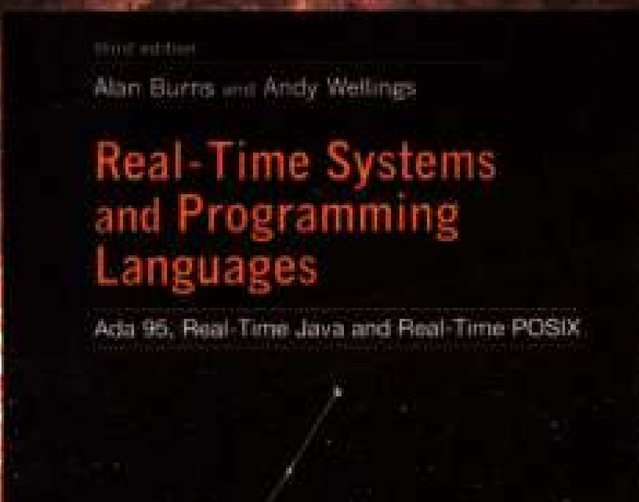
计算机科学丛书

原书第3版

实时系统与编程语言

(英) Alan Burns　Andy Wellings 著　王振宇　陈利　等译



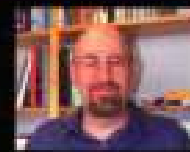
Real-Time Systems and Programming Languages

Ada 95, Real-Time Java and Real-Time POSIX, Third Edition

机械工业出版社 China Machine Press　中信出版社 CITIC PUBLISHING HOUSE

本书在国际上是一本实时系统方面的畅销教材。全面论述实时系统、嵌入式系统和分布式系统的特征，深入分析设计和实现实时嵌入式系统的需求，批评性地介绍了当前的编程语言和操作系统对设计和实现实时系统的支持，重点是Ada95、实时Java、实时POSIX以及实时CORBA。本书建议了对于实现不同的实时系统所使用的最佳编程语言。本书覆盖的丰富内容是其他关于实时（或并发）编程语言的书籍所无可比拟的。

作者简介



Alan Burns 是英国约克大学计算机科学系的教授，他撰写和与其他人合著了300多篇论文和10本书，大部分是关于Ada和实时领域的。他还曾是IEEE实时系统技术委员会的主席（2001-2003）。

Andy Wellings 是英国约克大学计算机科学系实时系统方面的教授，撰写了200多篇论文和报告以及5本书。他还是 *Software Practice and Experience* 杂志的欧洲主编。

ISBN 7-111-13987-9

9 787111 139874

华章图书

网上购书：www.china-pub.com

北京市西城区百万庄南街1号 100037
读者服务热线：(010)68995259，68995264
读者服务信箱：hzedu@hzbook.com
http://www.hzbook.com

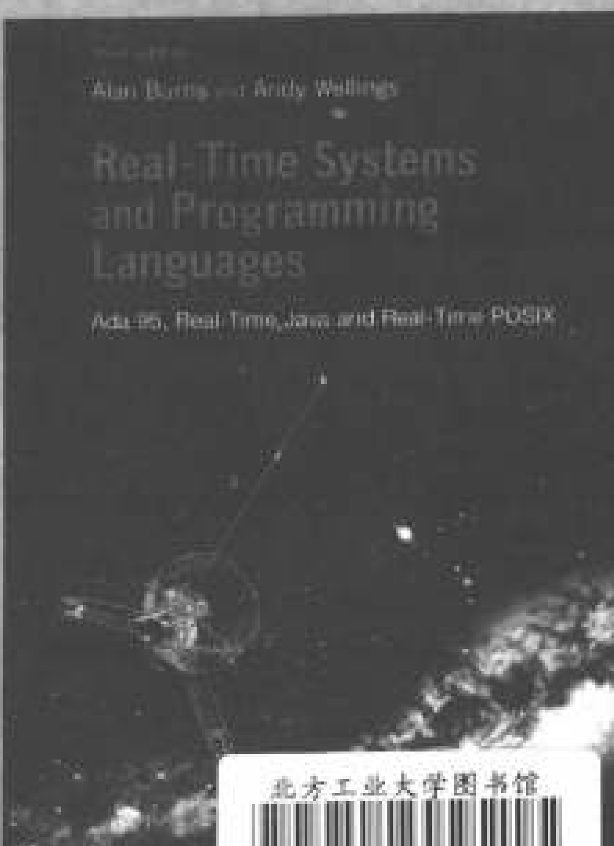
ISBN 7-111-13987-9/TP · 3471
定价：59.00 元

计 算 机 科 书

原书第3版

实时系统与编程语言

(英) Alan Burns　Andy Wellings 著　王振宇　陈利　等译



Real-Time Systems and Programming Languages

Ada 95, Real-Time Java and Real-Time POSIX, Third Edition

机械工业出版社 China Machine Press　中信出版社 CITIC PUBLISHING HOUSE

本书全面论述实时系统、嵌入式系统和分布式系统的特征，深入分析设计和实现实时嵌入式系统的需求，并讨论了当前的编程语言和操作系统如何满足这些需求，重点介绍Ada 95、实时Java和实时POSIX。本书还覆盖了在实时领域的最新成果，包括实时CORBA。

本书在国外是实时系统方面的畅销教材，涵盖的内容广泛，适合作为高等院校计算机专业的教材，供高年级本科生和研究生使用。

Alan Burns and Andy Wellings: Real-Time Systems and Programming Languages: Ada 95, Real-Time Java and Real-Time POSIX (Third Edition) (ISBN: 0201729881)

本书版权登记号：图字：01-2002-0605

图书在版编目（CIP）数据

实时系统与编程语言：原书第3版 /（英）波恩斯（Burns, A.），（英）威林斯（Wellings, A.）著；王振宇等译. －北京：机械工业出版社，2004.4

（计算机科学丛书）

书名原文：Real-Time Systems and Programming Languages: Ada 95, Real-Time Java and Real-Time POSIX (Third Edition)

ISBN 7-111-13987-9

Ⅰ.实… Ⅱ.① 波… ② 威… ③ 王… Ⅲ.① 实时操作系统 ② 程序语言 Ⅳ.① TP316.2 ② TP312

中国版本图书馆CIP数据核字（2004）第013761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：李 英

北京瑞德印刷有限公司印刷·新华书店北京发行所发行

2004年4月第1版第1次印刷

787mm × 1092mm 1/16 · 37.25印张

印数：0 001 - 4 000册

定价：59.00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，有的还专诚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: hzedu@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
周立柱	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		

秘　书　组

武卫东	温莉芳	刘　江	杨海玲

译 者 序

Alan Burns 和Andy Wellings的这本《实时系统与编程语言》于1989年出第1版，1997年出第2版，2001年出第3版，2003年则是第3版的第3次印刷，可见它是一本受欢迎的书。Alan Burns 和Andy Wellings 都在英国约克（York）大学计算机科学系工作。Burns教授是IEEE实时系统委员会的主席（2001-2003）。Andy Wellings 教授是*Software Practice and Experience*杂志的欧洲主编。

书中全面论述了实时系统、嵌入式系统和分布式系统的特征，深入分析了设计和实现实时嵌入式系统的需求，批判性地介绍了几种编程语言和操作系统对设计和实现实时系统的支持，重点是Ada 95、实时Java和实时POSIX。书中覆盖了此领域的最新成果，包括实时CORBA。

在这本书的翻译过程中，关于若干术语的译法碰到不少问题。这一方面是由于国外一些作者和机构对这些术语的用法历来就有不同，另一方面是由于中西文化背景的差异。我们的原则是这本书中的译法要尽可能统一，不致使读者混淆,并在必要时加注原文。另外，做到“信、达、雅”全面兼顾，当然很好，但由于种种原因无法兼顾的时候，只有牺牲“雅”了。相信翻译过科技著作的人会同意这种取舍。

原书作者针对第3版给出了勘误表，见:

http://www.booksites.net/burns

译文已据此做了相应更改。遗憾的是还发现另外一些错误，译文中也改正了。

参加本书翻译工作的除封面署名外还有：王志海（武汉理工大学）、张立（武汉化工学院）、陈靖（中船重工709研究所）、魏非（中船重工709研究所）、余扬（华南理工大学）、徐冠勇（中船重工709研究所）、李扬（中船重工709研究所）、李任（中船重工709研究所）等。

王振宇（中船重工709研究所）
陈利（华中师范大学）
2003年9月1日

前　言

1981年，一个软件错误导致一个静止的机器人突然移动，并以极快的速度冲到操作区的边缘，附近的一个工人被撞死。这只是嵌入式实时系统灾难的一个例子。但是这并非一个孤立的意外事件。每个月，《Software Engineering Notes》（软件工程通讯）都有几页关于实时系统的故障将公众或环境推入危险境地的例子。这些认真描述的事件证明了需要对嵌入式系统有一个系统范围的观点。确实，有人主张说现在需要将实时系统作为一个单独的工程学科来认识。本书就是要为建立这个学科做出贡献。当然，它不可能覆盖有关实时系统工程研究的所有问题，然而，它确实提供了对在此领域使用的编程语言和操作系统标准的全面描述和评估。特别强调的重点是语言原语及其在产生可靠、安全、可依赖软件中的作用。

本书面向的读者

本书针对计算机科学和相关专业高年级本科生和硕士研究生，也适合于专业软件工程师和实时系统工程师。本书假设读者具有像Pascal这种顺序编程语言的知识，并熟悉软件工程的基本原则。书中所用材料是作者过去多年里在各种大学和相关行业所开课程的内容，这些课程都是专门讲述实时系统和编程语言的。

结构和内容

为了各章的连贯性，本书详细地研究了4种编程语言：Ada、Java、occam2和C。之所以选中这些语言，是因为软件生产中真正使用这些语言。由于C是一个顺序语言，它要连同POSIX操作系统的接口一起使用（尤其是实时扩展）。本书也讨论其他的理论性或实验性语言，只要它们提供核心语言中没有的原语。这些语言的实践人员应当可找到他们需要的足够材料。作者相信，像对Ada或Java这样的语言的全面评价只能通过对它们进行比较性研究才能得出。

本书总共有18章，前13章分成五部分。第1章至第4章是一个深入的引言，提出了实时系统的特征和需求，然后给出此类系统的设计概述。设计不是本书的关注重点，然而它对于在适宜的环境中讨论实现问题是很重要的——第2章提供这样的背景。这一章还研究语言评估的一般标准。第3章和第4章通过关于*小型编程*和*大型编程*的讨论研究基本语言结构。这些章还介绍Ada、Java、occam2和C。熟悉这些语言和实时系统基本特性的读者可以快速地浏览这开头的四章。对于其他读者，这些材料有助于使本书更加自成体系。

第5章和第6章讨论可靠软件部件的生产问题。虽然考虑到了故障预防，但重点主要集中在容错，研究了向前和向后两种出错恢复技术。第6章讨论异常处理，描述恢复和终止两种模型，以及在Ada、Java、CHILL、CLU、C＋＋和Mesa中的语言原语。

实时系统的固有特征是并发，所以研究编程语言的这个方面是很重要的。第7章介绍进程的概念并概述语言和操作系统设计者使用的许多不同模型。在接下来的两章中研究进程之间的通信，首先描述共享变量方法，包括信号量、管程、互斥锁和保护对象的使用。在现代编程语言中，基于消息的模型也是很重要的，按它们进行通信和同步予以组合。第9章涵盖这些

模型，并特别关注Ada和occam2的会合原语。

对于在此书中是应当首先研究可靠性还是研究并发性的问题是有争论的。两位作者都试着反转次序，发现在这两种方法之间很难选择。事实上，这本书可以以任一种方式使用，只有一、两个主题是“不合适”的。首先讲解可靠性的决定反映了作者关于安全性是实时系统的根本需求的信念。

下一部分包括第10章和第11章。通常将系统进程之间的关系描述为合作关系（实现一个共同目标）或竞争关系（获得共享资源）。第10章通过描述如何将可靠的进程合作进行编程扩展了前面关于容错的讨论。这个讨论的核心是*原子动作*的概念和异步事件处理技术。接下来的一章研究竞争进程，评价了各种语言特征。这里的一个重要问题是并发模型里的条件同步和回避同步之间的区别。

时间性需求是实时系统的区别性特征。第12章和第13章详细讨论这些需求和用以满足需求的语言设施以及实现策略。硬实时系统的时间性约束是必须满足的；软实时系统可以偶尔不满足。二者都是在时限调度的背景下讨论的。*优先级*概念是同基于抢占式优先级系统的可调度性分析一起讨论的。

剩下的几章本质上是独立的。近期在硬件和通信技术方面的进展已经使分布式计算机系统在嵌入式应用领域成为单处理器和集中式系统的可行的替代品。虽然在某些方面，分布可被看作是实现问题；然而，当应用被分布时，出现的问题提出了超出实现细节的基础性问题。第14章研究相关的四个方面：语言支持、处理器和通信失效情况下的可靠性、分布式控制算法和分布式调度。把这些内容独立成章，目的是使上短期课程的学生可以跳过它。

许多实时系统的一个重要需求是加进来的外部设备必须作为应用软件的一部分编程（也就是说必须予以控制）。这种低级编程同作为软件工程特征的软件生产的抽象方法很不相同。第15章研究低级设施能够被成功地并入高级语言的方式。

对于实时系统的一个普遍错误概念是认为它们必须是高效的，这本身是不对的。实时系统必须满足时间约束（和可靠性需求）；高效的实现是扩展可能性范围的一种手段，但其本身不是目的。第16章概述了执行环境在获得高效可预测的实现中的作用。

本书最后的一章是用Ada实现的一个个案研究，使用了矿井控制系统的例子。一个经过裁减的个案研究不可能说明前面各章涵盖的所有问题，特别是没有涉及大小和复杂性等因素。然而，这个个案研究确实覆盖了实时系统的许多重要方面。

各章都有小结和相关阅读材料清单，多数章还有练习。选择这些练习是为了帮助读者巩固对每一章内容的理解。它们大体上代表了作者进行评估所使用过的练习。

Ada、Java、occam2和C

本书中Ada 95的例子符合ISO/ANSI标准。Java例子符合Java 2平台和Java扩展的实时规格说明（在本书中称为实时Java）。occam2例子符合INMOS给出的occam2定义。C例子符合ANSI C，POSIX原语是在POSIX.1、POSIX.1b、POSIX.1c、POSIX.1d和POSIX.13定义中给出的原语。

为方便四种语言的识别，使用了不同的表现风格。Ada使用加粗小写字母的关键字，程序标识符则以大小写混合方式给出。occam2的关键字用不加粗大写字母，标识符用混合大小写字母。C的关键字不加粗，标识符用小写字母。为将Java同C区分，Java关键字是加粗的，标

识符是混合大小写字母。

对第2版的修改

在第2版中，我们从Ada 83移到Ada 95，从Modula-2移到了C和POSIX。第3版根据实时领域的下列发展进行了改进:

- Java通过实时Java扩展成为一个实时语言。因此本书将Java作为核心语言之一对待。
- POSIX标准中补充的新实时功能，尤其是执行时间监控（Execution Time Monitoring）和偶发服务器（Sporadic Server）。第12章和第13章的更新反映了标准的这些修改。
- 为解决实时问题对COBRA的建议扩展。CORBA显然有理由成为一个主要议题，但详细讨论它超出了本书的范围。然而，第14章的更新还是反映了CORBA方法。

我们已经接受了使用本书教学的读者关于包含更多调度方面材料的建议。

教学辅助材料

从下列网站可得到本教科书的更多支持材料: http://www.booksites.net/burns。本书许多部分有投影胶片可用，还提供许多练习的答案。我们将随时添加更多的练习、适当的新例子和补充教学材料。使用本书的教师/讲师帮助完成了这些网页。

关于约克大学的实时系统研究

Alan Burns和Andy Wellings是英国约克大学计算机系实时系统研究组的成员。这个小组对实时系统的设计、实现和分析的所有方面进行研究。这个小组尤其致力于: 形式化和结构化开发方法、调度理论、重用、语言设计、内核设计、通信协议、分布和并行体系结构、程序代码分析。这个研究组的宗旨是进行基础性研究，并将现代技术、方法和工具带入工程实践。研究组的应用领域包括空间和航空电气系统、引擎控制器、汽车控制和多媒体系统。可通过http://www.cs.york.ac.uk/rts/找到这个小组活动的进一步信息。

第1版的致谢

本书中的材料是在过去五年里建立起来的，并在英国布拉德福德大学（Bradford）和约克大学计算机科学和电子学系向三年级本科生和研究生讲授过。我们要感谢他们对本书最终出版的贡献，没有他们将不可能写成这本书。

许多人读过本书的第一稿并提出意见。我们特别要感谢Martin Atkins、Chris Hoggarth、Andy Hutcheon、Andrew Lister和Jim Welsh。我们还要感谢我们在大学里的同事，他们为我们提供了激励性的环境和许多有价值的讨论，特别是Ljerka Beus-Dukic、Geoff Davies、John McDermid、Gary Morgan、Rick Pack、Rob Stone和Hussein Zedan。

1988年Alan Burns在澳大利亚昆士兰（Queensland）大学和美国休斯顿（Houston）大学进行假期研修。我们感谢在这里的所有同事，特别是Andrew Lister、Charles McKay和Pat Rogers。

如果不用通过JANET传送的电子邮件，也不可能写成这本书。我们要感谢英国大学大理事会和科学工程研究基金会的计算机处，它们提供了这种无价的服务。

最后，我们还要特别感谢Sylvia Holmes和Carol Burns，感谢Sylvia在最终手稿上进行辛苦的校对，感谢Carol容许我们进行许多晚间的会议和讨论。

第2版的致谢

许多人帮助我们出版本书第2版。我们要特别感谢Alejandro Alonso、Angel Alvarez、Sergio Arevalo、Neil Audsley、Martin Dorey、Michael Gonzalez Harbour、Stuart Mitchell、Gary Morgan、Offer Pazy和Juan de la Puente。

第3版的致谢

我们要感谢实时Java专家组，他们在建立实时Java规格说明时采用了开放的方式。还感谢Angel Alvarez、Jose Alvarez、Neil Audsley、Iain Bate、Jorge Diaz-Herrera、David Duke、Alan Grigg、Ian Hayes、George Lima、Greg Murphy、 Peter Puschner 和Pat Rogers，在写这一版时，他们向我们提供了各种形式的帮助。

我们还想感谢技术评审员Jorge Diaz-Herrera、Jörgen Hansson和Robert Holton，他们对本版的初稿提出了很有价值的意见。

最后，我们要感谢所有向本书第2版提出意见的人。

Alan Burns和Andy Wellings

于英国约克大学

2000年11月

目　录

第1章 实时系统引论

随着计算机变得更小、更快、更可靠和更便宜，其应用范围更宽了。最初制造的计算机只是作为方程求解装置，现在，其影响已经拓展到生活的各个方面，从洗衣机到空中交通管制。扩展最快的一个计算机应用领域所涉及的各种应用，其主要功能不是信息处理，然而需要信息处理以实现其主要功能。微处理器控制的洗衣机是这种系统的一个好例子。这里，基本功能是洗衣服，然而，为了洗不同种类的衣服，要执行不同的“洗衣程序”。这种类型的计算机应用通常被称为**实时**应用或**嵌入式**应用。估计世界上有99%的微处理器是用于嵌入式系统的。这些嵌入式应用的编程对计算机语言提出了特别的要求，因为它们同传统的信息处理系统有不同的特征。

本书是关于嵌入式计算机系统及其编程语言的。本书研究这些系统的特有性质，并讨论现代实时编程语言和操作系统的演变过程。 1

1.1 实时系统的定义

在进一步讨论之前，需要更精确地定义“实时系统”这个词。对实时系统的确切特性有许多解释，然而在响应时间概念方面是共同的：指从某些相关输入产生输出所花的时间。《Oxford Dictionary of Computing》（牛津计算词典）对实时系统给出下列定义：

> 实时系统是指那些产生输出的时间至关重要的系统。这通常是因为输入对应于外界的某个运动，而输出又必须与同一运动相关。自输入时刻到输出时刻的时间滞后必须充分小，以达到可接受的及时性。

这里，“及时性”这个词要在整个系统的背景中加以考虑。例如，在一个导弹制导系统里，输出要求在几个毫秒之内，而在一个计算机控制的汽车装配线上，响应可能只要求在一秒之内。为说明“实时系统”定义的各种方式，给出两个另外的定义。Young（1982）定义实时系统是：

> 任何必须在有限、指定的周期内对外部发生的输入激励做出响应的信息处理活动或系统。

PDCS（Predictably Dependable Computer Systems）（Randell等，1995）工程给出如下定义：

> 实时系统是这样的系统：需要在环境限定的时间间隔里对来自环境的激励做出反应（包括实际时间的推移）。

在最一般的意义下，所有这些定义涉及非常广泛的计算机处理活动。例如，像Unix这样的操作系统可被看作一个实时系统，因为当一个用户输入一个命令时，他（或她）会期待在几秒内得到响应。还好，如果响应没有如期而来，通常还不是一个灾难。这种系统有别于那些认为不能及时响应是与错误响应同样糟糕的系统。确实，在一定程度上，实时系统正是在这方面有别于那些响应时间虽然重要但并非决定性的系统。所以，*实时系统的正确性不仅依赖于计算的合理结果，还依赖于产生这个结果的时间*。实时计算机系统设计的人员常常区分**硬**实时和**软**实时系统。硬实时系统是那些在规定的时限前做出响应是绝对强制性要求的系统。软实时系统是那些响应时间虽然重要，但如果偶尔错过时限系统依然正常运行的系统。软实时系统同交互式系统的区别是：对后者而言无明显的时限。例如，战斗机的飞行控制系统是
2
一个硬实时系统，因为错过时限可能导致一场大灾难，而过程控制应用的数据获取系统是软实时的，因为它可被定义为按固定的时间间隔对输入传感器采样，但容许间歇的延迟。当然，许多系统同时有硬实时和软实时子系统。事实上，某些服务可以既有一个硬时限，又有一个软时限。例如，对某个警告事件的响应可以有一个50ms的软时限（用于做出最佳有效反应）和一个200ms的硬时限（保证不对设备和人员产生伤害）。在50ms和200ms之间，响应的“价值”（或用处）下降。

正如这些定义和例子所说明的，“软”这个词的使用并不是指一种单一的需求，它还伴随着一些不同的性质。例如：

- 可以偶尔错过时限（通常有一个在确定的时间间隔内的错过次数上限）。
- 可以偶尔推迟提供服务（同样，有一个延迟次数的上限）。

那些可被偶尔错过但推迟提交服务并没有带来好处的时限，叫做**固时限**。在某些实时系统中，对任选的固部件可以给出概率性需求（例如，硬服务必须每300ms产生一个输出，至少有80%的时间，这个输出由一个固部件X产生；在其他情况下，将使用一个硬的但功能上简单得多的部件Y）。

在本书中，“实时系统”这个术语既指软实时又指硬实时。在专门讨论硬实时系统的地方，将明确使用“硬实时”这个术语。

在硬实时或软实时系统里，计算机通常同某个物理设备直接对接，用于监视或控制那个设备的操作。所有这些应用的关键特征是计算机被用作更大工程系统中的一个信息处理部件。正因为如此，这种应用被称为**嵌入式计算机系统**。

1.2 实时系统的例子

说明了实时系统和嵌入式系统的含义之后，现在给出使用它们的例子。

1.2.1 过程控制

计算机作为更大工程系统中的一个部件的首次使用是在20世纪60年代早期的过程控制产业中，现在则使用微处理器。考察图1-1所示的简单例子，计算机完成单一的活动：通过控制
3
阀门确保管道中液体的均匀流动。

当检测到流量增加时，计算机必须通过改变阀门角度给予响应，该响应必须在一个有限时间段里发生以使管道接收端的设备不致过载。注意，实际响应可能涉及复杂的计算，以算出新的阀门角度。

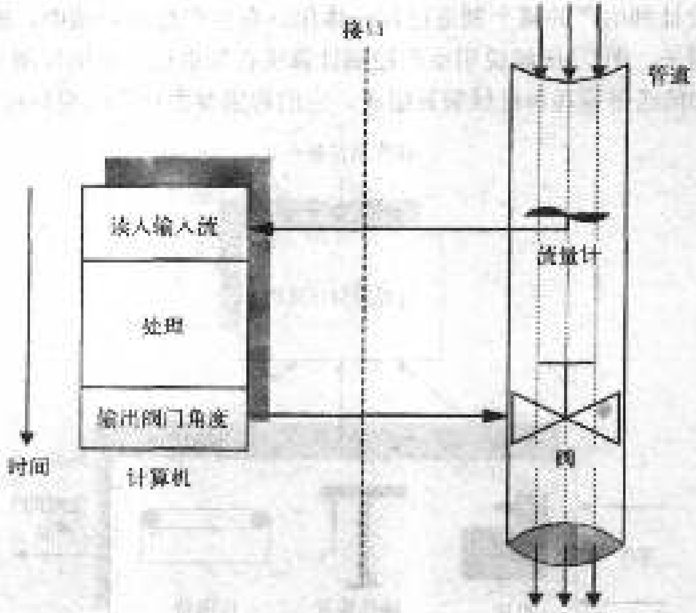


图1-1 液体控制系统

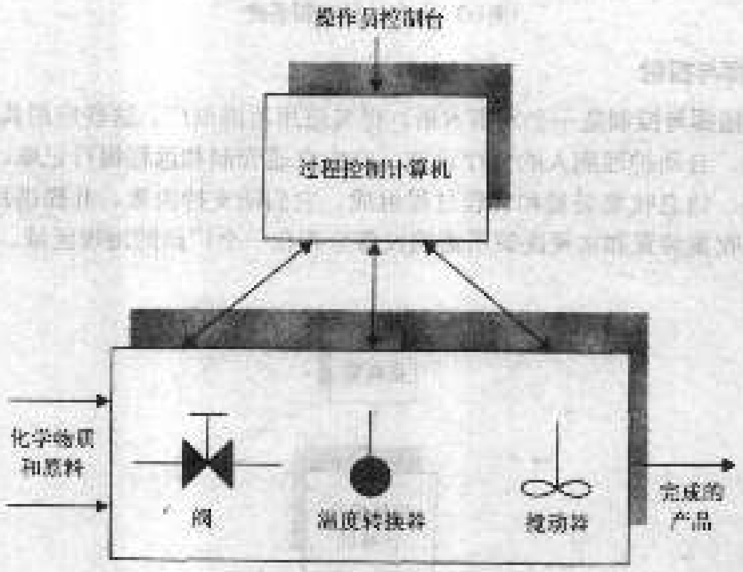


图1-2 过程控制系统

这个例子只展示了更大控制系统的一个部件。图1-2说明了嵌入到一个完整过程控制环境中的实时计算机的作用。此计算机同使用传感器和致动器的设备进行交互。阀门是致动器的一个例子，而温度和压力转换器是传感器的例子（转换器是产生一个同被测物理量成正比的电信号的设备）。计算机控制传感器和致动器的动作以确保在恰当的时间执行正确的设备操作。需要时，要在受控过程和计算机之间插入模数（和数模）转换器。

1.2.2 制造业

为保持生产的低成本和提高生产率，计算机在制造业的使用已变成必不可少的了。计算

机已经使自产品设计到生产的整个制造过程一体化。在生产控制领域中，嵌入式系统是说明
4 这种情况的最好例子。图1-3图解说明生产控制计算机在制造过程中的作用。实际系统由诸如机床、操作装置和传送带等各种机械装置组成，它们都需要由计算机控制和协调。

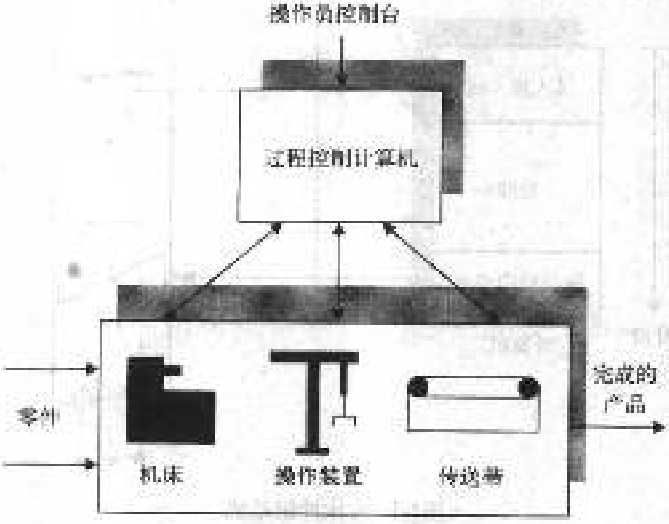


图1-3 生产过程控制系统

1.2.3 通信、指挥与控制

虽然通信、指挥与控制是一个军事术语，但其应用范围很广，这些应用具有相似的特征，例如，机票预订、自动护理病人的医疗设施、空中交通管制和远程银行记账。每个系统都由一组复杂的策略、信息收集装置和管理过程组成，它们能支持决策，并提供用以实现它们的手段。这些信息收集装置和实现决策所需的设备分布在一个广阔的地理区域。图1-4表示了这样一个系统。

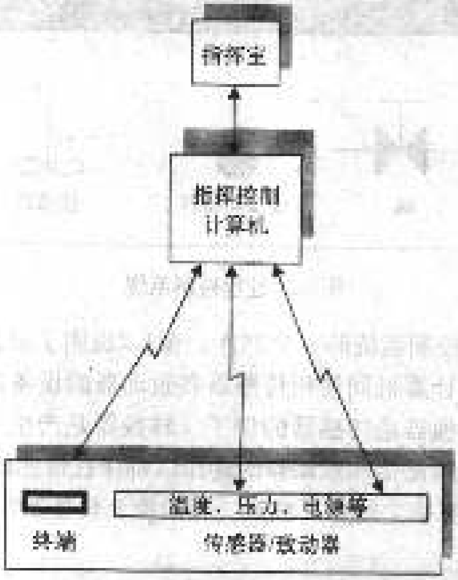


图1-4 指挥控制系统

1.2.4 广义嵌入式计算机系统

在所示的每个例子里，计算机都直接同现实世界的物理设备对接。为了控制这些设备，计算机需要按固定时间间隔对测量装置进行采样，因而需要一个实时时钟。这里通常设有一个操作员控制台以允许人工干预。通过各种显示（包括图形显示）方式将系统状态不断地通知操作员。

系统状态变化的记录放在数据库里，可由操作员查询，或者用于在系统崩溃时进行事后分析，或者是为管理提供信息。这种信息确实越来越多地用于支持在每天系统运行时所做的决策。例如：在化学工业和加工工业中，设备监控不单单对最大限度地提高产量是重要的，而且对于使经济效益最大化也是必不可少的。一个设备关于生产的决策可能对远距离的其他设备有严重的影响，特别是在一个过程的产品被用作另一过程的原材料的时候。

所以，一个典型的嵌入式计算机系统可用图1-5表示。控制系统操作的软件可以被编制成若干模块，它们反映环境的外部特性。通常有一个模块包含实际控制这些装置所必需的算法，一个模块负责记录系统状态的变化，一个模块检索并显示这些变化，还有一个模块同操作员进行交互。 5-6

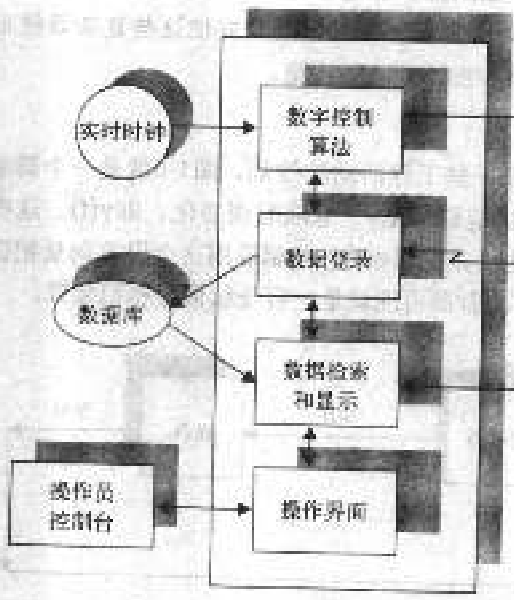


图1-5 典型的嵌入式系统

1.3 实时系统的特征

实时系统具有许多特征（固有的或强加的），以下几小节将阐明。显然，并非所有实时系统都具有所有这些特征；然而，用于实时系统有效编程的任何通用语言（和操作系统）都必须有支持这些特征的设施。

1.3.1 大且复杂

通常人们认为同开发软件相关联的大多数问题都是同大小和复杂性有关的问题。编写小程序不出现重大问题，因为设计、编码、维护和理解都由一个人完成。如果使用这个软件的

人离开了公司或机构，另一个人能够在相对短的时间里学会这个程序。确实，有一种构建这种程序的艺术或技巧，而且小就是美。

令人遗憾的是，不是所有软件都有这种理想的“小巧”特征。Lehman和Belady（1985）在试图描述大系统的特征时，拒绝了那种认为庞大就是同指令数目、代码行数或组成一个程序的模块数目成正比的简单而且或许是直观的概念。他们将庞大同**多样性**（variety）联系起来，将庞大的程度同多样性的数量联系起来。诸如指令数目和开发工作量这样的传统指标，就只是多样性的表征。

> 多样性就是现实世界中的需要和活动的多样性和它们在程序中的反映。但现实世界是不断变化的，它处于演变之中，而社会的需要和活动也是这样。因此，像所有复杂系统一样，庞大程序也一定是不断演变的。

由嵌入式系统的定义可知它们必须对现实世界的事件做出响应，因此必须提供同这些事件关联的多样性，所以，程序将呈现出不受欢迎的庞大性质。不断变化的概念是庞大的上述定义中固有的。为应付现实世界中不断变化的需求而重新设计或重新编制软件的成本是高昂的，所以在实时系统的生命周期中要进行不断的维护和增强。它们必须是可扩充的。

虽然实时系统常常是复杂的，但由实时语言和环境提供的设施使这些复杂系统可被分解
7~8 成能有效管理的较小部件。第2章和第4章将详细研究这些设施。

1.3.2 实数处理

如早些时候说过的，许多实时系统涉及一些工程活动的控制。图1-6就是一个简单控制系统的例子。受控实体，即设备，有一个输出变量向量y，它随时间变化，即y(t)。这些输出同理想（或参考）的信号r(t)进行比较，产生误差信号e(t)。控制器用这个误差向量把设备的输入变量改变为u(t)。对于非常简单的系统，控制器可能就是一台连续信号模拟装置。

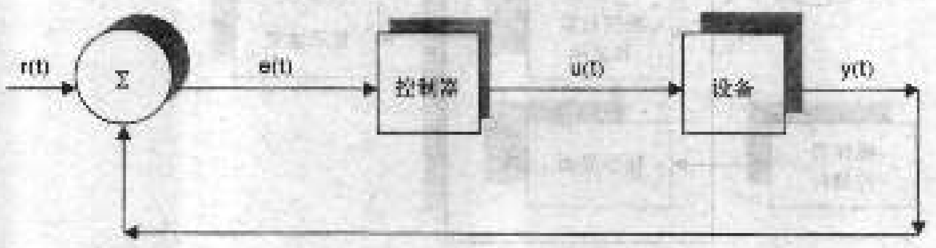


图1-6 简单控制器

图1-6举例说明了一个反馈控制器，这是最普通的形式，但也使用前馈控制器。为了计算出如何改变输入变量向量才能在输出向量上产生理想效果，该设备必须有一个数学模型。这些模型的推导属于另一个学科——控制论。设备的模型常常是一个一阶常微分方程组。这些微分方程建立了系统的输出同设备的内部状态和其输入变量之间的联系。改变设备的输出涉及求解这些微分方程，以得到所需的输入值。大多数物理系统都有惯性，所以改变不是瞬时的。在一个固定时间段里使系统移动到一个新的设置点上的实时需求，将增加数学模型和物理系统所需处理的复杂性。事实上，线性一阶方程仅仅是系统实际特性的一个近似，这个事实也展示了复杂程度。

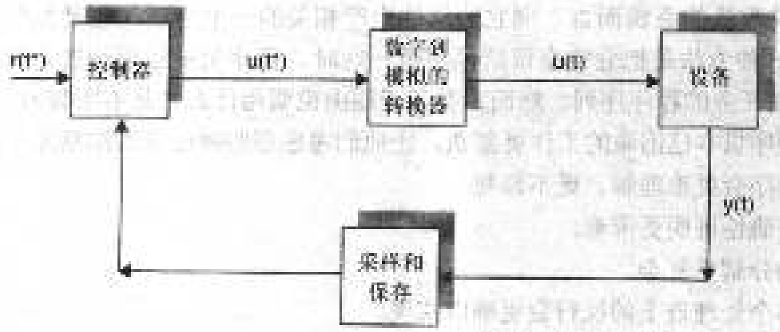


图1-7 一个简单计算机化的控制器

由于这些困难、模型的复杂性、不同（但非无关的）输入和输出的数目，大多数控制器都用计算机实现。将数字部件引入到系统里面改变了控制循环的性质。图1-7是对较早模型的修改。标记了*的项目现在是离散值，采样和保存操作由一个模拟/数字转换器进行，两个转换器都在计算机的直接控制之下。

在计算机上，微分方程可用数值方法求解出来，但算法本身需要调整，以将设备输出正被采样这个事实考虑进来。控制算法的设计是超出本书范围的一个论题，但本书会直接涉及这些算法的实现。它们可能在数学上是复杂的并要求较高精度。所以，对实时编程语言的一项基本要求就是能够处理实数或浮点数，第3章中将连同其他数据类型研究这个问题。

1.3.3 极其可靠和安全

社会越是把要害功能的控制让位给计算机，就迫使计算机越是不能失效。银行之间的自动资金转移系统的失效会导致无法挽回的数百万美元的损失；发电厂的一个有故障部件可能导致重症监护室的一个关键的生命支持系统失效；一个化工厂的过早关闭能够引起对设备的严重损坏或环境危害。这些颇为惊人的例子说明计算机硬件和软件必须是可靠的和安全的。即使在像军事应用这样的敌对环境里，系统的设计与实现也必须使系统只能在受控的方式下失效。此外，在需要操作员交互的地方，要仔细地设计界面，以把人为错误的可能性降低到最小限度。

实时系统的大块头和复杂性加重了可靠性问题，不仅必须考虑到应用中固有的、可预见的困难，还要考虑到有差错的软件设计引入的困难。

在第5章和第6章，将讨论生产出可靠、安全的软件的问题，同时还讨论语言中所引入的 9
一些用以对付预期的和非预期的出错状况的设施。这个问题还要在第10－14章进一步考察。 10

1.3.4 独立系统部件的并发控制

嵌入式系统通常由计算机和若干共存的外部构件组成，计算机程序必须同它们同时交互。并行地存在是这些外部现实世界构件的本性。在我们典型的嵌入式计算机例子中，程序要同一个工程系统（由许多并行部件组成，如机器人、传送带、传感器、致动器，等等）、计算机显示装置、操作员控制台、数据库和实时时钟交互。幸好，现代计算机的速度使这些动作能顺序进行，却给人以同时进行的错觉。然而，在某些嵌入式系统中就不是这样，例如，在地理上分布的各种站点收集并处理数据，或是多个单独部件的响应时间不能由单台计算机满足的情况。在这些情况下，必须考虑分布式和多处理器嵌入式系统。

对表现出并发性的系统而言，同它的软件生产相关的一个主要问题是如何在程序结构中表达并发性。一种方法是把它完全留给程序员，这时，程序员就必须构造系统以循环执行一个处理各种并发任务的程序序列。然而，有若干理由说明为什么这是不明智的：

- 它使得程序员本已困难的工作更复杂，让他们考虑那些同任务的控制无关的结构问题；
- 产生的程序会更难理解、更不雅观；
- 使程序正确性证明更困难；
- 使问题的分解更复杂；
- 程序在多个处理器上的执行会更难以实现；
- 把处理故障的代码放在哪里更成问题。

较早的实时编程语言，例如RTL/2和Coral 66，依赖操作系统对并发性的支持。C通常同Unix或POSIX联系在一起。然而，较现代的语言，如Ada、Java、Pearl和occam直接支持并发编程。在第7、8、9章详细研究各种并发编程模型。接着的两章将集中讨论在有设计错误的情况下如何实现并发进程之间的可靠通信与同步。在第14章，讨论分布式环境中的执行问题以
11 及容许处理器和通信失效问题。

1.3.5 实时设施

在任何实时系统中，响应时间都是决定性的。但是，要设计和实现保证在所有可能条件下都能在适当的时候产生适当的输出是非常困难的。为此，在所有时刻充分地利用计算资源常常是不可能的。由于这个原因，实时系统通常被构建成处理器的使用都有可观的空闲能力，以保证“最坏情况行为”不致在系统操作的关键阶段产生任何不希望的延迟。

有了适当的处理能力，需要语言和运行时的支持使程序员能够：

- 规定动作进行的时间。
- 规定动作完成的时间。
- 对所有时间需求都不能满足的情况做出响应。
- 对时间需求动态改变（例如，模式改变）的情况做出响应。

这些叫做实时控制设施。它们使程序能够同时间本身同步。例如，对于直接数字控制算法，必须在每天的某个时间（例如下午2点、下午3点等）或以固定间隔（例如，5秒）从传感器读采样数据，而所用的模数转换器的采样率可从几百赫兹到几百兆赫兹变化。作为这些读数的结果，还需要进行其他的动作。例如，在一个发电厂里，需要为家庭消费者在周一到周五的下午5点增加电力供应。这是对从工作场所回家的家庭开灯、做饭等等引起的峰值需要的响应。在英国，近几年中家庭电力需求在1990年世界杯足球赛决赛结束后达到峰值，当时几百万观众离开他们的起居室，打开厨房的灯，并打开烧水壶，为的是冲一杯茶或咖啡。

模式改变的例子可在飞机交通管制系统中找到。如果一架飞机发现压力降低，那就立即需要所有计算资源放弃原有工作去处理紧急情况。

为了满足响应时间，必须使系统的行为是可预测的。将在第12、13章连同语言设施（用于辅助对时间性极强的操作编程）一起讨论这些问题。

1.3.6 同硬件接口的交互

嵌入式系统的特性要求计算机部件同外部世界交互。它们需要为现实世界里种类繁多的设备监视传感器和控制致动器。这些设备经由输入和输出寄存器同计算机交互，它们的操作

需求是依赖于设备和依赖于计算机的。这些设备还产生中断，以通知处理器某些操作已经完成或引发了出错状态。 12

过去，同设备的对接要么是留给操作系统去控制，要么需要程序员使用汇编语言插入对寄存器和中断的控制和操纵。现在，因为设备多种多样，而且和它们相关的交互具有时间紧迫的本性，所以对它们的控制经常必须是直接的，不再经过操作系统的功能层次。此外，可靠性需求也反对使用低级编程技术。

在第15章，将要研究实时编程语言的某些设施，它们能用以给出设备寄存器和中断控制的规格说明。

1.3.7 高效的实现和执行环境

由于实时系统对时间的要求极为苛刻，实现效率要比在其他系统中更重要。有趣的是，使用高级语言的主要好处是使程序员能将实现细节抽象掉，集中精力去解决手头的问题。但是，嵌入式计算机系统程序员不能享受这种好处。他或她必须不断地关心使用特定语言功能的代价。例如，对某个输入的响应需要在一毫秒之内，那么就没有必要使用一个执行时间为一微秒的语言功能!

在第16章，将要讨论执行环境在提供有效的和可预测实现方面的作用。

小结

本章中，实时系统被定义为:

> 任何必须在有限、指定的周期内对外部发生的输入激励做出响应的信息处理活动或系统。

这种系统主要分为两类：硬实时系统，是那些在规定的时限前做出响应是绝对强制性要求的系统；软实时系统，是那些响应时间虽然重要但如果时限偶尔错过，系统功能依然正常运行的系统。

本章研究了实时系统或嵌入式系统的基本特征。这些特征是:

- 大且复杂
- 实数处理
- 极其可靠和安全 13
- 独立系统部件的并发控制
- 实时控制
- 同硬件接口的交互
- 高效的实现

相关阅读材料

Bailey, D. L. and Buhr, R. J. A. (1998) *Introduction to Real-Time Systems: From Design to Networking with C/C++*. Upper Saddle River, NJ: Prentice Hall.

Buttazzo, G. C. (1997) *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. New York: Kluwer Academic.

Joseph M. (1992) *Specification, Verification and Analysis*. Englewood Cliffs, NJ: Prentice Hall.

Kavi K. M. (1992) *Real-Time Systems: Abstractions, Languages and Design Methodologies*. Los Alamitos, CA: IEEE Computer Society Press.

Koptez H. (1997) *Real-Time Systems*. New York: Kluwer Academic.

Laplante P. (1997) *Design and Application of Real-Time Systems*. New York: Institute of Electrical & Electronic Engineers.

14 Schneider, S. (1998). *Concurrent and Real-Time Systems*. New York: John Wiley & Sons.

第2章 设计实时系统

显然，任何实时系统开发的最重要阶段是生成一个一致的、满足权威性需求规格说明的设计。在这一点上，实时系统同其他计算机应用没有什么不同，尽管它们的总体规模经常产生非常基本的设计问题。采用怎样的方法、工具和技术才能确保软件生产过程是可管理的，并构造出可靠的正确程序，围绕这一问题的研究，形成了软件工程学科。假定读者熟悉软件工程的基本原则，因而这里仅考虑实时嵌入式系统的特有问题和需求。即使这样，也不可能对众多设计方法学给出一个详尽说明。设计问题本身并不是本书关注的焦点。对使设计得以实现的语言和操作系统原语的研究才是本书的主题。在这个背景下，我们将会详细研究Ada语言、(实时) Java、occam2语言和C语言（连同实时POSIX）。读者可以参考本章后面的相关阅读材料，了解关于设计过程的更多内容。 15

虽然几乎所有设计方法都是自顶向下的，但它们都考虑到“较低级别上的可行性”。本质上，所有设计方法都包括一系列从初始的需求描述到执行代码的转换。本章给出了这条路所经过的典型阶段的概述，这几个阶段是:

- 需求规格说明
- 体系结构设计
- 详细设计
- 实现
- 测试

本章也会讨论以下重要活动:

- 最终实现之前的原型建造
- 人机界面设计
- 评价实现语言的标准

因为不同的活动是独立进行的，需要用一些记号去编制每个阶段的文档。因此，从一个阶段到另一个阶段的转换只不过是把一种记号翻译成另一种记号。例如，编译器从用编程语言表示的源代码生成可执行代码。遗憾的是，其他的转换（上升到设计层次）没有定义得这么好，通常是因为所用记号过于含糊和不严密，以至不能完全捕获需求或设计的语义。

2.1 记号系统的级别

有许多种方法将记号（或表示）系统分类。如McDermid（1989）给出了一个有用的分解，他命名了三种技术：

1）非形式化的

2）结构化的

3）形式化的

非形式化方法通常使用自然语言和各种不严密的图表形式。它们的优点是许多人（就是所有说这种语言的人）理解这些记号。然而，众所周知，这样的用语，例如英语短语，常常有多种不同的解释。

结构化方法通常用一种图形化表示方法，但不像非形式化图表，这些图是有明确定义的。它们由少数预定义的部件以受控的方式互连构成。这种图形方式也有一个用某种明确定义的语言给出的语法表示方法。

尽管结构化方法可以被设计得非常严格，但它们本身不能被分析或处理。如果需要进行
16 这种操作，这种记号系统就需要有数学基础。具有这种数学特性的方法通常被认为是形式化的。它们的明显优点是可以用这些记号系统进行精确描述。此外，还可能证明具有的必要性质，例如，证明顶层设计满足需求规格说明。形式化技术的不足是它们不易被那些不准备或不能熟悉这套记号系统的人所理解。

实时系统的高可靠性需求已经导致了非形式化方法向结构化方法的转移，并逐步转向形式化方法。严格的验证技术也开始用于实时系统行业，但在目前，很少有软件工程师有必要的数学技能去开拓验证的全部潜能。另外，形式化技术通常没有足够的结构化工具去处理大型的规格说明。结果，结构化和形式化方法的结合使用就变得越来越流行。

2.2 需求规格说明

几乎所有计算项目开始都用非形式化方法描述项目想要什么。然后紧接着的应是详尽的需求分析。正是在这一阶段定义系统的功能。根据特定的实时因素，系统的时间性行为应该分析得非常清楚，同样，可靠性需求和当部件失效时软件的预期行为也要分析清楚。在需求分析阶段也要确定应对软件进行何种验收测试。

除了系统本身外，还有必要建立一个应用环境的模型。同环境有重要交互是实时系统的一个特征。因此，比如最大中断率、动态外部对象（如空中交通管制系统中的飞机）的最大数量和失效模式等问题都是十分重要的。

在需求分析中使用一些结构化记号和技术 [如PSL（Teichrow and Hershey，1977）和CORE（Mullery，1979）]，它的明显优点就是有一组无歧义的需求。此外，面向对象方法正变得越来越普遍（Monarchi and Puhr，1992），并出现了一个行业标准：UML——参看2.4.4节。已经进行了一些需求分析的形式化方法的工作，尤其是FOREST项目（Goldsack and Finkelstein，1991），它为处理需求定义了合理的方案和需求启发方法。最近，Esprit II ICARUS项目已经开发出ALBERT（Agent-oriented Language for Building and Eliciting Requirement for Real-Time Systems）语言（Dubois等，1995；Bois，1995）。尽管有这些发展，但没有任何结构化或形式化记号系统可以捕获“顾客”未能提及的需求。

分析阶段提供权威性的需求规格说明，而设计正是从这里出现。尽管这是软件生命周期中的最关键阶段，但是自然语言文档仍然是这种规格说明的常规记号系统。一个例子是，尽管计算机语言的语法能够容易地被形式化表达（使用某种BNF的形式），其语义通常却要用自然语言表达。Ada语言的原始版本就是如此“定义”的，并且它需要一个常务委员会裁决这些定义文档（这是一个国际标准）的实际含义。编译器的开发者必须数千次地询问这个委员会。Ada的当前标准仍然使用自然语言定义。与此对照，occam语言的语义已经使用指称语义来定义（Roscoe，1985）。这使编译器的形式化验证和严格的程序处理（如转换）工具成为可能。 17

或许现在使用相当广泛的形式化方法是VDM（Jones，1986）。另一个获得大量支持的技术是Z（Spivey，1989）。这两种方法都使用集合论和谓词逻辑，代表了在非形式化和纯结构化技术上的相当可观的改进。但是它们现在的形式不能完全处理实时系统的规格说明。

2.3 设计活动

大型嵌入式系统的设计不可能在一次练习中进行。它必须通过某种方式构造出来。为了管理复杂实时系统的开发，通常使用两种互补的方法：分解和抽象。它们一起形成了大多数软件工程方法的基础。分解，正如它的字面含义，就是有计划地将复杂系统分解成越来越小的部分，直到各部件完全独立，可被个人或小型的开发组理解和设计。在分解的每一级上，应该有适当的描述级别和记录（表达）该描述的方法。抽象可以推迟考虑细节，特别是属于实现的细节。这种方法可以得到整个系统和它所包含对象的简化视图，不过它仍然包含基本的属性和特征。抽象和分解的使用遍及整个工程过程，并且已经影响了实时编程语言的设计和相关的软件设计方法。

如果需求规格说明使用了一个形式化记号系统，那么顶层设计可以使用同样的记号系统，因而可以证明该设计符合规格说明。然而，许多结构化记号系统被提倡或者用于顶层设计，或是完全代替形式化记号。其实，结构化的顶层设计实际上可能就是权威性的需求规格说明。

2.3.1 封装

软件的层次化开发引导规格说明和随后的程序子部件的开发。抽象需要规定这些子部件应该有明确定义的作用、清晰无歧义的相互连接和接口。如果整个软件系统的规格说明仅根据当前子部件的规格说明就能验证，那么分解被认为是**可合成**的。这是形式化程序分析的一个重要特性。 18

顺序程序特别容易按合成方法来处理，并且已经有许多技术用于封装和表示子部件。Simula语言引入了重要的**类**构造。Modula-2使用不太强大但依然重要的模块结构。较近的*面向对象*语言如C++、Java和Eiffel已经建立在类构造的基础上。Ada使用模块和类型扩展的组合以支持面向对象编程。第4章深入讨论这些设施。

虽然对象提供抽象接口，但如果它们用于并发环境则需要额外的设施。典型情况涉及到增加某种进程形式。因此，*进程*抽象将是本书要关注的内容。第7章将介绍进程的概念，第8章将介绍基于共享变量的进程交互。但是更受控的和抽象接口是由基于消息的进程通信提供的，这将在第9章讨论。

在可靠的嵌入式系统的设计和实现中，对象和进程抽象都重要。

2.3.2 内聚和耦合

以上两种形式的封装导致使用带有定义明确的（抽象）接口的模块。但是大型系统应如何分解为模块呢？在很大程度上，这个问题的答案位于所有软件设计活动的核心。但是，在讨论这些方法前，恰当的做法是更多考虑实现良好封装的一般原则。内聚和耦合就是用来描述模块间关系的两种度量。

内聚关心的是模块内部结合得有多好——它的内部强度。Allworth和Zobel（1987）给出了6种测量内聚的度量，包括从最弱的到最强的:

- **巧合的**——模块的元素只是由非常表面的方式连接在一起，例如，在同一个月写的代码。
- **逻辑的**——模块的元素在整体系统上是相关的，而不是根据实际软件相关联的，例如，所有的输出设备驱动程序。
- **时间的**——模块的元素在差不多的时间执行，例如，启动例程。
- **过程的**——模块的元素在程序的同一部分一起使用，例如，用户界面部件。
- **通信的**——模块的元素工作在同一数据结构上，例如，分析输入信号的算法。
- 19 **功能的**——模块的元素一起工作执行单个系统功能，例如，分布式文件系统的提供。

与之相比，耦合用来度量程序模块间的互相依赖性。如果两个模块间传递控制信息，它们就被认为拥有高耦合（或紧密耦合）。不然，如果仅仅是数据的传递则是松散耦合。另一种看待耦合的方法是考虑从一个完整的系统中移走一个模块并用另一个模块来代替它的容易程度。

在所有的设计方法中，好的分解是那种强内聚和松散耦合的分解。这一原则在顺序程序设计和并发程序设计领域中同样成立。

2.3.3 形式化方法

三十多年前，C.A.Petri提出使用库所－变迁网（Place-Transition net）（Brauer，1980）来为并发系统的行为建模。这些网由带标记的有向双向图构成，它有两种类型的结点：S－元素，表示本地原子状态，T－元素，表示变迁。图的弧提供S和T元素间的关联。图上的标记代表S－元素上的记号（token），记号的移动代表程序状态的改变。使用规则去指定一个记号何时和如何可以从一个S－元素经由变迁元素移动到另一个S－元素。Petri网有数学定义并经得起形式化分析。

库所－变迁网具有简单、抽象和图形化这些有用的特性，并提供一个通用框架来分析许多种并发和分布式系统。它们的不足是可能产生大量的且难处理的表示。为了对这些系统进行更简明的建模，提出了谓词－变迁网（Predicate-Transition net）。在这种网中，一个S－元素可以为几个正规S－元素（T－元素也与此类似）和记号建模，这个S－元素最初没有内部结构，可以用数据元组来“着色”。

Petri网的一个替代物是定时自动机和模型检查（Timed Automata and Model Checking）。在这种方法中，并发系统用一组有限状态机表示，使用状态探查技术来研究系统的可能行为。

Petri网和定时自动机代表了一种为并发系统建模的方法。其他严格的方法要求所选实现语言的形式化描述。有了这些公理，就可能建立分析并发系统行为的证明规则。但是，很少有实现语言开发时考虑到形式化描述（occam是一个明显的例外）。通信顺序进程（Communicating

Sequential Processes，CSP）记号系统，就是为规定和分析并发系统建立的。在CSP中，一个进程根据外部事件来描述，外部事件就是该进程同其他进程的通信。进程的历史用踪迹（trace）来表示，踪迹就是有限的事件序列。

可以分析用CSP表示的系统以确定它的行为。特别是，可以检测系统的安全性（*safety*）和活性（*liveness*）。Owicki和Lamport（1982）将这两个概念定义为：

- **安全性**——“不会发生一些坏事”；
- **活性**——“会发生一些好事”。 20

模型检查技术也可用于验证系统的安全性和活性。

尽管实时系统是并发的，但是它们也有时间性要求。因此对它们的分析需要适当的逻辑形式。时态逻辑是对命题和谓词演算的扩展，为了表示与实时相关的特性引入了一些新的算子。典型的算子是：always，sometime，until，since和leads-to。例如，sometime（<>）意味着其后紧跟着的性质在将来的某些时刻将为真——如，<>（y>N）表示最终y将大于N。

许多时态逻辑的应用用于验证已经存在的程序，而不是用于新系统的层次化规格说明和严格开发。人们可能要批评这种逻辑过于全局化且非模块化。为了分析系统的任何一部分，它通常都需要拥有完整的程序。为了克服这些困难，可以扩展形式体系以便将它们有效地转换为逻辑命题。Lamport（1983）、Barringer和Kuiper（1983）倡导这种方法。对时态逻辑的进一步改进能使这些时态算子附有时限。因此，如前面的例子，y不仅将大于N，而且将在限定的时间内发生。形式体系RTL（real-time logic，实时逻辑）是时间同一阶谓词逻辑（first-order predicate logic）结合起来的好例子（Jahanian and Mok，1986）。时间性需求的表示和时限调度这些重要问题将在第12章和第13章详细讨论。

2.4 设计方法

众所周知，早些时候大多数实时开发者提倡对象抽象过程，并且确实存在一些形式化技术能用以规定和分析有时间限制的并发系统。然而这些技术还不够成熟，不足以构成“尝试和测试”的设计方法。实时工业宁愿使用适用于所有信息处理系统的结构化方法和软件工程方法。这些方法没有对实时领域给出特别的支持，并且缺乏利用实现语言的全部能力所需的丰富性。

有许多结构化设计方法是以实时系统为目标的：MASCOT、JSD、Yourdon、MOON、OOD、RTSA、HOOD、DARTS、ADARTS、CODARTS、EPOS、MCSE、PAMELA、HRT-HOOD、Octopus等。Gomaa（1994）建议实时设计方法应有四个重要目标，它应该能够：

1）用并发任务构造系统

2）通过信息隐藏支持可重用部件开发

3）使用有限状态机定义行为外观

4）分析设计的性能，以确定它的实时特性

上面的方法中没有一个满足所有这些目标。例如，RTSA在任务构造和信息隐藏方面较弱， 21
JSD不支持信息隐藏或有限状态机。此外，这些方法中很少有直接支持普通的硬实时抽象（例如，周期性的和偶发的活动），这些抽象可在大多数硬实时系统中找到。甚至更少有方法采用一个保证能对最终系统进行有效时间性分析的计算模型。因此使用这些方法易于出错，并可能形成不能分析实时特性的系统。

PAMELA（Process Abstraction Method for Embedded Large Application，嵌入式大型应用的进程抽象方法）(Cherry，1986)、HOOD V4（Heitz，1995）和HRT-HOOD（Burns and Wellings，1994）或许是最符合上述需求的三种方法。PAMELA允许循环活动、状态机和中断处理器的图形表示。但是，该记号系统没有得到资源抽象的支持，因此设计不一定经得起时间性分析。HRT-HOOD支持四个需求中的三个，但是它缺乏使用有限状态机定义行为外观的能力。而且，因为仅仅是基于对象的，它对部件重用的支持较弱。HOOD V4试图支克服HOOD V3的缺点，同时结合HRT-HOOD的优点。但是，它对可重用体系结构设计的支持较弱，并且不能保证设计的性能分析。

EPOS（Lauber，1989）是另一种支持整个实时系统生命期的方法。它提供了三种规格说明语言：一种用于描述顾客需求，一种描述系统规格说明，一种用于描述项目管理、配置管理和质量保证。像HOOD和HRT-HOOD一样，这种系统规格说明有图形化表示和文本表示。周期性的和偶发的活动都能表示（但当处理这种图表时这些活动并不直接可见），还可以规定时间性需求。并且，在EPOS的环境里已经研究了应用程序的实时行为的早期识别（Lauber，1989），但是这项工作依赖于将系统规格说明做成动画，而不是时间性分析。

这些方法的比较可参看Hull等（1991）、Cooling（1991）、Calvez（1993）和Gomaa（1994）的工作。

如在引言中扼要说过的，大多传统软件开发方法都结合一个生命周期模型，在该模型中识别下列活动：

- 需求规格说明——在这一阶段，产生系统所需的功能性和非功能性行为的权威规格说明；
- 体系结构设计——在这一阶段，开发系统的顶层描述；
- 详细设计——在这一阶段，规定完整的系统设计；
- 编码——在这一阶段，实现这个系统。
- 测试——在这一阶段，测试系统的功效。

对于硬实时系统，这种模型的明显缺点是时间性问题将仅仅在测试期间才被识别，甚至
22 更糟，在软件部署使用以后才发现。

通常结构化设计方法使用一种图，在图上带标注的箭头表示通过系统的数据流，箭头指向的结点表示对数据进行转换的点（也就是处理）。在以下几节，将简要概述JSD和Mascot3，这两个技术在实时领域都被广泛使用，然后介绍Ada专有的方法HRT-HOOD。HRT-HOOD是专门针对硬实时系统的方法，第17章将使用它。最后研究UML。

2.4.1 JSD

Jackson的系统开发方法（Jackson，1975）使用一套精确的记号进行规格说明（顶层设计）和实现（详细设计）。有趣的是，实现并不仅仅是规格说明的详细再描述，而是对最初的规格说明应用转换后的结果，其目的是提高效率。

JSD图由一些进程和一个连接网络组成。有三种进程：

- 输入进程，检测环境中的活动，并将它们传送给系统。
- 输出进程，将系统响应传送给环境。
- 内部进程。

进程可以用两种不同方法连接:

- 通过被缓冲的异步数据流连接
- 通过状态向量（或审视）连接

状态向量连接允许一个进程不需要进行通信就看到另一进程的内部状态。

JSD图给出系统的体系结构。JSD图中还要加入表述系统中流动的信息的适当数据结构，以及加入到每个处理中的活动的细节。但是，JSD没有一个标准方式加入时间性约束，因此，必须在图中增加非形式化的标注。

当然，JSD提供的是表达设计的设施——它不是为你做设计。设计不可避免地要结合人的经验和创造性（如同编程）。人们常说结构化和形式化方法抑制创造力，并不是这样的。这些技术提供的是用于表达设计的易于理解的记号，检查创造性得到体现的技术，即检查设计是否符合规格说明、软件是否实现了设计。

用JSD设计的焦点是数据流。因此，“好”的设计应该并入这个自然流。前面已经讨论了导致良好分解的因素。这些正是影响JSD设计（以及所有其他设计方法）的问题。进程自然地 23
分为少数几个不同的类型，针对这些类型的自顶向下的设计方法将使设计易于实现。

以数据流为中心的另一优点是时间性约束常常被表示为流经系统的数据的属性。中断产生控制信号，本质上说，控制信号是中断的转换。这些转换在进程内进行，它们要占用时间。进程的合适选择将提供可调度的、明显的时限（尽管不能直接检查实际的可调度性）。

完成设计之后，必须用系统化的方式来完成实现。在实现语言中有基于消息的并发模型时，这项工作要容易得多，因为设计过程和缓冲的数据流都能作为程序进程编码。但是，这能导致进程增生和非常低效的实现。为对付这种情况，可采用两种方法:

- 转换设计使得只需要较少的进程。
- 得到过量进程的程序，并转换它以减少并发对象的数目。

大多数语言不采用转换技术（occam2又是一个明显的例外，因为它的语义是形式地定义的）。JSD中提倡称为倒置（inversion）的转换。在这种方法中，设计进程被过程代替，并带有一个单独的调度进程控制这些过程集合的执行；就是说，不是五个进程的一个管线，而是每次当数据项出现的时候，调度进程将调用这五个进程（如果这五个进程是相同的，显然仅仅只有一个过程，它将被调用五次）。这里，时间性约束又引起问题，因为倒置期间时间性约束难于维护。

尽管JSD最初不是用于实时应用，但它已经成功用于一些非常大型的系统。已经可以从JSD导出Ada或occam2的实现，许多代码能自动生成（Lawton and France，1988）。

2.4.2 Mascot3

虽然JSD仅仅最近才用于实时领域，Mascot却是特别为实时软件的设计、构造和执行而开发的。Mascot1出现于20世纪70年代早期，但是很快被Mascot2取代，在80年代又被Mascot3取代。

Mascot3的特色是使用图形化的数据流网络和层次化设计。模块化是这种方法的关键，它具有用于设计、构造、实现和测试的多个可识别的模块。重要的是，Mascot3设计可以用图形化形式也可以用文本形式表示。

除了数据流外，Mascot3描述可以包含: 24

- 子系统
- 通用双向通信数据区（intercommunication data areas，IDA）
- 活动（进程）
- 通道（channels，起缓冲区作用的IDA）
- 信息池（pools，起信息仓库作用的IDA）
- 服务器（同外部硬件设备通信的设计元素）

Mascot3给出了使用通道和信息池所必需的同步机制。子系统可以包含其他元素集，也可进一步包含子系统。

Mascot3设计的实现可以用两种完全不同的方法完成。一种是使用一种适当的并发程序设计语言，另一种是使用标准的运行时的执行程序。在Mascot2中，算法的编码用一种顺序程序设计语言来实现，如Coral66或RTL2（FORTRAN、Pascal、C和Algol 60也在Mascot2中使用），然后这个软件在Mascot运行时的执行程序的控制下工作。并发语言的使用使整个系统的实现可以用一种语言完成，因而明显易于集成。

像Ada这样的语言，支持分解并有基于消息的同步模型（预定义的或可编程的），则明显为实现活动提供更有利的解决方案。在Mascot3中使用Ada正得到越来越多的支持，尽管它们的基本动作模型并不完全兼容（Jackson，1986）。但是应该注意，进程增生的问题在Mascot中仍然存在。

2.4.3 HRT-HOOD

HRT-HOOD（Burns and Wellings，1995）不同于Mascot和JSD，它直接用于硬实时系统的设计。它把设计过程看作是逐步增加的特别规约（commitment）的发展过程（Burns and Lister，1991）。这些规约定义了系统设计的特性，而这些特性是在更详细层次上工作的设计者不能任意修改的。在某个特定层次上没有给出规约的那些设计方面，正是更低层设计必须承担的职责（obligation）的内容。在设计的早期，根据对象定义和关系，也许已经有了系统体系结构的规约。然而，所定义对象的详细行为仍然是进一步的设计与实现必须要满足的职责内容。

对设计求精的过程也就是将职责转换为规约的过程，它常常要服从主要由运行环境所强加的约束。运行环境指的是一系列硬件和软件部件（如处理器、任务分派器、设备驱动程序等），而系统就建立在它们之上。运行环境会给设计强加资源方面的约束（如处理器速度、通信带宽等）和机制的约束（如中断优先级、任务分派、数据加锁等）。在一定程度上运行环境
25 是不可改变的，这些约束是固定的。

职责、规约和约束对任何应用的体系结构设计有着重要的影响。因此，HRT-HOOD定义了体系结构设计的两种活动：

- 逻辑体系结构设计活动
- 物理体系结构设计活动

逻辑体系结构设计使规约具体化，这些规约能够独立于执行环境强加的约束来制定，其根本目的是要满足功能需求（尽管存在的时间性需求，如端点到端点的截止时间，将极大地影响逻辑体系结构分解）。物理体系结构设计则要将这些功能需求和其他约束一起进行考虑，并且包括了非功能需求。物理体系结构形成了一个基础，用以评估一旦进行详细设计和实现，应用

的非功能需求是否得到满足。它致力于解决时间和可依赖性需求，以及必要的可调度性分析，而所有这些都将保证一旦系统被建立起来，它将不仅在值域上而且在时域上都能正确运作。

虽然物理体系结构设计是对逻辑体系结构的精化，但它的开发常常是迭代和并发的过程，在这个过程中，两种模型都会被开发或修改。在物理体系结构设计中包含的分析技术应被尽早使用。我们可以定义初始的资源预算，然后，这个预算随着逻辑体系结构的求精而修改和修正。通过这种方式，可以从需求到部署跟踪“可行”的设计。

关于HRT-HOOD的更多细节将在第17章讨论，在这一章将有一个（按这种设计方法的）案例研究，以说明这一章之前涉及的一些问题。

2.4.4 统一建模语言（UML）

过去十年，开发出了过多面向对象的分析和设计方法（Graham（1994）说有超过60种！）。但是，由于缺乏标准化，妨碍了业界的使用。在实时领域，由于面向对象技术未能对硬实时系统提供足够的支持而使情况更糟糕。这是由于无视正在形成的对实时程序分析的理论支撑，这种理论支撑现在能支持大多数应用需求（见13章）。

最近取得了一个重大成就，对象管理组织（Object Managment Group，OMG）开发出了统一建模语言（Unified Modeling Language，UML）作为标准的面向对象的记号系统。UML是一种用于软件密集系统中的制造品的可视化、制定规格、构造和形成文档的图形化语言（Booch等，1999）。很明显，UML将成为未来十年主流的面向对象分析和设计记号系统。甚至其他面向对象的方法，如Fusion（Coleman等，1994），正在转变为这个新的记号系统。 26

在UML中，适合为实时嵌入式系统建模的主要特征包括（Douglass，1999）：

• 对象模型（包含数据属性、状态、行为、标识和职责）——能捕获系统结构；
• 用例（use case）场景——能从系统响应用户输入中识别出关键输出；
• 行为建模——用有限状态机（状态图）促进对系统行为进行动态建模；
• 包装——提供将建模元素按组组织起来的机制；
• 并发、通信和同步的表示——为现实世界实体建模；
• 物理拓扑模型——使用部署图显示组成系统的设备和处理器；
• 支持面向对象模式和框架——可以表示常见问题的通用解决方案。

但是，像它的名字所暗示的，UML提供的是一种语言而不是方法。因此，有必要为设计过程增加一些记号。Douglass（1999）提出将ROPES（Rapid Object-Orientented Process for Embedded Systems，嵌入式系统的快速面向对象过程）同UML一起使用。ROPES基于交互式生命周期，面向快速原型生成。这些活动包括：分析、设计、实现和测试。同其他UML过程一样（Jacobson等，1999），ROPES是由用例驱动的。一组用例按优先级、风险、共同性而被识别和排序。然后将它们用于产生系统原型（可能使用自动代码生成设施）。

针对UML的当前工作（Selic等，2000）正在研究如何为使用UML剖面为实时系统建模定义标准模式。要解决的问题包括：如何为时间和其他资源建模，如何预测设计的性能（使用可调度性分析）。

2.5 实现

在顶层需求规格说明和可执行机器代码之间的一个重要台阶是编程语言。用于实时系统的实现语言的开发是本书的中心主题。语言设计依然是一个非常活跃的研究领域。虽然系统

设计应该自然地通往实现，但是大多数现代语言的表达能力同当前的设计方法学不匹配。只
27 有理解了在实现阶段能做些什么，才能采取适当的设计方法。

有三类编程语言正用于或曾经用于实时系统的开发。它们是汇编语言、顺序系统实现语言和高级并发语言。

2.5.1 汇编语言

最初，大多数实时系统用嵌入式计算机的汇编语言编程。这主要是因为大部分微型机不能很好地支持高级编程语言，而且汇编语言编程看来也是惟一能访问硬件资源的高效实现的方法。

使用汇编语言的主要问题是它们面向机器而不是面向问题。程序员可能被与算法无关的细节所妨碍，结果算法本身变得难于理解。这使开发成本很高，而且在发现错误或要增强功能的时候，要修改程序很困难。

出现进一步的困难是因为程序不能从一种机器移植到另一种上，程序必须重写。而且如果需要员工在另一种机器上工作，就必须重新培训。

2.5.2 顺序系统实现语言

由于计算机功能变得更强，编程语言更成熟，以及编译技术的进步，所以用高级语言写实时软件的优点超过了缺点。由于缺乏像FORTRAN这样的语言，特别为嵌入式编程开发了新的语言。例如，在美国空军中通常使用Jovial。在英国，国防部将Coral 66标准化，大型工业企业如ICI（英国化学工业公司）将RTL/2标准化了。最近，C和C++编程语言变得流行起来。

所有这些语言有一个共同点——它们是顺序的。它们为实时控制和可靠性提供的设施也较弱。由于这些缺点，它们常常需要依靠操作系统的支持和嵌入汇编代码。

2.5.3 高级并发编程语言

尽管应用定制语言（如用于嵌入式计算机应用的顺序系统实现语言，用于数据处理应用的COBOL和用于科学和工程应用的FORTRAN）使用得越来越多，但是，在20世纪70年代，
28 由于基于计算机的系统变得更大更复杂，计算机软件的生产也逐步变得更困难。

通常把这些问题称为*软件危机*。人们已经认识到这种危机的几种征兆（Booch，1986）：

- 响应性——已经自动化了的生产系统常常不能满足用户需要；
- 可靠性——软件不可靠，常常不能按规格说明执行；
- 费用——很少能预测出软件开发费用；
- 可修改性——软件维护繁杂、昂贵和易出错；
- 及时——软件常常延迟交付；
- 可移植性——在一个系统上使用的软件很少能用于另一个系统；
- 效率——软件开发工作不能优化使用涉及的资源。

软件危机影响的最好例证也许就是美国国防部（DoD）想为它们所有的应用寻找一种公共高级编程语言。因为在20世纪70年代，DoD注意到硬件价格开始逐步下降，而嵌入式软件的费用上升。他们估计，1973年，花在软件上的费用达到30亿美元。关于编程语言的调查显示，在DoD的嵌入式计算机应用中至少使用了450种通用编程语言和不兼容的专门语言。1976年他们用一组要求对现存语言进行评估，评估得出四个主要结论（Whitaker，1978）：

- 没有一个现存语言满足要求。

- 单个语言是理想的目标。
- 语言设计的当前水平能满足这些要求。
- 应选择一个合适语言作为开发新语言的基础，建议用Pascal、PL/I和Algol 68。

结果是在1983年诞生了一个新的语言Ada。1995年，Ada语言被修订以反映10年来的使用经验和编程语言设计的进步。

虽然自从20世纪70年代初期以来，Ada语言就主导了嵌入式计算机编程语言的研究，但是其他的语言也出现了，如Modula-1（Wirth，1977b），它由Wirth开发，用于机器设备编程，它的目的是

> *战胜汇编语言，或者至少是有力地攻击它。*

也许在这一领域最成功的语言是C（Kernighan and Ritchie，1978）。的确，这种语言无可争辩地成为当今世界最流行的编程语言。 29

许多从实现和使用Modula中得到的经验反馈到Modula-2（Wirth，1983），这是一个更通用的系统实现语言。C语言产生了许多语言变种，最重要的是C++，它直接支持面向对象程序设计。其他新语言包括：PEARL，在德国，它广泛用于过程控制应用；Mesa（Xerox 公司，1985），被Xerox用于办公自动化设备；CHILL（CCITT，1980），它是为响应CCITT的需求开发的，用于电信应用编程。甚至有Basic的实时版本，尽管缺乏许多高级语言的常规特征（如用户自定义类型），但它提供了并发编程设施和其他与实时有关的特征。

随着互联网的到来，Java语言变得流行起来。虽然最初不适合实时编程，最近，已经为产生Java的实时版本进行了大量工作（US National Institute of Standards and Technology，1999；Bollella等，2000；J Consortium，2000）。

Modula、Modula-2、PEARL、Mesa、CHILL、Java和Ada都是高级并发编程语言，它们包含的功能以帮助嵌入式计算机系统开发为目标。与此相比，occam语言是一个小得多的语言。虽然有常规控制结构和非递归过程，但是它没有对编写大型复杂系统提供实际支持。occam的开发同传输机（transputer）紧密相关（May and Shepherd，1984）（传输机有一种带有片存储器和链路控制器的处理器，链路控制器使得多传输机系统易于构造）。occam在松散耦合的分布式嵌入应用领域扮演重要角色。但是，最近它已经不流行了。

2.5.4 通用语言设计标准

虽然实时语言可能主要是为满足嵌入式计算机系统编程的需求而设计的，但是它很少被限制在这一领域。大多数实时语言也用作应用的通用系统实现语言，例如编译器和操作系统。

Young（1982）列出了下列六条标准作为实时语言设计的基础（它们可能相互冲突）：安全性、可读性、灵活性、简明性、可移植性和高效性。Ada的最初需求也包括一个类似的清单。

1. 安全性

语言设计的安全性表示程序错误能否被编译器或语言运行时支持系统自动检测出来的程度的度量。能被语言系统检测出来的错误的类型和数量有明显的限制，例如，编程者的逻辑错误不能被自动检测出来。所以，安全的语言必须有很好的结构和可读性，以便能容易地发现错误。 30

安全性的好处包括：

- 开发程序时可以更早发现错误——可全面降低费用。

• 编译时检查不会增加运行时开销——相对于编译，程序更多时间是在执行。

安全性的缺点是它可能导致更复杂的语言，增加编译时间和使编译器更加复杂。

2. 可读性

语言的可读性依赖于多种因素，包括：适当选择关键字、定义类型的能力和程序模块化的设施。Young指出：

> 旨在提供足够清晰的语言记号系统，使得只需要读程序就能容易地理解有关具体程序的操作的主要概念，不需要求助于辅助的流程图和说明书。

良好可读性的优点包括：

• 降低文档费用
• 增加安全性
• 增加可维护性

主要缺点是它通常增加程序的长度。

3. 灵活性

语言必须足够灵活，它应允许程序员用直观一致的风格来表达所有需要的操作。否则，就像老式的顺序语言一样，程序员将不得不常常求助于操作系统命令或插入机器码来达到预期目标。

4. 简明性

简明性是任何设计都值得实现的目标，从国际空间站到简单计算器的设计。在编程语言
31 中，简明性有下列优点：

• 使产生编译器所需的工作量最小
• 降低程序员培训费用
• 消除由于对语言特征的误解而造成编程错误的可能性

灵活性和简明性也同语言的**表达能力**（表达各种问题的解决方案的能力）和**易用性**（易于使用）相关。

5. 可移植性

在一定程度上，程序应独立于它运行的硬件。Java的主要主张之一就是程序一次编译、到处运行。对于实时系统，这一点很难做到（甚至有了可移植的二进制代码也不行）（Venners，1999；X/Open Company Ltd.，1996），因为任何程序总有一部分涉及到硬件资源的操纵。但是，语言必须有能力将程序中与机器有关的部分同与机器无关的部分隔离。

6. 高效性

在实时系统中，必须保证响应时间，所以语言必须产生高效和可预知运行时间的程序。应避免使用不能预知运行时开销的机制。很明显，效率需求必须同安全性、灵活性和可读性需求相平衡。

2.6 测试

大多数实时系统有很高的可靠性需求，这是实时系统的本性，因此测试显然必须极其严格。测试的全面策略涉及到许多技术，大多数适用于所有软件产品。所以本书假设读者熟悉这些技术。

实时并发程序测试的困难在于大多数难以追踪的系统错误通常是进程间微妙交互的结果。错误也常常与时间相关，并且它们仅在稀有的状态下出现。Murphy定律指出，这些稀有状态也是极为重要的，并且它们仅仅发生在受控系统处于关键状态时。应该强调的是，适当的形式化设计方法不能降低对测试的需要。它们是互补的。

当然，测试并不限于最终装配好的系统。设计时做的分解和程序模块（包括过程）内的构造形成部件测试的自然体系结构。实时系统测试特别重要（和困难）的是不仅要测试在正确环境中行为是否正确，而且要测试在任意不正确环境中系统是否可依赖。所有错误恢复的路径必须演练，并且错误的影响要研究清楚。 32

为帮助复杂的测试活动，一个逼真的测试床是很有吸引力的。对于软件，这种测试环境称为模拟器（simulator）。

模拟器

模拟器是模仿嵌入了实时软件的工程系统的动作的程序。它模拟中断的生成和执行其他实时I/O动作。使用模拟器，“正常”和“不正常”系统行为都能建立起来。甚至在最终的系统已经完成以后，某些特定的错误状态只能用模拟器安全地进行实验。核反应堆的熔毁就是一个很好的例子。

模拟器能使实时系统中预期的事件序列准确地再现。另外，模拟器能以某种方式重复进行实验，而这在实际运行的操作中通常是不可能的。但是，为了如实重现同时发生的动作，也许需要多处理器的模拟器。此外，应该指出，对于非常复杂的应用，也许不可能构造一个合适的模拟器。

虽然模拟器没有高可靠性需求，但是它们在所有其他方面都是实实在在的实时系统。虽然偶尔的错误是可以容忍的，但是它们自身也必须经过彻底的测试。在没有可靠性需求的实时系统中（如飞行模拟器），有一种常用技术就是从共享资源中去掉互斥保护，这可提高效率，但代价是间歇性的失效。利用嵌入式系统自己的处理模型，模拟器的构造也是很容易的。第15章将说明如何将外部设备看成硬件进程，同时将中断映射到可用的同步原语。如果采用这种模型，那么用软件进程去替换硬件进程就相对简单直观了。

尽管如此，模拟器是非同小可的、昂贵的系统。它们甚至可能需要特殊硬件。在NASA的航天飞机项目中，模拟器的费用超过了实时软件本身。这笔钱花得很值，因为在模拟器“飞行”期间发现了许多系统错误。

2.7 原型建造

软件开发的标准“瀑布”方法的步骤是：需求分析、规格说明、设计、实现、集成、测试和交付，这一方法有个根本性的问题，那就是最初的需求或规格说明阶段的错误只能在产品交付以后才发现（或最好情况是在测试期间发现）。在这个后期阶段改正这些错误极为费时，且代价极高。原型建造试图通过给顾客提供一个系统仿制品来更早地找出这些错误。 33

原型的主要目的是帮助开发者确保在需求规格说明中抓住顾客真正想要的是什么。这包括两个方面：

- 需求规格说明正确吗（顾客想要什么）？
- 需求规格说明完整吗（包括了顾客想要的所有东西吗）？

运行原型的好处之一是顾客能实际经历那些以前只能模糊理解的状况。在这种活动中不可避免地要对需求进行修改。特别是，也许会出现新的错误状态和恢复路径。

在规格说明技术不是形式化的地方，原型建造可用于建立信心，即对整体设计一致性的信心。用一个大型的数据流图，原型可以帮助检查所有需要的连接是否恰当，以及所有通过系统的数据流是否确实访问了所需的活动。

为了成本效益好，构造的原型必须比实际软件便宜（便宜很多）。因此，使用同最终系统相同标准的相同设计方法是没有意义的。显然，使用更高级的语言能显著降低费用。APL语言（Iverson，1962）已经普遍用于原型建造，也使用了功能式程序设计语言和逻辑程序设计语言。它们的优点是它们确实能捕获系统重要的功能性行为，而不要求详细的非功能性方面。最近，设计工具已经集成了代码生成功能。但是，这通常不能产生能用于实际实现的高质量代码，但能用于原型建造。这种原型的明显缺点是它们不能演练应用程序的实时方面。为做到这一点需要进行系统模拟。

前面曾指出证明程序在所有操作条件下都满足时间约束是非常困难的。检查设计是否可行的一个方法是尝试模拟软件的行为。通过假设代码结构和处理器执行速度，可以构造模型去模仿实时特征。例如，在仍能满足时间约束的情况下，该模型可以显示出它能处理的最大中断率。

大型实时系统的仿真是昂贵的，它的开发费用很高，并且每个仿真器的运行可能需要相当多的计算资源。通常一个系统要进行几百次这样的仿真。不过，考虑到建立有错误的实时系统可能造成的经济和社会后果，那么，这些费用是值得的。

2.8 人机交互

从广义上说，所有实时系统都要涉及或者影响到人。事实上，大多数涉及到执行软件同
34 一个或多个操作员（人）之间的直接通信。由于人的行为是实时系统中最大的变异源，所以，HCI（Human-Computer Interaction，人机交互）部件的设计是整个设计中最关键的部分之一。有许多差劲的HCI设计的例子，以及由于这种弱点造成的悲惨后果。例如，在三英里岛（Three Mile Island）发生的核事故中，在一系列的紧急事件发生过程中，由于操作员不能应付数量极多的显示信息，做出了一些错误动作，因而发生事故（Kemeny等，1979）。

HCI部件构造的基本设计原则的研究目前非常活跃。在这方面的研究工作停滞多年以后，现在认为HCI是良好工程软件（所有软件）生产的核心问题。第一个重要的问题是模块化问题，HCI的活动应该被隔离开，并应指定定义明确的界面。这些界面自身被构造成为两部分，两部分的功能有很大不同。第一类定义操作员和软件之间传递的对象。第二类必须规定如何向用户展示这些对象和如何从用户抽取这些对象。第一类界面部件的严格定义必须包括判定，它是关于在给定的系统状态下，操作员可以采取的动作。例如，某种命令也许需要权限，或者在系统有某种倾向时，某些命令只能明智地或安全可靠地完成。比如电操纵飞机将忽略使其偏离安全飞行区域的命令。

在所有交互系统中一个特别重要的设计问题是：谁来控制？一个极端是由人直接控制计算机完成特定功能，另一个极端是计算机完全自己执行（尽管它偶尔问操作员一些信息）。显然，大多数实时系统是这两个极端的混合，称为**混合式主动系统**（mixed initiative system）。有时由用户控制对话（如给出新命令），另一些时候由系统控制（抽取必要数据以完成接到的

命令)。

界面部件设计的主要动机是捕获所有由用户引起的在界面控制软件中的出错，而不是应用软件或实时系统其余部分的出错。如果能做到这一点，那么这个应用软件就能假设**完美的用户**，结果，系统的设计和实现就得到简化（Burns，1983）。在这个讨论中“**出错**”(error)这个术语是指无意识的动作。Reason（1979）把它称为“**失误**”(slip)。他将术语“**错误**”(mistake）定义为有意的出错动作。虽然界面可能可以阻止导致危险的错误，但是它不可能将操作参数的合理改变当作错误。适当培训和监督操作员是消除这些错误的惟一途径。

虽然失误是不可避免的（所以必须加以防止)，但是如果第二个界面部件，也就是实际操作员输入/输出操作模块明确定义的话，那么失误发生的频率将明显降低。但是，在这个上下文里，“明确定义”这个术语本身就是难以定义的。I/O模块的规格说明本质上是心理学家的领域。我们必须从最终用户或操作员的模型开始。但是，存在许多不同的模型，Rouse（1981）给出了一个有用的概述。从对这些模型的理解，可以建立设计的原则，其中三条重要的原则是: 35

1）可预测性——应该给用户提供足够的数据，以便他们能从一个命令的知识无歧义地导出该命令可能产生的结果。

2）可交换性——命令参数的顺序不应该影响命令的含义。

3）灵敏性——如果系统有不同的操作模式，则总是显示当前的操作模式。

前两条原则由Dix等提出（1987)。实时系统有额外的困难，因为改变系统状态的参与者不仅仅是操作人员。中断可能改变操作模式，操作员输入的异步本性可能导致下面的状况：当操作员初始化命令时它是正确有效的，在新的操作模式中，同样的命令可能就不能正确执行了。

HCI涉及到的许多工作是屏幕设计，它使得数据显示更没有歧义性，数据输入只需最少次数的击键。但是，还有其他因素涉及到工作站的人类工作环境学和操作员的工作环境这样更广泛的问题。屏幕设计可以用原型进行测试，但是，职业满意度是一种在更长时间间隔才能测量出来的度量，原型系统的测试时间是达不到的。操作员可能每天工作8小时，一周工作5天，年复一年。如果增大工作压力（如空中交通控制)，那么，职业满意度和性能就严格地依赖于以下几点:

- 必须完成的大批相互依赖的任务
- 操作员具有的控制级别
- 操作员对整个系统操作的理解程度
- 允许操作员执行的建设性任务的数量

给操作员提供“正在发生什么”的图示将提高他们对问题的理解。的确，一些过程控制系统可以用图形、画面或电子报表来表示系统行为。操作员可以用数据来试验，以便为可以改进性能的方法建模。很有可能将*决策支持系统*加入到控制循环中，这样操作员就能看到微小的操作参数改变能带来什么效果。这样，生产率就能提高，操作员的工作环境就能改善。它也可能减少错误的发生。

2.9 设计的管理

本章试图概述与实时软件设计有关的一些重要问题。只有规格说明、设计、实现和测试

36 这些活动都高质量地完成才能产生可靠、正确的产品。现在有范围广泛的技术可帮助达到这一质量要求。恰当使用定义明确的高级语言是本书讨论的主要问题。为达到理想的目标，编程和设计两方面都需要进行管理。现代并发程序设计语言在这种管理过程中有重要作用。

达到质量要求的关键在于足够的验证和确认。在设计和实现的所有阶段中，需要有明确定义的过程去保证必要的技术和行动已经正确执行了。在需要高质量保证的地方，定理证明器和模型检验器被用于形式化地验证指定的部件。许多其他结构化活动，如危险性分析、软件故障树分析、失效模式和影响分析、代码审查等都被更广泛地使用。用软件工具来辅助验证和确认的需要日益增加。这些工具扮演着重要角色，它们的使用可能减少人工审查的需要。所以，这些工具本身必须是质量极高的。

但是，支持工具并不能代替训练有素、经验丰富的员工。本书描述的严格的软件工程技术的应用和语言特征的使用确实对实时系统的质量和成本意义重大。这些技术的实践者对社会负有责任，为保证嵌入式系统的安全，他们应理解和应用所需的任何知识。许多死亡是由于软件错误造成的。只有当工业界远离临时的（ad hoc）权宜过程、非正式方法和不适当的低级语言，才有可能在将来不发生类似错误。

小结

本章概述了实时系统设计和实现所涉及的主要阶段。它们通常包括：需求分析、规格说明、系统设计、详细设计、编码和测试。实时系统的高可靠性需求指出，只要可能，就应该采用严格的方法。

为了管理复杂实时系统的开发，要求适当地应用分解、封装和抽象。层次化设计方法隔离子部件或模块，这些子部件或模块是基于对象的或基于进程的。两种形式的模块都应是强内聚和松耦合的。

实现是本书关注的主要问题，它必然要用到编程语言。早期的实时语言缺少足够的表达能力去充分地处理这一领域的应用。最近的语言已经尝试加入并发和出错处理设施。这些特
37 征的讨论将在后续各章进行。下列通用标准被认为是实时语言设计的有用基础：

- 安全性
- 可读性
- 灵活性
- 简明性
- 可移植性
- 高效性

无论设计过程是否严格，适当的测试显然都是必须的。为了帮助测试，原型实现、模拟器和仿真器都起重要作用。

本章最后关注重要的人机界面。很长时间以来，人机界面被认为是没有什么科学性的，在没有它的时候，工程原理的系统化应用也取得了显著成就。但是现在的状况正在改变，人们认识到界面是潜在错误的重要来源，通过目前在人机界面方面的研究和开发活动，这些错误能被大大地减少。

在许多方面，本章成了漫无边际的一章。本章介绍了许多问题领域和工程问题，这也许超出了一本书能容纳的范围。通过概述设计过程，为本书其他内容作出了铺垫。下面关注语

言问题和编程活动，读者将能够理解什么是设计的最终产品，并可以判断什么样的方法学、技术和工具是合适的。

相关阅读材料

Booch G., Rumbaugh J. and Jacobson I. (1999) *The Unified Modeling Language User Guide*. Harlow: Addison-Wesley.

Burns A. and Wellings A. J. (1995) *Hard Real-Time HOOD: A Structured Design Method for Hard Real-time Ada Systems*. New York: Elsevier.

Cooling J. E. (1995) *Software Design for Real-Time Systems*. London: International Thompson Computer Press.

Dix A. (1991) *Formal Methods for Interactive Systems*. New York: Academic Press.

Douglass B. P. (1999), *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns*. Harlow: Addison-Wesley.

Gomaa H. (1993) *Software Design Methods for Concurrent and Real-Time Systems*. Reading, MA: Addison-Wesley.

Jacobson I., Booch G. and Rumbaugh J. (1999) The Unified Software Development Process. Harlow: Addison Wesley Longman. 38

Jones C. B. (1990) *Systematic Software Development Using VDM*. New York: Prentice Hall.

Joseph M. (ed) (1996) *Real-Time Systems: Specification, Verification and Analysis*. London: Prentice Hall, 1996.

Levi S. T. and Agrawala A. K. (1990) *Real-Time Systems Design*. New York: McGraw-Hill.

Robinson P. (1992) *Hierarchical Object Oriented Design* New York: Prentice Hall.

Rosen J.-P. (1997) *HOOD: An Industrial Approach to Software Design*. HOOD Technical Group.

Schneider, S. (1999) *Concurrent and Real-Time Systems: The CSP Approach*. New York: Wiley.

Woodcock J. (1994) *Using Standard Z*. Hemel Hempstead: Prentice Hall.

练习

2.1 下列因素对设计方法的选择的影响程度有多大？

(a) 可能的实现语言

(b) 支持工具

(c) 应用的可靠性需求

(d) 员工培训需求

(e) 市场考虑

(f) 以前的经验

(g) 费用

2.2 除了本章给出的标准外，还能用什么因素来评估编程语言？

2.3 在设计过程的什么阶段要考虑最终用户的观点？

2.4 软件工程师应对有错误的实时系统造成的后果负责吗?

2.5 新药在适当地检测和试验完成前不能被使用，实时系统也应服从类似的规定吗？如果系统太复杂，因而不能模拟，那么应该构造它吗?

2.6 Ada语言是惟一用于嵌入式实时系统实现的语言吗?

2.7 UML对硬实时系统的设计和分析的支持达到什么程度?

39 2.8 UML是一个结构化的还是形式化的设计记号系统?

第3章 小型编程

在考虑高级语言的特征时，区分两组特征是很有用的：一组辅助分解过程，另一组便于良好定义部件的编程。这两组特征被描述为：

- 支持大型编程
- 支持小型编程

小型编程是一项已充分理解的活动，在本章中将在概述Ada 95（本书中称为Ada）、Java 1.2（后面称为Java）、ANSI C和occam2语言的背景里讨论。第4章讨论大型编程，并且要论述管理复杂系统这个更加成问题的议题。

3.1 Ada、Java、C和occam2概述

在一本关于实时系统及其编程语言的书中，不可能对在此领域使用的所有语言给予详细描述。所以，决定只详细考察四种高级编程语言。Ada是重要的，因为它在安全至上系统中的使用在增加；Java已经成为基于Internet应用编程的事实上的标准语言。C（及其派生物C++）可能是当前在用的最大众化的编程语言，而occam2可能是体现CSP形式化体系的最接近的通用语言。occam2也是专门为多计算机执行设计的语言，它在实时领域中的应用在增加并有其重要性。然而，其他语言的专门特征也会在适当的地方予以研究。 41

这里给出的概述本身假设读者有类Pascal语言的知识。本章会给出每种语言的足够细节以理解书中稍后给出的程序例子。至于每种语言的全面描述，读者必须去阅读“相关阅读材料”中针对每种语言的专门书籍。

3.2 词法约定

程序编写一次，却要被阅读多次；因此语言语法的词法风格应当更适合读者，而不是编写者。对可读性的一个简单帮助是使用有意义的名字。语言不应当限制标识符的长度，Ada、Java、和C不这么做，occam2也不这么做。名字的形式可通过使用分隔符得以改进。Ada、Java和C允许在标识符中有“_”，occam2可包括“.”（这是一个不好的选择，因为“.”在语言中经常用于表示子成分，例如记录的域）。如果语言不支持分隔符，推荐混合使用大写字母和小写字母的技术。虽然Java允许使用“_”，却正在形成混用大写字母和小写字母的风格。下面是标识符的例子。

```
Example_Name_In_Ada
exampleNameInJava
example_name_in_C
example.name.in.occam2
```

不良词法约定的经典例子来自FORTRAN。该语言允许将空格字符用作分隔符。结果，程序中一个简单的打字错误（将“’”打成了“.”）据说导致美国的火星探测器跑不见了！原打算的一行是:

```
DO 20  I = 1, 100
```

这是一个循环构造（I取值1到100，循环标号为20），却被编译成了赋值（赋值运算符是“=”）:

```
DO 20 I = 1.100
```

名字中的空格又可被忽略，就成了:

```
DO20I = 1.100
```

FORTRAN中的变量是不必定义的，以“D”打头的标识符被认为是实数，而1.100是一个实数!

42 不用奇怪，所有现代语言要求显式定义变量。

3.3 整体风格

这里概述的四种语言在某种程度上都是块结构的。Ada中的块语句由三部分组成:

1）这个块的局部对象的声明（如果没有这种对象，该声明部分可略去）;

2）语句序列;

3）一组异常处理程序（若为空这个部分也可略去）。

这种块语句的模式如下:

```
declare
  声明部分
begin
  语句序列
exception
  异常处理程序
end;
```

异常处理程序捕获在块语句中出现的错误。在第6章中详细讨论它们。

Ada块语句可放置在程序中任何可以写普通语句的地方。因此可以分级地使用它们，并支持在程序单元里面的分解。下面的简单例子说明如何引进一个新的整型变量Temp，以交换两个整型变量A和B的值。注意，Ada中的注解从双连字符开始直到那一行的末尾。

```
declare
  Temp: Integer:= A;    -- 给临时变量赋以初值
begin
  A:= B;                -- :=是赋值运算符
  B:= Temp;
end;                    -- 无异常部分
```

在C和Java中，块（或复合语句）是用一个{和一个}分界，有下面的结构:

```
{
  声明部分
  语句序列
}
```

像Ada一样，Java在块结尾可以有异常处理程序，如果这个块被标记为“try”块的话（见第6章）。每个语句自身可以是复合语句。交换两个整型变量值的代码会是这样: 43

```
{
  int temp = A; /* 声明和初始化 */
  /* 注意，在C和Java (和occam2) 中，类型名先出现 */
  /* 而在Ada中，类型名在变量名后面 */

  A = B;
  B = temp;
}
```

注意C和Java中的赋值运算符是“=”，注解是用“/*”和“*/”分隔。Java还允许注解中用“//”结束一行。在本书中，“//”用于Java注解，这样能易于同C区分。

在考察该程序的occam2等价表示之前，必须对此语言给出几个介绍性的评论。Ada和Java是可以写并发程序的语言。C是顺序语言，它可以同操作系统一起使用以创建并发进程。相反，occam2中的所有程序是并发的。在Ada、Java或C中的语句序列在occam2中是进程序列。这种区别在比较四种语言的并发模型的特性的时候是一个基本区别（见第7、8、9章），但它不影响对语言基本要素的理解。

occam2像Ada一样，是一个完全的块结构语言。任何进程的前面都可以声明那个进程中要使用的对象。为交换两个整型（occam2中的`INT`）变量，需要`SEQ`结构，用以指明跟着它的赋值必须顺序执行:

```
INT temp:   -- 声明用冒号结束
SEQ
    temp := A
    A := B
    B := temp
```

有趣的是注意四种语言将动作分隔的不同方式。与Pascal不同，Ada使用分号作为语句终止符（所以在`B:= Temp;`末尾有一个分号）。C和Java也将分号用作语句终止符（但在复合语句之后没有分号，虽然Java容许这样做）。occam2完全不用分号！它要求每个动作（进程）在一个单独的行上。此外，缩进的使用在Ada、Java和C中只是（虽然很重要）为了提高可读性，在occam2中却是语法上很重要的东西。上面代码中的三个赋值必须从`SEQ`的`Q`的那一列开始。

3.4 数据类型

同所有高级语言一样，Ada、Java、C和occam2要求程序去处理一些对象，这些对象已经将它们的实际硬件实现抽象掉。程序员不需要关心他们的程序所处理实体的表示或者位置。此外，通过将这些实体分成不同类型，编译程序能够检查出不一致的使用，因而提高同使用语言相关的安全性。 44

Ada允许声明常量、类型、变量、子程序（过程和函数）和包。子程序在本章稍后考虑，

而包在第4章描述。常量、类型和变量的使用类似于Pascal，但它们可以以任意次序出现，只有对象是先声明后引用。类似地，C和Java允许定义常量、类型、变量和函数。Java还允许定义类和包——也见第4章。同这些语言相比较，occam2的类型模型比较受限制，特别是不允许用户定义的类型。

3.4.1 离散类型

表3-1列出了四种语言支持的预定义离散类型。

表3-1 离散类型

Ada	Java	C	occam2
`Integer`	`int`	`int`	`INT`
	`short`	`short`	`INT16`
	`long`	`long`	`INT32`
			`INT64`
	`byte`		`BYTE`
`Boolean`	`boolean`		`BOOL`
`Character`		`char`	
45 `Wide_Character`	`char`	`wchar_t`	

所有常用运算符对这些类型都是可用的。注意，Java的`char`类型支持Unicode，所以等价于Ada的`Wide_Character`或C的`wchar_t`。

Ada、Java和occam2是强类型的（即赋值和表达式必须包含同一类型的对象），但支持显式类型转换。C语言在类型上不安全。例如，整型值可以被赋给`short`类型，而不用显式的类型转换。Java只在不丢失信息时允许转换，例如从`integer`到`long`，反过来则不行。

Ada和C都允许基本整数类型是有符号的或无符号的。默认是有符号的，但可创建无符号（或模）类型。虽然语言并不要求，但是Ada实现可以支持`Short_Integer`和`Long_Integer`。

除这些预定义类型外，Ada和C允许定义枚举类型。occam2和Java都不提供这种功能，虽然已经有建议把它加到Java语言中去（Cairns，1999）。在C中，枚举的常量必须是惟一定义的，且实际上比整数常量定义多一点；然而，Ada模型（通过允许名字重载），没有这样受限制。两种语言都提供处理这些枚举类型对象的手段；C是通过标准的整数运算，Ada则是通过使用属性（Ada使用属性给出类型和对象的信息）。下面的例子说明这几点。

```
/*C */
{
  typedef enum {xplane, yplane, zplane} dimension;
  /* typedef 为一个新类型引进新名字，  */
  /* enum 指出它是一个枚举类型； */
  /* dimension 是类型的新名字*/

  dimension line, force;
  line = xplane;
  force = line + 1;
  /* force现在的值是 yplane, 因为编译器生成内部整型字面量 */
  /* xplane = 0, yplane = 1, zplane =2*/
}

-- Ada
```

```
type Dimension is (Xplane, Yplane, Zplane);
type Map is (Xplane, Yplane);
Line, Force : Dimension;
Grid : Map;
begin
  Line := Xplane;
  Force := Dimension' succ (Xplane);
  -- Force现在有值Yplane，这同实现技术无关

  Grid := Yplane;  -- 名字Yplane无歧义性，因为Grid是map类型的
  Grid := Line;  -- 不合法，类型冲突
end;
```

Ada支持的另一功能是使用子范围或子类型，以限制（特定基类型的）对象的值。这使得在程序中的对象和应用领域中那个对象的合理取值之间建立更紧密的联系。

```
-- Ada
subtype Surface is Dimension range Xplane ..Yplane;
```

注意，Ada有正整数和自然数的预定义类型。

重要的是在Ada 和 C中通过定义一个类型是原有类型的新版本，可以复制所有的类型： 46

```
-- 用Ada
type New_Int is new Integer;
type Projection is new Dimension range Xplane . . Yplane;

/* 用C */
typedef int newint;
typedef dimension projection;
```

虽然，（在表达式中）Ada类型及其子类型的对象可以混用，类型及其派生类型的对象却不能混用。它们是不同的类型：

```
-- Ada
D : Dimension;
S : Surface;
P : Projection;
begin
  D := S; -- 合法
  S := D; -- 合法，但如果D有值 "Zplane" 就会发生运行时错误
  P := D; -- 不合法，类型冲突
  P := Projection (D); -- 合法，显式类型转换
end;
```

这种规定（及其使用）显著提高了Ada程序的安全性。在C中，typedef不提供这种级别的安全性。

Java中，新类型是通过面向对象的编程设施创建的，这一点在4.4节讨论。

3.4.2 实数

许多实时应用（例如信号处理、仿真和过程控制）需要除了整数运算之外的数值计算设施。这一般需要能够处理实数，虽然所需运算的复杂性在应用之间有很大不同。从根本上说，在高级语言中表示实数值有两种不同方式：

1）浮点

2）标度整数

浮点数是对实数的有限近似，适用于不需要准确结果的计算。浮点数用三个值表示：尾数M、指数E和基数R。它的值的形式是$M \times R^E$。基数是（隐式）实现定义的，通常取值为2。因为尾数的长度是有限制的，这种表示具有有限精度。浮点数及其相应“实际”值的差别是

47 相对于数的大小而言的（说它有**相对误差**）。

标度整数的使用是为了准确的数值计算。标度整数是整数和标度的乘积。通过适当选择标度，可以适合任何值。当要求非整数计算时，标度整数是浮点数的替代物。然而，在编译时标度必须是已知的，如果直到执行时才提供值的标度，就必须使用浮点表示。虽然标度整数提供准确值，可是并非数学领域中的所有值都可以准确表示。例如，1/3就不能看作是有限标度的十进制整数。标度整数和它的“实际”值之差是其**绝对误差**。

标度整数在处理准确数值和使用整数运算方面有其优点（相对于浮点）。浮点运算或者需要特殊硬件（浮点部件），或者需要使数值运算比整数运算慢许多倍的复杂软件。然而，标度整数较难使用，特别是在需要对包含不同标度的值的表达式求值的时候。

通常，各种语言支持单一的浮点类型（通常称为real型），它的精度依赖于实现。标度整数的使用问题在正常情况下是留给用户的（那就是说，程序员必须利用系统定义的整数类型实现标度整数的运算）。

Ada和C将术语`float`用于依赖于实现的“实数”类型。然而，在occam2中没有等价物，这里必须指定二进制的位数。occam2设计者的观点是：与对抽象`real`类型的需要相比，更需要让程序员知道要进行运算的精度。occam2的实数类型是`REAL16`、`REAL32`和`REAL64`。C支持`double`和`long double`，用于特别高的精度。Java提供`float`和`double`，它们分别支持32位和64位IEEE 754浮点值。注意，Java的浮点字面量被自动认为是双倍精度的。所以，它们必须被显式地转换成**`float`**，或者是跟以字母f 。例如：

```
float bodyTemperature = (float) 98.6;
//或者
float bodyTemperature = 98.6f;
```

除了预定义的`Float`类型外，Ada提供了让用户创建不同精度的浮点数和定点数的设施。定点数被实现为标度整数。下面是几个类型定义的例子。为定义浮点类型，需要一个下界一个上界，和一个必要的精度（十进制）：

```
type New_Float is digits 10 range -1.0E18.. 1.0E18
```

此类型的子类型可以限制范围或精度：

48

```
subtype Crude_Float is New_Float digits 2;
subtype Pos_New_Float is New_Float range 0.0..1000.0;
```

精度定义了最低需求，实现可以给出更高的准确度。如果不能实现最低需求，则会产生编译时出错消息。如果没有给出精度，语言要求实现选择一个安全范围。

Ada的定点数消除了必需的标度整数运算符的实现细节，它们是预定义的。为构造定点类型需要范围信息和称为delta的绝对误差界。例如，下面的类型定义给出了0.05或1/20的delta：

```
type Scaled_Int is delta 0.05 range -100.00..100.00
```

为表示所有这些十进制值（－100.00，－99.95，－99.90…99.95，100.00）需要指定的二进制

位数。这容易算出来。离1/20最近（但要小于它）的2的幂是1/32，即2^{-5}。因此，需要5个二进制位提供小数部分。范围－100.00～100.00包括在－128～128里面，它需要8个二进制位（包括一个符号位）。所以，总共需要13个二进制位：

```
sbbbbbbb.fffff
```

其中，s是符号位，b代表一个整数位，f代表一个小数位。显然，这个定点数类型容易在16位体系结构上实现。

再次注意，虽然定点类型准确地表示二进制小数的范围，却并非在此正确范围内的所有十进制常数都有准确的表示。例如，十进制数5.1在上面定义的定点类型中将被表示为00000101.00011（二进制）或5.09375（十进制）。

3.4.3 结构化数据类型

能够十分容易地说出四种语言中关于结构化数据类型的规定。occam2和Java支持数组，Ada和C支持数组和记录。下例先说明数组：

```
-- occam2
INT Max IS 10:   -- occam2中的常量定义
[Max]REAL32 Reading:      -- Reading是一个有10个元素的数组,
                          -- Reading[0] .. Reading[9]
[Max] [Max]BOOL Switches:    -- 2维数组
```

occam2中的所有数组自第0个元素开始。

```
/* C */
#define MAX 10 /*定义 MAX为 10 */
typedef float reading_t[MAX]; /* 序标是 0 ~ Max-1 */
typedef short int switches_t [MAX] [MAX]; /* C中无布尔量 */

reading_t reading;
switches_t switches;

// Java
static final int max = 10; // 一个常量

float reading[] = new float[max]; // 序标是0 ~ max-1
boolean switches[] [] = new boolean[max] [max];

-- Ada
Max: constant Integer:= 10;
type Reading_T is array (0 .. Max-1) of Float;
Size: constant Integer:= Max - 1;
type Switches_T is array (0 .. Size, 0 .. Size) of Boolean;
Reading: Reading_T;
Switches: Switches_T;
```

49

注意，Ada的数组使用圆括号，而occam2、Java和C使用更传统的方括号。还有，Ada数组可以有任意的开始序标，而在C、Java和occam2却总是0。注意，在Java中，数组经常是用对象表示的。为创建Java中的对象，需要使用分配器（见3.4.4节）。

关于为什么在occam2中没有引入记录，虽然没有什么根本的原因，但它们的省略却指示了语言设计者和实现者的优先顺序。Ada的记录类型十分直截了当：

```
-- Ada
```

```
type Day_T is new Integer range 1 .. 31;
type Month_T is new Integer range 1 .. 12;
type Year_T is new Integer range 1900 .. 2050;
type Date_T is
  record
    Day: Day_T:= 1;
    Month: Month_T:= 1;
    Year: Year_T;
  end record;
```

然而，C中struc的使用却引起混乱，因为它引进了一个类型，却不用typedef给它一个名字（虽然也能使用typedef，并且是推荐的）：

50

```
/* C */
typedef short int day_t;
typedef short int month_t;
typedef int year_t;
struct date_t {
  day_t day;
  month_t month;
  year_t year; };
/* 因为 data_t 不是由 typedef引进的, */
/* 它的名字是 'struct date_t' */

typedef struct {
  day_t day;
  month_t month;
  year_t year; } date2_t;
/* 这里可以使用类型名 'date2_t'
```

在这个C的例子中，这些域是派生整数，而前面的Ada代码有几个不同的带约束的新类型用于这些部分。Ada例子还展示了可为记录的某些域（不必是所有域）赋予初值。两种语言都是用点记号指出单个的部分并允许记录赋值。Ada还支持使用记录聚集值进行完整记录赋值（也可以使用数组聚集）：

```
-- Ada
D: Data
begin
  D.Year:= 1989;   -- 点记号
  -- 初始化后D的值是1-1-1989
  D:= (3, 1, 1953); -- 完整赋值
  D:= (Year => 1974, Day => 4, Month => 7);
     -- 使用指名记号的完整赋值
  ...
end;
```

而C只允许使用静态数据进行完整的记录初始化。

```
/* C */
Struct date_t  D ={1, 1, 1};
```

Ada中指名记号的使用改善了可读性，并消除了位置差错可能引入的错误；例如写（1，3，1953）而不是（3，1，1953）。

Ada还允许为记录类型扩充新的域。这是为了方便面向对象编程，将在4.4节讨论。

Java没有这种记录结构。然而，使用类可以取得同样效果。类的细节在4.4节给出，但现在考虑下面的日期Java类:

```
// Java
class Date
{
  int day, month, year;
}

Date birthday = new Date ();

birthdate.day = 31;
birthdate.month = 1;
birthdate.year = 2000;
```

至于C，不可能为日期的成分值表达范围约束。还要注意，由于“记录”是Java对象，必须用分配器创建Date的对象。对象的初始化可用构造器实现（见4.2节）。 51

3.4.4 动态数据类型和指针

在许多编程情况里，数据对象集合的准确大小或组织情况无法在程序执行前预测。即使Ada和C支持变长数组，灵活和动态的数据结构也只能通过使用引用而非直接的命名实现（如果提供存储分配设施的话）。这就是Java数组和“记录”的情况。

动态数据结构的实现意味着语言的运行时支持系统的可观开销。正是这个原因，occam2没有动态结构。C允许指向已声明的任何对象的指针。下面给出一个链表的例子。

```
...
{
  typedef struct node {
    int value;
    struct node *next; /* 指向内包结构的一个指针*/
  } node_t;

  int V;
  node_t *Ptr;

  Ptr = malloc (sizeof (node_t) );
  Ptr->value = V; /* -> 间接引用 (de-reference) 该指针*/
  Ptr->next = 0;
  ...
}
```

过程malloc是一个动态分配存储器的标准库过程。运算符sizeof返回由编译器分配给struct node_t的存储单元数。

在Ada中，使用访问类型而不是指针（虽然概念是类似的）。

```
type Node;    -- 不完整声明
type Ac is access Node;
type Node is
  record
    Value: Integer;
    Next: Ac;
  end record;
```

```
V: Integer;
Al: Ac;
begin
  Al:= new (Node); -- 构造第一个节点
52  Al.Value:= V; -- 间接引用该访问变量，并指明其成分
  Al.Next:= null; -- 预定义的
  ...
end;
```

上面的程序段说明了（从堆上）动态分配存储区域的“new”的使用。与C对照，Ada的“new”是在语言中定义的运算符；然而，没有清除（dispose）运算符，而是提供了一个通用过程为指定的对象清除分配的存储。这个过程（称为`Unchecked_Deallocation`）并不检查是否有对对象的未完成引用。

Ada和C都不要求支持垃圾回收器。这种遗漏不足为怪，因为垃圾回收器通常带来执行时间中巨大的、不可预测的开销。这种开销对于实时系统来说是不可接受的（见15.9.1节）。

指针还可指向静态对象或栈上的对象。接下来的例子说明C是怎样许可静态类型指针和怎样让程序员进行指针运算的：

```
{
  typedef date_t events_t[MAX], *next_event_t;

  events_t history;
  next_event_t next_event;

  next_event = &history [0];
  /* 取数组第一个元素的地址*/

  next_event++;
  /* 增加指针 next_event;        */
  /* 在指针上增加 date记录的大小, */
  /* 所以next_event指向数组的下一个元素*/
}
```

C方法的缺点是指针所指对象以后可能超出作用域。这就是所谓**悬空指针**问题（dangling pointer problem）。Ada用别名类型（aliased type）为这种指针提供了安全解决办法。只有别名类型才可被引用。

```
Object : aliased Some_Type;
  -- 使用别名指出，它可能由访问类型引用

type General_Ptr is access all Some_Type;
  -- access all 指出，这个类型的访问变量既可指向静态对象，也可指向动态对象

Gp : General_Ptr := Object' Access;
53 -- 将对象引用赋给Gp
```

Ada访问类型定义的最后一种形式允许给访问类型的使用加上只读限制：

```
Object1 : aliased Some_Type;
Object2 : aliased constant Some_Type := ...;

type General_Ptr is access constant Some_Type;

Gp1 : General_Ptr := Object1' Access;
-- Gp1 现在只能读Object1的值
```

```
Gp2 : General_Ptr := Object2' Access;
-- Gp2 是一个对常量的引用
```

同Ada和C相反，Java中的所有对象都是对包含数据的实在对象的引用，所以不需要额外的访问或指针类型。此外，不像Ada，它不需要Node类型的前导声明:

```
// Java
{
  class Node
  {
    int value;
    Node next;
  }

  Node Ref = new Node ();

}
```

注意，由于将对象表示成引用值，在Java中比较两个对象就成了比较它们的指针，而不是它们的值。所以

```
Node Ref1 = new Node ();
Node Ref2 = new Node ();

...

if (Ref1 == Ref2) { ... }
```

将比较对象的位置，而不是对象所封装的值（这时，是域value和next的值）。值的比较需要一个类，它实现显式的compareTo方法。当然，对于Ada的访问变量这个结论也成立。然而，Ada允许访问变量有一个.all后缀，以指向对象的数据，而非其引用。

对象赋值也出现类似的情况。在Java中，赋值运算符是赋给指针。为创建对象的真正副本需要一个提供clone（克隆）对象方法的类。 54

3.4.5 文件

Ada、Java、C和occam2都不提供Pascal那样的文件类型构造器。相反，每个语言都允许通过库来支持文件。例如，Ada要求所有编译程序提供顺序访问文件和直接访问文件。

3.5 控制结构

虽然在支持的数据结构上有若干差异（尤其在occam2和其他语言之间），关于所要求的控制结构却有更接近的一致。随着编程语言从机器代码经过汇编语言进步到较高级的语言，控制指令也演变成控制语句。在顺序编程语言所需的控制抽象方面有着共同的一致。

这些控制抽象可分为三个类别：顺序、判断和循环。本书将依次研究它们。语言并发部分所必需的控制结构将在第7、8、9章研究。

3.5.1 顺序结构

语句的顺序执行是（非并发）编程语言的正常活动。大多数语言，包括Ada、Java和C在内，隐含地要求顺序执行，不提供专门的控制结构。在Ada和Java（以及C）中块语句的定义就意味着，在“开始”（{）和“结束”（}）之间有一个语句序列。要求按这个顺序执行。

在occam2中，语句（在occam2中称为进程）的正常执行是并发的，这是十分合理的。所

以，有必要显式地说明一组动作一定要遵循给定的顺序。这是通过使用早先说明的SEQ构造做到的。例如:

```
SEQ
  动作1
  动作2
  .
  .
  .
```

如果在一个特定环境中序列是空的，Ada要求显式地使用null语句:

```
begin   -- Ada
  null;
end;
```

55 Java和C允许空的块:

```
{ /* Java/C */
}
```

而occam2用SKIP进程隐含无动作:

```
SEQ
  SKIP  -- occam2
```

或者就只是

```
SKIP
```

空动作的另一个极端是导致序列不再进一步做任何事。Java和C提供一个叫做exit的专用预定义过程，使整个程序终止。occam2的STOP进程有类似效果。Ada没有等价的原语，但可编制异常，使得有相同结果（见第6章）或终止主程序。在所有四种语言中，过早终止程序的严厉动作只用于响应检测到不能被修复的出错状态。在把受控的系统带到安全状态之后，程序除了终止之外不执行进一步的有用动作。

3.5.2 判断结构

判断结构提供自程序序列的某一点到那个序列中一个稍后点的执行路径的选择。选取哪条路径依赖于相关数据对象的当前值。判断控制结构的重要性质是所有路径最终回到一起。有了这种抽象控制结构，就不再需要使用goto语句。goto语句经常导致程序难以测试、难以阅读和难以维护。Java和occam2没有goto语句。Ada和C有goto语句，但应少用。

判断结构的最常见形式是if语句。虽然对这一结构的要求是明确的，诸如Algol-60的较早期语言却由于不良语法形式而引起混乱。尤其是嵌套的if构造，不清楚跟着而来的else是对哪一个if的。令人遗憾的是C依然蒙受此类问题之苦，而Ada和occam2有清楚的无歧义结构。为了说明，研究一个简单的问题：看看B/A＞10是否成立——在此之前先检查以确保A不等于零。Ada的解决方案如下面所示，虽然这段代码有缺点：即如果A＝0，布尔变量High没有被赋值，但程序段含义是明确的，这归因于使用了显式的**end if**记号:

```
if A /= 0 then
  if B/A > 10 then
    High := True;
  else
```

```
      High := False;
    end if;
  end if;
```

Ada还提供了两种短路径控制形式**and then**和**or else**，它们能使上面的代码具有更简明的表示。 [56]

C结构不需要显式的“end”:

```
  if (A !=0)
    if (B/A > 10) high = 1;
    else high = 0;
```

然而，如果将if语句括起来，结构会清楚很多:

```
  if (A !=0) {
    if (B/A > 10) {
      high = 1;
    }
    else {
      high = 0;
    }
  }
```

Java沿用C的结构；然而，通过不允许使用歧义性语句避免了跟随的else问题。

occam2的IF同其他三种语言风格上不太一样，虽然功能是相同的。在下面的大纲中，令B1..Bn是布尔表达式，A1..An是动作:

```
  IF
    B1
      A1
    B2
      A2
    .
    .
   Bn
      An
```

首先，重要的是记住occam2程序的布局是有语法意义的。布尔表达式在单独一行上（自IF缩进两个空格），动作也是这样（再缩进两个空格）。像C一样，不用“then”记号。在这个IF构造的执行中，布尔表达式B1被求值。如果其值为TRUE，则执行A1，并完成了此IF的动作。然而，如果B1为FALSE，则B2被求值。所有的布尔表达式被求值，直到发现一个为TRUE的表达式，然后执行相关联的动作。如果没有一个布尔表达式为TRUE，那么这是一种出错状态，IF构造的行为就像上一节讨论过的STOP一样。

使用这种IF语句形式，不需要明显的ELSE部分；布尔表达式TRUE作为最后的检验一定会进行，如果所有其他选择都失败的话。因此用occam2描述b/a>10的例子，是这个样子: [57]

```
  IF
    a /= 0
      IF
        b/a >10
          high := TRUE
        TRUE
```

```
        high := FALSE
  TRUE        -- 最后这两行是需要的，
    SKIP      -- 这样使得当a为0时，IF不会变成STOP
```

为给出重要的“if”构造的另一个例子说明，研究一个多路分支。下面的例子（首先用Ada）是求出一个正整数变量Number的数字位数。假定最多为5位。

```
if Number < 10 then
  Num_Digits := 1;
else
  if Number < 100 then
    Num_Digits := 2;
  else
    if Number < 1000 then
      Num_Digits := 3;
    else
      if Number < 10000 then
        Num_Digits :=4;
      else
        Num_Digits :=5;
      end if;
    end if;
  end if;
end if;
```

这种形式是十分常见的，它含有**else**部分的嵌套，并在末尾跟随若干**end if**。为消除这种笨拙的结构，Ada提供了**elsif**语句。上面的代码可写得更简明些：

```
-- Ada
if Number < 10 then
  Num_Digits :=1;
elsif Number < 100 then
  Num_Digits :=2;
elsif Number < 1000 then
  Num_Digits :=3;
elsif Number < 10000 then
  Num_Digits :=4;
else
  Num_Digits :=5;
end if;
```

58 用occam2，代码十分清楚：

```
IF
  number < 10
    digits := 1
  number < 100
    digits := 2
  number < 1000
    digits := 3
  number < 10000
    digits := 4
  TRUE
    digits := 5
```

上面的代码是用一系列的二元选择构造的多路分支的例子。一般说来，多路判断可用case（或switch）结构更明确地表示，更高效地实现。四种语言都有这种结构，虽然occam2版本限制多一些。为了说明，考虑一个字符（字节）值“command”，将它用于判断四种可能动作：

```
-- Ada
case Command is
  when 'A'  | 'a'      => Action1;        -- A 或 a
  when 't'             => Action2;
  when 'e'             => Action3;
  when 'x'  .. 'z'     => Action4;        -- x、y 或 z
  when others          => null;           -- 无任何动作
end case;


/* C 和   Java */
switch (command) {
  case 'A'   :
  case 'a'   : action1;  break;               /* A 或 a */
  case 't'   : action2;  break;
  case 'e'   : action3;  break;
  case 'x'   :
  case 'y'   :
  case 'z'   : action4;  break;               /* x、y 或 z */
  default    : break;                         /* 无任何动作 */
}
```

注意，使用Java和C时，有必要插入若干break语句，使得能在识别出所需命令的时候退出switch语句。如果没有的话，控制会继续到下一个选择（就像“A”的情况）。

在occam2中，不支持替代值和范围，所以代码更长：

```
CASE command
  'A'
    动作1
  'a'
    动作1
  't'
    动作2
  'e'
    动作3
  'x'
    动作4
  'y'
    动作4
  'z'
    动作4
  ELSE
    SKIP
```

59

3.5.3 循环结构

循环结构使程序员能够指明一个语句或一个组语句被执行一次以上。构造这种循环有两

种不同的形式:

1）迭代

2）递归

迭代的明显特征是循环的每次执行完成之后下一次循环才能开始。对于递归控制结构，第一个循环被中断，以开始第二个循环，第二个循环又可能中断，以开始第三个循环等等。在某个点上，循环n被允许完成，然后允许循环$n-1$完成，然后是循环$n-2$等等，直到第一个循环终止。递归通常由递归的过程调用实现。这里的注意力集中在迭代上。

迭代有两种形式：一种循环是迭代次数通常在此循环构造执行前就已固定；一种循环是在每次迭代中进行完成检验。前一种通常称为for语句，后者称为while语句。大多数语言的for构造还提供一个计数器，可用于指出当前执行的是哪一次迭代。

下面的示例性代码说明四种语言的for构造，代码为数组A的前十个元素赋以它们在数组中的位置的值:

```
-- Ada
for I in 0 .. 9 loop          -- I 由该循环定义
  A (I) := I;                  -- I 在该循环中是只读的
end loop;                     -- I 在该循环后就出了作用域


/* C and Java  */
for (i =0; i <= 9; i++)  {     /* i 必须在前面定义 */
  A[i]= i;                    /* i 在该循环中可以读/写 */
}                             /* 在该循环后，i的值是有定义的 */
 -- occam2
SEQ i = 0 FOR 10              -- i 由该构造定义
  A[i] := i                   -- i 在该循环中是只读的
                              -- i在该循环后就出了作用域
-- 注意，像在Ada和C的例子中一样，i 的范围是自 0 至 9,
```

注意，Ada和occam2对循环变量的使用有限制。Java和C中这种变量的自由使用可能引起许多错误。由于这个原因，Java还允许在for循环内声明局部变量。

```
// Java
for (int i = 0; I <= max; I++) {
  A[i] = 1;
}
```

除了上述形式之外，Ada允许循环以相反次序执行。

while语句的主要变异是在什么地方进行退出循环的检验。最常见的形式包括在循环入口处和在以后的每次迭代之前检验:

```
--Ada
while 布尔表达式 loop
  语句
end loop;


/* Java 和C*/
while (表达式) {
```

60

```
  /* 表达式的求值结果为0就终止循环 */
  语句
}
```

```
-- occam2
WHILE布尔表达式
  SEQ
    语句
```

Java还支持一种变体，检验在循环的结尾处出现：

```
do {
  语句序列
} while (表达式);
```

Ada、Java和C还通过允许在循环内的任何一点退出循环（即循环终止）来增加灵活性： 61

```
-- Ada
loop
  .
  .
  exit when <布尔表达式>
  .
  .
end loop;
```

```
/* C和Java */
while (1)  {
  .
  .
  if (表达式) break;
  .
  .
}
```

常见的编程错误是循环要么（在必须终止的时候）不终止，要么在错误的状态终止。幸而现已熟知分析循环结构的形式化方法。这些方法包括定义循环的前置条件和后置条件，终止条件和循环不变量。循环不变量是一语句，它在每次迭代的结尾处为真，而在迭代期间可能不为真。从本质上说，循环分析包括指出前置条件将导致循环终止，以及在终止时循环不变量能证明后置条件被满足。为方便这些形式化方法的使用，不提倡在迭代期间退出循环。只要可能，最好使用标准while构造。

关于循环要说的最后一点是实时系统常常要求循环不终止。预计控制周期将无限运行下去（即直到切断电源）。虽然**while True**能帮助做到这种循环，但它可能是低效的，而且没有抓住无限循环的本质。为此，Ada提供简单的循环结构：

```
-- Ada
loop
  语句序列
end loop;
```

3.6 子程序

即使是在部件和模块的构造工作中，通常也希望进行进一步的分解。这是通过使用过程和函数实现的。它们一起被称为*子程序*。

子程序不仅辅助分解，还代表一种重要形式的抽象。它们使任意复杂的计算能被定义成
62 一个简单的标识符并在以后使用这个标识符来调用它。这使得这种部件能在程序内和程序之间被重用。子程序的这种通用性和用途当然通过使用参数而提高。

3.6.1 参数传递模式和机制

参数是一种通信形式，它是在子程序用户和子程序本身之间传递的数据对象。有多种方式描述这种数据传输的机制。首先，可以研究传输参数的方式。从调用者的视点看，有三种不同的传输模式:

1）传送进子程序的数据。

2）从子程序传送出来的数据。

3）传送进子程序的数据经过改变，然后又从子程序传送出来。

这三种模式经常被称为in、out和in out。

描述传输的第二种机制是考虑绑定子程序形式参数和调用的实际参数。有两种感兴趣的通用方法：参数可以由值绑定或是由引用绑定。由值绑定的参数只是将参数的值传给子程序（经常是通过复制到子程序的存储空间），这种参数不能给调用者返回任何信息。当参数是由引用绑定时，在子程序内对那个参数的任何更新都被认定会影响实际参数的存储位置。

研究参数传递机制的最后一个方式是考察实现使用的方法。编译程序必须满足语言使用模式或绑定表达的语义，但除此之外，如何尽可能高效地实现子程序却是自由的。例如，一个“值传送”的大数组参数不需要复制，如果子程序中没有对数组元素赋值的话。实际数组的单个指针会更有效，但行为上是等价的。类似地，具有引用参数的调用，可以用一个复制进来和复制出去算法实现。

Ada使用参数模式来表达数据是传送给子程序或是从子程序传送出来。例如，考虑一个过程，它返回一个二次方程的实根（如果它们存在的话）。

```
procedure Quadratic (A, B, C : in Float;
                     R1, R2  : out Float;
                     Ok      : out Boolean);
```

in参数（默认的）在子程序内是局部常量——在过程或函数入口处赋给形式参数的值。
63 函数只允许这种模式。在过程内，可以读写out参数。在过程终止处赋给调用参数一个值。“in out”参数的行为像是过程中的变量。在入口处，将值赋给形式参数；在过程终止时，所获得的值被传送回调用（实际）参数。

C按值传递参数（C称参数为**变元**）。如果结果是要返回的，必须使用指针。C和Java都只支持函数，过程被看成无返回值的函数（在函数定义中用void表示）。

```
void quadratic (float A, float B, float C,
                float *R1, float *R2, int *OK) ;
```

注意，没有function这个关键字。还有，即使上面的A、B、C已经复制给了函数，它们在此函数里仍然可被写。然而，任何更新的值**不**被复制回去。

在Java中，基本变元是通过复制传递的。“类”类型的变量是引用变量。所以，当它们被作为变元传递时，它们被复制，但效果是变量通过引用传递。如果不得由Java函数改变变元的值，那么变元应当在函数中被声明为final。

```
public class Roots {
  float R1, R2;
}
boolean quadratic (final float A, final float B, final float C,
                   Roots R);
```

注意在这个Java例子中，布尔标记是经过函数返回值返回的。还有，Java要求方程的根被作为“类”类型传送，因为基本类型（包括float）是通过复制传送的并且没有指针类型。

在occam2中，参数的默认方式是按引用传递，VAL标志用于表示按值传递。重要的是，在过程里面（在occam2中称为PROC），VAL参数的行为像是常量，因此，在Pascal中可能发生的遗漏VAL标志的错误可由编译程序捕获。

```
PROC quadratic (VAL REAL32 A, B, C,
                REAL32 R1, R2, BOOL OK)
  --像C和Java一样，参数分隔符不是分号
```

3.6.2 过程

所有四种语言中的过程体都十分明确，现在通过完成上面所给的“quadratic”定义来说明。所有过程都假设作用域中有一个sqrt函数。 64

```
-- Ada
procedure Quadratic (A, B, C : in Float;
                     R1, R2 : out Float;
                     Ok : out Boolean) is
  Z : Float;
begin
  Z:= B*B - 4.0*A*C;
  if Z < 0.0 or A = 0.0 then
    Ok:= False;
    R1:= 0.0;      -- 任意值
    R2:= 0.0;
    return;        -- 在到达逻辑终点前从过程返回
  end if;
  Ok:= True;
  R1:= (-B + Sqrt (Z) ) / (2.0*A);
  R2:= (-B - Sqrt (Z) ) / (2.0*A);
end Quadratic;
```

```
/* C */
void quadratic (float A, float B, float C, float *R1,
                float *R2, int *OK)
{
  float Z;

  Z= B*B - 4.0*A*C;
  if (Z < 0.0 || A == 0.0) {
```

```
    *OK = 0;
    *R1 = 0.0;      /* 任意值 */
    *R2 = 0.0;
    return;        /*在到达逻辑终点前从过程返回*/
  }
  *OK = 1;
  *R1 = (-B + SQRT (Z) ) / (2.0*A);
  *R2 = (-B - SQRT (Z) ) / (2.0*A);
}


// Java
public class Roots {
  float R1, R2;
}

boolean quadratic (final float A, final float B, final float C, Roots R) {

  // 注意需要到float的显式转换

  float Z;

  Z = (float)  (B*B - 4.0*A*C);
  if (Z < 0.0 || A == 0.0) {
65  R.R1 = 0f;      // 任意值
    R.R2 = 0f;
    return false;
  }
  R.R1 = (float)  ( (-B + Math.sqrt (Z) ) / (2.0*A) );
  R.R2 = (float)  ( (-B - Math.sqrt (Z) ) / (2.0*A) );
  return true;
};


-- occam2
PROC quadratic (VAL REAL32 A, B, C,
                REAL32 R1, R2, BOOL OK)
  REAL32 Z:
  SEQ
    Z:= (B*B) - (4.0* (A*C) ) -- 为完整指明表达式，括号是必须的
    IF
      (Z < 0) OR (A = 0.0)
        SEQ
          OK:= FALSE
          R1:= 0.0              -- 任意值
          R2:= 0.0
      TRUE                      -- occam2中没有return语句
        SEQ
          OK:= TRUE
          R1:= (-B + SQRT (Z) ) / (2.0*A)
          R2:= (-B - SQRT (Z) ) / (2.0*A)
:   -- 冒号是必须的，以指出PROC声明的结束
```

在所有四种语言中，调用这些过程都只要指出过程的名字，在括号中给出适当类型的参数。

除了这些基本特征之外，在Ada中还有改善可读性的额外设施。考虑一个枚举类型Setting和一个刻画10个不同阀的整数类型：

```
type Setting is (Open, Closed);
type Valve is new Integer range 1 .. 10;
```

下列的过程规格说明给出修改一个阀值的子程序：

```
procedure Change_Setting (Valve_Number : Valve;
                          Position : Setting := Closed;
                         );
```

注意，有一个参数给了默认值。对此过程的调用可有几个不同的形式：

```
Change_Setting (6, Open);       -- 常规调用
Change_Setting (3);             -- 使用默认值 'Closed'
Change_Setting (Position => Open,
                Valve_Number => 9); -- 指名记号
Change_Setting (Valve_Number => 4); -- 指名记号和默认值
```

默认值是有用的手段，如果某些参数几乎总是取同一个值的话。指名记号的使用消除位置性错误并提高可读性。 66

递归（和相互递归）过程调用在Ada、Java和C中是允许的。在occam2中不支持它们是由于运行时它们带来的动态开销。在Ada中过程（函数）可以嵌套，而在Java、C或occam2中不行。

3.6.3 函数

Ada支持函数的方式类似于对过程的支持。考虑一个返回两个整数值中的最小者的例子。

```
-- Ada
function Minimum (X, Y : in Integer) return Integer is
begin
  if X > Y then
    return Y;
  else
    return X;
  end if;
end Minimum;


/* C 和 Java */
int minimum (int X, int Y)
{
  if (X > Y) return Y;
  else return X;
}
```

Ada、Java和C允许函数返回任何合法类型的值，包括结构化类型。

函数的误用是程序中许多错误之源，关于函数的黄金规则是它们不应有副作用。表达式应当具有它所说的含义：

```
A := B + F (C)
```

A的值变成B的值加上C通过应用函数F后所得到的值。在上述代码的执行中，应当只有A的值被改变。

在上述表达式中能够以三种方式引入副作用。

1）F在返回一个值时，还会改变C的值。

2）F会改变B的值，所以按从左至右和从右至左求值上述表达式会产生不同的值。

67 3）F会改变D的值，这里D是作用域中的任何其他的变量。

Ada和Java通过只允许函数具有“in”模式参数来限制副作用。当然，因为这些参数可能引用其他对象，副作用还是完全有可能的。然而，occam2更进一步，定义了函数的语义，所以不可能有任何副作用。

像Ada一样，occam2函数的参数是按值传送的。此外，函数体被定义成是VALOF。VALOF是计算一个对象的值（它将由函数返回）所必需的语句序列。VALOF的重要属性是：只有在VALOF里面局部定义的那些变量才可能被改变值。这就禁止了副作用。所以，前面定义的简单最小值函数会有下面的形式：

```
INT FUNCTION minimum (VAL INT X, Y)
  INT Z:  -- Z将是返回值
  VALOF
    IF
      X>Y
        Z:= Y
      TRUE
        Z:= X
  RESULT Z
:
```

在并发语言中，另一种副作用形式是隐藏并发性，这会有许多不好的结果。occam2的VALOF进一步限制在其中不允许任何并发性。

最后，应当注意Java中的函数（和过程）只能在类定义中声明（见4.4.2节）。

3.6.4 子程序指针

Ada和C都允许指向过程和函数的指针。例如，下面的Ada类型声明（Error_Report）定义了指向具有字符串参数的过程的访问变量。然后声明一个过程，它有这种类型的一个参数，并且发出对这个过程的调用，调用时向Operator_Warning过程传送一个指针。

```
type Error_Report is access procedure (Reason: in String);

procedure Operator_Warning (Message: in String) is
begin
  -- 通知出错的操作员
end Operator_Warning;

procedure Complex_Calculation (Error_To : Error_Report; ... ) is
begin
  -- 如果在复杂计算期间检测到错误
  Error_To ("Giving Up");
68 end Complex_Calculation;
...
```

```
  Complex_Calculation (Operator_Warning'Access, ...);
...
```

如果用C语言，等价的出错报告指针是:

```
void (*error_report)  (char* Message);
```

通过使用

```
error_report = operator_warning;
```

可以得到函数的地址。

Java不允许函数（过程）指针，因为它们只能出现在类的上下文中，而类总是通过引用访问的。

3.6.5 插入式展开

虽然使用子程序显然有利于分解、重用和可读性，对某些实时应用来说，实现实际调用的开销却可能高得无法接受。降低这种开销的一个手段是在发出对子程序调用的时候就“插入式”地替换子程序代码。这种技术被称为**插入式展开**（inline expansion），它的好处是依然允许程序员使用子程序，又不引起运行时的开销。

有趣的是，形式化地规定语义的occam2，使用文本替换作为指明过程调用要做的事情的方法。然而，所有四种语言都使实现者能够按他们认为合适的方式处理子程序。对于这一点的惟一例外看来是Ada。在Ada中程序员可通过使用编用`Inline`，请求只要可能就将指明的子程序进行插入式展开。Ada中使用编用向编译程序发出指令——它们不是可执行语句。

小结

Wirth 在他那本有重大影响的著作的标题中表达了著名的格言:

```
"Algorithms +Data Structures = Programs"
(“算法 + 数据结构 = 程序”)
```

也许，因为非常大的程序呈现的困难是显而易见的，把这个格言说成“算法+数据结构=模块”会更好一些，模块是由个人或密切合作的小型软件工程师组设计和开发的部件程序。

在本章中给出了Ada、Java、C和occam2中表达算法和表示数据结构的必要语言特征。虽然这些语言有差别，它们却向程序员展现了非常相似的语义模型。确实，对于命令式语言来说，支持小型编程必需的原语已经被充分理解了。 69

为了表达算法，需要块结构以及构造良好的循环和判断结构。`goto`语句已经全面丧失名声。为了给出大多数非微小模块中见到的不同逻辑单元的具体实现，也需要子程序。过程语义相对来说没有争议（虽然存在着不同的参数传递模型），但因副作用问题，函数仍然是难处理的。occam2通过构造一种不可能有副作用的函数形式做出了榜样（occam1有关于副作用的更彻底的解决方案：它根本没有函数！）。

在Ada、Java、C和occam2中可看到数据结构丰富的多样性和类型规则，虽然occam2的设施不够全面，C的类型机制不太强。强类型已被广泛接受，成为生成可靠代码必需的助手。它所强加的限制可能导致困难，但所提供的进行显式类型转换的受控手段消除了这些问题。C缺乏强类型成为在高完整系统中使用该语言的一个不利条件。

数据类型本身可以按不同方式分类。在标量和结构化数据类型之间有明确的划分。标量

类型又可被划分为离散类型（整数类型和枚举类型）和实数类型。结构化数据类型可按三个属性分类：同质性、大小和访问方法。一个结构化数据类型被认为是同质的，如果它的所有子成分都是同一类型（例如，数组）。如果子成分可以是不同类型的（例如，记录），则称这种数据结构是异质的。大小属性可以是固定的或可变的。在某些语言中，记录可以像数组一样是定长的。诸如链表、树或图之类的动态数据结构是大小可变的，程序员通常用指针（或引用）类型和存储分配器构造它们。最后，有几种访问方法可得到结构的子成分。两种最重要的方法是直接的和间接的。直接访问，如其名称的隐含意思，允许对子成分的直接访问（例如，数组的元素或记录的域）。间接访问意味着子成分的寻址可能需要经由其他成分的一个访问链。大多数动态结构只有间接访问。

同质性、大小和访问方法，这三种属性在理论上能给出至少八种不同结构。在现实中，这些属性是有联系的（例如，固定大小隐含着直接访问），只需要下列的类别：

- 具有任意边界和维数的数组
- 记录（或类）
- 70 用于构造间接寻址的任意动态数据结构的指针

任何提供适当控制结构和所有这三类数据结构的语言（就像Ada、Java和C）就能很好地支持小型编程。大型编程的额外特征在第4章中研究。

虽然有了上述讨论，实时语言可能必须限制程序员可用的设施。如果不是不可能的话，对访问动态数据结构所需的时间进行估计也是困难的。此外，希望在程序开始执行前，能够保证程序有足够存储空间可用。正因为如此，动态数组和指针可能需要从实时应用的语言特征的“批准”清单上抹去。此外，递归和无界循环也需要受到限制。

相关阅读材料

Barnes J. G. P. (1998) *Programming in Ada 1995*. Reading, MA: Addison-Wesley.

Bishop J. (2001) *Java Gently*, 3rd Edition. Reading, MA: Addison-Wesley.

Burns A. (1988) *Programming in occam2*. Reading, MA: Addison-Wesley.

Galletly J. (1990). *Occam2*. London: Pitman.

Hatton L. (1995). *Safer C: Developing Software for High-Integrity and Safety-Critical Systems*. London: McGraw-Hill.

INMOS Limited (1988) *occam2 Reference Manual*. Chichester: Prentice Hall.

Kernighan B. W. and Ritchie D. M. (1988) *The C Programming Language*. Englewood Cliffs, NJ: Prentice Hall.

练习

3.1 Ada以**end**〈构造名字〉结束每个构造，C没有使用结束标志。这些语言设计的韵味和章法是什么？

3.2 occam2、Java和C是区分大小写的（case-sensitive），Ada不是。赞成和反对区分大小写的论点是什么？

3.3 occam2在所有子表达式上加圆括号，所以不需要算符优先规则。这对可靠性和效率的影响如何？

3.4 语言应当总是需要给变量赋初值吗?

3.5 Ada中`exit`语句的使用导致可读和可靠的程序吗?

3.6 为什么occam2中不允许递归?

3.7 列出安全编程的理想语言特征。

3.8 为什么Java不支持指针类型?

