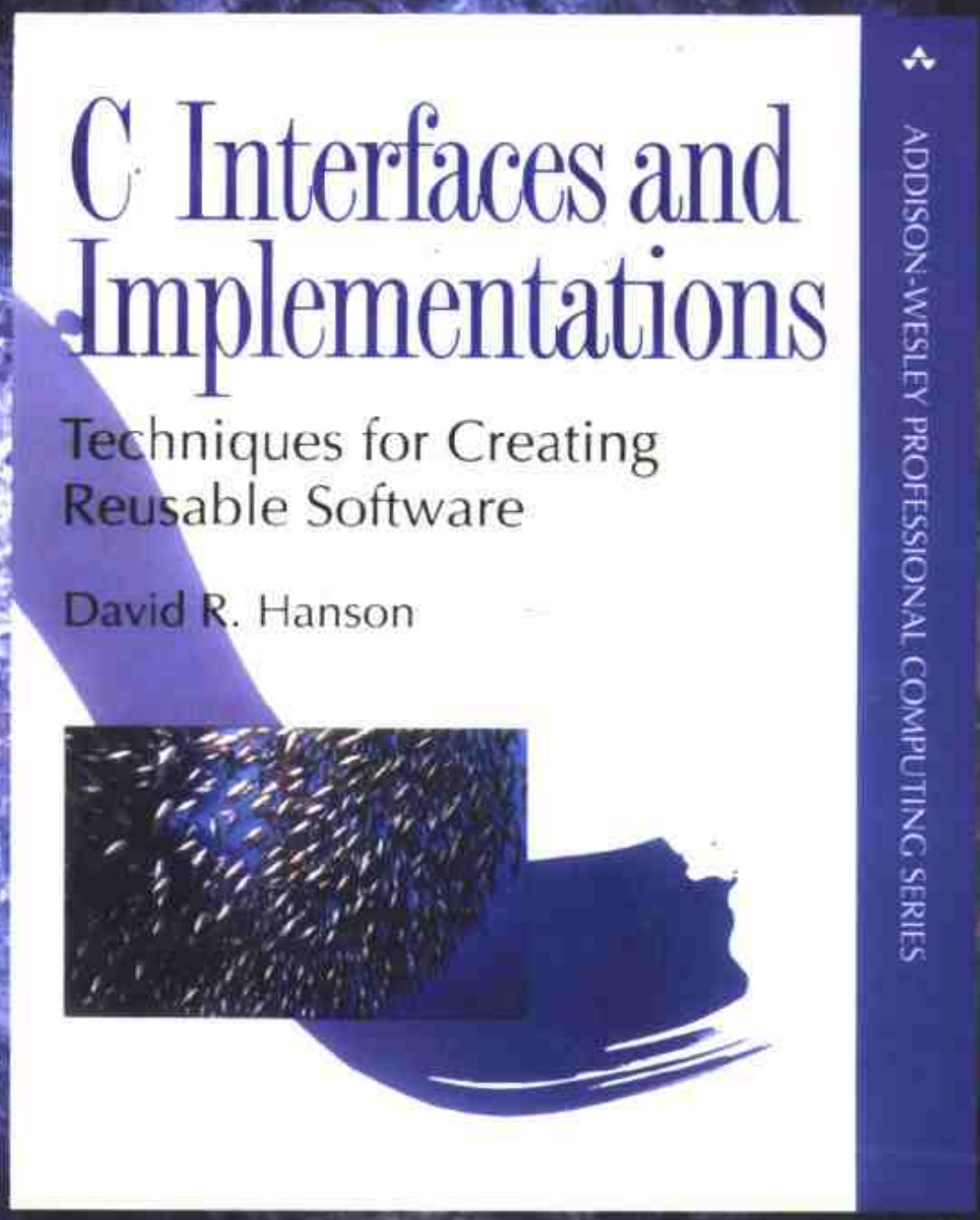
计 算 机 科 学 丛 书

C语言接口与实现

创建可重用软件的技术

（美） David R. Hanson 著　　傅蓉　周鹏　张昆琪　权威　译



C Interfaces and Implementations

Techniques for Creating Reusable Software

机械工业出版社
China Machine Press

“关于如何设计、实现和有效使用库函数的指南少之又少（如果说还有的话）。这本力作填补了这个空白。它可以作为下一代软件的工具书，所有的C语言程序员都应该阅读。”

—— W. Richard Stevens

“我向每位专业C语言程序员推荐这本书。C语言程序员们忽视书中所描述的各种技术已经太长时间了。”

—— Norman Ramsey，贝尔实验室研究员

每一位程序员和软件项目经理必须掌握创建可重用软件模块的技术：可重用软件模块是构建大规模、可靠应用的基石。与当前某些面向对象语言不同，C语言为创建可重用应用程序接口 (Application Programming Interface, API) 提供的语言和功能支持非常少。尽管大多数C语言程序员在自己所编写的每一个应用程序中都使用API和实现API的库，但只有相当少的程序员可以创建和发布新的、可广泛使用的API。本书阐述了如何用一种与语言无关的方法将接口的设计与实现独立开来，从而形成一种基于接口的设计途径来创建可重用的API。书中提供大量实例具体说明这种方法。作者详细描述了24个接口和它们的实现细节，有助于读者对这种设计方法的透彻理解。

本书具有如下特色

- 简洁明了的接口描述，为对接口设计感兴趣的程序员提供了一个参考手册
- 每一章接口的代码实现分析将帮助读者修改、扩充一个接口，或者设计相关接口
- 深入探讨了“算法工程”：阐述如何将数据结构以及相关算法打包到可重用模块中
- 24个API和8个实例程序的源代码都经过测试检查，每个程序都是按照“literate程序”的形式构成，为源代码提供了全面完整的解释
- 提供了非常少见的有关C语言编程技巧的文档记录
- 可以方便地在 http://www.cs.princeton.edu/software/cii/ 访问本书的所有源码

作者简介

David R. Hanson 普林斯顿大学计算机科学系教授，有着二十多年编程语言研究经验。他曾经同贝尔实验室合作开展研究工作，是适用于UNIX系统上的高质量C编译器——lcc的开发者之一。另与Christopher Fraser合著有《A Retargetable C Complier: Design and Implementation》一书，对lcc进行了讨论和分析。

ISBN 7-111-13005-7

9 787111 130055

封面设计 陈子平

HZ BOOKS 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 7-111-13005-7/TP · 2914

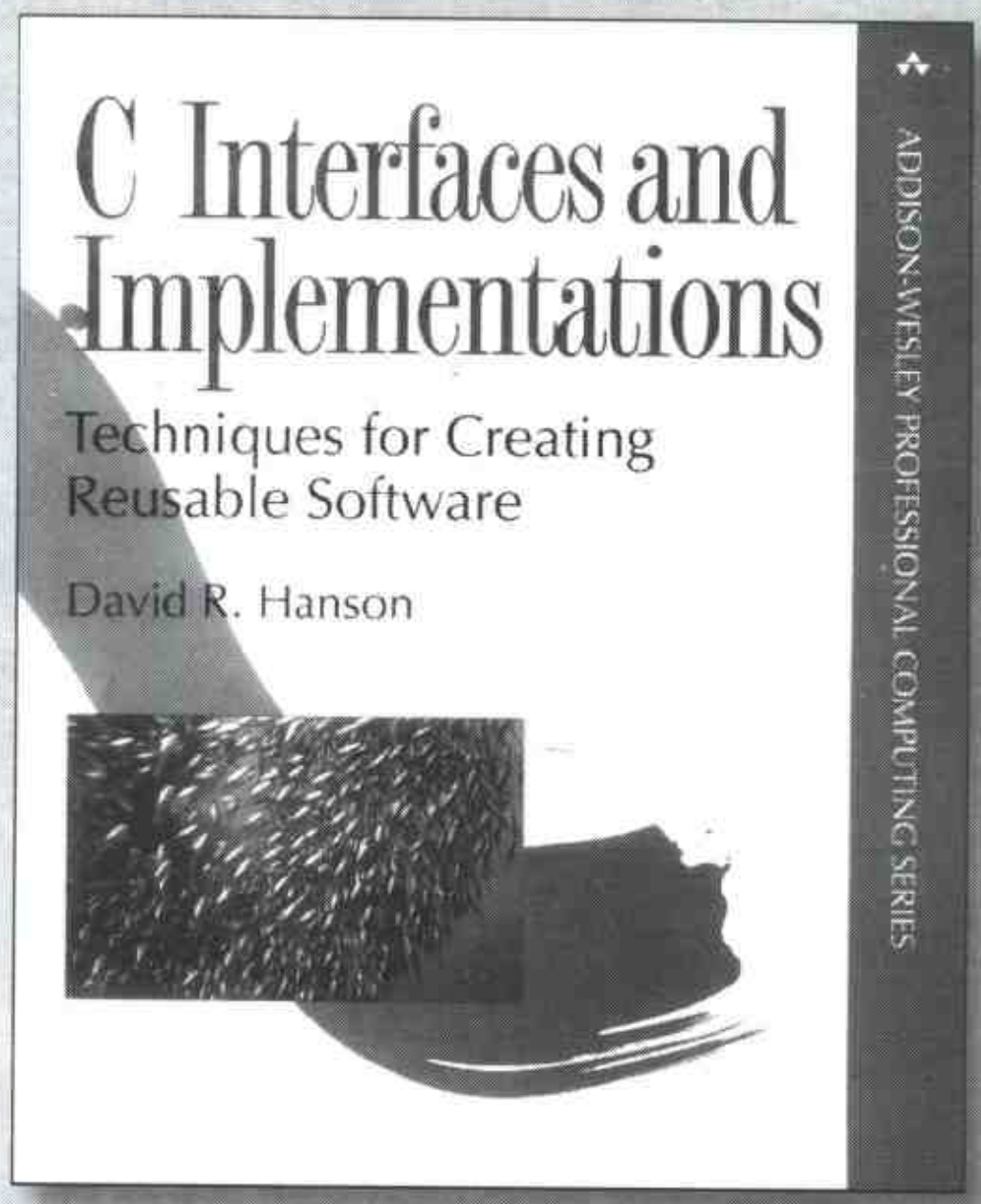
定价：35.00 元

计 算 机 科 学 丛 书

C语言接口与实现

创建可重用软件的技术

(美) David R. Hanson 著　傅蓉 周鹏 张昆琪 权威 译



C Interfaces and Implementations

Techniques for Creating Reusable Software

机械工业出版社
China Machine Press

本书概念清晰、内容翔实、实例详尽，是一本有关设计、实现和有效使用C语言库函数，掌握创建可重用C语言软件模块技术的参考指南。本书倡导基于接口的C语言设计理念及其实现技术，深入详细地描述了24个C语言接口及其实现。

本书通过叙述如何用一种与语言无关的方法将接口的设计与实现独立开来，从而形成一种基于接口的设计途径来创建可重用的API。本书是一本针对C语言程序员的不可多得的好书，也是值得所有希望掌握可重用软件模块技术的读者阅读的参考书籍。

本书版权登记号：图字：01-2002-3592

图书在版编目（CIP）数据

C语言接口与实现/（美）汉森（Hanson, D. R.）著；傅蓉等译. -北京：机械工业出版社，2004.1

（计算机科学丛书）

书名原文：C Interfaces and Implementations: Techniques for Creating Reusable Software

ISBN 7-111-13005-7

Ⅰ.C… Ⅱ.①汉…②傅… Ⅲ.C语言－程序设计 Ⅳ.TP312

中国版本图书馆CIP数据核字（2003）第080766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：蒋 祎

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1月第1版第1次印刷

787mm×1092mm 1/16·24.75印张

印数：0 001-5 000册

定价：35.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线电话：（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，不仅奠划了研究的范畴，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、研究及收藏。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，引进个关通行的教学辅导书“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
周克定	周傲英	孟小峰	岳丽华	范　明
郑国梁	施伯乐	钟玉琢	唐世渭	袁崇义
高传善	梅　宏	程　旭	程时端	谢希仁
裘宗燕	戴　葵			

译 者 序

可重用软件模块是构建大规模、可靠应用的基石。每一位程序员和软件项目经理必须掌握创建可重用软件模块的技术。literate程序设计是一种将编程代码实现与文档描述语言结合起来的编程方法，一个literate程序包含程序代码和文档，它关注文档描述，编写的代码针对于人而不是编译器。本书即使用这种激动人心的编程方式，通过叙述如何用一种与语言无关的方法将接口的设计与实现独立开来，从而设计、实现和有效使用C语言库函数，掌握创建可重用C语言软件模块技术。

本书倡导基于接口的C语言设计理念及其实现。深入详细地描述了24个C语言接口及其实现，内容包括：异常和断言、内存管理、链表、表格、集合、动态数组、序列、环、位向量、原子、格式化、低级字符串、高级字符串、扩展精度算法、任意和多精度算法以及线程等，是一本为C语言编程人员排忧解难的参考书。

C语言对于创建可重用API只提供了非常少的语言和功能支持。尽管大多数C语言程序员在自己所编写的每一个应用程序中都使用API和实现API的库，但只有相当少的程序员可以创建和发布新的、可广泛使用的API。

本书具有简洁明了的接口描述，为对接口设计感兴趣的程序员提供了一个参考手册。每一章接口的代码实现分析将帮助读者修改、扩充一个接口，或者设计相关接口，深入探讨了将数据结构以及相关算法打包到可重用模块中的技术和实现技巧。24个API和8个实例程序的源代码都经过测试检查，每个程序都是按照“literate程序”的形式构成，为源代码提供了全面完整的解释。本书还提供了非常有用但却很少见的有关C语言编程技巧的文档记录，另外读者还可以方便地在http://www.cs.princeton.edu/software/cii/访问本书的所有源码。

本书的最大特点是理论与实践相结合，书中从讲述相关的C语言基本知识和概念分析的方法入手，在此基础上结合作者的实践经验讲述如何实现基本数据结构和算法、字符串处理和并行编程接口方法。本书的作者是普林斯顿大学计算机科学系教授，有着二十多年编程语言研究经验。他曾经同贝尔实验室合作开展研究工作，是流行于Unix系统的用于C语言的高质量编译器lcc的合作开发者。书中的讲述思路清晰，语言表达明确，并且还具有丰富的示例，安排具有条理性，易于理解和掌握。使人读完有豁然开朗的感觉。

全书由傅蓉、周鹏、张昆琪、权威等进行翻译，参与翻译工作的还有刘建伟、张小辉、王秀英、周鹏、李家峻、成丽杰、吴文英、方舒凯、文静等。本书的出版是集体劳动的结晶，前导工作室全体工作人员共同完成了本书的录排、校对等工作。由于时间仓促，且译者的水平有限，在翻译过程中难免会出现一些错误，请读者批评指正。

译 者

2003年7月

前　言

现在程序员都面临着大量的关于应用程序接口（Application Programming Interface，API）的信息，大多数人都会使用API和程序库，并在其所写的每一个应用程序中实现它们，但是很少有人会创建或发布新的能广泛应用的API。事实上，程序员似乎倾向于循环使用他们自己的东西，而不愿意查找能满足他们要求的程序库，这或许是因为写特定应用程序的代码要比查找设计好的API容易。

我和下一代程序员一样感到心虚：lcc（Chris Fraser 和我给ANSI/ISO C编写的编译器）是建立在一定的背景之上的（在《A Retargetable C Compiler: Design and Implementation》一书中有关于lcc的描述，Addison-Wesley, 1995）。编译器展示了这样一种应用程序，该应用程序可以使用标准接口，并且能够创建在其他地方也可以使用的接口。这类程序的其他例子还有内存管理、字符串和符号表以及链表操作等等。但是lcc仅使用了很少的标准C库函数的例程，并且几乎没有代码能够直接应用到其他应用程序中。

本书提倡的是一种基于接口及其实现的设计方法，并且通过对24个接口及其实现的描述详细地演示了这种方法。这些接口涉及到计算机领域的很多知识，其中包括数据结构、算法、字符串处理和并发程序。这些实现并不是简单的玩具——它们是为在产品代码中使用而设计的。

C编程语言对基于接口设计方法的支持是极少的。而面向对象的语言，像C++和Modula-3，则鼓励将接口与实现分离。基于接口的设计独立于任何特定的语言，但是它要求程序员对像C一样的语言有更多的驾驭能力和更高的警惕性，因为这类语言很容易破坏带有隐含实现信息的接口，反之亦然。

然而，一旦掌握了基于接口的设计方法，就能够在服务于众多应用程序的通用接口基础上建立应用程序，从而加速开发。在一些C++环境中的基础类库就体现了这种效果。增加对现有软件的重用——接口实现库，能够减少初始开发成本，同时还能减少维护成本，因为应用程序的更多部分都建立在经过良好测试的通用接口的实现之上。

本书提供的24个接口来有多种来源，并且针对本书特别做了修正。一些数据结构中的接口——抽象数据类型，源于lcc代码和70年代末80年代初所做的Icon编程语言的实现代码（参见R.E.Griswold and M.T.Griswold, The Icon Programming Language, Prentice Hall, 1990）。其他的接口来自另外一些程序员的著作，我们将会在每一章的“参考书目浅析”部分给出详细信息。

书中提供的一些接口是针对数据结构的，但本书不是一本数据结构的书籍，因此，本书侧重点在算法引擎——包装数据结构以供应用程序使用——而不在数据结构算法本身。然而，好的接口设计总是依赖于恰当的数据结构和有效的算法，因此，本书与Robert Sedgewick的

《Algorithms in C》(Addison-Wesley, 1990)这样的传统数据结构和算法教材是相得益彰的。

大多数章节会描述一个接口及其实现；有少数章节还会描述与其相关的接口。每一章的“接口”部分将会单独给出一个明确且详细的接口描述。对于兴趣仅在于接口的程序员来说，这些节就相当于一本参考手册。少数章节还会包含“示例”部分，该节将会说明在一个简单的应用程序中一个或多个接口的使用。

每章的“实现”部分将会详细地介绍本章接口的实现代码。在一些例子中，对一个接口将会给出多种实现方法，以展示基于接口设计的优点。这些节对于修改或扩展一个接口或是设计一个相关的接口将大有裨益。许多练习题将会探究一些设计与实现的其他可行方法。如果仅是为了理解如何使用接口，可以不用阅读“实现”一节。

接口、示例和实现都以literate程序的方式给出，换句话说，源代码及其解释是按照最适合理解代码的顺序交织出现的。代码可以自动地从本书的文本文件中抽取，并按C编程语言所规定的顺序组合起来。其他包含C语言literate程序设计例子的书籍有《A Retargetable C Compiler》和D.E.Knuth写的《The Stanford GraphBase: A Platform for Combinatorial Computing》(Addison-Wesley, 1993)。

组织

本书材料可分成下面的几大类：

基础	1. 简介
	2. 接口与实现
	4. 异常与断言
	5. 内存管理
	6. 进一步内存管理
数据结构	7. 链表
	8. 表格
	9. 集合
	10. 动态数组
	11. 序列
	12. 环
	13. 位向量
字符串	3. 原子
	14. 格式化
	15. 低级字符串
	16. 高级字符串
算法	17. 扩展精度算法
	18. 任意精度算法
	19. 多精度算法
线程	20. 线程

通读第1到4章的内容将使大多数读者有所裨益，因为这几章形成了本书其余部分的框架。剩下的章可以按任何顺序阅读，尽管后面的某些章会参考其前面的内容。

第1章涵盖了literate程序设计和编程风格与效率的讨论；第2章提出并描述了基于接口的设计方法，定义了相关的术语，并演示了两个简单的接口及其实现；第3章描述了原子接口的实现原型，这是本书中最简单的具有产品质量的接口；第4章介绍了在每一个接口中都会用到的异常与断言；第5、6章描述了几乎所有的实现都会用到的内存管理；其余的每一章都描述了一个接口及其实现。

使用建议

我们假设本书的读者已经了解了在大学介绍性的编程课程中涉及到的关于C语言的内容，并且都实际使用过类似于C算法的以文本形式给出的基本数据结构。在普林斯顿，这本书的内容被用做大学二年级学生到研究生一年级的系统编程课程的教材。许多接口使用的都是高级C语言编程技巧，比如说不透明的指针和指向指针的指针等等，因此这些接口都是非常好的技术实例，且其中的技术在系统编程和数据结构课程中非常实用。

这本书可以以多种方式在课堂上使用，最简单的就是用在面向项目的课程中。例如，在编译原理课程中，学生通常需要为一个玩具语言编写一个编译器；在图形学课程中同样也经常有一些实际的项目。许多接口消除了具体项目所需要的一些令人厌烦的编程，这样就使得这类课程中的项目得到简化。这种用法可以帮助学生认识到在项目中重用代码可以节省大量劳动，并且引导学生在其项目中对自己所做的部分尝试使用基于接口的设计。后者在团队项目中特别有用，因为“现实世界”中的项目通常都是团队项目。

普林斯顿大学二年级系统编程课程的主要内容是接口与实现，其课外作业要求学生成为接口的用户、实现者和设计者。例如其中的一个作业，我在第8.1节中描述的Table接口、它的实现的目标代码以及第8.2节中描述的单词频率程序wf的说明，让学生只使用我们为Table设计的目标代码来实现wf。在下一个作业中，他们得到了wf的目标代码但必须实现Table。有时，我颠倒这些作业，但是这两种顺序对大部分学生来说都是很不一样的。他们不习惯在他们的大部分程序中只使用目标代码，并且这些作业通常都是他们第一次接触接口和程序说明中使用的半正式表示法。

最初布置的作业也介绍了作为接口说明必要组成部分的可检查的运行时错误和断言（assertion）。经过几个这样的作业之后，学生们才开始理解这些概念的意义。我禁止了突发性（unannounced）崩溃，也就是说断言错误的诊断是不会导致崩溃的。运行崩溃的程序将被判为零分。这样做似乎过于苛刻，但是它能够引起学生们的注意；而且也能够理解安全语言的好处，例如ML和Modula-3，在这些语言中，不会出现突发性崩溃（这种分级策略没有它听上去那么苛刻，因为在分成多个部分的作业中，只有产生冲突的那部分作业才会得到惩罚，而且不同的作业将得到不同的分数。我给过许多0分，但是从来没有因此导致任何一个学生的课程成绩很低）。

一旦学生们有了属于他们自己的少数几个接口后，接下来就让他们设计新的接口并沿用他们以前的设计选择。例如，Andrew Appel最喜欢的一个作业是一个原始的测试程序。学生

们以组为单位设计一个作业需要的任意算术精度的接口，作业的结果类似于第17到19章中描述的接口。不同的组设计的接口不同，完成后对这些接口进行比较，一个组对另一个组设计的接口进行评价，这样做很有启迪作用。Kai Li的需要一个学期来完成的项目也达到了同样的学习实践效果，该项目使用Tcl/Tk系统（J.K. Ousterhout，《Tcl and the TkToolkit》，Addison-Wesley 1994）以及学生们设计和实现的编辑程序专用的接口，构建了一个基于X的编辑程序。Tk本身就提供了另一个很好的基于接口设计的例子。

在高级课程中，我通常把作业打包成接口，让学生自由地修改和改进，甚至改变作业的目的。给他们一个出发点可以减少完成作业所需的时间，并允许他们做一些实质性的修改，这样鼓励了有创造性的学生去探索新的解决办法。通常，那些不成功的方法比成功的方法更有教育意义。学生不可避免地会走错路，为此也付出了更多的开发时间。但只有当他们事后再回过头来看，才会了解所犯的错误，也才会知道设计一个好的接口是很困难的，但是值得付出努力，而且到最后，他们几乎都会转到基于接口的设计上来。

如何得到本书的软件

本书中的软件已经在以下的平台上通过了测试：

处理器	操作系统	编译器
SPARC	SunOS 4.1	lcc 3.5
		gcc 2.7.2
Alpha	OSF/1 3.2A	lcc 4.0
		gcc 2.6.3
		cc
MIPS R3000	IRIX 5.3	lcc 3.5
		gcc 2.6.3
		cc
MIPS R3000	Ultrix 4.3	lcc 3.5
		gcc 2.5.7
Pentium	Windows 95	Microsoft Visual C/C++ 4.0
	Windows NT 3.51	

其中少数实现是针对特定机器的；这些实现假设机器使用的是二进制补码表示的整数和IEEE浮点算术，并且无符号的长整数可以用来保存对象指针。

本书中所有的源代码在ftp.cs.princeton.edu的目录pub/packages/cii下，用匿名账号就可以得到。使用ftp客户端软件连接到ftp.cs.princeton.edu，转到pub/packages/cii目录，下载README文件，文件中说明了目录的内容以及如何下载出版物。

大多数最新的出版物通常都是以ciixy.tar.gz或ciixy.zip的文件名存储的，其中xy是版本号，例如10是指版本1.0。ciixy.tar.gz是用gzip压缩的UNIX tar文件，而ciixy.zip是与PKZIP 2.04g版兼容的ZIP文件，ciixy.zip中的文件都是DOS/Windows下的文本文件，每一行是以回车和换行符结束的。ciixy.zip同时也可以在美国在线、CompuServe以及其他在线服务器上得到。

在World Wide Web中的URL地址http://www.cs.princeton.edu/software/cii/上同样也可以得到相应的信息。该页面还包括了一些错误报告说明。

致谢

自1970年末以来，在我自己的研究项目以及亚利桑那州大学和普林斯顿大学的课程中，我就已经使用过本书中的一些接口。选这些课程的学生成为了我设计的这些接口的初稿的试用者。这些年来他们的反馈是本书中代码与说明的重要来源。我要特别感谢的是普林斯顿大学的学生对COS 217和COS 596课程的参与，正是他们在不知不觉中参与了本书中大多数接口的初步设计。

利用接口开发是DEC公司（已与Compaq合并）的系统研究中心（System Research Center，SRC）的主要工作方式，而且1992年和1993年暑假我在SRC的工作经历——从事Modula-3项目的开发——消除了我对这种方法有效性的怀疑。我非常感谢SRC对我工作的支持，以及Bill Kalsow、Eric Muller和Greg Nelson提供的许多有意义的讨论。

我还要感谢IDA在普林斯顿的通信研究中心（Center for Communications Research，CCR）和La Jolla，感谢他们在1994年暑假和1995-1996休假年对我的支持。还要感谢CCR提供了一个理想的地方让我从容规划并完成了本书。

与同事和学生的技术交流也在许多方面对本书提供了帮助。一些即使看上去不相关的讨论也促使我对代码及其说明进行改进。感谢Andrew Appel、Greg Astfalk、Jack Davidson、John Ellis、Mary Fernández、Chris Fraser、Alex Gounares、Kai Li、Jacob Navia、Maylee Noah、Rob Pike、Bill Plauger、John Reppy、Anne Rogers和Richard Stevens。感谢Rex Jaeschke、Brian Kernighan、Taj Khattra、Richard O'Keefe、Norman Ramsey和David Spuler，他们仔细阅读了本书的代码和内容，对确保代码和内容的质量都给予了重要的帮助。

David R.Hanson

目　录

第1章 简 介

一个大型程序通常是由许多小模块组成的。这些模块给出了程序中使用的函数、过程和数据结构。理想的情况下，大部分模块都是现成的并且都来自于库函数；只有那些正在开发的应用程序专用的模块需要从头写。假设库代码已经彻底测试过，那么只有那些应用程序专用的代码才可能包含错误，因此程序调试可以只限定在这些代码中。

但不幸的是，这些理论上的理想情况在实际开发中很少发生。大多数程序都是从头书写，且一般只在使用I/O和存储器管理等最低级设备时才会使用库函数。即使对这类低级组件，程序员也常常编写应用程序专用的代码；比如说，在应用程序中，用定制的存储器管理函数替代C语言库函数malloc和free是经常会出现的。

出现这种情况有几个毋庸质疑的理由：其一是健壮性强、设计良好的通用模块库很少，其中一些可使用的库都很平庸且缺少标准。C语言库从1989年起就已经标准化了，但直到最近才出现在大部分平台中。

另一个理由就是库的大小：一些库太大了以致于很难掌握它们。如果掌握这些库函数所需要的努力接近于编写应用程序所花费的精力，程序员很可能为了方便而重新实现他们所需要的那部分库函数。最近发展迅速的用户界面库通常就存在这个问题。

程序库的设计和实现是很困难的。设计者必须小心处理通用性、简单性和有效性问题。如果一个程序库中的例程和数据结构太通用了，就有可能导致使用困难或难于达到它们想要达到的目的。如果它们太简单了，就可能不满足应用程序的使用需要。如果它们太易混淆，编程者也不会使用它们。C语言库本身有一些容易混淆的例子，例如它的realloc函数。

程序库的实现者面临着类似的困难。即使设计做得很好，如果实现得不好也无法吸引用户。如果一个实现运行太慢或代码量太大——或只是感觉上是这样——编程者也会设计他们自己的程序来代替库函数。最坏的情况是，如果一个实现有错误，那么它将打破上述理想状态并且使得程序库变得毫无用处。

本书描述了一个程序库的设计和实现，该程序库适用于用C语言编写的各种应用。这个程序库给出了一系列模块，为“小规模编程（programming-in-the-small）”提供函数和数据结构。这些模块适合用作应用程序中的“零件”或只有几千行的应用程序组件。

在接下来的章节中所描述的大部分工具都在大学的数据结构和算法课程中涉及到了。但是，在本书中，更多的注意力将被放在它们是如何打包的，以及怎样使它们更健壮上。每个模块都给出了一个接口及其实现。在第2章中讲述的设计方法将模块的说明从它们的实现中分离出来，提高了说明的清晰度和精确度，并且有利于提供健壮的实现。

1.1 literate程序

本书不是用规则来描述模块，而是用例子。每章都完整地叙述了一个或两个接口以及它们的实现。这些描述都是用literate程序表示的，即接口及其实现代码和解释它的语句交织一起。更重要的是，本书每章文字本身都是它所描述的接口和实现的源代码。代码自动提取自本书的文本，所见即所得。

literate程序由英文和带标签的程序代码块组成　例如：

```
⟨compute x • y⟩≡
  sum = 0;
for (i = 0; i < n; i++)
    sum += x[i]*y[i];
```

2

定义了一个名为<*compute x • y*>的代码块；它的代码计算了数组x和y的点积。这个块的使用是通过在另一个代码块中调用它来实现的：

```
⟨function dotproduct⟩≡
  int dotProduct(int x[], int y[], int n) {
      int i, sum;

      ⟨compute x • y⟩
      return sum;
  }
```

当代码块<*function dotproduct*>从包含该章的文本文件中提取出来的时候，它的代码被原样复制，块被其代码代替，依此类推。因此提取<*function dotproduct*>的结果是包含以下代码的文件：

```
int dotProduct(int x[], int y[], int n) {
    int i, sum;

    sum = 0;
    for (i = 0; i < n; i++)
        sum += x[i]*y[i];
    return sum;
}
```

literate程序可以用许多小的单元表示，但是文档却是完整的。英文包含传统的程序注释，并且不局限于程序语言的注释习惯。

代码块工具（chunk facility）将literate程序从由编程设计语言决定的次序限制中解脱出来。代码可以用最容易理解的任何一种次序显示，而不是用由规则指定的次序显示，例如程序实体必须在使用它们之前定义。

本书中使用的literate编程系统还有几个特征，这些特征有助于分段描述程序。为了说明

3

这些特征并提供一个literate C 程序的完整例子，本节接下来描述了一个名为double的程序，用于检测输入中相邻且相同的单词，例如“the the.”。例如如下UNIX命令：

```
% double intro.txt inter.txt
intro.txt:10: the
inter.txt:110: interface
inter.txt:410: type
inter.txt:611: if
```

说明“the”在文件intro.txt中出现了两次，第二次出现在第10行；例中还显示了在inter.txt文件中出现了两次“interface”、“type”和“if”。如果double不带参数调用，那么它从标准输入中读取数据，输出时省略文件名。例如：

```
% cat intro.txt inter.txt | double
10: the
143: interface
343: type
544: if
```

在本书示例中，用户输入的命令用斜体表示，而输出用代码体表示。

下面我们看看double，首先定义一个根代码块（root chunk），程序的每个组成部分用其他代码块表示：

```
⟨double.c 4⟩≡
  ⟨includes 5⟩
  ⟨data 6⟩
  ⟨prototypes 6⟩
  ⟨functions 5⟩
```

为了方便，根代码块用程序文件名标识；通过展开代码块*<double.c 4>*展开程序。其他代码块用double的顶级组件标识。这些组件按照由C程序设计语言规定的次序列出，但是它们可以按任何次序给出。

*<double.c 4>*中的数字4是代码块定义开始的页码。在*<double.c 4>*中使用的代码块中的数字是它们定义开始的页码，这些页码有助于读者浏览代码。

main函数处理double的参数。它打开每个文件并调用doubleword扫描文件：

```
⟨functions 5⟩≡
  int main(int argc, char *argv[]) {
      int i;

      for (i = 1; i < argc; i++) {
          FILE *fp = fopen(argv[i], "r");
          if (fp == NULL) {
              fprintf(stderr, "%s: can't open '%s' (%s)\n",
                  argv[0], argv[i], strerror(errno));
              return EXIT_FAILURE;
          } else {
              doubleword(argv[i], fp);
              fclose(fp);
          }
      }
```

```
    if (argc == 1) doubleword(NULL, stdin);
    return EXIT_SUCCESS;
}
```

⟨*includes* 5⟩≡
```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
```

函数doubleword需要从文件中读取单词。程序要求一个单词必须是一个或多个无空格字符，并且不区分大小写。getword从一个打开的文件中将下一个单词读入buf[0..size-1]并返回1；读到文件末尾时，返回0。

⟨*functions* 5⟩+≡
```
int getword(FILE *fp, char *buf, int size) {
    int c;

    c = getc(fp);
    ⟨scan forward to a nonspace character or EOF 6⟩
    ⟨copy the word into buf[0..size-1] 7⟩
    if (c != EOF)
        ungetc(c, fp);
    return ⟨found a word? 7⟩;
}
```

5

⟨*prototypes* 6⟩≡
```
int getword(FILE *, char *, int);
```

这个代码块说明了literate编程的另一个特征：代码块标识<*functions 5*>后面跟的“+≡”指出getword的代码是附加在代码块<*functions 5*>后的，因此代码块现在包含main和getword的代码。这个特征允许一个代码块中的代码一次只写一部分。后续扩充定义的代码块标签中的页码指向的是该代码块中的第一个代码段定义，因此根据它可以很容易找到一个代码块定义的开始。

既然getword在main之后，因此在main中调用getword需要一个原型，这也就是代码块<*prototypes 6*>的目的。这个代码块在某种意义上可以说是对C的必须在使用之前先声明规则的让步，但是如果定义是一致的，并且在根代码块中该代码块出现在<*functions 5*>之前，那么函数就可以以任意次序给出。

除了从输入中得到下一个单词，getword还会在遇到换行符时将linenum加1，doubleword会在显示输出结果时使用linenum。

⟨*data* 6⟩≡
```
int linenum;
```

⟨*scan forward to a nonspace character or EOF* 6⟩≡
```
for ( ; c != EOF && isspace(c); c = getc(fp))
    if (c == '\n')
```

```
        linenum++;
```

⟨*includes* 5⟩+≡
```
  #include <ctype.h>
```

linenum的定义给出了一个与C要求的顺序不一样的例子，代码块可以按任何次序放置。在这里，linenum是在第一次使用时给出定义，而不是像C中所要求的那样在文件的开始处给出或在定义getword之前给出。

size的值是对存储在getword中单词长度的限制，getword丢弃多余的字符并将大写字符转换成小写字符：

⟨*copy the word into* buf[0..size-1] 7⟩≡
```
  {
      int i = 0;
      for ( ; c != EOF && !isspace(c); c = getc(fp))
          if (i < size - 1)
              buf[i++] = tolower(c);
      if (i < size)
          buf[i] = '\0';
  }
```

索引i与size-1作比较，以保证在单词的末尾有空间存储一个空字符；if语句保证赋值操作能够处理size为0的情况，这种情况在double中不会发生，但是这种保护性编程有助于发现“不可能发生”的错误。

getword剩下的工作只是：如果buf中存在一个单词就返回1，其他情况则返回0：

⟨*found a word?* 7⟩≡
```
  buf[0] != '\0'
```

这个定义表明：代码块并不一定要与C语句或任何其他语言的语法单元相对应，它们只是简单的文字描述。

doubleword读入每个单词，并将它与前面的单词做比较，判断是否重复。它只检查以字母开头的单词：

⟨*functions* 5⟩+≡
```
  void doubleword(char *name, FILE *fp) {
      char prev[128], word[128];

      linenum = 1;
      prev[0] = '\0';
      while (getword(fp, word, sizeof word)) {
          if (isalpha(word[0]) && strcmp(prev, word)==0)
              ⟨word is a duplicate 8⟩
          strcpy(prev, word);
      }
  }
```

⟨*prototypes* 6⟩+≡

```
    void doubleword(char *, FILE *);

⟨includes 5⟩+≡
    #include <string.h>
```

输出结果是很容易的，但是要注意，文件名和它后面的冒号只有在name不为空时才打印：

```
⟨word is a duplicate 8⟩≡
    {
        if (name)
            printf("%s:", name);
        printf("%d: %s\n", linenum, word);
    }
```

这个代码块定义为一个合成语句，它可以作为使用它的if语句的结果出现。

1.2 编程风格

double实例说明了本书许多程序中所使用的格式上的习惯。对程序来说，更重要的是易于人们阅读和理解，而不是让计算机更容易编译。编译器不在乎变量的名字，也不在乎代码是如何排版的或程序是怎样被划分为模块的。但是，这类细节对于使程序员更容易地读懂一个程序有很大的影响。

本书中的代码沿袭C程序的格式习惯。它对变量、类型和例程的命名使用了一致性的原则，并且将格式限定为一致缩进风格。格式习惯并不是一组必须不惜任何代价去遵循的严格规则，它们表述的是一种寻求最大可读性和可理解性的编程哲学。因此，这些规则可以在任何需要的时候被改变，以利于强调代码的重要方面或使复杂的代码更具可读性。

8 通常，对全局变量和例程使用较长的、容易称呼的名字，而对局部变量用短的名字，可能就是普通的数学记号的翻版。在<*compute x · y*>中的循环变量i就是后一种习惯的例子。基于近于传统的目的，对索引和变量使用长名通常使得代码更难读懂。例如，在下面的代码中：

```
sum = 0;
for (theindex = 0; theindex < numofElements; theindex++)
    sum += x[theindex]*y[theindex];
```

变量名使得代码读不懂。

变量都是声明在它们第一次被使用的程序代码附近，可能是在代码块中。例如，getword中变量linenum的声明就是这样。局部变量尽可能声明在使用它们的复合语句的开始处，例如<*copy the word into buf[0..size-1] 7*>中i的声明。

通常，过程和函数选择的名字应能反映过程的作用以及函数返回值。因此，getword返回输入中的下一个单词，而doubleword寻找并显示至少出现两次的单词。大多数例程都很短，代码不超过一页；代码块则更短，通常不到12行。

代码中几乎没有什么注释，因为包含代码的代码块周围的上下文代替了注释。对注释格式习惯的看法各不相同。本书遵循C语言编程中的经典指导，即注释尽可能的少。清晰、使

用好的命名以及有缩进格式的代码通常就能解释它本身。注释只有在某些时候才有必要，例如数据结构的细节、算法中的特殊情况以及异常情况等等。编译器不能检查注释和代码的一致性；可能产生误导的注释比没有注释更糟。最后，那些杂乱的、过多的印刷样式中的注释，除了给代码带来混乱外，其他什么用处也没有。

literate编程避免了在注释问题中的许多争论，因为它不受程序设计语言注释机制的限制。程序员可以使用最能传达他们意图的任何格式，包括表格、等式、图片以及引用。literate编程提高了程序的正确性、精确性和清晰度。

9

本书的代码是用C编写的，它使用大量有经验的C程序员普遍接受且期望的习惯用法。其中一些对新手来说也许会有些迷惑，但是他们如果想熟练使用C的话就必须掌握这些。这些习惯用法还包括最容易混淆的指针，因为C对指针的使用提供了几个独特的、有表达力的操作。将一个字符串复制到另一个字符串并返回目的字符串的库函数strcpy就说明了老手和新手写的代码之间的差别，后者的代码常使用数组：

```
char *strcpy(char dst[], const char src[]) {
    int i;

    for (i = 0; src[i] != '\0'; i++)
        dst[i] = src[i];
    dst[i] = '\0';
    return dst;
}
```

而老手习惯使用指针：

```
char *strcpy(char *dst, const char *src) {
    char *s = dst;

    while (*dst++ = *src++)
        ;
    return s;
}
```

这两种代码都是strcpy合理的实现。指针实现版本使用了最普遍的习惯用法，它把指针赋值、自增以及测试赋值的结果融合到一个单一的赋值表达式中。它同时还修改了它的参数dst和src，这在C中是可以接受的，因为所有的参数都是通过值传递的，也就是说参数只是局部初始化。

然而在某种情况下应该选择数组版本而不是指针版本实现。例如，对所有的程序员，不管他们是否熟悉C，数组版本都更容易理解。而指针版本是最有经验的C程序员常常用到的，因此也是程序员阅读现存代码时最有可能遇到的。本书可以帮助你学习这些习惯用法，理解C强大的指针，避免犯常见的错误。

10

1.3 效率

程序员看上去对效率有些迷惑。他们可能要花几个小时修改代码以使其运行得更快，可

不幸的是，这种努力大部分都是白费的，程序员对猜测哪段程序运行时间比较长的直觉是非常不准的。

使程序运行得更快，往往也使得程序变得越来越大，更难理解的是，包含错误的可能性也更大。除非对执行时间的度量显示出程序运行得太慢，否则没有必要进行这样的改变。程序只需要运行得足够快而不必尽可能的快。

进行这样的改变通常是在未经测试的纯编程环境中完成的。如果程序运行得太慢，找到它的瓶颈的惟一方法就是度量它。一个程序的瓶颈很少发生在你猜想它们发生的地方，也很少是由你猜测的原因引起的。即使你找到了正确的地方，也只有在那个地方花费的运行时间占运行总时间的很大一部分时才需要进行这样的修改。如果I/O占了60%的程序运行时间，那么在查询例程中节省1%的运行时间是没有任何意义的。

进行这样的调整通常会引入错误。即便是运行最快的程序，如果运行时崩溃也不能算成功，因为可靠性比效率更重要。从长远来看，发布一个快但是经常崩溃的软件比发布一个运行稳定而又足够快的软件花费更大。

通常，这样的调整是在有问题的层次上完成的。本身就很快的算法的直接实现要优于对本身运行较慢的算法做手工调整。例如，压缩线性查找内部循环外的指令肯定不如首先使用二分查找更有效。

这样的调整不能改变拙劣的设计。如果程序在所有的地方运行都慢，那么这种低效表现很有可能是设计本身的问题。当设计源于编写得不好或不准确的问题说明书或根本没有进行总体设计时，常常会出现这种情况。

本书中的大部分代码都使用有效的算法，这些算法在一般情况下具有良好的性能，而且最坏情况下的性能也很容易刻画。它们在大部分应用中对典型输入的执行时间几乎总是足够
11 快的；在某些应用中，性能可能会有问题的情况也能明确识别出来。

一些C程序员为了追求效率而大量使用宏指令和条件编译指令。本书尽可能避免使用这两种指令。为了避免函数调用而使用宏指令是很没有必要的，只有当客观的度量表明调用的花费可能超过剩下代码的执行时间时才有必要使用宏指令。I/O是少数几个合理使用宏指令的地方之一，例如标准的I/O函数getc、putc、getchar和putchar通常是作为宏指令来实现的。

条件编译语句通常用来设置代码以适应特定的平台或环境，或用来关闭或打开代码的调试。这些都是实际存在的问题，条件编译语句通常是解决它们的最简单方法，但总是使得代码更难懂。通常更有用的方法是重写代码来实现在执行的过程中选择所依赖的平台。例如，可以选择六种体系结构中的一种，在执行期间产生代码的简单编译器，即交叉编译器，比必须配置和构造六种不同的编译器要有用得多，而且也更容易维护。

如果一个应用程序必须在编译时间内进行配置，版本控制工具也比C的条件编译工具要好。代码并不是用预处理指令给出的，这些预处理指令使得代码很难阅读，并且不清楚要编译什么，不要编译什么。而用版本控制工具，你所看到的就是所要执行的；同时，这些工具也可以很好地用以了解性能的改进。

参考书目浅析

ANSI标准（1990）以及技术上等价的ISO标准（1990）是标准C程序库的权威性参考书目，但是Plauger(1992)给出了一个更详细的描述和一个完整的实现。同样地，这些标准是关于C的总结文章，但是Kernighan和Ritchie（1988）可能是使用最广泛的参考书。Harbison和Steele（1995）可能是关于标准的最新版本，而且它也叙述了如何书写“干净的C程序（clean C）”——即C代码可以用C++编译器编译。Jaeschke（1991）将标准C中的精华精简成紧凑的字典格式，它对于C程序员来说是一本非常有用的参考书。

尽管Kernighan和Plauger（1976）用特别的工具包含了本书中的代码，但他们的软件工具却给出了早期literate程序的例子。WEB是明确为literate编程设计的最早的工具之一。Knuth（1992）叙述了WEB和它的一些变种及使用；Sewell（1989）是一本关于WEB的介绍指南。Simpler工具（Hanson 1987；Ramsey 1994）对提供众多WEB的核心功能大有帮助。本书使用notangle，它是Ramsey的noweb系统中一种展开代码块的程序。noweb同样也被Fraser和Hanson（1995）用来将一个完整的C编译器表示成一个literate程序。这个编译器也是一个交叉编译器。

double摘取自Kernighan和Pike（1984），是用AWK程序设计语言（Aho,Kernighan和Weinberger 1988）实现的。尽管有些年代了，但Kernighan和Pike仍然是关于UNIX编程原理最好的书之一。

学习好的编程风格的最好方法就是阅读使用好的风格书写的程序。本书沿袭了在Kernighan和Pike（1984）以及Kernighan和Ritchie（1988）中使用的经久不衰的编程风格。Kernighan和Plauger（1978）是关于编程风格的最经典的一本书，但是它没有包含任何用C写的例子。Ledgard的简明书籍（1987）提供了类似的建议，Maguire（1993）提出了一个从PC编程世界出发的观点。Koenig（1989）揭示了C的缺陷并且突出说明了应该避免之处。McConnell（1993）提供了关于程序构造许多方面的有用建议，并且从正反两方面对使用goto语句进行了讨论。

学习编写有效代码的最好的方法是对算法有全面的基础知识并阅读其他有效的代码。Sedgewick（1990）研究了所有大多数程序员应该知道的重要算法，Knuth（1973a）给出了基本算法的详细叙述。Bentley（1982）提供了170页的关于如何编写有效代码的常识和好的建议。

练习

1.1　一个单词以换行字符结束时，getword函数在<*scan forward to a nonspace or EOF 6*>中而不是在<*copy the word into buf[0..size-1] 7*>之后将linenum加1，说明理由。如果linenum在<*copy the word into buf[0..size-1] 7*>之后加1又会怎样？

1.2　当double函数碰到输入中有三个或更多相同单词时，输出结果是什么？修改double

12

函数以保持这个特征。

13 1.3 许多有经验的C程序员会在strcpy的循环中进行一个明确的比较:

```
char *strcpy(char *dst, const char *src) {
    char *s = dst;

    while ((*dst++ = *src++) != '\0')
        ;
    return s;
}
```

这个显式比较避免了赋值错误。一些C编译器和相关的工具，像Gimpel软件公司的PC-Lint和LCLint（Evans 1996），当赋值的结果被用作条件时会发出警告，因为这种使用方法是一个

14 常见的错误来源。如果你有PC-Lint或LCLint，可以对一些“测试过”的程序做这个实验。

第2章　接口与实现

一个模块由两部分组成：接口和实现。接口指明模块要做什么，它声明了使用该模块的代码可用的标识符、类型和例程；实现指明模块是如何完成其接口声明的目标的。一个给定的模块通常只有一个接口，但是可能会有许多种实现能够提供接口所指定的功能。每个实现可能使用不同的算法和数据结构，但是它们都必须符合接口所给出的使用说明。

客户调用程序（client）是使用某个模块的一段代码。客户调用程序导入接口；而实现导出接口。客户调用程序只需要了解接口即可。实际上，它们可能只有某个实现的目标代码。由于多个客户调用程序是共享接口和实现的，因此使用实现的目标代码避免了不必要的代码重复。同时也有助于避免错误，因为接口和实现只需一次编写和调试就可多次使用。

2.1　接口

接口只需要指明客户调用程序可能使用的标识符即可，应尽可能地隐藏一些无关的表示细节和算法，这样客户调用程序可以不必依赖于特定的实现细节。这种客户调用程序和实现之间的依赖——耦合——可能会在实现改变时引起错误；当这种依赖性埋藏在一些关于实现隐蔽的或是不明确的假设中时，这些错误可能很难修复。因此一个设计良好且描述精确的接口应该尽量减少耦合。

C语言对接口与实现的分离只提供最基本的支持，但是简单的约定能给接口/实现方法论带来巨大的好处。在C语言中，接口在头文件声明，头文件的文件扩展名通常为.h。该头文件声明了客户调用程序可以使用的宏、类型、数据结构、变量以及例程。用户用C语言的预处理指令#include导入接口。

下面的例子说明了本书的接口中所使用的一些约定。接口

⟨*arith.h*⟩≡

```
extern int Arith_max(int x, int y);
extern int Arith_min(int x, int y);
extern int Arith_div(int x, int y);
extern int Arith_mod(int x, int y);
extern int Arith_ceiling(int x, int y);
extern int Arith_floor  (int x, int y);
```

声明了6个整型算术函数，实现则提供了每个函数的定义。

该接口的名字为Arith，接口头文件也相应地命名为arith.h。接口的名字以前缀的形式出现在接口的每个标识符中。这个约定不是很合适，但C语言几乎没有提供其他的选择。文件范围内的所有标识符——变量、函数、类型定义以及枚举常量——共享一个专用名字空间；

所有的全局结构、共用体以及枚举标签共享另一个专用名字空间。在一个大型程序中，很容易在其他不相关的模块中使用相同的名字，而名字的用途不同。一种避免名字冲突的方法就是使用前缀，比方说模块名。大型程序很容易就有几千个全局标识符，但是通常只有几百个模块。模块名不仅提供了合适的前缀，而且还有助于整理客户调用程序代码。

Arith接口还提供了一些标准C函数库中没有的但很有用的函数，并为除法和取模提供了良好的定义，而标准C中并没有给出这些操作的定义或只提供基于实现的定义。

16 Arith_min和Arith_max返回其整型参数中的最小值和最大值。

Arith_div返回y除以x得到的商，Arith_mod返回相应的余数。当x和y同为正或同为负时，Arith_div(*x*, *y*)等价于*x*/*y*，Arith_mod(*x*, *y*)等价于*x*%*y*。而当操作数的符号不相同时，C的内嵌操作的返回值就取决于具体的实现。当*y*等于0时，Arith_div和Arith_mod也等价于*x*/*y*和*x*%*y*。

标准C认为只要*x*/*y*合法，(*x*/*y*) · *y*+*x*%*y*就一定等于*x*。当其中一个操作数是负数的时候，这种语义允许整数除法向零或向无穷小取整。例如，如果–13/5等于–2，那么标准C就认为–13%5一定等于–13–(–13/5) · 5=–13–(–2) · 5=–3；但是如果–13/5等于–3，那么–13%5就等于–13–（–3）· 5=2。

因此内嵌的操作只对正操作数有用。标准库函数div和ldiv接收两个整数或长整数作为输入，返回商和余数，并存入一个结构中相应的字段quot和rem。其语义已定义好了：它们总是向零取整，因此div(–13,5).quot总是等于–2。Arith_div和Arith_mod的语义也同样定义好了：它们总是趋近数轴的左侧取整；当操作数符号相同时趋近于0，而当操作数的符号不同时趋近无穷小，因此Arith_div(–13，5)返回–3。

Arith_div和Arith_mod都是用更精确的数学术语定义的。Arith_div(*x*, *y*)是不超过实数*z*的最大整数，其中*z*满足*z* · *y*=*x*。因此，若*x*=–13，*y*=5或*x*=13，*y*=–5，那么*z*=–2.6，所以Arith_div(–13,5)的结果是–3。Arith_mod(*x*,*y*)被定义为*x*–*y* · Arith_div(*x*,*y*)，因此Arith_mod(–13,5)等于–13–5 · (–3)=2。

函数Arith_ceiling和Arith_floor遵循类似的约定。Arith_ceiling(*x*,*y*)返回不小于实数商*x*/*y*的最小整数，而Arith_floor(*x*,*y*)返回不超过实数商*x*/*y*的最大整数。对所有的操作数，Arith_ceiling返回*x*/*y*数轴右边的整数，而Arith_floor返回*x*/*y*数轴左边的整数。例如：

```
Arith_ceiling( 13,5) =   13/5 =   2.6   =   3
Arith_ceiling(-13,5) =  -13/5 =  -2.6   =  -2
Arith_floor  ( 13,5) =   13/5 =   2.6   =   2
Arith_floor  (-13,5) =  -13/5 =  -2.6   =  -3
```

17

不幸的是，即使对于像Arith这样简单的接口，也要遵循这种繁琐的规定，但这对大多数接口而言是典型的和必要的。大多数程序设计语言在它们的语义中都存在漏洞，一些操作的准确含义被错误定义或根本就不定义。C的语义弥补了这些漏洞。设计良好的接口也能堵上这些漏洞，对于缺少定义的补充定义，对语言声明了但没有定义或只在实现定义了的行为也做了明确的判断。

Arith并不只是一个为了显示C的缺陷而刻意构造的例子。例如，对于包含取余的算法，

它是有用的，就像在散列表中使用的那样。假设i的范围是从0到$N-1$，其中N大于1，每次i加1和减1都必须以N为模。也就是说，如果i是$N-1$，那么$i+1$就是0；相反，如果i是0，那么$i-1$就是$N-1$。表达式

```
i = Arith_mod(i + 1, N);
i = Arith_mod(i - 1, N);
```

才是i加减1的正确表达式。表达式$i = (i+1)\%N$同样也是正确的，但是$i=(i-1)\%N$就不对了，因为当i是0时，$(i-1)\%N$可以是-1或$N-1$。程序员在$(-1)\%N$返回$N-1$的机器上使用$(i-1)\%N$时，会得到正确的结果，但是将代码放到一个$(-1)\%N$返回-1的机器上运行时，得到的结果可能会让他们感到非常吃惊。使用库函数div(x,y)也起不到任何帮助作用，它返回一个结构，结构的quot和rem字段保存了x/y的商和余数。当i是0时，div$(i-1,N)$.rem总是-1。还可以使用$i=(i-1+N)\%N$，但是它只有在$i-1+N$不会溢出时才能使用。

2.2 实现

一个实现导出一个接口。它定义了必要的变量和函数以提供接口所规定的功能。一个实现揭示了表示的细节和接口给出的特定行为的算法，但是理想的情况是，客户调用程序根本不需要看见这些细节。客户调用程序通常是通过从程序库中调用它们来完成所有目标代码的实现。

一个接口可能有多个实现，但只要实现符合该接口，那么它就可以改变而不会对客户调用程序产生影响。不同的实现可以提供更好的性能，例如，设计良好的接口可以避免对机器的依赖性，但是也可能使得实现必须依赖于机器，因此对使用接口的不同机器可能需要不同的或部分不同的实现。 18

在C语言中，一个实现是由一个或多个.c文件提供的。一个实现必须提供其导出的接口所指定的功能。实现应包含接口的.h文件，以保证它的定义与接口的声明是一致的。然而，除此之外，C中没有提供其他语言机制检查实现的一致性。

和接口一样，本书中描述的实现也有如arith.c所示的固定格式：

```
⟨arith.c⟩≡
  #include "arith.h"
  ⟨arith.c functions 19⟩

⟨arith.c functions 19⟩≡
  int Arith_max(int x, int y) {
      return x > y ? x : y;
  }

  int Arith_min(int x, int y) {
      return x > y ? y : x;
  }
```

除了<*arith.c functions 19*>外，涉及其他实现的代码块可能命名为<*data*>、<*types*>、

<macros>、*<prototypes>*等等。如果不会产生混淆，代码块中的文件名（像arith.c等）可以略去。

当Arith_div的参数的符号不同时，它执行除法有两种可能。如果按趋近0取整，且y不能整除x，那么Arith_div(x,y)等于x/y−1；否则，x/y执行以下语句：

```
⟨arith.c functions 19⟩+≡
  int Arith_div(int x, int y) {
      if (⟨division truncates toward 0 20⟩
      &&  ⟨x and y have different signs 20⟩ && x%y  != 0)
          return x/y - 1;
      else
          return x/y;
  }
```

19

在前面一节的例子中，用−13除以5，测试了除法取整的方式。通过测试x与y哪个小于0，并比较测试的结果，就能核对符号：

```
⟨division truncates toward 0 20⟩≡
  -13/5 == -2

⟨x and y have different signs 20⟩≡
  (x < 0) != (y < 0)
```

Arith_mod可以按它所定义的那样实现：

```
int Arith_mod(int x, int y) {
    return x - y*Arith_div(x, y);
}
```

Arith_mod也可以使用“%”操作符对Arith_div中同样的条件进行测试。当那些条件都成立时，

```
Arith_mod(x,y)  = x - y*Arith_div(x, y)
                = x - y*(x/y - 1)
                = x - y*(x/y) + y
```

带下划线的子表达式就是标准C所定义的x%y，因此Arith_mod就是

```
⟨arith.c functions 19⟩+≡
  int Arith_mod(int x, int y) {
      if (⟨division truncates toward 0 20⟩
      &&  ⟨x and y have different signs 20⟩ && x%y  != 0)
          return x%y + y;
      else
          return x%y;
  }
```

只要x不被y整除，那么Arith_floor就是Arith_div，而Arith_ceiling就是Arith_div+1：

```
⟨arith.c functions 19⟩+≡
  int Arith_mod(int x, int y) {
```

20

```
    return Arith_div(x, y);
}

int Arith_ceiling(int x, int y) {
    return Arith_div(x, y) + (x%y != 0);
}
```

2.3 抽象数据类型

抽象数据类型（abstract data type，ADT）是一个定义了数据类型以及基于该类型值提供的各种操作的接口。一个数据类型就是一组值。在C语言中，内嵌的数据类型包括字符、整数、浮点数等等。结构本身就定义了一种新的类型并且可以用来形成更高级的类型，例如表、树、查找表等等。

一个高级类型是抽象的，因为接口隐藏了它的表示细节，只说明了针对该类型值的合法操作。理想情况下，这些操作并不揭示表示细节，以免客户调用程序依赖这些细节。一个抽象数据类型（或ADT）的规范化例子是堆栈。它的接口定义了该类型及其五种操作：

```
⟨initial version of stack.h⟩≡
  #ifndef STACK_INCLUDED
  #define STACK_INCLUDED

  typedef struct Stack_T *Stack_T;

  extern Stack_T Stack_new  (void);
  extern int     Stack_empty(Stack_T stk);
  extern void    Stack_push (Stack_T stk, void *x);
  extern void   *Stack_pop  (Stack_T stk);
  extern void    Stack_free (Stack_T *stk);

  #endif
```

typedef定义了Stack_T类型，该类型是一个指向有相同名字标签的结构。这个定义是合法的，因为结构、共用体和枚举标签使用一个名字空间，该名字空间是从变量、函数和类型名空间中分隔出来的，这种用法将贯穿本书。类型名Stack_T是该接口所关心的名字；这个标签名也许只对实现来说才是重要的。使用相同的名字可以避免代码中出现过多的变量名，而这些变量名确实用得很少。 21

宏STACK_INCLUDED也破坏了名字空间定义惯例，但是_INCLUDED后缀有助于避免冲突。另一个常用的约定就是在这类名字之前加下划线作为前缀，例如_STACK或_STACK_INCLUDED。然而，标准C保留了前下划线以便于实现和将来的扩展，因此使用前下划线一定要谨慎。

这个接口说明，堆栈是用指向结构的指针来表示的，但是它并没有给出那些结构的内容。Stack_T是一个隐式的指针类型，客户调用程序可以自由地操作这样的指针，但是不能间接引用它们，也就是说，客户调用程序不能查看出该类型指针指向的结构的内部内容，只有实

现具有这个特权。

隐式指针隐藏了表示的细节而且有利于捕获错误。只有Stack_T能被传递给上面的函数；试图传递其他类型的指针，例如指向其他结构的指针，都会产生编译错误。惟一的例外情况是空指针，它可以传给任何类型的指针。

条件编译指令#ifdef和#endif，以及STACK_INCLUDED的#define，使得stack.h可以被包含多次，这种情况会在接口导入其他接口时出现。如果没有这种保护，第二次和接下来的包含都将引起类型定义中Stack_T重复定义的编译错误。

在少数几个可供选择的约定中，这种约定可能是最好的。禁止接口包含其他的接口可以完全避免重复包含的情况，但是这就要求接口必须采用别的方法将其他接口导入，例如在注释中导入，并强迫程序员提供包含。将条件编译指令放在客户调用程序中而不是接口中，这样可以避免不必要的接口访问，但这样做会使得这些指令出现在多处，而不像以前，只出现在接口中。上面讲述的方法将使得编译程序做更多的工作。

按照这种约定，给出了抽象数据类型定义的接口X将被命名为X_T。本书中的接口更进
22 一步加强这个约定，使用宏将X_T在接口内简写为T。使用这个约定，stack.h就变成：

```
⟨stack.h⟩≡
  #ifndef STACK_INCLUDED
  #define STACK_INCLUDED

  #define T Stack_T
  typedef struct T *T;

  extern T     Stack_new  (void);
  extern int   Stack_empty(T stk);
  extern void  Stack_push (T stk, void *x);
  extern void *Stack_pop  (T stk);
  extern void  Stack_free (T *stk);

  #undef T
  #endif
```

语义上这个接口同前一个接口是等价的。简写仅仅使得接口更容易阅读；T指的是接口的一个基本类型。然而客户调用程序必须使用Stack_T，因为stack.h末尾的#undef指令取消了这个简写。

这个接口提供了一个任意指针的无边界堆栈。Stack_new产生一个新的堆栈，返回一个T类型的值，该值可以传递给其他4个函数。Stack_push将指针入栈，而Stack_pop移出并返回栈顶的指针，栈为空时Stack_empty返回1，否则返回0。Stack_free接受指向T的一个指针作为参数，释放该指针指向的堆栈，并将类型T的变量设为空指针。这种设计可以避免出现悬空指针，即指向已释放空间的指针。例如，如果names定义和初始化如下

```
#include "stack.h"
Stack_T names = Stack_new();
```

语句

```
Stack_free(&names);
```

就释放了分配给names的堆栈并将names设为空指针。 23

当一个抽象数据类型用隐式指针表示时，输出的类型也是一个指针类型，这就是为什么Stack_T要定义为一个指向结构Stack_T的指针的原因。本书中大多数的抽象数据类型都使用类似的定义。如果一个抽象数据类型给出了它的表示，并导出了通过值接收和返回该结构的函数，那么它必须定义该接口类型为输出的类型。第16章中的接口Text说明了这个约定，该接口声明Text_T是struct Text_T的类型定义。总之，T是接口中基本类型的缩写。

2.4 客户调用程序的责任

接口是其实现与客户调用程序之间的一种契约。实现必须提供接口声明的功能，而客户调用程序必须按照接口中描述的那些显式或隐式的规则来使用这些功能。程序设计语言提供了一些隐式规则来管理接口中声明的类型、函数以及变量。例如，C中的类型检查规则就可以捕捉到类型错误以及接口函数的参数数目方面的错误。

对于那些C用法中没有说明或不在C编译器检查范围的规则，就必须在接口中予以说明。客户调用程序必须遵守这些规则，而实现也必须实现这些规则。接口通常说明的是不可检查的运行期错误（unchecked runtime error）、可检查的运行期错误（checked runtime error）以及异常（exception）情况。不可检查和可检查的运行期错误不是客户调用程序造成的，例如打开文件错误等等。运行期错误也是客户调用程序和实现之间契约的一部分，是不可恢复的程序错误。异常是一些可能发生但很少会发生的情况，程序可以从异常中恢复，我们将在第4章对其进行详细讲解。

不可检查的运行期错误不要求实现能够检查出来。如果发生了不可检查的运行期错误，程序可能继续执行，但是结果是不可预测的，也不可重复。好的接口应尽可能地避免出现不可检查的运行期错误，同时必须说明那些可能发生的不可检查运行期错误。例如Arith就必须说明被0除是个不可检查的运行期错误。Arith可以检查除数是否为0，但将其作为一个不可检查的运行期错误，这样可以让它的函数模拟C中内嵌的除法操作行为，C中内嵌的除法操作就没有要求检查除0的情况。用0做除数的另一个合理解决方法就是将其作为可检查的运行期错误。 24

可检查的运行期错误是实现保证能够检查出来的。这些错误说明，客户调用程序没有遵守某部分约定；避免这些错误是客户调用程序的责任。Stack接口就指出了三种可检查的运行期错误：

（1）向接口中的例程传递空Stack_T；

（2）向传给Stack_free的Stack_T传递一个空指针；

（3）或将一个空栈传给Stack_pop。

接口也可以指定异常以及引发异常的条件。正如第4章中解释的，客户调用程序可以处理异常并采取正确的动作，未处理的异常将被当作可检查的运行错误。接口通常列出由它及其导入的接口所引发的异常。例如，Stack接口导入了Mem接口，使用它来分配空间，因此，它声明Stack_new和Stack_push会引发异常Mem_Failed。本书中大多数接口都说明了类似的可检查运行期错误和异常。

有了这些Stack接口的附加信息，我们就可以来看看它的实现了：

```
⟨stack.c⟩≡
  #include <stddef.h>
  #include "assert.h"
  #include "mem.h"
  #include "stack.h"

  #define T Stack_T
  ⟨types 25⟩
  ⟨functions 26⟩
```

#define指令重新将T实例化作为Stack_T的简写。实现揭示了Stack_T的内容，其内部是个结构，该结构有个字段指向一个栈内指针的链表以及一个这些指针的计数。

```
⟨types 25⟩≡
  struct T {
      int count;
      struct elem {
          void *x;
          struct elem *link;
      } *head;
  };
```

Stack_new分配并初始化一个新的T：

```
⟨functions 26⟩≡
  T Stack_new(void) {
      T stk;

      NEW(stk);
      stk->count = 0;
      stk->head = NULL;
      return stk;
  }
```

NEW是Mem接口中的一个分配宏指令。NEW（p）将分配该结构的一个实例，并将其指针赋给p，因此，在Stack_new中使用它就可以分配一个新的Stack_T结构。

