个控制栅，这样可以对电子流实现更精确的控制。这种有两个栅极的真空管称为四极真空管（因为管内共有四个电极，tetrode），而有三个栅极的真空管

就叫做五极真空管（pentode）。三极真空管和五极真空管在通信和计算机中的应用最为普遍。通常，人们还将两三个三极真空管、或五极真空管组合起来封装在一个玻璃真空管内，这样它们可以共用一个加热器，也就可以减小器件的功耗。这些后代类型的真空管被称为微型管（miniature），因为它们的尺寸只有大约 2 英寸（5 厘米）高，直径约为半英寸（1.5 厘米）。而类似的全尺寸二极真空管、三极真空管和五极真空管仅仅比家用电灯泡稍微小一点。



后来，真空管变得越来越不适合用来制造计算机系统。即使最简单的真空管计算机也需要安装数千个真空管。加热这些真空管的阴极需要消耗巨大的电能。为了避免热熔化现象，真空管中的热量必须要尽快从系统中散去。如果我们让阴极的加热器以较低的电压工作，显然可以减小真空管的功耗和热耗散现象。但是这样一来，也会降低本来就比较慢的真空管的开关速度。尽管存在功耗和其他种种限制，但无论是模拟真空管计算机，还是数字真空管计算机，都为人们服务了许多年，并且它们是各种现代的计算机系统的体系结构基础。

现在，距离最后一台真空管计算机已经过去了几十年，但是人们仍然还在音频放大器中使用真空管。这些所谓“高端”放大器特别受到音乐家们的青睐，因为他们相信真空管所提供的带有回旋共振的美妙声音是使用固体器件的音响设备所无法获得的。■

1.5.3　第二代：晶体管计算机

事实上，第一代真空管技术的计算机并不是非常可靠。曾经有一些 ENIAC 的批评者认为这种系统永远不可能正常运行，理由是这些真空管被烧坏的速度实在是太快了，以至于人们来不及替换它们。虽然 ENIAC 系统的可靠性并不像批评者所预期的那样糟，但是真空管计算机系统的通常停机维修时间的确要比它们正常运行的时间还要长。

1948 年，贝尔实验室的三位研究员巴丁（John Bardeen）、布拉顿（Walter Brattain）和肖克利（William Shockley）发明了晶体管。这种新型的技术不但掀起了电子器件、电视和无线电广播等领域的革命，而且推动着计算机的发展进入了一个新的时代（1954—1965）。与真空管相比，晶体管体积更小，功耗更低，而且工作性能更加可靠。这样一来，晶体管计算机中的电子线路也随之变得更加小型化并具有更高的可靠性。尽管使用了晶体管，这一代的计算机系统的体积还是过于庞大且价格也相当昂贵。通常只有一些大学、政府机关和大型商业机构才买得起这种计算机。值得一提的是，这期间涌现了大量的计算机制造商。其中，IBM 公司、数字仪器公司（DEC）和 Univac（现在称作 Unisys）等公司主导了当时的计算机产业。据统计，IBM 公司在这期间成功销售了 7094 台作为科学应用和 1401 台作为商业应用的晶体管计算机。DEC 公司在当时则忙于建造 PDP-1 型的计算机系统。一个由 Mauchly 和 Eckert 创建的公司（不久便被卖掉）主要制造 Univac 品牌的计算机系统。在这一代计算机中，最成功的 Unisys 系统当属它的 1100 系列产品。另外还有一家公司，控制数据公司（CDC），则在 Seymour Cray 的领导下，建造了世界上第一台超级计算机 CDC6600。这台价格为 1000 万美元的 CDC6600 超级计算机每秒钟可以执行 1000 万条指令，采用 60 位（bit）字长，并且拥有在当时令人吃惊的 128K 字节的主存储器。

什么是晶体管

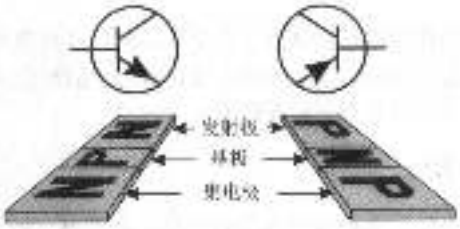
晶体管（transistor）为英文单词转移电阻（transfer resistor）的缩写。事实上，晶体管是一种利用固体器件制造的真空三极管。但是，现实中却没有固体器件形式的真空四极管或真空五极管。由于电子在固体介质中的运动比在真空管的开放空间中更容易控制，所以在固体电子器件中不再需要额外的控制电级。化学元素锗和硅都能够用作固体电子器件中的基本“固体”介质。纯净状态的锗和硅都

是电学上的不良导体。但是，如果它们与元素周期表中的某些邻近元素结合，就可以变成良导体。而且，我们可以十分方便地控制它们的导电性能。



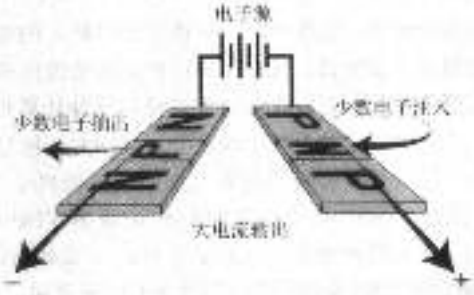
铝、铟和镓各元素周期表中位于硅和锗的左边。这些元素的外层电子壳层（或称为价带，valence）的电子数要比硅和锗元素的外层电子数少一个。如果我们将少量的铝元素添加到硅材料中，就会造成硅的外层电子壳层的其他不平衡。这时，铝原子会从周围的硅原子上吸引一个电子，形成有一个带有过剩电子的负电荷中心。利用这种方式对材料进行调剂（我们称之为掺杂，doped），硅或锗材料就变成了所谓P型（P-type）半导体材料。

类似地，如果在硅材料中加入少量的砷、磷或锑元素，硅材料晶体的价带将获得额外的电子，就变成N型（N-type）材料。在N型半导体中，这些束缚较弱的额外电子可以自由移动，形成小量电流。换句话说，如果在N型材料上施加一个正电压，电子将会从负极流向正极。如果施加反向外加电压，即给N型半导体加负电压，而P型半导体加正电压，材料中就不会有电流产生。这就是说，我们可以利用一个简单的N型和P型半导体结制造一个固体二极管。



固体三极管（即晶体管）由三层半导体材料组成。我们将一片P型材料夹在两层N型材料之中，或者是将一片N型材料夹在两层P型材料之中，组成所谓的三明治结构。前者称为NPN型晶体管，后者称为PNP型晶体管。晶体管结构的中间层称为基极，而其他两层分别称为发射极和集电极。

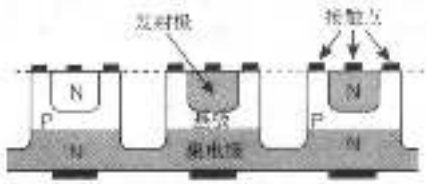
下图说明NPN和PNP型晶体管中的电流方向。晶体管中基极的作用就好像是真空二极管中的控制栅极，晶体管基极中一个小的电流变化，能够引起发射极和集电极之间的一个大的电流。



采用“TO-50”形式封装的分立元件（discrete component）的晶体管如第1个图所示。利用三根分别连接基极、集电极和发射极的引线（lead），可以将晶体管和外电路连接在一起。晶体管不但比真空管体积小，而且工作温度低，性能也可靠得多。真空管中的灯丝，就像电灯泡中的灯丝一样，烧起来发烫，最后很容易被烧断。采用晶体管元件的计算机自然要比前一代的真空管计算机体积要小得多，工作温度也要低得多。但最终的微型化并不是通过使用分立的晶体管来替代分立的真空管实现的，而是将整个电路压缩到一块硅片上。

集成电路（或称为芯片，chip）中集成了几百到几百万个微型晶体管。人们可以采用几种不同的技术来制造集成电路。其中一种创建集成电路的最简单的方法是利用计算机辅助设计软件，将组成芯

片的各个不同硅材料层的每一层都印制成放大的结构图，就像照相用的底片。然后，采用类似于照相制版的技术，利用这些计算机设计出来的不同层的结构图在硅片上生成各种不同的电路。利用紫外光透过结构图底片照射硅片，引起涂覆在硅片表面上的一层光阻材料（称为光敏感材料）的化学性质发生变化。然后，将芯片浸入某种化学液体中，硅片上被曝光的区域将会被冲洗掉，结果腐蚀后就留下各种电路的精细图案。这种技术称为光刻照相制版（photomicrolithography）。腐蚀过程完成后，在芯片衬底材料的表面淀积一层N型或者P型材料。然后，再将这一层材料表面也涂上光阻材料，类似于前面一层的处理，进行曝光和腐蚀处理。重复类似过程，直到所有电路层的腐蚀过程都完成，形成如上图所示的N型和P型材料的不同峰谷的结构。这些微型的电子元器件，包括晶体管，不但工作性能与较大尺寸的分立元器件完全相同，而且工作速度更快，功耗很小。■



1.5.4　第三代：集成电路计算机

随着集成电路的诞生，计算机真正进入了爆炸式的快速发展时期（1965—1980）。Jack Kilby发明了集成电路（IC），我们称之为微芯片（microchip）。这种集成电路是利用锗材料制成的。6个月后，一直从事集成电路设计工作的Robert Noyce采用硅材料，而不是锗材料，制造了类似的集成电路器件。这种硅芯片奠定计算机工业的基础。早期的集成电路，只是在单块硅芯片上集成了几十个晶体管。但是，这些集成电路的尺寸已经比"分立元件"的晶体管还要小。使用集成电路的计算机，速度更快、体积更小且价格也更加便宜。并且，集成电路的计算机在处理能力方面有了巨大提高。IBM公司生产的System/360系列计算机，就属于第一批完全采用固体元器件建造的商用计算机。360系列计算机也是IBM公司所提供的第一代可以相互兼容的系列产品，也就是说这个系列的计算机都使用相同的汇编语言。这样，一些小型计算机的用户可以很方便地升级到较大型的机器，而无需重新编写原来所有的机器软件。这在当时来说是一种具有革命性的新思想。



计算机元件大小的比较
从图的底部开始，按照时针方向分别为：
1）真空管
2）晶体管
3）包含有3200个2-输入与非门（NAND）的芯片
4）集成电路封装（左手角下方的银色方块就是一块集成电路）
感谢Linda Null提供的照片

在IC时代，还引进了分时共享（time-sharing）和多道程序处理（multiprogramming，即具有多个人同时使用一台计算机的能力）的概念。反之，多道程序处理又需要为这些计算机引进新的操作系统。具有分时共享能力的小型计算机系统的出现，例如DEC's，PDP-8和PDP-11，使得一些小公司和许多大学都负担得起利用计算机来进行计算工作。IC技术的发展同样也促进了一些功能更加强大的超级计算机的开发。Seymour Cray曾经于在律语CDC-6600系统时所学到的知识，开办了自己的计算机公司：Cray Research Corporation。这家公司从1976年推出价值880万美元的Cray-1开始，建造了许多超级计算机。Cray-1在性能上已经远远超过了CDC-6600，它可以在一秒钟内执行超过1.6亿条指令，并支持8兆字节的存储器。

1.5.5　第四代：超大规模集成电路计算机

在电子技术变革的第三代，已经出现了多个晶体管集成到一块芯片上的集成电路。随着半导体制

造技术和芯片技术的不断发展，单块芯片上可以集成的晶体管数目越来越多。集成电路技术的发展分为以下几个阶段：小规模集成电路（SSI）：每块芯片上只有 10－100 个元件；中规模集成电路（MSI）：每块芯片上集成 100－1000 个元件；大规模集成电路（LSI）：每块芯片上集成的元件多达 1000－10000 个；最后是超大规模集成电路（VLSI），每块芯片上集成的元件超过 10000。超大规模集成电路标志着第四代计算机的开始。

为了使大家对集成度有一个基本的概念，现在我们来考虑一个计划：就是要把 ENIAC 组装到一个芯片上。1997 年，为了纪念第一台电子计算机诞生 50 周年，一群宾夕法尼亚大学的学生制造了一个单芯片的“ENIAC”。原来那个占地 1800 平方英尺，重 30 吨且耗电 170 千瓦的庞然大物被微缩到只有拇指指甲大小的一块芯片。这块芯片包含约 174 569 只晶体管，这个数目还不到 20 世纪 90 年代后期相同大小的硅片上通常所集成的元件数目的十分之一。

利用 VLSI，Intel 公司在 1971 年创造出了世界上第一个微处理器 4004。这是一个全功能的 4 位系统，工作频率为 108KHz。Intel 公司还引进了随机存储器（RAM）芯片，单芯片上可以存储 4K 位的存储器。这样一来，第四代计算机和以前的固体元件计算机相比，体积更小、速度更快。

VLSI 的技术和它那难以令人置信的缩小电子线路的能力，催生了微型计算机（简称微机）的发展。微型计算机系统的体积非常小，价格也不贵，普通大众都可以买得起，用得起。最早的微型计算机是微型仪器和遥感技术公司（MITS）1975 年推出的 Alstair 8800。随后，很快涌现出了 Apple I、Apple II，Commodore 公司的 PET 和 Vic 20 等多款微机。到了 1981 年，IBM 公司推出了个人计算机（Personal Computer，PC 机）。

个人计算机是 IBM 公司生产的一种“入门级”计算机系统的第三次尝试。公司早期推出的 Datamaster 以及 5100 系列的桌上型计算机在市场上遭受了惨痛失败。尽管遭遇到这些早期的失败经历，但 IBM 公司的 John Opel 还是说服公司的管理层再进行一些这方面的尝试。他建议在远离 IBM 公司总部（Armonk，New York）的 Florida 州的 Boca Raton 成立一个相对自治的“独立事业部”。Opel 挑选了一位精力充沛又非常有能力的工程师 Don Estridge 负责代号为 Acorn 的新系统开发。鉴于 IBM 公司过去在小型系统领域的失败教训，公司的领导层对 Acorn 项目的进度和经费支出进行了严格的控制。最后，在 Opel 承诺一年之内推出产品后，这个项目才得以开始实施。看起来，在这样短的一个期限内似乎是不可能实现的。

Estridge 知道要在短短 12 个月计划中推出 PC 机，唯一的出路就是要打破 IBM 的传统惯例，尽可能多地使用“非定制的”的通用部件。因此，从这种思想出发，IBM 的 PC 机构思设计成为一种“开放式”的体系结构。尽管后来 IBM 公司的某些人后悔当时的这个决定使得 PC 机的体系结构很少有专利产品的成分，但正是因为这种广泛的公开性让 IBM 公司为计算机工业制定了行业标准。就在 IBM 的竞争者忙于起诉抄袭他们系统设计的公司时，PC 机的克隆产品已经如雨后春笋般遍地滋生开来了。不久，“IBM-兼容”微机的价格下降到每个小公司都能承受的水平。同样，要感谢那些克隆产品的制造者，因为他们大量生产的这类计算机系统产品很快就在人们的家中找到了真正的“个人使用”的市场。

IBM 公司最终丢失了它在微型计算机市场的主导地位，但是“魔鬼已经从瓶子中跑了出来”。无论好坏，IBM 的体系结构事实上仍然是微型计算机制造业的行业标准，而且每年都不断有更大更快的计算机系统面世。如今，普通的桌面计算机的运算能力已经超过 20 世纪 60 年代大型计算机系统许多倍了。

自从 20 世纪 60 年代以来，由于 VLSI 技术的突飞猛进，大型计算机系统在性价比方面已经取得了令人瞠目结舌的进步。尽管 IBM System/360 在当时已经完全是一个固体电子器件的系统，但是它仍然是一个需要水冷的和非常耗电的庞然大物。它每秒钟仅能执行 50 000 条指令，且只能支持 16 兆字节的存储器。通常情况下，这些系统中只装有数千字节左右的物理存储器。大型计算机实在是太贵了，以至于只有最大型的公司和大学才能够买得起或租得起一台这样的系统。当今的大型计算机，又称为“企业级服务器”，其价格仍旧高达上百万美元。但是，和过去相比这些大型计算机的处理能力已经提高了几千倍以上。20 世纪 90 年代后期，大型计算机系统的性能指标就已经跃过了每秒钟执行 10 亿条

指令的目标。这些大型计算机系统通常用作网络服务器，在日常工作中每分钟都要完成成千上万件事务的处理工作。

VLSI 技术带给超级计算机的强大处理能力实在是难以全面地加以评述。第一台超级计算机 CDC6600，每秒钟可以执行 1000 万条指令，并配有 128 K 字节主存储器。相比之下，今天的超级计算机的内部可能包含几千个处理器，可以编址上万亿字节（terabyte）的存储器，而且不久就可以达到每秒钟执行千万亿（quadrillion）条指令的运算速度。

到底会是什么样的技术标志着第五代计算机的开始呢？有些人说第五代计算机的标志是并行处理、网络应用和单用户工作站的广泛普及和接受。很多人相信现在实际上已经跨越了第五代。还有一些人把第五代计算机的特征定义为神经网络、DNA、或者是光学计算机系统的时代。也许在我们已经进步到第六代或者是第七代计算机之前，都不可能定义出什么是五代计算机，当然也无法预料后来的那些时代会给我们带来些什么。

1.5.6　摩尔定律

超大规模集成电路的高速发展究竟要到哪里才算是结束？晶体管到底可以做到多小？芯片上堆积的最大元件密度可以有多高？对于这些问题谁也说不清楚。事实上，科学家们每年都在不断突破那些预言者试图设置的所谓集成度的极限。在 1965 年，当 Intel 公司的奠基人摩尔（Gordon Moore）预言“集成电路中的晶体管数目将会每年翻一番”时，不少人都对此表示怀疑。对这个预言，现在的说法通常被解释为“硅芯片的密度每 18 个月翻一番”。摩尔的这一断言已经变成了著名的摩尔定律（Moore's Law）。摩尔当初预测他的这个假定只能够维持 10 年左右的时间。然而，芯片制造技术的进步已经让摩尔定律足足保持了差不多 40 年。而且，还有很多人甚至认为到了 21 世纪 10 年代时，摩尔定律仍旧会保持有效。

然而，采用当前的技术，摩尔定律所指出的集成电路的这种发展趋势不可能永远保持下去。很显然，各种物理和财务上的限制终究会出现。以当前集成电路缩小的速率计算，大约还需要 500 年就可以把整个太阳系集成到一块芯片上。当然，集成电路的技术发展还存在着这样或那样的许多制约因素。费用问题也许就是一个最终的限制。Intel 公司的早期投资人 Arthur Rock 曾提出了所谓的 Rock 定律，它可以作为摩尔定律的一个推论：即“制造半导体集成电路所需要的主要设备的成本每 4 年就要翻一番”。Rock 定律出自于一个像 Rock 这样的金融家的观察研究。他发现新的芯片生产设备的价格标签从 1968 年的大约 12 000 美元逐步升高到 90 年代后期的 1 200 万元美元。按照现在这样的速率发展下去，到 2035 年不但存储器单元的尺寸会小于一个原子，而且制造一个芯片就需要花费掉全世界的整个财富。这样，即使我们可以把芯片做得越来越小，速度也越来越快，但是最终的问题是我们是否能够付得起建造这些芯片的费用。

的确，如果摩尔定律保持正确，那么 Rock 定律必须失效。很显然，如果上面这两种情况都发生的话，计算机制造必须采用全新的技术。有关新的计算范式的研究在上个 10 年的后半期就已经着手开始进行了。在有机计算、超导、分子物理和量子计算等领域已经出现了新型计算机的实验室原型。量子计算机采用量子力学的奇妙思想来解决计算问题，这种构思特别令人兴奋。与以前所使用的任何计算方法相比，量子计算方法不但在运算速度上成指数增加，而且对计算问题的定义方式也有革命性的变革。那些在我们今天看来是有些愚蠢而不切实际的难题，可能会是下一代的学童能够很容易掌握的问题。事实上，也许那时的学童看到我们的“原型”计算机系统会觉得很好笑，就像我们现在去看 ENIAC 机器一样。

1.6　计算机的分层组织结构

如果需要计算机解决各式各样的问题，那么计算机就必须能够执行用各种不同语言编写的程序，从 FORTRAN 和 C 语言到 Lisp 和 Prolog 语言。正如读者在第 3 章将会看到的，打开计算机我们所面对的物理元件只是一些导线和逻辑门。在这些物理元件和高级语言（如 C++语言）之间存在着一个巨

大的空间（从语义学上来讲是存在一个可寻址的空间），作为一个实际应用的计算机系统，对于大多数的用户来说，这种语义学上的问题必须是不可见的。

有编程经验的人都知道，如果要解决一个很大的问题，我们应该把这个大问题分解，采用“分而治之”的方法。在编程时，通常会先把一个问题分成若干个模块，然后单独设计每一个模块。每个模块都执行一个特定的任务。在应用这些模块时，我们需要了解的仅仅是这些模块和其他模块之间的相互连接方式。

计算机系统的组成原理与此类似。通过抽象原理，我们可以设想计算机是按照不同的层次结构来建造的。这里的每一个层次都有某项特定功能，并有一个特定的假想机器与之相对应。对应计算机的每一个层次的这种假想的计算机称之为虚拟机（virtual machine）。每一层的虚拟机都执行自己特有的指令集，必要时还可以调用较低层次的虚拟机来完成各种工作任务。通过学习计算机的组成原理，读者将会了解到这种层次划分的合理性，各个层次的实现方式以及各个层次之间的相互连接方式（或称为接口，interface）。图 1-3 给出了现在人家普遍接受的代表不同抽象的虚拟机器的计算机结构层次。

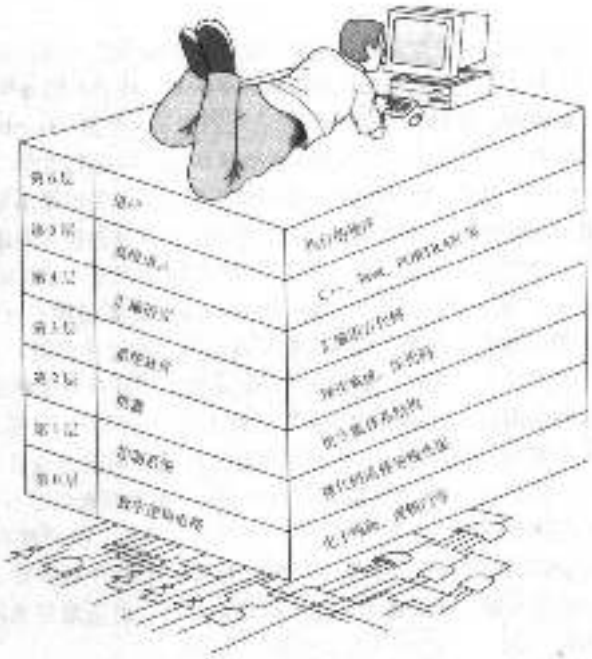


图 1-3 现代计算机系统组成的不同抽象层次

第 6 层是用户层（User Level），它由各种应用任务组成，这是大家最熟悉的层次。在这一个层次，主要是运行各种程序，比如说文字处理程序、图形程序包、或者是各种游戏等。从用户层来看，较低的层次基本上是不可见的。

第 5 层是高级语言层（High-Level Language Level），它由各种高级编程语言组成，例如 C、C++、FORTRAN、Lisp、Pascal 和 Prolog 等高级语言。我们必须利用某种编译程序或者解释程序将这些高级语言翻译成机器可以理解的语言。然后，将这些编译后的高级语言首先翻译成汇编语言，再将汇编语言转变成机器代码。在这一层次上，用户也基本上看不到较低级的层次。显然，编程人员必须知道数据类型和可用的各种指令，但是不必了解这些指令的具体执行方式。

第 4 层为汇编语言层（Assembly Language Level），它包含某种类型的汇编语言。如上所述，编译过的高级语言首先翻译成汇编语言，然后汇编语言直接转换成机器代码。这是“一对一”的翻译过程，也就是说一条汇编语言的指令会被严格翻译成一条机器语言的指令。由于具有这些独立的层次，所以我们就可以在应用过程中缩小高级语言（例如 C++ 语言）与实际机器语言（由一些 0 和 1 序列组成语

言）之间的语义上的间隙。

第3层为系统软件层（System Software Level），主要处理操作系统指令。这一层次负责多用户编程，存储器保护，过程同步和其他一些重要功能。通常，从汇编语言翻译过来的机器语言指令可以直接越过这一层而不会被修改。

第2层是指令集体系结构（Instruction Set Architecture，ISA），或称为机器层（Machine Level），它由特殊的计算机系统结构所能识别的机器语言组成。使用一种计算机的真正的机器语言编写的程序可以在一个由硬连线组成的计算机中直接用电路来执行，而不再需要任何解释程序、翻译程序或者是编译程序。读者将会在第4章和第5章中深入学习指令集体系结构。

第1层为控制层（Control Level）。在这个层次上，控制单元（control unit）将确保正确地译码并执行指令，并且适时地将数据传送到正确的位置。控制单元会逐条解释从上层传送过来的机器指令，一次解释一条指令，指挥机器执行所要求的正确动作。

控制单元的设计有两种方式：一种是导线直接连接（称为*硬连线*，hardwired），另一种是微程序控制（或称为*微编程*，microprogrammed）。在使用硬连线的控制单元中，控制信号由数字逻辑部件的电路模块发送出来。这些控制信号指引着各种数据和指令流向系统的合适部位。通常，硬连线的控制单元的速度非常快，因为它们都是一些真实的物理部件。但是，一旦这些硬连线的控制单元的实现完成后，就很难再进行修改。

设计控制单元的另一种选择是使用一个微程序来执行指令。微程序是一个利用某种低级语言编写的程序，这种低级语言可以由硬件直接执行。在计算机的第二个层次上生成的机器指令将被输入到这个微程序中，再由微程序来解释各种指令，然后激活硬件执行原始指令。一条机器层次的指令通常被翻译成几个微代码指令。这种翻译过程不再具有汇编语言和机器语言之间存在的一一对应关系。微程序方式的控制单元非常流行，因为对它们进行修改要相对容易一些。当然，微编程的缺点也是显而易见的。由于增加了额外的翻译过程，通常这种方法执行指令的速度会比较慢。

第0层是数字逻辑层（Digital Logic Level），在这里我们所面对的是计算机系统的物理构成：各种逻辑门和引线。这些都是基本的构造模块和数学逻辑的实现，它们是各种计算机系统的共同部件。第3章将详细介绍计算机的数字逻辑层。

1.7　冯·诺伊曼模型

在最早期的电子计算机器中，编程就是利用各种导线进行接插连线。由于没有计算机的分层结构，对早期的计算机进行编程可以说是电气工程上的一大壮举，哪怕程序仅仅是算法设计中的一个小练习题。在完成ENIAC的工作前，John W. Mauchly和J. Presper Eckert构思设计了一种可以改变计算机器的运行状态的简易方法。他们依赖于一种利用汞延迟线形式的存储设备，可以提供一种存储程序指令的方法，从而在每次求解新问题或者对原来的程序进行调试时，不必再对系统进行重新连线。Mauchly和Eckert将他们的想法写成文字计划，希望以此作为他们下一代的计算机EDVAC的基础。不巧的是，当时正值第二次世界大战，由于他们置身于最高机密的ENIAC计划的研究中，所以Mauchly和Eckert不能够立即发表这个具有远见的构想。

但是，对于在ENIAC研究计划外围工作的许多人来说，并不存在这样的禁止令。著名的匈牙利数学家冯·诺伊曼（John Von Neumman）正是其中之一。在阅读了Mauchly和Eckert有关EDVAC的计划书后，冯·诺伊曼发表和公开了这种存储器的思想。他成功地把这种概念公布于世，结果历来人们都公认这是他的一大发明。现在，所有的存储程序的计算机都称为使用冯·诺伊曼体系结构（Von Neumman architecture）的冯·诺伊曼系统。尽管本书并不排斥对于存储程序的计算机使用冯·诺伊曼体系结构的这种传统说法，但是我们在这里也不能不对这种思想的真正发明人John W、Mauchly和J. Presper Eckert给予应有的称颂和尊重。

现代版本的存储程序的计算机体系结构至少应该满足以下的基本特征：

- 由三大硬件系统组成：一个中央处理器（central processing unit，CPU），其中包含一个控制单元、一个算术逻辑单元（arithmetic logic unit，ALU）、若干个寄存器（register，一些小的存储单元）和一个程序计数器；一个主存储器系统（main-memory system），用来保存控制计算机操作的各种程序；以及一个输入/输出（I/O）系统。
- 具有执行程序指令的处理能力。
- 在主存储器系统和CPU的控制单元之间，包含一条物理上的或者是逻辑上的单一通道，可以强制改变指令和执行的周期。通常，这种单一通道称作冯·诺伊曼瓶颈（Von Neumman bottleneck）。

图1.4给出了现代计算机系统中是怎样将这些特性组合在一起的。注意：在图中，系统的所有I/O都是通过算术逻辑单元连接的（实际上，它们是通过累加器连接的，累加器是ALU的一部分）。这种体系结构按着一种称为冯·诺伊曼执行周期（Von Neumman execution cycle）的方式来运行程序。这种计算机的工作原理又称为取指-译码-执行周期（fetch-decode-execute cycle）。这种周期的一个循环过程如下：

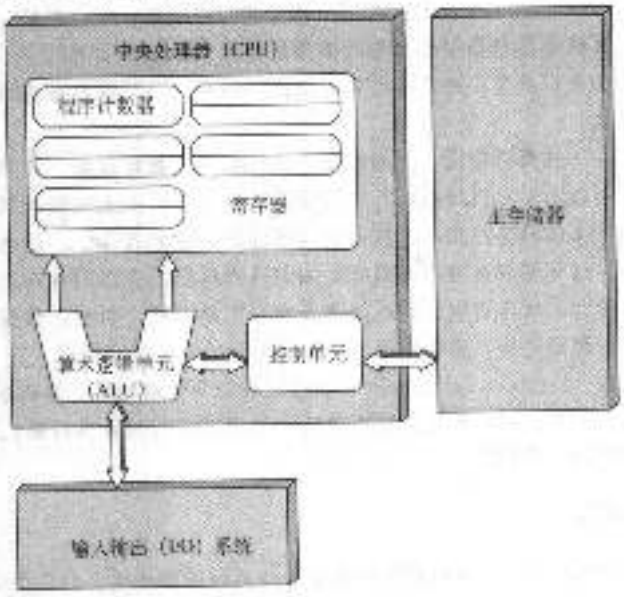


图1-4 冯·诺伊曼体系结构

- 控制单元从计算机的存储器中提取下一条程序指令，并利用程序计数器来决定这条指令的所在位置。
- 对提取的指令进行译码，变成ALU能够理解的一种语言。
- 从存储器中取出执行指令所需的各种操作数的数据，并把它们放入CPU的寄存器中。
- ALU执行指令，并将执行的结果存放到寄存器或者存储器中。

冯·诺伊曼体系结构的这种思想已经得到了很大的发展。现在，我们可以在程序执行之前，将程序和数据存放在某种慢速访问存储介质中（例如硬盘）。而在程序执行时，再将这些程序和数据复制到一个快速访问的易失性的存储介质中（例如RAM）。原来早期的计算机体系结构也已经逐渐发展成为现在称为系统总线模型（system bus model）的计算机体系结构，如图1.5所示。计算机系统中的数据总线将数据从主存储器中传递到CPU的寄存器中，反之亦然。同时，地址总线负责保持数据总线正在访问的数据地址。而控制总线则传送各种必要的控制信号，以指定信息传输发生的方式。

对冯·诺伊曼体系结构其他方面的一些改进包括：采用变址寄存器进行编址，增加了浮点数据，使用中断和异步的I/O结构，增加了虚拟存储器，以及增加了通用寄存器。在本书后面的章节中，读者可以进一步了解有关这些结构改进方面的大量知识。

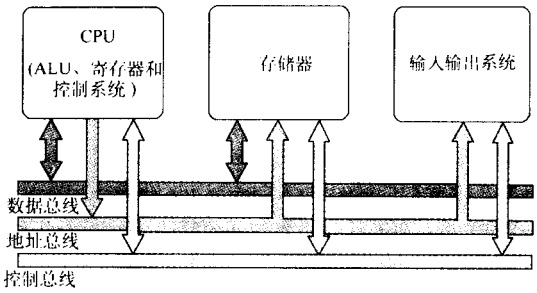


图 1-5　改进后的冯·诺伊曼体系结构，增加了系统总线

1.8　非冯·诺伊曼模型

长期以来，几乎所有的通用计算机都是按照冯·诺伊曼体系结构设计的。然而，冯·诺伊曼体系结构中的瓶颈问题使工程设计人员一直未能找到构建廉价且与大部分商用软件兼容的快速系统的途径。显然，未受到要与冯·诺伊曼体系结构保持兼容性要求所约束的工程师们，就可以自由地选择使用许多不同的计算模式。许多不同的研究领域都属于非冯·诺伊曼模型的范畴，包括神经网络（利用人脑模型的思想作为计算范式）、基因算法（利用生物学和 DNA 演化的思想开发的算法）、量子计算（前面业已讨论）和并行计算机。在这些新的体系结构中，并行计算的概念是目前最流行的。

今天，并行处理解决了我们面临的一些最大的问题，就如同过去的西部开拓者使用并排的牛来解决当时所面临的一些大问题一样。如果当时的开拓者只使用一头牛，就不足以搬动一棵大树。由于当时不可能饲养出更加强壮的牛，所以开拓者就采用两头牛一起并排来使用。同样的道理，如果一台计算机不够快，功能不够强大，我们为什么不简单地同时使用多台计算机，而要去建造更快和功能更强大的计算机呢？这也就是并行计算机所要做的事情。在 20 世纪 60 年代后期，诞生了第一台并行处理的计算机系统，当时的并行系统只有两个处理器。70 年代的超级计算机已经有多达 32 个处理器。到了 80 年代，有些系统已经拥有超过 1 000 个处理器。1999 年，IBM 公司宣布建成了一台名叫蓝色基因（Blue Gene）的超级电脑。这台巨大的并行计算机包含超过 100 万个处理器，每一个处理器都有自己专用的存储器。当时，蓝色基因第一个任务就是分析蛋白质分子的行为。

但是，即使是并行计算也有它的局限性。随着处理器数目的增加，如何将任务分配给各个处理器的上层管理工作也变得越来越复杂。有些并行处理系统需要配备额外的处理器来管理其他处理器以及在处理器之间进行资源分配。无论在一个计算机系统中放入多少个处理器，也不管分配给这些处理器多少资源，都会遇到各种的瓶颈问题。当然，应对瓶颈问题的最好方法就是尽量少使用系统中最慢的部分。这是从 Amdahl 定律（Amdahl's Law）中所衍生出来的一种思想。Amdahl 定律基本表述为：对于某种特定的系统改进，系统性能增强的可能性受到那些被改进的特征部位的使用次数的限制。这个定律潜在的含义是，每种算法都会有顺序执行的部分，算法的顺序执行部分最终将会限制利用多处理器执行时所能够获得的加速。

本章小结

本章简要介绍了计算机的组成原理和体系结构，以及它们之间的区别。同时，针对一个假想的计算机广告引入了一些专业名词和术语。这些专业名词和术语将会在后面的学习章节中详细介绍。

计算机在历史上曾经只是一种简单的计算机器。随着计算机的发展越来越先进，现在的计算机系统已经变成一种通用机器。人们可以按照不同的需求层次来看待计算机系统，而不再把它当作一个庞然大物。计算机分层结构中的每一层都有特定的作用，并且各个层次都有助于缩小高级编程语言（或称为应用任务）与组成计算机物理硬件的逻辑门电路和导线之间存在的语义上的差距。对于程序设计人员来说，也许在计算领域最重要的发展是引进了冯·诺伊曼体系结构的存储程序的概念。尽管还有其他的计算机的体系结构

模型，但是冯·诺伊曼体系结构在当今的通用计算机系统中占主导地位。

深入阅读

建议读者以本章的简单阐述为基础，阅读一些有关计算机历史方面的书籍。这方面的内容有些是关于历史人物的，有些是关于机器的，很容易引起读者的兴趣。其中一本是 John Atanasoff（1988）在 Mollenhoff 所著的名为《被遗忘的计算机之父》(forgotten father of the computer) 的书籍。该书描述了 Atanasoff 和 John Mauchly 之间非常奇特的关系，以及计算机界的两位巨人 Honeywell 和 Sperry Rand 之间的法律官司。审判的结果最终给了 Atanasoff 应有的历史地位。

如果想寻找一本较薄的有关计算机历史的书籍，不妨试读一下 Rochester 和 Gantz（1983）的著作。而在 Augarten（1985）的著作中阐述的计算机历史可以让人轻松阅读，书中包含数百幅难得一见的早期计算机和计算设备的照片。如果要了解计算机历史发展的完整过程，可以参阅 Cortada（1987）的 3 卷版的词典。Ceruzzi（1998）有关计算历史的论述非常有思想深度。如果对有关历史计算机的一些精彩案例的研究感兴趣的话，可以阅读 Blaauw 和 Brooks（1997）的著作。

通过阅读下面一些的书籍，读者同样也可以得到丰厚的回报，其中包括：McCartney（1999）关于 ENIAC 的介绍，Chopsky 和 Leonsis（1988）有关 IBM PC 的历史发展过程，以及 Toole（1998）有关 Lovelace 伯爵夫人 Ada 的传记等。Polachek（1997）的文章向读者展现了一幅有关计算弹道射击表的复杂性的生动画面。读过这篇文章后，你就会发现军方为什么愿意对那些能够使计算过程变得更快和更精确的承诺付出任何代价。Maxfield 和 Brown（1997）的书则从一个神奇的角度来看待计算的起源和历史，并且深刻透彻地解释了计算机的工作原理。

要了解有关摩尔定律的更多资料，可以参阅 Schaller（1997）的文献。有关早期计算机的详细描述和计算机工业的先驱者的回忆和生平简介，可以参考季刊 *IEEE Annals of the History of Computing*。计算机博物馆历史中心可以直接从网址 www. computerhistory. org 找到。在这里有各种各样的展览、研究报告、历史回顾和各种收藏品。现在，许多城市都设立了计算机博物馆，参观者可以接触到一些老式的计算机。

读者可以在本章中所列举（或者没有列举）的一些制定标准化规则的组织的网站找到有关计算机的大量信息。下面是有关网站的网址：IEEE 的网站 www. ieee. org，ANSI 的网站 www. ansi. org，ISO 的网站 www. iso. ch，BSI 的网站 www. bsi-global. com，以及 ITU-T 的网站 www. itu. int。ISO 的网站中提供了大量有关计算机的信息和计算机标准的参考文件。

WWW 计算机体系结构的网站主页 www. cs. wisc. edu/～arch/www/ 包含与计算机体系结构相关的信息的完整索引。许多 USENET 的新闻小组都对计算机有关的题目进行讨论并发表意见，其中包括 comp. arch 和 comp. arch. storage。

在麻省理工学院（MIT）的期刊杂志 *Technology Review* 的 2000 年 5/6 月号中，整期刊载了有关计算机体系结构的内容，这些文章可能是未来计算机的基础。这一期的内容很值得一读，当然这本杂志每一期的内容都值得一读。

参考文献

Augarten, Stan. *Bit by Bit: An Illustrated History of Computers*. London: Unwin Paperbacks, 1985.

Blaauw, G., & Brooks, F. *Computer Architecture: Concepts and Evolution*. Reading, MA: Addison-Wesley, 1997.

Ceruzzi, Paul E. *A History of Modern Computing*. Cambridge, MA: MIT Press, 1998.

Chopsky, James, & Leonsis, Ted. *Blue Magic: The People, Power and Politics Behind the IBM Personal Computer*. New York: Facts on File Publications, 1988.

Cortada, J. W. *Historical Dictionary of Data Processing*, Volume 1: *Biographies*; Volume 2: *Orga-*

nization, Volume 3: *Technology*. Westport, CT: Greenwood Press, 1987.

Maguire, Yael, Boyden III, Edward S., and Gershenfeld, Neil. "Toward a Table-Top Quantum Computer." *IBM Systems Journal 39*: 3/4 (June 2000), pp. 823–839.

Maxfield, Clive, & Brown, A. *Bebop BYTES Back (An Unconventional Guide to Computers)*. Madison, AL: Doone Publications, 1997.

McCartney, Scott. *ENIAC: The Triumphs and Tragedies of the World's First Computer*. New York: Walker and Company, 1999.

Mollenhoff, Clark R. *Atanasoff: The Forgotten Father of the Computer*. Ames, IA: Iowa State University Press, 1988.

Polachek, Harry. "Before the ENIAC." *IEEE Annals of the History of Computing 19*: 2 (June 1997), pp. 25–30.

Rochester, J. B., & Gantz, J. *The Naked Computer: A Layperson's Almanac of Computer Lore, Wizardry, Personalities, Memorabilia, World Records, Mindblowers, and Tomfoolery*. New York: William A. Morrow, 1983.

Schaller, R. "Moore's Law: Past, Present, and Future." *IEEE Spectrum*, June 1997, pp. 52–59.

Tanenbaum, A. *Structured Computer Organization*, 4th ed. Upper Saddle River, NJ: Prentice Hall, 1999.

Toole, Betty A. *Ada, the Enchantress of Numbers: Prophet of the Computer Age*. Mill Valley, CA: Strawberry Press, 1998.

Waldrop, M. Mitchell. "Quantum Computing." *MIT Technology Review 103*: 3 (May/June 2000), pp. 60–66.

基本概念和术语复习

1. 计算机组成原理和计算机体系结构之间有什么区别?
2. 什么是 ISA?
3. 计算机的硬件和软件等效原理有什么重要性?
4. 列举计算机的 3 种基本部件。
5. 前缀 giga-表示 10 的多少次幂? 它近似等于 2 的多少次幂?
6. 前缀 micro-表示 10 的多少次幂? 它近似等于 2 的多少次幂?
7. 通常用来测量计算机时钟频率的单位是什么?
8. 列举两种类型的计算机存储器。
9. IEEE 组织的基本任务是什么?
10. 最初使用 ISO 缩写的组织的全名是什么? ISO 是取英文首写字母的缩写吗?
11. ANSI 是哪个组织的英文名称的首写字母的缩写?
12. 专门负责有关电话、电信和数据通信事务的瑞士组织的名称是什么?
13. 谁被称为计算机之父? 为什么?
14. 穿孔卡片在计算机发展史上有什么重要意义?
15. 列举两个推动计算机发展的重要因素。
16. 是什么原因使得晶体管比真空管有了如此大的改进?
17. 集成电路与晶体管有什么不同?
18. 解释 SSI、MSI、LSI 和 VLSI 之间的区别。
19. 是什么技术催生了微型计算机的发展?
20. "开放的体系结构" 意味着什么?
21. 阐述摩尔定律, 摩尔定律能够永远保持下去吗?
22. Rock 定律与摩尔定律是如何相互关联的?
23. 阐述和解释大家所公认的计算机分层结构中的 7 个不同层次。这种分层结构对于理解计算机系统有什么帮助?

24. 冯·诺伊曼体系结构与它以前的计算机体系结构有什么不同?
25. 阐述冯·诺伊曼体系结构的特点。
26. 取指-译码-执行周期的工作原理是怎样的?
27. 并行计算是什么含义?
28. Amdahl 定律内在的前提含义是什么?

练习题

◆1. 计算机的硬件和软件在哪些方面不同? 而在哪些方面又是相同的?
2. a) 1 秒等于多少毫秒 (ms)?
   b) 1 秒等于多少微秒 (μs)?
   c) 1 毫秒等于多少纳秒 (ns)?
   d) 1 毫秒等于多少微秒?
   e) 1 微秒等于多少纳秒?
   f) 1 千兆字节 (GB) 等于多少千字节 (KB)?
   g) 1 兆字节 (MB) 等于多少千字节 (KB)?
   h) 1 千兆字节 (GB) 等于多少兆字节 (MB)?
   i) 20 兆字节 (MB) 等于多少字节 (B)?
   j) 2 千兆字节 (GB) 等于多少千字节 (KB)?
◆3. 在纳秒范围内运行的物体比在毫秒范围内运行的物体快多少个数量级?
4. 假设准备购买一台个人使用的计算机。首先,要从一些不同的杂志和报纸上阅读有关的广告,并把一些不太懂的项目和术语记录下来。然后,查阅计算机字典,写出这些项目和术语的简短解释。分析一下哪些重要因素会影响到选择计算机系统的决定,并将这些要素列出来。在选择好你所希望购买的系统后,请确认一下系统规格说明中有哪些项目是有关计算机硬件的,而哪些项目是有关计算机软件的?
5. 挑选一种你最喜欢的计算机语言编写一个小程序。在对程序进行编译后,看一下是否可以确定源代码的指令与由编译器产生的机器语言指令的比例是多少? 如果增加一行源代码,会对机器语言的程序有何影响? 现在添加一些不同的源代码指令,例如先添加一条加法指令,然后再添加一条乘法指令。请问机器代码文件的大小会随着不同的指令发生什么变化? 并给出对结果的意见和看法。
6. 下面对本章第 1.5 节的评论做出回应:如果现在才发明计算机的话,你认为应该给计算机取一个什么样的名字? 答案中至少给出一条好的理由。
◆7. 假设现在的集成电路芯片上一个晶体管的尺寸是 2 微米,请问根据摩尔定律,两年后晶体管的尺寸将会有多大? 摩尔定律与程序设计人员之间怎样相互关联?
8. 当时是什么样的环境背景帮助 IBM 公司取得如此成功?
9. 列举个人计算机的五种基本应用。计算机的应用存在哪些限制? 你能否想像出计算机在不久的将来会和现在完全不同,而且是激动人心的应用吗? 如果有,可能是什么样的应用?
10. 在冯·诺伊曼体系结构的计算机系统中,一个程序和相关的数据都存储在存储器中。如果某个程序原以为在存储器的某个位置保存有一段数据,但实际上这个位置却存储一段程序指令。这样一来,程序就可能会在无意中(或者是有意识地)修改程序本身。对于一个程序设计员来说,这种情况的出现意味着什么含义?
11. 阅读一份当地的报纸,或者浏览因特网上的一些最流行的职业介绍网站,搜索有关计算机方面的招聘信息。说明哪些工作明确地要求计算机硬件的知识? 而哪些工作又暗示了对计算机硬件知识的需求? 要求的硬件知识与招聘公司本身或者是公司的地理位置之间有何关联?

没有算术的生活将会是什么？这是一个多么恐怖的情景。

——Sydney Smith（1835）

第2章　计算机系统中的数据表示方法

2.1　概述

任何计算机的组成在很大程度上取决于数字、字符和控制信息的表示方法。反之亦然：许多年来建立起来的一些标准和惯例已经决定了计算机组成的某些特定的方面。本章主要描述计算机对数字和字符进行存储和操作的的各种方法。下面部分所阐述的思想将形成理解各种不同类型数字计算机系统的组成和功能的基础。

数字计算机中最基本的信息单元被称为一位（bit）。它是二进制数（binary digit）的英文缩写。具体说来，一个位不过是代表了计算机的电子电路中的“开”或者“关”的状态（或者是电位的“高”或“低”）。1964年，IBM system/360大型机的设计者建立了一套以8个位为一组作为计算机存储器编址的基本单元的约定。他们称这种8个数位的组合为一个字节（byte）。

计算机的字（word）由两个或者多个相邻的字节构成。计算机的字有时用来对存储器进行编址。多数情况下总是把字作为一个集合来处理。字的大小（word size）表示了一个特定的计算机体系结构能够处理的最有效的数据的大小。计算机的字可以是16位、32位、64位或者是计算机组成内容中有意义的其他位数（包括不是8的倍数的位数）。8位字节可以被对半分成两个4位，称为半字节（nibble，或nybble）。因为一个字节的每一位在位置编码计数系统中都具有一个明确的值。所以，包含最小值二进制数字的半字节称为低半字节，而另外的半个字节称为高半字节。

2.2　位置编码系统

直到在16世纪的某个时期，欧洲才采用了十进制数字（基数为10）的计数体系（或称为数制）。而此前阿拉伯和印度已经使用十进制数近一千年了。今天，我们理所当然地认为，数字243代表的是2个100加上4个10，再加上3个整数。虽然，0表示什么都没有。但事实上，大家都知道1和10这两个数所代表的意义有着实质性的差别。

位置编码系统（positional numbering systems）所包含的基本思想是，任意数字的值都可以通过表示成某个基数（或称为底，radix）的乘幂形式。这种计数体系通常也称为权重编码系统（weighted numbering system），因为数的每一个位置都是基数的幂次方。

位置编码系统中所使用的有效数字的数目等于系统的基数的大小。例如，在十进制体系中有10个数字0到9，而在三进制体系（基数为3）中的数字为0、1和2。任何一种计数体系（数制）中的最大的合法数字均为基数减1。所以，在任何基（数）小于9的计数体系中，8都是非法的数字。为了区别，不同的基采用下标的形式来表示，例如33_{10}，表示十进制数33（本书中没有下标的数都假定为十进制数）。任意十进制数都可以严格地采用其他任意的整数基数制来表示（参见例2-1）。

例2-1　将3个十进制数表为下列不同基数的幂指数形式。

$$243.51_{10}=2\times10^2+4\times10^1+3\times10^0+5\times10^{-1}+1\times10^{-2}$$

$$212_3=2\times3^2+1\times3^1+2\times3^0=23_{10}$$

$$10110_2=1\times2^4+0\times2^3+1\times2^2+1\times2^1+0\times2^0=22_{10}$$ ■

在计算机科学中，两个最重要的数制是二进制数（基数为2）和十六进制数（基数为16）。另外一个比较重要的数制是八进制数（基数为8）。二进制数只采用0和1两个数字，八进制计数体系采用0

到7八个数字，而十六进制计数体系除了3到9十个数字外，还使用A、B、C、D、E和F来表示数字10到15。图2-1给出了某些常用数制的表示方法。

2的幂次数
$2^{-2}=\frac{1}{4}=0.25$
$2^{-1}=\frac{1}{2}=0.5$
$2^0=1$
$2^1=2$
$2^2=4$
$2^3=8$
$2^4=16$
$2^5=32$
$2^6=64$
$2^7=128$
$2^8=256$
$2^9=512$
$2^{10}=1\,024$
$2^{15}=32\,768$
$2^{16}=65\,536$

十进制数	4位二进制数	十六进制数
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

图2-1 计算机中常用的数制

2.3 十进制数和二进制数之间的转换

Gottfried Leibniz (1646—1716) 率先将十进制（位置编码）计数体系的思想推广应用到其他的数制体系。作为一位思想深邃的数学家，Leibniz对二进制计数制的发展有着卓越的贡献。他将任意一个整数都可以通过一系列由0和1来表示这个事实与上帝从虚无（0）中创造宇宙（1）的思想联系起来。直到20世纪40年代后期，第一台二进制数字计算机建成之前，这种二进制计数体系对人们来说也只不过是一个数学上的新奇事物而已。而在今天，二进制计数体系实质上已经成为所有依赖于数字控制的电子设备的核心基础。

因为二进制数制的计数方法十分简单，所以可以很方便地将二进制数翻译转换成对应的电子电路，并且十分易于理解。有经验的计算机专业人士一眼就可以识别较小数值的二进制数（例如图2-1中所列出来的二进制数）。但是，对于较大数值的二进制数和分数的转换，通常需要用计算器或者在纸上用笔来完成。还好，这些转换技术只要稍加练习就很容易掌握。下面将介绍几个简单的转换方法和技巧。

2.3.1 无符号整数的转换

先从无符号数字开始，进行不同基数之间的数字转换。带符号数字（数字可以是正数或负数）的转换要复杂一些。重要的是，在进行带符号数字的转换前，首先需要掌握一些数字间相互转换的基本技巧。

不同基数的计数体系（简称数制）之间的转换，可以采用重复减法或者是除法余数方法来完成。重复减法的方法比较麻烦，并且要求对所使用的基数的幂指数形式比较熟悉。由于更加直观的缘故，下面首先介绍重复减法的转换方法。

例如，将十进制数104_{10}转换成三进制数字。大家知道，$3^4=81$是小于104的3的最大乘幂数值，因此，所转换的数在以3为基数的数制中会占有5个数字宽度（每位数字都代表了基数3的不同乘幂指数，0到4）。首先在81上记下1，并从104中减去81，得到差为23。很显然，接下来3的幂指数为3，$3^3=27$大于23，不够减，在该位上记下0。接着，看一看23除以$3^2=9$有几倍。不难看出，应该在该位置上记下2，并从23中减去18余下5。再从5中减$3^1=3$，该位记1，最后余2，即为2×3^0。上面的转换步骤如例2-2所示。

例 2-2 利用减法将十进制数 104_{10} 转换为基为 3 的数字。

$$
\begin{array}{l}
104 \\
-81 = 3^4 \times 1 \\
\hline
23 \\
-0 = 3^3 \times 0 \\
\hline
23 \\
-18 = 3^2 \times 2 \\
\hline
5 \\
-3 = 3^1 \times 1 \\
\hline
2 \\
-2 = 3^0 \times 2 \\
\hline
0
\end{array}
$$

$104_{10} = 10212_3$ ■

除法-余数方法比重复减法要简单和快速一些。采用连续被基数相除的思想事实上与连续减去基数的指数幂是相同的。连续被基数相除所得到的余数就是转换结果的数字，阅读方法是从下往上进行。这种方法的具体操作如例 2-3 所示。

例 2-3 利用除法余数将十进制数 104_{10} 转换为基数为 3 的数字。

```
3|104  2   104 除以 3 等于 34 余 2
 3|34  1    34 除以 3 等于 11 余 1
 3|11  2    11 除以 3 等于 3 余 2
  3|3  0     3 除以 3 等于 1 余 0
  3|1  1     1 除以 3 等于 0 余 1
   0
```

从下往上读余数，得：$104_{10} = 10212_3$。 ■

这种方法适用于任何基数之间的数字转换。由于这种方法计算简便，所以在从十进制数到二进制数的转换时特别有用。例 2-4 给出了一个这样的转换例子。

例 2-4 将十进制数 147_{10} 转换为二进制数字。

```
2|147  1   147 除以 2 等于 73 余 1
 2|73  1    73 除以 2 等于 36 余 1
 2|36  0    36 除以 2 等于 18 余 0
 2|18  0    18 除以 2 等于 9 余 0
  2|9  1     9 除以 2 等于 4 余 1
  2|4  0     4 除以 2 等于 2 余 0
  2|2  0     2 除以 2 等于 1 余 0
  2|1  1     1 除以 2 等于 0 余 1
   0
```

从下往上读余数，得：$147_{10} = 10010011_2$。 ■

一个 N 位的二进制数可以表示从 0 到 $2^N - 1$ 范围内的无符号十进制整数。例如，4 位二进制数可以表示的十进制数值是 0 到 15，而 8 位二进制数则可以表示 0 到 255 之间的数值。在对二进制数进行算术运算时，一个给定位数的二进制数所能表示的数值范围是非常重要的。考虑下面的一种情形，如果二进制数的长度是 4 位，而要将二进制数 1111_2（15_{10}）与 1111_2 相加。很明显，15 加 15 等于 30，而 30 不能用 4 位二进制数来表示。这种现象称为溢出（overflow），通常发生在对无符号的二进制数进行算术运算时，结果超出给定位数的二进制数所能表示的数值范围的情形。在本章第 2.4 节中讨论带符号数时，将会对溢出现象进行详细介绍。

2.3.2 分数转换

在任意数制中，分数都可以采用其他数制中基数的负指数幂来近似表示。利用小数点（radix

point）将一个数的整数部分和分数部分分开。十进制数中的小数点称为十进制小数点，而二进制数中的小数点称为二进制小数点。二进制分数采用二进制小数点来表示。

如果在某个数制表示中，小数点右边包含循环重复的数字串的分数，而在另一种数制的表示中就不一定也具有重复的数字序列。例如，2/3 是一个循环的十进制分数，而在三进制数中则表示为 0.2_3（$2\times3^{-1}=2\times1/3$），不再有数字循环。

同样，可以利用类似于整数转换所使用的重复减法和除法余数方法，来进行不同基数体系之间的分数转换。下面的例 2-5 就是使用重复减法将一个十进制分数转换成一个五进制的分数。

例 2-5 将 0.4304_{10}转化为基数为 5 的分数。

$$\begin{array}{r} 0.4304 \\ -0.4000 \\ \hline 0.0304 \end{array} = 5^{-1}\times 2$$

$$\begin{array}{r} -0.0000 \\ \hline 0.0304 \end{array} = 5^{-2}\times 0 \qquad \text{（一个占位符号。）}$$

$$\begin{array}{r} -0.0240 \\ \hline 0.0064 \end{array} = 5^{-3}\times 3$$

$$\begin{array}{r} -0.0064 \\ \hline 0.0000 \end{array} = 5^{-4}\times 4$$

从上往下读右边的倍数，得：$0.4304_{10}=0.2034_5$。 ■

在使用余数方法进行整数转换时，是利用基数的正指数幂原理来进行的。因此，可以合理推断在进行分数转换时应该使用乘法，因为分数是按照基的负指数来表示的。但是，转换结果不是像前面所做的一样去取余数，而是在每次乘上基数后，只取其积的整数部分。最后的结果是从上向下，而不是从下向上读取。例 2-6 表示的就是一个分数的转换过程。

例 2-6 将 0.4304_{10}转化为基数为 5 的分数。

$$\begin{array}{r} 0.4304 \\ \times\quad 5 \\ \hline 2.1520 \end{array}$$ 整数部分为 2，接下来乘法中将会被省略。

$$\begin{array}{r} 0.1520 \\ \times\quad 5 \\ \hline 0.7600 \end{array}$$ 整数部分为 0，作为一个占位符号。

$$\begin{array}{r} 0.7600 \\ \times\quad 5 \\ \hline 3.8000 \end{array}$$ 整数部分为 3，接下来乘法中将会被省略。

$$\begin{array}{r} 0.8000 \\ \times\quad 5 \\ \hline 4.0000 \end{array}$$ 分数部分现在为 0，转换完成。

从上向下读取整数部分，结果为：$0.4304_{10}=0.2034_5$。 ■

本例中所设计的转换过程只进行了几步就完成了。通常的情况并不会如此简单，经常会遇到循环分数。大部分的计算机系统都执行某些特定的取舍近似算法，以保证有足够的转换精度。但是本书中为了清楚起见，在达到所需要的精度时，只是简单地舍去（或者删去）多余的结果。参见例 2-7。

例 2-7 转换 0.34375_{10}为二进制数，小数点的的右面取 4 位。

$$\begin{array}{r} 0.34375 \\ \times\quad 2 \\ \hline 0.68750 \end{array}$$ （另一个占位符号。）

$$\begin{array}{r} 0.68750 \\ \times\quad 2 \\ \hline 1.37500 \end{array}$$

$$\begin{array}{r} 0.37500 \\ \times\quad 2 \\ \hline 0.75000 \end{array}$$

0.75000
×　　2
1.50000　（已经到了第4位，就此停止后续地运算。）

从上向下读取，直到小数点右面的第4位，即 $0.34375_{10}=0.0101_2$。■

前面所描述的方法可以用来进行各种基数之间的任意数字的直接转换，例如以4为基的体系和以3为基的体系之间的数的转换（如例2-8所示）。但是，大多数情形下都采用先把数字转换成以10为基的数，再将其转换为所要求的基数。这种方法要比各种基数之间的直接转换快捷和精确得多。这种规则存在一种例外情况，就是以2的指数幂组成的基数之间的数字转换，在下面的章节部分我们将会看到这种转换。

例2-8 将四进制数 3121_4 转换为三进制数。

首先转换成十进制数：

$$3121_4=3\times4^3+1\times4^2+2\times4^1+1\times4^0$$
$$=3\times64+1\times16+2\times4+1=217_{10}$$

然而转换成三进制数：

3|217　1
3|72　0
3|24　0
3|8　2
3|2　2
0　结果为 $3121_4=22001_3$。■

2.3.3 以2的指数幂为基数的数制之间的转换

通常可以将二进制数表示为十六进制或者是八进制数的形式，这样可以增加二进制数的可读性。因为 $16=2^4$，以4位为一组（称为一个十六进制字节），很容易被识别为一个十六进制的数字。类似地，由于 $8=2^3$，3位一组（称为一个八进制字节）表示一个八进制的数字。利用这样一种分组的对应关系，可以很方便地把一个二进制数转换为八进制数或者是十六进制数。

例2-9 将二进制数 110010011101_2 转换成八进制数和十六进制数。

110 010 011 101　八进制的转换以3位二进制数字为一组
6　2　3　5　$110010011101_2=6235_8$
1100 1001 1101　十六进制的转换以4位二进制数字为一组
C　9　D　$110010011101_2=C9D_{16}$ ■

如果分组的数字不够，可以在前面添加数字0。

2.4 带符号整数的表示方法

前面已经了解了怎样在不同的基数之间对无符号整数进行转换。我们需要一些附加的信息才能表示带符号的数字。当在程序中声明整数变量时，许多编程语言都会自动为该整数变量分配一个存储空间，并把该存储单元的第一位作为符号位。根据惯例，高位的“1”表示一个负数。这种存储单元可以是小到一个8位的字节，也可以大到几个字，这取决于编程语言和计算机的系统。符号位后面其余的位用来表示数字本身。

怎样表示一个带符号的数字取决于所使用的方法。通常有三种途径。最直观的方法为符号幅值表示法，即利用符号位后面的剩余的位数来表示数字的幅值（大小）。这种方法和其他两种采用互补（complement）概念的表示方法将在下面的章节中介绍。

2.4.1 符号幅值表示法

至此，我们还没有讨论如何利用二进制数的表示方法来表示负数。正整数和负整数的集合统称为

带符号的整数（signed integer）。将带符号的整数表示为二进制数值的关键问题是符号的表示方法，即如何对真实数字的符号进行编码。符号幅值表示法（signedmagnitude representation）就是解决这个问题的一种方法。顾名思义，一个带符号-幅值的数字的最左边的位（也称为高位，或者是最高有效的位）是一个符号，而余下的位用来表示数值的幅值（或称为绝对值）。例如，在 8 位字中，－1 表示为 10000001，而＋1 表示为 00000001。对于采用带符号幅值表示法和 8 位单元来存储整数的计算机系统，只有 7 位可以用来作为数字幅度的实际表示。这就是说，一个 8 位字可以表示的最大整数为 2^7-1，即 127（高位为 0，后面跟 7 个 1）。最小整数为 8 个 1，即－127。因此，N 位能够表示的数的范围为 $-2^{(N-1)}-1$ 至－ $(2^{(N-1)}-1)$。

当然，计算机必须能够执行采用这种符号表示的整数的算术运算。对符号幅值表示法进行的算术运算实际上与人们通常采用纸笔方法所进行的运算规则是相同的，但是这种运算很快就会让人糊涂起来。例如加法的运算规则。(1) 如果符号相同，大小相加，结果取相同的符号。(2) 如果符号相反，必须先决定哪一个操作数的幅值比较大。结果的符号应该与较大的幅值的操作数相同，结果幅值的大小为较大操作数的幅值减去其中较小操作数的幅值。仔细思考一下这种运算法则，这就是我们通常使用的带符号算术的手写运算方法。

我们可以基于数的符号，按照某种方式来安排操作数的次序，这样可以在运算时不必考虑它们的符号。而只要在计算结束后加上正确的符号即可。当采用上述思想在一个 8 位字的计算机系统中建立这样的运算规则时，需要非常小心地进行处理。结果的幅值只能有 7 位，对最高位（符号位）产生的进位必须舍弃。

例 2-10 利用符号幅值算术，对二进制数 01001111_2 和 00100011_2 进行加法运算。

```
        1 1 1 1      ⇐进位
0   1 0 0 1 1 1 1      (79)
0 + 0 1 0 0 0 1 1    ＋(35)
0   1 1 1 0 0 1 0      (114)
```

可见，如同十进制加法一样，符号幅值表示法的算术运算也是从右往左进行直到第 7 位，包括其中可以有进位。但是，如果是在最高位有进位，则表示有溢出发生，这个进位将会被丢弃，而导致错误的求和结果。本例中并没有发生这种溢出。

在带符号幅值表示中，上例的计算结果为 $01001111_2+00100011_2=01110010_2$。 ■

对符号位进行单独处理，是因为符号位只在加法完成后才有实际的意义。在例 2-10 中，所进行的是两个正数的求和，求和结果也是正数。如果运算结果的符号出现错误，也就表示带符号的数字发生了溢出现象，其计算结果也是错误的。

在符号幅值表示中，符号位仅仅用来表示数字的符号，因此不能对符号位进行进位运算。如果产生一个来自第 7 位的进位，运算结果将会对第 7 位作溢出处理，并对进位进行删节，同时给出一个错误的结果（例 2-11 就是发生这种溢出情形的一个例子）。谨慎的程序员为了避免这类“百万美元”的错误，通常都会非常仔细地检查各种可能发生的溢出情况，尽可能防止溢出。如果不丢弃溢出位，就会直接进位到符号位，产生两个正数求和结果为一个负数这类更加荒谬的结果（不妨可以设想一下，如果后续的程序还要对该结果进行平方根或对数运算，会发生什么样的情况）。

例 2-11 利用符号幅值算术，对二进制数 01001111_2 和 01100011_2 进行加法运算。

```
最后的进位   1←        1 1 1 1     ⇐进位
溢出         0     1 0 0 1 1 1 1      (79)
被摒弃       0+    1 1 0 0 0 1 1    ＋(99)
             0     0 1 1 0 0 1 0      (50)
```

得出一个错误结果：79＋99＝50。 ■

与加法运算一样，符号幅值表示法的减法运算也可以用类似于十进制算术的纸笔运算方法进行，只是有时需要从被减数（minuend）的数字上进行借位操作。

例 2-12 利用符号幅值算术，从二进制数 01100011_2 中减去二进制数 01001111_2。

```
        0 1 1 2        ←借位
0     1 1 0 0 0 1 1      (99)
0  −  1 0 0 1 1 1 1   −(79)
0     0 0 1 0 1 0 0      (20)
```

在符号幅值表示中，最后的运算结果为：$01100011_2-01001111_2=00010100_2$。 ■

例 2-13 利用符号幅值算术，从二进制数 01001111_2（79）中减去二进制数 01100011_2（99）。

仔细观察，不难发现减数 01100011 要比被减数 01001111 大。从例 2-12 的结果知道，这两个数的差为 00010100_2。因为减数比被减数大，所以现在所要做的就只是改变一下它们之间的差值的符号。为此，在符号幅值表示中有：$01001111_2-01100011_2=00010100_2$。 ■

大家知道，减法运算实际上就是加上与减数相反的数。即对要减去的数字取其负值，然后再进行相加而不是相减。这种做法通常要比执行减法的借位运算要来得容易，特别是在处理二进制数时更是如此。这里先回顾一下加法运算规则：(1) 如果符号相同，直接将幅值相加，其结果取相同的符号。(2) 如果符号不同，则需要先确定哪个操作数的幅值较大。结果的符号与幅值较大的操作数相同，而运算结果的大小为较大的幅值减去较小的幅值。

例 2-14 利用符号幅值算术，将两个二进制数 10010011_2（−19）和 00001011_2（+13）相加。

这里，第一个数（被加数）为负数，其符号位被设为 1。而第二个数（加数）为正数，所以现在实际上要做减法运算。首先，确定两个数的幅值的大小，取大的一个作为被减数，其符号就是运算结果的符号。

```
        0 1 2          ←借位
1     0 0 1 0 0 1 1      (−19)
0  −  0 0 0 1 1 0 1   +(13)
1     0 0 0 0 1 1 0      (−6)
```

包括符号位在内，符号-幅值表象中的计算结果为：$10010011_2-00001101_2=10000110_2$。 ■

例 2-15 利用符号幅值算术，从二进制数 10101011_2（−43）中减去二进制数 10011000_2（−24）。

可以先将减法运算转换成加法运算，对−24 取反得 24，再加上−43，这样原来的减法计算变成了求解新的问题：−43+24。由上述的加法规则可知，由于符号不同，实际上进行的运算是从两个操作数中较大的幅值中减去较小的幅值（或者说从 43 中减去 24，因为 43 大于 24），并取其结果为负值。

```
      0 2
    0 1 0 1 0 1 1      (43)
 −  0 0 1 1 0 0 0   − (24)
    0 0 1 0 0 1 1      (19)
```

注意，在完成减法运算前这里并没有涉及符号问题。很明显，答案应该为负值。因此，符号-幅值表象中最后的计算结果为：$10101011_2-10001100_2=10010011_2$。 ■

在讨论前面这些例子时，读者也许会产生一些疑问：哪个数字较大？怎样减去一个负数？需要从被减数中借位多少次？不难看出，按照上面这种工作方式设计的计算机在执行算术运算时需要做出如此多的判断和决定，尽管整个计算的过程会很快。事实上对于 0 这个数，符号-幅值表象中有两种表示方法：10000000 和 00000000。从数学上来说，这种情况根本就不可能发生，而且会造成逻辑和电路上的复杂性。因此，利用较简单的方法来表示带符号的数字，可以简化计算机的电路和降低电路的成本。这些简单的表示方法就是基于下面所述的补码体系的原理。

什么是倍乘转换法

一种将二进制数转换为十进制数的最快的方法称为*倍乘转换法*（double-dabble 或 double-dibble）。这种转换方法的基本思想是，二进制数的各位中，每个后面的位的 2 的乘幂（乘方）总是前面（其右边）位的 2 的乘幂的 2 倍。这种倍乘转换方法是从最左边的位开始进行计算，将第一位乘以 2 加到第

二位上去，得到的和再乘以 2 加上第三位，依此类推，直到最右边的位。

例 1

将二进制数 10010011_2 转换成十进制数。

步骤 1：写下该二进制数，注意在位与位之间留下间隔空间。

1 0 0 1 0 0 1 1

步骤 2：最高位乘以 2，复写到第二位的下面。

1 0 0 1 0 0 1 1
2
×2
2

步骤 3：与第二位相加后得到的和乘以 2，然后复写到第三位的下面。

1 0 0 1 0 0 1 1
2 4
+0
2
×2 ×2
2 4

步骤 4：重复步骤 3，直到完成所有的位。

1 0 0 1 0 0 1 1
2 4 8 18 36 72 146
+0 +0 +1 +0 +0 +1 +1
2 4 9 18 36 73 147 ←最后的结果为：$10010011_2=147_{10}$
×2 ×2 ×2 ×2 ×2 ×2 ×2
2 4 8 18 36 72 146

如果将十六进制数分成 4 位一组的二进制数（反之亦然），运用倍乘转换法可以非常方便地将十六进制数转换为十进制数。

例 2

将十六进制数 $02CA_{16}$ 转换成十进制数。

首先，通过十六进制数字进行分组，并将十六进制数转换为二进制数。

0 2 C A
0000 0010 1100 1010

然后，对二进制形式的数字运用倍乘转换法：

1 0 1 1 0 0 1 0 1 0
2 4 10 22 44 88 178 356 714
+0 +1 +1 +0 +0 +1 +0 +1 +0
2 5 11 22 44 89 178 357 714
×2 ×2 ×2 ×2 ×2 ×2 ×2 ×2 ×2
2 4 10 22 44 88 178 356 714

$02CA_{16}=1011001010_2=714_{10}$ ■

2.4.2 补码体系

研究算术的理论工作者早在几百年前就知道在十进制数中的减法运算中，可以通过加上减数与全 9 组成的数字的差，再加回一个进位的方法来实现。这种方法称为取减数 9 的补码。或者更正规地说，*求减数的十进制反码*（diminished radix complement）。例如，要求 167－52 的差。先求 999 减 52 的差，得到 947。这样，在 9 的补码算术中，有 167－52＝167＋947＝114。再将百位上的进位 1 加回到个位的位置上，就得到了正确的结果 167－52＝115。这种方法通常也称为“计算 9 的个数”，并且已经推广

到二进制数体系的运算中，以简化计算机的算法。这种补码体系相对于符号幅值的表示方法来说，其优点就是不再需要单独处理符号位。但是仍然可以通过检测最高位的方法，非常方便地了解一个数的符号。

理解补码体系的另外一种方法是，想像一个装在自行车上的里程表的工作过程。与汽车的里程表不同，当自行车做逆向骑行时，自行车里程表会倒转回去。假定自行车的里程表采用 3 位数字进行计数，例如表的计数从 0 开始，最后到 700 的位置结束。对此，我们无法确定自行车是向前行驶了 700 英里，还是向后逆向行驶了 300 英里。要解决这一问题的最简单的方法是简单地把计数的区间分成两半，利用 001－500 计算正的里程数，而使用 501－999 来计算负的里程数。这样就可以有效地把计算的距离缩减到里程表可以度量的范围内。如果里程表的读数为 997，我们就知道自行车是向后逆行了 3 英里，而不是向前行驶了 997 英里。501－999 表示数字 001－500 的*补码*（radix complement，即下面将要介绍的两种方法中的第二种），并用来表示负的里程数。

反码

如上所述，在基数为 10 的计数体系中，一个数的*反码*（one's complement，又称为 1 的补码）是通过将基数减去 1，即十进制中为 9，再减去该数字得到的。对于更一般的形式，如果已知基为 r，有 d 位数字，那么数字 N 的反码定义为 $(r^d-1)-N$。对于十进制数，$r=10$，基数减一就是 $10-1=9$。例如，2468 的十进制反码为 $9999-2468=7531$。这在二进制数的体系中也有相同操作，即从一个较小的基（2）中减去 1，即为 1。例如，0101_2 的反码为 $1111_2-0101=1010$。虽然我们也可以按照前面讨论的方法繁琐地进行借位和减法运算来计算反码，但做几个实验就可以发现，构成一个二进制数的反码很简单，就像将所有的 1 切换为 0，或者是将所有的 0 切换为 1 一样。在计算机硬件中，要实现这种切换翻位的操作是一件非常容易的事情。

值得注意的是，尽管我们可以求出任意十进制数和二进制数的反码，但是在这里最重要的是怎样利用反码形式来表示负数。大家知道，进行一个减法运算，比如 10－7，也可以设想为“加上一个相反的数”，这就是 10＋（－7）。利用反码表示法可以将减法运算转换成加法的运算形式，以简化减法运算。同时，反码表示法也提供了一种表示负数的方法。这里，我们不想使用特殊的二进制位来表示符号（就像在符号-幅值表象中所做的那样），所以需要记住的是如果一个数是负数，就应该将它转换成反码的形式。这种转化结果的最左边的二进制位应该是 1，表明是一个负数。当然，如果一个数是正数，则无需将其转换成反码的形式。所有正数最左边的位的位置上都应该是一个 0。例 2-16 用来说明这些概念。

例 2-16 利用 8 位二进制数的反码形式表示 23_{10} 和 -9_{10}。

$23_{10}=+(00010111_2)=00010111_2$

$-9_{10}=-(00001001_2)=11110110_2$ ■

假设要从 23 中减去 9。要执行反码的减法，首先要将减数 9 表示成反码的形式，然后再与被减数 23 相加，这样就得到了－9＋23。最高位会有一个 1 或者是 0 的进位，就把它加到和的最低位。这个过程称为*高低两端的进位循环*（end carry-around），产生的原因是使用了反码。

例 2-17 利用反码算术，进行十进制数的加法：$23_{10}+(-9_{10})$。

```
                1 ← 1 1 1   1 1          ⇐进位
                  0 0 0 1 0 1 1 1          (23)
最后的进位被加    +1 1 1 1 0 1 1 0        +(−9)
到求和结果中。      0 0 0 0 1 1 0 1
                              +1
                  0 0 0 0 1 1 1 0         14₁₀
```

■

例 2-18 使用反码算术，进行十进制数的加法：$9_{10}+(-23_{10})$。

```
最后的进      0 ← 0 0 0 0 1 0 0 1          (9)
位为0，求和     +1 1 1 0 1 0 0 0      + (−23)
计算完成。      1 1 1 1 0 0 0 1         −14₁₀
```

■

怎样知道求和的结果 11110001_2 实际上就是 -14_{10} 呢？只需简单地取这个二进制数的反码（记住，因为最左边的位为 1，所以这个数必为负数）。11110001_2 的反码是 00001110_2，即为 14。

反码表示法的主要缺点是有两种 0 的表示法：00000000 和 11111111。由于各种原因，对于二进制数，计算机工程师们在很久以前就已经停止使用了反码表示法，而是采用一种更有效的补码表示法。

补码

2 的补码（two's complement，简称为补码）是补码（radix complement）体系的一个特例。假设在基数为 r 的计数体系中，数 N 由 d 位数字组成。如果 $N\neq 0$，数 N 的补码定义为 r^d-N；如果 $N=0$，数 N 的补码定义为 0。通常补码比反码更加直观。回顾前面的自行车里程表的例子，向前行驶 2 英里的十进制的补码为 $10^3-2=998$，这就是前面所假设的负（向后逆行）的距离。类似地，在二进制数中，4 位数字 0011_2 的补码为 $2^4-0011_2=10000_2-0011_2=1101_2$。

仔细观察，不难发现补码其实只不过是反码加 1。要求一个二进制数的补码（称为取补），只需简单地将二进制数字的各个位进行翻位取反操作，然后再加 1 即可。这样处理也简化了加法和减法运算。因为减数（要取补和相加的数）在开始时就进行了加 1 操作，所以在减法的运算过程中就不再会有高低两端的进位循环问题。只需简单地舍弃所有与最高位有关的进位。需要牢记，只有负数才需要转换成补码形式。

例 2-19 采用 8 位二进制补码形式表示十进制数：23_{10}、-23_{10} 和 -9_{10}。

$23_{10}=+(00010111_2)=00010111_2$

$-23_{10}=-(00010111_2)=11101000_2+1=11101001_2$

$-9_{10}=-(00001001_2)=11110110_2+1=11110111_2$ ■

如果已知一个数的二进制表示，怎样来求其对应的十进制数呢？对于正数来说，这相当容易。例如，要将 00010111_2 的补码值转换成十进制数，只需简单地将这个二进制数转换成十进制数，即得到 23。但是，要转换表示成补码的负数，却要采用类似于从十进制数到二进制数转换的逆向操作过程。假设已知 11110111_2 为补码形式的二进制数值，求出相应的十进制数。这是一个负数，并且是用补码的形式表示的。首先要对该数字的每一位进行取反，然后加 1（即求出反码再加 1）。结果如下：$00001000_2+1=00001001_2$。这个数值等于十进制数 9。当然，因为开始时这个数字就是负数，因此二进制数 11110111_2 对应的十进制数即为 -9。

下面的两个例子表示的是怎样使用补码表示来进行加法运算。当然，也包括减法运算，因为减去一个数等于加上这个数的反数。

例 2-20 利用补码算术，计算加法：$9_{10}+(-23_{10})$。

```
  0 0 0 0 1 0 0 1      (9)
+ 1 1 1 0 1 0 0 1    +(-23)
-------------------
  1 1 1 1 0 0 1 0     -14
```

■

这里留给读者作为一个练习去验证，使用补码表示法的二进制数 11110010_2 实际上就是 -14_{10}。

例 2-21 利用补码算术，对十进位制数 -9_{10} 和 23_{10} 求和。

```
  1 ← 1 1 1   1 1 1        ⇐进位
摒弃   0 0 0 1 0 1 1 1        (23)
进位 + 1 1 1 1 0 1 1 1      +(-9)
       0 0 0 0 1 1 1 0       14
```

■

注意，在例 2-21 中摒弃的进位不会导致错误的结果。如果是两个正数相加，则可能产生溢出，结果是负值。而两个负数相加，也可能产生溢出，结果却是正的。当使用补码表示法时，一个正数和一个负数相加是不可能出现溢出的。

利用简单的计算机电路可以很方便地检测出溢出的条件，所采用的法则也很容易记忆。读者可能注意到，在例 2-21 中，进入符号位的进位（即从符号位的前一位进到符号位的一个 1）与移出符号位

的进位（即移出符号位并且被摒弃的一个 1）是相同的。如果这两个进位相等，表示没有溢出发生。如果它们不相等，就在算术逻辑单元中设置一个溢出指示，指出该结果不正确。

检测溢出条件的一个简单法则： 如果进入符号位和移出符号位的进位相等，那么没有溢出发生。如果不同，就有溢出发生，也就是出现一个错误。

难点在于程序员（或编译器）要不断地去校验这个溢出条件。例 2-22 就表示发生了一个溢出现象，原因是进入符号位的进位（移入的是 1）不等于移出符号位的进位（移出的是 0）。

例 2-22 利用补码算术，对二进制数 126_{10} 和 8_{10} 求和。

```
   0 ← 1 1 1 1            ⇐进位
摒弃    0 1 1 1 1 1 1 0     (126)
进位  +0 0 0 0 1 0 0 0     +(8)
      1 0 0 0 0 1 1 0  (−122???)
```

其中，一个 1 进位到最左边的位，而一个 0 从最左边的位移出来。因为这两个进位不相等，所以有一个溢出发生。读者可以很容易地看到，两个正数相加，其结果却是一个负数。 ■

补码是目前最通用的带符号的数字的表示方法。对于加法和减法来说，补码算法非常简单。而对于数值 0（所有的位都是 0）具有最好的表示方法。补码算法具有自反特征，很容易扩展到较大位数的数值。但是，这种补码表示方法的最大缺点在于，由 N 位二进制数所表示的数值的范围具有非对称性。例如，采用符号幅值表示法，4 位二进制数可以表示的数值范围是 −7 到 +7。而采用补码表示法时，却可以表示从 −8 到 +7 之间的数字。这常常会使读者在学习补码表示法时感到有些困惑。要理解为什么采用 4 位补码表示法所能表示的最大数为 +7，只要记住因为二进制补码的第一位必须是 0。如果剩下的位全部都是 1（可能表示的最大幅值），则为 0111_2，这就是 7。读者马上会联想到最小的一个负数应该是 1111_2。但是，在补码表示法中，二进制数 1111_2 实际上代表的是 −1（对所有的位进行取反，然后加 1，并且取负数）。然而，怎样用 4 位补码来表示 −8？二进制数 1000_2 就是。大家知道这是一个负数。如果将其所有位取反（为 0111），再加 1（得到 1000，即为 8），最后取负数，即为 8。

整数乘法和除法

除非使用巧妙的算法，否则乘法和除法都需要花费相当多的计算周期才能得出结果。这里仅仅讨论这些运算的最直接的方法。在实际的系统中，有专门设计的硬件用来优化乘法和除法的计算过程，有时是实现并行计算。有兴趣的读者可以参考本章结尾所引证的一些参考文献来探究一下这些先进的计算方法。

计算机使用的最简单的乘法运算与我们用纸和笔所进行的乘法运算相类似。一个完整的计算机的乘法表是再简单不过的：0 乘以任何数得 0，而 1 乘以任何数的结果还是原来的数值。

为了说明这种简单的计算机的乘法运算，先把乘数和被乘数写到两个不同的存储空间中，再留出第三个空间位置来存储乘积结果。从最低位开始，对乘数的每一位设置一个指示器（指针）。对应于乘数中的每一位，被乘数都依次向左边移动一位。当乘数为 1 时，被移位的被乘数就被加到一个部分积的连续求和中。由于是按照乘数中的每一位来移位被乘数，因此存放乘积的工作空间要求是乘数或者是被乘数的两倍。

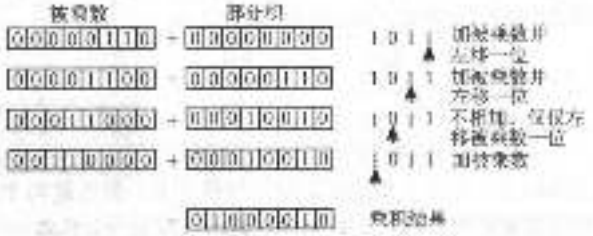
对于二进制数的除法运算可以用两种简单的方法来实现：或者使用从除数中连续替代减去分母的方法，或者采用我们在小学课堂中学过的试算法方式的长除法。正如前面所述，包括乘法在内，最有效的计算二进制数除法的方法已经超出了本书的范围，读者可以在本章结尾的参考资料中获取有关信息。

不论所使用的各种算法的相对效率如何，除法都是一种可能引起计算机系统发生崩溃的运算。特别是在尝试除以 0 或者是出现两个操作数的大小相差极为悬殊的情形。当除数远小于被除数时，就会产生一种“除法下溢（divide underflow）”的情况，也就是计算机视为等效被 0 除，这种计算过程是不

可能实现的。

计算机的整数除法和浮点除法之间有显著的区别。对于整数除法，其结果由两部分组成：商和余数。而浮点除法的结果则表示为一个二进制的分数的形式。由于这两类除法的运算过程差异非常大，所以需要各自使用专门的电子线路来分别实现这两种不同除法运算。浮点计算采用的是专门设计的所谓浮点单元（floating point units，FPU）的部件来实现的。

例 求 00000110_2 和 00010011_2 的积。



■

2.5 浮点表示法

在构建实际的计算机系统时，可以采用前面刚刚介绍的任何一种整数表示方法。下面将选取其中一种表示方法来实现计算机系统的设计任务。接下来工作就是决定计算机系统的字长。如果希望设计的系统不是太贵的话，就选择较小的字长，例如16位。在设置了一个符号位后，系统可以存储的最大整数是32767。下面要做的工作就是设法使设计的系统满足一个潜在顾客的需求：要对本年度购票进场观看专业体育比赛的观众人数进行登记。毫无疑问，这个数目会大于32767。当然，只需要将计算机系统的字长的位再扩大一些，比如说32位，这个问题就可以解决。现在设计的计算机的字长已经足以应对人们所面临的各种计数问题。但是如果这个顾客想知道每个观众在娱乐时间内每分钟所消费的金额数目，这个数目可能是一个小数。这时计算机又该怎么做呢？

最简单和最节约的办法是维持采用16位的计算机系统，并且对自己说“嗯，这里只是设计一个廉价的计算机系统。要想用它来完成一些神奇的工作，做一个好的程序设计员不就行了”。尽管这种说法从现代技术的角度听起来有点过于轻率，但它却是计算机早期发展年代的真实情况。早期的微处理器或计算机主机都没有浮点运算单元。事实上，好多年来，都是通过精巧的编程来使得当时的那些整数机器系统能够像浮点机器一样来实现浮点运算。

如果熟悉科学计数法的话，就会联想到怎样来处理浮点运算问题，即在一个整数机的系统中实现所谓浮点仿真（floating-point emulation）。在科学计数法中，数字被表示成两部分：小数部分，称为尾数（mantisse），和一个指数部分，写成10的幂指数形式。这两部分的乘积就代表了要表示的数。例如，在科学计数法的表示中，32767就写成了 3.2726×10^4。科学计数法可以简化一些涉及到很大的数字或很小的数字的计算过程。同时，科学计数法也是现代数字计算机中浮点计算法的基础。

2.5.1 一个简单的模型

在数字计算机系统中，浮点数字由三部分组成：符号位、指数部分（按照2的幂指数形式表示）和小数部分，称为有效数（significand，这是关于尾数一词的另一种表述）。指数部分和有效数部分所使用的位数取决于对数值表示范围（范围越大，使用的指数位数越多）和精度（精度越高，需要的有效数的位数就越多）的要求。在本节的后面部分，我们将使用一个14位模型，其中，5位指数，8位有效数和1位符号（参见图2-2），更加通用的表示形式将在第2.5.2节中介绍。



图2-2 浮点表示法

假设要在模型中存储一个十进制数17。大家知道，$17=17.0\times10^0=1.7\times10^1=0.17\times10^2$，类似

地，在二进制体系中，$17_{10}=10001_2\times2^0=1000.1_2\times2^1=100.01_2\times2^2=10.001_2\times2^3=1.0001_2\times2^4=0.10001_2\times2^5$。如果采用最后面的形式，小数部分是 10001000，而指数部分是 00101，如下所示。

0	0	0	1	0	1	1	0	0	0	1	0	0	0

采用这种形式，可以存储的数要比采用 14 位定点（fixed-point）表示法（需要总共使用了 14 位二进制数字，加上一个二进制小数点，或者简称小数点）所能表示数大得多。在这一模型中，$65536=0.1_2\times2^{17}$表示为：

0	1	0	0	0	1	1	0	0	0	0	0	0	0

这个模型存在的一个明显问题是没有负指数的表示。如果要存储分数 0.25，就无法完成。原因是 0.25 是 2^{-2}，指数部分为－2，无法在这个模型中表示。显然，可以通过在指数部分增加符号位的方法来解决这个问题。但是，通常更有效的做法是使用偏移（biased）指数，这样在比较两个浮点数的数值时，可以采用比较简单的整数电路。

使用偏移指数的思想就是要在允许的范围内将每一个整数都转化成一个非负整数，这样就可以把每个整数都作为一个二进制数来存储。在指数允许的表示范围之内，首先要通过在每个指数上加一个固定偏移值的方法来对整数加以调整。一般取偏移值为靠近指数可能取值范围中间的某个值，并且选择偏移值来表示 0。例如在本例中，可以选择 16 作为偏移值，因为 16 位于 0 和 31 的中间（这里指数部分占 5 位，允许的值为 2^5，或 32）。任何大于 16 的指数将表示为一个正值，而小于 16 的指数表示为一个负值。这种方法被称为 16 -余（excess-16）表示法，因为需要减去 16 才能得到真正的指数值。要注意的是，通常情况下全部为 0 和全部为 1 的指数是为某些特定的数所保留的（例如 0 或者是无限大）。但在本例中，允许全部为 0 和全部为 1 的指数出现。

现在回到前面存储数字 17 的例子，计算 $17_{10}=0.10001_2\times2^5$，偏移指数即为 16＋5＝21：

0	1	0	1	0	1	1	0	0	0	1	0	0	0

如果要存储 $0.25=0.1\times2^{-1}$，则为如下的表示：

0	0	1	1	1	1	1	0	0	0	0	0	0	0

这种数据体系仍然存在一个相当大的问题，即对每个数的表示方法并不是唯一的。下面的表示都是等效的：

0	1	0	1	0	1	1	0	0	0	1	0	0	0

=

0	1	0	1	1	0	0	1	0	0	0	1	0	0

=

0	1	0	1	1	1	0	0	1	0	0	0	1	0

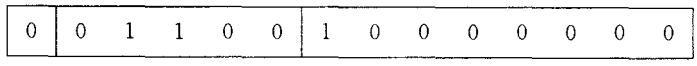
=

0	1	1	0	0	0	0	0	0	1	0	0	0	1

因为这一类的对称的表示形式不能很好地适用于数字计算机系统，所以大家接受了这样一种惯例，就是将有效数的最左边的位总是设置为 1。这种惯例称为规格化（normalization）。这种惯例做法还有一个好处就是，这个最左边的 1 对于有效数给出了一位额外的精度。

例 2-23 利用带 16-余偏移指数的规格化浮点表示十进制数 0.03125_{10}。

$0.03125_{10}=0.00001_2\times2^0=0.0001\times2^{-1}=0.001\times2^{-2}=0.01\times2^{-3}=0.1\times2^{-4}$。应用偏移值，指数部分为 16−4=12。

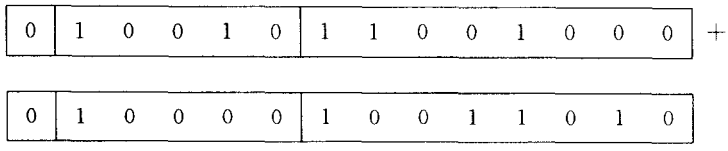


本例中并没有使用归范化表示法来表示这个数字，因为它已经包含了 1 在内。 ■

2.5.2 浮点算法

大家知道，要对两个用科学计数法表示的十进制数进行加法运算，例如 $1.5\times10^2+3.5\times10^3$，首先要将这两个数表示成具有同一基数的相同指数形式，然后才能将小数部分相加。即，$1.5\times10^2+3.5\times10^3=0.15\times10^3+3.5\times10^3=3.65\times10^3$。浮点加法和减法的原理与此相同，说明如下。

例 2-24 对下面两个利用带 16 -余偏移指数的规格化浮点形式表示的二进制数进行加法运算。



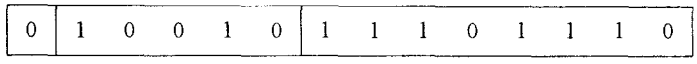
不难发现，加数比被加数要高两阶指数，利用二进制小数点移位可以将这两个数对齐：

11.001000

+0.10011010

11.10111010

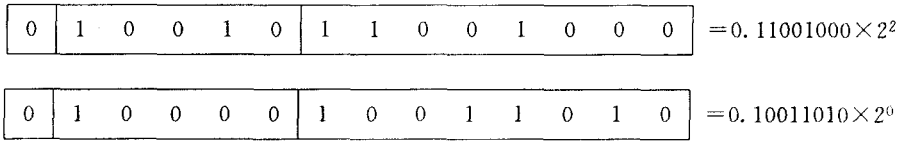
重新进行规格化处理，保留那个较大的指数，并删除有效数的低位，得到：



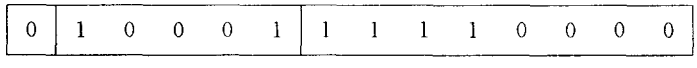
■

乘法和除法采用与十进制算法中的指数运算的相同法则进行，例如，$2^3\times2^4=2^1$。

例 2-25 乘法



0.11001000 乘以 0.10011010 的积为 0.0111100001010000，然后乘以 $2^2\times2^0=2^2$ 得到 1.1110000101。重新进行规格化，留取合适的指数，得到的浮点积为：



■

2.5.3 浮点误差

当我们用笔和纸求解三角代数问题和计算一个投资的回报利率时，很明显使用的是实数体系。实

数体系是一个无限区间，因为对于任意给定的一对实数，都可以找到另外一个实数位于这两个实数之间。

与人们想像中的数学不同，计算机是一个有限的系统，其存储能力也是有限的。在运用计算机进行浮点运算时，就是在运用一个有限的整数系统对无限的实数系统行建模。所做的一切事实上只是实数系统的一个近似（approximation）。使用的位数越多，这种近似就越好。但是，不管计算机使用多少位数，总是会存在一些误差。

浮点误差可能是明显的，也可能是细微的，或者是不被人们所注意的。明显的误差，比如，数的上溢或下溢是一些可能造成程序崩溃的大的误差。而一些细微的误差，常常会产生一些错误的结果，并且在发生问题之前一般很难检查出来。例如，在上面的简单模型中，我们可以表示从$-0.11111111_2 \times 2^{15}$到$+0.11111111_2 \times 2^{15}$之间范围内的规格化数值。很明显，这个模型不能储存$2^{-19}$，或是$2^{218}$，因为它们根本不在允许的范围内。但是，这个模型不能精确地存储数值 128.5 这一事实却不是那么明显，因为这个数就在模型适用的范围之内。如果把 128.5 转换成二进制数，就是 10000000.1，数值的长度有 9 位数字。模型中的有效数的位数只有 8 位。通常，低位的数字会被丢弃，或者舍入到相邻的高位。然而，不管怎样来处理这个问题，都已经在系统中引入了误差。

可以通过求误差的绝对值和数字的真实值的比例，来计算所采用的表示方法的相对误差。对于 128.5 的例子，有：

$$\frac{128.5-128}{128.5}=0.00389105\approx 0.39\%$$

如果不仔细处理的话，这个误差会在某些长串的计算过程中不断传递下去，造成计算精度上的重大误差。图 2-3 显示了这样一个误差的传递过程，图中使用了 14 位模型进行 16.24 乘以 0.91 的多次迭代运算。将这些数转换成二进制数时，开始就可以看到存在明显的误差。

乘数		被乘数	14 位乘积	真实乘积	误差
1000.001 (16.125)	×	0.11101000 = (0.90625)	1110.1001 (14.5625)	14.7784	1.46%
1110.1001 (14.5625)	×	0.11101000 =	1101.0011 (13.1885)	13.4483	1.94%
1101.0011 (13.1885)	×	0.11101000 =	1011.1111 (11.9375)	12.2380	2.46%
1011.1111 (11.9375)	×	0.11101000 =	1010.1101 (10.8125)	11.1366	2.91%
1010.1101 (10.8125)	×	0.11101000 =	1001.1100 (9.75)	10.1343	3.79%
1001.1100 (9.75)	×	0.11101000 =	1000.1101 (8.8125)	8.3922	4.44%

图 2-3　14 位浮点数字中的误差传递

经过 6 次乘法迭代运算后，乘积的误差已经增大到三倍。继续不断地迭代运算下去，最终会产生 100%的误差，因为乘积的结果最后等于 0。尽管 14 位模型太小，以至于误差被夸大了，但是所有的浮点系统的工作方式都是相同的。不管把系统设计得有多么大，利用一个有限的系统来表示一个实数，或多或少会引入不同程度的浮点误差。即使是最小的误差，也会导致灾难性的后果，尤其是使用计算机对一些物理事件进行控制时，例如在军事和医疗技术的应用中。计算机科学家所面临的一大挑战就是寻找最有效的算法；把这些误差控制在性能和经济允许的范围内。

2.5.4 IEEE-745 浮点标准

前面引入浮点模型的意义在于简化问题和便于理解浮点的概念。现在可以把这个模型推广到所需的任意位数的数字。直到上个世纪 80 年代以前，有关这方面的的工作还只是处于随意和杂乱的状态，没有统一的标准，这导致了不同制造商的系统存在着许许多多不兼容的表示方法。1985 年，电子与电气工程师协会（IEEE）发表了一份有关单精度和双精度浮点数的浮点表示标准。这个标准官方称为 IEEE-754（1985）。

IEEE-754 单精度标准采用 8 位指数，一个 127 余的偏移值和 23 位的有效数。包括一个符号位，单精度浮点数的总字长为 32 位。当指数等于 255 时，表示的量是±无穷大（有限位数是 0）或者“非数”（具有非 0 的有效位）。“非数”或 NaN，用来表示一个非实数的值，并且通常用做一个错误指示。

双精度数字采用带符号的 64 位字，其中 11 位指数和 52 位有效数字。偏移值是 1023。IEEE 双精度模型可以表示的数字范围如图 2-4 所示。当指数为 2047 时，表示 NaN。

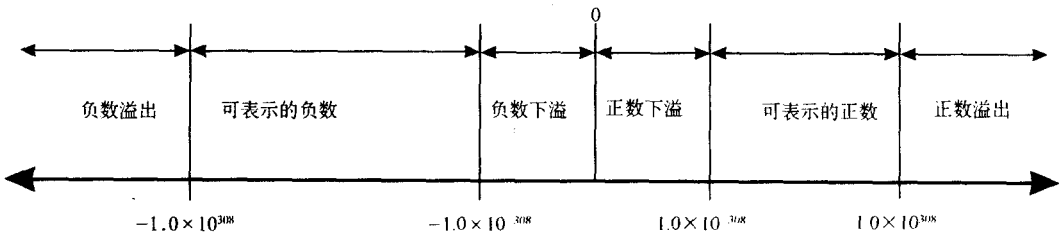


图 2-4 IEEE-754 双精度浮点数表示的范围

从性能的角度考虑，由于增加的成本不多，所以大多数的 FPU 都只是使用 64 位模型。因此，只需设计一组特殊的电路来实现双精度模型的运算。

无论是单精度还是双精度 IEEE-754 模型都有两种表示 0 的方法。当指数和有效数全部为 0 时，所存储的数量就是 0。这时候，符号位中存储的是什么数字无关紧要。因此，程序设计人员在对一个浮点值和 0 进行比较时，就需要特别的小心。

事实上，近来设计的每一种计算机系统都采用 IEEE-754 的浮点模型。不幸的是，在该标准公布以前，许多主流计算机系统都建立了它们自己的浮点体系。对于比较成熟的计算机体系结构，例如 IBM 主流机型，差不多花了十几年时间来适应这种新的浮点体系。现在，这些系统既支持传统的浮点体系，也支持 IEEE-754 体系。但是，1998 年以前，IBM 的计算机系统一直都采用与源自于 1964 年最初在 System/360 中所使用的浮点算术完全相同的体系结构。由于大量老的软件仍然在使用，所以人们希望新的计算机系统能够继续支持两种浮点体系。

2.6 字符编码

数字计算机使用二进制体系来表示和处理数字值。现在要考虑怎样可以将这些计算机内部使用的数值转换成人们所能理解的形式。这种转换方式既取决于计算机使用的编码系统，也和这些数值的存取方式有关。

2.6.1 二进制编码的十进制数

二进制编码的十进制数（binary-coded decimal，BCD）是最初在 IBM 主流机型和中间过渡机型上使用的数字编码系统。BCD 是将每个十进制数字编码成一个 4 位的二进制数形式。如果按照 8 位字节的形式进行存储时，其中的高 4 位称为区位（zone），而低 4 位称为数字（digit）。这种转换方式来源于穿孔卡片年代的思想，在穿孔卡片每一列的最上面 2 行的某一行中有一个“区位穿孔”，而在最下面的 10 行的某一行中有一个“数字穿孔”。BCD 字节的高 4 位表示符号，使用了其中的 3 个数值：1111 表

示无符号数，1100 表示正数，而 1101 则表示负数。十进制数字的 BCD 编码方式如图 2-5 所示。

从图中可以看出，还有 6 个二进制数值编码没有被使用：从 1010 到 1111。虽然看起来几乎有 40%的资源被浪费，但是在表示精度方面却得到了很大的提高。例如，十进制数 0.3 在存储为二进制数时是一个循环小数，存储时需要截取成一个 8 位小数，如果再从二进制数转换回十进制数却为 0.296875，结果造成了一个近似为 1.05%的误差。而在 BCD 编码系统中，十进制数 0.3 直接的存储为 0011 0011（这里假定了数据格式中隐含有小数点），根本就没有误差。

十进制数字	BCD 编码
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
区位	
1111	无符号数
1100	正数
1101	负数

图 2-5 二进制编码的十进制数

因为 BCD 编码数的数字只占一个 4 位的半字节，所以在将多个相连的数字放入相邻的半字节时，只需留下一个 4 位的半字节给出符号。这样可以简化计算，并且可以节省空间。这个过程称为压缩（packing），按照这种方式存储的十进制数称为压缩的十进制数字（packed decimal number）。

例 2.26 利用压缩的 BCD 编码在 3 个字节中表示−1265。

1265 的区位-数字编码形式为：

1111 0001 1111 0010 1111 0110 1111 0101

压缩后，这串字符变成：

0001 0010 0110 0101

在低位数字后面加上符号，在 3 个字节的高位补足 1，则有：

0000 0001	0010 0110	0101 1101

■

2.6.2 EBCDIC

在开发 IBM System/360 之前，IBM 公司一直使用 6 位 BCD 编码的变化形式来表示字符和数字。但是，这种编码方式在怎样表示和处理数据方面受到很大的限制。事实上，小写字母不能用作当时 IBM 机器指令系统的一部分。当时，System/360 的设计者们需要有更大的信息处理能力以及一种能够同时存储数字和数据的统一方法。为了保证与以前的计算机系统及其外围设备具有兼容性，IBM 公司的工程师们决定最好的方法是直接将 BCD 编码从 6 位扩展到 8 位。因此，这种新的编码方式称为扩展的二-十进制编码的交换代码（Extended Binary Coded Decimal Interchange Code，EBCDIC）。IBM 公司继续在其主流计算机和运算机型中使用 EBCDIC 编码。图 2-6 按照区位-数字的形式给出了 EBCDIC 代码。字符表示采用在区位后面添加数字位的方法。例如，在 EBCDIC 代码表示中，字符 a 是 1000 0001，数字 3 为 1111 0011。大写字母和小写字母的区别仅仅在于第 2 位不同。这里，只需简单地进行 1 位数字的翻位，即可实现大小写字母之间的转换。这种区位编码形式也同样使得程序员很容易测试输入数据的正确性。

2.6.3 ASCII

当 IBM 公司忙于建造自己具有创新思想的 System/360 时，其他设备制造商们却正在努力设计系统之间传输数据的各种更好的方式。美国信息交换标准代码（American Standard Code for Information Interchang，ASCII）就是其中之一。ASCII 是从使用了几十年的电传打字（telex）设备的编码方案中直接衍生出来的。这些电传设备使用 5 位摩尔编码。这是 19 世纪 80 年代发明的，起源于波特（Baudot）码的编码方式。到了 20 世纪 60 年代，这种 5 位编码方式的局限性已经变得非常明显。国际标准化组织（ISO）就设计了一种 7 位的编码方案，称为第 5 种国际字母。1967 年，这种字母表成为官方标准，就是我们现在所称的 ASCII 码。

数字

区位	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0000	NUL	SOH	STX	ETX	PF	HT	LC	DEL		RLF	SMM	VT	FF	CR	SR	SI
0001	DLE	DC1	DC2	TM	RES	NL	BS	IL	CAN	EM	CC	CU1	IFS	IGS	IRS	IUS
0010	DS	SOS	FS		BYP	LF	ETB	ESC			SM	CU2		ENQ	ACK	BEL
0011			SYN		PN	RS	UC	EOT				CU3	DC4	NAK		SUB
0100	SP										[	.	<	(	+	!
0101	&										]	$	*	)	;	^
0110	-	/									|	,	%	_	>	?
0111										`	:	#	@	'	=	"
1000		a	b	c	d	e	f	g	h	i						
1001		j	k	l	m	n	o	p	q	r						
1010		~	s	t	u	v	w	x	y	z						
1011																
1100	{	A	B	C	D	E	F	G	H	I						
1101	}	J	K	L	M	N	O	P	Q	R						
1110	\		S	T	U	V	W	X	Y	Z						
1111	0	1	2	3	4	5	6	7	8	9						

缩写词

NUL	空	TM	带标符	ETB	传输块结束
SOH	标题开始	RES	还原	ESC	转义
STX	文本开始	NL	新行	SM	设置模式
ETX	文本结束	BS	退路	CU2	Customer use 2
PF	切出	IL	空位	ENQ	询问
HT	水平制表	CAN	取消	ACK	肯认
LC	转换成小写	EM	载体结束	BEL	Ring the bell(beep)
DEL	删除	CC	光标控制	SYN	同步
RLF	翻转换行	CU1	Customer use 1	PN	切入
SMM	开始手册消息	IFS	互换文件分隔符	RS	记录分隔
VT	垂直制表	IGS	互换组分隔符	UC	转换成大写
FF	换页	IRS	互换记录分隔符	EOT	End of transmission
CR	回车	IUS	互换单元分隔符	CU3	Customer use 3
SO	移出	DS	数位选择	DC4	设备控制 4
SI	移入	SOS	有效位开始符	NAK	Negative acknowledgement
DLE	数据链路转义	FS	字段分隔符	SUB	换代字符
DC1	设备控制 1	BYP	By pass	SP	空格
DC2	设备控制 2	LF	换行		

图 2-6 EBCDIC 编码（图中的数值采用的是二进制区位-数字的形式）

从图 2-7 中可以看出 ASCII 代码定义了 32 个控制字符，10 个数字，52 个英文字母（大写和小写字母），32 个的特殊字符（例如，$ 和 #），以及空格符号。最高位（第 8 位）准预留用做奇偶校验位。

奇偶校验（parity）在错误检测方案中是最基本的方法。对于类似在电传打字这样简单的设备中都可以很方便地执行奇偶校验。奇偶校验位的“开”和“关”取决于字节中其他各位的和是偶数还是奇数。例如，如果采用偶数奇偶校验，在传送 ASCII 字符 A 时，低 7 位就是 100 0001。因为这些位的和是偶数，奇偶校验位将被设置为关，传送的数据就是 0100 0001。类似地，在传送 ASCII 字符 C，100 0011，奇偶校验位在传送 8 位字节前将会被设置为开，即 1100 0011。奇偶校验只能用来检测一些单位（single-bit）的错误，第 2.8 节将会讨论一些更高级的错误检测方法。

考虑到与通信设备的兼容性问题，计算机制造商更倾向于使用 ASCII 代码。然而，随着计算机硬件的性能变得越来越可靠，对奇偶校验位的需求开始减弱。在 20 世纪 80 年代，微型计算机和外围设备的制造商开始利用奇偶校验位来为 128_{10} 和 255_{10} 之间的数值提供一个“扩展的”字符集。

根据不同的设备制造商，这些具有较高数值的字符可以是从数学符号到特殊的外语字符，如 ñ。不过，这些努力并未能使 ASCII 成为一种真正的国际交换代码。

2.6.4 统一字符编码标准

无论是 EBCDIC 码，还是 ASCII 码，都是建立在拉丁语系字母的基础上的。因此，这些编码方式

0 NUL	16 DLE	32	48 0	64 @	80 P	96 `	112 p
1 SOH	17 DC1	33 !	49 1	65 A	81 Q	97 a	113 q
2 STX	18 DC2	34 "	50 2	66 B	82 R	98 b	114 r
3 ETX	19 DC3	35 #	51 3	67 C	83 S	99 c	115 s
4 EOT	20 DC4	36 $	52 4	68 D	84 T	100 d	116 t
5 ENQ	21 NAK	37 %	53 5	69 E	85 U	101 e	117 u
6 ACK	22 SYN	38 &	54 6	70 F	86 V	102 f	118 v
7 BEL	23 ETB	39	55 7	71 G	87 W	103 g	119 w
8 BS	24 CAN	40 (	56 8	72 H	88 X	104 h	120 x
9 TAB	25 EM	41)	57 9	73 I	89 Y	105 i	121 y
10 LF	26 SUB	42 *	58	74 J	90 Z	106 j	122 z
11 VT	27 ESC	43 +	59	75 K	91 [	107 k	123 {
12 FF	28 FS	44 ,	60 <	76 L	92 \	108 l	124 |
13 CR	29 GS	45 -	61 =	77 M	93]	109 m	125 }
14 SO	30 RS	46 .	62 >	78 N	94 ^	110 n	126 ~
15 SI	31 US	47 /	63 ?	79 O	95 _	111 o	127 DEL

[illegible]

NUL 空	DLE 数据链路转义
SOH 标题开始	DC1 设备控制1
STX 文本开始	DC2 设备控制2
ETX 文本结束	DC3 设备控制3
EOT 传输结束	DC4 设备控制4
ENQ 查询	NAK 否定应答
ACK 确认	SYN 同步
BEL 提示音（蜂鸣）	ETB 传输块结束
BS 退格	CAN 取消
HT 水平制表	EM 载体结束
LF 换行，新行	SUB 替换
VT 垂直制表	ESC 转义
FF 换页，新页	FS 文件分隔符
CR 回车	GS 组分隔符
SO 移出	RS 记录分隔符
SI 移入	US 单元分隔符
	DEL 删除/空闲

图 2-7 ASCII 代码（图中的数值采用十进制数字的形式）

在对非拉丁字母提供数据表示的能力方面受到了很大限制，而全球大多数人都使用非拉丁语系。随着全世界所有的国家都在开始使用计算机，每个国家都在设计出一套能够最有效地表示自己本国语言的编码。然而，这些编码都不能与其他国家的编码相互兼容，这样也就在融入全球经济的道路上设置了另一种障碍。

为了防止这种情况继续恶化，1991 年成立了一个由工业和社会领导人组成的协会，并建立了一种新的国际信息交换代码，称为统一字符编码（unicode）。这个组织也被恰当地命名为统一字符编码协会。

统一字符编码是一个 16 位编码的字母表，可以向下兼容 ASCII 码和拉丁 1 字符集，并且与 ISO/IEC 10646-1 国际字母表相一致。因为统一字符编码的基础是 16 位编码，所以能够对全世界所使用的每一种语言的大多数字符进行编码。如果这 16 位编码仍不够用，统一字符编码还定

字符类型	字符集说明	字符数目	十六进制数字
字母表	拉丁字母、希腊字母（Cyri）字母、希腊字母等	4192	0000 至 1FFF
符号	右箭符号、数学符号等	4096	2000 至 2FFF
CJK	中文、日文、韩文、语音符号和标点符号	4096	3000 至 3FFF
Han	统一的中文、日文和韩文	40,960	4000 至 DFFF
	Han 的扩展或重复	4096	E000 至 EFFF
用户定义		4095	F000 至 FFFF

图 2-8 统一字符编码的编码空间分配

义了一种扩展机制，允许人们再对100万个额外的字符进行编码。这样，统一字符编码足以对人类文明发展历史上已有的每一种书写语言进行编码。

统一字符编码的编码空间由五部分组成，如图2-8所示。一个统一字符编码的完整体系还允许对各个单一字符的编码进行重新组合，以构成一些复合字符。例如，由字符“´”和“A”组合成字符“Á”。这些组合字符所使用的算法，以及统一字符编码的扩展，可以查阅本章结尾处的参考文献。

虽然统一字符编码目前已成为美式计算机系统唯一使用的字符编码，但是大多数计算机制造商都在他们的系统中至少包含对统一字符编码的某种有限程度的支持。统一字符编码目前是Java编程语言的默认字符。最终，统一字符编码能否会被所有计算机制造商所接受，还要取决于他们定位其产品市场国际化的需求程度，以及生产出某种具有支持存储ASCII码和EBCDIC编码字符双重需求的磁盘驱动器的成本价格。

2.7 用于数据记录和传递的编码方式

ASCII码、EBCDIC码和统一字符编码都是在计算机存储器中使用的各种明确的表示方法（第3章将介绍怎样使用二进制数字设备来实现这种表示）。数字开关，例如在存储器中使用的开关，除了“开”和“关”两种状态外，不存在任何中间状态。然而，在把数据写入到某种类型的记录介质（例如，磁带或者磁盘），或者将数据进行长距离传送时，二进制数信号可能会变得模糊不清，特别是当信号中有一长串1和0时。造成这种模糊不清的原因，其中一部分要归咎于信号发送和信号接收之间定时系统出现的偏差。而磁记录介质，如磁带和磁盘等，同样会因为其制成的磁材料本身的电学行为而导致记录信号的不同步。数字信号的“高”和“低”两种状态之间的信号转换过程，对在数据记录和通信设备中保持数据同步非常有帮助。因此，ASCII码、EBCDIC码和统一字符编码通常都会在被传输或记录前，被转换成其他代码。这种转换一般由数据记录和发送设备中的电子控制部件自动完成。用户和计算机主机都不会意识到这种转变的发生。

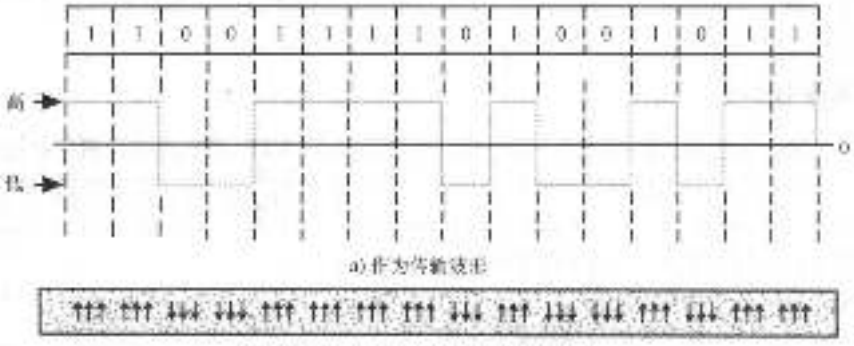
通信设备利用传输介质（例如铜导线）中传递的“高”、“低”脉冲信号来发送和接收数据字节。磁存储设备利用磁极性的变化，称为*磁通量反转*（flux reversals），来记录数据。某种类型的编码方式可能更适合于数据通信，而不是数据记录。为了适应记录方式的变革，以及不断改进的传输和记录介质，不断有新的编码方式发明出来。本书仅限于研究几个比较流行的记录和传输编码方式，并且讨论怎样克服此领域所遇到的一些挑战。为了简单起见，下面将采用*数据编码*（data encoding）这个术语来表示把一个简单字符代码，如ASCII码，转换为便于存储和传输的其他代码的过程。而*编码数据*（encoded data）则表示已经被编码的字符的代码。

2.7.1 不归零编码

最简单的数据编码方式是*不归零编码*（non-return-to-zero，NRZ）方式。使用这种编码方式意味着用“高”和“低”来表示1和0：1通常为高电压，0位低电压。通常高电压为正3伏或者正5伏，而低电压为负3伏或者负5伏（如果反过来，在逻辑上也是等效的）。

例如，英语单词*OK*的具有偶数奇偶校验的ASCII编码为：11001111 01001011。图2-9按照信号形式和磁通量形式给出了这个单词的NRZ编码模式。其中每一位在传输介质中占据一个小的时间段，或者在磁盘中占据一个小的空间。这些小的时间段和空间被称为*位元*（bit cell）。

从图中可以看出，在ASCII码O中有一长串1。如果发送单词*OK*的较长的形式*OKAY*，就会出现一个长串的0和一个长串的1：11001111 01001011 01000001 01011001。除非接收器与发送器可以精确同步，否则对于每个位元来说，不可能都知道其信号持续的准确时间。如果接收器同步较慢，或者是定时系统发生错位，那么就有可能接收到的*OKAY*编码的位序列为10011 0100101 010001 0101001，翻译成ASCII码，就是<*EXT*>( )，完全不是原来所发送的单词的意思（<*EXT*>用在这里仅仅表示一个单一的ASCII字符End-of-Text，用十进制数表示是26）。



a) 作为传输波形

b) 磁通量模式（箭头的方向指示磁极性）

图 2-9 单词 OK 的 NRZ 编码

这个例子说明了在 NRZ 编码中即使只丢失了一位编码，整个信息也可能变得不知所云。

2.7.2 反转不归零编码

反转不归零编码（nonreturn-to-zero-invert，NRZI）方法强调的是同步损失方面的问题。NRZI 采用信号的转变，或者从高到低，或者从低到高的变化，来表示一个二进制数的 1，没有变化则表示 0。英语单词 OK 的具有偶数奇偶校验的 NRZI 编码如图 2-10 所示。



图 2-10 单词 OK 的 NRZI 编码

尽管 NRZI 编码方法消除了二进制数 1 的丢失问题，但是仍然面临着一长串 0 的问题。这有可能造成接收器或阅读器发生时间上的异相漂移，从而导致数据位的丢失。

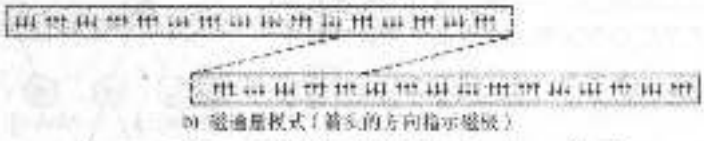
解决这个问题最明显的方法就是在发送的波形中注入足够多的信号转变，保证发送器和接收器的同步，从而完整地保留所发送的信息内容。

2.7.3 相位调制编码

通常称为相位调制（phase modulation，PM，或称为 Manchester coding）的编码方法就是为了处理同步问题而设计的。PM 对编码的每一位都提供一个信号转变，二进制数 1 由上升转变的信号给出，0 则伴随一个下降转变的信号。如果需要，会在每个位单元的边界提供一个额外的信号转变。单词 OK 的 PM 编码如图 2-11 所示。



a) 作为传输波形



b) 磁通量模式（箭头的方向指示磁极性）

图 2-11 单词 OK 的相位调制编码（Manchester 编码）

相位调制编码通常应用于数据传送，比如局域网，但还不足以应用在数据存储中。如果将 PM 用于磁带和磁盘系统，相位调制编码需要两倍于 NRZ 编码的位数（即每个位单元就要有一个磁通量的转变，参见图 2-11b）。大家知道，使用 NRZ 编码可能导致难以令人接受的高出错几率。因此，好的编

码方案应该是一种可以比较经济地在获取尽可能大的存储空间和产生尽可能少的错误几率之间取得平衡的方法。人们在寻求这种平衡的过程中，发展了许多编码方法。

2.7.4 频率调制编码

作为数字系统中的应用，频率调制（frequency modulation，FM）编码类似于相位调制编码，在每个位单元中至少提供一个信号跃变。这些同步的跃变发生在每个位单元的开始处。要对二进制数1进行编码，还需要在位单元的中间再提供一个额外信号跃变。例如，单词OK的FM编码如图2-12所示。



图2-12 单词OK的频率调制编码

从图中不难看出，相对于数据存储的要求来说，FM只是比PM好一点。人们还从FM方法中延伸出了一种被称为改进的频率调制（modified frequency modulation，MFM）的编码方法。这种方法只对具有连续的0的编码才在位单元的边界提供信号跃变。使用MFM编码方法，只需对每两个位单元至少提供一个信号跃变，而在PM或FM中对每一位位单元都必须至少提供一个信号跃变。



图2-13 单词OK的改进的频率调制编码

由于MFM编码需要的信号跃变数目比PM编码少，而又比NRZ多，因此，从错误控制和经济的角度来看，MFM编码系统是一种高效率的编码方法。多年来，MFM编码实际上是硬盘存储中所使用的唯一编码方式。单词OK的MFM编码如图2-13所示。

2.7.5 运行长度限制编码

运行（作业）长度限制编码（Run-Length-Limited，RLL）是一种对由字符编码组成的字进行分块的编码方式，例如，把ASCII码或EBCDIC码翻译成一组特殊设计的编码字，限制连续0出现的数目。如，*RLL*（*d*，*k*）编码允许在任何一对相邻的1之间，出现最少*d*个和最多*k*个连续的0。

很显然，RLL的编码字必须包含比原来的字符编码更多的位数。但是，由于RLL在磁盘上面是采用NRZI方式进行编码，所以RLL编码的数据实际上在磁介质上面会占有更少的空间，因为这种编码所涉及到的磁通量的转变要少得多。RLL使用的编码字用来防止磁盘出现同步损失，这种情况在使用一个"平"（flat）的二进制NRZI编码的时候是会遇到的。

尽管存在许多变化形式，但RLL（2，7）仍然是磁盘系统中使用的主流编码方式。从技术上，它是一个8位的ASCII或EBCDIC字符的16位映射。但是，按照磁通量反转的说法，它比MFM编码方式要高出50%的存储效率（关于这一点的证明留给读者作为一个练习）。

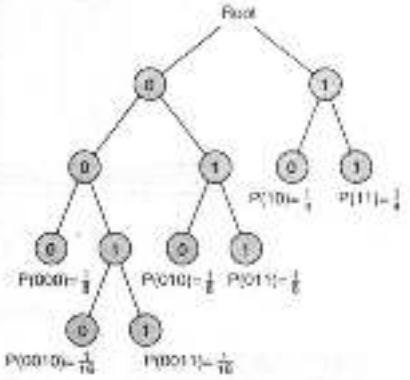


图2-14 RLL（2，7）编码的几率树

从理论上来说，RLL是一种被称为哈夫曼编码（Huffman coding）的数据压缩形式（将在第7章中进行讨论）。这里，核心思想是要使用最短的编码字的位组合方式来对尽可能多的信息位的组合方式进行编码。这也就是磁盘存储中所说的使用最小数目的磁通量反转。这种理论是基于这样一个基本假定：即在任意的位单元中，数字1或者没有数字1出现的几率相等。从这个假定出发，可以推导出在任一对相邻的位单元之间出现10组合的几率为0.25。（$P(b_i=1)=1/2$；$P(b_j=0)=1/2$；$\Rightarrow P(bb_j=10)=1/2\times1/2=1/4$。）图2-14给出了RLL（2，7）使用各种位的组合方式的几率树

分布。图 2-15 给出了 RLL (2, 7) 使用的位组合方式。

从表中不难看出，对于任意可能的位的组合方式，不可能有多于 7 个连续的 0 出现，但至少会有 2 个连续的 0 出现。

图 2-16 比较了分别采用 MFM 和 RLL (2, 7) NRZI 对单词 OK 的编码结果。MFM 有 12 个磁通量的转变发生，而 RLL 只有 8 个。如果磁盘设计中的主要限制因素是每平方毫米磁通量的转变数目的话，那么在相同的磁介质面积上，使用 RLL 编码方式比使用 MFM 编码方式多装入 50% 个单词 OK 的个数。因为这种原因，RLL 编码方式在制造高密度磁盘驱动器方面几乎是独一无二的。

数字符模式	RLL (2, 7) 编码
10	0100
11	1000
000	000100
010	100100
011	001000
0010	00100100
0011	00001000

图 2-15 RLL (2, 7) 编码

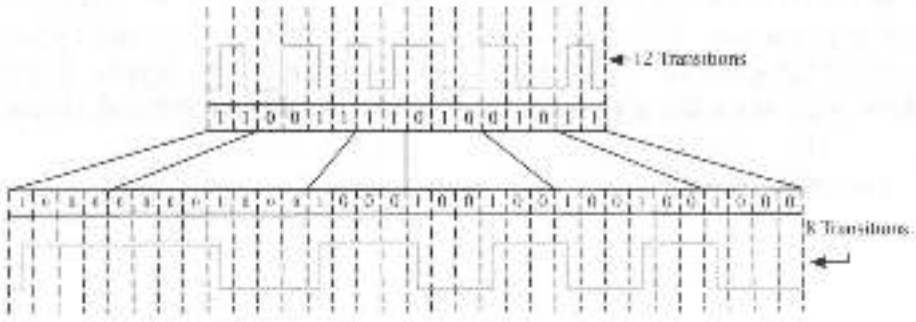


图 2-16 单词 OK 的 MFM 编码（图的上半部）和 RLL (2, 7) 编码（图的下半部）

2.8 错误检测与校正

不管所使用的编码方法如何，没有任何一种通信方式或者存储介质是完全没有错误的。不出现错误的这种情况，从物理上来说是不可能的。随着通信速率的加快，数据位与位之间的时间间隔变得越来越短。另外，随着每平方毫米的存储介质存放的数据位数越多，磁道密度也会随之增加。这样，出错的几率直接与每秒钟所发送的数据位的数目成正比，或者与每平方毫米磁介质存储的位数成正比。

在 2.6.3 节中，我们曾经提到通过将一个奇偶校验位加到 ASCII 码的字节上，以确定在数据传送过程中是否有数据位的损失。这种编码错误检测方法的有效性是很有限的：简单的奇偶校验只能检测每个字节中所产生的奇数个的出错数目。如果出现两位错误，这种方法就无能为力了。

在 2.7.1 节中，读者已经看到了传送的单词 OKAY 的 4 字节序列编码，有可能会被接收为 3 字节序列的<ETX>()。这里值得注意的是，所接收的第 2 个字节的序列编码中，奇偶校验位是正确的，接收器会将其视为好的数据通过。这实际上也是没有意义的。如果在传送财经信息或者处理序代码时，发生这一类的错误将会导致灾难性的后果。

在阅读后面的章节时，读者需要记住，既不可能生产完全没有错误的介质，也不可能百分之百地检测出，或者校正可能出现在介质上的所有错误。错误检测与校正仍然是设计现代计算机系统时需要综合考虑的一个研究课题。因此，精心设计的错误控制系统应该可以检测出合乎预期的合理的错误数目，或者在合理的经济成本范围内进行错误校正（这里的“合理（reasonable）”一词与错误检测和校正的实施过程有关）。

2.8.1 循环冗余码校验

求校验和的错误检测广泛应用于各种不同的编码系统中，从条形码到国际标准书号（International Standard Book Number，ISBN）的编码。这种自我检查的编码方式可以很快地检测出前面的数字是否

出现误读错误。循环冗余码校验（cyclic redundancy check，CRC）是一种最初用在数据通信中的求校验和的错误检测方法，这种方法可以决定在一大块或者是一长串信息字中是否出现一个错误。要校验的数据字块的规模越大，要求的校验和就越大，而且还需要对求校验和的方法提供某种适当的保护。求校验和的方法以及CRC方法都是一种系统性的误差检测（systematic error detection）方案，也就是把错误校验位加在原始信息数据位的后面。组成错误校验位的位组合称为校正子（sysdrome）。增加额外的错误校验位并不会改变原始的信息位的内容。

循环冗余码校验中的循环（cyclic）一词涉及到错误控制系统中有关的抽象数学理论。虽然有关这种理论的详细讨论已经超出了本书的范围，但是这里还是会给读者展示一下这种理论是如何发生作用的，这样有助于大家理解这种理论在经济有效地检验传输错误方面的能力。

模2算术

读者也许熟悉整数算术中有关模数的概念。例如采用12小时的时钟，就是一个我们每天都用来指示时间的模数为12的系统。如果在11：00上加2小时，就得到了时间1：00。模2（模数为2）算术采用的是两个没有借位或进位的二进制操作数，其操作结果同样也是一个二进制数，并且也是模2算术体系中的一个数。由于该体系中数的加法形成闭包（自封闭），而且具有单位元素，数学家称这种模2体系构成一个代数域（algebraic field）。

模2的加法运算的规则如下：

$$0+0=0$$
$$0+1=1$$
$$1+0=1$$
$$1+1=0$$

例2-27 求二进制数1011_2和110_2的模2的和。

$$\begin{array}{r} 1011 \\ +110 \\ \hline 1101_2\ (\text{模 }2) \end{array}$$

这种求和只有在模2体系中才有意义。 ■

模2除法是利用模2加法的规则，通过进行一系列的部分求和运算完成。例2-28说明了模2除法的计算过程。

例2-28 求1001011_2除以1011_2的商和余数。

$$\begin{array}{rl} 1011\overline{)1001011} & \text{1. 把除数直接写在被除数的第一位的下面。} \\ \quad\underline{1011} & \\ \quad 0010 & \text{2. 对这些数字的执行模2加法。} \\ & \text{3. 从被除数上取后面的各个位，使得余下部分的第一个1可以与除数中的第一个1对齐。} \\ \quad 001001 & \text{4. 根据步骤1重新写下除数。} \\ \quad\ \underline{1011} & \text{5. 按步骤2进行相加。} \\ \quad\ 0010 & \text{6. 再取下被除数的另一位。} \\ \quad\ 00101 & \text{7. 这时，}101_2\text{已经不够被}1011_2\text{来除，所以就是余数。} \end{array}$$

得到的商为1010_2。 ■

模2域的算术运算也具有多项式的形式，就如同整数域中有类似的算术多项式一样。如前所述，位置计数系统采用的是一种递增基数指数幂的形式来表示数字。例如，

$$1011_2=1\times2^3+0\times2^2+1\times2^1+1\times2^0$$

取$X=2$，则二进制数1011_2可以简写成如下的多项式形式：

$$1\times X^3+0\times X^2+1\times X^1+1\times X^0$$

而对于例 2-28 中的除法运算就变成了如下的多项式运算：

$$\frac{X^6+X^3+X+1}{X^3+X+1}$$

CRC 的计算和使用方法

经过了上面的背景知识介绍，现在来讨论怎样构建循环冗余码校验体系（CRC）。下面通过举例来加以说明。

1. 假设信息字节为 $I = 1001011_2$（可以使用任意大小的字节数来组成信息块）。

2. 发送器和接收器都对某个任意的二进制位组合模式达成协议，例如，$P = 1011_2$。（如果二进制的位组合模式的开始和结束位都是 1，则运行效果最好）。

3. 将 I 左移，左移的位数为 P 的位数减 1 位，就产生了一个新的 $I = 1001011000_2$。

4. 用 I 做被除数，P 做除数，进行模 2 除法（如同例 2-28 中所做的一样）。略去商只留下余数为 100_2。实际上，这个余数就是 CRC 体系要求的校验和。

5. 把余数加到 I 中，就组成要发送的信息 M：

$1001011000_2 + 100_2 = 1001011100_2$

6. 信息接收器将采用相反的过程来对 M 进行解码和校验。M 应该可以被 P 严格整除：

```
       1010100
1011)1001011100
     1011
     001001
       1011
       0010
       001011
         1011
         0000
```

当结果的余数不为 0 时，表示在 M 的传送过程中有一个错误发生。当采用大指数幂的互质多项式时，这种检测方法的效果最好。下面是广泛应用于这种检测方法中 4 个标准多项式。

- CRC-CCITT（ITU-T）：$X^{16}+X^{12}+X^5+1$
- CRC-12：$X^{12}+X^{11}+X^3+X^2+X+1$
- CRC-16（ANSI）：$X^{16}+X^{15}+X^2+1$
- CRC-32：$X^{32}+X^{26}+X^{23}+X^{22}+X^{16}+X^{12}+X^{11}+X^{10}+X^8+X^7+X^5+X^4+X+1$

CRC-CCITT、CRC-12 和 CRC-16 都是对成对字节进行操作的，而 CRC-32 则用于 4 字节方式，比较适合于 32 位字工作的系统。事实证明，采用这些多项式的 CRC 方法可以检测出超过 99.8%的全部的单位的错误。

CRC 可以采用查找错误表的方法来进行操作，而不必对字节的每一位都计算余数。将各种可能输入的位组合模式产生的余数表直接“烧录”到通信和存储电子设备中。这样一来，只需要一次查找就可以得到余数，而不必进行 16 个或 32 个循环周期的除法运算。很显然，这种方法的使用取决于对检测速度需求和采用更复杂控制电路所带来的成本问题的平衡。

2.8.2 海明编码

数据通信电路比磁盘系统更容易发生错误，同时容错能力也更强一些。在数据通信中，通常具有探测错误的能力就已经足够了。如果通信设备发现一条信息包含一个错误位，只需重新进行传送。而对于存储系统和存储器却没有这种方便。一张磁盘有时存储的是一些财务交易的数据，或者其他一些非再生的实时数据的集合。因此，存储设备和存储器不但要探测，而且还要校正合理数量的错误。

错误恢复编码技术在上个世纪得到了广泛的研究。其中一种最有效的编码方式，也是最老的，是海明编码（Hamming code）。海明编码采用的是奇偶校验的概念。在这里，错误检测和校正能力随着

信息字中所加入的奇偶校验位的数目成正比的增加。海明编码常常使用在随机错误最可能发生的情形。对于随机错误，可以假定每一位出错的几率都是固定的，与其他位的出错没有关联。对计算机存储器而言，这是一种常见的错误。因此，在下面的讨论中将按照存储器的位错探测和校正的方式介绍海明编码。

海明编码所使用的奇偶校验位，也称为校验位（check bit），或者是冗余位（redundant bit）。假设存储的信息字本身由 m 位组成，再加上 r 位冗余位来进行错误检验和校正。最后形成的字称为编码字，它是一个由 m 位数据和 r 位检验位组成的一个 n 位单元。对每个数据字都有一个唯一的、按如下方式构成的 $n=m+r$ 位的编码字：

m 位	r 位

两个编码字之间不同的位的位置数目称为两个编码字的海明距离（Hamming distance）。例如，右如下的两个编码字：

```
1 0 0 0 1 0 0 1
1 0 1 1 1 0 0 1
      *   *
```

其中，它们有3个位置上的位是不同的，所以这两个编码字的海明距离就是3。到目前为止，还没有讨论怎样来产生编码字，下面很快就会介绍。

两个编码字之间的海明距离对于错误检验是非常重要的。如果两个编码字之间的海明距离相隔 d 位，那么把一个编码字转换成另外一个编码字需要有 d 个单一位的错误产生。这也就意味着这种类型的错误将不会被检测出来，因为两者都是合法的编码字。如果要使生成某个编码可以确保能够探测出所有的单位错（只有一位出错），那么所有的编码字对之间的海明距离都应该大于2。如果某个 n 位的编码字没有被识别成一个合法的编码字，就会被认为是一个错误。

对于任意一种计算检验位的算法，都可以构建出其全部合法编码字的完整列表。一种编码方法中各对编码字之间的所存在最小的海明距离称为该编码的最小海明距离（minimum Hamming distance）。编码的最小海明距离常常用符号 $D(min)$ 表示。最小海明距离决定了编码校检错误的能力。简单地说，任意的编码字 X 如果要被当作为另外一个合法的编码字 Y 来接收的话，那么在 X 中应该至少发生了 $D(min)$ 个错误。因此，要检验 k 个（或少于 k 个）的单位错误，编码就必须具有 $D(min)=k+1$ 的海明距离。海明编码通常可以检测出 $D(min)-1$ 个错误，并能够校正 $[(D(min)-1)/2]$ 个错误[⊖]。因此，要能够校正 k 个错误，编码方法的海明距离必须大于 $2k+1$。

编码字是利用信息字加上 r 个奇偶校验位构建而成。在继续讨论错误校验前，先来看下面的一个简单例子。最常用的错误校验方法是在数据位上加一个单一的奇偶校验位（读者可以回忆一下ASCII字符表示方法）。这种编码字中的任何一位出现一个单位的错误都会产生错误的奇偶校验。

例2-29 假定某个存储器的编码方式是采用2个数据位和1个奇偶校验位（附在编码字末尾）。而且使用的是偶校验方法，即编码字中1的个数必须是偶数。2个数据位可以有4种不同字的组合。下面列出了这些数据字和相应的奇偶校验位，以及它们组成的4种可能的编码字。

数据字	奇偶校验位	编码字
00	0	000
01	1	011
10	1	101
11	0	110

编码字由3位二进制数组成。很显然，3位二进制数可以有8种不同的组合如下（其中标注“√”编码字为合法的编码字）：

⊖ 括号[]表示取整函数，也就是求小于括号内数字的最大整数，例如，[3.3]=3，[3.9]=3。

000*	100
001	101*
010	110*
011*	111

如果遇到编码字 001，则会认为是非法的。这表示该编码字的某个地方发生了错误。例如，假设存放在存储器中的正确的编码字为 011，但是由于产生了一个错误，结果变成了 001。这个错误可以被探测出来，却不能够被校正。这里还不可能准确地决定编码字中究竟有多少位被翻位和具体是哪一位出错。正如下面所讨论的，能够进行错误校正（或称为纠错）的编码方式则要求有多于一位的奇偶校验位。■

在上面的例子中，如果一个合法的编码字出现 2 个错误，又会发生什么情况呢？例如，编码字 011 变成了 000。这种错误是不可能被检测出来的。如果仔细研究上面的编码，就会发现 它的 $D(min)$ 为 2，这就是说这种编码方式只能保证检验单位错误。

编码方法的错误校验能力取决于编码的最小海明距离 $D(min)$，例 2-29 就已经从错误检测的观点展示了这种依赖关系。错误校正（纠错）需要在编码中包含有一些额外的冗余位。如果要对 k 个错误进行检测和校正，所要求的最小海明距离为 $D(min)=2k+1$。最小海明距离可以保证即使在发生 k 个变化的情形下，所有的合法编码字之间都有足够距离来加以区别，而原来的非法编码字只会和唯一的一个合法编码字比较接近（相似）。这一点非常重要，因为错误校正方法就是把非法的编码字变换成位数相差最少的合法的编码字。例 2-30 阐明的就是这种思想。

例 2-30 假定有下列的 4 个编码。（暂时先不考虑这些编码的产生方式，这个问题将很快会进行讨论。）

0 0 0 0 0
0 1 0 1 1
1 0 1 1 0
1 1 1 0 1

首先确定 $D(min)$。仔细观察所有可能的编码字对，不难发现最小海明距离为：$D(min)=3$。因此，这种编码方式可以检测到最多 2 位错误，而只能校正 1 位错误。但怎样进行错误校正呢？假设读到的错误编码字为 10000。很明显，至少存在 1 位错误，因为它不属于合法编码字中的任何一个。现在来计算读到的非法编码字与合法编码字之间的海明距离：它与第 1 个合法编码字相差 1 位，与第 2 个相差 4 位，与第 3 个相差 2 位，而与第 4 个相差 3 位。于是，得到了这样一个*差额矢量*（difference vector）：[1，4，2，3]。利用这种编码方法进行错误校正，会自动地把读到的非法编码字校正为最接近的那个合法编码字，校正结果为 00000。注意，这种所谓“校正”并不意味着结果一定正确。这里我们假设了可能出现的错误是最小数目的错误，即 1 个错误。但是，有可能原来的编码字为 10110，是因为发生了两个错误而变成了现在的非法编码 10000。

假设确实发生了两个错误，如读到的非法编码字为 11000。如果计算得到的差额矢量是 [2,3,3,2]并没有最接近的合法编码字，因此不能进行错误校正。海明距离为 3 的编码方式只能进行 1 个错误的校正。并且，如果发生出现多于一个错误的情况时，就不能进行错误校正处理，本例已证实了这一点。■

至此，我们只是罗列了各种各样的编码，并没有讨论到这些编码的生成方法。实际上，有许多种生成编码的方法，其中最直观的就是下面要介绍的用于编码设计的海明算法。在解释该算法的各个实际步骤之前，先来介绍一些背景知识。

假设要设计一种由 m 位数据位和 r 位校验位组成的字的编码方式，并要求这种编码方式可以进行单位错误的校正。显然，这种编码方式有 2^m 个合法的编码字，每个字都具有唯一的包含校验位的位组合形式。现在，我们只关注发生单一位错误的情形，先来考察与合法编码字相差 1 位的一组非法编码字。

每个合法的编码字都有 n 位，其中的每一位都有可能发生错误。这样，每个合法的编码字都有 n 相差 1 位的非法编码字。如果只考虑每个合法的编码字和由 1 个错误组成的非法编码字，那么就存在与每个编码字相联系的 $n+1$ 种位的组合方式（1 个合法字和 n 个非法字）。因为每个编码字由 n 位构成，$n=m+r$，所以共有 2^n 种可能的位的组合方式。因此有下面的不等式：

$$(n+1)\times 2^m \leqslant 2^n$$

其中，$n+1$ 是每个编码字对应的位的组合方式，2^m 是合法的编码字的数目，2^n 是各种可能的位的组合方式的总数。因为 $n=m+r$，所以不等式可以改写为：

$$(m+r+1)\times 2^m \leqslant 2^{m+r}$$

或

$$(m+r+1) \leqslant 2^r$$

这个不等式很重要，它指定了要构成一个有 m 位数据位和 r 位校验位，并且可以校正全部单一位错误的编码所要求的校验位数目的下限。

假定数据位的长度 $m=4$，有：

$$(4+r+1) \leqslant 2^r$$

就是说，r 必须大于或者等于 3。如果选择 $r=3$，这意味着要构建一个有 4 位数据位字，可以校正单位错误的编码系统，必须增加 3 位校验位。

海明算法为设计具有校正单位错误能力的编码系统提供了一种直接方法。按照以下步骤，可以构建任意大小的存储器字的错误校正编码系统。

1. 先确定编码所需的校验位数目 r，然后算出编码字的位长度 n（$n=m+r$），从右往左排列，从 1 开始（不是 0）。

2. 设置位数满足 2 的指数幂的位为奇偶校验位，其他位为数据位。

3. 对于各个编码位置，可以按照如下的方式指定不同的奇偶校验位来检验：第 b 位编码由满足关系 $b_1+b_2+\cdots+b_j=b$（这里“+”表示模 2 的和）的奇偶校验位 b_1，b_2，…，b_j 进行校验。

下面通过举例来说明这些步骤和错误校正的实际过程。

例 2-31 利用上面介绍的海明编码和偶校验，对 8 位 ASCII 字符 K 进行编码（最高位应该为 0）。同时，人为地引入一个单位的错误，并指出怎样找出这个错误。

首先确定 K 的编码字

步骤 1：确定所需校验位的数目，然后将这些校验位加到数据位中，编码字的全部位数总共有 n 位。

因为 $m=8$，有 $(8+r+1)\leqslant 2^r$，这样，r 必须大于或等于 4。这里选择 $r=4$。

步骤 2：从 1 开始，按照从右往左的顺序对 n 位编码进行编号，结果如下：

$$\overline{12}\ \overline{11}\ \overline{10}\ \overline{9}\ \square\!\!\!\!\overline{8}\ \overline{7}\ \overline{6}\ \overline{5}\ \square\!\!\!\!\overline{4}\ \overline{3}\ \square\!\!\!\!\overline{2}\ \square\!\!\!\!\overline{1}$$

奇偶校验位用方块来进行标记。

步骤 3：分配指定校验各个不同位置上的位的奇偶校验位。

因此，首先将所有各个位的位置按照 2 的幂指数写成求和形式：

1=1	5=1+4	9=1+8
2=2	6=2+4	10=2+8
3=1+2	7=1+2+4	11=1+2+8
4=4	8=8	12=4+8

因为第 1、3、5、7、9、11 位的求和表达式中有数字 1，所以第 1 位（最低位）的奇偶校验位将反映这些位置上各个位的奇偶校验特性。类似地，2 对 2、3、6、7、10、11 起作用，所以第 2 位的奇偶校验位将反映这组位置上的位的奇偶校验特性。第 4 位为第 4、5、6、7、12 位提供奇偶校验，而第 8 位为第 8、9、10、11、12 位提供奇偶校验。在没有标注方块的空白位置上依次写下数据位，然后加上奇偶

校验位，就得到了如下的编码字：

$$\underset{12}{0}\ \underset{11}{1}\ \underset{10}{0}\ \underset{9}{0}\ \underset{8}{\boxed{1}}\ \underset{7}{1}\ \underset{6}{0}\ \underset{5}{1}\ \underset{4}{\boxed{0}}\ \underset{3}{1}\ \underset{2}{\boxed{1}}\ \underset{1}{\boxed{0}}$$

因此，K 的编码字为 010011010110。

现在，在 b_9 位引入一个错误，就将编码字变成了 010111010110。然后利用奇偶校验位来校验各不同的位组合，结果如下：

第 1 位的奇偶校验位对第 1、3、5、7、9、11 位进行校验，因为采用的是偶校验，所以产生 1 个错误。

第 2 位的奇偶校验位校验第 2、3、6、7、10、11 位，没有出错。

第 4 位的奇偶校验位校验第 4、5、6、7、12 位，没有出错。

第 8 位的奇偶校验位校验第 8、9、10、11、12 位，产生 1 个错误。

第 1 位和第 8 位奇偶校验位的校验出错，它们共同校验的位为第 9 位和第 11 位，所以单位的错误一定出现在第 9 位或第 10 位。但是，由于第 2 位奇偶校验位在其校验的位置组合中包含了第 11 位，并指出第 11 位没有出错，所以错误一定发生在第 9 位，这也正是我们人为设置的错误位。即使没有任何发生错误的位置的线索，通过上面介绍的方法也可以准确界定错误发生的位置。要校正这个错误，只要简单地对这一位（第 9 位）执行取反操作。

因为这里的奇偶校验采用的是组合定位的方式，所以找出错误位的一个更简单的方法是将校验出错的奇偶校验位的位置进行相加，可以直接得出错误位的位置。例如，本例中第 1 位和第 8 位的奇偶校验都产生一个错误，1 + 8 = 9，第 9 位也正是编码出错的位置。 ■

在第 3 章中，读者将会看到只需利用一个简单的二进制电路，就可以非常方便地实现海明编码技术。因为这种方法简单可行，所以可以很经济方便地在系统中添加海明编码来实现数据保护，而且对系统的性能所造成的影响最小。

2.8.3 里德-所罗门编码

海明编码系统在出错率非常低的合理情况下非常有效。对于固定磁盘驱动器来说，出错率在一亿分之一的数量级。因此，上面所介绍的 3 位海明编码系统可以非常方便地校正这种类型的错误。但是，海明编码在遇到多个相邻数据位发生损坏的情况时，就会变得毫无用处。这种类型的错误称为区块错误（burst error，或称为突发错误）。因为操作处理不当和外界环境影响所引起的应力等主要因素，区块错误在移动性记录介质方面，比如磁带和压缩光盘（CD），是很普遍的。

如果出错的情况是成块发生，就应该使用能够处理块状错误的纠错编码系统，而不是海明编码。海明编码只能进行一些单位级别的出错处理。里德-所罗门（Reed-Soloman，简称 RS）编码属于 CRC 类型的编码，主要处理的是整个字符块，而不仅仅是几个位。RS 编码和 CRC 一样，是一种系统编码方式：即在信息字节块上加有奇偶校验位。RS（n，k）编码中的各参数定义为：

- s = 一个字符（或者“符号”）所占位的数目。
- k = 构成数据块的 s 位字符的数目。
- n = 编码字的位的数目。

RS（n，k）可以在 k 个信息字节组成的编码字中校正 $\frac{(n-k)}{2}$ 个错误。

因此，现在流行的 RS（255，233）编码方式是采用 223 个 8 位的信息字节，加上 32 个错误校正字节，来构成 255 字节的编码字。可以校正信息块中多达 16 个的错误字节。

里德-所罗门编码的编码发生器多项式由一个称为伽罗瓦域（Galois field）的抽象数学结构所定义的多项式给出（在这里，对伽罗瓦数学的细节讨论会有些离题太远，读者可以参考本章结尾部分的参考文献）里德-所罗门通用多项式为：

$$g(x)=(x-a^i)(x-a^{i+1})\cdots(x-a^{i+2t})$$

式中 $t = n-k$，x 是一个完整的字节（或“符号”），$g(x)$ 是对域 $GF(2^s)$ 执行操作（注意：这个多项式是在伽罗瓦域进行展开的，与常规代数中使用的整数域有很大区别）。

n 字节的 RS 编码字要使用下面的方程式进行计算：

$$c(x)=g(x)\times i(x)$$

其中，$i(x)$ 为信息块。

尽管这些代数看起来非常复杂而且令人生畏，但是里德-所罗门纠错算法可以非常有效地在计算机硬件中实现。它们已经成功使用在计算机主机的高性能磁盘驱动器，以及用于音乐和数据存储的压缩光碟之中。这些里德-所罗门算法的实现方法将在第 7 章介绍。

本章小结

本章主要介绍了数字计算机中采用的数据表示方法和数值运算的基本知识。要求掌握数字在不同的数基之间相互转换的技巧和方法，并且要熟记那些较小的十六进制数字和二进制数字。这些基本知识对后续章节的学习是非常有用的。有关十六进制编码的内容，对于需要在一个系统崩溃后读取和修复硬盘残余的核心数据，或者是希望在数据通信方面从事一些深入工作，都是非常有价值的。

对于浮点数字的运算，计算中的重复迭代过程可以使一个微小的误差演变成一个重大的错误。有许多不同的数学技巧可以用来对浮点运算的误差进行控制，详细讨论这些技术已经超出本书的范围。

目前的大多数计算机使用 ASCII 和 EBCDIC 代码来表示各种字符。通常没有必要记住所有字符的编码。但是，如果在工作中需要频繁使用这些字符编码，不妨掌握若干“关键字符的编码值”，从这些数值出发就可以算出大部分所需的的其他字符编码值。

统一编码字符是 Java 编程语言和 Windows 新版本操作系统的默认字符。它有可能取代 EBCDIC 和 ASCII，成为计算机系统中字符表示的基本方法。然而，原来老的编码方法，由于经济适用和深入普及，还将会继续使用较长的一段时间。

学习数据字节在磁盘和磁带上的存储方法有助于理解有关数据存储方面的许多问题。熟悉和理解各种错误控制方法对于学习数据存储和数据通信都很有益处。更多的有关数据存储方面的内容将在第 7 章介绍，而有关数据通信的最新课题也会在第 11 章中加以阐述。

错误检测和校正的编码技术实际上已经应用于计算技术的各个领域。了解错误控制的方法有助于我们在各种可利用错误校验方案中做出明智的决策。选择正确的方法需要考虑算法成本和所使用的存储和传输介质的容量等因素。

深入阅读

从有关西方文明中早期数学发展简史中；可阅读 Bunt（1988）的著作。

Knuth（1998）在其有关计算机算法系列丛书的第 2 卷中详尽有趣地讨论了数字体系和计算机算术的历史发展过程（建议每位计算机科学家们人手一套 Knuth 的著作）。

有关浮点算术的权威定义可以参阅 Goldberg（1991）的文章。Schwartz 等人（1999）介绍了 IBM System/390 计算机怎样使用原来的浮点表示形式和 IEEE 的新标准执行浮点操作。而 Soderquist 和 Leeser（1996）对有关浮点除法和求平方根的问题做了详尽而精妙的讨论。

有关统一编码字符的详细资料可以在同一编码字符协会的网站：www.unicode.org，以及其出版物：*Unicode Standard Version* 3.0（2000）中找到。

国际标准化组织的网站地址是：www.iso.ch。该组织的影响范围很大，从该网站可以找到很多有用的信息。同样，美国国家标准委员会的网站：www.ansi.org 也提供类似的有用信息。

与数据存储有关的数据编码的权威资料可以参考许多有关电气工程方面的书籍。这些书籍的内容涉及到各种物理介质的基本特性，以及各种不同的编码方法的优劣比较。其中，Mee 和 Daniel（1988）的著作也许会很有帮助。

在掌握了第 3 章的基本概念后，可以进一步阅读 Arazi 的著作（1988）。这本精心编撰的书籍介绍了如

何使用一些简单的数字电路来实现错误的检测和校正。该书的附录对应用在里德-所罗门编码中有关伽罗瓦域的数学做了详尽精辟的讨论。

如果需要认真细致地研究错误校正理论，学习 Pretzel（1992）的著作是一个很好的开始，该书结构合理、内容丰富，而且通俗易懂。

有关伽罗瓦域的详细论述可以参阅 Artin（1998）和 Warner（1990）的著作。Warner 的书籍清晰和全面地介绍了抽象代数的概念。学习抽象代数对研究数学密码学很有帮助，这是计算机科学中一个非常令人感兴趣的领域。

参考文献

Arazi, Benjamin. *A Commonsense Approach to the Theory of Error Correcting Codes.* Cambridge, MA: The MIT Press, 1988.

Artin, Emil. *Galois Theory.* New York: Dover Publications, 1998.

Bunt, Lucas N. H., Jones, Phillip S., & Bedient, Jack D. *The Historical Roots of Elementary Mathematics.* New York: Dover Publications, 1988.

Goldberg, David. "What Every Computer Scientist Should Know About Floating-Point Arithmetic." *ACM Computing Surveys 23:*1 March 1991. pp. 5–47.

Knuth, Donald E. *The Art of Computer Programming*, 3rd ed. Reading, MA: Addison-Wesley, 1998.

Mee, C. Denis, & Daniel, Eric D. *Magnetic Recording, Volume II: Computer Data Storage.* New York: McGraw-Hill, 1988.

Pretzel, Oliver. *Error-Correcting Codes and Finite Fields.* New York: Oxford University Press, 1992.

Schwartz, Eric M., Smith, Ronald M., & Krygowski, Christopher A. "The S/390 G5 Floating-Point Unit Supporting Hex and Binary Architectures." *IEEE Proceedings from the 14th Symposium on Computer Arithmetic.* 1999. pp. 258–265.

Soderquist, Peter, & Leeser, Miriam. "Area and Performance Tradeoffs in Floating-Point Divide and Square-Root Implementations." *ACM Computing Surveys 28:*3. September 1996. pp. 518–564.

The Unicode Consortium. *The Unicode Standard, Version 3.0.* Reading, MA: Addison-Wesley, 2000.

Warner, Seth. *Modern Algebra.* New York: Dover Publications, 1990.

基本概念和术语复习

1. 单词 *bit* 是哪两个词的缩写？
2. 解释名词位、字节、半字节和字，以及它们之间的相互关系。
3. 为什么二进制和十进制数制称为位置计数体系？
4. 什么是基数？
5. 对于图 2-1 中“需要记忆的数字”（包括其中的各种基数），想一想能够记住多少个数字？
6. 对于无符号数字，溢出是什么意思？
7. 列举数字计算机中带符号整数的三种表示方法的名称，并且解释这三种表示方法之间的区别。
8. 数字计算机系统最常采用的是三种整数表示方法中的哪一种？
9. 对于像自行车里程表这类的补码系统是如何构建的？
10. 使用本章中介绍的倍乘转换法是否要比其他传统的二-十进制的转换方法更容易一些？并解释原因。
11. 参照上一个问题，说明其他两种传统的二-十进制转换方法有什么缺点。
12. 什么是溢出？以及如何检测溢出？在带符号数中的溢出和在无符号数中的溢出有什么区别？
13. 如果计算机只能处理和存储整数，那么计算机系统在处理数字时会遇到哪些困难？如何克服这些困难？
14. 浮点数由哪三部分组成？
15. 什么是偏移指数？使用偏移指数有什么好处？
16. 什么是规格化？为什么要使用规格化？

17. 为什么使用二进制计算机进行浮点运算时中总是存在不同程度的误差？
18. 在 IEEE－754 浮点标准中，双精度数的长度是多少位？
19. 什么是 EBCDIC？它与 BCD 有何关系？
20. 什么是 ASCII？ASCII 是怎样产生的？
21. 一个统一编码字符需要用多少二进制位来表示？
22. 为什么会产生统一编码字符？
23. 在对磁盘进行写数据时，为什么要避免使用不归零的编码方式？
24. 在对磁盘进行写数据时，为什么曼彻斯特编码不是一个好的选择？
25. 试解释运行长度限制编码方式的工作原理。
26. 试解释循环冗余校验方法的工作原理是什么？
27. 什么是系统性的误差检测？
28. 什么是海明编码？
29. 什么是海明距离？为什么海明距离很重要？最小海明距离是什么？
30. 在错误校验编码中，冗余位的位数和数据位的位数之间有什么关系？
31. 什么是区块（集束）错误？
32. 说出一种可以修补区块错误的错误校验方法的名称？

练习题

◆1．利用减法或者除法-余数方法进行下列各个不同数基之间的数字转换：

◆ a) $458_{10}=$ _____ 3

◆ b) $677_{10}=$ _____ 5

◆ c) $1518_{10}=$ _____ 7

◆ d) $4401_{10}=$ _____ 9

2. 利用减法或者除法-余数方法进行下列各个不同数基之间的数字转换：

a) $588_{10}=$ _____ 3

b) $2254_{10}=$ _____ 5

c) $652_{10}=$ _____ 7

d) $3104_{10}=$ _____ 9

◆3．将下面的十进制小数转换成二进制数，小数点后最多取 6 位：

◆ a) 26.78125

◆ b) 194.03125

◆ c) 298.796875

◆ d) 16.1240234375

4. 将下面的十进制小数转换成二进制数，小数点后最多取 6 位：

a) 25.84375

b) 57.55

c) 80.90625

d) 84.874023

5．分别使用 8 位的符号幅值、反码和补码方式来表示如下的十进制数：

◆a) 77

◆b) －42

c) 119

d) －107

6. 使用一个 3 位的“字”，列出所有可能的带符号的二进制数，并写出它们用下列方法表示的十进制的等

效数字。

a) 符号幅值

b) 反码

c) 补码

7. 使用一个 4 位的“字”，列出所有可能的带符号的二进制数，并写出它们用下列方法表示的十进制的等效数字。

a) 符号幅值

b) 反码

c) 补码

8. 对于任意给定位数 x，从前面两个问题的答案，总结归纳出采用下列方法可以表示的十进制数值的范围：

a) 符号幅值

b) 反码

c) 补码

9. 假设有一个只有 6 位字长度的小型计算机系统，请问这个计算机采用如下方式可以表示的最小负数和最大正数分别是多少？

a) 反码

b) 补码

10. 假如读者在做环球航海旅行时，闯入了一个未知的文明社会。这里的人民自称为 Zebronians，他们使用 40 个分立的字符（可能是因为斑马身上有 40 根黑白相间的条纹）来进行数学运算。他们非常喜欢使用计算机，但是他们需要可以从事 Zebronian 数学运算的计算机，也是说一种可以表示全部 40 个字符的计算机。作为一位计算机设计人员，你决定最好的方式是使用 BCZ；即二进制编码的 Zebronian 字符（类似于 BCD 编码，只不过它是对 Zebronian 字符编码，而不是对十进制数进行编码）。请问，利用最少的二进制位来表示每个字符，需要多少位数？

11. 完成下列二进制数的乘法运算：

◆a) 1100
×101

b) 10101
×111

c) 11010
×1100

12. 完成下列二进制数的乘法运算：

a) 1011
×101

b) 10011
×1011

c) 11010
×1011

13. 完成下列二进制数的除法运算：

◆a) 101101÷101

b) 10000001÷101

c) 1001010010÷1011

14. 完成下列二进制数的除法运算：

a) 11111101÷1011

b) 110010101÷1001

c) 1001111100÷1100

◆15. 使用倍乘转换法将 10212_3 直接转换成十进制数（提示：要改变乘数）。

16. 使用符号幅值表示法，完成下面的演算：

$+0+(-0)=$

$(-0)+0=$

$0+0=$

$(-0)+(-0)=$

◆17. 假设一台计算机使用 4 位反码表示数字。不考虑溢出现象，请问：在下面的一个假想的编码过程结束后，变量 j 中存储的数值是什么？

```
0 → j    // Store 0 in j.
-3 → k   // Store -3 in k.
while k ≠ 0
  j = j + 1
  k = k - 1
end while
```

18. 假设某个计算机系统中存储的浮点数具有 1 个符号位、3 位指数和 4 位有效位数。

a) 如果存储系统是采用规格化的，试问：该系统可以存储的最大正数和最小正数是多少？假定没有其他隐含的位数，不使用偏移指数，指数采用补码表示法，并且允许全 0 和全 1 的数值。

b) 如果希望所有的指数都为非负数，应该使用的偏移值为多少？为什么选择这个偏移值？

◆19. 使用上一题的模型，包括所选择的偏移值，对下面的 2 个浮点数字进行加法运算，并且要求得到结果采用与下面的加数和被加数相同的表示方法：

0	1	1	1	1	0	0	0
0	1	0	1	1	0	0	1

如果结果与上一题有误差的话，请计算相对误差。

20. 假定采用本书所给的浮点表示法的简单模型（该表示使用 14 位格式，其中 5 位表示指数，指数的偏移值为 16，8 位是规格化的尾数，还有 1 个符号位）：

a) 证明计算机怎样使用这种浮点格式来表示数字 100.0 和 0.25。

b) 证明计算机如何通过改变其中一个浮点数，使得这两个浮点数都按照 2 的相同的幂指数表示，来实现上面这两个浮点数的加法运算。

c) 证明计算机如何使用上面的浮点表示法来表示这两个浮点数的求和结果。计算机实际存储的求和结果的十进制数值是多少？请解释原因？

21. 是什么原因造成除法下溢？使用什么方法可以解决这个问题？

22. 为什么计算机常常采用规格化形式来存储浮点数？试对比在指数中加入符号位的方法，使用偏移值的规格化形式的表示方法有什么优点？

23. 假设 $a=1.0\times2^9$，$b=-1.0\times2^9$，和 $c=1.0\times2^1$。利用书中介绍的浮点模型（该表示使用 14 位格式，其中 5 位表示指数，指数的偏移值为 16，8 位是规格化的尾数，还有 1 个符号位），进行下面的计算。要特别注意运算操作的次序。在这个有限模型中，浮点算术在代数性质上有什么特点？在进行乘法运算时，是否还会像加法运算一样，存在这种代数性质上的反常行为？

$$b+(a+c)=$$

$$(b+a)+c=$$

24. a) 已知 A 的 ASCII 编码为 1000001，请问：J 的 ASCII 编码为多少？

b) 已知 A 的 EBCDIC 编码为 11000001，请问：J 的 EBCDIC 编码为多少？

◆25. 假设计算机使用的是 24 位字。试在如下的情形下，利用 24 位来表示数值 295。

◆a) 如果计算机使用偶校验，如何表示十进制数 295？

◆b) 如果计算机使用 8 位 ASCII 编码和偶校验，如何表示字符串 295？

◆c) 如果计算机使用压缩的 BCD 编码，如何表示数字+295？

26. 假设下面的编码为 7 位 ASCII 字符，并且没有奇偶校验位，试对下面的 ASCII 信息进行解码：
1001010 1001111 1001000 1001110 0100000 1000100 1001111 1000101

◆27. 为什么系统设计人员都愿意采用统一编码字符作为新的计算机系统的默认字符集？如果不采用统一编码字符，你会给出什么理由？

28. 采用下面的编码方式，编写字符 4 的 7 位 ASCII 代码：
a) 非返回 0
b) 非返回 0-反转
c) 曼彻斯特编码
d) 频率调制
e) 改进的频率调制
f) 运行长度限制

29. 为什么 NRZ 编码很少在磁介质的数据记录中使用？

30. 假设要创建一种编码方式，使用 3 个信息位，1 个奇偶校验位（附加在信息位的后面），并采用奇校验。请列出这种编码的全部合法编码字。这种编码方式的海明距离是多少？

31. 纠错海明编码是系统性的编码方式吗？请解释原因？

32. 计算下列编码的海明距离：
0011010010111100
0000011110001111
0010010110101101
0001011010011110

33. 计算下列编码的海明距离：
0000000101111111
0000001010111111
0000010011011111
0000100011101111
0001000011110111
0010000011111011
0100000011111101
1000000011111110

34. 假设要求一个纠错编码可以校正长度为 10 的存储器字的全部单位错误，那么：
a) 需要有多少位奇偶校验位？
b) 如果采用本章介绍的海明算法来设计纠错编码，请求出 10 位信息字：1001100110 的编码字。

◆35. 假设正在使用的一种纠错编码可以校正长度为 7 的存储器字的全部单位错误。计算结果表明，需要 4 位校验位，编码字的全部长度为 11 位。编码字的产生方式是采用书中所介绍的海明算法。现在接收器收到如下代码字：
1 0 1 0 1 0 1 1 1 1 0
假定采用偶校验，请问收到的这个字是否为合法的编码字？如果不是，根据纠错编码，指出错误发生在哪一位？

36. 使用下面的代码字，重复练习 35：
0 1 1 1 1 0 1 0 1 0 1

37. 列举里德-所罗门编码与海明编码两种不同之处的名称。

38. 何时需要选择采用一个 CRC 编码而不是海明编码？何时需要采用海明编码而不是 CRC 编码？

◆39. 对于下列的模 2 除法运算，求出商和余数：
◆a) $1010111_2 \div 1101_2$

◆b) $1011111_2 \div 11101_2$

◆c) $1011001101_2 \div 10101_2$

◆d) $111010111_2 \div 10111_2$

40. 对于下列的模2除法运算，求出商和余数：

a) $1111010_2 \div 1011_2$

b) $1010101_2 \div 1100_2$

c) $1101101011_2 \div 10101_2$

d) $1111101011_2 \div 101101_2$

◆41. 使用CRC多项式1011，计算信息字：1011001的CRC编码字。在接收端进行除法校验。

42. 使用CRC多项式1011，计算信息字：01001101的CRC编码字。在接收端进行除法校验。

43. 任意挑选一种计算机结构（例如，80486、Pentium、Pentium IV、SPARC、Alpha或者是MIPS）。试研究一下所选的结构与本章所引入的各种基本概念有何关联。比如，对负数系统采用什么表示方法？而系统支持什么字符编码？

我永远都热爱布尔代数的世界。
——Claude Shannon

第3章　布尔代数和数字逻辑

3.1　概述

在阿伯拉罕·林肯（Abraham Lincoln）担任美国总统的时代，乔治·布尔（George Boole）正生活在英格兰。布尔是一位数学家和逻辑学家，他发明了一种采用代数体系来表达逻辑过程的方法。这种方法后来发展成为数学的一门分支学科，称为*符号逻辑学*（symbolic logic），或*布尔代数*（Boolean algebra）。后来，John Vincent Atanasoff 将布尔代数应用于数学计算。当时，Atanasoff 正在试图利用 Pascal 和 Babbage 采用的技术来建造一台计算机器，用于求解线形代数方程。在经历了多次失败以后，Atanasoff 有些沮丧。于是，他决定驾车外出。当他突然意识过来时，发现自己已经到了远离他的居住地 Iowa 州的 Ames 镇 200 英里以外的 Illinois 州。

Atanasoff 并没有打算开车到这么远的地方。当时，在 Illinois 州可以合法地在酒店买酒喝。于是，他便走进一家小酒店坐了下来，要了一杯波旁威士忌酒。他发现自己开车这么远，只是为了喝一杯酒。作者个人并不以为是这杯酒才使 Atanasoff 有了后来的灵感，事实上他根本就没有喝那杯酒。凭借深厚的物理和数学功底，他通过对过去失败的计算机器的反思，Atanasoff 在计算机器的新设计方案中做出了四个关键性的突破。

他决定采用电动方法来取代机械运动的方法。当时，使用真空电子管就可以实现。

因为要使用电动方法，所以 Atanasoff 决定采用基数为 2 的数制来代替基数 10 的计数体系。这样，基数 2 可以直接与电学开关的“开”或“关”的状态相关联。这种构想导致了后来发展的计算机系统是数字计算机，而不是模拟计算机。

他同时使用电容器作为存储器。因为电容器可以储存电荷，并且可以通过一个再生过程来避免电荷泄露。

对于计算方法，Atanasoff 将采用他称为“直接逻辑操作”的方法，而不是以前的各种计算机器所使用的枚举计数的计算方法。这种方法在本质上就是布尔代数。

要知道 Atanasoff 在当时并没有意识到他在解决问题时应用了布尔代数，而是经过反复试验，自己发明了直接逻辑操作方法。他当时并不清楚早在 1938 年，Claude Shannon 已经证明了 2 值的布尔代数可以描述 2 值的电学开关电路的操作。今天，人们都了解布尔代数在现代计算机系统设计中的重要性。因此，本书将用一章内容来讨论布尔逻辑，以及布尔代数与数字计算机的相互关系。

本章内容包括逻辑设计的简要介绍，布尔代数的基本知识，以及布尔代数与各种逻辑门和基本数字电路中的关系。也许读者已经从以前的编程经验中熟悉了一些基本的布尔算符。因此，更详细地学习有关布尔代数的内容显得很有必要。在本章中，读者将会看到布尔逻辑和计算机系统中的实际物理部件之间有着密切的联系。然而，作为一位计算机科学家，也许从来都不会有机会设计数字电路或者其他的物理部件。事实上，本章也不是为设计这些内容而准备的，只是希望读者可以具备足够的基本知识，来了解隐藏在计算机设计和具体实现中的一些基本动机和缘由。如果理解了布尔逻辑是如何影响各种计算机部件的设计，我们就可以从编程的角度更有效地使用各种不同的计算机系统。本章的结尾还列出了许多有关的资料，有兴趣的读者可以对相关的问题做更深入的探讨。

3.2　布尔代数

布尔代数处理的对象只能取两个值，如典型的例子是：对与错（或称为真与假）。当然，布尔代数

的对象可以是任意的一对数值。由于计算机是由一组或者处于“开”态，或者处于“关”态的电子开关构成，所以利用布尔代数来表示数字信息是一件非常自然的事情。实际上，数字电路是利用高低电压来表示各种数字信息，可以简单地理解为1和0。人们通常的解释是，数字1代表真，数字0代表假。

3.2.1 布尔表达式

除了表示这些二进制的对象外，布尔代数还可以对这些对象，或称为变量，进行操作运算。将变量和算符组合在一起就构成了布尔表达式（Boolean expression）。布尔函数（Boolean function）一般有一个或多个输入值，基于这些输入值而产生的一个操作结果：或者是0，或者是1。

常用的布尔算符有：*AND*、*OR* 和 *NOT*。为了更好地理解布尔算符，可以采用下面的方法来研究各个算符的行为：即将所有的输入值，及其各种组合列成表，并给出算符操作所得到的对应结果。这个表称为真值表（truth table）。对于特定的算符或函数，真值表采用表格的形式来表示其输入值与作用于这些输入变量所产生的结果之间的对应关系。真值表可以完整地描述一个布尔算符的行为。下面，分别采用布尔代数和真值表来表示布尔算符：*AND*、*OR* 和 *NOT*。

表3-1 AND的真值表

输入	输出
x y	xy
0 0	0
0 1	0
1 0	0
1 1	1

表3-2 OR的真值表

输入	输出
x y	$x+y$
0 0	0
0 1	1
1 0	1
1 1	1

表3-3 NOT的真值表

输入	输出
x	x'
0	1
1	0

逻辑算符AND（称为“与”）可以采用一个圆点来表示，或者中间无符号。例如，布尔表达式 xy 和表达式 $x \cdot y$ 是等效的，读作“x 与 y”。表达式 xy 也常常被称为布尔积（Boolean product）。当然，也利用真值表来表示的AND算符的行为特性，如表3-1所示。

可见，只有当两个输入值都为1时，表达式 xy 的结果才等于1；而对于其他的输入值，其结果都是0。表中的每一行代表了一个不同的布尔表达式。并且，所有 x 和 y 值的各种可能组合都分列在真值表的不同行中。

布尔算符OR，通常使用加号来表示。因此，表达式 $x + y$ 读为“x 或 y”。只有当输入都为0时，$x + y$ 的结果才为0，而其他结果都是1。表达式 $x + y$ 通常也被称为布尔和（Boolean sum）。OR的真值表如表3-2所示。

逻辑算符NOT，通常用上划线或单引号来表示。因此，$\bar{x}$ 和 x' 读作“非 x”。NOT的真值表如表3-3所示。

显然，布尔代数处理的是二进制变量和相关的逻辑运算。下面讨论将多个布尔变量和多个逻辑算符组合在一起构成布尔表达式的情形。例如，布尔函数：

$$F(x, y, z) = x + \bar{y}z$$

就是一个包含布尔变量 x、y 和 z 以及逻辑算符AND、OR和NOT的布尔表达式。其中到底首先执行那一个算符呢？布尔算符运算的顺序规则是：NOT为最高优先级，首先执行，其次是AND，然后为OR算符。这样，上面的函数 F 的运算过程为：先对 y 进行“非”操作取“反”，然后执行 $\bar{y}$ 和 z 的“与”操作，得到的结果最后和 x 一起完成“或”操作。

表3-4 $F(x, y, z) = x + \bar{y}z$ 的真值表

输入		输出
x y z	$\bar{y}$ $\bar{y}z$	$x+\bar{y}z=F$
0 0 0	1 0	0
0 0 1	1 1	1
0 1 0	0 0	0
0 1 1	0 0	0
1 0 0	1 0	1
1 0 1	1 1	1
1 1 0	0 0	1
1 1 1	0 0	1

同样，也可以采用真值表来表示这个表达式。生成真值表的形式对于表示类似这样的一些更复杂的函数很有帮助，可以利用真值表中的不同列来分别计算函数的各个部分结果，直到获得最终结果为止。函数 F 的真值表如表3-4所示。

真值表的最后一列给出了函数 F 对应于各种 x、y 和 z 组合的结果。实际上，函数 F 的真值表只是由表的前二列和最后一列构成。表中划阴影部分的列表示的是要求出最后结果所需要的中间步骤。按这种方式生成真值表，可以使各种复杂表达式的计算变得更容易。

3.2.2 布尔恒等式

布尔表达式常常不是以最简单的形式出现。回忆一下代数中情形，表达式 $2x+6x$ 就不是最简单的表达式，可以将其化简为 $8x$。布尔表达式也可以进行类似的化简。布尔代数有自己的一些恒等式（identity），或称为定律。这些恒等式可以作用于单个布尔变量或者是布尔表达式，现列于表 3-5。表中的每个定律（最后一个除外）都有 AND（积）和 OR（和）两种形式，这种规律称为对偶原理（duality principle）。

表 3-5 布尔代数的基本恒等式

恒等式	与（AND）形式	或（OR）形式
一定律	$1x=x$	$0+x=x$
零律	$0x=0$	$1+x=1$
幂等律	$xx=x$	$x+x=x$
互补律	$x\bar{x}=0$	$x+\bar{x}=1$
交换律	$xy=yx$	$x+y=y+x$
结合律	$(xy)z=x(yz)$	$(x+y)+z=x+(y+z)$
分配律	$x+yz=(x+y)(x+z)$	$x(y+z)=xy+xz$
吸收律	$x(x+y)=x$	$x+xy=x$
德摩根律	$\overline{(xy)}=\bar{x}+\bar{y}$	$\overline{(x+y)}=\bar{x}\bar{y}$
还原律	$\bar{\bar{x}}=x$	

一致律所表示的是任一布尔变量“与”1 操作和“或”0 操作，其结果还是原来的变量值；对“与”操作来说，1 是恒等元素；而对“或”操作，0 是恒等元素）。零律则表示任意布尔变量“与”0 结果为 0，而“或”1 的结果为 1。幂等律说明一个变量和变量自身的“与”和“或”操作结果还为原来的变量值。互补律则说明了一个变量与其反码的“与”操作和“或”操作产生结果分别为 0 和 1 的恒等式。结合律和交换律与普通代数相同。布尔变量可以通过相互交换来重新排序，改变结合来重新分组，而不会影响最终的结果。分配律则是表示如何把“或”操作转化为“与”操作，或者是反过来把“与”转换成“或”操作。

吸收律和德摩根律的关系看起来并不是十分明显，我们可以采用真值表来对这些等式进行证明。如果这些等式左右两边的表达式相等，那么这些表达式就应该有相同的真值表。表 3-6 表示“与”形式的德摩根律等式左右两边函数的真值表。其他定律的证明，如“或”形式的德摩根律以及吸收定律，留给读者作为一个练习。

表 3-6 “与”形式的德摩根律的真值表

x	y	(xy)	$\overline{(xy)}$	$\bar{x}$	$\bar{y}$	$\bar{x}+\bar{y}$
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

还原律所说明的是两次取反结果被还原的思想，这种情况在高中课程中就已经熟悉。还原律无论在数字电路还是在人们的日常生活中都很重要。例如，如果用 x 表示你所拥有的现金数量（假定为正值），如果现金被借走，就用 $\bar{x}$ 表示。当一个不值得你信任的熟人来向你借些现金时，你可以就实地告诉他手头没有钱。因为，$x=\overline{(\bar{x})}$，即使借走的现金刚刚还回来了。

初学者在进行布尔逻辑运算时，常犯的一个错误是认为：$\overline{(xy)}=\bar{x}\bar{y}$。这实际上不是一个正确的等式。从德摩根律可以清楚地看出这种运算的错误。但是，这的确是一个非常容易犯的错误，应该注意避免。

3.2.3 布尔表达式的化简

在代数学中，可以利用代数恒等式来对一些方程式进行化简。例如，可以把表达式 $10x+2y-x+3y$ 化简为最简化的形式 $(9x+5y)$。类似地，布尔恒等式也同样可以用来对布尔方程式进行化简。下面是一些应用举例。

例 3-1 假定函数 $F(x,y)=xy+xy$，利用“或”形式的幂等律，将方程式中的 xy 当作一个单一布尔变量。原来的表达式被化简为 xy，结果为：$F(x,y)=xy+xy=xy$。■

例 3-2 已知函数 $F(x,y,z)=\bar{x}yz+\bar{x}y\bar{z}+xz$，化简过程如下：

$F(x,y,z)=\bar{x}yz+x\bar{y}z+xz$

$=xy(z+\bar{z})+xz$ （分配律）

$=\bar{x}y(1)+xz$ （互补律）

$=\bar{x}y+xz$ （一致律） ■

通常，化简过程真截了当，就如同例 3.2 所示。然而，有时化简过程中恒等式的运用需要一些技巧，如例 3-3 所示。

例 3-3 已知函数 $F(x,y,z)=xy+xz+yz$，化简如下：

$=xy+\bar{x}z+yz(1)$ （一致律）

$=xy+\bar{x}z+yz(x+\bar{x})$ （互补律）

$=xy+\bar{x}z+(yz)x+(yz)\bar{x}$ （分配律）

$=xy+xz+x(yz)+\bar{x}(xy)$ （交换律）

$=xy+xz+(xy)z+(\bar{x}z)y$ （结合律）

$=xy+(xy)z+\bar{x}z+(\bar{x}z)y$ （交换律）

$=xy(1+z)+\bar{x}z(1+y)$ （分配律）

$=xy(1)+\bar{x}z(1)$ （零律）

$=xy+\bar{x}z$ （一致律） ■

例 3-3 所证明的就是大家熟知的一致性理论（Consensus Theorem）。

有时需要在表达式中插入一些项来简化布尔函数。这是一个技巧性问题。没有通用的法则，只有通过不断练习才能熟练掌握布尔代数的化简过程。当然，还有其他一些方法可以用来化简布尔方程式，这将在以后的章节中介绍。

同样，可以利用这些布尔恒等式来证明布尔等式。例如，要证明等式 $(x+y)(x+\bar{y})=x$，证明过程如表 3-7 所示。

要证明两个布尔表达式相等，可以分别生成两个真值表，然后进行比较。如果真值表相同，那么表达式也相等。作为一个练习，读者可以利用真值表来证明表 3-7 中的等式。

表 3-7 应用恒等式的例子

证 明	恒等式
$(x+y)(x+\bar{y})=x\bar{x}+xy+yx+yy$	分配律
$=0+xy+x\bar{x}+yy$	互补律
$=0+xy+y\bar{x}+y$	幂等律
$=xy+yx+y$	一致律
$=y(x+\bar{x})+y$	分配律（和交换律）
$=y(1)+y$	互补律
$=y+y$	一致律
$=y$	幂等律

3.2.4 反码

如例 3-1 所示，布尔恒等式可以应用于布尔表达式，而不仅仅是布尔变量（如例子中将 xy 当作单一布尔变量，然后应用幂等律）。这一点对于布尔算符来说也是相同的。最常用的布尔算符是 NOT 算符，可以应用于更复杂的布尔方程式，生成该表达式的反码（complement）。后面读者可以看到布尔函数与其采用电子线路组成的物理实现之间存在着一一对应关系。在多数情况下，从电路上实现一个函数的反码要比实现函数本身简单和经济一些。要实现反码功能，必须对原函数的输出结果进行取反操作。这只需简单地实施一个 NOT 操作。在数字系统中，反码是非常有用的。

可以使用德摩根律，来对一个布尔函数进行取反，求出其反码。"或"形式的德摩根律为 $\overline{(x+y)}=\bar{x}\bar{y}$。可以很方便地将该定律推广到 3 个或更多变量的情形。已知以下函数：

$F(x,y,z)=\overline{(x+y+z)}$

设 $w=(x+y)$，则有：

$F(x,y,z)=\overline{(w+z)}=\bar{w}\bar{z}$

现在，再应用德摩根律，得到：

$\bar{w}\bar{z}=\overline{(x+y)}\bar{z}=\bar{x}\bar{y}\bar{z}=F(x,y,z)$

因此，如果 $F(x,y,z)=(x+y+z)$，则有 $\overline{F}(x,y,z)=\overline{x}\,\overline{y}\,\overline{z}$。应用对偶原理，不难得出 $(\overline{xyz})=\overline{x}+\overline{y}+\overline{z}$。

很显然，求布尔表达式的反码，只需简单地用其反码取代各个变量（如用 $\overline{x}$ 取代 x），并且将表达式中的“与”操作和“或”操作互换。这正是德摩根律所阐述的。例如，函数 $x+yz$ 的反码为 $x(\overline{y}+\overline{z})$。注意，要加上适当的括号，以确保运算次序的正确性。

这是一条求布尔表达式的反码的简单经验法则，可以通过比较表达式与其反码的真值表来验证这种方法的正确性。利用真值表表示反码时，要注意原表达式输出值为 0 处，对应的反码输出应该为 1；而原表达式输出为 1 的地方，反码输出则对应为 0。表 3-8 为函数 $F(x,y,z)=x+yz$ 及其反码 $\overline{F}(x,y,z)=x(\overline{y}+\overline{z})$ 的真值表。阴影部分分别表示 F 和 $\overline{F}$ 的最终结果。

表 3-8 一个布尔函数及其反码的真值表

x	y	z	yz	$\overline{x}+\overline{yz}$	$\overline{y}+\overline{z}$	$x(\overline{y}+\overline{z})$
0	0	0	0	1	1	0
0	0	1	0	1	1	0
0	1	0	0	1	0	0
0	1	1	1	1	1	0
1	0	0	0	0	1	1
1	0	1	0	0	1	1
1	1	0	0	1	0	0
1	1	1	1	0	1	1

3.2.5 布尔函数的表示方法

对于一个已知的布尔函数，可以有多种表示方法。例如，可使用真值表，或者众多不同的布尔表达式中的某一个来表示。实际上，布尔函数可以有无限多个不同的布尔表达式，它们在逻辑上都是相互等效的（logically equivalent），参见例 3-4。

例 3-4 如果已知 $F(x,y,z)=x+\overline{x}y$。很显然，在形式上也可以用 $F(x,y,z)=x+x+\overline{x}y$ 来表示 F。因为幂等律已经证明了这两个表达式的结果是一样的。同样，利用分配律也可以将 F 表示为 $F(x,y,z)=x(1+\overline{y})$ 的形式。■

为了减小可能带来的混淆，逻辑设计人员一般使用一种规范（canonical）形式，或称为标准化（standardized）形式来描述特定的布尔函数。对于任意的布尔函数，都存在某个唯一的标准化形式。然而，不同的设计人员可以使用不同的“标准化”形式。最通用的两种标准化形式是积之和的形式，以及和之积的形式。

积之和形式（sum-of-products form）是将表达式先写成一些变量“与”项（乘积项）的集合，然后再利用“或”形式组合在一起。例如，函数 $F_1(x,y,z)=xy+y\overline{z}+xyz$ 就是积之和的形式。而函数 $F_2(x,y,z)=xy+x(y+z)$ 则不是积之和形式。当然，可以利用分配律来展开 F_2 函数中的括号项，使表达式变化为 $xy+xy+xz$，这就是一个积之和形式。

布尔表达式也可以表示为和之积（product of sums）形式。和之积的形式就是将表达式先写成各变量的“或”项（求和项）然后用“与”的形式组合在一起。例如，函数 $F_1(x,y,z)=(x+y)(x+\overline{z})(\overline{y}+z)(\overline{y}+\overline{z})$ 就是采用和之积的形式。一般来说，如果表达式中估值为真的情形多于估值为假时，可选用和之积的形式。函数 F_1 不属于这类情况，所以选择积之和的形式是合适的。同时，积之和的形式也较易处理和简化。因此，在下面部分的介绍中采用这种积之和的形式。

任何一个布尔表达式都可以表示为积之和的形式。因为任意一个布尔表达式也可以表示为真值表的形式，所以任意一个真值表也同样可以表示为积之和的形式。事实上，可以非常方便地将真值表转化为积之和的表示形式，如下例所示。

例 3-5 考虑一个简单的择多函数。已知函数有 3 个输入。如果少于一半的输入为 1，输出就为 0；如果至少一半的输入为 1，则输出也为 1。表 3-9 为 3 个输入的择多函数的真值表。■

为了将真值表转换为积之和的形式，首先从反过程来考虑这一问题。如果要使表达式 $x+y$ 等于1，那么 x 或者 y 中至少有一个（或者两个）等于1。而如果要 $xy=yz=1$，则要求或者 $xy=1$，或者 $yz=1$（或者两者都等于1）。反过来使用这一逻辑，并将其应用于例3-5中。可以发现，如果 $x=0$，$y=1$ 和 $z=1$ 时，函数的输出肯定是1。满足这一关系的乘积项为 xyz（很显然，当 $x=0$，$y=1$ 和 $z=1$ 时，该乘积项等于1）。第二个输出为1的情形为，$x=1$，$y=0$ 和 $z=1$，对应的乘积项是 xyz。第三个乘积项是 xyz，而最后一个乘积项为 xyz。归纳起来，要利用真值表生成积之和的表达式，就是找到真值表中输出值为1的对应行，将该行的输入变量写成积的形式。在生成的每个乘积项中，都必须将输入为0的变量取反。

表3-8 择多函数的真值表表示法

x	y	z	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

这样，择多函数可以按照积之和的形式表示为：$F(x,y,z)=\bar{x}yz+x\bar{y}z+xy\bar{z}+xyz$。注意，这一表达式可能不是最简单的形式，但是，它一定是一个标准形式。在表示布尔函数时，积之和的标准形式与和之积的标准形式是完全等价的。利用布尔恒等式可以从其中的一种表示形式转换成另一种表示形式。但是，无论是采用积之和，还是和之积的表示形式，表达式最终都要转化成最简单的表示形式，即将表达式化简成最少的项。化简的理由是因为布尔表达式与其电路的物理实现之间存在着一一的对应关系，这种对应关系将在下面的章节中介绍。表达式中不必要的项，会在物理实现中造成不必要的部件，结果难以生成理想的电路。

3.3 逻辑门

到目前为止，所讨论的逻辑算符 AND，OR 和 NOT 都是一些使用真值表和布尔表达式的抽象表示。实际的物理部件，或者说数字电路（digital circuit），例如计算机中执行算术操作或做出抉择的部件都是由一定数目的称为门电路（gate）的最原始的基本单元构成的。这些门电路是数字设计的基本构件，可以执行前面所讨论的各种基本逻辑功能。从形式上来说，门电路是一个计算各种2值函数的微小电子元件。更简单地说，门电路执行一个简单的布尔逻辑功能。在物理上构成一个门电路需要1到6个，或更多的晶体管（晶体管已经在第1章中做了介绍），具体晶体管的数目取决于所采用的半导体技术。归纳起来，计算机的基本物理部件是晶体管，而基本的逻辑单元是门电路。

3.3.1 逻辑门的表示符号

首先来讨论三种最简单的门电路，它们分别对应于逻辑算符 AND、OR 和 NOT。前面已经介绍了这些布尔算符的功能特性。图3-1是表示这些算符对应的门电路的图形符号。

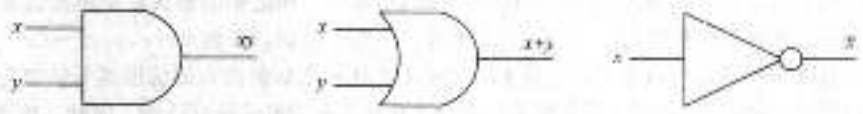


图3-1 三种基本门电路

注意，非门的输出端有一个圆圈，通常，这种圆圈表示取反（非）操作。

另外一个常用的门电路是异或（exclusive-OR，XOR）门，采用布尔表达式表示为：$x\oplus y$。如果两个输入值相等，XOR 输出为假（0）；否则输出为真（1）。图3-2是说明 XOR 功能行为的真值表和逻辑图形符号。

x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

a)异或门（XOR）的真值表



b)异或门（XOR）的逻辑符号

图 3-2 异或门

3.3.2 通用门电路

其他两个常用的门电路是与非门（NAND）和或非门（NOR），它们分别生成与门和或门的反码输出。这里，每种门电路都可以采用两种不同的逻辑符号来表示（作为一个练习，读者可以证明这两种符号表示在逻辑上是等价的，提示：使用德摩根律）。图 3-3 和图 3-4 分别表示与非门和或非门的逻辑符号图，以及与每种逻辑门的功能行为相对应的真值表。

x	y	x NAND y
0	0	1
0	1	1
1	0	1
1	1	0

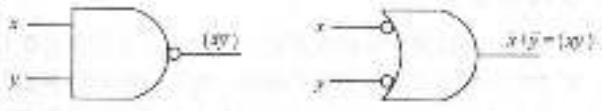


图 3-3 与非门（NAND）的真值表和逻辑符号

x	y	x NOR y
0	0	1
0	1	0
1	0	0
1	1	0

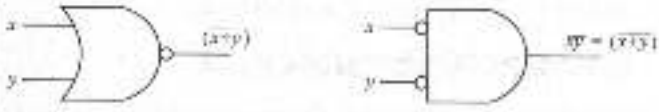


图 3-4 或非门（NOR）的真值表和逻辑符号

与非门通常也称为万能门（universal gate），因为只要使用与非门就可以构建所有的数字电路。为了证明这一点，图 3-5 给出了仅采用与非门组成的与门、或门和非门。

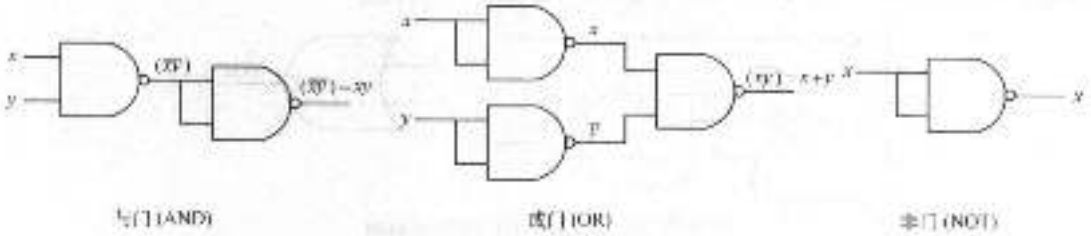


图 3-5 仅采用与非门构建的三种数字电路

读者也许会问，为什么不简单地采用已经熟知的与门、或门和非门，而要使用单一与非门来构建各种数字电路呢？事实上有两个理由：第一，使用与非门的电路成本比较低。第二，在物理上构建复杂的集成电路（这些内容将在后面的章节中讨论）时，使用相同的结构模块（比如，几个与非门）常常比使用一些基本结构模块的集合（例如，一些与门、或门和非门的组合）要容易得多。

对偶原理同样也适用于这种通用性规则。这样，可以使用或非门来构建任意的数字电路。正如前面所述，与非门和或非门之间是相互关联的。在逻辑上，它们分别对应于积之和与和之积的表现形式。这就是说，积之和形式的布尔表达式可以采用与非门来实现，而和之积的表示形式则可以使用或非门来实现。

3.3.3 多输入的门电路

到目前为止，所有讨论的门电路都只有两个输入端。显然，门电路的输入值可以多于两个。不同

的门电路可以有不同数目和种类的输入和输出方式。例如，这里使用一个三输入的或门来表示布尔表达式：$x+y+z$，如图 3-6 所示。

而图 3-7 表示的是表达式 $x\bar{y}z$。

在本章的后面部分，读者可以看到，数字电路中通常会将门电路的输出端用符号 Q 来表示，而同时也将输出的反码 $\bar{Q}$ 在图形符号中表示出来。这种图形表示法有时很有用。如图 3-8 所示。

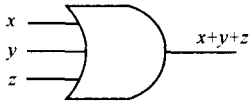


图 3-6 一个三输入的或门表示 $x+y+z$

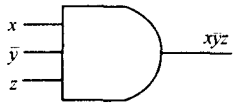


图 3-7 一个三输入的与门表示 $x\bar{y}z$

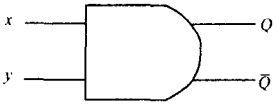


图 3-8 具有两个输入和两个输出的与门图形符号

3.4 数字电路元件

打开一台计算机，可以发现计算机系统由大量数字电路元件组成。组成计算机的大量的门电路通过导线连接，这些导线构成了系统的信号通路。这些门电路的集合常常是一些非常标准的电路。这些标准电路组成一系列的基本结构模块，可以用来构建整个计算机系统。令人惊讶的是，这些结构模块全部采用基本的“与”、“或”和“非”操作来构建的。在接下来的部分，我们将讨论各种数字电路，这些数字电路与布尔代数的相互关系，各种标准结构模块，以及利用这些结构模块来构建两种不同类型的数字电路例子，组合逻辑电路和时序逻辑电路。

3.4.1 数字电路及其与布尔代数的相互关系

到底布尔函数和数字电路之间存在着什么联系呢？从前面的讨论可知，一个简单的布尔操作（比如，“与”操作或者是“或”操作）可以用一个简单的逻辑门电路表示。各种复杂的布尔表达式可以表示为一些“与”门、“或”门和“非”门的组合形式，结果也就是描述整个布尔表达式的逻辑图。这种逻辑图实际上代表的是特定的布尔表达式的一种物理实现，即真实的数字电路。现在来考虑前面提到的函数 $F(x,y,z)=x+\bar{y}z$。图 3-9 表示了实现这个函数的一个逻辑图。

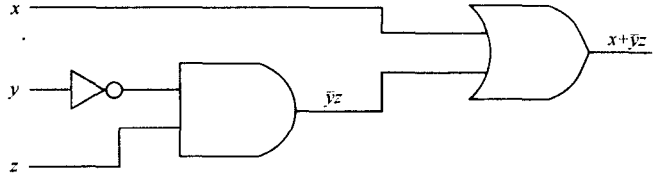


图 3-9 $F(x,y,z)=x+\bar{y}z$ 的逻辑图

很明显，对于任何布尔表达式都可以建立其逻辑图，而由逻辑图可以实现数字电路。

利用布尔代数可以分析和设计各种数字电路。因为布尔代数和逻辑图之间存在着相互对应的关系，所以通过化简布尔表达式可实现简化电路的目的。各种数字电路都是通过门电路来实现的。但是，在设计层次上，逻辑门和逻辑图并不是表示数字电路的最方便的形式。这一层次上，更好的选择是使用布尔表达式，因为布尔表达式更容易处理和化简。

布尔函数的表达式的复杂性将直接影响到实现布尔函数的数字电路的复杂程度。布尔表达式越复杂，其实现的数字电路也越复杂。应该指出的是，简化电路时并不一定要使用布尔恒等式。从前面的介绍可知，使用布尔恒等式来化简电路有时是很困难和非常费时的。因此，设计人员通常采用一种更加自动化的方法来完成函数的化简工作。这种方法就是使用卡诺图（Karnaugh maps，或 Kmaps）的方法。感兴趣的读者可以参考本章后面的专题内容，了解如何利用卡诺图来帮助化简数字电路。

3.4.2 集成电路

计算机由各种数字电路元件构成，这些元件通过导线连接在一起。就像一个好的程序一样，实际的计算机硬件由一些较大的数字电路模块组成。而这些电路模块是一些基本门电路的组合，不同的结构模块执行不同的功能。生成这些“结构模块”所需的门电路的数目取决于所采用的制造技术。因为电路制造技术不属于本书的内容，读者可以参阅本章后面的文献资料，了解更多有关电路制造技术方面的知识。