3.9 C在高完整系统中能够使用到什么程度? 71

第4章 大型编程

第3章指出，在管理大型嵌入系统的复杂特性时，分解和抽象是两种最重要的方法。这种复杂性不仅仅是因为代码的数量，更重要是因为与现实世界相互作用相应的各种各样的活动和需求。正如在1.3.1节中指出的，现实世界是不断变化的。而且，软件的设计、实现和维护工作常常管理得很差，那结果就是产生不能令人满意的产品。本章研究那些有助于体现和支持分解和抽象的语言特性。这些特性被称为是帮助*大型编程*（programming in the large）的。

模块是很重要的结构，但在诸如Pascal的早期语言中却没有提供。用一种非正式的说法，模块是一些逻辑上相关的对象和操作的集合。将系统功能隔离在模块之内，并对模块接口提供精确规格说明的技术称为**封装**。因此，模块结构可能支持:

- 信息隐藏
- 分别编译
- 抽象数据类型

在下面的几节中我们将描述模块结构的主要促成因素。本章用来作为例子的代码都采用Ada、Java和C。Ada和Java用**包**（Packages）的形式明确地支持模块。C对模块提供很弱的支持，它仅仅通过允许文件的分别编译提供间接的模块支持。相比之下occam2不支持模块分解。 73

虽然模块允许封装，但这基本上是静态的结构机制，而不是语言类型模型的一部分。更动态的封装机制以类和对象的形式提供（Java提供这种支持）。

4.1 信息隐藏

在一些简单语言里，所有持久变量必须是全局的。如果两个或多个过程想要共享数据，那么这个数据对程序的其他部分一定也是可见的。即使只有一个过程想在它每次被调用时更新某个变量，这个变量也必须在过程体外声明，因此这就存在变量误用和出错的可能性。

通过允许把信息隐藏在模块“体”内，模块结构支持降低的可见性。所有模块结构（有许多不同的模型）允许程序员控制对模块变量的访问。为了说明信息隐藏，我们考虑一个FIFO动态队列的实现。队列管理器（对程序其余部分的）接口由三个过程实现：向队列加入元素，从队列删除元素和测试队列是否为空。对模块外部来说有关队列的内部信息（例如队列指针）应该是不可见的。下面提供了这种列表结构的用Ada写的一个包。值得注意的是:

- Ada包总是声明为两部分：*规格说明*和*体*。只有规格说明里声明的实体对外部是可见的。

- Ada使用“开放作用域”。包声明部分里的所有可见标识符都可以在包内访问。Ada没有Modula-2、Java的那种导入清单。
- 通过引用包名和所需的标识符，例如Queuemod.Empty，所有导出的Ada标识符可以从包外访问。
- 只支持这个队列的一个实例，在包初始化段通过调用Create创建该实例。

```
package Queuemod is
  -- 假设类型 Element在作用域内
  function Empty return Boolean;
  procedure Insert (E : Element);
  procedure Remove (E : out Element);
```

74

```
end Queuemod;

package body Queuemod is

  type Queue_Node_T;  -- 前向声明
  type Queue_Node_Ptr_T is access Queue_Node_T;

  type Queue_Node_T is
    record
      Contents : Element;
      Next : Queue_Node_Ptr_T;
    end record;

  type Queue_T is
    record
      Front : Queue_Node_Ptr_T;
      Back : Queue_Node_Ptr_T;
    end record;

  type Queue_Ptr_T is access Queue_T;

  Q : Queue_Ptr_T;

  procedure Create is
  begin
    Q := new Queue_T;
    Q.Front := null; -- 严格说，这是不必要的
    Q.Back := null;  -- 因为指针总是被初始化为null
  end Create;

  function Empty return Boolean is
  begin
    return Q.Front = null;
  end Empty;

  procedure Insert (E : Element) is
    New_Node : Queue_Node_Ptr_T;
  begin
    New_Node := new Queue_Node_T;
    New_Node.Contents := E;
    New_Node.Next := null;
    if Empty then
      Q.Front := New_Node;
```

```
      else
        Q.Back.Next := New_Node;
      end if;
      Q.Back := New_Node;
    end Insert;

    procedure Remove (E : out Element) is
      Old_Node : Queue_Node_Ptr_T;
    begin
      Old_Node := Q.Front;
      E := Old_Node.Contents;
      Q.Front := Q.Front.Next;
      if Q.Front = null then
        Q.Back := null;                                75
      end if;
      -- 释放 old_node, 见 4.5.1节
    end Remove;
  begin
    Create; -- 创建队列
  end Queuemod;
```

包规格说明和体必须放置在同一个声明部分，虽然包规格说明和体之间可以定义其他的实体。通过这种方式，两个包可以互相调用子程序而不需要前向声明。

任何包只要在作用域内，都可以使用。为了减少过多的指名，Ada提供“use”语句:

```
declare
  use Queuemod;
begin
  if not Empty then
    Remove (E);
  end if;
end;
```

Java的包设施最好在面向对象的编程模式的上下文中解释，将在4.4节讨论它。

C语言没有提供这么正式的模块。相反，程序员必须利用分开的文件：头文件（通常以“.h”为文件扩展名）和体文件（通常以“.c”为文件扩展名）。再次考虑队列模块，首先是头文件（queuemod.h）。

```
/* 假设element在作用域内 */

int empty ();

void insertE (element E);
void removeE (element *E);
```

这就定义了模块的功能接口。模块使用者简单使用这个文件即可。模块体如下所示:

```
...
#include "queuemod.h" /*使模块规格说明对模块体可见 */

  struct queue_node_t {
      element contents;
      struct queue_node_t *next;
  };
```

76

```
struct queue_t {
  struct queue_node_t *front;
  struct queue_node_t *back;
} *Q; /* Q 现在指向 struct queue_t */
void create ()
{
  Q = (struct queue_t *) malloc (sizeof (struct queue_t) );
  Q->front = NULL;
  Q->back = NULL;
};

int empty ()
{
  return (Q->front == NULL);
}

void insertE (element E)
{
  struct queue_node_t *new_node;

  new_node = (struct queue_node_t *)
               malloc (sizeof (struct queue_node_t) );
  new_node->contents = E;
  new_node->next = NULL;
  if (empty) {
    Q->front = new_node;
  } else {
    Q->back->next = new_node;
  };
  Q->back = new_node;
}

void removeE (element *E)
{
  struct queue_node_t *old_node;

  old_node = Q->front;
  *E = old_node->contents;
  Q->front = Q->front->next;
  if (Q->front == NULL) {
    Q->back = NULL;
  }
  free (old_node);
}
```

应该注意的是，在C语言中，“.c”和“.h”文件之间并没有正式的关系。实际上，在queuemod.h中的规格说明和文件queuemod.c中的代码不需要有任何关系。它们的使用纯粹是一种约定。相反，在Ada语言中包是一个整体，对每一个子程序规格说明都必须有一个对应的子程序体。如果子程序规格说明没有一个对应的子程序体，Ada在编译时会产生编译错误，但对于C语言，直到链接时才能捕获没有函数体的错误。

怎么强调模块构造对实时程序设计的重要性都不会过分。然而，正如前面提到的，模块并不是第一类语言实体，不能定义模块类型，不能创建指向模块的指针等等。动态语言Simula的

倡导者早就指出自从20世纪60年代以来语言中已经有类的概念可用。而且，Smalltalk-80已经表现出动态模块结构的强大生命力。动态模块问题和面向对象程序设计紧密相关，这将在4.4节讨论。 77

4.2 分别编译

如果程序是由模块构造的，那么分别编译这些模块就有很明显的优点。我们说这样的程序是在库的上下文中被编译的。因此程序员可以只关注当前的模块，但是也能够构造整个程序（至少是其中的一部分），以使得他们的模块可以被测试。一旦经过测试，也可能要经过批准，新的单元可以以预编译的形式加入库中。如果整个程序不必因每一次很小的编辑改动而重新编译，那么将会节省资源和支持项目管理。

正如上一节说过的，在Ada（和C）中，包（模块）规格说明和体可以用一种很直接的方式预编译。如果一个库单元想访问其他库单元，那么它必须使用**with**子句（`#include`）明确地指示出来：

```
package Dispatcher is
  -- 新的可见对象
end Dispatcher;

with Queuemod;
package body Dispatcher is
  -- 隐藏的对象
end Dispatcher;
```

用这种方式，就构建了库单元之间依赖关系的层次结构。主程序使用`with`子句获得对它所需库单元的访问。

Ada模型一个重要特性（C也在一定程度上存在）是模块规格说明和体在库里被看作不同的实体。很明显在最终的程序编译阶段，两者都要存在（更准确地说，有些模块规格说明不要求模块体；例如，如果它们只是定义了类型和变量）。然而在程序开发阶段，库可能只包含规格说明。这些可以用来在具体的实现工作之前检查程序的逻辑一致性。在项目管理的背景下，规格说明由资深人员来做比较好，这是因为规格说明代表软件模块间的接口。在这些代码里的任何一个错误都比包体里的错误要严重，这是因为对规格说明的更改可能要求这个模块的所有用户进行更改和重新编译，而对包体的更改仅仅要求这个包体重新编译。

分别编译支持自底向上的编程。库单元是从其他单元建立的，这要持续到最后程序编码完成为止。在自顶向下设计的上下文中，自底向上的编程是完全可以接受的，特别是指根据规格说明（定义）的自底向上，而不是实现（体）的自底向上。然而，Ada还包括了更进一步的分别编译特征，它更直接支持自顶向下的设计。在程序单元里，使用关键字**is separate**可以留下“存根”，以后再来充实。例如下面的代码表明了过程`Convert`的实现如何留待主程序定义之后再进行： 78

```
procedure Main is
  type Reading is ...
  type Control_Value is ...
  procedure Convert (R : Reading; Cv : out Control_Value)
            is separate;
begin
```

```
    loop
      Input (Rd);
      Convert (Rd, Cv);
      Output (Cv);
    end loop;
  end;
```

稍后加入过程体:

```
  separate (Main)
  procedure Convert (R : Reading; Cv : out Control_Value) is
    -- 实际所需代码
  end Convert;
```

在Ada中，分别编译被整合进入语言规范之中。最重要的是：强类型规则同样适用于跨库单元和整个程序被构造成一个单元的情况。这是一种比C支持（FORTRAN、Pascal也有一些实现）的预编译单元链接更可靠的机制。对后一种方法，全面类型检查是不可能的。然而，C确实有一个叫做lint的专门工具，用来检查跨编译单元的一致性。

4.3 抽象数据类型

第3章指出高级语言的主要优点是程序员不必关心数据本身在计算机内的物理表示。数据类型的思想就来自这种分离。抽象数据类型（abstract data type，ADT）是这种概念的进一步扩展。为了定义ADT，模块要命名一种新的类型并给出可以运用在这种类型上的所有操作。ADT的结构隐藏在模块里。注意，现在可以支持这种类型的一个以上的实例，因此在模块接口里需要提供一个创建例程。

对模块规格说明和模块体要分别编译的需求使ADT的设施复杂化了。因为ADT的具体结
79 构需要隐藏，定义它的合理位置只能在模块体内。但当编译程序编译使用规格说明的代码时，编译程序并不知道类型的大小。一个解决这个问题的方法是强迫程序员使用间接的方式。例如，在C中，queuemod模块的接口变成如下的样子:

```
typedef struct queue_t *queue_ptr_t;

queue_ptr_t create ();

int empty (queue_ptr_t Q);

void insertE (queue_ptr_t Q, element E);
void removeE (queue_ptr_t Q, element *E);
```

尽管这是一个可接受的方法，但它并不总是合适的。由于这个原因，Ada也允许在规格说明定义里出现部分实现，但只可以从包体访问它们。这称为规格说明的**私有**部分。

为了比较，继续这个队列的例子。队列的ADT的Ada定义如下:

```
package Queuemod is
  type Queue is limited private;
  procedure Create (Q : in out Queue);
  function Empty (Q : Queue) return Boolean;
  procedure Insert (Q : in out Queue; E : Element);
  procedure Remove (Q : in out Queue; E : out Element);
private
```

```
  -- 以下声明在外部都是不可见的
  type Queuenode;
  type Queueptr is access Queuenode;
  type Queuenode is
    record
      Contents : Element;
      Next : Queueptr;
    end record;
  type Queue is
    record
      Front : Queueptr;
      Back : Queueptr;
    end record;
end Queuemod;

package body Queuemod is
  -- 基本上与原有代码相同
end Queuemod;
```

关键字**limited private**意味着该类型只能在定义在这个包里的子程序里使用。因此受限私有类型是真正的抽象数据类型。然而，Ada也认识到许多ADT需要赋值运算符和相等性测试操作。因此，与其在需要它们时定义它们，类型可被声明为只是**private**（私有）的——当然不是说在所有情况下都需要这样定义。如果是这种情况，那么除了已定义的子程序，赋值和相等测试例程对所有用户也是可用的。下面给出一个用Ada编制的ADT的常见例子。它提供了一个可以进行复合运算的包。注意到类型Complex里定义的子程序采取了重载操作的形式，因此允许写出“常规”的算术表达式: 80

```
package Complex_Arithmetic is
  type Complex is private;
  function "+"  (X, Y : Complex) return Complex;
  function "-"  (X, Y : Complex) return Complex;
  function "*"  (X, Y : Complex) return Complex;
  function "/"  (X, Y : Complex) return Complex;
  function Comp (A, B : Float) return Complex;
  function Real_Part (X: Complex) return Float;
  function Imag_Part (X: Complex) return Float;
private
  type Complex is
    record
      Real_Part: Float;
      Imag_Part: Float;
    end record;
end Complex_Arithmetic;
```

4.4 面向对象编程

把ADT变量称为对象和指定编程模式，这些已经变成了潮流，这也导致了面向对象编程（OOP）这个术语的使用。然而，对象抽象更严格的定义（Wegner，1987）在对象和ADT之间划出了一个明显的界限。一般来说，ADT缺少四个使它们适合于OOP的特性。它们是:

1）类型扩展（继承）

2）自动的对象初始化（构造器）

3）自动的对象终了化（析构器）

4）操作的运行时分派（多态性）

Ada和Java都以某种形式支持以上所有特性。在前面的队列例子里，需要声明一个队列变量，然后调用创建过程初始化它。在OOP中，当每个队列对象被声明时，这种初始化（使用
81 构造器例程）自动执行。类似地，当一个对象超出它的作用域时，自动执行析构器过程。

性质（2）和（3）是很有用的，但在对象抽象里更有意义的概念是可扩展性。它使得我们可以从一个已经定义的类型扩展出一个新类型。新类型继承"基"类型，但可能包含许多新的域和新的操作。一旦类型被扩展，就需要操作的运行时分派以确保类型家族的特殊实例可以调用其适宜的操作。

C并不支持OOP，但C的扩展C++已经非常流行，并成为一个ISO标准。

完整描述面向对象软件开发的含义已超出了本书的范围。然而，为了理解Java提供的并发和实时特性，必须很好掌握OOP原理。

4.4.1 OOP和Ada

Ada通过两种互补的机制支持面向对象编程：标记类型和类宽类型，这提供了类型扩展和动态多态性。子包和受控类型也是Ada提供支持的重要特性。

1. 标记类型

在Ada中，新类型可以从老类型派生而来，可以使用子类型改变类型的一些属性。例如，下面声明了一个新类型和子类型叫做Setting，它和Integer类型具有相同的属性，但却有一个受限的范围。Setting和Integer是不同的，不能互换：

```
type Setting is new Integer range 1 .. 100;
```

可以定义操作类型Setting的新操作，然而，不能加入新的成分。标记类型则打破这种限制，允许额外成分的加入。任何可能需要用这种方式扩展的类型都必须声明为标记类型。因为扩展类型不可避免导致类型变成记录，因此只有记录类型（或被实现为记录的私有类型）才能是标记类型。例如，考虑下面的类型和基本操作：

```
type Coordinates is tagged
  record
    X : Float;
    Y : Float;
  end record;

procedure Plot (P: Coordinates);
```

这个类型可以被扩展为：

82
```
type Three_D is new Coordinates with
  record
    Z : Float;
  end record;

procedure Plot (P: Three_D); -- 对子程序 Plot进行覆盖
Point : Three_D := (X => 1.0, Y => 1.0, Z => 0.0);
```

所有用这种方式派生的类型（包括最初的根）都属于同一**类**层次结构。当一个类型被扩展时，它自动继承父类型的任何基本操作。

在上例中的Three_D类的域对该类的用户是直接可见的。Ada95也允许用私有类型将这些属性完全封装：

```
package Coordinate_Class is
  type Coordinates is tagged private;

  procedure Plot (P: Coordinates);

  procedure Set_X (P: Coordinates; X: Float);
  function Get_X (P: Coordinates) return Float;
  -- 对Y类似
private
  type Coordinates is tagged
  record
    X : Float;
    Y : Float;
  end record;
end Coordinate_Class;
```

其他的设施包括抽象标记类型和抽象基本操作的概念。这些类似于4.4.2节要讨论的Java设施。注意Ada仅支持从单个父亲继承，然而使用这个语言的类属设施可以实现多重继承。

2. 类宽类型

标记类型提供一种可以增量式扩展类型的机制。其结果是程序员可以创建相关类型的层次结构。这样，程序的其他部分可以按照它们自己的目的来处理这个层次结构中的任何成员，而不需要太关心在任何时刻它处理的是哪一个成员。Ada是一种强类型的语言，因此需要一种机制来处理来自层次结构中任何成员的对象可以被作为参数传递的问题。例如，一个子程序可能希望采取坐标作为参数而不需要知道坐标是二维的或是三维的。

类宽类型（class-wide type）编程是一种使编写的程序可以处理类型家族问题的技术。与每个标记类型T相关，有一个T'Class类型，它由以T开始的类型家族的所有类型组成。因此，下面的子程序将允许传递二维或三维的坐标。

```
procedure General_Plot (P : Coordinates'Class);
```

83

在类宽类型上对基本操作的任何调用都将导致对实际被调用类型的正确操作，这个过程被称为**运行时分派**（run-time dispatching）。

```
procedure General_Plot (P : Coordinates'Class) is
begin
  -- 整理操作
  Plot (P);
  -- 依赖于P的实际值
  -- 调用已定义的一个Plot过程
end General_Plot;
```

尽管运行时分派是一种强大的机制，但它也使实时系统引起一些问题。尤其不可能从静态地考察代码知道调用了哪些操作，这使得静态的时间分析困难（见13.7节）。

3. 子包

使用子包的主要动机是为了给单一级别的包设施加入更多灵活性。如果没有子包，对导

致包规格说明改变的更改都要求重新编译所有使用这个包的客户。这就不符合面向对象编程的思想了，因为面向对象编程要方便增量式更改。而且，如果没有增加的语言特性，那么扩展私有标记类型就是不可行的——因为私有类型里的数据只可以从包体里面访问。

考察前一节给出的下列例子：

```
package Coordinate_Class is
  type Coordinates is tagged private;

  procedure Plot (P: Coordinates);

  procedure Set_X (P: Coordinates; X: Float);
  function Get_X (P: Coordinates) return Float;
  -- Y的操作与此类似
private
  type Coordinates is tagged
  record
    X : Float;
    Y : Float;
  end record;
end Coordinate_Class;
```

为了扩展这个类并具有父数据的可见性，可能需要改写这个包。

84 子包允许直接访问父类的私有部分，而不必通过父类的接口。因此，下面的代码使新的和被覆盖的基本操作的实现能访问原来类的数据属性。

```
package Coordinate_Class.Three_D is
    -- "." 指出包 Three_D 是 Coordinate_Class的子包

  type Three_D is new Coordinates with private;

  -- 新的基本操作
  procedure Set_Z (P: Coordinates; Z: Float);
  function Get_Z (P: Coordinates) return Float;

  procedure Plot (P: Three_D); -- 覆盖子程序Plot
private

  type Three_D is new Coordinates with
  record
    Z : Float;
  end record;
end Coordinate_Class.Three_D;
```

4. *受控类型*

受控类型对面向对象编程提供了进一步的支持。使用这种类型，我们可以定义当类型的对象进行下列操作时可以（自动）调用的子程序：

- 创建——*初始化*（initialize）；
- 终止存在——*终了化*（finalize）；
- 赋值——*调整*（adjust）。

为了获得这些特性，类型必须是从Controlled派生，Controlled是一个在库包Ada.Finalization里预定义的类型，这就是说，它必须是Controlled类层次结构的一部分。包Ada.Finalization定义了过程Initialize、Finalize和Adjust。当类型从

Controlled派生时，这些过程可能被覆盖。因为当对象走出它的作用域时它就不再存在，退出块的时候可能涉及多个对Finalize的调用。

4.4.2 OOP和Java

通常，至少有两种方式可以把面向对象的设施引入到语言中。其一是编程语言Oberon（Wirth，1988）所提倡的类型扩展机制。这种方法也被Ada采用。另一种是（也是更流行的一种）引入类的概念到语言中。 85

在第3章，我们介绍过一个简单的Java类。

```
class Date
{
  int day, month, year;
}
```

这个例子只是说明数据项是如何组织在一起的。类的完整设施允许封装数据项（Java称它们为实例变量或域），因此很容易产生抽象数据类型。

现在要用Java为队列抽象建立一个类。首先，可以声明一个包含队列的包（如果没有已命名的包，那么系统会假设一个无名包）。也可以导入来自其他包的项目。然后，给出在这个包里面声明的类。这些类里的一个类（Queue）被声明为“public”（公有的），这意味着它可以在包外被访问。关键字**public**在Java里是一个修饰符，类似的类修饰符有**abstract**和**final**。**abstract**类是一个不能被实例化的类。因此这种类必须被扩展（产生子类），并在创建对象之前必须使其成为非抽象的。修饰符**final**指示类不能有子类。无任何修饰符就表示这个类只能在包内访问。

每个类可以声明局部实例变量（域）、构造器方法和常用（成员）方法。构造器方法和其相关的类有同样的名字。当类的对象被创建时，就调用合适的构造器方法。对象的所有方法也都可以有相关的修饰符。这些修饰符指出方法的访问权限，修饰符有**public**、**protected**和**private**：**public**允许完全的访问，**protected**只允许同一个包或该定义类的子类访问，**private**只允许该定义类的访问。实例变量也可以使用这些修饰符。成员方法和实例变量还有一些其他修饰符，这将在本书的其他部分介绍。

队列抽象数据类型的代码如下:

```
import somepackage.Element; // 引入元素类型
package queues; // 包名字

class QueueNode // 这个包的局部类
{
  Element data;
  QueueNode next;
}

public class Queue // 包外可用的类
{

  QueueNode front, back; // 实例变量

  public Queue () // 公有构造器
  {
    front = null;
```

86

```
        back = null;
    }
    public void insert (Element E) //  可见的方法
    {
        QueueNode newNode = new QueueNode ();
        newNode.data = E;
        newNode.next = null;
        if (empty () ) {
            front = newNode;
        } else {
            back.next = newNode;
        }
        back = newNode;
    }
    public Element remove () // 可见的方法
    {
        if (!empty () ) {
            Element tmpE = front.data;
            front = front.next;
        if (empty ())back = null;
        } // 垃圾回收器将释放那些现在悬挂的QueueNode对象
        return tmpE;
    }
    public boolean empty () // 可见的方法
    {
        return (front == null);
    }
}
```

注意，这里没有我们在Ada里看到的包的规格说明的概念，然而，使用接口可以提供类似功能，见4.5.2节。

4.4.3 继承和Java

Java通过从一个类派生另一个类支持继承。Java不支持多重继承，但我们可以使用接口取得类似的效果（见4.5.2节）。

再次考察坐标的例子，该抽象类型的类结构如下:

87

```
package coordinate;
public class Coordinate // 记住Java是区分大小写的
{
    float X;
    float Y;
    public Coordinate (float initial_X, float initial_Y) // 构造器
    {
        X = initial_X;
        Y = initial_Y;
    };
```

```
public void set (float F1, float F2)
{
  X = F1;
  Y = F2;
};

public float getX ()
{
  return X;
};

public float getY ()
{
  return Y;
};
};
```

在派生类的定义中使用**extends**关键字，并跟上基类的名字，这样可以扩展生成一个新类。新类被放在同一个包里[⊖]。

```
package coordinate;
public class ThreeDimension extends Coordinate {
  // Coordinate的子类

  float Z; // 新的域

  public ThreeDimension (float initialX, float initialY,
        float initialZ) // 构造器
  {
    super (initialX, initialY); // 调用超类构造器
    Z = initialZ;
  };

  public void set (float F1, float F2, float F3) // 覆盖的方法
  {
    set (F1, F2); // 调用超类的Set
    Z = F3;
  };

  public float getZ () // 新方法
  {
    return Z;
  };
};
```

新类有了一些改变：加入了一个新的域Z，定义了构造器类，set方法被覆盖并提供了一个新的操作。构造器首先调用基类构造器（通过关键字**super**），然后初始化最后一维。类似的，方法set也调用了基类方法。 88

与Ada不同，Java的所有方法调用都是潜在分派（动态绑定）的。例如，再次研究上面的例子：

```
package coordinate;
```

⊖ 在Java中，public类必须驻留在它们自己的文件上。所以，包可以分布在一个或多个文件上，并可以不断增补。

```
public class Coordinate
{
  float X;
  float Y;

  public Coordinate (float initial_X, float initial_Y)
  {
    X = initial_X;
    Y = initial_Y;
  };

  public void set (float F1, float F2)
  {
    X = F1;
    Y = F2;
  };

  public float getX ()
  {
    return X;
  };

  public float getY ()
  {
    return Y;
  };

  public void plot ()
  {
  // 标绘一个二维点
  };
};
```

这里加入了方法plot。现在，如果方法plot在子类里面被覆盖:

```
package coordinate;
public class ThreeDimension extends Coordinate
{
  public ThreeDimension (float initialX, float initialY,
                         float initialZ)
  {
    super (initialX, initialY);
    Z = initialZ;
  };

  float Z;

  void set (float F1, float F2, float F3)
  {
    set (F1, F2);
    Z = F3;
  };

  float getZ ()
  {
```

89

```
    return Z;
  };
  public void plot ()
  {
    // 标绘一个三维点
  };
};
```

那么下面的代码

```
{
  Coordinate A = new Coordinate (0f, 0f);
  A.plot ();
}
```

将会标绘一个二维坐标，而

```
{
  Coordinate A = new Coordinate (0f, 0f);
  ThreeDimension B = new ThreeDimension (0f, 0f, 0f);
  A = B;
  A.plot ();
}
```

将会标绘一个三维坐标，即使A最初被声明为Coordinate。这是因为A和B都是引用类型，当把B的值赋给A时，改变的只是引用，而不是对象本身。

4.4.4 对象类

Java中所有的类都隐含地是根类Object的子类。程序4-1中给出了这个类的定义（throw将在6.3.2节讨论）。

程序4-1 Java的Object类

```
public class Object {
  public final class getClass ();
  public String toString ();
  public boolean equals (Object obj);
  public int hashCode ();
  protected Object clone ()
    throws CloneNotSupportedException;
  public final void wait ()
    throws IllegalMonitorStateException,
      InterruptedException;
  public final void wait (long millis)
    throws IllegalMonitorStateException,
      InterruptedException;
  public final void wait (long millis, int nanos)
    throws IllegalMonitorStateException,
      InterruptedException;
  public final void notify ()
    throws IllegalMonitorStateException;
```

```
    public final void notifyAll ()
      throws IllegalMonitorStateException;

    protected void finalize ()
      throws Throwable;
  }
```

Object类里面有六个方法是本书感兴趣的。三个wait和两个notify方法被用来进行并发控制，将在8.8节讨论。第六个方法是finalize，它在对象被销毁前调用。因此通过覆盖这个方法，子类可以提供它创建对象的终了化。当然，当类覆盖finalize方法时，它最后的行为应该调用它父类的finalize方法。然而，值得注意的是只有将要进行垃圾回收时，finalize方法才会被调用，这可能是在对象不再使用之后的某个时刻。Java没有C++中所熟知的析构器方法。因此这种设施并不是用来回收资源的，另一个值得推荐的替代方法是使用异常处理设施（见6.3.2节）。

4.5 可重用性

软件生产是一种昂贵的业务，每年软件成本都不断上升。高成本的一个原因是软件好像总是从头开始构建。相比之下，硬件工程师可以选择那些经过很好试验和测试的部件来组成
90~91 新的系统。对软件工程师来说，用类似的方法来提供软件部件是一个长期就有的追求。令人遗憾的是，除了一些特殊的领域（如数值分析），这个追求还远远没有实现。然而，可重用代码明显将会对软件构造的可靠性和生产率产生有益的深远影响。

诸如模块化和面向对象程序设计这样的现代编程语言技术为可重用软件库构造提供了一个很好的基础。

可重用性的一个障碍是我们在第3章提及的强类型模型。例如，在这样的模型下，对整数的排序模块不能用来对实数或记录排序，即使基本的算法是相同的。尽管这种类型的限制对某些原因来说是必要的，但它严重限制了可重用性。Ada和Java的设计者解决了这个问题，提供了有助于重用却不必破坏类型模型的设施。在Ada中，这种设施是基于类属模块的概念（C++提供一个类似的设施，可以定义称作**模板**（templates）的类属类）。Java使用了另一种方法，即接口的概念。下两小节将讨论这些设施。

4.5.1 Ada类属编程

类属（generic）是可以通过**实例化**生成实际部件的模板。本质上，类属可以处理对象而不必考虑其类型。实例化指定实际类型。这个语言模型确保在类属里关于类型的任何假设都会根据实例化时的类型命名进行检查。例如，一个类属可以假设**类属参数**是离散类型的，当实例化时，编译器会检查指定的类型是离散的。

类属可重用性的测度来源于在类属参数上施加的限制。例如，在一种极端的情况下，假若实例化的类型必须是一维整型数组，那么这种类属就不可重用了。相反，如果在实例化时可以使用任何一种类型，那么就得到高级别的可重用性。

Ada类属的参数模型是很广泛的，这里我们不详细讨论它。我们会给出几个高度可重用的例子。

定义类属参数的方式是在类属体内指定应用在它们上面的操作。如果对于参数来说，没有任何应用在其上的操作，那么参数是**受限私有的**（limited private）。如果只有赋值和测试相等操作，那么参数是**私有的**（private）。高可重用的部件具有受限私有或私有的参数。作为第一个

例子，我们看看包Queuemod。正如已经给出的，尽管给队列提供了抽象数据类型，但所有这样的队列只能拥有Element类型的对象。很明显，需要的类属是其被处理的对象的类型是参数。在包Queuemod的体内，类型Element的对象仅在进出队列时赋值，因此这个参数是私有的。 92

```
generic
  type Eelemnt is private;
package Queuemod_Template is
  type Queue is limited private;
  procedure Create (Q : in out Queue);
  function Empty (Q : Queue) return Boolean;
  procedure Insert (Q : in out Queue; E : Element);
  procedure Remove (Q : in out Queue; E : out Element);
private
  type Queuenode;
  type Queueptr is access Queuenode;
  type Queuenode is
    record
      Contents : Element;
      Next : Queueptr;
    end record;
  type Queue is
    record
      Front : Queueptr;
      Back : Queueptr;
    end record;
end Queuemod_Template;

package body Queuemod_Template is
  -- 同前
end Queuemod_Template;
```

该类属的实例化将创建一个实际的包:

```
declare
  package Integer_Queues is new Queuemod_Template (Integer);
  type Processid is
    record
      ...
    end record;
  package Process_Queues is new Queuemod_Template (Processid);
  Q1, Q2 : Integer_Queues.Queue;
  Pid : Process_Queues.Queue;
  P : Processid;
  use Integer_Queues;
  use Process_Queues;
begin
  Create (Q1);
  Create (Pid);
  ...
  Insert (Pid, P);
  ...
end;
```

每个包都对队列定义了一个抽象数据类型，但是它们是不同的包，因为元素类型是不同的。

上面的讨论集中在把类型作为类属参数。这里还存在其他形式的类属参数：常量、子程序、包。在实时程序中缓冲区是一种重要的构造。它们不同于队列，因为它们有固定的大小。下面是缓冲区的抽象数据类型类属包的规格说明。也把元素类型作为类属参数。另外，缓冲
93 区大小是一个类属常量参数，它的默认大小是32：

```
generic
  Size : Natural := 32;
  type Element is private;
package Buffer_Template is
  type Buffer is limited private;
  procedure Create (B : in out Buffer);
  function Empty (B : Buffer) return Boolean;
  procedure Place (B : in out Buffer; E : Element);
  procedure Take (B : in out Buffer; E : out Element);
private
  subtype Buffer_Range is Natural range 0..Size-1;
  type Buff is array (Buffer_Range) of Element;
  type Buffer is
    record
      Bf : Buff;
      Top : Buffer_Range := 0;
      Base : Buffer_Range := 0;
    end record;
end Buffer_Template;
```

可以通过以下实例化得到一个大小为32的整数缓冲区：

```
package Integer_Buffers is new Buffer_Template (Integer);
```

一个大小为64的记录类型Rec的缓冲区可以用类似的方法构造如下：

```
package Rec_Buffers is new Buffer_Template (64, Rec);
```

像子程序的参数一样，使用名字关联更具可读性：

```
package Rec_Buffers is new Buffer_Template (Size => 64,
                                            Element => Rec);
```

下面给出一个使用子程序作参数的类属例子。类属包定义了两个作用在Element数组上的过程。一个是寻找最大的Element，另一个是对数组排序。为了实现这两个过程，比较两个Element的大小是一个必要的操作。对数值类型，“>”是可用的操作符，但对一般的私有类型则不可用。这样就必须引入一个“>”函数。

```
generic
  Size : Natural;
  type Element is private;
  with function ">" (E1, E2 : Element) return Boolean;
package Array_Sort is
  type Vector is array (1 .. Size) of Element;
  procedure Sort (V: in out Vector);
  function Largest (V: vector) return Element;
end Array_Sort;
```

94

这个类属的实现留给读者作为练习（练习4.8）。

如果把包（这些包本身是类属实例化得到的）作为类属参数，那么可以创建更复杂的类属包。然而这里我们并不介绍，因为后面的章节不需要它们。

4.5.2 Java中的接口

Java中的接口使得类提高了代码可重用性。接口是类的一种特殊形式，它是由一系列方法和常量组成的规格说明。接口是抽象的，因此不能实例化。相反，一个或多个类可以实现接口，并且通过把参数定义为某个接口类型的，实现接口的对象可以作为变元传递给方法。实际上，接口所做的事就是能够在类层次结构之外建立起类之间的关系。

考虑对对象数组排序的“类属的”算法。所有可排序对象的共同点是它们支持‘<’或‘>’操作符。因此这个特性可以封装在接口里。

Ordered接口定义了一个方法lessThan，这个方法将实现Ordered的类的对象作为变元。任何实现接口Ordered的类都必须将它的对象和作为参数传进来的对象比较，并返回它是否小于这个对象[⊖]。

```
package interfaceExamples;

public interface Ordered {
  boolean lessThan (Ordered O);
};
```

复数类就是这样的一个类。

```
import interfaceExamples.*;
class ComplexNumber implements Ordered
{
  // 这个类实现 Ordered接口

  protected float realPart;
  protected float imagPart;

  public boolean lessThan (Ordered O) // 接口实现
  {
     ComplexNumber CN = (ComplexNumber) O; // 把参数进行类型转换

     if ((realPart*realPart + imagPart*imagPart) <
          (CN.getReal () *CN.getReal () + CN.getImag () *CN.getImag () ) )
     {
        return true;
     }
     return false;
  };

  public ComplexNumber (float I, float J) // 构造器
  {
    realPart = I;
    imagPart = J;
  };

  public float getReal ()
  {
    return realPart;
  };
```

95

⊖ 将这一点同Ada风格进行比较：两个对象的比较是由一个函数执行的，它取两个对象作为参数。

```
  public float getImag ()
  {
    return imagPart;
  };

}
```

现在能够写出一个算法，它将这个类和其他实现这个接口的类排序。例如，在4.5.1节中，对Array_Sort给出了一个Ada类属包，等价的Java包如下：

```
package interfaceExamples;

public class ArraySort
{
  public static void sort (Ordered oa[], int size) // 方法 sort
  {
    Ordered tmp;
    int pos;

    for (int i = 0; i < size - 1; i++) {
      pos = i;
      for (int j = i + 1; j < size; j++) {
        if (oa[j].lessThan (oa[pos]) ) {
          pos = j;
        }
      }
      tmp = oa[pos];
      oa[pos] = oa[i];
      oa[i] = tmp;
    };
  };

  public static Ordered largest (Ordered oa[], int size) //方法 largest
  {
    Ordered tmp;
    int pos;
    pos = 0;
    for (int i = 1; i < size; i++) {
      if (!oa[i].lessThan (oa[pos]) ) {
        pos = i;
      };
    };
    return oa[pos];
  };
}
```

96

sort方法有两个变元，第一个是实现Ordered接口的对象数组，第二个是在这个数组里元素的个数。该实现执行了一个交换排序。这个例子的重要之处在于当交换两个对象时，交换的只是引用的值；因此，对象的类型是没有关系的（只要支持Ordered接口）。

为了使用上述类和接口，需要下面的：

```
{
  ArraySort AR = new ArraySort ();
  ComplexNumber arrayComplex[] = { // 例如，有这些元素
```

```
                    new ComplexNumber (6f, 1f),
                    new ComplexNumber (1f, 1f),
                    new ComplexNumber (3f, 1f),
                    new ComplexNumber (1f, 0f),
                    new ComplexNumber (7f, 1f),
                    new ComplexNumber (1f, 8f),
                    new ComplexNumber (10f, 1f),
                    new ComplexNumber (1f, 7f)
    };
    // 数组未排序
    AR.sort (arrayComplex, 8);
    // 数组已排序
  }
```

事实上，包java.lang已经定义了一个名为Comparable的接口，它有一个方法compareTo，可以用来取代以上的Ordered接口。而且，包java.util.Arrays有一个名字为sort的静态方法实现了对象的合并排序。

接口还有三个应该注意的特性。首先，像类一样，接口可以参与同其他接口的单继承的关系。第二，一个类可以实现多个接口，因而通过这个特性可以取得多重继承的效果。第三，接口也提供了能够实现回调（call back）的机制。回调是指服务器调用定义在调用者的类里面的方法。

小结

在编程语言的演化中，模块是已出现的最重要的结构之一。这个结构使大型实时系统固有的复杂性得以控制和管理。特别是，它支持: 97

1）信息隐藏

2）分别编译

3）抽象数据类型

Ada和C都有静态模块结构。Ada使用开放作用域，因此在模块声明作用域内的所有对象对模块内部来说都是可见的。C只是非正式地支持模块，而Java以类的形式支持动态模块结构。Ada（和Java）的包以及Java的类有定义良好的规格说明，用来作为模块和程序其他部分的接口。

分别编译使预编译部件库能够得以构造。它鼓励重用，为软件设计提供一个仓库。但是，这种仓库必须服从合适的项目管理，使得诸如版本控制这样的问题不会成为不可靠性的来源。

大型编程的本质是将大型程序分解成模块或者类。但是，重要的是分解过程应该产生结构良好的模块。

抽象数据类型（ADT）或面向对象编程是程序员管理大型软件系统的主要工具之一。正是Ada和Java中的模块构造，使得能够建立和使用ADT。

对于强类型语言来说，由于其行为和参数、子部件的类型紧紧绑在一起，所以其模块不易被重用。这种依赖性经常是比需要的多。Ada和C++提供的类属单元是一种提高软件可重用性的尝试。类属包和类属过程被作为模板，实际的代码来自模板的实例化。Java通过接口取得同样的效果。这些设施的合理使用应该使实时程序降低成本，同时也提高其可靠性。

相关阅读材料

Ben-Ari, M. (1998) *Ada for Software Engineers*. Chichester: Wiley.

Darnell, P. A. and Margolis, P. E. (1996) *C: A Software Engineering Approach*. London: Springer-Verlag.

Meyer, B. (1992) *Eiffel: The Language*. New York: Prentice Hall.

Sommerville, I. (2001) *Software Engineering*, 6th Edition. Reading, MA: Addison-Wesley.

Horstmann, C. S. and Cornell, G. (1999) *Core Java Fundamentals*. Sun Microsystems.

98 Schach, S. R. (1997) *Software Engineering with Java*. Chicago, IL: Richard Irwin.

练习

4.1 为什么过程不足以作为程序模块？

4.2 区分分别编译、独立编译和多道程序设计这三个概念。

4.3 评价C语言的模块化编程方法。

4.4 把Ada包作为第一类语言对象有什么优缺点？

4.5 比较和对照Ada和C把函数作为参数传递给其他函数的机制。

4.6 比较和对照Ada和Java的OOP设施。

4.7 为时间定义一个抽象数据类型。在时间值上会有什么操作？

4.8 实现在4.5.1节中给出的Ada类属包。说明如何使用它来对任一个缓冲区数组排序（假设如果一个缓冲区相对另一个缓冲区有较多的元素，那么它“大于”另一个）。

4.9 OOP允许运行时的操作分派。那么实时编程语言是否应该只允许静态绑定？

4.10 在Java中，所有的对象都是动态分配的。讨论垃圾回收对实时编程语言的含义。

4.11 Ada95和Java在多大程度上支持多重继承？

99 4.12 Ada95支持子包的概念。讨论子包在Ada支持OOP中所起的作用。

第5章 可靠性和容错

对实时和嵌入式系统来说，可靠性和安全要求通常要比对其他计算机系统更严格。例如，如果用于科学问题求解的应用程序失败，那么中止该程序是合理的，因为只是损失了计算机时间。然而，对于嵌入式系统来说，中止程序也许是不能接受的动作。例如，负责大煤气炉操作的过程控制计算机，在出现故障时不能把炉子关掉去检修计算机。相反，它必须试着提供一种次一级的服务并且避免停工的损失。更重要的是，如果实时计算机系统放弃对它们的应用的控制，可能会危及到人的生命安全。控制核反应堆的嵌入式计算机一定不能让反应堆失控，否则一旦反应堆失控就可能导致堆芯熔化和辐射泄漏。航空电子系统至少应该允许驾驶员在飞机坠毁前弹射出来。

现在，越来越多以前由操作员或已被证明的模拟方法执行的控制功能，正被数字计算机 101
管理。在1955年，仅有10%的美国武器系统需要计算机软件，到20世纪80年代初，这个数字已经上升到80%（Leveson，1986）。由于软件中的差错导致任务失败的例子很多。20世纪70年代初，一颗控制高空气象气球的法国气象卫星，由于软件出错，发出了“紧急自爆”指令而不是“读入数据”指令，结果141个气球中的72个被毁（Leveson，1986）。许多类似的惊人例子被记录在案，可以设想还有更多的没有被记录。1986年，文献Hecht and Hecht（1986b）研究了大型软件系统并得出结论，一般每百万行代码中会有20 000个bug；通常90%在测试时被发现，其余的错误，有200个将会在操作的第一年出现，而剩下1 800个bug未被发现。例行维护通常会在修复200个bug的同时产生200个新错！

人类将越多的重要功能的控制交给计算机系统，就越应该保证那些系统不能失效。一般来说，有4种故障源导致嵌入式系统失效。

1）不充分的规格说明。绝大多数的软件故障都是由不充分的规格说明产生的（Leveson，1986）。

2）软件部件中由设计错误引入的故障。

3）由嵌入式系统的一个或多个的处理器部件失效引入的故障。

4）由通信子系统中出现的瞬时或持续的干扰引入的故障。

最后的三种故障影响到了用来实现嵌入式系统的程序设计语言。由设计差错产生的错误，

其后果一般来说是不可预计的，而那些由处理器和网络引发的故障在某种程度上是可以预知的。因此对任何实时程序设计语言的主要要求之一就是它必须便于构造高可靠系统。这一章将研究一些可以用来改善嵌入式系统整体可靠性的通用设计技术。第6章将讨论异常处理设施怎么能被用来帮助实现这些设计思想，特别是基于容错的思想。有关处理器和通信失效的问题到第14章再讨论。

5.1 可靠性、失效和故障

在继续讨论之前，必须更精确地定义可靠性、失效和故障。Randell 等（1978）定义系统
102 的**可靠性**是：

系统对于其行为的权威性规格说明的符合程度的度量。

理想的情况是，这个规格说明应该是完全的、一致的、可理解的并且无歧义的。应当注意系统的响应时间是规格说明的重要部分，但时限满足问题的讨论被放到第13章。现在能用上述可靠性定义来定义系统**失效**（failure）。再次引用Randell等的定义：

当系统的行为偏离了给它规定的行为时，就叫做失效。

5.9节将讨论可靠性的度量，就现在而言，高可靠性与低失效率被认为是同义的。

机警的读者可能已经注意到我们的定义到目前为止都与系统的行为相关，简言之，是它外部的表象。失效起因于系统内部的意外问题，这些意外问题最后在系统的外部行为中表现出来。这些问题被称为**错误**（error），而它们的机械或算法的起因就叫做**故障**（fault）。系统的有故障部件就是在系统生命期的一组特定情况下导致错误的部件。从状态迁移的观点来看，一个系统可以被看作一系列外部和内部的状态。某外部状态如果未被系统行为所说明则被看作是系统的失效。系统自己由许多部件组成，每个部件都有自己的状态，它们都对系统的外部行为做出贡献。这些部件的组合状态被称作系统的内部状态。未被规定的内部状态称为错误，产生非法状态迁移的部件被认为是有故障的（faulty）。

当然，系统通常是由部件组成的，每一个部件又都可以被看作一个系统。因此一个系统中的失效将会在另一个系统中导致故障，该故障在那个系统中会产生错误和潜在失效，从而依次将故障传入所有周围的系统中（如图5-1所示）。

图5-1 故障、错误、失效、故障链

能够区分三种类型的故障：

1）**瞬时故障** 瞬时故障开始于特定的时刻，在系统中保留一段时间，然后消失。这种故
103 障有时发生在对一些外部干扰（比如电场和无线电活动）有不利反应的硬件部件中。在干扰消失后故障也就消失了（虽然不一定导致错误）。通信系统中许多故障是瞬时的。

2）**永久故障** 永久故障开始于特定的时刻，并且保留在系统中直到被修复。例如，一根折断的电线或一个软件设计错误。

3）**间歇故障** 这些故障是间歇发生的瞬时故障。例如一个热敏感的硬件部件，它工作一段时间，然后停止工作，冷却以后又开始工作。

要建立可靠的系统，必须防止所有这些类型的故障引起错误的系统行为（也就是失效）。在安全至上系统的构建中计算机的不正当使用增加了这种困难。例如，1979年，用来设计核反应堆和它们的支持性冷却系统的程序中发现了一个错误。这个在反应堆设计时产生的故障没有在安装测试期间发现，因为它与管道和阀门的强度和结构化支持有关。人们本以为这个程序达到了地震时保证安全运转反应堆的标准。最后，这个缺陷的发现导致5家核电厂关闭（Leveson，1986）。

5.2 失效模式

系统可以以许多不同的方式失效。正在使用系统X来实现另外一个系统Y的设计者通常都会对系统X可能出现的失效模式做一些假设。如果X不是按假设的模式失效的话，那么系统Y也可能失效。

系统提供服务，因此，系统的失效模式可以按失效给系统交付的服务带来的影响进行分类。可以识别出两个一般的失效模式域：

- 数值失效——与服务相关联的数值有错
- 时间失效——服务在错误的时间被交付

数值失效与时间失效的组合通常称为**任意**（arbitrary）失效。

一般来说，数值错误可能仍然在数值的正确范围之内，或是在服务预期的范围之外。后者等价于在程序设计语言中的类型错误，被称为**约束错误**。识别出这类失效通常是很容易的。

在时间域中的失效能够导致正被交付的服务出现以下情况：

- 提前——与要求相比服务被提早交付
- 延迟——服务交付时间比要求要晚（通常称为**性能错误**） 104
- 无限延迟——服务永远不被交付（通常称为**遗漏失效**）

应该识别出一种更深层的失效模式，即未预期的服务被交付了。这种失效被称为**委托**或**无准备**的失效。把既是数值域又是时间域的失效同跟随有遗漏失效的委托失效区分开是很困难的。图5-2说明失效模式的分类。

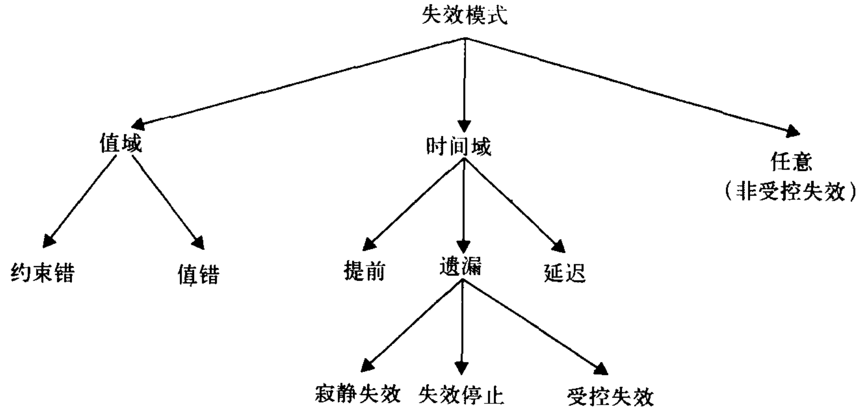


图5-2 失效模式分类

由以上给出的失效模式的分类，可以对系统可能如何失效做一些假设：

- **非受控失效**——在数值域和时间域上都产生随机错误的系统（包括无准备的错误）。

- **延迟失效**——在数值域上产生正确的服务，但遭受延迟时间错误的系统。
- **寂静失效**——在数值域和时间域上都提供正确的服务直到系统失效；惟一可能的失效就是遗漏失效，并且当遗漏失效发生时，所有后面的服务也将遭受遗漏失效。
- **失效停止**——具有所有寂静失效的属性的系统，但还允许其他系统检测到它进入了寂静失效状态。
- **受控失效**——系统在规定的受控方式下失效。
- **从不失效**——在时间域和数值域上总是提供正确服务的系统。

105 显然可能存在其他假设，但是以上假设对本书来说已经足够了。

5.3 故障预防与容错

可以区分两种帮助设计者改善系统可靠性的途径（Anderson and Lee，1990）。第一种通常叫做**故障预防**（fault prevention），它试图在系统运作前消除所有可能进入系统的故障。第二种是**容错**，它使系统在有故障出现的情况下可以继续运转。这两种途径都试图产生有明确定义的失效模式的系统。

5.3.1 故障预防

故障预防分两个阶段：**故障回避**（fault avoidance）和**故障排除**（fault removal）。

故障回避试图在系统构造期间限制引入有潜在故障的部件。对硬件来说，这可能需要（Randell等，1978）：

- 在给定的费用和性能限制以内，使用最可靠的部件；
- 为了部件的互联与子系统的装配，使用充分精湛的技术；
- 包装硬件，以阻拦预期形式的干涉。

现在大型嵌入式系统的软件部件比它们对应的硬件部件要复杂得多。尽管软件不会因为使用而变坏，但是很明显，在任何情况下编写无故障程序几乎是不可能的。然而，在第2章和第4章指出可以通过下列途径改善软件的质量：

- 严格的（即使是非形式化的）需求规格说明；
- 使用经过证明的设计方法学；
- 使用有数据抽象和模块化设施的语言；
- 使用软件工程工具帮助操纵软件部件从而管理复杂性。

尽管有故障回避技术，在系统构造后，故障还是不可避免地存在于系统中，尤其是在硬件部件和软件部件中可能都存在设计错误。因此，故障预防的第二步是*故障排除*。故障排除通常是由发现错误然后消除错误起因的过程组成的。尽管可以使用设计评审、程序验证、代码审查等技术，但是重点通常是放在系统测试上。遗憾的是系统测试不可能是穷尽式的，也不可能消除所有潜在的故障。特别存在以下问题：

106
- 测试仅仅能用来显示故障的存在而不能显示它们不存在。
- 有时不可能在真实的条件下测试——担心美国主动战略防御系统的主要原因之一是，除了在战争条件下，要实际地测试任何系统都是不可能的。大多数测试是系统在模拟状态下进行的，并且很难保证模拟的准确性。法国在太平洋上进行的最后一次核试验，据说允许进行数据收集，因此将来的试验可以被模拟得更加准确。

- 在系统开发的需求阶段引入的错误，可能直到系统开始运作才会显现出来。例如，在F18式飞机的设计中，曾对发射装置导弹的时间长度做了一个错误的假设。当导弹没能在点火后与发射器分开的时候发现了这个问题，这引起飞机剧烈失控（Leveson，1986）。

尽管有各种测试和验证技术，硬件部件还是失效；因此，当维修的频度和持续时间不可接受，或系统无法维护和进行修理的时候，故障预防方法是不成功的。后者的一个极端的例子就是旅行者号无人驾驶飞船。

5.3.2 容错

因为故障预防方法不可避免的局限性，嵌入式系统的设计者们必须考虑使用容错的方法。当然，这并不是说应当放弃预防系统故障的努力。然而，这本书将重点讨论容错而不是故障预防。

系统可以提供若干不同级别的容错。

- **完全容错**——系统在有故障存在的情况下继续运行，而没有功能或性能上的重大损失，虽然只是一段有限的时间。
- **性能降低**（graceful degradation）（或失效弱化（fail soft））——系统在有故障存在的情况下继续运行，在恢复或修补期间接受功能或性能上的部分降级。
- **故障保护**（fail safe）——当系统的操作接受临时停止时，系统维持它的完整性。

所需的容错级别取决于应用。尽管在理论上大多数安全至上的系统需要完全容错，实际上许多系统都设置为性能降低。特别地，能承受物理损坏的那些系统（例如战斗机）可以提供几个级别的性能降低。再者，在连续的基础上进行高度复杂的应用操作时（这些应用要求*高利用率*）性能降低是必需的，因为在不确定的时间段里完全容错是达不到的。例如，美国联邦航空管理部的先进自动化系统（AAS），为全美的空中交通和机场起降的管理者们提供自动化服务。它为区域控制计算机连接提供三种性能降低的级别（Avizienis and Ball，1987）。图5-3中说明了这一点。 107

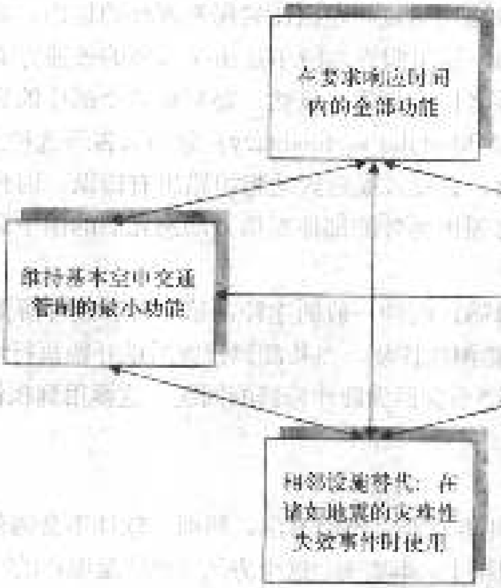


图5-3 空中交通管制系统中的性能降低和恢复

在某些情况下，以安全状态关闭系统是完全必要的。这些故障保护系统可以减少由失效带来的损害。例如，A310空中客车的机翼叶片控制计算机，在降落的过程如果检测到错误，就恢复系统到安全状态，然后关闭系统。在这种情况下，安全状态就是让双翼有相同的设置，在降落时只有不对称设置是危险的（Martin，1982）。

早期的容错系统设计方法做了三个假设：

1）正确地设计了系统的算法；

2）知道部件的所有可能失效模式；

3）已经预见到系统与环境间所有可能的交互。

然而，计算机软件复杂性的增加和超大规模集成电路硬件部件的引入，意味着做这些假设是不再可能了（如果曾经是可能的话）。因此，预见到的故障和预见不到的故障必须都被考
108 虑到。后者包括硬件和软件设计上的故障。

5.3.3 冗余

为检测故障并从中恢复，系统引入了额外的部件，所有实现容错的技术都依赖于这些部件，而它们并不是系统操作的正常模式所必需的。从这个意义上说这些部件是多余的。这通常称为**保护性冗余**（protective redundancy）。容错的目标就是：在系统的成本和大小约束的前提下，提供最大可靠性并使冗余最小。在构筑容错系统时一定要小心，因为增加的部件不可避免地增加了整个系统的复杂度。这本身可能导致产生低可靠的系统。例如，航天飞机的首次发射因为备份计算机系统的一个同步错误而取消（Garman，1981）。为了帮助减少同冗余部件之间交互有关的问题，把容错部件和系统其他部件分开是明智的。

冗余有若干种不同的分类，这些分类取决于考虑哪些系统部件和使用那些术语。软件容错是这章的主要关注点，硬件冗余技术只是偶尔提及。对于硬件，参考文献Anderson and Lee（1990）区分开了*静态*冗余和*动态*冗余。关于静态冗余，冗余部件在系统（或子系统）里用来屏蔽故障带来的影响。静态冗余的一个例子是**三模冗余**（Triple Modular Redundancy，TMR）。TMR由三个相同的子部件和多数表决电路组成。电路比较所有部件的输出，如果有一个与另外两个不同，那么就屏蔽掉这个输出。这里假设故障不是由子部件的普通方面（例如设计错误）引起的，而是瞬时的或由于部件老化引起的。显然，要屏蔽多个部件的故障需要更多的冗余。因此，术语*N模冗余*（NMR，*N* Modular Redundancy）常用来表示这种方法。

动态冗余是在部件里提供的冗余，它显式或隐式地指出输出有错误。因此它提供出错检测设备而不是错误屏蔽设备，恢复必须由另外的部件提供。动态冗余的例子是通信传播上的校验和以及存储器上的奇偶校验位。

对于软件设计错误的容错，能够确认两种一般的途径。第一种和硬件屏蔽冗余类似，称为*N*版本程序设计；第二种基于出错检测和恢复，当检测到错误后就开始执行恢复过程，在这个意义上，它类似于动态冗余。在14.5节会回到硬件容错的问题，这要用到软件技术。

5.4 N版本程序设计

109 硬件TMR和NMR的成功促使软件容错产生类似方法。然而，软件不会因使用而老化，因此该方法的使用常集中在检测设计错误上。事实上，这个方法（也就是现在的*N*版本程序设计）最早是由Babbage在1837年提倡的（Randell，1982）：

> 当公式非常复杂时，它可以用代数方法安排两种或两种以上不同的方法去计算，同时可以制作两套或两套以上的卡片。如果每副卡片现在使用一样的常数，并且在这些情况下结果是一致的，那么我们可以对它们的准确性完全放心。

独立地从同样的初始规格说明中产生*N*（*N*大于或等于2）个功能上相当的程序就叫做*N*版本程序设计（Chen and Avizienis，1978）。独立地产生*N*个程序意味着*N*个个体或组没有相互作用地生产出所需软件的*N*个版本（因为这个原因，*N*版本程序设计通常称作**设计多样性**）。一旦设计并且写好了程序，程序就在相同的输入下并发地执行并且由**驱动进程**比较它们的结果。原则上，结果应该是相同的，但在实践中可以有一些差异，在这种情况下，认为那个一致的结果（假定有一个）是正确的。

*N*版本程序设计基于如下假设：能够完全地、一致地并且无歧义地规定程序，并且独立开发的那些程序独立地失效。这就是说，一个版本中的故障与另一个版本中的故障没有任何联系。如果每个版本用同样的编程语言编写，这个假设可能会不成立，因为版本之间与语言的实现有关的错误可能是共同的。因此，应该使用不同的编程语言和不同的开发环境。否则，如果使用相同的语言就应该使用不同厂商的编译器和支持环境。此外，在任何一种情况下，为了避免出现物理故障，*N*个版本必须分布到有容错通信线路的不同机器上。在波音777飞机的控制系统中，生成一个单独的Ada程序，但使用三种不同的处理器和三种不同的编译器以达到设计多样性。

*N*版本程序是由驱动进程控制的，驱动进程主要负责：

- 调用每个版本
- 等待版本的完成
- 比较并按结果采取行动

到目前为止，都是隐式地假设程序或进程在比较结果之前已经完成运行，但是对嵌入式系统来说通常都不会是这种情况，这些进程可能永远不会完成。因此，驱动器和*N*个版本在它们执行的过程中必须进行通信。

由此可见，尽管这些版本是相互独立的，但它们必须同驱动程序交互。在版本的需求中规定了这种交互，它由三部分组成（Chen and Avizienis，1978）： 110

1）比较向量

2）比较状态指示器

3）比较点

版本怎样与驱动器通信和同步，将取决于使用的程序设计语言和它的并发模型（参见第7、8、9章）。如果不同的版本使用不同的语言，那么通常由实时操作系统提供通信和同步的手段。在图5-4中用图解显示了$N=3$的版本系统中各版本与驱动器之间的关系。

比较向量是表示输出或表决的数据结构，是由版本加上与它们计算相关的所有属性产生的，这些必须由驱动器进行比较。例如，在空中交通控制系统中，如果正在比较的值是飞机的位置，一个属性可以指示该值是最近的雷达读数，或是在原来读数的基础上计算出的结果。

比较状态指示器从驱动器传送到版本，它们指示每个版本根据驱动器的比较结果必须执行的动作。这些动作将依赖于比较的结果：是否表决同意以及它们是否按时交付。可能的结果包括：

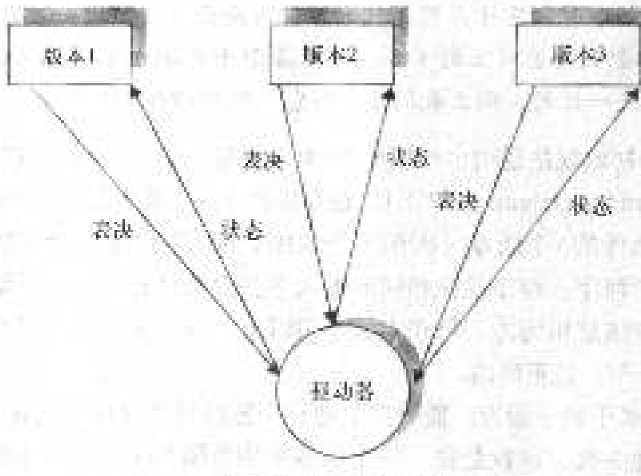


图5-4 N版本程序设计

- 继续
- 一个或多个版本终止
111 - 把一个或多个表决改为多数值后继续

比较点是版本中的点，在这些点上，必须将它们的表决通知给驱动进程。正如文献[Hecht and Hecht（1986a）]指出的，一个重要的设计决定是做比较的频率，这就是容错提供的粒度。大粒度的容错就是进行很少的比较，这将使比较策略中固有的性能损失减到最小并且在版本设计中允许大量的独立性。然而，大粒度可能在获得的结果上产生很大的差异，因为在比较之间要采取更多的步骤。在下一小节会讨论表决比较或投票（经常这样叫它）的问题。细粒度的容错需要程序结构在细节层次上的共同性，所以会降低各版本之间的独立程度。频繁的比较还增加了与这种技术相关的开销。

5.4.1 表决比较

对N版本程序设计至关重要的是效率和易用性，驱动程序用它们比较表决结果并决定是否有不一致的表决结果。对于文本处理或整数运算的应用程序通常会有一个单一的正确结果，处理器能够很容易地从不同版本中比较表决并且选择多数决定。

可惜不是所有的结果都是准确的。特别是，在表决要求实数运算的地方，不同的版本产生完全一样的结果是不太可能的。这可能是由于硬件对实数的不精确表示或具体算法的数据灵敏性引起的。比较这种类型结果采用的技术叫**不精确表决**（inexact voting）。一种简单的技术是进行范围检查，利用事先估计或从所有N个结果中取中位值的办法。然而，找到一个通用的不精确表决方法可能很困难。

另外一个同有限精确度运算有关的困难也就是所谓的**一致比较问题**（consistent comparison problem）（Brilliant等，1987）。当应用程序必须要执行基于规格说明给出的有限值的比较的时候，就会出现这个问题，比较的结果决定采取何种动作。例如，考虑一个过程控制系统，它监视温度和压力传感器，根据它们的值采取适当的动作，以保证系统的完整性。假定这些读数的任何一个超过了界限值的时候，都必须采取一些纠正的动作。现在考虑一个三版本的软件系统（V_1、V_2、V_3），每一个都必须读取两个传感器，决定某个动作，然后表决结果（在版本表决前它们之间没有通信）。由于有限精度计算的原因，每个版本将会计算不同的值（温度传感器的

值记作T_1、T_2、T_3，压力传感器的值记作P_1、P_2、P_3）。假设温度的界限值是T_{th}，压力的界限值是P_{th}，当两个读数在它们的界限值附近时，就出现了一致比较问题。

这种情况可能出现在T_1和T_2刚好低于T_{th}，而T_3刚好高于T_{th}的时候，因此V_1和V_2将沿着它们正常的执行路径而V_3将采取一些纠正的动作。现在如果版本V_1和V_2进行到了另一个比较点，那么这时在压力传感器中可能会出现P_1刚好低于P_{th}而P_2刚好高于P_{th}情况。总的结果将是：三个版本将会沿着不同的执行路径，因此产生不同的结果，每一个结果都是有效的。图5-5用图解表示了这个过程。 112

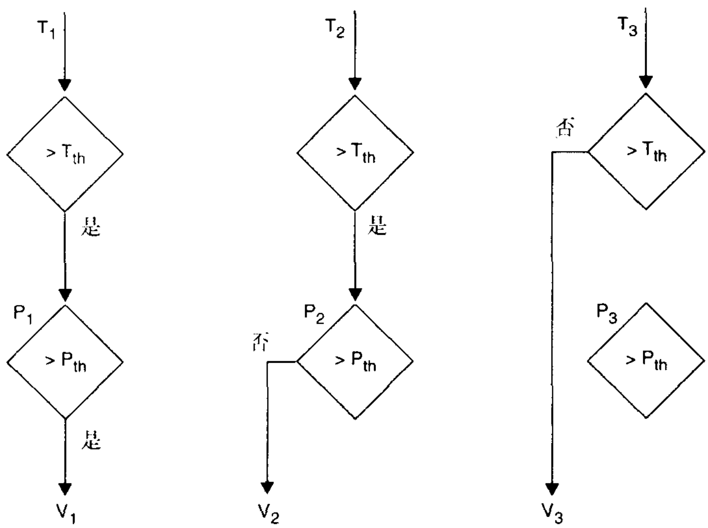


图5-5 三版本的一致比较问题

乍看之下，使用不精确比较技术和如果它们相差一个公差Δ就假设这些值是相等的似乎是恰当的，但如Brilliant等人（1987）指出的那样，当这些值接近界限值 ± Δ时，问题又会出现。

当同一问题存在多种解决方案时，还存在着表决比较的更深的问题。例如，一个二次方程可能有一个以上的解，这又是可能的不一致，尽管没有故障发生（Anderson and Lee，1990）。

5.4.2 N版本程序设计的主要问题

已经表明N版本程序设计的成功取决于几个问题，现在予以简要评述。

1）**初始规格说明**——绝大多数的软件故障都源自不充分的规格说明（Leveson，1986）。当前的技术与产生完全、一致、可理解并且无歧义的规格说明还相差很远，然而形式化的规格说明方法正在被证明是一条富有成效的研究路线。毫无疑问，规格说明的错误将会在实现的所有N个版本中表现出来。 113

2）**设计工作的独立性**——已做了一些实验（Knight等，1985；Avizienis等，1988；Brilliant等，1990；Eckhardt等，1991；Hatton，1997）以测试下述假设：独立生产的软件将显示截然不同的失效，然而，它们产生相互冲突的结果。Knight等人（1985）指出：对具有充分提炼的规格说明的特定问题，不得不否决这个假设，因为它同99%的置信度相距太远。相比之下，Avizienis等人（1998）发现在6版本系统的两个版本中找到同样的故障是非常罕见的。在比较他们的结果和那些由Knight等人产生的结果中，他们推断Knight等人研究的问题限制了多样性的

潜力，没有正式地规定程序设计过程，测试是很有限的，并且按照通用的工业标准，进行的验收测试远远是不充分的。Avizienis等人宣称*N*版本程序设计模式的严格应用将会使Knight等人报告的所有错误在系统验收前消失。然而，存在这样的担心：规格说明的有些部分是复杂的，这将不可避免地导致所有独立小组都对需求缺乏理解。如果这些需求还引用很少出现的输入数据，那么可能就不能在系统测试期间抓住共同的设计错误。在最近几年里，Hatton（1997）的研究发现三版本系统的可靠性仍然大约是单版本高质量系统的可靠性的5～9倍。

3）**充足的预算**——大多数嵌入式系统的主要成本是软件。因此三版本系统的预算需求和带来的维护人员问题将几乎是三倍。在竞争的环境中，要潜在的承包商提议使用*N*版本技术是不太可能的，除非是强制性的。而且，如果将构造*N*版本的资源用于生产单版本系统，也不大清楚能否产生比*N*版本更可靠的系统。

一些实例还表明，找到不精确表决算法是很困难的，并且甚至在无故障的情况下表决都会不同，除非对于一致比较问题特别小心。

尽管*N*版本程序设计在生产可靠的软件中可能有作用，但应该小心地使用它，并且连同别的技术一起使用，例如，连同下面讨论的技术一起使用。

5.5 软件动态冗余

*N*版本程序设计是静态冗余或屏蔽冗余的软件等价物，在这里，对外界隐藏了部件中的故
114 障。因为软件的每个版本与其他版本和驱动器都有固定的联系，并且无论故障是否发生它都要运转，所以*N*版本程序设计是静态的。就**动态**冗余来说，当检测到错误时，冗余部件才刚刚开始工作。

这种容错技术有四个组成阶段（Anderson and Lee，1990）。

1）**出错检测**——大多数故障最后都将以某种形式的错误表明自己，直到检测到那个错误才可以应用容错方案。

2）**损害隔离和评估**（damage confinement and assessment）——当检测到错误时，必须确定系统被破坏到什么程度（这通常称为*出错诊断*），从故障的发生到相关错误的公开表明之间的延迟意味着可能在整个系统范围内传播了错误的信息。

3）**出错恢复**——这是容错最重要的方面之一。出错恢复应该旨在把被破坏的系统转换为一种能继续正常操作的状态（也许会伴随着功能的降级）。

4）**故障处理和继续服务**——错误是故障的征兆，尽管损坏可能被修复了，故障却可能依然存在，因此错误可能会重现，除非进行某种形式的维护。

尽管是在软件动态冗余技术下讨论容错的这四个阶段，但它们显然能够应用到*N*版本程序设计中。像Anderson和Lee（1990）已经注意到的那样：出错检测是由进行表决检查的驱动器提供的；不需要损害评估，因为版本是独立的；出错恢复包括抛弃错误的结果，而故障处理仅仅是忽略被确定会产生错误值的版本。然而，如果所有版本产生了不一致的表决，那么就进行出错检测，但是没有进行恢复的设施。

以下几节简要地覆盖以上容错的阶段。对这个问题更完整的讨论，读者可以参考文献[Anderson and Lee（1990）]。

5.5.1 出错检测

任何容错系统的有效性取决于它的出错检测技术的有效性。可以确定两种出错检测技术。

- **环境检测**。这些是在程序执行的环境中检测到的错误。它们还包括那些由硬件检测到的错误，例如“执行非法指令”、“算术溢出”和“保护违规”。它们还包括由实时编程语言的运行时支持系统检测出的错误，例如，“数组界限出错”、“引用空指针”和“数值超出范围”。第6章里在Ada和Java程序设计语言中将讨论这些错误类型。 115
- **应用检测**。这些是由应用程序自己检测到的错误。应用程序使用的大多数技术分成以下几大类:
  - **复制检查**。已经证明N版本程序设计能用来进行软件故障的容错，并且这项技术可以用来提供出错检测（通过使用二版本冗余）。
  - **定时检查**。已经确定两种类型的定时检查。第一种包括**看门狗定时器**（watchdog timer）进程，如果部件没有在一个确定的时期内重置这个定时器，就假定这个部件存在错误。软件部件必须不断地重置这个定时器来表明它在正常工作。

    在及时响应至关重要的嵌入式系统中，需要使用第二种检查类型。这使故障的检测与错过的时限相关联。在时限调度由底层运行时支持系统执行的地方，时限错过的检测可以被认为是环境的一部分（有关时限调度的一些问题将在第13章讨论）。

    当然，定时检查仅仅能保证部件在按时工作，而不能确保部件在正常地工作！因此定时检查应该与其他的出错检测技术一起使用。
  - **反向检查**。这在输入和输出保持一对一（同构的）关系的部件中是可行的。这种检查取出输出，计算输入应该是什么，然后将计算出的值与实际的输入比较。例如，对一个计算算术平方根的部件，反向检查只是将输出做平方运算，然后和输入作比较（注意：当处理实数时可能必须使用不精确比较技术）。
  - **编码检查**。编码检查用来测试数据的损坏，它基于包含在数据里的冗余信息。例如，可以计算出一个值（校验和）并且和实际数据一起通过通信网络发送，当收到数据时，可以重新计算这个值并与校验和比较。
  - **合理性检查**。这是基于内部设计和系统构造知识的检查。合理性检查就是基于预期用途检查数据的状态或对象的值是否合理。对于现代实时语言，完成这些检查所需的大量信息可以由程序员提供（作为与数据对象相关的类型信息）。例如，被限制在确定值以内的整型对象可以用有明确范围的整数的子类型表示，这样运行时支持系统就可以检测到范围违规。

    有时，软件部件中包含显式的合理性检查，这通常称为**断言**，并且采用逻辑表达式的形式。如果没有检测到错误，就把在运行时求值的逻辑表达式的值置为真。 116
  - **结构检查**。结构检查常用来检查数据对象（例如，表或队列）的完整性。它可能由对象里元素的个数、冗余的指针或额外状态信息组成。
  - **动态合理性检查**。在一些数字控制器发出的输出中，通常两个连续输出之间有一定的联系。如果一个输出与前面的值相差很远就可以假定有错误发生。

注意：以上的许多技术也可以应用于硬件中，因此可能导致“环境错误”。

5.5.2 损害隔离和评估

由于在故障发生和检测到错误之间可能存在一定的延迟，因此评估可能发生的损害是很有必要的。虽然检测到的错误类型将给错误处理例程带来一些损害信息，但错误的信息可能已经传播到整个系统中，并且进入它的环境中。因而损害评估将和由系统设计者采取的损害

隔离预防措施密切相关。损害隔离与构造系统有关以使由故障部件引起的损害减到最小，它还被称作**防火墙**。

有两种技术可用于结构化系统以辅助损害隔离：**模块分解**和**原子动作**。在第4章已经讨论了模块分解的优点。在这里，重点是系统应该划分成部件，每个部件可以用一个或几个模块表示，然后各部件之间通过明确定义的接口交互，隐藏模块内部的细节并且不能从外部直接访问这些细节。这使一个部件中的错误任意地传入另一个部件变得更加困难。

模块分解为软件系统提供了*静态*结构，因为在运行时大多数结构都丢失了。系统的*动态*结构对损害隔离来说同样重要，因为它使软件的运行时行为的推理更容易。一种重要的动态结构化技术是基于原子动作的使用。

如果在部件的活动期间，该活动和系统之间没有交互，就说这个活动是原子的。

这就是说，对系统的其他部分而言原子动作可认为是*不可分的*并且是*瞬时发生的*。在原子动作和系统其他部分之间不能传递信息，反之亦然。原子动作通常称为**事务**或**原子事务**，它们常用来将系统从一个一致的状态移至另一个一致的状态并且约束部件之间的信息流。当两个或两个以上部件共享一个资源时，那么损害隔离将包括限制对那个资源的访问。原子动作的这方面的实现使用了在现代实时语言中出现的通信和同步原语，这些将在第10章讨论。

117

试图限制访问资源的其他技术都是基于**保护机制**的，其中的一些技术可以由硬件支持。例如，每个资源可能有一种或一种以上的运行模式，每个模式都有相关联的访问列表（例如，读、写和执行）。部件的活动或进程也将会有一个相关联的模式。每当进程访问一处资源时，可以将预定的操作与它的**访问许可**（access permission）进行比较，如果有必要，可以拒绝访问。

5.5.3 出错恢复

一旦检测到了错误状态并且进行了损害评估，就必须开始出错恢复过程。这可能是任何容错技术的最重要的阶段。出错恢复必须把错误的系统状态转变成能够继续使系统正常操作的状态，虽然可能使服务降级。已经提出了两种出错恢复的方法：**向前**恢复和**向后**恢复。

向前出错恢复试图通过对系统状态做选择性的矫正使它可以从错误的状态继续。由于失效，受控环境的某些方面可能是危险的或是损坏了，对嵌入式系统来说，向前恢复可能包括使受控环境的这些方面都成为安全的。虽然向前出错恢复可以是高效的，但它却是系统专用的，并依赖于对错误位置和起因的准确预测（也就是损害评估）。向前恢复技术的例子包括数据结构中的冗余指针和自动矫正码（比如Hamming码）的使用。如果当错误发生时有一个以上的进程提供服务，那么在恢复动作期间可能还需要中止或异步异常设施。

向后出错恢复把系统恢复到错误发生以前的安全状态，然后执行程序的一个替代段。这个替代段与产生故障的段有相同的功能，但使用不同的算法。同*N*版本程序设计一样，希望这种替代的方法将不会导致相同故障的复发。恢复进程的点称作**恢复点**（recovery point），通常把建立恢复点的行为称为**设立检查点**（checkpointing）。为了建立恢复点，必须保存系统在运行时的适当状态信息。

状态恢复的优点是清除了错误的状态，以及不依赖于找到故障的位置或起因。因此向后出错恢复可以用来恢复没有预料到的故障，包括设计错误。然而，向后恢复的缺点是：它不能撤消在嵌入式系统的环境中故障可能已经产生的任何影响，例如很难撤消一次导弹发射。此外，向后出错恢复在执行过程中可能是耗时的，这可能妨碍了它在一些实时应用程序中的

使用。例如，包含传感器信息的操作可能是依赖于时间的，因此昂贵的状态恢复技术可能完全不可行。所以，可以考虑用**逐步设立检查点**（incremental checkpointing）的方法来改善性能。**恢复缓存**（recovery cache）就是这种系统的一个例子（Anderson and Lee，1990）。其他的方法包括审计追踪或系统日志，在这些情形中，底层支撑系统必须通过反转日志中指出的动作来撤消进程的影响。 118

对于相互作用的并发进程，状态恢复不像迄今为止描绘的那样简单。考虑图5-6中描绘的两个进程。进程P_1建立恢复点R_{11}、R_{12}和R_{13}，进程P_2建立恢复点R_{21}和R_{22}。并且，两个进程通过IPC_1、IPC_2、IPC_3和IPC_4通信并使它们的动作同步。缩写IPC用来表示进程间通信。

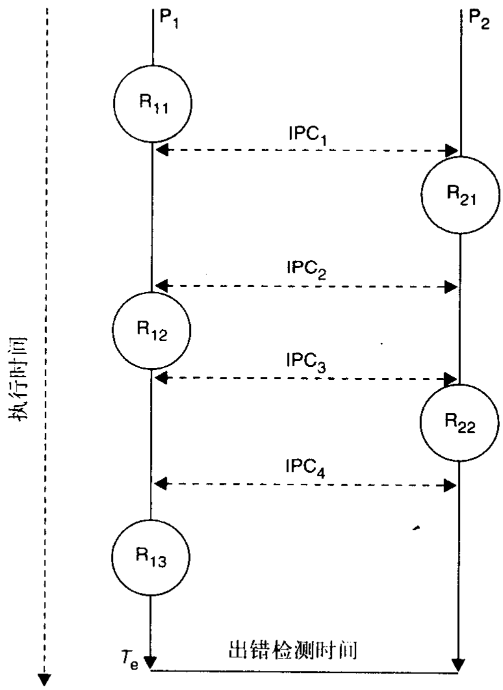


图5-6 多米诺效应

如果P_1在T_e时刻检测到错误，那么它只需要回卷到恢复点R_{13}即可。然而，考虑一下P_2在T_e时刻检测到错误的情形。如果P_2回卷到R_{22}，那么它必须撤消同P_1的通信IPC_4，这又需要P_1回卷到到R_{12}。但是如果P_1回卷到到R_{12}，P_2必须回卷到R_{21}来撤消通信IPC_3，如此类推。结果将是两个进程将回卷到到它们开始交互的位置。在许多情形下，这么做相当于中止两个进程！这种现象被称作**多米诺效应**。 119

显然，如果两个进程互相没有交互，那么就不会出现多米诺效应。当两个以上的进程交互时，多米诺效应发生的可能性会增加。在这种情况下，必须在系统中设计一致的恢复点，使得在一个进程中检测的错误不会导致与该进程交互的所有进程的全部回卷，相反，可以从一组一致的恢复点重新启动这些进程。这些通常称为**恢复线**（recovery line），它与本节前面介绍过的原子动作的概念联系紧密。第10章中将再次讲到并发进程中的出错恢复问题，在本章余下的部分中将只考虑顺序系统的情况。

已经介绍了向前恢复和向后恢复的概念，每种恢复都有它们的优缺点。嵌入式系统不仅

必须能从不可预见的出错恢复，它们还一定要在限定的时间内响应。所以，向前恢复和向后恢复这两种技术它们可能都需要。下一节将研究向后出错恢复在顺序实验性程序设计语言中的表达问题。在这一章中将不深入探讨向前出错恢复机制，因为用一种同应用无关的方式提供它是很困难的。然而，下一章在异常处理的共同框架内会研究两种形式的出错恢复的实现问题。

5.5.4 故障处理和继续服务

错误是故障的表现，尽管出错恢复阶段可能使系统回到无错误状态，但错误有可能重现，因此容错的最后阶段是从系统中根除故障，使正常的服务能够继续。

故障的自动处理是很难实现的，且倾向于系统专用的处理。所以，一些系统没有提供故障处理，它们假定所有的故障都是瞬时的；另一些系统假定出错恢复技术十分强大足以对付故障的再次发生。

故障处理可以分为两个阶段：故障定位和系统修复。错误检测技术有助于追踪部件的故障。对硬件部件来说这可能足够准确，并且部件可以简单地替换。在代码的新版本中可以消除软件故障，但对大多数不间断运行的应用程序而言，必须在运行状态下修改程序。这提出了一个有意义的技术问题，但在这里不深入探讨它。

5.6 软件容错的恢复块方法

恢复块（Horning等，1974）就是一般程序设计语言意义上的块，不同的是在块的入口有一个自动的**恢复点**，并且在出口有**接受测试**（acceptance test）。接受测试用以测试在块（或基
120 **本模块**）执行后系统是否处于可接受的状态。接受测试的失败会导致程序恢复到块起点上的恢复点并执行一个**替代模块**。如果替代模块的接受测试也失败了，那么程序又一次恢复到恢复点并执行另一个模块，如此类推。如果所有的模块都失败了，那么这个块就失败了，并且要在更高级别上进行恢复。图5-7说明了恢复块的执行过程。

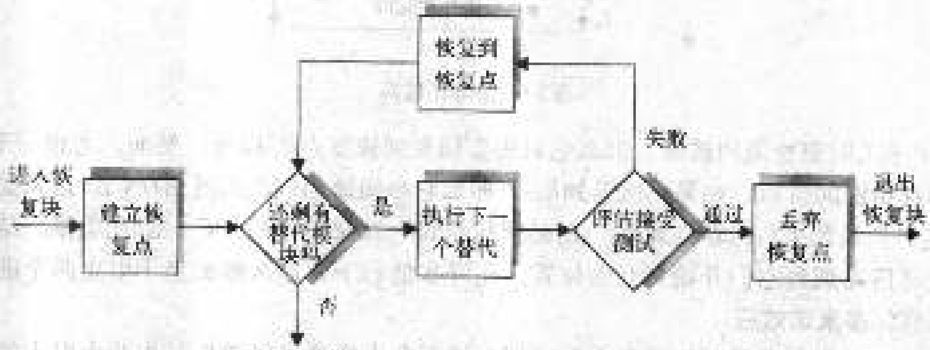


图5-7 恢复块机制

就软件容错的四个阶段来说，出错检测是通过接受测试完成的，损害评估是不需要的，因为向后出错恢复假定要清除所有错误的状态，故障处理是通过使用待用替换品实现的。

虽然市场提供的实时程序设计语言都没有利用恢复块的语言特征，还是开发了一些实验系统（Shrivastava，1978）（Purtilo and Jalote，1991）。以下说明了一种用于恢复块的可能的

语法:

```
ensure <接受测试>
by
  <基本模块>
else by
  <替代模块>
else by
  <替代模块>
  ...
else by
  <替代模块>
else error
```

121

和普通的块一样，恢复块也可以嵌套。如果在嵌套恢复块中的块的接受测试失败了，并且它所有的替代模块也失败了，那么将会恢复外层的恢复点，并且执行那个块的一个替代模块。

为了说明恢复块的用途，我们考虑各种用来得到微分方程的数值解的方法。由于这些方法并不给出精确的解，还易受各种错误的支配，可以发现一些方法对某些种类的方程要比对其他的方程表现得更好。令人遗憾的是，根据完成方法的执行需要的时间，实现那些适用面宽又给出精确解的方法是很昂贵的。例如，**显式Kutta**方法比**隐式Kutta**方法更有效。然而，它只能为特定问题给出可接受的误差公差。有一类方程称作**刚性**方程，它的使用显式Kutta方法的解导致了舍入误差的积累，而更昂贵的隐式Kutta方法可能更足以处理这个问题。下面说明一种使用恢复块的方法，它能够使非刚性方程采用更便宜的方法，但是当求解刚性方程时它也不会失败。

```
ensure 舍入误差在容许范围内
by
  显式 Kutta 方法
else by
  隐式 Kutta 方法
else error
```

在这个例子中，通常使用的是较便宜的显式方法，然而，当显式方法失败时就采用更昂贵的隐式方法。虽然这种误差是预期的，这种方法在显式算法的设计中仍然容许有误差。如果算法本身存在错误，并且接受测试足够全面可以检测两种类型的错误结果，就使用隐式算法。当接受测试不够全面时可以使用嵌套的恢复块。下面提供了完全的设计冗余度，同时总是尽可能使用更便宜的算法。

```
ensure舍入误差在容许范围内
by
  ensure 合理值
  by
    显式 Kutta 方法
  else by
    预测器-矫正器 K步方法
  else error
else by
  ensure 合理值
  by
```

```
      隐式 Kutta 方法
    else by
      变阶K步方法
    else error
  else error
```

122

上面给出了两种显式方法，当两种方法都没能产生切合实际的结果时就执行隐式Kutta方法。当然，如果由显式方法产生的值是合理的，但却不在要求的容许误差之内，也将执行隐式Kutta方法。只有当四种方法都失败时，方程才算没有解出来。

下面说明恢复块可以以另一种方式嵌套。在这种情况中，当不合理的结果也不在可接受的容许误差之内时将发生不同的行为。在第一种情况下，执行显式Kutta算法后将尝试预测器－矫正器法。在第二种情况下，将执行隐式Kutta算法。

```
ensure 合理值
by
  ensure舍入误差在容许限度内
  by
    显式 Kutta 方法
  else by
    隐式 Kutta 方法
  else error
else by
  ensure舍入误差在容许限度内
  by
    预测器-矫正器 K步方法
  else by
    变阶K步方法
  else error
else error
```

接受测试

接受测试提供出错检测机制，它使系统能够利用冗余。接受测试的设计对恢复块方案的功效至关重要。同所有的出错检测机制一样，在提供全面的接受测试和保持接受测试需要的系统开销最小之间有一个折衷方案，使得对正常无故障执行的影响尽可能小。注意，使用的术语是**接受性**而不是**正确性**，这就允许部件提供一种降级的服务。

5.5.1小节中讨论的所有出错检测技术可以用来形成接受测试。然而，它们的设计必须小心，因为有故障的接受测试可以导致检测不到残留的错误。

5.7 N版本程序设计和恢复块的比较

123 已经描述了提供容错软件的两种方法：*N*版本程序设计和恢复块。显然，它们共享基本原理的某些方面，但同时又是十分不同的。本节简要地评述并比较这两种方法。

- **静态冗余和动态冗余**

*N*版本程序设计是基于静态冗余的，不管是否有故障发生，所有的版本都是并行运行的。相反，恢复块是动态的，因为只有当检测到错误时才执行替代模块。

- **相关的开销**

*N*版本程序设计和恢复块都带来额外的开发费用，因为两者都需要开发替代算法。另外，

对于N版本程序设计，必须设计驱动器进程，而恢复块需要设计接受测试。

在运行时，N版本程序设计通常需要N倍于单个版本的资源。虽然恢复块在任何一个时刻只需要一套资源，恢复点的建立和状态恢复进程是昂贵的。然而，对恢复点的建立有可能提供硬件支持（Lee等，1980），并且只有当出现故障时才需要状态恢复。

- **设计多样性**

两种方法为了实现容许不可预见错误都利用了多样性设计。因此，两种方法易受到来自需求规格声明的错误的影响。

- **出错检测**

N版本程序设计用表决比较来检测错误，而恢复块则使用接受测试。在可以进行精确或不精确表决的地方，与之相关的开销要比使用接受测试的少。然而，在很难找到不精确表决技术的地方，在存在多种解决方案或有一致比较问题的地方，接受测试可以提供更大的灵活性。

- **原子性**

向后出错恢复受到批评是因为它不能撤消在环境中可能已经发生的任何损害。N版本程序设计避免了这个问题，因为假定所有的版本互不干涉：它们是原子的。这要求每个版本同驱动进程通信而不是直接和环境通信。然而，完全可以构造一个程序使不可恢复的操作不在恢复块中出现。

可能应该强调的是：尽管N版本程序设计和恢复块被说成是相互竞争的方法，还是可以认为它们是互补的。例如，没有什么可以阻止设计者在N版本系统的每一个版本中使用恢复块技术。 124

5.8 动态冗余和异常

在这一节里，要介绍实现软件容错的一种框架，它基于动态冗余和异常以及异常处理程序的概念。

到这一章为止，术语“错误”都是用来指出故障的表现，只要故障是偏离了部件的规格说明。这些错误可以是预见到的（例如，由硬件故障引起的传感器读数超出范围），或是预见不到的（例如，部件中的设计错误）。**异常**可以定义为错误的发生。使引起异常的操作的调用者注意到异常状态，称为**引发**（或**发信号**或**抛出**）异常并且把调用者的响应称为**处理**（或**捕获**）异常。异常处理可以认为是*向前出错恢复机制*，因为引发异常时系统没有返回到先前的状态，相反，控制权交给了处理程序以便可以启动恢复过程。然而，就像将在6.5节说明的那样，异常处理功能还可以用来提供向后出错恢复。

尽管已经把错误的发生定义为异常，仍然有关于异常的真正性质和什么时候使用异常的争论。例如，研究一个维护编译器符号表的软件部件或模块，它提供的操作之一是查找符号。查找有两种可能的结果：*符号存在*和*符号不存在*。每一种结果都是可以预见的并且有可能表现一个错误状态。如果*查找*操作是用来决定符号在程序体中的解释，那么*符号不存在*就对应于“未声明的标识符”，这是一个错误状态。然而，如果在声明过程中使用*查找*操作，*符号不存在*的结果也许是正常情况，而*符号存在*就是“重复定义”异常。因此，什么才是错误取决于事件发生的上下文。然而，在以上的任何一种情况中都会产生这样的争论：错误并不是符

号表部件或编译器的错误，因为每种结果都是可预见的并且形成了符号表模块功能的一部分，因此两种结果都不应该表示成异常。

异常处理设施过去没有被合并到程序设计语言中来针对程序员的设计错误，然而，在6.5节中将介绍它们可以如何用来这么做。引入异常的最初动机是来自处理程序执行环境中出现的不正常状态的需要。这些异常可以称作环境运行中的稀有事件，在程序里面也许能从异常恢复，也许不能。一个有故障的阀门或一个温度报警器可能产生异常。给定了足够时间，这些稀有事件就可能发生，因此必须容许。

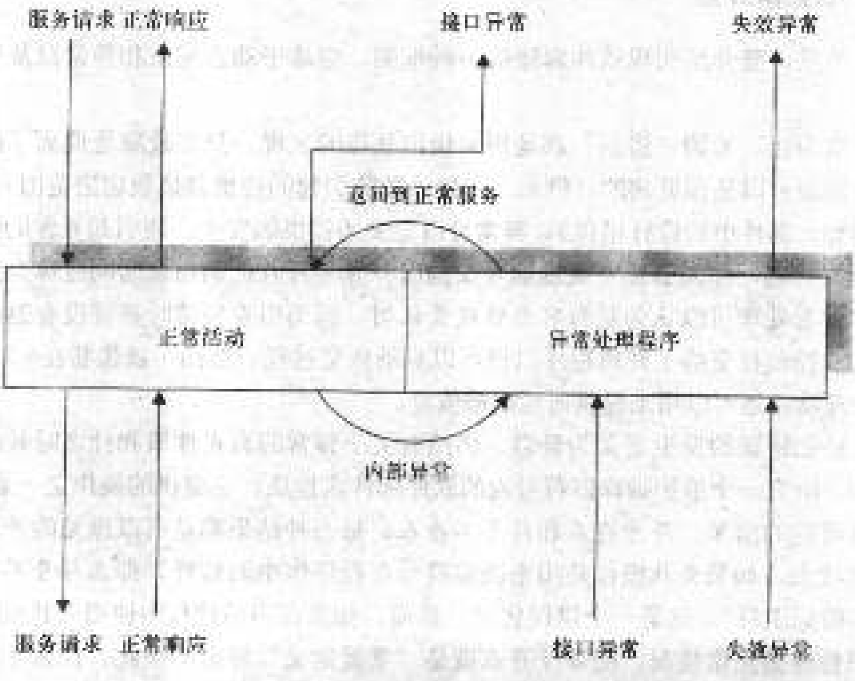
尽管如此，异常和它们的处理程序将不可避免地当作通用的出错处理机制来使用。总之，异常和异常处理可以用来：

- 应付出现在环境中的不正常状态
125 - 使得能够容忍程序设计故障
- 提供通用的出错检测和恢复设施

在第6章中会更详细地讨论异常。

理想的容错系统部件

图5-8显示了建立容错系统的理想部件（Anderson and Lee，1990）。此部件接受服务请求，并且如果必要的话，它会在产生响应之前召唤其他部件的服务。这可以是正常响应或是异常响应。在理想的部件中可能产生两种类型的故障：一类是那些由非法服务请求引起的故障，称作**接口异常**；另一类是那些由部件本身的误动作或为初始请求服务所需的部件中引起的故障。当部件无论使用向前出错恢复或向后出错恢复都不能容忍这些故障时，在调用部件中就产生了**失效异常**。在引发任何异常之前，部件必须尽可能将自身返回到一致状态，以便它能够为任何将来的请求服务。



126 图5-8 理想的容错部件

5.9 软件可靠性的测量和预测

硬件部件的可靠性度量已经建立了很长时间。传统上，把每个部件看作是同样成员的总体的一个代表，部件的可靠性是由测试时失效的样本的比例来估算的。例如，已经观察到，在初始的稳定时期之后，电子部件是按固定比率失效的，它们在t时刻的可靠性可以建模为:

$$R(t)=Ge^{-\lambda t}$$

这里G是常量，λ是所有组成部件的失效率的总和。通常使用的度量是平均失效间隔时间（Mean Time Between Failure，MTBF），无冗余系统的平均失效间隔时间等于$1/\lambda$。

软件可靠性的预测和测量还不是一个完善的学科。它被那些需要极其可靠系统的行业忽视了很多年，因为软件不因使用而损坏，它被视为要么可靠要么不可靠。并且，在过去，特定的软件部件只在事先预定了它们的系统中使用一次，因此，尽管消除了所有在测试期间发现的错误，但是这并不能导致开发出能在别处使用的更可靠的部件。这同大规模生产的硬件部件形成反差，后者可以纠正设计中发现的任何错误，使下一批更可靠。然而，现在认识到通过部件的重用，软件的可靠性是可以改善的，并且可以减少软件成本。令人遗憾的是，我们对怎样建立可重用的软件部件的理解还很不全面（参见4.5.1节）。

人们仍然普遍持有这种观点：软件要么是正确的，要么是不正确的。如果是不正确的，程序测试或程序验证将指出故障的位置，然后予以改正。本章试图说明：传统的软件测试方法（尽管是必不可少的）不能确保程序是无故障的，特别是那些残留着规格说明错误或设计错误的大型而又复杂的系统。并且，尽管在正确性证明领域有了飞速的进展，当前的技术水平对于那些非同小可的系统，特别是那些涉及时间概念的系统的应用仍然是力不从心的。正是这些原因，使我们要提倡通过使用冗余来改善可靠性的方法。令人遗憾的是，即使用这种方法也不能保证包含软件的系统不会失效。因此，发展预测或测量软件可靠性的技术是极其重要的。

可以认为软件可靠性是*给定的程序在指定的环境和指定时间长度内正确运行的概率*。已经提出了试图估计软件可靠性的几种模型，这些模型可以分为以下几类（Goel and Bastini，1985）:

- 软件可靠性增长模型
- 统计模型

增长模型试图在程序错误历史的基础上预测程序的可靠性。统计模型试图通过在不纠正发现的任何错误的情况下，确定程序对测试用例的一个随机样本的成功或失败响应来估计程序可靠性。特别是增长模型现在十分成熟，并且关于它们的应用有大量的文献和产业经验（Littlewood and Strigini，1993; Bennett，1994; Lytz，1995）。然而，Littlewood和Strigini（1993）争论说：单靠测试只能为最多10^{-4}（更精确地说，是每操作小时10^{-4}个错误）的可靠性估计提供证据。这应该与航空电子和核能应用通常引用的10^{-9}的可靠性需求进行比较。 127

5.10 安全性、可靠性和可依赖性

术语“安全性”（在第2章给过它非正式的定义）可以结合这一章出现的问题来延伸它的含义。这样，安全性可以定义为：*摆脱那些可能造成人员伤亡、职业病、设备损坏（或财产*

损失）或环境损害的状态（Leveson，1986）。然而，由于这个定义考虑到那些存在着与不安全使用相关的风险元素的大多数系统，所以软件安全性通常按照**事故**（mishap）来定义（Leveson，1986）。事故就是可能造成人员伤亡、职业病、设备损坏（或财产损失）或环境损害的**一个或一系列意外事件**。

虽然可靠性和安全性经常被认为是同义语，但它们强调的重点不同。可靠性被定义为系统成功符合其行为的权威性规格说明的测度。通常这用概率表示。而安全性是导致事故不发生的概率，*无论预期功能是否执行*。这两种定义可能互相冲突。例如，提高武器射击的可靠性的措施可能也会增加它意外爆炸的可能性。在许多方面，惟一安全的飞机就是从不起飞的飞机，然而，它也不是非常可靠的。

与可靠性一样，要确保嵌入式系统的安全性要求，系统安全分析必须贯穿它生命周期开发的所有阶段。进行安全分析的详细描述超出了本书的范围，对于软件故障树分析——一种用来分析软件设计安全性的一种技术——的一般性讨论，读者可以参考文献Leveson and Harvey（1983）。

可依赖性

在过去的十年间，在可靠性和容错计算这个领域进行了许多的研究。结果，这个术语已经变得过载了，研究人员们要寻找表达他们希望强调的特殊方面的新词和短语。术语“安全
128 性（safety）”和“保密性（security）”就是这种新术语的例子。为此，已经试图对此领域提出的基本概念建立清楚而又广泛接受的定义。为此，引入了**可依赖性**（dependability）的概念（Laprie，1985）（Laprie，1995）。

*系统的可依赖性就是系统使得它交付的服务能被信任的性质。*因此可依赖性包括可靠性、安全性和保密性等概念作为其特殊情况。图5-9以Laprie（1995）所给的图为基础，说明了这些特性和可依赖性的其他方面（这里认为保密性是完整性和机密性）。在图中，可靠性是连续交付正确服务的测度；可用性是不正确服务周期的频率的测度。

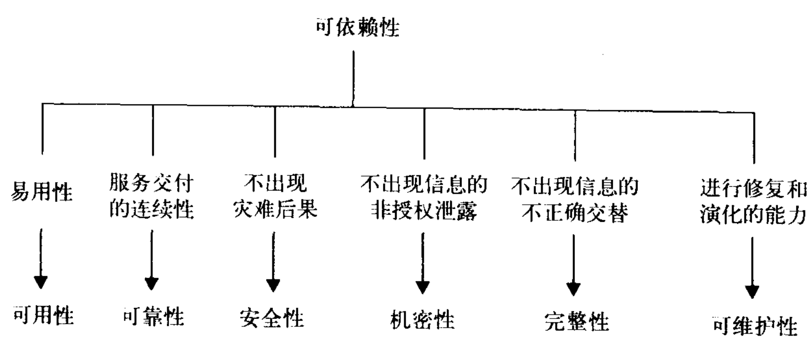


图5-9 可依赖性的各个方面

可以用以下三个部分来描述可依赖性（Laprie，1995）。

- 损伤——由不可依赖性产生或引起的情况；
- 手段——以要求的可信度交付可信赖的服务需要的方法、工具和解决方案；
- 属性——用以评价可信赖服务的方法和测度。

图5-10按这三个部分概括了可依赖性的概念。

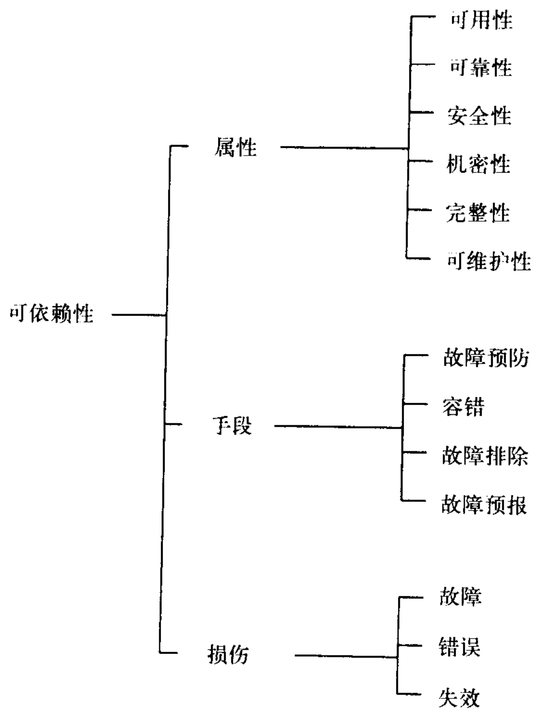


图5-10 可依赖性术语

尽管对可依赖性这个术语有某种一致意见，但离一致同意还差得很远，所以这个术语还在提炼之中。因此本书将继续使用本章主体部分使用的那些已经确定的名称。

小结

本章把可靠性确定为任何实时系统的主要需求。把系统的可靠性定义为系统关于其行为的权威性规格说明的符合程度的度量。当系统行为偏离了为它规定的行为时，就称作失效。失效是由故障引起的。把故障引入到系统中可以是有意的或是意外的。故障可能是瞬时的、永久的或间歇的。

有两种帮助确保潜在的故障不引起系统失效的系统设计方法：故障预防和容错。故障预防由故障回避（试图限制在系统中引入有故障的部件）和故障排除（找到并排除故障的过程）两部分组成。容错包括在系统中引入冗余部件以便可以检测故障和容错。通常系统会提供完全容错、性能降低或故障保护这三种行为。

本章还讨论了软件容错的两种普遍方法：N版本程序设计（静态冗余）和使用向前和向后出错恢复的动态冗余。N版本程序设计定义为从同一初始规格说明独立地产生N个（这里N大于或等于2）功能上等价的程序。一旦设计并且编写完成，这N个程序就以相同的输入并发地执行并且比较它们的结果。原则上这些结果应该是相同的，但实际中它们可能有差别，在这种情况下认为一致的结果（假定有一个）是正确的。N版本程序设计基于如下假设：能够完全地、一致地并且无歧义地指定一个程序并且独立开发的程序独立地失效。这些假设并不是总是有效，虽然N版本程序设计在生产可靠的软件中可能有其作用，但应该小心地使用它，并且连同基于动态冗余的其他技术一起使用。 129~130

动态冗余技术有四个组成阶段：出错检测、损害隔离和评估、出错恢复以及故障处理和继续服务。作为一种有助于损害隔离的结构化技术，本章还引入了原子动作的概念。一个最重要的阶段是出错恢复，已经建议了两种方法：向后恢复和向前恢复。为了使相互通信的进程达到一致的恢复点以避免多米诺效应，使用向后恢复是必要的。对于顺序系统，把恢复块作为适当的语言概念引入以表达向后恢复。恢复块就是一般程序设计语言意义上的块，不同的是在块的入口有一个自动的恢复点，并且在出口处有一个接受测试。接受测试常用来测试在主模块执行后系统是否处于可接受的状态。接受测试的失败会导致程序恢复到块起点上的恢复点并执行替代模块。如果替代模块的接受测试也失败了，那么程序又一次恢复到恢复点并且还要执行另一个模块，如此类推。如果所有的模块都失效了，那么这个恢复块就失效了。N版本程序设计和恢复块之间的比较说明了两种方法的相同点和不同点。

虽然向前恢复是系统专用的，但是异常处理被证明是实现向前恢复的合适框架。本章还介绍了一个使用异常的理想容错部件的概念。

在本章的最后，介绍了软件安全性和可依赖性的概念。

相关阅读材料

Anderson, T. and Lee, P. A. (1990) *Fault Tolerance, Principles and Practice*, 2nd edn. Englewood Cliffs, NJ: Prentice Hall.

Laprie J.-C. et al. (1995) *Dependability Handbook*. Toulouse: Cépaduès (in French).

Leveson, N. G. (1995) *Safeware: System Safety and Computers*. Reading, MA: Addison-Wesley.

131 Mili A. (1990) *An Introduction to Program Fault Tolerance*. New York: Prentice Hall.

Neumann, P. G. (1995) *Computer-Related Risks*. Reading, MA: Addison-Wesley.

Powell, D. (ed.) (1991) *Delta-4: A Generic Architecture for Dependable Distributed Computing*. London: Springer-Verlag.

Randell, B., Laprie, J.-C. , Kopetz, H. and Littlewood, B., (eds) (1995) *Predictable Dependable Computing Systems*. London: Springer.

Redmill, F. and Rajan, J. (eds) (1997) *Human Factors in Safety-Critical Systems*. Oxford: Butterworth-Heinemann.

Storey, N. (1996) *Safety-Critical Computer Systems*. Reading, MA: Addison-Wesley.

还有由Kluwer Academic出版社出版并由G. M. Koob and C. G. Lau编辑的关于“可依赖系统的基础”（Foundations of Dependable Systems）的系列丛书。

练习

5.1 如果程序的行为符合一个有错误的规格说明，那么这个程序是可靠的吗？

5.2 对计算机控制汽车来说，什么是降级服务的合适的标准？

5.3 写出对整数数组排序的恢复块。

5.4 可以在运行时检测恢复线到什么程度？（参见Anderson and Lee（1990），第7章）

5.5 图5-11说明了四个通信进程（P1，P2，P3和P4）的并发执行过程和它们相关的恢复点（例如，R11是进程P1的第一个恢复点）。解释由

（1）进程P1在时刻t

（2）进程P2在时刻t

检测到错误时会发生什么？

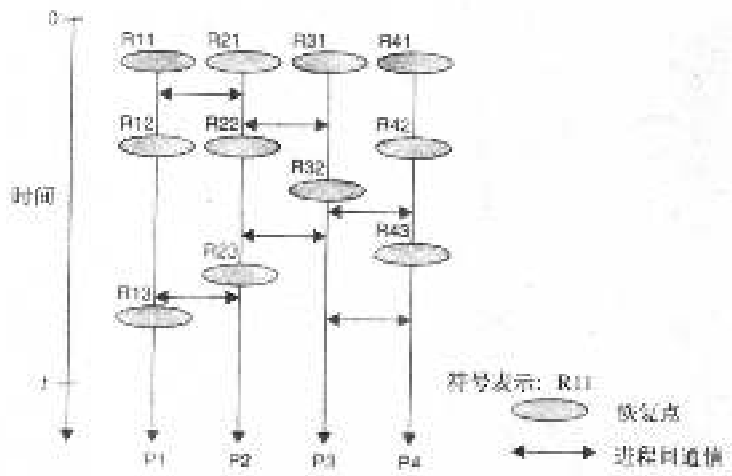


图5-11 练习5.5 四个进程的并发执行

5.6 当顺序读取文件时出现的文件结束状态是否应该作为异常来通知程序员？

5.7 数据多样性是补充设计多样性的一种容错策略。在什么情况下数据多样性比设计多样性更合适？（提示：参见参考文献[Ammann and Knight（1998）]） 132-133

5.8 系统的可依赖性是否应该由独立的评估员来评判？

第6章 异常和异常处理

第5章研究了怎样才能使系统更加可靠，并将异常作为实现软件容错的框架做了介绍。本章更详细地研究异常和异常处理，并讨论它们在各种具体实时编程语言中的情况。

对于异常处理设施有若干一般需求:

(R1) 像所有语言特征一样，这种设施理解和使用起来必须简单。

(R2) 异常处理代码不应过分突出，以致模糊了对程序的正常无错运行的理解。那种将正常处理代码和异常处理代码混合到一起的机制是难于理解和维护的，它很可能导致不太可靠的系统。

(R3) 这种机制应被设计得只在处理异常时才会有运行时开销。虽然大部分应用要求使用异常的程序的性能在正常操作状态下没有不利的影响，但并不总是这样。在某些情况下，特别是恢复速度具有根本重要性的情况下，应用可能只能容忍在正常无错运行时的很少的开销。

(R4) 这种机制应当能够统一处理由环境和由程序检测到的异常。例如，像**运算溢出**(arithmetic overflow) 这种由硬件检测的异常和程序的断言失败引发的异常应当以相同的方式处理。 135

(R5) 如在第5章中提到过的，这种异常机制应使恢复动作可被编程。

6.1 老式实时语言中的异常处理

虽然"异常"和"异常处理"这些术语只在最近才流行，但它们只不过表达了一种试图容许并处理出错情况的编程方法。所以，大多数编程语言都有能够至少处理某些异常的设施。本节简明地按上面设定的需求评估这些设施。

6.1.1 反常返回值

异常处理机制的一种最基本形式是来自过程或函数的*反常返回值*或*出错返回*。它的主要优点是简单，并且实现起来不需要任何新的语言机制。C支持这种方法，其典型用法如下:

```
if (函数调用 (参数) == AN_ERROR) {
  /* 出错处理代码 */
} else {
  /* 正常返回代码 */
};
```

可见，虽然它满足简单性需求R1，并使恢复动作可被编程 (R5)，却不满足R2、R3和R4。

代码是突出的，每次使用时都产生开销，并且不清楚怎样处理环境检测到的错误。

在本书中，C是同POSIX联合使用的。POSIX的出错状态是由非零返回值指示的（还带有符号名）。为了可靠性，对系统功能的每个调用应该检验返回值，以确保没有非预期错误发生。然而，如上面说明的，这可能使代码结构含糊。所以为教学目的，本书使用POSIX风格的接口。对于每个POSIX系统调用，假设有一个已经定义的宏进行出错检查。例如，一个名为sys_call的系统调用（它有一个参数）有如下的自动定义的宏：

```
# define  SYS_CALL (A) if (sys_call (A) != 0) error ()
```

136 其中error是进行出错处理的函数，所以，所示代码就是SYS_CALL (param)。

6.1.2 强迫性分支

在汇编语言中，异常处理的典型机制是子例程的*跳返回*（skip return）。换句话说，直接跟在子例程调用后面的指令被跳过，这用以指出错误的出现（或不出现）。做到这一点的办法是子例程给其返回地址（程序计数器）增加简单跳传指令的长度，以指出无错（有错）返回。在可能有一个以上的异常性返回的时候，子例程假设调用者在其调用的后面有一个以上的跳传指令，并相应地处理程序计数器。

例如，假设两个可能的出错状态，可用下面的代码调用一个子例程，它向设备输出字符。

```
jsr pc, PRINT_CHAR
jmp IO_ERROR
jmp DEVICE_NOT_ENABLED
# 正常处理
```

对于正常返回的子例程，将使返回地址增加两个jmp指令。

虽然这种方法带来的开销小（R3），并使恢复动作可被编程（R5），它却可能导致程序结构含糊，因而破坏R1和R2需求，R4也可能不满足。

6.1.3 非局部goto

有强迫转移的高级语言版本可能需要把不同的标号作为参数传递给过程或者有一些标准标号变量（标号变量是可以赋以程序地址的对象，并可用于传输控制）。RTL/2是一个早期实时语言的例子，它以非局部goto的形式提供后一种设施。RTL/2使用**砖块**（brick）构造其程序：砖块可以是数据（由关键字data enddata包住）、过程（由proc endproc包住）或栈（由stack关键字标识）。系统定义的数据砖块的专门类型叫做svc数据。有一个砖块（rrerr）提供标准的出错处理设施，它包含一个错误标号变量，名为erl。

下面的例子说明RTL/2如何使用错误标号。

```
svc data rrerr
  label erl; %一个标号变量%
  ...
enddata

proc WhereErrorIsDetected ();
  ...
  goto erl;
  ...
endproc;

proc Caller ();
```

137

```
    ...
    WhereErrorIsDetected ();
    ...
  endproc;

  proc main ();
    ...
  restart:
    ...
  erl := restart;
    ...
    Caller ();
    ...
  end proc;
```

注意，当以这种方式使用时，goto就不止是跳传，它隐含着从过程的非正常返回。所以，栈必须延伸到恢复的环境——就是包含该标号的声明的那个过程的环境。栈延伸的不利后果只在发生错误时才会有，所以满足了需求R3。虽然goto的使用很灵活（满足R4和R5），但它们可能导致很含糊的程序，所以不满足R1和R2。

6.1.4 过程变量

虽然RTL/2的例子说明了怎样使用错误标号进行出错恢复，但程序的控制流已经给破坏了。在RTL/2中，错误标号通常用于不可恢复的出错，而在控制应当返回到出错点的时候使用**出错过程变量**（error procedure variable）。下面的例子说明了这个方法。

```
  svc data rrerr;
    label erl;
    proc (int) erp; % erp 是一个过程变量 %
  enddata;

  proc recover (int);
    ...
    ...
  endproc;

  proc WhereErrorIsDetected ();
    ...
    if recoverable then
      erp (n)
    else
      goto erl
    end;
    ...
  endproc;

  proc Caller ();
    ...
    WhereErrorIsDetected ();
    ...
  endproc;

  proc main ();
    ...
```

138

```
    erl := fail;
    erp := recover;
      ...
    Caller ();
      ...
  fail:
    ...
  end proc;
```

对这种方法的主要批评也是程序变得非常难以理解和维护。

6.2 现代异常处理

异常处理的传统方法常常导致处理代码同程序的正常执行流混杂在一起。现代方法是直接在语言中引进异常处理设施，因而提供更加结构化的异常处理机制。这些设施的确切特性因语言而异，然而能够识别出若干共同点。下面几小节讨论它们。

6.2.1 异常及其表示

在5.5.1节提到有两类错误检测技术：环境检测和应用检测。还有，依赖于检测错误的延时，可能有必要同步地引发异常或异步地引发异常。同步异常是由代码段试图执行不合适操作引起的立即后果；异步异常是在导致错误发生的操作之后的某个时间引发的。异常可以在原来执行这个操作的进程中引发，也可以是在其他进程中引发。因此有四类异常：

1）由环境检测并同步引发的异常——数组界破坏和零做除数是这种异常的例子。

139 2）由应用检测并同步引发的异常——例如，程序定义的断言检查失败。

3）由环境检测并异步引发的异常——由供电失败或某些健康监控机制失败引发的异常。

4）由应用检测并异步引发的异常——例如，一个进程识别到出现了一个出错状态，它将导致另一进程不满足其时限或不能正确终止。

异步异常通常称为异步通知或信号，并经常是在并发编程的上下文中研究它。所以，本章将关注同步异常处理，而把异步异常处理的问题留到第10章。

关于同步异常，它们的声明有几个模型。例如，它们可以被看作：

- 常量名字，需要显式声明
- 特定类型的对象，它们可能需要也可能不需要显式声明

Ada要求异常像常量一样声明，例如，可由Ada运行时环境引发的异常是在包Standard中声明的：

```
package Standard is
  ...
  Constraint_Error : exception;
  Program_Error   : exception;
  Storage_Error   : exception;
  Tasking_Error   : exception;
  ...
end Standard;
```

这个包对所有Ada程序都是可见的。

Java和C++对异常采用一种更加面向对象的观点。在Java中，异常是Throwable类的子

类的对象。它们可由运行时系统和应用**抛出**（Thrown）（见6.3.2节）。在C++中，任何对象类型的异常都可被抛出而无须事先声明。

6.2.2 异常处理程序的定义域

在程序中，对一个具体异常可能有若干处理程序。同每个处理程序相关联的是定义域，它指明一个计算区域，在这个计算区域出现异常时，就激活这个处理程序。规定定义域的准确度将决定异常源可被定位的精确性。在像Ada这样的块结构语言中，这个定义域通常就是块。例如，考虑一个温度传感器，它的值应当在0 ~ 100°C之间。下面的Ada块定义温度是0 ~ 100之间的一个整数。如果计算出的值落在此范围之外，Ada的运行时支持程序引发Constraint_Error异常。对相关处理程序的调用使得能执行必要的改正动作。 140

```
declare
  subtype Temperature is Integer range 0 .. 100;
begin
  -- 读温度传感器并计算它的值
exception
  -- Constraint_Error的处理程序
end;
```

Ada细节将会在稍后给出。

在以块作为基础的其他单元（例如过程和函数）里，异常处理程序的定义域通常就是那种单元。

在其他语言中，如Java、C++和Modula-3中，不是所有的块都能有异常处理程序。异常处理程序的定义域是必须显式指出的，并且这种块被认为是要**守备的**（guarded），在Java中是用“try块”做这件事：

```
try {
  // 可能引发异常的语句
}
catch (ExceptionType e) {
  // e的处理程序
}
```

因为异常处理程序的定义域规定了错误能被精确定位到什么程度，可能会有争论说块的粒度不够。例如，考虑如下的计算序列，其中的每一个都可能引发Constraint_Error。

```
declare
  subtype Temperature is Integer range 0.. 100;
  subtype Pressure is Integer range 0 .. 50;
  subtype Flow is Integer range 0 .. 200;
begin
  -- 读温度传感器并计算它的值
  -- 读压力传感器并计算它的值
  -- 读水流传感器并计算它的值

  -- 按照需求调整温度、压力和水流
exception
  -- Constraint_Error的处理程序
end;
```

该处理程序的问题是判断哪个计算引起了异常的引发。在可能出现运算上溢和下溢的时候又引发进一步的困难。

在基于块的异常处理程序定义域中，这个问题的一个解决方案是减少块的大小和/或嵌套它们。用传感器的例子:

```
declare
  subtype Temperature is Integer range 0 .. 100;
  subtype Pressure is Integer range 0 .. 50;
  subtype Flow is Integer range 0 .. 200;
begin
  begin
    -- 读温度传感器并计算它的值
  exception
    -- 温度的Constraint_Error的处理程序
  end;
  begin
    -- 读压力传感器并计算它的值
  exception
    -- 压力的Constraint_Error的处理程序
  end;
  begin
    -- 读水流传感器并计算它的值
  exception
    -- 水流的Constraint_Error的处理程序
  end;
    -- 按照需求调整温度、压力和水流
exception
  -- 其他可能异常的处理程序
end;
```

另一种办法是可为每一个嵌套的块创建包含处理程序的过程。然而，不管哪种情况，这都可能变得冗长乏味。另一个解决方案是允许异常在语句级处理。用这种方法，上述例子可重写成这样:

141

```
-- 不合法的 Ada程序
declare
  subtype Temperature is Integer range 0 . . 100;
  subtype Pressure is Integer range 0 .. 50;
  subtype Flow is Integer range 0 .. 200;
begin
  Read_Temperature_Sensor;
    exception -- Constraint_Error的处理程序;
  Read_Pressure_Sensor;
    exception -- Constraint_Error的处理程序;
  Read_Flow_Sensor;
    exception -- Constraint_Error的处理程序;
  -- 按照需求调整温度、压力和水流
end;
```

CHILL编程语言（CCITT，1980）有这种设施。虽然这能使引发异常的原因被更精确地定位，但是，它却将异常处理代码同正常操作流混杂在一起，它可能产生不够清晰的程序并破坏了需求R2（在本章开始处给出的）。 142

对此问题的建议方法是使参数能同异常一起传递。这在Java里是自动的，因为异常是对象，所以可以包含程序员希望的诸多信息。相比之下，Ada提供一个预定义过程`Exception_Information`，它返回有关异常这次出现的、由实现定义的细节。

6.2.3 异常传播

同异常定义域概念密切相关的是异常传播的概念。我们的讨论一直都隐含着一个假设：如果块或过程引发了一个异常，那么在那个块或过程中有一个相关联的处理程序。然而，情况可能不是这样，并且有两种可能的方法处理不能找到直接的异常处理程序的情况。

第一种方法是将处理程序的缺失看成程序员错误，应在编译时报告。然而，常常有这种情况：在一个过程中引发的异常只能在调用此过程的上下文中处理。在这种情况下，不可能使处理程序位于这个过程中。例如，由于过程的参数问题使一个断言失败所引发的异常，只能在进行调用的上下文中处理。令人遗憾的是，编译器不总是能够检查发出调用的上下文是否包含有合适的异常处理程序，因为这可能需要复杂的流控制分析。这在一个过程调用了其他也可能引发异常的过程时尤其困难。所以，那种要求为这种情况生成编译报错的语言需要过程指明它可能引发哪些异常（即，未局部处理的异常）。这样，编译器就能检查发出调用的上下文是否有合适的处理程序并在必要时生成所需的出错消息。这是CHILL语言采用的方法。Java和C++使函数能够定义它可能引发的异常。然而，与CHILL不同，它们不要求在发出调用的上下文中有可用的处理程序。

当不能找到一个异常的局部处理程序时可采用的第二种方法，是在运行时搜寻调用者链以寻找处理程序，这称为异常**传播**。Ada和Java允许异常传播，C++和Modula 2/3也这样做。

当语言要求声明异常并因此给定作用域的时候，关于异常传播就出现了一个潜在的问题。在某些情况下，一个异常可能被传播出它的作用域，因而不可能找到其处理程序。为对付这种情况，大多数语言提供了一种“捕获所有的”（catch all）异常处理程序。这种处理程序还被用于避免程序员枚举许多异常名字。

一个未处理的异常引起一个顺序程序的中止。如果程序包含一个以上的进程，而一个特定进程没有处理它引发的异常，那么通常这个进程被中止。然而，并不清楚该异常是否应当传播到父进程。多进程程序中的异常问题将在第10章详细研究。

研究异常传播问题的另一种方式是按处理程序是静态地还是动态地同此异常相关联。静态关联，如在CHILL中那样，是在编译时进行的，所以不能允许传播，因为调用者链是未知的。动态关联是在运行时进行的，所以能够允许传播。虽然动态关联更灵活，但由于要搜索处理程序会导致更多运行开销；用静态关联，可以生成一个编译时的地址。 143

6.2.4 恢复模型与终止模型的对比

在任何异常处理设施中的一个关键考虑因素是异常的调用者在该异常被处理之后是否应当继续执行。如果调用者能够继续，那么处理程序就有可能解决引起异常引发的问题，且调用者有可能恢复，就像什么也没有发生一样。这被称为**恢复**（resumption）或**通知**（notify）模型。控制不返回给调用者的模型称为**终止**（termination）或**逃逸**（escape）。显然可能有这

样一个模型，处理程序能够决定是恢复引起异常的操作还是终止这个操作。这称为混合模型。

1. 恢复模型

为说明恢复模型，考虑三个过程P、Q和R。过程P调用Q，Q又调用R。过程R引发一个异常r，它由Q处理（假设在R中没有局部处理程序），r的处理程序是Hr。在处理r期间，Hr引发异常q，它由过程P（Q的调用者）中的Hq处理。一旦q被处理完毕，Hr继续执行，Hr完成后R又继续执行。图6-1用编号为1至6的弧图形化地表示了这个事件序列。

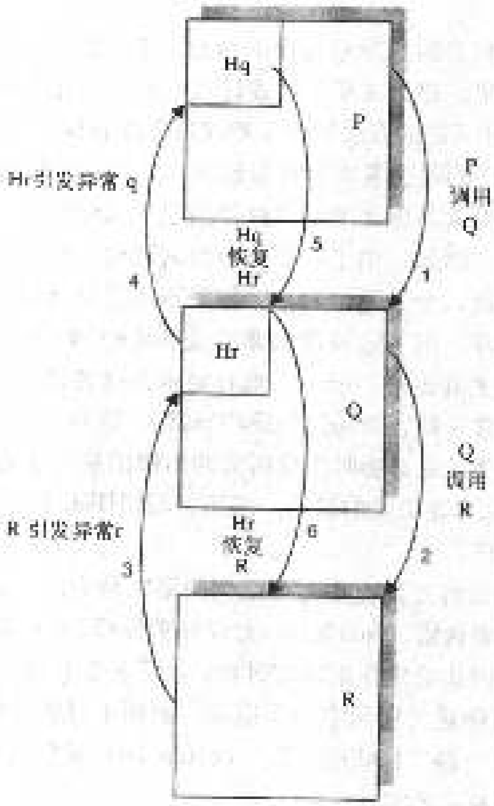


图6-1 恢复模型

通过将处理程序看成一个隐式过程（在异常被引发时就调用它），恢复模型是最容易理解的。

这个方法的问题是经常难以修复由运行环境引发的错误。例如，在复杂表达式序列的中间出现的运算溢出可能使若干寄存器含有部分求值结果。由于调用了处理程序，这些寄存器可能被写覆盖。

Pearl和Mesa语言都提供这样一种机制：处理程序能够返回到引发异常的上下文。两个语言也支持终止模型。

虽然实现严格的恢复模型是困难的，但折衷方案是再次执行同异常处理程序相关联的块。Eiffel语言（Meyer，1992）提供这样的设施，称为重试（retry），作为异常处理模型的一部分。处理程序能够设置一个局部标志，用以指出错误已经出现，并且这个块能测试这个标志。注意，为使这种方案工作，这个块的局部变量在重试时不得被再次初始化。

恢复模型的优点是，当异常已经被异步地引发时，当前进程执行几乎不做什么事。异步事件处理在10.5节详细讨论。

2. 终止模型

在终止模型中，在引发异常并且调用了处理程序的时候，控制不返回到发生异常的那一点。包含处理程序的块或过程被终止，控制转移到发出调用的块或过程上。所以，被调用的过程可以以几种状态终止。其中之一是**正常状态**，其他的是**异常状态**。 144~145

当处理程序是在一个块结构里面的时候，在异常被处理之后，控制转移到跟在那个块之后的第一个语句，如下例所示。

```
declare
  subtype Temperature is Integer range 0..100;
begin
  ...
  begin
    -- 读温度传感器并计算它的值，
    -- 可能导致引发一个异常
  exception
    -- 温度的Constraint_Error的处理程序
    -- 一旦处理完就终止这个块
  end;
  -- 当这个块正常退出时或当引发一个异常并已经处理时
  -- 执行这里的代码
exception
  -- 其他可能的异常的处理程序
end;
```

对过程来说，同块相反，控制流可以十分戏剧性地改变，如图6-2所示。过程P同样调用Q，Q再调用R。在R中引发的异常在Q中得到处理。

Ada、Java、C++、Modula-2/3和CHILL都具有异常处理的终止模型。

3. 混合模型

对混合模型来说，由处理程序决定错误是不是可恢复的。如果是，处理程序可返回一个值，并且语义与恢复模型相同。如果错误是不可恢复的，调用者被终止。Mesa和实时Basic（Bull and Lewis，1983）的信号机制提供这种设施。如前面说过的，Eiffel也支持受限的“重试”模型。

4. 异常处理和操作系统

在许多情况下，像Ada或Java这样的语言的程序是在诸如POSIX或NT之类的操作系统上执行的。这些系统将检测某些同步出错状态，例如，存储器破坏或非法指令。一般这会导致正在执行的进程被终止。然而，许多系统允许程序员去尝试出错恢复。例如，POSIX支持的恢复模型允许程序员通过将处理程序同异常关联起来去处理这些同步异常（经由POSIX中的信号）。当系统检测到这些出错状态时，就调用这种处理程序。一旦处理程序完成，该进程就在它被“中断”的地方恢复——所以POSIX支持恢复模型。

如果一个语言支持终止模型，那么，捕获出错并对程序状态进行必要处理就是那个语言运行时支持系统的责任，以使程序员可以使用终止模型。 146

在第10章中将详细研究POSIX信号，因为它们确实是异步并发机制。

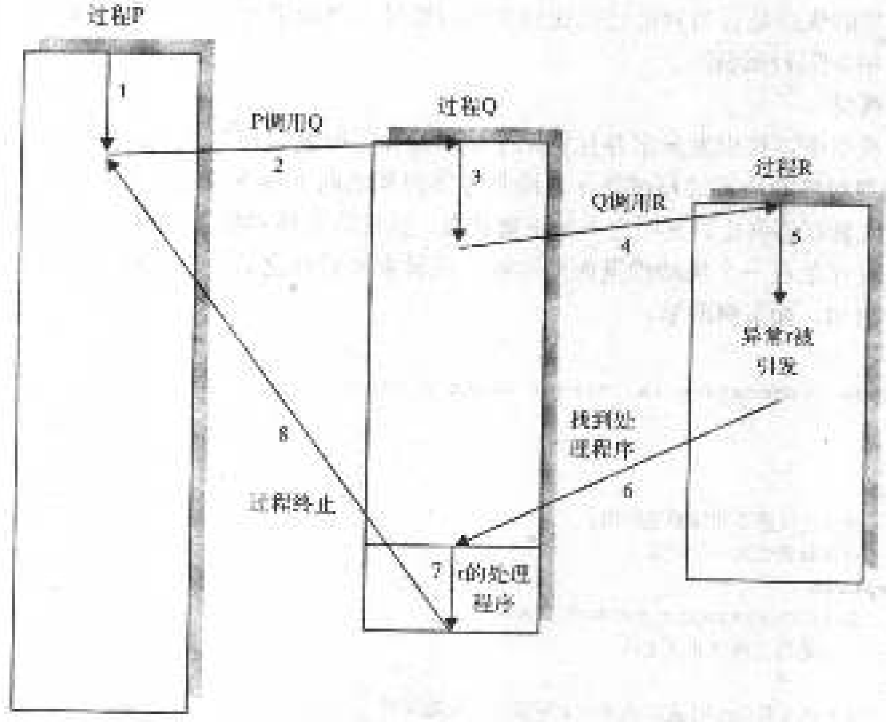


图6-2 终止模型

6.3 Ada、Java和C中的异常处理

现在研究顺序Ada、Java和C中的异常处理，三种语言都有不同的指导思想。并发系统中
147 的异常处理将在第10章中描述。

6.3.1 Ada

Ada语言支持显式异常声明、带有未处理异常传播的异常处理终止模型以及有限形式的异常参数。

1. 异常声明

Ada异常以两种方式声明。第一种方式与常量相同，常量的类型由关键字exception定义。下面的例子声明了一个名为Stuck-Valve的异常：

```
Stuck_Valve: exception;
```

另一种方式是使用预定义包Ada.Exceptions（见程序6-1），它定义一个私有类型叫做Exception_Id。使用关键字exception声明的每个异常有一个关联的Exception_Id，使用预定义属性Identify可得到它。上面给定的Stuck_Valve异常的身份可以这样找到：

```
with Ada.Exceptions;
with Valves;
package My_Exceptions is
  Id : Ada.Exceptions.Exception_Id :=
            Valves.Stuck_Valve' Identity;
end My_Exceptions;
```

这里假设Stuck_Valve是在包Valves中声明的。注意，现在将函数Exception_Name应用到Id，将返回字符串My_Exception.Id，而不是Stuck_Valve。

异常可以在任何其他声明出现的地方声明，并且像其他声明一样，有作用域。

语言有若干标准异常，它们的作用域是整个程序。这些异常可由语言的运行时支持系统引发，以响应某种出错状态。它们包括：

• Constraint_Error（约束错）

  例如，它在以下情况被引发：向对象所赋的值在其声明的范围之外；对数组的访问超出数组界；试图用空指针进行访问。执行预定义的数值运算不能交付正确结果（在实数类型的声明准确度之内），也引发这个异常，这包括以零做除数。

• Storage_Error（存储错）

当动态存储分配器不能满足存储要求时引发此异常，因为机器的物理限制存储已被耗尽。 148

程序6-1 包Ada.Exceptions

```
package Ada.Exceptions is
  type Exception_Id is private;
    -- 每个异常有一个相关联的标识符
  Null_Id : constant Exception_Id;
  function Exception_Name (Id : Exception_Id) return String;
    -- 返回那个以Exception_Id为Id的对象的名字

  type Exception_Occurrence is limited private;
    -- 每个异常出现有一个相关联的标识符
  type Exception_Occurrence_Access is
       access all Exception_Occurrence;
  Null_Occurrence : constant Exception_Occurrence;

  procedure Raise_Exception (E : in Exception_Id;
            Message : in String:= "");
    -- 引发异常 E 并将Message同该异常出现相关联

  function Exception_Message (X : Exception_Occurrence)
          return String;
    -- 允许由Raise_Exception传递的字符串可在此处理程序内被访问;
    -- 如果这个异常是由引发语句引发的,
    -- 该字符串包含有关此异常的实现定义的信息

  procedure Reraise_Occurrence (X : in Exception_Occurrence);
    -- 再次引发由该异常出现参数标识的异常

  function Exception_Identity (X : Exception_Occurrence)
          return Exception_Id;
    -- 返回作为参数传递的异常出现的异常标识符

  function Exception_Name (X : Exception_Occurrence)
          return String;
    -- 同 Exception_Name (Exception Identity (X) ) .

  function Exception_Information (X : Exception_Occurrence)
          return String;
    -- 同 Exception_Message (X) , 但包括更多细节,
```

```
    -- 如果该消息来自实现的话

  procedure Save_Occurrence (Target : out Exception_Occurrence;
                             Source : in Exception_Occurrence);
    -- 允许对类型为Exception_Occurrence的对象赋值

  function Save_Occurrence (Source : Exception_Occurrence)
                            return Exception_Occurrence_Access;
    -- 允许对类型为Exception_Occurrence的对象赋值
```

149

```
private
  ...   -- 不由语言规定
end Ada.Exceptions;
```

2. 引发异常

除由程序执行的环境引发的异常之外，还可由程序使用**raise**语句显式引发异常。下面的例子引发异常Io_Error（它必须已先行声明，并在作用域内），如果一个I/O请求产生了设备错误的话。

```
begin
  ...
  -- 请求一个设备执行某些I/O的语句
  if Io_Device_In_Error then
    raise Io_Error;
  end if;
  ...
end;
```

注意该if语句不需要else部分，因为控制不返回到跟在引发语句后面的语句。

如果Io_Error已被声明为一个Exception_Id，就有必要使用过程Ada.Exceptions.Raise_Exception引发该异常。这允许将一个文本字符串作为参数传递给这个异常。

异常的每个引发称为异常**出现**（occurrence），并被表示为Ada.Exceptions.Exception_Occurrence类型的值。当异常被处理时，就可找到Exception_Occurrence的值，并用于确定有关异常原因的更多信息。

3. 异常处理

如第3章所示，Ada中的每个块（每个子程序、接受语句或任务）可以包括一个可选的异常处理程序的集合，在块或子程序、接受语句或任务的末尾声明它们。每个异常处理程序是一个语句序列。引导这个序列的是：关键字**when**，一个可选的参数（将对它赋以异常出现的身份），处理程序为之服务的异常的名字和符号=>。例如，下面的块声明了三个异常并提供两个处理程序：

```
declare
  Sensor_High, Sensor_Low, Sensor_Dead : exception;
```

150

```
  -- 其他声明
begin
  -- 可能引起上述异常被引发的语句
exception
  when E: Sensor_High | Sensor_Low =>
  -- 如果引发了 Sensor_High 或 Sensor_Low，采取某些矫正动作
  -- E 包括异常出现
```

```
    when Sensor_Dead =>
    -- 如果引发了异常Sensor_Dead，响起警铃
  end;
```

为避免枚举所有可能的异常名字，Ada提供了一个**when others**处理程序名字。这允许作为最后的异常处理选择并代表在当前处理程序集合中前面没有列出的所有异常。例如，下面的块打印出关于异常的信息，并在引发除Sensor_Low或Sensor_High之外的任何异常（包括Sensor_Dead）时响一次警铃。

```
declare
  Sensor_High, Sensor_Low, Sensor_Dead : exception;
  -- 其他声明
  use Text_Io;
begin
  -- 可能引起上述异常被引发的语句
exception
  when Sensor_High | Sensor_Low =>
  -- 采取某些矫正动作
  when E: others =>
    Put (Exception_Name (E) );
    Put_Line ("caught. The following information is available ");
    Put_Line (Exception_Information (E) );
    -- 响起警铃
end;
```

在一个异常处理程序里面引发的异常不能够由那个处理程序或同一块（或过程）中的其他处理程序处理。相反，这个块被终止，并在外包的块中或在子程序的调用点上寻求处理程序。

4. 异常传播

如果在一个异常的内包块（或子程序或接受语句）中没有异常处理程序，该异常被再次引发。Ada就是这样**传播**异常。对于块的情况，这导致异常在内包块或子程序中被引发。对于子程序的情况，异常在其调用点被引发。

关于Ada的一个常见错误观念是异常处理程序可在包的初始化段里提供，以处理在它们嵌套的子程序执行里引发的异常。由子程序引发而未被子程序处理的异常被传播给子程序的调 151
用者。所以，这样的异常将只在初始化代码自身调用了子程序时才被初始化代码处理。下面的例子说明了这一点。

```
package Temperature_Control is
  subtype Temperature is Integer range 0 .. 100;
  Sensor_Dead, Actuator_Dead : exception;

  procedure Set_Temperature (New_Temp : in Temperature);
  function Read_Temperature return Temperature;
and Temperature_Control;

package body Temperature_Control is
  procedure Set_Temperature (New_Temp : in Temperature) is
  begin
  -- 将新温度通知致动器
    if No_Response then
```

```
      raise Actuator_Dead;
    end if;
  end Set_Temperature;

function Read_Temperature return Temperature is
begin
  -- 读传感器
  if No_Response then
    raise Sensor_Dead;
  end if;
  -- 计算温度
  return Reading;
exception
  when Constraint_Error =>
    -- 温度超出其预期范围，采取一些适当的动作
  end Read_Temperature;
begin
  -- 包初始化
  Set_Temperature (Initial_Reading);
exception
  when Actuator_Dead =>
    -- 采取一些矫正动作
end Temperature_Control;
```

在此例中，过程Set_Temperature可被包外调用，也在包初始化时被调用。此过程可能引发异常Actuator_Dead。在包的初始化段为Actuator_Dead给出的处理程序将只能*在初始化代码调用过程时捕获异常*。它不能捕获包外调用过程的异常。

152 如果包体的初始化代码本身引发了一个异常且没有局部处理，则该异常被传播到使这个包进入作用域的那一点。

5. 最后的希望

一个异常还可以通过程序在局部处理程序中再次引发该异常而传播。语句raise（或过程Ada.Exceptions.Reraise_Occurrence）具有再次引发上一个异常（或特定异常出现）的效果。这种设施在**“最后的希望”**（last wishes）编程中是很有用的。常常有这种情况：一个异常的意义对于局部处理程序来说是未知的，但必须被处理以清理在此异常被引发前可能已经发生的任何资源分配。例如，考虑一个分配若干设备的过程。在分配例程中引发的任何异常被传播给调用者，这些异常可能留下几个分配了的设备。所以，如果已经不可能分配整个请求的话，分配器希望回收有关的设备。下面的例子说明了这种方法。

```
subtype Devices is Integer range 1 .. Max;

procedure Allocate (Number : Devices) is
begin
  -- 请求依次分配设备
  -- 注意哪些请求被获准的
exception
  when others =>
    -- 回收已分配的设备
    raise; -- 再次引发该异常
end Allocate;
```

按这种方式使用，该过程可被认为是实现原子动作的失效原子性质，要么所有资源都分配了，要么一个都不分配（见第10章）。

为了进一步说明，考虑一个过程，它设置电操纵飞机在着陆阶段机翼上的前沿缝翼和襟翼的位置。这些位置改变飞机升力的大小，在着陆（或起飞）时不对称的机翼设置会使飞机变得不稳定。假设初始设置是对称的，下面的过程确保它们保持对称，即使引发了异常——或是由于物理系统的失败或是由于程序错误[⊖]。

```
procedure Wing_Settings ( -- 有关的参数) is
begin
  -- 进行所需的前沿缝翼和襟翼的设置；
  -- 可能引发异常
exception
  when others =>
  -- 确保设置是对称的
  -- 再次引发异常，以指出进行一次无前沿缝翼和襟翼的着陆
    raise;
end Wing_Settings;
```

153

Ada还为“最后的希望”编程提供另一种机制。本质上说，就是在过程中声明哑受控变量。所有受控变量在离开作用域的时候，都有被自动调用的过程。这种终结过程能够保证在异常出现时的终止状态。在上面的例子中，它会确保前沿缝翼和襟翼具有对称设置，见4.4.1节。

6. 在声明制作期间引发的异常

在子程序、块、任务或包的声明部分有可能引发异常（例如，为变量赋一个规定范围之外的值）。一般说来，出现这种情况时，就放弃这个声明部分，并首先在详尽描述块、子程序、任务或包的地方引发异常。

异常处理规则在所有这些情况下的完整定义要比这里列出的复杂一些。读者可参考《Ada 95 参考手册》（Ada 95 Reference Manual）第11章，以查阅全部细节。

7. 异常的屏蔽

在过去十年里，有一句格言在程序员中逐渐流行，它通常是这样说的：“没有免费的午餐！”异常处理设施的需求之一是：除非异常被引发了，它们不应当引起运行时开销（R3）。Ada提供的设施已经描述过了，表面上看它们似乎满足这个需求。然而，经常会有同检测可能的出错状态相关的某些开销。

例如，Ada提供了一个名为`Constraint_Error`的标准异常，它在使用了空指针、有数组界出错或对象被赋以超出其容许范围的值的时候被引发。为了捕获这些出错状态，编译器必须生成适当的代码。例如，当对象是经过指针访问的时候，如果没有任何全局流控制分析（或硬件支持），编译器就要在访问该对象之前插入检验指针是否为空的代码。虽然这个代码对程序员来说是不可见的，但即使在没有引发异常时，它也会执行。如果程序使用许多指针，这可能导致在执行时间和代码长度两方面的重大开销。进而，这种代码的出现可能在任何确认过程中都需要被检验，这可能是难于做到的。

Ada语言确实认识到由运行时环境引发的标准异常对具体应用而言可能是很昂贵的。所以，它提供了一个能够屏蔽这些检查的设施。这是通过使用`Suppress`编用删除所有的运行时检查 154

⊖ 这当然是一个用于说明方法的粗糙例子，实际使用的不一定是这种方法。

实现的。这个编用只影响它所在的那个编译单元。当然，如果一个运行时错误检查被屏蔽掉了，而随后这种错误发生了，那么，语言认为程序是“有错的”，程序的后续行为是不确定的。

8. 完整例子

下面的包说明异常在实现一个Stack的抽象数据类型中的使用。选这个例子是因为它能给出完整规格说明和体，而不再留什么东西给读者去想像。

包是类属的，因而可实例化为不同类型。

```
generic
  Size : Natural := 100;
  type Item is private;
package Stack is

  Stack_Full, Stack_Empty : exception;

  procedure Push (X:in Item);
  procedure Pop (X:out Item);

end Stack;


package body Stack is

  type Stack_Index is new Integer range 0..Size-1;
  type Stack_Array is array (Stack_Index) of Item;
  type Stack is
    record
      S : Stack_Array;
      Sp : Stack_Index := 0;
    end record;
  Stk : Stack;

  procedure Push (X:in Item) is
  begin
    if Stk.Sp = Stack_Index' Last then
      raise Stack_Full;
    end if;
    Stk.Sp :=Stk.Sp + 1;
    Stk.S (Stk.Sp) := X;
  end Push;

  procedure Pop (X:out Item) is
  begin
    if Stk.Sp = Stack_Index'First then
      raise Stack_Empty;
    end if;
      X := Stk.S (Stk.Sp);
      Stk.Sp := Stk.Sp - 1;
    end Pop;

  end Stack;
```

155

可以这样使用它:

```
with Stack;
```

```
with Text_Io;
procedure Use_Stack is
  package Integer_Stack is new Stack (Item => Integer);
  X : Integer;
  use Integer_Stack;
begin
  ...
  Push (X);
  ...
  Pop (X);
  ...
exception
  when Stack_Full =>
    Text_Io.Put_Line ("stack overflow!");
  when Stack_Empty =>
    Text_Io.Put_Line ("stack empty!");
end Use_Stack;
```

9. Ada异常模型的困难

虽然Ada语言为异常处理提供了一套全面的设施，但在它的易用性方面还有一些困难。

1）**异常和包**。可由包的使用引发的异常，在包的规格说明中可以与能被调用的子程序一道声明。但是，哪些子程序可以引发哪些异常是不明显的。如果包的用户没有注意它的实现，他们必然试图将异常的名字同子程序名字关联起来。在上面所给的关于栈的例子中，用户可能会假设异常`Stack_Full`是由过程Pop而不是由Push引发的！对于大型包，哪些异常可由哪些子程序引发是不明显的。在这种情况下，程序员必须在每次调用一个子程序时枚举所有可能的异常或是使用**when others**。所以，包的作者应当使用注解指出哪些子程序可以引发哪些异常。

2）**参数传递**。Ada不允许将除字符串之外的参数传递给异常。如果需要传递特定类型的对象，那么这是不方便的。

3）**作用域和传播**。异常可能被传播到它们声明的作用域之外。这种异常只能由**when others**捕获。然而，当进一步传播到动态链时，它们可能再次回到作用域。当使用块结构语言和异常传播时，这是使人为难的，虽然可能是不可避免的。 156

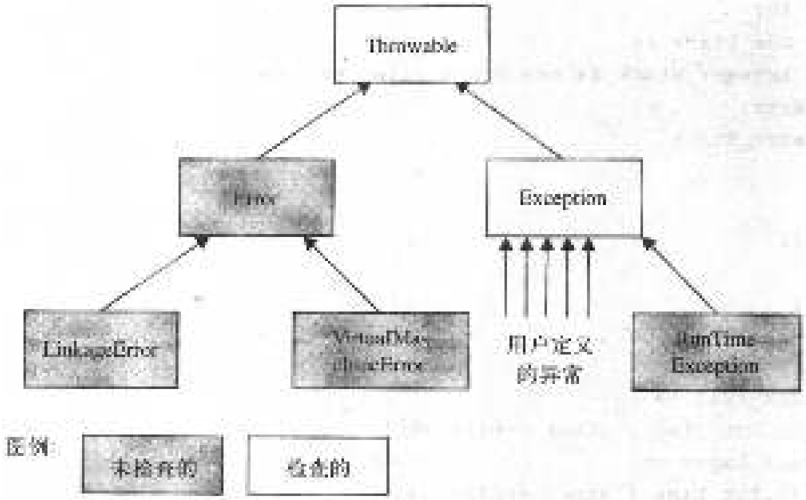
6.3.2 Java

在支持异常处理的终止模型方面，Java类似于Ada。不过，Java异常与Ada不同的是它们被集成到了面向对象模型里。

1. 异常声明

在Java中，所有异常都是预定义类`java.lang.Throwable`的子类。该语言也定义其他一些类，例如：`Error`、`Exception`和`RuntimeException`。图6-3画出了它们之间的关系。在本书中，Java异常这个术语用于表示从`Throwable`派生的任何类，而不只是从`Exception`派生的类。

从`Error`派生的对象描述Java运行时支持系统中的内部错误和资源耗尽。虽然这些错误显然对程序有重大影响，但是当这些错误被抛出（引发）时，程序不能做什么事，因为关于系统的完整性不能做任何假设。



157 图6-3 Java的预定义类Throwable的层次体系

从Exception层次体系派生的对象代表程序自己能处理的错误，并可能抛出它们自己。RuntimeException是那种作为程序出错的结果，由运行时支持系统引发的异常。它们包括诸如由不良类型转换（ClassCastException）、数组界错（IndexOutOfBoundException）、空指针访问（NullPointerException）、零做除数（ArithmeticException）等产生的错误。

从Error或RuntimeExceptions派生的可抛出对象被称为未检查异常。这意味着Java编译器不预期它们在方法声明的throws子句中被标识出来（见下面）。

例如，考虑在6.3.1节中给出的温度控制器的例子，可用Java写成下面的样子。首先，必须声明一个类以表示整数子类型的一个约束错误。因为这是一个一般出错状态，可能已经在库中提供了它。这个类的公有变量将包括关于出错原因的信息。

```
public class IntegerConstraintError extends Exception
{
  private int lowerRange, upperRange, value;

  public IntegerConstraintError (int L, int U, int V)
  {
    super (); // 调用父类构造器
    lowerRange = L;
    upperRange = U;
    value = V;
  }
  public String getMessage ()
  {
    return ("Integer Constraint Error: Lower Range"+
           java.lang.Integer.toString (lowerRange) + "Upper Range" +
           java.lang.Integer.toString (upperRange) + "Found" +
           java.lang.Integer.toString (value) );
  }
};
```

现在可以引入temperature类型:

```
import exceptionLibrary.IntegerConstraintError;

public class Temperature
{
  private int T;

  public Temperature (int initial) throws IntegerConstraintError
    // 构造器
  {
    ...;
  }

  public void setValue (int V) throws IntegerConstraintError
  {
    ...;
  };

  public int readValue ()
  {
    return T;
  };

  // 构造器和setValue 都可抛出IntegerConstraintError
};
```

158

在上述代码中，可抛出IntegerConstraintError异常的成员函数是正式标出的。把这一点和Ada方法对比，Ada是依赖于注解的。

现在可以定义类TemperatureController。它声明了一个类型，用以表示失败的致动器异常。在这种情况中，对象中无数据传递。

```
class ActuatorDead extends Exception
{
  public String getMessage ()
  {
    return ("Actuator Dead");
  }
};

class TemperatureController
{
  public TemperatureController (int T)
         throws IntegerConstraintError
  {
    currentTemperature = new Temperature (T);
  };

  Temperature currentTemperature;

  public void setTemperature (int T)
         throws ActuatorDead, IntegerConstraintError
  {
    currentTemperature.setValue (T);
  };

  int readTemperature ()
```

```
      {
        return currentTemperature.readValue ();
      }
159 };
```

通常，每个函数必须指明一个可抛出的已检查异常的列表**throws** A，B，C——在此情况下该函数可抛出此列表中的任何异常和任何未检查异常。A、B和C必须是Exception的子类。如果函数试图抛出一个其抛出列表不允许的异常，那么发生编译错误。

2. 抛出异常

在Java中，引发异常称为**抛出**异常。作为例子，考虑上节概略描述的类Temperature的完全实现。

```
import exceptionLibrary.IntegerConstraintError;

class Temperature
{
  int T;

  void check (int value) throws IntegerConstraintError
  {
    if (value > 100 || value  < 0) {
      throw new IntegerConstraintError (0, 100, value);
    };
  }

  public Temperature (int initial) throws IntegerConstraintError
    // 构造器
  {
    check (initial);
    T = initial;
  }

  public void setValue (int V) throws IntegerConstraintError
  {
    check (V);
    T = V;
  };

  public int readValue ()
  {
    return T;
  };
};
```

这里，**throw new** IntegerConstraintError (0, 100, value) 创建并抛出一个类
160 型 (IntegerConstraintError) 的对象，并给其实例变量赋予适当的值。

3. 异常处理

在Java中，异常只能在**try**块里处理。每个处理程序是用**catch**语句规定的。考虑下面的代码:

```
// 给定TemperatureController
try{
  TemperatureController TC = new TemperatureController (20);
```

```
  TC.setTemperature (100);
  // 操纵温度的语句
}
catch (IntegerConstraintError error) {
  // 捕获到异常，在标准输出上打印出错信息
  System.out.println (error.getMessage () );
}
catch (ActuatorDead error) {
  System.out.println (error.getMessage () );
}
```

catch语句像是一个函数声明，它的参数标识出被捕获的异常类型。在这个处理程序里面，对象名就像局部变量。

具有参数类型T的处理程序将捕获类型E的抛出对象，前提是

1）T和E是同一类型，或

2）T在抛出点是E的父（超）类。

正是这最后一条使Java的异常处理设施非常强大。在上述例子中，两个异常是从类Exception派生的：IntegerConstraintError和ActuatorDead。下面的try块将捕获这两个异常：

```
try {
  // 可能引发异常IntegerConstraintError 或ActuatorDead的语句
}
catch (Exception E) {
  // 处理程序将捕获所有异常类型及派生类型的异常;
  // 但只能是在Exception的方法是可访问的那些处理程序里面
}
```

当然，对于E.getMessage的调用将为抛出对象的类型分派适当例程。确实，catch (Exception E) 等价于Ada的**when others**。 161

4. 异常传播

像Ada一样，如果在函数调用的上下文中不能找到异常处理程序，则该调用上下文终止，且在它的调用上下文中寻找处理程序。所以，Java支持异常传播。

5. 最后的希望

6.3.1节讨论了**when others**怎么在Ada中被用来编写最后的希望。显然，catch (Exception E) 可用于实现这个效果。然而，Java还支持作为try语句一部分的**finally**子句。附加于这个子句的代码被保证执行，不管在try语句中发生了什么，不管异常是否被抛出、捕获或传播，或者甚至根本没有抛出任何异常。

```
try
{
  ...
}
catch (..)
{
...
}
finally
{
  //在所有情况下都要执行的代码
}
```

Ada使用受控变量和终了化（finalization）可实现同样效果。

6. 完整例子

下面是Java的一个关于栈的例子，前面曾用Ada给出过。关于这个例子有两点有趣的东西。

首先，栈是类属的：任何对象都可作为入栈的参数传递。所以，不需要将栈实例化（如在Ada里那样）。此外，每个栈可处理一种以上的对象类型（与Ada不同）。当然，这是因为Java的栈实际上是对象引用的栈。也可以用Ada显式地编制这个程序。

162 第二点是Java例子只处理对象，而不处理像整数这样的基本类型。这同Ada的例子形成对比，Ada处理所有类型。

```
public class EmptyStackException extends Exception
{
    public EmptyStackException () {
    }
}

public class FullStackException extends Exception
{
    public FullStackException () {
    }
}
public class Stack {

  public Stack (int capacity)
  {
    stackArray = new Object[capacity];
    stackIndex = 0;
    stackCapacity = capacity;
  }

  public void push (Object item) throws FullStackException
  {
    if (stackIndex == stackCapacity) throw new FullStackException ();
    stackArray[stackIndex++] = item;
  }

  public synchronized Object pop () throws EmptyStackException
  {
    if (stackIndex == 0) throw new EmptyStackException ();
    return stackArray[--stackIndex];
  }

  protected Object stackArray[];
  protected int stackIndex;
  protected int stackCapacity;
}
```

上面的类可以像下面这样使用：

```
import Stack;
import FullStackException;
import EmptyStackException;
```

```
public class UseStack
{
  public static void main (...)
  {
    Stack S = new Stack (100);
    try {
      ...
      S.push (SomeObject);
      ...
      SomeObject = S.pop ();
      ...
    }
    catch (FullStackException F) ...
    catch (EmptyStackException E) ...;
  }
}
```

163

最后，应当注意Java在它的util包中提供一个标准Stack类。

6.3.3 C

C没有在语言里定义任何异常处理设施。这种缺失显然限制了此语言在可靠系统的结构化编程中的使用。然而，可以利用语言的宏设施提供某种形式的异常处理机制。为说明这种方法，考虑实现一个简单的类似于Ada的异常处理宏。这个方法基于Lee（1993）所给的方法。

为用C实现终止模型，必须在异常定义域的入口处保存程序寄存器的状态等等，并在异常发生时恢复它们。传统上C已经同Unix联系在一起，且POSIX的setjmp和longjmp可用于此目的。例程setjmp保存程序状态，并返回0；例程longjmp恢复程序状态，并导致程序放弃它当前的执行，自setjmp被调用的位置重新启动。然而，这一次setjmp返回由longjmp传送的值，需要下面的代码结构：

```
/*异常域开始 */

typedef char *exception;
  /* 指向字符串的一个指针类型*/
exception error = "error" ;
  /* 一个名叫 "error" 的异常的表示 */

if ( (current_exception = (exception) setjmp (save_area) ) == 0) {
  /* 保存寄存器等到save_area */
  /*返回 0 */

  /*守备区 */

  /* 当验明了一个异常 "error" 时 */
  longjmp (save_area, (int) error);
  /* 无返回 */

}
else {
  if (current_exception == error){
    /* "error" 的处理程序 */
  }
```

164
```
      else {
      /* 在周围域中再次引发异常*/
    }
  }
```
上面的代码显然难以理解。不过，可以定义一组宏来使代码结构化。

```
#define NEW_EXCEPTION (name) ...
  /* 声明一个异常的代码 */
#define BEGIN ...
  /* 进入一个异常域的代码*/
#define EXCEPTION ...
  /* 开始异常处理程序的代码*/
#define END ...
  /* 离开一个异常域的代码*/
#define RAISE (name) ...
  /* 引发一个异常的代码*/
#define WHEN (name) ...
  /* 处理程序的代码 */
#define OTHERS ...
  /* 捕获所有异常处理程序的代码 */
```
考虑下面的例子:

```
  NEW_EXCEPTION (sensor_high);
  NEW_EXCEPTION (sensor_low);
  NEW_EXCEPTION (sensor_dead);
  /* 其他声明 */

BEGIN
  /* 可能引起上述异常被引发的语句, 例如 */
  RAISE (sensor_high);

EXCEPTION
  WHEN (sensor_high)
    /* 采取某个矫正动作 */
  WHEN (sensor_low)
    /*采取某个矫正动作*/
  WHEN (OTHERS)
    /* 响起警铃 */
END;
```
以上提供了一个类似于Ada的简单终止模型。

6.4 其他语言中的异常处理

在本书中，注意力主要是集中在Ada、Java、C和occam2语言上。当然，还有许多其他语
165 言用于实时应用，但不可能在一本书里全面覆盖它们。不过本书的宗旨之一是讨论其他语言的某些具体特征，如果它们有特色或重要的话。所以，本节简要评述CHILL、CLU、Mesa和C++的异常处理。

6.4.1 CHILL

CHILL同Ada类似的方面是：异常是语言定义的一部分并支持异常处理的终止模型。然而，

它提供的其他设施却与Ada显著不同。CHILL在类型名字和对象声明的语法方面同本书中考虑的任一种其他语言都稍不相同，所以，在这里给出的例子中将使用类似Ada的语法。

如在6.2.1节指出过的，CHILL中异常处理的域是语句、块或进程。所以，CHILL和Ada的主要不同是CHILL中异常处理程序可以追加在语句的末尾。CHILL不用关键字*exception*，而是用*on*和*end*把一个处理程序包围起来。在这两个关键字之间，语法类似于*case*语句。下面的例子说明了一个CHILL异常处理程序。

```
-- 用于说明概念的类Ada语法，不是合法的CHILL语法
declare
  subtype temperature is integer range 0 .. 100;

  A : temperature;
  B, C : integer;
begin
  ...
  A := B + C  on
              (overflow) :  .....;
              (rangefail) : .....;
              else      ....;
            end;
  ...
end on
       -- 这个块的异常处理程序
      (overflow) :  .....;
      (rangefail) : .....;
      else      ....;
    end;
```

CHILL定义若干与Ada类似的标准异常。在上面的例子中，在把B和C加起来时可能引发（算术）*overflow*异常；此外，如果赋给A的结果在0到100这个范围之外，可能发生一个*rangefail*异常。*else*选项等价于Ada的**when others**。

异常同其处理程序的关联是静态的，所以必须在编译时是已知的。所以，异常不需预先声明。然而，可由过程的过程首部中的适宜声明返回它们。再次使用类似Ada的语法: 166

```
procedure push (x: in item) exception (stack_full);
procedure pop (x: out item) exception (stack_empty);
```

对在语句S执行中发生的异常E，静态地为其确定一个处理程序的规则是:

- 它必须附加于S，或
- 它必须附加于直接外包S的块，或
- 它必须附加于直接外包S的过程，或
- 直接外包的过程必须将E定义在它的规格说明里，或
- 它必须附加于直接外包S的进程

当使用上述规则找不到异常处理程序时，程序就有错了，这里没有异常的传播。

6.4.2 CLU

CLU是一个实验性语言，虽然不是一个“实时”语言，却确实有某些有趣的特征，其中一个就是异常处理机制（Liskov and Snyder，1979）。它同C++和CHILL相似的是过程声明它

可能引发的异常。它也允许向处理程序传递参数。

例如，考虑函数`sum_stream`，它从字符流中读入一个有符号的十进制整数序列，返回这些整数的和。可能有下列异常：`overflow`（溢出）——和数超出了整数的实现范围；`unrepresentable_integer`（不可表示的整数）——字符流中的一个数超出了整数的实现范围；`bad_format`（不良格式）——字符流中包含非整数字段。对于后两种异常，违规字符串被传给处理程序。

```
sum_stream = proc (s : stream) return (int)
                 signals
                   (overflow,
                    unrepresentable_integer (string) ,
                    bad_format (string)
                   )
```

与CHILL一样，没有异常传播，异常必须在调用点上处理。

167

```
x = sum_stream
      except
        when overflow:
          S1
        when unrepresentable_integer (f : string) :
          S2
        when bad_format (f:string) :
          S3
      end
```

其中，`S1`、`S2`和`S3`是任意语句序列。

6.4.3 Mesa

在Mesa，异常被称为**信号**（signal）。因为它们能被传播，所以是动态的，而且它们的处理程序遵守混合模型。像Ada异常一样，信号必须被声明，但不同的是，其声明类似于过程类型，而不是常量声明。所以，它们可以有参数和返回值。当然，信号过程体就是异常处理程序。与过程体在编译时*静态地*同过程绑定在一起不同，异常处理程序是在运行时*动态地*同信号绑定在一起的。

处理程序可以同块相关联，这类似于Ada、C++和CHILL。像Ada一样，Mesa过程和函数不能指明它们能返回哪些信号，然而，处理程序可与过程调用相关联，给出类似效果。

有两类信号声明：使用关键字`signal`和使用`error`。被声明为错误（error）的信号不能由其相应的处理程序恢复。

6.4.4 C++

C++类似于Java，只是它不要求异常的显式声明，而是类的任何实例都可被作为异常抛出。它没有预定义异常。

一般说来，每个函数可以指明：

- 一个可抛出的对象列表，`throw (A, B, C)`——在这种情况，函数可抛出列表中的任何对象
- 可抛出对象的空列表，`throw ()`——这种情况下函数不抛出任何对象
- 没有可抛出对象的列表——这时，函数可抛出任何对象

如果函数试图抛出一个对象，而该对象又不是该函数的抛出列表所允许的，则会自动调用函数unexpected。unexpected的默认动作是调用terminate函数，它的默认动作是中止程序。这两个函数都可以通过下列调用使它们的默认操作被覆盖： 168

```
typedef void (*PFV)  ();

PFV set_unexpected (PFV new_handler);
/* 将默认动作置为new_handler, 并返回前一动作 */

PFV set_terminate (PFV new_handler);
/* 将默认动作置为new_handler, 并返回前一动作 */
```

类似于Java，C++有**catch**语句，它的参数标识要捕获的对象的类型。

带参数类型T的处理程序将捕获类型E的被抛出对象，前提

1）T和E是同一类型；

2）T是一个指针类型，而E是一个在抛出点可由C++标准指针转换器转换成T的指针类型；

3）T是一个指针类型，E是T指向的类型的对象——这时创建一个指向那个被抛出对象的指针，指针和对象二者都存在，直到退出该异常处理程序；

4）在抛出点T是E的基类。

可按与Java类似的方式构造异常层次体系。

6.5 恢复块和异常

在第5章中，恢复块是作为容错程序设计的机制引进的。它相对于向前出错恢复机制的主要优点是它可被用于从预见不到的错误恢复，特别是从软件部件的设计错误中恢复。本章到目前为止，只考虑了预见到的错误，虽然“全捕获”异常处理程序可被用于捕捉未知异常。本节描述使用异常和异常处理程序实现恢复块。

作为提醒，将恢复块的结构展示如下：

```
ensure <接受测试>
by
  <基本模块>
else by
  <替代模块>
else by
  <替代模块>
  .
  .
  .
else by
  <替代模块>
else error
```

169

出错检测设施是由接受测试提供的。这个测试不过就是那种使用向前出错恢复会引发异常的测试的“非”。剩下的问题只是实现状态保存和状态恢复。在下面的例子中，这被表示成一个实现恢复高速缓存的Ada包。过程save将程序的全局和局部变量的状态保存到这个恢复高速缓存中，这不包括程序计数器的值、栈指针等等。对Restore的调用将重置程序变量为保存的状态。

```
package Recovery_Cache is
  procedure Save;
  procedure Restore;
end Recovery_Cache;
```

显然，这个包里有点窍门，它需要运行时系统的支持，甚至可能还要为恢复高速缓存提供硬件支持。还有，这可能不是实现状态恢复最高效的方法。更为理想的是提供更多的基本原语，并允许程序使用其应用知识来优化已保存的信息数量（Rogers and Welling，2000）。

下一个例子的目的是要证明：如果有恢复高速缓存实现技术，就可在异常处理环境中使用恢复块。还要注意，通过使用异常处理程序，在恢复状态之前就能实现向前出错恢复。这克服了对恢复块的一种批评：难于重置环境。

所以恢复块方案可用带异常的语言加上来自底层运行时支持系统的少量帮助实现。例如，用Ada编写的三重冗余恢复块的结构可以是：

170

```
procedure Recovery_Block is
  Primary_Failure, Secondary_Failure,
          Tertiary_Failure: exception;
  Recovery_Block_Failure : exception;
  type Module is (Primary, Secondary, Tertiary);
  function Acceptance_Test return Boolean is
  begin
    -- 接受测试的代码
  end Acceptance_Test;

  procedure Primary is
  begin
    -- 基本算法的代码
    if not Acceptance_Test then
      raise Primary_Failure;
    end if;
  exception
    when Primary_Failure  =>
      -- 将环境向前恢复到所需状态
      raise;
    when others  =>
      -- 非预期错误
      -- 将环境向前恢复到所需状态
      raise Primary_Failure;
  end Primary;

  procedure Secondary is
  begin
    -- 次要算法代码
    if not Acceptance_Test then
      raise Secondary_Failure;
    end if;
  exception
    when Secondary_Failure  =>
      -- 将环境向前恢复到所需状态
      raise;
    when others  =>
```

```
        -- 非预期错误
        -- 将环境向前恢复到所需状态
        raise Secondary_Failure;
    end Secondary;

    procedure Tertiary is
    begin
      -- 第三位算法的代码
      if not Acceptance_Test then
        raise Tertiary_Failure;
      end if;
    exception
      when Tertiary_Failure   =>
        -- 将环境向前恢复到所需状态
        raise;
      when others  =>
        -- 非预期错误
        -- 将环境向前恢复到所需状态
        raise Tertiary_Failure;
    end Tertiary;

  begin
    Recovery_Cache.Save;
    for Try in Module loop
      begin
        case Try is
          when Primary => Primary; exit;
          when Secondary => Secondary; exit;
          when Tertiary => Tertiary;
        end case;
      exception
        when Primary_Failure =>
          Recovery_Cache.Restore;
        when Secondary_Failure =>
          Recovery_Cache.Restore;                                          171
        when Tertiary_Failure =>
          Recovery_Cache.Restore;
          raise Recovery_Block_Failure;
        when others  =>
          Recovery_Cache.Restore;
          raise Recovery_Block_Failure;
      end;
    end loop;
  end Recovery_Block;
```

小结

本章研究了顺序过程的各种异常处理模型。虽然有许多不同模型存在，但它们都致力于下述问题：

- 异常表示——异常可以在语言中显式表示，也可以不。
- 异常处理程序的定义域——同每个异常处理程序关联的有一个定义域，它规定一个计算

区域，如果在此区域引发异常，将激活这个处理程序。定义域一般是同块、子程序或语句相关联的。

• 异常传播——这同异常定义域的思想密切相关。当一个异常被引发时，可能在包围的定义域中没有异常处理程序。这时，或者异常被传播到下一个外层包围定义域，或者被认为是程序员错误（它常常能在编译时标志出来）。
• 恢复或终止模型——这决定了在处理异常后要进行的动作。在恢复模型中，异常的调用者在调用异常的后一个语句处被恢复。对于终止模型，包含处理程序的块或过程被终止，控制被传给调用块或过程。混合模型使处理程序能够选择是恢复还是终止。
• 传给处理程序的参数——允许或不允许。

各种语言的异常处理设施总结在表6-1中。

表6-1 多种语言的异常处理设施

语 言	定 义 域	传 播	模 型	参 数
Ada	块	是	终止	受限制
Java	块	是	终止	是
C++	块	是	终止	是
CHILL	语句	否	终止	否
CLU	语句	否	终止	是
Mesa	块	是	混合	是

对于是否应当在语言中提供异常处理设施的意见不一致。例如，C和occam2中没有。对于怀疑者，异常类似于`goto`，其目的地是不可确定的，其来源也是未知的。所以，它们可被看作是与结构化程序设计对立的。然而，本书不持此观点。

相关阅读材料

Cristian, F. (1982) Exception Handling and Software Fault Tolerance. *IEEE Transactions on Computing*, c-31(6), 531–540.

Cui, Q. and Gannon, J. (1992) Data-oriented Exception Handling. *IEEE Transactions on Software Engineering*, 18(5), 393–401.

Lee, P. A. (1983) Exception Handling in C Programs. *Software – Practice and Experience*, 13(5), 389–406.

Papurt, D. M. (1998) The Use of Exceptions. *JOOP*, 11(2), 13–17.