当count为0时Stack_empty返回1，否则返回0：

```
⟨functions 26⟩+≡
  int Stack_empty(T stk) {
      assert(stk);
      return stk->count == 0;
  }
```

assert(stk)实现了可检查的运行期错误，它禁止空指针传给Stack中的任何函数。assert(*e*)是一个断言，它确保表达式*e*不为0。如果*e*不为0，那么它什么也不做，否则将停止程序的运行。assert是标准库函数的一部分，但是第4章中的Assert接口定义了它自己的具有类似语义的assert函数，并提供了平稳的程序终止。assert用来检查所有的可检查运行期错误。

Stack_push和Stack_pop从stk->head所指向的链表的头部添加或移出元素：

26

```
⟨functions 26⟩+≡
  void Stack_push(T stk, void *x) {
       struct elem *t;

       assert(stk);
       NEW(t);
       t->x = x;
       t->link = stk->head;
       stk->head = t;
       stk->count++;
  }

void *Stack_pop(T stk) {
       void *x;
       struct elem *t;

       assert(stk);
       assert(stk->count > 0);
       t = stk->head;
       stk->head = t->link;
       stk->count--;
       x = t->x;
       FREE(t);
       return x;
  }
```

FREE是Mem的释放宏指令；它释放指针参数所指向的空间，然后将参数设为空指针，这同Stack_free所考虑的一样——避免出现悬空指针。Stack_free也调用FREE：

```
⟨functions 26⟩+≡
  void Stack_free(T *stk) {
       struct elem *t, *u;

       assert(stk && *stk);
       for (t = (*stk)->head; t; t = u) {
            u = t->link;
            FREE(t);
       }
       FREE(*stk);
  }
```

27

此实现展示了本书中所有的ADT接口都会遇到的一个不可检查运行期错误，故未给出说明。我们没有方法保证传给Stack_push、Stack_pop、Stack_empty的Stack_T和传给

Stack_free的Stack_T*是由Stack_new返回的有效的Stack_T。练习2.3探讨了针对这个问题的部分解决办法。

还有两个不可检查的运行期错误，它们的影响可能更细微。本书中的许多ADT都会遇到空指针；也就是说，它们存储并返回空指针。在任何ADT中存储一个函数指针，即指向一个函数的指针，就是一个不可检查的运行期错误。空指针是一个普通的指针，一个void*类型的变量可以存储指向任意对象的指针，包括预定义类型、结构以及指针。函数指针则不同，虽然许多C编译器允许将函数指针赋值为一个空指针，但是并不保证空指针可以存储函数指针。

任何对象指针可以通过空指针进行传播，且不丢失任何信息。例如，执行下列语句之后：

```
S *p, *q;
void *t;
...
t = p;
q = t;
```

对任何非函数类型S，p和q是相等的。但是，空指针的使用不能破坏类型系统。例如，在执行下列语句之后：

```
S *p;
D *q;
void *t;
...
t = p;
q = t;
```

q不一定等于p。由于类型S和D的边界调整限制，q可能是一个指向类型D的某个对象的空指
28 针。在标准C中，空指针和字符指针有相同的大小和表示，但是其他的指针可能会小一些或有不同的表示。因此，在ADT中存储一个指向S的指针，但是将它取出赋给类型D的变量，而S和D又是不同的对象类型，这就会导致一个不可检查的运行期错误。

当ADT函数不会修改指针所指对象时，声明一个常量型隐式指针参数是很有可能的。例如，Stack_empty可以写成以下的形式：

```
int Stack_empty(const T stk) {
    assert(stk);
    return stk->count == 0;
}
```

但这样使用const（常量定义）是不正确的。此处的目的是把stk声明成一个“指向不可改变的struct T的指针”，因为Stack_empty并不修改*stk。但是const T stk声明stk为一个“指向struct T的常量指针”，也就是说T的类型定义将struct T*包装成一个单一类型，而该整个类型的操作数是一个常数。const T stk对Stack_empty和它的调用者来说是没有用的，因为所有的标量（包括指针）在C中都是通过值传递的。Stack_empty不可能改变调用者实参的值，不管参数是带还是不带const限制符。

通过用结构T*代替T，可以避免这个问题：

```
int Stack_empty(const struct T *stk) {
    assert(stk);
    return stk->count == 0;
}
```

这种使用方法说明了为什么const不应该用在指向ADT的指针中：它暴露了部分实现，进而限制了采用其他可行方法的可能性。对Stack的这个实现来说，使用const不会有什么问题，但是这样做排除了其他可行的方法。假设为了重复使用栈元素，需要推迟释放栈元素，在调用Stack_empty时才释放它们。这个Stack_empty的实现就需要修改*stk，但是它不能进行修改，因为stk被声明为const。本书中ADT均不使用const。

29

2.5 效率

本书中大部分接口的实现所使用的算法和数据结构，其平均运行时间都与N成线性关系或更低（其中N是接口输入的大小），而且大部分都可以处理大型的输入。对不能处理大型输入的接口或者那些将性能作为一个很重要的考虑因素的接口都指定了一个性能标准（performance criteria）。实现必须达到这些标准，而客户调用程序可以期望其性能与指定的标准一样好，但不要指望更好的性能。

本书中的所有接口采用的是简单的但有效的算法。当N很大时更复杂的算法和数据结构可能会有更好的性能，但是N通常都很小。因此大多数实现都使用了基本的数据结构像数组、链表、散列表、树以及这些结构的组合。

本书中除了少数几个ADT，其他的类型定义都使用了隐式指针，因此像Stack_empty这样的函数都可以用于访问被实现隐藏了的字段。由于是调用函数而不是直接访问字段，所以对实际应用性能的影响通常是可以忽略不计的。针对可靠性及尽可能地找到运行期错误方面的改进措施都是值得考虑的，其价值远超过在性能上的改进所能带来的微小价值。

如果客观的度量表明性能的改进确实是必要的，那么它们就应该在不改变接口的前提下进行改进，例如通过定义宏指令。当这种方法不可行时，最好创建一个新的接口，并说明它的性能比修改一个已存在的接口更有效，修改一个已存在的接口会使得所有调用它的客户调用程序都变成无效的客户调用程序。

参考书目浅析

过程和函数库的重要性从1950年代就被认识到了。Parnas（1972）就是一篇关于如何将一个程序划分成多个模块的经典文章。这篇论文已有20多年了，然而它却陈述了程序员今天还要面对的问题。

C程序员每天都在使用接口：C语言库就是一个拥有15个接口的集合。标准I/O接口——stdio.h，定义了一个ADT——FILE以及对FILE指针的各项操作。Plauger（1992）给出了这15个接口

30

以及其匹配的实现的详细说明，和本书中介绍的一系列接口和实现所用的方法很相似。

Modula-3是一门相对比较新的程序设计语言，它在语言上支持接口和实现的分离，并且它是本书中使用的基于接口技术的起源（Nelson 1991）。不可检查和可检查的运行期错误的概念以及ADT中T的概念都来自Modula-3。Harbison（1992）是一本介绍Modula-3的教科书。Horning et al.（1993）描述了Modula-3系统中的核心接口。本书中的部分接口就是根据那些接口改写的。Roberts（1995）的书将基于接口的设计作为计算机课程介绍性教学的组织原则。

断言的重要性也是被广泛承认的，一些程序设计语言，像Modula-3和Eiffel（Meyer 1992）都在语言中嵌入了断言机制。Maguire（1993）用了一整章的篇幅来介绍C语言程序设计中断言的使用。

很多熟悉面向对象程序设计的程序员可能会认为：本书中的大多数ADT都可以（也许会更好）作为面向对象程序设计语言中的对象来提出，像C++（Ellis和Stroustrup 1990）和Modula-3。Budd（1991）是一本关于面向对象程序设计方法以及一些面向对象程序设计语言（包括C++）的介绍指南。本书中说明的接口设计原则对面向对象程序设计语言同样适用。例如，用C++重写本书中的ADT，对从C转换到C++的程序员来说，就是一个很有用的练习。

C++标准模板库（STL）提供了类似于本书中所描写的ADT。STL充分利用了C++模板，用具体的类型将ADT实例化（Musser和Saini 1996）。例如，STL提供了一个向量（vector）数据类型的模板，它可以用于实例化整数向量、字符串向量等等。STL同时也提供了一套函数，用于操作由模板产生的数据类型。

练习

2.1 预处理宏指令和条件编译指令，像#if，可以用于指定Arith_div和Arith_mod中除法是如何取整的。解释为什么-13/5 == -2是执行测试的一个较好的方法。

2.2 在Arith_div和Arith_mod中使用的测试-13/5 == -2只有在arith.c的编译器采用和调用Arith_div和Arith_mod时一样的工作方式进行算术运算时才有效。但是这个条件可能并不成立，例如用运行在机器X上的交叉编译器来编译arith.c，并生成机器Y的代码。不使用条件编译指令修改arith.c，以使得上述用交叉编译生成的代码也能正常工作。

31

2.3 与本书中所有的ADT一样，Stack接口省略了这样的说明“传递一个外部Stack_T到接口中的任何
例程是一个不可检查的运行期错误”。一个外部Stack_T是指它不是由Stack_new产生的。试修改stack.c使得它可以检查这种错误的发生。例如，一种方法就是在Stack_T的结构中增加一个字段，保存一个由Stack_new返回的对每个Stack_T是惟一的位模式。

2.4 通常，检查某种无效指针是可能的。例如，如果一个非空指针指向一个客户调用程序地址空间以外的地址，那么它就是无效的，而且指针通常要满足边界对齐限制；

例如，在某些系统上一个指向双精度数的指针必须是8的倍数。设计一个系统专用的宏指令isBadPtr(*p*)，使得当*p*是个无效指针时，assert(*ptr*)语句可以被类似assert(!isBadPtr(*ptr*))这样的断言来取代。

2.5 对堆栈有许多可行的接口，设计并实现几个其他可行的Stack接口。例如，一种方法是指定一个最大的尺寸作为Stack_new的参数。 32

第3章 原 子

原子是一个指向惟一的、不可改变的0个或任意多个字节序列的指针。大多数原子都是指向以空字符结束的字符串，但是任何一个指向任意字节序列的指针都可以是原子。任何原子都只能出现一次，这就是为什么它被称作原子的原因。如果两个原子指向同一个内存单元时，则这两个原子是相等的。仅仅比较两个字节序列相应的指针是否相等，就可以判断这两个字节序列是否相等了，这是使用原子的好处之一。另一个好处就是使用原子可以节省空间，因为每个序列只会出现一次。

原子通常被当作数据结构中的关键字使用，这些数据结构是用任意字节序列，而不是用整数来索引的。第8和第9章中叙述的表和集合就是这方面的例子。

3.1 接口

Atom接口很简单：

⟨*atom.h*⟩≡

```
#ifndef ATOM_INCLUDED
#define ATOM_INCLUDED

extern       int   Atom_length(const char *str);
extern const char *Atom_new   (const char *str, int len);
extern const char *Atom_string(const char *str);
extern const char *Atom_int   (long n);

#endif
```


Atom_new接收一个指向字节序列的指针以及该序列的字节数作为输入。它在原子表中增加一个该序列的拷贝，并且如果需要的话，返回原子表中指向该拷贝的指针（即原子）。Atom_new永远不会返回一个空指针。一旦原子创建好了，那么它在客户调用程序的整个执行期间都存在。原子总是以一个空字符结束，在必要的时候该空字符由Atom_new添加。

Atom_string与Atom_new类似，也是字符串原子的常规使用方法。它接收一个以空字符结束的字符串作为输入，在原子表中增加一个该串的拷贝，如果需要的话，返回该原子。Atom_int返回长整数n的字符串表示的原子，这也是原子的另一种常规使用方法。最后，Atom_length返回其原子参数的长度。

将一个空指针传递给该接口中的任何一个函数、传递一个负的len值给Atom_new，或传递一个不是原子的指针给Atom_length，这些都是可检查的运行期错误；而试图修改原子所指向字节的内容，则是不可检查的运行期错误。Atom_length执行所花费的时间与原子的数目成比例。Atom_new、Atom_string以及Atom_int都可能会引发Mem_Failed异常。

3.2 实现

Atom的实现对原子表进行维护。Atom_new、Atom_string以及Atom_int查找原子表，并且都有可能在原子表中添加一个新的元素，而Atom_length仅仅查找原子表。

```
⟨atom.c⟩≡
  ⟨includes 34⟩
  ⟨macros 37⟩
  ⟨data 36⟩
  ⟨functions 35⟩

⟨includes 34⟩≡
  #include "atom.h"
```

Atom_string和Atom_int可以在不知道原子表的表示细节的情况下执行相应操作。例如Atom_string就仅仅调用Atom_new:

34

```
⟨functions 35⟩≡
  const char *Atom_string(const char *str) {
      assert(str);
      return Atom_new(str, strlen(str));
  }

⟨includes 34⟩+≡
  #include <string.h>
  #include "assert.h"
```

Atom_int首先把它的参数转换成一个字符串，然后调用Atom_new:

```
⟨functions 35⟩+≡
  const char *Atom_int(long n) {
      char str[43];
      char *s = str + sizeof str;
      unsigned long m;

      if (n == LONG_MIN)
          m = LONG_MAX + 1UL;
      else if (n < 0)
          m = -n;
      else
          m = n;
      do
          *--s = m%10 + '0';
      while ((m /= 10) > 0);
      if (n < 0)
          *--s = '-';
      return Atom_new(s, (str + sizeof str) - s);
  }

⟨includes 34⟩+≡
  #include <limits.h>
```

Atom_int必须处理二进制补码数的不对称范围以及C的除法和取余运算的不确定性。无符号的除法和取余都具有良好的定义，因此Atom_int也可以通过使用无符号算术来避免使用有符号运算引起的不确定性。

35

最小的负长整数的绝对值不可能被表示出来，因为在二进制补码系统中，负数比正数多一个。因此在Atom_new将把它参数的绝对值赋值给无符号长整数m之前，它必须先检测这个异常情况。LONG_MAX的值存在标准头文件limits.h中。

loop循环从左到右形成m的十进制字符串表示；它计算最右边的数字，用10去除m，并且一直继续至做到m值变为0。计算出的每一个数字，将被存入--s中，s在str中是向后移动的。如果n是负数，那么会在字符串的开始处存一个负号。

转换完成以后，s指向所表示的字符串，这个字符串有&str[43]-s个字符。str有43个字符，这对任何机器上的任何整数的十进制表示来说都足够了。例如，我们假定长整数的长度是128位，对任何128位的八进制（以8为底数）有符号整数的字符串表示需要128/3+1=43个字符。十进制表示的数字不会比八进制的表示需要的字符数多，因此43个字符足够了。

str定义中的43是一个“魔术数”（magic number），通常有更好的方式来定义这类值的符号名，以确保同样的值能够在各处使用。然而，这儿的值仅出现一次，在任何需要使用该值的地方将使用sizeof来代替。定义一个符号化的名字可使代码更易阅读，但是这会使得代码更长，使名字空间变得混乱。在本书中，仅在某个值出现一次以上或者该值是接口的一部分时才定义符号化的名字。散列表的长度buckets小于2048，就是遵循这种习惯的另一个例子。

散列表显然是一个针对原子表的数据结构，散列表是一个入口表的指针数组，其中的每一个元素都存有一个原子：

```
⟨data 36⟩≡
  static struct atom {
      struct atom *link;
      int len;
      char *str;
  } *buckets[2048];
```

buckets[i]中的链表存储着散列值为i的原子。入口的link指向表中的下一个入口、len存储序列的长度、str指向序列本身。例如，在字长为32、字符长度为8的小尾数法计算机上，Atom_string(“an atom”)分配一个如图3-1所示的struct atom，下划线代表一个空格。每一个入口都有足够大的空间存储它的序列，图3-2显示了散列表的全部结构。

36

Atom_new计算由str[0..len-1]（如果len为0时，是一个空序列）给定序列的散列值，并用buckets的元素个数对其取模，搜索由buckets中该散列值元素所指向的链表。如果发现str[0..len-1]已存在于表中，它将只是简单地返回该原子：

```
⟨functions 35⟩+≡
  const char *Atom_new(const char *str, int len) {
      unsigned long h;
      int i;
```

```
    struct atom *p;

    assert(str);
    assert(len >= 0);
    ⟨h ← hash str[0..len-1] 39⟩
    h &= NELEMS(buckets)-1;
    for (p = buckets[h]; p; p = p->link)
        if (len == p->len) {
            for (i = 0; i < len && p->str[i] == str[i]; )
                i++;
            if (i == len)
                return p->str;
        }
    ⟨allocate a new entry 39⟩
    return p->str;
}

⟨macros 37⟩≡
  #define NELEMS(x) ((sizeof (x))/(sizeof ((x)[0])))
```

NELEMS的定义说明了C的一个常用习惯：数组中元素个数等于数组的大小除以每个元素的大小。sizeof是一个编译阶段的操作，故该计算只能在编译时已经知道了数组的大小时才可以使用。如定义中所演示的，在宏内使用的宏参数都用斜体高亮显示。

37

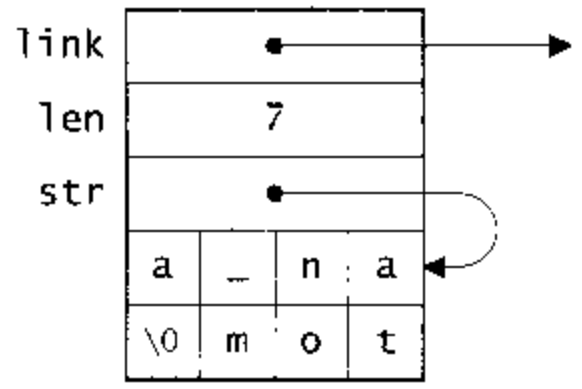


图3-1 针对 "an atom" 的struct atom的小尾数法布局

如果str[0..len-1]不在表中，Atom_new将分配一个struct atom和足够的附加空间以存储序列，将其添加到表中，并将str[0..len-1]拷贝到附加空间中，添加新的入口到buckets[h]中链表的表头。入口可以被附加到链表的表尾，但是将其加到链表的表头更简单一些。

```
⟨allocate a new entry 39⟩≡
  p = ALLOC(sizeof (*p) + len + 1);
  p->len = len;
  p->str = (char *)(p + 1);
  if (len > 0)
      memcpy(p->str, str, len);
  p->str[len] = '\0';
  p->link = buckets[h];
  buckets[h] = p;

⟨includes 34⟩+≡
  #include "mem.h"
```

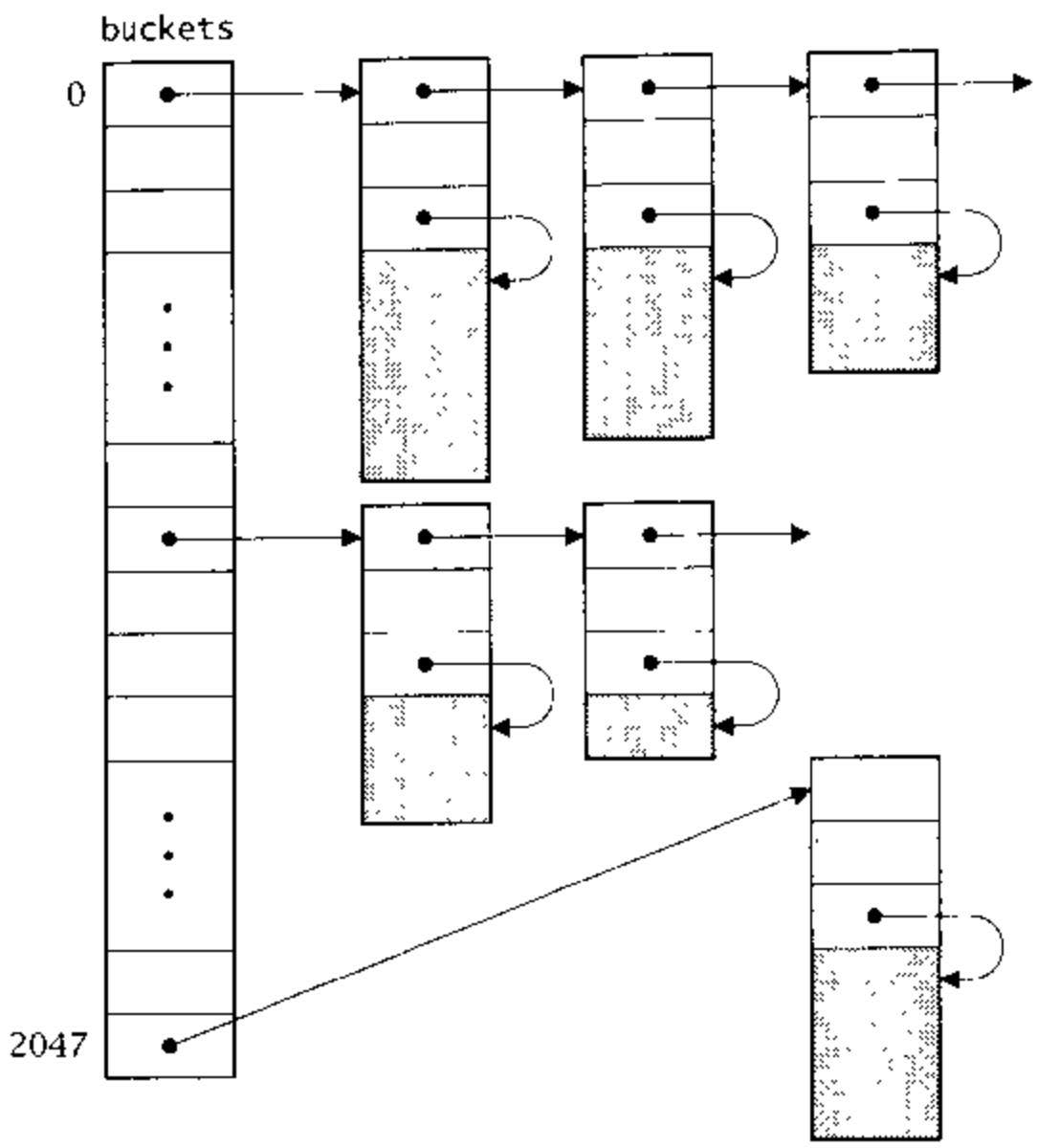


图3-2　Hash表结构

ALLOC是Mem的基本分配函数，它模拟标准库中的函数malloc，其参数是请求的字节数。Atom_new不能使用Mem中的NEW函数（在Stack_push中说明过），因为其字节的长度依赖于len，而NEW仅用在字节数在编译时就已知的情况下才可以使用。上面的ALLOC调用为atom结构和序列分配空间，序列被立即存入随后的字节中。

传给Atom_new的序列的散列操作包括计算代表序列的一个无符号数。理论上，对于N个序列，这些散列值应该均匀分布在0到NELEMS(buckets)-1之间。如果是这样分布的，那么buckets中的每一个链表都有N/NELEMS(buckets)个元素，且搜索一个序列的平均时间将是N/2 · NELEMS(buckets)。如果N小于2 · NELEMS(buckets)，那么搜索时间接近常数。

散列操作是一个值得好好研究的课题，现在也已经有许多很好的散列函数。Atom_new使用一个简单的查找表算法：

```
⟨h ← hash str[0..len-1] 39⟩≡
  for (h = 0, i = 0; i < len; i++)
      h = (h<<1) + scatter[(unsigned char)str[i]];
```

38
39

scatter是一个256人口的数组，它将字节映射为随机数。这些随机数通过调用标准库中的函数rand生成。实验表明，这种简单的方法有助于生成分布更加均匀的散列值。将str[i]转换为无符号字符能够避免C中关于“无格式（plain）”字符的不确定性：它们既可以是有符号的，也可以是无符号的。如果没有这种转换，在使用有符号字符的机器上，值超过127的str[i]将

变为负数。

```
⟨data 36⟩+≡
  static unsigned long scatter[] = {
  2078917053, 143302914, 1027100827, 1953210302, 755253631, 2002600785,
  1405390230, 45248011, 1099951567, 433832350, 2018585307, 438263339,
  813528929, 1703199216, 618906479, 573714703, 766270699, 275680090,
  1510320440, 1583583926, 1723401032, 1965443329, 1098183682, 1636505764,
  980071615, 1011597961, 643279273, 1315461275, 157584038, 1069844923,
  471560540, 89017443, 1213147837, 1498661368, 2042227746, 1968401469,
  1353778505, 1300134328, 2013649480, 306246424, 1733966678, 1884751139,
  744509763, 400011959, 1440466707, 1363416242, 973726663, 59253759,
  1639096332, 336563455, 1642837685, 1215013716, 154523136, 593537720,
  704035832, 1134594751, 1605135681, 1347315106, 302572379, 1762719719,
  269676381, 774132919, 1851737163, 1482824219, 125310639, 1746481261,
  1303742040, 1479089144, 899131941, 1169907872, 1785335569, 485614972,
  907175364, 382361684, 885626931, 200158423, 1745777927, 1859353594,
  259412182, 1237390611, 48433401, 1902249868, 304920680, 202956538,
  348303940, 1008956512, 1337551289, 1953439621, 208787970, 1640123668,
  1568675693, 478464352, 266772940, 1272929208, 1961288571, 392083579,
  871926821, 1117546963, 1871172724, 1771058762, 139971187, 1509024645,
  109190086, 1047146551, 1891386329, 994817018, 1247304975, 1489680608,
  706686964, 1506717157, 579587572, 755120366, 1261483377, 884508252,
  958076904, 1609787317, 1893464764, 148144545, 1415743291, 2102252735,
  1788268214, 836935336, 433233439, 2055041154, 2109864544, 247038362,
  299641085, 834307717, 1364585325, 23330161, 457882831, 1504556512,
  1532354806, 567072918, 404219416, 1276257488, 1561889936, 1651524391,
  618454448, 121093252, 1010757900, 1198042020, 876213618, 124757630,
  2082550272, 1834290522, 1734544947, 1828531389, 1982435068, 1002804590,
  1783300476, 1623219634, 1839739926, 69050267, 1530777140, 1802120822,
  316088629, 1830418225, 488944891, 1680673954, 1853748387, 946827723,
  1037746818, 1238619545, 1513900641, 1441966234, 367393385, 928306929,
  946006977, 985847834, 1049400181, 1956764878, 36406206, 1925613800,
  2081522508, 2118956479, 1612420674, 1668583807, 1800004220, 1447372094,
40 523904750, 1435821048, 923108080, 216161028, 1504871315, 306401572,
  2018281851, 1820959944, 2136819798, 359743094, 2354150250, 1843084537,
  1306570817, 244413420, 934220434, 672987810, 1686379655, 1301613820,
  1601294739, 484902984, 139978006, 503211273, 294184214, 176384212,
  281341425, 228223074, 147857043, 1893762099, 1896806882, 1947861263,
  1193650546, 273227984, 1236198663, 2116758626, 489389012, 593586330,
  275676551, 360187215, 267062626, 265012701, 719930310, 1621212876,
  2108097238, 2026501127, 1865626297, 894834024, 552005290, 1404522304,
  48964196, 5816381, 1889425288, 188942202, 509027654, 36125855,
  365326415, 790369079, 264348929, 513183458, 536647531, 13672163,
  313561074, 1730298077, 286900147, 1549759737, 1699573055, 776289160,
  2143346068, 1975249606, 1136476375, 262925046, 92778659, 1856406685,
  1884137923, 53392249, 1735424165, 1602280572
  };
```

Atom_length不能对其参数进行散列操作，因为它不知道参数的长度。但是该参数肯定是原子，因此Atom_length仅仅对buckets中链表的指针进行来回的比较就可以了。如果它找到了原子，将返回原子的长度：

```
⟨functions 35⟩+≡
  int Atom_length(const char *str) {
      struct atom *p;
      int i;

      assert(str);
      for (i = 0; i < NELEMS(buckets); i++)
          for (p = buckets[i]; p; p = p->link)
              if (p->str == str)
                  return p->len;
      assert(0);
      return 0;
  }
```

assert(0)实现了可检查的运行期错误，Atom_length必须在参数为原子而不是字符串指针时调用。assert(0)也用于标识发生了不期望发生的情况，即所谓的“不可发生”的情况。 41

参考书目浅析

原子已在LISP语言中使用了很长的时间，LISP也是原子这个名字的发源地；原子也一直用在字符处理（string-manipulation）语言中，比如SNOBOL4，这种语言实现了与这一章（Griswold 1972）中描述的几乎完全相同的字符串。C编译器lcc（Fraser 和 Hanson 1995）中有一个类似于Atom的模块，是Atom的实现的早期产品。lcc为出现在源程序中的所有标识符和常量在一个单一表中存储一个字符串，并且从不释放它们。这样做不会消耗太大的存储空间，因为在C程序中，无论源程序有多大，不同字符串的数量都是相当少的。

Sedgewick（1990）和Knuth（1973b）详细描述了散列，并为如何编写一个好的散列函数给了些指导。Atom（和lcc）中使用的散列函数是就是由Hans Boehm提出的。

练习

3.1 大多数文章建议对buckets的大小使用一个素数。使用素数和一个好的散列函数，通常能够得到buckets中链表的长度的一个很好的分布。Atom使用了2的幂数，这种做法有时被当作不好的例子加以引用。编写一个生成或读取比如说10 000个典型字符串的程序，并测试Atom_new的速度和链表长度的分布状况。然后修改buckets，使它具有2 039（小于2 048的最大素数）个入口，将语句

```
h &= NELEMS(buckets)-1;
```

改为

```
h %= NELEMS(buckets);
```

并且重复测试。使用素数起到作用了吗？对于你的特定机器，你的结果是多少？

3.2 翻阅一些关于更好的散列函数的文章，例如Knuth（1973b），还有关于算法和数据结构的类似文章及其参考文献以及关于编译器的文章，像Aho、Sethi和Ullman（1986）。试试这些函数，并测试它们的优点。 42

3.3 解释为什么Atom_new不使用标准C库函数strncmp比较序列。

3.4 声明原子结构的另一种方法如下：

```
struct atom {
    struct atom *link;
    int len;
    char str[1];
};
```

对len字节的字符串，我们使用ALLOC(sizeof(*p) + len)分配一个struct atom，该操作将为link和len以及能存储len + 1个字节的str分配空间。这种方法避免了由于将str声明为指针所间接导致的额外时间和空间请求。不幸的是，这种方式违背了C标准，因为客户调用程序访问的字节地址越过了str[0]，且这种访问的影响没被说明。实现这种方法，并测试间接开销。为了节约一点空间而违反C标准，这值得吗？

3.5 Atom_new将struct atom的len字段同输入序列的长度做了比较，以避免比较不同长度的字符序列。如果每一个原子的散列值（非buckets的编号）也存入struct atom中，那么彼此也能进行比较。实现这种“改进”，并测试其优点。这样做值得吗？

3.6 Atom_length的运行时间比较慢。试修改Atom的实现以使得Atom_length的运行时间与Atom_new的大致相同。

3.7 因为Atom接口的函数是客户调用程序最常用的一类函数，所以它发展为现在的这种形式。本练习以及随后的练习探讨的其他函数和设计也可能很有用。请实现：

43

```
extern void Atom_init(int hint);
```

这里hint估算了客户调用程序所希望创建的原子个数。你应该添加什么样的可检查的运行期错误以限制Atom_init的调用时机？

3.8 有几个释放原子的函数，它们应该是Atom的扩展接口应该提供的。例如，函数

```
extern void Atom_free (char *str);
extern void Atom_reset(void);
```

可以分别释放出str给定的原子和所有的原子。实现这些函数，别忘了说明和实现相应的可检查运行期错误。

3.9 有些客户调用程序希望在运行开始时就装入一些字符串作为原子供以后使用。请实现：

```
extern void Atom_vload(const char *str, ...);
extern void Atom_aload(const char *strs[]);
```

Atom_vload装入了可变数目的参数列表中给出的所有字符串，直到碰到一个空指针；Atom_aload对以空字符为结束符的字符串指针数组实现了同样的功能。

3.10 如果客户调用程序保证不释放字符串，那么就可以不用拷贝字符串，这对字符常量来说非常普遍。请实现：

```
extern const char *Atom_add(const char *str, int len);
```

它的功能类似于Atom_new，但没有拷贝字符序列。如果你提供了Atom_add和Atom_free（以及练习3.8中的Atom_reset），那么还需要说明和实现哪些可检查的运行时错误？

44

第4章　异常与断言

程序中通常会出现三种错误：用户错误、运行期错误以及异常。用户错误是在预料中的，因为它们可能是因为用户不正确的输入引起的。命名一个并不存在的文件，或是在电子表格中输入非正规的数据，或是提交给编译器的源程序本身就有语法错误，等等，都属于用户错误。程序必须计划并处理这类错误。通常，函数必须处理用户错误并返回错误代码，这些错误是计算过程中很正常的一部分。

在前面章节中提到的可检查的运行期错误是另一种类型错误。它们不是用户错误。根本不在预料中，并且总是揭示了程序的漏洞。因此，遇到这类错误，应用程序将无法恢复；但它们必须要合理地结束。本书中的实现使用断言来捕获这类错误，断言的处理将在4.3节中给出。断言总是会导致程序终止，终止的方式可能由机器或是应用程序决定。

异常是介于用户错误和程序错误之间的一类错误。异常是很少出现且可能不可预测的错误，但是从异常中恢复是可能的。某些异常反应了机器的性能，例如算术溢出、下溢和栈的溢出等等。其他异常显示了操作系统所检测的状态，这些状态也可以由用户来初始化，例如点击了“中断”键或写文件时发生写错误等，都属于这类错误。这类异常通常由UNIX系统的信号给出，并由信号处理器进行处理。当有限的资源耗尽时也会发生异常，例如当应用程序用尽了内存或是指定的电子表格太大等等。

异常并不会经常发生，因此可能会发生异常的函数通常不会返回错误代码；这样在少数情况下会造成代码的混乱，而多数情况下会给程序带来不确定性。如果应用程序产生的异常是可以恢复的，那么将会触发异常，并交由恢复代码处理。异常的活动范围是动态的：当发生某个异常的时候，它由最近实例化的处理程序来进行处理。将控制权转交给某个处理程序就像是非局部的goto语句，也就是说，处理程序可以在远离触发异常的某个例程中实例化。

某些程序设计语言嵌入了实例化处理程序和产生异常的工具。在C中，标准库函数setjmp和longjmp形成了结构化异常工具的基础。简单地说即setjmp实例化处理程序，而longjmp产生异常。

下面用一个例子进行详细说明。设函数allocate调用malloc来分配*n*个字节，并返回一个由malloc返回的指针。然而如果malloc返回的是空指针，这说明所需求的空间不能分配，那么allocate就要抛出Allocated_Failed异常。这个异常本身在标准头文件setjmp.h中被声明为jmp_buf：

```
#include <setjmp.h>

int Allocation_handled = 0;
jmp_buf Allocate_Failed;
```

除非实例化了某个处理程序，否则Allocation_handled为0，且allocate在抛出异常之前会先检查Allocation_handled:

```
void *allocate(unsigned n) {
    void *new = malloc(n);

    if (new)
        return new;
    if (Allocation_handled)
        longjmp(Allocate_Failed, 1);
    assert(0);
}
```

46

在分配失败而又没有实例化任何处理程序的时候，allocate用了一个断言实现了可检查的运行期错误。

处理程序是调用setjmp(Allocate_Failed)来实例化的，该调用会返回一个整数。setjmp最令人感兴趣的特征是它可以返回两次。调用setjmp时返回0，而在allocate中调用longjmp会引起第二次值的返回，该值由longjmp的第二个参数给出，这在上面的例子中就存在。因此，客户调用程序就可以通过测试setjmp的返回值来处理异常：

```
char *buf;
Allocation_handled = 1;
if (setjmp(Allocate_Failed)) {
    fprintf(stderr, "couldn't allocate the buffer\n");
    exit(EXIT_FAILURE);
}
buf = allocate(4096);
Allocation_handled = 0;
```

当setjmp返回0时，继续调用allocate。如果分配失败，allocate中的longjmp引起setjmp再一次返回，这次返回值为1，因此，继续调用fprint和exit。

这个例子并没有处理嵌套的程序，如果上面的代码调用函数makebuffer，就有可能出现嵌套的处理程序，因为makebuffer本身就实例化了一个处理程序并且调用了allocate。嵌套处理程序是必须提供的，因为客户调用程序可能不知道实现出于自己的目的也实例化了某个处理程序。而且Allocation_handled标志的使用也很麻烦，如果不在恰当的时候对它进行设置和清除就会引起混乱。在下一节中给出的Except接口将处理这些问题。

4.1 接口

Except接口在一系列宏指令和函数中包装了setjmp/longjmp，它们一起提供了一个结构化的异常处理工具。这个工具并不很完美，但是它避免了上面列出的错误，而使用异常的地方也用宏指令清楚地加以标识。

异常是Except_T类型的一个全局或静态变量：

47

```
⟨except.h⟩≡
  #ifndef EXCEPT_INCLUDED
  #define EXCEPT_INCLUDED
  #include <setjmp.h>

  #define T Except_T
  typedef struct T {
      char *reason;
  } T;

  ⟨exported types 53⟩
  ⟨exported variables 53⟩
  ⟨exported functions 48⟩
  ⟨exported macros 48⟩

  #undef T
  #endif
```

Except_T结构只有一个字段，它可以初始化为一个描述异常的字符串。当发生一个未处理的异常时，才会把该字符串打印出来。

异常处理程序处理的是异常的地址。异常必须是全局的或静态的变量，因此它们的地址惟一地标识了它们。如果把异常声明成一个局部变量或参数就会产生不可检查的运行期错误。

异常e由RAISE宏指令引发或由函数Except_raise引发：

```
⟨exported macros 48⟩≡
  #define RAISE(e) Except_raise(&(e), __FILE__, __LINE__)

⟨exported functions 48⟩≡
  void Except_raise(const T *e, const char *file,int line);
```

将空的e传给Except_raise是可检查的运行期错误。

处理程序是由TRY-EXCEPT和TRY-FINALLY语句来实例化的，这两个语句用宏指令实现。这两个语句可以处理嵌套异常，也可以管理异常状态的数据。TRY-EXCEPT语句的语法是：

```
TRY
    S
EXCEPT(e1)
    S1
EXCEPT(e2)
    S2
...
EXCEPT(en)
    Sn
ELSE
    S0
END_TRY
```

48

TRY-EXCEPT语句命名为e_1, e_2, ..., e_n, 的异常创建了处理程序，并执行语句S。如果S没有

触发异常，将跳过处理程序，继续执行END_TRY之后的语句。如果S产生了异常e，且e是e_1到e_n之间的一个，那么S的执行被中断，控制立即转给相对应的EXCEPT从句中的语句。随后拆除处理程序，执行EXCEPT从句中的处理语句S_i，然后继续执行END_TRY之后的语句。

如果S产生了一个不在e_1到e_n之间的异常，那么将终止处理程序，执行ELSE之后的语句，然后继续执行END_TRY之后的语句。ELSE从句是可选的。

如果S产生了一个不由任何S_i处理的异常，那么终止处理程序，且将异常传递给前一个执行TRY-EXCEPT或TRY-FINALLY语句所创建的处理程序。

TRY-END-TRY在句法上等价于一条语句：TRY标志一个新的作用域，该作用域在下一个EXCEPT、ELSE、FINALLY或END_TRY处结束。

为了说明这些宏指令的使用，我们把前一节结束时的例子改写一下。把Allocate_Failed变成一个异常，该异常是在malloc返回一个空指针时由allocate引发：

```
Except_T Allocate_Failed = { "Allocation failed" };

void *allocate(unsigned n) {
    void *new = malloc(n);

    if (new)
        return new;
49  RAISE(Allocate_Failed);
    assert(0);
}
```

如果客户调用程序代码想处理这个异常，那么它需要在TRY-EXCEPT语句内调用allocate：

```
extern Except_T Allocate_Failed;
char *buf;
TRY
    buf = allocate(4096);
EXCEPT(Allocate_Failed)
    fprintf(stderr, "couldn't allocate the buffer\n");
    exit(EXIT_FAILURE);
END_TRY;
```

TRY-EXCEPT语句是用setjmp和longjmp来实现的，因此标准C的使用这些函数的警告在TRY-EXCEPT中也同样适用。特别是，当S改变了某个自变量的值，而异常又使得程序继续执行某个处理程序语句S_i或继续执行END_TRY之后的语句时，那么这种改变就不存在了。例如，程序片段：

```
static Except_T e;
int i = 0;
TRY
    i++;
    RAISE(e);
EXCEPT(e)
    ;
```

```
END_TRY;
printf("%d\n", i);
```

可以打印出0或1．这决定于setjmp和longjmp的实现细节。在S中改变的自变量必须声明为volatile；例如，如果将i的声明改为

```
volatile int i = 0;
```

那么上面给出的例子将会打印出1。

TRY-FINALLY语句的语法是：

50

```
TRY
    S
FINALLY
    S1
END_TRY
```

如果S没有产生任何异常，那么执行S_1，然后继续执行END_TRY之后的语句。如果S产生了异常，那么S的执行被中断，控制立即转给S_1。S_1执行完后，引起S_1执行的异常重新产生，使得它可以由前一个实例化的处理程序来处理。注意S_1是在两种情况中都必须执行的。处理程序可以用RERAISE宏指令显式地重新产生异常：

```
⟨exported macros 48⟩+≡
  #define RERAISE Except_raise(Except_frame.exception, \
      Except_frame.file, Except_frame.line)
```

TRY-FINALLY语句等价于：

```
TRY
    S
ELSE
    S1
    RERAISE;
END_TRY;
S1
```

注意：不管是否产生了异常，S_1都要执行。

我们使用TRY-FINALLY语句的目的是给客户调用程序一个机会，在发生异常的时候来“清理”现场。例如：

```
FILE *fp = fopen(...);
char *buf;
TRY
    buf = allocate(4096);
    ...
FINALLY
    fclose(fp);
END_TRY;
```

51

不管分配成功或是失败都要关闭由fp打开的文件。如果分配失败，必须由其他处理程序来处理Allocate_Failed。

如果TRY-FINALLY语句中的S_1或是TRY-EXCEPT语句中的处理程序产生了异常，那么该异常也是由前面实例化的处理程序来处理。

简化语句：

```
TRY
    S
END_TRY
```

等价于：

```
TRY
    S
FINALLY
    ;
END_TRY
```

接口中的最后一个宏指令是

```
⟨exported macros 48⟩+≡
  #define RETURN switch (⟨pop 56⟩,0) default: return
```

RETURN宏指令用在TRY语句内部，用来替代return语句。在TRY-EXCEPT或TRY-FINALLY语句内部执行C的return语句是一个不可检查的运行期错误。如果TRY-EXCEPT或TRY-FINALLY中的语句必须要执行return，那么它们用这个宏指令代替通常C的return语句。在这个宏指令中还使用了switch语句，它可以将RETURN或是RETURN e扩展成句法上正确的C语句。*<pop 56>*的细节将在下一节中给出。

Except接口中的宏指令被公认为是很粗略的，甚至有些脆弱。接口中不可检查的运行期错误特别麻烦，很难发现。但它们对大多数应用来说都已经足够了，因为异常应该尽量地少用，
52 一般只在大型应用程序中少量使用。如果异常越来越多，这通常说明程序有更严重的设计错误。

4.2 实现

Except接口中的宏指令和函数一起维护了一个记录异常状态以及实例化处理程序结构的堆栈。结构中的字段env就是setjmp和longjmp使用的某个jmp_buf，因此这个堆栈可以处理嵌套的异常。

```
⟨exported types 53⟩≡
  typedef struct Except_Frame Except_Frame;
  struct Except_Frame {
      Except_Frame *prev;
      jmp_buf env;
      const char *file;
      int line;
      const T *exception;
  };

⟨exported variables 53⟩≡
  extern Except_Frame *Except_stack;
```

Except_stack指向异常栈顶的异常帧，每个帧的prev字段指向它的前一帧。就像前一节中RERAISE定义的那样，产生一个异常就是将异常的地址存在exception字段中，并分别在file和line字段中保存异常的附属信息——异常产生的文件以及行号。

TRY从句将一个新的Except_Frame压入异常栈，并调用setjmp。由RAISE和RERAISE调用Except_raise填充栈顶帧的字段exception、file和line，从异常栈中弹出栈顶Except_Frame，然后调用longjmp。EXCEPT从句检查该帧中的exception字段，决定应该用哪个处理程序。FINALLY从句执行清除代码，并重新产生已弹出的异常帧中存储的异常。

如果发生了异常却没有执行处理控制就达到了END_TRY从句，将会重新触发异常。

宏指令TRY、EXCEPT、ELSE、FINALLY和END_TRY一起将TRY-EXCEPT语句转化成如下形式的语句：

```
do {
    create and push an Except_Frame
    if (first return from setjmp) {
        S
    } else if (exception is e1) {
        S1
    ...
    } else if (exception is en) {
        Sn
    } else {
        S0
    }
    if (an exception occurred and wasn't handled)
        RERAISE;
} while (0)
```

do-while语句使得TRY-EXCEPT语句在语义上等价于C的语句，因此它可以像任何其他C语句一样使用。例如，它也可以用作if语句的结果语句。图4-1显示的是常规TRY-EXCEPT语句产生的代码，带阴影的代码块突出显示了扩展TRY和END_TRY宏指令所产生的代码；单线条框围住的代码块是EXCEPT宏指令的代码，而双线条框围住的代码块是ELSE宏指令的代码。图4-2显示的是扩展TRY-FINALLY语句；单线条框围住的代码块是FINALLY宏指令的代码。

Except_Frame的空间分配很简单，在由TRY开始的do-while主体中的复合语句内部声明一个该类型的局部变量即可：

```
⟨exported macros 48⟩+≡
  #define TRY do { \
      volatile int Except_flag; \
      Except_Frame Except_frame; \
      ⟨push 56⟩ \
      Except_flag = setjmp(Except_frame.env); \
      if (Except_flag == Except_entered) {
```

在TRY语句内有四种状态，由下面的枚举标识符给出

```
⟨exported types 53⟩+≡
  enum { Except_entered=0, Except_raised,
         Except_handled,   Except_finalized };
```

54

setjmp的第一个返回值将Except-flag设置为Except_entered，表示进入TRY语句，并且将某个异常帧压入异常栈。Except_entered必须为0，因为setjmp首次调用的返回值为0；随后，setjmp的返回值将被设为Except_raised，表示发生了异常。处理程序将Except_flag的值设成Except_handled，表示处理程序已经对异常进行了处理。

```
do {
   volatile int Except_flag;
   Except_Frame Except_frame;
   Except_frame.prev = Except_stack;
   Except_stack = &Except_frame;
   Except_flag = setjmp(Except_frame.env);
   if (Except_flag == Except_entered) {
      S
   if (Except_flag == Except_entered)
      Except_stack = Except_stack->prev;
} else if (Except_frame.exception == &(e1)) {
   Except_flag = Except_handled;
      S1
   if (Except_flag == Except_entered)
      Except_stack = Except_stack->prev;
} else if (Except_frame.exception == &(e2)) {
   Except_flag = Except_handled;
      S2
   if (Except_flag == Except_entered)
      Except_stack = Except_stack->prev;
} ...
} else if (Except_frame.exception == &(en)) {
   Except_flag = Except_handled;
      Sn
   if (Except_flag == Except_entered)
      Except_stack = Except_stack->prev;
} else {
   Except_flag = Except_handled;
      S0
   if (Except_flag == Except_entered)
      Except_stack = Except_stack->prev;
}
   if (Except_flag == Except_raised)
      Except_raise(Except_frame.exception,
         Except_frame.file, Except_frame.line);
} while (0)
```

图4-1 TRY-EXCEPT语句的扩展

```
do {
    volatile int Except_flag;
    Except_Frame Except_frame;
    Except_frame.prev = Except_stack;
    Except_stack = &Except_frame;
    Except_flag = setjmp(Except_frame.env);
    if (Except_flag == Except_entered) {
        S
        if (Except_flag == Except_entered)
            Except_stack = Except_stack->prev;
    } {
        if (Except_flag == Except_entered)
            Except_flag = Except_finalized;
        S1
        if (Except_flag == Except_entered)
            Except_stack = Except_stack->prev;
    }
    if (Except_flag == Except_raised)
        Except_raise(Except_frame.exception,
            Except_frame.file, Except_frame.line);
} while (0)
```

图4-2 TRY-FINALLY语句的扩展

Except_Frame的入栈操作是将它插入到Except_stack指向的Except_Frame结构链表的表头，而栈顶帧的出栈操作是将它从链表中移出：

```
⟨push 56⟩≡
  Except_frame.prev = Except_stack; \
  Except_stack = &Except_frame;

⟨pop 56⟩≡
  Except_stack = Except_stack->prev
```

EXCEPT从句变成了如图4-1所示的else-if语句。

```
⟨exported macros 48⟩+≡
  #define EXCEPT(e) \
          ⟨pop if this chunk follows S 57⟩ \
      } else if (Except_frame.exception == &(e)) { \
          Except_flag = Except_handled;

⟨pop if this chunk follows S 57⟩≡
  if (Except_flag == Except_entered) ⟨pop 56⟩;
```

55
i
56

使用宏指令处理异常将会产生一些断续的代码，就像<*pop if this chunk follows S 57*>代码块那样。这个代码块出现在上述EXCEPT定义中的else-if语句之前，而且仅仅只在第一个EXCEPT从句中将异常栈的栈顶元素弹出。如果在执行了S的时候没有发生异常，Except_flag的值将仍然为Except_entered，这样当控制到达if语句时，异常栈进行出栈操作。第二个以及接下来的EXCEPT从句将在Except_flag变为Except_handled的处理程序之后执行。因为前面的执行步骤，

异常栈已经进行了出栈操作，并且<*pop if this chunk follows S 57*>中的if语句保证不会再次进行出栈操作。

ELSE从句和EXCEPT从句类似，但是else-if只是一个else：

```
⟨exported macros 48⟩+≡
  #define ELSE \
          ⟨pop if this chunk follows S 57⟩ \
      } else { \
          Except_flag = Except_handled;
```

同样，FINALLY与ELSE类似，但是没有else语句：

```
⟨exported macros 48⟩+≡
  #define FINALLY \
          ⟨pop if this chunk follows S 57⟩ \
      } { \
          if (Except_flag == Except_entered) \
              Except_flag = Except_finalized;
```

在这里，Except_flag从Except_entered变为Except_finalized，说明并没有发生异常，但是却出现了FINALLY从句。如果发生了异常，Except_flag的值将仍为Except_raised，这样在执行完清除代码后，可以重新引发异常，该异常可通过检测在扩展END_TRY中Except_flag标志的值是否等于Except_raised来触发。如果没有发生异常，那么Except_flag的值将会是Except_entered或Except_finalized：

57

```
⟨exported macros 48⟩+≡
  #define END_TRY \
          ⟨pop if this chunk follows S 57⟩ \
          } if (Except_flag == Except_raised) RERAISE; \
  } while (0)
```

except.c中Except_raise的实现是最后一个难点：

```
⟨except.c⟩≡
  #include <stdlib.h>
  #include <stdio.h>
  #include "assert.h"
  #include "except.h"
  #define T Except_T

  Except_Frame *Except_stack = NULL;

  void Except_raise(const T *e, const char *file,
      int line) {
      Except_Frame *p = Except_stack;

      assert(e);
      if (p == NULL) {
          ⟨announce an uncaught exception 59⟩
      }
      p->exception = e;
```

```
        p->file = file;
        p->line = line;
        ⟨pop 56⟩;
        longjmp(p->env, Except_raised);
    }
```

如果在异常栈的栈顶有一个Except_Frame，那么Except_raise填写字段exception、file以及line，并将异常栈的栈顶元素弹出，然后调用longjmp。相应的setjmp调用将返回Except_raised；在TRY-EXCEPT或TRY-FINALLY语句中，Except_flag的值变为 58
Except_raised，并执行相应的处理程序。Except_raise对异常栈进行出栈操作，这样，如果某个处理程序中发生了异常，该异常可交由TRY-EXCEPT语句来处理，因为该异常的帧现在已经在异常栈的栈顶了。

如果异常栈为空，那么就不存在处理程序，因此Except_raise只能声明异常为未处理的异常并终止程序：

```
⟨announce an uncaught exception 59⟩≡
  fprintf(stderr, "Uncaught exception");
  if (e->reason)
      fprintf(stderr, " %s", e->reason);
  else
      fprintf(stderr, " at 0x%p", e);
  if (file && line > 0)
      fprintf(stderr, " raised at %s:%d\n", file, line);
  fprintf(stderr, "aborting...\n");
  fflush(stderr);
  abort();
```

abort是一个标准C库函数，它终止运行，有时还会执行与机器相关的附加操作。例如，它可能打开一个调试程序或仅仅进行内存的转储拷贝。

4.3 断言

一般标准要求头文件assert.h把assert(e)定义成一个提供诊断信息的宏指令。assert(e)对e求值，如果e为0，那么在标准错误上写上诊断信息并调用标准库函数abort中断执行。诊断信息包括失败的断言（e的内容）以及assert(e)出现的位置（文件以及行号）。信息的格式由具体实现定义。用assert(0)标识“不可能发生”的状态就是一种很好的方法。其他表示断言的方法，像：

```
assert(!"ptr==NULL -- can't happen")
```

可以显示更多有意义的诊断信息。

assert.h也使用了宏指令NDEBUG，但是没有对它进行定义。如果定义了NDEBUG，那么assert(e)一定与空表达式（(void)0）等价。因此，程序员可以通过定义NDEBUG，并重新 59
编译来关掉断言。因为e不一定要执行，因此它不能是有附带影响的基本计算，就像赋值操作那样，这点很重要。

assert(e)是一个表达式，因此大多数版本的assert.h在逻辑上都等价于：

```
#undef assert
#ifdef NDEBUG
#define assert(e) ((void)0)
#else
extern void assert(int e);
#define assert(e) ((void)((e)||\
    (fprintf(stderr, "%s:%d: Assertion failed: %s\n", \
    __FILE__, (int)__LINE__, #e), abort(), 0)))
#endif
```

（实际的assert.h通常与这个不一样，因为实际情况并不允许为了使用fprintf和stderr而包含stdio.h）。类似于$e_1||e_2$的表达式经常出现在条件语境中，例如if语句，但是它也可以作为单独的语句出现。当它单独出现的时候，效果等价于语句：

```
if (!(e1)) e2;
```

assert的定义使用了语句$e_1||e_2$，这是因为assert(e)必须扩展成一个表达式，而不是一个语句。e_2是一个逗号表达式，它的返回结果是一个值，这是||操作符的要求，而整个表达式最后的值是空，因为标准规定assert(e)是没有返回值的。在标准的C预处理器中，习惯用法#e将产生一个字符串，它的内容是e表达式内容中的字符。

Assert接口定义的assert(e)与标准中定义的类似，不同的是断言失败将触发一个Assert_Failed异常，而不是终止运行，并且不提供断言e的内容：

```
⟨assert.h⟩≡
  #undef assert
  #ifdef NDEBUG
60  #define assert(e) ((void)0)
  #else
  #include "except.h"
  extern void assert(int e);
  #define assert(e) ((void)((e)||(RAISE(Assert_Failed),0)))
  #endif

⟨exported variables 53⟩≡
  extern const Except_T Assert_Failed;
```

Assert模仿了标准中的定义，因此这两个assert.h头文件可交替地使用，这也就是为什么在except.h中会出现Assert_Failed的原因。这个结构的实现很简单：

```
⟨assert.c⟩≡
  #include "assert.h"

  const Except_T Assert_Failed = { "Assertion failed" };

  void (assert)(int e) {
      assert(e);
  }
```

函数定义中assert两端的括号禁止了宏指令assert的扩展，这样就可以按接口要求的方式来定义函数。

如果客户调用程序不处理Assert_Failed，那么断言失败会引起程序的中断，并给出如下信息：

```
Uncaught exception Assertion failed raised at stmt.c:201
aborting...
```

该信息在功能上等价于针对具体机器版本的assert.h中的诊断信息。

把断言打包，使得当它们失败时会引发异常，这有助于处理产品程序中出现了断言的情况。一些程序员建议不把断言留在产品程序中，并通过assert.h中使用NDEBUG的标准来支持这个建议。不把断言留在产品中最常用的两个理由就是有效性和隐藏诊断信息的可能性。

执行断言需要一定的时间，因此把断言移出可以使程序运行得更快。然而，有断言和没断言的程序在执行时间上的差别是可以度量的，而这个差别通常是很小的。为了有效性而消除断言和为了改进运行时间而进行的其他变动一样，其必要性依赖于客观的度量结果。 61

当度量的结果表明执行断言花费很高时，有可能为了减少花费而去掉断言，同时又不失去其好处。例如，假设*h*包含了某个开销很高的断言时，而*f*和*g*都调用*h*，且度量结果表明大部分的时间花费在*g*对*h*的调用上，因为*g*对*h*的调用是在循环内进行的。经过仔细地分析后，你会发现*h*中断言既可以移到*f*中，也可以移到*g*中，并放在*g*的循环之前。

关于断言的一个更严重的问题是，它们可能会引发像前面提到的断言失败之类的诊断信息，这样会迷惑客户调用程序。但是去掉断言，取代这些诊断信息可能会引起更大的问题。当某个断言失败时，那么程序就是错的。如果程序继续，它就会产生不可预测的结果，而且很有可能会崩溃。类似

```
General protection fault at 3F60:40EA
```

或是

```
Segmentation fault -- core dumped
```

的信息都不会比前面的断言失败的诊断信息好到哪去。更糟糕的是，在出现了某个可能会引起程序终止的断言失败后而又继续执行的程序，很可能会破坏用户的数据。例如，编辑器可能会破坏用户的文件，而这种破坏是无法弥补的。

要隐藏断言失败的诊断信息所带来的问题，可以用程序的产品版本中最外层的TRY-EXCEPT语句来处理。该程序可以找到所有的未处理的异常，并给出更有帮助的诊断信息。例如：

```
#include <stdlib.h>
#include <stdio.h>
#include "except.h"

int main(int argc, char *argv[]) {
    TRY
        edit(argc, argv);
    ELSE
```

62

```
        fprintf(stderr,
    "An internal error has occurred from which there is "
    "no recovery.\nPlease report this error to "
    "Technical Support at 800-777-1234.\nNote the "
    "following message, which will help our support "
    "staff\nfind the cause of this error.\n\n")
        RERAISE;
    END_TRY;
    return EXIT_SUCCESS;
}
```

当发生了一个未处理的异常时，这个处理程序在隐藏诊断信息之前先给出了报告错误的指令。例如，如果出现了断言失败的话，它将会打印出：

```
An internal error has occurred from which there is no recovery.
Please report this error to Technical Support at 800-777-1234.
Note the following message, which will help our support staff
find the cause of this error.

Uncaught exception Assertion failed raised at stmt.c:201
aborting...
```

参考书目浅析

好几种程序设计语言都嵌入了异常处理机制，例如Ada、Modula-3(Nelson 1991)、Eiffel(Meyer 1992)以及C++(Ellis和Stroustrup 1990)。Except接口中的TRY-EXCEPT语句就是模仿的Modula-3中的TRY-EXCEPT语句。

C语言提出了几种异常处理机制，它们都提供了类似TRY-EXCEPT语句的功能，有时在语法和语义上有些改变。Roberts(1989)描述了一个异常处理功能的接口，它与Except接口提供的功能相同。它的实现是类似的，但是在发生异常时，它更有效。Except_raise调用longjmp把控制转给某个处理程序。如果这个处理程序没有处理异常，那么Except_raise会再次调用longjmp。如果要处理异常的是异常栈中的第*N*个异常帧，Except_raise和longjmp将会被调用*N*次。Roberts的实现只需要调用一次就能找到相应的处理程序或是找到第一个FINALLY从句。
63 为了达到这个效果，它必须对TRY-EXCEPT语句中的异常处理程序的数目设置一个上界。一些C编译器，如微软的c编译器，就提供了结构化的异常处理工具作为程序设计语言的扩展。

一些程序设计语言同样也嵌入了断言机制，例如Eiffel。大多数程序设计语言使用的是类似assert宏指令或其他编译指令的工具来指明断言。例如，Digital的Modula-3编译器可以识别形如<*ASSERT expression*>的注释作为声明断言的编译参数。Maguire(1993)用了一整章来介绍了C程序中断言的使用。

练习

4.1 既有EXCEPT又有FINALLY的语句，其执行效果如何？语句的形式为

```
TRY
    S
EXCEPT(e1)
    S1
...
EXCEPT(en)
    Sn
FINALLY
    S0
END_TRY
```

4.2 修改Except接口与实现，使得Except_raise只调用longjmp来找到相应的处理程序或FINALLY从句，就像上面内容描述的，由Roberts(1989)实现的程序一样。

4.3 UNIX系统使用信号来通知某些异常情况，例如浮点溢出或是用户点击了中断键。研究UNIX的信号指令系统，设计并实现一个将信号转变成异常的信号处理程序的接口。

4.4 当程序被终止时，一些系统会打印出栈的记录。在程序终止时，这些记录会显示出过程调用栈的状态，还可能包括过程名和参数。修改Except_raise，当出现未处理的异常时打印出栈的记录。根据计算机的调用习惯，你可能会打印出过程名以及调用的行号。例如，记录的形式可能如下：

64

```
Uncaught exception Assertion failed
raised in whilestmt() at stmt.c:201
called from statement() at stmt.c:63
called from compound() at decl.c:122
called from funcdefn() at decl.c:890
called from decl() at decl.c:95
called from program() at decl.c:788
called from main() at main.c:34
aborting...
```

4.5 在一些系统上，当程序检测到错误的时候，可以自己调用调试程序。在断言失败很常见的开发阶段，这是特别有用的。如果你的系统支持这个功能，那么试着修改Except_raise，当出现未处理的异常时打开调试程序，而不是调用abort，并尝试将你的实现形成产品程序；也就是说，看看在运行阶段它是否会调用调试程序。

4.6 如果你可以得到某个C的编译器，例如lcc(Fraser和Hanson 1995)，那么尝试修改该编译器，但不使用setjmp和longjmp，使编译器能够支持异常、TRY语句以及具有本章中描述的语法和语义的RAISE和RERAISE表达式。你可能需要实现类似setjmp和longjmp的机制，但要求它专用于异常处理。例如，只使用少数的指令来实例化处理程序一般是可行的。警告：这个练习是大型的项目。

65

第5章 内存管理

所有不平凡的C程序都会在运行阶段分配内存单元的。标准的C程序库提供了四个内存管理的例程：malloc、calloc、realloc和free。接口Mem用一系列宏指令和例程重新包装了这些例程，使得这些例程错误更少，并且还提供了一些其他的功能。

不幸的是，内存管理错误在C中是很普遍的，而且它们通常很难检测和修复。例如，程序片段：

```
p = malloc(nbytes);
...
free(p);
```

调用malloc来分配一个大小为nbytes的存储块（block），并将存储块第一个字节的地址赋值给p，然后使用p和它指向的存储块，最后释放该内存单元。在调用完free后，p中存储指针变成悬挂指针，即一个指向逻辑上并不存在的内存单元的指针。随后对p的间接引用就会产生错误，尽管存储块并没有因为其他的目的而被重新分配，而这个错误也很难检测到，即使检测到了，其发生地和发生时间也可能是远离错误源的。

程序片段：

```
p = malloc(nbytes);
...
free(p);
...
free(p);
```


说明了另一种错误：释放空闲内存单元。这个错误通常会破坏内存管理函数所使用的数据结构，但是它可能直到下一次这些函数被调用时才被发现。

另一个错误是释放的内存单元并不是由malloc、calloc或realloc分配的。例如：

```
char buf[20], *p;
if (n >= sizeof buf)
    p = malloc(n);
else
    p = buf;
...
free(p);
```

上面这段程序的目的是避免在buf的大小大于等于n时进行空间分配；但是当p指向buf时，代码调用free就会引起错误。而且，这个错误通常会破坏内存管理的数据结构，而且到运行以后才会被检查到。

最后，函数

```
void itoa(int n, char *buf, int size) {
    char *p = malloc(43);

    sprintf(p, "%d", n);
    if (strlen(p) >= size - 1) {
        while (--size > 0)
            *buf++ = '*';
        *buf = '\0';
    } else
        strcpy(buf, p);
}
```

用十进制整数n来填充数组buf[0..size-1]，当表示的结果多于size-1个字符时用“*”来填充数组。这个代码看上去很健壮，但是它却至少包含了两个错误，第一个错误：当分配失败时malloc会返回一个空指针，而代码并没有检测该情况。第二个错误：代码产生了内存泄漏，它并没有将它分配的内存单元收回。在每次调用itoa时，程序会慢慢地消耗内存。如果itoa被经常调用，那么程序最后会耗尽内存而导致失败。同时，当size小于2的时候，itoa仍然会正确地工作，但是它将buf[0]置成了空字符。可能在某个更好的设计中会要求size必须超过2，并用一个可检查的运行期错误来增强该限制条件。

68

Mem接口中的宏指令和例程提供了一些保护措施来防止这类内存管理错误的发生。但是它们并没有消除这类错误。例如，它们不能防止间接引用已破坏的指针或是使用超出了范围的局部变量指针等等。C初学者通常会犯后面一种错误，以下这个简单版本的itoa函数就是这样的例子：

```
char *itoa(int n) {
    char buf[43];

    sprintf(buf, "%d", n);
    return buf;
}
```

一旦itoa返回的是其局部变量buf的地址，buf就不再存在了。

5.1 接口

Mem接口导出了异常、例程和宏指令：

```
⟨mem.h⟩≡
  #ifndef MEM_INCLUDED
  #define MEM_INCLUDED
  #include "except.h"

  ⟨exported exceptions 70⟩
  ⟨exported functions 70⟩
  ⟨exported macros 70⟩

  #endif
```

Mem的分配函数与标准C函数库中的分配函数类似，但是它们不接收大小为0的分配，也不会返回空指针：

69

```
⟨exported exceptions 70⟩≡
  extern const Except_T Mem_Failed;

⟨exported functions 70⟩≡
  extern void *Mem_alloc (long nbytes,
      const char *file, int line);
  extern void *Mem_calloc(long count, long nbytes,
      const char *file, int line);
```

Mem_alloc分配一个大小至少为nbytes的存储块，并返回一个指向第一个字节的指针。该存储块是按某个地址边界进行调整的，这种调整通过有很严格的边界调整要求的数据得以实现。存储块中的内容是未初始化的。如果nbytes是一个非正数，那么这将是一个可检查的运行期错误。

Mem_calloc分配足够大的存储块，使得可以存下count个元素的数组，每个元素的大小为nbytes，并返回一个指向第一个元素的指针。存储块采用与Mem_alloc相同的边界调整，并被初始化为0。空指针和0.0并不一定会表示成0，因此Mem_calloc可能会不正确地初始化存储块。如果count或nbytes为非正数，那么这也是一个可检查的运行期错误。

Mem_alloc和Mem_calloc的最后两个参数是调用所在文件的文件名和行号。它们由下面的宏指令来给出，这些宏指令也是调用这些函数的最常用的方式。

```
⟨exported macros 70⟩≡
  #define ALLOC(nbytes) \
      Mem_alloc((nbytes), __FILE__, __LINE__)
  #define CALLOC(count, nbytes) \
      Mem_calloc((count), (nbytes), __FILE__, __LINE__)
```