通常，门电路并不单独销售，而是以集成电路（integrated circuits，IC）的形式出现。集成电路芯片（一块小的半导体硅单晶片）上面集成了各种电子元件（晶体管、电阻和电容），用来实现各种门电路的功能。正如第 1 章所述，集成电路中的电子元件是直接在半导体芯片上腐蚀制成。因此，与对等的分立器件相比，集成电路上的元件的体积更小，工作时消耗的电能更少。这种芯片封装在有外接管脚的陶瓷或塑料的包装中。利用焊接引线连接芯片和外部的管脚，完成集成电路制造。早期的集成电路包含的晶体管数目比较少，称为小规模集成电路（SSI）。一般每块芯片上不到 100 个晶体管。现在使用的是超大规模集成电路（ULSI），每块芯片上的电子元件数目超过一百万个。图 3-10 是一个小规模集成电路的示意图。

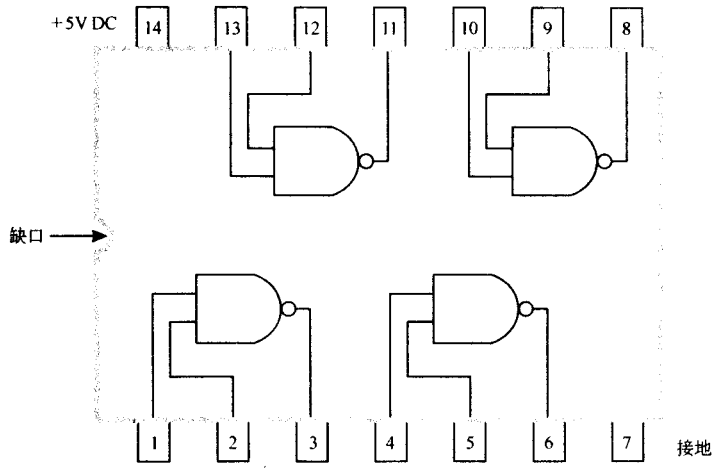


图 3-10 一个简单的小规模集成电路（SSI）

3.5 组合逻辑电路

电路是由数字逻辑芯片组合而成的。这些逻辑电路可分为组合逻辑和时序逻辑。本节介绍组合逻辑，时序逻辑将在第 3.6 节介绍。

3.5.1 基本概念

可以利用组合逻辑来构建包含基本布尔算符、输入和输出的数字电路。了解组合逻辑的关键是组合逻辑的输出完全取决于所给定的输入值。因此，组合逻辑电路的输出是输入值的函数，并且任意时刻的输出值都是由该时刻的输入值唯一确定的。一个组合逻辑电路可能会有几个输出。这种情况下，每个输出都代表了一个不同的布尔函数。

3.5.2 典型的组合逻辑电路

首先讨论一个称为“半加器（half-adder）”的非常简单的组合逻辑电路。下面考虑两个二进制数

字相加的问题。这里只有三种可能的运算：0+0=0，0+1=1+0=1 和 1+1=10。大家非常清楚这种电路的行为状态，并可以将电路的行为特性用真值表的形式表示出来。这里需要指出的是，半加器有两个输出，而不是一个。因为需要其中一个输出表示求和结果，另外一个输出表示进位。半加器的真值表如表 3-10 所示。

仔细观察，可以发现求和操作实际上是运算一个或非函数（XOR），而进位输出等效于一个“与”（AND）门的输出。将一个“或非”门和一个“与”组合在一起，就得到了半加器的逻辑图，如图3-11所示。

表 3-10 半加器的真值表

输入		输出	
x	y	和	进位
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

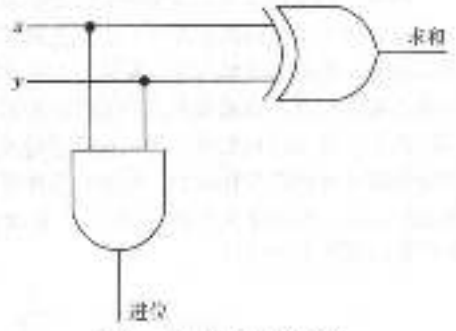
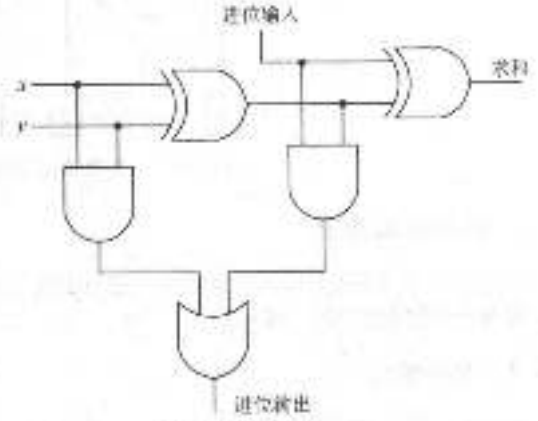


图 3-11 半加器的逻辑图

半加器是一个非常简单的数字电路，但并不是很实用。原因是半加器只能对两个一位的二进制数字进行相加。当然，可以对半加器进行扩充，使其电路能够完成较大的二进制数字的加法运算。例如，考虑十进制数的加法：先对最右边一列相加，即个位数字相加，进位加到十位上，如此类推。我们使用相同的方法，执行二进制数字的加法运算。因此，这个加法电路需要有三个输入（x、y 和进位输入）和两个输出（求和结果和进位输出）。图 3-12 表示一个全加器（full adder）的真值表和对应的逻辑图。不难看出，全加器由两个半加器和一个“或”门组成。

输入			输出	
x	y	进位输入	和	进位输出
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

a) 全加器的真值表



b) 全加器的逻辑图

图 3-12 全加器的真值表对应的逻辑图

读者也许会奇怪，这样一个全加器电路如何对二进制数进行相加，因为它只能加三位。事实上，这个电路的确不能完成多位数字的运算。但是，只需将上面的加法器电路复制 16 次，就可以构建一个 16 位字的全加器。具体做法是，将每个单一加法器电路的进位输出线连接到相邻左边的加法器的进位输入端，如图 3-13 所示。这种结构的电路称为逐位进位加法器（ripple-carry adder），因为产生的进位是按顺序逐位通过各个加法器。注意，图中没有画出构成加法器的所有门电路，而是采用一个黑盒子

(black box) 来代表加法器。采用黑盒子的方式可以忽略实际门电路的细节，而只需关心电路的输入和输出。对大多数数字电路来说，包括后面马上要介绍的译码器、多路复用器和加法器等，采用黑盒子表示法是一种常用方法。

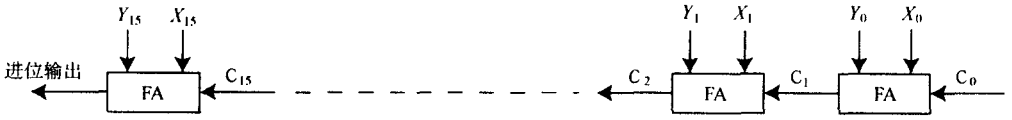


图 3-13 逐位进位加法器的逻辑图

因为这种逐位进位加法器的运算速度非常慢，所以现实中不会采用这种方法来实现加法运算。但是，这种加法器的原理很简单，有助于理解如何实现较大的二进制数字的加法运算。加法器设计中采用的各种改进方法包括：使用先行进位加法器、选择进位加法器和存储进位加法器，以及其他一些方法。各种改进方法的目的都是要缩短由于两个二进制数字相加的逐位进位过程所造成的延迟时间。事实上，这些改进设计的新加法器通过实行并行加法操作和减小最大进位路径的方法，可以获得比逐位进位加法器快 40%—90%的速度。加法器是一个非常重要的电路。如果不能进行数字加法运算，计算机就会变得毫无用处。另外，各种计算机都需要频繁使用的译码操作也同样非常重要。译码器是从一组 n 个输入得到最大的 2^n 个输出，从而实现对二进制信息的解码。*译码器* (decoder) 通过利用输入和对应的数值来选择某根特定的输出线。“选择一根输出线”是什么意思？这就意味着只有唯一的一根输出线被断言，或者说被设置成 1，而其他输出线都被设置为 0。译码器通常采用输入和输出的数目来定义。例如，一个 3 输入和 8 输出的译码器称为 3-8 译码器。

上面提到译码器是各种计算机中都需要频繁使用的部件。关于这一点，读者可能知道许多计算机中必须执行的算术操作，但却发现很难举出一个译码器的例子来。如果是这种情况，读者肯定对计算机访问存储器的原理不太熟悉。

计算机中的所有存储器地址都是采用二进制数字来指定分配的。当需要引用某个存储器地址时(也许是要读取数据，也许是为了写入数据)，计算机必须首先确定该存储器的有效 (实) 地址。这个操作需要利用译码器来完成。下面将举例说明有关译码器工作原理的各种问题和它的基本用途。

例 3-6 一个 3-8 译码器

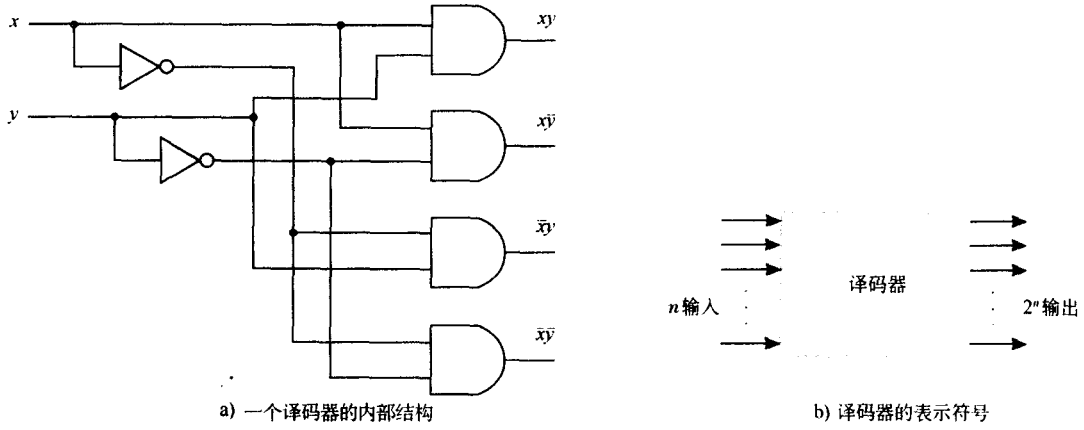
假设某个存储器由 8 片存储芯片组成，每片可以存储 8K 字节的信息。假定编号为 0 的芯片 0 的存储器地址为 0—8191，芯片 1 的地址为 8192—16383，其他如此类推。共有 8K×8=64K (65，536) 存储器地址。现在并不需要将所有 64K 地址都用二进制数写出来，而是只有少数地址采用二进制的形式编写 (有关这一点，下面将会详细说明)，这样就可以解释为什么需要一个译码器。

显然，$64=2^6$，$1K=2^{10}$，$64K=2^6\times2^{10}=2^{16}$，要表示 64K 地址的每一位需要 16 位二进制数。如果一时难于理解的话，可以先来看一些较小的地址数值。例如，现在有 4 个地址——地址 0、1、2 和 3，与这些地址对应的等效二进制编码数分别为 00、01、10 和 11。也就是说，表示 4 个地址只需要 2 位数字，因为 $2^2=4$。同样，如果考虑 8 个地址，从 0 到 7。如果用二进制数来编号的话，需要几位数字？答案是 3 位，因为 $8=2^3$。大家可以很方便地把这 8 个地址写出来。显然，2 的幂指数就是表示地址所需要的最小二进制数的位数。

现在回到原来的问题，芯片 0 上所有的地址的格式都为：000*xxxxxxxxxxxxx*。由于芯片 0 包含的地址范围是：0—8192，因此这些地址所对应的二进制数位于 0000000000000000—0001111111111111 的范围内。类似地，芯片 1 上的地址格式是：001*xxxxxxxxxxxxx*，其他的芯片也按此法类推。很明显，地址编码中最左边的 3 位二进制数表示每个芯片所处的位置。对于整个存储器系统，需要 16 位地址来进行编码。但是，对于每个存储器芯片来说，只有 2^{13} 个地址。所以，在某个芯片内部，只需要 13 位二进制数就可以唯一地确定每个地址的位置。这也正是最右边的 13 位二进制数字所代表的信息。

当计算机进行寻址操作时，首先要确定使用的是哪个芯片，然后再在指定的芯片内找出实际的地址。在本例中，计算机使用最左边的 3 位二进制数来选择存储器芯片，用余下的 13 位在所选定的芯片内部进行寻址。这 3 个高位实际上会被用作译码器的输入，计算机将通过译码器来选通所要访问的存储器芯片。如果前面的 3 位为 000，那么就是芯片 0 被选通。如果前面的 3 位为 111，芯片 7 会被选通。如果前面的 3 位数是 010，那么又是哪个芯片被选中？显然是芯片 2。只要开启连通某条特定的导线就可以选通某个存储器芯片。译码器的一个输出就是用来激活一个芯片，也只是连通唯一的一个由编码地址指定的存储器芯片。■

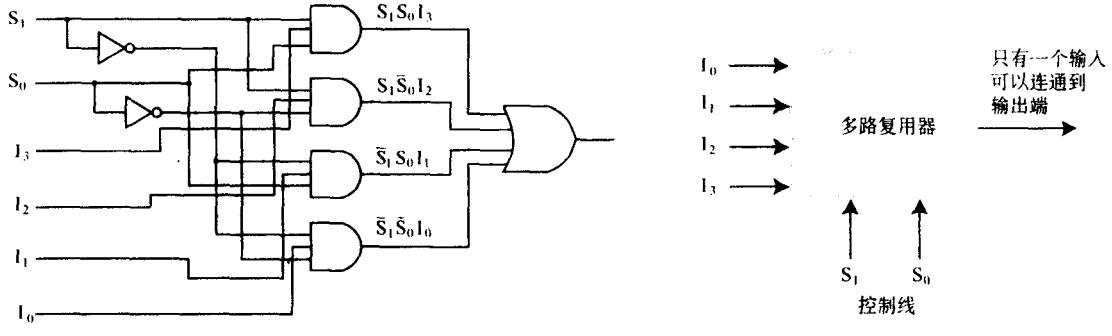
图 3-14 表示的是一个译码器的物理电路构成和译码器的表示符号。第 3.6 节将介绍译码器在存储器系统中的使用方法。



a) 一个译码器的内部结构　　b) 译码器的表示符号

图 3-14　译码器的内部结构及其表示符号

另外一个常用的组合逻辑电路是**多路复用器**（multiplexer）。多路复用器用来从众多的输入线中选择需要的二进制数的信息，然后直接用单一输出线输出。多路复用器通过一组选择变量，或称为控制线，来控制选定某个特定的输入线。在每个特定的时刻，都只有一个输入（被选中的输入）可以通过线路连通到输出端，而所有其他的输入都会被断开。如果控制线上的变量数值发生变化，那么连通到输出端的实际输入线也会随之发生改变。图 3-15 给出了一个多路复用器的物理电路构成和多路复用器的表示符号。



a) 一个多路复用器的内部结构　　b) 多路复用器的表示符号

图 3-15　多路复用器的内部结构及其表示符号

在什么情况下我们会使用多路复用器呢？分时共享的计算机就是使用多路复用器的输入连接多个用户使用终端。**调制解调器网络**（Modem pools）也是使用多路复用器将来自 Modem 的各个线路输入到计算机系统。

另外一个非常有用的组合电路是奇偶发生器和奇偶校验器电路（读者可以参阅第2章所介绍的奇偶校验的内容）。奇偶发生器（parity generator）电路可以生成所需要的奇偶位加到一个信息字上面；而奇偶校验器（parity checker）是用来检验一个信息字中的奇偶性（奇或者偶）是否正确。如果奇偶位有误，则表示检测到一个奇偶错误。

典型的奇偶发生器和奇偶校验器采用“或非”函数构成。假定采用的是奇校验，3位字的奇偶发生器真值表如表3-11所示。而4位字（3位信息位和1位奇偶校验位）的奇偶校验器的真值表如表3-12所示。如果检测到一个错误，奇偶校验器的输出为1；否则，输出为0。对应的奇偶发生器和奇偶校验器的逻辑图留给读者作为一个练习。

表3-11 奇偶发生器的真值表

x	y	z	奇偶校验位
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

表3-12 奇偶校验器的真值表

x	y	z	P	错误校验
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

组合逻辑电路的例子很多，如比较器、移位电路和可编程逻辑器件等，都是一些非常有用的组合数字电路，而且很容易理解。由于篇幅的缘故，在此不再一一列举。感兴趣的读者可以参阅本章后面有关组合逻辑电路的参考文献。在结束组合逻辑电路的话题之前，还要介绍一个很重要的组合逻辑电路——算术逻辑单元（arithmetic logic unit，ALU）。至此，我们已经学习了构建算术逻辑单元所需的所有部件。

图3-16是一个非常简单的ALU逻辑图，这个ALU可以执行4种基本操作：“与”、“或”、“非”和加法。加法运算的操作数是两个2位的机器字。利用两条控制线，f_1和f_0来确定CPU要执行的是何种操作。信号00表示加法运算（A+B）；01是NOTA；10为A OR B；而11则是A AND B。输入线A_2和A_1表示其中一个机器字的2位二进制，而B_2和B_1表示第二个字。C_0和C_1则是输出线。

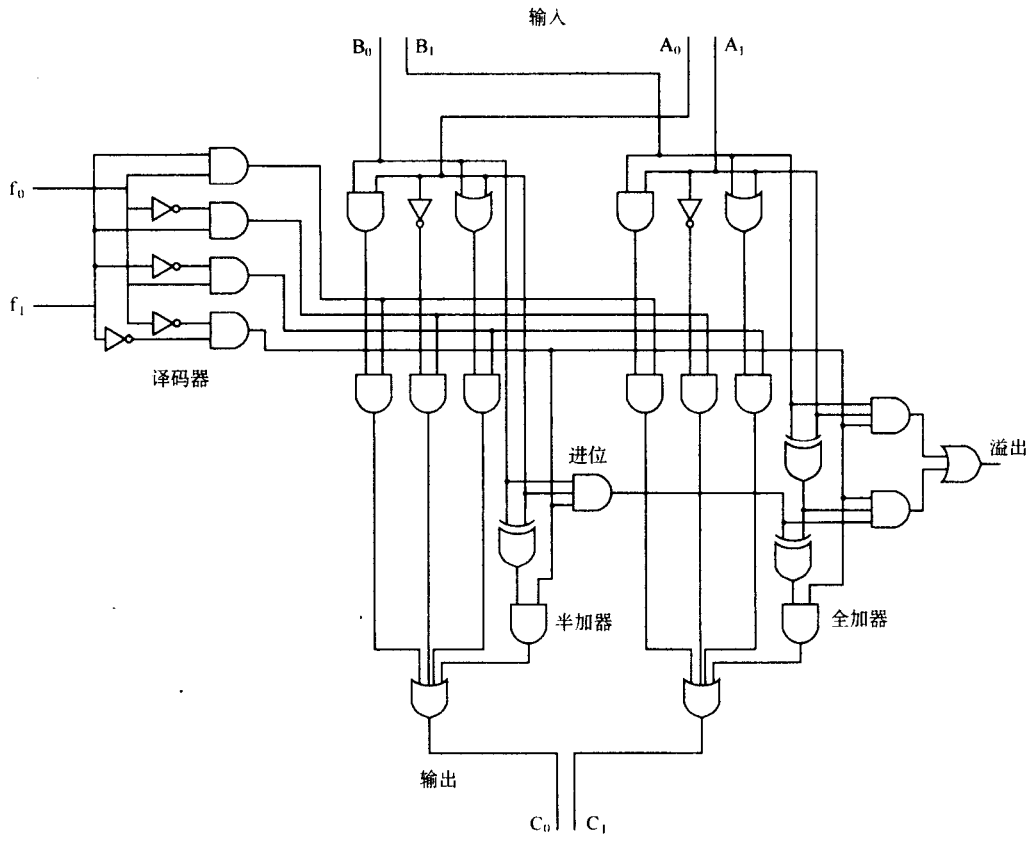


图 3-16 一个简单的 2 位 ALU

3.6 时序电路

上面讨论的是组合逻辑电路。研究组合逻辑电路的方法是分析布尔函数中各个变量与对应的函数值之间的关系。对于组合逻辑电路，函数的输出值完全取决于函数的输入值。如果输入值发生变化，则立即会对输出值产生直接影响。组合逻辑电路的最大缺点是没有存储的概念，即组合逻辑电路没有记忆功能。显然，组合逻辑电路在解决记忆问题的方面存在困难，而大家知道计算机需要有某种方法记忆各种数值。即使一个 soda 机（投币饮料机）中的一个非常简单的数字电路，也需要有记忆功能。当硬币投入机器中，电路必须记忆每次投币的数量。没有这种记忆能力，机器很难使用。因此，仅用组合逻辑电路是不能建造 soda 机的。要了解 soda 机的工作过程，从而最终能够理解计算机的工作原理，必须学习时序逻辑电路。

3.6.1 基本概念

时序逻辑电路的定义是，时序电路的输出同时是当前的输入与以前的输入的函数。因此，时序电路的输出与以前的输入状态有关。为了记忆前面的输入，时序电路必须具有某种类型的存储单元。通常，将时序电路中的这种存储单元称为触发器（flip-flop）。触发器的状态是此前的电路输入的函数。因此，即将发生的输出不但与当前的输入有关，而且与电路当前的状态有关。大家知道，组合逻辑电路是各种门电路的综合。同样，时序逻辑电路是各种触发器的综合。

3.6.2 时钟信号

在讨论时序电路之前，先介绍一下对事件进行排序的方法。因为时序电路必须依赖过去的输入来决定现在的输出，这意味着各种事件的发生必须有先后次序。有些时序电路是异步的（asynchronous），这就是说只要有任意的一个输入值发生改变，这类电路就被激活了。同步（synchronous）时序逻辑电路则是利用时钟来对各种事件进行排序。时钟（clock）是一种电子电路，它生成一系列具有精确宽度和间隔的连续脉冲。脉冲间隔称为时钟的周期时间（clock cycle time）。时钟速度一般用兆赫兹（MHz），或者说每秒钟百万个时钟脉冲。一般的时钟周期在 1 到几百 MHz 的范围。

时序电路使用时钟来确定何时刷新电路的状态（也就是何时将“现在”的输入变成“过去”的输入?）。这意味着电路的输入只能在一些特定的分立时刻影响时序电路的存储单元。本章主要讨论同步时序电路，因为与异步时序电路相比，同步时序电路更容易理解。从这个意义说，本书后面提到“时序电路”是指“同步时序逻辑电路”。

大部分时序电路是采用边缘触发的（另外，还有水平-触发电路），也就是电路的状态或者在时钟脉冲的上升边缘或者是在时钟脉冲的下降边缘发生改变，如图 3-17 所示。

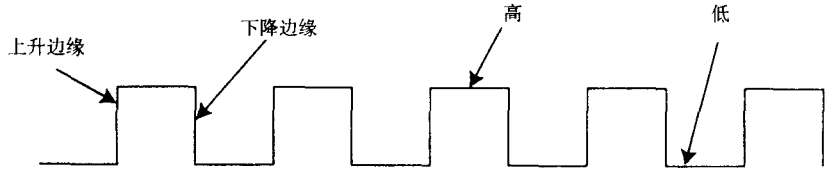


图 3-17 时钟脉冲信号示意图

3.6.3 触发器

水平触发电路只要时钟信号处于高电平或低电平时，就可以改变电路的状态。有时，人们常常会混淆锁存器（latch）和触发器（flip-flop）的这两个名称。从技术上来说，锁存器属于水平触发类型，而触发器则是边缘触发类型的电路。本书将使用触发器这一名称。

为了“记忆”过去的状态，时序电路使用了一个称为反馈（feedback）的概念。反馈就是将电路的输出返回连接到电路的输入端。一个双非门组成的简单反馈电路如图 3-18 所示。

在这个图中，如果输出 Q 为 0，就会一直为 0。如果输出 Q 为 1，就一直为 1。显然，这个电路不是非常有用，也没有技术上的兴趣，但有助于理解反馈的工作原理。

一个更有用的反馈电路由是由两个“与非”门组成的基本存储单元，称为 SR 触发器（SR flip-flop）。SR 表示“置位/复位（set/reset）”，其逻辑图如图 3-19 所示。

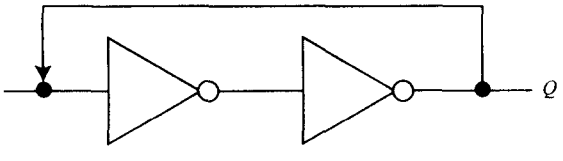


图 3-18 简单反馈电路的例子

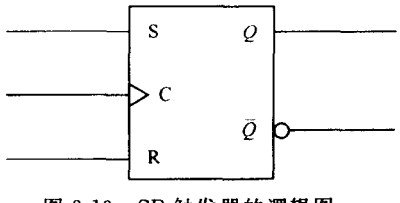
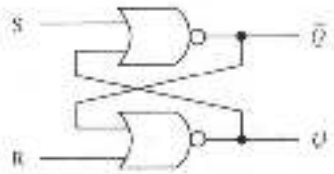


图 3-19 SR 触发器的逻辑图

我们可以使用所谓特征表（characteristic table）来描述触发器，特征表是基于电路的输入和当前的状态 Q 指出电路的下一个状态。一般采用符号 Q（t）表示电路当前的状态，Q（t+1）表示电路的下一个状态，或者说当时钟脉冲到来时，触发器应该进入的状态。图 3-20 表示一个 SR 触发器的物理电路的实现和特征表。



a) 一个实际的SR触发器

S	R	$Q(t+1)$
0	0	$Q(t)$（无变化）
0	1	0（复位为0）
1	0	1（置位为1）
1	1	未定义

b) SR触发器的特征表

图 3-20 SR触发器

SR触发器的工作特性是很令人感兴趣的。SR触发器有三个输入：S、R和当前的输出$Q(t)$。表3-13为SR触发器的真值表，可以说明其工作原理。

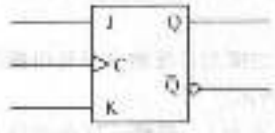
表 3-13 SR触发器的真值表

S	R	当前状态 $Q(t)$	下一状态 $Q(t+1)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	未定义
1	1	1	未定义

例如，如果S为0，R为0，且当前的状态$Q(t)$为0，那么下一个状态$Q(t+1)$也是0；如果S为0，R为0，且$Q(t)$为1，那么$Q(t+1)$为1。当(S,R)的实际输入为(0,0)时，时钟脉冲不会引起输出状态的变化。与此类似，如果输入(S,R)=(0,1)，则下一个状态$Q(t+1)$为0，而与当前的状态无关（即电路的输出被强制复位，reset），当(S,R)=(1,0)时，电路的输出被置位（set）为1。

如果S和R同时为1，这种特殊的触发器会是什么状态呢？应该是强制Q和$\bar{Q}$同时为0，但$Q=0=\bar{Q}$，这种情况显然是不可能的。这样会导致一种不稳定的电路状态。因此，对SR触发器来说，输入S和R同时为1的组合是不允许的。

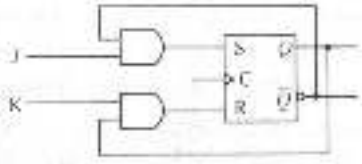
如果我们对SR触发器设置某些条件逻辑上的限制，就可以防止上面的非法状态发生。对RS触发器做如图3-21所示的简单修改，构成了一个JK触发器（JK flip-flop）。JK触发器是以Texas仪器公司的工程师Jack Kilby的名字命名的，他在1958年发明了集成电路。



a) JK触发器

J	K	$Q(t+1)$
0	0	$Q(t)$（无变化）
0	1	0（复位为0）
1	0	1（置位为1）
1	1	$\bar{Q}(t)$

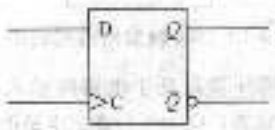
b) JK触发器的特征表



c) 利用SR触发器构成JK触发器的逻辑图

图 3-21 修改RS触发器

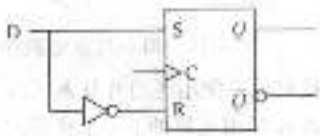
SR触发器的另外一种变化形式是D触发器（data，D flip-flop）。D触发器是一个真实的计算机存储器单元。D触发器可以存储一个信息位。如果将输入设置为1，那么当时钟脉冲到来时，D触发器的输出就变成了1；如果输入为0，则时钟脉冲会将输出也变成为0。因为输出Q表示电路的当前状态，所以输出值1表示当前“存储”了数值1。图3-22给出了D触发器的表示符号、特征表，以及利用SR触发器构成D触发器的逻辑图。



a) D触发器

D	$Q(t+1)$
0	0
1	1

b) D触发器的特征表



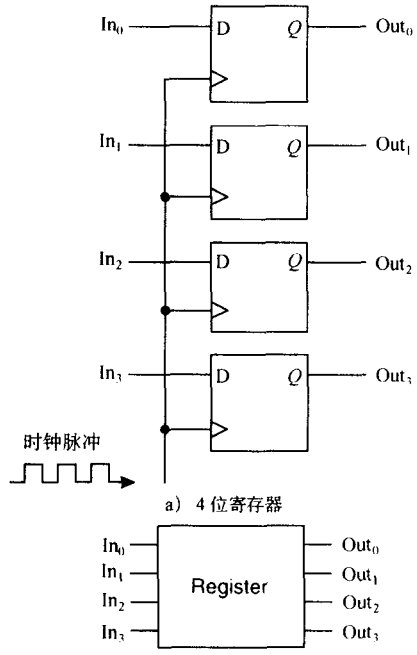
c) 利用SR触发器构成D触发器的逻辑图

图 3-22 D触发器的表示符号、特征表以及利用SR构成的逻辑图

3.6.4 典型的时序逻辑电路

利用锁存器和触发器可以实现许多复杂的时序逻辑电路。寄存器、计数器、存储器和移位寄存器都需要有存储功能，所以都采用时序逻辑电路来实现。

第一个例子是用 4 个 D 触发器来组成一个简单的 4 位寄存器。当然，要实现较大的字的寄存器，只需要简单地增加 D 触发器的数目即可。这个寄存器有 4 根输入线、4 根输出线和一根时钟信号线。时钟信号从定时的观点来说是非常重要的。因为寄存器不但要接收全部的新输入值，同时又要改变寄存器中各个存储单元的状态。对于同步时序电路，如果没有时钟脉冲的参与，电路的状态就不会发生改变。将 4 个相同的时钟信号连接到全部 4 个 D 触发器上，这样 4 个触发器可以同步变化。图 3-23 为这个 4 位寄存器的逻辑图和结构模块图。在真实电路中，还有电源线和接地线，以及一根清 0 线（即可以把整个寄存器的输出全部复位为 0）。但是，本书将把这些具体问题留给计算机工程师，而重点介绍这些电路的数字逻辑功能。



a) 4 位寄存器

b) 4 位寄存器的表示模块图

图 3-23 4 位寄存器的逻辑图和结构模块图

另外一个非常有用的时序电路是二进制计数器。二进制计数器会按照预先设置的状态，随着时钟脉冲发生序列变化。对于标准的二进制计数器，这些状态序列反映的是二进制数的序列。假设现在开始对二进制数进行计数：0000，0001，0010，0011，…，不难发现，随着数字的增加，较低位的每次变化都是对原来数字的取反操作。当计数器的某一位的状态从 1 变为 0 时，其相邻左边的位就会变成原来数字的反码。如果某个位右边所有的位都变为 1 后，下次变化就是使这个位的状态将从 0 变成 1。因为有对状态进行取反的要求，所以二进制计数器最好用 JK 触发器来实现。当 J 和 K 输入端都为 1 时，触发器的输出就是对当前状态的取反。在实际电路中，并是不直接将各独立的输入连接到对应的触发器，而是通过一根计数有效线（count enable line）连接到各个触发器。只有当计数有效线的设置为 1 时，计数器才可以对到来的每个时钟脉冲进行计数。如果有效线被设置为 0，即使有时钟脉冲出现，计数器也不会改变状态，即不会进行计数。读者可以仔细研究图 3-24 所示的计数器电路，按照不同的输入模式，跟踪电路的每个状态，了解计数器的输出如何从二进制数 0000 计数到 1111。还要特别留意，当计数器的当前状态为 1111 时，下一个时钟脉冲到来时电路将要进入的状态。

上面介绍的是一个简单的寄存器和一个计数器，下面要讨论一个非常简单的存储器电路。

图 3-25 表示一个可以存储 4 个 3 位字的存储器的逻辑图（通常称为 4×3 存储器）。图中电路的每一列代表一个 3 位的字。注意，对于各个字的每一位，触发器都是通过时钟控制来实现同步存储的。因此，存储器的每次读或写，都是读写一个完整的字。In_0、In_1 和 In_2 是用来存储，或者说写一个 3 位字到存储器的三根输入线。S_0 和 S_1 是两根地址线，用来决定存储器中的哪个字被选中。实际上，S_0 和 S_1 是一个 2-4 译码器的输入端，负责选择正确的存储器字。三根输出线（Out_0、Out_1 和 Out_3）用来从存储器中读出各个字。

电路还有另外一条控制线，写有效（write enable）控制线用来决定要进行的是读操作还是写操作。本电路中，输入线和输出线设计成分开连接的形式。但是在实际电路中，输入和输出都使用相同的线路。

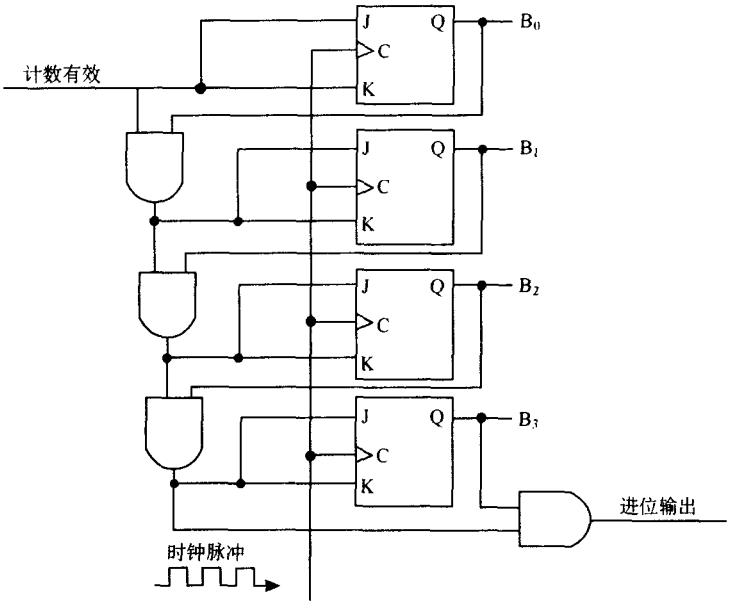


图 3-24 利用 JK 触发器构成的 4 位同步计数器

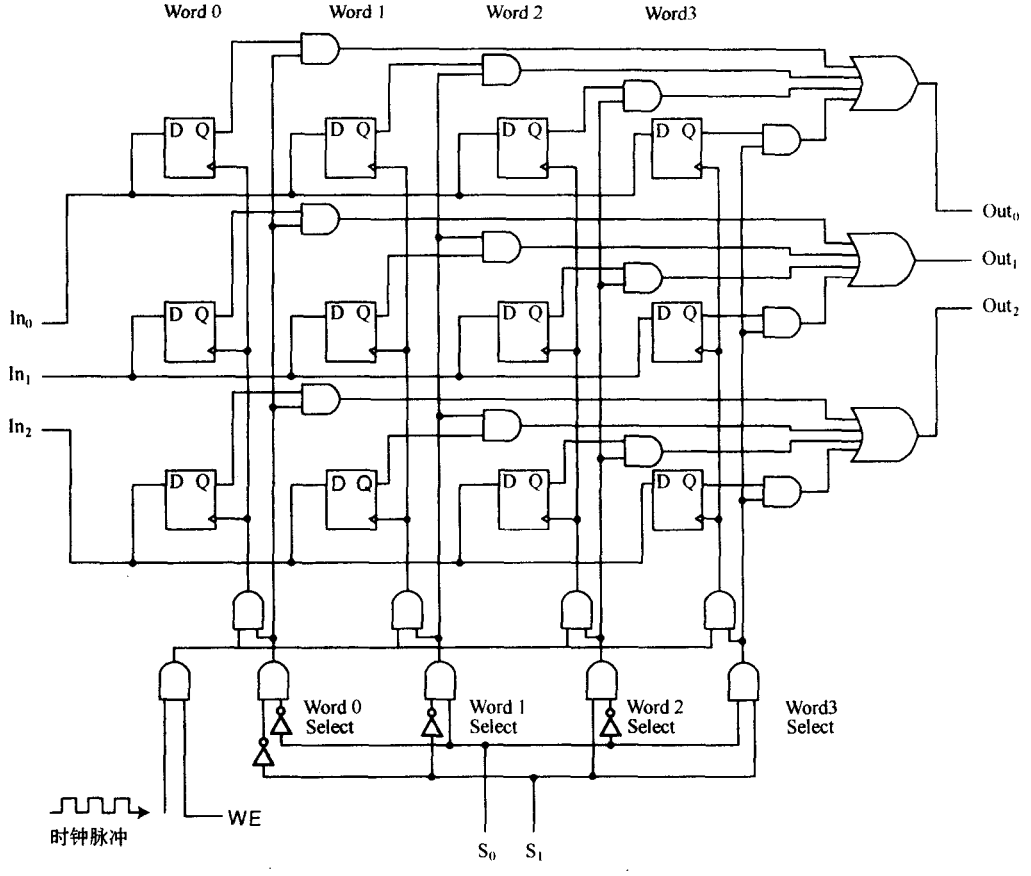


图 3-25 一个 4×3 的存储器逻辑图

总结上面有关存储器电路的讨论，将一个字写入到存储器需要执行下面几个关键步骤：

1. 在 S_0 和 S_1 上加地址信号。
2. WE（写有效）设置为高。
3. 译码器使用 S_0 和 S_1 地址信号，打开一个“与”门，在存储器中选择一个正确的字。
4. 在步骤 3 中选中的线路与时钟和 WE 信号一起选择一个字。
5. 对于选中的字，通过步骤 4 中已经打开的写入门电路加入时钟脉冲。
6. 在时钟脉冲的驱动下，输入线上的字被装入对应的三个 D 触发器中。

对于读取操作，也可以列出类似的执行步骤。这部分内容留给读者作为一个练习。还有一个有趣的练习留给读者，就是对该电路进行分析，看看如果将存储器从 4×3 存储器扩展为 8×3 存储器或者是 4×8 存储器，需要增加哪些额外的部件。

3.7 电路设计

在上面的内容中，引进了许多计算机系统中使用的各种电路部件。对此，本书不会提供读者足够的细节来开始进行计算机电路和系统的设计工作。数字逻辑电路设计不但需要熟悉数字逻辑电路，而且还要精通数字分析（digital analysis）—分析输出与输入之间的相互关系；数字综合（digital synthesis）—从真值表开始确定需要实现的各种功能的逻辑图，以及 CAD（计算机辅助设计）软件。从前面的讨论中大家可以知道，设计电路时必须非常仔细，以确保所设计的电路最小化。电路设计人员会面临许多问题，包括寻找最有效的布尔函数，尽量使用最少数量的门电路，采用一个不太昂贵的门电路组合，合理组织电路以安排满足最小占有空间和最小电能消耗的要求，尽可能在设计中采用标准化的功能模块等等。当然，还有许多目前尚未涉及到的问题，如信号的传输、信号的扇出、信号同步和外部接口等。不难发现，数字电路设计是一件非常复杂的工作。

至此，我们已经讨论了如何设计寄存器、计数器、存储器和各种其他数字电路结构模块。掌握了这些电路部件，电路设计人员可以利用硬件来实现各种计算机算法。读者一定还记得第 1 章中介绍的硬件和软件等效原理。编写一个计算机程序，也就是规定了一系列的布尔表达式。通常，编写一个程序要比设计一个实现该算法的硬件容易得多。但是，在许多实际应用中，利用硬件来实现的功能具有更好的性能。例如，在实时系统中，利用硬件实现比采用软件实现更快，肯定也会更好一些。当然，在某些情形时，利用软件的功能实现会更好一些。有时，人们会使用一个单一的可编程的微处理器芯片来取代一大堆数字电路部件。这样，就发展了所谓的嵌入式系统（embedded system）。家庭使用的微波炉和汽车很可能使用了嵌入式系统，其目的是为了取代某些可能会引发机械问题的附加硬件。编程嵌入式系统需要设计各种软件。通过这些软件，嵌入式系统可以读出各个输入变量，并且发送输出信号以执行各种任务，比方说开关电灯，让蜂鸣器发声，发出一个警报，或者打开一扇门。编写这些软件当然需要了解布尔函数的行为法则。

本章小结

本章的主要目的是介绍有关逻辑设计方面的基本知识，让读者对构建计算机系统的基本电路组合有一个全面的了解。本章的学习不是为了使读者具备设计这些基本电路部件的能力。而是帮助读者更好地理解后面将要介绍的有关体系结构的各种概念。

本章详细阐述了标准逻辑算符 AND、OR 和 NOT 的行为特性和实现这些算符功能的逻辑门电路。布尔函数可以用真值表来表示，而真值表可以转化为逻辑图，逻辑图则表示了实现逻辑功能所需的数字电路的各个部件和电路单元。因此，真值表提供了一种表示布尔函数及其逻辑电路特性的方法。在实际应用中，就是将这些简单的逻辑电路组合构成各种复杂的功能部件，如加法器、算术逻辑单元（ALU）、译码器、多路复用器、寄存器和存储器等。

布尔函数与其数字表示之间存在着一一对应关系。利用布尔恒等式可以化简布尔表达式，这样可以对组合数字电路和时序逻辑电路进行最小化处理，使电路变得最简单。最小化处理在电路设计中是非常重要

的内容。从芯片设计人员的角度来说，最重要的两个因素是速度和成本：电路的最小化设计不但可以降低电路成本，而且有助于提高电路的性能。

数字逻辑电路分为两类：组合逻辑电路和时序逻辑电路。组合逻辑电路，如加法器、译码器和多路复用器等，都是基于当前的输入直接产生输出结果。逻辑“与”门、“或”门和“非”门是组合逻辑电路的基本结构模块。当然，利用万能门，例如“与非”门和“或非”门，也能构建各种组合逻辑电路。时序逻辑电路，如寄存器、计数器和存储器等，所产生的输出不但与当前输入有关，而且取决于电路现在的状态。时序逻辑电路采用SR触发器、D触发器和JK触发器来构建。

所有这些逻辑电路都是建造计算机系统所必须的基本结构模块。第4章将把这些模块组合在一起，仔细研究计算机的工作原理。

本章练习题后面有一篇关于卡诺图专题的文章，有兴趣的读者可以参阅。

深入阅读

大部分有关计算机组成与体系结构的书籍都对数字逻辑和布尔代数有简短的讨论。Stallings（2000）、Patterson和Hennessy（1997）的著作包含了数字逻辑的精彩的概要论述。Mano（1993）对有关利用卡诺图进行电路简化（参见后面的专题论述），可编程逻辑器件，以及各种电路技术做了很好的讨论。如果需要深入了解数字逻辑的内容，可以阅读Wakerly（2000）、Katz（1994）或Hayes（1993）等人的书籍。

要学习布尔代数的运算技巧，可以参阅Gregg（1998）的著作。Maxfield（1995）的著作主要讨论了布尔逻辑的各种细节问题，具有一定深度，值得一读。要对各种门电路和触发器有直接和简单的了解（包括计算机的工作原理），可以阅读Petgold（1989）的书。Davidson（1979）则在其文章中叙述了一种基于“与非”门构建的各种数字电路的分解方法（因为“与非”门是一种万能门）。

如果读者对电路的实际设计有兴趣，可以使用一个非常好的免费仿真器。这套工具称为Chipmunk系统，能够应用于非常广泛的领域，包括电路仿真、图形编辑和曲线绘制。它有四个主要工具，但是对于电路仿真，Log就是所需的程序。利用Log程序中的Diglog部分可以生成和实际测试数字电路。读者可以从网上下载，在自己的计算机上运行。通用的Chipmunk程序发布在网站www.cs.berkeley.edu/~lazzaro/chipmunk/。这套仿真工具适合于许多计算机平台（包括PC和Unix机器）。

参考文献

Davidson, E. S. "An Algorithm for NAND Decomposition under Network Constraints," *IEEE Transactions on Computing:* C-18, 1098, 1979.

Gregg, John. *Ones and Zeros: Understanding Boolean Algebra, Digital Circuits, and the Logic of Sets.* New York: IEEE Press, 1998.

Hayes, J. P. *Digital Logic Design.* Reading, MA: Addison-Wesley, 1993.

Katz, R. H. *Contemporary Logic Design.* Redwood City, CA: Benjamin Cummings, 1994.

Mano, Morris M. *Computer System Architecture*, 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1993.

Maxfield, Clive. *Bebop to the Boolean Boogie.* Solana Beach, CA: High Text Publications, 1995.

Patterson, D. A. and Hennessy, J. L. *Computer Organization and Design, The Hardware/Software Interface*, 2nd ed. San Mateo, CA: Morgan Kaufmann, 1997.

Petgold, Charles. *Code: The Hidden Language of Computer Hardware and Software*, Redmond, WA: Microsoft Press, 1989.

Stallings, W. *Computer Organization and Architecture*, 5th ed. New York: Macmillan Publishing Company, 2000.

Tanenbaum, Andrew. *Structured Computer Organization*, 4th ed. Upper Saddle River, NJ: Prentice Hall, 1999.

Wakerly, J. F. *Digital Design Principles and Practices*, Upper Saddle River, NJ: Prentice Hall, 2000.

基本概念和术语复习

1. 为什么了解布尔代数对计算机科学家来说非常重要？
2. 哪个布尔操作被称为布尔积？
3. 哪个布尔操作被称为布尔和？
4. 分别建立布尔算符 OR、AND 和 NOT 的真值表。
5. 什么是布尔对偶原理？
6. 布尔表达式的最小化处理为什么对数字电路的设计至关重要？
7. 晶体管与逻辑门之间存在什么相互关系？
8. 列举 4 个基本逻辑门的名称。
9. 本章中介绍的是哪两个万能门？为什么这些万能门在数字设计中非常重要？
10. 描述数字逻辑芯片的基本结构。
11. 描述逐位进位加法器的工作原理。为什么现在大多数计算机不采用逐位进位加法器？
12. 通过几个输入及其对应值来选择某个特定的输出线的电路称为什么电路？举出这种电路元件的一个重要应用。
13. 什么样的电路可以从一组多输入线中选中二进制信息，并将其直接传送给某条单一的输出线？
14. 时序逻辑电路和组合逻辑电路有什么不同？
15. 构成时序逻辑电路的基本单元是什么？
16. 如果讲述一个时序逻辑电路是边缘触发而不是水平触发，意味着什么？
17. 什么是反馈？
18. JK 触发器与 SR 触发器有什么关系？
19. 为什么电路设计中通常优先选用 JK 触发器而不是 SR 触发器？
20. 哪种触发器可以实际代表一个计算机的存储器？

练习题

◆1. 构建下列函数的真值表：

◆a) $xyz+(\overline{x}\overline{y}\overline{z})$

◆b) $x(y\overline{z}+\overline{x}y)$

2. 构建下列函数的真值表：

a) $xyz+x\overline{yz}+\overline{x}\overline{y}\overline{z}$

b) $(x+y)(x+z)(\overline{x}+z)$

◆3. 如果函数 $F(x, y, z)=x(\overline{y}+z)$，利用德摩根律写出对函数 F 进行取反的反码表达式。

4. 如果函数 $F(x, y, z)=xy+\overline{x}z+y\overline{z}$，利用德摩根律写出对函数 F 进行取反的反码表达式。

◆5. 如果函数 $F(w, x, y, z)=xy\overline{z}(\overline{\overline{y}z+x})+(\overline{w}yz+\overline{x})$，利用德摩根律写出对函数 F 进行取反的反码表达式。

6. a) 运用布尔恒等式证明吸收律。

b) 证明德摩根律的正确性。

◆7. 证明下面的分配律的正确性：

$$x\text{XOR}(y\text{AND}z)=(x\text{XOR}y)\ \text{AND}\ (x\text{XOR}z)$$

8. 利用下列方法证明等式 $x=xy+x\overline{y}$：

a) 真值表

b) 布尔恒等式

9. 利用下列方法证明等式 $xz=(x+y)(x+\overline{y})(\overline{x}+z)$：

a) 真值表

◆b) 布尔恒等式

10. 使用布尔代数和布尔恒等式化简下列函数表达式，要求在化简过程的每一步列出相应的恒等式。

a) $F(x,y,z)=\bar{x}y\bar{z}+xy\bar{z}+xyz$

b) $F(w,x,y,z)=(xy+\bar{w}x)(w\bar{x}+y\bar{z})$

c) $F(x,y,z)=\overline{(x\bar{y})}\,\overline{(\bar{x}+\bar{y})}$

11. 使用布尔代数和布尔恒等式化简下列函数表达式，要求在化简过程的每一步列出相应的恒等式。

◆a) $\bar{x}yz+xz$

◆b) $(x+y)\overline{(x+\bar{y})}$

◆c) $\overline{\bar{x}\bar{z}+y}$

12. 使用布尔代数和布尔恒等式化简下列函数表达式，要求在化简过程的每一步列出相应的恒等式。

a) $(ab+c+df)\,ef$

b) $x+xy$

c) $(xy+xz)(w\bar{x}+y\bar{z})$

13. 使用布尔代数和布尔恒等式化简下列函数表达式，要求在化简过程的每一步列出相应的恒等式。

◆a) $xy+x\bar{y}$

b) $\bar{x}yz+xz$

c) $wx+w(xy+yz)$

14. 使用任意一种方法验证下列表达式的正误：

$yz+xyz+\bar{x}yz=xy+xz$

◆15. 利用布尔代数中的基本恒等式证明：

$x(x+y)=xy$

◆16. 利用布尔代数中的基本恒等式证明：

$x+\bar{x}y=x+y$

◆17. 使用布尔代数中的基本恒等式证明：

$xy+xz+yz=xy+\bar{x}z$

◆18. 下面为一个布尔表达式的真值表，按照积之和的形式写出布尔表达式。

x	y	z	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

19. 下面为一个布尔表达式的真值表，按照和之积的形式写出布尔表达式。

x	y	z	F
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

20. 画出以下函数的真值表，并按照两个积之和的取反形式重新改写布尔表达式。$x\bar{z}+\bar{y}z+\bar{x}y$

21. 已知函数 $F(x,y,z)=\bar{x}y+x y\bar{z}$，

◆a) 导出函数 F 的反码的代数表达式，采用积之和的形式。

b) 证明 $F\bar{F}=0$。

c) 证明 $F+\bar{F}=1$。

22. 已知函数 $F(x,y,z)=x\bar{y}z+\bar{x}\bar{y}z+xyz$

a) 列出函数 F 的真值表。

b) 画出原始布尔表达式的逻辑图。

c) 利用布尔代数和布尔恒等式化简布尔表达式。

d) 列出 c) 部分求得的化简表达式的真值表。

e) 画出 c) 部分求得的化简表达式的逻辑图。

23. 仅采用 AND、OR 门和 NOT 门构建 XOR 算符。

*24. 利用 NAND 门构建 XOR 门。

提示：$x\text{XOR}y=\overline{(\overline{x}y)\ (x\overline{y})}$

25. 设计一个电路。电路有 3 个输入（x，y，z），用来表示一个二进制数字的位数。电路还有 3 个输出（a，b，c）也表示一个二进制数的位数。当输入为 1、2 或 3 时，二进制输出应该比输入数值小 1。当二进制输入为 4、5 或 6 时，二进制输出应该比输入值大 1。当输入为 0 时，输出也是 0。而当输入等于 7 时，输出也等于 7。列出真值表，进行化简计算，并设计出最终电路。

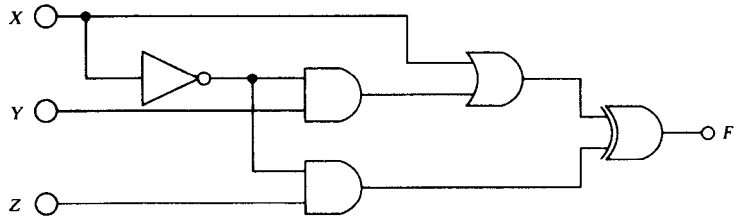
26. 画出一个组合逻辑电路，直接对下面的布尔表达式进行取反：

$F\ (x,\ y,\ z)=xz+\ (xy+\bar{z})$

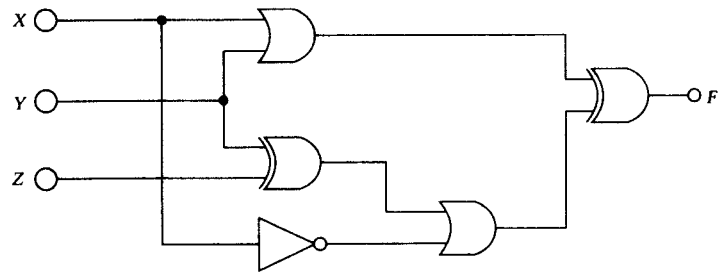
◆27. 画出一个组合逻辑电路，直接对下面的布尔表达式进行取反：

$F\ (x,\ y,\ z)=\ (xy\ \text{XOR}\ \overline{(y+\bar{z})})\ +\bar{x}z$

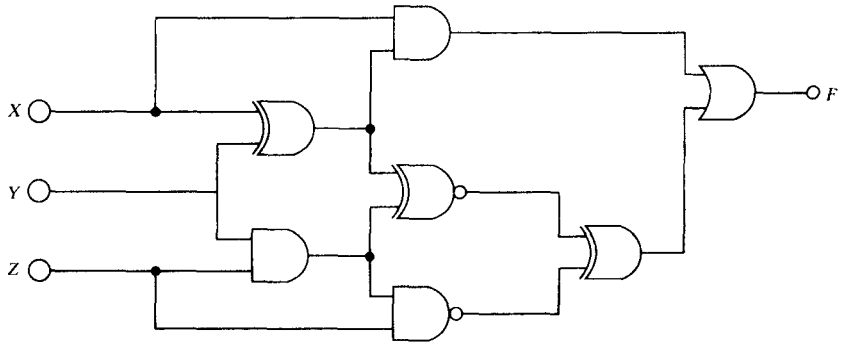
28. 求出描述下面电路的真值表：



◆29. 求出描述下面电路的真值表：



30. 求出描述下面电路的真值表：



31. 针对表 3-11 和 3-12，分别画出能够实现奇偶发生器和奇偶校验器的电路。
32. 画出仅使用 NAND 门组成的半加器。
33. 画出仅使用 NAND 门组成的全加器。
34. Tyrone Shoelaces 已经投入在股票市场大量的资金。他从不轻易相信别人告诉他的股票信息。每当他准备购买某种股票之前，一定会从三个不同的途径获取咨询。第一个信息来源是 Pain Webster，一个著名的股票交易所员。第二个信息来源是 Meg A. Cash，一个在股票市场自学成才的百万富翁。第三个要咨询的人是 Madame LaZora，一个世界闻名的具有心灵感应的大师。通过几个月来不断从这三种渠道获得忠告，他总结出以下的几点结论：

a) 如果 Pain 和 Meg 都建议购买，而只有心灵感应大师说不买时，就买进股票。

b) 如果心灵感应大师说可以购买时，也买进股票。

c) 而在其他情况下，不买股票。

对上面的结论，构建一个真值表，并求出对应的最小布尔函数。执行上述的逻辑过程告诉 Tyrone 先生何时可以购买股票。

◆ *35. 假如有一个规模很小的公司请你安装一套保安系统。这套保安系统是按照身份识别卡上的编码的位数来定价的。通过识别这种身份卡，持卡人可以出入公司的某些特定的场所。当然，这个小公司希望使用尽可能少的编码位数来满足公司所有的安全需求。这样可以花费尽可能少的经费。现在，首先要确定每张卡必需的编码位数。然后对每个保安场所的读卡器进行编程，以确保读卡器对身份卡进行扫描时可以做出恰当的响应。

该公司的雇员分为 4 类，公司有 5 处场所需要限制某些员工的出入。这些不同类型的人员和对他们的出入限制如下：

a) 大老板可以出入经理休息室和经理卫生间。

b) 大老板的秘书需要出入供应储藏室、员工休息室和经理休息室。

c) 计算机室的员工需求进入服务器房间和员工休息室。

d) 门卫有权进入工作场所的所有地点。

确定各类人员在身份卡上的编码方式，并分别为 5 个限制场所的读卡器构建相应的逻辑图。

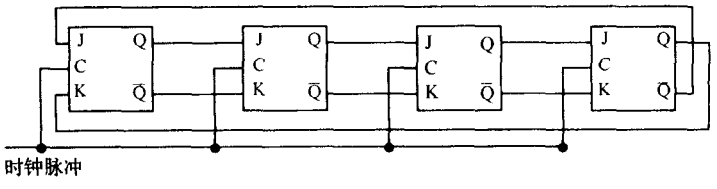
36. 构建一个容量为 4096 字节的存储器需要使用多少个 256×8 的 RAM 芯片？

a) 每个存储器需要有多少位地址？

b) 每个存储器芯片需要连接多少根地址线？

c) 存储器的片选输入需要多少根译码线？说明所需译码器的大小规格。

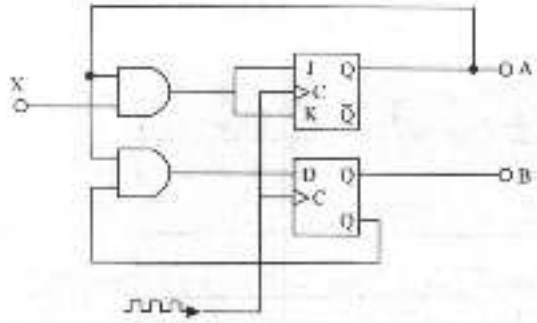
◆ *37. 研究分析下面电路的工作过程。假定电路的初始状态为 0000。按照时钟序列描绘电路的输出状态（Qs），并确定该电路的用途性质。答案要求描绘输出变化波形。



38. 描述下列电路的工作原理，并指出它们典型的输入和输出。同时，对每种电路画出仔细标注好的简单子示意图。

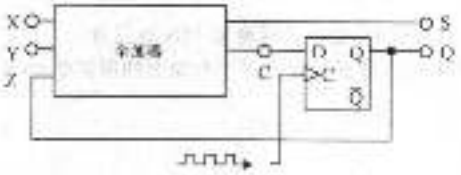
a) 译码器　　b) 多路复用器

◆39. 完成下列时序逻辑电路的真值表：



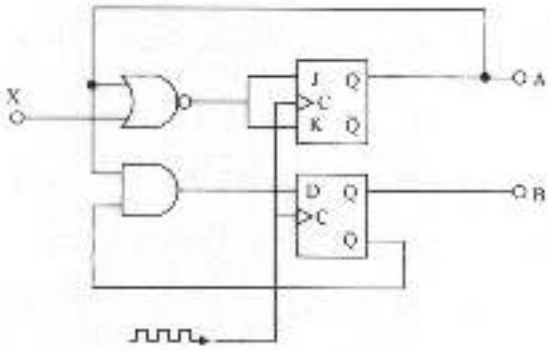
A	B	X	下一状态 A	下一状态 B
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

40. 完成下列时序逻辑电路的真值表：



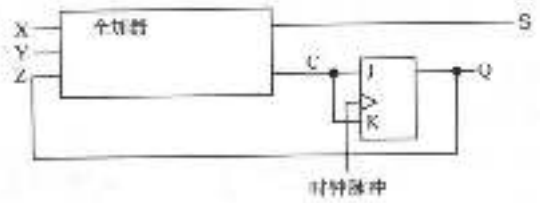
X	Y	Z	下一状态 S	下一状态 Q
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

◆41. 完成下列时序逻辑电路的真值表：



A	B	X	下一状态 A	下一状态 B
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

42. 一个时序逻辑电路包含一个触发器，两个输入线：X 和 Y，以及一个输出端：S。S 取决于下一状态。具体电路由一个全加器连接到一个 JK 触发器构成，如下图所示。完成该时序逻辑电路的特征表的最右边两列的填充：Next State（下一状态）和 Output（输出）栏。



目前状态 Q(t)	输入 X Y	下一状态 Q(t+1)	输出 S
0	0 0		
0	0 1		
0	1 0		
0	1 1		
1	0 0		
1	0 1		
1	1 0		
1	1 1		

43. 一个 Mex-Not 触发器（MN flip-flop）的工作特性如下：如果 M=1，触发器对当前的状态取反。如果 M=0，则触发器的下一个状态等于 N 的值。

a) 求出该触发器的特征表。

b) 如何通过增加门电路和反相器的方法，将一个 JK 触发器转换成 MN 触发器。

44. 列举从一个如图 3-25 所示的 4×3 存储器电路的存储器中读出一个存储字的所需的步骤。



卡诺图专题

3A.1 概述

本章主要讨论布尔表达式以及它们与数字电路的相互关系。对电路进行最小化处理有助于减少实际物理实现中所使用的电路元件的数目。使用较少元件的电路无疑会运行得更快。

化简布尔表达式可以采用布尔恒等式。但是，使用布尔恒等式来进行化简可能是一件十分困难的事情。原因是既没有一定的规则来指导怎样应用这些恒等式，也没有明确的化简步骤可循。布尔表达式的最小化处理，在某些方面就好像是做一道数学证明题。有时，即使方法正确，但是要完成化简任务还是一件十分耗时和麻烦的工作。在本附录中，将介绍一种化简布尔表达式的系统方法。

3A.2 卡诺图的描述和基本术语

卡诺图（Karnaugh Maps，或简写为 Kmaps）是表示布尔函数的一种图示法。卡诺图是对于给定的布尔表达式，将可能的各种输入值及其对应的布尔值值进行简单的列表。表格的行和列是各种可能的函数输入组合，而中间的单元表示对应于各种输入的函数输出值。

如果一个乘积项中只包括所有的变量一次，可以是变量的反码值，也可以是变量值的本身，那么这个乘积项称为小项（minterm）。例如，如果一个函数有两个输入值，x 和 y，就可以有 4 个小项，xy、$x\bar{y}$、$\bar{x}y$、和 $\bar{x}\bar{y}$，来表示相对于函数的所有可能的输入组合。如果是 3 个输入，x、y 和 z，总就有 8 个小项：xyz，$xy\bar{z}$，$x\bar{y}z$，$x\bar{y}\bar{z}$，$\bar{x}yz$，$\bar{x}y\bar{z}$，$\bar{x}\bar{y}z$ 和 $\bar{x}\bar{y}\bar{z}$。

作为一个例子，考虑布尔函数 $F(x, y) = xy + x\bar{y}$，x 和 y 的所有可能的输入值如图 3A-1 所示。

小项 xy 代表输入对（0，0）。类似地，小项 $x\bar{y}$ 表示（0，1），小项 $\bar{x}y$ 表示（1，0），而小项 $\bar{x}\bar{y}$ 表示(1，1)。

三变量的小项和它们代表的输入值，如图 3A-2 所示。

小项	x	y
xy	0	0
$x\bar{y}$	0	1
$\bar{x}y$	1	0
$\bar{x}\bar{y}$	1	1

图 3A-1 二变量的小项

小项	x	y	z
xyz	0	0	0
$xy\bar{z}$	0	0	1
$x\bar{y}z$	0	1	0
$x\bar{y}\bar{z}$	0	1	1
$\bar{x}yz$	1	0	0
$\bar{x}y\bar{z}$	1	0	1
$\bar{x}\bar{y}z$	1	1	0
$\bar{x}\bar{y}\bar{z}$	1	1	1

图 3A-2 三变量的小项

卡诺图是一个列表，每个小项在表格中占一个单元，也就是对于函数的真值表的每一行，卡诺图都有对应的一个单元，或称为卡诺图的元素。考虑函数 $F(x,y)=xy$ 和真值表，参见例 3A-1。

例 3A-1 $F(x,y)=xy$

x	y	xy
0	0	0
0	1	0
1	0	0
1	1	1

对应的卡诺图为：

x \ y	0	1
0	0	0
1	0	1

■

注意，当 $x=1$ 和 $y=1$ 时，卡诺图中只有一个元素的数值为 1，对应的就是 $xy=1$。下面来看另外一个例子，$F(x,y)=x+y$。

例 3A-2 $F(x,y)=x+y$

x	y	x+y
0	0	0
0	1	1
1	0	1
1	1	1

x \ y	0	1
0	0	1
1	1	1

■

例 3A-2 中有 3 个小项具有数值 1，正好对应于函数输出值为 1 的各种输入组合。要确定卡诺图中取值为 1 的单元，只需找到与真值表中取 1 所对应位置，将其设置为 1 即可。很明显，我们可以把函数 $F(x,y)=x+y$ 表示为所有取值为 1 的小项的“或”逻辑的形式。因此，函数 $F(x,y)$可以用表达式 $\bar{x}y+x\bar{y}+xy$ 来表示。显然，这个表达式并不是最简化的形式（因为这个函数本身就是 $x+y$）。现在利用布尔恒等式进行化简：

$$
\begin{aligned}
F(x,y)&=\bar{x}y+x\bar{y}+xy\\
&=\bar{x}y+xy+x\bar{y}+xy \qquad \text{（请记住：}xy+xy=xy\text{）}\\
&=y(\bar{x}+x)+x(\bar{y}+y)\\
&=y+x\\
&=x+y
\end{aligned}
$$

怎样才能知道需要添加额外的 xy 项呢？利用布尔恒等式进行代数化简是一件需要较高技巧的工作。当然，这也就是卡诺图的有用之处。

3A.3 利用卡诺图化简二变量函数

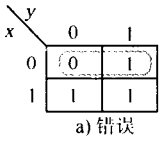
在上面的函数 $F(x,y)$的化简过程中，对表达式的各个项进行了分类组合，这样可以提取变量的公因子。化简过程中，增加了一项 xy 和 $\bar{x}y$ 进行组合。提出公因子 y，剩下 $\bar{x}+x=1$。但是，如果使用卡诺图进行化简，就不需要考虑增加什么项或者是采用哪一个布尔恒等式。因为这些都可以由卡诺图来完成。

下面再来看函数 $F(x,y)=x+y$ 的卡诺图，如图 3A-3 所示。

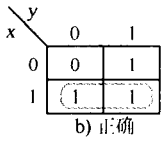
x \ y	0	1
0	0	1
1	1	1

图 3A-3 函数 $F(x,y)=x+y$ 的卡诺图

利用卡诺图化简布尔函数，只需简单地把取值为 1 的小项进行分类合并，本书简称为 1 项。这种分类合并的方法与采用布尔恒等式化简非常类似，但是必须遵守特定的法则。第一，只对取值为 1 的小项进行分组合并；第二，只对卡诺图中位于同一行或者是同一列的 1 项进行分组合并，而不考虑位于对角线上的 1 项。即需要合并的是相邻单元的 1 项。第三，分组合并的项数必须是 2 的指数幂。第四，分组中应该包含尽可能多的合并项。第五，要合并的所有的 1 项必须处在一个分组中（即使某些分组只有一个 1 项）。下面可以检验图 3A-4 到图 3A-7 中的一些正确和错误的分组合并。

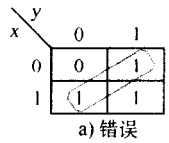


a) 错误

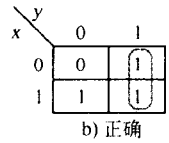


b) 正确

图 3A-4 只能对取值为 1 的小项进行分组合并

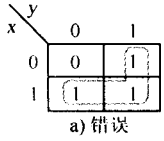


a) 错误

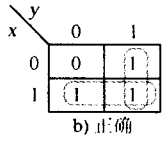


b) 正确

图 3A-5 不能对位于对角线的小项进行分组合并

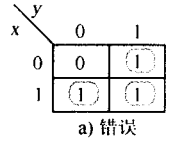


a) 错误

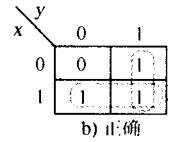


b) 正确

图 3A-6 合并的项数必须是 2 的指数幂



a) 错误



b) 正确

图 3A-7 分组合并的项数应该尽可能多

注意，图 3A-6b 和图 3A-7b 中的 3 个 1 项分别属于两个不同的分组。这种分组方法实际上等效于在布尔函数中增加一个 xy 项，类似于利用恒等式进行化简。卡诺图中的 xy 项将在化简过程中使用两次。

要利用卡诺图进行化简，首先要按照上述规则进行分组合并。完成全部分组后，仔细检查各个分组的小项，然后去掉每个分组中有变化的变量。例如，图 3A-7b 给出了函数 $F(x, y) = x + y$ 的正确分组。开始考虑分组中的第二行（$x = 1$），其中的两个小项分别是 $x\bar{y}$ 和 xy。这个分组表示这两项的逻辑或，即 $x\bar{y}+xy$。这两项中有变化的变量是 y，所以丢弃变量 y，只剩下变量 x。显然，利用布尔恒等式，可以得出相同的化简结果。而采用卡诺图可以直接去掉不需要的变量。第二个分组合并的结果为 $\bar{x}y+xy$，去掉有变化的变量 x，剩下变量 y。对第一组和第二组合并的结果取或逻辑，即为 $x+y$，这就是原始函数 F 的正确的化简形式。

3A.4 利用卡诺图化简三变量函数

卡诺图方法可以应用于多于两个变量的函数的情形。在本专题中，将介绍三变量函数和四变量函数的卡诺图。这种方法可以扩充到五变量或更多变量的情形。读者可以在本章的深入阅读部分所推荐的 Maxfield（1995）的著作中，获得有关卡诺图更详细的知识。

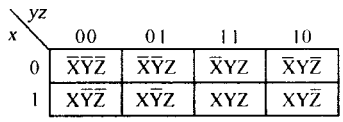


图 3A-8 三变量函数的小项和卡诺图形式

了解了怎样生成二变量函数的卡诺图，可以很方便地推广到三变量的函数，如图 3A-8 所示。

三变量卡诺图形式的第一个区别是，表中的两个变量 y 和 z，是组合在一起的。第二个区别是对列的编号并不是按照顺序进行的。图中列的标号顺序为 00，01，11，10，而不是正常的二进制数的大小排列次序：00，01，10，11。卡诺图的输入值必须按照这种方法进行排序，这样才能保证任意相邻的每个小项中只有一个变量互反。按照这种排列顺序，例如 01 后面为 11，对应的小项就是 $\bar{x}yz$ 和 $x\bar{y}z$，只有变量 y 互反。利用卡诺图化简需要去掉发生变化的变量，所以必须保证两个小项的组合中只有一个变量互反。

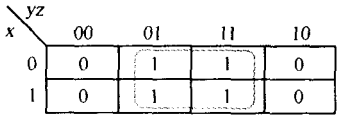
在前面的二变量函数的例子中所取的最大组合是两个取值为 1 的小项。当然，也可以有 4 个取值为 1 的小项组合，甚至可能大到 8 个 1 项的组合，这完全取决于函数本身。下面是几个利用卡诺图化简三变量函数的例子。

例 3A-3 $F(x,y,z)=\bar{x}\bar{y}z+\bar{x}yz+x\bar{y}z+xyz$

x \ yz	00	01	11	10
0	0	1	1	0
1	0	1	1	0

同样，要根据上面的分组原则来进行分组合并。很明显，可以有几种方法按两个小项为一组来进行分

组。但是，分组规则要求生成最大的分组，而且分组的大小必须是 2 的指数幂。因此，在本例中只能有一个 4 个 1 项组成的合并分组。分组方法如下：



x \ yz	00	01	11	10
0	0	1	1	0
1	0	1	1	0

■

完全没有必要再产生额外的两项组成的分组。分组越少，化简后函数的项数也越少。需要牢记的是，要化简表达式，必须保证所有的 1 项都处在各个分组之中。

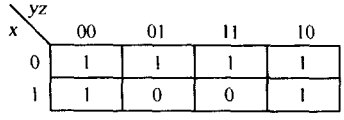
如果有一个 4 个 1 项的分组，如何进行化简呢？在分组中的两个数值为 1 的小项可以去掉一个变量，分组中 4 个数值为 1 的小项则可以去掉两个变量：即全部 4 项中都不同的两个变量。对于上例中的 4 单元分组，有 4 个小项：$\bar{x}\bar{y}z$、$\bar{x}yz$、$x\bar{y}z$ 和 xyz。这里，只有变量 z 是相同不变的，而变量 x 和 y 是不同的。所以要去掉变量 x 和 y，最后化简得到的函数为：$F(x, y) = z$。下面再利用布尔恒等式，来证明本例的化简结果的正确性。注意，这个函数最初就是表示为取值为 1 的小项的逻辑“或”的形式。

$$
\begin{aligned}
F(x,y,z) &= \bar{x}\bar{y}z+\bar{x}yz+x\bar{y}z+xyz \\
&= \bar{x}(\bar{y}z+yz)+x(\bar{y}z+yz) \\
&= (\bar{x}+x)(\bar{y}z+yz) \\
&= \bar{y}z+yz \\
&= (\bar{y}+y)z \\
&= z
\end{aligned}
$$

可见，利用布尔恒等式得到的结果和卡诺图化简的结果完全一致。

有时，分组的过程需要一些技巧。下面的例子就需要对卡诺图进行仔细的审查。

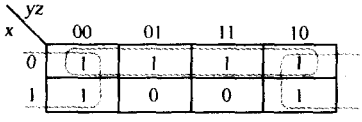
例 3A-4 $F(x,y,z)=\bar{x}\bar{y}\bar{z}+\bar{x}\bar{y}z+\bar{x}yz+\bar{x}y\bar{z}+x\bar{y}\bar{z}+xy\bar{z}$



x \ yz	00	01	11	10
0	1	1	1	1
1	1	0	0	1

这个问题需要一点技巧有两个理由：有些分组相互重叠，并且有一个分组是环绕连接在一起的（wraps around）。也就是第一列最左边的 1 与最后一列最右边的 1 可以组合为一组，因为第一列和最后一列在逻辑上是相邻的（可以想像是将卡诺图画在一个圆柱面上）。同样，卡诺图的第一行和最后一行在逻辑上也是相邻的，这一点在后面 4 变量卡诺图中会变得非常明显。

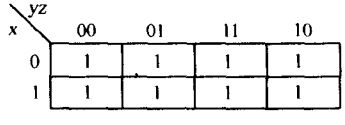
正确的合并分组为：



x \ yz	00	01	11	10
0	1	1	1	1
1	1	0	0	1

其中第一个分组化简的结果为 $\bar{x}$（4 个变量中只有一个是相同的），第二个分组化简的结果为 $\bar{z}$。因此，最终化简的结果为 $F(x,y,z)=\bar{x}+\bar{z}$。■

例 3A-5 下面考虑所有单元都是 1 项的卡诺图。如下面的卡诺图：



x \ yz	00	01	11	10
0	1	1	1	1
1	1	1	1	1

图中关于 1 项的最大分组为一个包含 8 个小项的组，即所有的 1 都分在同一组内。这样的问题如何进行化简呢？我们仍然可以遵守前面所用的相同法则。切记：包含 2 个小项的分组可以去掉一个变量，包含 4 个小项的分组可以去掉 2 个变量，而包含 8 个元素的分组可以去掉 3 个变量。其结果是去掉了所有的变量，最后得到的简化函数：$F(x,y,z)=1$。如果仔细检查这个函数的真值表，不难发现函数化简的结果的确是正确的。 ■

3A.5 利用卡诺图化简四变量函数

现在可以把卡诺图化简技巧推广到四变量函数的情形。4 个变量可以产生 16 个小项，如图 3A-9 所示。但是注意：对于卡诺图中的行和列都必须采用 11 后跟随 10 的特殊顺序排列。

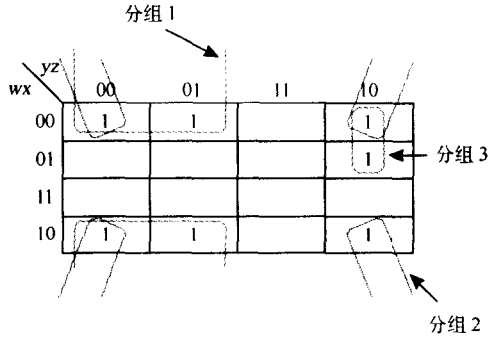
wx \ yz	00	01	11	10
00	$\overline{W}\overline{X}\overline{Y}\overline{Z}$	$\overline{W}\overline{X}\overline{Y}Z$	$\overline{W}\overline{X}YZ$	$\overline{W}\overline{X}Y\overline{Z}$
01	$\overline{W}X\overline{Y}\overline{Z}$	$\overline{W}X\overline{Y}Z$	$\overline{W}XYZ$	$\overline{W}XY\overline{Z}$
11	$WX\overline{Y}\overline{Z}$	$WX\overline{Y}Z$	$WXYZ$	$WXY\overline{Z}$
10	$W\overline{X}\overline{Y}\overline{Z}$	$W\overline{X}\overline{Y}Z$	$W\overline{X}YZ$	$W\overline{X}Y\overline{Z}$

图 3A-9 四变量的小项和卡诺图的形式

例 3A-6 为一个四变量函数的表示方法和化简过程。这里只画出元素为 1 的项，而略去 0 不写。

例 3A-6

$F(w,x,y,z)=\overline{w}\overline{x}\overline{y}\overline{z}+\overline{w}\overline{x}\overline{y}z+\overline{w}\overline{x}y\overline{z}+\overline{w}xy\overline{z}+w\overline{x}\overline{y}\overline{z}+w\overline{x}\overline{y}z+w\overline{x}y\overline{z}$



其中，分组 1 是一个如前面所示的“环绕（wrap-around）”分组。分组 3 是显而易见的组合。而分组 2 是一个极端环绕分组的情形：即它包括了卡诺图 4 个角上的元素 1。切记：卡诺图的 4 个角在逻辑上也是相邻的。最后，函数 F 被化简成 3 项，每个分组都只留下一项：$\overline{x}\overline{y}$（分组 1），$\overline{x}\overline{z}$（分组 2），和 $\overline{w}y\overline{z}$（分组 3）。所以，最后得到的化简结果为：$F(w,x,y,z)=\overline{x}\overline{y}+x\overline{z}+\overline{w}y\overline{z}$。 ■

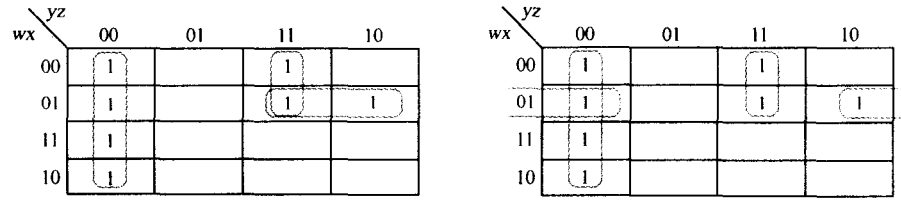
有时，在利用卡诺图进行化简时，可以有不同合并分组的选择。参见例 3A-7。

例 3A-7 卡诺图分组的不同选择

wx \ yz	00	01	11	10
00	1		1	
01	1		1	1
11	1			
10	1			

显然，第一列应该分为一组。同样，$\overline{w}\overline{x}yz$ 和 $\overline{w}xyz$ 的项也应该分为一组。但是，对于 $\overline{w}xy\overline{z}$ 的项的分

组却有不同的选择。它可以与 $\bar{w}xyz$ 的项分为一组，也可以与 $\bar{w}x\bar{y}\bar{z}$ 的项分为一组（一个环绕分组）。因此，有下面的两个分组方案：



其中，第一个分组方式得到的卡诺图化简结果为：$F(w,x,y,z)=F_1=\bar{y}\bar{z}+\bar{w}yz+\bar{w}xy$。第二个分组方式的卡诺图化简结果为：$F(w,x,y,z)=F_2=\bar{y}\bar{z}+\bar{w}yz+\bar{w}x\bar{z}$。这两个化简结果 F_1 和 F_2 的最后一项不同。但是，它们在逻辑上是等价的。这个问题的证明将留给读者，只需要生成这两个函数的真值表就可以检验 F_1 和 F_2 的等效性。这两个化简的结果有相同的项数和变量数目。根据化简的法则，卡诺图最小化处理可以产生最小化的函数（因而对应于一个最小的电路），但是这些最小化的函数的表示并不一定是唯一的。■

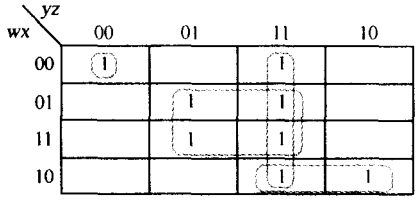
在介绍后面的内容之前，先对卡诺图化简方法的法则总结如下：

1. 只对包含 1 的单元进行分组合并，不必考虑包含 0 的单元。
2. 只有相邻的单元才能合并成一组，位于对角线上的单元不能进行合并。
3. 一个分组内包含 1 的单元的数目应该是 2 的指数幂。
4. 在满足化简法则的条件下，合并的分组应该尽可能大。
5. 全 1 元素的卡诺图的分组必须属于同一个分组，即使分组是单一的 1 项分组。
6. 分组时允许相互重叠。
7. 分组的元素可以环绕。
8. 尽可能地使用较少数目的分组。

使用上述法则，再讨论另外一个四变量函数的例子。例 3A-8 列举了上述法则的几种应用例子。

例 3A-8

$$F(w,x,y,z)=\bar{w}\bar{x}\bar{y}\bar{z}+\bar{w}\bar{x}yz+\bar{w}x\bar{y}z+\bar{w}xyz$$
$$+wx\bar{y}z+wxyz+w\bar{x}yz+w\bar{x}y\bar{z}$$



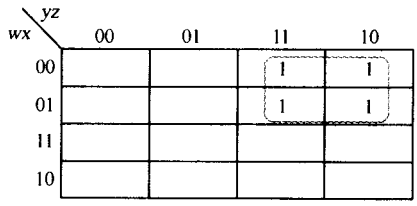
在本例中，有一个合并分组中只有一个单一元素。根据上面的法则，我们不可能将其他的元素与这一项合并。最后，上面卡诺图中所表示的函数化简为：$F(w,x,y,z)=yz+xz+\bar{w}\bar{x}\bar{y}\bar{z}+w\bar{x}y$。■

如果已知的函数不是表示为小项和的形式，我们仍然可以使用卡诺图对函数进行化简。但是在进行这类函数的化简之前，需要采用生成卡诺图的逆向过程来进行操作。例 3A-9 阐述了这种逆向操作过程。

例 3A-9　一个非小项和表示形式的函数

假设已知函数 $F(x,x,y,z)=\bar{w}xy+\bar{w}\bar{x}yz+\bar{w}\bar{x}y\bar{z}$。由于函数的最后两项是小项，我们很容易在卡诺图中对应位置找到 1 项的单元。但是，函数的第一项 $\bar{w}xy$ 为非小项。如果假定这一项是对一个卡诺图进行分组合并的结果，那么被去掉的变量就是 z 项。因此，这一项实际上等效于下面两项的和：$\bar{w}xy\bar{z}+\bar{w}xyz$。这样就可以在卡诺图中使用这两项来替代原来函数中的第一项，因为它们都是小项。

因此，得到了以下卡诺图：



最后，函数 $F(w,x,y,z)=\bar{w}xy+\bar{w}\bar{x}yz+\bar{w}\bar{x}y\bar{z}$ 化简为：$F(w,x,y,z)=\bar{w}y$。 ■

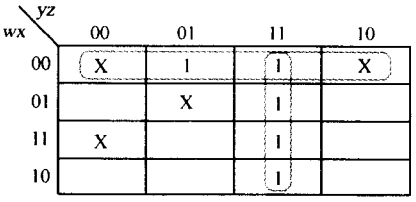
3A.6 无关条件

在某些情况下，不能完全指定一个函数。也就是说存在着某些输入，对于函数来说是没有被定义，或者是不明确的。例如，考虑一个 4 输入的函数，用作为从 0 到 10 进行计数的二进制的数位。这里采用位的组合：0000，0001，0010，0011，0100，0101，0110，0111，1000，1001，1010。但是，还有下面的一些输入组合：1011，1100，1101，1110，1111，没有被计数器所使用。因此，后面的输入组合是非法的。这就是说，如果从真值表来看，这些数值既不能是 0，也不能是 1。它们根本不应该在真值表中出现。

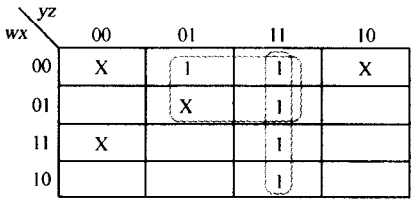
利用这些无关的（无影响的，don't care）输入可以帮助卡诺图化简过程。因为这些输入值可以根据需要把它们设置为 0 或 1，取决于是否最有帮助。基本的思想是以下面的方式设置这些“无关”的值：它们或者可以构成一个较大的合并分组，或者根本就不在分组中。例 3A-10 阐明的就是这样的概念。

例 3A-10　无关条件

通常，这些无关的值在对应的单元中用“X”来表示。下面的卡诺图将说明怎样使用这些值来帮助进行最小化处理。在第一行中我们将这些无关值当作 1 元素，从而可以构成一个 4 元素的合并分组。而在行 01 和 11 中将无关值当作 0 来处理。这样，函数简化成：$F_1(w,x,y,z)=\bar{w}\bar{x}+yz$。



还可以用另外的一种方式来对这些值进行合并分组：



采用上面的分组方法，最后的化简结果为：$F_2(w,x,y,z)=\bar{w}z+yz$。值得注意的是，这里的 F_1 和 F_2 不相等。但是，如果对两个函数生成真值表，不难发现，只有在输入无关的值的情况下，它们才会产生不相等。 ■

3A.7 小结

这部分内容简单介绍了卡诺图和利用卡诺图化简布尔函数的方法。一方面，直接使用布尔等式进行布尔函数的化简可能会很困难，并且很不方便。另一方面，卡诺图提供了一种寻找布尔函数最小表示的严格步骤。因此，利用卡诺图化简方法可以很方便地找到布尔函数表示所对应的最小电子线路。

练习题

1. 对于由下列卡诺图所定义的布尔函数，写出其化简后的表达式：

◆a)

x \ yz	00	01	11	10
0	0	1	1	0
1	1	0	0	1

◆b)

x \ yz	00	01	11	10
0	0	1	1	1
1	1	0	0	0

c)

x \ yz	00	01	11	10
0	1	1	1	0
1	1	1	1	1

2. 对于下面的布尔函数，生成其卡诺图和进行化简：

a) $F(x,y,z)=\bar{x}\bar{y}\bar{z}+\bar{x}yz+\bar{x}y\bar{z}$

b) $F(x,y,z)=\bar{x}\bar{y}\bar{z}+\bar{x}y\bar{z}+x\bar{y}\bar{z}+xy\bar{z}$

c) $F(x,y,z)=\bar{y}\bar{z}+\bar{y}z+xy\bar{z}$

3. 对于由下列卡诺图定义的布尔函数，写出其化简后的表达式：

a)

wx \ yz	00	01	11	10
00	1			1
01	1			1
11			1	
10	1		1	

b)

wx \ yz	00	01	11	10
00	1	1	1	1
01			1	1
11	1	1	1	1
10	1			1

c)

wx \ yz	00	01	11	10
00		1		1
01		1	1	1
11	1	1		
10	1	1		1

4. 首先生成对应的卡诺图，然后化简下列函数：

a) $F(w,x,y,z)=\bar{w}\bar{x}\bar{y}\bar{z}+\bar{w}\bar{x}y\bar{z}+\bar{w}x\bar{y}z+\bar{w}xyz+\bar{w}xy\bar{z}+w\bar{x}\bar{y}\bar{z}+w\bar{x}yz$

b) $F(w,x,y,z)=\bar{w}\bar{x}\bar{y}\bar{z}+\bar{w}\bar{x}\bar{y}z+w\bar{x}\bar{y}z+w\bar{x}y\bar{z}+w\bar{x}\bar{y}\bar{z}$

c) $F(w,x,y,z)=\bar{y}z+w\bar{y}+\bar{w}xy+\bar{w}\bar{x}y\bar{z}+w\bar{x}yz$

◆5. 对于下面的卡诺图，从代数上（利用布尔恒等式）证明这个包含 4 项的函数如何被化简为只有 1 项的函数。

x \ yz	00	01	11	10
0	0	1	1	0
1	0	1	1	0

6. 对于由下列卡诺图定义的布尔函数，写出其化简后的表达式：

◆a)

x \ yz	00	01	11	10
0	1	1	0	X
1	1	1	1	1

b)

wx \ yz	00	01	11	10
00	1	1	1	1
01		X	1	X
11			X	
10	1		X	1

当我们使用仪器工作时，自己千万不要将问题复杂化。
Leonardo da Vinci

第 4 章　MARIE：简单计算机模型

4.1　概述

今天，计算机设计是那些经过大量训练计算机工程师的工作。在类似本书这样的入门教材（教授计算机组成与体系结构的入门课程）中，对于设计和构建一台目前可以购买到的工作计算机的讲解不可能面面俱到。但是，本章将首先研究一种非常简单的计算机模型，称为 MARIE：一种非常直观和简单的计算机体系结构。然后，再简单介绍 Intel 和 MIP 结构的计算机系统：两种分别基于 CISC 和 RISC 设计思想的流行体系结构。本章的目的是向读者介绍计算机的工作原理。因此，遵从本书开始时引用的 Leonardo da Vinci 的建议，我们将尽可能简单介绍计算机体系结构。

4.1.1　CPU 的基本知识和组成原理

从第 2 章（数据表示方法）的讨论中我们了解到，计算机处理的必须是二进制编码的数据。同样，第 3 章的内容表明，存储器也是以二进制数的方式存储数据和程序指令。无论如何，计算机都必须正确地处理数据和执行程序。*中央处理器*（central processing unit，CPU）的任务就是负责提取程序指令，并对指令进行译码，然后按程序规定的顺序对正确的数据执行各种操作。要了解计算机的工作原理，首先必须熟悉计算机的组成部分，以及各组件之间的相互关系。为了学习下一节将要介绍的简单体系结构，我们首先从一般意义上来讨论在现代计算机控制级别的微观体系结构。

所有计算机都有中央处理器。中央处理器可以分成两部分。第一部分是*数据通道*（datapath），它是一种由存储单元（寄存器）和算术逻辑单元（对数据执行各种操作）所组成的网络。这些组件通过总线（总线是传递数据的电子线路）连接起来，并利用时钟来控制时间。CPU 的第二部分是*控制单元*（control unit），该模块负责对各种操作进行排序并保证各种正确的数据适时地出现在所需的地方。这两部分组合在一起，就可以完成 CPU 的各种工作任务：提取指令、指令译码和按规定的顺序执行各种操作。数据通道和控制单元的设计将直接影响到计算机的性能。因此，接下来的内容将详细介绍 CPU 的这些组成部分。

寄存器

计算机系统中使用寄存器来存储各种数据，例如：地址、程序计数器或者是程序执行所需要的数据。简单地说，*寄存器*（register）是一种存储二进制数据的硬件设备。寄存器位于处理器的内部，所以处理器可以非常快地访问寄存器中存储的各种信息。第 3 章中介绍了利用 D 触发器来构建寄存器。D 触发器等效于一个 1 位的寄存器，所以存储多位的数值需要一组 D 触发器的集合。例如，要构建一个 16 位的寄存器，需要将 16 个 D 触发器连接起来。从第 3 章的有关二进制计数器的内容中我们了解到，这些组成寄存器的触发器集合必须在时钟的控制之下统一同步工作。随着每个时钟脉冲的发生，各种数据可以输入到寄存器中。而且，这些被输入的数据在下一个时钟脉冲到来之前不会发生任何改变，也就是被存储在寄存器中。

计算机通常处理的是一些存储在寄存器中，具有固定大小的二进制字的数据。所以，大部分计算机配有特定大小的寄存器。通用的寄存器的大小有：16 位、32 位和 64 位。寄存器的数量一般会随着结构的不同而有所变化，通常是按照 2 的指数幂进行配置，其中 16 位和 32 位寄存器是最普遍的。寄存器可以存储数据、地址或控制信息。某些寄存器会被赋予“专门的用途”，或者只能用于存储数据，

或者只能用于存放地址，或者只用于存储各种控制信息。而其他一些寄存器则属于通用类型，可以在不同时刻分别保存数据、地址或控制信息。

可以向寄存器写入或从其读取信息，信息也可以在不同的寄存器之间传递。寄存器的编址方式与存储器不同。每一存储器字都有一个唯一的二进制地址，这些地址从 0 开始进行编码。寄存器则是由 CPU 内部的控制单元进行编址和处理的。

现代计算机系统中有各种不同类型的专用寄存器：存储信息的寄存器、进行数值移位的寄存器、进行数值比较的寄存器和计数寄存器。还有“中间结果”寄存器用来存储临时数据值，变址寄存器用来控制程序的循环操作，堆栈指针寄存器用于管理所处理的信息堆栈，状态寄存器用于保持各种工作状态或操作模式（比如溢出、进位或一些零条件等）。另外，通用寄存器是程序员可以访问的寄存器。大多数计算机都对寄存器进行分组，不同的组用于不同的特定目的。例如，在 Pentium 体系结构的 CPU 中，就有数据寄存器组和地址寄存器组。某些体系结构采用大量非常庞大的寄存器组，这些寄存器组可以使用一种全新的方式来加速指令的执行（这一部分内容将在第 9 章高级体系结构中再进行讨论）。

ALU

算术逻辑单元（arithmetic logic unit，ALU）在程序执行过程中用于进行逻辑运算（如比较运算）和算术运算（如加法和乘法运算）。第 3 章已经给出了一个简单的 ALU 的例子。一般情况下，ALU 有两个数据输入和一个数据输出。ALU 中的各种操作常常会影响*状态寄存器*（status register）的某些数据位的数值（设置这些数据位是为了指示某些动作，例如是否有溢出发生）。通过控制单元发出的信号，控制 ALU 执行各种规定的运算。

控制单元

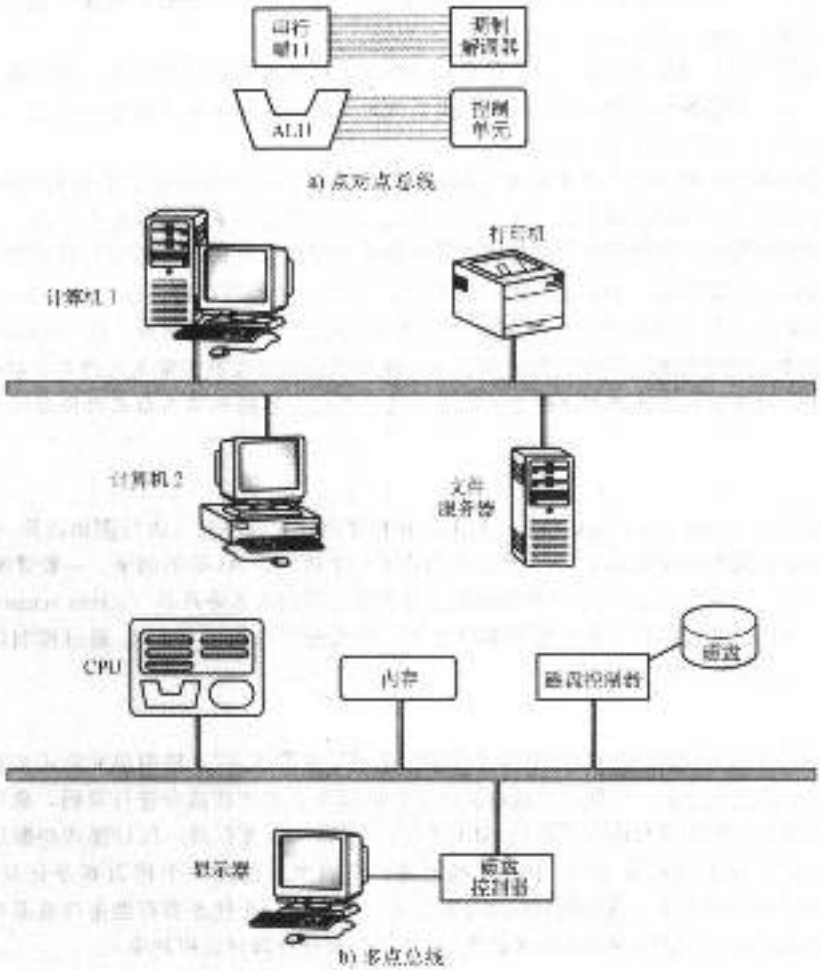
控制单元（control unit）是 CPU 中的“警察”或“交通管理员”。控制单元负责监视所有指令的执行和各种信息的传送过程。控制单元负责从内存提取指令，对这些指令进行译码，确保数据适时地出现在正确的地方。控制单元还负责通知 ALU 应该使用哪一个寄存器，执行哪些中断服务程序，以及对所需执行的各种操作接通 ALU 中的正确电路。控制单元使用一个称为*程序计数器*（program counter）的寄存器来寻找下一条要执行的指令的位置，并使用一个状态寄存器来存放某些特殊的操作状态，比如溢出、进位、借位和类似的状态等。4.7 节将更详细地讨论控制单元。

4.1.2 总线

CPU 通过总线与其他的部件连接起来。*总线*（bus）是一组导电线路的组合，它作为一个共享和公用的数据通道将系统内的各个子系统连接在一起。总线由多条线路构成，这样总线允许许多位数据并行传递。总线的制造成本低廉，用途广泛。利用总线可以很方便地进行新设备之间的互连，以及新设备与系统之间的连接。在任意时刻，只能有一个设备（可能是寄存器、ALU、内存或者是其他某个设备）使用总线。但是，这种共享总线的方式可能会引起通信上的瓶颈。总线的速度通常受总线长度和共享总线的设备数目的影响。在更通常的情况下，各种设备分为*主*（master）*设备*和*从*（slave）*设备*两种类型。*主设备*是最初启动的设备，而*从设备*是响应主设备请求的设备。

总线可以实现以*点对点*（point-to-point）的方式连接两个特定的设备（如图 4-1a 所示），或者将总线用作一条*公用通道*（common pathway）来连接多个设备。要求多个设备共享的总线称为*多点*（multipoint）总线，如图 4-1b 所示。

由于需要共享总线，所以总线协议（bus protocol，使用规则集）非常重要。典型的计算机总线包括数据总线、地址总线、控制总线和电源线，如图 4-2 所示。用于数据传递的总线称为*数据总线*（data bus），数据总线传递的是必须在计算机的不同位置之间移动的实际信息。计算机通过*控制总线*（control line）指示哪个设备允许使用总线，以及使用总线的目的（例如，是读还是写内存或 I/O 设备）。控制总线也传递有关总线请求、中断和时钟同步信号的响应信号。*地址总线*（address line）指出数据



a) 点对点总线

b) 多点总线

图 4-1 典型的计算机总线

读写（如内存中）的位置。电源线（power line）为计算机提供所需的电力。典型的总线事件包括传送读写地址信息，从内存传递数据到寄存器（称为内存读），以及从寄存器传递数据到内存（内存写）。另外，总线还用于从外围设备进行 I/O 设备的读写。各种信息的传递都发生在一个总线周期（bus cycle）内。总线周期是完成总线信息传送所需的时钟脉冲间的时间间隔。

由于总线传递的信息类型不同，使用总线的设备不同，所以总线还可以细分成各种不同的种类。处理器-内存总线（processor-memory bus）是长度较短的高速总线。这种总线将处理器和与机器匹配的内存系统最紧密地连接在一起，并最大限度地扩充这种总线的带宽（数据传递的带宽）。处理器-内存总线需要进行专门的设计。I/O 总线（I/O bus）通常比处理器-内存总线长，可以连接不同带宽的各种设备，这类总线可以兼容多种不同的体系结构。底板总线（backplane bus，或称为主板总线）实际上是构建在机器主板上的一条总线，如图 4-3 所示。利用这条主板总线可以将主板上的所有部件如处理器、I/O 设备和内存连接在一起，也就是所有的设备共享一条总线。许多计算机还采用总线分层结构，在同一系统中有两条（例如一条处理器-内存总线和一条 I/O 总线）或多条总线是很常见的。高性能的系统常常包含上述全部三类总线。

关于总线，个人计算机有自己专门的总线术语。PC 机使用一条内总线（称为系统总线，system bus）连接 CPU、内存和所有其他内部部件。而采用外总线（有时称为扩展总线，expansion bus）来

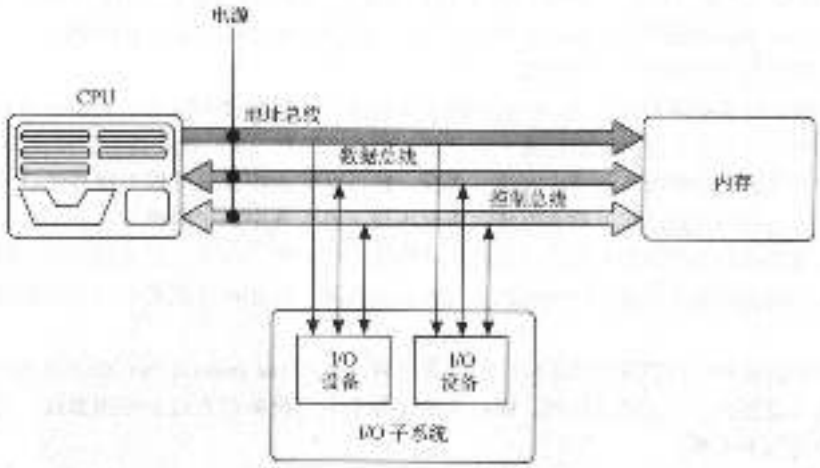


图 4-2 计算机的总线组成

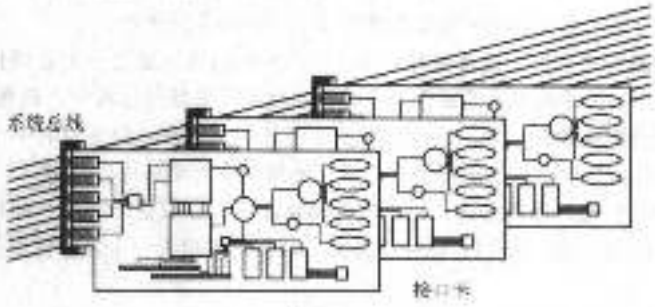


图 4-3 某种底板总线

连接各种外围设备，扩展插槽和I/O端口到计算机的其他部分。大多数个人计算机还有局部总线(local bus)，可以将外设直接连接到CPU的数据总线上。这些总线的速度很快，但只能连接数量有限、性质相近的设备。扩展总线的速度则要慢一些，但可以连接更广泛的设备。第7章会详细讨论这部分内容。

从物理结构上来说，总线只是一些导线的集合，但是，这些导线对于各种连接器、定时方式和信号特性有特定的标准，并且有严格的协议。同步（synchronous）总线是由时钟控制的，各种事件只有在时钟脉冲到来时才会发生，也就是事件发生的顺序由时钟脉冲来控制。每个设备都由时钟走动的速率，称为时钟频率（clock rate），来进行同步。前面提到的总线周期正好与总线的时钟频率成反比。例如，如果总线的时钟频率为133MHz，那么总线周期的时间长度就是1/133 000 000秒，或7.52ns。因为需要时钟来控制各种事件的发生，所以任何时钟脉冲产生的相位误差（clock skew）（或称为时钟脉冲的漂移）都可能带来一些严重的问题。这就意味着总线必须尽可能地保持短，这样时钟脉冲的发生漂移就不会变得太大。另外，总线周期时间不能短于信息在总线上传送所需要的时间。因此，总线的长度对总线的时钟频率和周期时间会有一定的限制。

各种控制线都是使用异步（asynchronous）总线来负责协调计算机的各种操作。这种异步总线必须采用一种较为复杂的握手协议（handshaking protocol）来强制实现与计算机其他操作的同步。例如，要从内存中读取一个数据字，协议就会要求有类似于如下的步骤：

1. ReqREAD：激活此总线控制线并同时将数据内存地址分配到适当总线上。
2. ReadyDATA：当内存系统已经把所需的数据送到总线数据线上时，这条控制线就会断言有效。

3. ACK：这条控制线用来指示 ReqREAD 控制信号或 ReadyDATA 控制信号已经作了应答。

使用协议而不是时钟脉冲来协调各种事件的发生，意味着异步总线可以在时间控制上与硬件技术配合得更好，而且可以支持更广泛的设备。

要使用总线，设备必须占用总线，任意时刻都只能有一个设备使用总线。正如前面所述，总线主控设备是允许最初的信息传递（控制总线）的设备，而总线受控（从）设备是由主控设备激活的模块。由于各种从设备是响应主控设备的请求来读写数据，所以只有主控设备可以占用总线以及为准备占用而保留总线。主从设备使用总线时都必须遵守通信协议，根据规定的定时要求协调工作。在某些非常简单的计算机系统（例如下面要介绍的一个简单计算机系统）中，常常只有处理器才是唯一的总线主控设备。这样做的优点是可以避免总线使用上的混乱，但缺点是处理器需要处理与使用总线有关的所有事件。

对于配备不止有一个主控设备的系统，则需要总线仲裁（bus arbitration）机制。总线仲裁机制是必须为某些主控设备设定一种优先级别，同时又保证各个主控设备都有机会使用总线。常用的总线仲裁方案可以分为以下 4 种：

1. **菊花链仲裁方式**：这种方案是通过使用一条“出让总线”（grant bus）的控制线，将总线使用权依次从最高优先级别向最低优先级别传递。这种方案不能保证仲裁的公平性，尤其是低优先级别的设备可能永远没有机会使用总线。这种方案非常简单，但是缺乏公平性。

2. **集中式平行仲裁方式**：每个设备都有一个总线请求控制线，通过一个总线仲裁器来选择使用总线的设备。采用这种类型的仲裁方式的缺点是可能会导致总线的使用过程中出现瓶颈效应。

3. **采用自选择的分配式仲裁方式**：这种方案类似于集中式仲裁，但是没有一个中央仲裁器来选择设备使用总线。而是由设备自己来决定哪个设备具有使用总线的最高优先级别。

4. **采用冲突检测的分配式仲裁方式**：每个设备都允许发出总线使用请求。如果总线检测到有任何冲突（即出现多个设备同时发出请求），这些设备就必须重新发出另一个总线使用请求。以太网络就使用这种仲裁方式。

第 7 章中将更详细地介绍关于总线和总线协议的内容。

4.1.3 时钟

每台计算机都有一个内部时钟，用来控制指令的执行速度。时钟也用来对系统中各个部件的操作进行协调和同步。在进行时钟脉冲时，系统时钟会为系统中所发生的各种事件设置步调，就像一个节拍器或交响乐的指挥一样。CPU 使用时钟信号来调控系统的各个进程，检测系统中的各种数字逻辑门电路是否会出现其他不可预测的执行速度。CPU 的每条指令执行都是使用固定的时钟脉冲数目。因此，计算机通常采用时钟周期（clock cycle）数目来量度系统指令的性能，而不是采用时间秒。时钟周期是时钟两次相邻嘀嗒声，即两个相邻时钟脉冲之间的时间间隔。时钟频率（clock frequency），有时称为时钟速率或时钟速度，是采用 MHz 来测量的。如第 1 章所述，1MHz 等于每秒 100 万周期，1Hz 表示每秒 1 个时钟周期。时钟周期时间（clock cycle time），或称为时钟周期（clock period），就是时钟频率的倒数。例如，一台标示 800MHz 的计算机的时钟周期时间为 1/800，000，000 秒，或者说 1.25ns。如果机器的周期时间是 2ns，则表示是一台 500MHz 的机器。

大多数计算机系统都是同步计算机：计算机只有一个主控时钟信号，主控时钟按照规定的时间间隔发生脉动，即时钟脉冲发生从 0 到 1 然后从 1 到 0 的规律性变化。计算机中的各个寄存器都必须等待时钟脉冲发生跃变才能输入新的数据。这样看起来，似乎只要提高时钟的速度，计算机就可以运行得更快。但实际上，在设计时钟周期时存在许多限制。当时钟脉冲发生跃变时，各个寄存器中会被装入新数据，同时寄存器的输出结果也可能发生改变。这些发生变化的输出值必须通过系统中的电路进行传递，直到下一组寄存器的输入端，然后存储于输入端。因此，时钟的周期时间必须足够长，以保证这些改变可以被传递到下一组的寄存器。如果时钟的周期时间太短，可能有某些变化的数值还未来

得及到达下一组寄存器，就中止了传递过程。这样将导致系统出现前后矛盾的状态，必须避免这种情况。因此，最小的时钟周期时间至少应该大于数据从每组寄存器的输出到下一组寄存器的输入所需要的传递时间，即电路的最大传输延迟时间。现在的问题在于，我们是否可以通过缩短寄存器之间的距离来减小这种传输的延迟呢？事实上，可以通过在输出寄存器和对应的输入寄存器之间增加寄存器的方法来减小传输延迟。注意，由于没有时钟脉冲的跃变，寄存器不会改变其输出值，所以增加额外的寄存器，事实上等于增加了时钟周期的数目。例如，原来需要 2 个时钟周期执行的一条指令，现在可能会需要 3 个或 4 个时钟周期（也可能需要更多的时钟周期，这取决于额外的寄存器的具体放置位置）。

大部分计算机指令的执行都需要 1 或 2 个时钟周期，但是某些指令则可能需要 35 个时钟周期或者更多。下面的公式表示的是 CPU 的时钟周期和时间秒的相互关系。

$$\text{CPU 时间}=\frac{\text{秒}}{\text{程序}}=\frac{\text{指令}}{\text{程序}}\times\frac{\text{平均周期}}{\text{指令}}\times\frac{\text{秒}}{\text{周期}}$$

注意，机器的体系结构对计算机的性能有很大影响。两台具有相同时钟速度的计算机并不表示执行指令时会使用相同数目的时钟周期。例如，乘法操作在早期的 Intel 286 机器上需要 20 个时钟周期，但在新款的 Pentium 机器上，只需 1 个时钟周期。这就是说，即使内部的系统时钟速度相同，新款的 Pentium 机器也要比老式的 286 机器快 20 倍。一般来说，乘法操作比加法操作需要更多的时钟周期，浮点运算比整数运算需要更多的时钟周期，而访问存储器要比访问寄存器需要更多的时钟周期。

通常我们所说的**时钟**（clock）是指**系统时钟**（system clock），即控制 CPU 和其他部件的主控时钟。然而，某些总线结构也配有自己的时钟。**总线时钟**（bus clock）通常比 CPU 时钟慢，这样就造成了系统的瓶颈问题。

系统的各个部件都有自己明确的性能界定范围，这些界定会标示出实现这些部件的功能所需要的最短时间。计算机零部件的制造商都可以保证他们生产的部件可以在给定的技术范围内的最极端条件下正常工作。如果把计算机的各个部件按某种设计顺序连接起来，这种顺序安排会要求某个部件必须在另一个部件的动作前完成其操作，这样整个计算机才能正常工作。因此，各个部件本身的性能界限对于计算机中各部件的同步工作是非常重要的。但是，许多人常常会超越这些技术限制，以增强系统的性能。例如，**超频**（over-clocking）运行就是人们为达到这种目的经常采用的一种方法。

虽然许多部件都可以超频运行，但是 CPU 是最流行的超频组件。CPU 超频的基本思想是让 CPU 运行在制造商规定给出的时钟频率或总线速度的上限之外。尽管这样可以增强系统的工作性能，但是操作时需要特别小心，以避免造成定时同步方面的故障，或者更糟糕的是使 CPU 过热。系统总线同样也可以超频，在系统的各部件之间通过总线进行通信时超频运行。对系统总线的超频运行可以大幅度改善系统的性能，但是同样也可能损害与总线相连的各种部件，或者可能会造成这些部件的工作不稳定。

4.1.4 输入/输出子系统

人们通过**输入和输出设备**（input and output device）与计算机进行交流。输入/输出是指各种外围设备和主存储器（也称为内存）之间的数据交换。通过输入设备，例如键盘、鼠标、读卡机、扫描仪、声音识别系统和触摸屏等，可以将数据输入计算机系统。而输出设备，例如显示器、打印机、绘图仪和扬声器等，则用来从计算机中获取信息。

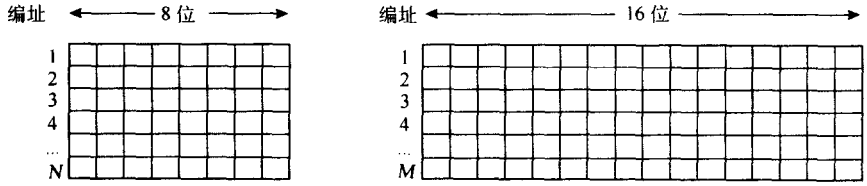
输入输出设备通常并不与 CPU 直接相连，而是采用某种**接口**（interface）来处理数据交换。接口负责系统总线和各外围设备之间的信号转换，将信号变成总线和外设都可以接受的形式。CPU 通过输入输出寄存器和外设进行交流，这种数据的交换通常有两种工作方式。在**内存交换（映射）的输入输出**（memory-mapped I/O）方式中，接口中的寄存器地址就在内存地址的分配表（或称为映射图）中。在这种方式下，CPU 对 I/O 设备的访问和 CPU 对内存的访问是完全相同的。很显然，这种输入输出方式的转换速度很快，但是却需要占用系统的内存空间。而对于**指令实现的输入输出**（instruction-based I/O）方式，CPU 需要有专用的指令来实现输入和输出操作。尽管这种工作方式不占用内存空间，但需要一些特殊的 I/O 指令，也就是只有那些可以执行这些特殊指令的 CPU 才可以使用这种输入

输出方式。中断在I/O中扮演了非常重要的角色，因为中断是一种非常有效的方法，可以通知CPU输入或输出是否已经准备就绪。

4.1.5 存储器组成和寻址方式

第3章中列举了一个规模非常小的存储器的例子，但是并没有讨论存储器组成与编址方式。显然，很好地理解这些概念对进一步学习计算机的知识非常重要。

我们可以将存储器设想成一个数据位的阵列。阵列的每一行的位长度通常等于机器的字大小。在物理上，我们利用一个寄存器来实现存储器阵列的一行的数据存储。每个寄存器（通常称为*存储单元*，memory location）都具有一个唯一的地址编号。存储器的地址通常是从0开始编号，按顺序递增。图4-4说明的是存储器的编址方式的概念。



a) *N*个8位存储器单元　　b) *M*个16位存储器单元

图4-4 存储器的编址方式

存储器的地址几乎都是采用无符号整数来表示。第2章讲过：一个4位二进制数称为一个半字节，而一个8位二进制数称为一个字节。正常情况下，存储器采用的是*按字节编址*（byte-addressable）的方式，也就是说每个字节都有一个唯一的地址。有些机器的字的大小可能会超过一个字节。例如，某台计算机能够处理32位字，即计算机可以通过不同的指令一次处理32位的二进制数。但是，这种计算机可能仍然使用的是按字节编址方式的体系结构。在按字节编址的情况下，如果一个机器字使用了多个字节，那么计算机将使用最低地址的字节来决定整个字的地址。当然，计算机的存储器也可以采用*按字编址*（word-addressable）的方式，即每个机器字（不是每个字节）都具有一个自己的唯一地址。但是，现在大部分流行的计算机都是采用按字节编址的方式，尽管这些机器使用的是32位或者更大的字。通常，一个存储器的地址存储在一个单一机器字中。

如果这里对具有不同大小字的计算机仍然采用字节编址方式的讨论使读者产生混淆，那么下面我们打个比方可能会有助于对这些概念的理解。存储器就好像是一条排满了公寓大楼的街道。每栋公寓大楼（对应于计算机中的字）都配备若干套的公寓房间（相当于字节），每套公寓房间都有自己的地址编号。所有这些公寓大楼都按照顺序来编号（或编址），从0开始编号直到全部公寓大楼的总数目。这些公寓大楼又会对各个房间进行分组编号。计算机中的字实际上起相同的作用。计算机的字是各种计算机指令所使用的基本单位。例如，即使在一台按字节编址的计算机上，我们也可以从内存读出一个字或者将一个字写入存储器。

如果计算机系统采用按字节编址的体系结构，并且指令系统的结构字大于一个字节，就会出现一个所谓*字节对齐*（alignment）的问题。例如，如果要从一个按字节编址的机器中读出一个32-位字，必须首先确定下面的两点内容：(1) 这个字是按照某种自然对齐的边界线来存储的；(2) 应该从这个对齐边界线开始访问。对于32-位字的情形，可以通过将要查找的地址乘以4倍来实现上面这种读取方式。有些机器允许以非对齐的方式进行存储器访问。在这种情况下，访问所查询的地址时无需从某个自然对齐的边界线开始。

存储器（指内存）由随机访问存储器（RAM）芯片构成，第6章将会详细讨论存储器。存储器通常使用符号L×W（长度×宽度）来表示。例如，4M×16的存储器表示存储器有4M长（具有$4M=2^2\times2^{20}=2^{22}$个字）和16位宽（每个字都是16位）。存储器的宽度（即上面表示法中的第2个数W）表示字的大小。要对这个存储器进行编址（假定按字进行编址），需要有能够唯一区分2^{22}个不同

内容的单，即需要 2^{32} 个不同的地址。因为地址采用从零开始的二进制数，所以要利用二进制数从 0 计数到 $(2^{32}-1)$。但是，这样编址方法需要使用多少位的二进制数呢？如果从 0 计数到 3（总共 4 项）需要 2 位二进制数。如果从 0 计数到 7（总共 8 项），需要 4 位二进制数。依照此规律，读者可以自己填写表 4.1 中的空格。

表 4.1 计算所求的编址位

总项	2	4	8	16	32
二次幂	2^1	2^2	2^3	2^4	2^5
位数	1	2	3	4	??

表中正确的答案应该是 5 位。如果计算机有 2^N 个需要编址的内存单位，则需要使用 N 位二进制数来对每个字节进行无重复的唯一编址。

主存储器通常使用的 RAM 芯片数目大于 1。通常利用多块芯片来构成一个满足一定大小要求的单一存储器模块。例如，利用 2K × 8 RAM 的芯片构建一个 32K × 16 的存储器，可以按照图 4.5 的方式将芯片连接成 16 行和 2 列的形式。



图 4-5 利用 RAM 芯片组合构成存储器

组合芯片的每一行可以编址 2K 字（假定机器是按字编址的），但需要用两个芯片来处理一个字的全部宽度（16 位）。对这个存储器的编址必须使用 15 位二进制数（需要访问的字的数目为 32K = $2^5 \times 2^{10}$）。每对芯片（即组合存储器的每一行）的内部只需要用 11 根地址线（每个芯片只能保持 2^{11} 位字）。在这种情况下，需要使用一个译码器对地址最左侧的 4 位地址进行译码，来决定是哪个芯片对存放了所需的地址。一旦选定了芯片对，剩余的 11 位地址将被送到另外一个译码器，以确定该芯片对中的正确的地址单元。

单一共享存储器模块可能会引起存储器访问上的顺序问题。存储器交叉存储技术（memory interleaving），即从多个存储器模块（或称为存储器组）中分离存储器单元的技术，有助于解决这个问题。交叉存储技术分为：低位交叉存储（low-order interleaving），使用地址的低位来选择存储器组；高位交叉存储（high-order interleaving），使用地址的高位来选择存储器组。

高位交叉存储是一种更直观的存储器组成结构。利用高位交叉存储技术，可以直接将地址分配给含有连续地址的存储器模块。具有 32 个地址的高位交叉存储方式如图 4.6 所示。



图 4.6 高位存储器的交叉存储

低位交叉存储器则将连续的存储器地址分配到不同的存储器模块中。图 4.7 给出了一个具有 32 个地址的低位交叉存储方式的示意图。

使用低位交叉存储方式并配合相应的总线技术，可以在一个存储器模块的读写操作完成前，就开始另一个存储器模块的读写操作。也就是存储器模块之间的读写操作可以相互重叠。

上面所讨论的有关存储器的概念非常重要，这些概念将会在本书余下的章节，特别是详细讨论存储器的第 6 章中频繁出现。值得关注的重要概念有：（1）存储器采用无符号的二进制数值进行编址（而通常人们所见的却是十六进制的数值，原因是它比较容易记忆）；（2）要编址的项目数决定了出现

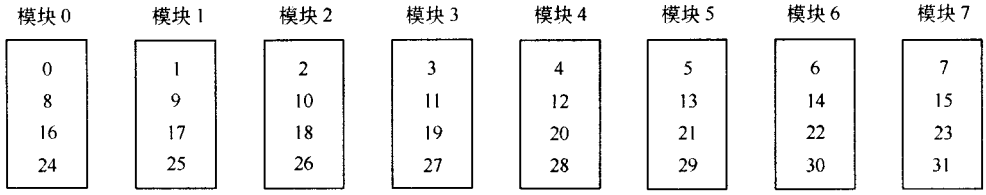


图 4-7 低位存储器的交叉存储

在地址中的二进制数的位数。当然，也可以采用比设计地址要求的更多的位数来进行编址。但实际上人们很少这样做，因为最小化原则是计算机设计中的一个重要概念。

4.1.6 中断

上面已经介绍了理解计算机体系结构所需要的基本硬件信息：CPU、总线、控制单元、寄存器、时钟、I/O 和存储器。但是，下面还有一个基本概念，这就是反映负责处理计算机中各个部件与处理器之间相互作用的概念：中断（interrupt）。中断就是改变（或中断）系统正常执行流程的各种事件。多种原因可以触发中断，其中包括：

- I/O 请求
- 出现算术错误（例如，被 0 除）
- 算术下溢或上溢
- 硬件故障（例如，存储器奇偶校验错误）
- 用户定义的中断点（比如，程序调试过程）
- 页面错误（第 6 章将详细讨论）
- 非法指令（通常由于指针问题引起）
- 其他原因

执行各种中断的操作称为中断处理，不同中断类型的中断处理方法各有不同。例如，通知 CPU 已经完成 I/O 请求的中断处理与处理被 0 除产生的中断过程是完全不同的。但是，这些操作都是由中断来处理的，因为这些过程都需要改变程序执行的正常流程。

由用户或系统发出（启动）的中断请求可以是屏蔽（maskable）中断（可以被禁止或忽略）或非屏蔽（nonmaskable）中断（高优先级别的中断，不能被禁止，必须响应）。中断可以出现在指令中或指令之间，可以是同步中断，即与程序的执行同步出现。也可以是异步中断，即随意产生中断。中断处理可能会导致程序的终止执行，或者当中断处理完毕后程序会继续执行。在 4.3.2 节和第 7 章中将详细讨论中断。

至此，已经对计算机系统的各个主要的功能部件作了一般性的介绍。下面通过一个简单的例子来说明功能、体系结构这些概念。

4.2 MARIE

MARIE，表示一个真正直观和简单的计算机体系结构。它包括存储器（存储程序和数据）和 CPU（由一个 ALU 和几个寄存器组成）。实际上，MARIE 包含一个实际的工作计算机所具备的全部功能部件。MARIE 模型将有助于说明本章和接下来的 3 章中所介绍的一些概念。下面小节介绍 MARIE 的体系结构。

4.2.1 体系结构

MARIE 具有下列特点：

- 使用二进制数和补码表示法
- 存储程序和采用字长度

- 按字（不是按字节的）编址的方式
- 主存储器容量为 4K 字（即每个地址需要使用 12 位二进制数）
- 16 位数据（16 位字）
- 16 位指令，4 位操作码和 12 位地址
- 一个 16 位的累加器（AC）
- 一个 16 位的指令寄存器（IR）
- 一个 16 位的存储器缓冲寄存器（MBR）
- 一个 12 位的程序计数器（PC）
- 一个 12 位的存储器地址寄存器（MAR）
- 一个 8 位的输入寄存器
- 一个 8 位的输出寄存器

图 4.8 为 MARIE 体系结构的示意图。

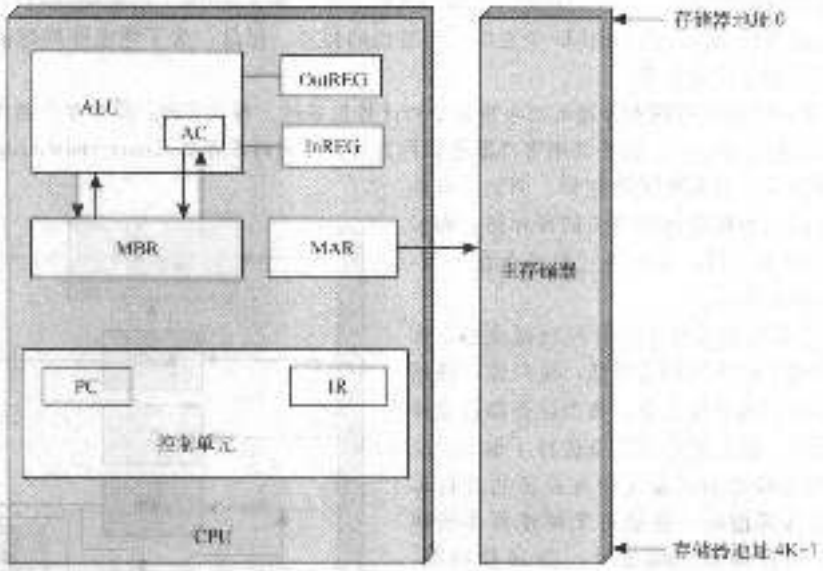


图 4.8 MARIE 体系结构

在讨论其他问题之前，首先强调存储器的一个重要观点。第 3 章介绍了一个利用 D 触发器构建的简单存储器。再次强调，存储器中的每一个存储单元都有一个唯一的地址，采用二进制数来表示，并且每个存储单元都可以存放一个数值。表示存储单元地址的二进制数需要很容易与存储单元中存放的内容（也是一个二进制数）混淆。为了避免这种概念上的混淆，可以想像一个有关邮局的例子。邮箱是按照不同的地址或号码进行编排的，邮箱里面放有邮件。如要取出邮件就需要知道邮箱的号码。从存储器中提取数据或指令同样需要知道存储单元的地址。通过指定存储单元的地址可以处理任意存储器地址中的内容。读者将会看到有多种不同的方式来获取存储单元的地址。

4.2.2 寄存器和总线

寄存器是 CPU 内部的一些存储单元（如图 4-8 所示）。CPU 中的 ALU（算术逻辑单元）部件负责执行所有进程（例如算术运算、逻辑判断等）。执行程序时，寄存器有着非常特殊的用途：即保持各种暂存的数值，如正在用某种方式处理的各种数据，或者是各种简单的计算结果。在许多情况下，计算机对寄存器的引用都隐含在指令之中，下面的第 4.2.3 节中，在讨论 MARIE 的指令集时，读者将会看到这种情况。

在 MARIE 中，有下列 7 种寄存器：

- AC：累加器（accumulator），用来保持数据值。它是一个通用寄存器（general purpose register），作用是保持 CPU 需要处理的数据。现在大部分的计算机都有多个通用寄存器。
- MAR：存储器地址寄存器（memory address register），用来保持被引用数据的存储器地址。
- MBR：存储器缓冲寄存器（memory buffer register），用来保持刚从存储器中读取或者将要写入存储器的数据。
- PC：程序计数器（program counter），用来保持程序将要执行的下一条指令的地址。
- IR：指令寄存器（instruction register），用来保持将要执行的下一条指令。
- InREG：输入寄存器（input register），用来保持来自输入设备的数据。
- OutREG：输出寄存器（output register），用来保持要输出到输出设备的数据。

MAR、MBR、PC 和 IR 寄存器用来保持一些专用信息，并且不能用作上述规定之外的其他目的。例如，我们不能将来自存储器的某个任意的数据值存放在 PC 寄存器中，而必须使用 MBR 或 AC 寄存器来存放这些任意的数据值。另外，还有一个状态或标志寄存器（status / flag register），用来保持指示各种状态或条件的信息，例如 ALU 中发生一个溢出的信息。但是，为了使主要问题清晰起见，下面所有图中均未包含该寄存器。

MARIE 是一个具有有限寄存器组的非常简单的计算机系统。现代 CPU 具有多个通用寄存器，用来执行类似 AC 的各种功能。这些通用寄存器通常称为用户可见的寄存器（user-visible register）。现代的计算机系统还有一些额外的寄存器。例如，有些计算机具有可以对数据值进行移位的寄存器；而另外还可能有一些寄存器，可以将它们组合在一起，用作各种数据值的列表。

MARIE 需要利用总线来将各种数据或指令传入和移出寄存器。在 MARIE 内部，我们假定使用的是一条公用总线的设计方案。每个设备都被连接到这条公用总线，并且对这些设备进行了编号。在各个设备使用总线之前，需先设置设备的身份编号。还需要设计其他的一些通道来加速程序的执行。在 MAR 和存储器之间设计一条通信路线。MAR 通过这条电路为存储器的地址线提供输入，指示 CPU 要读写的存储器单元的位置。另外，从 MBR 到 AC 之间有一条单独的数据通道，从 MBR 到 ALU 还有一条特殊的通道，允许 MBR 中的数据也可以应用在各种算术运算中。同时，各种信息也可以从 AC 流经 ALU，再流回 AC，而不需要经过公用总线。采用这三条额外通道的好处在于，在同一个时钟周期内，既可以将信息放置到公用总线，又可以同时将数据放置到这些额外的数据通道上。因此，MARIE 体系结构允许这些事件并行发生。图 4.9 给出了 MARIE 中的数据通路（即信息流动的通道）的示意图。

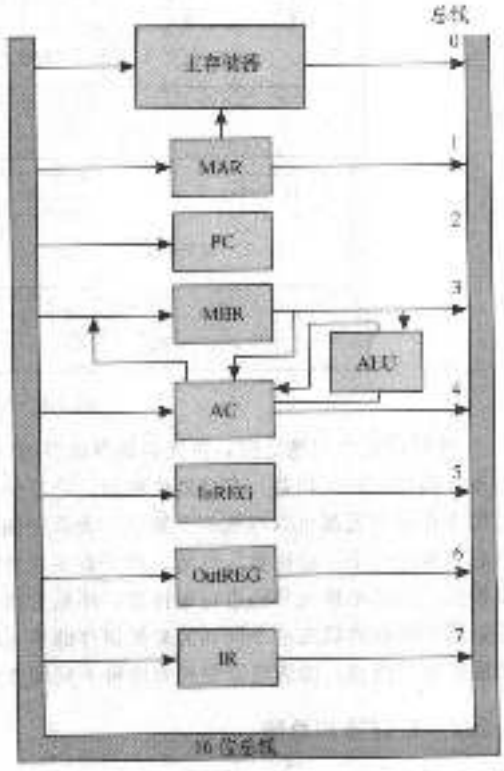


图 4.9 MARIE 中的数据通道

4.2.3 指令系统体系结构

MARIE 具有一个非常简单、但功能强大的指令系统（或称为指令集）。计算机的指令系统体系结

构（instruction set architecture，ISA）详细规定了计算机可以执行的每条指令及其格式。从本质上来说，ISA 是计算机软件和硬件之间的接口。典型计算机的指令系统体系结构需要包含几百条指令。正如前面所述，MARIE 的每条指令都由 16 位二进制数所构成。指令最左边的 4 位，从第 12 位到第 15 位，组成机器的操作码（opcode）。操作码表示的是要执行的指令，MARIE 总共可以有 16 条指令。而右边的 12 位，第 0 位到第 11 位，用来形成一个地址。因此，机器允许的最大存储器的大小为 $2^{12}-1$。MARIE 的指令格式如图 4-10 所示。



图 4-10 MARIE 的指令格式

大多数 ISA 由处理数据、移动数据和控制程序的执行序列的指令组成。表 4-2 给出了 MARIE 指令集中的各种指令。

表 4-2 MARIE 的指令集

指令编号		指令	意义
进制	十六进制		
0001	1	Load X	将地址为 X 的存储单元中的内容装入 AC
0010	2	Store X	将 AC 中的内容存储到地址为 X 的存储单元中
0011	3	Add X	将地址为 X 中内容和 AC 中的内容相加，然后将结果存入 AC 中
0100	4	Subt X	将 AC 中内容减去地址 X 中的内容，然后将结果存入 AC 中
0101	5	Input	从键盘输入一个数值到 AC 中
0110	6	Output	将 AC 中的数值输出到显示器
0111	7	Halt	终止程序的执行
1000	8	Skipcond	有条件地跳过下一条指令
1001	9	Jump X	将 X 的值装入到 PC 中

指令 Load 允许将数据从存储器（内存）装入 CPU。此操作需要通过 MBR 和 AC 两个寄存器来完成。来自内存的所有数据（即指令除外的其他任何信息）必须首先送到 MBR，然后再转送到 AC 或 ALU，这种体系结构并没有设计其他的数据传送方案。注意，在 Load 指令中没有指明 AC 为最终的目的操作数。因为 AC 实际上是隐含（implicit）在这条指令中。其他的指令也以类似的方式引用 AC。指令 Store 的作用是将数据从 CPU 送回到内存中。指令 Add 和 Subt 则执行加法和减法运算，分别将 AC 中的数据值加上或减去来自地址为 X 的存储器单元内的数据值。地址 X 处的数据会被复制到 MBR 中，并且保存在 MBR 中直到算术操作执行完成。利用 Input 和 Output 这两条指令可以实现 MARIE 和外界的通信交流。

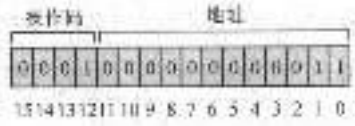
计算机的输入和输出是比较复杂的操作。现代的计算机都采用 ASCII 编码字符的方式进行输入和输出操作。这就是说，如果键盘输入数字 32，那么计算机实际上是先读 ASCII 字符 "3"，然后再读字符 "2"。这两个字符必须首先转换成数值 32，然后才能被存放在寄存器 AC 中。因为这里我们所关心的是计算机的工作原理，所以可以假定来自键盘的输入值都自动地进行了正确的转换。这种假定实际上掩盖了一个非常重要的观念，即如果输入或输出的信息实际上是一个 ASCII 字符，计算机如何识别

输入/输出的数值是 ASCII 字符还是数字？答案是，计算机是通过如何使用该数值的前后信息来识别的。在 MARIE 中，假设输入和输出的只有数字。输入的数值可以是十进制数，同样也假定有一个“神奇的转换”将输入的十进制数转换为计算机实际存储的二进制数。事实上，要使计算机能够正常运转还需要讨论一些问题。

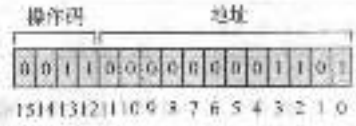
Halt 命令用于终止当前的程序执行。指令 Skipcond 可以实现一个条件分支转移（就像“While”循环语句或“if”语句所执行的分支转移一样）。执行 Skipcond 指令时，必须检查存储在 AC 中的数值。可以利用地址位中的两位来指定将要被测试的条件。这里假定选取的是最靠近操作代码段的两个地址位，即第 10 位和第 11 位。如果两个地址位的值是 00，那么这条指令就解释为“如果 AC 中的数为负值，则执行跳转”。如果地址位是 01（即第 11 位为 0 和第 10 位为 1），则指令解释为“如果 AC 中的数等于 0，执行跳转”。最后，如果两个地址位是 10（或者是 2），则表示“如果 AC 中的数值大于 0，则执行跳转”。跳转“skip”的作用就是简单地跳过下一条指令。这种跳转操作可以通过将 PC 指针加 1 来实现，实际上就是忽略紧接下来的指令，不再对这条指令执行取指操作。Jump 指令是一条无条件跳转指令，它同样会影响 PC 指针。Jump 指令是用 X 的值来替换 PC 中的数值，作为下一条要取指的指令地址。

本例的目的是既要保持体系结构和指令集尽可能简单，同时又要传达那些了解计算机工作原理的必要信息。因此，这里我们忽略了几个有用的指令。但是，读者马上就会发现这套指令集的功能还是非常强大的。一旦熟悉了计算机的工作原理，就可以对这个指令集进行扩充，这样可以使编程更加方便。

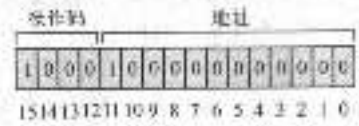
下面来研究 MARIE 所使用的指令格式。假定 MARIE 有如下形式的 16 位指令：



最左边的 4 位表示操作码，也就是要执行的指令。0001 是 1 的二进制数，代表指令 Load。余下的 12 位表示指令要装入的数值的地址。这里代表的是主存储器中的第 3 个地址。上面这条指令的意义是将位于主存储器中地址为 3 的存储单元中的内容复制到 AC 寄存器。下面再讨论另外一条指令：



最左边的 4 位为 0011，等于 3，这是一条（Add）加法指令。地址位的数值，如果按十六进制数，表示的是地址 00D（或者是十进制数 13）。这条指令的意义是到主存储器中地址为 00D 的存储单元中取出数据值，然后与 AC 中的数据值相加，得到的结果（和）又被送回 AC 中，替代 AC 中原来的数值。下面再举一个例子：



这条指令的操作码表示指令 Skipcond。指令的第 10 位和第 11 位（从左往右读）为 10，表示数值 2。这条指令的含义为“如果 AC 中的值大于 0，则执行跳转”。如果 AC 中的数值小于 0，将忽略该指令，程序简单地继续执行下一条指令。但是，如果 AC 中的数值大于 0 或等于 0，则这条指令会使

PC 寄存器中的值增量加 1，导致忽略程序中紧跟在这条指令后面的指令（在阅读有关指令周期时请务必牢记这一点）。

这个例子中最令人感兴趣的内容是将使用这些有限的指令集来编写程序。不知道读者是喜欢采用命令 Load、Add 和 Halt，还是喜欢使用与这些命令等效的二进制数 0001、0011 和 0111 来编写程序呢？一般说来，大多数人编程时都喜欢使用指令的名称，或者说助记符号（mnemonic），而不是二进制数值的指令。二进制指令又称为机器指令（machine instruction）。而对应的助记符号指令则称为汇编语言指令（assembly language instruction）。汇编语言和机器指令之间存在着一一对应关系。当使用汇编语言编写程序时（即使用表 4-2 中列出的各种指令），需要使用一个汇编程序（又称为编译器）将汇编语言转换成对应的二进制命令。有关编译器的内容将在第 4.5 节进行讨论。

4.2.4 寄存器传输表示法

数字计算机系统由许多基本部件组成，包括算术逻辑单元、寄存器组、存储器、译码器和控制单元等。这些单元通过总线相互连接，各种信息可以通过总线在系统中传递流动。MARIE 中的指令集实际上是这些计算机部件用来执行程序的一组机器级别的指令系统。这些指令看起来非常简单，但如果仔细研究在机器部件级别上这些指令的实际执行过程，不难发现每条指令实际上都包含多个操作。例如，Load 指令的作用是将给定的存储器单元中的内容装入 AC 寄存器中。但是，如果仔细观察这条指令在机器部件级别的执行过程，可以发现这条指令实际上包含了多个“微指令（mini-instruction）”的执行过程。首先，指令所指定的地址必须先装到 MAR 寄存器。然后将位于该地址的存储单元中的数据装入 MBR 寄存器。接着 MBR 中的内容被传送至 AC 寄存器。这些微指令又称为微操作（microoperation）。微指令规定了对寄存器中存储的数据可以执行各种最基本的操作。

描述计算机微操作的行为的符号表示法称为寄存器传输表示法（register transfer notation，RTN），或者称为寄存器传输语言（register transfer language，RTL）。在这种表示法中，我们使用符号 $M[X]$ 表示存放在地址为 X 处的存储单元中的数据，而符号←表示信息的传送。实际操作中，寄存器之间的数据传送包含下面两个步骤：数据先从源寄存器传送至总线，然后脱离总线传送到目的寄存器。但是，为了清晰起见，下面的叙述中没有包括总线的传送过程，而是假定读者已经了解了数据的传送必须涉及到总线周期。

下面介绍 MARIE 的指令系统中每条指令的寄存器传输表示法。

Load X

这条指令是将存储单元 X 中的内容装到 AC 寄存器。首先必须将地址 X 放入到 MAR 中。接着，将单元 M [MAR]（或地址 X）的数据移动至 MBR 中。最后，将这个数据移入 AC 寄存器中。

MAR ← X

MBR ← M[MAR], AC ← MBR

因为在地址 X 处的存储单元中的数据被装入到 MBR 之前，指令寄存器（IR）需要使用总线来复制 X 的值到 MAR，所以这个操作需要两个总线周期。很明显，这两个操作分别属于不同的操作线路，这表示它们不能在同一个总线周期内发生。然而，由于在 MBR 和 AC 之间设置有专门的连接通路，所以从 MBR 到 AC 的数据传送可以在数据被放入 MBR 后立刻进行，而不必等待总线周期。

Store X

使用这条指令可以将 AC 中的内容存放到地址 X 处的存储器单元中。

MAR ← X, MBR ← AC

M[MAR] ← MBR

Add X

将存储在地址 X 处的数据值加到 AC 累加器中。指令执行的过程如下：

```
MAR ← X
MBR ← M[MAR]
AC ← AC + MBR
```

Subt X

类似于 Add 指令，从累加器中减去存储在地址 *X* 处的数值，然后将计算结果返回到 AC 中：

```
MAR ← X
MBR ← M[MAR]
AC ← AC − MBR
```

Input

这条指令的功能是将来自输入设备的任何输入数据首先发送到 InREG 寄存器，然后再转移到 AC 中。

```
AC ← InREG
```

Output

这条指令是将 AC 中的内容先放到寄存器 OutREG，然后发送到输出设备上。

```
OutREG ← AC
```

Halt

没有与寄存器有关的操作，该指令简单地使机器终止程序的执行。

Skipcond

记住这一指令使用地址域中位 10 和位 11 的两位来决定对累加器 AC 所进行的比较运算。根据这两个位的不同组合，机器会检查 AC 中的数值，决定这个数值是否为负数，等于 0，或者是大于 0。如果给定的条件为真，则会跳过下一条指令。这种跳转操作是通过将 PC 寄存器中的指针进行增量加 1 来实现的。

```
If IR[11-10]=00 then        {if bits 10 and 11 in the IR are both 0}
   If AC < 0 then PC ← PC+1
else If IR[11-10]=01 then    {if bit 11 = 0 and bit 10 = 1}
   If AC = 0 then PC← PC + 1
else If IR[11-10]= 10 then   {if bit 11 = 1 and bit 10 = 0}
   If AC > 0 then PC ← PC + 1
```

如果第 10 位和第 11 位都为 1，就会产生一个错误条件。当然，也可以利用这两位都为 1 值来定义另外一个条件跳转。

Jump X

指令执行一个转向给定地址 *X* 处的无条件分支转移。所以，在执行这条指令时，地址 *X* 必须先行装入到 PC 中。

```
PC ←X
```

事实上，指令寄存器中的低 12 位或者说最右边的 12 位（或 IR [11-0]）表示的是地址 *X* 的值。所以这种转移可以更准确地描述为：

```
PC ←IR [11-0]
```

但是，也许读者会觉得符号 PC ← X 更容易理解，并且与实际的指令相关联。所以一般使用这种表示方法。

寄存器传输表示法（RTN）属于一种符号表示方法。它所表示的是，在执行特定的指令时计算机系统内部所发生的各种过程。寄存器传输表示法与数据通路有关。如果存在多个微操作必须共用总线的话，那么微操作必须按照顺序安排的方式，逐个执行。

4.3 指令的执行过程

至此，我们已经介绍了一种基本的计算机语言。利用这种语言，可以进行人机对话。下面讨论特定程序的执行过程。现在所有的计算机系统都遵循一个基本循环过程：取指、译码和执行周期。

4.3.1 取指-译码-执行周期

取指-译码-执行（fetch-decode-execute）表示计算机运行程序时所遵循的步骤。CPU 首先提取一条指令，即将指令从主存储器转移到指令寄存器；接着对指令进行译码，即确定指令的操作码和提取执行该指令所需的数据；然后执行这条指令，即执行指令规定的各种操作。注意，在这个循环过程中，计算机的大部分工作都是在执行从一个单元将数据复制到另一个单元的任务。当一个程序最初被装入到 CPU 时，第一条指令的地址也必须被装入 PC 中。下面列出了计算机工作周期的各个步骤，即特定时间周期内发生的各种操作。步骤 1 和步骤 2 组成了取指过程，步骤 3 表示译码过程，步骤 4 则是执行过程。

1. 将 PC 中的内容复制到 MAR，MAR←PC。

2. CPU 转向主存储器，提取由 MAR 给出的地址单元中的指令，并将指令放入指令寄存器 IR 中。同时，PC 自动加 1，现在 PC 指向程序的下一条指令：IR←M [MAR] 和 PC←PC+1。注意，因为 MARIE 是按字编址的，所以 PC 增量加 1 所产生的实际效果是下一个字的地址将占据 PC 寄存器。如果 MARIE 是按字节编址的，则 PC 需要增量加 2 才能指向下一条指令的地址，原因是每条指令占有两个字节的宽度。而如果 MARIE 是一个 32 位字的按字节编址的计算机系统，则 PC 需要进行增量加 4 的操作。

3. 将 IR 中最右边的 12 位复制到 MAR；并对 IR 最左边的 4 位进行译码，以确定操作码：MAR←IR [11-0] 和译码 IR [15-12]。

4. 如果需要，CPU 将使用 MAR 中的地址转向存储器提取数据，并将数据放入 MBR（可能是 AC）中，然后执行该指令：MBR←M [MAR]，并且执行实际的指令。

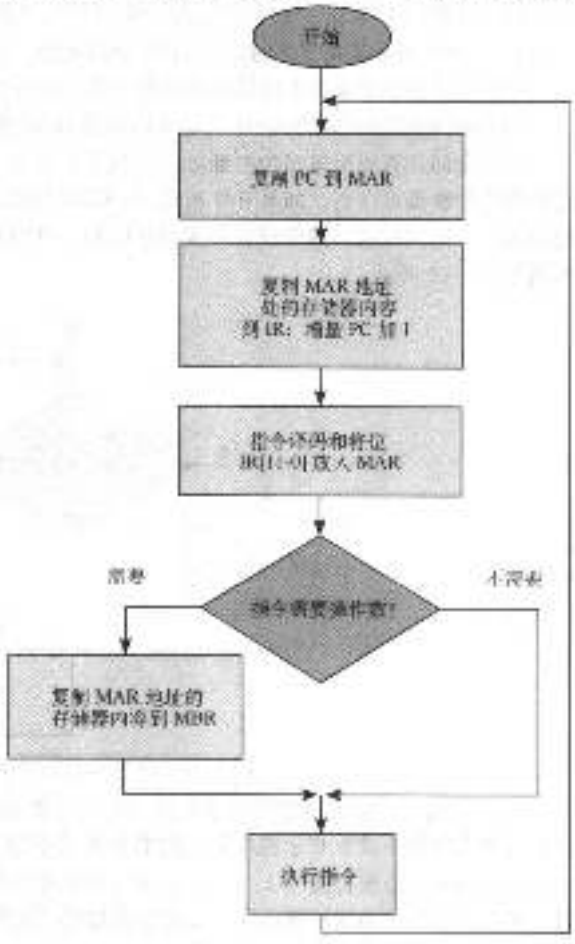


图 4-11 取指-译码-执行周期

计算机执行程序的这一循环过程可以采用流程图的形式来表示，如图 4-11 所示。

注意，现代的计算机系统，具有庞大的指令集，超长的指令和巨大的存储器系统，可以在瞬间执行几百万个这种取指-译码-执行的循环过程。

4.3.2 中断和输入/输出

第 7 章将详细讨论输入和输出，这里只介绍有关输入/输出（I/O）的基本概念，读者可以对程序执行的全过程有一个清晰的了解。

MARIE有两个寄存器负责处理输入和输出操作。输入寄存器保持由输入设备传送到计算机的数据，输出寄存器则是保持准备传送到输出设备的各种信息。从物理上来说，这两个寄存器所使用的定时机制非常重要。例如，如果采用键盘输入，并且打字的速度很快时，计算机必须有足够的时间来读取输入到输入寄存器的每一个字符。如果在计算机能够有机会处理完当前的字符之前，又有另外一个字符被送入到输入寄存器，那么当前的那个字符就会丢失。一种更可能发生的情况是，由于处理器的速度非常快，而键盘输入的速度很慢，所以处理器可能会多次从寄存器重复读取同一个字符。事实上，必须避免这两种情况发生。

在MARIE模型中，使用中断控制的输入输出（interrupt-driven I/O）来解决上述这些问题（有关各种类型的I/O机制的详细讨论，读者可以参见第7章的内容）。当CPU要执行输入或输出指令时，首先通知相应的I/O设备。然后，继续处理其他的一些工作任务，直到该I/O设备准备就绪。这时，I/O设备会向CPU发送一个中断请求信号。随后，CPU会响应和处理这个中断请求，完成输入或输出操作后，CPU会继续其正常的取指-译码-执行周期。这一具体的处理过程如下：

- CPU收到一个来自I/O设备的信号（中断信号），表示输入或输出的准备工作已经完成。
- 利用某些方法可以使CPU从正常的取指-译码-执行周期转去“识别”这个中断请求。

大部分的计算机所采用的中断处理方法是：在机器的每一个取指-译码-执行周期开始处，首先检查是否有中断请求存在。如果有中断请求，CPU先处理中断任务。中断处理完成后，CPU会继续进行机器的指令执行过程。如果没有发现中断请求，CPU则进行正常的程序执行过程。中断处理过程的流程图如图4-12所示。

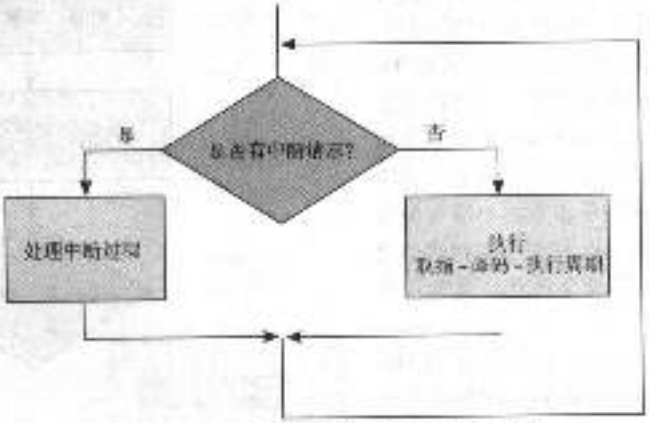


图4-12 为检查中断请求而修正的指令周期

通常情况下，输入或输出设备使用一个特殊的寄存器、状态或标志寄存器，来发送中断信号。通过在这个寄存器中设置某个特殊的二进制位来表示有一个中断请求产生。例如，只要键盘输入开始，这个特殊的位就会被设置为1，CPU会在每个机器周期的开始处，自动检查该位的状态。如果发现该位已经被设置为1，CPU就会转去处理中断过程。如果未设置该位，则CPU执行正常的取指-译码-执行周期，继续处理当前运行的程序中的指令。

如果CPU发现中断位被设置，就会执行中断处理任务。中断处理的具体任务是由所产生的中断的类型决定的。输入/输出中断并不是程序执行过程中可能产生的唯一的中断类型。读者是否使用过键入“Ctrl-break”或“Ctrl-C”来终止一个程序？这就是另外一种类型的中断的例子。计算机中有二种类型的中断：由外部事件（比如输入/输出，或电源掉电等）产生的外部中断；由于程序中的一些异常情况（例如，被0除，堆栈溢出，或保护系统侵犯等）产生的内部中断以及执行程序中的某条指令（例如，要求程序的执行从一种运行环境，如用户层，转到另一个运行环境，如内核层等）所引起的软件中断。

不管是哪种类型的中断调用，中断处理过程都是相同的。当CPU识别一个中断请求时，就能确定中断服务程序的地址（通常由硬件决定），并且执行这一中断服务的程序（非常类似于一个普通的程序进程）。CPU会从原来的程序运行转去执行某个特殊的程序进程来处理中断。CPU处理中断服务程序，也是对各种中断指令执行正常的取指-译码-执行周期，直到所有的中断程序编码运行完毕。然后，CPU又会返回到中断发生前所运行的程序位置。CPU必须严格返回到中断处理前的原始程序的运行位置。因此，当CPU要去执行中断服务程序时，必须先存储PC中的内容，CPU的所有寄存器中的内容，以及原始程序中原有的各种状态条件。而在中断服务程序完成后，CPU必须严格恢复到原始程序运行的真实环境，然后又开始对原来运行的程序进行取指、译码和执行指令的操作。

4.4 一个简单的程序

下面介绍为MARIE模型编写的一个简单的程序。第4.6节将列举另外几个例子来展示这一模型计算机体系结构的强大功能。MARIE模型甚至可以运行一些包含各种进程、循环结构和不同选项的程序。

第一个程序是实现两个数的相加（这两个加数均位于主存储器中），并将求和结果存储在存储器中（暂时不考虑输入/输出）。

表4-3列出了实现上述加法操作的汇编语言程序，及其对应的机器语言程序。指令栏中列出的指令序列就是实际的汇编语言程序。我们知道，程序的运行是从提取第一条程序指令开始进入取指-译码-执行周期的。因此，当装入程序开始执行时，首先要将第一条指令的地址装入到PC中。为简单起见，这里假定在MARIE模型中运行的程序总是从地址100（十六进制）开始装入内存的。

表4-3 实现两个数相加的一个简单程序

十六进制的地址	指令		存储器地址二进制内容	存储器十六进制内容
100	Load	104	0001000100000100	1104
101	Add	105	0011000100000101	3105
102	Store	106	0010000100000110	2106
103	Halt		0111000000000000	7000
104	0023		0000000000100011	0023
105	FFE9		1111111111101001	FFE9
106	0000		0000000000000000	0000

表中存储器地址的二进制内容一栏列出的指令序列就是实际的机器语言程序。因为十六进制程序要比二进制程序更易于阅读，所以存储器中的实际内容也都用十六进制数的方式。

程序首先将位于地址104的存储单元中的第一加数0023_{16}（十进制数值为35）装入AC中。然后加上地址105的十六进制数值FFE9（十进制数为−23）。求和的计算结果为12，并且被放回到AC中。Store指令会将这个求和结果存储到地址为106的存储单元中。当程序运行完毕后，地址106的存储单元的内容就变成了0000000000001100，十六进制数是000C，或十进制数12。图4-13给出了程序执行的具体步骤，以及所对应的各个寄存器中的内容。

图中部分c的最后一条RTN指令是将求和结果放到指定的存储单元。"decode IR [15-12]"语句只表示必须对指令进行译码，才能决定计算机所要执行的操作。译码过程可以利用软件（使用一个微程序）或者硬件（使用要连线的电路）来实现。这两个译码方法将在第4.7节详述。

a) 装入 104

Step	RTN	PC	IR	MAR	MBR	AC
(initial values)		100				
Fetch	MAR ← PC	100		100		
	IR ← M[MAR]	100	1104	100		
	PC ← PC+1	101	1104	100		
Decode	MAR ← IR[11-0]	101	1104	104		
	(Decode IR[15-12])	101	1104	104		
Get operand	MBR ← M[MAR]	101	1104	104	0023	
Execute	AC ← MBR	101	1104	104	0023	0023

b) 加入 105

Step	RTN	PC	IR	MAR	MBR	AC
(initial values)		101	1104	104	0023	0023
Fetch	MAR ← PC	101	1104	101	0023	0023
	IR ← M[MAR]	101	3105	101	0023	0023
	PC ← PC+1	102	3105	101	0023	0023
Decode	MAR ← IR[11-0]	102	3105	105	0023	0023
	(Decode IR[15-12])	102	3105	105	0023	0023
Get operand	MBR ← M[MAR]	102	3105	105	F2E9	0023
Execute	AC ← AC+MBR	102	3105	105	F2E9	030C

c) 存储 106

Step	RTN	PC	IR	MAR	MBR	AC
(initial values)		102	3105	105	FFE9	030C
Fetch	MAR ← PC	102	3105	102	FFE9	030C
	IR ← M[MAR]	102	2106	102	FFE9	030C
	PC ← PC+1	103	2106	102	FFE9	030C
Decode	MAR ← IR[11-0]	103	2106	106	FFE9	030C
	(Decode IR[15-12])	103	2106	106	FFE9	030C
Get operand	(not necessary)	103	2106	106	FFE9	030C
Execute	MBR ← AC	103	2106	106	030C	030C
	M[MAR] ← MBR	103	2106	106	030C	030C

图 4-13 两个数字相加的程序的具体执行过程

注意，汇编语言和机器语言指令之间存在着一一对应的关系。利用这种对应关系可以很方便地进行汇编语言和机器代码之间的转换。利用本章给出的指令列表，读者可以对本章举例中的各个程序进行编译处理。基于这种理由，我们在以后的介绍中，只讨论汇编语言代码。但是，在列举更多的例子之前，我们先来讨论程序的编译过程。

4.5 有关编译程序的讨论

对于表 4.3 中的程序，将汇编语言指令（例如 Load 104），转换成机器语言指令 1104（十六进制）很简单。但是，为什么要为这种语言之间的转换而烦恼呢？为什么不直接采用机器代码来编写程序？虽然对计算机来说，阅读二进制指令的效率是非常高的。但是，对我们来说，要理解这些由 0 和 1 序列组成的程序却是一件十分困难的事情。我们宁愿使用单词和符号，而不愿意去阅读那些长串的二进

制数字。因此，设计一个程序来完成这种简单的转换工作，似乎是一种很自然的解决方案。这样的程序称之为编译程序（assembler）。

4.5.1 编译程序的功能

编译程序的任务就是使用助记符号将汇编语言转换成机器语言，即完全由二进制数，或者说完全由0和1的数字串组成的语句。编译程序首先阅读程序员编写的汇编语言程序，这些汇编程序实际上是一些二进制数的符号表示形式。然后，将这些汇编程序转换成二进制的指令，或者说是对应的机器代码。这里，编译程序阅读的是由汇编语言编写的源文件（source file），生成的是由机器代码组成的目标文件（object file）。

利用简单的字母名称来替代操作码会使编程变得更容易。同样，可以使用标记符号（label）（简单名称）来标识或命名一些特定的存储器地址，这使编写汇编程序的工作变得更简单。例如，在上面介绍的两个数字相加的程序中，就可以使用标记符号来代表存储器地址，这样在编程时我们无需直接知道指令中各操作数的具体的存储器地址。表4-4说明的就是这样一种概念。

表4-4 使用标识符号的例子

地址		指令	
	100	Load	X
	101	Add	Y
	102	Store	Z
	103	Halt	
X:	104	0023	
Y:	105	FFF6	
Z:	106	0000	

如果指令的地址域放置的是一个标记符号（简称为标号），而不是真实的物理地址，编译程序仍然必须将其翻译成主存储器中的真实物理地址。大多数汇编语言都允许使用标记符号。通常编译程序会规定各种指令的所遵从的格式次则，其中包括有关标记符号的各种规定。例如，可以规定标号限制为3个字符，同样也可以要求标号出现在指令的第一个位置上（或）。MARIE则要求标号后面必须要加一个“,”的标点符号。

标号对编程人员来说非常方便。但是，对编译程序而言，标号的翻译却需要花费更多步骤。对于采用标号编写的汇编程序，编译程序必须进行两次转换。这就意味着编译程序需要通读程序两次，每次阅读都是按从上至下的顺序进行。在第一次通读时，编译程序会建立一组称为符号表（symbol table）的对应关系。例如，在上面的例子中，编译程序就建立了一个包含三个符号（X、Y和Z）的对应表。因为当编译程序从上至下阅读程序代码时，不可能一次就将全部的汇编语言指令转换成机器代码。原因是对于只使用标号的指令，编译程序还不知道指令的数据部分存放在存储器中的具体位置。但是，在符号表建立之后，编译程序可以进行第二次通读，并且会“填充原来的空白位置”。

在上述例子中，编译程序的第一次通读生成一个以下的符号表：

X	104
Y	105
Z	106

同时，编译程序也开始翻译程序指令。在进行第一次通读后，翻译出来的指令并不完整，如下表所示：

1	X
3	Y
2	Z
7 0 0	0

在第二次通读时，编译程序使用符号表来填充空白地址，并且生成相应的机器语言指令。这样，在第二次通读时，编译程序就知道了X是位于地址104的存储单元，并且会使用104来替代标记符号X。同样，也使用类似的过程来替代Y和Z。结果如下：

1	1	0	4
9	1	0	5
8	1	0	6
7	0	0	0

因为大多数的人都不喜欢读十六进制数，所以大多数汇编语言允许指定存储在存储器中的数据值是二进制数，十六进制数，或十进制数。通常，编译程序会使用某种类型的汇编指令（assembler directive）来专门指定解释这些数值所采用的基数。汇编指令是一种专门为编译程序设计的指令，它本身是不会被翻译成机器代码的。在 MARIE 汇编语言中我们用 DEC 表示十进制数，HEX 表示十六进制数。例如，可以使用汇编指令的方法重编写表 4-4 中的程序，结果如表 4-5 所示。

如果并不希望阅读实际的二进制的数据值（书写形式为十六进制数），可以使用汇编指令 DEC 来指定一个十进制的数值。编译程序能够识别该汇编指令，并且会在数据存放到存储器之前转换成相应的二进制数值。另外，该汇编指令不会被转换成机器语言，只是指示编译程序的某种工作方式。

表 4-5 使用汇编指令的常数表示法

地址		指令	
	100	Load	X
	101	Add	Y
	102	Store	S
	103	Halt	
X,	104	DEC	35
Y,	105	DEC	-23
S,	106	HEX	0000

另外注释分隔符（comment delimiter）是一种几乎对每一种编程语言都通用的汇编指令。它是一些特殊的字符，用来告诉编译程序（或编译器）进行编译转换时，忽略跟随在这个特殊符号后面的所有文本内容。MARIE 的注释分隔符是一个前斜杠“/”符号，其作用是忽略位于这个分隔符和该行的行末之间的所有文本内容。

4.5.2 为什么使用汇编语言

介绍 MARIE 的汇编语言的主要目的是为了使读者了解这种语言和计算机体系结构之间的相互关系。了解了怎样利用汇编语言编程就可以很好地理解计算机的体系结构，反之亦然。学习汇编语言不但可以了解计算机的基本体系结构，而且可以真正认识处理器的工作方式并深刻理解所编程的特定的计算机系统的具体内部构造。学习汇编语言编程还会有其他方面的一些益处。

大多数的程序员都认为通常一个程序中 10%的代码可能会占用 90%的 CPU 时间。对于一些时间因素非常关键的应用，常常需要对这部分 10%的代码进行优化处理。一般情况下，编译程序可以处理这种优化过程。它首先处理高级语言（如 C++语言），将其转换成汇编语言，然后再转换成机器代码。编译程序的应用已经有一段相当长的时间，而且在大多数情况下能够高效工作。但是，在某些情况下，程序员需要超出高级语言的某些限制，而必须直接使用汇编代码。通过这种方法，程序员可以使程序在时间和空间上都具有更高效率。通常，利用混合编程的方法，即一个程序的大部分内容使用高级语言，但某些部分直接使用汇编语言，可以让程序员充分发挥这两种语言的各自优势。

现在的问题是，在何种情况下，程序应该采用汇编语言来编写？如果一个程序的大小或响应时间是程序设计考虑的关键因素，汇编语言通常是编程的首选。这是因为编译程序会遮蔽掉有关各种操作的耗时信息。这样，程序员常常很难判断由编译程序编译的程序的实际运行过程。而汇编语言可以让程序员更贴近机器的体系结构，且更准确地控制程序的执行。如果程序员需要实现某些高级语言所没有的操作，那么使用汇编语言是必不可少的。

如果考虑响应特性和空间（即程序大小）的作为程序设计关键因素，则可以在嵌入式系统（embedded system）中找到一个理想的例证。在实际应用的嵌入式系统中，计算机通常被集成为某个系统中的一个部件，而不是纯粹的计算机。嵌入式系统的应用非常广泛，经常时间约束（time-constrained）的环境下使用。这些嵌入式系统常常被设计为只能执行某个单一指令或是执行某种非常专用的指令系统。也许我们每天都在使用某种类型的嵌入式系统。例如，消费者使用的电子产品（如照相机、摄录机、移动电话、PDA 和交互游戏设备等），家用电器（如洗碗机、微波炉和洗衣机等），汽车（特别是

其中的发动机控制和防抱死刹车系统），医疗仪器（如 CAT 扫描仪和心脏监视器等），以及工业设备（如生产过程的控制设备和航空设备）等，这些都只是嵌入式系统应用的一少部分例子。

对于一个嵌入式系统，软件是非常关键的。软件需要在某些非常特殊的响应参量之中操作运行，并且它所能够占用的空间是非常有限的。因此，在嵌入式系统中使用汇编语言编程是非常合适的。

4.6 MARIE 指令集的扩充

尽管利用 MARIE 的指令集足以编写任何所需的程序，但可以再增加几条指令使 MARIE 的编程任务变得更加简单。MARIE 的指令代码使用了 4 位二进制数，这意味着共可以生成 16 条指令，而我们只使用了其中的 9 条指令。表 4-6 列出了扩充 MARIE 的指令集所增加的指令。

表 4-6 MARIE 的扩充指令集

指令编号（十六进制）	指令	意　义
0	JnS X	将 PC 内容存储到地址 *X* 处，然后跳转到地址 *X*+1。
A	Clear	AC 中的所有位清 0。
B	AddI X	间接相加：进入地址 *X* 处，使用 *X* 单元的数值作为数据操作数的实际地址，取出操作数加到累加器 AC 中。
C	JumpI X	间接转移：进入地址 *X* 处，使用 *X* 单元的数值作为存储单元的实际地址，然后跳转至该存储单元。

JnS（跳转-和-存储）指令可以对某个返回指令存储一个地址指针，然后继续处理，对不同的指令设置新的 PC 值。利用这条指令可以进行过程调用和其他子程序的调用，而当子程序执行完成后系统又会返回到原程序代码中的调用点。Clear 指令会将累加器的全部二进制位设置为 0。利用这条指令可以缩短指令周期，否则完成相同的清零操作需要从存储器中装入 0 操作数而花费更多的时间。

AddI 指令（还有 JumpI 指令）采用的是不同的*寻址方式*（addressing mode）。前面介绍的所有指令都假定了指令数据部分的数值就是指令所需要的操作数的直接地址（direct address）。而 AddI 指令则是采用*间接寻址方式*（indirect addressing mode）。第 5 章将讨论更多种类的寻址方式。在这里，CPU 不是采用地址 *X* 的值作为操作数的实际地址，而是使用地址 *X* 的存储器单元内的数据值作为一个指针，指向一个新的存储器单元。这个新单元内的数值才是指令要使用的数据。例如，已知指令为 AddI 400。具体执行过程是首先到达地址 400 的存储单元，并假定读出存储单元 400 中的值是 240，然后 CPU 转向地址为 240 的单元以获取指令所需的实际操作数。从本质上来说，我们已经在程序语言中使用了多个地址指针。

如果采用寄存器传输表示法，这些新的指令可以表示如下形式：

JnS

```
MBR ← PC
MAR ← X
M[MAR] ← MBR
MBR ← X
AC ← 1
AC ← AC+MBR
PC ← AC
```

Clear

```
AC ← 0
```

AddI *X*

```
MAR ← X
MBR ← M[MAR]
MAR ← MBR
MBR ← M[MAR]
```

```
AC ← AC+MBR
JumpI X
MAR ← X
MBR ← M[MAR]
PC ← MBR
```

表 4.7 概括了 MARIE 的全部指令集。

表 4-7 MARIE 的完整指令集

操作码	指令	RTN
0000	JnS X	MBR ← PC MAR ← X M[MAR] ← MBR MBR ← X AC ← 1 AC ← AC+MBR PC ← AC
0001	Load X	MAR ← X MBR ← M[MAR], AC ← MBR
0010	Store X	MAR ← X, MBR ← AC M[MAR] ← MBR
0011	Add X	MAR ← X MBR ← M[MAR] AC ← AC+MBR
0100	Subt X	MAR ← X MBR ← M[MAR] AC ← AC-MBR
0101	Input	AC ← InREG
0110	Output	OutREG ← AC
0111	Halt	
1000	Skipcond	If IR[11-10]=00 then If AC < 0 then PC ← PC+1 Else If IR[11-10]=01 then If AC=0 then PC ← PC+1 Else If IR[11-10]=10 then If AC > 0 then PC ← PC+1
1001	Jump X	PC ← IR[11-0]
1010	Clear	AC ← 0
1011	AddI X	MAR ← X MBR ← M[MAR] MAR ← MBR MBR ← M[MAR] AC ← AC+MBR
1100	JumpI X	MAR ← X MBR ← M[MAR] PC ← MBR

下面来看看使用完整指令集的一些示例。

例 4-1 使用一个循环过程来编写 5 个数字相加的汇编程序。

```
Address  Instruction  Comments
    100  Load         Addr /Load address of first number to be added
    101  Store        Next /Store this address as our Next pointer
    102  Load         Num  /Load the number of items to be added
    103  Subt         One  /Decrement
    104  Store        Ctr  /Store this value in Ctr to control looping
    105  Clear             /Clear AC
Loop, 106 Load       Sum  /Load the Sum into AC
    107  AddI         Next /Add the value pointed to by location Next
    108  Store        Sum  /Store this Sum
    109  Load         Next /Load Next
    10A  Add          One  /Increment by one to point to next address
    10B  Store        Next /Store in our pointer Next
    10C  Load         Ctr  /Load the loop control variable
    10D  Subt         One  /Subtract one from the loop control variable
    10E  Store        Ctr  /Store this new value in the loop control variable
    10F  Skipcond     000  /If control variable < 0, skip next instruction
    110  Jump         Loop /Otherwise, go to Loop
    111  Halt              /Terminate program
Addr, 112 Hex         118  /Numbers to be summed start at location 118
Next, 113 Hex         0    /A pointer to the next number to add
Num,  114 Dec         5    /The number of values to add
Sum,  115 Dec         0    /The sum
Ctr,  116 Hex         0    /The loop control variable
One,  117 Dec         1    /Used to increment and decrement by 1
    118  Dec          10   /The values to be added together
    119  Dec          15
    11A  Dec          20
    11B  Dec          25
    11C  Dec          30
```

■

编程中使用了注释来对各条程序语句进行合理的解释。现在，我们还要针对例 4-1 做一些讨论。读者可以回顾一下符号表存储有关［标号、存储单元］对的情形。指令 Load Addr 现在变成了 Load 112，因为 Addr 被定为物理存储器中地址 112。然后将存放在 Addr 的数值 118 存放到 Next 处。Next 是一个地址指针，允许把要进行相加的 5 个数值（分别位于地址 118、119、11A、11B 和 11C 的单元内）“逐次装入”。变量 Ctr 跟踪记录程序所执行的循环迭代的次数。每次循环都从 Ctr 减去 1，然后检查 Ctr 是否为负值。如果 Ctr 为负值，则终止循环。然后将求和值 Sum（初始值设为 0）装入 AC 中。循环开始后，使用 Next 作为要加到 AC 中的各个数据的地址。当 Ctr 为负值时，Skipcond 语句就会终止循环，即跳过接下来的一条无条件跳转到循环体顶部的指令。CPU 执行到 Halt 指令时，整个程序的运行就终止了。

例 4-2 则表示怎样利用 Skipcond 指令和 Jump 指令来实现不同的选择。虽然本程序展示的是一个 if/else 结构，但是读者也可以方便地对程序进行修改，使其能够实现一个 if/then 结构，或者 case（或 switch）结构的功能。

例 4-2 下面的程序举例说明了如何使用一个 if / else 结构来实现多种选择。具体说，程序执行了如下的操作：

```
if X = Y then
  X := X × 2
else
  Y := Y - X;
```

```
   Address  Instruction      Comments
If,    100  Load       X     /Load the first value
       101  Subt       Y     /Subtract value of Y and store result in AC
       102  Skipcond   400   /If AC = 0, skip the next instruction
       103  Jump       Else  /Jump to the Else part if AC is not equal to 0
Then,  104  Load       X     /Reload X so it can be doubled
       105  Add        X     /Double X
```

```
        106  Store   X       /Store the new value
        107  Jump    Endif   /Skip over Else part to end of If
Else,   108  Load    Y       /Start the Else part by loading Y
        109  Subt    X       /Subtract X from Y
        10A  Store   Y       /Store Y - X in Y
Endif,  10B  Halt            /Terminate program (it doesn't do much!)
X,      10C  Dec     12      /Assume these values for X and Y.
Y,      10D  Dec     20
```

例 4-3 说明了怎样利用 JnS 和 JumpI 指令来实现子程序的调用。程序包括一条 END 语句，这是汇编指令的另一个例子。这条语句通知编译程序应该在哪里停止程序。其他一些可能的汇编指令包括下面一些语句：如让编译程序知道到哪里去寻找第一条程序指令，如何设置存储器单元，以及各个代码模块是否属于程序过程等。

例 4-3 下面的程序举例说明了如何利用一个简单的子程序对任意数字进行加倍操作，并且可以对子程序进行编码。注意，程序行的编号仅用于信息标示。

```
        100  Load   X     /Load the first number to be doubled
        101  Store  Temp  /Use Temp as a parameter to pass value to Subr
        102  JnS    Subr  /Store return address, jump to procedure
        103  Store  X     /Store first number, doubled
        104  Load   Y     /Load the second number to be doubled
        105  Store  Temp  /Use Temp as a parameter to pass value to Subr
        106  JnS    Subr  /Store return address, jump to procedure
        107  Store  Y     /Store second number, doubled
        108  Halt        /End program
X,      109  Dec    20
Y,      10A  Dec    48
Temp,   10B  Dec    0
Subr,   10C  Hex    0     /Store return address here
        10D  Clear       /Clear AC as it was modified by JnS
        10E  Load   Temp  /Actual subroutine to double numbers
        10F  Add    Temp  /AC now holds double the value of Temp
        110  JumpI  Subr  /Return to calling code
             END
```

利用 MARIE 的简单指令集可以实现任何高级编程语言的结构功能，例如 loop 语句和 while 语句。这些内容将在本章结尾留给读者作为练习。

4.7 有关译码的讨论：硬件译码和微程序控制译码

控制单元实际上是如何工作的呢？前面我们已经实施了一些人为的控制过程，并且简单地假设每个事件都会按照我们所描述的方式进行。对这种做法的基本理解是，对于每条指令，控制单元都能够控制 CPU 按正确的步骤执行操作。事实上，CPU 中必须有一些控制信号连线施加到各个数字部件，才使 CPU 可以按照上述的方式正确工作（读者可以回顾第 3 章描述的各个数字部件）。例如，在 MARIE 使用汇编语言执行 Add 指令时，实际上就已经假设了加法运算可以发生。因为控制信号将 ALU 设置为加法运算，并将结果存放到 AC 中。ALU 具有不同的控制线，可以决定执行哪一种操作。现在要解决的问题是，这些控制线实际上是怎样被选中有效？

可以采用两种方法来正确设置各条控制线。第一种方法是从物理上将各条控制线与实际的机器指令连接起来。一般来说，指令被划分成不同的域。指令中不同的位通过各种数字逻辑部件组合连接在一起，用来驱动不同的控制线。这种方法称为*硬连线控制*（hardwired control），如图 4-14 所示。

利用硬件可以实现控制单元，例如采用一些简单的与非门、触发器和计数器等。可以使用一些特殊的数字电路来作为控制单元的输入，例如连接来自指令操作码域的代码位，来自标志（或状态）寄存器的标志位，来自总线的信号以及来自时钟的信号等。当然，控制单元还要产生一些输出的控制信

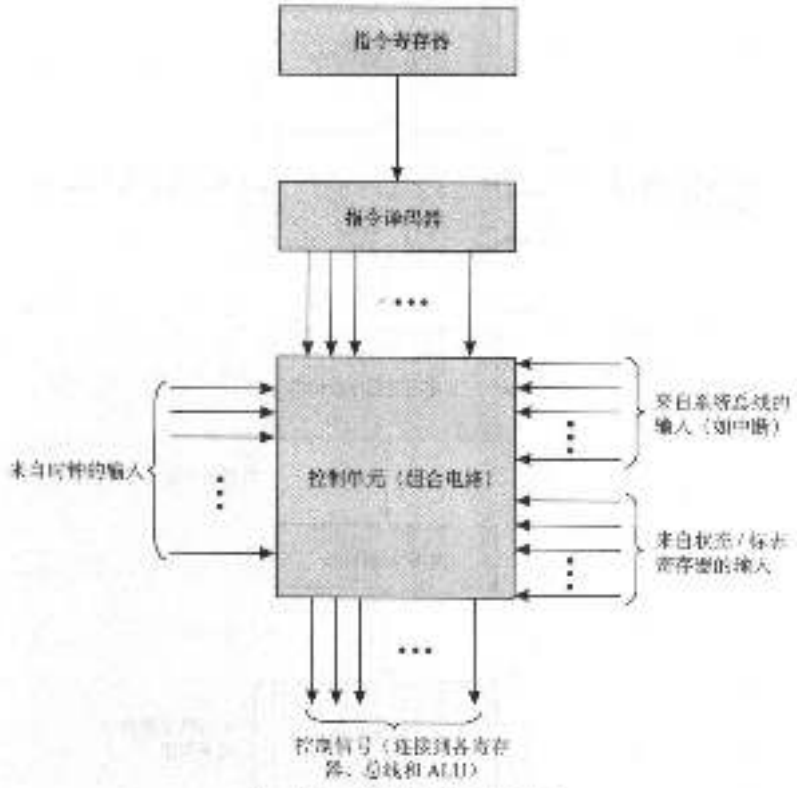


图 4.14 硬件控制单元示意图

号用来驱动计算机中的各个部件。例如，可以使用 4-16 译码器来对操作码进行译码。利用指令寄存器（IR）中的内容和 ALU 的状态，控制单元可以控制各个寄存器操作、ALU 的操作、所有的移位电路和总线访问等。

硬连线控制的优点是速度快。缺点是指令集和控制逻辑通过特殊的电路直接连接，使得电路变得比较复杂，设计或修改都比较困难。如果一台计算机系统是由硬连线控制的，那么要进行指令集的扩充（正如在 MARIE 机器中所做的一样），就必须改变计算机内部的物理部件。这样，费用会出奇地昂贵。原因是不但需要增加新的功能芯片，而且原有芯片也需要重新定位和放置。

另外一种控制方法是微编程（microprogramming），即使用软件来进行控制，如图 4.15 所示。

这里，所有的机器指令都被放置到一个专用的程序（微程序（microprogram））中。然后将这些指令转换成规定的控制信号。微程序其实是一个用微代码（microcode）编写的翻译器，这些微程序被存储在计算机固件（如 ROM、PROM 或 EPROM 等）中，称为控制存储器（control store）。这个微程序将由 0 和 1 组成的机器指令转换成各种控制信号。从本质上来说，这个微程序中的每条机器指令都有一个子程序。这种方法的优点是如果指令集需要修改，只需要简单地更新微程序，而实际的硬件部分不需要做任何改变。

微编程在设计方面非常灵活简单，而且有助于设计功能强大的指令集。微编程方法可以非常方便地让设计人员在硬件和软件之间作出权衡：如果设计上有些内容不能用硬件来实现（例如，机器上没有乘法指令），则可以使用微代码来实现。这种方法的缺点是所有指令都需要经过一次额外的翻译过程，这样会减慢整个程序的执行速度。除了时间上的代价外，还有实际开发成本问题，因为微编程还需要一些专门的工具。第 9 章将详细讨论有关硬连线控制和微编程控制的问题。

非常重要的一点是，无论使用硬连线控制还是使用微编程控制，关键问题在于同步和定时。控制单元负责发送实际的定时信号，来指挥所有的数据传送和协调计算机的各种动作。这些定时信号

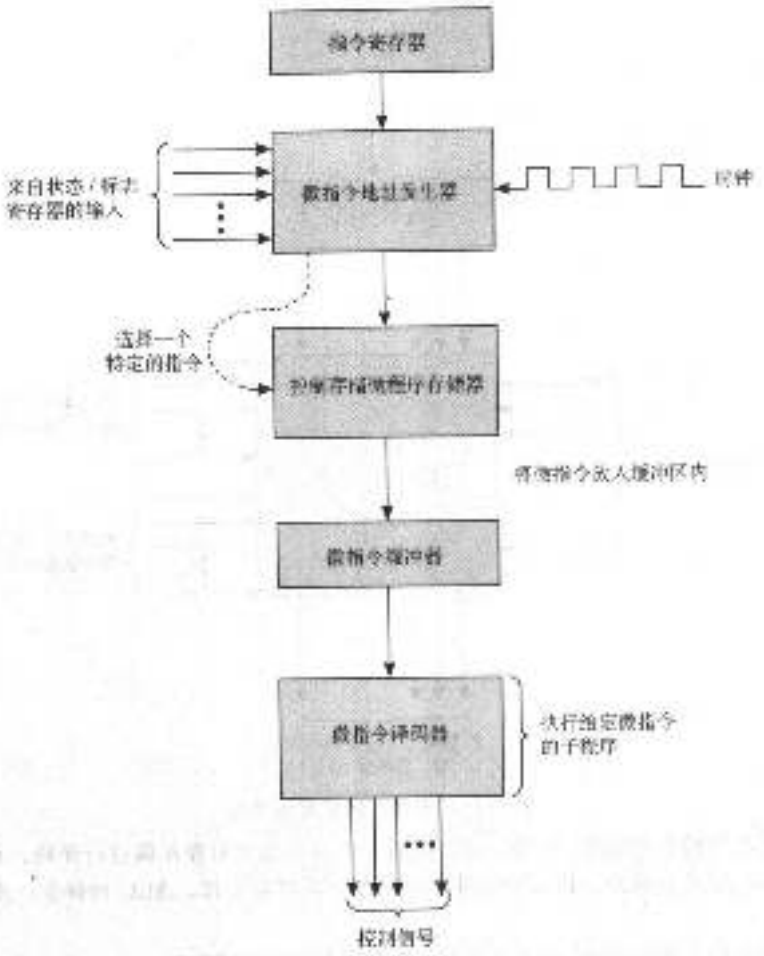


图 4-15 微程序控制方法

可以使用一个简单的二进制计数器按顺序产生。例如，某个计算机体系结构的定时信号可能包含 T_1、T_2、T_3、T_4、T_5、T_6、T_7 和 T_8 几个时钟周期。这些时钟定时信号将控制计算机中各种操作动作的发生，例如，取指操作可能只是发生在 T_1 周期，而提取操作数的操作可能只是发生在 T_4 周期。前面我们已经知道寄存器只有当时钟脉冲到来时，才可能改变其状态。同样，这些状态的改变需要与特定的定时信号相关联。例如，第 3 章中的存储器就含有一条写允许（Write Enable）的控制线，这条控制线和一个定时信号线一起构成“与”操作，以保证存储器的存储单元只能在规定的时间间隔内发生改变。

4.8 实际的计算机体系结构

本章介绍的 MARIE 体系结构是非常简单的，其目的是为了便于读者理解有关计算机体系结构的基本概念，这样不会让读者觉得计算机是一个深不可测的复杂系统。虽然 MARIE 体系结构和汇编语言功能十分强大，足以解决利用各种高级语言（如 C++、Ada、或 Java 语言）在现代计算机体系结构上实现的所有问题。但是，读者会发现 MARIE 体系结构在工作效率，编写程序的复杂性和程序调试等方面都存在局限性。例如，如果在 MARIE 的 CPU 中多增加一些寄存器，以提高存储能力，那么 MARIE 的性能就可以获得重大改进。而对于程序员来说，使得编程变得更加容易是 MARIE 体系结构中另一项需要改进的工作。例如，MARIE 程序员就非常希望使用一些带有参数的过程子程序，虽然

MARIE 可以执行某些子程序，例如，程序可以进行分支转移，可以跳转到不同的代码段去执行分支任务，然后返回等。但是，MARIE 还没有一种机制来支持子程序中的参量传递过程。尽管可以不使用参量来编写程序，但是很明显，参量的使用不仅可以提高程序的工作效率（特别是涉及重用的地方），而用使程序更易于编写和调试。

为了增加参量内容，MARIE 需要一个堆栈（stack）结构。堆栈是一种数据结构类型，它可以保持一系列的数据项，这些项只能从堆栈的某一端来访问。一个堆栈类似于放在橱柜中的一叠盘子：正常情况下，只能往盘子堆的上面逐一堆放盘子，也只能从盘子堆的顶部逐次取走盘子。因为这个理由，堆栈常常被称为后进先出（last-in-first-out）的结构。在本书结尾部分，附录 A 给出了各种不同数据结构的简要介绍。

如果我们对数据访问方式加以限制，可以利用主存储器中的某些存储单元来模仿一个堆栈结构。例如，假定把从 0000 到 00FF 的存储器单元用作一个堆栈，将 0000 单元当作堆栈顶。将数据加入（简称压入，pushing）堆栈必须从堆栈顶部开始，而将数据移出（简称弹出，popping）堆栈也必须从堆栈顶部开始。例如，如果要把数值 2 压入堆栈，数值 2 将首先会被放到 0000 单元。如果要继续压入数值 6，则数字 2 会被放到 0001 单元，而将数值 6 放入 0000 单元。如果执行堆栈弹出操作，则数值 6 会首先被移出。计算机使用一个堆栈指针（stack pointer）来跟踪记录应该被压入或弹出项的存放位置。

MARIE 具备现代计算机体系结构的许多特性。但是，由于 MARIE 过于简单，还不能对现代计算机系统的这些特性进行非常精确的描述。在接下来的两节内容中，我们将介绍两种现代的计算机体系结构，以便更好地阐明现代计算机体系结构的各种特性。当然，这里不再讨论按照 Leonardo da Vinci 思想设计的简单的 MARIE。首先要介绍 Intel 体系结构（包括 x86 系列和奔腾系列 CPU），然后再讨论 MIPS 体系结构。之所以选择介绍这两种体系结构，是因为尽管它们在某些方面有相似之处。但是，从本质上来说，这两种体系结构是建立在不同的设计思想的基础上的。Intel 体系结构中的 x86 系列 CPU 被称为 CISC（复杂指令集计算机，Complex Instruction Set Computer），而 Intel 的奔腾系列 CPU 和 MIPS 体系结构则是 RISC（精简指令集计算机，Reduced Instruction Set Computer）的范例。

CISC 机器具有数目庞大、长度各异和设计复杂的指令系统。其中大多数指令非常复杂，单一指令的执行常常需要多个操作才能完成。例如，这些操作可能是由单一（single）汇编语言指令构成循环结构。CISC 机器所遇到的基本问题是，这些复杂的 CISC 指令的一个小的子集就可能显著减慢 CPU 的运行速度。于是，设计人员们又决定重新采用不那么复杂的体系结构，并且对一些小的但是完整的指令集实行硬连线，使指令执行的速度变得非常快。这就是说，计算机是利用编译器来为指令系统（ISA）生成有效代码。采用这种设计思想的机器称为 RISC 机器。

RISC 在某种程度上是用词不当。RISC 结构的指令的数目的确是减少了。但是，RISC 机器的主要目的是简化指令，使指令的执行速度更快。在 RISC 系统中，每条指令只执行一个操作，所有指令的长度相同，只是格式上有少许差别。并且，所有的算术运算都在寄存器之间执行，存储器中的数据不能用作操作数。自从 1982 年以来，所有新设计的指令系统基本上都属于 RISC 结构，或者是 CISC 和 RISC 的某种组合。第 9 章将详细讨论 CISC 和 RISC。

4.8.1 Intel 体系结构

Intel 公司生产了多种不同体系结构的 CPU 产品，可能读者比较熟悉其中的某些体系结构。Intel 公司第一个非常流行的芯片是 8086 芯片，于 1979 年推出，并且成功地应用在 IBM PC 计算机上。8086 可以处理 16 位数据，20 位地址，并且可以寻址 1M 字节的内存。8086 还有一个近亲产品，8 位的 8088 芯片，应用在许多 PC 机上可以降低成本。8086 CPU 从功能上分为两部分：执行单元（execution unit），包含通用寄存器和 ALU；以及总线接口单元（bus interface unit），包含指令队列、段寄存器和指令指针。

8086 配有 4 个 16 位的通用寄存器，分别称为 AX（累加器）、BX（用于扩展寻址的基址寄存

器），CX（计数寄存器）和 DX（数据寄存器）。这些寄存器中的每一个都被分成两个区域：左边的 8 位被指定为高位字节，分别用 AH、BH、CH 和 DH 表示；而右边的 8 位被指定为低位字节，分别用 AL、BL、CL 和 DL 表示。各种 8086 的指令都使用一个专用的寄存器，即指令寄存器。这个指令寄存器也可以作为其他用途。8086 有 3 个指针寄存器：堆栈指针（SP），用来存放指示堆栈地址的偏移量；基址指针（BP），用作压入堆栈的参考地址参量；以及指令指针（IP），用于保持下一条指令的地址。类似于 MARIE 中程序计数器 PC，8086 还有 2 个变址寄存器：SI（源变址）寄存器，用在串操作时的源地址指针；DI（目的变址）寄存器，用作串操作的目的地址指针。8086 还配置了一个状态标志寄存器（status flags register），其中的不同位分别指示各种状态条件，如溢出、奇偶校验、进位和中断等。

8086 的汇编语言程序可以分成不同的段（segment），即一些特殊的程序块或区域，用来保留某些特定的信息类型。其中包括：代码段（存放程序），数据段（存放程序中的数据）和堆栈段（存放程序中的堆栈）。如果要访问任意段内的信息，需要指定相对于该段段首的地址偏移量。因此，需要段指针来存放这些段地址。这些寄存器包括代码段（CS）寄存器，数据段（DS）寄存器和堆栈段（SS）寄存器。8086 还带有第 4 个段寄存器，称为附加段（ES）寄存器，用于某些串操作时的存储器寻址。存储器地址是采用段地址/地址偏移量的寻址方式给出，具体形式为：*xxx*：*yyy*。其中，*xxx* 是段寄存器中的数值，而 *yyy* 是地址偏移量。

1980 年，Intel 公司推出了 8087，8087 在 8086 的机器指令集基础上增加了浮点指令，以及采用一个 80 位宽度的堆栈。在随后的时间里，Intel 公司又发布了许多新的芯片，但是从本质上来说，这些新的芯片与 8086 芯片具有相同的指令系统（ISA）。这些芯片包括：1982 年推出的 80285（可以寻址 16MB 字节）和 1985 年推出的 80386（寻址多达 4GB 字节）。80386 是一个 32 位芯片，也是 8086 系列中的第一个 32 位芯片，通常被称作 IA-32（Intel 体系结构的 32 位 CPU）。在 Intel 公司从 16 位的 80286 进步到 32 位的 80386 时，设计人员需要考虑这些结构能够向下兼容（backward compatible），即在老式的功能较差的处理器上编写的程序可以在新型的较快的处理器上运行。例如，在 80286 上运行的程序同样也可以在 80386 上运行。所以，Intel 在不断改进中还保留了相同的基本体系结构和寄存器组。由于在后续的产品中增加了许多新的特性，所以并不能保证系列芯片可以向前兼容。

依据命名惯例，80386 芯片在寄存器从 16 位变为 32 位后，对寄存器组的命名增加了一个前缀 "E"（E 代表 "扩展"）。这样，原来的 16 位寄存器 AX、BX、CX 和 DX，现在分别被命名为 EAX、EBX、ECX 和 EDX。这种命名惯例同样适用于其他的寄存器组。但是，对于程序员来说，仍然可以使用原来的名称，例如 AX、AL 和 AH 等，来访问原来的寄存器。图 4-16 给出了这种工作方式的示意图，以 AX 寄存器为例。



图 4-16 EAX 寄存器，划分成不同的区域

80386 和 80486 都是 32 位计算机，具有 32 位数据总线。80486 增加了一个高速缓冲存储器（cache），这大大提高了 CPU 的性能（详见第 6 章有关高速缓存和存储器的讨论）。

Intel 公司从奔腾（Pentium）CPU（Intel 公司之所以改变命名方法，从数字 80486 变为 Pentium 是因为数字不能用作注册商标）开始推出的奔腾系列处理器。奔腾处理器具有 32 位寄存器，64 位数据总线，并采用了超标量（superscalar）设计。这就是说，超标量的 CPU 可以有多个 ALU，每个时钟周期可以执行多条机器指令，即多条指令可以并行执行。Pentium Pro CPU 增加了分支预测功能，而 Pentium II 又增加了 MMX 技术来处理多媒体事件，尽管大多数人都认为这种技术算不上是一个巨大成功。Pentium III 则采用浮点指令，增加了对 3D 图形处理的进一步支持。从历史上来说，Intel 处理器采用的都是传统的 CISC 方法。直到最近，Pentium II 和 Pentium III 的设计开始采用一种组合方法，即运用具有 RISC 内核的 CISC 体系结构。这种结构可以将 CISC 指令转换成 RISC 指令。Intel 公司也

在顺应计算机的发展趋势，将 CPU 的设计从 CISC 思想转到 RISC 设计理念。

Intel 公司推出的第 7 代处理器系列是 Intel 的奔腾 4（P4）处理器。P4 处理器和 Intel 以前的处理器有很多方面的不同，其中的许多内容超出了本书的范围。下面简单介绍几点：P4 CPU 的时钟频率为 1.4GHz（或者更高），芯片使用了超过 4200 万个晶体管，并且实现了称为并发（netburst）的微体系结构（microarchitecture）。在此之前，Pentium 系列的处理器都是基于相同的微体系结构设计的，这里使用微体系结构来描述指令集下面的体系结构。P4 处理器的这种新的微结构由下面几个创新技术所组成：超流水线（hyper-pipeline）（有关流水线的内容将在第 5 章讨论），一个 400MHz（或更快的）系统总线，还有多项有关高速缓存和浮点操作的改进技术。这些技术使 P4 处理器在多媒体信息的应用方面的效果非常好。

2001 年 Intel 公司发布的安腾（Itanium）处理器标志着 Intel 的第一个 64 位芯片（IA-64）的诞生。Itanium 处理器包含基于寄存器的编程语言和非常丰富的指令集。Itanium 处理器还使用硬件仿真器来保持 Itanium 与 IA-32/x86 系列的指令集的向下兼容性。Itanium 处理器有 4 个整数处理单元，2 个浮点运算单元。Itanium 的高速缓存容量也有了大幅度提高，并且将高速缓存划分为 4 个不同的级别。本书在第 6 章中将详细介绍有关高速缓存的内容。Itanium 处理器配备 128 位的浮点寄存器和 128 位的整数寄存器，还有其他一些寄存器用于处理分支情况下指令的有效载入。Itanium 处理器至多可以编址 16GB 的主存储器。

体系结构的汇编语言实际上反映了有关体系结构的许多重要信息。为了对 MARIE 的体系结构与 Intel 的体系结构进行比较，我们可以再回到例 4-1 中的 MARIE 程序，即使用一个循环程序实现 5 个数字的相加。现在，使用 x86 系列处理器的汇编语言来对例 4-1 中的程序重写，如例 4-4 所示。这里要特别注意的是，整个程序由一个 Data 段指令和一个 Code 段指令组成。

例 4-4 在 Pentium 机器上运行的程序示例，该例是使用一个循环结构来执行 5 个数值加法运算。

```
        .DATA
Num1    EQU  10                ; Num1 is initialized to 10
        EQU  15                ; Each word following Num1 is initialized
        EQU  20
        EQU  25
        EQU  30
Num     DB   5                 ; Initialize the loop counter
Sum     DB   0                 ; Initialize the Sum

        .CODE
        LEA  EBX, Num1         ; Load the address of Num1 into EBX
        MOV  ECX, Num          ; Set the loop counter
        MOV  EAX, 0            ; Initialize the sum
        MOV  EDI, 0            ; Initialize the offset (of which number to add)
Start:  ADD  EAX, [EBX+EDI *4] ; Add the EBXth number to EAX
        INC  EDI               ; Increment the offset by 1
        DEC  ECX               ; Decrement the loop counter by 1
        JG   Start             ; If counter is greater than 0, return to Start
        MOV  Sum, EAX          ; Store the result in Sum
```

同样也可以使用一个 loop 语句使得上面的程序更易于阅读（但是这样做也使程序看起来不太像 MARIE 的汇编语言程序）。从句法上来说，循环指令类似于跳转指令，其中需要一个标号。将上述程序中的循环过程重写如下：

```
        MOV  ECX, Num              ; Set the counter
Start:  ADD  EAX, [EBX + EDI + 4]
        INC  EDI
        LOOP Start
        MOV  Sum, EAX
```

■

x86 系列处理器的汇编语言中的循环语句类似于 C 语言、C++语言和 Java 语言中的 do ... while 语句结构。所不同的是，这种汇编语言中的循环语句并没有显式的循环变量，而是采用 ECX 寄存器来

保持循环的计数值。每执行一次循环指令，处理器就将 ECX 的数值减 1。然后 CPU 会测试 ECX 的内容，检查 ECX 中的数值是否等于 0。如果 ECX 中的数值不等于 0，就会控制程序跳转到 Start 处，再次执行循环操作。如果 ECX 中的数值等于 0，则终止循环。循环语句是一种可以使程序员的编程变得更为方便的指令类型的例子，但是这类指令并不是计算机操作所必需的指令。

4.8.2 MIPS 体系结构

MIPS 系列 CPU 是目前最成功和设计类型最灵活的 CPU 之一。MIPS R3000、R4000、R5000、R8000 和 R10000 就是其中一些属于 MIPS 科技公司的注册商标。除了应用于计算机系统（例如，Silicon 图形计算机系统）外，MIPS 芯片还广泛应用于嵌入式系统和计算机控制的玩具中（例如，任天堂和索尼公司在它们的许多电子游戏产品中都使用 MIPS 的 CPU）。另外，思科公司是一个非常成功的因特网路由器制造商，同样在它们的产品中使用 MIPS 的 CPU。

第一代 MIPS 指令系统体系结构（ISA）是 MIPS I。接下来是 MIPS II，然后发展到 MIPS V。当前的 MIPS 的 ISA 体系结构称为 MIPS 32（针对 32 位体系结构）和 MIPS 64（针对 64 位体系结构）。这部分内容主要讨论 MIPS 32 结构。值得注意的是，MIPS 技术也有 Intel 技术类似的特点，这就是在 ISA 不断发展的变化过程中，始终保持体系结构的向下兼容性。与 Intel 系列产品一样，每种新 ISA 版本的 MISP 产品都会对包括操作性能和指令系统在内的诸多方面做出有效改进，并且能够有效地处理浮点数值。新型的 MIPS 32 和 MIPS 64 结构的 CPU 在 VLSI 技术和 CPU 组成方面都有了重大进步。相对于传统的体系结构来说，这些进步在提高 CPU 的性能和降低制造成本上带来了非常显著的好处。

就像 Intel 的 IA-32 和 IA-64 一样，MIPS 的 ISA 体系结构具有丰富的内置指令系统，包括算术运算指令、逻辑运算指令、比较运算指令、数据转移指令、分支指令、跳转指令、移位指令和多媒体指令等。MIPS 是一种*装载/存储体系结构*（load/store architecture），这就是说所有的指令（不仅仅是装入和存储指令）都必须使用寄存器作为指令的操作数，不允许使用存储器作为指令的操作数。MIPS 结构有 168 条 32 位的指令，但是许多指令的功能都很类似。例如，MIPS 有 6 条不同的加法指令。这 6 条加法指令都是执行数值相加的运算，其区别只是所使用的操作数和寄存器各不相同。在汇编语言的指令系统中，同一个操作具有多条指令的设计思想是非常普遍的。另外一条通用的指令是 MIPS 中的 NOP（no-op）指令。NOP 指令并不执行任何实际的操作，只是消耗时间而已。在第 5 章中，读者将会看到 NOP 指令应用于在流水线操作的情形。

采用 MIPS 32 体系结构的 CPU 具有 32 个 32 位的通用寄存器，编号从 r0 到 r31。其中的两个寄存器有着特殊用途：r0 通过硬连线连接到一个数值 0，而 r31 是专门用作某些特定指令的默认寄存器。也就是说，这些指令本身并不需要专门指定 r32 这个寄存器。在 MIPS 的汇编语言中，这 32 个通用寄存器分别被指定为 $0、$1、…和 $31。寄存器 1 保留使用，寄存器 26 和 27 用于操作系统的内核。寄存器 28、29 和 30 为指针寄存器。剩下的寄存器可以通过编号来加以引用，这些寄存器的命名惯例如表 4-8 所示。例如，可以使用编号 $8 或 $t0 来引用寄存器 8。

表 4-8 MIPS 32 寄存器命名惯例

命名惯例	寄存器编号	寄存器存放的数值
$v0-$v1	2-3	Results, expressions
$a0-$a3	4-7	Arguments
$t0-$t7	8-15	Temporary values
$s0-$s7	16-23	Saved values
$t8-$t9	24-25	More temporary values

MIPS 32 还有两个专用寄存器，HI 和 LO，用来保存某些整数运算的结果。当然，MIPS 也有一个 PC（程序计数器）寄存器，总共有 3 个专用寄存器。

MIPS 32 有 32 个 32 位的浮点寄存器，用于单精度浮点运算。而双精度浮点数值则按照划分奇-偶数对的方式存储这些寄存器中。还有供浮点单元使用 4 个专用的浮点控制寄存器。

下面，我们使用 MIPS 的汇编语言再来重新编写例 4-1 中的程序。这样，读者可以对这些不同的体系结构进行比较。

例 4-5

```
        . . .
                .data
        # $t0 = sum
        # $t1 = loop counter Ctr
Value:  .word 10, 15,20,25,30
        Sum = 0
        Ctr = 5
        .text
        .global main          # declaration of main as a global variable
main:   lw $t0, Sum           # Initialize register containing sum to zero
        lw $t1, Ctr           # Copy Ctr value to register
        la $t2, value         # $t2 is a pointer to current value
while:  blez $t1, end_while   # Done with loop if counter <= 0
        lw $t3, 0($t2)        # Load value offset of 0 from pointer
        add $t0, $t0, $t3     # Add value to sum
        addi $t2, $t2, 4      # Go to next data value
        sub $t1, $t1, 1       # Decrement Ctr
        b while               # Return to top of loop
        la $t4, sum           # Load the address of sum into register
        sw $t0, 0($t4)        # Write the sum into memory location sum
        . . .
```

■

这种程序类似于 Intel 中使用的程序代码。循环计数器的数值被复制到一个寄存器中，每次循环迭代寄存器中的数值自动减 1。同时，CPU 会检查寄存器中的数值是否小于或等于 0。在 MIPS 体系结构中，寄存器的命名看起来有些复杂。但是，如果理解了这种命名惯例，实际上使用起来也很方便。

如果读者对编写 MIPS 的程序有兴趣，但又苦于找不到一台 MIPS 机器的话，可以使用某些 MIPS 仿真器。现在最流行的 MIPS 仿真器是 SPIM，它是一种功能完备的仿真器，可以运行 MIPS R2000/R3000 的汇编语言程序。SPIM 还提供了一个简单的程序调试器，并且几乎可以执行 MIPS 汇编指令集中的全部指令。SPIM 的程序包包含源代码和一整套的管理文件。这个程序包可以运行于 Unix（包括 Linux）、Windows（PC）、Windows（DOS）和 Macintosh 系统。如果读者要了解进一步的信息，可以参阅本章结尾处的参考文献。

如果仔细研究例 4-1、例 4-4 和例 4-5，不难发现这些程序中的指令非常相似。除了调用寄存器的方式不同和命名不同外，其中包括的各种操作基本上是相同的。有些汇编程序语言具有比较大的指令集，较大的指令集可以使程序员在对不同的算法进行编码的过程中有更多的选择。但是，正如我们在 MARIE 例子中所见到的，一个大的指令集并不是完成计算机的各种操作所必需的。

本章小结

本章阐述了一个简单的计算机体系结构 MARIE。学习简单的 MARIE，非常有助于理解计算机的取指-译码-执行周期和计算机的实际工作原理。这个简单的体系结构具备一套自己指令系统（ISA）和汇编语言。本书详细介绍了指令集和汇编语言两者之间的相互关系，并且可以利用这些指令集和汇编语言为 MARIE 编写各种应用程序。

CPU 是各种计算机的主要核心部件。CPU 由数据通路和控制单元组成。数据通路包括若干个寄存器和一个由总线连接起来的 ALU。控制单元负责对各种操作和数据的移动进行排序，并产生相应的定时信号。计算机中的所有部件都要利用定时信号来实现同步工作。而输入/输出子系统则负责为计算机获取各种数据信息或将数据返回用户。

MARIE是一个专门设计的非常简单的计算机体系结构。利用MARIE模型可以清晰地阐明本章中引入的各种基本概念，而又不会陷入太多的技术细节中。MARIE机器采用16位的指令，并有4K的16位字的主存储器和7个寄存器。其中，只有一个通用寄存器，即累加器AC。MARIE的指令使用其中的4位作为操作码，而剩下的其他12位则表示地址。本章还引入了寄存器传输表示法作为一种符号表示方法，可以在寄存器层次上检查每条指令所执行的具体操作细节。

计算机运行程序所遵从的基本步骤是：取指-译码-执行的循环过程。首先要进行提取指令的操作，然后对指令译码，提取指令所需要的各种操作数，最后执行指令。中断需要在这种循环过程的开始处进行处理，中断处理完毕后，计算机会返回正常的取指-译码-执行周期。

机器语言由一系列二进制数字组成，这些二进制数字所代表的就是可执行的各种机器指令。而汇编语言使用助记符号指令，这些助记符号指令代表的是对应的机器语言程序中的数字数据。汇编语言是一种编程语言，但是它并不能为程序员提供大量的数据类型或指令。汇编语言程序所代表的是一种低级（面向机器）的编程方法。

从某种意义上来说，利用MARIE的汇编语言进行编程是件非常乏味的事情。不难看出，大多数的分支程序结构都必须由程序员利用跳转语句和分支转移语句来显式地进行程序实现。从汇编语言到高级语言（如C++语言或Ada语言）还有相当大的一段距离。因此，编译程序（编译器）就成为一个很好的中间步骤，利用编译程序可以将源代码转换成一些机器能够识别和理解的指令。这里的重点不是介绍汇编语言，本书的目的也不是为了让读者可以很快掌握汇编语言，并成为汇编语言的程序员。相反，这里引入汇编指令是为了帮助读者更好地理解计算机的体系结构，以及指令系统和体系结构之间的相互关系。汇编语言的知识也有助于理解各种高级语言（如C++、Java或Ada等语言）程序的执行过程中，在机器层上所发生的各种具体操作过程。编写x86系列和MIPS系列体系结构的汇编语言程序要比编写MARIE汇编语言程序容易一些。但是，相对于高级语言来说，编写和调试汇编语言程序还是要困难得多。

本章介绍了Intel和MIPS的汇编语言和体系结构，但没有涉及任何过程的细节。引入这两种体系结构有如下的两点理由：第一，对不同的体系结构进行比较分析是一件令人感兴趣的事情，也是学习计算机组成和体系结构非常必要的。我们先从非常简单的体系结构出发，进而介绍一些更加复杂和棘手的体系结构。通过比较研究，可以更加清晰地了解这些体系结构之间的异同。第二，虽然从形式上看起来，Intel和MIPS的汇编语言与MARIE的汇编语言有很大不同，但是在实际的功能上它们却是完全等效的。它们都包含一些访问存储器和寄存器的指令，移动数据的指令，执行算术运算和逻辑运算的指令，以及一些分支转移指令。MARIE的指令集非常简单，而且缺少许多Intel和MIPS指令集中的所谓“编程用户友好”的指令。Intel和MIPS的体系结构还比MARIE结构拥有更多的寄存器。除了在指令的数目和寄存器的数目方面存在着较大的差别外，这三种体系结构的语言功能几乎是完全一样的。

深入阅读

在本教材的主页可以下载一个MARIE的汇编语言仿真器。利用仿真器可以编译和运行MARIE的程序。

有关CPU组成原理和指令系统体系结构的细节资料可以参阅Tanenbaum（1999）和Stallings（2000）的著作。而Mano（1991）的著作中包含大量有关微编程体系结构的示例。Wilkes（1958）的文章也是一篇很好的有关微编程的参考资料。

关于Intel汇编语言编程的更多资料可以查阅Abel（2001）、Dandamudi（1998）和Jones（1997）的书籍。Jones的著作所采用的是一种直接和简单的方法介绍汇编语言编程。这三本书籍的介绍都十分详尽和深入。如果读者对其他汇编语言感兴趣的话，可以参考Struble（1975）：IBM汇编语言；Gill、Corwin和Logar（1987）：Motorola汇编语言；和Sun Microsysytems（1992）：SPARC汇编语言。而要了解嵌入式系统的汇编语言可以尝试阅读Williams（2000）的书籍。

如果想了解有关MIPS汇编语言编程的内容，Patterson和Hennessy（1997）的著作是一本很好的参考书，并且书中备有单独的附录，提供了许多有价值的信息。Donovan（1972）的著作同样也很好地介绍了

MIPS 的编程环境。而 Kane 和 Heinrich（1992）的著作无疑是一本有关 MIPS 计算机的指令系统和汇编语言程序的经典教科书。读者也可在 MIPS 公司的主页找到大量的有用信息。

要了解更多有关 Intel 体系结构的内容，请参考 Alpert 和 Avnon（1993），Brey（2003），以及 Dulon（1998）的书籍和文章。Shanley（1998）的著作可能是有关 Pentium 体系结构的最好的书籍之一。而有关 Motorola 体系结构、Ultrasparc 体系结构和 Alpha 体系结构，分别在 Circello（1995）、Horel 和 Lauterbach（1999），以及 McLellan（1995）的文章中有相关的讨论。对于高级的计算机体系结构，一般性的介绍可以参阅 Tabak（1991）的著作。

要学习更多有关 MIPS 体系结构的 SPIM 仿真器的知识，可参考 Patterson 和 Hennessy（1997）的著作，或直接浏览 SPIM 的主页。Waldron（1999）的书籍是一本很好的有关 RISC 汇编语言编程和 MIPS 的导论。

参考文献

Abel, Peter. *IBM PC Assembly Language and Programming*, 5th ed. Upper Saddle River, NJ: Prentice Hall, 2001.