练习

6.1 将异常处理以及软件容错的恢复块方法进行比较和对照。

6.2 给出下列几种异常的例子：

（1）由应用检测的同步异常

（2）由应用检测的异步异常

（3）由执行环境检测的同步异常

（4）由执行环境检测的异步异常

6.3 包`Character_Io`的规格说明如下，它提供一个函数，从终端读取字符。它还提供一个

过程，抛弃当前行上所有剩余的字符。此包可引发异常Io_Error。 172～173

```
package Character_Io is
  function Get return Character;
    -- 从终端读字符

  procedure Flush;
    -- 抛弃当前行上的所有字符

  Io_Error : exception;
end Character_Io;
```

另一个包Look包括函数Read，它能扫描当前输入行，寻找标点字符逗号（,）、句号（.）和分号（;）。此函数将返回找到的下一个标点符号，或引发异常Illegal_Punctuation（如果找到一个非字母数字字符的话）。如果在扫描输入行的过程中，遇到了一个Io_Error，则异常被传播给Read的调用者。Look的规格说明如下:

```
with Character_Io; use Character_Io;
package Look is
  type Punctuation is (Comma, Period, Semicolon);
  function Read return Punctuation;
    -- 从终端上读下一个，. 或;

  Illegal_Punctuation : exception;
end Look;
```

Look的包体大致上是使用Character_Io包从终端读入字符。在收到合法的标点字符、非法的标点字符或Io_Error异常的时候，应当将输入行的剩余部分丢掉。你可以假设输入行总是有合法的或非法的标点字符并随机地出现Io_Error。使用Look包，勾画出过程Get_ Punctuation的代码，它总是返回下一个标点字符而不管Look引发的异常。可以假设无穷的输入流。

6.4 在一个过程控制应用中，气体在一个封闭的箱中加热。箱子用冷却剂包住，以通过传导降低气体的温度。还有一个阀，打开时释放气体到大气中去。过程的操作由一个Ada包控制，它的规格说明在下面给出。为安全起见，这个包识别各种出错情况，通过引发异常使包的用户引起注意。异常Heater_Stuck_On由过程Heater_Off在不能关掉加热器的时候引发。异常Temperature_Still_Rising由过程Increase_Coolant在不能通过增加冷却剂使温度降低的时候引发。最后，异常Valve_Stuck由过程Open_Valve在不能使气体释放到大气中的时候引发。 174

```
package Temperature_Control is
   Heater_Stuck_On, Temperature_Still_Rising,
          Valve_Stuck : exception;

 procedure Heater_On;
   -- 打开加热器

 procedure Heater_Off;
   -- 关闭加热器
   -- 引发 Heater_Stuck_On

 procedure Increase_Coolant;
```

```
    -- 使环绕箱子的冷却剂流增加，直到温度到达
    -- 一个安全级别
    -- 引发 Temperature_Still_Rising

  procedure Open_Valve;
    -- 打开阀，以释放一些气体，避免爆炸
    -- 引发 Valve_Stuck

  procedure Panic;
    -- 响起警铃，并呼叫消防、医务和警察服务
end Temperature_Control;
```

写一个Ada过程，它被调用的时候将尝试关闭气体箱中的加热器。如果加热器关不了，则环绕在箱子周围的冷却剂流应当增加。如果温度依然上升，则脱逸阀应被打开以释放气体。如果这一点失败，警报应响起并通知紧急服务。

6.5 写一个通用目的类属Ada包，以实现嵌套的恢复块。（提示：基本模块、次级模块、接受测试等等应被封装在一个包中，并作为参数传送给这个类属包。）

6.6 在语言之外可由标准包实现Ada异常处理设施的哪些方面？

6.7 你怎么驳斥这种说法：Ada异常是一个goto，其目的地不可确定且来源是未知的。同样的论点对（a）异常处理的恢复模型和（b）CHILL终止模型成立吗？

6.8 比较下面显然等同的代码片断（变量Initial属于整型）的异常定义域：

```
procedure Do_Something is
  subtype Small_Int is Integer range -16..15;
  A : Small_Int := Initial;
begin
  ...
end Do_Something;
```

175

```
procedure Do_Something is
  subtype Small_Int is Integer range -16..15;
  A : Small_Int;
begin
  A := Initial;
  ...
end Do_Something;

procedure Do_Something is
  subtype Small_Int is Integer range -16..15;
  A : Small_Int;
begin
  begin
    A := Initial;
    ...
  end;
end Do_Something;
```

6.9 说明6.3.3节给出的C语言宏是能够实现的。

6.10 考虑下列程序：

```
I : Integer := 1;
J : Integer := 0;
```

```
K : Integer := 3;

procedure Primary is
begin
  J := 20;
  I := K*J/2
end Primary;

procedure Secondary is
begin
  I := J;
  K := 4;
end Secondary;

procedure Tertiary is
begin
  J := 20;
  I := K*J/2
end Tertiary;
```

这段代码准备在恢复块环境和异常处理环境中执行。下面是恢复块环境:

```
ensure I = 20
by
  primary;
else by
  Secondary;
else by
  Tertiary;                                          176
else
  I := 20;
end;
```

下面是异常处理环境:

```
Failed : exception;
type Module is (P, S, T);

for Try in Module loop
  begin
    case Try is
      when P =>
        Primary;
        if I /= 20 then
          raise Failed;
        end if;
        exit;
      when S =>
        Secondary;
        if I /= 20 then
          raise Failed;
        end if;
        exit;
      when T =>
```

```
        Tertiary;
        if I /= 20 then
          raise Failed;
        end if;
        exit;
      end case;
  exception
    when Failed =>
      if Try = T then
        I := 20;
      end if;
  end;
end loop;
```

假设使用异常处理的终止模型，比较和对照恢复块环境和异常处理环境两个代码段的执行。

6.11 说明怎样用C++和Java实现恢复块。提示见参考文献[Rubira-Calsavara and Stroud（1994）]。

6.12 用Java重做习题6.3。

6.13 用Java重做习题6.4。

6.14 解释在什么情况下Ada中的`when others (e: Exception_Id)`不等价于Java中的`catch (Exception e)`。

177 6.15 将异常集成到OOP模型的优点和缺点是什么？

6.16 已经为安全至上的嵌入式系统定义了Ada 95的一个子集。该语言有形式化语义和一组能使用静态分析技术的工具。它没有异常处理设施。设计者争辩说如果程序被证明是正确
178 的，异常就不可能发生。对于将异常处理设施引进这个语言，能够提出些什么论据呢？

第7章 并发编程

所有实时系统实质上都是并发的。对于打算用于此领域的语言来说，如果它们提供给程序员的原语同应用的并行性相匹配，那就会有更强的表达能力。

> 并发编程是一种表达潜在并行性和解决由此产生的同步和通信问题的编程记号和技术的名字。并行性的实现是计算机系统（硬件和软件）中的课题，它本质上同并发编程无关。并发编程是重要的，因为它提供一种用以研究并行性而不陷入实现细节的抽象框架（Ben-Ari，1982）。

本章及随后两章，集中讨论同通用并发编程相关联的问题。时间及其在程序中的表示和处理放到第12章讨论。

7.1 进程概念

无论是自然语言还是计算机语言都有双重性质：一方面是能表达，一方面是限制可以应用这种表达能力的框架。如果一个语言不支持一个特定观念或概念，那么使用该语言的人就不能应用那个概念，并且甚至完全没有意识到它的存在。

Pascal、C、FORTRAN和COBOL都是顺序编程语言，它们共享一些共同属性。用这些语言写的程序只有一条控制线程，它们从某个状态开始执行并前进，一次执行一个语句，直到程序终止。程序经由的路径可因输入数据的变化而不同，但对程序的任何特定执行只有一条路径。这一点对于实时系统的编程是不够的。 179

按Dijkstra（1968a）的探索性工作的说法，并发程序习惯上被看成由一个自治顺序进程的集合组成，这些进程（在逻辑上）是并行执行的。并发程序设计语言都显式或隐式地加进了进程概念，每个进程本身有一个控制线程⊖。

进程集合的实际实现（即执行）通常有三种形式：

1）进程在单处理器上多路执行。

2）进程在多处理器系统上多路执行，访问共享的存储器。

3）进程在几个处理器上多路执行，不共享存储器（这种系统通常叫做分布式系统）。

还可以是这三种方式的混合形式。

⊖ 操作系统的近期工作提出了线程和多线程进程的概念。本节稍后会再回到这个问题。

只有在（2）和（3）情况下，一个以上进程的真正并行执行才有可能。并发这个术语指潜在并行性。并发编程语言使程序员能表达逻辑上并行的活动，而无需关心其实现。有关真正并行执行的问题在第14章考虑。

图7-1简单说明了进程的寿命。一个进程被创建，进入初始化状态，直到执行并终止。注意，有些进程可能永远不终止，而另一些进程可能在初始化时就失败，从未执行就直接终止。终止后，进程就不存在，不再能被访问（就像走出了它的作用域）。显然，进程的最重要状态是执行，然而，由于处理器数量的限制，并非所有进程能立即执行。所以**可执行的**（executable）这个术语用于指出：如果有一个处理器可用的话，进程就能够执行。

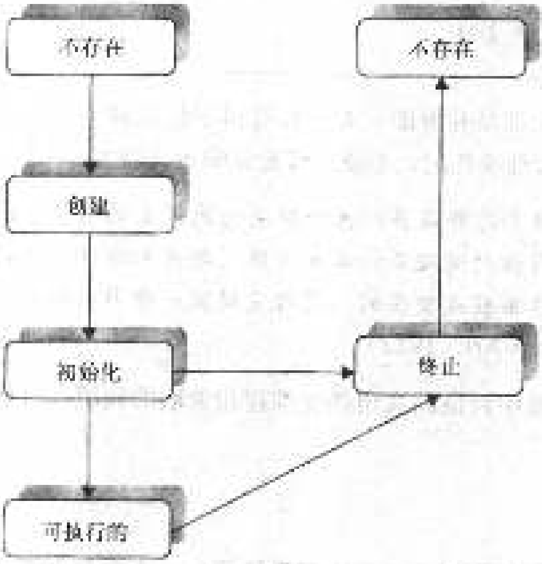


图7-1 进程的简单状态图

从对于进程的这种考虑来看，显然，并发程序的执行不像顺序程序的执行那样直截了当。进程必须被创建，被终止，并分派给可用处理器或从可用的处理器上离开。这些活动是由运行时支持系统（Run-Time Support System，RTSS）或运行时内核（Run-Time Kernal）完成的。RTSS具有操作系统中的调度程序的许多性质，在逻辑上位于硬件和应用软件之间。事实上，它可取下列形式之一：

1）作为应用的一部分的一个软件结构（即作为并发程序的一个部件），这是Modula-2语言采用的方法。

2）和程序目标码一起由编译器生成的标准软件系统。这是Ada和Java程序的结构。

3）在处理器里的微码化硬件结构（为了效率），运行在传输机上的occam2程序有这样的运行时系统。

RTSS的调度算法（即决定下一个要执行的进程，如果当前有多个可执行进程的话）将影响程序的时间行为，虽然对于构造良好的程序，程序的逻辑行为不依赖于RTSS。从程序的观点看，假设RTSS是非确定性地调度进程。对于实时系统，调度的特性十分重要。在第13章进一步讨论它们。

虽然本书的关注重点是并发实时语言，显然，一个替代的方法是使用顺序语言和实时操

作系统。所有操作系统都提供创建并发进程的设施。通常，每个进程在它自己的虚拟机上执行，以避免来自其他无关进程的干预。事实上，每个进程是个单一的程序。然而，近几年有趋势是提供在程序内创建进程的设施。现代操作系统允许在同一个程序内创建的进程对共享存储器进行无限制的访问（这样的进程常常叫做**线程**（thread））。所以，在那些遵循POSIX的操作系统里，必须区分程序（进程）之间的并发性和在程序（线程）内的并发性。通常，操作系统可见的线程和单独由库例程支持的线程之间也有区别。例如Windows 2000支持线程和**线丛**（fibers），后者对于内核是不可见的。 180~181

对于并发性的支持是由语言提供，还是只应由操作系统提供，在程序员、语言设计者和操作系统设计者之间已经有很长时间的争论。支持把并发性放进编程语言中去的论据有:

1）使程序较易读和易维护。

2）有多种不同的操作系统，在语言中定义并发性使程序更可移植。

3）嵌入式计算机可能没有驻留的操作系统可用。

这些争论显然对Ada和Java设计者们很有分量。反对把并发性放进编程语言中去的论据有:

1）不同语言有不同的并发模型，如它们都用并发性的同一操作系统模型，就更易于将不同语言的程序组合起来。

2）在操作系统的模型之上很难有效地实现一个语言的并发模型。

3）正在产生操作系统的标准，因此程序会变得更可移植。

在民用航空工业开发集成模块化航空电子学项目（Integrated Modular Avionics）的时候，选用了一个标准应用核心界面（叫做APEX）支持并发性，而没有选用Ada的并发性模型（ARINC AEE Committee，1999）。需要支持多种语言是这种选择的主要原因之一。毫无疑问，这种争论还会持续一段时间。

并发编程构造

虽然并发编程的构造在不同语言（和操作系统）之间各不相同，但是有三种基本设施是必须提供的。它们用以支持:

1）通过进程概念表达并发执行

2）进程同步

3）进程间的通信

在考虑进程交互时，需要区分三类行为: 182

- 独立的
- 合作的
- 竞争的

独立进程相互不通信也不同步。与之相比，合作进程则通常要定期通信和同步以完成某些共同的活动。例如，嵌入式计算机系统的一个部件可能有多个进程涉及到把容器中气体的温度和湿度保持在某个界限之内。这就可能需要频繁的交互。

计算机系统拥有有限的资源，它们可以在进程之间共享，例如，外部设备、存储器和处理器能力。为了使进程得到这些资源的公平共享，它们必须互相竞争。资源分配的活动不可避免地需要系统中进程之间的通信与同步。但是，虽然这些进程为得到资源而通信和同步，它们本质上却是独立的。

后续三章的重点是讨论支持进程创建和进程交互的设施。

7.2 并发执行

虽然进程的概念对所有并发编程语言是共同的，但它们采用的并发模型却有可观的差别。这些差别是关于:

- 结构
- 层次
- 粒度
- 初始化
- 终止
- 表示

可将进程的结构按如下分类:

- 静态的：进程数目是固定的，并在编译时已知。
- 动态的：进程可在任何时刻创建。现存的进程数目只在运行时才决定。

语言之间的另一个区别来自支持的并行层次。也可分成两种情况:

183 1）嵌套的：进程可在程序正文的任何层次定义，特别是进程可在其他进程中定义。

2）平坦的：进程只在程序正文的最外层定义。

表7-1给出了若干编程语言的结构和层次特征。在此表中，C语言被合并到了POSIX的“分叉”和“等待”原语。

表7-1 若干并发编程语言的结构和层次特征

语言	结构	层次
Concurrent Pascal	静态	平坦
occam2	静态	嵌套
Modula-1	动态	平坦
Modula-2	动态	平坦
Ada	动态	嵌套
Java	动态	嵌套
Mesa	动态	嵌套
C/POSIX	动态	平坦

在支持嵌套构造的语言中，还有一个有趣的区别，即所谓的**粗粒度**和**细粒度**并行性。粗粒度并发程序只包含少量的进程，每个进程都有一个显著的生存史。比较起来，细粒度并行性程序具有大量的简单进程，其中有一些只为单一的动作存在。大多数并发编程语言，以Ada为代表，具有粗粒度并行性。occam2是具有细粒度并行性的并发语言的好例子。

当创建进程时，可能需要为它的执行提供一些有关的信息（很像在过程被调用时需要提供的信息）。有两种方式进行这种**初始化**。第一种是以参数形式给进程传递信息；第二种是在进程开始执行后与它显式通信。

进程**终止**能够以各种不同的方式完成，允许进程终止的情况可简单概括如下:

1）进程体执行完毕;

2）自灭，通过执行“自我终止”语句;

3）中止，经由另一进程的显式动作;

4）非捕获出错条件的出现；

5）从不：假定进程执行一个无终止循环；

6）不再需要。

通过使用嵌套层次，可建立进程的分级体系，并形成进程之间的关系。对任何进程，区分负责创建它的进程（或块）和受其终止影响的进程（或块）很有用。前一种关系被称为父/子（parent/child）关系，并有这种属性：当子进程正被创建和初始化时，父进程可能被延迟。后一种关系被称为监护者/眷属（guardian/dependant）关系。一个进程可能依赖于监护者进程本身或监护者进程的内层块。直到块的所有依赖进程都已终止，监护者进程才能从这个块退出（也就是说，一个进程在其作用域外不存在）。由此推论出：监护者在其所有依赖者都已终止之前不得终止。这个规则有一个特别的推论：一个程序在它里面创建的所有进程终止之前不得终止。 184

在某些情况下，进程的父进程可能就是它的监护者。对于只允许静态进程结构的语言（例如occam2），就是这种情况。对于动态进程结构（也嵌套），父进程和监护者可能相同，也可能不同。这一点会在稍后有关Ada的讨论中阐明。

图7-2包括了上述讨论引入的新状态。

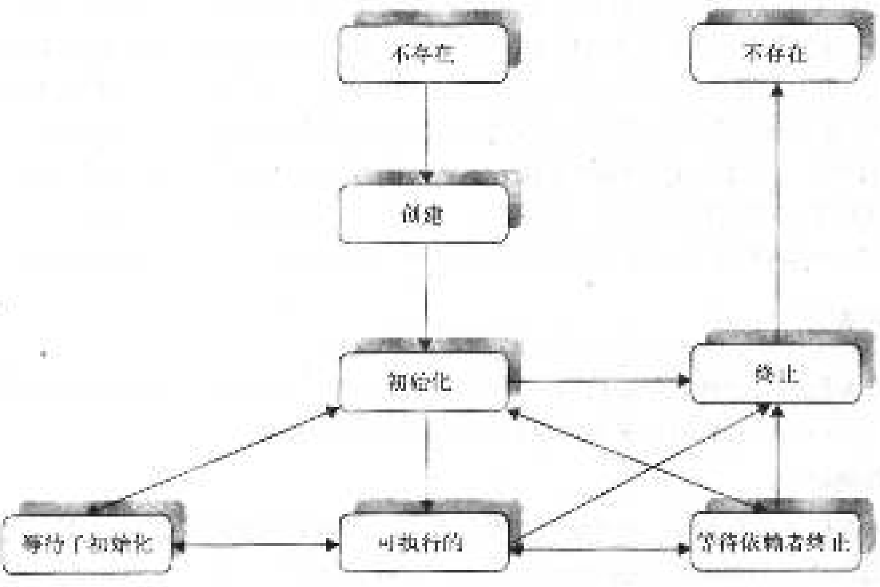


图7-2 进程的状态图

进程的终止方式之一是用应用的中止语句（上述（3））。并发编程语言中中止的存在性是一个有争论的问题，并放在第11章有关资源控制的上下文中研究。对于进程的分级体系而言，监护者的中止通常是必要的，它隐含着其所有依赖者（和它们的依赖者等等）的中止。 185

在上述清单中终止的最后一种情况将在描述进程通信方法的地方更详细地考虑。从本质上说，它允许一个服务器进程终止，如果所有能同它通信的进程已经终止的话。

进程与对象

面向对象编程模式鼓励系统（或程序）建造者将正在构建的东西看成是一组合作对象的（或者用一个更中性的术语，实体）集合。在此模式中，可考虑两类对象——主动的和反应式

的。**主动**对象执行自发的动作（在处理器的帮助下）：它能使计算继续。与之相比，**反应式**对象只在被主动对象“调用”时才执行动作。另一些编程模式（诸如数据流或实时网络）区分主动的代理和被动的数据。

只有主动实体引起自发动作。资源是反应式的，但可以控制对它们的内部状态（和它们控制的所有实在资源）的访问。某些资源在一个时刻只能被一个代理使用；在其他情况下，在给定时刻可以进行的操作取决于资源的当前状态。后者的常见例子是数据缓冲区，如果它是空的，就不能从中取出元素。**被动**这个术语被用于指明那种能够允许开放式访问的反应式实体。

资源实体的实现需要某种形式的控制代理。如果控制代理本身是被动的（像信号量），则此资源被称为**保护式的**（或**同步式**）。或者，如果需要主动代理去编码正确的控制级别，则这个资源在某种意义上是主动的。**服务器**这个术语被用于标识这种类型的实体，而**保护性资源**用于指示被动资源。这两个术语，连同**主动**和**被动**，是本书中用到的四种抽象程序实体。

在并发编程语言中，主动实体是由进程表示的。被动实体可被直接表示成数据变量或由某种模块/包/类构造封装，它们提供一个过程性接口。保护性资源也可被封装在像模块的构造中，并需要用到低级同步设施。服务器，由于需要编制控制代理，需要进程。

语言设计者的一个关键问题是是否对保护性资源和服务器的原语都提供支持。由于资源在典型情况下使用低级控制代理（例如，信号量），通常它们都能高效地实现（至少在单处理器系统上）。但它们可能不灵活，并对某些种类的问题产生不良的程序结构（这在第8章中进一步讨论）。服务器，由于其控制代理用进程编制，是特别灵活的。这种方法的缺点是它可能导致进程的增生，带来执行期间的大量上下文切换。如果语言不支持保护性资源，因而对所有这样的实体就必须使用服务器，这就特别成问题了。如我们将在本章和第8章、第9章阐明
186 的，Ada、Java和POSIX支持所有这些种类的实体。occam2则不同，只支持服务器。

7.3 进程表示

表示并发执行有三种基本的机制：分叉与汇合（fork and join）、cobegin和显式进程声明。合作例程（coroutine）有时也被用作一种表达并发执行的机制。

7.3.1 合作例程

合作例程同子例程相像，但允许在它们之间以一种对称的而不是严格层次化的方式显式传递控制。借助于resume语句使控制从一个合作例程传递给另一个合作例程，resume语句指出要恢复的合作例程的名字。当合作例程执行恢复时，它就停止执行并保留局部状态信息，这样若有另一个合作例程随后“恢复”它，它就能继续执行。图7-3说明了三个合作例程的执行，编了号的箭头代表控制流：从合作例程A开始，在合作例程B完成（两次访问了合作例程C）。

每个合作例程可以被看作实现一个进程，然而它们不需要运行时支持系统，因为合作例程自身排好了执行顺序。显然，合作例程对于真正的并行处理是不合适的，因为它们的语义
187 只允许一次执行一个例程。Modula-2是支持合作例程的例子。

7.3.2 分叉与汇合

这种简单方法不提供进程的可见实体，而只支持两个语句。fork语句指明一个指名的例程应当同该fork语句的调用者并发执行。join语句允许调用者同被调用例程的完成同步。例如：

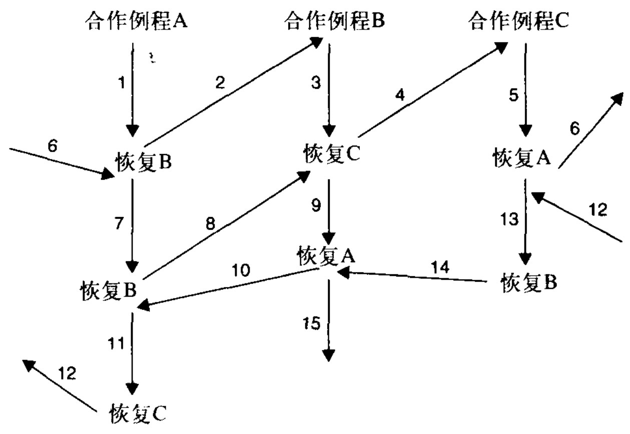


图7-3 合作例程控制流

```
function F return...;
.
.
.
end F;
procedure P;
  ...
  C:= fork F;
    .
    .
    .
  J:= join C;
  ...
end P;
```

在fork和join的执行中间，过程P和函数F是并发执行的。在汇合点，该过程将等待函数的完成(如果还没有完成的话)。图7-4说明了分叉与汇合的执行。

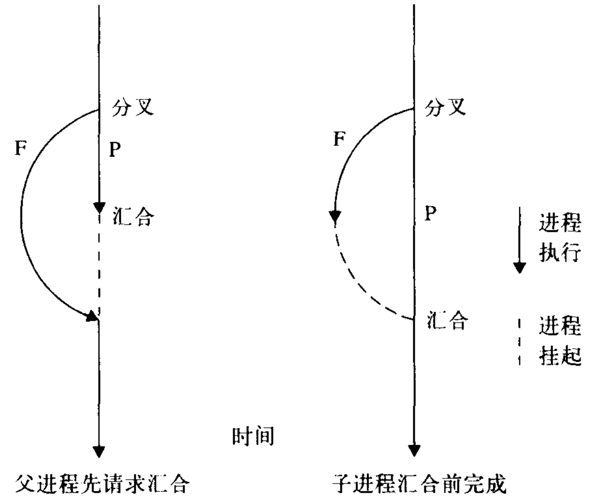


图7-4 分叉和汇合

分叉和汇合记号可在Mesa语言中找到。分叉和汇合的另一个版本也可在POSIX中找到，在这里，`fork`用于创建调用者的副本，而`wait`系统调用提供有效的汇合。

分叉和汇合允许动态进程创建，并提供通过参数向子进程传送信息的手段。通常子进程在终止时只返回一个单一的值。虽然灵活，但分叉和汇合没有为进程创建提供一个结构化方法，并在使用时易于出错。例如，监护者必须显式地使所有眷属“重新汇合”，而不只是等待它们的完成。

7.3.3 cobegin

cobegin（或者parbegin或者par）是表示一组语句的并发执行的一种结构化方式:

```
cobegin
  S1;
  S2;
  S3;
  .
  .
  .
  Sn
coend
```

这段代码使语句`S1`，`S2`等等并发地执行。当所有并发执行都已终止时cobegin语句终止。每个`Si`语句可以是语言允许的任何结构，包括简单赋值或过程调用。如果使用过程调用，数据可经由调用的参数传送给被调用进程。cobegin语句甚至可包含语句序列，而语句序列本身内部又有cobegin。可以以这种方式支持进程的层次体系。图7-5说明了cobegin语句的执行。

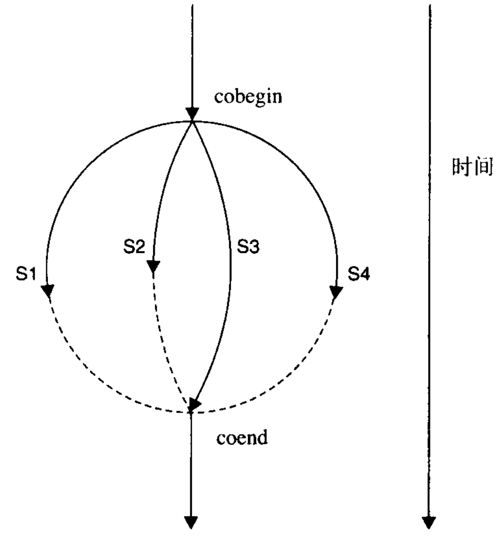


图7-5 cobegin

cobegin可在occam2中找到。

7.3.4 显式进程声明

虽然利用cobegin或fork能使顺序例程得以并发执行，但如果例程自身说出它们是否并发执行，那么并发程序的结构就会清楚得多。显式进程声明提供这种设施。下面的例子是用

Modula-1写的。它是机器人手臂控制器的一个简单结构。每一维的运动用一个单独的进程控制。这些进程循环往复，每次读入维的新设定，然后调用低级过程move_arm使手臂运动。

```
MODULE main;
  TYPE dimension = (xplane, yplane, zplane);

  PROCESS control (dim : dimension);
    VAR position : integer;      (* 绝对位置 *)
        setting : integer;       (* 相对运动 *)
  BEGIN
    position := 0;               (* 休息位置 *)
    LOOP
       move_arm (dim, position);
       new_setting (dim, setting);
       position := position + setting
    END
  END control;

BEGIN
  control (xplane);
  control (yplane);
  control (zplane)
END main.
```

在上述程序中，声明了进程control，并带有一个创建时要传递的参数。接着创建该进程的三个实例，每个实例传送一个不同的参数。 188~190

支持显式进程声明的其他语言还有隐式任务创建，例如Ada。在一个块结构中声明的所有进程在该块的声明部分末尾开始并发执行。

在上述讨论中，简述了并发执行的基本模型，列出了支持具体特性的语言。为了给出实际编程语言的具体例子，下面将研究occam2、Ada和Java的并发执行，接着考察POSIX的并发执行。

7.3.5 occam2的并发执行

occam2使用一个叫做PAR的cobegin结构。例如，考虑两个简单赋值（A:=1和B:=1）的并发执行:

```
PAR
  A : = 1
  B : = 1
```

可将此PAR结构（它可以嵌套）同顺序形式比较:

```
SEQ
  A : = 1
  B : = 1
```

注意，必须把一组动作显式定义成顺序执行还是并行执行。确实，可以认为PAR是一种更一般的形式，而且对使用来说应当是自然的结构，除非有问题的实际代码需要是顺序的。

如果用PAR指名的进程是一个参数化PROC（过程）的实例，则在进程被创建时可以传递数据给它。例如，下面的代码由同一个简单的PROC创建三个进程，并为每个进程传递一个数组元素:

```
PAR
  ExampleProcess (A[1])
  ExampleProcess (A[2])
  ExampleProcess (A[3])
```

还可以用一个“重复器”(replicator)增加PAR的能力，从同一个PROC创建一大堆进程:

```
PAR i=1 FOR N
  ExampleProcess (A[i])
```

在第3章，将SEQ和重复器一起使用，以引进标准的“for”循环。重复的SEQ和重复的PAR之间的惟一区别是重复次数（在上例中的N），对于PAR，它必须是个常数，即在编译时
191 知道它的值。因此，occam2具有静态进程结构。

下面的occam2程序段是前面为Modula-1给出的机器人手臂的例子。注意，occam2不支持枚举类型，所以用整数1、2、3表示维数。

```
PROC control (VAL INT dim)
  INT  position,                  -- 绝对位置
       setting:                   -- 相对运动
  SEQ
    position := 0                 -- 休息位置
    WHILE TRUE
      SEQ
        MoveArm (dim, position)
        NewSetting (dim, setting)
        position := position + setting
:
PAR
  control (1)
  control (2)
  control (3)
```

在occam2中，进程终止是十分直截了当的。既没有中止设施也没有任何异常设施。进程必须要么正常终止，要么根本不终止（如上例）。

一般说来，occam2具有细粒度并发性。大多数并发编程语言都为本质顺序性的框架填加进程概念。occam2不是这样的，进程概念是该语言的基础。所有活动，包括赋值操作和过程调用都被看作进程。确实，occam2中没有语句概念。一个程序就是单个进程，它是由其他进程的层次结构建立的。在最低层次，所有的基本动作被看作是进程（IF、WHILE、CASE等等），且构造器本身被看作是构造器进程。

7.3.6 Ada的并发执行

并行性的常规单元即顺序进程，在Ada中称为**任务**。任务可在任何程序层次声明，它们在进入它们的声明的作用域时被隐式创建。下例说明了包含两个任务（A和B）的过程:

```
procedure Example1 is
  task A;
  task B;

  task body A is
  -- 任务A的局部声明
  begin
```

```
  -- 任务A的语句序列
  end A;

  task body B is
  -- 任务B的局部声明
  begin
  -- 任务B的语句序列
  end B;

begin
  -- 在属于过程的语句序列的第一个语句之前，任务A 和 B 开始它们的执行
  .
  .
  .
end Examplel; -- 直到任务A和B终止，此过程才终止
```

192

任务像包一样，由规格说明和体组成。然而，可在创建时传递初始化数据给它们。

在上面的程序中，任务A和B（在过程被调用时创建）被说成属于无名类型，因为它们没有一个为它们声明的类型（与无名数组类型比较）。容易给A和B以类型:

```
task type A_Type;
task type B_Type;
A : A_Type;
B : B_Type;
task body A_Type is
  -- A任务体同前
task body B_Type is
  -- B任务体同前
```

有了任务类型，就可以用数组声明同一进程的多个实例:

```
task type T;
A, B : T;
type Long is array (1..100) of T;
type Mixture is
record
  Index : Integer;
  Action : T;
end record;
L : Long; M : Mixture;
task body T is ...
```

在Ada程序中使用任务的较具体例子是早些时候为Modula-1和occam2引入的机器人手臂系统。

```
procedure Main is
  type Dimension is (Xplane, Yplane, Zplane);
  task type Control (Dim : Dimension);
  C1 : Control (Xplane);
  C2 : Control (Yplane);
  C3 : Control (Zplane);

  task body Control is
    Position : Integer;      -- 绝对位置
```

193

```
      Setting : Integer;        -- 相对运动
    begin
      Position := 0;            -- 休息位置
      loop
        Move_Arm (Dim, Position);
        New_Setting (Dim, Setting);
        Position := Position + Setting;
      end loop;
    end Control;
  begin
    null;
  end Main;
```

值得注意的是，Ada只允许离散类型和访问类型作为任务的初始化参数传递。

通过给（任务）数组界以非静态值，就创造了动态个数的任务。动态任务创建可以通过对（任务类型的）访问类型使用“new”运算符显式地得到（译者注——即使用分配器）:

```
procedure Example2 is
  task type T;
  type A is access T;
  P : A;
  Q : A:= new T;
begin
  ...
  P:= new T;
  Q:= new T;
  ...
end Example2;
```

Q被声明为类型A，并被赋以T的新分配的“值”。这就创建了一个任务，并立即开始它的初始化和执行，此任务用**Q.all**指名（**all**是Ada的一个命名约定，用于指向任务本身，而非访问指针）。在此过程执行中，P被分配了一个任务（**P.all**），接着又分配了一个给Q。现在，在该过程中有三个任务在活动：**P.all**、**Q.all**和先前创建的任务。这个最先创建的任务现在无名，因为Q不再指向它。除了这三个任务，还有一个任务即执行过程代码本身的主程序，所以，总共有四条不同的控制线程。

由分配器（‘new’）操作创建的任务有一个重要性质：作为其监护者（Ada叫做**主宰**(master)）的块，并非是创建它的块而是包含有访问类型声明的块。为说明这一点，考虑下

194 面的程序段:

```
declare
  task type T;
  type A is access T;
begin
  .
  .
  .
  declare                -- 内层块
    X : T;
    Y : A:= new T;
  begin
```

```
      -- 语句序列
    end;  -- 必须等待X 终止，但不需等待Y.all终止
    .
    .      -- Y.all 可能仍然是活动的，虽然名字 Y 已经出了作用域
    .
end;       -- 必须等待Y.all 终止
```

虽然X和**Y.all**都在内层块创建，却只有X将此块作为其主宰。任务**Y.all**被看作是外层块的眷属，因而它不影响内层块的终止。

如果一个任务在初始化（Ada中叫做**激活**）时就失败了，则该任务的父任务就引发异常Tasking_Error。例如，如果给一个变量赋以不合适的初始值，就会发生这种情况。一旦任务开始真正执行，它就能捕获任何它引发的异常。

1. 任务标识

访问变量的一个主要用途是提供任务命名的另一种手段。Ada的所有任务类型都被看作是受限私有（limited private）的，所以不可能通过赋值将任务传递给另一数据结构或程序单元。例如，如果Robot_Arm和New_Arm是同一访问类型（该访问类型是从一个任务类型得到的）的两个变量，那么下列的语句是不合法的：

```
Robot_Arm.all :=New_Arm.all;       --非法Ada代码
```

然而，

```
Robot_Arm := New_Arm;
```

是合法的，并意味着现在Robot_Arm和New_Arm都指向同一任务。这里必须小心，因为重复命名会引起混乱，并导致程序难以理解。此外，有些任务会没有任何访问指针指向它，这种任务被称为**无名的**。例如，如果Robot_Arm指向一个任务，当它被重写为New_Arm时，前一个任务就变成无名的，因为没有别的指针指向它。 195

在某些情况下，使任务有一个独特的标识符（而不是一个名字）是很有用的。例如，服务器通常不关心客户任务的类型。确实，在下一章讨论通信和同步时，将会看到服务器并不直接知道它的客户是谁。然而，也有些偶然情况，服务器需要知道正与之通信的任务是不是前一个与之通信的任务。虽然核心Ada语言未提供这种设施，但系统编程附件（Systems Programming Annex）提供一种机制，使任务可以得到自己的惟一标识。这个标识可被传送给其他任务：

```
package Ada.Task_Identification is

  type Task_Id is private;
  Mull_Task_Id; constant Task_Id;

  function "=" (Left, Right : Task_Id) return Boolean;

  function Current_Task return Task_Id;
    -- 返回调用任务的独特Id

  -- 同此讨论无关的其他功能
private
  ...
end Ada.Task_Identification;
```

除这个包之外，该附件还支持两个属性：

1）对任务类型的任何前缀T，T'Identity返回类型Task_Id的值，它等于由T标记的任务的惟一标识符。

2）对于标记入口声明的前缀E，E.Caller返回类型Task_Id的值，它等于该入口调用正为之服务的那个任务的惟一标识符。此属性仅可在入口体或接受语句内部使用（见第8、9章）。

在使用任务标识符时必须要小心，因为不能保证在其后的某个时刻任务仍是活动的或仍在作用域中。

2. 任务终止

考虑过创建和表示之后，剩下的问题就是任务终止了。Ada提供了多种选择，一个任务将终止，如果：

1）它完成了其任务体的执行（或者正常地，或者作为未处理异常的结果）；

2）它执行了选择语句的一个“终止”选项（在9.4.2节解释），因而隐含着不再需要它；

196 3）它被中止。

如果未处理的异常导致了任务的终止，则该错误的影响被隔离在那个任务中。另一个任务（可通过一个属性）能够查询一个任务是否已经终止。

```
if T' Terminated then    -- 对于某个任务T
  -- 出错恢复动作
end if;
```

然而，查询任务不能区分其他任务是正常终止还是出错终止。

任务可以中止其名字在作用域内的任何其他任务。当一个任务被中止时，所有其依赖者都被中止。中止设施使得能够消除违背愿望的任务。然而，如果一个不良任务是无名的，因而不能被命名，也就无法中止它（除非它的主宰被中止）。所以，理想的办法是只让终止了的任务是无名的。

7.3.7 Java的并发执行

Java有一个预定义类java.lang.Thread，它提供用于线程（进程）创建的机制。然而，为避免所有线程成为Thread的子类，Java还提供了一个标准接口，叫做Runnable:

```
public interface Runnable{
  public abstract void run ();
}
```

所以，想要表达并发执行的类都必须实现这个接口并提供方法run。在程序7-1中给出的类Thread就是这样做的。

程序7-1 Java的Thread类

```
public class Thread extends Object implements Runnable
{
  // 构造器

  public Thread ();
  public Thread (String name);
  public Thread (Runnable target);
  public Thread (Runnable target, String name);

  public Thread (ThreadGroup group, String name);
```

```
//              抛出 SecurityException, IllegalThreadStateException
public Thread (ThreadGroup group, Runnable target);
//              抛出SecurityException, IllegalThreadStateException
public Thread (ThreadGroup group, Runnable target, String name);
//              抛出SecurityException, IllegalThreadStateException

public void run ();
public native synchronized void start ();
//             抛出IllegalThreadStateException

public static Thread currentThread ();

public final void join () throws InterruptedException;

public final native boolean isAlive ();

public void destroy ();
//         抛出SecurityException;

public final void stop ();
//         抛出SecurityException
// 已免除 (DEPRECATED)

public final synchronized void stop (Throwable o);
//         抛出SecurityException, NullPointerException
// 已免除

public final void setDaemon ();
//        抛出 SecurityException, IllegalThreadStateException

public final boolean isDaemon ();

// 将在本书中介绍的其他方法--完整的类描述见附录A

// 注意 RuntimeExceptions 不作为方法说明的一部分列出。
// 这里作为注解表示出来
}
```

197～198

Thread是Object的一个子类。它提供若干构造器方法和一个run方法。使用这些构造器，可以以两种方式创建线程（线程组在本节稍后予以考虑）。

第一种方式是声明一个类是Thread的子类，并覆盖run方法。然后就可以分配该子类的一个实例，并开始执行。例如，在机器人手臂的例子中，假定有如下的类和对象可用：

```
public class UserInterface
{
  public int newSetting (int Dim) { ... }
  ...
}

public class Arm
{
  public void move (int dim, int pos) { ... }
}

UserInterface UI = new UserInterface ();
Arm Robot = new Arm ();
```

在作用域中有了上面的类，下面将声明一个可用于表示三个控制器的类。

```
public class Control extends Thread
{
  private int dim;

  public Control (int Dimension) // 构造器
  {
    super ();
    dim = Dimension;
  }

  public void run ()
  {
    int position = 0;
    int setting;

    while (true)
    {
      Robot.move (dim, position);
      setting = UI.newSetting (dim);
      position = position + setting;
    }
  }
}
```

现在可创建三个控制器:

```
final int xPlane = 0;  // final 指出是常数
final int yPlane = 1;
final int zPlane = 2;

Control C1 = new Control (xPlane);
Control C2 = new Control (yPlane);
Control C3 = new Control (zPlane);
```

至此，线程已创建，已声明的所有变量都已初始化，并调用了类Control和Thread的构造
199 器方法（Java称这为新状态）。然而，线程直到方法Start被调用才开始执行⊖:

```
C1.start ();
C2.start ();
C3.start ();
```

注意，如果方法run是显式调用的，代码就顺序执行。

创建进程的第二种方式是声明一个实现Runnable接口的类。然后就可以分配该类的一个实例，并在线程对象的创建过程中被作为变元传递。记住，Java线程不是在其关联对象被创建时被自动创建的，而必须用Start方法显式创建并开始执行。

```
public class Control implements Runnable
{
  private int dim;
```

⊖ 方法Start有native和synchronized修饰符。修饰符native指出方法的体是用不同语言提供的。修饰符synchronized用于得到对对象的互斥访问。将在8.8节详细考虑Start方法。

```
    public Control (int Dimension) // 构造器
    {
      dim Dimension;
    }

    public void run ()
    {
      int position = 0;
      int setting;

      while (true)
      {
        Robot.move (dim, position);
        setting = UI.newSetting (dim);
        position = position + setting;
      }
    }
  }
```

现在可以创建三个控制器:

```
final int xplane = 0;
final int yplane = 1;
final int zPlane = 2;

Control C1 = new Control (xPlane); // 还没有创建进程
Control C2 = new Control (yplane);
Control C3 = new Control (zPlane);
```

接着创建相关线程并开始执行:

```
// 构造器传递了一个Runnable 接口，并创建了线程
Thread X = new Thread (C1);
Thread Y = new Thread (C2);
Thread Z = new Thread (C2);
X.start(); // 线程开始执行
Y.start();
Z.start();
```

200

像Ada一样，Java允许动态线程创建。不同的是，Java（利用构造器方法）允许任意数据被作为参数传递（当然，所有对象通过引用传递，这与Ada的访问变量类似）。

虽然Java允许线程层次并创建线程组，却没有主宰和监护者概念。这是因为Java靠垃圾回收器清理不再被访问的对象。主程序是个例外，当Java主程序的所有用户线程已经终止时，主程序也就终止（见后面的讨论）。

一个线程可以等待另一个线程（目标）的终止，办法是对目标的线程对象发出join方法调用。此外，isAlive方法可允许线程判断目标线程是否已经终止。

1. 线程标识

有两种标识线程的方法。如果线程的代码是Thread类的子类，那么，可以这样定义一个线程标识符:

```
Thread threadID;
```

Thread的任何子类均可被赋给这个对象（根据Java的参考语义）。注意，在线程代码是经

由Runnable接口传送给构造器的地方，线程标识符就是线程对象，而不是提供Runnable接口的对象。为了定义一个标识提供代码的对象的标识符，需要定义一个Runnable接口。

```
Runnable threadCodeID;
```

然而，一旦得到了对Runnable对象的引用，就不能再显式地在它上面做什么了。所有线程操作都由类Thread提供。

当前运行线程的身份可用方法currentThread找到。此方法有一个修饰符static，这意味着对Thread对象的所有实例只有一个方法。所以，总能用类Thread调用此方法。

2. 线程终止

Java线程可以以多种方式终止：

201 1）它完成了方法run的执行，或者是正常终止，或者是作为一个未处理异常的结果。

2）它的stop方法被调用，这时方法run即被停止，并在终止此线程前将线程类清理干净（释放它占有的锁并执行可能有的finally子句），该线程对象成了被回收的垃圾。如果有一个Throwable对象被作为参数传给了方法stop，那么该异常在目标线程中被抛出。这样就能更得体地退出方法run并清理自己[⊖]。方法stop本来就不安全，因为它释放对象上的锁，并能使这些对象处于不一致的状态。因此，这个方法被认为是过时的（Java叫做“已免除的”），因而不应再用它。

3）它的destroy方法被调用（或者由其他线程或者就是它自己）——destroy终止线程，使线程对象没有机会进行清理。这个方法在Java虚拟机中看来从未被实现。

虽然看起来（3）意味着任意线程能够破坏别的线程，程序可通过若干机制提供某些防护：

- 在创建新线程时覆盖这个方法，并在此方法内采取某些替代动作；
- 使用线程组限制访问；
- 把该线程包含在另一个类中，只允许某些操作被传递到该线程。

只有后两种方法可与方法stop一起使用，因为它们被标志成final，因而不能被覆盖。

Java线程有两种类型：**用户线程**和**守护**（daemon）线程。守护线程是那种提供通用服务并一般不会终止的线程。所以，当所有用户线程已经终止时，守护线程也会终止，且主程序也终止（守护线程提供与Ada的选择语句中的“or terminate”相同的功能——见9.4.2节）。在线程开始前必须调用方法setDaemon。

3. 线程组

线程组允许将线程集合组成组，并作为组而不是个体处理。它们还提供一种谁对什么进程做些什么的限制手段。Java中每个线程都是线程组的成员。同主程序关联的是一个默认的

202 组，所以除非另外指明，所有被创建的进程都属于这个组。线程组由程序7-2给出的类表示。

程序7-2 线程组的Java类摘录

```
public class ThreadGroup {

    // 构造一个新的线程组。这个新组的父亲是当前运行线程的线程组。
    public ThreadGroup (String name);
```

⊖ 事实上，在没有参数传给方法stop时，异常ThreadDeath被抛出。这是Error的一个子类，所以不能由程序捕获（见6.3.2节）。

```
    // 创建一个新的线程组。这个新组的父亲是指定的线程组。
    public ThreadGroup (ThreadGroup parent, String name);
    //  抛出SecurityException

    // 返回这个线程组的父亲。
    public final ThreadGroup get Parent ();

    // 检验这个线程组是不是一个守护线程组。
    public final boolean isDaemon ();

    // 检验这个线程组是否已经被破坏。
    public synchronized boolean isDestroyed ();

    // 改变这个线程组的守护状态。
    public final void setDaemon (boolean daemon);
    //                  抛出SecurityException

    // 检验这个线程组是线程组变元还是其一个祖先线程组。
    public final boolean parentOf (ThreadGroup g);

    // 确定当前运行线程是否有权修改该线程组。
    public final void checkAccess ();

    // 返回该线程组中活动线程的个数。
    public int activeCount ();

    // 返回该线程组中活动线程组的个数。
    public int activeGroupCount ();

    // 停止该线程组中的所有进程。
    public final void stop ();
    //                  抛出SecurityException

    // 破坏该线程组及其所有子组。
    public final void destroy ();
    //                      抛出 IllegalThreadStateException

  // 本书将要介绍的其他方法——完整类规格说明见附录A
}
```

203

当一个线程创建新的线程组时，它是在线程组里面做这件事的。所以，该新线程组是当前线程组的儿子，除非一个不同的线程组被作为参数传给构造器。用这两种构造器方法就能建立线程组的层次结构。

当线程被创建时，就能利用适当的Thread类构造将它们放进显式的线程组里。stop或destroy线程组的请求（如果安全管理员允许）被应用于组中的所有线程。

4. 同线程有关的异常

第6章介绍了Java的RuntimeException（运行时异常）。下列RuntimeException是与线程有关的。

IllegalThreadStateException在下列情况被抛出：

- 调用方法start，而线程已经启动了；
- 调用方法setDaemon，而线程已经启动了；
- 试图破坏一个非空线程组；
- 试图把一个线程放到线程组里（经由Thread构造器），而该线程组已经被破坏。

SecurityException在下列情况由安全管理器抛出⊖:

- 已经调用了Thread构造器，而它请求要把创建的线程放进一个它没有许可的线程组;
- 对线程调用方法stop或destroy，而调用者没有所请求操作的正当许可;
- 调用ThreadGroup构造器，并要求新创建线程组的父组是一个调用线程组没有许可的组;
- 调用一个线程组的方法stop或destroy，而调用者没有所请求操作的正当许可。

NullPointerException在下列情况被抛出:

- 向stop方法传递一个空指针;
- 向父组的ThreadGroup构造器传递一个空指针。

InterruptException在下列情况被抛出：若一个已经发出了join方法的线程由正被
204 中断的线程唤醒，而不是由目标线程的终止唤醒（见10.9节）。

5. 实时线程

Java编程语言缺乏为实时系统编程的许多设施，所以推出了实时Java。实时Java产生若干新的线程类，这些将在12.7.3节讨论。

7.3.8 Ada、Java和occam2的比较

上一节给出了occam2、Ada和Java进程模型的基本结构。然而，如果没有进程之间的通信，就不能介绍有意义的程序，这些将在以后弥补。即使如此，也可能说明occam2、Ada和Java进程模型之间的一些差别。在occam2中，两个过程调用明显地是顺序执行或是并发执行。

```
SEQ     -- 顺序形式
  proc1
  proc2

PAR     -- 并发形式
  proc1
  proc2
```

用Ada实现并发性，则需要引入两个任务:

```
begin     -- 顺序形式
  Proc1;
  Proc2;
end;


declare   -- 并发形式
  task One;
  task Two;
  task body One is
  begin
    proc1;
  end One;
  task body Two is
  begin
    Proc2;
```

⊖ Java安全管理器不在本书讨论，因为它与Java在开放分布式系统中的使用相关联的问题有关。

```
    end Two;
  begin
    null;
  end;
```

实际上，只用一个Ada任务也行，并让块语句本身发出另一个调用。 205

```
  declare
    task One;
    task body One is
    begin
      Proc1;
    end One;
  begin
    Proc2;
  end;
```

然而，这种形式就失去了算法的逻辑对称性，不值得推荐。

在Java中，有必要为每个过程创建一个实现Runnable接口的对象。

```
public class Proc1 implements Runnable
{
  public void run ()
  {
    // proc1的代码
  }
}

public class Proc2 implements Runnable
{
  public void run ()
  {
    // proc2的代码
  }
}

Proc1 P1 = new Proc1 ();
Proc2 P2 = new Proc2 ();

// 顺序形式
{
  P1.run;
  P2.run;
}

//并发形式
Thread T1 = new Thread (P1);
Thread T2 = new Thread (P2);

T1.start ();
T2.start ();
```

7.3.9 POSIX的并发执行

实时POSIX提供三种创建并发活动的机制。第一种是传统的Unix fork机制（及其相关的wait系统调用）。这引起创建和执行整个进程的一个副本。fork的细节在大多数操作系统教 206

科书中能够找到（例如，Silberschatz and Galvin（1944）），这里不予讨论（9.5节给出了用fork创建进程的例子）。第二个是Spawn系统调用，它等价于组合的fork和join的功能。

实时POSIX还允许每个进程包含执行的若干“线程”。这些线程都有权访问同一存储器位置并在单个地址空间中运行。因此，它们可同Ada任务、Java线程和occam2进程相比较。程序7-3说明了POSIX中线程创建的基本C接口。

程序7-3 C POSIX的线程接口

```
typedef ... pthread_t;  /* 细节未定义 */
typedef ... pthread_attr_t;
typedef ... size_t;

int pthread_attr_init (pthread_attr_t *attr);
  /* 初始化由attr指向的线程属性为它们的默认值 */

int pthread_attr_destroy (pthread_attr_t *attr);
  /* 破坏由attr指向的线程属性 */

int pthread_attr_setstacksize (pthread_attr_t *attr,
                               size_t stacksize);
  /* 设置线程属性的栈大小 */

int pthread_attr_getstacksize (const pthread_attr_t *attr,
                               size_t *stacksize);
  /* 获取线程属性的栈大小 */

int pthread_attr_setstackaddr (pthread_attr_t *attr,
                               void *stackaddr);
  /* 设置线程属性的栈地址 */

int pthread_attr_getstackaddr (const pthread_attr_t *attr,
                               void **stackaddr);
  /* 获取线程属性的栈地址 */

int pthread_attr_setdetachstate (pthread_attr_t *attr,
                                 int detachstate);
  /* 设置属性的分离状态*/

int pthread_attr_getdetachstate (const pthread_attr_t *attr,
                                 int *detachstate);
  /* 获取属性的分离状态*/

int pthread_create (pthread_t *thread, const pthread_attr_t *attr,
                    void * (*start_routine) (void *) , void *arg);
  /* 以给定属性创建一个新线程，并以给定变元调用给定的start_routine */

int pthread_join (pthread_t thread, void **value_ptr);
  /* 挂起调用线程直到指名的线程终止，任何返回值由value_ptr 指向的 */

int pthread_exit (void *value_ptr);
  /* 终止调用线程，并使指针value_ptr对任何汇合线程都可用 */

int pthread_detach (pthread_t thread);
  /* 当线程终止时，可以回收同指定线程关联的存储空间 */

pthread_t pthread_self (void);
```

```
  /* 返回调用线程的线程 id */

int pthread_equal (pthread__t tl, pthread_t t2);
  /* 比较两个线程的ids */

  /* 所有上述整型函数如果成功，返回0，否则返回错误号*/
```

POSIX的所有线程都有一些属性（例如，它们的栈大小）。为操纵这些属性，必须定义（类型pthread_attr_t的）属性对象，然后通过函数调用去设置属性和获取属性。一旦建立起正确的属性对象，就能创建一个线程并传送适宜的属性对象。每个被创建线程有一个关联的（类型pthread_t的）标识符，对同一进程的诸线程而言，这些标识符是各不相同的。一个线程可得到它自己的标识符（通过pthread_self）。

一个线程一旦通过pthread_create创建，就变成适合执行的。这里没有Ada激活状态或Java新状态的等价物。该函数有四个指针变元：一个线程标识符（经调用返回）、一个属性集、一个表示线程执行代码的函数和调用此函数时传给函数的参数集合。线程可通过从start_routine返回或调用pthread_exit或接受一个发送给它的信号（见10.6节）正常终止。它也可通过使用pthread_cancel中止。一个线程可通过pthread_join函数等待另一线程的终止。

最后，线程执行后的清除和存储回收活动被称为**分离**（detaching）。有两种方法做这件事：调用pthread_join函数并等待直到线程终止，或是设置线程的分离属性（或在创建时或动态地调用pthread_detach函数）。如果设置了分离属性，线程就不能汇合，而且在线程终止时其存储空间就自动回收。

由POSIX线程提供的接口在很多方面类似于编译器同其运行时支持系统对接所用的接口。确实，Ada的运行时支持系统可用POSIX线程很好地实现。提供高级语言抽象（像Ada、Java和occam2的那样）的好处是它排除了使用这种接口时出错的可能性。 207~208

为阐明POSIX线程创建设施的简单使用，下面显示机器人手臂程序：

```
#include <pthread.h>

pthread_attr_t attributes;
pthread_t xp, yp, zp;

typedef enum {xplane, yplane, zplane} dimension;

int new_setting (dimension D);
void move_arm (dimension D, int P);

void controller (dimension *dim)
{
  int position, setting;

  position = 0;
  while (1) {
    move_arm (*dim, position);
    setting = new_setting (*dim);
    position = position + setting;
  }
  /* 注意，没有对 pthread_exit的调用，进程不终止*/
}

#include <stdlib.h>
```

```
int main () {
  dimension X, Y, Z;
  void *result;

  X = xplane,
  Y = yplane;
  Z = zplane;
  if (pthread_attr_init (&attributes) ! = 0)
     /* 设置默认属性 */
     exit (EXIT_FAILURE);
  if (pthread_create (&xp, &attributes,
                      (void *) controller, &X) != 0)
     exit (EXIT_FAILURE);
  if (pthread_create (&yp, &attributes,
                      (void *) controller, &Y) != 0)
     exit (EXIT_FAILURE);
  if (pthread_create (&zp, &attributes,
                      (void *) controller, &Z) != 0)
     exit (EXIT_FAILURE);
  pthread_join (xp,  (void **) &result);
  /* 需要阻塞主程序 */

  exit (EXIT_FAILURE);
  /* 出错退出，程序不应当终止*/
}
```

209

上面的程序创建了一个线程属性对象，并带有一个默认属性集合。对pthread_create的调用创建controller线程的每个实例，并传递一个指出操作域的参数。如果从操作系统返回错误，程序通过调用exit例程终止。注意，如果终止带有正在执行线程的程序，那将导致那些线程被终止。所以，主程序必须发出pthread_join系统调用，即使这些线程没有终止。

在6.1.1节给出了处理POSIX返回的错误的风格。它假定每个调用都定义一个宏，它检验返回值和调用出错处理例程（若需要的话）。所以，能够将上述代码写成如下更易读的方式:

```
int main () {
  dimension X, Y, Z;
  void *result;

  X = xplane;
  Y = yplane;
  Z = zplane;

  PTHREAD_ATTR_INIT (&attributes);

  PTHREAD_CREATE (&xp, &attributes,  (void *) controller, &x);
  PTHREAD_CREATE (&yp, &attributes,  (void *) controller, &y);
  PTHREAD_CREATE (&zp, &attributes,  (void *) controller, &z);

  PTHREAD_JOIN (xp,  (void **) &result);

}
```

最后，应当注意，在概念上，将一个包含多个线程的进程（程序）分叉是不明确的，因为进程中的某些线程可能占有资源或正执行系统调用。POSIX标准规定子进程只有一个线程（见POSIX 1003.1c（IEEE，1995））。

7.4 一个简单的嵌入式系统

为了说明并发编程的优缺点，现在考虑一个简单嵌入式系统。图7-6画出该简单系统的轮廓：进程T从一组热偶读数据（经由模拟到数字的转换器，ADC），并对一个加热器作适当改变（经过一个数控开关）。进程P有类似功能，但是，它是对压力的（用一个数字到模拟的转换器，DAC）。T和P都必须同S通信，S通过屏幕向操作员显示测量结果。注意，P和T是主动的，S是个资源（它只响应来自T和P的请求）：它可被实现为一个保护资源或服务器，如果需要同用户进行更广泛交互的话。 210

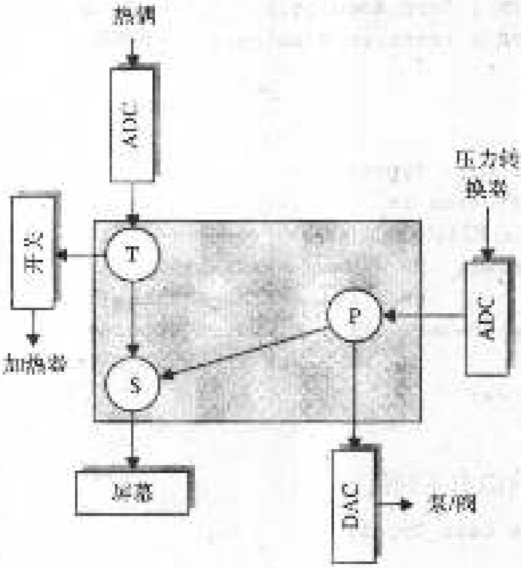


图7-6 一个简单嵌入式系统

此嵌入式系统的基本目标是使某个化学过程的温度和压力保持在确定的限值之内。这种类型的实际系统显然会更复杂些——例如，允许操作员改变限值。然而，即使对这个简单系统，实现也可采取如下三种形式之一：

1）用一个简单程序，忽略T、P和S的逻辑并发性，不需要操作系统的支持。

2）用一个顺序编程语言写T、P和S（或者是分离的程序或者是同一程序的不同过程），使用操作系统原语进行程序/进程的创建和交互。

3）使用一个单独的并发程序，保持T、P和S的逻辑结构。程序不需要操作系统的直接支持，但需要一个运行时支持系统。

为说明这些解决方案，考虑用Ada代码实现这个简单的嵌入式系统。为了简化控制软件的结构，假设已经实现了下面的被动包： 211

```
package Data_Types is
  -- 必需的类型定义
  type Temp_Reading is new Integer range 10..500;
  type Pressure_Reading is new Integer range 0..750;
  type Heater_Setting is (On, Off);
  type Pressure_Setting is new Integer range 0..9;
end Data_Types;
```

```
with Data_Types; use Data_Types;
package IO is
  -- 同环境进行数据交换的过程
  procedure Read (TR : out Temp_Reading); -- 来自 ADC
  procedure Read (PR : out Pressure_Reading);
    -- 注意，这是一个重载的例子；
    -- 定义了两个read，但它们有不同的参数类型；
    -- 下面的write也一样
  procedure Write (HS : Heater_Setting);   -- 到开关
  procedure Write (PS : Pressure_Setting);  -- 到DAC
  procedure Write (TR : Temp_Reading);      -- 到屏幕
  procedure Write (PR : Pressure_Reading);  --到屏幕
end IO;


with Data_Types; use Data_Types;
package Control_Procedures is
  -- 将读数转换成适宜的设置以供输出的过程
  procedure Temp_Convert (TR : Temp_Reading;
                          HS : out Heater_Setting);
  procedure Pressure_Convert (PR : Pressure_Reading;
                              PS : out Pressure_Setting);
end Control Procedures;
```

1. 顺序解决方案

简单的顺序控制程序可以有下列结构:

```
with Data_Types; use Data_Types;
with IO; use IO;
with Control_Procedures; use Control_Procedures;
procedure Controller is
  TR : Temp_Reading;
  PR : Pressure_Reading;
  HS : Heater_Setting;
  PS : Pressure_Setting;
begin
  loop
    Read (TR);     -- 来自 ADC的
    Temp_Convert (TR, HS); -- 读数被转换成设置
    Write (HS);    -- 到开关
    Write (TR);    -- 到屏幕
    Read (PR);     -- 像上面对压力一样
    Pressure_Convert (PR, PS);
    Write (PS);
    Write (PR);
  end loop; -- 无限循环，在嵌入式软件中是常见的
end Controller;
```

212

这段代码的直接不利因素是温度和压力必须以同样的速率读入，而这可能同需求不符合。使用计数器和适当的**if**语句会改善这种状况，但仍有必要将计算密集段（转换过程Temp_Convert和Pressure_Convert）划分成若干不同的动作，并交替执行这些动作，以满足

工作所需的平衡。即使这样做了，这个程序结构也还有一个严重的缺点：在等待读入一个温度数据时，就不能注意到压力（反之亦然）。此外，如果有一个系统失效，例如导致控制不能从温度的Read返回，那么，除了这个问题之外，压力的Read也就不能进行了。

对此顺序程序的改进是可以在IO包中加入两个布尔函数Ready_Temp和Ready_Pres，以指出数据的可读性。这样，控制程序成为：

```
with Data_Types; use Data_Types;
with IO; use IO;
with Control_Procedures; use Control_Procedures;
procedure Controller is
  TR : Temp_Reading;
  PR : Pressure_Reading;
  HS : Heater_Setting;
  PS : Pressure_Setting;
  Ready_Temp, Ready_Pres : Boolean;
begin
  loop
    ...
    if Ready_Temp then
      Read (TR);
      Temp_Convert (TR, HS);
      Write (HS);   -- 假设 write 是可靠的
      Write (TR);
    end if;
    if Ready_Pres then
      Read (PR);
      Pressure_Convert (PR, PS);
      Write (PS);
      Write (PR);
    end if;
  end loop;
end Controller;
```

这个解决方案更可靠，令人遗憾的是该程序在一个忙碌的循环中花了相当多的时间检查输入设备是否“就绪”。一般来说，“忙等待”的低效是不可接受的。它们束缚了处理器，使它难以对等待请求实施排队纪律。此外，依赖于忙等待的程序难以设计、理解或证明其正确性。 213

对顺序程序的主要批评是它对压力和温度周期是完全独立的子系统这一事实毫无认识。在并发编程环境中，这个问题可以通过将每个系统作为一个进程（或Ada中的任务）编码来改正。

2. *使用操作系统原语*

考虑一个类似POSIX的操作系统，它允许通过调用下述的Ada子程序创建和启动一个新进程/线程：

```
package Operating_System_Interface is
  type Thread_ID is private;
  type Thread is access procedure; -- 一个指针类型

  function Create_Thread (Code : Thread) return Thread_ID;
  -- 用于线程交互的其他子程序
```

```
private
  type Thread_ID is ...;
end Operating_System_Interface;
```

现在，该简单嵌入式系统可实现如下。首先，在包中放置两个控制器过程:

```
package Processes is
  procedure Pressure_Controller;
  procedure Temp_Controller;
end Processes;

with Data_Types; use Data_Types;
with IO; use IO;
with Control_Procedures; use Control_Procedures;
package body Processes is
  procedure Temp_Controller is
    TR : Temp_Reading;
    HS : Heater_Setting;
  begin
    loop
      Read (TR);
      Temp_Convert (TR, HS);
      Write (HS);
      Write (TR);
    end loop;
  end Temp_Controller;

  procedure Pressure_Controller is
    PR : Pressure_Reading;
    PS : Pressure_Setting;
  begin
    loop
      Read (PR);
      Pressure_Convert (PR, PS);
      Write (PS);
      Write (PR);
    end loop;
  end Pressure_Controller;
end Processes;
```

214

现在可给出Controller过程:

```
with Operating_System_Interface; use Operating_System_Interface;
with Processes; use Processes;

procedure Controller is
  Tc, Pc: Thread_Id;
begin
  -- 创建线程
  -- 'Access返回指向该过程的指针
  Tc := Create_Thread (Temp_Controller 'Access);
  Pc := Create_Thread (Pressure_Controller 'Access);
end Controller;
```

过程Temp_Controller和Pressure_Controller并发地执行，并且每个过程中包含一

个定义控制循环的无穷循环。当一个线程被挂起等待一个读入时，执行另一个线程；两个线程都被挂起时，就不执行忙碌循环。

虽然该解决方案比顺序方案要好，但是，缺乏表达并发性的语言支持意味着程序会变得难以理解和维护。对于上面给出的简单例子，所增加的复杂性还能承受。然而，对于有许多并发进程和进程间有潜在复杂交互的大系统，有一个过程性接口就使程序结构变得不清楚了。例如，哪个过程是真正的过程，哪个过程是打算用于并发活动的过程就不明显了。

3. *使用并发编程语言*

在并发编程语言中，并发活动可在代码中显式地标识出来: 215

```
with Data_Types; use Data_Types;
with IO; use IO;
with Control_Procedures; use Control_Procedures;
procedure Controller is
  task Temp_Controller;
  task Pressure_Controller;
  task body Temp_Controller is
    TR : Temp_Reading; HS : Heater_Setting;
  begin
    loop
      Read (TR);
      Temp_Convert (TR, HS);
      Write (HS);
      Write (TR);
    end loop;
  end Temp_Controller;

  task body Pressure_Controller is
    PR : Pressure_Reading; PS : Pressure_Setting;
  begin
    loop
      Read (PR);
      Pressure_Convert (PR, PS);
      Write (PS);
      Write (PR);
    end loop;
  end Pressure_Controller;

begin
  null;    -- Temp_Controller 和 Pressure_Controller
           -- 已经开始它们的执行
end Controller;
```

这样，应用逻辑就在代码中反映出来了，域的固有并行性通过程序中并发执行的任务表示出来了。

虽然这是一个改进，但是这个双任务解决方案还留下一个大问题。Temp_Controller和Pressure_Controller二者都向屏幕发送数据，但屏幕是一个资源，显然一次它只能被一个进程访问。在图7-6中，对屏幕的控制交给了第三个实体（S），它在程序中需要一个表示Screen_Controller。这个实体可以是一个服务器，或是一个保护对象（取决于Screen_

Controller所需行为的完备定义)。这就把问题从对被动资源的并发访问转化为任务间的通信，或者至少是任务同另外的并发性原语之间的通信。任务Temp_Controller和Pressure_Controller向Screen_Controller传送数据是必需的。此外，Screen_Controller必须确保一次只处理一个请求。这些要求和困难对于并发编程语言的设计来说是最重要的，并将在后续诸章研究。

小结

216 大多数实时系统的应用领域本来就是并行的。所以，在一个实时编程语言中是否包含进程概念，就使得语言的表达能力和易用性产生重大差别。这些因素又对降低软件开发成本，同时改善最终系统的可靠性做出重大贡献。

如果没有并发性，就必须把软件构造成一个单独的控制循环。这种循环的结构不能保持系统部件之间的逻辑差别。如果在代码中看不到进程概念的话，就特别难于给出面向进程的时间和可靠性需求。

然而，并发编程语言的使用不是没有代价的。尤其是必须使用运行时支持系统（或操作系统）以管理系统进程的执行。

进程的行为最好用状态描述，本章讨论了下列状态:

- 不存在
- 被创建
- 初始化
- 可执行
- 等待眷属的终止
- 等待子进程初始化
- 终止

在并发编程语言领域，采用的进程模型有很多种，可从六个方面分析它们。

1）结构——静态或动态进程模型
2）层次——只有顶层进程（平坦的）或多层（嵌套的）
3）初始化——有无参数传递
4）粒度——细粒度或粗粒度
5）终止——
    - 自然的
    - 自杀性的
    - 中止的
    - 未捕获出错
    - 永不终止
    - 不再需要时
6）表示——合作例程、分叉/汇合、cobegin，显式进程声明

occam2为嵌套静态进程采用了cobegin（PAR）表示，可把初始化数据给进程，然而它既
217 不支持中止，也不支持“不再需要时”终止。Ada和Java提供了动态模型，支持嵌套任务和各种终止选项。POSIX允许创建动态线程，线程具有平坦结构，线程必须显式终止或被杀掉。

相关阅读材料

Andrews, G. A. (1991) *Concurrent Programming Principles and Practice*. Redwood City, CA: Benjamin/Cummings.

Ben-Ari, M. (1990) *Principles of Concurrent and Distributed Programming*. New York: Prentice Hall.

Burns, A. and Wellings, A. J. (1995) *Concurrency in Ada*. Cambridge: Cambridge University Press.

Burns, A. (1998) *Programming in occam2*. Reading: Addison-Wesley.

Butenhof, D. R. (1997) *Programming With Posix Threads*. Reading, MA: Addison-Wesley.

Galletly, J. (1990) *Occam2*. London: Pitman.

Hyde, P. (1999) *Java Thread Programming*. Indianapolis, IN: Sams Publishing.

Lea, D. (1999) *Concurrent Programming in Java: Design Principles and Patterns*. Reading, MA: Addison-Wesley.

Oaks, A. and Wong, H. (1997) *Java Threads*. Sebastopol, CA: O'Reilly.

Nichols, B., Buttlar, D. and Farrell, J. (1996) *POSIX Threads Programming*. Sebastopol, CA: O'Reilly.

Silberschatz, A. and Galvin, P. A. (1998) *Operating System Concepts*. New York: John Wiley & Sons.

练习

7.1 一个特定操作系统有一系统调用（*RunConcurrently*），它取用一个未受约束数组。数组的每个元素是指向一无参过程的指针。该系统调用并发执行所有过程并在所有过程终止时返回。使用Ada任务设施实现该系统调用。假定操作系统和应用在同一地址空间内运行。

7.2 写出创建一个任务数组的Ada代码，每个任务有一个参数指出它在数组中的位置。

7.3 试用Ada实现*cobegin*。

7.4 不用Ada的任务间的通信，进程创建的*fork*和*join*方法能用Ada实现吗？

7.5 在下列过程中创建了多少POSIX进程？ 218

```
for (i=0; i<=10; i++) {
  fork ();
}
```

7.6 用Java、C和POSIX以及occam2重写7.4节阐述的简单嵌入式系统。

7.7 如果一个多线程进程执行了类似POSIX的fork系统调用，被创建的进程中应包含多少个线程？

7.8 用并发进程给出控制访问一个简单停车场的程序结构。假设该停车场有单一入口、单一出口和一个车位满标志。

7.9 在下列程序的帮助下解释Ada的任务终止规则和其异常传播模型之间的相互关系。要考虑变量C的初始值为2、1和0时的程序行为。

```
with Ada.Text_Io; use Ada.Text_Io;
procedure Main is
  task A;
```

```
    task body A is
      C : Positive := Some_Integer_Value;
      procedure P (D : Integer) is
        task T;
        A : Integer;
        task body T is
        begin
          delay 10.0;
          Put ("T Finished"); New_Line;
        end T;
      begin
        Put ("P Started"); New_Line;
        A := 42/D;
        Put ("P Finished"); New_Line;
      end P;

    begin
      Put ("A Started"); New_Line;
      P (C-1);
      Put ("A Finished"); New_Line;
    end A;

  begin
    Put ("Main Procedure Started");. New_Line;
  exception
    when others =>
      Put ("Main Procedure Failed"); New_Line;
  end Main;
```

219

7.10 对下列Ada程序中的每个任务指出其父亲和监护者（主宰）以及它可能有的儿子和眷属。还指出Main和Hierarchy过程的眷属。

```
procedure Main is
  procedure Hierarchy is
    task A;
    task type B;

    type Pb is access B;
    Pointerb : Pb;

    task body A is
      task C;
      task D;
      task body C is
      begin
        -- 语句序列, 包含
        Pointerb := new B;
      end C;
      task body D is
        Another_Pointerb : Pb;
      begin
        -- 语句序列, 包含
        Another_Pointerb := new B;
```

```
      end D;
    begin
      -- 语句序列
    end A;

    task body B is
    begin
      -- 语句序列
    end B;

  begin
    -- 语句序列
  end Hierarchy;

begin
  -- 语句序列
end Main;
```

7.11 能够用图7-2表示（a）POSIX的pthreads，（b）occam2进程和（3）Java线程的状态迁移图的哪些方面？

7.12 对于下列代码:

```
public class Calculate implements Runnable

{

  public void run ()
  {
  /*long 计算 */
  }
}

Calculate MyCalculation = new Calculate ();
```

试问:

```
MyCalculation.run ();
```

和

```
new Thread (MyCalculation) .start ();
```

之间有什么不同？ 220

7.13 阐明如何防止一个Java线程被一个任意线程破坏。 221

第8章 基于共享变量的同步和通信

并发编程的主要难点来自进程间的交互。不像我们在第7章末尾看到的一些简单例子，实际的进程是很少和其他进程无关的。并发程序的正确行为很大程度上取决于进程间的同步和通信。从最广泛的意义来说，同步就是对不同进程的动作交叉执行时的约束条件的满足（例如只在一个进程发生一个规定动作之后，另一个进程才能发生一个特定动作）。从狭义来说，这个术语也用来指使得两个进程同时进入预定义状态。通信是指从一个进程到另一个进程的信息传递。因为某些通信形式需要同步，且同步也可以被看成无内容的通信，所以这两个概念是相互关联的。

通常，进程间通信实际上是基于**共享变量**的或基于**消息传递**的。共享变量是可以被多个进程访问的对象，因此引用这些变量的进程可以在合适的时候进行通信。消息传递涉及两个进程之间通过消息形式的显式数据交换，消息通过某个代理从一个进程传递到另一个进程。值得注意的是，共享变量和消息传递的选择是语言或操作系统设计者的事情，它并不暗示应该使用哪种具体的实现方法。在进程之间有共享存储器的情况下，共享变量很容易支持，但即使硬件拥有一个通信介质，依然可以使用共享变量。类似地，消息传递原语可以通过共享存储器或是物理消息传递网络支持。此外，我们可以用以上的任何一种方式编写应用且获得同样的功能（Lauer and Needham，1978）。基于消息的同步和通信将在第9章讨论。本章将主要讲述基于共享变量的通信和同步原语，尤其是要讨论忙等待、信号量、条件临界区、管程、保护类型和同步方法。 223

8.1 互斥和条件同步

虽然共享变量看来是一种在进程间很直接的信息传递方式，但由于多个进程更新问题，对共享变量的不受限制使用是不可靠的和不安全的。考虑下面的情况：两个进程用下面的赋值语句同时更新一个共享变量X:

```
X := X + 1
```

在大多数硬件上，这个语句并不被作为**不可分的**（原子的）操作执行，而是用下面的三条指令实现：

1）把X的值载入到某个寄存器（或到栈的顶端）

2）对寄存器的值增加1

3）将寄存器的值存回X

由于这三个操作不是不可分的，两个同时更新这个变量的进程如果交叉执行的话，将产生一个错误的结果。例如，如果X的初值为5，那么两个进程分别将5装入各自的寄存器并增加1，然后存储结果为6。

临界段（critical section）是一个必须不可分地执行的语句序列。保护临界段所需的同步称为**互斥**（mutual exclusion）。虽然原子性对赋值操作不成立，在存储器级别却是假定成立的。因此，假若一个进程执行`X:=5`，同时另一个执行`X:=6`，那么结果将是5或6（不会有其他的结果）。如果不是这样的话，那么对并发程序的推理或实现较高级别的原子性（例如互斥同步），将变得很困难。然而，很明显，如果两个进程是在更新一个结构化对象，那么原子性将只适用于单个字的元素级别上。

互斥问题是由Dijkstra（1965）首次描述的。它是大部分并发进程同步的核心问题，具有很大的理论和实际意义。互斥并不是惟一重要的同步方法，实际上，如果两个进程之间没有共享变量，那么就不需要互斥。条件同步是另一个重要需求，在下述情况下就需要它：一个进程希望执行某个操作，而这个操作只有在另一个进程已经采取了某行动或处于某已定义状态之后进行才是合理的和安全的。

一个条件同步的例子是缓冲区的使用。两个交换数据的进程不直接进行通信，而是通过
224 缓冲区进行要更好些。这对消除进程之间的耦合有好处，并且它允许两个进程的工作速度可以有小的波动。例如，一个输入进程可能接受爆发式到来的数据，这些数据必须经过缓冲，再提供给合适的用户进程。在并发编程中使用缓冲区连接两个进程是普遍的，这被称为**生产者-消费者**系统。

在使用一个有限（有界）的缓冲区时，两个条件同步是必需的。第一，当缓冲区已满时，生产者进程不能试图向缓冲区存入数据。第二，当缓冲区为空时，消费者进程不能从缓冲区取出数据对象。而且，如果允许存入和取出操作同时进行的话，就必须确保互斥以使两个生产者不会搞乱缓冲区的“下一个空槽”指针。

任何一种同步形式的实现都隐含着有时必须阻止进程的执行直到适合它执行的时刻。在8.2节，我们将用**忙等待**（busy_wait）循环和**标志**（flag）来编程（用一个有显式进程声明的类Pascal语言）实现互斥和条件同步。从以上的分析来看，为了使需要同步的算法的编码容易一些，我们需要更多的原语，这应该是很清楚的。

8.2 忙等待

实现同步的一个方法是让进程设置并检查作为标志的共享变量。这个方法很适合条件同步，但对互斥却没有简单的方法。为了将一个条件作为信号发出去，一个进程设置一个标志的值；为了等待这个条件，另一个进程检查这个标志，只在读到合适的值时才继续前进：

```
process P1;    (* 等待进程*)
  ...
  while flag = down do
    null
  end;
  ...
```

```
end P1;

process P2;  (* 发信号进程*)
  ...
  flag := up;
  ...
end P2;
```

如果条件还没有被置位（这就是说，flag仍然是down），那么P1除了循环检查标志外没有别的选择。这就是**忙等待**，也被称作**自旋转**（spinning）（标志变量称为**循环锁**，spin lock）。

忙等待算法一般是效率低下的，因为当它们不能做任何有用工作时，它们包含的进程耗尽了处理周期。即使在多处理器系统上，它们也会在存储总线或网络上（如果是分布式）引起过量的流量问题。此外，如果有多个进程在等待一个条件（也就是说，检查标志的值），那么也不可能简单地实施排队规则。 225

由于互斥所需的算法要复杂得多，它产生了更多的困难。考虑有互斥临界段的两个进程（又是P1和P2）。为了保护对这些临界段的访问，我们可以假设每个进程在进入临界段前都会执行一个进入协议，同时在退出后执行一个退出协议。因此每个进程可以被看成如下的形式:

```
process P;
  loop
    进入协议
      临界段
    退出协议
    非临界段
  end
end P;
```

在对互斥问题给出一个合适的解决方案之前，先讨论三个不合适的方法。首先，考虑有两个标志的解决方案，它是忙等待条件同步算法的一种合理扩展:

```
process P1;
  loop
    flag1 := up;          (*宣布进入意图 *)
    while flag2 = up do
      null              (* 如果有其他进程在其临界段*)
    end;                (* 则忙等待 *)
    <临界段>
    flag1 := down;        (* 退出协议 *)
    <非临界段>
  end
end P1;

process P2;
  loop
    flag2 := up;
    while flag1 = up do
      null
    end;
    <临界段>
    flag2 := down;
    <非临界段>
```

```
    end
  end P2;
```

两个进程都宣称想要进入临界段，并检查是否有其他进程位于其临界段内。令人遗憾的是这个“解决方法”有一个并非不重要的问题。考虑如下的过程交叉执行情况:

P1 设置其标志 (flag1 = up)
P2 设置其标志 (flag2 = up)
P2 检查flag1 (flag1 = up，因此P2进入循环)
226 P2 进入忙等待
P1 检查flag2 (flag2 = up，因此P1进入循环)
P1 进入忙等待

其结果是两个进程都将保持它们的忙等待状态。由于对方不能退出循环，两者都不能退出循环。这种现象被称作**活锁** (livelock)，是一种严重的错误状态。

以上方法的困难，来自每个进程在检查它这样做是否可接受之前宣称要进入临界段。另一个方法是颠倒这两个操作的顺序:

```
process P1;
  loop
    while flag2 = up do
      null              (* 如果有其他进程在其临界段*)
    end;                (* 则忙等待 *)
    flag1 := up;        (*宣布进入意图*)
    <临界段>
    flag1 := down;      (* 退出协议 *)
    <非临界段>
  end
end P1;

process P2;
  loop
    while flag1 = up do
      null
    end;
    flag2 := up;
    <临界段>
    flag2 := down;
    <非临界段>
  end
end P2;
```

现在我们可以产生一个不互斥的交叉执行情况:

P1和P2处于非临界段 (flag1 = flag2 = down)
P1检查flag2 (down)
P2检查flag1 (down)
P2设置其标志 (flag2 = up)
P2进入临界段
P1设置其标志 (flag1 = up)
P1进入临界段
(P1和P2都处于临界段)

以上讨论的两个结构的主要问题在于设置自己的标志和检查另一个进程不能被作为一个不可

分的操作进行。因此，一个可能正确的方法是只用一个标志来指示下一个进入临界段的进程。由于这个标志决定了轮到谁进入临界段，因此我们用turn来命名。 227

```
process P1;
  loop
    while turn = 2 do
      null
    end
    <临界段>
    turn := 2
    <非临界段>
  end
end P1;

process P2;
  loop
    while turn = 1 do
      null
    end
    <临界段>
    turn :=1;
    <非临界段>
  end
end P2;
```

用这种结构，变量turn的值只可能是1或2。如果是1的话，那么P1肯定不会被无限期延迟，且P2不能进入它的临界段。而且，当P1处于它的临界段时turn不会变成2，因为turn可以被赋值为2的惟一地方是在P1进程中，且只有在P1处于退出协议时。对turn这个对称变元取值为2，就保证了互斥，而且，如果两个进程都循环执行，那么也不可能发生活锁。

令人遗憾的是后者有些问题。假若P1在其非临界段发生错误，那么turn的值为1并将保持下去（这就是说P2将不能进入临界段，即使P1不再执行）。甚至在正常执行时，使用单个turn变量要求进程以相同的速率循环执行。例如，对P1来说进入临界段三次而P2只进入一次这种情况是不可能的。这种限制对自治进程来说是不可接受的。

最后，我们给出一个算法，该算法不会引起前面例子的闭合耦合问题而且提供互斥，不会产生活锁。Peterson（1981）首先提出这个算法。另一个著名的算法，即Dekker算法由Ben-Ari（1982）提出。Peterson（和Dekker）的方法有两个标志（flag1和flag2）和一个turn变量，标志由“拥有”它们的进程操纵，turn变量用来察看是否存在进入临界段的竞争：

```
process P1;
  loop
    flag1 := up;     (* 宣布进入意图 *)
    turn := 2;       (* 给其他进程优先级 *)
    while flag2 = up and turn = 2 do
      null
    end;
    <临界段>
    flag1 := down;
    <非临界段>
  end
```

228

```
end P1;

process P2;
  loop
    flag2:= up;     (*宣布进入意图*)
    turn:= 1;       (*给其他进程优先级*)
    while flag1 = up and turn = 1 do
      null
    end;
    <临界段>
    flag2:= down;
    <非临界段>
  end
end P2;
```

如果只有一个进程想进入临界段，那么另一个进程标志将设为down，可以立即进入。然而，如果两个标志都为up，那么变量turn的值就很重要了。我们假设turn有初值1，那么可能存在四种交替执行的可能，这取决于每个进程给turn赋值和在while语句中检查它的值的顺序：

第一种可能情况 ------ 先P1 后P2
P1设turn为2
P1检查turn，并进入忙循环
P2设turn为1 (turn现在将保持1)
P2检查turn，并进入忙循环
P1再次检查turn，并进入临界段

第二种可能情况 ------ 先P2 后P1
P2设turn为1
P2检查turn，并进入忙循环
P1设turn为2 (turn现在将保持2)
P1检查turn，并进入忙循环
P2再次检查turn，并进入临界段

第三种可能情况 ------ P1和P2交叉
P1设turn为2
P2设turn为1 (turn现在将保持1)
P2进入忙循环
P1进入临界段

第四种可能情况 ------ P2 和P1交叉
P2设turn为1
P1设turn为2 (turn现在将保持2)
P1进入忙循环
229 P2进入临界段

所有四种情况都导致一个进程进入临界段，另一个进程处于忙循环。

一般来说，虽然单个交叉执行只能说明系统不满足其规格说明，但要容易地给出导致符合规格说明的所有可能交叉是不可能的。通常，为了展现这种符合性，需要证明的方法（包括模型检查）。

有趣的是，如果存在访问（对临界段的）竞争，以上的算法是相当公平的，也就是说P1

成功进入临界段后（经由第一或第三种可能），P2也肯定能在下一次进入临界段。当P1离开它的临界段时，它设flag1为down。这将可能让P2进入它的临界段，但即使P2没有这样做（因为实际上P2在该时刻并没有执行）而P1继续前进，进入和离开它的非临界段，设置flag1为up，设置turn为2，然后进入忙循环。P1将等待直到P2进入和离开临界段并重置flag2（作为它的退出协议）。

用可靠性的话说，一个进程在非临界段的失效将不会影响其他进程，但在协议和临界段失效时就不是这样了。进程过早地终止有可能引起程序其余部分的活锁困难。

以上详细的讨论说明，只使用共享变量而没有附加的原语（除了顺序语言里的原语之外）实现进程间的同步是很困难的。这些难点总结如下：

- 使用忙循环的协议很难设计、理解和证明其正确性（读者可以思考一下把Peterson算法推广到用于n个进程的情况）。
- 对程序进行的测试可能会检查不到那些罕见的违反互斥或引起活锁的交替执行路径。
- 忙等待循环效率低下。
- 不可靠（恶意）的任务可能误用共享变量，这将导致整个系统瘫痪。

没有哪个并发程序设计语言完全依赖忙等待和共享变量，现在已经有一些其他的方法和原语。对于共享变量的系统，信号量和管程是最重要的构造，我们将在8.4和8.6节描述。

8.3 挂起和恢复

忙等待循环的一个问题是它浪费了宝贵的处理器资源。一个替代方案是当调用进程等待的条件不能满足时就挂起该进程（也就是将其从可运行进程队列移出去）。例如，考虑一下使用一个flag的简单条件同步。一个进程置位该标志，另一个进程等待直到这个标志被置位，然后清除它。可以这样使用一个简单的挂起和恢复机制实现之： 230

```
process P1;  (* 等待进程*)
  ...
  if flag = down do
    suspend;
  end;
  flag := down;
  ...
end P1;
process P2;  (* 发信号进程 *)
  ...
  flag := up;
  resume P1; (*如果 P1 未挂起, 则没有任何效果 *)
  ...
end P2;
```

Java的类Thread的一个早期版本提供了支持该机制的方法：

```
public final void suspend ();
                  //抛出 SecurityException;
public final void resume ();
                  //抛出SecurityException;
```

因此以上的例子可以用Java实现如下：

```
boolean flag;
final boolean up = true;
final boolean down = false;

class FirstThread extends Thread {

  public void run () {
    ...
    if (flag == down) {
      suspend ();
    };
    flag = down;
    ...
  }
};

class SecondThread extends Thread { // T2

  FirstThread T1;

  public SecondThread (FirstThread T) {
    super ();
    T1 = T;
  }

  public void run () {
    ...
    flag = up;
    T1.resume ();
    ...
  }
}
```

231

令人遗憾的是这个方法会引起**竞争条件**（race condition）问题。线程T1可能测试flag，然后底层运行时支持系统（或操作系统）可能决定抢占它而运行T2，T2置位flag并恢复T1。当然由于T1没有挂起，因此resume不起作用。接着，当T1再次运行时，它认为flag已经是down，因此它挂起自己。

正如我们前一节给出的例子一样，出现这个问题的原因是因为flag是一个正在被测试的共享资源，采取的行动依赖于它的状态（进程挂起本身）。测试和挂起不是原子操作，因此可能出现来自其他进程的干扰。由于这个原因，Java的最新版本已经取消了这个方法。

目前已经有几个解决竞争条件问题方法，所有方法都提供了一种**两阶段挂起**操作的形式。本质上P1必须宣布它计划在将来的某个时刻挂起，任何发现P1没有被挂起的恢复操作都有一个延迟效果。当P1挂起时，它立即被恢复，也就是说，挂起操作本身没有效果。

虽然挂起和恢复是低级工具，在使用中又是易出错的，但它是一个可以用来构造高级同步原语的有效机制。由于这个原因，Ada提供了这种机制的一个安全版本，并把它作为实时附件的一部分。它围绕着对象**挂起**（suspension）这个概念，它的值可以是True或False。程序8-1给出了这个包的规格说明。

这个包定义的所有四个子程序相对于其他子程序都是原子操作。在从过程Suspend_Until_True返回时，引用的挂起对象被复位为False。

程序8-1 同步任务控制

```
package Ada.Synchronous_Task_Control is
  type Suspension_Object is limited private;
  procedure Set_True (S : in out Suspension_Object);
  procedure Set_False (S : in out Suspension_Object);
  function Current_State (S : Suspension_Object) return Boolean;
  procedure Suspend_Until_True (S: in out Suspension_Object);
    -- 如果有一个以上的任务试图立即挂起S，引发Program_Error
private
  -- 语言没有规定
end Ada.Synchronous_Task_Control;
```

232

因此，在本节前面所提到的简单条件同步问题可以很容易地解决了。

```
with Ada.Synchronous_Task_Control;
use  Ada.Synchronous_Task_Control;

...
Flag : Suspension_Object;
...
task body P1 is
begin
  ...
  Suspend_Until_True (Flag);
  ...
end P1;

task body P2 is
begin
  ...
  Set_True (Flag);
  ...
end P1;
```

挂起对象的行为在很多方面和二元信号量相同，这将在8.4.4节讨论。

虽然suspend和resume是很有用的低级原语，但没有哪种操作系统或语言单独依靠这些机制来实现互斥和条件同步。如果使用的话，它们就明显为第7章介绍的状态转换图引进了一种新的状态。所以，进程更一般的状态图可以扩展成如图8-1所示的样子。

8.4 信号量

信号量是一种互斥和条件同步编程的简单机制。它最初是由Dijkstra（1968a）设计的，有以下的两个好处:

1）简化同步协议。

2）消除了忙等待循环。

信号量是一个非负整型变量，除了初始化外，只有两个过程能对它发生作用。这两个过程被Dijkstra称为P和V，但在本书中称为wait和signal。wait和signal的语义如下:

1）wait（S）如果信号量S的值大于0，则将其值减1；否则延迟进程直到S大于0（然后将其值减1）。

233

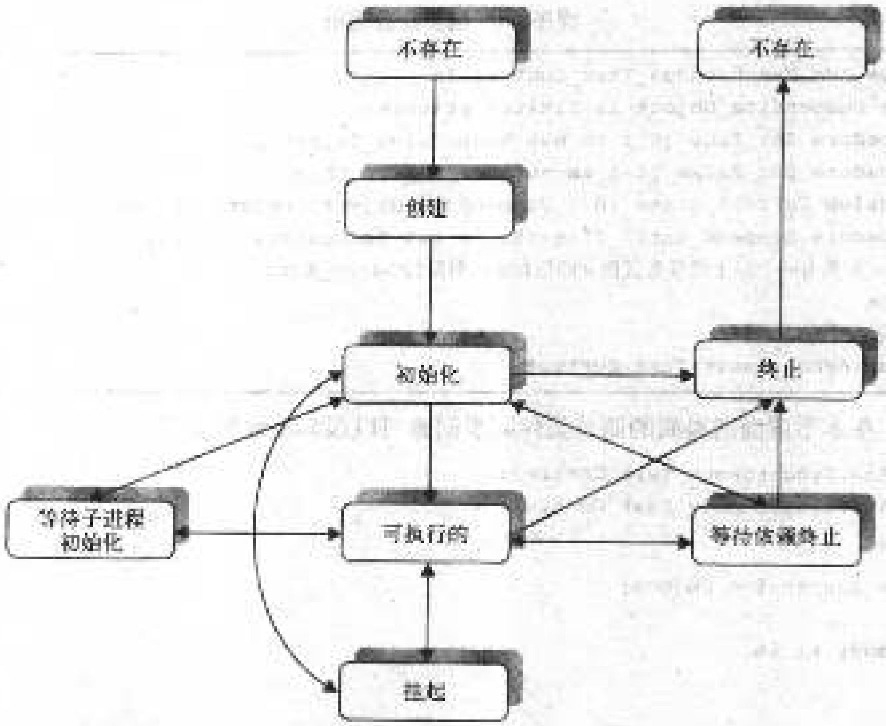


图8-1 进程的状态图

2) signal (S) 将信号量S的值增1。

一般的信号量被称为计数信号量，因为它们的操作是在整数上增或减1。wait和signal另外的重要属性是它们的操作是原子的（不可分）。两个进程都在同一信号量上执行wait操作，但不会互相干扰。而且，在执行信号量操作期间，进程不可能失效。

条件同步和互斥可以很容易地用信号量编程实现。首先，考察一下条件同步：

```
(* 条件同步 *)
var consyn : semaphore; (* 初值为 0 *)
process P1; (* 等待进程 *)
  ...
  wait (consyn);
  ...
end P1;

process P2; (* 发信号进程 *)
  ...
  signal (consyn);
  ...
end P2;
```

234

当P1在初值为0的信号量上执行wait操作时，它将被延迟，直到P2执行signal操作。这将把consyn设置为1，因此wait操作得以继续，P1将继续执行，同时consyn将减少到0。值得注意的是，如果P2先执行signal，信号量被设为1，那么P1将不会被wait的动作延迟。

互斥也是类似地直接明了：

```
(* 互斥 *)
```

```
var mutex : semaphore; (* 初始值为1 *)
process P1;
  loop
    wait (mutex);
      <临界段>
    signal (mutex);
    <非临界段>
  end
end P1;


process P2;
  loop
    wait (mutex);
      <临界段>
    signal (mutex);
    <非临界段>
  end
end P2;
```

如果P1和P2处于竞争中，那么它们将同时执行各自的wait语句。然而，由于wait是原子的，因此，在一个进程完成这个语句后，另一个语句才能开始。其中一个进程以mutex=1执行wait (mutex)，这使得该进程进入其临界段并设置mutex的值为0。接着另一个进程以mutex=0执行wait (mutex)，并被延迟。一旦第一个进程退出了其临界段，它将执行signal (mutex)。这引起信号量mutex的值变成1，并允许第二个进程进入其临界段（再次设置mutex为0）。

用一对wait/signal将一段代码包围起来，信号量的初值将限制代码段可以被并发执行的最大数目。如果其初值为0，那么将决不会有进程进入；如果为1，只允许一个进程进入（这就是互斥的情况）；对于大于1的值，允许这段代码的并发执行数就是这个值。 235

8.4.1 挂起进程

从wait的定义中，我们可以清楚地看到，如果信号量是0，那么调用进程将被延迟。延迟的一个实现方法（忙等待）前面已经介绍过了，也批评过了。在8.3节我们还介绍了一个更有效的机制——进程的挂起。实际上，所有的同步原语都以某种挂起的形式来处理延迟，从可运行进程队列中移出这个进程。

当一个进程在一个0值信号量上执行wait操作时，RTSS（运行时支持系统）被调用，使得这个进程从处理器上移出，并放到挂起进程队列（即挂在那个特定信号量上的进程队列）上。然后RTSS必须选择另一个进程运行。最终，如果程序正确的话，另一个进程将在信号量上执行signal。结果，RTSS选择一个在等待这个信号而被挂起的进程，使其再度可执行。

考虑到以上的原因，可以给出wait和signal的一个稍微有些不同定义如下，这个定义比较接近于实际的实现：

```
wait (S) :-
if S > 0 then
  S:= S-1
else
  number_suspended:= number_suspended + 1
```

```
  挂起调用进程

signal (S) :-
if number_suspended > 0 then
  number_suspended:= number_suspended - 1
  使一个挂起进程再次成为可执行的
else
  S:= S+1
```

这个定义避免了信号量增加后又立即减少。

注意：以上的算法并没有指明进程从挂起状态恢复的顺序。虽然对于一个真正的并发语言来说，程序员应该可以假设一个不确定的顺序（见9.4.3节），但它们通常是以FIFO（先进先出）顺序释放。然而，对于实时编程语言来说，进程的优先级在选择时起重要作用（见第13章）。

8.4.2 实现

虽然以上的算法需要队列机制的支持，但实现信号量还是很简单直接的。真正的难点在于要求wait和signal操作执行的不可分性。不可分性意味着一旦进程开始执行这些过程之
236 一，那么它将继续执行直到操作执行完毕。在RTSS的帮助下，这是很容易实现的，当进程执行wait或signal时，调度程序不会将该进程换出，因为它们是**不可抢占**的操作。

令人遗憾的是，RTSS并不总是完全控制着调度事件。虽然所有内部行为都处于其控制下，但异步发生的外部事件却可能打破信号量操作的原子性。为了防止这种情况，RTSS在不可分的语句序列执行期间禁止中断。通过这种方法就不会有外部事件的干扰。

禁止中断对于单处理器系统是合适的，但不适用于多处理器。在一个共享存储器的系统里，两个并行进程可以执行wait或signal（在同一个信号量上），但RTSS却不能阻止这种情况。在这些情况下，需要一种“锁”机制保护对操作的访问。目前使用两种机制。

在某些处理器上，提供了“测试并设置”指令。它允许进程以下面的方式访问一个二进制位。

1）如果该位为0，则设置为1并返回0。

2）如果该位为1，则返回1。

这些操作本身是不可分的。例如，两个并行进程都想执行wait，那么它们将测试并设置同一个锁位（初值为0）。一个进程将成功执行并设置该位为1；另一个进程将返回1，并因此会进入循环而不断重新测试这个锁。当第一个进程完成wait操作，它给锁赋值0（也就是对信号量解锁），这样另一个进程就继续执行wait操作。

如果没有“测试并设置”指令可用，那么可以用一条交换指令取得类似的效果。同样的，这个锁与一个初值为0的位相关联。一个希望在信号量上执行操作的进程把1交换进该锁位。如果它从该锁返回0，那么它可以继续前进；如果返回1，那么肯定有另一个进程在信号量上活动，必须重新测试该锁。

正如在8.1节中指出的，像信号量这样的软件原语不可能像变戏法一样解决有关互斥的问题。为了建立更高级的结构，需要让存储位置表现出互斥的本质。类似地，虽然由于使用信号量，忙等待已经从程序员的视野里消失了，但使用忙等待来实现wait和signal操作也是必要的。*然而应当注意到，忙等待的后一种使用仅仅是短时的（花费在执行wait或signal操作上的时间短），但是它对程序临界段的延迟访问却包含了大量循环花费的时间。*

8.4.3 活性

在8.2节，我们说明了活锁这种出错状态。令人遗憾的（也是不可避免的）是，同步原语的使用也产生了一些其他的出错状态。**死锁**是其中最严重的一个，它使一个进程集合处于一种其中任何一个进程都不能继续执行的状态。这同活锁类似，不同的是进程是被挂起的。为 237
了说明这种情况，我们来考察一下希望访问两个非并发资源（一次只能被一个进程访问的资源）的两个进程P1和P2，这两个资源由信号量S1和S2进行保护。如果两个进程都以同样的顺序访问资源，那就不会有问题出现：

```
  P1                                P2
wait (S1);                        wait (S1);
  wait (S2);                        wait (S2);
  .                                 .
  .                                 .
  .                                 .
  signal (S2);                      signal (S2);
signal (S1);                      signal (S1);
```

第一个进程首先在S1上执行wait成功，然后在S2上执行wait也成功，随后在两个信号量执行signal，并允许另一个进程进入。但是，如果其中的一个进程以相反的顺序使用资源就会发生问题，例如：

```
  P1                                P2
wait (S1);                        wait (S2);
  wait (S2);                        wait (S1);
  .                                 .
  .                                 .
  .                                 .
  signal (S2);                      signal (S1);
signal (S1);                      signal (S2);
```

在这种情况下，P1和P2穿插地成功执行S1和S2上的wait，但在等待另一个信号量时，两个进程都不可避免地被挂起，因为信号量为0。

一旦进程的一个子集变成了死锁，所有其他的进程最终会变成这个死锁集的一部分，这是一个相互依赖的并发程序的天性。

除了最明显的死锁外，软件测试很少能消除死锁问题，死锁虽然不经常发生，但造成的后果很严重。这种错误不是信号量所特有的，所有的并发编程语言都可能存在这个问题。设计一种可以禁止死锁的语言是理想的，但却是不可能达到的目标。与死锁避免、检测和恢复有关的问题将在第11和13章讨论。

无限制延期（有时也叫**停工或饿死**）是一种严重性小一点的出错状态，在这种情况下，一个进程希望通过临界段访问某个资源，但由于总有其他进程在它之前获得对资源的访问而导致该进程一直无法获得访问。在信号量系统中，一个进程可能会被无限期挂起（在信号量队列上），这是当信号来到时由RTSS从这个队列选取进程的策略造成的。即使延迟实际上并不是无限的，但却是我们无法确定的，这会在实时系统中引起错误。

如果一个进程不存在活锁、死锁和无限制延期问题，那么我们说这个进程是有**活性**（liveness）的。非正式地说，活性这个性质暗示着如果进程希望执行某种操作，那么它最终 238
必将可以做到。特别是，如果一个进程要求访问临界段，那么它将在有限的时间内获得访问。

8.4.4 二元信号量和定量信号量

一般信号量的定义是非负的整数，暗示着其实际的值可以增加到任何一个所支持的正整数。但是到目前为止本章所给的例子中（对于条件同步和互斥），只使用了0和1。一种简单形式的信号量被称为**二元信号量**，它被实现成只使用这两个值，也就是说，一个值为1的信号量的发信号操作是没有任何效果的——信号量保持其值为1。如果要求一般形式的信号量，那么可以从两个二位信号量和一个整数构造一般的信号量。

信号量正规定义的另一个变种是**定量信号量**。在这种结构中，`wait`减少的数量（`signal`增加的数量）不固定为1，而是作为过程的一个参数:

```
wait (S, i) :-    if S >= i then
                   S := S - i
                 else
                   delay
                   S := S - i
signal (S, i) :-   S := S + i
```

我们将在8.6.2节给出定量信号量使用的例子。

8.4.5 Ada信号量编程举例

Algol-68是第一个引入信号量的语言。它提供了一种有两个操作符`up`和`down`的类型`sema`。为了说明几个使用信号量的简单例子，我们使用Ada中信号量的一个抽象数据类型.。

```
package Semaphore_Package is
  type Semaphore (Initial : Natural := 1) is limited private;
  procedure Wait (S : in out Semaphore);
  procedure Signal (S : in out Semaphore);
private
  type Semaphore is ...
end Semaphore_Package;
```

Ada并不直接支持信号量，但`Wait`和`Signal`过程可以由Ada同步原语构造，这些我们还未讨论，因此这里不给出信号量类型和包体的完整定义（见8.7节）。但是，抽象数据类型的
239 本质就是可以不知其实现情况而照样使用它们。

第一个例子是生产者/消费者系统，它通过一个有界缓冲区在两个任务间传递整数:

```
procedure Main is
  package Buffer is
    procedure Append (I : Integer);
    procedure Take (I : out Integer);
  end Buffer;
  task Producer;
  task Consumer;

  package body Buffer is separate; -- 见下面
  use Buffer;

  task body Producer is
    Item : Integer;
  begin
    loop
      -- 生产item
```

```
            Append (Item);
          end loop;
        end Producer;

        task body Consumer is
          Item : Integer;
        begin
          loop
            Take (Item);
            -- 消费item
          end loop;
        end Consumer;
      begin
        null;
      end Main;
```

缓冲区本身必须防止并发访问、向满缓冲区写和从空缓冲区读这三种情况。因此它使用了三个信号量:

```
with Semaphore_Package; use Semaphore_Package;
separate (Main)
package body Buffer is
  Size : constant Natural := 32;
  type Buffer_Range is mod Size;
  Buf : array (Buffer_Range) of Integer;
  Top, Base : Buffer_Range := 0;

  Mutex : Semaphore; -- 默认值为 1
  Item_Available : Semaphore (0);
  Space_Avaliable : Semaphore (Initial => Size);

  procedure Append (I : Integer) is
  begin
    Wait (Space_Available);
    Wait (Mutex);
      Buf (Top) := I;
      Top := Top+1;
    Signal (Mutex);
    Signal (Item_Available);
  end Append;

  procedure Take (I : out Integer) is
  begin
    Wait (Item_Available);
    Wait (Mutex);
      I := Buf (Base);
      Base := Base+1;
    Signal (Mutex);
    Signal (Space_Available);
  end Take;
end Buffer;
```

240

三个信号量的初值是不同的。Mutex是一个普通的互斥信号量，默认的初值是1; Item_Available防止从空缓冲区读，初值是0; Space_Available（初值是Size）用来防止向满缓冲区执行Append操作。

当程序开始执行时，任何调用Task的消费者任务都将在Wait (Item_Available) 上挂起，只有在生产者任务调用Append并执行Signal (Item_Available) 时消费者任务才可能继续。

8.4.6 使用C和POSIX的信号量编程

虽然几乎没有哪种编程语言直接支持信号量，但许多操作系统却支持。例如POSIX提供计数信号量，使运行在独立地址空间的进程（或同一地址空间的线程）通过共享存储器实现同步和通信。但是，在同一地址空间使用互斥锁（mutex）和条件变量实现同步和通信的效率更高——见8.6.3节。程序8-2定义了POSIX C信号量的接口（也提供了用字符串命名信号量的函数，但这里我们略去了）。标准的信号量操作*initialize*、*wait*和*signal*在POSIX中称为sem_init、sem_wait和sem_post。接口还提供了sem_trywait用于非阻塞等待；sem_timedwait用于定时等待，因为是由例程确定当前信号量的值（sem_getvalue)。

研究一个在实时程序里以各种不同形式出现的资源控制器的例子。为了简单，该例子使用线程而不是进程。提供两个函数：allocate和deallocate，每个都有一个用来指明请求的优先级的参数。假设调用线程释放资源和请求资源的优先级相同。为了易于展示，例子不考虑资源是如何传输的。此外，这个解决方案也没有防止竞争条件问题（见练习8.27)。

241

程序8-2 POSIX C的信号量接口

```
#include <time.h> typedef ... sem_t;

int sem_init (sem_t *sem_location, int pshared, unsigned int value);
  /* 把在位置 sem_location上的信号量初始化为value */
  /* 如果pshared 为 1, 则信号量可在进程或线程之间使用 */
  /* 如果 pshared 为 0, 则信号量只能在同一进程的线程之间使用 */

int sem_destroy (sem_t *sem_location);
  /* 清除在位置sem_location上的无名信号量 */

int sem_wait (sem_t *sem_location);
  /*信号量上的标准等待操作*/

int sem_trywait (sem_t *sem_location);
  /* 试图减少信号量 */
  /* 如果调用可能阻塞调用进程, 返回-1 */

int sem_timedwait (sem_t *sem, const struct timespec *abstime);
  /* 如果信号量在时间abstime之前不能被锁住, 返回-1 */

int sem_post (sem_t *sem_location);
  /* 信号量上的标准发信号操作*/

int sem_getvalue (sem_t *sem_location, int *value);
  /* 取信号量的当前值到由value 指向的位置; 负值表示等待线程的个数 */

/*上述所有函数如果成功, 就返回0 , 否则返回-1. */
/* 当上述任何一个函数返回出错状态时, */
/* 共享变量 errno 包含这个出错的原因*/
```

```
#include <semaphore.h>

typedef enum {high, medium, low} priority_t;
```

```
typedef enum {false, true} boolean;
sem_t mutex;    /* 用于对等待和忙碌的互斥访问*/
sem_t cond[3]; /* 用于条件同步*/
int waiting;    /* 在一个优先级级别等待的线程的个数*/
int busy; /* 指出该资源是否在用*/
void allocate (priority_t P)
{
  SEM_WAIT (&mutex); /* 锁住 mutex */
  if (busy) {
    SEM_POST (&mutex); /* 释放 mutex */
    SEM_WAIT ( &cond [P] ); /* 在正确优先级上等待 */
    /*资源已被分配 */
  }
  busy = true;
  SEM_POST (&mutex); /* 释放mutex */
}
```

242

一个简单的信号量mutex被用来确保所有分配和回收请求以互斥的方式处理。三个条件同步信号量cond[3]，用于在三个优先级（high、medium和low）上排队等待线程。函数allocate在资源未被使用时（由标志busy指示）分配资源。

回收函数只是简单地通知优先级最高的等待者的信号量。

```
int deallocate (priority_t P)
{
  SEM_WAIT (&mutex); /* 锁住 mutex */
  if (busy) {
    busy = false;
    /* 释放最高优先级等待线程 */
    SEM_GETVALUE (&cond[high], &waiting);
    if (waiting < 0) {
      SEM_POST (&cond[high]);
    }
    else {
      SEM_GETVALUE (&cond[medium], &waiting);
      if (waiting < 0) {
        SEM_POST (&cond[medium]);
      }
      else {
        SEM_GETVALUE (&cond [low], &waiting);
        if (waiting < 0) {
          SEM_POST (&cond[low]);
        }
        else SEM_POST (&mutex);
        /* 没有等待者，释放锁*/
      }
    }
    /* 资源和锁传递给最高优先级线程*/
    return 0;
  }
  else return -1; /* 出错返回 */
}
```

243

初始化例程将标志busy置为false，并创建由allocate和deallocate使用的四个信号量。

```
void initialise () {
  priority_t i;

  busy = false;
  SEM_INIT (&mutex, 0, 1);
  for (i = high; i<= low; i++) {
    SEM_INIT (&cond[i] , 0, 0);
  };
}
```

由于C同POSIX的绑定使用非零返回值指示有错误发生，所以必须将每个POSIX调用都封装在一个if语句块里。这使得代码更难理解（Ada和C++同POSIX的绑定采取了当有错误发生时就引发异常的策略）。因而，同本书中的其他C例子一样，使用SYS_CALL代表对sys_call的调用和任何合适的出错恢复（见6.1.1节）。对于SEM_INIT可能还包括重试（retry）。

希望使用资源的线程可能会发出以下的调用：

```
priority_t my_Priority;

...
allocate (my_priority); /* 等待资源 */
  /* 使用资源*/
if (deallocate (my_priority) <= 0) {
  /* 不能回收资源，执行某个恢复操作 */
}
```

8.4.7 对信号量的批评

虽然信号量是一个优雅的低级同步原语，但是建立在只使用信号量上的实时程序还是易出错的。哪怕对信号量的一次遗漏和放错地方都会在运行时导致整个程序崩溃。当软件在处理稀少但却很关键的事件时，信号量可能不能确保互斥，并且可能发生死锁。我们需要的是一个更结构化的同步原语。

信号量所提供的就是一种在临界段编程处理互斥的方法。一个更结构化的方法将直接处理互斥。我们将在8.5到8.8节讨论这些结构。

在8.4.5节给出的例子表明了一个用Ada为信号量构建的抽象数据类型。然而，没有一种高级并发编程语言是完全依赖信号量的。它们有很重要的历史意义，但按理说它们对实时领域

244 并不合适。

8.5 条件临界区

条件临界区（conditional critical region，CCR）试图克服同信号量有关的一些问题。临界区是一段被确保可以互斥执行的代码。这必须和临界段的概念进行比较：临界段是应该互斥执行的代码段（但在出错的情况下却有可能不是）。很明显，把临界段编程为临界区肯定符合互斥要求。

那些受到保护不被并发使用的变量可以被分组成不同的命名区域，并标记为资源。当有一个进程在临界区内已经是活动的时候，其他进程不能进入这个临界区。条件同步由位于这些区的守备提供。当某个进程想进入一个临界区时，它会评估守备的值（在互斥的条件下），

如果守备为真，那么它就进入；反之该进程被延迟。正如信号量一样，当有多个进程被延迟而试图进入同一临界区时（不管有什么原因），程序员不应该假定任何访问顺序。

为了说明CCR的用法，下面给出有界缓冲区程序的一个框架：

```
program buffer_eg;
  type buffer_t is record
    slots        : array (1..N) of character;
    size         : integer range 0..N;
    head, tail   : integer range 1..N;
  end record;

  buffer : buffer_t;

  resource buf : buffer;

  process producer;
    ...
    loop
      region buf when buffer.size < N do
        -- 将字符放进缓冲区等
      end region
      ...
    end loop;
  end

  process consumer;
    ...
    loop
      region buf when buffer.size > 0 do
        -- 从缓冲区取字符等
      end region
      ...
    end loop;
  end
end
```

使用CCR的一个潜在性能问题是：每当进程离开指名这个资源的CCR时，这些进程必须重新测试它们的守备。为了测试守备，挂起的进程必须再次变成可执行的，如果测试结果仍然是false，该进程又必须返回到挂起状态。 245

一个CCR的版本已在Edison中实现（Brinch-Hansen，1981），Edison是一种在多处理器系统上实现的用于嵌入式应用的语言。由于每个处理器只执行单个进程，所以它可以在必要的时候很方便地测试其守备。但是，这也会引起网络流量过量问题。

8.6 管程

条件区的主要问题是它们可以分散在整个程序中。管程（monitor）提供更加结构化的控制区，目的是减轻这个问题。管程也使用了一种更有效实现的条件同步形式。

预期中的临界区被写成过程的形式，并封装在一起，称为管程的单个模块。作为模块，所有必须在互斥条件下访问的变量是隐藏的，另外，作为管程，所有在模块里的过程调用都被保证在互斥的条件下执行。

管程是作为条件临界区的一种改进形式出现的，Dijkstra（1968b）、Brinch-Hansen（1973）和Hoare（1974）进行过这种结构的最初设计和分析。许多编程语言都有管程这种设施，例如Modula-1、并发Pascal和Mesa。

为了比较，继续有界缓冲区的例子，一个缓冲区管程会有如下的结构：

```
monitor buffer;
  export append, take;
  var (* 必要变量的声明 *)

  procedure append (I : integer);
    ...
  end;

  procedure take (var I : integer);
    ...
  end;
begin
   (* 管程变量的初始化*)
end
```

对于像Modula-2和Ada这样的语言，把缓冲区作为一个不同的模块（包）编程是很自然的。管程也采用这种方法。模块和管程之间惟一的不同是，对于后者来说，（上例中）对append

246 和/或take方法的并发调用按定义是串行化的。对于管程来说，不需要互斥信号量。

虽然为互斥作好了准备，但在管程中仍然需要条件同步。理论上，信号量仍然可以使用，但通常只提供了一个简单的同步原语。在Hoare的管程中（Hoare，1974），这个原语被称为**条件变量**，有两个施加在该变量上的操作，由于这两个操作跟信号量结构类似，所以它们又被称为wait和signal。当一个进程进行wait操作时，该进程被阻塞（挂起）并且被放在与该条件变量相关联的队列中（这可以和一个值为0的信号量上的wait操作对比；然而，值得注意的是在条件变量上的wait操作总是阻塞，这一点与在信号量上的wait不同）。然后，被阻塞进程释放它在管程上的互斥，从而允许另一个进程进入。当一个进程执行signal操作时，它将释放一个被阻塞的进程。如果没有进程在特定的条件变量上阻塞，那么signal操作无任何影响。（再次注意，相比之下作用在信号量上的signal操作总是对信号量有影响。实际上，对管程的wait和signal在语义上更类似于suspend和resume。）有界缓冲区例子的完整代码如下：

```
monitor buffer;
  export append, take;
  const size = 32;
  var buf : array[0 ...size-1] of integer;
      top, base : 0..size-1;
      SpaceAvailable, ItemAvailable : condition;
      NumberInBuffer : integer;

  procedure append (I : integer);
  begin
    if NumberInBuffer = size then
      wait (SpaceAvailable);
    buf [top] : = I;
    NumberInBuffer := NumberInBuffer+1;
    top := (top+1) mod size;
```

```
      signal (ItemAvailable)
    end append;

    procedure take (var I : integer);
    begin
      if NumberInBuffer = 0 then
        wait (ItemAvailable);
      I : = buf[base];
      base := (base+1) mod size;
      NumberInBuffer := NumberInBuffer-1;
      signal (SpaceAvailable)
    end take;

  begin (* 初始化 *) .
    NumberInBuffer := 0;
    top : = 0;
    base := 0
  end;
```

如果一个进程在缓冲区为空的情况下调用take，那么它将在ItemAvailable上挂起。然而，向缓冲区加入项的进程将在有一个项成为可用时唤醒这个挂起的进程。 247

上面所给的wait和signal语义是不完整的，正如它们所表达的，两个或多个进程可以在一个管程内成为活动的。这种情况会紧跟着阻塞进程被释放的signal操作发生。被释放的进程和释放该进程的进程都在管程内执行。为了防止这种不希望有的活动，必须改变signal操作的语义。在语言中可以使用如下四种方法:

1）signal仅作为进程离开管程之前的最后操作（这就是上面缓冲区例子的情况）。

2）signal操作还具有执行返回语句的副作用，也就是说，进程将被强迫离开管程。

3）解除另一个进程阻塞的signal操作有阻塞自己的效果，该进程仅在管程空闲时后可以再次执行。

4）解除另一个进程阻塞的signal操作不会阻塞，而且，一旦发信号进程退出，被释放的进程必须通过竞争来获得对管程的访问。

（3）是Hoare在他的有关管程的最初论文里提出的，由于signal操作而被阻塞的进程放在“就绪队列”，管程空闲时，就先于在入口阻塞的进程被选择执行。（4）的情况是把被释放的进程放置在“就绪队列”里。

由于管程的重要性，下面简要介绍两种支持这种结构的语言。

8.6.1 Modula-1

Modula-1（它现在很知名）是Modula-2和Modula-3的先驱，但却有一个显著不同的进程模型。它使用显式进程声明（不是合作例程）和管程，而管程被称为**接口模块**。条件变量被称为信号（这可能会让人有些困惑），并且有三个作用于它的过程。

1）过程wait (s,r) 延迟调用进程直到它收到信号s为止。延迟时，给这个进程一个优先级r（或延迟级别r），r必须是一个默认值为1的正值整型表达式。

2）过程send (s) 发送信号s给在等待信号s中有着最高优先级的进程。如果几个等待进程有着同样的优先级，那么那个等待时间最长的进程接受该信号，执行发送的进程被挂起。如果没有等待进程，那么这个调用没有任何效果。

3）布尔型函数awaited（s）用来测试是否在s上有阻塞的进程，如果至少有一个，那
248 么函数值为真，反之为假。

下面是Modula-1接口模块的一个例子。它实现了前面用信号量编程的资源控制需求:

```
INTERFACE MODULE resource_control;

  DEFINE allocate, deallocate;  (* 出口表 *)

  VAR busy : BOOLEAN;
    free : SIGNAL;

  PROCEDURE allocate;
  BEGIN
    IF busy THEN WAIT (free) END;
    busy := TRUE;
  END;

  PROCEDURE deallocate;
  BEGIN
    busy := FALSE;
    SEND (free)
  END;

BEGIN (* 模块的初始化*)
  busy := FALSE
END.
```

注意在deallocate中可以插入

```
if AWAITED (free) then SEND (free)
```

但由于SEND（free）的效果为空，当AWITED（free）为假时，进行这个测试将不会获得任何效果。

8.6.2 Mesa

到目前为止，假设当挂起进程被解除阻塞时，引起它被阻塞的条件将不再成立。例如，等待资源释放（也就是说，不是忙等待）而被阻塞的进程可以假设当其再次执行时它已被释放了。类似地，在一个有界缓冲区管程里延迟的进程一旦再次执行，就可以继续它自己的动作。

Mesa（Lampson and Redell，1980）采用了一种不同的方法。Mesa有一个“wait”操作，但进程却不能假设当其被唤醒时那个引起阻塞的条件就消失了。“notify”（对比signal）操作只是表明被阻塞进程应该重新测试这个条件。Mesa也支持广播（broadcast）操作，该操作通知所有在某个特定条件下等待的进程（为了保持管程的排它性这些进程一次只能被唤醒一个）。执行notify或broadcast操作的进程不会被阻塞。

在Mesa管程里允许三种过程：入口过程，内部过程和外部过程。入口过程与管程锁一起
249 执行。只可以由入口过程发出对内部过程的调用（在Modula-1中，内部过程根本不在定义列表中出现）。外部过程可以在没有管程锁的情况下被调用，用来观察管程的当前状态。它们不能改变变量、调用内部过程或使用条件变量。这些限制在编译时进行检查。

为了给出一个Mesa管程的例子，研究一下资源分配问题的求精。这里不简单地是管理忙或闲的资源，而是管程必须控制对N个资源实例的访问。分配请求把要求的资源实例的个数（$<N$）作为参数传递。为了简化，在allocate和deallocate过程中这个请求参数将不被检查。为了获

得安全的结构，有必要对返回一组资源的进程进行检查，看看是否确实已经将它们返回。安全资源控制将在第11章讨论。Mesa代码就在下面，过程`free_resources`被用来说明外部过程。读者应该可以理解这个模块而不必知道更多的Mesa知识（注意在Mesa中赋值操作符是<-）。

```
Resource_Control :  monitor =
begin
  const N = 32;
  free : condition;
  resource_free : positive <- N;

  allocate : entry procedure[size : positive] =
    begin
      do
        if size <= resource_free then exit; -- 从循环退出
        wait free;
      endloop;
      resource_free <- resource_free - size;
  end;

  deallocate : entry procedure[size : positive] =
  begin
    localfree[size];
    broadcast free;
  end;

  localfree : internal procedure [S : positive) =
  begin
    resource_free <- resource_free + S;
  end;

  free_resources : external procedure return[p : positive] =
  begin
    p <- resource_free;
  end;
end;
```

因为回收操作可以为大量的阻塞分配进程继续执行释放足够的资源，所以进行一次广播。所有被阻塞的进程（在`free`上）将按次序执行：如果当前的`size`小于或等于`resource_free`，则获得资源，否则再次被阻塞。 250

如果Mesa不支持广播设施，那么程序员将不得不保持一个被阻塞进程的计数器，并为每个进程解除进程链中下一个进程的阻塞。这是由于notify操作的语义造成的，它不阻塞进程。在Modula-1中，这是隐式地做到的，因为send操作确实会引起到被唤醒进程的上下文切换。

```
PROCEDURE allocate (size : INTEGER);
BEGIN
  WHILE size > resource_free DO
    WAIT (free);
    SEND (free)
  END;
  resource_free := resource_free - size
END;

PROCEDURE deallocate (size : INTEGER);
```

```
BEGIN
  resource_free := resource_free + size;
  SEND (free)
END;
```

如果语言支持定量信号量，那么分配和回收过程会非常简单:

```
procedure allocate (size : integer);
begin
  wait (QS, size)
end;

procedure deallocate (size : integer);
begin
  signal (QS, size)
end;
```

这里QS是一个定量信号量，被初始化为系统的总资源数。

8.6.3 POSIX互斥锁和条件变量

在8.4.6节中，POSIX信号量被描述为在进程之间和线程之间使用的机制。假如支持对POSIX的线程扩展，那么在同一地址空间的线程之间使用信号量进行通信和同步的代价是很高的，高得如同非结构化的一样。**互斥锁**（mutex）和**条件变量**合起来提供了管程的功能，却有一个过程化接口。程序8-3定义了基本C接口以及一些处理属性对象的函数。

程序8-3 C同POSIX互斥锁和条件变量的接口

```
#include <time.h>

typedef ... pthread_mutex_t;
typedef ... pthread_mutexattr_t;
typedef ... pthread_cond_t;
typedef ... pthread_condattr_t;

int pthread_mutex_init (pthread_mutex_t *mutex,
                        const pthread_mutexattr_t *attr);
  /* 将一个互斥锁的某些属性初始化 */
int pthread_mutex_destroy (pthread_mutex_t *mutex);
  /* 销毁一个互斥锁 */
  /* 未定义行为，如果这个互斥锁被锁上的话 */

int pthread_mutex_lock (pthread_mutex_t *mutex);
  /* 锁上这个互斥锁; 如果已经锁上，挂起调用线程 */
  /* 该互斥锁的拥有者是锁上它的那个线程 */
int pthread_mutex_trylock (pthread_mutex_t *mutex);
  /* 像对锁一样，但如果该互斥锁已经锁上，返回一个错误 */
int pthread_mutex_timedlock (pthread_mutex_t *mutex,
                             const struct timespec *abstime);
  /*像对锁一样，但如果超时后还得不到锁，返回一个错误 */

int pthread_mutex_unlock (pthread_mutex_t *mutex);
  /* 为该互斥锁解锁，如果是由拥有者线程调用 */
  /* 未定义行为，如果调用线程不是拥有者 */
  /* 未定义行为，如果该互斥锁未锁上 */
```

```
  /* 成功时，结果是释放一个阻塞线程 */

int pthread_cond_init (pthread_cond_t *cond,
                       const pthread_condattr_t *attr);
  /* 将一个条件变量的某些属性初始化 */
int pthread_cond_destroy (pthread_cond_t *cond);
  /* 销毁一个条件变量 */
  /* 未定义行为，如果线程在这个条件变量上等待的话 */

int pthread_cond_wait (pthread_cond_t *cond,
                       pthread_mutex_t *mutex);
  /* 由拥有一个被锁上的互斥锁的线程调用 */
  /* 未定义行为，如果该互斥锁未锁上 */
  /* 原子地将该调用线程阻塞在cond 变量上，并释放mutex的锁 */
  /* 成功返回表示该互斥锁已被锁上 */
int pthread_cond_timedwait (pthread_cond_t *cond,
          pthread_mutex_t *mutex, const struct timespec *abstime);
  /* 同pthread_cond_wait，除了如果超时到期，返回一个错误 */

int pthread_cond_signal (pthread_cond_t *cond);
  /* 为至少一个被阻塞的线程解除阻塞 */
  /* 如果没有阻塞线程，则无任何影响 */
  /* 被解除阻塞的线程自动地竞争关联的互斥锁 */
iht pthread_cond_broadcast (pthread_cond_t *cond);
  /* 为所有阻塞线程解除阻塞 */
  /* 如果没有阻塞线程，则无任何效果 */
  /* 被解除阻塞的线程自动地竞争关联的互斥锁 */

/* 所有上述函数，如果成功，返回0 */
```

每个管程都有一个关联的（已初始化）mutex变量，在管程（临界区）上的所有操作都是由对mutex的加锁（pthread_mutex_lock）和解锁（pthread_mutex_unlock）的调用包围起来的。通过将mutex同条件变量关联起来提供条件同步。注意，当线程在条件变量(pthread_cond_wait，pthread_cond_timedwait）上等待时，它在相关联的mutex上的锁被释放。还有，当线程成功地从条件等待返回时，它又对mutex上锁。然而，因为可能会有多个线程被释放（pthread_cond_ signal)，程序员必须再次测试最初引起线程等待的条件。

考虑下面使用互斥和条件变量的整数有界缓冲区。缓冲区由以下几部分组成：一个mutex，两个条件变量（buffer_not_full和buffer_not_empty)，缓冲区的项目计数，缓冲区本身和缓冲区里第一个和最后一个项的位置。例程append对缓冲区加锁，当缓冲区满时，在条件变量buffer_not_full上等待。当缓冲区有空闲空间时，整型数据项被放入缓冲区，mutex被解锁，发送buffer_not_empty信号。take例程在结构上也是类似的。

```
#include <pthreads.h>

#define BUFF_SIZE 10

typedef struct {
  pthread_mutex_t mutex;
  pthread_cond_t buffer_not_full;
  pthread_cond_t buffer_not_empty;
  int count, first, last;
```

```
  int buf[BUFF_SIZE];
  } buffer;

int append (int item, buffer *B ) {
  PTHREAD_MUTEX_LOCK (&B->mutex);
  while (B->count == BUFF_SIZE)
    PTHREAD_COND_WAIT (&B->buffer_not_full, &B->mutex);
  /* 放数据到缓冲区, 并更新count 和 last */
  PTHREAD_COND_SIGNAL (&B->buffer_not_empty);
  PTHREAD_MUTEX_UNLOCK (&B->mutex);
251~253  return 0;
}

int take (int *item, buffer *B ) {
  PTHREAD_MUTEX_LOCK (&B->mutex);
  while (B->count == 0)
    PTHREAD_COND_WAIT (&B->buffer_not_empty, &B->mutex);
  /*从缓冲区取数据, 并更新count 和first */
  PTHREAD_COND_SIGNAL (&B->buffer_not_full);
  PTHREAD_MUTEX_UNLOCK (&B- >mutex);
  return 0;
}

/* 还需要一个函数 initialize () */
```

虽然互斥和条件变量都作为管程的一种类型，但当线程从条件等待释放和其他线程试图访问临界区时，它们的语义是不同的。在POSIX中并没有规定哪个线程去接替，除非使用基于优先权的调度（见13.14.2）。对于大多数管程，给予被释放的线程以优先。

8.6.4 嵌套管程调用

管程的使用还存在许多问题，但最引人关注的是在一个管程里调用另一个管程的过程的语义是什么。

> 有争议的是：如果一个发出嵌套管程调用的进程在另一个管程里被挂起，将会发生什么（如果有的话）。在最后管程调用里的互斥将被进程放弃，这是因为等待和等价操作的语义。然而，来自嵌套管程调用的互斥将不会被进程放弃。试图调用在这些管程里的过程的进程将被阻塞。这有性能上的意义，因为阻塞将会减少系统呈现的并发性的数量。(Lampson and Redell，1980)

已经提出了针对嵌套管程问题的各种解决方法。其中最流行的、也被Java、POSIX、Mesa所采用一个方法是维持锁。其他的方法包括禁止嵌套过程调用（如Modula-1那样）和提供一些构造，这些构造规定在远程调用时，某些管程过程可以释放它们的互斥锁。

8.6.5 对管程的批评

管程对有界缓冲区这样的互斥问题给出了一个结构化和雅致的解决方案。然而它不能很好地处理管程外的条件同步问题。例如，在Modula-1中，信号是一种通用编程特性而不局限
254 于管程内使用。因此基于管程的语言表示了高级和低级原语的混合。所有围绕使用信号量的批评同样针对条件变量（如果不是更多的话）。

另外，虽然管程封装了所有与资源相关的实体，同时也提供了重要的互斥，但由于条件

变量的使用，其内部结构仍然可读性差。

8.7 保护对象

对管程的批评集中在条件变量的使用。通过使用守备来取代这种同步方法，可以获得一种更结构化的抽象。这种管程形式被命名为**保护对象**。Ada是提供这种机制的为数不多的语言之一，因此下面用Ada来描述它。

Ada保护对象封装数据项，并且只可以通过保护子程序或保护入口访问这些数据项。Ada语言保证这些子程序和入口的执行方式确保数据更新在互斥下进行。条件同步通过在入口点的布尔表达式（它们就是守备，但在Ada中称为**屏障**（barrier））提供，在任务允许进入之前测试布尔表达式的值是否为真。因此保护对象非常像管程和条件临界区。保护对象提供了具有条件临界区的高级同步机制的管程结构化设施。

保护单元可被声明为类型或者是单个实例，它有规格说明和体（因此其声明方式类似于任务）。其规格说明可以包含函数、过程和入口项。

下面的声明描述了保护类型如何用来提供简单的互斥：

```
-- 一个简单的整数
protected type Shared_Integer (Initial_value : Integer) is
  function Read return  Integer;
  procedure Write (New_Value : Integer);
  procedure Increment (By : Integer);
private
  The_Data : Integer := Initial_Value;
end Shared_Integer;

My_Data : Shared_Integer (42);
```

以上的保护类型封装了一个共享整数。对象`My_Data`声明了该保护类型的一个实例并给封装的数据传递了一个初始值。封装的数据现在只可以通过三个子程序来访问：`Read`，`Write`和`Increment`。

保护过程对封装数据提供了互斥的读写访问。在这种情况下，对过程`Write`或`Increment`的并发调用将被互斥执行，也就是说在任何时刻都只能有一个操作执行。

保护函数对封装数据提供了并发只读访问。在上面的例子中，这意味着可以有多个对`Read`操作的调用同时执行。但是对保护函数和保护过程的调用仍然必须互斥执行。假如当前有一个过程调用执行，那么对`Read`的调用就不能执行；同样的，如果有一个或者多个并发执行的函数调用，那么对过程的调用也不能执行。 255

`Shared_Integer`的体就只是：

```
protected body Shared_Integer is
  function Read return Integer is
  begin
    return The_Data;
  end Read;

  procedure Write (New_Value : Integer) is
  begin
    The_Data := New_Value;
```

```
  end Write;

  procedure Increment (By : Integer) is
  begin
    The_Data := The_Data + By;
  end Increment;
end Shared_Integer;
```

保护入口类似于保护过程，因为它也确保在互斥的条件下执行，并对封装数据具有读写访问权。然而，保护入口在保护对象的体内有一个布尔表达式（屏障）作为守备，当发出入口调用时，如果该屏障测试为假，那么调用将被挂起直到屏障测试为真且在保护对象内没有其他任务活动。因此保护入口调用可以被用来实现条件同步。

考虑在几个任务之间共享的有界缓冲区。该缓冲区的规格说明如下：

```
-- 一个有界缓冲区

Buffer_Size : constant Integer := 10;
type Index is mod Buffer_Size;
subtype Count is Natural range 0 .. Buffer_Size;
type Buffer is array (Index) of Data_Item;

protected type Bounded_Buffer is
  entry Get (Item: out Data_Item);
  entry Put (Item: in Data_Item);
private
  First : Index := Index' First;
  Last : Index := Index' Last;
  Number_In_Buffer : Count := 0;
  Buf : Buffer;
end Bounded_Buffer;
```

256

```
My_Buffer : Bounded_Buffer;
```

以上代码声明了两个入口，它们代表了缓冲区的公有接口。在私有部分声明的数据项是那些必须互斥访问的数据项。在这种情况下，缓冲区是一个数组，经由两个序标访问，缓冲区还有一个记录缓冲区内数据项数目的计数器。

下面给出这个保护对象的体：

```
protected body Bounded_Buffer is

  entry Get (Item: out Data_Item)
    when Number_In_Buffer /= 0 is
  begin
    Item := Buf (First);
    First := First + 1; -- 求模类型运算，卷筒式轮转
    Number_ln_Buffer := Number_In_Buffer - 1;
  end Get;

  entry Put (Item: in Data_Item)
    when Number_In_Buffer /= Buffer_Size is
  begin
    Last := Last +1; -- 求模类型运算，卷筒式轮转
    Buf (Last) := Item;
    Number_In_Buffer := Number_In_Buffer + 1;
```

```
    end Put;
  end Bounded_Buffer;
```

Get入口使用屏障"**when** Number_In_Buffer/=0"作为守备，只有当它被测试为真时任务才执行Get入口；类似地，入口Put也是一样的。屏障定义了一个前置条件，仅当该条件测试为真时入口才可以被接受。

虽然对保护对象的调用可以延迟，因为该对象处于使用中（也就是说，不能执行请求的读或读/写访问操作），但Ada并没把这种调用看成被挂起。然而，由于入口屏障为假而延迟的调用被认为是挂起，并放入一个队列里。这样做的原因是：

- 假设保护操作是短时的。
- 一旦启动了保护操作，就不能挂起它的执行——所有可能引起挂起的操作都是禁止的，并会引发异常——它只可以重新排队（见11.4节）。

因此，当一个任务试图访问保护对象时，不应该把它延迟太长的时间——但在与调度的次序有关的情况下除外。一旦过程（或函数）调用获得访问权，它就立即开始执行子程序，而入口调用要对屏障求值，当然，如果屏障为假那么调用将被阻塞。在13.14.1节中，将研究实时系统附件所要求的实现策略，它保证当一个任务试图访问保护对象的时候决不会被延迟。 257

8.7.1 入口调用和屏障

为了发出一个对保护对象的调用，任务只需简单地指出该对象和所需的子程序或入口的名字。例如为了把一些数据放入上面的有界缓冲区，需要调用任务：

```
My_Buffer.Put (Some_Item);
```

在任何时刻，保护入口或是打开的或是关闭的。如果测试布尔表达式为真，那么它是打开的，否则是关闭的。通常，在下列情况对保护对象的保护入口屏障进行求值：

(a) 任务调用一个保护入口，并且相关屏障引用变量或属性，而自从屏障上次求值后这些变量或属性可能已经改变了；

(b) 任务离开保护过程或保护入口，并且在入口上还有排队任务，这些入口的屏障又引用变量或属性，而自从屏障上次求值后这些变量或属性可能已经改变了。

保护函数调用不会引起屏障的求值。值得注意的是，在一个保护入口或过程里面是不可能存在两个活动任务的，这是因为只有当任务离开对象时屏障才会被求值。

当一个任务调用保护入口或保护子程序时，保护对象可能已经被上锁：如果一个或多个任务正在该保护对象里面执行保护函数，则称该保护对象有一个活动的**读锁**；如果任务正在执行一个保护过程或保护入口，那么称该对象有一个活动的**读/写锁**。

如果有多个任务调用同一个关闭的屏障，那么这些调用会排队，默认排队方式是先来先服务。但是也可以改变这种默认的方式（见13.14.1）。

下面再给出另外两个例子。先考虑简单的资源控制器，该控制器在前面已由其他语言给出。当只请求（和释放）单个资源时，代码是很简单的：

```
protected Resource_Control is
  entry Allocate;
  procedure Deallocate;
private
  Free : Boolean := True;
```

```
end Resource_Control;

protected body Resource_Control is

  entry Allocate when Free is
  begin
    Free := False;
  end Allocate;
258 procedure Deallocate is
  begin
    Free := True;
  end Deallocate;

end Resource_Control;
```

该资源最初是可用的，因此Free标志为真。对Allocate的调用将改变该标志，因此关闭屏障，随后所有对Allocate的调用将被阻塞。当Deallocate被调用时，屏障被打开。这将允许一个等待任务通过执行Allocate而继续执行。这个执行的结果是再次关闭屏障，因此入口体不可能进一步执行（直到又一次调用Deallocate）。

有趣的是，一般的资源控制器（资源按组被分配和释放）如果只使用守备是比较难实现的。其原因将在第11章解释，那里将详细讨论资源控制问题。

每个入口队列都有一个同它关联的属性，该属性表明当前有多少个任务在排队。下面的例子中要用到这一点。假设任务希望对大量的等待任务广播（Message类型的）一个值，等待进程将调用Receive入口，该入口仅当有新消息到达时打开。在那个时候，所有等待任务被释放。

虽然所有任务现在可以前进，但它们必须按严格的次序通过保护对象（因为在对象里只能有一个活动任务）。最后出来的对象必须再次关闭屏障，这样随后对Receive的调用被阻塞，直到有一个新的消息被广播。这种显式地打开和关闭屏障的方法可以和条件变量的用法做比较，一旦所有的进程退出，条件变量的方法将不会有持续的效果（在管程里面）。广播例子的代码如下（注意属性Count代表在入口排队的任务数）：

```
protected type Broadcast is
  entry Receive (M : out Message);
  procedure Send (M : Message);
private
  New_Message : Message;
  Message_Arrived : Boolean := False;
end Broadcast;

protected body Broadcast is

  entry Receive (M : out Message) when Message_Arrived is
  begin
    M := New_Message;
    if Receive' Count = 0 then
      Message_Arrived := False;
    end if;
  end Receive;

  procedure Send (M : Message) is
  begin
259 if Receive' Count > 0 then
```

```
          Message_Arrived := True;
          New_Message := M;
        end if;
      end Send;

    end Broadcast;
```

因为可能没有任务等待消息，因此send过程必须检查count属性。只有count大于0时才会打开屏障（并记录新消息）。

最后，本节给出在8.4.5节所给出的信号量的完整Ada包实现。这表明保护对象不仅是一个优秀的结构抽象，而且也具有与信号量相同的表达能力。

```
package Semaphore_Package is
  type Semaphore (Initial : Natural := 1) is limited private;
  procedure Wait (S : in out Semaphore);
  procedure Signal (S : in out Semaphore);
private
  protected type Semaphore (Initial : Natural := 1) is
    entry Wait_Imp;
    procedure Signal_Imp;
  private
    Value : Natural := Initial;
  end Semaphore;
end Semaphore_Package;


package body Semaphore_Package is
  protected body Semaphore is
    entry Wait_Imp when Value > 0 is
    begin
      Value := Value - 1;
    end Wait_Imp;

    procedure Signal_Imp is
    begin
      Value := Value + 1;
    end Signal_Imp;
  end Semaphore;

  procedure Wait (S : in out Semaphore) is
  begin
    S.Wait_Imp;
  end Wait;

  procedure Signal (S :  in out Semaphore) is
  begin
    S.Signal_Imp;
  end Signal;
end Semaphore_Package;
```

260

8.7.2 保护对象和标记类型

很明显Ada并没有完全整合并发和面向对象编程的模型。例如，任务和保护对象都是不可

扩展的。详细讨论如何最好地利用Ada的特性来获得尽可能多的整合已超出本书的范围——见文献[Burns and Wellings（1998）]。此外，关于为Ada加入扩展保护类型的全面讨论，见文献[Wellings等（2000）]。

8.8 同步方法

在许多方面，Ada的保护对象很像基于类的面向对象编程语言里的对象。当然，主要的不同是Ada的保护对象不支持继承关系。Java是一种完全整合了并发和面向对象模型的语言，它提供了一种在类和对象的上下文中实现管程的机制。

在Java中，每个对象都有一个相关联的锁。这个锁不能被应用程序直接访问，但是

- 方法修饰符synchronized和
- 块同步

对它起作用。当一个方法带有synchronized方法修饰符的时候，获得同这个对象相关联的锁之后才能执行访问该方法。因此，同步方法对封装在对象中的数据进行互斥访问，如果那个数据只由其他同步方法访问的话。非同步方法不需要锁，因此可以在任何时刻调用。因此为了获得完全的互斥，每个方法都要标记为synchronized。下面的类表示了一个简单的共享整数:

```
class SharedInteger
{
  private int theData;

  public SharedInteger (int initialValue)
  {
    theData = initialValue;
  }

  public synchronized int read ()
  {
    return theData;
  };

  public synchronized void write (int newValue)
  {
    theData = newValue;
  };

  public synchronized void incrementBy (int by)
  {
    theData = theData + by;
  };
}

SharedInteger myData = new SharedInteger (42);
```

261

块同步提供一种同步机制，借以将块标记为同步的。synchronized这个关键字把一个对象作为参数，在块继续执行之前必须获得该对象的锁。因此同步方法可以有效地实现如下:

```
public int read ()
{
  synchronized (this) {
    return theData;
```

```
    }
  }
```

这里this是Java获得当前对象的机制。

按照它的最一般性的使用，同步块破坏了管程这类机制的一个优点：通过对同步约束的封装，把对象同程序中的一个位置关联起来。这是因为如果当其他对象在同步语句指名了一个特定对象时，只考察这个对象本身是不可能理解与这个对象相关联的同步问题的。但是，如果小心使用的话，这种设施扩大了基本的模型，使编程能实现更有表达能力的同步约束，下面将很快说明这一点。

虽然同步方法或同步块可以互斥地访问在一个对象里的数据，但如果数据是**静态的**，那么这种访问就不合适了。静态数据在所有从这个类创建的对象之间共享，为了保证对这种数据的互斥访问，就要求**所有**对象被上锁。

在Java中类本身也是对象，因此类也有一个相关的锁。通过用synchronized修饰符标记静态方法或者标识这个类在同步块语句中的对象，可以访问这个锁。后者可以从与对象相关联的Object类得到。然而要注意，当在对象上同步时，是不可能获得类范围的锁的。因此，为了获得在静态变量上的互斥访问，需要下面的类（例如）：

```
class StaticSharedVariable
{
  private static int shared;
  ...
  public synchronized int Read ()
  {
    synchronized (this.getClass () )
    {
      return shared;
    };
  }

  public synchronized static void Write (int I)
  {
    shared = I;
  };
}
```

262

8.8.1 等待和通知

为了得到条件同步，需要更进一步的支持。这些支持来自预定义对象类提供的方法：

```
public void wait ();
            // 抛出 IllegalMonitorStateException
public void notify ();
            // 抛出IllegalMonitorStateException
public void notifyAll ();
            // 抛出IllegalMonitorStateException
```

这些方法被设计成只在拥有对象锁的方法内调用（即它们是同步的）。如果在没有锁的方法里调用，就抛出异常IllegalMonitorStateException。

方法wait总是阻塞调用线程并释放对象的锁。如果调用来自于嵌套管程里面，那么，只

有内部锁是与wait相关联的，并释放这个内部锁。

方法notify唤醒一个等待线程，被唤醒的线程没有被Java语言定义（但是被实时Java定义了，参看13.14.3节）。注意notify并不释放锁，因此在被唤醒的线程可以继续执行之前它必须等待，直到获得锁。为了唤醒**所有**等待线程可以使用notifyAll方法，同样它也不释放锁，所有被唤醒线程必须等待，当锁释放时还必须相互竞争这个锁。如果没有等待线程，那么notify和notifyAll没有任何效果。

一个正在等待的线程也可以被唤醒，如果它被另一个线程中断的话。在这种情况下会抛出异常InterruptedException。本章忽略这种情况（这个异常被允许传播），但在10.9节会详细讨论这个问题。

虽然Java看起来也提供了与其他语言一样支持管程的等价设施，但却有一个显著的不同。这里没有显式的条件变量。因此当一个线程被唤醒时，不必假设其“条件”为真，因为不管是在等待什么条件，所有线程都有可能被唤醒。对许多算法来说这种限制不是一个问题，因为任务等待的条件是互斥的。例如，传统意义下的有界缓冲区有两个条件变量：WaitBuffer
263 NotFull和WaitBufferNotEmpty。然而，如果一个线程等待一个条件，肯定不会同时存在其他线程等待其他条件的情况。因此，线程能够假设：当它被唤醒的时候，缓冲区正处于合适的状态。

```
public class BoundedBuffer
{
  private int buffer[];
  private int first;
  private int last;
  private int numberInBuffer = 0;
  private int size;

  public BoundedBuffer (int length)
  {
    size = length;
    buffer = new int [size];
    last = 0;
    first = 0;
  };

  public synchronized void put (int item) throws InterruptedException
  {
      while (numberInBuffer == size) {
        wait ();
      };
      last = (last + 1) % size; // %是求模运算
      numberInBuffer++;
      buffer[last] = item;
      notify ();
  };

  public synchronized int get () throws InterruptedException
  {
      while (numberInBuffer == 0) {
        wait ();
```

```
        };
        first = (first + 1) % size; // %是求模运算
        numberInBuffer--;
        notify ();
        return buffer[first];
      };
    }
```

当然，如果notifyAll被用来唤醒线程，那么这些线程总是在进行前重新测试其条件。

一个标准的并发控制问题是**读者－写者**（readers-writers）问题。在这种情况下许多读者和写者都试图访问大数据结构。读者可以并发地读，因为它们不会修改数据；然而写者需要同其他读者和写者在数据上互斥。这种方案也存在许多不同的变种，这里考虑的是对等待的写者赋予优先权。因此只要存在写者，所有新的读者将被阻塞直到所有的写者完成。当然，在极端的情况下，这有可能导致读者饿死。 264

如果使用标准的管程，解决读者-写者问题要求四个管程过程：startRead，stopRead，startWrite，stopWrite。读者的结构是：

```
startRead ();
  // 读数据结构
stopRead ();
```

类似地，写者的结构是：

```
startWrite ();
  // 写数据结构
stopWrite ();
```

管程里面的代码提供了必要的同步，它使用了两个条件变量：OkToRead和OkToWrite。在Java中，这不能在单个管程里直接表达。下面研究解决这个问题的两种方法：

第一种方法是使用一个类：

```
public class ReadersWriters
{
  private int readers = 0;
  private int waitingWriters = 0;
  private boolean writing = false;

  public synchronized void startWrite () throws InterruptedException
  {
    while (readers > 0 || writing)
    {
      waitingWriters++;
      wait ();
      waitingWriters--;
    }
    writing = true;
  }

  public synchronized void stopWrite ()
  {
    writing = false;
    notifyAll ();
```

```
    }
    public synchronized void startRead () throws InterruptedException
    {
      while (writing || waitingWriters > 0) wait ();
      readers++;
    }
    public synchronized void stopRead ()
265 {
      readers--;
      if (readers == 0) notifyAll ();
    }
  }
```

在这个解决方案里，一个线程在等待请求后醒来时，必须重新测试它可以前进的条件。虽然这个方法允许多个读者或单个写者，但是可以证明它是效率低下的，因为在每次数据变得可用时它要唤醒所有的线程。而在这些被唤醒的线程中，有许多线程当它们最终获得访问管程的权利时，会发现它们仍然不能继续，因此不得不再次等待。

另一个由Lea（1997）提出的替代方案是使用另一个类来实现条件变量。即：

```
public class ConditionVariable {
  public boolean wantToSleep = false;
}
```

一般的方法是在另一个类里面创建这些条件变量的实例，然后使用块同步。为了避免在嵌套管程调用里等待，标志变量wantToSleep被用来指示管程是否想在条件变量上等待。

```
public class ReadersWriters
{
  private int readers = 0;
  private int waitingReaders = 0;
  private int waitingWriters = 0;
  private boolean writing = false;

  ConditionVariable OkToRead = new ConditionVariable ();
  ConditionVariable OkToWrite = new ConditionVariable ();

  public void startWrite () throws InterruptedException
  {
    synchronized (OkToWrite) // 取得条件变量的锁
    {
      synchronized (this) // 取得管程锁
      {
        if (writing | readers > 0) {
          waitingWriters++;
          OkToWrite.wantToSleep = true;
        } else {
          writing = true;
          OkToWrite.wantToSleep = false;
        }
      } // 放弃管程锁
      if (OkToWrite.wantToSleep) OkToWrite.wait ();
```

266

```
    }
  }
  public void stopWrite ()
  {
    synchronized (OkToRead)
    {
      synchronized (OkToWrite)
      {
        synchronized (this)
        {
          if (waitingWriters > 0) {
            waitingWriters--;
            OkToWrite.notify ();
          } else {
            writing = false;
            OkToRead.notifyAll ();
            readers = waitingReaders;
            waitingReaders = 0;
          }
        }
      }
    }
  }
  public void startRead () throws InterruptedException
  {
    synchronized (OkToRead) {
      synchronized (this)
      {
        if (writing | waitingWriters > 0) {
          waitingReaders++;
          OkToRead.wantToSleep = true;
        } else {
          readers++;
          OkToRead.wantToSleep = false;
        }
      }
      if (OkToRead.wantToSleep) OkToRead.wait ();
    }
  }
  public void stopRead ()
  {
    synchronized (OkToWrite)
    {
      synchronized (this)
      {
        readers--;
        if (readers == 0 & waitingWriters > 0) {
          waitingWriters--;
          writing = true;
```

```
            OkToWrite.notify ();
          }
        }
      }
    }
  }
```

267 每个条件变量是用在这个类声明的ConditionVariable类的一个实例表示的，这个类的行为同管程一样。条件在拥有管程锁和条件变量锁（它将在管程过程里面得到通知或等待）的时候求值。为了确保没有死锁发生，这些锁总是按同样的顺序获得。

8.8.2 继承和同步

面向对象模式和并行编程机制的结合引起所谓**继承反常**（inheritance anomaly）问题（Matsuoka and Yonezawa，1993）。如果一个类的操作之间的同步不是局部的，却依赖于这个类的操作的整个集合，那么将发生继承反常。当一个子类加入新操作时，为了应对这些新操作，必须改变定义在父类中的同步（Matsuoka and Yonezawa，1993）。

例如，考虑本节前面所提到的有界缓冲区。在Java中，测试条件（BufferNotFull和BufferNotEmpty）的代码被嵌入到方法里面。正如在第11章所示的，一般情况下这种方法有许多优点，因为它允许方法访问方法的参数以及对象属性。然而，当考虑继承的时候，它的确了引起一些问题。假设有界缓冲区被子类化，因而所有的访问都被禁止。加入两个新方法prohibitAccess和allowAccess。一个不太成熟的扩展版本可能包括下面的代码:

```
public class AccessError extends Exception{};
public class ControlledBoundedBuffer extends BoundedBuffer
{
  // 不正确代码

  boolean prohibited;

  ControlledBoundedBuffer (int length)
  {
    super (length);
    prohibited = false;
  };

  public synchronized void prohibitAccess () throws InterruptedException
  {
    if (prohibited) wait ();
    prohibited = true;
  }

  public synchronized void allowAccess () throws AccessError
  {
    if (!prohibited) throw new AccessError ();
    prohibited = false;
    notify ();
  }
}
```

268 这个扩展的主要问题是它没考虑超类也会发出wait和notify请求这样的事实。因此，作为在allowAccess方法里通知请求的结果，get和put方法可能醒来。

问题是超类方法里的代码必须改变，以反映新的同步条件。这是继承反常的核心所在。理想情况下，get和put的代码应该是完全可重用的。当然，假如已经知道有界缓冲区被编写的时候也将产生一个控制版本，那么可能已经选择一个不同的设计以便充分利用这种扩展。但是，无法预期所有的扩展。

可以采取的一个方法是总是将条件测试封装在while循环里，不管这种封装看起来是否必要，并且总是使用notifyAll。因此，如果缓冲区代码如下：

```
public class BoundedBuffer
{
  private int buffer[];
  private int first;
  private int last;
  private int numberInBuffer = 0;
  private int size;

  public BoundedBuffer (int length)
  {
    size = length;
    buffer = new int [size];
    last = size - 1;
    first = size - 1;
  };

  public synchronized void put (int item) throws InterruptedException
  {
      while (numberInBuffer == size) {
        wait ();
      };
      last = (last +1) % size;
      numberInBuffer++;
      buffer[last] = item;
      notifyAll ();
  };

  public synchronized int get () throws InterruptedException
  {
      while (numberInBuffer == 0) {
        wait ();
      };
      first = (first + 1) % size;
      numberInBuffer--;
      notifyAll ();
      return buffer[first];
  };
}
```

那么其子类可以写成： 269

```
public class LockableBoundedBuffer extends BoundedBuffer
{
  boolean prohibited;

  // 不正确代码
```

```
    LockableBoundedBuffer (int length}
    {
      super (length);
      prohibited = false;
    };

    public synchronized void prohibitAccess () throws InterruptedException
    {
      while (prohibited) wait ();
      prohibited = true;
    }

    public synchronized void allowAccess () throws AccessError
    {
      if (!prohibited) throw new AccessError ();
      prohibited = false;
      notifyAll ();
    }

    public synchronized void put (int item) throws InterruptedException
    {
      while (prohibited) wait ();
      super.put (item);
    }

    public synchronized int get () throws InterruptedException
    {
      while (prohibited) wait ();
      return (super.get () );
    }
  }
```

令人遗憾的是这种方法有一个小bug。考虑如下的情况：生产者试图把数据放入一个满缓冲区内。因为并不禁止对缓冲区的访问，所以调用super.put (item)，这个调用被阻塞，等待BufferNotEmpty状态。现在禁止对缓冲区的访问。对put和get的任何另外的调用被困在已覆盖的子类方法里面，对prohibitAccess方法的调用也是这样。现在发出对allowAccess方法的调用，这使得所有等待线程被释放。假设被释放线程获得管程锁的次序是：消费者线程、试图禁止对缓冲区访问的线程、生产者线程。消费者线程发现并没有禁止对缓冲区的访问，就从缓冲区取数据。它发出一个notifyAll请求，但现在没有任何线程在等待。运行的下一个线程现在禁止对缓冲区的访问。接下来就是运行生产者线程，并放数据项到缓冲区里，虽然是禁
270 止访问的！

虽然这个例子好像有故意刁难的嫌疑，但它的确说明了由于继承异常而可能发生的bug。

小结

进程交互要求操作系统和并发程序语言支持同步和进程间通信。通信可以基于共享变量或消息传递。本章探讨了共享变量、它们表现出来的多重更新难题以及为解决这些困难所需的互斥同步。本章介绍了以下的术语：

- 临界段——必须在互斥条件下执行的代码

• 生产者－消费者系统——两个或多个进程经由一个有限缓冲区交换数据
• 忙等待——一个进程不断地检查一个条件看它现在是否可以执行
• 活锁——一种出错状态，在这种状态下一个或多个进程被阻止前进，直到处理周期耗尽

使用一些例子来证明只使用共享变量进行互斥编程是十分困难的。为了简化这些算法和消除忙等待引入了信号量。信号量是只能受wait和signal过程作用的一个非负整数。这些过程的执行具有原子性。

提供信号量原语的一个后果是给进程引入了一个新的状态，即挂起。它也引入了两个新的出错状态：

• 死锁——一个不能前进的挂起进程的集合
• 无限期推迟——一个进程由于获取不到可用的资源而不能执行（也称为停工或饿死）

信号量由于太低级和在使用中易出错而倍受批评。在它们之后，引入了四个更加结构化的原语：

• 条件临界区
• 管程
• 保护对象
• 同步方法

管程是一个重要的语言特性，在Modula-1、并发Pascal和Mesa中得到使用。它们由模块 271
和入口组成，（由定义）确保入口是互斥的。在管程体内，如果条件不适合一个进程继续执行那么它将挂起自己。这种挂起通过使用条件变量实现。当一个挂起进程被唤醒（通过在条件变量上执行signal操作）的时候，必须保证这不会导致模块内同时有两个进程是活动的。为了确保这种情况不会发生，语言必须按照如下的约定：

1）signal操作仅作为管程内的进程的最后动作执行。

2）signal操作有强迫进行signal操作的进程退出管程的副作用。

3）如果进行signal操作的进程导致管程内的另一个进程变成活动的，那么它本身要被挂起。

4）发出signal操作的进程没有挂起，并且一旦该进程退出，空闲的进程必须竞争对管程的访问。

管程的一种形式可以使用过程式接口来实现。POSIX 的互斥锁和条件变量提供这种设施。

虽然管程给互斥提供了一种高级结构，但其他同步却必须使用低级条件变量编程实现。这造成了在语言设计中令人遗憾的原语混合。Ada的保护对象具有管程的结构化优点和条件临界区的高级同步机制。

整合并发性和OOP是充满困难的。Ada试图为OOP和保护类型提供分别的设施而避免这个问题。然而，Java通过为类提供同步化成员方法解决了这个问题。通过使用这种设施（以及同步语句和wait、notify两个原语）提供一种灵活的基于面向对象的类管程设施。令人遗憾的是使用这种方法也引起了继承反常问题。

下一章我们将讨论基于消息的同步和通信的原语。使用这些设施的语言实际上根据它自己的权限把管程提升为一个活动的进程。因为一个进程在一个时间只能做一件事情，这样互斥就得到保证。进程不再通过共享变量通信而是直接进行通信。因此构造一种将通信和同步组合起来的单个高级原语是可能的。这种概念首先由Conway（1963）提出，随后在高级实时编程语言中得到利用。它形成了Ada和occam2中的会合机制的基础。 272

相关阅读材料

Andrews, G. A. (1991) *Concurrent Programming Principles and Practice*. Redwood City, CA: Benjamin/Cummings.

Ben-Ari, M. (1990) *Principles of Concurrent and Distributed Programming*. New York: Prentice Hall.

Burns, A. and Davies, G. (1993) *Concurrent Programming*. Reading: Addison-Wesley.

Burns, A. and Wellings, A. J. (1995) *Concurrency in Ada*. Cambridge: Cambridge University Press.

Butenhof, D. R. (1997) *Programming With POSIX Threads*. Reading, MA: Addison-Wesley.

Hyde, P. (1999) *Java Thread Programming*. Indianapolis, IN: Sams Publishing.

Lea, D. (1999) *Concurrent Programming in Java: Design Principles and Patterns*. Reading, MA: Addison-Wesley.

Oaks, A. and Wong, H. (1997) *Java Threads*. Sebastopol, CA: O'Reilly.

Nichols, B., Buttlar, D. and Farrell, J. (1996) *POSIX Threads Programming*. Sebastopol, CA: O'Reilly.

Silberschatz, A. and Galvin, P. A. (1998) *Operating System Concepts*. New York: John Wiley & Sons.

Wirth, N. (1977) Modula: a Language for Modular Multiprogramming. *Software Practice and Experience*, 7(1), 3–84.

练习

8.1 说明如何修改8.2节中的Peterson算法以使在忙等待时允许高优先级的进程比低优先级的进程具有优先使用权。

8.2 考虑在单个生产者和单个消费者之间共享的一个数据项。生产者是一个周期性任务：读传感器并将值写入该共享数据项。消费者是一个偶发任务：取出在共享数据项里由生产者放入的最新的值。

下面的包声称提供了一个生产者和消费者可以安全通信的类属算法，该算法不需要互斥或忙等待。

273

```
generic
  type Data is private;
  Initialvalue : Data;
package Simpsonsalgorithm is
  procedure Write (Item: Data); -- 不阻塞
  procedure Read (Item : out Data); -- 不阻塞
end Simpsonsalgorithm;

package body Simpsonsalgorithm is

  type Slot is (First, Second);

  Fourslot : array (Slot, Slot) of Data :=
          (First => (Initialvalue, Initialvalue) ,
           Second => (Initialvalue, Initialvalue) );
```

```
  Nextslot : array (Slot) of Slot := (First, First);

  Latest : Slot := First;
  Reading : Slot := First;

  procedure Write (Item : Data) is
    Pair, Index : Slot;
  begin
    if Reading = First then
      Pair := Second;
    else
      Pair := First;
    end if;
    if Latest = First then
      Index := Second;
    else
      Index := First;
    end if;

    Fourslot (Pair, Index) := Item;
    Nextslot (Pair) := Index;
    Latest := Pair;
  end Write;

  procedure Read (Item : out Data) is
    Pair, Index : Slot;
  begin
    Pair := Latest;
    Reading := Pair;
    Index := Nextslot (Pair);
    Item := Fourslot (Pair, Index);
  end Read;
end Simpsonsalgorithm;
```

通过解释下列情况发生时将会发生什么，仔细描述该算法是如何工作的:

(1) Write后面跟着Read

(2) Read被Write抢占

(3) Write被Read抢占

(4) 一个Read被多个Write抢占

(5) 一个Write被多个Read抢占

解释算法如何在数据更新和安全通信之间进行权衡。 274

许多编译器优化代码，使经常被访问的变量被保持在任务的局部寄存器中。解释在这种背景下上述算法声称能进行安全通信是否站得住的。为了使它能够适合所有Ada95的实现，是否需要对该算法做一些改变?