如果Mem_alloc或是Mem_calloc不能分配所要求的内存单元，那么它们会产生异常Mem_Failed，并将file和line传给Except_raise，将产生错误的位置写入异常报告中。如果file是空指针，那么Mem_alloc和Mem_calloc将在Mem_Failed异常的实现中将位置信息补充上。

许多分配操作具有以下格式：

```
struct T *p;
p = Mem_alloc(sizeof (struct T));
```

它为结构T的某个实例分配一个内存单元，并返回一个指向该内存单元的指针。这种习惯用法的更好形式是：

70

```
p = Mem_alloc(sizeof *p);
```

用sizeof *p代替sizeof(struct T)，这样对除了空指针以外的任何类型的指针都可以完成分配，并且sizeof *p是独立于指针类型的。如果p的类型改变了，这种分配仍然是正确的，但是如果使用的是sizeof(struct T)，就必须修改语句来反映p的类型变化。也就是说语句

```
p = Mem_alloc(sizeof (struct T));
```

只有在p确实是指向结构T的指针时才是正确的。如果p变成了指向另一个结构的指针，而这个调用没有更新，那么该调用就可能分配过多的内存单元，这样会造成空间的浪费；或者分配过少的内存单元，这样客户调用程序有可能对未分配的内存单元中的内容进行修改，这将带来严重的问题。

这种分配习惯是很常用的，因此Mem提供的宏指令封装了分配和赋值操作：

```
⟨exported macros 70⟩+≡
  #define  NEW(p) ((p) = ALLOC((long)sizeof *(p)))
  #define NEW0(p) ((p) = CALLOC(1, (long)sizeof *(p)))
```

NEW(p)分配了一个未初始化的存储块来存储*p，并把存储块的地址赋给p。NEW0(p)完成同样的功能，不同的是它同时也清除了存储块中的内容。提供NEW是建立在假设大多数客户调用程序会在分配后立即对内存单元进行初始化的基础之上的。编译阶段的操作符sizeof的参数只用来指明它的类型，在运行阶段并不求值。因此NEW和NEW0只对p求一次值，并且在这两个宏指令中使用具有附加功能的表达式作为实参也是很安全的。例如NEW(a[i++])。

malloc和calloc都用了size_t类型的参数；sizeof产生一个size_t类型的常数。类型size_t是一个无符号的整数类型，可以表示能声明的最大对象的大小，并且不管对象的大小在哪里指定，它都可以在标准的库函数中使用。实际上，size_t既可以是无符号的整数，也可以是无符号的长整数。Mem_alloc和Mem_calloc采用整型参数，以避免出现将负数传给无符号参数的错误。例如：

```
    int n = -1;
    ...
71  p = malloc(n);
```

明显就是错误的，但是许多malloc的实现都不会发现这个错误，因为-1在被转换成了一个size_t类型的数后，通常是一个很大的无符号数。

内存单元的释放是由Mem_free实现的：

```
⟨exported functions 70⟩+≡
  extern void Mem_free(void *ptr,
      const char *file, int line);

⟨exported macros 70⟩+≡
  #define FREE(ptr) ((void)(Mem_free((ptr), \
      __FILE__, __LINE__), (ptr) = 0))
```

Mem_free接收一个指向需被释放的存储块的指针作为参数。如果ptr不为空，那么Mem_free释放该存储块；如果ptr为空，那么Mem_free什么也不做。宏指令FREE同样接收一个指针作为参数，并调用Mem_free来释放存储块，然后将ptr赋为空指针，就像2.4节中提到的那样，这样可以帮助我们避免出现悬挂指针。因为ptr在它所指的对象被释放后，就变为空，因此此后的间接引用通常会因为某类型的地址错误而引起程序崩溃。这种明显的错误比间接引用某个悬挂指针可能引起的不可预测的后果要好的多。注意：FREE对ptr进行了多次求值。

在随后章节的细节描述中，将有两个实现导出Mem接口。校验实现（checking implementation）通过检测运行期错误以帮助捕获前一节中所描述的那些存取访问错误。在该实现中，如果将一个并不是由Mem_alloc、Mem_calloc或Mem_resize返回的非空ptr或者某个已经传给Mem_free或Mem_resize的ptr传给Mem_free时，会产生可检查的运行期错误。Mem_free的file和line参数值用来报告这些运行期错误。

但是，在产品级实现（production implementation）中，这些存取访问错误都是不可检查的运行期错误。

函数：

```
⟨exported functions 70⟩+≡
  extern void *Mem_resize(void *ptr, long nbytes,
      const char *file, int line);
⟨exported macros 70⟩+≡
  #define RESIZE(ptr, nbytes) ((ptr) = Mem_resize((ptr), \
      (nbytes), __FILE__, __LINE__))
```

72

改变了前面调用Mem_alloc、Mem_calloc或Mem_resize分配得到的存储块的大小。与Mem_free一样，Mem_resize的第一个参数存放需要改变大小的存储块的地址的指针。Mem_resize扩充或缩小该存储块，使它至少有nbytes的存储单元，进行适当的边界调整，并返回一个指向调整过的存储块的指针。Mem_resize为了改变存储块的大小，可能会移动该存储块，因此逻辑上Mem_resize等价于分配一个新的存储块，从ptr复制一部分或所有的数据到新的存储块中，并释放ptr。如果Mem_resize不能分配一个新的存储块，那么它将会产生异常Mem_Failed，file和line为异常产生的位置。宏指令RESIZE将ptr的指向改为新的存储块，这也是Mem_resize最常用的方法。注意：RESIZE也对ptr进行了多次的求值。

如果nbytes超过了ptr指向的存储块的大小，那么多余的字节将不被初始化。否则，从ptr开始的nbytes将被复制到新的存储块中。

如果将空的ptr传给Mem_resize，或是nbytes为非正数，会产生可检查的运行期错误。在校验实现中，ptr不是由前面Mem_alloc、Mem_calloc或Mem_resize的调用所返回，或是已经传递给Mem_free、Mem_resize的ptr传递给Mem_resize，这是一个可检查的运行期错误。而在产品级实现中，这些存取错误都是不可检查的运行期错误。

Mem接口中的函数可以用做C标准库函数malloc、calloc、realloc和free的补充。也就是说，程序既可以使用标准C库函数中的分配函数，也可以使用Mem接口中的分配函数。只将那些校验实现所管理的内存单元存取错误视为可检查的运行期错误而报告。在任意给定的程序中，只能使用一个Mem接口的实现。

5.2 产品级实现

在产品级实现中，例程将标准函数库的内存管理函数的调用封装在由Mem接口说明的更安全的程序包中：

73

```
⟨mem.c⟩≡
  #include <stdlib.h>
  #include <stddef.h>
  #include "assert.h"
  #include "except.h"
  #include "mem.h"

  ⟨data 74⟩
  ⟨functions 74⟩
```

例如，Mem_alloc调用malloc，并在malloc返回空指针时，产生异常Mem_Failed：

```
⟨functions 74⟩≡
  void *Mem_alloc(long nbytes, const char *file, int line){
      void *ptr;

      assert(nbytes > 0);
      ptr = malloc(nbytes);
      if (ptr == NULL)
          ⟨raise Mem_Failed 74⟩
      return ptr;
  }

⟨raise Mem_Failed 74⟩≡
  {
      if (file == NULL)
          RAISE(Mem_Failed);
      else
          Except_raise(&Mem_Failed, file, line);
  }

⟨data 74⟩≡
  const Except_T Mem_Failed = { "Allocation Failed" };
```

如果某个客户调用程序没有处理Mem_Failed异常，那么Except_raise会给出调用者的位置，该位置信息在它报告未处理的异常时以参数传给Mem_alloc。例如：

```
Uncaught exception Allocation Failed raised @parse.c:431
aborting...
```

74

类似地，Mem_calloc函数也封装了对calloc的调用：

```
⟨functions 74⟩+≡
  void *Mem_calloc(long count, long nbytes,
      const char *file, int line) {
      void *ptr;

      assert(count > 0);
      assert(nbytes > 0);
      ptr = calloc(count, nbytes);
      if (ptr == NULL)
          ⟨raise Mem_Failed 74⟩
      return ptr;
  }
```

当count或nbytes为0时，calloc的行为在实现中给出了定义。Mem接口说明了在这些情况下会发生什么，这也是它的优点之一，并可以帮助避免错误的发生。

Mem_free仅仅调用函数free：

```
⟨functions 74⟩+≡
  void Mem_free(void *ptr, const char *file, int line) {
      if (ptr)
          free(ptr);
  }
```

标准的库函数允许空指针传递给函数free，但是Mem_free不允许，因为free原来的实现就不接收空指针。

Mem_resize比realloc函数的说明简单得多，这在其更简单的实现中得到了反映：

```
⟨functions 74⟩+≡
  void *Mem_resize(void *ptr, long nbytes,
      const char *file, int line) {

      assert(ptr);
      assert(nbytes > 0);
      ptr = realloc(ptr, nbytes);
      if (ptr == NULL)
          ⟨raise Mem_Failed 74⟩
    return ptr;
  }
```

Mem_resize的惟一目的就是改变一个已存在的存储块的大小。realloc也可以完成同样的功能，但是当nbytes为0时，realloc同时释放了该存储块；而当ptr为空指针时，realloc也同时分配了某个存储块。这些附加的功能与改变某个已存在的存储块的大小的关系并不大，但是却引入了错误。

5.3 校验实现

Mem接口的校验实现所导出的函数可以捕获在本章的一开始提到的那类存取访问错误，并把它们作为可检查的运行期错误来报告。

```
⟨memchk.c⟩≡
  #include <stdlib.h>
  #include <string.h>
  #include "assert.h"
  #include "except.h"
  #include "mem.h"

  ⟨checking types 80⟩
  ⟨checking macros 79⟩
  ⟨data 74⟩
  ⟨checking data 77⟩
  ⟨checking functions 79⟩
```

如果Mem_alloc、Mem_calloc和Mem_resize从来不会两次返回相同的地址，并且它们能记住它们返回的所有地址，以及哪个地址指向已分配的内存单元，那么Mem_free和Mem_resize就可以检查到存取访问错误。理论上说，这些函数都保存有一个集合S，其元素都是二元组（α，free）或（α，allocated），其中α是由分配操作返回的地址。值free说明地址α不再指向分配的内存单元；也就是说该内存单元已经明确地释放了，而值allocated说明地址α指向已分配的存储单元。

Mem_alloc和Mem_calloc添加一个二元组(ptr,allocated)到集合S中，其中ptr是它们的返回值，并且它们保证在添加之前，S中不会出现(ptr,allocated)或(ptr,free)。当ptr为空，或者
76 (ptr,allocated)在S中时，Mem free(ptr)是合法的。如果ptr不为空，且(ptr,allocated)已经在S中存在，那么Mem_free将释放ptr指向的内存单元，并将集合S中的相应元素项改为(ptr,free)。同样，Mem_resize(ptr,nbytes,...)也只有在(ptr,allocated)已经在S中存在时才是合法的。如果是这样，那么Mem_resize调用Mem_alloc分配一个新的存储块，将原来存储块中的内容复制到新的存储块中，并调用Mem_free释放原来的存储单元；这些调用都会使集合S产生相应的变化。

对于分配函数永远不会两次返回相同的值的情况，可以采用永不释放任何内存单元来实现。但是这个方法浪费了空间，一个更好的做法是：永不释放某个分配函数先前返回的地址中的字节。S也可以通过保存一个这些字节的地址表来实现。

这些方案可以通过在标准库函数的开头写一个内存分配程序来实现。这个分配程序保存含有存储块描述符的一个散列表：

```
⟨checking data 77⟩≡
  static struct descriptor {
      struct descriptor *free;
      struct descriptor *link;
      const void *ptr;
      long size;
      const char *file;
      int line;
  } *htab[2048];
```

ptr是存储块的地址，该存储块是在下面描述的其他地方分配的；size是存储块的大小。file和line是存储块的分配位置——其源位置将传给分配该存储块的函数。这些值并没有被使用，但是它们存储在描述符中，以便调试程序可以在调试阶段把它们打印出来。

link字段形成一个在htab中具有相同散列值的块描述符链表，而htab是一个指向这些描述符的指针数组。这些描述符同样也形成了一个空闲存储块的链表；该链表的表头是一个虚构的描述符。

```
⟨checking data 77⟩+≡
77  static struct descriptor freelist = { &freelist };
```

并且该链表是通过描述符中的free字段连起来的。该链表也是循环的，freelist是链表中的最

后一个描述符，它的free字段指向第一个描述符。在任何时候，htab都存有所有存储块的描述符，包括空闲的和已分配的，空闲的存储块存在freelist中。因此，如果存储块是已分配的，那么free字段的值为空，如果存储块是空闲的，那么free字段的值不为空，并且用htab来实现集合S。图5-1显示了某个时间点这些数据结构的状态。与每个描述符结构相关联的空间出现在描述符结构的后面。阴影的空间是已分配的；而空白的空间是空闲的，从link字段连出来的是实线，而虚线连着的是空闲链表。

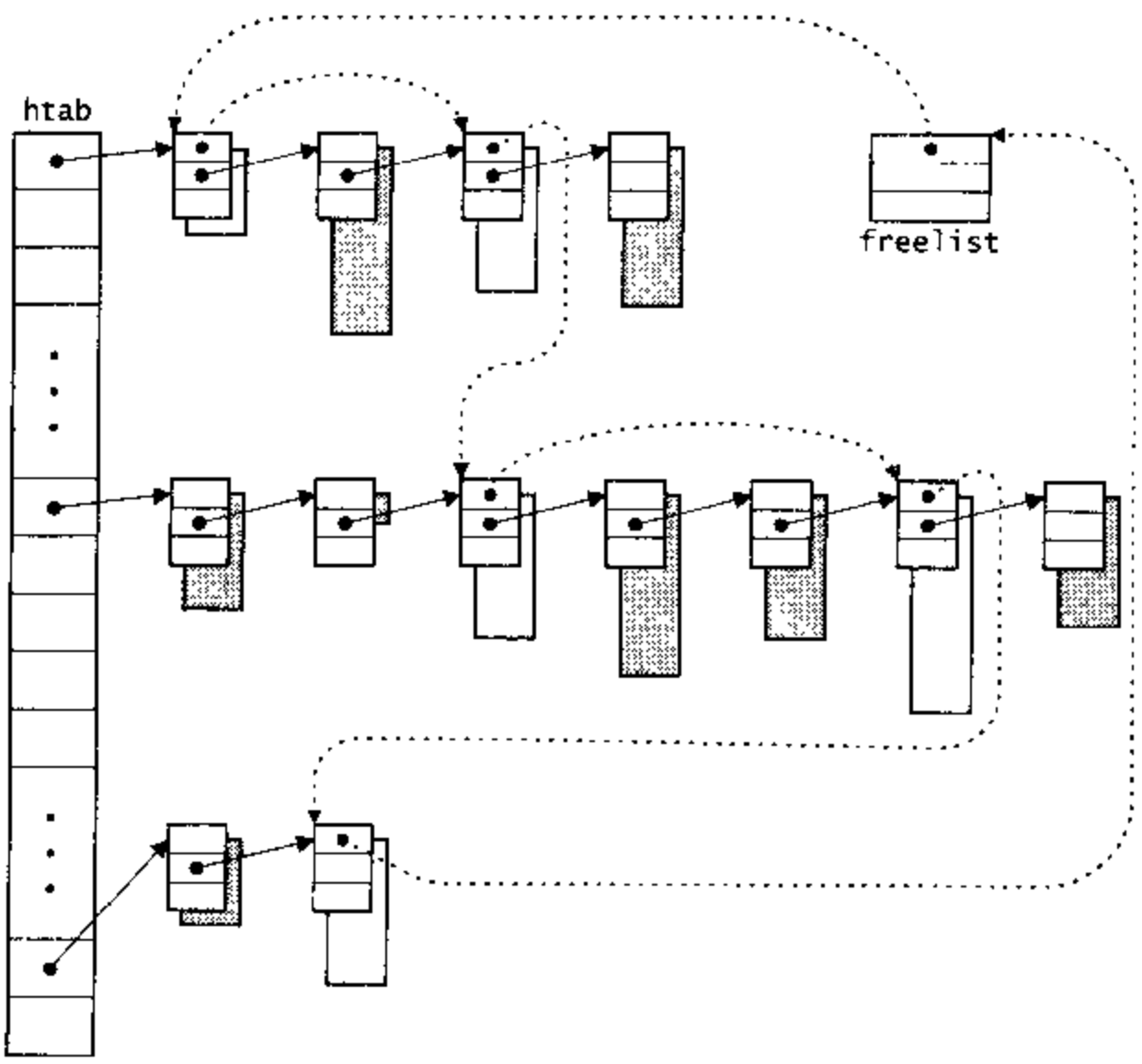


图5-1 htab和freelist的结构

78

给定一个地址，函数find查找它的描述符，它返回一个指向描述符的指针或者返回一个空指针：

```
⟨checking functions 79⟩≡
  static struct descriptor *find(const void *ptr) {
      struct descriptor *bp = htab[hash(ptr, htab)];

      while (bp && bp->ptr != ptr)
          bp = bp->link;
      return bp;
  }

⟨checking macros 79⟩≡
  #define hash(p, t) (((unsigned long)(p)>>3) & \
      (sizeof (t)/sizeof ((t)[0])-1))
```

hash宏指令以位模式来处理地址，右移三位，然后减去它并对htab大小取模。对于编写存取访问错误在运行期可测到的版本的Mem_free，find函数就足够了：

```
⟨checking functions 79⟩+≡
  void Mem_free(void *ptr, const char *file, int line) {
      if (ptr) {
          struct descriptor *bp;
          ⟨set bp if ptr is valid 79⟩
          bp->free = freelist.free;
          freelist.free = bp;
      }
  }
```

如果ptr非空且是一个有效的地址，那么存储块是通过把它添加到空闲的列表中来完成存储空间的释放，并且可以通过以后的Mem_alloc调用来重新使用。当指针指向某个已分配的存储块时称该指针是有效的：

```
⟨set bp if ptr is valid 79⟩≡
  if (((unsigned long)ptr)%(sizeof (union align)) != 0
  || (bp = find(ptr)) == NULL || bp->free)
      Except_raise(&Assert_Failed, file, line);
```

79

其中的测试语句((unsigned long)ptr)%(sizeof (union align))!=0避免了对无效地址调用find函数，这些地址不是最严格的地址边界的倍数，因此不可能是有效的存储块指针。

就像下面要说的那样，Mem_alloc返回的指针总是按某个地址的边界调整过的，该地址是以下共用体大小的倍数：

```
⟨checking types 80⟩≡
  union align {
      int i;
      long l;
      long *lp;
      void *p;
      void (*fp)(void);
      float f;
      double d;
      long double ld;
  };
```

这种边界调整保证任何类型的数据都可以存储在由Mem_alloc返回的存储块中。如果传递给Mem_free的ptr没有按这个边界调整，那么它可能不在htab中，因此可能是无效的。

Mem_resize通过同样的检查来捕获存取错误，然后调用Mem_free、Mem_alloc以及库函数memcpy:

```
⟨checking functions 79⟩+≡
  void *Mem_resize(void *ptr, long nbytes,
      const char *file, int line) {
      struct descriptor *bp;
      void *newptr;
```

```
        assert(ptr);
        assert(nbytes > 0);
        ⟨set bp if ptr is valid 79⟩
        newptr = Mem_alloc(nbytes, file, line);
        memcpy(newptr, ptr,
            nbytes < bp->size ? nbytes : bp->size);
        Mem_free(ptr, file, line);
        return newptr;
    }
```

80

同样，Mem_calloc也可以通过调用Mem_alloc和库函数memset来实现：

```
⟨checking functions 79⟩+≡
    void *Mem_calloc(long count, long nbytes,
        const char *file, int line) {
        void *ptr;

        assert(count > 0);
        assert(nbytes > 0);
        ptr = Mem_alloc(count*nbytes, file, line);
        memset(ptr, '\0', count*nbytes);
        return ptr;
    }
```

剩下来的就是分配描述符本身以及Mem_alloc的代码了。想要用一次分配同时完成这两个任务，一种办法就是分配一个足够大的存储块，既可以存储描述符也可以存储Mem_alloc调用所需的存储空间。这种方法有两个缺点：首先，它增加了难度，需要把空闲的内存单元块划分成满足要求的几个更小的块，而每个需求都必须有它自己的描述符；其次，它使得描述符很容易因借助指针或指向分配的存储块以外的索引的写操作而被破坏。

分配描述符分别将它们的分配从Mem_alloc中分离出来，这样减少了它们被破坏的机会，但是不能消除这种可能性。dalloc分配、初始化并返回一个描述符，将其从由malloc得到的512个描述符块中分离出来：

```
⟨checking functions 79⟩+≡
    static struct descriptor *dalloc(void *ptr, long size,
        const char *file, int line) {
        static struct descriptor *avail;
        static int nleft;

        if (nleft <= 0) {
            ⟨allocate descriptors 82⟩
            nleft = NDESCRIPTORS;
        }
        avail->ptr  = ptr;
        avail->size = size;
        avail->file = file;
        avail->line = line;
        avail->free = avail->link = NULL;
        nleft--;
```

81

```
        return avail++;
    }
```

⟨*checking macros* 79⟩+≡
```
#define NDESCRIPTORS 512
```

调用malloc可能会返回一个空指针，而dalloc又把这个空指针传递给dalloc的调用者。

⟨*allocate descriptors* 82⟩≡
```
avail = malloc(NDESCRIPTORS*sizeof (*avail));
if (avail == NULL)
    return NULL;
```

就像以下要说明的，当dalloc返回一个空指针时，Mem_alloc会产生Mem_Failed异常。

Mem_alloc对存储块的分配使用的是众多内存分配算法中的first-fit算法。它搜索freelist链表，找到第一个能够满足分配要求的存储块，然后从该存储块中划分出所需要的存储块。如果freelist链表中没有合适的存储块，Mem_alloc将调用malloc分配一个大于nbytes的内存块，并把它加到空闲的链表中，然后再次查找freelist列表。因为新的内存块大于nbytes，在第二次查找的时候，它就作为满足分配要求的存储块进行分配。代码如下：

⟨*checking functions* 79⟩+≡
```
void *Mem_alloc(long nbytes, const char *file, int line){
    struct descriptor *bp;
    void *ptr;
    assert(nbytes > 0);
    ⟨round nbytes up to an alignment boundary 83⟩
    for (bp = freelist.free; bp; bp = bp->free) {
        if (bp->size > nbytes) {
            ⟨use the end of the block at bp->ptr 83⟩
        }
        if (bp == &freelist) {
            struct descriptor *newptr;
            ⟨newptr ← a block of size NALLOC + nbytes 84⟩
            newptr->free = freelist.free;
            freelist.free = newptr;
        }
    }
    assert(0);
    return NULL;
}
```

82

Mem_alloc先把nbytes向上取整，使得它返回的每个指针都是共用体align大小的倍数：

⟨*round* nbytes *up to an alignment boundary* 83⟩≡
```
nbytes = ((nbytes + sizeof (union align) - 1)/
    (sizeof (union align)))*(sizeof (union align));
```

freelist.free指向空闲存储块链表的起始地址，也是循环开始的地方。第一个大小超过nbytes的存储块被用来进行分配。该空闲存储块底部的nbytes个字节被分割出来，分割出来

的存储块的描述符被创建、初始化并添加到htab中后，返回该存储块的地址。

```
⟨use the end of the block at bp->ptr 83⟩≡
  bp->size -= nbytes;
  ptr = (char *)bp->ptr + bp->size;
  if ((bp = dalloc(ptr, nbytes, file, line)) != NULL) {
      unsigned h = hash(ptr, htab);
      bp->link = htab[h];
      htab[h] = bp;
      return ptr;
  } else
      ⟨raise Mem_Failed 74⟩
```

图5-2说明了该代码块的效果：左边是一个描述符，它指向某个被划分之前的空闲空间；右边已分配的空间用阴影表示，并有一个新的描述符指向它。注意：新描述符的空闲链表为空。

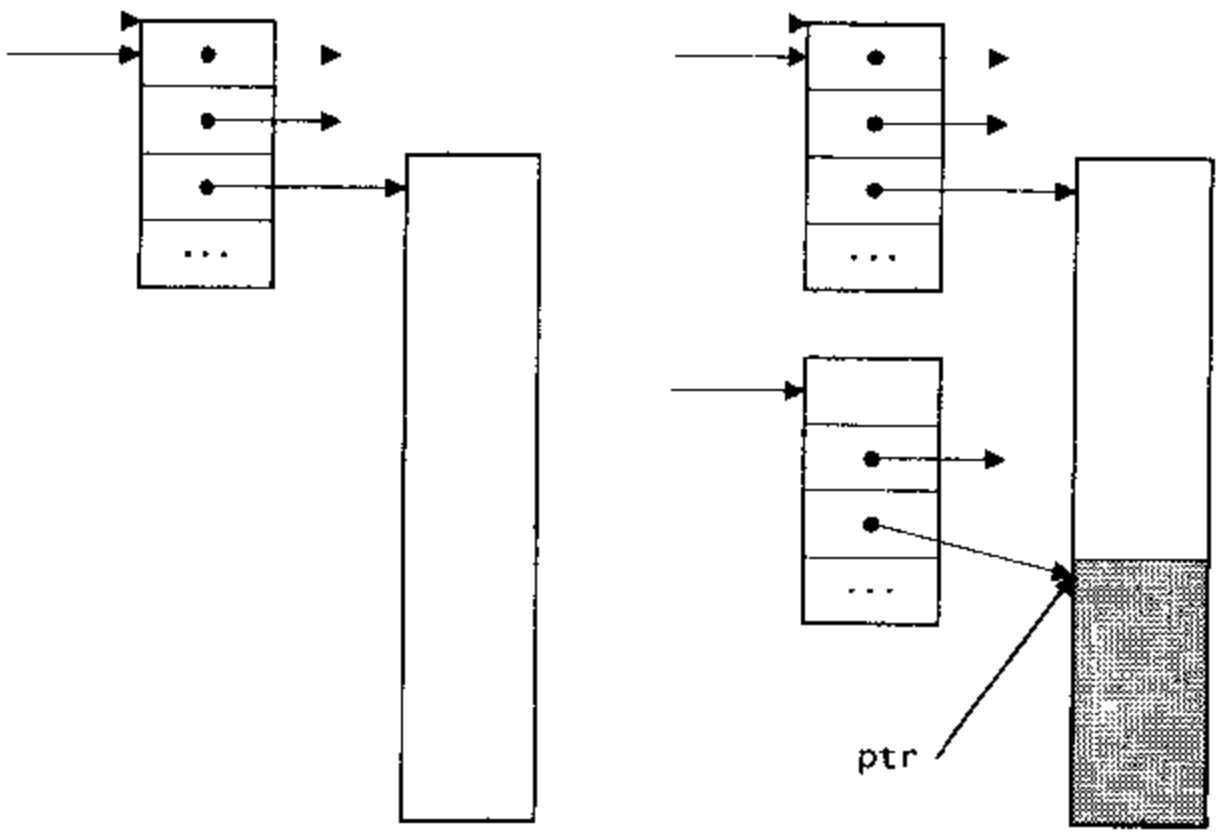


图5-2 分配一个空闲块的末端

测试语句bp->size > nbytes保证bp->ptr的值从不会被重复使用。大的空闲存储块被划分成小的存储块分配给小的需求，直到存储块的大小减小到sizeof(union align)个字节，在bp->size小于nbytes的时候就不再分配了。每个存储块的前sizeof(union align)个字节是从不进行分配的。

83

如果bp已到达freelist（即已搜索到链表的最后一个存储块），而链表中没有一个存储块的大小超过nbytes，在这种情况下，一个新的大小为

```
⟨checking macros 79⟩+≡
  #define NALLOC 4096
```

加上nbytes个字节的存储块，被添加到空闲存储块链表的表头；在下一次循环的迭代中，它将会被访问，并用来进行所需要的分配。这个新的存储块也有一个描述符，就好像它先前已

被分配和释放过一样：

```
⟨newptr ← a block of size NALLOC + nbytes 84⟩≡
  if ((ptr = malloc(nbytes + NALLOC)) == NULL
  ||  (newptr = dalloc(ptr, nbytes + NALLOC,
          __FILE__, __LINE__)) == NULL)
      ⟨raise Mem_Failed 74⟩
```

84

参考书目浅析

Mem接口的目的之一就是改进C的分配函数接口。Maguire(1993)对标准C分配函数进行了评价，并描述了一个类似的函数包。

内存分配错误在C程序中是很普遍的，以至有专门的公司致力于开发并出售这样的工具，帮助诊断和修复这类的错误。其中最好的工具是Purify(Hastings和Joyce 1992)，它能检查出几乎所有的存取类错误，包括在5.3节中给出的存取错误。Purify对每个调用和存储指令都进行了检查；由于它是通过编辑目标代码来实现这样的检查的，因此它在即使没有源代码的情况下也可以使用，例如在私有库中使用它。利用源代码来捕获存取错误是另一种实现方法。例如，Austin、Breach和Sohi(1994)描述了一个系统，该系统中的“安全”指针就带有足够的信息来检查大量的存取错误。LCLint(Evans 1996)具有像PC-Lint一类工具的许多特征，可以在编译阶段检查大量潜在的内存分配错误。

Knuth(1973a)研究了所有重要的内存分配算法，并对为什么first fit算法通常比其他算法（例如查找大小最接近所要求的存储块的best fit算法）更好做了解释。在Mem_alloc中使用的first-fit算法与8.7节中描述的Kernighan和Ritchie(1988)的算法很类似。

大多数内存管理算法都有许多的变种，通常都是为某个特定的应用或分配模式设计的，以改善其性能。Quick fit(Weinstock和Wulf 1988)算法是使用最广泛的一种算法。通过观察，发现许多应用中分配的存储块只有几种不同的大小，Quick fit算法正是利用了这一点。它保存有N个空闲的存储块链表，每个对应的是使用频率最高的N种需求的大小。分配其中一种大小的存储块只需简单的从相应的链表中取出一块即可；释放其中一种大小的存储块，把它添加到相应的链表中即可。当链表为空或所要求的大小不在这N个当中时，就使用其他的方法，例如first fit等。

Grunwald和Zorn（1993）描述了一种系统，该系统产生了适用于某个特定应用的malloc和free函数的实现。他们首先用不同版本的malloc和free运行该应用程序，收集关于存储块大小、分配与释放频率等等的统计信息；然后他们把这些数据输入某个程序，该程序产生为该应用定制的malloc和free函数的源代码。这些版本的malloc和free函数通常对一小部分、专用于某种应用的存储块大小的集合使用快速拟合。

85

练习

5.1 Maguire(1993)提倡将未初始化的内存单元用一些特殊的位模式来初始化，这样可

以帮助诊断出访问未初始化的内存单元而引起的错误。一个好的位模式的特征是什么？提出一个合适的位模式，并改变Mem_alloc的校验实现来实现该初始化方法。试着找一个使用这种方法可以捕获到错误的应用程序。

5.2 一旦在代码块<*use the end of the block at bp->ptr 83*>中，某个空闲存储块的大小削减到sizeof(union align)个字节，那么它可能永远也不会满足任何需求，但是它仍然在空闲的存储块链表中。修改该程序的代码，消除这种存储块。你是否能找到一个这样的应用，使得通过对它的度量可以察觉到这种改进的效果。

5.3 大部分first fit的实现，都和8.7节中给出的Kernighan和Ritchie（1988）的实现一样，将相邻的空闲存储块合并形成一个更大的空闲存储块。Mem_alloc的校验实现并没有合并相邻的空闲存储块，因为它不会返回相同的地址两次。为Mem_alloc设计一个算法，使得该算法可以合并相邻的空闲存储块，但又不会返回相同的地址两次。

5.4 一些程序员可能认为，在Mem_free中产生Assert_Failure异常，似乎对发生存取错误的处理过于严厉，因为如果不正确的调用只是被记录然后将其忽略，那么执行是可以继续的。请实现：

```
extern void Mem_log(FILE *log);
```

如果传递给Mem_log的文件指针非空，那么它通过往log中写信息，而不是产生Assert_Failure异常来通知发生了存取错误。这些信息可以记录不正确调用及其分配的位置。例如，当Mem_free被调用，而参数指针指向的是一个已经被释放了的存储块时，它有可能给出的信息为：

```
** freeing free memory
Mem_free(0x6418) called from parse.c:461
This block is 48 bytes long and was allocated from sym.c:123
```

86

类似地，当Mem_resize被调用，而参数指针指向的是一个无效指针时，它有可能给出的信息为：

```
** resizing unallocated memory
Mem_resize(0xf7fff930,640) called from types.c:1101
```

允许Mem_log(NULL)关掉事件记录并恢复存取错误的断言失败。

5.5 校验实现拥有它报告潜在内存泄漏时所需要的所有信息。就像在本章开始部分提到的一样，一个内存泄漏就是一个已分配的存储块不被任何指针引用，因此就不可能被释放。漏洞会导致程序最终耗尽内存。它们对短期运行的程序来说可能并不算是问题，但是对长期运行的程序，例如用户接口和服务器程序等等，就是一个严重的问题了。请实现

```
extern void Mem_leak(apply(void *ptr, long size,
    const char *file, int line, void *cl), void *cl);
```

该实现调用由apply所指向的函数来分配存储块；ptr是存储块的位置，size是它分配的大小，file和line是调用分配的位置。客户调用程序可以将应用程序专用的指针

cl传递给Mem_leak，而这个指针也作为apply的最后一个参数传递给apply。Mem_leak并不知道cl是干什么用的，但是可能apply知道。apply和cl一起被称做一个闭包：它们指定了一个操作和一些该操作的上下文专用的数据。例如：

```
void inuse(void *ptr, long size,
    const char  *file, int line, void *cl) {
    FILE *log = cl;

    fprintf(log, "** memory in use at %p\n", ptr);
    fprintf(log, "This block is %ld bytes long "
        "and was allocated from %s:%d\n", size,
        file, line);
}
```

87 可能将以下的信息

```
** memory in use at 0x13428
This block is 32 bytes long and was allocated from gen.c:23
```

写入前一个练习提到的日志文件中。inuse的调用是通过把它以及日志文件的文件指针一起作为参数传给Mem_leak来实现的：

88
```
Mem_leak(inuse, log);
```

第6章　进一步内存管理

大多数malloc和free的实现，使用的内存管理算法都必须基于对象的大小。在前面一章中所使用的first fit算法就是一个例子。在一些应用中，内存单元的释放都是成组的且都发生在同一时间。图形用户接口就是这样的例子。滚动条、按钮等对象的空间是在窗口被创建的时候分配的，当窗口被销毁（destroy）时这些空间被释放。编译器是另外一个例子。例如lcc在编译某个函数时进行内存单元的分配，当它结束该函数的编译时一次性释放分配的所有内存单元。

基于对象生存期的内存管理算法通常更适合于这类应用。基于堆栈的分配就是这类分配算法的一个例子，但是只有在对象的生存期是嵌套的时候才可以使用，而实际情况常常并非如此。

本章讲述了一个内存管理接口以及它的实现，它使用的是基于实存块（arena）的算法，该算法从某个实存块空间中分配内存单元，释放时一次释放整个实存块空间。如果调用malloc，要求后面必须调用free。就像前面章节中讨论的一样，实际上很容易忘记调用free，更糟糕的是释放某个已经被释放过的对象，或是释放某个不应该释放的对象。

使用基于实存块的分配程序，不一定每次调用malloc后就一定都要对应调用free；对在实存块空间内的所有内存单元的释放，只需在最后用一个free释放调用就可以了。使用基于实存块的分配程序，分配和释放都更有效，也没有存储漏洞。而该方案一个最重要的好处是它简化了代码。合用算法（Applicative algorithm）分配新的数据结构而不是改变已存在的某个数据结构。基于场的分配程序鼓励使用简单的合用算法代替那些更复杂的算法，这些算法可能会更有效地利用空间，但是必须记住什么时候调用free。

基于实存块的方案有两个缺点：可能会使用更多的内存单元，并且可能会产生悬挂指针。如果某个对象分配到错误的实存块中，而这个实存块在程序还没处理完对象就被释放了，那么程序将可能引用未分配的存储单元或被其他可能无关的实存块重复使用的内存单元。同样也有可能将对象分配到某个实存块空间，而没有尽早地释放，这样就会形成一个存储漏洞。在实际应用中，实存块的管理是很容易的，这些问题都很少发生。

6.1　接口

Arena接口指定了两个异常以及管理实存块和在实存块中进行内存分配的函数：

```
⟨arena.h⟩≡
  #ifndef ARENA_INCLUDED
  #define ARENA_INCLUDED
  #include "except.h"
```

```
#define T Arena_T
typedef struct T *T;

extern const Except_T Arena_NewFailed;
extern const Except_T Arena_Failed;

⟨exported functions 91⟩

#undef T
#endif
```

90 实存块的创建和销毁是通过下面的语句完成的：

```
⟨exported functions 91⟩≡
  extern T    Arena_new    (void);
  extern void Arena_dispose(T *ap);
```

Arena_new创建一个新的实存块，并返回一个指向实存块的隐式指针。这些指针然后被传递给其他实存块的函数。如果Arena_new不能分配实存块空间，它将产生异常Arena_NewFailed。Arena_dispose释放与某个实存块*ap相关的内存单元，然后销毁实存块本身，并清除*ap。如果传递某个空的ap或*ap给Arena_dispose，会发生一个可检查的运行期错误。

分配函数Arena_alloc和Arena_calloc与Mem接口中同名字的函数类似，惟一的不同是它们是在实存块空间中进行分配的。

```
⟨exported functions 91⟩+≡
  extern void *Arena_alloc (T arena, long nbytes,
      const char *file, int line);
  extern void *Arena_calloc(T arena, long count,
      long nbytes, const char *file, int line);
  extern void  Arena_free  (T arena);
```

Arena_alloc在arena中分配一个大小至少为nbytes的存储块，并返回一个指向第一个字节的指针。存储块是按某个地址边界对齐的，该地址边界即使对最严格的对齐所要求的数据也是合适的。存储块中的内容未初始化。Arena_calloc在arena分配一个存储块，其大小足够存放count个元素的数组，每个元素的大小为nbytes，并返回一个指向第一个字节的指针。该存储块和Arena_alloc中一样，会调整地址边界，并初始化为0。如果count或nbytes为非正数，会产生一个可检查的运行期错误。

Arena_alloc和Arena_calloc中的最后两个参数是调用所处文件的文件名和行号。如果Arena_alloc和Arena_calloc不能分配所要求的内存单元，那么它们会产生Arena_Failed异常，并将file和line作为参数传递给Except_raise，使得异常可以报告调用的位置。如果file是个空指针，那么它们在异常Arena_Failed的实现中给出错误源的位置。

Arena_free释放arena中所有的存储单元，即释放自创建arena或上一次对其调用
91 Arena_free开始，在arena中分配的所有单元。

在该接口中传递空的T到任何例程都是可检查的运行期错误。该接口中的例程可以和

Mem接口中的例程以及其他基于malloc和free的分配程序一起使用。

6.2 实现

```
⟨arena.c⟩≡
  #include <stdlib.h>
  #include <string.h>
  #include "assert.h"
  #include "except.h"
  #include "arena.h"
  #define T Arena_T

  const Except_T Arena_NewFailed =
      { "Arena Creation Failed" };
  const Except_T Arena_Failed    =
      { "Arena Allocation Failed" };

  ⟨macros 98⟩
  ⟨types 92⟩
  ⟨data 96⟩
  ⟨functions 93⟩
```

描述的是这样一个内存存储块（chunk of memory）：

```
⟨types 92⟩≡
  struct T {
      T prev;
      char *avail;
      char *limit;
  };
```

prev字段指向该存储块的头，该存储块以如下的实存块结构开始；limit字段指向紧靠存储块末尾的下一个内存单元；avail字段指向该存储块的第一个空闲位置，从avail直到limit的空间是可以分配的。

当N不超过limit-avail时，为了分配N个字节的存储块，avail将增加N，并返回它之前的值。如果N超过limit-avail，那么调用malloc分配一个新的存储块，*arena的当前值保存在新存储块的头部。arena字段被初始化了，这样就可以描述这个新的存储块，然后继续进行分配。 92

因此实存块结构实际是一个存储块链表，该链表通过每个存储块头部的实存块结构副本中的prev字段连接起来的。图6-1显示了某个实存块的状态，其中已经分配了三个存储块；阴影部分是已分配的空间；存储块的大小不等，如果分配的空间没有完全填满存储块，那么存储块的尾部也可能是未分配的空间。

Arena_new分配并返回一个实存块结构，它的字段被设为空指针，表明是个空实存块：

```
⟨functions 93⟩≡
  T Arena_new(void) {
      T arena = malloc(sizeof (*arena));
```

```
    if (arena == NULL)
        RAISE(Arena_NewFailed);
    arena->prev = NULL;
    arena->limit = arena->avail = NULL;
    return arena;
}
```

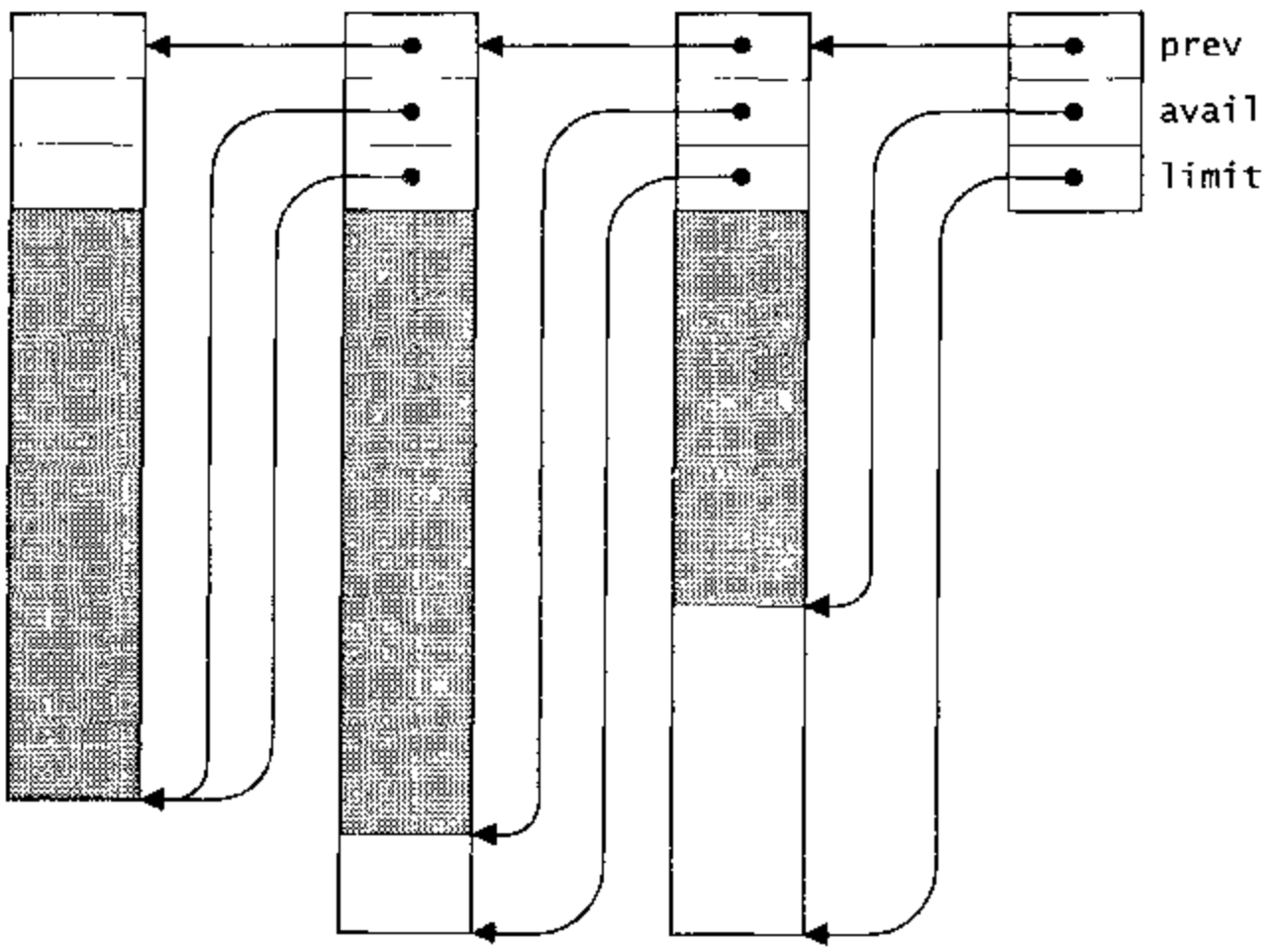


图6-1 有三个存储块的实存块

Arena_dispose调用Arena_free释放实存块内的存储块；然后它释放实存块结构本身并清除指向实存块的指针：

```
⟨functions 93⟩+≡
  void Arena_dispose(T *ap) {
      assert(ap && *ap);
      Arena_free(*ap);
      free(*ap);
      *ap = NULL;
  }
```

Arena用malloc和free代替Mem_alloc和Mem_free，因此它是独立于其他分配程序的。

大多数分配程序都是很直接的：它们将所需求的空间大小向上调整为合适的地址边界，将avail指针加上已取整的需求空间大小，并返回原来的值。

```
⟨functions 93⟩+≡
  void *Arena_alloc(T arena, long nbytes,
      const char *file, int line) {
      assert(arena);
      assert(nbytes > 0);
```

```
    ⟨round nbytes up to an alignment boundary 95⟩
    while (nbytes > arena->limit - arena->avail) {
        ⟨get a new chunk 95⟩
    }
    arena->avail += nbytes;
    return arena->avail - nbytes;
}
```

就像Mem接口中的校验实现中一样，共用体

93
~
94

```
⟨types 92⟩+≡
  union align {
      int i;
      long l;
      long *lp;
      void *p;
      void (*fp)(void);
      float f;
      double d;
      long double ld;
  };
```

的大小给出了主机上最小的对齐边界调整。它的字段都是最有可能有最严格地址边界调整要求的字段，并且用它来对nbytes向上取整：

```
⟨round nbytes up to an alignment boundary 95⟩≡
  nbytes = ((nbytes + sizeof (union align) - 1)/
      (sizeof (union align)))*(sizeof (union align));
```

对大多数调用来说，nbytes总是小于arena->limit-arena->avail的；也就是说存储块至少还有nbytes的空闲空间，因此上面Arena_alloc函数内的while循环的主体总是不执行的。如果请求不能在当前存储块中得到满足，那么就必须分配一个新的存储块。这造成了当前存储块尾部空闲空间的浪费，如图6-1中链表上的第二个存储块。

分配了一个新的存储块后，*arena的当前值保存在新的存储块的开头，并且arena的字段已被初始化，使得分配可以继续：

```
⟨get a new chunk 95⟩≡
  T ptr;
  char *limit;
  ⟨ptr ← a new chunk 96⟩
  *ptr = *arena;
  arena->avail = (char *)((union header *)ptr + 1);
  arena->limit = limit;
  arena->prev  = ptr;

⟨types 92⟩+≡
  union header {
    struct T b;
    union align a;
};
```

95

结构赋值*ptr = *arena把*arena保存在新的存储块的开始处，共用体header保证arena->avail的值是在这个新存储块中第一次分配时进行过适当边界调整的地址。

如下所示，Arena_free在产生自freechunks的空闲链表中保存了几个空闲的存储块，以减少调用malloc的次数。这个链表是通过存储块中初始实存块结构的prev字段链接起来的，而这些实存块结构中的limit字段指向紧邻存储块末尾的下一地址。nfree是链表中存储块的数目。Arena_alloc从该链表中或通过调用malloc得到一个空闲的存储块，并对局部变量limit赋值以在上面的<*get a new chunk* 95>中使用：

```
⟨data 96⟩+≡
  static T freechunks;
  static int nfree;

⟨ptr ← a new chunk 96⟩≡
  if ((ptr = freechunks) != NULL) {
      freechunks = freechunks->prev;
      nfree--;
      limit = ptr->limit;
  } else {
      long m = sizeof (union header) + nbytes + 10*1024;
      ptr = malloc(m);
      if (ptr == NULL)
          ⟨raise Arena_Failed 96⟩
      limit = (char *)ptr + m;
  }
```

如果必须分配一个新的存储块，那么需要分配存储块足够大，可以保存一个实存块结构加上nbytes空间，并还留有10K字节的可用空间。如果malloc返回空，那么分配失败，并且Arena_alloc将触发一个Arena_Failed异常：

```
⟨raise Arena_Failed 96⟩≡
  {
      if (file == NULL)
          RAISE(Arena_Failed);
      else
          Except_raise(&Arena_Failed, file, line);
  }
```

96

一旦arena指向一个新的存储块，Arena_alloc中的while循环会尝试再次分配。它仍然有可能失败：如果新的存储块来自freechunks，它就可能太小以至于不能满足需求，这就是为什么在这里用while循环来代替if语句的原因。

Arena_calloc仅仅调用Arena_alloc即可：

```
⟨functions 93⟩+≡
  void *Arena_calloc(T arena, long count, long nbytes,
      const char *file, int line) {
      void *ptr;
```

```
        assert(count > 0);
        ptr = Arena_alloc(arena, count*nbytes, file, line);
        memset(ptr, '\0', count*nbytes);
        return ptr;
    }
```

实存块的释放时，把它的存储块添加到空闲的存储块链表中即可，同时在遍历链表的时候，也把*arena恢复到初始状态：

```
⟨functions 93⟩+≡
  void Arena_free(T arena) {
        assert(arena);
        while (arena->prev) {
            struct T tmp = *arena->prev;
            ⟨free the chunk described by arena 98⟩
            *arena = tmp;
        }
        assert(arena->limit == NULL);
        assert(arena->avail == NULL);
    }
```

对tmp结构赋值，把由arena->prev指向的实存块结构中的所有字段复制到tmp中。这句赋值语句以及赋值语句*arena = tmp相当于由存储块链表形成的实存块结构栈的“出栈操作”。 97
且整个链表被遍历了，那么arena的所有字段都应该为空。

freechunks从所有场中收集空闲的存储块，因此会变得越来越大。链表的长度并不是问题，但是它存储的空闲内存单元可能会有问题。treechunks中的存储块对其他分配程序而言就像是已分配的内存单元，因此可能会使得对例如malloc等函数的调用失败。为了避免收集太多的内存空间，Arena_free使得在freechunks中存储不超过

```
⟨macros 98⟩≡
  #define THRESHOLD 10
```

个空闲存储块。一旦nfree达到了THRESHOLD，此后的存储块将通过调用free来释放：

```
⟨free the chunk described by arena 98⟩≡
  if (nfree < THRESHOLD) {
        arena->prev->prev = freechunks;
        freechunks = arena->prev;
        nfree++;
        freechunks->limit = arena->limit;
  } else
        free(arena->prev);
```

图6-2中，左边的存储块是需要被释放的。当nfree小于THRESHOLD时，存储块被添加到freechunk中。释放后的存储块显示在右边，而虚线表示的是异常代码中三个赋值语句的指针设置。

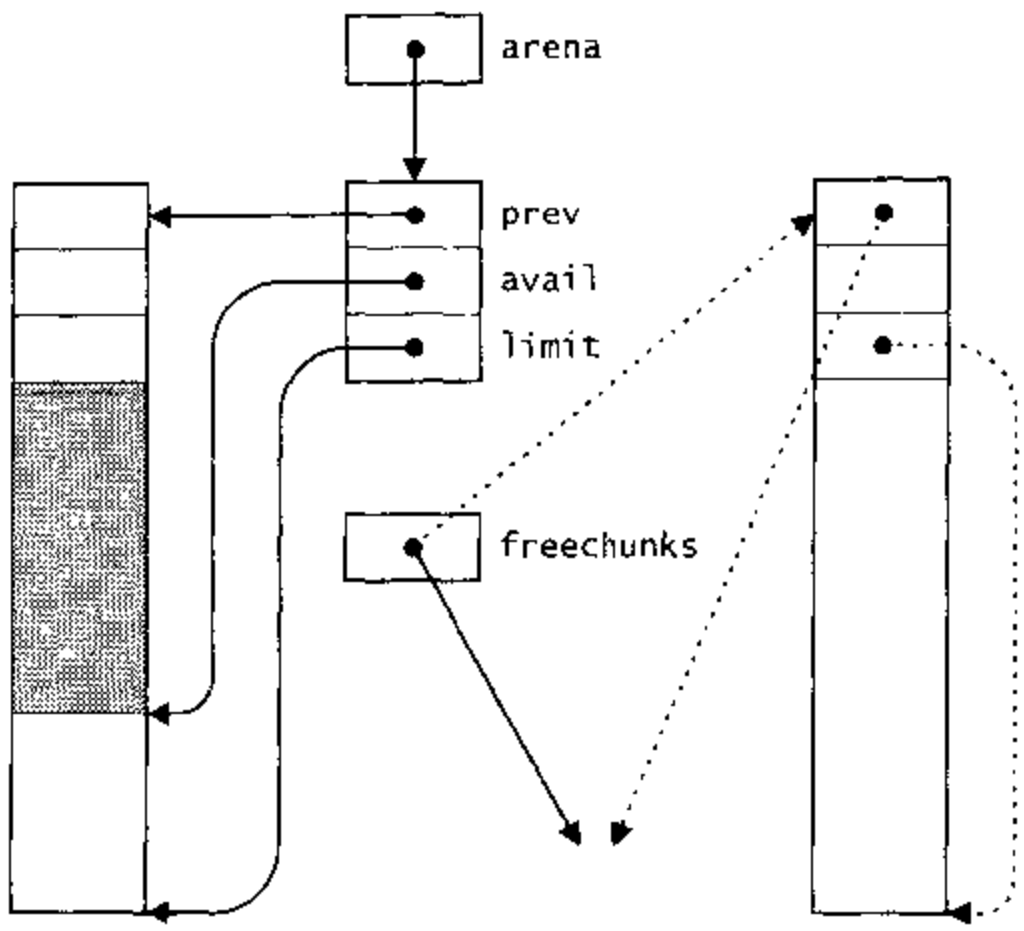


图6-2 当nfree < THRESHOLD时释放存储块

参考目浅析

基于实存块的分配程序也称做池分配程序（pool allocator），曾被多次介绍过。实存块的分配程序（Hanson 1990）最初是为了在lcc（Fraser和Hanson 1995）中使用而开发的。lcc的分配程序比实存块的分配程序稍简单一些：它的场是静态分配的，并且它的释放程序也是调用free完成的。在它最初的版本中，分配是通过直接处理实存块结构的宏指令来完成的，并且只有当需要新的存储块时才调用函数。

Barrett和Zorn（1993）叙述了如何自动选择正确的实存块。试验表明：分配点的执行路径是很好的可用来预测在该位置分配的存储块生存期的信息，该信息包括调用链以及分配点的地址，可以利用它们来选择几个应用程序专用实存块空间中的一个。

Vmalloc(Vo 1996)是个更通用的分配程序，可以用来实现Mem和Arena接口。Vmalloc允许客户调用程序将内存组织到一个区域中，并提供了管理每个区域内空间的函数。Vmalloc库函数包括一个malloc接口的实现，该实现提供了Mem接口的校验实现中类似的内存检查，并且这些检查可以通过设置环境变量来控制。

基于实存块的分配程序将许多显式释放语句压缩成一个。无用单元收集程序（Garbage collector）更进了一步：它们避免了所有的显式释放。在使用无用单元收集程序的程序设计语言中，程序员几乎可以忽略存储单元的分配，且存储分配错误也不可能发生，这种策略的优点怎么讲都不夸张。

98
~
99

使用无用单元收集程序，空间的回收是在需要的时候自动完成，通常是在分配要求不能满足的时候自动完成。无用单元收集程序能找到程序变量所引用的所有存储块，以及由存储

块中的字段引用的所有存储块等等。这些都是可以访问的存储块，而剩下的都是不可访问的，并且可以再使用。关于无用单元收集程序有大量的文献：Appel(1991)是个简短的综述，强调了最近的一些算法；而Knuth(1973a)和Cohen（1981）更深入地讲述了旧一点的算法。

为了找到可访问的存储块，大多数分配程序必须知道哪些变量指向存储块以及块中的哪些字段指向其他的存储块。在拥有足够的编译或运行时数据的程序设计语言中，回收程序通常用来提供必要的信息，例如LISP、Icon、SmallTalk、ML以及Modula-3。保留收集器(Boehm和Weiser 1988)可以处理不能提供足够类型信息的程序设计语言，像C和C++。他们提出了一个类似于指针的适应于任何边界调整的位模式，并将模式当作一个指针，其指向的存储块是可访问的。因此保留收集器会把一些不可访问的存储块也当作是可访问，这使得它经常处于繁忙状态，但是没有别的选择，只能尽量将可访问的存储块集合估算得很大。尽管保留收集器有这个明显的缺点，但是它在某些程序（Zorn 1993）中仍然工作得相当好。

练习

6.1 Arena_alloc只查找由arena描述的存储块。如果在那个存储块中没有足够的空闲空间，即使链表中后面的存储块中有足够的空间，它也分配一个新的存储块。修改Arena_alloc使得它在有某个存储块有足够的空间的情况，分配已存在的存储块中的空间，并度量一下这样做有什么好处。你是否能找到一个应用程序，它的内存使用量由于这样的修改而得到明显减少？

6.2 当Arena_alloc需要一个新的存储块时，如果空闲链表不为空的话，它从链表的表头取出一块进行分配。另一种更好的选择是找到满足要求的最大空闲存储块，如果freechunks中没有合适的存储块，那么分配一个新的存储块。记录freechunks中最大的存储块可以避免无意义的遍历。如果这么做，Arena_alloc中的while循环就可以用if语句来代替。实现这种方案并度量其性能。它是否会使Arena_alloc的运行速度明显地变慢？它是否可以更有效地使用内存？

100

6.3 将THRESHOLD设置为10，意味着空闲存储块链表永远不会有超过100K个字节的内存空间，因为Arena_alloc分配的存储块至少为10K字节。为Arena_alloc和Arena_free设计一种方法来监测分配和释放的模式，并根据这些模式动态地计算THRESHOLD。这样做的目的是使得空闲存储块链表尽可能地小，并且调用malloc的次数最少。

6.4 解释为什么Arena接口不支持函数

```
void *Arena_resize(void **ptr, long nbytes,
    const char *file, int line)
```

就像Mem_resize一样，该函数可以将ptr所指向的存储块的大小变为nbytes个字节，并返回一个指向已修改了大小的存储块的指针。该存储块也可能与原存储块位于同一个实存块空间中（但不一定在同一个块中），就像ptr指定的存储块一样。你如何

修改实现以支持这个函数？你修改的实现中有什么可检查的运行期错误？

6.5 在堆栈分配程序中，一个分配操作将新的内存空间压入给定堆栈的栈顶，并返回其第一个字节的地址。标记堆栈返回的该栈当前高度的编码值，而释放将使堆栈的栈顶元素推出，使堆栈返回到之前被标记的高度。为堆栈分配程序设计并实现一个接口。你能给出哪些可检查的运行期错误来捕获空间释放的错误？举例说明这类错误，如释放高于当前堆栈栈顶的空间，或释放某个已经被释放、而后来又被重新分配了的空间。

6.6 如果程序中有不止一个内存分配接口，就会面临这样一个问题，在这些接口之间做出选择；但是对一个特定的应用程序来说，你可能并不知道哪个接口是最好的。设计并实现一个单一的接口，它支持两种分配程序。例如该接口可能提供像Mem_alloc的分配函数，但是“分配环境”中的操作可以用其他函数来改变。该环境可以指定内存管理的细节，例如使用的是什么分配程序，如果使用的是基于实存块的分配，也可以指定使用的是哪个实存块空间。其他函数也可以将当前环境压入内部栈中，然后建立一个新的环境，最后进行出栈操作以恢复原来的环境。在你的设计中调研这些以及其他方法。

101

102

第7章 链 表

链表是由零个或多个指针组成的序列，零个指针的链表是空链表，链表中指针的数目就是链表的长度。几乎所有的非常规应用程序都以某种形式使用了链表。正是由于链表在程序中如此普及，所以一些程序设计语言把它们作为内嵌类型，像LISP、Scheme和ML都是最著名的例子。

链表很容易实现，因此程序员通常对每个正在开发的应用程序进行重新实现。尽管大多数为应用程序专门设计的链表接口都有很多的相似之处，但还是没有被广泛接受的标准接口。以下要说明的抽象数据类型List提供了许多特定应用设计的接口都会实现的功能。第11章中描述的序列是链表的另一种表示方法。

7.1 接口

完整的List接口代码为：

⟨*list.h*⟩≡

```
#ifndef LIST_INCLUDED
#define LIST_INCLUDED

#define T List_T
typedef struct T *T;

struct T {
    T rest;
    void *first;
};

extern T      List_append (T list, T tail);
extern T      List_copy   (T list);
extern T      List_list   (void *x, ...);
extern T      List_pop    (T list, void **x);
extern T      List_push   (T list, void *x);
extern T      List_reverse(T list);
extern int    List_length (T list);
extern void   List_free   (T *list);
extern void   List_map    (T list,
    void apply(void **x, void *cl), void *cl);
extern void **List_toArray(T list, void *end);

#undef T
#endif
```

List_T是一个指向List_T结构的指针。大多数ADT都隐藏了它们的类型表示细节。而List则揭示了这些细节，因为对这个特殊的ADT，隐藏这些细节所带来的复杂性远超过这样做能带来的好处。

List_T的表示形式非常相似，很难想像会存在许多其他的表示，它们的实现可以提供足够重要的优点来证明，隐藏链表元素是一个由两个字段组成的结构体这一事实是值得的。本章的练习中讨论了其中的一些表示方法。

展现List_T的表示可以简化接口以及它的几种使用方式。例如List_T类型的变量可以静态地定义和初始化，这对在编译阶段构建链表很有用，可以避免分配操作。另外，也可以在其他的结构中嵌入List_T结构。一个空的List_T就是一个空的链表，这也是空表最自然的表示，并且函数也不需要访问first和rest字段。

接口中的所有例程都可以接收一个空T作为链表参数，并将它解释成空的链表。

List_list创建并返回一个链表。它随*N*个非空指针加一个空指针一起调用，创建一个有*N*个节点的链表，这*N*个节点的first字段存有这*N*个非空指针，而它们的rest字段为空。例如，
104 赋值语句：

```
List_T p1, p2;
p1 = List_list(NULL);
p2 = List_list("Atom", "Mem", "Arena", "List", NULL);
```

返回空链表以及一个有4个分别存有指向字符串“Atom”、“Mem”、“Arena”和“List”的节点的链表。List_list可能产生异常Mem_Failed。

List_list假设其参数链表的可变部分传递的指针都是空的，而接口也没有提供必要的隐式转换的原型，因此程序员必须在传递其他不同于字符和空指针的参数作为第二个或后面的参数时进行类型转换。例如，为了构建一个有4个单一元素的链表，分别存有字符串“Atom”、“Mem”、“Arena”和“List”，正确的调用是：

```
p = List_list(List_list("Atom",  NULL),
      (void *)List_list("Mem",   NULL),
      (void *)List_list("Arena", NULL),
      (void *)List_list("List",  NULL), NULL);
```

如果忽略在这个例子中所示的转换将会导致不可检查的运行期错误。这样的转换是可变长度参数链表的缺点之一。

List_push(T list,void *x)在list的表头添加一个新的节点，存储x，并返回新的链表。List_push可能会产生异常Mem_Failed。List_push是另外一种构建新的链表的方法；例如语句：

```
p2 = List_push(NULL, "List");
p2 = List_push(p2,   "Arena");
p2 = List_push(p2,   "Mem");
p2 = List_push(p2,   "Atom");
```

创建了与上面相同的链表p2。

给定一个非空链表，List_pop(T list,void **x)将链表第一个节点的first字段赋值给*x，

如果x不为空，那么删除并释放第一个接点、并返回剩下的链表。如果给定的是一个空链表，那么List_pop仅仅返回空链表，对*x不做任何改变。

List_append(T list,T tail)在一个链表中添加一个链表：它将tail赋值给list链表的最后一个节点的rest字段。如果list为空，它返回tail。因此语句：

```
p2 = List_append(p2, List_list("Except", NULL));
```

105

将p2赋值为5个元素的链表，这个5个元素的链表是通过向前面创建的4个元素的链表中添加一个存有Except元素的链表形成的。

List_reverse反转了它链表参数中节点的次序并返回结果链表。例如：

```
p2 = List_reverse(p2);
```

返回一个存有“Except”、“List”、“Arena”、“Mem”和“Atom”的链表。

到目前为止描述的大部分例程都具有破坏性（destructive）或非适用性（nonapplicative）：它们可以改变传递给它们的链表并返回结果链表。List_copy是一个适用（applicative）的函数：它复制它的参数，并返回复制结果。因此，执行下列语句之后：

```
List_T p3 = List_reverse(List_copy(p2));
```

p3是一个“Atom”、“Mem”、“Arena”、“List”和“Except”的链表；p2没有改变。List_copy可能会产生异常Mem_Failed。

List_length返回其参数链表中的节点数。

List_free接收一个指向某个T的指针为参数。如果*list不为空，那么List_free释放*list中所有的节点，并把*list设为空。如果*list为空，那么List_free将不做任何操作。如果传递一个空指针给函数List_free，将产生可检查的运行期错误。

List_map对list中的每个节点调用apply所指向的函数。客户调用程序可以传递某个应用程序专用的指针cl给List_map，并且该指针也传给*apply，作为其第二个参数。对list中的每个节点，*apply连带一个指向节点的first字段的指针和cl指针一起被调用，因为*apply是和指向first字段的指针一起被调用的，所以它可以改变first字段的值。总的来说，apply和cl被称为一个闭包（closure）或回调（callback）：它们一起规定了一个操作，以及该操作的一些上下文专用的数据。例如，给定函数mkatom，

```
void mkatom(void **x, void *cl) {
    char **str = (char **)x;
    FILE *fp = cl;

    *str = Atom_string(*str);
    fprintf(fp, "%s\n", *str);
}
```

而调用List_map(p3,mkatom,stderr)用等效的原子节点代替了p3中的字符串，并在错误输出窗口中输出结果：

106

```
Atom
Mem
Arena
List
Except
```

另一个例子是函数applyFree：

```
void applyFree(void **ptr, void *cl) {
    FREE(*ptr);
}
```

该函数用来在链表本身被释放之前，释放出链表中的first字段所指向的空间。例如：

```
List_T names;
...
List_map(names, applyFree, NULL);
List_free(&names);
```