Alpert, D., & Avnon, D. "Architecture of the Pentium Microprocessor," *IEEE Micro 13*:3, April 1993, pp. 11–21.

Brey, B. *Intel Microprocessors 8086/8088, 80186/80188, 80286, 80386, 80486 Pentium, and Pentium Pro Processor, Pentium II, Pentium III, and Pentium IV: Architecture, Programming , and Interfacing*, 6th ed. Englewood Cliffs, NJ: Prentice Hall, 2003.

Circello, J., Edgington, G., McCarthy, D., Gay, J., Schimke, D., Sullivan, S., Duerden, R., Hinds, C., Marquette, D., Sood, L., Crouch, A., & Chow, D. "The Superscalar Architecture of the MC68060," *IEEE Micro 15*:2, April 1995, pp. 10–21.

Dandamudi, S. P. *Introduction to Assembly Language Programming—From 8086 to Pentium Processors*, New York: Springer Verlag, 1998.

Donovan, J. J. *Systems Programming*, New York: McGraw-Hill, 1972.

Dulon, C. "The IA-64 Architecture at Work," *COMPUTER 31*:7, July 1998, pp. 24–32.

Gill, A., Corwin, E., & Logar, A. *Assembly Language Programming for the 68000*, Upper Saddle River, NJ: Prentice Hall, 1987.

Goodman, J., & Miller, K. *A Programmer's View of Computer Architecture*, Philadelphia: Saunders College Publishing, 1993.

Horel, T., & Lauterbach, G. "UltraSPARC III: Designing Third Generation 64-Bit Performance," *IEEE Micro 19*:3, May/June 1999, pp. 73–85.

Jones, W. *Assembly Language for the IBM PC Family*, 2nd ed. El Granada, CA: Scott Jones, Inc., 1997

Kane, G., & Heinrich, J., MIPS RISC Architecture, 2nd ed. Englewood Cliffs, NJ: Prentice Hall, 1992

Mano, Morris. *Digital Design*, 2nd ed., Upper Saddle River, NJ: Prentice Hall, 1991.

McLellan, E. "The Alpha AXP Architecture and 21164 Alpha Microprocessor," *IEEE Micro 15*:2, April 1995, pp. 33–43.

MIPS home page: *www.mips.com*

Patterson, D. A., & Hennessy, J. L. *Computer Organization and Design: The Hardware/Software Interface*, 2nd ed. San Mateo, CA: Morgan Kaufmann, 1997.

Samaras, W. A., Cherukuri, N., & Venkataraman, S. "The IA-64 Itanium Processor Cartridge," *IEEE Micro 21*:1, Jan/Feb 2001, pp. 82–89.

Shanley, T. *Pentium Pro and Pentium II System Architecture*. Reading, MA: Addison-Wesley, 1998.

SPARC International, Inc., *The SPARC Architecture Manual: Version 9*, Upper Saddle River, NJ: Prentice Hall, 1994.

SPIM home page: *www.cs.wisc.edu/~larus/spim.html*

Stallings, W. *Computer Organization and Architecture*, 5th ed. New York: Macmillan Publishing Company, 2000.

Struble, G. W., *Assembler Language Programming: The IBM System/360 and 370*, 2nd ed. Reading, MA: Addison Wesley, 1975.

Tabak, D. *Advanced Microprocessors*, New York, NY: McGraw-Hill, 1991.

Tanenbaum, Andrew. *Structured Computer Organization*, 4th ed. Upper Saddle River, NJ: Prentice Hall, 1999.

Waldron, John. *Introduction to RISC Assembly Language*, Reading, MA: Addison Wesley, 1999.

Wilkes, M. V., Renwick, W., & Wheeler, D. J. "The Design of the Control Unit of an Electronic Digital Computer," *Proceedings of IEEE*, 105, Part B, No. 20, 1958, pp. 121–128, 1958.

Williams, Al. *Microcontroller Projects with Basic Stamps*, Gilroy, CA: R&D Books, 2000.

基本概念和术语复习

1. CPU 的主要功能是什么？
2. 数据通路的作用是什么？
3. 控制单元的任务是什么？
4. 寄存器安置在什么位置？有何种不同类型的寄存器？
5. ALU 怎样知道要执行哪一种功能？
6. 为什么总线通常是计算机内部通信的瓶颈？
7. 点对点的总线和多点总线之间有何区别？
8. 为什么总线协议非常重要？
9. 解释数据总线、地址总线和控制总线之间的差别。
10. 什么是总线周期？
11. 举出三种不同类型的总线，并指出这些总线在计算机中的位置。
12. 同步总线和异步总线之间有什么区别？
13. 总线仲裁的 4 种方式是什么？
14. 解释时钟周期和时钟频率之间的区别。
15. 如何区别系统时钟和总线时钟？
16. I/O 接口的主要功能是什么？
17. 解释存储器映射的 I/O 和基于指令的 I/O 之间的区别。
18. 一个字节和一个字有何不同？如何区分它们？
19. 解释按字节编址和按字编址的区别。
20. 为什么地址的对齐非常重要？
21. 列出并解释两种不同类型的存储器交互存储方式，并说明它们之间的差别。
22. 解释中断的工作原理，并列举 4 种不同类型的中断方式。
23. 可屏蔽中断和不可屏蔽中断之间有何不同？
24. 如果 MARIE 有 4K 字的主存储器，为什么地址需要有 12 位？
25. 说明 MARIE 的所有寄存器的功能。
26. 什么是操作码？
27. 阐述 MARIE 的每条指令的工作方式。
28. 机器语言和汇编语言有何区别？它们之间是否存在一一对应的转换关系（一条汇编语言指令等于一条机器语言指令）？
29. RTN 有什么重要性？
30. 一个微操作与一条机器指令是否是同一回事？
31. 微操作与常规的汇编语言指令有何区别？
32. 说明取指-译码-执行周期的各个步骤。
33. 中断驱动的 I/O 是如何工作的？

34. 解释编译程序的工作原理，包括怎样产生符号表，对源代码和目的代码所执行的操作，以及怎样处理标号。
35. 什么是嵌入式系统？它与常规的计算机系统有何区别？
36. 跟踪描述例 4-1 中程序的细节执行过程（类似于图 4-13）。
37. 解释硬连线控制和微编程控制的区别。
38. 什么是堆栈？堆栈对于编程有什么重要性？
39. 比较 CISC 机器和 RISC 机器的异同。
40. Intel 体系结构和 MIPS 体系结构有什么区别？
41. 分别列举 4 种 Intel 处理器和 MIPS 处理器。

练习题

1. CPU 的主要功能是什么？
2. 阐述 CPU 对中断的响应过程，包括 CPU 检测中断的方法，如何处理中断和中断服务完成后 CPU 的反应。

◆3. 如果对一个 2M×32 的存储器按下列方式进行编址，需要多少位地址？

◆a）存储器按字节编址？

◆b）存储器按字编址？

4. 如果对一个 4M×16 的主存储器（内存）按下列方式进行编址，需要多少位地址？

a）主存储器按字节编址？

b）主存储器按字编址？

5. 如果对一个 1M×8 的主存储器（内存）按下列方式进行编址，需要多少位地址？

a）主存储器按字节编址？

b）主存储器按字编址？

◆6. 如果采用 256KB×8 的 RAM 芯片来构建一个 2 M×16 的主存储器，并且存储器按字编址。

◆a）共需要多少 RAM 芯片？

◆b）每个字需要多少 RAM 芯片？

◆c）每个 RAM 芯片需要多少个地址位？

◆d）这个存储器有多少组？

◆e）所有的存储器需要多少地址位？

◆f）如果使用高位交互存储方式，地址 14（十六进制数 E）的存储单元位于什么位置？

◆g）对于低位交互存储方式，重做练习 f）。

7. 假设利用 512K×8 的 RAM 芯片构建一个 16 M×16 的存储器，重做练习 6。
8. 一个数字计算机的存储单元的每个字有 24 位。指令集由 150 个不同的操作组成。所有的指令都有一个操作代码部分（操作码）和一个地址部分（只涉及一个地址）。每条指令都存储于一个存储器字中。请问：

a）操作码需要多少位？

b）指令的地址部分还留下多少位？

c）允许的最大存储器容量是多少？

d）在一个存储器字中可能的最大的无符号二进制数是多少？

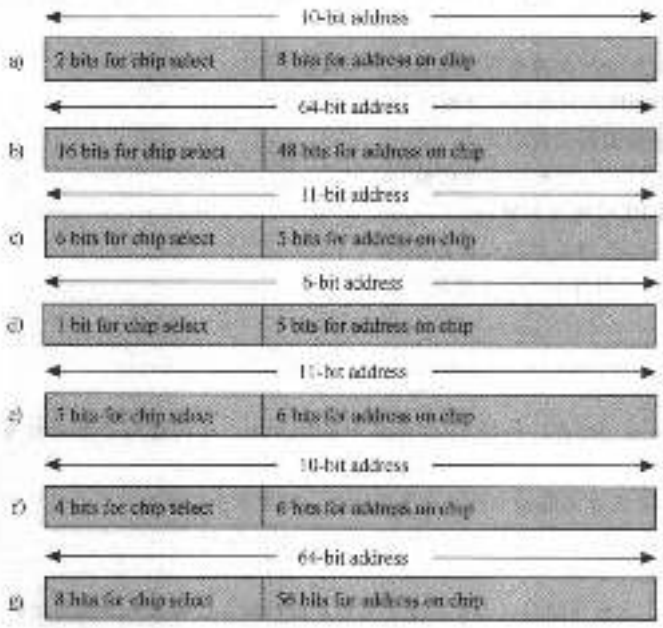
9. 假设有一个 2^{20} 字节的存储器：

◆a）如果存储器按字节编址，那么最低和最高的地址分别是多少？

◆b）如果采用一个 16 位字和按字编址，那么最低和最高的地址分别是多少？

c）如果采用一个 32 位字和按字编址，那么最低和最高的地址分别是多少？

10. 已知一个 2048 字节的存储器，由几个 64×8 RAM 的芯片组成。并且假定存储器是按字节编址的。请问：下列 7 个图中哪个图指示的是使用地址位的正确方式？为什么？



11. 阐述 CPU 的取指-译码-执行周期的各个步骤，请详细说明各个寄存器的变化过程。

◆12. 解释在 MARIE 中，为什么 MAR 寄存器是 12 位，而 AC 寄存器的宽度是 16 位？

13. 写出下列程序对应的十六进制数代码（并对程序进行编译）。

```
Label   Hex Address     Instruction
        100             Load A
        101             Add One
        102             Jump S1
S2,     103             Add One
        104             Store A
        105             Halt
S1,     106             Add A
        107             Jump S2
A,      108             HEX 0023
One,    109             HEX 0001
```

◆14. 对于上述程序，写出符号表中的内容。

15. 已知本章介绍的 MARIE 指令集：

a) 解密下列 MARIE 机器语言指令（利用等效的汇编语言写出这些指令）

◆i) 0010000000000111

ii) 1001000000001011

iii) 0011000000001001

b) 采用 MARIE 的汇编语言书写如下代码段：

```
if X > 1 then
  Y := X * X;
  X := 0;
endif;
  Y := Y + 1;
```

c) 判断一下：如果右如下代码片断（执行某个子程序的操作）在 MARIE 机器上运行，可能会出现的什么潜在问题（可能多于一个问题）？这个子程序假设了要传递的参量存放在 AC 中，并且应该将该

参量值加倍。程序的主体部分包括一个调用子程序的例子。可以假定这个程序片断是某个大程序的一部分。

```
Main,  Load   X
       Jump   Subl
Sret,         Store X
       . . .
Subl,  Add    X
       Jump   Sret
```

16. 编写一个 MARIE 程序，对表达式 A ×B + C× D进行估值。
17. 利用 MARIE 的汇编语言编写如下的代码段：

```
X := 1;
while X < 10 do
  X := X + 1;
endwhile;
```

18. 利用 MARIE 的汇编语言编写如下的代码段：

```
Sum := 0;
for X := 1 to 10 do
  Sum := Sum + X;
```

19. 采用重复相加的方式，编写一个包含循环过程的 MARIE 程序，计算两个正数的乘法。例如：乘法3×6，程序就要执行 6 次加法，或 3 + 3 + 3 + 3 + 3 +3。
20. 编写一个 MARIE 的子程序，实现两个数字相减。
21. 使用更多寄存器有利于提高 CPU 的工作速度，因为增加寄存器的数目可以减少程序访问存储器的总次数。列举一个算法例子来证明上述观点。首先，使用 MARIE 确定需要访问存储器的次数，并且利用两个寄存器来保存来自存储器的数据值（AC 和 MBR）。然后，对于一个采用多于三个寄存器来保存存储器数据值的处理器，执行与上面相同的算法计算。
22. MARIE 将一个子程序的返回地址存放到存储器中，存储单元的位置由跳转-存储（jump-and-store）指令来指定。在某些体系结构中，这个返回地址存放在一个寄存器内，大多数情况下是存放在堆栈中。请问：这两种方法中哪种方法处理循环递归效果最好？并解释理由。
23. 跟踪描述例 4-2 中程序的细节执行过程（类似于图 4-13）。
24. 跟踪描述例 4-3 中程序的细节执行过程（类似于图 4-13）。
25. 假设添加下面的一条指令到 MARIE 的指令集中：

    IncSZ Operand

    这条指令的作用对有效地址“Operand”进行增量操作，并且如果新近的增量值等于 0，程序计数器就加 1。基本上，我们是对操作数进行增量处理，并且如果新值等于 0，则跳过下一条指令。说明怎样使用 RTN 来编写这条指令。
26. 在 CPU 和存储器之间，采用的是同步总线，还是异步总线？为什么？
27. *挑选一种体系结构（不属于本章介绍的体系结构）。经过研究揭示出这个体系结构是怎样来处理本章引入的基本概念的，就像 Intel 和 MIPS 体系结构所处理的那样。

是非题

1. 如果一台计算机采用硬连线控制，微程序将决定机器的指令集。这个指令集永远都不能改变，除非对这个体系结构进行重新设计。
2. 一条分支指令是通过改变 PC 来改变信息流的。
3. 寄存器是 CPU 内部本身的存储单元。
4. 一个两次通读（two-pass）的编译程序，一般是在第一次阅读源程序时生成一个符号表，并且在第二次阅读时完成从汇编语言到机器语言的全部转换任务。

5. MARIE 中的 MAR、MBR、PC 和 IR 寄存器可以用来保存任意的数据值。
6. MARIE 中有一个公用总线的配置，这意味着有许多实体共享这条总线。
7. 一个编译器是一个程序，这个程序接受一个符号语言，并生成一个等效的二进制机器语言，在汇编语言的源程序和机器语言的目标程序之间产生一一对应的关系。
8. 如果一台计算机采用微程序控制，那么微程序决定机器的指令集。

每个程序中都至少存在一个程序缺陷，而且每个程序都可以精简为至少一条指令。由此推断，每个程序都可以被缩减成一条无法运行的指令。

——无名氏

第5章　指令系统体系结构概览

5.1　概述

计算机指令由操作码和操作数组成。操作码指定要执行的操作类型，而操作数指出数据所处的寄存器和内存单元。既然可以使用像C++、Java和Ada这样一些高级语言，为什么还需要学习计算机的机器指令呢？在使用高级语言编程时，我们很少会意识到第4章（或者本章）所讨论的一些概念。原因是高级语言隐藏了许多有关计算机体系结构的细节，利用高级语言的编程人员通常看不见这些有关体系结构的细节。计算机公司通常喜欢雇用一些具有汇编语言背景的程序员，并不是因为他们需要汇编语言的程序员，而是因为他们需要一些理解计算机体系结构的程序员，以便使编写的程序更高效，运行更加顺畅。

本章会对上一章的内容做进一步的展开，深入讨论计算机的指令系统体系结构。将介绍各种不同的指令类型和操作数类型，以及指令如何访问内存中的数据。不同体系结构的计算机系统具有不同的指令系统。学习指令系统的设计构成和指令的工作方式，有助于理解更复杂的计算机体系结构的具体细节。

5.2　指令格式

每条计算机的指令都有一个操作码和0个或多个操作数。第4章中的MARIE的指令长度为16位，且最多只有一个操作码。指令系统的编码方式是多种多样的。随着体系结构的不同，机器指令使用的二进制数的位数也可能不同（最常用的机器指令长度是16位、32位和64位），每条指令允许使用的操作数的个数不同，指令的类型和指令处理的操作数的类型也可能不同。更具体地说，各种指令集可能在特征上存在如下差别：

- 操作数在CPU中的存储方式（数据可以存储在堆栈结构或寄存器中）
- 指令直接作用的操作数的数目（最常用的操作数个数为0、1、2和3）
- 操作数的位置（如果简单地按照指令适用的操作数的组合，则各种指令可以分为寄存器-寄存器类型、寄存器-存储器类型或存储器-存储器类型）
- 操作（不仅包括操作的类型，而且指出指令是否可以访问存储器）
- 操作数的类型和长度（操作数可能是地址、数字或字符）

5.2.1　指令系统的设计

在设计计算机的体系结构时，必须首先确定计算机指令系统的格式。各种计算机系统设计中所选择的指令体系结构通常会有很大差别，原因是不同的体系结构必须配备不同的指令系统。精心设计的指令系统和体系结构通常可以持续使用许多年。因此，在计算机的设计阶段所做出的各种决定会具有较长远的影响。

指令系统体系结构（ISA）的效能可以用下面几个因素来衡量：（1）程序执行指令时占用内存空间的大小；（2）指令系统的复杂程度，主要指指令执行所需要的译码数量和指令所执行的任务的复杂性；（3）指令的长度；（4）指令系统中指令的总数目。设计指令系统时需要考虑的问题包括：

- 指令一般越短越好，因为较短的指令占用较少的内存空间，并且提取指令的速度也会更快。但是，采用短指令会限制指令的数量。因为指令的数量受到指令中能够进行编码的二进制数的位

数的限制。同样，较短的指令也会极大地限制操作数的大小和数量。

- 固定长度的指令的译码相对比较容易，但却浪费空间。
- 存储器的组成形式会影响指令的格式。例如，如果存储器有16位或32位字，而且不是按字节编址的，那么就很难访问一个单一字符。基于这种理由，甚至有些16位、32位或64位字的机器通常也是按字节编址设计的，也就是说每个字节都有一个唯一的地址，即使这些机器的字都大于一个字节。
- 固定长度的指令系统并不表示必须使用固定数量的操作数。我们可以设计一个指令总长度固定的指令系统体系结构（ISA），但是却可以允许操作数域的位数根据需要而改变。这种指令系统体系结构称为扩展操作码（expanding opcode），将在第5.2.5节详细讨论。
- 存在多种不同类型的寻址方法。第4章，MARIE的设计中就采用了两种寻址方法：直接寻址和间接寻址。在本章中，读者将会发现还有更多的寻址方法。
- 如果机器的字由多个字节组成，就需要考虑组成字的字节是按照怎样的次序存储到按字节编址的机器存储器中？例如，最低位的字节到底是存放在最高字节地址，还是最低字节地址。这是一个有关小端和大端的位序问题，将留在下一部分讨论。
- 设计的体系结构需要多少个寄存器且这些寄存器应该如何进行组织安排？操作数如何存放在CPU中？

小端和大端的位序、扩展操作码和CPU寄存器的组成将在下面的内容作深入讨论。与此同时，我们也会介绍上面列出的其他一些设计问题。

5.2.2 小端和大端的位序问题

术语——位端（endian）指的是计算机体系结构中的“位序”（byte order）。或者说，位端是指在计算机中存储一个多字节数据元素时，各个字节的排列方式。事实上，当今所有的计算机体系结构都是按字节编址的。因此，必须对存储多个单一字节的信息的方法制定一个标准。例如，某些计算机在存储一个2字节整数时，将最低位的字节首先存放到低位地址，然后再将最高位的字节存放到高位地址。这样一来，位于较低地址的字节就是数据信息的最低位。采用这种方式的机器称为小端（little endian）机器。而其他的一些机器在存储同样的2字节整数时，则是先将最高位的字节存放到低位地址，然后再存放最低位的字节。这种类型的机器称为大端（big endian）机器。原因是他们将最高位的字节存放在低位地址。大部分的UNIX计算机都是大端机器，而大多数的PC机则是小端机器。大部分新式的RISC体系结构也都是大端机器。

小端和大端这两个术语来自小说《格列佛游记》（Gulliver's Travels）。读者也许还记得，在这个故事中，小人国里的小人分成两大阵营：其中一部分小人颠破鸡蛋“大”的一头吃鸡蛋（big endian），而另一部分小人则颠破鸡蛋“小”的一头吃鸡蛋（little endian）。CPU制造商同样也分成两类。例如，Intel总是使用“小端”方式来设计机器，而Motorola则总是采用“大端”方式。值得一提的是，有些CPU既能处理小端问题也能处理大端问题。

例如，考虑如下所示的一个由4字节组成的整数：



在小端机器中，这个整数在存储器里面的安排顺序为：

```
Base Address + 0 = Byte0
Base Address + 1 = Byte1
Base Address + 2 = Byte2
Base Address + 3 = Byte3
```

而在一个大端机器中，这个长整数将按如下的方式存储：

```
Base Address + 0 = Byte3
Base Address + 1 = Byte2
Base Address + 2 = Byte1
Base Address + 3 = Byte0
```

假设现在有一个按字节编址的机器，要将一个 32 位 16 进制数值 12345678 存储在地址 0 的存储单元。每个十六进制的数字都需要半个字节，所以一个字节可以保存两个数字。这个十六进制数值将按照如图 5-1 所示的不同方式存储在存储器中，图中阴影部分表示存储器中的实际内容。

编址 ⟶	00	01	10	11
大端位序	12	34	56	78
小端位序	78	56	34	12

图 5-1 十六进制数值 12345678 分别按大端和小端的格式存储

这两种方法各有优缺点，不能断言哪种方法更好一些。对于大多数人来说，大端位序的存储方式更自然些，而且更便于阅读十六进制数编写的程序段。因为最高位的字节首先进行处理，所以总是可以通过检查位于最高位的符号位来判断这个数字的正负（面对于小端位序的存储方式，则必须先要知道数值的长度，然后跳过中间字节找到包含符号信息的字节）大端位序的机器存储整数和字符串时使用相同的次序，并且在某些字符串操作时速度会更快一些。大部分位图映射格式的图像都是采用“最高位在字符串的左边”的变换方法，也就是说对于像素大于一个字节的数据可以直接按照大端位序机器自身体系结构的安排顺序进行处理。而对于小端位序的机器来说，在处理一些较大的图形对象时，其性能上会受到某些限制。原因在于这些小端机器需要不断地反转处理这类图形数据的字节的排列次序。当对采用像霍夫曼（Huffman）和 LZW 这类方案（这些编码方法将在第 7 章讨论）编码的压缩数据进行译码时，如果数据采用的大端位序的存储方式，那么实际的编码字可以被当作进入到某个查询表中的一个索引来使用（编码的情况也是如此）。

当然，大端位序的存储方式也有缺点。如果计算机要进行从 32 位整数地址到 16 位整数地址的转换操作，则要求大端机器执行加法运算。高精度的算术运算在小端机器上会更快和更方便一些。大部分采用大端位序方案的体系结构都不允许计算机字按照非字地址边界的方法来写（存储）数据字。例如，如果一个字由 2 或 4 个字节组成，写数据字时就必须从一个偶数编号的字节地址开始。这样做显然会浪费存储空间。而对于像 Intel 这类的小端位序机器，则允许进行奇数地址的读写。这样可以使得在这类机器上的编程会变得更加方便。假如一个程序员编写了某条指令来读取一个字长度出错的数值。如果这条指令在大端机器上执行时，读取的总是一个错误的数值；而对于小端机器来说，则指令有时候可能会产生一个正确的读取数据。当然，Intel 公司最后也在指令系统中增加了一条指令，用来反转寄存器中的字节的次序。

计算机网络都是采用大端位序的体系结构。这就意味着，如果小端位序机器要将整数数据（例如，网络设备地址）传送到网络上时，必须将这些数据转换成网络所要求的字节次序。同样，小端机器从网络上接收整数数据时，也需要将这些数据转换回它们自己的本地表示形式。

尽管读者可能对有关小端和大端位序的争论不太熟悉，但是这一问题在现在的许多软件应用中却是非常重要的。任何程序在从某个文件中读写数据时，都必须搞清楚所运行的计算机中的字节次序。例如，因为 Windows BMP 图形格式是基于小端机器开发的，所以如果要在大端机器上观看 BMP 图形，用来处理这类图形格式的应用程序就必须首先反转数据的位序。当然，通用软件的设计人员都非常清楚这类的位序问题。例如，应用软件 Adobe Photoshop 使用大端位序，GIF 是小端格式，而 JPEG 属于大端格式；Macpaint 使用大端位序格式，PC Paintbrush 采用小端格式，而 Microsoft 的 RTF 也是使用小端位序结构，而 Sun 光栅文件却属于大端格式。有些应用程序可以同时支持上面的两种格式，例如，Microsoft 的 WAV 文件和 AVI 文件，TIFF 文件，以及 XWD（X Windows Dump）就支持两种格式，通常可以通过将一个标识符编码到文件中来实现。

5.2.3 CPU的内部存储机制：堆栈和寄存器

当存储器的位序确定之后，硬件设计人员需要决定CPU存储数据的方式。CPU的数据存储方式是区分不同指令系统体系结构（ISA）的最基本的方法。这里有三种不同的体系结构可供选择：

1. 堆栈体系结构
2. 累加器体系结构
3. 通用寄存器（GPR）体系结构

堆栈体系结构（stack architecture）的计算机使用一个堆栈来执行各种指令，而且指令的操作数就隐含地存放在堆栈的顶部。按照堆栈体系结构设计的机器通常具有好的编码密度和一个简单的表达式估值模型，但是由于不能对堆栈进行随机访问，使得采用堆栈结构的机器很难产生高效率的编码。*累加器体系结构*（Accumulator architecture）的计算机，比如MARIE，是将其中一个操作数隐含在累加器中。这样可以最大限度地降低机器的内部复杂性，而且允许使用非常短的指令。但是由于累加器只是临时存储，所以这类机器的对存储器的访问非常频繁。*通用寄存器体系结构*（general purpose register architecture）的计算机，采用多个通用寄存器组，这是当今计算机体系结构中最广泛接受的模型。这些寄存器组的访问速度比存储器快得多，非常方便编译器进行处理，并且可以十分有效和高效地使用。另外，由于硬件的价格急剧下降，所以现在可以以较少成本增加大量数目的寄存器。如果存储器的速度快，堆栈体系结构的设计可能是一个好方法；如果存储器的速度慢，通常采用通用寄存器体系结构的设计会比较好。当然，大家有许多理由可以解释为什么最近10年来大部分的计算机都使用通用寄存器体系结构。但是，由于所有的操作数都必须加以命名指定，所以使用寄存器结构会产生较长的指令，导致较长的取指时间和译码时间。对ISA设计人员来说，其中一个非常重要的目标是实现短指令。设计人员在选择某个指令系统体系结构时，应该明确这种体系结构可以在某个特定的环境中的工作效能性能最佳，而且需要对各种可能的方案仔细作出权衡。

通用寄存器体系结构可以根据指令的操作数所处的位置分成三种类型。*存储器-存储器*（memory-memory）体系结构可以有两个或三个操作数位于存储器内，允许有一条执行某种操作而不需要有任何操作数的指令存放在某个寄存器中。*寄存器-存储器*（register-memory）体系结构则是采用混合方式，这其中至少有一个操作数在寄存器中和一个操作数在存储器中。*装入-存储*（load-store）式体系结构则需要在任何对数据的操作执行之前，先将数据装入寄存器中。Intel和Motorola的计算机系统属于寄存器-存储器体系结构；数字仪器公司的VAX计算机的体系结构实行的是存储器-存储器操作；而SPARC、MIPS、ALPHA和PowerPC都是装入-存储式体系结构的计算机。

今天，大部分体系结构都是采用以通用寄存器为基础（GPR-based）设计的。下面，我们讨论用来区分通用寄存器体系结构的有关指令系统的两个主要特性。这两个特性就是指令的操作数的数目和操作数的编址方式。在第5.2.4节中，我们将考虑指令的长度和指令所包含的操作数的数目问题。对于通用寄存器体系结构，一条指令具有两个或三个操作数是很常见的。我们会对这些采用多操作数指令的体系结构与不使用或只使用一个操作数指令的体系结构进行比较研究，然后再讨论有关指令的类型问题。最后在第5.4节，我们将介绍现在的计算机系统中所采用的各种编址方式。

5.2.4 操作数的数目和指令的长度

描述计算机体系结构的传统方法是指定每条指令中所包含的操作数，或者说地址的最大数目。这种做法会对指令自身的长度产生直接影响。MARIE使用的是固定长度的指令，其中包括一个4位的操作码和一个12位的操作数。在现在的计算机体系结构中，指令构成的格式有如下的两种方式：

- *固定长度*（fixed length） 使用这种格式的指令系统会浪费一些存储空间，但是指令执行的速度快。在采用指令层次（instruction-level）的流水线结构时，固定长度的指令系统的性能会更好些。详见第5.5节。

• 可变长度（variable length） 这种指令系统的译码会变得比较复杂，但是却可以节省存储空间。

在实际设计中，通常会考虑一些折衷的方案，采用两到三种不同的指令长度。这样可以有不同的位的组合形式，便于简化指令的区分和译码。当然，指令的长度还必须配合机器字的长度。如果指令的长度严格等于机器字的长度，那么在将指令存储到主存储器时，指令间的对齐问题就会变得非常完美。由于存储器需要编址的缘故，指令总是按照机器字的长度对齐。所以，那些采用实际机器字长度的四分之一、二分之一、两倍或三倍长度的指令会浪费不少的存储空间。可变长度指令的长度显然并不都相同，同样要求按机器字进行对齐，这样也会造成存储空间的损失。

最常用的指令格式包括有零个、一个、两个或三个操作数。第 4 章介绍的 MARIE 机器的某些指令不包含操作数，而剩下的其他指令都带有一个操作数。通常情况下，算术运算或逻辑运算需要两个操作数。但是，如果将累加器当作为一个隐含的操作数，那么这些两个操作数的操作也可以按照一个操作数指令的方式来执行，正如 MARIE 中所执行的那样。如果把最终目标地址作为第三个操作数，那么从设计上就可以将这种思想扩展到三个操作数的情形。同样，使用一个堆栈结构也可以允许有不带操作数（零操作数）的指令。下面是几种常用的指令格式：

• 只有操作码（0 地址）

• 操作码 + 1 个地址（通常只有一个存储器地址）

• 操作码 + 2 个地址（通常是两个寄存器地址，或者是一个寄存器地址加上一个存储器地址）

• 操作码 + 3 个地址（通常是三个寄存器地址，或者是寄存器和存储器的某种组合形式）

对每种体系结构来说，每条指令所允许的最大操作数的数目是有限制的。例如，在 MARIE 中，指令的最大的操作数为 1，当然，某些指令可以没有操作数（Halt 和 Skipcond）。前面已经介绍了，零操作数、一个操作数、两个操作数和三个操作数是最常用的指令格式。对于包含一个操作数、两个操作数，或者甚至三个操作数的指令格式读者比较容易理解。但是，如果一个指令系统体系结构（ISA）开始就完全由零操作数的指令组成，反而会令人感到有些困惑难解。

不带操作数的机器指令必须使用堆栈来执行在逻辑上需要一个或两个操作数的操作（例如，一个 Add 指令）。堆栈是第 4 章引入的一种*后进先出*（last-in，first-out）的数据结构，堆栈中的所有数据插入和删除都必须从堆栈顶部进行，这些内容将在附录 A 中进行详细讨论。由于没有采用通用寄存器，基于堆栈的体系结构将操作数存放在堆栈的顶部，CPU 可以访问堆栈顶部的数据元素。堆栈是计算机体系结构中最重要的数据结构之一。堆栈这种数据结构不但可以为各种复杂计算操作的中间结果提供了一种非常有效的数据存储方式，而且为程序过程调用中的参数传递提供了一种有效方法。另外，堆栈也为存放局部程序模块和界定子程序和变量的界限范围提供了某些非常有用的方法。

在基于堆栈的计算机体系结构中，大部分机器指令都只包含操作码。但是，有些特殊指令，例如往堆栈中加入元素和从堆栈中移除元素的指令，往往会带有一个单一的操作数。堆栈结构需要有一个入栈（push）指令和一个出栈（pop）指令，这两个指令都必须带有一个操作数。Push X 指令表示将地址为 X 的存储器单元内的数据值放到堆栈中。Pop X 表示移走堆栈顶部的元素并将其存放到地址为 X 的存储器单元中。堆栈体系结构中，只有某些特定的指令才允许访问存储器，其他全部指令必须使用堆栈作为指令执行所要求的操作数。

对于要求两个操作数的操作，堆栈结构采用堆栈顶部最上面的两个元素作为操作数。例如，如果要执行加法（Add）指令，CPU 首先将堆栈顶部的两个元素相加，并且把这两个最上面的元素弹出移走，最后再将求和结果压入存放到堆栈的顶部。对于操作次序不能交换的各种操作，同样也是对堆栈顶部最上面的两个数据元素按次序进行。例如减法操作，首先将位于堆栈顶部最上面的第二个元素减去堆栈顶部的第一个元素，然后弹出堆栈顶部最上面的这两个元素，最后的计算结果也同样会被压回到堆栈顶部的单元中。

堆栈的这种组成结构，对于计算采用*反向波兰表示法*（reverse Polish notation，RPN）编写的长算术表达式非常有效。反向波兰表示法将算符放在操作数的后面，*称为后缀表示法*（postfix notation）。

作为对比，还有**前缀表示法**（prefix notation），即把算符放在操作数的前面；**中缀表示法**（infix notation）将算符放在两个操作数的中间。例如：

X + Y 为中缀表示法

+ X Y 是前缀表示法

X Y + 是后缀表示法

所有的算术表达式都可以写成上述三种表示法之一。不过，使用后缀表示法，并加上寄存器堆栈的组合是估值算术表达式最有效的方法。事实上，有些电子计算器（例如惠普公司的产品）就是要求用户使用后缀表示法的方式输入方程式。只要有少许计算器的操作经验，就可以快速地对包含多层嵌套括号结构的长表达式进行估值运算，而无需思考项是如何组合的。

考虑下面的表达式：

$$(X+Y) \times (W-Z) +2$$

使用反向波兰表示法（RPN），上面的表达式变为：

$$XY+WZ -\times 2+$$

注意，在使用反向波兰表示法后，原来用来保持计算次序的括号都被消除了。

为了举例说明有关零操作数、一个操作数、两个操作数和三个操作数这些概念，下面分别利用这4种格式，编写一段对一个算术表达式进行估值的简单程序。

例 5-1 假设对下面的表达式进行估值计算：

$$Z= (X\times Y) + (W\times U)$$

通常，当表达式需要三个操作数时，至少必须有一个操作数是一个寄存器。并且第一个操作数一般为目标操作数。采用3地址指令，估值表达式 Z 的程序代码如下：

```
Mult    R1, X, Y
Mult    R2, W, U
Add     Z, R2, R1
```

如果使用2地址指令，通常其中的一个地址要指定某个寄存器，而另外一个操作数则可以是一个寄存器或存储器地址。在2地址指令结构中，我们很少会将两个操作数都表示为存储器地址。采用2地址指令，这段程序代码变为：

```
Load    R1, X
Mult    R1, Y
Load    R2, W
Mult    R2, U
Add     R1, R2
Store   Z, R1
```

注意，搞清第一个操作数是源操作数还是目标操作数是非常重要的。在上面的指令中，我们假定了第一个操作数为目标操作数。对于需要将程序代码在 Intel 汇编语言和 Motorola 汇编语言之间进行转换的程序员来说，这一点往往会造成麻烦，原因是 Intel 的汇编语言指定第一个操作数为目标操作数，而 Motorola 的汇编语言却把第一个操作数作为源操作数。

如果使用1地址指令（正如 MARIE 中的情形）必须假定一个寄存器（通常是累加器）为默认指令结果的目标操作数。这样，估值 Z 的1地址指令程序代码为：

```
Load    X
Mult    Y
Store   Temp
Load    W
Mult    U
Add     Temp
Store   Z
```

注意，随着每条指令所允许的操作数的数目的减少，完成相同程序功能所需的指令数目将会增加。这就是一个典型的在空间和时间之间进行平衡折衷的例子。通常，采用较短的指令需要编写较长的程序。

下面来看一看，如果在一台使用 0 地址指令的堆栈结构的计算机中，上面的这个程序会是什么样子？基于堆栈结构的计算机的一些指令，如 Add、Subt、Mult 或 Divide，都没有操作数。这里需要的是一个堆栈和对堆栈执行的两种操作：Pop（出栈）和 Push（入栈）。而与堆栈进行通信的操作都必须包含一个地址域，用来指定操作数是从堆栈弹出还是被压入堆栈。除此之外的其他所有操作都是 0 地址指令。Push 指令把操作数存放到堆栈的顶部，而 Pop 指令将堆栈顶部的内容移出来当作操作数。在这种体系结构中，估值上述的方程式所需要编写的程序是最长的。假设这种结构的计算机使用堆栈顶部最上面的两个操作数执行各种算术运算，然后弹出最上面的两个操作数，并将运算结果压入堆栈顶。原来的程序代码就变为：

```
Push    X
Push    Y
Mult
Push    W
Push    U
Mult
Add
Pop     Z
```

■

操作码长度和所使用的操作数的数目肯定会影响到指令的长度。如果指令的长度固定，译码就会变得更方便。但是，如果考虑指令的向下兼容性和灵活性，操作码的长度有时可能需要发生变化，可变长度的操作码存在着可变长度指令所遇到的相同问题。许多设计人员都采用一种折衷的方案，即使用扩展操作码。

5.2.5 扩展操作码

扩展操作码（expanding opcode）代表了一种折衷的方案，就是既要求有尽可能多的操作码的数目，又要求采用尽可能短的操作码，所设计的指令长度也比较短。扩展操作码可以在这两者之间取得最佳平衡。其设计的基本思想是：选用短操作码，而当有需要时可以有某种方法将操作码加长。如果使用短的操作码，就留给操作数大量的指令位，这也就意味着每条指令可以有两个或三个操作数。如果指令不需要操作数，例如像 Halt 这样的一条指令，或者计算机使用一个堆栈进行操作时，指令所有的位都可以用作操作码，因而可以生成许多独特的指令。这样一来，有些指令的操作码比较长，但操作数比较少；而有些指令的操作码比较短，操作数却比较多。

假设一台计算机具有一个 16 位的指令系统和 16 个寄存器。因为这里使用的是一组寄存器而不是单一的累加器（就如 MARIE 机器），所以指令中需要使用 4 个二进制位来唯一指定其中的某个寄存器。因此，这里可以编码产生 16 条具有三个寄存器操作数的指令，也就意味着任何要操作的数据必须首先被装入到某个寄存器中。或者可以使用指令中的 4 位作为操作码，而剩余的 12 位作为一个存储器的地址，就像 MARIE 机器中一样（假定存储器的容量为 4KB 字节）。存储器的访问需要占用 12 位长度，只留下 4 位可以另作他用。但是，如果存储器中所有数据都是首先被装入到寄存器组中的某个寄存器，那么指令可以只使用其中的 4 位来选择这些存放在寄存器内的特殊数据元素（假设有 16 个寄存器）。这两种选择方案如图 5-2 所示。

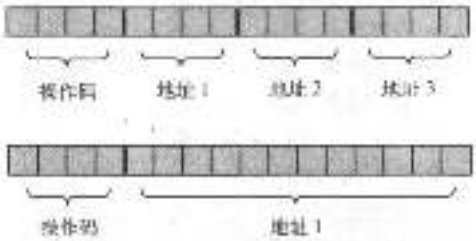


图 5-2 16 位指令格式的两种可能的形式

假设需要编码产生如下指令：

- 15 条 3 地址指令
- 14 条 2 地址的令
- 31 条 1 地址指令
- 16 条 0 地址指令

能否只用 16 位就可以编码产生上述的指令集呢？答案是肯定的，只要采用扩展操作码即可。具体编码方式如下：

```
0000 R1   R2   R3
...                          } 15 3地址编码
1110 R1   R2   R3

1111 0000 R1   R2
...                          } 14 2地址编码
1111 1101 R1   R2

1111 1110 0000 R1
...                          } 31 1地址编码
1111 1111 1110 R1

1111 1111 1111 0000
...                          } 16 0地址编码
1111 1111 1111 1111
```

采用扩展操作码的方法使指令的译码变得更加复杂。原因是在译码的过程中，不仅是检查指令中各个位的组合形式以决定是何种指令，而且需要使用类似于下面的方式对指令进行译码：

```
if (leftmost four bits != 1111 ) {
  Execute appropriate three-address instruction}
else if (leftmost seven bits != 1111 111 ) {
  Execute appropriate two-address instruction}
else if (leftmost twelve bits != 1111 1111 1111 ) {
  Execute appropriate one-address instruction }
else {
  Execute appropriate zero-address instruction
}
```

在上述译码过程的每一个步骤中，都需要有一个备用的编码来指示是否需要检查更多指令位。这是硬件设计过程中又会遇到的另外一种需要进行权衡折衷的例子：操作码的长度和操作数的长度之间的均衡问题。

5.3 指令类型

大多数的计算机指令都对数据执行操作，但是也有些指令的操作对象不是数据。计算机制造商通常将指令归纳为以下几种类型：

- 数据移动
- 算术运算
- 布尔逻辑运算
- 位操作（移位和循环换位）
- 输入/输出（I/O）
- 控制转移
- 专门用途

数据移动指令是最常用的指令。人们常常需要将数据从存储器移到寄存器，从寄存器移到寄存器，以及从寄存器移到存储器等。许多计算机都有多种不同的数据移动指令，其具体应用取决于数据来源和移动目的。例如，指令 MOVER 要求有两个寄存器操作数，而指令 MOVE 却是使用一个寄存器操作数和一个存储器操作数。某些计算机体系结构，例如 RISC 结构，会限制将数据从存储器移出或将数据移到存储器的指令，以加快系统运行速度。许多计算机中有多种载入（load）、存储（store）和移动（move）指令来处理不同大小的数据。例如，某些机器可能会设计 LOADB 指令来处理各种字节数据，而采用 LOADW 指令来处理计算机的数据字。

算术运算指令包括有整数和浮点运算的各种指令。许多机器还对不同大小的数据提供有不同的算术运算指令。与数据移动指令一样，有时计算机也会为用户设计多种不同的算术运算指令，以满足操作数为寄存器和以不同寻址方式访问存储器的各种组合方式的需求。

布尔逻辑指令执行布尔逻辑运算，与算术指令的运算方式非常类似。通常，逻辑运算指令执行的是逻辑与（AND）、逻辑非（NOT）、逻辑或（OR）和异或（XOR）等运算。

位操作指令用来在某个特定的数据字中对一些单独数据位（或者是一些位的集合）进行置位和复位操作。位操作指令包括了各种算术移位指令、逻辑移位指令和循环移位指令，这些移位操作既可以向左进行，也可以向右进行。逻辑移位指令简单地执行将二进制数左移或右移规定的位数，而在移位的反方向将对应数目的 0 移进来。算术移位指令通常用来对二进制数进行乘以 2 或除以 2 的操作，但并不移动最左边的位，因为最左边的位通常表示的是数字的符号。对于算术右移操作，符号位将会被复制到其右边的位置上。对于算术左移操作，数值向左移位，而将 0 移进来，但是符号位不能移动。循环移位指令只是简单地执行将移出的各个位在相反方向依次移进来。例如，对于循环左移 1 位的操作，结果是数据字的各个位依次左移一位，而最左边的一位被移出，循环后变成了最右边的一位。

输入/输出（I/O）指令随着计算机体系结构的不同而有很大区别。处理输入输出的基本方式有：程序控制的 I/O、中断驱动的 I/O 和直接存储器访问（DMA）的 I/O 三种方式。这些内容将在第 7 章详细介绍。

控制指令包括分支转移（branch）、跳过（skip）和进程调用（procedure call）等。分支转移分为条件转移和无条件转移。跳过指令实际上是没有指定地址的分支转移指令。因为不需要操作数，跳过指令通常使用地址域的二进制位来指定一些不同的环境（读者可以回忆 MARIE 中使用的 Skipcond 指令）。进程调用是一些特殊的分支转移指令，它们会自动存储程序返回的地址。不同的机器可能会采用不同的方法存储这个返回地址。有些机器将这个返回地址存放在存储器的某个特定的存储单元内，有些机器把地址存放在某个寄存器中，而其他的一些机器则把返回地址压入到一个堆栈中。至此，读者已经了解了堆栈还可以用作其他目的。

专用指令包括字符串处理的指令、高级语言支持的指令、保护和标志位控制指令，以及高速缓存（cache）指令等。大部分计算机体系结构都提供字符串处理的指令，其中包括字符串操作和搜索。

5.4 寻址

虽然寻址是一个指令设计问题，而且从技术上来说它是指令格式的一部分，但是由于寻址涉及到许多方面的问题，所以这里将寻址这个问题单独列出来讨论。与寻址有关的两个最重要的问题是：可以进行编址的数据类型和各种各样的寻址方式。这里只介绍一些最基本的寻址方式，利用这些基本的寻址方式可以构建一些比较复杂的特殊寻址方式。

5.4.1 数据类型

在讨论如何对数据进行编址前，先来简单介绍一下指令可以访问的各种数据类型。如果指令要引用某个特定的数据类型，必须有相应的硬件对该种类型的数据提供支持。第 2 章讨论了各种不同的数据类型，包括数字和字符。数字数据由整数数值和浮点数值组成。整数可以分为带符号整数和无符号整数，并且可以声明为不同长度的整数。例如，在 C++语言设计中，整数可以是短（short）整数（16

位）、整数（int）（即特定体系结构的机器字长度）或长（long）整数（32位）。而浮点数值的长度可以是32位、64位或128位。正如前面所述，计算机的指令系统中常有一些特殊指令，可以用来处理可变长度的数字数据。例如，指令系统中可能会有一条MOVE指令处理16位整数，而使用另外一条不同的MOVE来处理32位整数。

非数字数据包括字符串、布尔数值和指针。字符串指令一般有复制、移动、搜索或修改等操作。布尔逻辑操作包括AND（逻辑与）、OR（逻辑或）、XOR（异或）和NOT（逻辑非）等操作。指针实际上就是存储器中的存储单元的地址。尽管从本质上来说，它们都是一些数字，但是指针还是被认为是与整数和浮点数字不同的数据类型。MARIE采用直接寻址方式来处理指针这种数据类型。指令系统中采用这种方式的操作数实际上都是指针。对于使用指针的指令，操作数本质上就是一个地址，并且必须被当成一个地址处理。

5.4.2 寻址方式

在第4章中，MARIE的指令有一个12位的操作数域。这个操作数可以按两种方式来解释：12位的二进制数可以表示操作数的存储器地址，也可以是一个指示物理存储器地址的指针。实际上还可以解释成其他不同的内容，这样就可以形成几种不同的*寻址方式*（addressing mode）。寻址方式是指定指令中操作数的位置的方法。一种寻址方式所指定的内容可以是一个常数、一个寄存器或是存储器中的一个存储单元。某些寻址方式可以使用较短的地址，而有些寻址方式指出的就是实际操作数的位置。从动态的角度来看，实际操作数的位置称为操作数的*有效地址*（effective address）。下面讨论几种最基本的寻址方式。

立即寻址

立即寻址（immediate addressing）是指在指令中操作代码后面的数值会被立即引用。也就是说，要操作的数据本身是指令的一部分。例如，如果操作数的寻址方式是立即寻址，指令Load 008的操作就是将数值8直接装入累加器AC中。这里，12位的操作数域并不是指定一个地址，而是表示指令所要求的实际操作数。立即寻址方式的执行速度非常快，因为要加载的数值就包含在指令中。但是，由于编译时要加载的数值是固定的，所以这种寻址方式非常不灵活。

直接寻址

直接寻址（direct addressing）是指在指令中直接指定要引用的数值的存储器地址。例如，如果操作数的寻址方式是直接寻址，指令Load 008的操作就是将存储器地址为008的存储单元中的数值装入累加器AC中。直接寻址方式的操作也很快，因为尽管要加载的数值并不包含在指令中，但是这种直接的地址访问非常快。直接寻址方式比立即寻址方式要灵活得多，因为要加载的数值是存放在给定地址的存储单元中，这个数值是可变的。

寄存器寻址

寄存器寻址（register addressing）方式中，是采用一个寄存器，而不是存储器来指定操作数。这种方式与直接寻址方式非常类似，所不同的只是指令的地址域包含的是一个寄存器的引用，而不是某个存储器的地址。寄存器寻址方式中被指定的寄存器中的内容将用作指令的操作数。

间接寻址

间接寻址（indirect addressing）是一种非常有效的寻址方式，使用起来特别灵活。在间接寻址方式中，地址域中的二进制数用来指定一个存储器地址，该地址中的内容将被用作一个指针。操作数的有效地址是通过访问这个存储器地址来获取的。例如，如果操作数的寻址方式是间接寻址，指令Load 008的操作表示在存储器地址为008的单元中存放的数据值实际上就是所要求的操作数的有效地址。假设在008单元存放的数值为2A0。那么2A0就是需要读取数值的真实地址。执行这条指令的操作之后就将地址为2A0的存储器单元中的内容装入到累加器AC中。

作为间接寻址方式的一种变化形式，操作数域的二进制位也可以用来指定一个寄存器，而不是指定某

个存储器地址。这种方式称为寄存器间接寻址（register indirect addressing）。除了采用一个寄存器来代替某个存储器地址作为指针外，寄存器间接寻址的工作原理与间接寻址方式完全相同。例如，如果指令是 Load R1，并且采用的是寄存器间接寻址方式，那么指令可以在寄存器 R1 中找到所需要的操作数的有效地址。

变址寻址和基址寻址

在变址寻址方式（indexed addressing）中，一个变址寄存器（要么显式地指定，要么隐式指定）用来存储一个偏移量（offset），或称为位移量（displacement），将这个偏移量与操作数相加，就产生了指令所要求的数据的有效地址。例如，如果指令 Load X 中的操作数 X 是采用变址寻址方式编址的，若假定 R1 为变址寄存器，而且其中存放的数值为 1，那么操作数的有效地址实际上就是 X+1。基址寻址（based addressing）方式与变址寻址方式非常类似。他们的区别是，基址寻址方式使用的是基地址寄存器，而不是变址寄存器。从理论上来说，这两种方式的差别是如何使用这两种寻址方式，而不是如何计算操作数。一个变址寄存器保存的是一个变址（或称为索引，index），这个变址将会用作一个相对于指令地址域所给出地址的偏移量。基址寄存器保存的是一个基地址，而对应的指令地址域中的内容所表示的则是偏离该基地址的位移量。这两种寻址方式在访问数组元素和字符串中的字符时，都是非常有用的。事实上，大部分的汇编语言都提供有专用的变址寄存器，许多的字符串操作指令中都包含这类变址寄存器。如果对指令系统进行专门的设计，通用寄存器也可以在这种寻址方式中使用。

堆栈寻址

如果使用堆栈寻址（stack addressing）方式，则操作数就假定放在堆栈中。在第 5.2.4 节，读者已经了解这种寻址方式的工作原理。

其他的一些寻址方式

基于上述各种最基本的寻址方式，还可以生成许多变化。例如，有些机器有间接变址寻址（indirect indexed addressing）方式，即同时采用间接寻址和变址寻址。又如，基址/偏移量寻址（base/offset addressing）方式，是先将一个偏移量加到某个特定的基址寄存器中，然后再与指定的操作数相加，产生指令所使用的实际操作数的有效地址。还有自动增量（auto-increment）寻址方式和自动减量（auto-decrement）寻址方式，可以对所使用的地址寄存器中的内容进行自动增量或减量操作。使用这种寻址方式的优点是可以减小编码的长度。这种寻址方式在某些应用程序中（例如嵌入式系统）至关重要。而自相对寻址（self-relative addressing）方式，则是从当前指令获取偏移量，通过计算获得下一条指令的操作数的地址。当然，还有其他寻址方式。但是，熟悉掌握了立即寻址、直接寻址、寄存器寻址、间接寻址、变址寻址和堆栈寻址方式等这些基本概念，有助于理解其他的各种寻址方式。

举例说明这些不同的寻址方式。假设指令 Load 800，存储器和寄存器 R1 的安排如图 5.3 所示。

对指令中包含数值 800 的操作数或应用不同的寻址方式，并且假定寄存器 R1 隐含在变址寻址方式中，表 5-1 给出了实际装入累加器 AC 的数值。

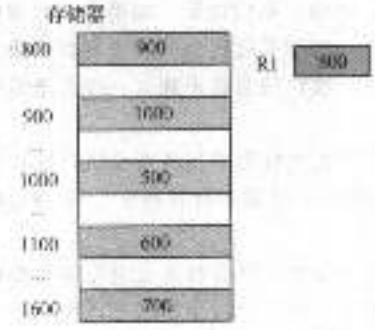


图 5-3 执行指令 Load 800 时，存储器单元中的内容

表 5-1 图 5-2 采用不同的寻址方式获取的指令执行结果

方式	装入 AC 中的值
Immediate	800
Direct	900
Indirect	1000
Indexed	700

如果使用的是寄存器寻址方式，指令 Load R1 将则数值 800 装入到累加器中。如果使用寄存器间接寻址方式，指令 Load R1 则是将数值 900 装入累加器中。

表 5-2 总结了各种不同的寻址方式。

表 5-2 基本寻址方式汇总

寻址方式	获取操作数的方法
立即寻址	操作数的数值直接包含在指令中
直接寻址	指令的地址域就是操作数的有效地址
寄存器寻址	操作数的数值存放在寄存器中
间接寻址	地址域指向实际操作数的地址
寄存器间接寻址	寄存器包含实际操作数的地址
变址寻址或基址寻址	通过将地址域的数值与寄存器中的内容相加产生操作数的有效地址
堆栈寻址	操作数位于堆栈中

利用各种不同的寻址方式，可以很大地扩大操作数的寻址范围。指令系统的设计不应该只限于一种或两种寻址方式。当然，任何的设计都需要均衡考虑。有时，我们牺牲地址计算的简便性和有限的内存引用以换取灵活性和更大的寻址范围。

5.5 指令流水线

至此，读者已经熟悉了第 4 章介绍的计算机的取指-译码-执行的周期循环过程。从概念上来说，计算机使用时钟脉冲来精确控制各个操作步骤的顺序执行。但是，有时还可以使用一些额外的脉冲来控制某个操作步骤中的一些较小细节。有些 CPU 会将取指-译码-执行周期分成一些较小的步骤，其中的某些较小的步骤可以并行执行。这种时间上的交叠可以加快 CPU 的执行速度。这种方法称为流水线（pipeline）。现在所有的 CPU 都采用该方法。

现在，假设将计算机的取指-译码-执行周期分解成如下小步骤（mini step）：

1. 取指令
2. 操作码译码
3. 计算操作数的有效地址
4. 取操作数
5. 执行指令
6. 存储结果

计算机的流水线概念与汽车组装生产线非常相似。计算机流水线中的每个步骤完成指令的一部分功能，如同汽车组装生产线一样，不同的步骤可以并行完成不同指令的各个部分。这其中的每一个步骤都称为流水线级（pipeline stage）。将流水线逐级连接起来，就构成了一条指令执行的管道。指令从流水线管道的一端进入，通过不同级的流水线作业，最后从管道的另一端出来。流水线分级的目的是要对通过流水线的每一级所花费的时间进行均衡控制。也就是要使经过每级流水线所花的时间基本上一致。如果流水线不能在时间上取得平衡，那么会发生较快的流水线级需要等待较慢的流水线级的情况。这里，举现实生活中的一个例子来说明这种不平衡带来的影响。下面考虑洗衣过程的分级管理。如果只有一个洗衣机和一个干衣机，你会发生洗衣完成后，等待干衣的情形。如果将洗衣看作是过程的第一级，而将干衣当作第二级，不难看到需要花费较长时间的干衣过程会造成衣物在两级之间发生堆积。如果再加入折叠衣物作为洗衣过程的第三级，那么人们很快就会意识到这一级总是需要等待其他的一个比较慢的级。

图 5-4 说明了一个具有重叠级的计算机流水线过程。图中清楚地标明了每条指令执行过程中的每个时钟周期和每一级流水线。这里，S1 表示取指令，S2 表示译码，S3 表示计算操作，S4 表示取操作数，S5 表示执行指令，S6 表示存储结果。

从图 5-4 可以看出，一旦第一条指令的取指任务完成，第一条指令就会被送去进行译码操作。与此同时，可以开始第二条指令的取指。当第一条指令取操作数时，第二条指令被送去译码，同时开始取第三条指令。注意，这些事件可以并行发生，就像汽车装配生产线。

假设有一个 k 级流水线，时钟周期时间为 t_p，即每级流水线需要 t_p 时间。并假定有 n 条指令（通

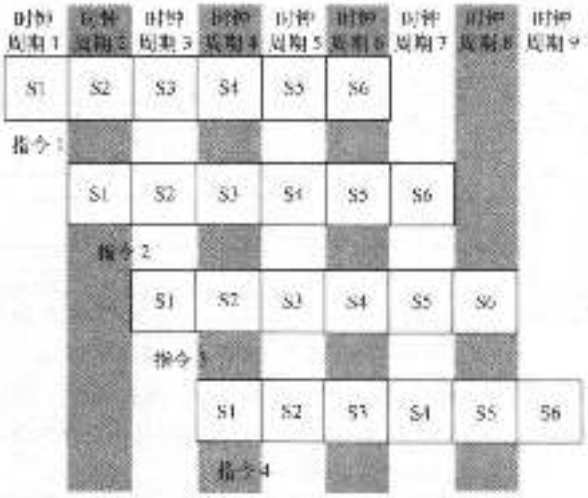


图 5-4 4 条指令经过 6 级流水线的过程

常称为 n 个任务，task）需要处理。任务 1（T_1）需要 $k \times t_p$ 时间才能完成。剩下的 $n-1$ 个任务每隔一个时钟周期就从流水线中出来一个，也就是说剩下的这些任务需要的总时间为 $(n-1)\,t_p$。因此，利用 k 级流水线完成 n 任务需要的时间为：

$$(k \times t_p) + (n-1)t_p = (k+n-1)t_p$$

或者为 $k+(n-1)$ 个时钟周期。

现在我们来计算使用流水线后所获得的速度上的提升，称为加速比（speedup）。没有流水线，执行 n 条指令共需要 nt_n 个周期，其中 $t_n = k \times t_p$。所以，获得的加速比（定义为没有使用流水线的时间除以使用流水线后的时间）为：

$$S = \frac{n t_n}{(k+n-1)t_p}$$

如果取 n 趋于无限大的极限条件，可以看出 $(k+n-1)$ 趋于 n，这样就得到了一个理论上的加速比结果：

$$S = \frac{k \times t_p}{t_p} = k$$

理论上的加速比 k 就是流水线的级数。

下面举一个例子，假设有一个 4 级流水线，其中：

- S1 = 取指令
- S2 = 译码和计算有效地址
- S3 = 取操作数
- S4 = 执行指令和存储结果

这里必须假定该体系结构还提供了一种可以并行地取数据和指令的方法。当然，只需采用两条独立的指令通路和数据通路可以做到这一点。但是，事实上大部分存储器系统都不这样做，而是把操作数存放在高速缓存中。这样，在大多数情况下，取指令和取操作数可以同时进行。这里，还假定指令 I3 是一个条件分支转移语句。这个分支转移指令会改变指令执行的顺序，即机器不再执行紧跟 I3 后面的指令 I4，而是将 CPU 控制转移去执行指令 I8。这种流水线操作的结果如图 5-5 所示。

从图中可见，机器对指令 I4、I5 和 I6 都进行了取指和接下来的各种操作。但是，在执行指令 I3（分支转移指令）后，不再需要指令 I4、I5 和 I6。只有在第 6 个时钟周期后，也就是指令 I3 执行完毕后，才对下一条可以执行的指令（I8）进行取指操作。这时，流水线的管道才会重新装满指令。从第 6

个时钟周期到第9个时钟周期，只有一条指令被执行。理想的情况下，一旦流水线管道开始装入指令后，每个时钟周期都应该有一条指令从流水线中输出。但是，上面的这个例子说明情况并非总是如此。

时钟周期 → / 指令 ↓	1	2	3	4	5	6	7	8	9	10	11	12	13
1	S1	S2	S3	S4									
2		S1	S2	S3	S4								
3（条件分支）			S1	S2	S3	S4							
4				S1	S2	S3							
5					S1	S2							
6						S1							
7							S1	S2	S3	S4			
8								S1	S2	S3	S4		
9									S1	S2	S3	S4	
10										S1	S2	S3	S4

图 5-5 带条件分支转移的指令流水线的例子

值得注意的是，并不是所有指令都必须经过管道中的每一级流水线。如果一条指令没有操作数，就没有必要经过第3级流水线。为了简化流水线的硬件设计和定时的时序问题，规定所有的指令都必须经过流水线的每一级，而不管指令是否需要这样做。

通过前面有关加速比的讨论，读者也许会觉得流水线的级数越多，计算机运行的速度就越快。从某种意义上来说，这一点是对的。将数据从存储器移动到寄存器涉及固定的开销。流水线的控制逻辑的数量和大小也会与流水线级数的增加成正比。这无疑会减慢系统总的执行速度。另外，还存在下面几种条件会导致“流水线冲突（pipeline conflict）”，这些情况会阻碍计算机实现每个时钟周期执行一条指令的目标。这些条件包括：

- 资源冲突
- 数据关联
- 条件分支语句

资源冲突（resource conflict）是指令级并行执行过程中要考虑的主要问题。例如，如果计算机要将某个数值存放到存储器中，同时又正在从存储器中提取另一条指令。这两个操作都需要访问内存。通常，解决这种冲突的办法是让存储指令继续执行，而强制让取指的指令等待。当然，某些冲突也可以通过使用两条独立的通道来加以解决：一条通道用于从内存中取数据，而另一条通道用于从内存中取指令。

数据关联（data dependency）是指当一条指令的执行尚未结束时，后面某条指令却要求该指令的执行结果作为其操作数。处理这种类型的流水线冲突有几种方法，技术上可以添加专门的硬件来检测某条指令的源操作数是否是流水线上前的指令的目标操作数。这种硬件通过在流水线中插入一个简短的延迟（通常是加入一条不执行操作的 no op 指令），让计算机有足够的时间来解决这种冲突。同样，也可以利用专门的硬件来检测这类冲突，并且引导数据经过存在于流水线各级之间的某些特殊通道。这样就可以减少指令访问其操作数所需要的时间。有些计算机体系结构采用编译器来解决这种冲突。特殊设计的编译器可以对各种指令进行重新排序，在装入任何存在冲突的数据时都会产生一个短暂的延迟，而不会对程序的逻辑和结果产生任何影响。

程序员可以使用分支转移语句来改变程序的执行流程。但是这样会对流水线造成重大问题。如果每个时钟周期都取一条指令，则可能会在前面的某条分支转移指令之前，已经提取好几条指令，甚至可能完成了译码操作。计算机处理这类条件分支语句非常困难。许多计算机体系结构都设计分支预测（branch prediction）机构，利用合理的逻辑来对下一条指令做出最优的预测。从本质上来说，分支预测结构主要是预测条件分支转移的结果。编译器试图通过重新安排机器代码的方式来产生一个延迟的分支转移（delayed branch）操作，以解决分支问题。另一种解决分支转移的问题是对程序进行重新排序，并且在程序中插入一些有用的指令。如果无法插入有用的指令，可以插入一些 no-op 指令，以保持流水线始终处于充满的状态。某些计算机采用的另外一种解决方案是对某个已知的条件分支转移的两条分支通道都开始执行取指操作，并将这些结果存储起来，等待这条分支语句的实际执行。当这条

分支语句真正执行时，需要执行的“真实”的分支路径实际上就已经知道了。

为了提高芯片硬件的工作性能，现代的 CPU 系统都采用了超标量设计（参见第 4 章），这是除流水线外的另一种新技术。超标量设计的芯片具有多个 ALU 执行单元，每个时钟周期可以流出一条以上的指令。因此，每条指令的执行时间实际上可以降到一个时钟周期以下。但是所带来的风险是使得程序的逻辑变得更加复杂。与程序的真正执行过程相比，超标量芯片需要设计更多逻辑过程来预先安排各种操作。即使采用了非常复杂的逻辑过程，也很难预先安排好动态的（on the fly）并行操作。

由于这种动态时序安排存在的各种限制，计算机设计人员开始考虑一种完全不同的体系结构，显式并行指令计算机（EPIC），读者可以参阅第 4 章所讨论的有关 Itanium 计算机的例子。EPIC 采用非常大的指令（例如 Itanium 的指令为 128 位指令），该指令可以规定几个需要并行执行的操作。由于这种并行特征是 CPU 设计中的固有特性，所以 EPIC 的指令系统强烈地取决于编译器（这意味着用户需要有一个非常复杂精巧的编译器来利用并行特征以获取机器性能上的最大优势）。安排操作的负担从处理器转移到编译器。这样，人们可以利用更多的时间来开发一种好的时序和分析可能的流水线冲突。

为了减少由于条件分支转移造成的流水线问题，IA-64 的体系结构引入了断定（predicated）指令。利用比较指令设置断定位，就像在 x86 系列 CPU 中设置条件代码一样，区别是这里有 64 个断定位。对应于每个操作都规定了一个断定位。只有当断定位等于 1 时，才执行该操作。事实上，所有的操作指令都会执行。但是，只将断定位为 1 的操作结果存储到寄存器文件中。这样做的结果是，需要执行更多指令，但是不会发生为了等待某个条件而造成的流水线阻塞冲突。

并行执行有几种级别，可以执行从简单到复杂的并行操作。几乎所有的计算机都不同程度地应用了并行的概念。指令均使用字作为操作数（通常计算机的字长为 16 位、32 位或 64 位），而不是每次都只对单一位发生作用。较高级的并行技术需要有专门设计的复杂硬件和操作系统的支持。

有关并行执行的深入研究已经超出了本书的范围，这里只是简单介绍两个并行问题的极端情形：程序级的并行执行（PLP）和指令级的并行执行（ILP）。PLP 实际上是指一个程序的各部分可以同时在多台计算机上运行。这个问题看似简单，但却要求对程序的算法进行正确的编码，才有可能发生并行执行。同时，需要确保在程序的模块之间有非常严格的同步。

ILP 是利用技术在时间上重叠执行指令。从本质上来说，我们希望单一程序中的多条指令并发地执行。ILP 分为两种类型：第一种类型是将一条指令的执行分解成若干步骤，并在时间上重叠执行这些步骤。这实际上就是流水线的功能。第二种类型的 ILP 是利用不同的指令在时间上重叠执行，即处理器本身可以在同一时刻执行多条指令。

ILP 除了有流水线体系结构外，还有超标量体系结构、超流水线体系结构和超长指令字（VLIW）体系结构。*超标量*（superscalar）体系结构是利用多条平行的流水线同时执行多个操作（参阅第 4 章）。超标量体系结构包括 IBM 公司的 Power PC、Sun 公司的 Ultrasparc 和 DEC 公司的 Alpha。*超流水线*（superpipelining）体系结构则是将超标量体系结构和流水线体系结构的概念组合起来，把流水线的各个级分割成更小的步骤。而 IA-64 体系结构却是一个*超长指令字*（VLIW）体系结构，即每条指令可以指定多个标量操作，也就是编译器可以把多个操作放到一条指令中。超标量和 VLIW 体系结构可以在一个时钟周期内进行取指和执行多条指令。

5.6 ISA 体系结构的真实案例

现在，我们再来讨论第 4 章介绍两种体系结构：Intel 体系结构和 MIPS 体系结构。目的是了解这两种体系结构的处理器的设计师是如何处理本章所遇到的一些问题：如指令格式、指令类型、操作数的数目、地址编址和流水线等。另外将介绍 Java 虚拟机以说明如何利用软件来创建一种抽象的指令系统（ISA），而可以完全不考虑计算机的真实指令系统。

5.6.1 Intel 体系结构

Intel 使用的是小端、双地址的体系结构，并且采用可变长度的指令系统。Intel 处理器的指令系统

采用的是寄存器-存储器结构，即所有的指令都可以对一个存储器单元进行操作，但另一个操作数却必须是寄存器。这种指令系统允许执行可变长度的操作，操作数据的长度可以是1字节、2字节或4字节。

从8086到80486都是单级流水线体系结构。架构师推断如果使用一条流水线很好，那么使用两条流水线会更好。Intel的奔腾结构就采用两条平行的5级流水线来执行指令，这两条流水线分别称为U线和V线，它们的不同级（步骤）分别包括：指令预取、指令译码、地址生成、指令执行和回写。为了CPU的高效率运行，这些流水线都必须保持被填满的状态，即要求并行地发出指令。编译器的任务就是确保这种并行过程的发生。而奔腾II系列就已经将两条流水线的步骤增加到12级，包括：指令预取、长度译码、指令译码、重命名/资源分配、微指令（UOP）事务安排/分派、指令执行、回写和退回。大部分新增加的流水线级都用来支持Intel的MMX技术，MMX技术是Intel公司为处理多媒体数据而对CPU体系结构功能的扩充。奔腾III的流水线增加到14级，而奔腾IV则增加到24级。这些附加的流水线级（指除本章所介绍流水线级的之外）包括一些决定指令长度的步骤，一些产生微操作的步骤和一些提交（commit）指令（即确认执行指令，并且执行的结果变成永久性）的步骤。而Itanium体系结构却只有一个10级的指令流水线。

Intel处理器支持本章所介绍的各种基本寻址方式，同时还支持多种由这些方式组合形成的寻址方式。例如，8086就有17种不同的访问存储器的方式，其中大部分都是几个基本寻址方式的变化形式。Intel公司最流行的奔腾处理器系列体系结构不但包含公司的前代产品所具有的相同寻址方式，而且同时还引入了一些新的寻址方式。通常，Intel体系结构的各种寻址方式都保持了向下的兼容性。出人意料的是，Intel的IA-64体系结构的存储器寻址方式却只有一种：寄存器间接寻址（具有可选的后增量）。看起来，这种对寻址方式的限制很不寻常，但却是遵循了RISC设计思想的指导原则。地址在通用寄存器中进行计算和存储。更加复杂的寻址方式需要特殊的硬件支持。通过限制寻址方式的数目，IA-64体系结构将对这种特殊硬件的需求减至最少。

5.6.2 MIPS体系结构

MIPS体系结构，最初代表的是无自锁流水线级的微处理器。MIPS是一种小端、按字编址、3地址和采用固定长度的指令系统。这是一种装入/存储式的体系结构，也就是说只有在装入指令和存储指令时才能访问存储器。而其他的所有指令都必须使用寄存器作为操作数，因此MIPS体系结构需要配备一个很大的寄存器组。MIPS体系结构还限制只有固定长度的操作，操作数据必须具有相同的字节数。

某些MIPS处理器，如R2000和R3000 CPU，有5级流水线。R4000和R4400处理器有8级超流水线。R10000处理器的流水线的级数取决于执行指令需要经过的功能单元：对于整数指令为5级流水线，装入/存储指令为6级流水线，而浮点运算指令为7级流水线。实际上，R5000和R10000处理器都属于超标量体系结构。

MIPS体系结构的指令系统共有5种类型的指令：执行简单算术操作的指令（例如，加法、异或、与非、移位等），数据移动指令（例如，装入、移动等），控制转移指令（如分支转移、跳转转移），多重循环指令（乘法、除法），以及其他杂类的指令（如存储PC，有条件存储寄存器）。MIPS机器的程序员可以使用立即寻址、寄存器寻址、直接寻址、间接寄存器寻址、基地址寻址和变址寻址等寻址方式。但是，MIPS的指令系统只提供一种寻址方式，即基地址寻址方式。而一些其他寻址方式是由MIPS的编译器提供的。MIPS 64处理器在嵌入式系统的优化应用时还有另外的两种寻找方式。

第4章介绍的MIPS的指令分为4个域：一个操作码域、两个操作数地址域和一个结果地址域。从本质上来说，MIPS具有三种类型的指令格式：I类型（immediate）指令、R类型（register）指令和J类型（jump）指令。

R类型指令包括一个6位操作码、一个5位的源寄存器、一个5位的目标寄存器、一个5位偏移

量和一个 6 位的功能代码。I 类型的指令包括一个 6 位操作数、一个 5 位的源寄存器、一个 5 位的目标寄存器或分支转移条件，以及一个 16 位的立即分支转移位移量或地址位移量。J 类型的指令则包括一个 6 位操作码和一个 26 位的目标地址。

5.6.3 Java 虚拟机

Java 是一种现在非常流行的计算机编程语言。有趣的是，Java 语言与工作平台无关。这就是说，如果利用 Java 语言在某种计算机体系结构（例如，奔腾）上进行代码编程，那么这些程序可以在另外一种不同的计算机体系结构（例如，Sun 的工作站）上运行，甚至无需对原程序进行修改和重新编译。

首次对 Java 程序进行编译时，Java 编译器不需要考虑运行程序的计算机体系结构内部的具体细节：例如寄存器的数目、内存的大小、或者是输入输出（I/O）端口等。但是，如果编译完成后，要执行这种 Java 程序，就需要在运行程序的实际机器体系结构上生成一个 Java 虚拟机（Java Virtual Machine，JVM）。虚拟机是真实机器的一种软件仿真。JVM 在本质上是硬件机器结构的“包装外壳（wrapper）”，并且是与工作平台有关的。奔腾机器的 JVM 不同于 Sun 公司的工作站的 JVM，它们也不同于 Macintosh 或者其他类型的机器的 JVM。然而，特定的体系结构的 JVM 一旦生成，就可以利用 JVM 来执行在任意指令系统（ISA）平台上编译的 Java 程序。程序执行过程中，由 JVM 负责装入、检查、寻找和执行字节编码。JVM 虽然是虚拟机器，但却是一个精心设计的指令系统的很好的例子。

不同体系结构的 JVM 是按照机器固有的指令集编写的。JVM 的作用相当于一个解释程序：读取 Java 字节码，并将这些字节码解释成各种清晰明了的机器指令。在对 Java 程序进行编译时，会生成特殊的字节码（bytecodes）。这些字节码就是 JVM 的输入源程序。我们可以将 JVM 比作一个巨大的开关（或 case）语句，这个开关语句每次分析一条字节码指令。每个字节码指令的执行都会造成程序跳转到某个特定的程序代码段，去完成给定的字节码指令。

这种程序的工作方式与读者熟悉的其他高级程序语言有很大的不同。例如，如果编译 C++程序，生成的结果代码一定是要针对特定的体系结构才适用。编译 C++程序将产生一个对应的汇编语言程序，该汇编语言程序然后被翻译成相对应的特定的机器代码。如果要使编译过的 C++程序能够在一个不同的工作平台上运行，则必须针对不同的目标体系结构对 C++程序进行重新编译。经过编译的程序语言会由指定的编译器翻译成可执行的二进制机器代码文件。显然，这种机器代码一旦生成，就只能在特定的目标体系结构上运行。编译好的程序语言通常具有非常好的可执行性能，而且可以与操作系统进行很好的交互访问。编译语言的例子包括：C 语言、C++语言、Ada 语言、FORTRAN 语言和 COBOL 语言等。

还有一些编程语言，比如 LISP 语言、PhP 语言、Perl 语言、Python 语言和 Tcl 语言，以及大部分的 BASIC 语言，都是解释性的编程语言。每次执行采用这些语言编写的程序时，都要重新对源程序进行解释。这种解释程序语言与工作平台无关，但是所付出的代价是程序执行的速度比较慢，通常会比编译程序慢 100 倍。第 8 章我们将对这个主题做更多的讨论。

还有一些编程语言同时以两种形式（即编译形式和解释形式）存在。这些编程语言通常称为 P 代码语言（P-code Language）。利用这种语言编写的源代码首先编译成一种中间形式的代码，称为 P 代码，然后机器会对 P 代码进行解释执行。P 代码程序语言的运行速度一般比编译程序语言慢 5 到 10 倍。Python 语言、Perl 语言和 Java 语言实际上是 P 代码语言，尽管通常人们称它们为解释语言。

图 5-6 展示 Java 语言编程环境的概要图。

除上面所介绍的 Java 语言与工作平台无关之外，更有趣的是 Java 语言的字节码事实上是一种基于堆栈的语言，其中的部分字节码由 0 地址指令组成。Java 语言的每条指令都是由一个单字节的操作码，

后面跟零个或多个操作数而构成。操作码本身就已经决定了操作码后面是否需要操作数，以及操作数（如果有的话）的形式。这类指令中的很多指令都不需要操作数。

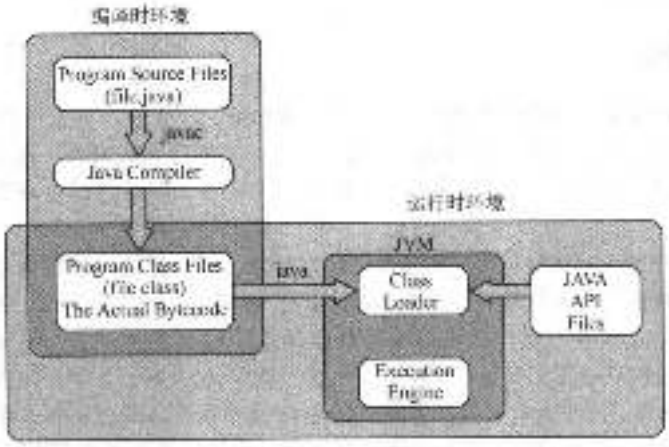


图 5-6 Java 编程环境

Java 语言采用补码形式（2 补表示法）表示带符号整数，但不允许有无符号整数。字符使用 16 位统一编码。Java 虚拟机有 4 个寄存器，支持访问 3 个不同的主存储器区域。所有对存储器的调用或访问都是基于寄存器中存放的偏移量。在 Java 虚拟机中，从来不使用指针或绝对地址。因为 JVM 是堆栈类型的机器，所以没有提供通用寄存器。当然，没有通用寄存器会限制机器的性能，因为需要产生更多的存储器调用过程。所以，我们需要在程序的可移植性和工作性能之间做出权衡。

下面考虑一个简短的 Java 程序和对应的字节码。例 5-2 是一个求两个数字中的最大数的 Java 程序。

例 5-2 一个 Java 程序，求两个数字中的最大数。

```
public class Maximum {

  public static void main (String[] Args)
  { int X,Y,Z;
    X = Integer.parseInt(Args[0]);
    Y = Integer.parseInt(Args[1]);
    Z = Max(X,Y);
    System.out.println(Z);
  }

  public static int Max (int A, int B)
  { int C;
    if (A>B)C=A;
    else C=B;
    return C;
  }
}
```

在完成对程序编译（使用 javac 编译源程序）后，可以通过发送以下命令，对程序进行拆解以检查字节码：

```
javac - C Maximum
```

于是，读者就可以看到下面的程序：

```
Compiled from Maximum.java
public class Maximum extends java.lang.Object {
    public Maximum();
```

```
    public static void main(java.lang.String[]);
    public static int Max(int, int);
}

Method Maximum()
   0 aload_0
   1 invokespecial #1 <Method java.lang.Object()>
   4 return
Method void main(java.lang.String[])
   0 aload_0
   1 iconst_0
   2 aaload
   3 invokestatic #2 <Method int parseInt(java.lang.String)>
   6 istore_1
   7 aload_0
   8 iconst_1
   9 aaload
  10 invokestatic #2 <Method int parseInt(java.lang.String)>
  13 istore_2
  14 iload_1
  15 iload_2
  16 invokestatic #3 <Method int Max(int, int)>
  19 istore_3
  20 getstatic #4 <Field java.io.PrintStream out>
  23 iload_3
  24 invokevirtual #5 <Method void println(int)>
  27 return

Method int Max(int, int)
   0 iload_0
   1 iload_1
   2 if_icmple 10
   5 iload_0
   6 istore_2
   7 goto 12
  10 iload_1
  11 istore_2
  12 iload_2
  13 ireturn
```

程序中每行的标号表示一个偏移量，或者说一条指令偏离当前方法的开始位置的字节数。注意：

```
Z = Max (X,Y);
```

会被编译成如下的字节码：

```
14 iload_1
15 iload_2
16 invokestatic #3 <Method int Max(int, int)>
19 istore_3
```

■

显然，Java 字节码是基于堆栈的编码。例如，指令 iadd 会从堆栈中弹出两个整数，将它们相加，然后将结果压回堆栈中。这里，不会出现像“add r0，r1，f2”或“add AC，X”这种类型的指令的情况。指令 iload _ 1（整数装入指令）也同样是使用堆栈，将第一列的元素压入堆栈中。主群组中的第一列元素包含整数 X，所以 X 被压入堆栈中。Y 也通过指令 15 被压入堆栈中。而指令 invokestatic 实际上执行的是 Max 方法调用。在 Max 方法完成之后，指令 istore_ 3 就将堆栈顶部的元素弹出，并且存储到数组 Z。

第 8 章将更详细地探讨有关 Java 语言和 Java 虚拟机（JMVM）的问题。

本章小结

指令系统体系结构的核心元素包括存储器模型（字长度和地址空间的分配方法）、寄存器、数据类型、指令格式、寻址方式和指令类型。虽然当今的大部分计算机系统都有通用寄存器组，并且可以通过存储器和寄存器单元的各种组合方式来指定操作数。但是，不同体系结构的计算机指令的大小、类型、格式和指令需要操作数的数目都有所不同。指令同样对操作数的存放位置也有严格的要求。操作数可以位于堆栈、寄存器组、存储器单元，或者是三者的某个组合形式中。

设计指令系统时，需要事先做出许多重要的决定。较大的指令系统允许较长的指令，使用较长的指令也就意味着需要较长的取指和译码时间。固定长度的指令译码比较方便，但可能会浪费空间。扩展操作码则是在采用大指令集和使用短指令之间的一种折衷方案。使用小端字节排序还是使用大端字节排序可能是最令人感兴趣的问题。

CPU 有三种内部存储方式：堆栈、累加器或通用寄存器。每种方式都各有优缺点，具体方案需要考虑所设计的计算机体系结构的应用范围。CPU 内部存储系统的选择方案会对设计指令系统的指令格式有直接影响，特别是会影响到指令允许引用的操作数的数目。堆栈结构的指令不需要操作数，非常适合于寄存器传输表示法（RPN）。

指令可以归纳为下面几种类型：数据移动指令、算术运算指令、逻辑运算指令、位操作指令、输入输出（I/O）指令、控制转移指令和某些专用指令。有些指令系统在各种类型的指令中都有多条指令，而另外的一些指令系统在各类指令中可能都只有为数不多的几条指令。在大多数的指令系统中，不同类型的指令数目一般比较均衡。

随着存储器技术的不断发展，使用的存储器越来越大。因此，需要有各种不同的寻址方式。计算机技术中引入了各种不同的寻址方式：立即寻址、直接寻址、间接寻址、寄存器寻址、变址方式寻址和堆栈寻址等等。各种寻址方式为程序员提供了很大的灵活性和便利性，使他们在无需改变 CPU 基本功能的情况下即可实现各种程序操作。

指令级的流水线是指令并行执行的一个典型例子。这是一种加速计算机的取指-译码-执行周期的通用技术，但是流水线技术本身比较复杂。利用流水线技术可以将指令的执行过程在时间上进行重叠处理，从而并行地执行多条指令。但是注意，流水线中可以并行执行指令的数目可能会由于指令功能的冲突而受到限制。流水线技术是同时执行多条指令的不同步骤，而超标量结构则可以同时执行多个操作。除了超长指令字（VLIW）外，本章对作为超标量技术和流水线技术组合的超流水线技术也做了简要的介绍。并行执行技术有许多类型，但是在计算机组成原理和体系结构的层次，实际上我们主要关心 ILP。

正如本章和第 4 章中所介绍的，Intel 指令系统和 MIPS 指令系统都是非常令人感兴趣的。然而，Java 虚拟机却有独一无二的指令系统，因为 JVM 的这种指令系统是利用软件来构建的。因此，Java 程序可以利用 Java 虚拟机在任何支持 JVM 的机器上运行。第 8 章将详细讨论 JVM。

深入阅读

有关指令系统、寻址方法和指令格式方面的内容在几乎每一本计算机体系结构的书籍都有详细的论述。Patterson 和 Hennessy（1997）的著作是一本有关这些内容的很好的参考书。有关 Intel x86 体系结构方面的著作很多，比如 Brey（2003），Messsmer（1993），Abel（2001）和 Jones（2001）等的书籍。如果读者对 Motorola 6800 系列感兴趣的话，可以参阅 Wray 和 Greenfield（1994）以及 Miller（1992）的书籍。

Sohi（1990）的文章很好地讨论了有关指令级流水线结构的问题。Kaeli 和 Emma（1991）则详细论述了分支转移指令对流水线结构性能的影响。要了解流水线结构的发展过程可以参阅 Rau 和 Fisher（1993）的论文。阅读 Wall（1993）的文章可以更好地了解流水线结构的局限性和可能遇到的问题。

虽然第 4 章讨论了特定的计算机结构，但是还有许多重要的指令系统体系结构值得一提。Atanasoff 的 ABC 计算机（Burks 和 Burks [1988]），Von Neumann 的 EDVAC 计算机以及 Mauchly 和 Eckert 的 UNIVAC 计算机（Stern [1981]）都带有简单的指令系统体系结构，需要使用机器语言进行编程。Intel 8080

(1 地址计算机) 属于第 4 章介绍的 80x86 系列芯片的一种早期产品。如需详细了解 Intel 早期的产品系列，可以阅读 Brey (2003) 的著作。而 Hauck (1968) 则介绍了 Burroughs 的 0 地址计算机；Struble (1968) 论述了 IBM 360 系列计算机；Brunner (1991) 详细讨论了 DEC 的 VAX 系统，这种机器采用较复杂的 2 地址指令系统；SPARC (1994) 的著作很好地介绍了 SPARC 体系结构。从 Meyer 和 Downing (1991) 的著作，Lindholm 和 Yellin 以及 Venner 的网站可以很好地了解有关 Java 虚拟机的知识。

参考文献

Abel, Peter. *IBM PC Assembly Language and Programming*, 5th ed., Upper Saddle River, NJ: Prentice Hall, 2001.

Brey, B. *Intel Microprocessors 8086/8088, 80186/80188, 80286, 80386, 80486 Pentium, and Pentium Pro Processor, Pentium II, Pentium III, and Pentium IV: Architecture, Programming, and Interfacing*, 6th ed., Englewood Cliffs, NJ: Prentice Hall, 2003.

Brunner, R.A. *VAX Architecture Reference Manual*, 2nd ed., Herndon, VA: Digital Press, 1991.

Burks, Alice, & Burks, Arthur. *The First Electronic Computer: The Atanasoff Story*. Ann Arbor, MI: University of Michigan Press, 1988.

Hauck, E. A., & Dent, B. A. "Burroughs B6500/B7500 Stack Mechanism," *Proceedings of AFIPS SJCC* (1968), Vol. 32, pp. 245–251.

Jones, William. *Assembly Language Programming for the IBM PC Family*, 3rd ed., El Granada, CA: Scott/Jones Publishing, 2001.

Kaeli, D., & Emma, P. "Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns." *Proceedings of the 18th Annual International Symposium on Computer Architecture*, May 1991.

Lindholm, Tim, & Yellin, Frank. *The Java Virtual Machine Specification*. Online at java.sun.com/docs/books/vmspec/html/VMSpecTOC.cod.html.

Messmer, H. *The Indispensable PC Hardware Book*. Reading, MA: Addison-Wesley, 1993.

Meyer, J., & Downing, T. *Java Virtual Machine*. Sebastopol, CA: O'Reilly & Associates, 1991.

Miller, M. A. *The 6800 Family, Architecture Programming and Applications*, 2nd ed., Columbus, OH: Charles E. Merrill, 1992.

Patterson, D. A., & Hennessy, J. L. *Computer Organization and Design, The Hardware/Software Interface*, 2nd ed., San Mateo, CA: Morgan Kaufmann, 1997.

Rau, B. Ramakrishna, & Fisher, Joseph A. "Instruction-Level Parallel Processing: History, Overview and Perspective." *Journal of Supercomputing 7* (1), Jan. 1993, pp. 9–50.

Sohi, G. "Instruction Issue Logic for High-Performance Interruptible, Multiple Functional Unit, Pipelined Computers." *IEEE Transactions on Computers*, March 1990.

SPARC International, Inc., *The SPARC Architecture Manual: Version 9*, Upper Saddle River, NJ: Prentice Hall, 1994.

Stallings, W. *Computer Organization and Architecture*, 5th ed., New York, NY: Macmillan Publishing Company, 2000.

Stern, Nancy. *From ENIAC to UNIVAC: An Appraisal of the Eckert-Mauchly Computers*. Herndon, VA: Digital Press, 1981.

Struble, G. W. *Assembler Language Programming: The IBM System/360 and 370*, 2nd ed., Reading, MA: Addison-Wesley, 1975.

Tanenbaum, Andrew. *Structured Computer Organization*, 4th ed., Upper Saddle River, NJ: Prentice Hall, 1999.

Venner, Bill. *Inside the Java Virtual Machine*. Online at www.artima.com.

Wall, David W. *Limits of Instruction-Level Parallelism*. DEC-WRL Research Report 93/6, Nov. 1993.

Wray, W. C., & Greenfield, J. D. *Using Microprocessors and Microcomputers, the Motorola Family*. Englewood Cliffs, NJ: Prentice Hall, 1994.

基本概念和术语复习

1. 解释有关寄存器-寄存器指令、寄存器-存储器指令和存储器-存储器指令的区别。
2. 在设计指令系统之前需要做出一些重要的决定，指出4种需要决定的事项并给出一些解释。
3. 什么是扩展操作码？
4. 如果要在一个32位字、按字节寻址的计算机中存储十六进制数值98765432，分别指出这个数值在小端机器和大端机器中的存储方式。为什么这些存储方式会与机器的大小端有关？
5. 设计指令系统时可以采用堆栈体系结构、累加器体系结构或通用寄存器体系结构。解释采用这些不同体系结构的指令系统有何差别，以及在何种情形下选择哪种结构比较好。
6. 存储器-存储器指令体系结构、寄存器-存储器指令体系结构以及装入-存储指令体系结构有何异同点。
7. 固定长度和可变长度的指令结构各有什么优缺点？当前哪一种指令结构更流行？
8. 解释一个基于零操作数的指令系统怎样从内存中获取数据值？
9. 分别采用0地址指令体系结构、1地址指令体系结构和2地址指令体系结构编写程序，哪一种程序的长度可能会更长一些（即包含更多的指令）？为什么？
10. 为什么堆栈体系结构可以使用反向波兰表示法（RPN）来表示算术表达式？
11. 列举7种类型的数据指令，并分别做出解释。
12. 什么是寻址方式？
13. 举例说明下列的寻址方式：立即寻址、直接寻址、寄存器寻址、间接寻址、寄存器间接寻址和变址寻址。
14. 变址寻址和基址寻址方式有何不同？
15. 为什么需要有多种不同的寻址方式？
16. 解释有关流水线体系结构的基本概念。
17. 假如一个时钟周期为20ns的4级流水线体系结构，需要处理100个任务，试问从理论上可以获得的加速比是多少？
18. 造成流水线体系结构速度减慢的冲突有哪些？
19. 有哪两种类型的ILP，它们之间有什么不同？
20. 分别解释什么是超标量体系结构、超流水线体系结构和VLIW体系结构。
21. 分别列举Intel指令系统和MIPS指令系统的几种异同点。
22. 解释什么是Java字节编码。
23. 分别举出一个当前流行的基于堆栈体系结构和基于GPR体系结构的例子。并说明它们有什么差别？

练习题

1. 假设现在有一个使用32位整数的按字节编址的计算机，并且要将十六进制数值1234存储在地址0处。
   ◆a）说明在大端机器中的存储方式。
   ◆b）说明在小端机器中的存储方式。
   c）如果将存储的十六进制数增大为123456，哪一种字节排序方式会更有效，大端方式还是小端方式？解释原因。
2. 说明如何将以下的数值存储到按字节编址的32位字计算机中，先采用小端排序格式，然后采用大端排序格式。假定每个数值的存储地址都从10_{16}开始。对每个数值的存放都画出内存分布图，并在各个相关（并加注标号）的内存单元位置放入正确的数值。
   a）$456789A1_{16}$
   b）$0000058A_{16}$
   c）14148888_{16}

◆3. 已知2M×16的主存储器的前两个字节中有如下的十六进制数值：

• 字节 0 处为 FE

• 字节 1 处为 01

如果这些字节保存的是一个 16 位 2 补整数，问如果按照如下的方式存储数据，那么实际存放的十进制数值分别是多少？

◆a）存储器是大端格式。

◆b）存储器是小端格式。

4. 如果将数据从大端机器转移到小端机器时，这种排序方式的改变会产生什么问题？并解释原因。

◆5. 美国人口研究学会负责监测美国全国的人口数。2000 年，该学会编写了一个程序，用来生成一个代表美国各州的人口数以及全国总人口数的文件。该程序在一台 Motorola 的处理器上运行，可以按照各种不同的规则来计算人口数，例如根据每年的平均出生率和死亡率。学会通过运行这个程序将输出结果文件报告给相关的政府机关部门，以用于各种不同的用途。但是，在宾夕法尼亚（Pennsylvania）州的政府机关，由于各部门使用的都是 Intel 的计算机，结果遇到了下面的困难。当 32 位无符号整数 $1D2F37E8_{16}$（代表 2003 年所预测的美国全国总人口数量）输入到州政府机关的计算机系统时，程序只是简单地输出这个输入数值。原因是 2003 年美国人口预测的数值太大，计算机无法处理。请帮助宾夕法尼亚州政府解释是什么原因造成了这种错误？

6. 对于计算机设计人员来说，有各种理由来要求所有的指令都应该具有相同的长度。请解释为什么对采用堆栈体系结构的计算机来说，设计相同长度的指令系统并不是一个好的方案？

◆7. 已知某台计算机使用 32 位指令和 12 位地址，如果共有 250 个 2 地址指令，请回答该计算机还可以设计多少个 1 地址指令，为什么？

8. 将下列表达式由中缀表示法变换成反向波兰（后缀）表示法：

◆a）X×Y+W×Z+V×U

b）W×X+W×（U×V+Z）

c）（W×(X+Y×(U×V)))/(U×(X+Y))

9. 将下列表达式由反向波兰（后缀）表示法变换成中缀表示法：

a）W X Y Z−−+×

b）U V W X Y Z+×+×+

c）X Y Z+V W−×Z++

10. a）采用后缀（反向波兰）表示法编写下面的等式，注意算术运算符的优先规则！

$$X=\frac{A-B+C\times(D\times E-F)}{G+H\times K}$$

b）利用一台堆栈体系结构、0 地址指令（只有 pop 和 push 指令可以访问内存）的计算机，编写一段程序对是上面算术表达式进行估值。

11. a）如果某台计算机的指令格式为：指令的长度共 11 位，而地址域的大小占 4 位。试证明采用这种指令格式的计算机是否可能有如下的指令形式，并验证你的答案：

5 个 2 地址指令

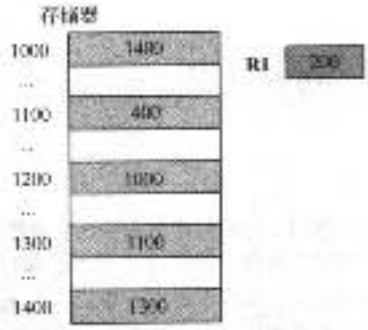
45 个 1 地址指令

32 个 0 地址指令

b）假如某台采用上述指令格式的计算机体系结构，并设计了 6 个 2 地址指令和 24 个 0 地址指令。试问可以在该指令系统中添加的 1 地址指令的最大数目是多少？

12. 直接寻址和间接寻址有何区别，并举一例说明。

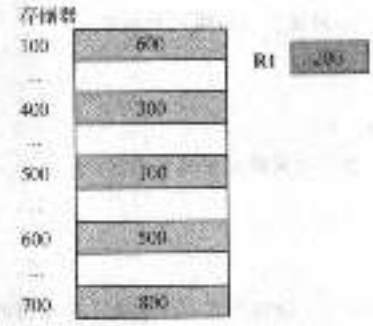
◆13. 如果有指令 Load 1000，并且已知存储器和寄存器 R1 中包含如下数值：



假定 R1 隐含在变址寻址方式中，试确定指令装入到累加器中实际数值，并填写下表：

寻址方式	装入 AC 的数值
立即寻址	
直接寻址	
间接寻址	
变址寻址	

14. 如果有指令 Load 500，已知存储器和寄存器 R1 中包含如下数值：



假定 R1 隐含在变址寻址方式中，试确定指令装入到累加器中实际数值，并填写下表：

寻址方式	装入 AC 的数值
立即寻址	
直接寻址	
间接寻址	
变址寻址	

15. 已知在某个非流水线系统中处理一项任务需要 200ns。现在将相同的任务放到一个时钟周期为 40ns 的 5 级流水线体系结构上进行处理。求出对于处理 800 项任务该流水线可以获得的加速比。比较非流水线单元，该流水线单元可以提升的最大加速比是多少？
16. 已知在某个非流水线系统中处理一项任务需要 100ns。现在将相同的任务放到一个时钟周期为 20ns 的 5 级流水线体系结构上进行处理。求出对于处理 100 项任务该流水线可以获得的加速比。比较非流水线单元，该流水线单元可以获得的理论上的加速比是多少？
17. 编写程序代码，分别在 3 地址、2 地址、1 地址和 0 地址计算机上实现表达式 A－(B＋C)×(D－E)。根据编程语言的惯例，计算表达式时不应该改变表达式中各个操作数的数值。
◆18. 某个数字计算机的存储器单元的每个字具有 24 位长度。该计算机的指令系统包含 150 个不同的操作。

所有的指令都有一个操作代码部分（操作码）和一个地址部分（只能使用1地址方式）。每条指令都存储在内存的一个字中。请回答：

◆a）操作码需要多少位？

◆b）指令的地址部分还留有多少位？

◆c）可使用的最大存储器数量是多少？

◆d）在一个存储器字中可以容纳的最大无符号二进制数是多少？

19. 已知一台计算机的存储器为256KB字，每个字字长为32位。计算机的指令格式具有4个区域：代码域；模式域，指定第1到第7种寻址方式；寄存器域，指定第1到第60个寄存器；存储器地址域。假定一条指令的长度为32位。试回答下列问题：

a）模式域必须有多大？

b）寄存器域必须有多大？

c）地址域必须有多大？

d）操作码域必须有多大？

20. 假设在某个非流水线体系结构的机器上，一条指令的执行需要4个时钟周期。其中：一个时钟周期取指，一个时钟周期译码，一个时钟周期执行算术逻辑单元（ALU）的操作，以及一个时钟周期存储结果。在一个具有4级流水线体系结构的CPU中，该指令的执行仍需4个时钟周期，那么流水线体系结构到底该如何来加速程序的执行？

*21. 任意选取一种计算机体系结构（不属于本章内容所介绍的体系结构）。研究并找出这种计算机体系结构是如何来处理本章所引入的一些基本概念，正如Intel体系结构、MIPS体系结构和Java体系结构所揭示的那样。

是非题

1. 大多数计算机通常可以归纳为以下三种CPU组织结构：（1）通用寄存器组织结构；（2）单累加器组织结构；（3）堆栈组织结构。

2. 0地址指令计算机的优点是它们可以有较短的程序；缺点是它们的指令需要许多位，这使指令变得很长。

RAM/缩写词/：永远都不够用的存储器

因为计算机配置的存储器越多，产生错误消息的速度就越快。

——无名氏

640KB的存储器应该对任何人都足够了。

——比尔·盖茨（Bill Gates）

第6章 存 储 器

6.1 概述

当今世界上大多数的计算机都是采用冯·诺伊曼模型建造的，这种计算机体系结构的核心就是存储器。计算机需要将执行各种任务的程序存放在存储器中。在第3章中，读者学习了一个小型的4×3位的存储器，并在第4章和第5章中学习了如何对存储器进行编址。从逻辑上来说，存储器的存储单元是按照线列排列的，存储器单元的地址从0开始编址，直到处理器可以寻址的最大容量。本章将讨论各种类型的存储器，以及它们如何组成计算机中不同层次的存储器系统。然后介绍高速缓存存储器，一种特殊的高速存储器。最后，讨论一种最大限度地有效使用存储器的方法，即通过分页的管理机制来实现虚拟存储器。

6.2 存储器的类型

人们常常会提出这样的问题："为什么有那么多不同类型的计算机存储器?"。我们对这个问题的回答是，计算机的快速发展使得CPU的设计不断获得改进，而要求与之配套的存储器必须在速度上与CPU的发展保持同步。然而存储器已经成为计算机技术发展中的一个瓶颈问题。虽然在过去的几年中，CPU的技术取得了重大进步，但由于采用了高速缓存存储器的技术，所以改进主存储器以便与CPU匹配的要求，实际上已经不像以往那么急迫。高速缓存存储器是一种小容量、高速度，同时也是高价格的存储器。高速缓存的作用是在频繁存取数据的过程中充当一个缓冲器。因为高性能的高速缓存存储器系统常常会掩盖掉较慢速度的存储器系统的作用，所以人们并不总是能够正确地判断使用高速缓存存储器这种技术所需要付出的额外开销。因此，在讨论高速缓存存储器之前，首先介绍各种不同的存储器技术。

尽管存储器技术类型繁多，但是大部分计算机系统都使用两种基本类型的存储器：随机存储器（random access memory，RAM）和只读存储器（read-only memory，ROM）。RAM一词在这里有些用词不当，更合适的名字是读写（read-write）存储器。RAM是计算机规格说明中给出的存储器。如果用户购买的计算机规格标明是128MB字节的存储器，即表示这台计算机具有128MB的RAM。同时，RAM也就是在本书的始终不断提及的"主存储器（main memory，或称为primary memory)"。在计算机执行程序时，RAM用来存放程序或数据。但是，RAM是一种易失性的存储器。当存储器系统掉电时，RAM中所存储的信息会丢失。现在的计算机系统一般都采用两种类型的存储器芯片来构建大规模的RAM存储器：静态随机存储器（SRAM）和动态随机存储器（DRAM）。

动态RAM由一个小电容构建而成。由于电容会泄漏电荷，所以DRAM需要每隔几个毫秒充电才能保存数据。相比之下，静态RAM技术只要电源供电不断，就可以维持它所保存的数据。SRAM由类似于第3章中介绍的D触发器的电路组成。SRAM的速度比DRAM更快，但价格更高。设计人员使用DRAM的原因是因为DRAM的存储密度更高（即单块芯片上可以存储更多的位数），消耗的功耗更低，比SRAM产生的热量要小得多。鉴于上述理由，通常使用两种技术的组合形式：DRAM用作主存储器，SRAM用作高速缓存存储器。所有DRAM的基本操作都是相同的，但是DRAM还是可以

分成许多类型，包括多层结构的 DRAM（MDRAM）、快速翻页模式（FPM）的 DRAM、扩展数据输出（EDO）的 DRAM、并发 EDO DRAM、同步动态随机存储器（SDRAM）、同步连接的（SL）DRAM、双倍数据传送率的（DDR）SDRAM，以及直接 RAM 总线的（DR）DRAM 等。而 SRAM 也可以划分成各种不同的类型：异步 SRAM、同步 SRAM 和流水线并发体系结构的 SRAM 等。要了解有关存储器类型的更多信息，可以阅读本章结尾列出的参考文献。

除了 RAM 存储器外，大部分计算机系统还使用一定数目的 ROM 存储器（只读存储器）来存放一些运行计算机系统所需要的关键信息，比如启动计算机系统所需要的程序。ROM 属于非易失性的存储器，可以长久保持它所存放的数据。这种类型的存储器也应用于嵌入式系统或是一些编程无需改变的系统中。许多家用电器、玩具和大部分的汽车都使用 ROM 芯片，在电源关闭后还可以保存其中的信息。ROM 存储器也广泛地应用于计算器和某些外围设备中，例如激光打印机采用 ROM 保存打印字符的点阵。有 5 种不同类型的基本 ROM 存储器：ROM、PROM、EPROM、EEPROM 和闪存（flash memory）。PROM 称为可编程只读存储器，它是 ROM 的一种改进类型。用户可以使用专门的设备对 PROM 进行编程。由于 ROM 采用的是硬连线，PROM 存储器中有许多熔断丝，可以通过烧断的方法来对芯片进行编程。但是，一旦编程完成，PROM 中的数据和指令都不能再更改。EPROM 为可擦除 PROM，是一种可以重复编程的可编程存储器。擦除 EPROM 需要有一个可以发射紫外线的特殊工具。要对 EPROM 进行再编程，首先需要将整块芯片中的原有信息擦除干净。EEPROM 是电可擦除 PROM，它克服了 EPROM 的许多缺点：芯片上的信息擦除只需要施加一个电场，而无需特殊的擦写工具；并且可以只擦除芯片上某些部分的信息，例如一次只擦除一个字节。闪存本质上也是一种 EEPROM。但是闪存的优点是可以按块来擦写数据，而不限于每次只能擦写一个字节。因此，闪存的擦写速度要比 EEPROM 快。

6.3 存储器的层次结构

要理解现代处理器性能，其中一个最重要的考虑因素就是存储器组成的层次结构。正如前面所见，各种存储器的性能有很大的差别。虽然某些类型的存储器存储效率比较低，但是价格却非常便宜。为了应对这种差异，能够达到最佳的性价比，当今的计算机系统都采用各种不同类型的存储器的组合配置。这种方案称为存储器分层结构（hierarchical memory）。通常存储器的速度越快，单位信息存储的成本也越高。通过采用存储器的分层组织结构，使不同层次的存储器具有不同的访问速度（或称为存取速度）和存储容量。计算机系统也可以由此获得比那些不采用各类存储器组合的系统更好的性能。通常，存储器分层结构系统的基本类型包括：寄存器、高速缓存、主存储器和辅助存储器。

现代计算机一般都具有一个数目较小的高速存储器系统，称为高速缓存。高速缓存用来暂时存放一些存储器单元中频繁使用的数据。这种高速缓存会被连接到一个容量较大的主存储器。通常主存储器的速度中等。然后再使用一个容量非常大的辅助存储器作为机器存储器系统的补充。这种辅助存储器系统一般由硬盘和一些可移动的存储介质组成。采用这种存储器分层结构的组织方式，可以提高对存储器系统的有效访问速度，而且只需使用少量快速（并且非常昂贵）的芯片。这样设计人员可以用一个合理的价格来创建具有满意性能的计算机系统。

我们通常可以按照存储器离开处理器的距离（distance）来对存储器分类，这种距离按照访问存储器所需要的机器周期数目来测量。距离处理器最近的存储器，肯定是最快的存储器。随着存储器离开主处理器距离的增加，访问这些存储器的时间也会变长。因此，较慢的存储器技术用于距离处理器较远的存储器系统，而较快的存储器技术用于靠近 CPU 的存储器系统。通常，存储器的技术越好，访问存储器的速度也就越快，但是存储器的成本也会变得越昂贵。也就是说，由于价格的因素，倾向于速度快的存储器的使用数量比速度慢的存储器要少。

在讨论存储器层次结构时，会涉及下面的一些基本术语：

- **命中**（hit）—CPU 请求的数据就驻留在要访问的存储器层中。通常，只是在存储器的较高层上才会关心所谓命中率的问题。

- **缺失**（miss）— CPU请求的数据不在要访问的存储器层。
- **命中率**（hit rate）— 访问某个特定的存储器层时，CPU找到所需数据的百分比。
- **缺失率，又称为未命中率**（miss rate）— 访问某个特定的存储器层时，CPU找不到所需的数据百分比。注意：缺失率=1—命中率。
- **命中时间**（hit time）— 在某个特定的存储器层中，CPU取得所请求的信息需要的时间。
- **缺失损失**（miss penalty）— CPU处理一次缺失事件所需要的时间，其中包括利用新的数据取代上层存储器中的某个数据块所需要的时间，再加上将所需数据传递给处理器所需要的附加时间。通常情况下，处理一次缺失事件所花费的时间要比处理一次命中所需要的时间更多。

存储器的层次结构如图6-1所示。图中的金字塔结构表示不同的存储器容量的相对大小。靠近金字塔顶部（也就是比较靠近CPU）的存储器一般规模较小，但是，与靠近金字塔较底层的存储器相比较，这些较小规模的顶层存储器的性能更好，但价格也更昂贵（针对存储器的每位的单位成本而言）。图中金字塔左边的数字表示CPU访问相应的存储器层通常所需要的时间。

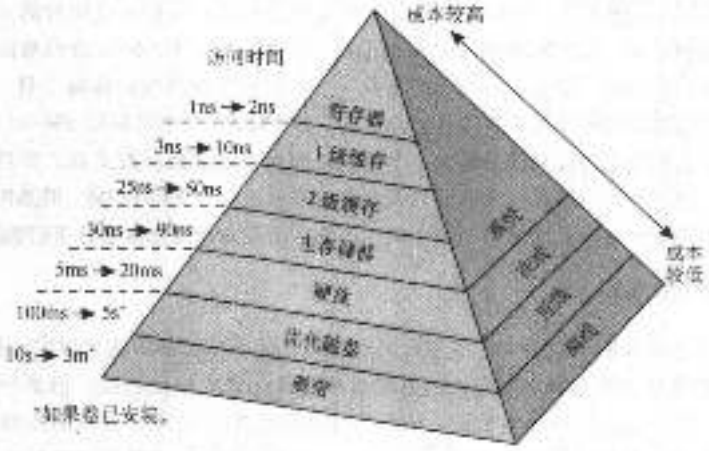


图6-1 存储器的层次结构

对于任何给定数据，处理器首先将访问数据的请求传送给存储器中速度最快、规模最小的存储器层。通常是高速缓存存储器，而不是寄存器，原因是寄存器通常会用于更专用的用途。如果在高速缓存中找到所需的数据，这些数据就会很快地被装入CPU的寄存器中。如果数据没有驻留在高速缓存中，那么访问数据的请求就会被传送到下一个较低层次的存储器系统，继续开始数据的搜索任务。如果在这一层的存储器中找到所需的数据，CPU会将数据所在整个字块的内容转移到高速缓存中。如果在这一层的存储器系统中没有找到所需数据，那么搜索数据的请求又会继续传送到下一个更低层的存储器系统，依此类推。这里我们要强调的核心思想是，较低层（显然也是速度较慢、规模较大和价格较便宜）的存储器系统会响应较高层的存储器对位于存储器单元X处的数据请求。同时，这些低层存储器也会将位于地址$X+1$，$X+2$，…，处的数据送出，也就是将整个数据块返回较高层的存储器系统。这样做的目的是希望这些返回的多余数据会在不久的将来被调用。事实上，在大多数时间，情况的确如此。存储器的分层组织结构的确非常有效，因为计算机的程序都具有"局部性"的特性。这种局部性的特点是允许处理器可以访问从地址$X+1$，$X+2$等处返回的数据。这样一来，即使发生某次缺失事件，例如高速缓存的X单元，但由于程序具有局部性的特征，CPU可能会对新近返回的整个数据块产生几次高速缓存命中。

引用的局部性

通常，处理器倾向于以一种非常规范的方式访问存储器。例如，如果没有分支转移，在MARIA

中的程序计数器（PC）会在每次提取一条指令后自动增量加 1。因此，如果处理器在 t 时刻访问了存储单元 X，那么在不久的将来访问存储单元 $X+1$ 的可能性会非常高。计算机程序对存储器的引用常常会有集中成组成簇的形式，这就是引用的局部性（locality of reference）的一个例证。通过对存储器实现分层结构的管理可以很好地利用这种局部性的优点。在处理某个缺失事件时，处理器不是简单地将所要搜索的数据传递给较高层的存储器，而是将包含该数据的整个数据块全部返回。由于引用的局部性特性，同时返回的额外数据很有可能在不久的将来被引用。如果确实需要这些数据，那么它们就会被从较快的存储器中很快地装入到 CPU 中。

引用的局部性有下面三种基本形式：

- 时间局部性（Temporal locality）：最近访问过的内容很可能在不久的将来再次被访问。
- 空间局部性（Spatial locality）：对存储器地址空间的访问形成团簇的集中倾向（例如，在数组或循环操作中）。
- 顺序局部性（Sequential locality）：访问存取器的指令倾向于按顺序执行。

这种局部性的原理使系统有机会使用少量速度非常快的存储器来有效加速对系统中主要存储器的访问。通常，在任意给定时刻计算机系统只需要访问整个存储空间中的某一个非常小的部分，而且存放在该位置的数值常常会被重复读取。因此，可以将这些数值从一个访问速度较慢的存储器复制到一个容量较小、速度很快且驻留在较高层的存储器中。利用这种方式组织的存储器系统可以将大量的信息存储在一个巨大的、低成本的存储器中。同时，却可以获得与使用速度很快，但是价格昂贵的存储器几乎相同的访问速度。

6.4 高速缓存存储器

计算机的处理器速度非常快，并且需要不断地从存储器读取信息。由于存储器的访问时间比处理器的速度慢，这样就造成了处理器需要等待存储器中信息的到达。高速缓存存储器是一种规模很小，但速度很快的存储器。处理器利用高速缓存临时存放一些即将需要的信息。

在现实生活中，有关非计算机的高速缓存的例子比比皆是。提及这些例子将有助于理解计算机高速缓存存储器的工作原理。我们设想某个房屋的屋主在车库里面有一个很大的工具箱。假如你就是房屋的屋主，并且要在地下室进行一项房屋改造工程。这项改造工程需要使用各种工具：如电钻、扳手、锤子、卷尺、几把不同规格的锯，以及各种不同类型和大小的螺丝起子。现在，首先要测量和锯一些木头。你先跑到车库，从一个巨大的工具箱中取出一把卷尺。然后，跑到地下室去测量木头。之后，跑回车库把卷尺放回工具箱，又拿起一把锯返回地下室去锯木头。随后，你决定将一些木块固定拴在一起。于是，你又到车库工具箱中找出电钻和钻头，回到地下室给木块钻孔，并将螺栓穿在孔中。然后返回车库将电钻放好，顺手拿起一把扳手，跑回地下室。结果却发现拿来的扳手尺寸规格不对，于是又得返回车库，从工具箱中拿来另外一把扳手……，你真的要如此地进行工作吗？当然不是！作为一个有头脑的人，你也许会做出某些思考："如果现在需要使用某把扳手的话，也许很快就会要用到另外一把不同尺寸的扳手，所以为什么不将整套扳手都拿到地下室呢？"。于是，你会做进一步思考并推断："如果现在需要使用某种工具的话，很有可能很快就会用到另外的一种工具，为什么不准备一个小的工具箱带到地下室去呢？"。这样一来，可以将一些急需的工具放在手边，用起来就会快得多。实际上，在这里已经应用了高速缓存的思想，存放了一些工具在手边，可以使拿取工具时更方便、快捷。而不太可能使用的工具仍然保留在某个较远的地方（例如，车库的大工具箱中），拿取它们需要花费更多的时间。这也正是计算机的高速缓存存储器所要做的事情：计算机会将已经访问过的和 CPU 可能将要访问的数据存放在一个速度较快的、比较靠近 CPU 的高速缓存存储器中。

另外一个有关高速缓存的类似例子是去商店购物。通常情况下，人们去杂货店不大可能只购买一件物品。可能会买一些马上要用的物品，或一些将来可能要用的物品。商店就像计算机系统中的主存储器，而你的家就相当于高速缓存存储器。下面再举一个相关的例子：大家想一想有多少人会带着一

整本厚厚的电话号码簿到处走来走去呢？相反，大部分人都是随身携带一个小的电话本，上面记录了一些可能经常联络的人员的姓名和电话号码。在小电话本上查找某个电话号码比在一个大电话号码簿上先查找姓名，再获取电话号码显然要快得多。人们通常会将小电话本随身携带，而把大电话簿留在家里，放在家中的茶几、书架或者是其他的地方。电话簿一般不会经常使用，可以放在某个不显眼的地方。与大电话簿相比，电话本的“存储容量”要小得多。但是当打电话时，却发现在小电话本上找到所需电话号码的几率非常高。

学生们在撰写研究工作报告时也会遇到有关高速缓存的例子。假设你正在书写一篇有关量子计算的论文。不知你是否愿意采取这样的方式来进行这项研究工作：先到图书馆，去查阅某本书籍，获取一些有用的信息资料带回家中；然后，又去图书馆查阅另外一本书，再返回家中继续写作；如此往复？答案肯定是否定的。而我们通常会采取的方法是：到图书馆去查阅所有可能需要的书籍，然后将这些书籍都带回家。图书馆就好比是主存储器，而家则相当于高速缓存存储器。

作为高速缓存的最后一个例子，读者可以想像一下本书其中的某位作者会怎样使用她的办公室。她会将不需要的资料，或者在六个月内不会使用的资料，放在一边，收藏在一组大的文件柜里面。而把那些需要经常使用的“数据”保留下来，堆放在办公桌上。这些数据就在手边，很方便随时查找。当需要从某个文件中取一些资料时，她很可能会将整个文件都拿出来，而不只是从文件夹中取出一、二篇论文。这时，取出来的整个文件都会被加入到办公桌上的文件堆中去。显然，文件柜成了作者的“主存储器”，而办公桌（桌上放了许多随意堆放的文件）就是高速缓存存储器。

高速缓存存储器的基本工作原理与前面所列举的例子基本类似。计算机系统会将那些需要频繁使用的数据复制到高速缓存存储器中，而不是通过直接访问主存储器来重新获取数据。高速缓存存储器中的内容既可以是没有组织好的，就像作者办公桌上未整理好的文件；也可以是组织有序的，就如同电话本中的内容一样。但是，无论采取何种组织方式，高速缓存中的数据都必须是可以访问的，也就是可以对数据进行定位查找。计算机中的高速缓存存储器与现实生活中的例子之间存在的一个最大的不同是：计算机没有办法，或者是通过某种推理（a priori）知道哪些数据是最有可能被计算机访问读取的。因此，计算机需要使用局部性原理，在访问主存储器时会将整个数据块从主存储器中复制到高速缓存存储器。如果被传递的数据块中有某些数据的使用率很高，那么整块数据的转移就会节省大量的存储器访问时间。高速缓存对这种新数据块的定位操作取决于两个因素：高速缓存的映射策略和高速缓存的大小。高速缓存的映射策略将在下一节讨论，而高速缓存的大小会直接影响到是否有足够的空间存放新的数据块。

对于不同的计算机，高速缓存存储器的容量有很大的差别。通常个人计算机的第二级（L2）高速缓存的大小为256KB或512KB。而第一级（L1）高速缓存要小一些，通常为8KB或16KB。L1高速缓存通常集成在微处理器中，而L2高速缓存则位于CPU和主存储器之间。因此，L1高速缓存要比L2高速缓存的速度快。有关商店的例子可以说明L1高速缓存和L2高速缓存之间的相互关系：如果将商店当作主存储器，那么可以把家里的冰箱看成是L2高速缓存，而家里的餐桌可以当作L1高速缓存。

使用高速缓存存储器的目的是为了加快存储器的访问速度。高速缓存将最近使用过的数据存放在靠近CPU的地方，而不是将这些数据存放到主存储器中。虽然高速缓存的容量远没有主存储器大，但其速度却快得多。通常主存储器由DRAM组成，其单次访问时间为60ns；而高速缓存由SRAM构成，比DRAM的访问速度快得多，访问周期也短得多。通常高速缓存存储器的单次访问时间为10ns。高速缓存存储器不需要规模很大，就可以获得非常优良的性能。设计高速缓存存储器的基本法则是：既要求高速缓存的容量尽量小，使高速缓存存储器每位的总平均成本可以接近于主存储器；又必须保证高速缓存的容量足够大，可以很好地满足系统的速度要求。由于快速存储器的价格非常昂贵，所以目前采用高速缓存存储器的技术来创建所有的主存储器是不切实际的。

到底高速缓存存储器有何“特殊”之处？高速缓存存储器并不通过地址进行访问，而是按照内容

进行存取。基于这种原因，高速缓存存储器有时也会被称为按内容寻址的存储器（content addressable memory），或简称为 *CAM*。在大部分高速缓存映射策略的方案中，都必须检查或搜索高速缓存的入口，以确定所需的数值是否存放在高速缓存中。为了简化定位搜索所需数据的过程，不同的高速缓存存储器采用了不同的高速缓存映射算法。

6.4.1 高速缓存的映射模式

要使高速缓存存储器发挥作用，则其中必须存放有用的数据。然而，如果 CPU 无法找到这些数据，那么这些数据也就变成了无用的数据。在访问数据或指令时，CPU 首先会生成一个主存储器地址。此时，如果要访问的数据已经被复制到高速缓存中，那么这个数据在高速缓存中的地址与主存储器中的地址不同。例如，位于主存储器地址 2E3 单元中的数据，可能会被存放在高速缓存存储器中非常靠前的某个单元中。现在的问题是：当数据被复制到高速缓存存储器后，CPU 如何在高速缓存中定位查找出这些数据？CPU 会采用某种特殊的映射模式，将主存储器的地址转换为一个对应的高速缓存存储器位置。

我们可以通过对主存储器地址的各个位规定某些特殊的意义来实现地址的转换。首先将地址的二进制位分成不同的组，称为域。根据映射模式的不同，主存储器地址的分组可以有两个或三个地址域。至于具体如何使用这些地址域，取决于所采用的特定的映射模式。高速缓存的映射模式决定了数据最初被复制到高速缓存存储器中时的存放位置，并且为以后 CPU 搜索高速缓存时提供了一种查找以前被复制的数据的方法。

在讨论映射模式之前，必须了解数据是如何被复制到高速缓存中的，这一点非常重要。主存储器和高速缓存的存储空间都会被划分成相同大小的字块（简称块），不同系统的存储块的大小可能有所不同。当生成一个存储器的地址时，CPU 会首先搜索高速缓存存储器，查找所要求的数据字是否已经存放在高速缓存中。如果在高速缓存中没有找到所要求的字，那么 CPU 会把主存储器中该字所在位置的整个块装入到高速缓存存储器中。正如前面所述，由于局部性原理，这种方法非常成功。如果一个数据字刚刚被引用过，那么在同一临近区域的数据字被引用的几率很大。因此，一个缺失（未命中）的数据字常常会导致查找到几个所需的字。例如，在地下室中工作时需要一些工具，某个工具可能没有找到（称为一次缺失），这样不得不去车库拿取。在车库里面，你收集了一组可能需要的工具返回地下室，希望在地下室进行房屋改造工程时可以找到几把需要的工具（即几次命中），而无需多次到车库里去拿取。因为访问高速缓存字（相当于某种工具已经在地下室中）要比访问主存储器字（类似于重新跑回到车库中）的速度快得多，所以高速缓存存储器可以加快计算机对存储器的总访问时间。

下面讨论如何使用主存储器地址的各个地址域。CPU 使用主存储器地址的其中一个地址域，对已驻留在高速缓存中的请求数据，直接给出数据在高速缓存中的位置，这种情况称为高速缓存命中（Cache hit）；而 CPU 对没有驻留在高速缓存中的请求数据，会指示出数据将要存放在高速缓存中的位置，这种情况称为高速缓存缺失（未命中，cache miss）。对于关联映射模式高速缓存，这一点略有不同，后面会简单讨论。接下来，CPU 会通过检查高速缓存块的一个有效位（valid bit），来验证所引用的高速缓存块的合法性。如果所检验的有效位为 0，即表示要引用的高速缓存块不正确，产生了一次高速缓存缺失，所以 CPU 必须访问主存储器。如果有效位为 1，表示所引用的高速缓存数据块正确，可能会产生一次高速缓存命中。但是，CPU 还需要继续完成下一个操作步骤后，才能完全确认这次的高速缓存命中事件。CPU 随后要将属于该高速缓存块的标记与主存储器地址的标记域（tag field）进行比较。标记是主存储器地址中的一组特殊的二进制位，用来标识数据块的身份。标记与对应的数据块一起存放到高速缓存中。如果标记匹配，表示找到了所需的高速缓存数据块，即产生了一次高速缓存命中。在此基础上，还需要在高速缓存块中定位找出所要求的数据字的存放位置。这项工作可以通过使用主存储器地址中一个称为字域（word field）的字地址来完成。字域代表数据字的块内地址，所有的高速缓存映射模式都要求有一个字域。最后，地址中剩余的字段由特定的映射模式来决定。下面的部分将讨论三种主要的高速缓存映射模式。

直接映射的高速缓存

直接映射的高速缓存采用模块方式来指定高速缓存和主存储器之间的映射关系。因为主存储器块通常比高速缓存块要多很多，所以很多主存储器中的块要通过竞争才能获取高速缓存中的对应位置。直接映射方式是将主存储器中的块 X 映射到高速缓存的块 Y，对应的模量为 N。其中，N 是高速缓存中所划分的存储空间块的总数。例如，假设高速缓存划分为 10 个块，那么直接映射方式是，主存储器中的第 0 块映射到高速缓存中的第 0 块，主存储器中的第 1 块映射到高速缓存中的第 1 块……主存储器中的第 9 块映射到高速缓存中的第 9 块；而主存储器中的第 10 块也会映射到高速缓存中的第 0 块。图 6-2 给出了这种映射关系。显然，主存储器的第 0 块和第 10 块（以及第 20 块，第 30 块等等）都需要竞争映射到高速缓存的第 0 块。

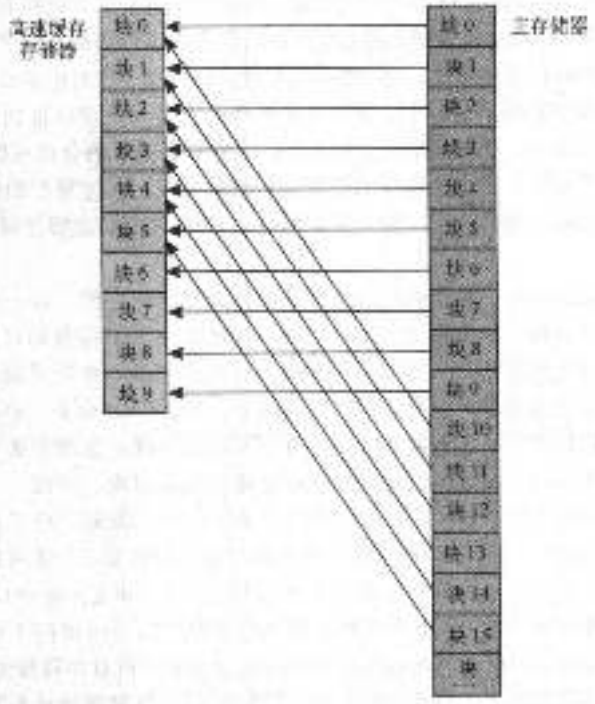


图 6-2 主存储器块到高速缓存块的直接映射关系

读者也许会奇怪，如果主存储器的第 0 块和第 10 块都映射到高速缓存的第 0 块，那么计算机又如何知道在某个特定的时刻，实际上是哪一个数据块驻留在高速缓存的第 0 块存储空间？正确的回答是，当每个数据块被复制到高速缓存时，它们的身份就已经由前面所描述的标记（tag）确定了。仔细研究可以发现，高速缓存中实际存储的信息内容要比从主存储器复制的数据信息多，如图 6-3 所示。图中有两个合法的高速缓存数据块。第 0 块包含从主存储器复制的多个数据字，并且使用了标记 "00000000" 来进行身份识别，第 1 块也同样包含多个数据字，但却使用了标记 "11110101" 来标识。而高速缓存中的其他两个数据块是非法的数据块。

数据块	标记	数据	合法性
0	00000000	字A, B, C, …	1
1	11110101	字L, M, N, …	1
2	————		0
3	————		0

图 6-3 高速缓存存储空间的分配图

为了实现直接映射功能，二进制的主存储器地址被划分为如图 6-4 所示的几个域。



图 6-4 采用直接映射方式的主存储器地址格式

每个域的大小取决于主存储器和高速缓存存储器的物理特性。字（word）域，有时又称为偏移量（offset）域，用来唯一地识别和确定来自某个指定的数据块中的一个数据字。因此，字域必须具有合适长度的数据位。同样，块（block）域也是如此，必须选择一个唯一的高速缓存块。地址中剩余的一个字段是标记（tag）域。在复制主存储器中的数据块到高速缓存时，标记会随着数据块一起被存储在高速缓存中，并且通过标记可以唯一识别和确定该数据块。当然，这三个字段的位数相加应该等于主存储器地址的总位数。

考察下面的例子：假设存储器由 2^{14} 个字组成，高速缓存共有 16 个存储空间块，每个块包含 8 个字。从这里可以算出，存储器共可以划分为 $\frac{2^{14}}{2^3}=2^{11}$ 块。显然，每个存储器的地址需要用 14 位二进制数来表示。在这个 14 位的地址中，最右边的 3 位反映的是字域，我们需要使用 3 位二进制数才能唯一确定一个数据块中 8 个数据字的一个字。这里，还需要使用 4 位才能在高速缓存中选定一个指定的数据块，高速缓存中共有 16 个数据块。因此，块域自中间的 4 位构成，而剩余的 7 位二进制数就组成标识域。不同字段的大小如图 6-5 所示。

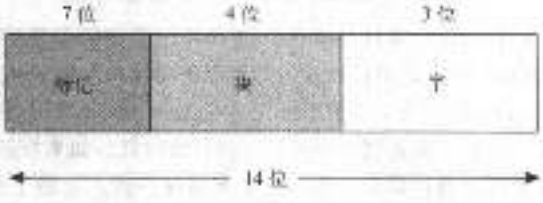


图 6-5 例子中的主存储器的地址格式

如前所述，每个块的标记应该随着该块一起存放在高速缓存中。在本例中，主存储器的第 0 块和第 16 块都要映射到高速缓存中的第 0 块存储空间，地址中的标记将允许系统能够识别存放在高速缓存第 0 块空间的是主存储器的第 0 个数据块还是第 16 个数据块。可以看到，第 0 块地址和第 16 块地址的最左边 7 位是不同的，也就是它们的标记是不同的，而且是唯一的。

为了理解这些地址的差别，下面来讨论一个规模较小而且简单的例子。假设某个计算机系统使用的是直接映射高速缓存，主存储器由 16 个字组成，划分为 8 块，其中每块包含 2 个字。再假定高速缓存的大小为 4 个空间块，共可以存放 8 个字。表 6-1 给出了这种主存储器到高速缓存的映射关系。

表 6-1 主存储器映射到高速缓存的例子

主存储器	映射到	高速缓存
块 0（地址 0，1）	→	块 0
块 1（地址 2，3）	→	块 1
块 2（地址 4，5）	→	块 2
块 3（地址 6，7）	→	块 3
块 4（地址 8，9）	→	块 0
块 5（地址 10，11）	→	块 1
块 6（地址 12，13）	→	块 2
块 7（地址 14，15）	→	块 3

这里：

- 主存储器地址为 4 位，因为主存储器中共有 2^4 或 16 个字。
- 这个 4 位的主存储器地址划分成 3 个域：字域占 1 位（只需要 1 位就可以区分一个块中的两个

字）；块域长2位（高速缓存中有4块，要求使用2位才能唯一确定每一块）；标记域只占1位（这也是所剩下的位）。

这样，主存储器地址可划分成不同的域，如图5.6所示。

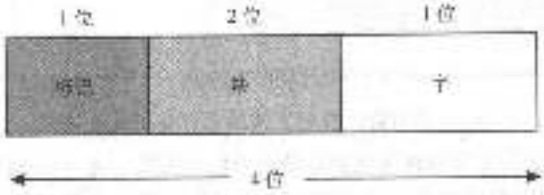


图5.6 一个16字的主存储器的地址格式

假设现在生成主存储器地址9。从上面的映射列表中可以看出，地址9代表的是主存储器中的第4个数据块。而主存储器的第4块应该映射到高速缓存的第0块。也就是说，主存储器中第4块的内容应该被复制到高速缓存的第0块。然而，计算机采用实际的主存储器地址来决定高速缓存的映射模块。这里使用二进制数来表示该地址，如图5.7所示。

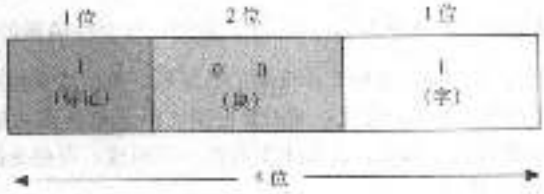


图5-7 主存储器地址9=1001₂的域的划分

当CPU生成该地址时，首先取出块域的位00，并利用这个数值引导CPU找到正确的高速缓存块。数值00表示应该对高速缓存第0块进行校验。如果高速缓存块检验正确，随后将标记域的数值1（存在于主存储器地址中）与高速缓存第0块中的标记进行比较。如果高速缓存的标记也为1，说明主存储器的第4个数据块目前正驻留在高速缓存中的第0块存储空间。如果标记为0，说明来自主存储器的第0个数据块存放在第0个高速缓存块的位置。为了明确这一点，比较下面两个主存储器地址：主存储器地址9=1001₂的数据位于高速缓存的第4块中，主存储器地址1=0001₂的数据位于高速缓存的第0块中。这两个地址只有最左边的位不同，这正是高速缓存用作标记的位。假设标记也相同，就表示来自主存储器的第4个数据块（地址分别为8和9）正驻留在高速缓存的第0块存储空间。字域的数值1用来选择该数据块中两个数据字的其中一个字。因为该位的数值为1，我们就选择具有偏移量为1的字（第2个字）。这前找到了从主存储器地址9的存储单元复制过来的数据。

下面再讨论一个有关这类问题的例子。假设CPU现在生成的存储器地址为4=0100₂，中间的两位数值（10）指示CPU搜索高速缓存块2。如果数据块正确，会将最左边的标记位（0）与存放在该高速缓存块中的标记进行比较。如果标记正确，则把该数据块的第一个字（偏移量为0）返回CPU。为了进一步理解这一过程，读者可以以主存储器地址12=1100₂为条件做一个类似的练习。

下面讨论一个规模较大的例子。假设系统采用15位主存储器地址和64个高速缓存块。如果每个块包含8个字，那么主存储器的15位地址将会被划分为一个3位的字域，一个6位的块域，以及一个6位的标记域。如果CPU生成一个如下的主存储器地址：



CPU将访问高速缓存块4。如果找到一个000010的标记，就会取出该数据块中偏移量为4的数据

字返回到CPU。

全关联高速缓存

直接映射的高速缓存方案的成本要比其他种类的高速缓存低，原因是直接映射方式并不需要进行任何的搜索操作。主存储器中的每个数据块都映射到高速缓存中指定的存储单元位置。当主存储器地址被转换成某个高速缓存地址时，CPU只需简单地检查地址的块域位，就能准确地知道需要到高速缓存的某个具体位置去查找该存储器块。这种工作方式与读者查找电话本的方式非常相似：电话本的每一页通常都有一个字母索引，如果要查找姓名"Joe Smith"，只需在s开头的列表中搜索即可。

下面讨论一个相反的极端情况，即我们并不为主存储器中的数据块唯一指定在高速缓存中的存放位置，而是允许主存储器中的数据块可以存放到高速缓存的任意位置。在这种情况下，要找到从主存储器映射的数据块的唯一方法是搜索全部高速缓存。这一点类似于要在本书作者的办公桌上查找所有的文件的情况。这样一来，整个高速缓存需要按照关联存储器（associative memory）的模式构建，以便CPU可以对这种高速缓存存储器执行平行搜索。也就是说，单一的搜索操作必须将所请求的标记与存放在高速缓存中的所有（all）标记进行比对，以确定要查找的数据块是否位于当前的高速缓存存储器中。关联存储器需要特殊的硬件来支持关联搜索，因此成本比较高。

使用关联映射方式时，需要将主存储器的地址划分成标记域和字域两部分。例如，对上面所述的存储器配置使用关联映射方式：主存储器有2^{14}字，高速缓存分为16个空间块且每个高速缓存块包含8个字。从图6-8可以看出，现在的字域仍然是3位，但是标记域变成了11位。该标记必须和数据块一起存放在高速缓存中。当要在高速缓存中搜索某个特定的主存储器数据块时，CPU会将该主存储器地址的标记域与高速缓存中存放的所有合法的标记域进行比对。如果发现有一个比对相同，就表示找到了所要求的数据块。记住，标记可以唯一地识别和确定一个主存储器中的数据块。如果没有一个标记匹配，就表示产生了一个高速缓存缺失，而且数据块必须从主存储器转移到高速缓存。

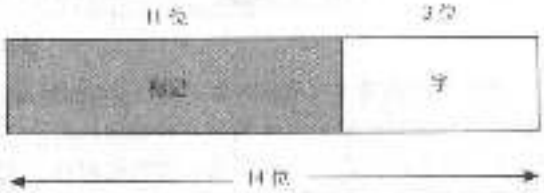


图6-8 使用关联映射的主存储器的地址格式

对于直接映射方式，如果一个已经被数据块占据的高速缓存单元需要存放新的数据块，那么就必须移除当前在高速缓存中的数据块。如果这些块的内容已经被修改过，就要把这些数据块重写到主存储器。如果数据块的内容没有发生变化，就直接使用新的数据块覆盖原来的数据块。然而，对于全关联映射方式，如果高速缓存已经装满，就需要一种置换算法来决定将从高速缓存中丢弃哪个数据块。被丢弃的块被称为牺牲块（victim block）。最简单的置换算法是先进先出（first in, first out）的置换算法，但是这种置换算法现在已经较少使用。置换算法的种类有很多，将在后面的章节中进行讨论。

组关联高速缓存

由于速度要求和结构复杂的缘故，关联高速缓存非常昂贵。虽然直接映射方式的高速缓存不太昂贵，但是使用功能却受到了很大的限制。要理解直接映射方式对高速缓存的功能限制，可以设想在上面介绍的计算机结构上运行一个程序。假设该程序在执行指令时，首先使用主存储器的数据块0，再使用数据块16；接着又使用数据块0，然后再使用数据块16，如此重复过程。由于主存储器中的数据块0和数据块16都映射到高速缓存中的同一个存储空间块，这就意味着程序会重复不断地将数据块0移出高速缓存，将数据块16移入高速缓存；又将数据块16移出，再将数据块0移入的这类操作，而不会去理会高速缓存空间中是否还有其他未使用的数据块。如果使用全关联映射高速缓存，就可以纠正这一问题。全关联高速缓存允许主存储器中的数据块存放到高速缓存存储空间的任意位置。但是，全关联映射方法都要求一个较大的标记与数据块一起存放，也就说需要一个较大规模的高速缓存存储器。同时，还需要有专门的硬件来对高速缓存中的数据块进行搜索。即高速缓存的价格会更加昂贵。

因此，非常有必要发展一种介于这两者之间的解决方案。

下面要介绍的第三种映射方式是 N 路的组关联高速缓存映射（N-way set associative Cache mapping），它是上面两种方法的某种组合形式。首先，组关联映射方法类似于直接映射高速缓存，使用地址将主存储器中的数据块映射到高速缓存中的某个指定的存储单元。但是，它与直接映射方法的重大区别在于：这种方法不是将数据块映射到高速缓存中的某一个空间块，而是映射到由几个高速缓存块组成的某个块组中。同一个高速缓存中的所有组的大小必须相同，当然，对于不同的高速缓存，组的大小可以各不相同。例如，在一个 2 路组关联高速缓存中，每组包含两个高速缓存块，如图 6-9 所示。从图中可见，组 0 中有两个高速缓存块，其中一个数据块是合法的，并且存放数据 A、B、C…；而另外一个数据块则是非法的。组 1 的情况也是如此。组 2 和组 3 也同样保存了两个数据块，在这两个组中只有第二个数据块是合法的。一个 8 路的组关联高速缓存，每个组中包含 8 个高速缓存块。由此可见，直接映射的高速缓存是一种特殊形式的 N 路的组关联高速缓存，直接映射的高速缓存中组的大小只有 1 个高速缓存块。

组号	标记	组中的第0块	合法性	标记	组中的第1块	合法性
0	00000000	字A,B,C,…	1	————		0
1	11010	字I,M,N,…	1	————		0
2	————		0	10110011	P,Q,R,…	1
3	————		0	11111100	Z,U,V,…	1

图 6-9 一个 2 路的组关联高速缓存

在组关联高速缓存的映射方式中，主存储器地址分为三部分：标记域、组域和字域。标记域和字域的作用与前面介绍的直接映射方式相同，而组域，表示主存储器中的数据块会被映射到高速缓存中的块组。假设系统有一个 2^{14} 字的主存储器，使用 2 路的组关联映射高速缓存。其中，高速缓存由 16 个块组成，每个高速缓存块中包含 8 个字。如果将高速缓存的 16 个块分成 8 个组，那么每个组包含 2 个高速缓存块。因此，组地址需要由 3 位二进制数组成。显然，字域也是 3 位，而标记域为 8 位，如图 6-10 所示。

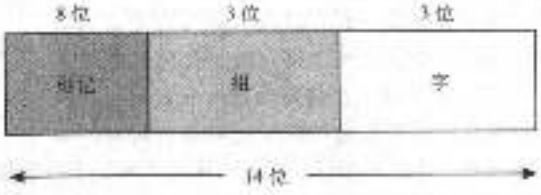


图 6-10 组关联映射的地址格式

6.4.2 置换策略

在直接映射方式中，如果多个主存储器的数据块争用某个高速缓存块，那么只有一种可能的动作：即将现有的数据块从高速缓存中踢出，为新的数据块留出存放空间。这一过程称为置换（replacement）。对于直接映射方式，不需要使用置换策略，原因是每个新的数据块所对应的映射位置都是预定的。但是，对于全关联高速缓存和组关联高速缓存，就需要用某种置换算法来确定踢一个块是要从高速缓存中被移除的“牺牲”块。当采用全关联高速缓存时，主存储器块的映射所对应的可能高速缓存块的位置有 K 个，其中 K 表示高速缓存中块的数目。当采用 N 路的组关联映射方式时，主存储器中的数据块可以映射到某个指定的高速缓存块组中的 N 块中的任意一块。现在的问题是，如何确定置换哪个高速缓存块？决定这种置换的算法称为置换策略（replacement policy）。

现在介绍几种常用的置换策略。其中有一种算法并不是实用的置换算法，但却可以用来作为置换策略的基准，用于测试其他算法是否是最佳（optimal）算法。事实上，我们希望保留高速缓存存储器中的那些在不久将要使用的数值；而要丢弃以后不再需要，或在某段时间内不需要使用的高速缓存块。最佳算法需要能够洞察未来，并且可以根据这两个标准准确地决定要保留的，或者是被排除的高速缓存块。这也正是最佳算法要完成的工作。最佳置换算法的基本思想是，将类似在未来最长的时间段内不再使用的高速缓存块。例如，假设要牺牲高速缓存块必须在块 0 和块 1 之间做出选择，如果块 0 在 5 秒后会再次被使用，块 1 在 10 秒后会再次被使用，那么就选择丢弃块 1。从实际的观点出发，人类是

不可能洞察未来的。然而，可以运行某个程序，然后再次运行这个程序，这样就可以有效地知道程序未来将要发生的事情。在第二次运行程序时，我们就可以应用最佳算法。显然，使用最佳算法可以保证最低的高速缓存缺失率。由于我们无法从每个程序的单次运行过程中看到未来将要发生的事情，因此最佳算法只能作为决定其他的某种算法优劣的一种量度方法。一种算法越接近最佳算法它就是越好的算法。

我们需要的是最接近最佳算法的算法。这里有几种不同的选择。例如，可以考虑时间局部性。推测最近没有被使用过的数值，在未来的短时间内也不太可能被再次使用。我们可以跟踪记录每个高速缓存块上次被访问的时间，为每个高速缓存块分配一个时间标签，选择最近最少被使用的高速缓存块作为牺牲块。这种算法称为*最近最少被使用*（least recently used，LRU）算法。不幸的是，LRU 算法要求系统保留每个高速缓存块的历史访问记录，这就要求高速缓存必须有相当大的存储空间，同时也会减慢高速缓存的操作速度。有多种方法近似 LRU 算法，但是这些内容超出了本书的范围（读者参阅本章结尾处的参考文献可以获取更多的信息）。

先进先出（first in，first out，FIFO）是另外一种较为流行的方法。利用这种方法，存放在高速缓存中时间最长的块将被选择作为牺牲块从高速缓存存储器中被移除，而不管这个块在最近何时被使用过。

另外一种选择牺牲块的方法是*随机*（random）选择。LRU 和 FIFO 遇到的问题是存在*简并引用*（degenerate reference）的情形，这时会产生对某个高速缓存块的*重复操作*（thrash）。即不停地重复将一个数据块移出高速缓存，并移回高速缓存；然后又移出高速缓存，再移回高速缓存；如此反复。对于随机选择方法，人们还存在一些争议。虽然它不会出现对某个块的重复操作，但有时却有可能把一些在即将需要的数据从高速缓存中丢掉。遗憾的是，很难有真正意义上的随机置换，并且随机置换方法会降低高速缓存的平均性能。

算法的选择通常要取决于计算机系统的使用方式。对于所有情况来说，不存在最好的单一的、实用的算法。基于这种理由，设计人员使用适合于多种变化环境的算法。

6.4.3 有效存取时间和命中几率

分层存储器系统的性能可以采用其*有效存取（访问）时间*（effective access time，EAT），或者称为每次访问所需要的平均时间来量度。EAT 是使用命中率与相连存储器层次的相对访问时间产生的加权平均。例如，假设高速缓存的访问时间是 10ns，主存储器的访问时间是 200ns，高速缓存存储器的命中率是 99%。那么，对于这样一个两级的存储器结构，处理器访问一个数据项的平均时间为：

$$\text{EAT} = \underbrace{0.99\,(10\text{ ns})}_{\text{高速缓存命中}} + \underbrace{0.01(200\text{ ns})}_{\text{高速缓存缺失}} = 9.9\text{ ns} + 2\text{ ns} = 11\text{ ns}$$

严格说来，这个平均时间的真正含义是什么？如果从一个较长时间周期的角度来看访问时间，这个系统的行为类似系统具有一个存取时间为 11ns 的单一的大的存储器。一个命中率为 99%的高速缓存可以使存储器系统具有很好的工作性能，虽然这种存储器的大部分都是采用存取时间为 200ns 的慢速技术构建而成。

计算两级的存储器有效存取时间的公式为：

$$\text{EAT} = H \times \text{Access}_C + (1 - H) \times \text{Access}_{MM}$$

其中，H 为高速缓存命中率，Access_C 是高速缓存的访问时间，并且 Access_{MM} 是主存储器的访问时间。

这个公式可以推广应用到 3 级、甚至 4 级的存储器系统，后面很快就会对此进行讨论。

6.4.4 何时高速缓存的方法会失效

当程序具有局部性时，高速缓存的操作方式非常有效。但是，如果程序的局部性不好，高速缓存就会失效，并且导致存储器的层次结构的性能很差。特别是面向对象的编程可能会导致程序的局部性不是最佳。另外一个局部性较差的例子出现在对二维数组的访问中。数组元素通常是按照*行优先*

(row-major) 顺序存储的。为举例起见，假设数组的一个行刚好可以存放在一个高速缓存块中，并且高速缓存可以存放数组的全部元素，而不只是数组的某个行。现在假设有一个程序要访问该数组，每次访问数组的一行，对数组的第一行访问会产生一次高速缓存缺失。但是，一旦数组块被转移到高速缓存后，那么接下来对该行的所有访问就都会产生命中。因此，在对一个 5×4 数组超过 20 次的访问中（假设程序正在访问该数组的每个元素），将会产生 5 次缺失和 15 次命中。如果程序要访问的是一个按照列优先 (column-major) 顺序存储数组，那么对该数组列的第一次访问会产生一次高速缓存缺失，处理器随后会将整行的数组元素转移到高速缓存中。但是，对该列的第二次访问仍然会导致另一次高速缓存缺失。因为这时正在对数组的访问是按照数组的列进行，所以已经移入高速缓存的数组行的数据此时并不会被使用。由于高速缓存的容量并不够大，所以 20 次此类的访问就会产生 20 次的高速缓存缺失。要介绍的第三个例子是一个程序对一个线性数组进行循环操作，而该数组不能完全存放到高速缓存中。如果按照这种方式使用存储器，那么程序的局部性就会大大降低。

6.4.5 高速缓存的写策略

除了要选择进行置换的牺牲块外，设计人员还必须决定对高速缓存的脏块 (dirty block) 的处理方案。高速缓存中的脏块是指已经被修改过的数据块。当处理器写入主存储器时，数据可能也会被写入到高速缓存中，原因是假设了处理器可能很快就会再次读这些数据。如果修改了某个高速缓存块的数据，高速缓存的写策略 (write policy) 会决定何时更新对应的主存储器数据块来保证与高速缓存块的数据一致性。高速缓存有两种基本的写策略：

- 写通 (write-through) ——写通策略是指在每次写操作时，处理器会同时更新高速缓存和主存储器中对应的数据块。写通策略的速度要比回写策略的速度慢，但是可以保证高速缓存中的内容始终与主系统存储器相一致。写通策略的明显缺点是，每次写操作都要访问主存储器。使用写通策略意味着每一次对高速缓存的写操作都必需伴随一次对主存储器的写操作，这样减慢了系统的速度。如果所有的访问都是写操作，那么存储器系统的速度基本上会减慢到主存储器的访问速度。但是，在实际应用中，大多数存储器的访问都是读操作。因此，可以忽略写通策略所带来的系统速度上的减慢。
- 回写 (write-back) ——也称为 copyback，是指当只有某个高速缓存块被选择作为牺牲块而必须从高速缓存中移除时，处理器才更新主存储器中对应的数据块。通常，回写策略的速度会比写通策略的速度快，原因是处理器不会因为每次写高速缓存时都要浪费一些时间写信息到存储器。这样一来，存储器访问的次数也就减少了。回写策略的缺点是，主存储器和高速缓存的对应单元在某些时刻可能会存放着不同的数值。而且，如果某个进程在回写主存储器完成之前发生中断（或崩溃），那么高速缓存中的数据可能会丢失。

为了改进高速缓存的性能，必须提高高速缓存的命中率。要提高高速缓存的命中率，可以使用更好的映射算法（粗略估计，算法改进可以增加 20%的命中率），更好的写操作技巧（具有增加 15%命中率的潜在能力），更好的置换算法（最多可有 10%的命中率增加）和更好的程序编码方案。正如前面介绍的有关数组的例子，按行访问的编码方案要比按列访问的编码方案最多增加 30%的命中率。另外，简单地增加高速缓存的容量大小可以改善大约 1～4%的命中率，但是并不能保证总是如此。

6.5 虚拟存储器

高速缓存允许计算机从一个较小规模但速度较快的高速缓存存储器中，频繁地访问使用过的数据。高速缓存位于靠近存储器层次结构的顶部。这种分层组织结构所固有的另外一个重要概念是虚拟存储器 (virtual memory)。虚拟存储器使用硬盘作为 RAM 存储器的扩充，增加了进程可以使用的有效地址空间。大多数个人计算机的主存储器的容量相对较小，通常小于 512MB。它通常没有足够的存储空间来并发地保持多种应用程序。比如同时运行一个字处理应用程序，一个电子邮件(e-mail)

程序和一个绘图程序，另外还要运行自身的操作系统。使用虚拟存储器，计算机可以寻址比实际主存储器更多的主存储器空间。计算机使用硬盘驱动器保持额外的主存储器空间。硬盘上的这部分区域被称为页文件（page file），因为这些页文件在硬盘上保持主存储器的信息块。理解虚拟存储器最简单的方法是将虚拟存储器想像为一个从概念上虚构的大存储器单元，其中所有的寻址问题都由操作系统来处理。

实现虚拟存储器最常用的方法是使用主存储器的分页机制（paging），这种方法是将主存储器划分成固定大小的块，并且程序也被划分成相同大小的块。通常，程序块会根据需要被存放到存储器中。没有必要将程序的连续块也存储到主存储器的连续块中。因为程序的各个片段可以无序存储，所以程序的地址一旦由 CPU 生成，就必须转换成主存储器的地址。如前所述，在高速缓存方式，主存储器的地址应该转换为高速缓存的位置。在使用虚拟存储器时，这点是相同的：每个虚拟地址都必须转换为物理地址。但是，怎样才能做到这一点呢？在对虚拟存储器进行深入讨论之前，我们先定义一些经常使用的通过分页实现虚拟存储器的术语：

- 虚拟地址（virtual address）——进程所使用的逻辑地址或程序地址。只要 CPU 生成一个地址，就总对应指虚拟地址空间。
- 物理地址（physical address）——物理存储器的实际地址。
- 映射（mapping）——一种地址变换机制，通过映射可以将虚拟地址转换成物理地址。这类似于高速缓存映射。
- 页帧（page frame）——由主存储器（物理存储器）分成的相等大小的信息块或数据块。
- 页（pages）——由虚拟存储器（逻辑地址空间）划分成的信息块或数据块。每页的大小与一个页帧相同。在硬盘上存储虚拟页及以供进程使用。
- 分页（paging）——将一个虚拟页从硬盘复制到主存储器的某个页帧的过程。
- 存储碎片（fragmentation）——变得不能用的存储器单元。
- 缺页（page fault）——当一个请求页在主存储器中没有找到时所发生的事件，必须将请求页从硬盘复制到存储器。

因为主存储器和虚拟存储器都被划分成相同大小的页，所以我们可以把程序进程的地址空间的一些片段移动到主存储器中，但并不需要将这些片断的内容按照连续的模式存储。如前所述，我们不必立即将所有进程全部装入主存储器中；虚拟存储器允许存储器中只需存在特定片断即可运行程序。暂时不用的程序部分会存储在硬盘上的页文件中。

可以使用不同的技术来实现虚拟存储器，包括分页、分段或分页与分段的组合方式。分页是最常用的技术。在学习操作系统时，将详细讨论这一主题。分页技术的成功与否，就如同高速缓存技术一样，在很大程度上依赖于局部性原理。如果在主存储器中没有所需要的数据，则会将该数据所在的整个块都从硬盘拷贝到主存储器，并且期望同一页的其他数据将在程序的继续执行过程中有用。

6.5.1 分页

分页的基本思想非常简单：按照固定大小的信息块（页帧）为各个进程分配物理存储空间，并且通过将信息写入页表（page table）的方式跟踪记录过程的不同页的存放位置。每个进程都有自己的页表，页表通常驻留在主存储器中，页表存储该进程的每个虚拟页的物理位置。页表一般有 N 行，N 代表该进程的虚拟页的页码数。如果当前进程的某些页不在主存储器中，则页表会通过设置一个有效位（valid bit）为 0 来指示；如果进程的某一页当前已经在主存储器中，则表示该页的有效位设置为 1。因此，每个页表的入口目录都由下面的两部分内容组成：有效位和帧数。

通常页表中还会附加一些其他内容，以便传递更多的信息。例如，增加一个修正位（dirty bit，或称 modify bit）来指示页中的内容是否已经发生了改变。这样做可以使处理器在进行返回页内容到硬盘的操作时会更有效率。如果页没有被修改，则不需要在磁盘上重写该页。还可以增加另外一个使用位

(usage bit) 来指示页的使用情况。只要处理器访问过某一页，设置该位为 1。一定时间周期后，又会设置该使用位为 0。如果该页被再次引用，则使用位又被设置为 1。但是，如果该使用位保持为 0，说明在某段时间间隔内未使用过该页。因此，系统就可能把该页从主存储器移出，放到磁盘上。这种做法当然可以使系统受益，因为系统释放了该页所占用的存储空间给进程最终需要使用的另外一页（在介绍置换算法时再做详细讨论这一内容）。

虚拟存储器中页的大小与物理存储器的页帧相同。程序进程的存储空间分为固定大小的页，当把程序的最后一页复制到存储器时，可能会产生潜在的内部碎片 (internal fragmentation) 现象。进程实际上可能并不需要占用整个页帧，但是又没有其他的进程使用该页帧的剩余部分。所以，程序进程所占用的这个最后页帧中使用的存储空间显然被浪费掉了。如果进程本身的全部内容需要占用的空间小于一个整页，而在复制到存储器时就必须占用一个完整的页帧，这时存储碎片的情况就可能会发生。对于某个特定的存储器分区（这里指分页），内部碎片会导致一些未使用的存储空间。

学习了分页的基本概念，下面讨论分页的工作原理。当某个进程生成一个虚拟地址时，操作系统必须动态地将虚拟地址转换成数据实际驻留的存储器的物理地址。为了简化问题起见，这里假设没有高速缓存存储器。例如，从程序的观点来说，可以看到一个 10 字节程序的最后一个字节将会存放在地址 9。假定程序是由 1 字节指令和 1 字节地址组成，存储器从地址 0 开始。但是，当将程序实际装入到存储器时，逻辑地址 9（在汇编语言程序中可能是对标号 X 的引用）可能实际驻留在物理存储器的位置是 1239，这也就意味着程序是从物理地址 1230 处开始装入。所以，需要有一种简单的方法将这个逻辑，或称为虚拟地址 0 转换成物理地址 1230。

为了实现这种地址之间的转换，可以把虚拟地址分成两个部分：页域 (page field) 和偏移量域 (offset field)。偏移量表示要使用的数据在页内的位置。这种地址转换的过程类似于高速缓存映射算法中将主存储器地址划分成不同域的过程。与高速缓存块的划分相似，页面的大小总是满足 2 的乘方关系。这种做法显然可以简化从虚拟地址中提取页码和偏移量的操作。

要访问给定虚拟地址的数据，系统需要执行以下的步骤：

1. 从虚拟地址中提取页码。

2. 从虚拟地址中提取偏移量。

3. 通过访问页表将页码转换为对应的物理页帧数。

A. 在页表中查找页码（使用虚拟地址的页码作为索引）。

B. 检验页的有效位。

1) 如果有效位=0，则表示系统产生了一个缺页事件。这时，操作系统必须介入完成以下任务：

a. 在磁盘上查找到所使用的页。

b. 寻找一个空的页帧（如果存储器满的话，则必须从主存储器中移除一个“牺牲”页，并且将其内容复制到硬盘）。

c. 把要使用的页复制到主存储器的空页帧。

d. 更新页表（新装入的虚拟页在已经修改过的页表中必须有自己的页帧数和有效位。如果有某个“牺牲”页，则必须将有效位设置为 0）。

e. 重新执行引起缺页的程序进程，继续到步骤 B2。

2) 如果有效位=1，则表示查找的页已经在主存储器中。

a. 使用实际帧数替代虚拟页码。

b. 对于给定的虚拟页，访问对应的物理页帧中的位于给定偏移量位置的数据。具体物理地址是，页帧加上偏移量。

值得注意的是，如果发生缺页，进程在主存储器中有空的页帧，那么新近取回的页可以放置在空页帧的任意页帧内。但是，如果分配给进程的主存储器已经满，则必须选择一个牺牲页。选择牺牲页的工作需要使用置换算法，这种置换算法与高速缓存中使用的置换算法非常类似。FIFO、随机选择和

LRU 都是将近期转页的各种可能的置换算法。要了解有关置换算法的更多信息，可以参阅本章结尾处的参考文献。

下面举例说明分页的工作过程。假设对于某个特定的进程，有一个 2^8 字的虚拟地址空间和包含 4 个页帧的物理存储器，但是没有高速缓存。这意味着程序生成的地址空间范围是 0 到 255_{10}，即 00 到 FF_{16}。再假定物理页帧的长度为 32 个字，虚拟地址需要有 8 位，而物理地址为 7 位，即 4 帧各 32 个字，总共是 128 个字，或 2^7 个字。另外，还假设进程中其他页已经被装入主存储器中。图 6-11 为系统当前状态的示意图。

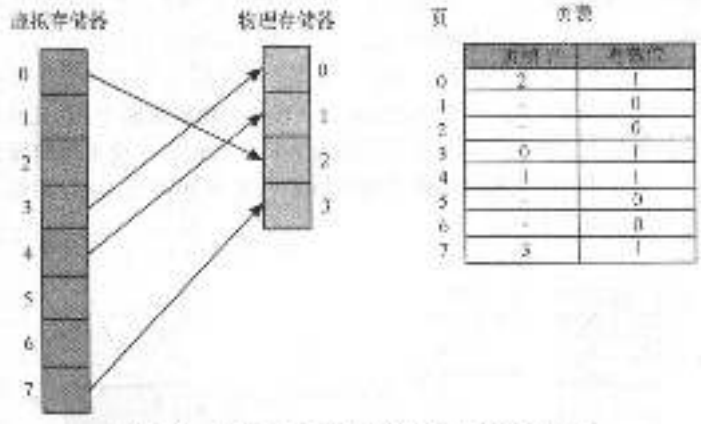


图 6-11 使用分页机制的当前状态和关联页表

每个虚拟地址具有 8 位，并且分为两个区域：页域为 3 位，表示总共有 2^3 个虚拟存储器页 $\left(\frac{2^8}{2^5}\right)=2^3$。每一页的长度为 $2^5=32$ 个字，所以页偏移量需要 5 位。因此，8 位虚拟地址格式如图 6-12 所示。

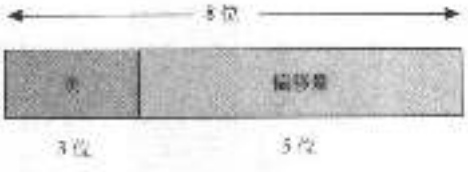


图 6-12 一个 8 位虚拟地址的格式，页的大小为 $2^5=32$ 个字

假设系统现在生成了一个虚拟地址 $13_{10}=0D_{16}=00001101_2$。首先将这个二进制地址划分为页域和偏移量域（参见图 6-13）。页域 $P=000_2$，而偏移量域等于 01101_2。为了继续代码转换过程，使用该页域的数值 000 作为一个索引进入页表查询。找到页表的第 0 个目录，发现虚拟页的第 0 页映射到物理页帧的第 $2=10_2$ 帧。这样，被转换的物理地址就变成了页帧 2，加上偏移量 13。注意，物理地址只有 7 位。其中，页帧占 2 位，共 4 帧，偏移量占 5 位。若采用二进制数，物理地址就是 1001101_2，或写成地址 $4D_{16}=77_{10}$，如图 6-14 所示。同样，也可以采用其他方式查找到该地址。每页有 32 个字，而要找的虚拟地址在虚拟页的第 0 页，被映射到物理页帧的第 2 帧。因为页帧 2 是从地址 64 开始的，而偏移量为 13，所以最后的物理地址就是 77。

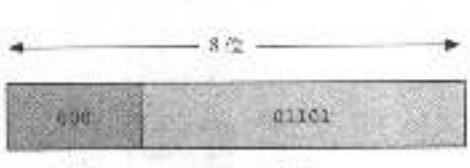


图 6-13 虚拟地址 $00001101_2=13_{10}$ 的格式



图 6-14 物理地址 $1001101_2=77_{10}$ 的格式

下面，我们研究一个真实的小系统的完整例子（同样，不涉及高速缓存）。假设，长度为 16 字节的程序，需要访问一个按字节方式编址的 8 字节的存储器。

这就是说，存储器的每个字节或字，都有自己的地址。每一页的长度是2个字（字节）。程序执行时，生成如下的地址引用的字符串（按十进制数值给出）：0，1，2，3，6，7，10，11。这个地址引用的字符串表示，程序进程将首先引用地址0，然后引用地址1，接下来是地址2，等等。最初，存储器中并没有保存该程序的页。当请求使用地址0时，地址0和地址1（它们都位于第0页中）的内容都被复制到主存储器页帧的第2帧内。这里假设，另一个进程正在占用存储器的第0帧和第1帧，因此这两个帧不可用。这是一个有关缺页的例子，因为要查找的程序页不在主存储器，需要从磁盘中提取。然而，当引用地址1时，数据已经存在于主存储器中，所以产生了一次页命中。接着，在引用地址2时，又会产生另一次缺页事件。该程序的第1页会被复制到存储器的第0帧。程序进程如此继续进行。当上面的这些地址都被引用，并且所有相关的程序页也都已经从磁盘复制到主存储器后，该系统的状态就变成了如图6-15a所示的状态。从图中可见，地址0包含了数据值A的程序页，目前正驻留在地址为$4=100_2$的存储器单元中。因此，CPU必须将虚拟地址0转换为物理地址4，并且使用前面介绍的转换方法来完成这种地址转换任务。注意，因为主存储器地址只有3位，对应于存储器只有8个字节，而虚拟地址必须有4位（对应于程序进程的长度在虚拟地址中包含16个字节），因此，这种地址的转换也必须是从4位地址转换成3位地址。

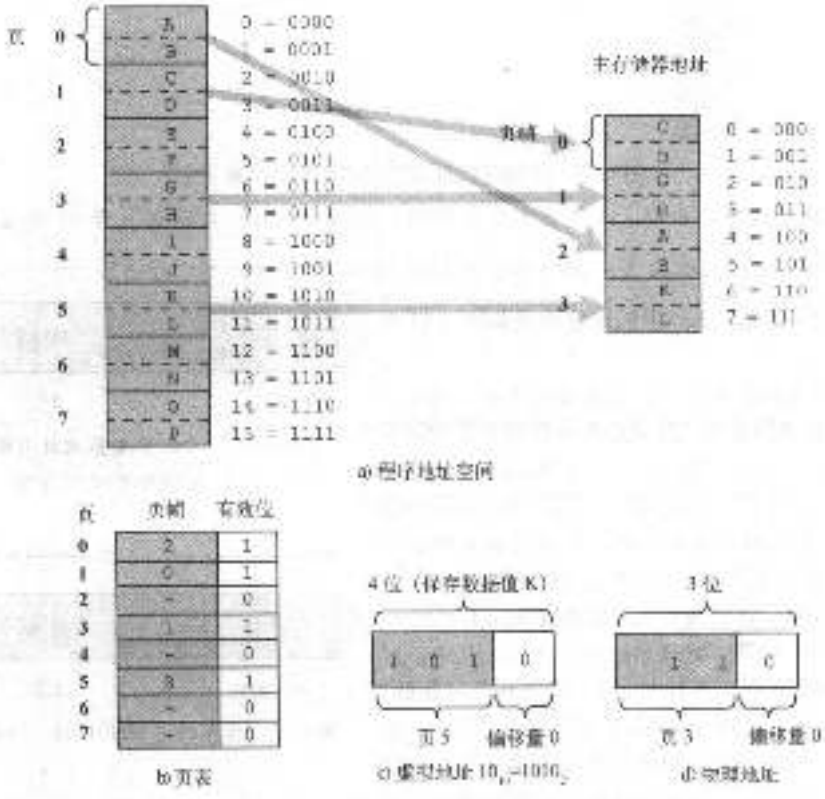


图6-15 一个小规模存储器的例子

图6-15b描述的是在程序访问了请求的虚拟页后的进程页表。不难看出，页0、页1、页3和页5进程都是合法的，并且都驻留在存储器中。但是，页2、页6和页7却是非法的，引用它们将会产生缺页的错误。

下面，我们来仔细分析一下这种地址的转换过程。假设CPU现在第二次生成程序地址，或者说是虚拟地址$10=1010_2$。从图6-15a中可见，存放数据“K”的程序地址单元驻留在主存储器地址为

$6=0110_2$ 的存储单元中。我们知道，计算机必须执行某个特定的转换过程来寻找数据。因此，可将虚拟地址 1010_2 划分为页域和偏移量域。因为程序的长度有 8 页，所以页域需要占用 3 位，余下 1 位地址就留给了偏移量。因为每页只有 2 个字，所以这种划分是正确的。这种域的划分如图 6-15c 所示。

一旦计算机知道了这些地址域的内容，将虚拟地址转化成物理地址的工作实际上是一件非常容易的事情。现在，我们使用页域中的数值 101_2 作为一个索引进入页表。因为 $101_2=5$，所以就用 5 作为偏移量查找页表。从对应的页表可以发现，虚拟页 5 映射到物理页的第 3 帧，参见图 6-15b。现在，采用物理页的帧数 $3=11_2$ 来取代虚拟页的页码 $5=101_2$，但需要保持相同的偏移量。新的物理地址为 110_2，如图 6-15d 所示。很显然，这一程序进程已经成功地实现了从虚拟地址到物理地址的转换，并且正如所要求的那样，地址的位数也从 4 位减少到 3 位。

上面介绍的是一个规模较小的存储器系统，下面将讨论一个更大规模，更接近实际的存储器系统的例子。假设虚拟地址空间的长度为 8KB 字，物理存储器的大小为 4KB 字。它是按字节方式进行寻址。物理存储器页的大小为 1KB 字。这里的系统也没有涉及高速缓存存储器，但是对本例的讨论会有助于更好地理解虚拟存储器的工作原理。最后，我们将会举例讨论一个既使用分页，又具有高速缓存功能的存储器。因为虚拟存储器的大小为 $8KB=2^{13}$，所以虚拟地址的位数有 13 位。其中，3 位地址用作页域，对应于虚拟页的总页码 $\frac{2^{13}}{2^{10}}=2^3$ 页；并且一个 10 位的偏移量，对应于每页有 $1KB=2^{10}$ 字节。显然，物理存储器地址只有 12 位，即 $4KB=2^{12}$ 字。其中，2 位地址用作为页域，主存储器仅有 2^2 个页帧；余下的 10 位地址就是页内偏移量。虚拟地址和物理地址的格式如图 6-16a 所示。

作为举例，假定页表内容如图 6-16b 所示。图 6-16c 为表示不同存储器地址（以 10 为基）的列表，该表对于说明存储器地址转换时非常有用。

假设 CPU 生成了一个虚拟地址 $5459_{10}=1010101010011_2$。图 6-16d 说明了如何将这个虚拟地址划分成页域和偏移量域，以及如何将这个虚拟地址转换为物理地址 $1363_{10}=010101010011_2$。从本质上来说，物理地址的帧页码 01 将代替虚拟地址中页域的数值 101，这反映了虚拟地址的页 5 将映射到物理地址的第 1 帧，参见图 6-16b 的页表。图 6-16e 说明了如何将虚拟地址 2050_{10} 转换为物理地址 2。图 6-16f 表示的是虚拟地址 4100_{10} 产生了一个缺页的错误；页 $4=100_{10}$ 在页表中属于非法页。

需要指出的是，要选择一个合适的页面大小是十分困难的。页的长度越大，所需要的页表就会越小，这样可以节省主存储器的存储空间。但是，如果每页的容量太大，可能造成页内部的碎片现象就会变得更加严重。较大的页容量同样也意味着从磁盘到主存储器的实际转移操作的次数会比较少，因为每次转移的信息块变大。但是，如果转移的信息块太大，程序的局部性法则会开始遭到破坏。而且，由于大信息块可能转移太多不需要的数据，而导致资源的浪费。

6.5.2 使用分页的有效存取时间

在分析高速缓存时，我们引入了有效存取时间的表示方法。在使用虚拟存储器时，我们同样也需要介绍地址有效存取时间（EAT）的概念。这里也存在着一个与虚拟存储器相关的时间损失（开销）：处理器每次访问存储器，都必须执行两次对物理存储器的访问操作。一次是引用页表，而另一次是访问要请求的实际数据。读者很容易理解这两次操作是如何影响有效存取时间的。假设访问一次主存储器需要的时间为 200ns，可能产生的缺页率为 1%。也就是说，其中 99%的时间处理器都可以在存储器中找到所需要的页。再假设每次引用不在存储器中的页时，需耗费的时间为 10ms。这个 10ms 时间包括了将缺页转移到存储器，更新页表，以及引用数据所需要的总时间。因此，该存储器访问的有效存取时间为：

$$EAT=0.99\ (200\ ns+200\ ns)\ +0.01\ (10\ ms)\ =\ 100\ 396\ ns$$

即使 100%的请求页都在存储器中，但有效存取时间仍然有：

$$EAT=1.00\ (200\ ns+200\ ns)\ =\ 400\ ns$$

即有效存取时间实际上为存储器访问时间的二倍。因为页表本身存放在主存储器中，所以访问页表也会耗费掉一个额外的存储器存取时间。

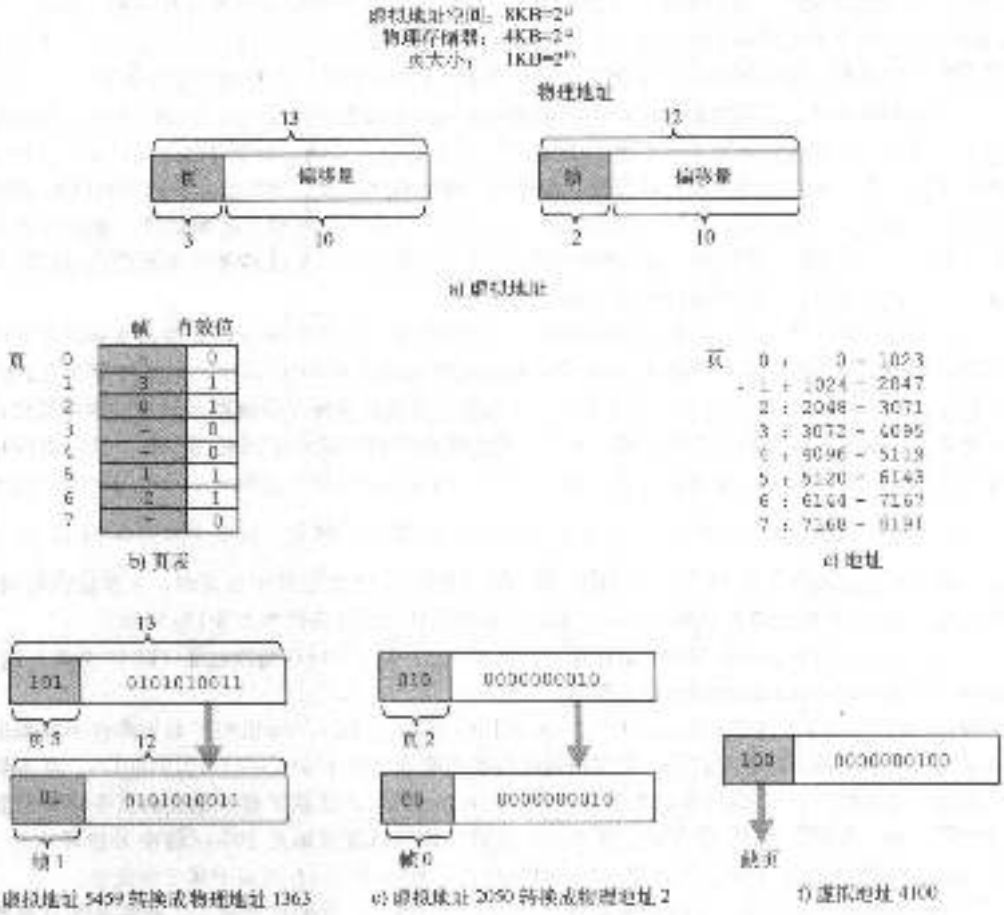


图 6-16 一个较大规模的存储器的例子

通过将最近的页查询数据值存放到一个被称为转换旁视缓冲器（translation look aside buffer，TLB）可以加速页表的查询时间。每个 TLB 的入口目录都由一个虚拟页的页码和对应的物理页帧的帧数组成。对于上面有关页表的例子，表 6-2 给出了一种 TLB 的可能状态。

表 6-2 对应于图 6-16 的 TLB 的当前状态

虚拟页码	物理页码
—	—
3	1
2	0
—	—
—	—
1	3
6	2

处理器通常使用关联高速缓存来实现 TLB，并且虚拟页码和物理页帧数对可以映射到 TLB 高速缓存的任何位置。当使用 TLB 时（参见图 6-17），一个地址的查询过程需要有如下的步骤：

1. 从虚拟地址中提取页码。

2. 从虚拟地址中提取偏移量。

3. 在 TLB 中搜索虚拟页码。

4. 如果在 TLB 中找到虚拟页码和物理页帧数对，则将偏移量加上物理页帧数，并直接访问相应的存储单元。

5. 如果发生一个 TLB 缺失，处理器就从页表中获取所要求的帧数。如果该页已经位于存储器中，就利用物理页帧数加上偏移量的方法生成对应的物理地址。

6. 如果所请求的页不在主存储器中，就会生成一个缺页错误。在完成缺页事件后，需重启存储器访问操作。

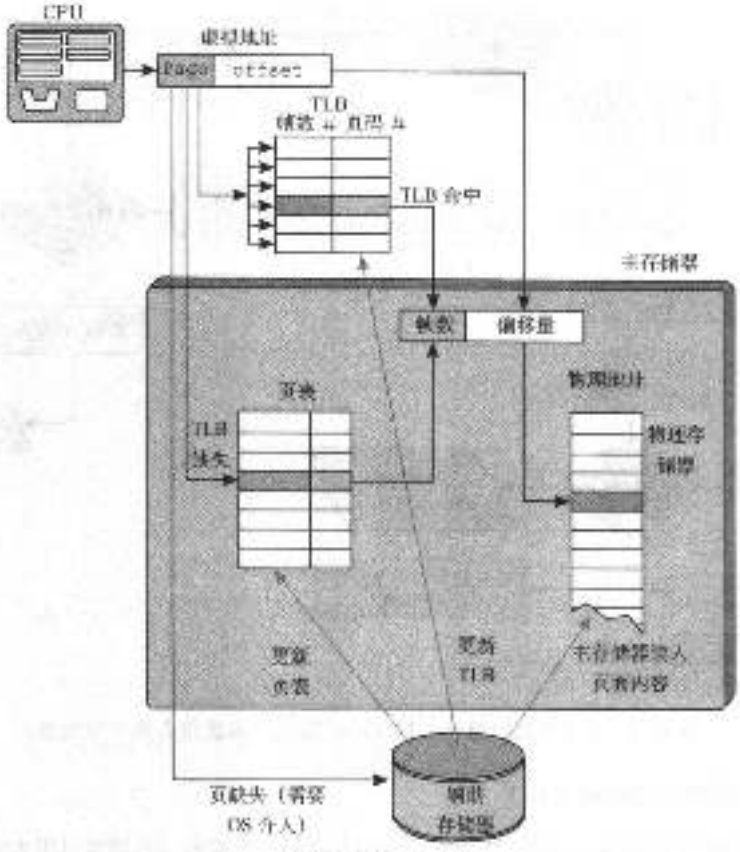


图 6.17 使用 TLB

6.5.3 综合举例：同时使用高速缓存、TLB 和分页

由于 TLB 本质上就是一个高速缓存，所以我们把所有这些概念放在一起可能会引起读者的混淆，但是，对于这类完整过程的讨论无疑会有助于读者全面掌握有关存储器的基本思想。当 CPU 生成某个地址时，指的是相对于程序本身的一个地址，即虚拟地址。在进行数据取回的处理之前，这个虚拟地址必须要转换成物理地址。实现这种地址间的转换任务有两种方法：(1) 通过定位搜索最近使用的高速缓存中的地址对（页/帧对），使用 TLB 来查找所要求的页帧；(2) 如果发生 TLB 缺失事件，则要使用页表在主存储器中查找相应的页帧（通常在该过程中会对 TLB 进行更新），将这个找到的页帧数和虚拟地址中给出的偏移量组合起来即可创建物理地址。

这时，虚拟地址已经被转换为对应的物理地址，但该物理地址的数据还并没有被取回。数据取回

也存在着两种可能性：(1) 搜索高速缓存，查找要求的数据是否已经驻留在高速缓存中；(2) 如果发生高速缓存缺失，处理器必须到实际的主存储单元中取回数据。这个过程通常也会对高速缓存同时进行数据更新。

图 6-18 给出了同时使用 TLB、分页和高速缓存存储器的进程。

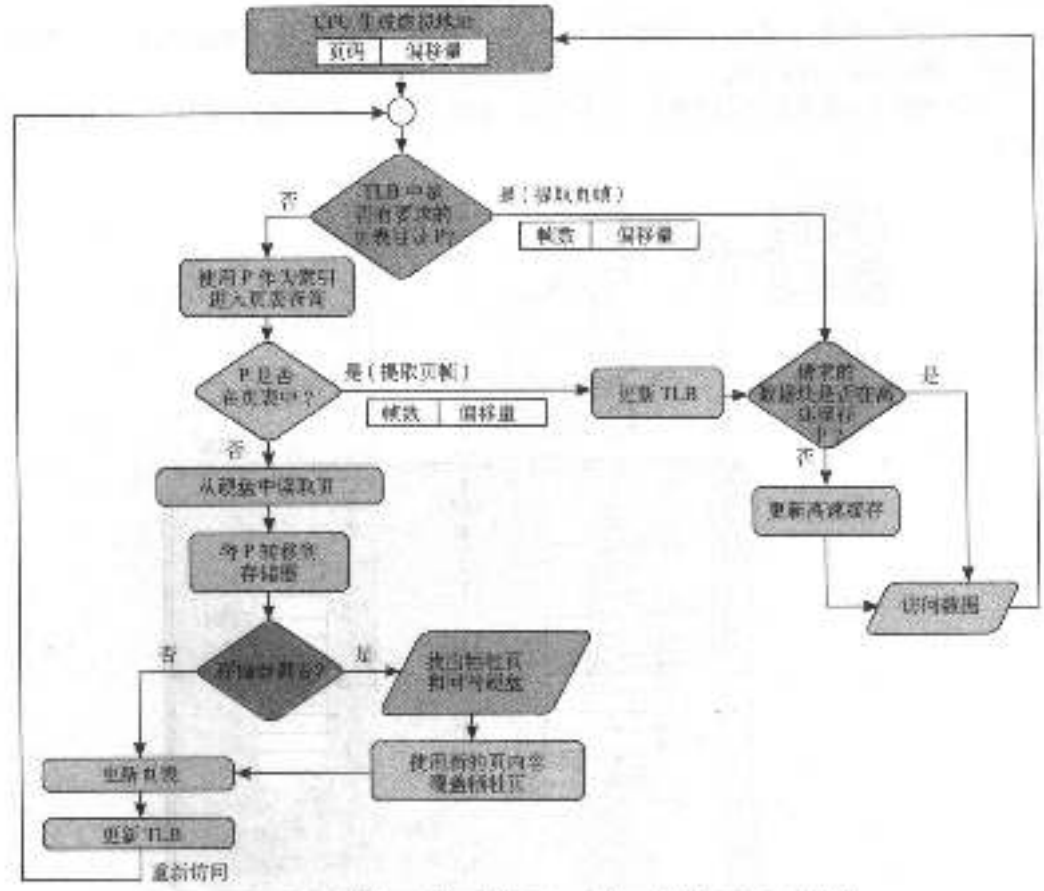


图 6-18 综合举例，组合运用 TLB、页表、高速缓存和主存储器

6.5.4 分页和虚拟存储器的优缺点

第 6.5.2 节讨论了在访问数据时，如何通过分页和增加一次额外的存储器引用来实现虚拟存储器。通过使用 TLB 高速缓存来存放页表的入口目录，可以减少分页的部分时间损失。但是，即使在 TLB 中有很高的命中率，这一进程仍旧会在地址转换上造成重大的开销。使用虚拟存储器和分页的另外一个缺点是，需要消耗额外的系统资源，即需要额外的存储器空间来存放页表。在极端情况下（例如运行一些非常大的程序），页表会占用很大比例的物理存储器。对于这类问题，有一种解决方案就是对页表再进行分页，这样一来使得问题变得更加复杂。此外，要使用虚拟存储器和分页，还必须有专门的硬件和操作系统的支持。

要想从使用虚拟存储器中获益就必须降低其缺点的影响，使得虚拟存储器的应用有利于提高计算机系统的性能。但是，使用虚拟存储器和分页又有什么优点呢？答案非常简单：运行的程序不再受到已有物理存储器容量大小的限制。虚拟存储器允许系统运行虚拟地址空间比实际物理存储器空间大的程序。实质上，虚拟存储器允许进程可以和该进程自身共享物理存储器。这样一来，编写程序的工作变得更简单了，因为程序员不再会为物理地址空间的限制而烦恼，由于每个程序需要的物理存储器空

间较小，因此虚拟存储器可以同时运行更多的程序。允许更多的人共享同一台机器，这使得系统能够处理总的地址空间远远超过系统物理存储器空间的容量，提高了 CPU 的使用率和系统处理能力。

从操作系统的角度来看，使用固定大小的页帧和页面简化了存储器空间的分配和地址的安排问题。而且，分页也允许操作系统以每页为基础实现特定的任务保护（例如，该页属于用户 X，而其他用户不得访问）和共享（例如，该页属于用户 X，其他用户可以阅读其内容）。

6.5.5 分段

虽然分页是目前计算机系统中最常用的方法，但并不是实现虚拟存储器的唯一方法。某些系统使用的实现虚拟存储器的第二种方法是分段（segmentation）机制。与分页不同，分段并不把虚拟地址空间划分为相等的、大小固定的页面，同时将物理地址空间也划分为相同大小的页帧，而是将虚拟地址空间划分为多个长度可变的逻辑单元，或称为段（segment）。物理存储器不再进行实际的空间分割或分区。当需要将某个段复制到物理存储器时，操作系统会自动查找足够大的自由存储空间块来存储整个段。每个段都有一个表示该段在存储器中的位置的基地址（base address）；还有一个指示段的大小的界限（bound limit）。每个程序都是由若干个的段组成。每个程序也都有一个相应的段表（segment table），而不是页表。段表中所包含的只是每个段的基址和界限对的集合。

只要提供段号和段内偏移量，就可以转换存储器的访问。系统还要进行错误校验以确保偏移量是在允许的界限之内。如果偏移量允许，再将该段的基地址值（可以在段表中找到）加到偏移量上面，由此产生实际的物理地址。因为分页采用的是固定大小的存储空间块，而分段使用的是逻辑存储空间块，所以使用分段更容易实现存储空间的保护和共享。例如，程序的虚拟地址空间可以被划分为一个代码段、一个数据段、一个堆栈段和一个符号表段。每个段的长度大小各不相同。系统用户可以很轻松地说："我希望与大家共享我所有的数据，因此数据段对每个人来说都是可访问的"。然后可以说："OK，我的数据存放在某某页上，现在已经找到了那 4 个数据页。其他用户可以访问其中的 3 页，而第 4 页只有其中的一半允许其他用户访问。"

与分页一样，分段也有存储碎片现象。分页产生内部碎片现象，原因在于处理器将一个完整的页帧配置给并不需要占用整个页帧的程序。另外，分段会造成外部碎片（external fragmentation）现象。在进行分段配置和解除分段处理时，存储器中的自由（空）空间块就会变得残缺不完整。最后，存储器中就会留下许多长度较小的自由空间块。但是这些残缺的空间块的大小都不足以存放一个整程序段。外部碎片和内部碎片的区别在于，对于外部碎片，存储器上总的空间足够分配给一个程序进程使用，但是这个存储空间是不连续的，上面存在着大量容量很小、无法使用的空洞。而对于内部碎片，由于系统将过多的存储空间分配给了某个并不需要如此大空间的程序进程，所以无法使用多余的空间。为了应对外部碎片的问题，系统可以使用某种类型的碎片收集（garbage collection）技术。这种处理技术只是对存储空间上已经占用的信息块进行重新调整配置，将磁盘小的碎片空间组合成规模较大的可利用的自由空间块。如果读者曾经进行过硬盘的碎片整理，就已经看到相似的处理过程：将硬盘上许多小的自由存储空间收集并创建少数较大的自由空间。

6.5.6 分页和分段的组合方式

分页与分段的机制并不相同。分页是基于一个纯粹的物理值：即程序地址和主存储器地址都被划分为相同大小的物理空间块。而另一方面，分段是将程序的逻辑地址部分划分为长度可变的不同分区。使用分段，用户可以知道段的大小和界限；而使用分页，用户并不知道存储空间的具体划分。相对来说，分页技术比较容易管理。当所有的元素都具有相同大小时，分配空间、释放空间、空间交换和重新部署等操作都会变得较为方便。但是，一般情况下分页的数目要比分段数目少，这意味着分页技术需要更多的系统开销，主要指页面记录和页面转换所占用的资源。分页消除了外部碎片问题，而分段却可以避免内部碎片问题。分段可以支持实现段的共享和保护，而对于分页则很难做到这些。

很明显，分页和分段的方式各有优点；而系统也不是只能使用其中的一种方案。通常，可以将这两种方法组合起来使用，以便获得最佳的效果。在组合方式中，虚拟地址空间被分割成一些长度可变的段，而这些段又被划分成许多固定大小的页面。主存储器的物理空间也对等地划分为相同大小的帧。

每个段都有一个页表，这就是说每个程序都有多个页表。系统的物理地址被划分为三个域。第一个域是段域，指示系统所对应的页表；第二个域是页码，用作进入页表的偏移量。第三个域是页内的偏移量。

使用分段和分页的组合方式对于存储器的管理非常有益，这种组合方法既可以从用户的角度来实现分段管理，也可以从系统的角度实现分页管理。

6.6 存储器管理实例

因为奔腾系列处理器具有现代存储器管理的显著特征，下面简单回顾一下奔腾处理器是如何处理有关存储器的问题。

奔腾处理器使用32位虚拟地址和32位物理地址。奔腾结构的分页功能所使用的页面大小为4KB或4MB。并且可以采用分页和分段管理机制的不同组合形式：包括没有分段和分页的存储器；有分页、没有分段的存储器；有分段、没有分页的存储器；以及分段和分页组合方式的存储器系统。

奔腾处理器采用两级高速缓存：L1和L2，它们都使用一个32位大小的存储区块。L1靠近处理器，而L2位于处理器和存储器之间。L1实际上由两个高速缓存组成：奔腾处理器（像大部分计算机一样）将L1分割为用于保存指令的高速缓存（称为I-Cache）和保存数据的高速缓存（称为D-Cache）。这两个L1 Cache都使用了一个LRU位来处理高速缓存块的置换操作。每个L1都有一个TLB（快表）：D-Cache的TLB具有64个入口目录，而I-Cache只有32个入口目录。这两个TLB都是利用4路的组关联映射方式，并且采用了一个伪LRU置换算法。而L1的D-Cache和I-Cache都使用2路的组关联映射方式。L2的缓存可以从512KB（早期型号的产品）增加到1MB（后期型号的产品）。L2 Cache与两个L1 Cache一样，都是采用2路的组关联映射方式。

为了管理对存储器的访问操作，奔腾处理器的I-Cache和L2 Cache都采用MESI高速缓存的一致性协议。每一路高速缓存线都使用2位的二进制数来存储下列MESI状态中的其中一种状态：(1) M：被修改过的，即高速缓存中的数据与对应的主存储器中不同；(2) E：独占的，即高速缓存的内容没有被修改过，并且与主存储器相同；(3) S：共享的，即高速缓存的线/块可以与其他高速缓存的线/块共享；(4) I：无效的，即所请求的线/块都不在高速缓存中。图6-19给出了奔腾处理器的存储器的层次结构。

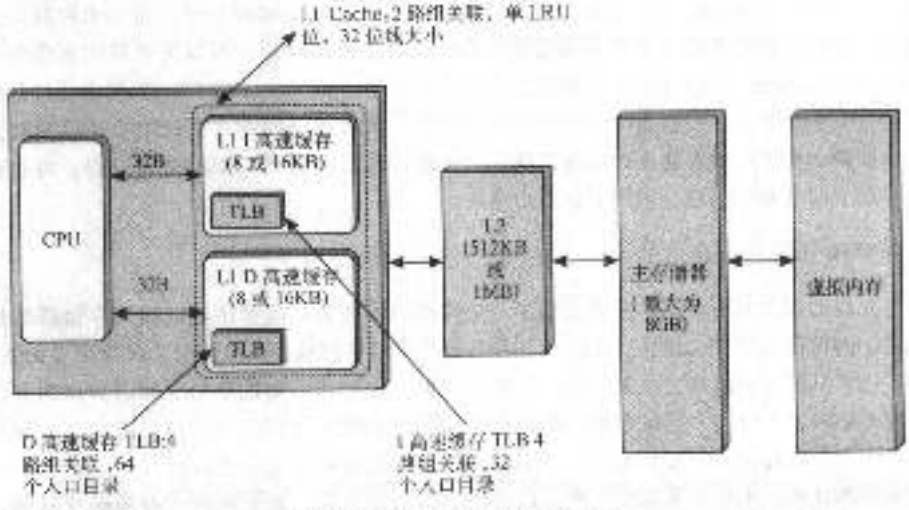


图6-19 奔腾存储器的层次结构

这里只是对奔腾处理器的存储器和存储器的管理方法做了一个简要基本的概述。有兴趣的读者，可以参阅本章后面的深入阅读部分。

本章小结

存储器是按层次结构的组织的存储容量较大的存储器的价格比较低廉，但速度却比较慢；而容量较小的存储器系统的速度比较快，但却价格不菲。一个典型的存储器层次结构一般包括：高速缓存、主存储器和辅助存储器（通常为磁盘驱动器）。局部性的原理有助于我们跨越各个相邻的不同层次结构之间的间隔，将存储器系统构建成一个整体结构。这样，对于程序员来说，看到的是一个很大且非常快的存储器系统，而无需考虑这种层次结构中不同层次之间数据转移的细节问题。

高速缓存的作用实际上是一个缓冲器，它的位置靠近 CPU。可以利用高速缓存来保存最频繁使用的主存储器中的信息块。采用存储器层次结构的其中的一个目的是，希望处理器可以获得一个非常接近高速缓存访问时间的存储器有效存取时间。是否能够达到这一目的，还要取决于所执行的程序的行为特征，高速缓存的组织方式和容量大小，以及高速缓存的置换策略。处理器要引用的内容如果能够直接在高速缓存中找到，称为高速缓存命中；如果在高速缓存中找不到，则称为高速缓存缺失。对于缺失事件，没有找到的所需的数据必须从主存储器中提取。在处理缺失事件时，处理器会将包含缺失数据的整个数据块从主存储器调到高速缓存中。

对于不同的存储器地址，高速缓存的组织结构决定了 CPU 搜索高速缓存的方式。高速缓存可以采用不同方式的组织结构：直接映射、全关联的映射，或者是组关联的映射方式。直接映射的高速缓存不需要使用置换算法。然而，全关联和组关联的映射方式，在高速缓存被装满后，都必须使用 FIFO、LRU 或者是其他的分配策略来确定要从高速缓存中移除的某个信息块，以便为新的信息块留出存放的空间。采用 LRU 算法可以取得最好的置换效果，但实现起来却十分困难。

使用存储器的层次结构的另外一个目的是，利用硬盘本身来扩充主存储器的容量，这种方法也称为虚拟存储器。利用虚拟存储器技术，可以在计算机上运行虚拟地址空间比物理存储器大的程序。虚拟存储器技术还允许多个程序进程在系统中并行运行。使用分页的管理机制可以实现对虚拟存储器的管理。但是分页的方法存在一些缺点：包括需要消耗额外的系统资源（例如存放页表所需的资源）和增加额外的存储器访问操作（系统首先要访问页表）。使用一个 TLB（快表）高速缓存来存放最近使用的虚拟地址和物理地址对可以克服这些缺点。虚拟存储器在将虚拟地址转换成物理地址时，也会产生转换开销（translation penalty）。而虚拟存储器在处理缺页事件，即所请求的页在当时并没有驻留主存储器而是在磁盘上时，也会导致时间上的额外损失（开销）。虚拟存储器和主存储器之间的相互关系与主存储器和高速缓存之间的关系非常相似。由于这种相似性，高速缓存存储器的概念和 TLB 的概念常常容易混淆。实际上，TLB 就是一个高速缓存。必须强调的是，在处理其他任何事件之前，必须首先将虚拟地址转换为物理地址，这也正是 TLB 要做的事情。虽然高速缓存存储器和使用分页的虚拟存储器看起来很相似，但它们要实现的目的是完全不同的。使用高速缓存的目的是为了改善对主存储器的有效存取时间，而分页的使用是为了扩充主存储器的存储容量。

深入阅读

要学习有关 RAM 的更多的内容，可以阅读 Mano（1991）和 Stallings（2000）的著作。Hamacher，Vranesic 和 Zaky（2002）在他们的书中深入讨论了高速缓存存储器。对于虚拟存储器，可以参阅 Stallings（2001），Tanenbaum（1999），或 Tanenbaum 和 Woodhull（1997）的著作。如果想了解有关存储器管理的更多知识，可以查阅 Flynn 和 McHoes（1991），Stallings（2001），Tanenbaum 和 Woodhull（1997），或 Silberschatz，Galvin 和 Gagne（2001）的书籍。Hennessy 和 Patterson（1996）讨论了有关决定高速缓存性能的问题。有关存储器技术的在线课程可以浏览网页 www. kingston. com/tools/umg。George Mason 大学同样也提供了一系列有关各种计算机课题的工作平台。对于虚拟存储器的工作平台，可以访问网址：cne. gmu. edu/workbenches/vmsim/vmsim. html。

参考文献

Davis, W. *Operating Systems, A Systematic View*, 4th ed., Redwood City, CA: Benjamin/Cummings, 1992.