8.3 考虑一个既可以读也可写的共享数据结构。说明如何使用信号量来实现多个并发读操作或单个写操作，但不能同时读写。

8.4 说明如何使用信号量来实现Hoare的条件临界区。

8.5 说明如何使用信号量来实现Hoare（Mesa或Modular-1）的管程。

8.6 说明如何使用单个Modula-1接口模块来实现二元信号量。为了处理一般的信号量，该方案

应该如何修改？

8.7 考虑从磁盘传输数据的请求队列的调度。为了最小化磁头的移动，所有对一个特定柱面的请求都将在同一次内完成。调度程序对每个柱面依次扫描服务请求。写一个提供必要同步的Modula-1接口模块。假设这里有两个接口过程：一个是对一个特定柱面的访问请求，另一个是释放该柱面。用户按下面的磁盘驱动模块处理调用过程：

```
module disk_driver;

  define read, write;

  procedure read (var data:data_t; addr:disk_address)
  begin
      (* 求柱面地址 *)
      diskheadscheduler.request (cylinder);
      (* 读数据*)
      diskheadscheduler.release (cylinder);
  end read;

   (* 对 write类似 *)

end disk_driver;
```

8.8 写一个对数据库里的文件进行读写访问控制的Modula-1接口模块。为了维护文件的完整性，一次只允许一个进程更新文件；然而，只要不是处于更新过程中，可以有任意多个进程读文件。而且，文件的更新请求只在没有读操作时才被允许。然而，一旦更新文件的请求被
275 接受，那么后续的所有读文件请求将被阻塞直到不再有更新操作。

接口模块应该定义四个过程：startread、endread、startwrite和endwrite。假设接口模块应该按如下的方式使用。

```
module file_access;
  define readfile, writefile;
  use startread, endread, startwrite, endwrite;

  procedure readfile;
  begin
    startread;
     read the file
    endread;
  end readfile;

  procedure writefile;
  begin
    startwrite;
     write the file
    endwrite;
  end writefile;
end file_access.
```

8.9 一个计算机系统被用来控制通过单行道公路隧道的交通流。为了安全起见，在同一时间在该隧道里的车不能超过N辆。在入口处的交通灯控制车辆的进入，入口和出口处的车辆探测器被用来测量车流。

请规划Modula-1的两个PROCESS和其间控制交通流量的一个INTERFACE MODULE

结构。第一个进程监控出口处的车辆探测器，第二个监控入口处的探测器。INTERFACE模块控制交通灯本身。你可以假设在作用域中已经写有下面的函数:

```
procedure CARS_EXITED: NATURAL;
begin
   (* 返回上次该函数调用以来离开隧道的车辆数 *)
end CARS_EXITED;

procedure CARS_ENTERED : NATURAL;
begin
   (* 返回上次该函数调用以来进入隧道的车辆数 *)
end CARS_ENTERED;

procedure SET_LIGHTS ( COL : COLOUR);
begin
   (* 置交通灯为 COL *)
   (* COLOUR 是定义域中的一个枚举类型，其定义是：*)
   (* type COLOUR = (RED, GREEN); *)
end SET_LIGHTS;

procedure DELAY_10_SECONDS;
begin
   (* 延迟调用进程 10 秒钟 *)
end DELAY_10_SECONDS;
```

276

你的解决方案应该每10秒读一次传感器（经由CASE_EXITED和CARS_ENTERED函数）直到隧道满或空。当隧道满（这时灯被设置为红色）时，入口监控任务不应该继续调用CARS_ENTERED函数。类似的，当隧道空时，出口监控任务不应该继续调用CARS_EXITED函数。此外，任务不应该忙等待。你不应该对Modula-1的运行时任务调度程序做任何假设。

8.10 对管程的一个非难是条件同步太低级和非结构化。解释这句话是什么含义。更高级的管程同步原语可能采取下面的形式

```
WaitUntil 布尔表达式;
```

其中进程被延迟直到布尔表达式的值变为真。例如

```
WaitUntil x < y + 5;
```

将延迟进程直到x < y + 5。

虽然条件同步的这种形式更加结构化，但在大多数支持管程的语言里却没有找到它。请解释为什么会出现这种情况。在哪些情况下对以上高级同步设施的反对将是无效的？说明在管程内如何使用WaitUntil同步原语解决有界缓冲区问题。

8.11 考虑一个系统，其中有三个抽烟人进程和一个代理进程。每个抽烟人连续地卷烟并抽烟。卷一支烟需要三种材料，即烟草、烟和火柴。这些进程中的一个有烟草，另一个有烟，第三个有火柴。代理进程保证无限地提供所有这三种材料。代理随机选择地在桌上放两种材料。拥有第三种材料的抽烟人可以卷烟并抽烟。一旦抽烟人抽完了烟，他就通知代理，这样代理将再次在桌上放两种材料。该过程就这样循环下去。

请规划一个同步三个抽烟人和代理的管程结构。

8.12 对比UNIX内核（Bach，1986）为互斥和条件同步提供的内部设施和为信号量提供的对应

设施。

277 8.13 说明UNIX（Bach，1986）提供的内部设施如何用来控制对共享数据结构的访问。假设这里有多个读者和写者。虽然多个读者可以同时并发访问该数据结构，但是在任意时刻却只允许一个写者任务。更进一步说，不允许读者任务和写者任务的混合。方案还应该考虑给读者任务以优先权。

8.14 说明信号量的操作如何在一个没有忙等待的单处理器系统的操作系统核心里实现。你的解决方案需要哪些硬件设施？

8.15 说明怎样使用POSIX互斥锁和条件变量来实现一个可读可写的共享数据结构。允许多个并发的读者或单个写者，但不能同时允许读者和写者。

8.16 说明如何使用POSIX互斥锁和条件变量实现资源控制器。

8.17 使用Ada保护对象实现练习7.8的进程同步。

8.18 对比POSIX互斥锁和条件变量提供的设施和Ada保护对象提供的设施。

8.19 使用保护对象重做练习8.11。

8.20 使用保护对象实现定量信号量。

8.21 说明如何使用一个或多个保护对象实现Hoare管程。

8.22 解释下面的Ada保护对象所实现的同步：

```
protected type Barrier (Needed : Positive) is
    entry Wait;
private
    Releasing : Boolean := False;
end Barrier;

protected body Barrier is

    entry Wait when Wait' Count = Needed or Releasing is
    begin
        if Wait' Count = 0 then
            Releasing := False;
        else
            Releasing := True;
        end if;
    end Wait;
end Barrier;
```

下面的包提供了对POSIX互斥锁和条件变量的简化Ada绑定。所有的互斥和条件变量都被初始化为默认属性。

278

```
package Pthreads is
    type Mutex_T is limited private;
    type Cond_T is limited private;

    procedure Mutex_Initialise (M: in out Mutex_T);
    procedure Mutex_Lock (M: in out Mutex_T);
    procedure Mutex_Trylock (M: in out Mutex_T);
    procedure Mutex_Unlock (M: in out Mutex_T);

    procedure Cond_Initialise (C: in out Con_T);
    procedure Cond_Wait (C: in out Cond_T;
```

```
                                M : in out Mutex_T);
      procedure Cond_Signal (C: in out Cond_T);
      procedure Cond_Broadcast (C: in out Cond_T);
   private
      ...
   end Pthreads;
```

说明怎样才能用这个包实现如上面定义的屏障。在解决方案中不要使用任何Ada的通信和同步设施。

8.23 使用前一问题中给出的Ada包`Pthreads`，用Ada重做练习8.7。不要使用任何Ada同步设施。假设这里有两个接口过程：一个是请求访问特定柱面，另一个是释放柱面。

```
package Disk_Head_Scheduler is
  Size_Of_Disk : constant := ...;
  type Disk_Address is range 1 . . Size_Of_Disk;
  type Cylinder_Address is mod 1 .. 20;
  procedure Request (Dest: Cylinder_Address);
  procedure Release (Dest: Cylinder_Address);
end Disk_Head_Scheduler
```

用户可以调用下面的磁盘驱动器包中的过程：

```
with Disk_Head_Scheduler; use Disk_Head_Scheduler;
package Disk_Driver is
  procedure Read (Data: out  Data_T; Addr : Disk_Address);
  procedure Write (Data: in  Data_T; Addr : Disk_Address);
end Disk_Driver;

package body Disk_Driver is

  procedure Read (Data: out  Data_T; Addr : Disk_Address) is
      Cylinder : Cylinder_Address;
  begin
      -- 求柱面地址
      Disk_Head_Scheduler.Request (Cylinder);
      -- 读数据
      Disk_Head_Scheduler.Release (Cylinder);
  end Read;

  -- 对 Write类似

end Disk_Driver;
```
279

这个解决方案不应该包括任何任务或POSIX进程/线程。

8.24 下面的包定义了一个Ada信号量抽象。

```
generic
  Initial : Natural:= 1; -- 信号量的默认初始值
package Semaphore_Package is
  type Semaphore is limited private;
  procedure Wait (S : Semaphore);
  procedure Signal (S : Semaphore);
private
```

```
  type Semaphore is ...; -- 对此问题不需要
end Semaphore_Package;
```

使用Semaphore_Package说明如何实现下面的通信模式（及其相关的包规格说明）。多路广播（Multicast）是一个可以发送同一数据到多个等待任务的任务。Multicast抽象的包规格说明如下:

```
package Multicast is

  procedure Send (I : Integer);
  procedure Receive (I : out Integer);

end Multicast;
```

接受者任务通过调用以上包定义的Receive过程（在本例中数据是Integer类型的）来表明其接受数据的意愿。任务被这个调用阻塞。发送者任务靠调用Send过程表明其想多路广播数据。当发送者调用Send过程时，在接受调用上被阻塞的所有任务都被释放。发送者发送的数据将传给所有等待的任务。一旦Send过程完成，任何对Receive的新调用都必须等待下一个发送者。

说明如何使用信号量实现Multicast的包体。

8.25 广播（Broadcast）类似多播，但它要求所有的潜在接收者都必须接受数据。Broadcast抽象的包规格说明如下。

```
package Broadcast is

    -- 对10个任务

    procedure Send (I : Integer);
    procedure Receive (I : out Integer);

end Broadcast;
```

280

假定一个系统有（例如说）10个接受者任务。当这些任务准备好接受广播时，它们都调用过程Receive。这些任务被该调用阻塞。发送者任务通过调用Send过程表明其想广播数据。在释放接受者和传送数据之前，Send过程将等待，直到所有的10个任务都已准备好接受广播。如果有多个Send调用发生，那么这些调用将被排队。

说明如果使用练习8.24中的信号量包来实现Broadcast的包体。

8.26 曾有建议York城应该限制同一时刻进入该城的机动车辆的数量。一个建议是在进城的每个入口处建立监控检查点，当进城的车辆数量达到上限时，将交通灯变成红色。为了表明城里的机动车辆已满，把红色的灯设置为闪烁。为了取得这个效果，将压力传感器放在公路的出口和入口处。每次当有车进城的时候，发送一个信号给BarController任务（与任务入口调用一样）；当有车出城时做类似的事:

```
Max_Cars_In_City_For_Red_Light : constant Positive := N;
Min_Cars_In_City_For_Green_Light : constant Positive := N - 10;

type Bar is (Walmgate, Goodramgate, Micklegate,
             Bootham, Barbican);

task type Bar_Controller (G : Bar) is
  entry Car_Entered;
  entry Car_Exited;
```

```
end Bar_Controller;

Walmgate_Bar_Controller : Bar_Controller (Walmgate);
Goodramgate_Bar_Controller : Bar_Controller (Goodramgate);
Micklegate_Bar_Controller: Bar_Controller (Micklegate);
Bootham_Bar_Controller : Bar_Controller (Bootham);
Barbican_Bar_Controller : Bar_Controller (Barbican);
```

请说明这些任务体应如何合作，使得这些任务之一调用City_Traffic_Lights_Controller（其规格说明如下）以表明是否允许更多车辆进出。

```
task City_Traftic_Lights_Controller is
  entry City_Is_Full;
  entry City_Has_Space;
end City_Traffic_Lights_Controller;

task body Traffic_Lights_Controller is separate;
-- 在此问题中，对这个任务体不感兴趣
```

8.27 解释为什么8.4.6节中的资源控制器会遇到竞争条件问题。算法应如何修改以消除这个问题。

8.28 说明如何用Java实现读者/写者任务问题，其中读任务可以被赋予优先级，写任务按FIFO的方式服务。 281

8.29 说明如何用Java实现一个资源控制器。

8.30 请用Java实现定量信号量。

8.31 使用类ConditionVariable的两个实例重做有界缓冲区的例子，这两个实例分别用于BufferNotFull和BufferNotEmpty的状态。

8.32 扩展以上有界缓冲区的解决方案，使其产生一个可以禁止访问的缓冲区。这个方法可以解决8.8.2节中的继承反常问题吗?

8.33 考虑下面的Java类

```
public class Event
{
  public synchronized void highPriorityWait ();
  public synchronized void lowPriorityWait ();
  public synchronized void signalEvent ();
}
```

说明如何实现这个类，使得signalEvent释放一个高优先级等待线程，如果有一个线程在等待的话；如果没有高优先级的等待线程，那么释放一个低优先级等待线程；如果没有任何线程在等待，则signalEvent没有任何效果。

现在考虑可以为方法关联一个Id的情况。该算法应如何修改以使signalEvent操作唤醒合适的阻塞线程。 282

第9章 基于消息的同步与通信

共享变量式同步和通信的替代物是基于消息传递的同步和通信。这种方法的特征是对同步和通信二者使用一个构造。然而，在这个大类别里面，有多种多样的语言模型。消息传递语义上的这种多样性由以下三个问题产生并支配:

1）同步的模型

2）进程指名方法

3）消息结构

本章依次研究这三个问题，然后讨论各种语言（包括Ada和occam2）的消息传递模型以及实时POSIX模型。Java不显式支持消息传递模型，然而能够产生实现此模型的类。不过，由于没有引入新的语言特征，本章不讨论Java。

9.1 进程同步

对所有基于消息的系统而言，有一个隐式的同步，即接受者进程不能在消息被发送前得到它。虽然这是十分明显的，却必须同共享变量的使用进行比较，在这种情况下，接受者进程可能读一个变量，而并不知道它是否由发送者进程写过了。如果进程执行无条件消息接受，而当时无消息可用，那么它将被挂起直到消息到达。 283

进程同步模型的变异来自发送操作的语义，可将它粗略分为以下三类:

- **异步（或不等待）**: 发送者立即前进，不管消息是否被接收到。异步发送出现在CONIC（Sloman等，1984）和包括POSIX在内的若干操作系统中。
- **同步**: 发送者只在消息被收到的时候才前进。在CSP（Hoare，1985）和occam2中使用同步发送。
- **远程召用**（remote invocation）: 发送者只在从接收者返回答复的时候才前进。远程召用发送是请求响应式通信的模型，并出现于Ada、SR（Andrews and Olsson，1993）、CONIC（Sloman等，1984）和各种操作系统中。（译者注：本书其他章中将call和invocation都译为“调用”。但在本章中，因多次出现remote procedure call和remote invocation，为避免混淆，将call译为“调用”，将invocation译为“召用”。）

为了了解这些方法之间的差别，考虑下面的类比。信件的邮寄是异步发送——发送者把信件投进邮箱之后就去过自己的日子，只有通过一封回信，发送者才能知道第一封信是否已实际

收到。从接收者的观点看，一封信只能告诉读信人过时的事件，关于发信人的现在情况一点都说不出来。

电话是同步通信的一个更好的类比。发送者一直等待，直到联络上并验明了接收者的身份之后，才把消息送出。如果接收者能立即回答（即在同一次呼叫中），那么这种同步是远程召用。因为发送者和接收者“集合到一起”进行同步通信，所以经常称之为**会合**（rendezvous）。这种远程召用形式被称为**扩展会合**（extended rendezvous），因为在回复送出之前（即在会合期间）可进行任意的计算。

显然，在这三种发送方式之间有一个关系。两个异步事件就能基本上组成一个同步关系，如果总是发送（并等待）一个确认消息的话：

```
        P1                                P2
asyn_send (message)              wait (message)
wait (acknowledgement)           asyn_send (acknowledgement)
```

此外，两个同步通信可用来构造一个远程召用：

```
        P1                                P2
syn_send (message)                   wait (message)
wait (reply)                           ...
                                      construct reply
                                       ...
                                     syn_send (reply)
```

284

因为一个异步发送可用于构造另外两个，可能有人会认为这个模型有最大的灵活性，语言和操作系统应当采纳它。然而，使用这个模型有不少缺点：

1）可能需要无穷多个缓冲区来存储未被读的消息（可能因为接收者已经终止）。

2）因为异步通信是过时的，大多数发送的编程都期望收到确认消息（即同步通信）。

3）异步模型需要较多的通信，所以程序更复杂。

4）更难以证明完整程序的正确性。

注意，在同步消息传递语言中想要进行异步通信的地方，可以很容易地构造缓冲区进程。然而，进程的实现不是没有代价的，所以，缓冲区进程太多可能对系统的整体性能有不利影响。

9.2 进程指名和消息结构

进程指名包括两个有区别的子问题：直接还是间接以及对称性。在直接指名方案中，消息的发送者显式指出接收者的名字：

```
send <消息> to <进程名字>
```

在间接指名方案中，发送者指出某个中间实体（有各种名称：通道、邮箱、链或管道）：

```
send <消息> to <邮箱>
```

注意，即使是对邮箱，消息传递也可以是同步的（即发送者将等待直到消息被读走）。直接指名的优点是简明，而间接指名有助于软件的分解，邮箱可看作程序不同部分之间的接口。

如果发送者和接收者双方互相（直接或间接地）指出名字，那么指名方案就是对称的：

```
send <消息> to <进程名字>
wait <消息> from <进程名字>

send <消息> to <邮箱>
```

```
wait <消息> from <邮箱>
```

如果接收者不指出具体来源，只是从任一进程（或邮箱）接收消息，那就是不对称的：

```
wait <消息>
```
285

不对称指名适合客户机/服务器模式：其中“服务器”进程提供服务以响应从多个“客户机”进程之一来的消息。所以，实现必须有能力支持等待服务器的进程队列。

如果指名是间接的，那么还有进一步的问题要研究，中间实体可以是：

- 多对一结构（即任意多个进程可以写它，而只有一个进程可以读它），这也适合客户机－服务器模式。
- 多对多结构（即多个客户机和多个服务器）。
- 一对一结构（即一个客户机和一个服务器），注意，这种结构不需要运行时支持系统维护队列。
- 一对多结构，当一个进程希望发送一个请求给一组工作者进程，且它并不在意是哪个进程为此请求服务的时候，这种结构就有用了。

消息结构

理想的情况是语言应当允许任何定义的（预定义的或用户的）类型的数据对象都能在消息中传输。实现这种理想是困难的，特别是当数据对象在发送者和接收者中有不同表示的时候，如果表示包括指针，就更困难了（Herlihy and Liskov，1982）。由于这些困难，某些语言（例如occam-1）限制消息内容为系统定义类型的非结构的、固定大小的对象。较现代的语言消除了这些限制，然而，现代操作系统依然要求在传输前将数据转换成字节。

9.3 Ada和occam2的消息传递语义

Ada和occam2都能进行基于消息传递的通信和同步。对于occam2，这是惟一可用的方法；对于Ada，如同在上一章里看到的，通信还可以通过共享变量和保护类型进行。然而，在两个语言纳入基于消息的方案之间有重要的差别。简单地说，Ada使用直接不对称的远程召用，而occam2却包含间接对称的同步消息传递。两个语言都使消息有灵活的结构。现在将描述两个语言。先描述occam2，因为它有更简单的语义。

9.3.1 occam2模型

occam2进程是没有名字的，所以在通信中必须使用经由**通道**的间接指名。每个通道只能由一个写者和一个读者进程使用。两个进程都指出这个通道的名字，语法很简洁： 286

```
ch ! X          -- 写表达式X的值到通道ch

ch ? Y          -- 从通道ch读到变量Y
```

上面的代码中，变量Y和表达式X是同一类型。通信是同步的，所以先访问这个通道的进程将被挂起。当另一个进程到达时，数据将从X传给Y（这可被看作分布式赋值Y:=X)。然后两个进程并发而且独立地继续它们的执行。为说明这种通信，考虑在它们之间传送1 000个整数的两个进程：

```
CHAN OF INT ch:
PAR
  INT V:
```

```
SEQ i = 0 FOR 1000                    --- 进程1
  SEQ
    -- 生成值V
    ch ! V
INT C:
SEQ i = 0 FOR 1000                    --- 进程2
  SEQ
    Ch ? C
    -- 使用C
```

对于两个循环的每次迭代，两个进程发生一次会合。

occam2中的通道是有类型的，并可被定义成传送任何合法类型（包括结构类型）的对象。也能定义通道的数组。

重要的是要认识到：通道上的输入和输出操作被认为是基础性的语言原语。它们成为occam2中的五个原语进程中的两个。其他的是SKIP、STOP和赋值（见第3章）。比较起来，Ada中的通信和同步不具有这样一个中心作用。

9.3.2 Ada模型

远程召用的语义同过程调用有许多表面的相似性。数据传送给接收者，接收者执行，然后返回数据。因为这种相似性，Ada以一种同过程、保护子程序和入口的定义兼容的方式支持程序消息的定义。尤其是参数传递模型是相同的（就是说，所有情况都用仅有的一个模型）。

为了使任务接受一个消息，它必须定义一个**入口**（entry）。像前面一样，允许任意多个、
287 任意模式和任意类型的参数。例如：

```
task type Screen_Output (Id : Screen_Identifier) is
  -- 一个任务类型定义
  一个entry Call (Value : Character; X_Coordinate,
                  Y_Coordinate: Integer);
end Screen_Output;

Display: Screen_Output (Tty1);
-- Tty1 的类型是Screen_Identifier

task Time_Server is -- 单个任务定义
  entry Read_Time (Now : out Time);
  entry Set_Time (New_Time : Time);
end Time_Server;
```

入口可定义为私有的，这意味着它只能被位于任务体的任务调用。例如，考虑下面的Telephone_Operator任务类型。它提供了三种服务：一个查询入口，需要订户的名字和地址；另一个查询入口，需要订户的名字和邮政编码；一个故障报告服务，需要故障线路的号码。此任务还有一个由其内部任务使用的私有入口：

```
task type Telephone_Operator is
  entry Directory_Enquiry (Person : in Name; Addr : in Address;
                           Num : out Number);

  entry Directory_Enquiry (Person : in Name;
        Zip : in Postal_Code; Num : out Number);

  entry Report_Fault (Num : Number);
```

```
private
  entry Allocate_Repair_Worker (Num : out Number);
end Telephone_Operator;
```

Ada还提供一个设施，借助它可以定义入口的数组——这被称为**入口族**（entry family）。例如，考虑有7个输入通道的多路器。Ada不是把每个通道表示成单独的入口，而是将它们定义成一个族⊖。

```
type Channel_Number is new Integer range 1 .. 7;
task Multiplexor is
  entry Channels (Channel_Number)  (Data: Input_Data);
end Multiplexor;
```

上面的代码定义了7个入口，它们都有相同的参数规格说明。 288

为调用一个任务（即向它发送一个消息），只需简单地指出接收者任务的名字及其入口的名字（指名是直接的），例如:

```
Display.Call (Char, 10, 20); -- Char 是个字符

Multiplexor.Channels (3)  (D);
-- 3 是入口族的序标
-- D的类型是Input_Data

Time_Server.Read_Time (T); -- T的类型是 Time
```

注意在最后一个例子中，被传送的惟一数据以相反的方向传送给消息自身（通过一个“out”参数）。这可能导致术语上的混乱，所以在Ada中通常不用“消息传递” （message passing）这个术语。“扩展会合”的说法歧义性较小。

从调用任务的观点看，Ada的保护入口和任务入口是等同的。

如果向一个不活动的任务发出了入口调用，就在调用点引发异常`Tasking_Error`。这就使得在一个任务非预期地过早终止时能够采取一个替代的动作，如下所示。

```
begin
  Display.Call (C, I, J);
exception
  when Tasking_Error => -- 登录出错并继续
end;
```

注意这不等价于一个任务是否可用的预先检查:

```
if Display' Terminated then
  -- 登录出错并继续
else
  Display.Call (C, I, J);
end if;
```

穿插执行可能引起一个任务在属性已经求值之后、调用被处理前终止。

接收一个消息包括接受适当入口的调用:

```
accept Call (C: Character; I, J : Integer) do
  Local_Array (I, J) : = C;
end Call;
```

⊖ Ada还允许保护入口族和保护私有入口族。

```
accept Read_Time (Now : out Time) do
  Now := Clock;
end Read_Time;

accept Channels (3)  (Data: Input_Data) do
  -- 保存从入口族中第3个通道来的数据
end Channels;
```

289 accept语句必须是在任务体（而不是被调用的过程）里出现，但它可被放在任何其他语句是合法的地方。它甚至可被放在另一个接受语句里（虽然不是同一个入口的）。所有入口（和族成员）应当有与之关联的接受语句。这些接受语句指名相关的入口，但不指名发出调用的任务。因此，指名是不对称的。

为了给出两个任务进行交互的简单例子，像occam2的例子一样，考虑两个循环往复并在它们之间传送数据的任务。在Ada代码中，这些任务交换数据。

```
procedure Test is

  Number_Of_Exchanges : constant Integer := 1000;

  task T1 is
    entry Exchange (I : Integer; J : out Integer);
  end T1;

  task T2;

  task body T1 is
    A, B : Integer;
  begin
    for K in 1 .. Number_Of_Exchanges loop
      -- 生产 A
      accept Exchange (I : Integer; J : out Integer) do
        J := A;
        B := I;
      end Exchange;
      -- 消费 B
    end loop;
  end T1;

  task body T2 is
    C, D : Integer;
  begin
    for K in 1 .. Number_Of_Exchanges loop
      -- 生产 C
      T1.Exchange (C, D);
      -- 消费D
    end loop;
  end T2;

begin
  null;
end Test;
```

虽然两个任务（T1和T2）之间的关系本质上是对称的，但Ada的不对称指名要求它们有十分不同的形式。应当将这一点与occam2代码进行比较，那里保持了对称性。 290

9.3.3 异常处理和会合

由于在会合期间可以执行任何合法的Ada代码，当然就有在accept语句里面引发异常的可能性。如果出现这种情况，那么，或者

- 在accept语句（或是作为accept语句的一部分，或是在accept语句里面的嵌套块里）里有一个合法的异常处理程序，这种情况下accept语句可正常终止；或者
- 在accept语句里面引发的异常未被处理，接受语句立即终止。

在后一种情况，指名的异常将在被调用任务和调用任务中再次被引发。被调用的任务将在接受语句后面立即引发该异常，调用任务将在入口调用之后引发它。然而，作用域问题可能使得该异常在调用任务中成为无名的。

为说明会合和异常模型之间的交互，考虑一个作为文件服务器的任务，其入口之一允许客户任务打开一个文件:

```
task File_Handler is
  entry Open (F : File_Type);
  ...
end File_Handler;

task body File_Handler is
  ...
begin
  loop
    begin
      ...
      accept Open (F : File_Type) do
        loop
          begin
            Device_Open (F);
            return;
          exception
            when Device_Off__Line =>
              Boot_Device;
          end;
        end loop;
      end Open;
      ...
    exception
      when File_Does_Not_Exist =>
        null;
    end;
  end loop;
end File_Handler;
```

291

在这段代码中，File_Handler调用一个设备驱动程序打开指定的文件。这个请求或是成功，或是导致引发两个异常Device_off_Line或File_Does_not_Exist之一。第一个异常在接受语句中被处理：尝试去引导这个设备，然后重复打开请求。由于该异常处理程序是在接受语句里面，客户任务意识不到这种活动（虽然如果设备拒绝引导，它将会被永远挂起）。第二个异常归因于用户任务的错误请求。所以在该接受语句中未被处理，并被传播到

调用任务，这个任务需要保护自己以防这个异常发生。

```
begin
  File_Handler.Open (New_File);
exception
  when File_Does_Not_Exist =>
    File_Handler.Create (New_File);
    File_Handler.Open (New_File);
end;
```

注意，服务器任务也通过在其外层循环构造中定义一个块来保护自己，以防这个异常发生。

9.4 选择性等待

到目前为止讨论过的所有消息传递形式中，消息的接收者必须等待，一直到指定的进程或通道交付通信。一般来说，这太受限制了。接受者进程实际上可能希望等待调用它的许多进程中的任意一个。服务器进程接收来自许多客户的请求消息，客户调用的次序对服务器而言是未知的。为方便这种常用程序结构，允许接收者进程选择性地等待多个可能的消息。但是，为了完整理解选择性等待，必须首先解释Dijkstra的守备命令（Dijkstra，1975）。

守备命令（guarded command）是一种仅在其守备求值为TRUE时才执行的命令。例如：

```
x < y -> m := x
```

这意味着，如果x小于y，则将x的值赋给m。守备命令本身不是一个语句，而是**守备命令集合**的一个成分。在这里，关心的只是选择（或备选）构造：

```
if   x <= y -> m := x
□    x >= y -> m := y
fi
```

□表示选择。在上面的例子中，程序的执行或者是将x赋给m，或者是将y赋给m。如果两个备
292 选都是可能的，即两个守备都为真（此例中是x=y），则进行一个*任意的*选择。程序员不能确定取哪条路径，所以，这种构造是**不确定性的**（non-deterministic）。一个构造良好的程序对所有可能的选择都是合法的。在此例中，当x=y时，两条路径都有相同效果。

重要的是要注意到这种不确定性结构十分不同于用正规if语句构造出来的确定性形式：

```
if    x <= y then m := x;
elsif  x >= y then m := y;
end if;
```

这里x=y将确保将x的值赋给m。

一般选择结构可以有任意多个守备成分。如果有一个以上的守备求值为TRUE，则选择是任意的。但如果没有一个守备求值为TRUE，则被看成一个出错状态，语句和执行它的进程将被中止。

守备命令是一种通用程序结构。然而，如果被守备的命令是消息操作符（正常的是接收操作符，虽然在某些语言中还有发送），这种语句就称为**选择性等待**（selective waiting）。它是在CSP（Hoare，1978）中首先引进的，在Ada和occam2中都提供了。

9.4.1 occam2的ALT

考虑一个进程，它接连从三个通道（ch1、ch2和ch3）读整数，然后把它接收的整数接

连输出到另一通道（chout）。如果整数到达的顺序就是三个通道的顺序，则一个简单的循环构造就足够了。

```
WHILE TRUE
  SEQ
    ch1     ? I          -- 对某个局部整数 I
    chout   ! I
    ch2     ? I
    chout   ! I
    ch3     ? I
    chout   ! I
```

然而，如果到达的次序是未知的，则进程每循环一次就必须在三个备选中进行一次选择：

```
WHILE TRUE
  ALT
    ch1     ? I
      chout    ! I
    ch2     ? I
      chout    ! I
    ch3     ? I
      chout    ! I
```

如果在ch1、ch2或ch3上有一个整数，它将被读出，并执行规定的动作。在这里总是输出最 293
近获得的整数到输出通道chout。在那种有一个以上的输入通道准备通信的情况，对读哪个通道做一个任意的选择。在研究无任何通道就绪的ALT语句的行为之前，先概略给出ALT语句的一般结构。它由一组守备进程组成：

```
ALT
  G1
    P1
  G2
    P2
  G3
    P3
    :
  Gn
    Pn
```

这些进程自身不受限制——它们是任何occam2进程。守备（它们也是进程）可取三种形式之一（第四种可能性包括时间延迟的规格说明，在第12章研究）。

```
<布尔表达式> & 通道输入操作

通道输入操作

<布尔表达式> & SKIP
```

所以最一般形式是一个布尔表达式和一个通道读，例如

```
NOT BufferFull & ch ? BUFFER[TOP]
```

如果布尔表达式就是TRUE，则可被完全省略（如在前面例子中那样）。守备的SKIP形式用于规定在其他备选动作被阻止的时候采取的某种备选动作，例如：

```
ALT
```

```
NOT BufferFull & ch ? BUFFER[TOP]
  SEQ
    TOP:= …
BufferFull & SKIP
  SEQ
    -- 交换缓冲区
```

在ALT语句执行时，布尔表达式被求值。如果没有一个为TRUE（并且没有默认的TRUE备选），
294 则ALT进程不能前进，并变成等价于STOP（出错）进程。假设ALT正确执行了，这些通道被检查，看看是否有进程正在等待向它们写。可能发生下列可能之一：

1）只有一个就绪备选，即有一个布尔表达式求值为真（连同一个等待写的进程或一个SKIP守备）——这个备选被选中，会合发生（如果不是SKIP的话），执行相关的子进程。

2）有多于一个的就绪备选——任意选中一个，它可能是SKIP备选，如果它出现并就绪的话。

3）无就绪备选——ALT被挂起，直到某个别的进程向ALT的一个开放通道写入。

所以，如果所有布尔表达式求值为FALSE，则ALT变成STOP进程，但如果没有未完成的调用，它将只被挂起。因为occam2不是共享变量模型，对任何其他进程而言，不可能改变布尔表达式中任何成分的值。

ALT同SEQ、IF、WHILE、CASE和PAR的组合提供occam2程序构造的完整集合。重复器可以用一种和联系其他构造相同的方式同ALT联系起来。例如，考虑一个连接器进程，它从20个进程读取（而不是前面说的3个），然而，服务器进程不是使用20个不同的通道，而是像下面这样使用通道数组：

```
WHILE TRUE
  ALT j = 0 FOR 20
    Ch[j] ? I
      Chout ! I
```

最后，应当注意到occam2提供ALT的一种变体形式，它不是任意地选择就绪的备选。如果程序员希望给一个特定通道以优先，那就应当把它放在PRI ALT的第一个成分上。PRI ALT的语义指出：文本上的第一个就绪备选被选中。下面是PRI ALT语句的例子。

```
PRI ALT
  VeryImportantChannel ? message
    -- 动作
  ImportantChannel ? message
    -- 动作
  LessImportantChannel ? message
    -- 动作

WHILE TRUE
  PRI ALT j = 0 FOR 20    -- ch[0]被给以最高优先
    ch[j] ? I
      chout ! I
```

295 使用PRI ALT的例子在11.3.1节给出。

有界缓冲区

occam2提供无共享变量的通信原语，所以，像缓冲区这样的资源控制器必须被实现为服务器进程（见7.2.1节）。为实现一个单一读者和单一写者的缓冲区，需要使用两个通道将缓冲

区进程和客户进程连接起来（对更多读者和写者的情况需要通道数组）:

```
CHAN OF Data Take, Append:
```

令人遗憾的是，这个缓冲区的自然形式会是:

```
VAL INT Size IS 32:
INT Top, Base, NumberInBuffer:
[Size]Data Buffer:
SEQ
  NumberInBuffer := 0
  Top := 0
  Base := 0
  WHILE TRUE
    ALT
      NumberInBuffer < Size & Append ? Buffer [Top]
        SEQ
          NumberInBuffer  := NumberInBuffer + 1
          Top := (Top + 1) REM Size
      NumberInBuffer > 0 & Take ! Buffer[Base]  -- 非法occam 代码
        SEQ
          NumberInBuffer : = NumberInBuffer - 1
          Base := (Base + 1) REM Size
```

occam2不允许在此上下文中有输出操作。只有输入操作可作为ALT守备的一部分。这种限制的理由是分布式系统的实现效率。问题的本质是对称守备的规定可能导致ALT在两端访问一个通道。所以，一个ALT的任意决定会依赖于另一个的决定（反之亦然）。如果ALT是在不同处理器上，那么集体决定的协议会涉及许多低层协议消息的传送。

为了突破对守备的限制，occam2迫使Take操作被编制成一个双重的交互。首先是客户进程必须指出它愿意进行TAKE操作，然后它必须Take，因此需要第三个通道:

```
CHAN OF Data Take, Append, Request:
```

客户必须发出下列调用

```
SEQ
  Request ! ANY         -- ANY是一任意语言要素
  Take ? D              -- D是DATA类型的
```

缓冲区进程本身有下列形式: 296

```
VAL INT Size IS 32:
INT Top, Base, NumberInBuffer:
[Size]Data Buffer:
SEQ
  NumberInBuffer := 0
  Top := 0
  Base := 0
  Data ANY:
  WHILE TRUE
    ALT
      NumberInBuffer < Size & Append ? Buffer[Top]
        SEQ
          NumberInBuffer  := NumberInBuffer + 1
```

```
      Top := (Top + 1) REM Size
  NumberInBuffer > 0 & Request ? ANY
    SEQ
      Take ! Buffer[Base]
      NumberInBuffer : = NumberInBuffer - 1
      Base := (Base + 1) REM Size
```

因此，缓冲区的正确功能发挥依赖于客户进程的正确使用。这种依赖性是不良模块性的反映。虽然Ada选择语句也是不对称的（即你不能在接受语句和入口调用之间选择），但数据可按相反方向传递给调用这一事实，排除了occam2中表明的困难。

9.4.2 Ada的select语句

Ada的多对一消息传递关系能够用以容易地处理多个客户都调用同一入口的情况。然而，当一个服务器必须处理对两个或两个以上不同入口的可能调用时，也需要一个ALT类型的结构。在Ada中把这称为**select语句**。为说明的需要，考虑一个服务器任务，它经由入口S1和S2提供两种服务。下面的结构通常就足够了（即循环包含一个提供两个服务的选择语句）：

297

```
task Server is
  entry S1 (...);
  entry S2 (...);
end Server;

task body Server is
  ...
begin
  loop
    -- 准备服务
    select
      accept S1 (...) do
        -- 这项服务的代码
      end S1;
    or
      accept S2 (...) do
        -- 这项服务的代码
      end S2;
    end select;
    -- 干点清理工作
  end loop;
end Server;
```

在循环的每次执行中，将执行其中的一个接受语句。

像第一个occam2的例子一样，这个Ada程序没有说明布尔表达式在守备中的使用。Ada选择语句的一般形式是：

```
select
  when <布尔表达式> =>
    accept <入口> do
      ..
    end <入口>;
    -- 任何语句序列
or
```

```
    -- 类似
    ...
  end select;
```

可以有任意多个备选。除了接受备选（是必须至少要有一个的）之外，还有三种其他形式（它们不能在同一语句中混用）：

1）一个terminate（终止）备选

2）一个else（否则）备选

3）一个delay（延迟）备选

延迟备选在第12章中将同occam2的等价物一同研究。else备选被定义为当（且仅当）没有其他备选立即可执行的时候执行。只有当入口处没有任何未定的调用且入口的布尔表达式求值为True（或根本没有布尔表达式）时，这才会发生。

终止备选在occam2中没有等价物，但却是一个重要原语。它有下列性质：

1）如果它被选中，执行选择语句的任务被终了化并终止。

2）它只能在不再有任何任务可以调用选择语句时才可被选中。

说得更精确些：一个任务将终止，如果依赖于同一主宰（master）的所有任务已经终止或者也相似地在选择语句的终止备选上等待。此备选的效果是能将服务器任务构造得不需要关心它们自己的终止问题，但它们不再被需要时将肯定终止。occam2缺乏这种规定，导致复杂的终止状态且有相关的死锁问题（Burns，1988）。 298

选择语句的执行遵循与ALT类似的顺序。首先将布尔表达式求值，那些取假值的表达式使得在选择语句执行时关闭那些备选。按这种方式，派生出了一个可能的备选集合。若该集合为空，则在选择语句后面立刻引发异常Program_error。对正常的执行，选中一个备选。如果有一个以上的备选具有未完成的调用，则选择是任意的。如果对于适当的备选没有未完成的调用，那么或者

- 执行else备选，如果有一个的话，或者
- 任务被挂起等待发出调用（或者超时到期——见第12章），或者
- 任务被终止，如果有terminate备选并且没有其他任务会调用它（如上所述）。

注意在守备中可以包含共享变量，但不推荐这样做，因为直到这个守备被再次求值的时候才会注意到对它们的改变。还有，实时系统附件允许按文本顺序对选择语句的枝干赋以优先级。这样就有了occam2的PRI ALT的同样功能。作为Ada选择语句的最后例子，考虑9.3.2节所给的Telephone_Operator的任务体。

```
task body Telephone_Operator is
  Workers : constant Integer := 10;
  Failed : Number;
  task type Repair_Worker;
  Work_Force : array (1 .. Workers) of Repair_Worker;
  task body Repair_Worker is ...;
  ...
begin
  loop
    -- 准备接受下一个请求
    select
      accept Directory_Enquiry (Person : in Name;
```

```
            Addr : in Address; Num : out Number) do
        -- 查电话号码并将值赋给Num
      end Directory_Enquiry;
    or
      accept Directory_Enquiry (Person : in Name;
            Zip : in Postal_Code; Num : out Number) do
        -- 查电话号码并将值赋给Num
      end Directory_Enquiry;
    or
      accept Report_Fault (Num : Number) do
        Failed := Num;
      end Report_Fault;
      -- 存放有故障的号码
    or
      when Unallocated_Faults =>
        accept Allocate_Repair _Worker (Num : out Number) do
          -- 取下一个故障号码
          Num := ...;
        end Allocate_Repair_Worker;
        -- 更新失败回收号码的记录
    or
      terminate;
    end select;
    ...
  end loop;
end Telephone_Operator;
```

299

局部任务类型Repair_Worker负责修理登记的线路故障。它们通过私有入口Allocate_Repair_Worker同Telephone_Operator通信。为确保该worker任务不连续地同Telephone_Operator通信，接受语句是带守备的。还要注意，Telephone_Operator在会合之外做尽可能多的工作，以使客户任务能尽可能快地继续下去。

客户任务（见12.4.2节）也可使用Ada选择语句且Ada选择语句也可处理异步事件（见10.8节）。

9.4.3 不确定性、选择性等待和同步原语

在上述讨论中，注意到了当在一个选择性等待构造中有一个以上的就绪备选时，在它们之间的选择是任意的。这种任意选择背后的基本理由是并发语言通常对进程执行的顺序不做什么假设。假设调度程序对进程进行不确定性的调度（不过具体的调度程序会有确定性行为）。

为说明这种关系，考虑将执行一个选择性等待构造的进程P，在此构造上可能调用进程S和T。如果假定调度程序的行为是不确定性的，那么这个程序有多种可能的穿插或“历史”：

1）P先运行，它被阻塞在选择语句上。然后S（或T）运行，并同P会合。

2）S（或T）先运行，并被阻塞在对P的调用上；然后P运行，并执行选择语句，其结果是发生同S（或T）的会合。

3）S（或T）先运行，并被阻塞在对P的调用上；然后T（或S）运行，也被阻塞在P上。最后，P运行，并执行T和S正在等待的选择语句。

这三种可能而合法的穿插执行导致P在选择性等待上没有、有一个或有两个未完成的调用。

选择语句的定义是绝对精确的，因为假设调度程序是不确定性的。如果P、S和T可以按任何次序执行，那么在情况（3），P应当能选择与S或T会合——它不影响程序的正确性。 300

类似的争论适用于由同步原语定义的任何队列。不确定性调度意味着所有这种队列应当以不确定的顺序释放进程。虽然信号量队列经常以这种方式定义，入口队列和管程队列经常被规定为FIFO式的。这里的基本理由是FIFO队列禁止饥饿。然而，这个争论是谬误的，如果调度程序是不确定性的，那么饥饿就能够发生（一个进程永远不被分配给处理器）。让同步原语去试图预防饥饿是不适合的。对于入口队列也应该是不确定性的做法，也是有争议的。

赋有优先级的进程调度问题在第13章中详细研究。

9.5 POSIX消息

POSIX通过消息队列的概念支持异步的、间接消息传递。一个消息队列可以有多个读者和多个写者。可为每个消息关联上优先级（见13.14.2节）。

消息队列的真正意图是（可能是分布式的）用于进程之间的通信。然而，没有什么禁止在同一进程的线程之间使用它们，虽然对这一点使用共享存储器和互斥锁会更有效（见8.6.3节）。

所有消息都有一些属性指出队列的最大长度、队列中每个消息的最大长度、当前排队的消息数等等。当创建队列时，有一个属性对象用于设置队列属性。队列的属性由函数`mq_getattr`和`mq_setattr`进行处理（这些函数处理属性本身，而不是属性对象，这不同于线程或互斥锁属性）。

消息队列在创建时被命名（类似于文件名字，但不一定在文件系统中表示出来）。为得到对队列的访问，就只要求用户进程去`mq_open`（打开）关联的名字。像所有Unix文件系统一样，`mq_open`用于创建并打开一个已经存在的队列（还有相应的`mq_close`和`mq_unlink`例程）。

发送和接收消息是通过`mq_send`和`mq_receive`例程完成的。从一个字符缓冲区读写数据。如果缓冲区是满的或空的，则发送进程/接收进程被阻塞，除非为此队列设置了属性`O_NONBLOCK`（这种情况给出一个错误返回）。在一个消息队列解除阻塞时，如果有发送者和接收者正在等待，没有规定唤醒哪一个，除非规定了优先级调度选项。如果进程是多线程的，每个线程都有权成为潜在的发送者/接收者。

进程还能指示：当一个空队列接收一个消息而没有等待的接收者时，应当发送一个信号给它。按这种方式，进程就能在一个和多个消息队列上等待信息到达的同时继续执行。进程还可能等待一个信号的到达。这样就能实现选择性等待的等价结构。 301

程序9-1概述了POSIX消息传递设施的一个典型C接口。

程序9-1 POSIX消息队列的C接口

```
typedef ... mqd_t;
typeder ... mode_t;
typedef ... size_t;
typedef ... ssize_t;

struct mq_attr {
  ...
  long mq_flags;
  long mq_maxmsg;
  long mq_msgsize;
```

```
  long mq_curmsg;
  ...
};

#define O_CREAT ...
#define O_EXCL ...
#define O_RDONLY ...

int mq_getattr (mqd_t mq, struct mq_attr *attrbuf);
  /* 获取同 mq 关联的当前属性 */
int mq_setattr (mqd_t mq, const struct mq_attr *new_attrs,
                struct mq attr *old_attrs);
  /* 设置同 mq 关联的当前属性 */

mqd_t mq_open (const char *mq_name, int oflags, mode_t mode,
               struct mq_attr *mq_attr);
  /* 打开/创建指名的消息队列 */

int mq_close (mqd_t mq);
  /* 关闭消息队列 */

int mq_unlink (const char *mq_name);

ssize_t mq_receive (mqd_t mq, char *msq_buffer,
                    size_t buflen, unsigned int *msgprio);
  /* 取来队列中的下一个消息，并将它存放在*/
  /* 由msq_buffer所指的区域里; */
  /* 返回消息的实际大小*/
ssize_t mq_receive (mqd_t mq, char *msq_buffer,
                    size_t buflen, unsigned int *msgprio,
                    const struct timespec *abs_timeout);
  /* 像对 mq_receive 一样，但有一个超时*/
  /* 如果超时到期，返回ETIMEDOUT */

int mq_send (mqd_t mq, const char *msq,
                    size_t msglen, unsigned int msgprio);
  /* 发送由 msq 所指向的消息 */

int mq_timedsend (mqd_t mq, const char *msq,
                  size_t msglen, unsigned int msgprio,
                  const struct timespec *abs_timeout);
  /* 发送由msq所指向的消息，带一个超时 */
  /* 如果超时到期，返回ETIMEDOUT */

int mq_notify (mqd_t mq, const struct sigevent *notification);
  /* 请求当有一个消息到达一个空消息队列，而且没有等待的接收者时，*/
  /* 向调用进程发一个信号*/

/* 所有上述整型函数如果成功，返回0，否则返回-1. */
/* 当任何上述函数返回一个出错状态时，*/
/* 共享变量errno 包含出错的原因 */
```

为了说明消息队列的使用，将第7章和第8章讨论过的简单机器人手臂用一个父进程分叉出三个controller粗略勾画出来。这一次，父进程同控制器通信以把手臂的新位置传送给它们。首先给出controller的代码。在这里假设MQ_OPEN和FORK是两个函数，它们测试来自

mq_open和fork系统调用的出错返回，并仅在调用成功时返回。

```
typedef enum {xplane, yplanc, zplane} dimension;

void move_arm (dimension D, int P);

# define DEFAULT_NBYTES 4
/* 假设坐标可用4个字符表示 */
int nbytes = DEFAULT_NBYTES;

#define MQ_XPLANE "/mq_xplane"    /* 消息队列名字 */
#define MQ_YPLANE "/mq_yplane"    /* 消息队列名字 */
#define MQ ZPLANE "/mq_zplane"    /* 消息队列名字 */
#define MODE ...                   /* mq_open的模式信息*/
/*消息队列的名字*/

void controller (dimension dim)
{
  int position, setting;
  mqd_t my_queue;
  struct mq_attr ma;
  char buf [DEFAULT_NBYTES];
  ssize_t len;

  position = 0;
  switch (dim) { /* 打开合适的消息队列*/
    case xplane:
      my_queue = MQ_OPEN (MQ XPLANE, O_RDONLY, MODE, &ma);
      break;
    case yplane:
      my_queue = MQ_OPEN (MQ YPLANE, O_RDONLY, MODE, &ma);
      break;
    case zplane:
      my_queue = MQ_OPEN (MQ_ZPLANE, O_RDONLY, MODE, &ma);
      break;
    default:
      return;
    };

  while (1) {
    /* 读消息 */
    len = MQ_RECEIVE (my_queue, &buf [0], nbytes, NULL);
    setting = * ( (int*)  (&buf [0]) );
    position = position + setting;
    move_arm (dim, position);
  };
}
```

现在可以给出主程序了，它创建控制器进程，并向它们传递合适的坐标:

```
int main (int argc, char **argv) {
  mqd_t mq_xplane, mq_yplane, mq_zplane;
    /* 每个进程一个队列 */
  struct mq__attr ma; /* 队列属性 */
  int xpid, ypid, zpid;
```

302~303

```
    char buf[DEFAULT_NBYTES];

      /* 设置所需的消息队列属性 */
      ma.mq_flags = 0;     /* 无特别行为 */
      ma.mq_maxmsg = 1;
      ma.mq_msgsize = nbytes;

      /* 设置这三个消息队列的实际属性的调用*/

      mq_xplane = MQ_OPEN (MQ_XPLANE, O_CREAT|O_EXCL, MODE, &ma);
      mq_yplane = MQ_OPEN (MQ_YPLANE, O_CREAT|O_EXCL, MODE, &ma);
      mq_zplane = MQ_OPEN (MQ_ZPLANE, O_CREAT|O_EXCL, MODE, &ma);

      /* 复制该进程，得到三个控制器*/
      switch (xpid = FORK () ) {
          case 0:       /* 子 */
            controller (xplane);
            exit (0);
          default:      /* 父 */
            switch (ypid = FORK () ) {
              case 0:       /* 子 */
                controller (yplane);
                exit (0);
              default:      /* 父 */
                switch (zpid = FORK () ) {
              case 0:       /*  子 */
                controller (zplane);
                exit (0);
              default:       /* 父 */
                 break;
          }
        }
    }

    while (1) {
      /* 寻找新位置，并设置缓冲区以把每个坐标传送给控制器，例如*/
      MQ_SEND (mq_xplane, &buf[0], nbytes, 0);
    }
  }
```

304

9.6 CHILL语言

本节给出CHILL中并发模型的一个简单描述。CHILL的其他设施，例如异常处理模型已经讨论过了。

CHILL的开发沿着一条与Ada相似的路线。然而，其预定应用领域更受限制——电信交换系统。虽然有这种限制，这种系统具有一般实时应用的所有特征：它们大而且复杂，有规定的时间约束和高可靠性需求。在20世纪70年代早期，CCITT（Comité Casullatif International Télégraphique et Téléphonique）认识到需要单独的高级语言，以使得电信系统更独立于硬件制造商。到1973年认定现有语言都不满足其需求，并成立一个小组提出初始语言建议。这个语言在几个试验性实现上使用，在多次设计迭代后，于1979年秋商定了一个最终语言建议。名字CHILL是从国际电信委员会派生的（Ccitt HIgh Level Language）。

这里对CHILL并发模型的兴趣来自其实用性设计。进程只能在最外层声明，并可以在它们启动时传递参数。考虑前面给出的简单机器人手臂控制器。进程类型control有一个参数是运动的维（即x、y或z平面）。进程的动作是为其动作平面读一个新的位置，然后让机器人手臂运动：

```
newmode dimension = set (xplane, yplane, zplane);
  /* 枚举类型 */

control = process (dim dimension);
  dcl position, setting int;  /* 声明变量position和setting为整数 */
  position := 0;
  do for ever;
    new_setting (dim, setting);
    position := position + setting;
    move_arm (dim, position);
  od;
end control;
```

305

为指出三个控制过程的执行（每维一个），使用启动语句：

```
start control (xplane);
start control (yplane);
start control (zplane);
```

这将使三个无名进程开始它们的执行。如果想要标识进程的每个实例，则给出独特的名字作为启动语句的一部分（名字是*instance*类型的变量）。

```
dcl xinst, yinst, zinst instance;

start control (xplane) set xinst;
start control (yplane) set yinst;
start control (zplane) set zinst;
```

进程实例通过执行stop语句终止自己。

CHILL中的通信

正是CHILL的通信和同步机制才使它被看成是实用的。它不是有一个简单的模型，而是支持三种不同的方法：

1）regions（区）——这些提供共享变量的互斥（即，它们是管程）
2）buffers（缓冲区）——使进程之间进行异步通信
3）signals（信号）——一种可与occam机制相比较的通道形式。

设计这三种不同结构的动机看来是因为认为不可能有单个模型在所有情况下都是最好的：

> *一种通信机制不会在分布式结构和共享存储体系结构两种情况中都是最优的*（Smedema等，1983）。

306

区结构显式地允许对其过程的访问，在区内events（事件）可用来延迟进程。事件执行delay和continue操作。这些同Modula-1中的wait和signal有相似的语义。下面的代码实现了一个简单的资源控制器：

```
resource : region
  grant allocate, deallocate;
```

```
    syn max = 10;                       /* 声明常量 */
    decl used int := 0;
    no_resources event;

    allocate : proc;
      if used < max then
        used := used + 1;
      else
        delay (no_resource);
      fi;
    end allocate;

    deallocate : proc
      used := used - 1;
      continue (no_resource)
    end deallocate;

  end resource;
```

缓冲区被预定义有send（put）和receive（get）操作符:

```
  syn size = 32;
  dcl buf buffer (size) int;        /* 声明整型缓冲区 */
  ...
    send buf (I);                   /* I 是整数 */
  ...
    receive case
       (buf in J) :                 /* 语句*/
    esac
```

接收操作放在一个case语句中，因为一般来说，进程可能尝试从一个以上的缓冲区读。如果所有缓冲区是空的，则进程被延迟。进程还有可能从缓冲区读，以知道在缓冲区放置数据的那个进程的名字。这在电信系统中尤为重要，那里必须建立消费者进程和生产者进程之间的链路。

信号是根据进行通信的数据的类型和目的进程（类型）定义的:

```
signal channel = (int) to consumer;
```

307 这里，consumer是一个进程类型。为经由这个信号传输数据（I），再次使用send语句:

```
send channel (I) to con;
```

其中con是类型consumer的一个进程实例。接收操作使用case语句，以提供选择性等待(无守备)。

研究一下应用于CHILL信号的指名约定是有趣的。发送操作指出信号和进程实例的名字，接收操作只提到信号的名字，但它可以找到相应进程的身份。

9.7 远程过程调用

目前为止，本章集中讨论过程怎么通信并同步它们的活动。第14章讨论有关当进程驻留在由网络连接起来的不同机器上的实现。然而，为了完备性，这里介绍**远程过程调用**（RPC, remote procedure call）的概念，因为这是在分布式环境中传送控制的常用方法。

在单处理器系统中，进程执行过程，以将控制从一段代码移送给另一段，只在它们需要和另一个进程通信和同步时，它们才要求进程间通信。远程过程调用将这种思想扩展，以使一个进程可以执行驻留在多台机器上的代码。它们允许在一台处理器上执行的一个进程去执行另一处理器上的过程，对于这些是不是应当对应用程序员透明，是有争议的，并将在第14章讨论。过程的执行可能涉及同驻留在远程机器上的进程的通信，这是由共享变量方法（例如管程）或是通过消息传递（例如，会合）实现的。

值得注意的是，远程召用消息传递可被做得语法上看起来像过程调用，消息被编码为输入参数，返回消息被编码为输出参数（例如，Ada的入口和接收语句）。然而，这种语法的便利可能会误导，因为远程召用消息传递的语义与过程调用十分不同。特别是，远程召用在执行显式操作方面依赖于接受者进程的主动合作。另一方面，过程调用不是进程间通信的形式，而是将控制移送给一段被动的代码。当过程是本地的（调用者和过程在同一台机器上），它们的体可由调用进程执行；在远程过程调用的情况下，这个体可能必须由在远程机器上的无名代理进程代表调用者执行。在这种情况下，另一个进程的卷入是一个实现问题，而不是语义问题。为使远程过程调用具有与（可重入）本地调用相同的语义，实现必须允许并发处理任意多个调用。这要求为每个调用创建一个进程/线程，或有足够大的进程/线程池以处理最多数量的并发调用。进程创建或维护的开销有时可能支配被限制的并发程度。 308

小结

Ada和POSIX提供了消息传递设施，它们还支持另外的通信模式。然而，在occam2中，进程通信的惟一方法是消息传递。

基于消息通信的语义由三个问题定义：

- 同步模型
- 进程指名方法
- 消息结构

进程同步模型的多样性来自发送操作的语义。有三大类：

- 异步——发送者不等待
- 同步——发送者进程等待消息被读走
- 远程召用——发送者进程等待消息被读走、被作用并生成回复

远程召用可被做得语法同过程调用相似。当在分布式系统中使用远程过程调用（RPC）时可能引起混淆。然而，RPC是一个实现策略，远程召用定义具体消息传递模型的语义。在这种通信中涉及的两个进程可能是在同一处理器上，也可能是分布式的，但语义是相同的。

进程指名包括两个不同的问题：直接或间接的和对称性。Ada使用直接不对称指名的远程召用。相反，occam2采用一个同步间接对称的方案。消息可用任何系统定义或用户定义类型的形式。POSIX支持异步对称方案。

对于occam2中两个进程的通信，要求定义（适当类型协议的）通道，并使用两个通道运算符？和！。Ada中的通信要求一个任务定义入口，并在其任务体内接受任何到来的调用。当一个任务调用另一个任务中的入口并被接受时，就发生会合。在POSIX中，通信是通过具有发送/接受原语的消息队列进行的。occam2支持一对一通信机制，Ada是多对一机制，而POSIX是多对多机制。

为了提高基于消息的并发编程语言的表达能力，必须允许进程在多个备选通信中进行选择。支持这种设施的语言原语被称为**选择性等待**。这里，进程可从不同备选中选择，对于任
309 何特定执行，在这些备选中的某些备选可以使用布尔守备。occam2的构造称为`ALT`，允许进程在任意多个守备的接收操作之间选择。Ada提供相似的语言特征（选择语句）。然而，它有两个额外的设施：

1）选择语句可以有一个else部分，如果在开备选上没有未完成的调用，就执行它。

2）选择语句可以有一个终止备选，它将引起执行该选择语句的进程终止，如果没有任何可以调用它的任务仍可执行的话。

在POSIX中，一个进程或线程能够指出，当消息队列为满或空时它不准备阻塞。通知机制被用于使操作系统能向一个进程发送它何时能够继续的信号。这种机制可被用于在一个或多个消息队列上等待消息。

Ada的扩展会合的一个重要特征是它同异常处理模型的合作。当一个异常被引发，但在会合中未被处理时，它就被传播给调用任务和被调用任务双方。所以，调用任务必须保护自己以防止正在被生成的无名异常（作为发出消息传递调用的结果）。

为了给出语言设计的进一步例子，给出了CHILL的概述。CHILL有一个不同寻常的实用设计，包括：

- 进程的数据初始化
- 进程名字
- 管程（称为区）
- 缓冲区（允许异步通信）
- 信号（同步通道机制）

相关阅读材料

Andrews, G. A. (1991) *Concurrent Programming Principles and Practice*. Redwood City, CA: Benjamin/Cummings.

Burns, A. (1988) *Programming in occam2*. Reading, Addison-Wesley.

Burns, A. and Davies, G. (1993) *Concurrent Programming*. Reading: Addison-Wesley.

Burns, A. and Wellings, A. J. (1995) *Concurrency in Ada*. Cambridge: Cambridge University Press.

Hoare, C. A. R. (1985) *Communicating Sequential Processes*. Englewood Cliffs, NJ: Prentice Hall.

310 Galletly, J. (1990) *Occam2*. London: Pitman.

Silberschatz, A. and Galvin, P. A. (1998) *Operating System Concepts*. New York: John Wiley & Sons.

练习

9.1 考虑到Ada支持保护对象，Ada应当把会合设施从语言中排除吗？

9.2 如果一个Ada任务有两个入口点，它是否有可能接受等待最长时间的那个调用任务的入口？

9.3 说明如何用Ada任务和会合实现二元信号量。如果一个任务在发出Signal之前，它发出的wait操作被中止，那么会怎么样？

9.4 讨论用会合而不用保护对象实现信号量的优点和缺点。

9.5 说明如何将Ada任务和会合用于实现Hoare的管程。

9.6 图9-1表示occam2中的一个进程。

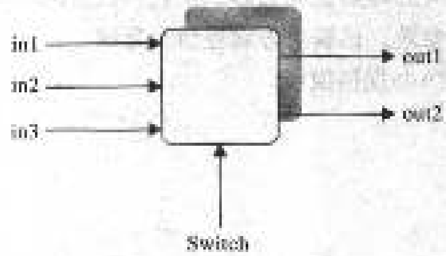


图9-1 occam2中的进程

整数被接连读进通道in1，in2，in3。进程在这些整数到达时取它们。开始时，所有输入整数被输出到通道out1。如果有输入由通道Switch进入，输出就移到通道out2。后续由Switch进入的输入将改变输出通道。写一个实现这个进程的occam2 PROC。

9.7 Ada会合的哪些方面可看作远程过程调用？讨论下列代码，它意在实现一个具体的远程过程。研究被调用过程和调用任务的含义这两个方面。

```
task type Rpc_Implementation is
    entry Rpc (Param1 : Type1; Param2 : Type2);
end Rpc_Implementation;

task body Rpc_Implementation is
begin
    accept Rpc (Param1 : Type1; Param2 : Type2) do
        -- 过程的体
    end Rpc;
end Rpc_Implementation;

-- 声明1000个Rpc_Implementation任务的数组
Concurrent_Rpc : array (1..1000) of Rpc_Implementation;
```

311

9.8 使用（a）Ada任务和会合以及（b）occam2进程同步的通道完成练习7.8。

9.9 使用POSIX消息队列，重做练习9.6。

9.10 考虑一个Ada系统，其中有三个抽烟人任务和一个代理任务。每个抽烟人连续地卷烟并抽烟。卷一支烟，需要三种材料，即烟草，烟和火柴。一个抽烟人任务有无限多的纸，另一个有无限多的烟草，第3个有无限多的火柴。代理任务保证三种材料都是无限多。每个抽烟人必须与代理任务通信得到他们没有的两种材料。

代理任务的规格说明如下：

```
task Agent is
  entry Give_Matches (...);
  entry Give_Paper (...);
  entry Give_Tobacco (...);
  entry Cigarette_Finished;
```

```
end Agent;
```

Agent任务体随机选择两种材料，然后接受在相关入口上的通信，以将材料传送给抽烟人。一旦将两种材料都传送了，它就在重复循环之前在Cigarette_Finished入口上等待。一个抽烟人接收到所需材料后，卷出一支烟并抽这支烟，通过调用Cigarette_Finished入口表示完结，再请求新的材料。

勾画出三个抽烟人任务的规格说明和体以及代理任务的任务体。如果有必要，为Agent任务的入口规格说明加上参数。该解决方案应当无死锁。

9.11 一个服务器任务有下面的Ada规格说明:

```
task Server is
  entry Service_A;
  entry Service_B;
  entry Service_C;
end Server;
```

312

写出任务Server的体，使之能完成所有下列操作。

- 如果客户任务在所有入口等待，这个任务应当以循环的次序为客户服务，即首先接收Service_A入口，然后是Service_B入口，然后是Service_C入口、Service_A入口等等。
- 如果不是所有入口都有客户任务等待，Server应当以循环次序为其他入口服务。Server任务不应被阻塞，如果有客户仍在等待服务的话。
- 如果Server任务没有等待的客户，那么它不应当忙碌等待；它应当阻塞，等待客户发出的请求。
- 所有可能的客户都终止了，Server应当终止。

假设客户任务不被中止，并且只发出简单的入口调用。

9.12 一个occam2服务器进程在下列通道上接收同步消息。

```
CHAN OF INT ServiceA, ServiceB, ServiceC;
```

写出此服务器进程的体，使它能完成下列所有操作。

- 如果客户进程在所有通道上等待，这个服务器应当以循环的次序为客户服务，即首先是ServiceA请求，然后是ServiceB请求，然后是ServiceC请求，然后又是ServiceA请求等等。
- 如果不是所有通道都有客户进程等待，服务器应当以循环次序为其他通道服务。服务器不应被阻塞，如果有客户仍在等待服务的话。
- 如果服务器进程没有等待的客户，那么它不应当忙碌等待；它应当阻塞，等待客户发出的请求。

9.13 下列的Ada包提供一种服务。这个服务可能引发异常A、B、C和D。

```
package Server is
  A, B, C, D: exception;
  procedure Service; --可能引发A, B, C, D
end Server;
```

在下列过程中创建了两个任务，任务One和任务Two会合。在会合期间，任务Two调用服务器包提供的Service。

313

```
with Server; use Server;
with Ada.Text_Io; use Ada.Text_Io;
procedure Main is
task One;

task Two is
  entry Sync;
end Two;

task body One is
begin
  Two.Sync;
exception
  when A =>
    Put Line ("A trapped in one");
    raise;
  when B =>
    Put_Line ("B trapped in one");
    raise C;
  when C =>
    Put_Line ("C trapped in one");
  when D =>
    Put_Line ("D trapped in one");
end;


task body Two is
begin -- 块X
  begin -- 块Y
    begin -- 块Z
      accept Sync do
        begin
          Service;
        exception
          when A =>
            Put_Line ("A trapped in sync");
          when B =>
            Put_Line ("B trapped in sync");
            raise;
          when C =>
            Put_Line ("C trapped in sync");
            raise D;
        end;
      end Sync;
    exception
      when A =>
        Put_Line ("A trapped in block Z");
      when B =>
        Put_Line ("B trapped in block Z");
        raise C;
      when others =>
        Put_Line ("others trapped in Z");
```

```
        raise C;
      end; -- 块 Z
    exception
314   when C =>
        Put_Line ("C trapped in Y");
        when others =>
          Put_Line ("others trapped in Y");
          raise C;
      end; --块 Y
    exception
      when A =>
        Put_Line ("A trapped in X");
      when others =>
        Put_Line ("others trapped in X");
  end; -- 块 X 和 任务 TWO

begin   -- 过程 main
  null;

exception
   when A = >
      Put_Line ("A trapped in main");
   when B = >
      Put_Line ("B trapped in main");
   when C = >
      Put_Line ("C trapped in main");
   when D =>
      Put_Line ("D trapped in main");
end Main;
```

过程Put_Line在包Text_Io中声明，在被调用时，在终端上打印出它的变元。

如果过程Service中发生了下面的事，将会输出什么？

(1) 引发异常A。

(2) 引发异常B。

(3) 引发异常C。

(4) 引发异常D。

假设输出不会受对Put_Line的并发调用的影响。

9.14 考虑下面的 occam2 程序段。

```
INT Number.Of.Procs is 10:
-- 需要的其他声明

PROC Controller
   -- 过程Controller的代码
:

PROC Stop.Go (VAL INT Id)
   --过程的代码
:

PAR
  PAR i = 0 FOR Number.Of.Procs
```

```
    SEQ
  --用户进程代码的A部分
  Stop.Go (i);
  --用户进程代码的B部分
Controller
```

这会产生10个进程（它们都执行其A部分代码，然后调用过程Stop.Go，再执行其B部分代码）和Controller进程。Stop.Go和Controller过程的目标是确保在B部分的任何执行之前至少6个用户进程执行它们的A部分。 315～316

说明如何才能实现Stop.Go和Controller过程。如果需要，引入另外的声明。

第10章　原子动作、并发进程和可靠性

第5章讨论了怎样在出现各种错误的情况下生产可靠的软件。模块分解和原子动作作为两种基本的技术用来进行损害隔离和评估。此外，介绍作为动态出错恢复方法的向前和向后出错恢复的概念。已经证明，只要进程进行通信、同步它们的活动，向后出错恢复就可能导致多米诺效应。在第6章中，将异常处理作为一种在顺序进程中提供向前和向后出错恢复的机制进行过讨论。第7、8和9章研究了操作系统和实时语言为并发编程提供的设施。本章把异常处理和并发性结合在一起讨论，以说明进程之间如何在其他进程出现和故障出现的情况下可靠地进行交互。本章还更细致地探讨了原子动作的概念，对异步事件和异步控制转移的概念也作了介绍。

合作和竞争进程

在第7章，根据进程之间的以下三种行为，描述了它们之间的交互：

- 独立性
- 合作性
- 竞争性

独立进程之间不进行通信或同步。因此，如果一个进程内出现错误，那个进程可以启动恢复过程，而与系统其他部分隔离。可以使用在第5章和第6章描述的恢复块和异常处理。

与独立进程相比，合作进程为了执行共同的操作而有规律地通信并同步它们的活动。如果有出错情况发生，所有相关的进程都必须执行出错恢复。这种出错恢复的编程是本章的主题。

竞争进程为了得到资源而进行通信和同步，然而，它们本质上是独立的。一个进程内的出错不应影响别的进程。令人遗憾的是，情况并不总是这样，尤其是出错发生在一个进程正在被分配资源的时候。资源的可靠分配将在第11章研究。

只要合作进程通过共享资源进行通信和同步，恢复就可能涉及到资源自身。资源分配的这个方面也将在第11章讲述。

317

10.1　原子动作

将并发进程引入一门语言的主要动机之一是它们使现实世界中的并行性能在应用程序中

得到反映。并发进程的引入使得能以更自然的方式表达这种程序，产生更可靠和更易维护的系统。然而令人失望的是，并发进程产生了许多在纯顺序程序中不存在的新问题。因此，后面几章致力于讨论这些问题的解决方案：尤其是（正确地）使用共享变量和消息传递的进程之间的通信和同步。这是用一种相当孤立的方式进行的，还没有考虑如何将一组并发进程结构化、使之协调它们的活动这种方式。

到目前为止，两个进程间的交互用单个的通信表达。然而，现实生活中却并不总是这样。例如，从银行账户中提款可能涉及到一个分类账进程和一个支付进程，它们之间一系列的通信验证提款人、检查账目是否平衡和付款。此外，有可能需要两个以上进程用这种方式交互，以执行所需的动作。在所有这种情形中，使所有涉及的进程看到一个一致的系统状态是绝对必要的。对并发进程而言，很容易在一组进程之间发生相互干扰。

一组进程执行共同活动需要的是**不可分的动作**或**原子动作**。当然，一个单独的进程也可
318 能需要保护自己免受其他进程的干扰（例如，在资源分配时）。由此可见，一个原子动作可能涉及一个或多个进程。原子动作也被称为“*多方*”（multiparty）交互（Evangelist等，1989；Yuh-Jzer and Smolka，1996）。

有几种几乎等价的方式表达原子动作的特性（Lomet，1977；Randell等，1978）。

1）一个动作是原子的，如果执行这个动作的进程在执行这个动作期间没有感知到别的活动进程的存在，并且别的活动进程也没有感知此进程的活动。

2）一个动作是原子的，如果在动作执行时，执行这个动作的进程不与别的进程通信。

3）一个动作是原子的，如果执行动作的进程除了它们自己改变的状态，检测不到任何状态改变，并且直到动作执行完成它们才发现它们的状态的改变。

4）动作是原子的，如果它们对其他进程而言是不可分的和瞬时的，使得对系统产生的影响就像是它们被穿插执行的而不是并发执行。

上面的定义并不完全相同。例如第二个表述：一个动作是原子的，如果执行动作的进程只在它们之间进行通信，而不与系统中别的进程通信。与其余的三个定义不同，这句话没有真正定义原子动作的本质。虽然它保证原子动作是不可分的，却对进程有很强的约束。一个原子动作与系统其余部分之间的交互是允许的，只要这些交互不影响原子动作的活动，并且不向系统其余部分提供任何有关该动作进展的信息（Anderson and Lee，1990）。通常，为了允许这些交互，需要有原子动作的功能和原子动作与系统其余部分之间接口的知识。因为一般的语言实现并不支持这个要求，按Anderson 和 Lee（1990）的说法，采用更严格的（第二个）定义就有吸引力了。然而，这一点要求仅在下列情况下可以得到满足：完成一个原子动作所需要的资源是通过底层实现得到，而不是通过程序里的指令得到。如果只当程序员乐意的时候资源才能获得和释放，原子动作里面的进程将不得不和通用资源管理器通信。

虽然一个原子动作被看作是不可分的，它仍可能有内部结构。为了允许原子动作的模块分解，引入了**嵌套原子动作**（nested atomic action）的概念。嵌套原子动作涉及的进程一定是更外层动作涉及进程的子集合。否则，嵌套原子动作就能窃取有关外层动作的信息给外部的
319 进程。外层动作因此不再是不可分的。

10.1.1 两阶段原子动作

理想情况下，原子动作里面涉及的所有进程应该在原子动作开始之前就得到（在动作持续期间）所需的资源。在原子动作终止之后，这些资源才被释放。如果上述规则得到遵守，

原子动作就不需要和外部实体交互，并且能采用更严格的原子动作的定义。

遗憾的是，这种理想想法导致资源利用率低下，因此需要一种更实用的方法。首先，必须允许原子动作在没有得到全套资源的情况下也能开始执行。在某些点上，原子动作内的进程将请求资源分配，于是该原子动作必须和资源管理器通信。资源管理器可能是一个服务器进程。如果采用原子动作的严格定义，这个服务器可能不得不成为原子动作的一部分，并产生把涉及服务器的所有原子动作串行化的效果。显然，这不是我们期望的，因此允许原子动作与资源服务器进行外部的通信。

在这个上下文里面，资源服务器定义为一个非共享系统实用程序的监管者。它保护这些实用程序，防止对它们的不适当访问，但资源服务器本身并不对这些程序执行操作。

如果允许进程在相关的原子动作完成之前就释放资源，就能在资源分配方面做进一步的改善。为了使资源的提前释放有意义，就必须使提前释放的资源的状态与直到原子动作执行完毕后才释放的资源的状态相同。显然，提前释放将增强整个系统的并发性。

如果资源推迟得到和提前释放，提前释放的资源就可能影响外部状态的改变，并观察到新资源的获取。这将破坏原子动作的定义。必须遵守的原则是，资源使用的惟一安全策略有两个不同的阶段。在第一步的“成长”阶段，可以（仅仅）申请资源；在随后的“收缩”阶段，可以释放资源（但不允许新的资源分配）。使用这种结构，确保了原子动作的完整性。然而，应该注意到的是，如果资源过早释放，当原子动作失效时就更难恢复。这是因为资源已经更新，而另一个进程可能观察到了此资源的新状态。任何在别的进程里引发恢复的尝试都可能导致多米诺效应（见5.5.3节）。

在随后的所有讨论里，假设原子动作是两阶段的，直到动作成功完成，可恢复动作才释放资源。

10.1.2 原子事务

在操作系统和数据库理论里，经常使用**原子事务**这个术语。原子事务除了有原子动作的所有特性外，还有它自己的特点：原子事务的执行允许成功或失败。如果失败，就意味着发生了事务不能恢复的错误，例如，处理器失效。如果一个原子动作失败，那么它操纵的系统部件就可能处于不一致的状态。在原子事务中，不可能出现这种错误，因为部件返回到它们的初始状态（就是事务开始之前它们所处的状态）。原子事务有时称作**可恢复动作**，不过，术语**原子动作**和**原子事务**经常被互换地使用。 320

原子事务的两个与众不同的特点是：

- 失败原子性，意思是事务要么完全成功，要么（在失败的情况下）无任何作用。
- 同步原子性（或隔离），意思是事务是不可分的，即它的部分执行不可能被任何并发执行的事务观察到。

虽然原子事务适宜于管理数据库的应用，但是它们本质上不适合于编写容错系统。这是因为它们隐含要求系统提供某种形式的恢复机制。应该修改这种机制，因为它使程序员不能控制原子事务的操作。虽然原子事务提供向后出错恢复的一种形式，但它们不允许执行恢复过程。虽然如此，原子事务仍在保护实时数据库系统的完整性方面拥有一席之地。

原子事务将在第14章的分布式系统和处理器失效中再次研究。

10.1.3 对原子动作的需求

如果一种实时编程语言准备支持原子动作，它就必须能够表达实现原子动作的必要需求。

这些需求和进程的概念以及编程语言提供的进程间通信形式无关。它们是：

- **定义良好的边界**

  每个原子动作应该有开始边界、结束边界和侧面边界。开始边界是原子动作内所涉及的每个进程中的一个位置，准备在这个位置开始这个动作。结束边界是原子动作内所涉及的每个进程中的一个位置，准备在这个位置结束这个动作。侧面边界把原子动作涉及的进程与系统的其他部分隔离开。

- **不可分性（隔离）**

  原子动作不允许在原子动作内的活动进程和外界的进程（资源管理器除外）进行信息交换。如果两个原子动作共享某个数据，那么这个数据在执行原子动作之后的值由按某种次序的这两个原子动作的严格顺序决定。原子动作开始时没有隐含的同步。进程能在不
321 同时间进入。然而，原子动作结束时却有隐含的同步，直到所有的进程愿意并能够离开原子动作才允许进程离开原子动作。

- **嵌套**

  原子动作可以嵌套，只要它们不互相重叠。所以，通常只允许严格的嵌套（如果一种结构完全包含在另一个结构中，这两种结构就是严格嵌套的）。

- **并发性**

  应当可能并发地执行不同的原子动作。增强不可分性的方法之一是顺序执行原子动作。然而，这可能严重影响整个系统的性能，所以应该避免顺序执行。不过，并发执行原子动作集合产生的整体效果必须和顺序执行这些原子动作获得的一样。

- 由于目的是原子动作应当作为损害隔离的基础，所以它们必须能够将恢复过程编程。

图10-1用图形表示了由6个进程组成的系统中嵌套原子动作的边界。动作B只涉及P3和P4，而动作A还包括P2和P5。其余的进程（P1和P6）在这两个原子动作之外。

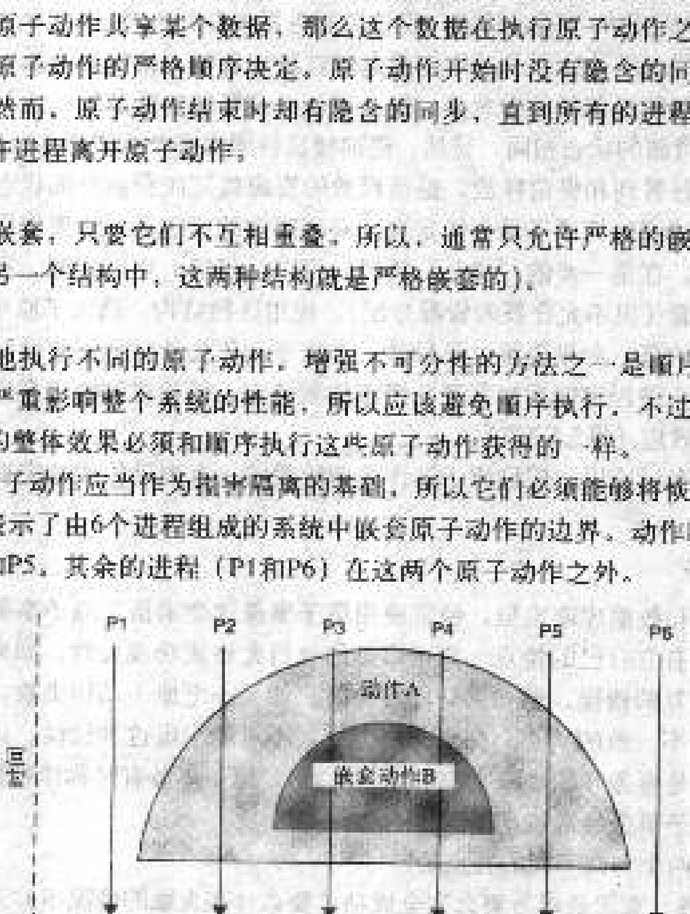


图10-1 嵌套原子动作

也许在这里应该注意的是，原子动作的某些定义要求所有的进程在原子动作的入口和出口处都同步。

10.2 并发语言中的原子动作

原子动作为大型嵌入式系统的软件提供了结构支持。为了充分利用这种支持的好处，还
322 要求有实时语言的支持。遗憾的是，没有任何一个主要语言提供这种支持。本节研究在第8章和第9章讨论过的各种同步和通信原语对原子动作编程的适宜性。紧接着，给出一个可行的语

言框架，然后扩展这个框架以提供向前和向后出错恢复。

资源分配问题放到第11章讨论。就现在而言，假定有两种资源使用模式：共享的和非共享的，某些资源两种模式都应适合。此外，假定所有的动作是两阶段的，并且资源管理器将确保对资源的合理使用。原子动作内的进程还同步它们之间对资源的访问，以避免产生干扰。

10.2.1 信号量

单个进程执行的一个原子动作可以用二元信号量的简单互斥来实现。

```
wait (互斥信号量);
  原子动作;
signal (互斥信号量);
```

然而，当原子动作涉及一个以上的进程时，这种“信号量”解决方案很复杂。例如，考虑一个需要由两个进程操纵的非共享资源。下面的代码说明进程P1和P2怎样实现这种资源操纵，同时又避免来自别的进程的干扰。信号量atomic_action_begin1和atomic_action_begin2仅允许两个进程进入原子动作。其余的进程则被阻塞。另外两个信号量atomic_action_end1和atomic_action_end2，用来保证任何一个进程都不能离开原子动作，直到另一个进程也准备离开。应该注意的是，还需要另外的信号量控制对共享资源的访问（并且首先分配资源）。

```
atomic_action_begin1, atomic_action_begin2 : semaphore := 1;
atomic_action_end1, atomic_action_end2 : semaphore := 0;

procedure code_for_first_process is
begin
    -- 启动原子动作
    wait (atomic_action_begin1);
       -- 获得非共享模式的资源
       -- 更新资源

       -- 发信号给第二个进程：可以访问资源了

       -- 可能有的最后处理

    wait (atomic_action_end2);
    -- 返回资源
    signal (atomic_action_end1);
    signal (atomic_action_begin1);
end code_for_first_process;


procedure code_for_second_process is
begin
    --启动原子动作
    wait (atomic_action_begin2);
       -- 初始处理

       -- 等待第一个进程发信号说可以访问资源了

       -- 访问资源

    signal (atomic_action_end2);
    wait (atomic_action_end1);
    signal (atomic_action_begin2);
end code_for_second_process;
```

323

这种结构提供了必要的保护，并且再次说明了信号量能用来编程大多数同步问题，遗憾的是，这种方法有几个缺点。首先，虽然只允许两个进程进入原子动作，但并不保证它们是那两个正确的进程。其次，我们已在前面的章节批评了信号量的使用，因为信号量是易出错的，没有保护的单个访问（更确切地说，省去wait-signal结构）将破坏原子性。最后，将这个解决方案扩展到N个进程将变得更复杂，因此更容易出错。

10.2.2 管程

通过把原子动作封装在管程里面，就更容易确保原子动作的部分执行的不可见性。上面的例子用管程实现后的情况如下所示。每个过程开始处的if语句保证只有一个进程进行访问。然后条件变量提供原子动作内的正确同步。

然而，这种解决方案存在两个问题。两个进程不可能在管程内同时处于活动状态。经常是限制太多，这是不必要的。此外，嵌套原子动作实现和资源分配将需要嵌套的管程调用。与嵌套管程调用相关的困难已在8.6.4节讨论过。执行嵌套管程调用时，对管程锁的维护工作
324 可能不必要地延迟试图在外层原子动作里面执行的进程。

```
monitor atomic_action
    export code_for_first_process, code_for_second_process;

    first_process_active : boolean := false;
    second_process_active : boolean := false;
    first_process_finished : boolean := false;
    second_process_finished : boolean := false;
    no_first_process, no_second_process : condition;
    atomic_action_ends1, atomic_action_ends2 : condition;

procedure code_for_first_process
begin
    if first_process_active then
      wait (no_first_process);
    first_process_active := true;
       -- 以非共享模式取得资源
       -- 更新资源

       -- 发信号给第二个进程：可以访问资源了

       -- 任何最后的处理
     if not second_process_finished then
       wait (atomic_action_end2);
     first_process_finished := true;
     -- 释放资源
     signal (atomic_action_end1);
     first_process_active := false;
     signal (no_first_process);
end;

procedure code_for_second_process
begin
    if second_process_active then
      wait (no_second_process);
    second_process_active := true;
      -- 初始处理
```

```
        -- 等待第一个进程发信号说可以访问资源了
        -- 访问资源
      signal (atomic_action_end2);
      second_process_finished := true;
      if not first_process_finished then
        wait (atomic_action_end1);
      second_process_active := false;
      signal (no_second_process);
  end;
```

通过把管程作为“动作控制器”，并且在管程之外执行原子动作，就可以去掉上述解决方案的许多限制。下面给出的Ada模型就采用了这种方法。 325

10.2.3 用Ada实现原子动作

对于通信和同步原语只是基于消息传递的语言，如果没有共享变量并且在动作执行期间进程自身不进行通信，单个进程的所有动作都是原子的。例如，在Ada中的扩展会合设计得可以为普通形式的原子动作编程。一个任务为了请求计算而与别的任务通信就是这种情形，被调用的任务承担计算工作，然后通过会合的输出参数回复。原子动作采用**接受**语句的形式，它具有同步原子性，只要：

- 它不更新别的任务可以访问的任何变量，并且
- 它不和任何别的任务会合。

三个任务使用的一个Ada原子动作可以用嵌套会合编写，然而，这将不允许在原子动作里面有任何并行性。

一个替代的模型是创建一个动作控制器，并且编写必要的同步。下面是使用这种控制器的保护对象所采用的方法。

```
package Action_X is
  procedure Code_For_First_Task (--参数);
  procedure Code_For_Second_Task (--参数);
  procedure Code_For_Third_Task (--参数);
end Action_X;


package body Action_X is
  protected Action_Controller is
    entry First;
    entry Second;
    entry Third;
    entry Finished;
  private
    First_Here : Boolean := False;
    Second_Here : Boolean := False;
    Third_Here : Boolean := False;
    Release : Boolean := False;
  end Action_Controller;

  protected body Action_Controller is
    entry First when not First_Here is
```

```
      begin
        First_Here : = True;
      end First;

      entry Second when not Second_Here is
      begin
        Second_Here := True;
      end Second;

      entry Third when not Third_Here is
      begin
        Third_Here := True;
      end Third;

      entry Finished when Release or Finished' Count = 3 is
      begin
        if Finished' Count = 0 then
          Release := False;
          First_Here := False;
          Second_Here := False;
          Third_Here := False;
        else
          Release := True;
        end if;
      end Finished;
    end Action_Controiler;

    procedure Code_For_First_Task (--参数) is
    begin
      Action_Controller.First;
      -- 获取资源
      -- 动作本身，通过资源同在此动作内的正在执行的任务通信
      Action_Controller.Finished;
      -- 释放资源
    end Code_For_First_Task;

    -- 第二个和第三个任务类似
  begin
    -- 局部资源的初始化
  end Action_X;
```

326 327

在上面的例子中，用保护对象Action_Controller对动作进行同步。这使得在任何时候在原子动作内只有三个任务是活动的，并且它们在出口处被同步。布尔变量Release用来编写Finished上所需的释放条件。因为屏障表达式的两部分都为假，对Finished的头两次调用将被阻塞。当第三次调用到来时，Count的值将变为3，屏障变成打开的，并且一个任务将执行入口体。Release变量确保释放另外两个任务。最后退出的任务必须确保屏障再次关闭。

关于怎样编写Ada中的原子动作的更多细节，可以在Wellings和Burns（1997）的文章中找到。

10.2.4 用Java实现原子动作

前一节说明了编写原子动作的基本结构。Java方法能够遵循相似的结构。然而，本节利用这个机会扩充这种方法，使得使用继承就能容易地扩展Java支持。

首先，定义三路原子动作的接口：

```
public interface ThreeWayAtomicAction
{
  public void role1 ();
  public void role2 ();
  public void role3 ();
}
```

使用这个接口，就可能提供几个实现各种模型的动作控制器。应用程序不用修改代码就能选择合适的控制器。

下面的动作控制器实现了与前面为信号量、管程和Ada给出的相同语义。其算法和Ada版本的非常相似。一个同步类`Controller`实现了所需的入口和出口同步协议。然而，方法`finished`比Ada版本的稍复杂些。这归因于`wait`和`notifyAll`的语义。特别是，为了知道何时为下一个动作复位内部数据结构，当进程离开原子动作时（用`toExit`），必须为进程计数。在Ada中，这种功能是利用`Count`属性实现的。

```
public class AtomicActionControl implements ThreeWayAtomicAction
{
  protected Controller Control;
  public AtomicActionControl () // 构造器
  {
    Control = new Controller ();
  }

  class Controller
  {
    protected boolean firstHere, secondHere, thirdHere;
    protected int allDone;
    protected int toExit;
    protected int numberOfParticipants;

    Controller ()
    {
      firstHere = false;
      secondHere = false;
      thirdHere = false;
      allDone = 0;
      numberOfParticipants = 3;
      toExit = numberOfParticipants;
    }

    synchronized void first () throws InterruptedException
    {
      while (firstHere) wait ();
      firstHere = true;
    }
```

328

```
    synchronized void second () throws InterruptedException
    {
      while (secondHere} wait ();
      secondHere = true;
```

```
    }

    synchronized void third () throws InterruptedException
    {
      while (thirdHere) wait ();
      thirdHere = true;
    }

    synchronized void finished () throws InterruptedException
    {
      allDone++;
      if (allDone == numberOfParticipants) {
         notifyAll ();
      } else while (allDone != numberOfParticipants) {
           wait ();
      }
      toExit--;
      if (toExit == 0) {
         firstHere = false;
         secondHere = false;
         thirdHere = false;
         allDone = 0;
         toExit = numberOfParticipants;
         notifyAll (); // 释放等待下一个动作的进程
      }
    }
  }

  public void role1 ()
  {
    boolean done = false;
    while (!done) {
      try {
        Control.first ();
        done = true;
      } catch (InterruptedException e) {
        // 忽略
      }
    }

    // .... 执行动作

    done = false;
    while (!done) {
      try {
        Control.finished ();
        done = true;
      } catch (InterruptedException e) {
        // 忽略
      }
    }
  };

  public void role2 ()
```

```
  {
    // 类似于 role1
  }

  public void role3 ()
  {
    // 类似于role1
  }
}
```

有了上述框架，现在就可以把它扩展到四路动作:

```
public interface FourWayAtomicAction extends ThreeWayAtomicAction {
  public void role4 ();
}
```

然后，

```
public class NewAtomicActionControl extends AtomicActionControl
             implements FourWayAtomicAction
{
  public NewAtomicActionControl ()
  {
    Control = new RevisedController ();
  }

  class RevisedController extends Controller
  {
    protected boolean fourthHere;

    RevisedController () {
      super ();
      fourthHere = false;
      numberOfParticipants = 4;
      toExit = numberOfParticipants;
    }

    synchronized void fourth () throws InterruptedException
    {
      while (fourthHere) wait ();
      fourthHere = true;
    }

    synchronized void finished  () throws InterruptedException
    {
      super.finished ();
      if (allDone == 0) {
        fourthHere = false;
        notifyAll ();
      }
    }
  }

  public void role4 ()
  {
    boolean done = false;
```

```
      while (!done) {
        try {
          // 由于Control 是 Controller类型的,
          // 它必须首先转换成RevisedController
          // 以调用第四个方法
          ((RevisedController) Control) .fourth ();
          done = true;
        } catch (InterruptedException e) {
          // 忽略
        }
      }

      // ....执行动作

      done = false;
      while (! done) {
        try {
          Control.finished ();
          done = true;
        } catch (InterruptedException e) {
          // 忽略
        }
      }
    }
  }
```

需要注意的是，有必要覆盖方法finished。由于方法调用的Java自动运行时分派，这里需要格外小心。原始代码中所有对finished的调用将被分派到已覆盖的方法。

10.2.5 用occam2实现原子动作

只要语言只支持消息传递，动作控制器本身也就是进程。动作的每个部件在它的原子操作开始和结束时，各发送一个消息给控制器。因为occam2的会合机制没有扩展，在原子动作结束时需要和控制器进行双重交互。又因为occam2通道仅有一个读者和一个写者，所以动作控制器必须为动作的每一个部件准备一个通道数组。潜在的进程从这个通道数组中分配一个通道。

329~331

```
VAL INT max IS 20: -- 在三个组的任一组中的最大客户进程数

[max] CHAN OF INT First, Second, Third:
CHAN OF INT First.Finished, Second.Finished, Third.Finished:
CHAN OF INT First.Continue, Second.Continue, Third.Continue:
-- 所有上述通道只用于同步
-- 它们默认地被定义为遵守INT 协议

PROC Action.Controller
  BOOL First.Here, Second.Here, Third.Here:
  INT Any:
  WHILE TRUE
    SEQ
      First.Here := FALSE
      Second.Here := FALSE
      Third.Here := FALSE
```

```
      WHILE NOT (First.Here AND Second.Here AND Third.Here)
        ALT
          ALT i = 0 FOR max
            NOT First.Here & First[i]?Any
              First.Here := TRUE
          ALT i = 0 FOR max
            NOT Second.Here & Second[i]?Any
              Second.Here := TRUE
          ALT i = 0 FOR max
            NOT Third.Here & Third[i]?Any
              Third.Here := TRUE
      First.Finished?Any
      Second.Finished?Any
      Third.Finished?Any
      PAR
        First.Continue!Any
        Second.Continue!Any
        Third.Continue!Any
:

PROC action.1 (CHAN OF INT first.client)
  INT Any:
  SEQ
    first.client!Any

    -- 动作自身

    First.Finished!Any
    First.Continue?Any
:
-- 类似地对 action.2 和 action.3

PAR
  -- 系统中的所有进程，包括
  Action.Controller
```

10.2.6 原子动作的语言框架

虽然上面描述的各种语言模型都可以表示一个简单的原子动作，但它们都依靠编程人员遵守某些规定来保证不与外部进程发生交互作用（资源分配器除外）。此外，它们假定原子动作内没有进程被中止；如果实时语言支持中止设施，那么可以从原子动作中异步地删除一个进程，从而导致动作处于不一致的状态。 332

通常，没有一个主流语言或操作系统直接支持原子动作的向后或向前出错恢复设施。（然而，Ada、Java和POSIX提供异步通知机制，这种机制能用来帮助程序进行出错恢复——见10.6、10.8和10.9节。）在面向研究的系统中，已经提出了语言机制。为了讨论这些机制，这里为原子动作引入一个简单的语言框架。然后在这个框架里讨论提出的出错恢复机制。

为了简化这个框架，只考虑静态进程。还假设所有参与原子动作的进程在编译时是已知的。参与原子动作的每个进程声明一个动作语句，它包括：动作名、其余参与动作的进程和在动作入口处声明的进程执行的代码。例如，进程P_1希望进入包含进程P_2和P_3的原子动作A，它会声明如下动作：

```
action A with (P2, P3) do
    -- 获得资源
    -- 同P2和P3通信
    -- 释放资源
end A;
```

假设三个进程都知道资源分配器，并且动作内的通信仅限于在资源分配器同这三个*P*进程之间进行（包括对资源分配器的外部调用）。编译时对这些限制进行检查。所有别的进程声明类似的动作，并且只要遵守严格的嵌套规定，就允许嵌套动作。如果在编译时进程是未知的，那么同进程的通信只在同一原子动作内的两个进程都处于活动状态才被允许。

在动作上施加同步机制按如下进行。进入动作的进程不被阻塞。进程只在下面的情况下被阻塞在动作里面：进程不得不等待分配资源；或进程试图与动作内的另一个进程通信，那个进程虽处于活动状态、但不处于接收通信的位置，或者那个进程在动作内不处于活动状态。

对动作内的任何进程而言，只有当动作内所有活动进程都希望离开时，才能离开这个动作。前面为信号量、管程、Ada和occam2举的例子不是这种情况。在那些情况下，假设所有进程必须在任何进程可以离开之前进入动作。这里，一个指名进程的子集合可能在进入动作后随后离开（不求助于与失踪进程的交互）。在时限很重要的实时系统里，这被认为是基本功能。它解决了只因一个进程没有到达、所有的进程都挂在一个动作里面的**离弃者**（deserter）
333 问题。这个问题将在下两节中和出错恢复一起讨论。

10.3 原子动作和向后出错恢复

在上一节中，我们讨论了原子动作的概念。原子动作非常重要，因为它们约束信息在系统和定义良好的边界内流动，并因此为损害隔离和出错恢复都提供了基础。本节描述并发进程之间的两种向后出错恢复方法。处理器失效中的向后出错恢复放到第14章讨论。

在第5章阐述过，当把向后出错恢复应用于几组通信进程时，有可能把所有进程回滚到进程执行时的起始状态。这就是所谓的*多米诺效应*。出现问题的原因是没有一致的恢复点集合或一个恢复线。原子动作自动提供恢复线。如果原子动作内发生错误，参与的进程能回滚到动作起始处并执行替代算法，原子动作确保进程在和动作外部的进程通信时没有传递任何错误值。这种使用原子动作的方式称作**会话**（conversations）（Randell，1975）。

10.3.1 会话

在会话方式下，每个动作语句包含一个恢复块。例如：

```
action A with (P2, P3) do
   ensure <接受测试>
   by
      -- 基本模块
   else by
      -- 替代模块
   else by
      -- 替代模块
   else error
end A;
```

会话内涉及的其余进程以类似方式声明它们在动作内的角色。会话的基本语义可概述如下：

- 在会话的入口，保存进程状态。入口点集合形成恢复线。
- 在会话内，只允许进程和会话内的其余活动进程和通用资源管理器通信。因为会话是从原子动作建立的，这个属性被继承。
- 为了离开会话，会话内的所有活动进程必须通过它们的接受测试。如果是这样，则会话完成，并且放弃所有恢复点。
- 如果有任何进程没有通过接受测试，则所有的进程恢复到会话开始时保存的状态，并执行替代模块。因此，假设在会话内执行的出错恢复必须被所有参与此会话的进程执行。 334
- 会话可以嵌套，但必须是严格嵌套。
- 如果会话内的所有替代模块失效，那么必须在更高层上执行恢复。

应该注意的是，正如Randell（1975）定义的那样，在会话中所有参与会话的进程必须在任一其他进程能够离开会话前进入会话。这不同于上面描述的语义。如果一个进程由于迟到或它已失败而没有进入会话，那么只要会话内的其余活动进程不希望和它通信，会话就能成功地结束。如果一个进程试图和失踪的进程通信，那么它或者阻塞，并等待失踪进程的到来；或者继续执行。采用这种方法有两个好处（Gregory and Knight，1985)：

- 这种方法允许指定会话的哪些部分不是强制参与的。
- 这种方法允许有时限的进程离开会话、继续执行、以及如果有必要，执行替代动作。

虽然会话允许各组进程协调它们的恢复，但还是遭到了批评。一个重要的原因是当一个会话失效时，要恢复所有的进程并使它们执行替代模块。这迫使相同的进程再次通信，以实现想要的效果，一个进程不能从会话中脱身。这可能不是所期望的。Gregory和Knight（1985）指出：在实践中，当一个进程在基本模块里通过和一组进程通信没能达到目标时，它可能希望在它的次级模块里和全新进程组通信。此外，次级模块的接受测试可能很不一样。用会话无法表示这些需求。

10.3.2 对话和会谈

为了克服和会话相关的一些问题，Gregory和Knight（1985）提出了另一个并发进程间向后出错恢复的方法。在他们的方案里，一组希望参与向后恢复的原子动作的进程，通过执行一个dialog（对话）语句来表明它们的愿望。dialog语句有三个功能：标识原子动作、为原子动作声明全局接受测试、规定动作内要使用的变量。对话语句采用以下形式：

```
DIALOG name_and_acceptance_test SHARES (变量)
```

注意，对话的名字和定义接受测试的函数的名字相同。 335

每个希望参与原子动作的进程定义一个指名原子动作的discuss（讨论）语句。讨论语句的格式如下：

```
DISCUSS 对话名 BY
  -- 语句序列
TO ARRANGE 布尔表达式
```

这个布尔表达式是对进程进入这个原子动作的局部接受测试。

讨论语句是原子动作的一个成分，所以它具有上面定义的所有性质。共同形成完整动作的讨论语句组定义了恢复线，因此保存了进入对话的每个进程的状态。除非所有的活动进程成功地通过局部接受测试并且通过全局接受测试，否则没有进程能离开对话。如果没有通过任何一个接受测试，那么对话必将失败，进程也恢复到在原子动作的入口处的状态。

上述情况将结束讨论语句的操作。没有可执行的替代模块，然而可以用另一个称作**对话序列**的语句与讨论语句组合使用。用类比的方法，如果讨论语句等价于Ada的接受语句，那么对话序列就等价于Ada的选择语句。它的语法表示如下：

```
SELECT
    dialog_1
OR
    dialog_2
OR
    dialog_3
ELSE
    -- 语句序列
END SELECT;
```

执行时，进程首先尝试dialog_1，如果成功，控制就转到选择语句后面的语句。如果dialog_1失败，就尝试dialog_2，依此类推。尤其要注意的是，与dialog_1相比，dialog_2可能包含一个完全不同的进程集合。相关选择语句的组合执行叫做**会谈**（colloquy）。

如果所有的对话尝试均失败，就执行ELSE后的语句。这就为程序员挽救对话提供了最后的机会。如果这也失败了，任何外包的会谈都将失败。注意，通过执行fail语句，进程可以显式地使对话或会谈失败。

为了使会谈概念在实时系统中得到使用，有必要把超时和选择语句关联起来，这将在12.8.2节讨论。

10.4 原子动作和向前出错恢复

在第5章中我们指出过，虽然向后出错恢复可以使原子动作从未预期的错误中恢复，但在
336 嵌入式系统的运行环境中执行的操作很难撤消。因此必须考虑向前出错恢复和异常处理。在本节中，讨论原子动作里面涉及的并发进程间的异常处理。

对向后出错恢复来说，当有一个错误发生时，原子动作里面涉及的所有进程都参与恢复。这种情况同样适用于异常处理和向前出错恢复。如果原子动作里面有一个活动进程出现异常，这个异常就在原子动作里的所有活动进程中引发。因为这种异常是发自别的进程，所以称它是**异步的**。下面是支持异常处理的原子动作的类Ada语法。

```
action A with (P2, P3) do
   --动作
exception
   when exception_a =>
       -- 语句序列
   when exception_b =>
       -- 语句序列
   when others =>
      raise atomic_action_failure;
end A;
```

按照异常处理的终止模型，如果原子动作内的所有活动进程都有一个处理程序，并且所有处理程序在处理异常时不引发新的异常，则原子动作就正常完成。如果使用恢复模型，当异常被处理完时，原子动作内的活动进程就在异常引发处恢复执行。

不管用什么模型，如果原子动作内的活动进程都没有异常处理程序，或者有一个异常处理程序失败，那么原子动作将失败，并引发标准异常*atomic_action_failure*。在所有相关进程内部引发这个异常。

当为原子动作增加异常处理时有两个问题必须考虑：并发引发的异常的分辨（resolution）和嵌套动作内的异常（Campbell and Randell，1986）。下面对这些问题作简要评述。

10.4.1 并发引发的异常的分辨

原子动作内的多个活动进程有可能同时引发不同的异常。正如Campbell和Randell（1986）指出的那样，如果由故障引起的出错不能由原子动作的每个成分提供的出错检测工具惟一标识，则上述情况就很可能发生。如果在一个原子动作里面同时发生了两个异常，每个进程里就可能有两个独立的异常处理程序，决定应该选用哪个异常处理程序可能是困难的。此外，连接在一起的两个异常形成了第三个异常，第三个异常表示前两个异常发生的条件已经出现。

为了分辨并发引发的异常，Campbell和Randell建议使用**异常树**（exception tree）。如果并发地引发了几个异常，那么用来标识处理程序的异常是包括所有异常的最小子树的根节点（虽然并不清楚怎样组合与这个异常关联的参数）。每个原子动作成分都能声明自己的异常树，一个原子动作内涉及的不同进程可以有不同的异常树。 337

10.4.2 异常和内部原子动作

在原子动作嵌套的情况下，一个原子动作里面的一个活动进程有可能引发异常，而这个原子动作内的其他进程参与嵌套动作。图10-2解释了这个问题。

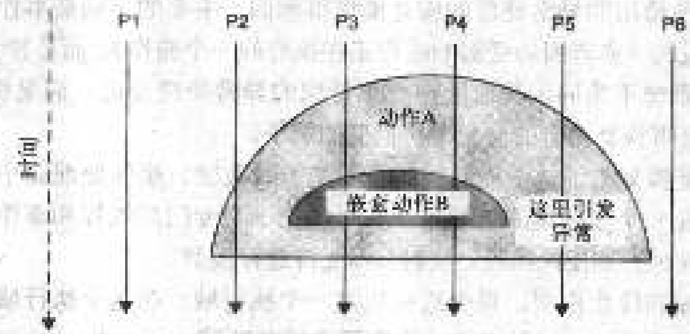


图10-2 嵌套原子动作中的异常

当异常被引发时，涉及的所有进程必须参与恢复动作。但是，依照定义，内部动作是不可分的。引发那个动作里面的异常有可能损害不可分性。此外，内部动作也许对可能引发的异常一无所知。

Campbell和Randell（1986）讨论了这个问题的两个可能的解决方案。第一个方案是阻止异常引发，直到内部动作完成。但由于下面的原因，他们放弃了这个方案：

- 在实时系统中，被引发的异常可能和时限错过有关联。阻止恢复过程可能给动作的及时响应埋下隐患。
- 检测到的出错条件可能表示：因为某种死锁条件的出现，内部动作永不终止。

因为这些原因，Campbell和Randell允许内部动作有预定义的中止异常。引发这个异常向动作表示：异常是在一个外包的动作内引发的，并且调用动作的前提条件不再成立。如果引 338

发了这种异常，内部动作应该调用容错措施中止自己。一旦这个动作被中止，外部动作就能处理最初的异常。

如果内部动作不能中止自身，那么它必须向原子动作失效异常发送一个信号。这个信号可以与未处理完的异常结合起来，去影响由外包动作执行的恢复的选择。如果没有定义中止异常，外包的动作必须等待内部动作的完成。另一个可选方案是，提供一个能引发原子动作失效异常的默认处理程序。

10.5 异步通知

虽然前面已经分别讨论了向前和向后出错恢复，但实际上，在许多实时系统中需要把这两种恢复结合起来。当需要从意料不到的出错恢复时，要用到向后出错恢复，当要撤销或改善与环境的交互时，需要向前出错恢复。实际上，向前出错处理能用来实现一种向后出错恢复方案——见10.8.2节。

如在10.2节讨论的那样，没有一个主流实时语言支持原子动作，因此有必要用更初级的语言设施来达到和原子动作相同的效果。恢复动作也是这种情况。恢复动作主要的需求之一是能够引起动作内被涉及进程的注意，并且通知它另一个进程内发生了错误。大多数语言和操作系统支持某种形式的异步通知机制。至于异常，有两种基本的模型：恢复模型和终止模型。

异步通知处理（经常被称作**事件处理**）的恢复模型的行为同软件中断相似。一个进程表明自己将处理哪些事件，当有信号触发事件时，进程被中断（除非进程暂时抑制了事件的发送）并执行事件处理程序。处理程序负责响应异步事件，然后进程继续从中断处执行。当然，这种机制和6.2.4节给出的异常处理的恢复模型很相似。主要的不同是事件通常不是由受影响的进程用信号触发的（或者因为受影响进程正在执行的一个操作），而是异步地用信号触发的。然而，许多操作系统不为同步异常处理提供特别的异常处理设施，而是使用异步事件机制。POISX信号设施是带恢复功能的异步事件模型的例子。

注意，对恢复模型来说，进程的控制流只是暂时改变，事件处理完毕后，进程又恢复到原来的控制流。在一个多线程的进程里，有可能把一个专门的线程和事件关联在一起，并且当事件被信号触发时，调度此线程。实时Java支持这种模型。

339 对于异步通知的终止模型，每个进程规定一个执行域，在这个执行域中进程准备接收引起这个域终止的异步通知。这种方式常称作**异步控制转移**（asynchronous transfer of control）或ATC。如果ATC请求发生在这个执行域外，可以把它忽略或将其排队。在处理完ATC后，控制返回到被中断的进程，但返回的位置与发出ATC时的不同。这种方式和异常处理的终止模型的非常相似。Ada和实时Java语言支持异步控制转移机制。

对于在语言（或操作系统）里包含异步通知机制是存在争议的，因为这使语言的语义复杂，还增加了运行时支持系统的复杂性。因此本节首先讨论应用的需求，这种需求证明语言包含异步通知设施的合理性。然后讨论异步事件处理的POSIX和实时Java模型，接着讨论Ada和实时Java的异步控制转移机制。

异步通知的用户需要

对异步通知设施的基本需求是能够使进程对于由另一个进程检测到的状态作出快速的响应。重点是快速的响应，显然进程总能用简单的查询或者等待事件来对事件作出响应。事件通知能够被映射到进程的通信和同步机制上。当处理进程处于准备接收事件的状态时，它只

发出适当的请求。

但是，有一些不适于查询事件或者等待事件发生的情形。这包括以下情形：

- **出错恢复**

本章强调过，当几组进程执行原子动作时，如果在一个进程内检测到一个错误，那么需要所有别的进程都参与恢复。例如，硬件故障意味着进程不能完成既定的执行步骤，因为进程继续执行的前提条件已不再成立，进程可能永远也到不了查询点。而且，可能发生时间性故障，这意味着进程不再满足交付服务的时限。在上述这些情况下，必须通知这个进程检测到了一个错误，而且它必须尽快地进行出错恢复。

- **模式改变**

一个实时系统通常有几种操作模式。例如，一个电操纵民用飞机有起飞模式、巡航模式和着陆模式。在许多情况下，可以仔细地管理模式之间的改变，并且在系统执行中的定义明确的点上才发生这种改变，就像正常的民用飞机飞行计划那样。不过，在某些应用领域，预期会出现模式改变，却不能制定出改变的计划。例如，一个故障可能导致飞机放弃起飞并进入紧急操作模式；或者制造过程中的事故要求立即进行模式改变以确保有 340
序地关闭工厂设备。在这些情况下，必须迅速和安全地通知进程，它们运作的模式已经改变，并且现在它们需要执行一套不同的动作。

- **用部分/不精确计算进行调度**

有许多算法，它们的计算结果的精确性依赖于分配给它们的计算时间。例如，数值计算、统计估计和启发式搜索，它们都是先产生所求结果的一个初步估计，然后经过求精，得到更大的精确度。运行时，先分配给算法一定数量的时间，当时间用完后，必须中断进程，停止对结果的求精。

- **用户中断**

在一般的交互式计算环境里，由于用户检测到了一个出错条件并希望重新开始计算，就常常希望停止当前的处理。

异步通知处理的一种方法是中止进程并允许另一个进程执行某种恢复。所有的操作系统和大多数并发编程语言提供此类设施。然而，中止进程的代价是昂贵的，而且是对许多出错条件的极端反应。因此，还需要某种形式的异步通知机制。

10.6 POSIX信号

POSIX提供一种叫做**信号**的异步事件处理机制，信号也用于一类环境检测的同步错误（例如用零除，非法指针等）。信号的定义早于线程，所以扩展信号模型到多线程环境有一些困难。单线程进程的信号模型相当简单（见程序 10-1，POSIX信号的C接口的规格说明）。有一些预先定义的信号，每个信号都分配了一个整数值。还有由实现定义的信号，可供应用程序使用。每个信号有一个默认的处理程序，它通常终止接收进程。信号的例子是：`SIGABRT`用于异常终止，`SIGALARM`用于报警时钟到期，`SIGILL`用于非法指令异常，`SIGRTMIN`用于第一个实时应用可定义异常的标识符，`SIGRTMAX`用于最后一个实时应用可定义异常的标识符。POSIX只认为其数字编码介于`SIGRTMIN`和`SIGRTMAX`之间的信号是实时的。实时信号是有额外信息的信号，由产生它的进程传送给处理程序。此外，它们是排队的。

进程有三种方法处理信号： 341

• 进程可以**阻塞**信号，以后再处理它或者接受它；
• 进程可以**处理**信号（通过在信号发生时设置被调用的函数）；
• 进程可以完全**忽略**信号（在这种情况下信号被简单地丢弃）。

未阻塞且未忽略的信号一**产生**就被**发出**。阻塞的信号**悬挂**起来暂不发出，或者可以调用 sigwait () 函数来**接受**它。

程序10-1 POSIX信号的C接口

```
union sigval {
  int sival_int;
  void *sival_ptr;
};

struct sigevent {
  /* 用于消息队列通知和定时器 */
  int sigev_notify; /* 通知: SIGEV_SIGNAL, */
                    /* SIGEV_THREAD 或 SIGEV_NONE */
  int sigev_signo;  /* 待生成的信号 */
  union sigval sigev_value; /* 待排队的值 */
  void (*) sigev_notify_function (union sigval s);
  /* 要被处理为线程的函数 */
  pthread_attr_t *sigev_notify_attributes;
  /* 线程属性 */
}

typedef struct {
  int si_signo;
  int si_code;
  union sigval si_value;
} siginfo_t;

typedef ... sigset_t;

struct sigaction {
  void (*sa_handler)  (int signum); /* 非实时处理程序 */
  void (*sa_sigaction)  (int signum, siginfo_t *data,
                         void *extra); /* 实时处理程序 */
  sigset_t sa_mask; /* 在处理程序期间的屏蔽信号 */
  int sa_flags; /* 指出信号是不是要排队 */
};

int sigaction (int sig, const struct sigaction *reaction,
               struct sigaction *old_reaction);
/*  为sig启动一个信号处理程序reaction */

/* 下列函数使进程能等待信号 */
int sigsuspend (const sigset_t *sigmask);
int sigwaitinfo (const sigset_t *set, siginfo_t *info);
int sigtimedwait (const sigset_t *set, siginfo_t *info,
                  const struct timespec *timeout);

int sigprocmask (int how, const sigset_t *set, sigset_t *oset);
/* 按how的值处理信号屏蔽                                */
```

```
/* how = SIG_BLOCK -> 此集合被加到当前集合       */
/* how = SIG_UNBLOCK -> 从当前集合减去此集合      */
/* how = SIG_SETMASK -> 给定的集合变成屏蔽集合    */

/* 下列例程使一个信号集合被创建和处理*/
int sigemptyset (sigset_t *s); /* 将集合初始化为空 */
int sigfillset (sigset_t *s); /* 将集合初始化为满 */
int sigaddset (sigset_t *s, int signum); /* 加一个信号 */
int sigdelset (sigset_t *s, int signum); /* 移走一个信号 */
int sigismember (const sigset_t *s, int signum);
   /* 如果是成员，返回1 */

int kill (pid_t pid, int sig);
   /* 向进程pid发信号sig */
int sigqueue (pid_t pid, int sig, const union sigval value);
   /* 发送信号和数据*/

/* 当发生错误时，所有上述函数返回-1 */
/* 共享变量 errno 包含有出错原因 */
```

10.6.1 阻塞信号

POSIX负责维护进程当前屏蔽的信号集合。函数sigprocmask用于操纵这个集合。参数how设置为SIG_BLOCK的含义是向集合中添加信号，设置为SIG_UNBLOCK的含义是从集合中减少信号，设置为SIG_SETMASK的含义是取代这个集合⊖。另外两个参数包括添加/减少/重置（set）信号集合的指针和旧集合（oset）的返回值。各种函数（sigemptyset、sigfillset、sigaddset、sigdelset和sigismember）使得可以操纵信号集合。

当信号被阻塞时，它保持悬挂状态直到它被解除阻塞或被接受。当它被解除阻塞时，就被发出。某些信号不能被阻塞。

10.6.2 处理信号

可以用函数sigaction启动信号处理程序。参数sig指明处理哪一个信号，reaction是指向包含处理程序信息的结构的指针，old_reaction指向上一个处理程序的信息。本质上，一个处理程序的信息包括指向处理程序函数的指针（如果信号是非实时信号，该指针是sa_handler；如果信号是实时信号，该指针是sa_sigaction）、处理程序执行期间屏蔽的信号集合（sa_mask）、信号是否排队（通过设置sa_flags的值为符号常量SA_SIGINFO来表示——只有值位于SIGRTMIN和SIGRTMAX之间的信号参与排队）。成员sa_handler表示和信号相关联的动作，它的值可以是： 342~343

- SIG_DFL——默认的动作（通常是终止这个进程）；
- SIG_IGN——忽略这个信号；
- 指向函数的指针——当信号发出时调用。

对非实时信号，当信号产生时，只能传递一个整数参数给处理程序。此参数的值一般表示信号本身（同一处理程序可用于处理多个信号）。然而，对实时信号，通过指向结构siginfo_t的指针可以传递更多的数据。这个结构包括信号编号（重复）、表示信号起因的代码（例如，一个定时器信号）和一个整数或指针值。

⊖ SIG_BLOCK、SIG_UNBLOCK和SIG_SETMASK是编译时常量。

如果不止一个实时信号在排队，首先发送信号值最小的信号（就是说，SIGRTMIN在SIGRTMIN+1之前发送，依此类推）。

进程可以用函数sigsuspend、sigwaitinfo或sigtimedwait等待信号的到来。函数sigsuspend用传给它的参数带来的值取代屏蔽码并挂起进程，直到:

1）发出一个解除阻塞信号，并且

2）执行相关的处理程序。

如果处理程序终止了这个进程，函数sigsuspend就不返回，否则它返回调用，并将信号屏蔽码重置为调用sigsuspend之前存在的状态。

函数sigwaitinfo也挂起调用进程，直到信号到达。然而，这一次必须阻塞信号，因此不调用处理程序。函数返回选择的信号编号，并将关于已发送信号的信息存放到info变元里。函数sigtimedwait和sigwaitinfo有相同的语义，但它允许为挂起指定超时。如果在超时值到达之前没有发出信号，sigwaitinfo返回 - 1，errno被设置为EAGAIN。

当用信号进行条件同步时必须特别小心。在查看信号是否已经到达和发出导致挂起的请求之间存在潜在的竞争条件。适当的协议是首先阻塞信号，然后测试信号是否已经出现，如

344 果没有出现，挂起并用上述函数之一为信号解除阻塞。

10.6.3 忽略信号

在对函数sigaction的调用中，只要设置sa_handler的值为SIG_IGN，就可以忽略信号。

10.6.4 生成信号

一个进程有两种方式生成一个准备发给另一个进程的信号。第一种方式是利用kill函数，第二种是利用sigqueue函数。后者只能发送实时信号。

然而，注意到当定时器到期时（例如，SIGALRM——见12.8.1节），当异步I/O完成时，进程能通过消息到达一个空消息队列（见9.5节）或者用C的引发语句请求给它自己发送信号。

对定时器、异步I/O和消息到达，POSIX接口允许传递struct sigevent的值。这定义了当接收到事件通知后将发生的动作。有三个选项:

- SIGEV_NONE——什么通知也不发;
- SIGEV_SIGNAL——用通常的方式生成信号;
- SIGEV_THREAD——调用sigev_notify函数，就像它是一个新创建线程的启动例行程序，该线程有sigev_notify_attributes属性。空属性表示这个线程应该被作为已经被分派的对待。

10.6.5 一个POSIX信号的简单例子

作为POSIX信号的一个实例说明，分析下面的程序段。一个进程周期性地进行某种计算。实际进行的计算依赖于系统范围的操作模式。模式改变通过应用程序定义的实时信号MODE_CHANGE传播到所有进程。信号处理程序change_mode就只是改变全局变量mode。进程在每次迭代开始时访问mode。为了确保在访问模式时mode不改变，MODE_CHANGE信号被阻塞。

```
#include <signal.h>

#define MODE_A 1
#define MODE_B 2
```

```
#define MODE_CHANGE SIGRTMIN +1

int mode = MODE_A;

void change_mode (int signum, siginfo_t *data, void *extra) {
  /* 信号处理程序 */
  mode = data -> si_value.sival_int;;
}
int main () {

  sigset_t mask, omask;
  struct sigaction s, os;
  int local_mode;

  SIGEMPTYSET (&mask);
  SIGADDSET (&mask, MODE_CHANGE);

  s.sa_flags = SA_SIGINFO;
  s.sa_mask = mask;
  s.sa_sigaction = &change_mode;
  s.sa_handler = &change_mode;

  SIGACTION (MODE_CHANGE, &s, &os); /* 分配处理程序 */

  while (1) {

    SIGPROCMASK (SIG_BLOCK,  &mask, &omask);
      local_mode = mode;
    SIGPROCMASK (SIG_UNBLOCK, &mask, &omask);

    /*周期性操作使用模式*/
    switch (local_mode) {
      case MODE_A:
        ...
        break;
      case MODE_B:
        ...
        break;
      default:
        ...
    }
  }
  ...
}
```

10.6.6 信号和线程

最初的POSIX信号模型来自Unix，当POSIX的实时扩展确定以后，对POSIX信号模型进行了扩展，以使它更适合于实时应用。随着POSIX线程的扩展，POSIX信号模型变得越来越复杂，并成了“每个进程一个信号”模型和“每个线程一个信号”模型之间的折衷。以下几点应该注意:

- 同步出错条件产生的信号（例如存储器违例）只发送给导致信号产生的线程。
- 其余的信号可以作为整体发送给进程，然而，每个信号只发送给进程中的单个线程。 345~346
- 函数sigaction为进程中的所有线程设置处理程序。

• 函数kill和sigqueue依然适用于进程。一个新函数pthread_kill

```
int pthread_kill (pthread_t thread, int sig);
```

使进程能发送信号给单个线程。

• 还能用函数pthread_cancel终止线程。

```
int pthread_cancel (pthread_t thread);
```

可以用函数pthread_setcancelstate使撤消请求的效果失效，或者用函数pthread_setcanceltype延迟它。

```
int pthread_setcancelstate (int state, int *oldstate);
int pthread_setcanceltype (int type, int *oldtype);
```

• 如果有不止一个线程适合接收一个发出的信号，没有定义选用哪一个线程。
• 如果一个处理程序为信号指定的动作是终止，那么**整个进程也终止**，而不仅仅是线程终止。
• 可以在基于每个线程的基础上用函数pthread_sigmask阻塞信号，此函数和sigprocmask有相同的参数集合。没有规定函数sigprocmask只用于多线程的进程。
• 函数sigsuspend、sigwaitinfo或者sigtimedwait的操作是在调用线程上，而不是在调用进程上。
• 一个新的函数sigwait

```
int sigwait (const sigset_t *set, int *sig);
```

使一个线程能等待几个阻塞的信号之一出现。它的行为和sigwaitinfo ( ) 一样，除了不返回和信号相关的信息。信号在引用的位置set处指定。当执行了一个成功的等待，函数返回零，由sig所引用的位置包含有接收的信号。

当调用函数时，如果信号中有一个已挂起，函数立即返回。如果不止一个挂起，并没有定义选择哪一个，除非只有实时信号挂起。在这种情况下，选择最小值的那个。

• 如果一个线程设置信号的动作为“忽略”，它并没有规定是立即丢弃产生的信号还是保持挂起状态。

虽然POSIX允许线程或进程处理异步事件，但必须小心，因为一些POSIX系统调用被称为
347 **不安全异步发信号**（async-signal unsafe）和**不安全异步撤消**（async-cancel unsafe）。如果一个信号中断了一个由信号捕获函数调用的不安全异步函数，其结果是没有定义的。例如，使用信号处理程序中的函数pthread_cond_signal是不安全的，因为它引入了与函数pthread_cond_wait的竞争条件。

10.6.7 POSIX和原子动作

假定原子动作里面的活动之间交互密切，认为动作是发生在POSIX线程之间比认为是发生在POSIX进程之间更合适。至少有两个方法实现线程之间的类似原子动作的结构:

1）用信号、setjmp和longjmp的组合编程实现必需的协调。但是，longjmp和所有的线程系统调用是异步不安全的。这意味着单个处理程序不能调用它们。

2）使用线程创建和撤消编程实现所需的恢复。因为POSIX线程的设计很便宜，这种方法没有与更重量级进程结构同样的性能损失。

由于使用恢复模型，所以需要这些方法。如果支持终止模型，就可得到一个更简单的结

构。与此有关的内容将在10.8节Ada的背景下和10.9节实时Java的背景下讨论。

10.7 实时Java中的异步事件处理

在实时Java中与POSIX信号等价的是异步事件。实际上，当实时Java在一个遵循POSIX的操作系统上实现时，有一个类`POSIXSignalHandler`，它使POSIX信号能被映射到实时Java事件（见15.5.2节）。

程序10-2展示实时Java中与异步事件相关的三个主要的类。每个`AsyncEvent`可以有一个或多个`AsyncEventHandler`。当事件发生时（通过对`fire`方法的调用指出），根据它们的`SchedulingParameters`，调度所有和事件相关的处理程序去执行——见13.14.3节。通过使用`bindTo`方法，也可以把事件的激发（fire）与依赖于实现的外部动作的发生关联起来。

每个处理程序为每个未完成的事件激发被调度一次。然而，处理程序使用类`AsyncEventHandler`中的方法能够修改未完成事件的数目。

虽然事件处理程序是一个可调度的实体，但它的目标是不蒙受应用程序线程那样多的开销。因此，不能假定每个处理程序有一个独立的实现线程，因为与一个特定的实现线程相关联的处理程序可能不止一个。如果需要一个专用线程，应该使用`BoundAsyncEventHandler`。 348

程序10-2 类AsyncEvent、AsyncEventHandler和BoundAsyncEventHandler

```
public class AsyncEvent
{
  public AsyncEvent ();

  public synchronized void addHandler (AsyncEventHandler handler);
  public synchronized void removeHandler (AsyncEventHandler handler);
  public void setHandler (AsyncEventHandler handler);
  // 将一个新的处理程序同此事件关联起来
  // 清除所有已存在的处理程序

  public void bindTo (java.lang.String happening);
  // 同外部事件绑定

  public ReleaseParameters createReleaseParameters ();
  // 创建一个 ReleaseParameters 对象，表示这个事件的特征

  public synchronized void fire ();
  // 为此事件执行处理程序集合中的方法run ()
  public boolean handledBy (AsyncEventHandler target);
  // 若此事件由此处理程序处理，返回true
}

public abstract class AsyncEventHandler implements Schedulable
{
  public AsyncEventHandler ();
  // 参数是从当前线程继承的

  public AsyncEventHandler (SchedulingParameters scheduling,
         ReleaseParameters release, MemoryParameters memory,
         MemoryArea area, ProcessingGroupParameters group);
  ... // 其他可用构造器
```

```
  // 实现 Schedulable 接口的方法，见第13章

  protected final synchronized int getAndClearPendingFireCount ();
  //  通过原子动作将此处理程序的悬挂执行个数置为0,
  //  并在清除前将此数值返回
  protected synchronized int getAndDecrementPendingFireCount ();
  protected synchronized int getAndIncrementPendingFireCount ();

  public abstract void handleAsyncEvent ();
  // 覆盖此方法，以定义这个处理程序执行的动作

  public final void run ();
}

public abstract class BoundAsyncEventHandler extends AsyncEventHandler
{
  public BoundAsyncEventHandler ();
  // 其他构造器
}
```

10.8 Ada中的异步控制转移

在POSIX中，通过在程序中的合适点上阻塞信号和解除信号阻塞，可以为信号处理程序设置定义域。然而，如果没有语言的支持，这会变成非结构化的和易出错的。Ada提供了一个异步通知处理的更结构化的形式，叫**异步控制转移**（asynchronous transfer of control，ATC）。此外，为了强调ATC是一种通信和同步的形式，ATC在任务间通信设施之上建立这种机制。

Ada的select语句已在第9章介绍过了。它有如下形式:

- 一个选择性的接受（为了支持会合的服务器方）——在9.4.2节讨论过;
- 一个限时和条件入口调用（对任务或保护入口）——将在12.4.2节讨论;
- 一个异步选择——在这里讨论。

异步选择语句用终止语义提供异步通知机制。

异步选择的执行以发出触发入口调用或发出触发延迟开始。如果触发语句是一个入口调用，首先像平常一样对参数求值，然后发出调用。如果该调用被排队，就执行在可中止部分的语句序列。

如果触发语句在可中止部分的执行结束前完成，则可中止部分被中止。当这些活动完成的时候，执行触发语句后面的可选语句序列。

如果可中止部分在入口调用结束前完成，就试图撤消入口调用，如果成功，就结束异步选择语句的执行。下面说明它们的语法:

349~350

```
select
  Trigger.Event;
  -- 在事件被接受之后执行的可选的语句序列
then abort
  -- 可中止的语句序列
end select;
```

注意，触发语句可以是延迟语句，因此，可中止部分可能与超时相关联（见12.4.3）。

如果触发事件的撤消因为保护动作或会合已经启动而失败，那么异步选择语句将在执行

触发语句后面的可选语句序列之前等待触发语句完成。

显然，即使在可中止部分开始执行之前，仍有可能发生触发事件。在这种情况下不执行可中止部分，可中止部分也因此不中止。

研究下面的例子：

```
task Server is
    entry Atc_Event;
end Server;

task To_Be_Interrupted;

task body Server is
begin
  ...
  accept Atc_Event do
    Seq2;
  end Atc_Event;
  ...
end Server;

task body To_Be_Interrupted is
begin
  ...
  select    -- ATC 语句
    Server.Atc_Event;
    Seq3;
  then abort
    Seq1;
  end select;
  Seq4;
  ...
end To_Be_Interrupted;
```

当上面的ATC语句执行时，执行哪一个语句要依赖于事件发生的顺序：

```
if 会合立即可用 then
   发出Server.Atc_Event
   执行Seq2
   执行Seq3
   执行Seq4 ( Seq1从不启动)
elsif 在 Seq1 完成前无会合开始then
   发出Server.Atc_Event
   执行Seq1
   取消Server.Atc_Event
   执行Seq4
elsif 在 Seq1 完成前会合完成 then
   发出Server.Atc_Event
   Seq1的部分执行同 Seq2并发地发生
   Seq1 被中止和终了化 (finalised)
   执行Seq3
   执行Seq4
else (在Seq1 完成后会合完成)
   发出Server.Atc_Event
```

351

```
    Seq1同Seq2的部分执行并发地执行
    尝试撤消Server.Atc_Event，但不成功
    Seq2 的执行完成
    执行Seq3
    执行Seq4
end if
```

注意，在Seq1完成和会合完成之间，有一个竞争条件存在。当Seq1确实完成但还是被中止时就是这种情形。

Ada允许一些操作**延期中止**（abort deferred）。如果Seq1包含一个延期中止操作，直到操作完成才会发生它的撤消。对一个保护对象的调用就是这种操作的一个例子。

上面的讨论集中于Seq1和触发会合的并发行为。实际上，在多处理器实现的情况下，Seq1和Seq2有可能并行执行。然而，在单处理器系统中，只有在动作导致触发事件的优先级比Seq1更高的情况下，触发事件才会发生。一般情况下的行为是Seq2抢占Seq1。当Seq2（触发会合）完成时，Seq1将在它能再次执行前中止。因此ATC是“立即的”(除非延期中止操作正在进行中)。

10.8.1 异常和ATC

对异步选择语句来说，有两个活动可能并发地发生：可中止部分可能和触发动作并发执行（当动作是一个入口调用时）。这些活动中的任一个都可能引发异常，并且不处理异常。因此，乍一看可能同时从选择语句传播两个异常。然而实际情况并不是这样：异常之一会被丢
352 弃（当可中止部分中止时产生的那个异常），因此只传播一个异常。

10.8.2 Ada和原子动作

在6.5节阐述过，异常处理可以实现顺序系统中的向后出错恢复。在本节，利用Ada 的ATC设施和异常处理来实现向后和向前出错恢复。假定底层的Ada实现和运行时支持系统是无故障的，因此Ada提供的强类型将保证Ada程序本身是可行的。

1. 向后出错恢复

下面的软件包是在6.5节中为了保存和恢复任务的状态而给出的软件包的类属版本。

```
generic
  type Data is private;
package Recovery_Cache is
  procedure Save (D : in Data);
  procedure Restore (D : out Data);
end Recovery_Cache;
```

考虑希望进入一个可恢复原子动作的三个Ada任务。每个任务调用下述包中给出的合适的过程。

```
package Conversation is

  procedure T1 (Params : Param);  -- 任务 1调用
  procedure T2 (Params : Param);  -- 任务 2调用
  procedure T3 (Params : Param);  -- 任务 3调用

  Atomic_Action_Failure : exception;

end Conversation;
```

包体封装动作，并确保对话期间只允许在这三个任务之间进行通信[⊖]。Controller保护对象负责把一个任务中的出错条件传播到所有的任务，在恢复缓存里保存和恢复持久的数据，确保所有的任务同时离开这个动作。它包括三个保护入口和一个保护过程。

- 入口Wait_Abort表示异步事件，当任务执行它们的动作部分时，将在这个事件上等待。 353
- 如果每个任务无错地完成其动作部分，它就调用Done。只有当三个任务都调用了Done后，才允许它们离开。
- 类似地，如果不得不执行恢复，每个任务都调用Cleanup。
- 如果任何任务发现了一个出错条件（或者因为引发异常，或者因为接受测试失败），它将调用Signal_Abort。这将设置Killed为真，表示必须恢复任务。

注意，当执行向后出错恢复时，任务并不关心出错的真正原因。当Killed变成真以后，动作中的所有任务都收到异步事件。一旦事件得到处理，所有的任务必须在Cleanup入口等待，以便它们同时终止会话模块。

```
with Recovery_Cache;
package body Conversation is

  Primary_Failure, Secondary_Failure,
          Tertiary_Failure: exception;
  type Module is (Primary, Secondary, Tertiary);

  protected Controller is
    entry Wait_Abort;
    entry Done;
    entry Cleanup;
    procedure Signal_Abort;
  private
    Killed : Boolean := False;
    Releasing_Done : Boolean := False;
    Releasing_Cleanup : Boolean := False;
    Informed : Integer := 0;
  end Controller;

  -- 可能有的动作间通信用的局部保护对象


  protected body Controller is
    entry Wait_Abort when Killed is
    begin
      Informed := Informed + 1;
      if Informed = 3 then
        Killed := False;
        Informed := 0;
      end if;
    end Wait_Abort;
```

⊖ 实际上，由于Ada的作用域规则，这一点难以确保。提高安全性的一条途径是要求包Conversation放在库一级，并且它的包体只引用纯粹（无状态)的包。这里提出的解决方案假设任务是行为良好的。为了简单，它还假设在正确的时刻正确的任务调用T1、T2和T3。

```
        procedure Signal_Abort is
        begin
          Killed := True;
        end Signal_Abort;

        entry Done when Done' Count = 3 or Releasing_Done is
        begin
354       if Done' Count > 0 then
            Releasing_Done := True;
          else
            Releasing_Done := False;
          end if;
        end Done;

        entry Cleanup when Cleanup' Count = 3 or Releasing_Cleanup is
        begin
          if Cleanup' Count > 0 then
            Releasing_Cleanup := True;
          else
            Releasing_Cleahup := False;
          end if;
        end Cleanup;
      end Controller;

      procedure T1 (Params : Param) is separate;
      procedure T2 (Params : Param) is separate;
      procedure T3 (Params : Param) is separate;

    end Conversation;
```

每个任务的代码包含在单个过程里：例如T1。在这个过程里，三次尝试执行动作。如果所有的尝试失败，就引发异常Atomic_Action_Failure。每个尝试被一个保存和恢复状态的调用所包围（如果尝试失败的话）。每个尝试被封装在一个独立的局部过程内（T1_ Primary等），此过程包含一个用控制器执行所需协议的“选择然后中止”语句。每个任务用恢复缓冲保存本地数据。

```
separate (Conversation)
procedure T1 (Params : Param) is
  procedure T1_Primary is
  begin
    select
      Controller.Wait_Abort; -- 触发事件
      Controller.Cleanup; -- 等待全体完成
      raise Primary_Failure;
    then abort
      begin
        -- 实现原子动作的代码
        -- 接受测试可能引发异常
        if Accept_Test = Failed then
          Controller.Signal_Abort;
        else
          Controller.Done; -- 完成信号
        end if;
```

```
        exception
          when others =>
            Controller.Signal_Abort;
        end;
      end select;
    end T1_Primary;                                                    355
    procedure T1_Secondary is ...;
    procedure T1_Tertiary is ...;

    package My_Cache is new Recovery_Cache (. .); -- 用于局部数据

  begin
    My_Cache.Save (. . );
    for Try in Module loop
      begin
        case Try is
          when Primary => T1_Primary; return;
          when Secondary => T1_Secondary; return;
          when Tertiary => T1_Tertiary;
        end case;
      exception
        when Primary_Failure =>
          My_Cache.Restore (..);
        when Secondary_Failure =>
          My_Cache.Restore (..);
        when Tertiary_Failure =>
          My_Cache.Restore (..);
          raise Atomic_Action_Failure;
        when others =>
          My_Cache.Restore (..);
          raise Atomic_Action_Failure;
      end;
    end loop;
  end T1;

  -- 类似地对T2和T3
```

图10-3说明了参与一个会话的任务的简单状态迁移图。

2. 向前出错恢复

Ada的ATC设施与异常一起用来实现在并发执行任务间带有向前出错恢复功能的原子动作。再次分析下面在三个任务间实现一个原子动作的软件包。

```
package Action is

  procedure T1 (Params : Param); -- 由任务1调用
  procedure T2 (Params : Param); -- 由任务2调用
  procedure T3 (Params : Param); -- 由任务3调用

  Atomic_Action_Failure : exception;
end Action;
```

如同向后出错恢复一样，包体封装动作并且确保只允许在三个任务之间进行通信。保护对象Controller负责将一个任务中引发的异常传播到所有任务，并负责确保所有的任务在同一时刻离开动作。 356

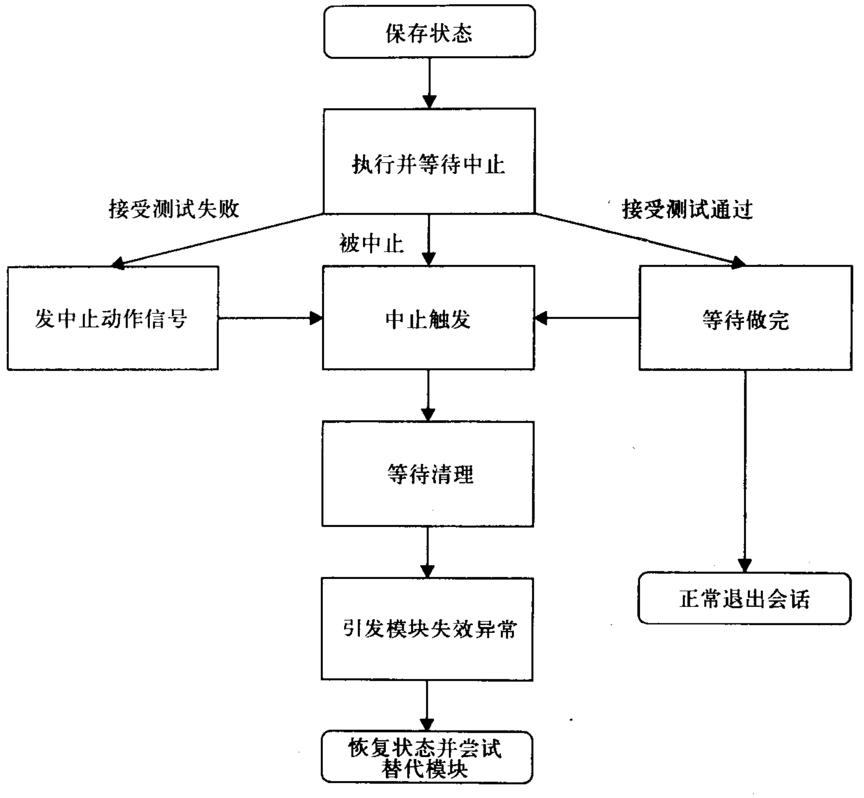


图10-3 会话的简单状态迁移图

```
with Ada.Exceptions;
use Ada.Exceptions;
package body Action is

  type Vote_T is (Commit, Aborted);
  protected Controller is
    entry Wait_Abort (E: out Exception_Id);
    entry Done;
    procedure Cleanup (Vote: Vote_T);
    entry Wait_Cleanup (Result : out Vote_T);
    procedure Signal_Abort (E: Exception_Id);
  private
    Killed : Boolean := False;
    Releasing_Cleanup : Boolean := False;
    Releasing_Done : Boolean := False;
    Reason : Exception_Id;
    Final_Result : Vote_T := Commit;
    Informed : Integer := 0;
  end Controller;

  -- 可能有的动作间通信用的局部保护对象

  protected body Controller is
    entry Wait_Abort (E: out Exception_Id) when Killed is
    begin
```

357

```
      E := Reason;
      Informed := Informed + 1;
      if Informed = 3 then
        Killed := False;
        Informed := 0;
      end if;
    end Wait_Abort;

    entry Done when Done' Count = 3 or Releasing_Done is
    begin
      if Done' Count > 0 then
        Releasing_Done := True;
      else
        Releasing_Done := False;
      end if;
    end Done;

    procedure Cleanup (Vote: Vote_T) is
    begin
      if Vote = Aborted then
        Final_Result := Aborted;
      end if;
    end Cleanup;

    procedure Signal_Abort (E: Exception_Id) is
    begin
      Killed := True;
      Reason := E;
    end Signal_Abort;

    entry Wait_Cleanup (Result: out Vote_T)
          when Wait_Cleanup' Count = 3 or Releasing__Cleanup is
    begin
      Result := Final_Result;
      if Wait_Cleanup' Count > 0 then
        Releasing_Cleanup := True;
      else
        Releasing_Cleanup := False;
        Final_Result := Commit;
      end if;
    end Wait_Cleanup;
  end Controller;

  procedure T1 (Params: Param) is
    X : Exception_Id;
    Decision : Vote_T;
  begin
    select
      Controller.Wait_Abort (X); -- 触发事件
      Raise_Exception (X); -- 引发公共异常
    then abort
      begin
        -- 实现原子动作的代码
```

```
            Controller.Done; -- 完成信号
          exception
            when E: others =>
              Controller.Signal_Abort (Exception_Identity (E) );
          end;
        end select;
      exception
        -- 如果在动作执行期间引发了任何异常
        -- 所有任务必须参与恢复
        when E: others =>
          -- Exception_Identity (E) 已经在所有任务中引发
          -- 处理异常
          if Handled_Ok then
            Controller.Cleanup (Commit);
          else
            Controller.Cleanup (Aborted);
          end if;
          Controller.Wait_Cleanup (Decision);
          if Decision = Aborted then
            raise Atomic_Action_Failure;
          end if;
      end T1;

      procedure T2 (Params : Param) is ...;

      procedure T3 (Params : Param) is ...;

    end Action;
```

每个动作部件（T1、T2和T3）有相同的结构。部件执行带可中止部分的选择语句。如果有任何部件指出一个异常已被引发并且未被任何部件局部处理，则保护对象Controller就发送信号给触发事件。可中止部分包含部件的实际代码。如果此代码无错地执行，Controller就会得到通知：这个部件准备执行动作。如果在可中止部分发生了任何异常，Controller就会得到通知并传送异常的身份。与向后出错恢复（在前面的小节中讲述过）不同的是，必须说明出错的原因。

358~359 如果Controller收到一个未处理异常的通知，它将释放所有在Wait_Abort触发事件处等待的任务（迟到的任务一尝试进入其选择语句就会立即收到这个事件）。这些任务中止它们的可中止部分（如果已开始），并且由向控制器发出入口调用语句的后面那个语句在每个任务中引发这个异常。如果这个部件成功地处理了这个异常，任务指出准备执行这个动作。否则，任务就指出这个动作必须被中止。如果有任何任务指出这个动作必须被中止，那么所有的任务将引发Atomic_Action_Failure异常。图10-4表示了使用一个简单状态迁移图的方法。

上面的例子说明用Ada编写带有向前出错恢复的原子动作是可行的。然而，对上面的例子必须注意两点：

- 只有要传递给Controller的第一个异常才在所有的任务中引发。不可能并发地引发异常，因为在可中止部分引发的异常在被中止时丢弃了。
- 这种方法不处理离弃者问题。如果动作参与者中的一个没有到达，别的参与者就在这个动作的后面等待。为了应付这种情况，有必要让每个任务在动作控制器里登记它的到达。

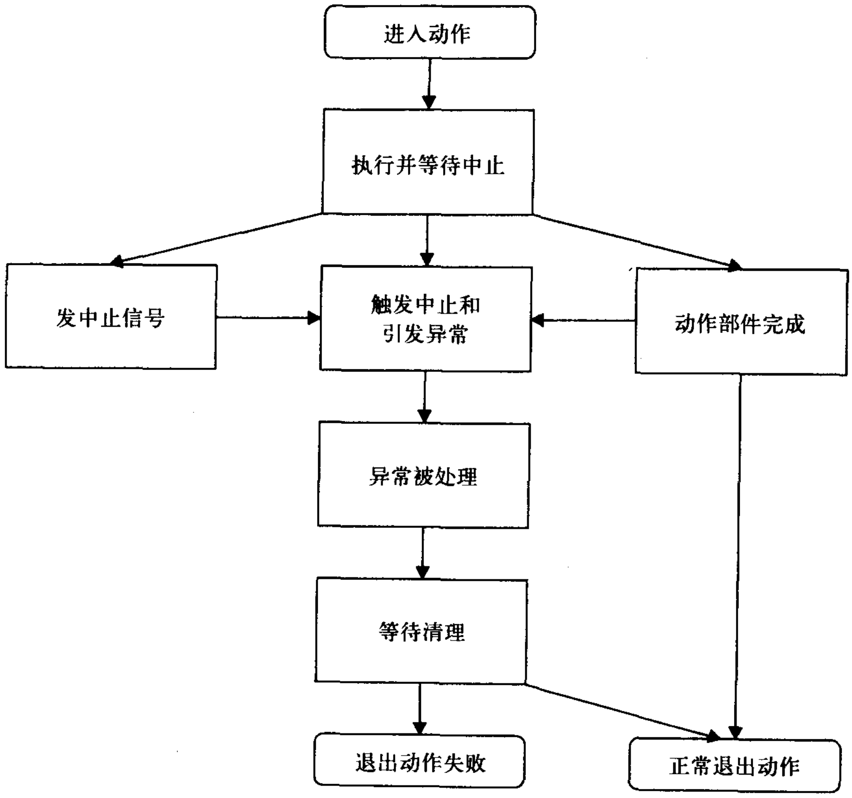


图10-4 说明向前出错恢复的简单状态迁移图

10.9 实时Java中的异步控制转移

Java的早期版本允许一个线程用下面的方法异步地作用于另一个线程。

```
public final void suspend () throws SecurityException;
public final void resume () throws SecurityException;

public final void stop () throws SecurityException;
public final void stop (Throwable except) throws SecurityException;
```

在7.3.7节讨论过方法suspend和resume。方法stop使线程停止它的当前活动，并抛出一个ThreadDeath异常，方法stop (Throwable except) 与此类似，只是在这时将作为参数传递的异常抛出。

上面的方法现在都已经过时，因此不应再使用。标准Java现在只支持下面的方法:

```
public void interrupt () throws SecurityException;
public boolean isInterrupted ();
public void destroy ();
```

一个线程能通过调用interrupt方法向另一个线程发出中断信号。这种动作的结果依赖于被中断线程的当前状态。

- 如果被中断线程阻塞在wait、sleep或join方法里面，信号的到来使该线程变成可执行的，并抛出InterruptedException。

- 如果被中断线程正在执行，就设置一个标记表示中断未完成。这对被中断的线程没有立即的影响。相反，被调用线程必须进行周期性的测试，以检查被中断线程是否已经被isInterrupted方法中断。

这种做法本身不能满足在10.5.1节中概述的用户需要。

destroy方法和Ada中止设施相似，它破坏线程，且不做任何清理。

实时Java基于异步控制转移（ATC）为中断线程提供替代方法。实时Java ATC模型和Ada的模型相似，在Ada中必须表明哪些代码能接收ATC请求。然而，实时Java模型在两个重要方面与Ada不同。

360～361 1）实时Java模型是与Java的异常处理设施集成在一起的，而Ada模型是同选择语句和入口处理机制集成在一起的。

2）实时Java模型要求每个方法表明它准备允许ATC发生。ATC被延期，直到线程正在这样一个方法里面执行。相反，如果从select-then-abort语句调用了一个子程序，Ada的默认反应是允许ATC，必须显式地处理一个延期的响应。

在和另外的线程/任务交互期间（例如Java中的同步语句/方法和Ada中的保护动作与会合）或在构造器和终了化子句（finally）中，两种语言都延期ATC。

实时Java ATC模型把Java异常处理模型和线程中断扩展结合在一起。最好把这个模型解释成两步。第一步是低级支持和整体方法，第二步是用高级支持提供一个处理ATC的结构化方法。使用基本的ATC设施要求三个活动:

1）声明一个AsynchronouslyInterruptedException（AIE）

2）标识可以中断的方法

3）给线程发送AsynchronouslyInterruptedException信号。

程序10-3展示了类AsynchronouslyInterruptedException的规格说明。这些方法将在适当的地方解释，现在所需要的是知道每个线程都对应一个类属AIE。

程序10-3 实时Java的类AsynchronouslyInterruptedException

```
public class AsynchronouslyInterruptedException extends
            Java.lang.InterruptedException
{
  public AsynchronouslyInterruptedException ();

  public synchronized boolean disable ();
  // 只在 doInterruptible里面才是合法的，如果成功，返回 true
  public boolean doInterruptible (Interruptible logic);
  // 在任何时刻，每个线程只有一个特定的Interruptible可以运行
  // 如果这个Interruptible被执行，返回true,
  // 如果为此线程已经有一个Interruptible在进行，返回false

  public synchronized boolean enable ();
  public synchronized boolean fire ();

  public boolean happened (boolean propagate);

  public static AsynchronouslyInterruptedException getGeneric ();
  // 返回 AsynchronouslyInterruptedException
  // 它是在RealtimeThread.interrupt () 被调用时生成的
```

```
  public boolean isEnabled ();

  public void propagate ();
}
```

AIE可以放在和方法相关联的throws列表中。例如，分析下面的类，它用包提供了一个可中断服务，而包声明了不可中断的服务（也就是，AsynchronouslyInterrupted Exception没有被包含在这些服务的throws列表中）。

```
import nonInterruptibleServices.*;

public class InterruptibleService
{
  public AsynchronouslyInterruptedException stopNow =
         AsynchronouslyInterruptedException.getGeneric ();

  public boolean Service () throws AsynchronouslyInterruptedException
  {
    // 散布有调用NonInterruptibleServices的代码
  }
}
```

现在假定一个实时线程t已经调用了这个类的一个实例以提供Service:

```
public InterruptibleService IS = new InterruptibleService ();

// 线程t的代码
if (IS.Service () ) { ... }
else {...};
```

而另一个线程中断t:

```
t.interrupt ();
```

这个调用的结果依赖于调用发生时t的当前状态。

- 如果在任何时刻t都在一个ATC延期段里面执行，将AsynchronouslyInterrupted-Exception标记为悬挂的。当t一旦离开ATC延期区域并且在throws列表里声明了AsynchronouslyInterruptedException的方法里面执行，就抛出这个异常。
- 如果t正在一个throws列表里没有声明AsynchronouslyInterruptedException的方法里面执行（例如在包nonInterruptibleServices里的方法），则将此异常标记为悬挂的。当t一返回（或调用）到一个在其throws列表里声明了Asynchronously-InterruptedException的方法，就抛出这个异常。 362~363
- 如果t正在一个方法的try块里面执行，而这个方法在其throws列表里声明了AsynchronouslyInterruptedException，那么就终止这个try块，并且控制转移到catch子句的第一个语句。如果没有发现合适的catch子句，就传播Asynchronously-InterruptedException到发出调用的那个方法。否则，执行合适的处理程序，并且完成AsynchronouslyInterruptedException处理（除非从处理程序里传播出AIE）。
- 如果t在一个方法的try块之外执行，而这个方法在其throws列表里声明了Asynchr-onouslyInterruptedException，那么就终止这个方法，并且立即在发出调用的那个方法里抛出AsynchronouslyInterruptedException。
- 如果t被阻塞在从一个方法里调用的wait、sleep或join方法里面，而这个方法在其

throws列表里声明了AsynchronouslyInterruptedException，就重新调度t，并且抛出AsynchronouslyInterruptedException。

- 如果t被阻塞在从一个方法里调用的wait，sleep或join方法里，而这个方法在其throws列表里没有声明AsynchronouslyInterruptedException，就重新调度t，并且将AsynchronouslyInterruptedException标记为悬挂的。当t一返回到在其throws列表里声明了AsynchronouslyInterruptedException的方法，就抛出这个异常。

一旦抛出了ATC，控制就转移到适当的异常处理程序，并有必要确定捕获的ATC是否就是中断线程期望的那一个。如果它是，就能处理异常。如果不是，应该把异常传播给发出调用的方法。在类AsynchronouslyInterruptedException中定义的happened方法用于此目的。研究下面的代码：

```
import NonInterruptibleServices.*;
public class InterruptibleService
{
  public AsynchronouslyInterruptedException stopNow =
          new AsynchronouslyInterruptedException ();

  public boolean Service () throws AsynchronouslyInterruptedException
  {
    try {
      // 分散的调用NonInterruptibleServices的代码
    }
    catch AsynchronouslyInterruptedException AIE {
      if (stopNow.happened (true) ) {
        // 处理 ATC
      }
      // 无else子句，参数true指出：
364   // 如果当前异常不是stopNow，就应该被立即传播到调用方法
    }
  }
}
```

这里，当AIE被抛出以后，控制传递给try块终点的catch子句。为AsynchronouslyInterruptedException找到一个处理程序。为了决定当前的AsynchronouslyInterruptedException是否是stopNow，调用方法stopNow.happened。如果stopNow是当前异常，调用返回真。如果它不是当前异常，那么当happened的参数为true时，就传播异常。如果参数为false，控制返回到catch语句，并带一个false值。在用propagate方法传播异常之前，线程可能要执行一些清理例程：

```
catch AsynchronouslyInterruptedException AIE {
  if (stopNow.happened (false) ) {
    // 处理ATC
  } else {
    // 清理
    AIE.propagate ();
  }
}
```

1. Interruptible接口

上面的讨论说明了实时Java为处理ATC提供的基本机制。为了便于它们的结构化使用，语

言提供了一个名为Interruptible的接口——见程序10-4。

程序10-4 实时Java的Interruptible接口

```
public interface Interruptible
{
  public void interruptAction (
              AsynchronouslyInterruptedException exception);

  public void run (AsynchronouslyInterruptedException exception)
         throws AsynchronouslyInterruptedException;

}
```

一个希望提供可中断方法的对象通过实现Interruptible接口来达到目的。run方法就是可中断的方法，如果run方法被中断了，系统就调用方法interruptAction。

一旦实现了这个接口，就能把这个实现作为参数传递给类AsynchronouslyInterruptedException中的doInterruptible方法。调用类AsynchronouslyInterruptedException中的fire方法，能中断doInterruptible方法。方法disable、enable和isEnabled实现对类AsynchronouslyInterruptedException的进一步控制。禁用的AsynchronouslyInterruptedException一直延期到被启用。使用后者的例子在10.9.1节给出。

注意，对于一个特定的AsynchronouslyInterruptedException，每次只有一个doInterruptible方法是活动的。如果调用未完成，此方法立即返回一个false值。 365

2. 多重AsynchronouslyInterruptedException

既然AsynchronouslyInterruptedException能延期，多重ATC也可能延期。这种情况可能出现在一个类（它实现了接口Interruptible）的run方法在AIE上调用方法doInterruptible的时候。相关联的run方法也可能调用另一个doInterruptible。因此一个线程有可能执行嵌套的doInterruptible。研究下面的例子:

```
import javax.realtime.*;

public class NestedATC
{
  AsynchronouslyInterruptedException AIE1 = new
           AsynchronouslyInterruptedException ();
  AsynchronouslyInterruptedException AIE2 = new
           AsynchronouslyInterruptedException ();
  AsynchronouslyInterruptedException AIE3 = new
           AsynchronouslyInterruptedException ();

  public void method1 ()
  {
    // ATC延期区
  }

  public void method2 () throws AsynchronouslyInterruptedException
  {
    AIE1.doInterruptible
       (new Interruptible ()
```

```
            {
              public void run (AsynchronouslyInterruptedException e)
                    throws AsynchronouslyInterruptedException
              {
                method1 ();
              }
              public void interruptAction (
                    AsynchronouslyInterruptedException e)
              {
                if (AIE1.happened (false) ) {
                   // 在这里恢复
                } else {
                 //清理
                 e.propagate ();
              }
            }
          }
        );
}
public void method3 () throws AsynchronouslyInterruptedException
{
  AIE2.doInterruptible
      (new Interruptible ()
        {
           public void run (AsynchronouslyInterruptedException e)
                 throws AsynchronouslyInterruptedException
           {
             method2 ();
           }
           public void interruptAction (
                 AsynchronouslyInterruptedException e)
           {
             if (AIE2.happened (false) ) {
                // 在这里恢复
             } else {
              // 清理
              e.propagate ();
           }
         }
       }
     );
}
public void method4 () throws AsynchronouslyInterruptedException
{
  AIE3.doInterruptible
      (new Interruptible ()
        {
           public void run (AsynchronouslyInterruptedException e)
                 throws AsynchronouslyInterruptedException
           {
```

```
                method3 ();
              }
              public void interruptAction (
                    AsynchronouslyInterruptedException e)
              {
                if (AIE3.happened (false) ) {
                  // 在这里恢复
                } else {
                 // 清理
                 e.propagate ();
              }
            }
          }
        );
      }
    }
```

现在假定线程t创建了一个NestedATC实例，并且调用method4，method4调用method3，method3调用method2，method2调用method1，而method1是ATC延期区。假定一个对AIE2.fire ( ) 的调用中断了此线程，这是拥有式悬挂。因为AIE3处于嵌套的更外层，如果AIE3现在进入，那么丢弃AIE2。如果AIE1进入，那么丢弃AIE1（因为它处于更低一级）。一旦method1返回，就抛出当前悬挂的AIE。

实时Java和原子动作

本节说明怎样用实时Java的ATC设施实现带有向前出错恢复的原子动作。

首先，与AtomicActionFailure异常一起定义AtomicActionException。

```
import javax.realtime.AsynchronouslyInterruptedException;

public class AtomicActionException extends
             AsynchronouslyInterruptedException
{
  public static Exception cause;
  public static boolean wasInterrupted;
}

public class AtomicActionFailure extends Exception;
```

使用一个和前面定义相似的ThreeWayRecoverableAtomicAction:

```
public interface ThreeWayRecoverableAtomicAction {
  public void role1 () throws AtomicActionFailure;
  public void role2 () throws AtomicActionFailure;
  public void role3 () throws AtomicActionFailure;
}
```

366~368

可以用与Ada相似的结构实现一个类RecoverableAction。

```
import javax.realtime.*;

public class RecoverableAction
             implements ThreeWayRecoverableAtomicAction
{
  protected RecoverableController Control;
```

```
private final boolean abort = false;
private final boolean commit = true;

private AtomicActionException aae1, aae2, aae3;

public RecoverableAction () // 构造器
{
  Control = new RecoverableController ();
    // 用于恢复
    aae1 = new AtomicActionException ();
    aae2 = new AtomicActionException ();
    aae3 = new AtomicActionException ();
}

class RecoverableController {
  protected boolean firstHere, secondHere, thirdHere;
  protected int allDone;
  protected int toExit, needed;
  protected int numberOfParticipants;
  private boolean committed = commit;

  RecoverableController ()
  {
    // 用于同步
    firstHere = false;
    secondHere = false;
    thirdHere = false;
    allDone = 0;
    numberOfParticipants = 3;
    toExit = numberOfParticipants;
    needed = numberOfParticipants;
  }

  Synchronized void first () throws InterruptedException
  {
    while (firstHere) wait ();
    firstHere = true;
  }

  synchronized void second () throws InterruptedException
  {
    while (secondHere) wait ();
    secondHere = true;
  }

  synchronized void third () throws InterruptedException
  {
    while (thirdHere) wait ();
    thirdHere = true;
  }

  synchronized void signalAbort (Exception e)
  {
    allDone = 0;
    AtomicActionException.cause = e;
```

```
    AtomicActionException.wasInterrupted = true;
    // 在所有参与者中引发 AsynchronouslyInterruptedException
    aae1.fire ();
    aae2.fire ();
    aae3.fire ();
  }

  private void reset ()
  {
    firstHere = false;
    secondHere = false;
    thirdHere = false;
    allDone = 0;
    toExit = numberOfParticipants;
    needed = numberOfParticipants;
    notifyAll ();
  }

  synchronized void done () throws InterruptedException
  {
    allDone++;
    if (allDone == needed) {
      notifyAll ();
    } else while (allDone ! = needed) {
        wait ();
        if (AtomicActionException.wasInterrupted)
        {
          allDone--;
        return;
        }
    }
    toExit--;
    if (toExit == 0) {
      reset ();
    }
  }

  synchronized void cleanup (boolean abort)
  {
    if (abort) { committed = false; };
  }

  synchronized boolean waitCleanup () throws InterruptedException
  {
    allDone++;
    if (allDone == needed) {
      notifyAll ();
    } else while (allDone != needed) {
        wait ();
    }
    toExit--;
    if (toExit == 0) {
      reset ();
```

```
    }
    return committed;
  };
};

public void rolel () throws AtomicActionFailure,
                            AsynchronouslyInterruptedException
{
  boolean Ok;
  // 入口协议
  // 直到原子动作里才有AIE
  boolean done = false;
  while (!done) {
    try {
      Control.first ();
      done = true;
    } catch (InterruptedException e) {
      // 忽略
    }
  }

  // 下面定义一个可中断代码段和代码被中断时调用的例程
  Ok = aael.doInterruptible
      (new Interruptible ()
        {
          public void run (AsynchronouslyInterruptedException e)
                throws AsynchronouslyInterruptedException
          {
            try {

              // 执行动作
              // 需要时调用 e.disable () 和 e.enable () 以延期AIE
              Control.done ();
            }
          catch (Exception x) {
            if (x instanceOf (AsynchronouslyInterruptedException))
              ((AsynchronouslyInterruptedException) x).propagate();
            else
              Control.signalAbort (x);
          }
        }

        public void interruptAction (
              AsynchronouslyInterruptedException e)
        {
          // 不需要什么动作
        }
      }
    };
if (!Ok) throw new AtomicActionFailure ();
  if (aael.wasInterrupted) {
    try {
      // 尝试恢复
```

```
          Control.cleanup (commit);
          if (Control.waitCleanup () != commit) {
            throw new AtomicActionFailure ();
          };
        }
        catch (Exception x) {
          throw new AtomicActionFailure ();
        }
      };
    }

    public void role2 () throws AtomicActionFailure,
                                AsynchronouslyInterruptedException
    {// 类似于 role1 };

    public void role3 () throws AtomicActionFailure,
                                AsynchronouslyInterruptedException
    {// 类似于 role1 };

  }
```

小结

如果将实时嵌入式系统用于关键性应用，进程的可靠执行是最根本的。当进程间交互时，必须限制进程间通信，以便在需要时能编写恢复过程。本章把原子动作做为一种机制讨论，通过这种机制，由许多任务组成的程序能够结构化，从而便于损害隔离和出错恢复。

动作是原子的，如果它们对其他进程而言是不可分的和瞬时的，使得对系统产生的效果就像是它们被穿插执行而非并发的。原子动作有定义良好的边界，并且能嵌套。原子动作中使用的资源在初始增长阶段分配，并作为后续收缩阶段的一部分释放，或在动作结束时释放（如果动作是可恢复的）。

原子动作的语法可以用动作语句表达。下列在进程P_1中执行的语句表示P_1希望和P_2、P_3一起进入原子动作。

```
action A with (P₂, P₃) do
    -- 语句序列
end A;
```

P_2和P_3必须执行相似的语句。 369~372

会话是带有向后出错恢复设施的原子动作（以恢复块的形式）。

```
action A with (P₂, P₃) do
   ensure <接受测试>
   by
      -- 基本模块
   else by
      -- 替代模块
   else error
end A;
```

在会话的入口保存进程的状态。在会话内部，只允许进程同会话内别的活动进程和通用资源管理器通信。为了离开会话，会话内的所有活动进程必须通过接受测试。如果有任何一

个进程没有通过接受测试，所有的进程都恢复到会话开始时保存的状态，并且执行它们的替代模块。会话可以嵌套，并且如果内部会话里的所有替代模块失效，就必须在外层执行恢复。

会话机制是有限的，因为当有一个会话失败时，所有的进程都必须恢复，并都进入它们的替代模块。这强迫相同的进程为了达到期望的效果而再次通信，进程不能打断会话。然而，经常当一个进程通过与一组进程通信没能达到主模块内的目标时，它可能希望在次要模块内和另一组全新的进程通信。对话和会谈消除了会话的限制。

经由异常处理程序，也可以把向前出错恢复加到原子动作中。如果有一个进程引发了一个异常，那么动作中所有的活动进程必须处理这个异常。

```
action A with (P₂, P₃) do
   -- 动作
exception
   when exception_a =>
      -- 语句序列
   when others =>
      raise atomic_action_failure;
end A;
```

当使用这种方法时，必须解决的两个问题是：并发引发的异常的分辨和内部动作里的异常。

很少有主流的语言或操作系统直接支持原子动作或可恢复的原子动作的概念。然而，大多数通信和同步原语使得原子动作的隔离特性能够编程。为了实现可恢复的动作，要求实现
373 异步通知机制。异步通知机制可能有恢复语义（在这种情况下它称作异步事件处理机制）或终止语义（异步控制转移）。POSIX用信号和线程撤销机制支持异步事件。信号可以被处理、阻塞或忽略。实时Java也支持异步事件。

Ada和实时Java都提供异步控制转移的终止模型。Ada机制建立在选择语句之上。与之相比，实时Java的ATC集成到异常和线程中断机制里面。这些终止方法与异常结合在一起，就能够优雅地实现可恢复动作。

相关阅读材料

Anderson, T. and Lee, P. A. (1990) *Fault Tolerance Principles and Practice*, 2nd edn. Englewood Cliffs, NJ: Prentice Hall.