释放了链表names中的数据，然后释放节点本身。如果apply改变了list，这将是一个不可检查的运行期错误。

给定一个有N个值的链表，List_toArray(T List,void *end)创建一个数组，它的元素0到N–1分别存储链表中first字段中的N个值，而第N个元素存储值end，值end通常是一个空指针。List_toArray返回一个指向该数组第一个元素的指针。例如p3中的元素可以以排好的顺序打印出来：

```
int i;
char **array = (char **)List_toArray(p3, NULL);
qsort((void **)array, List_length(p3), sizeof (*array),
    (int (*)(const void *, const void *))compare);
for (i = 0; array[i]; i++)
    printf("%s\n", array[i]);
FREE(array);
```

就如这个例子所示的一样，客户调用程序必须释放出List_toArray返回的数组。如果链表为空，那么List_toArray返回一个元素的数组。List_toArray也可能产生异常Mem_Failed。

107 compare以及它和标准库函数qsort的使用在8.2节中给出。

7.2 实现

```
⟨list.c⟩≡
  #include <stdarg.h>
  #include <stddef.h>
  #include "assert.h"
  #include "mem.h"
  #include "list.h"

  #define T List_T

  ⟨functions 108⟩
```

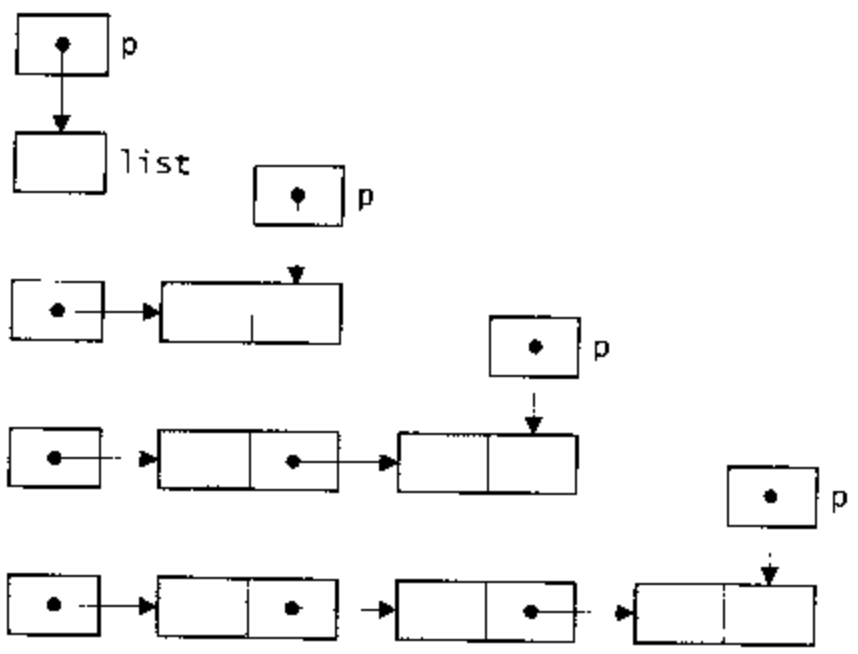
List_push是List接口中最简单的函数。它分配一个节点，初始化，然后返回一个指向它的指针：

```
⟨functions 108⟩≡
  T List_push(T list, void *x) {
      T p;

      NEW(p);
      p->first = x;
      p->rest  = list;
      return p;
  }
```

其他创建链表的函数：如List_list，稍复杂一些，因为它必须处理可变数目的参数，并且必须为每个非空的指针参数添加一个新的节点到正在变化的链表中。为了做到这些，它使用了一个指针，它所指向的指针指向新的节点且应该被赋过值：

```
⟨functions 108⟩+≡
  T List_list(void *x, ...) {
      va_list ap;
      T list, *p = &list;

      va_start(ap, x);
      for ( ; x; x = va_arg(ap, void *)) {
          NEW(*p);
          (*p)->first = x;
          p = &(*p)->rest;
      }
      *p = NULL;
      va_end(ap);
      return list;
  }
```


p开始指向list，因此指向第一个节点的指针被赋值给list。此后，p指向链表中最后一个节点的rest字段，因此，对*p的赋值就向链表中添加了一个节点。下面的图显示了p的初始化的效果以及当List_list构造了一个有三个节点的链表时，for循环体中的语句对p的效果。



每次循环都将对下一个指针参数赋给x，当它到达第一个空指针参数的时候停止，空指针也可能是x的初始值。这个习惯用法保证了List_list(NULL)返回的是一个空链表，即一个空指针。

List_list使用了指向指针的指针：List_T *s。这是许多链表操作算法的典型用法。它用了一种简明的机制来处理可能为空的链表中的初始节点，以及一个非空链表的内部节点。

109 List_append说明了这种习惯用法的另一个范例：

```
⟨functions 108⟩+≡
  T List_append(T list, T tail) {
      T *p = &list;

      while (*p)
          p = &(*p)->rest;
      *p = tail;
      return list;
  }
```

List_append中的指针p一直沿着list前进，直到它指向一个list末端的空指针，tail就应该被赋值为空指针。如果list本身就是空指针，那么p在指向list的时后就结束了，这正是添加tail到一个空链表所希望达到的效果。

List_copy是List接口中最后一个使用指向指针的指针的习惯用法的函数：

```
⟨functions 108⟩+≡
  T List_copy(T list) {
      T head, *p = &head;

      for ( ; list; list = list->rest) {
          NEW(*p);
          (*p)->first = list->first;
          p = &(*p)->rest;
      }
      *p = NULL;
      return head;
  }
```

指向指针的指针并不能简化List_pop或List_reverse，因此对这两个函数，也许用更明显的实现就已经足够了。List_pop删除非空链表中的第一个节点，并返回删除后新的链表，或仅仅返回一个空的链表：

```
⟨functions 108⟩+≡
  T List_pop(T list, void **x) {
      if (list) {
          T head = list->rest;
          if (x)
110           *x = list->first;
        FREE(list);
        return head;
```

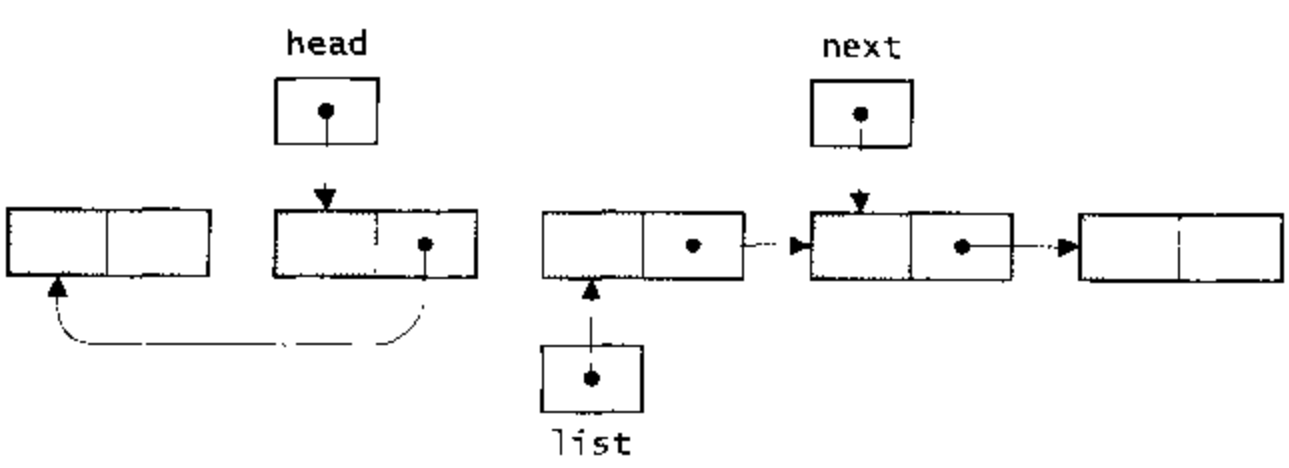
```
        } else
            return list;
    }
```

如果x非空，那么*x在链表中第一个节点被删除之前，被赋值为该节点的first字段。注意：List_pop必须在删除list指向的节点之前保存list->rest。

List_reverse有两个指针沿着链表前进，list和next，并在前进时使用这两个指针逆转链表；new总是指向已逆转的链表的第一个接点：

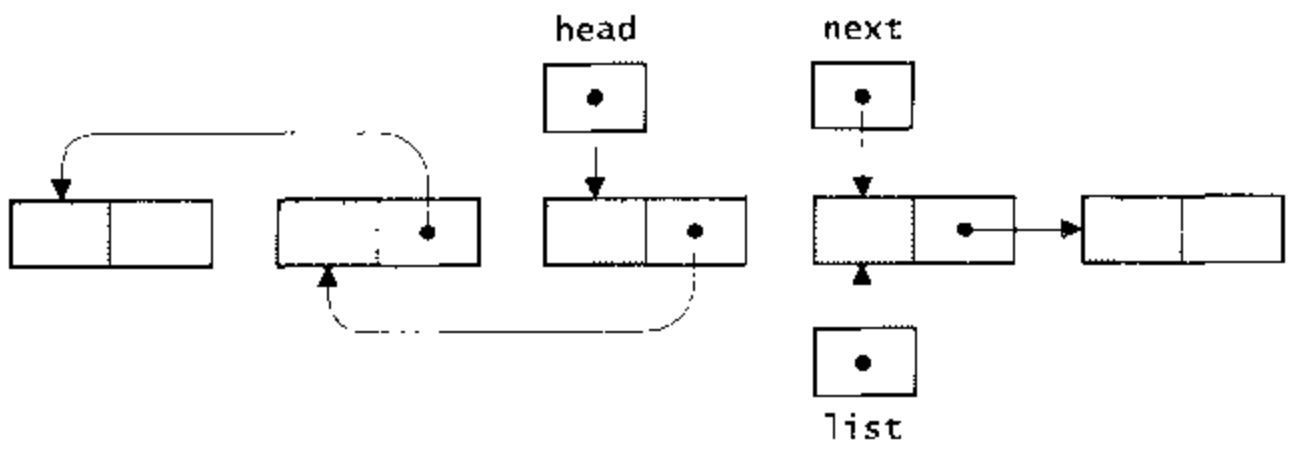
```
⟨functions 108⟩+≡
  T List_reverse(T list) {
      T head = NULL, next;

      for ( ; list; list = next) {
          next = list->rest;
          list->rest = head;
          head = list;
      }
      return head;
  }
```

下面的图说明了每次循环迭代，在循环体第一个语句执行之后的状态；对next的赋值，针对的是链表中的第三个元素。



111

next指向list的后继，当list指向最后一个节点时，next指向空值，而head指向已逆转的链表，它开始指向list的前驱；当list指向第一个节点时，head指向空值。循环体中的第二个或第三个语句将由list指向的节点拼接到head链表的表头，而递增表达式list = next把list向它的后继推进，其后链表的指针如下图所示：



下一次通过循环体时，next将再次前进。

List_length遍历list链表，对它的节点进行计数，而List_free遍历list链表，释放其每个节点：

```
⟨functions 108⟩+≡
  int List_length(T list) {
      int n;

      for (n = 0; list; list = list->rest)
          n++;
      return n;
  }

  void List_free(T *list) {
      T next;

      assert(list);
      for ( ; *list; *list = next) {
          next = (*list)->rest;
          FREE(*list);
      }
  }
```

112

List_map听起来很复杂，其实它很简单，因为闭包函数已经完成了所有的工作。List_map只需要遍历list链表，使用指向每个节点的first字段的指针以及客户调用程序专用的指针cl来调用闭包函数：

```
⟨functions 108⟩+≡
  void List_map(T list,
       void apply(void **x, void *cl), void *cl) {
      assert(apply);
      for ( ; list; list = list->rest)
          apply(&list->first, cl);
  }
```

List_toArray分配一个有$N+1$个元素的数组来保存N个元素链表的指针，并把这些指针复制到数组中：

```
⟨functions 108⟩+≡
  void **List_toArray(T list, void *end) {
      int i, n = List_length(list);
      void **array = ALLOC((n + 1)*sizeof (*array));

      for (i = 0; i < n; i++) {
          array[i] = list->first;
          list = list->rest;
      }
      array[i] = end;
      return array;
  }
```

为空链表分配一仅含一个元素的数组看上去是种浪费，但是这样做的目的是使得List_toArray

总是可以返回一个指向数组的非空指针，因此客户调用程序不需要检查返回值是否为空指针。

参考书目浅析

Knuth（1973a）描述了所有的处理单链表的重要算法，类似于List接口提供的这些算法，以及由Ring提供的处理双链表的所有重要的算法（第12章中给出了说明）。

113

在诸如LISP和Scheme的链表处理语言以及诸如ML（Ullman 1994）的函数语言中都使用了链表。Abelson和Sussman（1985）是众多讲述链表是如何被用来解决几乎任何问题的教材中的一本，它使用的语言是Scheme。

练习

7.1 设计一个链表ADT，隐藏链表的表示，并且不使用空指针来表示空链表。先设计该ADT的接口，然后完成一个实现。其中一种方法是设List_T是一个隐式指针，指向某个链表的表头，该表头保存一个指向链表本身的指针或者一个同时指向链表中第一个元素和最后一个元素的指针。链表的表头也可能存有链表的长度。

7.2 重写List_list、List_append和List_copy，不使用指向指针的指针。

7.3 使用指向指针的指针重写List_reverse

7.4 List_append是一个在许多应用程序中使用频率最高的链表操作，它必须遍历到链表的末尾，因此对N个元素的链表，它花费的时间是$O(N)$。循环链接表是单链表的另一种表示方法。Mem接口中校验实现的未分配链表就是一个循环链接表。在循环链接表中最后一个节点的rest字段指向第一个节点，而链表本身由一个指向最后一个节点的指针表示。因此，在常数时间内既可以访问到第一个节点也可以访问到最后一个节点，并且添加一个节点到循环链表也可以在常数时间内完成。设计一个链表ADT的接口，使用循环链接表，用隐藏或显式表示的接口来做实验。

114

第8章 表 格

一个关联表格（associative table）是一个关键字-值对的集合。它很像是数组，惟一的不同是它的下标可以是任何类型的值。在许多应用程序中都使用了表格。例如编译器就保存有符号表，该符号表将名字映射成那些名字的一组属性。一些窗口系统也保存了一个表格，它把窗口标题映射到与窗口关联的某类数据结构中。文档预备系统使用表格来表示索引：例如，索引可能是表格，表格中的关键字是一个字符的字符串——索引的每个区段一个关键字——而关键字的值是另外的表格，其关键字是索引项本身的字符串，其值是页码的列表。

表格有很多种使用方法，如果给每种使用举一个例子可能得花去一整章的笔墨。Table接口就是为了能够在这许多的用法中被使用而设计的。它保存有关键字-值对，但是它从来不检查关键字本身；只有客户调用程序通过传递给Table中的例程的函数来检查关键字。8.2节中给出了一个典型的Table客户调用程序，一个将其输入中出现的单词数目打印出来的程序。该程序wf同时也使用了Atom和Mem接口。

8.1 接口

Table接口用隐式指针类型来表示一个关联表格：

```
⟨table.h⟩≡
  #ifndef TABLE_INCLUDED
  #define TABLE_INCLUDED
  #define T Table_T
  typedef struct T *T;

  ⟨exported functions 116⟩

  #undef T
  #endif
```


导出函数对Table_T进行分配和回收，从这些表格中添加或移出关键字-值对以及在表格中访问关键字-值对。如果把一个空的Table_T或空的关键字传递给该接口中的任何函数，都会产生可检查的运行期错误。

Table_T是通过以下函数来进行分配和回收的：

```
⟨exported functions 116⟩≡
  extern T    Table_new (int hint,
      int cmp(const void *x, const void *y),
      unsigned hash(const void *key));
  extern void Table_free(T *table);
```

Table_new的第一个参数hint是对新表中期望存储的记录项数目的估计。不管hint的值是多少，所有的表格都可以存储任意数目的记录项，但是如果hint的值很精确的话，有可能会提高性能。如果hint的值为负数，也会产生可检查的运行期错误。函数cmp和hash处理客户调用程序指定的关键字。给定两个关键字x和y，cmp(x,y)必须返回一个小于0、等于0或大于0的整数，分别代表x小于y、x等于y或x大于y。标准的库函数strcmp就是一个以字符串为关键字的比较函数的例子。hash函数必须返回一个key的散列值；如果cmp(x,y)返回0，那么hash(x)必须等于hash(y)。每个表格都可以有它自己的hash和cmp函数。

原子通常被当作关键字来使用，因此如果hash是个空的函数指针，那么新表中的关键字将被当作原子，且Table的实现提供了一个合适的hash函数。同样地，如果cmp是一个空的函数指针，那么关键字也被当作原子，且如果$x = y$，关键字x和y相等。

Table_new可能产生异常Mem_Failed。

Table_new的参数：一个大小提示、一个hash函数以及一个比较函数，提供的信息比大多数实现所需要的更多。例如，在8.3节中描述的散列表的实现，它需要一个比较函数，只用来测试相等与否，而使用树的实现就并不需要大小提示或hash函数。这种复杂性就是一个允许多种实现的设计的代价，而这也是设计好的接口很难的原因之一。

116

Table_free释放*table，并把它设为空指针。如果table或*table为空，也是可检查的运行期错误。Table_free并不释放关键字或其值，它们在Table_map中被释放。

函数：

```
⟨exported functions 116⟩+≡
  extern int   Table_length(T table);
  extern void *Table_put   (T table, const void *key,
      void *value);
  extern void *Table_get   (T table, const void *key);
  extern void *Table_remove(T table, const void *key);
```

返回表格中关键字的数目、添加一个新的关键字-值对或改变一个已存在的关键字-值对的值、删除与某关键字关联的值以及删除一个关键字-值对。

Table_length返回在table中关键字-值对的数目。

Table_put添加一个由key和value给定的关键字-值对到table中。如果table已经存有key，那么用value覆盖先前的值，并且Table_put返回先前的值。否则，key和value的值被添加到table中，表格的记录项数加1，并且Table_put返回一个空指针。Table_put可能产生异常Mem_Failed。

Table_get在table中查找关键字key。如果找到，返回与该关键字关联的值。如果table不存在该关键字，Table_get返回空指针。注意：如果table本身就存有空指针的值，那么返回空指针就有可能会产生歧义。

Table_remove也在table中查找关键字key。如果找到，就从表格中删除该关键字-值对，因此表格的记录项数减1，并且返回被删除的值。如果table不存在关键字key，那么Table_remove对table不做任何操作，并返回空指针。

函数：

```
⟨exported functions 116⟩+≡
  extern void   Table_map    (T table,
      void apply(const void *key, void **value, void *cl),
      void *cl);
  extern void **Table_toArray(T table, void *end);
```

117

访问关键字-值对并把它们收集到一个数组中。Table_map以不确定的次序对table中的每个关键字-值对调用apply指向的函数。apply和cl指定了一个闭包：客户调用程序可以传递一个应用程序专用的指针cl给Table_map，并且这个指针在每次调用时又传递给apply。对table中的每个关键字-值对，apply是同它的关键字、指向它的值的指针以及cl一起调用。因为apply是同指向它的值的指针一起调用的，因此，它可以改变它的值。Table_map同时也可以在释放表格之前用来释放关键字或值。例如，假设关键字是原子，函数

```
static void vfree(const void *key, void **value,
    void *cl) {
    FREE(*value);
}
```

只释放了值，因此

```
Table_map(table, vfree, NULL);
Table_free(&table);
```

释放了表格中的值，然后释放表格本身。

如果在apply中，通过调用Table_put或Table_remove来改变table的内容，可能产生一个可检查的运行期错误。

对于N个关键字-值对的表格，Table_toArray构建一个有$2N+1$个元素的数组并返回一个指向其第一个元素的指针。关键字和值交替出现，关键字出现在奇数号元素，而它们关联的值在接下来的偶数号元素中。最后一个偶数号元素，即索引号$2N$赋值end，通常是个空指针。数组中关键字-值对的次序是不确定的。8.2节中描述的程序说明了Table_toArray的使用。

Table_toArray可能会产生异常Mem_Failed，而且客户调用程序必须释放它返回的数组。

8.2　例子：单词频率

wf列出了在命名文件列表或标准输入（如果没有指定文件）中出现的每个单词的次数。例如：

118

```
% wf table.c mem.c
table.c:
3   apply
7   array
13  assert
9   binding
18  book
2   break
```

```
10  buckets
...
4   y
mem.c:
1   allocation
7   assert
12  book
1   stdlib
9   void
...
```

就和这结果显示的一样，每个文件中的单词都按字母顺序列出，在单词之前还有它们在文件中出现的次数。对wf来说，一个单词是一个以字母开头，后0个或多于0个的字母或下划线组成的，大小写无关。

更通用地说，一个单词以first集合中的某个字符开头，由rest集合中0个或更多的字符组成。这种形式的单词由getword识别，该函数是1.1节中描述的double中的getword的常用定义。在本书中它就够用了，在它自己的接口中被单独打包成：

```
⟨getword.h⟩≡
  #include <stdio.h>

  extern int getword(FILE *fp, char *buf, int size,
      int first(int c), int rest(int c));
```

getword从fp打开的文件中读取下一个单词，把它当作一个以null结束的字符串存储在buf[0..size-1]中，并返回1。当它达到文件末尾，而没找到单词，它返回0。函数first和rest测试某个字符是否是first和rest集合中的成员。一个单词是字符的相邻序列；它以字符开始，
119 对该字符first函数返回一个非0值，外加由函数rest返回非0值的字符组成。如果一个单词长于size-2个字符，那么超过的字符被截断。size必须大于1，并且fp、buf、first以及rest必须不为空。

```
⟨getword.c⟩≡
  #include <ctype.h>
  #include <string.h>
  #include <stdio.h>
  #include "assert.h"
  #include "getword.h"

  int getword(FILE *fp, char *buf, int size,
      int first(int c), int rest(int c)) {
      int i = 0, c;

      assert(fp && buf && size > 1 && first && rest);
      c = getc(fp);
      for ( ; c != EOF; c = getc(fp))
          if (first(c)) {
              ⟨store c in buf if it fits 120⟩
              c = getc(fp);
```

```
            break;
        }
    for ( ; c != EOF && rest(c); c = getc(fp))
        ⟨store c in buf if it fits 120⟩
    if (i < size)
        buf[i] = 0;
    else
        buf[size-1] = 0;
    if (c != EOF)
        ungetc(c, fp);
    return i > 0;
}
```

⟨*store* c *in* buf *if it fits* 120⟩≡

```
{
    if (i < size - 1)
        buf[i++] = c;
}
```

120

这个版本的getword比double中的复杂一些，因为这个版本的getword必须在当某个字符在first集合中，而又不在rest集合中时才工作。如果first函数返回值非0，那么该字符存在buf中，并且只有接下来的字符才传给rest函数。

wf的主函数main处理wf的参数，此参数为文件命名。main打开每一个命名文件，然后以文件指针和文件名为参数调用wf：

⟨*wf functions* 121⟩≡

```
int main(int argc, char *argv[]) {
    int i;

    for (i = 1; i < argc; i++) {
        FILE *fp = fopen(argv[i], "r");
        if (fp == NULL) {
            fprintf(stderr, "%s: can't open '%s' (%s)\n",
                argv[0], argv[i], strerror(errno));
            return EXIT_FAILURE;
        } else {
            wf(argv[i], fp);
            fclose(fp);
        }
    }
    if (argc == 1) wf(NULL, stdin);
    return EXIT_SUCCESS;
}
```

⟨*wf includes* 121⟩≡

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
```

如果没有任何参数，那么main以空文件名和标准输入文件指针作参数调用wf。空文件名告诉

wf不用打印出文件的名字。

wf使用表格来存储单词和它们出现的次数。每个单词都换成小写，转换成原子，作为关键字使用。使用原子使得wf采用默认表格的hash函数和比较函数。虽然wf的值都是指针，但是它仍然需要把一个整数计数与每个关键字关联起来。因此它分配一个空间给计数器，并在
121 表格中存储指向该区域的指针。

```
⟨wf functions 121⟩+≡
  void wf(char *name, FILE *fp) {
      Table_T table = Table_new(0, NULL, NULL);
      char buf[128];

      while (getword(fp, buf, sizeof buf, first, rest)) {
          const char *word;
          int i, *count;
          for (i = 0; buf[i] != '\0'; i++)
              buf[i] = tolower(buf[i]);
          word = Atom_string(buf);
          count = Table_get(table, word);
          if (count)
              (*count)++;
          else {
              NEW(count);
              *count = 1;
              Table_put(table, word, count);
          }
      }
      if (name)
          printf("%s:\n", name);
      { ⟨print the words 123⟩ }
      ⟨deallocate the entries and table 124⟩
  }

⟨wf includes 121⟩+≡
  #include <ctype.h>
  #include "atom.h"
  #include "table.h"
  #include "mem.h"
  #include "getword.h"

⟨wf prototypes 122⟩≡
  void wf(char *, FILE *);
```

count是一个指向一个整数的指针。如果Table_get返回空，那么单词不在table中，因此wf为计数器分配空间，初始化为1，说明这是该单词的第一次出现，并把它添加到表格中。当
122 Table_get返回非空指针时，表达式(*count)++将该指针指向的整数加1。该表达式与*count++有很大的不同，后者将count加1而不是它所指向的整数。

first和rest集合中的成员是用标准头文件ctype.h中使用谓词定义的同名函数来测试的：

```
⟨wf functions 121⟩+≡
  int first(int c) {
      return isalpha(c);
  }

  int rest(int c) {
      return isalpha(c) || c == '_';
  }
```

```
⟨wf prototypes 122⟩+≡
  int first(int c);
  int rest (int c);
```

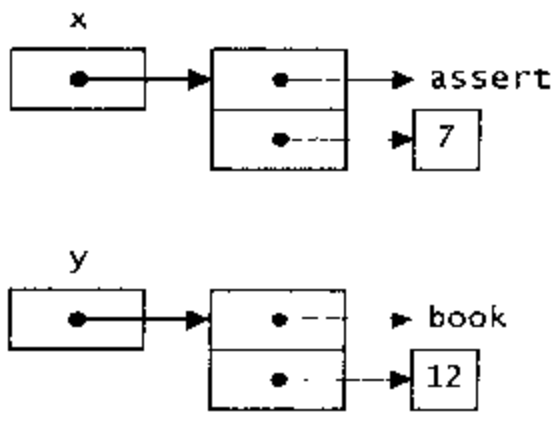
一旦wf已经读出了所有的单词，那么它必须将它排序并打印出来。标准C函数库中的排序函数qsort被用来排序数组。因此如果wf告诉qsort数组中的关键字-值对应该被当做一个单一的元素处理，那么它就可以排序Table_toArray返回的数组。然后wf可以通过遍历数组将单词和它们的计数打印出来：

```
⟨print the words 123⟩≡
  int i;
  void **array = Table_toArray(table, NULL);
  qsort(array, Table_length(table), 2*sizeof (*array),
      compare);
  for (i = 0; array[i]; i += 2)
      printf("%d\t%s\n", *(int *)array[i+1],
          (char *)array[i]);
  FREE(array);
```

qsort接收4个参数：数组、元素的个数、每个元素所占的字节数以及用来比较两个元素的函数。为了把N个关键字-值对当作一个单一的元素来处理，wf告诉qsort有N个元素，且每个元素的空间由两个指针占用。

qsort用指向元素的指针来调用比较函数。每个元素本身用到两个指针，一个指向单词，一个指向单词的计数，因此以两个指向字符的指针为参数来调用比较函数。例如，mem.c中的assert与book进行比较的时候，参数x和y如下：

123



比较函数也可以调用strcmp来比较单词：

```
⟨wf functions 121⟩+≡
  int compare(const void *x, const void *y) {
```

```
        return strcmp(*(char **)x, *(char **)y);
    }
```

⟨*wf includes* 121⟩+≡
```
#include <string.h>
```

⟨*wf prototypes* 122⟩+≡
```
int compare(const void *x, const void *y);
```

wf函数对每个文件名参数都进行调用。因此,为了节省空间，它应该在返回之前释放表格以及计数。调用Table_map释放计数，调用Table_free来释放表格本身。

⟨*deallocate the entries and* table 124⟩≡
```
Table_map(table, vfree, NULL);
Table_free(&table);
```

⟨*wf functions* 121⟩+≡
```
124 void vfree(const void *key, void **count, void *cl) {
        FREE(*count);
    }
```

⟨*wf prototypes* 122⟩+≡
```
void vfree(const void *, void **, void *);
```

关键字不被释放，因为它们是原子．因此关键字一定不能被释放。除此之外，一些关键字有可能出现在下面的文件中。

把各种wf.c的程序定义片段收集起来就形成了wf程序：

⟨*wf.c*⟩≡
```
⟨wf includes 121⟩
⟨wf prototypes 122⟩
⟨wf functions 121⟩
```

8.3 实现

⟨*table.c*⟩≡
```
#include <limits.h>
#include <stddef.h>
#include "mem.h"
#include "assert.h"
#include "table.h"

#define T Table_T

⟨types 125⟩
⟨static functions 127⟩
⟨functions 126⟩
```

散列表是用来表示关联表（树是另外一种，参见练习8.2）的常见数据结构之一。因此每个Table_T都是一个指向存有bindings结构的散列表的指针，该散列表中存储的是关键字-值对：

```
⟨types 125⟩≡
  struct T {
      ⟨fields 126⟩
      struct binding {
          struct binding *link;
        const void *key;
        void *value;
    } **buckets;
};
```

125

buckets指向有适当元素数目的数组。cmp和hash函数与某个特定的表关联，因此它们都存储在buckets结构中的元素中：

```
⟨fields 126⟩≡
  int size;
  int (*cmp)(const void *x, const void *y);
  unsigned (*hash)(const void *key);
```

Table_new用它的hint参数来选择一个素数作为buckets的大小，并且它保存cmp和hash或保存指向静态函数的指针，用来比较和散列原子：

```
⟨functions 126⟩≡
  T Table_new(int hint,
      int cmp(const void *x, const void *y),
      unsigned hash(const void *key)) {
      T table;
      int i;
      static int primes[] = { 509, 509, 1021, 2053, 4093,
          8191, 16381, 32771, 65521, INT_MAX };

      assert(hint >= 0);
      for (i = 1; primes[i] < hint; i++)
          ;
      table = ALLOC(sizeof (*table) +
          primes[i-1]*sizeof (table->buckets[0]));
      table->size = primes[i-1];
      table->cmp  = cmp  ?  cmp : cmpatom;
      table->hash = hash ? hash : hashatom;
      table->buckets = (struct binding **)(table + 1);
      for (i = 0; i < table->size; i++)
          table->buckets[i] = NULL;
      table->length = 0;
      table->timestamp = 0;
    return table;
}
```

126

for循环中将i设置为primes中的第一个等于或大于hint的元素的下标，而primes[i-1]给出了buckets中元素个数。注意：循环是从1开始的。Mem接口中的ALLOC为buckets分配结构和空间。Table用素数来作为其散列表的大小是因为它无权决定关键字的散列值的计算方式。primes中的值都是最靠近2^n (9<n<16)的素数，这使得散列表的大小范围很广。Atom中使用了

一个更简单的算法，因为它同时也计算了散列值.

如果cmp或hash是空的函数指针，那么就用以下两个函数来代替：

```
⟨static functions 127⟩≡
  static int cmpatom(const void *x, const void *y) {
      return x != y;
  }

  static unsigned hashatom(const void *key) {
      return (unsigned long)key>>2;
  }
```

因为如果$x = y$，那么原子x和y是相等的，所以当$x = y$时cmpatom返回0，否则返回1。Table的这个特殊实现只测试关键字是否相等，因此cmpatom并不需要测试x和y的相对次序。一个原子就是一个地址并且这个地址本身就可以用做散列值；它总是循环右移两位，因为每个原子很有可能都以某个单词的词头边界开始、因此最后两位很有可能为0

buckets的每个元素都是一个binding结构的链接表，该结构中存有关键字、其关联的值以及指向链表中下一个binding结构的指针,图8.1给出了一个例子。每个链表中所有的关键字都有相同的散列值。

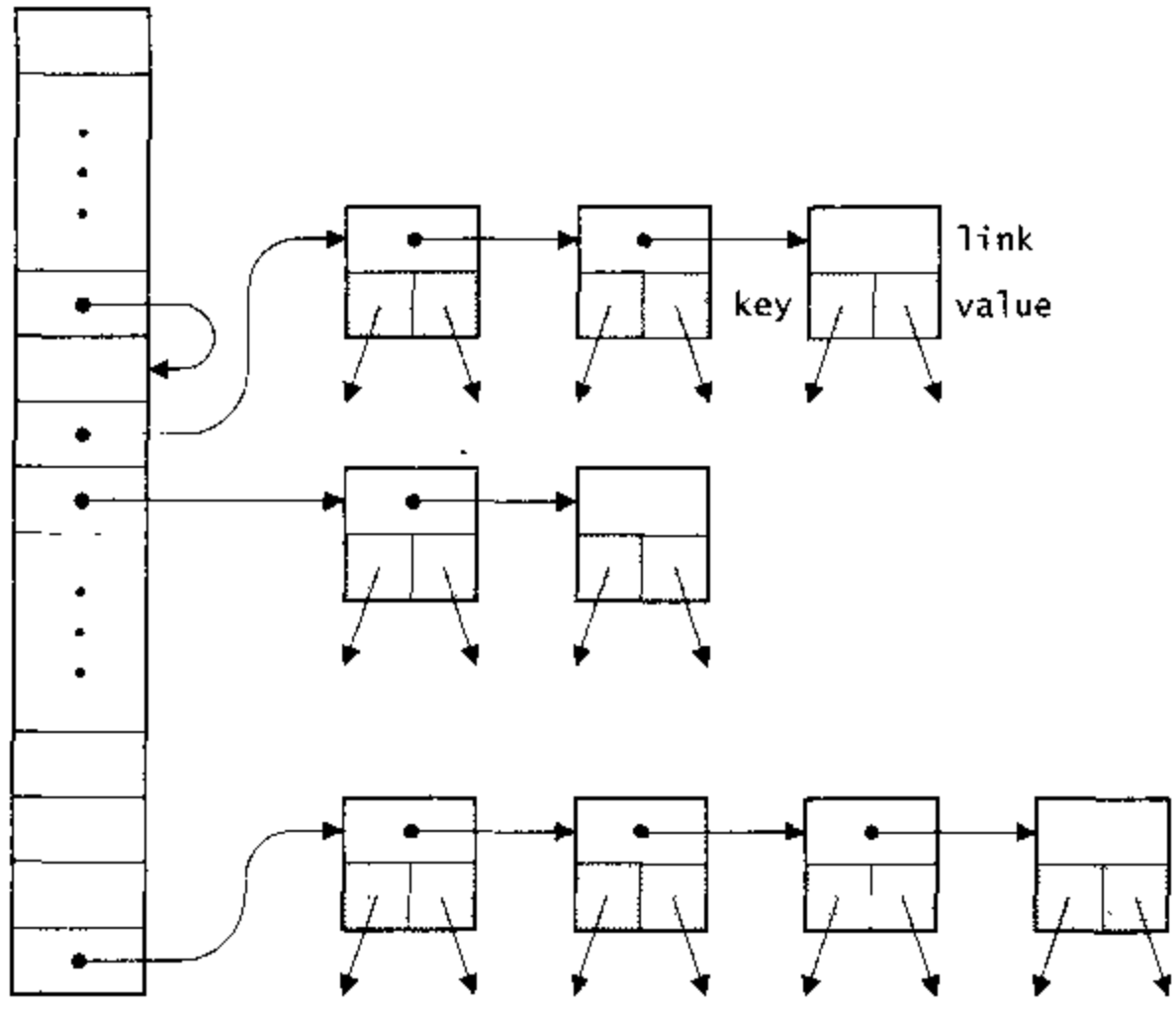


图8-1 表的布局

Table_get查找binding的过程如下：先散列它的关键字，将它对buckets的元素个数取余，并在查找列表中查找与key相等的关键字，通过调用table的hash和cmp函数来实现。

```
⟨functions 126⟩+≡
  void *Table_get(T table, const void *key) {
```

```
    int i;
    struct binding *p;
    assert(table);
    assert(key);
    ⟨search table for key 128⟩
    return p ? p->value : NULL;
}
```

```
⟨search table for key 128⟩≡
  i = (*table->hash)(key)%table->size;
  for (p = table->buckets[i]; p; p = p->link)
      if ((*table->cmp)(key, p->key) == 0)
          break;
```

当找到所需查找的关键字的时候，for循环结束，因此它使得p指向感兴趣的binding结构。否则，p为空。

127 ↓ 128

Table_put也是类似的。它查找某个关键字，如果找到，那么改变与其关联的值；如果Table_put没找到该关键字，它将分配并初始化一个新的binding结构，并把它添加到buckets上相应链表的表头。Table_put可以把新的binding链接到链表中的任何位置，但是添加到链表的表头是最容易也最有效的方法。

```
⟨functions⟩+≡
  void *Table_put(T table, const void *key, void *value) {
      int i;
      struct binding *p;
      void *prev;

      assert(table);
      assert(key);
      ⟨search table for key 128⟩
      if (p == NULL) {
          NEW(p);
          p->key = key;
          p->link = table->buckets[i];
          table->buckets[i] = p;
          table->length++;
          prev = NULL;
      } else
          prev = p->value;
      p->value = value;
      table->timestamp++;
      return prev;
  }
```

Table_put将每个表的两计数器加1：

```
⟨fields 126⟩+≡
  int length;
  unsigned timestamp;
```

length是表格中binding结构的数目；它由Table_length返回：

```
⟨functions 126⟩+≡
129   int Table_length(T table) {
        assert(table);
        return table->length;
      }
```

表格的timestamp在每次Table_put或Table_remove对表格进行改变的时候加1。Timestamp用来实现Table_map必须执行的可检查的运行期错误：当Table_map正在访问binding时，表格是不允许被改变的。Table_map将timestamp的值保存在记录项中，每次调用完apply后，它将断言表格的timestamp值是否仍等于该保存值。

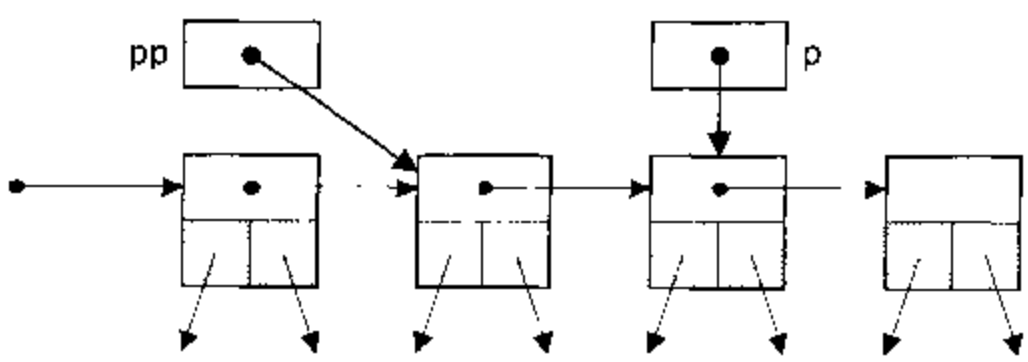
```
⟨functions 126⟩+≡
  void Table_map(T table,
       void apply(const void *key, void **value, void *cl),
       void *cl) {
       int i;
       unsigned stamp;
       struct binding *p;

       assert(table);
       assert(apply);
       stamp = table->timestamp;
       for (i = 0; i < table->size; i++)
           for (p = table->buckets[i]; p; p = p->link) {
               apply(p->key, &p->value, cl);
               assert(table->timestamp == stamp);
           }
  }
```

Table_remove同样也查找关键字，但是是通过一个指向binding结构的指针的指针来实现的，因此如果找到了这个关键字，它就可以删除该关键字的binding结构：

```
⟨functions 126⟩+≡
  void *Table_remove(T table, const void *key) {
       int i;
       struct binding **pp;

       assert(table);
       assert(key);
130    table->timestamp++;
     i = (*table->hash)(key)%table->size;
     for (pp = &table->buckets[i]; *pp; pp = &(*pp)->link)
         if ((*table->cmp)(key, (*pp)->key) == 0) {
             struct binding *p = *pp;
             void *value = p->value;
             *pp = p->link;
             FREE(p);
             table->length--;
```

```
            return value;
        }
    return NULL;
}
```

for循环在功能上等价于<*search table for key 128*>中的for循环，不同的是pp指向的是指向每个关键字binding结构的指针。pp开始指向table->buckets[i]，其后沿着链表前进，当第$k+1$个binding被检查完后，pp则指向链表中第k个binding的link字段，如下图所示。



如果*pp存有关键字，那么可以通过将*pp赋值为（*pp）->link将binding从链表中取出；而p保存*pp的值。如果Table_remove找到了需删除的关键字，那么它还将表格的长度减1。

Table_toArray与List_toArray类似。它分配一个数组，存储关键字-值对，以end指针结束，并通过访问table中的每个binding来填充数组：

```
⟨functions 126⟩+≡
  void **Table_toArray(T table, void *end) {
      int i, j = 0;
      void **array;
      struct binding *p;
    assert(table);
    array = ALLOC((2*table->length + 1)*sizeof (*array));
    for (i = 0; i < table->size; i++)
        for (p = table->buckets[i]; p; p = p->link) {
            array[j++] = (void *)p->key;
            array[j++] = p->value;
        }
    array[j] = end;
    return array;
}
```

131

p->key必须从类型const void *转换成类型void *，因为数组并不是声明为const的。数组中关键字-值对的次序是任意的。

Table_free必须释放binding结构和Table_T结构本身。释放binding结构只有在表格不为空的时候才需要：

```
⟨functions 126⟩+≡
  void Table_free(T *table) {
      assert(table && *table);
      if ((*table)->length > 0) {
          int i;
```

```
            struct binding *p, *q;
            for (i = 0; i < (*table)->size; i++)
                for (p = (*table)->buckets[i]; p; p = q) {
                    q = p->link;
                    FREE(p);
                }
        }
        FREE(*table);
    }
```

参考书目浅析

表格是很有用的一种数据结构，许多程序设计语言都把它作为内嵌的数据结构来使用。AWK（Aho, Kernighan和Weinberger 1988）是最新的例子，但是在AWK之前，表格就出现在SNOBOL4（Griswold 1972）和SNOBOL4的后继Icon(Griswold和Griswold 1990)中。在SNOBOL4和Icon中的表格可以保存任何类型的值，并通过该值来进行索引；但是AWK的表
132 格（也称做数组）只能存储字符串和数字类型的数据，并通过它们来进行索引。Table接口的实现使用了与Icon(Griswold和Griswold 1986)中表格的实现相同的技术。

PostScript(Adobe Systems 1990)，一种页描述语言，同样也有表格，但是它把表格称为字典。PostScript的表格可以用“名字”来索引，这个“名字”是PostScript对原子的另一种叫法，但是可以存储任何类型的值，包括字典。

表格同样也出现在面向对象的程序设计语言当中，要么作为内嵌的类型，要么在程序库中。SmallTalk和Objective-C中的基本程序库就包括了字典，非常类似Table接口导出的表格。这类对象通常被称做容器（container）对象，因为它们存有其他对象的集合。

Table接口的实现使用了固定大小的散列表。只要装填因子（表中填入的记录数除以散列表的长度）适当的小，那么关键字只需查找很小的一部分记录就可以找到。装填因子越大，其性能就越差。当装填因子太大的时候，可以通过扩充散列表而把装填因子控制在合理的范围内，比如说5，练习8.5就研究了动态散列表的一种有效的，但是很简单的实现，它将散列表扩充并重新计算所有已存储记录的散列值。Larson(1988)很详细地描述了一种更复杂的方法，它的散列表是逐渐扩充（或缩小）的，一次一个散列链。Larson的方法中不需要hint，并且它可以节省空间，因为所有表开始的时候都可以很小。

练习

8.1 对关联表ADT有许多种可行的实现方法。例如，在早期版本的Table接口中，Table_get返回指向值的指针而不是返回值本身，因此客户调用程序可以改变这些值。在设计中，即使表格中关键字已经存在，Table_put也总是往表格中添加一个新的binding，并有效地把先前具有相同值的binding“隐藏”起来，而Table_remove也只删除最近的binding。但是Table_map可以访问表格中所有的

binding。讨论这些以及其他实现方法的优劣。设计并实现一个不同的表格ADT。

8.2 Table接口的设计，使得可以使用其他数据结构来实现表格。例如，比较函数得到了两个关键字的相对顺序，使得它可以用树来实现。使用二分查找树或红-黑树来重新实现Table接口。关于这两种数据结构的细节请参看Sedgewick (1990)。 133

8.3 Table_map和Table_toArray函数访问表格中binding的顺序是不确定的。假设将接口修改使得Table_map可以以binding插入到表格的顺序来访问它们，并且Table_Array以相同的顺序返回数组。实现这种修改。这样修改的实际好处是什么？

8.4 假设接口规定Table_map和Table_array以排好的顺序来访问binding。这个规定会使得Table的实现变得复杂，但是可以简化诸如wf的客户调用程序对表格中binding的排序。讨论该提议的优点并实现它。提示：在当前的实现中，Table_put平均情况下的运行时间是常数，而Table_get平均情况下的运行时间也接近常数。那么在你修改后的实现中，Table_put和Table_get在平均情况下的运行时间又如何呢？

8.5 一旦buckets已经分配了，那么它就不能再扩充或缩小。修改Table接口的实现使得它可以使用一种探试方法，当关键字-值对被添加或删除时周期性地调整buckets的大小。设计一个测试程序，测试你这个探试方法的有效性，并度量它的好处。

8.6 实现Larson(1988)描述的线形动态散列算法，并把它的性能与你前面练习的答案进行比较。

8.7 修改wf.c，使得它可以度量因为原子从不释放而浪费的空间。

8.8 修改wf.c的比较函数，使得它按计数值的降序来对数组进行排序。

8.9 修改wf.c，使得它对每个文件参数按文件名的字母顺序打印出结果。做了这种修改后，在8.2节最开始给的例子中，mem.c中的计数就会出现在table.c的前面。 134

第9章 集 合

一个集合（set）是不同成员的无序聚集。对集合的基本操作包括对集合成员资格进行检查、增加成员和删除成员等，同时还包括集合的并（union）、交（intersection）、差（difference），以及对称差分（symmetric difference）等其他操作。假如有两个集合s和t，则并操作$s+t$就是包含s中所有元素和t中所有元素的集合；而交操作$s*t$是s和t中都包含的元素的集合；差操作$s-t$则表示在s中出现但不在t中出现的集合；至于对称差分操作，它常常被写成s/t的形式，是只在s中或只在t中出现的元素的集合。

集合通常以域（universe）的概念来描述——域即是所有可能的成员的集合。例如，字符集的集合通常与由256个8比特字符代码组成的域相联系。当域U被定义后，就能存在s的补集，即$U-s$。

集合由Set接口提供而不依赖于任何域。该接口输出操作集合成员的函数，但是从来不直接检测它们。同表（Table）接口一样，Set接口被设计成客户调用程序，能够提供对某个特定的集合里的成员的属性进行检测的函数功能。

应用程序使用集合同使用表格的方式非常相似。事实上，由Set提供的集合很像表（table）：集合成员是关键字，而同这些关键字相关联的值则被忽略。

9.1 接口

```
⟨set.h⟩≡
  #ifndef SET_INCLUDED
  #define SET_INCLUDED

  #define T Set_T
  typedef struct T *T;

  ⟨exported functions 138⟩

  #undef T
  #endif
```

由Set接口导出的功能被分成四组：分配和释放、基本的集合操作、集合遍历、接受集合的操作数和返回诸如集合的并集的新的集合的操作。前三个功能同表接口中的功能很相似。Set_T由下面的代码分配和释放。

```
⟨exported functions 138⟩≡
  extern T    Set_new (int hint,
      int cmp(const void *x, const void *y),
      unsigned hash(const void *x));
  extern void Set_free(T *set);
```

Set_new分配、初始化并返回一个新的T。hint是对集合所应该包含的成员数目的一个估计；精确的hint值虽然能够改善性能，但是任何非负的值也是合理的。cmp和hash用于比较两个成员并映射到无符号整型数上。假定有两个成员x和y，cmp（x,y）肯定返回一个大于、小于或者等于零的整数，分别对应于x大于y、x小于y和x等于y三种情况。如果cmp（x,y）是零，则x和y只有其中之一会显示在集合中，hash（x）也一定等于hash（y）。Set_new可以引发异常Mem_Failed。

如果cmp是空函数指针，则成员被认为是原子；如果x=y，则两个成员x和y被认为是一致的。否则，如果hash是空函数指针，那么Set_new将提供一个hash函数以适应于原子（atom）。

138

Set_free释放 *set，并给它分配一个空指针。Set_free不释放成员；Set_map可以释放成员。把一个空的set或者 *set传递给Set_free会产生可检查的运行期错误。

基本操作如下面函数所示：

```
⟨exported functions 138⟩+≡
  extern int   Set_length(T set);
  extern int   Set_member(T set, const void *member);
  extern void  Set_put   (T set, const void *member);
  extern void *Set_remove(T set, const void *member);
```

Set_length返回集合set的底数（cardinality），或者返回它所包含的成员数。Set_member返回1或0，如果成员在集合里面则返回1，如果不在，则返回0。Set_put将向集合添加一个成员，除非该成员已经存在；Set_put可以引发Mem_failed。如果集合包含某成员，则Set_remove可以从集合里面删除该成员，并且返回被删除的成员（其指针可能不同于原先指向该成员的指针）。否则，Set_remove不进行任何操作，且返回值为0。试图将一个空的集合或者空成员传递给上述这些例程中的任何一个都将会产生可检查的运行期错误。

下面的函数能够访问一个集合中的所有成员：

```
⟨exported functions 138⟩+≡
  extern void   Set_map    (T set,
      void apply(const void *member, void *cl), void *cl);
  extern void **Set_toArray(T set, void *end);
```

Set_map对集合set的每个成员都调用一遍apply函数。它把成员和由客户调用程序定义的指针cl传递给apply。但是它并不检查cl。值得注意的是，与Table_map不同，apply不能改变存储在集合中的成员。把一个空apply函数或者空集传递给Set_map和apply，通过调用Set_put或Set_remove来改变set都将会产生可检查的运行期错误。

Set_toArray返回一个$N+1$个元素的数组，该数组以任意的顺序存放着N个集合元素。end通常是一个空指针，它的值就是数组中第$N+1$个元素的值。Set_toArray可以引发异常Mem_Failed。客户调用程序必须释放返回的数组。把一个空的set传递给Set_toArray会产生可检查的运行期错误。

139

下列函数：

```
⟨exported functions 138⟩+≡
  extern T Set_union(T s, T t);
  extern T Set_inter(T s, T t);
  extern T Set_minus(T s, T t);
  extern T Set_diff (T s, T t);
```

执行的是在本章开始部分描述的四种集合操作。Set_union 返回s+t，Set_inter 返回s*t，Set_minus返回s-t，Set_diff返回s/t。这四种函数都能创建和返回新的T，也都可能引发异常Mem_Failed。它们把s或者t当作空的集合，但总返回一个新的、非空的T。因此，Set_union（s，NULL）将返回s的一个备份。如果s和t都为空，或者当s和t都为非空却有不同的比较和散列函数时，每个函数都会产生可检查的运行期错误。也就是说，s和t必须通过调用定义了相同的比较和散列函数后的Set_new函数来创建。

9.2　实例：交叉引用列表

xref打印在其输入文件中的标识符的交叉引用列表（cross-reference list），这有助于多方面的使用，如寻找在一个程序源文件中的所有特定标识符。例如下例：

```
% xref xref.c getword.c
...
FILE    getword.c: 6
        xref.c: 18 43 72
...
c       getword.c: 7 8 9 10 11 16 19 22 27 34 35
        xref.c: 141 142 144 147 148
...
```

上面的例子说明FILE在getword.c文件的第6行和xref.c文件的第18、43、72行中被用到。类似地，c出现在getword.c中11个不同的地方。其中每行只显示一次，即使是同一行出现了多个相同的标识符也是这样。输出时则按顺序列出了文件和行号。

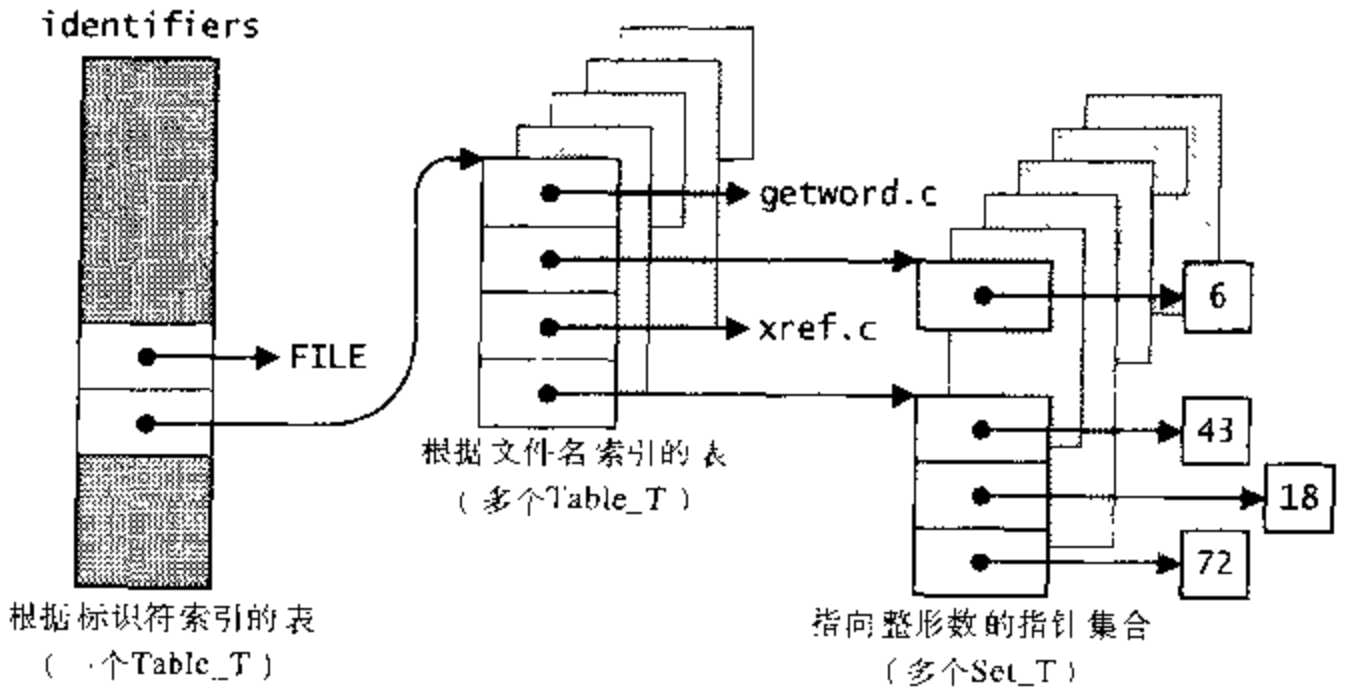
如果没有程序参数，xref会将一个标识符的交叉引用列表发送到一个标准输入上，但将会忽略上面输出实例中的文件名：

140

```
% cat xref.c getword.c | xref
...
FILE 18 43 72 157
...
c 141 142 144 147 148 158 159 160 161 162 167 170 173 178
185 186 ...
```

xref的实现将会显示集合和表在一起是如何使用的。它建立了一个由标识符索引的表，表中每个相关联的值都是另一个由文件名索引的表。该表中的值是一整型指针集合，指向文件的行号。图9-1描述了这种结构并显示了标识符FILE的细节，如上面所描述的那样。在单个顶级表（top-level table）（它是下面代码中identifiers的值）中与FILE相关联的值是第二级Table_T，它带有两个关键字：getword.c 的原子和xref.c的原子。与这些关键字相关联的是

Set_T，它们含有FILE所出现的行号的指针。顶级表中的每个标识符都有第二级表，第二级表中的每个关键字-值对都有一个集合。



141

图9-1 交叉引用列表的数据结构

⟨xref.c⟩≡
⟨xref includes 142⟩
⟨xref prototypes 143⟩
⟨xref data 146⟩
⟨xref functions 142⟩

xref的main函数很像wf的main函数：它创建标识符的表，然后处理它的文件名参数。它能打开每个文件，并用文件指针、文件名和标识符表来调用函数xref。如果没有参数，将用一个空指针、标准输入的文件指针和标识符表来调用xref：

⟨xref functions 142⟩≡
```
int main(int argc, char *argv[]) {
    int i;
    Table_T identifiers = Table_new(0, NULL, NULL);

    for (i = 1; i < argc; i++) {
        FILE *fp = fopen(argv[i], "r");
        if (fp == NULL) {
            fprintf(stderr, "%s: can't open '%s' (%s)\n",
                argv[0], argv[i], strerror(errno));
            return EXIT_FAILURE;
        } else {
            xref(argv[i], fp, identifiers);
            fclose(fp);
        }
    }
    if (argc == 1) xref(NULL, stdin, identifiers);
    ⟨print the identifiers 143⟩
    return EXIT_SUCCESS;
}
```

```
⟨xref includes 142⟩≡
  #include <stdio.h>
  #include <stdlib.h>
  #include <errno.h>
  #include "table.h"
```

xref建立了一个复杂的数据结构，但当通过搜索数据结构中的组件来第一次检查怎样打印它的内容时，会更容易理解它是怎样建立的。为每个组件分块或者分别写函数将有助于理解过程的细节。 142

第一步，建立一个标识符数组，赋值并按升序排列，然后通过另一个函数print来逐步输出数组，并处理其值。这一步很像wf的代码块*<print the words 123>*。

```
⟨print the identifiers 143⟩≡
  {
      int i;
      void **array = Table_toArray(identifiers, NULL);
      qsort(array, Table_length(identifiers),
          2*sizeof (*array), compare);
      for (i = 0; array[i]; i += 2) {
          printf("%s", (char *)array[i]);
          print(array[i+1]);
      }
      FREE(array);
  }
```

identifiers的关键字是原子（atom），所以，传递给标准库函数qsort的比较函数compare将与在wf中使用的compare一致，并使用strcmp比较标识符对（8.2节中对qsort的定义解释了qsort的参数）：

```
⟨xref functions 142⟩+≡
  int compare(const void *x, const void *y) {
      return strcmp(*(char **)x, *(char **)y);
  }

⟨xref includes 142⟩+≡
  #include <string.h>

⟨xref prototypes 142⟩≡
  int compare(const void *x, const void *y);
```

identifiers中的每一个值都是另外一个表，该值被传递给print函数。对文件名来说这个表的关键字是原子，因此，它们可以通过与上述代码相似的代码实现在数组中捕获、排序、来回移动等。 143

```
⟨xref functions 142⟩+≡
  void print(Table_T files) {
      int i;
      void **array = Table_toArray(files, NULL);
```

```
    qsort(array, Table_length(files), 2*sizeof (*array),
        compare);
    for (i = 0; array[i]; i += 2) {
        if (*(char *)array[i] != '\0')
            printf("\t%s:", (char *)array[i]);
        ⟨print the line numbers in the set array[i+1] 144⟩
        printf("\n");
    }
    FREE(array);
}
```

```
⟨xref prototypes 143⟩+≡
  void print(Table_T);
```

print可以使用compare，因为关键字都是字符串。如果没有文件名参数，则传递给print的每一个表将只有一个入口，并且关键字是一个零长度的原子。print用这种约定来避免在输出行号列表之前打印文件名。

表中每个值都被传递给print，它们是一些行号的集合。因为Set实现了指针集合，xref通过整型指针来表示行号并把该指针加入到集合中。为了打印它们，将调用Set_toArray函数来建立和返回带有整型指针的以空字符作为结束符的数组；然后给数组排序并打印这些整数：

```
⟨print the line numbers in the set array[i+1] 144⟩≡
  {
      int j;
      void **lines = Set_toArray(array[i+1], NULL);
      qsort(lines, Set_length(array[i+1]), sizeof (*lines),
          cmpint);
      for (j = 0; lines[j]; j++)
          printf(" %d", *(int *)lines[j]);
      FREE(lines);
  }
```

144

cmpint同compare相似，但是它含有两个指向整型指针的指针，并比较这两个整数：

```
⟨xref functions 142⟩+≡
  int cmpint(const void *x, const void *y) {
      if (**(int **)x < **(int **)y)
          return -1;
      else if (**(int **)x > **(int **)y)
          return +1;
      else
          return 0;
  }
```

```
⟨xref prototypes 143⟩+≡
  int cmpint(const void *x, const void *y);
```

xref建立了两个数据结构，该数据结构由刚刚讨论的代码输出。它用getword读取输入的标识符。对每一个标识符，它使用适当的数据结构形成集合并把当前的行号加入到集合中：

```
⟨xref functions 142⟩+≡
  void xref(const char *name, FILE *fp,
            Table_T identifiers){
      char buf[128];

      if (name == NULL)
          name = "";
      name = Atom_string(name);
      linenum = 1;
      while (getword(fp, buf, sizeof buf, first, rest)) {
          Set_T set;
          Table_T files;
          const char *id = Atom_string(buf);
          ⟨files ← file table in identifiers associated with id 147⟩
          ⟨set ← set in files associated with name 147⟩
          ⟨add linenum to set, if necessary 148⟩
      }
  }
```

145

```
⟨xref includes 142⟩+≡
  #include "atom.h"
  #include "set.h"
  #include "mem.h"
  #include "getword.h"

⟨xref prototypes 143⟩+≡
  void xref(const char *, FILE *, Table_T);
```

lincnum是一个全局变量，无论何时first扫描完一行字符，linenum都将增加；first是一个函数，它被传递给getword以识别在标识符中的初始字符。

```
⟨xref data 146⟩≡
  int linenum;

⟨xref functions 142⟩+≡
  int first(int c) {
      if (c == '\n')
          linenum++;
      return isalpha(c) || c == '_';
  }

  int rest(int c) {
      return isalpha(c) || c == '_' || isdigit(c);
  }

⟨xref includes 142⟩+≡
  #include <ctype.h>
```

getword和传递给它的first和rest函数在8.2节进行了描述。

```
⟨xref prototypes 143⟩+≡
  int first(int c);
  int rest (int c);
```

通过导航指向适当集合的代码必须能处理丢失的组件。例如，一个标识符在第一次遇到
146 时，标识符在identifiers中将没有入口，因此，代码将创建文件表，并把标识符-文件表对添加到空闲的identifiers中：

```
⟨files ← file table in identifiers associated with id 147⟩≡
  files = Table_get(identifiers, id);
  if (files == NULL) {
      files = Table_new(0, NULL, NULL);
      Table_put(identifiers, id, files);
  }
```

同样地，当一个标识符在一个新文件中第一次出现时，并没有行号的集合，因此一个新的集合将会在第一次需要它的时候将其创建和添加到文件表中：

```
⟨set ← set in files associated with name 147⟩≡
  set = Table_get(files, name);
  if (set == NULL) {
      set = Set_new(0, intcmp, inthash);
      Table_put(files, name, set);
  }
```

集合是整型指针的集合；intcmp和inthash比较并对这些整数进行散列。intcmp同上面的cmpint类似，但是它的参数是集合中的指针，因此它可以调用cmpint。整数本身也可以用作它自己的散列数（hash number）：

```
⟨xref functions 142⟩+≡
  int intcmp(const void *x, const void *y) {
      return cmpint(&x, &y);
  }

  unsigned inthash(const void *x) {
      return *(int *)x;
  }
```

```
⟨xref prototypes 143⟩+≡
  int      intcmp (const void *x, const void *y);
147 unsigned inthash(const void *x);
```

在控制到达<*add linenum to set,if necessary 148*>之前，set是当前行号应该被插入到的集合。由下述代码来完成：

```
int *p;
NEW(p);
*p = linenum;
Set_put(set, p);
```

但是如果set已经包含了linenum，上述代码将产生一个存储漏洞（memory leak），因为最新分配的空间的指针不会被加入到表中。这个漏洞可以通过只有linenum不在set中时才分配空间的措施来避免：

```
⟨add linenum to set, if necessary 148⟩≡
  {
      int *p = &linenum;
      if (!Set_member(set, p)) {
          NEW(p);
          *p = linenum;
          Set_put(set, p);
      }
  }
```

9.3 实现

Set的实现同Table的实现很相似。它用散列表表示集合，并且用比较函数和散列函数来寻找这些表中的成员。下面的例子探索了一些可行的集合实现和Table实现。

```
⟨set.c⟩≡
  #include <limits.h>
  #include <stddef.h>
  #include "mem.h"
  #include "assert.h"
  #include "arith.h"
  #include "set.h"
  #define T Set_T
  ⟨types 149⟩
  ⟨static functions 150⟩
  ⟨functions 149⟩
```

148

Set_T是一个散列表，里面包含成员：

```
⟨types 149⟩≡
  struct T {
      int length;
      unsigned timestamp;
      int (*cmp)(const void *x, const void *y);
      unsigned (*hash)(const void *x);
      int size;
      struct member {
          struct member *link;
          const void *member;
      } **buckets;
  };
```

length是集合中的成员数；timestamp用于检测Set_map中产生的可检查的运行时错误，Set_map禁止apply函数改变集合，cmp和hash分别含有比较函数和散列函数。

同Table_new一样，Set_new为buckets数组计算正确的元素数目，在size域里存储元素数目，并为结构T和buckets数组分配空间：

```
⟨functions 149⟩≡
  T Set_new(int hint,
      int cmp(const void *x, const void *y),
```

```
        unsigned hash(const void *x)) {
        T set;
        int i;
        static int primes[] = { 509, 509, 1021, 2053, 4093,
            8191, 16381, 32771, 65521, INT_MAX };

        assert(hint >= 0);
        for (i = 1; primes[i] < hint; i++)
            ;
        set = ALLOC(sizeof (*set) +
            primes[i-1]*sizeof (set->buckets[0]));
    set->size = primes[i-1];
    set->cmp  = cmp  ?  cmp : cmpatom;
    set->hash = hash ? hash : hashatom;
    set->buckets = (struct member **)(set + 1);
    for (i = 0; i < set->size; i++)
        set->buckets[i] = NULL;
    set->length = 0;
    set->timestamp = 0;
    return set;
}
```

149

Set_new用hint为buckets中的元素数目（见8.3相关定义）在prime中选择一个值。如果成员是由cmp或hash的空函数指针来说明的原子，则Set_new使用下面的比较和散列函数，在Table_new中也使用相同的函数。

```
⟨static functions 150⟩≡
  static int cmpatom(const void *x, const void *y) {
      return x != y;
  }

  static unsigned hashatom(const void *x) {
      return (unsigned long)x>>2;
  }
```

9.3.1 成员操作

对成员进行测试就像在一个表中查询一个关键字：散列潜在的成员，并搜寻从buckets中发出的适当的列表：

```
⟨functions 149⟩+≡
  int Set_member(T set, const void *member) {
      int i;
      struct member *p;

      assert(set);
      assert(member);
      ⟨search set for member 151⟩
      return p != NULL;
  }
```

150