Flynn, I. M., & McHoes, A. M. *Understanding Operating Systems.* Pacific Grove, CA: Brooks/Cole, 1991.

Hamacher, V. C., Vranesic, Z. G., & Zaky, S. G. *Computer Organization*, 5th ed., New York: McGraw-Hill, 2002.

Hennessy, J. L., & Patterson, D. A. *Computer Architecture: A Quantitative Approach*, 2nd ed., San Francisco: Morgan Kaufmann, 1996.

Mano, Morris. *Digital Design*, 2nd ed., Upper Saddle River, NJ: Prentice Hall, 1991.

Silberschatz, A., Galvin, P., & Gagne, G. *Operating System Concepts*, 6th ed., Reading, MA: Addison-Wesley, 2001.

Stallings, W. *Computer Organization and Architecture*, 5th ed., New York: Macmillan Publishing Company, 2000.

Stallings, W. *Operating Systems*, 4th ed., New York: Macmillan Publishing Company, 2001.

Tanenbaum, A. *Structured Computer Organization*, 4th ed., Englewood Cliffs, NJ: Prentice Hall, 1999.

Tanenbaum, A., & Woodhull, S. *Operating Systems, Design and Implementation*, 2nd ed., Englewood Cliffs, NJ: Prentice Hall, 1997.

基本概念和术语复习

1. SRAM 和 DRAM 哪一个速度更快？
2. 主存储器使用 DRAM 有什么好处？
3. 举出经常使用 ROM 的三种不同应用。
4. 解释存储器层次结构的基本概念。为什么本书会把存储器的层次结构描述为金字塔的形式？
5. 解释引用局部性的概念，并且阐述局部性原理对于存储器系统的重要性。
6. 计算机中存在哪三种形式的局部性？
7. 列举两个非计算机的高速缓存的例子。
8. L1 Cache 和 L2 Cache：哪一个的速度更快？哪一个的规模更小，为什么？
9. 高速缓存是利用其_____进行访问的，而主存储器是利用其_____进行访问的。
10. 直接映射的高速缓存中地址的三个域分别是什么？如何使用这些域来访问存放在高速缓存中的一个数据字。
11. 关联存储器与常规的存储器有何区别？其中哪种存储器价格更贵，为什么？
12. 解释全关联高速缓存和直接映射高速缓存之间有何区别。
13. 组关联高速缓存是如何将直接映射高速缓存的思想与全关联高速缓存的思想结合在一起的？
14. 直接映射的高速缓存通常被认为是组关联高速缓存中的组块大小为 1 的一种特例。那么，完全关联高速缓存又会是组关联的组块大小为_____的一种特例。
15. 组关联高速缓存的地址分成哪三个域？如何使用这三个域来访问高速缓存中的某个存储单元？
16. 解释说明本书中所介绍的四种高速缓存置换策略。
17. 为什么最佳高速缓存置换策略对高速缓存的性能非常重要？
18. 采用 LRU 和 FIFO 置换策略对高速缓存行为可能产生的最坏情况是什么？
19. 严格来说，什么是有效存取时间（EAT）？
20. 说明如何导出有效存取时间的公式？
21. 在什么情况下，高速缓存的作用会很差？
22. 什么是脏块？

23. 分别描述两种高速缓存写策略的优缺点？
24. 虚拟存储器地址和物理存储器地址有何不同？哪一个比较大？为什么？
25. 分页的目的是什么？
26. 讨论有关使用分页的正反两方面的意见。
27. 什么是缺页？
28. 造成内部碎片的原因是什么？
29. 虚拟地址的各个组成部分（域）是什么？
30. 什么是 TLB？TLB 是如何改善 EAT 的？
31. 虚拟存储器有何优点和缺点？
32. 什么情况下，系统需要对页表进行分页处理？
33. 什么原因会造成外部碎片？如何解决？

练习题

◆1. 假设某台计算机使用直接映射的高速缓存。计算机的主存储器的容量为 2^{20} 字，高速缓存有 32 个存储空间块，且每个高速缓存块包含 16 个字。问：

◆a) 主存储器可以划分为多少个信息块？

◆b) 从高速缓存的角度来看，存储器地址采用的是什么格式？即分别说明地址中的标记域、块域和字域的大小。

◆c) 存储器引用时，地址 $0DB63_{16}$ 将会被映射到高速缓存中的哪个空间块？

2. 假设某台计算机使用直接映射的高速缓存，计算机的主存储器的容量为 2^{32} 字，高速缓存有 1024 个存储空间块，且每个高速缓存块包含 32 个字。问：

a) 主存储器可以划分为多少个信息块？

b) 从高速缓存的角度来看，存储器地址采用的是什么格式？即分别说明地址的标记域、块域和字域的大小。

c) 存储器引用时，地址 $00063FA_{16}$ 将会被映射到高速缓存的哪个空间块？

◆3. 假设某台计算机使用全关联高速缓存，其主存储器的容量为 2^{16} 字，高速缓存中有 64 个存储空间块，每个高速缓存块包含有 32 个字。问：

◆a) 主存储器中有多少个信息块？

◆b) 高速缓存可以看到的存储器地址格式是什么样的？即分别说明地址的标记域、块域和字域的大小。

◆c) 存储器引用时，地址 $F8C9_{16}$ 将会被映射到高速缓存的哪个存储空间块？

4. 假设某台计算机使用全关联高速缓存，其主存储器的容量为 2^{24} 字，高速缓存有 128 个存储空间块，每个高速缓存块包含 64 个字。问：

a) 主存储器中有多少个信息块？

b) 高速缓存可以看到的存储器地址的格式是什么样的？即分别说明地址的标记域、块域和字域的大小。

c) 存储器引用时，地址 $01D872_{16}$ 将会被映射到高速缓存的哪个存储空间块？

◆5. 假设系统的存储器有 128MB 字，块长度为 64 个字，高速缓存由 32KB 块组成。如果使用 2 路的组关联高速缓存的映射方式，说明主存储器地址的格式，并确定地址中的各个域和域的大小。

6. 一个由 4 个块组构成的 2 路的组关联高速缓存。主存储器有 2KB 块，每块包含 8 个字。

a) 说明主存储器的地址格式，可以利用这种格式将主存储器地址映射到高速缓存存储器，并确定地址中的各个域和域的大小。

b) 如果一个程序执行三次从主存储器单元 8_{10} 到 51_{10} 的循环操作，请计算其命中率。这里可以使用分数的形式来表示命中率。

7. 假设某台计算机采用组关联高速缓存，其主存储器的容量为 2^{16} 字，高速缓存有 32 个存储空间块，并且每个高速缓存块包含 8 个字。问：

a) 如果高速缓存为2路的组关联高速缓存，那么从高速缓存的角度来看，主存储器的地址是什么格式？其中，标记域、组域和字域的长度分别是多少位？

b) 如果高速缓存为4路的组关联高速缓存，那么高速缓存所看到的存储器的地址是什么格式？

8. 假设某台计算机采用组关联高速缓存，其主存储器的容量为2^n字，高速缓存有64个存储空间块，每个高速缓存块包含4个字。问：

a) 如果高速缓存为2路的组关联高速缓存，那么从高速缓存的角度来看，主存储器的地址是什么格式？其中，标记域、组域和字域的长度分别是多少位？

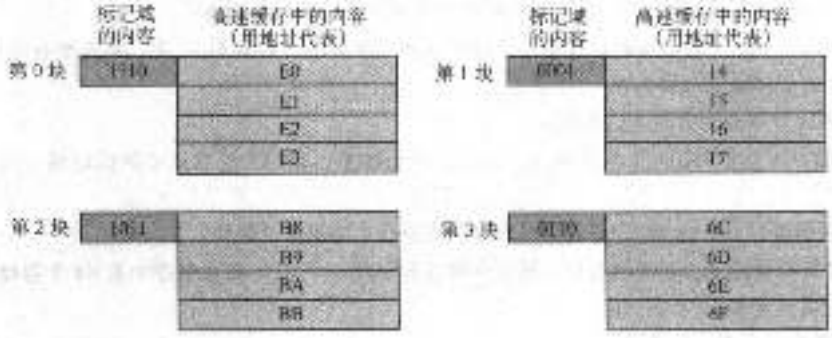
b) 如果高速缓存为4路的组关联高速缓存，那么高速缓存所看到的存储器的地址是什么格式？

*9. 假设某台计算机使用一个8位长度的存储器地址字。计算机还有一个16字节的高速缓存，其中每个高速缓存块包含4个字节。在一个程序的运行过程中，计算机需要访问若干数目的存储器单元。

现在假设，系统采用的是直接映射的高速缓存。从高速缓存所看到的存储器地址格式为：



程序执行时，系统将按照如下的顺序访问存储器地址（用十六进制表示）：6E，B9，17，E0，4E，4F，50，91，A8，A8，AB，AD，93和94。现在，假设前4个访问的存储器地址已经装入到高速缓存块中，如下图所示。其中，标记域的内容使用二进制数来表示，而高速缓存块中的内容部分只是简单地写上存放在该高速缓存单元的主存储器地址。问：



a) 在上述的存储器引用的整个顺序过程中的命中率是多少？

b) 当最后的一个地址访问结束后，存储器中的哪个数据块会存放在高速缓存中？

10. 一个使用直接映射的高速缓存由8个存储空间块组成。主存储器有4KB数据块，每块包含8个数据字。对高速缓存的访问时间为22ns，而将数据块从主存储器装入到某个高速缓存条所需要的时间是300ns（这一时间包括了系统确定高速缓存缺失块和将相关内容调入高速缓存的时间）。假设系统是并行地开始向高速缓存和主存储器发出访问请求。因此，如果在高速缓存中未找到所请求的项目，系统并不会将这个高速缓存的搜索时间增加到存储器的访问时间上。如果在高速缓存中没有找到某个信息块，则系统将整个块都装入到高速缓存中，并且重新访问存储器。最初，高速缓存是空的。

a) 说明主存储器的地址格式，能够利用这种地址格式实现从主存储器到高速缓存的映射。确定地址域的划分和域的大小。

b) 如果一个程序执行从主存储器单元0到67_{16}的4次循环操作，请计算其命中率。

c) 计算程序的有效存取时间。

11. 考虑一个按字节编址的计算机系统，它具有24位地址，一个可以共存放64KB数据的高速缓存存储器，以及32字节的块。对于如下的高速缓存方式，给出24位存储器地址的格式：

a) 直接映射

b) 关联

c) 4路组关联

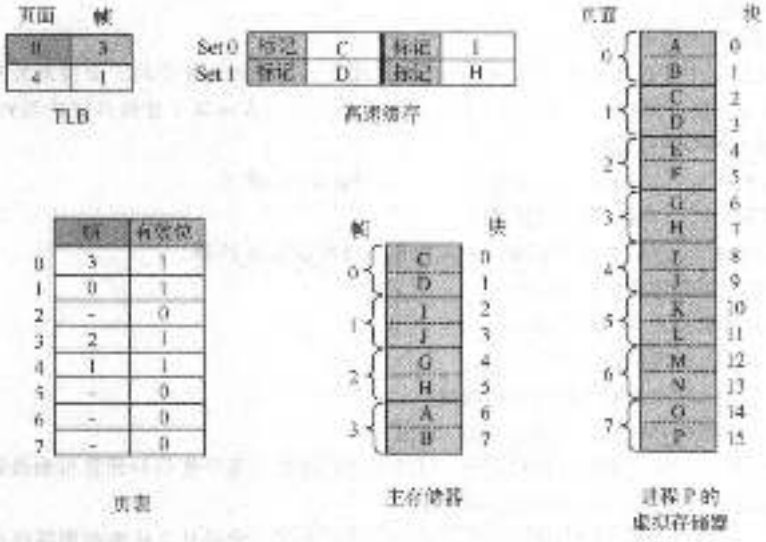
12. 假设某个进程的页表有如下（左）入口内容，采用图 6-15a 中的格式，指出该进程的各页在存储器中的位置。

◆13. 假设某个进程的页表有如下（右）入口内容，采用图 6-15a 中的格式，指出该进程的各页在存储器中的位置。

帧	有效位
1	1
-	0
0	1
3	1
-	0
	0
2	1
	0

帧	有效位
-	0
3	1
-	0
-	0
2	1
0	1
-	0
1	1

* 14. 一个虚拟存储器配有一个 2 入口的 TLB，一个 2 路组关联的高速缓存，以及一个进程 P 的页表。假设高速缓存块的大小为 8 个字和页面的大小为 16 个字。在下面的系统中，主存储器被划分为若干个数据块，其中每个数据块都由一个字母表示。两个块等于一个页帧。



如果系统的状态如上所示，试回答下列问题：

a) 进程 P 的虚拟地址为多少位？为什么？

b) 物理地址为多少位？为什么？

c) 求出虚拟地址 18_{10} 的地址格式（指定域名和大小），系统使用这个地址格式向物理地址进行转换操作，并随后将这个虚拟地址转换成相应的物理地址。（提示：先将 18 转换为等效的二进制数，然后将地址划分为对应的域）。解释各个域如何转换为对应的物理地址。

d) 如果已知虚拟地址 6_{10} 转换为物理地址 54_{10}，给出物理地址的形式（说明域名和大小）。通过这个地址形式可以确定该地址在高速缓存中的位置。并解释如何利用该地址形式来确定物理地址 54_{10} 在高速缓存中的位置（提示：将 54 转换为二进制数，并分为对应的域）。

e) 如果已知虚拟地址 25_{10} 位于虚拟页的第 1 页，偏移量为 9 的位置。正确指出该虚拟地址如何被转换

为对应的物理地址，以及如何访问所要求的数据。答案中应该包括如何使用 TLB、页表、高速缓存和存储器。

15. 已知一个虚拟存储器有一个 TLB、一个高速缓存和一个页表。并做出如下的各种假设：
    - 一次 TLB 命中需要 5ns。
    - 一次高速缓存命中需要 12ns。
    - 一次存储器的调用需要 25ns。
    - 一次磁盘的调用需要 200ms（此时间包括更新页表、高速缓存和 TLB 的时间）。
    - TLB 的命中率为 90%。
    - 高速缓存的命中率为 98%。
    - 页出错率为 0.001%。
    - 对于一次 TLB 或高速缓存的缺失事件，访问存储器需要的时间包括了 TLB 和或者高速缓存的更新时间，但是不会重启访问过程。
    - 对于一次页错误，缺页需要从磁盘提取，并执行所有的更新过程，需重启访问过程。
    - 所有的引用过程都按顺序进行（没有重叠发生，也没有并行过程）。

    试指出下面的各种情况是否可能发生。如果可能发生，求出访问指定数据所需要的时间：

    a) TLB 命中，高速缓存命中。

    b) TLB 缺失，页表命中，高速缓存命中。

    c) TLB 缺失，页表命中，高速缓存缺失。

    d) TLB 缺失，页表缺失，高速缓存命中。

    e) TLB 缺失，页表缺失。

    写出计算有效访问时间的方程式。

16. 假设某个系统对每个进程都采用一个一级页表来实现分页的虚拟地址空间。虚拟地址空间的最大容量为 16MB。运行程序进程的页表包含如下的合法入口（符号→表示某个虚拟页被映射到给定的页帧中，即虚拟页在该页帧中的位置）：

    虚拟页 2→页帧 4　　虚拟页 4→页帧 9　　虚拟页 1→页帧 2

    虚拟页 3→页帧 16　　虚拟页 0→页帧 1

    如果页面大小为 1024 字节，并且机器的最大物理存储器容量为 2MB。

    a) 每个虚拟地址需要用多少位表示？

    b) 每个物理地址需要用多少位表示？

    c) 页表中的最大入口数目是多少？

    d) 虚拟地址 1524_{10} 转换为哪个对应的物理地址？

    e) 什么样的虚拟地址可以转换为物理地址 1024_{10}？

17. a) 如果你是一个力图使自己的计算机系统尽可能地具有价格上竞争优势的计算机制造商，你会选择什么样的功能部件和组织结构来构建存储器结构？

    b) 如果你是一个想要获得最佳性能的计算机的购买者，你又会选择什么样的功能部件和组织结构来构建存储器结构？

* 18. 考虑一个多处理器的系统，其中的每个处理器都带有自己的高速缓存，但是所有的处理器都共享一个主存储器系统。问：

    a) 应该采用哪种高速缓存写策略？

    b) **高速缓存一致性问题**：设想一下，对于这个多处理器系统，如果其中一个处理器的高速缓存中存放存储器的数据块 A 的副本，而在第二个处理器的高速缓存中也有同样的一个数据块 A 的副本，然后更新主存储器的数据块 A，会产生什么问题？设计一种方法（可能一种以上的方法）避免这种情况的发生，或降低其影响。

* 19. 挑选一个特定的体系结构（本章没有介绍的体系结构）进行仔细研究，揭示这种体系结构是如何接近本章所引入的那些基本概念，就像 Intel 公司的奔腾处理器所做的那样。

Who is General Failure and why is he reading my disk?

——无名氏

第7章　输入/输出和存储系统

7.1　概述

如果我们不能将数据输入计算机并从计算机中提取信息，那么这种计算机是毫无用处的。拥有一台不能有效完成输入输出操作的计算机，其实也并不会比没有计算机好多少。当计算机的处理时间超过用户的“思考时间”时，用户常常会抱怨这台计算机的速度太“慢”。有时，计算机“慢速”的问题可能会对用户的工作效率造成实质的影响，这种影响的程度常常可以用金钱来衡量。但是，通常引发这种问题的根源并不在处理器或内存，而在于计算机如何处理系统的输入和输出。

I/O不仅仅是文件的存取。一个性能不佳的I/O系统可能会导致某种连锁反应，降低整个计算机系统的性能。在前面的章节里，我们介绍了虚拟存储器。即，系统如何利用磁盘对存储器中的数据块进行页面调度，以便在主存储器中为更多的用户进程设置运行空间。如果磁盘系统的速度很慢，那么进程执行的速度也就会减慢，由此导致CPU和磁盘队列中积压很多未执行的任务。解决这个问题最简单的办法是增加更多的系统资源。例如，购买更大的主存储器，更快的处理器。如果在增加系统资源方面受到限制，则可以简单地限制计算机中并发执行的进程的数量。

采用这些简单的解决方案，难免会造成系统资源的浪费。当我们真正理解了计算机系统的工作过程后，就可以最有效地利用现有的系统资源，而只是在真正需要的时候才会添加昂贵的资源。本章的目的是介绍I/O和存储系统以及一些优化系统的方法，让读者能够对磁盘存储系统做出合适的选择。本书最大的愿望是大家能使用这些内容作为今后进一步学习的一个跳板——甚至可以进行创新工作。

7.2　AMDAHL 定律

每次当某家微处理器公司发布其最新和最强的CPU时，这些技术上的飞跃都会成为全球媒体的焦点。全球的计算机的权威人士也认为这种进步应该值得称赞和炫耀。然而，在I/O技术中取得类似的进步时，可能只是在某些普通商业杂志的某页中略加介绍。在媒体的大肆渲染下，人们很容易忽视计算机系统的整体特性。很明显，计算机中某个部件的速度提升40%，肯定不会带来整个系统的速度提升40%。然而，媒体的宣传效果却常常会给读者造成一种相反的错觉。

1967年，George Amdahl发现了计算机中组成部件与计算机系统的整体工作效率之间的相互关系。Amdahl使用了一个公式把他观察的结果定量化，这就是大家所熟知的Amdahl定律。从实质上来说，Amdahl定律表述的是：计算机系统整体性能的速度提升（称为加速率，speedup）取决于某个特定部件本身的加速率和该部件在系统中的使用率。用公式表示为：

$$S = \frac{1}{(1-f)+f/k}$$

其中：

S 代表系统整体性能的加速率；

f 表示较快部件完成的工作部分；

k 是新部件的加速率。

假如在你的大部分日常处理工作中计算机需要花费70%的时间来执行CPU操作和30%的时间来等待磁盘服务。现在，假设有人试图卖给你一个处理器的升级换代产品，比你现有的处理器速度快50%，价格是10 000美元。而就在前一天，曾经有人打电话给你。为你提供一组价格7000美元的磁盘

驱动器。这种新磁盘系统的吞吐量（处理能力）是你现有磁盘系统的2.5倍。你很清楚你的计算机系统的性能正在开始落伍，因此你想要做某些升级处理。为了达到用最少的钱取得最大的性能改善的目的，你会选择上面的哪一种升级方案？

如果选择处理器，有：

$$f = 0.70, k = 1.5, \text{所以 } S = \frac{1}{(1-0.7)+0.7/1.5} = 1.30$$

因此，购买新的处理器要花费10 000美元，并且可以使系统整体性能的速度提升1.3倍或30%。

如果选择磁盘，有：

$$f = 0.30, k = 2.5, \text{所以 } S = \frac{1}{(1-0.3)+0.3/2.5} \approx 1.22$$

即磁盘升级的花费为7000美元，所带来的系统的加速率是1.22倍或22%。

初看起来，这两种选择方案的效果几乎是一样的，很难做出决定。对于升级处理器的方案，系统的性能每提升1%所要付出的代价是333美元。而通过升级磁盘，每增加1%的系统性能所要花费的代价为318美元。如果仅仅基于性能改善的每个百分点所花费的成本来判断的话，显然升级磁盘驱动器是一个稍好的选择方案。当然，还有其他的一些因素也会影响到你的决定。例如，磁盘寿命已经接近尾声，或者磁盘的空间已经耗尽。这种情形下，即使升级磁盘比升级处理器的费用高，你也可能会优先考虑对磁盘进行升级。

但是，在决定升级磁盘之前，最好先了解清楚其他选择方案。下面内容将介绍通用的输入输出(I/O)体系结构，并特别强调有关磁盘的I/O系统。磁盘的I/O系统紧随在CPU和存储器之后，是决定计算机系统的整体效能的重要因素。

7.3 输入/输出体系结构

输入/输出（I/O）定义为在外部设备和由CPU及主存储器组成的主机系统之间移动编码数据的一个子系统部件。输入输出子系统包括（但不限于）以下部分：

- 用于I/O功能的主存储器模块
- 提供将数据从系统中移入和移出所需要的总线通道
- 主机和外围设备中的控制模块
- 连接外部元件的接口，例如，连接键盘和磁盘
- 连接主机系统和外围设备的电缆或其他通信链接

图7-1描述的是如何将所有这些部件组合起来，构成一个集成的I/O子系统。I/O模块主要负责主存储器和某个特定设备的接口之间的数据传递。接口是专门设计的电路，用来与某种类型的外围设备通信，例如键盘、磁盘或打印机。接口用来处理某些通信方面的细节问题，比如确保外围设备已经为下一批的数据传送准备就绪，或者确认主机已经准备好接收来自外部设备的下一批数据。

在发送设备和接收设备之间交换的各种信号的具体形式和信号所代表意义称为协议（protocol）。协议包括：命令信号，例如“打印机重置”；状态信号，例如“磁带就绪”；或数据传递信号，例如“这就是所请求的字节”。在大多数数据交换协议中，接收设备必须对命令和发送过来的数据做出应答，或者指示接受设备已经准备好接收数据。这种类型的协议交换被称为“握手”(handshake)。

负责处理大量数据信息的外部设备（例如打印机、磁盘和磁带驱动器）通常都有缓冲存储器。缓冲器允许主机系统以尽可能最快的方式将大量数据发送到外围设备中，而不必等待直到慢速的机械设备实际完成数据的写入操作。磁盘驱动器的专用存储器通常是速度很快的高速缓存，然而打印机上通常配备较慢的RAM。

设备控制电路负责从系统的缓冲器中提取或输入数据，以确保在需要时可以及时获取数据。在写磁盘时，设备控制电路的工作还包括对磁盘进行合适的定位以确保数据可以写到某个特定的磁盘区域。

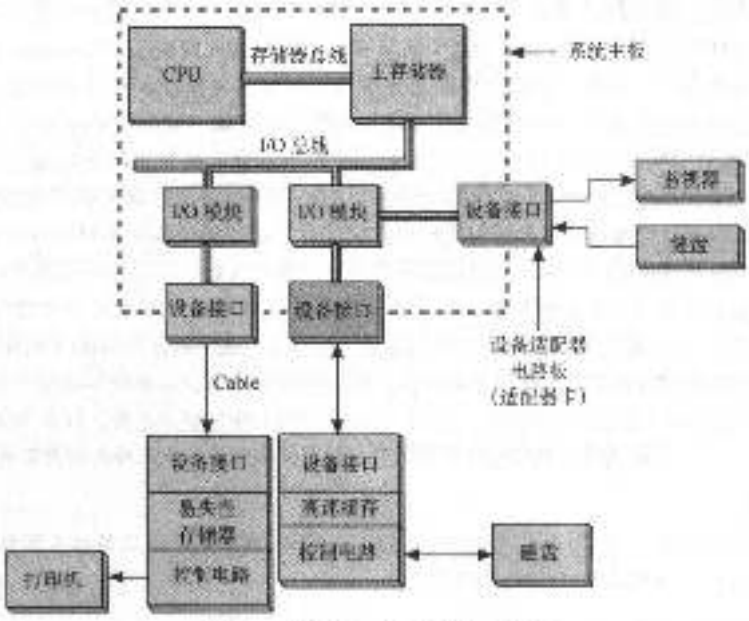


图 7.1 I/O 结构模型的基本配置

面对打印机来说，设备控制电路还要完成将打印头或激光束移动到下一字符位置上，启动打印头，弹出打印纸等操作。

磁盘和磁带属于持久性存储（durable storage）的形式，之所以这样说，是因为它们与易失性的主存储器相比，其上的数据可以保存较长的时间。然而，没有一种存储方法是永久性的，数据在磁性介质上可以保存大约 5 年，而在光学介质上大约能保存 100 年。

7.3.1 I/O 的控制方法

计算机系统通常有 4 种 I/O 控制方法，包括：程序控制的 I/O、中断控制的 I/O、直接存储器存取和通道控制的 I/O。虽然这些方法各有利弊，但是计算机所采用 I/O 的控制方法将会严重影响系统的整体设计和性能。我们的目的是让读者了解特定的计算机体系结构中所采用的 I/O 控制方法是否适合该系统的应用方式。

程序控制的 I/O

使用程序控制的 I/O（programmed I/O）的计算机系统需要为每个 I/O 设备至少准备一个专用的寄存器。CPU 会持续不断地监视每个寄存器，等待数据的到达。这种方法称为轮询（polling）。因此，程序控制的 I/O 有时也被称为轮询的 I/O（polled I/O）。一旦 CPU 检测到某个“数据就绪”的条件，就会根据为这个特定寄存器所准备的指令执行各种操作。

这种方法的好处是可以通过编程来控制每个外部设备的行为，通过改变程序就可以调整系统所控制的外部设备的数目和类型，以及轮询的权限和时间间隔。但是，不断地对寄存器轮询是一个比较麻烦的问题。采用这种方式工作的 CPU 会持续处于一个“繁忙等待（busy wait）”循环中，直到开始服务某个 I/O 请求。如果没有 I/O 任务要进行处理，CPU 就无法从事任何有用的工作。由于存在着这些限制，程序控制的 I/O 最适合应用在某些专用系统上，例如自动取款机和一些用来控制或监视外部事件的系统。

中断控制的 I/O

中断控制的 I/O（interrupt-driven I/O）与程序控制的 I/O 正好相反。使用中断控制方式的 CPU 不再需要持续地查询其附属设备是否有任何输入请求，而是在有数据发送需求时由外部设备来通知

CPU。如果没有外部设备发出服务请求来中断CPU，CPU就可以继续执行其他任务。通常，系统使用CPU的标志寄存器中的一个二进制位来指示中断信号，该位称为中断标志（interrupt flag）。

一旦设置了中断标志，操作系统就会中断正在执行的程序，并保存该程序的状态及各种可变的信息。然后，系统会提取有关指示I/O中断服务程序地址的地址矢量（address vector）。在完成I/O中断服务后，CPU会恢复中断发生时所保存的正在执行的程序的信息，并且重新开始继续执行原程序。

中断控制的I/O和程序控制的I/O的相似之处是，它们都可以对I/O服务程序进行修改来适应外部硬件设备的改变。对于运行相同类型和级别的操作系统的计算机，属于各种不同硬件类型的地址矢量通常会保存在系统中的相同位置。因此，我们很容易改变这些地址矢量，使其指向专用设备的代码。例如，如果有人需要使用一种新型的磁盘驱动器，但是目前流行的操作系统并不支持这个磁盘驱动器。那么，这种磁盘驱动器的生产厂家就会更新磁盘I/O地址矢量，指向属于这个磁盘驱动器的专用代码。不幸的是，一些早期的基于DOS的病毒制造者也使用这种思想。他们常常会用某些地址指示器来替代DOS的I/O矢量，而将系统引向他们自己编写的恶意代码。这样，会影响到进程中的许多系统。许多当今流行的操作系统都使用中断控制的I/O。幸运的是，这些操作系统都有一些保护机制来防止这种类型的矢量操作。

直接存储器存取

不管是采用程序控制的I/O，还是中断控制的I/O，CPU都要从I/O设备移入和移出数据。在I/O过程中，CPU会运行一些类似如下伪代码的指令：

```
WHILE More-input AND NOT Error
        ADD 1 TO Byte count
        IF Byte count > Total bytes to be transferred THEN
          EXIT
        ENDIF
        Place byte in destination buffer
        Raise byte-ready signal
        Initialize timer
        REPEAT
          WAIT
        UNTIL Byte-acknowledged, Timeout, OR Error
ENDWHILE
```

很明显，这些指令非常简单，完全可以使用某个专用芯片来编程。这就是直接存储器存取（direct memory access，DMA）的思想来源。如果一个系统使用DMA，那么CPU就不再需要执行冗长的I/O指令。为了有效地传送数据，CPU必须为DMA控制器提供要传输数据字节的地址、字节数，以及目标设备或存储器地址。这种通信联系通常要利用CPU上的专用I/O寄存器来完成。典型的DMA结构示意图如图7-2所示。

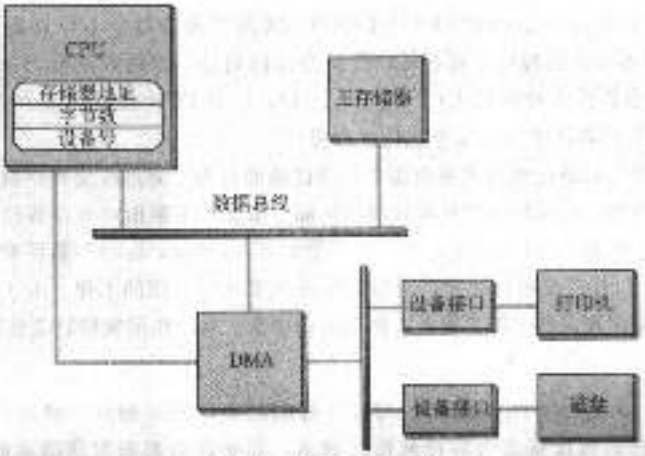


图7-2 DMA配置示例

一旦在存储器中装入了所要求的数值后，CPU 就会发送信号到 DMA 子系统。由 DMA 来负责 I/O的细节过程，而 CPU 会继续执行其下一个任务。在完成 I/O 处理后（或者是由于出错而结束 I/O 操作），DMA 子系统会发送另一次中断请求通知 CPU。

由图 7-2 可见，DMA 控制器和 CPU 共享存储器总线。任一时刻，两者之中只能够有一个设备控制这条总线。也就是说，成为总线主控设备（bus master）。一般来说，I/O 操作要比 CPU 从内存中提取程序指令和数据的优先级高，原因在于许多 I/O 设备的操作都限制在非常紧凑的时间参数内。如果在特定的时间周期内没有检测到有事件发生，这些 I/O 设备就会被强制超时休息（timeout），并中止当前的 I/O 进程。为了避免 I/O 设备发生超时休息的情况，DMA 会利用平时由 CPU 使用的存储器周期来完成 I/O 操作。这种现象称为周期窃取（cycle stealing），幸运的是，I/O 趋向于在总线上产生突发式（bursty）的传输，即成块或成组地发送数据。在这种突发式传输的间隙，CPU 将会被授权访问总线。由于这种总线的访问过程不会持续很长的时间，因此人们不会抱怨系统是在“I/O 过程中运行”。

通道控制的 I/O

在数据传输过程中，程序控制的 I/O 每次传输一个字节。中断控制的 I/O 每次可以按一个字节，或者是小数据块的形式来处理数据，具体的形式取决于参与 I/O 过程的设备类型。通常，速度较慢的设备，例如键盘，传输相同字节数的数据要比磁盘驱动器和打印机产生更多的中断过程。DMA 方法是面向数据块的 I/O 处理方式，它只是在一组字节的传输完成（或失败）后才会中断 CPU。在 DMA 发送 I/O 完成的信号后，CPU 会给出下一个要读取或写入的内存地址。而在传输失败的事件中，CPU 会独自做出适当的响应。因此，DMA 的 I/O 与中断控制的 I/O 相比，只需要很少的 CPU 参与。对于小型单用户计算机系统来说，这种 DMA 方式的管理开销很低；而对于大型多用户的计算机系统（比如大型机）来说，管理开销并不是很经济。大部分大型计算机都采用 I/O 通道（I/O channel）的智能型 DMA 接口。

利用通道控制的 I/O（channel I/O），一个或多个的 I/O 处理器可以控制多条不同的 I/O 路径。这些路径被称为通道路径（channel path）。对于“慢速”设备，如终端设备和打印机，通道路径可以组合在一起（复用，multiplex），允许几个这类的设备仅仅通过一个控制器来进行管理。在 IBM 的大型计算机中，一个多路复用的通道路径被称为多路复用器通道（multiplexor channel），而服务于磁盘控制器和其他“快速”设备的通道称为选择器通道（selector channel）。

I/O 通道由一些被称为 I/O 处理器（I/O processors，IOP）的小 CPU 来控制，这些 CPU 是专门为 I/O 优化设计的，不像 DMA 的控制电路，IOP 具有执行程序的能力，包括执行算术逻辑指令和分支转移指令。图 7-3 描述了一个简化的通道控制的 I/O 的配置。

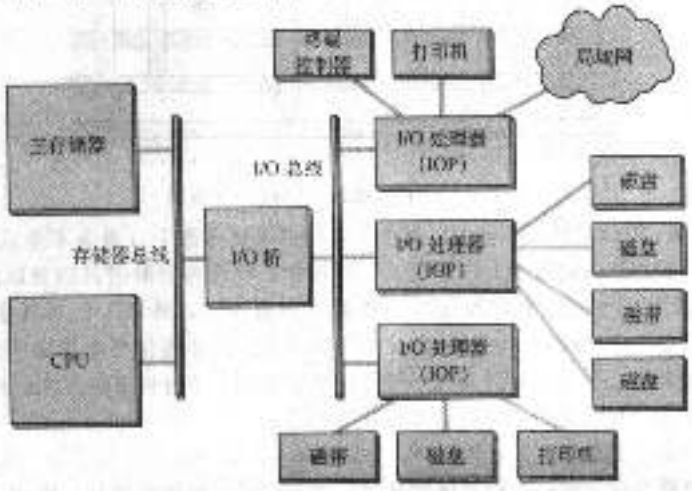


图 7-3 通道控制的 I/O 配置

IOP 执行由主机处理器放置在主系统存储器中的程序。这些程序，由一系列通道命令字（channel command words，CCW）组成，不但包括实际的传输指令，还包括控制 I/O 设备的命令。这些控制命令中包含不同类型设备的初始化，打印机纸张输出和磁带倒回等诸如此类的命令。一旦将 I/O 程序放置到内存后，主机的处理器将发出一个启动子通道（start sub channel）的命令（SSCH），通知 IOP 在那里可以找到程序的内存地址。在 IOP 完成任务后，它会在内存中放置任务已经完成的信息，并且向 CPU 发送一个中断信号。然后，CPU 会得到该完成信息，并针对相应的返回代码做出动作。

独立的 DMA 与通道控制的 I/O 的主要区别在于 I/O 处理器的智能特性。IOP 能够对协议进行协商，发出各种设备命令，将存储译码转换为内存译码，并可以独立于主机的 CPU 来传输多个完整的文件或文件组。而主机只负责为 I/O 操作创建程序指令，以及通知 IOP 在哪里能找到这些程序指令。

像独立 DMA 一样，一个 IOP 必须从 CPU 中窃取存储器周期。但与独立 DMA 所不同的是，通道控制的 I/O 系统都配备单独的总线，用来帮助隔离主机和 I/O 操作，例如，如果从磁盘中拷贝一个文件到磁带，IOP 仅仅会在从主存储器提取指令时，才使用系统存储器总线。余下的数据传输部分只需使用 I/O 总线来完成。由于具有智能性和总线隔离性的优点，通道控制的 I/O 通常在高吞吐量的事务处理环境中使用。在这种情况下使用通道控制的 I/O 的代价和复杂性是合理的。

7.3.2 I/O 总线操作

在第 1 章中，介绍了如图 7-4 图解所示的计算机总线体系结构。该图表达了如下的重要思想：

- 系统总线是一个由计算机系统中多个组件共享的资源。
- 必须控制对该共享资源的访问。这是计算机系统为什么需要一条控制总线的原因。

通过我们前面部分的讨论，很明显存储器总线和 I/O 总线可以是相互独立的实体。事实上，分离这两条总线通常是一个很好的想法，让内存有其自己的总线的一个好的理由是内存的数据传输可以同步（synchronous），这时系统可以使用多个 CPU 的时钟周期从主存储器中检索数据。在一个功能完善的系统中，永远都不会出现内存脱机，或者是由于某些相同类型的错误，比如打印机纸张用完，而困扰外围设备的问题。

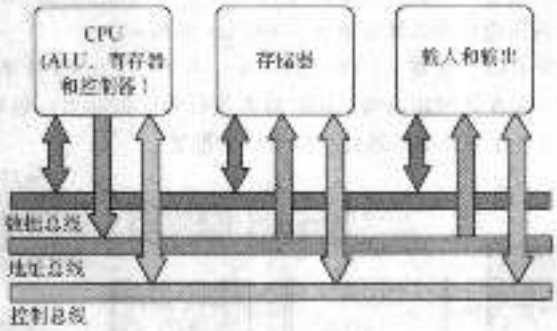


图 7-4 一个系统总线的高层次视图

从另一方面来说，I/O 总线并不能同步操作。它们必须要考虑 I/O 设备不能总是在准备处理 I/O 传输这个事实。I/O 的控制电路置于 I/O 总线上，并且在 I/O 设备之间进行协商以决定每个设备可能使用总线的时刻。因为这些握手事件在每次总线被访问时发生，所以 I/O 总线也称为异步（asynchronous）总线。通常，区分同步传输和异步传输的方法是：同步传输要求传输的发送者和接收者共享同一个公共的定时时钟。但是，异步总线协议也同样要求有一个时钟来作为位定时和描绘信号转变。在我们学习了一个例子之后，这种观点将会变得非常清楚。

重新考虑图 7-2 中的配置。为了清晰阐述，我们在图 7-2 中并没有将数据线、地址线和控制线分开表示。对于 DMA 电路和设备接口电路之间的连接，图 7-5 表示得更加准确，图中显示了单独的组件总线。

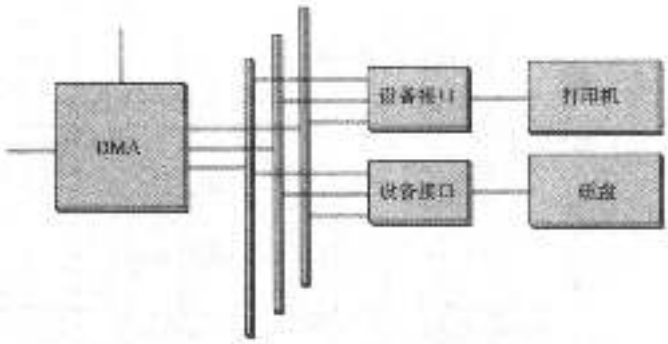


图 7-5 显示地址总线数据总线和控制总线的 DMA 配置

图 7-6 给出了磁盘接口电路连接三种总线的细节。地址总线和数据总线分别由一组单独的导线构成，每根导线可以传输 1 位的信息。数据线的数量决定了总线的宽度（width）。一个数据总线有 8 条数据线，每次可以传送 1 字节的数据。地址总线需要有足够多数目的导线来唯一地识别总线的每一个设备。

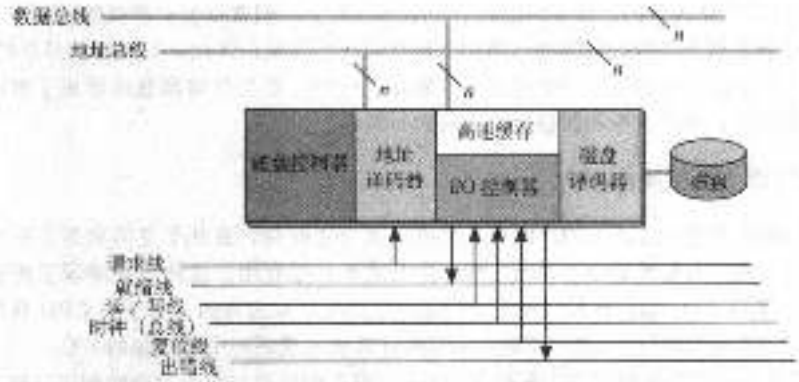


图 7-6 磁盘控制器接口与 I/O 总线的连接

图 7-6 中的一组控制线只画出了为举例说明所需的最少数目的控制线。实际的 I/O 总线通常会有多于 12 条控制线（IBM 公司早期的 PC 机的控制线就超过了 20 条）。控制总线负责协调总线及其连接到总线上的设备。为了将数据写入磁盘驱动器中，例子中的总线会执行如下的操作顺序：

1. DMA 电路将磁盘控制器的地址放置地址线上，并激活（断言有效）Request 和 Write 信号。

2. 在 Request 信号断言确认有效后，控制器中的译码电路会查询地址总线。

3. 按照所查询的地址，译码器启动磁盘控制电路。如果磁盘的写入数据有效，控制器就会在 Ready 线上施加一个信号。这时，就完成了 DMA 和控制器的握手。当 Ready 信号升起后，其他的设备就不可以使用总线。

4. 然后，DMA 电路会将数据放到总线上，并且撤销 Request 信号。

5. 当看到 Request 信号撤销后，磁盘控制器将字节从数据线上传送到磁盘缓冲器中。然后撤销控制磁盘控制器的 Ready 信号。

为了使这一处理过程的图像变得更加清晰和准确，工程师常常会使用时序图（timing diagram）来描述总线操作。磁盘写操作的时序图如图 7-7 所示。图中标注从 t_0 到 t_5 的垂直线表示不同信号的持续时间。在实际的时序图中，一个精确的持续时间将被划分为相等的时间间隔，通常间隔为 50 纳秒。总线上的信号只在时钟周期转换时，才会发生改变。注意，图中所描述的各种信号并不会立即上升和下降。这一点正反应了总线的物理本质。必须要有少量的时间来容许信号电平稳定下来。尽管这个稳定时间（settle time）很小，但是在一个长时间的 I/O 转换过程中也会引起一个大的延迟。

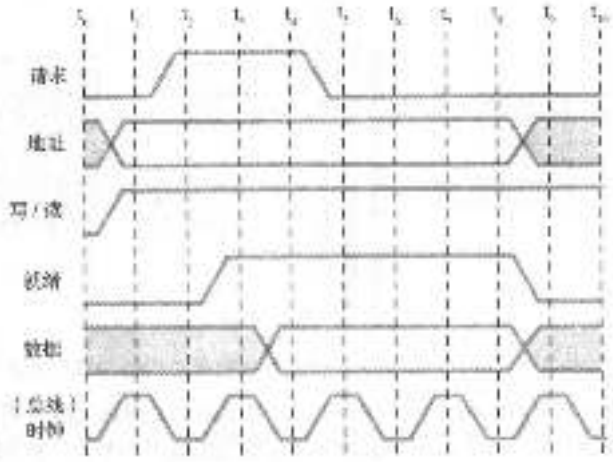


图 7.7 总线时序图

许多实际的 I/O 总线与例中的 I/O 总线并不完全相同。它们常常没有单独的地址和数据总线。由于 I/O 总线具有异步特征，因此数据线也能用来保持设备的地址。因此，只需添加另外的一条控制线，来指示数据线上传输的信号是代表一个地址还是数据。这种方法与存储器总线形成了鲜明的对比。对于存储器总线的情形，地址信号和数据信号必须同时有效。

7.3.3 深入讨论中断控制的 I/O

至此，我们已经假设了总线上的外围设备在收到命令之前会一直处于空闲状态。在小型计算机系统中，这种“仅当别人对你讲话你才说话”的工作方式不是很有用。这种假设暗示了所有的系统活动性都源于 CPU，即由 CPU 引起。但实际上，系统的活动是来自用户的。为了与 CPU 通信，用户不得不设计某种方法来引起 CPU 的注意。因此，小型的计算机系统采用中断控制的 I/O。

图 7-8 描述了一个系统如何实现中断控制的 I/O。现在的情形与前面讨论的例子一样，只不过现在是在外设和 CPU 之间增加了一种通信方式。系统中的每个外设都会连接到一条中断请求线。中断控制芯片对于每条中断线都有一个输入端，只要有中断线的信号被断言确认有效，控制器就会对该中断进行译码，同时激活 CPU 上的中断 Interrupt (INT) 输入信号。当 CPU 已经准备好处理该中断时，CPU 就会发送一个 Interrup Acknowledge (INTA) 信号。一旦得到这个确认信号，中断控制器就可以撤销 INT 信号。

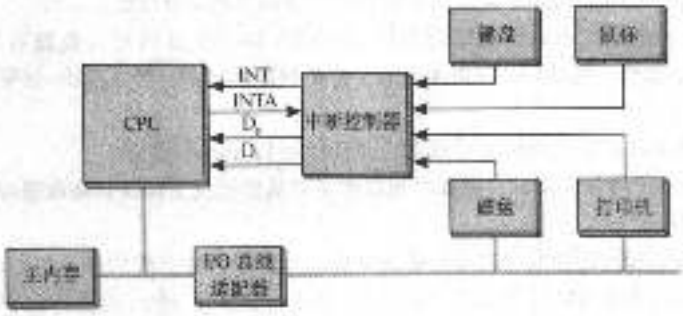


图 7-8 一个使用中断的 I/O 子系统

当然，系统设计者必须决定：当有多个设备同时请求中断时，应该优先处理哪一个设备。通常，我们会将这个设计决定固化在中断控制器中。使用相同操作系统和中断控制器将具有高优先权设备（例如键盘）连接到同一根中断请求线上，每个系统所允许的中断请求线的数目不相同，而且在某些情

况下，这种中断是可以共享的。如果我们非常清楚地知道在同一时刻不会有两个设备使用同一个中断时，共享的中断将不会引发任何问题。例如，一个扫描仪和一个打印机通常可以使用同一个中断而和平共处。但是，对于使用串行接口连接的鼠标和 modem，事情并不总是如此。由于安装程序事先并不知道它们有可能使用同一个中断，所以中断共享对这两个设备都会引发异常行为。 ■

字节、数据和信息……记录

计算机专家并非对计算机外的世界一窍不通。

——Bill Walsh

很多人使用 information 这个词作为 data 的同义词，并且把 data 作为 byte 的同义词。事实上，我们常常在文章中使用 *data* 作为 *byte* 的同义词以增加文章的可读性，希望使得上下文意思更清楚。但是，我们必须指出这些词在意义上的确有区别的。

从字面上来看，data 这个词是复数的。它来源于单数形式的拉丁词 *datum*。后来，由于要提及多个的 datum，所以人们就适当地使用 data 这个词。事实上，当有人说："The recent mortality data *indicate* that people are now living longer than they did a century ago" 这样的话语时，我们会觉得很自然。但是，我们很难解释为什么我们不会说 "A page fault occurs when data *are* swapped from memory to disk"。当用 data 来谈论存储在一个计算机系统中的东西时，我们实际上已经把 data 概念化成一个"不可区分的主体"，就像我们对空气和水的感觉一样。空气和水由分子的各种分立元素组成。相应地，由大量的分离元素组成的大量 data 称为 data。很明显，没有一个受过教育，英语流利的人会说 "She breathes *airs* "，或者说 "She takes a bath in *waters* "。因此，这样看起来下面的说法也是非常合理的，"…data *is* swapped from memory to disk"。大多数的学术资源（包括《美国传统词典》）现在都认同使用在这种方式时，把 data 作为一个单数的集合名词。

严格来说，计算机存储的是介质而不是存储 data。计算机存储的是被称为 byte 位组合模式（位图）。例如，如果使用二进制的扇区编辑器来检查磁盘的内容，就可能看到 010000100 模式。所以从结果可以得到什么启示呢？大家都知道，这种位图可能是程序的二进制代码、操作系统结构的部分内容、一张照片，或是某人的银行账户的余额。如果你知道这些位代表的是一些数字量（而不是程序代码或图像文件），并且还知道它是采用二进制补码形式来存储的。那么你就可以非常肯定地说，它代表的是十进制数 68。但是，仍然没有得到一个 datum。在得到 datum 之前，人们必须要把某些上下文与这个数字联系起来。例如，它是一个人的年龄或体重？是一个罐头开启工具的模型编号？如果知道 01000100 是来自某个自动气象站的一个包含温度输出的文件的话，你就有了自己的 datum。磁盘上的文件便可以正确地称为 data 文件。

现在，你可能会猜测天气数据使用华氏温度来表示，因为地球上没有地方的温度曾经达到 68℃。但是你还是没有得到 information。当前情况下，这个 datum 是没有意义的：它是荷兰的 Amsterdam 此刻的温度？还是三年前的 2：00AM 在美国的 Miami 记录的温度？只有当它对人们有意义时，datum 68 才变成 information。

最近普遍作为单数使用的另一个复数拉丁名词是 media。以前，受过良好教育的人，只是在想要提及一个以上的 medium 时才使用 media 这个词。报纸是一种类型的通信 medium。而电视是另外的一种 medium。整体来说，它们都是 media。但是现在的一些编辑接受这样的单数用法，"At this moment, the news media *is* gathering at the Capitol"。

因为艺术家可以使用水彩画 medium 或者油画 medium 来作画，计算机的数据记录设备可以将信息写到某个电子 medium 中，例如磁带或磁盘。整体来说，它们都是电子 media。但是我们却很少看到一个计算机的从业者会有意识地正确使用这个术语。更多的情况下，我们遇到的是类似于下面的这类陈述，"Volume 2 ejected. Please place new *media* into the tape drive"。在这里，大多数人是否能够理解下面的这个指令还是值得争论的事情，"…place a new *medium* into the tape drive"。

当计算机专家力图采用数字形式来表达人类思想时（反之亦然），他们会遇到这样的语义争论。在

该类型的转换过程中肯定会丢失信息，并且我们要学会去接受这样的事情。但是，有些事情可能会超出我们愿意接受的极限。这些限度有时被称为人的“本性”。■

7.4 磁盘技术

在磁盘驱动器技术出现之前，顺序存储介质，如打孔卡片、磁带和纸带，是唯一可用的持久性的存储介质。如果某个用户所需的数据写在磁带卷轴的尾部，则需要读取整卷才能读取，并且每次都只能阅读一个记录。这缓慢的读取设备和小的系统内存常常使数据的读取过程极其缓慢。纸带、磁带和卡片不仅处理速度很慢，而且由于它们所暴露的物理环境的影响，其性能也会很快衰减。纸带使用时总是缠绕容易断裂。开放的卷轴磁带不仅会缠绕，而且容易被操作人员误处理。卡片可能会被撕碎、丢失和弯曲折坏。

在存储技术的发展过程中，读者不难看出，1955 年 IBM 公司所发布的第一台商用磁盘计算机是如何彻底地改变了计算机的世界。这台机器被称为使用随机访问方法的会计和控制计算机，或者简称 RAMAC（Random Access Method of Accounting and Control Computer）。按照今天的标准，早期计算机使用的磁盘体积巨大并且速度缓慢，是我们所难以理解的。当时，每个磁盘的直径尺寸为 24 英寸，但是磁盘的每一面却仅能容纳 50 000 个 7-位的字符。可以想像，50 个双面的磁盘安装在一根转轴上，并用一个闪耀玻璃罩包围起来，大小与一个花园的杂屋差不多。每个转轴上磁盘的总存储容量约为 500 万字符，每次访问磁盘上的数据，平均需要经过一秒的时间。这种磁盘驱动器的总重量超过 1 吨，并且制造成本需要几百万美元。当时，没有人可能会从 IBM 购买这些设备。

相比之下，到了 2000 年初，IBM 公司开始推销一种用于掌上电脑和数码相机的高容量磁盘驱动器。这些磁盘的直径尺寸仅为 1 英寸，并且能保存 1GB 字节的数据，而数据的平均访问时间只有 15 毫秒。这种磁盘驱动器的重量不足 1 盎司，而零售价格低于 300 美元。

磁盘驱动器被称为随机（random）存储设备，有时也称为直接（direct）存储设备。磁盘上的每个存储单元，称为扇区（sector），都有一个独一无二的存取地址，可以与周围的扇区区别。如图 7-8 所示，这些扇区按照同心圆环的形式划分成一圈一圈的磁道（track）。对于大多数的磁盘系统，每一个磁道包含的扇区数相同，而且每一个扇区都包含相同数目的字节。因此不难看出，数据在磁盘中心部分写入的数据比外部边缘部分要更加密集。某些制造厂商会将全部扇区都制作成近似相同的容量，而在外部的磁道上放置比内部磁道更多的扇区，这样可以在磁盘上存储更多的信息。这种方式称为分区位（zoned-bit）记录法。现在很少使用这种记录法，因为与传统磁盘系统相比，它需要更加复杂的驱动控制电路。

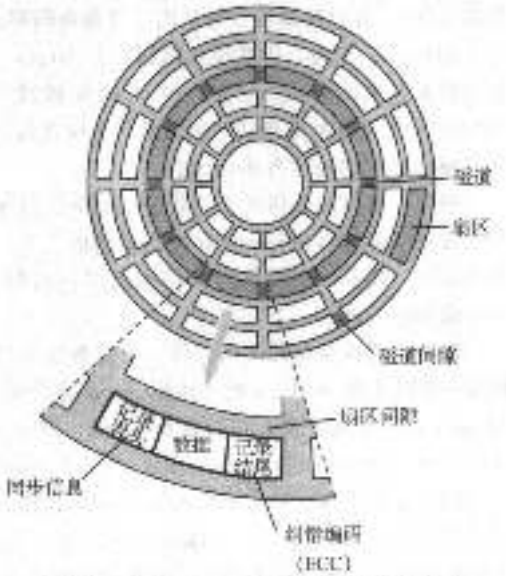


图 7-9 磁盘扇区，其中画出了扇区间隙和逻辑扇区的格式

磁道是从磁盘最外边的磁道 0 开始连续编号，然而，在一条磁道分布的圆周上，扇区的分配顺序可能是不连续的。它们有时“跳来跳去（skip around）”，这样允许驱动器电路有时间在读取下一个扇区之前处理完扇区进程的内容。这种方法称为交叉存储技术。根据磁盘的旋转速度和磁盘电路速度以及缓冲区的大小不同，这种交叉存储技术会有所变化。当今大多数的硬盘驱动器读取磁盘采用一次读取一条磁道的方式，而不是一次读取一个扇区的方式，现在所有交叉存储技术已经不是很流行了。

7.4.1 硬盘驱动器

硬盘（hard 或 fixed）包括控制电路和一个或多个金属或玻璃盘片。这些称为碟片（platter）的盘片上镀有一层薄的磁性材料薄膜。磁盘碟片堆叠在转轴上，通过传动箱中的一个马达来带动磁盘碟片旋转。磁盘的旋转速度可以达到每分钟15 000转（rpm），典型的磁盘转速为5400 rpm和7200 rpm。磁盘的读/写头通常安装在一个旋转的硬盘驱动臂（actuator arm）上，磁盘驱动臂通过其转轴上缠绕的线圈的感应磁场来进行准确的定位（参见图7.10）。如果给驱动臂提供传动能量，则整个梳状的读写头就可以向磁盘的中心移动，或者是从磁盘的中心移开。

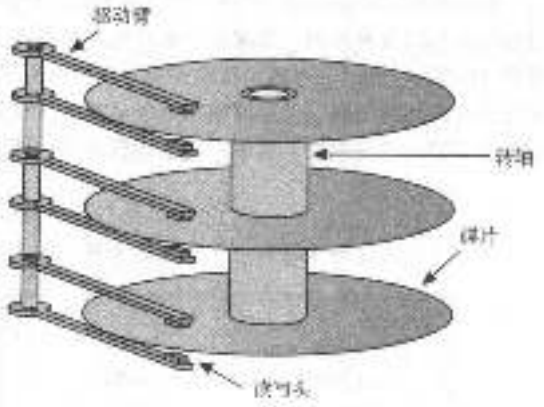


图7-10 硬盘驱动臂（连同读/写头）和磁盘

尽管磁盘技术得到了持续的发展，但是我们现在还不能大批量生产完全不出错的介质。尽管出错的概率非常小，但是出错总是难免的。目前，我们采用两个机制来减少磁盘表面可能出现的错误：对数据本身进行特殊的编码和使用纠错算法。这种特殊的编码方式和某些纠错的校验编码方法在第2章中已经讨论。这些工作将由磁盘控制器硬件中的特殊电路来处理。而磁盘控制器中的其他电路负责磁头的定位和磁盘的定时同步任务。

对于一个堆叠的磁盘结构，磁盘上的各个磁道上下一一对应，形成一个圆柱面。一组梳状的读写头每次可以访问一个柱面。柱面描述的是每个磁盘上的环状区域。

通常，磁盘的可用面上都有一个磁头。而对于一些老式的磁盘系统，特别是某些移动磁盘，最上层碟片的上表面和最下层碟片的下表面常常都是不用的。硬盘磁头从来都不会触及磁盘的表面，而是悬浮在磁盘表面的上方。它们之间仅仅相隔有几个微米厚的空气层。当磁盘系统断电后，磁头退到一个安全的地方。这一过程称之为停靠磁头（parking the heads）。如果读写头接触到磁盘的表面，将损坏磁盘。这种情况称为磁头撞击（head crash）。

磁头撞击是早期的磁盘驱动器很常见的现象。在第一代磁盘系统中，驱动器的机械部件和电子组件的价格与磁盘碟片相比，都是非常昂贵的。为了用最少的资金提供最多的存储器系统，计算机制造商使用了一种称为磁盘组（disk pack）的可移动磁盘来制造硬盘驱动器。然而，如果打开驱动器密封外壳，各种空气污染杂质物，如灰尘和水汽，都会进入到磁盘系统中。结果，在磁头到磁盘之间需要保持一个大的间隙来防止杂质导致磁头的撞损。尽管在磁头到磁盘之间留有较大的间隙，但是磁头撞击事件仍然会持续发生，以至于某些公司由此所经历的停工时间几乎与生产时间一样长。当然，加大磁头到磁盘之间的空隙，实际上是降低了数据的存储密度。磁头到磁盘的距离越大，需要更强的磁荷来感应读取数据所需的磁盘表面涂敷层中的磁通量变化。要得到更强的磁荷，就需要有更多的磁介质粒子（会占据更大的磁表面）来参与磁通量的转变过程，这样就会降低磁盘驱动器上的数据密度。

后来，由于控制器电路和机械组件的成本大幅度下降，使得密封式磁盘系统得到了广泛的应用。IBM公司最先发明了这种技术，当时这种技术开发使用的是“温彻斯特（Winchester）”编码，因此，温彻斯特很快变成了密封式磁盘的代名词。今天，没有人再生产可移动磁盘组的驱动器，所以我们也没有必要对磁盘进行细致的区分，密封式驱动器允许磁头到磁盘的间隙靠得更近，因而增加了数据的存储密度以及提高磁盘的旋转速度。这些要素构成了硬盘驱动器的性能特征。

寻道时间（seek time）是指磁盘驱动臂定位到指定的磁道上所需的时间。寻道时间并不包括磁头读取磁盘目录（disk directory）的时间。磁盘目录将逻辑文件信息，例如，my_ story.doc，映射到

对应的物理扇区地址，例如第7柱面中第3表面的第72扇区（简写为柱面7，表面3，扇面72）。一些高性能磁盘驱动器，会在磁盘的每个可用面的每个磁道上都提供一个读写头，这样实际上可以消除寻道时间。由于这种系统中没有可移动的驱动臂，所以访问数据时唯一可能延迟的是磁盘的旋转延迟。

旋转延迟（rotational delay）是读写头定位到指定的扇区上方所需要的时间。旋转延迟时间与寻道时间的和称为存取时间。如果将存取时间与从磁盘上实际读取数据所需要的时间相加，就得到了传输时间（tranfer time）。当然，传输时间的多少取决于所要读取的数据量的多少。反应时间（latency）是旋转速度的直接函数。反应时间用来衡量在驱动臂定位到目标磁道后，目标扇区移动到读/写头的下方位置所需要的时间总量。通常引用平均数表示，计算公式如下：

$$\left(\frac{60\text{秒}}{\text{磁盘旋转速度}}\times\frac{1000\text{ms}}{\text{秒}}\right)/2$$

为了对上面这些术语有一个全面的了解，图7-11给出了一个典型的磁盘的说明书。

系统配置：	
格式化容量，MB	1340
驱动控制器	SCSI
编码方式	RLL 1，7
缓冲区大小	64K
磁盘数目	3
数据使用面	5
每面的磁道数目	3 196
磁道密度	5 080 tpi
记录密度	92.2Kbpi
每块字节数目	512
每道扇区数目	132
物理性能：	
高	12.5mm
长	100mm
宽	70mm
重量	170g
温度范围（℃）	
工作时	5℃～55℃
贮存时	40℃～71℃
相对湿度	5%～95%
噪音	33dBA，空闲状态
可靠性和维护：	
MTTF	300 000小时
开启／停止 循环次数	50 000
设计寿命	5年（最少）
错误率	
（不可恢复）	<1每读10^{12}位
基本性能：	
寻道时间	
道-道	4.5 ms
平均	14 ms
平均反应时间	6.72 ms
转速	
（+/-0.20%）	4 464 rpm
控制器开销	<200 μ秒
数据传输率：	
往／返 介质	6.0 MB/秒
往／返 主机	11.1 MB/秒
启动时间	
（从静止）	5秒

电源要求

工作方式	+5VDC +5%～10%	功率 +5.0VDC
旋转	1000mA	5000mW
空闲	190mA	950mW
待命	50mA	250mW
睡眠	6mA	30mW

图7-11 磁盘驱动器制造商提供的一个典型的硬盘说明书

因为在每次执行读写操作之前都必须先读取磁盘目录，所以磁盘目录的存放位置对磁盘驱动器的整体性能有重大的影响。由于磁盘最外层的磁道在相同的面积内具有最低的位密度，因此与内层的磁道相比，这些区域最不容易出现位错误。为了保证最高的可靠性，磁盘目录可以存放在最外层的磁道，也就是磁道0。这就意味着，每次存取操作时，驱动臂都必须向外摆动到磁道0，然后再回到所需数据

的磁道上。因此，存取驱动臂大跨度的移动可能在很大程度上影响磁盘系统的性能。

随着记录技术和纠错算法的发展，现在已经允许将磁盘目录放到能够提供最佳性能的位置上：即放在最中间的磁道上。这种方法可以充分地减少驱动臂移动距离，提供最大可能的吞吐量。因此。有一些（但不是全部）现代磁盘系统采用在中央磁道存放磁盘目录。

目录存放是磁盘逻辑组织的一个要素。磁盘的逻辑组织与磁盘的操作系统密切相关。磁盘逻辑组织的一个主要方面是扇区和逻辑地址之间的映射关系。由于硬盘包括很多的扇区，不可能为每个扇区都保存一个标记。下面，我们考虑在图 7-11 的数据表中所描述的磁盘。磁盘系统有 5 个表面，每个表面有 3196 条磁道。而每个磁道包含有 132 个扇区。这就意味着，在磁盘系统中共有 2 109 360 个扇区。如果用一个分配表来列出每个扇区的状态，用 1 字节来记录每个状态，那么将会因此要消耗 2MB 字节的磁盘空间。这种做法不仅要花费大量的磁盘空间开销，而且在需要检查某个扇区的状态时，读取数据结构无疑也会或多或少地消耗一定的时间量。显然，检查扇区的状态是一个需要频繁执行的任务。正是因为这个原因，操作系统以群组的形式为扇区分配地址，称为区块（block，或 cluster）。采用这种地址分配方式，可以使文件管理变得更加简单。每个块中扇区的数目决定了分配表的大小。如果分配的块越小，则当一个文件不能装满整个块时，被浪费的磁盘空间也就越少。但是，如果磁盘块分配得太小，记录块的分配表就会变得很大，而且查找的速度也会变得很慢。下一节介绍软盘系统时，我们将会深入讨论目录和文件分配结构之间的相互关系。

图 7-11 中有关磁盘规格说明的最后要讨论的一点是：标题“可靠性和维护”下面的有关对磁盘可靠性的各种估计。根据制造厂商的说法，这种特殊设计的磁盘驱动器使用期限为 5 年，并且可以允许系统启停 50 000 次。同样，在这个标题下，系统的平均失效时间（Mean Time to Failure，MTTF）是 300 000 小时。可以肯定，该数字并不能表示这种磁盘驱动器的寿命的期望值是 300 000 小时。如果磁盘持续运行，该数字表示将会超过 34 年。

这个规格说明书很清楚地表明了这种磁盘驱动器的设计寿命仅可以持续 5 年。这种明显的差异是由于采用了制造业界通行的统计质量管理方法所造成的。除非这种磁盘是根据政府的合同制造的，否则用来计算 MTTF 的确切方法完全是由制造厂商来决定的。通常，这个计算过程包括：从生产线上随机取样，并在一些不太理想的条件环境中运行磁盘系统某个定量的时间，通常大于 100 个小时。然后将失效数目描绘成概率曲线，最后得到 MTTF 的数字。简言之，规格说明中有关“设计寿命”的数据更让人可信和容易理解。

7.4.2 软盘

软盘（Floppy Disk）的组织方式与硬盘非常相似，也是按照磁道和扇区来寻址的。它们通常之所以被称为软盘，是因为磁盘的磁性材料涂层是附着在一个软性的聚脂塑料（Mylar）基片上。事实上，由于软盘不能采用像硬盘同样的方式密封起来，所以软盘的数据密度和旋转速度（一般为 300 或 360 RPM）受到了很大的限制。另外，软盘的读写头还必须接触到磁盘的磁表面。当读写头上有帖粒时，读写头的磨擦会引起磁性涂层的磨损。因此，必须定期清洗磁头，以清除可能导致软盘磨损的微粒。

如果仔细检查 3.5"软盘，可以看到软盘中心的金属轴上有一个矩形孔。软盘驱动器的机电装置就是利用这个孔来确定第一个扇区的位置，第一个扇区位于软盘的最外部的边缘。

与硬盘相比，软盘的组织结构和操作规范更加统一。例如，考虑 3.5" 1.44MB DOS/Windows 软盘。软盘的每个扇区包括 512 个数据字节。每条磁道上有 18 个扇区，且软盘的每一面有 80 条磁道。扇区 0 是软盘的引导扇区（boot sector）。如果这个软盘是引导盘（也称为启动盘），那么这个引导扇区就包含能让系统从软盘而不是硬盘启动的信息。

紧接着引导扇区的是两个完全相同的文件分配表（File Allocation Table，FAT）的副本。标准的 1.44MB 软盘的每个 FAT 的大小是 9 个扇区长。在 1.44MB 软盘中，一个区块（cluster），即一个可编址的单元，是由一个扇区组成的。因此，对于磁盘上的每一个数据扇区，在 FAT 上都有一个条目。

磁盘的根目录占用了从扇区19开始的14个扇区。每个根目录的条目占用32字节，每个根目录存储文件名、文件属性（存档、隐藏、系统，等等）、文件的时间表、文件大小和文件存放的起始区块（扇区）的编号。起始的区块编号指向FAT中的某个条目。并且，如果某个数据文件占用多个区块的话，它还允许我们去跟踪这个数据文件所跨越的扇区链。

FAT是一个简单的表结构，它采用位图形式来记录磁盘上每一个区块的基本信息，指示区块是否处于空闲、保留、数据占用、或者坏区的状态。因为一个1.44MB磁盘包含18×80×2=2880个扇区，所以每个FAT条目需要12位，仅用来指出一个区块。实际上，软盘上的每一个FAT条目都是16位宽的。所以这种组织方法称为FAT16。如果一个磁盘文件跨越多个区块，那么该文件的第一个FAT条目中同样包含一个转向该文件下一个FAT条目的指针。如果一个FAT条目是文件的最后一个扇区，则“下一个FAT条目”的指针中包括文件结束的标记。FAT的链表结构允许文件存储于磁盘中的任意空闲扇区，而不管这些扇区的位置是否连续。

为了理解清楚这种概念，下面我们考虑图7.12所给出的FAT条目。如上所述，磁盘上的每个区块在FAT中都包含一个条目。假设文件占用了从扇区121开始的4个扇区。当读取该文件时，会发生如下操作过程。

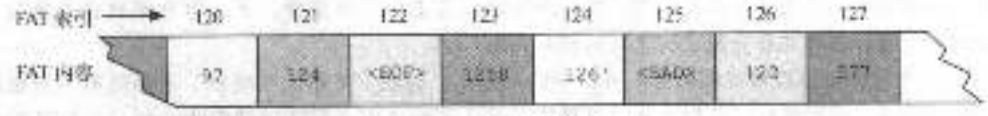


图7.12 一个文件的分配表

1. 读取磁盘目录查找文件起始区块（121）。读取第一个区块去取回文件的第一部分。

2. 读取位于区块121的FAT条目查找文件的剩余部分，这里给出了文件的下一个数据区块和相应的FAT条目（124）。

3. 读取区块124和相应的FAT条目，FAT条目指出下一个数据在扇区126。

4. 读取数据扇区126和相应的FAT条目，FAT条目指出下一个数据在扇区在122。

5. 读取数据扇区122和相应的FAT条目，直到在下一个扇区位置找到<EOF>标记，这时系统知道已经得到了文件的最后一个扇区。

我们很容易就可以发现在FAT磁盘的组织结构中还有很多性能改进的空间。这就是为什么FAT不被用在高性能的、大规模的系统上的原因。但是，FAT对软盘仍然很有用，这其中有两个原因。第一，对于软盘来说，性能不是考虑的重要因素。第二，软盘有标准的容量，而不像硬盘那样容量几乎每天都在增加。因此，简单的FAT结构不可能产生当磁盘容量开始超过32MB时FAT16结构通常所遇到的问题。如果使用16位的区块指针，则一个33MB磁盘的一个区块的大小至少必须有1KB。当磁盘驱动器的容量继续增加时，FAT16的扇区会变得更大。这时，如果一些小文件不能占满整个区块时，会浪费很多的磁盘空间。面对于大于2GB的磁盘系统，则要求的区块大小达到64KB。

不同的厂商使用了许多自己的技术方案来提高软盘上的数据密度。这些技术中最流行的是Zip驱动器，它是由Iomega公司首先开发出来的。这种技术使用的是磁-光设计，它结合了磁盘存储器的可重复读写和激光技术所提供的读/写头精确定位的优点。然而，为了获得高容量的长期数据存储器，随着价格便宜的光学存储方法的出现，软盘系统很快就变得过时了。

7.5 光盘

现在，光学存储器系统实际上可以提供无限制的存储容量，而价格上又可以与磁带竞争。光盘具有多种格式，最流行的是大家普遍使用的CD-ROM（compact disk-read only memory）。CD-ROM能够保存超过0.5GB的数据。CD-ROM是一种只读存储介质，用来发布软件和数据非常理想。CD-R（CD-recordable）、CD-RW（CD-rewritable）和WORM（write once read many）光盘也都是光学存储设备。

通常用作长期数据的存档和大量数据的输出。CD-R 和 WORM 光盘为文档和数据提供了大容量的防篡改存储器。为了数据的长期归档存储，一些计算机系统会直接将数据输出到光学存储器，而不是打印纸或微缩胶片上。这种光盘称为计算机输出激光盘（Computer Output Laser Disk，COLD）。带有机械手的自动存储库系统又称为自动光盘柜（optical jukeboxes），可以直接访问大量的光盘。自动光盘柜机可以存储大量的数据达到几百张光盘，总容量可达 50GB 到 1200GB，或者更多。光盘存储器的支持者声称光学存储器，不同于磁介质，能够储存 100 年而不会有明显的性能衰退。现在，又有其他什么存储器可能来挑战这种说法呢？

7.5.1 CD-ROM

CD-ROM 是一种利用聚碳酸酯材料（塑料）制造的光盘。光盘的直径大小为 120 毫米（4.8 英寸），并且涂敷一层铝反射膜。铝膜的上面使用一层丙烯酸材料的保护性密封涂层，以防止铝膜的磨损和腐蚀。光盘上的铝层会反射来自光盘下面的绿色激光二极管所发射的光束。反射光束通过一个棱镜，将光线偏转到一个光电检测器上。光电检测器就会把光学脉冲转换成电信号，然后发送给驱动器的电学译码装置，如图 7-13 所示。

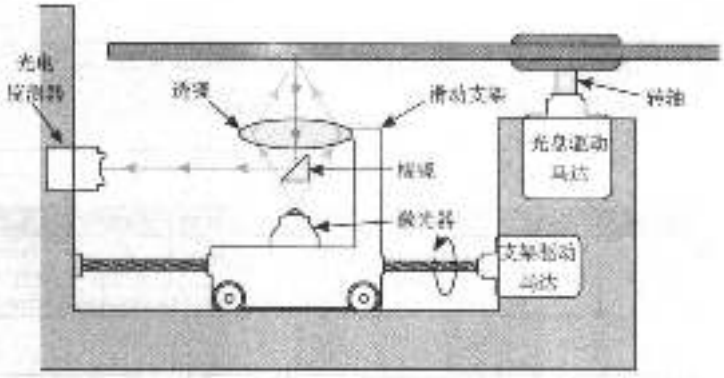


图 7-13 CD-ROM 驱动器的内部结构

光盘的数据写入使用的是从中心到外部边缘的方式，在聚碳酸酯基片上形成一条凹凸不平的单一螺旋状的轨道（光轨）。凹陷的区域称为凹坑（pit），因为从 CD 的上表面来看他们的确很像凹坑。在凹坑之间的平直空间称为平台（land）。凹坑的尺寸为 0.5 微米宽，0.83～3.56 微米长。凹坑的边缘位置对应于二进制数 1。从凹坑底部到凸起平台的高度为绿色激光二极管所产生激光波长的四分之一。这意味着由于光的干涉效应会导致从凹坑反射出来的激光束与来自激光器的入射光严格相消。这就导致了光脉冲的亮和暗，而被驱动器电路解释为二进制数字。

相邻圈圈的螺旋光轨的距离，称为轨道间距（track pitch），距离必须大于 1.6 微米（参见图 7-14）。

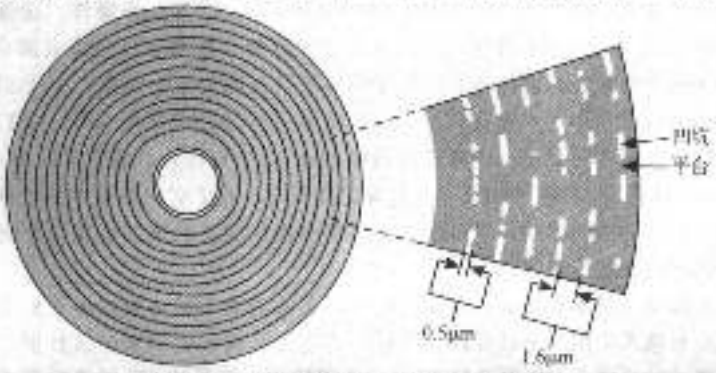


图 7-14 CD 的螺旋光轨和轨道的放大图

如果拆开一个 CD-ROM 或者音乐 CD，并把数据光轨平铺在地面上，那么由凹坑和平台组成的数据线将可以延伸到几乎 5 英里（8 千米）长。但是，它们的宽度却只有 0.5 微米宽，还不到人的头发丝粗细的一半，用肉眼几乎看不到。

尽管一个 CD 仅有一条光轨，但是在大多数据光盘文献中都将由凹坑和平台缠旋转 360°构成的一个圆圈视为一个光轨。与磁盘存储器不同的是，光盘中心的轨道和光盘外圈的轨道都有着相同的位密度。

CD-ROM 最初设计用来存储音乐和其他顺序音频信号的。它在数据存储方面的应用是人们后来的想法。图 7-15 给出了光盘的数据扇区的格式。可以看到，光盘上的数据存储在扇区的信息块中，信息块由扇区的 2352 字节组成，这些扇区沿着规道长度的方向排列。扇区由 98 个 588 位的基本单元组成，这些基本单元被称为信道帧（channel frame）。信道帧由同步信息、文件头和有效载荷的 33 个 17 位的符号组成，如图 7-16 所示。其中，17 位符号采用的是一个 RLL(2,10)编码方式，称为 EFM（8 到 14 的调制编码）。光盘驱动器读取和解释（称为解调，demodulate）信道帧，并创建另外一种被称为小帧（small frame）的数据结构。一个小帧有 33 字节宽。其中，用户数据占用 32 字节，剩下的一个字节用来作为子信道（subchannel）信息。这里，有 8 个子信道，分别命名为 P、Q、R、S、T、U、V 和 W。除了子信道 P 和 Q 外，所有的子信道只对音频应用有意义。子信道 P 表示开始到间和停止时间，而子信道 Q 包含的是控制信息。

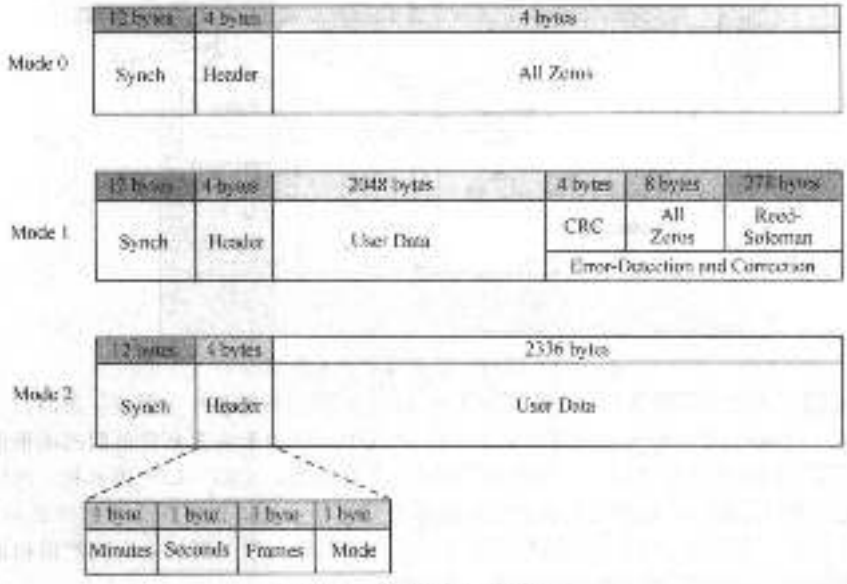


图 7-15 CD 数据扇区的格式

大多数光盘以一个恒定的线速度（constant linear velocity，CLV）来操作。这就是说，激光经过各个扇区的速率始终保持恒定，而不管这些扇区是位于光盘的开始部分还是结束部分。因此，为了保持恒定的线速度，访问外层光轨时要比访问内层光轨时加快光盘的旋转速度。一个扇区是按照激光束沿着光轨从光盘开始处（光盘中央）运行到该扇区所需要时间的分秒数来编号。我们还要对这些“分秒数”进行定标，这里的基本假设是 CD 播放器每秒处理 75 个扇区。计算机 CD-ROM 驱动器的速度已经快达到 Audio CD 速度的 44 倍（44X），今后肯定会更快。为了定位到某个指定的扇区上，激光头的滑动支架会垂直于光轨的方向移动，尽可能准确地猜测指定扇区的可能位置。在随意读取某个扇区后，激光头沿着光轨到达指定的扇区。

光盘中的扇区可以有三种不同格式，具体是哪种格式要取决于记录数据所用的模式。这里有三种不同的模式。模式 0 和模式 2 用来记录音乐，没有纠错能力。模式 1 用来记录数据，有两级错误检测和校正功能。这些格式的具体分布如图 7-15 所示。使用模式 1 记录的 CD 的总容量为 650MB。模式 0

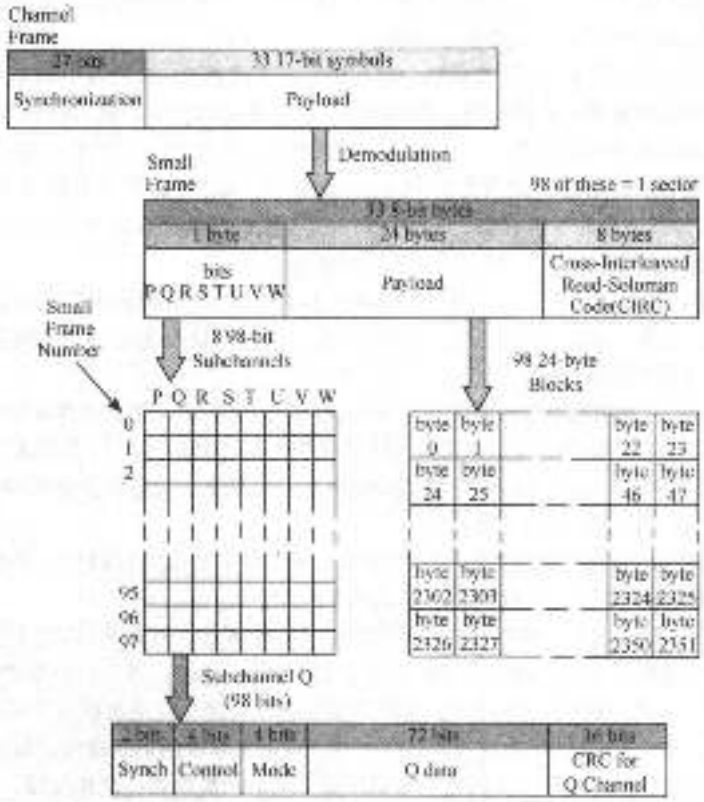


图 7-16 CD 的物理格式和逻辑格式

和模式 2 则能够保存 742MB 的信息，但不能可靠地用来记录数据。

如果使用多时间段（session）的记录方式，CD 中的轨道间距可以大于 1.6 微米。Audio CD 在各个段中都可以记录不同的歌曲。如果从下面看，这些时间段是一些不断扩大的同中心环。当 CD 开始用做数据存储时，这种音乐中"记录时间"的思想没有做任何的修改，就直接扩展到包括数据记录的时间段。CD 有多达 99 个时间段。对于 CD 的各个时间段的界定为：由一个段内数据内容表的 4500 个扇区（时间为 1 分钟）引入（lead in）和结束部分的一个 6750 扇区或者 2250 扇区引出（lead out，或者 runout）。光盘上的第一个时间段中有一个 6750 扇区引出部分，而后面的其他时间段的引出部分则比较短。在 CD-ROM 上，引出部分用来存放属于段内数据的目录信息。

7.5.2 DVD

多功能数码光盘（digital versatile disk），或者简称 DVD（以前称为数字视频光盘，digital video disk），被认为是 CD 密度的 4 倍。DVD 的旋转速度大约是 CD 速度的 3 倍。DVD 凹坑的尺寸大约是 CD 凹坑的一半（0.4 到 2.13 微米），而光轨间距为 0.74 微米。像 CD 一样，DVD 分为可刻录和不可刻录。与 CD 不同的是，DVD 有单面或双面，单层或双层多种形式。单层单面的 DVD 能存储 4.78GB 的数据，而双层双面的 DVD 可以容纳 17GB 的数据。DVD 的扇区大小为 2048 字节。与 CD 一样支持三种数据模式。由于具有更高的数据密度和更快的存取速度，因此大家都认为 DVD 会最终会替换 CD 成为长期数据存储器和信息发布介质。

7.5.3 光盘记录方法

CD 和 DVD 有不同的记录技术。最经济，也是最普遍的方法是使用热敏染料。这种染料夹在聚碳

酸酯基片和 CD 反射涂层之间。当被激光发出的光照射撞击时，这种染料就会在聚碳酸酯基片上产生一个凹坑。这种凹坑会影响 CD 反射层的光学特性。

可重写光学介质，如 CD-RW，则是采用一种金属合金来代替 CD-R 光盘上的染料和反射涂层。这种金属合金中包括不稳定元素，如铟、碲、锑和银等。在正常的状态下，这种合金涂层可以反射激光。但是，如果利用激光将这种合金加热到 500℃时，合金就会发生某种分子变化，并且合金的反射率会降低。化学家和物理学家们把这种过程称为相变（phase change）。如果只是将这种合金涂层加热到 200℃，那么合金会回到原来的反射状态。这样一来，光盘上的数据可以任意改变多次。但是，专家提醒大家，利用相变原理的 CD 记录“只”能工作 1000 次左右。

WORM 驱动器，通常使用在一些大型计算机系统上。WORM 系统采用了比个人所使用的光盘系统更高能量的激光。这里，低能量的激光用来读取数据，而高能量激光则用于不同的、也是更持久的记录方法。其中的三种方法如下：

- 烧录（Ablative）：利用高能量的激光在位于光盘保护层之间的反射金属涂层里熔化一个凹坑。
- 双金属合金（Bimetallic Alloy）：在光盘表面的保护涂层之间封入两个金属涂层。利用激光将这两种金属熔合在一起，引起下面的金属层的反射率发生改变。双金属合金的 WORM 光盘的制造商声称这种介质可以完好保持 100 年。
- 成泡（Bubble-Forming）：在两个塑料层之间压入一个单个热敏感材料层。当被高能量激光照射撞击时，热敏性材料中将产生气泡，引起反射率的改变。

尽管 CD-ROM、CD-R 和 CD-RW 光盘可以使用相同的帧格式，但在某些 CD-ROM 驱动器上可能无法读取数据。产生这种不兼容性的原因可能是由于 CD-ROM 光盘上的信息总是要被记录（或压入）到某个单段内。另一方面，如果 CD-R 和 CD-RW 光盘能够像软盘一样按增量的方式写入信息，那么它们会变得非常有用。第一个 CD-ROM 规范，ISO 9660，规定了单段的记录格式，而且光盘的总段数不多于 99 个。人们很快认识到 ISO 9660 的这些限制阻碍了光盘产品的更广泛地应用，于是，一些 CD-R 和 CD-RW 主导制造厂商就成立了一个协会来协商解决这个问题。经过他们努力，最后开发了一种通用光盘格式规范（Universal Disk Format Specification，UDF），这种光盘记录格式不限制每个光盘上的记录段的数量。这种新格式的核心思想是，用一个浮动不固定的内容表来替换原来与每个记录段相关联的内容表。这种浮动的内容表称为虚拟分配表（virtual allocation table，VAT），它会被写到光盘上紧跟在用户数据的最后一个扇区后面的引出部分。当有数据需要添加到前面某个已有内容的记录段上时，我们可以在新数据写入后，对 VAT 进行重写。这样的增量写入过程可以持续下去，直到 VAT 到达光盘的最后一个可用扇区。

7.6 磁带

磁带是一种最古老和最经济的大容量存储设备。第一代的数据记录磁带使用类似于录音机磁带的相同材料制造而成。这种磁带是在半寸宽（1.25 厘米）的醋酸纤维素薄膜带的表面上涂敷一层磁性氧化物，并将 1200 英尺长的材料缠绕在一根卷轴上，然后手工连接到磁带驱动器。这种磁带驱动器的大小与一个小冰箱相当。早期磁带的容量一般低于 11MB，而且需要将近半个小时才能读或者写完整个卷轴。

磁带上的数据记录方式是每次写入一个字节，对于数据的每一位都要创建一个磁道。另外增加一个磁道来检验数据的奇偶性，这样使得磁带共有九条磁道宽，如图 7-17 所示。这种九磁道的磁带采用的是相位调制编码方式和奇偶校验。奇偶校验采用的奇校验，以保定在一长串零（0s）的传输过程中至少发生一次磁通量的“反转”变化，这是数据库的记录方式的特征。

在过去的时间里，磁带技术的更新是非常显著的。许多制造厂商在磁带的每一线性英寸中不断地加入更多的字节数。更高密度的磁带不仅对购买和存储来说更加经济，而且可以使数据的备份工作更加快捷。也就是说，如果系统在拷贝文件时必须脱机，使用高密度磁带所造成的系统停工时间就会相

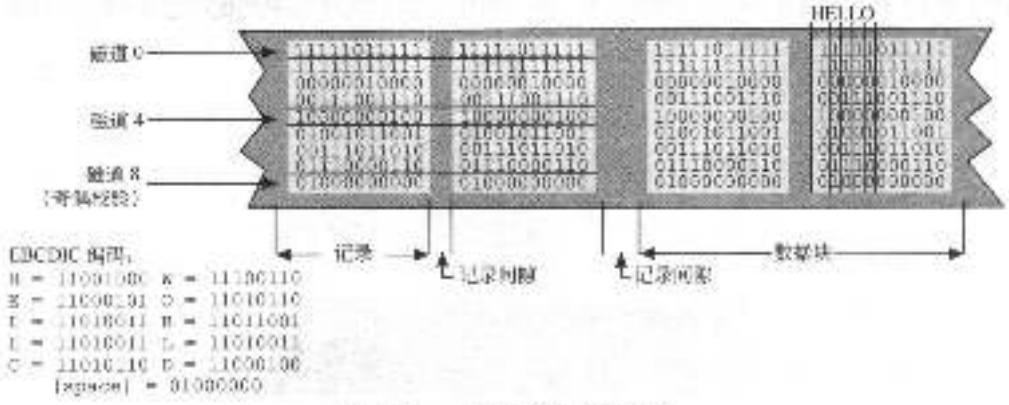


图 7-17 一个九磁道的磁带格式

对减少。如果在写入磁带前，对数据进行压缩处理还可以进一步节约成本（参见 7.8 节）。

各种磁带的革新技术所带来的后果是，产生了太多的磁带标准和出现了知识产权技术。各种尺寸和容量的盒（卡）式磁带已经取代了原来的九磁道并放式的卷轴磁带。类似于数字记录，磁带上使用的薄膜涂层已经取代了原来的氧化物涂层。现在的磁带支持各种不同的磁道密度，并且采用蛇形或螺旋扫描记录方法。

蛇形（serpentine）记录方法是将数据位按照串行方式存放在磁带上。与九磁道格式中字节垂直于磁带边缘的存放方式不同，蛇形记录方法将字节沿磁带长度方向“纵向”地写入，每个字节都平行于磁带的边缘排列。数据流会沿着磁带的长度方向写入直到磁带的末端，然后将磁带反转，在第一个磁道下面的一个磁道写入数据，如图 7-18 所示。这种写入过程会一直持续下去，直到写完磁带上的全部磁道容量。数字线性磁带（digital linear tape，DLT）和 1/4 英寸盒式（quarter inch cartridge，QIC）系统使用蛇形记录方法，每个磁带上有 50 条或者更多的磁道。

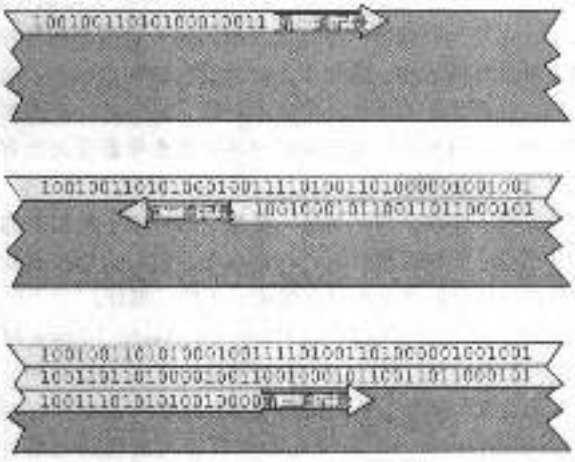
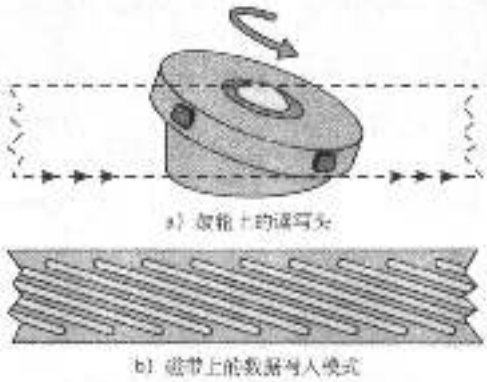


图 7-18 在蛇形线式磁带上的三次记录运动方式

数字音频磁带（digital audio tape，DAT）和 8mm 磁带系统使用螺旋（helical）扫描记录方法。在其他的记录系统中，磁带都是直接通过一个固定的磁头，类似于录音机中磁带的运动方式。DAT 系统是将磁带通过一个倾斜的旋转鼓轮（或称为收盘，capstan），鼓轮上分别有两个读出头和两个写入头，如图 7-19 所示。在写入操作过程中，当写入完成时，读出头便检验数据的完整性。鼓盘以 2000RPM 的速度旋转，旋转方向与磁带移动的方向相反，这种结构与 VCR 上使用的机制很相似。这两个读/写头的驱动装置以相互间成 40 度角的方式写入数据。由于数据是通过两个磁头交迭写入，所以增加了记

录密度。螺旋扫描系统的记录一般比较慢，并且磁带会比采用简单磁带路径的蛇形记录系统更容易磨损。



a) 旋转中的读写头

b) 磁带上的数据写入模式

图 7-19 一种螺旋扫描记录

磁带存储系统从一开始就是大型计算机使用的主要产品。一般看来，磁带以便宜的价格提供了"无限"的存储量。磁带会继续成为大型计算机系统上文件存储和系统备份的主要介质。尽管磁带这种介质本身非常便宜，但是对磁带进行目录编制和处理的代价可能会很高，特别是当磁带库中包含成百上千的磁带卷时。考虑到这个问题，一些厂商生产了各种各样的可以在几秒内对磁带进行编目、取带和装带操作的自动机械装置。很多大型数据中心都使用自动磁带库（robotic tape library），也被称为磁带仓库（tape silo）。现在，最大的自动磁带库可以有几百万亿字节（terabytes）的容量，并且能根据用户的请求在半分钟内装好所需的磁带盒。

7.7 独立磁盘冗余阵列

在 IBM 公司的 RAMAC 计算机引入后的 30 年里，只有一些最大型的计算机具有磁盘存储系统。早期的磁盘驱动器非常昂贵，而且与存储器容量成比率地占用大量的易址面积。磁盘系统需要一个严格受控的工作环境：太热会破坏控制电路；湿度太低可能会引起静电累积，而使磁盘表面磁通量的极化方向出现混乱。磁头撞损，或者其他不可修复的磁盘错误，在商业、科研和学术领域可能造成难以估量的损失。如果在临近工作日结束时发生磁头撞损意外，就意味着今天所有的数据输入都必须从上次备份点起（通常是昨天晚上）重新开始。

很显然，这种情形是大家都不能接受的。而且，随着人们对电子数据存储设备的依赖程度越来越高，这种情况会变得越来越严重。长期以来，人们一直在寻找各种可能的解决办法。归根结底，磁盘还不是非常可靠。当然，制造更可靠的磁盘也只是解决方法的一部分。

1988 年，美国加州大学 Berkeley 分校的 David Patterson、Garth Gibson 和 Randy Katz 三人发表了一篇题为"廉价磁盘的冗余阵列的实例"的论文。首次创造出了 *RAID* 一词。在论文中他们介绍了如何利用若干数量的"廉价"小磁盘（例如微机使用的磁盘）来替代通常用在大型机上的大型昂贵的单磁盘（single large expensive disk，SLED），来提高大型机磁盘系统的可靠性和性能。因为"廉价"（inexpensive）一词的意思是相对的，而且可能会被误导，所以现在大家普遍接受的这个缩写词的正确意思是*独立磁盘冗余阵列*（Redundant Array of Independent Disk，RAID）。

在这篇论文中，Patterson，Gibson 和 Katz 定义了 5 种类型的（称为级，level）的 RAID，每一级 RAID 都具有不同的性能和可靠性特征。原先这些级别的编号是从 1 到 5，后来大家又定义了 RAID 的第 0 级和第 6 级。当然，各个厂商还创造了其他的一些 RAID 级，但许这些层次在今后可能也会成为行业的标准。这些级一般都是被大家普遍接受的 RAID 级的组合形式。本节将简单介绍 7 个 RAID 级的每个层次，以及几个混合系统。这些混合系统为了满足某些特殊的性能和可靠性要求，而由几个不

同的 RAID 级构成的。

7.7.1 RAID Level 0

RAID Level 0，或简称 RAID-0，是将数据块以条带（stripe）的形式存放在几个磁盘表面上。这样一个记录就会占用几个磁盘表面的多个扇区。如图 7-20 所示。这种方法也称为磁盘跨区、块交错数据分带或磁盘分带。分带是简单地将逻辑顺序的数据进行分段，以便把各个分段可以写到多个物理设备上。这些分段可以小到一个单一位，如在 RAID-0 中，或者是某个特定大小的块。

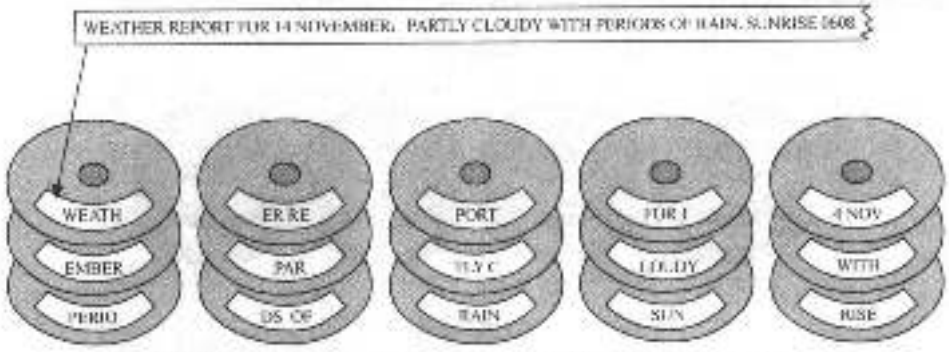


图 7-20 使用 RAID-0 写入的一个记录，没有冗余的块交错数据分带的方式

因为 RAID-0 不提供冗余，所以在各种 RAID 的配置结构中，RAID-0 具有最佳性能。特别是，如果每一个磁盘都有自己独立的控制器和高速缓存。并且 RAID-0 非常廉价。RAID-0 存在的问题在于系统的整体可靠性仅仅为单个磁盘的期望性能的一部分（几分之一）。例如，如果阵列由 5 个磁盘组成，每个磁盘的设计寿命为 50 000 小时（大约 6 年），那么整个系统的期望设计寿命为 50 000/5＝10 000 小时（大约 14 个月）。当磁盘数增加时，失效的概率会随之增加，最后达到某个确定不变的值。因为没有冗余，所以 RAID-0 没有容错能力。因此，RAID-0 所能提供的唯一好处是性能。显然，RAID 缺乏可靠性会引起惊慌。通常，推荐 RAID-0 用于一些非关键的而又要高速读取和写入的数据，或者是改变不太频繁和经常定期备份的数据和要求低成本的场合，以及用于如视频、图像编辑之类的应用程序。

7.7.2 RAID Level 1

RAID Level 1，或者简称 RAID-1（也称为磁盘镜像，disk mirroring），是所有的 RAID 级中具有最佳失效保护的一种方案。RAID-1 在每次写入数据时，都会将数据复制到磁盘驱动器中，称为镜像集（mirror set，或 shadow set）的第二组磁盘上，如图 7-21 所示。这种安排提供了令人满意的性能，特别是在镜像驱动器与主驱动器相差 180°同步旋转时，尽管写入性能要比 RAID-0 慢（因为数据必须写两次），但是 RAID-1 的读取速度更快，因为系统可以从更接近目标扇区的磁盘驱动臂上读取数据。这样在读取数据时，会减少一半的旋转反应时间。RAID-1 最适合于面向事务、高可用率的工作环境，和其他一些需要高容错率的应用，例如会计或工资表。

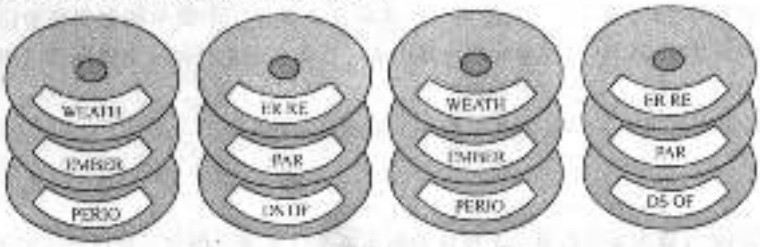


图 7-21 RAID-1，磁盘镜像

7.7.3 RAID Level 2

RAID 1 方案的主要问题是成本代价高：存储一定数量的数据，却需要使用两倍的磁盘容量。显然，更好的方式是只使用磁盘组中的一个或几个磁盘来存储其他磁盘中的数据信息。RAID-2 就定义了这些方法中的其中一种。

RAID-2 将数据分带的思想运用到了极端的情形。RAID-2 在每个条带中只写入一位的数据，而不是采用任意大小的块写入数据，如图 7-22 所示。这样至少需要 8 个磁盘表面才能存放数据。另外，还需要一组磁盘驱动器存放纠错信息，这些纠错位是采用海明编码生成的。

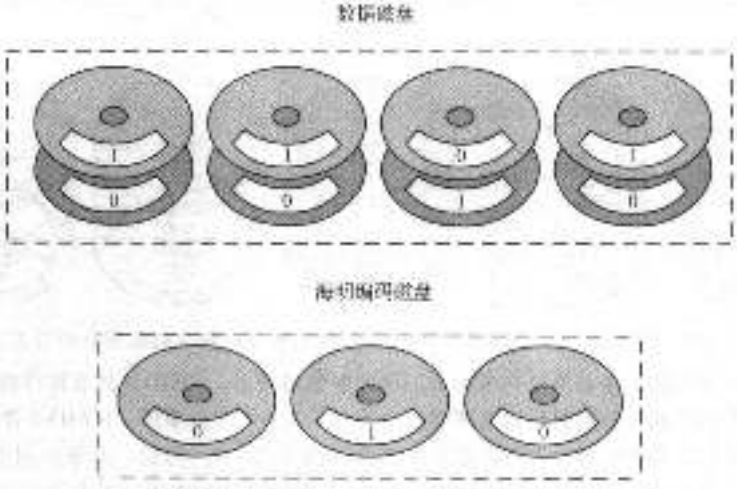


图 7-22 RAID-2 采用海明编码的位交错的数据分带方式

纠正单位错误的海明编码所需要的驱动器数目与需要被保护的数据驱动器的记录表的数目成正比。如果磁盘阵列中的任何一个驱动器损坏，可以使用海明编码字来重建这个出错的驱动器（显然，海明驱动器也可以使用数据驱动器来进行重建）。

因为每个驱动器都只写入一位数据，所以整个 RAID 2 的磁盘组就好像是一个大型的数据磁盘，所有可用存储容量是各个数据驱动器存储容量的总和。所有的驱动器，包括海明驱动器，都必须严格地同步，否则，数据会变乱，海明驱动器也就起不作用。因为生成海明编程非常耗时，所以 RAID-2 对大多数商业应用来说速度太慢。事实上，今天大多数的硬盘驱动器都有内置的 CRC 纠错功能，但是，RAID-2 方式在 RAID-1 和 RAID-3 之间构建了一条理论桥梁，使这两种方式都应用在现实世界中。

7.7.4 RAID Level 3

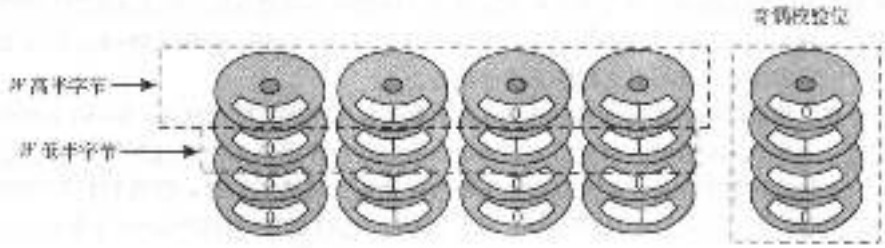
像 RAID-2 一样，RAID-3 是按照每次一位的方式将数据交错分配到各个数据驱动器的条带中。但是，与 RAID-2 所不同的是，RAID-3 只使用一个驱动器来保存一个简单的奇偶校验位，如图 7-23 所示。这种奇偶校验位使用专门的硬件很快就计算出来，具体方法是对每一个数据位（用 b_i 表示）执行一个如下所示的异或（XOR）操作（对偶校验）：

$$\text{Parity} = b_0 \text{XOR} b_1 \text{XOR} b_2 \text{XOR} b_3 \text{XOR} b_4 \text{XOR} b_5 \text{XOR} b_6 \text{XOR} b_7$$

等价为：

$$\text{Parity} = b_0 + b_1 + b_2 + b_3 + b_4 + b_5 + b_6 + b_7 \pmod 2$$

利用这种相同的计算方法可以对一个损坏的驱动器进行重建。例如，假设第 6 号驱动器损坏，并且被替换掉。我们可以对其他 7 个数据驱动器上的数据和奇偶校验驱动器上的数据按如下方式进行计



Device	ASCII	Parity (ASCII)	
		High Nibble	Low Nibble
1	0111 0111	1	1
2	0110 0110	1	0
3	0110 0011	1	1
4	0111 0111	1	1
5	0110 1011	1	1
6	0110 0111	1	0
7	0111 0011	1	1

图 7-23 RAID-3：使用奇偶校验的数据分割位交错磁盘

算，以恢复第 6 号磁盘上的数据：

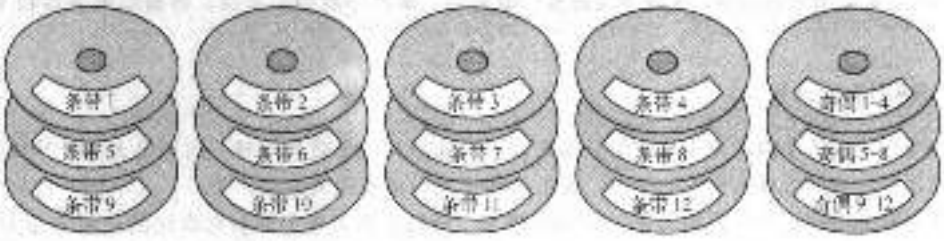
$b_6 = b_1 XOR b_1 XOR b_2 XOR b_3 XOR b_4 XOR b_5$ XOR Parity $XOR b_7$

RAID-3 要求使用与 RAID-2 相同的数据复制方法和同步操作，但是 RAID-3 比 RAID-1 和 RAID-2 都更经济，因为它只使用一个驱动器进行数据保护。RAID-3 在一些商业计算机系统上已经使用了很多年，但是 RAID-3 不太适合应用于面向事务的应用程序。RAID-3 最适合需要读写大块数据块的情况，例如图像或视频处理的应用。

7.7.5 RAID Level 4

RAID-4，像 RAID-2 一样，是另外一个“理论上的”RAID 级。如果将 RAID-4 应用于 Patterson 等人所描述情况，RAID-4 的性能会很糟糕。一个 RAID-4 的磁盘阵列，像 RAID-3 一样，由一组数据磁盘和一个奇偶校验位磁盘组成。RAID-4 将数据写入到统一大小的条带中，而不是每次写入一个数据位到各个驱动器上。RAID-4 会在所有的驱动器上都创建一个大小相同的条带，就如同 RAID-0 所描述的情况。通过对数据带的数据位相互间进行 XOR 操作来创建奇偶校验条带。

我们可以认为 RAID-4 是带有奇偶校验的 RAID-0 方式。但是，增加奇偶校验会导致系统性能的严重下降，原因是数据盘对奇偶校验盘的争用。例如，假设把 RAID-4 数据写入到跨越了 5 个磁盘（其中，4 个数据盘和一个奇偶校验盘）的条带 3 中，如图 7-24 所示。首先，我们必须读取当前占据条带 3 的数据以及对应的奇偶条带，将原来的数据与新的数据执行 XOR 操作，得出新的奇偶校验。然后写入数据带，同时更新奇偶校验条带。



奇偶1-4=(条带1)XOR(条带2)XOR(条带3)XOR(条带4)

图 7-24 RAID-4，带一个奇偶校验盘的块交错数据条带

设想一下，当我们正在处理奇偶校验块时如果有写入请求等待，例如有一个条带 1 的写入请求和

一个条带4的写入请求，磁盘系统将发生什么情况呢？如果正在使用RAID-0或RAID-1的话，这两个等待的请求就可能与写入条带3的操作一起被同时并发地服务。这样，奇偶校验驱动器就变成了一个瓶颈，从而丧失了多磁盘系统所具有的所有性能优势。

有些作者建议如果将条带的大小和要被写入的数据记录大小进行优化处理，RAID-4的性能可以得到改善。还有，这种方法对数据占用相同大小的记录的应用程序（例如声音或视频处理程序）会有好处。然而，大多数的数据库应用都涉及到记录大小的变化范围很大的数据，使得我们不可能对数据库中的大量记录找到一个“最佳”的长度大小。因为RAID-4的性能差，所以在商业上并没有得到应用。

7.7.6 RAID Level 5

大多数人都认为RAID-4可以对单一磁盘出错的情况提供足够的保护。但是，由于奇偶校验盘所导致的瓶颈问题，使得RAID-4不适合应用在需要高事务吞吐量的环境中。可以肯定地说，如果考虑某种类型的负载平衡，将奇偶校验位写到多个磁盘而不只是一个磁盘上，吞吐量问题无疑会得到改善。这也正是RAID-5要做的事情。RAID-5是RAID-4将奇偶校验盘分散到整个磁盘阵列中的情形，如图7-25所示。



奇偶1-3=（条带1）XOR（条带2）XOR（条带3）

图7-25 RAID-5，带分布式奇偶校验的块交错数据条带

因为一些请求能被并发地服务，所以RAID-5在所有的奇偶校验模型中提供了最佳的读操作吞吐量，并且在写操作时也具有令人满意的吞吐量。例如，如图7-25所示，磁盘阵列可以同时服务一个对磁盘4第6条带和一个对磁盘1第7条带的写入请求，因为这些服务请求无论是奇偶校验还是数据操作都分别使用不同的磁盘驱动臂组。然而在所有RAID层次中，RAID-5需要的磁盘控制器是最复杂的。

与其他RAID系统相比，RAID-5能够以最少的成本代价提供最佳的保护。正因为如此，RAID-5在商业应用上已经非常成功。在所有以RAID为基础的应用系统中，RAID-5安装的数量是最大的。RAID-5的主要应用领域包括：文件和应用程序服务器，邮件和新闻服务器，数据库服务器和Web服务器等。

7.7.7 RAID Level 6

前面所讨论的大多数的RAID系统只能允许一次最多有一个磁盘出错。问题是大型计算机系统的磁盘驱动器常常有成群成簇失效的倾向。发生这种情况一般有两大原因。第一，几乎同一时间生产的磁盘会在相同的时间到达它们预期使用寿命的终点。所以，如果被告知新磁盘组的使用寿命大约是6年，那么可以预计第6年磁盘系统就会有问题，或许出现多个磁盘同时并发失效的情况。

第二，磁盘驱动器的损坏通常是由于某些灾难性事件引起的，比如电涌波动。电涌波动可能会在同一时刻损坏所有的磁盘驱动器。最不耐用的磁盘最先损坏，接着是第二个不耐用的磁盘损坏，如此持续下去。类似这种连续的磁盘损毁可能会延续几天甚至几周的时间。如果这些磁盘的持续损坏碰巧

在磁盘的平均修复时间（Mean Time To Repair，MTTR）（包括打电话和修复人员到达所需时间）之内发生，那么还没有替换第一个损坏的磁盘之前第二个磁盘也可能损坏了，因而整个磁盘阵列就变得无法继续使用和服务。

对于具有高可用性的系统必须能够允许多个磁盘同时并发失效的情况发生，特别是在 MTTR 可能会是一个很大数字的情形。如果将磁盘阵列设计为可以承受两个磁盘驱动器的并发失效，就可以有效地把 MTTR 的数字扩大一倍。事实上，RAID 1 的磁盘系统就具有这种生存能力，只要某个磁盘及其镜像盘不在同一时刻失效，那么 RAID 1 磁盘阵列就可以在失去一半磁盘后还能够生存。

RAID-6 提供了解决这种大量磁盘失效问题的一种经济实用的方案。为此，RAID-6 对每排（rank）（或者每个水平行）驱动器使用了两组纠错条带。除了采用奇偶校验外，RAID-6 还使用 Reed Solomon 纠错编码来增加第二层保护。为每个数据条带配置两个校验的条带当然会增加存储器的成本。如果没有保护的数据存储在 N 个磁盘上，那么 RAID-6 增加的这种保护就需要 $N+2$ 个磁盘。因为需要写入两类奇偶校验，所以 RAID 6 所提供的写入性能相当差。一种 RAID-6 的配置结构如图 7-26 所示。

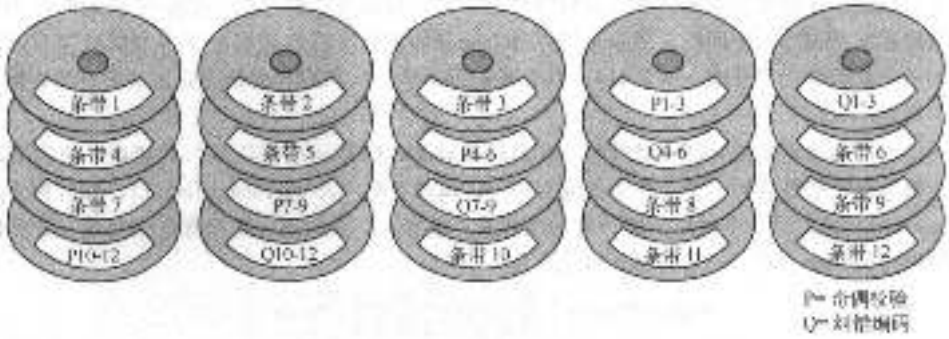


图 7-26 RAID-6：双重出错保护的块交错数据条带

直到最近，还没有商业配置的 RAID-6 系统。产生这种情况有两种原因。第一，生成 Reed Solomon 编码需要相当大的费用代价。第二，更新磁盘上的纠错编码要求双倍的读/写操作。IBM 公司是第一个（到现在为止也仅有的一个）利用它们的 RAMAC RVA 2 Turbo 磁盘阵列将 RAID-6 带到了市场上。RVA 2 Turbo 磁盘阵列将磁盘条带的运行“日志记录”存放在磁盘控制器的高速缓存存储器中，以消除 RAID-6 的写入损失。日志数据允许磁盘阵列每次处理一个数据条带，并且在数据被写入磁盘前为整个条带计算全部的奇偶校验位和纠错编码。在这里，系统决不会将数据重新写入到更新前数据所占据的同一个条带中。相反，一旦这个被更新条带已经在其他位置写入，那么这个先前被占用的条带就会被标记为空闲空间。

7.7.8 混合 RAID 系统

许多大型的计算机系统并不局限于只使用一种类型的 RAID。在某些情况下，平衡磁盘系统的高可用性和经济性是非常重要的。例如，我们可能希望使用 RAID-1 来保护包含操作系统文件的驱动器，而对于数据文件使用 RAID-5 就足够了。RAID-0 非常适合于存放长程序运行过程仅作为临时使用的“临时”文件，并且 RAID-0 较快的磁盘访问速度可以有效地减少程序执行的时间。

有时，可以使用多个 RAID 方案组合起来构建一种“新型”的 RAID。RAID-10 就是这样一种磁盘系统，它组合了 RAID-0 的分带方式和 RAID-1 的镜像功能。虽然代价非常昂贵，但是 RAID-10 可以提供优良的读取性能和最佳的可用性。不管成本如何，还是有几套 RAID 10 系统成功地进入了商业市场。

在阅读了前面几节的内容后，我们应该很清楚，编号较高的 RAID 层次不一定会是“更好”的 RAID 系统。无论如何，还是有许多人会很自然地认为更高编号的东西通常表示要比编号低的东西更好一些。由于这个原因，称为 RAID 咨询委员会（RAID Advisory Board，RAB）的行业协会最近已经

在组织和重新命名上述的RAID系统。本书选择保留了“Berkeley”的命名方法，因为这种命名方法得到了更加广泛的认同。

7.8 数据压缩

无论存储器多便宜，也不管购买的数量有多少，我们似乎从来都不会认为存储器系统已经足够。新购买的巨大磁盘通常很快就会装满原本希望可以存放到老磁盘上的内容。不久，我们就又开始在市场上购买另一组新的磁盘系统。很少有人或公司可以拥有无限的资源，因此我们必须对现有的资源进行优化使用。优化方法之一就是使要存储的数据变得更加紧凑，在写入磁盘之前对数据进行压缩。实际上，可以使用压缩技术为奇偶校验或镜象磁盘组留出空间，“免费”为系统增加RAID!

数据压缩除了节省空间外还有更多的好处。数据压缩同样也能节省时间和帮助优化资源。例如，如果压缩和解压的操作都在I/O处理器上执行，那么数据移入和移出存储器子系统所需要的时间就会更少，这样系统就会有更多的时间让I/O总线做其他的工作。

在通信线路中使用数据压缩技术发送信息所带来的好处是显而易见的：数据传输的时间会更短和主机上数据存放所占用的空间也会更少。尽管这个问题的细节研究已经超出了本书的范围（读者参阅本章的参考文献部分可以获取一些资料），但是要全面理解I/O和数据存储还应该了解有关数据压缩的一些基本概念。

在评估不同的压缩算法和压缩硬件时，我们通常最关心的是压缩算法执行的速度、速度有多快并且使用这种数据压缩算法后一个文件会变成有多小。*压缩系数*（compression factor），有时称为*压缩比*（compression ratio），是一个可以快速计算的统计量。事实上，人们很容易理解这个问题。计算压缩系数有多种不同的方法。这里。采用如下的计算方法：

$$\text{压缩系数} = 1 - \left[\frac{\text{压缩后的文件大小}}{\text{压缩前的文件大小}}\right] \times 100\%$$

例如，假设开始有一个100KB的文件，然后使用某种压缩算法对文件进行压缩处理。在算法完成后，文件的大小变成了40KB。因此，该特定文件所得到的压缩比为：$\left(1-\left(\frac{40}{100}\right)\right)\times 100\% = 60\%$。在推断出这一算法是否总是（always）可以产生60%的文件压缩的结论之前，我们应该进行大量的统计研究。然而，一旦具备了少许理论背景知识后，我们就能够确定某种压缩算法对于一些特定的信息或信息类型的期望压缩比。

数据压缩技术的研究属于*信息理论*（information theory）这个更大的研究领域内的一个分支。信息理论是研究信息存储和编码方式的科学。这种理论由贝尔（Bell）实验室的一位科学家Claude Shannon在1940年创建的。Shannon建立了很多信息度量的方法，其中最基本的内容是*熵*（entropy，或称为*平均信息量*）。熵用来测量一个消息中的信息量。具有较高熵的消息比较低熵的消息载有更多的信息。这种定义意味着一个载有较低信息量的消息会比一个载有较高信息量的消息压缩到更小的容量。

要确定一个信息的熵，首先需要确定这条消息中每个符号的频繁。读者很容易按照概率来思考符号频率这个概念。例如，在如下的著名程序输出语句中：

HELLO WORLD!

字母L出现的频率是3/12或者1/4。使用符号表示，则为$P(L)=0.25$。为了将该概率映射为某个编码字中的二进制位，取概率以2为底的的对数（log）。这里，编码字符L所需要的最小位数为：$-\log_2 P(L)=2$位。消息的熵是在该消息中编码每个符号所需要的二进制位数的加权平均值。如果某个符号x出现在一个消息中的概率为$P(x)$，那么该字符x的熵H为：

$$H = -P(x)\times\log_2 P(x)$$

整个消息的平均熵则是对消息中所有n个符号出现的概率加权求和：

$$\sum_{i=1}^{n} -P(x_i)\times\log_2 P(x_i)$$

熵为编码一个消息所需的二进制位数建立了一个低限。特别地，如果将整个消息的字符数乘以加权熵值，就得到了编码一个消息而不丢失信息所需要的理论上的最少二进制位数。除了这个低限外的其他位并不会增加信息，因此这些位是冗余的（redundant）。数据压缩的目的是在保留原有信息量的同时除去冗余性。可以通过下面的公式，对包含在长度为 l 的编码词中的一个长度为 n 的编码消息的每个字符的平均冗余量进行量化处理：

$$\sum_{i=1}^{n} P(x_i) \times l_i - \sum_{i=1}^{n} -P(x_i) \times \log_2 P(x_i)$$

当对某个给定的消息比较各种编码方案的有效性时，这个公式非常有用。按照数据压缩的观点，编码一个消息时，具有最少冗余量的编码方案是比较好的编码方案。当然，我们还必须考虑完成编码的速度和计算的复杂性，以及应用程序的细节问题，最后才能确定哪种方法更好。

上式的一个直接应用是求一个文本消息的熵和冗余。如果使用一个定长的编码方式，例如 ASCII 或者 EDCDIC，上式左边的求和就是编码的长度，通常为 8 位。在 HELLO WORLD! 的例子中，通过利用上式右边的求和，我们发现平均的符号熵大约是 3.022。这意味着如果达到理论上的最大熵值，只需使用 3.022 位/字符×12 字符 = 36.26 或 37 位来对整条消息进行编码。因此，这条消息的 8 位 ASCII 编码的表示形式中就有 96−37 = 59 个冗余位。

7.8.1 统计编码

上面所描述的熵的量度方法可以作为设计压缩编码的基础，利用统计编码方法可使被压缩的消息中的冗余最小化。一般来说，采用统计编码的压缩过程相对较慢，而且与 I/O 过程密切相关。在消息被压缩和写入存储器之前，系统需要通读文件两次。

要求通读文件两次的原因是，第一次对文件的通读只是用来计算每一个符号出现的次数。计数值将用来计算每一个不同的符号在源消息中出现的概率。我们会按照所计算的概率对源消息中的每一个符号赋值。随后，最新的赋值会与编码该文件所需的一些信息一起写入到文件中。如果这个被编码的文件，连同编码该文件所需的赋值表，比原来的文件要小，我们就说发生了数据压缩。

赫夫曼（Huffman）编码和算术编码是两种基本的统计数据压缩方法。在大量流行的压缩程序中，可以找到这两种方法的某些变化形式。首先从赫夫曼编码开始，在下面章节中将检查研究这两种方法。

赫夫曼编码

假设在确定了源消息中每个符号的概率后，需创建一个可变长度的编码来将最短的编码字分配给使用频率最高的符号。如果这个编码字短于信息字，那么压缩后的消息结果显然也会比信息字要短。摩尔斯（Morse）编码是 19 世纪开始出现的编码形式，它是 David A. Huffman 编码的其中一种形式。

摩尔斯编码是根据英语著作中各个字母的典型使用频率来设计的。从图 7-27 中可以看出，较短的编码代表了英语中使用频率较高的字母。这种统计的字母频率显然不适用于单一的消息。一个著名的例外情况是 “a telegram from Uncle Zachary vacationing in Zanzibar, requesting a few quid so he can quaff a quart of quinine!” 因此，这种最准确的统计模型对每条消息来说都需要个别地加以考虑。为了准确分配各个编码字，赫夫曼算法会利用从源消息中求出的符号概率来构建一个二进制树形网络。树的遍历给出了消息中每个符号的位图（位组合模式）分配形式。我们用一个简单的童谣来说明这一处理过程。为了更清楚，我们将童谣写成如下没有标点符号全部由大写字母组成的形式。

HIGGLETY PIGGLETY POP
THE DOG HAS EATEN THE MOP
THE PIGS IN A HURRY THE CATS IN A FLURRY
HIGGLETY PIGGLETY POP

我们把童谣中每个字符出现的次数写成表格形式。并且使用缩写符号＜ws＞（空白）来表示单词间的空格字符和换行字符。（参见表 7-1）

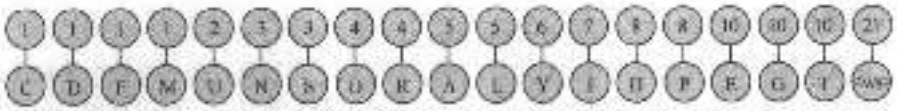


图 7-27 摩尔斯国际编码

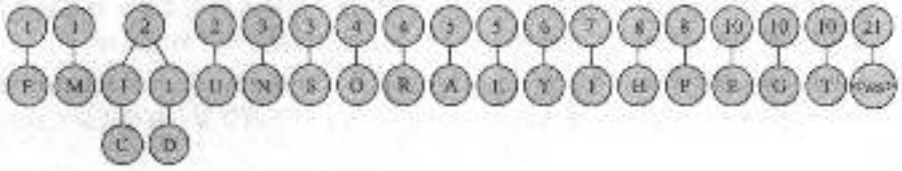
表 7-1 字母出现的频率

字母	计数	字母	计数
A	5	N	3
C	1	O	4
D	1	P	2
E	10	R	4
F	1	S	3
G	10	T	10
H	8	U	2
I	7	Y	6
L	5	<ws>	21
M	1		

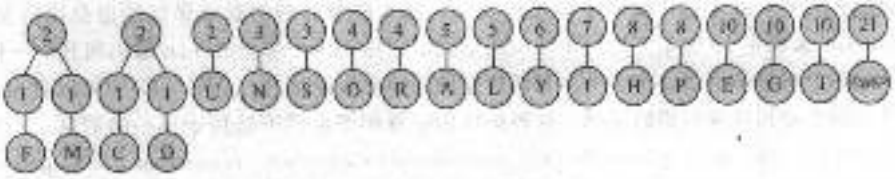
这些字母出现的频率利用树的两个节点与每个字母关联。这些树的集合（称为森林，forest）按照字母频率的循序排列成如下的直线形式：



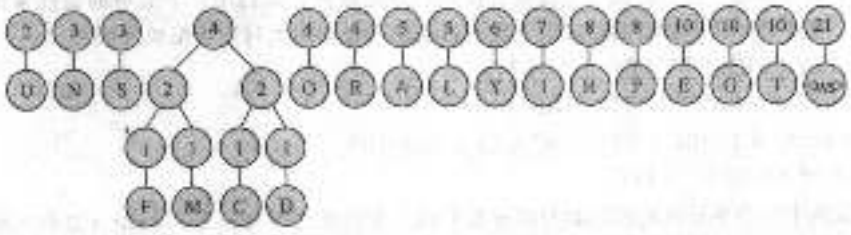
下面通过将两个频率最小的节点连接起来，开始构建二进制树。因为最小频率的节点有 4 个，所以可以任意选择最左端的两个节点。这两个节点的组合频率和为 2。接着，创建一个标记该频率和的母节点，并且将它放回到森林中，对应于母节点标记所决定的位置，如下图所示：



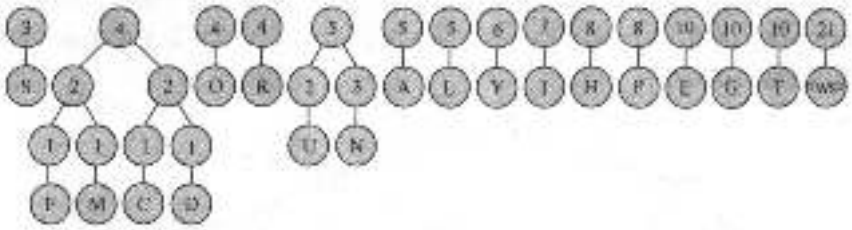
对最小频率节点重复上面的过程，如下图所示：



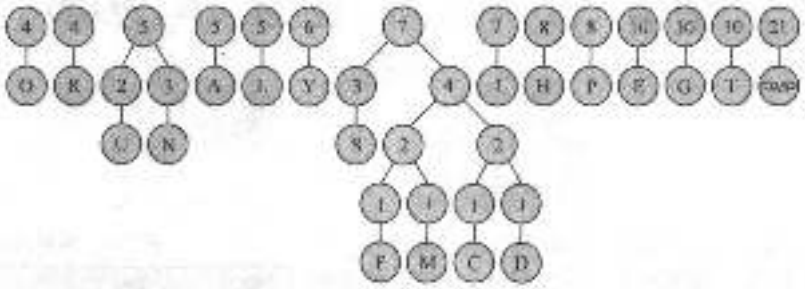
这两个最小节点是 F、M、C、D 的母节点，将它们放在一起，相加的频率和为 4，属于从左边开始的第 4 个位置，如下图所示：



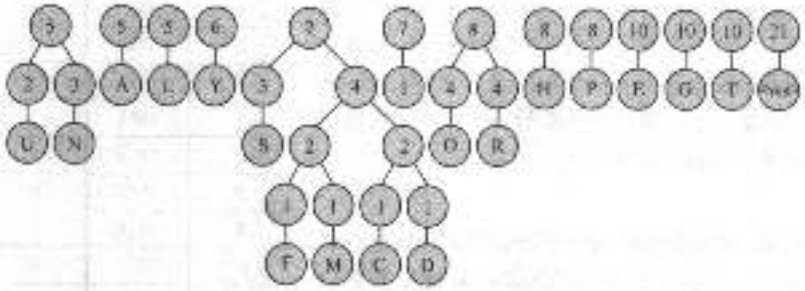
最左端两个节点的频率相加等于 5，于是要将它们移到树中间的新位置上，如下图所示：



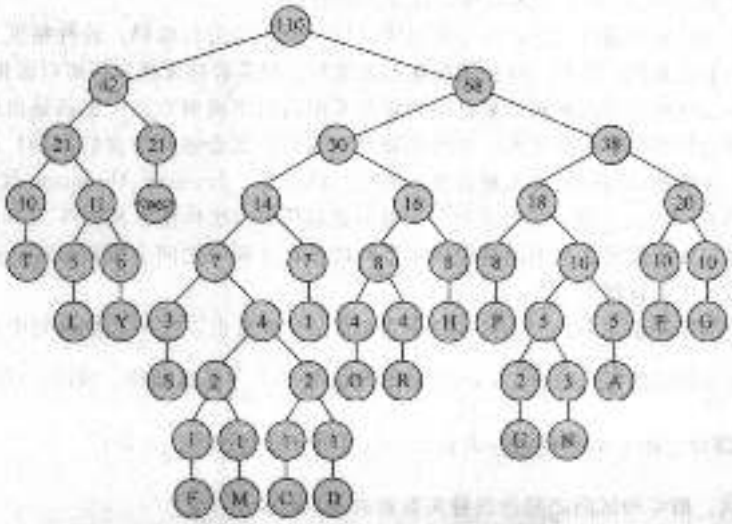
现在两个最小的节点的频率相加等于7，于是又要创建一个新的母节点，并将这个子树移到森林中何与其他频率为7的节点放在一起，如下图所示：



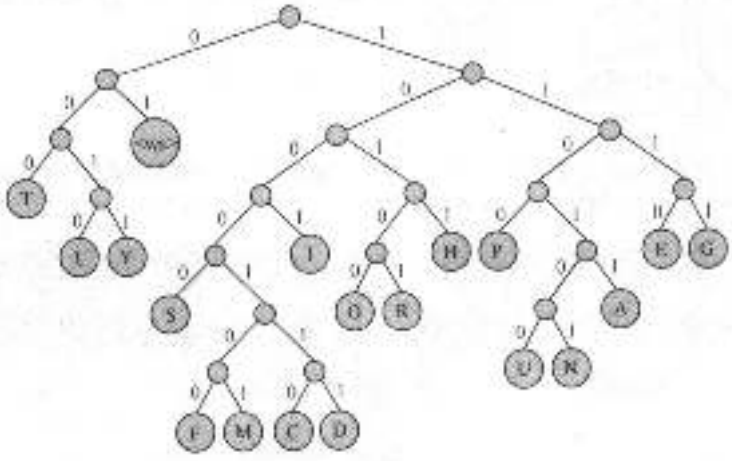
现又将最左端的两个节点组合起来创建一个频率为8母节点，并将它放回到森林中，如下图所示：



经过多次的叠加过程，最后树的形状如下图所示：



该树构建了一个框架，由此可以给消息中的每个符号分配一个赫夫曼编码值。利用二进制数1来标记二叉树中每一条右分支，而二叉树每一条左分支则标记为0。其结果如下图所示（删除了频率节点会使图看起来更加清楚）。



现在我们所要做的事情就是从树根节点出发遍历每一个叶节点，并记录下沿途所遇到的二进制数字。这样，所完成的编码方案如表7-2所示。

表7-2 编码方案

字符	代码	字符	代码
(sp)	01	O	10100
T	000	R	10101
L	0010	A	11011
Y	0011	U	110100
I	1001	N	110101
H	1011	F	1000100
P	1100	M	1000101
E	1110	C	1000110
G	1111	D	1000111
S	10000		

正如我们所看到的一样，频率最高的符号在编码中占用最少的位。这个消息的平均熵大约为3.82位/符号，对这个信息进行压缩的极限位数为110符号×3.82位=421位。赫夫曼编码对这个消息实际给出的编码数是426位，或者说比理论上所需要的最少位数大约多出了1%。

算术编码

从理论上来说，赫夫曼编码通常不能实现最佳压缩，这是因为赫夫曼编码在结果编码中受到限制需要使用整数位数。在上节介绍的案例中，符号S的平均熵近似等于1.58。一个最佳的代码会使用1.58位对每次出现的符号S进行编码。因此，如果采用赫夫曼编码方法，最少要使用2位二进制位进行编码。这种精度上的欠缺使得最后的结果多出了5个冗余位。当然，这种情况还不算太坏，但是看起来我们还可以做得更好一些。

赫夫曼编码不能达到最佳的编码效果是因为它所采用的概率映射方式。也就是说赫夫曼编码要将实数集的元素映射为整数子集中的元素。正因为如此，我们肯定会遇到上面的问题！那么，为什么不能设计某种实数对实数的映射形式来实现数据压缩呢？1963年，Norman Abramson就设想出了这种映射方式，随后由Peter Elias发表。这种实数对实数的数据压缩方法称作算术编码（arithmetic coding）。

从概念上来说，算术编码是利用消息中符号集的概率在0和1之间分割实数轴。使用越频繁的符号，分割所得到的区间块就越大。

返回大家喜爱的程序输出问题：*HELLO WORLD*！不难看出，这个祈使语句中共有12个字符。在这些符号中，出现最低的概率是$\frac{1}{12}$。而所有其他的概率都是$\frac{1}{12}$的整数倍。因此，可将0和1之间的区间划分为12个部分。除了L和O外，其他每个符号都分配得到了$\frac{1}{12}$的区间，符号L和O则分别获得了$\frac{3}{12}$和$\frac{2}{12}$的区间。概率和区间之间的映射关系如表7-3所示。

表 7-3 对于 HELLO WORLD! 概率与区间的映射关系

符号	概率	区间	符号	概率	区间
D	$\frac{1}{12}$	[0.0...0.083)	R	$\frac{1}{12}$	[0.667...0.750)
E	$\frac{1}{12}$	[0.083...0.167)	W	$\frac{1}{12}$	[0.750...0.833)
H	$\frac{1}{12}$	[0.167...0.250)	(space)	$\frac{1}{12}$	[0.833...0.917)
L	$\frac{3}{12}$	[0.250...0.500)	!	$\frac{1}{12}$	[0.917...1.0)
O	$\frac{2}{12}$	[0.500...0.667)			

通过连续地划分与符号所分配区间成比例的数值范围（从 0.0 开始到 1.0 结束）来对消息进行编码。例如，假如当前的区间位置是$\frac{1}{8}$，而字母 L 获得了$\frac{1}{4}$的当前区间，如上面表 7-3 所示。接下来，对符号 L 进行编码，将$\frac{1}{8}$乘以$\frac{1}{4}$得到字符 L 的新的当前区间$\frac{1}{32}$，如果下一个字符是另外一个字母 L，那么将$\frac{1}{32}$再乘以 1/4 又可以得到当前的区间值为$\frac{1}{128}$。这个编码过程一直会持续下去，直到整条消息编码的完成。在学习了下面的伪代码后，这个编码过程就会变得非常清楚。有关输出 *HELLO WORLD!* 的伪代码的详细描述如图 7-28 所示。

```
ALGORITHM Arith_Code (Message)
  HiVal ← 1.0                                  /* Upper limit of interval. */
  LoVal ← 0.0                                  /* Lower limit of interval. */
  WHILE (more characters to process)
       Char ← Next message character
       Interval ← HiVal - LoVal
       CharHiVal ← Upper interval limit for Char
       CharLoVal ← Lower interval limit for Char
       HiVal ← LoVal +  Interval * CharHiVal
       LoVal ← LoVal +  Interval * CharLoVal
  ENDWHILE
  OUTPUT (LoVal)
END Arith_Code
```

符号	区间	CharLoVal	CharHiVal	LoVal	HiVal
				0.0	1.0
H	1.0	0.167	0.25	0.167	0.25
E	0.083	0.083	0.167	0.173889	0.180861
L	0.006972	0.25	0.5	0.1756320	0.1773250
L	0.001743	0.25	0.5	0.17606775	0.17650350
O	0.00043575	0.5	0.667	0.176285625	0.176338395
<sp>	0.00007277025	0.833	0.917	0.1763462426	0.1763523553
W	0.00000611270	0.75	0.833	0.1763508271	0.1763513345
O	0.00000050735	0.5	0.667	0.1763510808	0.1763511655
R	0.00000008473	0.667	0.75	0.1763511373	0.1763511444
L	0.00000000703	0.25	0.5	0.1763511391	0.1763511409
D	0.0000000176	0	0.083	0.1763511391	0.1763511392
!	0.0000000015	0.917	1	0.176351139227	0.176351139239
				0.176351139227	

图 7-28 利用算术编码方法对 *HELLO WORLD!* 进行编码

利用这一编码过程的反过程可以对消息进行译码，下面给出了译码过程的伪代码。图 7-29 给出的是上述伪代码的跟踪过程。

```
ALGORITHM Arith_Decode (CodedMsg)
    Finished ← FALSE
    WHILE NOT Finished
       FoundChar ← FALSE              /*  We could do this search much more    */
       WHILE NOT FoundChar            /* efficiently in a real implementation. */
          PossibleChar ← next symbol from the code table
          CharHiVal ← Upper interval limit for PossibleChar
          CharLoVal ← Lower interval limit for PossibleChar
          IF CodedMsg < CharHiVal AND CodedMsg > CharLoVal THEN
             FoundChar ← TRUE
          ENDIF                       / * We now have a character whose interval  */
       ENDWHILE                       /* surrounds the current message value.     */
       OUTPUT(Matching Char)
       Interval ← CharHiVal - CharLoVal
    CodedMsgInterval ← CodedMsg   CharLoVal
    CodedMsg ← CodedMsgInterval / Interval
    IF CodedMsg = 0.0 THEN
       Finished ← TRUE
    ENDIF
END WHILE
END Arith_Decode
```

符号	LowVal	HiVal	区间	CodedMsg-Interval	CodedMsg
					0.176351139227
H	0.167	0.25	0.083	0.009351139227	0.112664328032
E	0.083	0.167	0.084	0.029664328032	0.353146762290
L	0.25	0.5	0.250	0.103146762290	0.412587049161
L	0.25	0.5	0.250	0.162587049161	0.650348196643
O	0.5	0.667	0.167	0.1503481966	0.900288860265
<sp>	0.833	0.917	0.084	0.0672886027	0.8010547935
W	0.75	0.833	0.083	0.051054793	0.615117994
O	0.5	0.667	0.167	0.11511799	0.6893293
R	0.667	0.75	0.083	0.0223293	0.2690278
L	0.25	0.5	0.250	0.019028	0.076111
D	0	0.083	0.083	0.0761	0.917
	0.917	1	0.083	0.000	0.000

图 7-29 *HELLO WORLD!* 解码的跟踪过程

大家可能已经注意到算术编码/解码算法都不包括任何一种类型的错误检测。之所以这样做，是为了更加清楚地描述问题。真正实现算术编码时，不但要保证在编码结果中有足够的位数来满足熵（信息量）的要求，而且要防止浮点数下溢的情况发生。

当对消息进行解码时，浮点数表示法的差异也可能引起算术编码算法丢失 0 的情况发生。事实上，在编码过程中总是会在消息的尾部插入一个终止消息字符，这样可以防止在解码时出现丢失 0 的情况。

7.8.2 Ziv-Lempel (LZ) 字典系统

尽管算术编码方法可以获得几乎最佳的数据压缩，但是它的速度甚至比赫夫曼编码方法更慢。这是因为在编码和译码的过程中，算术编码方法必须要执行浮点操作。如果速度是我们首要关心的问题，那么我们可能希望考虑其他的压缩方法，这也就意味着我们不可能获得一种理想的编码。的确，如果能够避免对输入消息进行两次扫描，那么就可以获得可观的速度。这就是下面要介绍的字典方法。

Jacob Ziv 和 Abraham Lempel 率先推出了在读取信息和写入编码字节的过程中构建一部字典的思想。基于字典的算法的输出内容既包含文字信息，又包含指示先前已经存放字典中的信息的指针。如果数据中存在大量的局部冗余，例如一长串的空格或数字 0，这种基于字典的压缩技术的效果会特别好。尽管我们称之为 LZ 字典系统，但是如果要使用作者的全名，则“Ziv-Lempel”会比“Lempel-Ziv”更好一些。

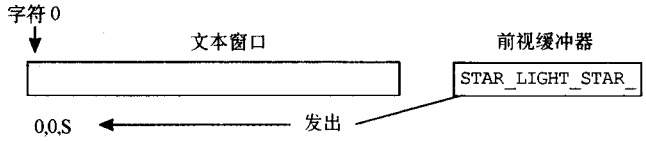
Ziv 和 Lempel 在 1997 年发表了他们的第一个算法，被称为 *LZ77 压缩算法*（LZ77 compression algorithm）。LZ77 压缩算法采用一个文本窗口和一个前视（look ahead）缓冲器。前视缓冲器包含需要编码的信息内容。文本窗口则作为字典。如果前视缓冲器中的任意字符可以在字典中找到的话，那么在窗口中对应的文本内容的位置和长度都会被写到算法的输出中。如果在字典中没有找到所要求的文本内容，那么在写这个没有编码符号时会设置一个标志，指示这个符号应该被作为文字。

现在有许多不同类型的 LZ77 压缩方法，但是所有的 LZ77 压缩方法都是基于同一种思想。下面通过一个例子，使用另外一个童谣来解释这一基本思想。为了清晰起见，我们用下划线来取代童谣中的所有空格。

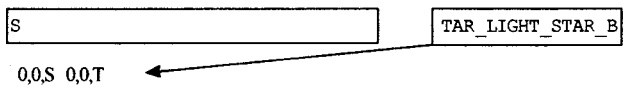
STAR _ LIGHT _ STAR _ BRIGHT _
FIRST _ STAR _ I _ SEE _ TONIGHT _
I _ WISH _ I _ MAY _ I _ WISH _ I _ MIGHT _
GET _ THE _ WISH _ I _ WISH _ TONIGHT

为了说明问题，我们将使用 32 字节的文本窗口和一个 16 字节的前视缓冲器（实际上，这两个领域通常相隔几千个字节）。首先将上面的文本读入到前视缓冲器中。但是文本窗口还没有任何内容，现在把符号 S 放进文本窗口并写入一个三重态（三元字节）到输出：

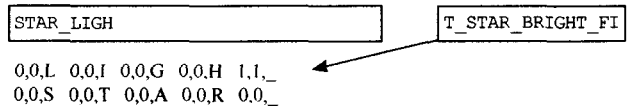
1. 文本窗口中匹配字符相对于文本的偏移量
2. 匹配字符串的长度
3. 前视缓冲器中紧跟匹配短语后面的第一个符号



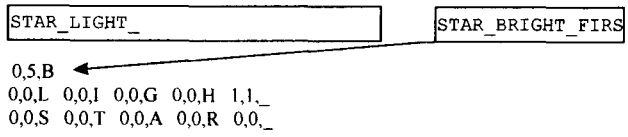
在上例中，因为在文本窗口没有匹配的内容，所以偏移量和字符串的长度都是 0。前视缓冲器中的下一个字符 *T* 也没有相匹配的内容，因此它也将作为一个索引和长度都是 0 的文字写入。



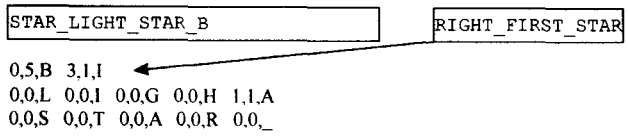
继续读取文字，直到前视缓冲器中的 *T* 作为第一个字符出现。这个 *T* 与文本窗口中位置 1 上的 *T* 相匹配。前视缓冲器中紧跟在字符 *T* 后面的字符是一个下划线，它也就是要写到输出的三重态中的第三项。



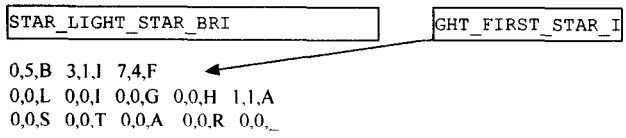
现在，前视缓冲器向前移位了两个字符。*STAR _* 已经成了前视缓冲器的开始部分。它与文本窗口中第一个字符位置（位置 0）相匹配。于是，我们写下 0，5，B，因为 B 是前视缓冲器中紧跟在 *STAR _* 后面的第一个字符。



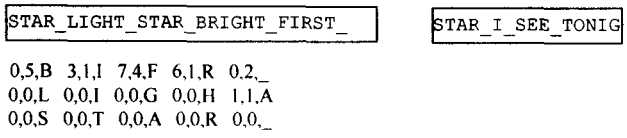
我们又向前移位前视缓冲器中的六个字符，然后查找字符 *R* 的匹配。在文本位置 3 中找到了一个 *R*，又写下 3，1，I。



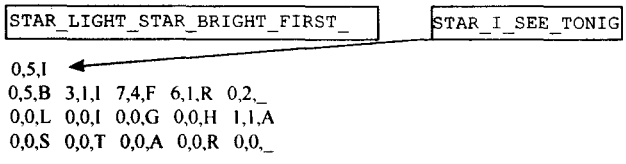
现在，*GHT* 成了前视缓冲器的开始部分。它与文本中从第 7 个位置开始的 4 个字符相匹配，我们再写入 7，4，F。



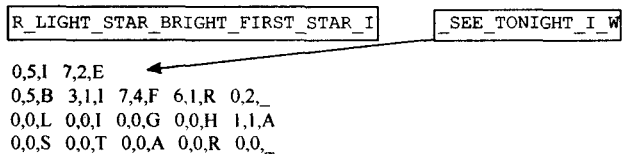
再经过几次这种重复过程以后，文本窗口几乎全满了，如下图所示：



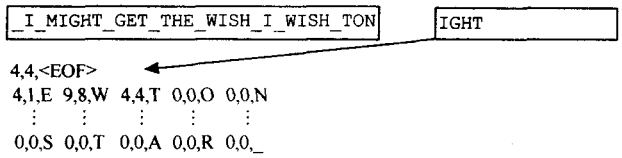
在匹配了 *STAR _* 和文本中位置 0 处字符串后，*STAR _ I* 这 6 个字符就会被移出缓冲器，而进入文本窗口。为了能够容纳全部的这 6 个字符，在处理完 *STAR _* 后，文本窗口必须要右移 3 个字符。



在为 STAR _ I 写好编码，并对文本窗口进行移位以后，*_ S* 现在处于缓冲器中的开始位置。这些字符与文本中位置 7 处的字符串匹配。



按照这种方式继续下去，最终可以到达文本的结束处。算法要处理的最后一些字符是 *IGHT* 。这些字符与位置 4 处的文本内容匹配。因为缓冲器中的 *IGHT* 后面已经没有了其他的字符，所以最后写出的三重态是一个文件字符结束的标记符号，<*EOF*>。



本例共有 36 个三重态被写到输出中。如果采用 32 字节的文本窗口，那么索引只需 5 位就可以指示任何一个文本字符。因为前视缓冲器的宽度是 16 字节，能够匹配的最长字符串也是 16 字节，所以最多需要用 4 位来存储这个字符串的长度。索引需要 5 位，字符串长度需要 4 位，每个 ASCII 字符要用 7 位，而每个三重态要求 16 位或 2 字节。这个童谣包含了 103 个字符，将它们存储在磁盘上需要 103 个未压缩的字节。而存储压缩消息却只要求 72 个字节，这样得到的压缩系数为 $\left(1-\left(\frac{72}{103}\right)\right)\times 100=30\%$。

我们可以合理地认为，如果增大文本窗口，就会增加在文本窗口中查找与前视缓冲器中字符相匹配的可能性。例如，字符串 _*TONIGHT* 出现在童谣中的第 41 个位置和第 96 个位置上。因为两次出现字符串 _*TONIGHT* 之间的间隔有 48 个字符，如果使用一个 32 个字符的文本窗口，那么第一次出现的字符串 _*TONIGHT* 就不可能作为第二次出现的字符串 _*TONIGHT* 的字典条目。如果将文本窗口扩大到 64 字节，那么第一次出现的 _*TONIGHT* 就可以对第二次出现的 _*TONIGHT* 进行编码，并且只会增加 1 位到每个编码的三重态中。然而，在本例中，一个扩展为 64 字节的文本窗口只会在输出中减少两个三重态，即从 36 个减少到 34 个。因为这个扩大的文本窗口要求有 7 位来表示索引，所以每个三重态需要由 17 位组成。这样，这条被压缩的讯息总共占用了 17×34＝578 位或 73 个字节。因此，本例中使用这个较大的文本窗口，实际上多用了几个位。

在压缩的过程中，如果在文本窗口和缓冲器之间没有找到任何匹配，那么就会有简并的情况发生。例如，如果有一个 36 字符的字符串，它由 26 个英文字母和 0 到 9 的数字组成，即 ABC...XY012...89，在这里没有可以匹配的字符。因此，压缩算法的输出将会是 36 个（0，0,?）形式的三重态。最后，将以一个原字符串长度的输出三重态结束，或者说压缩长度扩大（expansion）了 200%。

幸运的是，在实际中很少发生像刚才引用的这种例外情况。在许多流行的压缩工具中，LZ77 压缩方法有各种变化形式，包括用途比较广的 PKZIP 方法。IBM's RAMAC RVA 2 涡轮式磁盘阵列直接在磁盘控制电路中实现 LZ77 压缩方法。这种压缩过程以硬件的速度进行，而且对用户来说是完全透明的。

自从 Ziv 和 Lempel 在 1977 年发表了他们的算法以来，基于字典的压缩方法一直是非常活跃的一个研究领域。一年以后，当他们发布他们的第二个基于字典的压缩算法时，Ziv 和 Lempel 对他们原来的工作做了很大的改进，第二种基于字典的压缩算法就被称为 LZ78。LZ78 与 LZ77 不同，它取消了原来固定大小的文本窗口的限制。相反，LZ78 创造了一种被称为 trie 的特殊的树型数据结构。在从输入读取数据时，LZ78 会采用一些标记符号来填充这种树形结构。如果有需要，这种树的每个内部节点都有多个子节点。与 LZ77 压缩方法不同，LZ78 并不是将字符写入磁盘，而是把指针写入到树型结构的各个标记符号中。在完成消息的编码后，LZ78 会将整个 trie 写入磁盘，而且在对该消息进行译码前首先要读取这棵树。有关 trie 的更多的信息可以参阅附录 A。

7.8.3 GIF 和 PNG 压缩

实现 LZ78 数据压缩所面临的最大问题是如何对 trie 的标记符号进行有效的管理。如果字典太大，所需要的指针数目就可能变得比原始数据还要多。现在已经找到了解决这个问题的许多方法，其中的

一种方法还引起了激烈争论和法律问题。

1984年，Sperry计算机公司（现在称为Unisys）的一名雇员Terry Welsh发表了一篇论文，论文描述了一种管理LZ78字典的有效算法。这种解决方案主要涉及对trie中所使用的标记符号的大小进行有效控制，称为*LZW*（Lempel-Ziv-Welsh data compression）数据压缩。LZW压缩是支持图形交换格式（Graphics Interchange Format）GIF（发音为"jiff"）的一种基本算法，GIF是CompuServe公司的工程师们开发出来的，并且由www网推广普及。因为Welsh发明的算法是属于他在Sperry公司的工作职责的一部分，所以Unisys公司行使其权力为这个算法申请了专利。随后，Unisys公司就要求对每次使用GIF的网络服务的提供商或者广大的用户都收取少量的专利使用费。LZW不是专门为GIF设计的，LZW同样也可以用于TIFF图形格式和其他的一些压缩程序（包括Unix Compress）以及各种类型的软件应用程序（例如Postscript和PDF）和某些硬件设备（其中最有名的设备就是调制解调器，Modem）中。所以，Unisys公司在web社区内没有很好地收到他们所要求的专利使用费，这一点并不会令人感到意外，因为一些网站公开宣布联合起来永久抵制GIF。Cooler公司率先通过创造一些更好（至少是不同的）的算法，已经巧妙地避开了GIF这个问题。其中一种算法就是PNG，可移植网络图形格式（Portable Network Graphics）。

如果只是有关GIF专利使用费的争论并不会导致PNG（发音为"ping"）的出现，但是这种争论这无疑加速了PNG的开发进程。在1995年的短短几个月时间内，PNG就从一种草案变成了大家所接受的国际标准。但是令人感到惊讶的是，到2002年为止，PNG的规范只有两个较低级别的版本。

PNG对GIF做出了许多方面的改进，其中包括：

- 用户可选择的压缩模式：在0到3范围内的"快速压缩"或者"更佳压缩"。
- 比GIF有更高的压缩比，通常情况下可以改善压缩比5%到25%。
- 提供一个32位的CRC（ISO 3309/ITU-142）错误检测功能。
- 在顺序显示模式中具有较快的初始化显示。
- 一个公开的国际标准，可以免费使用，并且得到了www协会（W3C）和许多其他的组织和商业机构的支持。

PNG采用二级压缩方式：首先，使用赫夫曼编码简化信息。然后，在完成赫夫曼编码后再利用一个32字节的文本窗口进行LZ77压缩。

GIF具有一种PNG所没有的功能：GIF能够支持同一个文件中的多个图像，这样可以产生动画效果（虽然比较呆板）。为了克服这种限制，互连网协会提出了多图像网络图形（Multiple-imagine Network Graphics，或MNG，发音为"ming"）算法。MNG是PNG的扩展，它允许把多个图像压缩到一个文件中。这些文件可以是任意类型的文件，比如可以是灰度图像文件，真彩色图像文件，甚至是JPEG文件（参见下一节的内容）。MNG的1.0版本在2001年1月发布，后来不断地进行了修改和功能方面的增强。PNG和MNG压缩工具在因特网上都可以免费获取（包括源代码）。因此，我们可以合理地认为摒弃GIF压缩方法将只是一个时间问题。

7.8.4 JPEG压缩

在我们欣赏一个图形图像时，例如打印出来或在计算机屏幕上显示的一张照片，真正看到的是许多像素（pixel，或picture element）的圆点组成的集合。这些像素点在图像质量差的介质，例如报纸或连环画中，看起来特别的明显。当像素点很小，而且紧密聚集在一起时，我们的眼睛就会感觉到一个高品质的图像。作为一种主观的量度标准，"高品质"的定义是从大约300像素/英寸（或者120像素/厘米）开始的。而在高端应用中，大多数人都同意一个1600像素/英寸（或者640像素/厘米）的图像，如果不能被认为是一幅最佳的图像，也应该是一幅好图像。

像素包含图像的二进制编码，显示器和打印机的硬件能够对这种形式的编码进行译码解释。可以采用任意数目的二进制位对图形的像素进行编码。例如，如果我们要画一幅黑白的素描画，那么可以

采用每个像素占用一位的编码方式。这个像素的位要么是黑色的（像素=0），要么是白色的（像素=1）。如果我们决定要画一个有灰度的图像，那就需要考虑采用多少种色调的灰度才能满足图像的要求。如果要求 8 种色调的灰度，那么每个像素就需要使用 3 个二进制位。000 代表黑色，111 代表白色。在 000 和 111 之间的任何数都代表某种色调的灰度等级。

彩色像素是由红、绿、蓝三种颜色组成。如果想采用 8 种不同色调的红色、绿色和蓝色对一个图像着色，那么每种颜色成份（each color component）都需要使用 3 位二进制数。因此，每个像素就需用 9 位，可以产生 2^9-1 种不同的颜色。黑色仍旧采用全 0 来表示，即：R=000，G=000，B=000；白色也仍旧采用全 1 来表示，即：R=111，G=111，B=111。“纯”绿色为 R=000，G=111，B=000；R=011，G=000，B=101 则给出了某种色调的紫色；R=111，G=111，B=000 就产生了黄色。每种颜色所使用的位数越多，就越接近在我们周围所看到的“真彩色”。大多数的计算机系统对每种颜色都使用 8 位二进制数，可以表现 256 种不同的色调，所以构建的色彩都近似于真彩色。24 位的像素可以显示大约 1600 万种不同的颜色。

如果我们希望以某种方式存储一个 4×6 英寸（10×15 厘米）的照片图像，并且希望在以后打印或预览这个图像时，可以获得较好的图像质量。如果按照 300 像素/英寸，每个像素占用 24 位（3 个字节）的图像质量，存储这样的一幅图像就需要 300×300×6×4×3=6.48MB 存储空间。这张 4×6 英寸的照片可能只是张贴在网络上的某个销售广告的一部分内容。可以想像，如果一些使用调制解调器拨号上网的客户用了 20 多分钟的时间还没有完成广告页下载，那么我们将很可能会失去这些客户。如果使用每英寸 1600 像素的图像精度，那么存储空间将需要近 1.5GB，这意味着要下载和存储这些图像文件实际上是不可能的。

JPEG 就是一个专门设计用来解决这个问题的压缩算法。值得庆幸的是，一般照片图像包含大量的冗余信息。而且，某些理论上具有高熵值（信息量）的信息常常对图像的完整性并不重要。考虑到这些因素，ISO 和 ITU 专门联合组成了一个图像专家联合组（Joint Photographic Experts Group，或者简称 JPEG，读音为“jay-peg”）来制定一个国际图像压缩的标准。第一个 JPEG 标准，即 10928-1，于 1992 年完成发布。1997 年开始对这个标准进行重大修改和扩充。现在，新的标准称为 JPEG2000，于 2000 年 12 月定稿完成。

JPEG 是一组算法的集合，它能够提供极好的压缩效果，其代价是会损失一些图像信息。直到目前为止，我们所介绍的都是*无损失的数据压缩*（lossless data compression）。经过压缩后存储的数据与压缩前的数据是完全一样的，除非由于计算或介质错误带来的损失。有时，如果可以允许少量信息损失的话，那么可以实现好得多的图像压缩效果。照相图像特别适合于*有损失的数据压缩*（lossy data compression），原因是人类的眼睛具有补偿校正图形图像中出现少许失真的能力。当然，一些图像携带真实的信息，只有在对图像的“质量”进行仔细的界定之后，才可以对图像进行有损失的数据压缩。医学诊断图像，例如 X 光照片和心电图就是此类的图像。但是，家庭影集和销售小册子上的照片，都是一些允许损失大量“信息”的图像。即使这些信息丢失后，人们仍旧可以保留它们对视觉“质量”的幻觉。

JPEG 最突出的一个特点就是，用户可以通过在压缩图像之前提供有关的参数来控制信息损失的数量。即使是要求有 100%的图像保真度，JPEG 也能够产生非常明显的压缩效果。在图像保真度为 75%时，JEPG 压缩过程所损失的信息几乎是难以察觉的，而且压缩后的图像文件尺寸也只有原始图像文件大小的一小部分。图 7-30 展示了采用不同的质量参数压缩一个灰度图像后的效果图。图像压缩前的原始文件为一个 7.14KB 大小的位图，这个位图用作图中各个质量参数条件下的输入文件。

正如大家所看到的，只有当使用最低的质量因子时，JPEG 压缩所造成的信息损失才成为明显的问题。大家同样也会注意到，在最高压缩条件下，图像会呈现类似于纵横拼字迷的模糊外观。如果理解了 JPEG 的工作原理，那么造成上面这种情况的原因就会变得非常清楚。

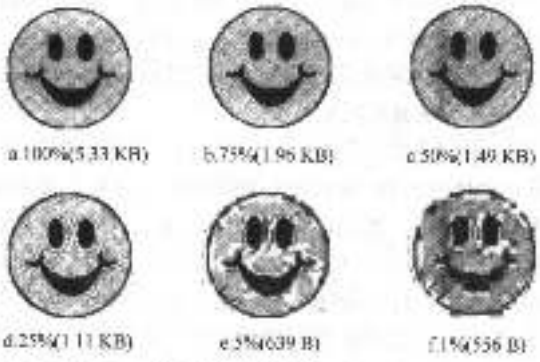


图 7-30 对于一个 7.14KB 的位图文件使用不同量化的 JPEG 压缩效果

在对彩色图像进行压缩时，JPEG 首先要将 RGB 的成分转化成亮度（luminance）和色度（chrominance）两部分，亮度是色彩的明亮程度，而色度是色彩本身的鲜艳程度。人们的眼睛对色度没有对亮度敏感。故此，在构建结果编码时，亮度部分在随后的各个压缩步骤中应该尽量不要有信息丢失，而灰度图像就不要求有这样的措施。

接下来，将图像划分成每边都是 8 个像素的许多正方形块，然后，利用一个如下所示离散的余弦变换（DCT），将这些 64 像素方块从空间域（x，y）转换到频率域（x，y）：

$$\mathrm{DCT}(i,j)=\frac{1}{4}\mathrm{C}(i)\times\mathrm{C}(j)\sum_{x=0}^{7}\sum_{y=0}^{7}\mathrm{pixel}(x,y)\times\cos\left[\frac{(2x+1)i\pi}{16}\right]\times\cos\left[\frac{(2y+1)j\pi}{16}\right]$$

其中

$$\mathrm{C}(a)=\begin{cases}\dfrac{1}{\sqrt{2}} & \text{如果 } a=0\\ 1 & \text{否则}\end{cases}$$

这种转换的输出结果是一个 8×8 整型矩阵，变化范围从 −1024 到 1023。其中，位于 $x=0$，$y=0$ 处的像素称为 *DC* 系数（DC coefficient），而且这个系数包含一个原始方块中的 64 像素的加权平均值。其他的 63 个值就称为 *AC* 系数（AC coefficient）。由于余弦函数的特性（cos0=1），变换结果的频率矩阵，即（i，j）矩阵在右下角聚集的是一些小数和 0，而大数集中在矩阵左上角位置。这种矩阵的模式非常有利于多种不同的压缩方法，但是目前还不准备介绍这些内容。

在对频率矩阵进行压缩之前，首先将矩阵中的每个值除以一个量化矩阵（quantization matrix）中对应的元素，量化过程的目的是把 11 位的 DCT 输出减少到一个 8 位值。在 JPEG 中，这个步骤会造成信息损失，这种损失的程度可以由用户选择。JPEG 规范提供了几个量化矩阵，用户可以根据自己的判断使用其中的任何一个矩阵。所有的这些标准矩阵都保证频率矩阵中的元素包含最多的信息（指朝向左上角的元素），而且保证在量化过程中损失尽可能少的信息量。

经过量化步骤后，频率矩阵变成了一个稀疏（sparse）矩阵，该矩阵包含 0 的元素要多于非 0 的元素，这些 0 元素都位于矩阵的右下角。因此，使用运行长度编码（run length coding）方式可以方便地对具有相同值的大数据块进行压缩。

运行长度编码是一种非常简单的压缩方法。运行长度编码不是采用 XXXXX 的编码方式，而是编码为（5，X），指示出运行 5 个 X 的长度。如果存储的是（5，X），而不是 XXXXX，那么将会节省 3 个字节，但是并不包括这种方法可能要求的任何定界符。很明显，要使用这种编码最有效的方法是尽量使每个事件都对齐，以便于可以得到尽可能多的相连的 0。通过对频率矩阵进行之字形（zig-zag）扫描完成了 JPEG 这种功能。这个扫描步骤的结果是通常包含一个长串 0 的一维矩阵（矢量）。图 7-31 说明了之字形扫描的工作原理。

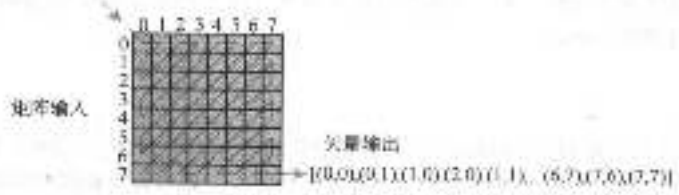


图 7-31 一个 JPEG 数字矩阵的之字形扫描过程

使用运行长度编码方式对这个矢量中的每个 AC 系数进行压缩。如果有 DC 系数，那么就要将 DC 系数编码为 DC 系数的原始值和前面方块中的 DC 系数之间的算术差。

然后，可以使用霍夫曼编码方法或者算术编码方法，对运行长度编码后的矢量结果做进一步的压缩。霍夫曼编码是首选的方法，因为算术算法有大量的专利限制。

图 7-32 总结了上面所描述的 JPEG 算法的各个步骤。显然，要实现解压算法只需将压缩过程的各个步骤反转过来。

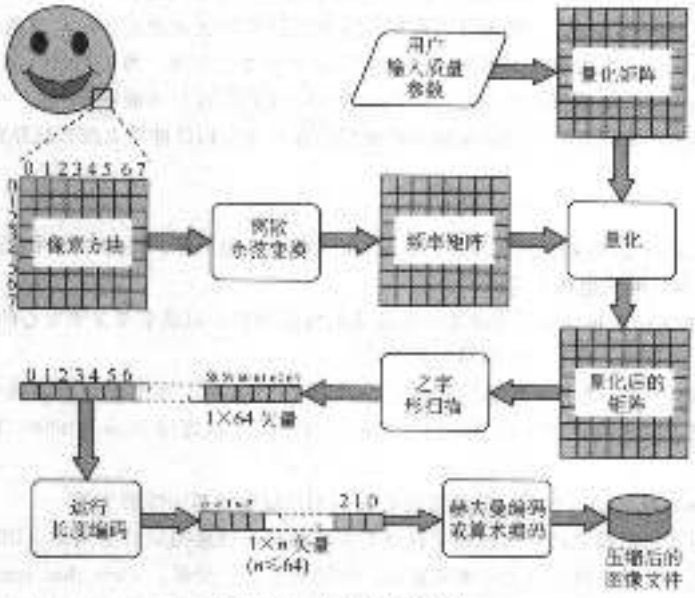


图 7-32 JPEG 压缩算法

JPEG 2000 对 1997 年颁布的 JPEG 标准做了许多的改进。JPEG 2000 的基础算法比以前更加完善和复杂，而且在量化参数和将多个图像合并到同一个 JPEG 文件中的处理方面具有了更大的灵活性。JPEG 2000 最显著的特点之一是允许用户定义兴趣区域（regions of interest）。兴趣区域就是图像中某个用户所关注的重点范围，并且兴趣区域内的图像不会受到和图像其他部分同等程度的有损压缩。例如，假设我们拍摄一张某个朋友站在湖边的照片，我们可能会告诉 JPEG 2000，照片中的这个朋友才是我们的兴趣区域。因此，湖的背景和周围的树木会被大幅度地压缩，而损失一些清晰度。然而，这个朋友的图像将被清晰地保留下来。这样一来，如果不对图像进行增强处理的话，图像的背景质量会比较差。

JPEG 2000 采用了一个小波变换来代替 JPEG 中的离散余弦变换。小波变换使用不同的方式对图像或信号进行采样和编码。JPEG 2000 的量化过程使用的是一个正弦函数，而不是早期版本中所采用的简单除法。这些更加复杂的算术处理要求比 JPEG 占用更多的处理器资源，这将会造成计算机系统性能的显著下降。在解决这个性能问题之前，JPEG 2000 只适用于需要特色处理而不要求增加计算能

力所产生的成本问题的地方。换言之，如果计算能力方面的成本降低已经变得与迟缓增加的计算机性能无关时，可以使用 JPEG 2000。

本章小结

本章比较全面地评述了有关计算机输入/输出和存储系统诸多方面的内容。读者可以了解不同类型的机器需要使用不同的 I/O 结构。从本质上来说，大型计算机系统中存储和访问数据的方式与小型计算机上所使用的方法是完全不同的。

本章阐述了如何将数据存储到各种不同的介质上，例如磁带、磁盘和光盘等。如果理解了磁盘工作的原理，那么对于编程、系统设计或者是故障诊断过程中分析磁盘性能是非常有帮助的。

本章中有关 RAID 系统的讨论，将有助于我们理解 RAID 是如何来改进系统的性能和增加系统的可用性。

同时，读者也学习了几种数据压缩的方法。数据压缩可以帮助我们节省磁盘空间和磁带的使用，并且还能够减少数据通信时的传输时间。了解这些压缩方法的细节过程可以帮助我们为特定的应用程序选择最好的压缩方法。而有关信息理论的简要介绍则是为我们今后在计算机科学领域的进一步学习做一些准备工作。

本章希望通过这些内容的讨论，读者可以在做出任何实际的系统决定前，对系统的性能有一个综合评价。读者可以看到，我们常常需要考虑如何在“更好”与“更快”之间，和“更快”与“更便宜”之间做出适当的选择。如果作为系统项目的负责人，我们必须确信我们的客户也能够和我们一样地理解这种性能之间的权衡关系。通常，我们需要非常有策略地使我们的客户完全相信世界上没有这样免费的午餐。

深入阅读

阅读 Amdahl（1967）的原始文献可以对 Amdah 定律有更多的了解。Hennesey 和 Patterson（1996）还给出了有关 Amdahl 定律的其他适用范围。

Rosch（1997）的著作中包含大量与本章主题相关的细节内容，但是本书主要关心的还是小型计算机系统。本书的内容组织得很好，而且写作风格清楚易读。

Rosch（1997）的著作还为 CD 存储技术做了一个很好的概述。对于更完整的内容介绍，包括 CD-ROM’S 物理的基础，数学的背景和电子工程方面的支撑等，可以阅读 Stan（1998）和 Williams（1994）的著作。

Patterson、Gibson 和 Katz（1988）的论文是有关 RAID 结构的基础性的文章。

到目前为止，IBM 公司设立的众多网站是提供有关详细技术信息的最好的网站。IBM 公司独自在网络上提供了大量的优秀文献以方便所有的搜索者获取。IBM 公司的主页是：www. ibm. com。除了他们的服务器产品热线 www. ibm. com/eservers 外，IBM 公司还设有许多特殊兴趣领域的专门网站，包括有关存储系统的网站 www. storage. ibm. com。IBM 公司的研发网页 www. research. ibm. com，包含大量与新技术发展相关的最新信息。高质量学术期刊可以通过网址 www. research. ibm. com/journal 查找。

关于数据压缩方面的内容并不缺乏优秀的著作和文献。Lelewer 和 Hirschberg（1987）论文是一篇经常被人们引用的理论综述。在 Nelson 和 Gailly（1996）著作中可以找到一种更加详细的处理方法，包括源代码。Nelson 和 Gailly 他们那种清晰明了和随意的写作方式，会使得读者学习神秘的数据压缩艺术的过程变成了一种真正愉快的经历。也可以在网上查找大量关于数据压缩的信息。只要查找任何一个本章中引进的有关数据压缩的关键项目，利用任意一个好搜索引擎都可以找到成百上千个网络链接。

在数据压缩和数据通信领域，小波（Wavelet）理论变得越来越重要。如果读者想在这一领域钻研发展的话，可以从阅读 Vetterli 和 Kovacěvić（1995）的著作开始。该书还介绍了大量有关图像压缩方面的内容，当然包括 JPEG 和 JPEG 2000 中应用的小波理论。

在本章结尾处的特别“专题（Focus On）”部分，讨论了许多 I/O 结构，其中包括光纤信道、SANS 和 HIPPI 等内容。在书籍方面，有关描写光纤信道或 SANS 的著作为数不多，较难找到。Clark（1999）和 Thornburgh（1999）的著作都对这一主题进行了很好的讨论。国家信息技术标准委员会（NCITS——前身

为通用标准委员会 X3，信息技术）产业协调小组，设立了一个综合性的网站 www. t11. org。在这个网站上可以找到最新的 SCSI-3 标准的草稿。而 HIPPI 规范可以在网站 www. hippi. org 上找到。

Rosch（1997）的著作中还包含大量关于 SCSI 与其他总线结构的信息，以及如何在小型计算机系统中实现这些总线结构。

参考文献

Amdahl, George M. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities." *Proceedings of AFIPS 1967 Spring Joint Computer Conference.* Vol. 30 (Atlantic City, NJ, April 1967), pp. 483–485.

Clark, Tom. *Designing Storage Area Networks: A Practical Guide for Implementing Fibre Channel SANS.* Reading, MA: Addison-Wesley Longman, 1999.

Hennessey, John L., & Patterson, David A. *Computer Architecture: A Quantitative Approach.* San Francisco, CA: Morgan Kaufmann Publishers, 1996.

Lelewer, Debra A., & Hirschberg, Daniel S. "Data Compression." *ACM Computing Surveys 19*:3, 1987, pp. 261–297.

Lesser, M. L., & Haanstra, J. W. "The Random Access Memory Accounting Machine: I. System Organization of the IBM 305." *IBM Journal of Research and Development 1*:1. January 1957. Reprinted in Vol. 44, No. 1/2, January/March 2000, pp. 6–15.

Nelson, Mark, & Gailly, Jean-Loup. *The Data Compression Book*, 2nd ed., New York: M&T Books, 1996.

Noyes, T., & Dickinson, W. E. "The Random Access Memory Accounting Machine: II. System Organization of the IBM 305." *IBM Journal of Research and Development 1*:1. January 1957. Reprinted in Vol. 44, No. 1/2, January/March 2000, pp. 16–19.

Patterson, David A., Gibson, Garth, & Katz, Randy. "A Case for Redundant Arrays of Inexpensive Disks (RAID)." *Proceedings of the ACM SIGMOD Conference on Management of Data,* June 1988, pp. 109–116.

Rosch, Winn L. *The Winn L. Rosch Hardware Bible.* Indianapolis: Sams Publishing, 1997.

Stan, Sorin G. *The CD-ROM Drive: A Brief System Description.* Boston: Kluwer Academic Publishers, 1998.

Thornburgh, Ralph H. *Fibre Channel for Mass Storage.* (Hewlett-Packard Professional Books series.) Upper Saddle River, NJ: Prentice Hall PTR, 1999.

Vetterli, Martin, & Kovačević, Jelena. *Wavelets and Subband Coding.* Englewood Cliffs, NJ: Prentice Hall PTR, 1995.

Welsh, Terry. "A Technique for High-Performance Data Compression." *IEEE Computer 17*:6, June 1984, pp. 8–19.

Williams, E. W. *The CD-ROM and Optical Recording Systems.* New York: Oxford University Press, 1994.

Ziv, J., & Lempel, A. "A Universal Algorithm for Sequential Data Compression." *IEEE Transactions on Information Theory 23*:3, May 1977, pp. 337–343.

Ziv, J., & Lempel, A. "Compression of Individual Sequences via Variable-Rate Coding." *IEEE Transactions on Information Theory 24*:5, September 1978, pp. 530–536.

基本概念和术语复习

1. 用文字表述 Amdahl 定律。
2. 什么是加速率？
3. 什么是协议？为什么协议在 I/O 总线技术中非常重要？
4. 列举三种不同类型的持久性存储器。

5. 解释程序控制的 I/O 与中断控制的 I/O 之间有什么不同。
6. 什么是轮流检测（或轮询法）?
7. 在中断控制的 I/O 中，如何使用地址向量?
8. 直接存储器访问（DMA）的工作原理是什么?
9. 什么是总线主控?
10. 为什么 DMA 需要周期窃取?
11. 当有人将 I/O 看作为突发式，它表示的是什么意思?
12. 通道控制的 I/O 与中断控制的 I/O 有什么不同?
13. 通道控制的 I/O 与 DMA 有何相似之处?
14. 什么是多路复用技术?
15. 同步总线和异步总线的区别是什么?
16. 什么是稳定时间，对稳定时间可以做些什么?
17. 为什么磁盘被称为直接存储设备?
18. 解释磁盘底板、磁道、扇区和区块之间的关系。
19. 组成硬盘驱动器的主要物理部件有哪些?
20. 什么是分区位记录法?
21. 什么是寻道时间?
22. 旋转延时和寻道时间的总和叫做什么?
23. 什么是文件分配表（FAT），它位于软盘中哪个位置?
24. 硬盘的转速比软盘的转速快多少个数量级?
25. 自动光盘库设备的名称是什么?
26. 直接写到光学介质上而不是写到纸带或微缩胶片上的计算机输出首字母的缩写词是什么?
27. 磁盘通过改变磁盘介质的磁极性来存储字节。那么光盘是如何存储字节的呢?
28. 存储音乐的 CD 格式与存储数据的 CD 格式有什么不同? 格式之间有什么相似之处?
29. 对于长期数据的存储来说，为什么 CD 特别的有用?
30. 存储数据的 CD 是使用记录时间方式吗?
31. DVD 如何存储比 CD 多得多的数据?
32. 写出记录 WORM 磁盘的三种方法?
33. 为什么磁带是一种非常流行的存储介质?
34. 解释蛇形线式记录和螺旋线扫描记录有什么不同之处?
35. 使用蛇形线式记录的两种流行格式是什么?
36. 哪一种 RAID 技术能够提供最好的性能?
37. 哪一种 RAID 技术最经济，而又可以提供足够的冗余量?
38. 哪一种 RAID 技术使用镜像磁盘组?
39. 什么是混合 RAID 系统?
40. 科学信息理论的奠基人是谁?
41. 什么是信息熵? 信息熵如何与信息冗余相关联?
42. 说明统计编码的优点和缺点?
43. 列出两种统计编码的类型。
44. LZ77 压缩算法属于哪一种数据压缩算法?

练习题

◆1. 你的朋友刚买了一台新的个人计算机。她告诉你她的新计算机系统运行速度是 1GHZ，这是否意味着她的新系统的速度要比她那台 300MHz 的老机器要快 3 倍多呢? 请解释原因?