Bollella, G., Brosgol, B., Dibble, P., Furr, S., Gosling, J., Hardin, D. and Turnbull, M. (2000) *The Real-Time Specification for Java*. Reading, MA: Addison-Wesley.

Lynch, N. A. (ed.) (1993). *Atomic Transactions*. San Mateo, CA: Morgan Kaufmann.

Lea, D. (1999) *Concurrent Programming in Java: Design Principles and Patterns*. Reading: Addison-Wesley.

Northcutt, J. D. (1987) *Mechanisms for Reliable Distributed Real-time Operating Systems: The Alpha Kernel*. New York: Academic Press.

Shrivastava, S. K., Mancini, L. and Randell, B. (1987) *On the Duality of Fault Tolerant Structures*, Lecture Notes in Computer Science, Volume 309, pp. 19–37. Berlin: Springer-Verlag.

练习

10.1 区别原子动作和原子事务。原子事务和会话之间有什么关系？

10.2 把10.2.1节中原子动作的信号量实现从两进程的交互扩展为三进程的交互。

10.3 重写10.2.2节中的原子动作的管程实现，允许动作中的两个进程都是活动的。

10.4 重写10.2.3节中的`Action_X` Ada包，使它变成控制一个三任务对话的通用包（提示：用类属）。

10.5你给练习10.4的解决方案能扩展为对付参与原子动作的任意数量的任务吗？

10.6 将Ada扩展到能使一个任务引发另一个任务中的异常的含义是什么？ 374

10.7 将异步通知和异常处理进行比较和对照。

10.8 下面的代码说明了两个进程之间简单对话。试将它构造成会谈。

```
x, y, z : INTEGER;

PROCESS B;

PROCESS A;
BEGIN
  ...
  ACTION conversation (B) do
    ENSURE A_acceptance_test
    BY
      -- A_primary
      -- 使用 x, y
    ELSE BY
      -- A_secondary
      -- 使用 y, z
    ELSE
      ERROR
  END conversation;
  ...
END A;

PROCESS B;
BEGIN
  ...
  ACTION conversation (A) do
    ENSURE B_acceptance_test
    BY
      -- B_primary
      -- 使用 x, y
    ELSE BY
      -- B_secondary
      -- 使用 y, z
    ELSE
      ERROR
  END conversation;
  ...
END B;
```

10.9 在10.8.2节中说明了在三个Ada任务间的向后和向前出错恢复。说明怎样把它们组合为一个解决方案，从而在相同的三个任务间提供向前和向后两种出错恢复。

10.10 修改10.8节中的解决方案，使它能处理离弃者问题。

10.11 研究下面的四段代码:

375

```
-- 第1段
select
    T.Call; -- 对任务T的一个入口调用
    Flag := A;
or
    delay 10.0;
    Flag := B;
    -- 花2秒执行的代码
end select;

-- 第2段
select
    T.Call; -- 对任务T的一个入口调用
    Flag := A;
else
    delay 10.0;
    Flag := B;
    -- 花2秒执行的代码
end select;

-- 第3段
select
    T.Call; -- 对任务T的一个入口调用
    Flag := A;
then abort
    delay 10.0;
    Flag := B;
    -- 花2秒执行的代码
end select;

-- 第4段
select
    delay 10.0;
    Flag := A;
then abort
    T.Call; -- 对任务T的一个入口调用
    Flag := B;
    -- 花2秒执行的代码
end select;
```

同T.Call的会合花费5秒钟来执行。在下面各种情况下，每段代码执行以后，变量Flag的值是什么。你可以假定Flag赋值语句不花费执行时间。

(1) 当执行选择语句时，可以调用T.Call。

(2) 当执行选择语句时，T.Call不可调用，并且在随后的14秒内仍不可调用。

(3) 当执行选择语句时，T.Call不可调用，但在2秒后可调用。

(4) 当执行选择语句时，T.Call不可调用，但在8秒后可调用。

10.12 分析下面的包规格说明，它提供了一个过程，用以在一个大字符数组中搜索惟一的定长字符串。如果发现了，过程就返回该字符串的开始位置。

376

```
package Search_Support is
  type Array_Bounds is range 1 .. 1_000_000_000;
  type Large_Array is array (Array_Bounds) of Character;
  type Pointer is access Large_Array;

  type Search_String is new String (1.. 10);

  procedure Search ( Pt: Pointer;
            Lower, Upper: Array_Bounds;
            Looking_For : Search_String;
            Found : out Boolean;
            At_Location : out Array_Bounds);
end Search_Support;
```

有三个任务为同一字符串对数组执行并发的搜索，它们是从同一任务类型派生出来的:

```
task type Searcher (Search_Array: Pointer;
                    Lower, Upper: Array_Bounds) is
  entry Find (Looking_For : Search_String);
  entry Get_Result (At_Location : out Array_Bounds);
end Searcher;
```

发现的字符串经由一个与任务的初始会合传递。勾画出这个任务类型的体（和任何别的你需要的对象），使得当一个任务发现此字符串时，将字符串的位置立即通知给所有其他任务，以避免进一步的无用搜索。假定三个任务之一将发现SearchString。此外，所有的任务必须准备通过GetResult入口传回结果。

10.13 分析下面的Ada代码段:

```
Error_1, Error_2 : exception;
task Watch;
task Signaller;

protected Atc is
  entry Go;
  procedure Signal;
private
  Flag : Boolean := False;
end Atc;

protected body Atc is
  entry Go when Flag is
  begin
    raise Error_1;
  end Go;

  procedure Signal is
  begin
    Flag := True;
  end Signal;
end Atc;

task body Watch is
begin
  ...
```

```
    select
      Atc.Go;
    then abort
      -- 花 100微秒执行的代码
      raise Error_2;
    end select;
    ...
  exception
    when Error_1 =>
      Put_Line ("Error_1 Caught");
    when Error_2 =>
      Put_Line ("Error_2 Caught");
    when others =>
      Put_Line ("Other Errors Caught");
  end Watch;


  task body Signaller is
  begin
    ...
    Atc.Signal;
    ...
  end Signaller;
```

假设任务之间的上下文切换可在任何时候发生，仔细描述此程序段的可能执行情况。

10.14 一个特殊的基于POSIX的应用由几个周期性进程组成，并且有两种操作模式：MODE A和MODE B。应用有一个只运行于模式A的进程。简要描述此进程的设计，假设当系统想进行模式转换时，它给所有进程发送一个信号，以指出当前的操作模式。还假设存在一个称做WAITNEXTPERIOD的例程，它将挂起进程，直到它的下一个执行周期到来。注意，模式改变应该只在每个周期开始时影响进程。

10.15 解释Ada的OOP模型怎样用来生成可扩展的原子动作。

10.16 比较并对照Ada和Java的异步控制转移模型。

10.17 标准Java在哪些方面能被用来实现原子动作？

10.18 为什么Java例程resume ()、stop () 和suspend () 过时了？

377～378 10.19 用实时Java重做练习10.14。

10.20 说明怎样实现异常处理的Ada终止模型，以响应POSIX信号的回执（receipt）。

第11章 资源控制

第10章研究了实现可靠的进程合作的问题。在这一章也指出了如果进程要共享访问稀有资源，例如外部设备、文件、共享数据域、缓冲区和编码算法，那么也需要进程间的协调配合。这些进程被称作**竞争**（competing）进程。实时软件的许多逻辑（即非时态的）行为涉及到竞争进程之间的资源分配。虽然进程并不直接彼此通信以传递有关它们自身活动的信息，但是它们可以相互通信以协调对共享资源的访问。只有少数资源听任无限制地并发访问，然而，大多数资源的使用是受某些限制的。

正如在第7章指出的，资源实体的实现需要某种形式的控制代理。如果控制代理是被动的，那么这个资源是**保护式的**（protected）（或**同步式的**（synchronized））。否则，如果需要主动代理来安排正确的控制级别，那么这种资源控制器被称为**服务器**（server）。

本章将讨论可靠的资源控制的问题。也考虑竞争进程之间的一般的资源分配问题。虽然这些进程是彼此独立的，但是资源分配的行为和可靠性是有关系的。尤其是一个进程的失败可能导致分配给它的资源不能为其他进程所用。如果允许一些进程独占资源，其他进程可能由于得不到这些资源而饿死。更有甚者，当若干进程拥有其他进程需要的资源、同时又请求更多资源时，它们可能变成死锁的。 379

11.1 资源控制和原子动作

虽然进程需要通信和同步以实现资源分配，但是这不需要通过原子动作的形式来实现。这是因为需要交换的信息仅仅是达到和谐的资源共享所必需的信息，不可能交换任意的信息（Shrivastava and Banatre，1978）。因此，采用保护式资源或服务器形式的资源控制器能保证任何对局部数据的修改具有全局可接受性。如果不是这种情况，那么当一个进程（它已经被分配了资源）失败时，可能有必要将该进程的失败通知所有最近同资源控制器通信的进程。不过，一个特定的进程同控制器通信的代码应该是原子动作，因此当该进程正在分配或释放资源时，系统中的其他进程不能中断它。而且，资源管理器和客户进程可以使用向前和向后出错恢复来应付任何预期的或非预期的出错情况。

虽然在上一章指出过通常没有任何实时语言直接支持原子动作，但是可以通过小心使用可用的通信和同步原语来达到这种不可分的效果。

11.2 资源管理

模块性（特别是信息隐藏）要求资源必须被封装起来，且只能通过高级程序接口来访问，例如，在Ada中，只要在可能情况下，都应该使用下面的包：

```
package Resource_Control is

  type Resource is limited private;
  function Allocate return Resource;
  procedure Free (This_Resource : Resource);

private
  type Resource is ...

end Resource_Control;
```

如果资源管理器是服务器，那么Resource_Control的体部分应包含一个任务（或一个任
380 务类型的访问对象）。保护资源将在包体内使用保护对象。

在occam2中，资源管理器的惟一形式是服务器进程（也就是说所有资源控制器都必须编程为主动服务器）。这样的服务器应该通过带通道参数的过程（PROC）来实例化。

```
PROC resource.manager ([] CHAN OF Any request,
                       [] CHAN OF resource allocate,
                       [] CHAN OF resource free)
  ...
:
```

利用基于管程的同步，诸如具有条件变量和互斥锁的POSIX或Java的同步类，可以将保护资源自然地封装在一个管程内。例如，在Java中：

```
public class ResourceManager
{
  public synchronized Resource allocate ();
  public synchronized void free (Resource r);
}
```

其他的同步形式，如忙等待和信号量，由于没有给出适当级别的封装，因此在这一章不进一步研究它们。本章也不直接评价条件临界区（CCR），因为保护对象本质上就是CCR的一个现代形式。

下一节是关于各种资源管理方法的表达能力和易用性的讨论。在这个讨论之后，关于安全性的小节将考虑资源控制器如何能保护自己不被误用。

11.3 表达能力和易用性

Bloom（1979）提出了用于资源管理的同步原语的评价标准。对用于资源控制的同步原语的表达能力和易用性的分析形成本节的基础。要评价的原语包括管程/同步方法（利用条件同步）、服务器（带基于消息的接口）和保护式资源（作为保护对象实现）。后两种都为同步使用了守备，因此这个分析的一个方面是比较**条件同步**（condition synchronization）和**回避同步**（avoidance synchronization）。

Bloom使用术语“表达能力”来表示语言表达同步约束需求的能力。同步原语的易用性包括：

• 易于表达每一个同步约束，
• 易于将约束组合起来实现更复杂的同步方案。 381

在资源控制中，可以将表达这些约束所需的信息分类如下（按Bloom的分类）:

• 服务请求的类型;
• 请求到达的顺序;
• 服务器的状态和它管理的对象;
• 请求的参数。

Bloom的原始约束集合包括“对象的历史”（也就是所有以前服务请求的序列）。这里假设可以扩充对象的状态以包括所有需要的历史信息。对这个列表另外增加了一条，因为Bloom没有包括它:

• 客户的优先级。

第13章给出了进程优先级的完整讨论。对本章的目的而言，进程的优先级由进程的重要性来决定。

正如以上指出的，通常有两种语言上的方法来限制对服务的访问（Liskov等，1986）。第一种方法是**条件等待**（conditional wait）：系统接收所有的请求，但是如果不能马上满足该请求，就将发出请求的进程挂起到一个队列上。传统的管程代表了这种方法：系统将请求不能得到满足的进程在一个条件变量上排队，当能为该请求服务时恢复该进程。第二种方法是**回避**（avoidance）：系统不会接受请求除非能满足它们。能安全地接受请求的条件被表示成接受动作上的守备。

以下将分别研究评价同步方法的五个标准。

11.3.1 请求类型

可以使用操作请求的类型信息来决定一种请求优先于另外一种（例如：对实时数据库的读请求优先于写请求）。利用基于管程的同步和同步化方法，可以将读和写操作编写为不同的过程，但是根据管程的语义，对这些管程过程的未完成调用可以按任意的、优先级的或FIFO 的方式来处理。因此不可能优先处理读请求，也不可能知道有多少未完成的对管程过程的调用。

在Ada中，不同的请求类型可以很容易地用服务器任务或保护对象中的入口来代表。一个请求在得到实体访问权之前（为了在一个入口上排队），也没有其他途径去获得优先选择。但是将优先选择权给予排队的特定请求的一个自然的方法是通过守备，守备使用入口的“计数”（count）属性。以下代码显示`Updata`请求优先于`Modify`请求: 382

```
protected Resource_Manager is
  entry Update (...);
  entry Modify (...);
  procedure Lock;
  procedure Unlock;
private
  Manager_Locked : Boolean := False;
  ...
end Resource_Manager;

protected body Resource_Manager is

  entry Update (...) when not Manager_Locked is
```

```
    begin
      ...
    end Update;

    entry Modify (...) when not Manager_Locked and
                       Update' Count = 0 is
    begin
      ...
    end Modify;

    procedure Lock is
    begin
      Manager_Locked := True;
    end Lock;


    procedure Unlock is
    begin
      Manager_Locked := False;
    end Unlock;

  end Resource_Manager;
```

对于保护对象，只有入口可以有守备；一旦过程得到对象的访问权，它们将立即执行，因此不能使用过程给特定请求类型予优先。

在occam2语言中，每个请求类型和不同的通道组相联系。为了在两个备选动作之间做出选择，服务器进程必须使用选择性等待构造，幸运的是occam2提供了一种选择性等待的形式，该形式给每个备选一个不同的优先级。因而可以容易地按以下方式构造update-modify服务器：

383

```
WHILE TRUE
  PRI ALT
    ALT i=0 FOR max
      Update[i] ? object
        -- 更新资源
    ALT j=0 FOR max
      modify[j] ? object
        -- 修改资源
```

请记住如果有任何操作请求把信息传回给调用者，就需要双重的交互：

```
WHILE TRUE
  PRI ALT
    ALT i=0 FOR max
      update[i] ? object
        -- 更新资源
    ALT j=0 FOR max
      read[j] ? Any
        -- 从资源中抽取合适的成分
        output[j] ! object
```

调用者可发出以下调用：

```
SEQ
  read[MyChannelToServer] ! Any
  output[MyChannelToServer] ? Result
```

PRI ALT的使用给出了一个静态确定性选择。在Ada中，如果使用在实时附件中定义的一个编用（Queuing_Policy），可以得到同以上选择语句等价的形式。这个编用为入口和选择语句定义了排队方针。它允许编程进行确定性的选择（与此相对应的是使用文本顺序指出优先级），但是这需要所有调用任务有相同的优先级（如果不是这样，那么调用任务的优先级优先于选择项的静态顺序）。

11.3.2 请求顺序

某些同步约束可以表示为请求被接收的顺序（例如，为了保证公平或为了避免客户饿死）。就像已经观察到的，管程通常按FIFO顺序处理请求，因此它立即满足这种需求。在Ada中，如果选择适当的排队策略，同一类型的未完成请求（调用同一入口）也能用FIFO方式来服务。但是，用这种排队策略，不同类型的请求（例如：在一个选择语句内调用不同的入口）以任意顺序得到服务。这就不受程序员的控制。因此，没有办法按请求到达的顺序为不同类型的请求服务，除非使用FIFO策略并且所有客户首先调用公共的“注册”入口：

```
Server.Register;
Server.Action (...);
```

但是这个双重调用不是容易实现的（将在11.3.4节解释）。

利用occam2的一对一（one to one）指名结构，按照已发出请求到达的顺序处理对单个ALT构造的请求是不可能的。可以有许多服务通道供进程选择，但是进程不可能检测哪一个通道上的进程等待的时间最长。这里也应该注意，occam2服务器进程必须知道它拥有的可能的客户数量，必须给每一个客户分配一个独立的通道。Ada的模型更适合于客户－服务器模式，因为可以有任意多的客户调用一个入口，每个入口可以按FIFO顺序来处理。对于服务器任务和保护资源（对象）是这样的。 384

11.3.3 服务器状态

只有当服务器和它所管理的对象处于某种特定状态时才能允许某些操作。例如，只有当一个资源是空闲的才能分配它，只有缓冲区有空槽时才能放一个项目到里面。对于回避同步，基于状态的约束被表示为守备，对于服务器，约束是基于接收语句的位置（或消息接收操作符）。对于管程，约束是用条件变量实现的，因而管程用于描述状态同样是非常合适的。

11.3.4 请求参数

服务器操作的顺序可能受到包含在请求参数中的信息的约束。这些信息一般涉及到请求的身份或大小（在资源是可量化的情况下，如存储器）。对于通用的资源控制器，可以直接构造一个管程结构（在Java中）。一个对资源集合的请求包含一个参数，这个参数指出该请求集合的大小。如果没有足够的资源可用，则调用者被挂起；当有任何资源被释放时，系统唤醒所有挂起的客户，以查看现在是否能满足它们的请求。

```
public class ResourceManager
{
  private final int maxResources = ...;
  private int resourcesFree;

  public ResourceManager ()
  {
```

```
        resourcesFree = maxResources;
      }

      public synchronized void allocate (int size) throws
                          IntegerConstraintError, InterruptedException
        // 见 6.3.2节IntegerConstraintError的定义
      {
        if (size > maxResources) throw new
                          IntegerConstraintError (1, maxResources, size);
        while (size > resourcesFree) wait ();
        resourcesFree = resourcesFree - size;
      }

      public synchronized void free (int size)
      {
        resourcesFree = resourcesFree + size;
        notifyAll ();
      }
    }
```

385

利用简单的回避同步，守备仅仅访问服务器的局部变量（或保护对象），直到接收了调用才能访问该调用携带的数据。因此需要把请求构造为双重的交互。以下将讨论一个针对这个问题的资源分配器，主要讨论如何将这种资源分配器构造成Ada中的服务器。对于（Ada中的）保护对象的构造和occam2中的服务器的构造留给读者练习（见练习11.3和11.4)。这个Ada例子由Burns等（1987）给出的一个例子改编而来。

1. 资源分配和Ada——一个例子

在Ada中处理这个问题的办法是将每一个请求类型同一个入口族相联系。每个允许的参数值被映射到这个族的惟一索引上，这样带不同参数的请求指向不同的入口。很明显，只有参数是离散类型的，这才是适当的方法。

下面的包阐明了这种方法，它说明了由于表达能力的缺乏导致了一个复杂的程序结构(也就是易用性不好)。然后再次考虑一个资源分配的例子，用请求的大小作为请求的参数。就像上面指出的，标准的方法是将请求参数映射到入口族的序标上，这样不同大小的请求指向不同的入口。对于较小的范围，可以使用先前描述的“请求类型”技巧，用选择语句列举入口族的每个入口。但是，对于较大的范围就需要一个更复杂的解决方案。

```
package Resource_Manager is
  Max_Resources : constant Integer := ...;
  type Resource_Range is new Integer range 1..Max_Resources;
  subtype Instances_Of_Resource is Resource_Range range 1.. ...;
  procedure Allocate (Size : Instances_Of_Resource);
  procedure Free (Size : Instances_Of_Resource);
end Resource_Manager;

package body Resource_Manager is

  task Manager is
    entry Sign_In (Size : Instances_Of_Resource);
    entry Allocate (Instances_Of_Resource); -- 入口族
    entry Free (Size : Instances_Of_Resource);
  end Manager;
```

```
procedure Allocate (Size: Instances_Of_Resource) is
begin
  Manager.Sign_In (Size);  -- size 是一个参数
  Manager.Allocate (Size); -- size 是入口族的序标
end Allocate;

procedure Free (Size : Instances_Of_Resource) is
begin
  Manager.Free (Size);
end Free;

task body Manager is
  Pending : array (Instances_Of_Resource) of
    Natural := (others => 0);
  Resource_Free : Resource_Range := Resource_Range' Last;
  Allocated : Boolean;
begin
  loop
    select
      accept Sign_In (Size : Instances_Of_Resource) do
        Pending (Size) := Pending (Size) + 1;
      end Sign_In;
    or
      accept Free (Size : Instances_Of_Resource) do
        Resource_Free := Resource_Free + Size;
      end Free;
    end select;
    loop -- 主循环
      loop -- 接受任何悬挂的Sign-In 或 Free, 不等待
        select
          accept Sign_In (Size : Instances_Of_Resource) do
            Pending (Size) := Pending (Size) + 1;
          end Sign_In;
        or
          accept Free (Size : Instances_Of_Resource) do
            Resource_Free := Resource_Free + Size;
          end Free;
        else
          exit;
        end select;
      end loop;
      Allocated := False;

      for Request in reverse Instances_Of_Resource loop
        if Pending (Request) > 0 and Resource_Free >= Request then
          accept Allocate (Request);
          Pending (Request) := Pending (Request) - 1;
          Resource_Free := Resource_Free - Request;
          Allocated := True;
          exit; -- 循环, 接受新的 Sign_In
        end if;
      end loop;
```

```
          exit when not Allocated;
        end loop;
      end loop;
    end Manager;
  end Resource_Manager;
```

386~387 管理器给大请求以优先。为了获得资源，需要同管理器进行一个两阶段的交互：一个"签到"(sign-in)请求和一个"分配"(allocate)请求。通过在包中封装管理器和提供单个的过程(Allocate)去处理请求，从而对资源的用户隐藏了这个双重交互。

当没有未完成调用时，管理器等待签到请求或释放请求（释放资源）。当一个签到请求到达时，它的大小被记录在悬挂请求的数组中。然后进入一个循环去记录所有未完成的签到请求和释放请求。一旦不再有请求这个循环就终止。

然后使用一个for循环去扫描悬挂数组，系统接受那个它所能接纳的最大请求。在有一个更大的请求试图签到时，就重复执行主循环。如果没有可以服务的请求，就退出主循环，管理器等待新的请求。

由于需要双重会合事务，这个解决方案是复杂的。它也是昂贵的，因为每个可能的请求大小都需要一个入口。

在SR语言（Andrews and Olsson，1993）中可以发现一个简单得多的系统。该系统允许守备访问（也就是引用）参数。因此，仅当服务器或保护资源知道自身处于可以容纳请求的状态时，它才会接收一个资源请求。不需要双重调用。例如（对于保护资源使用类似Ada的代码，但不是合法Ada代码）：

```
protected Resource_Control is  -- 不合法的Ada代码
  entry Allocate (Size : Instances_Of_Resource);
  procedure Free (Size : Instances_Of_Resource);
private
  Resource_Free : Resource_Range := Max_Resources;
end Resource_Control;

protected body Resource_Control is

  entry Allocate (Size : Instances_Of_Resource)
      when Resources_Free >= Size is     -- 不合法的Ada代码
  begin
    Resource_Free := Resource_Free - Size;
  end Allocate;

  procedure Free (Size : Instances_Of_Re source) is
  begin
    Resource_Free := Resource_Free + Size;
  end Free;

end Resource Control;
```

虽然语法上很简单，但这种解决方案也有不足，它不清楚在什么情况下需要对守备或屏障碍进行再求值。这可能导致低效的实现。一个替代方法是保持守备的简单性，但是通过增加一个**重排队**设施来增强通信机制的表达能力。在11.4节将详细解释这一点，然而，本质上说，这种方法允许一个已接受的调用（也就是通过了守备求值的调用）在一个新入口（或完全一样的入口）重排队，在该入口它必须再次通过守备。下面说的东西也激发了对重排队设

施的需求。 388

2. 双重交互和原子动作

在上面的讨论中，给出的Ada和occam2的两个例子都需要客户进程对服务器发出双重调用。必须要进行双重交互的主要因素之一是简单的回避同步缺乏表达能力。因此，为了编写可靠的资源控制过程，这种结构必须用原子动作来实现。在occam2中，双重调用

```
SEQ
  read[MyChannelToServer] ! Object
  output[MyChannelToServer] ? Result
```

形成了一个原子动作，因为可以保证客户会读取已经发送了请求的对象（与此类似，保证服务器送出了数据）。令人遗憾的是Ada不能做出这种保证（Wellings等，1984）。在两次调用之间，也就是"签到"之后"分配"之前，从"原子动作"的外面可以看到客户的中间状态:

```
begin
  Manager.Sign_In (Size);
  Manager.Allocate (Size);
end;
```

另一个任务可能在两次调用之间中止该客户，在这个意义上说，这个中间状态是可观察的，它给服务器留下了一些困难:

- 如果服务器假设客户将发出第二个调用，那么这个中止将使服务器变为等待状态，它等待客户的下一次调用（因而发生死锁）。
- 如果服务器保护自己不受客户的中止的影响（通过不等待不确定的第二次调用），那么当客户仅仅是稍慢发出调用时，服务器可以假定客户已经中止了，因此客户被错误地阻塞。

在实时软件的上下文中，已经提出了三种方法来处理这种中止问题:

- 定义中止原语以应用到原子动作上，而不是进程上，当同服务器通信时可以使用向前或向后的出错恢复。 389
- 假定中止只用于原子动作的分解并不重要的极端情形下。
- 试图保护服务器免受客户中止的影响。

在Ada中，第三种方法涉及到通过重排队第一次调用而消除对双重调用的需要（而不是让客户发出第二次调用）。就像上面指出的，这一点将在讨论过最后一个的评价标准后在11.4节再详细解释。

11.3.5 请求者优先级

评价资源管理同步原语的最后一个标准涉及客户优先级的使用。如果进程集合是可运行的，那么分派器可以根据优先级安排进程的执行。但是分派器不能控制已挂起的等待资源的进程。因此有必要通过客户进程的相对优先级来约束资源管理器的操作顺序。

在Ada、实时Java和POSIX中可以定义一个按优先级顺序的排队策略，但是通常在并发程序设计语言中，进程通过信号量或条件变量这样的原语用任意的或FIFO的方式来启动；管程在入口上通常使用FIFO方式；选择性等待通常使用任意的或静态的文本优先级顺序。在后一种情形中，可以安排客户通过不同的接口（如入口或通道）来访问资源。对于小的优先级范围，它等价于*请求类型*约束。对于大的优先级范围，它等价于*请求参数*，因而能使用前面已介绍的方法。

虽然通常说管程有一个FIFO的排队原则，但这并不是一个根本特性。很明显，管程可能按优先级排序。其实，在管程的POSIX和实时Java实现中，不仅允许优先级队列，(从概念上讲）也允许将外部队列（等待进入管程的进程）和内部队列（通过条件变量发信号启动的进程）合并，以给出一个单一的优先级排序队列。因此一个等待访问管程的高优先级进程将优先于内部启动的进程。

11.3.6 小结

在以上的讨论中，使用了五个要求去判断当前语言结构处理一般的资源控制是否合适。管程利用它的条件同步很好地处理了请求参数；基于回避的原语在处理参数方面需要加强，但它在处理请求类型方面有优势。

但是应该指出的是这些要求并不是彼此相容的。在客户的优先级和请求到达的顺序之间，或在被请求的操作和请求者的优先级之间也许存在冲突。例如，在occam2中（利用它的确定性选择等待），通过以下程序可以给update请求分配一个超过modify请求的优先级：

390

```
WHILE TRUE
  PRI ALT
    ALT i=0 FOR max
      update[i] ? object
        -- 更新资源
    ALT j=0 FOR max
      modify[j] ? object
        -- 修改资源
```

但是，如果modify的调用者的优先级高于update的调用者的优先级，那么modify操作将首先执行。这些目标不能都得到满足，因此需要完善同步原语，使它允许编程者用一种结构化的方法来处理这种冲突。

我们已经说明了必须将客户同资源管理器的交互构造成原子动作。没有直接的语言原语支持原子动作，中止和异步控制转移（ATC）使这个问题更加困难，进程可能被异步地从管程中清除，或者在对服务器进程的两次相关调用之间被终止。中止被用于消除出错的或冗余的进程。如果正确地构造了一个管程，那么可以假定一个无赖进程在管程中不会造成伤害(假定只是在这个进程进来之后才知道它是个无赖)。同样如果一个无赖客户正等待第二次同步调用，它也不会对管程造成伤害。

所以，有一些进程不应被中止（更准确地说是中止仍将发生但被推迟）的很明确的情形。于是产生了**延期中止区**（abort-deferred regions）的概念。

Java中有延期ATC区的概念（ATC-deferred region)。但是Java的destroy方法引起了资源控制器的问题，因此应该不使用它。同样的理由，Java的Thread类中过时的方法也不应使用。

Ada将保护对象（以及会合）的执行定义为延期中止区。可是，同资源控制器的交互通常可能处于挂起状态，并且不能从保护对象内部进行这样的调用。例如，不能将对一个服务器任务的双重调用封装在保护对象里面。然而，正如前面指出的，重排队方法解决了许多这样的资源控制问题，这将在下一节描述。

11.4 重排队设施

通过对Bloom标准的讨论可以看出，回避同步是编写资源管理器的更结构化的方法，但是

同低层条件同步相比它缺乏表达能力。增强回避同步可用性的一个方法是增加重排队设施。Ada语言有这样的设施，因此下面的讨论将集中在这种语言模型上。

重排队中的关键概念是将一个任务（已经通过了一个守备或屏障的任务）移开使之远离另一个守备。作为模拟，考虑一个人（任务）等待进入一个有一个或多个门（守备入口）的房间（保护对象）。一旦位于房间里面，这个人可以被从这个房间逐出（重排队），并马上将它放入另一个（可能是关闭的）门的后面。

Ada允许在任务入口和保护对象入口之间重排队。重排队可以是在同一入口上、同一个单元的另一个入口上、或者完全是在另一单元上。也允许从任务入口到保护对象入口的重排队（反之亦然）。但是，重排队的主要用途是发送调用任务到同一单元（该单元是重排队执行的地方）的不同入口上。 391

在10.8.2节给出了实现向前出错恢复和利用Controller实现双重交互的代码（也就是，调用cleanup，然后接着调用Wait_Cleanup）。通过在保护入口里面的在Wait_Cleanup上的重排队，就可以将双重调用改为对Cleanup的单个调用（Wait_Cleanup变成私有入口）:

```
entry Cleanup (Vote:Vote_T;Result : out Vote_T) when True is
begin
  if Vote = Aborted then
    Final_Result := Aborted;
  end if;
  requeue Wait_Cleanup with abort;
end Cleanup;
```

with abort设施将在以后描述。该调用的结果是将调用任务移到Wait_Cleanup入口（就像从外部发出调用）。重排队语句的执行终止了原来的被调用入口的执行。

资源控制问题提供了另一个重排队应用的例子。在以下算法中，将一个不成功的请求重排队到保护对象的私有入口（称为Assign）。现在这个保护对象的调用者仅对Request发出了单个调用。无论何时释放了资源，都有记录记载Assign入口上有多少任务。这些任务可以重试，要么获得分配，要么在同一个Assign入口的最后重排队。最后重试的任务打开屏障:

```
type Request_Range is range 1..Max;

protected Resource_Controller is
  entry Request (R : out Resource; Amount : Request_Range);
  procedure Free (R : Resource; Amount : Request_Range);
private
  entry Assign (R : out Resource; Amount : Request_Range);
  Freed : Request_Range := Request_Range' Last;
  New_Resources_Released : Boolean := False;
  To_Try : Natural := 0;
  ...
end Resource_Controller;


protected body Resource_Controller is
  entry Request (R : out Resource; Amount : Request_Range)
        when Freed > 0 is
  begin
    if Amount <= Freed then
```

392

```
      Freed : = Freed - Amount;
      -- 分配
    else
      requeue Assign;
    end if;
  end Request;


  entry Assign (R : out Resource; Amount : Request_Range)
    when New_Resources_Released is
  begin
    To_Try : = To_Try - 1;
    if To_Try = 0 then
      New_Resources_Released := False;
    end if;
    if Amount <= Freed then
      Freed := Freed - Amount;
      -- 分配
    else
      requeue Assign;
    end if;
  end Assign;


  procedure Free (R : Resource; Amount : Request_Range) is
  begin
    Freed := Freed + Amount;
    -- 释放资源
    if Assign' Count > 0 then
      To_Try := Assign' Count;
      New_Resources_Released := True;
    end if;
  end Free;
end Resource_Controller;
```

要注意的是只有Assign入口的排队原则是FIFO的，这个算法才能工作。基于优先级的算法留给读者作练习（练习11.5）。

应当看到如果保护对象记录了最小的未完成请求，那么能够得到一个更高效的算法。如果Freed>=Smallest，那么应该只在Free中打开屏障（或在Assign中仍然打开）。

然而，即使使用这样的解决方案，给某些请求分配一个不同于FIFO或任务优先级排序的优先级也是困难的。就像早先指出的，编写这个级别的控制程序需要一个入口族。但是利用重排队可以给出一个改进的结构，而不是用以下方法：

```
procedure Allocate (Size:Instances_Of_Resource) is
begin
  Manager.Sign_In (Size);  -- size 是一个参数
  Manager.Allocate (Size);  -- size 是一个族序标
end Allocate;
```

393 和一个包含

```
accept Sign_In (Size:Instances_Of_Resource) do
  Pending (Size) := Pending (Size) + 1;
end Sign_In;
```

的服务器任务，通过以下方法可以使双重调用成为原子动作：

```
procedure Allocate (Size : Instances_Of_Resource) is
begin
  Manager.Sign_In (Size); -- size 是一个参数
end Allocate;
```

和

```
accept Sign_In (Size : Instances_Of_Resource) do
  Pending (Size) := Pending (Size) + 1;
  requeue Allocate (Size);
end Sign_In;
```

通过将Allocate入口变为私有的可以使这个服务器任务更安全：

```
task Manager is
  entry Sign_In (Size : Instances_Of_Resource);
  entry Free (Size : Instances_Of_Resource);
private
  entry Allocate (Instances_Of_Resource)
                 (Size : Instances_Of_Resource);
end Manager;
```

这个算法不仅比先前给出的算法更直接，而且也有一个优点，那就是任务在将自身从一个入口队列上移除（在计数属性已经承认了它的存在后）这一点上，该算法是有弹性的。一旦一个任务已经被重排队，它不可能被中止或受制于在入口调用上的超时——见以下的讨论。

11.4.1 重排队的语义

重要的是要意识到重排队不是一个简单的调用。如果过程P调用过程Q，那么在Q结束后，控制被传给P。但是如果入口X在入口Y上重排队，控制就不传回给X。在Y结束后，控制传回到调用X的对象。因此，当入口或接受体执行了重排队后，接受体就结束了。

这种情况的一个后果是，当重排队是从一个保护对象到另一个保护对象时，一旦任务进行了重排队就放弃了原来对象上的互斥。等待进入第一个对象的其他任务将能够做到这一点。但是，重排队到同一个保护对象将保留互斥锁（如果目标入口是打开的）。

重排队语句中命名的入口（叫做**目标**（target）入口）要么没有参数，若有参数，参数剖面要同发出重排队指令的入口（或接受）语句中的参数等价（即类型相容）。例如，在资源控制程序中，Assign的参数要同Allocate的参数一样。因为这个规则，调用时不必给出实际参数，其实是禁止调用时给出实际参数（以防程序员试图改变它们）。因此，如果目标入口没有参数，就没有传递信息；如果它有参数，那么把执行重排队的实体中的相应参数映射到目标入口。因为这个规则，先前给出的从多个不同入口得到FIFO排序的算法——也就是使用单个注册入口的算法： 394

```
Server.Register;
Server.Action (...);
```

——不能使用重排队方法编写为单个的调用（因为不同动作的参数可能是不同的）。

在重排队语句中可以使用with abort子句选项，通常当一个任务在入口队列中时，它将留在那儿直到得到服务，除非它发出一个限时入口调用（参见12.4.2节）或由于使用异步控制转移或中止移除它，一旦任务被接受了（或开始执行一个保护对象的入口），就取消任务的超时时间，推迟任何中止尝试的影响，直到任务从入口中出来。然而，有一个问题是关于重排队的后果是什么。研究中止问题，很明显有两个观点：

- 由于第一个调用已经被接受，中止应当仍然推迟，以保证任务或保护对象进行第二个调用。
- 如果重排队将调用任务放回入口队列，那么中止就应是再次可能的。

对于超时也有一个类似的论证。重排队语句允许用这两种观点编程，默认情况是不允许进一步的超时或中止，加上with abort子句使得能够将任务从第二个入口清除。例如，在本节开始给出的向前出错恢复算法就需要这些语义。

（决定是否使用with abort的）真正问题是：在守备或屏障打开时，把客户任务重排队的保护对象（或服务器任务）是否预计客户任务就在那里，如果该对象的正确行为需要任务的出现，那么不应该使用with abort。

11.4.2 重排队到其他入口

虽然重排队到同一实体代表了重排队的一般的用法，但是也有使用这个语言特性的最为一般性的情形。

考虑资源被一个对象的层次体系控制的情况。例如，一个网络路由器可以选择三个通信
395 线路传送消息：`Line_A`是首选路径，但是如果它超载了，使用`Line_B`，如果它也超载了，使用`Line_C`。每条线路由一个服务器任务控制，它是一个活动实体，因为它要执行整理（housekeeping）操作，一个保护单元作为路由器的接口：它决定使用三个通道中的哪一个，然后使用重排队传递请求到合适的服务器，这个解决方案的结构见图11-1，程序如下：

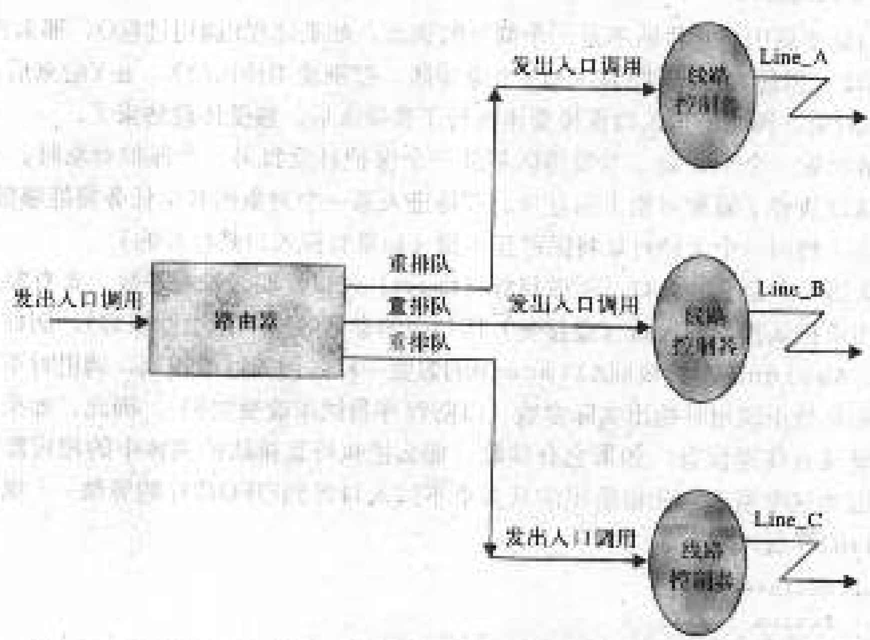


图11-1 网络路由器

```
type Line_Id is (Line_A, Line_B, Line_C);
type Line_Status is array (Line_Id) of Boolean;

task type Line_Controller (Id : Line_Id) is
  entry Request (...);
end Line_Controller;

protected Router is
  entry Send (...);  --  与Request相同的参数剖面
  procedure Overloaded (Line : Line_Id);
  procedure Clear (Line : Line_Id);
private
  Ok : Line_Status := (others => True);
end Router;
La : Line_Controller (Line_A);
Lb : Line_Controller (Line_B);
Lc : Line_Controller (Line_C);


task body Line_Controller is
 ...
begin
  loop
    select
      accept Request (...) do
        -- 服务请求
      end Request;
    or
      terminate;
    end select;
    -- 整理操作, 可能包括
    Router.Overloaded (Id);
    -- 或者
    Router.Clear (Id);
  end loop;
end Line_Controller;


protected body Router is

  entry Send (...) when Ok (Line_A) or Ok (Line_B) or Ok (Line_C) is
  begin
    if Ok (Line_A) then
      requeue La.Request with abort;
    elsif Ok (Line_B) then
      requeue Lb.Request with abort;
    else
      requeue Lc.Request with abort;
    end if;
  end Send;

  procedure Overloaded (Line : Line_Id) is
  begin
```

```
  Ok (Line) := False;
  end Overloaded;

  procedure Clear (Line : Line_Id) is
  begin
    Ok (Line) := True;
  end Clear;

end Router;
```

396~397

11.5 不对称指名和安全性

在具有直接对称指名的语言中，服务器进程（或保护资源）总是知道它处理的客户进程的身份。对于基于一对一中介（如occam2中的通道）的间接指名方案也是这样。但是，不对称指名导致服务器不知道客户的身份。前面曾指出，它的优点是能编写通用服务器，但导致较差的资源使用的安全性。特别是服务器可能希望知道客户的身份以便:

- 出于预防死锁（见11.7节）的考虑或公平考虑（若满足该请求对其他客户就不公平）拒绝请求
- 保证资源只能由占用它的进程释放。

在CHILL中，进程与一个**实例**或名字相关联（见9.6节），这使资源控制器（在CHILL中称为**区**）知道调用进程的身份。在Ada中用任务标识（见7.3.6节）或在Java中用`Thread.currentThread`可以编写一个相似的设施。考虑一个简单的Ada资源控制器，它只管理一个资源。

```
protected Controller is
  entry Allocate;
  procedure Free;
private
  Allocated : Boolean := False;
  Current_Owner : Task_Id := Null_Task_Id;
end Controller;


protected body Controller is

  entry Allocate when not Allocated is
  begin
    Allocated := True;
    Current_Owner := Allocated' Caller;
  end Allocate;

  procedure Free is
  begin
    if Current_Task /= Current_Owner then
      raise Invalid_Caller; -- 一个合适的异常
    end if;
    Allocated := False;
    Current_Owner := Null_Task_Id;
  end Free;

end Controller;
```

398

注意，用这个Ada设施，入口的调用者是通过`Caller`属性标识的，而通过函数`Current_Task`获得过程的调用者。这个不同的原因在这里并不重要（其理由见文献Burns and Wellings (1998)）。

11.6 资源的使用

当竞争或合作进程需要资源时，正常的操作方式是：*请求*（`request`）资源（如果需要就等待）、*使用*（`use`）资源、然后*释放*（`release`）资源。通常用两种访问模式请求一个资源。这两种模式是共享访问或互斥访问。共享访问允许资源被多个进程并发使用：例如一个只读文件。互斥访问要求在同一时间只允许一个进程访问资源：例如一个物理资源，如行式打印机。某些资源能按任一种模式使用。在这种情况下，如果一个进程请求按共享方式访问资源，但该资源正被互斥访问时，那么该进程必须等待。如果资源已经处于共享访问模式，那么该进程就能访问该资源。类似地，如果请求互斥地访问一个资源，那么发出请求的进程必须等待正共享访问资源的进程结束访问。

因为当进程请求资源时可能被阻塞，所以进程只有在它需要资源时才请求资源。而且，一旦给进程分配了资源，进程应尽可能快地释放资源。如果进程没有这么做，系统的性能就可能显著降低，因为系统中的进程将持续等待稀有资源。但是，如果进程释放资源过早并导致失败，那么它们可能通过资源传递错误信息。因为这个原因，必须修改10.1.1节介绍的*两阶段*资源用法，以便直到动作结束才释放资源。如果成功执行了恢复，那么动作里面的任何恢复过程必须要么释放资源，要么撤销对资源的任何影响。如果没有完成恢复，则需要执行撤销，并且系统必须恢复到一个安全状态。对于向前出错恢复，可以通过异常处理程序执行这些操作。对于向后出错恢复，如果没有一些额外的语言支持，就不可能执行这些操作（Shrivastava，1979）。

11.7 死锁

因为许多进程竞争有限的资源，那么可能发生以下情况：进程P_1独占地访问资源R_1，同时等待访问资源R_2。如果进程P_2已经独占对资源R_2的访问并同时等待访问R_1，那么**死锁**（deadlock）就发生了，因为两个进程都在等待对方释放资源。由于死锁，所有受影响的进程被无限期挂起。一个类似的敏感情形是一个进程集合的推进被抑制但仍在执行。这种情形被称为**活锁**（livelock）。一个典型的例子是相互交互的进程集合陷入了无限循环中，它们不能 399
向前推进，但是它们正做着无用的工作。

另一个可能的问题发生在当几个进程连续不断地试图独占地访问相同资源的时候。如果资源分配方案不公平，那么一个进程可能从来没有机会访问资源。这种情况叫做**无限延期**（indefinite postponement）或**饿死**（starvation）或**停工**（lockout）（见第8章）。在并发系统中，**活性**（liveness）意味着如果假设某事件会发生，那么它最终将发生。由于死锁、活锁或饿死导致了活性的破坏。注意活性不是像公平性那样强的条件。但是给公平性一个单一的精确定义是困难的。

本节剩下的部分将关注死锁和如何预防死锁、避免死锁和死锁的检测与恢复。

11.7.1 死锁发生的必要条件

必须同时具备以下四个必要条件才会发生死锁：

- **互斥**——只有一个进程能够立即使用资源（即资源是非共享的或至少限制并发访问）；
- **拥有并等待**——必须存在这样的进程，它拥有资源并等待其他资源；
- **非抢占**——资源只能由进程自愿释放；
- **循环等待**——必须存在一个进程循环链，每个进程都拥有资源，而链中的下一个进程正请求这些资源。

11.7.2 处理死锁的方法

如果想要一个实时系统是可靠的，它就必须解决死锁问题。有三个可能的方法：

- 死锁预防
- 死锁避免
- 死锁检测与恢复

下面做简要的讨论。读者可参考任何操作系统书籍查看更全面的讨论。

1. *死锁预防*

400 可以通过保证死锁四个必要条件中的至少一个条件决不发生来预防死锁。

互斥：如果资源是共享的，那么它们不可能卷入死锁。遗憾的是，虽然少数资源允许并发访问，但是大多数资源的使用是受限制的。

拥有并等待：避免死锁的一个非常简单的办法是要求进程要么在运行前请求资源，要么在运行中没有资源分配时请求资源。但是这可能导致资源的低效使用和饿死。

非抢占：如果放宽从进程中抢占资源的限制，就能预防死锁。有几种方法，包括：如果一个进程试图分配资源但是失败了，就释放它已拥有的所有资源；如果一个进程请求一个资源，但该资源已分配给了一个被阻塞的进程（等待另一资源），就将该资源窃取过来。被阻塞进程现正在等待一个额外的资源。这种方法的缺点是它需要保存和恢复资源的状态，在许多情况下这也许是不可能的。

循环等待：为了避免循环等待，给所有资源类型进行线性的排序。根据这个排序，给每一个资源类型分配一个编号。假设R是一个资源类型集合

$$R = <R_1, R_2, \cdots R_n> \text{所有资源的集合}$$

函数F的参数为资源类型，返回值为资源在线性排序中的位置。如果进程拥有资源R_J，而且它请求资源R_K，只在以下条件成立时考虑该请求：

$$F(R_J) < F(R_K)$$

或者一个进程请求资源R_J前必须首先释放所有资源R_I，R_I应满足下式

$$F(R_I) < F(R_J)$$

注意函数F应根据资源的用法推导得出。

另一个预防死锁的方法是形式化地分析程序，因此验证不可能存在循环等待。这样的分析能否易于进行密切依赖于所使用的并发模型。低级同步原语极难用这种方法来检验；与此相比，基于消息的结构更符合形式化证明规则的应用。occam2就是特别为这个目的而设计的。它的语义已形式化地进行了详细说明，而且为CSP开发的无死锁编程的许多理论都适用于occam2（Hoare，1985）。

2. *死锁避免*

如果知道更多关于资源用法模式的信息，那么就可能构造一个算法，该算法允许死锁的

所有四个必要条件都发生，但是它保证系统不会进入死锁状态。死锁避免算法动态检查资源分配的状态，然后采取动作保证系统不会进入死锁。但是，仅仅询问下一个新状态是否是死锁是不够的，因为如果下一个状态是死锁，也许不可能采取替代动作避免这个情况。相反的做法是必须询问系统是否仍处于安全状态。资源分配状态由下式给出： 401

状态＝可用资源数、已分配资源数以及每个进程最大资源需求

如果系统能按某种顺序为每个进程分配资源（直到它们的最大需求），而且仍然避免了死锁，那么系统是**安全的**（safe）。例如，考虑一个需要访问大辅助存储设备的数据获取系统。它有12个磁带机和三个进程：P_0最多需要10台磁带机、P_1最多需要4台磁带机、P_2最多需要9台磁带机。假设在时间T_i时，P_0已拥有5台、P_1已拥有2台、P_2已拥有2台（还剩3台），见表11-1。

表11-1 安全状态

进程	已分配	需要
P_0	5	10
P_1	2	4
P_2	2	9
	共分配＝9	
	还剩下＝3	

这是一个安全状态，因为存在一个序列<P_1、P_0、P_2>允许所有进程结束。如果在时间T_i+1时，P_2请求1台磁带机且系统将磁带机分配给它，这时系统状态变为：P_0有5台、P_1有2台、P_2有3台并还剩2台，见表11-2。

表11-2 不安全状态

进程	已分配	需要
P_0	5	10
P_1	2	4
P_2	3	9
	共分配＝10	
	还剩下＝2	

402

这时系统不再是安全状态，利用剩下的磁带机，只有P_1能够结束，P_1结束后：P_0有5台、P_2有3台但只剩4台。此时P_0或P_2都不能得到满足，因此系统可能进入死锁状态。所以系统应该阻塞进程P_2在时间T_i＋1时的请求。

很明显，死锁是一个不安全状态。系统处于不安全状态可能导致死锁，但是也可能不导致死锁。但是如果系统处于安全状态，它将不会死锁。当资源类型多于一个时，死锁避免算法变得更复杂。通常使用的是**银行家**（Banker's）算法。这种算法的讨论可在大多数操作系统书中找到。

3. 死锁检测和恢复

在许多通用的并发系统中，资源的分配用法事先并不知道。即使事先知道，死锁避免的代价常常是昂贵的。因此许多系统忽略死锁问题直到系统进入死锁状态。然后它们采取一些纠正动作。遗憾的是如果系统中有许多进程，也许很难检测死锁。一种方法是使用**资源分配图**（resource allocation graph）（有时称为**资源依赖图**（resource dependency graph））。

为了检测一组进程是否发生了死锁，需要资源分配系统知道已经给哪些进程分配哪些资源，哪些进程被阻塞等待已分配的资源。如果系统有这些信息，它就可以构造一个资源分配图。

使用资源分配图，系统能检测是否存在死锁。系统首先检测没有被阻塞的进程。然后通过移走这些进程将图简化，因为假设如果这些进程继续运行，它们最终将终止。

通常，如果在资源分配图中没有循环，就没有发生死锁。但是如果有一个循环，系统可能发生死锁，也可能不发生，这取决于图中的其他进程。因此资源分配图能用于预防死锁。每当有资源请求时，就修改这个图。如果存在一个循环，那么潜在的死锁就可能发生，因此必须拒绝资源请求。

如果能检测出死锁就能通过以下方法恢复：打破一个或多个资源的互斥；中止一个或多个进程；或者从一个或多个死锁进程中抢占一些资源。打破互斥虽然容易实现，但是可能使系统进入不一致的状态，如果一个资源是可共享的，它应该被保证是可共享的。中止一个或多个进程是一个非常严厉的措施，也可能使系统进入不一致状态。如果采用这些方法中的任何一个，必须计算这种动作的代价，包括考虑进程优先级、进程已经执行了多长时间、它还需要多少资源等等。

抢占需要选择一个或多个牺牲者。选择时要考虑进程优先级，是否简单地抢占资源，进
403 程离完成还有多远。一旦选定了，抢占的过程将依赖于使用何种形式的出错恢复。对于向后出错恢复，进程从该资源分配前的一个恢复点重新开始执行。对于向前出错恢复，在牺牲进程内引发一个适当的异常，该进程必须采取某种纠正动作。

显然必须小心谨慎，以保证同样的进程不会因不断回滚而造成资源饥饿。这种问题的最常见解决办法应包括一个进程被抢占的次数作为代价因素的一部分。

本节简要概述了系统可用于避免或应付死锁的可能方法。这些方法的大多数实现起来都是非常昂贵的，在实时系统中最好禁止使用。特别是在分布式系统中更是这样。因此，一些系统在资源分配请求上使用简单的超时。如果超时到期，进程可以假定有潜在的死锁，然后采取一些替代的动作过程。在第13章将介绍一个基于优先级调度的资源共享方法。这个方法有一个明显有用的特性，它是无死锁的——至少在单处理器系统上是这样。

小结

在每一个计算机系统中，有许多进程竞争有限的资源集合。需要算法来管理资源的分配和回收过程（资源分配的机制），并保证按照预定义的行为给进程分配资源（资源分配的策略）。这些算法也负责确保进程在等待资源分配请求完成时不死锁。

在进程间共享资源需要这些进程互相通信和同步。因此，最根本的是：实时语言提供的同步设施应有足够的表达能力，使得能够规定一个广泛范围的同步约束。可以把这些约束分类如下：资源调度必须考虑到

- 服务请求的类型；
- 请求到达的顺序；
- 服务器的状态和它管理的对象；
- 请求的参数；
- 客户的优先级。

管程（带有条件同步）能很好地处理请求参数，在基于消息的服务器或保护对象中的回

避同步足以应付请求类型。

如果同步设施的表达能力不够，进程常常被迫同资源管理器进行双重交互。双重交互必须用原子动作实现，否则客户进程有可能在第一个交互后、第二个交互前被中止。如果这个可能性确实存在，那么编写可靠的资源管理器就是非常困难的。增强守备的表达能力的一个方法是允许重排队。这个设施使回避同步与条件同步一样有效。 404

资源管理的一个关键需求是提供无死锁的用法。必须具备以下四个必要条件才会发生死锁：

- 互斥
- 拥有并等待
- 非抢占
- 循环等待

如果一个实时系统是可靠的，它必须解决死锁问题。有三种可能的方法：

- 死锁预防——通过保证以上四个条件中至少有一个不会发生来预防死锁；
- 死锁避免——使用资源用法模式的信息来构造算法，该算法允许所有四个条件发生，但是它也保证系统决不会进入死锁状态；
- 死锁检测与恢复——允许死锁发生（通过检测发现），然后通过以下方法恢复：破坏一个或多个资源的互斥；中止一个或多个进程；从一个或多个已死锁进程中抢占一些资源。

相关阅读材料

Bloom, T. (1979) Evaluating Synchronization Mechanisms. *Proceedings of the Seventh Symposium on Operating Systems Principles*, ACM, pp. 24–32.

Silberschatz, A. and Galvin, P. A. (1998) *Operating System Concepts*. New York: John Wiley & Sons.

练习

11.1 下面的资源控制器试图将请求同优先级联系起来。

```
type Level is (Urgent, Medium, Low);

task Controller is
    entry Request (Level)  (D:Data);
end Controller;

task body Controller is
  ...
begin
  loop
    ...
    select
      accept Request (Urgent)  (D:Data) do
        ...
      end;
    or
      when Request (Urgent) ' Count = 0 =>
        accept Request (Medium)  (D:Data) do
```

405

```
          ...
        end;
    or
      when  Request (Urgent) ' Count = 0  and
            Request (Medium) ' Count = 0 =>
        accept Request (Low)  (D:Data) do
          ...
        end;
    end select
    ...
  end loop;
end Controller;
```

解释这个解决方案，指出它将在什么条件下失败。为什么扩充上面的解决方案以应付0～1 000范围内的数值优先级是不明智的？拟定一个替代的解决方案以应付较大的优先级范围。你可以假设调用任务只发出简单入口调用并且不会被中止。

11.2 说明如何用Java实现11.3.1节给出的资源管理器（它没有重排队设施）。

11.3 说明如何用保护对象实现11.3.4节给出的Ada资源管理器。

11.4 说明如何用occam2实现11.3.4节给出的Ada资源管理器。

11.5 说明如何改写11.4节给出的Ada资源管理器，使它允许高优先级客户优先于低优先级客户。

11.6 说明如何使用POSIX的互斥锁和条件变量实现一个资源控制器，该资源控制器用于几个相同资源的分配和释放。客户可以使用以下接口请求和释放一个或多个资源。

```
typedef struct{
    ... /*填入 */
  } resource;

void allocate (int size, resource *R);
void deallocate (int size, resource *R);

void initialize (resource *R);
```

11.7 考虑一个有五个进程（P_1，P_2，$\cdots P_5$）和7类资源（R_1，R_2，$\cdots R_7$）的系统。资源2、5、7各有1个，资源1、3、4、6各有2个。进程1已经得到了1个资源R_1，还需要1个资源R_7。进程2已经得到了资源R_1、R_2、R_3各1个，还需要1个资源R_5。进程3已经得到了资源R_3、R_4各1个，还需要1个资源R_1。进程4已经得到了资源R_4、R_5各1个，还需要1个资源R_2。进程5已经得到了1个资源R_7。

该系统处于死锁中吗？给出你的理由。

11.8 一个系统处于表11-3描述的状态，这个系统是处于安全还是不安全状态？给出你的理由。

表11-3 练习11.8中的系统的状态

进　程	已 分 配	最 多 需 要
进程1	2	12
进程2	4	10
进程3	2	5
进程4	0	5

（续）

进　程	已分配	最多需要
进程5	2	4
进程6	1	2
进程7	5	13
	还剩下=1	

11.9 一个处于不安全状态的系统并不一定会死锁。解释为什么这是对的。给出一个不安全状态的例子，说明进程如何能在没有死锁发生的情况下执行完毕。 406~407

11.10 说明如何用Java实现11.4.2节给出的网络路由器的例子。

第12章 实时设施

第1章已经提到用于嵌入式系统编程的语言需要一些用于实时控制的设施。事实上“实时”已经被当作这类系统的同义词。既然实时性对于很多嵌入式系统是如此重要，那么直到第12章才来讨论这个主题似乎很奇怪。但是实时控制的设施通常是建立在语言的并发模型之上的，所以必须先讨论并发模型。

将时间的概念引入到编程语言中可以大致分为三个独立的主题：

1）同“时间”的对接。例如：时钟访问可以计算出一段时间的长度；延迟进程直到将来某一时刻；针对超时编程可以识别出某些事件没有出现并进行处理。

2）时间性需求的表示。例如：规定执行的速度和时限。

3）时间性需求的满足。

本章主要是关于前两个主题的，虽然是以对时间概念本身进行的讨论开始。第13章研究系统的实现方法，以便能够预测最坏的时序行为，并满足时间性需求。 409

12.1 时间的概念

我们的日常经历都和过去、现在和未来的概念紧紧联系，但令人惊讶的是我们基本上还不能回答“时间是什么”这个问题。哲学家、数学家、物理学家还有最近工程师们都对“时间”做了精细的研究，但至今仍无法对时间的定义在理论上达成一致。正如St. Augustine所说：

> 什么是时间呢？如果没人问我，我知道它是什么。一旦我想对问我这个问题的人解释，我就不知道了。

哲学关于时间的讨论中的一个关键问题可以简单地表述为：“究竟是我们存在于时间里抑或时间是我们存在的一部分？”两大思想主流学派是还原学派和柏拉图学派。他们都同意人类史和生物史都是由有序的事件构成的。柏拉图学派的学者们认为时间是自然的一种基本属性，它是连续的，没有开始也没有结束，“并且拥有这些性质是一种必然”。我们关于时间的概念都是自历史事件到这些外部时间参照系的映射派生的。

比较起来，还原主义者没有利用这些外部参照系。由历史事件组成的历史时间是惟一有

意义的时间概念。他们假设一些确定的事件（例如，日出、冬至、原子振动等等）是定期发生的，由此发明了一种能测度时间长度的有效的时间参照。但是这只是人为构造的模型，并非实际存在的。

还原主义者观点的一个推论是，没有变化，时间将不能前进。如果宇宙诞生自大爆炸，那么那次爆炸就是第一个历史事件，所以时间自身连同“空间”从这个第一个纪元开始了。对于柏拉图学派的学者们而言，这次大爆炸也只是无穷无尽的时间长河中的一个事件。

爱因斯坦告诉我们，在远距离上相对论效应不但直接影响时间，而且还影响了事件的时序。在狭义相对论中，事件的观察者建立起一个参照系。某个观察者可能看到A事件先于B事件，而另一个观察者（在另一个参照系里）看到的次序可能会颠倒。由于时序的难题，爱因斯坦引入了**因果次序**。如果所有可能的观察者都看见A事件在B事件之前发生，我们说A事件导致了B事件。另一个确定这种因果关系的方法假设存在一个不超过光速的、从A事件传播到B事件的信号。

时间的这些不同主题可以通过同时事件（simultaneous event）很好地加以说明。对于柏拉图学派的人，同时事件发生在同一个时间，对于还原主义者来说，同时事件是“一起发生的”。在狭义相对论里，同时事件是不存在因果关系的事件。

从数学的观点看来，时间有很多不同的拓扑结构。最常见的一种是用一条实数线表示时

410 间。因此时间是线性的、传递的、非自反的和稠密的：

- 线性：$\forall x, y: x<y \vee y<x \vee x=y$
- 传递性：$\forall x, y, z: (x<y \wedge y<z) \Rightarrow x<z$
- 非自反性：$\forall x: \neg(x<x)$
- 稠密性：$\forall x, y: x<y \Rightarrow \exists z: (x<z<y)$

关于时间的工程看法极大地忽略了哲学的时间观点。一个嵌入式实时计算机系统要将它的执行和环境的“时间”进行协调。用“实时”这个词是为了表示和计算机时间的区别。“实”是因为它是外部的。这个外部的参照系是还原主义者的构造还是近似于柏拉图学派的“绝对”时间体系并不重要。而且，对于大多数的应用，相对论效应也是可以忽略的。在实时系统的数学拓扑结构方面，将时间视为稠密的还是离散的存在着一些分歧。因为计算机是在离散时间上工作的，所以建立基于离散时间的计算模型有好处。其他工程学科利用稠密时间模型取得良好效果的重要经验则支持相反的方法。结合两种方法的努力只不过产生了第三种方法——混合系统。如果一系列事件被广泛一致地认为是定期发生的，那么就有可能定义一个标准的时间测量。过去存在很多标准。表12-1简要描述了一些比较重要的标准。这些描述摘自参考文献[Hoogeboom and Halang（1992）]。

表12-1 时间标准

名字	描述	注
真实太阳日	两次连续中天（太阳最高点）之间的时间	一年中变化15分钟（近似）
瞬时小时	日出和日落之间时间的12分之一	一年中有可观变化
国际标准时（UT0）	格林尼治子午线的平均太阳时	1884年确定
秒（1）	平均太阳日的1/86 400	
秒（2）	1900回归年的1/31 566 925.974 7	1955年确定的历表时间
UT1	因极地运动对UT0的修正	

（续）

名　字	描　述	注
UT2	因地球旋转速度变化对UT1的修正	
秒（3）	9 192 631 770个对应于铯（caesium）133原子基态的两个超精细等级之间迁移的放射周期的时间长度	当前铯原子钟的准确度为10^{13}分之一（即每300 000年一个时钟误差）
国际原子时（IAT）	以铯原子钟为基础	
国际协调时（UTC）	一个IAT时钟，和增补了偶然移位时间信号的UT2同步	UT2（它是基于天文测量的）和UTC（它是基于原子测量的）之间的最大差值保持低于0.5秒

12.2 时钟访问

如果一个程序准备以任何有意义的方式与环境的时间框架交互，它就必须能够访问那些能“告诉时间”的方法，至少要有测量时间流逝的方法。有两种不同的方法来做到这一点：

- 直接访问环境的时间框架
- 通过内部硬件时钟对环境中的时间流逝做适宜的近似

第一种方法不常见，但可以通过很多手段来实现。最简单的方法可以为环境提供一个内部计时的定期中断。另一种极端是，系统（或者其实是分布式系统的任一节点）能装上无线电接收机接收某个国际时间信号。例如德国的一些转发器用来发射UTC和IAT信号。全球定位系统（GPS）也提供UTC服务。

从程序员的角度看来，对时间的访问可以由语言里的时间原语或内部时钟、外部时钟或者无线电接收机的设备驱动程序提供。设备驱动程序的编程是第15章的主题，下面的小节通过一些例子来说明occam、Ada、Java和POSIX是如何提供时钟抽象的。 411~412

12.2.1 occam2中的TIMER

任何一个occam2进程都可以通过读取**TIMER**获得本地时钟的值（没有访问日历时间的设施）。为了和occam2的（一对一）通信模型一致，每个进程必须使用不同的**TIMER**。读取**TIMER**要遵守读取管道的语法，但在语义上是不同的，读取**TIMER**不会导致挂起，即时钟总是准备输出。

```
TIMER clock:
INT Time:
SEQ
  clock ? Time     -- 读时间
```

TIMER产生一个**INT**类型的值，但其含义依赖于实现：它给出一个相对的、而非绝对的时钟值。因此单独一次读取**TIMER**是没有意义的，将两次读取的值相减就可以得到两次读取之间的时间长度。

```
TIMER clock:
INT old, new, interval:
SEQ
  clock ? old
  -- 其他计算
```

```
clock ? new
interval := new MINUS old
```

考虑到循环越界的情况，使用MINUS运算符而不用“-”。这是因为TIMER返回的整数对每一个时间单位都会加1，最终达到最大值，所以下一个滴答（tick）到达后这个整数就变成了最小的负值并继续增长。只要用户使用恰当的算术运算符（语言定义的）如MINUS、PLUS、MULT和DIVIDE，他们就可以不用关心这种动作。其实所有TIMER都是用一个“PLUS 1”操作来实现时钟的每次增加。

如上面所说明的，oocam2提供的设施是很基本的（虽然可以证明完全够用了）。由于只为时钟分配了一个整数，自然要在时钟的粒度（时钟每次滴答的时间间隔）和能够记录的时间范围之间作一个折衷。对于一个32位的整数来说，表12-2给出了典型的值。

表12-2 32位整数的TIMER粒度

粒 度	范围（近似）
1μs	71.6min
100μs	119h
1ms	50d
1s	136y

12.2.2 Ada的时钟包

Ada中对时钟的访问是由预定义（强制）的Calendar库包和一个可选的实时设施提供的。Calendar库包（见程序12-1）实现了一个抽象数据类型Time。该类型提供了一个读取时间的Clock函数和各种用于Time与人们理解的时间单位——年、月、日和秒——之间相互转换的子程序。前面三项是以整型的子类型给出的，但秒被定义成基本类型Duration的子类型。

程序12-1 Ada的Calendar包

```
package Ada.Calendar is

  type Time is private;

  subtype Year_Number is Integer range 1901..2099;
  subtype Month_Number is Integer range 1..12;
  subtype Day_Number is Integer range 1..31;
  subtype Day_Duration is Duration range 0.0..86400.0;

  function Clock return Time;

  function Year(Date:Time) return Year_Number;
  function Month(Date:Time) return Month_Number;
  function Day(Date:Time) return Day_Number;
  function Seconds(Date:Time) return Day_Duration;

  procedure Split(Date:in Time; Year:out Year_Number;
                  Month:out Month_Number; Day:out Day_Number;
                  Seconds:out Day_Duration);

  function Time_Of(Year:Year_Number; Month:Month_Number;
                  Day:Day_Number; Seconds:Day_Duration := 0.0)
                  return Time;
```

```
    function "+" (Left:Time;Right:Duration) return Time;
    function "+" (Left:Duration;Right:Time) return Time;
    function "-" (Left:Time;Right:Duration) return Time;
    function "-" (Left:Time;Right:Time) return Duration;

    function "<" (Left,Right:Time) return Boolean;
    function "<=" (Left,Right:Time) return Boolean;
    function ">" (Left,Right:Time) return Boolean;
    function ">=" (Left,Right:Time) return Boolean;

    Time_Error:exception;
    -- Time_Error 是由 Time_Of, Split," +" 和 "-" 引发的
  private
    -- 依赖于实现
  end Ada.Calendar;
```

类型Duration是预定义的定点实数类型，表示一段时间间隔（相对时间）。它的精度和表示范围都是依赖于实现的，不过它的范围必须至少是 -86 400.0 ~ 86 400.0，以保证能记录一天的秒数。它的粒度不能超过20ms。通常Duration型值的精度为秒。注意，除了上面的子程序以外，Calendar库包定义了Duration和Time参数组合的算术运算符和Time类型值的比较运算符。 413~414

测量执行一项计算所用时间的代码是很简单的。注意“-”运算符的用法，它取两个Time类型的值，返回一个Duration类型的值。

```
declare
  Old_Time, New_Time : Time;
  Interval : Duration;
begin
  Old_Time := Clock;
  -- 其他计算
  New_Time := Clock;
  Interval := New_Time - Old_Time;
end;
```

可选的Real_Time包提供了另一种语言时钟，它形式上和Calendar很相似，但是用于给出更精细的粒度。常量Time_Unit是Time类型表示的最小时间值。Tick值必须不大于一毫秒，Time的范围（从程序启动开始）必须至少是五十年。

除了提供更精细的粒度，Real_Time包的Clock被定义为单调的。Calendar时钟意在提供一个“挂钟”的抽象，提供闰年、闰秒和其他的调节。程序12-2简略描述了Real_Time包。

程序12-2 Ada的Real_Time包

```
package Ada.Real_Time is
  type Time is private;
  Time_First: constant Time;
  Time_Last: constant Time;
  Time_Unit: constant :=  -- 实现定义的实数;

  type Time_Span is private;
  Time_Span_First: constant Time_Span;
```

```
    Time_Span_Last: constant Time_Span;
    Time_Span_Zero: constant Time_Span;
    Time_Span_Unit: constant Time_Span;

    Tick: constant Time_Span;
    function Clock return Time;

    function "+" (Left: Time; Right: Time_Span) return Time;
    ...
    function "<" (Left, Right: Time) return Boolean;
    ...
    function "+" (Left, Right: Time_Span) return Time_Span;
    ...
    function "<" (Left, Right: Time_Span) return Boolean;
    ...
    function "abs" (Right : Time_Span) return Time_Span;

    function To_Duration (Ts : Time_Span) return Duration;
    function To_Time_Span (D : Duration) return Time_Span;

    function Nanoseconds  (Ns: Integer) return Time_Span;
    function Microseconds (Us: Integer) return Time_Span;
    function Milliseconds (Ms: Integer) return Time_Span;

  type Seconds_Count is range -- 实现定义

    procedure Split(T : in Time; Sc: out Seconds_Count;
                    Ts : out Time_Span);
    function Time_Of(Sc: Seconds_Count; Ts: Time_Span) return Time;
private
    -- 不由语言规定
end Ada.Real_Time;
```

12.2.3 实时Java中的时钟

标准Java支持挂钟的概念，因此具有和Ada相似的设施。在Java中调用java.lang包中的静态方法System.currentTimeMillis可以得到当前的时间。这个方法返回从格林尼治标准时间1970年1月1日午夜开始的毫秒数。java.util中的Date类在创建日期对象时默认使用这个方法。

415~416

实时Java为这些实时时钟设施添加了高精度的时间类型。程序12-3是这个高精度时间类的基本定义的一个概要。其中包括了读、写和比较时间值的方法。这个抽象类有三个子类：一个表示绝对时间，一个表示相对时间（类似于Ada的Duration类型），还有一个表示有理时间。程序12-4给出了它们的定义。绝对时间实际上是表示为从格林尼治国际标准时间1970年1月1日开始的时间。有理时间是相对时间类型，有一个与之相关联的频率。它被用于反映某些事件（例如周期线程的执行）发生的速度。

程序12-3 实时Java的HighResolutionTime类的摘要

```
public  abstract class HighResolutionTime implements
        java.lang.Comparable
```

```
{
  public abstract AbsoluteTime absolute(Clock clock,
                                        AbsoluteTime destination);

  ...

  public boolean equals(HighResolutionTime time);

  public final long getMilliseconds();
  public final int getNanoseconds();

  public void set(HighResolutionTime time);
  public void set(long millis);
  public void set(long millis, int nanos);
}
```

类HighResolutionTime、AbsoluteTime、RelativeTime和RationalTime从字面上就能很好理解。但absolute方法需要进一步的讨论。它的作用是将封装的时间（可能是绝对、相对或者有理时间）转换为相对于某个时钟的绝对时间。如果作为参数传递的是一个AbsoluteTime对象，这个对象就被更新，以反映封装的时间值。此外，这个更新后的对象也被这个函数返回。这是因为如果将一个空对象作为参数传递，就创建并返回一个新对象。在Java中参数是按值复制的，因而参数的值不会被修改。所以，这个函数有必要创建一个新的AbsoluteTime对象并返回它。

程序12-4　实时Java的AbsoluteTime、RelativeTime和RationalTime类

```
public class AbsoluteTime extends HighResolutionTime
{
  // 各种构造器方法，包括
  public AbsoluteTime(AbsoluteTime T);
  public AbsoluteTime(long millis, int nanos);

  public AbsoluteTime absolute(Clock clock, AbsoluteTime dest);

  public AbsoluteTime add(long millis, int nanos);
  public final AbsoluteTime add(RelativeTime time);
  ...
  public final RelativeTime subtract(AbsoluteTime time);
  public final AbsoluteTime subtract(RelativeTime time);
}

public class RelativeTime extends HighResolutionTime
{
  // 各种构造器方法，包括
  public RelativeTime(long millis, int nanos);
  public RelativeTime(RelativeTime time);

  public AbsoluteTime absolute(Clock clock, AbsoluteTime destination);

  public RelativeTime add(long millis, int nanos);
  public final RelativeTime add(RelativeTime time);

  public void addInterarrivalTo(AbsoluteTime destination);

  public final RelativeTime subtract(RelativeTime time);
```

```
  ...
}
public class RationalTime extends RelativeTime
{
  // 各种构造器方法，包括
public RationalTime(int frequency, RelativeTime interval)
       throws IllegalArgumentException;

public AbsoluteTime absolute(Clock clock, AbsoluteTime destination);

public void addInterarrivalTo(AbsoluteTime destination);
public int getFrequency();
public RelativeTime getInterarrivalTime(RelativeTime destination);
public void set(long millis, int nanos)
       throws IllegalArgumentException;
public void setFrequency(int frequency)
       throws ArithmeticException;
}
```

程序12-5中给出的实时Java的`Clock`类定义了派生所有时钟的抽象类。Java语言容许很多不同类型的时钟，例如，可以有一个测量正在用掉的执行时间的时钟。但是总会有一个与外
417~418 部世界同步的实时时钟。`getRealtimeClock`方法可以获取这个时钟[⊖]。该类还提供了其他的一些方法读取时钟的精度和设置时钟精度（如果硬件允许的话）。

程序12-5 实时Java Clock类

```
public abstract class Clock
{
  public Clock();

  public static Clock getRealtimeClock();

  public abstract RelativeTime getResolution();

  public AbsoluteTime getTime();
  public abstract void getTime (AbsoluteTime time);

  public abstract void setResolution(RelativeTime resolution);

}
```

测量执行某项运算所花时间的代码如下：

```
{
  AbsoluteTime oldTime, newTime;
  RelativeTime interval;
  Clock clock = Clock.getRealtimeClock();

  oldTime = clock.getTime();
  // 其他计算
  newTime = clock.getTime();

  interval = newTime.subtract(oldTime);
}
```

⊖ 这是一个静态方法，因此可以直接调用，而无须任何子类的知识。

12.2.4 C和POSIX中的时钟

ANSI C中有一个标准的库用于同“日历”时间对接。它定义了一个基本的时间类型time_t和几个处理时间类型对象的例程。程序12-6定义了其中几个函数。POSIX允许在一个实现里支持几个时钟。每个时钟都有自己的（clockid_t类型的）标识符，IEEE标准还要求至少支持一个时钟（CLOCK_REALTIME）。程序12-7是一个典型的POSIX时钟的C接口。时钟的返回值（通过clock_gettime）是一个timespec结构，tv_sec表示自从1970年1月1日以来经过的秒数，tv_nsec表示纳秒数（它是一个很大的数，但总是一个小于1 000 000 000的非负数——C没有提供子类型的机制）。POSIX要求CLOCK_REALTIME的最小精度是50Hz（20ms），函数clock_getres可以设置时钟的精度。

执行时间时钟将在12.8.1小节讨论。 419

程序12-6　ANSI C的日期和时间接口

```
typedef ... time_t;

struct tm {
  int tm_sec;      /* 分之后的秒数 - [0, 61] */
                   /* 61 允许 2 个闰秒 */
  int tm_min;      /* 小时之后的分钟数 - [0, 59] */
  int tm_hour;     /* 午夜后的小时数 - [0, 23] */
  int tm_mday;     /* 月的日 - [1, 31] */
  int tm_mon;      /* 自元月的月数 - [0, 11] */
  int tm_year;     /* 自 1900的年数 */
  int tm_wday;     /* 自星期日的日数 - [0, 6] */
  int tm_yday;     /* 自元月1日的日数- [0, 365] */
  int tm_isdst;    /* 改变夏令时间的标志 */
};
double difftime(time_t time1, time_t time2);
  /* 两个时间值相减 */
time_t mktime(struct tm *timeptr); /* 组成一个时间值 */
time_t time(time_t *timer);/* 如果timer不为空，返回当前时间*/
  /* 它还将该时间放到那个位置*/
```

程序12-7　C到POSIX时钟的接口

```
#define CLOCK_REALTIME ...;
#define CLOCK_PROCESS__CPUTIME_ID ...;
#define CLOCK_THREAD_CPUTIME_ID ...;

struct timespec {
  time_t tv_sec;    /* 秒数 */
  long   tv_nsec;   /* 纳秒数 */
};
typedef ... clockid_t;

int clock_gettime(clockid_t clock_id, struct timespec *tp);
int clock_settime(clockid_t clock_id, const struct timespec *tp);
int clock_getres(clockid_t clock_id, struct timespec *res);

int clock_getcpuclockid(pid_t pid, clockid_t *clock_id);
```

```
int clock_getcpuclockid(pthread_t_t thread_id, clockid_t *clock_id);

int nanosleep(const struct timespec *rqtp, struct timespec *rmtp);
/* 注意，如果休眠被中断，nanosleep 通过一个信号返回-1。*/
/* 在这种情况下，rmtp 拥有剩余的休眠时间   */
```

420

12.3 进程延迟

除了访问时钟外，进程还必须能够将其执行延迟一段相对时间或直到将来的某个绝对时间。

12.3.1 相对延迟

相对延迟使进程可以加入到某个未来事件的等待队列，而不必在访问时钟上忙等待。例如，下面的Ada代码表示一个任务是如何循环等待10s的。

```
Start := Clock;  -- 来自calendar
loop
  exit when (Clock - Start) > 10.0;
end loop;
```

为了减少对这种忙等待的需要，大多数的语言和操作系统都有某种形式的延迟原语。在Ada中，这是一个延迟语句：

```
delay 10.0;
```

delay后面的数值（Duration类型）是一个相对值（即上面的表达式表示从当前的时间开始延迟10s，负值当作零处理）。

在POSIX中，当需要的粒度不高时（比如说以秒计）可以利用“sleep”系统调用来延迟，如果要求更精细的粒度则使用“nanosleep”（见程序12-7）。第二个系统调用是按照CLOCK_REALTIME时钟计时的。Java提供了与POSIX相似的设施。Thread类的sleep方法可以使线程以毫秒级的粒度延迟自身。RealtimeThread类支持相对某个时钟的高精度休眠。

重要的是要理解延迟或者休眠仅仅能确保进程在设定的时间过后是可运行的[⊖]。在任务重新开始运行前的实际延迟当然和其他竞争处理器的进程有关。还应当注意，延迟的粒度和时钟的粒度不一定要相同。例如POSIX和实时Java允许到纳秒级的粒度，但现有的系统很少能支持到这个级别。还有，内部时钟也许是通过中断实现的，中断可能会被阻塞片刻。图12-

421 1说明了影响延迟的因素。

12.3.2 绝对延迟

使用Ada的delay（POSIX或实时Java中的sleep）可以支持一段相对时间的延迟（例如从现在起往后10s）。如果需要延迟到一个绝对的时间点，那么或是程序员自己计算出延迟时间的长度，或是系统提供另外的原语。例如，要使一个动作在另一动作开始之后10s执行，可以使用下面的Ada代码（使用Calendar）：

```
Start := Clock;
```

⊖ 事实上，在POSIX里如果收到信号，进程可以提前被唤醒。这时进程返回一个出错指示，rmpt参数返回剩下的延迟时间。Java里也有相似的情况。如果一个线程被中断了，它会被唤醒并且抛出InterruptedException对象。在Ada中，被延迟的任务只有在select-then-abort语句中才能被提前唤醒。

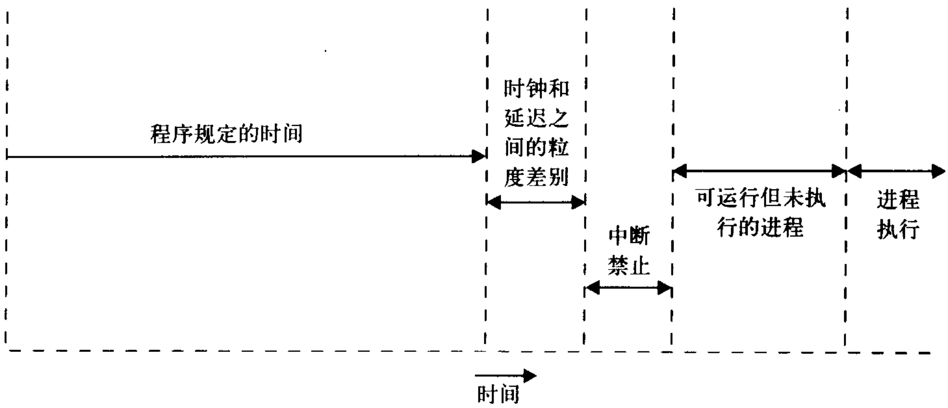


图12-1 延迟时间

```
First_Action;
delay 10.0 - (Clock - Start);
Second_Action;
```

但是，这段代码可能不会得到希望的结果。要让这种算法按照设想的行为执行，那么

```
delay 10.0 - (Clock - Start);
```

应该是一个不可中断（原子）动作，而事实上并不是。比如，假设First_Action用了2s，那么

```
10.0 - (Clock - Start);
```

就等于8s。但算出这个值后，如果这个任务被另一个任务抢占了，可能要过一段时间（比如说3s）它才能重新运行。这时它还是会延迟8s而不是5s。为了解决这个问题，Ada引入了**delay until**语句: 422

```
Start := Clock;
First_Action;
delay until Start + 10.0;
Second__Action;
```

像延迟一样，**delay until**产生的延迟只有下界是精确的。涉及的任务在这种语句设定的时间到达之前是不会从延迟中恢复的，但在设定的时间到达之后可以恢复。

相对和绝对延迟产生的时差叫做**局部漂移**，它是不能消除的。然而，有可能消除**积累漂移**，它是由局部漂移叠加产生的。下面的Ada代码说明了如何使Action运算平均每7s执行一次。这段代码会补偿任何局部漂移。例如，如果相邻的两次调用Action实际间隔7.4s，下一次延迟就会只有6.6s（近似值）。

```
declare
  Next : Time;
  Interval : constant Duration := 7.0;
begin
  Next := Clock + Interval;
  loop
    Action;
    delay until Next;
    Next := Next + Interval;
```

```
  end loop;
end;
```

实时Java的sleep方法（定义在RealtimeThread类里）可用于相对和绝对两种延迟。

occam2只支持绝对延迟。为了强调其可调整的语义，它使用了关键字AFTER。如果一个进程要等待10s，它必须首先读取TIMER时钟，加上10s，然后延迟到那个时刻。在这段程序里我们引入了常量G来给出当前实现的粒度（G是TIMER每秒更新的次数），增加的10s就是通过G得到的

```
SEQ
  clock ? now
  clock ? AFTER now PLUS (10 * G)
```

occam2中避免累积漂移的代码如下：

423

```
INT next, now:
VAL interval IS 7*G:
SEQ
  clock ? now
  next := now PLUS interval
  WHILE TRUE
    SEQ
      ACTION
      clock ? AFTER next
      next := next PLUS interval
```

利用一个绝对定时器和等待定时到期时发出的信号，POSIX也可以构造绝对延迟（见12.8.1小节）。

12.4 超时的编程

也许对于嵌入式系统最简单的时间约束是要求能够对外部事件没有发生的事实做出识别和反应。例如，一个温度探测器可能需要每秒记录一次读数，规定10s之内没有读数为故障。一般来说，**超时**（timeout）是进程准备为等待一次通信所花时间的限制。第8章和第9章已经详细讨论了进程间通信的设施。任何一种在实时环境下使用的通信设施都需要用到超时。

除了等待通信外，动作的执行也要需要超时。比如说一个程序员也许需要一段代码在确定的时间内执行。如果在运行时没有做到这一点，可能就需要某种出错恢复。

12.4.1 共享变量通信和超时

在第8章里讨论了各种基于共享变量的通信和同步机制。临界段的互斥访问和条件同步都是很重要的需求。如果要访问已经有其他进程活动的临界段，进程就会被阻塞。进程被阻塞的时间取决于临界段代码的运行时间和希望进入临界段的其他进程的数目。因此，并不认为必须有一个关于试图进入临界段的超时（然而，POSIX确实提供了这种超时）。13.10节会详细研究分析这种阻塞时间的问题。

同临界段互斥相比，与条件同步相关的阻塞取决于特定的应用并且难于确定。例如，一个试图在一个满的缓冲区放置数据的生产者进程必须等待消费者进程取出数据来腾出空间。而消费者进程可能在相当的一段时间里没能做好读取数据的准备。因此，让进程能够在等待条件同步的时候有超时选择是很重要的。

第8章讨论的条件同步设施包括:

- 信号量
- 条件临界区
- 管程、互斥锁和同步方法中的条件变量
- 保护对象的入口 424

实时Euclid（Kligerman and Stoyenko，1986）使用信号量并且扩充了wait的语义使之包含时间界。下面这个表达式说明了一个进程如何在一个超时值为10s的CALL信号量下挂起自己的:

```
wait CALL noLongerThan 10 : 200
```

如果这个进程在10s之内没有收到信号，就会引发异常200（实时Euclid中的异常是按数字编号的）。然而，要注意这条语句并没有要求进程真的在10s之内被调度并执行。所有的超时设施都是这样的，它们只是指出进程应该转到可运行状态。

POSIX支持等待信号量或者条件变量的显式超时（见程序8-2和8-3）。下面的语句说明了一个进程如何在一个绝对超时值为timeout的信号量call上挂起自己:

```
if (sem_timedwait(&call, &timeout) < 0) {
  if (errno == ETIMEDOUT) {
    /* 发生了超时 */
  }
  else {
    /* 其他出错 */
  }
} else {
  /* 锁上信号量 */
};
```

如果这个信号量没有被timeout锁住，errno就被置为ETIMEDOUT。

Ada的保护对象使用了回避同步，因此在保护入口调用的地方必须使用超时。由于Ada用和任务入口调用相同的方式处理保护入口调用，所以需要使用相同的超时机制。下面将从消息传递的角度来描述这种方法。

对于Java来说，wait方法可以使用毫秒粒度或纳秒粒度的超时。

12.4.2 消息传递和超时

第9章已经讨论了几种不同的消息传递方式。所有的方式都需要用到同步。即便是在异步系统里，一个进程也会希望等待某个消息（或发送进程的消息缓冲区装满了）。如果利用同步消息传递，进程一旦参与一次通信就必须一直等待到通信结束。因此要有超时设施。为了说明关于超时的编程，首先考虑一个控制器任务，它被某个另外的驱动器任务调用，并返回新的温度读数（Ada代码）: 425

```
task Controller is
  entry Call(T : Temperature);
end Controller;

task body Controller is
  -- 声明
begin
  loop
```

```
      accept Call(T : Temperature) do
        New_Temp := T;
      end Call;
      -- 其他动作
    end loop;
  end Controller;
```

现在需要修改一下这个控制器，使得缺失的入口调用得以起作用。我们可以利用前面讨论过的构造提供这个需求。第二个任务用于延迟自己一段超时时间，然后调用控制器。如果控制器在正常的Call调用前被该任务调用，那么就发生了超时。

```
task Controller is
  entry Call(T : Temperature);
private
  entry Timeout;
end Controller;

task body Controller is
  task Timer is
    entry Go(D : Duration);
  end Timer;
  -- 其他声明
  task body Timer is
    Timeout_Value : Duration;
  begin
    accept Go(D : Duration) do
      Timeout_Value := D;
    end Go;
    delay Timeout_Value;
    Controller.Timeout;
  end Timer;
begin
  loop
    Timer.Go(10.0);
    select
      accept Call(T : Temperature) do
        Mew_Temp := T;
      end Call;
    or
      accept Timeout;
      -- 超时动作
    end select;
    -- 其他动作
  end loop;
end Controller;
```

426

尽管这段代码只处理首次超时时段，但能够把它加以修改（很可观的改动）使之有持续性。此外，由于需要用到超时的情况很普遍，因此希望有一个更简明地表示它的方式。在实时语言里，它常常作为选择性等待的一种特别备选形式提供。上面的例子可以更恰当地编码如下：

```
task Controller is
  entry Call(T : Temperature);
```

```
end Controller;

task body Controller is
  -- 声明
begin
  loop
    select
      accept Call(T : Temperature) do
        New_Temp := T;
      end Call;
    or
      delay 10.0;
      -- 超时动作
    end select;
    -- 其他动作
  end loop;
end Controller;
```

当延迟时间到期后，延迟备选变为就绪状态。一旦这个备选被选中（即10s内还没有调用Call），就执行delay后面的语句。

上面的例子使用了相对延迟，但也可用绝对延迟。考虑下面的代码，它使一个任务能一直到Closing_Time时刻之前都接受登录。

```
task Ticket_Agent is
  entry Registration(...);
end Ticket Agent;

task body Ticket_Agent is
  -- 声明
  Shop_Open : Boolean := True;
begin
  while Shop_Open loop
    select
      accept Registration(...)do
        -- 登录的具体情况
      end Registration;
    or
      delay until Closing_Time;
      Shop_Open := False;
    end select;
    -- 处理登录
  end loop;
end Ticket_Agent;
```

427

在Ada模型中，else部分、终止备选和延迟备选是不能混合使用的。这三种结构是互斥的，因此一个选择语句最多只能有其中的一个。但是如果是使用延迟备选，则选择语句可以有很多延迟备选，前提是它们必须都是同一种延迟（都是delay或都是delay until）。每次只有延迟时间最短的或绝对到达时间最早的才可能被执行。

超时设施在基于消息的并发编程语言里很常见。和Ada一样，occam2使用delay原语作为声明超时的选择性等待的一部分：

```
WHILE TRUE
  SEQ
    ALT
      call ? new_temp
        -- 其他动作
      clock ? AFTER (10 * G)
        -- 超时动作
```

这里时钟是TIMER。

occam2和Ada的例子都说明了如何编写关于消息接收的超时。Ada还进一步容许消息发送的超时。为了说明这一点，设想一个在上面的Ada代码中将温度读数送往控制器的设备驱动程序:

```
loop
  -- 取来新温度 T
  Controller.Call(T);
end loop;
```

由于新的温度读数在不断地产生（并且把过时的数据提供给控制器是没有意义的），设备驱动程序可能希望在撤销这个调用前为等待控制器只挂起半秒钟。通过使用一个具有单一入口调用和单一延迟备选的特殊形式的选择语句可以实现这一点:

```
loop
  -- 取来新温度 T
  select
    Controller.Call(T);
  or
    delay 0.5;
    null;
  end select;
end loop;
```

这个**null**不是一定必须的，但它表示了delay后面可以有任意的语句，如果在入口调用被接收之前延迟时间已到，就会执行这些语句。这是一种特殊的选择形式，它不能有一个以上的入
428 口调用也不能混合入口调用和接受语句。它调用的动作叫做**限时入口调用**。必须要强调的是，在这个调用里规定的时间是被接收的调用的超时值，*它不是相关联的接受语句终止的超时值*。

如果一个任务只是当被调用的任务立即响应调用时才发出入口调用，那么就不要发出时间为零的限时入口调用，而要发出**条件入口调用**:

```
select
  T.E;  -- 任务T的入口E
else
  -- 其他动作
end select;
```

只有当任务T没有准备好立即响应E时，“其他动作”才会执行。“立即”意味着T已经被挂在**accept** E或者一个具有打开备选（并被选中）的选择语句上。

上面的例子已经利用进程间通信的超时，同样可以在Ada中的保护对象上进行限时或条件入口调用:

```
select
```

```
  P.E;    -- E是保护对象P中的一个入口
or
  delay 0.5;
end select;
```

12.4.3 动作上的超时

在10.5节讨论了允许进程用异步通知改变其控制流的问题。超时可以被看作是这样的一种通知，因此如果支持异步通知，就可以使用超时。恢复模型（异步事件）和终止模型（异步控制转移）都可以增补超时。然而，对于动作上的超时来说，需要的是终止模型。

例如，在10.8节介绍了Ada的异步控制转移（ATC）设施。在ATC中，如果在动作完成之前发生了触发事件，这个动作可被中止。其中一个允许的触发事件是时间的流逝。为了说明这一点，考虑一个任务，它包含一个必须在100ms内完成的动作。下面的代码直接支持这个需求：

```
select
  delay 0.1;
then abort
  -- 动作
end select;
```

如果这个动作花了太长的时间，触发事件就会发生，并且这个动作将被中止。显然这是一个捕捉“失控代码”的有效方法。 429

超时常常同出错状态相关，如果通信没有在Xms内发生，就可能发生了意外的事情，必须采取矫正动作。但这并不是它惟一的用途。考虑有一个强制部分和一个可选部分的任务。强制部分的计算（很快地）得到满意的结果，赋给一个保护对象。这个任务必须在固定的时间里完成，但如果在强制部分计算完成后还有时间，可选算法可以增量式地改进输出的值。要编写出这样的程序同样需要动作上的超时。

```
declare
  Precise_Result : Boolean;
begin
  Completion_Time := ...
  -- 强制部分
  Results.Write(...); -- 调用外部保护对象中的过程
  select
    delay until Completion_Time;
    Precise_Result := False;
  then abort
    while Can_Be_Improved loop
      -- 改进结果
      Results.Write(...);
    end loop;
    Precise_Result := True;
  end select;
end;
```

注意，即使在写保护对象过程中超时时间已过，它还是会正确完成它的操作，这是因为对保护对象的调用是一个中止延期动作（即中止的影响推迟到任务离开该保护对象之后）。

在实时Java里，动作上的超时是由一个名为Timed的AnsychronouslyInterrupted Exception子类提供的。在程序12-8里定义了这个类（见14.4.3小节关于java.io.Serializable的信息）。

程序12-8 实时Java的类Timed

```
public class Timed extends AsynchronouslyInterruptedException
                                        implements java.io.Serializable
{
  public Timed (HighResolutionTime time) throws IllegalArgumentException;

  public boolean doInterruptible(Interruptible logic);

  public void resetTime (HighResolutionTime time);
}
```

430 上面Ada的例子可以用实时Java写成：

```
public class PreciseResult
{
  public resultType value; // 结果
  public boolean preciseResult; // 指出它是否不精确
}

public class ImpreciseComputation
{
  private HighResolutionTime CompletionTime;
  private PreciseResult result = new PreciseResult();

  public ImpreciseComputation(HighResolutionTime T)
  {
    CompletionTime = T;  // 可以是绝对的或相对的
  }

  private resultType compulsoryPart()
  {
    // 计算强制部分的函数
  };

  public PreciseResult Service()  // 公共服务
  {
    Interruptible I = new Interruptible()
    {
      public void run(AsynchronouslyInterruptedException exception)
                throws AsynchronouslyInterruptedException
      {
        // 这是一个改善强制部分的可选函数
        boolean canBeImproved = true;

        while(canBeImproved)
        {
          // 改进结果
          synchronized(this) {
            // 写结果 —— 同步语句确保写操作的原子性
          }
```

```
            }
            result.preciseResult = true;
          }
          public void interruptAction(
                 AsynchronouslyInterruptedException exception)
          {
            result.preciseResult = false;
          }
        };
        Timed t = new Timed (CompletionTime);
        result.value = compulsoryPart();  // 计算强制部分
        if(t.doInterruptible (I))
          //执行timer的可选部分
          return result;
        else...;
      }
    }
```

431

超时是实时系统的重要特性，但它远非惟一重要的时间约束。这一章余下部分以及下一章讨论时限这个更广泛的主题以及如何确保满足它。

12.5 规定时间性需求

对于许多重要的实时系统来说，软件在逻辑上正确还不够，程序还必须满足底层物理系统所确定的时间性约束。这些约束远远超出了简单的超时。令人遗憾的是，现有关于大型实时系统的工程实践基本上还只是为特定目的的。通常一个逻辑正确的系统被规定、设计和构造（也许只是一个原型）之后，就通过测试看看是否满足它的时间性需求。如果不满足，接下来就是各种细微的调试和重写。其结果是一个难于理解和维护升级费用很高的系统。因此我们需要更加系统化处理时间的方法。

关于更加严格地处理实时系统的时间性需求的研究工作基本上有两种不同的方法。一种发展的方向是考虑使用形式化定义的语言语义和时间性需求，配合以能表示和分析时序属性的记号和逻辑。另一种是关注基于在现有资源（处理器等等）上调度所需工作负荷的可行性上的实时系统的性能。

这本书主要集中在后一种工作上。这有三个原因：首先，形式化技术还没成熟到能对大型的复杂实时系统进行推理；其次，很少报道在实际的实时系统中使用这种技术的经验；最后，完整地讨论这个方法要介绍大量超出本书范围的素材。这并不是意味着这一领域与实时系统无关。对建立在CSP、时态逻辑、实时逻辑、模型检查上的形式化技术和融入时间概念的规格说明技术的理解正变得越来越重要。例如RTL（实时逻辑）能用来验证系统的时序需求，从而补充了在分析功能需求时VDM和Z之类方法的应用。

模型检查在验证功能特性中的成功最近扩展到了实时领域。系统被看作是一个定时自动机（即带时钟的有限状态机）的模型，模型检查用于“证明”不会到达非理想状态，或在某个内部时钟到达临界时间（时限）之前进入理想的状态。后面的特性称为*有界活性*（bounded liveliness）。尽管模型检查的探索受制于状态爆炸，还是有一些现有的工具能处理一定规模的问题。这项技术可能在未来几年内成为标准工业方法。 432

实时系统的验证可以从使用上解释成一个两步过程:

1）需求/设计验证——给定一个无限高速可靠的计算机，时序需求是否一致连贯？换句话说，它们都有被满足的可能吗？

2）实现验证——通过一套有限的（可能是不可靠的）硬件资源，能否满足时序需求？

如上所述，第一个问题可能要用到形式化推理（或模型检查或者两者同时使用）来验证能否满足必需的时序（和因果）次序。例如，假设A事件必须在B事件之前结束，但它依赖于发生在B事件之后的某个C事件，这时不管处理器有多快都不可能满足这些需求。因此早一点认识到这种困难是很有用的。第二个问题（实现验证）是后面一章的主题。这一章剩下的部分将主要讨论如何用语言表示时序需求。

12.6 时序作用域

为了便于规定在实时应用中的各种不同时间约束，引入**时序作用域**（temporal scope）的概念是很有用的。这种作用域标识一组语句和与之相关联的一个时间约束。图12-2描述了时序作用域（TS）可能的属性，它们包括:

1）时限——TS的执行必须完成的时间点;

2）最小延迟——TS开始执行前必须经过的最短时间;

3）最大延迟——TS开始执行前容许经过的最长时间;

4）最长执行时间——TS的最长执行时间;

5）最长存在时间——TS的最长存在时间。

具有这些属性的组合的时序作用域也是可能的。

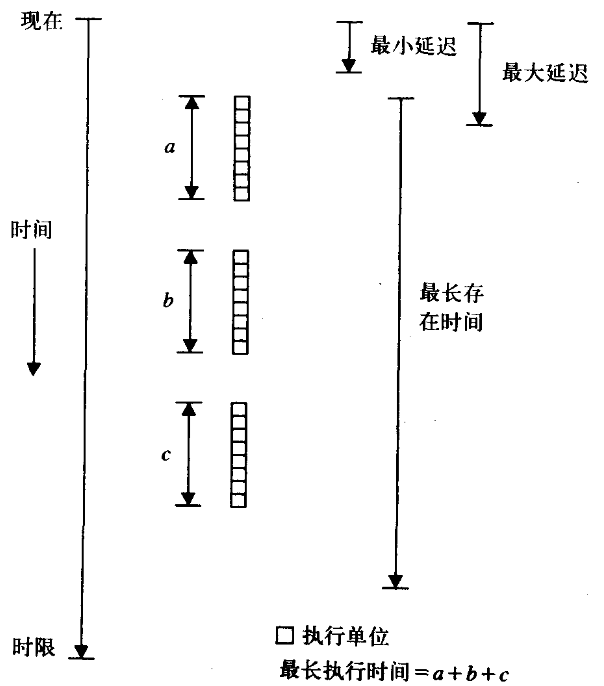


图12-2 时序作用域

时序作用域可以被描述成**周期的**或者是**非周期的**。周期时序作用域通常进行数据采样，或者执行一个控制循环，并有必须满足的显式时限。使用非周期或**偶发的**时序作用域常常是因为出现在嵌入式计算机之外的异步事件。这些作用域都有自己特定的响应时间。

通常认为非周期时序作用域是被随机激活的并服从泊松（Poisson）分布。这种分布允许外部事件集中到达，并不排除非周期活动的任何可能的集中。因此无法做最坏情况分析（在给定的时间发生任何数目的非周期事件都有非零概率）。为了可以做最坏情况估算，常常定义两次非周期事件（同一来源）之间的最短时间。如果是这种情况的话，涉及的进程就称为**偶发的**。本书一般情况下使用“非周期的”这个术语，“偶发的”则用于需要最小延迟的情况。

在很多实时语言里，时序作用域实际上与体现它们的进程相关联。进程依据其内部时序作用域的属性分为周期的、非周期的或是偶发的。因此前面列出的大部分时间属性可以这样满足：

1）以正确的速率运行周期性进程；

2）在时限之前完成所有的进程。 433~434

因此满足时间约束的问题就变成了一个调度进程满足时限的要求或者**时限调度**问题。将时序作用域分解为两个优先性相关的进程可以实现最大延迟需求。第一个进程代表TS的早期阶段，因此可以有一个时限保证不违反最大开始延迟。需要这种结构的应用来自一个先读出传感器的读数，然后产生一个输出的计算。为了精确控制读传感器的时间，这个初始动作需要一个严格的时限。这样，输出可以在后一个时限到达之前产生。传感器读入或者制动器值输出时间的变化叫做**输入抖动**和**输出抖动**。

尽管所有的计算机系统都尽力做到高效，并且很多被描述成实时的，还是需要进一步的分类，以充分区分各种应用中时间性的不同重要程度。如在这本书的引言部分指出的，如果系统的时限必须不能错过否则系统就会失败，我们就称这种系统是**硬**实时的。相比之下，如果系统容许时限错过，那么它是**软**实时的。如果没有特定的时限而只是力求“足够的响应时间”，则系统仅仅是**交互式的**。更精确一点，具有软时限的进程仍会在时限错过后提供服务——系统仍然从推迟的服务中获取某种有用的东西。具有确定的严格时限（即推迟的服务是无用的或无价值的）的非硬实时进程称作**固**实时进程。

在容错系统中，硬、固和软实时的区别变得有点模糊不清。然而，如果错过的时限触发一个特定的出错恢复（故障保护）例程就使用术语“硬实时”，如果应用的本性容许偶尔错过时限或者稍微错过时限就使用术语“软实时/固实时”。最后，要注意很多硬实时系统会有一些软时限和固时限。

规定进程和时序作用域

在实时系统里，需要显式地处理时间性需求，前面给出了五种类型。下面是一个周期性进程的一般框架：

```
process periodic_P;
  ...
begin
  loop
    IDLE
    时序作用域的开始
      ...
```

```
        时序作用域的结束
      end;
    end;
```

435 时间约束采用了IDLE的最长和/或最短时间的形式，并包括了在某个时限之前结束这个时序作用域的需求。这种时限本身可以用下面任一种方式表示:

- 绝对时间
- 自时序作用域开始起的执行时间
- 自时序作用域开始起流逝的时间

如果这个进程正在采样数据，采样的动作应该放在该时序作用域的开始，因此IDLE周期的准确性很重要。这个时序作用域应该包含对这个数据的必要处理（或仅仅是缓冲），它结束的时限只是保证该进程能够进入下一轮循环并及时进行下一个读取。

对于一个发送定时控制信号的进程来说，时序作用域加入所有计算信号值所需的运算（可能包括获取外部读入）。这个信号在该时序作用域结束时发送出去，因此时限同这个事件相关联。

对于非周期进程来说，类似的时限也是必要的，此时时序作用域是由通常以中断形式出现的外部事件触发的。

```
process aperiodic_P;
  ...
begin
  loop
    等待中断;
    时序作用域的开始
      ...
    时序作用域的结束
  end;
end;
```

显然周期性进程有一个确定的周期（即进程循环执行的频率），这个测量可能同样被应用在非周期进程上，这时它意味着该进程循环的最大速率（中断到来的最大速率）。根据前面提到的，这样的非周期进程被称为偶发的。

在某些实时系统里，时限可能和在一些进程间传递的数据相关（这通常被称为实时事务）。为了调度这些进程，必须在操纵这些数据的进程之间分割可用的时间。如果各个进程处理数据的次数是动态的（就是依赖于数据的）或者更糟糕，数据在进程间传递的路径也是依赖于数据的，时间的分割可能变得十分复杂。当前没有实时语言明确地致力于这个问题。

12.7 时序作用域的语言支持

在下面几节里，（从对时序作用域支持的角度）描述了一些语言。所选择的语言代表了现
436 在可用的各种工程语言:

- Ada、occam2和C/POSIX
- 实时Euclid和Pearl
- 实时Java

- DPS
- Esterel

12.7.1 Ada、occam2和C/POSIX

Ada、occam2 和 C/POSIX同许多其他实时语言一样（显著的例外在后面讨论），都不支持带时限的周期或非周期进程的显式规定，而是必须在一个循环进程里使用延迟原语、定时器等（然而，POSIX允许设置周期性定时器）。

例如，在Ada里一个周期性任务必须采用下面的形式：

```
task body Periodic_T is
  Next_Release : Time;
  Release_Interval : constant Duration := ...; -- 或者
  Release_Interval : constant Time_Span := Milliseconds(...);
begin
  -- 读时钟并计算下一个启动时间 (Next_Release)
  loop
    -- （例如）数据采样，或计算和发送一个控制信号
    delay until Next_Release;
    Next_Release := Next_Release + Release_Interval;
  end loop;
end Periodic_T;
```

和Ada相比，更加冗长的C/POSIX表示要求显式地使用定时器和信号处理。

```
#include <signal.h>
#include <time.h>
#include <pthread.h>
void periodic_thread() /* 肯定是该线程*/
{
  int signum;                     /* 捕获的信号 */
  sigset_t set;                   /* 要等待的信号*/
  struct sigevent sig;            /* 信号信息 */
  timer_t periodic_timer;              /* 周期性线程的定时器 */
  struct itimerspec required, old; /* 定时器详细情况 */
  struct timespec first, period;    /* 开始并重复 */
  long Thread_Period = ....       /* 按纳秒的实际周期 */

  /*设置信号接口 */
  sig.sigev_notify = SIGEV_SIGNAL;
  sig.sigev_signo = SIGRTMIN; /* 例如 */
  /* 例如，对系统初始化而言，允许从现在开始的1 秒钟*/
  CLOCK_GETTIME (CLOCK_REALTIME, &first);     /* 获得当前时间 */
  first.tv_sec = first.tv_sec + 1;
  period.tv_sec = 0;              /* 为period 设置重复时间*/
  period.tv_nsec = Thread_Period;
  required.it_value = first;             /* 初始化定时器的细节*/
  required.it_interval = period;
  TIMER_CREATE(CLOCK_REALTIME, &sig, &periodic_timer);
  SIGEMPTYSET (&set);                 /* 初始化信号set, 置为 null */
  SIGADDSET(&set, SIGRTMIN);         /* 只允许定时器中断 */
  TIMER_SETTIME(periodic_timer, TIMER_ABSTIME, &required, &old);
```

```
    /* enter periodic loop */
    while(1) {
      SIGWAIT(&set, &signum);
      /* 这里是每个周期要执行的代码*/
    }
  }

  int init ()
  {
    pthread_attr_t attributes;       /* 线程属性*/
    pthread_t PT;                    /* 线程指针 */

    PTHREAD_ATTR_INIT(&attributes); /* 默认属性 */
    PTHREAD_CREATE (&PT, &attributes,
                    (void *) periodic_thread, (void *)0);
  }
```

由中断触发的Ada偶发任务不会包含显式的时间信息，但通常会用一个保护对象去处理中断并启动任务的执行。

```
protected Sporadic_Controller is
  procedure Interrupt;   -- 被映射到中断
  entry Wait_For_Next_Interrupt;
private
    Call_Outstanding : Boolean := False;
end Sporadic_Controller;

protected body Sporadic_Controller is

  procedure Interrupt is
  begin
    Call_Outstanding := True;
  end Interrupt;

  entry Wait_For_Next_Interrupt when Call_Outstanding is
  begin
```

437 ~ 438

```
    Call_Outstanding := False;
  end Wait_For_Next_Interrupt;
end Sporadic_Controller;
task body Sporadic_T is
begin
  loop
    Sporadic_Controller.Wait_For_Next_Interrupt;
    -- 动作
  end loop;
end Sporadic_T;
```

上面的例子表明在Ada（以及其他许多所谓的实时语言）里，惟一能保证满足的时间约束是在时序作用域开始之前的最少时间。它是通过延迟原语实现的。

12.7.2 实时Euclid 和Pearl

支持时限调度的语言，都有一些用于表达时限规格说明的合适时间性原语。在实时Euclid（Kligerman and Stoyenko，1986）中进程是静态的并且不可嵌套。每个进程定义必须包含适

合自己实时行为的激活信息（用“**框架**”(frame）这个术语代替时序作用域)。这个信息采用下面与周期和偶发进程有关的两种形式中的一种:

1）周期性*frameInfo*首先激活*timeOrEvent*

2）atEvent *conditionId frameInfo*

*frameInfo*子句定义进程的周期（包括偶发进程的最大速率)。它最简单的形式是一个按实时单位的表达式:

```
frame realTimeExpn
```

这些单位的值在程序的开始处设置。

一个周期进程可以以两种方式被首次激活。它可以有一个为它定义的开始时间或者等待一个中断发生。此外它还可以等待这些情况中的任一种。因此*timeOrEvent*的语法必须是下面的一种:

(1) atTime *realTimeExpn*

(2) atEvent *conditionId*

(3) atTime *realTimeExpn*或atEvent *conditionId*

*ConditionId*是一个和中断相关联的条件变量。它也用于偶发进程。

为了给出一个部分实时Euclid程序的例子，研究一个循环温度控制器。它的周期是60个单位（即，如果时间单位设为一秒，就是每分钟一次）并且它将在600个单位（10min）后或当*startMonitoring*中断到达时变成活动的: 439

```
realTimeUnit := 1.0 % 时间单位 = 1 sec

var Reactor: module % Euclid 是基于模块的
var startMonitoring : activation condition atLocation 16#A10D
% 这定义一个条件变量，它被映射到中断上

  process Tempcontroller : periodic frame 60 first activation
                         atTime 600 or atEvent startMonitoring
  % 导入列表
  %
  % 执行部分
  %
  end TempController
end Reactor
```

注意这个进程里没有循环。是调度程序控制所需的和规定的周期性执行。

为了说明这段代码如何用Ada构造，给出这个进程（任务）的片断（这里需要一个循环，强制任务循环）如下:

```
task body Tempcontroller is
  -- 定义，包括
  Next_Release : Duration;
begin
  select
    accept Startmonitoring; -- 或是一个保护对象上的限时入口调用
  or
    delay 600.0;
  end select;
```

```
    Next_Release := Clock + 60.0; -- 留意下一个启动时间
    loop
      -- 执行部分
      delay until Next_Release;
      Next_Release := Next_Release + 60.0;
    end loop;
  end Tempcontroller;
```

不但代码更麻烦，而且调度程序也不知道这个任务的时限。它的正确执行将依赖于在延迟到期后任务能否几乎立即变成活动的。

Pearl

Pearl语言（Werum and Windauer，1985）同样支持关于进程开始、频率和终止的显式的时间信息。一个简单的周期为10s的任务T可以这样激活：

440
```
EVERY 10 SEC ACTIVATE T
```

要在一个特定的时间点（比如每天的中午十二点）激活：

```
AT 12:00:00 ACTIVATE LUNCH
```

由中断IRT启动的偶发任务S可以定义为：

```
WHEN IRT ACTIVATE S;
```

或者需要一个1s的初始延迟：

```
WHEN IRT AFTER 1 SEC ACTIVATE S;
```

尽管语法不同，Peral提供了和描述过的实时Euclid几乎一样的功能。然而，温度控制器的例子说明了一个显著区别。一个Pearl任务可以被时间调度或者中断激活但不能两者同时使用。所以下面的每一种方式在Pearl都是可接受的：

```
AFTER 10 MIN ALL 60 SEC ACTIVATE TempController;

WHEN startMonitoring ALL 60 SEC ACTIVATE TempController;
```

`All 60 SEC`表示在首次执行后每60s周期性地重复。

12.7.3 实时Java

实时Java在面向对象的框架里提供与实时Euclid和Pearl类似的支持。被调度的对象必须实现`Schedulable`接口，这个内容将在下一章涉及。除此之外，对象必须：

- 通过类`MemoryParameters`规定存储器的需求——在第15章讨论；
- 通过类`SchedulingParameters`规定调度需求——在第13章讨论；
- 通过类`ReleaseParameters`规定时间性需求。

最后一个是通过`ReleaseParameters`类的层次体系实现的。程序12-9定义的抽象基类给出了所有可调度对象所需的一般参数。可调度对象可以有一个和每次启动执行相关联的时限和时间开销。这个时间开销是调度程序应该给这个对象的执行时间长度。如果在它的时限或者时间开销到期后对象还在运行，相应的事件处理程序就被调度执行。应当注意实时Java
441 并不要求实现支持对执行时间的监视。但它确实要求实现检测错过的时限。也许还应当注意，非周期和偶发线程的启动事件当前没有明确定义。因此，很难知道实现如何检测时限超期。当然，程序可以传递一个空的处理程序表明它不考虑错过的时限。

程序12-9 类ReleaseParameters

```
public abstract class ReleaseParameters
{
  protected ReleaseParameters (RelativeTime cost, RelativeTime deadline,
                               AsyncEventHandler overrunHandler,
                               AsyncEventHandler missHandler);
  public RelativeTime getCost();
  public AsyncEventHandler getCostOverrunHandler();

  public RelativeTime getDeadline();
  public AsyncEventHandler getDeadlineMissHandler();

  public void setCost(RelativeTime cost);
  public void setCostOverrunHandler(AsyncEventHandler handler);

  public void setDeadline(RelativeTime deadline);
  public void setDeadlineMissHandler(AsyncEventHandler handler);
}
```

程序12-10说明ReleaseParameters类的子类支持周期、非周期和偶发启动参数。所有的相对时间值都相对于相应线程开始（启动）的那个时间点。

利用上面的参数类可以表示下面可调度对象的时间特性:

- RealtimeThread
- NoHeapRealtimeThread——在13.14.3小节讨论
- AsyncEventHandler

程序12-10 类PeriodicParameters、AperiodicParameters和SporadicParameters

```
public class PeriodicParameters extends ReleaseParameters
{
  public PeriodicParameters (HighResolutionTime start,
         RelativeTime period, RelativeTime cost,
         RelativeTime deadline, AsyncEventHandler overrunHandler,
         AsyncEventHandler missHandler);

  public RelativeTime getPeriod();
  public HighResolutionTime getStart();
  public void setPeriod(RelativeTime period);
  public void setStart(HighResolutionTime start);
}

public class AperiodicParameters extends ReleaseParameters
{
  public AperiodicParameters (RelativeTime cost,
         RelativeTime deadline, AsyncEventHandler overrunHandler,
         AsyncEventHandler missHandler);
}

public class SporadicParameters extends AperiodicParameters
{
  public SporadicParameters(RelativeTime minInterarrival,
         RelativeTime cost, RelativeTime deadline,
```

```
        AsyncEventHandler overrunHandler,
        AsyncEventHandler missHandler);

  public RelativeTime getMinimumInterarrival();
  public void setMinimumInterarrival(RelativeTime minimum);
}
```

实时线程

实时Java线程是基本语言线程的一个扩展。程序12-11定义了这个类（ProcessingGroupParameters在13.8.2小节讨论）。在创建一个实时线程时可以没有启动参数。事实上，它们没有特别的时间需求。

程序12-11 类RealtimeThread的摘要

```
public class RealtimeThread extends java.lang.Thread
       implements Schedulable
{
  public RealtimeThread(SchedulingParameters s);
  public RealtimeThread(SchedulingParameters s,ReleaseParameters r);
  public RealtimeThread(SchedulingParameters s,ReleaseParameters rp,
        MemoryParameters m, ProcessingGroupParameters p,
        java.lang.Runnable r);

  // Schedulable 接口的方法
  public synchronized void addToFeasibility();
  public MemoryParameters getMemoryParameters();
  public ReleaseParameters getReleaseParameters();
  public Scheduler getScheduler();
  public SchedulingParameters getSchedulingParameters();
  public synchronized void removeFromFeasibility();
  public void setMemoryParameters(MemoryParameters p);
  public void setReleaseParameters(ReleaseParameters p);
  public void setScheduler(Scheduler s)
        throws IllegalThreadStateException;
  public void setSchedulingParameters(SchedulingParameters s);

  public static RealtimeThread currentRealtimeThread();

  public synchronized void schedulePeriodic();
  // 将周期线程加到可调度对象列表中
  public synchronized void deschedulePeriodic();
  // 从可调度对象列表中移走周期线程
  // 这只在线程发出了waitForNextPeriod () 时起作用
  public boolean waitForNextPeriod() throws IllegalThreadStateException;

  public MemoryArea getMemoryArea();
  public ProcessingGroupParameters getProcessingGroupParameters();
  public void setProcessingGroupParameters (ProcessingGroupParameters p);

  public synchronized void interrupt();
  // 覆盖 java.lang.Thread.interrupt()
  public static void sleep(Clock c, HighResolutionTime time)
        throws InterruptedException;
```

```
  public static void sleep(HighResolutionTime time)
         throws InterruptedException;
}
```

周期线程是那些用PeriodicParameters创建的实时线程。调用Start方法来首次启动这种线程。一旦线程已执行，它调用waitForNextPeriod告诉调度程序它应该在下一个周期到达的时候再次被置为可运行的。下面的程序片断说明了一个有5ms时限的周期为10ms的周期线程的结构，它的首次执行一直延迟到绝对时间A。这个线程应当消耗不超过1ms的处理器时间。

```
public class Periodic extends RealtimeThread
{
  public Periodic( PriorityParameters PP, PeriodicParameters P)
  { ... };

  public void run()
  {
    while(true) {
      // 每个周期要运行的代码
      ...
      waitForNextPeriod();
    }
  }
}

{
  AbsoluteTime A = new AbsoluteTime(...);
  PeriodicParameters P = new PeriodicParameters(
          A, new RelativeTime(10,0),
          new RelativeTime (1, 0), new RelativeTime (5,0),
          null, null);
  PriorityParameters PP = new PriorityParameters(...);

  Periodic ourThread = new Periodic (PP, P); // 创建线程

  ourThread.start (); // 启动它

}
```

442~444

非周期和偶发线程可以以相似的方式创建。但这些线程应该是等待一个启动事件而不是调用waitForNextPeriod。已经提到过，非周期和偶发线程的启动事件现在还没有明确定义。

在10.5节讨论了异步通知：介绍了事件处理的终止模型和恢复模型。指出了一个用于多线程进程中的异步通知处理的恢复模型等价于执行一个响应这个通知的非周期/偶发线程。这是实时Java采用的模型。然而，不是把处理程序看成一个显式的线程，而是看成一个支持Schedulable接口的对象。事实上，可能不止一个处理程序绑定到一个线程上，而且这种绑定是在事件发生时动态进行的。类BoundAsyncEventHandler能用来永久地将事件处理程序绑定到一个线程上（因此可以更快地响应这个事件）。假设一个异步事件的处理程序是一个可调度类，那么必须给它恰当的启动参数。由于事件处理程序有一个显式的启动，可以很容易地检测到它的时限超期。

12.7.4 DPS

尽管Pearl、实时Euclid和Java将时序作用域与进程（线程）相关联，并且因此使得进程必

须规定自己的时间约束，其他的语言——比如DPS（Lee and Gehlot，1985）——提供应用在代码块一级的局部时间性设施。

一般来说，一个DPS时序块（作用域）需要规定三个不同的时间需求（和前面讨论过的更多全局需求相似）:

- 延迟开始于一个已知的时间段
- 在已知的时限前完成执行
- 在一段不超过设定值的时间里进行运算

为了说明这些结构，设想冲制和饮用速溶咖啡的重要实时行为:

```
get_cup
put_coffee_in_cup
boil_water
445 put_water_in_cup
drink_coffee
replace_cup
```

冲制一杯咖啡应该不会超过10min，喝咖啡要复杂得多。3min的延迟应该能保证嘴巴不会被烫伤，咖啡应该在25min内喝完（之后就凉了）或者在17: 00（下午5点下班回家）之前喝完。这样需要两个时序作用域:

```
start elapse 10 do
  get_cup
  put_coffee_in_cup
  boil_water
  put_water_in_cup
end

start after 3 elapse 25 by 17:00 do
  drink_coffee
  replace_cup
end
```

由于时序作用域是重复执行的，所以一个时间循环结构是很有用的，即:

```
from <start> to <end> every <period>
```

例如，很多软件工程师要求工作日里定时供应咖啡:

```
from 9:00 to 16:15 every 45 do
  make_and_drink_coffee
```

这里`make_and_drink_coffee`可以由上面给出的两个时序作用域组成（去掉喝咖啡那个块里的“by”约束)。注意，如果这样做的话，这个循环每次迭代的最长持续时间会是35min，比循环的周期要短，因此在喝两杯咖啡之间要有间隙。

尽管可以用这种方法声明块级的时间约束，但它们导致进程在执行过程中会经历不同的时限，有时甚至会没有时限。将这些进程分解成子进程，每个子进程都是单独的块，这样可以将所有时限表示成进程级别的约束，从而运行时调度程序更易于实现。例如，在下一章里要讨论的一些算法中，一个静态的优先级方案已经足够了，调度程序不需要显式地意识到时限。

12.7.5 Esterel

Esterel（Boussinot and de Simone，1991）是同步语言的一个例子，其他的同步语言还包

括 Signal（le Guernic等，1991）和Lustre（Halbwachs等，1991）。这些语言试图通过对程序的时序行为做某些假设来支持验证。支撑这种计算模型的基本假设是*理想（或完美）的***同步假设**（Berry，1989）： 446

理想的系统中输出的产生是与它们的输入同步的。

因此假定所有的计算和通信都不花时间（也就是说所有的时序作用域瞬时地执行）。很明显这是一个很强的而且不现实的假定，但它可以使事件的时序次序更容易确定。在实现的过程中，*理想的同步假设*被解释成“系统必须执行得足够快，以保证这个同步假设成立”。在现实中，它意味着：在任何输入之后，与之对应的输出必须在新的输入发生之前出现。这时我们说这个系统“紧跟”它的环境。Esterel程序是事件驱动的并且使用信号通信（广播）。tick信号假定是定时的（尽管没有定义它的粒度）。这样，定义了下面的重复（每10个滴答）的周期性模块：

```
module periodic;
input tick;
output result(integer);
var V : integer in
  loop
    await 10 tick;
    -- 进行为设置V的所需计算
    emit result(v);
  end
end
```

偶发模块有同样的形式。

同步假设的一个推论是所有的动作都是原子的。由于动作是瞬时完成的，并发动作间的交互就不可能进行了。在上面的例子里，结果在等候的滴答那个瞬时立即发送出去（因此这个模块不会有局部漂移和累积漂移）。等待结果的偶发模块会在这个周期模块的“同一时间”执行。这种行为有效地减少了不确定性。但是它也导致了潜在的因果问题。考虑：

```
signal S in
  present S else emit S end
end
```

这个程序是前后不一致的。如果S未出现就发送它，相反如果它出现就不发送。

Esterel行为语义的形式化定义有助于消除这些问题（Boussinot and de Simone，1991）。可以检查程序的前后一致性。实现合法的Esterel程序是很简单的，用同步假设总是能够构造一个有限状态机。因此一个程序从一个初始状态（读可能有的输入）移动到一个结束状态（产生可能有的输出）。当它在这些状态间移动时，不会和环境发生交互。正如前面指出的，只要有限状态机（**自动机**）以足够的速度运行，原子性的假设就可以认为是成立的。 447

12.8 容错

本书从头至尾都假设实时系统有高可靠性的需求。实现这种可靠性的一种方法是将容错加入到软件当中去。时间约束的加入也引入了这些约束被破坏的可能性，比如超时到期或者没有满足时限。

对于软实时系统，进程可能需要知道是否错过了一个时限，尽管在正常运行下可以容忍这一点。更重要的是，在有些硬实时系统（或子系统）里，时限是很关键的，一个错过的时限需要触发某个出错恢复例行程序。如果系统显示出在最坏情况的执行时间下是可调度的，那么可以证明时限不会错过。但是，在前面章节里对可靠性的讨论明确指出需要一种多方面的可靠性方法，也就是要证明所有的事情都不会出错，并包括能够合适地处理出错时出现问题的例程。在这种特定的情况下，在一个“被证明了的”系统里可能会错过一个时限，如果：

- 最坏情况执行时间（WCET）的计算是不准确的
- 可调度性检查中的假设是无效的
- 可调度性检查本身有错误
- 调度算法无法处理，尽管在理论上是可调度的负载情况
- 系统运行超出了它的设计参数。

在最后一种情况（比如，一个以无法接受的中断速率表现出来的信息溢出），系统设计人员仍然会希望有软失效或故障保护行为。

总的来说，要对时间失效容错，系统必须能检测：

- 时限的超出
- 最坏情况执行时间（WCET）的超出
- 超出预期频度发生的偶发事件
- 通信的超时

当然，这个列表里最后三个“失效”不是一定表示会错过时限，例如，在一个进程中超
448 出WCET可以通过一个偶发事件以小于其最大容许频率的速度发生的方法补偿。因此提供容错的损害隔离和评估阶段必须决定采取什么动作。向前和向后的恢复都是可行的。下面的小节将讨论这里的前三个时间性故障，通信超时已经在12.4节讨论过了。

12.8.1 时间性错误检测和向前出错恢复

如果要处理时间性错误，首先要检测到它们。如果运行时环境或操作系统知道一个进程的突出特征（比如采用实时Java的方法），那么它将能检测到问题并使应用注意到这些问题。否则，就必须提供使应用能检测自己的时间性错误的原语设施。

1. 时限超出的检测

这一章已经与时序作用序域一起说明了语言和操作系统为周期与偶发进程的规定提供的设施。

Ada的运行时支持系统并不知道其应用任务的时间性需求，因此不得不提供原语机制来检测时限的超出。这是使用10.8节讨论的异步控制转移设施来实现的。因此，为了检测一个12.7.1节提到的周期任务的时限的超时，需要主循环嵌入一个select then abort语句。

```
task body Periodic_T is
  Next_Release : Time;
  Next_Deadline : Time;
  Release_Interval : constant Duration := ...; -- 或
  Release_Interval : constant Time_Span := Milliseconds(...);
  Deadline : constant Time_Span := Milliseconds(...);
begin
  -- 读时钟并计算下一个启动时间(Next_Release)
  -- 和下一个时限(Next_Deadline)
  loop
```

```
    select
      delay until Next_Deadline;
      -- 在这里检测时限超出
      -- 进行恢复
    then abort
      -- (例如)数据采集或计算和发送一个控制信号
    end select;
    delay until Next_Release;
    Next_Release := Next_Release + Release_Interval;
    Next_Deadline := Next_Release + Deadline;
  end loop;
end Periodic_T;
```

449

可以用相似的方法检测偶发任务中的时限的超出。

这种方法的一个问题是它假定恢复策略要求任务停下正在进行的工作。这当然是一种选择，但是还有其他方法，例如，允许任务以一个不同的优先级继续执行。为了做到这一点，对于检测时限超出更加合适的响应是引起一个异步的事件。在实时Java里，当一个周期线程在它的时限到期后还在执行的时候，虚拟机会用信号发出一个异步事件。实时Java中的偶发事件处理程序没有显式的时限超出检测，它们被假定为软实时的。

C/POSIX允许创建和设置定时器，当它们到期时会产生用户定义的信号(默认的是SIGALRM)。因此，可以使进程决定应该采取什么样的正确动作。程序12-12显示了一个典型的C接口。

程序12-12 C到POSIX定时器的接口

```
#define TIMER_ABSTIME ..

struct itimerspec {
  struct timespec it_value; /* 第一个定时器信号 */
  struct timespec it_interval; /* 后续间隔 */
};
typedef ... timer_t;

int timer_create(clockid_t clock_id, struct sigevent *evp,
                 timer_t *timerid);
  /* 使用规定的时钟为每个进程创建一个定时器，作为时间性基础。*/
  /* evp 指向一个结构，它包含生成信号所需的所有信息*/

int timer_delete (timer_t timerid);
  /* 删除一个进程定时器*/

int timer_settime(timer_t timerid, int flags,
                  const struct itimerspec *value,
                  struct itimerspec *ovalue);
  /* 为指定的定时器设置下一个到期时间*/
  /* 如果flags被设置为TIMER_ABSTIME, 则 */
  /* 当时钟到达*value.it_value 规定的绝对值时，定时器将到期 */
  /* 如果flags未设置为 TIMER_ABSTIME, 则*/
  /* 当由value->it_value规定的间隔过去时，定时器将到期 */
  /* 如果 *value.it_interval 不为0, */
  /* 则一个周期性定时器将在value->it_value到期后 */
  /* 抛弃每一个value->it_interval */
  /* 任何以前的定时器设置被返回到*ovalue中 */
```

```
int timer_gettime(timer_t timerid, struct itimerspec *value);
  /* 得到当前定时器的详细信息*/

int timer_getoverrun(timer_t timerid);
  /* 如果支持实时信号，返回由此定时器生成而未处理的信号个数*/

/* 除 timer_getoverrun外，所有上述函数如果成功，返回0，否则，返回-1. */
/* timer_getoverrun 返回超限的个数 */
/* 当上述函数中的任何一个返回出错状态时，共享变量errno 中包含有出错原因*/
```

在5.5.1节介绍了用于错误检测的定时器。它很容易用POSIX的信号编写。例如，研究这么一种情况，一个线程（monitor）创建了另一个线程（server）并希望监视它的进展看它是否满足了时限。struct timespec deadline给出server的时限。monitor线程为每个进程创建了一个定时器，当定时器到期的时候，它就指定执行一个信号处理程序。然后，monitor创建server线程，并传递一个指针给定时器。server线程执行自己的动作，然后删除这个定时器。如果发出了警报，server就迟到了。

```
#include <signal.h>
#include <timer.h>
#include <pthread.h>

timer_t timer; /*monitor和server之间共享的定时器*/

struct timespec deadline = ...;
struct timespec zero = ...;

struct itimerspec alarm_time, old_alarm;

struct sigevent s;

void server(timer_t *watchdog)
{
  /* 进行所需服务*/
  TIMER_DELETE(*watchdog);
}

void watchdog_handler (int signum, siginfo_t *data, void *extra)
{
  /* SIGALRM 处理程序 */

  /* server迟到: 进行恢复*/
}
void monitor()
{
  pthread_attr_t attributes;
  pthread_t serve;


  sigset_t mask, omask;
  struct sigaction sa, osa;
  int local_mode;

  SIGEMPTYSET(&mask);
  SIGADDSET(&mask, SIGALRM);

  sa.sa_flags = SA_SIGINFO;
```

450~451

```
    sa.sa_mask = mask;
    sa.sa_sigaction = &watchdog_handler;

    SIGACTION(SIGALRM. &sa, &osa); /* 分配处理程序 */


    alarm_time.it_value = deadline;
    alarm_time.it_interval = zero; /* 单冲定时器 */

    s.sigev_notify = SIGEV_SIGNAL;
    s.sigev_signo = SIGALRM;

    TIMER_GREATE(CLOCK_REALTIME, &s, &timer);

    TIMER_SETTIME (timer, TIMER_ABSTIME, &alarm_time, &old_alarm);

    PTHREAD_ATTR_INIT (&attributes);
    PTHREAD_CREATE (&serve, &attributes, (void *)server, &timer);
}
```

但是，如在10.6.6节中提到的，如果进程是多线程的，信号就是发送给整个进程而不是某个单独的线程。因此，一般来说确定哪个线程超过了它的时限是困难的。

实时Java同样支持定时器。程序12-13和12-14定义了相关的类。

程序12-13　实时Java的类Timer

```
public abstract class Timer extends AsyncEvent
{
  protected Timer(HighResolutionTimer time, Clock clock,
                  AsyncEventHandler handler);

  public ReleaseParameters createReleaseParameters();

  public AbsoluteTime getFireTime ();
  // 获取激发这个事件的时间

  public void reschedule(HighResolutionTimer time);
  // 改变这个事件的调度时间

  public Clock getClock();

  public void disable();
  // 禁用这个定时器，防止它激发。然而，一个被禁用的定时器在它被禁用的时候还继续计时

  public void enable();
  // 重新启用一个被禁用了的定时器
  // 如果该定时器的激发时间已过，它会立即激发

  public void start(); // 开始定时器滴答
}
```

程序12-14　实时Java的类OneShotTimer和PeriodicTimer

```
public class OneShotTimer extends Timer
{
  public OneShotTimer(HighResolutionTimer time,
                      AsyncEventHandler handler);
```

```
  // 创建AsyncEvent的一个实例，它将在给定的时间期满时执行它的激发方法

  public OneShotTimer(HighResolutionTimer start, Clock clock,
                      AsyncEventHandler handler);
  // 基于给定的时钟创建AsyncEvent的一个实例，
  // 它将在给定的时间期满时执行它的激发方法
}

public class PeriodicTimer extends Timer
{
  public PeriodicTimer(HighResolutionTimer start, RelativeTime interval,
                       AsyncEventHandler handler);
  //创建AsyncEvent的一个实例，它周期性地执行它的激发方法

  public PeriodicTimer (HighResolutionTimer start,
                        RelativeTime interval,
                        Clock clock, AsyncEventHandler handler);
  // 创建AsyncEvent的一个实例，
  // 它将基于一个特定时钟从一开始就周期性地执行它的激发方法

  public ReleaseParameters createReleaseParameters();

  public void setInterval(RelativeTime interval);
  // 重新设置该定时器的间隔
  public RelativeTime getInterval();

  public void fire();
  // 覆盖类AsyncEvent中的fire方法
  public AbsoluteTime getFireTime();
}
```

通过DPS的局部时间结构，将时间性错误和异常关联起来是很合适的。

```
start <时间约束> do
  -- 语句
exception
  -- 处理程序
end
```

除了损害隔离、出错恢复等所需的必要计算外，处理程序还希望延长时限的长度和继续执行原来的程序块。因此恢复模型比终止模型更合适（第6章）。

在依赖于时间的系统里，也可能必须给出处理程序的时限约束。通常处理程序的执行时间是占用时序作用域本身的，比如在下面，语句序列将会在19个时间单位后提前结束。

```
start elapse 22 do
  -- 语句
exception
  when elapse_error within 3 do
  -- 处理程序
end
```

对于所有的异常模型，如果处理程序本身引发一个异常，这种情况只能在这个程序层次中的更高级别上处理。如果在进程级的处理程序中发生了时间性错误，那么这个进程必须终止（或者至少终止该进程的当前循环）。因此应该由一些系统级的处理程序来处理失败的进程，

或者让应用软件去识别和应付这些事件。

如果将异常处理程序加入到前面给出的冲制咖啡的例子里，程序代码就会有下面的形式（"没有咖啡杯"之类的逻辑错误异常没有包括进去）。程序假定只有boil_water和drink_coffee有显著的时序性质，因此时间性错误是由于这些行为的时间超出造成的。

```
from 9:00 to 16:15 every 45 do
  start elapse 11 do
    get_cup
    boil_water
    put_coffee_in_cup
    put_water_in_cup
  exception
    when elapse_error within 1 do
      turn_off_kettle   -- 为安全起见
      report_fault
      get_new_cup
      put_orange_in_cup
      put_water_in_cup
    end
  end

  start after 3 elapse 26 do
    drink
  exception
    when elapse_error within 1 do
      empty_cup
    end
  end
  replace_cup
exception
  when any_exception do
    null    -- 继续下一次迭代
  end
end
```

452~454

2. 最坏情况执行时间的超出

好的软件工程实践尽力将错误的后果限定在程序明确定义的区域里。模块、包和原子动作等设施有助于实现这个目标。然而，如果一个进程消耗了超过预期的CPU资源，那么它可能不是错过了时限的那个进程。例如，它可能是一个有相当长空闲时间的优先级高的进程，而会错过时限的进程可能是有较少空闲时间的优先级低的进程。理想的情况下，应该可能就在导致时间性错误的进程里捕获这种错误。这意味着必须在一个进程超出了程序员所允许的最坏情况执行时间时能够检测到它。当然，如果一个进程是非抢占式调度的（不会因为等待资源而被阻塞），它的CPU执行时间就等于这个进程存在的时间，并且可以使用那些用于检测时限超出的机制。然而进程常常是抢占式调度的，因此导致测量占用的CPU时间变得更加困难。

POSIX通过它的时钟和定时器来支持执行时间监视。它定义了两个时钟：CLOCK_PROCESS_CPUTIME_ID和CLOCK_THREAD_CPUTIME_ID。这些时钟能以和CLOCK_REALTIME相同的方式使用。每个进程/线程都有一个与之关联的执行时间时钟，调用： 455

```
clock_settime(CLOCK_PROCESS_CPUTIME_ID, &some_timespec_value);

clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &some_timespec_value);

clock_getres(CLOCK_PROCESS_CPUTIME_ID, &some_timespec_value);
```

将设置/取得执行时间或者取得与调用进程关联的执行时间时钟的分辨率（对线程也是类似的）。

有两个函数使一个进程/线程能够得到另一个进程/线程的时钟并进而访问这个时钟。

```
int clock_getcpuclockid(pid_t pid, clockid_t *clock_id);

int pthread_getcpuclockid(pthread_t thread_id, clockid_t *clock_id);
```

程序12-12定义的定时器可以用来创建定时器，并进而可以用这些定时器在设定的执行时间期满时产生进程信号。如果和`timer_create`连同使用的`clock_id`与调用进程或线程的`clock_id`不同，那么程序的行为是由实现定义的。由于定时器期满时产生的信号是指向进程的，在一个线程的执行时间定时器期满时哪个线程会收到这个信号是依赖于应用的。应用可以不允许线程使用定时器（由于支持这种设施的开销）。

至于所有的执行时间监视，在上下文切换和中断存在的情况下很难保证执行时间时钟的准确性。

实时Java允许把“时间开销”值关联到一个可调度对象的执行上。如果实现支持的话，当超过这个“时间开销”值时这允许激发一个异步事件。

3. 偶发事件的超出

比预期激发得更频繁的偶发事件会给一个试图满足硬时限的系统带来不正常的后果。因此有必要保证：或者这是禁止的，或者当它发生时检测它并采取矫正动作。

本质上有两个禁止偶发事件超出的方法。第一个方法是用在这种事件由一个硬件中断触发的地方。在大多数的情况下，能够通过操纵相关联的设备控制寄存器抑制这种中断的发生。

另一个方法是利用偶发服务器技术，参见13.8.2节。如果没有使用偶发服务器并且中断不能禁止，那么需要检测什么时候偶发事件发生得过于频繁。可惜，大多数的实时语言和操作
456 系统缺乏对这种检测的支持。

12.8.2 时间性错误检测和向后出错恢复

作为向前出错恢复的替代方法，时间性错误能够使用向后出错恢复。所有的向后出错恢复技术都包含接受测试。因此在这些测试中可能包含时序性需求。如果时限被超出了，运行时系统能够异步地放弃接受测试。使用一个合并到对话和会谈（Gregory and Knight，1985）内的超时设施（在第10章讨论过的）说明了这一点。对话序列现在变成了：

```
SELECT
    dialog_1
OR
    dialog_2
OR
    dialog_3
TIMEOUT value
    -- 语句序列
ELSE
```

```
    --语句序列
  END SELECT;
```

和前面一样，进程在执行时首先尝试dialog_1。如果成功，控制转到跟在选择语句后面的语句。如果这个对话失败，就尝试dialog_2等等。但是为了使会谈概念能够处理错过的时限，可以把超时同这个选择关联起来。时间性约束规定了一个时间间隔，在此期间这个进程可以执行尽可能多的对话尝试。对话中的不同进程可能有不同的超时值。如果一个超时期满了，那么当前执行的对话就失败了，这个进程的状态恢复到执行选择语句时的状态，并且执行在TIMEOUT子句后的语句。这时，这个对话涉及的其他进程也就失败了，但它们的行为由它们的选择语句中设定的选项决定。它们可以尝试另一个对话、超时或一起失败。至于否则（else）子句，可以将TIMEOUT后的语句看作是实现这个进程目标的最后的尝试。如果它失败了那么其外包的会谈也就失败了。

现在就能够编写一个使用对话序列的简单时限机制，这个对话序列由一个对话尝试和一个超时组成。例如，设想一个不显式支持时限规定的类Ada的实时语言，当一个任务同许多任务通信时，可以以下面的方法从时限错误中恢复过来：

```
task body deadline_example is
begin
  loop
    ...
    time := calculated_time_to_deadline;
    slack := calculate_time_for_degraded_algorithm;
    restore := state_restoration_time;
    timeout value := time - (slack + restore);
    ...
    SELECT
      dialog_1;
    TIMEOUT timeout_value
      -- 从错过的时限恢复的语句序列
    ELSE
      fail;
    END SELECT;
  end loop;
end deadline_example;
```

457

这个任务首先计算到下一个时限的时间延迟。然后，它从中减掉估计用来从一个错过的时限中恢复的时间和用于状态恢复的时间。这些值之间的差叫做空闲时间。如果这个对话由于时间性错误或者设计错误而没有在这个时间里完成，那么就启动恢复序列。

12.8.3 模式改变和基于事件的重配置

在上面的讨论中，通常是假设一个错过的时限可以由实际负责该时限的进程或者线程处理。实际情况并不总是这样。一个时间性错误的后果经常会是：

- 其他的进程必须改变它们的时限，或者甚至终止它们正在干的事。
- 可能需要启动新的进程。
- 非常重要的计算可能需要比当前可用的更多的处理器时间，为了获得额外的时间，其他相对不重要的进程可能需要被“挂起”。

• 可能需要把进程“中断”，以便于采取下面的措施（典型的）之一:
  —— 立即返回目前已经得到的最好结果;
  —— 改用一个更快（可能会相对不够准确）的算法;
  —— 忘掉现在正在做什么，并且准备执行新的指令：“restart without reload”。

这些动作有时被称为**基于事件的重配置**。

某些系统可能会进入一些预料时限易于错过的情形。经历过**模式改变**的系统很好地说明了这一点。在这样的环境里发生一些事件，这些事件导致不再需要某些已经开始的计算。如果系统完成这些计算的话，那么可能会错过其他的时限，因此有必要提前终止那些包含这些计算的进程或时序作用域。

为了执行基于事件的重配置和模式改变，需要在相关的进程之间通信。由于这种通信的
458 异步特点，有必要使用出现在一些语言如Ada、Java和C/POSIX中的异步通知机制（见10.5节）。这些机制是低级的，可以证明确实需要这些机制，以便能通知调度程序停止调用某些当前不需要的进程，并开始调用其他的进程。实时Java朝着这个方向发展，它把一些方法同实时线程相关联，用以通知调度程序某线程当前不再需要。

实时Euclid在这个方面也很有趣，因为它将它的异步事件处理机制同它的实时抽象连结到一起。在实时Euclid里，时间约束和进程关联，并且可以定义编了号的异常。每个进程都必须提供处理程序。例如，考虑下面定义了三个异常的温度控制器进程：

```
process TempController : periodic frame 60 first activation
                          atTime 600 or atEvent startMonitoring
  % 导入列表
  handler (except_num)
    exceptions (200,201,304)  % 例如
    imports (var consul, ...)
    var message : string(80), ...
    case except_num of
      label 200:  % 非常低的温度
        message := "reactor is shut down"
        consul := message
      label 201:  % 非常高的温度
        message := "meltdown has begun - evacuate"
        consul := message
        alarm := true   % 激活警报设备
      label 304:   % 传感器上的超时
                   % 重新引导传感器设备
    end case
  end handler
  %
  % 执行部分
  %
end TempController
```

实时Euclid允许一个进程引发另一个进程的异常。它支持三种不同的引发语句：*except*、*deactivate*和*kill*，从字面上就反映了它们的严厉程度一个比一个高。

*except*语句基本上和Ada的raise（引发）一样，不同的是一旦处理程序已经执行，控制就返回到它离开的地方（也就是恢复模型）。相比之下，*deactivate*语句导致（周期）进程的迭

代终止。被终止的进程仍然执行异常处理程序，但只有在下一个周期到来时才会恢复活动。要终止一个进程，可以用*kill*语句，它显式地从一批活动的进程里删除一个进程（可以是自己）。它和无条件中止不同的是在终止之前会执行异常处理程序。它的优点是进程可以完成一些重要的“临终仪式”，缺点是在处理程序里的错误还是可以造成这个进程的误动作。

为了说明这些异常的使用，可以给前面给出的温度控制进程的执行部分添加一些细节。注意在这个例子里，在同一个进程里异常是同步引发和处理的，而在另一个进程里则是异步进行的。首先，这个进程等待一个条件变量，规定了一个超时并且给出了一个异常编号（如果超时，则通过*except*引发这个编号的异常）。然后读出并记录温度。对温度值的检测会导致其他异常被引发。一个过低的值会产生一个恰当的消息并且直到下个周期之前被停止使用；一个过高的值会产生一个更加恰当的——如果有点繁琐的话——消息，报警进程中引发一个异常并且让这个温度控制器终止。所有可用的处理器时间现在可以都给这个报警进程。 459

```
process TempController : periodic frame 60 first activation
                         atTime 600 or atEvent startMonitoring
  % 导入列表
  handler (except_num)
    exceptions (200,201,304) % 例如
    imports (var consul, ...)
    var message : string(80), ...
    case except_num of
      ... % 同前
    end case
  end handler

  wait(temperature_available) noLongerThan 10 : 304
  currentTemperature := ... % 低级 i/o
  log := currentTemperature
  if currentTemperature < 100 then
    deactivate TempController : 200
  elseif currentTemperature > 10000 then
    kill TempController : 201
  end if
  % 其他计算
end TempController
```

为了在Ada里完成这种形式的重配置，有两个机制可以使用：

- 中止——类似于*kill*
- ATC——类似于*deactivate*

Ada允许用受控变量编写“临终仪式”（见第4章）。正如在10.8节指出的，ATC 特征是一个普遍特征，因此它可以处理*deactivate*和其他大多数形式的基于事件的重配置。参见文献Real and Wellings（1999a）以及Real and Wellings（1999b）的关于Ada中模式改变的讨论。

小结

对时间的管理有很多的困难，这些困难使得嵌入式系统和其他计算机应用区别开来。当前的实时语言通常不能充分支持这个重要的领域。 460

为实时语言引入时间的概念用四个需求描述：

- 访问时钟
- 延迟
- 超时
- 时限规定和调度

不同语言测量一段时间的手段是不同的。occam2支持仅仅返回一个含义由实现定义的整数的TIMER设施。Ada和实时Java进了一步，为时间提供了两个抽象数据类型和一个同时间有关的运算符集合。C/POSIX为时钟和定时器（包含周期定时器）提供了一组全面的设施。

如果一个进程要暂停一段时间，需要延迟（或休眠）原语防止这个进程忙等待。这样的原语通常保证挂起这个进程至少一段指定的时间，但它不能促使调度程序在延迟到期后立即运行这个进程。所以，虽然可以限制由反复执行延迟引起的累积漂移，但却无法避免局部漂移。

对于许多实时系统，逻辑正确的软件还不够，程序必须满足时间性约束。令人遗憾的是，大型实时系统的现有工程实践还只是针对特定目的的。为了便于规定时间性约束和需求，引入一个“时序作用域”的概念是很有帮助的。时序作用域可能的属性有:

- 执行完成的时限
- 在执行开始之前的最小延迟
- 在执行开始之前的最大延迟
- 最长执行时间
- 最长存在时间

这一章考虑了在编程语言里如何规定时序作用域的问题。

时间性需求的重要程度是刻画实时系统的一个有用方法。必须满足的约束被称为硬的，那些可以偶尔错过的或少量错过的约束称为固或软的。

要做到时间性出错的容错，必须能够检测:

- 时限的超出
- 最坏情况执行时间的超出
- 461 发生得比预期要频繁的偶发事件
- 通信的超时

检测之后，可能需要进行基于事件的重配置。

相关阅读材料

Buttazzo, G. C. (1997) *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. New York: Kluwer Academic.

Galton, A. (ed.) (1987) *Temporal Logics and their Application*. London: Academic Press.

Halang, W. A. and Stoyenko, A. D. (1991) *Constructing Predictable Real-Time Systems*. New York: Kluwer.

Joseph, M. (ed.) (1996) *Real-Time Systems: Specification, Verification and Analysis*. Englewood Cliffs, NJ: Prentice Hall.

Koptez, H. (1997) *Real-Time Systems*. New York: Kluwer Academic.

Turski, W. M. (1988) Time Considered Irrelevant For Real-time Systems. *BIT*, 28(3), 473–488.