```
⟨search set for member 151⟩≡
  i = (*set->hash)(member)%set->size;
  for (p = set->buckets[i]; p; p = p->link)
      if ((*set->cmp)(member, p->member) == 0)
          break;
```

如果搜索成功则p非空，否则p为空。因此，对p的测试决定着Set_member的输出。

添加一个新的成员与此类似：为成员搜寻集合，如果搜索失败则把它加进去。

```
⟨functions 149⟩+≡
  void Set_put(T set, const void *member) {
      int i;
      struct member *p;

      assert(set);
      assert(member);
      ⟨search set for member 151⟩
      if (p == NULL) {
          ⟨add member to set 151⟩
      } else
          p->member = member;
      set->timestamp++;
  }

⟨add member to set 151⟩≡
  NEW(p);
  p->member = member;
  p->link = set->buckets[i];
  set->buckets[i] = p;
  set->length++;
```

timestamp用在Set_map中来执行可检查的运行期错误。

Set_remove通过将一个指向成员结构的指针的指针pp移动到适当的散列链的方法来删除成员，直到*pp为空或（*pp）->member是相关的成员为止，在这种情况下，下面的分配*pp = (*pp)->link将从链中删除该结构。

151

```
⟨functions 149⟩+≡
  void *Set_remove(T set, const void *member) {
      int i;
      struct member **pp;

      assert(set);
      assert(member);
      set->timestamp++;
      i = (*set->hash)(member)%set->size;
      for (pp = &set->buckets[i]; *pp; pp = &(*pp)->link)
          if ((*set->cmp)(member, (*pp)->member) == 0) {
              struct member *p = *pp;
              *pp = p->link;
              member = p->member;
              FREE(p);
```

```
                set->length--;
                return (void *)member;
            }
        return NULL;
    }
```

将pp移到散列链与Table_remove中的方式相同。请参见8.3节实现部分的相应定义。Set_remove和Set_put通过增加或减少length域来跟踪集合中的成员数，其中length是由Set_length返回的。

```
⟨functions 149⟩+≡
  int Set_length(T set) {
      assert(set);
      return set->length;
  }
```

如果集合非空，Set_free首先必须在用散列链中释放集合本身，并在清除*Set之前释放成员结构。

```
⟨functions 149⟩+≡
  void Set_free(T *set) {
      assert(set && *set);
      if ((*set)->length > 0) {
        int i;
        struct member *p, *q;
        for (i = 0; i < (*set)->size; i++)
            for (p = (*set)->buckets[i]; p; p = q) {
                q = p->link;
                FREE(p);
            }
      }
      FREE(*set);
  }
```

Set_map几乎与Table_map结构一致：它通过调用apply函数来为每个成员移动散列链。

```
⟨functions 149⟩+≡
  void Set_map(T set,
      void apply(const void *member, void *cl), void *cl) {
      int i;
      unsigned stamp;
      struct member *p;

      assert(set);
      assert(apply);
      stamp = set->timestamp;
      for (i = 0; i < set->size; i++)
          for (p = set->buckets[i]; p; p = p->link) {
              apply(p->member, cl);
              assert(set->timestamp == stamp);
          }
  }
```

不同的一点是Set_map传递给apply的是每个成员，而不是每个成员的指针，因此，apply不能改变集合中的指针。然而，它可以用强制类型转换（cast）来改变这些成员所指向的值，但这修改了集合的语义。

Set_toArray比Table_toArray更简单；同List_toArray类似，Set_toArray分配一个数组并仅仅把成员拷贝进去。

```
⟨functions 149⟩+≡
  void **Set_toArray(T set, void *end) {
      int i, j = 0;
    void **array;
    struct member *p;

    assert(set);
    array = ALLOC((set->length + 1)*sizeof (*array));
    for (i = 0; i < set->size; i++)
        for (p = set->buckets[i]; p; p = p->link)
            array[j++] = (void *)p->member;
    array[j] = end;
    return array;
}
```

153

p->member必须从const void* 变为void*，因为数组没有被声明为常量。

9.3.2 集合操作

所有四种集合操作都有相似的实现。例如，s+t是通过把s和t的每个元素加入到一个新的集合中来实现的，其做法可以是先复制s，然后如果t有成员不在s的副本中，再把t的该成员加入到该副本里面即可：

```
⟨functions 149⟩+≡
  T Set_union(T s, T t) {
      if (s == NULL) {
          assert(t);
          return copy(t, t->size);
      } else if (t == NULL)
          return copy(s, s->size);
      else {
          T set = copy(s, Arith_max(s->size, t->size));
          assert(s->cmp == t->cmp && s->hash == t->hash);
          { ⟨for each member q in t 154⟩
              Set_put(set, q->member);
          }
          return set;
      }
  }

⟨for each member q in t 154⟩≡
  int i;
```

154

```
struct member *q;
for (i = 0; i < t->size; i++)
    for (q = t->buckets[i]; q; q = q->link)
```

内部函数copy返回它的参数的一个副本，其中该参数必须是非空的。

```
⟨static functions 150⟩+≡
  static T copy(T t, int hint) {
      T set;

      assert(t);
      set = Set_new(hint, t->cmp, t->hash);
      { ⟨for each member q in t 154⟩
          ⟨add q->member to set 155⟩
      }
      return set;
  }
```

```
⟨add q->member to set 155⟩≡
  {
      struct member *p;
      const void *member = q->member;
      int i = (*set->hash)(member)%set->size;
      ⟨add member to set 151⟩
  }
```

Set_union和copy都有权访问特权信息：它们知道集合的表示方法，因此可以通过把适当的hint传递给Set_new的方法来为一个新的集合定义散列表的大小。在复制s时，Set_union提供了一个hint；它使用s和t中较大的那个散列表的大小，因为作为结果的集合将至少有与Set_union的最大参数相等的成员数。copy可以调用Set_put来把每个成员添加到副本中去，但是它通常使用<*add q->member to set 155*>来直接添加成员，从而避免了Set_put的没有结果的搜索。

交s*t用s和t中较小的散列表创建了一个新的集合，并只把在s和t中都出现的元素加到这个新的集合中：

155

```
⟨functions 149⟩+≡
  T Set_inter(T s, T t) {
      if (s == NULL) {
          assert(t);
          return Set_new(t->size, t->cmp, t->hash);
      } else if (t == NULL)
          return Set_new(s->size, s->cmp, s->hash);
      else if (s->length < t->length)
          return Set_inter(t, s);
      else {
          T set = Set_new(Arith_min(s->size, t->size),
              s->cmp, s->hash);
          assert(s->cmp == t->cmp && s->hash == t->hash);
          { ⟨for each member q in t 154⟩
```

```
            if (Set_member(s, q->member))
                ⟨add q->member to set 155⟩
        }
        return set;
    }
}
```

如果s比t成员数少，则Set_inter调用自己并且交换s和t。这将导致代码块最后是在较小的一个集合里进行交集运算循环。

差s-t创建一个新的集合并把在s中但不在t中的成员添加到里面。下述代码将交换参数名，从而能够使用代码块<*for each member q in t 154*>来按顺序排列s：

```
⟨functions 149⟩+≡
  T Set_minus(T t, T s) {
      if (t == NULL){
          assert(s);
          return Set_new(s->size, s->cmp, s->hash);
      } else if (s == NULL)
          return copy(t, t->size);
      else {
          T set = Set_new(Arith_min(s->size, t->size),
              s->cmp, s->hash);
          assert(s->cmp == t->cmp && s->hash == t->hash);
          { ⟨for each member q in t 154⟩
            if (!Set_member(s, q->member))
                ⟨add q->member to set 155⟩
          }
          return set;
      }
  }
```

对称差分s / t，是只在s或只在t中含有的元素的集合。如果s或t为空，则s / t就是s或t。否则，s / t等于（s-t）+（t-s），可以通过下述方法来完成：先忽略s，把t中没有的所有元素加到新的集合中，然后再忽略t，将s中没有的所有元素加到新的集合中。代码块<*for each member q in t 154*>用于在互相传递时交换s和t的值：

```
⟨functions 149⟩+≡
  T Set_diff(T s, T t) {
      if (s == NULL) {
          assert(t);
          return copy(t, t->size);
      } else if (t == NULL)
          return copy(s, s->size);
      else {
          T set = Set_new(Arith_min(s->size, t->size),
              s->cmp, s->hash);
          assert(s->cmp == t->cmp && s->hash == t->hash);
          { ⟨for each member q in t 154⟩
              if (!Set_member(s, q->member))
                  ⟨add q->member to set 155⟩
```

```
        }
        { T u = t; t = s; s = u; }
        { ⟨for each member q in t 154⟩
            if (!Set_member(s, q->member))
                ⟨add q->member to set 155⟩
        }
        return set;
    }
}
```

有关这四个操作的更有效的实现也是可能的；其中一些会在练习中探讨。一个特殊的情
157 况是s和t中的散列表大小相等，这可能对某些应用很有用，请参见练习9.7。

参考书目浅析

由Set输出的集合被应用到Icon中的集合上（Griswold and Griswold 1990），实现也同Icon的相似（Griswold and Griswold 1986）。位向量通常用于代表固定的、小的域集合；第13章将描述使用该方法的接口。

Icon是把集合作为内嵌数据类型的为数不多的语言之一。集合在SETL中是主要的数据类型，并且大部分操作和控制结构都被设计成能处理集合。

练习

9.1 用Table实现Set。

9.2 用Set实现Table。

9.3 Set和Table的实现有很多共同点。设计并实现一个第三类接口，使之能体现它们的共同属性。该接口的目的是使ADT具有像Set和Table那样的执行能力。用新的接口再次实现Set和Table。

9.4 为包（bag）设计一个接口。包类似于一个集合，但是其成员可以显示一次或多次；例如，{1 2 3}是一组整数，而{1 1 2 2 3}是一整数包。用先前练习中设计的支持接口实现该接口。

9.5 copy复制了集合的参数，每次复制一个成员。既然知道备份中的成员的个数，就能够一次分配所有的member结构，然后一小部分一小部分发送到适当的散列链中，直到完成整个备份。执行这个方案并度量它的优点。

9.6 一些集合操作可能通过下述方法变得更有效，把散列值存放在member结构中，以
158 便只为每个成员调用一次hash函数，并只有在散列数值相等的时候才调用比较函数。分析此法所带来的好处，执行程序并分析结果。

9.7 如果s和t有相同的buckets数，$s+t$ 等于某个子集的集合，其中该子集的成员都在同一个散列链中。也就是说，$s+t$ 中每个散列链都是在s和t的相应的散列链中的元素的集合。这种情况会经常发生，因为有很多应用，无论它们何时调用Set_new，都

定义相同的hint。改变$s + t$、$s * t$、$s - t$和s/t的实现以检查上述情况，并使用适当的、更简单有效的方法来实现。

9.8 如果连续几行都出现了同一标识符，xref将会输出每个行号。例如：

```
c       getword.c: 7 8 9 10 11 16 19 22 27 34 35
```

修改xref.c，用行的范围来代替连续的两行或者更多行：

```
c       getword.c: 7-11 16 19 22 27 34-35
```

9.9 xref分配了许多内存，但是只释放了由Table_toArray创建的内存数组。修改xref，使之能够最终释放出它分配的所有内存（当然要除去原子）。采取在打印数据结构时递增的方法实现上述情况是最容易的。使用解决练习5.5的方法来检查是否释放了所有的内存。

9.10 请解释为什么cmpint 和intcmp采用直接比较的方法来比较整数而不是返回它们相减的结果。也就是说，下述代码哪里有问题（显然非常简单）？是否是cmpint的版本问题？

```
int cmpint(const void *x, const void *y) {
    return **(int **)x - **(int **)y;
}
```

第10章 动态数组

数组（array）是有关值的同构序列，序列里面的元素同相邻连续范围的索引是一一对应的。在实际的编程语言中，某些形式的数组是作为内嵌数据类型来显示的。在一些语言中，如C，所有的数组索引具有相同的下限（lower bound），而在其他语言中，如Modula-3，每个数组都有自己的范围。在C中，所有数组的索引都是从零开始的。

数组的大小要么在编译时定义，要么在运行时定义。静态数组的大小当然是在编译时定义。例如，在C中，已声明的数组在编译时必须要具有长度；即在声明数组int [n]时，n必须固定；静态数组也可以在运行时定义，例如局部数组就是在运行时通过调用显示它们的函数被分配的，但是数组的大小必须在编译时确定。

由Table_toArray等函数返回的数组就是动态数组，因为分配给它们的空间是通过调用malloc或是等价的分配函数来实现的。因此，它们的长度可以在运行时确定。其中一些语言，如Modula-3，能很好地支持动态数组。可是在C中，它们必须由诸如Table_toArray这样的函数来清楚地构建。

各种toArray函数显示了动态数组的用途；本章中所描述的Array ADT提供了一个相似但更一般的工具。它导出分配和释放动态数组的函数，用边界检查方法来访问它们并扩展或缩减它们以使能容纳更多或者更少的元素。

本章还描述了ArrayRep接口。它为少数需要更有效地访问数组元素的客户调用程序揭示了动态数组的表示方法。同时，Array和ArrayRep说明了两级接口（two-level interface）或者分层接口（layered interface）。Array定义了一个数组ADT的高层视图（high-level view），ArrayRep则在较低的层面上定义了ADT的另外一个更详细的视图。这么组织的优点是通过导入ArrayRep能清楚地识别那些依赖动态数组表示方法的客户调用程序。表示方法的改变只会影响这些客户调用程序，而不会影响其他大多数只导入Array的客户调用程序。

10.1 接口

下列Array ADT

```
⟨array.h⟩≡
  #ifndef ARRAY_INCLUDED
  #define ARRAY_INCLUDED

  #define T Array_T
  typedef struct T *T;

  ⟨exported functions 162⟩
```

```
#undef T
#endif
```

导出了对一个N元素数组进行操作的函数，其中的数组通过索引0到N-1来访问。在一个给定数组中，每个元素都有固定的大小，但是不同的数组，允许元素具有不同的大小。Array_T由下述代码来分配和释放：

```
⟨exported functions 162⟩≡
  extern T    Array_new (int length, int size);
  extern void Array_free(T *array);
```

Array_new分配、初始化并返回一个具有length个元素的新数组，范围是从0到length - 1，除非length是0，此时，数组中没有元素。每个元素都占用size字节的空间。每个元素的字节的初始值都是0。size必须包括对齐可能需要的填充任何元素，以便能够通过分配length · size
162 个字节来创建一个确定的数组，此时，length必须为正。如果length为负或size非正，会产生可检查的运行期错误，Array_new可能引发异常Mem_failed。

Array_free释放并清除*array。如果array或者*array为空则会产生可检查的运行期错误。

与本书中其他大多数建立空指针结构的ADT不同，Array接口对于元素的取值没有任何限制；每个元素都是size字节的序列。这样设计的基本原理是，Array_T更多地用于建立其他的ADT；第11章中所描述的队列就是一个例子。

函数

```
⟨exported functions 162⟩+≡
  extern int Array_length(T array);
  extern int Array_size  (T array);
```

返回array中的元素个数和它们的大小。

数组元素由下述代码来访问：

```
⟨exported functions 162⟩+≡
  extern void *Array_get(T array, int i);
  extern void *Array_put(T array, int i, void *elem);
```

Array_get返回第i个元素的指针；它同&a[i]类似，此时假定a是一个已经定义了的C语言数组。客户调用程序可以通过由Array_get返回的指针来访问元素的值。Array_put用一个由指针elem指向的新的元素值来覆盖第i个元素的值。不同于Table_put，Array_put返回值是elem。它不能返回第i个元素先前的值，因为该元素没有所必需的指针，并且它们长度可以为任意字节。

如果i大于或者等于数组array的长度，或者elem为空，会产生可检查的运行期错误。调用Array_get，然后在废除由Array_get返回的指针之前由Array_resize改变array的长度则会产生不可检查的运行期错误。从elem开始的存储无论以任何方式覆盖array的第i个元素的存储均会产生不可检查的运行期错误。

```
⟨exported functions 162⟩+≡
  extern void Array_resize(T array, int length);
163 extern T    Array_copy  (T array, int length);
```

Array_resize改变array的大小，以便它能容纳length个元素，并在必要的时候增加或减少。如果length超过了数组的当前长度，则新的元素被初始化成0。调用Array_resize将使先前调用Array_get所返回的值变得无效。Array_copy与此类似，但是它返回的是第一个有length个元素的数组array。如果length超过了array里的元素的个数，备份中超出部分元素的值初始化为0。Array_resize和Array_copy都可能引发异常Mem_Failed。

Array不具有和Table_map和Table_toArray类似的功能，因为Array_get已经提供了必要的机制来完成等价的操作。

在该接口中，把一个空的T传递给任何函数都将会产生可检查的运行期错误。

ArrayRep接口揭示了一个描述符（descriptor）的指针是如何表示一个Array_T的，其中，描述符是一个结构体，它的域包括数组元素的个数、元素的大小以及存储数组的指针。

```
⟨arrayrep.h⟩≡
  #ifndef ARRAYREP_INCLUDED
  #define ARRAYREP_INCLUDED

  #define T Array_T

  struct T {
      int length;
      int size;
      char *array;
  };

  extern void ArrayRep_init(T array, int length,
      int size, void *ary);

  #undef T
  #endif
```

图10-1显示了含有100个整数的数组的描述符，该数组是由机器上的Array_new（100，sizeof int）函数返回的，每个整数有四个字节。如果数组中没有元素，则array域为空。数组描述符有时被称为消息向量（dope vector）。

164

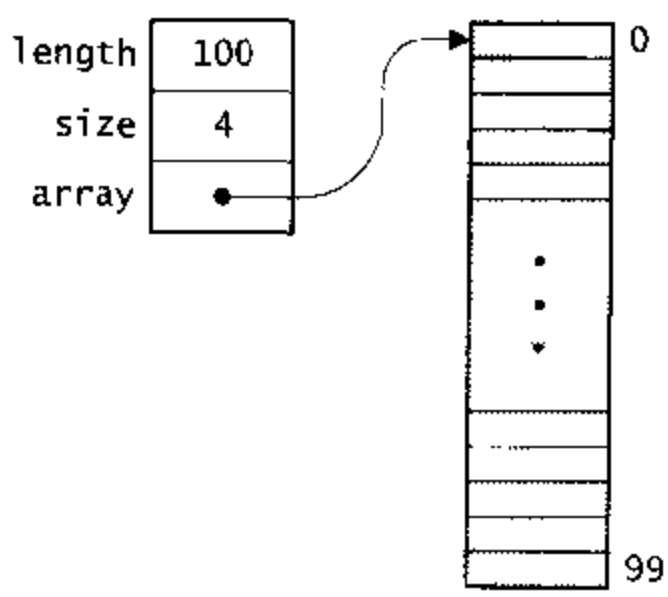


图10-1 由Array_new（100，sizeof int）创建的Array_T

ArrayRep的客户调用程序只能读取一个描述符的域，而不能对其进行写操作；对它们进行写操作将会产生不可检查的运行期错误。ArrayRep保证当array是一个T结构、i非负且小于array->length时，array->array + i*array->size是元素i的地址。

ArrayRep也输出ArrayRep_init，其中ArrayRep_init初始化了由array指向的结构Array_T中的域，分别给参数length、size和ary赋值。该函数使得客户调用程序能初始化它们已经插入到其他结构里的Array_T。如果array为空、size非正、length非零或ary非空，都会产生可检查的运行期错误。使用除调用Array Rep_init外的其他方法来初始化T，将会产生不可检查的运行期错误。

10.2 实现

165 单个动态数组实现会导出Array和ArrayRep两个接口：

```
⟨array.c⟩≡
  #include <stdlib.h>
  #include <string.h>
  #include "assert.h"
  #include "array.h"
  #include "arrayrep.h"
  #include "mem.h"

  #define T Array_T
  ⟨functions 166⟩
```

如果length为正，则Array_new为数组的描述符和数组本身分配空间，并调用ArrayRep_init来初始化描述符：

```
⟨functions 166⟩≡
  T Array_new(int length, int size) {
      T array;

      NEW(array);
      if (length > 0)
          ArrayRep_init(array, length, size,
              CALLOC(length, size));
      else
          ArrayRep_init(array, length, size, NULL);
      return array;
  }
```

ArrayRep_init是初始化描述符的惟一有效的方法；使用其他方法分配的描述符必须调用ArrayRep_init来对它们进行初始化。

```
⟨functions 166⟩+≡
  void ArrayRep_init(T array, int length, int size,
      void *ary) {
      assert(array);
```

```
    assert(ary && length>0 || length==0 && ary==NULL);
    assert(size > 0);
    array->length = length;
    array->size   = size;
  if (length > 0)
      array->array = ary;
  else
      array->array = NULL;
}
```

166

通过调用ArrayRep_init来初始化一个结构T有助于减少耦合：这些调用能清楚地识别为自己分配描述符并因此依赖于该表示的客户调用程序。只要ArrayRep_init不改变，可以在不影响这些客户调用程序的情况下来添加域。例如，当一个识别序列号的域被加到结构T中，并自动被ArrayRep_init初始化时，可能会出现这种情况。

Array_free释放数组本身和结构T，并清除它的参数：

```
⟨functions 166⟩+≡
  void Array_free(T *array) {
      assert(array && *array);
      FREE((*array)->array);
      FREE(*array);
  }
```

Array_free不用检查(*array)->array是否为空，因为FREE接受空指针。

Array_get和Array_put取出并存储Array_T中的元素：

```
⟨functions 166⟩+≡
  void *Array_get(T array, int i) {
      assert(array);
      assert(i >= 0 && i < array->length);
      return array->array + i*array->size;
  }

  void *Array_put(T array, int i, void *elem) {
      assert(array);
      assert(i >= 0 && i < array->length);
      assert(elem);
      memcpy(array->array + i*array->size, elem,
          array->size);
    return elem;
}
```

167

注意：Array_put返回它的第三个参数，而不是数组元素的字节所存储的地址。

Array_length和Array_size返回类似的指定描述符的域：

```
⟨functions 166⟩+≡
  int Array_length(T array) {
      assert(array);
      return array->length;
  }
```

```
int Array_size(T array) {
    assert(array);
    return array->size;
}
```

ArrayRep的客户调用程序能够从描述符中直接访问这些域。

Array_resize通过调用Mem的RESIZE来改变数组中的元素的个数，并相应地改变数组的length域。

```
⟨functions 166⟩+≡
  void Array_resize(T array, int length) {
      assert(array);
      assert(length >= 0);
      if (array->length == 0)
          array->array = ALLOC(length*array->size);
      else if (length > 0)
          RESIZE(array->array, length*array->size);
      else
          FREE(array->array);
      array->length = length;
  }
```

不同于使用Mem的RESIZE，零长度是合法的，此时数组被释放，而描述符此后将描述一个空的动态数组。

Array_copy与Array_resize非常相似，只是Array_copy把array的描述符和数组的一部分或者全部复制下来：

168

```
⟨functions 166⟩+≡
  T Array_copy(T array, int length) {
      T copy;

      assert(array);
      assert(length >= 0);
      copy = Array_new(length, array->size);
      if (copy->length >= array->length
      && array->length > 0)
          memcpy(copy->array, array->array, array->length);
      else if (array->length > copy->length
      && copy->length > 0)
          memcpy(copy->array, array->array, copy->length);
      return copy;
  }
```

参考书目浅析

一些语言支持各种动态数组。例如，Modula-3（Nelson 1991）允许在运行时建立任意范围的数组，但是不能扩展或者缩减它。Icon（Griswold and Griswold 1990）中的列表很像能够扩展或者缩减的数组，其中扩展或者缩减是通过在序列的末尾添加或者删除元素来实现

的，这一点非常像下一章将要描述的队列。Icon也支持从一个列表中取出子列表、用不同大小的列表来代替一个子列表等操作。

练习

10.1 设计并实现一个能提供关于指针的动态数组的ADT，它应该能够通过与Table提供的函数相类似的函数对这些数组的元素进行安全访问。在实现中使用Array和Array_Rep。

10.2 为一个动态矩阵（即两维数组）设计一个ADT，并用Array实现它。看看能否归纳出设计*N*维数组的方法？

169

10.3 为一个稀疏动态数组（即数组中大部分元素的值为零）设计并实现ADT。设计应该能接受一个特殊的零值数组，其实现只存储那些非零元素。

10.4 把下列函数

```
extern void Array_reshape(T array, int length,
    int size);
```

加入到array的接口和它的实现中。Array_reshape能够分别改变array数组的长度length和元素的大小size。如同Array_resize，重新形成后的数组保留原始数组的前length个元素；如果length超过了原始长度，超过部分元素的值设为0。array中第*i*个元素成为重新形成后的数组的第i个元素，如果size比原始长度小，则每个元素都被截断；如果size比原始值大，则超出部分的字节被设成0。

170

第11章 序 列

一个序列（sequence）含有N个值，用从0到N-1的整数索引，其中N为正数。一个空序列不含有值。同数组相似，序列中的值可以通过索引来访问，也可以在序列的任何一端增添和删除值。序列会自动做必要的扩展以容纳它们的内容。序列的值就是指针。

序列在本书中是最有用的ADT之一。尽管它们的定义都非常简单，但它们可以用作数组、列表、栈、队列和双端队列等，而且它们经常具有为数据结构分离ADT的能力。序列可以看做是非常抽象的数组，数组在前面章节已经讲过。序列隐藏了数据细节并在执行时恢复其细节。

11.1 接口

序列是一个在Seq接口中定义的隐式指针类型的实例：

```
⟨seq.h⟩≡
  #ifndef SEQ_INCLUDED
  #define SEQ_INCLUDED
  #define T Seq_T
  typedef struct T *T;
⟨exported functions 172⟩

#undef T
#endif
```


把一个空的T传递给该接口的任何程序都会产生可检查的运行期错误。

序列由下面的函数创建：

```
⟨exported functions 172⟩≡
  extern T Seq_new(int hint);
  extern T Seq_seq(void *x, ...);
```

Seq_new创建并返回一个空序列。hint是对这个新序列所容纳的最大值的一个估计。如果那个最大值不确定，则hint为零，并创建一个较小的序列。必要时序列会扩展容量而不管hint的值。如果hint为负，则会产生可检查的运行期错误。

Seq_seq创建并返回一个序列，它的初始化值在它的非空指针参数中。参数列表由第一个空指针来终止。因此，下述代码

```
Seq_T names;
...
names = Seq_seq("C", "ML", "C++", "Icon", "AWK", NULL);
```

创建了一个带有五个值的序列并把它赋给names，参数列表中的值与从0到4的索引相关联。在Seq_seq的可变参数列表变量部分传递的指针被假定为空指针，因此在传递字符型或void型以外类型的指针时，程序员必须提供强制类型转换，请参见7.1节的相关内容。Seq_new和Seq_seq都可以引发异常Mem_Failed。

```
⟨exported functions 172⟩+≡
  extern void Seq_free(T *seq);
```

释放并清除*seq。如果seq或者*seq为空指针，则会产生可检查的运行期错误。

```
⟨exported functions 172⟩+≡
  extern int Seq_length(T seq);
```

172

返回序列seq的值的数目。

*N*个值的序列中的值与从0到*N*−1的整数索引相关联。这些值能被下述函数访问：

```
⟨exported functions 172⟩+≡
  extern void *Seq_get(T seq, int i);
  extern void *Seq_put(T seq, int i, void *x);
```

Seq_get返回seq中的第i个值。Seq_put将第i个值改成x并返回原先的值。如果i大于或等于*N*，则会在运行时出现检测错误。Seq_get和Seq_put在固定的时间访问第i个值。

序列可以通过在两端中的任何一端增添值的方法来扩展。

```
⟨exported functions 172⟩+≡
  extern void *Seq_addlo(T seq, void *x);
  extern void *Seq_addhi(T seq, void *x);
```

Seq_addlo将x加到低端并返回x值。在序列的开始增加一个值将会使得已经存在的值的索引和序列的长度都增加1。Seq_addhi将x加到seq的高端并返回x值，在序列的末端增加一个值只使序列的长度加1。Seq_addlo和Seq_addhi可以引发异常Mem_Failed。

类似地，序列可以通过在seq任意一端删除值的方法来缩减：

```
⟨exported functions 172⟩+≡
  extern void *Seq_remlo(T seq);
  extern void *Seq_remhi(T seq);
```

Seq_remlo在seq的低端删除并返回一个值。在序列的开始删除一个值使得已经存在的值的索引和序列的长度都减1。Seq_remhi将在seq的高端删除并返回一个值。在序列的末端删除一个值只使序列的长度减1。Seq_remlo和Seq_remhi可以引发异常Mem_Failed。

173

11.2 实现

就像在本章开始说明的那样，序列是一个动态数组的高度抽象化。因此它的描述包含动态数组——不是一个指向Array_T的指针，而是Array_T结构本身——它的实现将导入Array和ArrayRep：

```
⟨seq.c⟩≡
  #include <stdlib.h>
  #include <stdarg.h>
  #include <string.h>
  #include "assert.h"
  #include "seq.h"
  #include "array.h"
  #include "arrayrep.h"
  #include "mem.h"

  #define T Seq_T

  struct T {
      struct Array_T array;
      int length;
      int head;
  };

  ⟨static functions 179⟩
  ⟨functions 175⟩
```

length域包含序列中的值的数目，array域包含序列中的值所存储的数组。该数组始终至少有length个元素，但是如果length小于array.length时，有一部分元素没有被使用。数组被用做循环的缓冲区以容纳序列值。序列的第0个值存储在数组头元素head中，连续的值就存储在以数组大小为模的连续的元素中。也就是说，如果序列的第i个值存储在第array.length-1个元素数中，则第i+1个值存储在数组的第0个元素中。图11-1显示了一个7值序列可以存储在一个16元素的数组中的一种方法。左边的表格是Seq_T和已经插入的Array_T，用浅色的阴影表示。右边的表格是数组，它的阴影区显示了被序列中的值所占用的元素。 174

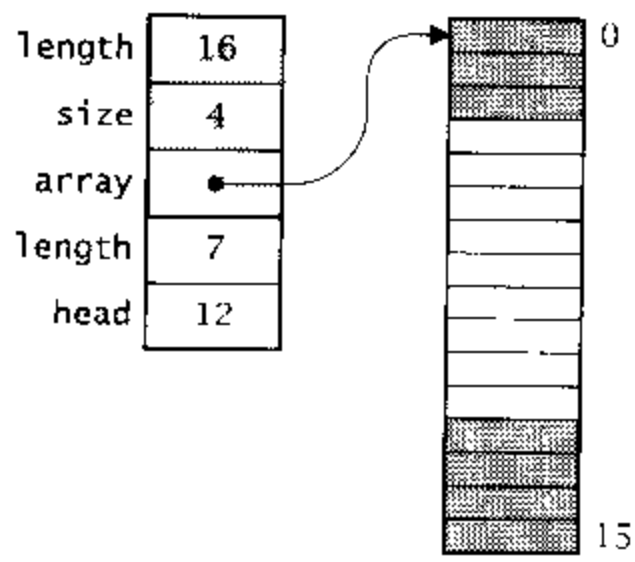


图11-1　16元素的序列

正如下面将要描述的那样，可通过以数组大小为模减小head的方法将数值加到序列的开头，又通过以数组大小为模增加head的方法将数值从序列开头删除。一个序列始终带有一个数组，即使序列是空的也不例外。

通过分配一个动态数组可以创建一个新的序列，该动态数组包含hint指针，当hint为0时

含有16个指针：

```
⟨functions 175⟩≡
  T Seq_new(int hint) {
      T seq;

      assert(hint >= 0);
      NEW0(seq);
      if (hint == 0)
          hint = 16;
      ArrayRep_init(&seq->array, hint, sizeof (void *),
          ALLOC(hint*sizeof (void *)));
      return seq;
  }
```

用NEW0把length和head域初始化为0。Seq_seq调用Seq_new，创建一个空序列，然后通过它的参数调用Seq_addhi，从而把每个值添加到这个新的序列中：

175

```
⟨functions 175⟩+≡
  T Seq_seq(void *x, ...) {
      va_list ap;
      T seq = Seq_new(0);

      va_start(ap, x);
      for ( ; x; x = va_arg(ap, void *))
          Seq_addhi(seq, x);
      va_end(ap);
      return seq;
  }
```

Seq_seq使用宏来处理变长参数列表，这一点很像List_list，请参见第7.2节相应内容。

Array_free可以释放一个序列，包括释放数组和它的描述符：

```
⟨functions 175⟩+≡
  void Seq_free(T *seq) {
      assert(seq && *seq);
      assert((void *)*seq == (void *)&(*seq)->array);
      Array_free((Array_T *)seq);
  }
```

能够调用Array_free仅仅是因为在代码中，有*seq的地址等于&(*seq)->array的断言。也就是说，结构Array_T一定是结构Seq_T的第一个域，以使在Seq_new中由NEW0返回的指针既指向Seq_T，又指向Array_T。

Seq_length只返回序列的length域：

```
⟨functions 175⟩+≡
  int Seq_length(T seq) {
      assert(seq);
      return seq->length;
  }
```

序列中第i个值存储在数组中的第（head + i）mod array.length个元素中。一个类型强制转换（type cast）使得直接对数组进行索引成为可能：

176

```
⟨seq[i] 177⟩≡
  ((void **)seq->array.array)[
      (seq->head + i)%seq->array.length]
```

Seq_get只返回该数组的元素，Seq_put把它设成x：

```
⟨functions 175⟩+≡
  void *Seq_get(T seq, int i) {
      assert(seq);
      assert(i >= 0 && i < seq->length);
      return ⟨seq[i] 177⟩;
  }

  void *Seq_put(T seq, int i, void *x) {
      void *prev;

      assert(seq);
      assert(i >= 0 && i < seq->length);
      prev = ⟨seq[i] 177⟩;
      ⟨seq[i] 177⟩ = x;
      return prev;
  }
```

Seq_remlo和Seq_remhi从一个序列中删除值。Seq_remhi在这两个函数当中更简单一些，因为它只减少length域并返回由一个新的length索引的值：

```
⟨functions 175⟩+≡
  void *Seq_remhi(T seq) {
      int i;

      assert(seq);
      assert(seq->length > 0);
      i = --seq->length;
      return ⟨seq[i] 177⟩;
  }
```

Seq_remlo稍微复杂一点，因为它必须返回由head索引的值（head的值在序列中由0索引），按照数组大小来增加head，并降低length：

177

```
⟨functions 175⟩+≡
  void *Seq_remlo(T seq) {
      int i = 0;
      void *x;

      assert(seq);
      assert(seq->length > 0);
      x = ⟨seq[i] 177⟩;
      seq->head = (seq->head + 1)%seq->array.length;
      --seq->length;
      return x;
  }
```

Seq_addlo和Seq_addhi都是将值加到序列中，因此考虑处理数组饱和的可能性，当length等于array.length时会出现这种情况。当满足上述条件时，这两个函数都将调用expand来扩展数组，这是通过调用Array_resize来实现的。同样，Seq_addhi是这两个函数中比较简单的一个，因为扩展检查后，它用由length给的索引来存储这个新值并增加length：

```
⟨functions 175⟩+≡
  void *Seq_addhi(T seq, void *x) {
      int i;

      assert(seq);
      if (seq->length == seq->array.length)
          expand(seq);
      i = seq->length++;
      return ⟨seq[i] 177⟩ = x;
  }
```

Seq_addlo也进行扩展检查，但是然后按照数组大小降低head，并把x存储在由新的head值索引的数组元素中，即是序列中由0索引的值：

```
⟨functions 175⟩+≡
  void *Seq_addlo(T seq, void *x) {
      int i = 0;
    assert(seq);
    if (seq->length == seq->array.length)
        expand(seq);
    if (--seq->head < 0)
        seq->head = seq->array.length - 1;
    seq->length++;
    return ⟨seq[i] 177⟩ = x;
  }
```

178

Seq_addlo用seq->head = Arith_mod(seq->head -1,seq->array.length)来降低seq->head。

expand封装了对Array_resize的调用，其中Array_resize使一个序列数组的大小加倍：

```
⟨static functions 179⟩≡
  static void expand(T seq) {
      int n = seq->array.length;

      Array_resize(&seq->array, 2*n);
      if (seq->head > 0)
          ⟨slide tail down 179⟩
  }
```

正如上述代码所显示的那样，expand也必须处理将数组作为圆形缓冲区的用法。除非head恰好为0，否则原始数组末端的元素——从head往下——必须移到扩展后的数组的末端以便能打开中间的元素，如图11-2所示，head也必须做相应的调整：

```
⟨slide tail down 179⟩≡
  {
      void **old = &((void **)seq->array.array)[seq->head];
```

```
    memcpy(old+n, old, (n - seq->head)*sizeof (void *));
    seq->head += n;
}
```

179

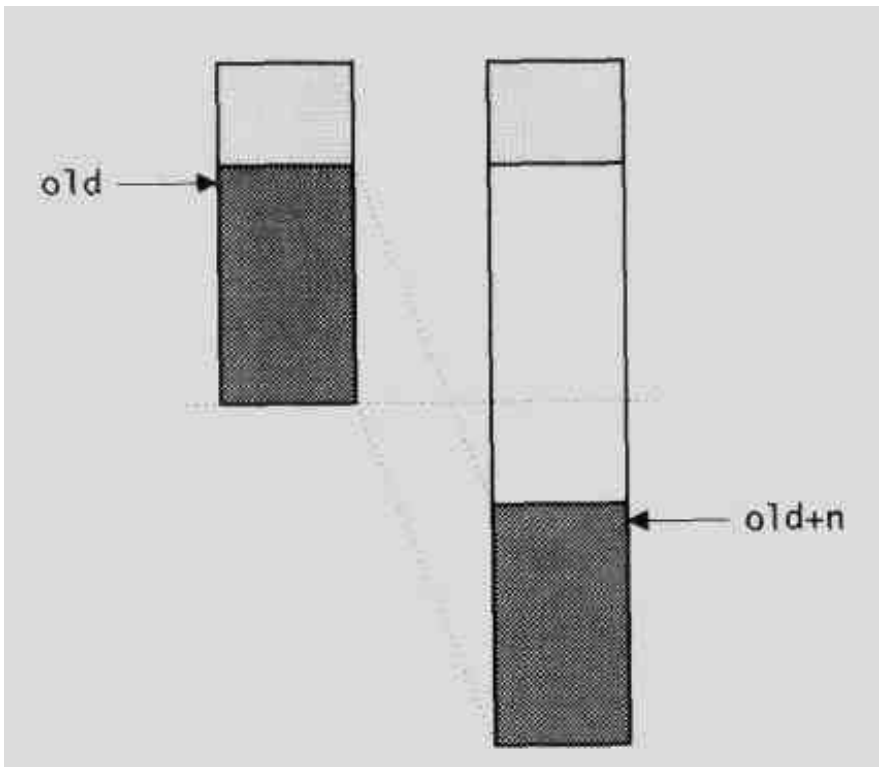


图11-2　扩展一个序列

参考书目浅析

序列与Icon中的列表几乎一致（Griswold and Griswold 1990），但是操作的名称来自库中的Sequence接口，该库与Modula-3（Horning，et al. 1993）的DEC实现联系在一起。本章中描述的实现也与DEC实现相似。练习11.1探讨了Icon的实现。

练习

11.1　Icon实现列表——它的序列版本——带有一个存储块的双链接列表，其中每个存储块有M个值。这种表示方法避免了使用Array_resize，因为如有必要，新的存储块可以加到列表的任意一端，以有效地调用Seq_addlo和Seq_addhi。这种表示方法的缺点是存储块必须移动访问第i个元素，所花的时间与i / M成比例。使用这种表示方法为Seq建立一个新的实现并开发一些测试程序来测试它的性能。假定对值i的访问几乎总是在对值i-1或值i+1的访问之后，你能修改实现使得这种情况能以固定的时间运行吗？ 180

11.2　为一个不使用Array_resize的Seq设计一个实现。例如，当N个元素的数组被填满后，它可能被转化成数组指针的数组，其中每个数组有$2N$个元素，因此转换后的序列可包含$2N^2$个值。如果N等于1024，则转换后的序列将有超过两百万个元素，每个元素都能在固定的时间被访问。在这种边缘矢量（edge-vector）表示法中，每个具有$2N$个元素的数组只要存有数值，都可以随意地分配。

11.3　假定禁止使用Seq_addlo和Seq_remlo，设计一个实现，使之能够递增地分配空间，

但是能在对数时间访问任何元素。提示：Skip list（Pugh 1990）。

11.4 序列只扩展而从不缩减。修改Seq->remlo和Seq->remhi，使之在数组中半数以上的元素没有使用时，即当seq->length小于seq->array.length/2时，可以缩减序列。什么时候不应做这样的修改？提示：thrashing（系统颠簸）。

11.5 再次用序列而不是集合来实现xref以保存行号。既然文件是按顺序读取的，而且因为它们都将按升序在序列中显示，所以不必对行号进行排序。

181 11.6 改写Seq_free，使它无需再使用断言。注意：不能使用Array_free。

第12章 环

环（ring）很像序列：它含有由N个从0到$N-1$的整数索引的N个值，其中N为正数。一个空的环没有值。所谓的值就是指针。同序列的值一样，环里的值也可以由索引来访问。

与序列不同的是，值可以添加到环的任何地方，并且环中的任何值都可以被删除。另外，值可以重新编号：向左“旋转”环以环的长度为模会降低每个值的索引，而向右“旋转”则以环的长度为模增加每个值的索引。以上做法的代价是，当访问第i个元素时，不能保证花费的时间是固定的。

12.1 接口

顾名思义，环是一个双链接的列表的抽象化，但是Ring ADT揭示了环只是一个隐式指针类型的实例：

```
⟨ring.h⟩≡
  #ifndef RING_INCLUDED
  #define RING_INCLUDED

  #define T Ring_T
  typedef struct T *T;

  ⟨exported functions 184⟩
  #undef T
  #endif
```


将一个空的环传递给该接口的任何程序都将会产生可检查的运行期错误。

创建环的函数与在Seq接口中所使用的函数类似：

```
⟨exported functions 184⟩≡
  extern T Ring_new (void);
  extern T Ring_ring(void *x, ...);
```

Ring_new创建并返回一个空的环。Ring_ring创建和返回一个环，它的值被初始化成它的非空指针参数。参数列表由第一个空指针参数来终止。因此

```
Ring_T names;
...
names = Ring_ring("Lists", "Tables", "Sets", "Sequences",
    "Rings", NULL);
```

创建了一个带有上面显示的5个值的环，并把它赋给names。参数列表中的值与从0到4的索引相关联。在环的参数列表的可变部分传递的指针被假定为空指针，因此当传递的不是空指

针或者字符型指针时，程序员必须提供强制转换，请参见7.1节的相关讨论。Ring_new和Ring_ring可以引发异常Mem_Failed。

```
⟨exported functions 184⟩+≡
  extern void Ring_free  (T *ring);
  extern int  Ring_length(T  ring);
```

Ring_free释放*ring指向的ring并清除*ring。如果ring或者*ring为空指针则会产生可检查的运行期错误。Ring_length返回在ring中的值的数目。

184 一个长度为N的环中的值由整数0到$N-1$索引。这些值由下列函数进行访问：

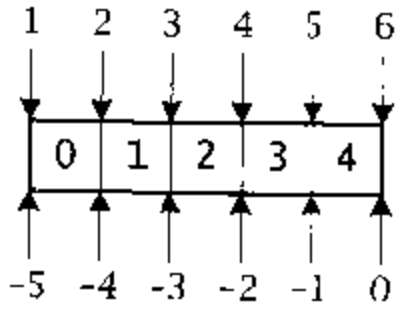
```
⟨exported functions 184⟩+≡
  extern void *Ring_get(T ring, int i);
  extern void *Ring_put(T ring, int i, void *x);
```

Ring_get返回ring中的第i个值。Ring_put把ring中的第i个值改成x并返回原先的值。如果i大于或者等于N则会产生可检查的运行期错误。

值通过下列函数添加到ring中的任何地方：

```
⟨exported functions 184⟩+≡
  extern void *Ring_add(T ring, int pos, void *x);
```

Ring_add在位置pos将x添加到ring中，并返回x。一个有N个值的ring中的位置如下图所示，这是一个带有从0到4的整数索引的五元素环。



图中中间一行数字表示索引，最上面一行是正的位置，最下面一行是非正的位置。非正的位置定义了环的末端具体的位置，它并不知道环的长度。位置0和1在空的环中也是有效的。Ring_add能接受任何形式的位置。定义一个并不存在的位置会产生可检查的运行期错误，不存在的位置包括超过环的长度的位置和负位置的绝对值超过环的长度的位置。

增加一个新的值使得它右边的值的索引和整个环的长度都增加一。Ring_add可以引发异常Mem_Failed。

函数

```
⟨exported functions 184⟩+≡
  extern void *Ring_addlo(T ring, void *x);
185 extern void *Ring_addhi(T ring, void *x);
```

等同于在Seq接口中命名的类似的函数。Ring_addlo同Ring_add（ring，1，x）等价，Ring_addhi同Ring_add(ring,0,x)等价。Ring_addlo和Ring_addhi都可以引发异常Mem_Failed。

函数

```
⟨exported functions 184⟩+≡
  extern void *Ring_remove(T ring, int i);
```

删除并返回ring中的第i个值。删除一个值将使得在它右面的部分的剩余的值索引减1。如果i大于或者等于ring的长度，则会产生可检查的运行期错误。

同Seq函数有类似的函数名，下列函数

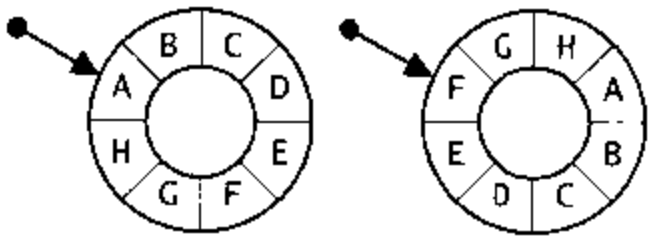
```
⟨exported functions 184⟩+≡
  extern void *Ring_remlo(T ring);
  extern void *Ring_remhi(T ring);
```

删除并返回ring中低端或者高端的值。Ring_remlo同Ring_remove(ring,0)等价，Ring_remhi同Ring_remove(ring, Ring_length(ring)-1)等价。将一个空的ring传递给Ring_remlo或Ring_remhi将会产生可检查的运行期错误。

“ring”的名字来自下列函数：

```
⟨exported functions 184⟩+≡
  extern void Ring_rotate(T ring, int n);
```

该函数通过左旋或者右旋对ring中的值进行重新编号。如果n为正，ring右旋（即顺时针旋转）n个值，每个值的索引也以ring的长度为模增加n。将一个含有元素A到H的ring向右旋转三次的情况如下图所示；其中箭头指向第一个元素。



186

如果n为负，ring向左旋转（即逆时针旋转）n个值，每个值的索引以ring的长度为模减少n。如果n以ring的长度为模是0，Ring_rotate无效。如果n的绝对值超过了ring的长度，则会产生可检查的运行期错误。

12.2 实现

实现把一个环描绘成一个带有两个域的结构：

```
⟨ring.c⟩≡
  #include <stdlib.h>
  #include <stdarg.h>
  #include <string.h>
  #include "assert.h"
  #include "ring.h"
  #include "mem.h"

  #define T Ring_T

  struct T {
```

```
    struct node {
        struct node *llink, *rlink;
        void *value;
    } *head;
    int length;
};
```

⟨*functions* 188⟩

head域指向一个node结构的双向链接列表，其中node结构里的value域含有ring中的值。head指向与索引0相关联的值，连续的值存放在由rlink域链接的node里面，并且每个node的llink域都指向它的前者。图12-1显示了一个带有六个值的ring的结构。点划线源自llink域并逆时针运动，而实线源自rlink域并顺时针旋转：

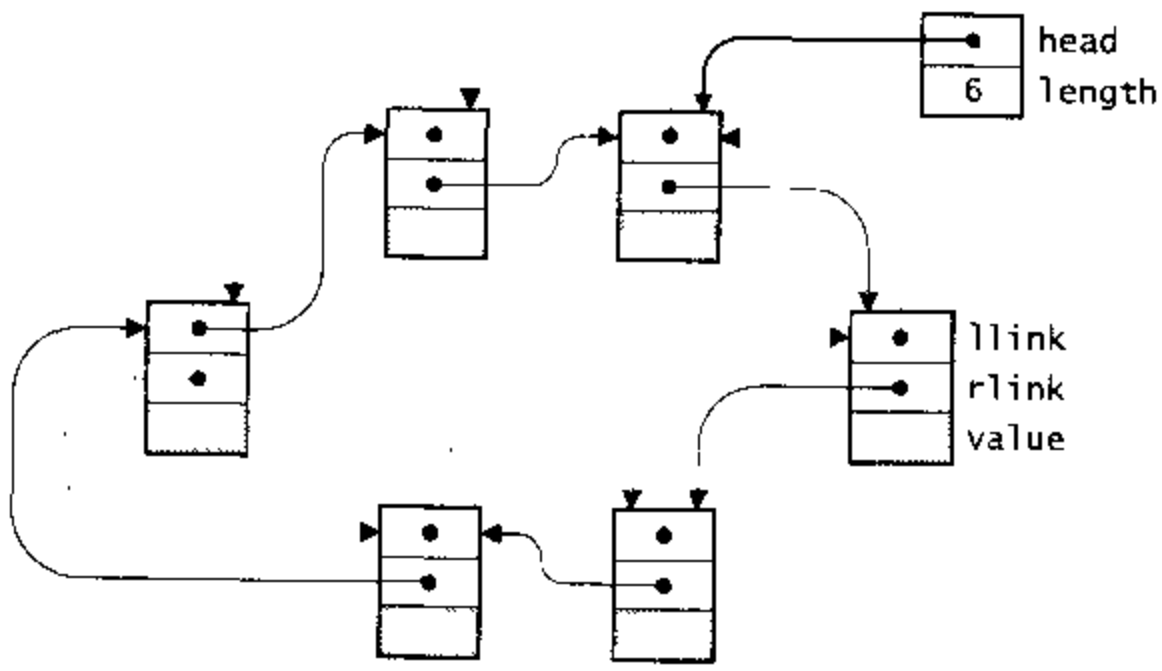


图12-1 一个六元素的环

```
⟨functions 188⟩≡
  T Ring_new(void) {
      T ring;

      NEW0(ring);
      ring->head = NULL;
      return ring;
  }
```

Ring_ring创建了一个空的环，然后调用Ring_addhi添加它的每个指针参数，直到第一个空指针为止，但不包含第一个空指针：

```
⟨functions 188⟩+≡
  T Ring_ring(void *x, ...) {
      va_list ap;
      T ring = Ring_new();

      va_start(ap, x);
      for ( ; x; x = va_arg(ap, void *))
```

```
            Ring_addhi(ring, x);
        va_end(ap);
        return ring;
    }
```

187 ~ 188

释放一个环首先要释放node结构，然后释放环头。因为node释放的顺序无关紧要，所以Ring_free只是以rlink指针的顺序释放。

```
⟨functions 188⟩+≡
  void Ring_free(T *ring) {
      struct node *p, *q;

      assert(ring && *ring);
      if ((p = (*ring)->head) != NULL) {
          int n = (*ring)->length;
          for ( ; n-- > 0; p = q) {
              q = p->rlink;
              FREE(p);
          }
      }
      FREE(*ring);
  }
```

函数：

```
⟨functions 188⟩+≡
  int Ring_length(T ring) {
      assert(ring);
      return ring->length;
  }
```

返回环中的值数。

Ring_get和Ring_put必须找到环中的第i个元素。这样算来，要把列表移动到第i个节点结构中，该操作是由下述函数完成的：

```
⟨q ← ith node 189⟩≡
  {
      int n;
      q = ring->head;
      if (i <= ring->length/2)
          for (n = i; n-- > 0; )
              q = q->rlink;
      else
          for (n = ring->length - i; n-- > 0; )
            q = q->llink;
  }
```

189

上述代码以最短的路径到达第i个节点：如果i不超过环长度的一半，第一次循环要通过rlink指针顺时针旋转到指定节点。否则，要通过llink逆时针旋转。例如，在图12-1中，0到3通过右旋来达到，4和5要通过左旋来达到。

给定了这个代码块后，这两个访问函数就很简单了：

```
⟨functions 188⟩+≡
  void *Ring_get(T ring, int i) {
      struct node *q;

      assert(ring);
      assert(i >= 0 && i < ring->length);
      ⟨q ← ith node 189⟩
      return q->value;
  }

  void *Ring_put(T ring, int i, void *x) {
      struct node *q;
      void *prev;

      assert(ring);
      assert(i >= 0 && i < ring->length);
      ⟨q ← ith node 189⟩
      prev = q->value;
      q->value = x;
      return prev;
  }
```

向环添加值的函数必须分配一个节点，对它进行初始化，并把它插入到双向链接列表的适当的位置。它们也必须能处理向一个空环添加节点的操作。Ring_addhi是这些函数中最简单的一个：它把一个新的节点插入到由head指向的节点的左边，如图12-2所示。深色的阴影用于区分新的节点，右边图中较重的线显示了那条链路改变了。下面是代码：

190

```
⟨functions 188⟩+≡
  void *Ring_addhi(T ring, void *x) {
      struct node *p, *q;

      assert(ring);
      NEW(p);
      if ((q = ring->head) != NULL)
          ⟨insert p to the left of q 191⟩
      else
          ⟨make p ring's only value 191⟩
      ring->length++;
      return p->value = x;
  }
```

向一个空环添加一个值很简单：ring->head指向这个新的节点，而节点的链接指向节点本身。

```
⟨make p ring's only value 191⟩≡
  ring->head = p->llink = p->rlink = p;
```

如图12-2所示的那样，Ring_addhi在环中的第一个节点就指向了q，并把新值插入到它的左边。这种插入包括初始化新节点的链，并改变q的llink和q的前者的rlink的方向：

```
⟨insert p to the left of q 191⟩≡
  {
      p->llink = q->llink;
      q->llink->rlink = p;
      p->rlink = q;
      q->llink = p;
  }
```

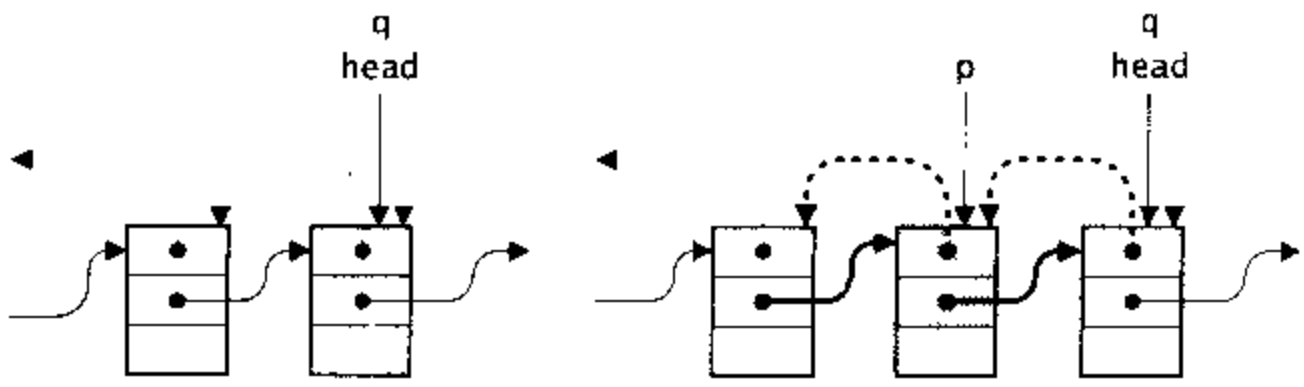


图12-2 将一个新值插入到head的左边

191

图12-3序列的第二到第五个图说明了这四个语句各自产生的影响。每一步中深色的曲线表示新的链。当q指向双向链接列表中的惟一的节点时，重新画出这个序列是有益的。

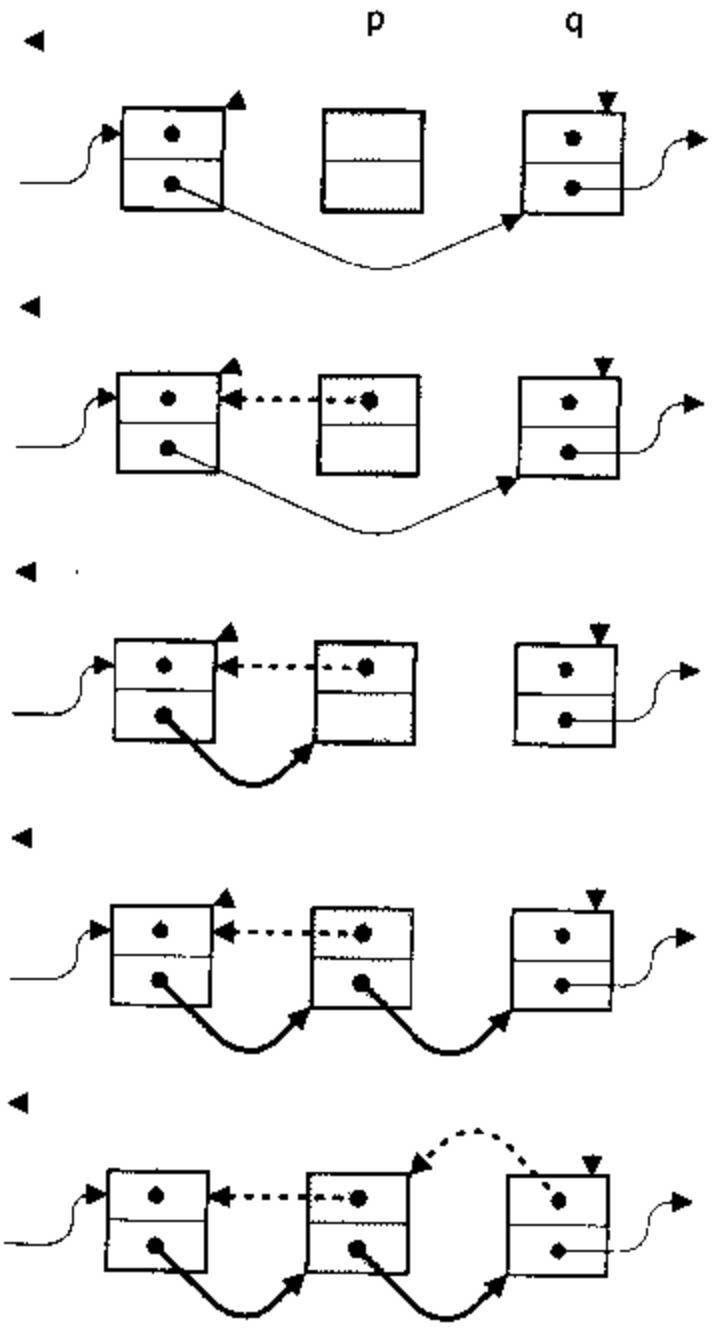


图12-3 将一个新的节点插入到q的左边

192

Ring_addlo几乎是一样的简单，只是这个新的节点成为环中的第一个节点。这种转换可用通过调用Ring_addhi然后再向右旋转一次来实现，而向右旋转是通过把head的值设成它的前者的值来实现的。

```
⟨functions 188⟩+≡
  void *Ring_addlo(T ring, void *x) {
      assert(ring);
      Ring_addhi(ring, x);
      ring->head = ring->head->llink;
      return x;
  }
```

Ring_add是这三个添加新值的函数里面最复杂的一个，因为它将处理前面讲述的任意位置的添加情形，包括环的两端。这些情况可以利用Ring_addlo和Ring_addhi处理末端添加问题的方式来解决，顺便注意环为空的情况。对于其他情况，转换值的索引的位置到右边，然后将新的节点添加到它的左边，如上所述。

```
⟨functions 188⟩+≡
  void *Ring_add(T ring, int pos, void *x) {
      assert(ring);
      assert(pos >= -ring->length && pos<=ring->length+1);
      if (pos == 1 || pos == -ring->length)
          return Ring_addlo(ring, x);
      else if (pos == 0 || pos == ring->length + 1)
          return Ring_addhi(ring, x);
      else {
          struct node *p, *q;
          int i = pos < 0 ? pos + ring->length : pos - 1;
          ⟨q ← ith node 189⟩
          NEW(p);
          ⟨insert p to the left of q 191⟩
          ring->length++;
          return p->value = x;
      }
  }
```

193

前两个if语句是关于环末尾的特殊位置。i的初始化处理与j1到ring->length-1的索引相应的位置。

这三个删除值的函数都比三个增加值的函数简单，因为删除值没有界限的限制，惟一的一个限制是删除环中最后一个值时需要注意。Ring_remove在这三个函数中是最典型的，它寻找第i个节点并从双向链接的列表中将它删除：

```
⟨functions 188⟩+≡
  void *Ring_remove(T ring, int i) {
      void *x;
      struct node *q;

      assert(ring);
      assert(ring->length > 0);
      assert(i >= 0 && i < ring->length);
```

```
        ⟨q ← ith node 189⟩
        if (i == 0)
            ring->head = ring->head->rlink;
        x = q->value;
        ⟨delete node q 194⟩
        return x;
    }
```

如果i为0，Ring_remove删除第一个节点，因此必须改变head的方向指向一个新的第一个节点。

增加一个节点需要对四个指针进行操作，而删除一个节点只需要对两个指针进行操作：

```
⟨delete node q 194⟩≡
  q->llink->rlink = q->rlink;
  q->rlink->llink = q->llink;
  FREE(q);
  if (--ring->length == 0)
      ring->head = NULL;
```

图12-4的第二和第三个图表说明在这个代码块的开始两行语句各自产生的影响。受影响的链用深色的曲线表示，在<*delete node q 194*>中第三条语句释放节点，最后两条语句减小ring的length，并在最后一个节点被删除时清除它的head指针。同样，画出从一个或两个节点的列表中删除一个节点的序列是有益的。 194

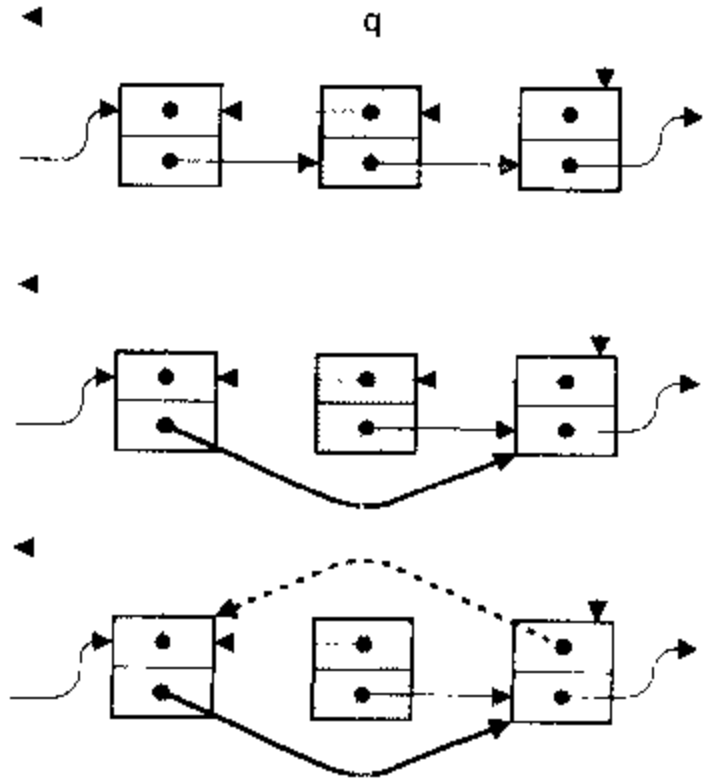


图12-4 删除节点q

Ring_remhi与此类似，只是寻找指定的节点更容易些：

```
⟨functions 188⟩+≡
  void *Ring_remhi(T ring) {
      void *x;
      struct node *q;

      assert(ring);
      assert(ring->length > 0);
```

```
        q = ring->head->llink;
        x = q->value;
        ⟨delete node q 194⟩
        return x;
    }
```

195 如上所示，Ring_addlo通过调用Ring_addhi和改变ring的head值使之指向它的前者来实现的。Ring_remlo的实现方式是：改变ring的head使之指向后者，并调用Ring_remhi。

```
⟨functions 188⟩+≡
  void *Ring_remlo(T ring) {
        assert(ring);
        assert(ring->length > 0);
        ring->head = ring->head->rlink;
        return Ring_remhi(ring);
  }
```

最后一个操作是旋转一个环。如果n为正，一个N值的环顺时针旋转，这就意味着以N为模的索引n成为新的head。如果n为负，环逆时针旋转，这意味着它的head移到索引n+N所指的值上。

```
⟨functions 188⟩+≡
  void Ring_rotate(T ring, int n) {
        struct node *q;
        int i;
        assert(ring);
        assert(n >= -ring->length && n <= ring->length);
        if (n >= 0)
            i = n%ring->length;
        else
            i = n + ring->length;
        ⟨q ← ith node 189⟩
        ring->head = q;
  }
```

这里，使用<*q<–ith node 189*>确保旋转路径最短。

参考书目浅析

Knuth（1973a）和Sedgewick（1990）都有详细处理双向列表的算法。

196 Icon提供的一些增加和删除列表中的值的操作与Ring提供的类似。练习12-4探讨了Icon的实现。Ring_add中定义位置的程序来自Icon。

练习

12.1 改写Ring_free中的循环，删除变量n；使用列表结构来决定循环何时结束。

12.2 仔细检查Ring_rotate的实现，解释为什么第二个if语句一定要写成i=n+ring->length。

12.3 调用Ring_get(ring，i)通常紧跟着另外一个调用，如Ring_get(ring,i+1)。修改实现，

使环能够记忆它最近访问的索引和相应的节点，并利用该信息尽可能的避免在*<q<-ith node 189>*中循环。不要忘了当增加或者删除值时要更新信息。设计一个测试程序，来证明你的改善带来的好处。

12.4 Icon实现列表间环相似，同带有*N*个值的数组的双向链接链表也类似。这些数组被用做圆形缓冲区，就像数组在Seq的实现中那样。寻找第i个值可以近似地向下搜索环列表中的i／*N*个数组，然后计算数组中第i个值的索引。增加一个值，要么把它加到一个已经存在的数组的空闲位置，要么增加到一个新的数组。删除一个值则空出数组中的一个位置。而且，如果要删除的值是数组中最后一个，则此时也从列表中删除并释放。这种表示方法比本章中描述的要复杂得多，但是它对于较大的环性能较好。使用这种方法再次实现环，并测试这两种实现的性能。环要多大才能检测到性能的改善？

197

第13章 位 向 量

第9章讲述的集合可以带有任意个元素，因为这些元素仅仅由客户调用程序提供的函数来进行处理。整数集合灵活性更差一些，但是使用它们通常就足够确定一个独立的ADT。Bit接口导出处理位向量（bit vector）的函数，位向量可以表示从0到$N-1$的整数集合。例如，256比特的向量可有效地表示字符型集合。

Bit提供了大多数Set提供的集合处理函数，还有少量函数是专门针对位向量的。不同于由Set提供的集合，由位向量提供的集合有一个非常好的定义域，是从0到$N-1$的整数型集合。因此，Bit可以提供Set不能提供的函数，如一个集合的补集（complement）。

13.1 接口

位向量这个名字解释了整数型集合的表示方法在本质上是比特序列。然而，Bit接口只导出一个表示一个位向量的隐式类型：