2. 假定一台计算机的日常工作量由 60%CPU 活动率和 40%磁盘活动率所组成。客户正在抱怨计算机系统太慢了。在经过一番研究以后，你发现可以用 8 000 美元来升级磁盘，升级后的磁盘速度比目前正在使用的磁盘快 2.5 倍。同样。也可以用 5 000 美元来对 CPU 进行升级，升级后 CPU 的速度比现在的 CPU 快 1.4 倍。

a) 如果想用最少的钱，获得最好的系统性能改善，你会选择哪一种方案？

b) 如果你并不在乎钱的因素，只要求系统的速度更快些，那你又会选择哪一种建议呢？

c) 对于上述的系统升级，收支平衡点是什么？也就是说，对于 CPU 升级方案（或磁盘，只改变其中之一），什么样的价格可以使得对于这二者 1%的增长，最后的成本是相同的？

◆3. 在第 2 题中，如果系统的活动率变为由 55%的处理器时间和 45%的磁盘活动所组成，那么上面问题的答案又是什么？

4. 写出 4 种类型的 I/O 结构。并说明这些结构通常情况下在什么地方使用？为什么？

5. 一个带有中断控制的 I/O 的 CPU 正在忙于服务某个磁盘请求。如果 CPU 正在执行磁盘服务例行程序的中途，此时又有另外一个 I/O 中断将要发生。请问：

◆a) 接下来会发生什么情况？

◆b) 这是一个问题吗？

◆c) 如果不是一个问题，为什么？如果是一个问题，那么应该做些什么？

6. 为什么要为 I/O 总线提供时钟信号？

7. 如果需要使用一条地址总线能够对 8 台设备进行编址，那么需要多少根导线？如果每一台设备都要求能够对 I/O 控制设备做出应答，那又会是什么情况？

8. 本章我们曾指出 I/O 总线并不一定要求有独立的地址总线。画出一个类似于图 7-7 时序图，对于一个写操作描述在 I/O 控制器和磁盘控制器之间的握手协议过程（提示：需要增加一个控制信号）。

*9. 如果图 7-7 中所显示的时间间隔为 50 纳秒，那么传输 10 个字节的数据需要多长时间？请构思设计一个总线协议，可以减少上面的传输过程发生所需要的时间，控制线的数目可以根据需要而定。如果去掉地址线，而是使用数据线来代替地址线，那又会发生什么样的情况？（提示：需要一个附加的控制线。）

10. 写出寻道时间、旋转延时和传输时间的定义。并解释它们之间的相互关系。

◆11. 你认为将磁盘驱动器说成随机访问设备是用词不当吗？为什么？

12. 为什么不同的系统要把磁盘目录放置在磁盘上不同的磁道位置？说明使用各自的目录区域有什么优点？

◆13. 验证图 7-11 的磁盘规范中的平均反应速率。在这个计算中为什么要除以 2？

14. 通过仔细检查图 7-11 中的磁盘规范，你认为这个磁盘驱动器是否使用分区位记录方式？

15. 图 7-11 的磁盘规范中给出了从磁盘读取数据时的数据传输率为 6.0MB，而向磁盘写入数据时的数据传输率为 11.1MB。为什么这两个数据会产生不同？

16. 你是否相信磁盘驱动器的 MTRR 数字？请解释原因。

17. 假定磁盘驱动器具有如下特性：

- 有 4 个记录面
- 每个面有 1 024 个磁道
- 每个磁道有 128 个扇区
- 512 字节/扇区
- 磁道对磁道的寻道时间为 5 毫秒
- 旋转速率为 5 000RPM

◆a) 驱动器的容量是多少？

◆b) 访问时间是多少？

18. 假定磁盘驱动器具有如下特性：

- 5 个记录面
- 每个面有 1 024 个磁道

- 每个磁道有256个扇区
- 512字节/扇区
- 磁道对磁道的寻道时间为8毫秒
- 旋转速率为7 500RPM

a) 磁盘驱动器的容量是多少？

b) 访问时间是多少？

c) 这个磁盘是否要比第17题中描述的磁盘快？请解释原因。

19. 每个磁盘区块只有少量扇区的优点和缺点分别是什么？

* 20. 给出几种改进1.44MB软盘性能方法的建议。

21. 在一张1.44MB软盘上，能容纳的最大根目录条目的数目是多少？为什么？

22. 光盘的组织结构与磁盘的组织结构有哪些方面的不同？

23. 讨论DLT和DAT记录数据方式之间的区别。并说明为什么你会认为其中一种方式比另一种更好些？

24. 光学文档存储系统的纠错要求与以文本形式存储相同信息的纠错要求有什么不同？对于光学存储设备来说，使用不同的纠错级别有何优点？

25. 如果需要存档大量的数据，你正在决定是要使用磁带存储方法还是光学存储方法。这种数据有什么特性？并说明为什么会做出这样的选择？

* 26. 某Web电子商务服务器使用了一台特殊的高性能计算机系统。这台系统每小时可以完成总共10 000美元的交易量。估计的纯利润是每小时1 200美元。换句话说，如果系统崩溃了，那么该公司每小时将损失1 200美元，直到系统修复为止。而且，已经损坏的磁盘上的任何数据都会丢失。其中有些数据可以在昨晚备份的数据中重新找到，而其余的数据将会永远地丢失。可以想像，一个难以控制的磁盘损坏可能导致公司几十万美元的直接收入损失，以及数不清的永久性商业损失。事实上，困扰你的问题是这个系统目前没有使用任何类型的RAID系统。

也许你主要关心的问题是数据完整性和系统的可用性，然而小组中的其他一些成员可能更关注系统的性能。他们觉得如果安装了RAID，系统的速度变慢，从长远的观点来看，可能会造成更多的收入损失。他们特别指出，如果带有RAID的系统的运行速度只有当前系统的一半，将会导致每小时的总收入下降到只有每小时5 000美元。

80%的系统电子商务活动会涉及到数据库的事务处理。数据库的事务处理由60%的读操作和40%的写操作组成。平均说来，磁盘的访问时间是20毫秒。

现在，该系统的磁盘空间已经基本用完，而且所有磁盘的预期寿命也接近结束，所以马上必须预定新的磁盘来代替。于是你会觉得这是一个安装RAID的好时机，当然还需购买一些额外的磁盘。你找到一种适合于系统的磁盘，每一个容量10GB的磁盘组的价格为2 000美元。新磁盘的平均访问时间是15毫秒，MTTF为20 000小时而MTTR为4小时。你计划需要有60GB的存储容量来保存现有的数据和预期随后5年内可能增加的数据。将要替换掉原来的所有磁盘。

a) 对于反对在系统中增加RAID的人，他们有关磁盘速度减慢50%将会导致收入下降到每小时5 000美元的断言是否正确？并证明你的答案。

b) 如果你决定采用RAID-1，那么在新系统中的平均磁盘访问时间是多少？

c) 如果系统使用一个带有两组4磁盘的RAID-5阵列，而且25%的数据库事务处理必须等待某个事务处理完毕直到该磁盘空闲，那么平均磁盘访问时间又为多少？

d) 对于RAID-1和RAID-5，哪一种配置的成本费用更加合理一些？并解释理由。

27. a) 在本章所介绍的RAID系统中，哪一种系统不允许有单一磁盘的失效？

b) 哪一种系统可以允许一个以上的磁盘同时失效？

28. 计算图7-30中每个JPEG图像的压缩系数。

29. 对于在第7.8.2节中使用的“Star Bright”童谣，生成一个赫夫曼树并分配赫夫曼代码。其中，空格使用符号<ws>而不是下划线。

30. 完成第7.8.2节中说明的LZ77数据压缩的全过程。

31. 使用 JPEG 2000 来压缩素描图像是一个不太明智的选择，正如图 7-30 所示。为什么？你可否建议采用其他的压缩方法？说明选择这种方法的理由？
32. a）列举 LZ77 胜过赫夫曼编码的一种优点。
    b）列举赫夫曼编码胜过 LZ77 的一种优点。
    c）哪一种压缩方法更好一些？
33. 陈述 PNG 的一种特性，并且要能够用它来说服别人 PNG 是一种比 GIF 更好的算法？



选择磁盘存储器的实现专题

7A.1 概述

本部分的内容将简要介绍一些的重要的I/O系统，显然读者在今后的职业生涯中可能会遇到这些I/O系统。如果对这些系统有一个综合的理解，无疑有助于我们做出正确的决定，在不同的应用中选择最好的方法。最重要的是，我们将了解现代存储系统正在以它们特有的方式成为独立的系统，这些存储系统的体系结构模型与一台计算机主机系统的内部结构有很大的区别。在讨论复杂的体系结构之前，首先介绍有关的数据传输模式。

7A.2 数据传输模式

信息字节可以在一个主机和一个外围设备之间进行传输。如果在传输过程中每次发送一位数据称为串行（serial）通信方式，而每次发送一个字节称为并行（parallel）通信方式。每种传输模式都会在主机和设备接口之间建立一种特定的通信协议。在接下来的章节中，我们将讨论在存储系统中使用的几种重要协议。这里的思想许多都会延伸到数据通信的领域（参见第11章）。

7A.2.1 并行数据传输

并行通信系统的工作方式与一个主机的存储器总线的操作非常类似。至少要求8根数据线（每一位数据对应于一根数据线），而且还需要一根线来进行同步。这根同步线有时也称为选通线（strobe）。

并行连接对于短距离传输非常有效，通常短于30英尺，具体长度主要取决于信号的强度、信号的频率和电缆的质量。在长距离的数据传输时，电缆中的信号会因为导体的内部电阻而不断减弱。这种电信号随着距离和时间的增加而损失的现象称为衰减（attenuation）。通过学习一个例子后，这种与衰减相关的问题就会变得更加清楚。

图7A-1给出了一个并行打印机接口的简化时序图。图中标注为nStrobe和nAck的两条线分别用来发送选通信号和应答信号，加载低电压有效。当施加高电压时，Busy和Data信号送出。换言之，Busy和Data信号属于正逻辑信号，而nStrobe和nAck信号都是负逻辑信号。在时序图的顶部，列出了从t_0到t_6之间的任意参考时间。两个连续参考时间之差Δt决定总线的速度。通常情况下，Δt的范围在1毫秒和5毫秒之间。

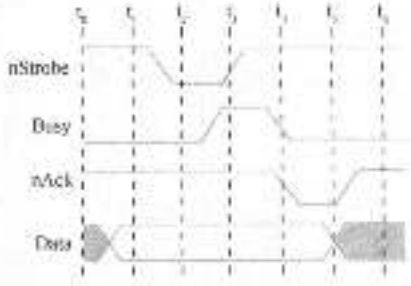


图7A-1 一台并行打印机的简化时序图

数据信号表示8根不同的线。每根数据线或者是高电平或者是低电平（信号1或信号0）。这些数据线上的信号在nStrobe信号发出之前和在nAck信号发出之后都是没有意义的（图中阴影部分）。

图7A-1说明了在一个打印机接口电路（位于主机上）和一台并行打印机的主机接口之间所发生的握手协议。当在8个数据线的每根线路上放置1位数据时，这个握手的过程就开始了。接下来，就是检测busy信号线是否处于低电平。一旦检测到busy信号线为低电平，则有效送出strobe信号，这样打印机就知道了数据线上有数据送入。只要打印机检测到strobe信号，它就会读取数据线。同时，将busy信号升起来变为高电平，以防止主机向数据线上放置更多的数据。在打印机已经读取数据线上的数据之后，busy信号变成低电平。同时发出一个应答信号nAck，让主机知道已接收数据。

值得注意的是，尽管已经确认数据信号，但是这并不能

保证这些数据的正确性。主机和打印机都假设接收到的信号和发送的信号是完全一样的。对于短距离传输，这样的假设是相当安全的。但是对于长距离传输来说，情况并非如此。

现在假定总线操作的电平为+5V 或 -5V。因此，0 和+5V 之间的任何电压都会被认为是“高”电平，而 0 和-5V 之间的任何电压都会被认为是“低”电平。主机在不同的数据线上设置+5V 电压或-5V 电压，分别对应于数据字节中不同位的数据 1 和 0。随后，主机会设置 strobe 线为-5V 电压。

在伴随有“轻度（mild）”衰减的情况下，打印机检测 nStrobe 信号或者是主机去检测 nAck 信号时的速度可能会减慢。如果连接有多个打印机时，这种速度上的滞阻现象几乎很难被察觉到。但是，在我们通常希望能够立即响应的并行磁盘接口上，这种速度上的迟缓是令人可怕的。

如果采用一根很长的传输电缆，在打印机的终端上可能会收到完全不同的电压信号。当信号到达打印机时，“高”电平可能是+1V，而“低”电平可能是-3V。如果+1V 电压并不足以高于逻辑 1 的门限电压，那么在打印机的信号处理过程中，我们就会在应该输出一个 1 的地方得到一个 0 输出的结果。同样，经过了长距离的传输，Strobe 信号可能会在数据位信号之前到达打印机。这种情形下，打印机就会对检测到 nStrobe 信号时放置在数据线上各种信号进行打印。这时，可能发生的极端情况是一个文本字符被误认为一个控制字符。这样就可能引起打印机输出乱码和出现死机的情况。

7A.2.2 串行数据传输

从上面我们可以看到，并行数据传输是沿着数据总线每次移动一个信息字节。在并行数据传输中，每一位数据都要求有一根数据线，并且这些数据线由一根独立的 strobe 线的脉冲来激活。与并行数据传输不同，串行数据传输只用一根导线来发送数据，每次只能传送 1 位数据，就像单数据线上的脉冲信号。当然，串行数据传输还需要有其他的一些导线来传送协议所定义的特殊信号。RS-232-C 就是这样的一个串行通信协议，它要求有独立的信号线。但是，它的数据发送仍然只用一条数据线（参见第 11 章）。串行存储器的接口会将这些特殊的信号加入到沿着数据通路进行交换的协议帧中。本节后面将仔细研究一些串行存储器协议。

串行传输方法也可以被用作时间敏感的同步（isochronous）数据传输。同步协议主要用于实时数据的传送，比如语音和视频信号。因为语音和视频信号主要用于人们的感觉，所以一点偶尔的传输错误并不会造成重大的影响。这种数据的近似特性允许少量的错误控制。因此，这种从起始（源）端发送到目标端的数据流动可以具有最小的由于协议造成的延迟时间。

7A.3 SCSI

小型计算机系统接口，简称 SCSI（发音“skuzzy”），是由 Shugart Associates 和 NCR 公司在 1981 年发明的。Shugart Associates 是当时磁盘驱动器的主要制造厂商，而 NCR 也曾经在小型计算机市场中扮演关键的角色。这个接口开始时称为 SASI 接口，即 Shugart Associates Standard Interface。因为这种接口设计精良，所以在 1986 年成为了 ANSI 标准。ANSI 委员会将这个新的接口称为 SCSI，认为使用更通用的名称会比较好。

最初的标准 SCSI 接口（现在称为 SCSI-1）定义了一组命令、传输协议和若干物理连接。SCSI 规定了连接到 CPU 的驱动器的最大数目（7 个），和最高传输速率：每秒 5MB。这种接口的创新思想是将智能化的技术引进到接口中，使得 SCSI 或多或少具有一些自我管理的能力。这样，CPU 可以被解放出来从事计算工作，而不是一直忙于 I/O 接口的任务。在 20 世纪 80 年代早期，大多数小型计算机系统运行的时钟速率在 2MHz 和 8.44MHz 之间，这使得 SCSI 总线的吞吐量在当时看起来还是相当引人注目的。

如今，SCSI 已经发展到了第三代，即 SCSI-3。SCSI-3 包括多个接口标准。SCSI-3 是一种体系结构，正式称为 SCSI-3 体系结构模型（SCSI-3 Architecture Model，SAM）。SCSI-3 中定义了一个包含有层间通信协议的分层系统。这种体系结构中既包括了“标准的”并行 SCSI 接口，也包含有 3 个串行接口和一个混合接口。在第 7A.3.2 节中，我们会更详细地讨论 SAM 结构。

7A.3.1 “标准”的并行 SCSI

假设有人对你说，“我们刚刚安装了一个带有 3 个巨型 SCSI 驱动器的新 BackOffice 服务器。”或者说，“我的系统自从升级到 SCSI 后运行速度飞快”。那么说话人所提及的可能是指一个 SCSI-2 或者是一个并行 SCSI-3 的磁盘驱动器系统。在 20 世纪 80 年代，这种说法有些令人其词，原因是当时在连接和配置第一代 SCSI 设备时非常棘手。而如今，不但 SCSI 的传输速率提高了几个数量级，而且 SCSI 设备中的智能化程度更高，实际上消除了早期 SCSI 适配器中一些棘手的问题。

并行 SCSI-3 磁盘驱动器支持很宽的传输速率范围：从 10MB/s 到 80MB/s，分别对应于向下兼容早期的 SCSI-2 和最新的宽带。快速和超级的 SCSI-3 设备的实现。SCSI-3 诸多优点中的其中之一是，单一 SCSI 总线可以支持上述范围的设备速度，而无需重新走线或者换驱动器。但是对于这一点，没有人可以给予任何性能上的保证。有关 SCSI 的一些代表性特征可以参见表 7A-1。

表 7A-1 不同 SCSI 能力的概况

SCSI 名称	电缆线数目	最大理论传输速率（MB/s）	连接设备的最大数目
SCSI-1	50	5	8
快速 SCSI	50	10	8
快速和宽带	2×68	40	32
超级 SCSI	2×68 或 50 和 68	80	16

SCSI-3 并行体系结构的灵活性和强大功能可能大部分都要归咎于，SCSI 设备能够在自身之间进行通信。SCSI 设备是沿着一条总线构成菊花链基（daisy-chained）的结构，即一个设备传输出利用电缆连接到另一个设备的输入。CPU 只与其 SCSI 主机适配器进行通信，当有需要时发布各种 I/O 命令，在接下来的其他时间，CPU 会处理自己的工作任务，而由适配器来负责管理输入或输出操作。图 7A-2 为一个 SCSI-2 系统的组织结构图。

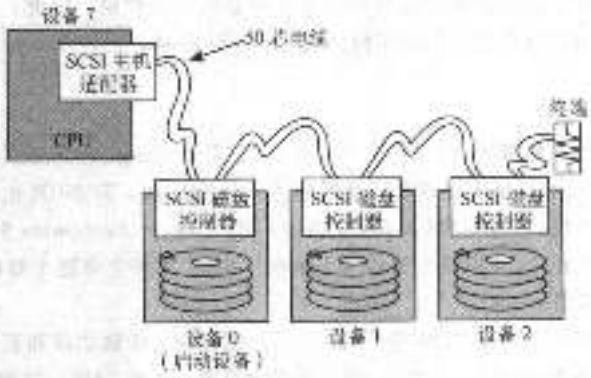


图 7A-2 SCSI-2 的配置

“快速”并行 SCSI-2 和 SCSI-3 电缆中有 50 根导线，其中的 8 根用于数据传输，11 根用作各种不同类型的控制，剩下的导线用于电学接口。在开始进行一个数据或命令传送之前，先要将设备选择（SEL）信号放到数据总线上，因为只有 8 根数据线，所以最多只能够支持 7 个设备，另外一个设备是主机适配器。“快速和宽带”SCSI 电缆有 16 位的数据总线，允许支持 14 个设备，数据传输的速率也增加到 2 倍。某些“快速和宽带”的 SCSI 系统使用 2 根 68 芯电缆，与使用 1 根 68 芯电缆的系统相比，它们能够支持设备数目和数据传输速率都增加了 1 倍。表 7A-2 说明了 50 芯导线的 SCSI 电缆引脚的功能。

表 7A-2 SCSI 的 D 类型连接器的引脚

信号	针脚编号	信号	针脚编号	信号	针脚编号
Ground	1→12	Ground	33	n ACKnowledge	34
Termination power	13	Mirror power	35	n reset	35
12V or 5V power	14	12V or 5V power	37	n MeSsaGe	46
12V or 5V (logic)	15	Ground	39、40	n SELect	47
Ground	17→26	n Attention	41	n C/D	48
Data bit 0 →Data bit 7	27→33	Synchronization	42	n REQuest	49
Parity bit	34	n BuSY	43	nI/O	50

注：信号名称的首字母 n 表示负逻辑信号，当信号为负值时，该信号断言有效。

并行 SCSI 设备彼此之间以及与主机适配器之间使用一个异步协议进行通信。通信的过程按照 8 个相来运行，每个相都定义有严格的时序。也就是说，如果某个相没有能够在规定的几毫秒内完成（具体数值取决于总线的速度），计算机会认为这个相出错，而且该协议就要从当前相的开始处重启。发送数据的设备称为启动设备（initiator），接收数据的设备称为目标（target）设备。SCSI 协议的 8 个相的具体描述如下。图 7A-3 为利用状态图来阐明这 8 个相之间的关系。

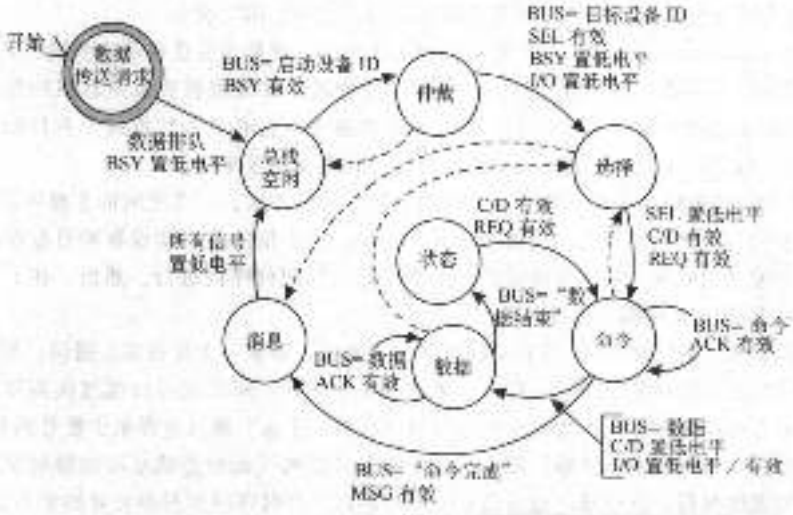


图 7A-3 并行 SCSI 的 8 相的状态图（虚线表示出错的条件）

- **总线空闲**（bus free）：启动设备首先对 "bus busy"（BSY）信号线进行查询，判断在进入下一个阶段前总线是否空闲；或者是在数据传输完成后将 BSY 信号置为低电平。
- **仲裁**（arbitration）：启动设备会将设备的 ID 放到总线上并升高 busy 信号电平，来竞争获取总线的控制权。如果同时有两个设备请求控制总线，那么具有最大 ID 值的设备将获得总线的控制权。而没有取得控制总线权的设备必须等待另一个 "总线空闲" 状态。
- **选择**（selection）：随后，启动设备将目标设备的地址放到数据总线上，升高选择（SEL）信号电平，并将 BSY 信号置为低电平。当目标设备在总线上看到自己的设备 ID，SEL 信号又为高电平，以及 BSY 和 I/O 信号为低电平时，就会将 BSY 信号电平升高，并存储启动设备的 ID 以备后用。如果发现 BSY 信号线被断言有效，启动设备也就知道了目标设备已经准备就绪，并且会通过降低 SEL 信号电平做出响应。
- **命令**（command）：一旦目标设备检测到启动设备的 SEL 信号无效，那它就会在命令/数据

(C/D) 线上发送“命令准备好”信号，指示目标设备已经准备好接收一个命令。然后，目标设备会升高 REQ 信号，请求命令信号本身。当启动设备检测到 C/D 和 REQ 信号有效后，就会把第一个命令放到数据总线上，同时发出 ACK 信号。目标设备会对传送的命令做出响应，然后升高自己的 ACK 信号对启动设备做出应答，表示该命令已经被接收。接下来，目标设备和启动设备就会利用 ACK 信号相互应答，交换命令的各个字节，直到全部的命令字节传输完毕。

在这一过程中，启动设备和目标设备会释放总线，以便在磁盘的读/写头对磁盘进行定位期间，其他的设备可以使用总线。这样一来，就允许有多个总线的并发过程。但是这显然会产生更多的成本费用，原因在于在数据被转移到启动设备之前，必须重新协商总线的控制权问题。

- **数据** (data)：在接收到完整的命令后，目标设备会降低 C/D 信号电平，将总线设置为“数据”模式。根据数据传输过程是从起始端到目标端的输出（例如，磁盘写），还是从目标端到起始端的输入（例如，磁盘读），CPU 会将输入/输出线分别置于无效或断言有效状态。然后将数据字节送到总线上，并采用与命令阶段所使用的相同的“REQ/ACK”握手协议来完成该数据字节的传送。
- **状态** (status)：一旦该数据的所有字节传输完毕，目标设备就会升高 C/D 信号电平，将总线重新置于命令方式。然后，目标设备会发送一个 REQ 信号，并等待一个来自启动设备的 ACK 信号，指示启动设备目前空闲和已经准备好接收一个命令。
- **消息** (message)：当目标设备得知启动设备已经准备就绪后，就会把“命令完成”的代码送到数据线上，然后断言“消息”线，MSG 有效。如果看到“命令完成”的消息，启动设备就将总线上的所有信号都设置为低电平，这样使总线返回到“总线空闲”状态。
- **重新选择** (reselection)：假如某个传输过程发生中断，例如当总线被释放的同时又在等待一个磁盘或磁带的服务请求，那么就要利用上述的总线仲裁阶段来重新获取对总线的控制权。如果启动设备通过自己的异或操作看到 SEL 线和 I/O 线都被断言有效和数据线上有目标设备的 ID 时，就判定它已经进行重新选择阶段。然后，协议会在数据阶段重启。

同步 SCSI 数据传输原理非常类似于刚才介绍的异步传输方法。二者之间的主要区别在于同步方法在每个数据字节的传送之间并不要求有握手协议。相反，同步传输在启动设备和目标设备之间需要协商一个最小的传输时间间隔。数据传送就在这个协商好的时间间隔内进行。然而，在下一个数据块发送之前，将会有 REQ/ACK 握手协议的情况发生。

读者很容易看到，时序对 SCSI 的有效性是非常的重要。如果一个设备发生错误，规定的等待时间上限可以防止接口电路被挂起而中止工作。如果情况并非如此，那么就可以通过从驱动器中移走软盘来阻止一个对硬盘的访问操作，因为这时的总线可能会被“永远”地（或者至少要等到系统重新启动）标记为忙的状态。在长距离电缆传输过程中所产生的信号衰减可能会造成超时的情况发生，这样会使得整个的系统的速度变慢和不可靠。相对而言，串行接口允许时序同步发生差异的能力会大得多。

7A.3.2 SCSI-3 体系结构模型

SCSI 已经从一个由协议、信号和导线组成的单片系统发展成为现在的分层接口规范，这种分层结构将物理连接与传输协议和接口命令分隔开来。这种新的规范称为 SCSI-3 体系结构模型 (SAM)。SAM 定义了这些物理分层结构，以及分层与一个称为 SCSI-3 通用访问方法 (Common Accesss Method，CAM) 的命令层主机体系结构之间是如何相互作用的。对于实际连接到计算机系统的任何类型设备，这种新规范都能够实现串行和并行 I/O 功能。各个分层通过协议服务请求、指示、响应和确认来进行相互间的通信。类似这样的松散耦合的协议堆栈（或叠层）使得我们在选择接口硬件、软件和介质时可以有较大的灵活性。并且，在某个层上面进行的技术改进不会影响到其他层的操作。对于磁盘存储系统来说，SAM 的灵活性已经开辟了 SCSI 接口在速度和适应性方面的一个新天地。

图 7A-4 表示 SAM 的各组件的连接示意图。虽然 SAM 体系结构保持了向下与 SCSI 并行协议和接口的兼容性，但是最大型的和最快速的计算机系统现在仍然使用串行方法。SAM 的各种串行协议包括串行存储

器体系结构（Serial Storage Architecture，SSA）、串行总线（Serial Bus，也称为 IEEE1394 或 FireWire）和光纤信道（Fibre Channel，FC）。尽管所有的串行协议都支持一个并行 SCSI 到串行的转换映射，但是类属包协议（Generic Packet Protocol，GPP）在处理这个问题方面却是最灵活通用的。由于 SCSI-3 总线的高速性能和系统之间相互连接的多样性，“小型计算机系统接口”中的这个“小”其实已经变成一种误用。事实上，各种各样的 SCSI 已经应用于从最小型个人计算机到最大型主机系统的所有设备中。

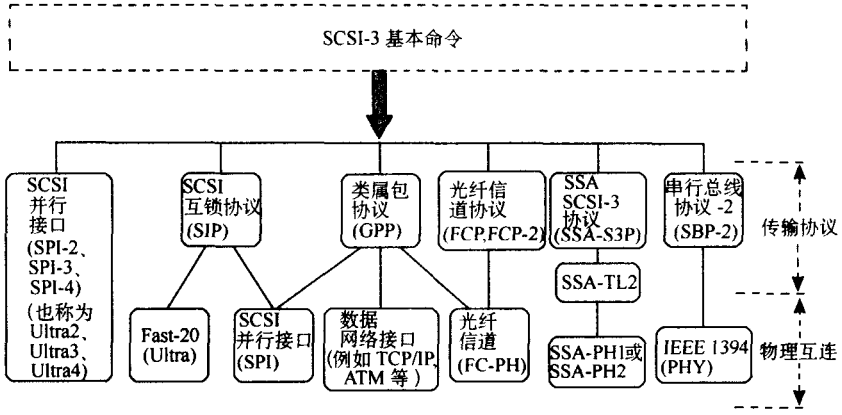


图 7A-4 SCSI 体系结构模型（SAM）

每个 SCSI-3 串行协议都有自己的协议堆栈，协议堆栈都要服从于顶层的 SCSI-3 通用访问方法，而且还要清楚定义传输协议和底层的物理接口系统。串行协议以包（或帧）的形式发送数据。这些数据包由一组字节构成，其中包含标识信息（即包头）、数据字节（称为包的有效载荷）和包的结束处的某种尾部界定符。在许多 SAM 协议中，包的尾部还包含错误检测代码。

在本节的余下部分，我们会仔细研究几个更令人感兴趣的 SAM 串行协议。

IEEE 1394

20 世纪 80 年代后期，并行 SCSI 系统是占主导地位的接口系统。现在称之为 IEEE 1394 的接口系统起源于苹果计算机公司。当时，苹果计算机公司注意到有必要创建一个比 SCSI 系统更快、更可靠的系统。苹果公司将这个接口称为 FireWire，现在 FireWire 可以提供 40MB/s 总线速度，而且预期在未来可能会有更高的速度。

IEEE 1394 不仅只是一个存储器的接口系统，它是一个对等存储器网络。它连接的设备都具有智能化的功能，允许设备彼此之间以及设备与主机控制器之间进行相互通信。这些通信包括了传输速度的协商和总线的控制等内容。遍及 IEEE 1394 协议的各个层次都具有这些功能，如图 7A-5 所示。

IEEE 1394 不仅比早期的并行 SCSI 提供更快的数据传输，而且仅使用一种很薄的电缆连接线。电缆中只有 6 根带线，其中 4 根用于数据和控制，另外 2 根用于电源。这种小型的电缆比带有 50 芯的 SCSI-1 或 SCSI-2 的电缆更加经济和更易于管理。而且，IEEE 1394 电缆可以在设备之间延长到 15 英尺（4.5 米）。并允许高达 63 个设备在一条总线上构成菊花链连接。IEEE 1394 的连接器是一种标准的接头，在样式上与游戏机的连接器非常相似。

整个的 IEEE 1394 系统是自配置的系统。在系统正在运行时，IEEE 1394 接口允许大量的设备非常方便地进行热插拔（hot-plugging），或者说即插即用（plug and play）。然而，热插拔需要花费一定资源开销。计算机需要采用轮流检测来确定连接到接口的各个设备，这样最终会限制接口的吞吐量。而且，如果某个连接器正在忙于处理一长串的同步数据时，那么它就不可能在传输数据过程中对一个正在接插设备立即做出应答。

我们还可以将设备插到其他一些设备的附加端口上，这样产生一个如图 7A-6 所示的树型结构。对于数据 I/O 来说，树型结构的用途是非常有限的。由于支持同步数据传输，因此 IEEE 1394 在消费

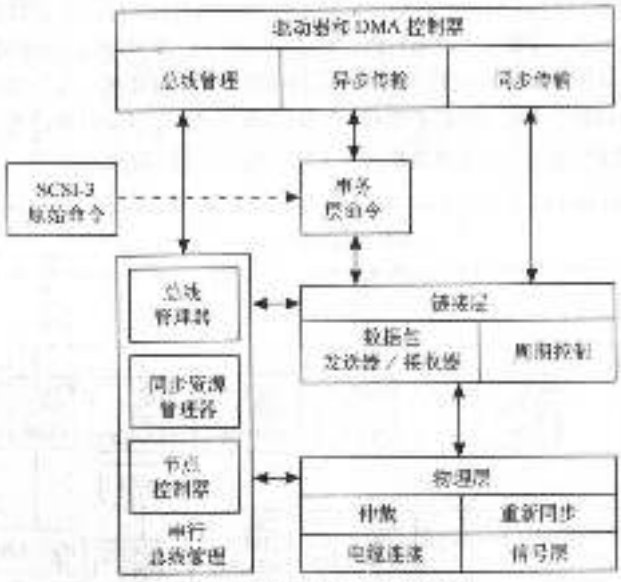


图 7A-5 IEEE 1394 的协议堆栈

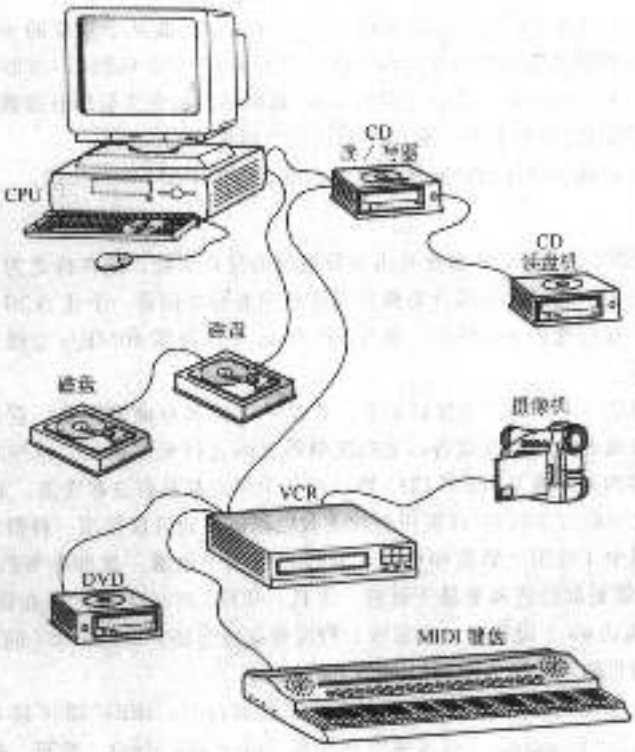


图7A-6 IEEE 1394 的树型结构配置，连接有不同的消费类电子产品

类电子产品中得到了广泛的认同。在实验室的数据获取应用方面，IEEE 1394 也正在取代以往采用的 IEEE 488 通用接口总线（General Purpose Interface Bus，GPIB）。由于主要专注于实时数据的处理，所以 IEEE 1394 不太可能替代 SCSI 成为高容量的数据存储器接口。

串行存储器体系结构

尽管串行存储器体系结构（Serial Storage Architecture，SSA）有许多优点，但是在存储器接口的领域中，它面临着被淘汰的可能。在 20 世纪 90 年代早期，IBM 公司和许多计算机制造公司一样，都在寻找一种更快更可靠的替代并行 SCSI 接口的产品用于大型计算机的磁盘存储器系统。对于长距离电缆传输，IBM 公司的工程师决定采用串行总线。这种串行总线要求既简洁又具有低的费用。而且，还要求总线的吞吐能增加和保持向下兼容 SCSI-2 协议。经过充分的精简和修改，IBM 公司终于在 1992 年的年底向 ANSI 建议将 SSA 定为一种工业标准。在 1996 年的下半年该标准被批准通过。

SSA 的设计可以利用一个环形结构的连接方式，支持多个磁盘驱动器和多台主机系统，如图 7A-7 所示。连接电缆是一根由两对双绞铜线组成的 4 芯导线，允许信号在环形结构中进行相反方向传送。因为这种连接上的冗余性，所以即使有一个驱动器或主机适配器失效，也并不会影响对其他磁盘的访问。

这种 SSA 体系结构的双环拓扑结构同样使接口的基本吞吐量从 40MB/s 增加到 80MB/s。如果所有节点都能够正常运转，那么各个设备就能够按照全双工（full-duplex）模式进行相互间通信，即在环形体系结构中数据可以同时进行双向传输。

SSA 设备可以管理它们自己的某些 I/O 系统。例如，在图 7A-7 中，当主机适配器 B 正在向磁盘 3 号数据的同时，主机适配器 A 可以读取磁盘 0 的数据，磁盘 1 正在发送数据到磁带单元，而磁盘 2 也正在向打印机传输数据。在这里，并不会因为总线本身的原因而导致吞吐量的下降。IBM 称这种思想为空间复用（spatial reuse）。这是因为，如果在起始端和目标端设备存在畅通的数据通道，那么系统的任何部分都不需要等待总线。

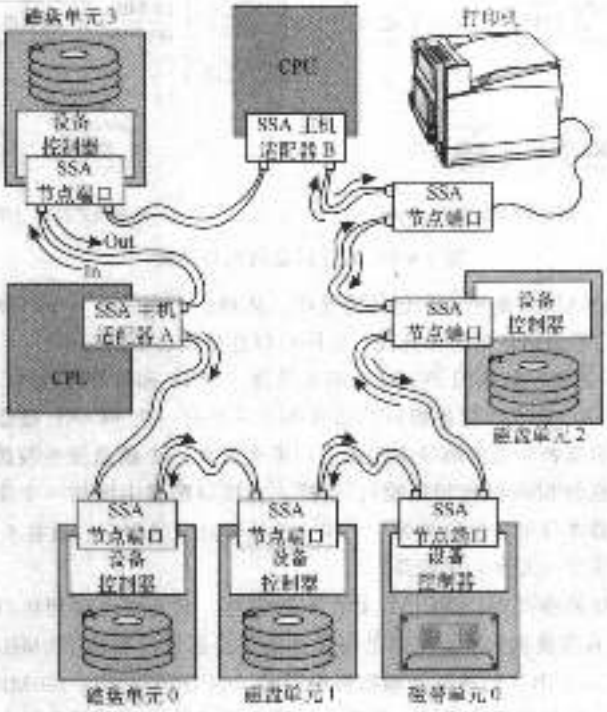


图 7A-7 串行存储器体系结构（SSA）的配置

由于具有简洁、高速和可靠的特点，SSA 过去一直是大型计算机系统的主要互连方法，直到后来光纤信道的出现。

光纤信道

1991 年，在瑞士日内瓦，欧洲原子核研究组织（CERN）实验室的工程师开始设计一种在光纤介质

上传输网络通信的系统。他们称这种系统为光纤信道（Fibre Channel），这是光纤的欧洲拼法。第二年，HP（Hewlett-Packard）、IBM 和 Sun 公司组成一个联盟体共同使用光纤信道作为磁盘接口系统。这个团体后来发展成为光纤信道协会（Fibre Channel Association，FCA）。该组织与 ANSI 一起，为存储器设备的高速接口建立了一个精确和牢固的物理模型，虽然最初的目标是定义光纤接口，但是光纤信道协议也适用于双绞线和同轴铜芯电缆。光纤信道的存储器系统具有三种类型的拓扑结构：交换式结构、点对点结构和环状结构。环状拓扑结构又称为光纤信道仲裁环（Fibre Channel Arbitrated Loop，FC-AL），是这三种结构中应用最广泛的，也是成本费用最低的结构。光纤信道的各种拓扑结构如图 7A-8 所示。

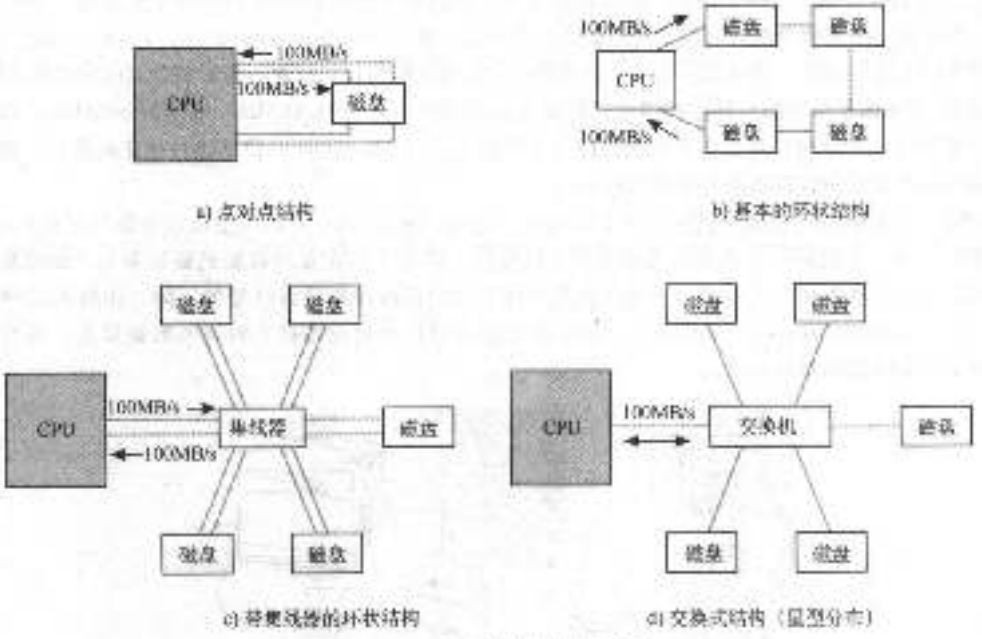


图 7A-8 光纤信道的拓扑结构

FC-AL 可以提供 100MB/s 的单方向包传输速率。从理论上来说，一个环状结构可以连接设备的最大数目为 127。但是，考虑到各种限制情况，实际可以连接设备数目为 60。

注意，图 7A-8 展示了两种版本的 FC-AL，有集线器（c 图）和没有集线器（b 图）。图 c 中的集线器是一个交换设备。FC-AL 集线器配备端口旁路开关。只要有一个 FC-AL 磁盘失效时，端口旁路开关就会起作用。如果没有某种类型的端口旁路能力，那么只要一个磁盘发生故障时，整个环就不能够正常工作（注意将这一点与 SSA 结构相比较）。这样，在接口配置中增加一个集线器可以引进磁盘失效保护机制。因为集线器本身可能也会成为一个单一的故障点（尽管它们通常不会失效），对于要求高可用性的系统可以安装多个（冗余）集线器。

交换式光纤存储系统能够提供比 FC-AL 宽得多的带宽，而且对连接到接口的设备数目（可多达 2^{24}）实际上没有限制。在交换机和节点之间的每条连接线路都可以提供 100MB/s 的传输速度。因此，在两个磁盘相互之间以 100MB/s 的速度传输数据的同时，CPU 还能够以 100MB/s 速度向另外的一个磁盘传输数据等。正如我们所想象的那样，交换式光纤信道结构比环状结构更加昂贵，主要是因为交换式光纤信道需要一些更加复杂的交换设备。而且这些设备的数目必须具有一定的冗余性，以保证了系统能够持续不间断地操作。

光纤信道是数据网络和存储器接口的混合结构。光纤信道协议堆栈既满足 SAM 规范又符合国际认可的网络协议堆栈。光纤信道协议堆栈如图 7A-9 所示。因为高层协议映射的缘故，一个光纤信道存储器在配置安排上并不一定要求与 CPU 直接连接；光纤信道的协议包可以封装在一个网络传输包内，

或者作为一个 SCSI 命令直接传送。FC-4 层就负责处理这些细节问题。

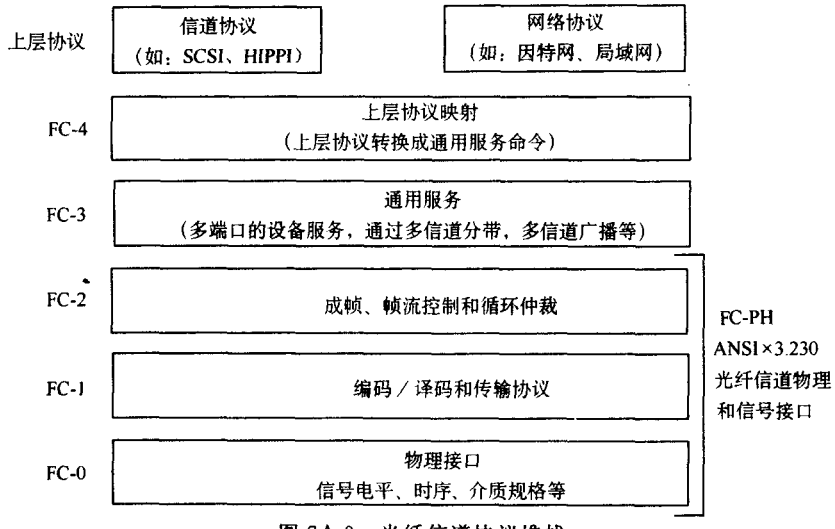


图 7A-9 光纤信道协议堆栈

FC-2 层负责生成一个协议包（或帧），这个协议包中包含来自上层的命令或数据，或者是来自下层的响应和数据。这个协议包的大小固定为 2148 字节，其中的 36 个字节用做定界，路由安排和误差控制，如图 7A-10 所示。

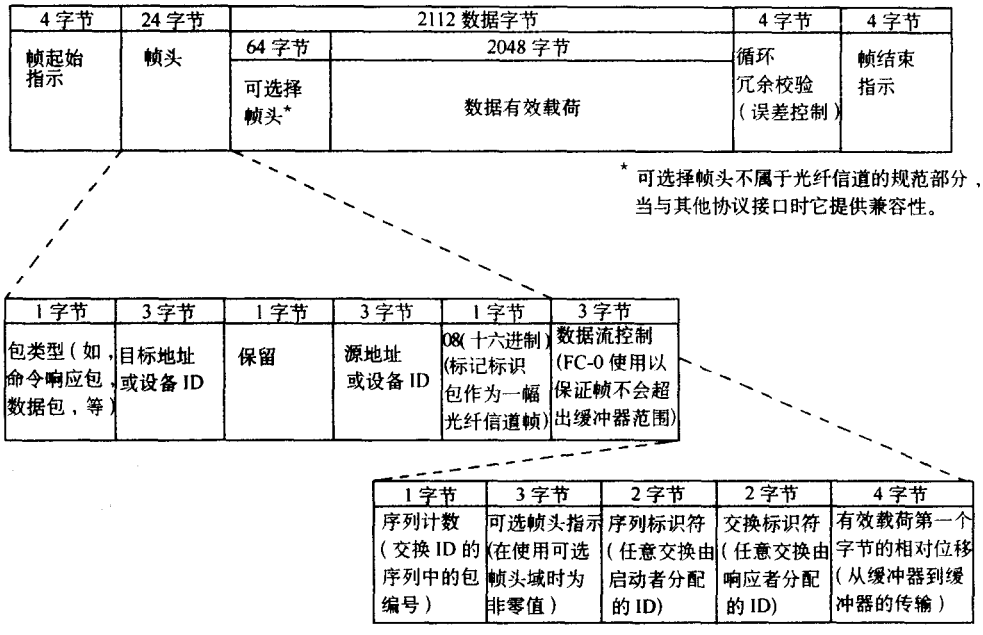


图 7A-10 光纤信道协议包

当 FC-AL 通电时，FC-AL 环自身会进行初始化。这时，各个相关的设备广播他们自己的身份，相互协商设备（端口）号，并且会选择一个主控设备。然后，通过包的交换发生数据传输。

FC-AL 是一种点对点协议，在某些方面与 SCSI 协议相类似。如果只有两个节点，即只有发起者和响应者，它们之间的每次通信都可以使用总线。当发起者想使用总线时，它会将一个被称为 ARB（x）的特殊信号放到总线上。这就是说设备 x 希望仲裁控制总线。如果此时没有其他的设备控制总线，那么环

上的每个节点将会沿着逆时针方向逐次将 ARB (x) 传到下一个相邻的节点。直到这个数据包最终又回到发起者。当发起者看到自己的 ARB (x) 在总线上没有发生变化时，就知道它已经赢得了总线的控制权。

如果在这个环上有另外一个设备控制着总线，那么在返回到发起者之前，ARB (x) 包将会被改变成一个 ARB (FF) 包。此后，该发起者会再次进行尝试。如果同时有两个设备试图取得总线的控制权，那么最高节点数的设备将会赢得总线的控制权，而另外的设备稍后会重试。

发起者会通过接通与响应者的连接来宣称自己对总线的控制。这一过程通过发送 OPN (yy)（对全双工而言）或者 OPN (yx)（就半双工而言）命令来完成。直到接收到 OPN (FF) 命令之前，响应者会一直处于“准备”状态，并且会发送“接收器就绪”命令来通知发起者。一旦数据传输完成，发起者发出一个“关闭 (CLS)”命令来放弃对总线的控制权。

数据传输协议的细节取决于在环路或光纤中应用的服务类型。有些服务类型要求对发送的数据包进行确认（要求最大的准确性），而有些服务类型却并不要求对发送的数据包进行应答（要求最大的传输速度）。

目前，对于光纤信道的数据传输来说，定义了 5 种服务类型。当然，并不是所有服务类型都已经在真实产品上得到了实现。而且，如果没有足够的传输带宽可用，某些类型的服务可以混合使用。当传输环或信道没有被第一类服务占用时，有些具体的实现过程允许第二类帧和第三类帧可以混合进行传输。表 7A-3 归纳总结了目前已经定义的光纤信道的各种服务类型。表 7A-4 归纳总结了 IEEE 1394、SSA 和 FC-AL 的主要特点。

表 7A-3 光纤通道的不同服务类型

类 型	描 述
1	带有数据包应答方式的专用连接服务。但是由于连接管理过于复杂，所以这种服务类型并没有得到许多销售商的支持
2	类似于第一类服务，但并不要求专用连接。如果在网络中管理不同的传输路径，数据包可以在信号序列的范围外进行传送。第二类服务适合于低速信道、没有频繁突发传输的装置
3	无连接不应答的传输服务。数据包的发送和信号排序问题由上层协议进行管理。在具有充分带宽的小型网络中，通常这种传送很可靠，利用该协议可以协商实现实时的信号传输通路，所以非常适合于 FC-AL
4	从网络的全带宽中分割出来的虚拟电路。例如，一个 100MB/s 的网络可以支持一个 75MB/s 和 25MB/s 的连接。这种虚拟电路中的每一个都允许有不同的服务类型。到 2002 年，市场上还没有商业化的第 4 类产品
5	带有应答传输方式的从一个信号源到另一个信号源的多信道广播。对于音频或视频广播非常有用。为了防止广播节点的扩散（如果对广播传输采用第 3 类服务的话将会发生节点扩散问题），需要在网络上放置一个独立的节点来管理广播的应答事务。到 2002 年为止，还没有第 5 类产品投入市场

表 7A-4 一些 SCSI-3 体系结构模型的速度和容量

接 口	设备之间允许的最大电缆长度	最大数据传输率	控制器可以连接的最大设备数目
IEEE 1394	4.5 m (15 ft)	40MB/s	63
SSA	Copper: 25 m (66 ft) Fiber: 680 m (0.4 mi)	40MB/s	129
FC-AL	Copper: 50 m (165 ft) Fiber: 10 km (6 mi)	25MB/s 100MB/s	127

7A.4 存储器的区域网络

光纤通道技术的发展使我们可以为存储器的访问和管理来构建一种专用的网络。这种网络称为存

储器的区域网络（storage area networks，SAN）。从逻辑上来说，SAN 扩展了存储器的局域总线，使得一组大量的存储器设备的集合能够被各种计算机的平台所访问，包括小型计算机、中型计算机和大型计算机。各种存储器设备可以通过多个主机系统来进行配置安排，或者将这些存储器设备用作为几英里之外的原始操作场所的“热”备份系统。

与网络连接的存储器（network attached storage，NAS）模型相比，SAN 可以更方便更快速地访问大量的存储器。在一个典型的 NAS 系统中，所有文件的访问都必须通过一个特殊的文件服务器。这样可能会产生各种各样的协议管理费用，以及与网络相关的通信拥塞。这种磁盘访问协议（SCSI-3 体系结构模型的命令）被嵌入在网络数据包中，会造成两层协议的管理费用和对数据包的进行编译装配/分解的两次迭代操作过程。

SAN 有时也称为“网络背后的网络（the network behind the network）”，它与普通的网络通信是分离开来的。光纤信道存储器网络（既可以是交换式的，也可以是 FC-AL）潜在地要比 NAS 系统快很多，原因是光纤信道存储器的网络只有一个协议堆栈来控制整个网络系统。因此，它们避开了使用传统的文件服务器，这些文件服务器通常可能会成为网络通信的瓶颈。图 7A-11 和图 7A-12 分别给出 NAS 和 SAN 配置结构的示意图作为比较。

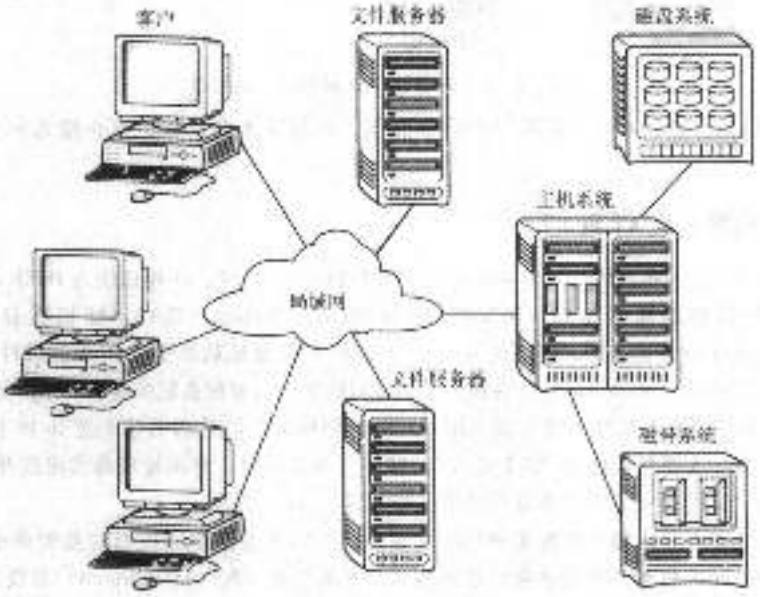


图 7A-11 网络连接的存储器（NAS）

因为 SAN 独立于任何特定的网络协议（例如 Ethernet）或各种主机的专用能属协议，所以我们可以利用任何经合理配置后就能够识别 SAN 存储器设备的平台，通过 SAM 的上层协议很方便地访问存储器的区域网络。而且，大大地简化了存储器的管理，因为所有的存储器都位于一个 SAN 网络上。这一点与文件服务器和磁盘阵列不同。我们可以通过电子传递的方法，在遥远的地方进行数据传输，或者将数据备份到一个磁带上，而不会影响到网络或主机系统的工作。由于具有高速、灵活和强大的功能，SAN 可以提供高可用性的海量存储器，正成为大用户社区的首选系统。

7A.5 其他的 I/O 连接

还有大量的 I/O 体系结构不属于 SCSI-3 体系结构模型的范围，但是这些 I/O 体系结构都能够在某种程度上实现与 SCSI 连接，其中 AT 个人计算机附属设备的 I/O 接口最为流行。这种 I/O 体系结构主要应用于低端的计算机系统。除了 Intel 公司的范例外，为计算机体系结构而设计的其他一些

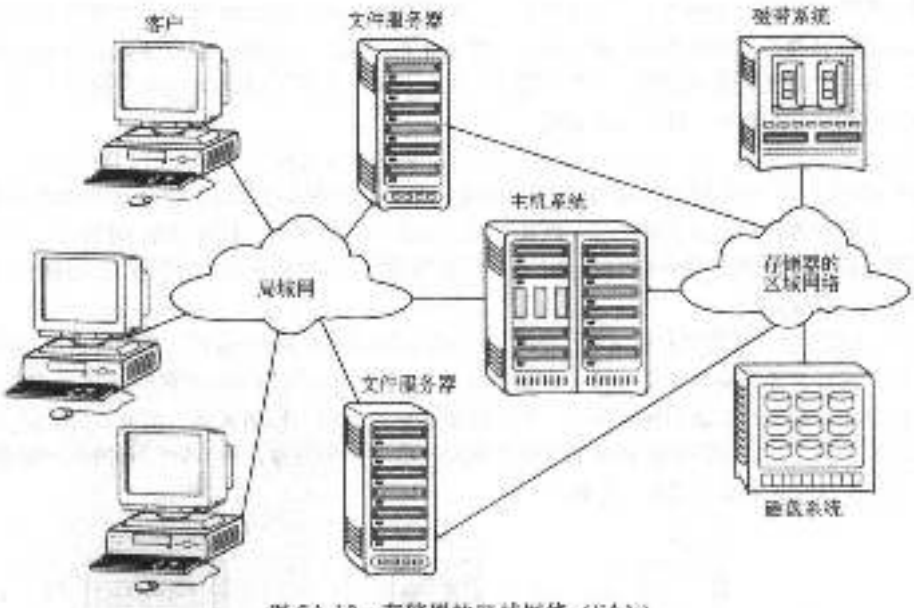


图 7A-12 存储器的区域网络（SAN）

I/O也在不同类型的计算机平台上得到了广泛的应用。在接下来的部分，将介绍几种比较流行的 I/O 连接。

7A.5.1 并行总线：从 XT 到 ATA

第一个 IBM PC 机所使用的是一个 8 位总线，称为 PC/XT 总线。这种总线为 IEEE 组织所接纳，并且重新命名为行业标准体系结构（Industry Standard Architecture，ISA）。最初该总线工作速度为 2.38MB/s，而且由于其带宽较窄，所以访问一个 16 位的存储地址就需要占用两个时钟周期。因为 XT 运行的速度为 4.77MHz，所以 XT 总线表现出了良好的性能。随着配备较快速的 80286 处理器的 PC/AT 计算机的出现，8 位的总线显然已经不能再用。最直接的解决办法是将总线拓宽到 16 位数据线，增加时钟频率到 8MHz，这种总线称为“AT 总线”。然而，不久以后，当微处理器的速度开始超过 25MHz 时，这种新的 AT 总线就成了一个非常严重的系统瓶颈。

在接下来的几年内，市场上出现多种解决方案，这些解决方案中应用时间最长的是 AT 总线的变化形式，有几种不同的版本：这就是我们熟知的 AT 附属设备（AT Attachment）总线、ATAPI 总线、快速 ATA 总线和 EIDE 总线。最后一种的缩写代表的是增强型集点驱动器电路（Enhanced Integrated Drive Electronics）。之所以这样称呼，是因为许多原来位于磁盘驱动器接口卡上的控制功能现在都被集成到磁盘驱动器本身的控制电路之中。AT 附属设备总线能够向下兼容 16 位的 AT 接口卡，同时对于磁盘驱动器和其他的设备允许连接 32 位的接口。外部设备不能够直接连接到 AT 附属设备总线上，而内部设备的数量限制为 4。根据所使用的是编程控制的 I/O 还是 DMA 控制的 I/O 系统，AT 附属设备总线可以支持 22MB/s 或 16.7MB/s 的传输速率，而理论上的最大传输速率为 66MB/s。在这样的传输速度下，ATA 为当今市场上的小型计算机系统提供了最佳性价比的总线系统。

7A.5.2 外围设备互连

到 1992 年，AT 总线已经变成了制约整个小型计算机系统的性能的主要因素。考虑到 AT 总线的时代即将结束的这种趋势，Intel 公司开始资助一个行业组织为小型计算机系统设计一种更快更灵活的 I/O 总线。而他们努力的结果是产生了外围设备互连（Peripheral Component Interconnect，PCI）总线结构。

PCI 总线是系统数据总线的一种扩展，它能够取代计算机系统中的任何其他的 I/O 总线。当 PCI

在一个 CPU 字的全带宽条件下，其运行速率可以高达 66MHz。因此，对于一个 32 位的 CPU 来说，理论上的数据吞吐量为 264MB/s（即，66MHz×（32b/s÷8bits/byte）=264MB/s）。而对于一个 64 位运行在 66MHz 时钟速率的 PCI 总线，最大的传输速率为 528MB/s。尽管 PCI 与系统总线相连接，但是 PCI 总线可以自行协商总线的速度和数据的传输，而不需要 CPU 的干预。PCI 总线既快又灵活。因此，各种版本的 PCI 总线不但在小型的家用计算机中使用，而且应用在能够支持实时数据获取和科学研究的高性能计算机系统中。

7A.5.3 串行接口：USB

通用串行总线（Universal Serial Bus，USB）并不是一个真正意义上的总线。实际上，USB 只是一个连接微处理器扩充总线的串行外围接口，就像任何其他的扩充卡一样。目前，第二代的 USB 2.0 版本，无论是在性价比还是在使用的方便性方面，都要超过 AT 附属设备总线，这使得 USB 接口在家用计算机系统的应用领域非常具有吸引力。USB 2.0 的设计者宣称，他们的产品使用起来就像把一台电话插到墙上的插孔中一样方便。

USB 需要在主机系统上安装一个叫做根集线器（root hub）的适配器卡。根集线器可以连接一个或更多的多端口集线器，多端口集线器能够直接与外围设备相连接，其中包括视频照相机和电话等。而多端口集线器可以进行相互的级联，直至 5 级连接。一个单一的根集线器最多可以找出 127 个外围设备。

USB1.1 的主要缺点是比较慢，速度仅为 12MB/s。这样的数据传输速率，USB1.1 与那些慢速的设备的配合还是比较好的，例如打印机、键盘和鼠标等。但是，USB1.1 很少应用于磁盘系统或者是同步数据的传输。USB2.0 的主要改进是在理论上它的最大数据传输速率达到 480MB/s，远远超过了目前的桌面型计算机系统的需求。USB 的最大优点是功耗低，非常适合于便携式计算机系统的选择。

7A.5.4 高性能外围设备接口：HIPPI

在宽带系统方面主要应用的是高性能外围设备接口（High Performance Peripheral Interface，HIPPI）。HIPPI 是以千兆比特速率互连的主机系统和超型计算机的 ANSI 接口标准。1990 年，ANSI X3T11 技术委员会发布了第一套 HIPPI 规范。如果配合适当的硬件系统，HIPPI 也可以用于高容量的存储器接口以及用作局域网的骨干主链协议。HIPPI 目前最大的传输速度为 100MB/s。如果将两个 100MB/s 的连接复用就可以组成一个 200MB/s 的连接。为 HIPPI 制定一个 6.4MB 比特的标准目前正在研究之中，这个新的 HIPPI 标准以全双工的模式运行时可以提供 1.6MB 字节的带宽。

人们常常把 HIPPI 电缆比作灭火水龙带，它由 100 根铜导线组成，这些铜导线为 50 对相互屏蔽的双绞线。如果没有转发器，HIPPI 能够沿着铜导线传输大约 150 英尺（50 米）的距离。光纤 HIPPI 连接在不使用转发器的情况下可以将最大传输距离扩展到 6 英里（10km），具体距离还要取决于所使用的光纤类型。作为大量并行的计算机系统的一种并行互连的设计方案，HIPPI 可以与许多其他的总线和协议进行互连，其中包括 PCI 和 SAM 总线。

7A.6 小结

本专题概要介绍了一些适用于各种大型和小型计算机系统的流行的 I/O 体系结构。对于小型计算机系统，适用的 I/O 体系结构有：ATA、IDE、PCI、USB 和 IEEE 1394 等；而为高容量的大型计算机系统设计的 I/O 体系结构有：HIPPI 和某些 SCSI-3 协议。SCSI-3 的体系结构模型已经被重新定义为高速接口。由于计算机和存储器系统具有越来越多的互连趋势，所以 SCSI-3 的体系结构模型的某些方面已经覆盖到了数据通信的领域。

光纤信道是目前应用在服务器群上的最快的接口协议之一，而其他的一些协议也正在开始出现。现在从“管理存储器”的概念中开始发展出一个产业来，它就是由第三方负责为客户公司进行短期的和长期的磁盘存储器管理。我们可以预期这种外判的服务领域能够继续发展壮大，带来更多的新思想、协议和体系结构。

练习题

1. 本节所讨论的哪一种类型的存储器体系结构能够在大型的数据中心或服务器群上找到？如果在数据中心的环境中，使用其他类型的一种体系结构会出现什么问题？
2. 在仲裁阶段完成后，会有多少个 SCSI 设备被激活？
3. 假如在异步并行 SCSI 数据的传输过程中，有人从预定的传输对象的驱动器中移走一个软盘。在下面的这些阶段，启动设备是如何知道发生了错误？
   - 总线空闲
   - 状态
   - 选择
   - 消息
   - 命令
   - 重新选择
   - 数据

   a) 如果数据传输是“写”操作，那么在上面的哪个阶段有可能将好的数据写到软盘上？
   b) 如果数据传输是“读”操作，那么又在上面的那一个阶段，系统会有好的数据存放在缓冲器中吗？系统会对该数据进行确认吗？
4. 假定你的经理决定用一个高速和宽带的 SCSI-3 适配器取代原来老的 SCSI-2 主机适配器，来改善文件服务器的吞吐量。她还决定用一些比原来大得多的高速和宽带 SCSI-3 驱动器来替换老的 SCSI-2 驱动器。当将所有的文件从老的 SCSI-2 驱动器上移到新的 SCSI-3 驱动器上之后，你又对原来老的 SCSI-2 驱动器进行了格式化，以便它们还可以在其他地方重新使用。当完成这一切时，你的经理告诉你把老的 SCSI-2 驱动器留在服务器中，因为她知道 SCSI-2 是向下和 SCSI-3 兼容的。作为一个好的员工，你会按照经理的要求去做。然而，几天以后，SCSI-3 的升级并没有取得经理所期望的系统性能的改进。对于经理的失望，你一点都不觉得惊奇。到底发生什么样的情况？你可以如何来解决这个问题？
5. ◆ 你刚刚将系统升级为一个高速和宽带的 SCSI-3 接口。系统带有一个软盘驱动器，一个 CD-ROM 和 8GB 的硬盘驱动器。请问主机适配器的设备号是多少？为什么？
6. SCSI-2 与 SCSI-3 体系结构模型的原理有何不同？
7. SCSI-3 体系结构的模型为计算机和外围设备制造厂商带来了哪些方面的好处？
8. 假设你现在要设计一个连接许多计算机和摄像机的视频会议系统。你会选择哪种接口模型？用于传输视频的协议包和用于传输数据的协议包是否完全相同？有什么样的协议内容是包含在其中的一个协议包内，而在另外一个协议包中却是没有的？
9. 一个 SSA 总线的配置是如何从单磁盘失效的故障中恢复数据的？假设在第一个节点修复之前，另一个节点又发生失效故障，这时系统该如何完成恢复任务？
10. 假设你现在被分配到一个工作小组，任务是在化学工厂里安装自动化控制系统，要将几百个传感器放置于遍及整个工厂厂区的储液桶、缸和容器之中。从传感器送回的所有数据都会被输入到一组功能强大的计算机中，以便值班经理和管理人员能够监视和控制生产的各个过程。你准备在传感器和计算机之间使用哪种类型的接口？如果所有的计算机都要求能够访问所有的传感器输入，那么对于各个计算机之间互连，你是否会使用同一种类型的连接？你会使用哪一种 I/O 控制模式？
11. 假设现在有一个工程师提议要改变公司制造的计算机系统的总线体系结构。她声称如果修改后的总线可以直接支持网络协议，那么计算机系统就不再需要配备网络接口卡。她还宣称这样可以取消 SAN，而将客户的计算机直接连接到磁盘阵列上。你会反对这个建议吗？请解释原因。

程序就是在计算机上进行的一种拼写转换，将输入变成错误信息。

—— 无名氏

第8章 系统软件

8.1 概述

在工作中，我们常常会遇到这样的情形：由于只有某类机器才能运行公司要求的特殊软件，所以我们不得不购买不是最理想的计算机硬件。虽然这与我们的更佳判断事与愿违，但是我们必须意识到一个完整的计算机系统不仅要有硬件还需要有软件的支持。软件是一个窗口，用户需要通过这一窗口与计算机系统进行对话。不管计算机硬件设备如何精良，但是如果软件不能够按照用户的期望提供服务，那么人们还是会认为整个系统是不合乎要求的。

在第1章中，我们介绍了计算机由6个不同的机器层次组成，其中门电路层上面的每一层都为其层提供一种抽象。在第4章中，我们讨论了汇编程序以及汇编语言与计算机体系结构之间的相互关系。本章将学习在计算机体系结构的第三层中应用的软件，并将这些思想与位于第四层和第五层的软件联系在一起。所有这三个层次的软件集合都运行在应用程序的下面，并且恰好就在指令系统结构层的上面。利用这三个层次的软件的集合，“机器”和用户应用程序的源代码之间就可以进行交互。这三个层次中的程序一起协同工作，提供了对用来执行应用程序中各种命令的硬件资源的访问。但是，如果我们只是将计算机系统看作从应用程序源代码直接运行到门电路层的单一线程，则会严重限制我们对计算机系统的理解。同时，我们也会忽略掉计算机体系机构中每一层次能够为用户提供的丰富服务。

尽管在这里的计算机系统模型中，只是把操作系统放在“系统软件”层上，但是系统软件的内容通常还包括编译器和其他一些实用程序（或称为工具，utility），以及一类有时称为中间件（middleware）的复杂程序。一般来说，中间件是一种广义分类，是指位于操作系统之上和应用程序之下的服务性软件。大家可以回忆一下，在第1章中我们讨论了有关在物理元件与高层语言及应用程序之间存在语义间隙。我们知道用户无法感知这种语义间隙，中间件是提供这种必需的不可见性的中间软件。操作系统是所有系统软件的基础，所以所有系统软件实际上都与操作系统发生一定程度的交互作用。我们先简单介绍操作系统的内部工作方式，然后再讨论一些更高的软件层。

8.2 操作系统

最初，操作系统的主要作用是帮助各种应用程序与计算机硬件进行交互。操作系统提供了一组计算机所必需的功能，允许软件包控制计算机硬件。如果计算机没有操作系统，那么所运行的每个程序都需要有自己的显卡驱动程序、声卡驱动程序、硬盘驱动程序，等等。

虽然现代操作系统仍执行此功能，但用户对操作系统的期望已有了相当大的变化。人们设想操作系统能够使系统管理和资源管理更容易。这种期望催生了“拖放（drag and drop）式”的文件管理和“即插即用（plug and play）”的设备管理。从程序员的角度来看，操作系统隐藏了计算机系统中较低体系结构层次的细节问题，而使用户的注意力能够集中到解决高层次问题上。我们已经看到，要在机器语言和汇编语言的层次上进行编程是一件非常困难的事情。操作系统与大量软件协同工作，创建了一个可以有效且高效利用系统资源的友好环境，在这种环境下无需再使用机器代码编程。操作系统不仅为程序员提供了这样一个编程接口，而且充当了应用软件与机器实际硬件之间的中间层。无论是从用户还是从应用程序代码行的角度来看，操作系统在本质上都是提供了从硬件到软件接口的虚拟机器。操作系统负责处理实际的设备和硬件，这样用户和应用程序则无需关心此工作。

操作系统本身只是一段普通的计算机程序。它与其他大多数软件的区别在于，操作系统在计算机启动时就会被装入，并且由处理器直接执行。因为操作系统的一项重要任务是CPU的进程调度，所以操作系统必须能够控制处理器和其他一些资源。对于各种应用程序来说，在应用程序执行期间，操作系统会放弃对CPU的控制。当应用程序不再请求CPU，或者是因为要等待其他资源而放弃CPU时，操作系统可以依靠处理器重新获得对CPU的控制权。

前面已经提到，对于用户和应用程序来说，操作系统是到底层的硬件的一个重要接口。除了作为接口之外，操作系统还有三大主要任务。进程管理也许是这三项工作中最令人感兴趣的任务，其他两项任务包括系统资源管理和保护系统资源不受错误进程的影响。在讨论这些任务之前，我们先来回顾一下操作系统的发展历史，看一看操作系统是怎样随着计算机硬件的演变过程而发展的。

8.2.1 操作系统的发展史

今天的操作系统提供了丰富的图形处理工具，力求做到不论是新手还是专业用户都能够轻松地使用计算机。但是在过去，情况却并非如此。几乎在这一代计算机之前，计算机的资源非常宝贵，每个机器周期都要用来完成一些有用的工作。由于计算机硬件的价格过高，人们对于计算机中的时间分配问题极为小心。当时，要使用计算机首先要登记上机时间。上机时得自己装入程序的穿孔卡片。机器的运行是采用单用户与人机对话的模式。然而，在装入程序之前，必须先在机器上装入程序的编译器。在要输入的卡片组中，最前面的一组卡片中包含引导装入程序，这个程序将负责完成其他剩余的卡片装入工作。此后，需要对程序进行编译。如果程序代码中有错误，必须迅速找到错误，并对出错的卡片重新打孔。然后，再将卡片组重新装入计算机，并尝试另一次编译。如果不能及时找到程序的出错位置，需要另外再登记时间等到下次再来尝试。一旦程序编译完成，接下来的工作就是链接结果代码和库代码文件，生成实际运行的可执行文件。显然，这一过程对于昂贵的计算机系统和操作人员的时间来说，都是一种可怕的浪费。为了让更多的人可以使用计算机硬件设备，人们引入了*批处理*（batch processing）。

利用批处理，专业的操作员会将多个的卡片组整合成若干批，或称为捆（bundle）。利用合适的指令可以让这些成批的卡片在最短的时间间隔内完成卡片的处理工作。通常这些分批的卡片都记录一些类型相似的程序。例如，可能一批卡片记录的是FORTRAN程序，而接着的一批是COBOL程序。这样一来，操作员就可以先将机器设置为FORTRAN程序的工作模式，读取并执行完成全部FORTRAN程序。然后，再将机器切换为COBOL模式。为此，人们设计了一种称为*常驻监控程序*（resident monitor）的程序。常驻监督程序允许机器在无人工干预的情况下处理各种程序，不同于将卡片组输入到读卡器再进行人工编译的过程。

常驻监控程序实际上是现代操作系统的前身，它的功能非常简单，即由监控程序启动程序任务，并将计算机的控制权交给该任务。当任务完成后，监控程序又重新接管机器的控制权。常驻监控程序将这些原来的人工工作交给计算机来处理，从而大大增加了系统的工作效率和实用性。然而，作者还记得，完成这种批处理任务所花费的周转时间是相当长的。我们还能够回忆起早先那些呆在数据中心收拾成堆的要处理的汇编语言卡片的痛苦日子。有时，我们得胆战心惊地等上差不多24小时才能取回运算结果。批处理的工作方式使得调试变得更加困难，或者更加准确地说是非常耗时。有时，程序中的一个死循环就可能在系统中造成严重破坏。后来，人们在监控程序中增加了一个计时器，以防止某个进程独占整个系统。然而，监控程序的作用由于没有提供额外的保护而受到了很大限制。由于没有保护，一个批作业任务可能会影响正在等候的批任务。例如：一个“坏”作业任务可能会因为读取了过多的卡片，而致使下一个程序也发生错误。此外，批作业任务甚至还可能会影响到监控程序的程序代码。为了解决这一问题，人们在计算机系统中设计了专门的硬件，允许计算机既可以在用户模式下工作，也可以在监控程序模式下工作。当程序在用户模式下运行时，如果有某个系统调用的需要，就可以将系统切换到监控程序的工作模式。

随着CPU性能的不断提高，这种依靠打孔卡片的批处理方式的工作效率显得越来越低。读卡器已经不能再保持CPU一直处于忙碌的工作状态。使用磁带为更快地处理程序卡片的任务提供了一种新的方法。人们将读卡器和打印机都连接到小型计算机系统上，这种小型计算机负责把卡片组上面的数据读取到磁带上。一条磁带可以包含多项作业任务。这样，主机CPU无需读卡即可继续在不同程序进程中进行切换。对于计算机的输出也可以使用类似的操作过程。首先将输出结果写到磁带上，然后再将磁带上的结果转移到小型计算机上，最后利用小型计算机完成实际打印输出。监控程序必须定期检查计算机是否需要I/O操作。在各种作业任务中都需要加入一个计时器，这样允许产生一些短暂的中断过程，让监控程序有时间将正在等候的I/O发送到磁带设备。这样一来，I/O操作和CPU的计算就可以并行发生。这种处理方式在60年代末到70年代末期间非常流行，被称为*外围设备联机并发操作*(Simultaneous Peripheral Operation Online，简称SPOOLing)，这是多道程序处理的一种最简单的形式。SPOOLing这个词也被收录到计算机词典中，但是这个单词现在的意义是指假脱机技术，即打印输出的数据在送到打印机之前要先写到磁盘上。

60年代末期创建的并且一直沿用至今的*多道程序处理系统*（multiprogramming system）则扩充了spooling和批处理的思想，允许多个程序同时在存储器（内存）中并发执行。这种并发处理是通过在程序进程之间进行轮流切换来实现的。这里，计算机安排每个程序进程在某个特定时间段内使用CPU。监控程序可以在一定范围内处理多道程序的操作。监控程序可以启动各种作业任务，实现假脱机，执行I/O操作，在不同用户任务之间进行切换，以及在不同任务之间提供某种保护。然而，随着监控程序的任务变得越来越复杂，我们需要有更加精巧复杂的软件。监控程序正是从此时开始演变发展成为我们现在所熟知的操作系统（operating system）。

尽管操作系统减轻了程序设计人员和计算机操作人员大量繁重的工作，但是用户总是希望能够与计算机进行更加密切的交流。尤其是单纯的批处理作业并不是非常具有吸引力。如果用户能够通过人机对话向计算机系统提交任务并且得到立即反馈，岂不是一件非常好的事情？分时共享的计算机系统(time-sharing system）就实现了这样的功能。分时共享的计算机系统连接多个用户终端，允许多个用户同时进行访问。因为交互式的编程促进了分时技术（也称为时间分割技术）的发展，所以批处理技术很快就过时了。在一个分时共享的计算机系统中，CPU通过依次分配给每个用户一个小段的处理时间，在与之对话的各个用户之间进行快速切换。这种切换过程称为*关联转换*（或称为*现场切换*，context switching)。从本质上来说，操作系统执行这种关联转换的速度非常快，这样就为各个用户提供了一台个人的虚拟计算机。

分时技术允许许多用户共享同一个CPU。将这种思想拓展开来，计算机系统也能够让许多用户共享一个应用程序。大型的交互式系统，例如飞机的机票预定系统，就要同时为成百上千的客户服务。因为使用了分时系统，大型交互式系统的用户并不知道在系统上同时还有其他用户。

引入多道程序处理和分时技术需要有更复杂的操作系统软件。在一个关联转换过程中，所有当前正在执行的程序的相关信息都必须保存起来，以便各个程序进程被安排到再次使用CPU时可以恢复到先前中断时的确切状态。这要求操作系统知道所有的硬件细节问题。大家记得，在第6章中我们介绍了在现代的计算机系统中都会使用虚拟存储器和分页技术。在一个关联转换的过程中，页表和其他与虚拟存储器有关的信息都必须保存起来。CPU的寄存器中的内容在关联转换发生时也必须保存，因为这些寄存器包含正在执行程序进程当前的状态信息。这种关联转换在资源或时间上都有很大开销。为了使得这种操作变得有价值，操作系统必须快速和高效地处理这些关联转换。

有趣的是，我们发现操作系统的演变和计算机体系结构的发展进步有着非常密切的联系。第一代计算机使用的是真空管和继电器，运算速度非常慢。因此，当时的计算机实际上就没有必要有操作系统，因为这些机器并不能处理多个并发任务。所需的任务管理事务由操作人员来执行。第二代计算机使用晶体管，在速度和CPU处理能力上都有较大提高。虽然CPU的处理能力增强了，但是这种计算机价格仍然非常昂贵，所以人们总是希望能够最大限度地利用它们的功能。这时引进了批处理的工作

方式，以提高 CPU 的利用率。监控程序可以帮助完成这种处理操作，提供最小的保护和中断处理。集成电路的应用是第三代计算机的标志。同样，集成电路也使得计算机处理速度大幅提高。这时，单独的 spooling 技术已经不能保证 CPU 高效工作，因此产生了分时技术。虚拟存储器和多道程序处理要求有一个更加复杂精巧的监控程序，这种监控程序发展成为现在的操作系统。第四代计算机技术，VLSI，使得个人计算机的市场蓬勃发展。网络操作系统和分布式操作系统正是这种技术发展的产物。电路的最小化也大大减少了 CPU 在芯片上所占用的空间，这样可以在芯片上集成更多的电路来管理流水线、数组（阵列）操作和多道程序处理等。

早期的操作系统在设计上有很大差别。计算机生产商们经常针对一个特定的硬件平台生产出一个或者多个专用的操作系统。同一个计算机的制造厂商为不同平台设计的操作系统在系统操作和它们所提供的服务上可能是完全不同的。在那个时候，当一种新型计算机出现时，生产厂商引进一种新的操作系统是一件非常普通的事情。直到 20 世纪 60 年代中期，IBM 公司引入 360 系列计算机时才基本上结束了上述的这种局面。尽管 360 家族中的每台计算机在性能上相差很大，但是所有的计算机都运行同一个基本的操作系统，OS/360。

Unix 是另外的一个操作系统，它也是一种操作系统可以跨越多个硬件平台的设计思想的范例。美国电话电报公司（AT&T）贝尔实验室的汤普森（Ken Thompson）从 1969 年开始致力于 Unix 操作系统的研究开发工作。汤普森最初利用汇编语言来编写 Unix。但是由于汇编语言是硬件专用的语言，在一种硬件平台下编写的任何程序代码，对于另一种硬件平台来说都必须重写和重新进行汇编。汤普森想到对于每一种不同机器都需要重新编写 Unix 操作系统，感觉非常沮丧。抱着希望减轻未来劳动的想法，汤普森创造了一种称为 B 语言的新的高级语言。然而，要支持操作系统的各种活动，B 语言的速度实在是太慢了。里奇（Dennis Ritchie）随后也加入和汤普森一起开发 C 编程语言，并在 1973 年发布了第一个 C 语言的编译器。汤普森和里奇利用 C 语言重写 Unix 操作系统，从此根除了操作系统一定要用汇编语言来编写的这个信条。因为 Unix 是用高级语言编写的，所以 Unix 能够在不同硬件平台进行编译，也就是说 Unix 具有很强的可移植性。这种与传统操作系统的主要区别使得 Unix 操作系统非常流行。虽然 Unix 进入市场的步伐比较缓慢，但是现在选择使用 Unix 的用户已经达到了几百万。Unix 所表现出来的与硬件无关的特性，允许用户为他们的应用程序选择最佳的硬件设备，而不会局限于某个特定的硬件品平台。现在从文字上统计就有几百种不同风格的 Unix 操作系统，其中包括 Sun 公司的 Solaris，IBM 公司的 AIX，HP 公司（Hewlett-Packard）的 HP-UX，以及为 PC 机和服务器设计的 Linux。

实时、多处理器、分布式网络操作系统

在最近几年，对于操作系统的设计人员来说，最大的挑战可能就是引入了实时、多处理器、分布/网络系统。*实时系统*（Real-time system）主要用于制造工厂的生产过程控制、装配线、机器人技术，以及类似空间工作站这样的复杂物理系统。当然，这里列举的只是少数几个例子。实时系统有非常严格的时间限制。如果不能满足特定的时间限制，就有可能造成物理上的损坏，或者是其他人力或物力损失。因为这些实时系统必须对外部事件做出反应，所以正确的进程调度是问题的关键。试想一下，如果一个控制核电厂的实时系统不能对反应堆的临界高温报警信号做出足够快的响应，将会发生什么情况？在一些硬（时间要求非常苛刻）*实时系统*（hard real-time system）中是不能出任何差错的。否则，如果时限不能满足，可能会有潜在的致命后果。在一些软（时间要求不那么苛刻）*实时系统*（soft real-time system）中，虽然对时限的要求也是必须的，但是如果时限条件不满足，不至于导致灾难性的结果。QNX 是*实时操作系统*（real-time operating system，RTOS）的一个典型代表，它能够严格满足时间调度的要求。QNX 同样适合于嵌入式系统，QNX 功能强大，另外带有一个小的轨迹覆盖区（footprint）（要求占用少许内存），并且趋向于非常的安全和可靠。

多处理器系统（multiprocessor system）有其本身所面临的挑战，因为需要对多个处理器进行调度安排。操作系统如何将各个程序进程分配给不同的处理器是设计时需要考虑的主要问题。通常，在一

个多处理器环境中，为了解决问题需要多个 CPU 相互协调，并行地工作来达到一个共同的目标。处理器活动的协调工作要求各个处理器之间有某种相互联络（通信）的方法。系统同步性要求决定了是采用紧密耦合的通信方式，还是松散耦合的通信方式来设计处理器。

紧密耦合的多处理器（tightly coupled multiprocessor）共享一个单一的集中式存储器，这要求操作系统必须非常小心地对各个程序进程进行同步处理以保护各种资源的安全。这种耦合方式通常用于 16 个或者更少一些的处理器上。对称式多处理器（symmetric multiprocessor，SMP）是紧密耦合的体系结构中非常流行的一种形式。这些系统具有共享存储器和 I/O 设备的多个处理器。所有处理器执行同样的功能，要处理的任务将会被分配到所有的处理器。

松散耦合的多处理器（loosely coupled multiprocessor）有一个物理上的分布式存储器，也就是我们所谓的分布式系统（distributed system）。对于分布式系统，我们可以用两种不同的方法来观察。局域网（LAN）上的分布式工作站的集合通常称为网络系统（network system），系统中的每台工作站都有自己的操作系统。设计这类系统的目的是为了让多台计算机共享资源。要将多台计算机连接到网络上，网络操作系统应该包括一些必要的功能，例如远程命令执行，远程文件访问，以及远程登陆等。用户的程序进程同样可以通过网络和另一台机器上的程序进程相互通信。网络文件系统是计算机网络系统的最重要的应用之一。尽管每台计算机处于不同的地理位置，具有不同的体系结构和不同的操作系统，但是网络文件系统允许多台计算机共享一个逻辑文件系统。这些系统之间的相互同步是一个非常重要的问题，但是系统相互间的通信问题可能更加重要，因为这些通信可能发生在远距离网络之间。虽然网络系统中的计算机可以分布在不同的地理位置，但是这种系统并不是真正的分布式系统。

真正的分布式系统与这些网络系统的重要区别在于：分布式操作系统在所有计算机上并发地运行，从用户的角度来看，分布式系统就是一台单一的计算机系统。与此相反，在一个网络系统中，计算机的用户能够感知到其他机器的存在。因此，透明性是分布式系统中的一个重要议题。在分布式系统中，不会简单地因为文件保存在不同的位置，而要求用户为文件使用不同的名字。用户只需对不同的机器使用不同命令，或者执行其他任何只与机器位置有关的交互（对话）操作。

最重要的一点是，为多处理器设计的操作系统和为单一处理器设计的操作系统之间并没有重大的区别。但是，这两种操作系统之间的主要差别是时间调度问题，因为多个 CPU 都必须保持忙碌状态。如果时间调度安排不恰当，多处理器并行执行的固有优势就不能完全体现出来。特别是如果操作系统不能提供合适的工具来开发多处理器的并行执行功能，多处理器系统的性能就会大打折扣。

正如我们所提到的，实时系统要求专门设计的操作系统。实时系统和嵌入式系统都要求尽量小的操作系统和使用最少的系统资源。另外，无线网络结合了嵌入式系统的紧凑性和网络系统的诸多特征，也已经在推动着操作系统设计的革新。

个人计算机操作系统

个人计算机操作系统的应用目标与大型计算机操作系统是不同的。大型机系统主要希望能够提供出色的性能和硬件利用率，同时也考虑怎样让系统变得方便使用。而个人计算机操作系统的主要目的是为用户提供更友好的使用环境。

当 Intel 公司在 1974 年推出 8080 微处理器时，曾经邀请 Gary Kildall 帮公司编写一个操作系统。Kildall 开发了一个软盘控制器，将磁盘和 8080 连接起来，并且为 8080 的控制系统书编了一个操作系统。Kildall 将这种基于磁盘的操作系统称为 CP/M（微型计算机的控制程序，Control Program for Microcomputers）。BIOS（基本输入输出系统，Basic Input/Output System）允许 CP/M 能够很方便地移植到不同种类的 PC 机上，原因是 BIOS 为输入和输出装置提供了所必需的交互作用。因为 I/O 设备在不同的系统之间是最可能发生变化的组件，所以通过将这些设备的接口打包成一个模块，实际操作系统就可以相对于不同机器保持不变。而需要改变的只有 BIOS。

Intel 公司错误地认为基于磁盘的计算机系统没有什么发展前途。在决定不使用这种新的操作系统

后，Intel 公司将 CP/M 的专利权交给了 Kildall。1980 年，IBM 公司需要为其 IBM PC 机装备一个操作系统。尽管 IBM 公司首先与 Kildall 进行洽谈，但是这桩交易最终交给了 Microsoft 公司。Microsoft 公司花 15 000 美元从西雅图的计算机产品公司购买了一个名为 QDOS（quick and dirty operating system）的基于磁盘的操作系统。这个软件后来并重新命名为 MS-DOS，其他的事情现在已经成为历史。

早期的个人计算机操作系统是通入键盘输入命令来操作的。当两位 Xerox Palo Alto 研究中心人员，GUI（图形用户界面，graphical user interface）的发明者 Alan Key 和鼠标的发明者 Doug Engelbart 将他们的思想结合在一起时，就永久地改变了操作系统的外观。通过他们的努力，原来的命令提示符已经由窗口、图标和下拉菜单所代替。Microsoft 公司通过 Windows 系列操作系统：Windows 1. x、2. x、3. x、95、98、ME、NT2000 和 XP，普及了这种图形界面的思想，当然这些思想并不是他们所发明的。Macintosh 的图形操作系统，MacOS，比 Windows GUI（图形界面）要提前几年出现，也经历了许多版本。Unix 通过 Linux 和 OpenBSD 在个人计算机中也变得流行起来。当然，还有许多其他磁盘操作系统，例如：DR DOS、PC DOS 和 OS/2 等。但是，它们都没有 Windows 操作系统这样流行，也没有 Unix 操作系统这样多的变体。

8.2.2 操作系统设计

因为计算机本身所使用的最重要的单一软件部分是计算机操作系统，所以我们必须对操作系统的设计给予足够的重视。操作系统控制计算机的一些基本功能，包括存储器的管理和 I/O，以及计算机的“视觉和触觉（look and feel）”的界面。操作系统与大部分其他软件的区别在于操作系统是事件驱动（event driven）的，这也意味着操作系统是通过响应输入命令、应用程序、I/O 设备和中断的事件来执行相关的任务。

计算机操作系统设计有 4 个主要的驱动因素：性能、功率、成本和兼容性。到现在为止，我们应该对什么是操作系统有了一种感性认识。但是，关于操作系统应该是什么样这个问题却有着许多不同的观点。也因此出现了各式各样的操作系统。大多数操作系统都有类似的接口，但是在如何执行任务方面却有很大的不同。有些操作系统选择的是最小化设计，只能完成一些最基本的功能。然而，其他一些操作系统却试图包含所有可以想像得到的功能特征。某些操作系统有出色的接口界面，但在其他的方面却相对欠缺些。另外一些操作系统则可能在内存管理和 I/O 控制方面很优秀，但在用户友好界面的方面却显得不足。没有一个操作系统能够在各个方面都做得很完美。

在操作系统设计中有两个关键组件：内核和系统程序。内核（kernel）是操作系统的核心。操作系统中的进程管理程序、调度程序、资源管理程序和 I/O 控制程序等都要使用内核。内核负责操作系统的事务调度、同步、保护/安全、内存管理和中断处理。内核具有对主要硬件设备的控制功能，包括中断、控制寄存器、状态字和定时器等。内核中装载了所有的设备驱动，可以提供一些常用的功能和协调所有的 I/O 事件。内核必须了解硬件的具体细节，以便将硬件的各个部分全部组合起来构成一个工作系统。

内核设计中的两种极端情况是：微核（microkernel）体系结构和单（monolithic）内核体系结构。微内核提供一些最基本的操作系统功能，要依赖于其他模块的支持来完成特定的任务。这样，微内核将许多典型的操作系统的任务都移到了用户空间。这样就允许系统重启或重新配置许多服务程序，而不需要重新启动整个操作系统。微内核可以提供一定程度的安全保障，因为在用户层运行的各种服务程序只能对系统的资源进行有限制的访问。与单内核相比，微内核更容易定制和移植到其他硬件平台。但是，在内核和其他模块之间需要增加额外的通信，这样常常会造成系统的速度减慢和效率降低。微内核设计的典型特征是规模较小，具有方便的可移植性，以及大量的服务程序运行在内核层的上层而不是运行在内核本身。SMP 和其他多处理系统的发展，大大促进了微内核的开发。微内核操作系统的范例包括：Windows 2000、Mach 和 QNX 操作系统。

单内核利用一个单一进程来提供操作系统的所有基本功能。因此，单内核比微内核大得多。通常，

针对特定的硬件系统，单内核直接与硬件发生交互。所以，与微内核的操作系统相比，单内核操作系统更容易进行优化。正是由于这个原因，单内核不容易移植。典型的单内核操作系统包括：Linux、MacOS 和 DOS 操作系统。

除了管理资源外，操作系统本身还会消耗资源，所以设计人员还必须考虑产品的总大小。例如，Sun 公司的 Solaris 操作系统的完全安装需要 8M 的磁盘空间，Windows 2000 需要大约两倍的空间。这些统计数字表明在过去的 20 年中，操作系统的功能在不断扩充。一张 100K 的软盘足以安装 MS-DOS1.0。

8.2.3 操作系统服务

在前面有关操作系统体系结构的讨论中，我们提到了操作系统可以提供的一些最重要的服务功能。操作系统负责监视所有关键性的系统管理任务，包括存储器管理，进程管理，保护，以及与 I/O 设备的交互操作。作为一个接口的角色，操作系统决定了用户怎样与计算机进行交互对话，在用户和硬件设备之间起到缓冲器的作用。操作系统的这些功能中的每一项都是决定整个系统的性能和可用性的重要因素。事实上，有时人们宁愿牺牲一些系统功能，而选择一个使用较为方便的计算机系统。这种平衡的考量在图形用户界面的领域会更加明显。

人机接口

操作系统在用户和计算机硬件之间提供了一个抽象层次。不论是用户还是应用程序都不需要直接面对硬件，这是因为操作系统提供了一种接口，可以隐藏机器硬件的细节。操作系统有三个基本接口，每种接口都是为某类特殊用户提供一种不同的视角。硬件开发人员感兴趣的是操作系统能作为硬件的一种接口。应用程序设计人员将操作系统视为各种应用程序和服务程序的一种接口。而普通用户最关心的是图形界面（接口），这也是接口这一术语最通常所包含的意义。

操作系统的用户界面分为两大类：*命令行界面*（command line interface）和*图形用户界面*（graphical user interfaces，GUI）。命令行界面中有一个命令提示符。在命令提示符下，用户可以输入各种命令，包括复制（拷贝）文件、删除文件、显示目录列表和更改目录结构的命令等。命令行界面要求用户了解系统命令的语法，这一点通常对普通用户来说很困难。然而，对于精通某种特殊命令词汇的人来说，使用直接命令来执行任务比使用图形界面的效率更高。另一方面，图形用户界面（GUI）为普通用户提供了一个更容易使用的界面。现代的图形用户界面由放置在桌面上的许多窗口组成。这些窗口中包含各种图标和代表文件的图示，利用鼠标对这些图标进行操作。命令行界面的例子有 Unix shells 和 DOS 界面。图形用户界面包括 Microsoft Windows 和 MacOS 等不同风格的操作系统。设备价格，特别是处理器和存储器价格的不断下降，使在许多其他操作系统增加图形用户界面变得切实可行。特别令人感兴趣的是，许多 Unix 操作系统都包含普通的 X Window 系统。

用户界面是一个程序，或者是一个小的程序集，这些程序组成了显示管理程序（display manager）。这个用户接口模块通常和驻留在操作系统内核中的核心操作系统功能是分离的。现代操作系统会创建一个整体的操作系统程序包，其中包含接口模块，文件处理模块，以及其他一些与操作系统内核紧密捆绑在一起的应用程序模块。这些模块之间的相互链接方式是当今操作系统中必须定义的一个特性。

进程管理

进程管理是操作系统服务的核心部分。进程管理包括从创建进程（即设置一些合适的结构来保存每个进程信息）开始，到调度各种资源在进程中的使用，直到删除进程和进程结束后的清理工作的所有事件。操作系统跟踪每个进程，包括进程的状态（其中包括变量值、CPU 寄存器中的内容和进程的实际状态：例如运行、就绪或等待），进程正在使用的资源情况，以及一些要求的其他信息。操作系统时刻监视每个进程的活动以防止出现同步问题（synchronization problem）。当一些并发的进程需要访问某些共享资源时，就会产生同步问题。显然，操作系统必须对这些活动非常仔细进行监控，以避免数据的不一致性和受到某些意外的干扰。

在任意时刻，操作系统内核都在管理一组进程的集合，包括用户进程和系统进程。大多数进程之间都是相互独立的。然而，当它们需要相互交流来完成某个共同的目标时，就要依赖于操作系统的帮助来实现进程之间的通信任务。

进程调度是操作系统的主要日常工作之一。首先，操作系统必须决定哪些进程允许进入系统（通常称为*长程调度*，long-term scheduling）。然后，操作系统还必须决定在任意给定时刻，哪个进程会得到CPU许可被执行（称为*短程调度*，short-term scheduling）。为了完成短程调度，操作系统需要保持一个已经准备就绪的进程队列的名单。这样，操作系统就能够区分哪些进程在等待资源，哪些进程在准备调度和运行。如果一个正在执行进程需要I/O或者是其他资源，则该进程会自动放弃CPU，并将自己置于等待的进程队列上，这样就可以对另外一个要执行的进程进行调度，此过程就是一个关联转换（context switch）。

在关联转换过程中，必须保存所有与现在正在运行的进程相关的信息，这样当重新执行该进程时，系统可以恢复到该进程中断时的确切状态。关联转换中需要保存的信息包括：CPU寄存器中的内容、页表和与虚拟存储器有关的其他信息。一旦这些信息被安全地保存起来，先前中断的进程（即正在准备使用CPU的进程）就可恢复到该进程先前中断时的确切状态（新进程没有先前的状态）。

进程可以采用两种方式放弃CPU。在*非占先调度*（nonpreemptive scheduling）事务中，某个进程会自动放弃CPU，可能是因为该进程需要某些没有预先安排好的其他资源。但是，如果系统的设置是按分段时间使用的，则该进程有可能被操作系统从一个正在运行的状态转移到一个等待的状态。这种过程称为*占先调度*（preemptive scheduling），因为该进程被别的进程占先取代，并且CPU的使用权被剥夺。当各个进程都按照优先级来进行任务调度和产生中断时，也会发生这种占先调度的问题。例如，如果一个低优先级的任务正在执行的过程中，有一个高优先级的任务需要使用CPU，则低优先级的任务马上会被放置到等待队列中（执行一个关联转换），允许高优先级的任务立即执行。

在进程调度中操作系统的主要任务是决定等待队列中哪一个将是下一个使用CPU的进程。影响调度的决定因素包括：CPU的利用率、吞吐量、周转时间、等待时间和反应时间。短程调度可以有很多种方案。这些方法包括：先到先服务（FCFS）调度，最短作业优先（SJF）调度，循环调度和优先级调度。*在先到先服务的调度*（first-come, first-served scheduling）中，操作系统是按照进程请求的顺序来为各个进程分配处理器资源。当正在执行的进程终止时，就会放弃对CPU的控制。FCFS调度是一个非占先的调度算法，其优点是这种调度方法很容易实现。但是，这种调度方法并不适合于支持多用户的计算机系统，因为一个进程要使用CPU必须等待的平均时间会有很大差别。另外，某个进程有可能会独占CPU，造成其他正在等待的进程的执行过程被无限制地延期。

在最短作业优先调度（shortest job first scheduling）方案中，具有最短执行时间的进程优于系统中所有其他进程取得CPU使用的优先权。可以证明，SJF是一种最佳的进程调度算法。但是，SJF遇到的主要问题是，还没有方法能够预先准确地知道一个将要运行的任务到底有多长。使用最短作业优先调度方案的系统会应用某些试探性的方法来“*估测*（guesstimate）”任务的运行时间。但是，这些试探性的方法还远远不够完善。最短作业优先调度方法可以是占先调度，也可以是非占先调度。这里，占先通常被解释为*最短剩余时间优先*（shortest remaining time first）。

循环调度（round robin scheduling）是一种公平和简单的优先调度方案。操作系统为每个进程分配一定的CPU时间片段。如果分配给某个进程的时间片段终止时，即使该进程还在执行，操作系统也会通过关联转换将该进程交换出去。随后，正在等待队列中的下一个进程将会被赋予一个属于它自己的CPU时间片段。循环调度已经广泛应用于分时共享的操作系统中。如果调度程序采用的时间片段足够小，那么用户们就感觉不到他们是在共享系统的资源。但是，这种循环调度的时间片段也不能划分得太小，否则相比之下关联转换时间会显得太长了。

优先级调度（priority scheduling）的方法与每个进程的优先级有关。当短程调度程序要从等待队

列中选择一个进程时，将会选中具有最高优先级的进程。FCFS 赋予所有的进程以相同的优先级，SJF 将优先权赋给最短的作业任务。而优先级调度所遇到的最严重问题就是存在潜在的资源缺乏（或者说饿死，starvation）问题，或者说存在着无限期阻塞的可能性。试想一下，如果要在一个繁忙的计算机系统上运行一个大的作业任务，而其他用户又在不停地在你的任务运行前提交一些较短的作业任务，这将会是一件多么令人沮丧的事情？对此还有这样的一个传说，在某个规模很大的大学里有一台大型计算机停止了工作，结果在等待队列中发现一项试图运行已经有好几年的任务。

某些操作系统采用的是多种调度方案的组合方式。例如，一个系统可能组合使用占先调度、基于优先级的调度、先到先服务的调度算法。一些支持企业级计算机系统的高度复杂的操作系统允许用户在一定程度上控制时间片段的长度，控制允许的并发执行的任务数目，以及为不同类型的作业任务分配不同的优先级。

多任务处理（multitasking，允许多个进程并发运行）和多线程技术（multithreading，允许一个进程向下划分为多个不同的控制线程）为 CPU 的进程调度带来了新的挑战。线程是系统中的最小可调度单元。各个线程共享与它们的进程母体相同的执行环境，包括进程的 CPU 寄存器和页表。正因为这样，线程中的关联转换所产生的开销更小。因此，这些线程中的关联转换发生的速度要比一个涉及整个进程的关联转换快得多。根据并发执行的程度需求，可以有一个单线程的进程，一个多线程的进程，多个单线程的进程，或者多个多线程的进程。支持多线程的操作系统必须能够处理上述的各种组合形式。

资源管理

除了进程管理之外，操作系统还要管理系统资源。因为这些系统资源比较昂贵，所以最好能够实现资源共享。例如，多个进程可以共享一个处理器，多个程序可以共享一个物理存储器，以及多个用户和多个文件可以共享一个磁盘。操作系统主要关心的资源有三种：CPU、存储器和 I/O。操作系统利用调度程序来控制对 CPU 的访问。而对存储器和 I/O 的访问则需要使用不同组的控制和功能子程序。

第 6 章讲过现代操作系统具有某种类型的用于扩充 RAM 的虚拟存储器。这意味着几个程序的部分内容可能会在存储器中共存，而且每个进程都必须有一个页表。最初，在操作系统的设计可以处理虚拟存储器之前，程序员使用一种覆盖（overlay）技术来实现虚拟存储器。如果一个程序太大无法放入到内存中，则程序员会把它分割成一些较小的片段。在特定的时刻，只装载那些运行程序所必需的数据和指令。如果程序需要新的数据或指令，程序员负责（并借助编译器的部分帮助）来确保正确的程序片段已经被装载到存储器中。当时是由程序员负责管理存储器。现在，操作系统已经接管了这一类琐事。现在的计算机系统由操作系统负责将虚拟地址翻译成物理地址，把页面传给磁盘和从磁盘上取回页面，并且负责对存储器的页表进行维护。而且，操作系统还要确定主存储器的分配和跟踪记录自由的页帧。在完成存储器空间的重新分配后，操作系统执行“垃圾回收（garbage collection）”的操作，把一些小的空闲的存储器碎片组合成一些比较大的可用的存储空间块。

除了共享单个有限的存储器外，进程还要共享 I/O 设备，在一个应用程序的请求下完成大部分的输入和输出操作。操作系统为输入和输出操作提供了必要的服务。当然，应用程序在没有操作系统的帮助下也可以处理它们自己的 I/O 操作。但是，这种方法除了要需要付出双倍的代价外，还会带来数据保护和设备访问方面的诸多问题。如果有几个不同的进程想同时使用同一个 I/O 设备，那么就需要对这些请求进行仲裁。当然，这种仲裁任务由操作系统负责完成。操作系统通过各种不同的系统调用为 I/O 提供某种类属接口。这种系统调用允许应用程序通过操作系统来请求一个 I/O 服务。操作系统随后调用设备的驱动程序。这些设备驱动程序中包括实现和特定的 I/O 设备相关的标准功能的软件。

操作系统还要管理磁盘文件，负责文件创建、文件删除、目录创建和目录删除。并且，还为处理

文件和目录以及它们映射到辅助存储器设备的操作提供最基本的支持。虽然I/O设备的驱动程序负责处理许多特殊的细节问题，但是还需要操作系统来协调那些支持实现I/O系统各种功能的设备驱动程序的活动。

安全和保护

作为一个资源和进程管理者，操作系统还必须保证各种事件能够正确、公平和高效地工作。然而，资源共享会引发多次暴露的问题。例如，会出现非授权的访问和数据修改的可能性。因此，操作系统还应该担当资源的保护者，以确保某些“坏家伙”和“臭虫软件”不至于毁坏一些其他人都需要的东西。并发的进程之间必须实施保护，而且还要在所有操作系统进程与所有用户进程之间实施保护。如果没有这类保护措施，某个用户程序有可能会擦除掉操作系统的某些代码，比方说处理中断的代码。对于多用户的计算机系统，我们需要增加额外的安全服务措施来保护共享资源（如存储器和I/O设备）和非共享资源（如个人文件）。存储器保护可以防止由于某个用户程序中的一个错误而影响到其他程序，或者是避免某个恶意程序控制整个系统。CPU保护可以确保用户程序不会陷于无限循环中而消耗掉其他作业任务所需要的CPU周期。

操作系统可以采用不同的方式来提供安全服务。首先，可以限制活动的进程只能在它们自己的存储器空间中执行。来自各个进程的所有I/O请求和其他资源的服务请求都要通过操作系统传送，然后由操作系统处理这个请求。操作系统将在用户模式下执行大多数命令，而其他命令则在内核模式下执行。这样，系统的资源就受到保护，非授权用户不能访问。同时，操作系统还提供了某些方法来控制用户的访问，通常是通过登陆名和密码访问的方式。如果限制进程在一个单一的子系统或者分区中运行，则可以提供更强的保护。

8.3 保护环境

为了提供保护，多用户操作系统要防止出现进程在系统中无节制运行的情况。进程的执行必须与操作系统和其他进程隔离开来。必须对共享资源的访问进行控制和协调，以避免产生冲突。在系统中建立保护屏障的方法有很多种。在这一部分，我们将讨论其中的三种：虚拟机、子系统和分区。

8.3.1 虚拟机

在20世纪50年代和60年代的分时系统中，不断遇到一些与共享资源有关的问题：例如存储器、磁盘存储器、读卡器、打印机和处理器的时钟周期等。这个时期的硬件系统还不能支持许多计算机科学家头脑中的解决方案。一个比较好的解决方案是，每个用户进程都有其自己的处理器，即一种虚拟机。这种虚拟机可以在一个真实的计算机系统内部与许多其他虚拟机和平共处。到了20世纪60年代末和70年代初，硬件的发展最终变得足够先进，可以把这种“虚拟机”思想应用在通用的分时计算机系统中。

从概念上来说，虚拟机非常简单。真实的计算机系统中的实际硬件都受到一个控制程序（controlling program，或内核）的专用命令的控制。控制程序可以创建一个任意数目的虚拟机。这些虚拟机在这种内核环境下执行，就好像它们是一些普通的用户进程，受到类似于运行在用户空间的程序所受到的相同限制。这个控制程序所展示出来的每个虚拟器的图像非常类似于实际计算机的硬件。每个虚拟机所看到的都是一个由CPU、寄存器、I/O设备和（虚拟）存储器组成的环境，而且这些资源就像是这个虚拟机所专用的。因此，虚拟机是一种反映完整系统资源的假想的机器。如图8-1所示，在虚拟机范围内执行的一个用户程序可以访问为虚拟机定义的任何系统资源。例如，当某个运行在虚拟机中的程序要调用一个系统服务来将数据写入磁盘时，这个程序会执行一个就像在实际机器上运行时的相同的系统调用。虚拟机会接收I/O请求，并将请求传送给控制程序，以便在实际硬件上执行。

我们可以在一个虚拟机上运行一个不同于内核操作系统的操作系统，也可以在每个虚拟机上运行

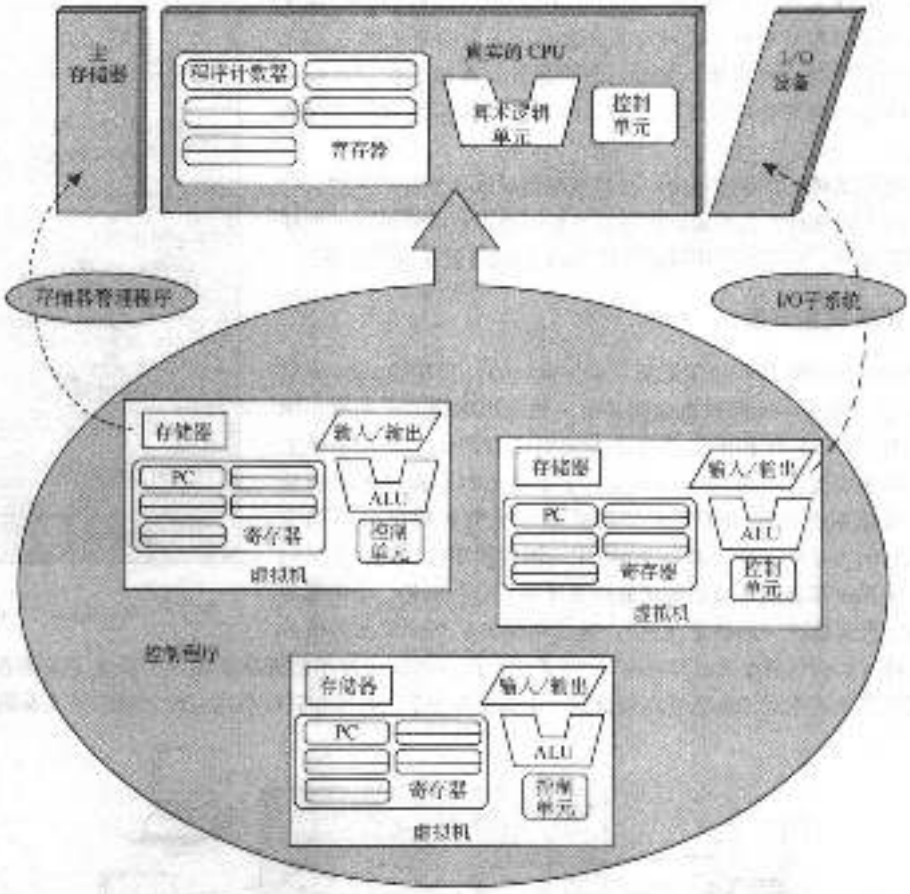


图 8-1 在一个控制程序下运行的虚拟机的图像

一个彼此都不相同的操作系统。事实上，通常的情况就是如此。

如果在 Microsoft Windows（从 Windows 95 到 Windows XP）上打开一个“MS-DOS”命令符窗口，这就已经实例化了一个虚拟机的环境。这些 Windows 版本上的控制程序被称为 Windows 虚拟机管理程序（VMM）。VMM 是一个 32 位保护模式的子系统（参见下一节）。VMM 负责创建、运行、监控和终止一个虚拟机。VMM 在开机时就被装载到内存中。当通过命令接口调用时，VMM 将创建一个运行在模拟 16 位 Intel 8086/8088 处理器的图像下的“MS-DOS”机器。尽管实际的系统有更多的寄存器（32 位），但是在 DOS 环境中执行的任务却只能够看到有限个数的具有 8086/8088 处理器特征的 16 位寄存器。VMM 控制程序首先将 16 位指令转换（或者用虚拟机的术语来说，进行形式转换）为 32 位指令，然后再在实际的系统处理器中执行。

为了服务硬件中断的需要，只要 Windows 启动时，VMM 就会自动装入一组预先定义的虚拟设备驱动程序集（virtual device drivers，VxD）。VxD 能够仿真外部的硬件设备或者仿真一个通过特权指令访问的编程接口。VMM 和 32 位保护模式下的动态链接库（将在第 8.4.3 节中解释）一起协同工作，允许虚拟设备截取中断和错误。这样，VMM 就可以控制一个应用程序对硬件设备和系统软件的访问。

当然，虚拟设备使用虚拟存储器。虚拟存储器必须和操作系统的存储器以及在系统中运行的其他虚拟机的存储器相共存。Windows 95 的存储器的地址分配图如图 8-2 所示。赋予每个进程介于 1MB 和 1GB 之间的专用地址空间。专用地址空间不能被其他进程访问。如果有某个非授权的进程试图使用被保护的另一个进程或者是操作系统的存储器（内存），将会发生一个保护错误（protection fault），计

算机会通过蓝屏信息发出警告。当然，系统还提供共享存储器区间，允许这里的数据和程序代码可以在多个进程中共享。除了 DDL 的部分对所有的进程开放外，图中上面的部分将保持系统虚拟机的组件。图中下面的部分是不可寻址的，这部分用作检测指针错误的一种方法。

现代操作系统支持虚拟机时，这些系统可以提供更好的保护、安全性和大型企业级计算机所要求的易管理性。虚拟机同样具有多种硬件平台的兼容性。第 8.5 节中所描述的 Java 虚拟机就是这类机器。



图 8-2 Windows 95 内存图

8.3.2 子系统和分区

Windows VMM 是一个子系统（subsystem），它在 Windows 引导时就启动。Windows 同样也启动其他一些专用功能的子系统，例如文件管理、I/O 操作和配置管理等。子系统会创建一些在逻辑上可以进行单独配置和管理的不同环境。子系统运行在操作系统内核的上面，操作系统内核为子系统访问基本系统资源提供必要的支持，例如访问多个子系统必须共享的 CPU 调度程序。

每个子系统都必须在控制系统的语境环境下进行定义。子系统的定义包括：资源描述（如磁盘文件）、输入和输出队列，以及其他各种硬件组件（如会话终端和打印机等）。子系统下面的内核通常不能直接看到为子系统定义的各种资源，而是要通过子系统本身才能看到这些资源。图 8-3 为一个子系统和其他系统资源之间相互关系的原理图。

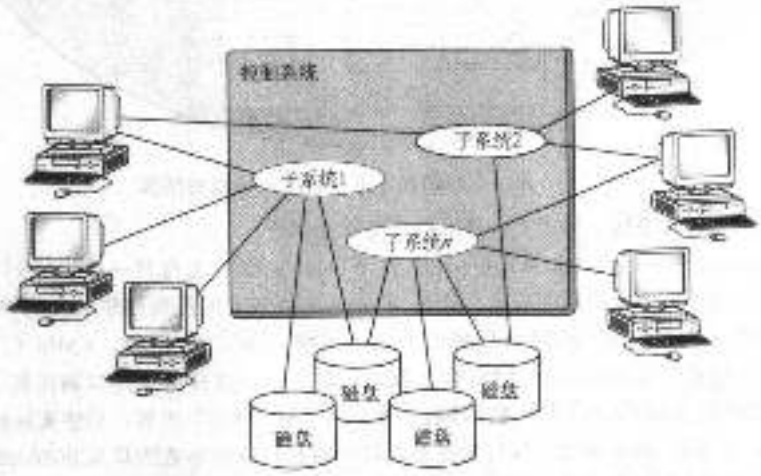


图 8-3 可以为多个子系统定义的单个资源

子系统可以帮助管理高度复杂的大型计算机系统的各种活动。因为每个子系统都是系统本身的一个分立的可控制实体，所以系统管理员能够单独启动和停止各个子系统，而不会干扰操作系统的内核或者是其他任何正在运行的子系统。每个子系统都能够通过重新分配系统资源来单独进行调节，例如增加或删除磁盘空间或存储器空间。而且，如果一个进程脱离了其个子系统的控制，或者是损坏子系统本身，通常只有该进程所运行的子系统会受到影响。因此，子系统不仅使得各个系统更加容易管理，而且也使得这些系统更加牢固可靠。

在一些非常大的计算机系统中，子系统在对机器和系统资源的分割上不能满足要求。有时，为了方便资源管理和提供安全保障，系统需要构筑一个更加完善和复杂的屏障。在这类例子中，系统可能会被分解为多个逻辑分区（logical partition），有时称为 LPAR，如图 8-4 所示。LPAR 是在一个物理

系统中创建多个可区分的机器，而且这些机器彼此之间并没有蕴含任何可以共享东西。一个分区的资源不能访问另外一个分区，就像这些分区都是独立运行在不同的物理系统上。例如，如果一个系统有两个分区 A 和 B，那么只有当这两个分区双方都同意在它们之间建立某种相互共享的资源时，如某种管道或信息队列，A 区才能从 B 分区中读取一个文件。一般来说，在逻辑分区之间的文件复制只能够通过使用某种文件传输协议或者是由系统供应商专为此目的而编写的应用程序来实现。

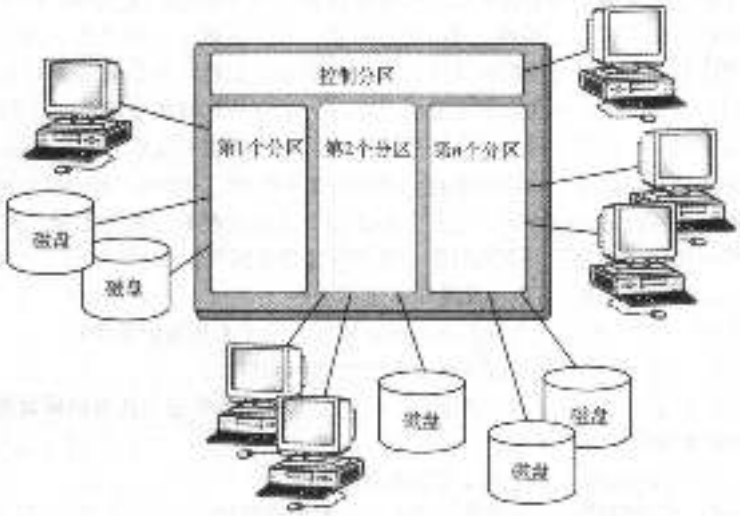


图 8-4 逻辑分区和它们的控制系统：资源不能轻松地在分区间共享

逻辑分区在为用户调试或者测试新程序创建的“沙盒”环境时特别有用。沙盒环境的名字来源于这样的想法：只要游戏是在沙盒范围内进行，使用这种沙盒的人就可以随意地按自己内心的想法来“玩耍（play around）”。沙盒环境对访问系统资源设置有严格的限制。在某个分区中运行的各个进程决不可以有意或无意地访问驻留在其他分区内的数据或进程。分区通过对系统中的各种资源实施隔离，避免这些资源被那些未授权的进程访问，这样就增加了系统的安全性。

虽然子系统和分区在如何定义它们的组成资源方面彼此不同，但是我们可以将它们想像为一个计算机系统的分层体系结构中的子模型结构。在分区环境中，这些子模型结构的层次看起来就像是一个相邻的多层生日蛋糕，从机器硬件层一直扩展到应用层。另一方面，子系统相互之间的区别并不十分明显，这些子系统的主要差别大部分发生在系统软件的层面。

8.3.3 保护环境和计算机系统体系结构的发展进程

直到最近，虚拟机、子系统和逻辑分区都被认为是“老技术”的大型计算机系统的产物。整个 20 世纪 90 年代，人们普遍相信小型计算机系统要比大型计算机系统具有更好的性价比。“客户机-服务器（client server）”的范式被认为是具有更好的用户友好界面和能够更好地响应动态商业环境。对小型计算机的应用程序的开发工作很快吸引了一批天才的编程人员。不久，人们发现像文字处理和日程管理这类的办公自动化程序，可以更平滑地运行在由多个小型文件服务器所支持的协作网络的工作环境中。利用打印服务器来控制支持网络的激光打印机，可以在普通打印纸上产生清晰干净的输出结果。而且，这些激光打印机的速度要比大型计算机系统中使用的、以特殊形式输出的和效果很差的线性打印机要快得多。事实证明，桌面型计算机和小型服务器系统可以为用户提供与大型计算机系统所能提供的相同的原始计算能力和方便性，而成本却只有大型计算机系统的一小部分。当然，原始计算能力只是一个企业计算系统的一部分功能。

当办公自动化系统移到办公桌面时，人们构建了办公室网络以便把桌面型计算机系统相互连接起

来，并且与用做文件档案和其他重要商业记录的存储库的文件服务器连接在一起。后来，人们使用了应用程序服务器来负责运行那些执行核心商业管理功能的程序。当各大公司开始支持因特网连接时，人们就在网络上增加了电子邮件功能和 Web 服务器。如果在网络上有任何服务器发生处理故障，当时最简单的解决方案是增加另一台服务器来分担这台服务器的负载任务。到 20 世纪 90 年代末期，大型企业都拥有巨大的服务器站（server farm），可以支持成百上千台独立的服务器处于环境上受控的和安全可靠的机制下工作。很快服务器站就开始变得需要耗费相当可观的人力资源，因为每台服务器都需要不定期的大量维护工作。每台服务器中的内容都必须备份到磁带上。为了安全起见，这些磁带随后被转移下来作为离线保存。由于每台服务器都有失效的潜在可能性，所以故障诊断和添加补丁程序成为服务器维护人员的日常任务。不久，事实证明小型的、便宜的计算机系统不是人们开始时所希望获得的那种产品。这一点对于需要支持上百个小型服务器系统的企业来说是千真万确的。

每个主要的企业计算机制造厂商现在都提供一种服务器整合（server consolidation）的产品。不同制造商对于这个问题有不同的解决方案。这些方案中最令人感兴趣的一种思想是，在一个单一的大型计算机上创建一些包含许多虚拟机的逻辑分区。服务器整合有如下优点：

- 管理一个大系统要比管理大量的小系统更加方便一些。
- 单个大系统消耗的电力要比一大群具有同样计算能力的小系统的耗电少。
- 用电越少，产生的热量就越少，这样就会节约空调的使用。
- 大型计算机系统能够提供更好的容错保护，因为大系统中通常包含热备份磁盘和多个处理器。
- 单个的系统更容易进行备份和恢复。
- 单个系统的占地空间较小，可以减少房屋使用的成本。
- 单个大系统的软件使用版权费用要比大量的小系统的费用低。
- 对于单个系统来说，在对系统软件和用户程序软件进行升级时所需要的人力要比多系统少。

大型计算机系统的供应商，如 IBM、Unisys 和 Hewlett-Packard 等，很快抓住了这种服务器整合的机会。IBM 公司将他们的大型机和中型机的生产线改造成为 E 系列（eSeries）的服务器的生产线。System/390 系列大型计算机改成了 Z 系列（zSeries）的服务器。Z 系列的服务器支持 32 个逻辑分区，而每个运行 IBM 虚拟机（VM）操作系统的逻辑分区可以定义上千个虚拟的 Linux 系统。图 8-5 为一个 eSeries /Linux 模型的配置示意图。就像一个独立的 Linux 系统一样，每个虚拟的 Linux 系统都同样能够支撑企业级的应用程序和电子商务活动，但却不需要任何管理费用。因此，原来一个足球场大小的服务器站现在可以由一个 Z 系列“机箱”所替代，这个“机箱”的尺寸通常比家用电冰箱稍微大一些。可以说，服务器整合技术的进步过程是贯穿操作系统演变发展的一个缩影。计算机系统制造商通过应用不断进步的计算机资源，一直在努力使他们制造出来的系统变得更加容易管理，同时系统的功能也变得越来越强大。

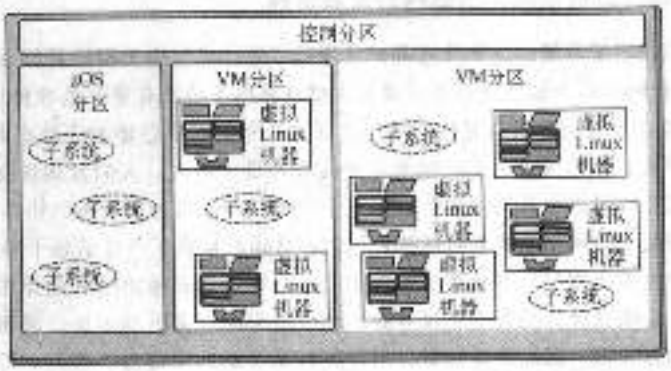


图 8-5 IBMZ 系列服务器的逻辑分区中的 Linux 机器的配置图

8.4 编程工具

操作系统及其应用程序的集合为编程用户和运行程序的系统提供了一个相互联系的接口。其他一些实用程序，或者称为编程工具，对于实现更加机械的软件开发是十分必要的。我们将在下面的章节中讨论有关编程工具的内容。

8.4.1 汇编程序和汇编

在我们讨论的计算机系统的分层体系结构中，直接在操作系统层上面的一个层是汇编语言层。在第4章中，我们提出了一个非常简单的，称为MARIE的假想的计算机体系结构。这个结构过于简单，实际上并没有使用这种结构的机器。首先，这个假想的体系结构需要不断地从存储器中提取操作数，将使得这个系统变得非常慢。实际的计算机系统都通过采用足够数量的可寻址的片内寄存器，来将存储器的提取次数减至最小。另外，任何实际计算机中的指令系统结构都要比MARIE机器丰富得多。许多微处理器的指令集中有超过1000条不同的指令。

尽管我们介绍的机器与实际的计算机有着非常大的区别，但是这里所讨论的汇编过程却是相同的。实际上，现在人们使用的每一种汇编程序，都要通读源代码两次。第一次通读构建一个符号表，并且汇编得到尽可能更多的代码；第二次通读是使用从第一次通读时构建的符号表中取得的地址值，来完成二进制指令的转换。

大多数汇编程序的最终输出结果是一段可重新定位（或者说具有浮动地址，relocatable）的二进制指令流。操作系统可以将程序代码装载到它想要存放的存储器的任何地方。如果操作数的地址与程序装入内存中位置有关，则二进制代码可以进行重定位。例如，从表4-5中取如下的MARIE代码：

```
    Load x
    Add y
    Store z
    Halt
x, DEC 35
y, DEC -23
z, HEX 0000
```

汇编后的代码输出看起来可能类似如下：

```
1+004
3+005
2+006
7000
0023
FFE9
0000
```

例中的“+”不能取字面的意义。它发信号通知程序装载器（操作系统的一个组件）。表示第一条指令中的004是相对于该程序的起始地址。下面，我们讨论一下如果程序装载器正好将该程序存放到250h开始的内存中，会发生什么情况。表8-1给出了这种情况下的程序代码在内存中的图像。

如果碰巧认为把程序装载到从地址400h开始的内存中会更好，那么内存图像看起来类似表8-2所示。

与可重新定位的代码相比，绝对代码（absolute code）是一种必须存放在内存中的某个特定位置的可执行二进制代码。不可重新定位的代码是某些计算机系统中具有一些特殊用途的代码。通常，这些特殊应用包括附属设备的显式控制或系统软件的操作处理，其中我们总是可以在某些明确界定的存储单元中找到一些特殊的调度程序。

当然，二进制机器的代码不能通过符号“+”来区分可重新定位的代码和不可重新定位的代码。区分这两种代码的特定方式取决于执行这些代码的操作系统的设计。一种最简单的区分方法是对它们

表 8-1 从地址 250h 开始装入程序时的内存图像

地址	存储器内容
250	1254
251	3255
252	2256
253	7000
254	0023
255	FF89
256	0300

表 8-2 从地址 400h 开始装入程序时的内存图像

地址	存储器内容
400	1404
401	3405
402	2406
403	7000
404	0023
405	FF89
406	0300

使用不同的文件类型（扩展名）。MS-DOS操作系统使用.COM（命令文件）扩展名来表示不可重新定位的代码。而使用.EXE（可执行文件）来代表可重新定位的代码。COM文件总是从地址100h开始装入。EXE文件则可以被存放到存储器的任意地方，甚至还可以不占用连续的内存空间。可重新定位的代码和不可重新定位的代码也可以通过下面一种方法来加以区分：即为所有可执行的二进制代码预先加上前缀或者是引导信息。这样，当程序装载器从磁盘中读取程序时，就可以让它知道其中的选择。

当可重新定位的程序代码被装到内存时，通常会有一个专用寄存器为该程序提供一个基地址。随后，程序中所有地址都会被认为是偏离存放在该寄存器中的基地址的偏移量。在表8-1中，说明了程序装载器是如何从地址0250h开始装入程序代码的。这里，实际的系统会简单地把0250存放在程序的基地址寄存器中，并且直接使用这个程序而无需修改。如表8-3所示。显然，在加上基地址寄存器中的数值0250后，每个操作数的地址都变成有效地址。

表 8-3 使用基地址寄存器从地址250h开始装入程序时的内存图像

地址	存储器内容
250	1004
251	3005
252	2006
253	7000
254	0023
255	FF89
256	0300

不管是使用可重新定位的代码还是不可重新定位的代码，程序的指令和数据都必须与实际的物理地址关联（或绑定）。将指令和数据绑定到存储器地址的过程可以在编译时，装载时，或者执行时进行。绝对代码就是编译时绑定（compile-time binding）的一个例子。这里，指令和数据的引用在程序编译时就和物理地址绑定在一起。只有在预先知道一个进程映像（即程序在存储器中的存放图像）被如何装载到存储器单元的情况下，编译时绑定才会起作用。然而，对于编译时绑定而言，如果进程映像的起始位置发生改变，那么必须重新编译整个程序的代码。假如编译时我们并不知道进程映像装入存储器的具体位置，那么就要生成可重新定位的代码。这种代码可以在装载时或者是运行时绑定实际的物理存储器地址。在将二进制代码模块装载到存储器时，对于每次代码地址的引用，装载时绑定（load-time binding）都会添加进程映像的起始地址。但是，由于整个进程的起始地址必须保持相同，所以这种装载时绑定的进程映像在执行过程中不能移动。运行时绑定（run-time binding），或者称为执行时绑定（execution-time binding），是指直到进程实际运行时才对程序代码的物理地址进行绑定。运行时绑定允许进程映像在运行时可以从一个内存地址转移到另一个内存地址。运行时绑定需要有专门的硬件支持地址映射（address mapping），或者说是将逻辑地址转换为物理地址。这里，需要采用一个专门的基地址寄存器来存放程序的起始（首）地址。然后，在CPU所生成的每次引用中都会加入这个起始地址。如果某个进程映像发生了移动，就会更新基地址寄存器的内容来反映该进程起始地址的变化。如果要快速实行这种地址转换操作，必须增加额外的虚拟存储器的硬件。

8.4.2 链接编辑器

在大多数计算机系统中，程序编译器的输出结果必须首先通过一个链接编辑器（link editor，或称为目标代码链接器，简称链接器，linker）进行通读，然后才能在目标系统上执行。链接过程是将程序的外部符号与来自其他文件的所有输出符号进行匹配，生成一个不包含未解析的外部符号的单一二进制文件。链接器的主要工作是将相关的程序文件组合成一个统一的可装载的模块，如图 8-6 所示。图中的举例使用的是 DOS/Windows 环境下的文件扩展名。构成模块的各个二进制文件可以是完全由用户编写的（user-written），也可以是由标准的系统例行程序组合而成的。这主要取决于应用程序的需求。另外，二进制链接器的输入可以由任何一个编译器产生。除此之外，链接过程还允许程序的不同段落部分使用不同编程语言来编写。因此，为了编码方便起见，程序的某一部分采用 C++语言编写，而另一部分可以使用汇编语言编写，这样能够加速执行速度特别慢的程序部分。

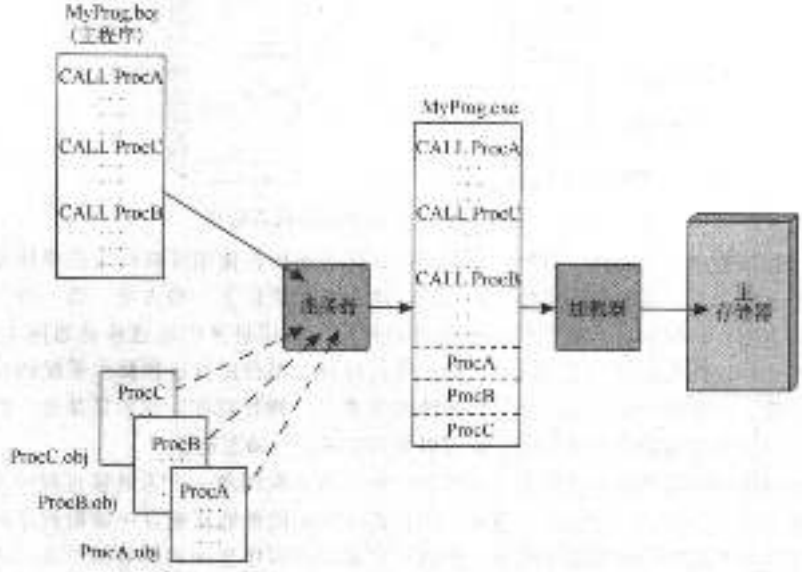


图 8-6 链接和装载二进制代码模块

在使用汇编程序时，大多数链接编辑器都需要两次通读才能生成包含所有外部输入模块的一个完全的装载模块。在第一次通读过程中，链接器生成一个全局外部符号表，表中包含每个外部模块的名称和这些外部模块关于整个链接模块开始处的相对起始地址。在第二次通读过程中，所有模块（原来这些模块都是分立的外部模块）之间的引用会被来自符号表中的这些模块位移量所取代。在链接器的第二次通读过程中，与平台相关的代码也会被加入这个组合模块中，生成一个统一的和可装载的二进制程序文件。

8.4.3 动态链接库

有些操作系统，例如著名的 Microsoft Windows，在生成可执行模块之前并不需要对程序中使用的所有过程都进行链接编辑。如果在源程序中使用适当的程序语法，某些特定的外部模块就可以在执行时再进行链接。因为这种链接过程只是在程序或模块的首次调用时进行，所以这类外部模块被称为动态链接库（dynamic link libraries，DLL）。动态链接过程的示意图如图 8-7 所示。当每个过程被装入后，过程的地址就会被放置到位于主程序模块中的交叉引用表（cross-reference table）中。

这种方法有很多好处。第一，多个程序反复使用某个外部模块，那么静态链接会要求每个引用的程序都要包含这个外部模块的二进制代码的一个副本，一段相同程序代码有多份副本分散在各个程序之中，显然是对磁盘空间的一种浪费。因此，在执行时进行链接的方式将节省磁盘空间。动态链接的第二个好处是：如果外部模块的代码内容发生了变化，那么每个与这个外部模块链接的模块都不需要

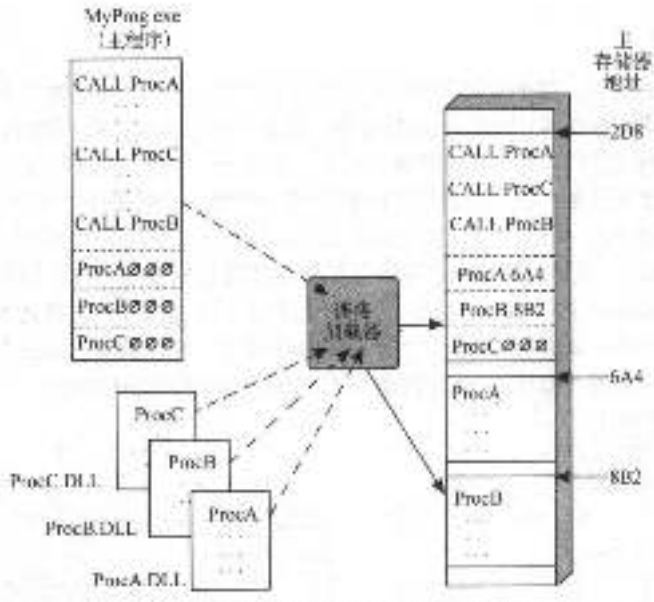


图 8-7 带有装载时地址解析的动态链接

重新进行链接来保持程序的完整性。另外，要跟踪记录哪些模块在使用哪些特定的模块是非常困难的事情，对于大型系统来说，甚至是不可能的。第二，动态链接提供了一些方法，第三方可以利用这些方法来创建一些通用库。例如由为某个特定系统编写程序的程序员来生成这样的通用库。换句话说，如果你正在为某个特定的操作系统编写程序，那么可以授权在每台运行这种操作系统的计算机上使用某些特定的通用库。你自己不必关心这个操作系统的版本号、修补程度，或者是其他一些可能经常发生变化的东西。只要没有删除这个通用库，就可以使用它来进行动态链接。

动态链接过程可以在装载程序或者某个正在运行的程序首次调用一个未链接过程时发生。在装载时进行动态链接会造成程序启动的延迟。这时，操作系统不是简单地从磁盘中读取程序的二进制代码并运行它。现在操作系统不仅要装载主程序，而且还要装载该程序使用的所有模块的二进制代码。在执行第一个程序语句之前，装载器（程序）要为主程序提供每个模块的装载地址。对于某些应用程序来说，如果在用户调用该程序和程序实际开始执行之间存在着时间上的延迟是不能接受的。另一方面，运行时链接（run-time linking）不会导致装载时链接（load-time linking）的启动开销，即启动时的时间延迟，因为一个模块仅仅在被调用时才会被链接。这样就可以节省大量的工作，因为相对而言实际上只调用为数不多的模块。然而，当一个程序频繁地停顿下来去装载某些库程序时，有些用户却拒绝去感知这种不规则的反应时间。

有关动态链接的另外一个不太明显的问题是，编写模块的程序员不能直接控制动态链接库程序的内容。因此，如果链接库代码的编写者决定要改变其功能，可以进行这种修改过程而无需得到库的使用者的认知或同意。此外，任何编写过商业程序的人都知道，这些库程序的少许变化可能会在整个系统中引起连锁反应，这些反应的后果可能是破坏性的，而且很难追查它的根源。幸运的是，这种奇怪的现象非常少。因此，在各种操作系统上发布商业的二进制代码程序时，使用动态链接仍然是一种大家所喜爱的方法。

8.4.4 编译器

汇编语言编程可以完成许多高级编程语言不能实现的功能。首先最重要的一点是，汇编语言允许程序员直接访问下面的计算机硬件结构。用于控制的程序和与外设进行通信的程序，通常都是用汇编语言编写的，原因是汇编语言通常具有一些高级语言所没有的专用指令。例如，程序员可以不依赖操

作系统的服务直接控制通信端口。使用汇编语言，可以让计算机做任何事情，甚至是操作系统所有提供的服务。尤其是程序员常常使用汇编语言来操作一些专用硬件，因为高级语言的编译器并不是设计来处理不通用或者是不常用的设备。而且，精心编写的汇编代码的运行速度极快。每条原始指令都可以进行精心优化，这样汇编指令可以在系统上产生最适时和最有效的行为。

但是，尽管汇编语言具有这些优点，但是在普通的应用程序开发中，并不鼓励人们使用汇编语言。事实上，使用汇编语言编程非常困难，而且容易出错。汇编语言程序的维护甚至要比程序的编写更加困难，特别是如果负责维护的程序员不是程序代码的编写者时。最重要的是，汇编语言不能移植到不同的机器结构上。因为这些原因，大多数通用系统软件都很少使用或者不使用汇编指令。汇编代码仅仅是在非常必需时才使用。

现在，所有系统程序和应用程序实际上几乎都使用高级语言，很少有例外。当然，“高级（higher-level)”是一种相对的术语，常常容易产生误解。一种大家都可以接受的编程语言分类法是从将二进制的机器代码称为“第一代（first-generation)”的计算机语言（1GL）开始的。在使用第一代计算机语言（1GL）之前，程序员需要利用计算机控制台上的连接开关直接把程序指令输入计算机。后来，越来越多的“特权（privileged)”用户们把二进制指令打孔到纸带或卡片上。在 20 世纪 50 年代早期，出现了第一个汇编程序，这大大地提高了编程效率。这些第二代语言（2GL）消除了由于使用人工方法将指令翻译成机器代码而产生的错误。到了 20 世纪 50 年代后期，引进了可以编译的符号语言，或称为“第三代(third-generation)”语言（3GL)，从而再次大幅度提高了编程的效率。在 1957 年，John Backus 和他的 IBM 团队发布的 FORTRAN（Formula Translation）语言是第一个第三代语言。从此，真正的字母类第三代语言在程序设计领域大量涌现。在这些第三代语言中，有些是使用首字母来命名的，例如 COBOL、SNOBOL 和 COOL 语言。而有些是以人物来命名的，例如 Pascal 和 Ada 语言。另外，还有一些第三代语言是根据语言设计者的喜好来称呼的，例如 C 语言、C++语言和 JAVA 语言。

“每一代”程序设计语言都与当时人们如何思考问题和机器如何解决这些问题密切相关。有些第四代和第五代编程语言非常容易使用，以至于现在的程序设计任务已经不再像以前那样要求有经过训练的专业程序员，普通的计算机最终用户就可以很方便地完成。这其中关键的思想是：用户只需要告诉计算机做什么，而不再需要告诉计算机如何去做，因为接下来就可以由编译器来完成剩下的工作。为了方便用户使用，最新的程序设计语言为计算机系统提供了更多的管理开销。最后，由于实际工作的数字系统硬件只能执行二进制代码，因此所有指令都必须通过这种语言分层结构向下转换成对应的机器语言指令。

在第 4 章中，我们曾经指出汇编语言的语句与机器实际运行的二进制代码之间存在一一对应的关系。而在编译语言程序中，这变成了一种一对多的对应关系。例如，考虑变量存储定义，对于高级语言语句：x= 3* y，在 MARIE 模型的汇编语言中将至少需要用 12 条语句来表示。随着源程序语言的复杂度的增加，源代码指令数目和对应的二进制机器指令数目的比率变得越来越小。语言的层次“越高”，每条高级语句所对应生成的机器指令数就越多。编程语言层次结构中的这种关系如图 8-8 所示。

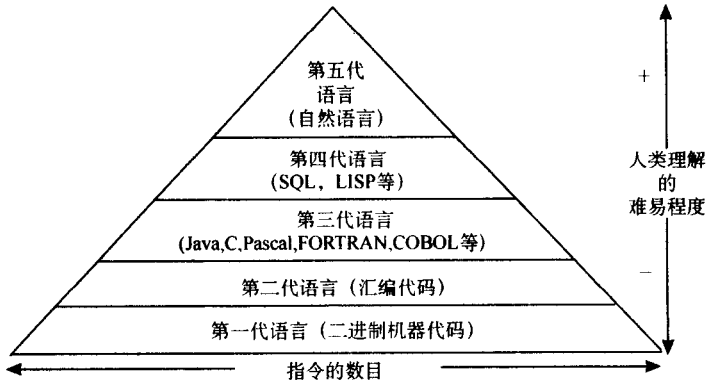


图 8-8 编程语言层次结构

自从 20 世纪 50 年代末出现了第一个编译器后，编译器技术得到了迅速发展。通过编译器结构的发展成就，软件工程科学已经证明编译器能够把看似非常棘手的问题变成普通程序设计任务。这种问题的复杂性主要在于如何连接人们理解的程序语言与机器理解的程序语言之间所存在的语义上间隙。

大多数编译器都使用一个 6 阶段（six-phase）过程来实现这种代码转换，如图 8-9 所示。代码编译的第一步为**词法分析**（lexical analysis），目的是从一个文本结构的源代码流中抽取一些有意义的语言原语，或者是标记符号（token）。这些标记符号由特定的编程语言中的保留字（例如 if、else）、布尔操作符和数学操作符、文字（例如 12.27）和程序员定义的变量组成。当词法分析器在创建标记符号流时，也同时在为符号表构建框架。这时，符号表中很可能包含用户定义的标记符号（变量和过程名），以及说明这些标记符号位置和数据类型的注解。如果在源代码中发现有该语言不能识别的字符和结构时，就会出现词法错误。例如，程序员定义的变量 1Dayspay，在大多数编程语言中将会发生词法错误，因为变量名通常不能以数字开头。如果没有发现词法错误，编译器将继续分析标记符号流的语法问题。

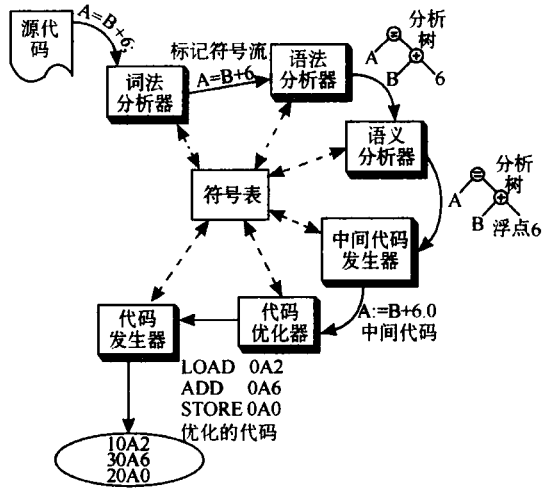


图 8-9 程序编译的 6 个阶段

标记符号流的语法分析（syntax analysis 或 parse）的过程包括创建一种称为**分析树**（parse tree，或称为**语法树**，syntax tree）的数据结构。无序遍历某个分析树通常可以给出刚遍历过的表达式。例如，考虑下面的程序语句：

```
monthPrincipal = payment-(outstandingBalance * interestRate)
```

这个语句的一个正确的语法树如图 8-10 所示。

语法分析器会检查符号表中现有的位于分析树上的程序员定义的变量。如果分析器发现一个在符号表中没有其描述信息的变量，会发出一个出错信息。语法分析器同样也会检查不合法的结构，例如 $A=B+C=D$。然而，语法分析器不会检查“=”或“+”这些操作符对于变量 A、B、C 和 D 是否正确有效。这些工作将在下一阶段由**语义分析器**（semantic analyzer）完成。语义分析器使用语法分析树作为输入，并利用从符号表中读取信息来检查输入的数据类型是否正确。语义分析器也可以进行正确的数据类型升级。例如，将一个整型数据转换为浮点数值或变量，条件是只要语言规则支持这种转换即可。

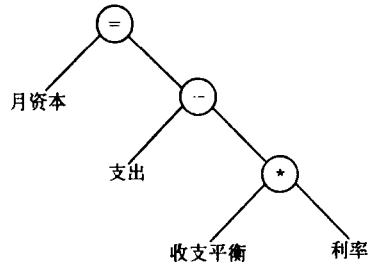


图 8-10 语法分析树

在完成分析功能后，编译器将使用来自语义分析阶段的语法树开始其综合阶段。代码综合的第一步是从语法树中创建**伪汇编**（pseudo-assembly）代码。这种代码通常称为 3 地址代码（three-address code），因为它支持像 A= B+ C 这类的语句，而大多数汇编语言都不支持这种语句。这种中间代码使得编译器可以在许多种不同的计算机系统上移植。

一旦所有的标记符号流分析、构建分析树和语义分析的工作完成后，下一步的工作变得相对容易一些。主要是编写一个 3 地址的代码翻译器，它可以为许多不同的指令集产生一个输出。大多数计算机的指令系统（ISA）都使用 2 地址指令代码，所以在代码的转换过程中必须解决编址方式的差异问题。读者可以回忆一下，MARIE 模型中的指令集是一个 1 地址的体系结构。然而，编译器在最后阶段的工作通常不仅仅是把中间代码翻译成汇编指令代码。好的编译器会尝试对指令代码进行优化。在优

化过程中，编译器通常可以考虑不同的存储器和寄存器的组织结构，并提供执行任务所需的功能最强大的指令。代码优化过程同样也会涉及到移除不必要的临时变量，将重复的表达式整合为一个单一表达式，而且还要标记无效（不可到达）的死代码。

在完成所有的指令生成和可能的优化作业后，编译器会生成适合在目标系统上进行链接和运行的二进制目标代码。

8.4.5 解释器

与编译语言一样，解释语言在源代码和可执行的机器指令之间也存在着一对多关系。但是，与编译器不同的是，解释程序一次只处理一条源代码的程序语句。而编译器却在产生一个二进制的代码流之前，需要通读整个源代码文件。

正因为有如此多的工作都是“即时（on the fly）”完成的，所以解释程序的工作速度要比编译器慢得多。编译器所要求的6个工作步骤中至少有5个步骤同样也必须在解释程序上执行，而且这些步骤都是被“实时”地执行。这种方法不可能提供代码优化的过程。另外，解释程序中的错误检测通常也仅限于语言语法和变量类型的检查。例如，很少有解释程序可以在错误发生前检测出可能的非法算术运算，或者能够在变量值超越数组界限之前就提出警告。

一些早期的编译器，如某些著名的BASIC编译器，在定制的编辑器中提供了语法检查功能。例如，如果用户将“else”输入成“esle”，编辑器会立即对这一输入结果做出标记。另外一些解释程序允许用户使用普通的文本编辑器编辑程序，最后等到程序执行时才进行语法检查。后一种方法在用于一些关键性的商业应用程序时会有很大风险。如果应用程序恰好执行到一段尚未进行正确的语法检查的代码分支结构时，该程序会发生崩溃。而留给用户的可能是一个他所不愿意看到的系统命令提示符，这时用户的文件可能只更新了部分内容。

虽然具有很慢的执行速度和延期的出错检查的缺点，但是我们还是有很多理由使用解释语言。其中最重要的一点是：解释语言允许用户进行源程序级（source-level）的调试，这样解释语言非常适合于编程初学者和最终用户。这也就是在1964年两位英国Dartmouth的教授John G. Kemeny和Thomas E. Kurtz为什么发明了BASIC语言的原因。BASIC即为*初学者通用符号指令代码*（Beginners All-purpose Symbolic Instruction Code）。那时，学生们通常的第一个编程经历是将FORTRAN程序指令穿孔到一些80列的卡片上。然后，再把这些卡片放到一个大型计算机的编译器上运行，通常这一过程需要几个小时。有时，为了得到一个明确的编译和执行结果甚至需要几天。与采用批处理模式编译程序的方式完全不同，BASIC语言允许学生在一个交互式终端的会话环境中键入程序代码。一直在主机上运行的BASIC解释程序会立即对学生的操作给出反馈信息。这样，学生能够迅速纠正各种语法错误和逻辑错误，从而提高学习的积极性和效率。

基于这些相同的理由，BASIC语言是早期个人计算机系统上优先选择的一种编程语言。当时，许多在第一时间购买计算机的用户都不是有经验的编程人员，所以他们需要一种可以轻松学会计算机编程的语言。BASIC语言就是能够满足这种目的的一种理想选择。而且，在单用户的个人计算机系统上，很少有人会在意BASIC解释语言的执行速度比编译语言慢得多。

8.5 Java：一种综合语言

在20世纪90年代初期，James Gosling博士和他领导的Sun公司的研究团队开始研发一种可以在任意计算机平台上运行的编程语言。他们的目标是创造一种“编写一次，到处运行（write once，run-anywhere）”的计算机语言。1995年，Sun公司发布了Java编程语言的第一个版本。由于具有可移植性和开放的规范标准，Java语言变得非常流行。Java语言的程序代码几乎可以在从最小型的掌上设备到大型计算机系统的所有计算机平台上运行。Java语言是基于因特网的大范围商业活动开始出现时应运而生的跨平台语言。Java语言是一种理想的跨平台计算模型。虽然本书在第5章中简要介绍了Java

语言及其一些基本特性，但是在这里我们将进行更加深入的讨论。

对于初学习过 Java 编程语言的人来说，应该知道 Java 编译器的输出是一个二进制的类（class）文件。这种类文件可以由一个 Java 虚拟机（Java Virtual Machine，JVM）来执行，Java 虚拟机在很多方面类似于实际的计算机。Java 虚拟机拥有一个只能由该机器中运行的进程寻址的专用存储器空间。它还有真正（bona fide）属于它自己的指令系统。这个指令系统（ISA）是一个基于堆栈的体系结构。这种指令系统使得虚拟机变得非常简单，而且实际上它可以移植到任何计算平台上。

当然，Java 虚拟机并不是一个真实的计算机。事实上，它是位于操作系统和应用程序（二进制的类文件）之间的一个软件层。类文件包含各种变量和操作这些变量的方法（method）。

图 8-11 给出了 JVM 如何构成一个具有自己的存储器和方法区的虚拟计算机器。注意：存储器堆集、方法代码和本机方法接口（native method interface）区是由所有运行在该机器上的进程所共享的。

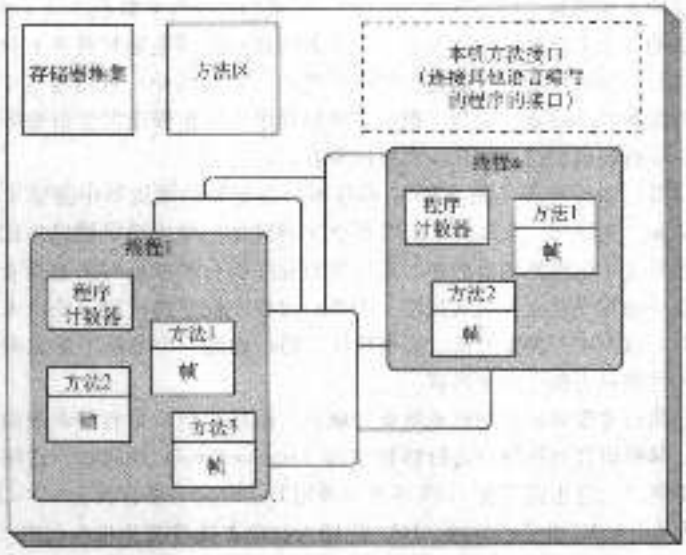


图 8-11 Java 虚拟机

存储器堆集（heap）是一个主存储器空间，当通过线程的执行而产生和消灭各种数据结构时会对这些存储器堆集进行空间分配和取消空间分配。Java 中取消堆集存储器的分配通常称为垃圾回收（garbage collection），垃圾回收由 JVM（代替操作系统）自动完成。Java 的本机方法区（native method area）为非 Java 的二进制对象提供工作场所，例如编译过的 C++，或者是汇编语言的模块。JVM 的方法区（method area）包括运行驻留在 JVM 中的每个应用线程所需的二进制代码。这也正是类变量数据结构和由类所要求的程序语句驻留的地方。Java 语言的可执行程序语句用一种称为字节码（bytecode）的中间代码来保存。这在第 5 章中也做了介绍。

Java 方法的字节码在各种不同线程中被行。有几个线程是由 JVM 自动启动的，其中包括主程序线程。在每个线程中一次只能激活一种方法，而多个程序可以产生多个额外的线程来并发执行。当一个线程调用某种方法时，会为该方法创建一个存储器帧。这个存储器帧的一部分空间用来保存该方法的局部变量，另一部分用作该方法的专用堆栈。在线程已经定义了方法堆栈之后，该线程就会把该方法的各个参数压入堆栈，并将程序计数器指向该方法中的第一条可执行语句。

每个 Java 类都包含一种称为常数池（或称为常数存储器，constant pool）的符号表。常数池是一个数组，其中包含某一类的各种变量的所有数据类型和各变量的初始值，以及各变量的访问标志（例如，对于这个类来说，该变量是公共变量还是专用变量）。常数池中还包含一些不是程序员定义的结构。这也就是为什么 Sun 公司将常数池（数组元素）中的入口条目称为属性（attribute）的缘故。在每个 Java 类的属性中，我们会发现一些 Java 处理内部事务的项目如 Java 源文件的名称，Java 类的继

承结构体部分和转向其他JVM内部数据结构的指针。

为了说明JVM如何执行方法字节码，我们考虑图8-12中的Java程序。

```
public class Simple {
  public static void main (String[] args) {
    int i = 0;
    double j = 0;
    while (i < 10) {
     i = i + 1;
     j = j + 1.0;
    } // while
  } // main()
} // Simple()
```

图8-12 一个简单的Java程序

Java需要一个命名为Simple.java的文本文件，其中保存这个类的源代码。Java编译器会读取Simple.java文件，并执行其他编译器要做的相同工作。Java编译器的输出是一个二进制的字节码流，命名为Simple.class。这个Simple.class文件能够由任何具有创建该类的编译器相同版本或更新版本的JVM运行。这些步骤如图8-13所示。

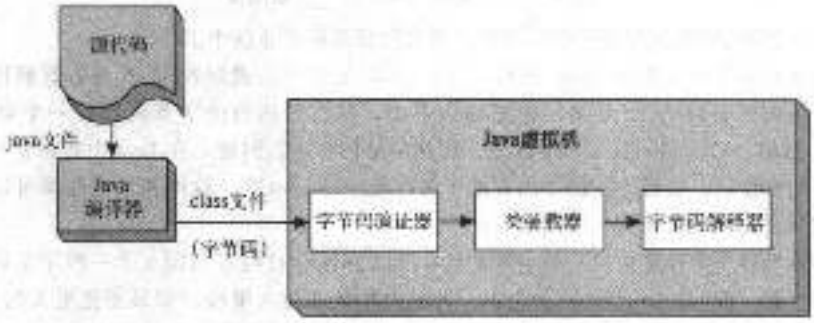


图8-13 Java类的编译和执行过程

在执行时，Java虚拟机必须在一个主机计算机系统上运行。当JVM装入一个类文件后，首先要验证文件的完整性，包括检查类文件的格式和字节码指令的格式，并且要确保没有不合法的引用。在这种初步的验证顺利完成后，装载器将字节码装入存储器，并执行一系列运行时的检查。

完成所有验证步骤后，装载器调用字节码解释器。解释器将有如下6个步骤任务：

1. 执行字节码指令的链接编辑，要求装载器提供所有被引用的类和系统代码（如果它们还没有被装入的话）。

2. 创建和初始化主堆栈帧和局部变量。

3. 创建和启动执行线程。

4. 在线程执行时，通过取消未使用的存储器分配来管理垃圾存储器。

5. 当每个线程终结时，取消该线程的资源分配。

6. 在程序终止时，取消所有剩余的线程并终止JVM。

图8-14给出了类文件Simple.class的十六进制的字节码的图像。将位于第一列（阴影）中的数值加到位于第一行（阴影）中的偏移量上，即可得到每个字节的地址。为了方便，我们已经把字节码翻译为字符。这里的二进制值具有一个有意义的7位ASCII值。我们可以看到命名为Simple.java的源文件是从地址06Dh开始。而类名始于地址080h，熟悉Java的读者都知道这个Simple类也被认为是.this类，而其超类（super class）是java.lang.Object，其名称开始于地址089h。

注意，这里的类文件以十六进制数CAFEBABE开始。这是指示一个类文件开始的魔术数字（magic number）。在这个魔术数字的后面紧跟着一个8字节序列，表示这个类文件的语言版本。如果

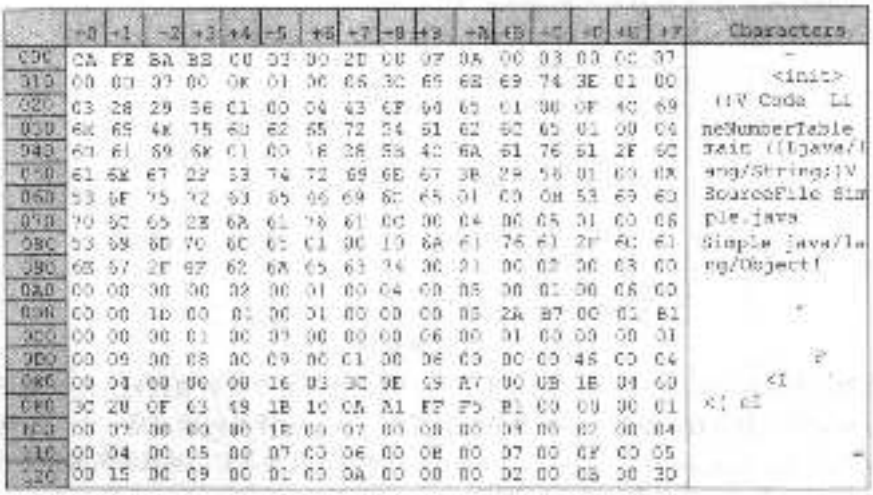


图 8-14 simple.class 的二进制图像

这个序列数大于解释 JVM 所能支持的版本号，那么验证器将中止这个 JVM。

这些可执行的字节码从地址 0E6h 开始。地址 3E5h 处的十六进制数值 16 是告诉解释程序这个可执行的方法字节码的长度为 22 字节。在汇编语言中，每个可执行的字节码都有一个对应的助记符(mnemonic)。目前，Java 语言定义了 204 个不同的字节码指令。因此，在 Java 中只需要一个字节来就可以表示所有的操作码。这种小的操作码有助于保持较小的 Java 类，这样使得这些类可以在主机系统上更快地装载和更方便地转换为二进制指令。

因为在计算机程序中经常使用一些小的常数，所以 Java 中已经专门定义了一些字节码，用于在需要时提供这些常数，例如，助记符 iconst_5 表示把整数 5 压入堆栈。如果要把更大的数压入堆栈，需要使用两个字节码，第一个字节码表示操作，而第二个表示操作数。我们在前面曾经提到，每个类的局部变量都保存在一个数组中。从特性上来说，数组中位于前面的几个元素是最活跃的，所以 Java 中有一些字节码是专门针对最开始的几个局部数组元素的。访问数组中位于其他位置的元素需要一个 2 字节的指令，其中一个字节用来表示操作码，第二个用来表示数组元素的偏移量。

按照上面所说，我们考虑 Simple.class 的 main() 方法的字节码。从图 8-14 中提取相关的字节码列于图 8-15 中，并且添加相应的助记符和解注。图中最左边一栏给出每条指令的相对地址。线程专用程序计数器使用这些相对地址来控制程序的流程。我们现在来跟踪这个字节码的执行过程，看它是如何工作的。

当解释程序开始执行这个代码时，会将 PC 初始化设置为 0，并执行 iconst_0 指令。这是执行 simple.java 程序源代码中的第三行语句 int i = 0。PC 自动增量加 1，接着会执行每一条初始化指令，直到第 4 条指令遇到 goto 语句。这条指令将一个十进制数 11 加到程序计数器中，所以 PC 的值变成 0Fh，指向 load_1 指令。

此时，JVM 已经为 i 和 j 分配了初始值，并接着检查 while 循环中的初始条件以判断循环体是否应该执行。为此，JVM 会将从局部变量数组中提取的变量 i 的数值压入堆栈中，然后又把比较值 0Ah 压入堆栈。注意：这里编译器已经为我们做了少许的代码优化工作。在默认条件下，Java 利用 32 位来保存一个整数，因此要占用 4 个字节。然而，编译器发现利用一个字节足以保存这个较小的十进制常数 10，所以编译器就编写代码只将一个字节压入堆栈，而不是压入 4 个字节。

比较操作指令 if_icmplt，会从堆栈中弹出 i 和 0Ah，并比较这两个数值。这个助记符后面的 lt 表示比较操作要查找小于 (less than) 的条件。如果 i 小于 10，就将 PC 的值减去 08h，得到的结果 7 也就是循环体的起始地址。当这个循环体中所有指令都完成后，执行过程回到地址 0Fh 处的判断条件处理语句。一旦

这个判断条件变为假（false）后，解释程序会在进行清理一些工作后，把控制权交还给操作系统。

偏移量（PC 值）	字节码	助记符	意　义
- - - - - - - - - - - -			变量初始化
0	03	iconst_0	将整数 0 压入堆栈
1	3C	istore_1	从堆栈顶部移走整数并放到局部变量数组的地址 1 处
2	0E	dconst_0	将双精度常数 0 压入堆栈
3	A7	goto	由下面的两个字节给出跳转字节数
	00	0b	表示将跳过接下来的 11 个字节
	0B		表示将数值 0B 加到程序计数器
- - - - - - - - - - - -			循环体
7	1B	lload_1	将一个整数值从局部数组压入堆栈
8	04	iconst_1	将整数常数 1 压入堆栈
9	60	iadd	将位于堆栈顶部的两个整数操作数相加（求和结果压入堆栈）
A	3C	istore_1	从堆栈顶部移走整数并放到局部变量数组的地址 1 处
B	28	dload_2	从局部数组装入双精度变量，并放入堆栈中
C	0F	dconst_1	将双精度常数 1 压入堆栈
D	63	dadd	将位于堆栈顶部的两个双精度数值相加
E	49	dstore_2	从堆栈的顶部取出双精度求和结果存放到局部变量数组的位置 2 处
- - - - - - - - - - - -			循环条件
F	1B	lload_1	从局部变量数组的位置 1 处装入整数
10	10	bipush	将如下字节值压入堆栈中（十进制数 10）
	0A		
12	A1	if_icmplt	按照“小于”条件比较位于堆栈顶部的两个整数值
	FF		如果结果为真（i< 10），将下列值加到程序计数器
	F5		作意，FFF5 = -11（十进制数）
15	B1	return	否则，返回

图 8-15　为 Simple.class 加注解的字节码

如果有 Java 编程人员不明白解释程序是如何知道哪个源代码行引起了程序出错，可以研究一下图 8-14 中从地址 108h 开始的二进制类文件，就不难得到答案。这个地址也就是源程序中行数表（line number table）的开始处，行数表描述的是源程序中某个的特定行与程序计数器值的对应关系。在地址 106h 开始的两个字节通知 JVM 在接下行数表中有 7 个入口条目。通过补充细节，我们可以构建一个程序计数器值和源程序行的相互对照表（也称为交叉引用，cross-reference），如图 8-16 所示。

PC	Source Line 处	
	1	public class Simple {
	2	public static void main (String[] args) {
00	3	int i =0;
02	4	double j = 0;
04	5	while (i< 10) {
07	6	i = i + 1;
0B	7	j = j + 1.0;
0F	8	} // while
15	9	} // main()
	8	} // Simple()

图 8-16　有关 Simple.class 的程序计数器和源程序行的相互对照表

例如：当 PC=9 时，程序会发生崩溃，出错的源程序行应该是第 6 行。当 PC 大于或等于 0Bh 且小于 0Fh 时，将开始解释由源代码的第 7 行所生成的字节码。

因为 JVM 在装载和执行字节码时要做很多工作，所以 JVM 的性能不可能与编译程序语言的性能相匹配。即使是采用即时（Just-In-Time，JIT）编译器的加速软件，这种情况也是如此。然而，这种权衡带来的好处是，类文件可以在某个计算机平台下创建和保存，而在另一个完全不同的计算机平台上执行。例如，我们可以在一台 Alpha RISC 服务器上编写和编译一个 Java 程序，这个程序将能够运行在那些下载这种类字节码的 CISC 结构的奔腾系列客户机上。这种“编写一次，到处运行”的范式对于使用不同结构的系统和地理上分散的系统的企业来说是非常方便的。Java 小程序（applet，指一些运行在浏览器中的字节码）对于基于 Web 的交易和电子商务活动来说是必需的。基本上，在 Java 小程序的支持下，用户所要做的一切就是合理地运行当前的浏览器软件。由于 Java 的可移植性和易用性特点，使得 Java 语言及其虚拟机环境非常适合于作为一个理想的中间件平台。

8.6 数据库软件

显然，企业最有价值的资产不是办工室或工厂，而是企业的数据资料。不管企业的性质如何：私人企业，教育机构，或者是政府部门，有关企业的过去历史和现状的明确记录都会在它的数据资料中留下烙印。如果这些数据和企业的状态不一致，或者数据本身之间不一致，那么这些数据的可用性就会受到质疑，并且肯定会产生麻烦。

支撑一个企业的任何计算机系统都是一些相关的应用程序的技术平台。这些应用程序会不断进行数据的更新工作，以保持企业的数据和企业的状态变化相一致。人们通常将这些相关的应用程序组称为*应用程序系统*（application system），因为这些程序需要像一个集体一样协同工作：很少有程序的独自工作会非常有效。应用程序系统的各个组件会共享同一组数据，通常（但不是必须）也会共享相同的计算环境。现在，应用程序系统使用许多不同工作平台：桌面微型机、文件服务器和大型计算机系统。随着基于 Web 的相互合作的计算方式的普及，人们有时根本不知道或者不用关心他们的应用程序是在哪里运行的。虽然从数据管理科学的角度来说，每种计算机平台都有自己的优势和面临的挑战，但是数据库管理软件的基本观念 30 多年来一直没有改变过。

早期应用程序系统采用磁带或打孔卡片来记录数据。由于具有顺序的特点，磁带和打孔卡片上的数据更新工作必须作为一个组来运行，或者说是批处理方式，以提高数据处理效率。因为磁盘上的任何数据单元都能够直接进行访问，所以使用磁盘的计算机系统结构已经不再强制使用依赖平面文件（flat file）的批处理更新方式。然而，旧的习惯一时很难打破，而且大量程序的重写工作需要花费很高的代价。因此，平面文件的处理方式在大多数读卡器都变成博物馆的陈列品而不再使用后还持续了很多年。

在平面文件中，每个应用程序都可以自由定义自己所需的数据对象。正因为这样，人们对于应用程序系统很难有一个统一的观点。例如，假如有一个应收款账户管理系统，它是一个用来记录公司债务人的名称、债务数目和负债时间的应用程序系统。这个程序系统按月份自动生成发票，可以将每月的交易事项发布到一个名为 CUST_ OWE 的 6 位数字域（或数据元素）。现在，按月核对账目的人员可能恰好命名这个域为 CUST_ BAL，并且希望这个域的宽度为 5 位数字。这里，几乎可以肯定有些东西出现了重叠，将会丢失某些信息或者产生混乱。也许在某个月的某个时间，出现了几千美元的账目“未予说明（unaccounted for）”的事情。最后，我们对该程序进行排错调试时却发现 CUST_ OWE 和 CUST_ BAL 是同一个数据元素，而上面的问题是由于数据截取或某个域的溢出条件所引起的。

数据库管理系统（database management systems，DBMS）的发明就是为了防止出现上述的困境。数据库管理系统对于基于文件的应用程序系统强制实行顺序和一致性。使用数据库系统，编程人员不再能够按自己喜欢的方式来任意描述和访问某个数据元素。在数据库管理系统中只有一种数据元素的定义方式。即系统的*数据库模式*（database schema）。在某些计算机系统上，程序员看待数据库的观点

与计算机系统看待数据库的观点之间存在着很大差别，前者称为数据库逻辑模式（logical schema），后者称为数据库物理模式（physical schema）。数据库管理系统将数据库的逻辑模式和物理模式整合成一体。应用程序在数据库管理系统和操作系统的控制下，运用数据库管理系统所呈现的逻辑模式来读取和更新位于物理模式中的数据。图 8-17 描述了这种关系。

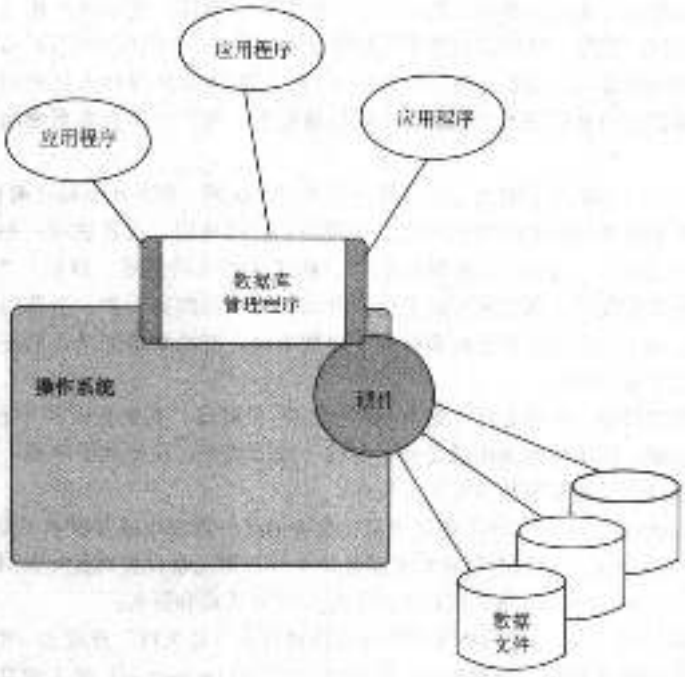


图 8-17 数据库管理系统与其他系统组件之间的相互关系

我们把由某个数据库模式所定义的单个数据元素组织称为记录（records）的逻辑结构。将记录组合在一起就成为文件。相关文件的集合即构成数据库。

数据库设计人员充分考虑到应用程序的需求，以及创建逻辑模式和物理模式后的性能。通常的设计目标是既要将冗余性和浪费的空间减至最小，同时又要保持一个满足需求的性能水平。这种性能水平通常按照应用程序的反应时间来测评。例如，银行系统不会将客户名字和地址放到数据库中的每个已经取消了的支票记录上。该信息仅会保存在一个账户主文件中，这个账户主文件使用账户号作为关键字域（key field）。然后，每个被取消的支票只会出现账号以及和支票本身有关的信息。

数据库管理系统对于如何从物理上组织数据变化很大。实际上，每个数据库供应商都开发了自己专有的管理和文件索引的方法。大多数数据库系统都使用不同的 B+ 树型数据结构（参见附录 A 的详细说明）。数据库管理系统通常会独立于操作系统管理磁盘存储器。通过从磁盘管理过程中除去操作系统层，数据库系统能够按照数据库的模式和索引设计来优化磁盘系统的读写操作。

在第 7 章中，我们学习了磁盘文件的组织结构。大家知道大多数磁盘系统都是以成块的方式从磁盘中读取数据，最小的寻址单元为一个扇区。大多数大型系统的一次读操作可以读取一个完整磁道的数据。当索引结构变得很深时，我们需要执行多次读操作才能使遍历索引树的可能性大大增加。所以，现在的关键问题是如何组织这个索引树来尽可能降低磁盘 I/O 操作的频率。其中一种方案是创建非常大的内部结点以便使每个节点上可以覆盖更多的记录值。显然，这种大节点可以减少索引树的每一层的结点数，而且有可能通过一次读操作即可访问一个完整的索引树层。另外一种方案是让内部结点的规模变小以便在一次读操作中可以读取索引树的更多层。到底哪种是比较好的方式呢？这个问题的答案与数据库所运行的特定计算机系统有关。要得到一个最佳的答案甚至可能取决于数据本身。例如，

如果数据的关键字较稀疏（sparse），也就是说可能会有很多没有使用的关键字的值，这种情况下我们也许会选择某种特定的索引组织方案。但是对于密集型的索引结构，可能会选择另外一种方案。不管数据库的实现方式如何，数据库的调整对于理解数据库的管理软件、系统的存储结构以及系统管理的数据分布的细节来说，都是一项非常重要的任务。

数据库文件通常有多个索引。例如，假如有一个的客户数据库，使用客户账号和客户姓名来查找记录不失为一个好办法。当然，每个索引都会增加系统在空间（用于存放索引）和时间（因为当增加或删除记录时，所有索引都必须立即更新）方面的开销。数据库系统设计人员所面临的一个主要挑战是，既要确保有足够的索引来实现大多数情况下的快速检索，而且又不会在系统的管理事务上增加太重的负担。

数据库管理系统的目标是提供对大量数据及时和方便的访问，但是还要设法确保数据库的完整性。这意味着数据库管理系统还必须允许用户自己定义规则和管理规则，或者说将一些限制（constraints）加到某些关键的数据元素上。有时，这些限制只是一些非常简单的规则，例如，“客户号不能为空”。当然，还有许多更复杂的规则，例如规定哪类用户可以看到哪类数据元素，以及包含相关数据元素的文件如何更新等等。对于任何数据库管理系统的实用性来说，有关系统安全性的定义和措施，以及数据完整性的限制都是十分关键的。

数据库管理系统的另外一个核心组件是数据库的事务管理器。事务管理器（transaction manager）负责控制更新数据对象，以确保数据库总是处于前后一致的状态。从形式上来说，事务管理器控制数据状态的变化，这样每个处理事务都具有如下特性：

- 原子性（atomicity）——所有相关的更新都在事务的范围内发生或者根本不发生更新。
- 一致性（consistency）——所有更新都要满足对全部数据元素设置的各种限制。
- 隔离性（isolation）——没有事务可以干涉其他事务的活动和更新。
- 持久性（durability）——成功的事务要尽可能快地写入“持久性”介质上（例如磁盘）。

这就是大家所熟知的有关事务管理的 ACID 特性（ACID properties）的 4 项基本内容。通过下面的例子，我们可以很容易理解这种 ACID 特性的重要性。

假设你已经支付了信用卡的月账单，并立即将账单邮寄出去。现在，你到了附近的一个商店，用信用卡进行另一次购物。假如恰好在售货员用读卡机刷卡时，银行会计也正在将你的月付款输入到银行的数据库。图 8-18 给出了一台中央计算机系统处理这些事务的一种方式。

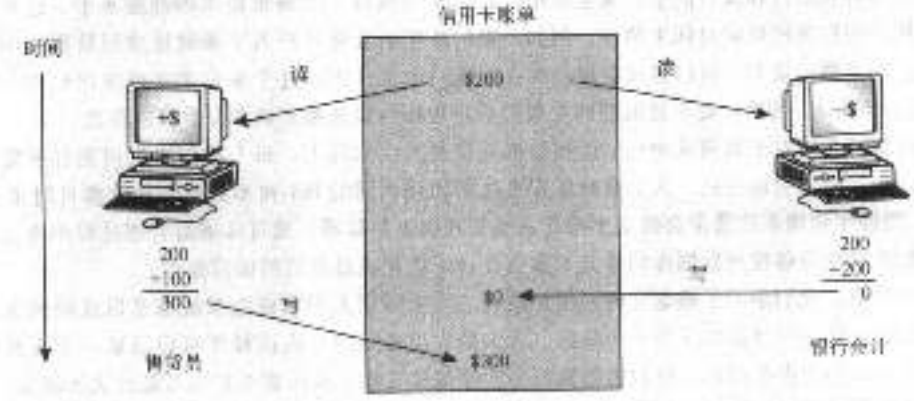


图 8-18 一种假想事务处理的方式

图中，银行会计在售货员完成刷卡之前完成了数据更新，结果留下一个 300 美元的未付账单。当然，也有可能容易发生图 8-19 所示的处理事务。这里，售货员先完成数据的更新，因此信用卡账户上的结余为 0.00 美元，也就是说你刚刚得到了免费商品。

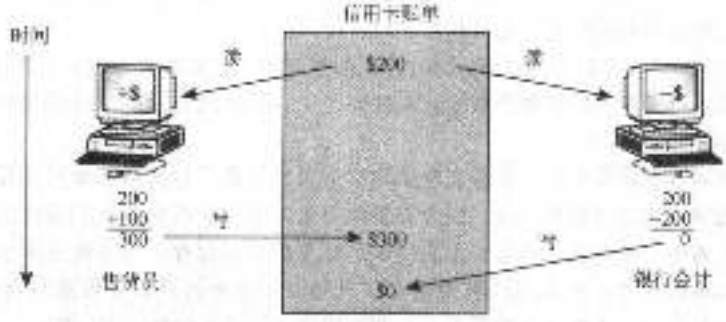


图 8-19 另一种假想事务处理的方式

虽然免费得到商品可能会让你感到高兴，但是同样可能出现的事情是要支付账单两次。当然，你会和银行会计争论，直到出错的记录被更正过来。刚才我们描述的这种情形被称为竞争状态（race condition），因为数据库最终的状态并不取决于数据更新的正确性，而是取决于最后完成的处理事务。

事务管理器通过强制实施的处理事务的原子性和隔离性来防止这种竞争状态的发生。为此，人们会在数据记录上安置各种不同类型的锁。如在 8.18 的例子中，银行会计将会在信用卡记录上一个“专用”锁。这把锁只有在完成将更新过的数据写回磁盘后才会打开。当银行会计的处理事务正在运行时，售货员会得到一个系统正忙的信息。当银行数据更新处理完成后，事务管理器就会打开银行会计的这把锁，并且立即为售货员上另一把锁、正确的事务处理如图 8-20 所示。

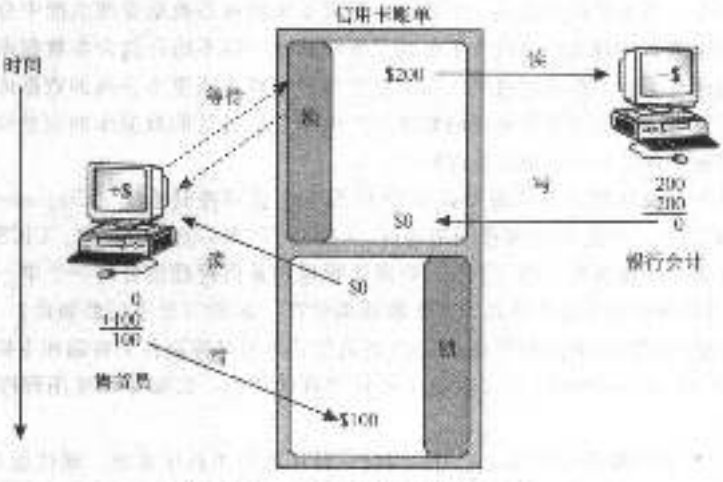


图 8-20 一个孤立的原子处理事务

这种解决方案也存在一些风险。如果在任意时刻一个复杂系统中的某个实体被锁住的话，就有潜在的死锁危险。当然，各种数据库系统都能够非常巧妙地管理系统中的各种锁以降低发生死锁的风险，但是每种预防或检测死锁的措施都会给系统带来更多的管理成本。如果管理锁的任务太多的话，事务处理的性能就会受到影响。一般来说，死锁的预防和检测并不是数据库性能考虑中的最重要因素。死锁的情形很少发生，而性能却是每个事务处理中的一个关键因素。

另外一个需要考虑的性能问题是数据日志。在更新记录的过程中（包括记录删除的过程），数据库事务管理器要把事务的图像写入日志文件（log file，或称为运行记录文件）。因此，每次数据更新都需要至少两次写入：一次写原始文件，另一次写日志文件。日志文件非常重要。如果由于出现一个错误而必须中止某个事务时，日志文件可以帮助系统保持处理事务的完整性。例如，如果数据库管理系统在实际更新发生前获得将要更新的记录的一个图像，那么这个老的图像能够被快速地写回到磁盘系统，

从而擦除掉以后对该记录的所有更新。在某些数据库系统中，更新“前”和更新“后”的记录图像都会被捕捉记录，这样使得错误恢复会变得相对容易一些。

作为*检查跟踪*（audit trail，也称为审计跟踪），数据库日志也是非常有用的。数据库日志可以显示谁在什么时候更新了哪个文件以及哪个数据元素被改变了。一些细心的系统管理员常常会把日志文件放在磁带库中保存几年时间。

对于数据备份和数据恢复来说，日志文件是非常重要的工具。有些数据库实在是太大了，以至于不可能每天都用磁带或光盘来进行备份，因为需要花费太多时间。而实际上的做法是，数据库文件的全备份工作一周只需做一至两次，但是日志文件却是每天都必须保存。如果在这些全备份之间的某个时刻发生了任何灾难性事件，那么可以利用记录该时间段的事务的日志文件进行*向前恢复*（forward recovery），重建每日的处理事务就好像是由早先那些用户重新输入到数据库一样。

我们刚刚讨论过了有关数据库访问的控制问题，即安全、索引管理和锁管理，这些事务消耗了大量的系统资源。事实上，对于早期的系统来说，这个管理费用实在是太大了以至于有些人怀疑是否要继续使用这种基于文件的数据库系统，原因是他们的计算机主机系统难以处理数据库管理的重负。即使是采用当今功能非常强大的计算机系统，如果我们不对数据库系统进行适时的调整和恰当的维护，系统的吞吐量也会受到很大影响。系统操作人员和数据库分析员都必须精心地监控处理大量事务的环境，努力使数据库管理系统保持最佳的工作性能。

8.7 事务管理器

一种改善数据库性能的方法是简单地让数据库少做些工作，而是把数据库的一些功能分离出来转移到其他系统组件上。事务管理就是从一个数据库管理系统的核心数据管理功能中分离出来的一个数据库组件。独立的事务管理器通常也组合了负载平衡和其他一些不适合包含在数据库核心软件中的最优化的特征，从而改善了整个系统的效率。当商业交易跨越两个或更多分离的数据库时，事务管理器会显得特别有用。没有一个参与事务处理的数据库能够为它们对等的数据库的完整性负责，但利用一个外部事务管理器就能保持所有这些数据库同步。

其中一个最早和最成功的事务管理器是 IBM 的顾客信息和控制系统（Customer Information and Control System，CICS）。自从 1968 年推向市场后，CICS30 年来一直表现良好。CICS 之所以如此引人注目，是因为它是第一个将事务处理（TP）、数据库管理和通信管理整合到一个单一的应用程序套件中的系统。然而，CICS 的各个组件都是非常松散地耦合在一起的（至今仍然如此），这样可以把每个组件都当作一个单独的实体来调整和管理。CICS 的通信管理组件控制各个终端和主机系统之间交互活动，我们称为*会话*（conversation）。由于没有了协议管理的负担，数据库和应用程序能够更加有效地完成自己的工作。

CICS 是第一个在客户服务器环境中采用远程过程调用的应用程序系统。现代版本的 CICS 能够管理上千上万个因特网用户和大型主机之间的事务处理。直到现在，CICS 还是非常类似 20 世纪 60 年代开始时的那种体系结构，实际上 CICS 的体系结构已经成为此后发明的所有事务处理系统的范式。现代 CICS 结构示意图如图 8-21 所示。

从图中可以看出，称为*事务处理监视器*（transaction processing monitor，TP monitor）的程序是这个系统中的关键组件。它从远程通信管理器接收输入，并通过保存哪些用户被授权哪些事务的清单的数据文件来对系统中的事务进行鉴别监视。有时，这种安全信息包括某些特定的信息，例如定义在什么地方可以运行某些特定的事务（如在内部网还是互连网，intranet versus internet）。一旦这个 TP 监控器鉴别了这个事务，它就开始执行用户所请求的应用程序。当应用程序需要数据时，TP 监控器向数据库管理软件发送一个请求。事务处理监视器在完成所有这些工作，同时在多个并发执行的应用程序进程中的保持每个处理事务的原子性和隔离性。

也许读者已经想到，要求所有的这些 TP 软件的各个组件都必须驻留在同一台主机计算机上是毫

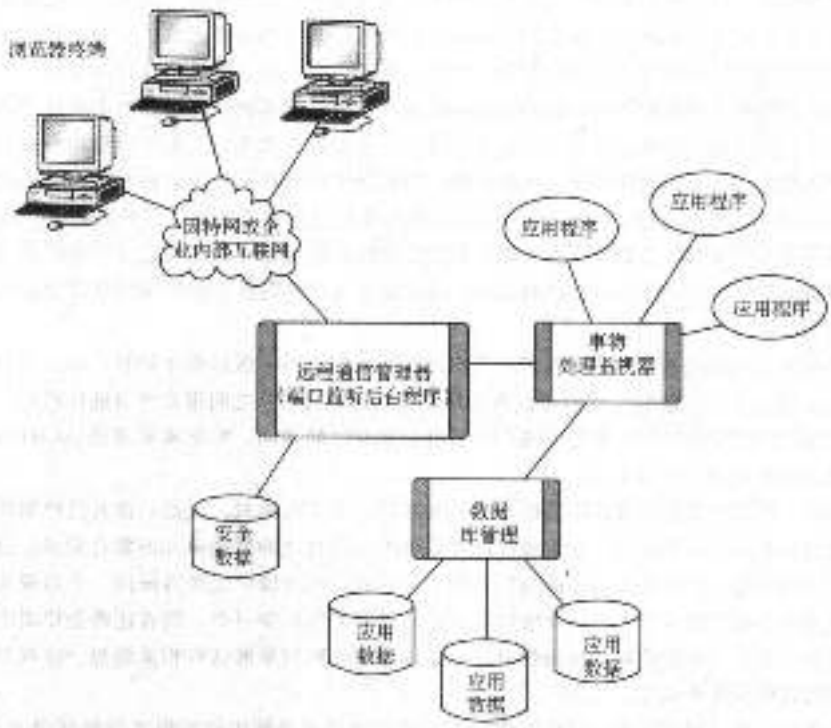


图 8-21 CICS 的体系结构

无道理的。的确，我们没有理由要把这些组件都放在一起。一些分布式的体系结构专门利用一个小组服务器来运行 TP 监控器。运行监视器的系统和包含数据库管理软件的系统从物理结构上来说是不同的。事实上，也没有必要让运行 TP 监控器的系统和运行数据库管理软件的系统是同一类的系统。例如，我们可以使用 Sun 公司的 Unix RISC 结构的系统来进行通信管理，而使用 Unisys 公司的 ES/7000 系统在 Windows Datacenter 操作系统环境下运行数据库软件。可以通过桌面计算机或移动式个人计算机来输入各种处理事务。这种配置被认为是一个 3 层的体系结构（3-tiered architecture），每个平台代表一个层次。更通常情况是一个 n 层（n-tiered）的体系结构，或者是多层体系结构（multi-tiered architecture）。随着 Web 计算和电子商务的出现，n 层 TP 体系结构变得越来越流行。许多系统供应商，包括 Microsoft、Netscape、Sybase、SAP AG 和 IBM 的 CICS，都已经能够成功支持各种不同的 n 层事务处理系统。当然，我们不可能对某个特定的企业说其中哪个会“更好”一些。这里的每种产品都有自己的优缺点。在决定哪种体系结构最适合某个特定的环境之前，如果在设计一个 TP 系统时能够同时考虑系统的成本和所有可靠性因素应该是一种比较谨慎的做法。

本章小结

本章描述了计算机硬件与软件之间的相互关系。计算机软件和硬件结合起来，才能构成一个具有特定功能和高效率的计算机系统。系统软件，包括操作系统和应用软件，是连接计算机用户和计算机硬件的一个接口。系统软件让我们能够对低层计算机体系结构进行抽象处理，这样就为用户创造一个工作环境，这里用户主要考虑的是问题的解决方案，而不是系统操作。

在操作系统设计中，计算机硬件与软件之间的相互作用和深度依赖关系是十分显然的。从历史发展来看，操作系统起源于一种“开放式计算工作站（open shop）”的方式，然后转变为操作员控制的批处理方式，进而发展到支持交互式的多道程序处理和分布式计算。现代操作系统提供了一个用户接口和各种各样的服务，包括内存管理、进程管理、通用资源管理、调度和保护等。

有关操作系统概念的知识对于每位计算机专业人员来说都是至关重要的。事实上，所有系统活动都与操作系统的服务紧密相关。如果操作系统出了问题，那么整个系统会出现问题。然而，我们也应该清楚意识到，并不是所有的计算机都有，或者需要有一个操作系统。嵌入式系统就是这样的情形。在汽车或微波中的计算机要执行的任务都非常简单，所以也就没有必要使用操作系统。但是，对于不只是完成运行单一程序的简单任务的计算机系统来说，操作系统是必需的。利用操作系统，人们可以更加有效和方便地使用计算机。操作系统是一种大型软件系统。一般来说，研究操作系统对于系统软件开发是十分有价值的。由于各种各样的理由，在这里我们衷心地希望大家能够对计算机操作系统的设计和开发进行更加深入的探索。

利用汇编程序和编译器，人们可以将人类可能够阅读的计算机语言转换成适合于在机器上执行的二进制形式。计算机的解释程序也能生成二进制代码，但是通常这种代码没有由汇编程序所生成的二进制代码的速度快和效率高。

Java 编程语言生成的代码可用一个位于字节码和操作系统之间的虚拟机来解释。Java 代码运行的速度要比二进制程序慢得多，但是 Java 代码可以在各种不同的计算机平台之间很方便地进行移植。

数据库系统软件通常通过事务处理系统来控制对数据文件的访问。数据库系统的 ACID 特性能够确保数据库中的数据总是保持一致的状态。

构建大型的、可靠的系统是现在计算机科学所面临的一个主要挑战。至此，读者已经知道了计算机系统远不是只有硬件和程序这样简单。企业级计算机系统由一些独立的处理单元的集合构成，这里的每个处理单元都有自己的功能。如果仅从用户的角度来看，这些独立的处理单元中的任何一个出现失效或者性能低下，都会对整个系统的运行带来破坏性的后果。在今后的工作和学习中，读者还将会详细学习到本章的许多问题。作为一个系统管理员和系统编程用户，必须深刻理解和掌握这些相关思想。这些思想将会具体地应用于各种特定的操作环境中。

无论人们编写的程序如何智能，都难以弥补由于运行程序的系统组件的拙劣性能所带来的各种问题。在第 10 章中我们将进行深入研究，更加详细地探讨有关系统性能方面的各种问题。

深入阅读

系统软件领域中的最令人感兴趣的阅读材料是产品供应商所提供的附带资料。事实上，人们可以经常通过供应商提供的文献的质量和关注度来评判其产品的质量。浏览产品供应商的网站，有时也可以获得大量的有关产品的理论背景知识的第一手资料。其中两个最好的网站是供应商 IBM 公司和 Sun 公司的网站：www. software. ibm. com 和 www. java. sun. com。如果坚持不懈地搜索，无疑能够找到更多内容。

Hall（1994）关于客户-服务器的书籍是一本极好的客户-服务器理论的入门读物。书中介绍了当时大量的流行产品。

Stallings（2001）、Tanenbaum（1997），以及 Silberschatz、Galvin 和 Gagne（2001）的著作都很好地讨论了本章引入的有关操作系统的基本概念和一些高级内容。Stallings 的书中还详细介绍了各种不同的操作系统以及它们与实际机器硬件之间的相互关系。读者可以在 Brooks（1995）的著作中找到有关 OS/360 开发的精彩内容。

Gorsline（1998）的有关汇编语言的书籍较好地介绍了汇编程序的工作原理。他还讨论了链接和宏汇编的细节问题。Aho、Sethi 和 Ullman（1986）编写了一本“限定式（the definitive）”编译器的书。因为封面上的插图的缘故，这本书常常被称为“龙书（The Dragon Book）”。因为讲述清楚和内容全面，这本关于编译器理论的书籍近 20 年来不断再版。每位计算机科学家手头都应该有这样一本书。

Sun 公司的出版物是有关 Java 语言的主要资料来源。Addison-Wesley 出版社出版了一系列有关详细介绍 Java 语言的书籍。Lindholm 和 Yellin（1999）的“Java 虚拟机规范（The Java Virtual Machine Specification）”就是这一系列书籍中的一本。它提供了有关类文件结构的一些细节说明，而本书没有介绍这些内容。Lindholm 和 Yellin 的著作还包括 Java 字节码指令和它们对应的二进制指令的一个完整清单。仔细研究这部分内容无疑会让读者对 Java 语言有一种新的视野。

虽然这些内容稍微有些过时，但是 Gray 和 Reuter（1993）的有关事务处理的著作仍是一本全面易读的

书籍。它为在这个领域的深入学习提供了一个很好的基础。Silberschatz、Korth 和 Sudarshan（2001）的著作全面介绍了有关数据库的理论和应用，受到了大家的高度关注和好评。

参考文献

Aho, Alfred V., Sethi, Ravi, & Ullman, Jeffrey D. *Compilers: Principles, Techniques, and Tools.* Reading, MA: Addison-Wesley, 1986.

Brooks, Fred. *The Mythical Man-Month.* Reading, MA: Addison-Wesley, 1995.

Gorsline, George W. *Assembly and Assemblers: The Motorola MC68000 Family.* Englewood Cliffs, NJ: Prentice Hall, 1988.

Gray, Jim, & Reuter, Andreas. *Transaction Processing: Concepts and Techniques.* San Mateo, CA: Morgan Kaufmann, 1993.

Hall, Carl. *Technical Foundations of Client/Server Systems.* New York: Wiley, 1994.

Lindholm, Tim, & Yellin, Frank. *The Java Virtual Machine Specification, 2nd ed.* Reading, MA: Addison-Wesley, 1999.

Silberschatz, Abraham, Galvin, Peter, & Gagne, Greg. *Operating System Concepts, 6th ed.* Reading, MA: Addison-Wesley, 2001.

Silberschatz, Abraham, Korth, Henry F., & Sudarshan, S. *Database System Concepts, 4th ed.* Boston, MA: McGraw-Hill, 2001.

Stallings, W. *Operating Systems, 4th ed.* New York: Macmillan Publishing Company, 2001.

Tanenbaum, Andrew, & Woodhull, Albert. *Operating Systems, Design and Implementation, 2nd ed.* Englewood Cliffs, NJ: Prentice Hall, 1997.