练习

12.1 解释如何变换一个系统，使其所有时间性失效都以值失效表明自己。可以实现反向变换吗？

12.2 Ada的限时入口调用应当对会合的完成（而不是会合的开始）规定超时吗？给出一个例子说明这种方法何时有用。如果不这么做，怎样得到相同的效果？

12.3 在一个将异常用于向前出错恢复的实时类Ada语言中，建议了两个预定义的新异常（可能由运行时系统引发）：

DEADLINE_ERROR——当一个代码块不能在所需的时间间隔（相对于块的开始）内完成的时候引发；

WCET_ERROR——当一个块使用了多于规定的CPU时间的时候引发（即超过了最坏情况执行时间）。

讨论：

(1) 怎样将这些异常集成到一个类Ada语言中，特别是如何规定所需的时限和WCET值；

(2) 怎样实现这些异常（在运行时系统中）；

(3) 在应用中如何使用它们。

12.4 考虑下面一个简单（粗略的）延迟机制的管程接口：

```
monitor TIME is
  procedure TICK;
  procedure DELAY (D :NATURAL);
end TIME;
```

DELAY的调用者希望挂起D个滴答。过程TICK由某个时钟例程调用。每当它被调用时，阻塞在DELAY的每个进程就减少某个计数器（它被初始化为D），当计数器到0时，就退出，否则被再次阻塞。

说明如何实现这个管程的体。使用条件变量（在这些变量上定义有等待和发信号操作）。指出为信号所假设的语义。

12.5 讨论Ada中的包Time如何提供和练习12.4中同样的延迟机制（不要使用Ada的延迟语句）。

12.6 能够检测到实时Java中周期线程的时限超出吗？

12.7 时限在实时Java事件处理器规格说明中的作用是什么？

12.8 基于事件的重配置可以在实时Java中执行到什么程度？ 462~463

第13章 调　度

在一个并发程序中，不一定要规定进程执行的精确顺序。同步原语被用以施加局部顺序约束，例如互斥，但程序的整体行为展现了重大的不确定性。如果程序是正确的，那么，不管内部行为或实现细节如何，程序的功能性输出将是一样的。例如，在一个单处理器上有5个独立的进程，它们按非抢占的方式执行，则有120种不同的执行顺序。在多处理器系统上或在可抢占系统上，则有无穷多种的交叉执行。

虽然程序的输出在所有可能的交叉执行情况下都是相同的，时间性行为却有相当大的变化。如果上面五个进程中有一个有很紧迫的时限，那么也许只有这一进程首先执行才能满足程序的时间性需求。一个实时系统需要限制并发系统中的不确定性。这个过程称为调度。通常，调度方案提供两个功能: 465

- 一个安排系统资源（特别是CPU）使用顺序的算法
- 一个在应用调度算法时预测最坏情况系统行为的手段

这样，就可以用预测结果来证实系统的时间性需求得到满足。

调度方案可能是静态的（如果预测是在执行前进行的）或是动态的（使用运行时决策）。本章主要讨论静态方案，主要关注基于优先级的抢占方案。这里，进程被分配了优先级，任何时刻都是执行最高优先级的进程（如果它没有被延时或挂起）。所以一个调度方案包括优先级分配算法和可调度性测试。

13.1 简单进程模型

不可能容易地分析一个极为复杂的并发程序，以预测它的最坏情况的行为。所以需要对实时并发程序的结构给出一些限制。本节将介绍一个非常简单的模型以描述一些标准的调度方案。这一模型将在本章的后续部分进一步泛化（并在14章和16章进行进一步考察）。基本模型有下列特征:

- 假设应用是由固定数目的进程组成的。

- 所有进程是周期性的，并具有已知的周期。
- 各个进程彼此完全独立。
- 所有系统开销、上下文切换时间等被忽略（即假设开销为0）。
- 所有进程的时限等于它们的周期（即每个进程必须在它下一次启动前完成）。
- 所有进程有固定的最坏情况执行时间。

进程独立性的一个必然推论是可以假设所有的进程能在同一时间同时启动。这表示了处理器的最大负载能力，并被称为**临界瞬间**（critical instant）。

表13-1给出了进程特征的标准记号集。

表13-1 标准记号集

记号	描述
B	进程的最坏情况阻塞时间（如果适用）
C	进程的最坏情况计算时间（WCET）
D	进程的时限
I	进程的干扰时间
J	进程的启动抖动
N	系统中进程的个数
P	分配给进程的优先级（如果适用）
R	进程的最坏情况响应时间
T	进程启动间的最小时间（进程周期）
U	每个进程的利用率（等于*C/T*）
a – *z*	进程的名字

13.2 循环执行方法

对于一组数目固定的纯周期性的进程，可以安排一个完整的调度方案，该调度方案的重
466 复执行将使所有进程以正确的速率运行。从本质上说，循环执行是一个过程调用表，每个过程代表一个“进程”的部分代码。完整的调用表被称为**大循环**（major cycle），它通常由几个**小循环**（minor cycle）组成，每个小循环有固定的持续时间。例如，4个小循环每个持续时间为25ms，那么大循环有100ms的持续时间。执行期间，时钟每25ms中断一次，它使调度程序依次调度这4个小循环（每次调度一个）。表13-2给出了一组进程，它们必须用一个简单的4槽大循环来实现。循环执行的可能映射如图13-1所示，该图说明了在任一时刻处理器正在执行的工作。这样一个系统的代码可用如下的简单形式表示：

```
loop
  wait_for_interrupt;
  procedure_for_a;
  procedure_for_b;
  procedure_for_c;
  wait_for_interrupt;
  procedure_for_a;
  procedure_for_b;
  procedure_for_d;
  procedure_for_e;
  wait_for_interrupt;
```

```
    procedure_for_a;
    procedure_for_b;
    procedure_for_c;
    wait_for_interrupt;
    procedure_for_a;
    procedure_for_b;
    procedure_for_d;
  end loop;
```

表13-2　循环执行进程集合

进　程	周期，T	计算时间，C
a	25	10
b	25	8
c	50	5
d	50	4
e	100	2

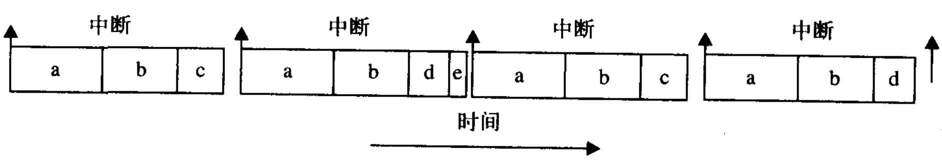


图13-1　进程集合的时间线

就是这么一个简单的例子也说明了这种方法的一些重要特征:

- 在运行时没有实际进程存在，每个小循环仅仅是一个过程调用序列。
- 这些过程共享一个公共地址空间，它们之间可以相互传递数据。这些数据不需要加以保护（例如，利用信号量），因为没有并发访问。
- 所有“进程”的周期必须是小循环时间的倍数。

最后一个性质是循环执行方法的主要缺点之一，其他缺点包括（Locke，1992）:

- 加入偶发进程的困难;
- 加入长周期进程的困难，在没有二级调度时，大循环时间是进程能采用的最大周期（即二级调度是指在大循环中的一个过程，每N个大循环调用一次二级过程）;
- 难以实际构造这种循环执行;
- 任何一个有相当长计算时间的“进程”将需要被划分为固定数目的固定长度的过程（这种划分从软件工程的角度看可能破坏代码结构，所以是易出错的）。

如果可以构造一个循环执行，就不需要进一步的可调度性测试（在构造过程中就已经证明了）。但是，对于高利用率的系统，这种循环执行方式的构造是有问题的。这个问题与经典的装箱问题（bin packing problem）类似。在装箱问题中，不同大小（一维）的物品必须放在最小数目的箱子里，且没有箱子是过满的。装箱问题被公认为NP难度问题，因此对于相当大的问题（一个典型的实际系统可能包含40个小循环和400个入口），它是不可计算的。所以必须采用启发式次优方案。 467~468

虽然对于简单的周期系统来说，循环执行仍然是一个合适的实现策略，但是，更灵活和适应性更强的方法是基于进程的调度方案。因此本章的其余部分将关注这些方法。

13.3 基于进程的调度

在循环执行方法中，运行时只有一系列的过程调用被执行。在执行期间没有进程（线程）的概念。另一个可选的方法是直接支持进程执行（这在通用操作系统中是一个标准），并通过使用一个或多个调度属性以决定系统在任一时刻执行哪个进程。使用这种方法，一个进程将处于下列状态之一（假设没有进程间通信）:

- 可运行的
- 挂起等待时间性事件——适用于周期进程
- 挂起等待非时间性事件——适用于偶发进程

13.3.1 调度方法

一般说来，有许多不同的调度方法。本书将考虑三种:

- 固定优先级调度（Fixed-Priority Scheduling，FPS）——这是一种最广泛使用的方法，也是本章要重点介绍的。每个进程有一个固定的、**静态**的优先级，优先级是在运行前计算的。可运行的进程是按它们的优先级决定的顺序执行的。*在实时系统中，进程的“优先级”是源自它的时序需求，而不是源自系统正确功能的重要性或完整性。*
- 最早时限优先（Earliest Deadline First，EDF）调度。这里根据进程的绝对时限决定可运行进程的执行顺序，下一个将运行的进程是有最短（最近）时限的进程。虽然通常只知道每个进程的相对时限（例如启动后25ms），但绝对时限是在运行时计算的，因此这种方案被认为是**动态**的。
- 基于值的（Value-Based Scheduling，VBS）调度。如果一个系统可能变得超载，那
469 么简单的静态优先级或时限是不够的，因此需要一个更有**适应性**的方案。通常采用的形式是给每个进程赋一个**值**，使用联机的基于值的调度算法去决定下一个运行哪一个进程。

如前面曾指出的，本章的主要内容是关于FPS，因为它被各种实时语言和操作系统标准所支持。EDF的使用也是很重要的，下面的讨论给出了一些对它的分析基础。本章末尾的13.13节给出使用VBS的一个简短描述。

13.3.2 抢占和非抢占

在基于优先级的调度中，一个高优先级的进程可能在一个低优先级进程的执行过程中被启动。在**抢占**（preemptive）方案中，将立即切换到高优先级进程。相反，在**非抢占**（non-preemption）方案中，在执行其他进程之前，允许低优先级进程完成运行。通常，抢占方案使高优先级进程更有活性，因为它们是首选的。在抢占和非抢占两个极端之间，有一个可供选择的策略，那就是允许较低优先级的进程继续运行一段限定的时间（但不一定完成）。这种方案被称为**延期抢占**（deferred preemption）**或合作分派**（cooperative dispatching）。在13.12.1节将再次考虑这个问题。在这一节之前，分派将被假定为可抢占的。EDF和VBS等方案也能采用抢占或非抢占方式。

13.3.3 FPS和速率单调优先级分配

对于13.1节描述的简单模型，存在一个简单的最优优先级分配方案，该方案被称为**速率单调**（rate monotonic）优先级分配。每个进程根据它的周期被赋一个（惟一的）优先级：周期

越短，优先级越高（也就是说，对两个进程i和j，$T_i<T_j \Rightarrow P_i>P_j$）。速率单调优先级分配在下面的意义下是最优的：如果任一进程集合能用固定优先级分配方案进行调度（使用可抢占的基于优先级的调度），那么，给定的进程集合也能用速率单调分配方案进行调度。表13-3列出了一个有5个进程的集合，并显示了进程优先级同进程周期的关系。需要注意的是，优先级用整数表示，较大的数表示较高的优先级。在阅读基于优先级调度的其他书籍和论文时需要特别小心，因为优先级可能按其他方式来排序，也就是，1代表最高优先级。而在本书中，1代表最低优先级，因为这是大多数编程语言和操作系统的常规用法。 470

表13-3 优先级分配的例子

进 程	周期，T	优先级，P
a	25	5
b	60	3
c	42	4
d	105	1
e	75	2

13.4 基于利用率的可调度性测试

本节描述一个非常简单的关于FPS的可调度性测试，虽然它不够精确，但是其简明性使它很有吸引力。

Liu和Layland（1973）指出，如果仅考虑进程集合的利用率，可以获得一个可调度性测试（当使用速率单调优先级排序时）。如果下列条件为真，那么所有N个进程将满足它们的时限（注意要对所有进程的利用率求和）：

$$\sum_{i=1}^{N}\left(\frac{C_i}{T_i}\right) \leqslant N(2^{1/N}-1) \qquad (13\text{-}1)$$

表13-4显示了N较小时利用率的界限（百分比）。当N很大时，界限接近69.3%。因此，当使用可抢占的基于优先级的调度方案进行调度、且优先级是用速率单调算法进行分配时，任何利用率之和小于69.3%的进程集合总是可调度的。

表13-4 利用率界限

N	利用率界限
1	100.0%
2	82.8%
3	78.0%
4	75.7%
5	74.3%
10	71.8%

下面给出三个简单的例子来说明这种测试的使用。在这些例子中，没有定义时间单位（绝对时间量）。只要所有的值（T、C等）是同样的单位，这种测试就适用。因此，在这些例子（以及后面的例子）中，时间单位被看作是某个假定时间基的滴答（tick）数。 471

表13-5 进程集合A

进程	周期，T	计算时间，C	优先级，P	利用率，U
a	50	12	1	0.24
b	40	10	2	0.25
c	30	10	3	0.33

表13-5包含3个进程，并且使用速率单调算法给它们分配了优先级（进程c有最高优先级，a有最低优先级）。它们的组合利用率为0.82（或82%）。这个利用率在3个进程的利用率门限值（0.78）之上，因此，这个进程集合没有通过利用率测试。

这个进程集合的实际行为可以通过画时间线（time-line）来说明。图13-2显示了3个进程的执行过程，它们都从时间0开始执行。注意，在时间50时，进程a仅执行了10个滴答，而它需要执行12个滴答，因此，它错过了它的第一个时限。

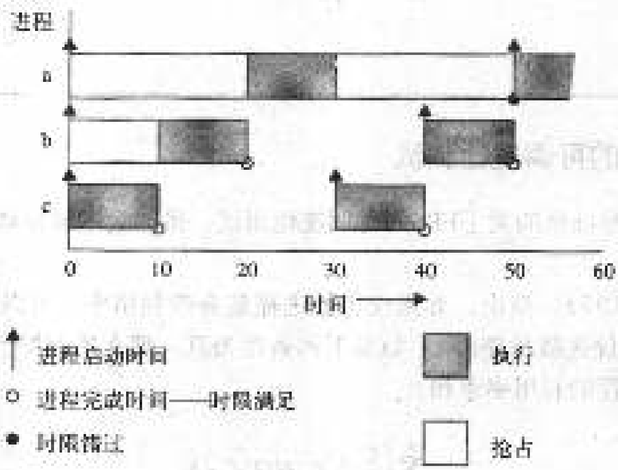


图13-2 进程集合A的时间线

时间线是说明执行模式的有用方法，图13-2也可用Gantt图（Gantt chart）来表示，如图13-3所示。

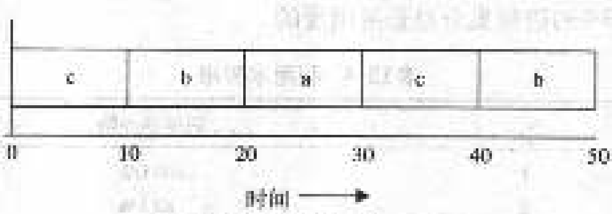


图13-3 进程集合A的Gantt图

第二个例子见表13-6。这3个进程的组合利用率为0.775，低于界限，因此这个进程集合肯定能满足所有的时限。如果画出这个进程集合的时间线，从图上可以看出所有的时限都满足。

虽然较麻烦，时间线确实能用于可调度性测试。但是，时间线应画多长才能作出结论呢？如果进程集合的启动时间相同（即它们共享一个*临界瞬间*），时间线的长度等于最长的进程周期就足够了（Liu and Layland，1973）。所以，如果所有进程满足它们的第一个时限，那么它们将满足所有将来的时限。

表13-6　进程集合B

进　程	周期，T	计算时间，C	优先级，P	利用率，U
a	80	32	1	0.400
b	40	5	2	0.125
c	16	4	3	0.250

最后一个例子见表13-7。这仍然是一个3个进程的系统，但是组合利用率为100%，显然，它不能通过测试。但是在运行时，它们的行为似乎是正确的，所有的时限在时间80时都能得到满足（见图13-4）。进程集合没有通过测试，但是在运行时没有错过时限，因此，测试是充分条件而不是必要条件。如果一个进程集合通过测试，它将满足所有时限；如果它没有通过测试，则在运行时可能失败也可能不失败。最后要指出，基于利用率的测试仅提供了一个简单的"是"或"不是"的回答。它并没有对进程的实际响应时间给出任何指示。进程的响应时间将在13.5节中讨论。

表13-7　进程集合C

进　程	周期，T	计算时间，C	优先级，P	利用率，U
a	80	40	1	0.50
b	40	10	2	0.25
c	20	5	3	0.25

472
473

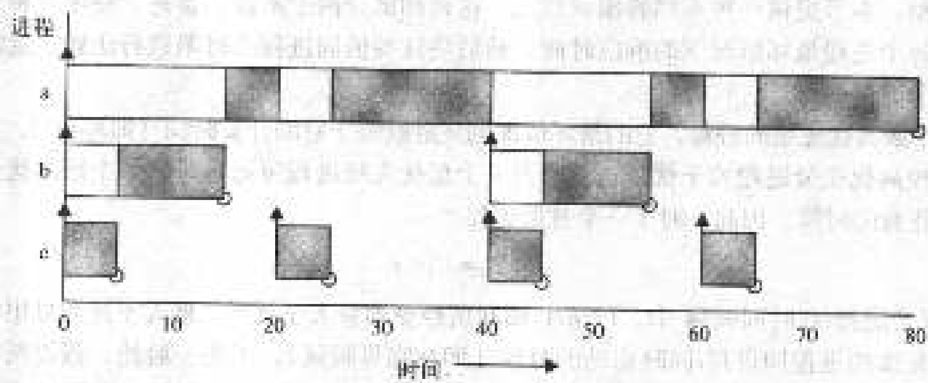


图13-4　进程集合C的时间线

EDF的基于利用率的可调度性测试

Liu和Layland（1973）的论文不仅介绍了针对固定优先级调度的基于利用率的测试，而且也为EDF给出了一个基于利用率的测试：

$$\sum_{i=1}^{N}\left(\frac{C_i}{T_i}\right) \le 1 \qquad (13\text{-}2)$$

很明显，这是一种简单得多的测试。只要进程集合的利用率小于处理器的总容量，那么所有时限将能被满足（对于简单进程模型）。在这个意义下，EDF优于FPS，它总是能调度FPS能调度的任何进程集合，但是并非所有通过EDF测试的进程集合都能使用固定优先级方法

来调度。既然有这个优势，那为什么EDF不是基于进程调度的首选方法呢？理由是FPS有一些超过EDF的优势：.

- FPS更容易实现，因为调度属性（优先级）是静态的；EDF是动态的，因此需要更复杂的运行时系统，开销也更高。
- 将没有时限的进程加入到FPS比较容易（只给它们分配优先级就可以了），而给进程一个任意的时限是更人为的做法。
- 时限属性不是惟一重要的参数，将其他因素加入优先级概念又比将这些因素加入时限容易些。
- 在超载（也许是故障状态）的情况下，FPS的行为更可预测（较低优先级的进程将首先
474 错过时限）；EDF在超载时是不可预测的，可能产生多米诺效应，导致大量进程错过时限。在13.13节将进一步讨论这一点。
- 对于简单模型，基于利用率的测试是一种误解，因为它对于EDF是充分必要条件，而对FPS它只是充分条件，因此，FPS通常能达到更高的利用率。

即使有最后这一点，EDF确实有一个超过FPS的优点，那就是它有更高的利用率。因此，在一些实验系统中，EDF持续地被研究和使用。

13.5 FPS的响应时间分析

FPS的基于利用率的测试有两个明显缺点：它们不准确，而且它们不真正适合于更一般的进程模型。本节提供一种不同的测试形式。这种测试分两个阶段。首先，使用一种分析方法去预测每个进程最坏情况下的响应时间。然后将这些值同进程的时限进行比较。这需要单独分析每个进程。

对于最高优先级的进程，它的最坏情况响应时间等于它的计算时间（即$R=C$）。其他进程将受到较高优先级进程的**干扰**，这就是当一个低优先级进程可运行时，一个较高优先级进程执行所花费的时间。因此，对于一个普通进程i:

$$R_i = C_i + I_i \tag{13-3}$$

其中，I_i是进程i在时间间隔［t，$t+R_i$）期间所经受的最大干扰[⊖]。最大干扰明显出现在所有更高优先级的进程同进程i同时启动的时候（即在临界瞬间）。不失一般性，假设所有进程在时间0时启动。假设进程j的优先级高于进程i，那么在间隔［0，R_i）期间，进程j将会启动多次（至少一次）。启动次数可以使用一个高限函数（ceiling function）来简单地表达：

$$\text{启动次数} = \left\lceil \frac{R_i}{T_j} \right\rceil$$

高限函数（$\lceil\ \rceil$）给出大于分式的最小整数。比如，1/3的高限是1，6/5的高限是2，6/3的高限
475 是2。不需要考虑负数的高限。

因此，如果R_i是15，T_j是6，那么进程j有3次启动（在时间0、6、12）。进程j的每次启动将施加一个C_j长度的干扰。所以，

⊖ 注意，因为这种分析中使用离散时间模型，所有的时间间隔必须在开始是闭区间（用‘[’表示），在结束是开区间（用‘)’表示），因此一个进程能在一个更高优先级进程启动的同一时刻完成运行。

$$\text{最大干扰} = \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

如果$C_j = 2$，那么在间隔 [0，15）之间有6个时间单位的干扰。每个更高优先级的进程都对进程i产生干扰，所以:

$$I_i = \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

这里$hp(i)$是比进程i优先级高的进程的集合。将该式代入等式（13-3）得到（Joseph and Pandya，1986):

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{13-4}$$

虽然干扰公式是准确的，但是实际的干扰数是未知的，因为R_i是未知的（它是正在计算的值)。在方程（13-4）的两边都有R_i，但因高限函数，使方程难以求解。这实际上是一个不动点方程的例子。一般来说，R_i可取许多值来满足方程（13-4)。最小的R_i值代表进程在最坏情况下的响应时间。求解方程（13-4）最简单的方法是构造一个递推关系（Audsley等，1993a):

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j \tag{13-5}$$

显然，$\{w_i^0, w_i^1, w_i^2, \cdots, w_i^n, \cdots\}$ 值集是单调非降的。当$w_i^n = w_i^{n+1}$时，方程的解就找到了。如果$w_i^0 < R_i$，那么w_i^n是最小解，也正是所需要的值。如果方程没有解，那么w值将不断上升(如果所有进程的总利用率超过100%，这种情况将发生在低优先级进程上)。一旦w值超过了进程的周期T，就可以确定进程将不能满足它的时限。基于以上分析可以得出下面的计算响应时间的算法:

```
for i in 1..N loop -- 依次对每个进程
  n := 0
  w_i^n := C_i
  loop
    按式(13.5)计算新的w_i^{n+1}
    if w_i^{n+1} = w_i^n then
      R_i := w_i^n
      exit {值找到}
    end if
    if w_i^{n+1} > T_i then
      exit {值未找到}
    end if
    n : = n + 1
  end loop
end loop
```

476

由定义的隐含含义，如果求出了响应时间，它将小于T_i，也因此小于进程的时限D_i（请记住在简单进程模型中，$D_i = T_i$)。

在以上讨论中，w_i仅仅作为一个数学实体用于求解一个不动点方程。但是，可以从问题领域给w_i一个直觉上的解释。考虑进程i的启动点，从这一点开始，直到进程结束，处理器将执行优先级等于或大于P_i的进程。处理器被认为是正在执行一个P_i - **忙碌期**（busy period）。将w_i看作一个时间窗口，它将滑过这个忙碌期。在时间0（进程i的假定启动时间），假设所有更高优先级的进程都已启动，因此

$$w_i^1 = C_i + \sum_{j \in hp(i)} C_j$$

这将是忙碌期的结束点，除非有更高优先级进程再次启动。如果是这样，那么时间窗口就需要向后移动。这一过程随着时间窗口的扩大持续下去，因此更多的计算时间落入了这个时间窗口。如果这一过程无限持续下去，那么忙碌期是无限的（也就是没有解）。但是，如果在任一时刻，一个正在扩大的时间窗口没有遇到更高优先级进程的再次启动，那么，忙碌期就已完成，并且忙碌期的长度就是进程的响应时间。

为了说明如何使用响应时间分析，考虑在表13-8中给出的进程集合D。

表13-8　进程集合D

进　程	周期，T	计算时间，C	优先级，P
a	7	3	3
b	12	3	2
c	20	5	1

最高优先级进程a的响应时间等于它的计算时间（例如，$R_a = 3$）。下一个进程的响应时间
477 需要计算。令w_b^0等于进程a的计算时间，也就是3。用式（13-5）来推导w的下一个值：

$$w_b^1 = 3 + \left\lceil \frac{3}{7} \right\rceil 3$$

即$w_b^1 = 6$。这个值满足等式$w_b^2 = w_b^1 = 6$，进程b的响应时间就得到了（$R_b = 6$）。

对最后一个进程的计算如下：

$$w_c^0 = 5$$

$$w_c^1 = 5 + \left\lceil \frac{5}{7} \right\rceil 3 + \left\lceil \frac{5}{12} \right\rceil 3 = 11$$

$$w_c^2 = 5 + \left\lceil \frac{11}{7} \right\rceil 3 + \left\lceil \frac{11}{12} \right\rceil 3 = 14$$

$$w_c^3 = 5 + \left\lceil \frac{14}{7} \right\rceil 3 + \left\lceil \frac{14}{12} \right\rceil 3 = 17$$

$$w_c^4 = 5 + \left\lceil \frac{17}{7} \right\rceil 3 + \left\lceil \frac{17}{12} \right\rceil 3 = 20$$

$$w_c^5 = 5 + \left\lceil \frac{20}{7} \right\rceil 3 + \left\lceil \frac{20}{12} \right\rceil 3 = 20$$

因此，R_c有最长的响应时间20，这意味着它刚好满足它的时限。图13-5用Gantt图说明了这种行为。

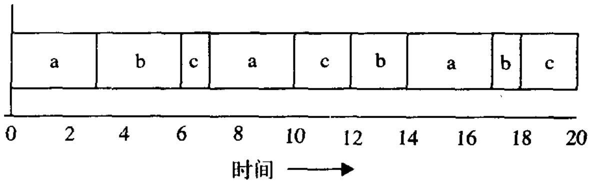


图13-5　进程集合D的Gannt图

再次考虑进程集合C。这个集合没有通过基于利用率的测试，但是在时间80时满足了所有进程的时限要求。表13-9显示了用以上方法计算的这个集合的响应时间。需要注意的是所有进程都能在它们的时限前完成。

表13-9　进程集合C的响应时间

进　程	周期，T	计算时间，C	优先级，P	响应时间，R
a	80	40	1	80
b	40	10	2	15
c	20	5	3	5

响应时间计算的优点是它们是充分必要条件——如果进程集合通过了测试，它们就满足所有的时限；如果没有通过测试，那么在运行时，一个进程将错过它的时限（除非估计的计算时间过于悲观）。因为这种测试优于基于利用率的测试，本章将重点关注扩大响应时间方法的适应性。

13.6　EDF的响应时间分析

EDF方案的一个缺点是当所有进程在临界瞬间启动时，无法得出每个进程的最坏情况响应时间。在那种情况下，只有那些有较短相对时限的进程进行干扰。但是，以后可能存在一种场合，在这里所有进程（至少是更多）有较短的绝对时限。例如，考虑在表13-10中描述的一个3进程系统。进程b的行为就说明了这个问题。在时间0，也就是临界瞬间，b仅仅受到进程a的干扰（一次），它的响应时间为4。但是在进程b的下一次启动时（在时间12），进程c仍然在执行，且c有较短的时限（16对24），因此进程c优先执行；进程b第二次启动的响应时间为8，两倍于在临界瞬间得到的值。后来的启动也许得到更大的值，当然这个值最大被限定在12，因为这个系统使用EDF是可调度的（利用率是1）。因此，寻找最坏情况是非常复杂的。这需要考虑所有进程的启动，以查看哪一个进程受到了其他有较短时限的进程的最大干扰。

表13-10　EDF的进程集合

进　程	T (=D)	C
a	4	1
b	12	3
c	16	8

在所有进程都是周期进程的简单模型中，整个进程集合将以每个**超周期**（hyper-period）重复执行，超周期就是所有进程周期的最小公倍数（LCM）。例如，在一个有4个进程的小系统中，4个进程的周期分别是：24、50、73和101个时间单位，这时LCM是4 423 800。求出 478~479

EDF的最坏情况响应时间需要考虑在4 423 800个时间单位内的每一次启动——而FPS只需要分析第一次启动（即需要考虑的最长时间是101个时间单位）。

虽然要考虑更多的启动，但是也可以推导出一个公式用于计算每一个进程的响应时间，计算方法同上面关于FPS的计算方法类似。这里我们将不给出这个推导，有兴趣的读者可以参看本章后面所列的相关阅读材料，这些书中有关于EDF的描述，也包括其他与EDF调度相关的内容。

13.7 最坏情况执行时间

到目前为止，在讨论过的所有调度方法中（循环执行、FPS和EDF），都假定每个进程的最坏情况执行时间是已知的。它是任一进程调用需要的最大值。

最坏情况执行时间的估计（用*C*表示）可以通过测量或分析得到。测量方面的问题是难以确保最坏情况什么时候已经出现了。分析的缺点是必须有一个可用的处理器模型（包括高速缓存、管道、内存等待状态等）。

大多数分析技术涉及到两个不同的活动。第一个活动将进程代码分解为基本块的有向图。这些基本块代表直线代码（straightline code）。分析的第二个活动是取来对应于基本块的机器代码，使用处理器模型去估计它的最坏情况执行时间。

一旦知道了所有基本块的时间，有向图就能被折叠。例如，有两个基本块的简单选择结构将被折叠为单个的值（取两个值中的最大的一个）。循环用最大界限的知识进行折叠。

如果可以得到足够的语义信息，就可以采用更精巧的图形简化技术。关于这一点，给出一个简单的例子，考虑以下代码:

```
for I in 1..10 loop
  if Cond then
    -- 时间开销为100的基本块
  else
    -- 时间开销为10的基本块
  enf if;
end loop;
```

当没有进一步信息时，这个结构总的“费用”是10 × 100＋循环结构自身的时间开销，也就是1 005个时间单位。但是，它是可缩减的（通过代码的静态分析），条件Cond可能仅仅最
480 多三次为真。因此更乐观的时间开销是375个时间单位。

通过消除那些不可能执行的路径，代码内部的其他关系可以减少可行路径的数量，例如，当条件语句中的“if”分支成立时，可以排除“else”分支。采用这种语义分析的技术通常需要在代码中增加注释。图形简化过程可以使用像ILP（整数线性规划）这样的工具来产生最坏情况执行时间的接近的估算。这些工具也能对输入数据给出建议，这些数据能驱动程序走那条引起这个估计值的路径。

很明显，如果要分析一个进程的最坏情况执行时间，进程的代码就需要有一些限制。比如，所有循环和递归必须是有界的，否则就不可能脱机预测代码什么时候终止。而且，由编译器产生的代码也必须是可分析的。

最坏情况执行时间分析面临的最大挑战来自使用有片式高速缓存、管道和分支预测器等的现代处理器。所有这些特性都是为了减少平均执行时间，但是它们对最坏情况行为的影响

可能是难以预测的。如果忽略这些特性，估算结果可能是非常悲观的，但是要包括它们又总不是一件简单的事。办法之一是假设非抢占执行，那么，所有来自高速缓存等特性的好处都可以在估算中考虑。在后面的分析阶段，再来计算实际抢占的数目，并对高速缓存未命中和管道重填这样的情况进行惩罚。

为现代处理器的时序性行为建立详细模型不是一件很简单的事，这也许需要一些可能很难得到的专有信息。对于实时系统，要么采用更简单（但功能不强）的处理器体系结构，要么在测量工作上作出更多努力。考虑到所有高完整性的实时系统将受制于大量的测试，对代码单元（基本块）用测试和测量，而对完整的部件用路径分析法，将这两种方法结合起来使用，对当今的技术似乎是合适的。

13.8 偶发和非周期进程

为了扩充13.1节介绍的简单进程模型，让其包含偶发（和非周期）进程需求，值T被解释为最小（平均）到达时间间隔（minimum(or average) inter-arrival interval）（Audsley等，1993a）。T为20ms的一个偶发进程保证每20ms到达次数不会超过1次。实际上它的到达频率常常远小于每20ms一次，但是响应时间测试将保证能被支持的最大比率（如果通过了测试）。

把偶发进程包含进来的另一个需求涉及到时限的定义。简单模型假设$D=T$。对于偶发进程，这是不合理的。通常，偶发进程被用于封装错误处理例程或去响应告警信号。系统的故障模型可能说错误处理例程将很少被调用——但是当调用它时，它是紧急的，因此它有一个很短的时限。因此我们的模型必须区分D和T，并且允许$D<T$。的确，对于许多偶发进程，使应用能够定义小于周期的时限值是很有用的。 481

再回头审视一下13.5节描述的关于固定优先级调度的响应时间算法，可以看出：

- 只要把停止标准改为$w_i^{n+1}>D_i$，对于$D<T$的情况该算法依然工作得很好。
- 对任何优先级排序，它工作得很好——$hp(i)$总是给出更高优先级进程的集合。

虽然一些优先级排序优于另外的一些，但是对于给定的优先级排序，这个测试都能提供最坏情况响应时间。

在13.9节，将定义（并证明）$D<T$的最优优先级排序。稍后的小节将为$D<T$、$D=T$或$D>T$的一般情况研究一个扩充算法和最优优先级排序。

13.8.1 硬进程和软进程

对于偶发进程，可以定义平均到达率和最大到达率。遗憾的是，在许多情况下，最坏情况的值远远高于平均值。中断常常大量到达，不正常的传感器输入可能导致显著的额外计算。这时用最坏情况下的数值测量可调度性可能导致在实际运行系统中观察到非常低的处理器利用率。作为最低需求的指南，总是应该遵守下列两条规则：

- 规则1——按照平均执行时间和平均到达率，所有进程应该是可调度的。
- 规则2——按照所有进程（包括软进程）的最坏情况执行时间和最坏情况到达率，所有硬实时进程应该是可调度的。

规则1的一个推论是可能存在一些不可能满足所有当前时限的情形。这种状态被称为**暂时超载**（transient overload），但是，规则2确保没有硬实时进程错过它的时限。如果规则2引起了“正常执行”无法接受的低利用率，应该直接采取的动作是试着减少最坏情况执行时间（或到达率）。

13.8.2 非周期进程和固定优先级服务器

在一个基于优先级的方案中，调度非周期进程的一个简单方法是使这些进程运行的优先级低于硬进程的优先级。这样，非周期进程作为后台活动运行，因此，在抢占式系统中，它不能窃用硬进程的资源。虽然这是一个安全的方案，但是它没有对软进程提供足够的支持——如
482 果软进程只作为后台进程运行，它们常常错过它们的时限。为了改善软进程的这种状况，可以利用**服务器**。服务器保护硬进程需要的资源，但是以另外的方式使软进程尽可能快地运行。

自从服务器方法在1987年被首次提出以来，已经定义了许多种方法。这里仅考虑两种：可延期服务器（DS，Deferrable Server）和偶发服务器（SS，Sporadic Server）（Lehoczky等，1987）。

在DS中，采用一种分析方法（例如，使用响应时间方法），可以引进一个新的、在最高优先级的进程[⊖]。这个进程，也就是服务器，有一个周期T_s和一个容量C_s。选择这些值使系统中的所有硬进程仍然是可调度的，即使服务器以周期T_s和执行时间C_s周期性地执行也是如此。在运行时，当一个非周期进程到达时，如果服务器有可用容量，它就立即开始执行，直到它完成或服务器容量耗尽。在后一种情况下，非周期进程被挂起（或转换为后台优先级）。在DS模型中，服务器容量每T_s个时间单位被补充一次。

SS的操作不同于DS之处在于它的容量补充策略。在SS中，如果一个进程在时间t到达，并要求使用容量c，那么服务器在时间t后的T_s个时间单位补充容量c。通常，SS能提供比DS高的容量，但是它增加了实现开销。13.14.2节描述了POSIX如何支持SS，DS和SS可以用响应时间分析法来分析（Bernat and Burns，1999）。

因为所有服务器限制了非周期软进程可用的容量，它们也能用于保证偶发进程不会执行得比预期的更频繁。一个时间间隔为T_i和最坏情况执行时间为C_i的偶发进程，如果它不直接实现为一个进程，而是经由服务器实现，该服务器的$T_s=T_i$、$C_s=C_i$，那么这个偶发进程对较低优先级进程的影响（干扰）就被限定了，即使偶发进程到达得过快（这将是一种错误状态）也是这样。

所有服务器（DS、SS和其他）都可以被描述为*带宽预留*（bandwidth preserving），因为它们试图

- 使CPU资源对非周期进程立即可用（如果有容量）；
- 如果当前没有非周期进程，尽可能长地保留容量（通过允许硬进程执行）。

另一个常常比服务器技术性能更好的带宽预留方案是**双重优先级调度**（dual-priority scheduling）（Davis and Wellings，1995）。这里，优先级的范围被分为三档：高、中、低。所有非周期进程运行在中优先级。硬进程，当它们启动时，运行在低优先级，但是当它们要满足时限时，它们的优先级被及时提升到顶层。因此在执行的第一阶段，它们将为非周期活动让路（但如果没有这样的活动，它们就执行）。在第二阶段它们将转到较高优先级，优先于非
483 周期进程运行。在高优先级级别中，优先级根据时限单调方法（参看下面）来分配。将优先级提升到这一档发生在时间$D-R$。实现双重优先调度方案需要使用动态优先级方法。

13.8.3 非周期进程和EDF服务器

随着固定优先级系统的服务器技术的发展，大多数常见方法已经在动态EDF系统的背景

[⊖] 在其他优先级上的服务器也是可能的，但是如果给服务器的优先级高于所有硬进程的优先级，描述就更简单直观一些。

下被重新解释。例如，有一种动态偶发服务器（Spuri and Buttazzo，1994）。尽管静态系统需要优先级分配（在运行前进行），但是动态系统需要在它每次运行时都计算时限。从本质上说，当（且仅当）需要为一个重要的非周期进程服务、而服务器又有足够容量时，运行时算法分配给服务器以最短的当前时限。一旦服务器的容量耗尽，它就被挂起，直到被补充容量。

有一个不同方案，它同FPS的双重优先方案有许多类似之处，该方案就是**最早时限最后**（EDL，earliest deadline last）服务器，EDL由Chetto和Chetto（1989）定义。这里，硬进程被从在EDF下执行（如果没有非周期进程）切换到在EDL下执行（如果有非周期进程）。EDL方案保证每个硬进程满足它的时限，但是尽可能长地推迟进程的启动。因此，非周期进程可以立即使用服务器的容量（如果有多余的容量）。当硬进程的优先级被提升后，它将抢占软进程，并刚好在它的时限前运行完成。

13.9 D<T的进程系统

上面关于偶发进程的讨论中曾论证说，通常定义一个进程的时限小于它的到达时间间隔（或周期）一定是可能的。前面也曾指出当$D=T$时，对于固定优先级方案，速率单调优先级排序是最优的。Leung和Whitehead（1982）指出，当$D<T$时，也能定义一个类似的公式——时限单调优先级排序（deadline monotonic priority ordering，DMPO）。在这里，进程的固定优先级同它的时限成反比：（$D_i<D_j \Rightarrow P_i>P_j$）。表13-11对一个简单进程集合给出了合适的优先级分配。它也包含了最坏情况响应时间——用13.5节的算法计算。注意，速率单调优先级排序调度不了这些进程。

表13-11 DMPO的进程集合示例

进 程	周期，T	时限，D	计算时间，C	优先级，P	响应时间，R
a	20	5	3	4	3
b	15	7	3	3	6
c	10	10	4	2	10
d	20	20	3	1	20

在下面的子小节中，将要证明DMPO的最优性。有了这个结果和这种进程模型的响应时间分析的直接适应性，我们可以清楚地看到，固定优先级调度能充分处理这个更一般的调度需求集合。而EDF调度不能做到这一点。一旦进程的$D<T$，就不能使用简单的利用率测试（总利用率小于1）。此外，在13.6节讨论过的响应时间分析对于EDF要比对于FPS复杂得多。

说了EDF的这么多困难，但是请记住，EDF是更有效的调度方案。因此，任何通过FPS可调度性测试的进程，如果在EDF下执行，将总是满足它的时间需求。FPS的充分必要测试也能看作是EDF的充分测试。

证明DMPO是最优的

如果任一进程集合Q用优先级方案W是可调度的，用时限单调优先排序（DMPO）也是可调度的，那么DMPO就是最优的。DMPO最优性的证明包括Q的优先级（由方案W分配优先级）的变换，直到优先级按照DMPO排序。变换的每一步都保持可调度性。

令i和j是Q中的两个进程（它们的优先级相邻），且在方案W下有：$P_i>P_j$和$D_i>D_j$。定义方案W'，除了进程i、j相互交换，W'的其他方面同W是一样的。考虑Q在方案W'下的可调

度性:

- 所有优先级高于P_i的进程不会受到P_i的优先级变低的影响。
- 所有优先级低于P_j的进程不会受到影响。因为同交换前一样，它们都会受到来自i、j的干扰。
- 进程j在方案W下是可调度的，现在它有一个更高的优先级，会受较少的干扰，因此，在W' 下它必然是可调度的。

现在剩下的问题是需要证明: 进程i的优先级降低了，但它仍然是可调度的。

在方案W下，$R_j \leqslant D_j$，$D_j < D_i$且$D_i \leqslant T_i$，因此，在进程j执行期间，进程i仅仅受到一次干扰。一旦这两个进程被交换，进程i的新响应时间等于进程j的旧响应时间。这是真的，因为在两种优先级排序方法下，两进程总的计算时间为$C_j + C_i$，它们运行完成时会受到来自更高优先级进程同样程度的干扰。进程j在R_j期间仅启动一次，因此，在方案W' 下进程i执行期间仅受到j的一次干扰。可以得出:

$$R_i' = R_j \leqslant D_j < D_i$$

484~485 因此可以得出结论，进程i在交换之后是可调度的。

现在，通过再选择两个没有按DMPO排序的进程，并将它们交换，优先方案W' 就可以变换到W"。每一次这种交换保持了可调度性。最后没有进程需要交换，这时，排序就已经是DMPO所需要的排序，进程集也仍然是可调度的。因此，DMPO是最优的。

注意，对于$D = T$的特殊情况，上面的证明也可用于证明下述结论: 在这种情况下速率单调排序也是最优的。

13.10 进程交互和阻塞

在13.1节描述的系统模型中，包含有一个过分简化的假设，那就是需要进程是彼此独立的。这明显不合理，因为在所有有意义的应用中都需要进程交互。在第8章和第9章已经指出，通过对共享数据的某种形式的保护（例如，信号量、管程或保护对象）或者是直接交互（使用某种形式的会合），进程可以安全地交互。所有这些语言特征导致一个进程可能被挂起，直到将来一些必需的事件发生（如: 等待获得一个信号量的锁，或等待进入一个管程，或直到某个别的进程准备接受一个会合请求）。通常，同步通信导致更悲观的分析，因为当进程执行之间有许多依赖关系时，更难以定义真实的最坏情况。因此，当进程通过受保护的共享资源交换数据进行异步的通信与分析相关时，下面的分析就更精确。以下两节的主要内容是关于固定优先级调度。在讨论结束时，将考虑这个结果对EDF调度的适用性。

如果一个进程被挂起等待一个较低优先级进程完成一些必需的计算，那么这个优先级模型在某种意义上就被破坏了。在理想的情况下，这种**优先级反转**（Priority inversion）（Lauer and Satterwaite，1979）（即高优先级进程不得不等待较低优先级的进程）不应当存在。然而，通常它不可能完全消除。但是，它的负面影响能够被最小化。如果一个进程等待一个较低优先级进程，称为它被**阻塞**（blocked），为了测试可调度性，阻塞必须是有界的和可测量的，它也应该是很小的。

为了说明一个优先级反转的极端例子，考虑4个周期进程的执行: a、b、c和d。假设已经根据时限单调方案给它们分配了优先级，进程d的优先级最高，而进程a的优先级最低。进一

步假设进程d和a（以及进程d和c）共享临界段（资源），临界段用Q（和V）表示，并用互斥加以保护。表13-12给出了4个进程的细节和它们的执行序列，在表中，E代表一个滴答的执行时间，Q（或V）代表访问Q（或V）临界段的一个滴答的执行时间。因此，进程c执行了4个滴答，中间的两个滴答访问临界段V。

表13-12 执行序列

进 程	优 先 级	执 行 序 列	启 动 时 间
a	1	EQQQQE	0
b	2	EE	2
c	3	EVVE	2
d	4	EEQVE	4

图13-6说明了表中给出的各进程开始时间的执行序列。进程a首先启动，它执行并锁住临界段Q，然后它被进程c抢占，进程c执行一个滴答，锁住V，然后被进程d抢占。这个较高优先级的进程开始执行，直到它也希望锁住临界段Q。这时它必须被挂起（因为Q已被进程a锁住）。在这时，进程c重新得到处理器并继续执行。一旦它终止了，进程b将开始执行。仅当进程b已经完成时，进程a才能再次执行，它将完成对Q的使用，并允许d继续执行完成。在这种情况下，进程d在时间16完成，因此它的响应时间为12；进程c的响应时间为6，b为8，a为17。

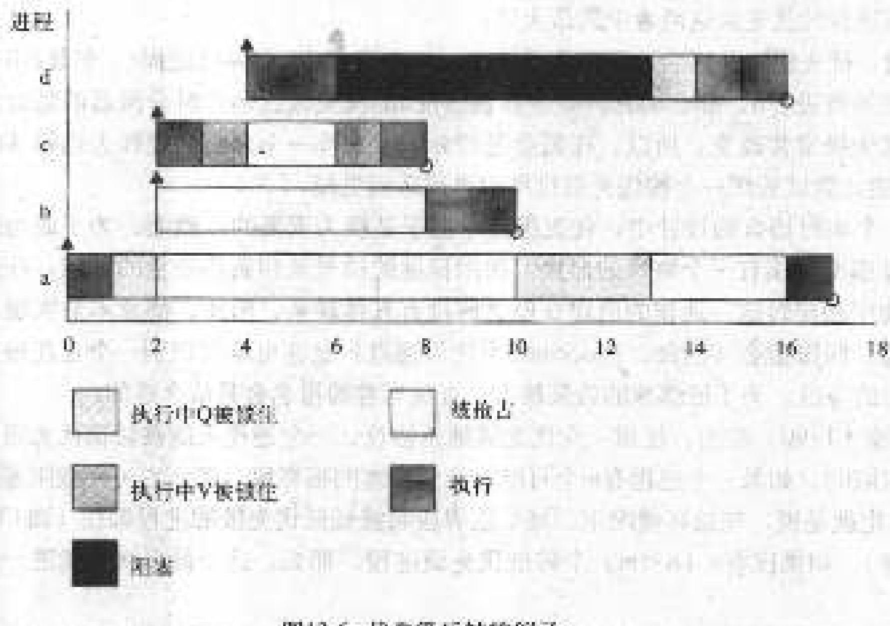


图13-6 优先级反转的例子

审视图13-6可以看出，进程d遭受了相当大的优先级反转。它不仅被进程a阻塞，也被b、c阻塞。一些阻塞是不可避免的，如果要维护临界段（以及共享数据）的完整性，那么进程a必须在d之前执行（当a有锁时）。但是进程d被进程c和b阻塞就是不应该的，并且会严重影响系统的可调度性（因为对进程d的阻塞过多）。 486/487

这种优先级反转是纯固定优先级方案的结果。限制这种影响的一种方法是使用**优先级继承**（priority inheritance）（Cornhill等，1987）。在优先级继承中，进程的优先级不再是静态的，

如果进程p被挂起，等待进程q进行某种计算，那么q的优先级变为等于p的优先级（如果它的优先级开始时比p的低）。在上面给出的例子中，进程a将被改变为进程d的优先级，因此它将先于进程b、c执行。图13-7显示了这种情况。注意，这个算法的一个后果是，现在进程b将经受阻塞，即使它不使用共享对象。也应注意到现在进程d被阻塞两次，但是它的响应时间已经减少到9。

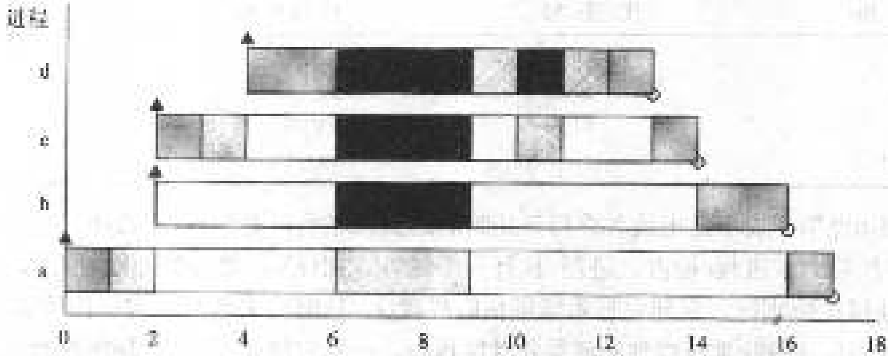


图13-7　优先级继承的例子

在这种简单继承规则中，一个进程的优先级是它自己的默认优先级和在那一时间依赖于它的所有进程的优先级这两者中的最大值。

通常，优先级继承不会被限制在单步中。如果进程d正在等待进程c，但是c不能运行，因为它正在等待进程b，那么b将会同c一样被分配d的优先级。运行时分派器的隐含含义是一个进程的优先级常常改变，所以，在需要进行分派时选择一个合适的进程去运行（或使它可运行），可能比尝试管理一个按优先级排序的进程队列更好。

在一个实时语言的设计中，优先级继承似乎是极为重要的。然而，为了成为最有效的模型，并发模型应该有一个特殊的形式。使用标准的信号量和条件变量的时候，在进程变为挂
488 起这一动作和反转这一动作的进程身份之间没有直接联系。因此，继承不易实现。使用同步消息传递、间接指名（例如，在occam2中使用通道）也许也难以识别一个正在被其他进程等待的进程的身份。为了使继承的效果最大，直接对称的指名会是最合适的。

Sha等（1990）指出，使用一个优先级继承协议，一个进程可能被较低优先级进程阻塞的次数是有限的。如果一个进程有m个可能导致它阻塞的临界段，那么它可能被阻塞的最多次数就是m。也就是说，在最坏情况下，每个临界段将被较低优先级的进程锁住（图13-7中的情况就是这样）。如果仅有n（$n<m$）个较低优先级进程，那么，这个最多次数能进一步缩减（减为n）。

如果B_i是进程i所能容许的最多阻塞次数，那么，对于这个简单的优先级继承模型，可以容易地找到一个计算B_i的公式。令K是临界段（资源）的数目，式（13-6）提供了B_i的上限。

$$B_i=\sum_{k=1}^{K}usage(k,\ i)C(k) \tag{13-6}$$

其中$usage$是一个0/1函数：$usage(k,\ i)=1$，如果资源k至少被一个优先级低于P_i的进程使用，且至少有一个优先级高于或等于P_i的进程；否则它的值为0。$C(k)$是关于k临界段在最坏情况下的执行时间。

这个算法对这个继承协议不是最优的，只是用于说明计算B时要考虑的因素。在13.11节，将描述一个更好的继承协议，也将给出计算B的改进公式。

响应时间计算和阻塞

假设已经得到了B的值，就能修改响应时间算法，把阻塞因素加进去[⊖]：

$$R = C + B + I$$

也就是，

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \qquad (13\text{-}7)$$ 489

当然这也能通过构造一个递推关系来求解:

$$w_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j \qquad (13\text{-}8)$$

注意，现在这个公式也许是悲观的（即不一定是充分必要条件）。一个进程是否实际容许它的最多阻塞次数将依赖于进程所处环境。例如，如果所有进程是周期的，且都有同样的周期，那么不会发生抢占，因此也不会发生优先级反转。但是，通常公式（13-7）代表了包含有合作进程的实时系统的有效调度测试。

13.11 高限优先级协议

标准的继承协议给出了一个高优先级进程可能遇到的阻塞次数的上限，但这个上限可能仍会导致不可接受的最坏情况计算。这又混合进了阻塞链生长（传递阻塞）的可能性，也就是，进程c被进程b阻塞，而b被a阻塞，等等。因为共享数据是系统资源，从资源管理的角度，不仅应该使阻塞最小化，而且应该消除诸如死锁之类的失效状态。高限优先级协议（ceiling priority protocols）解决所有这些问题（Sha等，1990）。本章将考虑其中两个协议：**原始高限优先级协议**（original ceiling priority protocols）和**立即高限优先级协议**（immediate ceiling priority protocols）。首先描述原始协议（OCPP），接着描述更简单一点的立即协议（ICPP）。当这两个协议中的任何一个用于单处理器系统时:

- 一个高优先级进程在其执行期间最多能被较低优先级进程阻塞一次。
- 死锁被预防了。
- 传递阻塞被预防了。
- 资源的互斥访问得到保证（被协议自身保证）。

高限协议可能最好用临界段保护的资源来描述。本质上，这种协议保证如果一个资源被锁住，假如被进程a锁住，可能导致一个更高优先级进程（b）被阻塞，那么不允许别的进程（除了进程a）锁住其他可能阻塞进程b的资源。因此，一个进程试图锁住一个先前已上锁的资源，以及当上锁可能导致更高优先级进程的多重阻塞时，该进程都被延迟。

原始的协议采取以下形式:

1）每个进程分配有一个静态的默认优先级（也许是由时限单调方案分配的）。 490

⊖ 阻塞也能加入到基于利用率的测试，但是现在每个进程必须被单独地考虑。

2）每个资源有一个确定的静态高限值，这个值是使用它的进程的最高优先级。

3）一个进程有一个动态优先级，它是进程自己的静态优先级和任何它继承的优先级中的最大值。它继承的优先级来自于它阻塞的较高优先级进程。

4）只有在一个进程的动态优先级高于任何当前锁住资源的高限值的时候（除了它自己锁住的资源），该进程才能锁住一个资源。

允许锁住第一个系统资源。该协议的效果是保证：仅当不存在使用两个资源的更高优先级进程，第二个资源才能被锁住。因此，一个进程能被阻塞的最长时间等于被较高优先级进程访问的任何较低优先级进程中的最长临界段的执行时间，于是，式（13-6）变为：

$$B_i = \max_{k=1}^{K} usage(k, i)C(k) \tag{13-9}$$

高限协议的好处是高优先级进程只能被较低优先级进程阻塞一次（每次激活时），这个结果的代价是更多的进程将经历这种阻塞。

不能通过一个例子来说明这个算法的所有特性，但是图13-8显示的执行序列很好地说明了算法如何工作，并提供了同前面方法的比较（该图说明的是图13-7和13-6所用的同一进程序列）。

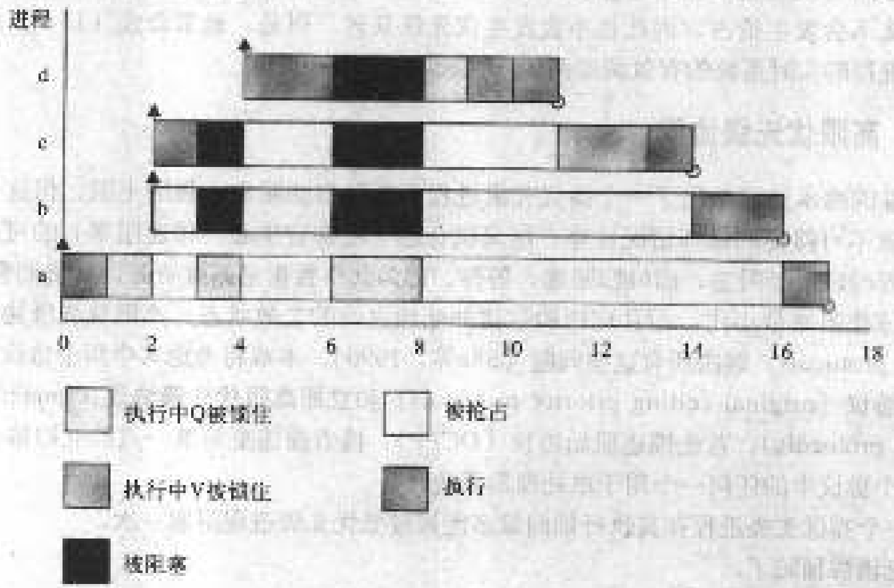


图13-8 优先级继承的例子——OCPP

在图13-8中，进程a再次锁住第一个临界段，因为没有其他资源已经被锁住，它又被进程c抢占，但是现在c试图锁住第二个临界段V没有成功，因为它的优先级（3）不高于当前的高限（高限是4，因为Q被锁住并被进程d使用）。在时间3，a阻塞c，因此a以其优先级3运行，因而阻塞b。更高优先级进程d在时间4抢占a，但是随后当它试图访问Q时就被阻塞了。因此a将继续执行（用优先级4），直到它释放Q上的锁，且它的优先级降为1。现在，d能继续运行直到它完成（响应时间为7）。

高限优先级协议保证一个进程在每次调用期间仅被阻塞一次。但是，图13-8显示进程b（还有c）遭受了两次阻塞，实际情况是单个的阻塞被进程d的抢占分为两部分。式（13-9）确

定。所有进程（除了进程a）将遭受最长为4的单个阻塞。图13-8说明，对于这个特定的执行序列，进程c和进程d实际遭受一个长为3的阻塞，进程d遭受一个长仅为2的阻塞。

13.11.1 立即高限优先级协议

立即高限优先级算法（ICPP）采取一个更直接的方法，当它一旦锁住一个资源，它就立即提升进程的优先级（而不是只在它实际阻塞一个较高优先级进程时）。该协议定义如下：

- 每个进程分配有一个静态的默认优先级（也许是由时限单调方案分配的）。
- 每个资源有一个确定的静态高限值，这个值是使用它的进程的最高优先级。
- 一个进程有一个动态优先级，它是进程自己的静态优先级和它锁住的任何资源的高限值中的最大值。

作为最后这条规则的推论，一个进程将仅在它刚开始执行时遭受阻塞。一旦这个进程实际开始执行，它所需要的所有资源必须都是空闲的；如果不是这样，那么，某个进程就会拥有相等或更高的优先级，而这个进程的执行将被延迟。

前面曾用过的那个进程集合现在可以在ICPP下执行（见图13-9）。

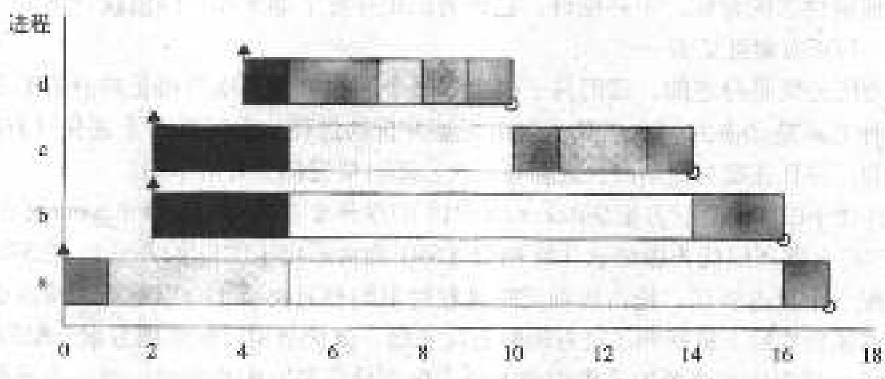


图13-9　优先级继承的例子——ICPP

进程a在时间1锁住Q，接着在优先级4运行4个滴答。因此，进程b、c、d都不能开始运行。一旦a解锁Q（它的优先级也随之降低），其他进程按优先级顺序执行。注意所有的阻塞发生在实际执行前，进程d的响应时间现在仅仅是6。然而，这有点令人误解，因为两种协议的最坏情况阻塞时间是一样的（见式（13-9））。 491-492

尽管两种高限方案的最坏情况行为是相同的（从调度的观点），但也有一些不同点：

- ICPP比OCPP易于实现，因为不需要监控阻塞关系。
- ICPP导致更少的上下文切换，因为阻塞在首次执行之前。
- ICPP需要更多的优先级移动，因为这会在各种资源使用中发生；OCPP仅在实际阻塞发生时才改变优先级。

最后，注意协议ICPP在POSIX中称为优先级保护协议（Priority Protect Protocol），在实时Java中称为高限优先级仿真（Priority Ceiling Emulation）。

13.11.2 高限协议、互斥和死锁

尽管以上两种高限协议的算法是用资源上的锁定义的，但是必须要强调的是，是这些协议本身而不是其他的同步原语提供了对资源的互斥访问（至少在单处理器系统上是这样）。考

虑一下ICPP，如果一个进程访问某个资源，那么它运行时的优先级等于高限值。其他使用这个资源的进程的优先级不可能比这个更高，因此，正在执行的这个进程要么不受阻碍地执行，且能使用该资源，要么它被抢占，但新的进程不会使用这个特定资源。不管哪种方法，互斥都得到保证。

高限协议的另一主要性质（也是针对单处理器系统）是它们不会产生死锁。在第11章，对无死锁资源的使用问题进行了详细研究。高限协议是一种死锁预防的形式。如果一个进程
493 在拥有一个资源的同时申请另一个，那么第二个资源的高限优先级不可能低于第一个的高限。的确，如果（不同进程）以不同的顺序使用两个资源，那么它们的高限必然是一样的。因为一个进程不会被另一个同样优先级的进程抢占，一旦进程获得了一个资源，那么对于所有其他的资源，在需要它们时，都是空闲的。没有循环等待的可能，且死锁被预防了。

13.11.3 阻塞和EDF

当考虑共享资源和阻塞时，在EDF和FPS之间有一个直接的类比。FPS会遭受优先级反转，而EDF会遭受时限反转。这发生在一个进程需要一个资源，但是该资源正被另一个有较长时限的进程锁住了的时候。不必惊奇，已经为EDF开发了继承和高限协议，但是，正如前面做的比较，EDF方案更复杂一点。

因为优先级是静态的，因而易于决定哪一个进程可能阻塞当前正被分析的进程。在EDF中，这种关系是动态的，它依赖于当前正被分析的进程启动时哪一个进程（有较长时限的）是活动的。并且在超周期期间，进程每一次启动时情况都会有所不同。

或许关于EDF的最好方案是Baker（1991）的*堆栈资源策略*（Stack Resource Policy，SRP）。它同FPS的立即高限优先级协议非常相似（SRP确实影响了ICPP的开发）。在SRP下，每个进程被分配一个抢占级别。抢占级别反映进程时限的相对关系，时限越短，抢占级别越高，因此，抢占级别实际上就指明了进程的静态优先级，就像是用时限单调方案分配的优先级一样。在运行时，基于使用资源的进程的最大抢占级别给资源分配高限值。当一个进程启动时，仅当它的绝对时限比当前运行进程短，且它的抢占级别高于当前被锁住资源的最高高限时，它才能抢占正运行的进程。这个协议的结果同ICPP是一样的。进程仅遭受一次阻塞，且发生在它启动时，死锁被预防了，而且有一个计算阻塞时间的简单公式。

13.12 一个可扩充的进程模型

前面曾指出，13.1节介绍的模型过于简单化难以实际使用。在随后的小节里，取消三个重要限制，从而:

- 时限能小于周期（$D<T$）。
- 偶发和非周期进程同周期进程一样得到支持。
- 将阻塞作为响应时间公式的考虑因素，进程交互成为可能。

在这一节里，将给出五个进一步的推广。只研究固定优先级调度，因为EDF分析在这一
494 领域还不成熟。本节将用一个通用的优先级分配算法作为结束。

13.12.1 合作调度

上面描述的模型都需要真正的抢占式分派。在这一节，将介绍一个替代方案（使用延期抢占）。它有一些优点，且仍能使用体现在公式（13-7）中的调度技术来进行分析。在公式

（13-7）中，有一个阻塞项B，它计算当较高优先级进程可运行时较低优先级进程可以继续执行的时间。在应用领域，这种情况可能是由不同优先级的进程共享数据（互斥使用）而引起的。然而，阻塞也可能由运行时系统或内核引起。许多系统将非抢占式上下文切换的时间作为最长的阻塞时间（例如，一个较高优先级进程的启动被内核延迟一段时间，这就是内核切换上下文到一个较低优先级所用的时间——即使会接着立即发生到较高优先级进程的切换）。

使用立即高限优先级协议（去计算和限制B）的优点之一是阻塞不会累积。一个进程不可能被一个应用进程和一个内核例程两者阻塞——当更高优先级进程启动的时候，只有一个阻塞可能实际发生。

合作调度通过增加阻塞可能出现的情形利用了这种非累积性质。令B_{MAX}是系统中的最长阻塞时间（使用传统方法）。应用程序代码被分为非抢占块，这些块的执行时间被限定在B_{MAX}内。在每个块的最后，应用程序代码向内核提供一个"剥夺调度"（de-scheduling）请求。如果一个高优先级进程现在是可运行的，内核将激发上下文切换；如果没有可运行的高优先级进程，当前运行进程将继续运行下一个非抢占块。

这样，应用程序代码的常规执行完全是合作的。一个进程将继续执行直到它提供一个剥夺调度请求。因此，只要任何临界段被完全包含在剥夺调度调用之间，互斥就得到保证。所以，这种方法确实需要小心放置剥夺调度调用。

为了对被破坏的（或不正确的）软件提供某种级别的保护，如果任何非抢占块运行时间超过B_{MAX}的话，内核可能使用异步信号或中止去清除应用进程。

使用延期抢占有两个重要优点。它增加了系统的可调度性，而且它能导致较低的C值。在方程（13-4）的求解中，因为w的值正在扩大，新启动的较高优先级进程可能进一步增加w的值。在延期抢占中，在最后块的执行期间没有干扰发生。令F_i是最后块的执行时间，那么当该进程已经运行了$C_i - F_i$个时间单元的时候，最后块才（刚刚）开始。方程（13-4）现在要对$C_i - F_i$而不是对C_i求解：

$$w_i^{n+1} = B_{MAX} + C_i - F_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j \qquad (13\text{-}10)$$

当迭代收敛时（$w_i^{n+1} = w_i^n$），响应时间由下式给出：

$$R_i = w_i^n + F_i \qquad (13\text{-}11)$$ 495

实际上，进程的最后块执行时的优先级（最高的）比进程其余部分的优先级要高。

延期抢占的另一个优点是它能更精确地预测进程的非抢占基本块的执行时间。现代处理器有高速缓存、预取队列和管道，这些都明显减少了直线代码的执行时间。通常，最坏情况执行时间的简单估计被迫忽略掉这些优点，从而得到非常悲观的结果，因为抢占使高速缓存和管道无效。非抢占的知识能用于预测实际执行的速度。但是，如果提供上下文切换的开销很高，那将对这些优点产生影响。参看16.3.4节关于如何为高速缓存效应对可调度性分析的影响建模的讨论。

13.12.2 启动抖动

在简单模型中，所有进程被假设是周期性的，并且完全周期性地启动；也就是说，如果进程l的周期为T_l，那么它精确地按照这个频率启动。通过假设偶发进程的最小到达时间间隔

为T将它们并入到了这个模型中来。然而，这不总是一个现实的假定。考虑一个偶发进程s，它被一个在另一处理器上的周期进程l启动。进程l的周期是T_l，偶发进程将有同样的速率，但是，如果认为偶发进程s是一个周期进程，且周期$T_s=T_l$，可以用式（13-4）或（13-5）来表示s施加在低优先级进程上的最大负载（干扰），那是不正确的。

为了理解为什么这是不正确的，考虑进程l的连续两次执行。假设启动进程s的事件发生在周期进程执行的最后阶段。在进程l的第一次执行时，假设该进程直到最后可能的时间才完成，也就是R_l时才完成。然而，在它第二次运行时，假设它没有受到干扰，所以它在C_l时间内就完成了。因为这个值可能是任意小的，所以让它等于0。这时，偶发进程的两次执行的间隔就不是T_l，而是T_l-R_l。图13-10说明了这种行为，这里$T_l=20$，$R_l=15$，且最小的C_l等于1（也就是，偶发进程的两次启动发生在6个时间单位之内）。注意这种现象仅在进程l是远程进程的时候才有意义。如果不是这种情况，那么进程s启动的这种变异可以用标准的公式计算出来，在这里要假定在启动进程和被启动进程间有一个临界瞬间。

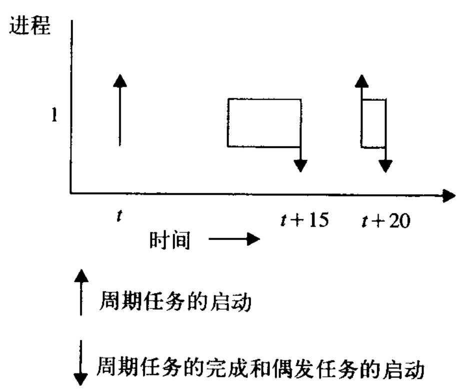


图13-10 偶发进程的启动

为了正确捕获偶发进程对其他进程的干扰，必须修改递推关系。进程启动的最大变异被称为它的**抖动**（用J表示）。例如，在上面的例子中，进程s的抖动值为15。按照它对更低优先级进程的最大影响，这个偶发进程将在时间0、5、25、45等等被启动。也就是在时间0、$T-J$、$2T-J$、$3T-J$等等被启动。观察可调度性公式的推导可以发现，如果R_i在0到$T-J$之间，也就是$R_i\in[0, T-J)$，进程i将受进程s的一次干扰，如果$R_i\in[T-J, 2T-J)$，则受2次干扰，如果$R_i\in[2T-J, 3T-J)$，则受3次干扰，等等。对这些条件做微小的重新整理就可得到，如果$R_i+J\in[0, T)$则有一次干扰，$R_i+J\in[T, 2T)$则有两次干扰，等等。这一结论可以用同先前的响应时间公式一样的形式描述如下（Audsley等，1993a）：

$$R_i = B_i + C_i + \sum_{j\in hp(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil C_j \tag{13-12}$$

通常，周期进程不会有启动抖动。然而，实现也许会限制系统定时器（它用于启动周期进程）的粒度。在这种情况下，周期进程可能也会有启动抖动。例如，T为10，而系统定时器的粒度为8，抖动值将为6——在时间16时周期进程才会被第二次启动（在时间10发出的调用）。如果要测量相对于实际启动时间的响应时间（现在被记为$R_i^{periodic}$），那么抖动值必须加入到先

前的计算中:

$$R_i^{periodic} = R_i + J_i \tag{13-13}$$

如果新的值比T_i大，那么必须使用下面的分析方法。

13.12.3 任意的时限

为了满足D_i（因此也可能包括R_i）可能大于T_i的要求，必须再次调整分析方法。当时限小于（或等于）周期时，只需要考虑每个进程的单次启动。当所有较高优先级进程在同一时间被启动时，临界瞬间表示最大的干扰，因此紧接着在临界瞬间启动的响应时间必然是最坏情况。但是当时限大于周期时，就必须考虑多个启动了。下面假设一个进程的启动将被延迟直到同一进程的前面任一次的启动都已完成。 496~497

如果一个进程执行到下一个周期，就必须分析两次启动，看哪一次会引起最长的响应时间。此外，如果第二次启动在第三个周期开始时还未完成，那么新的启动也要被考虑，等等。

对于每一个潜在的重叠启动，定义一个单独的窗口$w(q)$，q是一个整数，用于标识一个特定窗口（也就是，$q=0,1,2,\ldots$）。等式（13-5）能被扩充为以下形式（忽略抖动）(Tindell等，1994):

$$w_i^{n+1}(q) = B_i + (q+1)C_i + \sum_{j\in hp(i)} \left\lceil \frac{w_i^n(q)}{T_j} \right\rceil C_j \tag{13-14}$$

例如，当$q=2$时，在这个窗口中将出现这个进程的三次启动。对于每个q值，能通过迭代找到一个稳定的$w(q)$值——就像式（13-5）一样。响应时间可以给出如下:

$$R_i(q) = w_i^n(q) - qT_i \tag{13-15}$$

例如，在时间$2T_i$启动的进程将进入$q=2$的窗口，因此响应时间是窗口大小减去$2T_i$。

需要考虑的启动次数被限定在满足下式的最小的q值以内。

$$R_i(q) \leqslant T_i \tag{13-16}$$

这样，进程在下一次启动前完成，因此后面的窗口不会重叠。最坏情况响应时间就是每个q的最大值:

$$R_i = \max_{q=0,1,2,\ldots} R_i(q) \tag{13-17}$$

注意，对于$D \leqslant T$，当$q=0$时，式（13-16）为真。在这种情况下，式（13-14）和（13-15）简化回以前的等式。如果$R>D$，那么进程是不可调度的。

当这个关于任意时限的公式要结合启动抖动的影响时，必须对以上的分析做两个变更。首先，就像以前一样，如果较高优先级进程遭受启动抖动，那么干扰因素就必然增加:

$$w_i^{n+1}(q) = B_i + (q+1)C_i + \sum_{j\in hp(i)} \left\lceil \frac{w_i^n(q)+J_j}{T_j} \right\rceil C_j \tag{13-18}$$ 498

另一个变更涉及进程自身。如果它可能遭受启动抖动，如果响应时间加上抖动大于周期的话，那么两个连续的窗口可能重叠。为了适应这一点，式（13-15）必须被改为:

$$R_i(q) = w_i^n(q) - qT_i + J_i \tag{13-19}$$

13.12.4 容错

通过向前或向后出错恢复来容错总会引起额外计算。这可能是一个异常处理程序或一个恢复块。在实时容错系统中，即使有某种级别的故障出现，时限应当仍然得以满足。这种容错的级别被称作**故障模型**（fault model）。如果C_i^f是进程i的错误引起的额外计算时间，那么响应时间公式可以容易地被改为:

$$R_i = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \max_{k \in hep(i)} C_k^f \tag{13-20}$$

其中，$hep(i)$是优先级等于或高于进程i的进程的集合。

在这里，故障模型定义了一个故障的最大值，并且这里还有一个假设，即一个进程将用与常规计算同样的优先级执行恢复动作。可以容易地在式（13-20）中加入被允许的故障的数目（F）:

$$R_i = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \max_{k \in hep(i)} F C_k^f \tag{13-21}$$

确实，可以这样分析一个系统: 增加F的值，看看可以容忍多少故障（爆发式到达）。另一方面，故障模型可以指出最小的故障到达间隔，在这种情况下等式变为:

$$R_i = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \max_{k \in hep(i)} \left(\left\lceil \frac{R_i}{T_f} \right\rceil C_k^f \right) \tag{13-22}$$

其中T_f是故障间的最小到达间隔时间。

在式（13-21）和（13-22）中，做了一个假设，即在最坏情况下，故障总是出现在有最长恢复时间的进程中。

13.12.5 引入偏移量

本章到目前为止介绍的调度分析都假定所有进程共享同一启动时间。这个临界瞬间是指
499 所有进程同时启动的时刻（通常发生在时间0）。对于固定优先级调度，这是一个安全的假设，当一起启动时，如果所有进程都满足它们的时间需求，那么它们就总是可调度的。然而也有一些周期进程集合可因明确地选择启动时间而从中获益，所以它们不共享一个临界瞬间。这时，一个进程被认为有一个关于其他进程的**偏移量**（offset）。考虑表13-13中说明的3个进程。

表13-13 进程集合的例子

进　程	T	D	C
a	8	5	4
b	20	10	4
c	20	12	4

如果假定有一个临界瞬间，那么进程a的响应时间为4，进程b的响应时间为8，但是第三个进程有一个最坏情况的响应时间16，这超过它的时限。对于进程c，来自进程b的干扰足够长，以至引起更一步来自进程a的干扰，这一点至关重要。然而，如果进程c被给予偏移量（O）10（也就是，保持相同的周期和相对时限，但是在时间10第一次启动）那么它将永远不会在

同进程b一样的时间执行。结果是一个可调度的进程集——见表13-14。

表13-14　进程集合的响应时间分析

进　程	T	D	C	O	R
a	8	5	4	0	4
b	20	10	4	0	8
c	20	12	4	10	8

遗憾的是，有任意偏移量的进程集合不易于进行分析。选择合适的偏移量以使进程集具有最优的可调度性是一个很强的NP难度问题。的确如此，甚至连检查带有偏移量的进程集是否共享临界瞬间这样的问题都是一个远非简单的问题[⊖]。

虽然有这些理论上的问题，也存在能够通过相对简单（虽然不一定是最优）的方法进行分析的进程集合。在大多数现实的系统中，进程周期不是任意的，但很可能互相相关。正如刚才列举的例子，两个进程拥有相同的周期。在这种情况下，可以很容易地给一个进程以偏移量（$T/2$），并使用一种消除偏移量的变换技术去分析这样产生的系统——这时临界瞬间分析就适用了。在这个例子中，进程b和c（c有偏移量10）都可被一个周期为10、计算时间为4、时限为10、但是没有偏移量的假想进程替换。这个假想进程具有两个重要的性质。 500

- 如果它是可调度的（当同所有其他的进程共享一个临界瞬间时），当两个实际进程中的一个被给予半个周期的偏移量时，这两个实际进程将能满足它们的时限。
- 当受到假想进程（和所有其他的高优先级进程）的干扰时，如果所有较低优先级进程是可调度的，那么，当这个假想进程被两个实际进程（其中一个有偏移量）替换时，它们也仍然可调度。

这些性质来自以下观察：假想进程使用的CPU时间总是比这两个实际进程更多（或相等）。表13-15说明这种分析就能适用于变换后的进程集合。这个假想进程在表中以“n”命名的。

表13-15　变换后进程集合的响应时间分析

进　程	T	D	C	O	R
a	8	5	4	0	4
n	10	10	4	0	8

更一般地，假想进程参数可以从实际进程a和b计算得出，如下所示：

$$T_n = T_a/2\ (\text{或}T_b/2\text{因为}T_a = T_b)$$
$$C_n = Max(C_a, C_b)$$
$$D_n = Min(D_a, D_b)$$
$$P_n = Max(P_a, P_b)$$

其中P表示优先级。

很明显，适合于两个进程的结论，对于三个或者更多进程也是适用的。Bate和Burns（1997）给出了这些技术更全面的描述。概要地说，虽然不可能有效地分析任意的偏移量，但是明智地使用偏移量和转换技术可以使分析问题得以简化，回到共享临界瞬间的简单进程集

⊖ 一个有趣的结果是具有互质（co-prime）周期的进程集总是有临界瞬间，不管选择什么偏移量（Audsley and Burns，1998）。

合的状况。因此，本章前面小节中给出的所有分析方法都适用。

13.12.6 优先级分配

前面给出的针对任意时限的公式有一个特性，那就是没有一个简单算法（如速率或时限
501 的单调排序）给出最优的优先级排序。在本节里，给出一个定理和一个算法，用于在任意时限情况下分配优先级。这个定理考虑了最低优先级进程的行为（Audsley等，1993b）。

定理：*如果进程p被分配了最低优先级并且是可行的，那么，如果完整的进程集合存在一个可行的优先级排序，则存在进程p被分配最低优先级的排序。*

对这个定理的证明来自对可调度性等式的考虑——例如式（13-14）。如果进程具有最低优先级，那么它就受到所有更高优先级进程的干扰。这种干扰是不依赖于这些更高优先级的实际顺序的。因此，如果一个进程在被分配它能允许的最低优先级时仍是可调度的，那么剩下的工作就是分配其他$N-1$个优先级。幸运的是，这个定理能被重新运用于这个缩减了的进程集。因此通过连续的重复运用，就可得到一个完整的优先级排序（如果存在的话）。

以下Ada语言代码实现了这个优先级分配算法，其中Set是想要按优先级排序的进程数组，Set（N）为最高优先级，Set（1）为最低优先级。过程Process_Test测试进程K在数组中的位置是否是可行的。这个双循环先将一个进程放在最低优先级的位置，看其是否可行，不行就用下一个进程，直到找到一个进程可以放在该位置，这个进程就被固定在这个位置。然后考虑下一个优先级的位置。如果在任何时刻内部循环不能找到可行的进程，整个过程就失败了。注意如果有一个额外的交换可用，就可能有一个简明的算法。

```
procedure Assign_Pri (Set : in out Process_Set; N : Natural;
                      Ok : out Boolean) is
begin
  for K in 1..N loop
    for Next in K..N loop
      Swap(Set, K, Next);
      Process_Test(Set, K, Ok);
      exit when Ok;
    end loop;
      exit when not Ok;  -- 寻找可调度进程失败
  end loop;
end Assign_Pri;
```

如果可行性的测试是准确的（必要且充分），那么优先级排序就是最优的。因而对于任意的时限（没有阻塞），就得到一个最优的排序。

13.13 动态系统和联机分析

本章前面注意到，针对不同的应用需求，已经建立了各种各样的调度方案。对于硬实时
502 系统，脱机分析是理想的（其实，它经常是强制性的）。为了进行这种分析，需要：

- 来临工作的到达模式是已知的和有界的（这样导致了一个固定的进程集合，它具有已知的周期或者最坏情况到达间隔）；
- 有界的计算时间；
- 一个导致应用进程的可预测执行的调度方案。

本章已经说明了固定优先级调度（在一定范围内也包括EDF）是如何提供一个可预测的执行环境的。

与硬实时系统不同，动态软实时应用的到达模式和计算时间事先并不知道。虽然还能使用某个级别的脱机分析，但会是不完整的，因此需要某种形式的联机分析。

联机调度方案的主要任务是管理由于动态系统环境造成的极有可能出现的超载。本章前面就指出了EDF是一个动态调度方案，它是最优的调度规则。但是，EDF有一种特性：在短暂的超载期间，它的表现非常差。它可能导致一个级联式的后果——当一个进程错过它的时限时，由于它占用充足的资源，因而导致下一个进程也错过时限。

为了对付这个有害的多米诺效应，许多联机方案有两种机制。

1）处理器有一个进入许可控制模块，它限制允许竞争处理器的进程数目；

2）有一个EDF分派例行程序，它对允许进入的进程进行分派。

理想的进入许可算法应防止处理器超载，以便EDF例程有效地工作。

如果想要一些进程被允许进入、其他的被拒绝，每个进程的相对重要性必须是已知的。通常通过赋值可以达到。值可以按如下分类：

- 静态的——无论进程何时启动，它总是有同样的值。
- 动态的——进程的值仅在进程启动时计算（因为它依赖于环境因素或者当时的系统状态）。
- 适应的——系统具有动态特性，进程的值会在执行过程中改变。

为了分配静态值（或者为动态或适应的方案构造算法和定义输入参数）要求领域专家清晰明白地说出他们对系统理想行为的理解。就像其他的计算领域一样，知识推导（knowledge elicitation）也有它自身的问题。但是这些问题不在这里考虑（参看文献（Burns等，2000））。

联机分析的基本问题之一是，需要在调度决策的质量和资源及时间之间做出权衡。一个极端是，每次一个新的进程抵达，整个进程集合进行精确的脱机测试，例如本章前面描述过的那些测试。如果进程集合是不可调度的，有最小值的进程就被排除，再重复测试（直到得到可调度的进程集）。对于值的静态或动态分配，这个方法（被称为*最佳工作量*（best effort）方法）是最优的——但这仅仅是在测试的开销被忽略的前提下。一旦计入了这种开销，这个方法的有效性就要严重受损。通常，联机调度必须使用启发法，并且任何单个的方法不可能适用于所有的应用。这仍然是一个活跃的研究领域。很明显，现在所需要的不是在语言或操作系统标准中定义的单个办法，而是需要一套机制，在该机制下，应用可以制定它们自己的方案去满足它们特殊的需求。 503

本节最后要说的是包含硬成分和动态成分的混合系统。在许多应用领域这些将很可能成为标准。即使在完全静态的系统中，对软进程或固进程进行增值计算（value-added computation）以改进硬进程的质量，也是构造系统的一个有吸引力的方式。在这些情况下，如13.8.1节所描述的，必须保护硬进程，以免受到由非硬进程的行为引起的超载的影响。实现这一点的一个方法是对硬进程使用固定优先级调度，对其他进程使用服务器。服务器可以体现理想的进入许可策略，并使用EDF为进来的动态工作提供服务。

13.14 基于优先级系统的编程

很少有程序设计语言明确定义优先级作为其并发设施的一部分。它们常常只提供初步的模型。例如，在occam2中有一个PAR构造的变体，它指出应该给指定的进程分配静态优

先级:

```
PRI PAR
  P1
  P2
  PAR
    P3
    P4
  P5
```

这里用到相对优先级，PRI PAR中进程的文本顺序有着重要意义。因此P1有最高优先级，P2有次高优先级，P3和P4又都是下一级优先级，P5有最低优先级。它不需要实现支持最小的优先级范围，也不支持优先级继承。

试图给予更多完整供应的是Ada语言，因此，将在下一小节详细论述Ada。传统上，基于优先级的调度是一个操作系统的问题，而不是一个语言问题。因此，讨论完Ada后再评述POSIX

504 的设施。Ada和POSIX都假定任何可调度性分析已经脱机执行。最近，实时Java试图提供同Ada和POSIX一样的设施，但增加了对联机可行性分析的支持选项，这将在13.14.3节研究。

也许应该指出，虽然大多数通用操作系统提供了进程或线程优先级的概念，但是它们的设施通常不足以用于硬实时编程。

13.14.1 Ada

如同在本书序言中指出的，Ada语言由一个核心语言加上一些用于特定应用领域的附件组成。这些附件不包含任何新的语言特性（新的语法），只是定义了一些使用特定附件所必须支持的程序编用和库包。本节将研究实时系统附件提供的某些设施。尤其是要考虑允许给任务（和保护对象）分配优先级的那些设施[⊖]。在System包中，有如下声明:

```
subtype Any_Priority is Integer range 实现定义的;
subtype Priority is Any_Priority range
                 Any_Priority' First . . 实现定义的;
subtype Interrupt_Priority is Any_Priority range
                  Priority' Last+1 .. Any_Priority' Last;
Default_Priority : constant Priority :=
                 (Priority' First + Priority' Last)/2;
```

一个整数的范围被分为标准优先级和（更高的）中断优先级两部分。实现必须为System.Priority支持一个至少有30个值的范围，并且至少有一个不同的System.Interrupt_Priority值。

一个任务通过在规格说明中包含一个编用设置它的初始优先级:

```
task Controller is
  pragma Priority(10);
end Controller;
```

如果一个任务类型的定义包含这样一个编用，那么这种类型的所有任务就有同样的优先级，除非在编用中使用一个判别式:

```
task type Servers(Task_Priority : System.Priority) is
```

⊖ 优先级也能分配给入口队列和选择语句的操作。但本节将重点关注任务优先级和保护对象高限优先级。

```
    entry Service1(...);
    entry Service2(...);
    pragma Priority(Task_Priority);
  end Servers;
```

对于作为中断处理程序的实体，定义一个特殊的编用: 505

```
pragma Interrupt_Priority(Expression);
```

或简单地定义为:

```
pragma Interrupt_Priority;
```

按中断级别定义和使用不同的编用改善了程序的可读性，帮助我们排除了由于任务和中断优先级的混淆而导致的错误。然而，因为在Interrupt_Priority中使用的表达式的值可以小到Any_Priority，因此，可能给中断处理程序一个相对低的优先级。如果缺少了该表达式，则分配最高可能的优先级。

使用这些编用中的一个分配的优先级被称作**基础优先级**（base priority）。一个任务也可能有一个更高的**活跃优先级**（active priority）——这将在适当的时候解释。

由一个假定的环境任务执行的主程序可以通过在主子程序里放置Priority编用来设置它的优先级。如果没有这么做，就使用在System中定义的默认值。未能使用编用的其他任务具有默认的基础优先级，该基础优先级等于创建它的任务的基础优先级。

为了使用ICPP，Ada程序必须包含以下编用:

```
pragma Locking_Policy(Ceiling_Locking);
```

实现可以定义其他的锁定策略，Ada的实时系统附件仅需要Ceiling_Locking。如果漏掉了这个编用，则默认策略由实现定义。为了指定每个保护对象的高限优先级，使用上一节定义的Priority和Interrupt_Priority编用。如果漏掉了这个编用，高限被假定为System.Priority' Last。

如果一个任务用比一个保护对象的高限还高的优先级调用这个保护对象，则将引发异常Program_Error。如果允许这种调用，将破坏对该对象的互斥保护。如果是一个中断处理程序用一个不适当的优先级进行调用，那么程序就出错了。通过适当的测试和/或静态程序分析，最终必须预防这种情况的发生。

有了Ceiling_Locking，一个有效的实现将使用调用任务的线程，不仅去执行保护调用的代码，还要执行由原来的调用动作偶然启动的任何其他任务的代码。例如，考虑如下简单的保护对象:

```
protected Gate_Control is
    pragma Priority(28);
    entry Stop_And_Close;
    procedure Open;
private
    Gate: Boolean := False;
end Gate_Control;

protected body Gate_Control is
    entry Stop_And_Close when Gate is          506
    begin
      Gate := False;
    end Stop_And_Close;
```

```
    procedure Open is
    begin
      Gate := True;
    end Open;
  end Gate_Control;
```

假设有一个任务T，优先级为20，它调用Stop_And_Close，并被阻塞。稍后，任务S（优先级27）调用Open。实现S的线程将采取下列动作：

1）为S执行Open的代码。

2）计算入口的屏障，并注意到T现在可以继续前进。

3）为T执行Stop_And_Close代码。

4）再次计算入口的屏障。

5）在调用了保护对象后，S继续执行。

结果，没有上下文切换。另一可选办法是S使T在第（2）点时变成可运行的，现在T有一个比S高的优先级（28），因此系统必须切换到T，从而在Gate_Control内完成它的执行。当T离开后，需要切换回S。这种方法开销更高。

当一个任务进入一个保护对象时，它的优先级可以上升到由Priority或Interrupt_Priority编用定义的基础优先级之上。用于决定分派顺序的优先级是任务的**活跃优先级**。这个活跃优先级是任务的基础优先级和它继承的优先级中的最大值。

保护对象的使用是任务继承较高活跃优先级的方法之一，还有其他方法，例如：

- 在激活期间——一个任务将继承创建它的父任务的活跃优先级，记住父任务这时被阻塞，等待它的子任务的完成，这可能是没有这条继承规则时优先级反转的一个来源。
- 在会合期间——执行接收语句的任务将继承发出入口调用的任务的活跃优先级（如果该优先级较大）。

注意最后一种情况不一定排除优先级反转的所有可能情况。考虑一个服务器任务S，它有入口E和基础优先级L（低的）。一个高优先级任务调用E。一旦会合已经开始，S将以更高的优先级执行。但是在S到达E的接收语句之前，它将以L优先级执行（即使高优先级任务被阻塞）。这个方法和其他优先级继承方法可以被实现支持。然而，这种实现必须提供一个编用，用户

507 能利用它去明确地选择补充的条件。

实时系统附件试图提供灵活和可扩展的特性。很明显，这是不容易的。Ada 83受害于规定太多。但是，缺乏一个确定的分派策略是令人遗憾的，因为它不能对软件开发和可移植性提供帮助。如果基础优先级已经被定义了，那么就假设它采用基于优先级的抢占式调度。在多处理器系统中，由实现定义调度是在每个单处理器的基础上还是跨越整个处理器簇。

为了给出可扩展性，分派策略可以使用以下编用来选择：

```
pragma Task_Dispatching_Policy(Policy_Identifier);
```

实时系统附件试图定义一个可能的策略：FIFO_Within_Priority。这里，任务共享同样的优先级，它们按FIFO顺序排队。因此，当任务变为可运行时，它们被放到该优先级别的假想运行队列的队尾。这种情况的一种例外是当一个任务被抢占时，这时该任务被放在那个优先级别的假想运行队列的前面。

如果一个程序指定了Fifo_Within_Priority选项，那么它也必须选择在前面定义的Ceiling_Locking策略。总之，它们代表了一个一致的和可用的模型，用于建造、实现和

分析实时程序。

其他Ada设施

Ada还提供了其他设施，这些设施对于各种各样的系统的编程是很有用的。例如，动态优先级、有优先级的入口队列、任务属性、异步任务控制设施等等。读者可以参考Ada参考手册的系统编程和实时附件或文献Burns and Wellings（1998），以了解更多细节。

目前，Ada不支持偶发服务器。但是可以参看Harbour等（1998）关于如何近似实现偶发服务器的讨论。

13.14.2 POSIX

POSIX支持基于优先级的调度，可以选择支持优先级继承和高限协议。优先级可以动态设置。在基于优先级的设施方面，有四个策略:

- FIFO——进程/线程运行直到它完成或被阻塞，如果进程/线程被一个更高优先级的进程/线程抢占，那么它被放在它的优先级所在运行队列的头部。
- 轮转（round-robin）——进程/线程运行直到它完成或被阻塞或它的时间片到期，如果进程/线程被更高优先级进程抢占，它被放在它的优先级所在运行队列的队头；但是，如果它的时间片到期，它被放到队尾。 508
- 偶发服务器——进程/线程作为偶发服务器运行（见下面）。
- 其他——由实现定义的策略（必须形成文档）。

对于每一个调度策略，都有一个必须支持的最小优先级范围，对于FIFO和轮转，优先级范围至少是32。调度策略可以按每进程或按每线程为基础进行设置。

线程也可以用“系统争用”方式创建，这时它们根据它们的策略和优先级同其他系统线程竞争。另一种方法是，线程可以用“进程争用”方式创建，在这种情况下，它们必须同父进程中的其他线程（用“进程争用”创建的）竞争。POSIX系统没有规定这些线程相对于其他进程中的线程或者相对于全局争用的线程是如何调度的。特定的实现必须决定是否支持“系统争用”或“进程争用”或两者都支持。

如同在13.8.2节讨论的，偶发服务器分配一个限定量的CPU容量去处理事件，它有一个补充周期、一个预算和两个优先级。当服务器有某些预算剩余时，它运行在高优先级，预算用尽后运行在低优先级。当服务器运行在高优先级时，从预算中减去它消耗的执行时间量。在服务器被激活时以及在补充周期，消耗的预算被补充。当服务器的预算到达0时，它被设为低优先级。

程序13-1说明了C到POSIX调度设施的接口。这些函数被分为两部分，一部分用于操纵进程的调度策略和参数，另一部分用于操纵线程的调度策略和参数。如果一个线程修改了它自己进程的策略和参数，对线程的影响将依赖于它的争用范围（或级别）。如果它正在系统级别上竞争，这种改变将不会影响该线程。但是，如果它正在进程的级别上竞争，将会对该线程有影响。

程序13-1 C到一些POSIX调度设施的典型接口

```
#define SCHED_FIFO ...          /* 抢占优先级调度 */
#define SCHED_RR ...            /* 带时间片的抢占优先级 */
#define SCHED_SPORADIC ...          /* 偶发服务器 */
#define SCHED_OTHER ...                /* 实现定义的调度程序*/
#define PTHREAD_SCOPE_SYSTEM ...     /* 系统级争用 */
```

```
#define PTHREAD_SCOPE_PROCESS ...   /* 局部争用 */
#define PTHREAD_PRIO_NONE ...        /* 无优先级继承*/
#define PTHREAD_PRIO_INHERIT ...     /* 基本优先级继承 */
#define PTHREAD_PRIO_PROTECT ...     /* ICPP */

typedef ... pid_t;
struct sched_param {
  ...
  int sched_priority; /* 用于 SCHED_FIFO 和 SCHED_RR */
  int sched_ss_low_priority
  timespec sched_ss_repl_period
  timespec sched_ss_init_budget
  int  sched_ss_max_repl
  ...};

int sched_setparam (pid_t pid, const struct sched_param *param);
/* 设置进程 pid 的调度参数*/

int sched_getparam(pid_t pid, struct sched_param *param);
/* 获取进程 pid 的调度参数*/

int sched_setscheduler(pid_t pid, int policy,
                       const struct sched_param *param);
/* 设置进程 pid 的调度策略和参数*/

int sched_getscheduler(pid_t pid);
/* 返回进程 pid 的调度策略*/

int sched_yield(void);
/* 将当前线程/进程放在运行队列的队尾*/

int sched_get_priority_max(int policy);
/* 返回指定策略的最大优先级*/

int sched_get_priority_min(int policy);
/* 返回指定策略的最小优先级*/

int sched_rr_get_interval(pid_t pid, struct timespec *t);
/* 如果 pid /= 0, 在由t引用的结构中为调用进程/线程设置时间片 */
/* 如果 pid = 0,在由t指向的结构中为调用进程/线程设置时间片*/

int pthread_attr_setscope (pthread_attr_t *attr,
                           int contentionscope);
/* 为一个线程属性对象设置争用作用域属性 */

int pthread attr_getscope(const pthread_attr_t *attr,
                          int *contentionscope);
/* 获取一线程属性对象的争用作用域 */

int pthread_attr_setschedpolicy(pthread_attr_t *attr,
                                int policy);
/* 为一个线程属性对象设置调度策略属性 */

int pthread_attr_getschedpolicy(const pthread_attr_t *attr,
                                int *policy);
/* 获取一线程属性对象的调度策略属性 */

int pthread_attr setschedparam(pthread_attr_t *attr,
```

```
                                        const struct sched_param *param);
/* 为一个线程属性对象设置调度参数属性 */

int pthread_attr_getschedparam(const pthread_attr_t *attr,
                                        struct sched_param *param);
/* 获取一线程属性对象的调度参数属性 */

int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,
                                           int protocol);
/* 设置优先级继承协议 */
int pthread_mutexattr_getprotocol(pthread_mutexattr_t *attr,
                                           int *protocol);
/* 获取优先级继承协议 */

int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,
                                                int prioceiling);
/* 设置高限优先级 */
int pthread_mutexattr_getprioceiling(pthread_mutexattr_t *attr,
                                              int *prioceiling);
/* 获取高限优先级*/

/* 除 sched_get_priority__max和 sched_get_priority_min外, */
/* 所有上述函数如果成功的话，返回0*/
```

为了预防优先级反转，POSIX允许将继承协议同互斥锁变量相关联。POSIX既支持基本的优先级继承，也支持立即高限优先级协议（在POSIX中叫做优先级保护协议）。

其他POSIX设施

POSIX提供了其他对实时系统有用的设施。例如，它允许:

- 消息队列按优先级排序;
- 有动态获取和设置线程优先级的函数;
- 线程可以指定它的属性是否可被它的子线程继承。

13.14.3 实时Java

实时Java有可调度对象的概念。这个对象可以是支持程序13-2里面给出的Schedulable接口的任何对象。类RealtimeThread、NoHeapRealTimeThread以及类AsyncEventHandler都支持这个接口，这些类的对象都有调度参数（见程序13-3）。实时Java的实现必须至少支持28个实时优先级级别。与Ada语言和POSIX一样，整数值越大，优先级越高（因而有更高的执行资格）。 509~511
非实时线程被给予的优先级在最小的实时优先级之下。注意，线程的调度参数在创建线程时被限定（参看程序12-11）。如果参数被改变了，对关联的线程会有立即的影响。

程序13-2　实时Java的Schedulable接口

```
public interface Schedulable extends java.lang.Runnable
{

  public void addToFeasibility();
  public void removeFromFeasibility();

  public MemoryParameters getMemoryParameters();
  public void setMemoryParameters(MemoryParameters memory);

  public ReleaseParameters getReleaseParameters();
```

```
    public void setReleaseParameters (ReleaseParameters release);

    public SchedulingParameters getSchedulingParameters();
    public void setSchedulingParameters(SchedulingParameters scheduling);

    public Scheduler getScheduler();
    public void setScheduler(Scheduler scheduler);
  }
```

程序13-3 实时Java的SchedulingParameters类及其子类

```
public abstract class SchedulingParameters
{
  public SchedulingParameters();
}

public class PriorityParameters extends SchedulingParameters
{
  public PriorityParameters (int priority);

  public int getPriority();
  public void setPriority(int priority) throws
                         IllegalArgumentException;
  ...
}

public class ImportanceParameters extends PriorityParameters
{
  public ImportanceParameters(int priority, int importance);
  public int getImportance();
  public void setImportance(int importance);
  ...
}
```

与Ada和实时POSIX一样，实时Java支持基于优先级的抢占式分派策略。但是，不同于Ada和实时POSIX的是，实时Java不需要将被抢占线程放在同其优先级相关的运行队列的开头，实时Java也支持一个高级调度程序，它的目的是:

- 根据可利用的资源和可行性算法决定是否允许新的可调度对象进入。
- 根据与可行性算法相关的优先级分配算法来设置可调度对象的优先级。

所以，Ada与实时POSIX关注于静态脱机可调度性分析，而实时Java更多地致力于动态系统的联机分析。

程序13-4中给出了抽象类Scheduler。方法isFeasible只考虑已添加至它的可行性列表中的(通过方法addToFeasibility和removeFromFeasibility)可调度对象的集合。当给定对象的启动和存储参数改变时，方法changeIfFeasible检查它的对象集合是否依然可行，如果可行，参数才能被改变。静态方法允许查询或设置默认的调度程序。

程序13-4 实时Java的类Scheduler

```
public abstract class Scheduler
{
  public Scheduler();
```

```
  protected abstract void addToFeasibility(Schedulable schedulable);
  protected abstract void removeFromFeasibility(Schedulable schedulable);

  public abstract boolean isFeasible();
  //检查当前可调度对象的集合

  public boolean changeIfFeasible(Schedulable schedulable,
                ReleaseParameters release, MemoryParameters memory);

  public static Scheduler getDefaultScheduler();
  public static void setDefaultScheduler(Scheduler scheduler);

  public abstract java.lang.String getPolicyName();
}
```

类Scheduler的一个已定义子类是类PriorityScheduler（在程序13-5中定义），它实现了标准的基于优先级的抢占式调度。 512~513

程序13-5　实时Java的类PriorityScheduler

```
class PriorityScheduler extends Scheduler
{
  public PriorityScheduler()

  protected void addToFeasibility(Schedulable s);
  protected void removeFromFeasibility(Schedulable s);

  public boolean isFeasible();
  //检查当前可调度对象的集合

  public boolean changeIfFeasible(Schedulable Schedulable,
                ReleaseParameters release, MemoryParameters memory);

  protected void addToFeasibility(Schedulable s);
  protected void removeFromFeasibility(Schedulable s);

  public void fireSchedulable(Schedulable Schedulable);

  public int getMaxPriority();
  public int getMinPriority();
  public int getNormPriority();
  public java.lang.String getPolicyName();

  public static PriorityScheduler instance();

  ...
}
```

必须再次强调的是：实时Java不需要实现提供联机的可行性算法。测试可行性的方法简单地返回false就足够了。

程序13-6　支持优先级继承的实时Java类

```
public abstract class MonitorControl
{
  public MonitorControl();

  public static void setMonitorControl(MonitorControl policy);
  // 设置默认值
```

```
  public static void setMonitorControl(java.lang.Object monitor,
                                       MonitorControl policy);
  // 设置单个对象策略
}

public class PriorityCeilingEmulation extends MonitorControl
{
  public PriorityCeilingEmulation(int ceiling);

  public int getDefaultCeiling();
  // 获取此对象的高限
}

public class PriorityInheritance extends MonitorControl
{
  public PriorityInheritance();

  public static PriorityInheritance instance ();
}
```

对优先级继承的支持

当访问同步类时，实时Java允许使用优先级继承算法。为了达到这个目的，定义了三个类（见程序13-6）。考虑下面的类：

```
public class SynchronizeClass
{
  public void Method1() { ... };
  public void Method2() { ... };
}
```

这个类的一个实例能够通过下列代码让它的控制协议设置将立即高限优先级继承（在实时Java里称为高限优先级仿真）的优先级设置为10。

```
SynchronizeClass SC = new SynchronizeClass();
PriorityCeilingEmulation PCI = new PriorityCeilingEmulation(10);
...
MonitorControl.setMonitorControl (SC, PCI);
```

13.14.4 实时Java的其他设施

必须注意，在实时Java里所有的队列是按优先级排序的，这一点很重要。

实时Java中值得一提的一个设施是，它用进程组的形式提供对非周期线程的支持。一组非514~515 周期线程能够被连接在一起，并被赋予有助于可行性分析的特性，例如，能够定义线程在一个给定的周期内能消耗不超过cost的CPU时间。支持进程组的类的完整定义在附录里给出。

小结

一个调度方案有两个方面：它定义一个共享资源的算法，以及一个在使用那种资源共享方式时预测应用程序在最坏情况下行为的手段。

大多数当前的周期性实时系统的实现都使用循环执行方法。基于这一方法，应用程序代码必须被打包成固定数目的“小循环”，这样的小循环序列（称为大循环）的循环执行将使所有的时限得到满足。尽管对小型系统而言它是一个有效的实现策略，但是这种循环执行方法

仍有几个缺陷:

- 随着系统的增长，小循环的打包变得越来越困难。
- 很难容纳偶发活动。
- 长周期（那就是，比大循环还要长）进程不能得到有效支持。
- 有长计算时间的进程必须分解，以便它们能被打包成一系列小循环。
- 循环执行的结构使它很难改变以容纳变化着的需求。

因为这些困难，本章主要关注基于优先级的调度方案的使用。描述了一个简单的基于利用率的测试（它只可应用于受限的进程模型）之后，为一个灵活的模型推导出了响应时间计算方法。这个模型能容纳偶发进程、进程交互、非抢占式的段、启动抖动、非周期服务器、容错系统，以及进程时限（D）和它的最小到达间隔（T）之间的任意关系。

进程之间的同步（例如，互斥访问共享数据），可能导致优先级反转，除非使用某种优先级继承。本章详细描述了两个具体协议：原始高限优先级协议和立即高限优先级协议。

对于基于优先级的调度来说，分配优先级以反映进程加载的时序特性是非常重要的。在本章描述了以下三种算法:

- 对$D=T$，使用速率单调算法
- 对$D<T$，使用时限单调算法
- 对$D>T$，使用任意算法

本章以Ada、实时Java和实时POSIX为例说明如何实现固定优先级调度结束。 516

相关阅读材料

Audsley, N. C., Burns, A., Davis, R., Tindell, K. and Wellings, A. J. (1995) Fixed Priority Preemptive Scheduling: An Historical Perspective. *Real-Time Systems*, 8(3), 173–198.

Buttazzo, G. C. (1997) *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. New York: Kluwer Academic.

Burns, A. and Wellings, A. J. (1995) *Concurrency in Ada*, 2nd edn. Cambridge: Cambridge University Press.

Gallmeister, B. O. (1995), *Programming for the Real World POSIX.4*. Sebastopol, CA: O'Reilly.

Halbwachs, N. (1993) *Synchronous Programming of Reactive Systems*. New York: Kluwer Academic.

Klein, M. H. et al. (1993) *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. New York: Kluwer Academic.

Joseph, M. (ed.) (1996) *Real-time Systems: Specification, Verification and Analysis*. Englewood Cliffs, NJ: Prentice Hall.

Natarajan, S. (ed.) (1995) *Imprecise and Approximate Computation*. New York: Kluwer Academic.

Rajkumar, R. (1993) *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. New York: Kluwer Academic.

Stankovic, J. A. et al. (1998) *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms*. New York: Kluwer Academic.

练习

13.1 有三个逻辑进程P、Q和S，它们分别有下列特性：P：运行周期为3，所需运行时间为1；Q：运行周期为6，所需运行时间为2；S：运行周期为18，所需运行时间为5。

说明如何用速率单调调度算法调度这些进程。

说明如何构造一个循环执行来实现这三个逻辑进程。

13.2 考虑三个进程P、Q和S。P的周期为100ms，所需处理时间为30ms；Q和S的相应值是（5，1）和（25，5）。假设P是本系统中最重要的进程，其次是S，再次是Q。

（1）如果优先级是基于重要性的，调度程序的行为是怎样的？

（2）进程P、Q和S的处理器利用率是多少？

（3）应如何调度这些进程以满足所有的时限？

517 （4）写出一个能调度这些进程的调度方案。

13.3 给上一题的进程集合加入第四个进程（R）。这个进程的运行故障不会导致对系统安全的破坏。R的周期为50ms，但有一个依赖于数据的处理需求，并且处理时间在5～25ms之间变化。论述如何将R进程与P、Q和S集成起来。

13.4 图13-11 给出了4个周期性进程w、x、y、z的行为。这些进程根据速率单调方案确定的优先级为：优先级（w）＞优先级（x）＞优先级（y）＞优先级（z）。

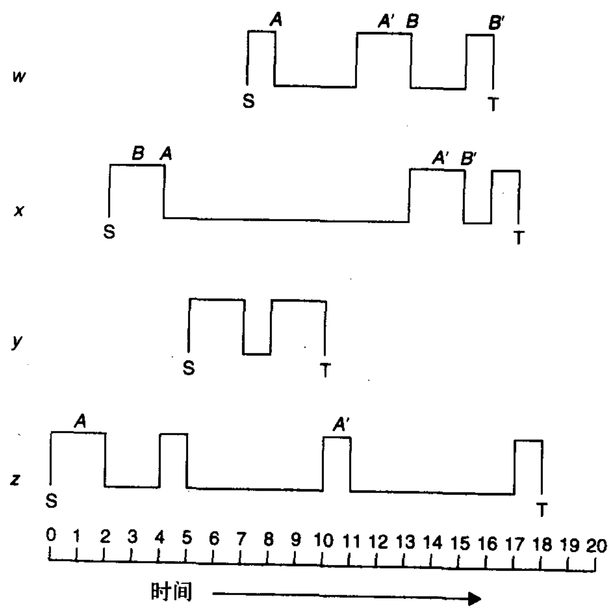


图13-11 练习13.4中四个周期性进程的行为

每个进程的周期都在S时刻开始，T时刻终止。4个进程共享2个被二元信号量A和B保护的资源。图中标记A（和B）意味着“在信号量上执行等待操作”；标记A'（和B'）意味着“在信号量上执行发信号操作”。表13-16概括了4个进程的需求。

图13-11显示了4个进程根据静态优先级执行的历史纪录。例如：x开始于时刻2，时刻3时在B信号量上成功地执行了等待操作，但却在时刻4执行等待信号量A操作失败（z

已经锁住了A)。在时刻13时x再次执行（这次它锁住了A)，它在时刻14时释放了A，在时刻15时释放了B。这时它被进程w抢占，但它在时刻16再次执行，最后在时刻17结束。

表13-16　练习13.4中进程需求概要

进　程	优 先 级	启动时间	所需处理器时间	所用信号量
w	10	7	4	A，B
x	8	2	5	A，B
y	6	5	4	-
z	4	0	5	A

如果使用优先级继承，重画该练习所给的图，显示出这些进程的行为。

13.5 如果使用立即高限优先级继承，重画上一题所给的图，显示出这些进程的行为。

13.6 使用高限优先级协议，可以计算出任一进程被优先级较低的进程所阻塞的最长时间。计算这种阻塞的规则是什么？通过计算下面例子中进程的最长阻塞时间，说明上述的规则。一个程序包含5个进程，a、b、c、d、e（它们按优先级由高到低排列），有6个资源R1，…R6（由实现高限优先级协议的信号量加以保护）。表13-17给出访问这些资源时最坏情况的运行时间。

表13-17　练习13.6中进程的执行需求概要

R1	R2	R3	R4	R5	R6
50ms	150ms	75ms	300ms	250ms	175ms

表13-18给出了进程访问资源的情况。

表13-18　练习13.6中进程的资源需求概要

进　程	使用资源
a	R3
b	R1，R2
c	R3，R4，R5
d	R1，R5，R6
e	R2，R6

13.7 使用式（13-1）给出的简单的基于利用率的测试，表13-19中的进程是可调度的吗？如果使用响应时间分析，该进程集是可调度的吗？ 518~519

表13-19　练习13.7中进程的属性概要

进　程	周　期	执行时间
a	50	10
b	40	10
c	30	9

13.8 使用式（13-1)，表13-20中的进程集不是可调度的，由于进程a是至关重要的，它必须被分配最高的优先级。如何转换这个进程集使它是可调度的呢？注意由a代表的计算仍然必

须给予最高优先级。

表13-20 练习13.8中进程的属性概要

进程	周期	执行时间	重要性
a	60	10	高
b	10	3	低
c	8	2	低

13.9 使用式（13-1），表13-21所给的进程集不是可调度的，但是当使用固定优先级调度时，所有时限都满足，请解释原因。

表13-21 练习13.9中进程的属性概要

进程	周期	执行时间
a	75	35
b	40	10
c	20	5

13.10 在13.8节中，将偶发进程定义为有一个最小到达时间间隔。通常偶发进程的到达都是爆发式的。修改式（13-4）以应付偶发进程的爆发到达，例如有N个调用紧挨着周期T任意出现。

13.11 扩充上述对付偶发进程爆发到达的解决方法，此时在周期T内有N个调用，并且每一个调用被至少M个时间单位隔开。 520

13.12 在本章中给出的响应时间公式能够应用于除CPU之外的哪些资源？例如，该公式能否用于存取磁盘的调度？

13.13 在安全至上的实时系统中，一组进程能用于监视关键的环境事件。通常，在一个事件的发生和输出（对这个事件的响应）之间规定一个时限。描述怎样将周期进程用于监测这样的事件。

13.14 考虑表13-22中所列的所有事件和响应每个事件的计算开销。如果每个事件用一个独立进程来处理（用基于优先级的抢占式调度方法来调度这些进程），描述怎样应用速率单调分析去保证所有的时限得到满足。

表13-22 练习13.14中的事件概要

事件	时限	计算时间
A	36	2
B	24	1
C	10	1
D	48	4
E	12	1

13.15 表13-23中的进程集如何能被最优地调度（使用固定优先级调度）？这个任务集是可调度的吗？

13.16 一个实时系统设计师希望在同一个处理器上，混合运行安全至上的、使命至上的和无关紧要的周期和偶发Ada任务。他（她）使用基于优先级的抢占式调度，已经运用响应时

间分析公式预测所有任务能满足它们的时限。给出原因说明为什么系统在运行时仍然可能错过它的时限。给Ada运行时支持系统提供什么增强设施才能帮助消除这个问题?

表13-23　练习13.15中的任务概要

进　程	T	C	B	D
a	8	4	2	8
b	10	2	2	5
c	30	5	2	30

13.17 说明在Ada中如何实现最早时限优先调度。

13.18 用代码说明Ada怎样支持偶发任务。一个任务怎样才能保护自己，避免执行频率超过最小到达间隔。

13.19 能用Ada实现偶发服务器的哪些方面?

13.20 Ada允许使用下列的包对任务的基础优先级进行动态设置。

```
with Ada.Task_Identification;
with System;
package Ada.Dynamic_Priorities is

  procedure Set_Priority(Priority : System.Any_Priority;
              T : Ada.Task_Identification.Task_Id :=
              Ada.Task_Identification.Current_Task);
    -- 如果 T 是 Null_Task_Id ，引发Program_Error
    -- 如果此任务已终止，无任何作用

  function Get_Priority (T : Ada.Task_Identification.Task_Id :=
           Ada.Task_Identification.Current_Task)
           return System.Any_Priority;
    -- 如果此任务已终止，引发Tasking_Error
    -- 或者如果 T 是 Null_Task_Id， 引发Program_Error
private
  ... -- 此问题不需要
end Ada.Dynamic_Priorities;
```

用这个包说明如何实现一个模式改变协议，该协议要求一组任务必须用单个的原子操作来改变它们的优先级。

13.21 实时POSIX支持动态高限优先级，但是Ada 不支持。解释支持动态高限优先级的赞成与反对的理由。

13.22 实时Java调度程序能使用`ProcessingGroupParameters`支持偶发服务器的哪些方面?

13.23 说明如何才能在实时Java中利用“所有进程的利用率小于100%”的可行性测试来实现最早时限优先调度算法。 521~522

第14章 分布式系统

在过去30年里，微处理器和通信技术的价格持续下降，这使得分布式计算机系统在许多嵌入式应用领域里成为单处理器和集中式系统的可行替代选择。分布式系统的潜在优点有：

- 通过利用并行性改善性能
- 通过利用冗余提高可用性和可靠性
- 将计算能力分散到使用它的地方
- 通过处理器和通信链的添加和增强而逐步增长的便利

本章讨论一些在使用多个处理器实现实时系统时带来的问题。

14.1 分布式系统的定义

针对本章的目的，**分布式计算机系统**被定义成“为了一个共同目的或实现一个共同目标的多个自治处理元素合作的系统”。这个定义宽得足以符合大多数直观概念，而不会详细到物理上的散布情况、通信手段等等的细节。这个定义排除了流水线式处理器和阵列式处理器，因为它们的元素不是自治的；也排除了计算机网络（例如因特网），因为它们的节点工作无共同目的[⊖]。那些被人们认为是微处理器体系结构的大多数应用落入此定义的范围之内——例如指挥控制、银行（和其他面向事务的商业应用）和数据获取。图14-1展示了一个分布式制造系统。 523

即使是现代飞机设计（包括民用和军用），也嵌入了分布式系统。例如，集成模块航空电子设备（Integrated Modular Avionics）（AEEC，1991）通过一个ARINC 629总线允许将一个以上的处理模块互连，如图14-2所示。

通常将分布式系统分为**紧密耦合的**（处理元素或节点访问一个公共存储器）和**松散耦合的**（处理元素或节点不访问一个公共存储器）两种。这种分类的重要意义在于：在紧密耦合的系统中，同步和通信可以通过基于共享变量的技术实现，而在松散耦合系统中，同步和通信最终需要某种形式的消息传递。在松散耦合系统中包含一些自身是紧密耦合系统的节点也是许可的。

本章中，“分布式系统”这个术语是指松散耦合体系结构。此外，一般假定处理器之间是

⊖ 然而，随着通信技术的持续改善，越来越多的交互式网络上的工作将符合分布式系统的这个定义。

全连通的——同消息路由有关的问题不予考虑，对这些问题的全面讨论，见参考文献（Tanenbaum，1998）。此外，假定每个处理器访问它自己的时钟，并且这些时钟是松散同步的（即允许它们之间有一个不超过某个界限的差）。

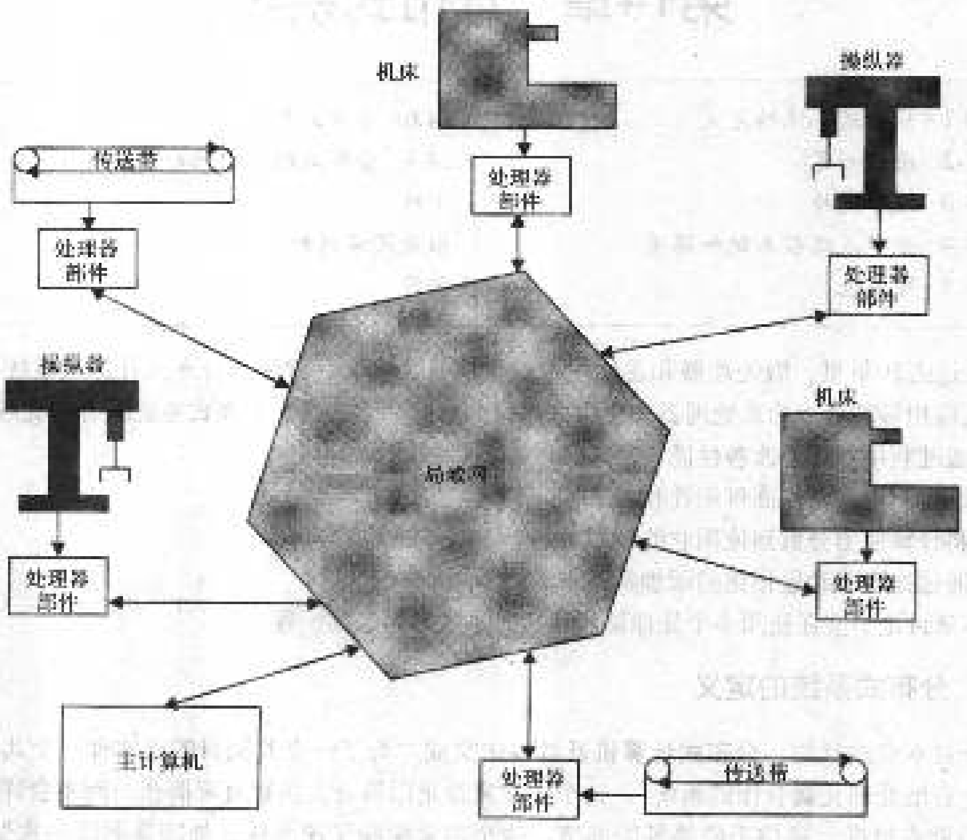


图14-1 分布嵌入式制造系统

图14-2 民用航空电子设备分布嵌入式系统

有一个基于系统中处理器多样性的单独分类。同质系统是所有处理器都是同一类型的系

统；**异质系统**则包含不同类型的处理器。异质系统带来程序和数据的不同表示问题，这些问题虽有重要意义，但在这不予考虑。本章假设所有处理器是同质的。

14.2 论题一览

至此，本书中“并发编程”这个词已用于讨论通信、同步和可靠性，而没有太多涉及过程是如何实现的。然而，在考虑分布式应用时提出的一些论题会引出超出实现细节的根本问题。本章的目的是研究这些论题和它们对实时系统的含义。它们是: 524~525

- **语言支持**——如果语言及其编程环境支持分布式应用的划分、配置、分配和重配置，结合以对远程资源的同位置无关的访问，编写分布式程序的过程就会容易得多。
- **可靠性**——多处理器的可用性使应用变得容许处理器失效——应用应当能够利用这种冗余。虽然多处理器的可用性使应用变得容许处理器失效，也引入了系统中发生更多故障的可能性，而这些故障在集中式单处理器系统中是不会发生的。这些故障同**部分**系统故障相关，应用程序要么必须防止它们，要么能够容许它们。
- **分布式控制算法**——应用中的真正并行性、物理上分布的处理器以及处理器和通信链失效的可能性的出现，意味着资源控制需要许多新算法。例如，可能需要访问存放在其他机器上的文件和数据，另外，机器或网络失效必须不损害这些文件和数据的可用性和一致性。还有，由于在分布式系统内常常没有公共的时间参照系，每个节点有它自己的局部时间，因而很难得到整个系统的一致视图。当试图对分布式数据提供互斥时，这可能会带来问题。
- **时限调度**——在第13章讨论了单处理器系统满足时限的进程调度问题。当进程被分布时，最优单处理器算法不再是最优的，需要新算法。

现在依次讨论这些问题。然而，在一章里难以公平对待这些领域里的所有新活动。

14.3 语言支持

在一个分布式硬件系统上执行的分布式软件系统的生成，包括许多单处理器生成程序时并不需要的步骤:

- **划分**——将系统划分成为部件（分布的单位）的过程，使这些部件适合于安放在目标系统的处理元素上。
- **配置**——将程序划分出来的部件同目标系统的特定处理元素相关联。 526
- **分配**——将已配置系统调整成可执行模块并下载它们到目标系统的处理元素的实际过程。
- **透明执行**——分布式软件的执行，对远程资源能以一种与位置无关的方式访问。
- **重配置**——对软件构件或资源的位置进行动态改变。

大多数明显为分布式编程设计的语言都至少对系统开发中的划分步骤提供语言支持。例如，进程、对象、划分、代理（agent）和守护神（guardian）都被建议作为分布单位。所有这些构造都提供定义良好的接口，使它们能封装局部资源并提供远程访问。有些方法允许将配置信息放进源程序，而另一些方法提供单独的**配置**语言。

分配和重配置一般需要编程支撑环境和操作系统的支持。

或许正是在透明执行方面已进行了大多数的工作以达到跨越各种方法的某个标准化级别。目标是使分布式进程之间的通信尽可能容易和可靠。然而，实际上，通信常常是在跨越不可

靠网络的异质进程之间进行的，并且实际上需要复杂的通信协议（见14.5节）。真正需要的是提供这样的机制：

- 进程不必处理底层形式的消息。例如，它们不必将数据翻译成适合于传输的二进制字符串或将消息分割成信息包。
- 可以假定用户进程接收的所有消息都是完好的。例如，如果将消息分割成了信息包，运行时系统将只在所有的信息包都到达接收节点并能正确装配时才交付它们。此外，如果一个消息中的二进制位已被搅乱，那么这个消息或是不发送，或是在发送前重构；显然，需要一些冗余信息以进行出错检查和改正。
- 进程接收到的消息就是进程预期的消息类型。进程无须进行运行时检查。
- 不限制进程的通信只能使用预定义的内部类型。进程能够使用应用感兴趣的值进行通信。理想的是，如果应用是通过抽象数据类型定义的，那么这种类型的值就可在消息中使用。

可以认为有三种事实上的标准用于分布式程序的相互通信：

527 • 使用一个网络传输协议的应用程序接口（API），例如套接字（socket）
- 使用远程过程调用（remote procedure call，RPC）模式
- 使用分布式对象模式

网络协议问题将在14.5节讨论，Java 同套接字的接口将在14.3.3节简要地讨论，本小节的剩余部分将考虑RPC和分布式对象，包括公共对象请求代理体系结构（Common Object Request Broker Architecture，CORBA）。

14.3.1 远程过程调用

远程过程调用背后被掩盖的目标是使分布式通信尽可能简单。RPC典型的用法是用于同一语言所写程序之间的通信（例如Ada或Java）。一个过程（服务器）被标识为可被远程调用的过程。利用服务器规格说明，有可能能够自动产生两个另外的过程：一个**客户桩**、一个**服务器桩**。在发出远程调用的站点上，客户桩被用于代替服务器。服务器桩和服务器过程在同一站点上使用。这两个过程的目的是以透明的方式提供客户和服务器之间的连接（因而满足上节中列出的所有需求）。图14-3说明了在客户和服务器之间经由两个桩过程的RPC的事件序列。桩有时称为**中间件**，这是因为它们位于应用和操作系统之间。

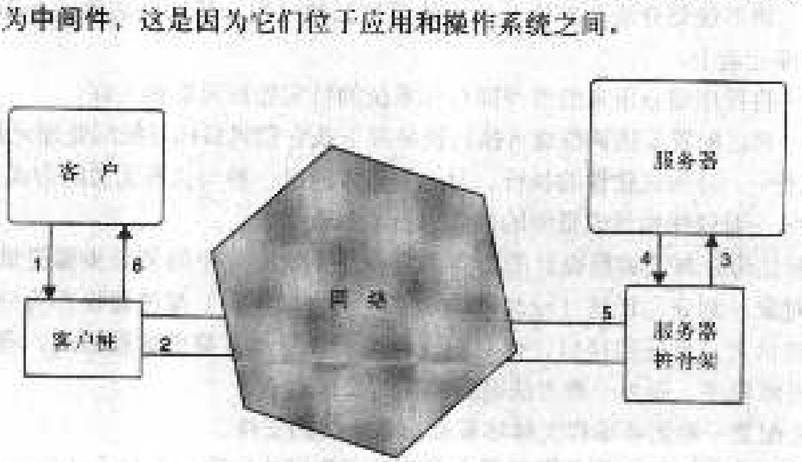


图14-3 RPC中客户和服务器之间的关系

客户桩的作用是:

- 标识服务器（桩）过程的地址;
- 将远程调用的参数转换成适于跨网络传输的字节块——该活动常被称为**参数编组**(parameter marshalling)；
- 向服务器（桩）发送过程执行请求;
- 等待服务器（桩）的回答，并将参数或可能传播来的异常反向编组（unmarshal）；
- 将控制返回给客户过程（连同返回的参数），或在客户过程中引发异常。

服务器桩的作用是:

- 接受来自客户（桩）过程的请求;
- 参数反向编组;
- 调用服务器;
- 捕获由服务器引发的任何异常;
- 编组返回参数（或异常），使之适合跨网络传输;
- 向客户（桩）发送回答。

在客户和服务器过程是用不同语言编写或是在不同机器体系结构的场合，参数的编组和反向编组机制把数据转换成与机器无关和语言无关的格式（见14.4.4节）。

14.3.2 分布式对象模型

过去几年里，在不同场合使用了**分布式对象**（或**远程对象**）这个术语。在最一般的意义下，分布式对象允许:

- 在远程机器上动态创建一个对象（以任何语言）；
- 识别被确定并保存在任何机器上的对象;
- 对象中远程方法的透明调用，就好像它是一个本地方法，并同编写该对象的语言无关;
- 方法调用的跨网络透明运行时分派。

并非所有支持分布式对象的系统都提供支持所有这些功能的机制。如我们在以下各节中所展现的那样:

Ada 支持静态的对象分配，允许远程Ada对象的识别，方便远程方法的透明执行，并支持方法调用的分布式运行时分派; 528~529

Java 允许Java对象的代码跨网络传送、远程创建实例、Java对象的远程指名、它的方法的透明调用和分布式运行时分派;

CORBA 允许在不同机器上以不同语言创建对象，方便远程方法的透明执行，并支持方法调用的分布式运行时分派。

14.4 分布式编程系统和环境

分布式应用的数量巨大，从简单的嵌入式控制系统到大而复杂的多语言通用信息处理平台。全面讨论如何设计和实现这些系统超出了本书范围。然而，本书将研究四种方法:

1）occam2——用于简单嵌入式控制系统;
2）Ada——用于较复杂的分布式实时应用;
3）Java——用于单语言Internet类型的应用;
4）CORBA——用于多语言多平台应用。

14.4.1 occam2

occam2被专门设计得使程序能在多传输机（transputer）网络的分布式环境中执行。通常，occam2的进程不共享变量，所以划分的单元就是进程本身。配置是由PLACED PAR构造实现的。一个作为顶层PAR构建的程序

```
PAR
  P1
  P2
  P3
  P4
  P5
```

可以是分布式的，如下所示：

```
PLACED PAR
  PROCESSOR 1
    p1
  PROCESSOR 2
    PAR
      p2
      p3
  PROCESSOR 3
    PAR
      p4
      p5
```

530 重要的是要注意，从一个简单PAR的程序到一个使用PLACED PAR的程序的变换不会使程序无效。然而，occam2确实允许变量可被同一处理器上的多个进程读。所以，如果程序员使用了这个设施，这种变换就可能就不行了。

对于传输机，必须把每个外部通道同一个合适的传输机链路结合起来。这一点通过PLACE AT构造实现。例如，在图14-4所示的下列整数通道考虑上述例子。

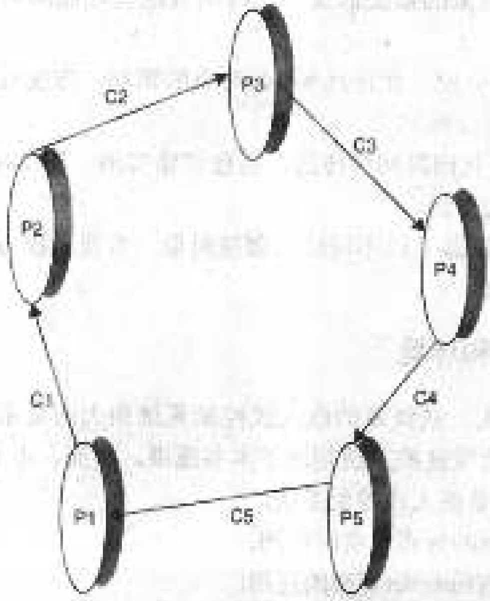


图14-4 由5个通道连接的5个occam2进程

在单个传输机上执行的程序是

```
CHAN OF INT c1, c2, c3, c4, c5:
PAR
  P1
  P2
  P3
  P4
  P5
```

如果将程序配置成三个传输机，如图14-5所示，则occam2程序变成： 531

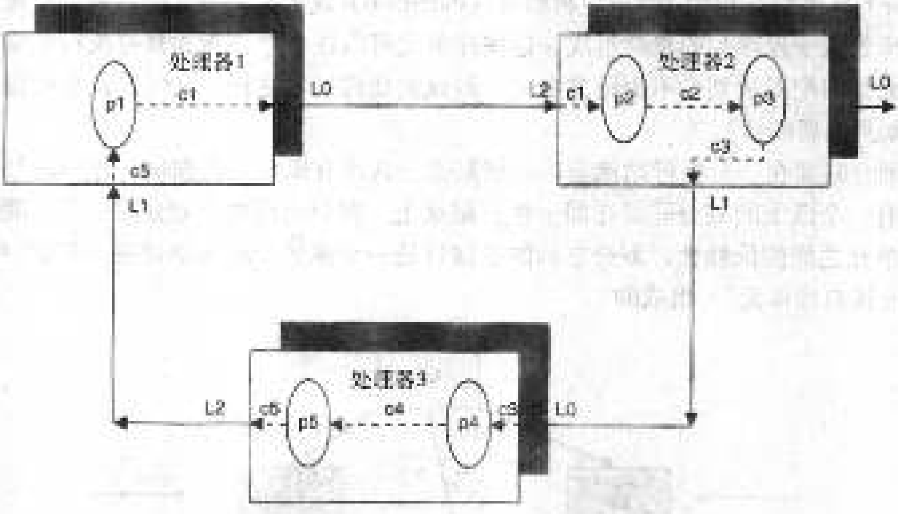


图14-5 由3个传输机配置的5个occam2进程

```
CHAN OF INT c1, c3, c5:
PLACED PAR
  PROCESSOR 1
    PLACE c1 at 0:
    PLACE c5 at 1:
    p1
PROCESSOR 2
  PLACE c1 at 2:
  PLACE c3 at 1:
  CHAN OF INT c2:
  PAR
    p2
    P3
PROCESSOR 3
  PLACE c3 at 0:
  PLACE c5 at 2:
  CHAN OF INT c4:
  PAR
    p4
    p5
```

occam2程序易于配置得能在分布式系统上执行，这是occam2的主要吸引力之一。

532 occam2语言没有定义分配，也没有任何重配置设施。此外，对资源的访问是不透明的。

实时能力

occam2对实时的支持是有限的。然而，在这些限制内，分布式系统模型是一致的。

14.4.2 Ada

Ada将分布式系统定义为：

> 一个或多个处理节点（具有计算和存储能力的系统资源）和零个或多个存储节点（只有存储能力的系统资源，具有可由一个以上处理节点寻址的存储器）的互连。

用于分布式系统编程的Ada模型将划分（Partition）规定为分布单位。划分由库单元（分别编译的库包或子程序）的集群组成，这些库单元可以在一个分布式目标执行环境中共同执行。库单元如何配置成划分不由语言定义，假设实现将提供这种定义（以及分配设施和重配置设施，如果必要的话）。

每个划分驻留在一个执行站点上，在该站点上其所有库单元占据同一逻辑地址空间。然而，可能有一个以上的划分驻留在同一执行站点上。图14-6说明了划分的一个可能结构，箭头表示库单元之间的依赖性。划分之间的主接口是一个或多个包规格说明（在图14-6中被标记为“划分接口库单元”）组成的。

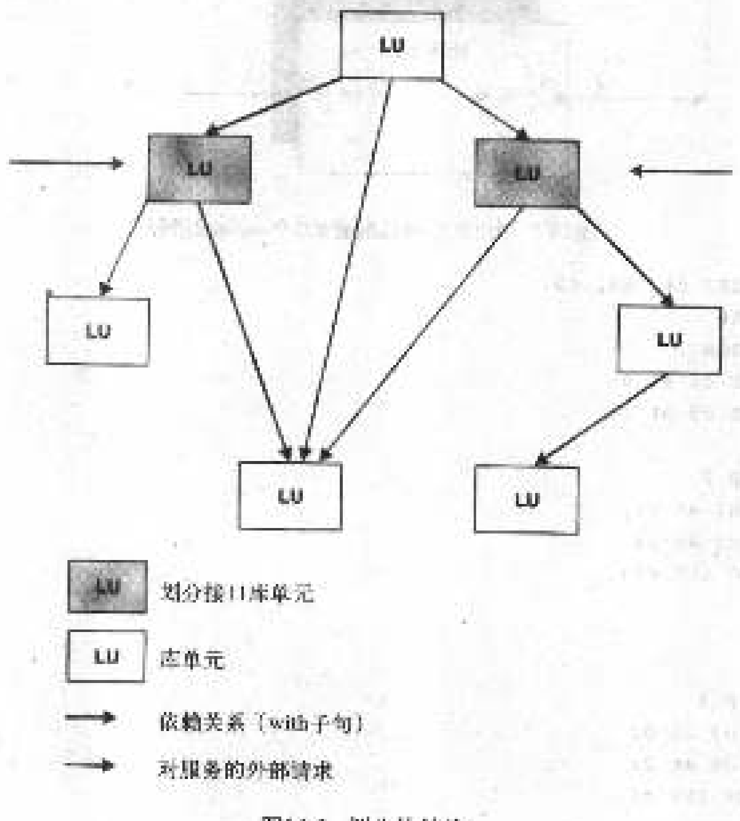


图14-6 划分的结构

划分可以是主动的或被动的。组成主动划分的库单元在同一处理元素上驻留并执行。相反，组成被动划分的库单元驻留在存储元素上，存储元素可由引用它们的不同主动划分节点

直接访问。该模型确保主动划分不能直接访问其他主动划分中的变量。主动划分之间直接共享变量只能通过将变量封装在一个被动划分里实现。主动划分之间的通信由语言定义为经由远程子程序调用（然而，实现可以提供别的通信机制）。

1. 类别化编用

为辅助分布式程序的构造，Ada区分各种库单元类别，并给这些类别加上限制，以维持跨分布式程序的类型一致性。通过下列编用指定这些类别（其中一些本身就很有用，同程序是否是分布式的无关）。

`Preelaborate`（预制作）

预制作库单元是不需要在运行时执行代码就可制作的单元。

`Pure`（纯粹）

纯粹包是加上进一步限制的预制作包，这些限制使它们能自由地在不同的主动划分和被动划分中复制而不会引入任何类型不一致问题。这些限制关注对象和类型的声明，尤其是变量和指名访问类型是不允许的，除非它们是在子程序、任务单元或保护单元里面。 533~534

`Remote_Types`（远程类型）

`Remote_Types`包是一个预制作包，在其可见部分不得包含任何变量声明。

`Shared_Passive`

`Shared_Passive`库单元用于管理主动划分之间共享的全局数据。所以，它们被配置在分布式系统的存储节点上。

`Remote_Call_Interface`

`Remote_Call_Interface`包定义主动划分之间的接口。它的体只存在于一个单独的划分中。所有其他的出现将分配以库桩。

`Remote_Call_Interface`包的规格说明必须是预制作的，除了施加的其他限制外，它不得包含变量的定义（以确保无远程数据访问）。

没有用类别化编用归类的包被称为*正规库包*。如果它被包括在一个以上的划分中，它就被复制，且所有类型和对象被看作是不同的。例如，`Calendar`包就是正规的。

以上编用方便了Ada程序的分布且确保容易识别出不合法的划分（它允许在划分之间进行直接远程变量访问）。

2. 远程通信

主动划分之间进行直接通信的惟一预定义方式是通过远程子程序调用。它们还可通过被动划分中的数据结构间接通信。

调用划分可以用三种不同方式发出远程子程序调用：

- 直接调用已在另一划分的远程调用接口包中声明的子程序
- 通过间接引用一个远程子程序的指针
- 通过对远程对象的方法使用运行时分派

重要的是要注意，在第一种通信中，调用划分和被调用划分是在编译时静态绑定的。然而，对后两种通信，划分是在运行时动态绑定的。所以，Ada支持对资源的透明访问。

许多远程调用只有“in”或“access”参数（即数据只能按调用的方向传送），并且调用者可能希望尽可能快地继续其执行。在这些情况下，有时将这些调用指定为*异步*调用是合适 535

的。过程是被同步地调用还是异步地调用，Ada认为这是过程的性质，而不是调用的性质。这一点是在声明过程时由一个编用Asynchronous指出的。

Ada已经定义怎样划分分布式程序以及必须支持什么形式的远程通信。然而，语言设计者没有对语言做过多的规定，也没有为Ada程序规定一个分布式运行时支持环境。他们希望实现者能提供自己的网络通信协议，并且只要合适，就允许使用其他ISO标准，例如，ISO远程过程调用标准。为实现这些目标，Ada语言假设有一个处理所有远程通信的标准实现提供的子系统（划分通信子系统，PCS）存在。这使编译器能生成对一个标准接口的调用，而无须关心底层实现。

程序14-1定义的包说明了到远程过程（子程序）调用（RPC）支持系统的接口，该支持系统是PCS的一部分。

程序14-1 Ada的包System.RPC

```
with Ada.Streams;
package System.RPC is

  type Partition_ID is range 0..
implementation_defined;

  Communication_Error : exception;

  type Params_Stream_Type ...

  -- 同步调用
  procedure Do_RPC (
          Partition   : in Partition_ID;
          Params      : access Params_Stream_Type;
          Result      : access Params_Stream_Type);

  -- 异步调用
  procedure Do_APC(
          Partition   : in Partition_ID;
          Params      : access Params_Stream_Type);

  -- 到来的 RPC的处理程序
  type RPC_Receiver is access procedure (
          Params      : access Params_Stream_Type;
          Result      : access Params_Stream_Type);

  procedure Establish_RPC_Receiver(Partition : Partition_ID;
          Receiver : in RPC_Receiver);

Private
  ...
end System.RPC;
```

类型Partition_Id是用于标识划分的。对任何库级的声明，D、D' partition_Id产生制作该声明的那个划分的标识符。在远程过程调用期间由System.RPC检测到一个错误时，引发异常Communication_Error。为了在划分之间发送数据，有一个流类型Param_Stream_Type的对象，用于对参数或远程子程序调用的结果进行编组（将数据翻译成适宜的流形式）和反向编组。该对象也用于标识被调用划分中的特定子程序。

在参数被平整成消息之后，调用桩就调用过程Do_RPC。在向远程划分发送该消息后，过程Do_RPC将调用任务挂起直到收到一个回复。过程Do_APC的行为类似于Do_RPC，不同的是，在向远程划分发送该消息后它立即返回。只要为远程调用过程指定了Asynchronous编用，Do_APC就被调用。在制作一个主动划分之后就立即调用Establish_RPC_Receiver，不过是在调用主子程序之前，若有的话。Receiver参数指定一个实现提供的过程，它接受消息，并且调用适当的远程调用接口包和子程序。

3. 实时能力

虽然Ada为单处理器和多处理器系统定义了一个一致的实时模型，它对分布式实时系统的支持是很有限的。在分布式系统附件和实时附件之间没有整合。有争议的是，在这个领域的语言支持技术未被充分承认有标准化的价值。

14.4.3 Java

构造分布式Java应用本质上有两种方式：

1）在独立的机器上执行Java程序，并使用Java网络设施； 536~537

2）使用远程对象。

1. Java网络化问题

14.5节将介绍两个网络通信协议：UDP和TCP。它们是当今在用的重要通信协议，并且Java环境提供了能够访问它们的类（在java.net包中）。到这些协议的API是通过类Socket（对于可靠的TCP协议）和类DatagramSocket（对UDP协议）。详细讨论这个方法超出了本书的范围——请看本章末尾的相关阅读材料，以得到另外的信息源。

2. 远程对象

虽然Java提供了访问网络协议的方便方法，但这些协议还是很复杂的，对于编写分布式应用依然是个障碍。所以，Java通过**远程对象**的概念支持分布式对象通信模型。

Java模型的核心是java.rmi包的使用，这个包建立在TCP协议之上。在这个包中有Remote接口：

```
public interface Remote { };
```

这是编写分布式Java应用的起点。编制该接口的扩展以提供客户和服务器之间的连接。例如，考虑一个服务器，它会返回当地天气预报的详细情况。合适的接口可能是：

```
public interface WeatherForecast extends java.rmi.Remote
  // 客户和服务器之间共享
{
  public Forecast getForecast() throws RemoteException;
}
```

方法getForecast的抛出列表中必须有一个RemoteException类，使得低层实现能够指出远程调用已失败。

Forecast是一个对象，它有今天天气的详细情况。由于该对象会被跨网络复制，所以它必须实现Serializable接口[⊖]。

⊖ 类似于Remote接口，Serializable接口是空的。在这两种情况下，它就作为是给编译器的一个标志信息。对于Serializable接口，它指出对象可被转换为字节流以适于I/O。

```
public class Forecast implements java.io.Serializable
{
  public String Today() {
    String today = "Wet" ;

    return today;
  }
}
```

538

一旦定义了合适的远程接口，就可声明一个服务器类。它又是被用于指出该类的对象可以被远程调用，其常用方式是扩充包java.rmi.server中的一个预定义类。当前，有两个类：RemoteServer（它是由类Remote派生的一个抽象类）和UnicastRemoteObject，它是RemoteServer的一个具体扩充。后者是为非复制服务器提供的类，并使用TCP协议，具有到每个客户的点到点连接。预计对Java的未来扩充可能提供另外一些类，诸如使用一个多路广播通信协议的复制服务器。

下面的例子展示了提供英国Yorkshire郡的天气预报的服务器类：

```
public class YorkshireWeatherForecast extends UnicastRemoteObject
                                       implements WeatherForecast
{
  public YorkshireWeatherForecast()throws RemoteException
  {
    super(); // 调用父构造器
  }
public Forecast getForecast() throws RemoteException
  {
    ...
  }
}
```

一旦写好了这个服务器类，就必须生成服务器桩和每个可被远程调用的方法的客户桩。注意，Java为服务器桩使用**骨架**（skeleton）这个术语。Java编程环境提供一个叫做“rmic”的工具，它取来服务器类，并自动生成适当的客户桩和服务器骨架。

对客户来说，现在需要的就是能够获取一个能访问服务器对象的客户桩。这通过注册实现。注册是一个单独的Java程序，它在每个有服务器对象的宿主机器上执行。它监听标准的TCP端口，并提供在程序14-2中所摘记的类Naming的一个对象。

程序14-2 Java类Naming的摘要

```
public final class Naming
{
  public static void bind (String name, Remote obj)
    throws AlreadyBoundException, java.net.MalformedURLException,
           UnknownHostException, RemoteException;
  // bind the name to the obj
  // name takes the form of a URL such as
  //           rmi://remoteHost:pot/objectName

  public static Remote lookup (String name)
    throws NotBoundException, java.net.MalformedURLException,
           UnknownHostException, RemoteException;
```

```
// looks up the name in the registry and returns a remote object
...
}
```

每个服务器对象可利用类Naming将其远程对象和一个名字绑定起来。这样，客户可访问一个远程注册以获取到该远程对象的一个引用。当然，这是到该服务器的一个客户桩对象的引用。客户桩被加载到客户的机器上，一旦获取了该引用，就可调用服务器的方法了。

3. 实时能力

上面描述的设施是标准Java的设施，它们不是打算在实时约束下操作的。实时Java当前对分布式实时编程没有提供支持。然而，这个问题是今后几年要致力研究的课题。

14.4.4 CORBA

公共对象请求代理体系结构（CORBA）提供最通用的分布式对象模型。其目标是便于为来自不同供货商的中间件软件支持的不同平台、用不同语言编制的应用之间的互操作性。它是由对象管理组织（Object Management Group，OMG）——软件供货商、开发者和终端用户的合伙组织——设计的，遵循图14-7所画的对象管理体系结构模型。

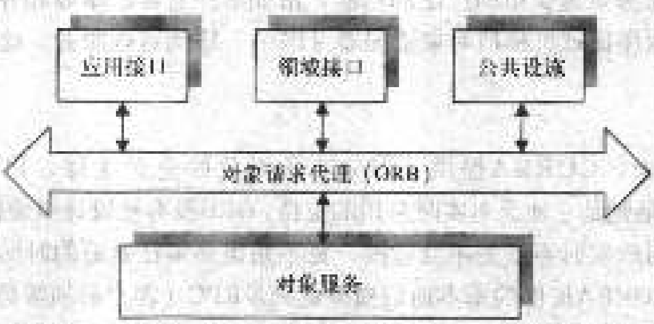


图14-7 对象管理体系结构模型

该体系结构的核心是**对象请求代理**（Object Request Broker，ORB）。这是一个软件通信总线，是提供便于异质应用之间互操作性的主要基础设施。CORBA经常是指ORB。该体系结构的其他成分有：

对象服务——支持ORB的基本服务集合，例如，支持对象创建、Ｉ名和访问控制以及追踪重定位对象。

公共设施——跨范围宽广的应用领域的一个公共功能集合，例如，用户界面、文档和数据库管理。

领域接口——一组接口，它们支持诸如银行和金融或电信这样的特定应用领域。

应用接口——终端用户的专门应用接口。

为确保来自不同供货商的ORB之间的互操作性，CORBA定义了一个通用的Inter-ORB协议，它位于TCP/IP之上（见14.5.2）。

编写CORBA应用的中心问题是**接口定义语言**（Interface Definition Language，IDL）。CORBA接口类似于上节讨论过的Java远程接口。除了描述对象属性外，IDL（类似于C++语言）用于描述由应用对象提供的设施以及传送给一个给定方法的参数及其返回值。上节给出的天气预报应用的接口例子如下： 539/540

```
interface WeatherForecast{
  void GetForecast(out Forecast today);
}
```

一旦定义了应用的IDL，就要有用于“编译它”的工具。IDL编译器产生现有的几个编程语言之一（诸如Java或Ada）的若干新文件。文件包括：

- 客户桩，它们提供客户和ORB之间的通信通道；
- 使ORB能调用服务器上的函数的一个服务器骨架。

现在可以编写服务器和客户代码了。服务器包括两个部分：应用对象自身的代码和服务器进程的代码。应用对象必须同服务器骨架关联。实现方式依赖于目标语言。例如，对Java，通过产生一个类可以做到这一点，这个类是作为所生成服务器骨架的子类。应用对象的代码就这样完成了。服务器进程本身可以写成一个主程序，它创建应用对象，初始化ORB，并告
541 诉它该对象已准备好接受客户请求。客户结构是类似的。

实际上，CORBA提供了较以上描述丰富得多的设施。除了客户和服务器的静态关联之外，CORBA客户可以动态地发现一个服务器的IDL接口，而不需要有关服务器细节的先验知识。此外，支持ORB的服务可提供范围广泛的功能，诸如事件服务、事务和并发控制、持久对象和交易服务。应用程序通过可移植对象适配器（POA）访问这些服务。这是为服务器对象提供运行时环境的库。

1. 实时能力

虽然，一般说来，CORBA提供了对分布式对象的全面支持，但从本书的角度看，CORBA的一个主要限制是它缺乏对实时应用的支持。ORB没有被设计得使操作服从时间约束，结果，有许多操作对软实时系统是不合适的，更不用提必须在紧迫的时间约束之下进行操作的系统。此外，由CORBA提供的基本通信模型是同步RPC（客户必须等待服务器的回复）和延迟同步RPC（客户线程继续执行，并在以后轮询回复）。如在第13章所说的，同步通信模型的时间性质较异步模型更难以分析，并且结果更不如意。

由于这些原因，以及在实时应用中更多地使用CORBA，OMG在过去几年里致力于同主要CORBA标准相关联的性能问题。他们从三个方面处理这些问题：

1）最小CORBA

2）CORBA消息通信（CORBA Messaging）

3）实时CORBA

最小CORBA是CORBA2.2规范的一个子集，它删除了许多服务，包括所有同客户和服务器对象之间的动态配置相关的服务。该子集定位于具有有限资源的嵌入式系统。

CORBA消息通信是CORBA标准的一个扩充，支持异步消息传送和服务质量参数。

实时CORBA（RT CORBA）（Object Management Group，1999）规范定义支持分布式CORBA应用可预测性的机制。它假定实时行为是通过使用基于固定优先级的调度获得的，并因而同大多数实时操作系统兼容——尤其是同那些支持实时POSIX接口的实时操作系统兼容。RT CORBA的关键方面是使应用能够配置和控制处理器和通信这两个方面的资源。现在简要
542 描述这些设施。

2. 管理处理器资源

RT CORBA规范使客户/服务器应用能够管理以下性质：

- 服务器处理客户请求的优先级。RT CORBA规定全局CORBA优先级和将这些优先级映射到宿主RT ORB的特定实时操作系统的优先级范围。使用这些映射，支持三个全局优先级模型：（1）服务器声明模型，由服务器规定它所服务的请求的优先级；（2）客户传播模型，由客户规定优先级并传播给服务器；（3）优先级变换模型，请求的优先级是基于诸如当前服务器负载或全局调度服务状态等外部因素的。
- 服务器上多线程的数量等级。RT CORBA使用线程池概念控制多线程的数量等级。通过使用一个实时POA，服务器应用可以规定：初始创建程的默认数目、可被动态创建的最多线程数和所有线程的默认优先级。这些线程是从一个线程池分配的，它们可能被、也可能不被不同的应用共享。进一步的灵活性由带泳道的线程池概念给出。这时，不仅能控制并发性的总量，还能控制在一个特定优先级上所完成的工作量。
- 优先级继承和优先级高限协议的影响。RT CORBA定义类似POSIX的互斥锁以确保封装共享资源的同步协议的一致性。

3. *管理网络资源*

虽然对资源的透明访问使编写通用的分布式应用变得容易了，却使实时性能的实际分析成为不可能。因此，RT CORBA允许建立起客户和服务器之间的显式连接，例如，在系统配置时建立这种连接。也可以控制客户请求是如何在这些连接上发送的。RT CORBA方便客户和服务器之间的多重连接，以减少由于使用非实时传输协议产生的优先级反转问题。此外，也支持私有传输连接——使客户对服务器的调用能够进行，而不用担心此调用同在此连接上的其他多重调用进行竞争。

即使有上述支持，如果能够设置在Inter-ORB通信协议底层的具体通信协议的服务质量参数也是有利的。RT CORBA提供从客户和服务器两端选择和配置这些属性的接口。 543

4. *调度服务*

由上述讨论应当清楚看到RT CORBA提供许多设施以方便实时分布式CORBA应用。然而，设置全部所需参数（因而得到一个一致的调度策略）是困难的。因此，RT CORBA使应用能够根据它的周期、最坏执行时间、其临界状态等特性的规定调度需求。这是离线完成的，并且每个由应用调度的实体（称为活动）被分配一个文本名字。在运行时刻提供一些接口，这些接口使应用能通过*调度服务*去调度已命名的活动。该服务设置所有必需的优先级参数，以实现诸如时限或速率单调调度等等的特定调度策略。

14.5 可靠性

说分布能提供使系统更加可靠的手段，又同时说分布提供的方法在系统中引入了更多的潜在失效，这看起来几乎是相互矛盾的。虽然多处理器的可用性使应用变得容许处理器失效，它也引入了发生在集中式单处理器系统中不会出现的故障的可能性。尤其是多处理器引入了部分系统失效的概念。在单处理器系统中，如果处理器或存储器失效，则通常整个系统失效。（有时，处理器能够在部分存储器失效的时候继续并恢复，但一般说来系统要崩溃）。然而，在分布式系统中，单个处理器失效而其他处理器继续执行是可能的。此外，经由基础通信网络的传播延迟是可变的，并且消息可取各种路由。这一点连同不可靠的传输媒介，可能导致消息丢失、破坏或以不同于发送次序的次序交付。容许这种失效所需的软件的复杂性增加，也能冲击系统的可靠性。

14.5.1 开放系统互连

在网络和分布式系统的通信协议方面已经进行了大量工作，论述其细节已超出本书范围，读者应参考本节末尾的相关阅读材料。一般说，通信协议是分层的，以方便其设计与实现。然而，有许多不同的网络存在，每个网络都有自己的“网络体系结构”概念和相关的通信协议。所以，如果没有某种国际标准，要想组成不同来源的互连系统是极端困难的。国际标准化组织（ISO）已经定义了一些标准，涉及*开放系统互连*（Open Systems Interconnections）或OSI概念。术语“开放”用于指出：通过遵循这些标准，一个系统对于世界上其他也遵循同一
544 标准的所有系统都是开放的。这些标准已经变成熟知的*OSI参考模型*。应当强调，这个模型并不关心计算机通信网络的具体应用，而是关心提供可靠的、与制造厂家无关的通信服务所需的通信协议的结构。OSI参考模型是个分层模型，层次如图14-8所示。

分层的基本思想是：为了给较高层呈现服务，每一层都加上由较低层提供的服务。从上面看一个特定层，它下面的各个层可被看作是实现一种服务的黑箱。一个层使用由下层提供的服务的方式是通过那一层的接口。接口定义了跨越相邻层之间的边界进行信息交换的规则和格式。实现层的模块通常被称为**实体**。

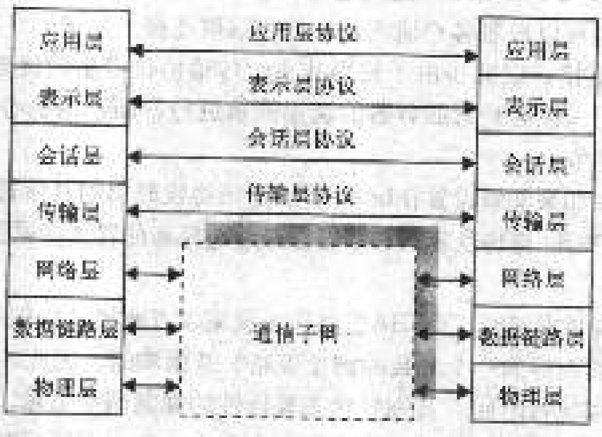


图14-8 OSI参考模型

在网络和分布式系统中，每一层都可被分布到多台机器上，为提供它的服务，同一层在不同机器上的实体可能需要交换信息。这种实体被称为**同行实体**（peer entity）。**协议就是一个管理同行实体之间通信的规则集合。**

OSI模型本身并不定义协议标准，通过将网络功能分解到层，它确实建议应当在什么地方建立协议标准，但这些标准在这个模型本身之外，然而，这些标准已经开发出来了。

现在简要描述每一层的功能。

1）**物理层**

物理层关心的是在信道上传输原始数据，其工作是确保在没有出错的情况下，一端发送
545 了一个“1”，就接受一个“1”，而不是一个“0”。

2）**数据链路层**

数据链路层将一个不可靠的传输通道转换成一个可靠的传输通道以供网络层使用。它还负责消解与同一个传输通道相连的节点之间对传输通道进行访问的可能竞争。

为使数据链路层提供可靠的信道，它必须能校正错误。使用了两种基本又熟悉的技术：*向前出错控制*和*向后出错控制*。向前出错控制需要在每条消息中有足够的冗余信息，以校正在其传输中可能发生的错误。一般说，所需冗余量会随着信息位位数的增长而快速增长。向后出错控制只需要检测到这些错误，一旦检测到了，可利用再传输方案，以得到正确的消息（这是数据链路层的工作）。向后出错控制在网络和分布式系统领域中占优势。

大多数向后出错控制技术加入了计算*校验和*（checksum）的概念，校检和连同消息一起发送并描述该消息的内容。在接收方，校检和被再次计算，并同发送的那个校检和进行比较。它们不一致就表示发生了传输错误。关于这一点，数据链路层提供三类基本的服务：无确认的无连接服务，有确认的无连接服务或面向连接的服务。对第一种，不提供进一步的服务。发送者不能察觉消息未被完整无损地收到。对第二种，一旦消息被正确收到时，就通知发送者，在某个时刻段里没有收到通知消息就表示发生了错误。第三种服务类型建立一种发送者和接收者之间的连接，并保证所有消息都正确收到并按正确次序收到。

3）**网络层**

网络层（或通信子网层）关注从传输层来的信息怎样经由通信子网发送到目的地。消息被分割成信包（Packet），这些信包可能经由不同路径发送，网络层必须把这些信包重新组装，并处理可能发生的阻塞。关于网络层是否应当通过网络提供一个完整的信道，没有明确的一致意见。这种服务的提供有两个极端：*虚拟线路*（virtual circuit）（面向连接）和*数据报*（datagram）（无连接）。对于虚拟线路服务，提供完整的信道。所有消息信包依次到达。对于数据报服务，网络层试图孤立地发送每个信包。所以，消息可能不按次序到达或完全不到达。

物理层、数据链路层和网络层是依赖于网络的，并且其操作细节可能因网络的不同而千差万别。

4）**传输层**

传输层（或宿主到宿主层）提供可靠的宿主到宿主通信给会话层使用。它必须向会话层隐藏通信子网的所有细节，这样使得一个子网可由另一子网代替。实际上，传输层将网络的顾客部分（第5～7层）同运营商部分（第1～3层）屏蔽开来。 546

5）**会话层**

会话层的作用是使用传输层的设施在两个应用级进程之间提供一条通信路径。用户之间的连接通常被称为会话，并可能包括一个远程登录或文件传送。设置会话（叫绑定）所涉及的操作包括权限识别和记账。一旦开始了一次会话，这个层就必须控制数据交换，并使两个进程之间的数据操作同步。

6）**表示层**

表示层完成数据上的常用变换，以克服有关数据表示对应用的异质问题。例如，它使交互式程序能在一组不兼容的显示终端之间转换。它可以进行文本压缩或加密。

7）**应用层**

应用层提供网络的高级功能，诸如对数据库、邮件系统等等的访问。应用的选择可以支配由较低层提供的服务。所以，特定应用领域可以规定一组贯穿所有七层的协议，它们是支持所需的分布式处理功能所要求的。例如，通用汽车公司的一个倡议就定义了一组协议以实现自动制造工厂的开放互连。这些协议被称为*制造自动化协议*（manufacturing automation protocol，MAP）。

14.5.2 TCP/IP层

OSI模型现在有些过时，并且不直接面对Internet工作的问题。TCP/IP参考模型只有5层，如图14-9所示。

应用层
传输层
Internet层
网络接口层
物理层

图14-9　TCP/IP参考模型

物理层等价于OSI物理层。网络接口层完成与OSI数据链路层相同的功能。Internet层规定跨互联网发送的信息包的格式，并完成所有同OSI网络层相关的路由功能，就在这里定义了Internet协议（IP）。

传输层等价于OSI传输层，它提供两个协议：用户数据协议（UDP）和传输控制协议（TCP）。UDP提供不可靠的无连接协议，它使得能在下层高效访问IP协议。TCP协议提供可靠的端到端的字节流协议。

14.5.3 轻量级协议和局域网

OSI和TCP/IP模型最初是为广域网开发的，使之能开放式访问，广域网的特征是低带宽通信、高出错率。大多数分布嵌入式系统会使用局域网技术，并且对外部世界是关闭的。局域网的特征是高带宽通信、低出错率。它们可以使用像广播技术（如以太网那样）或点对点基于交换的技术（例如，ATM（异步传输模式））。所以，虽然有可能用OSI或TCP/IP方法（例如，Java RMI）实现语言级进程间通信（比如Java RMI），而实际上，花销禁止这样做。因此，许多设计人员是按语言（应用）的需求和通信介质裁剪协议。这称为*轻量级协议*。设计中的关键问题是它们容许通信失败的程度。