```
⟨bit.h⟩≡
  #ifndef BIT_INCLUDED
  #define BIT_INCLUDED

  #define T Bit_T
  typedef struct T *T;

⟨exported functions 200⟩

#undef T
#endif
```


当Bit_new创建一个位向量后，该位向量的长度就固定了：

```
⟨exported functions 200⟩≡
  extern T   Bit_new   (int length);
  extern int Bit_length(T set);
  extern int Bit_count (T set);
```

Bit_new创建了一个新的length个位长的向量，并把所有的位都设为0。位向量表示包含了从0到length-1位的整数集合。如果length为负则会产生可检查的运行期错误。Bit_new可以引发异常Mem_Failed。

Bit_length返回set中的位数，而Bit_count返回set中1的数目。

除Bit_union、Bit_inter、Bit_minus和Bit_diff之外，将一个空的T传递给该接口的任何例程都会产生可检查的运行期错误。

```
⟨exported functions 200⟩+≡
  extern void Bit_free(T *set);
```

释放并清除*set。如果set或者*set为空，则会产生可检查的运行期错误。

set中的各个元素（它的向量中的每一位）由下述函数来处理：

```
⟨exported functions 200⟩+≡
  extern int Bit_get(T set, int n);
  extern int Bit_put(T set, int n, int bit);
```

Bit_get返回比特n值并测试n是否在set中；即Bit_get当比特n在set中时返回1而当n不在set中时返回0。Bit_put根据bit在位向量中设置比特n并返回比特n对应的位向量中先前值。n为负或者大于等于set的长度，或者bit既不是0也不是1时，会产生可检查的运行期错误。

上述函数处理set中的各个位；而下列函数则处理set中的相邻的比特序列——一个set的
200 子集。

```
⟨exported functions 200⟩+≡
  extern void Bit_clear(T set, int lo, int hi);
  extern void Bit_set  (T set, int lo, int hi);
  extern void Bit_not  (T set, int lo, int hi);
```

Bit_clear清除从lo到hi的比特位，Bit_set设置从lo到hi的比特位，Bit_not取从lo到hi比特位的补集。如果lo超过hi、lo或者hi为负，以及lo或者hi大于等于set的长度，都会产生可检查的运行期错误。

```
⟨exported functions 200⟩+≡
  extern int Bit_lt (T s, T t);
  extern int Bit_eq (T s, T t);
  extern int Bit_leq(T s, T t);
```

Bit_lt在s ⊂t时返回1，否则返回0。如果s ⊂t，则s是t的子集。如果s ⊂t，则Bit_eq返回1、否则返回0。如果s ⊆t，则Bit_leq返回1，否则返回0。对所有这三个函数，如果s和t的长度不等，则会产生可检查的运行期错误。

函数：

```
⟨exported functions 200⟩+≡
  extern void Bit_map(T set,
      void apply(int n, int bit, void *cl), void *cl);
```

从零比特位开始，为set中的每个比特位调用apply函数。n是比特序号，其值从0到集合的长度减1，bit是位 n的值，cl由客户调用程序提供。不同于传递给Table_map的函数，apply可以改变set的值。如果为比特n调用apply却改变了比特k的值，其中k>n，则该变化就会在随后对apply的调用中看到，因为Bit_map必须在合适的位置来处理比特位。否则就要在处理它的比特之前让Bit_map对向量进行备份。

下述函数执行四个标准的集合操作，在第9章中已经有所讲述。每个函数都返回一个新的集合，其值就是操作结果。

```
⟨exported functions 200⟩+≡
  extern T Bit_union(T s, T t);
  extern T Bit_inter(T s, T t);
  extern T Bit_minus(T s, T t);
  extern T Bit_diff (T s, T t);
```

201

Bit_union返回s和t的并集，用s + t表示，是两个位向量做或（OR）运算。Bit_inter返回s和t的交集，用s*t表示，是两个位向量做逻辑与（AND）运算。Bit_minus返回s和t的差，用s–t表示，是s和t的补做逻辑与运算。Bit_diff返回s和t的对称差分（symmetric difference），用s / t表示，是s和t做异或（exclusive OR）运算。

这四个函数都接受s或t的空指针（但不能s和t均为空），并把对应集合解释成空集合。Bit_union（s，NULL）返回s的一个副本。这些函数总是返回非空的T。s和t均为空，会产生可检查的运行期错误，s和t有不同的长度也会产生可检查的运行期错误。这些函数都可引发异常Mem_Failed。

13.2 实现

Bit_T是带有位向量的长度和比特本身的结构的指针：

```
⟨bit.c⟩≡
  #include <stdarg.h>
  #include <string.h>
  #include "assert.h"
  #include "bit.h"
  #include "mem.h"

  #define T Bit_T

  struct T {
      int length;
      unsigned char *bytes;
      unsigned long *words;
  };

  ⟨macros 203⟩
  ⟨static data 207⟩
  ⟨static functions 212⟩
  ⟨functions 203⟩
```

202

length域给出了向量中的比特数，bytes指向至少 |length / 8] 个字节。通过索引bytes可以访问比特，bytes [i] 指向带有从8 · i到8 · i + 7比特的字节，其中8 · i是字节中最低有效数字位，注意，该规定只用于每个字符8位的情况；对于超过8位的机器，则超出的位将不使用。

如果所有访问单个比特的操作，比如Bit_get，都使用相同的约定来访问位，则可以将比特存储到无符号长整型数组中来进行处理。Bit使用字符数组来允许表驱动实现Bit_count、Bit_set、Bit_clear和Bit_not。

一些操作，比如Bit_union，并行处理所有的比特位。对于这些操作，通过words可以在同一时间访问BPW个比特的向量

```
⟨macros 203⟩≡
  #define BPW (8*sizeof (unsigned long))
```

其中words必须指向一个无符号长整型数；nwords计算一个带有length比特位的位向量所含的长整型数的个数：

```
⟨macros 203⟩+≡
  #define nwords(len) ((((len) + BPW - 1)&(~(BPW-1)))/BPW)
```

Bit_new使用nwords来分配一个新的T：

```
⟨functions 203⟩≡
  T Bit_new(int length) {
      T set;

      assert(length >= 0);
      NEW(set);
      if (length > 0)
          set->words = CALLOC(nwords(length),
              sizeof (unsigned long));
      else
          set->words = NULL;
      set->bytes = (unsigned char *)set->words;
      set->length = length;
      return set;
  }
```

Bit_new至多可以分配sizeof（unsigned long）-1个额外的字节。这些额外的字节必须为0以使下列函数能正确地执行。

Bit_free释放了集合并清除它的参数，Bit_length返回length域。

```
⟨functions 203⟩+≡
  void Bit_free(T *set) {
      assert(set && *set);
      FREE((*set)->words);
      FREE(*set);
  }

  int Bit_length(T set) {
      assert(set);
      return set->length;
  }
```

13.2.1 成员操作

Bit_count返回集合中的成员数——即集合中每一个位值为1的个数。该函数可以简单地遍历集合并测试每个比特位，但是它使用两个“半字节”（即它的两个“四比特半字节”），

同使用表的索引一样为所有16个可能的半字节提供位值为1的个数：

```
⟨functions 203⟩+≡
  int Bit_count(T set) {
      int length = 0, n;
      static char count[] = {
          0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4 };

      assert(set);
      for (n = nbytes(set->length); --n >= 0; ) {
          unsigned char c = set->bytes[n];
          length += count[c&0xF] + count[c>>4];
      }
      return length;
  }
```

204

```
⟨macros 203⟩+≡
  #define nbytes(len) ((((len) + 8 - 1)&(~(8-1)))/8)
```

nbytes计算 [len / 8]，它用于进行对位向量一次排序一位的操作。循环的每个执行都通过将length和字节的两个四比特半字节的比特数值之和相加来计算集合中第n字节中的比特数。该循环可以访问外部存储空间的比特位，但是因为Bit_new将所有比特均初始化为0，所以这种访问不能改变函数执行结果。

比特n是字节n / 8中的位数为n%8的表示数值，一个字节中的位数从0开始并从右向左增加，即最低有效数字位是位0，最高有效数字位是位7。Bit_get返回比特n的值，是通过将字节n / 8向右移动n%8位并只返回最右边的位的方法来实现的：

```
⟨functions 203⟩+≡
  int Bit_get(T set, int n) {
      assert(set);
      assert(0 <= n && n < set->length);
      return ⟨bit n in set 205⟩;
  }

⟨bit n in set 205⟩≡
  ((set->bytes[n/8]>>(n%8))&1)
```

Bit_put使用相同的方式设置比特n：如果bit为1，Bit_put将比特1向左移动n%8位，并将其结果与set中第n / 8字节内容进行或运算。

```
⟨functions 203⟩+≡
  int Bit_put(T set, int n, int bit) {
      int prev;

      assert(set);
      assert(bit == 0 || bit == 1);
      assert(0 <= n && n < set->length);
      prev = ⟨bit n in set 205⟩;
      if (bit == 1)
```

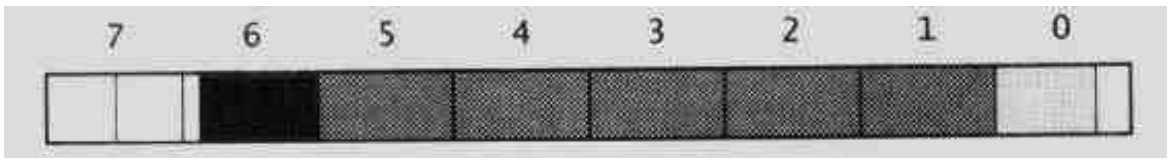
205
```
            set->bytes[n/8] |=   1<<(n%8);
        else
            set->bytes[n/8] &= ~(1<<(n%8));
        return prev;
    }
```

如上所示，Bit_put通过形成一个掩码来清除比特n，在该掩码中，第n%8位是0而其他位都是1，然后将这个掩码与set中第n/8字节内容进行与运算。

Bit_set、Bit_clear和Bit_not都采用相似的手段来设置set中的某段位范围，或清除和取补集，但是它们的处理更加复杂，因为它们必须处理范围超过字节边界限制的情况。例如，如果set有60位，函数

```
Bit_set(set, 3, 54)
```

将设置首字节中第3到第7位、第一到第五字节中所有的位和第六字节中的第0到第6位，其中字节是从0开始计数的。这三个从右到左的区域，如下图，分别用不用的阴影表示。



第七字节中高四位比特位不使用，因此始终为0。Bit_set函数代码块反应了这三个区域：

```
⟨functions 203⟩+≡
  void Bit_set(T set, int lo, int hi) {
      ⟨check set, lo, and hi 206⟩
      if (lo/8 < hi/8) {
          ⟨set the most significant bits in byte lo/8 207⟩
          ⟨set all the bits in bytes lo/8+1..hi/8-1 207⟩
          ⟨set the least significant bits in byte hi/8 207⟩
      } else
          ⟨set bits lo%8..hi%8 in byte lo/8 208⟩
  }

⟨check set, lo, and hi 206⟩≡
```

206
```
      assert(set);
      assert(0 <= lo && hi < set->length);
      assert(lo <= hi);
```

当lo和hi指向不同字节中的位时，在字节lo/8中设置的位数将依赖于lo%8：如果lo%8为0，将设置所有的比特位，如果为7，只设置最高有效数字位。所有的可能性都存储在由lo%8索引的掩码表中：

```
⟨static data 207⟩≡
  unsigned char msbmask[] = {
      0xFF, 0xFE, 0xFC, 0xF8,
      0xF0, 0xE0, 0xC0, 0x80
  };
```

msbmask[lo%8]与字节lo/8取或将设置适当的位：

```
⟨set the most significant bits in byte lo/8 207⟩≡
  set->bytes[lo/8] |= msbmask[lo%8];
```

在第二个区域，每个字节的所有的位都设为1。

```
⟨set all the bits in bytes lo/8+1..hi/8-1 207⟩≡
  {
      int i;
      for (i = lo/8+1; i < hi/8; i++)
          set->bytes[i] = 0xFF;
  }
```

hi%8决定着设置字节hi / 8中哪些位：如果hi%8为0，只设置最低有效数字位；如果是7，将设置所有的比特位。同样，hi%8可用作一个表的索引，以便选择适当的掩码与hi / 8取或：

```
⟨set the least significant bits in byte hi/8 207⟩≡
  set->bytes[hi/8] |= lsbmask[hi%8];

⟨static data 207⟩+≡
  unsigned char lsbmask[] = {
      0x01, 0x03, 0x07, 0x0F,
      0x1F, 0x3F, 0x7F, 0xFF
  };
```

207

当lo和hi指向同一个字节中的位时，由msbmask［lo%8］和lsbmask［hi%8］提供的掩码可以结合起来设置适当的比特位。例如：

```
Bit_set(set, 9, 13)
```

将设置集合中第二个字节的第一到第五位，这是在掩码0x3E中取或完成的，该掩码是msbmask [1] 和lsbmask [5] 做与运算得来的。一般情况下，这两个掩码在应该设置的比特位部分正好重叠，因此处理这种情况的代码如下所示：

```
⟨set bits lo%8..hi%8 in byte lo/8 208⟩≡
  set->bytes[lo/8] |= ⟨mask for bits lo%8..hi%8 208⟩;

⟨mask for bits lo%8..hi%8 208⟩≡
  (msbmask[lo%8]&lsbmask[hi%8])
```

Bit_clear和Bit_not同Bit_set处理类似，msbmask和lsbmask的使用方法也类似。对Bit_clear而言，msbmask和lsbmask提供了分别与字节lo / 8和hi / 8做与的掩码的补：

```
⟨functions 203⟩+≡
  void Bit_clear(T set, int lo, int hi) {
      ⟨check set, lo, and hi 206⟩
      if (lo/8 < hi/8) {
          int i;
          set->bytes[lo/8] &= ~msbmask[lo%8];
          for (i = lo/8+1; i < hi/8; i++)
              set->bytes[i] = 0;
          set->bytes[hi/8] &= ~lsbmask[hi%8];
      } else
```

```
            set->bytes[lo/8] &= ~⟨mask for bits lo%8..hi%8 208⟩;
    }
```

Bit_not必须翻转从lo到hi的比特位，这是通过与掩码的或非运算覆盖适当的比特位来实现的：

```
⟨functions 203⟩+≡
  void Bit_not(T set, int lo, int hi) {
208       ⟨check set, lo, and hi 206⟩
      if (lo/8 < hi/8) {
          int i;
          set->bytes[lo/8] ^= msbmask[lo%8];
          for (i = lo/8+1; i < hi/8; i++)
              set->bytes[i] ^= 0xFF;
          set->bytes[hi/8] ^= lsbmask[hi%8];
      } else
          set->bytes[lo/8] ^= ⟨mask for bits lo%8..hi%8 208⟩;
  }
```

Bit_map为集合中的每个位都调用apply函数。它传递位序号、它的值和客户调用程序提供的指针。

```
⟨functions 203⟩+≡
  void Bit_map(T set,
      void apply(int n, int bit, void *cl), void *cl) {
      int n;
      assert(set);
      for (n = 0; n < set->length; n++)
          apply(n, ⟨bit n in set 205⟩, cl);
  }
```

如上所示，Bit_map用与隐含在Bit_get和那些用位序号作为参数的Bit函数中的相同的编号方式来传递位。n／8的值每八个字节只改变一次，因此它试图将每个字节从set->bytes[n/8]中复制到一个临时变量，然后一次次地通过移位和掩码运算来取出每个比特位。但是这种改变破坏了接口，接口规定如果apply改变了它还没有看到的比特位，则它将在随后的调用中看到新的值。

13.2.2 比较

Bit_eq比较集合s和t，如果它们相等则返回1，如果不等则返回0。这是通过比较s和t中相对应的无符号长整型数来完成的，并且一旦知道了s不等于t，则立即退出循环：

```
⟨functions 203⟩+≡
  int Bit_eq(T s, T t) {
209       int i;
      assert(s && t);
      assert(s->length == t->length);
      for (i = nwords(s->length); --i >= 0; )
```

```
        if (s->words[i] != t->words[i])
            return 0;
    return 1;
}
```

Bit_leq比较集合s和t并决定s是等于t还是t的子集。如果s ⊆t，那么对s中的每一个值为1的位，t中相应的位都为1。按照集合的定义，如果s和t的补集的交集为空，则s ⊆t。因此，如果s&~t等于零，则s ⊆t；这种关系在s和t中的每个无符号长整型数中也都存在。如果对于所有的i，s->u.words[i] ⊆t->u.words[i]，则s ⊆t。一旦知道了输出结果，Bit_leq利用这种属性立即停止比较：

```
⟨functions 203⟩+≡
  int Bit_leq(T s, T t) {
      int i;

      assert(s && t);
      assert(s->length == t->length);
      for (i = nwords(s->length); --i >= 0; )
          if ((s->words[i]&~t->words[i]) != 0)
              return 0;
      return 1;
  }
```

如果s是t的一个子集，则Bit_lt返回值是1；如果s ⊆t且s ≠t，则有s ⊂t，这是通过确保s->u.words[i]&~t->u.words[i]为零并且至少有一个s->u.words[i]不等于对应的t->u.words[i]来实现的：

```
⟨functions 203⟩+≡
  int Bit_lt(T s, T t) {
      int i, lt = 0;

      assert(s && t);
      assert(s->length == t->length);
      for (i = nwords(s->length); --i >= 0; )
          if ((s->words[i]&~t->words[i]) != 0)
              return 0;
          else if (s->words[i] != t->words[i])
              lt |= 1;
      return lt;
  }
```

210

13.2.3 集合操作

实现集合操作s+t、s*t、s-t和s/t的函数可以一次处理一个长整型操作数，因为这些函数的功能是独立于位序号的。这些函数也把空T看作是一个空的集合，但是s和t中的一个集合必须非空，以便函数决定结果的长度。这些函数有相似的实现代码，除了有三个处理的不同之处：s和t指向同一个集合时的结果、它们对空参数进行的操作和它们怎样为两个非空集合形成结果。代码处理的相似性可以通过定义宏setop来实现：

```
⟨macros 203⟩+≡
  #define setop(sequal, snull, tnull, op) \
      if (s == t) { assert(s); return sequal; } \
      else if (s == NULL) { assert(t); return snull; } \
      else if (t == NULL) return tnull; \
      else { \
          int i; T set; \
          assert(s->length == t->length); \
          set = Bit_new(s->length); \
          for (i = nwords(s->length); --i >= 0; ) \
              set->words[i] = s->words[i] op t->words[i]; \
          return set; }
```

Bit_union代表了这些函数：

```
⟨functions 203⟩+≡
  T Bit_union(T s, T t) {
      setop(copy(t), copy(t), copy(s), |)
  }
```

如果s和t指向同一个集合，函数执行结果即为集合的副本。如果s或t为空，函数执行结果即
211 为另外一个非空的集合的副本。否则，函数执行结果是一个集合，该集合的无符号长整型值是s和t中的无符号长整型值的按位取或。

私有函数copy分配一个新的与参数长度相同的集合并从它的参数里复制位：

```
⟨static functions 212⟩≡
  static T copy(T t) {
      T set;

      assert(t);
      set = Bit_new(t->length);
      if (t->length > 0)
          memcpy(set->bytes, t->bytes, nbytes(t->length));
      return set;
  }
```

如果它的任何一个参数为空的话，Bit_inter将返回一个空的集合；否则，它返回一个操作数按位取与的集合：

```
⟨functions 203⟩+≡
  T Bit_inter(T s, T t) {
      setop(copy(t),
          Bit_new(t->length), Bit_new(s->length), &)
  }
```

如果s为空，则 s − t 是一个空集合，但是如果t为空，s − t 等于s。如果s和t均为非空，s − t 是s和t的补按位取与，当s和t是相同的Bit_T时，s − t 是一个空集。

```
⟨functions 203⟩+≡
  T Bit_minus(T s, T t) {
      setop(Bit_new(s->length),
          Bit_new(t->length), copy(s), & ~)
  }
```

setop的第三个参数& ~导致的循环体是：

```
set->words[i] = s->words[i] & ~t->words[i];
```

Bit_diff实现了对称差分，用s / t表示，是s和t按位取或非运算。如果s为空，s / t等于t，反之亦然。 212

```
(functions 203)+≡
  T Bit_diff(T s, T t) {
      setop(Bit_new(s->length), copy(t), copy(s), ^)
  }
```

如上所示，当s和t指向同一个Bit_T时，s / t为空集合。

参考书目浅析

Briggs和Torczon（1993）描述了一个集合的表示方法，它被设计成大的、稀疏的集合，并可以在固定的时间初始化这些集合。Gimpel（1974）引入了空间多元集合，练习13.5将对此进行描述。

练习

13.1 在稀疏集合中，大部分位都为0。修改Bit的实现，通过不存储集合中的大量的0元素的方法来节省稀疏集合的空间。

13.2 设计一个接口，使之能支持Briggs和Torczon（1993）描述的稀疏集合，并实现该接口。

13.3 Bit_set用循环

```
for (i = lo/8+1; i < hi/8; i++)
    set->bytes[i] = 0xFF;
```

来设置从字节lo / 8+1到hi / 8的所有比特位。Bit_clear和Bit_not有相似的循环。如果可能，修改这些循环对无符号长整型数进行清除、设置和取与运算，而不是对字节进行操作。你能找到一个能在执行期间出现可度量的性能改善的应用吗？

13.4 假定Bit函数跟踪一个集合中的值为1的比特数。Bit函数能简化或者改善什么？实现这个想法并设计一个测试程序来确定提高的性能质量。说明在什么条件下值得付出一定的代价来获得相应的好处？ 213

13.5 在一个空间多元集合（spatially multiplexed set）中，比特按字存储。在32位整型计算机上，一个带有*N*个无符号整型的数组可以保存32个*N*-比特的集合。数组的每个只含1比特位的列都是一个集合。一个只设置了比特i的32位的掩码能识别出列i的集合。这种表示方法的优点是只执行这些掩码就可以在固定的时间内完成一些操作。例如，对于两个集合的合集，其掩码就是两个执行合集操作的集合掩码的合集。许多*N*-比特的集合都可以共享一个*N*-字的数组；分配一个新的集合也就是在数组中分配一个自由列。这种属性可以节省空间但给存储管理增加了相当的复杂度，因为实现必须跟踪这个*N*-字的数组，该数组具有*N*位的任何值的自由列。用这种方法重新实现Bit；如果需要强制改变接口，则重新设计一个。 214

第14章 格 式 化

标准C库函数printf、fprintf和vprintf格式化输出数据，sprintf和vsprintf将数据格式化到字符串中。这些函数同一个格式化字符串和一个参数列表一起被调用，这个参数列表中的值将被格式化。格式化是由插入到字符串的形式为%c的转化说明符（conversion specifier）控制的。%c的第i次出现将描述如何格式化参数列表中的第i个参数，参数列表在格式字符串后面。而其他的字符被逐字复制。例如，如果name是字符串“Array”，count为8，则

```
sprintf(buf, "The %s interface has %d functions\n",
    name, count)
```

用字符串“The Array interface has 8 functions\n”填充buf，其中\n通常是换行符。转换说明符也可以包含宽度、精度和填充规范（padding specification）。例如，在上面的格式字符串中用%06d代替%d将会用字符串“The Array interface has 000008 functions\n”填充buf。

这些函数虽然非常有用，但是，至少有四个缺点：第一，转换说明符的设置是固定的，没有办法来提供基于客户调用程序的代码；第二，被格式化的结果只能在字符串中被打印或者存储，没有办法定义一个基于客户调用程序的输出程序；第三个也是最危险的一个缺点是sprintf和vsprintf可能试图输出比它们所能存储的字符串更多的字符，没有方法规定输出字符串的大小；最后，对在参数列表的变量部分传递的参数无法进行格式检查。Fmt接口解决了前三个缺点。

14.1 接口

Fmt接口有11个函数、一种类型、一个变量和一个异常：

```
⟨fmt.h⟩≡
  #ifndef FMT_INCLUDED
  #define FMT_INCLUDED
  #include <stdarg.h>
  #include <stdio.h>
  #include "except.h"

  #define T Fmt_T
  typedef void (*T)(int code, va_list *app,
      int put(int c, void *cl), void *cl,
      unsigned char flags[256], int width, int precision);

  extern char *Fmt_flags;
  extern const Except_T Fmt_Overflow;
```

```
⟨exported functions 216⟩

#undef T
#endif
```

从技术上讲，Fmt不是一个抽象数据类型，但是它的确导出一种类型Fmt_T，此类型定义了与每种格式化代码相关联的格式转化函数的类型，下面将详细介绍。

14.1.1 格式化函数

下面是两种基本的格式化函数：

```
⟨exported functions 216⟩≡
  extern void Fmt_fmt (int put(int c, void *cl), void *cl,
      const char *fmt, ...);
  extern void Fmt_vfmt(int put(int c, void *cl), void *cl,
      const char *fmt, va_list ap);
```

216

Fmt_fmt根据第三个参数fmt给定的格式字符串来格式化它的第四个及以后的参数，并调用put（c，cl）来给出每个已经格式化的字符c；c被当作无符号字符，因此传递给put的值始终为正。Fmt_vfmt根据格式化字符串来格式化ap指向的参数，该格式化字符串由fmt给定，做法同下面所讲的Fmt_fmt类似。

参数cl可以指向客户调用程序提供的数据，它仅仅不作解释地被传递到客户调用程序的put函数中。put函数返回一个整型值，通常是它的参数。Fmt函数不用这种方法，但是该设计允许在某些机器上当FILE* 被当作cl来传递时，把标准I/O函数fputc当作put函数来用。例如，

```
Fmt_fmt((int (*)(int, void *))fputc, stdout,
    "The %s interface has %d functions\n", name, count)
```

当name是Array、count为8时，在标准输出上输出

```
The Array interface has 8 functions
```

类型转换是必需的，因为fputc有int(*)(int,FILE*)型，而put有int(*)(int,void*)型。这种用法只有在FILE指针有与一个空指针相同的表示方法时才正确。

图14-1所示的语法表定义了转换说明符的语法。在转换说明符中的字符定义了一条通过这张图的路径，有效的说明符从开始到结束走完路径。一个说明符从a%开始，后面是可选的标志字符，其解释依赖于格式化代码；可选的域为：宽度、周期和精度；并以一个单个字符的格式化代码来结束，如图14-1中的C所表示的那样。有效的标志符是那些出现在Fmt_flags指向的字符串中的字符；它们通常定义了对齐（justification）、补足（padding）和截取（truncation）。如果一个标志字符在一个说明符中出现次数超过255次，则会产生可检查的运行期错误。如果星号出现在宽度域和精度域，则下一个参数被假定为整型并用于宽度域和精度域。因此，一个说明符可以使用任意多个参数，这依赖于是否有星号出现和与格式化代码相关联的特殊的转换函数。如果宽度或者精度定义的值等于INT_MIN——多半为负整数时，则会产生可检查的运行期错误。

217

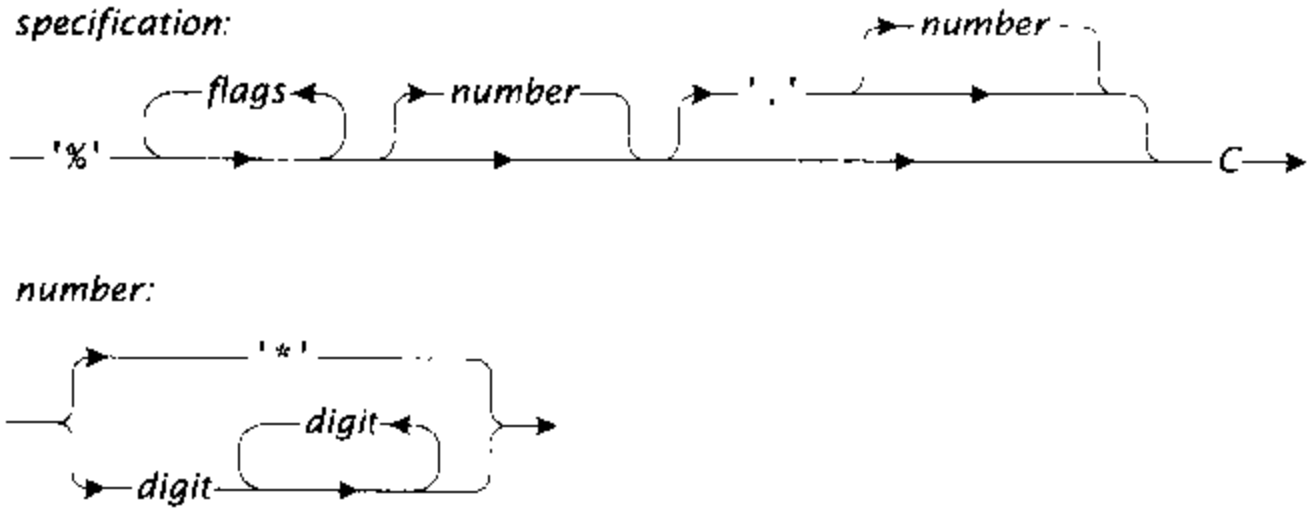


图14-1 转换说明符语法

标志、宽度和精度的详细说明依赖于与转换说明符相关联的转换函数。这些函数调用是那些在调用Fmt_fmt时注册的函数。

缺省的转换说明符和与它们相关联的转换函数都是标准I/O库中的printf函数及相关函数的子集。Fmt_flags的初始值指向字符串“- + 0”，因此，这些字符是有效的标志符，“-”使被转换的字符串在给定的宽度里向左对其（left-justified）。“+”代表一个从“-”或者“+”开始的有符号的转换的结果。空格“ ”表示为正时从空格开始的有符号转换的结果。0表示数字转换，用添零的方法填充域；否则将使用空白。一个负值的宽度带有标志“-”和相应的正值的宽度。精度为负被认为没有给定精度。

在表14-1中列出缺省的转换说明符。这些转换说明符是定义在标准C库里的说明符的子集。

下述函数同C库函数printf、fprintf、sprintf和vsprintf类似：

```
⟨exported functions 216⟩+≡
  extern void Fmt_print (const char *fmt, ...);
  extern void Fmt_fprint(FILE *stream,
      const char *fmt, ...);
extern int Fmt_sfmt   (char *buf, int size,
    const char *fmt, ...);
extern int Fmt_vsfmt(char *buf, int size,
    const char *fmt, va_list ap);
```

218

Fmt_fprint根据fmt给定的格式化字符串来格式化第三个及其后的参数，并把它的已经格式化的结果写到标准输出中。

Fmt_sfmt根据fmt给定的格式化字符串来格式化第四个及其后的参数，并将这些结果作为一个以空字符为结束标志的字符串存储在buf[0..size-1]中。Fmt_vsfmt与此类似，只是它从可变长度参数列表的指针ap中获得参数。这两个函数都返回存储在buf中的字符数，结束符不计算在内。如果Fmt_sfmt和Fmt_vsfmt给出超过size个字符（包含结束符），则它们将引发异常Fmt_Overflow。如果size非正，会产生可检查的运行期错误。

下面两个函数

```
⟨exported functions 216⟩+≡
  extern char *Fmt_string (const char *fmt, ...);
  extern char *Fmt_vstring(const char *fmt, va_list ap);
```

同Fmt_sfmt和Fmt_vsfmt类似，只是它们能分配足够大的字符串来容纳结果并返回这些字符串。客户调用程序负责释放它们。Fmt_string和Fmt_vstring都可能引发Mem_Failed。

将一个空的put、buf或者fmt传递给上面的那些格式化函数都将导致运行时出现检测错误。

14.1.2 转换函数

每个格式字符C都与转换函数相关联。这些联系可以通过调用下面的函数来改变：

```
⟨exported functions 216⟩+≡
219   extern T Fmt_register(int code, T cvt);
```

Fmt_register安装cvt，作为由code提供的格式字符串的转换函数，并返回先前函数的指针。因此客户调用程序可以暂时忽略转换函数，然后再恢复先前的函数。如果code小于0或者大于255则会产生可检查的运行期错误。如果格式字符串使用没有与格式化函数相关联的转换说明符也会产生可检查的运行期错误。

许多转换函数都是使用%d和%s转换说明符的函数的变种。Fmt导出两个实用函数，该函数被它的内部的数字和字符串转换函数使用。

```
⟨exported functions 216⟩+≡
  extern void Fmt_putd(const char *str, int len,
      int put(int c, void *cl), void *cl,
      unsigned char flags[256], int width, int precision);
  extern void Fmt_puts(const char *str, int len,
      int put(int c, void *cl), void *cl,
      unsigned char flags[256], int width, int precision);
```

Fmt_putd假定str [0..len–1] 带有一个无符号数的字符串表示形式，并根据转换来发送字符串，其中转换是由表14-1中描述的%d的flags、width和precision来定义的。类似地，Fmt_puts也根据表14-1中描述的%s的flags、width和precision定义转换来发送字符串。将一个空的str、一个负的len、一个空的flags或者一个空的put传递给Fmt_putd或者Fmt_puts都会产生可检查的运行期错误。

表14-1 缺省的转换说明符

描述符 参数类型	描　述
c int型	该参数被当作无符号字符来说明和发送
d int型	该参数被转换成有符号十进制形式。如果给定了精度，则定义数字的最小值；如果必要话，前面要加零。缺省的精度值是“-” 如果“-”和“0”标志都出现，或者给定了精度，则忽略“0”标志。如果“+”和空格标志同时出现，则忽略空格标志 如果参量和精度均为零，则在转换结果中没有字符。
o u x unsigned型	参数被转换成无符号型的8进制（o）、十进制（u）或者16进制（x）的形式。对于16进制，超过9的数将用abcdef六个字母来表示。标志和精度的说明同d中标志和精度的说明类似。
f double型	该参数被转换成十进制x.y的表示形式 精度给出了小数点右面的数值的位数，缺省为6。如果明确说明精度为零，则小数点被忽略。如果带小数点，则x至少有一位数。精度超过99会产生可检查的运行期错误。标志和精度的说明同d中标志和精度的说明类似。

（续）

描述符 参数类型	描　述
c double型	该参数被转化成十进制*x*.ye ±*p*的表示形式。其中*x*总是一位数而*p*总是两位数。标志和精度的说明同d中标志和精度的说明类似。
g double型	该参数根据它的值被转换成f型或者e型的表示方式。精度的缺省值是　。如果p小于-4或者大于等于精度，则结果写成*x*.ye ±*p*的表示形式；否则写成*x*.*y*的形式。*y*后面没有零，如果*y*为零，则小数点被省略。精度超过99会产生可检查的运行期错误。
p void *	该参数将其值由十进制转换成16进制。标志和精度的说明同d中标志和精度的说明类似。
s char *	来自该参数的连续字符将被发送，直到遇到一个空字符或者所有给定精度的字符均被发送完为止，此时除"-"外，其他的标志均被忽略。

Fmt_putd和Fmt_puts本身不是转换函数，但是它们可以被转换函数调用。当写基于客户调用程序的转换函数时，它们是最有用的。下面将进行说明。

Fmt_T定义了转换函数的签名（signature）——即它的参数的类型和它的返回类型。转换函数由7个参数来调用。前两个是格式化代码和一个指向可变长度参数列表指针的指针，其中参数列表指针用来访问要被格式化的数据。第三个和第四个参数是客户调用程序的数据函数和相关的数据。后三个参数是标志、域的宽度和精度。标志由含256个元素的字符数组来给定，其中第i个元素等于标志字符i在转换说明符里出现的次数。当width和precision没有明确给定时，等于INT_MIN。

转换函数必须使用如下表达方式

```
va_arg(*app, type)
```

来获得参数，其中这些参数将要根据与转换函数相关联的代码来格式化。type是所期望的参数类型。该表达获得参数的值，然后增加*app以便指向下一个参数。如果转换函数错误地增加了*app，则会产生可检查的运行期错误。

220 ~ 221

Fmt对代码%s的私有转换函数说明了怎样写转换函数和怎样使用Fmt_puts。说明符%s类似于printf的%s：它的函数发送字符串中的字符，直到遇到空字符或者将给定精度的所有字符都发送完为止。"-"标志或者负的宽度说明要向左对齐（left-justification）。转换函数使用va_arg从可变长度参数列表中取出参数，并调用Fmt_puts：

```
⟨conversion functions 222⟩≡
  static void cvt_s(int code, va_list *app,
      int put(int c, void *cl), void *cl,
      unsigned char flags[], int width, int precision) {
      char *str = va_arg(*app, char *);
      assert(str);
      Fmt_puts(str, strlen(str), put, cl, flags,
          width, precision);
  }
```

Fmt_puts解释flags、width和precision，并相应的发出字符串：

```
⟨functions 222⟩≡
  void Fmt_puts(const char *str, int len,
       int put(int c, void *cl), void *cl,
       unsigned char flags[], int width, int precision) {

       assert(str);
       assert(len >= 0);
       assert(flags);
       ⟨normalize width and flags 223⟩
       if (precision >= 0 && precision < len)
            len = precision;
       if (!flags['-'])
           pad(width - len, ' ');
       ⟨emit str[0..len-1] 223⟩
       if ( flags['-'])
           pad(width - len, ' ');
  }
```

222

```
⟨emit str[0..len-1] 223⟩≡
  {
       int i;
       for (i = 0; i < len; i++)
           put((unsigned char)*str++, cl);
  }
```

对无符号型字符进行类型转换将确保传递给put的值总是小的、正的整数，就像在Fmt的定义里规定的那样。

如果忽略宽度和精度，则width和precision等于INT_MIN。该接口为基于客户调用程序的转换函数使用所有的宽度（被明确说明的或被省略的）、精度和重复标志的组合提供所需的灵活性。但是缺省的转换不需要这样，它们把所忽略的宽度当作0宽度，把负宽度当作带有“-”标志的相应的正宽度，把负的精度忽略，把重复出现的标志看作只出现一次等。如果有明确的精度，则忽略0标志，并且，如上所示，至多有precision个字符从str中发出。

```
⟨normalize width and flags 223⟩≡
  ⟨normalize width 223⟩
  ⟨normalize flags 224⟩

⟨normalize width 223⟩≡
  if (width == INT_MIN)
       width = 0;
  if (width < 0) {
       flags['-'] = 1;
       width = -width;
  }
```

223

```
⟨normalize flags 224⟩≡
  if (precision >= 0)
       flags['0'] = 0;
```

正如调用pad所显示的，必须发送width-len个空格以正确地对齐输出：

```
⟨macros 224⟩≡
  #define pad(n,c) do { int nn = (n); \
      while (nn-- > 0) \
          put((c), cl); } while (0)
```

pad是宏定义，因为它要访问put和cl。

下一节将描述其他缺省的转换函数的实现。

14.2 实现

Fmt的实现包括在接口中定义的函数、与默认转换说明符相关联的转换函数和将转换说明符映射到转换函数的表。

```
⟨fmt.c⟩≡
  #include <stdarg.h>
  #include <stdlib.h>
  #include <stdio.h>
  #include <string.h>
  #include <limits.h>
  #include <float.h>
  #include <ctype.h>
  #include <math.h>
  #include "assert.h"
  #include "except.h"
  #include "fmt.h"
  #include "mem.h"
  #define T Fmt_T

  ⟨types 226⟩
  ⟨macros 224⟩
  ⟨conversion functions 222⟩
  ⟨data 225⟩
  ⟨static functions 225⟩
  ⟨functions 222⟩

⟨data 225⟩≡
  const Except_T Fmt_Overflow = { "Formatting Overflow" };
```

224

14.2.1 格式化函数

Fmt_vfmt是实现的核心，因为所有其他的接口函数都要调用它来完成格式化。Fmt_fmt是最简单的例子，它初始化参量列表的变量部分的指针va_list，并调用Fmt_vfmt：

```
⟨functions 222⟩+≡
  void Fmt_fmt(int put(int c, void *), void *cl,
      const char *fmt, ...) {
      va_list ap;
```

```
        va_start(ap, fmt);
        Fmt_vfmt(put, cl, fmt, ap);
        va_end(ap);
    }
```

Fmt_print和Fmt_fprint将调用Fmt_vfmt，并把outc当作put函数、把标准输出流或者给定的流当作相关数据：

```
⟨static functions 225⟩≡
  static int outc(int c, void *cl) {
      FILE *f = cl;

      return putc(c, f);
  }

⟨functions 222⟩+≡
  void Fmt_print(const char *fmt, ...) {
      va_list ap;
      va_start(ap, fmt);
      Fmt_vfmt(outc, stdout, fmt, ap);
      va_end(ap);
  }

  void Fmt_fprint(FILE *stream, const char *fmt, ...) {
      va_list ap;
      va_start(ap, fmt);
      Fmt_vfmt(outc, stream, fmt, ap);
      va_end(ap);
  }
```

225

Fmt_sfmt调用Fmt_vsfmt：

```
⟨functions 222⟩+≡
  int Fmt_sfmt(char *buf, int size, const char *fmt, ...) {
      va_list ap;
      int len;

      va_start(ap, fmt);
      len = Fmt_vsfmt(buf, size, fmt, ap);
      va_end(ap);
      return len;
  }
```

Fmt_vsfmt用put函数和一个结构指针来调用Fmt_vsfmt，其中结构指针跟踪被格式化进buf的字符串和此字符串所带有的字符的数目：

```
⟨types 226⟩≡
  struct buf {
      char *buf;
      char *bp;
      int size;
  };
```

buf和size是Fmt_vsfmt有类似命名的参数的备份，bp指向buf中下一个被格式化的字符要存储的位置。Fmt_vsfmt初始化该结构的一个局部实例，并将它的一个指针指向Fmt_vfmt：

226

```
⟨functions 222⟩+≡
  int Fmt_vsfmt(char *buf, int size, const char *fmt,
      va_list ap) {
      struct buf cl;

      assert(buf);
      assert(size > 0);
      assert(fmt);
      cl.buf = cl.bp = buf;
      cl.size = size;
      Fmt_vfmt(insert, &cl, fmt, ap);
      insert(0, &cl);
      return cl.bp - cl.buf - 1;
  }
```

上面对Fmt_vfmt的调用使用了私有函数insert和每个要发送的字符，还有Fmt_vsfmt的局部buf结构指针。insert用来检测是否有容纳字符的空间，并把它存储在由bp域给定的位置，然后再增加bp域的大小：

```
⟨static functions 225⟩+≡
  static int insert(int c, void *cl) {
      struct buf *p = cl;

      if (p->bp >= p->buf + p->size)
          RAISE(Fmt_Overflow);
      *p->bp++ = c;
      return c;
  }
```

Fmt_string和Fmt_vstring有类似的实现方式，只是它们使用不同的put函数。Fmt_string要调用Fmt_vstring函数：

```
⟨functions 222⟩+≡
  char *Fmt_string(const char *fmt, ...) {
      char *str;
      va_list ap;

      assert(fmt);
      va_start(ap, fmt);
      str = Fmt_vstring(fmt, ap);
    va_end(ap);
      return str;
  }
```

227

Fmt_vstring将一个buf结构初始化为一个能容纳256个字符的字符串中，并将该结构的指针传递给Fmt_vfmt：

```
⟨functions 222⟩+≡
  char *Fmt_vstring(const char *fmt, va_list ap) {
      struct buf cl;

      assert(fmt);
      cl.size = 256;
      cl.buf = cl.bp = ALLOC(cl.size);
      Fmt_vfmt(append, &cl, fmt, ap);
      append(0, &cl);
      return RESIZE(cl.buf, cl.bp - cl.buf);
  }
```

append同Fmt_vsfmt的put函数类似，只是它在必要时，能将空闲的字符串的长度加倍，以便能容纳更多格式化的字符。

```
⟨static functions 225⟩+≡
  static int append(int c, void *cl) {
      struct buf *p = cl;

      if (p->bp >= p->buf + p->size) {
          RESIZE(p->buf, 2*p->size);
          p->bp = p->buf + p->size;
          p->size *= 2;
      }
      *p->bp++ = c;
      return c;
  }
```

当完成Fmt_vstring后，由buf域指向的字符串可能太长了，这就是为什么Fmt_vstring要调用RESIZE来释放超出的字符的原因。

该轮到Fmt_vfmt了。它解释了每个格式化字符串，并为每个格式化说明符调用适当的函数；而对格式化字符串中的其他字符，将调用put函数：

228

```
⟨functions 222⟩+≡
  void Fmt_vfmt(int put(int c, void *cl), void *cl,
      const char *fmt, va_list ap) {
      assert(put);
      assert(fmt);
      while (*fmt)
          if (*fmt != '%' || *++fmt == '%')
              put((unsigned char)*fmt++, cl);
          else
              ⟨format an argument 229⟩
  }
```

*<format an argument 229>*中的大部分工作是使用flags、width和precision，并处理转换说明符没有相应的转换函数的可能性。在下面的代码中，将width赋给宽度域，而将precision赋给精度域。

```
⟨format an argument 229⟩≡
  {
```

```
        unsigned char c, flags[256];
        int width = INT_MIN, precision = INT_MIN;
        memset(flags, '\0', sizeof flags);
        ⟨get optional flags 230⟩
        ⟨get optional field width 231⟩
        ⟨get optional precision 232⟩
        c = *fmt++;
        assert(cvt[c]);
        (*cvt[c])(c, &ap, put, cl, flags, width, precision);
    }
```

cvt是一个指向转换函数的指针数组，它由一个格式化字符来索引。在上述代码中，将c声明为一个无符号字符，以确保*fmt被解释成一个0～255之间的整数。

假定按ASCII码排序，则cvt被初始化成带有缺省转换说明符的转换函数：

```
⟨data 225⟩+≡
  static T cvt[256] = {
   /*   0-  7 */ 0,     0, 0,     0,     0,     0,     0,     0,
   /*   8- 15 */ 0,     0, 0,     0,     0,     0,     0,     0,
   /*  16- 23 */ 0,     0, 0,     0,     0,     0,     0,     0,
   /*  24- 31 */ 0,     0, 0,     0,     0,     0,     0,     0,
   /*  32- 39 */ 0,     0, 0,     0,     0,     0,     0,     0,
   /*  40- 47 */ 0,     0, 0,     0,     0,     0,     0,     0,
   /*  48- 55 */ 0,     0, 0,     0,     0,     0,     0,     0,
   /*  56- 63 */ 0,     0, 0,     0,     0,     0,     0,     0,
   /*  64- 71 */ 0,     0, 0,     0,     0,     0,     0,     0,
   /*  72- 79 */ 0,     0, 0,     0,     0,     0,     0,     0,
   /*  80- 87 */ 0,     0, 0,     0,     0,     0,     0,     0,
   /*  88- 95 */ 0,     0, 0,     0,     0,     0,     0,     0,
   /*  96-103 */ 0,     0, 0, cvt_c, cvt_d, cvt_f, cvt_f, cvt_f,
   /* 104-111 */ 0,     0, 0,     0,     0,     0,     0, cvt_o,
   /* 112-119 */ cvt_p, 0, 0, cvt_s,     0, cvt_u,     0,     0,
   /* 120-127 */ cvt_x, 0, 0,     0,     0,     0,     0,     0
  };
```

Fmt_register通过将一个指向它的指针存储到cvt适当的元素中的方法来安装一个新的转换函数。它返回那个元素原先的值：

```
⟨functions 222⟩+≡
  T Fmt_register(int code, T newcvt) {
      T old;

      assert(0 < code
          && code < (int)(sizeof (cvt)/sizeof (cvt[0])));
      old = cvt[code];
      cvt[code] = newcvt;
      return old;
  }
```

扫描转换说明符的代码块遵循图14-1中的语法，在运行时增加fmt。第一个说明符使用flags：

```
⟨data 225⟩+≡
  char *Fmt_flags = "-+ 0";
```

```
⟨get optional flags 230⟩≡
230 if (Fmt_flags) {
      unsigned char c = *fmt;
      for ( ; c && strchr(Fmt_flags, c); c = *++fmt) {
          assert(flags[c] < 255);
          flags[c]++;
      }
  }
```

第二个说明符使用width：

```
⟨get optional field width 231⟩≡
  if (*fmt == '*' || isdigit(*fmt)) {
      int n;
      ⟨n ← next argument or scan digits 231⟩
      width = n;
  }
```

宽度和精度中可以出现星号，在这种情况下，下一个整型参数将提供它们的值。

```
⟨n ← next argument or scan digits 231⟩≡
  if (*fmt == '*') {
      n = va_arg(ap, int);
      assert(n != INT_MIN);
      fmt++;
  } else
      for (n = 0; isdigit(*fmt); fmt++) {
          int d = *fmt - '0';
          assert(n <= (INT_MAX - d)/10);
          n = 10*n + d;
      }
```

正如这段代码显示的那样，当一个参数定义了宽度和精度，就不能定义INT_MIN（缺省值）。如果要明确给定宽度和精度，则它们不能超过INT_MAX，即约束条件是10 · n+d ≤ INT_MAX——亦即10 · n+d不能溢出。该测试不能造成溢出，所以在上述断言代码中又重新声明了约束条件。

231 下面的循环声明了一个渐进的可选的精度：

```
⟨get optional precision 232⟩≡
  if (*fmt == '.' && (*++fmt == '*' || isdigit(*fmt))) {
      int n;
      ⟨n ← next argument or scan digits 231⟩
      precision = n;
  }
```

注意：如果“.”后面不带星号，也不使用数字，被解释为精度忽略。

14.2.2 转换函数

%s的转换函数cvt_s在14.1.2节有所描述。cvt_d是%d的转换函数，它是典型的格式化数字的函数。它获得整型参数，然后把它转化成一个无符号整型，并在局部缓冲区中产生一个字符串，最高有效数字位优先，然后再调用Fmt_putd输出字符串。

```
⟨conversion functions 222⟩+≡
  static void cvt_d(int code, va_list *app,
      int put(int c, void *cl), void *cl,
      unsigned char flags[], int width, int precision) {
      int val = va_arg(*app, int);
      unsigned m;
      ⟨declare buf and p, initialize p 233⟩

      if (val == INT_MIN)
          m = INT_MAX + 1U;
      else if (val < 0)
          m = -val;
      else
          m = val;
      do
          *--p = m%10 + '0';
      while ((m /= 10) > 0);
      if (val < 0)
          *--p = '-';
      Fmt_putd(p, (buf + sizeof buf) - p, put, cl, flags,
          width, precision);
  }
```

232

```
⟨declare buf and p, initialize p 233⟩≡
  char buf[43];
  char *p = buf + sizeof buf;
```

cvt_d完成无符号的算术转换，原因与jAtom_int相同。请参见3.2节，那里也解释了为什么buf有43个字符。

```
⟨functions 222⟩+≡
  void Fmt_putd(const char *str, int len,
      int put(int c, void *cl), void *cl,
      unsigned char flags[], int width, int precision) {
      int sign;

      assert(str);
      assert(len >= 0);
      assert(flags);
      ⟨normalize width and flags 223⟩
      ⟨compute the sign 233⟩
      { ⟨emit str justified in width 234⟩ }
  }
```

Fmt_putd必须发送由flags、width和precision定义的str中的字符串。如果给定精度，则它定义了必须显示的数字的最小个数。有许多数字需要输出，其中可能需要在前面补零。Fmt_putd先确定是否需要一个标记或者在前面补零，然后再给那个字符设置sign：

```
⟨compute the sign 233⟩≡
  if (len > 0 && (*str == '-' || *str == '+')) {
      sign = *str++;
      len--;
  } else if (flags['+'])
      sign = '+';
  else if (flags[' '])
      sign = ' ';
  else
      sign = 0;
```

233 在<*compute the sign 233*>中if语句的顺序实现了“+”标记优先于空格标记的规则。转换结果的长度n取决于精度、被转换的值和标记：

```
⟨emit str justified in width 234⟩≡
  int n;
  if (precision < 0)
      precision = 1;
  if (len < precision)
      n = precision;
  else if (precision == 0 && len == 1 && str[0] == '0')
      n = 0;
  else
      n = len;
  if (sign)
      n++;
```

n是要传送的字符数，该段代码处理零值在零精度情况下的转换，在这种情况下，不输出转换后的结果中的字符。

如果输出是左边对齐的，则Fmt_putd可以发送标记；否则，如果输出是左面补零右边对齐，则Fmt_putd发送标记和实际的值；如果输出是左面补空格右面对齐，则Fmt_putd发送实际的值和标志。

```
⟨emit str justified in width 234⟩+≡
  if (flags['-']) {
      ⟨emit the sign 234⟩
  } else if (flags['0']) {
      ⟨emit the sign 234⟩
      pad(width - n, '0');
  } else {
      pad(width - n, ' ');
      ⟨emit the sign 234⟩
  }

⟨emit the sign 234⟩≡
```

```
    if (sign)
        put(sign, cl);
```

234

Fmt_putd可以完整地发送转换后的结果。如果规定了精度，则包括前面的零，如果输出是左对齐的，则只包括实际的值。

```
⟨emit str justified in width 234⟩+≡
  pad(precision - len, '0');
  ⟨emit str[0..len-1] 223⟩
  if (flags['-'])
      pad(width - n, ' ');
```

cvt_u比cvt_d更简单，但是它使用Fmt_putd的所有机制来发送转换结果。它为下一个无符号整型发送十进制表示形式：

```
⟨conversion functions 222⟩+≡
  static void cvt_u(int code, va_list *app,
      int put(int c, void *cl), void *cl,
      unsigned char flags[], int width, int precision) {
      unsigned m = va_arg(*app, unsigned);
      ⟨declare buf and p, initialize p 233⟩

      do
          *--p = m%10 + '0';
      while ((m /= 10) > 0);
      Fmt_putd(p, (buf + sizeof buf) - p, put, cl, flags,
          width, precision);
  }
```

八进制和十六进制的转换同无符号十进制转换类似，只是输出底数不同，它简化了转换本身。

```
⟨conversion functions 222⟩+≡
  static void cvt_o(int code, va_list *app,
      int put(int c, void *cl), void *cl,
      unsigned char flags[], int width, int precision) {
      unsigned m = va_arg(*app, unsigned);
      ⟨declare buf and p, initialize p 233⟩

      do
          *--p = (m&0x7) + '0';
      while ((m >>= 3) != 0);
      Fmt_putd(p, (buf + sizeof buf) - p, put, cl, flags,
          width, precision);
  }

static void cvt_x(int code, va_list *app,
      int put(int c, void *cl), void *cl,
      unsigned char flags[], int width, int precision) {
      unsigned m = va_arg(*app, unsigned);
      ⟨declare buf and p, initialize p 233⟩
```

235

```
        ⟨emit m in hexadecimal 236⟩
    }

⟨emit m in hexadecimal 236⟩≡
    do
        *--p = "0123456789abcdef"[m&0xf];
    while ((m >>= 4) != 0);
    Fmt_putd(p, (buf + sizeof buf) - p, put, cl, flags,
        width, precision);
```

cvt_p以十六进制发送指针。这里忽略精度和除“-”之外的所有的标志。参数是指针，并且把它转换成一个无符号长整型数，然后在该指针中进行转换，因为无符号型所占空间不会大于指针。

```
⟨conversion functions 222⟩+≡
    static void cvt_p(int code, va_list *app,
        int put(int c, void *cl), void *cl,
        unsigned char flags[], int width, int precision) {
        unsigned long m = (unsigned long)va_arg(*app, void*);
        ⟨declare buf and p, initialize p 233⟩

        precision = INT_MIN;
        ⟨emit m in hexadecimal 236⟩
    }
```

cvt_c是与%c相关联的转换函数，它格式化单个的字符，width个字符向左或者向右对齐。它忽略了精度和其他标志。

236

```
⟨conversion functions 222⟩+≡
    static void cvt_c(int code, va_list *app,
        int put(int c, void *cl), void *cl,
        unsigned char flags[], int width, int precision) {
        ⟨normalize width 223⟩
        if (!flags['-'])
            pad(width - 1, ' ');
        put((unsigned char)va_arg(*app, int), cl);
        if ( flags['-'])
            pad(width - 1, ' ');
    }
```

cvt_c获得一个整数而非字符，因为在参数列表的变量部分传递的字符参数阻碍了缺省参数的作用，因此要转化成整数再传递。cvt_c将整型结果转换成一个无符号字符以使有符号的、无符号的和无格式的字符都能以相同的方式发送。

要以独立于机器的方式将一个浮点数转换成精确的十进制数是很困难的。基于硬件的算法更快也更精确，因此，与e、f和g等转换说明符相关联的转换函数用下面的代码将val的绝对值转化到buf中，然后再发送buf。

```
⟨format a double argument into buf 237⟩≡
    {
```

```
        static char fmt[] = "%.dd?";
        assert(precision <= 99);
        fmt[4] = code;
        fmt[3] =      precision%10 + '0';
        fmt[2] = (precision/10)%10 + '0';
        sprintf(buf, fmt, va_arg(*app, double));
    }
```

浮点型转换说明符之间的区别在于它们怎样格式化浮点型值的各个部分。最长的输出来自说明符%.99f，可能需要DBL_MAX_10_EXP+1+1+99+1个字符。DBL_MAX_10_EXP和DBL_MAX是在标准头文件float.h中定义的。DBL_MAX是能用双精度表示的最大的值，DBL_MAX_10_EXP是$\log_{10}$DBL_MAX，即这是能够由一个双精度数表示的最大的十进制位数。对于在IEEE 754格式中的64比特双精度数，DBL_MAX为1.797693×10^{308}，DBL_MAX_10_EXP为308。fmt [2] 和fmt [3] 的分配假定按照ASCII排序序列进行。 237

因此，如果DBL_MAX被转化成转换说明符%.99f说明的形式，则结果包括小数点之前的DBL_MAX_10_EXP+1位数、小数点、小数点后99位数和一个空的结束符。将精度限制在99即是限制了容纳转换结果所需的缓冲区的大小，并使之在编译时能被识别。来自其他转换说明符%e和%g的转换结果，比%f的结果所带的字符要少。cvt_f处理所有这三部分的代码如下：

⟨*conversion functions* 222⟩+≡

```
    static void cvt_f(int code, va_list *app,
        int put(int c, void *cl), void *cl,
        unsigned char flags[], int width, int precision) {
        char buf[DBL_MAX_10_EXP+1+1+99+1];

        if (precision < 0)
            precision = 6;
        if (code == 'g' && precision == 0)
            precision = 1;
        ⟨format a double argument into buf 237⟩
        Fmt_putd(buf, strlen(buf), put, cl, flags,
            width, precision);
    }
```

参考书目浅析

Plauger（1992）描述了C库函数printf输出函数家族的实现，包括将字符串转换成浮点型值或将浮点型值转换成字符串型值的底层代码。它的代码也显示了怎样实现其他的printf风格的格式化标志和代码。

Hennessy和Patterson（1994）中第4.8节描述了IEEE 754浮点标准和浮点型加法和乘法的实现。Goldberg（1991）研究了程序员最关心的浮点型算术的属性。

浮点型转换已经被实现了很多次，但是由于它们很容易变得不精确或者太慢，从而需要

不停地修补。这些转换的最终结果是如果给定一个浮点型值x，则输出转换产生一个字符串，
238 几输入转换从该字符串中重新产生一个y与x按位一致。Clinger（1990）描述了怎样精确地实现输入转换，还描述了对于一些x的任意精度的转换算法。Steele和White（1990）描述了怎样实现一个精确的输出转换。

练习

14.1 Fmt_vstring使用RESIZE来释放返回的字符串中不使用的部分。设计一种方法，使得可以只有在值得释放的时候才完成释放，即当所释放的空间值得为释放它所付出的代价时才完成释放。

14.2 使用Steele和White（1990）中描述的算法，实现e、f和g的转换。

14.3 写一个转换函数，它带有来自下一个整型参数的转换说明符，并将它和字符@相关联。例如：

```
Fmt_string("The offending value is %@\n",
    x.format, x.value);
```

将根据x.format中所带的格式化代码来格式化x.value。

14.4 写一个转换函数，来发送整型序列Bit_T中的元素，序列中是以范围划分的各个数
239 串。例如，1 32-45 68 70-71。

第15章　低级字符串

C本质上不是处理字符串的语言，但是它确实包含处理字符型数组的工具，通常这些字符型数组被称为字符串。一般来说，一个*N*字符的字符串就是一个*N*+1个字符串的数组，其中第*N*+1个字符是空字符，即它的值为零。

C语言本身只有两个用来处理字符串的特性。字符指针可以用来移动字符数组，并且字符串可以用来初始化字符数组。例如，

```
char msg[] = "File not found";
```

是

```
char msg[] = { 'F', 'i', 'l', 'e', ' ', 'n', 'o', 't',
    ' ', 'f', 'o', 'u', 'n', 'd', '\0' };
```

的快捷方式。顺便提一下，字符常量，比如“F”，一般是整数而非字符串，这也是为什么sizeof ‘F’ 等于sizeof（int）的原因。

字符串也可以表示给定字符的初始化数组。例如，

```
char *msg = "File not found";
```

等价于

```
static char t376[] = "File not found";
char *msg = t376;
```


其中t376是由编译器产生的内部名。

在任何可以用只读数组名的地方都可以使用字符串。例如，Fmt的cvt_x在下面的表述中使用字符串：

```
do
    *--p = "0123456789abcdef"[m&0xf];
while ((m >>= 4) != 0);
```

来等价于下面的冗长的表述：

```
{
    static char digits[] = "0123456789abcdef";
    *p++ = digits[m&0xf];
}
```

其中，digits是编译生成的名称。

C库函数包含一组处理以空字符结束的字符串的函数。这些库函数在标准头文件string.h中定义，它们复制、搜索、扫描、比较和传送字符串。strcat是其中的一个典型函数：

```
char *strcat(char *dst, const char *src)
```

该函数将src添加到dst的尾部，即，它从src中复制包括空字符在内的所有字符，并连续添加到dst的末尾，并且覆盖了dst末尾的空字符。

strcat也显示了在头文件string.h中定义的函数的两个缺点。第一，客户调用程序必须给结果分配空间，比如strcat中的dst；第二，也是最重要的缺点，所有的函数都不安全——它们都不能检查结果字符串是否足够大。如果dst不足够大，不能容纳来自src中的多余的字符，则strcat将在未分配的存储区或者其他存储区写入乱码。一些函数，比如strncat，带有一些额外的参数来限制复制到结果中的字符数，这有一定好处，但是仍然会出现分配错误。

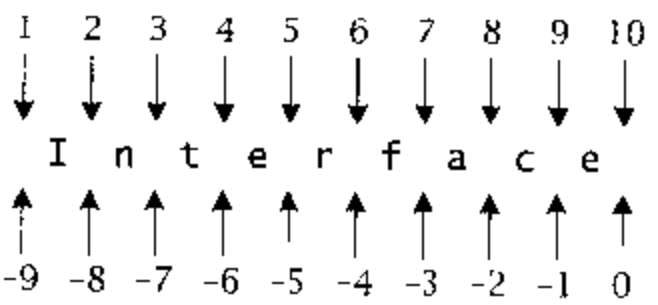
本章描述的Str接口中的函数避免了这些缺点，并为处理字符串参数的子字符串提供了一个方便的方法。这些函数比在string.h中定义的函数更安全，因为大部分Str函数都为它们的结果分配空间，这使得分配更加安全。

242 通常是需要这些分配的，因为string.h函数的客户调用程序在它们的大小依赖于计算输出时，必须分配结果。就string.h函数而言，Str函数的客户调用程序仍然必须释放结果。下一章将要描述的Text接口导出另外一组处理字符串的函数，它们能避免Str函数的一些分配。

15.1 接口

```
⟨str.h⟩≡
  #ifndef STR_INCLUDED
  #define STR_INCLUDED
  #include <stdarg.h>

  ⟨exported functions 244⟩

  #undef T
  #endif
```

Str接口中函数的所有字符串参数都由一个字符数组指针（以空字符结束）和位置（position）给定。类似于Ring的位置，字符串位置识别字符的具体位置，包括最后一个非空字符的位置。正的位置从字符串的左端定位，位置1是左端第一个字符；非正的位置从字符串的右端定位，位置0是右端第一个字符。例如，下面的图表显示了字符串Interface中的位置。



243

字符串s中的位置i和j定义了它们之间的子字符串，由s [i:j] 表示。如果s指向字符串“Interface”，则s [−4:0] 就是子字符串“face”。这些位置可以以任何顺序给定：s [0:−4] 也是指子字符串“face”。子字符串可以为空，s [3:3] 和s[3:−7] 都指的是“Interface”中“n”

和“t”之间的子字符串，即空子字符串。对任意有效的i，s [i:i+1] 总是指i右面的那个字符，除非i已经在最右边。

字符索引是另外一种定义子字符串的方法，并且可能更简单一些，但是这样也有缺点。当用字符索引定义字符串时，顺序是很重要的。例如，字符串“Interface”中的索引是从0到9。如果子字符串由两个索引定义，其中该子字符串从第一个索引之后开始到第二个索引之前结束，则s [1..6] 定义了子字符串“terf”。但是这种方法必须允许空字符串索引，以便用s [4..9] 表示子字符串face，并且它不能定义最前面的空字符串。改变这种方法使得所定义的子字符串在第二个索引后的一个字符结束，这样就不会定义空的字符串了。另外一种方法是使用负索引，但是它比使用位置的方法更麻烦。

使用位置的方法比使用字符索引的方法好，因为它们避免了界限模糊的情况。而且非正的位置可以用来访问一个字符串的尾部，而无需知道它的长度。

Str输出一些函数，这些函数创建并返回以空字符结束的字符串，同时也返回字符串信息和位置。创建字符串的函数是：

```
⟨exported functions 244⟩≡
  extern char *Str_sub(const char *s, int i, int j);
  extern char *Str_dup(const char *s, int i, int j, int n);
  extern char *Str_cat(const char *s1, int i1, int j1,
      const char *s2, int i2, int j2);
  extern char *Str_catv   (const char *s, ...);
  extern char *Str_reverse(const char *s, int i, int j);
  extern char *Str_map    (const char *s, int i, int j,
      const char *from, const char *to);
```

所有这些函数都为它们的结果分配空间，也都能引发异常Mem_Failed。将一个空字符串指针传递给该接口中的任意函数都将会产生可检查的运行期错误，除了下面描述的Str_catv和Str_map两种情况：

244

Str_sub返回s的子字符串s [i:j]。例如下列代码都返回“face”。