基本概念和术语复习

1. 与现代操作系统相比，早期操作系统的主要目标是什么？
2. 常驻监控程序给计算机的操作带来了什么改进？
3. 关于打印机的输出，假脱机（spool）一词的由来？
4. 描述多道程序处理系统和分时系统之间的区别。
5. 在硬实时计算机系统的操作中最重要的因素是什么？
6. 多处理器系统是否能够按照它们之间的通信方式来分类？本章中是如何对它们分类的？
7. 分布式操作系统与网络操作系统有什么区别？
8. 透明是什么含义？
9. 描述有关操作系统内核设计中的两种不同的思想。
10. GUI 操作系统接口有什么优缺点？
11. 长程调度和短程调度有什么区别？
12. 占先调度是什么含义？
13. 在分时共享的环境中什么样的进程调度方法最有效？
14. 什么进程调度方法证明是最佳的？
15. 描述实现关联转换所涉及的步骤。
16. 除了进程管理外，操作系统的另外两大功能是什么？
17. 什么是覆盖？为什么在大型计算机系统中不再需要使用覆盖技术？
18. 操作系统和用户程序对虚拟机有不同的见解。解释它们之间有什么区别。
19. 子系统和逻辑分区有什么区别？
20. 列举服务器整合的好处。是否对每个企业来说服务器的整合都是一个好的方案？
21. 描述编程语言的分层结构。为什么三角形是表示这种分层结构的合适符号？
22. 绝对代码与可重新定位的代码之间有什么区别？
23. 链接编辑器有什么作用？它与动态链接库有什么不同？

24. 描述编译器的每个阶段（phase）的作用。
25. 解释程序与编译器有什么区别。
26. 在不同的分立硬件环境下具有可移植性的 Java 编程语言有什么突出特点？
27. 汇编程序生成的机器代码在链接编辑后是可执行的。Java 编译器会生成什么编码是要在执行过程中进行解释的？
28. 识别 Java 类文件的魔术数字是什么？
29. 逻辑数据库模式和物理数据库模式有何不同？
30. 在对数据库建立索引时哪种数据结构是最通用的？
31. 为什么数据库重组是必要的？
32. 解释一个数据库系统的 ACID 特性。
33. 什么是竞争状态（race condition）？
34. 数据库日志有哪两种用途？
35. 事务管理器可以提供什么服务？

练习题

1. 你认为如果一个计算机没有操作系统会受到什么限制？用户如何装载和执行程序？
2. 微内核是试图提供尽可能小的内核，而将操作系统支持的大部分功能放到其他模块中。你认为内核必须提供的最少的服务是什么？
3. 如果你在为一个实时系统书写代码，那么将会在系统上加上什么限制？
4. 多道程序处理和多重处理（multiprocessing）技术之间有什么区别？多道程序处理和多线程技术之间又有什么区别？

◆5. 在什么情况下，需要将一组进程和程序放入运行在一个大型计算机上的子系统中？在这个系统上创建逻辑分区有什么好处？

6. 在同一个计算机上同时使用子系统和逻辑分区有什么好处？

◆7. 何时适合于使用不可重新定位的二进制程序代码？为什么人们偏爱可重新定位的代码？

8. 如果没有可重新定位的程序代码。存储器的分页将会变的如何复杂？

◆9. 讨论动态链接的优缺点。

10. 汇编程序需要克服哪些困难，才能通过一次通读源文件生成完整的二进制代码？为一次通读汇编程序编写的代码与为二次通读汇编程序书写的代码之间有什么区别？
11. 为什么在普通应用程序开发中应该避免使用汇编语言？而在什么情况下需要或者优先使用汇编语言？
12. 你认为在什么样的情况下，使用汇编语言开发一个应用程序更加合适？
13. 与解释语言相比，编译语言有什么优点？在什么情况下你会选择一种解释语言？
14. 讨论下列与编译器有关的问题：
    a）编译器的哪个步骤会给出一个语法错误？
    b）哪个步骤会指出没有定义的变量？
    c）如果想添加一个整数到一个字符串中，编译器的哪个步骤会发出错误信息？
15. 为什么 Java 类的执行环境被称为虚拟机？这种虚拟机如何与运行 C 语言代码的真实机器进行类比？
16. 为什么你会认为 JVM 的方法区对所有在虚拟机环境下运行的线程是全局性的？
17. 我们曾经说明过，运行于 JVM 每个线程中一次只能有一个方法处于活动状态。你认为为什么会是这种情况？
18. 访问局部变量数组的一个类的字节码的长度最多有两个字节。其中一个字节用作操作码，另外一个字节用来指示在数组中的偏移量。在局部变量的数组中最多能够保存多少个变量？如果超过这个数目，会发生什么情况？

◆19. Java 被称为一种解释语言，而且 Java 也是一种产生二进制输出流的编译语言。试解释这种语言是如何

既可以被编译也可以被解释的。

20. 我们曾经说明过，运行于 JVM 的 Java 程序的性能不可能与一个常规的编译语言相匹配。试解释原因？

21. 针对数据库处理回答下列问题：

◆a) 什么是竞争状态（race condition)？试举例说明。

◆b) 如何防止竞争状态？

◆c) 在竞争状态的预防过程中会有什么风险？

22. 多层事务处理的体系结构在哪些方面优于单层结构？通常哪种体系结构的代价会更高？

23. 为了改善系统的性能，你的公司决定通过几个服务器来复制公司的产品数据库。这样不是所有事务都要经过一个单一的系统。这里，需要考虑什么类型的问题？

24. 我们说过，任何时刻只要有某个系统资源被锁住，那么总是会存在死锁风险。描述一种死锁可能发生的方式。

* 25. 研究各种不同的命令行（command-line）的接口界面（例如 Unix、MS-DOS 和 VMS）以及各种不同的 Windows 接口界面（例如任何 Microsoft 的 Windows 产品、MacOS 和 KDE)。

a) 下面考虑一些主要命令，例如列出目录表、删除文件、或者改变目录。试解释在你所学过的各种操作系统上，这些命令的每条是如何执行的。

b) 列出并解释使用命令行接口界面比使用 GUI 界面更方便的一些命令。同时列出并解释使用 GUI 界面比使用命令行接口界面更方便的一些命令。

c) 你更喜欢哪种类型接口？为什么？

品质永远都不是偶然的；品质是坚定的意志、艰辛的努力、正确的方向和熟练的技巧的综合结果；品质代表了从许多的可选物中做出明智选择。

——William A. Foster

计算机技术似乎已经达到了发展的极限，但是我们这样说的时候需要非常小心，因为在未来的5年中这些言论听起来可能会有些愚蠢。

——John von Neumann（1949）

第9章　可选择的体系结构

9.1　概述

在前面的章节中，我们重点介绍了计算技术的一些背景知识。从计算机科学从业者的角度来看，这些知识主要专注于单处理器的计算机系统。通过学习，大家可以了解各种不同的计算机硬件的功能，以及这些部件对系统的整体性所起的作用。这些知识不但对计算机的硬件设计，而且对高效率的算法实现都是至关重要的。大多数人都是通过使用个人计算机和工作站的经验来熟悉计算机硬件的。这使得他们没有触及到计算机体系结构中的一个重要领域：一些可选择的计算机体系结构。因此，本章重点介绍一些超越传统冯·诺伊曼方法的计算机体系结构。

本章将讨论RISC计算机，使用指令级并行执行的体系结构和多重处理的体系结构（其中简要地介绍并行处理）。首先从臭名昭彰的RISC和CISC的争论开始说起，让大家了解这两种指令系统（ISA）的差异及其相对的优点和缺点。接下来会介绍一种不同体系结构的分类方法，并且研究如何分类并行运算的各种体系结构。随后，将讨论一些与指令级的并行结构相关的主题，重点强调超标量的体系结构并且重新说明EPIC（显式并行的指令计算机）和VLIW（超长指令字）的设计。最后，简要介绍多处理器系统和一些实现并行执行的新方案。

在20世纪80年代早期，计算机硬件设计师开始重新评价各种不同的体系结构理论。重新评价的首个对象就是指令系统的体系结构。令设计师感到非常困惑的是，为什么计算机需要一个非常复杂的扩展指令集，而在大多数的时间里指令集中却只有20%的指令在使用。这个问题导致了RISC机器的发展。在第4章和第5章中，已经引入了RISC的简单介绍，本章我们将使用一节的篇幅来阐述这个问题。RISC设计的深入人心导致了一种独一无二的CISC和RISC的结合。现在，许多的计算机体系结构都使用RISC的核心来实现CISC体系结构。

在第4章和第5章中介绍了一些新的体系结构，例如VLIW、EPIC和多处理器结构，它们在计算机硬件市场上占据着一个较大的份额。开发指令级并行执行的体系结构的发明，导致了一种新技术的诞生。利用这种技术，计算机可以在程序代码实际执行之前就能够准确地预测出程序代码中的分支转移的结果。基于这些预测的指令预取技术极大地提高了计算机的性能。除了可以预测下一条要提取的指令外，高度的指令级并行技术还引发了诸如推测执行之类的新观点，这样处理器可以在实际计算前就猜测出结果值。

可选择的体系结构也包括了多处理器的计算机系统。对于这些体系结构，我们是借鉴了我们的祖先和友善的牛的故事。如果我们使用一头牛去拉倒一棵树，而这棵树又太大的话，我们不会去尝试养一头更大的牛。反之，我们会同时使用两头牛。多重处理的体系结构与多头牛的问题非常相似。当需要解决一些困难棘手的问题时，就需要引入这种思想。但是，多处理器系统带给了我们一些前所未有的挑战，特别是关于高速缓存和存储器一致性问题。

值得注意的是，尽管我们正在掌握一些可选择的体系结构，但是这些体系结构的高性能都是建立在提高成本的基础之上的。当前，先进的计算机系统所带来的性能上的提高和它们的成本之间的关系

是非线性的。在大多数情况下，成本的增加要远远超过所获得的性能提高。这也就使得我们难以将这些体系结构加入到主流的应用体系结构中。然而，新的体系结构在市场上也有它们的位置。在一些需要高度数字化的科学和工程应用中，常常需要一些性能高于普通单处理器系统的计算机。对于这类应用中的计算机来说，成本通常都不是问题。

在学习本章时，读者也许还会记得在第 1 章中曾介绍了计算机发展的不同时代。许多人都相信我们现在已经进入了一个以这些可选择的体系结构为基础的计算机新时代，特别是在并行处理的领域。

9.2 RISC 计算机

在第 4 章和第 5 章有关指令系统设计的内容中，曾经介绍了 RISC 计算机的体系结构。之所以命名为 RISC（精简指令集计算机），是因为与 CISC（复杂指令集计算机）相比，RISC 计算机最初只需一个较小的指令集。随着 RISC 计算机的不断发展，RISC 中的术语“精简（reduced）”已经变得有些用词不当，现在就更不用说了。RISC 设计的最初想法是只提供一组能够执行所有最基本的操作的最小指令集：数据移动、ALU 操作和分支转移。只允许显式的 load 和 store 指令访问存储器。

实际上，设计复杂指令系统是因存储器的成本过高而引起的。如果每条指令的复杂程度越高，就意味着程序变得越小，从而占用更少的存储器空间。CISC 的指令系统采用长度可变的指令，这样可以保持简单的指令尽量短，同时也允许拥有比较长的更加复杂的指令。另外，CISC 的体系结构中还包含大量能够直接访问存储器的指令。这样，CICS 的结构拥有密集、强大和变长的指令集，也就导致了每条指令的执行需要不同的时钟周期。某些复杂指令，特别是访问存储器的指令，往往需要数百个时钟周期。在某些情况下，计算机的设计师发现，为了允许有足够的时间来完成指令，常常需要减慢系统的时钟频率（可以使时钟跳动的间隔变大一些）。这样就导致了较长的程序执行时间。

人类语言也表现出一些 RISC 和 CISC 的特点，我们可以将人类语言作为理解 RISC 和 CISC 之间区别的一个很好的类比。假如你有一个中国的笔友。你们俩都能同时流利地使用英语和中文说写。虽然你们都喜欢彼此的长篇大论，但也都希望将双方的通信费用降到最低。在通信时，可以选择使用昂贵的航空信纸，这样能节省相当多的邮资，或者使用普通信纸而支付额外的邮票费用。第三种选择就是在每一张纸上书写尽可能多的信息。

与中文相比，英语简单而冗长。中文字符比英语单词更加复杂，一个需要 200 英文字母的句子可能只需用 20 个中文字符。用中文通信需要较少的字符，可以节省纸张和邮资。但是，读写中文比较费劲，原因是每一个中文符号包含更多的信息。英语单词类似于 RISC 指令，而中文字符则更类似于 CISC 指令。对于大多数说英语的人来说，用英语“处理”信件花费的时间比较少，但是也需要更多的物理资源。

有很多的报道都吹捧 RISC 结构是一种新的革命性的设计，而 RISC 的起源却要追溯到 20 世纪 70 年代中期。IBM 公司的 John Cocke 的早期工作撒下了 RISC 的种子。1975 年，Cocke 开始建造他的实验模型 801 主机。这个系统最初几乎没有受到人们的关注，直到许多年后才公开机器的细节。这期间，David Patterson 和 David Ditzel 于 1980 年发表了一篇得到广泛称赞的论文，“精简指令集计算机的案例”。这篇文章开创了对计算机体系结构设计的一种变革性的新思路，并将首字母的缩写词 CISC 和 RISC 写进了计算机科学的词典中。这种由 Patterson 和 Ditzel 构思的新结构提倡使用简单的指令，而且所有的指令都有相同的长度。这样一来，每条指令都执行较少的任务，然而每条指令的执行所需要的时间却是固定的和可以预测的。

CISC 机器上进行的编程结果为 RISC 机器提供了数据上的支持。这些研究表明，数据移动指令大概占总指令的 45%，ALU 操作（包括算术运算、比较操作和逻辑操作）占总指令的 25%，而分支转移指令（或者说程序流的控制）则占了 30%。尽管在 CISC 机器中有大量的复杂指令，但却是很少使用。这一发现，并伴随着大量较便宜的存储器的出现，以及超大规模集成电路（VLSI）技术的发展，导致了各种不同类型的体系结构的开发。价格便宜的存储器就意味着程序可以占用更多的存储器空间。这样，我们可以使用一些简单的，可预测的指令组成较长的程序，来代替由可变长度的复杂指令所组

成的较短的程序。简单指令的执行允许使用较短的时钟周期。另外，较少的指令系统也意味着在计算机芯片中需要更少的晶体管。晶体管数目的减少使得芯片的制造成本更加便宜，可以有更多的芯片空间来实现其他的功能。指令的可预测性和 VLSI 技术的发展允许各种各样的性能增强技术可以利用计算机的硬件来实现，例如流水线操作。而 CISC 结构却不提供这些性能提升的机会。

可以使用如下的基本的计算机性能等式来量化 RISC 和 CISC 的差别：

$$\frac{\text{时间}}{\text{程序}}=\frac{\text{时间}}{\text{周期}}\times\frac{\text{周期数}}{\text{指令}}\times\frac{\text{指令数}}{\text{程序}}$$

可以通过程序的执行时间来度量计算机性能。计算机的性能与时钟周期、每条指令需要的时钟周期数和程序所包含的指令数成正比。如果可能的话，缩短时钟的周期，就能够改善 RISC 和 CISC 的性能。所不同的是，CISC 机器通过减少程序中的指令数达到性能提升的效果，而 RISC 则通过将每条指令需要的时钟周期数减至最少来增强计算机的性能。这两种体系结构在相同的时间内差不多产生相同的结果。在门电路的层次上，这两种系统完成等量的工作。但是，在程序层次与门电路层次之间又会发生什么样的情况呢？

CISC 机器是利用微代码来处理指令的复杂性问题。微代码会告诉处理器如何去执行每一条指令。因为执行效果的原因，微代码通常是紧凑且有效率的，而且必须是正确无误的。然而，微代码的效率受制于可变长度的指令系统，不同长度的指令会减慢译码的操作过程。而且在可变长度的指令系统中，每条指令所要求的时钟周期数也会有所不同，这样就很难实现指令的流水线操作。另外，微代码是在指令从存储器中取出后才对每条指令进行解释。这种额外的指令转换过程也会消耗时间。越是复杂的指令集，查找指令所需要的时间就越多。同时，也需要花费更多的时间使计算机的硬件系统适合于这些指令的执行。

RISC 体系结构采用的是一种不同的解决方案。大部分 RISC 指令的执行是在一个时钟周期内完成。为了实现指令的加速执行，RISC 结构使用硬件连线控制来取代 CISC 结构中的微程序控制，从而加快了指令的执行。这就使得 RISC 机器比较容易实现指令的流水线操作，但是要在硬件层次上处理指令的复杂性问题非常困难。在 RISC 系统中，有关指令的复杂性问题已经从指令集中移除，将它放到编译器域的层次上进行处理。

为了说明起见，我们来看下面的一条指令。假如要计算乘积 5×10 的结果。在 CISC 机器上，程序的代码可能会是：

```
mov ax, 10
mov bx, 5
mul bx, ax
```

一个最小化的 RISC 指令系统中没有乘法指令。因此，在 RISC 系统中，对这个乘法问题的处理可能为：

```
        mov ax, 0
        mov bx, 10
        mov cx, 5
Begin:  add ax, bx
        loop Begin    :causes a loop cx times
```

虽然 CISC 代码比较简短，但是需要执行更多的时钟周期。假设在这两种体系结构中，对于寄存器到寄存器的移动，加法运算和循环运算，每一个运算都只占用一个时钟周期。同样，这里假设乘法指令需要 30 个时钟周期[⊖]。比较上面的两种代码段：

CISC 指令：

总的时钟周期＝（2 次移动×1 个时钟周期）＋（1 次乘法×30 个时钟周期）＝32 个时钟周期

RISC 指令：

总的时钟周期＝（3 次移动×1 个时钟周期）＋（5 次加法×1 个时钟周期）＋
（5 次循环×1 个时钟周期）＝13 个时钟周期

⊖ 这并不是一个不实际的数目。在 Intel 8088 上，两个 16 位数的乘法运算需要 133 个时钟周期。

事实说明，RISC需要的时钟周期数通常少于CISC的时钟周期数。应该了解的是，即使RISC需要更多的程序指令，但是程序在RISC系统上的实际执行时间却要比CISC的执行时间短。这就是RISC设计背后的主要动机。

我们已经提过降低指令的复杂度可以使计算机芯片的结构更加简单。以前在CISC指令的执行过程中，晶体管被用作为数据管道、高速缓存和寄存器。在这三者中，寄存器提供了改善计算机性能的最大潜力。因此，增加寄存器的数目和用全新的方式使用寄存器是十分有意义的。一种寄存器的创新的使用方式是使用*寄存器窗口组*（register window sets）。虽然这种方式不像RISC体系结构的其他创新技术被人们广泛地接受，然而寄存器开窗口是一个非常令人感兴趣的观点，在此将做简要的介绍。

高级语言的效率取决于程序的模块化。过程调用和参数传递就是使用这些模块所自然带来的副作用。调用一个过程是一个比较重大的任务。它包括保存一个返回地址、保存寄存器的值、传递参数（或者将参数压入堆栈，或者使用寄存器）、分支转移到子程序，并执行这个子程序。直到子程序完成，必须保存参数值的修改。在返回调用程序时的执行之前，必须恢复先前的寄存器的值。保存寄存器、传递参数和恢复寄存器都需要消耗大量的工作量和系统资源。RISC的芯片中的数百个寄存器，要实现保存和恢复的序列操作简化只需简单地改变寄存器的环境即可。

为了充分地理解寄存器窗口的这个概念，可以尽力想像将所有的寄存器分成若干个组。当一个程序在某个环境下执行时，仅仅只有一个特定的寄存器组是可见的。如果这个程序改变到另一个环境中（例如调用一个过程）执行，那么新环境中的可见寄存器组就会发生变化。例如，当主程序在运行时，可能看见的只是从第0个寄存器到第9个寄存器的寄存器组。当调用某个特定的过程时，可能看到的组将是从第10个寄存器到第19个寄存器。通常，一个实际的RISC体系结构中包含16个寄存器组（或者称为窗口），而每一个寄存器组中有32个寄存器。在任意时刻，CPU都受到限制只能在某个单一的寄存器窗口中工作。因此，从程序员角度来看，只有32个可用的寄存器。

寄存器窗口本身对于过程调用或参数传递并没有帮助。但是，如果我们对这些窗口进行仔细地重叠覆盖，那么从一个模块到另一个模块的参数传递活动就变成了只需简单地从一个寄存器组移动到另一个组。为了实现参数的传递，必须共享的寄存器的两个组需要进行覆盖。这可以通过将寄存器窗口划分成不同的分区来实现。其中包括：*全局寄存器*（对所有窗口共用）、*局部寄存器*（属于当前窗口）、*输入寄存器*（与前一个窗口的输出寄存器覆盖），和*输出寄存器*（与下一个窗口的输入寄存器覆盖）。当CPU从一个过程转换到下一个过程时，它就在不同的寄存器窗口之间进行切换。这种重叠的窗口允许实现参数的“传递”，只需简单地从调用模块中的输出寄存器改变到被调用模块的输入寄存器。一个*当前窗口指针*（current window pointer，CWP）可以指示任意时刻正被使用的寄存器窗口组。

下面，我们考虑过程1（procedure one）调用过程2（procedure two）的假想情况。在每一组的32个寄存器中，假设有8个寄存器是全局的，8个是局部的，8个用作输入和8个用作输出。在过程1调用过程2时，所有需要传递的参数都存放到过程1的输出寄存器组中。一旦过程2开始执行，这些寄存器就变成了过程2的输入寄存器组。这个过程如图9-1所示。

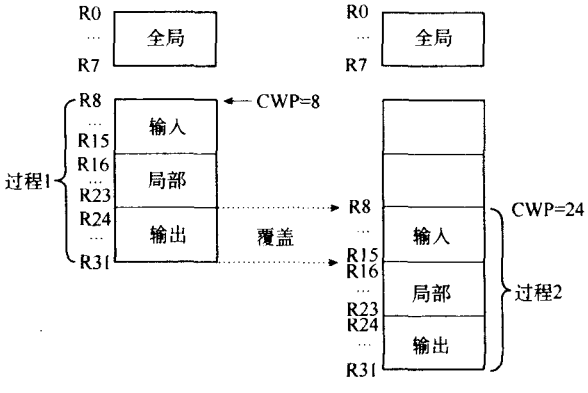


图9-1 重叠的寄存器窗口

值得注意的一个重要信息是，RISC机器中的寄存器组具有循环使用的特性。对于具有高度嵌套的程序来说，有可能会占满机器所提供的所有寄存器。如果发生这种情况，主存储器就会接管寄存器的工作，把具有最小编号的寄存器窗口存储在主存储器中，窗口中所包含的

是来自最早过程激活中的值，最高编号的寄存器（最新近激活的寄存器）然后会回绕到最低编号的寄存器。当从执行的过程返回时，嵌套的层次降低，计算机会按照原来的保存顺序从存储器中恢复寄存器的值。

除了简单外，RISC机器中的定长指令和高效率的流水线极大地提高了这种体系结构的处理速度。指令越简单，占用的芯片空间就越小。这样不仅仅会带来更多的可用空间，而且在芯片的设计和生产时也会变得更轻松和省时。

应该指出的是，要把现在的处理器分类为RISC体系结构和CISC体系结构已经变得越来越困难。这些体系结构的区分界线也已经变得很模糊。现在一些流行的体系结构同时采用RISC和CISC两种方式。如果我们浏览一些新的芯片手册，就会发现现在的RISC机器变得越来越奢侈，甚至有一些比CISC机器更复杂的指令。例如，RISC体系结构的PowerPC计算机比CISC体系结构的Pentium系列具有更大的指令集。随着VLSI技术不断地把晶体管制造得更小更便宜，指令集的大小现在已经很少成为CISC和RISC之间争论的话题，而寄存器的使用和装入或/存储（Load/store）式的体系结构则变成了争论的热点议题。

正如上面所述，表9-1中谨慎地总结了一些有关RISC体系结构和CISC体系结构之间的典型区别。

表9-1 RISC与CISC机器的对比

RISC	CISC
多寄存器组，通常由256个以上的寄存器组成	单寄存器组，典型的寄存器总数为6到16个
每条指令允许有三个寄存器操作数（例如，add R1, R2, R3）	每条指令允许有一个或两个寄存器操作数（例如，add R1, R2）
通过高效率的片上（on-chip）寄存器窗口实行参数传递	利用低效率的片外（off-chip）存储器实现参数传递
单周期指令（load和store除外）	多周期（multiple-cycle）指令
硬件流线控制	微程序控制
高度流水线作业	低度流水线作业
数目较少的单指令	数目众多的复杂指令
固定长度指令	可变长度指令
使用编译器解决复杂性问题	利用微代码解决复杂性问题
只有装载和存储指令才能访问存储器	许多指令都可以访问存储器
寻址方式较少	许多寻址方式

我们已经提过，尽管有许多报刊资料盛赞RISC设计的革命性创新，许多在RISC机器上使用的思想（包括流水操作和简单指令），实际上在20世纪60年代和70年代的计算机主机就已经实现了。许多新设计并不是真正意义上的新设计，仅仅只是简单的轮回应用。创新并不一定意味着发明一个新的轮子，简单的创新可能是发现了现有的这个轮子的一种最好的使用方法。这也为你将来从事计算领域的工作提供一个很好的启发。

9.3 FLYNN分类方法

在过去的许多年，为了找到一种分类计算机体系结构的满意方法已经进行了多次尝试。尽管没有一种方法是完美的，但是今天被大家最广泛接受的分类方法是1972年由Michael Flynn提出的分类方法。弗林分类方法（Flynn's taxonomy）主要考虑两大因素：指令的数目和流入处理器的数据流的数目。一台计算机可以有一个或多个的数据流，而且可能是一个或多个处理器在这个数据上工作。这就提供了4种可能的组合：SISD（单指令流，单数据流）、SIMD（单指令流，多数据流）、MISD（多指

令流，单数据流）和 MIMD（多指令流，多数据流）。

单处理器是 SISD 机器。SIMD 机器上有一个单一的控制点，能够在多个数据值上执行相同的指令。SIMD 的类别包括阵列处理器、矢量处理器和脉动阵列（systolic arrays）。MISD 机器具有在相同的数据流上操作的多个指令流。MIMD 机器采用多个控制点，且具有独立的指令和数据流。多处理器和大多数现代的并行处理系统都是 MIMD 机器。SIMD 机器的设计比 MIMD 机器简单，但是也十分缺乏灵活性。所有的 SIMD 多处理器都必须同时执行相同的指令。如果考虑到这一点，那么执行某些类似分支转移条件的简单事情可能代价马上就会变得高昂。

Flynn 分类法在某些方面存在缺点。第一，对于 MISD 机器，能够运行的应用程序（如果有的话）很少。其二，Flynn 假定并行执行都是同构的。然而，一组处理器的集合可能是同构，也可能是异构的。一台机器可以理所当然地拥有 4 个独立的浮点加法器、2 个乘法器和 1 个单一的整数单元。因此，这台机器可以并行地执行 7 个操作，但是却不容易满足 Flynn 的分类系统。

这种分类方法遇到的另外一个问题是 MIMD 的类别。一个多处理器的体系结构属于这种类别，但是却没有考虑到处理器之间是如何进行连接的，也没有考虑到处理器是如何看待存储器的。还有几种方法试图细分 MIMD 的类别。修改的建议包括将 MIMD 细分为共享存储器和不共享存储器的两类不同的系统，以及按照系统是否是基于总线或总线切换来对 CPU 进行分类。

对于共享存储器的系统，所有的处理器都可以访问全局存储器，并且处理器之间通过共享变量进行通信，就像在单处理器上所进行的处理过程一样。如果多处理器不共享存储器，那么每一个处理器都必须拥有自己的一部分存储器。结果，所有的处理器都必须通过消息传递的方式来进行通信，这种通信方式的代价昂贵而且效率低下。有些人认为使用存储器作为划分硬件的一个决定性因素会带来一个问题，那就是共享存储器和消息传递实际上是一种编程模型，而不是一种硬件模型。因此，它们更适合属于系统软件的范畴。

并行体系结构的两个主要范式是 SMP（对称多处理机，symmetric multi-processors）和 MPP（大量信息并行处理机，massively parallel processors）。它们都是 MIMD 体系结构，但是它们在使用存储器上有区别的。SMP 机器，例如一个双处理器的 Intel PC 和一个 256 处理器的 Origin 3000 之间共享存储器。而对于 MPP 处理器，例如 nCube、CM5 和 Cray T3E 处理器之间就不能共享存储器。这些特殊的 MPP 机器通常会在一个单个的大机柜内安装几千个 CPU 和连接数百千兆字节的存储器，这些系统的价格通常高达几百万美元。

起初，术语 MPP 是用来描述紧密式耦合的多个 SIMD 处理器，如 Connection Machine 和 Goodyears' s MPP。然而现在，术语 MPP 用来表示有多个独立节点和专用存储器的并行体系结构，所有的这些节点通过网络进行通信。区分 SMP 和 MPP（指现在的定义）的一个简单方法如下：

MPP= 许多处理器 + 分布式存储器 + 通过网络通信

然而，

SMP = 几个处理器 + 共享存储器 + 通过存储器通信

分布式计算是 MIMD 体系结构的另外一个例子。分布式计算（distributed computing）通常定义为一组网络连接的计算机协同作业来解决某个问题。然而，这种合作可以通过多种不同的方式进行。

工作站网络（network of workstations，NOW）是指一组按并行方式工作的分布式工作站的集合，而且网络工作站的各个节点并不作为一个常规的工作站使用。NOW 通常由异构系统组成，有不同类型的处理器和软件，并通过因特网进行通信。个人用户必须在网络上建立适当的连接才能加入到并行计算。工作站群集（cluster of workstations，COW）类似于 NOW 的工作站集合，但它却要求有一个单一的实体来进行管理。这种结构的节点通常具有通用的软件，如果一个用户能访问其中的某个节点，那么他也就能够访问所有的节点。专用群集式并行计算机（dedicated cluster parallel computer，DCPC）是一种专门连接起来从事特定的并行计算工作的工作站集合。这些工作站都有相同的软件和文件系统，并实行单一实体的管理方式。这些工作站之间通过因特网进行通信，但是并不会作为普通的

工作站使用。PC 机群（piles of PCs，POPC）是指一种专门针对使用异构硬件来构建一个并行系统的群集。一般来说，DCPC 配备数目相对较少、但价格较高和速度较快的组件，而 POPC 却使用大量的速度慢但相对较便宜的节点。

1994 年，由 Goddard 太空飞行中心的 Thomas Sterling 和 Donald Becker 引进的 BEOWULF 计划，就是一个 POPC 的体系结构。它成功地用一个特殊设计的软件将各种不同的计算机硬件平台捆绑在一起，使得这个体系结构看起来和感觉上就是一个统一的并行机器。BEOWULF 网络上的节点都是通过一个专用网络进行连接的。如果你有一台老式的 Sun SPARC 计算机、若干 486 的机器、一台 DEC Alpha（或者就只是一大堆老式的 Intel 机器！）和一种将这些机器连接网络方法，那么就可以安装 BEOWULF 软件来创建一个属于你个人的、却是功能非常强大的并行计算机。

Flynn 分类法最近已经扩大到包括 SPMD（单个程序多个数据）的体系结构。一个 SPMD 由多个处理器组成，其中每一个处理器都有自己的数据集合和程序存储器。同一个程序在各个处理器上执行，并在不同的全局控制点上进行同步。虽然每个处理器都装载同一个程序，但是每个处理器可能执行不同的指令。例如，一个程序可能具有类似如下的代码：

```
If myNodeNum = 1 do this, else do that
```

这样，不同的节点执行同一程序中的不同指令。SPMD 实际上是在 MIMD 机器上使用的一个编程范例，但它却与 SIMD 不同，因为 SIMD 中的各个处理器在同一时刻做的是不同的事情。超级计算机通常使用一个 SPMD 的设计。

在 Flynn 分类法上面的某个层次上，需要再添加一种特性，这就是区分体系结构是利用指令驱动的，还是利用数据驱动的。经典的冯·诺伊曼体系结构是属于指令驱动类型。所有的处理器活动都由一个程序代码序列来决定。程序指令作用于数据。而数据驱动类型，或者说数据流类型的体系结构却恰好相反。在这里，数据的特性决定了处理器的事件序列。第 9.5 节将详细地研究这种思想。

在添加了数据流计算机以及对 MIMD 分类法进行了细化之后，得到如图 9-2 所示的分类方法。也许大家希望在接下来章节的学习中会用到这个分类图。我们先从树的左边分支开始，介绍与 SIMD 和 MIMD 体系结构相关的内容。



图 9-2 计算机体系结构的分类方法

9.4 并行和多处理器体系结构

自从计算科学开始以来，科学家一直致力于制造各种计算机器来更好更快地解决各种问题。小型化的技术促进了电路性能的改善，以及能够在一个芯片上放置更多的元器件，计算机的时钟频率也变

得更快，现在的 CPU 能在千兆赫兹的时钟频率下工作。然而，大家都知道有些物理障碍控制了单处理器（single-processor）的性能的提高。例如，热和电磁效应的影响会限制芯片中晶体管的密度。即使这些问题都已解决（当然还不清楚是在什么时候），处理器速度总是会受到光速制约。除了这些物理上的限制因素外，还有经济上的问题。从某种意义上来说，不断提升处理器的速度所带来的费用将可能超过人们的承受能力。最终，将导致无法继续提升处理器的性能。当然，一种可行的解决办法是将计算负担分配到多个处理器上。正是因为这些原因，并行执行的概念变得越来越流行。

然而，值得注意的是，并不是所有的应用程序都能够从并行执行中受益。例如，多重处理的并行机制会增加开销，如在处理同步问题和其他过程管理方面的开销。如果一个应用程序并不适合并行执行的解决方案，那么将它移植到一个多重处理并行结构上，通常会得不偿失。

如果程序执行正确，并行执行机制通常能带来较高的吞吐量，更好的容错性，以及一个更吸引人的性价比。虽然并行执行可以大幅度地提升程序处理的速度，但是这种加速决不是完美无缺的。如果有 n 个处理器以并行处理的方式运行，那么理想的加速率就意味着一个计算任务能够在原来 $1/n$ 的时间内完成，也就是系统的处理能力会提升 n 倍，或者说运行时间降低到原来的 n 分之一。

回忆 Amdahl 定理，就可以认识到为什么这种完美的加速是不可能的。如果两个处理部件分别以两个不同的速度运行，那个较慢的速度将会起主导的作用。这个定理同样也决定了使用并行体系结构解决某个问题所能得到的加速率。不管你把一个应用程序的并行化处理得多好，总是会有一小部分工作必须由某个处理器按顺序执行。在这个顺序处理完成之前，其他的处理器除了等待外什么工作都不能做。这其中蕴含的意义是，任何一种算法都会有按顺序执行的部分，这些顺序执行最终会制约通过多处理器执行所能达到的加速。顺序处理的部分越大，则应用一个多重处理的并行结构的效果就会越差。

使用多处理器执行单一任务仅仅是许多并行处理机制中的一种。在前面的章节中，我们曾经介绍了几种并行执行的机制，其中包括流水操作、VLIW 和 ILP 的方法。并阐述了每一种特定类型的并行机制产生的动机。其他类型的并行执行的体系结构可以处理多（或者是并行）数据问题。这类例子中包括 SIMD 机器，例如矢量处理器、神经处理器和脉动处理器等。还有许多的体系结构都可以执行多重或并行处理，所有的 MIMD 机器都具有这种特性。非常重要的一点是，要注意“并行”有多种不同的意义。而且，区分不同的意义也同样非常重要。

本节先讨论 ILP 体系结构的一个例子，然后将讨论有关 SIMD 和 MIMD 体系结构的内容。最后部分将引入一些新（非主流）的并行处理方法，包括脉动阵列、神经网络和数据流计算。

9.4.1 超标量和 VLIW 体系结构

本部分我们会重新回顾超标量体系结构，并将它们与 VLIW 体系结构进行比较。超标量体系结构和 VLIW 体系结构都属于指令级的并行执行，但它们在处理方法上有区别。为了使下面的讨论有一个很好的基础，我们首先从超流水线的定义开始。大家还记得，在流水线操作中，将取指-解码-执行周期分成不同步骤（或阶段）。在同一个时间内，我们会把一组指令分配到不同的步骤中。一个最理想的情形应该是，每个时钟周期都会有一条指令从流水管道中移出来。但是，由于代码存在分支转移指令以及数据的相互依赖性，所以每个时钟周期都有一条指令输出的目标不可能达到的。

当一个流水线中的步骤的执行只要求少于半个时钟周期时，就会发生*超流水线作业*（superpipelining）。这时，可以添加一个内部时钟信号。如果内部时钟以两倍的外部时钟的速度运行时，就能够实现在每个外部时钟周期内完成两个任务。尽管超流水线作业同样可以应用于 RISC 体系结构和 CISC 体系结构，但是在大多数情况下，超流水线技术应用在 RISC 体系结构的处理器中。超流水线作业是超标量体系结构设计的一个方面。正因为此原因，人们常常会混淆超流水线和超标量体系结构这两个概念。

所以，到底什么是一个超标量体系结构的处理器？大家知道 Pentium 系列处理器是一个超标量体系结构的处理器，但是目前为止我们还没有讨论超标量体系结构的真正含义。*超标量*（superscalar）

是一种允许在一个时钟周期内同时执行多条指令的设计方法。尽管超标量体系结构和流水线作业在许多方面有所不同（在下面将要简短讨论它们的区别），但是最终的效果却是相同的。超标量体系结构的设计中获得系统加速的方式类似于在一个繁忙的单通道高速公路增加另一条通道。这种方法需要增加额外的“硬件”，但最后，可以有更多的汽车（或指令）能够在相同的时间内从 A 点到达 B 点。

类似于额外的高速公路车道的超标量部件被称为*执行单元*（execution units）。执行单元包括浮点加法器和乘法器，整数加法器和乘法器，以及其他一些专用的部件。虽然这些单元同样也可以独立地工作，但是非常重要的一点是，这种超标量体系结构中有足够多数量的专用单元并行地处理多条指令。在这里，重复的执行单元是很常见的。例如，一个超标量的系统可以有一对完全相同的浮点单元。通常，执行单元是按流水技术安排的，这样可以提供更好的性能。

这种体系结构中的一个关键部件是有一个*特殊的取指单元*（instruction fetch unit），它可以从存储器中同时取出多条指令。这个单元依次将指令传递给一个*综合译码单元*（decoding unit），以确定这些指令是否是独立的（独立的指令可以同时执行），或者是否具有某种类型的相关性的（在这种情况下，所有的指令不能同时执行）。

作为一个例子，考虑 IBM RS/6000 计算机。这种处理器有一个取指令单元和两个处理器，每个处理器中都有一个 6 级的浮点单元和一个 4 级的整数单元。取指令单元设置成两级流水线作业。其中，第一级的任务是每次提取一个 4 指令的指令包，第二级则是将这些指令传送到合适的处理单元上。

超标量计算机是一种通过流水线作业和单元复制来实现并行执行的体系结构。超标量设计中包括超流水线作业，同时提取多条指令，一个能够确定指令相关性和动态地组合指令以确保没有违反相关性的综合译码单元，以及适合于多条指令并行执行所需要的足够多数量的系统资源。大家注意到，尽管这种类型的并行执行机制需要专门的硬件设备，但超标量体系结构同样也需要一个复杂精巧的编译器来完成事务的调度操作，这样可以最佳地利用机器的资源。

超标量处理器同时依赖于硬件（对相关性进行仲裁）和编译器（产生适当的事务调度），然而 VLIW 处理器则完全依赖于编译器。VLIW 处理器会把许多独立的指令包装组合成一条长指令，这条长指令会依次地告诉执行单元将要做什么。有许多人争辩说，由于编译器对代码中的各种相关性有更好的*全局观*（overall picture），所以这种方法可以产生更好的性能。但是，由于编译器不能对运行时间的代码有一个全局观，所以编译器在调度时被迫趋向保守。

当一个 VLIW 的编译器创建非常长的指令时，同样也要对所有指令的相关性进行仲裁。这种长指令要在编译时才能确定，通常包含 4 到 8 条程序指令。因为这些长指令被固定后，任何可能会影响指令的事务调度的修改工作（例如改变存储器的等待时间），都需要对该代码进行重新编译，这显然也给软件供应商带来大量潜在的问题。VLIW 体系结构的支持者指出了这种技术可以通过把复杂性问题转移给编译器从而达到实现简化硬件的目的。而超标量体系结构的支持者则反驳说，虽然 VLIW 可能简化硬件，但是却会大大地增加了所生成的代码数量。例如，如果不使用程序控制域，那么浪费了存储器空间和带宽。事实上，一个典型的 FORTRAN 程序，在 VLIW 机器上编译时会扩大到正常规模的 2 倍，有时甚至是 3 倍。

Intel 公司的 Itanium IA-64 处理器是 VLIW 处理器的一个例子。大家记得，IA-64 使用的是一个 EPIC 类型的 VLIW 处理器。EPIC 体系结构与普通的 VLIW 处理器相比有很多优点。像 VLIW 一样，EPIC 体系结构也会捆绑指令，以便传送到不同的执行单元。但是，与 VLIW 的区别在于，EPIC 捆绑的指令包不需要具有相同的长度。利用一个特定的分隔符来指示一个指令捆绑的结束和下一个指令捆绑的开始。EPIC 通过硬件对指令字进行预取，并识别指令，然后对独立的指令捆绑包进行事务调度以便并行执行。这是希望克服由于编译器缺少对运行时间代码的全局观所带来的各种限制的一种尝试方法。在捆绑包中的各条指令都没有涉及到相关性，这样就可以对这些指令并行执行，而无需考虑指令的顺序。大多数人都认为，EPIC 其实就是 VLIW。尽管 Intel 公司在这一点上不这样认为，一些顽固的结构设计师也会引用上面所提及的一些微小区别（还有其他的几个微小区别）来争辩，但是实际上

EPIC 只不过是 VLIW 的一个升级版本。

9.4.2 矢量处理器

矢量处理器（vector processors）通常被称为超级计算机。矢量处理器是专门设计的高度流水线作业的处理器。矢量处理器能够对整个矢量和矩阵上行高效率的操作。这类处理器非常适合于能从高度并行执行中受益的应用程序，例如天气预报、医学诊断和图形处理等。

为了理解矢量处理，首先必须了解矢量的算法。矢量是一个确定长度的一维的数值排列，或者是一个有序的标量序列。对于矢量，我们定义了各种不同的算术运算，包括加法、减法和乘法运算。

矢量计算机属于高度流水线作业，所以各种算术运算之间可以重叠。每条指令都指定了一组对整个矢量所执行运算的集合。例如，要把矢量 V1 和 V2 相加，并将结果送入 V3。在一个传统的处理器上，运算的代码将会包含如下循环：

```
for i = 0 to VectorLength
   V3[i] = V1[i] + V2[i];
```

然而，在一个矢量处理器中，代码就变成了：

```
LDV    V1, R1        ;load vector1 into vector register R1
LDV    V2, R2
ADDV   R3, R1, R2
STV    R3, V3        ;store vector register R3 in vector V3
```

矢量寄存器（vector register）是一种可以一次保存多个向量元素专用的寄存器。它每次发送一个元素到矢量流水线。来自流水线输出的结果也是每次传送一个元素回到矢量寄存器中。因此，这些寄存器是能够保存许多值的先进先出（FIFO）的队列寄存器。通常，矢量处理器中有多个这样的寄存器。矢量处理器的指令集中包括装载寄存器，对寄存器中的元素执行操作，以及将矢量数据存回到存储器等指令。

矢量处理器通常可以按照指令如何访问操作数分为两类。*寄存器-寄存器矢量处理器*（register-register vector processor）要求所有的操作都使用寄存器作为源操作数和目标操作数。*存储器-存储器矢量处理器*（memory-memory vector processor）允许操作数从内存直接传送到算术单元。随后，操作结果又被传送存回到存储器。寄存器-寄存器类型的处理器的缺点就是长矢量必须被拆分为确定长度的片断，而且这些片断还必须小到可以存放到寄存器中。然而，存储器-存储器类型的机器也由于有存储器的延迟，因而有一个较长的启动时间。启动时间是指从初始化指令开始到指令执行的第一个结果从流水线出现所需要的时间。然而，在流水线充满时，这个缺点就消失了。

矢量指令之所以效率高有两个原因。首先，机器每次提取较少的指令，这就意味着译码的工作量、控制单元的费用和内存带宽的使用率都会大大减少。其次，处理器知道它会有一个连续的数据来源，并且可以预取出相对应的数据值。如果使用交叉存储的内存，那么每个时钟周期可以到达一个数据对。最著名的矢量处理器是 Cray 系列的超级计算机。在过去的 25 年中，它的基本体系结构很少有所改变。

9.4.3 互连网络

在并行执行的 MIMD 系统中，为了实现同步处理和数据共享，处理器彼此之间通信是必不可少的。消息在系统组件中的传输方式决定了整个系统的设计。消息传输方式有两种：使用共享存储器，或者是采用一个互连网络模型。共享存储器的系统有一个能够被所有的处理器进行同等访问的大存储器。在互连网络的系统中，每个处理器都有自己的存储器，但是允许处理器通过网络访问其他处理器的存储器。当然，这两种方式都有它们的优缺点。

互连网络通常按照拓扑结构、路由策略和交换技术进行分类。*网络拓扑*（topology），也就是各个组件互连的方式，是在信息传递中影响管理费用的一个主要的决定性因素。消息传递的效率受到下列因素的限制：

- 带宽（bandwidth）——网络传递消息的容量
- 消息延迟（message latency）——一条消息的第一个位到达目的地所需要的时间
- 传输延迟（transport latency）——在网络上传输消息所花费的时间
- 开销（overhead）——消息发送者和接收者的消息处理活动

相应地，网络设计的原则总是努力使所要传输的消息量和消息所必须经过的传输距离减至最小。

互连网络既可以是静态（static）网络，也可以是动态（dynamic）网络。动态网络允许两个实体（两个处理器或者一个处理器和一个存储器）之间的路径可以从一次通信到下一次通信时发生改变，而静态网络却不能。互联网络还可以是阻塞型（blocking）网络或者非阻塞型（nonblocking）网络。非阻塞型网络允许有其他同时发生的连接存在的情况下建立新的连接，而阻塞型网络却不能如此。

静态互连网络主要用来传递信息和各种类型的服务，这其中很多服务类型都是我们所熟悉的。处理器之间通常采用静态网络进行互连，而处理器-存储器对之间一般采用动态网络连接。

完全连接网络（completely connected networks）是指网络中的所有组件都相互连接在一起的网络。建立完全连接网络的费用非常昂贵，而且当有新实体要加入到网络时，很难管理网络。星型网络（star-connected networks）有一个中央集线器，所有消息都必须通过中央集线器进行传递。尽管集线器可能会是一个中央的瓶颈，但是集线器可以提供出色的连通性。线性阵列（linear array）或环形网络（ring networks）都允许任意一个实体同它的两个邻居实体直接相连，但是任何一个非邻居间的通信都需要穿越多个实体才能到达目的地。环形连接是线性阵列的一个变体，我们只需要把线性阵列的首尾两个实体直接相连即可。网状网络（mesh networks）是把每一个实体与4或6个邻居实体相连接，其中连接实体的个数取决于是二维网络还是三维网络。这种网络的扩展可以进行环绕连接，非常类似于一个线性网络环绕连接构成一个环形网络。

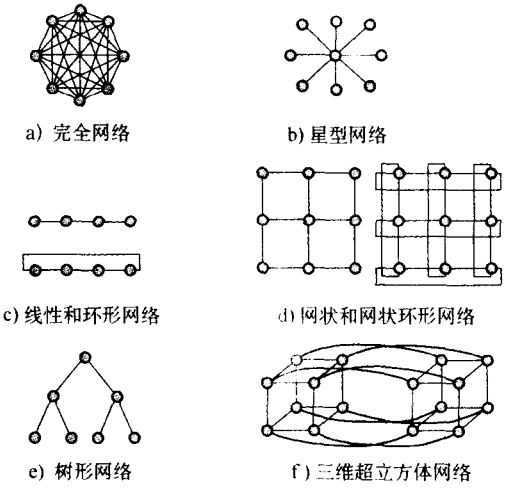


图 9-3 静态网络拓扑图

树形网络（tree networks）是将实体排列成非循环结构，但是在树的根部会形成了一个潜在的通信瓶颈。超立方体网络（hypercube）是网状网络的多维扩展。在超立方体网络中每一维都有两个处理器（超立方体网络通常用来连接处理器，而不是连接处理器-存储器的集合）。二维的超立方体网络由通过一个直接链接进行连接的处理器对组成，这些互连处理器的标号的二进制表示严格说来必须有一个数位，而且也只能有一个数位不同。在一个 n 维的超立方体网络结构中，每一个处理器都直接与 n 个其他处理器相连接。有趣的是，我们将一个超立方体结构网络的两个处理器标号中不相同的数位位置的总数称为海明距离（Hamming distance），海明距离同样也用来表示两个处理器之间的最短路径的通信连接数目。图 9-3表示不同类型的静态网络。

动态网络是采用下面两种方式之一对网络实施动态配制：或者使用总线，或者使用一个交换开关可以改变通过网络的路由。适中的实体数目和成本是网络构建中主要考虑的因素，基于总线的网络（bus-based networks）是最简单和最有效的，如图 9-4 所示。显然，这种方式的最大缺点是当总线上连接的实体数目变得很大时所产生的瓶颈问题。并行总线可以减轻这个瓶颈问题，但是需要花费的代价也是相当大的。

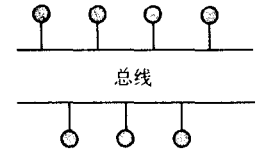


图 9-4 一种基于总线的网络

交换网络使用开关来动态地改变网络的路由。这里有两种类型的开关：交叉开关和2×2开关。交叉开关（或纵横开关，crossbar switch）就是简单的开关，只有打开或关闭两种功能。通过闭合开关就可以实现任意两个实体之间的连接，即产生一个连接。由交叉开关所组成的网络属于完全连接的网络，因为所有的实体都能够直接与其他实体进行连接，并且允许在不同的处理器和存储器对之间同时进行通信。但是，一个特定的处理器一次至多只能有一个连接。而开关断开就可以阻止消息的传输。因此，交叉式网络是一种非阻塞型网络。然而，如果在每一个交叉点需要有一个单独的开关，那么n个实体就需要有n^2个开关。实际上，许多多处理器都要求在一个交叉点上有许多开关。这样一来，管理大量开关的任务即刻就变得非常困难和代价很高。交叉开关实际上只适用于高速多处理器的矢量计算机中。图9-5给出了一个交叉开关的网络配置。图中的蓝色开关代表闭合开关。一个处理器一次只能连接一个存储器，所以图中的每一列中最多只能有一个开关闭合。

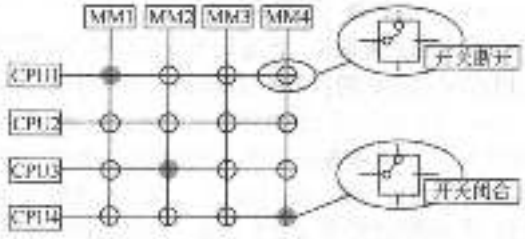


图9-5 交叉式网络

第二种类型的开关是2×2开关。2×2开关和交叉开关非常相似，但是交叉开关只能简单打开或者关闭通信信道，而2×2开关可以将它的输入端连接到不同的目的端。一个2×2交换开关有两个输入端和两个输出端。在任意时刻，一个2×2的开关可以处于下列的4种状态之一：通过（through）、交叉（cross）、上部广播（upper broadcast）和下部广播（lower broadcast）。如图9-6所示，在通过状态，上部的输入端连接到上部的输出端，下部的输入端也连接到下部的输出端。更简单地说，输入是直接通过开关的。在交叉状态，上部的输入端连接到下部的输出端，而下部的输入端连接到上部的输出端。在上部广播中，上部的输入同时广播到上部输出和下部输出。在下部广播中，下部的输入同时广播到上部输出和下部的输出。通过状态和交叉状态是与互连网络相关的两种状态。

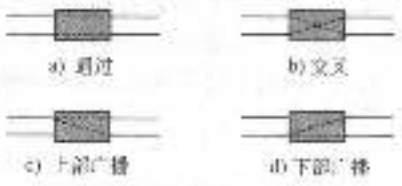


图9-6 2×2交换开关的各种可能状态

目前，最先进的网络类型，多级互连网络（multistage interconnection networks）就是利用2×2开关构建的。这种网络的基本思想是使用多级开关的组合，通常是将处理器放在开关的一边，而把存储器放在开关的另外一边。并且利用一系列开关元素作为网络内部节点。这些开关会动态地配置自己，可以在指定的处理器和指定的存储器之间构建一条通路。所使用的开关的数目和开关的级数对每一条通信信道的路径长度都有贡献。当由开关决定把某个信息从特定的源地址传送到指定的目的地所要求的配置时，可能会发生轻微的延迟。这种多级网络通常称为洗牌网络（shuffle networks），这种说法暗示了开关之间的连接模式。

许多的拓扑结构适合在多级交换网络中使用。这些多级交换网络可以在松散耦合的分布式系统中用于各个处理器之间的连接，也可以使用在紧密耦合的系统中用来控制处理器到存储器的通信。一个开关在某个时间只能处在一种状态，所以很显然可能会发生阻塞现象。例如，考虑这类网络的一个简单的拓扑结构，如图9-7所示的欧米伽网络（Omega networks）。如果将开关1A和开关2A都设置成通过状态，CPU 00和存储器模块00之间就可能进行通信。但是与此同时，在CPU 10和存储器模块01之间不可能进行通信。因此，开关1A和开关2A都必须要设置成交叉状态，这样CPU 10和存储器模块01之间才可以进行通信。因此，这种Omega网络是一个阻塞型网络。通过增加更多的开

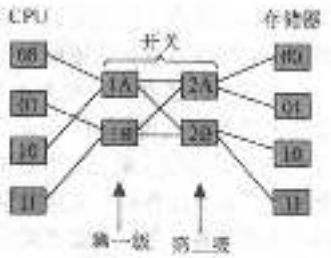


图9-7 一个二级的Omega网络

关数目和更多的开关级数可以构建非阻塞型的多级网络。一般来说，一个 n 节点的 Omega 网络需要有 $\log_2 n$ 级开关和每一级有 $n/2$ 个开关。

令人感兴趣的是，配置这些开关实际上并没有看起来那么困难。当在网络上传输消息时，传输的目的模块的二进制表示可以用来对这些开关进行编导控制。对于多级网络的每一级都使用目的地址的一位，基于这个位的值对每个开关进行编程。如果这个位是 0，就把输入连接到上部输出。如果这个位是一个 1，就把输入连接到下部输出。例如，假设 CPU 00 希望与存储器模块 01 进行通信，则可以使用目的地址的第一位（0）将开关 1A 设置成通过状态（即把输入连接到上部输出），使用第二位（1）将开关 2A 设置成交叉状态（即把输入连接到下部输出）。如果 CPU 11 希望与存储器模块 00 进行通信，则将开关 1B 和开关 2A 都设置成交叉状态（因为这两个输入都必须连接到上部输出）。

另一种确定开关设置的方法是比较源地址和目的地址的对应位。如果这些位相同，就把开关设置成通过状态。如果这些位不同，则将开关设置成交叉状态。例如，假设 CPU 00 希望与存储器模块 01 进行通信。则可以比较两个对象的第一位（0 和 0），把开关 1A 设置成通过状态。并比较它们的第二位（0 和 1），然后把开关 2A 设置成交叉状态。

互连处理器的各种方法都有其优缺点。例如，当只有中等规模数量的处理器时，基于总线的网络是最简单也是效率最高的互联解决方法。然而，当有许多处理器都同时请求访问存储器时，总线就会变成传输的瓶颈。表 9-2 给出了基于总线的网络、交叉开关网络和多级互联网络的特性比较。

表 9-2 各种不同的互连网络的特性比较

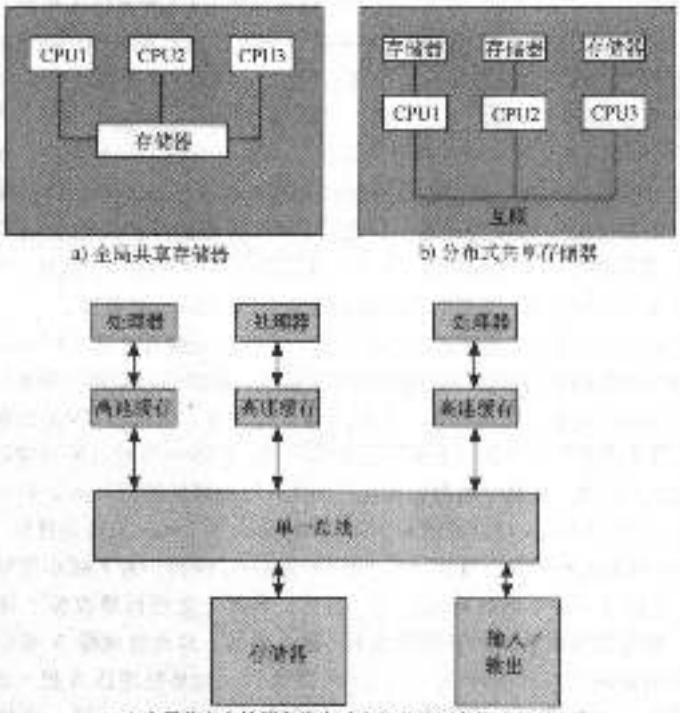
特性	总线网络	交叉式网络	多级互联网络
带宽	低	高	中等
成本	低	高	中等
可靠性	低	高	高
可配置性	高	低	中等
复杂性	低	高	中等

9.4.4 共享存储器的多处理器

前面曾提过，多处理器可以按照存储器的组织方式进行分类。紧密耦合式系统使用相同的存储器，称为共享存储器的处理器（shared memory processors）。但是这并不意味着所有的处理器都必须共享一个很大的存储器。每个处理器都可以有一个局部存储器，而且这个局部存储器必须可以被其他处理器共享。同样，也可以让每个处理器都拥有一个局部高速缓存和一个单一的全局存储器。图 9-8 说明了上述的这二种观点。

共享存储器的多处理器（SMM）的概念可以追溯到 20 世纪 70 年代。当时，Carnegie-Mellon 大学使用交叉开关将 16 个处理器和 16 个存储器模块连接起来构建了世界上第一台 SMM 机器。早期受到人们广泛称赞的 SMM 机器是由 16 个 PDP-11 处理器和 16 个存储器仓库组成的 cm* 系统，所有的这些组件都利用一个树形网络连接。在所有的处理器中均等划分系统中的全局共享存储器。如果某个处理器生成了一个地址，那么处理器会首先检查其局部存储器。如果在局部存储器中没有找到这个地址，就将这个地址传送到控制器。然后，控制器将在占据子树的处理器中尝试查找该地址。对于子树而言，该处理器就是根。如果所请求的地址还是没有找到，那么这个请求将被传送到树的上层直到找到该数据为止，或者系统到其他地方查找。

市场上已经有了一些商业 SMM 系统，但这些机器并不是特别流行。其中一个最早的商业 SMM 计算机是 BBN（Bolt，Beranek 和 Newman）公司的 Butterfly。这台计算机使用了 256 个 Motorola 公司 68 000 处理器。最近，市场上可以购买到 KSR-1（来自 Kendall Square Research）系统，这种系统将会被证明对于计算科学应用程序非常有用。每个 KSR-1 的处理器都包括一个高速缓存，但是这个系统没有主存储器。系统是通过保持在每个处理器中的高速缓存目录来对数据进行访问。KSR-1 中的处理单元是采用一个单向环形拓扑结构来进行连接的，如图 9-9 所示。消息和数据只能在环上进行单向传输。每个一级环可以连接 8 到 32 个处理器。一个二级环形结构最多可以连接 34 个一级环，也就是最大可以连接 1088 个处理器。假如某个处理器需要引用一个位于地址 x 处的数据项时，则包含地址 x 的



a) 全局共享存储器　　b) 分布式共享存储器

c) 全局共享存储器和带有独立的高速缓存的处理器

图 9-8　共享存储器

处理器高速缓存就会把这个请求的高速缓存块放到连接环上，其入口条目（包含这个请求的数据）将沿着环发生迁移，直到到达发出请求的处理器为止。这种类型的分布式共享存储器系统称为共享虚拟存储器系统（shared virtual memory systems）。

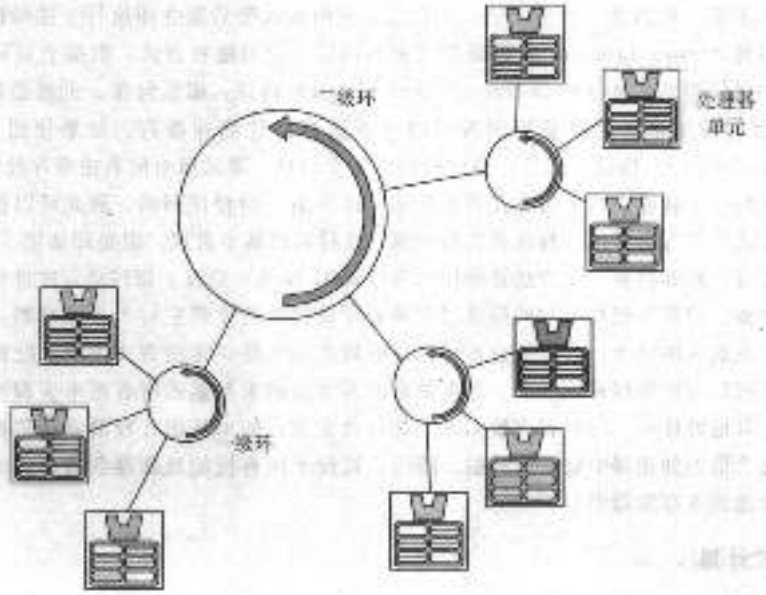


图 9-9　KSR-1 环形拓扑的层次结构

按照存储器操作的同步方式，我们可以把共享存储器的 MIMD 机器分成两类。在*均匀存储器访问*（Uniform Memory Access，UMA）的系统中，所有的存储器访问都花费相同的时间。一台 UMA 机器有一个共享的存储器池，它通过一条总线或一个开关网络与一组处理器相连接。按照已经建立的互连网际协议，所有的处理器都能均等地访问存储器。当处理器数目 n 增加时，在一个 UMA 机器上的开关互连网络（需要有 2^n 个连接）很快就变得非常昂贵。基于总线的 UMA 系统，在当总线的带宽不足以满足系统中的处理器数目要求时，就会变得饱和。而多级网络在处理器的数目变得非常大时，也会遭到物理连线上的限制和时间上的重大延迟。因此，UMA 机器的可扩展性受到了互连网络特性的制约。对称多处理器是著名的 UMA 体系结构。UMA 计算机的一些特殊例子包括：Sun 的超级企业服务器、IBM 的 i 系列和 p 系列服务器，HP 的 900 服务器和 DEC Alpha 服务器。

为了解决 UMA 体系结构中的一些固有的内在问题，*非均匀存储器访问*（nonuniform memory access，NUMA）的计算机为每个处理器都提供一个自己的存储器块。系统的存储器空间被分配到所有的处理器上，但是这些处理器都会把这个存储器看作一个连续编址的实体。虽然 NUMA 的存储器构成了一个单一编址的实体，但是这个实体的分布特征并不是完全透明的。读取距离较近的存储器要比读取较远处的存储器花费更少的时间。因此，在整个机器的地址空间内，存储器的访问时间是不一致的。NUMA 体系结构的实例包括 Sequent 的 NUMA-Q 计算机和 Silicon Graphics 的 Origin2000 计算机。

NUMA 机器容易出现*高速缓存一致性*（cache coherence）问题。为了减小存储器的访问时间，每一个 NUMA 处理器都保持一个专用的高速缓存。然而，当某个处理器修改位于其自己的局部高速缓存中的一个数据时，就会与这个数据的其他副本不一致。例如，假设处理器 A 和处理器 B 在其自己的高速缓存存储器中都有数据元素 x 的拷贝。假设 x 的值为 10。如果处理器 A 把 x 设置为 20，处理器 B 的高速缓存（仍旧包含一个数值 10）中就有一个旧的或过时的（stale）x 值。这种数据的不一致性是不允许出现的，因此必须提供某些处理机制以确保高速缓存的一致性。我们可以利用一种专门设计的，称为 Snoopy cache controllers 的硬件单元来监视整个系统中的所有高速缓存。这种窥视高速缓存控制器会执行系统的高速缓存一致性协议。应用窥视高速缓存技术并维持高速缓存一致性的 NUMA 机器被认为是*高速缓存一致性的* NUMA（cache coherent NUMA，CC-NUMA）体系结构。

解决高速缓存一致性问题最简单的方法就是要求具有过时值的处理器，或者从高速缓存中将过时的 x 值删除，或者把 x 更新为一个新的值。如果当 x 的值被改变后就立即执行上述操作，那么这种系统采用了一种*写通*（write-through）高速缓存的更新协议。使用这种方式，数据会被同时写入高速缓存和存储器。如果使用*写更新*（write-through with update）协议，那么包含 x 的新值消息将会广播到所有其他高速缓存控制器中，以便控制器可以更新其自己的高速缓存。如果使用*写无效*（write-through with invalidation）协议，那么广播中将包含一个消息，要求所有的高速缓存控制器将过时的 x 值从它们的高速缓存中移除。写无效方式只是在第一次更新 x 时使用网络，因此可以保持总线通信量较轻。写更新方式让所有高速缓存都保持当前的值，这样可以减小延迟，但是却增加了总线的通信量。

保持高速缓存一致性的第二种方法是使用*回写*（write-back）协议，回写协议在进行修改时只改变高速缓存中的数据。只有等到修改过的高速缓存块必须被替换时才将它写入主存储器，否则不修改主存储器的内容。在这种协议下，数据值在被写入存储器之前先按正常的方式读取，处理器必须在得到修改数据的专有权后才能执行修改任务。数据的修改是通过请求数据的所有权来实现的。在数据的所有权被授予后，其他处理器上的所有该数据副本都将会失效。如果其他处理器希望读取这个数据，则都必须从拥有这个值的处理器中请求该数据。随后，被授予所有权的处理器会放弃它的所有权，并把这个当前数据传送到主存储器中。

9.4.5 分布式计算

分布式计算是多重处理的另外一种形式。尽管分布式计算已经成为了计算科学中一种越来越流行的方式，但是这对大家来说并不意味着有什么不同。从某种意义上来说，所有的多处理器系统同样都

是分布式系统，因为它们都是把要处理的工作负荷划分到一组处理器，让这些处理器一起协作来解决一个问题。但是大多数人在使用分布式系统（distributed system）时，通常指一种松散耦合的多计算机系统。大家记得，多处理器可以使用局部总线进行连接（参见图 9-8c），或者是通过网络来进行连接，如图 9-10 所示。松散耦合的分布式计算机通过网络在处理器之间进行通信。

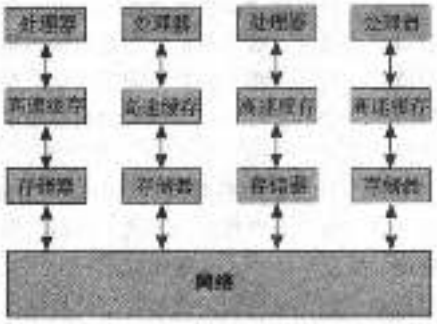


图 9-10 通过网络连接的多处理器

最近，人们正在使用单独的微型计算机和工作站网络（NOW）来进行分布式系统的实践。这种分布式计算的思想也得到了实例的证明。这些系统可以让那些闲散的 PC 处理器来处理某个大型问题中的一个很小的部分。最近，人们通过利用数千台个人计算机的资源解决了一个与密码相关的问题。每台计算机都通过使用少数几个的可能的消息密码来实现非常复杂的密码分析。

位于美国 Berkeley 的加利福尼亚大学有一个射电天文学研究小组，SETI（寻找外星生命，Search for Extra Terrestrial Intelligence），他们的任务就是分析来自射电天文望远镜的数据。为了帮助完成这个计划，PC 机用户可以在他们的家庭计算机上安装一个 SETI 屏幕保护程序。这个屏幕保护程序会在处理器的空闲时间中分析信号数据。SETI 是一个 NOW 体系结构。这个计划非常成功，在短短的 18 个月中就积累了 50 万年的 CPU 计算时间。

对于一般用途的计算工作，在分布式系统中透明性的概念是非常重要的。我们应该尽可能地隐藏掉分布式网络的许多细节问题。而且，使用远程系统资源应该像使用本地系统资源一样轻松方便。

远程过程调用（remote procedure calls，RPC）扩展了分布式计算的概念，有助于为资源共享提供必要的透明性。通过使用远程过程调用，一台计算机可以通过调用一个过程来使用另一台计算机上的可用资源。这个调用的过程本身驻留在远程的机器上，但是这个远程调用的操作类似于在本地调用系统内部的一个过程调用。我们可以通过微软公司的分布式组件对象模型（Distributed Component Object Model，DCOM），开放组织的分布计算机环境（Distributed Computing Environment，DCE），通用对象请求代理体系结构（Common Object Request Broker Architecture，CORBA）和 Java 的远程方法调用（Remote Method Invocation，RMI）来实现远程过程调用。今天的软件设计师都倾向于面向对象的分布式计算的观点，这也就使得 DCOM、CORBA 和 RMI 变得流行。

9.5 新的并行处理方法

各种计算机的书籍都在努力地介绍一些特别新的先进的计算机体系结构。尽管我们不能在这个简短的章节里讨论所有这些体系结构，但是将会给大家介绍几种有别于传统的冯·诺伊曼体系结构的著名计算机体系结构。这些系统采用一些新的思考方式来看待计算机和计算。它们包括数据流计算，神经网络和脉动阵列处理。

9.5.1 数据流计算

冯·诺伊曼机器所使用的是按顺序控制的流程。程序计数器会决定要执行的下一条指令。在这里，数据和指令是分离的。如果程序计数器的数值按照引用一个数据值的语句的输出结果发生了变化，则数据是改变指令执行顺序的唯一方法。

在数据流（dataflow）计算中，程序的控制直接与数据本身相关联。数据流计算是一种非常简单的方式：当执行中需要的数据可用时，就执行一条指令。因此，指令的实际顺序和指令最终执行的顺序没有关系。执行流完全由数据相关性决定。在这类系统中没有共享数据存储器的概念，也没有控制执行流程的程序计数器。数据连续地流动，而且在同一时间适用于多条指令。每一条指令都被认为是一个独立的

过程。指令不会引用存储器，相反，这些指令会引用其他指令。数据是从一条指令传递到下一条指令。

我们可以通过分析数据流系（data flow graph）来理解数据流计算机的计算顺序。在数据流图中，节点代表指令，弧形指示指令之间的数据相关性。数据以数据令牌（data tokens）的形式在数据流图上流动。当一条指令拥有所需的全部数据令牌时，其节点就会激活（fire），数据令牌将会通过节点。当激活一个节点时，节点就会消耗数据令牌，执行所要求的操作，并且将作为结果的数据令牌放到一个输出弧形上。图 9-11 说明了这种思想。

图 9-11 中的数据流图给出的是一个静态数据流的体系结构的例子。在这种静态体系结构中，令牌以一种分级流水线作业的方式在数据流图中流动。而在一个动态数据流的体系结构中，令牌中标记有上下文（环境）信息，而且会被存储到一个存储器中。在每个时钟周期内，都会搜索存储器，查找激活节点所需要的一组令牌。只有节点在同一环境（上下文）中找到一组完整的输入令牌时，节点才会被激活。

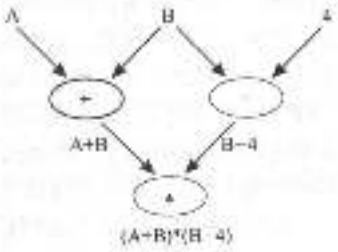


图 9-11 计算 $N-(A+B)\times(B-4)$ 的数据流图

数据流机器的程序必须用为这种体系结构专门设计的语言来编写。这种编程语言包括 VAL、Id、SISAL 和 LUCID。数据流程序的编译会产生一个类似于图 9-11 中所示的数据流图。当程序执行时，这些令牌将沿着弧形向前传播。

下面考虑计算 N! 的示例代码：

```
(initial j < -n;k < -1
While j > 1 do
  New k <  k * j;
  New j <  j - 1;
return k)
```

对应的数据流图如图 9-12 所示。其中有 N 和 1 两个值，被输入到数据流图中。N 转变成令牌 j。当 j 与 1 进行比较时，如果它比 1 大，j 令牌就可以被传输通过并且会被加倍复制。其中一个令牌的副本会被传递通过“−1 节点”，而另外的一个副本会传递通过乘法节点。当 j 不大于 1 时，k 的值将会作为程序输出结果被传递通过。只有当新的 j 令牌和新的 k 令牌两个都可用时，才会激活乘法节点。输入 N 和 1 的两个“向右走（right facing）”的三角形是“合并”的节点，而且只要有任何一个输入值可用时，这两个节点都会激活（执行操作）。一旦 N 和 1 被送入图中，他们将会被“耗尽”，而新的 j 值和新的 k 值将引发合并节点激活。

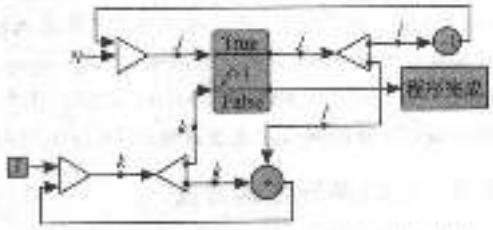


图 9-12 计算 N! 的程序相对应的数据流图

1977 年，美国 Utah 大学的 Al Davis 建造了世界上第一台数据流计算机。1979 年，在法国的 CERT-ONERA 研发了第一个多处理器的数据流系统（其中包含 32 个处理器）。英国曼彻斯特大学在 1981 年创建了第一台加标记令牌（tagged-token）的数据流计算机。具有商业价值的曼彻斯特标记数据流模型（Manchester tagged dataflow model）是基于动态标记方法建造的，是一种功能强大的数据流计算机的范例。这种特殊的体系结构被描述为标记模型，是因为各种数据值（令牌）都采用了它们唯一的标识符来标记。这样利用这些标示符可以说明和指示当前程序的迭代层次。由于程序可以是可重入的，所以需要使用标记符。可重入意味着可以使用具有不同数据的同一个代码。通过比较这些标记符，系统可以决定在每一次迭代过程中使用哪一个数据。对于同一条指令来说，已经匹配标记符的令牌会促使这个节点激活。

我们可以通过一个循环程序的例子很好地解释标记的概念。如果这个循环的每次迭代都是在子图

的一个单独实例中执行，那么就可以实现一个更高层次的并发。这个子图只是该循环的数据流图的一个副本。但是，如果在循环中有很多的迭代过程，那么也就会有很多的数据流图的副本。程序可以在一个图的不同实例中共享节点，这样会比多次重复复制一个图更好、更有效率。对于每一个实例来说，它们的令牌都必须是可标识的，而可以通过给每一个令牌加上一个标记来实现。每个令牌的标记都是用来标识令牌所属的实例。这就是说，用于第三次迭代的令牌不能促使第四次迭代中的节点激活。

数据流机器的体系结构由许多相互之间必须发生通信的处理单元组成。每个处理单元都有一个*使能单元*（或称为启动单元，enabling unit），使能单元会按顺序接受输入的令牌，并将它们存放在存储器中。当有某个被编址的令牌激活一个节点时，就会从存储器中抽取出相关的输入令牌，并将这些令牌与该节点本身结合起来形成一个可执行的包。由处理单元中的*功能单元*（functional unit）计算出一些必要的输出值，并将这些输出值和目的地址组合起来形成更多的令牌。然后，这些令牌会在它们启动其他节点的时候被送回使能单元。在标记令牌的机器中，使能单元分成两个独立的阶段：*匹配单元*（matching unit）和*提取单元*（fetching unit）。匹配单元把输入令牌存储到存储器中，并决定一个给定目的节点的实例是否已经激活。在标记型的体系结构中，目的节点的地址和标记都必须匹配。对于一个节点，当所有匹配的令牌都可用时，就会将这些令牌送到提取单元，提取单元会把这些令牌和该节点的一个副本组合成一个可执行的包，然后这个可执行的包被传送到功能单元。

因为在数据流系统中是数据驱动处理流程，所以数据流的多处理器不会发生竞争和高速缓存不一致的问题。显然，这些问题对于设计控制驱动的多处理器的人来说是非常头痛的事情。非常有趣的是，冯·诺伊曼曾经研究过在本质上类似于数据流计算机的数据驱动的体系结构的可能性，而冯·诺伊曼的名字在传统计算机体系结构中却被赋予了冯·诺伊曼瓶颈。特别地，冯·诺伊曼还研究了神经网络的可行性。从性质上来说，神经网络也属于数据驱动类型，我们将在下一部分进行讨论。

9.5.2 神经网络

传统的计算机体系结构非常适合于一些需要进行快速算述运算和执行确定性程序。然而，传统体系结构不太适合具有大量并行应用序、大量容错、或者需要适应变化环境的情形。另一方面，*神经网络*（neural networks）在不能用公式来表示一个确切的算法方案的动态环境中非常有用。在这里，处理过程通常是要基于前面行为的一种累积结果。

冯·诺伊曼计算机是基于处理器/存储器的体系结构，而神经网络则是基于人脑的并行体系结构。神经网络试图实现简单的生物神经网络形式。神经网络代表了一种新的具有高度互连性和简单处理单元的多处理器计算的形式。神经网络可以处理不精确的和随机性的信息，并且具有某些机制允许处理单元之间发生自适应的相互作用。神经网络（或者称为 neural nets）就像生物学的网络一样，可以从经验中学习。

神经网络计算机由大量简单的处理单元组成。每个单元都各自处理一个大问题中的一小部分。简单地说，神经网络由处理单元（PE）组成，这些处理单元会将各种输入值乘以不同的权（重）值，产生一个单一的输出值。与神经网络有关的实际计算过程会让人觉得很简单；一个神经网络真正的处理能力在于互连的 PE 的并行处理和各组权值的自适应的特性。创建神经网络的困难在于如何确定神经元之间的连接方式，在网络边缘应该设置多大的权值，以及应该如何在这些权值上施加不同的阈值。此外，当神经网络在学习时，可能会出错。当这种情况发生时，这些权值和阈值都必须做出改变以补偿修正这个错误。神经网络的*学习算法*（learning algorithm）是用来管理如何改变这些权值和阈值的一组规则。

神经网络还有许多人们所熟知的其他不同的名称，包括*连接器系统*（connectionist）、*自适应系统*（adaptive system）和*并行分布式处理系统*（parallel distributed processing system）。如果神经网络可以使用一个大型的包含大量的先前例子的数据库和从先前的经验中学习的话，那么这些系统的功能会特别强大。神经网络已经非常成功地应用在大量实际应用程序中，包括质量管理，天气预报，财政和经

济预报，语音和模式识别，石油和天然气开发，卫生保健成本降低，破产预测，机器诊断，证券交易和市场营销。值得注意的是，每一种神经网络都是专为某个特定任务而设计的。所以，我们不能利用一个为预报天气而设计的神经网络来预测经济，并期望它对经济的预测做得很好。

一个最简单的神经网络的实例是感知器（perceptron），它是一个单一的可训练的神经元。感知器会根据从多个输入接收到的值产生一个布尔输出。感知器是可训练的，因为其阈值和输入权值都是可修改的。图 9-13 描述了一个具有输入 x_1，x_2，…，x_n 的感知器，这些输入值可以是布尔值，或者是实际值。Z 是感知器的布尔输出。W_i 代表感知器边缘的权值，而且都是一些实际值。T 表示阈值，也为实际值。在这个例子中，当输入 $w_1x_1+w_2x_2+\cdots+w_nx_n$ 大于阈值 T 时，输出值 Z 将为真（1）。否则，Z 的值为 0。

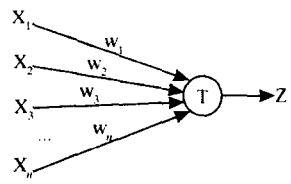


图 9-13 感知器

在特定的输入下，感知器会按照它是如何被训练的来产生一个输出结果。如果这种训练完全正确，则可以给感知器任意输入并且得到合理的正确结果。即使感知器以前从来没有见过某一组特定的输入，感知器也应该能够得出一个合理的输出。输出结果的“合理性”取决于感知器的训练程度的好坏。

感知器的训练过程可以使用监督或无监督学习算法。监督（supervised）学习算法假设先前的知识都是正确的结果，这些正确的知识会在训练阶段输入到神经网络中。学习时神经网络会被告知最后的结果是否正确。如果输出的结果不正确，神经网络就会修改输入权值以得到所要求的输出结果。无监督（unsupervised）学习算法是在训练过程中不给神经网络提供正确的输出。神经网络只是适应着去响应它的各种输入，学习如何识别输入组合中的各种模式和结构。在本例中，我们采用的是监督学习算法。

训练一个神经网络的最好办法是编译一个大范围的实例，可以展现出我们所感兴趣的各种特征。因为一个神经网络最多可以做到与训练的数据一样好，所以必须非常仔细地选择足够多数量的正确实例。例如，如果一个小孩所看到过的唯一的一只鸟是一只鸡，那么我们显然不能期望他能够识别所有的鸟。通过为感知器提供一些输入值，然后检查感知器的输出结果，这样训练过程就发生了。如果输出的结果不正确，感知器就会被告知需要改变其输入权值和阈值以避免在将来发生同样的错误。而且，如果我们给一个小孩看一只鸡、一只麻雀、一只鸭子、一只鹰、一只鹅鹏和一只乌鸦，我们不能期待这个孩子只看了一次后就可以记住这些动物的相似和不同的地方。类似地，神经网络也必须多次看相同的例子，才能推断出这些输入数据的特性。

人们可以非常容易地训练一个感知器识别出 AND 操作。假设有 N 个输入，只有当所有的输入都等于 1 时，输出结果才为 1。如果将感知器的阈值设置为 n，所有的边缘的权值都设为 1，那么就得出了正确的结果。另一方面，为了计算一组输入值的 OR，这里只需要简单地将阈值设置为 1。这样，只要有一个输入为 1 时，输出也将为 1。

对于 AND 和 OR 这两个操作符来说，我们知道阈值和权值应该是多少。但是，对于一些复杂的问题，这些值是不知道的。例如，如果我们不知道为 AND 操作的权值，我们可以从权值 0.5 开始，给感知器以不同的输入值，当输出产生不正确的结果时就对权值进行修改。神经网络就是这样训练完成的。通常，神经网络的初始权值是一个在 −1 到 1 之间随机数值。一个正确的训练往往需要上千个步骤。训练所需的时间取决于网络的大小。当感知器的数量增加时，可能的“状态”数目也会随之增加。

下面，我们考虑一个更复杂的例子，确定一辆坦克是否隐藏在一张图片中。首先可以对神经网络进行设置，使神经网络的每一个输出值都与一个确切的像素相关联。如果这个像素是图像中坦克的一部分，神经网络应该输出一个 1。否则，网络的输出结果为 0。输入的信息很可能是由像素的颜色组成。通过输入大量包含有坦克和无坦克的图片来对这个神经网络进行训练。这种训练会一直持续下去，直到神经网络能够正确地识别图片中是否包含坦克为止。

美国军方曾经从事过一项类似于上面描述过的研究计划。研究中使用了两组照片，其中一组的一百张照片中包含隐藏在树和灌木丛后面的坦克，另外一组的一百张照片中只有普通风景而没有坦克。

先从每组中各取出 50 张照片“秘密”放置起来，而用剩下的照片来训练神经网络。在每次输入一张照片前，神经网络先利用一个随机的权值进行初始化。当神经网络的输出结果不正确时，就调整其输入权值直到能得到正确的结果。在神经网络完成训练过程后，再将每组中的 50 张“秘密”照片送到神经网络，结果这个神经网络正确地识别了每一张照片中是否存在坦克。

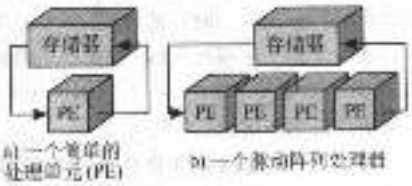
这里，我们需要解决训练的一个实际问题是，神经网络是否真地学会了如何识别坦克？美国国防部五角大楼的这个很自然的疑问引来了更多的测试。他们拍摄了另外的一些照片，并将它们输入到神经网络。令研究人员沮丧的是，输出的结果完全是随机。这个神经网络不能正确地识别照片中的坦克。在经过了一番研究后，研究人员发现原先试验的 200 张照片中，所有存在坦克的照片都是在一个阴天拍摄的，而那些没有坦克的照片都是在晴天拍摄的。这个神经网络就是根据天空的颜色来恰当地区分了这两组照片，而并不是根据照片中是否存在一辆隐藏的坦克。这样一来，政府部门现在就非常自豪拥有了一个价昂贵的神经网络，这个神经网络可以正确地区分晴天和阴天。

这是一个大家都认为有关神经网络的最大问题的一个很好的例子。如果存在大于 10 到 20 个神经元的情况下，想要理解神经网络是如何得出其输出结果简直就是不可能的。人们不可能告诉你神经网络是否正在基于正确的信息做出决定，或者就像上面介绍的例子，有些事情完全是不相关的。神经网络具有一种非凡的能力，就是它能够从人脑难以分析复杂数据中推演出这些数据的含义和提取出数据的模式。但是，有些人相信神经网络能在被训练的领域成为专家。神经网络可以应用在销售预测、风险管理、客户研究、海底矿藏探测、面部识别，以及数据验证这些领域。神经网络前景非常光明，而且在过去的几年中所取得了进步，并且已经有大批的资金投入到神经网络的研究中。然而，还是有很多人对一些人类还不可能完全理解的事情缺乏信心。

9.5.3 脉动阵列

脉动阵列（systolic array）计算机的名字源于它与生物中血液如何有规律地流过心脏的相似情形。脉动阵列计算机是一种处理单元的网络，它通过数据不断在系统中循环而有节奏地对数据进行计算。它是 SIMD 计算机的一种变化形式，并由大量使用先进流水线处理数据流的简单处理器组合构成，如图 9-14b 所示。自从 20 世纪 70 年代引入脉动阵列计算机以来，脉动阵列计算机已经对特殊功能的计算产生了重要的影响。一个著名的脉动阵列是 CMU 的 iWarp 处理器，1990 年由 Intel 制造完成。这个系统由通过一条双向数据总线连接的处理器的线性阵列组成。

虽然图 9-14b 描述的只是一个一维的脉动阵列，但是二维脉动阵列也是很常见的，随着 VLSI 技术的不断发展，三维阵列也正变得流行起来。



a) 一个简单的处理单元(PE)　　b) 一个脉动阵列处理器

图 9-14

脉动阵列采用一种高度并行执行的机制（通过流水线），而且可以保持一个非常高的吞吐量。通常，脉动阵列的连线简短、设计简单，而且具有很好的可缩放性。现在，脉动阵列计算机也趋向于更加坚固、高度紧凑、高效率和生产成本更加便宜。缺点是，脉动阵列计算机是高度专用化的，因此它们所能解决的问题类型与大小都是很难发生改变的。

应用脉动阵列的一个很好例子是多项式的计算。为了计算多项式 $y=a_0+a_1x+a_2x^2+\ldots+a_nx^n$，可以使用 Horner 法则：

$$Y=(\cdots(a_nx+a_{n-1})\times x+a_{n-2})\times x+a_{n-3})\times x\cdots a_1)\times x+a_0$$

按照 Horner 法则，使用一个处理器成对排列的线性脉动阵列来计算多项式的值，如图 9-15 所示。处理器会把它的输入乘以 x，并将结果传递到右边的处理器。下一个处理器则把传来的结果加上 a_i，再将结果继续向右传递。从计算开始，经过最初的 $2n$ 个周期的延迟后就可以完



图 9-15　使用脉动阵列计算一个多项式

成整个计算，每个周期都要计算一项多项式。

脉动阵列通常用于一些需要重复计算的任务，包括傅立叶（Fourier）变换、图像处理、数据压缩、最短路径问题、排序分类、信号处理和各种各样的矩阵计算（例如矩阵的转置和乘法）。简而言之，脉动阵列计算机非常适合于需要使用大量简单处理单元来进行并行处理的计算问题。

本章小结

本章概述了一些有关多处理器和多计算机系统的重要问题。这些系统提供了一些高效地处理各种复杂问题的方法。

有关 RISC 和 CISC 的争论越来越多地集中到了芯片体系结构的比较，而不是指令系统。人们真正所关心的是程序的执行时间。无论是 RISC 的设计者，还是 CISC 的设计者都会继续不断地改进系统的性能。

Flynn 分类法是按照指令和数据流的数量来对计算机的体系结构进行分类。MIMD 类型的机器可以进一步划分为使用共享存储器和不使用共享存储器的两种类型。

现在的数字计算机具有惊人的处理能力。在超标量体系结构和超流水线体系结构中，内部处理器的并行执行机制已经大大提升了系统的处理能力。原先，处理器一次只能做一件事情，但是现在处理器执行多个并发的操作已经是一件很普通的事。矢量处理器支持矢量操作，而 MIMD 计算机中包括多个处理器。

SIMD 处理器和 MIMD 处理器之间可以通过互连网络进行连接。共享存储器的多处理器系统可以将一组物理存储器的集合作为一个单一的实体，而分布式存储器的体系结构则允许每个处理器单独地访问自己的存储器。这两种方法都允许普通用户以可承受的价格得到超级计算。目前，最流行的多处理器体系结构是 MIMD，共享存储器和基于总线的多处理器系统。

使用传统的计算模型不可能解决一些高度复杂的问题。新的体系结构对解决特殊的应用问题是十分必要的。数据流计算机使用数据来驱动和控制计算过程，而不是其他方式。神经网络会自我学习来解决某些高度复杂的问题。脉动阵列计算机会组合大量小处理单元的处理能力，推动数据穿过整个脉动阵列直到问题解决。

深入阅读

人们曾经多次尝试修改 Flynn 分类法。Hwang（1987）、Bell（1989）、Karp（1987），以及 Hockney 和 Jesshope（1988）都对 Flynn 分类法进行了不同程度的扩充。

现在有许多介绍高级体系结构的好的教科书，其中包括 Hennessy 和 Patterson（1990），Hwang（1993）和 Stone（1993）的著作。Stone 在书中详细讨论了流水线操作和存储器组织，矢量计算机和并行处理。要想详细了解现代 RISC 处理器中的超标量体系结构，可以阅读 Grohoski（1990）的文章。在 Patterson（1985）以及 Patterson 和 Ditzel（1980）的论文中非常透彻地讨论了 RISC 的原理，并很好地解释了指令流水线的工作流程。

有关 VLSI 技术和计算机处理器设计之间的相互影响的优秀文章，可以参阅 Hennessy（1984）的论文。Leighton（1982）的书籍虽然有些过时，但在体系结构和算法方面有很好的见解。同时，它也是有关互连网络的一本很好的参考书。Omondi（1999）对计算机微型体系结构的实现有一个权威性的介绍。

有关数据流体系结构和不同的数据流机器之间比较分析的优秀研究文章，可以参见 Dennis（1980）、Hazra（1982）、Srini（1986）、Treleaven et al.（1982）和 Vegdahl（1984）的论文。

参考文献

Amdahl, G. "The Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities." *AFIPS Conference Proceedings 30*, 1967, pp. 483–485.

Bell, G. "The Future of High Performance Computers in Science and Engineering," *Communications of the ACM*. Vol. 32, pp. 1091–1101, 1989.

Bhuyan, L., Yang, Q., and Agrawal, D. "Performance of Multiprocessor Interconnection Networks. *Computer*, *22*:2, 1989, pp. 25–37.

Circello, J. et al. "The Superscalar Architecture of the MC68060." *IEEE Micro*, *15*:2, April 1995, pp. 10–21.

Dennis, J. B. "Dataflow Supercomputers." *Computer*, *13*:4, November, 1980, pp. 48–56.

Dulon, C. "The IA-64 Architecture at Work." *Computer*, *31*:7 (July 1998), pp. 24–32.

Flynn, M. "Some Computer Organizations and Their Effectiveness." *IEEE Transactions on Computers*, Vol. C-21, p. 94, 1972.

Goodman, J., & Miller, K. *A Programmer's View of Computer Architecture*. Philadelphia: Saunders College Publishing, 1993.

Grohoski, G. F. "Machine Organization of the IBM RISC System/6000 Processor." *IBM J. Res. Develop. 43*(1), January 1990, pp. 37–58.

Hazra, A. "A Description Method and a Classification Scheme for Dataflow Architectures." *Proceedings of the 3rd International Conference on Distributed Computing Systems*, October 1982, pp. 645–651.

Hennessy, J. L. "VLSI Processor Architecture." *IEEE Trans. Comp. C-33*(12) (December 1984), pp. 1221–1246.

Hennessy, J. L., & Patterson, D. A. *Computer Architecture: A Quantitative Approach*, San Francisco: Morgan Kaufmann, 1990.

Hockney, R., & Jesshope, C. *Parallel Computers 2*, Bristol, United Kingdom: Adam Hilger, Ltd., 1988.

Horel, T., & Lauterbach, G. "UltraSPARC III: Designing Third Generation 64-Bit Performance." *IEEE Micro*, *19*:3 (May/June 1999), pp. 73–85.

Hwang, K. *Advanced Computer Architecture*. New York: McGraw-Hill, 1993.

Hwang, K. "Advanced Parallel Processing with Supercomputer Architectures." *Proc. IEEE*, Vol. 75, pp. 1348–1379, 1987.

Karp, A. "Programming for Parallelism." *IEEE Computer*, *20*(5), pp. 43–57, 1987.

Kogge, P. *The Architecture of Pipelined Computers*. New York: McGraw-Hill, 1981.

Leighton, F.T. *Introduction to Parallel Algorithms and Architectures*. New York: Morgan Kaufmann, 1982.

McLellan, E. "The Alpha AXP Architecture and 21164 Alpha Microprocessor." *IEEE Micro*. *15*:2 (April 1995), pp. 33–43.

MIPS home page: www.mips.com.

Nitzberg, B., & Lo, V. "Distributed Shared Memory: A Survey of Issues and Algorithms." *IEEE Computer*, *24*:8, pp. 52–60, 1991.

Omondi, Amos. *The Microarchitecture of Pipelined and Superscalar Computers*. Boston: Kluwer Academic Publishers, 1999.

Ortega, J. *Introduction to Parallel and Vector Solution of Linear Systems*. New York: Plenum Press, 1988.

Patterson, D. A. "Reduced Instruction Set Computers." *Communications of the ACM 28*(1) (January 1985), pp. 8–20.

Patterson, D. & Ditzel, D. "The Case for the Reduced Instruction Set Computer." *ACM SIGARCH Computer Architecture News*, October 1980, pp. 25–33.

Patterson, D. A., & Hennessy, J. L. *Computer Organization and Design: The Hardware/Software Interface*, *2nd ed.* San Mateo, CA: Morgan Kaufmann, 1997.

Reed, D., & Grunwald, D. "The Performance of Multicomputer Interconnection Networks." *IEEE Computer*, June 1987, pp. 63–73.

Siegel, H. *Interconnection Networks for Large Scale Parallel Processing: Theory and Case Studies*. Lexington, MA: Lexington Books, 1985.

Silc, Jurij, Robic, B., & Ungerer, T. *Processor Architecture: From Dataflow to Superscalar and Beyond*. New York: Springer-Verlag, 1999.

SPARC International, Inc., *The SPARC Architecture Manual: Version 9*. Upper Saddle River, NJ: Prentice Hall, 1994.

SPIM home page: www.cs.wisc.edu/~larus/spim.html.

Srini, V. P. "An Architectural Comparison of Dataflow Systems." *IEEE Computer*, March 1986, pp. 68–88.

Stallings, W. *Computer Organization and Architecture, 5th ed.* New York: Macmillan Publishing Company, 2000.

Stone, H. S. *High Performance Computer Architecture, 3rd ed.* Reading, MA: Addison-Wesley, 1993.

Tabak, D. *Advanced Microprocessors*. New York: McGraw-Hill, 1991.

Tanenbaum, Andrew. *Structured Computer Organization, 4th ed.* Upper Saddle River, NJ: Prentice Hall, 1999.

Treleaven, P. C., Brownbridge, D. R., & Hopkins, R. P. "Data-Driven and Demand-Driven Computer Architecture." *Computing Surveys*, *14*:1 (March, 1982), pp. 93–143.

Trew, A., & Wilson, A., Eds. *Past, Present, Parallel: A Survey of Available Parallel Computing Systems*. New York: Springer-Verlag, 1991.

Vegdahl, S. R. "A Survey of Proposed Architectures for the Execution of Functional Languages." *IEEE Transactions on Computers*, *C-33*:12 (December 1984), pp. 1050–1071.

基本概念和术语复习

1. 当时为什么会提出 RISC 体系结构？
2. 为什么 RISC 处理器比 CISC 处理器更容易实现流水线作业？
3. 说明寄存器窗口如何使过程调用的效率更高。
4. Flynn 分类法是基于哪两大特性来分类计算机体系结构的？
5. 我们提议在 Flynn 分类法上增加一个层次。在这一较高层次上，计算机的显著特征是什么？
6. 是否所有的程序问题都能够并行执行？其中的限制因素是什么？
7. 定义超级流水线作业。
8. 超标量体系结构的设计和超级流水线设计有什么区别？
9. VLIW 设计在哪些方面不同于超级流水线设计？
10. EPIC 和 VLIW 之间的异同点是什么？
11. 解释寄存器到寄存器的矢量处理体系结构的固有（内在）局限性。
12. 列出矢量处理器高效的两种原因。
13. 画出 6 种主要互连网拓扑结构的示意图。
14. 有三种类型共享存储器的组织结构。它们是哪三种？
15. 描述本章中所讨论的其中一种高速缓存一致性协议。
16. 什么是 SETI，它是如何使用分布式的计算模型的？
17. 数据流体系结构和“传统”的计算体系结构有什么区别？
18. 什么是可重入代码？
19. 神经网络的基本计算单元是什么？
20. 描述神经网络是如何学习的？
21. 脉动阵列是通过怎样的比喻取得这个名字的？为什么说这个比喻相当地准确？
22. 什么样的问题适合于利用脉动阵列来解决？

练习题

◆1. 为什么 RISC 机器要对寄存器操作？

2. RISC 系统中的哪些特性能够直接在 CISC 系统中实现？又有哪些 RISC 机器的特性不能在 CISC 机器上实现？（参考表 9-1 中列出的两种体系结构的特性的定义）

◆3. 在精简指令集计算机中的“精简”的实际含义是什么？

4. 假设一个 RISC 机器使用具有如下特性的重叠寄存器窗口：

10 个全局寄存器

6 个输入参数寄存器

10 个局部寄存器

6 个输出参数寄存器

请问每个重叠寄存器窗口有多大？

◆5. 一个 RISC 处理器具有 8 个全局寄存器和 10 个寄存器窗口。每个寄存器窗口有 4 个输入寄存器，8 个局部寄存器和 4 个输出寄存器。在这个 CPU 中共有多少个寄存器？（提示：由于窗口的循环特性，最后一个窗口的输出寄存器被共享为第一个窗口的输入寄存器）

6. 一个 RISC 处理器中共有 152 个寄存器，12 个全局寄存器。共有 10 个寄存器窗口，每个窗口有 6 个输入寄存器和 6 个输出寄存器。在每个窗口中有多少个局部寄存器？

7. 一个 RISC 处理器中有 186 个寄存器，18 个全局寄存器。共有 12 个寄存器窗口，每个寄存器窗口有 10 个局部寄存器。在每一个寄存器窗口中有多少个输入和输出寄存器？

8. 假设一个 RISC 机器使用重叠寄存器窗口在各个过程之间进行参数的传递。机器有 298 个寄存器，每个寄存器窗口有 32 个寄存器，其中 10 个是全局变量，而 10 个是局部变量。回答下面问题：

a) 有多少个寄存器可以用来输入参数？

b) 有多少个寄存器可以用来输出参数？

c) 有多少个寄存器窗口可以使用？

d) 在每次过程调用中，当前窗口指针（CWP）会增量多少？

◆9. 回顾第 8 章关于关联转换的讨论。当一个进程停止使用 CPU，而另一个进程开始使用 CPU 时，就会发生关联转换。在这种情形下，寄存器窗口被认为 RISC 机器的一个潜在的缺点。解释为什么会有这种情况？

10. 假设一台 RISC 机器使用 5 个寄存器窗口：

a) 在寄存器的内容必须被保存到存储器之前，过程调用可以进行多深？（也就是说，在我们需要把任意寄存器的内容保存到存储器中前可以产生的“活动的”过程调用的最大数目是多少？）

b) 如果要在 a) 部分中达到最大值后再增加两次调用，那么结果会有多少个寄存器窗口必须被保存到存储器中？

c) 现在假设最新近调用的过程返回。试解释会发生什么情况？

d) 现在假设再多调用一个过程，那么又有多少个寄存器窗口需要保存到存储器中。

11. 在 Flynn 分类法中：

◆a) SIMD 代表什么意思？给出一个简单的描述和举一个例子。

b) MIMD 代表什么意思？给出一个简单的描述和举一个例子。

12. Flynn 分类法包含 4 个主要的计算模型。简要地描述每个类别，并给出每个模型可能使用的一个高级问题的例子。

◆13. 解释松散耦合的体系结构和紧密耦合的体系结构之间的区别。

14. 描述 MIMD 多处理器可以区别于多计算机系统或计算机网络的特性。

15. SIMD 和 MIMD 有什么相似之处？它们之间又有什么区别？注意，这里不是定义术语，而是要比较模型。

16. SIMD 和 SPMD 之间有什么区别？

◆17. SIMD 最适合于什么类型的程序级的并行执行（数据或控制）？而什么类型的程序级的并行执行是最适合 MIMD 的？

18. 针对指令级的并行执行，简要描述和比较 VLIW 模型和超标量体系结构模型。

◆19. 哪一种模型，是 VLIW 还是超标量体系结构，对编译器提出更大的挑战？为什么？

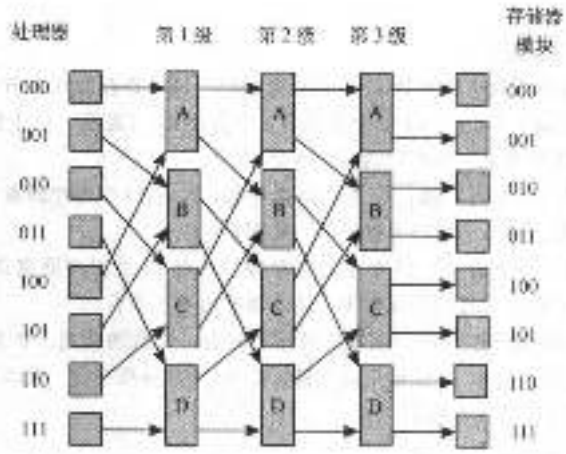
◆20. 比较和对比超标量体系结构和 VLIW 体系结构。

◆21. 为什么需要分布式系统？

22. UMA 和 NUMA 之间有什么不同？

◆23. 在互连网络中使用交叉开关有什么主要问题？而使用基于总线的连接又会出现什么问题？

24. 已知在下面的 Omega 网络中，允许 8 个 CPU（P0 到 P7）访问 8 个存储器模块（M0 到 M7）：



a) 描述下面的各种连接是如何通过网络实现的（解释如何设置每个开关）。引用开关是使用符号 1A，2B 等：

i) P0 -> M2

ii) P4 -> M4

iii) P6 -> M3

b) 这些连接是否可以同时发生，或者它们会发生冲突吗？解释原因。

c) 列出一个在 a) 中没有给出的将会与 P0 -> M2 发生冲突（或被阻塞）的处理器到存储器的访问过程。

d) 列出一个没有在 a) 中给出的不会被 P0 -> M2 阻塞的处理器到存储器的访问过程。

◆25. 描述用于存储器共享系统中的写通和回写高速缓存修改策略，并简述这两种方案的优缺点。

26. 数据流系统中的存储器应该是关联存储器还是基于地址的存储器？解释原因。

◆27. 神经网络是按顺序处理信息的吗？解释原因。

28. 对于神经网络，比较和对比监督学习算法和无监督学习算法。

◆29. 从一个数学的观点来描述神经网络中的有监督的学习过程。

30. 下面有两个问题涉及到神经网络的单一感知器：

a) 逻辑非（NOT）比逻辑与（AND）和逻辑或（OR）的情况稍微难处理一些，但也可以完成。在这种情况下，只有一个布尔值输入。这里设置哪些权值和阈值才能使得这个感知器能够识别逻辑非（NOT）的操作符？

b) 证明对于两个输入 x_1 和 x_2，使用一个简单的感知器不能解决二进制的异或（XOR）问题。

31. 解释 SIMD 与一维脉动阵列计算之间的区别。

32. 相对于 Flynn 分类法，脉动阵列属于哪一类？工作站群集又属于哪一类？

33. 在对应的空白处标记一个**C**（CISC）或者是一个**R**（RISC），分别指出下面的内容是属于 CISC 还是 RISC？

_____① 平均每个时钟周期要执行的简单指令。
_____② 单一寄存器组。
_____③ 编译器中的复杂性问题。
_____④ 高度流水线作业。
_____⑤ 任何指令都可以引用存储器。
_____⑥ 指令被微程序解释。
_____⑦ 固定长度、容易译码的指令格式。
_____⑧ 高专用的、不经常使用的指令。
_____⑨ 使用重叠寄存器窗口。
_____⑩ 相对较少的寻址模式。

没有度量就没有管理。
——Peter Drucker
数字并不是总能描述事实。
——美国谚语

第 10 章　性能度量和分析

10.1　概述

本章开始所引的两条谚语凸现了计算机性能评估方面所面临的两难境遇。我们必须要有定量的工具来度量计算机的性能，但是我们又如何确定为某个任务所选择的度量工具是否满足评价的目标呢？事实上，人们不可能总是非常确定这类事情。此外，系统供应商常常有强烈的意愿去突出一些真实的数字，以便使自己品牌的计算机系统看起来优于他们的竞争者。

如果大家对计算机性能评价的基本知识有一个全面的了解，那么就不难抵御那些通过统计数字所做的宣传。本章中所介绍的基础知识，无论是对于选购新计算机，还是对于改进现有系统的性能都是非常有用的。

本章还会介绍影响处理器、程序和磁盘存储器性能的一些因素。本章的主要思想是考虑系统的调试。好的系统性能测量工具（通常由生产厂家提供）是保持系统运行在最佳状态的一个不可缺少的辅助手段。在学完这一章后，我们将知道应该在系统的调试报告中去寻找什么，以及其中的每个信息项怎样来构建系统的整体性能。

10.2　基本的计算机性能方程式

在前面章节中，我们已经多次看到了这个基本的计算机的性能方程式。这个等式是度量计算机性能的基础，它度量的是 CPU 的处理时间：

$$\frac{\text{时间}}{\text{程序}}=\frac{\text{时间}}{\text{周期}}\times\frac{\text{周期数}}{\text{指令}}\times\frac{\text{指令数}}{\text{程序}}$$

这里每个程序的时间就是所需的 CPU 时间。对这个方程进行分析不难看出 CPU 的优化过程可以对机器性能产生很大影响。我们已经讨论了利用这个方程式来提升系统性能的几种方式。RISC 结构的计算机采用的是尽力减少每条指令需要的时钟周期数，而 CISC 机器则是设法减少每个程序中的指令数。矢量处理器和并行处理器也是通过降低 CPU 时间来提升性能。其他一些改进 CPU 性能的方法将在本章后面的部分讨论。

当然，对 CPU 进行优化并不是增强系统性能的唯一途径。存储器和 I/O 对系统处理能力也有很大的影响。但是，存储器和 I/O 的影响不能够用这个方程式来说明。要增强系统的整体性能，可以有以下几种选择方案：

- CPU 优化（CPU optimization）——最大限度地提高 CPU 所执行的各种操作的速度和效率（性能方程式强调的就是这个优化过程）。
- 存储器优化（memory optimization）——最大幅度地提升代码的存储器管理的效率。
- I/O 优化（I/O optimization）——最大限度地增强输入/输出操作的效率。

应用程序的整体性受到上述三种因素的其中一种因素的限制，我们分别称为 CPU 约束（CPU bound）、存储器约束（memory bound）和 I/O 约束（I/O bound）。在本章中，我们在所有这三个层次上进行计算机系统的优化。

在研究优化技术之前，先来回忆一下 Amdahl 定律。这个定律制约着通过所有优化方法所能达到

的潜在的系统加速结果。Amdahl 定律指出计算机通过使用某种较快的执行模式可以获得的性能改善程度，受到这种快速模式所占用的系统时间的比例的限制：

$$s=\frac{1}{(1-f)+f/k}$$

其中，S 表示系统获得的加速率，f 代表由新的快速部件（或增强部件）所完成的工作部分，k 代表新部件（或增强部件）的加速率。

因此，人们意识到只有当使用最频繁的部件的性能得到改进时，系统性能才有可能实现重大改进。简单地说，在改善系统性能过程中，我们通过加速最常用的组件，就可以获得最大的收益。要改善系统的性能，首先要知道系统或程序是 CPU 约束，存储器约束，还是 I/O 约束。在讨论和进行系统性能改进时，读者应该要牢记这些观点。下面先开始讨论系统整体性能的各种度量方法，然后再介绍一些与单独的系统组件的性能相关的因素。在开始讨论这些话题之前，首先介绍一些理解计算机性能度量所必需的基本的数学概念。

10.3 数学预备知识

计算机性能评价是一门定量化的科学。一些数学和统计工具提供了许多方法来评估计算机系统的整体性能和计算机的各个组成部件的性能。事实上，因为有很多方法都可以用来对计算机系统的性能进行量化评测，所以选择一种正确的统计方法已经变成度量方法中的一种挑战。本节将描述最通用的“普通”计算机性能的度量方法，然后介绍每种方法的适用范围。本节第二部分介绍一些其他方法，这里由于错误推理会误用这些数量信息。在继续讨论之前，需要先给出一些定义。

系统性能的度量通常取决于个人观点。计算机用户最关心的是系统响应时间（response time）：系统完成一个任务需要多长时间？系统管理员最关心的是系统吞吐量（或称为处理能力，throughput）：在不会对反应时间产生较大不利影响的情况下能够执行多少个并发任务？显然，这两个观点是反向关联的。更明确地讲，如果一个系统在 k 秒内执行一个任务，则它执行这些任务的吞吐量就是每秒 $1/k$。

要比较两个系统的性能，我们可以测量在每个系统完成等量的任务时，各自所需要的时间。如果同一个的程序在两个不同的系统上运行，例如系统 A 和系统 B，那么如果满足下面的关系，就表示系统的运行速度 A 比系统 B 快 n 倍：

$$\frac{\text{在系统 } B \text{ 上的运行时间}}{\text{在系统 } A \text{ 上的运行时间}}=n$$

如果满足下面的关系，则表示系统的运行速度 A 比系统 B 快 $x\%$：

$$\left(\frac{\text{在系统 } B \text{ 上的运行时间}}{\text{在系统 } A \text{ 上的运行时间}}-1\right)$$

下面考虑两台赛车的性能。赛车 A 用 3 分钟完成 10 英里的路程，而赛车 B 则在 4 分钟内完成同样的 10 英里路程。现在使用上面的性能公式，可以知道赛车 A 的性能比赛车 B 快 1.25 倍：

$$\frac{\text{赛车 } B \text{ 行使 10 英里的时间}}{\text{赛车 } A \text{ 行使 10 英里的时间}}=\frac{4}{3}=1.25$$

同样可以得出，赛车 A 比赛车 B 快 25%：

$$\frac{\text{赛车 } B \text{ 行使 10 英里的时间}}{\text{赛车 } A \text{ 行使 10 英里的时间}}=\left(\frac{4}{3}-1\right)\times 100=25\%$$

这些公式在比较一个系统的平均性能和另一个系统的平均性能时非常有用。然而，最后得出的数字结果主要取决于“平均”的定义在实际的机器性能上所代表的含义。

10.3.1 平均数的意义

统计科学告诉我们，如果想得到有意义的信息，必须进行足够多的测试来证明基于这些测试结果所得出的推论是正确的。测试过程中的变化性越大，测试取样量也应该越多。在进行“足够多”次数的测试后，需要使用一种合理的方式对数据进行组合，或平均，以形成一种简明的“集中趋势的度量（measure of cen-

tral tendency)"。集中趋势的度量可以指出取样系统（对象总数，population）的预期行为。但是，并不是所有的数据平均的方法都是等价的。要选择的方法取决于数据本身的特点和测试结果的统计分布。

算术平均值

算术平均值是大家最熟悉的一种方式。如果有 5 个测量结果，将它们加在一起并除以 5，得到的结果就是算术平均值（arithmetic mean）。当大家看到一些计量的平均结果时（例如过去一年的汽油价格），通常是指按照一些给定时间间隔对价格进行取样后求得的平均值。

算术平均值方法不应该用于数据变化幅度较大，或者是数据朝着较低数值或较高数值发生倾斜的情况。下面，考虑表 10-1 所示的 3 台计算机的性能数值。这些数值是 5 个程序分别在 3 台计算机上的运行时间的结果。观察这些运行时间，可以看出这 3 台计算机的性能虽然是不同的。如果仅仅是使用运行时间的算术平均值来报告计算机的性能，那么性能不同这个事实将会完全被掩盖掉。

表 10-1 5 个程序分别在 3 个系统上运行的平均时间，时间单位为秒

程序	系统 A 执行时间	系统 B 执行时间	系统 C 执行时间
v	50	100	500
w	200	400	600
x	250	500	500
y	400	800	800
z	5000	4100	3500
平均值	1180	1180	1180

在恰当使用算术平均的方法时，人们又使用了加权算术平均（weighted arithmetic mean）来对算术平均进行改进，因为加权算术平均提供了一个有关系统的预期（expected）行为的清晰图像。如果我们可以得到一些有关在系统日常事务处理中这 5 个程序运行的频繁程度（频率）的话，那么就可以使用这 5 个程序执行的组合过程来计算每个系统的相对频繁性集。加权平均值是通过每个程序运行的频率及其运行时间的乘积来计算的，把加权运行时间进行平均就得到加权算术平均值。

表 10-2 中重新列出了表 10-1 中系统 A 和系统 C 的程序运行时间，再在运行时间表中补充每个程序的执行的频率。例如，在系统 A 上，在程序组合 v、w、x、y、z 的每 100 次执行中，程序 y 执行了 5 次。我们得出，在系统 A 上这 5 个程序执行时间的加权平均值为：

$50\times0.5+200\times0.3+250\times0.1+400\times0.05+5000\times0.05=380$

类似的计算指出，在系统 B 上 5 个程序执行时间的加权平均值是 695 秒。通过使用加权平均，我们可以清楚地看到在这种特定的任务环境下，系统 A 大约比系统 C 快 83%。

在使用加权平均时，最容易出错的方式是使用一些一成不变的假设。现在，假设某公司的计算机工作量的执行组合形式如图 10-2 所示。基于这些信息，公司愿意购置系统 A 而是不系统 C。假如有一个精明的用户 Wally，他断定程序 z 可以输出和运行程序 w 同样的结果。然后他把程序 z 的结果用作为程序 w 的输入。另外，因为程序 z 需要很长时间来运行，所以 Wally 也就有了很好的借口来喝杯咖啡休息一下。不久，Wally 的发现传遍了整个办公室，实际上 Wally 部门中的每个人都在采用他的思想。在这些日子里，系统 A 的工作量的分布图看起来就如表 10-3 所示。管理部门的人员肯定不知道全新的计算机系统为什么会突然表现出如此差的性能。

表 10-2 在两个系统上 5 个程序执行组合和运行时间的加权平均值

程序	执行频率	系统 A 执行时间	系统 B 执行时间
v	50%	50	500
w	30%	200	600
x	10%	250	500
y	5%	400	800
z	5%	5000	3500
加权平均		380 秒	695 秒

表 10-3 使用一个修改的执行组合后，系统 A 的运行时间的加权平均值

程序	执行时间	执行频率
v	50	25%
w	200	5%
x	250	10%
y	400	5%
z	5000	55%
加权平均		2817.5 秒

几何平均值

从前面的讨论中我们知道，如果测量中数据有比较大的变化性则不宜使用算术平均值。另外，除非我们对静态的和具有代表性的系统工作量有一个清楚的认识，否则使用加权算术平均值同样无效。几何平均值（geometric mean）则不管数据的分布如何都能给我们一个可以用来实行系统比较的一致的数字。

从形式上看，几何平均值定义为 n 个测量结果乘积的 n 次根。用公式表述如下：

$$G=(x_1\times x_2\times x_3\times\cdots\times x_n)^{\frac{1}{n}}$$

在比较两个系统的相对性能时，几何平均值比算术平均值更有用处。当我们相对于一个只是用作为参考的通用机器来说明这些系统时，系统之间性能结果就会变得很容易进行比较。当采用一个程序在参考机器上的运行时间与相同的程序在要评估的系统上的运行时间的比值时，我们说这些要评估的系统相对于参考机器是归一化的（normalized）。

为了求出归一化比值的几何平均值，我们先把这 n 个比值相乘后再取 n 次根。系统 A 和系统 C 相对于系统 B 的归一化的几何平均值计算如下：

系统 A 的几何平均值 $=(100/50\times400/200\times500/250\times800/400\times4100/5000)^{1/5}\approx1.6733$

系统 C 的几何平均值 $=(100/500\times400/500\times500/500\times800/800\times4100/3500)^{1/5}\approx0.5898$

计算的细节过程如表 10-4 所示。

表 10-4 几何平均值，由每个系统的归一化执行时间的乘积的 5 次根求出

程序	系统 A 执行时间	相对于 B 的归一化执行时间	系统 B 执行时间	相对于 B 的归一化执行时间	系统 C 执行时间	相对于 B 的归一化执行时间
v	50	2	100	1	500	0.2
w	200	2	400	1	600	0.6667
x	250	2	500	1	500	1
y	400	2	800	1	800	1
z	5000	6.82	4100	1	3500	1.1714
几何平均值		1.6733		1		0.5898

几何平均值的一个很好的特性是不管选取哪个系统作为参考机器，都会得到相同的结果。表 10-5 显示了用系统 C 作为参考机器时的结果。注意：不管选择哪个系统作为参考机器，几何平均值的比值都是一致的。

表 10-5 利用系统 C 作为参考系统所求得的几何平均值

程序	系统 A 执行时间	相对于 C 的归一化执行时间	系统 B 执行时间	相对于 C 的归一化执行时间	系统 C 执行时间	相对于 C 的归一化执行时间
v	50	10	100	5	500	1
w	200	3	400	1.5	600	1
x	250	2	500	1	500	1
y	400	2	800	1	800	1
z	5000	6.7	4100	0.8537	3500	1
几何平均值		3.4258		1.6497		1

$\dfrac{\text{系统 A 的几何平均值}}{\text{系统 B 的几何平均值}}\approx1.67$，而

$\dfrac{\text{系统 B 的几何平均值}}{\text{系统 C 的几何平均值}}\approx1.45$，而

$$\frac{\text{系统 A 的几何平均值}}{\text{系统 C 的几何平均值}} \approx 2.43$$

如果将系统 A 作为参考机器，也会得到相同的比值。

几何平均值证实了我们有关系统 A 和系统 C 相对性能的思考的直觉。通过求几何平均值的比率，不难发现系统 A 比系统 C 给出一个差得多的性能结果。然而，几何平均值并不是线性关系。虽然系统 A 与系统 C 的性能的几何平均值之比是 2.43，但是这不意味着系统 C 比系统 A 快 2.43 倍。从原始数据来看，这一事实是很明显的。因此，想购买系统 C 的人如果认为系统 C 在性能上是系统 A 的 2 倍，那他们肯定会大失所望。与加权算术平均值不同，几何平均值在明确表述计算机系统的实际行为的统计期望结果方面是绝对没有帮助的。

几何平均值的第二个问题在于，较小的数值对整体结果的影响不是成正比的。例如，如果系统 C 的制造商将测试组中最快（可能是最简单的）的程序性能提升 20%，即运行时间由 500 秒降至 400 秒，那么归一化的几何平均值将提升 4.5%。而如果将这一性能提升 40%（这样程序可以在 300 秒内完成），则归一化的几何平均值的增加超过 16%。无论将哪个程序的性能提升 20%或是 40%，我们发现相对几何平均值都同样会减少。可以预期要将一个大型复杂的程序的运行时间缩减 700 秒会比将一个较小的简单程序的运行时间缩减 200 秒要困难得多。从 Amdahl 定律可知，在实际的计算机世界中，最大、最耗时的程序将对系统性能产生最大的影响。

调和平均值

当我们把数据表示为速率时，例如每秒钟完成的操作数，无论是用几何平均值，还是用算术平均值来评价计算机的性能都是不合适的。为了计算平均速度或平均比率，应该使用调和平均值（harmonic mean）。利用调和平均值可以组成一个系统吞吐量的数学期望值，还可以比较系统或系统组件之间的相对吞吐量（处理能力）。调和平均值的求值方法是将各个数据的倒数相加，然后用这个求和结果去除取样的数据元素数目。数学公式表示为：

$$H = n \div (1/x_1 + 1/x_2 + 1/x_3 + \cdots + 1/x_n)$$

下面考虑一个简单的有关汽车旅行的例子，看一下调和平均值是如何应用于速率的。假设我们开车旅行 30 英里，前 10 英里的速度是每小时 30 英里，第二个 10 英里的速度是每小时 40 英里，最后 10 英里的速度是每小时 60 英里。如果对这些速率进行算术平均，那么得到的旅行平均速度是每小时 43.3 英里。但这显然是不正确的。实际上，通过第一个 10 英里所需的时间是 1/3 小时。通过第二个 10 英里的耗时为 1/4 小时，而最后 10 英里花费了 1/6 小时。旅行的总计时间为 3/4 小时，由此求出平均速度为 30 英里÷3/4 小时＝ 每小时 40 英里。调和平均值就可以非常简洁地给出正确的结果：

$$3 \div (1/30 + 1/40 + 1/60) = 40 \text{ 英里/小时}$$

在本例中为了对给定的距离求出相关的平均值，我们不得不非常仔细地在旅行的每一段行程中都行驶相同的距离。如果汽车以每小时 60 英里的速度行驶 100 英里（代替 10 英里）得到的调和平均值将会是相同的。调和平均值并不是告诉我们完成了多少工作，而仅仅是完成的这项工作的平均速率。

调和平均值与几何平均值相比有两个优点。首先，调和平均值是一个非常合适的计算机行为预测工具。因此，调和平均值的结果在性能比较的范围之外也很有用。其次，耗时较多的程序比耗时较少的程序对调和平均值的影响更大。这个事实不仅对“快速调整（quick fix）”优化过程有影响，也同样反应系统的真实情况。大而慢的任务比小而快的任务可能会消耗更多的机器周期。因此，加速大而慢的任务可以获得更大的机器性能的改善。

与几何平均值一样，调和平均值也可以采用相对性能的比值形式。但是，调和平均值对参考机器的选择更加敏感。换句话说，调和平均值的比值不像几何平均值那样，对于不同的参考机器都是一致的。然而，在利用几何平均值比较机器的性能之前，必须建立所谓“作业（work）”的定义。在本章的后面部分，读者将会看到这是一种非常含糊的思想。

刚才我们利用一个旅行的例子证明了算术平均值不适合用来描述平均速率问题。采用归一化表示

的算术平均值结果同样也是不正确的。表 10-6 总结了各种平均值方法的正确应用范围。

表 10-6 数据特性和所适用的平均值方法

平均值方法	均匀分布的数据	不对称分布式数据	表示为比率的数据	在共知工作量下的系统性能指示器	表示为比率的数据
算术平均	×			×	
加权算术				×	
几何平均		×	×		
调和平均				×	×

偶尔，我们会在一些最不希望看到的地方发现误用统计方法的情形。平均值方法的不正确应用只是在公正和客观地评价计算机系统性能方面所遇到的几个障碍中的一个。

10.3.2 统计学和语义学

人的本性驱动着我们尽可能以最生方式塑造我们自己和我们的信念。对于产品销售者来说，其动机完全是出于自身的利益和生存的需要，当产品被一大堆眼花缭乱的介绍和广告所包围时，人们很难看清楚产品的真实面目，即使这个产品是一个非常好的产品。知道一些所谓修辞学的逻辑概念的读者，都明白某些非常荒谬的推理能够应用在销售和广告上。一个经典的例子就是一个“在电视上扮演医生”的演员为某种令人烦恼的疾病患者治疗方法。在修辞学的逻辑中，这种情况被称为哗众取宠，或者说是求助于无资格的权威的谬误。一个演员，除非他还有医学学位，否则没有资格对任何疾病做出某种处理方案是合适的断言。虽然，演员“在电视上扮演计算机科学家”来推介计算机主机的情形并不常见，但是某些计算机供应商的销售广告可以说完全是一些投入较好的娱乐内容。

计算机购买者常常对计算机销售资料上引用的各种数字感到困惑。我们已经提到了平均数是如何被误用的。即使采用了正确的统计方法，对许多人来说这些数字还是不容易理解。计算机供应商通常会提供一些“定量”的信息数据，使得他们所宣称的先进系统蒙上一层可信的光环。在第 10.4 节中，我们讨论了一些客观的计算机性能的度量方法。声誉好的计算机供应商都会正确地引用这些度量结果。但是，即使是一种优秀的度量机制也会被误用。在本节的后面部分，我们将介绍三种非常流行的带有修辞色彩的谬论，这些谬论可能是读者在购买新计算机硬件或系统软件时曾经遇到的。

不完整信息

在 2002 年早期，一些主要的商业和销售杂志上都有一个整页的广告，广告的主要内容是，“我们已经为我们的产品进行一个测试，并公布了测试结果。X 公司却没有发表他们产品的相同测试的结果，因此，我们的产品速度更快”。我们真正知道的一切只不过是这个广告中引用的统计数字，有关这些产品的相对性能却未提及。

有时，不完整信息的欺骗性也可能采取这样的形式，即系统供应商所引用的信息只是一些比较好的测试结果，而不会提及在同一时间同一系统上得到的不太有利的结果。这种情况的一个例子就是采用“单一品质因素（single figure of merit）”。系统供应商关心的只是在市场上比其竞争产品更具有优势的单个品质因素。实际上，这些单个的度量标准并不能代表正在讨论的系统的实际工作量。不完整信息表现的另外一种方式是系统供应商只引用“最高”的性能参数，而忽略平均参数或者是更一般的情况。

模糊信息和不适当的度量方法

如果在评估系统的相对性能的内容中出现一些像“更多”、“更少”、“将近”、“基本上”、“几乎”和类似不精确的词语，通常应该立刻引起警惕。如果这些词语有一些适当的数据支持，那么它们的使用可能是合理的。然而，敏感的读者可能知道“差不多”可以意味着在产品 A 和产品 B 的性能之间“只不过”有 50%的区别。

一个最近的广告小册子就使用了一些不适当和不可比的度量方法所得到的一种不准确结果来宣传某个特定品牌的系统软件。广告传单上的内容大概说，“测试中，使用 Text X 运行了软件 A 和软件 B。我们有软件 B 运行 Test Y 的结果。结果证明，Test X 和 Test Y 几乎等价。由此，可以得出结论软件 A 更快一些。”到底 Test X 和 Test Y 有多少不等价呢？是否有可能 Test X 的设计是为了使软件 A 看起来更好呢？在这种情况下，读者看到的不仅是（可能收到好处）广告作者在比较苹果和桔子，而且他甚至说不出这些水果的总量到底有多少。

寻求流行

下面的这种谬误（欺骗性）是目前为止最为普遍的，通常也是最难抵御的，特别是在大量采购时（例如计算机采购委员会）。这种策略就是，“百分之 X 的美国财富杂志排名 500 强的大公司使用我们的产品。”这通常是一个难以抗拒的事实，它显示该公司构建很好，并且可能比较稳定和值得信赖。这些非定量的考虑在系统和软件选择方面的确是非常重要的因素。然而，正因为百分之 X 的美国财富杂志排名 500 强的公司使用这个产品，但是并不意味着这个产品也适合于你的公司。这是一件复杂得多的事情。

10.4 基准

性能基准是一个系统相对于另一个系统做出客观的性能评价的一门科学。基准对于评价通过升级计算机或升级计算机部件获得的性能改进同样是非常有用的。好的基准可以让读者看穿广告宣传和误用统计数字的伎俩。最后，好的基准也有助于读者辨别在合理的价格内可以提供最好性能的系统。

在做完仔细的采购之后，什么是“合理”价格的问题通常都是不言而喻的。然而，“好”的性能却是很难懂的，几十年来都很难有一个好的定义。但是，好的性能是“看了就知道”的事情，而差的性能则在问题得到纠正之前肯定会让你的生活痛苦不堪。简单地说，如果一个计算机系统能够使用尽可能少的占用时间（或时钟）来运行应用程序，那么该系统就达到了一个最佳性能。当然，同样的计算机系统没有必要也能够用最短的可能时间来运行其他人的应用程序。

如果有某种方法可以按照单一的性能参数，或度量标准（metric）来对计算机系统进行分类，那么购买计算机的用户就会变得轻松很多。这种方法的显著优点是很少或不懂计算机的客户也可以知道他所支付的金钱得到了多少价值。如果存在一种简单的度量标准，我们可以使用性价比（price-performance ratio）来指示哪个系统是最佳的购买选择。

例如，我们定义一个称为“zing”的虚构的全包含的度量标准。用 150 000 美元购买一个能够提供 150zing 系统会比用 125 000 美元购买一个提供 120zing 的系统要好一些。

第一个系统的性价比为：

$$\frac{150\ 000\ 美元}{150zings}=\frac{1\ 000\ 美元}{zing}$$

第二个系统的性价比为：

$$\frac{125\ 000\ 美元}{120zings}=\frac{1\ 042\ 美元}{zing}$$

现在的问题就变成了我们是否能够为了节省 25 000 美元而忍受 120 zing 的系统，还是购买“大的经济型”的系统。当然，我们知道要完全利用这个“大的经济型”系统的资源容量还需要有一段时间。

这种方法的困难在于目前还没有一种类似于“zing”的通用的计算机性能的度量，而且要求这种度量方法可以应用于各种工作负荷下的所有系统上。所以，如果我们寻找的是一个能够应对（I/O 约束）高事务处理的系统，例如飞机的订票系统，那么应该更关心 I/O 性能而不是 CPU 速度。类似地，如果系统的主要任务是计算强度很高的应用程序（CPU 约束），例如天气预报或计算机辅助绘图等，那么主要注意力应该放在 CPU 的处理能力上，而不是 I/O。

10.4.1 时钟速率、MIPS 和 FLOPS

CPU 的速度本身就是一个会误导人的度量标准。非常不幸的是，计算机商家经常使用这个度量标

准来吹嘘自己的系统肯定要比其他系统优越。在体系结构相同的系统中，如果一个 CPU 以 2 倍于另一个 CPU 的时钟速度运行，那它可能会有更好的 CPU 处理能力。但是，人们在比较不同商家提供的产品时，系统的体系结构可能是不一样的。否则，这些商家都可不能声称比对方更有竞争上的性能优势。

一个被广泛引用的，与时钟速率有关的度量标准是每秒钟执行的百万条指令数（millions of instructions per second，MIPS）。然而，许多人都以为 MIPS 实际代表的意思是“针对销售人员的毫无意义的性能指标（Meaningless Indicator of Performance for Salesmen）”。这个度量标准测量的是某个系统执行一个典型的浮点、整数和逻辑操作的组合的速率。需要再次强调的是，使用这种度量标准的最大问题是不同的计算机体系结构通常需要用不同的机器周期数目来执行一个给定的任务。MIPS 的度量标准并没有考虑到完成一个特定任务需要多少条指令数目。

在对 RISC 系统和 CISC 系统进行比较时，我们不难看到这种最鲜明的对比。假如让这两种系统都执行一个整数除法操作。CISC 系统可能只需要执行 20 条二进制机器指令就可以得出计算结果。而 RISC 系统则可能需要执行 60 条指令。如果这两个系统都在一秒钟内得出答案，那么 RISC 机器的 MIPS 的等级将会是 CISC 系统的三倍。这能够公正地说明 RISC 系统比 CISC 系统快三倍吗？当然不能，因为在这两种情况下我们都是在一秒钟时间内得到答案。

使用 FLOPS 的度量标准中也有类似的问题。FLOPS 表示每秒钟执行的浮点操作数（floating-point operations per second）。百万 FLOPS（megaflops），或称为 MFLOPS，此度量标准最初使用在描述超级计算机的处理能力上，但是现在个人计算机中也经常引用。FLOPS 的度量标准比 MIPS 的度量标准更加麻烦，因为人们在如何组成一个浮点数操作上还没有达成一致。在第 2 章中，我们曾经解释过计算机是如何通过一系列的部分乘积、算术移位和加法操作来完成乘法和除法运算。在每个这些原始的操作过程中，都是对浮点数进行操作的。因此，难道我们可以说只有计算一个中间的部分和才是一个浮点操作吗？如果这样，并且我们的度量标准是 FLOPS，那么将会损害使用高效算法的计算机供应商家。高效算法可以使用较少的步骤来达到同样的结果。如果要求得答案所耗费的时间与使用效率较低的算法所需的时间相同，那么这个更有效率的计算机系统的 FLOPS 速率会较低。如果我们不打算计算部分和的加法步骤，那么如何证明计算其他浮点加法操作是合理的呢？而且，有些计算机根本不使用浮点指令。早期 Cray 超级计算机和 IBM PC 机都是使用整数程序来模仿浮点操作。因为 FLOPS 的度量标准只考虑浮点操作，如果只是单独依赖于这种度量标准，则这些计算机系统将会变得毫无价值。但是无论如何，像 MIPS 一样，MFLOPS 是销售人员采用的一种非常流行的度量机制，原因是它听起来像是一个“硬”价值指标，并且它代表一种简单而直观的概念。

尽管存在各种缺点，但是时钟速度、MIPS 和 FLOPS 在比较由相同的计算机厂家所提供的一系列相似计算机的相对性能时是很有用的度量方法。因此，如果一个厂家提供从现有的几个 MIPS 的系统升级到 2 倍的 MIPS 的系统时，你所支付的金钱将会得到合理的性能上的提升。实际上，许多计算机制造厂商为此准备了他们自己的一组度量标准。有道德的销售人员不会用他们自己公司拥有的专用度量标准来评价其竞争者的系统。如果使用某个制造厂商的专用度量标准来描述其竞争对手的系统性能时，那些潜在的客户将不会知道，这个专用的度量标准是否特别设计去关注某个特定类型系统的强项，而忽略了系统的弱点。

很显然，任何一个依赖于特定系统的组织和指令系统的度量标准都可能忽视一些计算机购买者所要寻找的东西。计算机购买者需要有一些比较客观的方法来了解哪个系统能够以最低的价格为他们的工作提供最大的处理能力。

[redacted]

在 20 世纪 70 年代和 80 年代，两大计算机制造商，IBM 和数字设备公司（DEC）之间展开了激烈的竞争。虽然 DEC 公司不生产巨型机系统，但是它的最大型的计算机系统也适合于使用较小型的 IBM 系统的客户。

为了打开新品牌 VAX 11/780 的市场，DEC 工程师在 IBM 370/158 和他们的 VAX 上运行一些小

的综合基准程序。IBM 公司传统上将 370/158 作为“1 MIPS”的机器进行销售。因此，当基准程序在 VAX 11/780 机器上耗费相同的时间时，DEC 公司就开始将他们的系统作为一个具有竞争力的“1 MIPS”系统销售。

在商业上，VAX 11/780 是一个很成功的产品。这个系统非常流行以至于后来变成标准的 1 MIPS 系统。许多年来，VAX 11/780 曾经是大量基准测试的参考系统。这些基准测试的结果可以为任何被测试的系统得出一个“近似的 MIPS”等级。

毫无疑问，VAX 11/780 的计算能力和 IBM 370/158 的计算能力相当。但是，VAX 11/780 还需要经过严格的审查才能被冠以“1 MIPS”机器的概念。测试的结果表明，由于特殊的指令系统，VAX 11/780 这个机器运行基准程序时仅执行了大约 500 000 个机器指令。因此，事实上，这个所谓的“1 MIPS标准系统”根本就只是一个 0.5 MIPS 的系统。后来，DEC 公司通过指定的 VUP（VAX 机器的性能，vax unit of performance）来宣传其计算机系统，VUP 实际上指出系统与 VAX11/780 机器的相对速度。 ■

10.4.2 综合基准：Whetstone、Linpack 和 Dhrystone

很久以来，计算机研究人员一直在寻找一种单一的基准。这种基准可以独立于任意类型的计算机组成原理和体系结构来进行公平和可靠的性能比较。寻求一个理想的性能度量工具最早开始于 20 世纪 80 年代晚期。这个时期流行的观点是人们可以通过一个标准化的基准应用程序来独立地比较许多不同系统的性能。由此得出的结论就是，人们可以用第三代语言（例如 C 语言）编写一个程序，在不同的系统上编译和运行这个程序，然后测试在不同系统上每次运行该程序所需的时间。这种测试结果的执行时间作为所有被测试系统的一个单一的性能度量标准。源于这种方式的性能度量标准被称为综合基准（synthetic benchmarks），因为它们并不需要代表某些特定的工作量或应用程序。其中，三种比较知名的综合基准是 Whetstone、Linpack 和 Dhrystone 度量标准。

1976 年，英国国家物理实验室的 Harold J. Curnow 和 Brian A. Wichman Whetstone 发表了 *Whetstone* 基准程序。Whetstone 基准是强调浮点计算的，还有许多计算三角函数和指数函数的库子程序的调用。测试结果用每秒执行 1 000 条 Whetstone 指令（KWIPS）或每秒执行 1 兆条 Whetstone 指令（MWIPS）来表示。

另一种用于浮点性能测试是 Linpack 基准。Linpack 是线性代数程序包（LINear algebra PACKage）的缩写，这是一个被称为基本线性代数子程序（Basic Linear Algebra Subroutines，BLAS）的子程序的集合，它使用双精度算术来求解线性方程系统。1984 年，美国 Argonne 国家实验室的 Jack Dongarra、Jim Bunch、Cleve Moler 和 Pete Stewart 为了测试超级计算机的性能开发了 Linpack 基准。开始，Linpack 基准使用 FORTRAN 77 语言来编写，后来又使用 C 语言和 Java 语言。尽管存在一些严重的缺陷，但是 Linpack 的一个很好的方面就是为 FLOPS 设立了一种标准的度量方法。即使对于一个根本没有浮点运算电路的系统，如果它能够正确执行 Linpack 基准程序，那么它也可以获得 FLOPS 等级评定。

高速浮点计算肯定不会对每个用户都很重要。意识到这个问题之后，西门子尼克道夫（Siemens Nixdorf）信息系统公司的 Reinhold P. Weicker 在 1984 年编写了一个基准程序，这个基准程序主要关心的是字符串处理和整数操作。他把这个程序称为 Dhrystone 基准，据称是作为 Whetstone 基准的一个双关语。这个程序是 CPU 约束的，不需要执行 I/O 或系统调用。不像 WIPS，Dhrystone 基准的结果简单地用每秒的 Dhrystone 数来表示，在一秒内测试程序运行的次数，而不是用 DIPS 或 Mega-DIPS 来表示。

Whetstone、Linpack 和 Dhrystone 基准程序在算法和报告的结果方面都具有简单和易于理解的优点。不幸的是，这也正是它们的主要限制。因为这些程序的各种操作都有非常清晰的定义，所以对于编译器的编写者来说，可以很容易在他们的产品上装备带有“Whetstone”、“Linpack”或“Dhrys-

tone”的编辑开关。当具有这些设置时，这些编译器的选择项就会调用为基准程序编写的专用的被优化的程序代码。而且，被编译对象很小，以至于程序的绝大部分都可以驻留在现在的计算机系统的高速缓存中。这样一来，也就失去了评价一个系统的存储器的管理能力的机会。

运行基准程序是一种实践惯例，它和综合基准本身一样历史悠久。人们利用基准来设计各种编译器和计算机系统，就是为了获得最佳的系统功能。只要发布一个好的测试结果数据可以带来经济利益时，计算机制造商都会不遗余力地使他们的数字看起来非常好。如果测试数据比竞争者的结果更好，那么他们立刻就会做广告宣传他们的“优势”系统。这种广泛流行的做法已经变成了著名的基准方法(benchmarking)。当然，不管这些数据有多好，基准测试的结果实际上只能告诉我们一件事，即被测试的系统运行这个基准程序有多好。但是，这些数据绝不会告诉我们运行其他程序方面的情况，特别是我们所需的特定的工作量。

10.4.3 标准性能评估公司基准

计算机性能度量科学大大受益于 Whetstone、Linpack 和 Dhrystone 基准所做的贡献。一方面，这些程序给出了一种有价值的思想，即有一个通用的标准，所有系统都可以通过这个通用标准进行比较。更重要的是，虽然是无意识的，但这些基准证明了当一个精心设计的测量基准又小又简单时，对于计算机的制造厂商来说，优化他们的产品性能变得非常容易。很显然，解决这个问题的办法就是设计一种更加复杂的，但是同样能够提供一些易于理解的结果的基准。这就是 SPEC CPU 基准的目标。

标准性能评估公司（SPEC，Standard Performance Evaluation Corporation）成立于 1988 年，是由一个计算机制造商协会和电气工程时代杂志（Electrical Engineering Times）合作创建的。SPEC 的主要目标是为计算机的性能度量建立一些公正和实际的方法。今天，这个组织包括超过 60 家的会员公司和 3 家分支委员会。这些委员会是：

- 开放系统小组（OSG），主要从事有关工作站、文件服务器和桌面计算环境方面的工作。
- 高性能小组（HPG），主要关心企业级多处理器和超级计算机。
- 图形性能特性小组（GPC），主要致力于多媒体和高密度图形系统。

这些组织同计算机用户一起工作，确定一些能够代表典型工作量的程序，或者是能够从许多系统中区分出一个优越系统的应用程序。如果将 I/O 的例行程序和其他非 CPU 密集型的程序代码从这些应用程序中剔除出来，那么最后得到的结果程序称为内核（kernel）。SPEC 委员会仔细地从不同的应用团体提交的程序中挑选出内核程序。最后选择的内核程序集合称为基准套件（benchmark suite）。

SPEC 基准中最著名的（和受人尊敬的）是 SPEC 的 CPU 套件，它可以用来测试 CPU 的吞吐量，高速缓存和存储器的访问速度，以及编译器的效率。这个基准的最新版本是 CPU2000，CPU2004 的版本现在正在开发中。CPU2000 包括两个部分：CINT 2000 和 CFP2000。CINT 2000 用来评测系统执行整数处理的能力，而 CFP2000 用来评测 CPU 的浮点运算性能（参见表 10-7）。CINT 和 CFP 中的“C”代表“组件（component）”，这个名称强调了这些基准仅仅测试系统中的一个组件。

CINT 2000 由 12 个应用程序组成。其中，11 个应用程序是用 C 语言编写的，另外 1 个是用 C++ 语言编写的。CFP2000 套件由 14 个应用程序组成，其中的 6 个采用 FORTRAN 77 编写，4 个用 FORTRAN 90 编写，另外 4 个用 C 语言编写。这些结果（系统吞吐量）是通过运行这些测试程序得到的。这些结果表示为在被测系统上运行这个内核所需时间与在参考机器上运行这个相同的内核所需时间的比率。对于 CPU2000 来说，参考机器是一台带一个 300MHz CPU 的 Sun 公司的 Ultra 10 计算机系统。被测试的系统几乎可以确定要比 Sun Ultra 10 系统快，所以可以看到计算机制造厂家所引用的测试结果是一些很大的正数。这个数越大，性能就越好。在大多数系统上完成整个 SPEC CPU 2000 套件的运行仅需要两天多一点的时间。运行 CINT2000 和 CFP2000 的报告结果是所有组件内核运行结果的比率的一个几何平均值（细节请参考补充的“计算 SPEC CPU 基准”）。

表 10-7 SPEC CPU 2000 基准套件的组成内核

SPEC CINT2000 基准内核程序				
基 准	参考时间	语言	应用类别	简单说明
164. gzip	1400	C	压缩	压缩一个 TIFF（标签图像格式文件），Web 服务器日志，二进制程序代码、"随机"数据和 tar 文件源程序
175. vpr	1400	C	现场可编程门阵列电路的布局和布线	使用一个组合优化程序来表射（绘图）FPGA 电路的逻辑模块和需要的电路连接，这些程序一般是集成电路的 CAD 程序
176. gcc	1100	C	C 编程语言编译器	利用 gcc 编译来自 5 个不同的输入源文件的 Motorola 88100 的机器代码
181. mcf	1800	C	组合优化	求解一个通常在公共交通或划领域遇到的单一总站的汽车调度问题
186. crafty	1000	C	国际象棋	对于可变的搜索域"深度"求解 3 种不同的棋盘输入布局图的可能的下一步走法
197. parser	1800	C	文字处理	使用一个 40 000 字的字典解析输入的句子找出英语的句法
252. eon	1300	C++	计算机虚拟化	使用概率光线寻迹方法寻找三维光线的交汇点
253. perlbmk	1800	C	PERL 编程语言	处理 5 种 Perl 脚本，生成邮件、HTML 和其他输出
254. gap	1100	C	群理论解释程序	解释一种编写的群理论语言，以处理一些组合逻辑问题
255. vortex	1900	C	面向对象的数据库	从 3 个面向对象的数据库中操作数据
256. bzip2	1500	C	压缩	压缩一个 TIFF、二进制程序和 tar 类型的源文件
300. twolf	3000	C	布局和布线仿真器	在一个微芯片上寻找一种最佳晶体管布局问题的近似解决方案
168. wupwise	1600	FORTRAN 77	量子色动力学	物理学家在研究量子色动力学中模拟夸克（quark）相互作用
171. swim	3100	FORTRAN 77	浅水建模	使用数学建模技术预测天气，游动（swim）通常用作超级计算机性能的一个基准
172. mgrid	1800	FORTRAN 77	三维势场的求解程序	计算三维标量伯松方程的求解问题，这个内核基准来自 NASA
173. applu	2100	FORTRAN 77	振物线-椭圆型偏微分方程	使用稀疏程序可比矩阵求解 5 个非线性的偏微分方程
177. mesa	1400	C	3-D 图形库	把三维图形输入转换成三维图形输出
178. galgel	2900	FORTRAN 90	计流体力学	决定引起对流流动转变为振动流动时流体容器壁上温度差的特征值
179. art	2600	C	图像识别	在一幅图像中找出一个直升机和飞机的图像，这种算法使用神经网络
183. equake	1300	C	地震波的传播仿真	使用有限元分析法从一次地震事件的结果来恢复地表运动的历史
187. facerec	1900	FORTRAN 90	面部识别	使用"弹性图形匹配"方法来识别由加标示图形描述的面部
188. ammp	2200	C	计算化学	通过计算一个系统中的分子运动来求解分子动力学问题
189. lucas	2000	FORTRAN 90	原始状态测试	开始一个确定某个大 Mersenne 数字（2^{p-1}）的初始状态的过程，这个结果找不到，因而应先测量中间结果
191. fma3d	2100	FORTRAN 90	有限元撞击仿真	仿真非弹性的三维固体碰撞效果
200. sixtrack	1100	FORTRAN 77	高能核物理加速器设计	仿真通过一个粒子加速器的粒子轨迹行为
301. apsi	2600	FORTRAN 77	污染物分布	利用初始速度、风速和温度等参数来求解离开一个真定源的污染物粒子的速度

由于系统销售非常依赖于有利的基准结果，所以人们不难想到计算机制造厂商会竭尽所能寻找各种欺骗应对 SPEC 规则的方法来运行这些基准程序。第一种伎俩就是使用编译器的“基准开关”，这种方法在 Whetstone、Linpack 和 Dhrystone 基准程序上已经变成惯例。然而，为了使用 SPEC 套件，要找到一组完美的编译器选择并不像在早期的综合基准中那样简单。为了优化套件中的每个内核，通常需要有不同的设置，并且要找到这些设置方法是一件非常耗时和沉闷的工作。

在发布 CPU95 套件之前，SPEC 开始意识到这种使用“基准专用的”编译器选择的情况。SPEC 尝试要终结这种基准专用编译器，具体做法是强制要求套件中所有使用相同语言编写的程序都必须使用具有相同组的编译器标志来进行编译。这种做法立即招致制造厂商方面的批评，他们争辩说客户有权知道一个系统可能达到的最佳性能。而且，如果客户的应用程序和其中的一个内核程序相似的话，那么客户会由于知道最佳的编译器选择而从中大大受益。

这些批评足以驱使 SPEC 允许套件中的每个程序都使用不同的编译器标志。然而，为了对大家都公平起见，计算机制造厂商们会为 SPEC CPU 2000 公布两组测试结果。其中一组结果是对于整个基准套件测试所有的编译器设置中有哪些相同的地方（称为基础度量标准），第二组结果是给出通过优化的设置所获得的结果（即最高度量标准）。在基准编译过程中的两种结果数字都需要进行报告，并且在每次运行时要完全公开这些编译器设置。

SPEC 基准程序的用户需要为基准套件的源代码和指令的安装和编译支付一定的管理费用。而且，SPEC 还鼓励制造厂家（不是必需的）提交一份包括基准测试的结果报告，供 SPEC 进行评估检查。如果这些测试是按照 SPEC 的指导方针运行，并且 SPEC 对这些测试结果表示满意，那么 SPEC 会在其网站上发布这些基准测试的结果并公开这些配置报告。SPEC 会十分仔细地确保计算机制造厂商已经正确地使用这些基准软件，并且保证完全公开这些系统的配置情况。

虽然处理器制造厂商之间的激烈竞争引发了大量的计算机商业新闻报道，但是 Amdahl 规律告诉我们，有用的计算机系统所需要的不仅仅是一个快速的 CPU。计算机的购买者感兴趣的往往是整个系统在它们自己特定的任务负载下的工作性能。针对这种情况，SPEC 还创建了一系列其他度量标准，包括 Web 服务器使用的 SPEC Web，高性能计算机的使用 SPEC HPC，以及测试客户端 Java 性能的 SPEC JVM。SPEC JVM 是测试 Java 服务器性能时使用的 SPEC JBB（Java Business Benchmark，Java 商业基准）的补充。SPEC 的每一个基准都坚持了 SPEC 要建立公正和客观的系统性能度量的思想。 ■

计算 SPEC CPU 基准

在本书中我们曾经讲到，计算 SPEC 基准结果的第一步是将一台基准内核在一台参考机器上运行所需的时间归一化到相同的内核在被测试系统上运行所花费的时间上。这种归一化的结果是一个简单的比值，然后把这个比值乘以 100。例如，如果有一个假想的系统在 476 秒时间内完成了 164.gzip 内核运行，而参考机器运行这个相同的程序所花费的时间是 1400 秒。那么归一化的时间比率为：1400÷476×100 = 294（截取整数）。最后的 SPECint 的整数性能的结果是在整个整数程序套件中所有的归一化比率的几何平均值。计算结果如表 10-8 所示。

表 10-8 一组假想的 SPECint 结果集合

基 准	参考时间	运行时间	参考时间的比率
164.gzip	1400	476	294
175.vpr	1400	437	320
176.gcc	1100	342	322
181.mcf	1800	687	262
186.crafty	1000	325	308
197.parser	1800	759	237
252.eon	1300	461	282
253.perlbmk	1800	823	219
254.gap	1100	449	245
255.vortex	1900	686	277
256.bzip2	1500	472	318
300.twolf	3000	516	581

要决定这个几何平均值，必须首先求出所有 12 个归一化基准时间的乘积：

$$294\times320\times322\times262\times308\times237\times282\times219\times245\times277\times318\times581 \approx 4.48\times10^{29}$$

然后计算这个乘积的 12 次方根：

$$(4.48\times10^{29})^{1/21} \approx 296$$

因此，这个系统（有一种公平的感觉）的 CINT 度量标准的结果为 296。如果这个结果是在运行利用标准的（也是比较保守的）编译器设置所编译过的基准程序时得到的，那么这个结果将会作为一个“基础”的度量标准 SPECint _ base _ 2000 来报告。否则，这个结果就是这个系统的 SPECint 2000 评价的结果。

当每次只运行每个基准程序的一个版本映像时，CINT2000 套件和 CFP2000 套件测量的是 CPU 的性能。但是，这个单线程模型不能告诉我们系统处理并发进程的能力好坏。SPEC CPU“速率（rate)”的度量方法在这里会给我们一些启示。计算 SPECint _ rate 的度量方法比计算单线程 SPECint 度量方法要稍微复杂一些。

为了找到一种速率的度量机制，可以在主机系统中启动若干个完全相同的基准内核程序进程。针对上面的例子，我们启动 4 个并发的 164. gzip 进程。在所有 164. gzip 程序实例终止后，通过把最后一个实例的完成时间减去第一个实例开始的时间，得到这 4 个并发进程所消耗的时间。假设该时间差为 450 秒。用这个时间去除 3600 秒得到“每小时的运行数”的比率，结果就是：

$$\frac{3600}{450}=8$$

接下来考虑套件中最大的参考时间，即浮点套件中的 171. swim 基准程序，把它归一化到基准程序的参考时间。164. gzip 的归一化数字大约为：

$$\frac{1400}{3100}\approx 0.45161$$

把这两个数字与运行进程的副本数目相乘就得到这个基准程序的 SPECint _ rate2000 度量标准。因为这里运行了 4 个基准程序的副本，所以有（截取到一位小数）：

$$4\times 8\times 0.45161=14.5$$

这里要报告的该系统的 SPECint _ rate2000 度量标准是所有 CINT2000 组件的内核的几何平均值。当然，可以利用相同的过程来确定 SPECfp _ rate 的结果。 ■

10.4.4 事务性能委员会基准

SPEC 基准对那些主要关心 CPU 性能的计算机购买者来说非常有用。但是，他们对于购买企业级事务处理服务器的用户来说受益却没有那么大。对于这种类型的系统来说，购买者最感兴趣的是服务器处理大量并发的持续时间较短的事务的能力。这里，每个事务都在一定程度上涉及到通信和磁盘 I/O。

对于商务活动来说，缓慢的事务处理系统的成本是非常高昂的，而且会引发大量的深层次问题。如果客户服务系统的处理速度很慢，会引起客户的不满，因而也会影响到这个零售商店的形象。在结账柜台前排成长龙缓慢移动的顾客们并不会关心信用卡的认证系统是否出现了麻烦。如果等得太久的话，昏睡的自动取款机和反应迟钝的收款机会陆续地赶走顾客。事务处理系统不仅仅对客户服务部门非常重要，而且就是企业生命所必须的血液。精明的商业领导人都愿意在这方面投入少量的资金以使客户满意。因为存在太多的问题，所以非常有必要找到某些方法来客观地评价支撑这些关键商业活动的处理系统的整体性能。

通常，计算机的制造厂商都有测试自己的系统性能的方法。这些测量不是为公众消费者设计的，而是为工程师内部使用而设计的，工程师使用这些方法来改进系统（系统的某些部分）的性能。在 20 世纪的 80 年代早期，IBM 公司就开发了一个这样的基准来帮助设计公司的大型机系统。这个基准，被称为 TP1（TP 即事务管理），最终还是进入了公用的领域。许多竞争的系统供应厂家开始使用这个基准并发布他们（令人惊异）的结果。事务处理专家对这种做法表示不满，因为这个基准并不是设计用来模仿真实的事务处理环境。一方面，这个基准忽略了网络延迟情况和用户“思考时间”的可变性。另一方面，TP1 能做的一切只是测量在理想条件下的最大处理能力（吞吐量）。虽然这些测量对于系统设计师来说很有用，但是它们却不能给计算机的购买者们太多的忠告。

1985 年，一个著名的 TP1 的批评者 Jim Gray，和一大群 TP1 的反对者共同提出了一个基准，这

个基准主要针对的是 TP1 的一些缺点。他们把这个基准称为 DebitCredit，用以强调它主要用来评价商业事务的处理性能。除了详细说明这个基准工作原理外，Gray 和他的团队还提议系统的测试结果应该随同被测试系统配置的总价格一起报告。他们还提供了一些方法可以按比例地缩放这种基准程序，这样使得这些测试对各种不同大小的系统来说显得比较公平。

DebitCredit 基准深受系统制造厂商们的欢迎，因为它提供了一种清楚和客观的性能度量方法。不久，大多数系统制造厂商都开始使用这个基准，并纷纷发布他们度量的优良结果。不幸的是，没有一种正式的机制来证明或反驳这些系统制造厂商公布的结果。实质上，制造厂商可以随意引用那些在他们看来比竞争对手更有优势的结果。显然，在这个领域，迫切需要有一些独立的评估和控制手段。为此目的，在 1988 年，Omri Serlin 成功地说服了 8 家计算机制造厂商联合起来组建了一个独立的事务处理委员会（Transaction Processing Council，TPC）。现在，TPC 大约由 40 家公司组成，其中包括系统软件和硬件制造商。

摆在 TPC 面前的第一个任务是发布一个正式的具有官方权威性质的基准套件。这个基准于 1990 年发布，称为 TPC-A。为了与技术革新和基准测试科学的进步保持同步，TCP-A 现在变成了经过三次重大修订后的第 5 个版本，TPC-C Version 5。

TPC-C 基准测试套件可以对一个批发产品销售公司的活动进行建模。这个套件是一种受控的 5 种事务类型的混合，这 5 种事务类型都是典型的订单执行系统。这些事务包括新订单开始，库存等级查询，订单状态查询，商品交付记录和货款支付处理。这些事务中资源最密集的是新订单开始的事务，它至少构成了这种事务组合的 45%以上工作量。

TPC-C 采用远程终端仿真软件来模仿一个用户和系统之间的交互过程。每次交互活动都是通过一个格式化的数据输入屏来进行的，这个数据屏的设计非常便于数据输入员的使用。仿真器程序从菜单中选择一项事务，就像是一个真实的人将要进行的操作一样。但是，这种选择从统计的角度来看是随机的，这样系统就可以执行正确的事务组合。诸如客户姓名和部件编号这样的输入值也同样是随机挑选的，这样可以避免出现重复的高速缓存命中相同的数据值，而引起频繁的 I/O 操作。

TPC-C 的终端到终端的响应时间是从“用户”完成所要求的输入的时刻起到系统在特定的终端上显示出必要的反应的时刻止的这段时间。在最新的 TPC-C 规则下，除了库存等级查询外，90%的事务都必须准确地在 5 秒钟时间内完成。库存查询被排除在 5 秒规则之外，是因为库存等级可能会在一个单次的查询事务中需要检查多个不同的仓库。这个事务处理和其他事务相比，需要更多的 I/O 操作。

要记住的是，TPC-C 套件使用真实的系统来模仿真实的商业行为。因此，每个更新的事务都必须支持一个产品数据库的 ACID 特性。这些特性已经在第 8 章中进行了描述。TPC-C 的 ACID 特性包括加锁和解锁，以及由对一个数据库日志文件进行日志更新所提供的回滚（或称为重新运行，rollback）能力。

TPC-C 度量标准是每分钟内完成的新订单事务的数目（tmpC），虽然其他各种事务的组合并发地执行在同一系统上。TPC-C 的结果报告中还包含一个性价比，这个比值是用系统的成本价格除以吞吐量度量的结果。因此，如果一个价格为 90 000 美元的系统提供了 15 000 tpmC 的吞吐量，那么系统的性价比为 6 美元/tpmC。系统的价格包括执行 TPC-C 事务所必须的硬件、软件和网络组件的价格。测试中所使用的各个组件都必须可以在结果报告的时间对公众销售。这个规则是为了防止厂商使用一些基准专用的组件。在生成基准测试结果报告中所涉及到的所有系统组件都必须非常详细地在一份完全公开的报告中列出，并提交给 TPC。公开的配置说明也必须包括所有在测试过程中有影响的调整参数（例如，编译器的开关）[⊖]。这个完全公开的报告还要包括“机器所有者的总费用”的数字，其中要把维护和支持整个系统三年所需的费用考虑进去。

⊖ 对于系统管理员来说，完全公开报告中所提供的调整信息，与 TPC-C 报告中所涉及的各种信息一样，都是非常有价值的资源。这些资源可以用来帮助寻找优化系统性能的方法。因为一个实际的工作负载不等同于一个 TPC-C 套件中设计的工作负载，所以报告中的调整信息也许不能给出一个最佳的结果，但它通常是寻找优化方法的一个好的起点。

当系统生产厂商把 TPC-C 结果提交给 TPC 时，会有一家独立的审计公司对报告中的所有信息进行审计，以确保报告的完整性和准确性。当然，审计人员不可能重新运行这种测试，因此这些吞吐量的数字通常都是一些表面数值，条件是这种测试被正确地执行。一旦 TPC 接受了报告，就会把报告公布在它的网站上供客户和竞争对手检查。有时，某个竞争对手或者 TPC 自己会质疑某个制造厂商的报告。遇到这种情况时，制造厂商要么撤回报告，要么为报告辩护。有时一个报告会悄悄地撤下来，原因是为结果进行答辩的代价实在是无法承受，即使这个制造厂商的要求很有根据。有时，系统供应商也可能自己选择撤回报告结果，因为测试中的配置已经不再可用，或者是系统的下一个模型已经比这个旧的模型有重大的改进。

TPC-C 仅仅是事务处理委员会所推出的多个基准中之一。在 TPC 成立时，商业计算领域主要由各种事务处理系统组成，这些系统同样支持财务处理，例如簿记和工资单。这些系统都必须有一组清晰定义的输入和产生一个清晰定义的输出，通常都是采用打印报告和表格的形式输出。这种确定性的模型缺乏在当今商业环境中所要求的灵活性，即难以提供深层次的数据分析工具。许多公司都用决策支持（decision support）工具来取代静态报告。这些应用程序可以访问大量的输入数据来提供各种商业情报信息，为市场导向和商业活动服务。从某种意义上来说，决策支持应用程序与事务处理应用程序是完全相反的类型。因此，它们需要使用不同类型的计算机系统。事务处理环境处理的是大量的持续时间较短的过程，而决策支持系统处理的是一些数量少持续时间较长的过程。

决策支持系统不会对在线查询给出即时的结果。但是，不管最后的结果如何有用，人们愿意等待的时间是有限的。事实上，如果决策支持系统的速度“太慢”，企业管理人员都不会很情愿地使用它，因此这也就脱离了它原来的目的。即使在人们都愿意为得到答案而等待“一会儿”的情况下，性能仍旧是一个大问题。

为了应对这个相对较新的计算领域，TPC 又创建了两个基准，TPC-H 和 TPC-R，用来描述决策支持系统的性能。虽然这两个基准都是直接针对决策支持系统的，但是在预先知道系统的报告参数时，TPC-R 基准程序可以度量性能，即可以利用 TPC-R 基准程序为这类报告建立数据库索引和对数据库进行优化。TPC-H 基准程序度量一个系统可以产生特别（ad hoc）查询结果的能力，在这些查询中查询参数都不是预先知道的。TPC-H 测试结果采用每小时的查询数目来表示，即 *QphH*。而在 TPC-R 中的测试结果则表示为 QphR。TPC 按照查询时所运行的数据库的大小来对这些测试结果进行分类，因为在一个 100G 字节的数据中执行查询和在一个 1T 字节的数据中执行查询是两个完全不同的任务。

为了保持与时代同步，TPC 同样也定义了度量 Web 服务器性能一个基准，TPC-W。这个套件的操作与 TPC-C 基准程序在思想上非常相似。通过“远程浏览器仿真器”输入一些随机挑选的事务，访问一组提交给客户的选择项目。这些顾客的选择是零售电子商务网站上的典型选择项目：浏览物品的分类目录，查看购物车，以及使用一个安全的连接来购买购物车中的物品。TPC-W 度量单位是每秒钟 Web 上的交互次数（Web interactions per second），或者简称为 WIPS[⊖]。TPC-W 使用两种混合的选择操作模式。这两种模式都是通过分析从实际的电子商务网站上搜集的日志，利用经验来确定的。其中一种选择操作的组合 WIPSb，主要是针对访问大多数情况下只是浏览而不是购买商品的 Web 站点的。而另外一种 WIPSo，是为那些客户大多数情况下是购买（或订购）商品而不是浏览的 Web 站点服务的。同 TPC-C 一样，TPC-W 的事务必须保持正确的数据库事务的 ACID 特性。

与 TPC-C 一样，TPC-H、TPC-R 和 TPC-W 的结果都同样要受到严格的审计控制。因此，在选择满足企业计算服务需求的系统时，这些度量方法都是一种可信赖的帮助工具。在使用 TPC 基准的过程中遇到的主要问题是，人们总是假设这些基准能够准确地预测在某人的特定的工作任务下的系统性能。但这并不是 TPC 的主张或打算。研究表明一组特定的实际工作负载在许多方面都与 TPC 的工作负载不同（参见下面的补充内容）。对于计算机基准来说，在使用正确时是一个不可缺少的工具。但是如果

⊖ 这里的 WIPS 不要与 Whetstone Instructions per Second 相混淆。我们已经用完了唯一的 4 个单词的首字母缩写吗？

使用不正确，它会带来风险。

TPC 基准：一个真实的效验

事务处理委员会已经做了各种努力，试图用其 TPC-C、TPC-H、TPC-R 基准来模拟真实世界情况。要确保在这些基准中包含一个普通的商业事务和活动的真实组合是一件非常困难的事情。随机选取的这些事务活动要产生尽可能多的 I/O 操作。特别是数据应该更频繁地从磁盘中提取，而不是从高速缓存或者是其他快速存储器中取出。

这其中所包含的思想是这种测试不应该偏向于某种特殊类型的计算机体系结构。如果数据不是随机产生的，那么基准将会有利于那些有大量高速缓存存储器的计算机系统，但是在真实的环境中却可能很难（成比例地）实现。

很多年来，这个想法一直没有引起人们的质疑。直到 IBM 公司一名攻读博士学位的学生 Windsor W. Hsu 从事了一系列的经验研究后，情况就发生了改变。在 IBM 公司的 Almaden 研究中心的赞助下，Hsu 和他的研究伙伴，在 IBM 公司的 10 个最大客户的系统上监视了几百万个处理事务。Hsu 的工作验证 TPC 基准的很多方面的内容，包括工作负载的各种组合情况。但是，Hsu 发现真实世界的活动在两个重要的方面与 TPC 模型不同。

第一，TPC 基准展示的是一个持续和稳定的事务比率。这是基准专门设计的，是为了让系统可以在最高工作负载比率的情况下完全应对各种事务处理。但是，Hsu 发现实际的工作负载是突发性的。一组事务发生后，到下一组活动发生之前将会有一个事务暂停的时段。对于系统设计者来说，这意味着如果在系统软件和硬件中增加有效的动态资源分配设施，将会改进系统的整体性能。虽然许多系统制造厂商事实上都使用了动态资源分配，但是他们的这些努力并没有得到 TPC 基准的认可。

Hsu 的第二个主要结果对 TPC 基准中随机选取数据和工作负载的观点提出了严重的挑战。他发现真实系统表现出比 TPC 程序要大得多的“伪顺序性（pseudo-sequentiality）”。这个发现是非常重要的，因为许多系统都会从磁盘和存储器中预取数据。如果使用预取数据的操作，实际的工作负载将会大大受益。更进一步来说，数据访问模式的伪顺序性使得数据自己会很好地适应最近最少使用（LRU）的高速缓存和存储器的页面替换策略。而 TPC 基准却不会这样做。

Hsu 的工作并不是反对 TPC 基准。相反，它对人们可以从基准结果中推断出特定的真实世界的情况的愚蠢想法进行了夸大。虽然 TPC 基准不能对这种真实世界和一些类似的情形建模，但是它在比较不同系统的体系结构的性能时仍然可以继续担当一种诚实、公平和可靠的准绳。 ■

10.4.5 系统仿真

TPC 基准与 SPEC 基准不同，TPC 基准致力于模仿完整的计算环境。虽然 TPC 基准的主要目的是度量性能，但是他们模仿的环境可能对预测在不同的条件下的系统性能，进而优化系统的性能也同样有用。一般来说，仿真方法为我们提供了一些工具，利用这些工具可以对系统行为的各个方面进行建模和预测，而无需使用仿真器要模拟的完全真实的环境。

仿真对于估计尚未存在的系统或系统配置的性能非常有用。聪明的系统设计师常常会在构建商业版本的产品之前，对新硬件或软件系统进行仿真研究。1967 年，斯坦福（Stanford）大学的博士研究生 Norman R. Nielson 在一篇名为“The Analysis of General Purpose Computer Time-Sharing Systems”的论文中充分展示了这种仿真方法的智慧之处。Nielson 的论文总结了他在当时还没有发布的 IBM 360/67 分时共享计算机系统（TSS）上的仿真研究及结果。利用 IBM 公司发表的规格说明书，Nielson 的研究工作揭示了 360/67 分时系统存在的一些严重缺陷。Nielson 的发现促使 IBM 在广泛发布这个系统之前，改进了原来的系统设计。

仿真是完整系统的各个特定方面的模型。这些模型可以让系统设计师在一个与真实系统分离的可控环境中随意实现各种“如果就怎样”的假设性测试。例如，假设我们现在对一个系统可以支持的最大化的并发任务数目感兴趣。调整的参数包括每个任务的存储器空间分配和它们占用 CPU 的时间片断

的长度。基于我们所了解的每个任务特性，在仿真过程中可以调整各种参数值直到找到一个最佳的平衡分布为止。在有真实任务和真实用户的实际环境中进行这样的修补工作将会引发一些真正的风险，最糟糕的情况是可能没有人能够完成任何工作。

系统仿真遇到的最大挑战之一是如何确定系统工作量的特性。系统工作负荷量的各种组合形式应该与被模拟的系统组件相对应。确定工作量的一种方法是从检查系统的日志开始推演出一个综合的（统计的）系统工作量的分布图。这是 TPC 在为它的 TPC-W 基准生成测试工作量组合时所采用的一种办法。

如果仿真器只把注意力放在系统的某个组件上，那么这样捕获到的整个系统或整个工作量的行为将不会产生足够的数据粒度（或间隔大小，granularity）。例如，假设系统设计师正在设法确定一个理想的组关联的存储器 Cache 配置结构。这个配置包括一级和二级高速缓存的大小，以及为每个高速缓存块所设置的组的大小。对于这种类型的仿真，仿真器需要详细的存储器访问数据。这种信息通常是通过对系统日志进行跟踪得出的。

系统跟踪（system traces）可以通过使用硬件或软件深入探测感兴趣的组件的各种活动来搜集详细的系统行为信息。探测工作会跟踪每个系统组件的实际行为的细节，可能包括二进制指令和存储器的引用。探测所搜集到的跟踪信息仅仅由几秒钟的系统活动组成，因为系统数据集的输出实在是太大了。为了产生一个在统计上有意义上的模型，一般需要进行多个跟踪探测。

在设计仿真器时，必须对仿真器的目的做一个清晰的定义。这里，人们需要有好的工程判断力来从一大堆不重要的系统特性中区分出重要的特性。如果模型过于详细，则模型运行的代价会很高，而且编写模型会非常耗时。相反，如果模型太简单而忽略一些关键因素，那么仿真器会产生错误的结果令人误导。系统仿真是一种出色的工具，但是就像其他任何工具一样，我们必须保证它适合于特定的任务。我们还要对系统仿真器进行有效性验证，以确保构建模型所做的各种假设的正确性。当然，最简单的模型是最容易验证的。

10.5 CPU 性能优化

长期以来，CPU 的性能一直是系统优化工作的重点。现在，没有一种单一的方法来增强 CPU 的性能，因为 CPU 的处理能力受到许多因素的影响。例如，程序代码会影响指令计数，编译器会影响指令计数和每条指令的平均时钟周期数，指令系统决定了指令计数和每条指令的平均时钟周期数，实际的硬件组织结构设定了每条指令的时钟周期数和时钟周期时间。

一些潜在的 CPU 优化技术包括集成的浮点单元、并行执行单元、专用指令、指令流水线、分支预测和代码优化。因为除了最后两项外，其他内容都已经在前面章节中做了介绍，所以这里我们集中讨论分支预测和用户代码优化。

10.5.1 分支优化

至此，读者应该已经非常熟悉 CPU 的取指-译码-执行周期。指令流水线对 CPU 的性能有非常重要的影响，因此大多数的现代体系结构中都加入了指令流水线作业。然而，分支转移会给流水线作业带来不利影响。考虑一个条件分支的情况，在这种情况下只有等到当前指令执行完成后才知道下一条指令的地址。这样会迫使通过流水线的指令流产生延迟，因为直到处理器完成执行这条分支指令前，处理器不知道接下来要执行哪一条指令。事实上，流水线越长（指流水线有更多的级），那么在知道接下来是哪一条指令进入流水线之前，流水线必须等待的时间越长。

现代处理器中使用的流水线越来越长。通常，大约有 20%到 30%的机器指令涉及到分支转移，而且研究指出大概有 65%的分支转移被 CPU 采用了。此外，在两次分支转移之间，各种程序平均只有 5 条指令，这样会造成许多流水线停止。因此，人们迫切地需要降低这种由于分支转移给流水线带来的不利影响。造成流水线停止的各种因素被称为冒险（hazard）。这些冒险因素包括数据相关性、资源冲

变和来自存储器的提取访问延迟。除了在检测到一个冒险时停止流水线外，我们对于流水线几乎没有其他控制。然而，分支优化处理是在我们所能够控制的范围之内。基于这个理由，分支预测已经成为人们在提高 CPU 性能方面努力发展的一个重点方向。

延迟转移（delayed branching）是处理分支转移对流水线的影响的一种办法。例如，当执行一个条件转移指令时，CPU 先执行一条或多条该分支指令后面的指令，而不管这条分支执行的结果如何。我们可以使用这种思想，否则分支后面的几个时钟周期会被浪费。要实现这种想法，可以在分支指令后插入一条指令，并在实际转移前执行这条指令。这样做的实际效果是紧跟在分支指令后的指令会在分支指令生效产生结果之前被执行。

下面的例子给出了这个概念的最好解释。考虑下面一个程序：

```
        Add    S1, R2, R3
        Branch Loop
        Div    S4, R5, R6
Loop:   Mult
```

下面是跟踪流水线的执行结果：

时间片断：	1	2	3	4	5	6
Add	F	D	E			
Branch		F	D	E		
Div			F			
Mult				F	D	E

除法指令代表了一个被浪费的指令时间片断，因为这个指令只有取指，但是由于有分支转移，这条除法指令永远不会被解码和执行。这个时间片断可以被另外的指令来填充。填充过程是通过颠倒分支指令和其他将要执行的指令的执行顺序来实现的。

时间片断：	1	2	3	4	5	6
Branch	F	D	E			
Add		F	D	E		
Mult			F	D	E	

读者应该非常清楚地知道，尽管延迟分支使用的是可能会被浪费的分支延迟的时间片断，但是延迟分支实际上对指令执行的顺序进行了重新排序。因此，编译器必须执行一个数据相关性分析来确定延迟转移是否可能。在分支转移后可能会出现没有指令可移动的情形（进入了延迟时间片断）。在这种情况下，要在分支转移后放置一个 NOP（no operation）指令，即不执行任何操作的指令。很显然，在这种情况下，分支指令的影响就与没有使用延迟转移是一样的。

编译器可以有很多方式在延迟时间段选择指令。第一种选择是不管是否有分支的出现执行一条有用的指令。分支代码段前的指令是最佳选择。另外可能包括的指令是如果发生转移执行的指令，以及不发生转移也不会出现危害的指令。相反的情况也同样考虑：如果不发生转移时的执行程序段，但是在转移发生时也不会产生危害，入选指令也包括不管是否发生转移都不会产生危害的指令。推迟转移有减低硬件代价的好处，并非常依赖于编译器在延迟时间段的填充指令。

另一种减小由于分支转移而引起的不利影响的方法是采用分支预测（branch prediction）。分支预测是指试图在指令流中猜测下一条指令的处理过程，这样可以避免由于分支指令造成流水线作业的阻塞。如果预测成功，在流水线中就不会引入延迟；如果预测不成功，则必须对流水线进行刷新，必须抛弃所有由于这种错误计算所产生的计算结果。分支预测技术根据分支转移的特点分为：循环控制分支转移、if/then/else 分支转移或子程序分支转移。

为了充分采用分支预测的好处，流水线必须保持满。因此，一旦做出一个预测，就要提取指令并开始执行。这个过程称为推测执行（speculative execution）。这些指令会在确认它们是否需要执行之前就执行。如果发现一个预测不正确，必须取消推测执行的工作。

分支预测就像一个黑盒子，我们输入不同数据，得到一个预测的目标指令作为输出。如果只是简单地将要讨论的代码输进黑盒子，这种方法就是大家所知的*静态预测*（static prediction）。如果除了这种代码之外，我们还将历史状态（有关这个分支指令的先前信息和它过去的结果）也同样输入，那么这个黑盒子就在使用*动态预测*（dynamic prediction）。*固定预测*（fixed prediction）是总是相同的预测：要么执行这个分支，要么不执行，并且每次碰到这个分支指令时，预测都是相同的。*真预测*（true prediction）只有两种可能的结果，要么“*采用分支*（take branch）”，要么“不采用分支”。

在固定预测中，如果假设不采用分支结果，那么这里所包含的意义就是已经假定了分支转移不会发生，而要继续正常的指令顺序流程。但是，分支处理还会并行地完成，以防止万一发生分支转移的情况。如果预测是正确的，那么这些预处理信息会被删除，而执行过程会继续下去。如果预测是不正确的，那么这种推测处理会被删除，并使用预处理信息继续沿着正确的指令路径执行。

在固定预测中，如果假设总是采用分支，那么同样要为一次不正确的预测做好准备。在推测处理开始前，要保存各种状态信息。如是猜测是正确的，则这个保存的信息会被删除。如果预测是不正确的，这个推测处理会被删除，而这个保存的信息要用来恢复原来采用正确路径时的执行环境。

由于使用了先前各种分支转移情况的历史记录，所以动态预测可以增强分支预测的准确性。这种信息随后会和代码一起进行组合，并输入分支预测器（即黑盒子）。用来进行分支预测的主要部件是*分支预测缓冲器*（branch prediction buffer）。这个高速缓冲器是由分支指令的低地址部分来索引的，地址的其他位用来指示这个分支转移最近是否被采用。分支预测缓冲器总是使用一个小的位数来返回预测结果。*一位动态预测*（one-bit dynamic prediction）使用一个单一位来记录上次出现的分支转移是否被采用。*二位预测*（two-bit prediction）为一个给定的分支指令保留两个先前的分支转移的历史。一个额外的位则用来帮助在循环的最后减少错误预测（即这个循环退出时，而不是像先前一样的分支转移）。这两个分支预测位可以用不同的方式表示状态信息。例如，4 种可能的位组合模式可以指示采用这个分支转移的历史概率（11：大量采用；10：很少采用；01：很少不采用；00：大量不采用）。只有当一个错误预测发生两次时，这个概率才会发生改变。

早期分支预测的实现几乎无一例外地都是使用静态预测类型。大多数比较新的处理器（包括 Pentium、PowerPC、UltraSparc 和 Motorola 68060）都使用二位动态分支预测，这样预测具有较高的准确性和较低的错误预测率。一些超标量体系结构的处理器要求使用二位动态预测，而一些其他处理器只是将它作为一种选择方案。许多系统都采用专用电路来处理分支预测问题，这样可以产生更及时和准确的预测。

10.5.2 使用好的算法和简单的代码

最好的处理器硬件和最佳的编译器也只能在一定程度上加快程序运行。它们永远无法与一个精通高效算法科学和代码设计的人相比。读者可以回忆第 6 章中介绍的有关比较行优先和列优先访问数组的例子。其中的基本思想是如果将数据的访问方式和数据的存储方式匹配得更加紧密的话，就可以达到增强系统性能的目的。例如，如果一个数组是采用行优先存储的，而我们要使用一种列优先顺序进行数据访问，那么数据的局部性原理将会减弱，潜在地导致系统的性能下降。尽管编译器可以在一定程度上改善系统的性能，但是从根本上来说，这种改善的范围是由低层次的代码优化所限制的。程序代码对系统性能的各个方面都有非常大的影响，包括从流水线作业到存储器访问直到 I/O 操作。本节将介绍一些代码优化机制。作为一个程序员，可以应用这些方法使得计算机系统达到最佳的操作性能。

操作计数（operation counting）是提高程序性能的一种方法。利用这种方法，先估算出在一个循环中所执行的指令类型的数目，然后确定每种指令类型所需的机器周期数。随后，这些信息会用来实现更好的程序指令平衡。这种思路是在某个给定的体系结构下力图使用最好的指令组合来编写程序的循环（例如，装载、存储、整数操作、浮点操作和系统调用等）。要记住的是，适用于某个硬件平台的

好的指令组合，对于另外一个不同的硬件平台未必是一个好的指令组合。

因为在程序中广泛地使用循环过程，因此循环也是系统优化的最佳选择对象。特别是嵌套循环会表现出许多令人感兴趣的优化机会。在经过一番研究后，我们可以改进存储器访问模式和增加指令级的并行执行。*循环展开*（loop unrolling）是程序员可以方便使用的一种方法。循环展开就是展开一个循环的过程，这样每个新迭代中都包含几个原来的迭代操作，因而每个循环迭代操作都可以完成更多的计算工作。利用这种方法，每次都可以通过展开的循环处理多个循环迭代操作。例如：

```
for (i = 1; i <= 30; i++)
   a[i] = a[i] + b[i] * c;
```

展开（二次）后变成：

```
for (i = 1; i <= 30; i+=3)
   { a[i] = a[i] + b[i] * c;
     a[i+1] = a[i+1] + b[i+1] * c;
     a[i+2] = a[i+2] + b[i+2] * c; }
```

粗看起来，这似乎是一种很差的代码编写方式。但是，这种代码方式减少了循环的管理费用（如下标变量的维护），而且有助于在流水线作业中控制各种冒险。这种代码结构通常允许不同循环迭代操作并行执行。另外，由于数据相关性变小，所以可以进行更好的指令调度和更好地使用寄存器。很显然，代码的数量增加了，所以这并不是在程序的每个循环中都应该采用的技术。这种方法最好是使用在占用非常多的执行时间的代码片段中。如果把循环展开应用在能够获得最大性能改进的程序部分，那么这种代码优化努力也就得到了最好的回报。这种循环展开技术也同样适用于 while 循环，虽然这种应用并不是十分直截了当。

另一种非常有用的循环优化技术是循环并合。*循环并合*（loop fusion）是使用相同的数据项将多个循环组合起来。这样可以增强高速缓存的性能，增加指令级的并行执行，并减少循环开销。下面是一个循环并合的例子：

```
for (i=0; i<N; i++)
   C[i] = A[i] + B[i];
for (i=0; i<N; i++)
   D[i] = E[i] + C[i];
```

并合结果为：

```
for (i=0; i<N; i++)  {
   C[i] = A[i] + B[i];
   D[i] = E[i] + C[i];}
```

有时，潜在的循环合并可能性不像在上述代码中显而易见。例如下面的程序段：

```
for (i=1; i<100; i++)
   A[i] = B[i] + 8;
for (i=1; i<99; i++)
   C[i] = A[i+1] * 4;
```

如何并合这些循环看起来并不是非常清楚，因为第二个循环使用了一个数组 A 中的参数值，而数组 A 是这个循环计数值前面的一个数组。但是，我们可以非常方便地将代码改写为：

```
A[1] = B[1] + 8;
for (i=2; i<100; i++)  {
   A[i] = B[i] + 8;}
for (i=1; i<99; i++)  {
   C[i] = A[i+1] * 4; }
```

现在就可以对这些循环进行并合：

```
i = 1;
A[i] = B[i] + 8;
for (j=0; j<98; j++)  {
   i = j + 2;
   A[i] = B[i] + 8;
   i = j + 1;
   C[i] = A[i+1] * 4; }
```

循环拆分（loop fission）是把一些大循环拆分成一些小循环，这种方法同样也可以用来进行循环优化，因为它可以消除某些数据相关性和减少由于数据冲突引起的高速缓存延迟。循环拆分的一个例子是循环剥离（loop peeling），这个过程是从循环中移去开始语句或结束语句。通常，这是一些包含循环边界条件的语句。例如，下面的代码：

```
for (i=1; i<N+1; i++)  {
   if (i==1)
      A[i] = 0;
   else if (i==N)
      A[i] = N;
   else
      A[i] = A[i] + 8;}
```

可以变成：

```
A[1] = 0;
for (i=2; i<N; i++){
   A[i] = A[i] + 8; }
A[N] = N;
```

这个循环剥离的例子可以产生更多的指令级并行执行，因为它从循环中移走了分支转移指令。

实际上，我们已经间接提到了循环交换（loop interchange）的概念，循环交换是一种对循环进行重新安排的处理过程，这样可以使得访问存储器的方式与数据存储的方式更加接近。在大多数编程语言中，都是采用行优先（row-major）的次序来存储循环过程。使用行优先而不是列优先的次序访问数据，可以减少高速缓存的缺失率，并产生更好的引用局部性。

循环优化是改进程序性能的一个重要工具。它很好地示范了如何使用计算机组成原理和体系结构的知识来编写优秀的程序代码。当进行程序代码优化时，请参考本书下面的补充内容中所列出的一些程序优化技巧。大家可以思考一些方法，在这些方法中每种程序优化技巧都会考虑各种不同的系统组件。并且，应该能够解释每种方法的基本原理。

尝试的事情越多，成功的机会就越大。应该记住的是，看起来应该可以提高系统性能的调整措施通常不是马上就可以取得成功。在很多时候，为了使优化的效果更加明显，必须将这些优化的概念组合起来使用。 ■

程序优化技巧

- 为编译器提供尽可能多的目的信息。尽可能使用常数和局部变量。在程序语言规则允许的情况下，定义一些原型和声明一些静态函数。尽可能使用数组来代替指针。
- 避免不必要的类型构造和尽量减少浮点数到整数的转换。
- 避免程序出现上溢和下溢。
- 使用合适的数据类型（例如，浮点、双精度、整型）。
- 考虑使用乘法来代替除法。
- 除去不必要的分支指令。
- 在可能的情况下，使用迭代方法来代替递归方法。
- 在最可能的情况下，首先建立条件语句（例如，if、switch、case）。
- 在结构中声明变量时，要遵循最大的元素首先声明的次序。
- 当程序有性能问题时，在开始程序优化之前先对程序进行剖析（profiling）处理。成型是把程序

代码拆分成小块的处理过程，而且要对这些代码小块进行定时标记以确定哪一个代码块占用的时间最多。

- 永远不要抛弃使用一种只是基于系统原始性能的算法。只有当完全优化所有的算法后才会有一个公平的算法比较出现。 ■

10.6 磁盘性能

尽管 CPU 和存储器的性能是系统性能中最重要的因素，但是理想的磁盘性能对系统的处理能力（吞吐量）也是至关重要的。大多数用户与系统交互过程都涉及到某种类型的磁盘输入或输出。另外，当出现页面错误（缺页）时，也可能发生磁盘 I/O。这种 I/O 操作的发生时间和持续长短都不是用户或程序员能够控制的。如果具有一个功能正常的 I/O 子系统，整个系统的吞吐量会比 I/O 子系统功能不佳时高出一个数量级。由于系统性能的要求很高，所以从一开始就必须对磁盘系统进行精心地设计和配置。在系统的使用寿命期内，必须经常检查磁盘子系统并进行性能调整。本节介绍有关 I/O 系统性能的主要内容。这里引入的一些基本概念，无论是对于挑选一个新的磁盘系统，还是尽力使现有的磁盘系统运行达到最佳状态，都是非常有用的。

10.6.1 性能问题

磁盘驱动器的性能问题在整个系统性能中很重要，因为相对于 CPU 或内存的速度来说，从磁盘中取回一条信息所需的时间是相当长的。CPU 在当所有的操作数都在寄存器中时，执行一条指令仅仅只需几纳秒的时间。如果在 CPU 完成一个任务之前需要从内存中取出一个操作数，那么这个执行时间将会上升到几十纳秒。但是，如果这个操作数必须从磁盘中取出时，完成任务所需的时间将猛增到几十毫秒，增加了 100 万倍！更有甚者，因为 CPU 分派 I/O 请求的速度要比磁盘驱动器跟随响应的速度快得多，所以磁盘驱动器就变成了系统吞吐量的瓶颈。事实上，当系统表现出“比较低”的 CPU 利用率时，很可能就是因为 CPU 是在频繁地等待 I/O 请求操作的完成。这样的一个系统就是属于 I/O 约束的。

I/O 性能的一个最重要的度量标准是*磁盘利用率*（disk utilization），即测量的磁盘忙于处理 I/O 请求所占时间的百分比。换一种说法就是，磁盘利用率是在另一个 I/O 请求到达磁盘服务队列时磁盘忙的概率。磁盘利用率是由磁盘速度和 I/O 请求到达服务队列的速率决定的。从数学上表述如下：

$$\text{利用率}=\frac{\text{请求到达速率}}{\text{磁盘服务速度}}$$

这里，到达速率用每秒钟的请求数来表示，磁盘服务速度则用每秒钟的 I/O 操作数（IOPS）表示。

例如，考虑一个特定的磁盘驱动器，它可以在 15 ms 内完成一个 I/O 操作。这意味着它的服务速度是大约每秒 67 个 I/O 操作（每个操作 0.015 秒 = 每秒 66.7 个操作）。如果这个磁盘每秒可以得到 33 个 I/O 请求，那么其利用率大约是 50%。在实际的任何系统中，50%的利用率已经给出了一个很好的磁盘性能。但是，如果这个系统开始发现有一个持续的速度为每秒 60 个请求的 I/O 请求速率时，又会发生什么情况呢？或者每秒 64 个请求呢？我们可以使用一个排队理论（queuing theory）的结果来模拟负载增加时的影响。简单地说，一个请求在排队上所花的时间直接与服务时间和磁盘忙的概率有关，并且间接地与磁盘空闲的概率有关。用公式表示，有：

$$\text{排队时间}=\frac{(\text{服务时间}\times\text{利用率})}{(1-\text{利用率})}$$

通过替代，很容易看到当一个 I/O 请求的到达速度是每秒 60 个请求，服务时间为 15ms 时，系统的利用率是 90%。因此，一个请求将不得不在队列中等待 135 ms。这个请求的总服务时间就是 135+15=150 ms。在每秒有 64 个请求时（仅增加 7%），完成请求所需要的时间就会激增到 370 ms（147%的增加）。而在每秒 65 个请求时，服务时间就可能超过了半秒……这哪里是 15 ms 的磁盘处理器！

从上面的公式，可以得出排队时间和利用率的关系，如图 10-1 所示。从图中可以看到，曲线的“膝部（knee）”（斜率变化最大的位置）大约是 78%。这就是为什么对大多数磁盘驱动器来说，80% 的利用率是经验法则的上限。

通过这个模型，我们很容易看到事情是如何失去控制的。如果我们有一个持续的每秒 68 个 I/O 请求的请求速率，这个磁盘就会变得不堪负荷。如果其中 25%的请求产生两个磁盘操作又会发生什么情况呢？这些假想的情形将会导致更加复杂的模型，但是这个问题的最终解决方案在于找到一些方法将服务时间保持在一个绝对的最低值。这也正是磁盘性能优化所要介绍的内容。

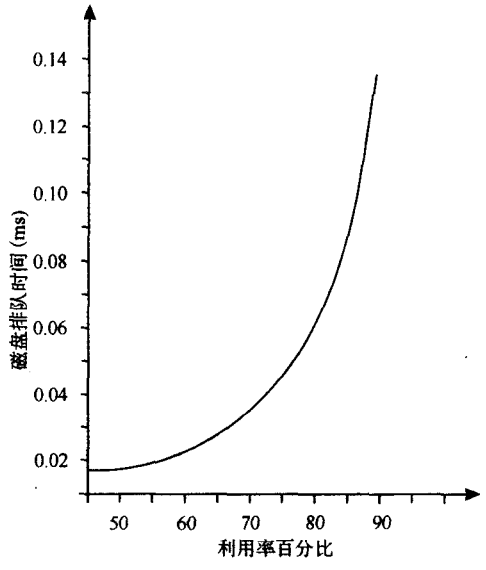


图 10-1 磁盘队列时间与百分比利用率的关系图

10.6.2 物理性能

在第 7 章中，我们介绍了确定磁盘物理性能的度量机制。这些度量机制包括旋转延迟（磁盘转速 RPM 的一个函数）、寻道时间（磁头臂定位到特定磁道所需的时间）和传输速率（磁盘读写头将数据从磁盘表面传送到系统总线的速率）。旋转延迟时间和寻道时间之和代表了磁盘的访问时间。

较低的磁盘访问时间和较高的传输速率会产生较低的总服务时间。在磁盘上增加磁碟数，或者在系统上增加更多的磁盘数目同样也可以降低服务时间。在 I/O 系统上增加双倍的磁盘数通常可以把吞吐量提高 50%。用相同数目的快速磁盘来替换现有的磁盘同样可以得到显著的性能改善。例如，用 10 000 RPM的磁盘替换 7200 RPM 的磁盘可以带来 10%～50%的性能提升。物理磁盘性能度量标准通常会公开在由磁盘生产厂家所提供的规格说明书上。因此，不同品牌之间进行性能比较，通常是非常简单的事情。但是常言道，每个人的目标是不同的。磁盘性能的提高与如何使用磁盘以及磁盘固有的容量密切相关。原始的磁盘速度是有其限制的。

10.6.3 逻辑性能

有时，要解决慢速磁盘系统的唯一办法就是增加磁盘或换掉磁盘。但是这一步骤只是在所有其他方法都失败之后才会使用。在本节接下来的部分，我们讨论一些有关磁盘的逻辑性能方面的知识。考虑磁盘性能中的逻辑，可以对磁盘进行调校和调整，而这些磁盘调整的任务应该是系统操作过程中的一种例行的工作程序。

磁盘调度

在大多数磁盘配置结构中，磁盘驱动臂的移动是花费服务时间最大的部分。平均旋转延迟，即目标扇区移动到读写头下所需的时间，对 7200 RPM 的磁盘来说大约是 4 ms，对 10 000 RPM 的磁盘来说大约是 3 ms（这个延迟时间是按照磁盘旋转半周所需的时间计算的）。对同类型的磁盘来说，平均寻道时间，即读写头位于目标磁道上所需的时间，大约为 5 到 10 ms。在许多情况下，这个时间是磁盘旋转延迟时间的两倍。而且，实际的寻道时间可能比平均时间糟糕得多。一个全程寻道（full-stroke seek）时间（磁头臂从最里面的磁道到最外面的磁道，反之亦然）可能会长达 15 到 20 ms。

很显然，改善磁盘性能的一种途径就是找到某些方法来减少磁头臂的移动。这个方法可以通过优化访问服务磁盘的扇区顺序来实现。*磁盘调度*（disk scheduling）可以是磁盘控制器的功能，也可以由主机操作系统负责处理，但是它不应该由这两者同时来完成。因为两者同时操作可能会发生调度冲突，反而降低磁盘的吞吐量。

使用最多的磁盘调度策略是先来先服务（first-come，first-served，FCFS）的方案。顾名思义，所有 I/O 请求都是按照它们到达磁盘服务队列的顺序来服务的。理解这个问题的最简单办法是列举一个例子。假设有某个磁盘有 100 个磁道，编号从 0 到 99。在这个系统运行的程序进程是按照如下的顺序发出请求从磁盘中读取磁道：

28，35，52，6，46，62，19，75，21

在 FCFS 调度策略下，假设当前正在的服务磁道是第 40 号磁道，则磁头臂移动的踪迹如图 10-2 所示。从图中可以看出，磁头臂的运动方向改变了 6 次，在完成这一系列请求的过程中一共遍历了 291 个磁道。

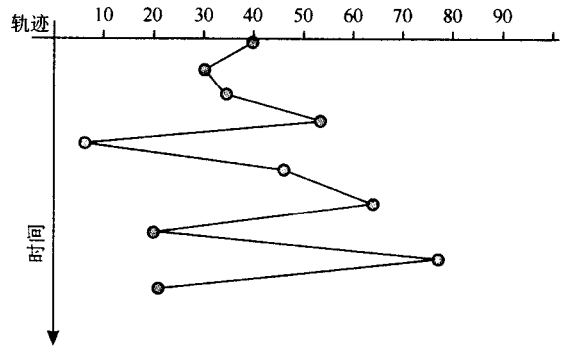


图 10-2 使用先来先服务的磁盘调度策略的寻道轨迹示意图

可以肯定，如果对这些请求进行排序，使得磁头臂每次都只移动到距离当前位置最近的磁道，那么可以大大减少磁头臂移动。最短寻道时间优先（shortest seek time first，SSTF）的调度算法就是采用了这种思想。对于上面所列出的相同的磁道请求，使用 SSTF 磁盘调度系统将会按如下顺序执行：

35，28，21，19，6，46，52，62，75

这种调度的模式如图 10-3 所示。可以看出，磁头臂的方向仅改变了一次，共遍历了 103 个磁道。SSTF 的一个缺点是有可能发生饥饿现象（资源缺乏，starvation）：理论上，只要有磁道更接近当前磁道位置的请求达到，在磁盘“远端”部分的请求磁道将会一直排在队列的最后位置。有趣的是，这个问题是磁盘利用率低的最糟糕的情况。

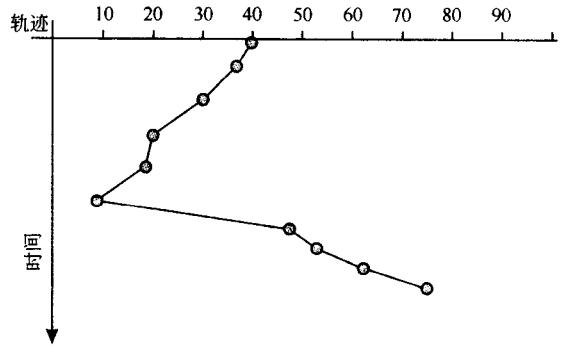


图 10-3 使用最短寻道时间优先调度算法的磁头臂移动示意图

为了避免 SSTF 的饥饿风险，必须在系统中设计一些公平的机制。为此，采用的一个简单的方法是让磁头臂在磁盘表面继续扫描，直到到达磁盘服务队列中存在的某个请求磁道时停上。这种方法被称为电梯算法（elevator algorithm），因为它与摩天大楼里的电梯载人服务的情况很相似。在磁盘调度上下文中，电梯算法被大家称为 SCAN（不是缩写）。下面将通过一个例子来描述 SCAN 是如何工作的。假设磁头臂碰巧定位在 40 号磁道，并正在向磁盘内部扫描更大编号数的磁道。现在利用与前面相同的请求队列，SCAN 会按下面的顺序读取磁道：

46，52，62，75，35，28，21，19，6

磁头臂在读取第 75 号和第 35 号磁道之间会经过第 99 号磁道，然后在读完第 6 号磁道后移动到第 0 号磁道，如图 10-4 所示。SCAN 有一种变体形式，称为 C-SCAN，即循环的 SCAN。在这种办法中，第 0 号磁道会被认为是第 99 号磁道相邻磁道。换句话说，磁头臂只沿一个方向读取，如图 10-5 所示。一旦经过了第 99 号磁道后，磁头又会返回到第 0 号磁道。因此，在本例中，C-SCAN 方法按如下的顺序读取磁道：

46，52，62，75，6，19，21，28，35

我们甚至还可以使用 LOOK 和 C-LOOK 算法，进一步减少 SCAN 和 C-SCAN 方法中的磁头臂移

动。在本例中，SCAN 和 C-SCAN 继续扫过所有的 100 个磁道。但是，事实上，请求的磁道的编号最低为 6，最高为 75。因此，如果磁头臂在读取最高编号和最低编号的磁道后就改变方向，那么磁头臂共只需要遍历 69 个磁道。这样就可以在 SCAN 和 C-SCAN 算法的基础上再节省大约 30%的磁头臂移动。

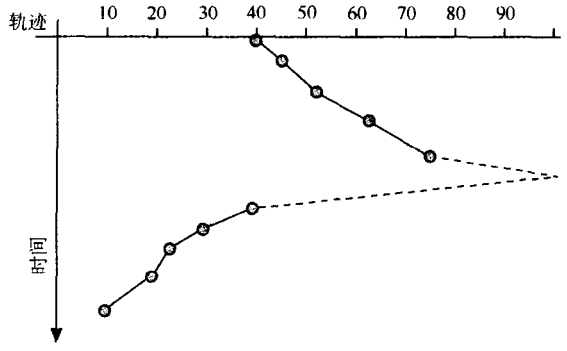


图 10-4 使用 SCAN 磁盘调度算法的磁头臂移动的示意图

有趣的是，在高利用率的情况下，SSTF 会比 SCAN 或 LOOK 完成的效果稍微好一些。但是，依然存在某个单独的请求出现饥饿的风险。在非常低的利用率（低于 20%）情况下，所有这些算法的性能都是可接受的。

鉴于前面磁盘调度算法的讨论，这里简单讨论一下有关文件的顺序存放问题。如果把使用率最高的文件存放在磁盘的中心区域将可以实现最大的磁盘性能。这一点对于磁盘目录和存储器页面（交换）文件来说特别重要。中心位置的磁头移动是最少的，因此对 SSTF 和 SCAN/LOOK 算法来说，都能达到最佳的访问时间。当文件非常零碎时磁盘性能最差，即文件被存放在多个不连续的磁盘位置。如果这个磁盘使用 SCAN/LOOK 调度方法，那么在找到文件的尾部时，可能需要经历多个磁头全程移动。因为这样，磁盘应该进行碎片整理（defragmented），或者是在某种规则基础上进行重组。此外，磁盘不应该太满。另外一个经验规则就是当磁盘容量达到 80%时，要开始考虑移走一些文件。如果没有文件可以移动，就需要增加一个新磁盘。

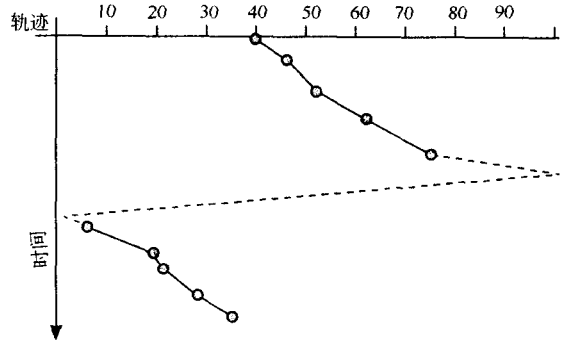


图 10-5 使用 C-SCAN 磁盘调度算法的磁头臂移动的示意图

磁盘缓存和预取

的确，减少磁头臂移动的最好方式是尽可能避免使用磁盘达到磁盘的最大可能范围。为此，许多磁盘驱动器，或者磁盘驱动器的控制器，都配有高速缓存存储器。这个高速缓存存储器可以通过留出一些主存储器的页面作为 I/O 子系统专用。磁盘高速缓存通常是关联高速缓存。因为关联高速缓存的搜索稍微有点费时，所以使用较小的磁盘高速缓存的实际性能可能会更好一些，原因是磁盘高速缓存的命中率通常会比较低。

用于 I/O 子系统的主存储器页面适合作为磁盘的二级高速缓存。在大型服务器中，为此目的所留出的页面数目是一个可调的系统参数。如果主存储器的利用率很高，分配给 I/O 子系统的页面数就必须减少。否则，将会导致过多的页面出错（缺页）数目，这会破坏使用主存储器作为 I/O 子系统高速缓存的整体目标。

主存储器的 I/O 高速缓存可以通过运行在主机上的操作系统软件进行管理，或者由产生 I/O 请求的应用程序进行管理。应有级的高速缓存管理通常可以提供优越的性能，因为这种方式可以利用应用程序的特定参数。最好的应用程序可以为用户提供对 I/O 高速缓存规模大小的某些控制，这样可以针对主机存储器的利用率来调整高速缓存的大小，防止出现过多的页面错误（缺页率）。

许多基于磁盘驱动器的高速缓存都使用预取（prefetching）技术来减少磁盘的访问。从概念上来

说，预取和 CPU 到存储器（CPU-to-memory）的高速缓存非常相似：它们都是利用局部性原理来获得更好的性能。当使用预取技术时，会同时读取磁盘中包含目标扇区的后续的若干个扇区，并且期望其中一个或多个后续扇区不久就会成为所需求的扇区。经验研究表明，超过 50%的磁盘访问性质上是顺序访问。平均来说，预取技术可以提升 40%的性能。

预取技术的缺点是存在*高速缓存污染*（Cache pollution）现象。如果高速缓存中被填满一些没有任何进程所需的数据时，就发生了高速缓存污染，这样为有用的数据留下的空间就非常少了。在使用主存储器高速缓存时，可以使用各种不同的置换算法来保持高速缓存的清洁。这些策略包括在 CPU 到存储器的高速缓存中使用置换算法（例如，LRU、LFU 和随机置换算法）。另外，由于磁盘高速缓存用作写入磁盘的数据的转运站，所以某些磁盘高速缓存的管理方案是在高速缓存中的数据被写入磁盘后，简单地清除磁盘高速缓存中的所有数据字节。

从磁盘中读取数据和向磁盘写入数据这两种操作在本质上存在的差别会引发了一些非常棘手的高速缓存问题。首先也是最重要的问题是高速缓存是一种易失性的存储器。在一个大规模的系统失效事件中，高速缓存中的数据将会丢失。假设某个运行在主机上的应用程序认为数据已经提交到了磁盘，但是实际上这些数据只是驻留在高速缓存中。如果高速缓存失效，这些数据也就消失了。当然，这种情况可能引起严重的数据不一致问题，例如一个 ATM 机器在没有登记账目的情况下就给顾客付了款。

为了防止由于断电而造成高速缓存的数据丢失，一些基于磁盘控制器高速缓存通常都使用*镜像*（mirrored）高速缓存，并且提供电池作为备份电源。当对高速缓存进行镜像映射时，磁盘控制器会有两个采用前后串联方式的完全相同的高速缓存在工作，这两个高速缓存在任何时刻都保存相同的数据。另一种途径是采用在第 6 章中讨论的写通（write-through）高速缓存。在这种高速缓存中会保留一份数据副本，以防止这些数据“不久”会再次需要，但同时这些数据也会被写入到磁盘中。操作系统只会在数据已经实际存放在磁盘上后，才发出 I/O 已经完成的信号。因此，为了获得更好的可靠性，通常需要在一定程度上进行系统性能平衡。

当系统的吞吐量比可靠性更重要时，可以采用回写高速缓存策略。读者不妨回忆一下，实际上有两种类型的回写策略。最简单的方法是磁盘周期性地读取高速缓存（通常一分钟读两次），并将所找到的所有的脏数据块写入磁盘。如果主要关心的是系统的可靠性，那么可以将提交数据的间隔时间变得更短（以牺牲性能为代价）。一个更加复杂的回写算法是*采用机会*（opportunistic）写入。使用这种方法，脏数据块会一直在高速缓存中等待，直到有一个位于同一磁道柱面的读请求到来。接着，在读操作时会进行“*捎带确认*（piggybacked）”的写入操作。这种方法会对磁盘的读性能的降低产生影响，但是对写性能会有帮助。许多系统都采用周期回写和机会回写这两种策略的组合形式，以设法在效率和可靠性之间达到某种平衡。

在优化磁盘性能方面，这些平衡性能的折衷方案常常会面临一些非常困难的选择。我们的首要责任是要确保数据的可靠性和一致性。但是，如果使用易失性高速缓存来补偿访问磁盘时间的延迟，能够实现最大的吞吐量。使用电池备份的高速缓存的成本非常高。增加磁盘数来提高吞吐量同样也是一种昂贵的选择。如果从磁盘写操作中移去高速缓存这个中间介质可能会导致磁盘性能的降低，特别是在磁盘利用率高的情况下。用户很快就会抱怨过长的磁盘反应时间，而当你为磁盘升级向财务人员申请经费时，他们也会抱怨费用太高。需要牢记的是，不管新磁盘的价格如何，升级磁盘子系统通常比替换丢失的数据要合算得多。

本章小结

本章介绍了有关计算机性能的两个方面的内容：性能评价和性能优化。通过本章的学习，读者应该了解一些计算机性能的关键测量方法，并能够正确地总结和归纳这些方法。特别是读者应该理解算术平均值对高度变化的数据并不合适，而且不应该使用在速率和比率的测评上。对于高度变化的数据，几何平均值

非常有用。但是，几何平均值不能用来预测性能。调和平均值在比较速率时很适合，而且也是一个有用的性能预测器。但是，如果使用调和平均值来比较系统的相对性能，那么调和平均值在选择参考机器时要比几何平均值更加敏感。

本章我们解释了一些非常流行的基准程序和套件。在这些基准中，最可靠的是由像 SPEC 和 TPC 这样的公平监管的组织所颁布和管理的基准。不管使用哪种基准，都应该按照用户特定的应用内容对基准程序进行解释。记住，没有一个单一的度量标准能够对所有的情况都通用。

计算机性能直接依赖于计算机组件的优化。这里，我们仔细研究了影响计算机系统组件主要性能的各种因素。Amdahl 定律为我们提供了一种方法，利用 Amdahl 定律可以确定各种不同的优化技术可以获得的潜在的系统性能的加速率。而且，Amdahl 定律也为性能增强的可能性设置了一个上限。性能优化需要考虑的内容包括 CPU 性能、存储器性能和 I/O 性能。CPU 性能与程序代码、编译器技术、指令系统（ISA）和硬件技术密切相关。分支指令对流水线的性能有重大影响，进而对 CPU 性能产生很重要的影响。分支预测是一种补偿由于分支转移所带来的程序执行复杂性的方法。固定分支预测和静态分支预测方法都没有动态技术准确，但是它们实现的成本也比较低。

I/O 性能同时是磁盘驱动器逻辑特性和物理特征的函数。只要不更换硬件，就无法改善磁盘系统的物理性能。但是，仍然可以对磁盘的很多逻辑性能进行调整和优化。这些因素包括磁盘利用率，文件存放位置，以及存储器高速缓存的大小。好的性能报告工具不仅可以提供完整的 I/O 统计数据，而且可以提供各种调整方案的建议。

系统性能的评估和优化是系统管理人员的两项最重要的任务。在本章中，我们只是介绍了一些与系统平台无关的知识内容。最有帮助和最有趣的信息可以在计算机系统制造厂商所提供的各种手册和培训研讨会中获取。这些资源对于读者继续从事调整系统的有效性的工作是很重要的。

深入阅读

在整体计算机设计方面，Hennessy 和 Patterson（1996）介绍了计算机性能。他们的书中包含了有关计算机体系结构性能的各个方面的内容，这本书的处理手法受到大家的好评。Lilja（2000）的著作覆盖了系统度量方法的设计和分析，对计算机性能有一个全面的介绍。这本书中所介绍的数学评估方法，思想特别清晰，可读性很强。要了解高性能计算的详细信息，出色的编程和调整软件，以及基准等知识，可以参阅 Severance 和 Dowd（1998）的书籍。Musumeci 和 Loukides（2002）的著作也同样非常好地介绍了有关系统性能调整方面的知识。

除了上面介绍的参考书外，Fleming 和 Wallace（1996）和 Smith（1998）的论文则讨论了如何选择正确的统计方法的数学基础。并且对某些有争议的性能度量标准提出了自己的一些深入见解。

在本书有关统计方法的缺陷和谬论的讨论过程中，没有涉及到统计学的图形表示方法，这种方法也会产生一些不正确的结论和假设。如果希望了解如何利用图形工具来传递（和遮蔽）各种统计信息的艺术，推荐读者完整地阅读 Tufte（2001）的著作。读者在阅读之余肯定会有惊奇、欣喜和困惑等滋味。这里，还推荐 Huff（1993）经典著作。这本薄薄的书籍（第一版于 1954 年发行）中的各种资料和插图已经为读者服务了 50 多年。

Price（1989）的文章很好地介绍了许多早期的综合基准，包括 Whetstone、Dhrystone 和许多其他在本章中没有提到的基准。Serlin（1986）的文章是另外一篇全面介绍早期基准的文章。Weicker（1984）的文章包含对他自己的 Dhrystones 基准的最原始的介绍。Grace（1996）的著作则是描述每种主要基准的百科全书。这本书中包括了本章中讨论的所有基准和许多其他基准，甚至包括针对 Windows 和 Unix 环境的专用基准。

令人奇怪的是，从标准基准程序中得到 MFLOPS 数字与从早期的浮点 SPEC 基准得到数字有很强的相关性。在 Giladi（1996）的文章中对这个异常现象做了很好的讨论，而且这篇文章也同样提供了有关 Linpack 浮点基准的详细讨论。现在，有关 SPEC 基准的完整信息可以在 SPEC 的站点 www. spec. org 上找到。

对于 TPC 基准来说，开创性的文章是由 Jim Gray (1985) 匿名发表的。他把这个荣誉赠与了许许多多对这篇文章的最后成型有影响的其他人，他曾经将他的文章给上千人传阅以争求他们的意见。他拒绝使用个人的著作权，并且匿名发表这篇文章。因此，许多人在引用这篇文章时都是用 "Anon.，et al"。这篇文章给出了事务处理委员会的背景和基本思想。现在，在 TPC 的网站 www.tpc.org 可以找到更多的信息。

Hsu、Smith 和 Young (2001) 在文章中详细地研究了使用真实的工作量对各种 TPC 基准的比较。这篇文章也展示了使用跟踪分析方法搜集到的各种性能数据。

有关 I/O 系统性能方面的介绍也有大量的资料。在 Hennessy 和 Patterson (1996) 的书中就详细讨论了这个问题。有关磁盘调度策略的研究（虽然有些过时）可以参阅 Oney (1975) 的论文。Karedla、Love 和 Wherry (1994) 的文章则对磁盘高速缓存的性能做了清晰和详细的讨论。Reddy (1992) 对 I/O 系统体系结构做了很好的全面回顾。

参考文献

Fleming, Philip J., & Wallace, John J. "How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results." *Communications of the ACM 29:3* (March 1996), pp. 218–221.