乍看起来，似乎对于编制高效、可靠的分布式应用而言，根本的是要有完全可靠的通信。然而，事情并不总是这样。考虑一下分布式应用能引入的出错类型。如果两个分布式进程正在为提供服务而通信和同步，可能的错误会来自：

- 由物理通信介质的干预带来的瞬时错误
- 负责屏蔽通信子系统中的瞬时错误的软件的设计错误
- 在服务器进程和在提供服务中所需的任何其他服务器之间协议的设计错误
- 两个服务器进程自身之间协议的设计错误

547~548 为防止发生后一种错误，服务器进程必须提供应用级检查（端对端检查）。系统设计的端对端论点（Saltzer等，1984）说：有了对可靠服务准备的这种检查，就不再需要在协议层次的较低层重复这些检查，特别是在通信介质（例如像以太网或令牌环这样的局域网）提供低出错率（但不完备）传输设施的时候。对于这些情况，有一个高速而可靠性低于100%的通信设施要比一个低速、可靠性为100%的设施好。需要高可靠性的应用可以在应用层以效率换可靠性。然而，需要高速（但不必可靠）服务的应用就不能利用其他方法以可靠性换效率。

对于局域网通信协议有一些标准，特别是在数据链路层，这些标准被分成了两个子层：介质访问控制（Medium Access Control，MAC）和逻辑链路控制（Logical Link Control，LLC）。MAC关注物理通信介质的接口，CSMA/CD（Carrier Sense Multiple Access with Collision Detection，载波侦听多路访问/冲突检测）总线（例如以太网）、信息包交换（例如ATM），令牌总线和令牌环都有标准。LLC关注提供无连接或面向连接的协议。

如早先描述过的，远程过程调用（ISO/IEC JTC1/SC21/WG8，1992）是一个公共的、面向语言的轻量级协议。通常它被直接实现在由局域网提供的基本通信设施之上（例如，LLC层）。对于像Java和Ada这样的语言，在没有机器失效的情况，远程过程调用被认为是可靠的。那就是说，对每个远程过程调用，如果调用返回了，然后过程被执行一次且仅仅一次，这被称为**恰好一次**RPC语义。然而，在机器失效的情况下，这是难以实现的，因为过程可能被部分或全部执行了若干次，这依赖于在什么地方发生崩溃和程序是否重新启动。Ada假设调用被执行*至多一次*，因为在Ada程序失败之后没有重新启动部分Ada程序的概念。

对于实时局域网及其相关协议，重要的是在消息传输中提供有界的和已知的延迟。14.7.2将再次讨论这个议题。

14.5.4 组通信协议

远程过程调用（或远程方法调用）是分布式系统中客户和服务器之间的常见通信形式。然而，它限制通信必须在两个进程之间进行。常常有一组进程进行交互的情况（例如，完成一个原子动作），需要把通信发送给整个组。**多路广播**通信模式就提供这样的设施。一些网络，例如以太网，提供硬件多路广播机制作为其数据链路层的一部分。否则，就得加上进一步的软件协议。

所有网络和处理器都或多或少地不可靠。所以，可以设计一个组通信协议族，其中每一个都提供有特殊保证的多路广播通信设施： 549

- **不可靠多路广播**——不提供交付到组的保证，多重广播协议提供数据报级服务的等价物。
- **可靠多重广播**——协议对于将消息交付到组做出最努力的尝试，但不提供交付的绝对保证。
- **原子多路广播**——协议保证如果组中的某个进程收到了消息，那么组中的所有成员都收到消息，所以消息要么交付到全组，要么没有交付给组中的任何成员。
- **有序原子多路广播**——除了保证多路广播的原子性之外，协议还保证组中的所有成员都以同样次序接收来自不同发送者的消息。

协议给出的保证越多，它们实现的开销就越大。再者，开销还因使用的失效模型而异（见5.2节）。

对于原子和有序原子多路广播，重要的是限制实现算法终止的时间。没有这一点，就不可能预期它们在硬实时系统中的性能。

现在考虑一个简单的有序原子多路广播，其失效模型是：

- 处理器安静地失效
- 所有通信失效都是失效不作为
- 不发生多于N个的相邻的网络失效不作为
- 网络是全连通的，没有网络分组

为实现原子消息传输，需要的就只是传输每个消息$N+1$次，这叫做消息**漫射**（diffusion）。为了通过漫射实现有序性质，需要知道完成每个消息传输所需的时间。假定所有消息的最坏传输值是T_D，而网络中的时钟松散同步的最大差值是C_Δ。每个消息由其发送者打上时间戳，其值为其当地时钟的值（C_{sender}）。每个接受者能在当地时钟大于$C_{sender}+T_D+C_\Delta$时将消息交付给它的进程，因为到了这个时间，所有接受者都被保证已经收到消息，而且消息是**有效的**。这些消息可按其生效时间排序。如果两个消息有相同的生效时间，可给它们以任意的次序（例如，使用处理器的网络地址）。图14-10针对$N=1$说明了这个方法。

注意，虽然上述简单算法保证所有处理器以相同次序收到并处理消息，却不保证这个次序就是消息发送的实际次序。这是因为用于确定次序的值是基于处理器的局部时钟，它们可能差C_Δ。

14.5.5 处理器失效

第5章说明过提供容错硬件和软件的两个一般方法，即静态冗余法和动态冗余法。虽然硬件容错确实在处理器和通信失效时能在实现可靠性方面起主要作用，但需要过量的硬件去实现三模冗余，所以是昂贵的（但是快）。所以，本节集中讨论通过软件方法提供容错。

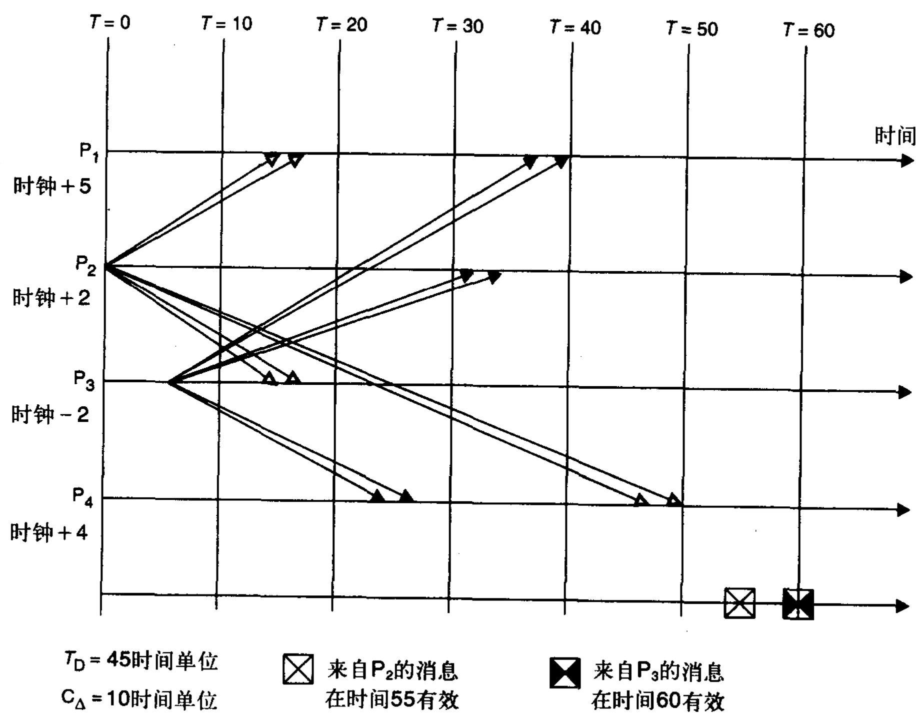


图14-10 一个简单的基于漫散的有序原子多路广播

550 ≀ 551 就现在而言，假设系统中的所有处理器都是**寂静失效**的。这意味着，如果处理器以任何方式失灵，它将蒙受一个永久的不作为失效。如果处理器不是寂静失效，那么它们可能互相发送无效消息。这会对下面的讨论引入额外的复杂性。

1. 通过静态冗余容许处理器失效

在第5章里实现静态容许设计错误的背景下讨论过N版本编程。显然，如果N版本程序的

每个版本都驻留在不同的处理器上，那么这种方法也容许处理器失效。然而，即使不利用设计多样性，对某些系统部件的全同复制以得到所需的可用性也是令人满意的。这常被称为**主动复制**（active replication）。

假设一个应用是按分布式对象模型设计的，能够将对象复制到不同的处理器上，甚至可按具体对象的重要性变化复制的程度。对于那些对应用程序员是透明复制的对象，它们必须有确定性行为，这意味着对对象的给定请求序列，对象的行为是可预测的。如果不然，在一个复制对象集合中的每个复制对象的状态会以不同的方式结束。这样，对那个对象集合发出的任何请求可能产生有变化范围的结果。所以，对象集合必须保持一致，它的成员不得各行其是。

如果要复制对象，就必须也复制每个远程方法调用。而且，它必须有恰好一次的RPC语义。如在图14-11中说明的，每个客户对象将潜在地执行一个一对多过程调用，而每个服务器过程将接受一个多对一请求。运行时系统负责协调这些调用并确保所需语义。这就需要用专门调用周期性地探察服务器站点，以确定它们的状态，响应失败将指出一个处理器崩溃。事实上，运行时系统必须支持某种形式的成员关系协议和一个有序原子多路广播组通信协议。

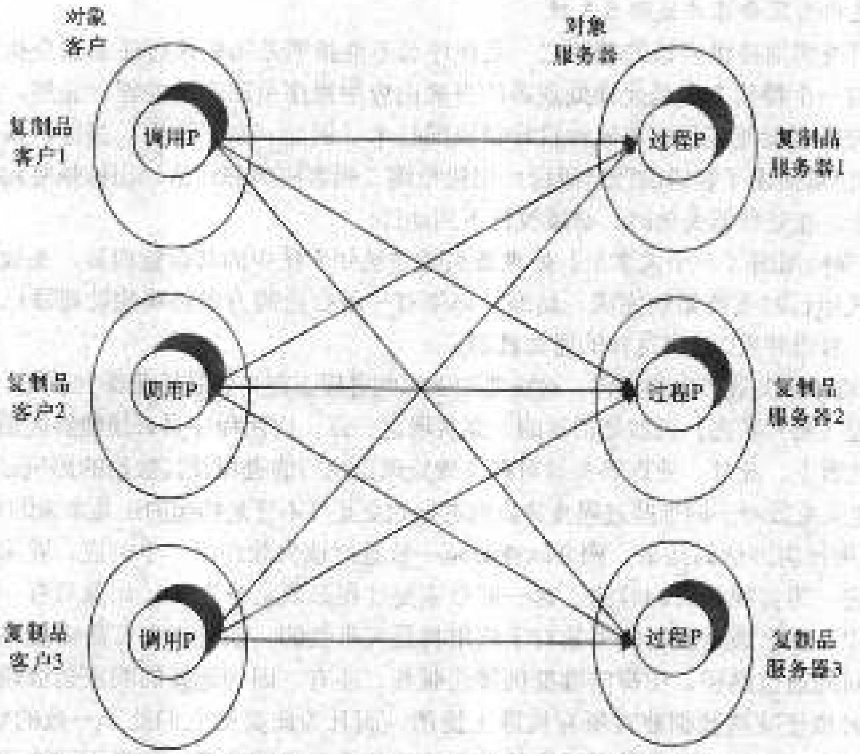


图14-11 复制的远程过程调用

显式地允许复制（进程级）的语言例子是容错并发C（Fault Tolerant Concurrent C）（Cmelik等，1998）。该语言假设处理器有一个失效-停止模型（fail-stop model），并提供分布式一致协议，以确保所有复制品以确定性方式行事。容错并发C有一个非常类似于Ada的通信和同步模型，因而它必须确保：如果选择语句的一个分支访问一个复制品，那么这个分支就

访问所有的复制品（即使这些复制品并不是在上锁步骤上执行）。也曾经有使Ada容错的研究工作，例如（Wellings and Burns，1996；Wolf，1998）。

虽然对象的透明复制方法对于提供容许处理器失效能力是有吸引力的，然而，底层多路广播和一致性协议的开销却禁止硬实时系统这样做。尤其是在对象是主动的且包括多于一个任务（可能有嵌套任务）的场合，运行时一致性协议（确保复制品之间的一致性）必须在每次调度决策时都要执行。用定期状态保存提供**暖**（warm）备份可能提供更便宜的解决方法。用这种方法，对象可被复制到多个处理器上。然而，与完全复制不同的是，在任何时候只有一个拷贝是活动的。该对象的状态被定期保存（一般是在同另一对象通信之前和之后）到驻留复制品的处理器上。如果底层节点失效，就可以从最后一个检查点重新启动一个备份。

在分配给对象使用的处理器资源的高效使用和失效恢复时间之间显然有一个平衡问题。主动复制在处理器资源方面较贵，但只需要较少的恢复时间；对比之下，暖复制（常被称为**被动**复制）较便宜，但恢复得较慢。这导致一种名为**先行者－追随者**（leader-follower）复制的折衷方案（Barrett等，1990），在这个复制方案里，所有复制品是主动的，但有一个被认为是基本版本，并做出所有可能是不确定性的决策。将这些决策告诉所有追随者。这种复制形式比主动复制便宜，也不像被动复制的恢复时间那样长。

2. 通过动态冗余容许处理器失效

552～553 向应用透明地提供容错的问题之一是程序员不能指明是降级执行还是安全执行。静态冗余和复制的一个替代方案是允许处理器的失效由应用程序员动态地处理。显然，本书到目前为止讨论的所有能使应用容许软件设计错误的技术（例如，原子动作）都提供某种程度的硬件容错。第5章描述了容错的四个阶段：出错检测、损害隔离和评估、出错恢复和继续服务中的故障处理。在处理器失效时，必须执行下列动作：

1）必须检测出（一个或多个）处理器失效并通知系统中的其余处理器。失效检测通常由底层分布式运行时支持系统完成。然而，必须有一种合适的方式将哪些处理器已经失效告诉应用软件。容错并发C提供这样的通知机制。

2）必须评估失效产生的损害，这需要知道哪些进程正在失效的处理器上运行，哪些处理器和哪些进程（及其状态）仍然是活动的。要实现这一点，应用程序员必须能够控制把哪些对象放在哪些机器上。此外，处理器失效对在失效处理器执行的进程及其数据的影响必须清楚，还有，必须定义失效对于同那些进程及其数据进行的交互（不管是挂起的还是未来的）的影响。

3）使用损害评估的结果，剩余软件必须一致地对该失效作出一个响应，并采取行动去实现那个响应。为实现最大的容错，这一部分恢复过程必须是分布的。如果只有一个处理器执行恢复操作，那么该处理器的失效对于应用将是灾难性的。恢复工作可能需要那些正被改变的进程之间的通信路径，使得能够提供替代服务。还有，因为选择的响应会依赖于应用的整体状态，必须使某些数据能在所有机器上使用，而且为此要使它们处于一致的状态。例如，在航空电子系统中处理器失效后采取的动作将依赖于飞机的高度。如果不同处理器对这个高度有不同的值，那么所选的恢复过程将为不同目的服务。

4）只要实际可行，就修复失效处理器和/或其相关软件，且系统返回其无错状态。

本书中考察的实时编程语言都不提供应对处理器失效后动态重配置的足够设施。

3. Ada和容错

Ada语言没有对程序实现的失效模型做任何假设。只是规定了：如果一个划分试图同另一

个划分通信，而通信子系统检测到一个错误，就引发预定义异常Communication_Error。 554

Ada也不支持任何特定的容错方法，但允许实现提供适宜的机制。以下的讨论假设一个Ada分布式实现在失效－停止式处理器上执行。

Ada也没有显式地提供复制划分能力（虽然实现在这一点上是自由的——例如，见（Wolf，1998））。然而，划分通信子系统（PCS）可予扩充，所以可以提供复制的远程过程调用设施。每个复制的划分可关联一个可由系统使用的组标识符。所有远程过程调用都潜在地是复制的调用。包体可能需要访问一个有序多路广播设施。然而要注意，对于每个调度决策涉及到整个Ada运行时系统的情况，这个方法不允许任意的划分都被复制。所以，需要进一步的限制。

通过在扩展的PCS中提供保护对象能够提供异步通知。运行时系统检测到处理器失效时可调用一个过程。这样就能打开一个入口，它会在一个或多个任务中引起异步控制转移。如其不然，一个单独的任务或一个任务组可以等待失效的发生。下面的包说明这个方法。

```
Package System.RPC.Reliable is

  type Group_Id is range 0 . . 实现定义的值;

  -- 同步复制的调用
  procedure Do_Replicated_RPC(Group : in Group_Id;
            Params : access Params_Stream_Type;
            Results : out Param_Stream_Access);

  -- 异步调用
  procedure Do_Replicated_APC(Group : in Group_Id;
            Params : access Params_Stream_Type);

  type RRPC_Receiver is access procedure (Service : in Service_Id;
            Params : access Params_Stream_Type;
            Results : out Param_Stream_Access);

  procedure Establish_RRPC_Receiver(Partition : in partition_Id;
            Receiver : in RRPC_Receiver);

  protected Failure_Notify is
    entry Failed_Partition (P : out Partition_Id);
  private
    procedure Signal_Failure(P : Partition_Id);
    ...
  end Failure_Notify;

private
    ...
end System.RPC.Reliable;
```

一旦任务接到通知，它们必须评估已对系统造成的损害。这需要实际配置的知识。通过使用语言的动态绑定设施可进行重配置（见文献Burns and Wellings（1998））。 555

4. 实现occan2程序的可靠执行

虽然occan2是为用于分布式环境设计的，它却没有任何失效语义。由于内部错误（例如，数组界出错）失败的进程等价于STOP进程。若一个进程正在一个通道上等待通信，而与之通信的另一个进程是在一个失效的处理器上，那么它将永远等待下去。除非它利用“超时”。然而，可以想象对这种事件合适的occam2语义会将两个进程都作为停止了的进程，并提供一个

能够通知某些其他进程的机制。

5. 实现实时Java程序的可靠执行

Java要求可被远程调用的每个方法在其抛出表中声明`RemoteException`。这使底层实现能够在一个远程调用失败时给出通知。此外，Java对于定义支持复制的远程服务的可能性还没有定论。现在实时Java未在分布式系统中使用。

14.6 分布式算法

本章至此都在关注分布式程序使用像Ada、Java和occam2这些语言的表达问题，以及容许处理器和通信失效的一般问题。考察在分布式环境中控制和协调资源访问所需的特定分布式算法超出本书范围。然而，建立一些在分布式环境中可依赖的特性是很有用的。特别是需要说明事件怎么排序，如何组织存储器以使其内容在处理器失效时还能存在，以及怎么在处理器出故障时达成一致。这些算法在实现原子多路广播协议时是经常需要的。

14.6.1 分布式环境中的事件排序

在许多应用中，需要决定系统中已发生事件的顺序。这对于单处理器或紧密耦合系统没有困难，因为它们有公共存储器和公共时钟。然而，分布式系统没有公共时钟，并且在处理
556 器之间发送消息有一个延迟，这意味着这些系统有两个重要特性：

- 对任何给定的事件序列，不可能证明两个处理器会观察到完全相同的同一序列。
- 因为状态改变可以看作是事件，不可能证明任何两个进程对于系统状态的给定子集有相同的全局视图。在第12章引入了事件的因果次序概念，以帮助解决这个问题。

如果分布式应用中的进程需要协调和同步它们的活动以响应事件的发生，就必须给这些事件以因果次序。例如，为了检测共享资源进程之间的死锁，重要的是要知道一个进程在请求资源B之前释放资源A。这里给出的算法能使分布式系统中的事件排序，该算法属于Lamport(1978)。

考虑进程P，它执行事件$p0$、$p1$、$p2$、$p3$、$p4$、$\cdots pn$。因为这是一个顺序进程，事件$p0$必须在事件$p1$之前已经发生，$p1$必须在事件$p2$前已经发生，等等。这被写成$p0 \to p1 \to p2 \to \cdots \to pn$。类似地对进程$Q$：$q0 \to q1 \to q2 \to \cdots \to qn$。如果两个进程是分布的且二者之间没有通信，就不可能说$p0$在$q0$、$q1$之前或之后发生，等等。现在考察这种情况：事件$p1$发送一个消息给Q，而$q3$是接收来自P的消息的事件。图14-12说明了这种交互。

由于接收消息这件事必然在发送这个消息之后发生，所以$p1 \to q3$，由于$q3 \to q4$，可得$p1 \to q4$（即$p1$到$q4$可能有因果关系）。关于$p0$和$q0$谁先发生依然无任何信息。这些事件被称为**并发事件**（无因果性次序）。由于它们之间无任何事件影响其他事件，所以考虑哪个先发生并不重要。然而，重要的是所有基于这个次序作出决策的进程都使用同一次序。

为了将分布式系统中的所有事件全部排序，必须为每个事件关联一个**时间戳**(time stamp)。然而，这不是一个物理时间戳，而是一个逻辑时间戳。系统中的每个处理器保持一个逻辑时钟，它在那个处理器上每发生一个事件时增加一次。进程P中的事件$p1$在同一进程的$p2$之前发生，如果$p1$的逻辑时钟值小于$p2$的逻辑时钟值的话。显然，对于同每个进程关联的逻辑时钟，能用这个方法显示出同步关系。例如，P在$p1$的逻辑时钟可能大于Q在$q3$的逻辑时钟，然而，$p1$必须在$q3$之前发生，因为一个消息在发送前不可能被接收。为解决这个问题，必须让在进

程之间发送的每个信息上带有发送该消息的那个事件的时间戳。而且，进程每接收到一个消息，必须将它的逻辑时钟置成一个值，这个值是自己拥有的值和下面值中的较大者：

- 消息中发现的时间戳至少加1——对于异步消息传送；或
- 时间戳——对于同步消息传送。

这能确保无任何消息在发送前就被接收。

用上述算法，系统中的所有事件都关联了一个时间戳，并可据此部分排序。如果两个事件有相同时间戳，那就能使用任意条件（例如使用进程标识符的数值），给出事件的一个人工全序。

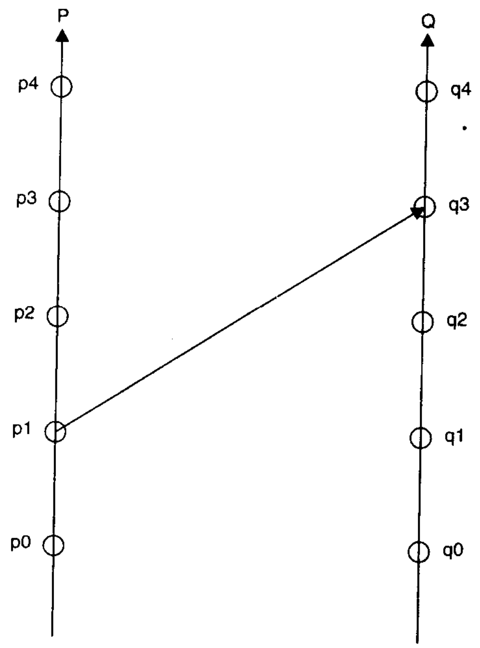


图14-12 两个交互进程

14.6.2 全局时间的实现

上面描述的全局排序法使用逻辑时钟。一个替代方案是使用一个基于物理时间的全局时间模型。这可以通过所在节点访问同一时间源的办法实现，但对于节点拥有自己的时钟的情况更为有用。所以，必须协调这些局部时钟。做这件事有许多算法。所有这些算法都涉及补偿不可避免的时钟漂移，即使时钟在概念上都等同的情况，这种漂移也会发生。

使用石英晶体，两个时钟在6天左右即可漂移1秒。如果时间性事件是在微秒级的，那么8分钟的漂移就会成为问题。

为限定时钟漂移，必须在每个节点操纵时间基准。时间是不会倒退的，虽然可以让慢的时钟向前跳若干“滴答”，快的时钟就只能让它慢下来，而不是把时间向后拨。 557~558

时钟协调算法将为任何两个时钟之间的最大差值提供一个界限Δ（被一个外部参考时钟观察到）。现在可假设事件A先于事件B，当(且仅当)

$$t(A)+\Delta<t(B)$$

其中$t(A)$是事件A的时间。

实现节点之间同步的一种方法是，中心时间服务器进程S根据请求按它自己的时钟提供时

间。这个时间服务器可能被关联到一个外部时间源或有一个较精确的硬件时钟。或者它可以是系统的任意节点，但被当作系统时间源。在分布式系统中，通信不是瞬时的，所以由S提供的时间会有两个误差源——由S返回消息所花时间的变异性和一旦客户接收该消息后的灵敏性引入的某种不确定性。S在读其时钟和送出消息之间是不被抢占也是很重要的。

为了减少消息往返引入的抖动，S可以定期送出自己的时间。此外，如果使用一个高优先级消息（S自身有高优先级），则可使变异性的其他来源最少。

对S的客户的另一种方法是连同消息发送出自己的时间。S在接收消息时看看自己的时钟，并算出一个校正因子δ:

$$\delta = t(client) + min - t(S)$$

其中min是来自客户消息的最小通信延迟。

然后，校正因子被传送回客户。第二个消息的传输时间并不是时间至上的。在接受第二个消息时，若δ的值是负的，客户就向前拨自己的时钟（以δ），若δ是正的，就以这个量减慢自己的时钟。单个时钟源的使用方案简单，但易受S（或是执行它的节点）的单点失效支配。其他算法使用一种分散方法，即所有节点广播它们的时间并进行一致性表决。这样的表决可能去除界外的值。这些时钟同步算法必须满足两个特性，即**一致性**和**准确性**。

- 一致性条件——仅当任意两个无故障时钟之间的偏斜在界内时，它才满足；
559 • 准确性条件——仅当所有无故障时钟相对于真实时间的漂移都在界内时，它才满足。

14.6.3 实现稳定存储

在许多实例中，必须访问当处理器崩溃时其内容得以保存的存储器，这称为**稳定存储**(stable storage)。由于任何处理器的主存储器都是易坏的，必须用磁盘（或其他形式的不易坏存储器）作为稳定存储装置。但是，写磁盘操作不是原子操作，因为该操作可能在中途崩溃。发生这种事的时候，恢复管理程序不可能确定该操作是否已经完成。为解决此问题，数据的每个块都在磁盘上的不同区域存储两次。这些区域的选择使得当一个区域的读头损坏时不会破坏另一区域，如果需要，它们可以在物理上分离的磁盘驱动器上。假设磁盘装置将指出一个单独的写操作是否成功完成（利用冗余检查）。在没有处理器失效的情况下，这个方法是将数据块写到磁盘的第一个区域（这个操作可能需要重复，直到它成功），只在成功的情况下，这个块才被写到磁盘的第二个区域。

在修改稳定存储时若发生了崩溃，可执行如下的恢复例程。

```
read_block1;
read_block2;
if both_are_readable and block_1 = block2 then
    --什么也不做，崩溃没有影响稳定存储
else
    if one_block_is_unreadable then
        -- 将好块复制到坏块
     else
        if both_are_readable_but_different then
            -- 复制block1 到block2(或反过来)
        else
            -- 发生了灾难性失效，两个块都不可读
        end
```

```
    end;
  end;
```

即使在该算法的执行过程中有后续的崩溃，它也会成功。

14.6.4 故障性进程出现时达成一致

在本章前面假定：若一处理器失效，它就寂静地失效。这意味着处理器有效地停止所有执行并且不参与同系统中任何其他处理器的通信。确实，上面给出的稳定存储算法就做了这样的假设：处理器崩溃导致处理器立即停止其执行。没有这个假设的话，有毛病的处理器就可能进行任意的状态迁移并向其他处理器发送谬误消息。因此，即使逻辑上正确的程序也不能保证产 560
生希望的结果。这会使得看起来不可能实现容错系统。虽然可以做出各种努力去建造不论部件失效与否都能正确操作的处理器，却不可能保证有限数量的硬件做到这一点。所以，必须假设失效数目的一个界限。本节考察在不同处理器上执行的一组进程怎样在组中出现有界数目的故障性进程时能够达成一致性。假设所有通信是可靠的（即足够的复制以保证可靠的服务）。

拜占庭将军问题

驻留在故障性处理器上进程之间的值一致性问题通常被表达为**拜占庭将军问题**（Byzantine Generals Problem）（Lamport等，1982）。拜占庭军队的若干分队（每个分队由自己的将军指挥）包围一个敌人营地。将军们可通过通信员通信，并必须对是否攻击敌人营地达成一致。为此，每个将军都观察敌军营地，并将他或她的观察告诉其他人。但是，有一位或多位将军可能是奸细，并且可能传送错误信息。问题是让所有忠诚的将军得到相同的信息。一般来说，如果他们进行$m+1$轮消息交换的话，$3m+1$位将军就能对付m个奸细。为了简明性，我们用四位将军对付一个奸细的算法来说明这个方法，并做出如下假设：

1）每个发出的消息都正确交付。

2）消息的接收者知道谁发送这个消息。

3）能够检测到消息的丢失。

读者可参看文献以得到更一般的解决方案（Pease等，1980；Lamport等，1982）。

考察将军Gi和他/她观察到的信息Oi。每位将军维持一个他/她从其他将军那里接收的信息的向量V。开始时，Gi的向量只有值Oi，即$Vi(j)=null(for\ i<>j)\ and\ Vi(i)=Oi$。每位将军派一个通信员到其他将军那里去传达他/她的观察。忠诚的将军总是发送正确的观察；奸细将军可能发送一个假观察并可能向不同将军发送不同的假观察。每位将军接收这些观察后就更新他/她的向量，并将其他三位的观察值发送给其他将军。显然，奸细可能发送不同于他/她接受的观察，或者根本不发送。对于后一种情况，将军们可选择任意值。

交换消息之后，每位忠诚的将军就能根据从每位将军的观察接收的三个值的多数值构造一个向量。如果没有多数值存在，他们就假没有进行观察。

例如，假定一位将军的观察导致他/她做出三个结论之一：攻击（A）、退却（R）或等待 561
（W）。考虑$G1$是个奸细、$G2$的结论是攻击、$G3$是攻击、$G4$是等待的情况。每个向量的初始状态如图14-13所示。

向量（1..4）的序标给出将军的编号，那个序标的项目内容给出那个将军的观察和谁报告了这个观察。开始时，第4位将军在其向量的第四个位置放了一个"wait"，指出这个观察来自他自己。当然，在一般情况下，不可能证实是谁报告了某个观察。然而，在本例中假设奸细没有那么狡猾。

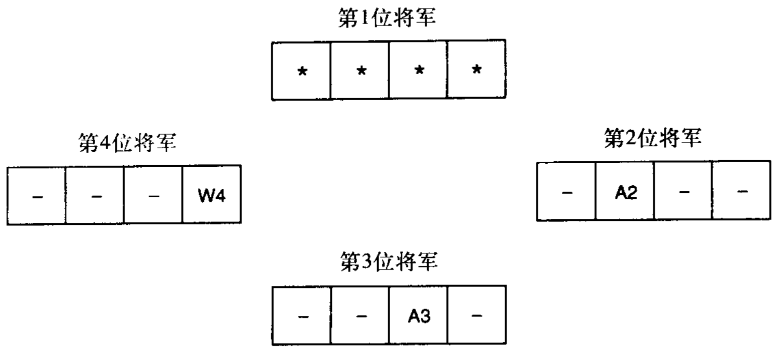


图14-13 拜占庭将军问题——初始状态

在第一次交换消息之后，这些向量如图14-14所示（假设奸细认识到敌人营地是易受攻击的并随机地向其他将军发送退却和等待消息）。在第二次交换之后，图14-15给出了每位将军可用的信息（再次假设奸细随机地发送“退却”和“等待”消息）。

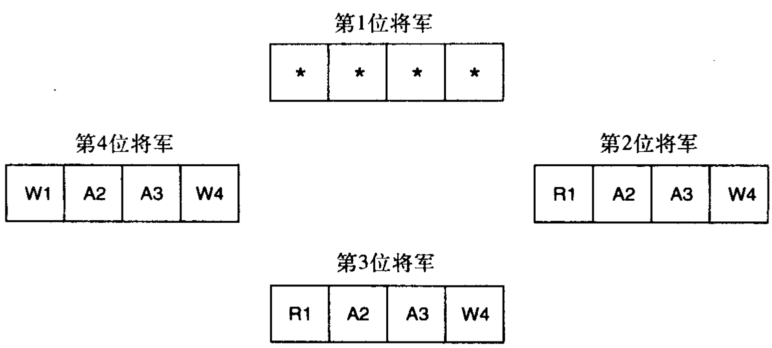


图14-14 拜占庭将军问题——第一次交换消息之后的状态

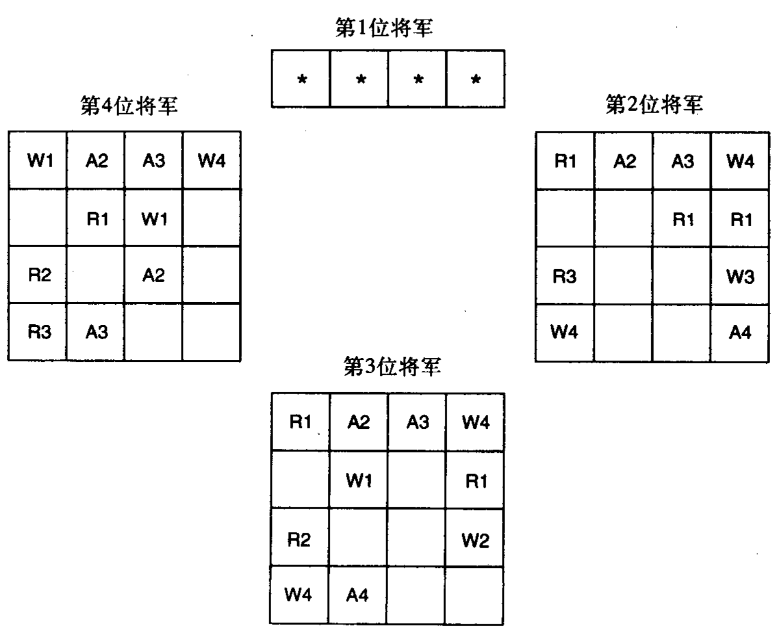


图14-15 拜占庭将军问题——第二次消息交换

为说明此表是怎样得来的，考察第4位将军的最后一行。它是从第3位将军那里得到的（如3所指出的），并包括了第一位和第二位将军的决定。所以，在第一列的R3表示：第3位将军认为第1位将军希望撤退。

图14-16给出了最终（大多数）向量。忠诚的将军对于每个其他人的观察有一致的观点，因此可以做出一致的决定。

如果能够进一步限制奸细的行为（即更严格的失效模型），则容忍m个奸细（失效）所需的将军（处理器）数可以减少。例如，如果奸细不能改变忠诚将军的观察（即他必须传送一个观察的签名副本，签名也不能伪造或修改），则只需要$2m+1$个将军（处理器）了。

利用拜占庭将军问题的求解方法，通过让内部复制处理器达成一个拜占庭协议，就可能构造出一个寂静失效处理器（Schneider，1984）。如果无故障处理器检测出了不一致，它们就 562~563
停止执行。

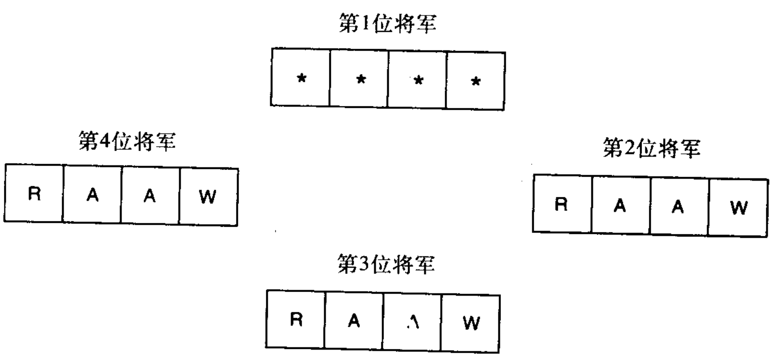


图14-16 拜占庭将军问题——最终状态

14.7 分布式环境中的时限调度

考察几个分布式体系结构、通信协议和算法之后，现在可以返回到理解建立在分布式环境上的应用的时态行为的关键问题。在这里，系统的不同部件可以不同速度前进，而且通信延迟变得不可忽视了。不仅环境的时间必须同计算机系统的时间相连接，不同处理器/节点也必须有某种形式的时间连接形式。**同步**这个术语（在这个上下文中）是指具有下列特性的分布式系统：

- 消息延迟有个上界，延迟由发送所用时间、在某通信链路上的传输和接收消息的时间组成。
- 每个处理器有一局部时钟，并在任意两个时钟之间有一个有界漂移率。
- 处理器本身至少用最小速度前进。

注意，这不意味着不会发生故障。实际上，消息延迟的上界只用于无故障处理器在无故障链路上通信的时候。确实，上界的存在可用于提供失效检测。

不具备上述三个特性中的任一特性的系统叫做**异步**的。

本节只考虑同步系统。将要研究两个主要议题。第一个是进程到处理器的分配问题，然后讨论通信调度问题。

14.7.1 分配 564

为分布式（和多处理器）系统开发合适的调度方案是一件问题很多的事。Graham（1969）

指出：多处理器系统的行为很难用它们展现的时间方面的行为预测。他使用动态分配（即在进程变成可运行时分配给处理器），能够说明下列反常现象：

- 某个进程P执行时间的减少可能导致它的响应时间增加。
- 提高进程P的优先级，可能导致它的响应时间增加。
- 增加处理器的数目，可能导致P的响应时间增加。

所有这些结果都显然同直觉冲突。

Mok和Dertouzos（1978）指出对单个处理器系统最优的算法在处理器数目增加时并不最优。例如，考察3个周期过程P_1、P_2和P_3，它们必须在两个处理器上执行。令P_1和P_2有相同的时限要求，即有一个50个时间单位的周期和时限以及25个时间单位的执行需求（每周期）；令P_3的相应需求是100和80。如果使用（第13章讨论过的）速率单调算法，P_1和P_2将有最高优先级并在两个处理器上（并行地）运行它们需要的25个单位。这就会要P_3在可用的75个单位里完成80个单位的执行。P_3有两个处理器可用这一事实与此不相干（一个将保持空闲）。作为使用速率单调算法是结果，P_3将错过它的时限，即使平均处理器利用率只有65%。然而，将P_1和P_2映射到一个处理器和把P_3映射到另一处理器的分配容易满足所有时限。

类似地，另外的例子可以说明最早时限优先（EDF）方案不是最优的。使用最优单处理器算法的困难不足为怪，因为，众所周知，多处理器系统的最优调度是NP难度的（Graham等，1979）。所以，必须寻找简化问题的途径，并提供给出适宜次优结果的算法。

1. 周期进程的分配

上述讨论显示进程的合理分配可显著影响调度性能。考察另一个例子：4个进程在两个处理器上执行，令它们的周期时间为10、10、14和14。两个为10的进程分配到同一处理器上（隐含着两个为14的进程分配给另一个处理器），那么可达到100%的处理器利用率。即使4个进程的执行时间是5、5、10、4，系统也是可调度的。然而，如果将一个为10、一个为14的进
565 程分配到同一处理器上（作为动态分配的结果），则最大可利用率降至83%。

看来，这个例子表明：将周期进程静态分配比使其移动要好些，后一种情况会使分配不均衡并可能使系统性能下降。即使在使用单个运行时分配器的紧耦合系统上，使一个进程保持在同一个处理器上也要比试图利用一个空闲处理器（并冒不均衡分配的风险）要好。

如果使用静态部署，则时限单调算法（或其他优化的单处理器方案）可测试出每个处理器上的调度性能。在进行分配时，能相互“融洽”的进程应当部署在一起（即在同一处理器上），这样有助于提高利用率。

2. 偶发和非周期进程的分配

就像看来对周期进程使用静态分配比较有利一样，对于偶发进程，有一个类似方法看来是个有用的模型。如果所有进程都被静态映射，那么第13章讨论的算法可用于每个处理器（即每个处理器实际上运行它自己的调度程序/分派程序）。

计算执行时间（最坏情况和平均情况）需要有关潜在阻塞的知识。在本地处理器上的阻塞可能是由继承或高限协议决定的。然而，在多处理器系统中有另一种形式的阻塞：当一个进程被另一处理器上的一个进程延迟时就是这种情况。这叫做远程阻塞而且不易限制。在分布式系统中，通过增加进程去管理数据的分布可将远程阻塞消除。例如，不是被阻塞去等待读取来自远程站点的数据，而是在那个远程站点上增加一个额外进程，其作用是把数据转发到需要它的地方。因此这种数据是局部可用的。对设计的这种修改可以系统化地进行，然而，

它确实使应用复杂化了（但是也确实导致一个较简单的调度模型）。

纯静态分配策略的一个缺点是不能从一个处理器的剩余容量中得到好处（同时另一个处理器正在短暂地繁忙）。对硬实时系统，每个处理器都需要有能力为周期进程处理最坏情况执行时间和为其偶发进程处理最大到达时间和执行时间。为改善这种状况，Stankovic等（1985）以及Ramamritham和Stankovic（1985）提出过更灵活的任务调度算法。

在他们的方法中，所有安全至上的周期和偶发进程被静态分配，而非安全至上的非周期进程可以移动。使用下列协议：

- 每个非周期进程到达网络中的某个节点上。
- 非周期进程到达的节点检查这个新进程是否能同已有的负载一起调度。如果能，就称该进程是受此节点保证的。
- 如果说节点不能保证该新进程，就寻找有能力保证它的其他可供选择的节点。做这件事要使用有关整个网络状态的知识，并竞争其他节点的剩余能力。 566
- 这时，该进程被移动到新节点上，在那里调度它的概率很大。然而，由于竞争条件，当它到达时新节点可能不会马上调度它。所以，保证测试是局部进行的，测试失败的进程必须再次移动。
- 按这种方式，非周期进程或者被调度（被保证），或者无法满足时限。

他们这个方法的有用性通过使用一个确定无保证进程应当移动到哪里的线性启发式算法而得到增强。这个启发式算法的计算不昂贵（不像优化算法，它是NP难度的），但确实有很高的成功率；即用该启发式算法导致一个非周期进程被调度的概率很大（就像是完全可调度的）。

执行该启发式算法和移动非周期进程的开销是由保证例程计算的。不过，这个方案只在能够移动非周期进程并在这种移动是高效的时候才可行。某些非周期进程可能同硬件紧密耦合，而每个节点的硬件又各不相同，可能至少有一个部件必须在本地执行。

14.7.2 调度对通信链路的访问

分布式系统中不同机器的进程之间的通信需要底层通信子系统传输和接收消息。一般来说，这些消息要相互竞争，以获得对网络介质（例如，交换机、总线或环路）的访问权。为使硬实时进程满足它们的时限，必须以一种同每个处理器上进程调度一致的方式调度对通信子系统的访问。如果不这样做，那么在高优先进程试图访问通信链路时就可能发生优先级反转。诸如同以太网相关的标准协议不支持硬时限传输，因为它们使用的是按FIFO次序将消息排队，并在消息冲突时使用不可预测的后退算法。

虽然通信链路只是另一种资源，但至少有四个方面的问题把链路调度和处理器调度问题区分开来：

- 与处理器不同（它只有一个访问点），一个通信通道有许多访问点——每个附属的物理节点有一个。因而需要一个分布式协议。
- 虽然抢占式算法对单处理器的进程调度是适用的，消息传输中的抢占将意味着整个消息需要再次传输。
- 除了由应用进程加上的时限外，缓冲区的使用也有时限——在新数据放进去之前必须将缓冲区内容传输出去。 567

虽然分布式环境中用了许多特定目的的方法，至少有三种方案确实能够得到可预测的行为。现予以简要讨论。

1. TDMA

对于单处理器调度使用循环执行的一个自然扩充是对通信使用循环方法。如果所有应用进程是周期性的，就有可能产生一个按时间分槽的通信协议。这样的通信协议称为TDMA（Time Division Multiple Access，时分多路访问）。每个节点有一个同其他所有节点时钟同步的时钟。在一个通信周期内，为每个节点分配若干它可以进行通信的时间槽。它们同每个节点的循环执行的执行槽是同步的。不会发生消息冲突，因为每个节点知道它什么时候可以写，此外，每个节点知道什么时候有一个消息需要它去读。

TDMA方法的困难来自调度的构造。这个困难随系统中的节点数指数性地增长。使用TDMA取得可观成功的一个体系结构是TTA（Time Triggered Architecture，时间触发体系结构）（Kopetz，1997）。它用复杂的图归纳启发式算法去构造调度。TDMA的另一个缺点是难以计划什么时候能够传输偶发消息。

2. 定时令牌传递方案

纯时间槽方法的一个推广是使用令牌传递方案。这时有一种特殊的从节点传递到节点的消息（**令牌**，token）。只有当节点拥有令牌时才能发送消息。因为只有一个令牌，因而不会发生消息冲突。每个节点拥有令牌的时间不能超过一个最长时间，因而有一个有界**令牌轮转时间**。

有不少协议使用这个方法。光纤FDDI（Fiber Distributed Data Interface，光纤分布数据接口）协议就是一个例子。这里消息被分为两类：同步的和异步的。同步消息有硬时间约束，并被用于确定每个节点的令牌拥有时间，并因而称为*目标令牌轮转时间*。异步消息不像这样有硬约束，它们可以通过一个节点进行通信，只要该节点没有同步消息要发送或令牌已先到达该节点（因为其他节点已经没有什么要传输）。该协议的最坏情况行为发生在恰好一个节点刚刚把令牌交出就有一个消息到达的时候。此外，到那个时间为止没有节点传输消息，所以令牌很早就交出。在有新同步消息的节点交出令牌之后，其余节点有很多消息要发送。第一个节点发送它的同步消息，并且由于令牌早已到达，它发送完整的异步消息集，所有后续节点发送它们的同步消息。到令牌回到感兴趣节点的时候，一个等于两倍*目标令牌轮转时间*的
568 时间段已经过去了。这就是有界交付时间。

3. 基于优先级的协议

知道了处理器上使用基于优先级调度的好处，就有理由假设基于优先级方法对于消息调度是很有用的。这种协议有两个阶段。在第一阶段，每个节点指出想要传输的消息的优先级。显然，消息集中的最大优先级会占先。在第一阶段的末尾，一个节点获得传输消息的权利。第二阶段就只是这个消息的实际通信。在某些协议中，两个阶段可以重叠（即当一个消息被广播时，该消息的部分被修改，得以确定下一个消息的优先级）。

虽然，基于优先级的协议已经存在一些时候，其缺点是通信协议只支持小范围的优先级。它们已被用于区分消息的广泛类别而不是用于消息调度。如第13章指出的，基于优先级调度的最好结果出现在每个进程（或这里讨论的消息）有一个不同的优先级的时候。幸好，现在有些协议确实提供大范围的优先级。一个例子是CAN（Controller Area Network，控制器区域网络）（Tindell等，1995）。

在CAN 2.0A中，8字节消息的前面有一个11位的标识符，它就是优先级。在传输序列的开始，每个节点（同时）向广播总线写其最大优先级消息标识符的第一位。CAN协议的行为

就像一个大与（AND）门，如果有任何节点写了一个0，则所有节点就读一个0。0位被称为**支配位**。协议这样前进（对标识符中的每个位）:

- 如果一个节点传输了一个0，它继续到下一位。
- 如果一个节点传输了一个1并读回一个1，则继续到下一位。
- 如果一个节点传输了一个1并读回一个0，它就退回，并不再进一步参与该轮传输。

标识符的值越小，优先级就越高。由于标识符是各不相同的，协议被强迫终止在11轮逐位仲裁后留下来的那个节点。于是该节点传输它的消息。

CAN的价值在于它是个真正的基于优先级的协议，所以第13章进行的所有分析都适用。CAN使用的这类协议的不足是它限制了通信的速度。为使所有节点“同时”写其标识符位并读取接着到来的“与”值（并在发送出去之前处理这个值，或者是下一位），必须给传输速度一个严格的界限。这个界限实际上是用于传输这些位的线路长度的一个函数，所以CAN不适合地理分散的环境。它实际是为现代汽车的信息系统设计的——这是它取得相当大成功的地方。 569

4. ATM

ATM可以跨广域网和局域网使用。目标是支持范围很宽的通信需求，诸如需要声音和视频以及数据传输。ATM通过一个或多个交换机支持点对点通信。典型情况是，网络中的每台计算机通过两根光纤连到交换机：一根将通信流量传输到交换机，另一根转接来自交换机的通信流量。

所有传输的数据被分割成定长的包，叫做**信元**（cell），每个信元有一个5字节的首部和48字节的数据段。应用经由**虚拟通道**（virtual channel，VC）通信。在一个特定VC上传输的数据可有某种与之相关联的时间性行为，诸如其位率、周期或时限。**适配层**（adaptation layer）提供支持用户特定种类数据的专门服务，该层的精确行为是可变的，以适应具体系统的数据传输需要。正是在这个适配层进行端对端出错纠正和同步，并完成用户数据到ATM信元的分段和重装配。

典型的ATM网络有多台交换机，每台交换机有一个与之关联的信元排队策略——大多数早期的商业交换机使用基本的FIFO策略，但现在却迎合基于优先级排队。在单个交换机内部，依据连接表，可建立若干交换机输入端口和输出端口之间的一组连接。在输入和/或输出端口可能发生信元的排队现象。

实时ATM解决方案采取的常用方法如下:

- 预先定义所有VC时间性需求。
- 预先定义分配给每个VC的路径和网络资源。
- 控制每个VC在它路由经过的每个网络资源所需的总带宽。
- 计算所产生的延迟，并评价从VC到网络硬件的特定分配的可行性。

需要控制每个VC使用的带宽以避免网络拥挤。这样做的动机是支持每个VC和网络作为一个整体的可预测行为。当拥挤出现时，ATM信元就处于被放弃（根据ATM层协议的正常规则）的风险之中，并且难以给出一个不太悲观的最坏情况延迟的预测（使得能够检测丢掉的信元和可能的重新传输）。 570

14.7.3 整体调度

一个相当大的分布式实时系统可能包含数10个处理器和2至3个不同的信道。处理器和通信子系统都可以被调度以使最坏情况时间性行为是可预测的。使用静态分配方法有利于这样

做。对系统的每个构件进行分析之后，就有可能将这些预测组合到一起来检测同整个系统的时间性需求是否一致（Tindell and Clark，1994；Palencia Gutierrez and Gonzalez Harbour，1998；Palencia Gutierrez and Gonzalez Harbour，1999）。为讲述这个*整体调度*（holistic scheduling）问题，要考虑两个重要因素：

- 一个构件行为的变异会严重地影响系统其他部分的行为吗？
- 每个构件的最坏情况行为的简单叠加会导致悲观的预测吗？

变异量依赖于调度的方法。如果使用纯时间触发方法（即经由TDMA通道链接的循环执行），从循环行为偏离的程度就很小。然而，如果在处理器和通道上使用基于优先级调度，就可能有相当大的变异。例如，考虑由另一节点上一个周期进程的执行启动的一个偶发进程。平均来说，该偶发进程同那个周期进程有相同的执行率（例如，每50ms一次）。但是，周期进程（以及后续的启动该偶发进程的通信消息）将不会在每个周期同样的时间发生。该进程的执行对于一个调用可能相对晚一些，而对下一调用却早一些。结果，偶发进程可能仅在前一启动之后的30ms就被第二次启动。将偶发进程作为周期为50ms的周期进程建模是不正确的，并会导致所有时限均满足这一错误结论。幸而，启动时间的这种变异可利用13.12.2节给出的启动抖动分析精确地建模。

虽然为单个处理器调度引入的响应时间分析是必要且充分的（即给出处理器的真实行为的精确描述），整体调度却可能是悲观的。当在一个子系统上发生最坏行为就隐含着在其他部件上会经历并非最坏的情况时，就是这样。常常只在仿真研究时会允许（统计地）确定悲观的等级。需要进一步的研究去确定整体调度的有效性。

关于整体调度最后一个要注意的问题是可以把支持放到分配问题上。静态分配看来是最适于使用的。但是，推导最佳静态分配依然是一个NP难度问题。针对这个问题考虑过许多启发式算法。最近，诸如模拟退火和遗传算法这样的搜索技术已被用于整体调度问题。已经证明它们是十分成功的，并能容易地扩展到用于为容错而进行复制的系统（在这些地方当把复
571 制品分配给不同构件时必须想到分配问题）。

小结

本章将分布式计算机系统定义为共同目的而合作或实现共同目标的自治处理元素的集合。当考虑分布式应用时提出的某些议题引出了简单实现之外的一些基本问题。它们是：语言支持、可靠性、分布式控制算法和时限调度。

语言支持

在分布式硬件系统上执行的分布式软件系统的生产涉及：**划分**——将系统划分成为部件（分布的单位）的过程，这些部件适合于安放在目标系统的处理元素上；**配置**——将程序划分出来的诸部件同目标系统的特定处理元素相关联；**分配**——将已配置系统调整成可执行模块的集合并下载它们到目标系统的处理元素的实际过程；**透明执行**——分布式软件的执行，对远程资源能以一种同位置无关的方式访问；**重配置**——对软件构件或资源的位置进行改变。

虽然occam2是为分布式环境设计的，但还是有些低级，进程是划分的单元；配置在语言中是显式的，但既不支持分配也不支持重配置。而且，对资源的远程访问也不是透明的。

Ada允许将库单元集合分组成“划分”，划分之间通过远程过程调用和远程对象通信。这个语言既不直接支持配置，也不支持分配和重配置。

Java允许对象分布，对象可通过远程过程调用或“套接字（socket）”进行通信。Java语言不直接支持配置、分配和重配置。

CORBA有利于用不同语言为不同平台编制的应用之间的分布式对象和互操作性，这些平台得到来自不同供货商的中间件软件的支持。

可靠性

虽然多处理器的可用性使应用变得容许处理器失效，它也引入了发生集中式单处理器系统中不会出现的故障的可能性。尤其是多处理器引入了部分系统失效的概念。此外，通信介质可能丢失、损坏或改变消息的次序。 572

跨机器边界的进程间通信需要协议的分层，以容许瞬时的出错状况。已经确定了支持这些协议层次的标准。OSI参考模型是一个由应用、表示、会话、传输、网络、数据链路和物理层组成的分层模型。它主要是为了使广域网能进行开放式访问而开发的，广域网以低带宽通信和高出错率为特征。TCP/IP是另一个主要以广域网为目标的协议参考模型。然而，大多数分布嵌入式系统使用局域网技术，并对外部世界是关闭的。局域网以高带宽通信和低出错率为特征。所以，虽然能够用OSI方法实现语言级的进程间通信，但实际中其开销却经常禁止这样做。因此，许多设计者将通信协议按语言（应用）和通信介质的要求进行裁剪。这些称为轻量级协议。

当进程组进行交互时，通信必须被送到整个组。**多路广播**通信模式提供这样的设施，可以设计一个组通信协议族，每个协议提供一个有特定保证的多路通信设施：**不可靠多路广播**——不提供交付到组的保证；**可靠多路广播**——协议对于将消息交付到组做最好的尝试；**原子多路广播**——协议保证如果组中的某进程收到了消息，则组中的所有成员都收到消息；**有序原子多路广播**——除了保证多路广播的原子性之外，协议还保证组中的所有成员都以同样顺序收到来自不同发送者的消息。

可通过静态或动态冗余容许处理器失效。静态冗余涉及在不同处理器上复制应用部件。复制的数量可依具体构件的重要性而异。为应用提供透明容错的问题之一是程序员不可能规定降级执行还是安全执行。静态冗余的另一替代方案是容许处理器失效由应用程序员动态处理。这需要：检测处理器失效并告诉系统中的其余处理器；必须评估已经发生的破坏；其余软件用这些结果对响应达成一致并进行必要的活动以实现响应；尽可能快地修复失效的处理器和/或其相关软件，系统返回到正常的无错状态。很少有实时编程语言提供合适的设施去对付处理器失效之后的动态重配置问题。 573

分布式控制算法

应用中真正并行性的出现、物理上分布的处理器以及处理器和通信链路失效的可能性，需要资源控制的许多新算法。本章研究了下列算法：事件排序、稳定存储实现和拜占庭一致性协议。许多分布式算法假设处理器是“失效－停止”型的，这意味着要么它们正确地工作，要么在故障发生时立即停止。

时限调度

遗憾的是，将进程动态地分配给处理器的一般问题（使得系统范围的时限得到满足）是个昂贵的计算。所以必须实现一个不那么灵活的分配方案。一个粗略的方法是静态部署所有进程或只允许非关键的进程移动。

一旦处理器分配好了，就必须对通信介质进行调度。TDMA、时间性令牌传递和基于优先级的调度是合适的实时通信协议。最后，端到端时间性需求必须通过考虑整个系统的整体调度予以验证。

相关阅读材料

Brown, C. (1994) *Distributed Programming with Unix*. Englewood Cliffs, NJ: Prentice Hall.

Comer, D. E. (1999) *Computer Networks and Internets*. New York: Prentice Hall.

Coulouris, G. F., Dollimore, J. and Kindberg, T. (2000) *Distributed Systems, Concepts and Design*, 3rd Edition. Harlow: Addison-Wesley.

Farley, J. (1998) *Java Distributed Computing*. Sebastopol, CA: O'Reilly.

Harold, E. (1997) *Java Network Programming*. Sebastopol, CA: O'Reilly.

Halsall, F. (1995) *Data Communications, Computer Networks and OSI* 2nd Edition. Reading, MA: Addison-Wesley.

Hoque, R. (1998) *CORBA 3*. Foster City, CA: IDG Books Worldwide.

Horstmann, C. S. and Cornell, G. (2000) *Core Java 2, Volume II – Advanced Features*. Sun Microsystems.

Kopetz, H. (1997) *Real-Time Systems: Design Principles for Distributed Embedded Applications*: New York: Kluwer International.

574 Lynch, N. (1996) *Distributed Algorithms*. San Mateo, CA: Morgan Kaufmann.

Mullender, S. (ed.) (1993) *Distributed Systems*, 2nd Edition. Reading, MA: Addison-Wesley.

Tanenbaum, A. S. (1994) *Distributed Operating Systems*. Englewood Cliffs, NJ: Prentice Hall.

Tanenbaum, A. S. (1998), *Computer Networks*. Englewood Cliffs, NJ: Prentice Hall.

练习

14.1 讨论分布式系统的缺点。

14.2 Ada支持一个实时系统附件和分布式系统附件。讨论这两个附件组成分布式实时系统附件的哪些方面。

14.3 从数据抽象的观点讨论为什么在远程对象的接口中不应看到变量。

14.4 考察在一个分布式Ada环境中定时和条件入口调用的含义。

14.5 下列occam2进程有5个输入通道和3个输出通道。从输入通道接收的所有整数被输出到所有输出通道:

```
INT I, J, temp:
  WHILE TRUE
    ALT I = 1 FOR 5
      in[I] ? temp
        PAR J = 1 FOR 3
          out[J]! temp
```

因为这个进程有8个通道接口，它不可能在单个传输机上实现，除非其客户进程在同一传

输机上。将这段代码变换使之能在三个传输机上实现（假设一个传输机只有4个链路）。

14.6 比较并对照Ada、Java和CORBA的远程对象模型。

14.7 用Java能够实现CORBA的消息传递服务的哪些方面？

14.8 画出联合国安理会上一位法国代表想同俄国代表交谈的通信层次。假设译员能翻译到一种公共语言（例如，英语），然后将消息传递给电话接线员。这种层次化的通信遵循ISO OSI模型吗？

14.9 为什么远程过程调用的语义同普通过程调用的语义不同？

14.10 如果把OSI网络层用于实现RPC设施，应当使用数据报还是虚拟线路服务？

14.11 比较并对照在处理器失效时能实现可靠系统数据存活的稳定存储方法和复制数据方法。

14.12 重做在14.6.4给出的拜占庭将军问题，假设G1是奸细、G2的结论是退却、G3的结论是等待而G4的结论是攻击。

14.13 假定实时通信子系统可以选择以太网或令牌环状网，如果主要考虑的是重负载下的确定性访问，应当选用哪一种网？ 575~576

14.14 将10.8.2节给出的算法修改成在分布式系统中的向前出错恢复算法。

第15章 低级编程

嵌入式系统的一个主要特征是它同专用输入输出（I/O）设备交互的要求。但是，存在很多不同类型的设备接口和控制机制。这主要是因为：不同的计算机提供不同的设备连接方法；不同的设备对控制有单独的要求；不同制造商制造的类似设备有不同的接口和控制要求。

为了提供低级设备编程所需的高级语言设施的基本原理，必须了解基本的硬件I/O功能。因此，这一章首先研究这些机制，然后讨论一般的语言特征，最后给出具体语言的细节。

嵌入式系统通常有存储容量的限制，因此程序员必须确保编译器只分配正在进行的作业所需的存储。此外，必须小心地控制动态存储分配，以确保时间至上的代码不会被系统功能（例如垃圾回收）的不合时宜的执行所抢占。

15.1 硬件输入/输出机制

就输入和输出设备而言，计算机体系结构有两个大类：一类是存储器和I/O设备逻辑上各有一条总线，而另一类是存储器和I/O设备在同一逻辑总线上。图15-1和15-2概略地表示了它们。 577

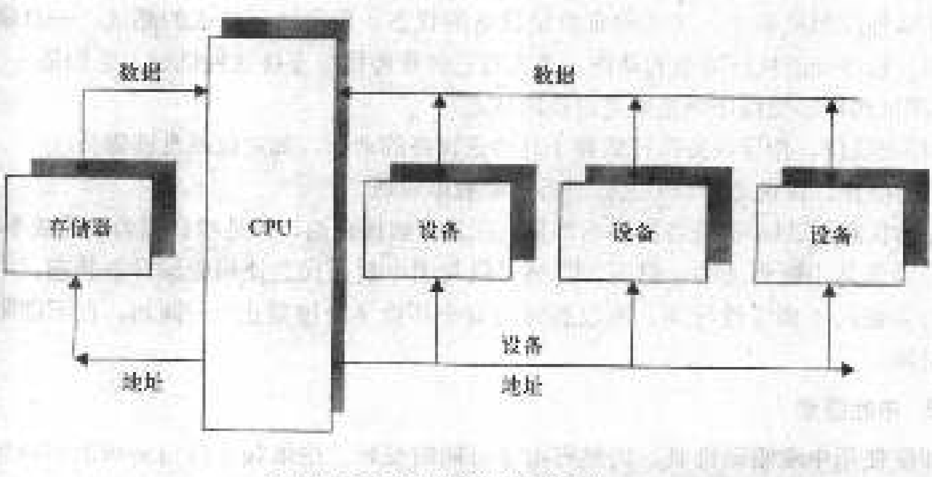


图15-1 有各自总线的体系结构

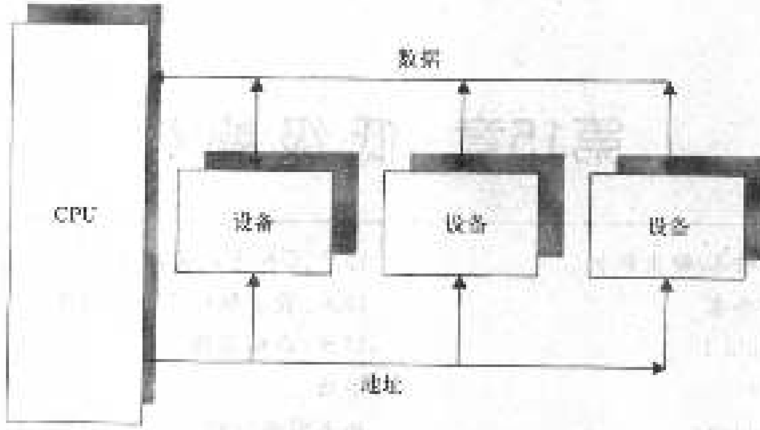


图15.2 存储-映射式体系结构

设备接口通常是通过一组寄存器。对于存储器和I/O设备逻辑上各有一条总线的方式，计算机必须有两组汇编指令：一组访问存储器，另一组访问设备寄存器，后者通常采取如下形式：

```
IN    AC,  PORT
OUT   AC,  PORT
```

578

其中，IN把由PORT标识的设备寄存器的内容读到累加器AC，OUT将累加器的内容写入设备寄存器（术语PORT在这里用于指明I/O设备总线上的一个地址）。读取设备状态可能还有其他指令。Intel 486和Pentium就是允许通过特定指令访问设备的体系结构的例子。

对于设备位于同一逻辑总线上的情况，某些地址访问存储器位置，另一些地址访问设备。这种方法被称为**存储-映射式I/O**（memory-mapped I/O）。Motorola M68000和PowerPC系列的计算机都有存储-映射式I/O。

为了控制设备的操作（例如，设备初始化和设备准备传送数据）和控制数据传送（例如，设备开始进行或是进行数据传送），都必须与设备对接。可以描述两种通用的执行和控制输入输出的机制：状态驱动控制机制和中断驱动控制机制。

15.1.1 状态驱动

在这种控制机制中，为了确定给定设备的状态，程序执行显式的测试。一旦确定了设备的状态，程序就能执行适当的动作。通常有三种硬件指令支持这种机制，它们是：

- 测试操作，使程序确定给定设备的状态；
- 控制操作，指导设备执行依赖于非传送设备的动作，如定位磁盘读磁头；
- I/O操作，在设备和CPU之间进行实际数据传送。

虽然状态驱动I/O的设备价格不昂贵，在几年前很普遍，但是现在因为硬件成本不断下降，大部分设备是中断驱动的。然而，当然可以关闭中断而代之使用轮询设备状态。中断也增加了实时系统的不确定性行为，所以有时在安全环境中会被禁止——例如，在不能限制中断数量的时候。

15.1.2 中断驱动

即使使用中断驱动机制，仍然有很多可能的变异，这依赖于传送必须怎样初始化和怎样

579 控制。其中的三种变异是：中断驱动程序控制、中断驱动程序启动和中断驱动通道程序控制。

1. 中断驱动程序控制

这里，设备的中断请求是作为遇到某些事件（例如数据到达）的结果。当请求被确认的时候，处理器挂起正在执行的进程，调用指定的中断处理进程，这个进程执行适当的动作去响应中断。当中断处理进程完成它的功能时，处理器的状态被恢复到中断前的状态，然后处理器的控制权归还给先前被挂起的进程。

2. 中断驱动程序启动

这种类型的输入/输出机制通常被称为是**直接存储器访问**（或是DMA）。DMA设备位于输入/输出设备和主存储器之间。在I/O设备和存储器之间传送数据方面，DMA设备取代了处理器。虽然I/O是通过程序启动的，但是DMA设备控制实际的数据传送（每次一块）。对于每一段要被传送的数据，DMA设备在每个存储周期产生一个请求，当请求被接受后就开始传送。当整块的数据传送完毕后，DMA设备产生一个传送完毕中断。这是由中断驱动程序控制机制处理的。

术语**周期窃取**常常用于指DMA设备对访问存储器产生的影响。它可能导致不确定性的行为，并使得很难计算程序的最坏情况执行时间（参看13.7节）。

3. 中断驱动通道程序控制

通过尽可能地减少中央处理器介入处理输入/输出设备，通道程序控制的输入/输出扩展了程序启动输入/输出的概念。这种机制由三种主要的部件组成：硬件通道和连接设备、通道程序（通常被称为"脚本"）和输入/输出指令。

硬件通道操作包括以上所说的DMA设备的操作。此外，按照通道程序的指令，通道指示设备控制操作。通道程序的执行从应用程序里面启动。一旦通道被命令执行通道程序，选定的通道和设备就独立于中央处理器执行，直到指定的通道程序结束，或者是发生了某个异常情况。通道程序通常由一个或多个通道控制字组成，这些控制字每次由通道解码并执行一个控制字。

在很多方面，硬件通道可以被看作是一个与中央处理器共享存储器的自治进程。它对中央处理器的存储器访问次数的影响是不可预测的（同使用DMA设备一样）。 580

15.1.3 中断驱动设备所需的要素

从上一节可以看出，中断在控制输入/输出中的作用是非常重要的。它使输入/输出得以异步执行，从而避免了"忙等待"或是不断的状态检测（如果使用纯粹的状态控制机制，这种检测就是必需的）。

为了支持中断驱动的输入和输出，需要以下机制：

1. 上下文切换机制

当中断发生时，必须保存当前处理器的状态，然后启动适当的服务例程。一旦中断服务完成，先前的进程被恢复并继续执行。或者是，由于中断，调度程序选择了一个新的进程继续执行。整个的这种过程被称为**上下文切换**，它的动作可以概括如下：

- 立即保存中断发生前的处理器状态；
- 安排处理器到所需状态，以处理中断；
- 中断处理结束后，恢复挂起进程的状态。

在处理器上执行的进程的状态包括：

- 在执行序列中当前（或下一条）指令的存储器地址；

- 程序状态信息（可能包括关于处理方式、当前优先级、存储器保护、允许的中断等信息）；
- 可编程寄存器的内容。

提供的上下文切换类型可以用进程状态保护的程度和恢复程度（这是通过硬件完成的）刻画。可以区别三种级别的上下文切换：

- 基本的——只保存程序计数器，加载新的上下文；
- 部分的——保存程序计数器和程序状态寄存器，加载新的上下文；
- 完全的——进程所有的上下文都被保存，加载新的上下文。

依赖于处理器状态保存程度的需求，可能需要通过显式软件支持对硬件动作进行增补。例如，局部上下文切换对于将处理程序看作过程或子程序的中断处理模型来说可能足够了。
581 然而，如果处理程序被看作是一个有自己的栈和数据区的单独的进程，就需要有一个在中断进程和处理进程之间进行完全上下文切换的低级处理程序。另一方面，如果硬件承担了完全上下文切换，就不需要这样的低级处理程序。大部分处理器只提供部分上下文切换。然而ARM处理器确实提供了快速中断，这时，还保存一些通用寄存器。

需要注意的是，一些现代的处理器允许指令获取、解释和执行的同步进行。有些处理器还允许指令不按照程序指定的顺序执行。本章假设中断是**精确的**（precise）（Walker and Cragon，1995），它的意思是，当中断处理程序执行时：

- 中断指令前发布的所有指令必须执行完，并正确地修改程序状态；
- 中断指令后发布的指令没有执行，没有程序状态被改变；
- 如果是中断指令本身引起中断（例如，造成存储器破坏的指令），那么要么执行完这条指令，要么根本不执行。

如果中断是不精确的，就假设恢复对于中断处理软件是透明的。参阅参考文献[Walker and Cragon（1995）]中关于中断处理的分类。

2. 中断设备的标识

设备在被控制的方法上有所不同，因此它们需要不同的中断处理例程。为了调用适当的处理程序，必须有一些标识中断设备的手段。可以区分出四种中断设备的标识机制：向量化、状态、轮询和高级语言原语。

标识中断设备的**向量化**机制是由一组专门的（通常连续的）、称为**中断向量表**的存储位置和设备地址到中断向量的一个硬件映射组成的。中断向量表可以被一个特定的设备使用，也可以被几个设备共同使用。程序员必须将特定的中断向量表的位置显式地与中断服务例程关联起来。通过将向量表的位置设置为服务例程的地址、或设置为引起转移发生所需例程的指令而做到这一点。通过这个方法，服务例程直接与中断向量表位置连接在一起，这又进而间接地连接到一个设备或是一组设备上。因此，当一个特殊的服务例程被调用并执行时，中断设备也就隐含地被识别出来了。

标识中断设备的**状态**机制，被用于几个设备连接到一个设备控制器的机器配置，并且这些机器配置没有各自的中断向量表的情况。这种机制也常用于一个广义的服务例程将要开始
582 处理所有中断的情况。使用这种机制，每个中断有一个相关的状态字，它指明引起中断的设备和中断的原因（还有其他的东西）。对于一个给定的中断或中断类，状态信息可以由专门存储位置中的硬件自动提供，或者可以通过一些适当的指令去检索。

轮询设备标识机制是所有这些里面最简单的一种。当中断发生时，通用的中断服务例程被调用去处理中断。为了确定是哪个设备请求中断，要轮询每一个设备的状态。当中断设备被标识后，再用适当的方法对中断进行服务。

在一些现代的计算机体系结构中，中断处理直接与一个**高级语言原语**相关联。在这些系统中，中断通常被看作是到相关通道的同步消息。在这种情况下，设备由那个变成主动的通道来标识。

3. 中断识别

一旦设备被识别了，适当的中断处理例程就必须确定它为什么产生中断。通常，这可以通过设备提供的状态信息得到，或是通过弄清楚同一设备通过不同的向量位置或通道发生的不同中断得到。

4. 中断控制

一旦设备接通并启动了，虽然它可以产生中断，但是中断通常被忽视，除非这个设备处于允许中断状态。中断的这种控制（允许/禁止）通过下面的中断控制机制实现。

中断状态控制机制提供标记（flag）以允许和禁止中断，这种标记或者在中断状态表中，或者是通过设备和程序状态字。标记可以通过常规的、面向位的指令或是专门的位测试指令访问（也可以修改）。

中断屏蔽控制机制将每个设备中断和一个字中特定的位位置联系起来。如果这个位被设定为1，则中断被禁止；如果设为0，则中断被允许。中断屏蔽字可以通过常规的面向位的（或面向字的）指令寻址或只能通过特定的中断屏蔽指令访问。

级别中断控制机制将设备和确定的级别联系起来。当前处理器的级别确定哪一个设备可以或不可以中断。只有那些有较高逻辑级别的设备可以中断。如果最高逻辑级是活动的，那就只允许那些不能被禁止的中断（例如电源失效）。这并不明确地禁止中断，所以处于比当前处理器级别低的逻辑级的中断还是可以产生的，并且当处理器的级别适当降低的时候，也不用重新使之被允许。 583

5. 优先级控制

有些设备比其他设备更紧急，因此经常把优先级设施与中断关联起来。这种机制可以是静态的也可以是动态的，并且通常与设备中断控制设施和处理器的优先级有关系。

15.1.4 一个简单的I/O系统的例子

为了阐明I/O系统的各种成分，这里描述了一个简单的机器。它不严格地基于摩托罗拉68000系列的计算机。

机器上支持的每种设备都有它的操作必需的、很多不同类型的寄存器。这些寄存器是存储-映射式的。最常用的类型就是包含了所有设备状态信息，并能表示允许（enable）和禁止（disable）设备中断的**控制和状态**寄存器。**数据缓冲区**寄存器是作为通过设备传送到机器内或发送出去的临时存储数据的缓冲区寄存器。

计算机的典型控制和状态寄存器的结构如下：

```
位
   15 - 12  :  出错        -- 设备出错时置1
   11       :  忙          -- 设备忙时置1
   10 - 8   :  单元选择    -- 有多个设备需要被控制
```

```
7       :  做完/就绪        --  I/O完成或设备就绪
6       :  允许中断        --  置1时允许中断
5 - 3   :  保留            --  为未来使用保留
2 - 1   :  设备函数        --  置1指出所需函数
0       :  设备允许操作    --  置1指出设备允许操作
```

面向字符的设备使用的数据缓冲区寄存器的典型结构是:

```
位
15 - 8 : 未用
 7 - 0 : 数据
```

第0位是寄存器最低有效位。

对于一个设备，这些寄存器每种可能有不止一个，确切的数目依赖于设备的种类。例如，一个特定的Motorola并行接口和定时器设备有14个寄存器。

中断使设备能在需要服务时通知处理器，它们是向量化的。当中断发生时，处理器将当前程序计数器（PC）和当前处理器状态字（PSW）存储到系统栈中。PSW还包含了处理器的优先级（还有别的东西）。它的实际格式如下:

```
位
15 - 11      : 模式信息
10 -  8      : 未用
 7 -  5      : 优先级
 4 -  0      : 条件码
```

584 条件码包含了处理器最后操作的结果信息。

新的PC和PSW是从两个预先分配的连续的存储位置（中断向量）加载的。第一个字包含了中断服务例程的地址，第二个包含了PSW，包括中断将被处理的优先级。一个低优先级的中断处理程序可以被一个更高优先级的中断打断。

稍后会再回到这个I/O系统的例子。

15.2 语言要求

如上所述，嵌入式系统的一个主要特征就是需要与输入输出设备交互，这些设备都有自己具体的特性。这些设备的编程传统上是汇编程序员的根据地，但是像C、Modula-1、Java、occam2和Ada等语言现在也尝试对这些低级功能提供日益增多的高级机制。这使得设备驱动程序和中断处理例程更易于阅读、编写和维护。然而，主要的难点在于：为了方便地编写可用的设备处理程序，要决定在高级语言中需要哪些功能。虽然这个主题还不成熟，但**封装设施**和设备处理的**抽象模型**可以被认为是特有的需求。

15.2.1 模块性和封装设施

低级设备的接口必然是依赖于机器的，因此通常是不可移植的[⊖]。

在任何软件系统中，将不可移植的代码与可移植的代码分开是很重要的。只要可能，建议将所有依赖机器的代码封装到一个可明确标识出来的单元中。在Modula-1中，与设备相关的代

⊖ 注意近些年有产生一种统一驱动程序接口（Uniform Driver Interface, UDI）的尝试。UDI定义了一个体系结构和一组操作系统中立（和硬件中立）的接口，以供在设备驱动程序和外围系统之间使用。这样就使设备驱动器和操作系统能相互独立地开发，并且多种操作系统和硬件平台可以使用相同的驱动程序。

码必须被封装到一个被称为**设备模块**的专门类型的模块中。在Ada中，使用了包和保护类型功能。在Java中，类和包是合适的封装机制。在occam2中，仅有的设施是过程；在C中是文件。

15.2.2 设备处理的抽象模型

一个设备可以被认为是一个执行固定任务的处理器。因而，一个计算机系统可以被认为是几个并行的进程。设备“进程”与在主处理器内部执行的进程进行通信和同步的模型有几种。所有的模型都必须提供:

1）**表示、寻址和操纵设备寄存器的设施** 585

设备寄存器可以表示为程序变量、对象、甚至是通信通道。

2）**适当的中断表示**

可能有以下表示:

a）**过程**

中断被看作是过程调用（在某种意义上，它是来自设备进程的远程过程调用）。所需的任何通信和同步必须编写到中断处理过程中。这种过程不嵌套：只能访问全局状态或处理程序的局部状态。

b）**偶发进程**

中断被看作是执行进程的请求。处理程序是偶发的进程，可以访问局部持续数据和全局数据（如果在并发模型中可用共享变量通信的话）。

c）**异步通知**

中断被看作是一个直接发向进程的异步通知。处理程序可以访问进程的局部状态和全局状态。恢复模型和终止模型都是可以的。

d）**共享变量条件同步**

中断被看作是在共享变量同步机制下的条件同步，例如，在管程中信号量上的发信号操作或在条件变量上的发送操作。处理程序可以访问进程/管程的局部状态和全局状态。

e）**基于消息的同步**

中断被看作是发送到通信通道的无内容的消息。接收进程可以访问进程的局部状态。

除了过程方法外，以上所有的方法要求完全的上下文切换，因为处理程序在进程的作用域内执行。如果处理程序是受限制的，可以进行优化。例如，如果在异步通知模型中的处理程序有恢复语义，并且不访问进程的任何局部数据，那么中断就可以仅通过局部的上下文切换来处理。

并不是所有这些模型都可以在实时语言和操作系统中找到。最流行的是过程模型，因为它只需要很少的支持。通常用C和C++中实现的实时系统采用这种模型，并且将设备寄存器表示成程序变量。对于顺序系统，异步事件模型实际上和过程模型相同，因为只有一个进程被中断，因而不需要识别进程或事件。Ada模型是过程模型和共享变量条件同步模型的混合体。中断被映射成保护过程，寄存器通过程序变量访问。实时Java认为中断是异步事件，处理程序是可调度的对象。 586
Modula-1和实时Euclid分别将中断映射到条件变量和信号量（寄存器再次被表示为程序变量），因而是纯粹的共享变量模型。occam2认为中断是发送到通道的消息，设备寄存器也被表示为通道。

以下详细研究五种语言：Modula-1、Ada、实时Java、occam2和C。

15.3 Modula-1

Modula-1是最先具有设备驱动程序编程功能的高级编程语言之一。

在Modula-1中，模块化和封装的单位是模块。有一种称为**接口模块**的专门模块用于控制对共享资源的访问，它具有管程的特性。进程通过信号（条件变量）进行交互，使用WAIT、SEND和AWAITED（参看第8章）操作符。第三种模块称为**设备模块**，是一种专门类型的接口模块，用于封装与设备的交互。仅在设备模块里面可以使用用于处理中断的设施。

15.3.1 设备寄存器的寻址和操纵

将变量与寄存器关联起来是相当简单的。在Modula-1中，在声明中用名字后面跟着的一个八进制的地址表示。例如，对于在15.1.4节中介绍的简单I/O体系结构的数据缓冲区寄存器就可定义成：

```
var rdbr[177562B] char;
```

这里，177562B表示一个八进制的地址，它就是寄存器在存储器中的位置。

字符到字符缓冲区寄存器的映射也是一件简单的事，因为这种类型没有内部结构。控制和状态寄存器就更有趣了。在Modula-1中，只有标量数据类型可以被映射到设备寄存器，因此有内部结构的寄存器被认为是预先定义的*bits*类型，它的定义是：

```
TYPE BITS = ARRAY 0:no_of_bits_in_word OF BOOLEAN;
```

这种类型的变量被打包为一个单独的字。所以在八进制地址177560B的控制和状态寄存器可以使用下面的Modula-1代码定义：

```
VAR rcsr[177560B] : BITS;
```

587 为了访问寄存器中的各种字段，程序员提供了数组的序标。例如，以下的代码将启用设备：

```
rcsr[0] :=TRUE;
```

下面的代码将关闭中断：

```
rcsr[6] := FALSE;
```

通常，这些设施在能力上不足以方便地处理所有类型的寄存器。通用的控制和状态寄存器的结构前面已给出：

```
位
  15 - 12  :  出错
  11       :  忙
  10 - 8   :  单元选择
   7       :  做完/就绪
   6       :  允许中断
   5 - 3   :  保留
   2 - 1   :  设备函数
   0       :  设备允许操作
```

使用布尔变量设置选择单元（位8~10）是很笨拙的方法——例如，用以下语句将设备单元设置为值5。

```
rcsr[10] := TRUE;
rcsr[9]  := FALSE;
rcsr[8]  := TRUE;
```

值得注意的是，在很多机器上有不止一个设备寄存器映射到相同的物理地址。因此，几个变量可以被映射到存储器中的相同位置。此外，这些寄存器经常是只读或只写的。因此，操纵设备寄存器时必须小心。在上面的例子中，如果控制和状态寄存器是一对映射到相同位置的寄存器，那么给出的代码可能达不到预期的效果。这是因为设置特定的位需要产生将当前的值读入到机器累加器的代码。因为控制寄存器是只写的，这会产生状态寄存器的值。因此，建议在程序中用其他的变量表示设备寄存器。这些可以用正规的方法进行操纵。当要求的寄存器的格式已经构建好的时候，就可以分配给实际的设备寄存器了。这些变量通常被称为**影子设备寄存器**（shadow device register）。

15.3.2 中断处理

Modula-1中处理中断的设施是基于理想硬件设备的观念。这个设备必须具有以下属性（Wirth，1997a）：

- 对于每项设备操作，产生多少中断是已知的。
- 当中断发生后，设备的状态指明是否将要发生另一个相关的中断。 588
- 没有意料外的中断到达。
- 每一个设备都有惟一的中断位置。

Modula-1提供的设施可以总结为如下几点：

- 每个设备有一个相关的设备模块。
- 每个设备模块在紧跟模块名字后面的首部规定了一个硬件优先级。
- 模块内所有的代码都在指定的硬件优先级上执行。
- 设备模块内每个要处理的中断都需要一个称为**设备进程**的进程。
- 当设备进程执行时，它独占地访问这个模块（就是说，它使用在设备模块首部指定的高限优先级拥有管程锁）。
- 不允许设备进程调用任何非局部的过程，也不能发送信号到其他设备进程。这是为了确保设备进程不会被无意地阻塞。
- 当设备进程发送信号时，发送操作的语义不同于那些普通的Modula-1进程，在这种情况下，接收进程是不恢复的，但是发信号的进程要继续。这又是为了确保进程不会被阻塞。
- 在设备进程中的`WAIT`语句是惟一的“一号”（最高级）。
- 中断被看作是信号的一种形式，然而设备进程不是发布`WAIT`请求，而是发布`DOIO`请求。
- 在进程首部中规定了用以进行设备中断的向量的地址。
- 只有设备进程可以含有`DOIO`语句。
- `DOIO`和`WAIT`调用降低处理器的优先级，因此释放管程锁。
- 一个设备进程只有一个实例可被激活。

例如，考虑一个在15.1.4节中概述的简单机器体系结构处理实时时钟的设备模块。当收到中断时，处理程序发送信号到等待时钟滴答的那个进程。

```
DEVICE MODULE rtc[6];  (* 硬件优先级 6 *)

    DEFINE tick;
    VAR tick : SIGNAL;

    PROCESS clock[100B];
        VAR csr[177546B] : BITS;
```

```
589        BEGIN
               csr[0] := TRUE;    (* 允许设备操作 *)
               csr[6] := TRUE;    (* 允许中断 *)
               LOOP
                   DOIO;
                   WHILE AWAITED(tick) DO
                       SEND(tick);
                   END
               END
           END;
       BEGIN
           clock; (* 创建clock进程的一个实例 *)
       END rtc;
```

设备模块的首部指定了中断优先级为6，模块内所有的代码都在这个优先级执行。进程首部上的100B这个值指明设备将通过在地址（八进制）100的向量中断。允许中断后，设备进程进入等待中断的简单循环（DOIO）然后发送足够的信号（就是说，每个等待进程一个）。注意当设备进程发送信号时，它并不放弃对模块的互斥访问，而是一直继续，直到执行了一条WAIT或DOIO语句。

以下说明Modula-1是怎样处理在15.1.2和15.1.3节中概述的中断驱动设备的一般特征的。

- **设备控制**——I/O寄存器由变量表示。
- **上下文切换**——中断引起到中断处理进程的立即上下文切换，该进程使用DOIO等待。
- **中断设备标识**——中断向量的地址与设备进程首部一起给出。
- **中断标识**——在上述的例子中，只可能有一个中断。然而，在其他的情况下，应该检测设备状态寄存器以确定中断的原因。
- **中断控制**——中断控制是状态驱动的，并且是通过设备寄存器中的标记提供的。
- **优先级控制**——设备的优先级在设备模块首部给出。模块中的所有代码在这个优先级运行（就是说，设备模块有硬件高限优先级，并以立即优先级高限协议执行（见13.11节））。

15.3.3 一个终端驱动程序的例子

为了更进一步举例说明Modular-1设备驱动的方法，介绍一个简单的终端设备模块。这个终端有两个部件：一个显示器和一个键盘。每个部件有一个相关的控制和状态寄存器、一个
590 缓冲区寄存器和一个中断。

在程序中提供了两个过程，使程序的其他进程能读写字符。这些过程访问一个有界缓冲区，它使输入的字符能在此缓冲并输出。这些缓冲区必须包含在设备模块中，因为设备进程不能调用非局部的过程。虽然对显示器和键盘已经使用了单独的模块，但是它们还是被组合起来说明一个设备模块可以处理不止一个中断。

```
DEVICE MODULE terminal[4];

  DEFINE readch, writech;

  CONST n=64;                 (* 缓冲区大小 *)

  VAR KBS[177560B] : BITS; (* 键盘状态 *)
      KBB[177562B] : CHAR; (* 键盘缓冲区 *)
      DPS[177564B] : BITS; (* 显示器状态 *)
      DPB[177566B] : CHAR; (* 显示器缓冲区*)
```

```
    in1, in2, out1, out2 : INTEGER;
    n1, n2 : INTEGER;
    nonfull1, nonfull2,
    nonempty1, nonempty2 : SIGNAL;
    buf1, buf2 : ARRAY 1:n OF CHAR;
PROCEDURE readch(VAR ch: CHAR);
BEGIN
  IF n1 = 0 THEN WAIT (nonempty1) END;
  ch := buf1[out1];
  out1 := (out1 MOD n)+1;
  DEC(n1);
  SEND(nonfull1)
END readch;
PROCEDURE writech(ch : CHAR);
BEGIN
  IF n2 = n THEN WAIT(nonfull2) END;
  buf2[in2] := ch;
  in2 := (in2 MOD n)+1;
  INC (n2);
  SEND(nonempty2)
END writech;


PROCESS keyboarddriver[60B];
BEGIN
  KBS[0] := TRUE;  (* 允许设备操作 *)
  LOOP
    IF n1 = n THEN WAIT(nonfull1) END;
    KBS [6] := TRUE;
    DOIO;
    KBS [6] := FALSE;
    buf1[in1] := KBB;
    in1 := (in1 MOD n)+1;
    INC(n1);
    SEND(nonempty1)
  END
END keyboarddriver;
PROCESS displaydriver[64B];
BEGIN
  DPS[0] := TRUE;  (*允许设备操作*)
  LOOP
    IF n2 = 0 THEN WAIT(nonempty2) END;
    DPB := buf2[out2];
    out2 := (out2 MOD n)+1;
    DPS [6] := TRUE;
    DOIO;
    DPS [6] := FALSE;
    DEC(n2);
    SEND(nonfull2)
```

```
    END
  END displaydriver;

BEGIN
  in1 :=1; in2 := 1;
  out1 := 1; out2 :=1;
  n1 :=0; n2 := 0;
  keyboarddriver;
  displaydriver
END terminal;
```

1. 时间性设施

Modula-1不提供操纵时间的直接设施，这些必须由应用程序提供。这需要一个设备模块处理时钟中断并定期（例如，每秒）发布一个信号。现在给出这个模块，它是先前定义的模块的一个修改版本。假定硬件时钟每五十分之一秒滴答一下。

```
DEVICE MODULE hardwareclock[6];
    DEFINE tick;
    VAR tick : SIGNAL;

    PROCESS handler[100B];
        VAR count : INTEGER;
            statusreg [ 177546B] : BITS;
    BEGIN
        count := 0;
        statusreg[0] := TRUE;
        statusreg[6] := TRUE;
        LOOP
            DOIO;
            count := (count+1) MOD 50;
            IF count = 0 THEN
                WHILE AWAITED (tick) DO
                    SEND(tick)
                END
            END
        END
    END handler;
BEGIN
    handler
END hardwareclock;
```

591 ~ 592

现在可以很容易地提供一个维持一天时间的接口模块。

```
INTERFACE MODULE SystemClock;
    (* 定义获取和设置一天的时间的过程 *)
    DEFINE GetTime, SetTime;

    (* 导入抽象数据类型time和tick信号 *)
    USE time, initialise, add, tick;

    VAR TimeOfDay, onesec : time;

    PROCEDURE SetTime(t: time);
    BEGIN
        TimeOfDay := t
```

```
    END SetTime;

    PROCEDURE GetTime(VAR t: time);
    BEGIN
        t := TimeOfDay
    END GetTime;

    PROCESS clock;
    BEGIN
        LOOP
            WAIT(tick);
            addtime(TimeOfDay, onesec)
        END
    END clock;
BEGIN
    inittime(TimeOfDay, 0, 0, 0);
    inittime(onesec, 0, 0, 1);
    clock;
END SystemClock;
```

注意时钟进程逻辑上是多余的。设备进程可以直接增加`TimeOfDay`，从而节省上下文切换。然而，在Modula-1中不允许设备进程调用非局部过程。

2. 延迟进程

在实时系统中，经常需要延迟进程一段时间（参看12章）。虽然Modula-1没有直接实现这一点的设施，但可以编程实现。这留给读者作为练习（参看练习15.6）。 593

15.3.4 Modula-1设备驱动方法的问题

Modula-1被设计成用于攻击汇编语言编程的根据地——设备接口。总体上，它被认为是很成功的，然而，还是有一些针对它的功能的批评。

- Modula-1不允许设备进程调用非局部过程，因为设备进程必须尽可能地保持小，并且必须在设备的硬件优先级上执行。调用定义在其他模块中的过程，而其实现又向进程隐藏，这可能会导致无法接受的延迟。Modula-1又进一步要求这些过程在设备的优先级执行。遗憾的是，由于这种限制，程序员要么必须将那些不是直接与驱动设备有关的额外的功能合并到设备模块中（如在终端驱动程序的例子中，有界缓冲区被包含在设备模块中），要么不得不引入额外的进程等待设备进程发送信号。在前一种情况下，这会导致设备模块非常大，而对后者，是不必要的低效。
- Modula-1只允许设备进程的一个实例，因为进程首部含有将进程与中断关联起来必需的信息。这使得在相似设备间共享代码更加困难，这个问题又由于不能调用非局部过程而更难。
- Modula-1是为存储－映射式计算机设计的，因此，很难对由专门指令控制的可编程设备使用它的设施。然而，容易设想一个简单的扩展去解决这个问题。Young（1982）建议使用下列记号的可能性：

```
VAR x AT PORT 46B : INTEGER;
```

当端口被编址时编译器就可以把这种端口识别出来，并可以产生正确的指令。

- 前面已经指出许多设备寄存器是只读或只写的。在Modula-1中不可能定义只读或只写的

变量。此外，这里有一个隐含的假设，即编译器将不优化对设备寄存器的访问，并在局部寄存器中高速缓存它们。

15.4 Ada

在Ada中，任务间相互同步和通信有三种方法：
- 通过会合；
- 使用保护单元；
594 • 通过共享变量。

通常，Ada假设设备和程序能够访问共享存储设备寄存器，这些寄存器可用其表示规格说明技术规定。在Ada 83中，中断通过硬件产生的任务入口调用表示。在Ada当前的版本中，这项功能被认为是过时的，并且可能从该语言的下一个版本中删除。因此在这本书中不做讨论。

设备驱动的首选方法是在保护单元中封装设备操作。可以由保护过程调用处理中断。

15.4.1 设备寄存器的寻址和操作

Ada向程序员展示了规定数据类型的实现的一组全面设施。这些统称为**表示子句**，它们指出语言的类型怎样被映射到底层硬件。一种类型只能有一个表示。表示是和类型的逻辑结构分开指定的。当然，类型表示的规格说明是可选的，也可以委托给编译器。

表示子句是抽象和具体结构之间的折中。有四种不同的规格说明：

1）**属性定义子句**——允许设置对象、任务或子程序的各种属性，例如，对象的大小（按位）、存储对齐、任务的最大存储空间、对象的地址。

2）**枚举表示子句**——给枚举类型的字面值以具体的内部值。

3）**记录表示子句**——记录成分可以在单个的存储单元内部被分配偏移和长度。

4）**At子句**——这是Ada 83安置对象到特定地址的主要机制，这项功能现在已经过时了（可以使用属性），因此将不再进一步讨论。

如果一个实现不能服从规格说明的要求，那么编译器要么拒绝程序，要么在运行时引发异常。

为了举例说明这些机制的使用，考虑以下的类型声明，它们表示了前面定义的简单机器的典型控制和状态寄存器。

595

```
type Error_T is (None, Read_Error, Write_Error,
                 Power_Fail, Other);
type Function_T is (Read, Write, Seek);
type Unit_T is new Integer range 0..7;

type Csr_T is record
    Errors    : Error_T;
    Busy      : Boolean;
    Unit      : Unit_T;
    Done      : Boolean;
    Ienable   : Boolean;
    Dfun      : Function_T;
    Denable   : Boolean;
end record;
```

枚举表示子句为枚举类型的字面值指定内部代码。例如，上述设备所需函数的内部代码

可以是:

```
01  -- READ
10  -- WRITE
11  -- SEEK
```

在Ada中这被指定为:

```
type Function_T is (Read, Write, Seek);
for Function_T use (Read=>1, Write=>2, Seek=>3);
```

类似地，对于Error_t:

```
type Error_T is (None, Read_Error, Write_Error, Power_Fail, Other);
for Error_T use (None =>0, Read_Error =>1, Write_Error =>2,
                 Power_Fail =>3, Other =>4);
-- 注意，这实际上是默认赋值
```

记录表示子句指定记录的存储表示，即它的成分的次序、位置和大小。记录的位从0开始计算，在成分子句中的范围规定分配的位数。

例如，控制状态寄存器为:

```
Word : constant :=2;  -- 一个字中存储单元的个数
Bits_In_Word : constant :=16;  -- 字中的二进制位数
for Csr_T use
  record
    Denable  at 0*Word range 0..0;  -- 在第0字的第0位
    Dfun     at 0*Word range 1..2;
    Ienable   at 0*Word range 6..6;
    Done     at 0*Word range 7..7;
    Unit     at 0*Word range 8..10;
    Busy     at 0*Word range 11..11;
    Errors   at 0*Word range 12..15;
end record;

for Csr_T' Size use Bits_In_Word; -- Csr类型对象的大小
for Csr_T' Alignment use Word; -- 对象应当字对齐
for Csr_T' Bit_order use Low_Order_First;
-- 首位是字节的最低有效位
```

596

大小属性（Size）规定与一个类型关联的存储量。在这个例子中，寄存器是一个单独的16位字。对齐（Assignment）属性指示编译器应当总是将对象放在整数存储单元边界上，在这里是字的边界。位序（Bit_order）属性指定机器将0位作为最高有效位（大尾数）或最低有效位（小尾数）。注意，第3、4和5位（被保留为以后使用）没有被指定。

最后，需要声明一个实际的寄存器，并把它放在所要求的存储器位置上。在Ada中，Address是在包System中定义的实现定义的类型。子包（System.Storage_Elements.To_Address）提供了将整型变量转换为地址类型的功能。

```
Tcsr : Csr_T;
for Tcsr' Address use System.Storage_Elements.To_Address(8#177566#);
```

构建了寄存器的抽象数据表示，并在正确的地址放置一个适当定义的变量之后，就可以通过对这些变量赋值操纵硬件寄存器了:

```
Tcsr := (Denable => True, Dfun => Read,
         Ienable => True, Done => False,
         Unit => 4, Errors => None);
```

使用这个记录聚集要假设整个寄存器同时被赋值。为了确保Dfun没有在记录的其他域之前被设置，必须使用一个临时（影子）控制寄存器：

```
Temp_Cr : Csr_T;
```

这个临时寄存器被赋为控制值并拷贝到实际的寄存器变量：

```
Tcsr := Temp_Cr;
```

这个赋值代码在大多数情况下都确保整个控制寄存器由一个单独的动作更新。如果还存在疑问，可以使用编用Atomic（它指示编译器作为一个简单操作产生这个更新或者产生一个出错信息)。

I/O操作完成后，设备自身可以改变寄存器的值，这在程序中被看作是记录成分的值的改变：

```
if Tcsr.Errors = Read_Error then
  raise Disk_Error;
end if;
```

因此，对象Tcsr是一个共享变量的集合，它被设备控制任务和设备自身共享。两个并发（和
597 并行）的进程间的互斥对于提供可靠性和性能是必需的。这一点在Ada中是通过使用保护对象实现的。

15.4.2 中断处理

Ada定义了以下的中断模型：

- 中断表示由硬件或系统软件检测到的一类事件。
- 中断的**出现**由它的**生成**和**交付**组成。
- 中断的生成是在底层硬件或系统中使得中断对于程序成为可用的事件。
- 交付是响应中断发生而调用一段程序（称之为中断处理程序）的动作。在中断生成和交付之间，称为中断被**悬挂**（pending)。对中断的每次交付，都要调用处理程序一次。中断的**潜伏期**（latency）是指在悬挂状态消耗的时间。
- 当中断被处理时，从同一个源来的其他中断被**阻塞**；防止交付所有未来发生的中断。被阻塞的中断是否保持悬挂或丢弃通常是依赖于设备的。
- 某些中断是被**预留**（reserved）的。不允许程序员对预留的中断提供处理程序。通常，预留中断直接由Ada的运行时支持系统处理（例如，用于实现延时语句的时钟中断）。
- 每个非预留中断有一个由运行时支持系统指派的默认处理程序。

使用保护过程处理中断

在Ada中，中断处理程序的主要表示是一个无参保护过程。每个中断有一个由系统支持的惟一的显然有别的标识符。这个惟一标识符如何表示是由实现定义的，例如，它可能是与中断相关的硬件中断向量的地址。当Ada在一个支持信号的操作系统上实现时，每个信号可能被映射到特定的中断标识符，因而使信号处理程序能用Ada编程。

标识中断处理保护过程是通过使用两个编用之一实现的：

598
```
pragma Attach_Handler(Handler_Name, Expression);
  -- 这可以出现在一个库级保护对象的规格说明或体中，并允许将
```

```
    -- 一个已命名的处理程序同由此表达式标识的中断静态关联起来;
    -- 在创建这个保护对象时，该处理程序就附加上了。
    -- 在下列情况引发Program_Error:
    --   (a) 当保护对象被创建，而该中断被预留的时候;
    --   (b) 如果该中断已经有了一个用户定义的处理程序;
    --   (c) 如果定义的高限优先级不在Ada.Interrupt_Priority的范围之内。
  pragma Interrupt_Handler(Handler_Name);
    -- 这可以出现在一个库级保护对象的规格说明中，并允许将
    -- 一个已命名的无参过程同一个或多个中断的中断处理程序
    -- 动态关联起来。从这样一个保护类型创建的对象必须是库级的。
```

程序15-1定义了系统编程附件对中断标识符和处理程序的动态附加的支持。在引发Program_Error的所有情形中，当前附加的处理程序没有被改变。

程序15-1 包Ada.Interrupts

```
package Ada.Interrupts is
  type Interrupt_Id is 实现定义的; -- 必须是离散类型
  type Parameterless_Handler is access protected procedure;

  function Is_Reserved(Interrupt : Interrupt_Id) return Boolean;
    -- 如果中断是预留的，返回True,
    -- 否则返回 False
  function Is_Attached(Interrupt : Interrupt_Id) return Boolean;
    -- 如果中断是附加于一个处理程序的，返回True,
    -- 否则返回 False
    -- 如果中断是预留的，引发Program_Error
  function Current_Handler (Interrupt : Interrupt_Id)
                            return Parameterless_Handler;
    -- 返回指向该中断的当前处理程序的一个访问变量。
    -- 如果没有附加用户处理程序，返回一个表示默认处理程序的值。
    -- 如果中断是预留的，引发Program_Error
  procedure Attach_Handler(New_Handler : Parameterless_Handler;
                           Interrupt : Interrupt_Id);
    -- 将当前处理程序赋值为New_Handler
    -- 如果 New_Handler 是 null，则默认处理程序被恢复。
    -- 在下列情况引发Program_Error:
    --     (a) 如果同New_Handler 关联的保护对象未被标识有
    --         pragma Interrupt_Handler,
    --     (b) 如果中断是预留的，
    --     (c) 如果当前处理程序是用pragma Attach_Handler静态附加的。
  procedure Exchange_Handler(
              Old_Handler : out Parameterless_Handler;
              New_Handler : Parameterless_Handler;
              Interrupt : Interrupt_Id);
    -- 将中断的当前处理程序赋值为New_Handler ,
    -- 并返回Old_Handler的前一处理程序
    -- 如果 New_Handler 是 null，则默认处理程序被恢复。
    -- 在下列情况引发Program_Error:
    --     (a) 如果同New_Handler 关联的保护对象未被标识有
    --         pragma Interrupt_Handler,
```

```
    --      (b) 如果中断是预留的,
    --      (c) 如果当前处理程序是用pragma Attach_Handler静态附加的。
  procedure Detach_Handler(Interrupt : Interrupt_Id);
    -- 恢复指定中断的默认处理程序。
    -- 在下列情况引发Program_Error:
    --      (a) 如果中断是预留的,
    --      (b) 如果当前处理程序是用pragma Attach_Handler静态附加的。
  function Reference(Interrupt : Interrupt_Id)
           return System.Address;
    -- 返回一个Address (地址), 它可被用于通过一个入口上的地址子句
    -- 将一个任务入口附加于一个中断上。
    -- 如果该中断不能以此方式附加, 引发Program_Error 。
private
  ... --不由语言规定
end Ada.Interrupts;
```

应当注意，Reference函数提供了支持中断任务入口的机制。正如前面所提到的，这种中断处理的模型被认为是过时的因而不应该再使用。

程序15-2表明，实现也能使中断与名字关联。这在以下的例子中将会用到。

程序15-2 包Ada.Interrupts.Names

```
package Ada.Interrupts.Names is
  实现定义的 : constant Interrupt_Id := 实现定义的;
  ...
  实现定义的 : constant Interrupt_Id := 实现定义的;
private
  ... -- 不由语言规定
end Ada.Interrupts.Names;
```

15.4.3 一个简单的驱动程序的例子

一类常见的附属到嵌入式计算机系统中的设备是模拟数字转换器（ADC）。转换器对一些环境要素（比如温度或压力）进行采样，它将收到的测量值转换，测量值通常是毫伏，然后提供标度的整型值给硬件寄存器。考虑一个单独的转换器，它在硬件地址8#150000#有一个16位的结果寄存器，在8#150002#有一个控制寄存器。计算机是16位的机器，控制寄存器结构如下：

位	名字	含义
0	A/D启动	置1启动转换
6	中断允许/禁止	置1允许中断
7	做完	转换完成时置1
8-13	通道	转换器有64个模拟输入，所需的具体通道由通道值指出
15	出错	若设备功能出错，由转换器置1

这个ADC的驱动程序将被构造为一个包里面的保护类型，所以它产生的中断可以被作为一个保护过程调用处理，因而可以提供给多个ADC:

```
package Adc_Device_Driver is
  Max_Measure : constant := (2**16)-1;
  type Channel is range 0..63;
```

```
  subtype Measurement is Integer range 0..Max_Measure;
  procedure Read(Ch: Channel; M : out Measurement);
    -- 可能阻塞
  Conversion_Error : exception;
private
  for Channel' Size use 6;
  -- 指出只能用6个位
end Adc_Device_Driver;
```

对任何请求，驱动程序在引发异常前尝试三次。包体如下:

```
with Ada.Interrupts.Names; use Ada.Interrupts;
with System; use System;
with System.Storage_Elements; use System.Storage_Elements;
package body Adc_Device_Driver is
  Bits_In_Word : constant := 16;
  Word : constant := 2; -- 字中的字节数
  type Flag is (Down, Set);

  type Control_Register is
  record
    Ad_Start   : Flag;
    Ienable    : Flag;
    Done       : Flag;
    Ch         : Channel;
    Error      : Flag;
  end record;

  for Control_Register use
    -- 规定控制寄存器的格式
    record
      Ad_Start at 0*Word range 0..0;
      Ienable  at 0*Word range 6..6;
      Done     at 0*Word range 7..7;
      Ch       at 0*Word range 8..13;
      Error    at 0*Word range 15..15;
  end record;

  for Control_Register' Size use Bits_In_Word;
    -- 寄存器16位长
  for Control_Register' Alignment use Word;
    -- 在字边界上
  for Control_Register' Bit_order use Low_Order_First;

  type Data_Register is range 0 .. Max_Measure;
  for Data_Register' Size use Bits_In_Word;
    -- 寄存器16位长

  Contr_Reg_Addr : constant Address := To_Address(8#150002#);
  Data_Reg_Addr : constant Address  := To_Address(8#150000#);
  Adc_Priority : constant Interrupt_Priority := 63;
  Control_Reg : aliased Control_Register;
      -- 别名指出用指针去访问它
  for Control_Reg' Address use Contr_Reg_Addr;
```

599 ~ 601

```
    -- 规定控制寄存器的地址
Data_Reg : aliased Data_Register;
for Data_Reg' Address use Data_Reg_Addr;
    -- 规定数据寄存器的地址

protected type Interrupt_Interface(Int_Id : Interrupt_Id;
               Cr : access Control_Register;
               Dr : access Data_Register) is
  entry Read(Chan : Channel; M : out Measurement);
private
  entry Done (Chan : Channel; M : out Measurement);
  procedure Handler;
  pragma Attach_Handler(Handler, Int_Id);
  pragma Interrupt_Priority(Adc_Priority);
    -- 见13.11节关于优先级的讨论
  Interrupt_Occurred : Boolean := False;
  Next_Request : Boolean := True;
end Interrupt_Interface;

Adc_Interface : Interrupt_Interface(Names.Adc,
                Control_Reg' Access,
                Data_Reg' Access);
  -- 这假设了 'Adc' 被作为 Ada.Interrupts.Names 中的
  -- 一个Interrupt_Id登录
  -- 'Access 给出对象的地址


protected body Interrupt_Interface is

  entry Read(Chan : Channel; M : out Measurement)
       when Next_Request is
    Shadow_Register : Control_Register;
  begin
    Shadow_Register := (Ad_Start => Set, Ienable => Set,
          Done => Down, Ch    => Chan, Error => Down);
    Cr.all := Shadow_Register;
    Interrupt_Occurred := False;
    Next_Request := False;
    requeue Done;
  end Read;

  procedure Handler is
  begin
    Interrupt_Occurred := True;
  end Handler;

  entry Done(Chan : Channel; M : out Measurement)
                          when Interrupt_Occurred is
  begin
    Next_Request := True;
    if Cr.Done = Set and Cr. Error = Down then
         M := Measurement(Dr.all);
    else
```

```
        raise Conversion_Error;
      end if;
    end Done;
  end Interrupt_Interface;


  procedure Read(Ch : Channel; M : out Measurement) is
  begin
    for I in 1..3 loop
      begin
        Adc_Interface.Read(Ch,M);
        return;
      exception
        when Conversion_Error => null;
      end;
    end loop;
    raise Conversion_Error;
  end Read;
end Adc_Device_Driver;
```

602~603

客户任务只需调用Read过程，指出读的通道号和保存实际读入值的一个输出变量。在过程里面，一个内层循环通过调用与转换器相关联的保护对象里面的Read入口，尝试三次转换。在这个入口里面，控制寄存器Cr被设置成适当的值。一旦控制寄存器被赋值，客户任务在一个私有入口重排队等待中断。Next_Request标记被用于确保只有一个对Read的调用是未完成的。

一旦中断到达（作为一个无参保护过程调用），在Done入口的屏障被设为真，这导致Done入口被执行（作为中断处理程序的一部分）以确保Cr.Done被置位并且没有引起出错标记。如果是这种情况，根据缓冲区寄存器的值，使用一个类型转换，构造出一个输出参数M（注意这个值不能超出子类型Measurement的范围）。如果转换不成功，就引发Conversion_Error异常，这由Read过程捕获，该过程允许出错传播前总共可以尝试三次转换。

上面的例子说明，在写设备驱动程序时，经常需要将对象从一种类型转换到另一种类型。在这种情况下，Ada的强类型特征可能是一个障碍。然而，可以通过使用作为预定义库单元提供的一个类属函数来避免这个困难:

```
generic
  type Source (<>) is limited private;
  type Target (<>) is limited private;
function Ada.Unchecked_Conversion(S : Source) return Target;
pragma Convention(Intrinsic, Ada.Unchecked_Conversion);
pragma Pure(Ada.Unchecked_Conversion);
```

不加检查的转换的作用是将源的位模式拷贝给目标。程序员必须确保转换是明智的，并且所有可能的模式都是目标可接受的。

15.4.4 通过特别指令访问I/O设备

如果需要特别的指令，可以将汇编代码集成到Ada代码中去。机器代码的插入机制允许程序员编写包含明显的非Ada对象的Ada代码。这是通过在子程序体的上下文里只允许有机器代码指令的这种受控方式实现的。此外，如果一个子程序包含代码语句，那么它就只能包含代

码语句和“use”子句（注解和编用也照常允许）。

就像预期的那样，使用代码插入的细节和特征主要依赖于实现，实现专有的编用和属性可用于对定义代码指令的对象的使用强加特殊限制和调用约定。一个代码语句有如下结构：

604 代码语句 ::= 限定表达式

限定表达式必须是名为System.Machine_Code的预定义库包中声明的一种类型。就是这个包提供了记录声明（标准Ada中）去表示目标机器的指令。以下的例子说明了这个方法：

```
D : Data; -- 待输入

procedure In_Op; pragma Inline(In_Op);

procedure In_Op is
  use System.Machine_Code;
begin
  My_Machine_Format' (Code => In_Instruction, Reg => 1, Port => 1);
  My_Machine_Format' (CODE => SAVE, REG =>, S' Address);
end;
```

编用Inline指示：只要使用了子程序，编译器把插入式代码（而不是过程调用）包含进来。

虽然这个代码插入方法是在Ada中定义的，但该语言说得很清楚（ARM 13.8.4）：实现不需要提供一个Machine_Code包，除非系统编程附件是得到支持的。如果不是这样的，就禁止使用机器代码插入。

15.5 实时Java

虽然Java最初是为嵌入式系统设计的，并且对图形编程和文件处理的设施同等重视，然而这种语言没有涉及如何对设备驱动程序或处理中断进行编程。实时Java企图纠正这种状况，但目前只能提供有限的支持。

15.5.1 设备寄存器的寻址和操纵

实时Java支持通过原始存储器（raw memory）的观念访问设备寄存器的容量。允许实现支持一定范围内的存储类型，例如DMA存储器或共享存储器。这样的一个存储器可以分配给I/O寄存器：IO_Page。类RawMemoryAccess（在程序15-3中定义）可用于读或写这种存储器。这个方法不允许用户定义的对象被映射到原始存储器，因为这可能潜在地破坏类型机制。因此，必须作为基本数据类型（byte、int、short、long）访问单个的寄存器并且

605 使用低级的逐位操作。

程序15-3 类RawMemoryClass的摘录

```
public class RawMemoryAccess
{
  protected RawMemoryAccess(long base, long size);
  protected RawMemoryAccess(RawMemoryAccess memory, long base,
                                                    long size);

  public static RawMemoryAccess create(java.lang.Object type, long size)
         throws SecurityException,OffsetOutOfBoundsException,
                SizeOutOfBoundsException,
                UnsupportedPhysicalMemoryException;
```

```
    public static RawMemoryAccess create(java.lang.Object type,
                                         long base, long size)
           throws SecurityException, OffsetOutOfBoundsException,
                  SizeOutOfBoundsException,
                  UnsupportedPhysicalMemoryException;

    public byte getByte(long offset)
           throws SizeOutOfBoundsException, OffsetOutOfBoundsException;

    public void getBytes(long offset, byte[] bytes, int low,
                         int number) throws
                SizeOutOfBoundsException, OffsetOutOfBoundsException;

    // 对整型、长整型、短整型类似

    public void setByte(long offset, byte value) throws
                SizeOutOfBoundsException, OffsetOutOfBoundsException;

    public void setBytes(long offset, byte[] bytes, int low,
                         int number) throws
                SizeOutOfBoundsException, OffsetOutOfBoundsException;

    // 对整型、长整型、短整型类似
    ...
}
```

再次研究在15.4.3节中给出的简单ADC的控制和状态寄存器。

位	名字	含义
0	A/D启动	置1启动转换
6	中断允许/禁止	置1允许中断
7	做完	转换完成时置1
8-13	通道	转换器有64个模拟输入，所需的具体通道由通道值指出
15	出错	若设备功能出错，由转换器置1

606

首先必须为寄存器创建一个类。这个类的构造器创建RawMemoryAccess类的一个实例，指明存储器类型为IO_Page。然后方法SetControlWord可以用于访问寄存器自身。

```
public class ControlAndStatusRegister
{
  RawMemoryAccess rawMemory;

  public ControlAndStatusRegister (long base, long size)
  {
    rawMemory = RawMemoryAccess.create (IO_Page, base, size);
  }

  public void setControlWord (short value)
  {
    rawMemory.setShort (0, value);
  }
}
```

现在使用影子设备寄存器和逐位逻辑操作符，就能够构造正确的位模式。例如，在通道6开始一个转换：

```
byte shadow, channel;
```

```
final byte start = 01;
final byte enable = 040;
final long csrAddress = 015002;
final long csrSize = 2;
ControlAndStatusRegister csr = new
        ControlAndStatusRegister (csrAddress, csrSize);

channel = 6;
shadow = (channel << 8 | start | enable);
csr.setControlWord(shadow);
```

15.5.2 中断处理

实时Java将中断看作异步事件（见10.7节）。因而发生中断等同于方法fire被调用。在中断和事件间的关联是通过调用AsyncEvent类的bindTo方法实现的。参数是字符串类型的，并以一种依赖于实现的方式使用——一个可以传递中断向量地址的方法。当中断发生时，调用适当处理程序的fire方法。现在，可以将处理程序和一个可调度的对象联系起来并给它合适的优先级和启动参数。

607

```
AsyncEvent Interrupt = new AsyncEvent();
AsyncEventHandler InterruptHandler = new BoundAsyncEventHandler(
                  priParam, releaseParam, null, null, null);
Interrupt.addHandler(InterruptHandler);
Interrupt.bindTo("177760");
```

对于实时Java程序在一个符合POSIX的操作系统上执行的情况，类POSIXSignalHandler（见程序15-4）可以用于将异步事件处理程序和一个POSIX信号的出现关联起来。有趣的是，实时Java不支持POSIX实时信号。

程序15-4 类POSIXSignalHandler的摘录

```
public final class POSIXSignalHandler
{
  public static final int SIGABRT;
  public static final int SIGALRM;
  public static final int SIGBUS
  ...

  public static synchronized void addHandler(int signal,
                 AsyncEventHandler handler);

  public static synchronized void removeHandler(int signal,
                 AsyncEventHandler handler);

  public static synchronized void setHandler(int signal,
                 AsyncEventHandler handler);
}
```

15.6 occam2

本章的前面几节已经说明了通信和同步的基本共享变量模型是怎样被映射到具有存储-映射式I/O的机器上的。然而，这种模型并不处理具有优雅专用指令而是求助于特定的专用过

程的机器、嵌入的汇编代码或被编译器识别的特殊类型的变量。在这一节，occam2语言将作为基于消息的并发编程语言的例子来考察，它使用消息进行设备控制。

虽然occam2是为传输机设计的，但在下面的讨论中它被看作是一种与机器无关的语言。首先介绍模型，然后考虑它在存储－映射式机器和专用指令机器上的实现。由于使用了共享变量设备驱动，必须考虑设备封装、寄存器操纵和中断处理三个问题。 608

1. 模块性和封装设施

occam2提供的惟一封装设施是过程，因而，必须把它用于设备驱动程序的封装。

2. 设备寄存器的寻址和操纵

设备寄存器被映射到PORT，它在概念上与occam2的通道相似。例如，如果一个16位寄存器是在地址X上，那么PORT P被定义为:

```
PORT OF INT16 P:
PLACE P AT X:
```

注意，这个地址可被解释成一个内存地址，或者是一个设备地址，这依赖于实现。同设备寄存器的交互是通过读或写这个端口得到的:

```
P ! A  --  写A 的值到端口
P ? B  --  读端口的值到B
```

一个端口不能被定义为只读或只写的。

在occam2中，端口和通道的一个显著差别就是端口的交互不与任何同步相关联。读和写都不能导致执行进程被挂起，一个值总是被写到指定的地址，相似地，值总是被读。因此，端口就是有一个伙伴程序总是准备好进行通信的通道。

occam2提供使用移位操作和逐位逻辑表达式来操纵设备寄存器的设施。然而，它没有和Modula-1的位类型或是Ada的表示规格说明等价的东西。

3. 中断处理

在occam2中，中断作为与硬件进程的会合来处理。与中断相关联的，必须有一个依赖于实现的地址，在这一章描述的简单的输入/输出系统中，这个地址就是中断向量的地址，通道被映射到这个地址（ADDR):

```
CHAN OF ANY Interrupt:
PLACE Interrupt AT ADDR:
```

注意这是通道而不是端口。这是因为存在与中断相关联的同步，而没有与访问设备寄存器相关联。这个通道的数据协议也是依赖于实现的。

中断处理程序可以这样等待从指定的通道来的输入: 609

```
INT ANY:  -- 定义ANY是协议类型的
SEQ
  -- 使用端口允许中断
  Interrupt ? ANY
  -- 中断出现时的必需的动作
```

因此当外部中断发生时，运行时支持系统必须与指定的通道同步。为了得到响应，处理中断的进程通常被给予高优先级。因而通过中断事件它不仅被执行，而且在较短的时间内它一定实际执行（假设没有别的高优先级的进程在运行）。

为了接待因在一个特定时间段内没有处理而丢失的中断，必须将硬件看成发出了一个通信超时。所以，必须想像硬件这样发出：

```
ALT
  Interrupt ? ANY
    SKIP
  CLOCK ? AFTER Time PLUS Timeout
    SKIP
```

而且处理程序必须执行：

```
Interrupt ! ANY
```

这是因为只有输入请求可以有一个与之关联的超时。

4. *在存储－映射式机器和专用指令机器上的实现*

为了将occam2的设备驱动模型映射到存储－映射式机器上，只需要把在端口的输入输出请求映射成设备寄存器上的读操作和写操作。为了将此模型映射到专用指令机器上，需要进行如下的工作：

- 使用PLACE语句将一个occam2 PORT同一个I/O端口相关联；
- 将发送到occam2 PORT的数据放到适当的累加器以供输出机器指令使用；
- 执行完输入指令后，通过适当的累加器使从occam2 PORT接收的数据成为可用的。

15.6.1 一个设备驱动程序的例子

为了举例说明occam2提供的低级输入/输出设施的使用，将建立一个用于一台存储－映射
610 式机器控制模拟数字转换器（ADC）的进程。这个转换器与15.4.3节中描述的用Ada和Java实现的相同。为了读取一个特定的模拟输入，在8到13位给出了通道地址（不能与occam2的通道弄混），将0位置1启动转换器。当把一个值被加载到结果寄存器时，这个设备就中断处理器。在读结果寄存器之前将检查出错标记。在这个交互过程中最好禁止中断。

设备驱动程序将循环地接收请求并提供结果，它被编程为一个有两通道接口的PROC。当地址（对于8个模拟输入通道之一的）被传到input时，16位的结果将通过通道output返回。

```
CHAN OF INT16 request:
CHAN OF INT16 return:
PROC ADC (CHAN OF INT16 input, output)
  -- PROC的体，见下
PRI PAR
  ADC(request, return)
  PAR
    -- 程序的其余部分
```

PRI PAR是理想的，因为每次它被使用时ADC一定处理一个中断，因而应当在最高优先级运行。

在PROC体的内部，必须首先声明中断通道和两个PORT：

```
PORT OF INT16 Control.Register:
PLACE Control.Register AT #AA12#:
PORT OF INT16 Buffer.Register:
PLACE Buffer.Register AT #AA14#:
CHAN OF ANY Interrupt:
```

```
PLACE Interrupt AT #40#:
INT16 Control.R: -- 表示控制寄存器的变量
```

这里#AA12#和#AA14#是为两个寄存器定义的16进制地址，#40#是中断向量地址。

为了指示硬件进行一个操作，要求将控制寄存器的第0位和第6位置1，同时除了第8到13位（包含）外的其他的位必须置为0。这通过使用以下的常数实现：

```
VAL INT16 zero IS 0:
VAL INT16 Go IS 65:
```

从‘input’通道收到地址后，它的值必须赋到控制寄存器的8到10位。这是通过使用移位操作实现的。因此，为了开始转换必须执行的动作是：

```
INT16 Address:
SEQ
  input ? Address
  IF
    (Address < 0) OR (Address > 63)
    output ! MOSTNEG INT16 -- 出错情况
    TRUE
      SEQ
        Control.R := zero
        Control.R := Address << 8
        Control.R := Control.R BITOR Go
        Control.Register ! Control.R
```

611

一旦中断到达，就读取控制寄存器，然后检查出错标记和‘Done’。要做到这一点，控制寄存器必须被给以适当的常量值：

```
VAL INT16 Done IS 128:
VAL INT16 Error IS MOSTNEG INT16:
```

MOSTNEG的表示是1 000 000 000 000 000。

再进行下面的检查：

```
SEQ
  Control.Register ? Control.R
  IF
    ((Done BITAND Control.R)= 0) OR
    ((Error BITAND Control.R)<> zero)
      -- 出错
    TRUE
      -- 缓冲区寄存器有了适当的值
```

虽然设备驱动程序将在高优先级上运行，而客户进程通常却不，因此，如果驱动程序企图直接调用客户而客户还没有就绪，驱动程序就要延迟。对于异步产生数据的输入设备，这个延迟可能导致驱动程序错过一个中断。为了克服这个问题，输入数据必须被缓冲。下面给出了一个合适的循环缓冲区。注意，因为客户希望从缓冲区中读数据，又因为缓冲区中的ALT不可能有输出守备，需要另一个单独的缓冲区。为了确保设备驱动程序不因调度算法而延迟，两个缓冲区进程（以及驱动程序）都必须在高优先级执行。

```
PROC buffer(CHAN OF INT put, get)
  CHAN OF INT Request, Reply:
```

```
PAR
  VAL INT Buf.Size IS 32:
  INT top, base, contents:
  [Buf.Size]buffer:
  SEQ
    contents := 0
    top := 0
    base := 0
    INT ANY:
    WHILE TRUE
      ALT
        contents < Buf.Size & put ? buffer [top]
          SEQ
            contents := contents + 1
            top := (top + 1) REM Buf.Size
        contents > 0 & Request ? ANY
          SEQ
            Reply ! buffer[base]
            contents := contents - 1
            base := (base + 1) REM Buf.Size
  INT Temp:    -- 单个的缓冲区进程
  VAL INT ANY IS 0:    -- 哑值
  WHILE TRUE
    SEQ
      Request ! ANY
      Reply ? Temp
      get ! Temp
:
```

612

现在可以给出PROC的完整代码。设备驱动程序被重新构造，使得为了得到正确的读入值可以尝试三次。

```
PROC ADC(CHAN OF INT16 input, output)
  PORT OF INT16 Control.Register:
  PLACE Control.Register AT #AA12#:
  PORT OF INT16 Buffer.Register:
  PLACE Buffer.Register AT #AA14#:

  CHAN OF ANY Interrupt:
  PLACE Interrupt AT #40#:
  TIMER CLOCK:

  INT16 Control.R:   -- 表示控制缓冲区的变量
  INT16 Buffer.R:    -- 表示结果缓冲区的变量
  INT Time:

  VAL INT16 zero IS 0:
  VAL INT16 Go IS 65:
  VAL INT16 Done IS 128:
  VAL INT16 Error IS MOSTNEG INT16:
  VAL INT Timeout IS 600000: -- 或某个其他适宜的值
  INT ANY:
  INT16 Address:
```

```
BOOL Found, Error:
CHAN OF INT16 Buff.In:

PAR
  buffer(Buff.In, output)
  INT16 Try:
  WHILE TRUE
    SEQ
      input ? Address
      IF
        (Address < 0) OR (Address > 63)
          Buff.In ! MOSTNEG INT16
                    -- 出错情况

          TRUE
            SEQ
              Try := 0
              Error := FALSE
              Found := FALSE
              WHILE (Try < 3) AND ((NOT Found) AND (NOT Error))
                -- 为从ADC获取一个读数尝试三次。
                -- 这个读数可能是正确的，也可能被标记为出错。
                SEQ
                  Control.R := zero
                  Control.R := Address << 8
                  Control.R := Control.R BITOR Go
                  Control.Register ! Control.R
                  CLOCK ? Time
                  ALT
                    Interrupt ? ANY
                      SEQ
                        Control.Register ? Control.R
                        IF
                          ((Done BITAND Control.R) = 0) OR
                               ((Error BITAND Control.R) <> zero)
                            SEQ
                              Error := TRUE
                              Buff.In ! MOSTNEG INT16 -- 出错情况
                          TRUE
                            SEQ
                              Found := TRUE
                              Buffer.Register ? Buffer.R
                              Buff.In ! Buffer.R
                      CLOCK ? AFTER Time PLUS Timeout
                        -- 设备未响应
                        Try := Try + 1
              IF
                (NOT Found) AND (NOT Error)
                  Buff.In ! MOSTNEG INT16
                TRUE
                  SKIP
```

15.6.2 occam2设备驱动的困难

上面的例子说明了在occam2中写设备驱动程序和中断处理程序的一些困难。特别是在设备的硬件优先级和分配给驱动程序进程的优先级之间没有直接的联系。为了确保高优先级的设备优先，在PRI PAR构造中必须将所有的设备驱动程序在程序外层进行适当排序。

613~614 另一个主要的困难是缺乏表达设备寄存器的数据结构。这导致程序员不得不使用低级的位操作技术，而这可能是易出错的。

15.7 C和老式实时语言

第一代实时编程语言（RTL/2、Coral 66等等）并没有提供对并发编程或设备编程的真正支持。中断典型地被看成过程调用，通常访问设备寄存器的惟一可行的方法是允许将汇编语言代码嵌入到程序中。例如，RTL/2（Barnes，1976）有如下的代码语句:

```
code code_size, stack_size
    mov R3, @variable
    ...
    ...
@rtl
```

这种方法的缺点之一是：为了访问RTL/2变量，必须知道由编译器生成的代码的结构。

早期实时语言的另一个共同特征是它们都趋向于弱类型。因此变量可以被处理为定长的位串。这就使得能使用低级操作符操纵寄存器的单个的位和位段，比如逻辑移位和循环移位指令。可是，弱类型的缺点远远大于这个灵活性的好处。

虽然C语言比RTL/2和Coral 66出现得晚，却也延续了这个传统。设备寄存器由可以赋值到寄存器的存储器位置的指针变量寻址。通过低级逐位逻辑操作符或通过使用结构定义中的位段操纵它们。后者看起来类似于Ada中的记录表示子句，但实际上是既依赖于机器又依赖于编译器。为了说明C的位操作功能，下面介绍两个例子，第一个使用低级逐位逻辑操作符，第二个使用位段。

重新来看15.4.3节中给出的简单ADC的控制和状态寄存器。

为了使用逐位逻辑操作符，首先需要定义对应每一个位位置的一个屏蔽码组。

```
#define START  01 /* 自0开始的数是16进制的 */
#define ENABLE 040
#define ERROR  0100000
```

channel位段要么是在逐位的基础上定义的，要么将这个值移到它的正确的位置上。下面使用后一种方法，其中需要6号通道:

```
unsigned short int *register, shadow, channel;

register = 0xAA12;
channel = 6;
shadow =0;

shadow |= (channel << 8)|START | ENABLE
*register = shadow
```

615

使用位段，就成为:

```
struct {
```

```
  unsigned int start     : 1; 一位长位段
  unsigned int           : 5; 五位长无名位段
  unsigned int interrupt : 1; 一位长位段
  unsigned int Done      : 1; 一位长位段
  unsigned int Channel   : 6; 六位长位段
  unsigned int error     : 1; 一位长位段
} control_register;

control_register *register, shadow;
register = 0xAA12;
shadow.start = 1;
shadow.Interrupt = 1;
shadow.channel = 6;
shadow.error = 0;

*register = shadow;
```

这个例子中有两点需要注意:

- C不能保证位段的排序，因此编译器可以决定使用不同于程序员意图的次序，将位段打包到字里面。
- C语言不试图决定机器的数位是从左到右或是从右到左。

有了这两点，位段不应用于访问设备寄存器就很清楚了，除非程序员知道特定的编译器对使用的机器是怎样实现位段的。即使在这种情况下，代码也是不能移植的。

为了可移植性，C程序员被迫使用低级的逐位逻辑操作符。下面的例子显示了这些代码是怎样很快变成不可读的（虽然可以争辩说它们产生更高效的代码）。下面的过程将reg所指向的寄存器从位置p开始的n位设置为x。

```
unsigned int setbits(unsigned int *reg, unsigned int n,
        unsigned int p, unsigned int x)
{
  unsigned int data, mask;

  data = (x & (~(~0 << n)))<<(p); /* 要被屏蔽的数据*/
  mask = ~ (~0<<n); /* 屏蔽码 */
  *reg &= ~ (mask<<(p)); /* 清除现有的位*/
  *reg |= data; /* 数据中的OR */
}
```

在这个例子中C代码相当简洁: ~的意思是逐位求补，<< 是向左移位（用0填充），&是逐位的“与”，| 是逐位的“或”。 616

按照15.1.4节中概述的简单的I/O体系结构，通过将无参过程调用的地址放置在适当的中断向量位置而指定中断处理程序。一旦执行了这个过程，就一定可以直接将同程序其余部分的任何通信和同步编程。

虽然POSIX提供了替代机制，它在理论上可用于提供一个中断处理的替代模型（例如，将中断和条件变量关联起来)，但现在仍然没有将用户定义的处理程序附加到中断上的标准机制。

15.8 设备驱动程序的调度

因为许多实时系统都有I/O部件，在调度分析中加入低级编程专有的特征就是很重要的了。

已经注意到，DMA和通道程序控制技术是太不可预测（在它们的时态行为方面）了，以至于无法分析它们。因此这一节关注的是中断驱动程序控制方法和状态驱动方法。

只要中断启动偶发进程执行，就必须给中断处理程序自身分配一定的开销。处理程序的优先级很可能比偶发进程的优先级高，这意味着具有比偶发进程高优先级的进程（但是优先级低于中断处理程序）将受到干扰。其实，这是优先级反转的例子，因为处理程序的惟一工作就是启动偶发进程——它的理想优先级应该与偶发进程相同。令人遗憾的是，大部分的硬件平台要求中断优先级比普通的软件优先级高。为了给中断处理程序建模，在可调度性测试中包含了一个额外的“进程”。它的“周期”等于偶发进程的周期，其优先级与中断优先级级别相同，执行时间等同于它的最坏情况的行为。

使用状态驱动设备，控制代码可以用通常的方法分析。然而这样的设备带来了特别的困难。使用一个输入设备的协议常常是这样的：请求读，等待硬件进行读，再访问寄存器将读的数据取到程序中。问题在于在读的时候怎样管理延迟。依赖于延迟持续时间的长短，可能有三种方法：

- 在“done”标记上忙等待
- 在未来的某个时候重新调度进程
- 对于周期进程，将动作在周期之间分解

对于很短的延迟，忙等待是可接受的。从调度的观点看，“延迟”都是计算时间，因此只要“延迟”是有界的，分析的方法就没有改变。为了防止设备中的失效（即它从不设置done

617 位），可以使用超时算法（见12.4节）。

如果延迟相当长，采用挂起进程、执行其他工作、然后在值应该成为可用的的某个未来时刻返回到I/O进程的方法更有效。所以，如果读的时间是30ms，代码就是：

```
begin
  -- 准备读
  delay Milliseconds(30);
  -- 取走读数并使用
end;
```

从调度的观点看，这个结构有三个重要的暗示。首先，响应时间不容易计算。进程的每一半都必须单独地分析。总响应时间是将子响应时间和30ms的延迟相加得到的。虽然在进程中有延迟，但是在考虑这个进程对低优先级进程的影响时，这个延迟可以忽略。第二，包含在延迟中的额外计算时间和重新调度的时间必须加到进程的最坏情况执行时间（参看16.3节关于怎样计算系统开销的讨论）。第三，对阻塞有影响。记得简单的计算进程响应时间的方程是（见13.7节）：

$$R_i = C_i + B_i + I_i$$

B_i是阻塞时间（就是说，进程可以被低优先级进程的动作延迟的最长时间）。在13.10节中考虑了对于资源共享的各种协议。所有有效的协议都有一个性质：B_i仅由一个阻塞组成。然而，当进程延迟时（并允许较低优先级的进程执行），当它从延迟队列启动时有可能再次被阻塞。因此响应时间的方程变为：

$$R_i = C_i + (N+1)B_i + I_i$$

这里N是内部延迟的数目。

对于周期性进程，有另外一种方法管理这种显式延迟。这个方法被称为**周期移位**，它在一个周期开始读，但是在下一个周期取走读的值。例如：

```
-- 准备第一次读
loop
  delay until Next_Release;
  -- 检查done标记设置
  -- 取走读入值并使用
  -- 准备下一次读
  Next_Release := Next_Release + Period;
end loop;
```

这是一个直接的方法，对调度没有影响。当然，读数是过时了一个周期，对应用来说这可能是不可接受的。为了确保在一个执行结束和下一个执行开始之间有足够的间隙，可以调整进程的时限。所以，如果S是设备的还原时间，要求的约束就是$D < T - S$。注意读的最大停滞限制为$T+D$（或$T+R$，在计算了最坏情况响应时间的时候）。 618

15.9 存储管理

嵌入式实时系统通常只有有限数量的存储器可用，这或者是因为成本，或者是因为与周边系统相关的其他约束（例如，大小、功率或重量约束）。因此，需要控制这些内存如何分配以便它们能被更有效地使用。此外，当系统中有不止一种类型的存储器时（有不同的访问特征），必须指示编译器将某些数据类型放置在某些位置。这样做程序才能提高性能和可预测性，又能与外部世界交互。

这一章已经考虑了数据项怎样被分配到特定的存储位置，在存储单元中的某些位段怎样用于表示指定的数据类型。这一节考虑存储管理比较一般的问题。注意力集中在编译器在运行时用于管理数据的两个基本部件的管理：堆和栈。

15.9.1 堆管理

大部分编程语言的运行时实现提供了大量存储器（称为**堆**），以使程序员可以在运行时提出分配大块存储器的请求（例如，为了保存一个在编译时不知道边界的数组）。分配器（通常是new操作符）就是用于这个目的。它返回一个指向对于程序数据结构有足够大小的堆存储空间的指针。运行时支持系统负责管理这个堆。关键的问题是决定需要多大的空间和什么时候可以释放分配的空间。一般来说，第一个问题需要应用的知识。第二个问题可以采用几种方法处理，包括：

- 要求程序员显式地返回内存——这是易出错的，但是易于执行；这是C编程语言采用的方法，使用malloc和free函数——函数sizeof可以给出数据类型以字节为单位的大小（从编译器得到）；
- 要求运行时支持系统管理存储器，并决定什么时候在逻辑上它不再被访问——Ada的作用域规则允许它的实现采用这个方法；当访问类型离开作用域的时候，与那个访问类型相关的所有内存都被释放；
- 要求运行时支持系统管理内存，并释放不再使用的大块（**垃圾回收**）——这可能是最通用的方法，即使其关联的访问类型仍然在作用域之内也允许释放内存。 619

从实时的观点看，上述方法对程序时间性属性分析能力的影响依次增大。特别是，垃圾

回收可能在堆是空的时或通过异步行为执行（增量式垃圾回收）。在任何一种情况下，运行垃圾回收将对一个时间至上的任务的反应时间有显著的影响。虽然在实时垃圾回收方面做了很多研究工作，并且持续地取得了进展（例如，参看Lim等（1999），Siebert（1999）和Kim等（1999）），但在时间至上的系统中仍然不愿意依赖这些技术。

以下的小节将更深入地研究Ada和实时Java提供的功能。

1. Ada中的堆管理

在Ada中，堆是由一个或多个**存储池**表示的。存储池与一个特定的Ada划分（对于非分布式的Ada系统就是整个程序）相关联。一种访问类型的每个对象都有一个与之关联的存储池。分配器(new)从目标池中获得存储。`Ada.Unchecked_Deallocation`将数据返回到池中。实现可以支持一个单独的、当划分终止时将被恢复的全局池，或是可以支持定义在不同的可访问级别的多个池，当退出相关联的作用域时把它们恢复。默认的方式是实现为每种访问类型选择一个标准的存储池。注意，所有直接访问的对象（不通过指针）都放置在栈中，而不是堆中。

为了在存储管理上提供更多的用户控制，Ada定义了称为`System.Storage_Pools`的包，程序15-5中给出了它。

程序15-5 Ada的包System.Storage_Pools

```
with Ada.Finalization;
with System.Storage_Elements;

package System. Storage_Pools is

  pragma Preelaborate (System.Storage_Pools);

  type Root_Storage_Pool is abstract new
       Ada.Finalization.Limited_Controlled with private;

  procedure Allocate(Pool : in out Root_Storage_Pool;
            Storage_Address : out Address;
            Size_In_Storage_Elements : in System.
                 Storage_Elements.Storage_Count;
            Alignment : in System.Storage_Elements.Storage_Count)
            is abstract;

  procedure Deallocate(Pool : in out Root_Storage_Pool;
            Storage_Address : in Address;
            Size_In_Storage_Elements : in System.
                 Storage_Elements.Storage_Count;
            Alignment : in System.Storage_Elements.Storage_Count)
            is abstract;

  function Storage_Size(Pool: Root_Storage_Pool) return
            System.Storage_Elements.Storage_Count is abstract;
private
  ...
end System.Storage_Pools;
```

程序员通过扩充`Root_Storage_Pool`类型可以实现他们自己的存储池，并对子程序体提供具体的实现。为了将访问类型和存储池相关联，首先声明这个池，然后使用`Storage_Pool`属性。

```
My_Pool : Some_Storage_Pool_Type;

type A is access Some_Object;
for A' Storage_Pool use My_Pool;
```

现在，所有使用A的‘new’调用都将自动调用Allocate；对Ada.Unchecked_ Deallocation的调用将调用Deallocate；它们都引用My_Pool。此外，当访问类型离开作用域时，实现将调用Deallocate。

最后，应当注意，Ada不要求实现支持垃圾回收。然而，它确实支持一个编用Controlled，这个编用指示对一个特定类型不执行垃圾回收。

2. 实时Java中的堆管理

与Ada相反，Java中所有的对象都在堆中分配，Java要求垃圾回收有一个有效的实现。实时Java认识到，必须使存储管理能不受各种古怪风格的垃圾回收的影响。为此，它引入**存储区域**的概念，一些存储区域存在于传统的Java堆之外，并且从不受垃圾回收的影响。程序15-6为所有存储区域定义了抽象类。当进入特定的存储区域时，所有对象的分配都在那个区域内进行。 620~621

程序15-6 Java的抽象类MemoryArea

```
public abstract class MemoryArea
{
  protected MemoryArea(long sizeInBytes);

  public void enter(java.lang.Runnable logic);
  // 在方法logic.run执行期间，将此存储区域同当前线程关联起来

  public static MemoryArea getMemoryArea(java.lang.Object object);
  // 获取同此对象关联的存储区域

  public long memoryConsumed();
  // 在此存储区域消耗的字节数
  public long memoryRemaining();
  // 所剩字节数

  public synchronized java.lang.Object newArray(java.lang.class type,
        int number) throws IllegalAccessException,
        InstantiationException, OutOfMemoryError;
  // 分配一个数组

  public synchronized java.lang.Object newInstance(java.lang.class type)
        throws IllegalAccessException, InstantiationException,
        OutOfMemoryError;
  //  分配一个对象

  public long size(); // 存储区域的大小
}
```

使用这个抽象类，实时Java定义了各种类型的存储器，包括如下的:

- *永久存储器* 永久存储器被应用中的所有线程共享。在永久存储器中创建的对象不会受制于垃圾回收并且只有当程序终止时才会被释放。

```
public final class ImmortalMemory extends MemoryArea
```

```
{
  public static ImmortalMemory instance();
}
```

还有一个称为ImmortalPhysicalMemory的类与永久存储器具有相同的特征，但是它使对象能够在一个物理地址范围内分配。

• *作用域存储器*

作用域存储器是可以分配有明确定义生命期的对象的存储区域。作用域存储器可以显式
622 进入（通过使用enter方法）或是在线程创建的时候可以隐式地附加到RealtimeThread。同每一个作用域存储器相关联的是一个引用计数，每次调用enter和在每个相关的线程创建时它都会增加。当enter方法返回和每个相关的线程退出时它会减少。当引用计数到0时，所有驻留于作用域存储器的对象都执行它们的终了化方法（finalization），然后存储器被恢复。通过嵌套调用enter方法，作用域存储器可以嵌套。

ScopedMemory类（在程序15-7中定义）是一个抽象类，它包含下面几个子类:

• VTMemory——分配可能占用的时间不定;
• LTMemory——以线性时间分配（与对象的大小有关）;
• ScopedPhysicalMemory——允许对象在物理存储位置分配。

程序15-7 实时Java的类ScopedMemory

```
public abstract class ScopedMemory extends MemoryArea
{
  public ScopedMemory(long size);

  public void enter(java.lang.Runnable logic);

  public int getMaximumSize();

  public MemoryArea getOuterScope();

  public java.lang.Object getportal ();

  public void setPortal (java.lang.Object object);
}
```

这些子类的定义在附录A中给出。

为了避免可能发生的悬挂指针，对各种存储区域的使用设置了一套访问限制。

• 堆对象——只可以引用别的堆对象和在永久存储器中的对象（就是说它不能访问作用域存储器）;
• 永久对象——只可以引用堆对象和永久存储器对象;
• 作用域对象——只可以引用堆对象、永久对象和在同一个作用域或外层作用域的对象。

在实时线程和异步事件处理程序创建时可以给出存储器参数。或是作为允许控制策略的一部分，和/或是为了确保适当的垃圾回收的目的，调度程序可以使用存储器参数。程序15-8定义了这个类。

程序15-8 实时Java的类MemoryParameters

```
public class MemoryParameters
{
```

```
  public static final long NO_MAX;

  public MemoryParameters(long maxMemoryArea, long maxImmortal)
         throws IllegalArgumentException;

  public MemoryParameters(long maxMemoryArea, long maxImmortal,
                          long allocationRate)
         throws IllegalArgumentException;

  public long getAllocationRate();
  public long getMaxImmortal();
  public long getMaxMemoryArea();

  public void setAllocationRate(long rate);
  public boolean setMaxImmortal(long maximum);
  public boolean setMaxMemoryArea(long maximum);
}
```

例如，考虑一个实时线程，它希望它的内存默认地是从永久存储器分配的。然而，在计算的某些部分，它希望创建一些具有明确定义的生命期的临时对象。首先，定义线程的代码。方法computation代表需要临时存储的代码部分。它创建了某个线性时间作用域存储器指明它需要的最小和最大大小。然后它定义了一个局部的Runnable以包含实际的计算。代码myMem. enter限制了存储器的作用域。

```
import javax.realtime.*;
public class ThreadCode implements Runnable
{
  private void computation()
  {
    final int min = 1*1024;
    final int max = 1*1024;
    final LTMemory myMem = new LTMemory(min, max);
    myMem.enter(new Runnable()
    {
      public void run()
      {
      // 此处的代码需要访问临时存储器
      }
    });
  }

  public void run()
  {
  ...
  computation();
  ...
  }
}
```

623
~
624

现在可以创建线程了。注意，在此例中，除存储区域和Runnable之外，没有别的参数。

```
ThreadCode code = new ThreadCode();

RealtimeThread myThread = new RealtimeThread(
```

```
        null, null, null, ImmortalMemory.instance(),
        null, code);
```

15.9.2 栈管理

和管理堆一样，嵌入式程序员也必须关心栈的大小。虽然指定任务/线程的栈大小只需微小的支持（例如，在Ada中是通过将`Storage_Size`属性应用到任务上；在POSIX中是通过pthread属性），但计算栈的大小却更困难。在任务进入阻塞和执行过程的时候，它们的栈就增长。为了精确地估计这个增长的最大范围，需要知道每个任务的执行行为。这种知识类似于进行最坏情况执行时间（WCET）分析（见13.7节）所需的知识。因此，WCET和最坏情况的栈使用限制都可以从执行任务代码控制流分析的单独工具中得到。

小结

嵌入式系统的一个主要特征是同专用输入输出设备进行交互的需求。为了用高级语言编制设备驱动程序，需要：

- 将数据和控制信息传递到设备和从设备中传递出来的能力；
- 处理中断的能力。

通常控制和数据信息是通过设备寄存器传递到设备上。这些寄存器要么通过存储－映射式I/O体系结构中专门的地址，要么通过专用机器指令访问。中断处理需要上下文切换、设备
625 和中断标识、中断控制和设备优先级。

设备编程一直是汇编语言程序员的根据地，但是像C、Modula-1、occam2和Ada等语言逐步地尝试对这些低级功能提供高级机制。这使得设备驱动程序和中断处理例程更容易阅读、编写和维护。对高级语言的主要要求是提供设备处理的抽象模型。封装功能也是需要的，以使程序中不可移植的代码能够与可移植的部分分开。

设备处理的模型是建立在语言的并发性模型上的。设备可以认为是执行固定进程的处理器。因此计算机系统可以被建模为若干个需要通信和同步的并行进程。中断的建模有几种方法。它们必须有：

1）寻址和操纵设备寄存器的设施。

2）中断的合适表示。

在设备驱动的纯粹共享变量模型中，驱动程序和设备通信使用共享设备寄存器，中断提供条件同步。Modula-1给程序员提供了这样的模型。驱动程序进程被封装到有管程功能的设备模块里面。设备寄存器被作为标量对象或位数组访问，中断被看作是条件变量上的信号。

在Ada中，使用一套全面的把类型映射到底层硬件的设施，设备寄存器可以被定义为标量和用户定义的记录类型。中断被看作是对保护对象的由硬件生成的过程调用。

只有occam2给程序员展现了设备驱动的基于消息的视图。设备寄存器作为专门的通道被访问，称之为端口，中断被作为无内容的消息对待传送给通道。

实时Java支持通过`RawMemoryClass`访问存储－映射式I/O寄存器，然而，它缺乏操纵设备寄存器的表达能力。中断被看作是异步事件。

低级编程也涉及管理处理器的存储资源的更一般的问题。这一章考虑了栈和堆的管理。

Ada不要求垃圾回收——存储可以显式地回收，并且该语言的作用域规则使得在访问类型出了作用域时能够自动回收。Ada也允许定义用户定义的存储池，这使得程序员可以定义他们自己的存储管理策略。

实时Java认识到Java的存储分配策略对于实时系统是不能胜任的。从而，它允许在堆外分配存储器，并支持作用域存储器的概念，这样就能自动恢复存储，而无须垃圾回收。 626

相关阅读材料

Barr, M. (1999) *Programming Embedded Systems in C and C++*. Sebastopol, CA: O'Reilly.

Bollella, G., Brosgol, B., Dibble, P., Furr, S., Gosling, J., Hardin, D. and Turnbull, M. (2000) *The Real-Time Specification for Java*. Reading, MA: Addison-Wesley.

Project UDI (2000) *Uniform Driver Interface (UDI)* `http://www.project-udi.org/`, accessed June 2000.