```
Str_sub("Interface",  6, 10)
Str_sub("Interface",  6,  0)
Str_sub("Interface", -4, 10)
Str_sub("Interface", -4,  0)
```

这些位置可以按任意顺序给定。将一个用i和j无法确定的s中的子字符串传递给该接口的任意函数都会产生可检查的运行期错误。

Str_dup返回s [i:j] 的n个备份字符串。n为负则会产生可检查的运行期错误。Str_dup通常用来复制字符串，例如，Str_dup（“Interface”，1，0，1）将返回字符串Interface的一个备份。注意使用位置1和0来定义Interface的方法。

Str_cat返回s1 [i1:j1] 和s2 [i2:j2] 相连接后的结果，即将字符串s2 [i2:j2] 中的字符放到字符串s1 [i1:j1] 的字符后面。Str_catv与此类似，它所带的参数每三个一组，每组代表一个字符串和它的两个位置，然后返回这些子字符串连接后的结果。参数列表由空指针作为结束

符。例如，

```
Str_catv("Interface", -4, 0, " plant", 1, 0, NULL)
```

将返回字符串face plant。

Str_reverse将返回一个包含s [i:j] 中的字符的字符串，只不过与在s中显示的顺序相反。

Str_map将返回一个包含s [i:j] 中的字符的字符串，只不过是按照由from和to给定的值的对应关系来显示的。s [i:j] 中显示在from位置的每个字符都映射到to位置的相应字符中。没有在from显示的字符则映射它们本身。例如，

```
Str_map(s, 1, 0, "ABCDEFGHIJKLMNOPQRSTUVWXYZ",
                 "abcdefghijklmnopqrstuvwxyz")
```

将返回s的一个备份，只不过大写的字母被相应的小写字母代替。

如果from和to都为空，将使用最近一次调用Str_map所定义的映射。如果s为空，则忽略i
245 和j，此时from和to只用于建立缺省的映射，同时Str_map将返回空值。

以下情况将会产生可检查的运行期错误：from和to中只有一个指针为空，from和to非空但是定义的字符串长度不等，s、from和to均为空的情况和在第一次调用Str_map时from和to均为空，等等。

Str接口中其他函数将返回字符串信息或者在字符串中的位置，它们都不分配空间。

⟨exported functions 244⟩+≡

```
extern int Str_pos(const char *s, int i);
extern int Str_len(const char *s, int i, int j);
extern int Str_cmp(const char *s1, int i1, int j1,
    const char *s2, int i2, int j2);
```

Str_pos返回对应于s [i:i] 的正的位置。正的位置通过减一总能转换成索引，因此，在需要索引时，常常使用Str_pos。例如，如果s指向字符串“Interface”，则

```
printf("%s\n", &s[Str_pos(s, -4)-1])
```

将输出“face”。

Str_len返回s [i:j] 中的字符数。

Str_cmp返回小于零、等于零和大于零的值，分别对应于s1 [i1:j1] 小于、等于和大于s2 [i2:j2] 的情况。

下面的函数为字符和其他字符串搜索字符串。如果搜索成功，则这些函数将返回反映搜索结果的正的位置，如果搜索失败，它们将返回零。函数名中带有_r的函数将从它们的参数字符串的最右端开始搜索，其他的函数从左端开始搜索。

⟨exported functions 244⟩+≡

```
extern int Str_chr  (const char *s, int i, int j, int c);
extern int Str_rchr (const char *s, int i, int j, int c);
extern int Str_upto (const char *s, int i, int j,
    const char *set);
extern int Str_rupto(const char *s, int i, int j,
```

```
    const char *set);
extern int Str_find (const char *s, int i, int j,
    const char *str);
extern int Str_rfind(const char *s, int i, int j,
    const char *str);
```

246

如果c出现在s [i:j] 中，则Str_chr和Str_rchr分别返回c从左至右第一次或从右至左第一次出现时在s中的位置的前一个位置；如果c不出现在s [i:j] 中，则返回零。

如果set中的任意字符c出现在s [i:j] 中，则Str_upto和Str_rupto分别返回c从左至右第一次或从右至左第一次出现时在s中的位置的前一个位置；如果c不出现在s [i:j] 中，则返回零。在将一个空set传向这两个函数时会产生可检查的运行期错误。

如果str出现在s [i:j] 中，则Str_find和Str_rfind分别返回str从左至右第一次或从右至左第一次出现时在s中的位置的前一个位置；如果str不在s [i:j] 中，则返回零。在将一个空str传向这两个函数时会产生可检查的运行期错误。

下述函数

```
⟨exported functions 244⟩+≡
extern int Str_any   (const char *s, int i,
    const char *set);
extern int Str_many  (const char *s, int i, int j,
    const char *set);
extern int Str_rmany (const char *s, int i, int j,
    const char *set);
extern int Str_match (const char *s, int i, int j,
    const char *str);
extern int Str_rmatch(const char *s, int i, int j,
    const char *str);
```

均是处理字符串的，它们返回匹配子字符串之处之前或之后的正的位置。

如果字符s [i:i+1] 在set中，则Str_any返回该字符在s中的位置的后面相邻的位置；否则返回零。

如果s [i:j] 是以set中的一个或多个连续字符序列开始，则Str_many返回该字符序列的最后一个字符在s中的位置后面相邻的位置；否则，返回零。如果s [i:j] 是以set中的一个或多个连续字符序列结束，则Str_rmany返回该字符序列第一个字符在s中的位置之前相邻的位置；否则，返回零。将空的set传递给Str_any、Str_many和Str_rmany都会产生可检查的运行期错误。

247

如果s [i:j] 是以str开头的，则Str_match返回str在s中的位置后面的位置；否则，返回零。如果s [i:j] 是以str结束的，则Str_rmatch返回str在s中的位置前面的位置；否则，返回零。将空的str传递给Str_match或Str_rmatch都将导致可验查的运行期错误。

Str_rchr、Str_rupto和Str_rfind从它们的参数列表的最右端开始搜索，但是返回它们所搜索到的字符或者字符串的左面的位置。例如，下列代码

```
Str_find ("The rain in Spain", 1, 0, "rain")
Str_rfind("The rain in Spain", 1, 0, "rain")
```

都返回5，因为rain在它们的第一个参数中只出现一次。下列代码

```
Str_find ("The rain in Spain", 1, 0, "in")
Str_rfind("The rain in Spain", 1, 0, "in")
```

则分别返回7和16，因为in出现了三次。

Str_many和Str_match向右操作，并返回它们越过的字符后面的位置。Str_rmany和Str_rmatch则向左操作，它们返回字符前面的位置。例如，

```
Str_sub(name, 1, Str_rmany(name, 1, 0, " \t"))
```

返回name的一个备份，但是即使它后面有空格和制表符，也不会返回这些字符。函数basename显示了这些方法的另外一种典型的应用。basename接受UNIX风格的路径名并只返回文件名而不返回它前面的目录或者后缀，下面的例子说明了这一点

```
basename("/usr/jenny/main.c", 1, 0, ".c")       main
basename("../src/main.c",     1, 0, "")         main.c
basename("main.c",            1, 0, "c")        main.
basename("main.c",            1, 0, ".obj")     main.c
basename("examples/wfmain.c", 1, 0, "main.c")   wf
```

248 basename使用Str_rchr寻找最右端的斜线，用Str_rmatch来分开后缀。

```
char *basename(char *path, int i, int j, char *suffix) {
    i = Str_rchr(path, i, j, '/');
    j = Str_rmatch(path, i + 1, 0, suffix);
    return Str_dup(path, i + 1, j, 1);
}
```

由Str_rchr返回并赋给i的值，是最后一个斜线前面的位置，如果有则为1，没有则为0。无论是哪种情况，文件名都从位置i+1开始。Str_match检查文件名并返回后缀之前或者文件名之后的位置。还有，无论是哪种情况，j都是文件名之后的位置。Str_dup返回i+1和j之间的path中的子字符串。

函数

```
⟨exported functions 244⟩+≡
  extern void Str_fmt(int code, va_list *app,
      int put(int c, void *cl), void *cl,
      unsigned char flags[], int width, int precision);
```

是一个转换函数，它能和Fmt接口中的格式化函数一起使用来格式化子字符串。它带有五个参数：一个字符串指针和两个位置，并按照Fmt的%s定义的风格来格式化子字符串。如果字符串指针、app或者flag为空，则会出现可检查的错误。

例如，如果由下列函数将Str_fmt和格式化代码S关联在一起，

```
Fmt_register('S', Str_fmt)
```

则

```
Fmt_print("%10S\n", "Interface", -4, 0)
```

将打印“_____face”，其中“_”表示空格。

15.2 例子：打印标识符

在输入中打印关键字和标识符的程序说明了Str_fmt的用处和为字符或者其他字符中检查字符串的函数的用处。 249

```
⟨ids.c⟩≡
  #include <stdlib.h>
  #include <stdio.h>
  #include "fmt.h"
  #include "str.h"

  int main(int argc, char *argv[]) {
      char line[512];
      static char set[] = "0123456789_"
          "abcdefghijklmnopqrstuvwxyz"
          "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

      Fmt_register('S', Str_fmt);
      while (fgets(line, sizeof line, stdin) != NULL) {
          int i = 1, j;
          while ((i = Str_upto(line, i, 0, &set[10])) > 0){
              j = Str_many(line, i, 0, set);
              Fmt_print("%S\n", line, i, j);
              i = j;
          }
      }
      return EXIT_SUCCESS;
  }
```

当内部的while循环为下一个标识符扫描line [i:0] 时，从i=1开始。Str_upto返回line中下一个下划线或者line [i:0] 中的字母的位置，并且将位置值赋给i。Str_many返回数字、下划线和字母后面的位置。这样，i和j都标识下一个识别符，Fmt_print打印该识别符，同时也打印Str_fmt，它与格式化代码S相关联。把j赋给i，再次运行while循环来寻找下一个标识符。当line包含对main的声明时，传递给Fmt_print的i和j的值如下图所示。

```
j      4     9   13    18     24     30
    int main(int argc, char *argv[]) {
i   1     5     10   14     20     26
```

250

在该程序中没有进行分配。在这些应用中使用位置时，常常可以不分配。

15.3 实现

```
⟨str.c⟩≡
  #include <string.h>
  #include <limits.h>
  #include "assert.h"
```

```
#include "fmt.h"
#include "str.h"
#include "mem.h"

⟨macros 251⟩
⟨functions 252⟩
```

实现必须处理位置和索引之间的转换，因为函数要使用索引访问具体的字符。正的位置i右面的字符的索引是i-1，负的位置i右面的字符的索引是i+len，其中len是字符串中的字符总数。宏

```
⟨macros 251⟩≡
  #define idx(i, len) ((i) <= 0 ? (i) + (len) : (i) - 1)
```

将这些定义封装在一起。假如给定了长度为len的字符串中的位置i，则idx（i，len）就是i右面的字符的索引。

Str函数将它们的位置参数转换成索引，然后再使用这些索引访问字符串。宏convert封装了转换的步骤：

```
⟨macros 251⟩+≡
  #define convert(s, i, j) do { int len; \
      assert(s); len = strlen(s); \
      i = idx(i, len); j = idx(j, len); \
      if (i > j) { int t = i; i = j; j = t; } \
      assert(i >= 0 && j <= len); } while (0)
```

251

位置i和j被转换成0到字符串s的长度之间的索引，如有必要，它们可以互相交换，因此i不可能超过j。代码块结束处的断言利用可检查的运行期错误执行来保证i和j所标识的s中的位置是有效的，一旦转换完成，j-i就是被说明的子字符串的长度。

Str_sub显示了convert的典型应用。

```
⟨functions 252⟩≡
  char *Str_sub(const char *s, int i, int j) {
      char *str, *p;

      convert(s, i, j);
      p = str = ALLOC(j - i + 1);
      while (i < j)
          *p++ = s[i++];
      *p = '\0';
      return str;
  }
```

子字符串的末尾位置被转换成该子字符串后面的字符的索引，这个字符有可能是空的结束符。因此j-i就是得到的子字符串的长度，包括空字符在内，共需要j-i+1个字节的存储区。

Str_sub和一些其他的Str函数能用标准C库函数里的字符串程序进行操作，例如 strncpy，请参见练习15.2。

15.3.1 字符串操作

Str_dup为s [i:j] 的n个备份和空的结束符分配空间，然后将s [i:j] 复制n遍，这里假定s [i:j] 非空。

```
⟨functions 252⟩+≡
  char *Str_dup(const char *s, int i, int j, int n) {
      int k;
      char *str, *p;

      assert(n >= 0);
      convert(s, i, j);
      p = str = ALLOC(n*(j - i) + 1);
      if (j - i > 0)
          while (n-- > 0)
              for (k = i; k < j; k++)
                  *p++ = s[k];
      *p = '\0';
      return str;
  }
```

252

Str_reverse类似于Str_sub，只是它以相反的顺序复制字符：

```
⟨functions 252⟩+≡
  char *Str_reverse(const char *s, int i, int j) {
      char *str, *p;

      convert(s, i, j);
      p = str = ALLOC(j - i + 1);
      while (j > i)
          *p++ = s[--j];
      *p = '\0';
      return str;
  }
```

Str_cat只能调用Str_catv，但是已足够保证它自己的特定实现：

```
⟨functions 252⟩+≡
  char *Str_cat(const char *s1, int i1, int j1,
                const char *s2, int i2, int j2) {
      char *str, *p;

      convert(s1, i1, j1);
      convert(s2, i2, j2);
      p = str = ALLOC(j1 - i1 + j2 - i2 + 1);
      while (i1 < j1)
          *p++ = s1[i1++];
      while (i2 < j2)
          *p++ = s2[i2++];
      *p = '\0';
      return str;
  }
```

253

Str_catv有一点复杂，因为它必须两次遍历它的参数的变量个数：

```
⟨functions 252⟩+≡
  char *Str_catv(const char *s, ...) {
      char *str, *p;
      const char *save = s;
      int i, j, len = 0;
      va_list ap;

      va_start(ap, s);
      ⟨len ← the length of the result 254⟩
      va_end(ap);
      p = str = ALLOC(len + 1);
      s = save;
      va_start(ap, s);
      ⟨copy each s[i:j] to p, increment p 255⟩
      va_end(ap);
      *p = '\0';
      return str;
  }
```

第一次遍历计算结果的长度，即计算参数子字符串的长度的总和。当为结果分配完空间后，第二次遍历将每个分组给出的子字符串添加结果中。第一次遍历是通过将位置转换为索引的方法计算每个子字符串的长度的，如此，就能得到长度：

```
⟨len ← the length of the result 254⟩≡
  while (s) {
      i = va_arg(ap, int);
      j = va_arg(ap, int);
      convert(s, i, j);
      len += j - i;
      s = va_arg(ap, const char *);
  }
```

254 第二个遍历几乎与此一致：惟一的不同点是len的任务被复制子字符串的循环代替：

```
⟨copy each s[i:j] to p, increment p 255⟩≡
  while (s) {
      i = va_arg(ap, int);
      j = va_arg(ap, int);
      convert(s, i, j);
      while (i < j)
          *p++ = s[i++];
      s = va_arg(ap, const char *);
  }
```

Str_map建立了一个数组map，其中map [c] 是c的映射，如同由from和to定义的那样。通过将s [i:j] 中的字符当作map的索引，可以把s [i:j] 映射并复制到一个新的字符串中。

```
⟨map s[i:j] into a new string 255⟩≡
  char *str, *p;
```

```
    convert(s, i, j);
    p = str = ALLOC(j - i + 1);
    while (i < j)
        *p++ = map[(unsigned char)s[i++]];
    *p = '\0';
```

强制类型转换符（cast）保证值超过127的字符不会转换成负的索引。

map通过初始化来建立，以使map [c] 等于c，即每个字符都与它本身对应。然后from中的字符再索引map中的元素，其中map中的字符就是相应的to中的元素的值：

```
⟨rebuild map 255⟩≡
  unsigned c;
  for (c = 0; c < sizeof map; c++)
      map[c] = c;
  while (*from && *to)
      map[(unsigned char)*from++] = *to++;
  assert(*from == 0 && *to == 0);
```

上述声明实现了对from和to的长度是否相等的运行期错误检查。

当from和to都非空时，Str_map使用这段代码块，当s非空时，它使用〈*map s[i:j] into a new string 255*〉中的代码。

255

```
⟨functions 252⟩+≡
  char *Str_map(const char *s, int i, int j,
      const char *from, const char *to) {
      static char map[256] = { 0 };
      if (from && to) {
          ⟨rebuild map 255⟩
      } else {
          assert(from == NULL && to == NULL && s);
          assert(map['a']);
      }
      if (s) {
          ⟨map s[i:j] into a new string 255⟩
          return str;
      } else
          return NULL;
  }
```

最初，map中所有的元素都为零。没有办法在to中定义一个空字符，因此对map ['a'] 非零的声明将检查第一次调用Str_map时from 和to是否有空指针，有则会出错。

索引为i的字符左边的正位置是i+1。Str_pos用这个特点来返回到s中任意的位置i相对应的正的位置。它将i转化成索引、对它进行验证并将它转化回正的位置，然后返回。

```
⟨functions 252⟩+≡
  int Str_pos(const char *s, int i) {
      int len;

      assert(s);
```

```
    len = strlen(s);
    i = idx(i, len);
    assert(i >= 0 && i <= len);
    return i + 1;
}
```

256 Str_len通过将i和j转化成索引并返回它们之间的字符数的方法来返回子字符串s [i:j]的长度：

```
⟨functions 252⟩+≡
int Str_len(const char *s, int i, int j) {
    convert(s, i, j);
    return j - i;
}
```

Str_cmp的实现很简单直接，但是又很冗长，因为它含有一些细节定义：

```
⟨functions 252⟩+≡
int Str_cmp(const char *s1, int i1, int j1,
    const char *s2, int i2, int j2) {
    ⟨string compare 257⟩
}
```

Str_cmp先将i1和j1转化成s1中的索引，将i2和j2转化成s2中的索引：

```
⟨string compare 257⟩≡
convert(s1, i1, j1);
convert(s2, i2, j2);
```

然后，再调整s1和s2以分别指向它们的第一个字符。

```
⟨string compare 257⟩+≡
s1 += i1;
s2 += i2;
```

s1[i1:j1]和s2[i2:j2]中较短的那个字符串决定了要比较的字符的数目，这是通过调用strncmp来完成的。

```
⟨string compare 257⟩+≡
if (j1 - i1 < j2 - i2) {
    int cond = strncmp(s1, s2, j1 - i1);
    return cond == 0 ? -1 : cond;
} else if (j1 - i1 > j2 - i2) {
    int cond = strncmp(s1, s2, j2 - i2);
    return cond == 0 ? +1 : cond;
} else
257     return strncmp(s1, s2, j1 - i1);
```

如果s1[i1:j1]比s2[i2:j2]短并且memcmp返回值为零，则s1[i1:j1]是s2[i2:j2]的一个前缀，因此s1[i1:j1]比s2[i2:j2]小。第二个if语句处理相反的情况，else语句则在s1和s2的参数的长度相等的时候使用。

标准规定strncmp（和memcmp）必须将s1和s2中的字符当作无符号型字符，这样有助于在s1或s2中的字符值大于127时得到定义良好的结果。例如，strncmp（“\344”，“\127”，1）

必须返回一个正值，但是strncmp的一些实现没有正确地比较“无格式”字符，其中这些字符可能是无符号的，也可能是有符号的。对于这些实现，strncmp（“\344”，“\127”，1）可以返回一个负值。memcmp的一些实现也可能产生同样的错误。

15.3.2 分析字符串

其余的函数从左向右或者从右向左检查字符串，搜索是否有字符或者其他字符串出现。如果搜索成功，则它们都返回一个正的位置值，否则返回零。Str_chr是其中的典型：

```
⟨functions 252⟩+≡
  int Str_chr(const char *s, int i, int j, int c) {
      convert(s, i, j);
      for ( ; i < j; i++)
          if (s[i] == c)
              return i + 1;
      return 0;
  }
```

Str_rchr与此类似，只是它从s [i:j] 的最右端开始搜索：

```
⟨functions 252⟩+≡
  int Str_rchr(const char *s, int i, int j, int c) {
      convert(s, i, j);
      while (j > i)
          if (s[--j] == c)
              return j + 1;
      return 0;
  }
```

258

如果c出现在s [i:j] 中，则这两个函数都返回与c左相邻的正的位置。

Str_upto和Str_rupto同Str_chr和Str_rchr类似，只是它们寻找一个集合中的任意字符是否在s [i:j] 中出现：

```
⟨functions 252⟩+≡
  int Str_upto(const char *s, int i, int j,
      const char *set) {
      assert(set);
      convert(s, i, j);
      for ( ; i < j; i++)
          if (strchr(set, s[i]))
              return i + 1;
      return 0;
  }

  int Str_rupto(const char *s, int i, int j,
      const char *set) {
      assert(set);
      convert(s, i, j);
      while (j > i)
          if (strchr(set, s[--j]))
```

```
            return j + 1;
        return 0;
    }
```

Str_find搜索是否出现s [i:j] 中的字符串。在实现Str_find时，将长度为0或1的字符串当作特殊情况来处理。

```
⟨functions 252⟩+≡
  int Str_find(const char *s, int i, int j,
      const char *str) {
      int len;

      convert(s, i, j);
      assert(str);
      len = strlen(str);
      if (len == 0)
          return i + 1;
259   else if (len == 1) {
        for ( ; i < j; i++)
            if (s[i] == *str)
                return i + 1;
    } else
        for ( ; i + len <= j; i++)
            if (⟨s[i...] ≡ str[0..len-1] 260⟩)
                return i + 1;
      return 0;
  }
```

如果str没有字符，搜索总能成功。如果str只有一个字符，Str_find等价于Str_chr。通常情况下，Str_find在s [i:j] 中搜索str，但是要注意不能接受超过该子字符串末尾但符合条件的搜索结果：

```
⟨s[i...] ≡ str[0..len-1] 260⟩≡
  (strncmp(&s[i], str, len) == 0)
```

Str_rfind也有相同的三种情况，但是它被用于反向比较字符串。

```
⟨functions 252⟩+≡
  int Str_rfind(const char *s, int i, int j,
      const char *str) {
      int len;

      convert(s, i, j);
      assert(str);
      len = strlen(str);
      if (len == 0)
          return j + 1;
      else if (len == 1) {
          while (j > i)
              if (s[--j] == *str)
                  return j + 1;
```

```
        } else
            for ( ; j - len >= i; j--)
                if (strncmp(&s[j-len], str, len) == 0)
                    return j - len + 1;
        return 0;
    }
```

260

调用Str_rfind时也必须小心，它不能接受超过该子字符串的开头但符合条件的结果。

Str_any和与它一组的函数都不搜索字符或者字符串，而只是扫描这些字符或字符串并检查它们是否出现在所讨论的子字符串的开始或末尾。如果s[i:i+1]是集合set中的字符，则Str_any返回Str_pos(s,i)+1：

```
⟨functions 252⟩+≡
  int Str_any(const char *s, int i, const char *set) {
      int len;

      assert(s);
      assert(set);
      len = strlen(s);
      i = idx(i, len);
      assert(i >= 0 && i <= len);
      if (i < len && strchr(set, s[i]))
          return i + 2;
      return 0;
  }
```

如果测试成功，则i+1加一转化成正的位置，这就是为什么Str_any返回i+2的原因。

Str_many将扫描集合set中的一个或多个出现在s[i:j]开头的字符：

```
⟨functions 252⟩+≡
  int Str_many(const char *s, int i, int j,
      const char *set) {
      assert(set);
      convert(s, i, j);
      if (i < j && strchr(set, s[i])) {
          do
              i++;
          while (i < j && strchr(set, s[i]));
          return i + 1;
      }
      return 0;
  }
```

261

Str_rmany将从右至左扫描集合set中的一个或多个出现在s[i:j]末尾的字符

```
⟨functions 252⟩+≡
  int Str_rmany(const char *s, int i, int j,
      const char *set) {
      assert(set);
      convert(s, i, j);
      if (j > i && strchr(set, s[j-1])) {
```

```
        do
            --j;
        while (j >= i && strchr(set, s[j]));
        return j + 2;
    }
    return 0;
}
```

当do-while循环结束时，j等于i-1或是不在集合set中的字符的索引。对于第一种情况，Set_rmany必须返回i+1；而第二种情况，必须返回字符s [j] 右面的位置。在这两种情况下，j+2都是正确的。

如果str出现在s [i:j] 的开头，则Str_match返回Str_pos(s,i)+strlen（str）。同Str_find类似，长度为0或1的字符串要当作特殊情况处理：

```
⟨functions 252⟩+≡
  int Str_match(const char *s, int i, int j,
      const char *str) {
      int len;

      convert(s, i, j);
      assert(str);
      len = strlen(str);
      if (len == 0)
          return i + 1;
      else if (len == 1) {
          if (i < j && s[i] == *str)
              return i + 2;
      } else if (i + len <= j && ⟨s[i...] ≡ str[0..len-1] 260⟩)
262       return i + len + 1;
      return 0;
  }
```

一般情况下，不考虑超过s [i:j] 末尾的匹配字符串。

Str_rmatch中情况与此类似，这里必须避免超过s [i:j] 开头的匹配字符串，而且长度为0或1的字符串要当作特殊情况处理。

```
⟨functions 252⟩+≡
  int Str_rmatch(const char *s, int i, int j,
      const char *str) {
      int len;

      convert(s, i, j);
      assert(str);
      len = strlen(str);
      if (len == 0)
          return j + 1;
      else if (len == 1) {
          if (j > i && s[j-1] == *str)
              return j;
```

```
    } else if (j - len >= i
    && strncmp(&s[j-len], str, len) == 0)
        return j - len + 1;
    return 0;
}
```

15.3.3 转换函数

最后一个函数是Str_fmt，这是一个定义在Fmt接口中的转换函数。转换函数的调用序列在第14.1.2节有所描述。flags、width和precision参数将确定字符串是怎样被格式化的。

Str_fmt最重要的特征就是它带有三个参数，这三个参数都来自参数列表的变量部分，其中参数列表将被传递给其中一个Fmt函数。这三个参数定义了一个字符串和字符串中的两个位置。这些位置给出了子字符串的长度，并且同flags、width和precision一起决定该子字符串是怎样被发送的。Str_fmt使用Fmt_puts来说明这些值并发送字符串： 263

```
⟨functions 252⟩+≡
  void Str_fmt(int code, va_list *app,
      int put(int c, void *cl), void *cl,
      unsigned char flags[], int width, int precision) {
      char *s;
      int i, j;

      assert(app && flags);
      s = va_arg(*app, char *);
      i = va_arg(*app, int);
      j = va_arg(*app, int);
      convert(s, i, j);
      Fmt_puts(s + i, j - i, put, cl, flags,
          width, precision);
  }
```

参考书目浅析

Plauger（1992）曾经对在string.h中定义的函数给予了简短的批评，并说明了如何去实现它们。Roberts（1995）描述了一个简单的字符串接口，该接口与Str类似并且也是基于string.h的。

Str接口的设计逐字逐句地通过Icon程序语言（Griswold and Griswold 1990）的字符串处理工具得到改进。使用位置而非索引并且使用非正的位置来定义相对于Icon产生的字符串的末尾的位置。

Str的函数被Icon中的同名函数所模仿。Icon函数处理能力更强，因为它们使用Icon的直达目标的评价机制。例如，Icon的find函数可以返回一个字符串在另一个字符串中所有出现的位置。Icon也有字符串扫描功能，再加上直达目标的评价机制，便有了强大的模式匹配能力。

Str_map也可用于实现多种类型的字符串转化。例如，s是一个含有7个字符的字符串，

```
Str_map("abcdefg", 1, 0, "gfedcba", s)
```

264 将返回s的倒序。Griswold（1980）探讨了这种映射的使用。

练习

15.1 扩展ids.c，使得它能识别并忽略C语言的注释、字符串文字和关键字。泛化你扩展的版本，使它能接受命令行参数来定义其他将要被忽略的识别符。

15.2 Str的实现能使用标准C库里的字符串和内存函数，比如使用Strncpy和memcpy，来复制字符串。例如，Str_sub可以写成如下形式：

```
char *Str_sub(const char *s, int i, int j) {
    char *str;

    convert(s, i, j);
    str = strncpy(ALLOC(j - i + 1), s + i, j - i);
    str[j - i] = '\0';
    return str;
}
```

一些C编译器能够识别对string.h中的函数的调用，并且产生内部代码，其中内部代码相对于C循环要快得多。高度优化的汇编语言的实现通常也更快些。在可能的地方使用String.h中的函数来重新实现Str，并在一台专门的机器上用专门的C编译器度量结果，然后再描述每个函数在字符串参数的长度方面的改进。

15.3 设计并实现一个函数，使它能搜索一个由正则表达式（regular expression）（比如那些AWK支持的和在Aho、Kernighan和Weinberger（1988）中描述的表达式）定义的子字符串。该函数需要返回两个值：开始匹配的位置和子字符串的长度。

15.4 Icon有一个字符串扫描扩展机能。它的“?”操作符建立了一个支持字符串和该字符串中的位置的扫描环境。字符串函数，比如find，可以只由一个参数来激活
265 当前扫描环境中操作字符串和位置。学习Icon的字符串扫描工具（在Griswold和Griswold（1990）中讲述），并设计和实现一个提供类似功能的接口。

15.5 String.h定义了函数

```
char *strtok(char *s, const char *set);
```

将s分成由集合set中的字符分离的记号。通过反复调用strtok，将字符串s分成多个记号。s只有在第一次调用strtok时才被传递，并且strtok搜索第一个不在集合set中的字符并用一个空字符覆盖，并返回s。随后的调用（它们都具有strtok（NULL，set）的形式），将使strtok在它离开的位置继续搜索第一个在set中的字符，并用一个空字符覆盖，然后返回指向所代表字符串头部的指针，在不同的调用中，set可以不同。如果搜索失败，strtok返回空值。用提供类似能力的函数来扩展Str接口，

但是不要修改它的参数。你可以改进strtok的设计吗？

15.6 Str函数总是为它们的结果分配空间，其中这些空间在某些应用中可能不是必须的。假定接受了一个可选的目标，并且只有在目标是空指针时，才分配空间。例如，

```
char *Str_dup(char *dst, int size,
    const char *s, int i, int j, int n);
```

在dst非空时将结果存储在dst[0..size-1]并返回dst。否则，它将为结果分配空间，就像当前版本做的那样。设计一个基于这种方法的接口。说清楚size太小时会发生什么事情。将你的设计和Str接口比较，哪一个更简单？哪一个更不容易出错？

15.7 下面是另外一个避免在Str函数中分配空间的建议。假定函数

```
void Str_result(char *dst, int size);
```

266

将dst所谓的结果字符串传递给下一次Str函数调用。如果结果字符串非空，Str函数将它们的结果存储在dst [0..size-1] 中并清除结果字符串指针。如果结果字符串为空，则它们与通常一样，为结果分配空间。对此建议从优劣两方面进行讨论。

267

第16章　高级字符串

上一章讲述的Str接口导出的函数增加了C中处理字符串的能力。按照惯例，字符串就是字符数组，其中数组的最后一个字符为空。虽然这种表示方法在很多应用中已经足够了，但是它有两个严重的缺点。第一，寻找一个字符串的长度需要搜索该字符串的结束标志空字符，因此计算字符串的长度所用的时间与字符串的长度成比例。第二，Str接口中的函数和标准库中的一些函数都假定字符串可以改变，因此无论是它们还是它们的调用者都必须为字符串分配空间，而在那些不修改字符串的应用中，不必为字符串分配空间。

本章要讲述的Text接口所使用的表示方法稍有不同，从而克服了上述两个缺点。计算长度的时间是固定的，因为它们同字符串联系在一起，而且只有在必要的时候才会分配空间。由Text提供的字符串是固定的——即它们不能被随意改变——并且它包含一个内部的空字符。Text为它的字符串表示方法和C形式的字符串之间的转换提供了一些函数，而这些转换就是Text性能改善所付出的代价。

16.1　接口

Text接口用一个两元素的描述符来表示字符串，其中该描述符给出了字符串的长度并指向它的第一个字符：


```
⟨exported types 270⟩≡
  typedef struct T {
      int len;
      const char *str;
  } T;
```

```
⟨text.h⟩≡
  #ifndef TEXT_INCLUDED
  #define TEXT_INCLUDED
  #include <stdarg.h>

  #define T Text_T

  ⟨exported types 270⟩
  ⟨exported data 274⟩
  ⟨exported functions 271⟩

  #undef T
  #endif
```

由str域指向的字符串并不用空字符作结束符。由Text_T s指向的字符串可以包含任意字符，包括空字符。Text显示了描述符的表示方法，以便客户调用程序可以直接访问域。如果给定

一个Text_T s，则s.len给出了字符串的长度，并通过s.str[0..s.len-1]来访问具体的字符。

客户调用程序可以读取Text_T的域以及它所指向的字符串中的字符，但是它们不能改变域及字符，除非通过该接口中的函数，或者通过它们初始化的Text_T中的函数，或者由Text_box返回的函数才能改变。改变一个由Text_T描述的字符串将会产生可检查的运行期错误。将一个带有负的len域或空的str域的Text_T传递给该接口中的任何函数，也会产生可检查的运行期错误。

Text导出各种函数，这些函数通过值来传递并返回描述符，即，函数传递和返回的是描述符本身而非将指针传递给描述符。其后果是所有的Text函数都不分配描述符。

在必要的时候，一些Text函数的确为字符串本身分配空间。该字符串空间完全由Text来
270 处理；除了下面描述的情况外，客户调用程序不能释放字符串。用外部工具释放字符串，比如调用free或者Mem_free，会产生不可检查的运行期错误。

函数

```
⟨exported functions 271⟩≡
  extern T    Text_put(const char *str);
  extern char *Text_get(char *str, int size, T s);
  extern T    Text_box(const char *str, int len);
```

将在描述符和C形式的字符串之间转换。Text_put将以空字符结束的字符串str复制到字符串空间并为新的字符串返回一个描述符。Text_put可以引发异常Mem_Failed。如果str为空，则会产生可检查的运行期错误。

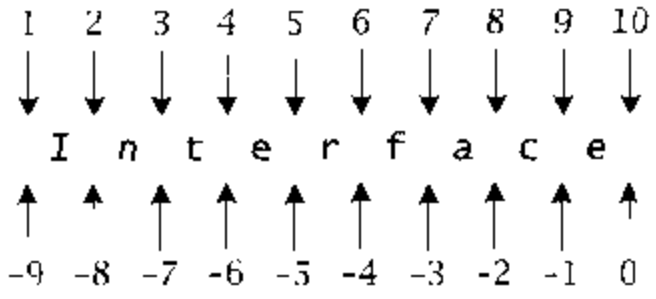
Text_get将s所描述的字符串复制到str[0..size-2]中，添加一个空字符并返回str。如果size小于s.len+1，则会产生可检查的运行期错误。如果str为空，则Text_get将忽略size、调用Mem_alloc来分配s.len+1个字节的空间，然后将s.str复制到里面并返回该分配空间的开头指针。如果str为空，Text_get可以引发异常Mem_Failed。

客户调用程序调用Text_box来为常量字符串或者它们自己分配的字符串建立描述符。它用一个描述符将str和len“装箱”并返回该描述符。例如，

```
static char editmsg[] = "Last edited by: ";
...
Text_T msg = Text_box(editmsg, sizeof (editmsg) - 1);
```

将“Last edited by:”赋给msg。注意Text_box的第二个参数忽略了editmsg末尾的空字符。如果该空字符没有被忽略，则它将被认为是msg所描述的字符串的一部分。str为空或者len为负都将会产生可检查的运行期错误。

许多Text函数都接受字符串位置，这里的位置同在Str中定义的位置类似。位置识别字符间的具体定位，包括第一个字符之前的位置和最后一个字符之后的位置。正的位置从字符串
271 的最左端第一个字符开始，而非正的位置从字符串的最右端第一个字符开始。例如，下图摘自第15章，它显示了在字符串“Interface”中的位置。



函数

```
⟨exported functions 271⟩+≡
  extern T Text_sub(T s, int i, int j);
```

返回字符串s中位于i和j之间的子字符串的描述符，其中位置i和j顺序可以任意。例如，如果

```
Text_T s = Text_put("Interface");
```

则下列表达式

```
Text_sub(s,  6, 10)
Text_sub(s,  0, -4)
Text_sub(s, 10, -4)
Text_sub(s,  6,  0)
```

都返回子字符串“face”的描述符。

既然客户调用程序不改变字符串中的字符，而且字符串也不需要空字符作为结束符，所以Text_sub值返回一个Text_T，其中str域指向s的子字符串的第一个字符，len域是子字符串的长度。s和返回值可以共享一个具体的字符串中的字符，并且Text_sub不分配空间。然而，客户调用程序不能依靠s和共享同一个字符串的返回值，因为Text可能将空的字符串和只显示字符的字符串当作特殊情况来处理。大部分由Text导出的函数都同Str导出的函数类似，但是它们当中有很多不接受位置参数，因为Text_sub能以最小的代价提供相同的功能。 272

函数

```
⟨exported functions 271⟩+≡
  extern int Text_pos(T s, int i);
```

返回任意位置i在s中相应的正的位置。例如，如果s等于“Interface”，则

```
Text_pos(s, -4)
```

将返回6。

如果Text_pos中的i或者Text_sub中的i或j定义了一个在s中不存在的位置，则会产生可检查的运行期错误。

函数

```
⟨exported functions 271⟩+≡
  extern T Text_cat    (T s1, T s2);
  extern T Text_dup    (T s, int n);
  extern T Text_reverse(T s);
```

连接、复制和颠倒字符串。所有这些函数都可以引发异常Mem_Failed。Text_cat返回字符串

s1和s2连接后的字符串的描述符，如果s1或者s2为空字符串，将返回其他参数。而且，Text_cat只有在必要时才复制s1和s2。

Text_dup将返回n个s相连接后的字符串的描述符;如果n为负，则会产生可检查的运行期错误。Text_reverse将以相反的顺序返回字符串s。

```
⟨exported functions 271⟩+≡
  extern T Text_map(T s, const T *from, const T *to);
```

将如下根据from和to指向的字符串来返回s的映射结果。对s中每个出现在from中的字符，to中相应的字符就出现在结果字符串中。如果s中的字符没有出现在from中，则字符本身在输出中不改变显示。例如，

273
```
Text_map(s, &Text_ucase, &Text_lcase)
```

将返回s的一个备份，只是原先s中的大写字母被转换成相应的小写字母。Text_ucase和Text_lcase是由Text输出的预先定义的描述符的例子。完整的列表是：

```
⟨exported data 274⟩≡
  extern const T Text_cset;
  extern const T Text_ascii;
  extern const T Text_ucase;
  extern const T Text_lcase;
  extern const T Text_digits;
  extern const T Text_null;
```

Text_cset是一个含有所有256个8位字符的字符串，Text-ascii含有127个ASCII字符，Text_ucase是字符串“ABCDEFGHIJKLMNOPQRSTUVWXYZ”，Text_lcase是字符串“abcdefghijklmnopqrstuvwxyz”，Text_digits是0123456789，而Text_null则表示空字符串。客户调用程序可以用这些字符串的子字符串来组成新的字符串。

Text_map将记住最近非空的from和to值，并且当from和to均为为空时将使用这些值。from和to中只有一个为空，或者当from和to非空时from->len不等于to->len，则会产生可检查的运行期错误。Text_map可以引发异常Mem_Failed。

字符串通过调用如下函数进行比较：

```
⟨exported functions 271⟩+≡
  extern int Text_cmp(T s1, T s2);
```

函数返回小于零、等于零和大于零的值，分别对应按词典序s1的字符少于s2、等于s2和大于s2三种情况。

Text导出一组字符串分析函数，这些函数与Str导出的那些函数几乎一致。下面将要描述的函数的确接受所检查的字符串中的位置，这些位置通常产生表示分析的状态的编码。在下面的描述中，s [i:j] 表示s中在位置i和j之间的子字符串，而s [i] 表示在位置i右边的字符。

下面的函数将搜索单个字符或者一组字符。在任何情况下，如果i或者j表示一个不存在
274 的位置，则会产生可检查的运行期错误。

```
⟨exported functions 271⟩+≡
  extern int Text_chr  (T s, int i, int j, int c);
  extern int Text_rchr (T s, int i, int j, int c);
  extern int Text_upto (T s, int i, int j, T set);
  extern int Text_rupto(T s, int i, int j, T set);
  extern int Text_any  (T s, int i, T set);
  extern int Text_many (T s, int i, int j, T set);
  extern int Text_rmany(T s, int i, int j, T set);
```

Text_chr返回c在s [i:j] 中最左出现的左邻正位置，Text_rchr返回最右出现的左邻正位置。这两个函数当c不出现在s [i:j] 中时都返回零值。Text upto返回s [i:j] 中最左出现set中任意元素的左邻正位置，Text_rupto返回最右出现左邻正位置。如果set中元素不出现在s [i:j] 中，则这两个函数都返回零值。

如果s [i] 等于c，则Text_any返回Text_pos(s,i)+1，否则，返回零。如果s [i:j] 由集合set中的一个字符开始，则Text_many返回一个正的位置，后面是set中的非空字符的序列，否则，返回零值。如果s [i:j] 由set中的一个字符结束，则Text_rmany返回set中的字符组成的非空序列的前面的位置。否则，返回零值。

还有一些寻找字符串的分析函数。

```
⟨exported functions 271⟩+≡
  extern int Text_find  (T s, int i, int j, T str);
  extern int Text_rfind (T s, int i, int j, T str);
  extern int Text_match (T s, int i, int j, T str);
  extern int Text_rmatch(T s, int i, int j, T str);
```

Text_find返回str在s [i:j] 中出现的最左端的位置之前的一个正位置，而Text_rfind返回str在s [i:j] 中出现的最右端的位置之前的一个正的位置。如果str在s [i:j] 中不出现，则这两个函数都返回零值。

如果s [i:j] 从字符串str开始，则Text_match返回Text_pos(s,i)+str.len，否则返回零值。

函数

275

```
⟨exported functions 271⟩+≡
  extern void Text_fmt(int code, va_list *app,
      int put(int c, void *cl), void *cl,
      unsigned char flags[], int width, int precision);
```

可以在Fmt接口中作为转换函数来使用。它使用Text_T的指针并根据可选的参数flags、width和precision来格式化字符串，其中格式化方式同printf代码%s格式化它的参数的方式相同。使用Text_T的指针是因为在标准C库函数中，在可变长度参数列表的变量部分传递一个小的结构可能不方便。如果Text_T的指针为空，或者app或flags为空，都会产生可检查的运行期错误。

Text给了客户调用程序有限的控制权来控制字符串空间的分配，其中字符串空间用来存储上面所描述的能返回描述符的函数所返回的结果字符串。特别地，下面的函数将存储空间

当作堆栈来处理。

```
⟨exported types 270⟩+≡
  typedef struct Text_save_T *Text_save_T;

⟨exported functions 271⟩+≡
  extern Text_save_T Text_save(void);
  extern void        Text_restore(Text_save_T *save);
```

Text_save返回隐式指针类型Text_save_T的值，其中Text_save_T对字符串空间的顶部（top）进行编码。该值以后可以传递给Text_restore来释放在Text_save_T被创建时所分配的字符串空间部分。如果h是Text_save_T类型的值，则调用Text_restore(h)将使所有在h之后创建的描述符和Text_save_T的值无效。将一个空的Text_save_T传递给Text_restore将会产生可检查的运行期错误。而使用这些值将会产生不可检查的运行期错误。Text_save可以引发异常Mem_Failed。

16.2 实现

Text的实现同Str的实现非常相似，但是Text函数可以对一些重要的特殊情况加以利用，
276 下面将详细介绍。

```
⟨text.c⟩≡
  #include <string.h>
  #include <limits.h>
  #include "assert.h"
  #include "fmt.h"
  #include "text.h"
  #include "mem.h"

  #define T Text_T

  ⟨macros 278⟩
  ⟨types 287⟩
  ⟨data 277⟩
  ⟨static functions 286⟩
  ⟨functions 278⟩
```

常量描述符都指向一个由所有256个字符组成的字符串：

```
⟨data 277⟩≡
  static char cset[] =
      "\000\001\002\003\004\005\006\007\010\011\012\013\014\015\016\017"
      "\020\021\022\023\024\025\026\027\030\031\032\033\034\035\036\037"
      "\040\041\042\043\044\045\046\047\050\051\052\053\054\055\056\057"
      "\060\061\062\063\064\065\066\067\070\071\072\073\074\075\076\077"
      "\100\101\102\103\104\105\106\107\110\111\112\113\114\115\116\117"
      "\120\121\122\123\124\125\126\127\130\131\132\133\134\135\136\137"
      "\140\141\142\143\144\145\146\147\150\151\152\153\154\155\156\157"
      "\160\161\162\163\164\165\166\167\170\171\172\173\174\175\176\177"
      "\200\201\202\203\204\205\206\207\210\211\212\213\214\215\216\217"
```

```
        "\220\221\222\223\224\225\226\227\230\231\232\233\234\235\236\237"
        "\240\241\242\243\244\245\246\247\250\251\252\253\254\255\256\257"
        "\260\261\262\263\264\265\266\267\270\271\272\273\274\275\276\277"
        "\300\301\302\303\304\305\306\307\310\311\312\313\314\315\316\317"
        "\320\321\322\323\324\325\326\327\330\331\332\333\334\335\336\337"
        "\340\341\342\343\344\345\346\347\350\351\352\353\354\355\356\357"
        "\360\361\362\363\364\365\366\367\370\371\372\373\374\375\376\377"
        ;
    const T Text_cset   = { 256, cset };
    const T Text_ascii  = { 127, cset };
    const T Text_ucase  = {  26, cset + 'A' };
    const T Text_lcase  = {  26, cset + 'a' };
    const T Text_digits = {  10, cset + '0' };
    const T Text_null   = {   0, cset };
```

277

Text函数接受指针，但是将它们转换成字符索引以便能访问字符串中的字符。一个正的位置通过减一转换成索引，而一个非负的位置通过加上该字符串的长度而转换成索引：

```
⟨macros 278⟩≡
  #define idx(i, len) ((i) <= 0 ? (i) + (len) : (i) - 1)
```

相反地，一个索引通过加一转换成一个正的位置，如Text_pos的实现所显示的那样，在Text_pos中，先将位置参数转化成索引，然后再将索引转化回一个正的位置。

```
⟨functions 278⟩≡
  int Text_pos(T s, int i) {
      assert(s.len >= 0 && s.str);
      i = idx(i, s.len);
      assert(i >= 0 && i <= s.len);
      return i + 1;
  }
```

Text_pos中的第一个assert函数会产生可检查的运行期错误，要求所有的Text_T s必须带有非负的len域和非空的str域。第二个assert函数声明是位置i——现在是索引——相应于s中的有效的位置的可检查的运行期错误。如果s有N个字符，则有效的索引是从零到$N-1$，但是有效的位置是1到$N+1$，这就是为什么第二个assert函数声明有N个索引的原因。

Text_box和Text_sub都建立和返回一个新的描述符。

```
⟨functions 278⟩+≡
  T Text_box(const char *str, int len) {
      T text;

      assert(str);
      assert(len >= 0);
      text.str = str;
      text.len = len;
      return text;
  }
```

278

Text_sub与此类似，但是它必须将它的位置参数转换成索引以便能计算结果的长度：

```
⟨functions 278⟩+≡
  T Text_sub(T s, int i, int j) {
      T text;

      ⟨convert i and j to indices in 0..s.len 279⟩
      text.len = j - i;
      text.str = s.str + i;
      return text;
  }
```

如上所示，当它们从位置转换成索引后，在i和j之间有j-i个字符。用以转换的代码也交换i和j以便i总能代表最左端的字符的索引。

```
⟨convert i and j to indices in 0..s.len 279⟩≡
  assert(s.len >= 0 && s.str);
  i = idx(i, s.len);
  j = idx(j, s.len);
  if (i > j) { int t = i; i = j; j = t; }
  assert(i >= 0 && j <= s.len);
```

最后一个字符右边的位置被转化成一个不存在的字符的索引，并且断言声明也接受这样的位置。只有在这些索引不用于获取或者存储字符时才不使用<*convert i and j to indices in 0..s.len 279*>。例如，Text_sub只使用它们计算开始的位置和长度。其他的Text函数只有在检测到i和j是有效的索引时才使用i和j的值。

Text_put和Text_get将字符串复制进空间或者从空间复制出字符串。Text执行它自己的分配函数*alloc(int len)来分配len字节的字符串空间。这样做有一些原因，第一，alloc不使
279 用在通用分配程序中使用的存储块头部，以便可以将字符串相连。这是对Text_dup和Text_cat重要的优化。第二，alloc可以避免对齐（alignment），因为字符没有对齐限制。最后，alloc必须同Text_save和Text_restore共同作用。在16.2.2节中将开始一起描述alloc、Text_save和Text_restore。

Text_put是分配字符串空间的少数典型的Text函数。它调用alloc来分配所需的空间、将参数字符串复制到分配的空间中并返回适当的描述符：

```
⟨functions 278⟩+≡
  T Text_put(const char *str) {
      T text;

      assert(str);
      text.len = strlen(str);
      text.str = memcpy(alloc(text.len), str, text.len);
      return text;
  }
```

Text_put调用memcpy而非strcpy，因为它不能向text.str中添加空字符。

Text_get正好相反：它将字符串空间中的字符串复制成C形式的字符串。如果C形式的字符串指针为空，则Text_get调用Mem的通用分配程序来为字符串和空结束符分配空间：

```
⟨functions 278⟩+≡
  char *Text_get(char *str, int size, T s) {
      assert(s.len >= 0 && s.str);
      if (str == NULL)
          str = ALLOC(s.len + 1);
      else
          assert(size >= s.len + 1);
      memcpy(str, s.str, s.len);
      str[s.len] = '\0';
      return str;
  }
```

Text_get调用memcpy而非strncpy，因为它必须复制出现在s中的空字符。 280

16.2.1 字符串操作

Text_dup将Text_T的参数s复制n次。

```
⟨functions 278⟩+≡
  T Text_dup(T s, int n) {
      assert(s.len >= 0 && s.str);
      assert(n >= 0);
      ⟨Text_dup 281⟩
  }
```

有几种重要的特殊情况不需要对s的n个副本进行分配。例如，如果s是空字符串或者n等于0，则Text_dup返回一个空的字符串，如果n为1，则Text_dup就返回s：

```
⟨Text_dup 281⟩≡
  if (n == 0 || s.len == 0)
      return Text_null;
  if (n == 1)
      return s;
```

如果最近已经创建了s，则s.str可能位于字符串空间的末尾，即，s.str+s.len可能等于下一个空闲字节的地址。如果是这样，则只需要n-1个s，因为初始的s可以作为第一个备份。在第16.2.2节中定义的宏isatend（s，n），将检查s.str是否在字符串空间的末尾，还检查是否有能容纳至少n个字符的空间。

```
⟨Text_dup 281⟩+≡
  {
      T text;
      char *p;
      text.len = n*s.len;
      if (isatend(s, text.len - s.len)) {
          text.str = s.str;
          p = alloc(text.len - s.len);
          n--;
      } else
          text.str = p = alloc(text.len);
      for ( ; n-- > 0; p += s.len)
```

281
```
            memcpy(p, s.str, s.len);
        return text;
    }
```

Text_cat返回两个字符串s1和s2相连接后的字符串。

```
⟨functions 278⟩+≡
  T Text_cat(T s1, T s2) {
      assert(s1.len >= 0 && s1.str);
      assert(s2.len >= 0 && s2.str);
      ⟨Text_cat 282⟩
  }
```

对于Text_dup，有一些重要的特殊情况可以避免分配。第一种情况是s1和s2中有一个为空时，Text_cat就可以只返回另外一个字符串：

```
⟨Text_cat 282⟩≡
  if (s1.len == 0)
      return s2;
  if (s2.len == 0)
      return s1;
```

s1和s2可以已经是相连的，这种情况下Text_cat就可以返回这个整体的描述符：

```
⟨Text_cat 282⟩+≡
  if (s1.str + s1.len == s2.str) {
      s1.len += s2.len;
      return s1;
  }
```

如果s1位于字符串空间的末端，则只需要复制s2。否则，两个字符串都要复制：

```
⟨Text_cat 282⟩+≡
  {
      T text;
      text.len = s1.len + s2.len;
      if (isatend(s1, s2.len)) {
          text.str = s1.str;
282       memcpy(alloc(s2.len), s2.str, s2.len);
      } else {
          char *p;
          text.str = p = alloc(s1.len + s2.len);
          memcpy(p, s1.str, s1.len);
          memcpy(p + s1.len, s2.str, s2.len);
      }
      return text;
  }
```

Text_reverse将以相反的顺序返回s，它只有两种重要的特殊情况：s为空字符串时和只有一个字符时：

```
⟨functions 278⟩+≡
  T Text_reverse(T s) {
```

```
        assert(s.len >= 0 && s.str);
        if (s.len == 0)
            return Text_null;
        else if (s.len == 1)
            return s;
        else {
            T text;
            char *p;
            int i = s.len;
            text.len = s.len;
            text.str = p = alloc(s.len);
            while (--i >= 0)
                *p++ = s.str[i];
            return text;
        }
    }
```

Text_map的实现与Str_map的实现类似。首先，它用from和to字符串建立一个映射到字符的数组。如果给定输入字符c，则map [c] 就是出现在输出字符串中的字符。map被初始化成map [k] 等于k，然后from中的字符被用来索引map中的元素，其中这些元素将要与to中的相应字符一一对应：

```
⟨rebuild map 283⟩≡
    int k;
    for (k = 0; k < (int)sizeof map; k++)
        map[k] = k;
    assert(from->len == to->len);
    for (k = 0; k < from->len; k++)
        map[from->str[k]] = to->str[k];
    inited = 1;
```

283

在map被初始化后，标志inited设为1。inited用来实现可检查的运行期错误，第一次调用Text_map必须定义非空的from和to字符串，否则将通过inited生成可检查的运行期错误。

```
⟨functions 278⟩+≡
    T Text_map(T s, const T *from, const T *to) {
        static char map[256];
        static int inited = 0;

        assert(s.len >= 0 && s.str);
        if (from && to) {
            ⟨rebuild map 283⟩
        } else {
            assert(from == NULL && to == NULL);
            assert(inited);
        }
        if (s.len == 0)
            return Text_null;
        else {
            T text;
```

```
            int i;
            char *p;
            text.len = s.len;
            text.str = p = alloc(s.len);
            for (i = 0; i < s.len; i++)
                *p++ = map[s.str[i]];
            return text;
        }
    }
```

Str_map不需要inited标志，因为它不可能将一个字符同Str_map中的空字符对应起来，断言声明map ['a'] 非零就足以实现可检查的运行期错误（请参见15.3节相关内容）。然而，Text_map允许所有可能的映射，因此不能使用map中的值来进行错误检查。

284

Text_cmp比较字符串s1和s2，并根据s1小于s2、s1等于s2和s1大于s2三种情况分别返回小于0、等于0和大于0的值。一个重要的特殊情形是当s1和s2指向同一个字符串时，短的字符串将小于长的字符串。同样地，如果一个字符串是另一个字符串的前缀，则短的那个字符串小。

```
⟨functions 278⟩+≡
  int Text_cmp(T s1, T s2) {
      assert(s1.len >= 0 && s1.str);
      assert(s2.len >= 0 && s2.str);
      if (s1.str == s2.str)
          return s1.len - s2.len;
      else if (s1.len < s2.len) {
          int cond = memcmp(s1.str, s2.str, s1.len);
          return cond == 0 ? -1 : cond;
      } else if (s1.len > s2.len) {
          int cond = memcmp(s1.str, s2.str, s2.len);
          return cond == 0 ? +1 : cond;
      } else
          return memcmp(s1.str, s2.str, s1.len);
  }
```

16.2.2 内存管理

Text执行它自己的内存分配程序，以便它能利用Text_dup和Text_cat中相邻的字符串。既然字符串空间只容纳字符，Text的分配程序也可以避免存储块头部（block header）和对齐问题，这样就可以节省空间。分配程序是第6章讲述的场分配程序（arena allocator）的一个简单变体。字符串空间类似一个单个的场，在这里被分配的存储块（chunk）出现在从head引出的列表中：

```
⟨data 277⟩+≡
  static struct chunk {
      struct chunk *link;
      char *avail;
      char *limit;
  } head = { NULL, NULL, NULL }, *current = &head;
```

285

limit域指向存储块末尾后面的一个字节，avail指向第一个空闲字节，link指向下一个存储块，所指向的这些字节或者存储块都是空闲的。current指向当前的存储块，即是在当前存储块中分配的块。上述定义初始化current，使之指向一个零长度的存储块；第一次分配将向head添加一个新的存储块。

alloc从当前存储块中分配len个字节，或者分配一个至少10k字节的新存储块：

```
⟨static functions 286⟩≡
  static char *alloc(int len) {
      assert(len >= 0);
      if (current->avail + len > current->limit) {
          current = current->link =
              ALLOC(sizeof (*current) + 10*1024 + len);
          current->avail = (char *)(current + 1);
          current->limit = current->avail + 10*1024 + len;
          current->link = NULL;
      }
      current->avail += len;
      return current->avail - len;
  }
```

current->avail是字符串空间的尾部空闲字节的地址。如果s.str+s.len等于current->avail，则Text_T型的s出现在字符串空间的尾部。因此宏isatend定义如下：

```
⟨macros 278⟩+≡
  #define isatend(s, n) ((s).str+(s).len == current->avail\
      && current->avail + (n) <= current->limit)
```

Text_dup和Text_cat只有在该存储块中有足够的空闲空间时才能利用出现在字符串空间尾部的字符串，此空闲空间是针对isatend的第二个参数而言。

Text_save和Text_restore为客户调用程序提供了存储和恢复字符串空间尾部位置的方法，其中该位置的值是由current和current->avail的值确定的。Text_save返回下述实例的一个隐式指针：

286

```
⟨types 287⟩≡
  struct Text_save_T {
      struct chunk *current;
      char *avail;
  };
```

该实例中带有current和current->avail的值。

```
⟨functions 278⟩+≡
  Text_save_T Text_save(void) {
      Text_save_T save;

      NEW(save);
      save->current = current;
      save->avail = current->avail;
      alloc(1);
```

```
        return save;
    }
```

Text_save调用alloc（1）在字符串空间中创建一个“洞（hole）”，在洞生成之前，isatend无法分配任何字符串。因此，字符串不能跨越返回给客户调用程序的字符串空间的尾部。

Text_restore恢复current和current->avail的值，释放当前块后面所有的存储块。

```
⟨functions 278⟩+≡
  void Text_restore(Text_save_T *save) {
      struct chunk *p, *q;

      assert(save && *save);
      current = (*save)->current;
      current->avail = (*save)->avail;
      FREE(*save);
      for (p = current->link; p; p = q) {
          q = p->link;
          FREE(p);
      }
      current->link = NULL;
  }
```

287

16.2.3 分析字符串

由Text导出的其他函数都检测字符串，但都不分配新的字符串。

Text_chr搜索一个字符出现在s [i:j] 中最左端的位置：

```
⟨functions 278⟩+≡
  int Text_chr(T s, int i, int j, int c) {
      ⟨convert i and j to indices in 0..s.len 279⟩
      for ( ; i < j; i++)
          if (s.str[i] == c)
              return i + 1;
      return 0;
  }
```

如果s.str [i] 等于c，则i+1就是s中那个字符左面的位置。Text_rchr功能相近，但是寻找c出现的最右面的位置。

```
⟨functions 278⟩+≡
  int Text_rchr(T s, int i, int j, int c) {
      ⟨convert i and j to indices in 0..s.len 279⟩
      while (j > i)
          if (s.str[--j] == c)
              return j + 1;
      return 0;
  }
```

Text_upto和Text_rupto与Text_chr和Text_rchr功能也类似，只是它们搜索一个字符集合中的字符出现的位置，其中该集合是由Text_T说明的：

```
⟨functions 278⟩+≡
  int Text_upto(T s, int i, int j, T set) {
      assert(set.len >= 0 && set.str);
      ⟨convert i and j to indices in 0..s.len 279⟩
      for ( ; i < j; i++)
          if (memchr(set.str, s.str[i], set.len))
              return i + 1;
      return 0;
  }

  int Text_rupto(T s, int i, int j, T set) {
      assert(set.len >= 0 && set.str);
      ⟨convert i and j to indices in 0..s.len 279⟩
      while (j > i)
          if (memchr(set.str, s.str[--j], set.len))
              return j + 1;
      return 0;
  }
```

288

Str_upto和Str_rupto使用标准C库函数来检查s中的字符是否出现在set中。Text函数不能使用strchr，因为s和set都可能包含空字符，因此它们使用memchr，memchr不把空字符解释为字符串的结束符。

Text_find和Text_rfind寻找字符串在s [i:j] 中出现的位置，它们也有类似的问题：这些函数的Str变量使用strncmp比较子字符串，但是Text函数必须使用memcmp，因为memcmp能处理带有空字符的情况。Text_find使用memcmp搜索str给定的字符串在s [i:j] 中出现的最左面的位置。值的注意的是str是空字符串时和str只有一个字符时这两种情况。

```
⟨functions 278⟩+≡
  int Text_find(T s, int i, int j, T str) {
      assert(str.len >= 0 && str.str);
      ⟨convert i and j to indices in 0..s.len 279⟩
      if (str.len == 0)
          return i + 1;
      else if (str.len == 1) {
          for ( ; i < j; i++)
              if (s.str[i] == *str.str)
                  return i + 1;
      } else
          for ( ; i + str.len <= j; i++)
              if (equal(s, i, str))
                  return i + 1;
      return 0;
  }
```

289

```
⟨macros 278⟩+≡
  #define equal(s, i, t) \
      (memcmp(&(s).str[i], (t).str, (t).len) == 0)
```

通常情况下，Text_find不能检查超过s [i:j] 的字符串，这是循环结束的条件。

Text_rfind类似于Text_find，但是它搜索str出现的最右边的位置，从而避免检测出现在s

[i:j] 前面的字符。

```
⟨functions 278⟩+≡
  int Text_rfind(T s, int i, int j, T str) {
      assert(str.len >= 0 && str.str);
      ⟨convert i and j to indices in 0..s.len 279⟩
      if (str.len == 0)
          return j + 1;
      else if (str.len == 1) {
          while (j > i)
              if (s.str[--j] == *str.str)
                  return j + 1;
      } else
          for ( ; j - str.len >= i; j--)
              if (equal(s, j - str.len, str))
                  return j - str.len + 1;
      return 0;
  }
```

Text_any扫描s中位于i右面的字符，如果该字符出现在set中则返回Text_pos(s,i)+1。

```
⟨functions 278⟩+≡
  int Text_any(T s, int i, T set) {
      assert(s.len >= 0 && s.str);
      assert(set.len >= 0 && set.str);
      i = idx(i, s.len);
      assert(i >= 0 && i <= s.len);
      if (i < s.len && memchr(set.str, s.str[i], set.len))
          return i + 2;
      return 0;
  }
```

290

如果s [i] 在set中，则Text_any返回i+2，因为i+1是s [i] 的位置，因此i+2是s [i] 后面的位置。

Text_many和Text_rmany经常在Text_upto和Text_rupto的后面调用。它们扫描一个或更多个由一个集合set提供的字符并返回第一个不在集合中的字符的左面的位置。Text_many扫描出现在s [i:j] 开头的字符。

```
⟨functions 278⟩+≡
  int Text_many(T s, int i, int j, T set) {
      assert(set.len >= 0 && set.str);
      ⟨convert i and j to indices in 0..s.len 279⟩
      if (i < j && memchr(set.str, s.str[i], set.len)) {
          do
              i++;
          while (i < j
          && memchr(set.str, s.str[i], set.len));
          return i + 1;
      }
      return 0;
  }
```

Text_rmany向左扫描set中出现在s [i:j] 尾部的一个或更多的字符：

```
⟨functions 278⟩+≡
  int Text_rmany(T s, int i, int j, T set) {
      assert(set.len >= 0 && set.str);
      ⟨convert i and j to indices in 0..s.len 279⟩
      if (j > i && memchr(set.str, s.str[j-1], set.len)) {
          do
              --j;
          while (j >= i
          && memchr(set.str, s.str[j], set.len));
          return j + 2;
      }
      return 0;
  }
```

do-while循环在j不是set中的字符的索引或者j等于i-1时结束。第一种情况，j+2是导致循环终止的字符右面的位置，因此是set中字符左面的位置。第二种情况，j+2是s [i:j] 左面的位置，其中s [i:j] 含有set中所有的字符。 291

如果s [i:j] 从str给定的字符串开始的，则Text_match扫描那个字符串是否出现。与Text_find类似，Text_match的重要的特殊情况是str为空字符串时和str只有一个字符时。Text_match不能检测s [i:j] 外面的字符串，下面第三个if条件确保只检查s [i:j] 中的字符。

```
⟨functions 278⟩+≡
  int Text_match(T s, int i, int j, T str) {
      assert(str.len >= 0 && str.str);
      ⟨convert i and j to indices in 0..s.len 279⟩
      if (str.len == 0)
          return i + 1;
      else if (str.len == 1) {
          if (i < j && s.str[i] == *str.str)
              return i + 2;
      } else if (i + str.len <= j && equal(s, i, str))
          return i + str.len + 1;
      return 0;
  }
```

Text_rmatch与Text_match类似，但是如果s [i:j] 以那个字符串结束，则它返回str中字符串之前的位置。注意不要检查s [i:j] 前面的字符。

```
⟨functions 278⟩+≡
  int Text_rmatch(T s, int i, int j, T str) {
      assert(str.len >= 0 && str.str);
      ⟨convert i and j to indices in 0..s.len 279⟩
      if (str.len == 0)
          return j + 1;
      else if (str.len == 1) {
          if (j > i && s.str[j-1] == *str.str)
              return j;
      } else if (j - str.len >= i
```

```
        && equal(s, j - str.len, str))
292         return j - str.len + 1;
        return 0;
    }
```

16.2.4 转换函数

最后一个函数是Text_fmt，它是一个格式转换函数，同Fmt接口导出的函数一起使用。Text_fmt用于以与printf的%s格式相同的样式打印Text_T。它调用Fmt_puts，Fmt_puts为Text_T规定了flags、width和precision，方式与printf为C字符串定义的方式相同。