Giladi, Ran. "Evaluating the MFLOPS Measure." *IEEE Micro.* (August 1996), pp. 69–75.

Grace, Rich. *The Benchmark Book.* Upper Saddle River, NJ: Prentice Hall, 1996.

Gray, Jim, et al. "A Measure of Transaction Processing Power." *Datamation 31*:7 (1985), pp. 112–118.

Hennessy, John L., & Patterson, David A. *Computer Architecture: A Quantitative Approach.* San Francisco: Morgan Kaufmann Publishers, 1996.

Hsu, Windsor W., Smith, Alan Jay, & Young, Honesty C. "I/O Reference Behavior of Production Database Workloads and the TPC Benchmarks—An Analysis at the Logical Level." *ACM Transactions on Database Systems 26*:1 (March 2001), pp. 96—143.

Huff, Darrell. *How to Lie with Statistics.* New York: W.W. Norton & Company, 1993.

Karedla, Ramakrishna, Love, J. Spencer, & Wherry, Bradley G. "Caching Strategies to Improve Disk System Performance." *IEEE Computer* (March 1994), pp. 38—46.

Lilja, David J. *Measuring Computer Performance: A Practitioner's Guide.* New York: Cambridge University Press, 2000.

Musumeci, Gian-Paolo, & Loukides, Mike. *System Performance Tuning, 2nd ed.* Sebastopol, CA: O'Reilly & Associates, Inc., 2002.

Oney, Walter C. "Queueing Analysis of the Scan Policy for Moving-Head Disks." *Journal of the ACM 22* (July 1975), pp. 397–412.

Price, Walter J. "A Benchmarking Tutorial." *IEEE Microcomputer* (October 1989), pp. 28–43.

Reddy, A. L. Narasimha. "A Study of I/O System Organization." *ACM SIGARCH Proceedings of the 19th Annual International Symposium on Computer Architecture 20*:2 (April 1992), pp. 308–317.

Serlin, Omri. "MIPS, Dhrystones, and Other Tales." *Datamation.* June 1, 1986, pp. 112–118.

Severance, Charles, & Dowd, Kevin. *High Performance Computing, 2nd ed.* Sebastopol, CA: O'Reilly & Associates, Inc., 1998.

Smith, James E. "Characterizing Computer Performance with a Single Number." *Communications of the ACM 32*:10 (October 1998), pp. 1202–1206.

Tufte, Edward R. *The Visual Display of Quantitative Information, 2nd ed.* Cheshire, CT: Graphics Press, 2001.

Weicker, Reinhold P. "Dhrystone: A Synthetic Systems Programming Benchmark." *Communications of the ACM 27* (October 1984), pp. 1013–1029.

Weicker, Reinhold P. "An Overview of Common Benchmarks." *IEEE Computer* (December 1990), pp. 65–75.

基本概念和术语复习

1. 解释程序或系统属于存储器约束的含义。本章还讨论了其他什么类型的约束？
2. 关于性能优化，Amdahl 定律告诉了我们什么？
3. 对于比较速率，哪种平均值方法最有用？
4. 什么类型的数据使用算术平均值不合适？
5. 给出一个最佳性能（optimum performance）的定义。
6. 什么是性价比？什么原因使得它很难应用？
7. 使用 MIPS 或 FLOPS 作为系统吞吐量（处理能力）的度量手段有什么缺点？
8. Dhrystone 基准与 Whetstone 和 Linpack 基准之间有什么区别？
9. 针对 SPEC 的 CPU 基准来说，Whetstone、Dhrystone 和 Linpack 基准有什么不足之处？
10. 解释术语基准测试（benchmarking）。
11. TPC 重点关注的问题和 SPEC 有什么不同？
12. 解释延迟分支转移（delayed branching）。
13. 什么是分支预测？分支预测用来做什么？
14. 列举二种流水线冒险的例子。
15. 定义下列术语：循环并合（loop fusion）、循环分拆（loop fission）、循环剥离（loop peeling）和循环交换（loop interchange）。
16. 按照排队理论，临界磁盘百分比利用率是多少？
17. 在使用 SSTF 磁盘调度算法中会遇到什么风险？
18. LOOK 和 SCAN 算法有什么不同？
19. 什么是磁盘预取？它有什么优点和缺点？
20. 使用高速缓存的磁盘写入方式有什么优点和缺点？

练习题

◆1. 表 10-2 表示的是两台计算机系统 A 和系统 C 的任务执行组合和程序运行时间。在这个例子中，系统 C 比系统 A 快 83%。表 10-3 描述的是系统 A 使用不同的任务执行组合时的程序运行时间。利用表 10-3 给出的执行组合，计算出系统 C 比系统 A 快的百分比。与表 10-2 中的原始统计数据相比较，系统 A 在新的执行组合条件下，性能下降了多少？

2. 利用第 10.3 节中针对程序 v、w、x、y 和 z 所引用的性能数据，求出这些程序在系统 B 和系统 C 上的运行时间的几何平均值，计算中选取系统 A 作为一个参考机器。证明这个平均值比值和选取其他两个系统作为参考系统所得到的结果一致。

◆3. 在下表中给出了 3 个系统运行 5 个基准的执行时间。分别使用算术平均值和几何平均值比较这些系统之间的相对性能（即，A 比 B，B 比 C 和 A 比 C）。是否有奇怪的现象？请解释原因。

4. 下表中给出了 3 个系统运行 5 个基准的执行时间。分别使用算术平均值和几何平均值比较这些系统之间的相对性能（即，A 比 B，B 比 C 和 A 比 C）。是否有奇怪的现象？请解释原因。

程 序	系统 A 执行时间	系统 B 执行时间	系统 C 执行时间
v	150	200	75
w	200	250	150
x	250	175	200
y	400	800	500
z	1000	1200	1100

程 序	系统 A 执行时间	系统 B 执行时间	系统 C 执行时间
v	45	125	75
w	300	275	350
x	250	100	800
y	400	800	500
z	800	1200	700

5. 一家销售数据库管理优化软件的公司与你联系，想推销产品。公司的销售代表声称这个存储器管理软件可以降低系统的页面出错率（缺页率）。她给你提供这个软件的 30 天的免费试用期。但是，在安装该软件之前，你决定首先为系统确定一个基准线。在这天的某个特定的时间，你对系统进行了采样，并记录页面出错率（使用这个系统的诊断软件）。在安装该管理软件之后，你又做了同样的诊断。请问，这个新的管理软件所带来的系统的平均性能提高是多少？

出错率和诊断时间如右表所示：

时　间	软件安装前的出错率	软件安装后的出错率
02:00—03:00	35%	45%
10:00—11:00	42%	38%
13:00—14:00	12%	10%
18:00—19:00	20%	22%

6. 使用像 Whetstone 和 Dhrystone 之类的综合基准有什么限制？你认为能否扩展综合基准的这些概念以克服这些限制？解释你的答案。

◆7. 如果供应商声称其系统运行 50% 的 SPEC 的基准内核程序要比它的主要竞争系统快两倍，你会怎样认为？这里出现了什么样的统计谬论？

8. 假设你正在进行调查准备购买一个新的计算机系统。除了系统 X 的型号 Q 外，你已经知道了所考虑的所有系统基准结果。但是你有系统 X 的型号 S 的基准结果，而且这些结果没有其他一些竞争品牌的性能好。为了完成调查研究，你打电话到系统 X 的计算机公司，问他们计划何时公布型号 Q 的测试结果。他们告诉你近期不会公布这些结果，但是因为型号 Q 的硬盘驱动器的平均访问时间是 12ms，而型号 S 为 15ms，所以型号 Q 的性能将比型号 S 的性能高出 25%。你怎样评价系统 X 的型号 Q 的这种性能度量方法呢？

◆9. 你认为对于两个不同的 SPEC CPU 基准的版本，即 SPEC 95 和 SPEC 2000，应该比较的数值是什么？

10. 除了零售商部门以外，还有其他什么部门需要性能好的事务处理系统。证明你的结论。

◆11. 如果你要购买一台计算机系统打算从事 DNA 研究，那么在本章中所讨论的哪一个基准最有用？为什么会选择这一基准？对于其他基准，你是否还感兴趣？请说明为什么感兴趣，或者为什么不感兴趣？

12. 假设一位朋友让你帮助做出选择，作为在家个人使用应该购买什么类型的计算机。在不同的品牌和型号比较中，你要寻找的是什么样的系统性能？而如果你是在帮助雇主购买一台 Web 服务器来从事在网上接受客户订单的工作，那么在这种情形下，你的想法又会有什么不同？

◆13. 假设你恰好被分配到某个委员会，委员你购买一台新的企业级文件服务器。这个服务器能够支持客户的各种账目活动，同时还要有许多的管理功能，例如可以生成每周的工资表。委员会的一位同事正好知道某个特定的系统在 SPEC CPU 2000 基准的竞赛中脱颖而出。现在，这位同事坚持要委员会购买一台这种系统的计算机。请问，你对这个事情有什么反应？

*14. 我们曾经讨论过调和平均值在用于评估计算机性能时所受到的限制。一些批评者建议 SPEC 应该使用调和平均值来代替。设计一个“工作”单元，这个工作单元适合用来重新对 SPEC 基准进行规划，使其成为一个速率度量标准。请使用来自 SPEC 网站 www.spec.org 的一些结果，来测试你的理论。

*15. SPEC 和 TPC 都为 Web 服务器系统发布了基准。分别浏览这两个组织的网站（www.spec.org 和 www.tpc.org），努力找出已经在两个网站都公布了结果的相同的（或者是可以进行比较的）系统，并对发现进行讨论。

16. 我们曾经提到过，在系统探测跟踪的过程中会搜集到大量的数据。为了给你一个有关这些数据的实际数量的一些概念，假设现在计划安装了一个硬件探测器来记录系统的程序计数器、指令寄存器、累加器、内存地址寄存器，以及存储器缓冲寄存器的内容。系统有一个在 1GHz 频率运行的时钟。在系统时钟的每个时钟周期内，5 个寄存器的内容都会被写入探测器电路附属的非易失性存储器中。假设每个寄存器都是 64 位宽，如果要搜集 2 秒时间的数据，那么探测器需要配备多大的存储器？

17. 在第 10.5.2 节中的补充内容中介绍了一些提升程序性能的技巧方法。其中每个小技巧都涉及到一个计算机组成原理或者是体系结构的问题。如果真是这样，请给出这些技巧建议者后的理由。如果你认为这些补充内容中缺少了什么，请在分析过程中加入你自己的建议。

18. 在我们讨论磁盘性能的物理特性时，曾经指出用 10000 RPM 的磁盘替代 7200 RPM 的磁盘可以带来 10～50%的性能改善。为什么只有 10%的性能提升呢？是否有可能根本就没有改进作用？解释原因。
19. 对于下面给出的一系列磁道服务请求，分别使用 FCFS、SSTF、SCAN 和 LOOK 算法，计算出磁头臂遍历的磁道数。当第一个请求到达磁盘请求队列时，读/写头正位于第 50 号磁道处，并且是向外层（编号较低的）磁道方向移动（提示：计算磁头臂所经过的磁道数的总和，而不管这个磁道是否会被读取）。

    54，36，21，74，46，35，26，67
20. 使用如下的磁道请求顺序，计算第 19 题的问题：

    82，97，35，75，53，47，17，11
21. 在某个特定品牌的磁盘驱动器上，磁头臂不停顿地经过一个单一磁道所花费的时间为 500 纳秒。然而，一旦磁头到达一个有服务请求的磁道时，则需要停留在目标磁道上方有 2 毫秒的“安置（settle）”时间，然后才开始数据的读写操作。基于上面这些时间规定，分别比较使用 FCFS、SSTF 和 LOOK 算法执行如下的调度时所需要的相对时间。这里，需要比较 SSTF 对 FCFS，LOOK 对 FCFS，以及 LOOK 对 SSTF 的时间。就像在前面的问题中一样，当第一个请求到达磁盘请求队列中时，读/写头在第 50 号磁道的地方，并且正在向外层（编号较低的）磁道方向移动。请求磁道如下：

    35，53，90，67，79，37，76，47
22. 使用如下磁道请求顺序重做练习题 21（假定读写头位于第 50 号磁道处，正在向外移动）：

    48，14，85，35，84，61，30，22

*23. 在讨论 SSTF 磁盘调度算法时，我们曾经指出饥饿问题“是低磁盘利用率的最差的情况”。解释为什么会是这样。

24. 某个特定的微处理器完成各种不同的操作可能需要 2、3、4、8 或 12 个机器周期数。其中，25%的指令需要 2 个机器周期，20%的指令需要 3 个机器周期，17.5%的指令需要 4 个机器周期，12.5%的指令需要 8 个机器周期，还有 25%的指令需要 12 个机器周期。

    ◆a) 这个微处理器的每条指令的平均机器周期数是多少？

    ◆b) 如果微处理器为一个“1MIPS”的处理器，那么要求时钟频率（每秒机器周期数）是多少？

    ◆c) 假设这个系统需要额外的 20 个时钟周期从存储器中取回一个操作数。其中 40%的时间必须用来访问存储器。这个微处理器每条指令的平均机器周期数又会是多少？其中包括了微处理器的存储器提取指令。

25. 某个特定的微处理器完成各种不同的操作可能需要 2、4、8、12 或 16 个机器周期。其中，17.5%的指令需要 2 个机器周期，12.5%的指令需要 4 个机器周期，35%的指令需要 8 个机器周期，20%的指令需要 12 个机器周期，还有 15%的指令需要 16 个机器周期。

    a) 这个微处理器每条指令的平均机器周期数是多少？

    b) 如果这个微处理器是一个“1MIPS”的处理器，那么要求时钟频率（每秒机器周期数）是多少？

    c) 假设这个系统需要额外的 16 个周期来执行从存储器中取回一个操作数。其中 30%的时间必须用来访问存储器。那么这个微处理器每条指令的平均机器周期数又是多少？其中包括微处理器的存储器提取指令。

网络就是计算机。
——Sun 微系统公司

第 11 章　网络组织和体系结构

11.1　概述

Sun 微系统公司在 20 世纪 80 年代发起了一场大规模的广告活动，其宣传口号是脍炙人口的“网络就是计算机”。20 年前，这句话确实很难引起人们的注意，但是它就像旷野中的呼唤，向人们预告着今天这个处于全球商业核心的万维网的到来。如今，孤立的商业计算机已经变得过时和无用了。

本章将介绍庞大而复杂的数据通信领域，其中重点关注因特网。我们将会从历史的、理论的、实践的观点来看待网络的结构模型（网络协议）。理解了网络运行的原理之后，我们将学习构建网络组织的各种设备组件。本章的目的是给读者一个宽广的视野，使大家尽可能了解每个计算机专业人员在职业生涯中可能会遇到的各种技术和术语。所以说，要理解计算机同样也需要理解网络。

11.2　早期的商业计算机网络

当今的计算机网络沿着两种不同的方向发展。一种致力于更快更准确的商务交易，而另一种则致力于促进学术和科学领域中的协作和知识共享。

各种数字网络方式以最简单、快速和经济高效的方式共享计算机资源。计算机价格昂贵，众多用户要求共享资源的愿望也就会越强烈。在 20 世纪 50 年代，大部分计算机的价格都要几百万美元，所以只有财力雄厚的公司才购买得起多台计算机系统。当然，在偏远地区工作的职员与在总部工作的同事一样需要共享计算机资源，因此人们发明了一些将他们的计算机与总部计算机连接起来的方法。实际上，几乎每个计算机制造商在当时都有自己独特的连接方案，其中占主导优势的是 IBM 公司，它拥有自己的系统网络体系结构（system network architecture，SNA）。这种通信体系结构，通过不断完善，已经持续使用了 30 多年。

IBM 的系统网络体系结构（SNA）是一种端到端（end-to-end）的通信规范，利用这种规范，可以实现物理设备（称为物理单元，physical unit，PU）之间的逻辑会话（称为逻辑单元，logical unit，LU）。在最初的体系结构中，这种通信系统的物理部件由终端、打印机、通信控制器、多路复用器和前端处理器组成。前端处理器位于主机系统和通信线路之间，它负责管理所有的通信开销，包括轮询（轮流检测）每个通信控制器，而这些通信控制器又负责轮询每个与它们连接的终端。这种体系结构如图 11.1 所示。

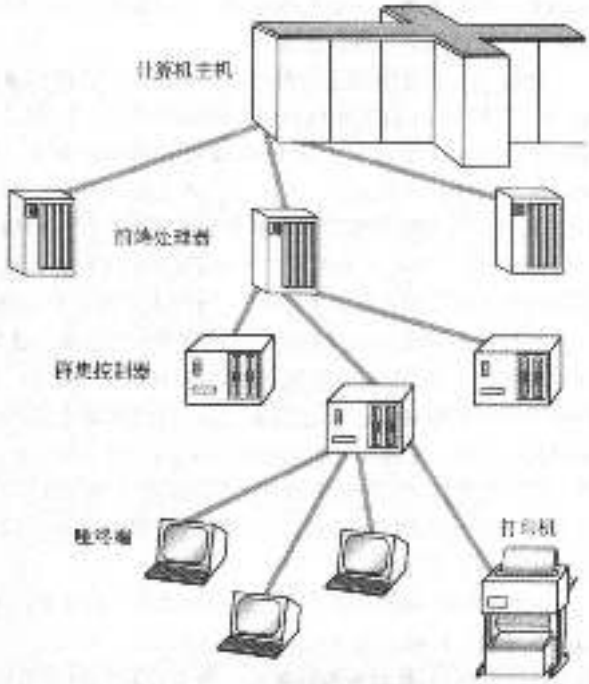


图 11.1　一种分层的轮询网络

IBM 的系统网络体系结构是设计用于高速的事务处理和客户服务查询的。即使在 9600 b/s（位/秒）的中等线路速度的条件下，如果所有网络设备在正常的负载下都正常运行，那么对主机上的数据访问也几乎是同时进行的。然而，这种体系结构的速率是以牺牲系统的灵活性和设备之间的协同工作能力为代价的。管理和维护这些网络方面需要巨大的人力开销，而且在与其他制造商的设备和网络的连接方面通常需要高度的软件和硬件工程技巧。在过去 30 多年，SNA 已经适应了千变万化的商务需求和网络环境，但是这些基本的概念本质上还是几十年前提出来的。事实上，这种结构设计得非常好，SNA 的各方面的内容后来成为权威的国际通信体系结构 OSI 的基础。OSI（即开放系统互连）将在 11.4 节中讨论。虽然 SNA 对于新兴的数据通信科学的贡献颇多，但是这种技术多年来几乎只是按常规发展。在大多数网络装置中，系统网络体系结构已经被“开放式”因特网协议所取代。

11.3 早期的学术和科学网络：因特网的起源和体系结构

在冷战期间，大多数研究实验室的美国科学家都在政府的合同下辛苦工作，致力于维护美国在军事上的领先地位。曾经一度在技术竞争上处于落后地位时，美国政府成立了一个称为高级研究计划署（Advanced Research Projects Agency，ARPA）的组织。然而，当时这个组织完成各种使命所需要的各种高级计算机，即使按照五角大楼（美国国防部）的标准，这些计算机也是非常稀少和极其昂贵的。不久，有人提议建立一种通信链接，将分散在美国各地的少数几台超级型计算机连接起来，这样许多志趣相投的研究人员可以共享计算资源。而且，即使核战争摧毁了网络中的大量节点或者通信线路，这种网络还设计有足够多的冗余资源，可以维持计算机之间的不间断通信。1968 年 12 月末，在美国马萨诸塞州的剑桥，BBN 咨询公司（Bolt、Beranek 和 Newman，现在的 Genuity 公司）拿到了这个合同，负责构建该网络。1969 年 12 月，4 个节点，即犹他大学、加利福尼亚大学洛杉矶分校、加利福尼亚大学圣芭芭拉分校和斯坦福（Stanford）研究所，实现了联机运行。此后，ARPA 网络不断扩大，包括了更多的美国政府和研究机构。后来，美国总统里根将 ARPA 改名为 DRAPA（Defense Advanced Research Projects Agency），ARPAnet 就变了 DARPAnet。在 20 世纪 80 年代早期，这个网络的节点扩充速率大约是每月增加一个节点多一点。然而，军事研究家们最终还是放弃了 DARPAnet，而决定采用更安全的通信信道。

1985 年，为了方便进行科学和学术研究，美国国家科学基金会创立了自己的网络 NSFnet。NSFnet 和 DARPAnet 有着相似的用途和用户群体，然而，NSFnet 在性能上要优于 DARPAnet。结果，在军方放弃 DARPAnet 时，NSFnet 吸收了 DARPAnet，而且 NSFnet 发展成为了今天众所周知的因特网。到 20 世纪 90 年早期，NSF 已经派生出 NSFnet，于是 NSF 就开始建造一个更快、更可靠的 NSFnet。后来，公用网络的管理事务落到了一些私有化的国家和地区性的企业的手中，例如 Sprint、MCI 和 PacBell 等。这些公司买下了 NSFnet 的中继线路，称为主干网络（或高速链路，backbones），并且通过向各个网络服务提供者（ISP）出售主干网络的流量来盈利。

最初的 DARPAnet（和现在的因特网）可以从核战争中幸存下来，原因是 DARPAnet 网不同于所有 20 世纪 70 年代构建的其他网络，在 DARPAnet 中，系统和系统之间没有专用的连接线路。相反，DARPAnet 只要有可以利用的路径就可以通过路由发送信息。属于单一对话的数据流部分可以经过不同的路径到达目的地。这里的关键是采用一种自带地址信息的数据报（datagram，一种自带寻址信息的、独立地从发送端行走到接收端的数据包）的思想，这种数据报是以数据块而不是以 SNA 模型所使用的数据流的形式传送数据。每个数据报都包含地址信息，这样每个数据报都可以作为一个单一的、不连续的单元进行路由。

DARPAnet 的第二个革命性的内容就是，对于不同类型的主机之间通过不同速度的网络进行的通信，创立了一种统一的协议。因为它连接了不同种类的网络，所以 DARPAnet 也称为一种互连网络（internetwork）。根据最初的规定，每个计算机的主机需要通过一个接口消息处理器（interface message processor，IMP）的方式才能够连接到 DARPAnet。IMP 负责从 DARPAnet 的语言到主机系统本

身的通信语言之间的协议转换。因此在 IMP 和主机之间可以使用任意的通信协议。现在，路由器（将在 11.6.7 节中讨论）已经替代了 IMP。并且现在的通信协议也远没有 20 世纪 70 年代时的那么多种类了。但是基本原理仍然是相同的，而且互连网络的普通概念也与因特网中是一样的。一个现代互连网络的结构如图 11-2 所示。图中说明的是 4 个路由器构成了网络的核心。这些路由器连接多种不同类型的设备，数据报自己决定怎样以尽可能最有效的方式到达目的地。

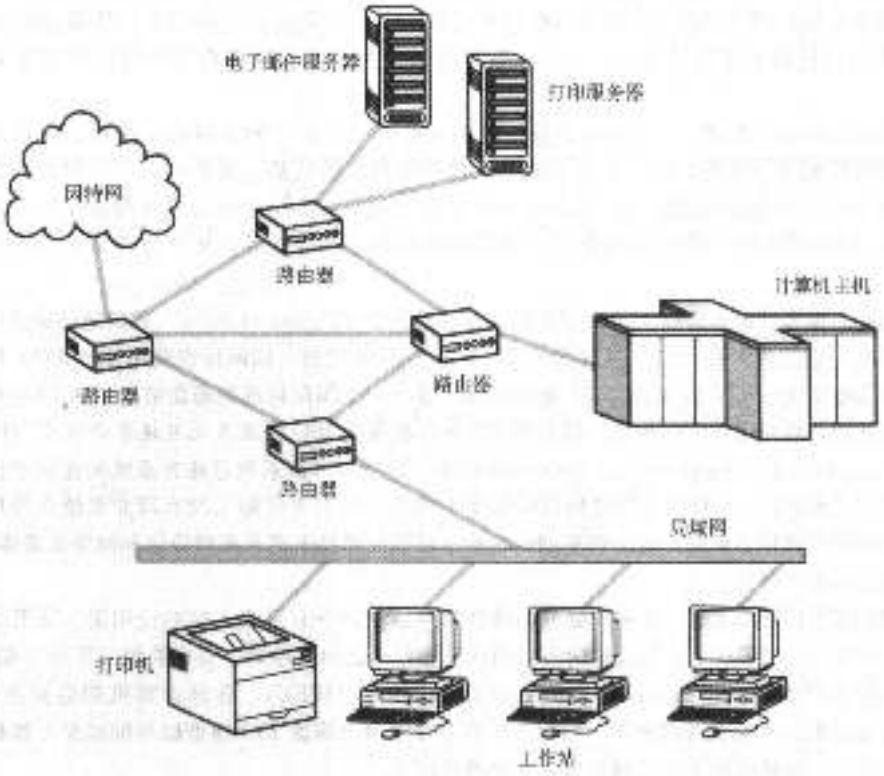


图 11-2 一个互连网络的例子

网络远远不只是一组数据通信的规范。也许它是一种哲学。哲学最重要的思想就是自由和信息共享的开放世界，人类通过人和思想的结合决定世界的命运。公开的体现是一种风格，而这种风格创建了网络标准。在因特网架构委员会（Internet Architecture Board，IAB）的支持下，通过一个民主过程形成了网络标准。因特网架构委员会是在非盈利网际协会（Internet Society，ISOC）的监管下运作的。因特网工程任务组（Internet Engineering Task Force，IETF）则属于因特网架构委员会，是由制定详细网际协议规范的工业界专家组成的一个松散联盟。IETF 以 RFC（Request of Comment，请求评论）的形式发布各种标准的建议。RFC 对任何人的检查和评论都是公开的。现在，有两个最重要的 RFC：RFC791（网际协议 4）和 RFC793（传输控制协议），它们形成了当今全球网络的基础。

ISOC 委员会的多层组织结构可能带有官僚机构的混乱管理模式，产生一些模棱两可的规范。但是由于整个过程的公开性和评论家们的天资，所以 RFC 在各种网络文献中属于最清楚和最易读懂的协议文件。因此，不足为奇，网络设备制造商很快就适应了网际协议。目前，网际协议应用于各种网络：包括公众网络和专用网络。以前，网络标准是由某个权力委员会或设备供应厂商传下来的。这种方式导致了 ISO/OSI（国际标准化组织/开放系统互连网）协议模型的形成，这些内容将在下面讨论。

11.4 网络协议 1：ISO/OSI 协议

在第 7 章中，我们曾介绍了各种数据存储器接口是如何使用协议（堆）栈的。SCSI - 3 体系结构模型就是其中之一。一般来说，协议堆栈使得各种类型的接口具有可移植性、可维护性且非常容易描述。其中最重要和最全面的是 ISO/OSI（国际标准化组织/开放系统互连）协议堆栈，它是许多存储器和数据通信接口和协议的理论模型。每个协议在实现细节上各有不同，但是总体思想是一致的：协议接口的每一层都只与它本身相邻的协议层相连接。不允许跳过某个协议层。协议会话发生在运行在两个不同的机器上的相同协议层之间。这种通信发生的确切方式在 ISO 国际标准中有明确定义。

在 20 世纪 70 年代末期，几乎每台计算机制造商都设计了自己的专用通信协议。协议发明者有时会对这些协议的细节保密，目的是为了确保各自产品的市场优势。通常，A 厂商制造的设备是不能和 B 厂商生产的设备进行通信的，除非在两台设备之间安装协议转换工具（黑盒子）。尽管这样，黑盒子还有可能不像所预期的那样起作用，因为通常在黑盒子造好后，某个公司又会改变某个协议参数。

两个权威的世界性的标准制定组织意识到，这种通天塔（Tower of Babel）式的通信成本越来越昂贵，最终会阻碍信息共享的发展。在 20 世纪 70 年代末 80 年代初，国际标准化组织（ISO）国际组织和国际电报和电话顾问委员会（CCITT）都试图独立建立一个国际标准的通信结构。在 1984 年，这两个组织达成共识，创建了统一的模型，就是现在大家所熟知的 ISO *开放系统互连参考模型*（ISO Open System Interconnect Reference Model，ISO/OSI RM）。这里，开放系统意味着系统的连接对任何一个公司来说都不是私有的。ISO 的文件之所以叫做参考模型，是因为实际上没有商业系统会使用这个模型中所规范的所有特性。但是，对于理解如何在标准模型的环境下把真正的协议和网络设备组合起来，ISO/OSI 模型非常有帮助。

OSI RM 包含 7 个协议层，从第一层的物理介质互连层，一直到第 7 层的应用层。这里必须强调的是，OSI 模型只定义了这 7 个协议层的每层的功能和各层之间的接口。实现的细节不属于模型部分。许多不同的标准化组织，包括美国电气和电子工程师协会（IEEE）、欧洲计算机制造商协会（ECMA）、国际电信联盟的电信标准部门（ITU-T）和 ISO 本身（指该 ISO 模型以外的组织）都提供这些实现过程的细节。在最高层次的实现可以完全由用户定义。

11.4.1 一个比喻

在着手研究 OSI 参考模型的技术之前，我们先用一个比喻来帮助说明分层协议的工作原理。假设你和你姐姐打赌输了，结果是必须花一整天的时间来陪你的外甥 Billy。Billy 是个非常淘气的小孩，一旦不如意就会生气。今天，他决定要到街上 Dumpy 餐馆吃烤牛肉三明治。这种烤牛肉三明治要就着芥末和咸菜一起食用。一样不能多，一样也不能少。

进了餐馆，你安排 Billy 坐好，并从柜台附近的自动取号机拿到一个号码。柜台后有一位收银员在接受顾客订单，另外有一个人在食物加工区处理订单。餐馆里到处都是等着吃午餐的饥饿的工人们。你发现今天的服务好像特别慢。Billy 开始大声嚷饿了，并不停地用小手拍桌子。

虽然你非常想尽快拿到 Billy 的三明治，但还是要等待收银员叫到你们的号码，你必须服从某种协议，抢在别人前面是不可能的。事实上，如果违反协议的话，有可能被餐馆拒绝，反而会使事情变得更糟。

当最终轮到你来到了柜台前时（此时 Billy 的吵闹声正很大），你把订单交给收银员，并为自己叫了一份金枪鱼三明治和薯片。收银员去为你取饮料，并告诉食物处理员准备一份金枪鱼三明治和一份带芥末和咸菜的烤牛肉三明治。尽管厨师在一片嘈杂中能够清楚地听到 Billy 大声嚷嚷中的每一个字，但是她仍然等着收银员告诉她准备什么食物。

因此，不管他叫得多么大声，Billy 都不能跳过餐馆协议中的收银员这一层。在准备三明治之前，食物处理员必须知道订单是合法有效的，而且顾客愿意付钱。她只能通过收银员的告知才能知道这些信息。

三明治做好后，厨师把它们单独放在餐馆包装袋里，并标记好里面是什么食品。收银员取来三明治，与薯片和两罐可乐一起放入一个褐色袋子。她告诉你应该付 6.25 美元，你付给她 10 美元，她找回你 4.75 美元。由于她找钱的数目发生了错误，所以你就得站在柜台前面直到她找回的零钱正确无误后，才回到了自己的桌子。

打开三明治袋子后，Billy 发现他的烤牛肉三明治变成腌牛肉三明治，于是开始新一轮的吵闹。这时，你除了重新取一个号码等着叫号外别无选择。

在找错钱时你拒绝离开柜台可以看作是协议各层之间发生的错误校验。直到接收层从发送层接收到满意的信息时，传输过程才会继续进行。

三明治连同它们的包装相当于 OSI 协议数据单元（Protocol Data Unit，PDU）。一旦数据被上层协议封装起来，那么下层协议就不会对它的内容进行检查。无论你还是收银员都不会打开三明治的包装。而且当收银员将你订的其他物品一起放进那个褐色的袋中时，她也是做好了另一个 PDU。因为你订的食品都在袋子中，所以你可以轻松地拿到自己的桌子上。

在 Dumpy 餐馆，你知道如果想吃什么东西的话，不能直接去找厨师，也不能去找另外的一位顾客，或者是去找维修工。要享受午餐服务，只能在取号后去找收银员。从取号机上拿取的号码可以类似于一个 OSI 的服务访问点（Service Access Point，SAP）。只有当出示号码证明你是排队中的下一位时，收银员才会允许你下订单。

11.4.2 OSI 参考模型

OSI 参考模型如图 11-3 所示。从图中可以看出，协议共由 7 层组成。相邻层之间的接口通过服务访问点接入，当协议数据单元经过协议堆栈时，每层协议增加或者移走它自己的头部。在发送端的末尾增加头部，在接收端的末尾移走头部。PDU 的内容在对话每一边的对等层之间创建一个会话。这个会话就是它们的协议。

上面已经用一个比喻解释了 PDU 和 OSI 的思想，下面我们再给出另一个比喻进一步解释 OSI 的参考模型。在这个比喻中，假设我们自己管理自己的超级菜汤和美味茶点生意，产品是一些精美的菜汤和茶点，销售给整个北美地区具有独特品味的人士。为了将这些美味食品送到客户手中，你雇用了名叫 Ginger，Lee 和 Pronto（或称 GL&P）的私人运输公司。制造商品的系统和老顾客们的最终消费需要经过一系列的程序过程，这些过程非常类似于 OSI 参考模型的那些层次所执行的过程。我们将在下面的章节中具体介绍。

OSI 物理层

许多不同种类的介质都可以在通信的源端（启动器）和目标端（应答器）之间传输比特数据。启动器和应答器都不需要考虑它们之间的会话是在铜导线、卫星链路还是在光缆上发生。OSI 模型的物理层负责完成传输信号的工作。物理层会从上层数据链路层接收到二进制的数据流，并对这些数据进行编码，然后根据大家所达成的协议和信号标准把这些数据放到通信介质上进行传输。

OSI 物理层的功能可以比作负责将产品送往顾客的 GL&P 运输公司的车辆。当你将包裹放到运输公司后，通常不会在乎包裹是通过火车、卡车、飞机还是轮船运送到目的地，你所关心的是只要包裹能够在合理的时间内完整地到达目的地。途中包裹处理人员通常也并不关心包裹中的内容，而他们只是关心盒子上面的地址（有时，他们甚至都没有注意到盒子上的易碎物品的提示——fragile）。与货物公司运送盒子类似，OSI 物理层运送的是传输数据帧，有时我们将这些数据帧称为*物理协议数据单元*（physical protocol data unites），或者简称为*物理 PDU*。每个物理协议数据单元都携带一个地址和带有包围 PDU 的*有效负载*（payload，或内容）的分界符的信号模式。

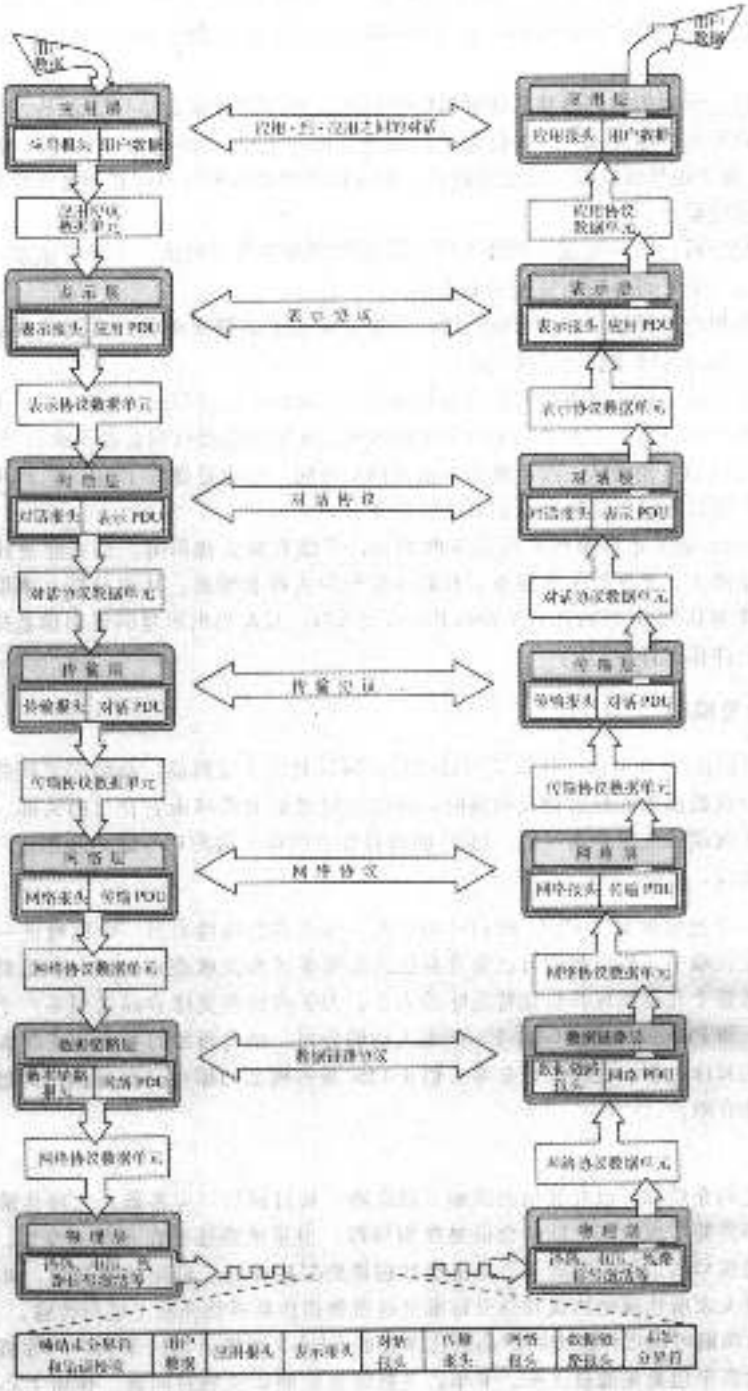


图 11-3 OSI 参考模型：接口垂直操作，协议水平操作

OSI 数据链路层

当送递包裹时，将物品放到一个合适的箱子里，写上送货地址的行为就相当于 OSI 数据链路层的功能。数据链路层负责将信息数据字节组织成一些大小适合于沿着物理介质传输的数据帧。如果你准

备运送 50 公斤的菜汤和茶点，而 GL&P 公司规定一个包装的重量不能超过 40 公斤，那么你需要至少两个单独的盒子来运送物品。数据链路层所完成的就是同样的工作。发送端的数据链路层会与接收端的数据链路层协商传输帧的大小和速率。

帧传输的定时称为流控制（flow control）。如果数据帧的发送速度太快，接收方的缓冲器将会发生溢出，导致数据帧的丢失。如果数据帧的发送速度不够快，接收器就会超时并断开连接。在这两种情况下，如果接收方不能在规定时间间隔内确认发送的数据包，数据链路层会自动监测到这个问题。如果没有收到这种确认信号，发送方就会重新发送这个数据包。

OSI 网络层

假定你告诉 GL&P 公司，送递这些包裹时要经过新泽西州（New Jersey）的 Newark，原因是中转站纽约市（New York）总是非常拥挤，使得包裹不能及时到达。换一种说法，如果你告诉送货人运送包裹所走的路径，那么你就在履行 OSI 模型的网络层所要完成的相同功能。然而，在费城终点站的包裹处理员如果知道了纽约市的塞车情况后，便决定通过 Newark 运送包裹，那么他们也是在执行网络层的功能。这种类型的局部决定对于每个大型网络的运行都是至关重要的。大部分网络结构都非常复杂，要每个终端节点计算机了解到达每个目标节点的各种可能的路径是不可能的，因此网络层的功能要遍及整个系统传播。

在发送端计算机中，网络层只是将来自传输层的地址信息添加到协议数据单元（PDU）上，然后将 PDU 传送到数据链路层。当经过各个中间（intermediate）节点传输 PDU 时，网络层是在执行其最重要和最复杂的任务，而这些中间节点在网络中的作用就像货运中转站。网络层不但负责建立路由路径，而且要确保 PDU 的大小与所有位于发送端和接收端之间的设备兼容。

OSI 传输层

现在假设包裹的目的地是加拿大魁北克（Quebec）的一个食品分销商店。在包裹到达后，运送职员会打开包裹以证实商品在传输过程中没有损坏。她会打开每个盒子，检查一下是否有压扁的罐头和撕裂的茶叶盒。她并不关心这些汤是否太咸了，或者是茶点是否太酸了。她所关心的仅仅是到达后的商品的完整性和完好性。一旦包裹通过了检查，这个职员会在 GL&P 公司返回给你的收据上签字，让你知道商品已经安全到达了目的地。

类似地，OSI 传输层为协议堆栈中它上面的各个层次提供质量保证功能。OSI 传输层也通过它和连接的另一端的传输层之间的握手协议，分担另一层的端到端确认和错误校正任务。传输层是能够了解网络或协议的 OSI 模型的最低层。一旦传输层将来自会话层 PDU 的协议信息剥离，会话层就可以安全地认为在 PDU 中没有发生网络错误。

OSI 会话层

ISO 的会话层负责仲裁两个通信节点之间的对话，在需要时开发和关闭对话。会话层可以控制对话方向和方式。这种通信方式可以是半双工（half-duplex，每次只能沿一个方向进行），也可以是全双工（full-duplex，同时双向进行）。如果通信方式是半双工的，会话层会决定由哪个节点来控制线路。在文件传输过程中，会话层还提供恢复检查点。如果在接收正常的情况下，在各个检查点（checkpoint）每次都会发布一个确认信息的数据包，或者是数据块。如果在一个大型文件的传输过程中发生了一个错误，会话层将从最后一个检查点重新发送所有的数据。如果没有检查点的处理过程，则需要重新发送整个文件。

如果运送职员注意到包裹在传输过程中受到损坏，她将会通知（以及 GL&P 公司）物品没有完整地到达目的地，并且会要求你再次发送物品。如果损坏的包裹是 50kg 中 10kg 的那个盒子，那么你只要替换 10kg 的盒子里的物品就行了，另外 40kg 的商品无需替换可以继续送到顾客手中。

OSI 表示层

如果定购你的送汤和茶点的顾客居住在魁北克的法语商业区，而装汤的罐头盒上的标签是用英文

写的，那你该怎么办呢？当然，你想把汤卖给那些说法语的朋友，所以希望汤罐头的包装上有双语标签。假如魁北克食品分销商店的职员为你完成这些双语标签，那么他们所做的工作正是 OSI 模型中表示层所要做的事情。

表示层为它上面的应用层提供高级数据解释服务。例如，假设某个网络节点是一台 IBM 公司的 z 系列服务器，这台服务器使用 EBCDIC 代码存储和传输数据。主机服务器需要向一台刚刚发出请求的基于 ASCII 代码的微型计算机发送一些数据。在各自系统上的表示层会决定由哪台机器执行 EBCDIC 和 ASCII 之间的代码转换。这两台机器都可以同等效率地进行这种转换工作。值得注意的是：主机系统是向它自己的应用层发送 EBCDIC 代码，而客户机上的应用层则是从协议堆栈中下面的表示层中接收 ASCII 代码。在通信会话期间，如果我们使用加密码或者进行某种数据压缩，那么表示层的服务同样会起到重要的作用。

OSI 应用层

OSI 的应用层负责为通信的一端用户，以及使用系统资源（程序和数据文件）另一端的接口界面处的用户提供有意义的信息和服务。应用层还提供一组程序，需要时可以供用户调用。如果应用层提供的应用程序不能完成这项工作，那么用户可以自己编写程序。关于通信方面，这些应用程序唯一需要做的是向表示层的服务访问点发送消息，较低的层次会负责这些任务中较困难的部分。

为了享受美味的超级汤料，所有法国裔的加拿大美食家们只需打开罐头、加热、然后食用。因为 GL&P 公司和当地的食品分销商已经做了所有的其他事情，使得放到加拿大的餐桌上汤鲜美可口就像刚从自己家中厨房里端出来的一样。

11.5 网络协议 2：TCP/IP 网络结构

当 OSI 和 CCITT 为协议堆栈的改进争论不休时，TCP/IP 迅速传播很快地遍及全球。凭借它在学术和科学通信领域的声誉，TCP/IP 迅速成为了事实上的全球数据通信标准。

尽管 TCP/IP 的出发点并非如此，但是它现在是一个大家所偏爱和非常有效的协议堆栈。TCP/IP 有三层结构，能够映射到 OSI 模型的 7 层结构中的第 5 层上面。TCP/IP 的三层协议堆栈如图 11-4 所示。因为协议中的 IP 层可以和 OSI 中的数据链路层和物理层松散地进行耦合，所以 TCP/IP 能够适用于任何类型的网络，甚至适用于单一会话中的不同网络类型。它唯一的要求是所有参与的网络必须至少运行版本 4 以上的网际协议（Internet Protocol，IPv4）。

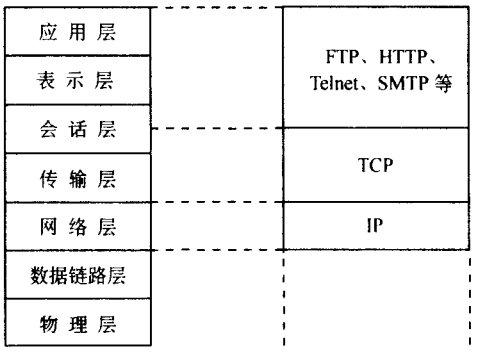


图 11-4 TCP/IP 协议堆栈和 OSI 协议堆栈

目前使用的 IP 协议有两个版本：IPv4 和 IPv6。IPv6 是针对 IPv4 的许多限制而改进的。尽管 IPv6 有许多优点，但是由于大量的机器安装了 IPv4，所以 IPv4 仍然可以应用许多年。IPv4 和 IPv6 之间的一些主要区别将在第 11.5.5 节中讨论。这里，我们首先详细介绍 IPv4。

11.5.1 IPv4 网际协议层

TCP/IP 协议堆栈的网际协议（IP）层，在本质上与 OSI 参考模型的网络层和数据链路层的功能相同。它把 TCP 数据包分成数据报（datagram）协议数据单元（PDU），然后再加上把数据报传送到目的地需要的路由路径的信息。数据报的概念是 ARPAnet 网络发展的基础，现在，ARPAnet 网就是因特网。数据报可以在没有网络管理员的干预下采用任何可用的路由路径。例如，参见如图 11-5 所示的网络。如果中间节点 X 拥塞或者失效，则中间节点 Y 能够通过节点 Z 路由数据，直到节点 X 恢复可

以全速支持路由。路由器是网络中最关键的部件，研究专家们一直在寻找各种方法来提高它们的效率和性能。在第 11.6.7 节中，我们将详细介绍路由器。

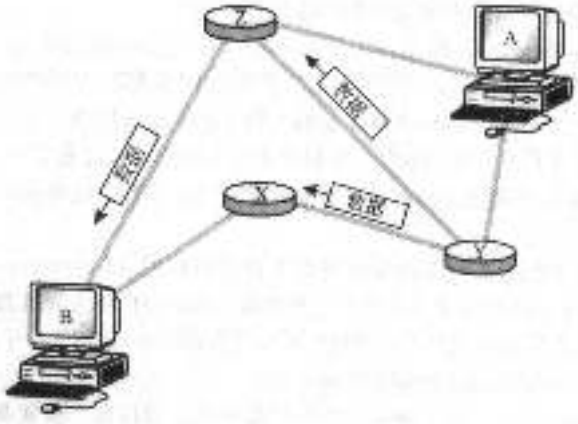
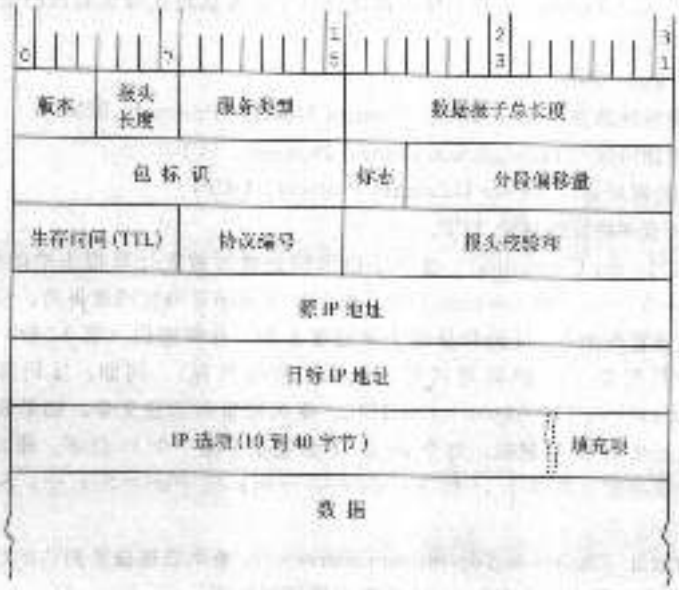


图 11-5 IP 协议中数据报的路由路径

组成 TCP/IP 协议数据单元包含的二进制数据字节称为 8 位字节（octets）。这是因为在当时设计 ARPAnet 协议时，字节（byte）一词被认为是 IBM 主机使用的 9 位位组的专用术语。大多数有关 TCP/IP 的文献中都使用 8 位字节（octets）。但是为了清楚起见，我们这里还是使用字节（byte）。 ■

IPv4 数据报的报头



每个 IPv4 的数据报至少包含 40 个字节，每个字节又包含一个 24 字节的报头（又称为标题），如上图所示。水平行表示 32 位字。仔细观察上图会发现，服务类型域占用了从第 8 位开始直到第 15 位的位置，而标识符域占据了报头位置的第 32 位到第 47 位。报头的最后一个域是填充域，填充域是为了确保紧跟在报头后面的数据可以从整 32 位的边界开始。填充域中总是放置一些数字 0。IPv4 报头的其他部分（域）分别为：

- 版本（Version）：指示正在使用的 IP 协议的版本。版本号会在路由路径上告诉所有硬件有关数据报长度和报头域中的内容。对 IPv4 来说，这个域的值总是 0100（因为 $0100_2=4_{10}$）。
- 报头长度（Header Length）：用 32 位字的方式给出报头的长度。IP 报头的长度大小是可变的，取决于 IP 选项域的值，但是正确的报头的最小的值是 5。
- 服务类型（Type of Service）：控制由中间节点给予数据报的优先级别。数值的范围从“常规”（routine）000 到“关键”（critical）101。网络控制的数据报用 110 和 111 来指示。
- 数据报总长（Total Length of Datagram）：给出整个数据报长度的字节数。从上面的布局图可以看出，为这个域保留了 2 字节。因此，IP 数据报允许的最大长度是 $2^{16}-1$，即 65 535 字节。
- 包标识（Packet ID）：为网络上的每个数据报分配一个系列号。这种主机 ID 和包 ID 的组合可以唯一识别所有 IP 数据报。
- 标志（Flag）：指定数据报是否可以通过中间节点进行分段（fragment，分割成一些更小的数据报）。IP 网络必须可以处理至少 576 字节的数据报。大部分网络可以处理长度大约为 8KB 的数据包。例如，如果标志位的设置为“不分段”（Don't fragment），那么一个 8KB 的数据报就不能通过一个只能处理 2KB 数据包的网络上路由传播。
- 分段偏移量（Fragment Offset）：指定某个数据报中的分段位置。也就是说，分段偏移量可以告诉我们分段属于数据报的哪一部分。
- 生存时间（Time to Live，TTL）：生存时间最初的目的是用来测量数据报可以保持有效时间的秒钟数。数据报在一个路由循环过程中应该能够被节点捕捉。在数据报出现拥塞问题之前，TTL（理论上）将会终止。实际上，每次数据报经过一个中间网络节点，TTL 都会递减。因此，TTL 字段并不能真正测量数据报可以生存的秒钟数。但是，TTL 可以计算数据报到达目的地前的跳跃数（简称跳数，the number of hops）。
- 协议编号（Protocol Number）：指示哪个高级协议正在发送跟在报头后面的数据。这个域中的几个重要的值如下：

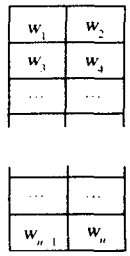
  0＝预留（Reserved）

  1＝因特网控制消息协议（Internet Control Message Protocol，ICMP）

  6＝传输控制协议（Transmission Control Protocol，TCP）

  17＝用户数据报协议（User Datagram Protocol，UDP）

  在 11.5.3 节中将详细讨论 TCP。
- 报头校验和（Header Checksum）：这个字段的值是通过首先计算报头中的所有 16 位字的反码（又称为 1 的补码，one's complement）和，然后取求和结果的反码求得的。校验字段本身最初设置均为 0。反码和是两个字的算术和，并将进位（第 17 位）加到求和结果的最低位。（读者可以复习 2.4.2 节的内容）。例如：反码算术 $11110011+10011010=110001101=10001110$。报头校验和的意义是，如果我们有一个如下图形式的 IP 数据报，每个 w_i 在 IP 数据报中是一个 16 位字。那么完整的求校验和是每次计算两个 16 位字：$w_1+w_2=S_1$；$S_1+w_3=S_2$；…；$S_k+w_{k-2}=S_{k+1}$。



- 源地址和目的地址（Source and destination addresses）：表示数据报要到达的目的地址。在第 11.5.2 节中将会对这个 32 位字段做详细介绍。
- IP 选项（IP Options）：提供某些诊断信息和路由控制。IP 选项是可选项。 ■

11.5.2 IPv4 遇到的困难

在 IP 报头中为每个字段分配的字节数目反映了 IP 设计的技术时代。如果回到 ARPAnet 网络的年代，没有人能够想像网络会如何发展，甚至也没有人想像网络可以应用于普通的民用。

由于当今最慢的网络也要比 20 世纪 60 年代最快的网络快，所以将 IP 数据包的长度限制为 65536 个字节已经成为一个问题。对于某些网络设备来说，现在的数据包的移动速度实在是太快了，以至于无法保证在中间节点之间数据包可以完整无损。如果以千兆比特（GB）的速度传输，65535 字节的 IP 数据报通过某个点的时间还不到一毫秒。

到目前为止，IPv4 报头所遇到最严重的问题是有关地址的问题。每台主机和路由器在整个网络中必须有一个独一无二的地址。为了确保不会出现网络节点的重复，主机 ID 由一个权威机构来管理，也就是因特网域名和编号分配组织（Internet Corporation for Assigned Names and Numbers，ICANN，或称为因特网域名管理中心）。ICANN 负责跟踪记录由区域性权威机构分配或指定的所有 IP 地址。ICANN 还要协调协议中使用的各种参数值，以便使每个人都了解各个参数值会引起什么样的网络行为。

从 IP 报头的分配工具条可以看出，大约可以有 2^{32} 或 43 亿个主机标识 ID。人们可以合理地认为主机的地址数目是足够多的，其实不然。事实上，问题在于这些地址并不能像序列号一样连续地分配给下一个想要地址的人。实际上情况要复杂得多。

IP 允许有三种类型的网络，分别指定为 A 类、B 类和 C 类。这些网络类型通过每种类型直接支持的节点（称为主机，host）的数目的不同来区分。A 类网络能够支持的主机数目最多，C 类则最少。

IP 地址中的前三位表示网络的类型。A 类网络地址总是是从 0 开始，B 类网络是从 10 开始，而 C 类网络则是从 110 开始的。地址中剩余的位用作为网络编号和在该网络编号中的主机标识 ID，如图 11-6 所示。

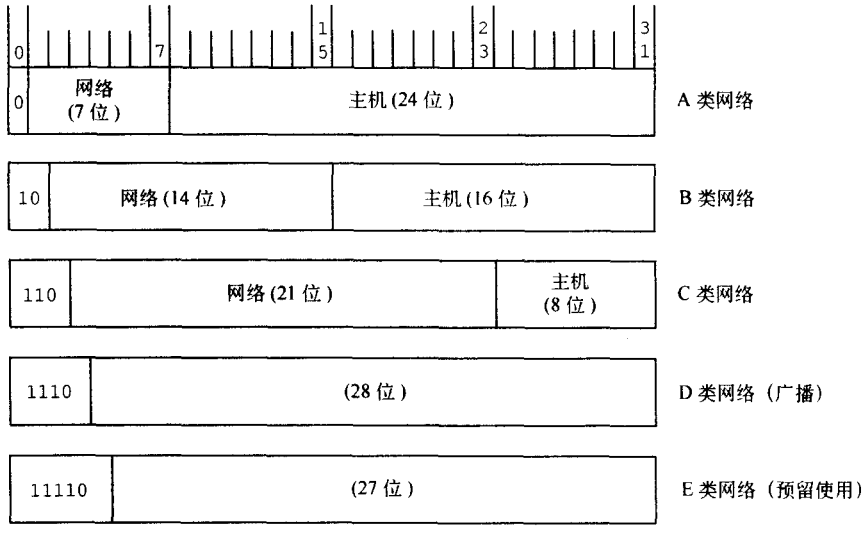


图 11-6 IP 地址分类

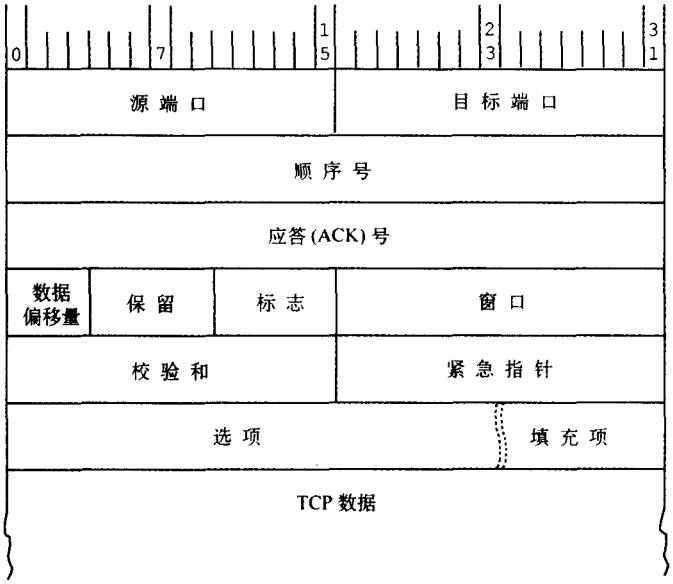
IP 地址为 32 位二进制数，通常用带点的十进制表示法来表示，例如，18.7.21.69，或者 146.186.157.6。每个十进制数代表 8 位的二进制信息，因此它们可以是位于 0 和 255 之间的一个十进制数值。127.X.X.X 是一个 A 类网络，但是保留用作回送测试（loopback testing），检查运行在主机上的 TCP/IP 协议过程。在回送测试过程中没有数据报进入网络。0.0.0.0 网络地址通常预留给网络中的默认路由使用。

考虑到预留的网络 0 和 127，实际上只可以使用 7 位网络字段来定义 126 个 A 类网络。A 类网络是所有网络中最大的，每个 A 类网络大约能够支撑 1.67×10^7 个节点。尽管 A 类网络不可能需要所有的 1.6×10^7 个可用地址，但是从 1.0.0.0 到 126.255.255.255 的 A 类地址很久以前就已经分配给了像 MIT 和 XEROX 公司这样早期网络使用者。而且，所有的 16382 个 B 类网络 ID（从 128.0.0.0 到

191.255.255.255）也已经分配完毕。每个 B 类网络可以包含 65534 个唯一的节点地址。因为很少有组织需要超过 100000 个地址，所以人们的下一个选择是 C 类网络，每个 C 类网络只有 256 个地址，C 类网络空间的地址分配是从 192.0.0.0 到 233.255.255.255。显然，这甚至远远不能够满足一个中等规模的公司或者研究机构的需求。因此，许多网络不能获得连续的 IP 地址块，而导致网络中的每个节点都可能有自己的地址。于是，人们设计了许多非常精巧的工作区来处理这个问题，但是这个问题的最终解决办法是要重新设计整个 IP 地址的结构。在第 11.5.6 节中，我们将讨论这种新的地址结构。还有 D 类网络和 E 类网络存在，但是它们根本不是网络，而只是一些预留地址组。D 类地址是从 224 到 240，是由一些共享某个相同特性的主机群用作多信道广播。E 类地址是从 241 到 248，是为以后使用预留的。

除了地址空间最终会被耗尽外，IPv4 中还存在其他一些问题。网络最初的设计者并没有预料到网络的迅速发展和由于地址分类方案而导致的路由问题。通常，在一个主干网络的路由器的*路由表*（routing table）中有超过 70000 个路由路径选择。目前 IPv4 的路由的基础结构需要进行修改，以减少路由器必须存储路由路径的数目。由于使用了高速缓存存储器，比较大的路由器存储器会造成路由信息的检索非常慢。同样，在 IP 层次上也必须有一定的安全保障措施。目前，正在为 IP 层定义一个称为 IPSec（*因特网协议安全*，Internet Protocol Security）的协议。然而，这个协议是可选的，目前尚未标准化和广泛采用。 ■

TCP 分段格式



TCP 分段（也称为报文段）的格式如上图所示。图的上方显示的数字是每个域所覆盖的范围。水平行代表 32 位字。各个字段（域）的定义如下：

- 源端口和目标端口（Source and Desitination Ports）：用来指示运行在 TCP 上方的应用程序的接口。TCP 通过端口号来分辨这些应用程序。
- 顺序号（Sequence Number）：指出有效载荷中数据第一个字节的顺序号。TCP 会给每个传输的字节分配一个顺序号。如果有 100 个数据字节，每次发送 10 个字节，那么第一个分段的顺序号可能是 0，第二个是 10，第三个是 20，依次类推。当然，这个起始的顺序号可以不必是 0，只要这个数字在发送器和接收器之间是唯一的。
- 应答号（Acknowledgement Number）：包含接收器所期望的下一个数据的顺序号。TCP 使用这个值来决定在传播途中是否丢失了任何数据报。

- 数据偏移量（Data Offset）：在报头中所包含的 32 位字的数目，或者等价地说，是在分段中数据开始的字的相对位置。也称作报头长度。
- 保留（Reserved）：直到有人对这 6 位提出一个好的使用方案，否则这 6 位一定要设置为 0。
- 标志（Flag）：包含用于大多数协议管理的 6 位字段。当这 6 位域的值不为 0 时，就将它们设置为“真（true）”。TCP 的各种标志和含义如下：

  URG（紧急比特）：指示这个分段中存在的紧急数据。紧急指针字段会指示跟随在紧急信息之后的第一个字节的位置。

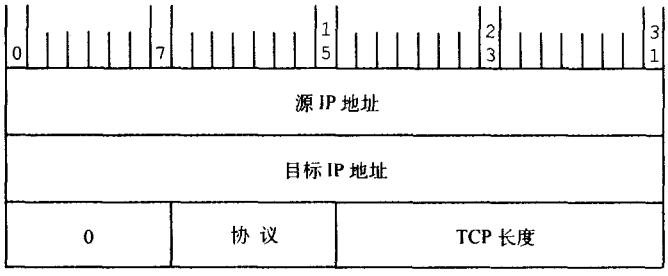
  ACK（确认比特）：指示应答号域中是否包含重要信息。

  PSH（急迫比特）：告诉在连接中的所有 TCP 程序进程要清空它们的缓冲处理器，即将数据“压入（push）”接收器中。当有效载荷中存在紧急数据时，应该设置这个标志。

  RST（复位比特）：重启一个连接。通常，它会强制所有接收到的包进行有效性验证，并且把接收器置回到“收听更多数据（listen for more data）”的状态。

  SYN（同步比特）：指示分段的目的是为了同步顺序号。如果发送器传输［SYN，SEQ # = x］，则接下来它应该从接收器收到［ACK，SEQ # = x+1］。在两个节点之间建立一个连接时，它们双方会相互交换彼此的初始顺序号。

  FIN（终止比特）：这是一个“完成（finish）”标志。它让接收器知道发送器已经完成了传输，开始关闭连接过程。
- 窗口（Window）：允许两个节点通过陈述各自在任意分段中希望接受字节数来定义它们各自的数据窗口的大小。例如，如果发送方传输的字节数是从 0 到 1023，而接收方在应答（ACK #）域中应答字节数是 1024 且窗口值是 512，则发送方应该发送数据字节数 1024 到 1535 进行回复。可能会发生接收方的缓冲器被填满的情况，这时接收方会要求发送方减慢发送速率，以便接收方可以跟得上传输的速率。值得注意的是：如果接收方的应用层运行得非常慢，例如每次只能从缓冲器拉入 1 个或 2 个字节的数据，则在接收器中运行的 TCP 进程应该等待，直到应用层缓冲器空，有足够的空间判断是否要发送另外一个数据分段。如果接收方发送的窗口的大小是 0，这个结果表示需要确认应答号所指示的所有字节，并且停止进一步的数据传输，直到具有一个非 0 窗口大小的相同的应答号被再次发送过来。
- 校验和（Checksum）：这个域包含了求在 TCP 分段中的各个字段以及一个如下的 IP 伪报头（pseudoheader）的校验和，数据填充域和校验和域本身除外：



正如前面对 IP 校验和的解释一样，TCP 的校验和是求 TCP 报头和 TCP 分段正文中的所有 16 位字的 16 位反码（1 的补码）。

- 紧急指针（Urgent Pointer）：指向紧跟在紧急数据后的第一个字节。这个字段（域）只有当设置了 URG 标志时才有意义。
- 选项（Option）：在其他各种事务中，选项主要关注窗口大小的协商问题和是否使用选择性的应答（Selective acknowledgment，SACK）。如果在传输过程中某个位置发生了一个分段丢失，那

么SACK允许在一个窗口中重新传输某些特殊的TCP分段，而不是要求重新传输整个窗口。在讨论了TCP流的控制后，读者将会对这个概念理解得很清楚。 ■

11.5.3 TCP

IP协议的唯一目的是要在网络中正确地路由传输数据报。我们可以将IP看作一个运送包裹的邮递员，这个邮递员并不关心要运送的包裹的内容，或包裹运送的顺序。传输控制协议（Transmission Control Protocol，TCP）是IP服务的使用过程（用户），因此TCP会真正关心这些内容，以及许多其他事情。

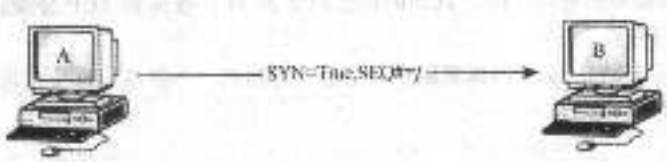
在两个TCP进程之间的协议连接要比IP层中的协议连接更加复杂。IP只是基于报头信息而单独接受或拒绝数据报，而TCP则会与一个运行在某个远程系统上的TCP进程建立一个会话，我们称为建立一个连接（connection）。一个TCP连接与电话通话非常相似，它有自己的协议"礼节(etiquett)"。会话开始时，TCP会开放一个服务访问点（SAP），SAP就是在TCP上层运行的应用程序。在TCP中，SAP是一个称为端口（port）的数字值。端口号、主机ID和协议指定名组合起来就变成了一个套接字（或称为接插件，socket）。套接字对于运行在TCP上层的应用程序来说，在逻辑上相当于一个文件名（或称为句柄，handle）。这个应用程序不是通过它的键盘文件名访问各种数据，而是通过这个套接字来使用TCP读取数据。从0到1023的端口号称为"专用（well-known）"端口号，因为这些端口都是为某些特殊的TCP应用程序所预留的。例如，TCP/IP文件传输协议（File Transmission Protocol，FTP）的应用程序使用端口号20和21。而远程终端协议（Telnet terminal ProtocoD则使用端口号23。编号从1024到65535的端口号为用户自定义的执行程序所用。

TCP要确保它提供给应用程序的数据流是完整的，并且具有正确的顺序和没有重复的数据。TCP还要确保它所发送的各个分段（segment，指一些带有报头的数据包）不能太快，以免这些数据报淹没中间节点或接收器。这样，TCP可以对网络传输过程中发生的各个不规则事件进行补偿校正处理。一个TCP分段的报头要求至少有20个字节。数据有效载荷的长度是可选的。一个报文段包括报头的长度最多为65515字节，这样整个报文段就可以适合于IP的有效载荷。如果需要或者某个中间节点请求的话，IP也可以分割成一个的TCP分段。

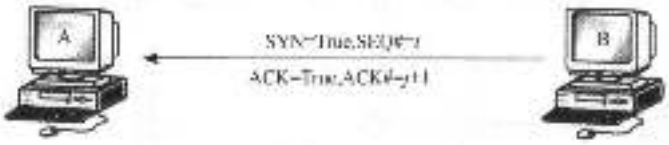
TCP提供了一种可靠的、面向连接的服务。面向连接（connection oriented）就是在主机可以进行任何信息交换（类似于电话呼叫）之前必须先建立这个连接。TCP通过分配给每个报文段的顺序号来保证连接的可靠性。TCP同时使用应答机制来验证所收到的分段内容，而且应答信号必须在规定的时间周期内发送和接收。如果没有应答信号返回，数据就要重新传输。在下一节中，我们会简单地介绍TCP的工作原理。

11.5.4 TCP的工作原理

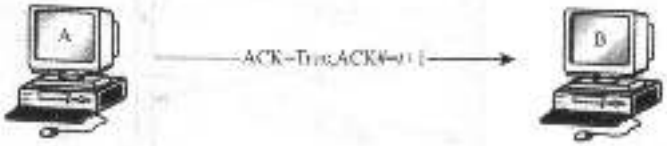
我们如何才能组合TCP的所有内容，在独立系统上运行的两个或更多的TCP程序进程之间建立一个可靠有序和没有错误的连接呢？成功的通信需要有三个基本过程，第一是启动连接，第二是交换数据，第三是关断连接。首先，启动系统（initiator，这里将它称作A）将一个"开启（open）"的初始基元信号发送到运行在远程系统B上的TCP进程。我们假定B正在监听"开启"请求。这种"开启"初始基元的形式如下：



如果B已经准备好接收来自发送方的一个TCP连接，那么B会做如下回答：



A 将做出如下的响应：



现在，A 和 B 已经相互进行了应答并对起始的顺序号进行了同步。A 的下一个顺序号是 $i+2$，而 B 的下一个顺序号是 $j+2$。像这样的协议交换常常称为三次握手（three way handshakes）协议。大多数网络文献都会给出这种交换过程的示意图，如图 11-7 所示。

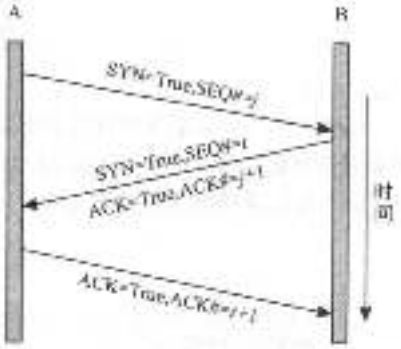


图 11-7 TCP 三次握手协议过程

在 A 和 B 之间的连接建立起来后，接下来它们会开始协商窗口大小并为已经建立的连接设置一些其他选项。窗口会告诉发送方在两次应答之间发送多少字节的数据。例如，假定 A 和 B 协商窗口的大小为 500 字节和一个数据有效载荷的大小为 100 字节，双方都同意不使用选择性的应答方式（这一点将在后面讨论）。图 11-8 表示了 TCP 如何在两个主机之间管理数据流。需要注意的是，如果某个报文段丢失会发生什么情况：那就是整个窗口需要重新传输，即使后续传送的各个报文段都没有错误。

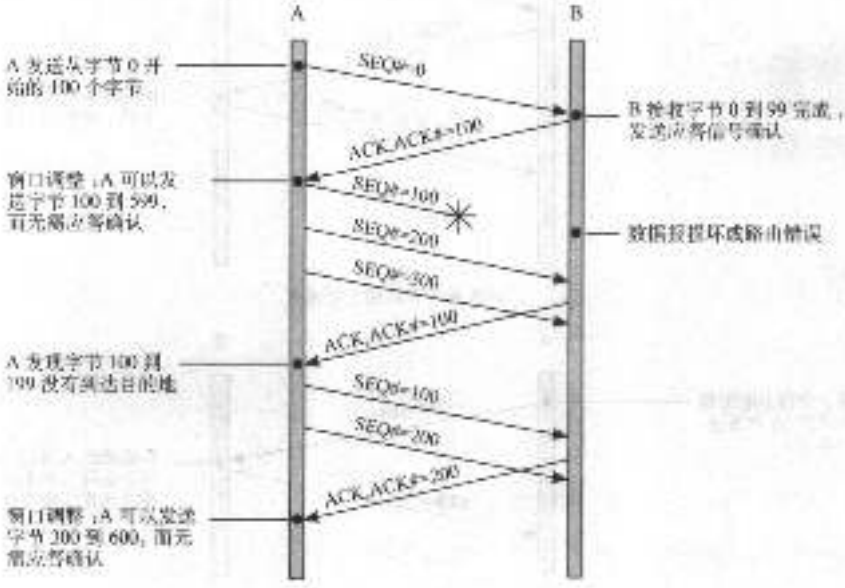


图 11-8 某个报文段丢失时的 TCP 数据传输过程

但是，如果某个应答信号丢失，那么接下来的应答可以避免发生数据的重传过程，如图 11-9 所

示。当然，必须及时送出应答信号，以防止发生“超时（timeout）”重传。

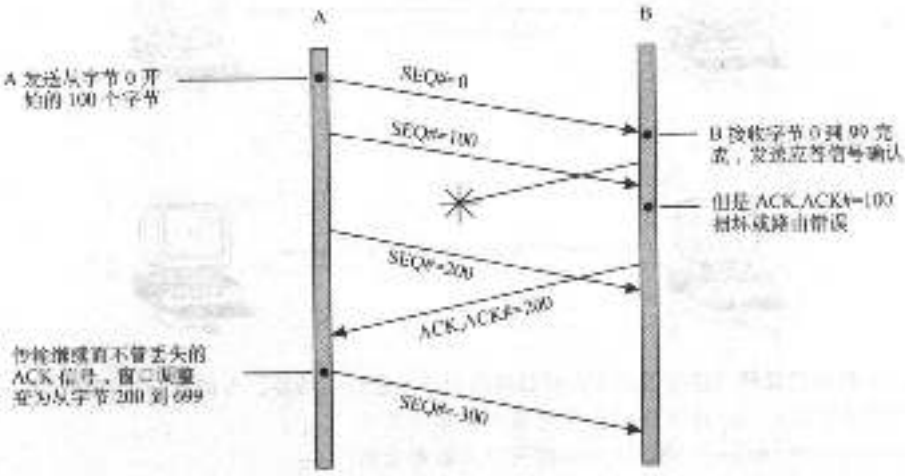
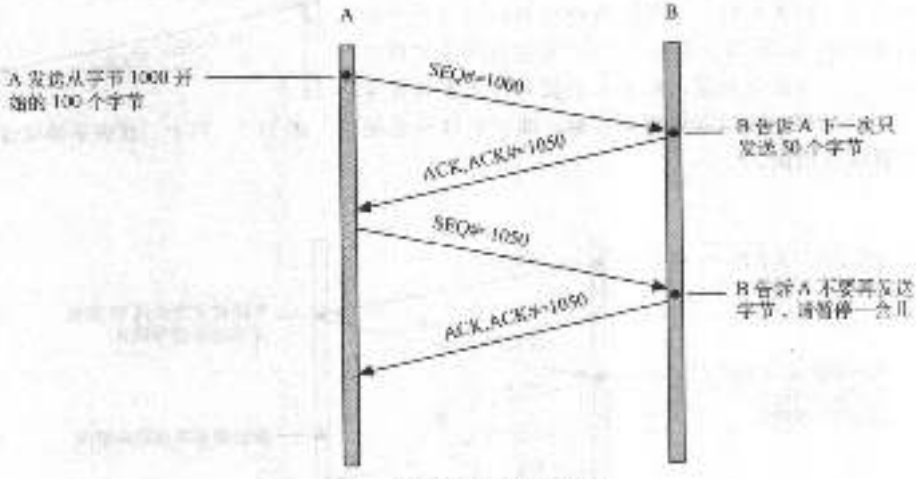
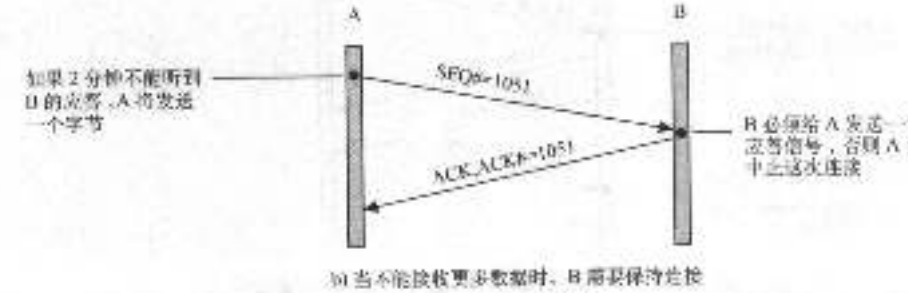


图 11-9 一个应答信号丢失

利用应答号，接收方也可以请求发送方减慢或终止传输。特别是当接收方缓冲器变得太满时，这样做是非常必要的。这种方法的工作原理如图 11-10a 所示。图 11-10b 说明了当 B 不能接收更多的数据时，B 是如何继续保持连接畅通的。



a) B 通知 A 减慢发送速度



b) 当不能接收更多数据时，B 需要保持连接

图 11-10 TCP 流控制

当完成数据交换后，一个或两个 TCP 的程序进程将会终止这个连接。连接的其中一方，假设 A，可以通过发送一个 FIN 标志设置为真的报文段，来向连接的另一方 B 指示数据交换任务已经完成。这样就可以有效地关闭从 A 到 B 的连接。但是，B 会继续连接它的会话端，直到没有数据发送为止。一旦 B 完成数据交换，它也会传送设置有 FIN 标志的报文段。如果 A 应答 B 的 FIN，那么两端的连接都会被终止。如果 B 在经过超时还没有接收到应答信号，B 将自动终止连接。

由于 TCP 的规则并不十分严格和苛刻，所以 TCP 允许发送方和接收方协商一个超时的时间周期。对于速度比较慢的连接，超时的周期应该设置为一个较大的数值。发送方和接收方也可以同意使用选择性应答。当选择性应答（SACK）有效时，接收方必须对每个数据报都做出应答。换句话说，就是不使用活动窗口。当有错误发生时，SACK 可以节省一些带宽，因为只有没有应答确认的报文段（而不是整个窗口）才需要进行重新传送。但是，如果交换是没有错误的，那么因为需要发送应答报文段而会浪费一些带宽。基于这个原因，只有当接收方没有太多的 TCP 缓冲器空间时，才选择使用 SACK。接收方缓冲器空间越大，它接收无序报文段的“不规则空间（wiggle room）”就会越多。总之，TCP 会竭尽其能为运行在 TCP 上层的应用程序提供一个没有错误和有序的数据流。

11.5.5 IPv6

1994 年，IP 的 B 类地址问题在发展过程中似乎出现了重大危机，这种危机使得网络的爆炸性增长有突然中断的潜在可能性。由于有接近末路的紧迫感，网络工程任务组（IETF）开始商议 IPv4 的后继产品，现在称为 IPv6。在这段时间里，IETF 的参与者发布了许多称为 IPv5 的实验性协议。这些协议版本经过修正和改进，成为现在众所周知的 IPv6。专家预测 IPv6 要到 21 世纪前 10 年的后期才会广泛实行。每天都会有许多的网络应用程序要进行修改以适应 IPv6 的要求。事实上，有些反对人士争论说，IPv6 永远不可能完全配置使用，因为使用 IPv6 协议需要替换许多非常昂贵的硬件，而且因为已经发现了在 IPv4 中固有的最令人烦恼的工作区问题。但是，与这些批评者观点相反，人们相信 IPv6 更有办法解决 B 类地址短缺的问题。事实上，IPv6 可以解决大多数人还没有意识到的许多难以处理的事情。下面，我们会对这些问题加以解释。

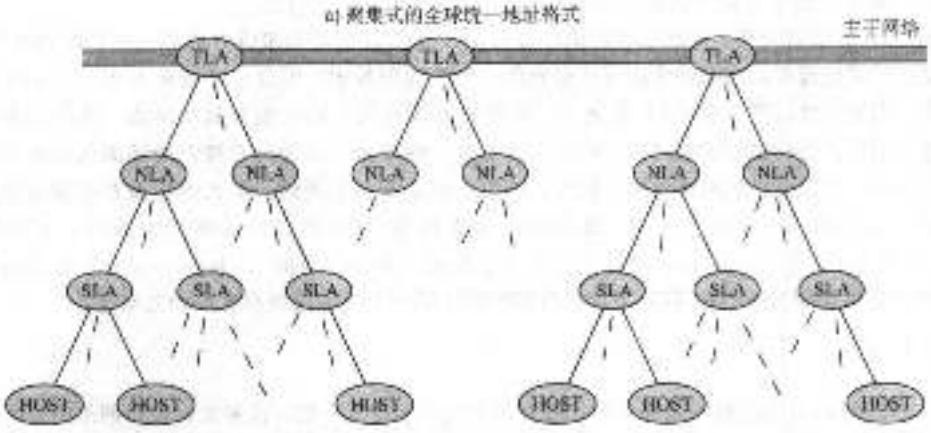
当然，IETF 设计 IPv4 后继者的主要目的是拓展 IP 地址空间，即从当前的 32 位地址扩展为源主机和目标主机的地址都是 128 位。如此庞大的地址空间简直令人难以置信，IPv6 可以产生 2^{128} 个可能的主机地址。具体地说，如果每个 IP 地址分配一个重 28 克（1 盎司）的网卡，那么 2^{128} 个网卡的重量是整个地球的 1.61×10^{15}（千万亿，quadrillion）倍！IPv6 可以提供的地址数量几乎是无穷的。

如此大的地址空间所带来的负面影响就是对这些地址的管理变成了非常关键的因素。如果随意分配这些地址而没有很好地进行组织安排，那么要进行有效的数据包路由传输是不可能的。网络上的每个路由器都要求有超型计算机的存储容量和速度以应对随之而来的路由表数目的激增。为了避免这个问题，IETF 提出一个分层地址的组织结构，并称之为全球统一地址格式（Global Unicast Address Format），如图 11-11a 所示。IPv6 地址的前三位构成一个标志，表示该地址是一个全球统一地址。接下来的 13 位形成所谓的最高层聚集标识码（Top-level Aggregation Identifier，TLA ID）。接下来是 8 位预留位，如果有需要，可以利用这 8 个保留的位对 TLA ID 或者次高层聚集识别码（Next-level Aggregation Identifier，NLA ID）进行扩展。一个 TLA 的实体可能是一个国家或一个主要的全球性的电信公司。而 NLA 的实体则可能是大公司、政府、学术机构、因特网服务提供商（ISP）或者是小规模电信公司。跟在 NLA ID 后面的 16 位是站点聚集标识符（Site-level Aggregation Identifier，SLA ID）。NLA 的实体可以使用这个字段来创建自己的分层结构，允许每个 NLA 实体有 65536 个子网，每个子网可以有 2^{64} 个主机。这种分层结构的示意图如图 11-11b 所示。

初看起来，为每个子网预留 2^{64} 个主机的观点与 IPv4 的网络类型系统一样浪费地址空间。但是，这样大的地址空间对于支持跨国地址自动配置（stateless address autoconfiguration）是完全必要的，跨国地址自动配置是 IPv6 的一种新功能。在无状态地址自动分配中，一个主机会使用固化在它的网络

3位	13位	8位	24位	16位	64位
前缀001	最高层聚集ID	预留	次高层聚集ID	站点层聚集ID	接口ID

a) 聚集式的全球统一地址格式



b) 聚集式的全球统一地址的分层结构

图11-11 聚集式的全球统一地址格式和分层结构

接口卡中的48位地址（即它的MAC地址，将在第11.6.2节中介绍），以及从该主机附近的路由器收到的网络地址信息一起形成主机的完整IP地址。如果在这个过程中没有发生任何其他问题，那么网络中的每台主机都可以在没有网络管理员干涉的情况下自动配置自己的地址信息。这种特性是网络管理员所乐意看到的，特别是当一个实体要更换其网络服务提供商（ISP）或者通信公司时。这里，网络管理员只需要改变路由器的IP地址，无状态地址自动分配将会自动更新网络中每个节点的TLA或者SLA的字段内容。

IPv6地址的书写语法也不同于IPv4地址的书写语法。IPv4地址使用小数点十进制计数法表示，例如146.186.157.6。IPv6地址使用十六进制表示，用冒号隔开，例如：

```
30FA:505A:B210:224C:1114:0327:0904:0225
```

使得IPVA更容易辨认等价的二进制IP地址。

IPv6地址可以缩写，有时0可以省略。如果有16位的一组数字是0000，则可以写成0，或者0全部省略。如果由于省略0而导致有两个以上的连续冒号，那么这些连续的冒号可以缩减为两个冒号。规定在地址格式中只能有一组多于两个的连续的冒号。例如，下面的IPv6地址：

```
30FA:0000:0000:0000:0010:0002:0300
```

可以简写为：

```
30FA:0:0:0:10:2:300
```

或进一步简写为：

```
30FA::10:2:300
```

但是，类似30FA::24D6::12CB这样的地址是无效的。

IETF还建议两种其他的路由方法的改进，实现多播和泛播。多播（或多信道广播，multicasting）是指在网络的某处放上一条消息，然后由多个节点读取。泛播（anycasting）是指任何一个逻辑节点群都是消息的接收者，数据包并没有指定某个特定的接收器。IPv6的这一特性，以及跨国地址自动分配特性，非常利于支持移动设备。移动设备是网络用户中一个越来越重要的部分，特别是国际之间的通

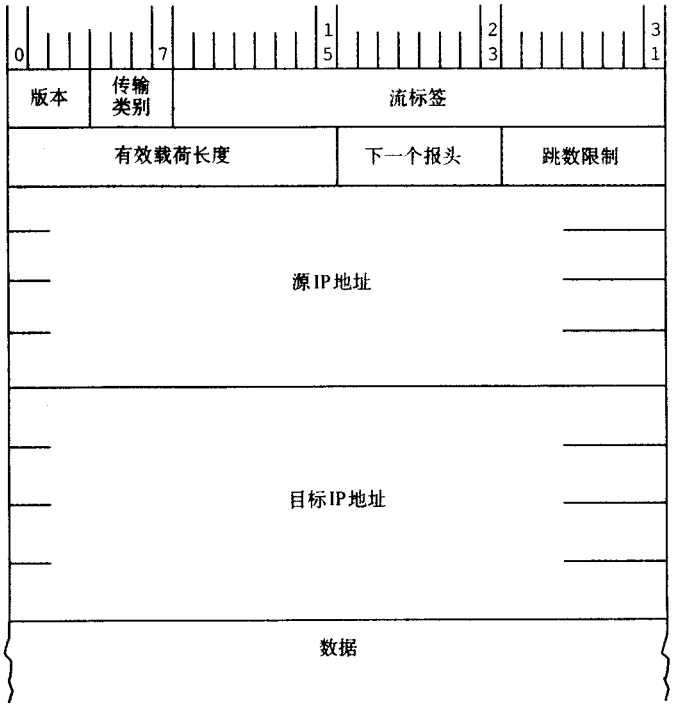
信大多数都是使用无线网络。

正如前面所说，安全是IPv6区别于IPv4的一个重要的领域。所有IPv4的安全特性（IPSec）都是一个“可选（optional）”项，这意味着没有规定必须采取任何类型的安全措施。实际上，大多数的安装都不选安全项。在IPv6中，IPSec是强制性的。IPv6安全改进方法是一种防止地址欺骗（address spoofing）的机制，所谓地址欺骗是指一台主机可以与使用伪造的IP地址的另一台主机进行通信。IP的电子欺骗常常被用来破坏路由器和用来阻止外来入侵的防火墙。IPSec还支持使用加密和其他措施，使不法分子寻找没有授权的信息变得更加困难。

也许IPv6最好特点是提供了一种过渡方案，可以让网络逐渐转向新的格式。当然，支持IPv4是IPv6功能的一个组成部分。使用这两种协议的设备被称作双栈（dual stack）设备，因为它们同时支持IPv4协议堆栈和IPv6协议堆栈。目前，市场上有许多路由器都是双栈设备。可以预期，在不久的将来使用IPv6将成为现实。

相对于IPv4来说，IPv6的优点非常明显：大得多的地址空间，更好和内置的服务质量，以及更好和更有效的路由方案。所以，使用IPv6，只不过是一个时间问题。对IPv6的商务需求和一些必要的应用程序的开发将会促使IPv4向IPv6的转变。虽然替换硬件的开支问题是这种转变过程中的一个重大壁垒，但是技术人员的培训和一些较小的IP设备（例如网络传真机和打印机）的替换也会在整个转换的开支中占一定的份额。随着网络汽车（IP-ready automobile）和许多其他网络设备的出现，IPv4已经不再能够满足许多当前的应用需求了。 ■

当然，有关IPv4最明显的问题是32位地址域。IPv6通过将地址字段扩充到128位来改进IPv4地址受限制的这个缺点。为了使IPv6的报头尽可能小（可以加快路由速度），极少使用的IPv4报头字段都没有包含在IPv6主要报头中。如果需要这些字段的话，IPv6还提供了一个下一个报头（Next Header）的指针。通过这种下一个报头区域，可以想像IPv6能够支持大量的报头域。因此，未来对IP功能的增强所带来的破坏性比从IPv4到IPv6的转变所预示的情况要小得多。IPv6的报头中各个字段解释如下图所示：



- 版本（Version）：总是 0110。
- 传输类别（Traffic Class）：IPv6 最终能够说出实时传输（real-time transmission，例如声音和视频）和时间敏感性较低的数据传输通信（less time-sensitive data transport traffic）之间的区别。这个字段用来区别这两种不同的传输类型。
- 流标签（Flow Label）：这是为仍在改进中的规范准备的另外一个字段。一个数据“流（flow）”就是一个会话，既可以对所有的节点广播，又可以只在两个特定的节点之间启动。流标签字段标识出一个特定的流串，以及中间路由器需要采用与这个流标签字段中的代码一致的方式来路由传送数据包。
- 有效载荷长度（Payload Length）：用字节的形式指出 IPv6 的有效载荷的长度，其中包括附加报头的大小。
- 下一个报头（Next Header）：如果有任何跟在主报头的后面其他报头，这个字段的作用是指示这些报头的类型。如果 IPv6 的协议交换需要比单一报头携带更多的信息，那么这个下一个报头的字段将会提供一个扩展的报头。这些扩展报头将被放置在 IPv6 的报文段的有效载荷中 如果没有 IP 扩展报头，那么这个字段将包含“TCP”的值，是指在有效载荷中第一个报头的数据属于 TCP，而不是属于 IP。一般来说，只有目标节点会检查扩展报头的内容。而中间节点会把它们看作像普通的有效载荷的数据一样进行传递。
- 跳数限制（Hop Limit）：占 16 位，这个字段比 IPv4 大得多，允许有 256 个跳数。在 IPv4 版本中，每个中间节点的路由器会对这个字段的值递减。如果这个值变为 0，该数据包将会被丢弃，并且通过一个 ICMP 信息通知发送方（对 IPv6 而言）。
- 源地址和目标地址（Source and Destination Address）：比 IPv4 中的源地址和目标地址大得多，但是含义是相同的。读者可以回顾一下有关地址格式讨论的内容。 ■

11.6 网络组织结构

通常，计算机网络是根据网络的地理服务范围进行分类的。最小的网络是局域网（local area networks，LAN）。尽管局域网可以包含成千上万个节点，但是它通常在同一栋建筑物使用，或者在彼此非常靠近的一群建筑物中使用。当 LAN 覆盖两个或两个以上的建筑物时，通常叫做校园网（campus network）。LAN 所覆盖的区域（财产）通常也与其本身一样具有相同的所有权（或控制权）。区域网（metropolitan area netwroks，MAN）是覆盖一个城市和周边地区的网络。区域网通常是跨地区的，这些地区的所有权并不属于网络的拥有者。广域网（wide area networks，WAN）是覆盖多个城市或者跨越整个世界的网络。

曾经有一段时间，局域网、区域网和广域网所使用的协议彼此之间互不相同。区域网和广域网通常被设计成高速吞吐量，因为它们要用作为多个慢速局域网的主干系统，或者是因为它们要为远距离的终端用户提供访问大型计算机主机的数据中心的服务。然而，随着网络技术的发展，现在这些网络在速度或协议方面彼此之间已经没有了太大的区别，但是它们的所有权还是不同的。某个人的校园局域网可能是另一个人的区域网。事实上，随着局域网的速度越来越快，并且更易于与广域网技术集成，可以想像区域网的概念最终会完全消失。

本节将讨论局域网、区域网和广域网通用的物理网络设备。先从网络组织的最低层，物理介质层（第 1 层）开始。

11.6.1 物理传输介质

实际上，任何具有传输信号能力的介质都可以支持数据通信。有两种通用的通信介质：导向（guided）传输介质和非导向（unguided）传输介质。非导向传输介质是利用红外线、微波、卫星或无线电载波信号等透过广播频率进行数据传送的。导向传输介质是一些直接连接到每个网络节点的物理

连接器，例如铜线或光缆。

导向传输介质的物理和电气特性，决定了介质在一定频率下可以准确输送信号跨越不同距离的能力。在第7章中，我们曾经提到了远距离传输时信号会衰减（减弱）。距离越长，信号的频率越高，这种衰减越大。铜线中信号的衰减来源于一些电现象的相互作用，其中主要原因是铜导体的内部电阻和当信号传输线路相互之间非常靠近时所产生的电学干扰（电感和电容）。外部电场，例如荧光灯和电动马达等，也能够使铜线上传输的信号发生衰减，甚至失真。总体上来说，阻碍信号准确传输的电现象称为噪声（noise）。信号和噪声的强度都用分贝（dB）来测量。电缆是根据有噪声存在时，不同频率下传输信号的质量来进行评价的。就通信信道而言，这种评价的物理量是信噪比（signal-to-noise rating），信噪比也是利用分贝来测量的：

$$\text{信噪比（dB）} = 10\log_{10}\frac{\text{信号（dB）}}{\text{噪声（dB）}}$$

从技术上来说，一种传输介质的带宽（bandwidth）是传输介质能够传输的信号频率的范围，用赫兹（Hz）表示。传输介质的带宽越大，能够传输的信息越多。在数字通信中，带宽是传输介质的信息传输能力的通称，利用位/秒（b/s）来表示。数字通信中的另外一个重要度量是二进制信号的误码率（bit error rate，BER），误码率是已接收到的出错位数和已接收到的二进制信号的总位数的比值。如果信号的频率超过了线路的信号传送能力，误码率（BER）可能会变得极为严重，以致于连接的设备在错误校正上所花费的时间比正常工作时间还要多。

同轴电缆

同轴电缆曾经是数据通信所选择的传输介质。它传输信号的频率可以达到T赫兹（每秒万亿次），而且信号的衰减很小。目前，同轴电缆主要应用于广播和闭路电视中。同轴电缆也可以为住宅区的网络服务传输信号，这种服务架构于有线电视线路。

同轴电缆的中心部分是一个厚（12～16 gauge）的内导体，外面由电介质（dielectric）的绝缘层所包围。绝缘层的外面包围着一层铜箔屏蔽层，目的是防止额外电磁场的影响。铜箔屏蔽层的本身是用铜丝或钢丝编织层缠绕，可以作为电缆的电学接地。整个电缆最后采用耐久性的塑料层封装起来，如图11-12所示。

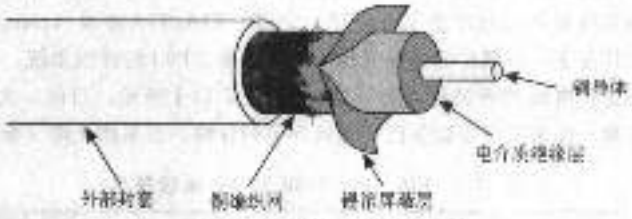


图11-12 同轴电缆的剖面

有限电视服务使用的同轴电缆叫做宽带（broadband）同轴电缆，因为它的传输能力至少是2Mb/s。宽带通信使用多路复用技术的形式提供多数据信道。计算机网络通常使用窄带（narrowband）同轴电缆，它是一个优化的单信道结构，典型的宽带为64Kb/s。

双绞线

两台计算机之间最简单的连接方法是仅仅采用一对铜导线，一根用来发送数据，另一根用来接收数据。当然，两个系统之间的距离越远，信号需要越强，这样可以防止信号在长距离传送过程中衰减消失。同样，系统间的距离也会影响到数据传输的速度。系统相距越远，这种传输线的速度必须越慢，以避免发生过多错误。使用较厚的导体（较小的电线gauge数）能够减少衰减。当然，较厚的电线要比较薄的电线更贵。

除了衰减外，电缆的制造商还需要解决电磁感应（industance）问题。当两根导线完全平行地靠放在一起时，电线中较强的高频信号会在铜导体周围产生（感应）磁场，这种感应磁场会干扰两根平行

电线中的信号传输。

减少导体之间电感应现象的最简单方法是把两根电线绞拧在一起。这里，每直线尺所用的绞线越多，线与线之间彼此干扰所产生的衰减就越少。制造绞线要比制造非绞线要贵，因为每直线尺消耗的电线要多，并且还要仔细控制拧绞的程度。双绞线（twisted pair）电缆，指其中有两对绞线，应用于今天的大部分局域网的连接安装（参见图 11-3）。双绞线有两种类型：屏蔽双绞线和非屏蔽双绞线。更常用的是非屏蔽双绞线。

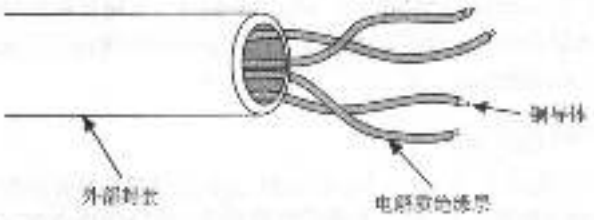


图 11-13 双绞线电缆

屏蔽双绞线主要适用于有大量电干扰存在的环境中。目前的商业环境中充满了各种可以干扰网络信号的电磁辐射源。这些电磁辐射源有的看起来和荧光灯一样温和，但有的像嗡嗡叫的电源变压器一样烦人。任何产生磁场的设备都有干扰网络通信链路的潜在可能性。干扰信号可以影响网络的传输速度，因为较高的信号频率对任何种类的信号失真都会非常敏感。作为防止环境干扰（称之为电磁干扰（electromagnetic interference，EMI）或射频干扰（radio-frequency interference，RFI））的安全措施，安装使用屏蔽双绞线有助于在各种不利的环境中保持网络通信的完整性。

因为屏蔽双绞线的材料成本和安装费用都很高，所以一些专家不太赞成使用屏蔽双绞线。他们指出，使用屏蔽双绞线时如果接地不当，实际上会导致出现更多的问题。特别是屏蔽双绞线可以作为一个天线，实际上会吸引各种信号进向导体。

无论是屏蔽双绞线还是非屏蔽双绞线，作为网络导体都必须具有适合于现在所使用的网络技术的信号输送能力。1991 年，电子工业联合会（EIA）和电信工业协会（TIA）联合起来为网络电缆连接建立了一个评级系统。评级系统最新的版本是 EIA/TIA 568B。EIA/TIA 分类（category）评级标准规定了在信号不会过多衰减的情况下，电缆传输所能支持的最大频率。ISO 的评级系统，不像 EIA/TIA 的分类系统那样经常使用，把这些电线的等级称为类（class），如表 11-1 所示。目前，大多数局域网所安装的是 5 类或更好的电缆连接。许多网络安装业已完全放弃了铜导线，而采用光缆（参见下一节）。

表 11-1 EIA/TIA－568B 和 ISO 电缆规范

EIA/TIA	ISO	最大干扰
Category 1		语音和 " low speed" 数据（4～9.6 kHz）
Category 2	Class A	1 Mbps 或更少
Category 3	Class B	16 MHz
Category 4	Class C	20 MHz
Category 5	Class D	100 MHz
Category 6	Class E	250 MHz
Category 7	Class F	600 MHz

注意，表 11-1 中列出的电缆等级的信号传送能力，使用的单位为兆赫兹（MHz），它与兆位（Mb）是不同的。在第 2.7 节中读者看到，在任意给定的频率下，传送的位数与网络中所使用的编码方法有关。在 100Mb/s 以下运行的网络可以很方便和经济地使用曼彻斯特编码（Manchester coding）方法；曼彻斯特编码对于每传输一位只要求有两次信号转换。对于运行在 100Mb/s 或者更高速率的网络，则使用不同的编码方案。目前，其中最流行的编码方案是 4B/5B 编码方法，使用 NRZI 信号方式

每 5 个波特传输 4 位信号，如图 11-14 所示。

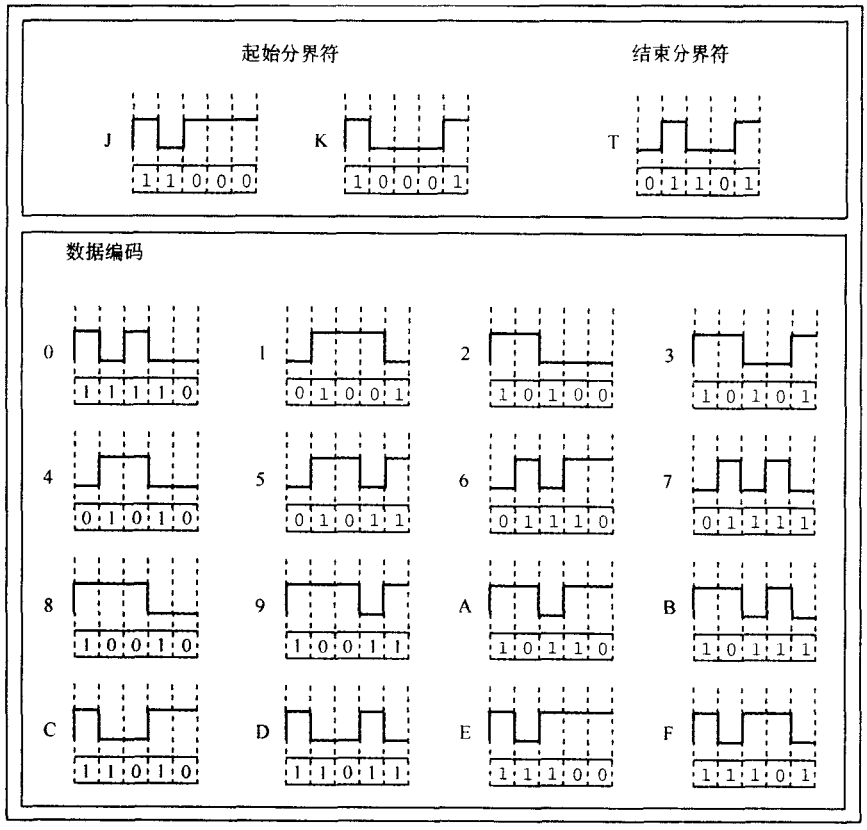


图 11-14　4B/5B 编码方式

波特（baud）是一种信号转换（跃变）数目的度量单位，这种度量是针对在一个媒体中由某种传输介质或传输方法所支持的信号的传输过程。对于非语音电话网而言，线路的速度是用赫兹单位来度量的。而对于数字信号来说，赫兹单位和波特单位是等价的。正如在图 11-14 中所看到的那样，如果网络使用 4B/5B 编码方法，那么 125MHz 的信号传送能力要求传输线路必须有一个 100Mb/s 的比特率。

光缆

光缆网络介质比双绞线或同轴电缆传输信号更快并且更远。理论上，光缆能够支持的传输频率在太赫（THz，百亿赫兹）范围，但是通常的传输速率在 2 千兆赫（GHz）以内，传输的距离为 10～100km（不需要使用转发器）。光缆是由一束细小（1.5～125 μm）的外面带有防护塑料外套的玻璃纤维或塑料线绳组成。虽然光纤传输的物理原理完全不同，但是我们可以把光纤看作是一种光的导体，就像把铜导体看作电的导体一样。光缆是一种“光导（light guide）”类型，它可以把光从光缆的一端传输另一端。在发送端，发光二极管或激光二极管会发射出各种光脉冲，这些光脉冲将沿着玻璃纤维绳传播，就好像水流过水管一样。在接收端光探测器会把光脉冲转化为电信号，这些电信号可以由各种电子设备进行处理。

光缆支持 3 种不同的传输模式，具体使用哪种传输模式要取决于所使用光缆的类型。各种不同的光缆类型如图 11-15 所示。最细小（narrowest）的光缆叫做单模（single-mode）光纤，只能传送一种波长的光，典型的波长为 850、1300 或 1500 nm。单模光纤可以有最快数据传输率和最远的传输距离。

多模（multimode）光纤可以通过一个较大的光纤芯同时传输几个不同的波长光。在多模光纤中，激光波会在光纤芯的壁上来回反射，这样导致比单模光纤有更大的衰减。这里，光波不但会发生散射，而且会在某种程度上相互碰撞，从而造成进一步的衰减。

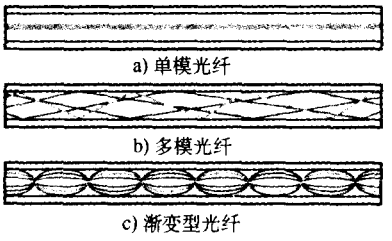


图 11-15 光纤

多模渐变型（multimode graded index）也支持多个不同波长的光同时传输，但是与常规的多模光纤相比，多模渐变型光纤采用一种更加可控的方式进行光传输。多模渐变型光纤由一些同轴心的塑料或玻璃层组成，每层的折射特性都优化成适合传送特定的波长。像常规多模光纤一样，光波会以反射波动的方式通过多模渐变型光纤。但是与常规的多模光纤不同的是，在多模渐变型光纤中，不同波长的光会被限制在光纤中的不同区域，这些区域适合于传播特定波长的光。因此，不同的波长的光通过纤维同时传输时相互之间并不会发生干涉。

光纤介质相对于铜导线来说有更多的优点。最明显的特点是光纤介质信号具有巨大的信号传送能力。而且不受 EMI 和 RFI 的干扰，是工业设施上是应用的理想材料。光纤既小又轻，一根光纤可以替代几百对铜导线。

但是，光缆很脆，购买和安装的费用也都比较高。由于这些原因，光纤通常用作网络的主干高速电缆（backbone cable），能够承受上百个或上千个用户的通信量。这种高速电缆就像一条州际公路，车辆只能够通过特定的入口和出口的访问进入，然而在这条高速公路上可以很高速地通过大量的交通流量。当车辆到达最终目的地时，必须要离开高速公路，然后可能要驶入一条住宅区的街道。住宅区街道的网络最通常采用的形式是铜双绞线。这种“住宅区街道”的铜导线有时也称为横向电缆（horizontal cable），它不同于主干电缆（纵向电缆，vertical cable）。毫无疑问，随着成本的降低，“光纤连到书桌”这种情况最终会成为现实。与此同时，在同一个电缆上集成传输语音和视频数据的需求正在稳步增加。随着这些新技术的发展，在下一代高速电缆引进之前，网络介质可能会延伸到它们的极限。

11.6.2 网络接口卡

传输介质是通过网络接口连接到客户机、主机和其他网络设备的。由于这些接口常常是在一些可移动电路板上实现的，所以它们通常被称为网络接口卡（network interface cards，NIC），或简称为网卡（请不要称作“NIC 卡”!）。网卡通常嵌入 OSI 协议堆栈的最下面的三层。它是一个连接网络物理设备和计算机系统之间的桥梁。NIC 一般直接连接到系统的主总线上或者是专用的 I/O 总线上。网卡会将系统总线上通过的并行信号转换为通信介质上传播的串行信号。网卡还负责把二进制编码的数据转化为网络上的曼彻斯特编码或 4B/5B 编码的数据，反之亦然。网卡还同时提供各种物理连接，以及协商允许将信号放到网络介质上。

每个网卡都有唯一的物理地址并固化在网卡的电路中。这个地址称为介质访问控制（Media Access Control，MAC）地址，长度为 6 个字节。前 3 个字节是制造商的标识码，由 IEEE 来指定。后 3 个字节是由制造商分配网卡的唯一标识码。世界上任何两个网卡的 MAC 地址都是不同的。网络的协议层会将这个物理 MAC 地址映射到至少一个逻辑地址。这个逻辑地址是网络上的一个节点相对于其他节点的名称或地址。一台计算机（逻辑地址）可能有两个或更多的网卡，但是每个网卡都有一个不同的 MAC 地址。

11.6.3 转发器

一个小型办公室的局域网的安装，相互之间只有几英尺的间隔，可能需要许多网卡。然而，在

一个复杂的办公室局域网中，网卡可能会被几百英尺长的电缆隔开。电缆越长，信号衰减越大。要减小这种信号衰减的影响，可以减小传输速度（通常这是一个令人难以接受的选择），或者向网络增加转发器。转发器（repeater，或称为中继器）可以通过放大信号来应对信号通过网络物理电缆时所发生的衰减问题。任何网络所需要的转发器的数目取决于信号传输的距离、传输使用的介质和线路中信号传输的速度。例如，高频铜导线每公里需要的转发器数量要比工作在同等频率下的光纤需要的转发器多。

转发器是网络介质的一部分。从理论上来说，它们是一些完全无需人为干预的被动工作的设备。同样，转发器不包含任何网络地址的设备。但是，现在有些转发器可以提供一些高级服务，帮助网络管理，以及进行故障诊断和修复。图 11-16 是一个转发器再生衰减数字信号的示意图。

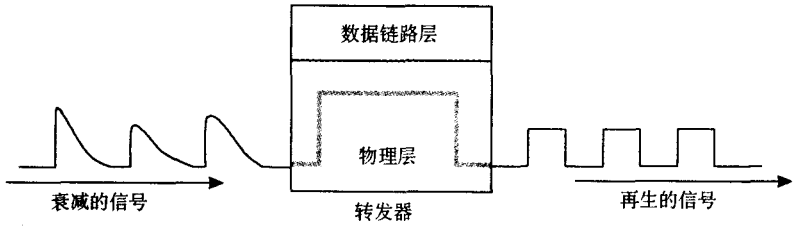


图 11-16　在 OSI 模型中的转发器的功能

11.6.4　集线器

转发器是物理层设备，只有一个输入端口和一个输出端口。集线器（hub）也是物理层设备，但是它可以有多个输入端口和输出端口。集线器既可以接收来自一个或多个位置的数据包，也可以在网络上将数据包传播到一个或多个设备。集线器允许计算机相互连接在一起形成网络链接段（network segment）。最简单的集线器是连接不同的网络分支的转发器。物理网络分支通过集线器连接在一起，不会以任何方式将网络分割开来。严格地说，集线器是第 1 层设备，它们并不会关心数据包的来源和目的地。不管集线器存在与否，网络上的每个站点都会与网络上的其他站点继续竞争来占用带宽。因为集线器是第 1 层设备，所以连接集线器的所有端口的物理介质必须是相同的。可以认为集线器除了提供多站点来访问物理网络之外，并不比转发器的功能多。图 11-17 表示一个有 3 个集线器的网络。

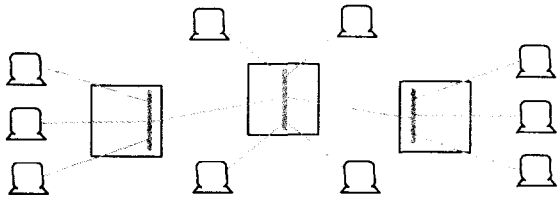


图 11-17　采用集线器连接的网络

随着集线器结构的发展，现在有许多集线器可以连接不同类型的物理介质。尽管像这样介质互连是第 2 层的功能，但是制造商继续把这些设备叫做集线器。交换式集线器（switching hub）和智能型集线器（intelligent hub）的出现进一步削弱了集线器属于“第 1 层设备”的概念。这些复杂的设备不仅能够连接不同类型的介质，而且可以执行完全属于第 3 层的路由选择和协议会话的基本功能。

11.6.5　交换机

交换机是第 2 层设备，可以在它的输入端口和输出端口之间产生点对点连接。虽然集线器和交换

机执行相同的功能，但是它们在内部处理数据的方式是不同的。集线器向网络上所有的计算机传播数据包，而且每次只处理一个数据包。另一方面，交换机能够处理与它们相连的计算机之间的多路通信。如果网络上只有两台计算机，那么集线器和交换机的行为方式完全相同。如果网络上有多台计算机正在进行通信，那么交换机具有更好的性能，因为交换机的两端可以使用网络的全带宽。因此，在大多数网络安装中，交换机优于集线器。在第 9 章中，我们介绍了处理器连接存储器或处理器连接处理器的交换机。那里的交换机和我们现在讨论的交换机属于同一种类型。交换机包含一定数目的带缓冲器的输入端口，相同数目的输出端口，一个开关网络结构（switching fabric，这个结构由多个开关单元、集成电路和允许控制交换路径的程序的组合构成）和数字硬件系统。数字硬件系统在网络数据帧到达输入缓冲器时，负责解释网络数据帧中编码的地址信息。

与所讨论的大多数网络设备一样，通过增加可编址和管理特性可以改进交换机的性能。现在，大多数交换机都可以报告它们所处理的信息量的数量和种类，甚至还可以根据用户提供的参数过滤某些网络数据包。因为所有的交换功能都采用硬件的形式执行，所以对于高性能网络设备的互连，交换机是优先选择的设备。

11.6.6 网桥和网关

网桥（bridge）和网关（gateway）的目的都是在两种不同类型的网络段之间提供一种链接。二者都可以支持不同的介质（和网速），而且它们也都是“存储和转发（store and forward）”设备，在发送帧之前会保存整个数据帧。这些都是它们的相似之处。

网桥用来连接两个同类型的网络，这样使它们看起来像是一个网络。如果使用网桥，那么这个网络上的所有计算机都属于同一个子网（subnet，子网是由具有同样前缀的 IP 地址的设备组成的网络）。网桥是相对简单的设备，它的主要功能集中在第 2 层。也就意味着网桥不了解协议的任何内容，只是简单地依据目标地址将数据转发出去。网桥可以连接具有不同介质访问控制协议的不同类型的介质，但是经过 OSI 堆栈中的所有较高层次来自 MAC 层的协议，在两个链接的网络段中必须是相同的。这种关系如图 11-18 所示。

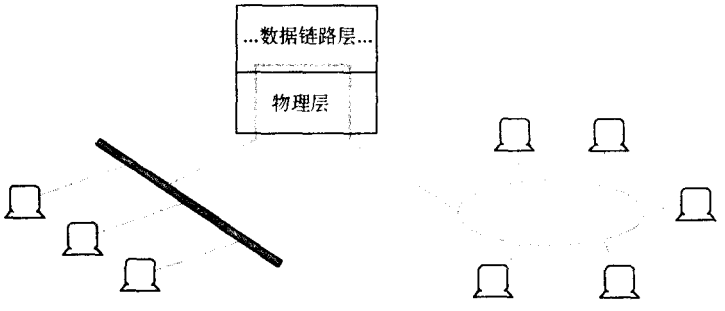


图 11-18 连接两个网络的网桥

连接到任何一个特定网桥的每个节点都必须有唯一的地址。这里，MAC 地址是最常用的地址。网络管理员必须利用这些地址和网络上每个有效节点的网络段号对网桥进行简单编程。允许通过网桥的唯一数据是正在被发送到网桥另一端的一个有效地址的数据。对于大型网络而言，各种情况的变化很频繁（大多数网络都是如此），这种频繁的再编程是非常乏味的，即浪费时间又容易出错。后来发明了透明网桥（transparent bridge）就是为了减轻这种网络管理员的再编程问题。透明网桥是一些复杂的设备，它们能够知道在每个网络段上的每个设备的地址。透明网桥还提供了信息管理，如报告吞吐量等。这样的功能实际上暗示了网桥已经不再是完全属于第 2 层的设备。然而，网桥仍要求在互连的两个网络段具有相同的网络层协议和相同类型的接口。

图 11-18 表示的是两个不同种类的局域网通过一个网桥连接在一起。这是网桥的典型使用方法。

但是，如果这些局域网上的用户需要连接到一个使用完全不同的协议的系统，例如一个公共交换电话网或者是一个使用非标准专用协议的主机，那么就需要使用网关（gateway）。网关是另一个网络的入口点。网关是一种完备的计算机系统，它可以跨越所有的 OSI 的 7 个层次提供通信服务。网关的系统软件可以进行转换协议和字符编码的工作，而且能够提供加密和解密服务。因为网关是利用软件处理大量的这类工作，所以网关不能够提供类似于基于硬件的网桥的吞吐量，然而网关通过提供更多的功能来弥补这个缺陷。通常，网关是直接连接到交换机或路由器的。

11.6.7 路由器和路由

路由器是继网关之后，网络上第二复杂的设备。事实上，路由器是一些小型的专用计算机。路由器（router）是至少连接两个网络的设备，路由器可以决定一个数据包应该被转发到的目的地。路由器通常设置在网关上。如果操作正确，路由器能够使网络运行得更快并做出及时的响应。如果操作不正确，则有故障的路由器可能会降低整个系统的性能。本节将揭示路由器的内部工作原理，并且讨论路由器需要解决的一些棘手问题。

尽管路由器非常复杂，但是通常把它们看作第 3 层设备，因为它们的大部分工作都是在 OSI 参考模型的网络层上完成的。然而，大多数路由器还提供了一些网络监视、管理和故障诊断服务。因为路由器是定义在第 3 层的设备，所以它们能够桥接不同类型的网络介质（例如光纤对铜导线）和连接运行在第 3 层或以下的不同的网络协议。由于具有这些功能，所以在网络标准文献中，路由器有时也被称为“中介系统（intermediate system）”或“网关”。在第一个网络标准写成时，还没有出现路由器这个词。

设计路由器是专门为了将两个网络连接在一起，通常是一个 LAN 连接一个 WAN。路由器是一些非常复杂的设备，因为它们不但包含缓冲器和开关逻辑，而且还配有足够的存储器和处理能力，计算发送一个数据包到目的地的最好方法。典型的路由器的内部概念模型如图 11-19 所示。

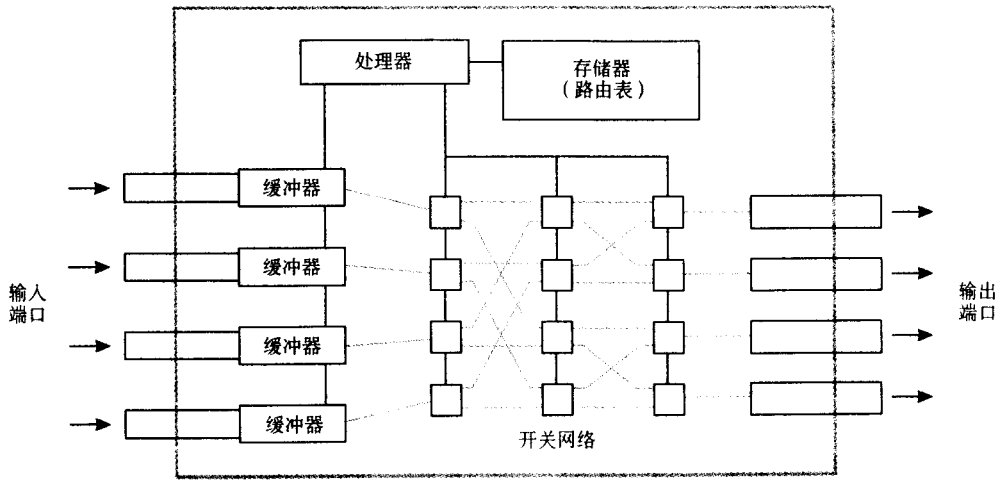


图 11-19 路由器的解剖图

在大型网络中，路由器会对一个基本上属于完全 NP 的问题找到了一个近似的解决办法。一个完全 NP 的问题在理论上不可能有最好的求解方法，但是如果计算的时间周期足够短，那么对解决网络问题还是有帮助的。

考虑如图 11-20 所示的网络。我们可以认为这个图是一个完全图（K_5）。在包含 n 个节点的完全图中，有 $n(n-1)/2$ 条边。在这个示意图中，有 5 个节点和 10 条边。这些边分别代表每个节点之间的路由或跳。

如果节点1（路由器1）要向节点2发送一个数据包，那么它有如下路由路径可以选择：

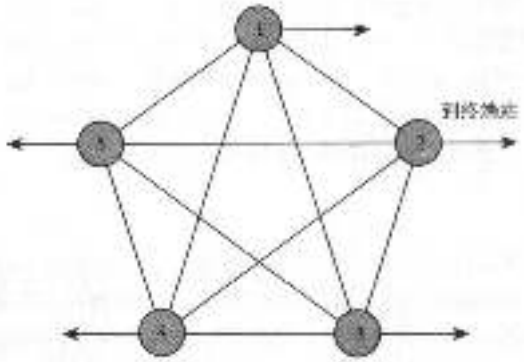


图 11-20 完全连接网络图

1个跳1种路由：1→2

2个跳3种路由：1→3→2　　1→4→2　　1→5→2

3个跳6种路由：1→3→4→2　　1→3→5→2　　1→5→4→2

1→4→3→2　　1→5→3→2　　1→4→5→2

4个跳6种路由：1→3→4→5→2　　1→4→3→5→2　　1→5→4→3→2

1→3→5→4→2　　1→4→5→3→2　　1→5→3→4→2

当节点1和节点2不是直接连接时，两个节点之间的交通至少要通过一个中间节点。如果考虑到所有的情况，那么可能发生的路由数是N!种算法。当代价或权值也要应用于路由路径时，这个问题会变得更加复杂。更糟糕的是，权值随着交通流量的变化而变化。例如，如果节点1和节点2之间的连接是一条收费的高等待（high-latency，慢）的线路，那么我们可能会选择使用1→4→5→3→2这条路径。很显然，在实际的网络中有几百个路由器，问题将会变得更加庞大。如果每个路由器对每个到达的数据包都要考虑各种可能性而挑出理想的发送路由的话，那么数据包就不可能迅速到达目的地。

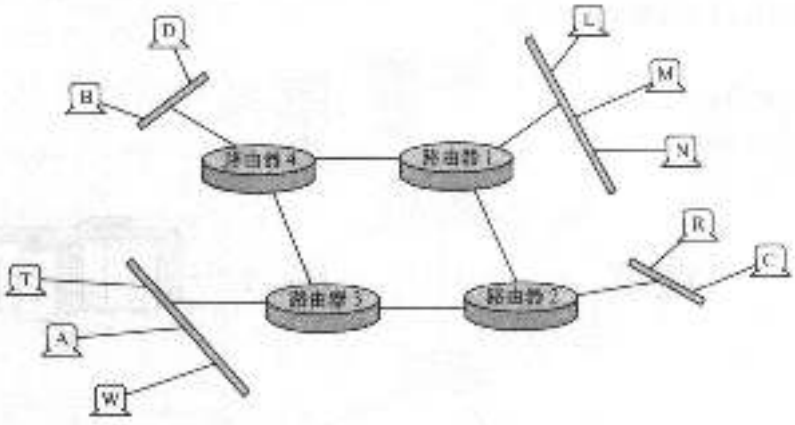
当然，在只有少数几个节点的非常稳定的网络中，每个路由器总是使用最佳路由的情况是可能的，这种情况称为静态路由（static routing）。对于集中在某个位置的大量用户都使用一个中央主机，或一个网关，而在另外位置也类似的网络来说，这种最佳路由是切实可行的。从短期的观点来看，这是一种非常有效的系统互连方式。但是如果其中的某个互连链路或路由器出现了问题，那么大量用户都将无法连接到主机。这样，要求网络管理人员必须快速做出反应来恢复中断的服务。在网络状态频繁发生改变的情况，静态路由不是一种合理的选择。这就是说，对于大多数网络来说都不选择使用静态路由。相反，静态网络是可预测的和可控制的网络，每个数据包要选取的路由路径（也就是跳数）总是已知的。静态路由同样是非常稳定的，它产生的是无路由协议交换的通信方式。

动态路由器会自动建立路由路径，并对网络中的改变做出响应。这些路由器同样可能会选择一个最优的路由和一条备用路由路径，以便出现问题时可以有一种路由选择。动态路由器不会改变路由指令，但是允许路由表发生动态改变。

动态路由器可以通过与网络中的其他路由器的信息交换来自动检测它们所在的网络。通过路由器交换的信息包含显示它们的地址和从一个节点到另一个节点所需的代价。通过使用这些信息，每个路由器都会在它们的存储器中装配一个路由表。实际上，路由表是网络上每个节点的一个可到达性列表，再加上一些默认值。通常情况下，每个目标节点都是和它要连接的相邻路由器，即下一跳（next-hop），一起列表的。

在创建路由表时，动态路由器采取下面两种方法中之一：它们或者使用两个节点之间的路由距离，或者根据测量的延时时间来考虑网络状况。使用第一个方法的算法是距离矢量路由（distance vector routing）算法。使用第二个方法的算法是链接状态路由（link state routing）算法。

距离矢量路由算法是从 1957 年和 1962 年发明的两个算法中衍生而来的，这两个算法分别是 Bellman-Ford 算法和 Ford Fulkerson 算法。距离矢量路由中的距离（distance）通常是指对数据包到达目标节点前必须经过的节点（跳）数的一种测量，但是可以使用任何一种测量方法。例如，已知一个网络如图 11-21a 所示，其中有 4 个路由器和 10 个节点连接在一起。如果节点 B 想发送一个数据包到达节点 L，则有两种选择：第一个方案是 B→路由器 4→路由器 1→L。在路由器 4 和路由器 1 之间有一次跳跃。第二种路由方案选择在路由器之间进行三次跳跃：B→路由器 4→路由器 3→路由器 2→路由器 1→L。通过使用距离矢量路由算法，目标总是要选择最短的路由路径，所以在这里路由路径 B→路由器 4→路由器 1→L 是最简而易见的选择。



a) 一个带有 4 个路由器和 10 个节点的网络例子

路由器 3

目标节点	下一跳	跳数
A	--	0
B	B4	1
C	B2	1
D	B4	1
L	B4	2
M	B4	2
N	B4	2
R	B2	1
T	--	0
W	--	0

路由器 1

目标节点	下一跳	跳数
A	B2	2
B	B4	1
C	B2	1
D	B4	1
L	--	0
M	--	0
N	--	0
R	B2	1
T	B4	2
W	B4	2

路由器 2

目标节点	下一跳	跳数
A	R3	1
B	R3	2
C	--	0
D	R3	2
L	R1	1
M	R1	1
N	R1	1
R	--	0
T	R3	1
W	R3	1

b) 从路由器 1 和路由器 3 的路由表来初始路由器 2 的路由表

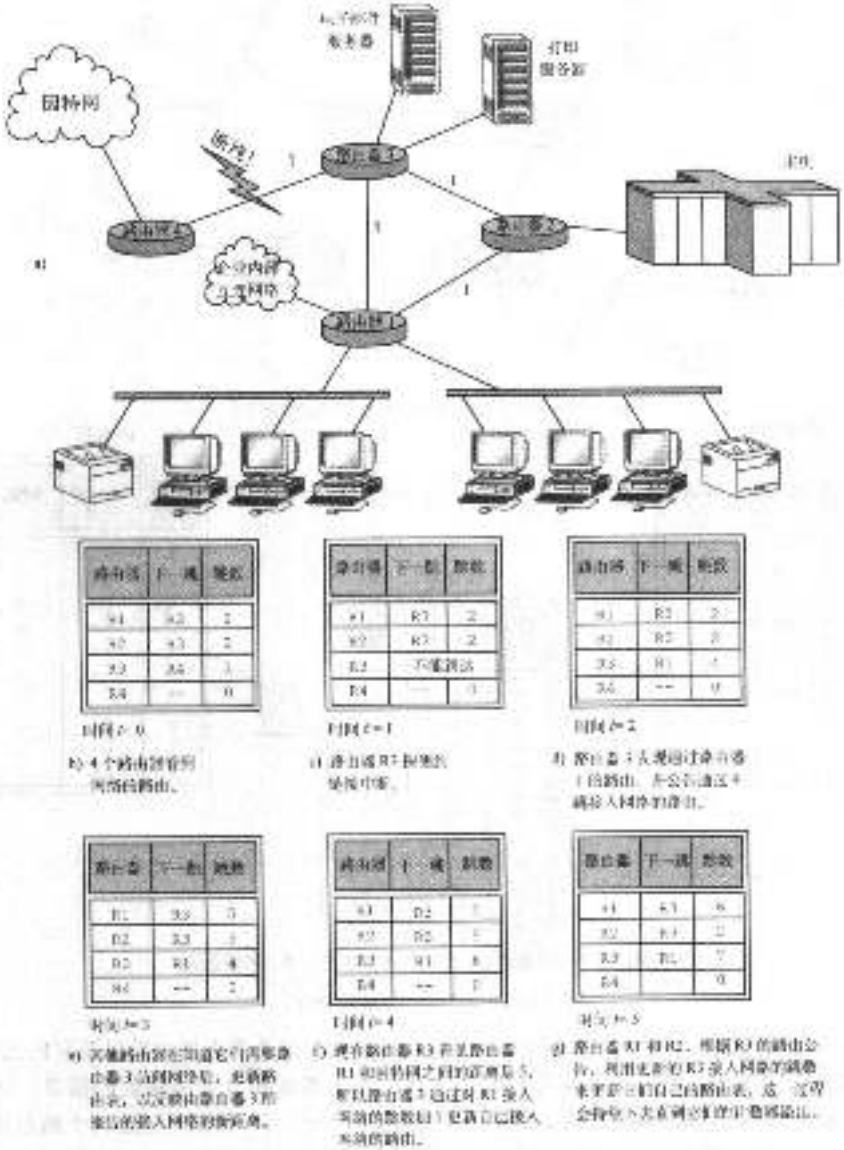
图 11-21 带有路由器的例子

在距离矢量路由算法中，每个路由器需要知道与它相连接的每个节点的身份以及它们之间的跳数。为了有效地完成这项工作，路由器必须与它相邻连接的路由器交换节点和跳数的信息。例如，在图 11-21a所示的网络中，路由器 1 和路由器 3 的路由表如图 11-21b 所示。然后，把这两个路由表发送到路由器 2。如图所示，路由器 2 在考虑所有的路由表中报告的路由路径后，会选择到达任何一个节点的一条最短路径。最终的路由表中包含直接连接到路由器 2 的各个节点的地址信息，以及一个可以通过其他路由器到达的目标节点和节点的跳数的列表。注意，路由器 2 的最终路由表的跳数增加 1 的原因是，考

度了路由器1和路由器2以及路由器2和路由器3之间有1个跳跃。一个实际的路由表还应该包含没有直接连接到网络的节点所使用的默认路由地址，例如某个远方的局域网（LAN）或网络目标的站点。

距离矢量路由算法很容易实现，但是它却存在一些问题。首先，在大型的网络中稳定（或者会聚，converge）路由表需要占用的时间比较长。另外，当路由表发生更新时，网络上会产生数量可观的信息流。第三种情况是，那些过时的路由器经仍然会存放在路由表中，这将造成错误路由或丢失数据包的现象发生。最后一个问题是存在无限计数（count to infinity）的缺点。

通过学习图11-22a中的网络，可以理解无限计数的问题。值得注意的是，网络上存在许多冗余的路径。例如，如果路由器3发生脱机掉线，客户端仍然可以连通主机和网络，但是直到路由器3再次正常运行时，它们才能够输出打印信息。


因特网
打印
服务器
1
1
1

各个路由器接入网络的所有路径如图 11-22b 所示。我们把这种网络快照（snapshot）的开始时间记为 $t=0$。正如图中所示，路由器 1 和路由器 2 都使用路由器 3 连接到因特网。在 $t=0$ 和 $t=1$ 之间的某个时间，路由器 3 和路由器 4 之间的连接信号在不断减弱。据说是有人拔掉了这些路由器之间的连接电缆。在 $t=1$ 时，路由器 3 发现连接中断，但是它刚刚已经收到了来自相邻两个路由器的路由表更新数据。这两个路由器都公告（advertise）说它们自己可以利用 3 跳来接入网络。接着，路由器 3 认为它能够使用其中的一个路由器连接到网络，并且更新相应的路由表。路由器 3 就选路由器 1 作为自己接入网络的下一跳，然后在 $t=2$ 时刻向路由器 1 和路由器 2 发送了自己的路由表。在 $t=3$ 时刻，路由器 1 和路由器 2 都收到了路由器 3 已经更新的接入网络需要的跳数。因此，它们把路由器 3 的值加 1（因为它们知道距离路由器 3 还需要 1 跳），接下来就广播它们自己的新的路由表。这个循环会一直持续下去，直到所有的路由器的跳跃计数都达到无穷大为止。这意味着存放跳数的寄存器最终会溢出，导致整个网络瘫痪。

通常，可以使用两种方法来防止这种情况。一种方法是对于无穷大使用一个较小的值，便于较早地检测出问题，如在某个寄存器溢出之前就发现问题。另一种方法是采用某种方式来阻止像我们这个例子中所出现的短周期问题。

一些高级复杂的路由器使用一种方法叫做水平分割（split horizon）路由，来保持网络的不会出现短周期。简单的思路是：路由器不使用相邻路由器提供的路由选择，也包括本身的路由。类似地，路由器可以自我发现和使用自参考路由，但是要设置路径的值为无穷大。这种方式称为带有破坏性反转的水平分割。这种路由是“破坏性的”，因为它被标记为不可到达。就上面这个例子而言，使用水平分割路由方式的路由表交换的接入网络的路径将会收敛（converge），如图 11-23 所示。当然，仍然会发生较大循环的问题。例如路由器 1 指向路由器 2，路由器 2 指向路由器 3，路由器 3 指向路由器 1。这种问题可以得到某种程度的改善，做法是只有当一个链接需要更新时，路由器才交换它们的路由表。这种更新称为触发更新（triggered update），采用这种方式发生的更新，能够导致路由图中的循环数目较少，而且可以减少网络中的交通流量。

路由器	下一跳	跳数
R1	R3	2
R2	R3	2
R3	不可到达	
R4	--	0

时间 $t=1$

a) 路由器 R3 在接入网络的路径中检测到失效的链接。

路由器	下一跳	跳数
R1	R4	3
R2	R1	4
R3		∞
R4	--	0

时间 $t=2$

b) 因为接入网络的所有其他路由都告 R3，所以 R3 会公告它接入网络的跳数为无穷大。看到这个公告，其他的路由器会找到一条较短的路由路径。

路由器	下一跳	跳数
R1	R2	3
R2	R1	4
R3	R1	4
R4	--	0

时间 $t=3$

c) R3 看见了它接入网络的最短路由是经过 R1，并且会相应地更新其路由表。

图 11-23 具有破坏性反转的水平分割路由

在大型网络中，跳数可能是一个会产生误导的度量标准。特别是当网络包含各种不同的设备和线路速度时。例如，如果一个数据包可以有两种方式到达某处。一种方式是在 100Mb/s 的局域网（LAN）上经过 6 个路由器。另一种方式就是在 64Kb/s 的租用电话线路上经由 2 个路由器。虽然这个 100Mb/s 的 LAN 可以提供超过 10 倍的吞吐量，但是跳数的度量结果将使数据包的信息流进入较慢的租用电话线路。如果我们不测量跳数，而是测量实际线路的延时，那么就可以防止出现这样的反常情况。这正是链接状态路由方法（link state routing）的基本思想。

与距离矢量路由方法一样，链接状态路由也是一种自我管理的系统。每个路由器通过周期性发送问候（Hello）数据包来发现它自己及其相邻的路由器之间的线路速度。只要路由器一释放问候包，它

就开始计时。接下来，接收到问候包的每个路由器都会立即发送一个回复。当启动的路由器收到一个回复后，它就会停止计时，并将计时的结果值除以 2，这样就估算出到达给出回复的路由器的单向链接所需要的时间。在收到了所有的回复之后，路由器会把所有的时间整合成一个路由状态值表。然后，这个路由状态表会向其他所有的路由器广播，但是和它相邻的路由器除外。不邻近的路由器会利用这个信息更新包括发送路由器在内的所有路由路径。最后，在路由范围内的所有路由器都有同样的路由表。简单地说，在这种独同现象发生后，一次单一的网络快照会存在于每个路由器的路由表中。然后，路由器会使用图像来计算其路由表中每个目标节点的最优路径。

在计算最优路由时，每个路由器都会被编程，认为它自己是一个树结构的根节点，而每个目标节点都是这棵树的一个内部叶节点。利用这种概念，路由器使用 Dijkstra 算法[1]来计算到达每个目标节点的最优路径。找到这个最优路径后，路由器只是存储沿着这个路径的下一跳，而不是存储整个路径。下一个节点（沿最优路径下游）的路由器也应该已经计算出了同样的最优路径，或者是数据包到达节点时刻计算出的更好路径。而这个数据包采用的路径是上游节点计算出的最优路径中的下一个链接。对图 11-22 中的路由器 1 应用了 Dijkstra 算法后，路由器 1 所看到的网络路由如图 11-24 所示。

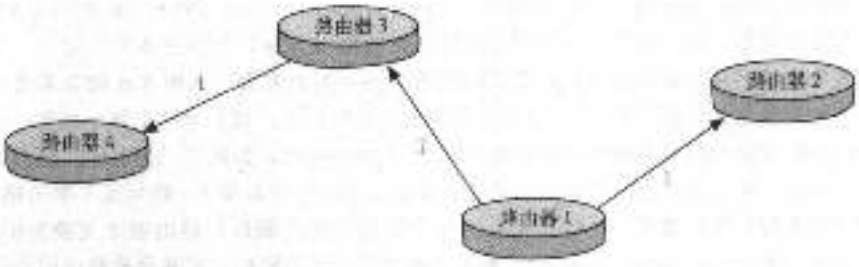
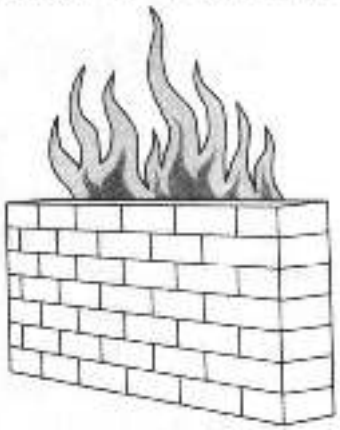


图 11-24 在图 11-22a 中使用链接状态路由和 Dijkstra 算法后，路由器 1 如何看待网络

很清楚，路由器只能保持有限的信息。一旦网络达到性能开始下降的规模时，网络就必须被分割成若干个子网络，或称为网络段（segments）。通常，这种情况的发生有许多原因，而不仅仅是路由表饱和。在一些大型网络中，应用分层结构的拓扑将有助于保持系统的可管理性。这种分层结构组合了交换技术和路由技术。最好的网络设计师非常清楚地了解网络设计中何时应该使用什么技术。当然，网络设计的最终目标是使网络的吞吐量达到最大化，同时保持网络易于管理和性能稳定。 ■

什么是防火墙？

实际上，政府机关、工业和学术界的每个人在日常事务过程中都会使用网络。每一个人都可以接触到网络，甚至包括窃取或摧毁某个公司的计算资源的人。因此，如何保持网络对每个使用网络进行工作的人员有足够的访问权限，同时又保证公司的财产有足够安全呢？解决这个问题的首选办法是在企业内部网和因特网之间设置一道防火墙。



防火墙（firewall）的名称，是根据在相邻建筑物之间设置一堵真砖墙的类似得来的。如果其中的某个建筑物发生了火灾，那么相邻的建筑物就会因为中间高墙而受到保护，免受火灾的侵袭。因此，防火墙是一种网络的防火墙，利用防火墙可以把网络的内部用户与某些可能会破坏内部网络结构的外来用户分隔开来。

防火墙有许多种类。最常用的有两种类型，分别是基于路由器的防火墙和基于主机的防火墙，或者说是基于代理服务器的防火墙。这两种防火墙都采用同一种规则基础进行规划，这种规则被称为策略

[1] 对于 Dijkstra 算法的解释请参阅附录 A。

(policy)。防火墙的策略是规定哪些类型的网络地址允许访问哪些类型的服务。策略的一个非常好的示例是与文件传输有关的例子。通过对防火墙进行编程，允许网络内部用户（网络中受保护的一方）可以从因特网下载文件，而禁止网络外部用户从内部网络上下载文件。原因是网络内部的数据可能包含有一些敏感的私人信息。任何一个防火墙都可以对一个要禁止的地址列表编程，有时候也叫黑名单。列入黑名单的地址中通常会包括散播一些引起异议的信息网站。

两种类型的防火墙还能够区分入站和出站的流量。这样可以防止地址欺骗，愚弄防火墙误以为使用这个地址用户在网络内部，而实际上这是一个网络外部的用户。如果防火墙不能识别地址欺骗，那么网络外部的用户就可以自由出入内部网络。

路由器防火墙和代理服务器防火墙都有能力对网络的交通进行加密。*加密*（encrytion）是使用某种算法和密钥值打乱消息的排列顺序的过程，只有具有相应的密钥的设备才能够读取加密的消息。密钥值是周期性的改变，通常是每天都发生改变。当使用密钥交换程序对防火墙进行编程时，这种改变过程就会自动发生。路由器倾向于使用较简单的加密算法，常用的算法是使用消息和密钥值进行简单的移位和逻辑与（AND）操作。其中一种算法是*美国联邦数据加密标准*（Data Encrytion Standard，DES)。为了更加安全，有时要对消息加密 3 次，这种做法称为 3-DES。

读者可以想像，代理服务器防火墙要比路由器防火墙的速度更慢、更容易失效。但是它们比路由器防火墙具有更多的特点。首先，它们具有可以为内部网络用户担当代理的能力，因此又叫做*代理*(proxy) 服务器。这些系统通常配备 2 块网卡，即它们有*双向作用*（dual homed)。其中一块网卡连接内部网络，另一块网卡连接外部网络。利用这样的配置，服务器可以对外部网络用户完全屏蔽掉内部网络的特性。外部网络用户所能够看到的仅仅是连接外部网络的网卡地址。

服务器防火墙还可以保存大量的网络日志。通过这些日志，安全管理员能够发现大部分黑客的入侵企图。在某些情况下，日志可以提供有关入侵者的信息。 ■

11.7 高容量数字链路

既然读者理解了数据网络的物理构成和网络运行的协议，那么可以进入全球通信基础设施的领域。下面的部分将介绍有关构建一个数字管道网络的概念和问题，数字管道是一种数据、音频和视频共享的网络。只是在过去的 10 年中，集成不同类型的信息流才变得有可能实现，而且这种集成的工作仍然在不断地改进完善之中，成功实现还有一段时间。

11.7.1 数字分层体系

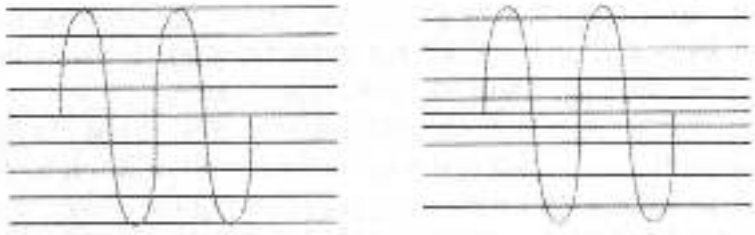
铺设大型数字管道的概念是由拨号语音服务提供商提出的。直到 20 世纪 60 年代末期，电话服务通常在其系统中都使用模拟信号。高容量的模拟线路（主干线或中继线，trunk line）连接到全球各个城市的长途电话中心。中继线使用一种被称为*频分多路复用*（frequence division multiplexing，FDM）的技术，在一对双绞线上可以同时传输多个通话。FDM 类似于收音机接收器。我们可以通过旋转调谐刻度盘来选择收听任何一个广播电台。所有的这些信号都可以共享同一种通信介质，因为它们存在不同的频率。当 FDM 用于长途电话电缆时，每对话路都被分配自己的频带，允许一打或更多的话路在同一根导体上进行传输而不会引起相互干涉。

使用模拟系统，会存在一些问题，其中一个就是衰减问题。为了防止信号在长距离传输时衰减消失，沿着线路每隔几英里就需要安放一个模拟放大器。模拟放大器可以增强语音信号，同时也放大了从外面进入到线路中的各种干扰信号和背景噪声。结果造成了通话的质量比较差，超远距离连接时还伴随有恼人的静电噼啪声。

在 20 世纪 60 年代，Bell 系统公司开始把中继线从模拟信号转换为数字信号。在这里不再使用放大器来增强信号，数字载波使用转发器再生信号。转发器不仅比放大器便宜很多和更加可靠，而且它们只是再生有用的线路信号。背景噪声不是有用信号的一部分，所以背景噪声不会被再生和沿着线路

传播下去。当然，由于人类语音是一个模拟信号，因此在将它加载到数字载波上进行传送之前一定要转化为数字信号。这种模拟信号到数字信号的转换技术称为脉码调制（pulse-code modulation，PCM）。

PCM 依赖于这样一个事实，正常人的声音所产生的最高频率大约是 4000Hz。因此，如果对一个电话会话语音每秒钟采样 8000 次，那么语音的振幅和频率能够准确地以数字形式表现出来。这个思想可以用图 11-25 说明。图 11-25a 说明了使用均匀间距（水平）量化（quantization）电平的脉冲幅度调制。每个量化电平都可以使用二进制值来进行编码。在 4000Hz 宽带音频和低端。这种调制结构的每一位都传输同样多的信息。然而，由人的声音所产生的信息和由人的耳朵所感听的信息都不是均匀分布的。而是在宽带的中间位置聚集成束。因为这个原因，如果 PCM 的量化电平在宽带的中间部位也形成类似的聚集束，那么就可以实现人声音的真实表现，如图 11-25b 所示。因此，PCM 载波的信息在某种意义上反映了它是如何产生和解释的。



a) 脉冲幅度调制　　b) 脉码调制

图 11-25

经过主观测试，BellCore（贝尔系统公司的工程单位，现在叫做 Telcordia）的工程师发现，当他们使用 127 量化电平时，不能区分脉码调制信号和一个纯的模拟信号。因此，每次取样只能用 7 位二进制数来传输信号的振幅。在最早期的 PCM 配置中，BellCore 为了发送信号和控制目的，将第 8 位二进制数添加到 PCM 取样中。现在，全部 8 位都被使用。通过一个语音连接所产生的单串 PCM 信号需要有一个 64Kb/s（8 位×8000 次取样/秒）的带宽。数字信号 0（DS 0）是 64Kb/s PCM 二进制位串的信号比率。

为了形成一个传输帧，来自于 24 种不同语音连接的一系列 PCM 信号都被放置到线路中，利用按

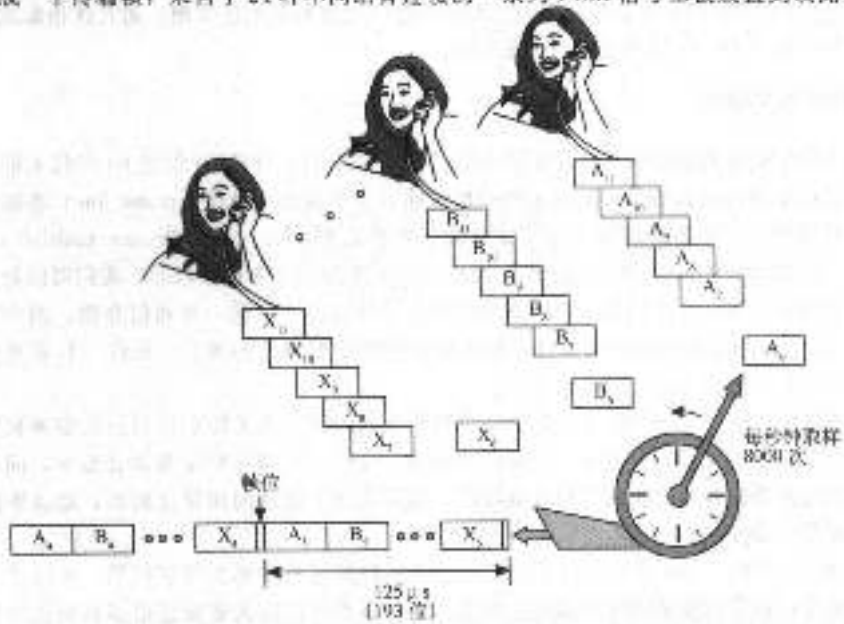
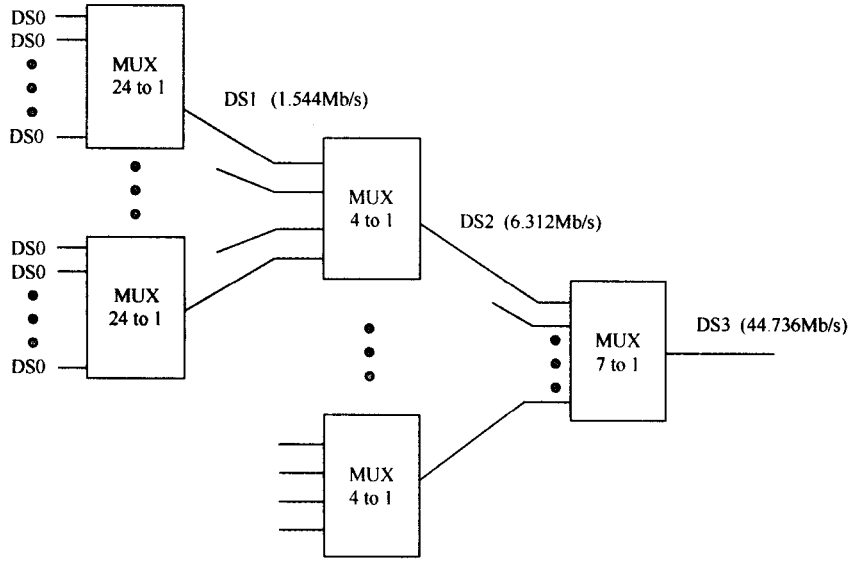


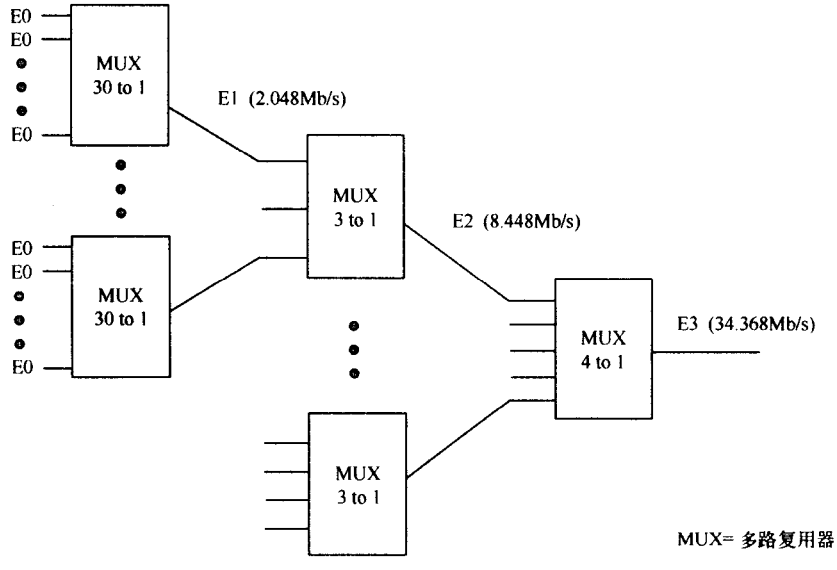
图 11-26　时分复用

制信道和帧位形成 125μs 传输帧，如图 11-26 所示。因为每个连接大约取得 125μs 帧的 1/24，所以这个过程被称为时分复用（time division multiplexing，TDM）。对于每个连接每秒钟取样 8000 次，这种语音信道、信号信道和帧位的组合要求总的带宽为 1.544Mb/s。

1.544Mb/s 和 2.048Mb/s 的线路速度（传输速度）分别叫做 T-1 和 E-1，并且它们传输的是 DS-1 信号。正如可以对 24 路和 32 路电话会话进行复用形成一个传输帧一样，这些帧本身也可以被复用到一个高速的链路上实现多路传输。图 11-27 表示一个低速从属线路如何复用到高速的主干中继线路上。这组载波速率是由准同步数字体系（Plesiochronous Digital Hierarchy，PDH）的复用层所产生的。这个体系之所以叫做准同步（区别于同步），是因为每个网络元件（例如一个开关或复用器）都有自己的



a) T 载波体系



b) E 载波体系

图 11-27 准同步数字体系

时钟，它与这个体系中在它层次之上的时钟进行周期性的同步。在载波上没有设置单独定时信号，就如同一个真正的同步网络的情况一样。这个系统中最顶层的时钟非常准确，其误差漂移可以忽略不计。然而，由于定时交换信号要通过这个分层结构进行传播，于是就产生了误差。当然，越深的层次结构，在到达底层之前信号就越可能发生漂移或滑动。

PDH 内在时序问题的最终解决办法是，网络中的所有元件都使用一个单一的定时信号。在 20 世纪 80 年代，BellCore 和美国国家标准化组织（ANSI）制定了一个同步光纤网的标准，SONET。这个标准最终变成了北美地区的主要光学载波系统。由于 SONET 是以传输帧的大小为 193 位的 T 载波构建的，而欧洲的通信系统使用的是一个 256 位的帧，所以 SONET 的系统不适合在欧洲使用。相反，欧洲把 SONET 应用于 E 载波系统，称为同步数字体系（synchronous digital hierarchy，SDH）。正如 T 载波系统的基本信号是速率为 1.544Mb/s 的 DS-1 一样，基本的 SONET 信号是速率为 51.84Mb/s 的 STS-1（同步传输系统 1，synchronous transport system 1）。当 STS 信号通过光载波网络时，这个信号就叫做 OCx，其中 x 是载波速率。基本的 SDH 信号是 STM-1。它传输信号的速率可达155.52Mb/s。表 11-2 列出了光载波体系 SONET 和 SDH 的对比。使用 T 载波系统和 E 载波系统的位速率要低下图中给出的数字。

表 11-2 北美（SONET）和欧洲（SDH）光载波系统

信号系统		光学载波	Mb/s
SONET	SDH		
STS-1		OC-1	51.84
STS-3	STM-1	OC-3	155.52
STS-9	STM-3	OC-9	466.56
STS-12	STM-4	OC-12	622.08
STS-18	STM-6	OC-18	933.12
STS-24	STM-8	OC-24	1244.16
STS-36	STM-12	OC-36	1866.24
STS-48	STM-16	OC-48	2488.32
STS-96	STM-32	OC-96	4976.64
STS-192	STM-64	OC-192	9953.28

对于长距离传输而言，T-3 和 E-3 都要转换成 OC-1 和 SDH-1（和更高级别）载波。然而，使用 SONET 或 SDH 可以不需要类似于图 11-27 中所示的阶梯分层结构。从理论上来说，如果有需要的话，64 OC-3 载波能够直接被复用到 OC-192 载波上。这样做可以使网络的管理更加简单，而且减少了通过系统的延迟时间。

光载波系统非常适用于作为超快的广域网（WAN）和因特网的主干网络的载体服务。这个问题现在已经变成了如何将高容量服务提供到家庭和小型商业企业。电话线的“最后一里”，即本地回路（local loop），是连接中继局及其客户。大部分的本地回路都是使用低宽带的铜双绞线，不适用于高级的数据传输。但是随着各个地区的电话公司不断对其有线路进行升级，以及一些参与竞争的地区性交换承运商在他们有希望获得利润的地区采用新（光纤）的线路，这种情况正在改善之中。

11.7.2 ISDN

把数字通信服务直接连接到家庭和企业的概念已经不再是什么新名词了。这个概念可以追溯到数字语音传输的起源。但是，在 30 多年前这种思想还是有些新潮超前的。1972 年，ITU-T 组织（当时称为 CCITT）开始从事一系列推广活动，推广一种全数字网络。全数字网络可以把音频、视频

和数据直接传送给用户。这种网络具有一个通用接口，可以很容易地连接到任何一个有合适设备的用户，相应地，这种网络被称为综合业务数字网（Integrated Services Digital Network，ISDN）。1982 年，发表了这个系列活动的第一份建议书，在随后的 10 年时间中不断出现各种补充和改进的版本。

ISDN 是严格依照 ISO 参考模型来设计的，ISDN 中的组件跨越了整个的 7 层模型。但是这些建议中的大部分内容都属于第 1 层到第 3 层的范围。这些 ISDN 建议的主要内容集中在位于 ISDN 模型中的某些特定参考点的不同网络终端设备和接口上面。这种系统的组织结构如图 11-28 所示。

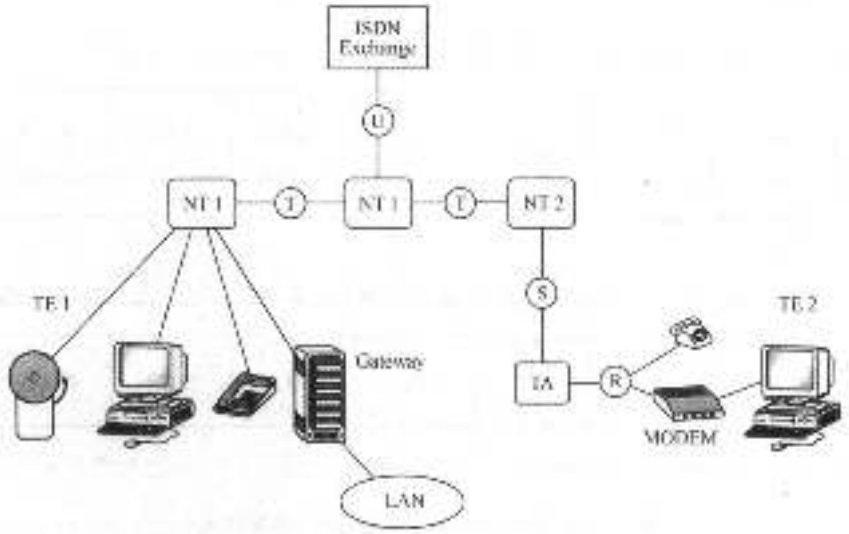


图 11-28 ISDN 系统的组织结构

网络终端 1（Network Termination，NT 1）是一种完全放置于用户前面的网络终端设备。NT-1 通过 T 型接口（终端）连接到数字网络。NT 1 设备可以支持多路的 ISDN 信道和各种类型的数字设备。有些数字设备，例如报警系统，可以一直对网络开放（处于接通状态）。一个 ISDN NT-1 设备可以直接连接多达 8 个设备。能够直接连接到 NT-1 设备的设备叫做 TE-1，即第 1 种类型的终端设备（Terminal Equipment Type-1）。

不能够直接连接到 NT-1 端口的设备可以通过一个 ISDN 终端适配器（Terminal Adaptor，TA）接入数字网络。这类设备被称为第 2 种类型的终端设备（Terminal Equipment Type-1，TE-2）。在 TA 和 TE-2 之间的参考点被称为 R（速率）参考点。TE-2 包括了所有的不能直接连接到数字线路上的设备，例如家用计算机必须要通过一个调制解调器（Modem）才能连接到因特网。终端适配器（TA）要通过 S（系统）参考点接到网络。

网络终端 2（Network Termination，NT-2）是一种智能型的 ISDN 接口，它可以提供从第 1 层到第 3 层的各种服务。要求高比特率的设备，如专用的自动（电话）交换分机系统和局域网（LAN），可以直接连接到一个 ISDN 的 NT 2 端口。NT 2 设备通过 T 参考点接入 ISDN 网络。从实用的角度来看，S 接口与 T 接口是一样的，所以有时称为 S/T 接口（S/T interface）。NT-2 设备称为信道服务单元/数据服务单元（Channel Service Unit/Data Service Units，CSU/DSUs）。

一个 ISDN 的本地回路服务通过 U（用户）参考点连接到 ISDN 交换局。ISDN 交换局包含与模拟电话设备不同的数字交换机设备，而且它们之间是完全隔离开来的。模拟电话系统会基于一种带内（in-band）信号在呼叫双方之间建立和中断交换电路。换言之，当使用模拟电话呼叫时，系统会使用和传送声音时相同的传输频率对路由信息（呼叫站点的编号）进行编码。特定频率的各种声调，或者是通过拨号方法产生的脉冲信号也都可以编码成路由信息。但是，ISDN 是使用独立的信号信道来输送这

些信息的，这些独立的信号信道与 ISDN 帧的数据信道多路复用。

ISDN 支持两种信号速率结构：基本（basic）速率和主要（primary）速率。使用基本速率和主要速率的线路连接分别称为基本速率接口（Basic Rate Interface，BRI）和主要速率接口（Primary Rate Interface，PRI）。基本速率接口由两个速率为 64Kb/s 的 B 信道（B-Channel）和一个速率为 16Kb/s 的 D 信道（D-Channel）组成。这些信道完全占据了一个 T-1 帧的两个信道再加上第 3 个信道的四分之一，如图 11-29 所示。ISDN 的主要速率接口占据了整个 T-1 帧，可以提供 23 个速率为 64Kb/s 的 B 信道和整个 64Kb/s 的 D 信道。可以对 B 信道进行多路复用，提供较高速数据速率，例如速率为 128Kb/s 的住宅区网络服务。

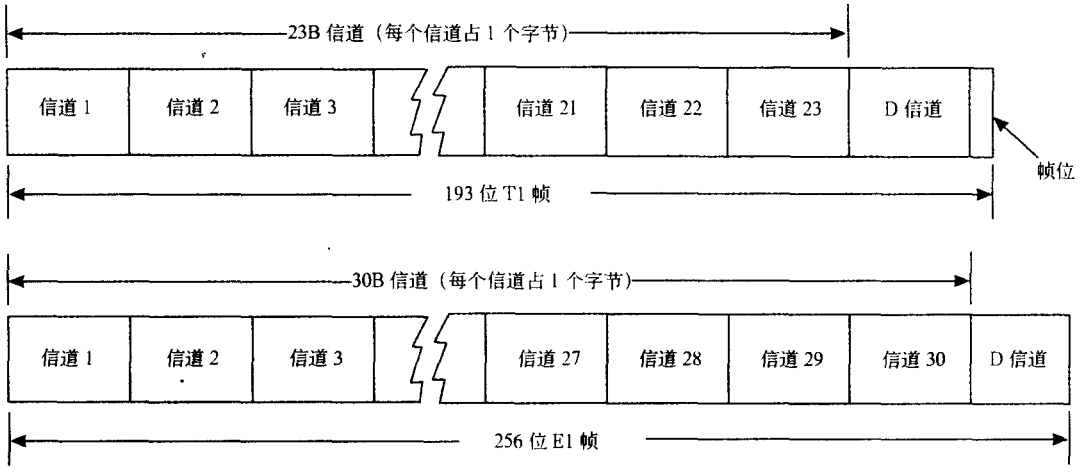


图 11-29 基本速率和主要速率的 ISDN 信道

D 信道可以对在 B 信道上的通信传输提供各种信号和网络管理服务。通过使用一种被称为系统信号传输 7（System Signal 7，SS7）的协议可以实现这种信道之间的连接管理，SS7 已经超出了本课程的范围（读者可以参考本章结尾处的参考文献）。

不幸的是，ISDN 委员会在发布建议书时，既没有足够的远见，也没有做出迅速的反应。几乎过了 20 年，才把第一个 ISDN 产品推向市场，而这时企业和用户所需求的带宽已经远远超出了 ISDN 所传输的能力范围。ISDN 为语音提供太多的带宽，而为数据提供的带宽却非常少。因此，除了相对少数的家庭网络用户之外，ISDN 实际上已经变成了技术上的孤儿。但是，ISDN 的重要性在于：它构成了连接一种更先进的称为异步传输模式（Asynchronous Transfer Mode，ATM）的通用数字系统的桥梁。

11.7.3 异步传输模式

通常使用在语音通信中的传统的时分复用技术，并不能充分利用网络的传输带宽。当两个人在进行（有礼貌的）对话时，其中一方会在另一方开始说话之前暂停 1 秒或 2 秒时间。在这 2 秒期间或者说是这段“寂静的空气（dead air）”时间，可能会有 16000 个空的信道时间段沿着这条线路传送。由于采用的是时分复用技术，所以这个宽带将会被浪费，原因是整个信道都要为这个呼叫需要的持续时间而保留起来，而不管信道是否在传递有用的信息。

如果能够截取这些空的信道时间段，那么这些空时间段就可以用来传输数据，甚至可以进行另一个对话。然而，要实现这一点，我们必须将每个信道拆分成一些独立的单元，并彻底放弃采用 24 个固定的位信道占据一个固定的 125us 信道帧的思想。这就是异步传输模式的关键思想。每个对话和每个数据的传输都由一系列离散的大小为 53 个字节的单元（cell）组成，我们可以对这些单元进行独自管

理和路由传输，最佳地利用只要是可用的带宽。

正如上面所述，CCITT 最终在 1988 年完成了 ISDN 的规范，实际上所有的成员都知道，这种技术在它使用之前就已经陈旧过时了。与此同时，人们立即开始了构思设计下一代的高速数字载体的工作。CCITT 意识到通信的未来在于能够将语音、数据和实时视频的信息传输集成到某个单一的载体服务中。这些不同类型的服务至少要求有 150Mb/s 的传输带宽，这远远超过了 CCITT 所制定的 ISDN 规范的容量范围。于是，CCITT 决定把下一代的数字服务称为宽带 ISDN（broadband ISDN，B-ISDN），用来区别于已经被取代的传统的（窄带）ISDN。

在设计上，宽带 ISDN 要向下兼容 ISDN。实际上，宽带 ISDN 使用同一个参考模型，如图 11-30 所示。异步传输模式（ATM）是实现 B-ISDN 的首选体系结构。异步传输模式可以直接支持 3 种传输服务模式：速率为 155.52Mb/s 的全双工通信模式，速率为 622.08Mb/s 的全双工通信模式，和一种异步模式。这种异步传输模式的上传速率（传到网络上）为 155.52Mb/s 和下传速率（从网络传出）为 622.08Mb/s。

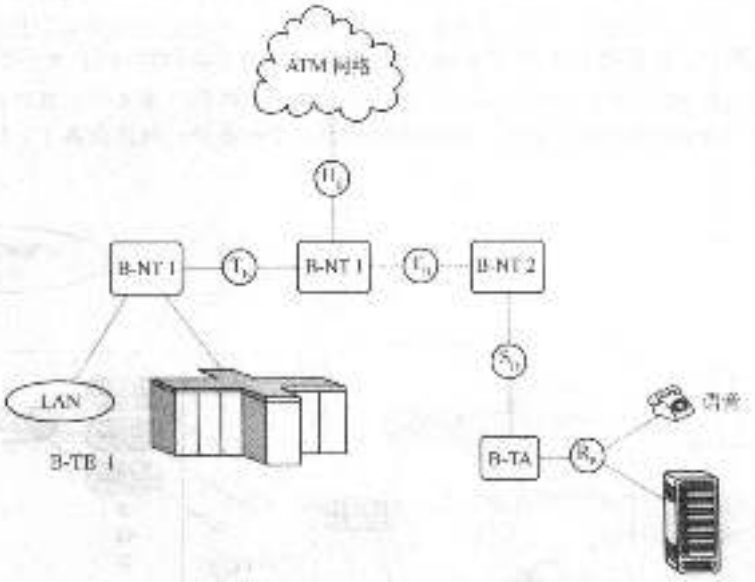


图 11-30 宽带 ISDN 的参考模型

异步传输模式（ATM）可以在 53 字节单元的有效载荷中携带数据、音频或视频的传输信息。这些小的单元（或物理协议数据单元，PDU）允许使用相对简单的硬件设备来处理交换和路由事宜。通过使用虚拟路径，可以简化数据单元的路由路径。虚拟路径是在单个易于管理的数据流中上接入几个虚拟的连接。实际上，任何一种介质：双绞线、同轴电缆或光纤，都能够支持速度范围在 44.736Mb/s 和 155Mb/s 之间的 ATM 信号。

最初，ITU-T 并没有看到 ATM 有任何优势，认为它不过是一个宽区域载体服务。但是，实际上 ATM 适用于任何较高层次的协议。ATM 能够以不同的速度传输各种不同的协议。由于这些原因，ATM 论坛（网络设备供应商的一个协会），从 20 世纪 90 年代初，就一直致力于把 ATM 推广应用到私营的网络。ATM 极有可能成为一个伟大的统一者，为各种型号和不同速度的网络提供一种单一的网络连接技术，包括 LAN、MAN 和 WAN。但是，在普及 ATM 技术之前，它的安装费用必须大幅度地降低，才能够与当前的网络技术相竞争。当在同一个介质上传输数据、音频和视频的价值得到体现时，ATM 作为一种唯一的高比特率的长距离网络技术将占统治地位。

11.8 因特网的概况

第11.3节介绍了因特网如何从开始时期的一个封闭的军用研究网络演变为今天的全球通信的基础设施。但是，不幸地，访问因特网并不是像使用拨号语音电话那样的简单。大多数个人和企业用户都要通过私营的网络服务提供商（internet service providers，ISP）才能连接到因特网。每个网络服务提供商都拥有一个交换中心，称为网络接入点（point of presence，POP）。许多网络服务提供商都有一个以上的POP。因特网的用户通常可以通过域名来知道这些POP，POP的域名一般都是以.com、.net、.biz或者是以类似.uk、.de这样的国家代码结束的。有些POP通过高速线路（T-1或更高）连接到一些区域性的POP或其他主要中介POP。大体上说，从底端到顶端（bottom to-top）的连接分层结构是终端（PC机和工作站）连接到本地的ISP，然后连接到区域性的ISP，最后连接到国家和国际性的ISP（通常称作国家主干网络提供商，National Backbone Providers，NBP）。这个体系结构很容易添加新的分支，就像很方便地增加新的层次一样。NBP本身也必须相互连接在一起，这种NBP的连接是通过网络访问点（network access points，NAP）来实现的，NAP是一些区域性的ISP之间相互连接所使用的一种特殊的交换中心。在美国，一些当地的POP把它们的分支信息传输主要集中到T-3或OC-12线路上，这些线路再与某个NAP相连接。这种ISP POP NAP的分层体系结构如图11-31所示。每个ISP和POP都是根据通过NAP的连接线路上的信息量来支付其网络使用费。信息量越大，每个用户支付的费用也就越多，很多的网络收入又被投资到网络设备上，以维持更好的因特网客户服务。

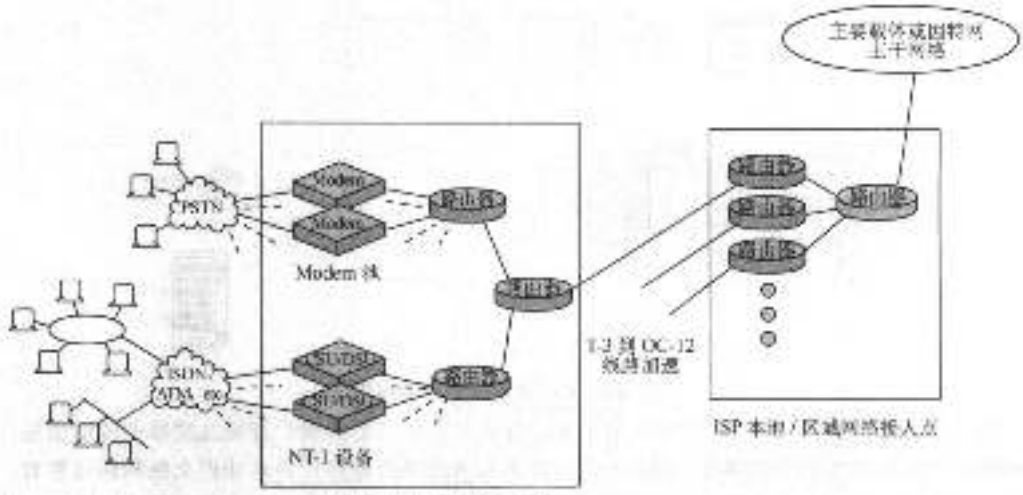


图 11-31 因特网的分层结构

11.8.1 走进因特网

一些主要的因特网用户，例如大公司、政府机关和学术机构，有能力承担租用位于用户和ISP之间的直接高容量数字线路的费用。但是，这些线路租用的费用远远超出了个人和小型企业所能承受的范围。这样一来，需要中等带宽的网络用户通常使用标准电话线来满足他们的通信需求。因为标准电话线只能够传输模拟（语音）信号，所以利用电话线传输计算机产生的数字信号之前，首先必须进行信号转换，或称为调制（modulated），将数字信号转换成模拟信号。在接收端，这些信号又必须被解调（demodulated），从模拟信号转换回数字信号。这种信号转换设备称之为调制器/解调器，或简称调制解调器（modem）。大多数的家用计算机都配有内置的调制解调器，这些调制解调器直接连接到I/O系统总线。

传统拨号线路和调制解调器

音频波段的电话线路能够传输信号的频率范围是：300～3300Hz，传输线路的总带宽为 3000Hz。1924 年，信息理论学家奈奎斯特证明了没有信号能够以比其频率快 2 倍的速率传输信息。用公式表示如下：

最大数据传输率 ＝ 2 ×（带宽）× $\log_2$（信号电平数）波特

其中，波特是线路的信号发送速度。

因此，从数学上来说，3000Hz 的信号能够传输的二进制数据的最高传输速率为 6000 波特。

1984 年，香农扩展了奈奎斯特的理论，考虑线路中存在噪声的情况，公式中使用线路的信噪比（signal-to-noise ratio）。结果表示为：

$$\text{最大数据传输率} = \text{带宽} \times \log_2\left(1+\frac{\text{信号}}{\text{噪声}}\right)\text{b/s}$$

公用交换电话网（public switched telephone network，PSTN）的典型信噪比为 30dB。根据香农的结果，语音波段的电话线路的最大数据传输率近似等于 30 000b/s，而不考虑所使用信号电平的数目。有些调制解调器使用的数据压缩技术可以将吞吐量提升到 56Kb/s，但是压缩数据的传输速率仍然不能够超过 30Kb/s。具有讽刺意味的是，因为大多数调制解调器使用的是从 Ziv-Lempel 算法衍生出来的 ITU-T v42. bis 的压缩标准，所以当利用这类 Modem 下载一个已经压缩的文件时，例如 JPEG、GIF 或 ZIP 文件，吞吐量会显著地下降。回顾一下第 7 章的内容，如果信息中包含的冗余不足够时，基于字典的压缩方案实际上会导致压缩后的文件被扩大，就如同文件已经被压缩的情况一样。

调制（modulating）一个数字信号到模拟载体上，就意味着需要改变这个模拟载体的某些特性，这样模拟信号才可以传输数字信息。改变振幅、改变频率或者改变信号的相位都能够产生一个数字信号的模拟调制。这些调制的形式如图 11-32 所示。图 11-32a 表示一个没有调制的载波信号，频率不能超过 3000Hz。图 11-32b 表示的是使用信号振幅变化方式（幅度调制）进行调制后的同一个载波，载波传输的是 ASCII 编码字符“K”。图 11-32c 和图 11-32d 分别表使用频率调制和相位变化调制方式进行调制后的同一个位组合模式。在图 11-32d 中，通过信号的相移 180 度来区分二进制信号的 1 和 0。这种调制方法有时也称为相移键控（phase shift keying）。

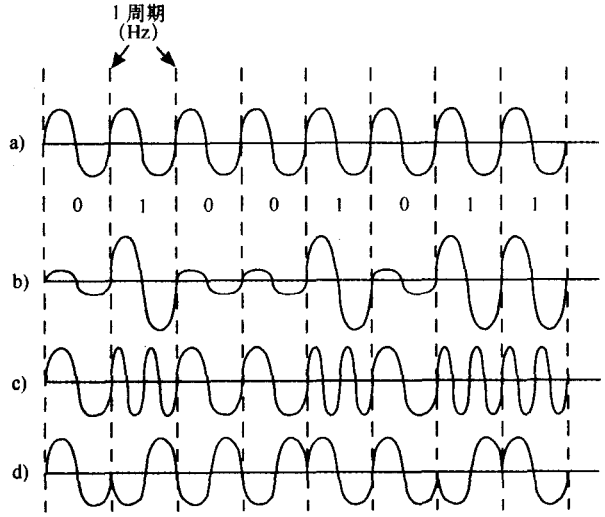
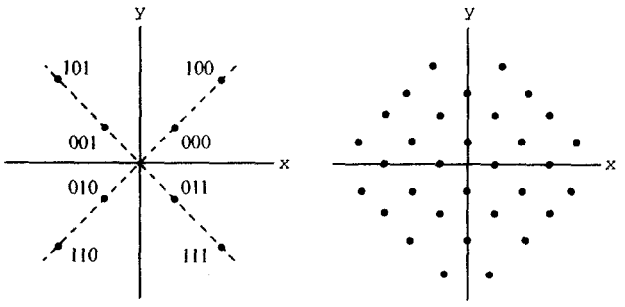


图 11-32 a）一个简单的载波信号
b）字母“K”的载波的幅度调制
c）字母“K”的频率调制
d）使用 180 度相位变化的相位变化调制

从实用的角度出发，使用简单的振幅调制、频率调制或相位变化调制方法最大的吞吐量大约为 2400b/s。为了突破这个限制，调制解调器的设计者们每次改变信号的两个特性而不只是改变一个特性。其中一种调制方法称为正交幅度调制（quadrature amplitude modulation，QAM），同时改变载波信号的相位和振幅。QAM 使用彼此之间相位相差 180 度的两个载波信号。读者可以认为这两个信号的其中一个是正弦波，而另一个是余弦波。利用 X，Y 坐标平面来描述编码的指定的位组合模式的信号点（signal point）。因此，对于 Y 坐标调制正弦波，对于 X 坐标调制余弦波。将正弦波和余弦波叠加在一起就产生一个单一的模拟波，这种信号可以非常方便地沿着线路进行传输。

通过调制正弦波和余弦波在笛卡儿坐标平面上描述的信号点的集合称为信号星座图（signal constellation）或信号点阵（signal lattice）。利用平面上的每个点阵点都可以对几个二进制的位进行编码。图 11-33a 表示每波特上有一个 3 位的虚构编码。在这里，相位变化 90 度，或者信号幅度的一个变化就可以从一种位组合模式变成另一种位组合模式。调制解调器的速度越快，每波特传输的位就越多，而信号星座图也就变得越密。图 11-33b 表示一个使用格码调制（trellis code modulation，TCM）技术的调制解调器的信号点阵图。TCM 与 QAM 非常类似，但是 TCM 在每个信号点增加了一个奇偶位，允许在传输信号中进行向前纠错。



a) 正交幅度调制　　b) 格码调制

图 11-33　正交幅度和格码调制

调制（和解调）一个数字信号只是调制解调器的一部分工作。调制解调器是连接计算机主机的二进制与公用电话线路的模拟世界的一座桥梁。调制解调器通过与主机系统以及线路另一端的调制解调器进行协议交换来控制连接。因为从本质上来说信号流只有一个通道，所以模拟传输必须是串行的、异步的传输。最老和最持久的串行协议是 IEEE RS-232-C 标准。另外，还有其他标准，例如 EIA 232-D、RS-449、EIA530、ITU-T V24 和 V28 协议。所有的协议都由他们相应的标准化组织在官方出版的文献中进行了详细的介绍，但是异步串行通信的基本思想，自从 1969 年 RS-232-C 标准首次发布以来就一直没有改变过。

从正式标准来说，RS-232-C 是由一个标准的 25 针的“D”型连接器上命名为 24 根线路组成。调制解调器在调制信号到达传输介质之前，会控制主机与调制解调器之间的数据流。在标准的文献中，将调制解调器称作为数据通信设备（data communications equipment，DCE），计算机主机被称为数据终端设备（data terminal equipment，DTE）。这两个名称反映了编写原始标准的技术时代。在 RS-232-C标准中定义的 24 根线路中，只有其中的 8 根是大多数串行通信中所必须的。因此，一些国外的调制解调器的制造商在调制解调器和主机系统之间使用一个 9 针的“D”连接器。25 针和 9 针连接器的插脚引线如表 11-3 所示。

表 11-3 IEEE RS-232-C 串行连接器的插脚引线和线路说明

DB-25针	DB-9针	线路名称	说明	来源
2	3	BA	发送的数据（TxD）	DTE
3	2	BB	接收的数据（RxD）	DCE
4	7	CA	发送请求（RTS）	DTE
5	8	CB	清除发送（CTS）	DCE
6	6	CC	数据设备就绪（DSR）	DCE
8	1	CF	载波探测（CD）	DCE
20	4	CD	数据终端就绪（DTR）	DTE
22	9	CE	振铃指示（RI）	DCE

要进行一个对外呼叫，调制解调器首先要接通线路（就如同拿起电话），并且听到一个拨号音。如果线路已经连通，则调制解调器会发送其载波检测信号（Carrier Detect，CD）。收到这个 CD 信号后，主机会提供要拨号连接的号码，而调制解调器会将拨号信号放到线路上。通常这个信号是以拨号音的形式存在的。如果接收端的调制解调器使用正确的应答音调回复了这个呼叫，那么两个调制解调器都将发送它们的数据设备就绪信号（Data Set Ready，DSR）。一旦启动的 DTE 发现 DSR 信号，则它就会发送一个准备发送指令（Ready To Send，RTS）。当调制解调器注意到主机已经发送了 RTS 信号时，它将会发送清除发送（Clear To Send，CTS）和数据载波检测（Data Carrier Detect，DCD）信号做出应答。

在整个数据传输会话过程中，当数据填满缓冲器和较高层的协议执行误差检测与检查点程序时，RTS、CTS 和 DSR 信号可能会被升起和降低。一旦这个对话过程完成，启动的主机就会关闭其 RTS 和 CTS 信号，造成调制解调器在线路上呈现“挂起”的状态。当另一端的调制解调器注意到连接已经被终止时，它就会断开载波连接，并降低其 DCD 信号让其主机知道这个会话过程已经被终止了。

RS-232-C 连接可以用于许多不同种类的低速数据传输。在微型计算机的早期，实际上所有的打印机都是串行连接到主机系统，并且都是 RS-232-C 连接。甚至今天，有时人们还用零调制解调器（null modem）来从一个系统到另一个系统转移文件。零（空）调制解调器只有一对 9 针或 25 针的电缆连接器，如表 11-4 所示。使用这种配置结构，两个系统都会被以为在它们之间存在有一个调制解调器，而实际上并没有调制解调器。利用这种方式，两台计算机之间可以交换大量的信息，而无需使用磁盘、磁带或者网络资源。

表 11-4 Null modem 插脚引线

线路	本端 DB-25针	去到 DB-25针	本端 DB-9针	去到 DB-9针
地线地	1	1		
TxD-RxD	2	3	3	2
RxD-TxD	3	2	2	3
RTS-CTS	4	5	7	8
CTS-RTS	5	4	8	7
信号地	7	7	5	5
DSR-DTR	6	20	6	4
CD-DTR	8	20	1	4
DTR-DSR	20	6	4	6
DTR-CD	20	8	4	1

尽管许多家用计算机都配备了内置调制解调器，但是网络服务提供商和其他数据服务公司仍有大量的拨号上网的用户，通常这些用户使用的是由几十个、甚至上百个调制解调器组成的调制解调器库。

这种调制解调器库配备多根入站线路。如果某根特定的线路忙，那么网络连接就会从一根入站线路“滚动（roll）”到下一个入站线路，直到找到一个空闲线路。这种连接转换的发生不需要呼叫者的应答或积极参与。只有当所有的入站线路都忙的时候，才会发送忙信号。

数字用户线路

香农定律对模拟电话调制解调器的限制速率为 30Kb/s，这种限制使得要承诺无边界的网络世界对任何一个销售商都开放会遭遇到一个重大障碍。近几十年来，虽然长途电话链接已经变得快速和数字化，但是从电话交换中心到用户之间的本地回路连接电路仍然继续使用着上百年老式的模拟技术。事实上，“最后一里”的本地回路所跨越的距离非常长，要将从前的模拟电话服务带入到当今的数字世界的代价是极其昂贵的。

幸运地，电话线中的物理导体有足够厚，可以支持几英里的中速数字信息传输，而不会产生太大的衰减。意识到这一点，通信集团已经开发一些技术，为住宅用户和小型企业提供比较廉价的数字服务。*数字用户线路*（Digital Subsriber Line，DSL）就是这样的一种技术，它能够与*简易老式电话业务*（plain old telephone service，POTS）共存，利用同一根双绞线传送数字信息。目前，大多数 DSL 服务只适用于到中央电话交换局（CO）的连接铜线少于 18000 英尺（5460 米）的用户。当然，这并不是说只要客户在中央电话交换局距离 18000 英尺的范围内就行了，因为服务电缆的路由路径很少是直线。

传统模拟电话业务，POTS，在某个框架（frame）内，或者说是在中央电话交换局的交换中心终止本地环路业务。从一个交换中心到另一个交换中心都有专用电路，也许这个专用电路连接到一个长途中继线上，也许是直接从交换中心出来又回到同一个本地回路。所有传统的电路交换都建立在模拟连接的特征基础上。

正如在 ISDN 讨论中所解释的一样，模拟中央电话交换局的配置和数字技术是不兼容的。与这些模拟电话转换开关等价的数字产品是 DSL 访问*多路复用器*（DSL access multiplexer，DSLAM）。DSLAM 组合来自多个用户线路的 DSL 通信量集中到交换以太网，T-1/E-1，T-3/E-3 或者是 ATM 的载体上，来实现对因特网或其他数字服务的访问。有些 DSLAM 还可以连接到 POTS 的交换框架上，提供语音和数字服务。

根据用户的要求，某些 DSL 会使用一个分配器把语音从数字信息流中分离出来。数字信号会在一个被称为 DSL 调制解调器的编码器或解码器设备上终止传输。

DSL 使用两种不同的且不兼容的调制方法：*无载波振幅相位*（Carrierless Amplitude Phase，CAP）和*离散多音频服务*（Discrete MultiTone Service，DMT）。其中，CAP 是一种比较老且简单的技术，而 DMT 对 DSL 来说是一种 ANSI 的标准。

CAP 使用三种频率范围：0 到 40KHz 是为语音服务的，25KHz 到 160KHz 是为了“*上传*（upstream）”通信量（比如通过浏览器发送一个命令来请求浏览特定的 Web 页），240KHz 到 1.5MHz 是为了“*下传*（downstream）”通信量（递送所请求的 Web 页）。这种不平衡访问方法称为*非对称数字用户线*（Asymmetric Digital Subsriber Line，ADSL）。因为大多数的 Web 会话过程涉及大量的下传通信量而不是上传通信量，所以 ADSL 十分适合于家庭和小型企业的因特网访问。

当然，由于 CAP 使用固定大小的信道，CAP 的用户就会被锁定在一个 135KHz 的上传带宽上。这对于希望上载大量文件或者要求远程连接到一个局域网（LAN）上的用户来说，可能就不是理想的情况了。在这些情况下，DMT DSL 可以提供更好的性能。DMT 把一个 1MHz 频率的带宽分割成 256 个 4KHz 的信道，称为*音调*（tone）。这样的信道能够以同时适合于用户和供应商的任何方式进行配置。例如，DMT 可以很方便地接纳某个需要 192 个 4KHz（768KHz）的上载信道和仅要求 64 个 4KHz（256KHz）的下载信道的用户。尽管这种服务是可能的，但是 DMT 更经常的是利用它的灵活性来适应线路质量的波动。当 DMT 设备检测到在某个信道上存在太大的串音或过多的衰减时，它会停止使用这条信道，直到这种情况得到改善。虽然在采取这样的措施后，线路的比特率会降低，但是实际的吞吐量提高了，因为有故障的信道可能会引起多次数据重发的现象，通常这种情况下造成的实

际吞吐量的减少要比损失1个或2个4KHz信道所产生的影响更大。

DSL决不会只限制在居民区使用。高容量的DSL技术可以应用在商业T-1/E-1、T-3/E-3和ATM服务中。在20世纪90年代期间出现了许多DSL技术，相信以后会出现更多这样的技术。至于哪种是"最好的"技术目前还没有一致的定论，估计在今后的十多年时间将会由市场筛选出几种能够生存的技术产品。为了帮助读者在这种混乱的局面中找到合适的解决方案，我们在表11-5中列举了一些流行的DSL技术的简短归纳，供大家参考。

表11-5 ADSL词汇表

技 术	描 述
ADSL	非对称数字用户线，将本地回路传输线分割成不同的带宽，下传带宽比上传带宽大得多。通常支持同一双绞线上的POTS
SDSL	对称DSL，能够在上传和下传数据量中间等分配
VDSL	极高比特率的DSL，为短距离传输而设计，通过光学介质连接两个邻近的网络。VDSL支持4500英尺距离内速度为13.5Mb/s的传输和1000英尺距离内速度为52.8Mb/s的传输
IDSL	通过ISDN的DSL，允许DSL的使用超过18000英尺的限制，但是吞吐量限制为128Kb/s
RADSL	速率自适应的DSL，根据线路质量自我配置传输速率，具有一定的能力可以超越18000英尺限制进行传输（通过降低传输速率）
ReachDSL	另外一种可以突破18000英尺壁垒的DSL，有些供应商声称其ReachDSL服务可以覆盖距离CO 30000英尺的范围
G.Lite	一个为住宅区用户设计的DSL专用版本，除了数据通信外还支持语音通信，上传速度为500Kb/s和下传速度为1.5Mb/s
G.SHDSL	ITU-T的SDSL标准（G.165.2），在各种不同商业层次的DSL中相互使用，它使用两对双绞线支持数据传输速率从192Kb/s到2.3Mb/s的范围

11.8.2 遨游因特网

模拟本地回路仅仅是因特网目前面临的许多问题中的其中之一。另外一个更严重的问题是主干网络的高速链路路由器的交通堵塞问题。在美国，沿着不同的主干网络上有超过50 000个路由器，更新路由表的工作常常是高速链路上的重要流量。路由表的汇总任务也需要花费大量的时间。老旧的线路会长时间阻碍通信流量，当系统试图解决出现的错误时会引起更多的阻塞问题。当路由器出现故障，会产生一些非常严重的问题，出错的路由器会向整个主干网络系统广播错误的路由。这就是大家所熟知的路由器不稳定性问题（router instability problem），并且这还是一个正在继续研究的领域。

毫无疑问，一旦IPv6得到广泛采用，这些问题将不复存在，因为路由表会变得越来越小（假定使用可聚集的全局统一地址）。但是，由于每天有几百万个路由公告发生，所以仍存在许多困难，而且要试图使数万个路由表任何时候都对特网络的看法一致，还存在许多物理限制。最后，如果高速链路的发展要与市场的需求保持同步，则需要进行一些更深层次的分析研究工作。这些工作的结果可能会导致产生新一代路由的协议。可以肯定的是，从长远的观点来说，简单地给因特网增加带宽并不使得网络的速度会变得更快。因特网必须变得更加智能化。

本章小结

本章全面概述了构建数据通信系统中所使用的网络设备和协议。每个网络设备、网络进程，都在一个分层协议堆栈中的某个层次执行相应的任务。网络工程师使用OSI参考模型的不同层次来描述各种网络设备的作用和功能。当一台计算机与另一台计算机进行通信时，在协议堆栈上运行每一层都会与运行在远程计算机系统上相对应的层进行会话。而系统中各个协议层之间也会使用各自的服务访问点与它们相邻的层进行交流。

大部分网络应用都依赖于TCP/IP协议。到目前为止，TCP/IP协议是最广泛应用的数据通信协议。虽然我们经常提到TCP/IP协议，实际上它是这两种协议的组合。TCP提供了一种在不可靠的IP协议的顶部

建立可靠的通信流的方法。IP 第 4 版中的设备受到 32 位地址域的限制。IP 第 6 版将可以解决这个问题，因为 IPv6 的地址字段有 128 位的宽度。由于这些地址字段的宽度比较大，所以路由选择就成了一个很困难的问题。基于这种原因，IETF 设计了一种分层结构的地址方案，即可聚集的全局统一地址格式。这种方法可以使数据包的路由变得更方便更快捷。

本章讨论了许多在大部分数据通信网络中通用的设备。这些设备中最重要的是物理介质和路由器。选择物理介质时必须考虑预期的通信负荷和网络覆盖的距离。需要时，物理介质可以利用转发器来延伸。路由器是用来监视网络的状态的复杂设备。路由器中的内置程序允许路由器为网络通信量选择最优的路径。

公用交换电话网将继续为将因特网连接到家庭用户和小型企业服务。但是，用户和电话交换中心之间的连接还是模拟电路，最大传输速率只有 30000b/s。人们采用一些方法来改善这种状况，ISDN 和 DSL 就是两种适合于家庭用户和小型企业的数字载体服务。

但是，公用交换电话网仅仅是网络发展过程中遇到的障碍之一。另外一个问题是主干网络路由器的交通堵塞。随着因特网作为商用介质不断地呈指数形式增长，路由中遇到的各种问题也会成比例增加。这些问题的最终解决方法可能需要重新思考现在的网络结构，以及目前人们熟知的构成因特网基础的某些基本假定。

深入阅读

关于计算机网络主题的文献很多。但是，现在想找到好的网络资料却不是一件容易的事情。在数据通信方面最适合的书籍包括：Tanenbaum（1996）、Stallings（2000）以及 Kurose 和 Ross（2001）等人的著作。Tanenbaum 是按照 OSI 协议堆栈来组织内容的，读者非常容易地理解有关数据通信和网络的许多重要概念。Kurose 和 Ross 详细讨论了在本章中提出的大部分议题，而且内容的深度可以为大多数有兴趣的读者接受。Stallings 的著作和 Tanenbaum 的著作中的内容基本相同，但是 Stallings 的著作比较严谨和详细。Sherman（1990）同样很好地介绍了数据通信方面的内容（可能会有点陈旧）。Sherman 对数据通信历史发展的看法还是值得阅读的。

有关网络标准（草案）最权威信息的来源可以访问因特网工程任务组的网站 www. ietf. org。以及本章中相关的 RFC 资料为：

- RFC791 “网际协议版本 4（IPv4）”
- RFC793 “传输控制协议（TCP）”
- RFC1180 “TCP/IP 指南”
- RFC1887 “IPv6 统一地址分配的体系结构”
- RFC2460 “网际协议版本 6（IPv6）详细说明”
- RFC2026 “网际标准过程”
- RFC1925 “网络的基本原理”

由 Rodriguez、Getrell、Karas 和 Peschke（2001）合编的 IBM 的 TCP/IP 指南是 IETF 之外最经济最易读的资料。与 IETF 网站的内容不同，这本说明书是以一个特定的供应商的产品为出发点讨论了实现 TCP/IP 的方式，这里不可能有虚构的成分。Minoli 和 Schmidt（1999）讨论了网络的基本设施，重点关注的是服务质量的问题。

Clark（1997）详细全面地论述了有关电话通信（重点集中在英国）的问题。这本书讨论了公用电话交换网的一些重要方面，其中包括电话网络传送数据通信的能力。Burd（1997）有关 ISDN 和 Prycker（1995）有关 ATM 的书籍都是这些方面的权威著作。IBM（1995）的 ATM 指南，虽然没有 Prycker 论述严谨，但是也客观和详细地对 ATM 最突出的特点进行了精辟论述。

要了解有关因特网主干网络路由器的不稳定性问题的更多信息，可以参考 Labovitz、Malan 和 Jahanian（1998）的论文。密歇根大学还设立了一个网站来讨论有关网络性能的问题。网站地址为：www. merit. edu/ipma/。

要保持接触数据网络技术的最新发展的唯一方法是，经常不断阅读有关的专业期刊和商业期刊。在

ACM 和 IEEE 出版的刊物中可以找到最前沿（avant-garde）的信息。IEEE/ACM《Transactions on Networking》和《IEEE Network》是两种非常优秀的专业期刊。行业杂志是获取新信息的另一种好来源，特别是可以了解各个网络提供商如何来实现最新的网络技术。有两种 CMP 出版这类的杂志，分别是网络计算（www. network. computing. com）和网络杂志（www. networkmaganize. com）。《网络世界》是由 CW 通信公司出版的周刊杂志，它不仅提供了高质量的印刷版面，而且还附有相关的网站 www. nwfusion. com，提供大量的信息和资源。

许多设备供应商都在其网站公布了一些很好和真实的学习资料。这些公司网站包括 IBM 公司、Cisco 公司和 Corning Glass 公司。当然，如果读者想探究本章提出的相关主题的一些特定技术，也能够发现其他一些非常好的商业网站。当谈到数据通信问题时，不管学习多么努力，但似乎人们所了解的知识永远都是不够。

参考文献

Burd, Nick. *The ISDN Subscriber Loop.* London: Chapman & Hall, 1997.