练习

15.1 考虑一台嵌入到病人监护系统的计算机（采用本章给出的简单的I/O系统）。这个系统的安排是：每次病人的心脏跳动时，都通过向量位置100（八进制）产生一个最高硬件优先级中断。另外，通过设备控制寄存器管理轻度的电击，寄存器的地址是177760（八进制）。此寄存器的设置是：每次把一个整型值x赋给这个寄存器的时候，病人在很短的时间内会收到x伏的电压。

如果在5秒的周期内没有心跳的记录，病人的生命就处于危险中。当病人心跳停止时必须采取两个行动：第一，应当通知“监护者”任务，使之响起医院警报，第二就是必须使用一次5伏的电击。如果病人没有反应，电压必须每过5秒就增加1伏。

写一个Ada程序监视病人的心跳并开始上面描述的动作。你可以假设监护者任务的规格说明如下：

```
task Supervisor is
  entry Sound_Alarm;
end Supervisor;
```

15.2 西班牙政府正在考虑提出使用汽车高速公路的收费问题。一个可能的机制是沿着所有汽车高速公路在等间隔的地方修建监测站，当车辆通过监测站的时候，记录它的详细信息，并记录公路费用。在每个月末，向车主发送他或她的汽车高速公路的使用账单。

每个车辆需要一个接口设备，当监测站需要它的详细信息时，这个设备中断一个单板计算机。计算机字长16位，有存储－映射式I/O，所有I/O寄存器的长度是16位。中断是向量化的，与监测站关联的中断向量地址是8#60#。收到中断后，在位置8#177760#的只读输入寄存器包含了使用当前汽车高速公路路程的基本费用（比塞塔数——比塞塔(peseta)，西班牙币名）。中断的硬件优先级是4。 627

计算机软件必须在5秒内响应中断，然后通过接口设备将汽车的详细信息传送到监测站。它通过写地址在8#177762#的5个控制和状态寄存器堆来做这些事。这些寄存器的结构如表15-1。

表15-1 道路收费的寄存器结构

寄存器	位	含义
1	0-7	车辆注册特性1
1	8-15	车辆注册特性2
2	0- 7	车辆注册特性3
2	8-15	车辆注册特性4
3	0-7	车辆注册特性5
3	8-15	车辆注册特性6
4	0-7	车辆注册特性7
4	8-15	车辆注册特性8
5	0	置1以传送数据
5	1-4	旅行详细情况 1 = 商务 2 = 休闲 3 = 外国旅游者 4 = 警察 5 = 军事 6 = 紧急服务
5	5-15	安全码（0-2047）

CSR寄存器库（bank）是只写的。

写一个与监测站对接的Ada任务。这个任务应当响应来自接口设备的中断并负责发送正确的车辆详细信息。它还应当读包括公路使用费用的数据寄存器，然后将当前旅程总费用传送到一个任务，这个任务将它输出到在汽车仪表上的可见显示部件上。你可以假设有下面的代码可用:

628

```
package Journey_Details is
  Registration_Number : constant String(1..8) :=
                      "......."; 

  type Travel_Details is
       (Business, Pleasure, Overseas_Tourist,
        Police, Military, Emergency_Service);
  for Travel_Details use
       (Business =>1, Pleasure =>2, Overseas_Tourist =>3,
        Police =>4, Military =>5, Emergency_Service =>6);

  subtype Security_Code is Integer range 0..2047;

  function Current_Journey return Travel_Details;

  function Code return Security_Code;
end Journey_Details;

package Display_Interface is

  task Display_Driver is
      entry Put_cost(C : Integer);
      -- 将费用显示在操纵板显示器上
  end Display_Driver;

end Display_Interface;
```

假设编译器表示`Character`类型为8位值，`Integer`类型为16位值。

15.3 一个基于时槽环的局域网是这样的网：它包含有一些时槽，数据可以被放置在其中进行传输，这些时槽不断地绕着环转。考虑一个特定的环，它只有一个时槽，称之为分组（packet），有40位宽。分组的结构如图15-3所示。

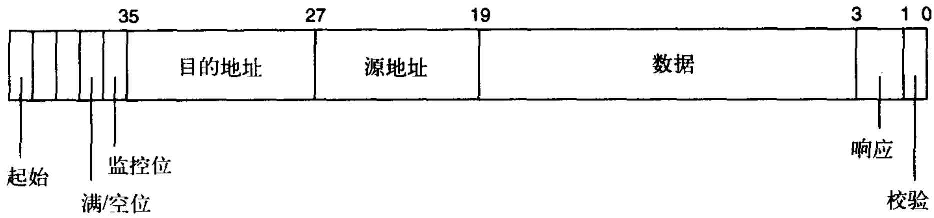


图15-3 时槽环

第39位指明分组的开始，通常被设置为1。第36位指明分组是满的或是空的（就是说，时槽是否正在使用），第27到34位用于指明数据的目的地址，第19到26位用于保存数据的源地址，第3到18位用于存放要发送的数据，第0位是校验位，在这个问题中可以被忽略。响应位（第1到2位）和监控位（第35位）在下面说明。

一个寻找空分组的传送过程将置分组的满/空位为1，将响应位清除为0，设置目的和源地址，放置要传送的数据。分组绕着环转，环上的每个站点都检查这个分组，以查看它是否是分组的目的地址。如果是，它就将数据从分组中拷贝出来，然后设置响应位（设为二进制的11）指明数据已经收到。最初的发送方检查响应位，然后通过设置满/空位为0移开数据，从而指明分组是空的。然后空分组被发送。如果发送方希望发送另外一条消息它必须等到它重新收到一个空的分组，这防止一个发送者独占这个环。 629

虽然这个环的出错率很低，但是在分组中的数据可能被破坏。特别是，发送方地址可能被破坏。为了避免相同的一个满分组不停地绕着环转的可能性，引入一个监控站。这个监控站读每个分组并置监控位为1。当发送站发送了一个分组时就将监控位置为0。因此如果监控站读到一个分组是空的，并且监控位被置为1，那么这个分组总是绕着环转，因此地址一定被破坏了。如果这个错误状态发生了，满/空位应该被设为0以指明分组可以被重用。站和环的接口由4个16位的存储映射寄存器控制。第一个寄存器是控制和状态寄存器，它常驻于八进制位置177760。这个寄存器的结构如表15-2所示。

表15-2 时槽环的控制寄存器结构

位	含 义
0	校验位
1，2	响应位
3	监控位
4	满/空位
6	允许中断
10	传送分组

第二和第三个寄存器是源和目的地址寄存器，分别常驻于八进制地址177762和177764。它们的结构如表15-3所示。

表15-3 时槽环的地址寄存器结构

位	含 义
0-7	地址
8-15	未用

最后一个寄存器是数据寄存器，它常驻于八进制地址177766。所有四个寄存器都可以读和写。当分组到达时会有一个中断信号。中断在八进制地址60向量化，中断优先级是6。在中断时，控制和状态寄存器将指明满/空位、监控位、响应位和校验位的值。通过写控制和状态寄存器可以改变这些位的值。如果第10位被中断处理进程置1，分组就被发送。数据的内容、源和目的地址可以被类似地读和更改。

写一个Ada包通过一个分槽环发送和接收整型值。包的规格说明给出如下：

```
package Slotted_Ring_Driver is

    type Station_Id is private;
    Station1 : constant Station_Id;
    Station2 : constant Station_Id;
    Station3 : constant Station_Id;
    Station4 : constant Station_Id;

    -- 等等

    procedure Transmit (To_Station : Station_Id;
                        Data : Integer);

    procedure Receive (From_Station : out Station_Id;
                       Data : out Integer);
private

    type Station_Id is new Short_Integer;  -- 16位
    Station1 : constant Station_Id := 1;
    Station2 : constant Station_Id := 2;
    -- 等等
end Slotted_Ring_Driver;
```

你可以假设如果当分组返回到发送方时分组的响应位被设为0，那么数据没有被收到。然而，不需要重试——数据被丢弃。

你也可以忽视奇偶校验，并假设“短整型”占据了16位存储器。

15.4 用occam2、Modula-1和实时Java重写练习15-2的答案。

15.5 考虑一个简单的机器人手臂，它与一个有简单I/O系统的计算机相连接，它只可以沿着水平轴移动。这个设备由两个寄存器控制：一个位于八进制位置177234的数据寄存器和一个位于八进制位置177326的控制寄存器。当设备处于允许操作状态时（通过设置控制寄存器的第6位），把一个坐标放入数据寄存器，机器人的手臂就移向那个坐标，当手臂到达新位置上时，产生一个中断（通过八进制位置56，硬件优先级为4）。

630~631 定义一个Modula-1设备模块以便一个Modula-1进程通过调用由设备模块定义的例程MOVETOPOSITION将手臂移动到特定的位置，使用一个参数指定要求的位置。当手臂位于新的位置时这个过程必须返回。你可以假设每次只有一个进程调用MOVETOPOSITION。

15.6 设计一个Modula-1设备模块，使得一个调用的进程能够被延迟几个时钟滴答。调用进程应该通过一个称为DELAY的过程与设备模块交互，这个过程有一个整型参数指明延迟持

续的滴答数。当延迟时间到期时过程返回。你可以假设时钟设备的优先级是6，中断向量位置在八进制位置100处，控制和状态寄存器在八进制地址177546，长度是16位。这个寄存器的第6位置1后允许中断。

15.7 用occam2重写在15.3.3节中给出的键盘设备驱动程序。

15.8 比较和对照Ada和Modula-1在设备驱动程序编程上的限制。

15.9 英国政府关注汽车高速公路上汽车的速度。今后将沿着汽车高速公路在等间隔的地方修建信标，它们将不断地发送当前速度限制。新式的汽车将带有计算机，它可以监测当前速度限制并在超出速度限制时通知驾驶员。

现在正设计的汽车（Yorkmobile）已经有了必需的硬件接口。这些接口如下:

- 每辆汽车有一个具有存储－映射式I/O的“速度控制”的16位计算机，所有的I/O寄存器都是16位长的。
- 一个位于八进制地址177760的寄存器与监测公路边信标的设备接口。寄存器总是保存着从路边信标收到的最新速度限制的值。
- 一对寄存器与一个根据一套限制来检测汽车速度的智能速度计设备接口。如果速度限制被超过了，设备通过八进制地址60产生一个中断。这个中断的优先级为5。这个中断每5秒重复一次直到汽车不再超速。
- 这对寄存器由一个“控制和状态”寄存器（CSR）和一个数据缓冲寄存器（DBR）组成。CSR寄存器的结构如表15-4所示。

表15-4 控制寄存器结构速度计算机

位	含　义
0	设备允许操作
1	置为1时，将DBR中发现的值用作汽车的当前速度限制
5-2	未用
6	允许中断
11-7	未用
15-12	出错位（0=无出错，>0 非法限制值）

CSR寄存器既可以读又可以写，位于八进制地址177762。DBR寄存器只保存设置的表示汽车速度限制的整型值。如果这个值超出了0～70的范围，就规定了一个非法的限制，然后设备使用当前的限制继续工作。DBR寄存器的地址是八进制177764。

- 一个闪光灯（在汽车仪表板上）可以通过设置位于八进制地址177750的寄存器值为1打开，这个灯只闪烁5秒。当设置为0时灯关闭。

设计一个实现下列速度控制算法的occam2设备驱动程序。

每60秒通过速度控制计算机从路边的信标中读当前的速度限制。这个值不加检测地被传送到速度计设备，如果汽车超出了速度限制或是速度限制是不合法的，速度计设备就产生中断。如果汽车速度超出了限制，仪表板上的灯将闪烁起来，直到汽车的速度回到当前的限制以内。

15.10 使用Modula-1、Ada和实时Java重做练习15.9。

15.11 比较和对照Modula-1的设备驱动的共享存储模型和occam2的消息发送模型。 632～633

第16章 执行环境

实时系统必须对周围环境发生的事件做出及时的响应，这是它们的本性。这导致形成了以下观点：实时系统必须尽可能地快，由语言或操作系统中那些支持高级别抽象（如管程、异常、原子动作等）的特性所引入的任何开销都是不能容忍的。术语‘效率’通常用于描述编译器生成的代码的质量或者操作系统或运行时支持系统所支持的机制提供的抽象级别。但是这个术语并没有良好的定义。而且，从许多方面来看，用效率来评价一个应用和它的实现都是一个拙劣的度量标准。在实时系统中，真正重要的是满足时限或在一个特定的执行环境中得到能满足需要的响应时间。本章将考虑同实现这个目标相关联的一些问题。首先将考虑执行环境对实时系统的设计和实现的影响。接着将讨论剪裁软件执行环境以满足应用需要的方法。然后，从促进应用的全面可调度性分析的角度评述内核的调度模型。最后将说明执行环境中的硬件是如何支持本书中提出的一些抽象的。

16.1 执行环境的作用

在第13章，将可调度性分析作为预测软件的实时属性的根本点。但是，除非知道建议的执行环境的细节，否则进行这种分析是困难的。术语“执行环境”是指同应用代码一起使用构成完整系统的那些部件：处理器、网络、操作系统等。建议的执行环境的特性将指出一个特定的设计是否满足它的实时需求。显然，执行环境的使用越是“高效”，就越有可能满足需求。但不总是这种情况，一个拙劣的结构设计，不管它如何高效地实现，它都不能满足它的需求。例如，一个有明显优先级反转的设计，不管如何高效地实现它，都不能满足可达到时限——就像在火星探险者任务例子中一样（Jones，1997；Reeves，1997）。 635

设计过程可以看作是逐步明确的**规约**和**职责**的发展过程。这些规约定义了系统设计的特性，设计者在更详细的层次上操纵这些特性，而不能随意改变它们。在设计的层次体系中某个特定层次上没有给出规约的那些设计方面，则是更低层设计必须承担的职责的内容。

对设计的求精过程——也就是将职责转换成规约的过程——常常要服从主要由执行环境所施加的**约束**。执行环境的选择和如何使用它可能也是受约束的。例如，可能有一个需求规定要使用一个空间大小已经限死的处理器，或者，可能有一个认证需求规定处理器（或网络）的容量不得超过50%。

许多设计方法将逻辑设计同物理设计区分开来。逻辑设计关注满足应用的功能需求，它假设有一个足够快速的执行环境。而物理体系结构是将功能和建议的执行环境相结合以产生

一个完整的软件和硬件体系结构设计。

物理体系结构形成了进行下述断言的基础：一旦进行了详细设计和实现，所有的应用需求就得到满足。它进行时间性（甚至是可依赖性）分析，这种分析将确保（保证）系统一旦建立起来，就（在某些明确陈述的失效假设范围内）满足实时需求。为了进行这种分析，必须对系统（硬件和软件）的某些资源的使用进行初步评估。例如，可以对应用的时间特性进行初步评估，然后进行可调度性分析以确信一旦最终系统实现的时候，时限将能得到满足。

物理体系结构设计的关注重点是将功能体系结构映射到执行环境提供的设施上。在这个活动过程中，必须解决所有功能体系结构所做的假设和执行环境所提供的设施之间的不匹配问题。例如，功能体系结构可能假设所有功能在所有其他功能中是立即可得到的。当将功能映射到物理体系结构中的处理器上时，基础设施可能没有提供直接的通信路径，所以，这就必须补充额外的应用级路由功能。而且，基础设施可能只支持低级消息传递，而功能可能使用过程调用来
636 通信，所以，它必须提供一个应用级的RPC设施。显然，在产生物理体系结构的过程中，在完美复杂的执行环境和在功能体系结构中增加额外的应用设施之间有一个权衡。然而，如果应用不需要，执行环境就不要提供复杂完美的机制，这一点很重要，或者更糟的情况是，应用只需要比较原始的设施，却一定要试图用高级设施去构造它们。这通常被称为**抽象反转**。

一旦完成了初步的体系结构设计，详细设计就能认真地开始，应用的所有构件就产生了。当实现了这一点的时候，必须使用工具来分析这些构件，以测量应用的特征诸如它的最坏情况执行时间（或者它的复杂性，如果考虑可依赖性的话），去证实估计的最坏情况执行时间是否准确，或者由于某个模块是复杂的，因此软件易于出错，这导致需要设计的多样性。如果这些估计是不准确的（对一个新应用，这是经常发生的情况），那么要么修改详细设计（如果偏差小），要么设计者必须回到体系结构设计（如果存在严重问题）。如果估计是好的，那么就进行应用的测试。这应当包括代码实际执行时间的测量，发现的bug数等。图16-1说明了这个过程（这实际上是第2章介绍的HRT-HOOD设计方法支持的生命周期，第17章给出的案例研究使用了它）。

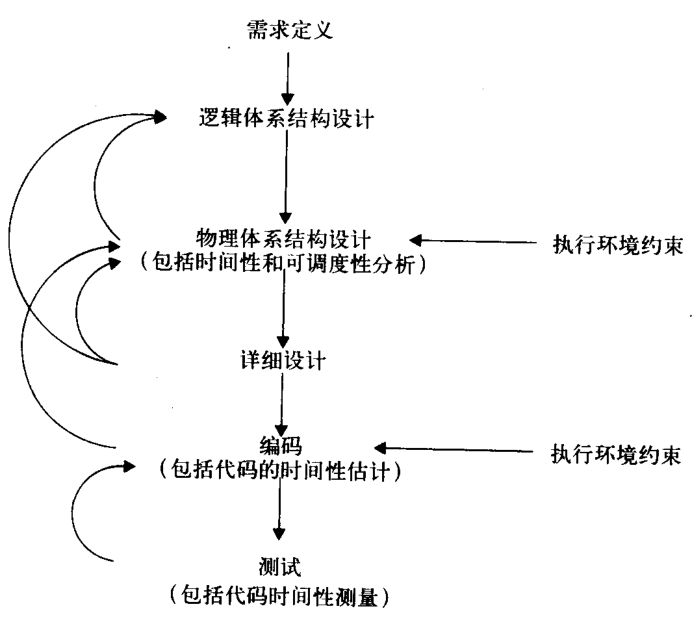


图16-1 硬实时生命周期

因此，重要的不是已编译代码的效率或操作系统开销，而是应该在生命周期中尽可能早地进行时间性分析。当然，使用一个粗劣低效的编译器是明显不合适的，这种低效必将导致一个低劣设计的产品。

16.2 剪裁执行环境

现代操作系统和同像Ada这样的语言相关联的运行时支持系统在功能上是齐平的，因为它们试图尽可能通用。显然，如果一个特定应用没有使用操作系统的某些特性，那么这有利于定制它运行时的形态。这个能力是很重要的，理由有三:

1）它避免了不必要的资源使用，即处理器时间或存储器;

2）它降低了需要验证其正确性的软件量;

3）许多开发标准要求清除‘死代码’。

本节将考虑Ada和POSIX提供的辅助这个活动的设施。通常实时Java不支持可选部件，因为这违背了*一次编写、到处运行*的原则。但是，它也接受某些部件无法实现这种现实，如果底层支持系统没有提供这样的功能的话。最明显的情况是与POSIX的信号接口的类。

16.2.1 Ada中的受限任务

Ada的实时系统附件允许程序员规定一组限制，运行时系统应该能识别出这些限制，并通过更有效的支持来“酬劳”它们。以下是通过编用给出限制的例子，这些限制在运行前检查和执行。 637~638

- `No_Task_Hierarchy`——这显著简化了对任务终止所需的支持。
- `No_Abort_Statement`——这将影响运行时支持系统的所有方面，因为不用担心任务在会合时、在保护操作中、异常传播时、等待子任务终止时被中止。
- `No_Terminate_Alternatives`——再次简化对任务终止所需的支持。
- `No_Task_Allocators`——允许将运行时系统配置为有静态数目的任务，消除对动态存储分配的需要。
- `No_Dynamic_Priorities`——简化了对任务优先级支持的许多方面，因为优先级将不会动态改变（除了使用高限优先级）。
- `No_Asynchronous_Control`——这将影响运行时支持系统的所有方面，因为不用担心任务在会合时、在保护操作中、异常传播时、等待子任务终止时收到异步事件。
- `Max_Select_Alternatives`——允许使用静态数据结构，消除了对动态存储分配的需要。
- `Max_Task_Entries`——再次允许使用静态数据结构，消除了对动态存储分配的需要。0值表示没有会合。
- `Max_Protected_Entries`——再次允许使用静态数据结构，消除了对动态存储分配的需要。0表示对保护对象不允许条件同步。
- `Max_Tasks`——指定最大任务数，因此允许运行时提供固定数目的静态支持结构。

注意，Ada的安全和保密附件将以上所有限制设为0（*就是没有任务了！*）。它也引入了一个进一步的限制，那就是不允许保护类型和对象。在安全至上应用领域，当前的做法禁止使用任务和中断。这是令人遗憾的，因为能够为任务设施定义一个既是可预测的又是服从分析的子集。还可能规定运行时系统使之能够按高级别的完整性得以实现。

未来十年Ada技术人员面临的挑战之一是证明并发程序设计是一个有效的和安全的技术，

甚至是对最苛刻的的需求也是这样。为了这个目标，第8届国际实时Ada学术讨论会（Burns，1999）定义了一个任务剖面（就是Ravenscar剖面）用于高完整性或性能敏感的应用。在Ravenscar剖面中，禁止使用以下特性:

- 任务类型和对象声明，除了在库级别外。因此没有任务类型的层次结构。
- 保护对象和任务对象的不加检查的回收（所以还有终了化）。可以允许这种对象的动态分配，但是只是在高完整性语言剖面的顺序部分允许动态分配其他对象的时候。
- 重排队。
- ATC（通过select then abort语句实现的异步控制转移）。
- 中止语句。
- 639 任务入口。
- 动态优先级。
- 包`Calendar`。
- 相对延迟。
- 保护类型，除了在库级别外。
- 多入口的保护类型。
- 带屏障的保护入口，在同一保护类型内声明单个布尔变量除外。
- 试图在单个的保护入口上排队多个任务。
- 除了高限锁（Ceiling locking）之外的上锁策略。
- 除了优先级内的FIFO之外的调度策略。
- 所有形式的选择语句。
- 用户定义的任务属性。

除了这些限制，实现可以假设没有任何程序任务会终止。注意，这些约束的大多数，但不是所有，都能用编用`Restrictions`定义。甚至在这些限制下，遵守Ravenscar 剖面的应用仍然可以有:

- 任务对象，限制如上所述。
- 保护对象，限制如上所述。
- 挂起对象。
- 原子编用的和短暂(volatile)编用。
- “Delay until”语句。
- 高限上锁策略和优先级内的FIFO分派。
- 属性Count（但不能在入口屏障内）。
- 任务标识符。
- 任务判别式。
- `Real_Time`包。
- 作为中断处理程序的保护过程。

只有子程序接口的保护类型可以实现简单的互斥。一种特定形式的保护入口（也就是，每个保护对象只有一个入口，并且那个入口最多只有一个调用者）可用于事件发信号机制，它能够支持偶发和非周期任务。

除了以上描述的特征，实时系统附件定义了许多的实现需求、文档需求和度量。使用这

些度量可以得到运行时系统（在处理器周期内的）的开销。它们也能指示哪个原语可能导致阻塞，哪个一定不会。

精确定义了时间性特征（即实时时钟和延迟原语）。例如，可以知道一个任务延时期满和它被放入运行队列之间的最长时间。在执行环境上下文中的应用的分析需要所有这些信息。 640

16.2.2 POSIX

POSIX由各种标准组成。有基本标准、实时扩展、线程扩展等。如果是在单个系统上实现，它将包含大量软件。为了帮助生成更紧凑的、符合POSIX规格说明的操作系统版本，建立了一套应用环境**剖面**（profile），定义剖面的想法是具体实现者可以支持一个或多个剖面。对于实时系统，已经定义了四个剖面：

- PSE50——最小实时系统剖面——用于小型单/多处理器嵌入式系统，以控制一个或多个外部设备，不需要操作员交互，没有文件系统。只支持单个的多线程进程。
- PSE51——实时控制系统剖面——对PSE50的扩充，用于多处理器，带文件系统接口和异步I/O。
- PSE52——专门用于实时系统的剖面——对PSE50的扩充，用于带存储管理单元的单或多处理器系统，包含多个多线程进程，但没有文件系统。
- PSE53——通用实时系统剖面——有混合运行实时和非实时进程的能力，运行在单/多处理器系统上，带有存储管理单元、海量存储设备、网络等。

表16-1说明了由PSE50、PSE51、PSE52提供的功能类型。

表16-1 POSIX 实时剖面功能

功　能	PSE50	PSE51	PSE52
pthreads	√	√	√
分叉	×	×	√
信号量	√	√	√
互斥锁	√	√	√
消息传递	√	√	√
信号	√	√	√
定时器	√	√	√
同步I/O	√	√	√
异步I/O	×	√	√
优先级调度	√	√	√
共享存储对象	√	√	√
文件系统	×	√	×

通常，一个POSIX系统可以不支持任何它选择的可选功能单元，所以可以更精细地控制要支持的功能。所有实时和线程扩展都是可选的。但是，遵守这些剖面中的一个意味着必须支持所有必要的功能单元。

16.3 调度模型

执行环境对应用的时间特性有明显的影响。在有软件内核的地方，在对应用进行可调度性分析时，必须考虑内核引起的开销。以下是许多实时软件内核的典型特征：

- 进程之间上下文切换的开销是不可忽略的，而且可能不是一个单个的值。切换到更高优先级周期进程（如时钟中断）的开销可能高于切换到更低优先级进程（在高优先级进程

运行的末尾）的开销。对于有大量周期进程的系统来说，有一个额外的开销用于操纵延时队列（对于周期任务，当它们执行时，是一个Ada的delay until语句）。

- 所有上下文切换操作是非抢占的。
- 处理中断的开销（除了时钟中断）和启动一个应用偶发进程的开销并非是不显著的。此外，对于DMA和通道程序控制设备，共享存储访问的影响可能对最坏情况性能有并非轻微的影响，这样的设备最好避免在硬实时系统中使用。
- 时钟中断（例如，每10ms）可能导致将周期进程从延迟队列移动到分派队列。这个操作的开销因要移动的进程的数量不同而异。

除了以上这些，可调度性分析必须考虑底层硬件的特性——例如高速缓存和管道的影响。

16.3.1 非微小的上下文切换时间的建模

大多数调度模型忽略上下文切换时间。但是，当上下文切换的总开销同应用代码的开销
641~642 相比不再是微不足道的时候，这种方法就过于简单了。图16-2说明了在一个典型的周期进程执行中发生的一些重要事件。

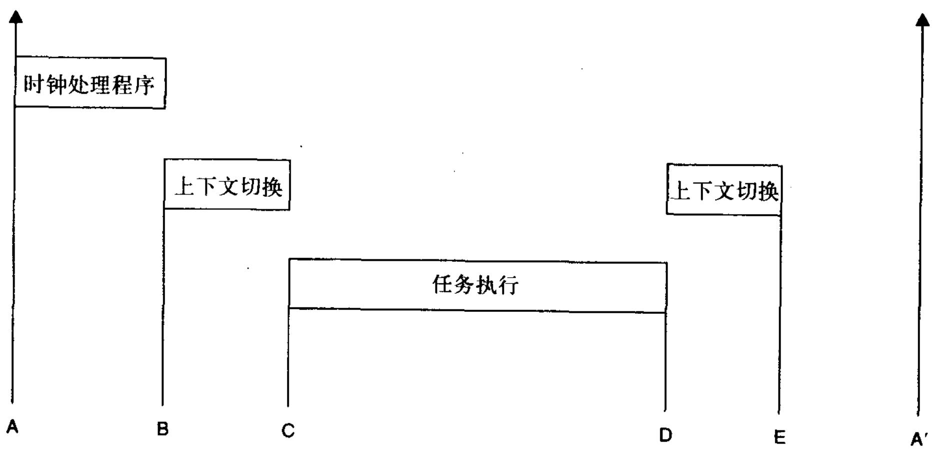


图16-2 执行进程时的开销

A——指定进程应当在某个预想时间启动的时钟中断（假设没有启动抖动和非抢占延迟，如果由于上下文切换导致禁止这些中断，那么时钟中断处理将延迟执行，在调度公式中，通过阻塞因子B表示考虑了这一点）。

B——时钟中断处理程序可以完成的最早时刻，它表示上下文切换到进程的开始时间（假设该进程是具有最高优先级的可运行进程）。

C——进程执行的实际开始

D——进程的完成（该进程在C和D之间可能被多次抢占）

E——离开该进程的上下文切换的完成时刻

A'——进程的下一次启动

这个进程的典型需求是它在下一次启动前完成（也就是D＜A'），或是在先于它下一次启动的某个时限之前。对于这两种情况中的任一种，D是一个重要时间，而E不是。需求的另一种形式在执行的开始和结束时间之间加一个界限（也就是D－C）。这发生在第一个动作是输入，而最后一个动作是输出时（在两者之间有一个时限需求）的情形。这些因素虽然影响进程自

己的时限的含义（因此影响了它的响应时间），却不影响该进程对低优先级进程的干预，在这里要计算两次上下文切换的全部开销。回忆基本的调度公式（13-7）: 643

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

现在该公式变为（只对周期进程）:

$$R_i = CS^1 + C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (CS^1 + CS^2 + C_j) \qquad (16\text{-}1)$$

其中，CS^1是初始上下文切换（切换进入进程）的开销，CS^2是在进程执行结束时离开它的上下文切换的开销。将进程放入延迟队列的开销（如果它是周期进程）并入C_i。注意，实际上这个值依赖于队列的大小，要将最大值并入C_i。

响应时间的测量从图16-2中的B点开始。为了从C点开始测量，将CS^1从公式中移除。为了从A点开始测量（进程的预想真正启动时间），需要测量时钟的行为（见16.3.3节）。

16.3.2 偶发进程的建模

对于由其他偶发或周期进程启动的偶发进程，公式（16-1）是一个有效的行为模型。但是，进程的计算时间C_i必须包括控制它启动的代理的阻塞开销。

当偶发进程由中断启动时，可能发生优先级反转。即使偶发进程有一个低优先级（由于它有一个很长的时限），但是中断本身将在高硬件优先级上执行。令Γ_s是由中断启动的偶发进程的集合。假设每个中断源与它启动的偶发进程有同样的到达特性。这些中断处理程序对每个应用进程的额外干扰由下式给出:

$$\sum_{k \in \Gamma_s} \left\lceil \frac{R_i}{T_k} \right\rceil IH$$

这里IH是中断处理的开销（和返回到已经启动该偶发进程的正在运行进程的开销）。

这种表示假设所有中断处理程序引起同样的开销，如果不是这种情况，那么必须为每个k定义IH。公式（16-1）现在变为:

$$R_i = CS^1 + C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (CS^1 + CS^2 + C_j)$$ 644

$$+ \sum_{k \in \Gamma_s} \left\lceil \frac{R_i}{T_k} \right\rceil IH \qquad (16\text{-}2)$$

16.3.3 实时时钟处理程序的建模

为了支持周期进程，执行环境必须访问实时时钟，实时时钟会在适当的时间产生中断。一个理想系统将使用一个间隔计时器，仅当需要启动周期进程时才产生中断。但是更通常的方法是以一个有规律的速率产生时钟中断（例如每10ms），中断处理程序必须决定是否必须启动一个或多个周期进程，或者不启动。可以用以前介绍的对于偶发进程的相同方法（见16.3.2节），为实时时钟处理的这种理想方法建模。对于有规律的时钟方法，必须建立一个更详细的模型，因为时钟处理程序的执行时间可能有很大的不同。对于这个中断处理程序（时钟周期

为10ms）表16-2给出了可能的时间。注意，如果假设发生这种最坏情况，10%以上的处理器都不得不分配给时钟处理程序。而且，所有这些计算发生在高（最高）硬件优先级上，因此将发生相当大的优先级反转。例如，利用表16-2给出的数字，被启动进程中最高优先级的应用进程将受到25个周期进程的LCM（最小公倍数）次1 048μs的干扰。如果进程是自己启动的，那么它将只受到88μs的干扰。这个时间间隔可用图16-2中的B－A表示。

表16-2 时钟中断处理的开销

队列状态	时钟处理时间，μs
队列上没有进程	16
队列上有进程，但都没有移走	24
移走1个进程	88
移走2个进程	128
移走25个进程	1 048

通常，从延迟队列移动N个周期进程到分派队列的开销可用下式表示：

$$C_{\text{clk}} = CT^c + CT^s + (N-1)CT^m$$

这里CT^c是不变开销（假设至少有一个进程总是在延迟队列），CT^s是移动单个进程的开销，CT^m是每个后续移动的开销。由于观察到移动一个进程的开销常常高于移动额外进程的附加开
645 销，所以这个模型是合适的。对于这里考虑的内核来说，这些开销如下：

CT^c	24μs
CT^s	64μs
CT^m	40μs

假设时钟中断处理程序每次运行都要消耗C_{clk}，为了减少这个计算开销，可以将这个负载分布在多个时钟滴答上。如果任一应用进程的最短周期$T_{\min}$大于时钟周期T_{clk}，那么这种方法是有效的。令M由下式确定：

$$M = \left\lceil \frac{T_{\min}}{T_{\text{clk}}} \right\rceil$$

如果M大于1，那么时钟中断处理程序的负载就可以分散到M次执行。在这种情况下，时钟中断处理程序就能被建模为一个具有周期$T_{\min}$和计算时间C'_{clk}的进程：

$$C'_{\text{clk}} = M(CT^c + CT^s) + (N-M)CT^m$$

这里假设$M <= N$。

公式（16-2）现在变为：

$$\begin{aligned} R_i = {} & CS^1 + C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (CS^1 + CS^2 + C_j) \\ & + \sum_{k \in s} \left\lceil \frac{R_i}{T_k} \right\rceil IH \\ & + \left\lceil \frac{R_i}{T_{\min}} \right\rceil C'_{\text{clk}} \end{aligned} \tag{16-3}$$

为了进一步改进这个模型，需要时钟中断处理程序实际执行的更准确表示。例如，只使用CT^c和CT^s，可以容易地导出以下公式：

$$R_i = CS^1 + C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (CS^1 + CS^2 + C_j)$$
$$+ \sum_{k \in \Gamma_s} \left\lceil \frac{R_i}{T_k} \right\rceil IH + \left\lceil \frac{R_i}{T_{\text{clk}}} \right\rceil CT_c$$
$$+ \sum_{g \in \Gamma_p} \left\lceil \frac{R_i}{T_g} \right\rceil CT_s \qquad (16\text{-}4)$$ 646

这里Γ_p是周期进程的集合。

在以上模型中并入时钟中断处理的三参数模型留给读者练习（见练习16.2）。

16.3.4 高速缓存对最坏情况执行时间分析的影响

在13.12.1节，已经提到了对于在现代处理器上执行的进程进行WCET分析的问题。尤其是要为处理器的高速缓存和管道的行为建模。在公式（16-4）中，C_i和C_j的值受到它们的影响。如果通过详细分析处理器的体系结构来计算这些值，那么在调度公式中，必须计入中断引起的抢占。否则，使用的这些值是乐观的。幸运的是，对于硬实时系统，必须对中断发生的频率加以限制。每个中断处理程序被视为一个高优先级偶发进程，并用对待高优先级周期进程同样的方式来考虑它们。公式（16-4）已经规定了进程i执行时可以发生的抢占数目。它就是在进程i的响应时间期间每个更高优先级进程的启动次数。每次抢占都可能刷新高速缓存和管道。这导致了以下融入抢占的方法。假设C_i是用模型计算出的进程的最坏情况执行时间，这种模型解释了没有中断发生时高速缓存和管道的获益。计算由中断产生的最大可能的惩罚γ，这是重填高速缓存和管道花费的时间。现在可以修改公式（16-4）以计算中断对进程i的影响(Busquets and Wellings，1996)：

$$R_i = CS^1 + C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (CS^1 + CS^2 + C_j)$$
$$+ \sum_{k \in \Gamma_s} \left\lceil \frac{R_i}{T_k} \right\rceil (IH + \gamma) + \left\lceil \frac{R_i}{T_{\text{clk}}} \right\rceil (CT_c + \gamma)$$
$$+ \sum_{g \in \Gamma_p} \left\lceil \frac{R_i}{T_g} \right\rceil CT_s \qquad (16\text{-}5)$$

当然，这个公式是非常悲观的，因为不是所有的抢占都需要对整个高速缓存进行重填。而且，某些被替换的存储块会已经被替换过了。一个不太悲观的方法尝试标识出高速缓存块的号码。 647

16.4 硬件支持

当在任何实时问题的解决方案中引入并发进程的时候，就出现了诸如调度、进程间通信等的开销。16.3.3节试图在可调度性分析中为这些开销建模。通过提供直接硬件支持，已经做出了几个尝试以减少这些开销。本节简要考虑两个硬件内核。第一个是为高效运行occam2程

序而设计的传输机，第二个是Ada任务协处理器（ATAC）。

在最近几年，已经出现直接用硬件来支持Java虚拟机（例如，Sun公司的picoJava处理器（Sun Microsystems，2000）或来自aJile Systems公司的aJ-100（aJile Systems，2000））。这些支持超出了只对并发执行的支持，并试图去改善Java字节码解释执行的性能。

16.4.1 传输机和occam2

传输机被设计成一个occam2机器，它在单个芯片上有一个32位的处理器、一个64位的浮点协处理器、内部存储器和一些通信链路，这些通信链路可以直接同其他传输机通信。利用一个连续的地址空间，一个地址总线将外部存储器和内部提供的存储器联合在一起。通常，一个传输机有16k字节的内部存储器，实际上，这些存储器可看作一个用于执行进程的巨大的非共享寄存器集合。

通信链路通过链路接口同主处理器相联。这些接口能独立管理链路的通信（包括直接内存访问）。结果，传输机能同时在所有链路上通信（双向）、执行一个内部进程和进行一个浮点操作。

传输机有一个精简指令集，但是有一个仅有三个寄存器的操作栈。每条指令的设计目标是使它在occam2编译器的代码生成阶段都是有用的；虽然允许直接用汇编语言编程，但是，在指令集的设计中没有考虑这一点。在一个精简指令集的机器中，不是所有指令都立即可用，那些直接可访问的指令正好是真正的occam2 程序生成的那些指令。

令人遗憾的是传输机只支持一个受限的优先级模型。但是，通过这个限制，可以提供一个基本上是用硬件（硅刻）实现的运行时支持系统。这种体系结构（加上只在操作栈为空时才会发生上下文切换这种自明之理）使得上下文切换时间是非常短的。

虽然单个传输机的操作特征给人以深刻印象，但是只有它们聚集成群时，才能发挥它们的全部潜能。传输机使用点到点通信，它的缺点是如果没有直接的链路相连，一个消息可能不得不通过一个中间媒介来转发到目的地。不过由于链路传输速率非常高而传输失败的比率
648 非常低，所以给了实时引擎相当大的能力和可靠性。

16.4.2 ATAC和Ada

已经有了几个生产Ada机器的尝试，例如（Ericsson，1986；Runner and Warshawsky，1988）。这里考虑由Roos（1991）设计的Ada任务协处理器（ATAC）。

ATAC是一个为支持Ada83任务和时钟模型而设计的硬件设备。它也期望支持Ada95的某些特性，例如支持优先级继承和delay until。它的目标是减轻应用CPU支持Ada任务的负担，因而使任务能高效运行，而没有通常由Ada运行时支持系统引起的开销。

CPU和ATAC之间的通信基于标准的存储－映射式的读和写指令。一个原语操作集合提供了接口，包括：

- `CreateTask`——创建一个新任务；
- `ActTasks`——激活一个或多个已创建任务；
- `Activated`——向正创建的任务发信号激活；
- `EnterTBlock`——进入一个新任务块；
- `ExitTBlock`——等待依赖任务退出任务块；

- EntryCall——发出一个入口调用;
- TimedECall——发出一个限时入口调用;
- SelectArg——打开一个备选项;
- SelectRes——在选择语句中选择一个备选;
- RndvCompl——在会合完成后设置调用者为可运行状态;
- Activate——使一个挂起任务变成可运行的;
- Suspend——挂起当前任务;
- Switch——执行重调度;
- Delay——延迟一个任务。

ATAC也防守所有中断，并且仅当一个较高优先级任务变成可运行时才中断CPU。用一个内部时钟来支持Ada的延迟设施和包calendar。

ATAC的整体目标是增强Ada任务的性能，使它的性能超过纯软件运行时系统两个数量级。

小结

执行环境是任何已实现的实时系统的关键部分。它支持应用，但也引入开销，并对应用可能使用的设施施加约束。可以使用一个成熟的操作系统（OS）来提供执行环境，但是由于以下原因通常拒绝这样做: 649

- OS的大小（即占有多少内存）;
- （诸如上下文切换的）关键功能的效率;
- 整个OS的复杂性和由此产生的可靠性。

这一章说明了如何剪裁执行环境以适应应用的特定需要，如何为它的开销建模，以及如何提供硬件支持。本书的其他部分也介绍了关于执行环境的一些重要问题。例如:

- 它在提供损害隔离机制（也就是防火墙）中的作用;
- 它在错误检测中的作用;
- 它在分布式系统中对通信的支持作用。

第二个问题有多个方面。可以监视应用运行的不同方面（数组越界、内存越界、超时）。为了隔离有故障部件和为故障排除产生维护数据，还可以在后台运行“内部测试”设施以实际运用硬件的各个部分。

因为执行环境的许多特性对范围宽广的应用都是很重要的，因此有必要重用可信的构件，向着提供标准环境前进。标准化语言和操作系统接口的使用将有助于实现这个目标。

相关阅读材料

Allen, R. K., Burns, A. and Wellings, A. J. (1995) Sporadic Tasks in Hard Real-Time Systems. *Ada Letters*, XV(5), 46–51.

Burns, A., Tindell, K. and Wellings, A. J. (1995) Effective Analysis for Engineering Real-Time Fixed Priority Schedulers. *IEEE Transactions on Software Engineering*, 21(5), 475–480.

Venners, B. (1999) *Inside the Java 1.2 Virtual Machine*. New York: Osborne McGraw-Hill.

练习

16.1 实时系统的程序员应该知道所有实现语言特性的实现开销吗？

16.2 建立一个时钟处理的模型，在模型中加入三个参数CT^c、CT^s、CT^m（见16.3.3节）。

16.3 除了用时钟中断来调度周期进程，只访问实时时钟会有什么结果？

650～651 16.4 一个周期为40ms的周期进程由一个粒度为30ms的时钟中断来控制。如何计算这个进程的最坏情况响应时间？

第17章 Ada案例研究

在本章中介绍一个案例研究，它包括本书中描述过的许多设施。理想情况是应当分别用Ada、实时Java、C和（POSIX）以及occam2给出这个案例研究。令人遗憾的是空间有限，所以研究只限于Ada。

17.1 矿井排水

选择的例子以文献中经常出现的一个例子为基础。它涉及采矿环境中管理一个简化的水泵控制系统所需的软件（Kramer等，1983；Sloman and Kramer，1987；Shrivastava等，1987；Burns and Lister，1991；Joseph，1996；de la Puente等，1996），它具有嵌入式实时系统的许多有代表性的特征。假设此系统在具有存储－映射式I/O体系结构的单个处理器上实现。

这个系统将汇集于矿井底的一个池子中的水用水泵抽到地面。主要安全需求是当矿井中甲烷气体的含量达到高位值时，就不得操作水泵，否则有爆炸的危险。系统的简单示意图如图17-1所示。

控制系统和外部设备之间的关系如图17-2所示。注意，只有低水位和高水位传感器经由中断通信（用虚箭头表示）；所有其他设备要么被轮询，要么是直接控制。 653

17.1.1 功能需求

这个系统的功能规格说明可分为四部分：水泵操作、环境监控、操作员交互和系统监控。

1. 水泵操作

水泵控制器的所需行为是监控池中的水位。当水到达高水位（或操作员请求）时，水泵就打开，从池中排水，直到水到达低水位。这时（或操作员请求），水泵被关上。如果需要的话，可以检测管道中的水流。

只有在矿井中的甲烷气含量低于一个临界值时，才允许水泵操作。

2. 环境监控

必须监控环境，以检测甲烷在空气中的含量，有一个含量水平，如果高于它的话，采煤和操作水泵是不安全的。监控工作还测量矿井中一氧化碳的含量和是否有足够的空气流动。如果瓦斯含量或空气流到达临界值，就必须发出警报。

3. 操作员交互

系统从地面经由一个操作员控制台控制。所有关键事件都要通知操作员。

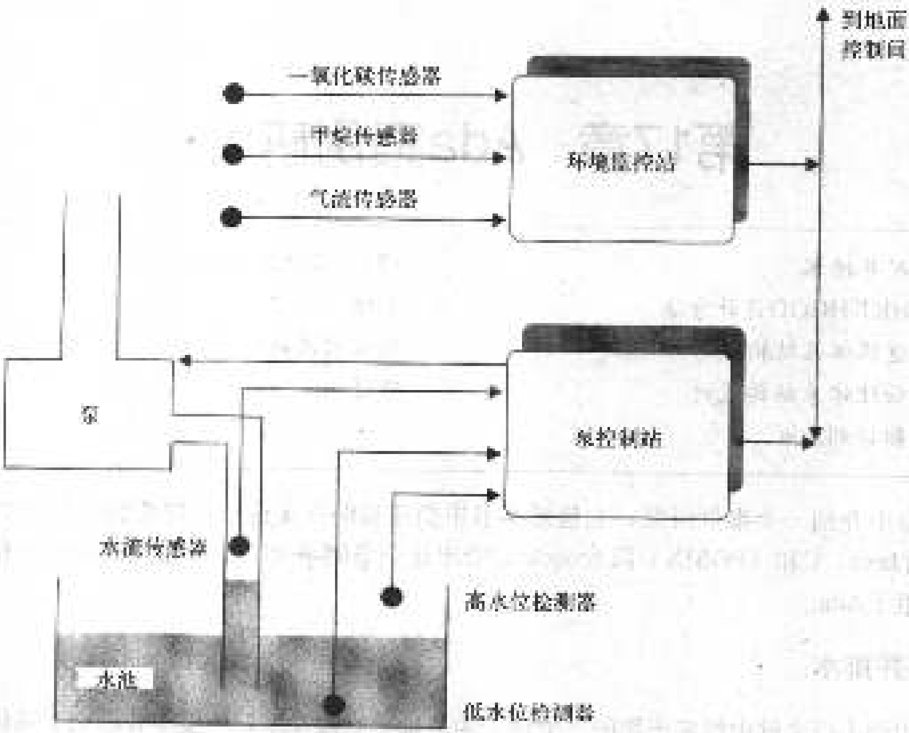


图17-1 矿井排水控制系统

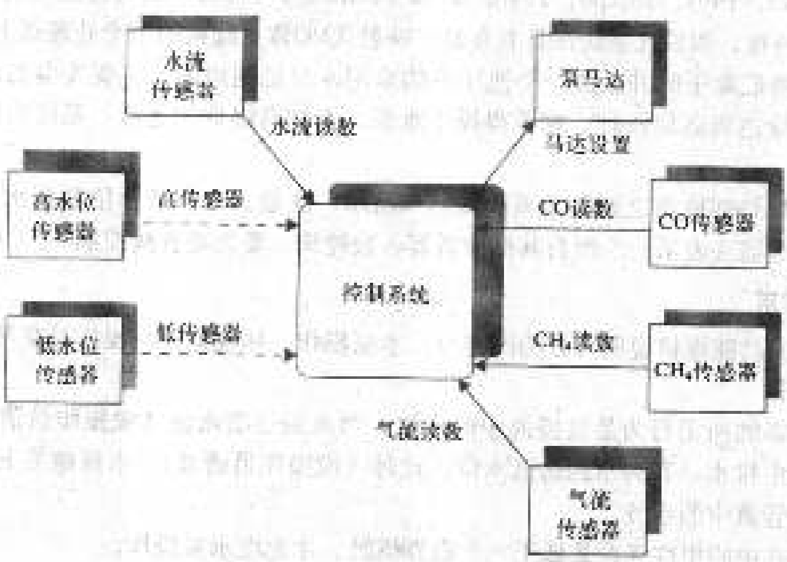


图17-2 说明外部设备的图

4. 系统监控

所有系统事件都要存储到档案数据库中，并可根据请求检索和显示。

654–655 17.1.2 非功能需求

非功能需求可分为三个：时间性的、可依赖性和安全性。本案例研究主要关心时间性需求。

所以不讨论可依赖性和安全性（对可依赖性和安全性方面的完整研究见文献（Burns and Lister, 1991））。

与系统动作的时间性相关的需求有若干个，以下内容来自参考文献[Burns and Lister (1991)]:

(i) 监控周期

读取环境传感器的最大周期受法规支配。对于此例的目的而言，假设这些周期对所有传感器是相同的，即100ms。对于甲烷，可能有一个更严格的需求，这要基于水泵的距离并确保当甲烷含量到达临界高值时水泵不操作。这在下面的（ii）中讨论。在15.8节，描述过怎么分析一个设备驱动程序。在本案例研究中，将把“周期移位法”用于CH_4和CO传感器。这些环境传感器每次都需要40ms以使读数成为可用的。所以，它们需要一个60ms的时限。

水流对象周期性地执行，它有两个作用。在水泵操作时它检查有无水流；但在水泵关上（或禁用）时，它还检查水是否已经停止流动。后一种检查用于证实水泵确实已经停止。由于水流中的时间滞后，这个对象被给予1秒的周期，并且它使用两个相邻读数的结果以确定水泵的实际状态。为确信两个相邻读数真的有1秒钟的间隔（近似的），给这个对象以40ms的紧时限（即，两个读数至少隔960ms，但不多于1040ms）。

假设水位检测器是事件驱动的，系统必须在200ms内响应。这个应用的物理过程指出，来自两个水位指示器的中断之间必须至少有6秒钟。

(ii) 崩溃时限

为避免爆炸，有一个时限：一旦甲烷含量超过临界值，必须将水泵断开。这个时限同甲烷采样周期有关，同甲烷积累的速度有关，并同甲烷含量的临界值和引起爆炸的含量值之间的安全差值有关。使用传感器的直接读数，这个关系可以表示成一个不等式:

$$R(T+D)<M$$

其中，

R——甲烷积累的速度；

T——采样周期；

D——崩溃时限；

M——安全差值。 656

如果使用“周期移位法”，就再需要一个时间周期:

$$R(2T+D)<M$$

注意，周期T和时限D之间可以互相权衡，而且两者都可以同安全差值M权衡。周期或时限越长，安全差值就一定越稳健；周期或时限越短，就越接近于矿井可操作的安全界限。所以，设计者可以改变D、T或M中的任意一个，以满足时限和周期性需求。

在这个例子中，假设甲烷矿穴的出现可以引起甲烷含量快速上升，所以假设200ms的时限需求（从甲烷上升到水泵被关闭）。这可以通过设置甲烷传感器的速度为80ms、时限为30ms满足。注意，这个含量将确保从传感器取到正确的读数（即两个读数之间的时间至少是50ms）。

(iii) 操作员信息时限

检测到临界的高甲烷或一氧化碳读数时必须在1秒钟之内通知操作员，对临界低空气流读

数是2秒，水泵操作失败是3秒。相比其他时间性需求而言，这些需求是容易满足的。

表17-1概略地确定了这些传感器的周期或最小到达间隔时间和时限。

表17-1 周期和偶发实体的属性

	周期/偶发	“周期”	时 限
CH_4传感器	P	80	30
CO传感器	P	100	60
气流	P	100	100
水流	P	1 000	40
水位检测器	S	6 000	200

17.2 HRT-HOOD设计方法

在第2章介绍过HRT-HOOD开发过程。它的关注重点是逻辑和物理体系结构的设计和使用一种基于对象的记号。本章使用这个方法的简化版本。

657 HRT-HOOD通过提供不同的对象类型方便系统的逻辑体系结构设计。这些类型是:

- **被动对象**——这种重入对象不能控制什么时候执行对其操作的调用，也不主动地调用其他对象中的操作。
- **主动对象**——这种对象可以控制什么时候执行对其操作的调用、并可主动调用其他对象中的操作。主动对象是最一般的对象类，对它们没有限制。
- **保护对象**——这种对象可以控制什么时候执行对其操作的调用，但不主动调用其他对象中的操作；通常保护对象不可以有任意的同步约束，并且它们施加给其调用者的阻塞时间必须是可分析的。
- **循环对象**——表示周期活动的对象，它们可以主动地调用其他对象中的操作并且只有非常受限的接口。
- **偶发对象**——表示偶发活动的对象，偶发对象可以主动调用其他对象中的操作；每个偶发对象有一个**单独**的操作以供调用这个偶发对象。

使用HRT-HOOD设计的硬实时系统将只会在末端级（即在全部设计分解之后）包含循环、偶发、保护和被动对象。因为主动对象不能被完全分析，因此只允许它们出现在背景活动中。在主系统的分解过程中可以使用主动对象，但必须在到达末端级之前被变换到上述几种对象中的一种。

图17-3说明了HRT-HOOD设计的图式表示。它指出“Parent”对象被层次式分解为两个子对象（“Child1”，“Child2”）。这个“Parent”对象是个主动对象（由对象左上角的字母A指出）并有两个操作“操作1”和“操作2”。每个子对象实现一个操作的功能。为了实现“操作1”的功能，“Child1”使用了由“Child2”（一个被动对象）提供的设施和一个“Uncle”。Uncle对象是一个在更高层分解中定义的对象。这个图还给出了数据流和异常流。

17.3 逻辑体系结构设计

逻辑体系结构设计致力于同执行环境施加的物理约束（例如，处理器速度）无关的需求。17.1.1节所确认的功能需求属于这一类。对其他系统需求的考虑放到物理体系结构的设计中，

658 这在后面描述。

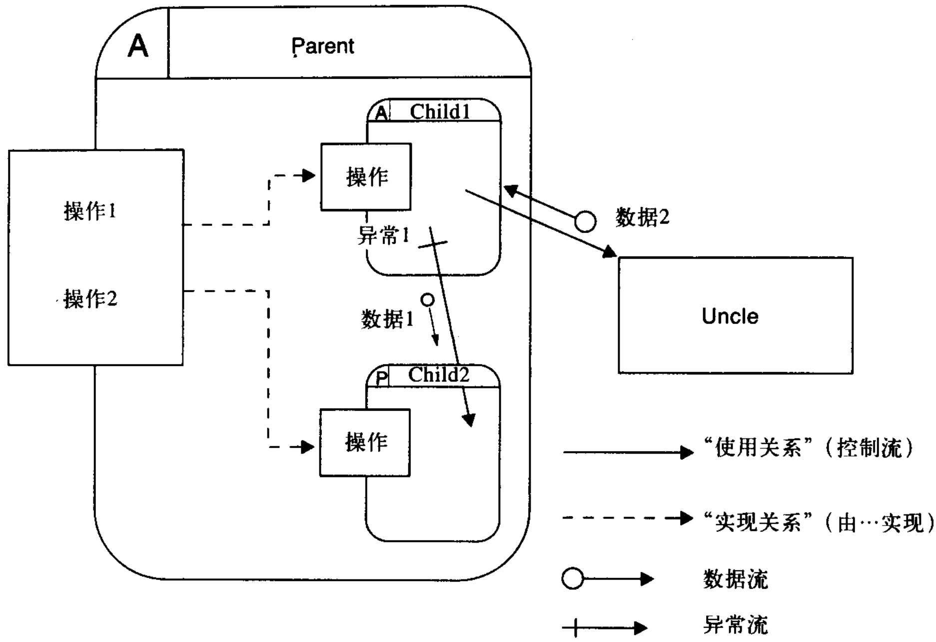


图17-3 HRT-HOOD图式记号

17.3.1 第一级分解

建立逻辑体系结构的第一步是确认可用来构建系统的合适的对象类。系统的功能需求建议四个不同的子系统:

1）水泵控制子系统——负责水泵的操作

2）环境监控子系统——负责监控环境

3）操作员控制台子系统——负责到操作员的接口

4）数据记录子系统——负责记录操作和环境数据

图17-4说明了这个分解。水泵控制系统有四个操作:“不安全”（not safe）和“安全”（safe）操作由环境监控器调用，它们给水泵控制器指出水泵的操作是否安全（归因于环境中甲烷的含量等级）。“请求状态”（request status）和“水泵设置”（set pump）操作由操作员控制台调用。作为一个补充的可靠性特性，在水泵启动前，水泵控制器将总是检查甲烷含量等级是否处于低等级（通过调用环境监控器中的“安全检查”（check safe））。如果水泵控制器发现水泵不能启动（或者水泵被假想打开而不出现水流），则引发一个操作员警报。 659

环境监控器有单一操作“安全检查”，由水泵控制器调用。

操作员控制台有报警（alarm）操作，它由水泵控制器调用，如果有任何读数过高的话，也由环境监控器调用。收到了报警调用后，操作员控制台能够请求水泵的状态并尝试通过直接操作水泵去改变高水位传感器和低水位传感器的读数。然而，在后一种情况，如果还在做甲烷含量检查，就通过一个正被使用的异常去通知操作员水泵不能打开。

数据记录器有六个操作，它们都只是一些由水泵控制器和环境监控器调用的数据记录动作。

17.3.2 水泵控制器

水泵控制器的合适分解如图17-5所示。水泵控制器被分解为三个对象。第一个对象控制水

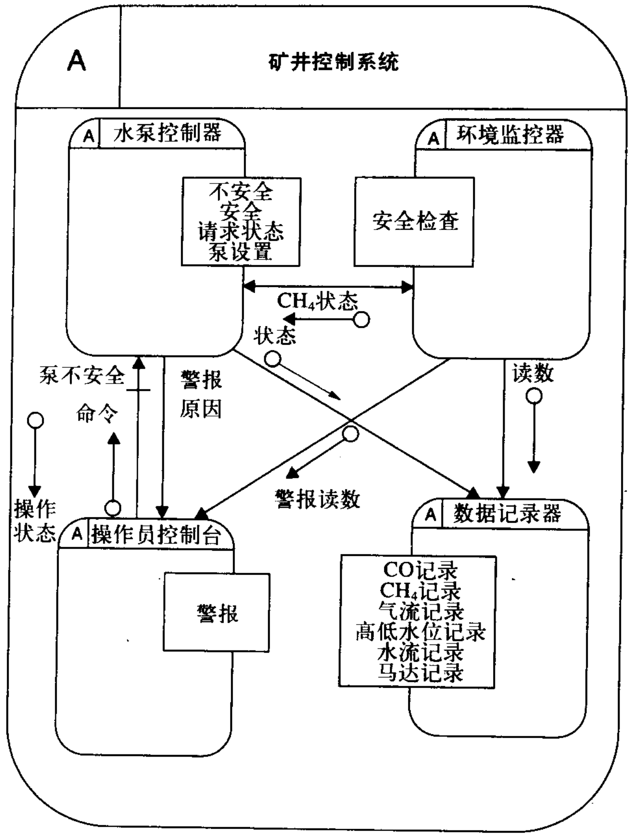


图17-4 控制系统的第一级层次分解

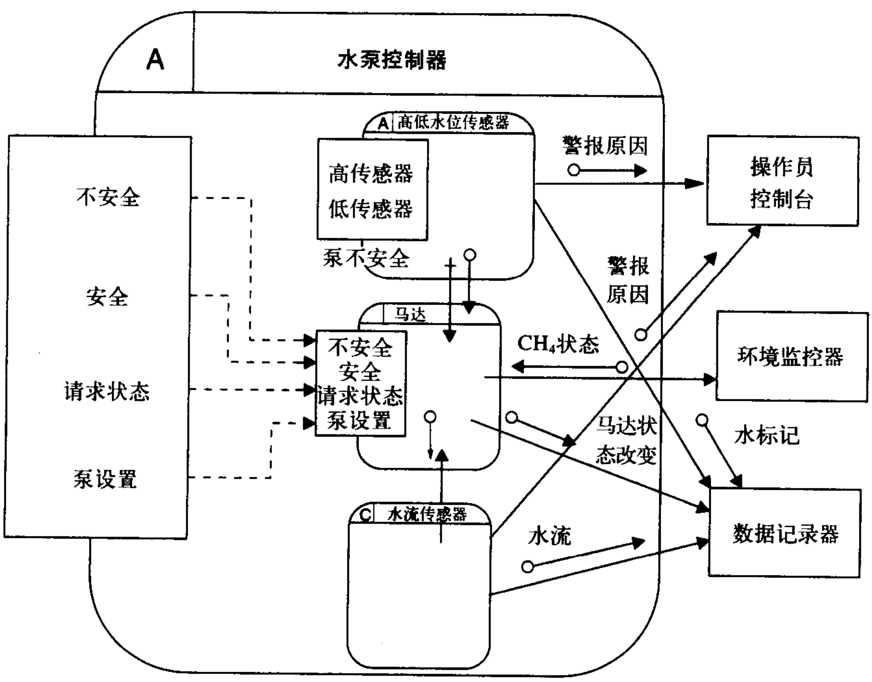


图17-5 水泵对象的层次分解

泵马达。因为这个对象只是简单地响应命令，它的操作需要互斥，并且不主动调用其他对象，所以它是一个保护对象。水泵控制器的所有操作都是由马达对象实现的。因为系统是实时的，这些操作都不能被随意阻塞（虽然它们需要互斥）。马达对象将向其所有权对象发出调用。

另外两个对象控制水传感器。水流传感器对象是一个循环对象，它持续地监控矿井中的水流。高-低水位传感器是一个主动对象，它处理来自高-低水位传感器的中断。它被分解为一个保护对象和一个偶发对象，如图17-6所示。 660~661

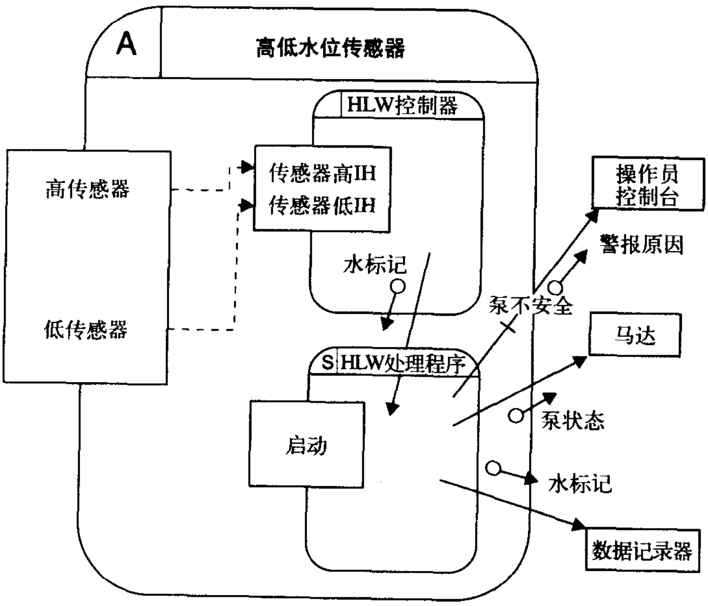


图17-6 高-低水位传感器的分解

17.3.3 环境监控器

环境监控器分解成四个末端对象，如图17-7所示。其中有三个循环对象，监控矿井环境中的CH_4等级、CO等级和气流。只有CH_4等级是由系统中的其他对象请求的，所以，用一个保护对象去控制对当前值的访问。

17.3.4 数据记录器和操作员控制台

本案例研究不涉及数据记录器和操作员控制台的细节。然而，有一个需求，就是只使实时线程延迟一段有界的时间。所以，假设它们的接口包含保护对象。

17.4 物理体系结构设计

HRT-HOOD通过下列设施支持物理体系结构的设计： 662

- 允许与对象关联时间性属性，
- 提供一个可以定义可调度性方法和进行末端对象分析的框架，
- 提供抽象，使设计者可用以表达对时间性出错的处理。

在17.1.2节确认的非功能性的时间性需求被变换成在方法和线程上的标注（annotation）。

为说明第13章中描述的分析，将使用固定优先级调度，并进行响应时间形式的分析。表17-2总结了在逻辑体系结构中引入对象的时间性属性。

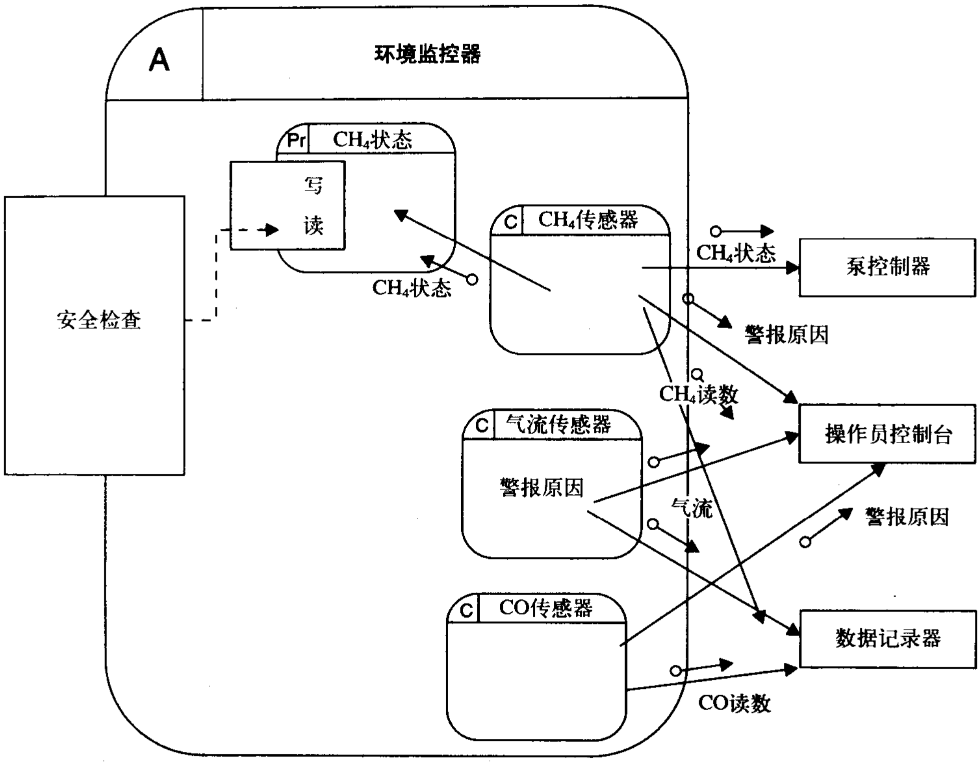


图17-7 环境监控器的层次分解

表17-2 设计对象的属性

	类型	"周期"	时限	优先级
CH_4传感器	周期	80	30	10
CO传感器	周期	100	60	8
气流传感器	周期	100	100	7
水流传感器	周期	1 000	40	9
HLW处理器	偶发	6 000	200	6
马达	保护			10
HLW控制器	保护			11
CH_4状态	保护			10
操作员控制台	保护			10
数据记录器	保护			10

调度分析

一旦开发出了代码，就必须进行分析以得到最坏情况执行时间。如在15.8节指出的，这些值可以从直接测量或通过对硬件建模得到。所导出的代码都不是特别多，所以有理由假设低速处理器就够了。表17-3包含了设计中的每个对象的最坏情况执行时间的几个有代表性的值

(以毫秒计)。注意，每个线程的时间包括花费在其他对象中执行的时间、花费在执行异常处理程序上的时间和相关上下文切换的时间。为了给中断处理程序的效果建模，引入了一个“伪”偶发对象（最长异常处理程序执行时间为2ms)。

表17-3 最坏情况执行时间

	类 型	WCET
CH_4传感器	周期	12
CO传感器	周期	10
气流传感器	周期	10
水流传感器	周期	10
HLW处理器	偶发	20
中断	偶发	2

执行环境设置了它自己的一组重要参数——这些在表17-4中给出。注意，时钟中断有足够粒度以确保周期性进程没有启动抖动。

表17-4 开销

	符 号	时 间
时钟周期	T_{CLK}	20
时钟开销	CT^c	2
单个任务移动的开销	CT^s	1

所有线程的最长阻塞时间发生在操作员控制台发出一个对马达对象的调用的时候。可以假设发出这个调用的线程有低优先级。此保护操作的最坏情况执行时间假设为3ms。 663~664

现在可将上述信息综合，以便为系统中所有线程的响应时间提供一个全面的分析。表17-5中给出了这个分析。分析的结论是所有时限都能满足。

表17-5 分析结果

	类 型	T	B	C	D	R
CH_4传感器	周期	80	3	12	30	25
CO传感器	周期	100	3	10	60	47
气流传感器	周期	100	3	10	100	57
水流传感器	周期	1 000	3	10	40	35
HLW处理器	偶发	6 000	3	20	200	79

17.5 翻译到Ada

HRT-HOOD支持到Ada的系统化翻译。对于每个末端对象，生成两个包：第一个包只包含定义对象的实时属性的一组数据类型和变量；第二个包包含对象自身的代码。

图17-4所示的每个对象都潜在地能够在一个单独的处理器上实现。然而，为了这个例子的目的，采用单个处理器实现。

对水泵控制器适宜的分解已在图17-5中展示，高－低水位传感器对象在图17-6中给出。现在可以给出这些对象的代码了。

17.5.1 水泵控制器对象

1. 马达

665 首先给出马达对象的实时属性。为了简明，只给出高限优先级属性。

```
package Motor_Rtatt is
  Ceiling_Priority: constant := 10;
end Motor_Rtatt;
```

马达对象的接口是:

```
package Motor is -- 保护的
  type Pump_Status is (On, Off);
  type Pump_Condition is (Enabled, Disabled);

  type Motor_State_Changes is (Motor_Started,
       Motor_Stopped, Motor_Safe, Motor_Unsafe);

  type Operational_Status is
    record
      Ps : Pump_Status;
      Pc : Pump_Condition;
    end record;

  Pump_Not_Safe : exception;

  procedure Not_Safe;
  procedure Is_Safe;

  function Request_Status return Operational_Status;
  procedure Set_Pump(To : Pump_Status);
end Motor;
```

马达的状态由两个变量定义。一个指出水泵应当是开还是应当关，另一个是允许或禁用水泵。当水泵的操作不安全时，它被禁用。类型Motor_State_Changes用于向数据记录器指出状态的改变。

马达的状态迁移图如图17-8所示。只在“开-允许”（允许操作）状态才能实际对水泵操作。

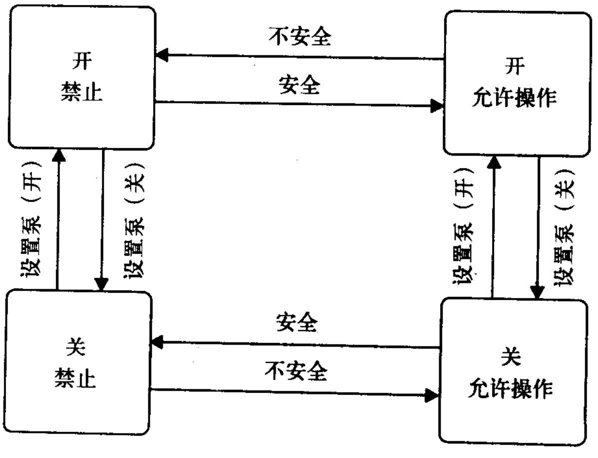


图17-8 马达的状态迁移图

所以，包体实现状态改变。因为这些必须是原子地完成的，所以使用保护对象。本章中，为实现同步约束生成的所有保护对象被称为Agent（代理）。以下代码假设在名为Device_Register_Types的包中声明了设备寄存器类型。

```
package Device_Register_Types is
  Word : constant := 2;        -- 一个字中两个字节
  One_Word : constant := 16; -- 一个字中16个二进制位
  -- 寄存器位段类型
  type Device_Error is (Clear, Set);
  type Device_Operation is (Clear, Set);
  type Interrupt_Status is (I_Disabled, I_Enabled);
  type Device_Status is (D_Disabled, D_Enabled);

  -- 寄存器类型本身
  type Csr is

  record
    Error_Bit   : Device_Error;
    Operation   : Device_Operation;
    Done        : Boolean;
    Interrupt   : Interrupt_Status;
    Device      : Device_Status;
  end record;
-- 寄存器字段的位表示
for Device_Error use (Clear => 0, Set => 1);
for Device_Operation use (Clear => 0, Set => 1);
for Interrupt_Status use (I_Disabled => 0,
                          I_Enabled => 1);
for Device_Status use (D_Disabled => 0,
                       D_Enabled => 1);
for Csr use
  record at mod Word;
    Error_Bit  at 0 range 15 ..15;
    Operation  at 0 range 10 ..10;
    Done       at 0 range 7 .. 7;
    Interrupt  at 0 range 6 .. 6;
    Device     at 0 range 0 .. 0;
  end record;
  for Csr' Size use One_Word;
  for Csr' Alignment use Word;
  for Csr' Bit_order use Low_Order_First;
end Device_Register_Types;
```

包Motor的体只包括保护类型Agent的实现。外部操作只是调用保护子程序。 666~667

```
with Data_Logger;
with Ch4_Status; use Ch4_Status;
with Device_Register_Types; use Device_Register_Types;
with System; use System;
with Motor_Rtatt; use Motor_Rtatt;
with System.Storage_Elements; use System.Storage_Elements;
package body Motor is
  Control_Reg_Addr : constant Address := To_Address(16#Aa14#);
```

```
Pcsr : Device_Register_Types.Csr :=
     (Error_Bit => Clear, Operation => Set,
      Done => False, Interrupt => I_Enabled,
      Device => D_Enabled);
for Pcsr' Address use Control_Reg_Addr;

protected Agent is
  pragma Priority(Motor_Rtatt.
                  Ceiling_Priority);
  procedure Not_Safe;
  procedure Is_Safe;
  function Request_Status return Operational_Status;
  procedure Set_Pump(To : Pump_Status);
private
  Motor_Status : Pump_Status := Off;
  Motor_Condition : Pump_Condition := Disabled;
end Agent;

procedure Not_Safe is
begin
  Agent.Not_Safe;
end Not_Safe;

procedure Is_Safe is
begin
  Agent.Is_Safe;
end Is_Safe;

function Request_Status return Pump_Status is
begin
  return Agent.Request_Status;
end Request_Status;

procedure Set_Pump(To : Pump_Status) is
begin
  Agent.Set_Pump(To);
end Set_Pump;

protected body Agent is
  procedure Not_Safe is
  begin
    if Motor_Status = On then
      Pcsr.Operation := Clear; -- 关掉马达
      Data_Logger.Motor_Log(Motor_Stopped);
    end if;
    Motor_Condition := Disabled;
    Data_Logger.Motor_Log(Motor_Unsafe);
  end Not_Safe;

  procedure Is_Safe is
  begin
    if Motor_Status = On then
      Pcsr.Operation := Set; -- 启动马达
```

```
      end if;
      Motor_Condition := Enabled;
      Data_Logger.Motor_Log(Motor_Safe);
    end Is_Safe;

    function Request_Status return Operational_Status is
    begin
      return (Ps => Motor_Status, PC => Motor_Condition);
    end Request_Status;

    procedure Set_Pump(To : Pump_Status) is
    begin
      if To = On then
        if Motor_Status = Off then
          if Motor_Condition = Disabled then
            raise Pump_Not_Safe;
          end if;
          if Ch4_Status.Read  = Motor_Safe then
            Motor_Status := On;
            Pcsr.Operation := Set; -- 打开马达
            Data_Logger.Motor_Log(Motor_Started);
          else
            raise Pump_Not_Safe;
          end if;
        end if;
      else
        if Motor_Status = On then
          Motor_Status:= Off;
          if Motor_Condition = Enabled then
            Pcsr.Operation := Clear; -- 关掉马达
            Data_Logger.Motor_Log(Motor_Stopped);
          end if;
        end if;
      end if;
    end Set_Pump;
  end Agent;
end Motor;
```

2. 水流传感器处理对象

水流传感器是一个循环对象，所以有下列实时属性:

```
with System; use System;
with Ada.Real_Time; use Ada.Real_Time;
package Water_Flow_Sensor_Rtatt is
  Period : Time_Span := Milliseconds(1000);
  Thread_Priority : constant Priority := 9;
end Water_Flow_Sensor_Rtatt;
```
668～669

这个对象没有接口，虽然对数据记录器而言需要一个类型声明。还需要一个编用以确保包体被制作。

```
package Water_Flow_Sensor is -- 循环的
  pragma Elaborate_Body;
```

```
    type Water_Flow is (Yes, No);
                       -- 调用 Operator_Console.Alarm
                       -- 调用Data_Logger.Water_Flow_Log
                       -- 调用Motor.Request_Status
  end Water_Flow_Sensor;
```

这个包体包含两个子程序，一个用于传感器初始化（Initialize），另一个用于每个周期要执行的代码（Periodic_Code）。每次调用的时候，任务只检查水泵开时水是否在流动，水泵关时是否无水流动。如果这两个不变的要求被破坏了，就响起警报。任务Thread实现这些正确的周期时间性属性。

```
with Ada.Real_Time; use Ada.Real_Time;
with Device_Register_Types; use Device_Register_Types;
with System; use System;
with Water_Flow_Sensor_Rtatt; use Water_Flow_Sensor_Rtatt;
with Motor; use Motor;
with Operator_Console; use Operator_Console;
with Data_Logger; use Data_Logger;
with System.Storage_Elements; use System.Storage_Elements;
package body Water_Flow_Sensor is
  Flow : Water_Flow := No;
  Current_Pump_Status, Last_Pump_Status : Pump_Status := Off;

  Control_Reg_Addr : constant Address := To_Address (16#Aa14#);
  Wfcsr : Device_Register_Types.Csr;
  for Wfcsr' Address use Control_Reg_Addr;

  procedure Initialize is
  begin
    -- 允许设备操作
    Wfcsr.Device := D_Enabled;
  end Initialize;

  procedure Periodic_Code is
  begin
    Current_Pump_Status := Motor.Request_Status;
    if (Wfcsr.Operation = Set) then
      Flow := Yes;
    else
      Flow := No;
    end if;
    if Current_Pump_Status = On and
        Last_Pump_Status = On and Flow = No then
      Operator_Console.Alarm(Pump_Fault);
    elsif Current_Pump_Status = Off and
        Last_Pump_Status = Off and Flow = Yes then
      Operator_Console.Alarm(Pump_Fault);
    end if;
    Last_Pump_Status := Current_Pump_Status;
    Data_Logger.Water_Flow_Log(Flow);
  end Periodic_Code;

  task Thread is
```

```
    pragma Priority (Water_Flow_Sensor_Rtatt.Thread_Priority);
  end Thread;

  task body Thread is
    T: Time;
    Period: Time_Span := Water_Flow_Sensor_Rtatt.Period;
  begin
    T := Clock;
    Initialize;
    loop
      Periodic_Code;
      T := T + Period;
      delay until(T);
    end loop;
  end Thread;
end Water_Flow_Sensor;
```

3. HLW控制器对象

HLW控制器对象负责处理来自高、低水位检测器的中断。其目的是把这两个中断映射到对一个名为“HLW处理程序”的单个偶发对象的调用上。这是必需的，因为HRT-HOOD不允许由多于一个的启动操作去调用一个偶发对象。这些中断处理程序是一个Ada保护对象中的过程。

```
with System; use System;
with Ada.Real_Time; use Ada.Real_Time;
package Hlw_Controller_Rtatt is
  Ceiling_Priority : constant Priority := 11;
end Hlw_Controller_Rtatt;

with Hlw_Controller_Rtatt; use Hlw_Controller_Rtatt;
package Hlw_Controller is -- 保护的
  procedure Sensor_High_Ih;
  procedure Sensor_Low_Ih;
end Hlw_Controller;

with Hlw_Handler; use Hlw_Handler;
with System; use System;
with Ada.Interrupts; use Ada.Interrupts;
with Ada.Interrupts.Names; use Ada.Interrupts.Names;
-- Ada.Interrupts.Names 定义 Waterh_Interrupt 和 Waterl_Interrupt
package body Hlw_Controller is
  protected Agent is
    pragma Priority(Hlw_Controller_Rtatt.
                    Ceiling_Priority);
    procedure Sensor_High_Ih;
    pragma Attach_Handler(Sensor_High_Ih, Waterh_Interrupt);
              -- 分配中断处理程序
    procedure Sensor_Low_Ih;
    pragma Attach_Handler(Sensor_Low_Ih, Waterl_Interrupt);
              -- 分配中断处理程序
  end Agent;

  procedure Sensor_High_Ih is
```

671

```
  begin
    Agent.Sensor_High_Ih;
  end Sensor_High_Ih;

  procedure Sensor_Low_Ih is
  begin
    Agent.Sensor_Low_Ih;
  end Sensor_Low_Ih;

  protected body Agent is
    procedure Sensor_High_Ih is
    begin
      Hlw_Handler.Start(High);
    end Sensor_High_Ih;

    procedure Sensor_Low_Ih is
    begin
      Hlw_Handler.Start(Low);
    end Sensor_Low Ih;
  end Agent;
end Hlw Controller;
```

4. HLW处理程序

HLW处理程序对象处理应用对中断的响应，即请求打开或关闭水泵。它包含一个任务，这个任务经由一个保护对象入口（Wait_Start）去等待一个中断。中断是由HLW控制器对象调用Start操作发出的（一种更精巧的映射将使中断设施上的过度消耗能够被检测并处理）。

```
with System; use System;
package Hlw_Handler_Rtatt is
  Ceiling_Priority : constant Priority := 11;
  Thread_Priority : constant Priority := 6;
end Hlw_Handler_Rtatt;
```

672

```
with Hlw_Handler_Rtatt; use Hlw_Handler_Rtatt;
package Hlw_Handler is -- 偶发的
  type Water_Mark is (High, Low);
  procedure Start(Int : Water_Mark);
end Hlw_Handler;

with Device_Regiater_Types; use Device_Register_Types;
with System; use System;
with Motor; use Motor;
with Data_Logger;
with System.Storage_Elements; use System.Storage_Elements;
package body Hlw_Handler is
  Hw_Control_Reg_Addr : constant Address := To_Address(16#Aa10#);
  Lw_Control_Reg_Addr : constant Address :=  To_Address(16#Aa12#);
  Hwcsr : Device_Register_Types.Csr;
  for Hwcsr' Address use Hw_Control_Reg_Addr;
  Lwcsr : Device_Register_Types.Csr;
  for Lwcsr' Address use Lw_Control_Reg_Addr;

  procedure Sporadic_Code(Int : Water_Mark) is
  begin
```

```
    if Int = High then
      Motor.Set_Pump(On);
      Data_Logger.High_Low_Water_Log(High);
      Lwcsr.Interrupt := I_Enabled;
      Hwcsr.Interrupt := I_Disabled;
    else
      Motor.Set_Pump(Off);
      Data_Logger.High_Low_Water_Log(Low);
      Hwcsr.Interrupt := I_Enabled;
      Lwcsr.Interrupt := I_Disabled;
    end if;
  end Sporadic_Code;

  procedure Initialize is
  begin
    Hwcsr.Device := D_Enabled;
    Hwcsr.Interrupt := I_Enabled;
    Lwcsr.Device := D_Enabled;
    Lwcsr.Interrupt := I_Enabled;
  end Initialize;

  task Thread is
    pragma Priority(Hlw_Handler_Rtatt.Thread_Priority);
  end Thread;

  protected Agent is
    pragma Priority(Hlw_Handler_Rtatt.
                    Ceiling_Priority);
    -- 对Start操作
    procedure Start(Int : Water_Mark);
    entry Wait_Start(Int : out Water_Mark);
  private
    Start_Open : Boolean := False;
    W : Water_Mark;
  end Agent;

  procedure Start(Int : Water_Mark) is
  begin
    Agent.Start(Int);
  end Start;

  protected body Agent is
    procedure Start (Int : Water_Mark) is
    begin
      W := Int;
      Start_Open := True;
    end Start;

    entry Wait_Start(Int : out Water_Mark)
            when Start_Open is
    begin
      Int := W;
      Start_Open := False;
    end Wait_Start;
```

```
    end Agent;

    task body Thread is
      Int : Water_Mark;
    begin
      Initialize;
      loop
        Agent.Wait_Start(Int);
        Sporadic_Code(Int);
      end loop;
    end Thread;
  end Hlw_Handler;
```

17.5.2 环境监控

环境监控子系统的分解已在图17-7中说明。注意，check_safe子程序使水泵控制器能通过CH_4状态保护对象无阻塞地观察甲烷等级的当前状态。所有其他成分都是周期性活动。

1. CH_4状态对象

CH_4状态对象只包含指出操作水泵是否安全的数据。

673~674

```
with System; use System;
with Ada.Real_Time; use Ada.Real_Time;
package Ch4_Status_Rtatt is
  -- 对保护对象
  Ceiling_Priority : constant Priority := 10;
end Ch4_Status_Rtatt;

package Ch4_Status is -- 保护的
  type Methane_Status is (Motor_Safe, Motor_Unsafe);

  function Read return Methane_Status;
  procedure Write (Current_Status : Methane_Status);
end Ch4_Status;


with Ch4_Status_Rtatt; use Ch4_Status_Rtatt;
package body Ch4_Status is
  protected Agent is
    pragma Priority(Ch4_Status_Rtatt.
                    Ceiling_Priority);
    procedure Write (Current_Status : Methane_Status);
    function Read return Methane_Status;
  private
    Environment_Status : Methane_Status := Motor_Unsafe;
  end Agent;

  function Read return Methane_Status is
  begin
    return Agent.Read;
  end Read;

  procedure Write (Current_Status : Methane_Status) is
  begin
    Agent.Write(Current_Status);
```

```
    end Write;

    protected body Agent is
      procedure Write (Current_Status : Methane_Status) is
      begin
        Environment_Status := Current_Status;
      end Write;

      function Read return Methane_Status is
      begin
        return Environment_Status;
      end Read;
    end Agent;
  end Ch4 Status;
```

2. CH_4传感器处理对象

CH_4传感器的功能是测量环境中甲烷的含量等级。要求是它不应当上升到门限值以上。不可避免地，这个传感器将在这个门限值附近不断地发出安全和不安全的信号。为避免这种抖动，使用了门限值的下界和上界。注意，因为ADC需要花些时间产生结果，在一个周期末尾请求的转换将在下一个周期启动时使用。 675

```
with System; use System;
with Ada.Real_Time; use Ada.Real_Time;
package Ch4_Sensor_Rtatt is

  Period : Time_Span := Milliseconds(80);
  Thread_Priority : constant Priority := 10;

end Ch4_Sensor_Rtatt;


package Ch4_Sensor is -- 循环的
  pragma Elaborate_Body;
  type Ch4_Reading is new Integer range 0 . . 1023;
  Ch4_High : constant Ch4_Reading := 400;
                      -- 调用 Motor.Is_Safe
                      -- 调用Motor.Not_Safe
                      -- 调用Operator_Console.Alarm
                      -- 调用Data_Logger.Ch4_Log
end Ch4_Sensor;


with Ada.Real_Time; use Ada.Real_Time;
with System; use System;
with Ch4_Sensor_Rtatt; use Ch4_Sensor_Rtatt;
with Device_Register_Types; use Device_Register_Types;
with Operator_Console; use Operator_Console;
with Motor; use Motor;
with Data_Logger; use Data_Logger;
with Ch4_Status; use Ch4_Status;
with System.Storage_Elements; use System.Storage_Elements;
package body Ch4_Sensor is
  Ch4_Present : Ch4_Reading;
```

```
Control_Reg_Addr : constant Address := To_Address(16#Aa18#);
Data_Reg_Addr : constant Address := To_Address(16#Aa1a#);
Ch4csr : Device_Register_Types.Csr;
for Ch4csr' Address use Control_Reg_Addr;
-- 定义数据寄存器
Ch4dbr : Ch4_Reading;
for Ch4dbr' Address use Data_Reg_Addr;
Jitter_Range : constant Ch4_Reading := 40;

procedure Initialize is
begin
  -- 允许设备操作
  Ch4csr.Device := D_Enabled;
  Ch4csr.Operation := Set;
end Initialize;

procedure Periodic_Code is
begin
  if not Ch4csr.Done then
    Operator_Console.Alarm(Ch4_Device_Error);
  else
    -- 读设备寄存器中的传感器值
    Ch4_Present := Ch4dbr;
    if Ch4_Present > Ch4_High then
      if Ch4_Status.Read = Motor_Safe then
        Motor.Not_Safe;
        Ch4_Status.Write(Motor_Unsafe);
        Operator_Console.Alarm(High_Methane);
      end if;
    elsif (Ch4_Present < (Ch4_High - Jitter_Range)) and
          (Ch4_Status.Read = Motor_Unsafe) then
      Motor.Is_Safe;
      Ch4_Status.Write(Motor_Safe);
    end if;
    Data_Logger.Ch4_Log(Ch4_Present);
  end if;
  Ch4csr.Operation := Set; -- 为下一次循环启动转换
end Periodic_Code;

task Thread is
  pragma Priority(Ch4_Sensor_Rtatt.
                  Thread_Priority);
end Thread;

task body Thread is
  T: Time;
  Period : Time_Span := Ch4_Sensor_Rtatt.Period;
begin
  T:= Clock + Period;
  Initialize;
  loop
    delay until(T);
    Periodic_Code;
```

```
      T := T + Period;
    end loop;
  end Thread;
end Ch4_Sensor;
```

17.5.3 气流传感器处理对象

气流传感器是另一个周期对象，它只监控矿井中的空气流动。

```
with System; use System;
with Ada.Real_Time; use Ada.Real_Time;
package Air_Flow_Sensor_Rtatt is
  Period : Time_Span := Milliseconds(100);
  Thread_Priority : constant Priority := 7;
end Air_Flow_Sensor_Rtatt;


package Air_Flow_Sensor is -- 循环的
  pragma Elaborate_Body;
  type Air_Plow_Status is (Air_Flow, No_Air_Flow);
                        -- 调用 Data_Logger.Air_Flow_Log
                        -- 调用Operator_Console.Alarm
end Air_Flow_Sensor;

with Device_Register_Types; use Device_Register_Types;
with System; use System;
with Ada.Real_Time; use Ada.Real_Time;
with Air_Flow_Sensor_Rtatt; use Air_Flow_Sensor_Rtatt;
with Operator_Console; use Operator_Console;
with Data_Logger; use Data_Logger;
with System.Storage_Elements; use System.Storage_Elements;
package body Air_Flow_Sensor is
  Air_Flow_Reading : Boolean;

  Control_Reg_Addr : constant Address := To_Address(16#Aa120#);
  Afcsr : Device_Register_Types.Csr;
  for Afcsr' Address use Control_Reg_Addr;

  task Thread is
    pragma Priority(Air_Flow_Sensor_Rtatt.
                    Initial_Thread_Priority);
  end Thread;

  procedure Initialize is
  begin
    -- 允许设备操作
    Afcsr.Device := D_Enabled;
  end Initialize;

  procedure Periodic_Code is
  begin
    -- 读水流指示的设备寄存器
    -- (操作位置为 1);
    Air_Flow_Reading := Afcsr.Operation = Set;
    if not Air_Flow_Reading then
```

676
~
677

```
        Operator_Console.Alarm(No_Air_Flow);
        Data_Logger.Air_Flow_Log(No_Air_Flow);
      else
        Data_Logger.Air_Flow_Log(Air_Flow);
      end if;
    end Periodic_Code;

    task body Thread is
      T: Time;
      Period: Time_Span := Air_Flow_Sensor_Rtatt.Period;
    begin
      T:= Clock;
      Initialize;
      loop
        delay until(T);
        Periodic_Code;
        T := T + Period;
      end loop;
    end Thread;
  end Air_Flow_Sensor;
```

17.5.4 CO传感器处理对象

CO传感器在其实现方面也很简单。

```
with System; use System;
with Ada.Real_Time; use Ada.Real_Time;
package Co_Sensor_Rtatt is

  Period : Time_Span := Milliseconds(100);
  Thread_Priority : constant Priority := 8;
end Co_Sensor_Rtatt;


package Co_Sensor is  -- 循环的
  pragma Elaborate_Body;
  type Co_Reading is new Integer range 0 .. 1023;
  CO_High : constant Co_Reading := 600;
                     -- 调用 Data_Logger.Co_log
                     -- 调用Operator_Console.Alarm
end Co_Sensor;


with Ada.Real_Time; use Ada.Real_Time;
with System; use System;
with Device_Register_Types; use Device_Register_Types;
with Co_Sensor_Rtatt; use Co_Sensor_Rtatt;
with Operator_Console; use Operator_Console;
with Data_Logger; use Data_Logger;
with System.Storage_Elements; use System.Storage_Elements;
package body Co_Sensor is
  Co_Present : Co_Reading;

  Control_Reg_Addr : constant Address := To_Address(16#Aa1c#);
```

```
    Data_Reg_Addr : constant Address := To_Address(16#Aa1e#);
    Cocsr : Device_Register_Types.Csr;
    for Cocsr'Address use Control_Reg_Addr;
    -- 定义数据寄存器
    Codbr : Co_Reading;
    for Codbr'Address use Data_Reg_Addr;

    procedure Initialize is
    begin
      -- 允许设备操作
      Cocsr.Device := D_Enabled;
      Cocsr.Operation := Set; -- 启动转换
    end Initialize;
    procedure Periodic_Code is
    begin
      if not Cocsr.Done then
        Operator_Console.Alarm(Co_Device_Error);
      else
        -- 读设备寄存器中的传感器值
        Co_Present := Codbr;
        if Co_Present > Co_High then
          Operator_Console.Alarm(High_Co);
        end if;
        Data_Logger.Co_Log(Co_Present);
      end if;
      Cocsr.Operation := Set; -- 启动转换
    end Periodic_Code;

    task Thread is
      pragma Priority(Co_Sensor_Rtatt.Thread_Priority);
    end Thread;

    task body Thread is
      T : Time;
      Period : Time_Span := Co_Sensor_Rtatt.Period;
    begin
      T := Clock + Period;
      Initialize;
      loop
        delay until(T);
        Periodic_Code;
        T := T + Period;
      end loop;
    end Thread;
  end Co_Sensor;
```

678
~
679

17.5.5 数据记录器

只给出数据记录对象的接口。

```
with Co_Sensor; use Co_Sensor;
with Ch4_Sensor; use Ch4_Sensor;
with Air_Flow_Sensor; use Air_Flow_Sensor;
with Hlw_Handler; use Hlw_Handler;
```

```
with Water_Flow_Sensor; use Water_Flow_Sensor;
with Motor; use Motor;
package Data_Logger is -- 主动的
  procedure Co_Log(Reading : Co_Reading);
  procedure Ch4_Log(Reading : Ch4_Reading);
  procedure Air_Flow_Log(Reading : Air_Flow_Status);
  procedure High_Low_Water_Log(Mark : Water_Mark);
  procedure Water_Flow_Log(Reading : Water_Flow);
  procedure Motor_Log(State : Motor_State_Changes);
end Data_Logger;
```
680

17.5.6 操作员控制台

只给出“操作员控制台”对象的接口。

```
package Operator_Console is -- 主动的
  type Alarm_Reason is (High_Methane, High_Co, No_Air_Flow,
                Ch4_Device_Error, Co_Device_Error,
                 Pump_Fault, Unknown_Error);
  procedure Alarm(Reason: Alarm_Reason;
                 Name : String := "Unknown";
                 Details : String:= "");

  -- 调用Pump_Controller中的 Request_Status
  -- 调用Pump_Controller中的 Set_Pump
end Operator_Console;
```

17.6 容错和分布

第5章说明过可以导致嵌入式系统失效的故障的四种来源:

1）不充分的规格说明。

2）在软件部件中的设计错误引入的故障。

3）嵌入式系统中一个或多个处理器部件的失效引入的故障。

4）在支持通信子系统中的短暂或持续干扰引入的故障。

本书关注后三种。现在依次按同本案例研究的关系予以讨论。Ada的软件容错方法是使用异常处理作为构造出错恢复的框架。

17.6.1 设计错误

因为本案例研究必须要简明，所以软件设计错误的容错范围是很小的。在这个例子中，HRT-HOOD设计方法学以及Ada数据抽象设施已经被用于尝试在设计和实现阶段防止故障进入系统。在实际应用中，后面会接着有一个全面测试阶段以排除已经被引入的故障。还可以使用仿真。

程序中驻留的任何设计错误将会引起预料不到的错误出现。虽然向后出错恢复或N版本程序设计对于这类错误的恢复工作是理想的，这个例子中的设计多样性却只有很小的范围。如
681 果这个例子假设所有非预期错误会导致引发异常，每个任务或操作可以用一个“when others”异常处理程序加以保护。例如，水流线程可以被改成：如果有一个非预期的错误出现，就通知操作员。

```
task Thread;

task body Thread is
  T: Time;
  Period : Time_Span := Water_Flow_Sensor_Rtatt.Period;
  use Ada.Exceptions;
begin
  T:= Clock;
  Initialize;
  loop
    Periodic_Code;
    T := T + Period;
    delay until(T);
  end loop;
exception
  when E: others =>
    Operator_Console.Alarm(Unknown_Error, Exception_Name(E),
                           Exception_Information(E));
    Motor.Not_Safe; -- Pump_Controller.Not_Safe
    Ch4_Status.Write(Motor_Unsafe);
    -- 在终止前尝试并关掉马达
  end Thread;
end Water_Flow_Sensor;
```

虽然矿井水泛滥是严重的，但是应用的需求说火灾更危险，所以出错处理总是试图确保水泵被关闭（故障保护）。

应当注意，操纵水泵和甲烷状态的所有交互都应当用原子动作的形式。上面给出的代码使得能用另一个任务去确定马达处于不安全状态，即使甲烷状态可能指示说水泵操作是安全的（见练习17.2）。

17.6.2 处理器和通信失效

通常，如果矿井控制系统在一个单处理器计算机上实现并且处理器的任何部分发生失效，那么整个系统会处于危险之中。所以，必须使用某种形式的硬件冗余或分布。在矿井中看到的控制系统自然是分布的。图17-4中说明的顶层分解显示四个部件显然能在不同的进程上执行（见练习17.4）。

在第14章注意到Ada不定义部分失效程序的失效语义。然而，当它不能同远程划分联系时，会由底层实现引发一个异常`Communication_Error`。

已经假设所有短暂的通信失效会由底层分布式系统的实现屏蔽掉。如果有一个较持久的失效发生，那么应该由实现引发一个能由应用处理的合适异常。例如，对`Is_Safe`的远程调用产生了一个异常，水泵就应当禁止操作。 682

17.6.3 其他硬件失效

上面假设只有处理器和通信子系统会失效。显然，同样会有传感器的失效（或是由于退化，或是由于损坏）。在本章介绍的例子中，没有试图去提高传感器的可靠性，因为本书只仅仅触及硬件冗余技术。一种方法是复制每个传感器，并且由不同的任务控制每一个复制品。这些任务必须进行通信以比较结果。这些结果不可避免地会有轻微的不同，所以会需要某种形式的匹配算法。

小结

本案例研究已经包括了本书中讨论的某些问题。令人遗憾的是，一个单独的、相对小的应用不能运用已经讲到的所有重要概念。尤其是显然不可能在这个背景中致力于大小和复杂性的问题。

不过，希望这个案例研究有助于巩固读者对许多问题的理解，例如:

- 自顶向下设计和分解
- 并发性和Ada的进程间通信模型
- 向前出错恢复技术和容错设计
- 周期和偶发进程
- 优先级分配和调度分析
- 分布式程序设计

相关阅读材料

本章给出的案例研究曾经由下列参考文献中的其他几位作者讨论过:

Burns, A. and Lister, A. M. (1991) A Framework for Building Dependable Systems. *Computer Journal*, 34(2), 173–181.

683 Joseph, M. (ed.) (1996). *Real-Time Systems: Specification, Verification and Analysis*. Englewood Cliffs, NJ: Prentice Hall.

Shrivastava, S. K., Mancini, L. and Randell, B. (1987) *On The Duality of Fault Tolerant Structures*. Berlin: Springer-Verlag, pp. 19–37.

Sloman, M. and Kramer, J. (1987) *Distributed Systems and Computer Networks*. Englewood Cliffs, NJ: Prentice Hall.

练习

17.1 如果矿井中渗水的速度和水泵抽水的速度差不多，就可能产生许多次高水位中断。这会影响软件的行为吗?

17.2 在这个矿井控制系统设计中所给出的任务交互里面，确认哪些交互必须是原子动作。怎么将这个解决方案修改以支持这些动作?

17.3 本章中给出的所有周期性任务都有类似的结构。它们能被表示成一个单独的类属任务（封装在一个包中）的实例或参数化任务的实例吗?

17.4 修改本章中给出的解决方案，使之能在分布式环境中执行。假设图17-4中给出的每个顶层任务是主动划分。

17.5 练习17.4的答案的时间特性可被分析到什么程度?

17.6 数据记录器可以确定已发生事件的实际次序吗? 如果不能，怎么修改代码使之给出一个有效的全局排序? 分布式实现的隐含意义是什么?

17.7 在矿井控制系统的这个分析中，时钟以100ms（或10ms）运行的后果是什么?

17.8 在矿井控制任务集合上进行敏感性分析。依次考虑每个任务，它们的计算时间必须增加多少，才能使该任务集合成为不可调度的。用原始C值的百分比来表示这个值。

684 17.9 如果CO传感器和气流传感器的时限都是50，那么系统会是不可调度的。怎样使两个传感器有同样的周期以得到一个可调度系统?

第18章 结　　论

实时系统的显著特征是其正确性不仅仅是程序执行的逻辑结果的函数，而且是产生这些结果的时间的函数。这个特征使实时系统的研究非常不同于其他计算领域。许多实时系统的重要性还赋予实践人员独特的责任。随着越来越多的计算机被嵌入到工程应用中，人类的风险、生态灾难或经济灾难就更大了。这些风险来自于不能去证明（或至少是可信地论证）所有时间性的和功能性的约束会在所有情况下都满足。

实时系统可以以多种方式分类。第一种是按应用能够容忍系统响应缓慢的程度。那些有一些灵活性的叫做*软*实时系统；那些具有时间严格性的称为*硬*实时系统。三小时的时限可能是硬的，但容易达到；而三微秒（硬或软）的时限给开发者提出了相当大的难度。时限或响应时间非常短的系统经常称为*实*实时系统。

非实时系统几乎可以无限地等待处理器和其他系统资源。只要系统具有*活性*，它就适当地执行。实时系统不是这种情形。因为时间是受限的，而处理器又不是无限快的，一个实时程序必须被看作是在具有有限资源的系统上执行。所以，就必须在来自同一程序的不同部分的竞争请求之间调度这些资源的使用。这远不是一件微小的琐事。

典型的现代实时系统的其他特征是:

- 它们经常在地理上是分布的
- 它们可能包含非常大而复杂的软件部件
- 它们必须同并发的现实世界实体交互
- 它们可能包含一些受制于价格、尺寸或重量约束的处理要素

从大多数实时应用的本性得出：存在着对于高可靠性的严格要求。这一点可以被正式表述为对可依赖性和安全性的需要。在计算机系统及其当前环境之间存在着几乎是共生的关系。没有另一个，任何一个都不能工作，像电操纵式飞机一样。为了给出高可靠性，需要容错的硬件和软件。需要容许功能缺失和错过时限（甚至是对硬实时系统）。 685

时间性需求、受限资源、并发环境实体和高可靠性需求（连同分布式处理）的组合向系统工程师提出了特有的问题。实时系统工程现在被认识到是一个不同的学科。它有自己的知识本体和理论基础。从对大型实时系统科学的理解看，将出现下列的研究内容:

- 可以捕获时间性和容错需求的规格说明技术
- 以时间性需求为核心的和能够处理提供容错和分布方法的设计方法
- 能用于实现这些设计的编程语言和操作系统

本书已经讨论了实时系统的特征和需求、容错技术、并发性模型、与时间有关的语言特性、资源控制和低级编程技术。通贯全书，主要讲解并发实时语言提供的设施，尤其是Ada、Java（及其实时扩展）、C（增补以POSIX实时和线程扩展）和occam2。表18-1总结了这些语言提供的设施。

Ada依然是高完整性系统和有硬实时约束的系统最合适的语言。实时Java现正以其表达能力同Ada竞争。然而，还要再看一看是否能将其实现得足够高效，足以用于有效的实际部署。

确实，可能是这种情形：实时Java作为语言是合适的，但实时Java作为虚拟机却不合适。所以，或者是程序需要编译而不是解释，或者是需要硬件辅助的虚拟机器。可能实时Java比Ada做得好的是支持更加动态的软实时系统。实时Java正是在这种应用领域找到它的初步应用，特别是在可移植性很重要的应用中。

C语言，扩展以实时操作系统（遵照或不遵照POSIX）将继续被用于那些有资源约束的小型嵌入式系统。对比之下，对occam2的使用将继续减少。在学术上，它有一些有趣的性质，但它缺乏对大型编程的支持，这宣告它只不过是一个深奥的语言。

在过去五年里有一个趋势——在可预见的将来还会继续——即在一个应用中使用一种以上的语言。这是需要使用遗留代码的结果，也因为认识到对应用中所有的类使用同一语言是不适宜的。例如，一个实时应用有一个重要的用户接口部件，可能是用Ada和Java混合写的。一个“智能”的实时系统可能需要基于规则的部件，而对这种部件最适宜的语言可能是Prolog。

表18-1 Ada、实时Java、C/POSIX和occam2提供设施的总结

	Ada	实时Java	C/POSIX	occam2
对大型编程的支持	包	包	文件	无
	类属	接口		
对并发编程的支持	任务	线程	进程	进程
	会合	同步方法和语句	线程	会合
	保护类型		互斥锁	
对在分布式环境中	RPC	远程方法	待定义	分布式进程
执行的支持	划分	分布式对象		
	分布式对象			
容错编程的设施	异常	异常	信号	无
	ATC	ATC		
		事件		
实时设施	时钟和延迟	时钟和定时器	时钟和定时器	时钟和延迟
调度设施	固有优先级模型	固有优先级模型	固有优先级模型	受限优先级模型
	动态优先级	动态优先级	动态优先级	静态优先级
设备处理的模型	共享存储器	有限的共享存储器	无定义	消息传递
执行环境	受限的任务性ATAC	无子集	剖面	传输机

为了给出实时系统的一个实际例子，本书描述了一个案例研究。这必然是一个相对小的系统。然而实时计算面对的许多挑战是只在大且复杂的应用中呈现的。为了给出在不远的将
686～687 来要部署的这类系统的一个印象，考虑一下国际空间站。它的计算机的基本功能是对使命和生命的支持。其他活动包括飞行控制（特别是轨道传输载体）、外部监控、实验的控制和协调以及使命数据库的管理。机载软件的特别重要的方面是它呈现给飞行人员的界面。

空间站的机载执行环境有下列有关特性：

- 它大而复杂（即有大量形形色色的要计算机处理的活动）
- 它必须不停地执行
- 它有很长的操作寿命（可能超过30年）
- 它将经历进化式的软件更改（不用停下来）
- 它必须有高可依赖的执行
- 它的有些部件有硬的或软的实时时限

• 分布式系统将包括异质处理器

为了迎接这类应用的挑战，实时系统科学必须发展自己。还有许多研究课题要探索。甚至连认识当前状态——它们已经成为本书的关注焦点——都难以实践。在对“下一代”实时系统的探索中，Stankovic确认了若干研究问题（Stankovic，1988）。虽然自1988年以来已经取得了很大进展，下列研究课题对于这门学科的发展依然是极其重要的：

• 规格说明和验证技术，要能够处理有大量交互部件的实时系统的需要。
• 从设计过程一开始就考虑时间性质的设计方法学。
• 有表达与时间有关行为的构造的编程语言（尤其与分布计算有关的）。
• 能够处理复杂进程结构和资源约束、各种粒度的时间性需求和概率性保证的调度算法。
• 实时排队模型。
• 为处理容错资源使用而设计的运行时支持或操作系统功能。
• 预测复杂现代处理器上软件最坏情况和平均执行时间的工具支持。
• 最坏情况和平均情况性能度量的集成。
• 高效处理那些需要在Internet上及时交付的消息的通信体系结构和协议。
• 容错和动态重配置的体系结构支持。
• 人工智能（例如机器学习）部件的集成支持。
• 对原子动作、恢复块、会话和群通信协议的编程语言和操作系统支持。
• 对更改管理有显式支持的编程语言（即在不停止的系统上进行软件升级的能力）。
• 支持有严格时间性约束的实时应用“编写一次、到处运行”的实时虚拟机。
• 使应用能进行配合以响应环境改变的实时反应式（自修改）体系结构。

希望本书的一些读者能对这些研究课题有所贡献。 688~689

附录　实时Java规格说明

本附录包含本书中讨论的实时Java类的规格说明。这些规格说明与文献（Bollella等，2000）一书中的一致。但是，应该注意到实时Java是在不断演化的，而且，在写本书时，还没有用实现测试它。因此读者应该参考实时Java专家组的网站（http: //www.rtj.org/）以了解这个项目的当前状态。

这个规格说明按字母顺序给出类头。注意，虽然Thread和ThreadGroup类不是实时Java规格说明的一部分，但仍然在这个附录中给出。

A.1　AbsoluteTime

```
package javax.realtime;
public class AbsoluteTime extends HighResolutionTime
{
  // constructors
  public AbsoluteTime();
  public AbsoluteTime(AbsoluteTime time);
  public AbsoluteTime(java.util.Date date);
  public AbsoluteTime(long millis, int nanos);

  // methods
  public AbsoluteTime absolute(Clock clock, AbsoluteTime destination);
  public AbsoluteTime add(long millis, int nanos);
  public AbsoluteTime add(long millis, int nanos,
                          AbsoluteTime destination);
  public final AbsoluteTime add(RelativeTime time);
  public AbsoluteTime add(RelativeTime time,
                          AbsoluteTime destination);
  public java.util.Date getDate();
  public void set(java.util.Date date);
  public final RelativeTime subtract(AbsoluteTime time);
  public RelativeTime subtract(AbsoluteTime time,
                               RelativeTime destination);
  public final AbsoluteTime subtract(RelativeTime time);
  public AbsoluteTime subtract(RelativeTime time,
                               AbsoluteTime destination);
  public java.lang.String toString();
}
```

691

A.2　AperiodicParameters

```
package javax.realtime;
public class AperiodicParameters extends ReleaseParameters
{
  // constructors
  public AperiodicParameters(RelativeTime cost, RelativeTime deadline,
```

```
        AsyncEventHandler overrunHandler,
        AsyncEventHandler missHandler);
}
```

A.3 AsyncEvent

```
package javax.realtime;
public class AsyncEvent
{
  // constructors
  public AsyncEvent();

  // methods
  public synchronized void addHandler(AsyncEventHandler handler);
  public void bindTo(java.lang.String happening);
  public ReleaseParameters createReleaseParameters();
  public synchronized void fire();
  public boolean handledBy(AsyncEventHandler target);
  public synchronized void removeHandler(AsyncEventHandler handler);
  public void setHandler(AsyncEventHandler handler);
}
```

A.4 AsyncEventHandler

692

```
package javax.realtime;
public abstract class AsyncEventHandler implements Schedulable
{
  //constructors
  public AsyncEventHandler();
  public AsyncEventHandler(boolean nonheap);
  public AsyncEventHandler(SchedulingParameters scheduling,
         ReleaseParameters release, MemoryParameters memory,
         MemoryArea area, ProcessingGroupParameters group);
  public AsyncEventHandler(SchedulingParameters scheduling,
         ReleaseParameters release, MemoryParameters memory,
         MemoryArea area, ProcessingGroupParameters group,
         boolean nonheap);

  //methods
  public void addToFeasibility();
  protected final synchronized int getAndClearPendingFireCount();
  protected synchronized int getAndDecrementPendingFireCount();
  protected synchronized int getAndIncrementPendingFireCount();
  public MemoryArea getMemoryArea();
  public MemoryParameters getMemoryParameters();
  public ProcessingGroupParameters getProcessingGroupParameters();
  public ReleaseParameters getReleaseParameters();
  public Scheduler getScheduler();
  public SchedulingParameters getSchedulingParameters();
  public abstract void handleAsyncEvent();
  public void removeFromFeasibility();
  public final void run();
```

```
  public void setMemoryParameters(MemoryParameters memory);
  public void setProcessingGroupParameters(
              ProcessingGroupParameters parameters);
  public void setReleaseParameters(ReleaseParameters parameters);
  public void setScheduler(Scheduler scheduler);
  public void setSchedulingParameters(
              SchedulingParameters parameters);
}
```

A.5 AsynchronouslyInterruptedException

```
package javax.realtime;
public class AsynchronouslyInterruptedException
       extends java.lang.InterruptedException
{
  // constructors
  public  AsynchronouslyInterruptedException();

  // methods
  public synchronized boolean disable();
  public boolean doInterruptible (Interruptible logic);
  public synchronized boolean enable();
  public synchronized boolean fire();
  public static AsynchronouslyInterruptedException getGeneric();
  public boolean happened (boolean propagate);
  public boolean isEnabled();
  public void propagate();
}
```

693

A.6 BoundAsyncEventHandler

```
package javax.realtime;
public abstract class BoundAsyncEventHandler
                extends AsyncEventHandler
{
  // constructors
  public BoundAsyncEventHandler();
  public BoundAsyncEventHandler(SchedulingParameters scheduling,
          ReleaseParameters release, MemoryParameters memory,
          MemoryArea area, ProcessingGroupParameters group,
          boolean nonheap);
}
```

A.7 Clock

```
package javax.realtime;
public abstract class Clock
{
  // constrcutors
```

```
  public Clock();

  // methods
  public static Clock getRealtimeClock();
  public abstract RelativeTime getResolution();
  public AbsoluteTime getTime();
  public abstract void getTime(AbsoluteTime time);
  public abstract void setResolution(RelativeTime resolution);
}
```

A.8 HighResolutionTime

694

```
package javax.realtime;
public abstract class HighResolutionTime
       implements java.lang.Comparable
{
  // methods
  public abstract AbsoluteTime absolute(Clock clock,
                                        AbsoluteTime destination);
  public int compareTo(HighResolutionTime time);
  public int compareTo(java.lang.Object object);
  public boolean equals(HighResolutionTime time);
  public boolean equals(java.lang.Object object);
  public final long getMilliseconds();
  public final int getNanoseconds();
  public int hashCode();
  public void set(HighResolutionTime time);
  public void set(long millis);
  public void set(long millis, int nanos);
}
```

A.9 ImmortalMemory

```
package javax.realtime;
public final class ImmortalMemory extends MemoryArea
{
  // methods
  public static ImmortalMemory instance();
}
```

A.10 ImportanceParameters

```
package javax.realtime;
public class ImportanceParameters extends PriorityParameters
{
  // constructors
  public ImportanceParameters(int priority, int importance);
```

```
  // methods
  public int getImportance();
  public void setImportance(int importance);
  public java.lang.String toString();
}
```

A.11 Interruptible

```
package javax.realtime;
public interface Interruptible
{
  public void interruptAction(
              AsynchronouslyInterruptedException exception);
  public void run(AsynchronouslyInterruptedException exception)
         throws AsynchronouslyInterruptedException;
}
```

695

A.12 LTMemory

```
public class LTMemory extends ScopedMemory
{
  //constructors
  public LTMemory(long initialSizeInBytes, long maxSizeInBytes);
}
```

A.13 MemoryArea

```
public abstract class MemoryArea
{
  // constructors
  protected MemoryArea(long sizeInBytes);

  // methods
  public void enter(java.lang.Runnable logic);
  public static MemoryArea getMemoryArea(java.lang.Object object);
  public long memoryConsumed();
  public long memoryRemaining();
  public synchronized java.lang.Object newArray(
         java.lang.class type, int number)
         throws IllegalAccessException, InstantiationException,
         OutOfMemoryError;
  public synchronized java.lang.Object newInstance(
         java.lang.class type)
         throws IllegalAccessException, InstantiationException,
         OutOfMemoryError;
  public long size();
}
```

A.14 MemoryParameters

```
public class MemoryParameters
{
  // fields
  public static final long NO_MAX;

  // constructors
  public MemoryParameters(long maxMemoryArea, long maxImmortal)
         throws IllegalArgumentException;

  public MemoryParameters(long maxMemoryArea, long maxImmortal,
                          long allocationRate)
696      throws IllegalArgumentException;

  // methods
  public long getAllocationRate();
  public long getMaxImmortal();
  public long getMaxMemoryArea();
  public void setAllocationRate(long rate);
  public boolean setMaxImmortal(long maximum);
  public boolean setMaxMemoryArea(long maximum);
}
```

A.15 MonitorControl

```
package javax.realtime;
public abstract class MonitorControl
{
  // constructors
  public MonitorControl();

  // methods
  public static void setMonitorControl(MonitorControl policy);
  public static void setMonitorControl(java.lang.Object monitor,
                                       MonitorControl policy);
}
```

A.16 NoHeapRealtimeThread

```
package javax.realtime;
public class NoHeapRealtimeThread extends RealtimeThread
{
  //constructors
  public NoHeapRealtimeThread(SchedulingParameters scheduling,
         MemoryArea area) throws IllegalArgumentException;

  public NoHeapRealtimeThread(SchedulingParameters scheduling,
         ReleaseParameters release, MemoryArea area)
         throws IllegalArgumentException;

  public NoHeapRealtimeThread(SchedulingParameters scheduling,
         ReleaseParameters release, MemoryParameters memory,
```

```
        MemoryArea area, ProcessingGroupParameters group,
        java.lang.Runnable logic)
            throws IllegalArgumentException;
}
```

697

A.17 OneShotTimer

```
package javax.realtime;
public class OneShotTimer extends Timer
{
  // constructors
  public OneShotTimer(HighResolutionTime time,
                      AsyncEventHandler handler);
  public OneShotTimer(HighResolutionTime start, Clock clock,
                      AsyncEventHandler handler);
}
```

A.18 PeriodicParameters

```
package javax.realtime;
public class PeriodicParameters extends ReleaseParameters
{
  // constructors
  public PeriodicParameters(HighResolutionTime start,
         RelativeTime period, RelativeTime cost,
         RelativeTime deadline, AsyncEventHandler overrunHandler,
         AsyncEventHandler missHandler);

  // methods
  public RelativeTime getPeriod();
  public HighResolutionTime getStart();
  public void setPeriod(RelativeTime period);
  public void setStart(HighResolutionTime start);
}
```

A.19 PeriodicTimer

```
package javax.realtime;
public class PeriodicTimer extends Timer
{
  // constructors
  public PeriodicTimer(HighResolutionTime start,
                       RelativeTime interval,
                       AsyncEventHandler handler);
  public PeriodicTimer(HighResolutionTime start,
                       RelativeTime interval,
                       Clock clock, AsyncEventHandler handler);

  // methods
  public ReleaseParameters createReleaseParameters();
```

```
  public void fire();
698  public AbsoluteTime getFireTime();
  public RelativeTime getInterval();
  public void setInterval(RelativeTime interval);
}
```

A.20 POSIXSignalHandler

```
package javax.realtime;
public final class POSIXSignalHandler
{
  //fields
  public static final int SIGABRT;
  public static final int SIGALRM;
  public static final int SIGBUS;
  public static final int SIGCANCEL;
  public static final int SIGCHLD;
  public static final int SIGCLD;
  public static final int SIGCONT;
  public static final int SIGEMPT;
  public static final int SIGFPE;
  public static final int SIGFREEZE;
  public static final int SIGHUP;
  public static final int SIGILL;
  public static final int SIGINT;
  public static final int SIGIO;
  public static final int SIGIOT;
  public static final int SIGKILL;
  public static final int SIGLOST;
  public static final int SIGLWP;
  public static final int SIGPIPE;
  public static final int SIGPOLL;
  public static final int SIGPROF;
  public static final int SIGPWF;
  public static final int SIGQUIT;
  public static final int SIGSEGV;
  public static final int SIGSTOP;
  public static final int SIGSYS;
  public static final int SIGTERM;
  public static final int SIGTHAW;
  public static final int SIGTRAP;
  public static final int SIGTSTP;
  public static final int SIGTTIN;
  public static final int SIGTTOU;
  public static final int SIGURG;
  public static final int SIGUSR1;
  public static final int SIGUSR2;
  public static final int SIGVTALRM;
  public static final int SIGWINCH;
  public static final int SIGXCPU;
699  public static final int SIGXFSZ;
  // methods
  public static synchronized void addHandler(int signal,
        AsyncEventHandler handler);
  public static synchronized void removeHandler(int signal
        AsyncEventHandler handler);
  public static synchronized void setHandler(int signal,
```

```
        AsyncEventHandler handler);
}
```

A.21 PriorityCeilingEmulation

```
package javax.realtime;
public class PriorityCeilingEmulation extends MonitorControl
{
  // constructors
  public PriorityCeilingEmulation(int ceiling);

  // methods
  public int getDefaultCeiling();
}
```

A.22 PriorityInheritance

```
package javax.realtime;
public class PriorityInheritance extends MonitorControl
{
  // constructors
  public PriorityInheritance();

  // methods
  public static PriorityInheritance instance();
}
```

A.23 PriorityParameters

```
package javax.realtime;
public class PriorityParameters extends SchedulingParameters
{
  // constructors
  public PriorityParameters(int priority);

  // methods
  public int getPriority();
  public void setPriority(int priority) throws IllegalArgumentException;
  public java.lang.String toString();
}
```

700

A.24 PriorityScheduler

```
package javax.realtime;
public class PriorityScheduler extends Scheduler
{
  // constructors
```

```
  public PriorityScheduler() ;

  // methods
  protected void addToFeasibility(Schedulable schedulable);
  boolean changeIfFeasible(Schedulable schedulable,
          ReleaseParameters release, MemoryParameters memory);
  public void fireSchedulable(Schedulable schedulable);
  public int getMaxPriority();
  public static int getMaxPriority(java.lang.Thread thread);
  public int getMinPriority();
  public static int getMinPriority(java.lang.Thread thread);
  public int getNormPriority();
  public static int getNormPriority(java.lang.Thread thread);
  public java.lang.String getPolicyName();
  public static PriorityScheduler instance();
  public boolean isFeasible();
  protected void removeFromFeasibility(Schedulable schedulable);
}
```

A.25 ProcessingGroupParameters

701

```
package javax.realtime;
public class ProcessingGroupParameters
{
  // constructors
  public ProcessingGroupParameters(HighResolutionTime start,
         RelativeTime period, RelativeTime cost,
         RelativeTime deadline, AsyncEventHandler overrunHandler,
         AsyncEventHandler missHandler);

  // methods
  public RelativeTime getCost();
  public AsyncEventHandler getCostOverrunHandler();
  public RelativeTime getDeadline();
  public AsyncEventHandler getDeadlineMissHandler();
  public RelativeTime getPeriod();
  public HighResolutionTime getStart();
  public void setCost(RelativeTime cost);
  public void setCostOverrunHandler(AsyncEventHandler handler);
  public void setDeadline(RelativeTime deadline);
  public void setDeadlineMissHandler(AsyncEventHandler handler);
  public void setPeriod(RelativeTime period);
  public void setStart(HighResolutionTime start);
}
```

A.26 RationalTime

```
package javax.realtime;
public class RationalTime extends RelativeTime
{
  // constructors
  public RationalTime(int frequency);
  public RationalTime(int frequency, long millis, int nanos)
         throws IllegalArgumentException;
```

```
  public RationalTime(int frequency, RelativeTime interval)
         throws IllegalArgumentException;

  // methods
  public AbsoluteTime absolute(Clock clock, AbsoluteTime destination);
  public void addInterarrivalTo(AbsoluteTime destination);
  public int getFrequency();
  public RelativeTime getInterarrivalTime(RelativeTime destination);
  public void set(long millis, int nanos)
         throws IllegalArgumentException;
  public void setFrequency(int frequency)
         throws ArithmeticException;
}
```

A.27 RawMemoryAccess

```
package javax.realtime;
public class RawMemoryAccess
{
  // constructors
  protected RawMemoryAccess(long base, long size);
  protected RawMemoryAccess(RawMemoryAccess memory, long base,
                            long size);

  // methods
  public static RawMemoryAccess create(java.lang.Object type,
         long size) throws SecurityException, OffsetOutOfBoundsException,
         SizeOutOfBoundsException, UnsupportedPhysicalMemoryException;

  public static RawMemoryAccess create(java.lang.Object type,
                                       long base, long size)
         throws SecurityException, OffsetOutOfBoundsException,
         SizeOutOfBoundsException, UnsupportedPhysicalMemoryException;
```

702

```
  public byte getByte(long offset) throws SizeOutOfBoundsException,
         OffsetOutOfBoundsException;
  public void getBytes(long offset, byte[] bytes, int low,
                       int number) throws
             SizeOutOfBoundsException, OffsetOutOfBoundsException;
  public int getInt(long offset) throws SizeOutOfBoundsException,
         OffsetOutOfBoundsException;
  public void getInts(long offset, int[] ints, int low,
                      int number) throws
             SizeOutOfBoundsException, OffsetOutOfBoundsException;
  public long getLong(long offset) throws SizeOutOfBoundsException,
         OffsetOutOfBoundsException;
  public void getLongs(long offset, long[] longs, int low,
                       int number) throws
             SizeOutOfBoundsException, OffsetOutOfBoundsException;
  public long getMapped Address();
  public short getshort(long offset) throws SizeOutOfBoundsException,
         OffsetOutOfBoundsException;
  public void getshorts(long offset, short[] shorts, int low,
                        int number) throws
             SizeOutOfBoundsException, OffsetOutOfBoundsException;
  public long map();
  public long map(long base);
```

```
  public long map(long base, long size);
  public void setByte(long offset, byte value) throws
              SizeOutOfBoundsException, OffsetOutOfBoundsException;
  public void setBytes(long offset, byte[] bytes, int low,
                       int number) throws
              SizeOutOfBoundsException, OffsetOutOfBoundsException;
  public void setInt(long offset, int value) throws
              SizeOutOfBoundsException, OffsetOutOfBoundsException;
  public void setInts(long offset, int[] ints, int low,
                      int number) throws
              SizeOutOfBoundsException, OffsetOutOfBoundsException;
  public void setLong(long offset, long value) throws
              SizeOutOfBoundsException, OffsetOutOfBoundsException;
  public void setLongs(long offset, long[] longs, int low,
                       int number) throws
              SizeOutOfBoundsException, OffsetOutOfBoundsException;
  public void setShort(long offset, short value) throws
              SizeOutOfBoundsException, OffsetOutOfBoundsException;
  public void setShorts(long offset, short[] shorts, int low,
                        int number) throws
              SizeOutOfBoundsException, OffsetOutOfBoundsException;
  public void unmap();
}
```

A.28 Realtime Security

703

```
package javax.realtime;
public class RealtimeSecurity
{
  // constructors
  public RealtimeSecurity();

  // methods
  public void checkAccessPhysical() throws
         SecurityException;
  public void checkAccessPhysicalRange(long base, long size) throws
         SecurityException;
  public void checkSetFactory() throws
         SecurityException;
  public void checkSetScheduler() throws
         SecurityException;
}
```

A.29 Realtime System

```
package javax.realtime;
public class RealtimeSystem
{
  // fields
  public static final byte BIG_ENDIAN;
  public static final byte BYTE_ORDER;
  public static final byte LITTLE_ENDIAN;
```

```
    // methods
    public static GarbageCollector currentGC();
    public int getConcurrentLocksUsed();
    public int getMaximumConcurrentLocks();
    public static RealtimeSecurity getSecurityManager();
    public void setMaximumConcurrentLocks(int number);
    public void setMaximumConcurrentLocks(int number, boolean hard);
    public static voidgetSecurityManager(RealtimeSecurity manager) throws
            SecurityException;
}
```

A.30 RealtimeThread

```
package javax.realtime;
public class RealtimeThread extends java.lang.Thread
        implements Schedulable
{
    // constructors
    public RealtimeThread();
    public RealtimeThread(SchedulingParameters scheduling);
    public RealtimeThread(SchedulingParameters scheduling,
                          ReleaseParameters release);
    public RealtimeThread(SchedulingParameters scheduling,
            ReleaseParameters release, MemoryParameters memory,
            MemoryArea area, ProcessingGroupParameters group,
            java.lang.Runnable logic);

    // methods
    public void addToFeasibility();
    public static RealtimeThread currentRealtimeThread();
    public synchronized void deschedulePeriodic();
    public MemoryArea getMemoryArea();
    public MemoryParameters getMemoryParameters();
    public ProcessingGroupParameters getProcessingGroupParameters();
    public ReleaseParameters getReleaseParameters();
    public Scheduler getScheduler();
    public SchedulingParameters getSchedulingParameters();
    public synchronized void interrupt();
    public void removeFromFeasibility();
    public synchronized void schedulePeriodic();
    public void setMemoryParameters(MemoryParameters parameters);
    public void setProcessingGroupParameters(
                ProcessingGroupParameters parameters);
    public void setReleaseParameters(ReleaseParameters parameters);
    public void setScheduler(Scheduler scheduler)
            throws IllegalThreadStateException;
    public void setSchedulingParameters(
                SchedulingParameters scheduling);
    public static void sleep(Clock clock, HighResolutionTime time)
            throws InterruptedException;
    public static void sleep(HighResolutionTime time)
            throws InterruptedException;
    public boolean waitForNextPeriod()
            throws IllegalThreadStateException ;
}
```

704

A.31 RelativeTime

```
package javax.realtime;
public class RelativeTime extends HighResolutionTime
{
  // constructors
  public RelativeTime();
  public RelativeTime(long millis, int nanos);
  public RelativeTime(RelativeTime time);

  // methods
  public AbsoluteTime absolute(Clock clock,
                               AbsoluteTime destination);
  public RelativeTime add(long millis, int nanos);
  public RelativeTime add(long millis, int nanos,
                          RelativeTime destination);
  public final RelativeTime add(RelativeTime time);
705 public RelativeTime add(RelativeTime time, RelativeTime destination);
  public void addInterarrivalTo(AbsoluteTime destination);
  public RelativeTime getInterarrivalTime(RelativeTime destination);
  public final RelativeTime subtract(RelativeTime time);
  public RelativeTime subtract(RelativeTime time,
                               RelativeTime destination);
  public java.lang.String toString();
}
```

A.32 ReleaseParameters

```
package javax.realtime;
public abstract class ReleaseParameters
{
  // constructors
  protected ReleaseParameters(RelativeTime cost, RelativeTime deadline,
            AsyncEventHandler overrunHandler,
            AsyncEventHandler missHandler);

  // methods
  public RelativeTime getCost();
  public AsyncEventHandler getCostOverrunHandler();
  public RelativeTime getDeadline();
  public AsyncEventHandler getDeadlineMissHandler();
  public void setCost(RelativeTime cost);
  public void setCostOverrunHandler(AsyncEventHandler handler);
  public void setDeadline(RelativeTime deadline);
  public void setDeadlineMissHandler(AsyncEventHandler handler);
}
```

A.33 Schedulable

```
package javax.realtime;
public interface Schedulable extends java.lang.Runnable
```

```
{
  // methods
  public void addToFeasibility();
  public MemoryParameters getMemoryParameters();
  public ReleaseParameters getReleaseParameters();
  public Scheduler getScheduler();
  public SchedulingParameters getSchedulingParameters();
  public void removeFromFeasibility();
  public void setMemoryParameters(MemoryParameters memory);
  public void setReleaseParameters(ReleaseParameters release);
  public void setScheduler(Scheduler scheduler);
  public void setSchedulingParameters(SchedulingParameters scheduling);
}
```

706

A.34 Scheduler

```
package javax.realtime;
public abstract class Scheduler
{
  // constructors
  public Scheduler();

  // methods
  protected abstract void addToFeasibility(Schedulable schedulable);
  public boolean changeIfFeasible(Schedulable schedulable,
               ReleaseParameters release, MemoryParameters memory);
  public static Scheduler getDefaultScheduler();
  public abstract java.lang.String getPolicyName();
  public abstract boolean isFeasible();
  protected abstract void removeFromFeasibility(Schedulable schedulable);
  public static void setDefaultScheduler(Scheduler scheduler);
}
```

A.35 SchedulingParameters

```
package javax.realtime;
public abstract class SchedulingParameters
{
  // constructors
  public SchedulingParameters();
}
```

A.36 ScopedMemory

```
package javax.realtime;
public abstract class ScopedMemory extends MemoryArea
{
  // constructors
  public ScopedMemory(long size);

  // methods
```

```
  public void enter(java.lang.Runnable logic);
  public int getMaximumSize();
  public MemoryArea getOuterScope();
  public java.lang.Object getPortal();
  public void setPortal(java.lang.Object object);
}
```

707

A.37 ScopedPhysicalMemory

```
package javax.realtime;
public class ScopedPhysicalMemory extends ScopedMemory;
{
  // constructors
  protected ScopedPhysicalMemory(long base, long size);
  protected ScopedPhysicalMemory(ScopedPhysicalMemory memory,
                                 long base, long size);

  //methods
  public static ScopedPhysicalMemory create(java.lang.Object type,
              long base, long size) throws SecurityException,
              OffsetOutOfBoundsException, SizeOutOfBoundsException,
              UnsupportedPhysicalMemoryException;
  public static void setFactory(PhysicalMemoryFactory factory);
}
```

A.38 SporadicParameters

```
package javax.realtime;
public class SporadicParameters extends AperiodicParameters
{
  //constructors
  public SporadicParameters(RelativeTime minInterarrival,
         RelativeTime cost, RelativeTime deadline,
         AsyncEventHandler overrunHandler,
         AsyncEventHandler missHandler);

  // methods
  public RelativeTime getMinimumInterarrival();
  public void setMinimumInterarrival(RelativeTime minimum);

}
```

A.39 Thread

```
// Not in package javax.realtime;
public class Thread extends Object implements Runnable
{
  // fields
```

```
static int MAX_PRIORITY;
static int MIN_PRIORITY;
static int NORM_PRIORITY;
```

708

```
// constructors
public Thread();
public Thread(Runnable target);
public Thread(Runnable target, String name);
public Thread(String name);
public Thread(ThreadGroup group, String name);
        throws SecurityException, IllegalThreadStateException
public Thread(ThreadGroup group, Runnable target);
        throws SecurityException, IllegalThreadStateException
public Thread(ThreadGroup group, Runnable target, String name);
         throws SecurityException, IllegalThreadStateException

// methods
static int activeCount();
void checkAccess() throws SecurityException;
int countStackFrames()
    throws IllegalThreadStateException;
public static Thread currentThread();
public void destroy() throws SecurityException;
static void dumpStack();
static int enumerate(Thread[] tarray)
        throws SecurityException;
ClassLoader getContextClassLoader() throws SecurityException;
String getName();
int getPriority();
ThreadGroup getThreadGroup();.
void interrupt() throws SecurityException;
static boolean interrupted();
public final native boolean isAlive();
public final boolean isDaemon();
boolean isInterrupted();
public final void join() throws InterruptedException;
public final void join(long millis)
        throws InterruptedException;
public final void join(long millis, int nanos)
        throws InterruptedException;
public void resume()
        throws SecurityException;  // DEPRECATED
public void run();
public void setContextClassLoader(ClassLoader cl)
        throws SecurityException;
public final void setDaemon()
        throws SecurityException, IllegalThreadStateException;
public void setName(String name)
        throws SecurityException;
public void setPriority(int newPriority)
        throws class IllegalArgumentException, InterruptedException;
public static void sleep(long millis)
        throws InterruptedException;
public static void sleep(long millis, int nanos)
        throws class IllegalArgumentException, InterruptedException;
public native synchronized void start()
        throws IllegalThreadStateException;
public final void stop()
        throws SecurityException;  // DEPRECATED
public final synchronized void stop(Throwable o);
        throws SecurityException, NullPointerException;  // DEPRECATED
```

709

```
  public void suspend()
         throws SecurityException;   // DEPRECATED
  public String toString();
  public static void yield();
}
```

A.40 ThreadGroup

```
// Not in package javax.realtime;
public class ThreadGroup {
  // constructors
  public ThreadGroup(String name)
         throws SecurityException;
  public ThreadGroup(ThreadGroup parent, String name)
         throws SecurityException, NullPointerException;

  // methods
  public int activeCount();
  public int activeGroupCount();
  public boolean allowThreadSuspension(boolean b);  // DEPRECATED
  public final void checkAccess()
         throws SecurityException;
  public final void destroy()
         throws IllegalThreadStateException, SecurityException;
  public int enumerate(Thread[] list)
         throws SecurityException;
  public int enumerate(Thread[] list, boolean recurse)
         throws SecurityException;
  public int enumerate(ThreadGroup[] list)
         throws SecurityException;
  public int enumerate(ThreadGroup[] list, boolean recurse)
  public int getMaxPriority()
  public String getName()
  public final ThreadGroup getParent()
         throws SecurityException;
  public void interrupt()
         throws SecurityException;
  public final boolean isDaemon();
  public synchronized boolean isDestroyed();
  public void list()
  public final boolean parentOf(ThreadGroup g);
  public void resume()
         throws SecurityException;  // DEPRECATED
  public final void setDaemon(boolean daemon);
               throws SecurityException;
  public void setMaxPriority(int pri)
         throws SecurityException;
  public final void stop()
               throws SecurityException;  // DEPRECATED
  public void suspend()
710      throws SecurityException;  // DEPRECATED
  public String toString();
  public void uncaughtException(Thread t, Throwable e)
}
```

A.41 Timed

```
package javax.realtime;
public class Timed extends AsynchronouslyInterruptedException
                   implements java.io.Serializable
{
  // constructors
  public Timed(HighResolutionTime time) throws
              IllegalArgumentException;
  // methods
  public boolean doInterruptible(Interruptible logic);
  public void resetTime(HighResolutionTime time);
}
```

A.42 Timer

```
package javax.realtime;
public abstract class Timer extends AsyncEvent
{
  // constructors
  protected Timer(HighResolutionTime time, Clock clock,
                  AsyncEventHandler handler);
  // methods
  public ReleaseParameters createReleaseParameters();
  public void disable();
  public void enable();
  public Clock getClock();
  public AbsoluteTime getFireTime();
  public void reschedule(HighResolutionTime time);
  public void start();
}
```

A.43 VTMemory

```
package javax.realtime;
public class VTMemory extends ScopedMemory
{
  // constructors
  VTMemory(int initial, int maximum);
}
```

711

参考文献

AEEC (1991). Design guidance for integrated mondular avionics, *ARINC 651 (Draft 9)*, AEEC.

aJile Systems (2000). aJ-100 datasheet, *http://www.ajile.com/ aJ-100_datasheet.htm*, aJile Systems.

Allworth, S. and Zobel, R. (1987). *Introduction to Real-Time Software Design*, MacMillan.

Ammann, P. and Knight, J. (1988). Data diversity: An approach to software fault tolerance, *IEEE Transactions on Computers* **C-37**(4): 418–425.

Anderson, T. and Lee, P. (1990). *Fault Tolerance Principles and Practice:*, 2nd edn, Prentice Hall International.

Andrews, G. and Olsson, R. (1993). *The SR Programming Language - Concurrency in Practice*, Benjamin/Cummings.

ARINC AEE Committee (1999). Avionics application software standard interface.

Audsley, N. and Burns, A. (1998). On fixed priority scheduling, offsets and co-prime task periods, *Information Processing Letters, Elsevier* **67**: 65–69,.

Audsley, N. et al. (1993a). Applying new scheduling theory to static priority preemptive scheduling, *Software Engineering Journal* **8**(5): 284–292.

Audsley, N., Tindell, K. and Burns, A. (1993b). The end of the line for static cyclic scheduling?, *Proceedings of the 5th Euromicro Workshop on Real-Time Systems*, IEEE Computer Society Press, Oulu, Finland, pp. 36–41.

Avizienis, A. and Ball, D. (1987). On the achievement of a highly dependable and fault-tolerant air traffic control system, *Computer* **20**(2): 84–90.

Avizienis, A., Lyu, M. and Schutz, W. (1988). Multi-version software developmnt: A UCLA/Honeywell joint project for fault-tolerant flight control systems, *CSD-880034*, Department of Computer Science, University of California, Los Angeles.

Bach, M. (1986). *The Design of the UNIX Operating System*, Prentice Hall.

Baker, T. P. (1991). Stack-based scheduling of realtime processes, *Real Time Systems*.

Barnes, J. (1976). *RTL/2 Design and Philosophy*, Heyden International Topics in Science.

Barrett, P., Hilborne, A., Verissimo, P., Rodrigues, L., Bond, P., Seaton, D. and Speirs, N. (1990). The Delta-4 extra performance architecture(XPA), *Digest of Papers, The Twentieth Annual International Symposium on Fault-Tolerant Computing*, Newcastle, pp. 481–488.

Barringer, H. and Kuiper, R. (1983). Towards the hierarchical, temporal logic, specification of concurrent systems, *Proceedings of the STL/SERC workshop on the analysis of concurrent systems*, Spinger-Verlag.

Bate, I. and Burns, A. (1997). Timing analysis of fixed priority real-time systems with offsets, *9th Euromicro Workshop on Real-Time Systems*, pp. 153–160.

Ben-Ari, M. (1982). *Principles of Concurrent Programming*, Prentice Hall.

Bennett, P. (1994). Software development for the Channel Tunnel: A summary, *High Integrity Systems* **1**(2): 213–220.

Bernat, G. and Burns, A. (1999). New results on fixed priority aperiodic servers, *Proceedings of the 20th IEEE Real-Time Systems Symposium*, pp. 68–78.

Berry, G. (1989). Real time programming: Special purpose or general purpose languages, *in* G. Ritter (ed.), *Proceedings of Information Processing '89*, Elsevier Science Publishers.

Bloom, T. (1979). Evaluating synchronisation mechanisms, *Proceedings of the Seventh ACM Symposium on Operating System Principles*, Pacific Grove, pp. 24–32.

Bois, P. D. (1995). Semantic definition of the Albert II language, *Technical Report RP-95-007*, Computer Science Dept., University of Namur, Namur, Belgium.

Bollella, G., Brosgol, B., Dibble, P., Furr, S., Gosling, J., Hardin, D. and Turnbull, M. (2000). *The Real-Time Specification for Java*, Addison-Wesley.

Booch, G. (1986). *Software Engineering with Ada*, 2nd edn, The Benjamin/Cummings Publishing Company, Inc.

Booch, G., Rumbaugh, J. and Jacobson, I. (1999). *The Unified Modeling Language User Guide*, Addoson-Wesley.

Boussinot, F. and de Simone, R. (1991). The Esterel langauge, *Proceedings of IEEE*, pp. 1293–1304.

Brauer, W. (1980). Net theory and applications, *Lecture Notes in Computer Science*, Springer-Verlag.

Brilliant, S., Knight, J. and Leveson, N. (1987). The consistent comparison problem in N-version software, *ACM Software Engineering Notes* **12**(1): 29–34.

Brilliant, S., Knight, J. and Leveson, N. (1990). Analysis of faults in an N-version software experiment, *IEEE Transactions on Software Engineering* **16**(2): 238–47.

Brinch-Hansen, P. (1973). *Operating System Principles*, Prentice Hall, New Jersey.

Brinch-Hansen, P. (1981). Edison: A multiprocessor language, *Software-Practice and Experience* **11**(4): 325–361.

Bull, G. and Lewis, A. (1983). Real-Time BASIC, *Software - Practice and Experience* **13**(11): 1075–1092.

Burns, A. (1983). Enhanced input/output on Pascal, *SIGPLAN Notices*.

Burns, A. (1988). *Programming in occam 2*, Addison Wesley.

Burns, A. (1999). The Ravenscar Profile, *ACM Ada Letters* **XIX**(4): 49–52.

Burns, A. and Lister, A. M. (1991). A framework for building dependable systems, *Computer Journal* **34**(2): 173–181.

Burns, A. and Wellings, A. J. (1994). HRT-HOOD: A design method for hard real-time Ada, *Real-Time Systems* **6**(1): 73–114.

Burns, A. and Wellings, A. J. (1995). *Hard Real-Time HOOD: A Structured Design Method for Hard Real-Time Ada Systems*, Elsevier.

Burns, A. and Wellings, A. J. (1998). *Concurrency in Ada*, second edn, Cambridge University Press.

Burns, A., Lister, A. and Wellings, A. J. (1987). A review of Ada tasking, *Lecture Notes in Computer Science, Volume 262*, Springer-Verlag.

Burns, A., Prasad, D., Bondavalli, A., Giandomenico, F. D., Ramamritham, K., Stankovic, J. and Stringini, L. (2000). The meaning and role of value in scheduling flexible real-time systems, *Journal of Systems Architecture* **46**: 305–325.

Busquets, J. and Wellings, A. J. (1996). Adding instruction cache effect to schedulability analysis of preemptive real-time systems, *Proceedings of RTAS96.*

Cairns, P. (1999). Enumeration types in Java, *Software-Practice and Experience* **29**(3): 291–297.

Calvez, J. (1993). *Embedded Real-Time Systems: A Specification and Design Methodology*, Wiley.

Campbell, R. and Randell, B. (1986). Error recovery in asynchronous systems, *IEEE Transactions on Software Engineering* **1**(8): 811–826.

CCITT (1980). CCITT high level language (CHILL) recommendation Z.200.

Chen, L. and Avizienis, A. (1978). N-version programming: A fault-tolerance approach to reliability of software operation, *Digest of Papers, The Eighth Annual International Conference on Fault-Tolerant Computing*, Toulouse, France, pp. 3–9.

Cherry, G. (1986). *Pamela Designers Handbook*, Thought Tools Incorporated.

Chetto, H. and Chetto, M. (1989). Some results of the earliest deadline scheduling algorithm, *IEEE Transactions on Software Engineering* **15**(10): 1261–1269.

Cmelik, R., Gehani, N. and Roome, W. (1988). Fault Tolerant Concurrent C: A tool for writing fault tolerant distributed programs, *The Eighteenth Annual International Symposium on Fault-Tolerant Computing Digest of Papers.*

Coleman, D. et al. (1994). *Object-Oriented Development: The Fusion Method*, Prentice Hall.

Conway, M. E. (1963). Design of a separable transition-diagram compiler, *Communications of the ACM* **6**(7): 396–408.

Cooling, J. (1991). *Software Design for Real-Time Systems*, Chapman and Hall.

Cornhill, D., Sha, L., Lehoczky, J., Rajkumar, R. and Tokuda, H. (1987). Limitations of Ada for real-time scheduling, *Proceedings of the International Workshop on Real Time Ada Issues, ACM Ada Letters*, pp. 33–39.

Davis, R. and Wellings, A. J. (1995). Dual priority scheduling, *IEEE Proceedings of Real-Time Systems Symposium.*

de la Puente, J., Alonso, A. and Alvarez, A. (1996). Mapping HRT-HOOD designs to Ada 95 hierarchical libraries, *Ada-Europe'96 Conference, Springer-Verlag.*

Dijkstra, E. (1965). Solution of a problem in concurrent program control, *Communications of the ACM* **8**(9): 569–.

Dijkstra, E. (1968a). Cooperating sequential processes, *in* F. Genuys (ed.), *Programming Languages*, Academic Press, London.

Dijkstra, E. (1968b). The structure of THE multiprogramming system, *Communications of the ACM* **11**(5): 341–.

Dijkstra, E. (1975). Guarded commands, nondeterminacy, and formal derivation of programs, *CACM* **18**(8): 453–457.

Dix, A., M.D.Harrison and C.Runciman (1987). Interactive models and the principled design of interactive systems, *Proc ESEC'87* pp. 127–135.

Douglass, B. (1999). *Doing Hard Time: Developing Real-Time Systems with UML, Objects Frameworks and Patterns*, Addison-Wesley.

Dubois, E., Bois, P. D. and Zeippen, J. (1995). A formal requirements engineering method for real-time, concurrent, and distributed systems, *Proc. of the Real-Time*

Systems Conference RTS'95, Paris (France).

Eckhardt, D., Caglayan, A., Knight, J., Lee, J., McAllister, D., Vouk, M. and Kelly, J. (1991). An experimental evaluation of software redundancy as a strategy for improving reliability, *IEEE Transactions on Software Engineering* **17**(7): 692–702.

Ericsson (1986). SDS80/A standardised computing system for Ada, *Document number L/BD, number = 5553*, Ericsson.

Evangelist, M., Francez, M. and Katz, S. (1989). Multiparty interactions for interprocess communication and synchronization, *IEEE Transactions on Software Engineering* **15**(11): 1417–1426.

Garman, J. (1981). The bug heard round the world, *Software Engineering Notes* **6**(3): 3–10.

Goel, A. and Bastini, F. (1985). Software reliability, *IEEE Transactions on Software Engineering* **SE-11**(12): 1409–1410.

Goldsack, S. and Finkelstein, A. (1991). Requirements engineering for real-time systems, *Software Engineering Journal* **6**(3): 101–15.

Gomaa, H. (1994). Software design methods for the design of large-scale real-time systems, *Journal of Systems and Software* **25**(2): 127–146.

Graham, I. (1994). *Object Oriented Methods*, 2nd edn, Addison-Wesley.

Graham, R. (1969). Bounds on multiprocessing timing anomalies, *SIAM J. Appl. Math.* **12**(2): 416–429.

Graham, R. et al. (1979). Optimization and approximation in deterministic sequencing and scheduling: A survey, *Ann. Discrete Math.* **5**: 287–326.

Gregory, S. and Knight, J. (1985). A new linguistic approach to backward error recovery, *The Fifteenth Annual International Symposium on Fault-Tolerant Computing Digest of Papers*, pp. 404–409.

Halbwachs, N., Caspi, P., Raymond, P. and Pilaud, D. (1991). The synchronous dataflow programming language Lustre, *Proceedings of IEEE*, pp. 1305–1320.

Harbour, M. G., Rivas, M. A. and Gutierrez, J. P. (1998). Implementing and using execution time clocks in Ada hard real-time applications, *Reliable Software Technologies - Ada Europe 98, Lecture Notes in Computer Science* **1411**: 90–101.

Hatton, L. (1997). N-version design versus one good version, *IEEE Software* **14**(6): 71–76.

Hecht, H. and Hecht, M. (1986a). Fault-tolerant software, *in* D. Pradhan (ed.), *Fault-Tolerant Computing Theory and Techniques Volume II*, Prentice Hall, pp. 659–685.

Hecht, H. and Hecht, M. (1986b). Software reliability in the systems context, *IEEE Transactions on Software Engineering* **SE-12**(1): 51–58.

Heitz, M. (1995). HOOD Reference Manual: Release 4.0, *HRM4-9/22/95*, HOOD Technical Group.

Herlihy, M. and Liskov, B. (1982). A value transmission method for abstract data types, *ACM TOPLAS* **4**(4): 527–551.

Hoare, C. (1974). Monitors - an operating system structuring concept, *CACM* **17**(10): 549–557.

Hoare, C. (1978). Communicating sequential processes, *CACM* **21**(8): 666–677.

Hoare, C. (1985). *Communicating Sequential Processes*, Prentice Hall International.

Hoogeboom, B. and Halang, W. (1992). The concept of time in the specification of real-time systems, *in* K. Kavi (ed.), *Real-Time Systems: Abstractions, Languages and Design Methodologies*, IEEE Computer Society Press, pp. 19–38.

Horning, J. J., Lauer, H. C., Melliar-Smith, P. M. and Randell, B. (1974). A program structure for error detection and recovery, *in* E. Gelenbe and C. Kaiser (eds), *Lecture Notes in Computer Science 16*, Springer-Verlag, pp. 171 – 187.

Hull, M., O'Donoghue, P. and Hagan, B. (1991). Development methods for real-time systems, *Computer Journal* **34**(2): 164–72.

IEEE (1995). Portable operating sysrem interface: Amendment 2: Threads extension [C language], *IEEE/1003.1c*, IEEE.

ISO/IEC JTC1/SC21/WG8 (1992). Open system interconnection - remote procedure call specification part 1: Model, *CD, number = 11578-1.2*, ISO/IEC JTC1/SC21/WG8.

Iverson, K. (1962). *A Programming Language*, Wiley New York.

J Consortium (2000). Real-time core extensions for the Java platform, *Revision 1.0.10*, J Consortium.

Jackson, K. (1986). Mascot 3 and Ada, *Software Engineering Journal* **1**(3): 121–135.

Jackson, M. (1975). *Principles of Program Design*, Academic Press Inc.

Jacobson, I., Booch, G. and Rumbaugh, J. (1999). *The Unified Software Develoment Process*, Addison-Wesley.

Jahanian, F. and Mok, A. (1986). Safety analysis of timing properties in real-time systems, *IEEE Transactions on Software Engineering* **12**(9): 890–904.

Jones, C. (1986). *Systematic Software Development Using VDM*, Prentice Hall.

Jones, M. (1997). What happened on Mars?, *http://www.cs.cmu.edu/afs/cs.cmu.edu/project/art-6/www/mars.html*, Accessed 28/4/2000.

Joseph, M. and Pandya, P. (1986). Finding response times in a real-time system, *BCS Computer Journal* **29**(5): 390–395.

Joseph, M. (ed.) (1996). *Real-Time Systems: Specification, Verification and Analysis*, Prentice Hall.

Kemeny, J. et al. (1979). *Report of the President's Commission on the Accident at Three Mile Island*, Government Printing Office, Washington.

Kernighan, B. W. and Ritchie, D. M. (1978). *The C Programming Language*, Prentice Hall Inc., New Jersey.

Kim, T., Chang, N., Kim, N. and Shin, H. (1999). Scheduling garbage collector for embedded real-time systems, *Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'99), ACM SIGPLAN Notices*, pp. 55–64.

Kligerman, E. and Stoyenko, A. (1986). Real-Time Euclid: A language for reliable real-time systems, *IEEE Transactions on Software Engineering* **SE-12**(9): 941–949.

Knight, J., Leveson, N. and St.Jean, L. (1985). A large scale experiment in N-version programming, *Digest of Papers, The Fifteenth Annual International Symposium on Fault-Tolerant Computing*, Nichigan, USA, pp. 135–139.

Kopetz, H. (1997). *Real-Time Systems: Design Principles for Distributed Embedded*

Applications, Kluwer International Series in Engineering and Computer Science.

Kramer, J., Magee, J., Sloman, M. and Lister, A. (1983). CONIC: an integrated approach to distributed computer control systems, *IEE Proceedings (Part E)*, pp. 1–10.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system, *CACM* **21**(7): 558–565.

Lamport, L. (1983). Specifying concurrent program modules, *Transactions on Programming Languages and Systems* **5**(2): 190–222.

Lamport, L., Shostak, R. and Pease, M. (1982). The Byzantine generals problem, *Transactions on Programming Languages and Systems* **4**(3): 382–401.

Lampson, B. and Redell, D. (1980). Experience with processes and monitors in Mesa, *CACM* **23**(2): 105–117.

Laprie, J. (1985). Dependable computing and fault tolerance: Concepts and terminology, *Digest of Papers, The Fifteenth Annual International Symposium on Fault-Tolerant Computing*, Michigan, USA, pp. 2–11.

Laprie, J. (1995). Dependable - its attributes, impairments and means, *in* B. Randell et al. (eds), *Predictable Dependable Computig Systems*, Springer.

Lauber, R. J. (1989). Forecasting real-time behaviour during software design using a CASE environment, *The Journal of Real-Time Systems* **1**: 61–76.

Lauer, H. and Needham, R. (1978). On the duality of operating system structure, *Proceedings of the Second International Symposium on Operating System Principles*, IRIA, pp. 3–19.

Lauer, H. and Satterwaite, E. (1979). The impact of Mesa on system design, *Proceedings of the 4th International Conference on Software Engineering*, IEEE, pp. 174–182.

Lawton, J. and France, N. (1988). The transformation of JSD specification into Ada, *Ada User* **8**(1): 29–44.

le Guernic, P., Gautier, T., le Borgne, M. and le Maire, C. (1991). Programming real applications in Signal, *Proceedings of IEEE*, pp. 1321–1313.

Lea, D. (1997). *Concurrent Programming in Java*, Addison Wesley.

Lee, I. and Gehlot, V. (1985). Language constructs for distributed real-time programming, *Proceedings of the Real-Time Systems Symposium*, IEEE Computer Society Press, pp. 57–56.

Lee, P. (1983). Exception handling in C programs, *Software - Practice and Experience* **13**(5): 389–406.

Lee, P., Ghani, N. and Heron, K. (1980). A recovery cache for the PDP-11, *IEEE Transactions on Computers* **C-29**(6): 546–549.

Lehman, M. and Belady, L. (1985). The characteristics of large systems, *Program Evolution - Processes of Software Change, APIC Studies in Data processing No. 27*, Academic Press, pp. 289–329.

Lehoczky, J. P., Sha, L. and Strosnider, J. K. (1987). Enhanced aperiodic responsiveness in a hard real-time environment, *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 261–270.

Leung, J. and Whitehead, J. (1982). On the complexity of fixed-priority scheduling of periodic, real-time tasks, *Performance Evaluation (Netherlands)* **2**(4): 237–250.

Leveson, N. (1986). Software safety: Why, what and how, *ACM Computing Surveys* **18**(2): 125–163.

Leveson, N. and Harvey, P. (1983). Analyzing software safety, *IEEE Transactions on Software Engineering* **SE-9**(5): 569–579.

Lim, T. F., Pardyak, P. and Bershad, B. N. (1999). A memory-efficient real-time non-copying garbage collector, *Proceedings of the International Symposium on Memory Management (ISMM-98), ACM SIGPLAN Notices*, pp. 118–129.

Liskov, B. and Snyder, A. (1979). Exception handling in CLU, *IEEE Transaction on Software Engineering* **SE-5**(6): 546–558.

Liskov, B., Herlihy, M. and Gilbert, L. (1986). Limitations of remote procedure call and static process structure for distributed computing, *Proceedings of Thirteenth Annual ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, pp. 150–159.

Littlewood, B. and Strigini, L. (1993). Validation of ultrahigh dependability for software-based systems, *Communications of the ACM* **36**(11): 69–80.

Liu, C. and Layland, J. (1973). Scheduling algorithms for multiprogramming in a hard real-time environment, *JACM* **20**(1): 46–61.

Locke, C. (1992). Software architecture for hard real-time applications: cyclic executives vs. fixed priority executives, *Real-Time Systems* **4**(1): 37–53.

Lomet, D. (1977). Process structuring, synchronisation and recovery using atomic actions, *Proceedings ACM Conference Language Design for Reliable Software SIGPLAN*, pp. 128–137.

Lytz, R. (1995). Software metrics for the Boeing 777: a case study, *Software Quality Journal* **4**(1): 1–14.

Martin, D. (1982). Dissimilar software in high integrity applications in flight controls, *AGARD Symposium on Software for Avionics*, p. 36:1.

Matsuoka, S. and Yonezawa, A. (1993). Analysis of inheritance anomaly in object-oriented concurrent programming languages, *Research Directions in Concurrent Object-Oriented Programming*, MIT Press, pp. 107–150.

May, D. and Shepherd, R. (1984). Occam and the transputer, *Proc IFIP Workshop on Hardware Supported Implemention of Concurrent Languages in Distributed Systems*, Univ. of Bristol, UK.

McDermid, J. (1989). Assurance in high integrity software, *in* C. Sennett (ed.), *High Integrity Software*, Pitman.

Meyer, B. (1992). *Eiffel: The Language*, Prentice Hall.

Mok, A. and Dertouzos, M. (1978). Multiprocessor scheduling in a hard real-time environment, *Proc. 7th Texas Conf. Comput. Syst.*

Monarchi, D. and Puhr, G. (1992). A research typology for object-oriented analysis and design, *Communications of the ACM* **35**(9): 35–47.

Mullery, G. (1979). CORE - A method for controlled requirement specification, *IEEE Computer Society Press*.

Object Management Group (1999). Realtime CORBA joint revised submission, *OMG Document orbos/99-02-12*, Object Management Group.

Owicki, S. and Lamport, L. (1982). Proving liveness properties of concurrent programs, *Transactions on Programming Languages and Systems* **4**(4): 455–495.

Palencia Gutierrez, J. and Gonzalez Harbour, M. (1998). Schedulability analysis for tasks with static and dynamic offsets, *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pp. 26–39.

Palencia Gutierrez, J. and Gonzalez Harbour, M. (1999). Exploiting precedence relations in the schedulability analysis of distributed real-time systems, *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pp. 328–339.

Pease, M., Shostak, R. and Lamport, L. (1980). Reaching agreement in the presence of faults, *Journal of the ACM* **27**(2): 228–234.

Peterson, G. (1981). Myths about the mutual exclusion problem, *Information Processing Letters* **12**(3): 115–16.

Purtilo, J. and Jalote, P. (1991). An environment for developing fault-tolerant software, *IEEE Transactions on Software Engineering* **17**(2): 153–9.

Ramamritham, K. and Stankovic, J. (1984). Dynamic task scheduling in hard real-time distributed systems, *IEEE Software* **1**(3): 65–75.

Randell, B. (1975). System structure for software fault tolerance, *IEEE Transactions on Software Engineering* **SE-1**(2): 220–232.

Randell, B. (1982). *The Origins of Digital Computers: selected papers*, Springer Verlag.

Randell, B., Laprie, J.-C., Kopetz, H. and Littlewood(Eds.), B. (1995). *Predictably Dependable Computing Systems*, Springer.

Randell, B., Lee, P. and Treleaven, P. (1978). Reliability issues in computing system design, *ACM Computing Surveys* **10**(2): 123–165.

Real, J. and Wellings, A. J. (1999a). The ceiling protocol in multi-moded real-time systems, *Reliable Software Technologies - Ada-Europe 99, Lecture Notes in Computer Science* **1622**: 275–286.

Real, J. and Wellings, A. J. (1999b). Implementing mode changes and shared resources in Ada, *Proceedings 11th Euromicro Conference on Real-Time Systems*, IEEE Society Press, York, England, pp. 86–93.

Reason, J. (1979). Actions not as planned: The price of automation, *in Aspects of Consciousness ed G. Underwood and R. Stevens*.

Reeves, G. (1997). What really happened on mars? – authoritative account, *http://www.research.microsoft.com/~mbj/mars_pathfinder/authoritative_account.html*, Accessed 28/4/2000.

Rogers, P. and Wellings, A. J. (2000). An incremental recovery cache supporting software fault tolerance mechanisms, *Computer Systems Science and Engineering* **15**(1): 33–48.

Roos, J. (1991). Designing a real-time coprocessor for Ada tasking, *IEEE Design Test of Computers* **8**(1): 67–79.

Roscoe, A. (1985). Denotational semantics for occam, *Lecture Notes in Computer Science*, Vol. 197, Springer-Verlag.

Rouse, W. (1981). Human-computer interaction in the control of dynamic systems, *Computer Surveys* **13**(1): 71–99.

Rubira-Calsavara, C. and Stroud, R. (1994). Forward and backward error recovery in C++, *Object-Oriented Systems* **1**(1): 61–85.

Runner, D. and Warshawsky, E. (1988). Synthesizing Ada's ideal machine mate, *VLSI Systems Design* **9**(10): 30–39.

Saltzer, J., Reed, D. and Clark, D. (1984). End-to-end arguments in system design, *ACM Transactions on Computer Systems* **2**(4): 277–288.

Schneider, F. (1984). Byzantine generals in action: Implementing fail-stop processors, *Transactions on Computer Systems* **2**(2): 145–154.

Selic, B., Moore, A., Bjorkander, M., Gerhardt, M. and Watson, B. (2000). Response to the OMG RFP for schedulability, performance and time, *http://cgi.omg.org/cgi-bin/doc?ad/00-08-04*, Accessed 18/8/2000.

Sha, L., Rajkumar, R. and Lehoczky, J. P. (1990). Priority inheritance protocols: An approach to real-time synchronisation, *IEEE Transactions on Computers* **39**(9): 1175–1185.

Shrivastava, S. (1978). Sequential Pascal with recovery blocks, *Software - Practice and Experience* **8**(2): 177–186.

Shrivastava, S. (1979). Concurrent pascal with backward error recovery: Language features and examples, *Software - Practice and Experience* **9**(12): 1001–1020.

Shrivastava, S. and Banatre, J. (1978). Reliable resource allocation between unreliable processes, *IEEE Transactions on Software Engineering* **SE-4**(3): 230–240.

Shrivastava, S., Mancini, L. and Randell, B. (1987). On the duality of fault tolerant structures, *Lecture Notes in Computer Science*, Vol. 309, Springer-Verlag, pp. 19 – 37.

Siebert, F. (1999). Guaranteeing non-disruptiveness and real-time deadlines in an incremental garbage collector, *Proceedings of the International Symposium on Memory Management (ISMM-98), ACM SIGPLAN Notices*, pp. 118–129.

Silberschatz, A. and Galvin, P. (1994). *Operating System Concepts*, fourth edn, Addison-Wesley.

Sloman, M. and Kramer, J. (1987). *Distributed Systems and Computer Networks*, Prentice Hall.

Sloman, M., Magee, J. and Kramer, J. (1984). Building flexible distributed systems in Conic, *in* D. A. Duce (ed.), *Distributed Computing Systems Programme*, Peter Peregrinus Ltd., pp. 86–105.

Smedema, C., Medema, P. and Boasson, M. (1983). *The Programming Languages Pascal, Modula, CHILL, Ada*, Prentice Hall.

Spivey, M. (1989). *The Z Notation A Reference Manual*, Prentice Hall (see also http://spivey.oriel.ox.ac.uk/~mike/zrm/index.html) Accessed 19/10/2000.

Spuri, M. and Buttazzo, G. (1994). Efficient aperiodic service under earliest deadline scheduling, *Proceedings IEEE Real-Time Systems Symposium*, pp. 2–11.

Stankovic, J. (1988). Misconceptions about real-time computing: A serious problem for next generation systems, *IEEE Computer* **21**(10): 10–19.

Stankovic, J., Ramamritham, K. and Cheng, S. (1985). Evaluation of a flexible task scheduling algorithm for distributed hard real-time systems, *IEEE Trans. Computers* **34**(12): 1130–1143.

Sun Microsystems (2000). picoJava II datasheet, *http://www.sun.com/microelectronics/databook/datasheets.html*, Sun Microsystems.

Tanenbaum, A. S. (1998). *Computer Networks*, Prentice Hall, Englewood Cliffs, N.J.

Teichrow, D. and Hershey, E. (1977). PSL/PSA: A computer-aided technique for structured documentation and analysis of information processing systems, *IEEE Transactions on Software Engineering* **SE-3**(1): 41–48.

Tindell, K. and Clark, J. (1994). Holistic schedulability analysis for distributed hard real-time systems, *Euromicro Journal (Special Issue on Parallel Embedded Real-Time Systems)* **40**: 117–134.

Tindell, K., Burns, A. and Wellings, A. J. (1994). An extendible approach for analysing fixed priority hard real-time tasks, *Real-Time Systems* **6**(2): 133–151.

Tindell, K., Burns, A. and Wellings, A. J. (1995). Calculating controller area network (CAN) message response times, *Control Engineering Practice* **3**(8): 1163–1169.

US National Institute of Standards and Technology (1999). Requirements for real-time extensions for the Java platform, *Nistir*, US National Institute of Standards and Technology.

Venners, B. (1999). *Inside the Java 1.2 Virtual Machine*, Osborne McGraw-Hill.

Walker, W. and Cragon, H. (1995). Interrupt processing in concurrent processors, *Computer* **28**(6): 36–46.

Wegner, P. (1987). Dimensions of object-based language design, *ACM SIGPLAN Notices* **22**(12): 168–182.

Wellings, A. J. and Burns, A. (1996). Programming replicated systems in Ada 95, *Computer Journal* **39 5**: 362–373.

Wellings, A. J. and Burns, A. (1997). Implementing atomic actions in Ada 95, *IEEE Transactions on Software Engineering* **23**(2): 107–123.

Wellings, A. J., Johnson, B., Sanden, B., Kienzle, J., Wolf, T. and Michell, S. (2000). Integrating object-oriented programming and protected types in Ada 95, *ACM TOPLAS*.

Wellings, A. J., Keeffe, D. and Tomlinson, G. (1984). A problem with Ada and resource allocation, *Ada Letters* **3**(4): 112–123.

Werum, W. and Windauer, H. (1985). *Introduction to PEARL Process and Experiment Automation Realtime Language*, Friedr. Vieweg Sohn.

Whitaker, W. (1978). The U.S. department of defense common high order language effort, *ACM SIGPLAN Notices* **13**(2): 19–29.

Wirth, N. (1977a). Design and implementation of modula, *Software-Practice and Experience* **7**: 67–84.

Wirth, N. (1977b). Modula: a language for modular multiprogramming, *Software-Practice and Experience* **7**(1): 3–35.

Wirth, N. (1983). *Programming in Modula-2*, second edn, Springer-Verlag.

Wirth, N. (1988). The programming language Oberon, *Software - Practice and Experience* **18**(7): 671–690.

Wolf, T. (1998). *Replication on Non-Deterministic Objects*, Ph.D. thesis, Swiss Federal

Institute of Technology in Lausanne, Switzerland.

Xerox Corporation (1985). Mesa language manual verson 5.0.

X/Open Company Ltd. (1996). *X/Open Guide: Architecture Neutral Distribution Format*, X/Open Company Ltd., UK.

Young, S. (1982). *Real Time Languages: Design and Development*, Ellis Horwood Publishers, Chichester.

Yuh-Jzer, J. and Smolka, S. A. (1996). A comprehensive study of the complexity of multiparty interaction, *Journal of the ACM* **43**(1): 75–115.

索　　引

索引中的页码为英文原书页码，与书中边栏的页码一致。

B

C

D

E

F

G

H

I

J

K

L

M

S

T