Clark, Martin P. *Networks and Telecommunications: Design and Operation 2nd ed.*, Chichester, England: John Wiley & Sons, 1997.

de Prycker, Martin. *Asynchronous Transfer Mode: Solution for Broadband ISDN.* Upper Saddle River, NJ: Prentice Hall, 1996.

IBM Corporation. *Asynchronous Transfer Mode (ATM), Technical Overview 2nd ed.*, Raleigh, NC: International Technical Support Organization (ITSO Redbook), 1995. (www.redbooks.ibm.com)

Kessler, Gary C. & Train, David A. *Metropolitan Area Networks: Concepts, Standards and Services.* New York: McGraw-Hill, 1991.

Kurose, James F. & Ross, Keith W. *Computer Networking: A Top-Down Approach Featuring the Internet.* Boston, MA: Addison Wesley Longman, 2001.

Labovitz, C., Malan, G. R., & Jahanian, F. "Origins of Internet Routing Instability." *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE* , Volume 1 (1999), pp. 218–226.

Labovitz, C., Malan, G. R., & Jahanian, F. "Internet Routing Instability." *IEEE/ACM Transactions on Networking*, 6:5 (Oct. 1998), pp. 515–528.

Minoli, Daniel & Schmidt, Andrew. *Internet Architectures.* New York: John Wiley & Sons, 1999.

Rodriguez, Adolfo, Getrell, John, Karas, John, & Peschke, Roland. *TCP/IP Tutorial and Technical Overview, 7th ed.* Triangle Park, NC: IBM International Technical Support Organization (ITSO Redbook), 2001. (www.redbooks.ibm.com)

Sherman, Ken. *Data Communications: A User's Guide, 3rd ed.* Englewood Cliffs, NJ: Prentice Hall, 1990.

Stallings, William. *Data and Computer Communications, 6th ed.* Upper Saddle River, NJ: Prentice Hall, 2000.

Tanenbaum, Andrew S. *Computer Networks, 3rd ed.* Upper Saddle River, NJ: Prentice Hall, 1996.

基本概念和术语复习

1. 一个轮询网络的组织结构和因特网网络的组织结构有哪些区别？
2. 什么协议设备对于 DARPAnet 网络的基础是关键性的设备？
3. 谁为因特网制定了网络标准？
4. 因特网标准的正式名称是什么？
5. ISO/OSI 参考模型的哪一层负责协议帧大小和传输速率？
6. 如果一个通信会话要使用加密或者压缩，那么 ISO/OSI 参考模型中的哪一层负责完成这项任务？
7. 根据第 11. 5. 1 节中给出的 IPv4 格式，IP 协议数占据了多少位的位置？这个域（字段）的目的是什么？
8. 为什么某些类型的 IP 地址会非常稀少？

9. 解释 TCP 协议的一般用途？
10. IPv6 对 IPv4 如何进行改进的？
11. 导向数据传输介质和非导向数据传输介质的区别是什么？分别举几个例子。
12. 什么因素决定了传输介质的质量？可以采用什么样度量方法？
13. 衰减的主要原因是什么？什么方法有助于减少衰减？
14. 线路中的波特率和比特率的区别是什么？
15. 光缆有哪三种类型？哪一种光缆传输信号的速度最快？
16. 在哪里能够找到 MAC 地址？一个 MAC 地址有多少个字节？
17. 简要描述转发器、集线器、交换机和路由器彼此之间有哪些区别？
18. 网桥和网关之间有什么不同？哪一个速度更快？为什么？
19. 何时不宜使用静态路由方法？
20. 给出链接静态路由和距离矢量路由所不同的两种重要方式？
21. 距离矢量路由产生的三个主要问题是什么？
22. 防火墙采用什么样的方式提供安全？
23. 什么是脉码调制？
24. 什么是时分复用？
25. PDH 和 SONET/SDH 的区别是什么？
26. ISDN 存在什么问题？
27. 使用 ATM 的主要好处是什么？
28. ATM 主要的限制是什么？
29. 相位变化调制是如何工作的？
30. 格码调制和正交幅度调制有如何的区别？
31. DSL 主要的限制是什么？
32. DSL 有哪两种类型？哪一个是 ANSI 标准？
33. 描述路由器的不稳定性问题。

练习题

1. 早期的企业计算机网络的通信业务和早期的科学学术网络的通信业务有什么区别？现在，这两种类型的系统还有这样的区别吗？
2. 为什么 ISO/OSI 协议堆栈被称作参考模型？你认为将来也是这种情况吗？
3. 如何区别网络层协议和传输层协议？
4. 网际协议标准是通过世界上许多人的共同努力设计出来的，而不管这些人在数据通信领域有任何特殊的背景。在另一方面，专用协议却都是由少数人创建的，这些人都直接或间接地为同一个雇主工作。
   a) 你认为每一种方法提供的优点和缺点是什么？哪一种方法可以产生更好的结果？而哪一种方法可以更快地产生结果？
   b) 你认为为什么 IETF 的方法会在与专用方法的竞争中取得优势？
◆5. 在描述 TCP 头部窗口字段时，我们曾经说到：
   如果发现接收方的应用程序运行得非常慢，比如每次只从缓冲器拿出一个或两个字节的数据，那么运行在接收端的 TCP 进程就应该等待，直到应用程序的缓冲器有足够的空间可以发送另一个报文段。请问：
   判断可以发送另一个报文段的理由是什么？
6. OSI 协议堆栈除了应用层外，还包括会话层和表示层。TCP/IP 的应用程序，例如 Telnet 和 FTP，没有这种分立的定义层次。你认为需要有这样的分隔吗？给出将 OSI 方法应用到 TCP/IP 中的一些优点和缺点？

7. 一个 TCP 报文段的长度为什么要限制为 65515 个字节？

8. 为什么 IETF 要使用字 8 位位组而不是字节？你认为这种做法应该继续实行吗？

◆9. 下面的 IP 地址分别属于哪类网络？

◆a) 180.265.14.3

◆b) 218.193.149.222

◆c) 92.146.292.7

10. 下面的 IP 地址分别属于哪类网络？

a) 223.52.176.62

b) 127.255.255.2

c) 191.57.229.163

11. 运行 TCP/IP 协议的某个站点需要向主机传送一个文件。这个文件包含 1024 字节。问：在传输文件时将传送多少字节？其中包括全部的 TCP/IP 开销，有效载荷的大小是 128 个字节，两个系统都运行 IPv4（同时还假定三次握手和窗口大小的协商已经完成，而且在传输过程中没有发生错误）。请问：

◆a) 协议开销的大小是多少（用百分比表示）？

◆b) 如果假定两个客户端都使用 IPv6，进行与上面相同的计算？

12. 运行 TCP/IP 协议的某个站点需要向主机传输一个文件。这个文件包含 2048 字节。问：在传输文件时将传送多少字节？其中包括全部的 TCP/IP 开销，有效载荷的大小是 512 字节，两个系统都运行 IPv4（同时还假定三次握手和窗口大小的协商已经完成，而且在传输过程中没有发生错误）。

a) 协议开销的大小是多少（用百分比表示）？

b) 如果假定两个客户端都使用 IPv6，进行与上面相同的计算？

◆13. 运行 TCP/IP 协议的两个站点要传输一个文件。文件长度为 100KB，有效载荷的大小为 100 字节，协商的窗口大小为 300 字节。发送方从接收方收到一个 ACK 1500。

◆a) 下面将发送哪些字节？

◆b) 如果接收方没有发送一个 ACK 信号，那么可以发送的最后一个字节数是多少？

14. 运行 TCP/IP 协议的两个站点要传输一个文件。文件长度为 10KB，有效载荷的大小为 100 字节，协商的窗口大小为 2000 字节。发送方从接收方收到一个 ACK 900。

a) 下面将发送哪些字节？

b) 如果接收方没有发送一个 ACK 信号，可以发送的最后一个字节数是多少？

15. 如果 TCP 不允许发送方和接收方协商一个超时窗口，那么将会出现什么问题？

16. IP 是一个无连接协议，而 TCP 却是一个面向连接的协议。这两个协议如何在同一个协议堆栈中共存？

17. 在第 11.6.1 节中谈到，当使用 4B/5B 编码时，一个比特率为 100Mb/s 的传输介质要求 125MHz 的信号传送容量。

a) 如果使用曼彻斯特（Manchester）编码将需要多大的信号传送容量？

b) 如果使用改进的频频率调制（MFM）编码将需要多大的信号传送容量？假设 0 和 1 发生的几率相等。

（曾在第 2 章中对曼彻斯特编码做了解释）

18. a) 对于某种特定类型的网络连线来说，信号功率是 8733.26dB，而且在这个特定的信号强度和 100MHz 传输频率下，噪声等级为 41.8dB。试计算这个导体的信噪比。

b) 假定 a 部分的网络连线的噪声等级为 9.5dB，当传输一个 200MHz 的信号时噪声等级为 36.9dB。信号强度是多少？

◆19. a) 对于某种特定类型的网络连线来说，信号功率是 2898dB，而且在这个特定的信号强度和 100MHz 传输频率下，噪声等级为 40dB。试计算这个导体的信噪比。

b) 假定 a 部分的网络连线的噪声等级为 0.32dB，当传输一个 200MHz 的信号时噪声等级为 35dB。信号强度是多少？

20. 一个物理 PDU 有多大？这个问题的答案决定了在许多网络体系结构中同时传输的数目。如果一个信号

通过铜导线传播的速率是 2×108 米/秒，然后在一个以速率为 10Mb/s 运行载波，每位脉冲长度由下式给出：

$$\frac{\text{传播速度}}{\text{总线速度}}=\frac{2\times10^{8}\text{m/s}}{10\times10^{6}\text{b/s}}=20\text{ 米/位}$$

如果数据帧为 512 位长，那么整个帧占据的长度为：

（每位长）×（帧大小）=20×512=10240 米

a）如果网络运行速度 100Mb/s，那么一个 1024 位的数据包有多大？

b）如果网络速度增加到 155Mb/s，其结果又是多大？

c）在速度为 100Mb/s 时，通过网络中的某个特定点时需要多长时间？

21. 看起来在图 11-14 中的 4B/5B 位单元是非常小。实际上，在一个 125MHz 的链路上这样一个位单元到底有多长？（使用上一个问题的常数和公式。）
22. 参考图 11-21，假定路由器 4 要从路由器 3 和路由器 1 的路由表中推演出自己的路由表。使用与其他 3 个路由器的路由表的相同的格式，完成路由器 4 的路由表。
23. 因为已经使用了格码和其他相位变化技术来增加普通电话线路的信号传输容量，所以我们难道不可以在数字线路上做同样的事情吗？
24. 在 11.8.1 节中曾经介绍，利用 Shannon 定律，可以知道一个标准模拟电话线路的最大数据速率大约是 30 000b/s，而信噪比是 30dB。这里，信号噪音的比值为 1000，正如在 11.6.1 节中所解释的，信噪比是 $10\log_{10}$信号 dB/噪声 dB。

    如果通过一个信噪比为 20dB 的 10KHz 信道发送二进制信号，那么可以获得的最大数据速率是多少？
25. 使用与上一个问题相同的思想，假定一个信道的计划容量是 10Mb/s，带宽是 4MHz。要获得这样的数据速率，信道的最小信噪比是多少？（用 dB 表示）
26. 构建一个时序图来说明两个调制解调器如何使用 RS-232-C 协议建立一个连接。
27. 北美地区的 TDM DS-0 帧通过线路上的某个点需要 125μs。欧洲等价的数据帧通过线路上的某个点需要多少毫秒？
28. 图 11-33b 说明了一个 TCM 的星座图。设计一个位串编码用于这个星座图。

    （提示：这 32 个信号点可以编码 4 位长的位串。第 5 位用作奇偶位。在这个练习中，既可以采用奇校验，也可以采用偶校验。）
29. 设计一个格码调制的点阵图和每波特可以编码 4 位的编码方案（包括校验位）。

当人们都享行前人栽树，后人乘凉时，一种文明就繁荣起来了。
——希腊寓言

附录 A　数据结构和计算机

A.1　概述

在整个教材的学习过程中，我们认为读者是理解计算机数据结构的基本知识的。当然，这些知识对于阅读本书并不是必须的，但是它对读者更加细致地掌握计算机组成原理和体系结构的某些知识点是非常有帮助的。本附录的目的是想对已经具备数据结构基本知识的读者做一些知识词汇的扩展。对于以前学过数据结构的读者，本附录也适合作为一个复习材料。基于这些思想，本部分的介绍将尽量简短，而且主要倾向于硬件方面的考虑。如果读者有兴趣进一步学习数据结构，建议阅读本附录结尾处所引用的参考文献。当读者学习本附录时，应该意识到在这些例子中使用的所有的存储器地址都采用十六进制数。如果对十六进制数还没有很好理解，读者可以先阅读本书第 2 章的内容。

A.2　基本结构

A.2.1　数组

数据结构（data structure）的术语是信息组织的一种方式。有了合适的数据结构，计算机的执行过程可以在需要时非常方便地访问各种数据。数据结构通常与数据结构的实现无关，因为数据的组织方式是逻辑上的，而不必是物理上的。

最简单的数据结构是线性数组。从程序设计的经验中读者可能知道，线性数组是计算机存储器中的一块连续区域，一般在程序中为这个区域指定了一个名字。存储在这个区域中的实体群组必须是同构的（即这些实体必须具有相同的大小和类型），而且能够被单独编址，通常使用下标进行编址。例如，假定有如下的 Java 声明：

```
char [] charArray[10];
```

操作系统会给变量 charArray 分配一个存储器的值，表示数组的基址（base address，或称为起始地址）。计算机可以通过距离这个基础位置的偏移量来访问后续的字符。利用数组原始数据类型的大小来递增偏移量，在这里，数组是字符数组 char。Java 中的字符是 16 位宽，所以字符数组每个数组元素的偏移量是 2 个字节，例如，字符数组 charArray 结构存储在地址 80A2，程序声明：

```
char aChar = charArray[3];
```

在存储器地址 80A8 位置处取回 2 个字节。因为 Java 数组的下标编号是从 0 开始，所以刚才已经将数组中的第 4 个元素存储到了字符变量 aChar 中：

80A2＋2 字节/字符×距离基址开始的 3 个字符偏移量＝80A2＋6＝80A8

二维数组是由一维数组组成的线性数组，因此存储器中的偏移量值必须要考虑数组的行的大小和原始的数组数据类型的大小，例如，考虑下面的 Java 声明：

```
char[] charArray[4][10];
```

这里，我们定义 4 个线性数组，每个线性数组有 10 个存储器空间。但是，如果把这个结构当作是一个 4 行 10 列的二维数组结构来考虑，问题会更简单。如果字符数组 charArray 的基址仍然是

80A2，那么字符数组元素 [1] [4] 将位于地址 80BE 位置处。这是因为数组的第 0 行占据了从 80A2 到 80B5 的地址位置。数组第一行的起始地址是 80B6，而我们正要访问的是第二行的第 5 个元素：

80B6＋2 字节/字符×距离基址开始的 4 个字符偏移量＝80B6＋8＝80BE

当解决了程序问题，允许程序在数组的存储位置的一个小的子集中返回地址时，数组存储方式是一个很好的选择。如果我们编写一个十五子棋的游戏程序，就是上述的情况。例如，每个“点”就是“棋盘”数组上的一个位置。这个程序会在每次允许移动之前，只检查对骰子所掷的数字是合法移动的棋盘点的位置。

数组的另外一个好的应用是基于时间天数或月天数的数据收集任务。例如，我们可能计数每天在不同的时间通过高速公路某一特定点的车辆的数目。如果以后需要统计在上午 9：00 到 9：59 之间的平均交通流量，那么只需要对这段时间内收集的数据，即每天 24 小时数组的第 10 个元素求平均值（午夜到上午 1：00 是第 0 个元素）。

A.2.2 队列和链表

当处理一些响应服务要求的项目时，数组通常不是非常有用的数据结构。通常情况下根据请求的时间来处理服务要求。换句话说，先来先服务。

下面考虑一个 Web 服务器，处理来自因特网用户的超文本传输协议（Hypertext Transfer Protocol，HTTP）的请求。可以将这些服务请求序列整理出来，如表 A-1 所示。

表 A-1 一个 Web 服务器的 HTTP 请求

时 间	源 地 址	HTTP 命令
07:22:03	10.122.224.5	http：//www. spiffywebsite. com/sitemap. html
07:22:04	10.167.14.190	http：//www. spiffywebsite. com/shoppingcart. html
07:22:12	10.148.105.67	http：//www. spiffywebsite. com/spiffypix. jpg
07:23:09	10.72.99.56	http：//www. spiffywebsite. com/userguide. html

毫无疑问，我们可以把每个请求都放到一个数组里面，然后在准备服务下一个请求时，搜索数组查找最低的请求时间值。然而，这个执行过程的效率非常低，因为数组的每个元素每次都需要被询问。而且，如果某天的交通流量非常大，则可能出现数组存储空间溢出的风险。由于这些原因，队列是一种非常适合于先来先服务的数据结构。队列数据结构要求元素按照队列的顺序依次排队。银行和超市排队等候服务就是很好的队列的例子。

实现队列可以有不同的方式，但是所有的队列实现都需要有 4 个组成部分：一个用来指向队列的第一个元素（队头，head）的存储器变量，一个用来指向队列的最后一个元素（队尾，trail）的存储器变量，用来存放队列的各个元素的存储器空间和一组针对这种队列数据结构的操作。队头指针指示下一个将要服务哪一个元素。队尾指针在队列的末尾增加元素时非常有用。如果队列的队头指针为空（0），则表示这个队列是空队列。队列的操作通常包括在（正在排队）列表的末尾添加一个条目，在列表的开始处删除一个条目，以及检查队列是否为空。

实现一个队列最常用的方法是使用一个链表（linked list）。在链表中，队列中的每个元素都包含一个指示队列中下一个元素的指针。当元素出队列时，队头指针会使用在刚刚被删除的节点处找到的信息来定位下一个节点。因此，在上面介绍的 Web 服务器的例子中，在第一个元素被服务（这个元素就从这个队列中被删除）后，队头指针将指向第二个数据元素。

现在考虑上面的例子中的表 A-1，假设这里的队头地址为 7049，这个地址中存放的是第一个 HTTP 请求，www. spiffywebsite. com/sitemap. html。队头指针的值为 7049。在此存储器地址 7049 的位置，有如下的条目：

07：22：03，10.122.224.5，www. spiffywebsite. com/sitemap. html，70E6

其中，70E6 是下面的元素的地址：

07：22：04，10.167.14.190，www.spiffywebsite.com/shoppingcart.html，712A。

这个队列的全部内容如表 A-2 所示。

表 A-2 在存储器中一个 HTTP 请求队列的实现

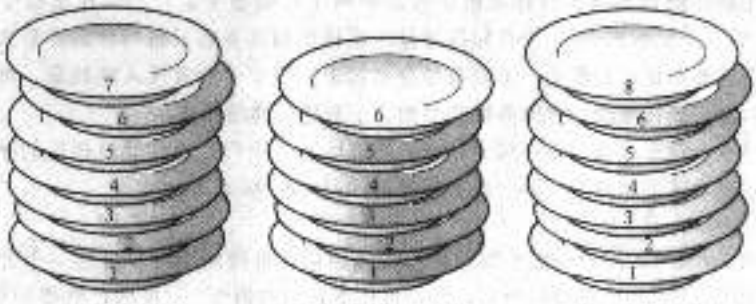
存储器地址	队列数据元素			指向下一个元素的指针
7049	07:22:03	10.122.224.5	http://www.spiffywebsite.com/sitemap.html	70E6
...	...	...	...	...
70E6	07:22:04	10.167.14.190	http://www.spiffywebsite.com/shopping-cart.html	712A
...	...	...	...	...
712A	07:22:12	10.148.135.67	http://www.spiffywebsite.com/spiffypic.jpg	01B3
...	...	...	...	...
01B3	07:23:09	10.72.94.88	http://www.spiffywebsite.com/userguide.html	null

队头指针设置为 7049，队尾指针设置为 01B3。如果有另一个用户的请求到来，那么系统就会在存储器中为这个新的请求找到一个存放空间，并且更新最后一个队列元素（指向这个新的条目）以及队尾指针。读者应该注意到，当使用这种指针结构时，与数组不同，它并不要求数据元素在存储器中是连续存放的。这样一来，在需要时就允许这种结构去扩大添加元素。另外，它也不要求地址值是递增的。队列元素可以存放在存储器中的任何位置。这些指针会保持队列的顺序。

我们可以对上面介绍的队列结构进行修改，创建一个固定大小的队列（通常称为循环队列，circular queue），或者是创建一个优先级队列（priority queue）。在优先级队列中，某种类型的条目可以优先于其他条目。即使有这些增加的内容，队列也还是很容易实现的数据结构。

A.2.3 堆栈

队列有时也称为先进先出（first-in，first-out，FIFO）列表，理由是非常明显的。某些应用却需要相反的顺序，或者说后进先出（last-in，first-out，LIFO）。堆栈（stacks）就是符合 LIFO 顺序的数据结构。堆栈名字的由来非常类似于在自助餐中向客户提供盘子的方式。自助餐的服务人员把一些热的、湿的和干净的盘子放到一个装有弹簧管状容器的顶部，而把冷的和干燥的盘子下压到管状容器的底部。下位顾客会从这种堆栈的顶部取走一个盘子。这种顺序过程如图 A-1 所示。图 A-1a 表示的是一堆盘子，1 号盘是第一个放到堆栈中的，7 号盘是最后一个放到堆栈中的。而 7 号盘是第一个从堆栈中取走的盘子，如图 A-1b 所示。当下一个盘子到来，也就是 8 号盘，将会被放到堆栈的最顶



a) 初始的堆栈　b) 移走 7 号盘（出栈）　c) 加入 8 号盘（入栈）

图 A-1 盘子堆栈

层，如图 A-1c 所示。向堆栈中增加一个元素的操作称为压入（push），移走一个元素的操作称为弹出（pop）。要询问堆栈的栈顶（堆栈顶部）元素，不是采用移走栈顶元素的方法，而是窥视（peek）这个元素。

如果在一个程序中需要处理一系列的嵌套子程序调用，那么堆栈是一个非常有用的数据结构。当下一个地址到来之前，如果在进行分支转移到下一个地址之前，将当前的地址压入堆栈顶，那么程序就可以沿着到来时的同样的路径返回。接下来所要做的事情就是，在需要时，依次弹出每个地址（进行出栈操作）。下面举一个日常生活中的例子，假如我们按如下的顺序访问一系列的城市：

1. New York, NY
2. Albany, NY
3. Buffalo, NY
4. Erie, PA
5. Pittsburgh, PA
6. Cleveland, OH
7. St. Louis, MO
8. Chicago, IL

如何从 Chicago 返回 New York 呢？很简单，人们首先会拿出一个地图（找到一条更直接的路线），或者已经“知道”去找到 80 号高速公路然后一直向东行驶。当然，计算机并没有人类的智能。所以计算机最容易做的事情是沿着原来的路线返回。堆栈（类似于表 A-3 所给出的）正是最合适做这项工作的数据结构。计算机需要做的就是，在遍历路线过程中，将当前的位置依次压入堆栈顶。通过从堆栈顶依次弹出原先的那个城市就可以很方便地找到返回路线。

表 A-3 访问城市的一个堆栈

堆栈位置	城 市
7（顶部）	St. Louis, MO
6	Cleveland, OH
5	Pittsburgh, PA
4	Erie, PA
3	Buffalo, NY
2	Albany, NY
1	New York, NY

可以采用各种不同的方式来实现堆栈数据结构。最流行的软件实现方式是通过线性数组和链表来完成的。系统堆栈（堆栈的硬件版本）是采用一个固定的存储器分配空间来实现的，即系统为系统堆栈留出一大块专用的存储器空间。管理系统堆栈需要两个存储器变量。一个存储器变量指示堆栈顶（即进入堆栈的最后一个元素），另一个存储器变量对堆栈中的元素数目进行计数。最大堆栈容量（即堆栈允许的最高存储器地址）会作为一个常数被存储起来。当一个元素压入堆栈后，堆栈指针（堆栈顶的存储器地址）会按照在堆栈中存储数据类型的大小来进行递增操作。

现在考虑这样一个例子，我们要存储字母表中的最后三个字母，并且要以相反的顺序取出这三个字母。用十六进制表示这些字母的 Java（统一编码，Unicode）编码为：

X=0058，Y=0059，Z=005A。

存储器中从 808A 到 80CA 的地址空间保留给堆栈使用。堆栈的最大容量为一个常数 MAXSTACK，设置为 20（十六进制）。因为堆栈最初为空，所以堆栈指针的初值为 0，堆栈计数器的初值也是 0。表 A-4 跟踪记录了存储这三个统一编码字母时，堆栈的状态和堆栈的管理变量。

表 A-4 添加字母 X、Y 和 Z 入堆栈（虚线代表不相关存储器的值）

存储器地址	堆栈内容
8091	---
8090	---
808F	---
808E	---
808D	---
808C	---
808B	00
808A	58

堆栈顶＝808A

a）X（0058）压入堆栈，堆栈指针按照数据元素的大小（2 个字节）增加。

存储器地址	堆栈内容
8091	---
8090	---
808F	---
808E	---
808D	00
808C	59
808B	00
808A	58

堆栈顶＝808C

b）Y（0059）压入堆栈，堆栈指针的值再增加 2 个字节。

存储器地址	堆栈内容
8091	---
8090	---
808F	00
808E	5A
808D	00
808C	59
808B	00
808A	58

堆栈顶＝808E

c）Z（005A）压入堆栈，堆栈指针的值再增加 2 个字节。

要取回这三个字母，需要执行三次出堆栈操作（弹出）。每弹出一次，堆栈指针的值就递减 2。当然，每次增加数据或取出数据，必须要检查堆栈内部的状态。必须确保不会在堆栈满时向堆栈增加元素，而不会在堆栈空时从堆栈移走元素。在计算机系统的硬件和软件中，堆栈都有广泛的应用。

A.3 树

队列、堆栈和数组对于处理一些事物的列表是非常有用的，无论列表中有多少个元素，这些元素在列表中的位置（相互之间的相对位置）是不会发生改变的。显然，在日常生活中使用数据收集的许多性质并不是如此。现在，我们考虑一个管理地址簿的程序。对这类数据进行排序的一种有用的方法是按照人名的姓氏顺序来排序。*二分搜索*（binary search，或称为对分查找）的方法可以迅速查找列表中的任何一个名字，这种方法每次限制只搜索列表的一半。图 A-2 说明了如何使用二分搜索方法，在一些著名数学家名单中查找名为 Kleene 的数学家。首先要决定列表的中间位置（Hilbert），并将这个值与关键字进行比较。如果它们相等，就找到了要找的名字。如果关键字 Kleene 比这个列表的中间位置的值大，那么就在列表底部的一半进行查找，如图 A-2b 所示。通过二分搜索能够有效地减少查找空间。现在，我们在列表底部那一半找到它的新的中间位置（Markov）。因为关键字比这个新的中间值小，所以我们应该在列表底部一半的上半部分在进行查找，如图 A-2c 所示。如果仍然没有找到关

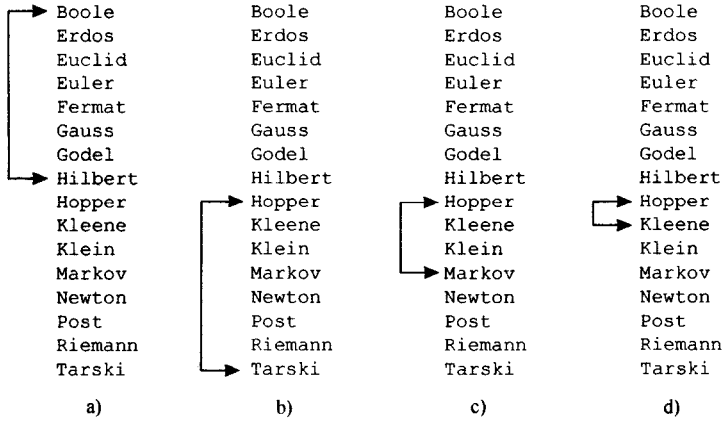


图 A-2 二分搜索 Kleene

键字，那么再把这个上半部分列表进行对半查找。按照这种方法，继续拆分列表直到找到要求的关键字为止，或者证明这个关键字不在列表中。设计这个例子属于最坏的情况。在这个有 16 个元素列表中，通过 4 次操作找到了要求的关键字。如果碰巧要查找的是 Hilbert，那么第一次操作就可以找到。无论这个列表多么大，任何一个名字都能够定位，查找需要的时间与列表中元素总数的以 2 为基的对数成正比。

很显然，二分搜索要求数据按照它的关键字的值来进行排序。当想要往地址簿中增加一个名字时，会发生什么情况呢？我们必须将这个新名字放到一个合适的位置，以便能够可靠地使用二分搜索。如果地址簿是使用线性数组的形式存储的，那么可以非常容易地计算出这个新元素位于哪个位置，假设位置为 k。但是，要在这个数组中插入这个元素时，就必须为这个元素移出空间来。这就意味着首先要将这个数组中从位置 k 到位置 n（假设 n 是地址簿中的最后一个元素）的所有数据元素，移动到 $k+1$ 到 $n+1$ 的位置上。如果这个地址簿的容量比较大，则这种移位过程可能会比我们想像的要慢。进一步地说，如果这个数组恰好只能容纳 n 个元素，那么这个问题就会变得非常麻烦。这时，必须定义一个新的数组，从旧的数组载入新的数组，而花费更多的时间。

采用链表的实现方法也不是太好。因为链表的中间位置很难找到。搜索链表的唯一方法是追随列表中的元素链，直到找到新元素应该放置的位置为止。如果这个链表非常长，线性搜索在操作上几乎是不可行的。搜索速度非常慢，可能大家都难以忍受。

因此，要保持顺序性和可维护性的列表，一个好的数据结构应该是允许我们迅速找到要求的事项，方便地增加和删除一些元素而不会产生过多的开销。事实上，有几种数据结构就很适合于这种情况。其中最简单的就是二叉树（binary tree）。像链表一样，二叉树通过指向存储器空间的指针来控制邻近的数据元素。并且，也与链表一样，二叉树的结构可以生长到无穷大。按照这种方式成长的二叉树，可以很容易地从这棵树上取到任意的一个关键字。二叉树之所以称为二叉（binary），是因为在它们的图形表示法中，每个节点（顶点）最多只有两个分支（孩子，child）节点。超过两个分支节点的树称为 n 叉树（n-ary trees）。一些二叉树的例子如图 A-3 所示。这些图形看起像是一些倒长的树，不必奇怪，它们都是数学意义上的树。二叉树的每个节点都连接到这种图形上，这意味着每个节点都可以从第一个节点到达。而且，这种图形并没有包含环（cycle），也就是说在二叉树上搜索时，不会以循环结束。

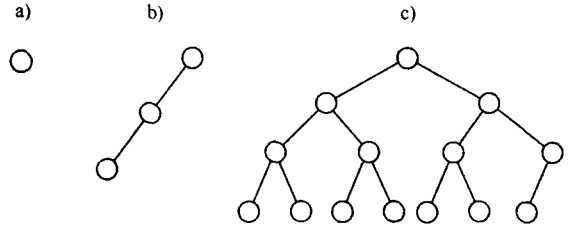


图 A-3 一些二叉树

树的最顶层的节点叫做树根（root）。树根只是树的一部分，树根必须是独立的。其他所有节点都可以通过树根来引用。访问节点的方法是利用存储在每个节点的两个存储器指针的值。每个指针都指示在哪里可以找到该节点的左子节点或右子节点。树的叶子（leaves），就是树结构的最底端节点，它们的孩子节点都是空（0）值。从树根到叶子之间的距离，也就是层（level）的数目，称之为树的高度（height）。那些不是叶子的节点称为树的内部节点（internal node，或称为中间节点）。内部节点至少有一棵子树（subtree），即使这棵子树是一片叶子。

除了指针外，二叉树的节点周围还包含数据（或者是数据的关键字的值），这些数据组成了树的结构。二叉树的组织结构通常是把所有小于某个特定节点的关键字值的关键字值都存储在这个节点的左子树上，而所有大于或等于某个特定节点关键字值的关键字值都存储在这个节点的右子树上。这种思

想的例子如图 A-4 所示。

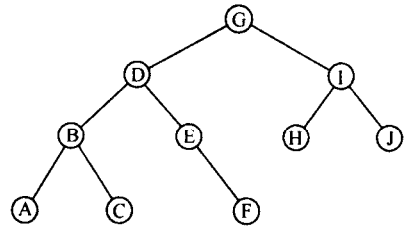


图 A-4 按照不减少的关键字值排序的二叉树

图 A-4 所示的二叉树也是一种平衡的（balanced）二叉树。从形式上来看，如果一棵二叉树是一棵平衡的二叉树，那么每个节点的左子树和右子树的深度的差别最多为 1。这一点的重要意义在于，这颗树所引用的任何数据项都可以被定位，查找需要的时间和树中节点数的 2 的对数成正比。因此，在一棵包含 65535 个数据关键字的树中查找某个特定的元素（或者证明在树中找不到这个元素）最多需要 15 次的存储器操作。与存储在线性数组中的分类链表不同（这里，每次搜索都需要相同的运行时间），在二叉树中保持这组关键字的值会变得非常容易。要插入一个元素，只需重新安排一些存储器指针，而不是重构整个列表。在平衡的二叉树中插入和删除一个节点的运行时间都是与树中节点数的 2 的对数成正比。因此，对于维护一组分类数据元素，这种数据结构比数组或简单的链表更好。

虽然我们的图形对于形成树的逻辑结构的概念非常方便，但是需要记住计算机的存储是线性的，因此这里的图像只是一种抽象的概念。在表 A-5 中，给出了存储图 A-4 的树的 64 字节的存储器映射方式。为了便于阅读，这里采用了表格形式。例如，位于列 5（标号为 5 的列）和行 1（标号为 1 的行）的字节的十六进制地址是 15。行 0 列 0 所指的是地址 0。节点的关键字是十六进制的 ASCII 编码，这些编码的值在存储器映射表的上方表中给出。

表 A-5 图 A-4 中的二叉树的存储器映射

字符（关键字值）	ASCII 编码（十六进制）	字符（关键字值）	ASCII 编码（十六进制）
A	41	F	46
B	42	G	47
C	43	H	48
D	44	I	49
E	45	J	4A

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	--	--	--	--	--	--	--	00	46	00	--	--	00	45	07	--
1	00	43	00	--	00	48	00	--	--	--	--	--	--	--	--	--
2	--	00	4A	00	--	2B	44	0C	--	--	--	31	42	10	--	--
3	--	00	41	00	--	--	25	47	3B	--	--	14	49	21	--	--

这里的存储器映射中，树根位于地址 36 到 38（行 3，列 6－8）。它的关键字的值的地址为 37。树根的左子树（孩子）的地址为 25，右子树的地址为 3B。如果我们看地址 3B，会发现这个关键字 I，它的左子树位于地址 14，右子树位于地址 21。在地址为 21 的位置，会找到叶子节点 J，它的两个孩子指针都是 0。

二叉树在许多应用中都是十分有用的，例如在编译器和汇编程序中（参考第 8 章）。然而，当谈到要从大型数据集中存储和取回关键字值的时候，有几种数据结构要优于二叉树。例如，考虑一个为 800 万人口的纽约市设计在线电话簿的任务。假定电话本里大约有 800 万个电话号码，那么采用二叉树则最少要需要 23 层。而且，至少有一半以上的节点在树的叶子上。这意味着大多数情况下，在找到要求的电话号码之前，我们必须要读取指针多达 22 次。

虽然二叉树的设计并不是完全不适合这种应用，但是我们可以对它进行改进。一个比较好的方式是使用一个 *n* 叉树结构，叫做 trie（发音为“try”）。*n* 叉树结构并不是在每个节点上存储整个的关键

字，tries 使用关键字的分段技术。在沿着这个 trie 的搜索过程中，关键字的值被组合安装起来。内部节点包含足够的指针数来指引搜索查找所要求的关键字，或者是到 trie 的下一层去。*n* 叉树结构特别适合于具有可变长度关键字的数据，例如，电话簿的例子。在 *n* 叉树的结构中，较短的关键字位于靠近顶层，而较长的关键字位于数据结构的底部。

在图 A-5 中，描述了一个包含著名数学家名字的 *n* 叉树结构。这个图中的每个内部节点都包含 26 个字母。这种数据的性质表明，还可以有一些比图中给出的结构更有效率的 *n* 叉树结构。我们发现，要找到名字以 ZQX 开头的著名数学家是非常困难的。事实上，设计内部节点的结构是 *n* 叉树结构中最难的一部分。这里使用关键字的值作为一个索引，而不是采用一个字符。例如，不采用包含字母表中的所有字母作为一个单一单元，而是对字母进行分组。在改变图 A-5 中的 *n* 叉树结构的根节点的关键字的多样性后，这个树会变矮（压扁）一层。这种修改后的结构如图 A-6 所示。如果仔细进行平整，那么这颗树变平的结果会使得搜索的速度更快。在图 A-6 中，在树根中只选择两个组合的关键字（ER 和 EU）来减少一层结构。如果将每个关键字都加倍的话，则树根将会包含 676 个关键字（从 AA 到 ZZ）。这相对于所存储的数据来说，会使得这种数据结构变得不必要的庞大和难以处理。

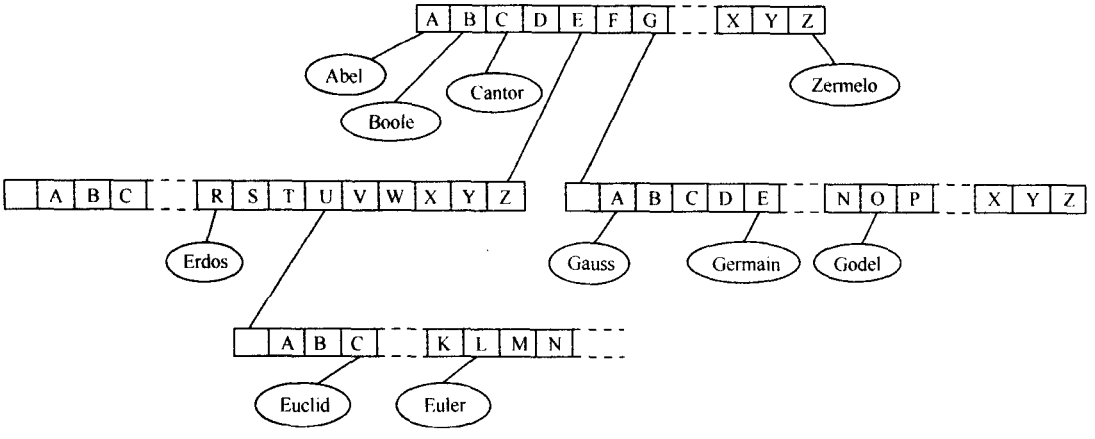


图 A-5 一个包含著名数学家名字的 *n* 叉树结构

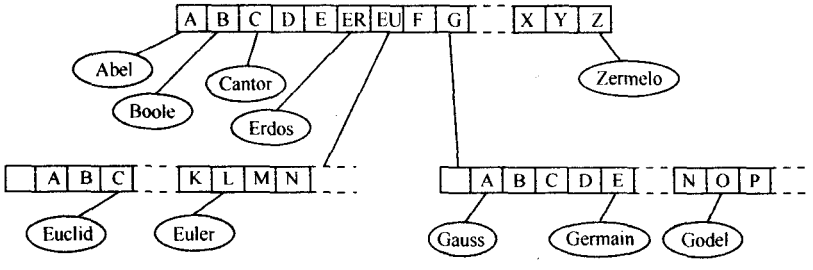


图 A-6 一个较平的著名数学家名字的 *n* 叉树结构树

实际上，对于大量数据的存储和检索，结构设计时更多考虑的是数据存储的介质，而不是数据本身的性质。通常，人们会精心设计树的索引节点，以便通过对存储索引的磁盘驱动器的一次读操作，就可以访问位于树的某一层的内部节点的某些整数。其中一种数据结构就是 B+树，它通常在大型数据库系统中使用。

B+树（B+tree）是一种由指示索引结构或者实际数据记录的指针构成的分层体系结构。当从数据库中增加或删除记录时，需要更新 B+树的叶子节点。当不可能更新现有的叶子节点时，就要增加内部节点。B+树的内部节点的集合被称为*索引部分*（index part），而叶子节点被称为顺

序部分（sequence part），因为叶子节点总是按照大小次序排列的。一个 B+树的局部示意图如A-7所示。

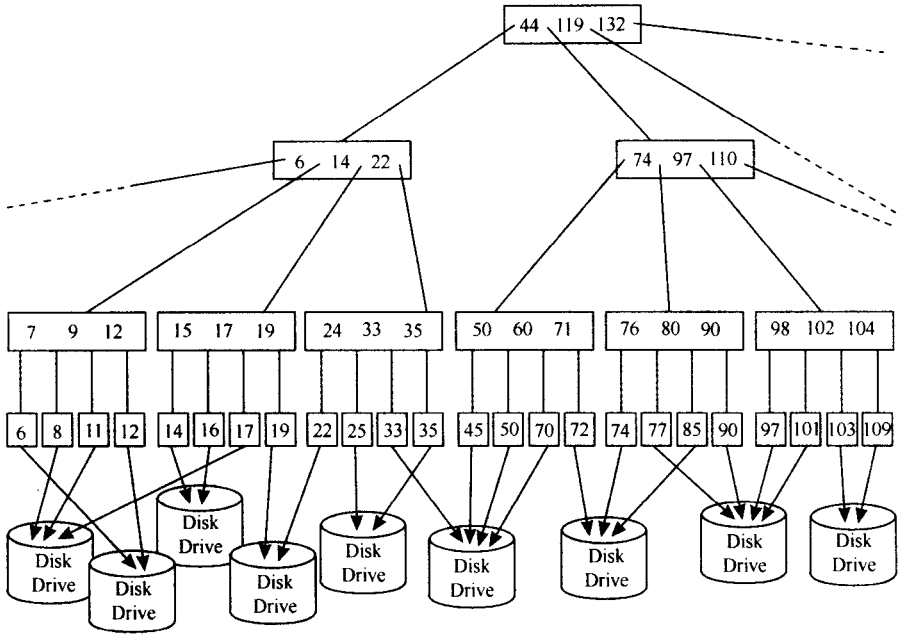


图 A-7 一个 B+树的局部

图中显示的数字是记录关键字的值。连同 B+树的叶子节点中的每个关键字的值，数据库管理系统（参考第 8 章）会为这些物理记录的存储位置都提供一个指针。操作系统会使用这个指针值来从磁盘中读取记录。实际上物理记录可以位于存储系统的任何位置，但是数据结构的顺序部分总是按次序存放的。对 B+树进行遍历可以确保我们根据记录的关键字的值迅速找到任何的记录。

要使用图 A-7 中的 B+树来定位一个关键字的值，只需要将要查找的关键字的值与存储在内部节点的值进行比较。如果所求的关键字的值比内部节点的一个关键字的值小，则这颗树就从左边遍历。相应地，如果所求的这个关键字的值大于或等于内部节点的关键字的值，则往右边遍历这颗树。当一个内部节点已经达到其最大容量，还需要向数据库中增加记录时，就需要往这种层次结构中添加附加的层次。然而，记录删除时，并不会立即造成树被平整（压扁），而仅仅是指针发生一个位移。B+树的层次结构被平整化的过程叫做*数据库重组*（database reorganization，或 reorg）。在大型数据库中，重组可能是非常耗时的，因此通常只是在绝对必须的情况下才执行这种操作。

最佳的数据库检索方法必须考虑数据库运行系统的体系结构。特别地，最佳的系统性能，一定要保持磁盘读操作的次数为最小值（参考第 10 章）。如果一个数据文件索引的实质部分没有被存放在存储器的高速缓存中，那么访问记录至少需要两次读操作：一次是读取索引，另一次是取回记录。对于 B+树索引中频繁使用的文件，树的前几层都从高速缓存存储器中读取，而不是从磁盘中读取。这样，只有当检索索引较低的树结构层次和数据记录本身时，才需要读取磁盘。

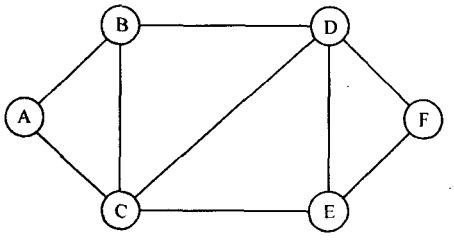
A.4 网络图

根据定义，树结构不包含环。对于数据存储和检索来说，树结构是非常有用的。当然，按照计算的复杂性来说，这也是一项比较简单的任务。很明显，处理一些困难的问题要求采用更加复杂的结构。例如，考虑在 A.2 节中提出的行程安排问题，需要找到一条从 Chicago 到 New York 的返回路径。在那里，我们并没有讨论寻找最短返回路径的问题，而只是采用了最简单的方法，简单地按原

路返回。要寻找最短路径，或者是最优路径，就要求使用一种不同类型的数据结构，允许有环的数据结构。

n 叉树可以变成一个更普遍的网络图，只要允许叶子节点相互指向连接。但是现在要考虑是这样的一个事实：任何一个节点都可能指向图中其他 $n-1$ 个节点。如果简单地讨论从二叉树数据结构来构建一个网络数据结构，那么每个节点将需要 $n-1$ 个指针。当然，我们可以做得更好一些。

如果所讨论的网络是静态的，也就是说，在算法的执行过程中网络既不会增加也不损失节点，则可以使用邻接矩阵（adjacency matrix）来表示。邻接矩阵就是一个二维数组，每个节点都具有相应的行和列。考虑一个如图 A-8a 所示的网络图。它有 6 个相互连接的节点。图中节点之间的连接（称为边，edge）在邻接矩阵中用 1 来表示。在邻接矩阵中，将 1 放置在一个节点的列以及与它相连的节点的行的交叉位置上。完成的邻接矩阵如图 A-8b 所示。



a) 一个网络图

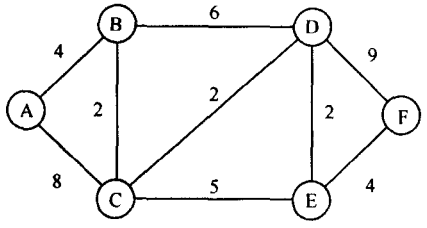
	A	B	C	D	E	F
A	1	1	1	0	0	0
B	1	1	1	1	0	0
C	1	1	1	1	1	0
D	0	1	1	1	1	1
E	0	0	1	1	1	1
F	0	0	0	1	1	1

b) 网络图的邻接矩阵

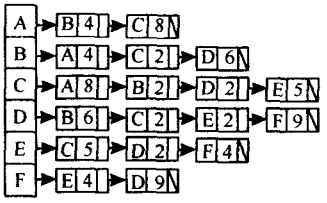
图 A-8 网络图

现在，让我们回到在两个城市之间找到一个优化路径的例子。首先把城市连接地图表示成一个带有权值（weighted）边的图。这些边的权值代表了城市之间的相应距离，或者是从城市之间往返的费用。在这里，不是把一些 1 放到邻接矩阵中，而是在两个城市之间存在的通行路径的位置标注旅行所需的费用。

我们也可以把一个连接图表示成一个链接的邻接列表（adjacency list）。要实现一个邻接列表结构通常使用一个用线性数组来保存图中的各个节点，线性数组会指示节点之间的连接方式。这个排列的好处是可以非常容易地找到图中的任何一个节点，而且在一个节点到另一个节点之间的旅行费用也可以与数组中的列表元素一起标示出来。一个带有权值的网络图和相应的邻接列表的数据结构如图 A-9 所示。



a) 带有权植的网络图



b) 网络图的邻接列表

图 A-9 网络图

正如上面所述，这些普遍的网络图广泛应用于解决通信路由的问题。这些算法中，其中最重要的算法是 Dijkstra 算法。Dijkstra 算法的思想是通过由所有节点之间的所有最短的连接组成的网络图中计算出最低费用的路径。这种算法会首先检查图与起始节点相邻的所有路径，并且利用从起始节点到达各个节点的费用更新每个节点的值。接着会检查通往相连节点的每个路径，并且使用到达该节点的费用更新这些相连节点的值。如果这个节点中已经包含了一个费用值，那么只有当从起始节点到达该节点的旅行费用小于已经记录在这个节点的值的情况下，才会把这个节点选为下一个目的节点。这个过

程如图 A-10 所示。

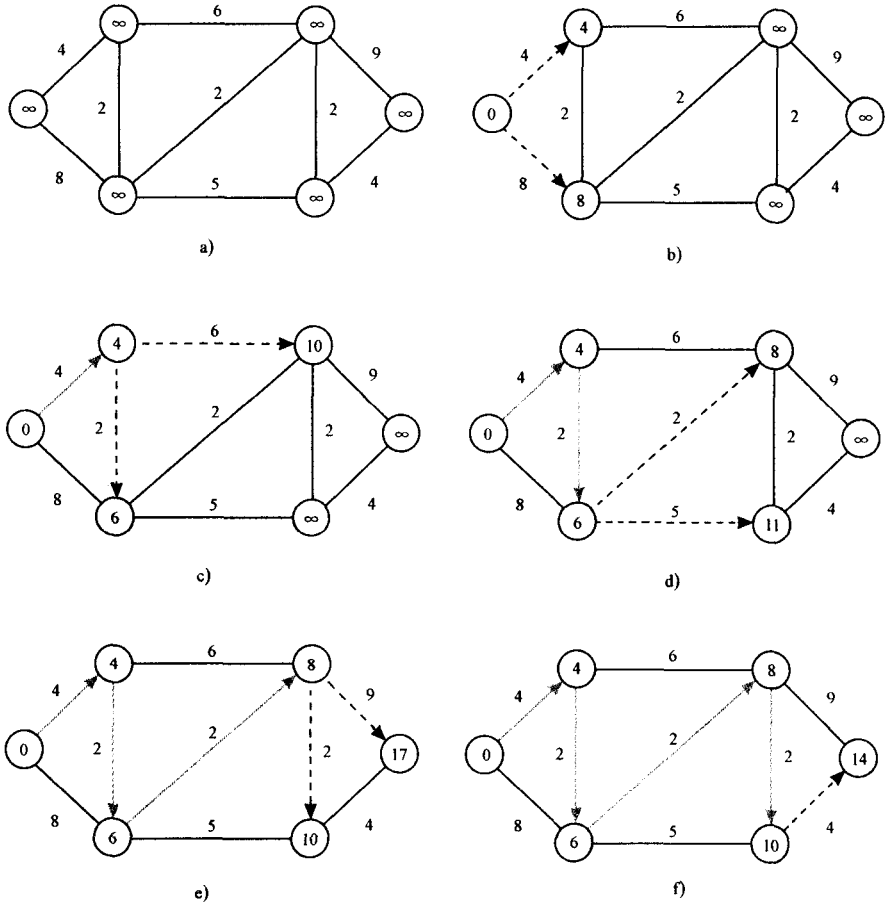


图 A-10 Dijkstra 算法

图 A-10a，先把到达所有节点的值设置为无穷大。然后检查从第 1 个节点到相邻节点之间的路径，并且用到达节点的费用更新每个节点的值，参见图 A-10b。接着再检查从最小值节点到其相邻节点之间的路径，如果从第一个节点到达相邻节点的费用值比先前存储在每个节点的值小，那么就用这个较小的费用更新这些相邻节点的值。这就是图 A-10c 的左侧边节点上所发生的情况。重复这个过程直到找到最短的路径，如图 A-10f 所示。

Dijkstra 算法中比较棘手的部分要涉及到大量的数据结构。算法不仅需要提供这种网络图本身，而且还必须以某种方式来记录每个节点的路径，以便在需要时可以读取。这里留给读者作为一个练习，表示所要求的数据结构以及在数据结构上运算 Dijkstra 算法构建伪代码。

小结

本附录描述了几种重要的计算机系统中最常使用的数据结构。堆栈和队列是系统最低层结构中最重要的数据结构，这些数据结构的简单性正好与发生底层的简单操作相匹配。在系统软件层次上，编译器和数据库系统主要依赖于树型数据结构来实现更快的信息存储和检索。最复杂的数据结构都是应用于高级语言的层次。这些复杂的数据结构可以由多个附属的（子）数据结构组成。正如本附录中所介绍的，需要同时使用数组和链表才能完整描述网状图。

深入阅读

很好地学习和理解这个简要的附录中所讨论的内容，对于继续学习计算机系统和编程方法是十分必要的。如果读者是第一次学习这种数据结构，那么建议大家阅读 Rawlins（1992）的有关算法的著作。本书组织合理、生动有趣和具有丰富的内容。对于有兴趣进行深入学习和提高的读者，Knuth（1998），以及 Cormen、Leiserson、Rivest 和 Stein（2001）的著作都提供了更加详细的内容。另外，Weiss（1995）、Horowitz 和 Sahni（1983）的书籍则比较简洁和易于理解，内容也都包含了本附录中所描述的各种重要的数据结构。

参考文献

Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., & Stein, Clifford. *Introduction to Algorithms, 2nd ed.* Cambridge, MA: MIT Press, 2001.

Horowitz, Ellis, & Sahni, Sartaj. *Fundamentals of Data Structures.* Rockville, MD: Computer Science Press, 1983.

Knuth, Donald E. *The Art of Computer Programming, 3rd ed.* Volumes 1, 2, and 3. Reading, MA: Addison-Wesley, 1998.

Rawlins, Gregory J. E. *Compared to What? An Introduction to the Analysis of Algorithms.* New York: W. H. Freeman and Company, 1992.

Weiss, Mark Allen. *Data Structures and Algorithm Analysis, 2nd ed.* Redwood City, CA: Benjamin/Cummings Publishing Company, 1995.

练习题

1. 对于下面的各种数据结构，至少举出一个最适合的应用例子：
   a) 数组
   b) 队列
   c) 链表
   d) 堆栈
   e) 树
2. 正如本附录中所介绍的，优先级队列也是一个队列。如果某些元素满足特定的条件，那么优先级队列就允许这些元素跳到队列的头部。请设计一个数据结构和一种合适的算法来实现优先级队列。
3. 假如你不想使用树型数据结构来保存一组分类数据元素，那么可以选择链表来实现。很明显，链表方法的效率非常低。这个列表是按关键字的值的递增顺序进行排列的。也就是说，最小关键字的值排在列表的开始位置。为了定位查找某个数据元素，必须要按顺序搜索这个列表直到找到一个关键字的值比要求的关键字的值大为止。如果搜索的目的是向这个列表中插入另一个元素，那么如何完成这个插入过程？换句话说，编写一种伪代码算法列出每一步过程。还可以通过稍微改变这个列表的数据结构来使这种算法更高效。
4. 下面是一个二叉树的存储器映射表。请画出这棵树。

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	--	--	--	--	--	00	45	00	--	--	--	--	--	--	--	--
1	00	41	00	--	--	--	--	--	--	05	46	37	--	--	--	00
2	43	00	--	--	--	--	--	--	--	--	--	--	10	42	1F	--
3	--	--	2C	44	19	--	--	00	47	00	--	--	--	--	--	--

5. 下面是一个二叉树的存储器映射表。请画出这棵树。

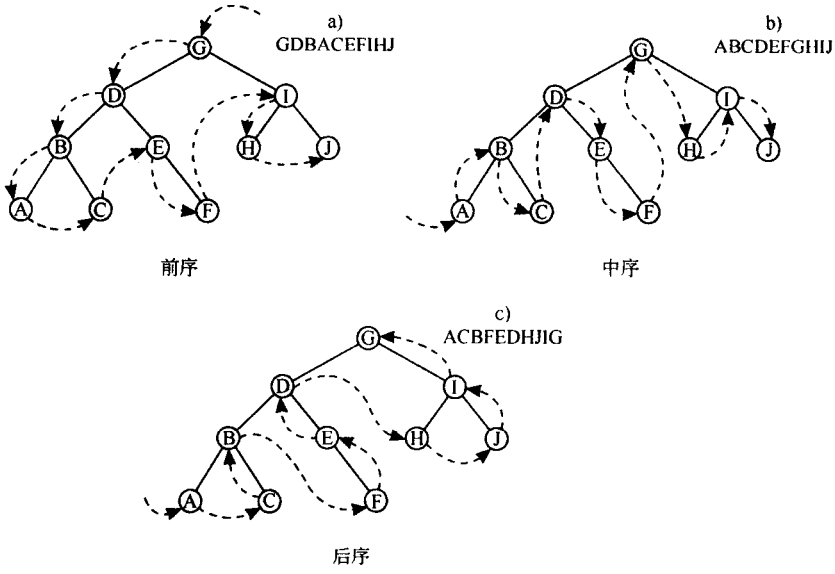
	0	1	2	3	4	5		7	8	9	A	B	C	D	E	F
0	--	--	--	--	--	--	--	--	24	46	12	--	--	--	00	42
1	30	--	00	45	00	--	--	--	--	--	--	--	--	--	--	--
2	--	--	--	--	0E	47	00	--	--	--	00	43	00	--	--	--
3	00	44	00	--	--	--	--	--	--	--	--	08	41	2A	--	--

6. 下面是一个二叉树的存储器映射表。这颗树的叶子节点包含的关键字为：H（48），I（49），J（4A），K（4B），L（4C），M（4D），N（4E）和O（4F）。请画出这棵树。

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	2E	46	39	00	4B	00	48	35	44	04	14	11	41	08	35
1	FF	19	42	22	3F	FF	01	43	3C	00	48	00	48	20	41	00
2	4A	15	00	49	00	42	00	4E	00	47	0C	45	16	08	00	4C
3	00	00	4F	00	43	00	4A	00	45	00	4D	00	26	47	31	41

7. 发明一个公式计算可以放到一个 n 层的二叉树中的最大的节点数目。

8. 图的遍历就是询问（访问）图中每一个节点的过程。当以某种顺序（也可能无序）向树中增加节点和以某种其他的顺序检索节点时，遍历是非常有用的。下图中显示了三种常用的遍历方式，分别是*前序*（preorder）遍历、*中序*（inorder）遍历和*后序*（postorder）遍历。其中，a 图表示前序遍历，b 图表示中序遍历，c 图表示后序遍历。



前序　　中序

后序

a）重新排列上面的树，使得在前序遍历过程中可以按照字母表的顺序打印出节点关键字的值。只改变节点中的关键字的值，对于中序遍历也实现同样的要求。

b）对在 a 部分中重画的两棵树来实现其他两种遍历。

9. 大部分有关算法和数据结构的书籍都提出遍历算法就是递归过程。（递归过程是调用自己的子程序或函数。）然而，计算机是使用迭代法来完成这种递归过程的！下面的这个算法是使用一个堆栈来

完成树的一种迭代的前序遍历（参考上一题）。当每个节点被遍历时，会像上面的图一样打印出它的关键字的值。

```
ALGORITHM Preorder
   TreeNode : node
   Boolean : done
   Stack: stack
   Node ← root
   Done ← FALSE
   WHILE NOT done
      WHILE node NOT NULL
         PRINT node
         PUSH node onto stack
         node ← left child node pointer of node
      ENDWHILE
      IF stack is empty
         done ← TRUE
      ELSE
         node ← POP node from stack
         node ← right child node pointer of node
      ENDIF
   ENDWHILE
END Preorder
```

a) 修改这个算法，让程序可以实现中序遍历。

b) 修改这个算法，让程序可以实现后序遍历（提示：当离开一个节点去沿着它的左子树而行时，更新这个节点的值表示这个节点已经被访问过）。

10. 考虑图 A-6 中的 n 叉树的根节点，如果发现一个名叫 Ethel 的著名数学家，将会出现什么样的复杂情况？应该如何防止这类问题发生？

11. 使用 Dijkstra 算法，寻找一条从 New York 到 Chicago 的最短路径，使用下面的邻接矩阵中所给出的里程。值为“无穷大（∞）”表示在给定的两个城市之间没有直接的连接。

	奥尔巴尼	布法罗	芝加哥	克里夫兰	伊利	纽约	匹兹堡	圣路易斯
奥尔巴尼	0	290	∞	∞	∞	155	450	∞
布法罗	290	0	∞	∞	100	400	∞	∞
芝加哥	∞	∞	0	350	∞	∞	∞	300
克里夫兰	∞	∞	350	0	100	∞	135	560
伊利	∞	100	∞	100	0	∞	130	∞
纽约	155	400	∞	∞	∞	0	∞	∞
匹兹堡	450	∞	∞	135	130	∞	0	∞
圣路易斯	∞	∞	300	560	∞	∞	∞	0

12. 构思一种，在存储邻接矩阵时可以占用更小的主存储器空间的方案。

13. 设计一种具有适合的数据结构的算法，来实现 Dijkstra 算法。

14. 本附录中哪种数据结构最适合用来创建字典（这个字典将作为一个字处理器中的拼写检查器）？

术 语 表

1s Complement Notation：参见 One's Complement Notation（反码表示法）。

2s Complement Notation：参见 Two's Complement Notation（补码表示法）。

Access Time（存取时间）：（1）硬盘的旋转延迟和寻道时间的总和。（2）从磁盘或存储器上找到和返回要求信息所需要的时间。

Accumulator Architecture（累加器体系结构）：该体系结构假定了一个操作数位于累加器中，指令中无需显式地引用累加器。

ACID Properties（ACID 特性）：数据库系统或事务处理系统的 4 个特点。（1）原子性：所有相关的更新或者在事务的范围内发生，或者根本就不发生更新；（2）一致性：所有的更新都要满足对全部数据元素设置的限制；（3）隔离性：事务不能干涉其他事务的活动或更新；（4）持久性：成功的事务要尽可能快地写入到“持久性”的介质上（如磁盘）。

Actuator Arm（磁头臂）：保持读/写头的磁盘驱动器的机械设备。

Addend（加数）：在算术加法运算中，加数与被加数的值相加构成算术求和。

Address Binding（地址绑定）：将符号地址映射到实际的物理存储器的过程。

Address Bus（地址总线）：微处理器读或写操作时传送地址的总线部分。

Address Spoofing（地址欺骗）：一种黑客攻击技术，即一台主机使用非法的假 IP 地址与另一台主机进行通信。IP 欺骗常常用来破坏用来防止外部用户入侵企业内部互连网的过滤路由器和防火墙。

Addressing Mode（寻址模式）：通过对操作数的解释方式，说明指令的操作数的存放位置。

Adjacency List（邻接列表）：一种建模有向图形（或网络）的数据结构，使用节点元素之间的指针来显示图中两个节点之间路径的存在。

Adjacency Matrix（邻接矩阵）：一种建模有向图形（或网络）的二维数组。如果这个图包含 n 个节点，那么邻接矩阵将有 n 行和 n 列。如果图中节点 x 和节点 y 之间有路径存在，那么在邻接矩阵 x 列 y 行处将包含一个非 0 值。反之，相应的位置为 0。

Aggregatable Global Unicast Address Format（可聚集的全局统一地址格式）：在 IPv6 下可能构成 2^{128} 个主机地址的一种方案。

AGP，Accelerated Graphics Port（加速图形端口）：Intel 公司为 3D 图形专门设计的一种图形端口。

Algebraic Field（代数域）：具有单位元素的一组数，在加法和乘法两种运算中是封闭的。域还满足代数的结合率、交换率和分配率。实数系统是一个域。

Amdahl's Law（Amdahl 定律）：这个定律阐述了改进方法特性的使用量限制了系统性能的提高。更正式地说，计算机系统的整体加速率取决于特定部件的加速率和该部件在系统中的使用率。用公式表示为：S=1÷［（1－f）＋f÷k］，S 表示加速率，f 表示较快部件完成的工作部分，k 表示新设备的加速率。

American National Standards Institute（美国国家标准化组织）：一个代表美国利益的制定计算机行业标准的国际性组织。

American Standard Code for Information Interchange（美国信息交换标准代码）：参见 ASCII。

Analytical Engine（分析机）：1833 年由 Charles Babbage 设计的一种通用计算机器。

Anycasting（泛播）：一种网络消息的发送方式，它允许任何一组逻辑节点接收消息，在消息中并不指定特定的接收器。

Aperture Grill，AG（栅格式）：荫罩（使用一种精细的金属网孔）的一种代替品，在 CRT 显像管

中，利用一些细长金属带迫使电子束照射荧光屏的正确部位。

Arithmetic Coding（算术编码）：一种数据压缩的方法，利用一组被压缩消息的符号的概率在 0 和 1 间隔内来分割实数轴。出现概率较高的符号得到较大的间隔段。

Arithmetic Logic Unit，ALU（算术逻辑单元）：组合电路负责完成 CPU 中的算术和逻辑功能。

Arithmetic Mean（算术平均值）：一种集中趋势的测量方法，利用一组数据值的和再除以数据值的个数就得到了算术平均值。在通常使用中，对一组数值取“平均”就是这些值的算术平均值。

ARPAnet：参见 DARPAnet。

ASCII，American Standard Code for Information Interchange（美国信息交换标准代码）：

使用 7 位字符编码来表示数字、字符、特殊格式字符和控制符。

Assember Directive（汇编指令）：专门用于汇编程序的一种特殊指令，这种指令不能被译成机器代码，但是可以通知汇编程序去完成一个特定的功能，例如在一个程序清单中产生一个分页符。

Assembly Language（汇编语言）：使用助记符的低级语言，对于特定体系结构汇编语言和机器语言存在着一一对应的关系。

Associative Memory（关联存储器）：按照内容而不是按照地址来寻址的存储器，主要是为了进行并行搜索而专门设计的存储器。

Asynchronous Circuits（异步电路）：只要有任何输入值的变化就会激活的时序电路。

Asynchronous Transfer Mode（异步传输模式）：参见 ATM。

AT Attachment（AT 附属设备）：参见 EIDE。

ATAPI，AT Attachment Packet Interface（AT 附属包接口）：AT 附属包接口的缩写，支持 CR-ROM 驱动器的 EIDE 扩展。

ATM，Asynchronous Transfer Mode（异步传输模式）：一种数字网络技术，采用固定长度的单元或信息包在同一网络上传输数据、音频和视频。

Attenuation（衰减）：电信号随时间和距离发生的损耗，可能会在接收方导致错误的数据。

Augend（被加数）：参见 addend。

B+ Tree（B+树）：一种非循环的数据结构，由索引结构的指针或数据记录组成。B+树的内部节点的集合称为树的索引部分（index part），而叶子节点称为顺序部分（sequence part），因为它们总是按照顺序存储的。

Backbone（主干网络（或高速链路））：一种传输数字网络通信的大容量通信（主干线）线路。

Backward Compatible（向下兼容）：一个程序向下兼容是指这个程序能够运行同一软件的老版本所创建的文件和数据。一台计算机向下兼容是指这个计算机能够运行同一种体系结构的老版本计算机上所开发的软件。

Bandwidth（带宽）：模拟通信介质能够传输的频率范围，用单位赫兹（Hz）表示。在数字通信中，带宽是一个介质信息传输能力的通称，表示为 b/s（位/秒）。

Base Address（基地址）：数据结构的第一个元素的地址。这种结构中所有其他元素的地址都是通过距离基地址的偏移量来确定。

Base/Offset Addressing（基地址/偏移量寻址）：一种寻址模式，偏移量先被加到某个指定的基址寄存器中，然后再加上某个指定的操作数来生成数据的有效地址。

Based Addressing（基地址寻址）：一种寻址模式，使用一个基址寄存器（或显式地指定或者隐式地指定）来存放一个偏移量（或者位移），将偏移量加上操作数就产生数据的有效地址。

Basic Input/Output System（基本输入/输出系统）：参见 BIOS。

Batch Processing（批处理）：一种计算模式，在正常的情况下，除了启动一个批处理任务之外，整个计算过程中不需要人为干预。在批处理模式下，多个相似的任务可以组合在一起按顺序执行。

Baud（波特）：在某个传输介质或传输方式中测量信号转变数目的一种测量单位。

BCD，Binary Coded Decimal（二进制编码的十进制数）：一种使用 4 位二进制数来表示从 0 到 9 的十进制数的编码系统。BCD 编码是 EBCDIC 编码的基础。

Benchmark Suite（基准套件）：一组用来度量计算机系统性能的内核程序的集合。通过使用不同的内核程序，可以准确地评估计算机系统的处理能力。

Benchmarketing（营销基准）：一种计算机厂商广泛采用的营销方法，通常是广告比竞争对手的系统占优势的系统基准测试结果。

Benchmarking（基准）：一门客观评价计算机硬件系统或软件系统的性能的科学。基准有助于确定升级一台计算机或提高计算机的某个部件的性能。

BER：参见 Bit Error Rate。

Biased Exponent（偏置指数）：一种对浮点数的指数部分进行的调整，可以消除指数中对符号位的需求。偏置值表示 0，是指数可能取值范围中靠近中间的一个数值。大于偏置值的指数是正指数。

Big Endian（大端位序）：一种存储器的位序安排，采用大端位序是将多字节字的最高字节存放在存储器中的最低地址的位置。

Binary Coded Decimal（二进制编码的十进制）：参见 BCD。

Binary Search（二分搜索）：一种在分类数值的列表内定位查找关键字的方法，逐次将列表对半拆分进行查找。

Binary Tree（二叉树）：一种由树根、内部节点和叶子组成的非循环的树形数据结构。在二叉树结构中，根节点和每个内部节点最多可以有两个指针引向其他的节点（即最多有两个分支）。而叶子节点没有分支节点。

Binding Time（绑定时间）：在符号地址绑定发生时所引用的操作。装载时间绑定将地址作为二进制模块一起装入到存储器中。而运行绑定则要延时到这个进程实际运行时才进行地绑定。

BIOS，Basic Input/Output System（基本输入/输出系统）：可编程的集成电路，其中包含某些特殊的系统部件的信息和程序。微型计算机操作系统是通过系统的 BIOS 来实现 I/O 操作和其他专用设备的行为。

Bit（位）：二进制数的缩写，二进制数的值只能取 0 或 1。

Bit Cell（位单元）：指在字节的存储或传输过程中，一个位所占据的线性空间大小或时间的数量。

Bit Error Rate，BER（误码率）：收到的出错的位数和收到的总位数的比率。

Black Box（黑盒子）：（1）实现某个功能的一个模块，模块怎样实现功能的内部细节对于模块外部任何一个人或任何一个操作都是不透明的。（2）在数据通信中，为适应系统协议而接入的某个功能设备，但在最初，并不这个设备做具体设计。

Block Field（块域）：地址中用来指定对应的缓冲区空间块的部分。

Blocking Interconnection Network（阻塞型互连网络）：一种网络类型，阻塞型互连网不允许在有其他同时发生连接存在的情况下，建立新的连接。

Boolean Algebra（布尔代数）：一种代数学，操作的对象只包含两个值，通常是真或假两个值。布尔代数也称为符号逻辑。

Boolean Expressions（布尔表达式）：一个由布尔变量和布尔操作符组成的表达式。

Boolean Function（布尔函数）：一个函数有一个或多个布尔输入值产生一个布尔结果。

Boolean Identities（布尔恒等式）：一些有关布尔表达式和布尔变量的定律。

Boolean Product（布尔积）：一个与（AND）操作的结果。

Boolean Sum（布尔和）：一个或（OR）操作的结果。

Boot，Bootstrapping（引导程序（导入））：利用一个小程序来启动计算机全部操作的过程。

Bootstrap Loader（引导程序装载器）：执行引导程序的计算机固件。

Bootstrapping（引导程序）：参见 Boot。

Branch Prediction（分支预测）：在指令执行之前先猜测指令流中的下一条指令的过程，以避免由于分支转移导致流水线的阻塞现象。如果预测正确，则流水线不会产生延迟。如果预测错误，就要冲洗流水线，将丢弃由该错误计算产生的所有计算结果。

Bridge（网桥）：一种 2 层的网络元件，连接两个类型相似的网络，使它们看起来像一个网络。网桥是一种“存储和转发”的设备，先接收和存储一个整个的传输帧，然后转发出去。

British Standards Institution，BSI（英国标准协会）：一个代表英国利益的制定计算机行业标准的一个国际性组织。

Broadband Cable（宽带电缆）：一种容量至少为 2Mb/s 的导向式网络媒体。宽带通信采用多路技术的形式提供多信道的数据。

Burst Error（突发错误（区块错误））：一种多个相邻的位被损坏的错误模式。

Bursty Data（突发数据）：一种 I/O 操作，主要以大块的、成簇的方式发送数据，而不是以恒定的数据流发送数据。

Bus（总线）：计算机各组件之间传送数据的一组共享线路。同步总线采用时钟同步，只有当定时脉冲到达时，才会发生事件。异步总线使用控制线来协调各种操作，并要求有复杂的握手协议来执行强制定时。请参见 Address Bus、Data Bus 和 Control Bus。

Bus Arbitration（总线仲裁）：决定由哪一个设备来控制总线的过程。

Bus Cycle（总线周期）：两个相邻总线时钟脉冲之间的时间间隔。

Bus Protocol（总线协议）：使用总线的一组规则。

Bus-Based Network（基于总线的网络）：一种允许处理器和存储器通过一个共享总线进行通信的网络。

Byte（字节）：一组连续的 8 位二进制数。

Byte-addressable（按字节编址）：指每个单独的字节都有一个唯一的地址，或者说最小可编址的位串为一个字节。

Cache Coherence Problem（高速缓存的一致性问题）：当存储在高速缓存中的值与存储在存储器的值不同时所产生的问题。

Cache Mapping（高速缓存映射）：覆盖寄存器地址到高速缓存位置的过程。

Cache（高速缓存，简称缓存）：一种专用的高速存储器，用来存放频繁的访问数据或者最近访问的数据。它有两种类型：存储器高速缓存和磁盘高速缓存。存储器高速缓存比主存储器容量小但速度快。存储器高速缓存有两种类型：一级高速缓存（L1）和二级高速缓存（L2）。L1 是一种又小又快的存储器高速缓存，构建在微处理器的芯片内部，有助于加速访问频繁使用的数据。L2 是一组高速的内置式的存储器芯片，位于微处理器和主存储器之间。磁盘高速缓存是一种用来从磁盘上存读的数据的专用缓冲器。

Campus Network（校园网）：在小范围内跨越多个建筑物的私有数据通信网络。校园网通常是局域网（LAN）的延伸，并使用 LAN 的协议。

Canonical Form（标准形式（范式））：相对于布尔表达式来说，指两个标准的形式：积之和形式或和之积形式。

CD Recording Mode（CD 记录模式）：指定在 CD-ROM 上存储数据所使用的格式。模式 0 和 2 用于音乐记录，但没有纠错能力。模式 1 用于数据记录，具有两级查错和纠错能力。一个 CD 利用模式 1 进行记录时的总容量为 650MB。而采用模式 0 和模式 2 进行记录的总容量为 724MB，但是模式 0 和模式 2 都不能可靠地用来记录数据。

CD-ROM（只读光盘）：一类存储数据容量超过 1/2GB 的光盘。其他类型的光盘包括：CD-R（可记录光盘）、CD-RW（可重写光盘）和 WORM（只能写一次可以多次读取的光盘）。

CEN，Comite Europeen de Normalisation（欧洲标准化委员会）：计算机行业标准化的欧洲委员会。

Central Processing Unit，CPU（中央处理器）：负责取指令、译码指令、执行指令和对正确的数据完成指定的操作序列的计算机部件。CPU 是由 ALU、寄存器和控制单元组成的。

Channel I/O（通道控制的 I/O）：使用智能型的 DMA 接口来选择 I/O 设备，也称为 I/O 通道。利用一些小 CPU 来控制 I/O 通道称为 I/O 微处理器（IOP），I/O 微处理器是专门为 I/O 通道优化设计的。

Checkpoint（检查点）：在数据库更新或网络文件传输过程中，检查点每次发布一个数据块，没有出错的数据块会被提交到持久性的存储介质。如果处理过程中发生了一个错误，那么直到最后一个检查点的数据都被认为是有效的。

Checksum（校验和）：对一个或多个数据字节进行某种算术运算后得到的一组二进制位。通常，校验和位会被添加到一个信息字节块的后面，以保持在数据存储和传输过程中的信息完整性。比较流行的校验和是循环冗余校验（CRC）方法。

Chip（芯片）：由实现各种门电路所需要的电子元件（晶体管、电阻和电容）组成的微型硅半导体晶体。

CISC，Complex Instruction Set Computer（复杂指令集计算机）：一种计算机设计思想，CISC 计算机包含大量的、长度可变的、复杂结构的指令。

Client-Server System（客户服务器系统）：参见 N-Tiered Architecture。

Clock Cycle Time（时钟周期）：时钟频率的倒数，也称为时钟周期（clock period）。

Clock Skew（时钟漂移）：在一个系统或网络中原来协调好的时钟逐渐失去同步的情况。

Clock Speed（时钟速度）：处理器的速度，通常用兆赫兹（每秒百万个脉冲，MHz）或千兆赫兹（每秒十亿个脉冲，GHz）来测量。有时也称为时钟频率（clock frequency）或时钟速率（clock rate）。

C-LOOK：参见 LOOK。

Cluster of Workstation，COW（工作站集群）：类似于 NOW 分布式工作站的一个集合，但它需对单一的实体负责管理。

Coaxial Cable（同轴电缆）：一种连接导线，由包围铜质或钢质芯线的绝缘层，再加上网状编织的接地屏蔽层和保护外套所组成。

Code Word（编码字）：一种包含 m 位数据，r 位校验位的 n 位二进制单元，通常用于错误检测和错误校正。

COLD，Computer Output Laser Disk（计算机输出激光盘）：COLD 是计算机的一种输出方法，用来代替纸张和微型胶片。COLD 能够提供数据的长期存档储存。

Combinational Circuit（组合电路）：一种逻辑电路部件，组合电路的输出完全取决于电路的输入状态。

Common Pathway Bus（公共通道总线）：由许多设备共享的总线（也称为多点总线）。

Compact Disk-Read Only Memory（只读光盘）：参见 CD-ROM。

Compilation（编译）：使用一个编译器的过程。

Compiler（编译器）：负责将整块的源代码一次就翻译成目标代码的一种程序。

Complement（补码）：对布尔表达式或布尔变量进行取反操作的结果。

Completely Connected Network（完全连接网络）：一种所有的部件都被连接到其他部件的网络。

Compression Factor Ratio（压缩系数（比））：测量数据压缩操作的有效性。从数学上讲，压缩系数＝1－［已压缩的文件大小÷未压缩的文件大小］×100%。这里，文件大小的测量用字节来表示。

Computer Architecture（计算机体系结构）：主要关注计算机系统的结构和行为，主要指程序员所见的系统实现的逻辑方面。计算机体系结构的内容包括指令集和指令格式、操作码、数据类型、寄存器的数目和种类、寻址方式、访问主存储器的方法和各种不同的 I/O 机制。

Computer Level Hierarchy（计算机层次结构）：一种抽象的计算机分层结构。大多数的现代计算机

的结构层次分为：从下到上依次为：数字逻辑层、控制层、机器层、系统软件层、汇编语言层、高级语言层和用户层。

Computer Organization（计算机组成）：主要强调如控制信号和存储器类型，以及包括计算机系统的所有物理方面。

Computer Output Laser Disk（计算机输出激光盘）：参见 COLD。

Context Switch（关联转换）：操作系统从一个执行进程到另一个执行进程的转换过程。

Control Bus（控制总线）：用于传输控制信号的一部分总线。

Control Unit（控制单元）：CPU 中用来控制指令执行、数据移动和定时的部分。控制单元既可以是硬连线（由产生控制信号的物理门电路组成）也可以是微程序（利用微代码解释指令和把这些指令转换成各种正确的控制信号）。

CPU Bound（CPU 约束）：一种系统的性能条件。在 CPU 约束条件下，一个或一组程序进程的大部分时间都在执行 CPU 的操作或者等待 CPU 的资源。

CPU Scheduling（CPU 调度）：选择某个正在等待的进程去执行的一个进程。调度的方法包括先来先服务（依次选取队列中的进程），循环复用（给每个进程都分配部分 CPU 的时间），最短作业优先（选取一个执行时间最短的任务）和优先级调度（取决于一些预先确定的因素，比如一个指示进程重要性的编号）。

CRC：参见 Cyclic Redundancy Check。

C-SCAN：参见 SCAN。

Cycle Stealing（周期窃取）：参见 DMA。

Cyclic Redundancy Check，CRC（循环冗余校验）：一种主要用于数据通信的校验和。利用校验和确定在一个大的数据块或信息字节流中是否会有错误发生。

Daisy Chaining（菊花链）：一个 I/O 设备的连接方法。菊花链是指以串行的方式将一个设备的输入和另一个设备的输出连接起来。

DARPAnet，Defense Advanced Research Project Network（国防部高级研究计划网络）：一种美国国防部主持的研究网络系统，通常指最初的因特网。由于这个国防研究计划机构在不同的时期分别被命名为 ARPA（国防研究计划机构）和 DARPA，所以这种最初的网络就被称为 ARPAnet 和 DARPA-net。

DASD，Direct Access Storage Device（直接访问存储设备）：DASD 通常是指连接到大型计算机系统的一个大的磁盘库。DASD 的命名源于这样的思想，即磁盘上的每个磁盘存储单元都有一个唯一的地址，并且系统可以独立地访问这个地址周围的磁盘扇区。

DAT：参见 Serpentine Recording。

Data（数据）：代表某个测量性质的一个数字值，指一个事实。

Data Bus（数据总线）：用于传输实际数据的一部分总线。

Data Dependency（数据相关性）：指在程序执行过程中出现的情况：当没有完全执行一条指令时，却把该指令的结果当成一条后续指令的操作数。这种数据的相关性可能会减慢 CPU 的流水线作业。

Data structure（数据结构）：对大量相关的信息进行组织便于访问数据的一种方式。数据结构通常与数据结构的实现方式无关，因为数据结构的组织方式是一种逻辑上的而不必是物理上的。

Data Token（数据令牌）：表示数据流图中流动的数据的一种符号。在数据流图中，各个节点的激活需要接收到所有必需的数据令牌。

Database Management System，DBMS（数据库管理系统）：能够提供管理服务和保持一组相关文件的顺序性和一致性的软件。

Dataflow Architecture（数据流体系结构）：一种数据控制的计算机体系结构。对于数据流体系结构的机器，是由数据可用性来驱动程序，而不是按照指令执行顺序来控制程序（就像在指令控制的体系

结构中一样)。

Datagram(数据报):一种网络协议数据单元(PDU),可以作为一个单独分立的单元进行路由。数据报通常是两个通信实体之间的一个对话或一个会话的组成部分。因此,数据报还包含保持传输的数据包次序的顺序信息和防止数据包丢失的信息。

Datapath(数据通路):一种由寄存器、ALU 和连接(总线)的网络。指示在系统中数据必须遍历的路径。

DBMS:参见 Database Management System。

Decoder(译码器):一种使用输入值来选择一根特定输出线路的组合电路。

Dedicated Cluster Parallel Computer, DCPC(专用集群并行计算机):一组计算机工作站的集合,专门从事某个特定的并行计算任务。

Demodulation(解调):从一个被调制的模拟信号中提取二进制代码的过程。参见 Modulation。

Dhrystone(基准程序):一个基准程序,主要关注字符串操作和整数操作。1984 年由 Siemens Nixdorf 信息系统公司的 Reinhold P. Weicker 开发的。据报道,命名为 Dhrystone 是一种和 Whetstone 基准有关的双关语,因为"Dhrystone 基准不处理浮点"。

Difference Engine(差分机):1822 年,Charles Babbage 设计的一种计算机器,可以自动求解多项式函数。

Digital Signal 0(数字信号 0):参见 DS-0。

Digital Subscriber Line(数字用户线路):参见 DSL。

Digital Versatile Disks(数字通用光盘):参见 DVD。

Dijkstra's Algorithm(Dijkstra 算法):在网络图中寻找一条最低成本费用的路径的一种算法。算法的基本思想是在所有节点之间的最短连接链路组成的图中找到一个最低费用的路径。

Diminished Radix Complement(基数减 1 的补码):给定一个数 N,基数为 r,占 d 位,则 N 的计数减 1 的补码定义为 $(r^d-1)-N$。对十进制数来说,$r=10$,并且基数减 1 就是 $10-1=9$。

Direct Access Storage Device(直接访问存储设备):参见 DASD。

Direct Addressing(直接寻址):一种计算机的寻址模式。直接寻址是指在指令中直接指定要引用的值的存储器地址。

Direct Mapped Cache(直接映射高速缓存):一种高速缓存映射方案,采用模块的方法将存储器的数据块映射到对应的高速缓存空间块。

Direct Memory Access(直接存储器存取):参见 DMA。

Disk Scheduling(磁盘调度):一种磁盘管理策略,决定访问磁盘扇区的请求的服务顺序。常用的磁盘调度策略有:FCFS(先来先服务)、最短寻道时间优先(SSTF)、SCAN、C-SCAN、LOOK 和 C-LOOK。

Disk Striping(磁盘分带):在 RAID 驱动器中使用的一种映射方法,在 RAID 驱动器中把一些连续的数据块(条带)以循环的方式映射到不同的磁盘驱动器上。

Disk Utilization(磁盘利用率):磁盘忙于服务 I/O 请求所占的时间百分比。磁盘利用率是由磁盘的速率和请求到达服务队列的速率所决定的。从数学上表述为:利用率=请求到达的速率÷磁盘服务的速率。这里,请求到达的速率是每秒钟服务请求的数目,磁盘服务速率就是每秒钟 I/O 操作的次数。

Distributed Computing(分布式计算):指一组网络连接的计算机协同工作去解决某一个问题的情形。

Divide Underflow(除法下溢):等效为计算机被 0 除的操作。在这里,存放在累加器中的除数值太小。

DLL:参见 Dynamic Link Library。

DLT:参见 Serpentine Recording。

DMA，Direct Memory access（直接存储器存取）：一种 I/O 控制方式。DMA 是利用专门的电路（而不是 CPU）来控制 I/O 活动。但是，由于 DMA 和 CPU 共享存储器总线，所以 DMA 要消耗原来由 CPU 使用的存储器周期，这也称为周期窃取。

Dot Pitch（点距）：在显示器上两个最近邻的相同颜色点（或像素）之间的距离。点距越小，图像越清晰。

DRAM，Dynamic RAM（动态随机存储器）：要求对保存的数据进行周期性再充电的 RAM。动态 RAM 与静态 RAM 不同，静态 RAM 只要有电源存在，就可以一直保持所存储的内容。

DS-0（数字信号 0）：指一种信号速率为 64Kb/s PCM（通用载波电话的脉码调制）位串。参见 Plesiochronous Digital Hierarchy。

DSL，Digital Subscriber Line（数字用户线路）：通过公用交换电话网络为居民区和小型办公提供数字服务。两种不兼容的 DSL 技术类型是无载波振幅相位（CAP）技术和离散多音频服务（DMT）技术。其中，CAP 是一种比较老的且较简单的技术，而 DMT 是 ANSI 的 DSL 标准。

DS-x（数字信号 x）：T-载波系统所使用的信号系统。DS-0 是一种通过单向电话会话调制产生的 64Kb/s 信号。由 T-1 载波传输 DS-1 信号，而由 T-3 载波传输 DS-3 信号。

Dual Stack（双堆栈）：使用两种不同协议的通信设备。目前，大多数双堆栈设备（路由器）都支持 IPv4 和 IPv6 协议堆栈。

Duality Principle（对偶原理）：对偶原理证明了布尔等式中的积形式与和形式有相似的关系。

Durable Storage（持久性存储介质）：一种存储介质，无需持续提供电流而可以长久保存数据。例如磁盘、磁带和光盘都是持久性存储介质。

DVD，Digital Versatile Disk（数字通用光盘）：通常也称为数字视频（video）光盘，是一种高密度光盘存储介质。120 毫米直径的单层和双层的 DVD 分别能够容纳 4.7GB 和 8.54GB 的数据容量。

Dynamic Interconnection Network（动态互连网络）：允许两个实体（可能是两个处理器也可能是一个处理器和一个存储器）之间的路径从一种通信方式到另一种通信方式的变化。

Dynamic Link Library，DLL（动态链接库）：对于一个可执行的程序模块，动态连接库是一组链接（程序）装载器使用的二进制对象的集合。

E Carrier System（E 载波系统）：与北美 T 载波系统相对应的欧洲 E 载波系统。E-1 传输速率是 2.048Mb/s，E-3 传输速率是 34.368Mb/s。

EBCDIC，Extended Binary Coded Decimal Interchange Code（扩展的二-十进制交换代码）：是由 IBM 公司发明的一种 8 位代码，它同时支持大小写字母，以及一定数目的其他字符（包括用户自定义的代码），这些其他字符超过了当时的 6 位和 7 位代码的表现能力。

EEPROM，Electronically Erasable PROM（电可擦除可编程只读存储器）：一种可以使用电场进行编程和擦除的 PROM。

Effective Access Time，EAT（有效存取时间）：在存储器体系结构中表示平均访问时间的加权平均值。

Effective Address（有效地址）：操作数在存储器中的实际位置。

EIDE，Enhanced Integrated Drive Electronics（增强型集成驱动电子接口）：一个有较高成本效率比的硬件接口，这是 IDE 接口标准的一种较新的版本，用于计算机及其大容量存储器设备之间的连接。

Elevator Algorithm（电梯算法）：参见 SCAN。

Embedded System（嵌入式系统）：一种将计算机集成到设备中的系统。

Encoding（编码）：一种将普通文本转换成适合于数字数据的存储或传输形式的过程。

Encryption（加密）：采用某种算法和密钥打乱一个消息的过程，这样要读取该消息必须有相应的密钥。

Entropy（熵）：根据信息理论，熵是对消息的信息内容的一种度量。

EPROM，Erasable PROM（可擦除的 PROM）：一种可以进行编程和擦除（使用专用设备）以及再编程的 PROM。

Error Correcting Code（纠错码）：一种代码结构，可以检测错误的存在和自动校正部分或所有的错误。

Error Detecting Code（检错码）：一种代码结构，能够检测到错误的存在。

Ethernet（以太网）：在 1976 年发明的一种主要的局域网络技术。支持的数据传输速率达到 100Mb/s，超过同轴电缆。IEEE 802.3 标准是构建在以太网基础上的，但是它与以太网并不完全相同。

Expanding Opcodes（扩展操作码）：一种指令设计，允许操作码根据指令所要求的操作数的数目来改变指令的长度。

Expansion Bus（扩展总线）：一种外部总线，用来连接外围设备、内部元件、扩展槽和 I/O 端口到计算机其他部分。

Extended Binary Coded Decimal Interchange Code（扩充的二-十进制交换代码）：参见 EBCDIC。

External Fragmentation（外部碎片）：指存储器中的许多小孔（碎片），可以自由地使用这些碎片，但是由于碎片太小而不能保存有意义的信息。参见 Fragmentation。

Fast ATA（快速 AT 附属设备），参见 EIDE。

FC-AL：参见 Fibre Channel。

FDM：参见 Frequency Division Multiplexing。

Fetch-Decode-Execute Cycle（取指-译码-执行周期）：计算机执行程序的指令周期。

Fiber Optic Cable（光纤）：参见 Optical Cable。

Fiber Channel（光纤信道）：一种串行数据传输技术，传输数据的速率可达到 1Gbit/s。光纤信道仲裁环（FC-AL）是三种光纤信道拓扑结构中使用最广泛的，而且价钱最便宜。

Firewall（防火墙）：一种第 3 层的网络层设备（配上相应的程序组），利用基于设备内的编程策略来限制对某个网络的访问。防火墙可以保护网络或某种网络服务不被未经授权的用户访问。

FireWire（串行总线）：一种自行配置的串行 I/O 连接技术，现在是 IEEE 1394 标准。火线支持传统的数据传输，以及同步的输入和输出，其数据传输速率可达 40Mbit/s。

First In，First Out（FIFO）Replacement Algorithm（先进先出的置换算法）：一种置换算法，算法的思想是在存储器中驻留时间最长的项将被替换。

Flash Memory（闪存）：一种 EEPROM，允许成块地编写数据或擦除数据。

Flip-Flop（触发器）：时序电路的最基本部件，可以用作存储单元，即使输入取消后也能保持一个稳定的输出。同时，触发器也是一种最简单的时序电路。触发器和锁存器不同，触发器是边沿触发，而锁存器是电平触发。

Floating-Point Emulation（浮点仿真）：一种针对浮点数操作的程序指令的仿真。

Floating-Point Operations Per Second（每秒钟浮点操作数）：参见 FLOPS。

Floating-Point Unit（浮点单元）：为了优化二进制小数计算的性能而采用的专用计算机电路。

Floppy Disk（软盘）：一种可移动的、低密度的磁性存储介质，是在一个有弹性的聚酯薄膜衬底上涂上一层磁性薄膜而组成的。

FLOPS（每秒钟浮点操作数）：FLOPS 是评价计算机性能的一种过时方法，并没有明确的定义。某些基准程序（例如 Whetstone 和 Linpack）对于测试的系统可以产生每秒钟上百万个浮点操作数（MFLOPS）。

Flux Reversal（磁通量反转）：用在计算机磁带或磁盘存储器设备上的磁性涂层材料的磁极性变化。取决于编码方法的不同，通量反转可以表示二进制数 0 或者 1。

Flynn's Taxonomy（Flynn 分类法）：一种基于并行处理的数据流数目和指令流数目的计算机体系结

构的分类方案。参见 SISD、SIMD、MISD 和 MIMD。

FM：参见 Frequency Modulation。

Forest（森林）：一个或多个不相连的 n 叉树或二叉树的集合。

FPU：参见 Floating-Point Unit。

Fragmentation（碎片（分段））：（1）当存储器或磁盘空间变得不可用时就会产生碎片。参见 Internal Fragmentation 和 External Fragmentation。（2）分段是为了满足特定的网络需求将 IP 数据报分成一个个小部分的过程。

Frequency Division Multiplexing，FDM（频分多路复用）：一种利用介质来传输几个通信流的方法。当 FDM 用在长途电话电缆时，会给每一个呼叫分配一个它自己的频带，允许十多个通话方便地在同一个导体上传输，而不会产生相互之间的干扰。

Frequency Modulation，FM（频率调制）：在数字技术应用时，FM 是一种信息存储或传输中使用的编码方法。在频率调制过程中，每个位单元最少需要有一次信号转变。这些同步的信号转变发生在位单元边界的开始处。

Full Duplex（全双工通信）：一种传输模式。在全双工模式下，一个通信介质或一根数据总线可以同时进行双向的数据传输。

Full-Adder（全加器）：一种执行 3 位二进制数加法的电路，其中一位是一个进位。全加器产生两个输出，一个是输出和，另一个是进位。

Full-Stroke seek（完全（全程）寻道）：磁盘驱动臂从最里层的磁道移动到最外层磁道的过程，反之亦然。

Fully Associative Cache（全关联高速缓存）：一种高速缓存映射方案，允许主存储器中的数据块可以映射到高速缓存上的任何一个空间块的位置。对于高速缓存要求关联存储器。

G：前缀，表示 2^{30}，大约为十亿。

Galois Field（伽罗瓦域）：一个有限元素的代数域。常用的伽罗瓦域是通过使用一个素数进行取模操作来定义：$GF(p) = \{0, 1, \cdots p-1\}$，对于所有的整数 $Z \bmod p$。

Gate（门电路）：一种计算二值信号的各种函数的小型电子部件。

Gateway（网关）：从外部网络进入某个网络的一个入口点。

General Purpose Register Architecture（通用寄存器体系结构）：一种使用一组通用寄存器来保存指令操作数的计算机体系结构。

General Purpose Register（user-visible register）（通用寄存器（用户可见寄存器））：能够被程序员访问和用于各种目的的寄存器。

Geometric Mean（几何平均值）：在计算机性能分析中，经常使用的集中趋势的一种度量标准。几何平均值 G 是 n 个测量值的积的 n 次方根：$G = (x_1 \times x_2 \times x_3 \times \cdots \times x_n)^{1/n}$。几何平均值给出一个一致性的数字，可以利用几何平均值来进行性能比较，而不必考虑数据的分布情况。

Graphics Interchange Format，GIF（图像交换格式）：参见 LZW Compression。

Guided Transmission Media（导向性传输介质）：指如铜线或光缆这样的物理连接器，它们与网络部件直接进行物理连接。

Half Duplex（半双工通信）：一种传输模式。在半双工模式下，通信介质或一根数据总线一次只能进行一个方向的数据传输。

Half-adder（半加器）：一种执行 2 位二进制数加法的电路。半加器产生两个输出，一个是输出和，另一个是进位。

Hamming Code（海明编码）：一种错误纠正代码，代码中增加有一个带有校验位（或冗余位）的信息字节。

Hamming Distance（海明距离）：编码字不同的二进制位的位距。在一个编码的所有字对中的最小

海明距离就是编码的最小海明距离。最小海明距离决定了编码的错误检测和错误校正的能力。

Handshake（握手协议）：在发送方和接收方之间传输数据之前所要求的协议。发送方通过握手协议和接收方进行联络，并启动字节的传输。随后，接收方必须应答发送方的这个请求，并且指出接收数据已经就绪。

Harmonic Mean（调和平均值）：在计算机性能分析中，用于速率或比率测量时经常使用的一种集中趋势的一种度量标准。调和平均值由公式给出：$H=n\div(1/x_1+1/x_2+1/x_3+\cdots+1/x_n)$。

Hazard（冒险）：造成流水线阻塞的一种因素。包括以下方面：数据相关性、资源冲突和存储器提取访问延迟。

Head Crash（磁头撞损）：由于读写头碰到磁盘的表面而引起刚性磁盘的破坏性损坏。

Heap（堆集）：主存储器空间。在进程执行期间，由于数据结构的创建和清除，会引起对主存储器堆集的分配和取消分配。

Helical Scan Recording（螺旋线扫描记录）：一种把数据位存放置到磁盘介质上的方法，让磁带通过一个有两个读写头的倾斜旋转鼓轮（或称为绞盘）。

Hertz（赫兹）：测量时钟频率的度量单位。一赫兹就是每秒一个周期。

Hextet：每四位一组，用来表示一个单个的十六进制数。

High Performance Peripheral Interface（高性能外围设备接口）：参见 HiPPI。

High-order Interleaving（高位交叉存储）：交叉存储器使用地址的高位来选择存储器模块。

HiPPI，High Performance Peripheral Interface（高性能外围设备接口）：一个高容量的存储接口和局域网中的高速链路协议。

Hit（命中）：当在一个指定的存储器层次发现有所请求的数据，或者是在存储器中发现有所请求的页面时，就发生了一个命中。

Hit Rate（命中率）：在给定的存储器层次找到所要求的数据的存储器访问的百分比。

Hit Time（命中时间）：在给定的存储器层次访问请求的信息所需要的时间。

Hollerith Card（穿孔卡片）：由 Herman Hollerith 发明的用于计算机输入输出的穿孔卡片，这个穿孔卡片通常含有 80 列。

Hot Plugging（热插拔）：当计算机正在运行时，添加或删除设备的能力。

Hub（集线器）：一种 OSI 参考模型第 1 层的设备，有多个输入输出端口。负责把从一个或多个位置收到的数据包向一个或多个网络设备进行广播传送。

Huffman Coding（赫夫曼编码）：一种统计的数据压缩方法。赫夫曼编码从输入符号中创建二叉树。输出结果是从输入符号出现的频率和一个译码二进制串的字典中得到的一个二进制编码。

Hypercube Networks（超立方体网络）：网状网络的多维延伸，在每一维都有两个处理器。

ICANN：参见 Internet Corporation for Assigned Names and Numbers。

IETF：参见 Internet Engineering Task Force。

Immediate Addressing（立即寻址）：一种寻址模式，指令要引用的值就跟在指令中操作码的后面。

Indexed Addressing（变址寻址）：一种寻址模式，变址寻址方式使用一个变址寄存器（既可以明显式地指定也可以隐式地指定）来存放一个偏移量（或位移），最后需要把偏移量加上操作数来产生数据的有效地址。

Indirect Addressing（间接寻址）：一种寻址模式，间接寻址使用地址域的几个数位来指定一个存储器地址，这个存储器的地址用作指向实际操作数的一个指针。

Indirect Indexed Addressing（间接变址寻址）：一种寻址模式，间接变址寻址同时采用了间接寻址和变址寻址两种寻址模式。

Inductance（电感）：在导体内抵抗电流变化的一个物理量。同时，当有电流通过导体时，导体附近就会产生磁场。

Industry Standard Architecture Bus，ISA（工业标准结构总线）：20 世纪 80 年代 PC/XT 总线的 IEEE 标准。

Infix Notation（中缀表示法）：操作符和操作数的一种安排顺序，中缀表示法是操作符放在操作数中间，例如 2+3。

Information Theory（信息理论）：主要研究信息本身、信息存储和编码方法的一种理论。

Information（信息）：对人们有意义的数据。

Input-Output Devices（输入输出设备）：一种允许用户和计算机进行通信，或者提供对数据进行读写访问的设备。

Input-Output System（输入输出系统）：一个负责在外部设备和主机之间移动编码数据的子系统部件，一般由 CPU 和主存储器组成。

Institute of Electrical and Electronic Engineers，IEEE（电子与电气工程师协会）：一个致力于电子和计算机工程行业发展的组织。

Instruction Cycle（指令周期）：参见 Fetch-Decode-Execute Cycle。

Instruction Set Architecture，ISA（指令集系统体系结构）：在机器上运行的所有软件和执行软件的硬件之间的协议接口界面。指令系统规定了计算机可以执行的指令和每个指令的格式。

Instruction-based I/O（基于指令的 I/O）：一种 I/O 方法。这里，CPU 有完成输入和输出功能的专用指令。

Integrated Circuit，IC（集成电路）：在第三代计算机中使用的半导体技术。集成电路允许在单个芯片上集成多个晶体管。芯片被封装在一个带有外部引脚的陶瓷或塑料包装中。

Integrated Service Digital Network（综合业务数字网）：参见 ISDN。

Interconnection Network（互连网）：一种连接多个处理器和存储器的网络。

Interface（接口）：一种将计算机系统连接到一个外部实体的设备。硬件接口包括软件、控制单元，以及连接 I/O 设备和计算机的物理连接器。同时，还包括人与机器交互的方式。两种常用的系统接口界面是命令行接口界面和图形用户接口界面（GUI）。

Interleaving（交叉存储技术）：一种对磁盘驱动器中的扇区进行编址的方法。在这里，磁盘扇区在磁道边界周围并不是按连续的次序分布的。这是一种比较老的技术，主要是用来补偿磁盘驱动器的旋转速度和从磁盘中读取数据的速率之间的差别。参见 Memory Interleaving、Low-order Interleaving 和 High-order Interleaving。

Intermediate Node（中间节点）：网络路由器的另一个名字。

Internal Fragmentation（内部碎片）：一个给定的存储空间块的内部碎片。除了已经获得块授权的进程外，其他的任何进程都不能使用这种内部碎片。

International Organization for Standardization，ISO（国际标准化组织）：一个协调全球范围内标准开发活动的组织。

International Telecommunication Union，ITU（国际电信联盟）：一个主要关注通信系统协同工作能力的组织。

Internet Corporation for Assigned Names and Numbers，ICANN（因特网域名管理中心）：一个非盈利性的国际组织，主要负责协调网络地址，网际协议中使用的参数值和高层域名例如 .org 和 .edu 等的分配工作。

Internet Engineering Task Force，IETF（因特网工程任务组）：一个由行业专家组成的松散联盟，主要负责为网际协议制定详细规范。IETF 是在因特网架构委员会（IAB）管理下运作的一个组织，而 IAB 又是在一个非盈利的网际协会（ISOC）的监督下运作的。IETF 所有的提议标准都是以草案（RFC）的形式来发布的。

Internet Protocol（网际协议）：参见 IP。

Internet Service Provider，ISP（网络服务提供商）：一些提供一个或多个网络服务接入点（POP）的私有企业。ISP 通过提供访问网络来赚钱，还常常提供一些增值服务，比如网站。

Internetwork（因特网网络）：由一些使用不同协议的子网组成的网络。

Interpreter（解释程序）：一种通过分析将源代码转换成目标代码的程序，解释程序每次执行一行源代码。

Interrupt（中断）：在系统中，改变（或中断）正常的取指-译码-执行周期的一个事件。

Interrupt Cycle（中断周期）：指令周期的一部分，在中断周期，CPU 会去检查是否有一个中断请求在等待中。如果有，就调用一个中断处理程序。

Interrupt Handling（中断处理）：执行某个特定的程序去处理一个中断程序的过程。

Interrupt-driven I/O（中断控制的 I/O）：一种 I/O 方法。这种 I/O 处理方式最具高效性，因为只有当输入设备有数据准备处理时，才向 CPU 发信号，而其他时间则允许 CPU 去处理其他工作。

I/O bound（I/O 约束）：一种系统的性能条件。在 I/O 约束条件下，一个或一组程序进程的大部分时间都在执行 I/O 操作或者是等待 I/O 的资源。

IP，Internet Protocol（网际协议）：一种无连接的网络层通信协议，IP 以数据包的形式传输信息，这种数据包称为*数据报*（datagram）。每个数据报中包括地址信息和数据信息。

IR，Instruction Register（指令寄存器）：在一个程序中存放下一条将要执行的指令。

ISA Bus（ISA 总线）：参见 Industry Standard Architecture（ISA）Bus。

ISDN，Integrated Service Digital Network（综合业务数字网）：一种过时的通信技术，它试图提供一个统一的公用电话网来传输数据、音频和视频信号。在许多商业安装上 ISDN 都被 ATM 所取代。

ISO Open System Interconnect Reference Model（开放系统互连参考模型）：参见 ISO/OSI RM。

ISO/OSI RM：一种由 7 层结构组成的数据通信协议模型。这 7 层结构分别是：应用层、表示层、会话层、传输层、网络层、数据链路层和物理层。ISO 的工作之所以称为一个参考模型，是因为它的复杂性。实际上，没有一种商业系统使用模型中所严格规定的所有特性。

ISOC：参见 Internet Engineering Task Force。

Isochronous Data（同步数据）：在某种程度上是时间敏感的数据。如果数据传输时间太迟，数据的许多内容（信息）会丢失。实时数据是同步数据的一个例子，例如要求实时传送的语音和视频数据传输。

Joint Photographic Experts Group（联合图像专家组）：参见 JPEG。

JPEG：一种带有损耗的数据压缩方法，是由联合图像专家组资助设计的。JPEG 是一种 8 步骤的压缩方法，允许用户指定在压缩图像中可以接受视觉效果的降级。JPEG2000 是 JPEG 版本中比较低且较复杂的一种软件工具。

K：一种前缀，它的值为 2^{10}或 1024（大约为 1000）。例如，2KB 就是 2048 位。

Kernel（内核）：（1）指当将 I/O 常规程序和其他非 CPU 集约的代码从一个应用程序中剥离后所保留下来的一个功能有限的程序。在创建基准程序套件中常常使用内核程序。（2）指一种规模最小的，但是提供关键功能的程序模块。在操作系统的环境中，内核程序是系统操作时需要持续执行的操作系统部分。

LAN，Local Area Network（局域网）：在同一建筑物内所构建的计算机网络。目前，大多数 LAN 使用以太网技术，速率可达到 100Mb/s。

Latch（锁存器）：一种电平触发的触发器。

Least Recent Used（LRU）Replacement Algorithm（最近最少使用的置换算法）：一种替换算法，置换最近最少使用的项目。

Linear Array Network（Ring Network）（线性阵列（或环形）网络）：允许任何一个实体可以和它的两个邻居实体直接进行通信，但是任何一种其他通信都必须要经过多个实体后才能到达目的地。

Tree Network（树形网络）：一种在非循环树状结构中排列实体的网络。

Link Editor（链接编辑器）：参见 Linking。

Linked List（链表（链接列表））：一种数据结构，其中每个元素都包含一个指针，这个指针既可以是空指针，也可以指向同种类型的另外一个数据元素。

Linking（链接）：一个程序的外部符号和来自其他文件的所有输出符号的一种匹配过程，链接结果产生一个单一的没有不可决定的外部符号的二进制文件。

Linpack（线性代数程序包）：一种用来度量浮点性能的软件。这种线性代数包是一组称为基本线性代数子程序（BLAS）的子程序的集合，使用双精度算术来求解线性方程的系统。

Little Endian（小端位序）：一种存储器的位序安排，采用小端位序是将多字节字的最低字节存放在存储器中的最低地址的位置。

Load-Store Architecture（装载存储体系结构）：一种计算机体系结构。这种结构的指令系统中只有装入和存储这两条指令可以访问存储器。所有其他指令都必须使用寄存器访问数据。

Local Area Network（局域网）：参见 LAN。

Local Bus（局域总线）：在 PC 机中，外围设备直接连接到 CPU 中的一种数据总线。

Local Loop（本地回路）：通常指连接中央电话交换局、住宅区和小型企业所在地的低带宽的铜双绞线。本地回路有时也称为“最后一里”，尽管客户所在地和中央电话交换局之间的连接经常超过一里。

Locality，Locality of Reference（引用的局部性）：计算机程序的一种特性，程序倾向于以集中成团成簇的形式访问数据或指令。

Logical Partition（逻辑分区）：一种计算机系统的非物理性分区，逻辑分区会产生一种在物理上有分立的实体分割的幻觉。

LOOK（C-LOOK）：一种磁盘调度算法。在这种算法中，只有当所请求的最外层磁道和最内层磁道都完成数据的读取或写入后，磁盘臂才会改变运动方向。LOOK 还有一种变化形式，称为 C-LOOK（即循环 LOOK），在这里磁盘上最高编号的磁道和 0 磁道被认为是相邻的磁道。

Loop Fission（循环分裂）：大循环分裂成小循环的过程。在循环体优化时，循环分裂有重要作用，可以消除数据相关性和减少由于冲突引起的高速缓存延迟。参见 Loop Peeling。

Loop Fusion（循环并合）：组合多个使用相同数据项的循环的过程。循环并合可以增强高速缓存的性能，增加指令级的并行执行和减少循环开销。

Loop Interchange（循环交换）：一种重新排列循环的过程，可以按照更加接近于数据存储的方式访问存储器。

Loop Peeling（循环剥离）：一种类型的循环分裂。指从循环体中移去开始语句或结尾语句的过程。

Loop Unrolling（循环展开）：一种循环展开的过程。循环展开后每次新的迭代都包括几个原来的迭代过程，这样每个循环迭代都可以完成更多的计算。

Loopback Test（回送测试）：用来检查通信设备的功能和运行在主机系统上的协议。在回送测试期间没有数据进入网络。

Loosely-coupled Multiprocessors（松散耦合的多处理器）：具有物理上的分布式存储器的多处理机系统。也称为分布式系统。

Low-order Interleaving（低位交叉存储）：交叉存储器使用地址的低位来选择存储器模块。

LPAR：参见 Logical Partition。

Large Scale Integration，LSI（大规模集成电路）：每个芯片上包含 1000 到 10000 个元件的集成电路。

LZ77 Compression（LZ77 压缩）：一种使用文本窗口的数据压缩方法。这种压缩方法中，文本窗口用作为一个字典，并与一个包含被编码的信息的前视缓冲器协同作业。如果能够在字典中找到任何一

个前视缓冲器里面的字符，那么窗口中文本的长度和位置都被写入到输出。如果没有找到文本，则这个未编码的符号被写入时要增加一个标志位，用来指出这个符号应该被用作一个文字。

LZ78 Compression（LZ78 压缩）：一种不同于 LZ77 的一种数据压缩方法。它取消了固定大小的文本窗口的限制。相反，它增加了一个带有输入令牌的树形结构。整个 trie 都会跟在编码信息后面被写入到磁盘，而在译码信息之前首先读取这个 trie。

LZW Compression（LZW 压缩）：一种更有效的实现 LZ78 压缩技术的方法，在这里，trie 的大小受到了严格的控制。LZW 是 GIF 数据压缩方法的基础。

M：前缀，2^{20}或 1048576（大约 100 万）。例如，2MB 就是 2^{21}字节，或者大约 200 万字节。

MAC：参见 Medium Access Control。

MAC Address（MAC 地址）：固化在网络接口卡（NIC）电路中的一个唯一的 6 字节物理地址。前 3 个字节是制造厂商的标识码，由 IEEE 来指定。最后 3 个字节是制造厂商分配给 NIC 的一个唯一的标识码。在全世界范围内，没有任何两个网络接口有同样的 MAC 地址。网络协议层映射这个物理 MAC 地址到最少一个逻辑地址。

Main Memory（主存储器）：用于存放程序指令和数据的存储器。通常由 RAM 存储器来实现。

MAN，Metropolitan Area Network（城域网）：城域网是一种高速网络，主要用于覆盖城市和城市的周边地区。

Manchester Code（曼彻斯特编码）：也称为相位调制（PM）编码，是信息存储或传输中使用的一种编码方法，无论是一个 0 还是一个 1，每位都提供一个信号转变。在 PM 中，每个二进制的 1 都由一个向上的信号转变来表示，而每个二进制的 0 都由一个向下的信号转变来表示。如果需要，会在每个位单元的边界提供一个额外的信号转变。

Mantissa（尾数）：在科学计数法中一个数的尾数部分。与指数部分不同的是，指数表示基数的指数幂，尾数乘以指数就得到了要求的数值。

MAR，Memory Address Register（存储器地址寄存器）：一种保存要引用的数据的存储器地址的寄存器。

Maskable Interrupt（可屏蔽中断）：能够被忽略或被中止的一类中断。

MBR，Memory Buffer register（存储器缓冲寄存器）：一种既可以保存刚从存储器中读取的数据又可以保存准备好写入到存储器中的数据的寄存器。

Mean Time to Failure（平均失效时间）：参见 MTTF。

Medium Access Control（MAC）（介质访问控制）：一种连接节点到网络的方法。介质访问控制两种主要的方法是令牌传递和载波监听/冲突检测（CSMA/CD）。

Medium Access Control Address（介质访问控制地址）：参见 MAC Address。

Memory Bound（存储器约束）：一种系统的性能条件。在 CPU 约束条件下，一个或一组程序进程的大部分时间都在执行主存储器的操作或者是等待存储器的资源。

Memory Hierarchy，Hierarchical Memory（存储器分层结构）：不同层次的存储器的使用方法。每层的存储器都有不同的访问速度和存储容量，主要是为了取得更好的性价比，而不是只使用一种单一存储器类型。

Memory Interleaving（存储器交叉存储）：把存储器分割成多个存储器模块（或库）的一种技术。参见 Low-order Interleaving 和 High-order Interleaving。

Memory-mapped I/O（存储器映射的 I/O）：指当接口中的寄存器出现在计算机的存储器映射中时，访问存储器和访问 I/O 设备之间并没有真正的区别。

Memory-Memory Architecture（存储器-存储器体系结构）：一种计算机的体系结构。这种结构允许一条指令执行一个操作，而这种操作并不要求至少有一个操作数位于寄存器中。

Mesh Network（网状网络）：每个实体都连接 4 个或 6 个（取决于这种网络是二维网络还是三维网

络）相邻实体的一种网络。

Message latency（消息延迟）：一个消息的第一位到达目的地所需要的时间。

Metric（度量）：描述系统性能的一个数字。

Metropolitan Area Network：See MAN。

MFM：参见 Modified Frequency Modulation。

Microcomputer（微型计算机）：使用一个微处理器的计算机。

Microkernel（微内核）：一个操作系统的部件，指一个相对较小的程序进程，它提供基本的操作系统功能，并协助外部模块来完成特定的任务。

Microoperations（微操作）：一种“微型”指令，规定了一些可以对寄存器中的数据所执行的基本操作。

Microprocessor（微处理器）：在单个芯片上完成 CPU、存储器和 I/O 功能的处理器。

Microprogram（微程序）：一种用来把指令转换成机器语言的软件。

Middleware（中间件）：一种粗略的软件分类法。指位于操作系统层之上，而在应用程序层之下运行的提供服务的程序软件。

Millions of Instructions per second（每秒钟百万条指令）：参见 MIPS。

MIPS，Multiple Instruction，Multiple Data（多指令多数据）：一种计算机体系结构，采用多个控制点，每个控制点都有其自己的指令和数据流。

Minuend（被减数）：在一个算术减法中，被减数减去减数得到的结果。

MIPS（每秒钟百万条指令）：一种过时的测量计算机系统性能的方法。从数学上，MIPS＝（执行的指令数）÷（执行时间×10^6）。MIPS 过于依赖计算机的体系结构而用处不大。这个度量单位还引起了首字母缩写的创造性解释，例如，“处理器速度的误导指示器”和“推销员业绩的无意义的指示器”。

MISD，Multiple Instruction，single Data（多指令单数据）：一种计算机体系结构，可以在同一个数据流上操作多个指令流。

Miss Penalty（缺失损失）：处理一个缺失所需的时间，其中包括在替换一个上层存储器中的数据块的时间，再加上传送请求的数据到处理器的时间。

Miss Rate（缺失率）：在一个给定的存储器层级上，没有找到所要求的数据的存储器访问的百分比。

Miss（缺失）：当在给定的存储器层次上找不到所要求的数据时，就会发生缺失。

MNG，Multiple-image Network Graphics（多图像网络图形）：MNG 是 PNG 的一种扩展，它允许压缩多个图像到一个文件中。这些文件可以是任何类型的文件，例如灰度、真彩色或者甚至是 JPEG。

Modem，Modulator/Demodulator（调制解调器）：调制是将数字信号转换成模拟信号，并可以利用模拟线路（电话线）进行传输，而解调是把传送过来的模拟信号变换成等效的数字信号。

Modified Frequency Modulation，MFM（调制的频率调制）：数据存储或传输的一种编码方法。这种方法只是在连续 0 之间的位单元边界才提供信号转变。对于 MFM，每一对位单元只需至少提供一个信号转变，这不同于 PM 或 FM 中每一个位单元都需要一个信号转变。

Modulation（调制）：指按照信息信号改变载波特性的过程。数字信号可以通过模拟载波进行传输，这里只需要改变模拟载波信号的某些特性来表示一个二进制的代码。可以把模拟信号振幅、频率或者是振幅和频率同时调制成能够传输二进制信号的载波。参见 Modem。

Moore's Law（摩尔定律）：一个预测定律：硅芯片的集成度每隔 18 个月将增加一倍。

MPP，Massively Parallel Processor（大量信息并行处理机）：一种 MIMD 的分布式存储器体系结构，这种结构中的多处理器并不共享存储器。

MSI，Medium Scale Integration（中规模集成电路）：每个芯片上有 100 到 1000 个元件的集成电路。

MTTF，Mean Time to Failure（平均失效工作时间）：MTTF 是一个元部件寿命的数学期望值，

MTTF 是利用制造业中通常使用的统计质量管理方法推演出来的。这只是一个理论值，并不一定反应了元部件的实际寿命。

Multicast（多播）：一种网络消息发布方法，发送一个消息由多个节点读取。

Multiple-image Network Graphics（多图像网络图形）：参见 MNG。

Multiplexer（多路复用器）：一种有多个输入一个输出的组合电路，利用控制线来选择多个输入线中的某一条线路，并直接连接到单一输出线路上。

Multiplexing（多路技术）：在许多个互不相关的、独立的连接中共享一个通信介质。一个数字载波中，通过交叉时间段分配给每个连接一个信道。而在宽带载波的情况下，通过宽带介质分配给每个连接一个特定波长（频率）的载波。

Multipoint Bus（多点总线）：由多个设备共享的总线，也称为公用通路总线。

Multiprocessor System（多处理器系统）：包含多个 CPU 的计算机系统。

Multiprogramming（多道程序处理）：在单个 CPU 中，多个程序的并发执行。

Multistage Interconnection Network（Shuffle Network）（多级互连网络）：使用 2×2 开关和多级状态构造的交换网络，例如 Omega 网络，也称为洗牌网络。

Multitasking（多任务处理）：并发地运行多个进程。多任务处理不同于多道程序处理，多道程序处理中的多个程序进程总是属于同一个用户。

Multithreading（多线程技术）：为了增加并发性，将一个进程细分为多个不同的控制线程的过程。

NAP，Network Access Point（网络访问点）：为了连接其他的 ISP，由区域网络服务提供商使用的交换中心。

Narrowband Cable（窄带电缆）：一种为单频传输优化设计的导向网络媒体类型。

N-ary Tree（N 叉树）：由树根、内部节点和叶子组成的一个非循环的数据结构。树根和内部节点最多可以有 n 个指针指向其他节点（即最多有 n 个分支）。而没有分支节点的节点是叶子。

Network Access Point（网络访问点）：参见 NAP。

Network Interface Card，NIC（网络接口卡）：一种 I/O 扩展电路板，它常常包含 OSE 协议堆栈中最少的三个协议层。网络接口卡会将系统总线上传输的并行数据转换成串行信号在通信介质上传播。网络接口卡还负责把系统数据从二进制数转换为网络使用的编码数据，反之亦然。

Network of Workstation（NOW）（工作站网络）：一组并行运行的分布式工作站的集合，而这些网络节点不能用作常规的工作站。

Neural Network（神经网络）：一种计算机系统的类型。这种计算机由大量简单的处理单元组成，这些处理单元都独立地处理一个非常大的问题的一小部分。这些处理单元必须通过使用一种特殊的学习算法经历训练过程。

NIC：参见 Network Interface Card。

Noise（噪声）：一种干扰信号准确传输的电学现象，噪声强度通常用分贝（dB）来度量。

Nonblocking Interconnection Network（无阻塞型互连网）：一种网络类型，无阻塞型互连网络允许在有其他同时发生的连接存在的情况下建立新的连接。

Nonmaskable Interrupt（非屏蔽中断）：一种高优先级的中断，不能够被中止，而且必须要得到应答。

Non-Return-to Zero（NRZ）（不归零编码）：一种为数据传输设计的编码方式，在 NRZ 编码中，1 总是表示高，0 总是表示低，或反之亦然。典型情况下，“高”为＋5v 或＋3v，“低”为－5v 或－3v。如果发送方和接收方不能够严格同步的话，则这种编码是无效的。在磁性存储器中，NRZ 编码是由磁通量反转来实现的。

Non-Return-to-Zero-Invert（NRZI）（反转不归零编码）：一种用于数据存储和传输的编码方式，这种编码为二进制数 1 提供一种信号转变，而对二进制数 0 没有信号转变。信号转变可能是从高到低，也

可能是从低到高。这种频繁的信号转变有助于保持信号同步。

Nonuniform Memory Access，NUMA（非均匀存储器访问计算机）：一种共享存储器的 MIMD 计算机。MIMD 机上的每个处理器配有自己的一块存储器空间，从而会导致附近的存储器访问要比访问属于其他处理器的存储器所花费的时间少。

Normalization（归一化（或规格化））：（1）在计算机性能分析中，归一化是表示统计的性能测量的一种方法，归一化性能是度量系统性能和一个标准系统性能的比值；（2）在浮点表示法中，对一个数进行规格化就是要调整表示中的指数使得小数部分最左边的位为 1。

NOW：参见 Network of Workstation。

NRZ：参见 Non-Return-to-Zero。

NRZI：参见 Non-Return-to-Zero-Invert。

N-Tiered Architecture（N 层体系结构）：这是一种执行环境，其中进程在一个以上的计算机系统上执行。客户服务器系统通常使用一个 3 层的体系结构，第一层是桌面计算机，第二层是应用程序服务器，第三层是数据库服务器。应用程序服务器负责管理这些应用程序，以及桌面计算机和数据库服务器之间的交互作用。

Nybble or Nibble（半字节）：一个字节的一半。字节是由一个高半字节和一个低半字节组成。

Nyquist Law（奈奎斯特定律）：奈奎斯特定律证明，没有信号能够以比它的频率快两倍的速率来传输信息。用公式表示为：最大数据率$=2\times$带宽$\times\log_2$（信号电平数）波特。

Octet（八位字节）：（1）利用三位一组来表示一个八进制数字；（2）在互连网中指的是一组 8 个相邻的位（在其他地方也称为一个字节）。

OC-x：参见 SONET。

One's Complement Notation（反码表示法）：又称为 1 的补码表示法，是一种用来表示带符号的二进制数值的方法。正数简单地用符号幅值格式来表示，对于负数，则将相应的正数表示中的所有位翻位（按位取反）。

Opcode（操作码）：指令中规定所要执行操作的部分。

Operating System（操作系统）：一种控制计算机系统整体操作的软件，操作系统包括：进程调度和管理，进程保护，存储器管理，I/O 操作和安全。

Operation Counting（操作计数）：一个进程，负责计算在一个循环中所执行的指令类型的数目，然后决定每种指令类型所要求的机器周期的数目。接着这些信息会越来越实现更好的指令平衡，如果可能的话，将提高程序的性能。

Optical Cable（光缆）：一种导向网络媒体的类型，通常称为光纤电缆。光缆是由一束细小（1.5 到 125μm）的玻璃纤维或塑料线绳组成，外面用一个防护塑料套缠绕。光纤传导光就类似于铜导线传输电流和水流过管道一样。

Optical Carrier（光纤载波）：参见 SONET。

Optical Jukebox（光盘柜）：一种自动光盘存储库，可以直接访问大量的光盘。光盘柜能够存放几十个到几百个光盘，总容量可以达到 50 到 1200GB 或者更多。

Overclocking（超频）：一种用来提高计算机系统性能的方法，将一些特定的系统部件的性能应用到极限。

Overflow（溢出）：如果寄存器容量不足以去存放一个算术运算的结果时，就会发生溢出。在带符号的算术运算中，当进入符号位的进位与移出符号位的进位不相等时，就检测到一个溢出的情况。

Overlay（覆盖）：一种存储器的管理方法。在这里，程序员需要控制装入程序子模块的定时同步。现在，这个任务通常是由系统的存储器管理设备自动完成的。

Packed Numbers（压缩码）：在一个字节中，利用相邻 4 位字节来存放 BCD 编码的值，并留给符号一个 4 位字节。

Page Fault（页面错误（缺页））：当一个请求的页不在主存储器中，而必须从磁盘复制到存储器中时，就发生了缺页。

Page Field（页域）：地址中用来指定请求数据的所在页（可能是虚拟页，也可能是物理页）的部分。

Page Frames（页帧）：在实现虚拟存储器时，主存储器（物理存储器）被分成的相同大小的空间块，称为帧。

Page Mapping（页映射）：一种把虚拟地址转换为物理地址的机制。

Page Table（页表）：一个记录有进程的虚拟页的物理位置的记录表。

Pages（页）：虚拟存储器（逻辑地址空间）被分成固定大小的空间块，每个空间块的大小和一个页帧相等。虚拟页被存放到磁盘上，直到需要时才调入存储器。

Paging（分页）：（1）一种实现虚拟存储器的方法。分页时，主存储器被划分为固定大小的空间块（帧），而程序也被划分为同样大小的块；（2）把一个虚拟页面从磁盘中复制到主存储器中的一个页帧的过程。

Parallel Communication（并行通信）：通过通信介质一次传输一个整字节（或整字）的通信。通信介质（数据总线或外围设备接口电缆）必须为每一位都提供一根传输导线。还需要有其他导线负责管理数据的交换。时钟信号（选通脉冲）对正确地处理并行数据传输是非常关键的。参考 Serial Communication。

Parity（奇偶校验）：最简单的错误检测方法，它是一个字节中 1 的和的函数。根据字节中所有其他位的和是奇数还是偶数，奇偶校验位来决定是“开”还是“关”。

Parking Heads（停靠磁头）：当硬盘系统关断电源时，读写磁头会退到一个安全的地方，以防止损坏介质。

PC（Program Counter）（程序计数器）：保持程序中要执行的下一条指令地址的寄存器。

PCI：参见 Peripheral Component Interconnect。

PCM：参见 Pulse-Code Modulation。

P-code Language（P 代码语言）：一种既能够进行编译又能够进行解释的语言。

PDH：参见 Plesiochronous Digital Hierarchy。

PDU：参见 Protocal Data Unit。

Perceptron（感知器）：在神经网络中的一个可训练的神经元。

Peripheral Component Interconnect，PCI（外围设备互连）：一种 Intel 公司发明的局部总线标准，支持多种外部设备的连接。

Phase Modulation（PM）：参见 Manchester Code。

Physical Address（物理地址）：物理存储器的实际地址。

Pile of PCs（POPC）（PC 机群）：一组不同类型的专用硬件，用来构建一个并行计算机系统。BEOWULF 就是 POPC 的一个例子。

Pipelining（流水线作业）：一种计算机技术，通过把取指-译码-执行周期分割成一些较小的步骤（流水线级），在流水线中这些较小的步骤可以相互重叠和并行执行。

Plesiochronous Digital Hierarchy，PDH（准同步数字体系）：一组载波速率的集合：从 T-1 到 T-4，通过连续使用多路技术形成的。这种体系之所以称为准同步（区别于同步）体系，是因为每个网络元件（例如交换机或多路复用器）都有自己的时钟，这些时钟与其层次上面的时钟保持周期性的同步。与一个实际的同步网络的区别是，在载波上没有设置单独定时信号。在北美 PDH 已经被 SONET 系统所取代，而在世界的其他地区 PDH 被 SDH 系统所替换。

Plug-and-Play（即插即用）：计算机自动配置设备的能力。

PM：参见 Manchester Code。

PNG，Portable Network Graphics（可移植网络图形格式）：一种数据压缩方法，首先使用赫夫曼编码方法编码消息，然后再使用 LZ77 压缩方法再次压缩消息。

Pointer（指针）：一个存放在寄存器或存储器中作为一个数据值的存储器地址。

Point-of-Presence（存在点）：参见 Internet Service Provider。

Point-to-point Bus（点对点总线）：在系统中连接两个特定设备的一条总线。

Polling（轮询）：系统不断地检查寄存器或通信端口，测试是否有数据信号存在。

POP：参见 Internet Service Provider。

Port（端口）：（1）计算机系统中的一种连接插口（接口），用来提供对 I/O 总线或控制器的访问；（2）在 TCP/IP 协议套件中，端口是标识某个特定协议服务的一个数字值。

Portable Network Graphics（可移植网络图形格式）：参见 PNG。

Postfix Notation（后缀表示法）：一种操作符和操作数的排序方法。后缀表示法是把操作符放在操作数后面。例如 23+。参见 Reverse Polish Notation。

POTS，Plain Old Telephone Service（简易老式电话业务）：简易老式电话业务提供模拟服务和少量的增值服务。

Prefetching（预取）：一种减少访问存储器或磁盘的技术。使用预取技术时，会从存储器中读取包括请求页的多个后续页面，或者从磁盘中读取包括请求扇区的多个连续扇区，期望而这些后续的页面或扇区中的一个或多个不久就需要被用到。

Prefix Notation（前缀表示法）：一种操作符和操作数的排序方法。前缀表示法是把操作符放在操作数的前面，例如+23。

Price-Performance Ratio（价格性能比）：一种建立在系统规格性能上的某个特定系统的“价值”的度量方法。从数学上，价格性能比是系统价格除以一种有意义的系统性能度量的比值。价格性能比要充分考虑系统性能度量方法和系统拥有者要承担的总费用才有意义。

Principle of Equivalence of Hardware and Software（硬件和软件等效性原理）：这种原理说明了硬件可以完成软件能够做的任何事情，而软件也能够实现硬件所有功能。

Product-of-Sum Form（和之积的形式）：一种布尔表达式的标准形式，是一组和项的与（AND）操作的集合。

Profiling（细化重组）：一种程序分拆过程，把一个程序代码划分为一些小的程序块，对每个小程序块进行时间分配，决定哪些块会占用大部分的程序时间。

Program Counter Register（PC）（程序计数器寄存器）：一种专用寄存器，用来保存下一条将要执行的指令地址。

Programmed I/O（程序控制的 I/O 方式）：在这种 I/O 方式中，CPU 必须等待 I/O 模块完成一个请求的 I/O 操作后才可以执行其他操作。

PROM，Programmable ROM（可编程的只读存储器）：一种可以用适当的设备对 ROM 进行编程的只读存储器。

Protection Fault（保护错误）：当一个进程试图去使用受到另外一个进程或操作系统保护的存储器时所产生的一种出错情况。

Protocol 协议：通信实体在进行信息交换时所要遵守的一组规则。

Protocol Data Unit，PDU（协议数据单元）：一种包含协议信息和数据有效载荷的数据通信包。

PSTN：参见 Public Switched Telephone Network。

Public Switched Telephone Network，PSTN（公用电话交换网）：一种公用通信设施的系统，包括传输线路、交换系统和其他设备。

Pulse-Code Modulation，PCM（脉码调制）：在电话通信中，将模拟信号转化为数字信号的一种方法。模拟信号采样频率是 8000 次/秒。在每次采样时，都为信号波的值分配一个适合于数字传输的二

进制值。参见 Quantization。

QAM：参见 Quadrature Amplitude Modulation。

QIC：参见 Serpentine Recording。

Quadrature Amplitude Modulation（正交幅度调制）：一种信号调制方法，同时改变载波信号的相位和振幅进行数字信息编码。

Quantization（量化）：将一组输入映射到一组输出上的近似函数。通常是把实数（或连续模拟信号）值转化为整数值，使得这些输入值在数字介质中是可以表示的。

Queue（队列）：一种为先来先服务（FIFO）的数据处理方式而优化设计的数据结构。队列结构按照元素到达队列时的相同顺序移走队列中的元素。

Race Condition（竞争状态）：数据最终的状态并不取决于更新的正确性，而是取决于访问数据顺序的情况。

Radix Complement（补码）：已知一个数值 N 占据 d 位，基数为 r，N 的补码被定义为：r^d-N，对于 $N\neq 0$；而 $N=0$ 时，补码为 0。

Radix Point（小数点）：用来区别数的整数部分和小数部分的分隔符。

RAID（独立冗余磁盘阵列）：一种通过使用大量廉价的（独立的）小磁盘来代替一个昂贵的大磁盘来提高可靠性和性能的存储系统。这个系统最初的名字为：冗余廉价磁盘阵列。现在，RAID 通常被解释为独立冗余磁盘阵列。

RAID-0：也称为磁盘跨区技术，把数据分放到几个磁盘的条带上。RAID-0 没有冗余。

RAID-1：也称为磁盘镜像技术，把数据的两份副本分别写到两个独立的磁盘上。

RAID-10：RAID-0 分条技术和 RAID-1 镜像技术的结合。RAID-10 具有优良读取性能，同时提供最佳的可用性。

RAID-2：包含一组数据驱动器和一组海明驱动的 RAID 系统。数据的一位被写入到一个数据驱动器中，同时海明码驱动器为数据驱动器提供错误恢复信息。RAID-2 是一种理论上的 RAID 设计，没有进行商业实现。

RAID-3：一种比较流行 RAID 系统，采用每次一位的方法把数据分别存放到所有的数据驱动器上，同时利用一个驱动器来存储一个简单的奇偶校验位。利用硬件对每个数据位上使用 XOR 操作可以很快地完成这种奇偶校验位的计算。

RAID-4：一种理论上的 RAID（像 RAID-2）。一个 RAID-4 的阵列，与 RAID-3 一样，由一组数据磁盘和一个奇偶校验磁盘组成。不是采用每次一位的方法将数据写入到所有的数据驱动器，RAID-4 是按照统一大小的条带方式写入数据，就像 RAID-0 中所描述的，在所有的数据驱动器上都生成一个条带。条带中的各个数据位相互进行异或（XOR）操作生成奇偶校验的条带。从本质上来说，RAID-4 是带有奇偶校验的 RAID-0。

RAID-5：一种构建在 RAID-4 基础上的流行 RAID 系统，它的奇偶校验磁盘分布在整个磁盘阵列。

RAID-6：一种 RAID 系统，其中对于每一横列（或水平行）的驱动器采用了两组纠错条带。除了奇偶校验外，还使用 Reed-Soloman 纠错编码增加了第二层的保护。

RAM，Random Access Memory（随机访问存储器）：在计算机上用来存储程序和数据的一种易失性（非永久性）的存储器。每个存储单元都有一个独一无二的地址。

RAMAC，Random Access Method of Accounting and Control（会计和控制随机存取计算机）：RAMAC 是在 1956 年由 IBM 公司发布的第一台商用的磁盘计算机系统。

Real-Time System（实时系统）：一种处理物理事件的实时数据的计算机系统，由于将与物理事件发生同时反应，所以这种系统要求有严格的定时限制。在硬实时系统中，不能有定时错误发生。如果不能满足时限的要求，则可能会有致命的结果。而在软实时系统中，也要满足时限的要求。但是如果超过了时限约束，还不至于导致灾难性的后果。

Recording Mode（记录模式）：参见 CD Recording Mode。

Reed－Soloman（RS）（里德-所罗门编码）：可以被认为是一种 CRC 编码，但它是对整个字符进行操作，而不是只对几个位进行操作。RS 编码，与 CRC 编码一样，是一种系统性的编码：奇偶校验字节会被加到一个信息字节块的后面。

Reentrant Code（可重入代码）：可使用不同数据的一种代码。

Refresh Rate（刷新率）：在屏幕上一个图像被重复显示的速率。

Register（寄存器）：一种存储二进制数据的硬件电路。寄存器位于 CPU 里面，速度非常快。有些寄存器是用户可见的，有些是用户不可见的。

Register Addressing（寄存器寻址）：一种寻址模式。在寄存器寻址方式中，寄存器中的内容作为一个操作数。

Register Indirect Addressing（寄存器间接寻址）：一种寻址模式。寄存器中的内容用作一个指向操作数实际存储位置的指针。

Register Transfer Notation，RTN（寄存器传输表示法）：也称为 Register Transfer Language（RTL），寄存器传输语言：用来描述微操作行为的一种符号表示法。

Register Window Sets（寄存器窗口组）：也称为重叠寄存器窗口（Overlapped Register Windows）：是在 RISC 体系结构中使用的一种技术，通过简单地改变当前的执行过程对不同寄存器的可见性来实现参数传递。

Register-Memory Architecture（寄存器-存储器体系结构）：一种计算机的体系结构，要求至少有一个操作数位于寄存器中，而另一个操作数位于存储器中。

Repeater（转发器）：一种 OSI 参考模型中第 1 层的设备，用来对长距离网络电缆中传输的信号进行放大。

Replacement Policy（置换策略）：一种用来选择一个要被置换的高速缓存牺牲块或页面的策略。这种策略在组关联高速缓存和分页技术中是必需的。

Request for Comment（请求注解）：参见 Internet Engineering Task Force。

Resident Monitor（常驻监控程序）：一种早期类型的操作系统，允许程序能够在没有人为干预（除了在读卡机上放上一组卡片外）的情况下被处理。它是现代操作系统的前身。

Resource Conflict（资源冲突）：如果两条指令需要使用同一个资源，就会发生资源冲突。资源冲突会减慢流水线的 CPU 速度。

Response Time（响应时间）：一个系统或者是一个系统部件执行某项任务所需要的时间。

Reverse Polish Notation，RPN（反向波兰表达式）：也称为后缀表示法。一种操作符和操作数的排序方法，RPN 把操作符放在操作数后面，例如 23＋。

RISC，Reduced Instruction Set Computer（精简指令集计算机）：一种计算机体系的设计思想，每个计算机的指令只完成一种操作，所用指令的大小相同，指令的体系结构的区别很小，而且所有的算术运算都必须在寄存器之间来完成。

RLL：参见 Run-Length-Limited。

Robotic Tape Library（自动磁带库）：也称为磁带仓库，一个自动磁带库系统可以装载和卸下，并且跟踪记录（编目录）大量的磁带盒。自动磁带库的总容量达到几百千兆字节，并且根据用户的请求，在不到半分钟的时间内能够装载一个磁带盒。

Rock's Law（洛克定律）：摩尔定律的一个推论，它说明了用来制造半导体芯片的主要设备的成本费用将每 4 年增加一倍。

ROM，Read-only Memory（只读存储器）：一种可以永久性地保留数据的存储器。

Rotational Delay（旋转延迟）：将一个目标扇区定位到磁盘读写头的下方所需要的时间。

Router（路由器）：一种最少可以连接两个网络的复杂硬件设备，路由器可以决定数据包应该被发

送的目的地。

Run-length-limited，RLL（运行长度限制编码）：一种编码方法。在这里，一些块字符编码字，例如ASCII或EBCDIC，会被转换成一些专门设计的限制编码中出现连续0的个数的编码字。RLL（d，k）编码允许在任何一对连续1之间最少有d个和最多有k个连续的0出现。

SCAN（C-SCAN）：一种磁盘调度算法。在SCAN算法中，磁盘臂连续不断地扫描磁盘的表面，直到遇到服务队列中请求的磁道时才会停止。这种方法又称为电梯算法，因为它非常类似于摩天大厦中的电梯服务乘客的方式一样。SCAN有一种变体，称为C-SCAN（循环SCAN），这里将磁盘上的磁道0看成是与最大编号的磁道相邻的磁道。

SCSI，Small Computer System Interface（小型计算机系统接口）：一种计算机接口技术，允许多个设备连接成菊花链形式，并且通过一个单一主机适配器对这些设备进行单独编址。现在，人们已经通过SCSI-3的结构模型（SAM）扩展了多种连接方法。SAM是一种带有协议的分层系统，可以实现层与层之间的通信。SAM还将串行存储器体系结构（SSA）、串行总线（也称为IEEE 1394，或火线）、光纤信道和类属包协议（GPP）合并成统一的体系结构模型。

SDH：参见Synchronous Digital Hierarchy。

SDRAM，Synchronous Dynamic Random Access Memory（同步动态随机存储器）：与经过优化的处理器总线的时钟速率达到同步的存储器。

Secondary Memory（辅助存储器）：远离CPU位于系统本身之外的一种存储器。例如，磁盘、磁带和CD-ROM等。

Seek Time（寻道时间）：磁盘臂定位到目标磁道所需的时间。

Segment Table（段表）：一个记录进程段的物理位置的表。

Segmentation（分段）：类似于分页，区别在于，不是把虚拟地址空间分为相等的和固定大小的页面，以及把物理地址空间分为相等大小的页帧，而是把虚拟地址空间划分为一些逻辑的、可变长度的单元（或称为段（segment））。

Self-Relative Addressing（自相对寻址）：一种寻址模式，需要从当前指令中计算出操作数的地址作为一个偏移量。

Semantic Gap（语义图）：一种存在于计算机物理部件和高层语言之间的逻辑图。

Sequential Circuit（时序电路）：一种逻辑电路部件，时序电路的输出不仅与当前的输入有关，而且还与以前的输入有关。

Serial Communication（串行通信）：一种传输数据的方法，每次只传输数据字节的一位。串行通信是异步通信。在传输介质中，串行通信并不要求单独定时信号。比较并行传输。

Serial Storage Architecture（串行存储器体系结构）：参见SSA。

Serpentine Recording（蛇形（曲线）记录法）：一种将数据位放置到磁带介质上的“长度智能(length-wise)”方法。每个字节都要沿着磁带的长边平行对齐。目前流行的蛇形线磁带的格式包括数字线性磁带、DLT和1/4英寸卡式磁带（QIC）。

Server Consolidation（服务器整合）：将多个服务器（通常是小型服务器）集成为一个服务器（通常是大型服务器）的过程。

Server Farm（服务器站）：一个由大量小型服务器组成的大型计算机设施的管理环境。

Service Access Point（SAP）（服务访问点）：在通信会话过程中用来识别请求的协议服务的一个数值。在TCP协议中，SAP是一个被称为端口的数值。

Set Associative Cache（组关联高速缓存）：也称为N路的组关联存储器（n-way Set Associative Cache）。一种高速缓存的映射方案，把高速缓存划分为一些关联存储器的空间块组，主存储器会按照模块的方式映射到一个给定的高速缓存组。

Set Field（组域）：地址中用来指定相应的高速缓存组的部分。

Shannon's Law（香农定律）：1948 年，香农证明了，一个非理想的传输介质的单传输容量，可以用公式表示为：最大数据传输率＝带宽×$\log_2$［1＋（信号 dB÷噪音 dB）］波特。

Shared Memory Systems（共享存储器系统）：一种存储器系统。所有的处理器都可以访问一个全局存储器，通过一些共享变量进行通信。

Shortest Seek Time First（SSTF）（最短寻道时间优先）：一种磁盘调度算法，通过算法安排访问请求，首先服务离磁盘臂的当前位置最近的请求磁道。

Shuffle Network（洗牌网络）：参见 Multistage Interconnection Network。

Signal-to-Noise Ratio（信噪比）：一种通信信道质量的度量方法。信噪比与线路上载波信号的频率成正比（频率越高，信噪比就越大）。从数学上，信噪比（dB）＝$10\log_{10}$（信号 dB÷噪声 dB）。

Signed Magnitude（符号幅值表示法）：一种二进制数的表示方法，最左边的位为符号位，其余位由数的绝对值组成（或称为幅值）。

Significand：参见 Mantissa。

SIMD，Single Instruction，Multiple Data（单指令多数据）：一种计算机体系结构，它具有单点控制，同时对多个数据值来执行同一条指令。例如，矢量处理器和阵列处理器。

SISD，Single Instruction，Single Data（单指令单数据）：一种计算机体系结构。这种体系结构只有一个单一指令流和一个唯一的数据流。包括现在应用在大多数 PC 机中的冯·诺伊曼体系结构。

SLED，Single Large Expensive Disk（大型昂贵的单磁盘）：随着 RAID 概念而出现的一个术语。SLED 系统相对于 RAID 系统来说，可靠性较低，性能也比较差。

Small Computer System Interface（小型计算机系统接口）：参见 SCSI。

SMP，Symmetric Multiprocessors（对称多处理器）：一种 MIMD 的共享存储器体系结构。

SNA：参见 Systems Network Architecture。

SONET（同步光纤网络）：一种为 T 载波系统构建的光纤传输标准。

SPEC（标准性能评估公司）：SPEC 目的是为计算机性能测试建立公正的和现实的方法。SPEC 负责产生相应的基准套件，内容包括文件服务器、Web、桌面计算环境、企业级的多处理器系统、超型计算机、多媒体和图形增强系统。

Speculation Execution（推测执行）：在确定指令是否需要执行之前，先取出指令并在流水线中开始执行的一种行为。如果发现预测不正确，则必须要取消预测执行。

Speedup（加速）：参见 Amdahl's Law。

SPMD，Single Program，Multiple Data（单程序多数据）：一种弗林分类法的扩展，这种体系结构由多个处理器组成，每个处理器有自己的数据组和程序存储器。

Spooling（假脱机，外围设备联机并发操作）：指一个程序在被送往打印机之前，先将要打印输出的内容写入到磁盘中，这样有利于弥补 CPU 速度和打印机速度之间的巨大差异。

SRAM，Static RAM（静态随机存储器）：只要电源不会断电，RAM 就可以保持存储器中的内容。这与动态 RAM 不同，动态 RAM 要求不断刷新才能保持数据。

SSA，Serial Storage Architecture（串行存储器体系结构）：SSA 的设计采用一个冗余循环的配置支持多个磁盘驱动器和多个主机。由于这种冗余性，即使一个驱动器或主机适配器失效，剩余的磁盘还可以进行访问。SSA 的双循环拓扑结构还允许基本吞吐量加倍，从 40MB/s 增加到 80MB/s。由于光纤信道的使用，SSA 正在逐渐失去它的市场份额。

SSI，Small Scale Integration（小规模集成电路）：每个芯片上有 10 到 100 个元件的集成电路。

SSTF：参见 Shortest Seek Time First。

Stack（堆栈）：一种简单的为后进先出（LIFO）数据元素处理优化设计的数据结构。堆栈结构按照元素进入堆栈顺序的相反次序从堆栈中移走数据元素。

Stack Addressing（堆栈寻址）：一种寻址模式，假定操作数在系统的堆栈中。

Stack Architecture（堆栈体系结构）：一种使用堆栈执行指令的计算机体系结构，这里总是隐式地假定操作数就在（堆）栈顶。

Standard Performance Evaluation Corporation（标准性能评估公司）：参见 SPEC。

Star-Connected Network（星型网络）：一种所有的消息都必须通过一个中心集线器的网络。

Static Interconnection Network（静态互连网）：一种在两个实体之间建立固定路径的网络。这两个连接实体可能是两个处理器，也可能是一个处理器和一个存储器，这两个实体之间的路径从一个通信到下一个通信不能发生变化。

Statistical Coding（统计编码）：一种数据压缩方法。在统计编码中，符号出现的频率决定了输出符号的长度。符号出现的几率会连同译码消息所要求的信息一起被写入到文件中。输入中频繁出现的最大符号在输出中会变成最小符号。

Status Register（状态寄存器）：一种专用的寄存器，用来监视和记录一些特殊的条件，例如溢出、进位和借位等。

STM-1：基本的 SDH 信号，能够以 155.52Mb/s 的速率传输信号。STM 是由欧洲同步数字体系传送的信号。类似于北美的 SONET 光载波系统。

Storage Area Network（存储器区域网络）：为数据存储访问和管理而专门构建的网络。

STS-1：参见 SONET。

Subnet（子网）：大型网络被细分为一系列的子网。在 TCP/IP 协议下，子网是一种网络中的所有设备的 IP 地址都具有相同的前缀的网络。

Subsystem（子系统）：一种逻辑计算环境通，建立子系统是为了方便相关应用进程的管理。

Subtrahend（减数）：参见 Minuend。

Sum-of-Products Form（积之和形式）：一种布尔表达式的标准形式，它是一组乘积项的或（OR）运算一个集合。

Superpipelining（超流水线）：一种组合超标量和流水线概念的计算机技术。通过把流水线的步骤分成许多更小的步骤，使得计算机可以在一个时钟周期内执行多个步骤。

Superscalar（超标量）：一种计算机体系结构的设计方法。在超标量结构中，CPU 包含多个 ALU，而且每个时钟周期能够发出多条指令。

Supervised Learning（监督学习算法）：一种在训练神经网络中使用的学习方法，这种学习方法是将一些正确的知识结果输入神经网络，对神经网络进行训练。

Switch（交换机）：在系统部件之间提供点对点的互连的一种设备。在数据通信中，交换机是第 2 层的设备，它可以在其输入端口的某一点和输出端口的某一点之间创建一个点到点的连接。

Switching Network（开关网络）：一种通过开关（既可以是交换开关，也可以是 2×2 开关）来连接处理器和存储器的网络，这个开关网络允许动态路由。

Symbol Table（符号表）：由汇编程序创建的一个表，用来存放各个表之间和存储器地址之间的一一对应关系。

Symbolic Logic（符号逻辑）：参见 Boolean algebra。

Synchronous Circuits（同步电路）：一种时序电路，使用时钟脉冲对事件进行排序。这些电路的输出值只能随着时钟脉冲的跳动而发生改变。

Synchronous Digital Hierarchy，SDH（同步数字体系）：欧洲使用的与 SONET 对等的体系，SDH 使用的 256 位帧。基本的 SDH 信号是 STM-1，可以 155.52Mb/s 的速率传输信号。

Synchronous Optical Network（同步光纤网络）：参见 SONET。

Synchronous Transport System 1（同步传输系统 1）：参见 SONET。

Syndrome（出错位组）：一组错误校验位。

Synthetic Benchmark（综合基准程序）：从一个针对特定的系统部件编写的程序得到的性能度量标

准。通过在不同的系统上运行综合基准程序，程序的执行时间结果会对所有测试的系统产生一个单一性能度量标准。比较好的综合基准程序有：Whetstone、Linpack 和 Dhrystone 度量标准。

System Bus（系统总线）：通常在 PC 机中连接 CPU、存储器和其他所有内部组件的内部总线。

System Simulation（系统仿真）：预测系统各种行为的软件模型，利用仿真器建模无需使用真正的系统环境。仿真对于评估一些目前还不存在的系统或系统配置的性能是非常有用的。

Systematic Error Detection（系统错误检测）：一种错误检测方法，把一些错误校验位加到原来的信息字节的后面。

System Network Architecture，SNA（系统网络体系结构）：一种由 IBM 公司发明的早期专用网络技术。这个系统最初是由一些哑终端组成的，利用一个通信控制器对终端设备进行轮流测试（轮询）。接着，这个通信控制器会与一个连接到主机计算机的前端处理器进行通信。

Systolic Array（脉动阵列）：SIMD 计算机的一种变化形式。脉动阵列采用一种简单处理器的大型阵列，对数据流使用矢量流水线作业，而且具有高度的并行性。

T Carrier System（T 载波系统）：一种数字载波系统，采用时分复用技术来产生不同的传输速率。目前，只有 T-1（1.544Mb/s）和 T-3（6.312Mb/s）被广泛使用。许多装置都被移植到光纤载波上。

Tag Field（标记域）：地址中用来指定高速缓存标记的部分。

Tape Silo（磁带库）：参见 Robotic Tape Library。

TCM：参见 Trellis Code Modulation。

TCO：参见 Total Cost of Ownership。

TCP：网络中使用的一种面向连接协议。TCP 是一个自我管理的协议，能够确保准确顺序的数据段传输。

TDM：参见 Time Division Multiplexing。

Thrashing（摆动）：当所需的数据块或页被频繁地放入存储器中但又立即被替换时，就发生了摆动现象。

Throughput（吞吐量）：在不会造成响应时间有重大降低的情况下，一个系统可以执行并发任务的数目的一种度量。

Thunking（形实转换）：在 Microsoft Windows 系统中，把 16 位指令转换为 32 位指令的一个过程。

Time Division Multiplexing，TDM（时分复用）：一种多路技术方法，将单个通信介质或载波划分成信道，分配给分立不相关的连接。每个连接会被指定一个小的、重复的传输时间段。

Timesharing（分时共享）：一种多用户计算机系统。CPU 在用户进程之间切换速度非常快，为每个用户分配一个小段的时间片（处理器的小部分时间）。

Timeslice（时间片）：参见 Timesharing。

Total Cost of Ownership（总体拥有成本）：一个计算机系统在一个给定的时间周期内，所需的总费用，这个时间段也许会超过系统的预期寿命。一个有用的 TCO 数字至少要包括以下的内容：购买特定的系统配置的费用，其中包括系统软件费用，预期的系统扩展和升级的费用，还有硬件和软件的维护费用，操作人员的人工，以及设施的费用，其中包括房屋租用、电源消耗和空调系统的费用。

TPC（事务处理委员会）：TPC 委员会专门负责为支持事务处理，Web 商务和数据仓库（决定支持系统）的服务器创建各种基准套件。

Transaction Processing Council（事务处理委员会）：参见 TPC。

Transfer Time（转移时间）：访问时间与从硬盘上实际读取数据所花费的时间之和。转移时间会随着读取数据方式不同而发生变化。

Transistor（晶体管）：真空三极管的一种固体形式，用来放大信号，接通电路或关闭电路。在第二代计算机中使用晶体管技术。

Translation Look-aside Buffer，TLB（转换旁视缓冲器）：也称为快表，一个页表的高速缓存。它的

入口条目由一些数据对组成，即由虚拟页数和物理帧数组成。

Transmission Control Protocal（传输控制协议）：参见 TCP。

Transparet Bridge（透明网桥）：一种复杂的网络设备，有能力知道每个网络段上各个设备的地址。透明网桥也能够提供管理信息，例如汇报吞吐量。

Transport Latency（传输延迟）：消息在网络中所花费的时间。

Trellis Code Modulation，TCM（格码调制）：一种信号调制方法，同时改变载波信号的相位和振幅来编码数字信息。TCM 非常类似于 QAM，但是有一点不同的是，TCM 对每个信号点都增加一个奇偶校验位，允许在传输的信号中进行某些前向误差校正。

Trie（树形数据结构）：一种非循环的 n 叉树的数据结构，将部分数据关键字的值存放在每一个节点。关键字的值被组装会继续沿着 Trie 树搜索。内部节点包含足够多的指针数来指引搜索所要求的关键字或者导向树的下一层。

Truth Table（真值表）：一个描述逻辑函数的表，以表格的形式说明了所有可能的输入值和函数输出结果之间的关系。

Twisted Pair（双绞线）：一对绝缘的导线相互扭在一起，来传输电信号。

Two's Complement Notation（补码表示法）：又称为 2 的补码表示法，是一种用来表示带符号的二进制数值的方法。正数简单地用符号幅值表示法来表示，对于负数，则将相应的正数表示中的所有的位翻位（按位取反）后再加 1。

ULSI，Ultra Large Scale Integration（超大规模集成电路）：每个芯片上有多于 100 万个元件的集成电路。

Underflow（下溢）：参见 Divide Underflow。

Unguided Transmission Media（非导向性传输介质）：能够传输数据通信信号的电磁波或光波等载波介质。这种媒体类型包括无线广播、微波、卫星通信和空间光通信。

Unicode：一种 16 位的国际字符编码，能够表示世界上每一种语言。统一字符集中低的 127 位字符和 ASCII 字符集的字符是完全相同的。

Uniform Memory Access，UMA（均匀存储器访问计算机）：一种共享存储器的 MIMD 计算机。这种计算机中的任何一个处理器访问任何一个存储器需要相同的时间。

Universal Gate（通用门电路）：之所以称为通用门电路，是因为任何数字电路都可以仅使用一种门电路构建而成。与非门（NAND）和或非门（NOR）都是通用门电路的例子。

Universal Serial Bus（通用串行总线）：参见 USB。

Unsupervised Learning（无监督学习算法）：在训练神经网络中所使用的一种学习类型。这种学习算法在训练期间不向神经网络提供正确的输出。

USB，Universal Serial Bus（通用串行总线）：一种用于 USB 接口的外部总线标准，支持各种热插拔的设备。

Vacuum Tube（真空管）：一种用在第 0 代计算机的不太可依靠的技术。

Vector Processors（矢量处理器）：一种专用的高度流水线作业的处理器，可以对整个矢量和矩阵立即执行非常有效的操作。寄存器-寄存器类型的矢量处理器要求所有的操作都要使用寄存器作为源操作数和目标操作数。而存储器-存储器类型的矢量处理器允许来自存储器的操作数被直接发送到算术单元中。

Virtual Address（虚拟地址）：为了响应程序的执行，由 CPU 生成的一种逻辑地址或程序地址。无论 CPU 何时产生地址，总是位于虚拟地址空间。

Virtual Machine（虚拟机）：（1）一种假想的计算机；（2）一种自包含的操作环境。这种环境会给人一种存在于一个独立的物理机器幻想；（3）一台实际机器的一种软件仿真。

Virtual Memory（虚拟存储器）：一种利用硬盘扩展 RAM 的方法，这样可以增加程序进程的可用地

址空间。

VLIW（Very Long Instruction Word）Architecture（超长指令字体系结构）：一种体系结构的特性。在超长指令字结构中，每个指令都能够规定多个标量操作。

VLSI，Very Large Scale Integration（超大规模集成电路）：每个芯片上有多于 10000 个部件的集成电路。这种技术在第四代计算机中使用。

Volatile Memory（非永久性（易失性）存储器）：在电源断电后，存储器中的内容都会丢失的一种存储器类型。

Von Neumann Architecture（冯·诺伊曼体系结构）：一种存储程序的计算机体系结构，由 CPU、ALU、寄存器和主存储器组成。

Von Neumann Bottleneck（冯·诺伊曼瓶颈）：指在主存储器和 CPU 控制单元之间的只有单一通道。这条单一通道会迫使指令周期和执行周期交替进行，这样常常会产生瓶颈现象，或导致系统的速度减慢。

Wall Clock Time（挂钟时间）：也称为逝去的时间，是一种度量计算机性能的方法，特别是在系统正在运行程序时，测量墙上的挂钟时间。

WAN（广域网）：WAN 能够覆盖多个城市或整个世界。

Weighted Arithmetic Mean（加权算术平均数）：一种算术平均数，通过结果值乘以结果值出现的频率（按照百分比的形式表示）求得。加权算术平均数给出一个系统性能的数学期望值。加权算术平均数也可以称为加权平均。

Weighted Numbering System（加权计数系统）：一种通过增加基数的幂来表示一个数值的计数系统。二进制和十进制计数系统都是加权计数系统的例子。而罗马数字就不是这种类型。

Whetstone：一种基准程序，是一种浮点增强型的程序。为了计算三角函数和指数函数，程序中有许多的库函数调用过程。测试结果的报告表示为每秒钟千条 Whetstone 指令（KWIPS），或者表示为每秒钟兆条 Whetstone 指令（MWIPS）。

Wide Area Network（广域网）：参见 WAN。

Winchester Disk（Winchester 硬盘）：简称硬盘，是一种完全封闭式的硬盘。这个名称来源于 IBM 公司开发这种技术时所采用的编码名字（Winchester 编码）。

Word Field（Offset Field）（字域（或称为偏移量域））：在一个给定的数据块或页面中指定唯一字的地址部分。

Word（字）：一组可编址的连续位。通常情况下，字由 16 位、32 位或者 64 位组成。但是，一些早期的计算机体系及结构使用字的大小并不是 8 的整数倍。

Word-addressable（按字编址）：每个字（不必是每个字节）都有自己的地址。

Write-Back（回写）：一种高速缓存更新策略。只有当一个高速缓存块被选中要从高速缓存中被移走时，才更新主存储器。允许存储的高速缓存值与它相应的存储器的值不一致。

Write-Through（写通）：一种高速缓存更新策略。每一次写操作，会同时更新高速缓存和主存储器。

Zoned-Bit Recording（区位记录法）：一种增加磁盘容量的实践方法。具体做法是，将磁盘的所有扇区都划分为大小基本相同的单元，把更多的扇区放置在外层的磁道上而不是放置在内层的磁道上。而其他类型的记录方法是在磁盘的所有磁道有同样数目的扇区。

部分练习题答案和提示

第 1 章

1. 在硬件和软件之间，一个用来提供更快的速度，而另一个用来提供更大的灵活性。这里，哪一个是硬件？哪一个是软件？硬件和软件通过软硬件的等效性原理相互关联。是否有一个能够解决某个问题而另一个却不能解决的地方？

3. 100 万，或者 10^6。

7. 0.75 微米。

第 2 章

1. a) 121222_3
   b) 10202_5
   c) 4266_7
   d) 6030_9

3. a) 11010.11001
   b) 11000010.00001
   c) 100101010.110011
   d) 10000.000111

5. a) 符号-幅值：01001101
      反码：01001101
      补码：01001101
   b) 符号-幅值：10101010
      反码：11010101
      补码：11010110

9. a) 最大值正数：011111 (31)
      最小值负数：100000 (−31)

11. a) 111100

13. a) 1001

15. 104

17. 提示：按照如下方式开始跟踪程序执行过程：

J	(二进制)	K	(二进制)
0	0000	−3	1100
1	0001	−4	1011 (1100+1110)(在运行反码加法时把上一个进位添加到求和中)
2	0010	−5	1010 (1011+1110)
3	0011	−6	1001 (1010+1110)
4	0100	−7	1000 (1001+1110)
5	0101	7	0111 (1000+1110)(溢出，但可以忽略)

19.

0	1	1	1	1	0	1	0

误差=2.4%

25.

二进制值	00000000	00000001	00100111
ASCII 编码	10110010	00111001	00110101
压缩 BCD 编码	00000000	00101001	01011100

27. 提示：考虑语言和存储空间的兼容性

32. 4

35. 第 5 位发生错误

39. a) 1101 余数 110
    b) 111 余数 1100
    c) 100111 余数 110
    d) 11001 余数 1000

41. 编码字：1011001011

第 3 章

1. a)

x	y	z	xyz	$(\overline{xyz})$	$xyz+(\overline{xyz})$
0	0	0	0	1	1
0	0	1	0	1	1
0	1	0	0	1	1
0	1	1	0	1	1
1	0	0	0	1	1
1	0	1	0	1	1
1	1	0	0	1	1
1	1	1	1	0	1

b)

x	y	z	$y\bar{z}$	$\bar{x}y$	$(y\bar{z}+\bar{x}y)$	$x\ (y\bar{z}+x\bar{y})$
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	1	1	1	0
0	1	1	0	1	1	0
1	0	0	0	0	0	0
1	0	1	0	0	0	0
1	1	0	1	0	1	1
1	1	1	0	0	0	0

3. $F(x,y,z) = x(\bar{y}+z)$

$$\begin{aligned}\bar{F}(x,y,z) &= \overline{x(\bar{y}+z)}\\ &= \bar{x}+\overline{(\bar{y}+z)}\\ &= \bar{x}+y\bar{z}\end{aligned}$$

5. $F(w,x,y,z) = xy\bar{z}((\overline{\bar{y}z+x})+(\overline{\bar{w}yz+\bar{x}}))$

$$\begin{aligned}\bar{F}(w,x,y,z) &= \overline{xy\bar{z}((\overline{\bar{y}z+x})+(\overline{\bar{w}yz+\bar{x}}))}\\ &= \overline{xy\bar{z}}+((\overline{\overline{\bar{y}z+x}})(\overline{\bar{w}yz+\bar{x}}))\\ &= \overline{xy\bar{z}}+((\bar{y}z+x)(\overline{\bar{w}yz+\bar{x}}))\\ &= \bar{x}+\bar{y}+z+((\bar{y}z+x)(\overline{\bar{w}yz}\,\bar{x}))\\ &= \bar{x}+\bar{y}+z+((\bar{y}z+x)((w+\bar{y}+\bar{z})x))\end{aligned}$$

7. 错。一种方法是使用真值表，而另外一种更具挑战性的方法是利用下面的关系通过恒等式来证明。

$$a\text{XOR}b = a\bar{b} + \bar{a}b$$

9. b) $(x+y)(x+\bar{y})(\bar{x}+z)$

$= x(x+\bar{y})(\bar{x}+z) + y(x+\bar{y})(\bar{x}+z)$

$= x(x\bar{x} + xz + x\bar{y} + \bar{y}z) + y(x\bar{x} + xz + x\bar{y} + \bar{y}z)$

$= x(xz + x\bar{y} + \bar{y}z) + y(xz + x\bar{y} + \bar{y}z)$

$= xxz + x\bar{y}z + x\bar{y}z + xyz + x\bar{y}y + y\bar{y}z$

$= xz + x\bar{y}z + xyz$

$= xz(1 + \bar{y} + y) = xz$

11. a) $\bar{x}yz + xz = \bar{x}yz + xz(1)$ 一致律

$= \bar{x}yz + xz(y + \bar{y})$ 互补律

$= \bar{x}yz + xyz + x\bar{y}z$ 分配律和交换律

$= \bar{x}yz + (xyz + xyz) + x\bar{y}z$ 幂等律

$= (\bar{x}yz + xyz) + (xyz + x\bar{y}z)$ 结合律

$= (\bar{x} + x)yz + (y + \bar{y})xz$ 分配律(二次应用)

$= 1(yz) + (1)xz$ 互补律

$= yz + xz$ 一致律

b) $\overline{(x+y)}\,\overline{(\bar{x}+\bar{y})} = (\bar{x}\bar{y})(\bar{\bar{x}}\,\bar{\bar{y}})$ 德摩根律

$= (\bar{x}\bar{y})(xy)$ 双补律

$= (\bar{x}x)(\bar{y}y)$ 交换律和结合律

$= (0)(0)$ 互补律(2 次)

$= 0$ 幂等律

c) $\overline{((\bar{x})\,\overline{(\bar{x})}y)} = \overline{((\bar{x})(x)y)}$ 双补律

$= (\overline{0y})$ 互补律

$= (\bar{0})$ 零律

$= 1$ 补码的定义

13. a) $xy + x\bar{y}$

$= x(y + \bar{y})$ 分配律

$= x(1)$ 互补律

$= x$ 一致律

15. $x(\bar{x} + y)$

$= x\bar{x} + xy$

$= 0 + xy = xy$

17. $xy + \bar{x}z + yz$

$= xy + \bar{x}z + (1)yz$

$= xy + \bar{x}z + (x + \bar{x})yz$

$= xy + \bar{x}z + xyz + \bar{x}yz$

$= (xy + xyz) + (\bar{x}z + \bar{x}yz)$

$= xy(1 + z) + \bar{x}z(1 + y)$

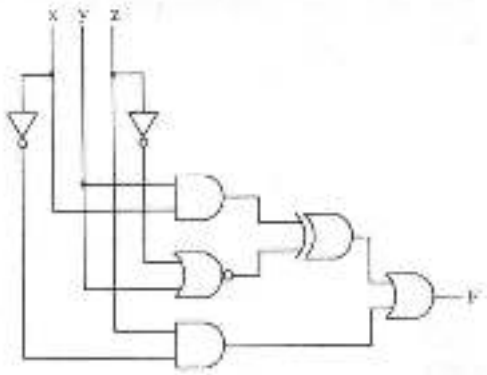
$= xy + \bar{x}z$

18. $\bar{x}\bar{y}z + \bar{x}y\bar{z} + x\bar{y}z + xy\bar{z}$

21. (没有简化的)

a) $\overline{\bar{x}y + xy\bar{z}} = x\bar{y} + xz + \bar{x}\bar{y} + \bar{y} + \bar{y}z$

27.



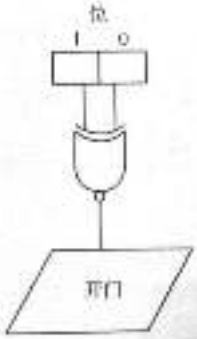
28.

x	y	z	F
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

35. 赋予输入的值（卡的编码方法）决定了每个读卡器的设计。一种编码方法如下表所示。

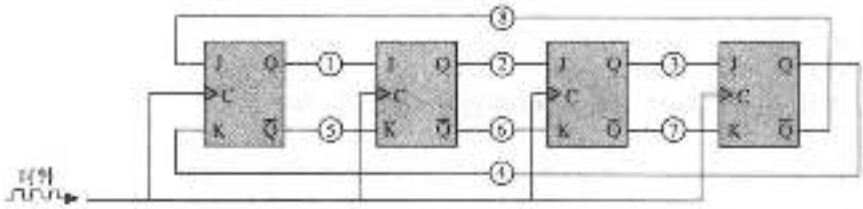
编　码	雇员等级	授权进入的区间				
		储藏室	服务器机房	服务室	处理室	环境卫生间
00	IT 员工		X	X		
01	秘书	X		X	X	
10	大楼板				X	X
11	门卫	X	X	X	X	X

基于上面的编码，服务器机房的读卡器可以按如下的方式实现：



这个设计的其他部分？

37. 从如下图所示对触发器之间的连线进行编号开始：



对于 $i=0$ 到 8，完成下面的表：

i	断开的线路	接通的线路
0	1，2，3，4	5，6，7，8
1	????	????
…	…	…
8	????	????

39.

			下一状态	
A	B	X	A	B
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	0
1	0	0	1	1
1	0	1	0	1
1	1	0	1	0
1	1	1	0	0

41.

			下一状态	
A	B	X	A	B
0	0	0	1	0
0	0	1	0	0
0	1	0	1	0
0	1	1	0	0
1	0	0	1	1
1	0	1	1	1
1	1	0	1	0
1	1	1	1	0

第 3 章专题

3A.1. a) $xz=xz$

b) $xz+xy+xyz$

3A.5. $xyz+xyz+xyz+xyz$

$xz(y+y)+xz(y+y)$

$\bar{x}z+xz$

$z(\bar{x}+x)=z$

3A. 6. a) $x+\bar{y}$

(没有包括“无关（don't care)”条件，因为对问题并没有多大帮助。)

第 4 章

3. a) 总字节为：2M×4 字节$=2\times2^{20}\times2^{2}=2^{23}$所以，一个地址需要 23 位。

b) 2M 字$=2\times2^{20}=2^{21}$，所以，一个地址需要 21 位。

6. a) 16（2 列 8 行）

b) 2

c) 256K$=2^{18}$，所以 18 位

d) 8

e) 2M$=2^{21}$，所以 21 位

f) 存储器组 0（000）

g) 存储器组 6（110）

9. a) 2^{20}字节，需要 20 位地址，编址范围：地址 0 到 $2^{20}-1$。

b) 只有 2^{19}字，编址范围：地址 0 到 $2^{19}-1$

12. 提示：考虑数据和地址之间的差别。

14.

A	108
One	109
S1	106
S2	103

15. a. i) 存储 007

18. 提示：把 FOR 循环变成 WHILE 循环。

22. 提示：递归暗指需要许多子程序的调用。

第 5 章

1. a)

地址⟶	00	01	10	11
大端位序	00	00	12	34
小端位序	34	12	00	00

3. a) $FE01_{16}=1111\ \ 1110\ \ 0000\ \ 0001_{2}=-511_{10}$

b) $01FE_{16}=0000\ \ 0001\ \ 1111\ \ 1110_{2}=510_{10}$

5. 提示：它们在不同的处理器上运行。

7. 6×2^{24}

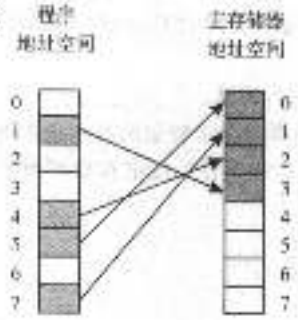
8. a) XY×WZ×VU×++

13.

寻址方式	值
立即寻址	1000
直接寻址	1400
间接寻址	1300
变址寻址	1000

18. a) 8
    b) 16
    c) 2^{16}
    d) $2^{24}-1$

第 6 章

1. a) $2^{20}/2^{4}-2^{16}$
   b) 20 位地址中，有 11 位是标记域，5 位是块域以及 4 位是字域。
   c) 32 块
3. a) $2^{16}/2^{5}-2^{11}$
   b) 15 位地址中，有 11 位是标记域，5 位是字域。
   c) 因为是关联高速缓存，所以可以映射到任何地方。
5. 每个地址有 27 位，其中 7 位是标记域，14 位是组域，以及 6 位是字域。
13. 

第 7 章

1. 提示：考虑如何应用 Amdahl 定律。
3. a) 选择磁盘升级。对于磁盘，性能每提升 1%需要的费用是 216.2 美元，而对于 CPU 需要的费用是 268.24 美元。
   b) 磁盘升级产生较大的性能改进，磁盘为 36.99%，而处理器为 18.64%。
   c) 收支平衡点是采用 9922 美元的费用进行磁盘升级。
5. a) 在进入某个中断服务程序之前，CPU 应该禁止所有中断，所以这个中断根本就不应该发生。
   b) 不是问题。
   c) 如果所有的中断都被禁止，那么第二个中断决不会发生，所以这不是问题。
11. 有的人认为从某个特殊的磁盘上取回指定的数据不是“随机”的行为。
13. 旋转时延（平均反应时间）＝4464RPM＝74.4rev/sec＝0.01344sec/rev＝13.44ms/rev。（或者是，60 000ms/min/4464rev/min＝13.44ms/rev）。平均值是整个数值的一半，或者是 6.72ms。
15. 提示：考虑磁盘缓冲器的影响。
17. a) 256MB（1MB＝2^{20}B）
    b) 11ms
19. 提示：要考虑取回时间和存档文件要求的存放时间。

第 7 章专题

5. 主机适配器通常具有最高的设备号，因此主机适配器总是能够赢得总线的仲裁。“快速和宽带”的 SCSI-3 接口可以支持多达 3 个设备，所以，主机适配器的设备号总是为 32。
11. 提示：除了考虑在 CPU 和存储器之间传输字节数据外，还要考虑总线的其他用途。

第 8 章

3. 提示：考虑造成不可预测的反应时间的一些事情（例如，如何时延一个存储器的访问?）。
5. 如果一些进程共享一组特定的资源，那么合理的做法是将这些进程组合成一个子系统。如果为了测试这个系统使用一组给定的进程，则明智的做法是把这些进程也组合成一个子系统，就好像这些进程崩溃或反常，只有运行这些进程的子系统才会受到影响。如果要在有限的时间，或用有限的资源访问一组特定的人群，则需把这些用户进程也组合成一个子系统。
7. 当需要对代码进行更多的压缩时，常常使用不可重新定位的代码。因此，在嵌入式系统中，由于空间的限制（例如，微波或车用电脑），这种情况非常普遍。不可重新定位的代码的运行速度较快，所以它使用在那些对很小的时序时延都非常敏感的系统（例如实时系统）中。可重新定位的代码需要有硬件的支持，因此，不可重新定位的代码适用于没有这种硬件支持的情况（例如，任天堂游戏机）。
9. 为什么动态链接节省磁盘空间，产生较少的系统错误以及允许代码共享？然而，动态链接会造成装载时间的时延，而且如果动态链接库程序被改变，那么其他使用被修改库的程序将无法完成库的链接。
19. 首先将 Java 编译成字节代码，然后利用 JVM 对这些中间字节代码进行解释。
21. a) 如果出现不同的计算结果（例如，输出、数据变量的值），这些结果取决于特定的时序或是跨越一些独立的线程、进程或事务语句的执行结果的顺序，则会出现竞争条件。假定有如下两个事务，访问一个初始结余 500 的账户：

事务 A	事务 B
获取账户余额	获取账户余额
在余额中加上 100	从余额中减去 100
存储新的余额	存储新的余额

    很显然，新的结余值取决于两个事务运行的顺序。对于新的账户余额，可能的值会是多少？

    b) 通过孤立地运行各个事务处理和提供原子性，可以防止竞争条件的出现。在数据库中，通过加锁确保原子事务处理。

    c) 采用加锁可能会导致死锁。假定事务 T1 在数据项目 X 上获得一个专用锁（这意味着其他事务不可以共享该锁），而事务 T2 则在项目 Y 上获得一个专用锁。假定 T1 需要占用 X 但现在却需要 Y，而 T2 必须占用 Y 但现在却需要 X，则这时两个事务都会一直等待对方，而且都不会释放它们自己拥有的锁，因此发生了死锁。

第 9 章

1. RISC 机器限制只有装载和存储指令可以访问存储器。这就意味着所有的其他指令都要使用寄存器。这样要求较少的指令周期，可以加快代码的执行速度，因而提高硬件的性能。RISC 体系结构的目标是实现单周期指令，而当指令必须访问存储器而不是寄存器时，这是不可能实现的。
3. “精简”的原意是提供一组最少的、可以执行所有基本操作的指令：数据移动、ALU 操作和分支处理。然而，RISC 体系结构现在最主要的目标是简化指令，便于加快指令执行的速度。每条指令只完成一种操作，所有指令的长度都相同，只是指令布局结构有所不同，而且所有的算术运算都必须使用寄存器来执行（即存储器中的数据不能用作为操作数）。
5. 128
9. 在关联转换过程中，当前正在执行的进程的所有信息必须被保存，其中包括寄存器窗口中的值。当保存该进程时，也必须保存该进程的寄存器窗口的值。进程信息的保存取决于寄存器窗口的大小，保存存储寄存器窗口的值可能是一个非常费时的进程。

11. a) SIMD：单指令、多数据。一个特定的指令执行多个数据。例如，一个矢量处理器会使用下面指令来进行数组相加：C [i] =A [i] +B [i]。而这条指令可以对几个数据执行：C [1] =A [1] +B [1]，C [2] =A [2] +B [2]，C [3] =A [3] +B [3] 等等。执行数据的数目取决于处理器中包含多少个 ALU。
13. 松散耦合和紧密耦合是两个用来描述多处理器如何处理存储器的术语。如果有一个大的、集中式的和共享的存储器，则称这个系统是紧密耦合的。如果有多个分布在不同物理位置的存储器，则认为这个系统是松散耦合的。
17. SIMD：数据并行执行；MIMD：控制或任务并行执行。为什么？
19. 超标量处理器既取决于硬件（仲裁相关性）又取决于编译器（产生近似的调度安排），VLIW 处理器则完全取决于编译器。因此，VLIW 将任务的复杂性完全移给编译器。
20. 这两种体系结构都配备少量数目的并行流水线来处理指令。然而，VLIW 体系结构依靠编译器以一种正确和有效的方式来预包装和调度指令。在超标量体系结构中，指令的调度是由硬件来完成。
21. 分布式系统允许共享并存在冗余。
23. 如果在一个交叉（纵横）互联网络中增加处理器，则交叉开关的数目会很快变得难以管理。总线网络就会出现潜在的瓶颈和争用的问题。
25. 使用写通时，新值会立即被写到服务器。这样总是可以保持服务器处于最新状态，但是写操作会占用较长时间（这样会损失高速缓存提升的处理速度）。而使用回写时，这个新值会经过一个给定的时延后才流向服务器。这样保持了服务器速度的提升，但是这也意味着如果在新值存放到服务器之前服务器发生了崩溃，数据就会丢失。这是一个很好的例子，它说明性能增加必须要付出一定的代价。
27. 是的。单个的神经元会接收输入、处理输入和产生输出。然后，另外一个神经元会使用这个输出，按照上面的步骤处理。但是，这些神经元本身是以并行的方式作业的。
29. 当神经网络正在学习时，错误的输出指示用于对网络中的权值进行调整。这些调整基于不同的优化算法。当这种权重平均汇聚到某个给定的数值时，该学习过程完成。

第 10 章

1. 系统 C 执行时间的加权平均值为：2170/5=434，因此，得出：（563.5/434=1.298387-1）×100=30%。系统 A 的性能已经下降：（9563.5/79-1）×100=641.4%。
3. 系统 A：算术平均值=400；几何平均值：1、0.7712 和 1.1364。
   系统 B：算术平均值=525；几何平均值：1.1596、1 和 1.3663。
   系统 C：算术平均值=405；几何平均值：0.7964、0.6330 和 1。
5. 提示：使用调和平均数。
7. 这是不完整信息的谬误。为什么？
9. 没有这种比较数值。因为每个版本的基准都是由不同的程序组成，所以得到的结果不能够进行比较。
11. 在这种情况下，最好的基准是 SPEC CPU 系列。为什么？
13. 首先，这是一种外交辞令。建议这个小组研究 TPC-C 基准是否适用于这个系统。无论 TPC-C 的数字是否适用，都需要向人们阐述 Amdahl 定律的意义，以及为什么一个快速的 CPU 不一定能够决定整个系统是否可以处理我们所要求的工作量。
18. 提示：旋转延迟不是决定磁盘性能的唯一因素。
24. a) 5.8 周期/指令
    b) 5.8MHz（为什么?）
    c) 13.25

第 11 章

5. TCP 报文段的有效载荷（数据）应该尽可能的大，这样可以将发送这个报文段所需的网络开销减到最小。如果有效载荷仅由 1 或 2 个字节组成，则这种网络开销比数据传输至少要大一个数量级。

7. 提示：回顾 TCP 报文段格式的数据偏移量域的定义。

9. a) B 类
   b) C 类
   c) A 类

11. a) 我们可以假定在 TCP 或 IP 报头中没有设置选项，而且忽略任何会话停播消息。在 TCP 报头中有 20 个字节，并且在 IP 报头中也有 20 个字节。对于一个 1024 字节的文件，如果使用 128 字节的有效载荷，则需要传送 8 个有效载荷。因此，有 8 个 TCP 和 IP 报头。如果每个传输单元需要 40 字节开销，则有 8×40 字节的开销加上 1024 字节的有效载荷。总的传输字节数为 1344 字节。开销所占的百分比为：320÷1344×100%=23.8%。
    b) IPv6 报头的最小长度是 40 字节。再加上 TCP 报头中的 20 字节，则每个传输都包含 60 字节的开销。因此，要发送 8 个有效载荷，开销的总字节数为 480 字节；开销所占的百分比为：480÷1504×100%=31.9%。

13. a) 1500～1599 字节
    b) 1799 字节

19. b) 信噪比（dB）$=10\log_{10}\frac{2888\text{dB}}{40\text{dB}}=18.6\text{dB}$

    b) $0.32\text{dB}=10\log_{10}\frac{\text{信号 dB}}{35\text{dB}}\Rightarrow$信号 dB$=37.68\text{dB}$

附录 A

3. 一个更有效率的链接表的实现是在链表中的每个节点设置 3 个指针。这个附加指针的作用是什么？

5.