```
⟨functions 278⟩+≡
  void Text_fmt(int code, va_list *app,
      int put(int c, void *cl), void *cl,
      unsigned char flags[], int width, int precision) {
      T *s;

      assert(app && flags);
      s = va_arg(*app, T*);
      assert(s && s->len >= 0 && s->str);
      Fmt_puts(s->str, s->len, put, cl, flags,
          width, precision);
  }
```

不同于Text接口中所有其他的函数，Text_fmt带有一个指向Text_T的指针，而不是Text_T本身。Text_T很小，一般为两个字大小，一般是不可能辨别两个字的结构和可变长度参数列表中的双精度型之间的区别的。因此，一些C实现不能通过可变长度参数列表的值来传递两个字的结构。传递Text_T指针在所有的实现中避免了这些问题。

参考书目浅析

Text_T同SNOBOL4（Griswold 1972）和Icon（Griswold和Griswold 1990）在语义和实现上都是类似的。这两种语言都是通用的、字符串处理的语言，并且都有与Text导出的函数
293 类似的内嵌特征。

类似的表示和处理字符串的技术已经长期用于编译器和其他对字符串进行分析的应用中。XPL编译器发生器（McKeeman，Horning和Wortman1970）是一个早期的例子。在识别Text_T的系统里，无用单元收集程序来处理字符串空间。Icon用XPL的无用单元收集程序来回收没有被任何已知的Text_T（Hanson 1980）所引用的字符串空间。它通过将Text_T复制到字符串空间的开始部分来使字符串变得紧凑。

Hansen（1992）描述了字符串的一个完全不同的表述方法，在该字符串中，一个子字符串描述符带有足够的信息来恢复它所属的较大的字符串。这种表示方法使得在其他事物中间向左或者向右扩展字符串成为可能。

“rope”是另外一种表示字符串的方法，在这种表示方法中，字符串由子字符串的树来表示（Boehm、Atkinson和Plass 1995）。在rope中的字符可以在线性时间内移动，就像那些在Text_T中的字符或者在C字符串中的字符那样，但是子字符串操作将花费对数时间。但是它的连接速度快得多：连接两个rope花费的是常量时间。另外一个有用的特性是一个rope可以由一个能产生第i个字符的函数来描述。

练习

16.1 使用Text函数重新写第15.2节中讲述的ids.c。

16.2 Text_save和Text_restore并不十分完善。例如下面的序列是错误的，但是没有被检测到。

```
Text_save_T x, y;
x = Text_save();
...
y = Text_save();
...
Text_restore(&x);
...
Text_restore(&y);
```

294

调用Text_restore(&x)之后，y就无效了，因为它描述的是x之后的一个字符串空间的位置。修改Text的实现，以便能检测到该错误。

16.3 Text_save和Text_restore只允许堆栈式的分配。无用单元收集程序可能更好一些，但是需要已知所有可以访问的Text_T。设计一个Text函数的扩展版本，使它包含注册Text_T的函数和Text_compact函数，其中Text_compact函数利用Hanson（1980）中描述的方法将所有已经注册的Text_T所引用的字符串放到字符串空间的开始部分，从而回收被没有注册的Text_T所占用的空间。

16.4 将搜索字符串的函数加以扩展，比如Text_find和Text_match，使它们接受的Text_T可以指定正则表达式而不仅仅是字符串。Kernighan和Plauger（1976）描述了正则表达式和匹配它们的自动机的实现。

16.5 基于Hansen（1992）所描述的子字符串模型，设计一个接口和实现。

295

第17章　扩展精度算法

32位整型格式的计算机能够表示从–2 147 483 648到+2 147 483 647的有符号整数（用2的补码表示）和从0到4 294 967 295的无符号整数，这样的范围对于大部分应用程序来说，已经足够大了，但是也有一些应用程序需要更大的范围。在相对紧凑的数值范围里，整型表示每一个整数值。而在极大的数值范围里，浮点数表示相对较少的值。仅当可以接受精确值的近似时，才能用浮点数，如同在许多科技应用中那样，但当需要很大的数值范围里所有的整数值时，就用不上浮点数。

本章表述了一个低级接口，XP，其导出的函数针对在固定精度的扩展整数上的算法运算。XP表示值的范围仅局限于所利用的存储器的大小。此接口被设计用来服务于下两章将会讨论到的高级接口。在那些需要用到潜在的极大数值范围里的整数的应用程序中，将会用到这些接口。

17.1　接口

一个n位无符号整数x可由多项式 $x = x_{n-1}\,b^{n-1} + x_{n-2}\,b^{n-2} + \ldots + x_1\,b^1 + x_0$ 表示。在这里b为底数且$0 \leqslant x_i < b$。在有着32位无符号整型格式的计算机上，n是32、b是2，并且用32位中的一位表示系数x_i。此种表示能被推广表示在任何底数下的一个无符号整数。例如，如果b是10，那么每一个x_i是0到9之间的一个数，并且x能用一个数组表示，比如用数组表示数字2 147 483 647，

```
unsigned char x[] = { 7, 4, 6, 3, 8, 4, 7, 4, 1, 2 };
```

这里x[i]保存x_i。数字x_i出现在x里、并从x_i里的最低有效数字开始，这是用来实现算法操作最便利的排序方式。

选择较大的底数可以节省存储器容量，因为底数越大，数字也就越大。例如，如果b是$2^{16} = 65\ 536$，那么每一个数字是0到包括65 535之间的一个数，因此仅需要两个数字（四个字节）来表示2 147 483 647：

```
unsigned short x[] = { 65535, 32767 };
```

以及64位数字

34905295108476594914784961990389813341776463849338784399082057 7

可由具有14个元素（28个字节）的数组表示：

```
{ 38625,  9033, 28867,  3500, 30620, 54807, 4503,
  60627, 34909, 43799, 33017, 28372, 31785,    8 }.
```

但如果b是2^k，并且k的位数大小是C中预先定义的无符号整数类型的一种，那么用较小的底数就会节省空间。而且也许更为重要的是，大的底数会使一些算法操作实现起来更为复杂。如下面详述的那样，如果一个无符号长整数能够保存b^3-1，那么就能避免这些复杂操作。XP使用的是$b=2^8$并且存储一个无符号字符里的每一位，这是因为标准C保证一个无符号长字符至少有32位，其中至少保存3个字节，因此一个无符号长字符能够保存值$b^3-1=2^{24}-1$。用$b=2^8$，四个字节就可以表示值2 147 483 647：

```
unsigned char x[] = { 255, 255, 255, 127 };
```

298 27个字节就可以表示如上所示的64位数字：

```
{ 225, 150, 73, 35, 195, 112, 172,  13, 156, 119, 23, 214, 151, 17,
  211, 236, 93, 136, 23, 171, 249, 128, 212, 110, 41, 124,   8 }.
```

XP接口揭示了下面的表示细节：

```
⟨xp.h⟩≡
  #ifndef XP_INCLUDED
  #define XP_INCLUDED

  #define T XP_T
  typedef unsigned char *T;

  ⟨exported functions 299⟩

  #undef T
  #endif
```

更确切地说，XP_T就是一个无符号字符数组，保存以2^8为底数的n位数，最低有效位在前。

以下描述的XP函数都把n作为输入参数，XP_T作为输入输出参数，数组必须足够容纳n个数字。如果向接口传送一个空XP_T，或者传送的这个XP_T太小，或者其没有非正的长度，那么会产生不可检查的运行期错误。XP是一个危险的接口，因为它忽略了多数可检查的运行期错误。进行这样的设计有两方面的原因。XP面对的客户端程序是高级接口，为了避免错误也许其自身会实现可检查的运行期错误，第二，如果必须考虑性能的话，XP的接口应尽可能的简单，这样一些函数可以用汇编语言实现。第二个因素也是为什么XP的任何函数都不进行分配操作的原因。

函数

```
⟨exported functions 299⟩≡
  extern int XP_add(int n, T z, T x, T y, int carry);
  extern int XP_sub(int n, T z, T x, T y, int borrow);
```

实现了z=x+y+carry和z=x−y−borrow。这里和以下的x、y和z指的是数组x、y和z表示的整数值，它们被认为含有n位数。carry和borrow必须为0或1。XP_add用数组z[0..n-1]来保存n

299 位求和 x+y+carry，并返回最终进位的最高有效位。XP_sub用数组z[0..n-1]来保存n位求差x−y−borrow，并返回最终借位的最高有效位。因而，如果XP_add返回1，那么x+y+carry不填充n

个数字，如果XP_sub返回1，那么y>x。，针对这两个函数，对于任何x、y和z，它们属于同一个XP_T并不是一个错误。

```
⟨exported functions 299⟩+≡
  extern int XP_mul(T z, int n, T x, int m, T y);
```

实现z=z+x·y，这里x有n个数字，y有m个数字。z必须足够的大以保存n+m个数字：XP_mul将z加上n+m位数字的乘积x·y，并把结果赋于z。当z被初始化为0时，XP_mul用数组z[0..n+m-1]来保存x·y，XP_mul返回n+m位数字乘积最终进位的最高有效位。对于z来说，如果与x或y有相同的XP_T则会产生不可检查的运行期错误。

XP_mul阐明了在何处const限定词可以帮助区分输入和输出参数并记录这些类型的运行期错误。声明

```
extern int XP_mul(T z, int n, const unsigned char *x,
                               int m, const unsigned char *y);
```

使得XP_mul读x，y和写z的过程更为清楚。因而指出z不应该与x或y相同。x和y不能用const T，因为const T的意思是“指向一个无符号字符的常量指针”，而不是想要的“指向一个无符号常量字符的指针”（见2.4节相关内容定义）。练习19.5研究了其他一些正确使用const T的定义。

然而，因为一个无符号char*能够传递给const无符号char*，所以const限定符不能阻止传递与x和z（或者y和z）相同的XP_T。但是const的用法却允许一个const无符号char*传递x和y；在以上针对XP_mul的XP的声明里，必须用类型强制转换来传递这些值。在XP中，const的巨大好处并不能抵消它的冗长。

函数

```
⟨exported functions 299⟩+≡
  extern int XP_div(int n, T q, T x, int m, T y, T r,T tmp);
```

实现了除法：它计算了q=x/y和r=x mod y；q和x有n位数，r和y有m位数。如果y等于0，XP_div返回0并保持q和r不变。否则，将返回1。tmp必须至少能够有n+m+2位数。q或r是x或y中的一个、q和r有相同的XP_T，或者tmp太小，以上这些情况都会产生不可检查的运行期错误。 300

函数

```
⟨exported functions 299⟩+≡
  extern int XP_sum     (int n, T z, T x, int y);
  extern int XP_diff    (int n, T z, T x, int y);
  extern int XP_product (int n, T z, T x, int y);
  extern int XP_quotient(int n, T z, T x, int y);
```

实现了加法、减法、乘法以及一个n位XP_T被底数为2^8的单位数y除的除法。XP_sum用数组z[0..n-1]保存x+y并返回最终进位的最高有效位。XP_diff用数组z[0..n-1]保存x-y并返回最终借位的最高有效位。对于XP_sum和XP_diff，y必须为正并且不能超出其底数2^8。

XP_product用数组z[0..n-1]保存x · y并返回最终进位的最高有效位；进位可能与2^8-1一样大。XP_quotient用数组z[0..n-1]保存x/y并返回，x mod y；余数能与y-1一样大。对于XP_product和XP_quotient来说，y不能超过2^8-1。

```
⟨exported functions 299⟩+≡
  extern int XP_neg(int n, T z, T x, int carry);
```

XP_neg用数组z[0..n-1]保存~x+carry并返回最终进位的最高有效位。当carry为0时，XP_neg实现了一个数的取反；当carry为1时，XP_neg实现了一个数的补码。

301 XP_T可以通过函数XP_cmp进行比较

```
⟨exported functions⟩+≡
  extern int XP_cmp(int n, T x, T y);
```

函数会返回一个小于、等于或大于0的值，分别对应x<y、x=y或x>y。

XP_T能通过以下函数进行移位操作

```
⟨exported functions 299⟩+≡
  extern void XP_lshift(int n, T z, int m, T x,
      int s, int fill);
  extern void XP_rshift(int n, T z, int m, T x,
      int s, int fill);
```

函数将把左移或右移s位的x的值赋给z，这里z有n位数字，x有m位数字。当n超出m时，因x最高有效位的移动而所需的填充值，如果右移就等于0，如果左移就等于fill值。空出位由fill来填充，必须等于0或1。当fill为0时，XP_rshift实现一个逻辑右移；当fill位1时，XP_rshift能被用来实现一个算术右移。

```
⟨exported functions 299⟩+≡
  extern int           XP_length (int n, T x);
  extern unsigned long XP_fromint(int n, T z,
      unsigned long u);
  extern unsigned long XP_toint  (int n, T x);
```

XP_length返回x里的数字位个数；就是说，它返回数组x[0..n-1]里的最高有效非0位的索引值加1。XP_fromint用数组z[0..n-1]保存u mod 2^{8n}并返回$u/2^{8n}$；就是说，返回不能存放在z里的u的位数。XP_toint返回x mod (ULONG_MAX+1)；就是说，它返回的是x里的8 · sizeof(unsigned long)的最低有效位。

剩下的XP函数可以在字符串和XP_T之间进行转换。

```
⟨exported functions 299⟩+≡
  extern int   XP_fromstr(int n, T z, const char *str,
      int base, char **end);
  extern char *XP_tostr  (char *str, int size, int base,
      int n, T x);
```

XP_fromstr像C库函数中的strtoul；它把str里的字符串当作以base为底数的一个无符号整数。它忽略引导空白符，并接受以base为底数的一个或更多的数字。对于11和36之间的底数，

XP_fromstr把大写或小写的字符当作大于9的数字。如果底数base小于2或者大于36，就会产生可检查的运行期错误。

n位数的XP_T，z，利用通用的倍增算法来累加str里的指定整数：

302

```
for (p = str; *p is a digit; p++)
    z ← base·z + *p's value
```

z不初始化为0；客户必须正确地初始化z。XP_fromstr返回base · z乘积最终进位的第一个非0值，否则返回0。因而，如果数字不能存放在数组z里，那么XP_fromstr就会返回最终进位的非0值。

如果end非空，由于可以扫描到乘法溢出和是否为数字，那么分配给*end的指针就指向结束XP_fromstr解释执行的字符。如果str里的字符不是以base为底数的一个整数并且如果end非空，那么XP_fromstr就会返回0，并设置*end为str。str如果为空值，就会产生可检查的运行期错误。

XP_tostr用含空终止字符的字符串来填充str，这些字符串是以base为底数的数字x的字符表示，XP_tostr的返回值是str。而x被设置为0。当底数base超过10时，可以用大写字符来表示超出9的数字。如果底数base小于2或者大于36，就会产生可检查的运行期错误。如果str为空、或者size太小，也会产生可检查的运行期错误；就是说，如果x的字符表示加上一个空字符要求了大于size个字符空间会产生可检查的运行期错误。

17.2 实现

```
⟨xp.c⟩≡
  #include <ctype.h>
  #include <string.h>
  #include "assert.h"
  #include "xp.h"

  #define T XP_T
  #define BASE (1<<8)

  ⟨data 320⟩
  ⟨functions 304⟩
```

XP_fromint和XP_toint显示了这种类型的XP函数必须执行的算术操作。XP_fromint初始化一个XP_T，使得XP_T等同于一个无符号长整型值：

303

```
⟨functions 304⟩≡
  unsigned long XP_fromint(int n, T z, unsigned long u) {
      int i = 0;

      do
          z[i++] = u%BASE;
      while ((u /= BASE) > 0 && i < n);
      for ( ; i < n; i++)
```

```
            z[i] = 0;
        return u;
    }
```

u%BASE并不是严格需要的，因为数组z[i]的分配隐含了取余操作。所有XP的算术函数都显式地进行此类操作，以此来提供它们用到的算法的文档。由于底数是2的固定常数幂值，大多数编译器将乘法、除法或取余运算转换成对底数进行对应位数的左移、右移或逻辑与。

XP_toint是XP_fromint的逆过程：它返回作为无符号长整型的XP_T的8 · sizeof (unsigned long)的最低有效位。

```
⟨functions 304⟩+≡
  unsigned long XP_toint(int n, T x) {
      unsigned long u = 0;
      int i = (int)sizeof u;

      if (i > n)
          i = n;
      while (--i >= 0)
          u = BASE*u + x[i];
      return u;
  }
```

当一个非0、n位的XP_T有一个或更多的引导0时，它的有效位数小于n。XP_length返回有效数字的个数，而不计算引导0的个数。

```
⟨functions 304⟩+≡
  int XP_length(int n, T x) {
      while (n > 1 && x[n-1] == 0)
          n--;
      return n;
  }
```

304

17.2.1 加法和减法

实现加减的算法是笔算技术的系统再现。一个底数为10的例子最有效地说明了加法 $z=x+y$：

$$\begin{array}{rcccc} & 1 & 0 & 1 & 0 \\ & 9 & 4 & 2 & 8 \\ + & & 7 & 3 & 2 \\ \hline 1 & 0 & 11 & 06 & 10 \end{array}$$

加法从最低有效位开始一直加到最高有效位，并且，在此例中，进位的初始值为0。每一步形成总和S=carry+x_i+y_i；为$S \bmod b$，新的进位是S/b，在此例中，b是底数10。在最上面一行的小数字是进位值。在最底下一行的双位位数数值是S的值。在此例里，最终进位值是1，这是因为总和不能保存成4位数字。XP_add准确执行了此算法，并且返回了最后的进位值。

```
⟨functions 304⟩+≡
  int XP_add(int n, T z, T x, T y, int carry) {
      int i;

      for (i = 0; i < n; i++) {
          carry += x[i] + y[i];
          z[i] = carry%BASE;
          carry /= BASE;
      }
      return carry;
  }
```

在每一次反复执行加法的运算中，carry暂时保存了单位位数数字总和S；那么它也就正好保存了进位值。每一个数字是0到$b-1$之间的一个数，并且进位值可以是0或1，因此$(b-1)+(b-1)+1=2b-1=511$是一个单位位数数字总和的最大值，很容易保存在一个整型中。 305

减法，$z=x-y$，类似于加法：

$$\begin{array}{rrrrr} 0 & 1 & 1 & 0 & 0 \\ & 9 & 4 & 2 & 8 \\ - & & 7 & 3 & 2 \\ \hline & 18 & 06 & 09 & 16 \end{array}$$

减法从最低有效位进行减法一直到最高有效位，在此例里，借位的初始化值是0。每一步形成的差$D=x_i+b-\text{borrow}-y_i$；$z_i$是$D \bmod b$，新的借位值是$1-D/b$。在最上面一行的小数字是借位值，在最底下一行的两位位数数字是D的值。

```
⟨functions 304⟩+≡
  int XP_sub(int n, T z, T x, T y, int borrow) {
      int i;

      for (i = 0; i < n; i++) {
          int d = (x[i] + BASE) - borrow - y[i];
          z[i] = d%BASE;
          borrow = 1 - d/BASE;
      }
      return borrow;
  }
```

D至多为$(b-1)+b-0-0=2b-1=511$，可以保存在一个整型中。如果最后的借位非0，那么x小于y。

单位位数数字的加法和减法比许多通用函数要简单一点，并且它们用第二个操作数作为进位或借位。

```
⟨functions 304⟩+≡
  int XP_sum(int n, T z, T x, int y) {
      int i;

      for (i = 0; i < n; i++) {
```

```
            y += x[i];
            z[i] = y%BASE;
            y /= BASE;
        }
        return y;
    }
```

306

```
    int XP_diff(int n, T z, T x, int y) {
        int i;

        for (i = 0; i < n; i++) {
            int d = (x[i] + BASE) - y;
            z[i] = d%BASE;
            y = 1 - d/BASE;
        }
        return y;
    }
```

XP_neg同一位数字加法类似，但是在做加法之前，x的数字已被求补：

```
⟨functions 304⟩+≡
    int XP_neg(int n, T z, T x, int carry) {
        int i;

        for (i = 0; i < n; i++) {
            carry += (unsigned char)~x[i];
            z[i] = carry%BASE;
            carry /= BASE;
        }
        return carry;
    }
```

强制类型转换确保了~x[i]小于b。

17.2.2 乘法

如果x有n位数字，y有m位数字，那么$z=x \cdot y$是m个部分组成的乘积，每一部分有n位数字，这m部分乘积的和形成了$n+m$位乘积结果值。下例说明了当z的初始值为0时，$n=4$和$m=3$的乘法过程：

307

$$
\begin{array}{rrrrrrr}
 & & & & 7 & 3 & 2 \\
 & & & \times & 9 & 4 & 2 & 8 \\
\hline
 & & & & 5 & 8 & 5 & 6 \\
 & & & 1 & 4 & 6 & 4 & \\
 & & 2 & 9 & 2 & 8 & & \\
+ & 6 & 5 & 8 & 8 & & & \\
\hline
 & 6 & 9 & 0 & 1 & 2 & 9 & 6
\end{array}
$$

部分乘积不必显式计算；当计算乘积里的各个数字时，每一部分乘积值都加入到了z里。例如，在第一个部分乘积的数字里，$8 \cdot 732$，从最低有效位计算到最高有效位。按照加法里一般的进位运算，部分乘积的第i位数字加到了z的第i位数字。第二个部分乘积的第i位数字，

2·732，被加到了z的第$i+1$位数字。总之，当计算涉及x_i的部分乘积时，数字要加入到z中的位置开始于z的第i位。

```
⟨functions 304⟩+≡
  int XP_mul(T z, int n, T x, int m, T y) {
      int i, j, carryout = 0;

      for (i = 0; i < n; i++) {
          unsigned carry = 0;
          for (j = 0; j < m; j++) {
              carry += x[i]*y[j] + z[i+j];
              z[i+j] = carry%BASE;
              carry /= BASE;
          }
          for ( ; j < n + m - i; j++) {
              carry += z[i+j];
              z[i+j] = carry%BASE;
              carry /= BASE;
          }
          carryout |= carry;
      }
      return carryout;
  }
```

308

当部分乘积中的数字在第一个嵌套循环里加入到了z上时，进位最大可与$b-1$一样大，因此存储在carry里的总和能与$(b-1)(b-1)+(b-1)=b^2-b=65\ 280$一样大，它被保存为无符号数。在把部分乘积加入到z后，第二个嵌套循环把进位加入到z中剩余的数字上中，并记录在当次加法中形成的z的最高有效末位的进位值。如果进位一直等于1，那么$z+x\cdot y$的最终进位等于1。

一位数字乘法等同于调用XP_mul，m等于1，z初始化为0：

```
⟨functions 304⟩+≡
  int XP_product(int n, T z, T x, int y) {
      int i;
      unsigned carry = 0;

      for (i = 0; i < n; i++) {
          carry += x[i]*y;
          z[i] = carry%BASE;
          carry /= BASE;
      }
      return carry;
  }
```

17.2.3 除法和比较

除法是最复杂的算术函数。有若干算法可以用来完成除法设计，但每一种算法都有各自的优缺点。最容易的算法大概就是从下面的算术规则衍生出来的，这些规则通过计算$q=x/y$和$r=x \bmod y$来实现。

if $x < y$ then $q \leftarrow 0, r \leftarrow x$
else
  $q' \leftarrow x/2y, r' \leftarrow x \bmod 2y$
  if $r' < y$ then $q \leftarrow 2q', r \leftarrow r'$ else $q \leftarrow 2q' + 1, r \leftarrow r' - y$

当然，牵涉到q'和r'的中间计算过程必须利用XP_T来实现。

利用递归算法的问题是如何对q'和r'进行分配。因为lg x（以2为底的对数）是最大的递归深度，所以这些分配与lg x值一样多。XP接口禁止这些隐式的分配。

309

在通常情况下，当$x \geq y$以及y至少有两个有效位时，XP_div采用了效率很高的迭代算法；当$x<y$以及y有一位有效数字时，针对这种较容易的情况，XP_div使用了一种更为简单的算法。

```
⟨functions 304⟩+≡
  int XP_div(int n, T q, T x, int m, T y, T r, T tmp) {
      int nx = n, my = m;

      n = XP_length(n, x);
      m = XP_length(m, y);
      if (m == 1) {
          ⟨single-digit division 311⟩
      } else if (m > n) {
          memset(q, '\0', nx);
          memcpy(r, x, n);
          memset(r + n, '\0', my - n);
      } else {
          ⟨long division 312⟩
      }
      return 1;
  }
```

XP_div首先要检查一位数字除法，这是因为要处理除数为0的情况。

一位数字除法很简单，因为它使用了C语言中通用的无符号整数除法来计算商值。除法从最高有效位计算一直进行到最低有效位，进位的初始值是0，以10为底数，9 428与7相除按以下步骤所示。

$$\begin{array}{r|l} & 1\ \ 3\ \ 4\ \ 6 \\ \hline 7 & 09\ 24\ 32\ 48\ \ 6 \end{array}$$

在每一步中，部分被除数R=carry $\cdot\ b+x_i$；商值$q_i=R/y_0$，新的进位是$R \bmod y_0$。进位值是上面的小数字，进位的最后值是余数。除法操作由XP_quotient准确实现，并返回余数：

```
⟨functions 304⟩+≡
  int XP_quotient(int n, T z, T x, int y) {
  int i;
  unsigned carry = 0;

  for (i = n - 1; i >= 0; i--) {
```

310

```
        carry = carry*BASE + x[i];
        z[i] = carry/y;
        carry %= y;
    }
    return carry;
}
```

R——XP_quotient里的carry值——能与（$b-1$）$b+(b-1)=b^2-1$=65 535一样大，存成一个无符号数值。

在XP_div里，调用XP_quotient会返回r的最低有效位，因此其余位必须显式设置为0：

⟨*single-digit division* 311⟩≡

```
if (y[0] == 0)
    return 0;
r[0] = XP_quotient(nx, q, x, y[0]);
memset(r + 1, '\0', my - 1);
```

在一般情况下，当n≥m并且m>1时，一个n位被除数可被一个m位除数相除。以10为底数，615 367与296相除按以下步骤所示。在被除数的首位添加0，使得n超出m。

```
                2  0  7  8
        ---------------------
2  9  6 |  0  6  1  5  3  6  7
         -  0  5  9  2
         ---------------------
               0  2  3  3
               0  0  0  0
               ---------------
               2  3  3  6
               2  0  7  2
               ---------------
                  2  6  4  7
                  2  3  6  8
                  ------------
                     2  7  9
```

计算每一个商值，q_k，因为计算涉及m位运算，所以效率是长整型除法的症结所在。 311

假定现在我们知道如何计算商值，以下伪代码略述了长整形除法的实现。

```
rem ← x with a leading zero
for (k = n - m; k >= 0; k--) {
  compute qk
  dq ← y*qk
  q->digits[k] = qk;
  rem ← rem - dq·b^k
  }
r ← rem
```

rem开始时与x相等，且首位为0，通过用m位数字去除rem的前m+1位数字，且高位优先计算，循环共进行了n-m+1次商值计算。在每次计算结束时，rem减去qk和y的乘积，这会使得rem减少一位。如上面的例子，n=6，m=3，循环体执行4次，k=6-3=3，2，1和0。下表列出了在每一次计算中k、rem、qk和dq的值。第二列的划线部分确定了rem的前缀，这里rem与y相除，y的值是296。

k	rem	qk	dq
3	0615367	2	0592
2	023367	0	0000
1	23367	7	2072
0	2647	8	2368
	279		

XP_div需要空间来保存两个临时的rem和dq；rem需要n+1位字节，dq需要m+1位字节。在理解了上面的伪代码后，长整型除法的代码块为

```
⟨long division 312⟩≡
  int k;
  unsigned char *rem = tmp, *dq = tmp + n + 1;
  assert(2 <= m && m <= n);
  memcpy(rem, x, n);
  rem[n] = 0;
  for (k = n - m; k >= 0; k--) {
      int qk;
      ⟨compute qk, dq ← y·qk 314⟩
      q[k] = qk;
      ⟨rem ← rem - dq·b^k 315⟩
  }
  memcpy(r, rem, m);
  ⟨fill out q and r with 0s 313⟩
```

312

tmp[0..n]保存rem的n+1位数字，tmp[n+1..n+1+m]保存了dq的m+1位数字。在tmp[0..k+m]里，rem总是有k+m+1位数字。代码计算了一个n-m+1位的商和一个m位的余数；q和r的其余数字必须设置为0：

```
⟨fill out q and r with 0s 313⟩≡
  {
      int i;
      for (i = n-m+1; i < nx; i++)
          q[i] = 0;
      for (i = m; i < my; i++)
          r[i] = 0;
  }
```

所有的余数一直参与商值计算。有一个简单的——但不合适的——方法，在开始时使qk等于b-1，并逐渐递减qk，直到y·qk超出rem的m+1位前缀：

```
qk = BASE-1;
dq <- y·qk;
while (rem[k..k+m] < dq) {
  qk--;
  dq ← y·qk;
}
```

此方法太慢：循环可能需要b-1次反复计算，每一次计算需要一次m位乘法和一次m+1位比较。一个更好的方法是利用常用的整数运算更精确地估计qk，当估计错误时就纠正qk。三位数rem前缀与两位数y前缀相除可以估计出qk是正确还是太大。因此，上面的循环可用单个测试来代替：

313

```
⟨compute qk, dq ← y·qk 314⟩≡
  {
      int i;
      assert(2 <= m && m <= k+m && k+m <= n);
      ⟨qk ← y[m-2..m-1]/rem[k+m-2..k+m] 314⟩
      dq[m] = XP_product(m, dq, y, qk);
      for (i = m; i > 0; i--)
          if (rem[i+k] != dq[i])
              break;
      if (rem[i+k] < dq[i])
          dq[m] = XP_product(m, dq, y, --qk);
  }
```

如上所示，XP_product计算y[0..m-1]×qk，把结果赋予dq，返回最后的进位，也就是dq的最后位值。循环比较rem[k..k+m]和dq，一次比较一位。如果dq超出rem的m+1位前缀，那么qk就太大了，因此qk要递减，dq也要重新计算。

用常用的整数除法能够估计qk：

```
⟨qk ← y[m-2..m-1]/rem[k+m-2..k+m] 314⟩≡
  {
      int km = k + m;
      unsigned long y2 = y[m-1]*BASE + y[m-2];
      unsigned long r3 = rem[km]*(BASE*BASE) +
          rem[km-1]*BASE + rem[km-2];
      qk = r3/y2;
      if (qk >= BASE)
          qk = BASE - 1;
  }
```

r3最大能与$(b-1)b^2 + (b-1)b + (b-1) = b^3-1$=16 777 215一样大，能保存在一个无符号长整形数中。计算限制了BASE的选取。一个无符号长整形能够存储小于2^{32}的数值，这也就表明$b^3-1<2^{32}$，因此BASE必须小于$2^{10.6666}$，还不能超过1 625．256是不超过1 625的2的最大幂，这也是嵌入类型的尺寸。

314

关于长整形除法的最后一点迷惑是从rem的m+1位前缀减去dq，这会减少rem并将其缩短一位数字。从理论上来说，通过将dq左移k位数字，然后再从rem中减去该值，就可以实现减法。如上所示的XP_sub能够通过把指针指向rem里合适的数字来完成这种减法：

```
⟨rem ← rem - dq·b^k 315⟩≡
  {
      int borrow;
      assert(0 <= k && k <= k+m);
      borrow = XP_sub(m + 1, &rem[k], &rem[k], dq, 0);
```

```
        assert(borrow == 0);
    }
```

在<*compute qk, dq<–y · qk 314*>里的代码指出了两个多位位数数字通过比较它们的位数来完成比较，最高有效位优先比较。XP_cmp通过两个XP_T参数准确完成了比较：

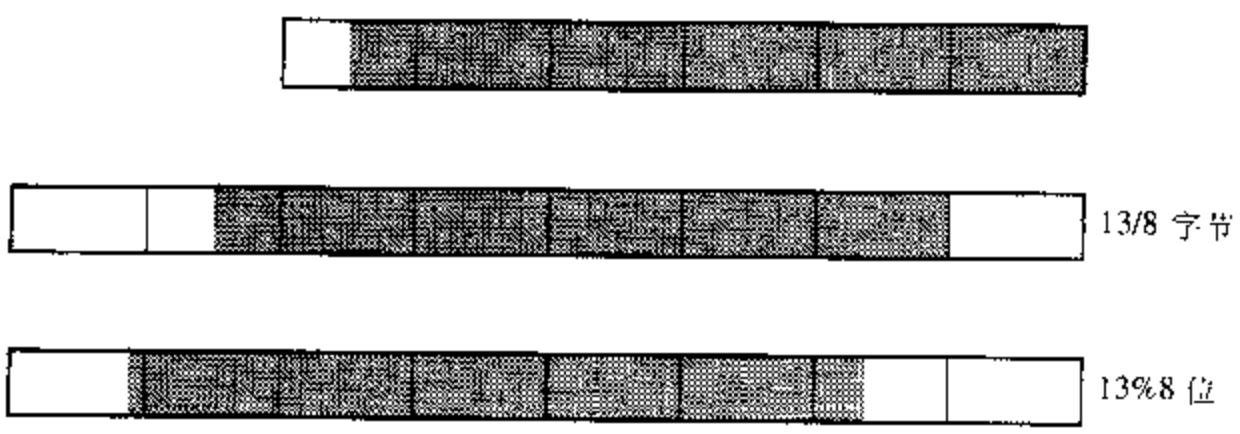
```
⟨functions 304⟩+≡
  int XP_cmp(int n, T x, T y) {
      int i = n - 1;

      while (i > 0 && x[i] == y[i])
          i--;
      return x[i] - y[i];
  }
```

17.2.4 移位

XP里的两个移位函数实现了左移和右移XP_T指定的数位。移动s位可按以下两步执行：第一步是通过一次转移一个字节来移动8·(s/8)个位，第二步是移动其余的s mod 8位，每次移动s mod 8位。fill是用来设置全为1或全为0的字节，能够被用于一次填充一个字节，如下所示：

```
⟨functions 304⟩+≡
  void XP_lshift(int n, T z, int m, T x, int s, int fill) {
      fill = fill ? 0xFF : 0;
      ⟨shift left by s/8 bytes 316⟩
315   s %= 8;
      if (s > 0)
          ⟨shift z left by s bits 317⟩
  }
```

这些步骤可由下图所示，下图显示了将一个含有44个“1”的6位XP_T左移13位转变为8位XP_T的过程；右边的浅色部分标识空位，并设置为fill值。



左移s/8字节能够通过以下分配实现。

```
z[m+(s/8)..n-1] ← 0
z[s/8..m+(s/8)-1] ← x[0..m-1]
z[0..(s/8)-1] ← fill.
```

第一次分配把z中那些左移s/8字节将会影响到的数字清0。在第二次分配中，x_i被复制到$z_{i+s/8}$，最高有效位优先。第三次分配将z的s/8最低有效位设置为fill值。每一次分配涉及一个循环，当n小于m时，初始化代码会处理这种情况：

```
⟨shift left by s/8 bytes 316⟩≡
  {
      int i, j = n - 1;
      if (n > m)
          i = m - 1;
      else
          i = n - s/8 - 1;
      for ( ; j >= m + s/8; j--)
          z[j] = 0;
      for ( ; i >= 0; i--, j--)
          z[j] = x[i];
      for ( ; j >= 0; j--)
          z[j] = fill;
  }
```

316

在第二步时，s减至移动位数。移位位数等价于z和2^s相乘，然后设置z的最低有效s位值为fill：

```
⟨shift z left by s bits 317⟩≡
  {
      XP_product(n, z, z, 1<<s);
      z[0] |= fill>>(8-s);
  }
```

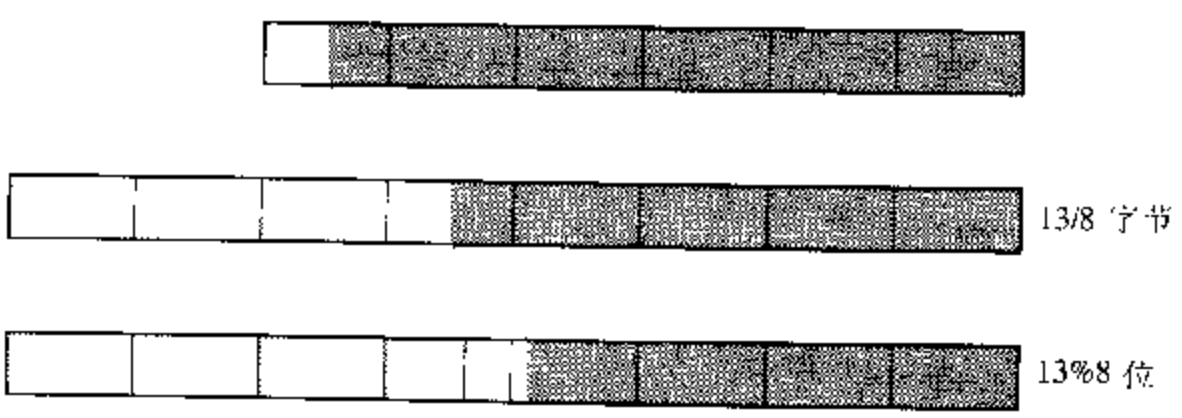
fill或者为0或者为0xFF，因此fill>>(8–s)在一个字节的最低有效位形成了s个填充位。

相似的两步骤处理也用在右移过程中：第一步向右移动s/8字节，第二步移动其余的s mod 8位。

```
⟨functions 304⟩+≡
  void XP_rshift(int n, T z, int m, T x, int s, int fill) {
      fill = fill ? 0xFF : 0;
      ⟨shift right by s/8 bytes 318⟩
      s %= 8;
      if (s > 0)
          ⟨shift z right by s bits 318⟩
  }
```

把有44个“1”的6位数XP_T右移13位转变为一个8位数XP_T，下图说明了上述步骤；左边的浅色部分标识空位数和超出位，这些被设置成fill值。

317



右移的三次分配是

```
z[0..m-(s/8)-1] ← x[s/8..m-1]
z[m-(s/8)..m-1] ← fill
z[m..n-1] ← fill.
```

第一次分配把x_i复制到$z_{i-s/8}$。从s/8字节开始，最低有效字节优先。第二次分配设置空字节为fill值，第三次分配将z里的一些不会出现在x里的数字设置成fill值。当然，第二次和第三次分配可以在同一个循环中完成：

```
⟨shift right by s/8 bytes 318⟩≡
  {
      int i, j = 0;
      for (i = s/8; i < m && j < n; i++, j++)
          z[j] = x[i];
      for ( ; j < n; j++)
          z[j] = fill;
  }
```

第二步把z右移s位，这等价于z除以2^s：

```
⟨shift z right by s bits 318⟩≡
  {
      XP_quotient(n, z, z, 1<<s);
      z[n-1] |= fill<<(8-s);
  }
```

318

表达式fill<<(8−s)在字节的最高有效位形成了s个填充位，然后，此字节与z的最高有效字节进行或操作。

17.2.5 字符串转换

最后的两个XP函数把XP_T转换成字符串，以及从字符串转换成XP_T。XP_fromstr转换字符串到XP_T；它接受一个可选空白区域以及1位或多位数的底数，范围是从2到包括36的整数区间。底数如果超出10，可以用字符表示超出9的数字。当遇上不合法的字符或者空字符时，或者当乘法中的最终进位非0时，XP_fromstr将停止扫描字符串参数。

```
⟨functions 304⟩+≡
  int XP_fromstr(int n, T z, const char *str,
      int base, char **end) {
      const char *p = str;

      assert(p);
      assert(base >= 2 && base <= 36);
      ⟨skip white space 320⟩
      if (⟨*p is a digit in base 320⟩) {
          int carry;
          for ( ; ⟨*p is a digit in base 320⟩; p++) {
              carry = XP_product(n, z, z, base);
              if (carry)
```

```
                break;
            XP_sum(n, z, z, map[*p-'0']);
        }
        if (end)
            *end = (char *)p;
        return carry;
    } else {
        if (end)
            *end = (char *)str;
        return 0;
    }
}
```

319

⟨*skip white space* 320⟩≡

```
while (*p && isspace(*p))
    p++;
```

如果end非空，XP_fromstr把*end设置成指向最后终止扫描的字符指针。

如果c是一个数字字符，那么map[c-'0']是其对应的数字值；例如，map['F'-'0']是15。

⟨*data* 320⟩≡

```
static char map[] = {
     0,  1,  2,  3,  4,  5,  6,  7,  8,  9,
    36, 36, 36, 36, 36, 36, 36,
    10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
    23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
    36, 36, 36, 36, 36, 36,
    10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
    23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35
};
```

针对那些对应于ASCII顺序的0到z之间的少数无效数字，map[c-'0']等于36。如果map[c-'0']小于base，那么选择此类数值，可以使c成为以base为底数的数字。因而，XP_fromstr会按如下语句测试是否*p是一个数字字符：

⟨*(*p *is a digit in base* 320⟩≡

```
(*p && isalnum(*p) && map[*p-'0'] < base)
```

XP_tostr利用通用的算法来计算x的字符串表示，先剥去数字的最后一位，然后一步一步来实现，但是XP_tostr使用了XP函数来完成算法。

⟨*functions* 304⟩+≡

```
char *XP_tostr(char *str, int size, int base,
    int n, T x) {
    int i = 0;

    assert(str);
    assert(base >= 2 && base <= 36);
    do {
        int r = XP_quotient(n, x, x, base);
        assert(i < size);
        str[i++] =
```

320

```
                "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"[r];
            while (n > 1 && x[n-1] == 0)
                n--;
        } while (n > 1 || x[0] != 0);
        assert(i < size);
        str[i] = '\0';
        ⟨reverse str 321⟩
        return str;
    }
```

数字以str逆序结束，因此XP_tostr终止操作时，将它们逆转。

```
⟨reverse str 321⟩≡
    {
        int j;
        for (j = 0; j < --i; j++) {
            char c = str[j];
            str[j] = str[i];
            str[i] = c;
        }
    }
```

参考书目浅析

XP中的大多数算术函数直接实现了我们每个人在小学就已经学过的运算法则。Hennessy和Patterson（1994）的第4章以及Knuth（1981）的4.3节表述了实现算术操作的经典算法，Knuth（1981）对那些算法的长期发展历史给予了精彩的总结。

除法的实现是困难的，这是因为强加在计算上的商值限制。用在XP_div中的算法来自Brinch Hansen（1994），其中有所估计的商值最大偏离为1的证明。Brinch Hansen指出了如何通过缩放操作数的比例来避免在大部分时间内纠正qk。缩放要花费一个附加的个位数乘法和除法，但在必须消减qk的情况下，可以避免二次调用product。

321

练习

17.1 用XP_div采用的Brinch-Hansen算法来实现递归除法算法，并比较它的执行时间和空间性能。递归算法在什么条件下更优越？

17.2 实现Hennessy和Patterson（1994）的第4章描述的“移位并相减”除法算法，并对它和用在XP_div里的Brinch-Hansen算法进行性能比较。

17.3 大部分XP函数花费的计算时间与操作数中的数字个数成比例。以底数为2^{16}来表示XP_T，将会使函数的运行速度快一倍。然而，对于除法却产生了一个问题，因为

$$(2^{16})^3-1 = 28\ 147\ 497\ 610\ 655$$

超出了大多数32位计算机的ULONG_MAX值，并且标准C的整数算法不能用来估计此方式下的商值。针对这个问题设计一个方法，用底数2^{16}来实现XP，并权衡利

弊。获得的好处是否值得增加除法复杂性?

17.4 用底数2^{32}完成练习17.3。

17.5 如果底数较大、如2^{32}，那么扩展精度算法可以用汇编语言实现，因为许多机器有双精度指令，并且能够容易地捕获进位和借位。汇编语言实现起来速度很快。在自己钟爱的计算机上用汇编语言实现XP，并确定速度提高的程度。

17.6 实现一个产生随机数的XP函数，随机数要求均匀地分布在指定范围内。

322

第18章　任意精度算法

本章介绍了AP接口，这个接口提供任意精度的有符号整数及其相关算法操作。与XP_T不同，AP提供的整数可以为正或为负，而且它们可以是任意位的数字。它们所能够表示的数值仅受可用存储器的限制。这些整数可用于需要无限范围整数值的应用程序。例如，一些共同基金公司跟踪股票价格非常接近于百分之一美分——1美元的1/10 000——这样，所有的计算可能都以百分之一美分为单位进行。32位的无符号整数仅可表示429,496.7295美元，而这仅仅是一些基金所持有的数十亿美元中极微小的一部分。

当然，AP使用了XP，但是AP是个高级接口：它仅展现了一种表示任意精度的带符号整数的隐式类型。AP会导出函数以分配和释放这些整数，并在这些整数上执行常用算法操作。它还实现了XP忽略的可检查的运行期错误。大多数应用程序都应使用AP或下一章所介绍的MP接口。

18.1　接口

AP接口将一个任意精度带符号整数的表示隐藏于一种隐式指针类型后面：

⟨*ap.h*⟩≡
```
#ifndef AP_INCLUDED
#define AP_INCLUDED
#include <stdarg.h>

#define T AP_T
typedef struct T *T;

⟨exported functions 324⟩

#undef T
#endif
```

给这个接口的任一函数传递空AP_T会产生一个可检查的运行期错误，但下列情况除外。下列函数可生成AP_T：

⟨*exported functions* 324⟩≡
```
extern T AP_new    (long int n);
extern T AP_fromstr(const char *str, int base,
    char **end);
```

AP_new生成一个新的AP_T，并将其初始化为值n，然后返回。AP_fromstr也生成一个新AP_T，将由str和base指定的值作为其初始值，然后返回。AP_new和AP_fromstr可以引发Mem_Failed，

AP_fromstr就像C库函数中的strtol；它把str里的字符串按base解释成一个整数。它忽略前导空格，并接受一个可选符号，其后跟随着一位或多位以base为底数的数字。对于11和36之间的底数，AP_fromstr用小写字母或大写字母解释大于9的数字。base小于2或大于36则会产生一个可检查的运行期错误。

如果end非空，*end被赋值为一个指针指向终止AP_fromstr解释的那个字符。如果str中的字符没有一个以base为底数的整数，AP_fromstr就返回空，而且，如果end不为空，就把*end设置为str。str为空则会产生一个可检查的运行期错误。

函数

```
⟨exported functions 324⟩+≡
  extern long int AP_toint(T x);
  extern char    *AP_tostr(char *str, int size,
    int base, T x);
extern void     AP_fmt(int code, va_list *app,
    int put(int c, void *cl), void *cl,
    unsigned char flags[], int width, int precision);
```

324

提取并打印AP_T所表示的整数。AP_toint返回一个符号与x相同、数值等于|x| mod (LONG_MAX+1)的长整型整数，这里LONG_MAX是最大的正长整型整数。如果x是LONG_MIN——在二进制补码机器上是-LONG_MAX-1——AP_toint就返回-((LONG_MAX+1) mod (LONG_MAX+1))，即0。

AP_tostr用无终止字符串填充str，这个字符串是以base为底数的x字符表示形式，并返回str。大写字母用于表示base超出10时那些大于9的数字。base小于2或大于36则会产生一个可检查的运行期错误。

如果str非空，AP_tostr将给str填充size数量的字符。size太小就会产生可检查的运行期错误——更确切地说，x的字符表示加上一个空字符需要多于size的字符。如果str为空，就忽略size；AP_tostr将分配足够大的字符串容纳x字符表示，并返回那个字符串。客户调用程序有责任释放分配的字符串。如果str为空，AP_tostr会引发Mem_Failed。

AP_fmt可与Fmt接口中的函数一起用作转换函数来格式化AP_T。它用掉一个AP_T，并根据可选的flags、width和precision，按照与printf的限定符%d格式化其整数参数相同的方式进行格式化。AP_fmt可以引发Mem_Failed。app或flags为空则会产生一个可检查的运行期错误。

AP_T由

```
⟨exported functions 324⟩+≡
  extern void AP_free(T *z);
```

释放。

AP_free释放*z的内容并将其设置为空。z或*z为空则会产生一个可检查的运行期错误。

下列函数在AP_T上执行算法操作。每个函数都返回一个AP_T作为结果，而且每个都可以引发Mem_Failed。

```
⟨exported functions 324⟩+≡
  extern T AP_neg(T x);
  extern T AP_add(T x, T y);
  extern T AP_sub(T x, T y);
  extern T AP_mul(T x, T y);
  extern T AP_div(T x, T y);
  extern T AP_mod(T x, T y);
  extern T AP_pow(T x, T y, T p);
```

325

AP_neg返回-x；AP_add返回x +y；AP_sub返回x -y；AP_mul返回x ·y；AP_div返回x/y；AP_mod返回x mod y．这里，x和y指的是x和y所表示的整数值。除法向左取齐：x或y有一个为负则趋向于负无穷，反之则趋向于0。因此，余数总为止。更为精确的是，商q是不超出实数w的最大整数，w ·y =x，余数定义为x-y ·q．这个定义与第2章中介绍的Arith接口所实现的定义一致。对于AP_div和AP_mod，如果y为0则为可检查的运行期错误。

如果p为空AP_pow返回x^y；如果p非空，AP_pow则返回（x^y）mod p。如果y为负数，或p非空且小于2，则为一个可检查的运行期错误。

简易函数

```
⟨exported functions 324⟩+≡
  extern T    AP_addi(T x, long int y);
  extern T    AP_subi(T x, long int y);
  extern T    AP_muli(T x, long int y);
  extern T    AP_divi(T x, long int y);
  extern long AP_modi(T x, long int y);
```

与上述函数类似，只是y取长整型。举个例子，AP_addi（x，y）等价于AP_add（x，AP_new（y））。关于除法和取余的规则与AP_div和AP_mod相同。这些函数中每个都可以引发Mem_Failed。

可用下列函数移位AP_T：

```
⟨exported functions 324⟩+≡
  extern T AP_lshift(T x, int s);
  extern T AP_rshift(T x, int s);
```

AP_lshift返回一个AP_T，其值等于x左移s位后的值，就等价于x乘以2^s。AP_rshift则返回一个值相当于x右移s位的AP_T，等价于x除以2^s。这两个函数返回值的符号都与x相同，除非进行移位的值是0；同时空出的位都设置成0。s为负则为一个可检查的运行期错误；移位函数还可以引发Mem_Failed。

326

AP_T可通过如下函数进行比较

```
⟨exported functions 324⟩+≡
  extern int AP_cmp (T x, T y);
  extern int AP_cmpi(T x, long int y);
```

x<y、x=y或x>y时，这两个函数分别返回小于0、等于0或大于0的整数。

18.2 示例：一个计算器

这里用一个执行任意精度计算的计算器说明AP接口的使用；下一节介绍的AP接口实现则阐述了XP接口的使用。

计算器calc使用波兰后缀符号：值推入堆栈顶端；运算符从栈中弹出它们的操作数并将结果推入栈。值就是一个或多个相邻十进制数；运算符则如下：

- 取反
+ 加法
- 减法
* 乘法
/ 除法
% 取余
^ 求幂
d 复制栈顶值
p 打印栈顶值
f 从栈顶向下打印栈中所有值
q 退出

空格字符用于分隔数值但本身会被忽略不记；其他字符都声明为未被识别的运算符。堆

327 栈的大小仅受限于可用存储器，但是诊断程序会声明堆栈下溢

calc是个简单的程序。它有三项主要的工作：解释输入、计算值及管理堆栈。

```
⟨calc.c⟩≡
  #include <ctype.h>
  #include <stdio.h>
  #include <string.h>
  #include <stdlib.h>
  #include "stack.h"
  #include "ap.h"
  #include "fmt.h"

  ⟨calc data 328⟩
  ⟨calc functions 328⟩
```

正如stack.h中包含的内容所表明的那样，calc的堆栈使用了第2章中介绍的堆栈接口：

```
⟨calc data 328⟩≡
  Stack_T sp;

⟨initialization 328⟩≡
  sp = Stack_new();
```

sp为空时calc肯定不能调用Stack_pop，因此它把所有的弹出操作封装在一个检查堆栈下溢的函数里：

```
⟨calc functions 328⟩≡
  AP_T pop(void) {
      if (!Stack_empty(sp))
          return Stack_pop(sp);
      else {
          Fmt_fprint(stderr, "?stack underflow\n");
          return AP_new(0);
      }
  }
```

即使堆栈为空也总是返回一个AP_T，以简化calc中其他地方的错误检查。

calc中的主循环读取下一“标志”——数值或运算符——然后根据读取内容执行相应操作： 328

```
⟨calc functions 328⟩+≡
  int main(int argc, char *argv[]) {
      int c;

      ⟨initialization 328⟩
      while ((c = getchar()) != EOF)
          switch (c) {
          ⟨cases 329⟩
          default:
              if (isprint(c))
                  Fmt_fprint(stderr, "?'%c'", c);
              else
                  Fmt_fprint(stderr, "?'\\%03o'", c);
              Fmt_fprint(stderr, " is unimplemented\n");
              break;
          }
      ⟨clean up and exit 329⟩
  }
```

```
⟨clean up and exit 329⟩≡
  ⟨clear the stack 333⟩
  Stack_free(&sp);
  return EXIT_SUCCESS;
```

输入字符可以是空格、数值的首位数、运算符或是上述代码默认情况下所显示的错误输入。空格就被忽略：

```
⟨cases 329⟩≡
  case ' ': case '\t': case '\n': case '\f': case '\r':
      break;
```

数值以数字开头；calc把跟随在第一个数字后面的数字收集在一个缓冲区，并用AP_fromstr将这串数字转换成一个AP_T：

```
⟨cases 329⟩+≡
  case '0': case '1': case '2': case '3': case '4':
  case '5': case '6': case '7': case '8': case '9': {
      char buf[512];
      ⟨gather up digits into buf 333⟩
```

329

```
        Stack_push(sp, AP_fromstr(buf, 10, NULL));
        break;
    }
```

每个运算符从栈中弹出0个或多个操作数，并将0个或多个结果推入堆栈。加法非常具有代表性：

```
⟨cases 329⟩+≡
  case '+': {
      ⟨pop x and y off the stack 330⟩
      Stack_push(sp, AP_add(x, y));
      ⟨free x and y 330⟩
      break;
  }

⟨pop x and y off the stack 330⟩≡
  AP_T y = pop(), x = pop();

⟨free x and y 330⟩≡
  AP_free(&x);
  AP_free(&y);
```

很容易产生这样的错误：堆栈中出现某个AP_T的两个或多个副本；这样就不可能知道应当释放哪个AP_T。上面的代码显示了避免出现这种问题的一个简单协议：堆栈中仅有那些“永久”的AP_T；调用AP_free释放其他所有的AP_T。

减法和乘法在形式上都与加法类似：

```
⟨cases 329⟩+≡
  case '-': {
      ⟨pop x and y off the stack 330⟩
      Stack_push(sp, AP_sub(x, y));
      ⟨free x and y 330⟩
      break;
  }
  case '*': {
      ⟨pop x and y off the stack 330⟩
      Stack_push(sp, AP_mul(x, y));
      ⟨free x and y 330⟩
      break;
  }
```

330

除法和取余也很简单，但是它们必须防止除数为0。

```
⟨cases 329⟩+≡
  case '/': {
      ⟨pop x and y off the stack 330⟩
      if (AP_cmpi(y, 0) == 0) {
          Fmt_fprint(stderr, "?/ by 0\n");
          Stack_push(sp, AP_new(0));
      } else
          Stack_push(sp, AP_div(x, y));
```

```
        ⟨free x and y 330⟩
        break;
    }
    case '%': {
        ⟨pop x and y off the stack 330⟩
        if (AP_cmpi(y, 0) == 0) {
            Fmt_fprint(stderr, "?%% by 0\n");
            Stack_push(sp, AP_new(0));
        } else
            Stack_push(sp, AP_mod(x, y));
        ⟨free x and y 330⟩
        break;
    }
```

求幂必须防止指数为非正数：

```
⟨cases 329⟩+≡
    case '^': {
        ⟨pop x and y off the stack 330⟩
        if (AP_cmpi(y, 0) <= 0) {
            Fmt_fprint(stderr, "?nonpositive power\n");
            Stack_push(sp, AP_new(0));
        } else
            Stack_push(sp, AP_pow(x, y, NULL));
        ⟨free x and y 330⟩
        break;
    }
```

331

将栈顶的数值弹出堆栈可以复制该值，这样就会检测到堆栈下溢，并将该值及其副本推入堆栈。拷贝AP_T的惟一途径就是让它加0。

```
⟨cases 329⟩+≡
    case 'd': {
        AP_T x = pop();
        Stack_push(sp, x);
        Stack_push(sp, AP_addi(x, 0));
        break;
    }
```

将AP_cvt与某一格式代码相关联，并在传递给Fmt_fmt的格式字符串中使用那个代码，就可以打印AP_T；calc使用D。

```
⟨initialization 328⟩+≡
    Fmt_register('D', AP_fmt);

⟨cases 329⟩+≡
    case 'p': {
        AP_T x = pop();
        Fmt_print("%D\n", x);
        Stack_push(sp, x);
        break;
    }
```

打印堆栈的所有数值暴露出Stack接口中的一个弱点：不能访问栈顶元素下面的数值，也就不能断定堆栈中有多少数值。更好的堆栈接口可以包含Table_length和Table_map这样的函数；没有这些函数，calc就必须创建一个临时堆栈把主堆栈的所有内容全部倒入临时堆栈中，同时按顺序打印数值，然后再从临时堆栈中把所有数值倒回主栈。

```
⟨cases 329⟩+≡
  case 'f':
      if (!Stack_empty(sp)) {
          Stack_T tmp = Stack_new();
          while (!Stack_empty(sp)) {
          AP_T x = pop();
          Fmt_print("%D\n", x);
          Stack_push(tmp, x);
      }
      while (!Stack_empty(tmp))
          Stack_push(sp, Stack_pop(tmp));
      Stack_free(&tmp);
  }
  break;
```

332

剩下的就是数值取反、清空堆栈并退出：

```
⟨cases 329⟩+≡
  case '~': {
      AP_T x = pop();
      Stack_push(sp, AP_neg(x));
      AP_free(&x);
      break;
  }
  case 'c': ⟨clear the stack 333⟩ break;
  case 'q': ⟨clean up and exit 329⟩
```

```
⟨clear the stack 333⟩≡
  while (!Stack_empty(sp)) {
      AP_T x = Stack_pop(sp);
      AP_free(&x);
  }
```

calc清除堆栈的时候会释放堆叠的那些AP_T以避免生成不能达到且永远无法释放所占内存的对象。

calc的最后部分代码将一串一个或多个数字读入buf：

```
⟨gather up digits into buf 333⟩≡
  {
      int i = 0;
      for ( ; c != EOF && isdigit(c); c = getchar(), i++)
          if (i < (int)sizeof (buf) - 1)
              buf[i] = c;
      if (i > (int)sizeof (buf) - 1) {
          i = (int)sizeof (buf) - 1;
```

333

```
        Fmt_fprint(stderr,
            "?integer constant exceeds %d digits\n", i);
    }
    buf[i] = 0;
    if (c != EOF)
        ungetc(c, stdin);
}
```

正如这段代码所示，calc声明数字过长截断了它们。

18.3 实现

AP接口的实现阐述了XP接口有代表性的使用。AP使用带符号数值表示有符号数：AP_T指向含有这个数的符号以及其绝对值的结构XP_T。

⟨*ap.c*⟩≡

```
#include <ctype.h>
#include <limits.h>
#include <stdlib.h>
#include <string.h>
#include "assert.h"
#include "ap.h"
#include "fmt.h"
#include "xp.h"
#include "mem.h"

#define T AP_T

struct T {
    int sign;
    int ndigits;
    int size;
    XP_T digits;
};

⟨macros 337⟩
⟨prototypes 336⟩
⟨static functions 335⟩
⟨functions 335⟩
```

334

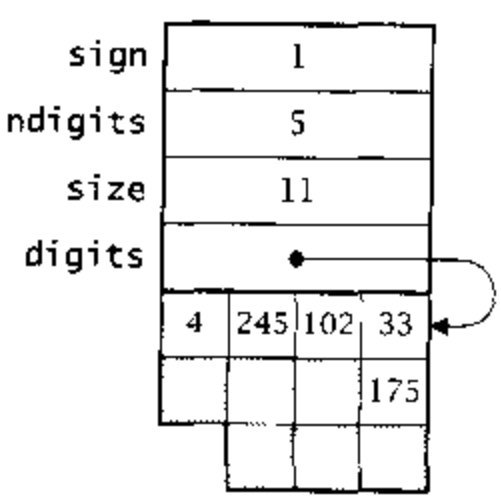


图18-1 Little endian上一个等于751,702,468,129的AP_T的小尾数法布局

sign为1或-1，size是分配给digits所指空间的位数，它可以大于使用中的位数ndigits。即，一个AP_T用digits[0..ndigits-1]表示XP_T给定的数。AP_T总是经过规格化的：它们最高有效数字位非零，除非这个数值就是零。因而，ndigits经常小于size。图18-1显示了一台小尾数法计算机上具有32位单词和8位字符的一个等于751 702 468 129的11位数AP_T。隐藏了digits数组中的未使用元素。

AP_T由如下函数进行分配

```
⟨functions 335⟩≡
  T AP_new(long int n) {
      return set(mk(sizeof (long int)), n);
  }
```

该函数调用静态函数mk执行实际的分配；mk会分配一个能够容纳size大小数字的AP_T，并将它初始化为0。

```
⟨static functions 335⟩≡
  static T mk(int size) {
      T z = CALLOC(1, sizeof (*z) + size);
      assert(size > 0);
      z->sign = 1;
      z->size = size;
      z->ndigits = 1;
      z->digits = (XP_T)(z + 1);
      return z;
  }
```

335

在带符号表示法中，零有两种表示方法；通常，AP只使用正数表示法，就像mk中代码那样。

AP_new调用静态函数set把AP_T初始化为一个长整型数值，而且，通常，set把最大的那个负长整型作为一种特例：

```
⟨static functions 335⟩+≡
  static T set(T z, long int n) {
      if (n == LONG_MIN)
          XP_fromint(z->size, z->digits, LONG_MAX + 1UL);
      else if (n < 0)
          XP_fromint(z->size, z->digits, -n);
      else
          XP_fromint(z->size, z->digits, n);
      z->sign = n < 0 ? -1 : 1;
      return normalize(z, z->size);
  }
```

z->sign的赋值是习惯用法，确保这个带符号数值为1或-1，且零的符号只有一个。XP_T是没有规格化的，因为它最高有效数字位可以是0。当AP函数产生一个可能没有规格化的XP_T的时候，它调用normalize计算正确的ndigits字段以进行调整：

```
⟨static functions 335⟩+≡
  static T normalize(T z, int n) {
      z->ndigits = XP_length(n, z->digits);
      return z;
  }

⟨prototypes 336⟩≡
  static T normalize(T z, int n);
```

336

通过如下函数可以释放AP_T：

```
⟨functions 335⟩+≡
  void AP_free(T *z) {
      assert(z && *z);
      FREE(*z);
  }
```

AP_new是可以分配AP_T的惟一途径，因此，对于AP_free，从安全方面而言，应“了解”这个结构和数字数组所占的空间是由单个分配方案来分配的。

18.3.1 取反和乘法

取反是需要实现的算法操作中最简单的，它说明了带符号数值表示的循环问题：

```
⟨functions 335⟩+≡
  T AP_neg(T x) {
      T z;

      assert(x);
      z = mk(x->ndigits);
      memcpy(z->digits, x->digits, x->ndigits);
      z->ndigits = x->ndigits;
      z->sign = iszero(z) ? 1 : -x->sign;
      return z;
  }

⟨macros 337⟩≡
  #define iszero(x) ((x)->ndigits==1 && (x)->digits[0]==0)
```

除了值为0，x取反都只是拷贝数值且翻转符号。宏iszero利用了AP_T是规格化的这一约束：零值只有一位。

x·y的值是|x|·|y|，而且它可能具有与x和y中数字总数同样多的数字。x和y符号相同或者x或y其一为零时结果为正，反之则为负。符号是-1或1，因此x和y符号相同时比较

337

```
⟨x and y have the same sign 338⟩≡
  ((x->sign^y->sign) == 0)
```

为真，反之为假。AP_mul调用XP_mul计算|x|·|y|并计算自身的符号：

```
⟨functions 335⟩+≡
  T AP_mul(T x, T y) {
      T z;
```

```
    assert(x);
    assert(y);
    z = mk(x->ndigits + y->ndigits);
    XP_mul(z->digits, x->ndigits, x->digits, y->ndigits,
        y->digits);
    normalize(z, z->size);
    z->sign = iszero(z)
        || ⟨x and y have the same sign 338⟩ ? 1 : -1;
    return z;
}
```

再次调用XP_mul计算z=z+x·y，而且mk将z初始化成一个规格化的零和一个未格式化的零。

18.3.2 加法和减法

加法更复杂一些，因为根据x与y的符号和值，加法可能还需要减法。下表归纳了加法的各种情况。

	y<0	y≥0
x<0	−(|x|+|y|)	y−|x| 如果y≥|x| (|x|−y) 如果y<|x|
x≥0	x−|y| 如果x>|y| −(|y|−x) 如果x≤|y|	x+y

338 如果x和y非负，|x|+|y|就等于x+y，因此，位于对角线上的情况都可以通过计算|x|+|y|，并将符号设成x的符号即可解决。结果的位数可能比x和y中最长的那个数的位数多1。

```
⟨functions 335⟩+≡
  T AP_add(T x, T y) {
      T z;

      assert(x);
      assert(y);
      if (⟨x and y have the same sign 338⟩) {
          z = add(mk(maxdigits(x,y) + 1), x, y);
          z->sign = iszero(z) ? 1 : x->sign;
      } else
          ⟨set z to x+y when x and y have different signs 340⟩
      return z;
  }

⟨macros 337⟩+≡
  #define maxdigits(x,y) ((x)->ndigits > (y)->ndigits ? \
      (x)->ndigits : (y)->ndigits)
```

add调用XP_add执行实际的加法：

```
⟨static functions 335⟩+≡
  static T add(T z, T x, T y) {
      int n = y->ndigits;

      if (x->ndigits < n)
          return add(z, y, x);
      else if (x->ndigits > n) {
          int carry = XP_add(n, z->digits, x->digits,
              y->digits, 0);
          z->digits[z->size-1] = XP_sum(x->ndigits - n,
              &z->digits[n], &x->digits[n], carry);
      } else
          z->digits[n] = XP_add(n, z->digits, x->digits,
              y->digits, 0);
      return normalize(z, z->size);
  }
```

339

add中的第一个比较测试确保x为较长的操作数。如果x比y长，XP_add计算z->digits[0..n-1]中的n位总和并返回carry，carry与jx->digits[n..x->ndigits-1]的和成为z->digits[n..z->size-1]。如果x与y位数相同，XP_add就像前面那样计算n位数总和，而且，carry就是z最高有效位。

加法的另一种情况也可以简化。x<0、y≥0且|x|>|y|时，x+y的值就是|x|-|y|。符号为负。x≥0、y<0且|x|>|y|时，x+y的值也是|x|-|y|，但是符号为正。这两种情况结果的符号都与x的符号相同。下面介绍的sub执行的是减法；cmp则比较|x|和|y|。其结果可能与x的位数相同。

```
⟨set z to x+y when x and y have different signs 340⟩≡
  if (cmp(x, y) > 0) {
      z = sub(mk(x->ndigits), x, y);
      z->sign = iszero(z) ? 1 : x->sign;
  }
```

x<0、y≥0且|x|≤|y|时，x+y的值就是|y|-|x|，符号为正；x≥0、y<0且|x|≤|y|时，x+y的值也是|y|-|x|，但符号为负。两种情况下，结果的符号都与x的符号相反，而且结果可能与y的位数相同。

```
⟨set z to x+y when x and y have different signs 340⟩+≡
  else {
      z = sub(mk(y->ndigits), y, x);
      z->sign = iszero(z) ? 1 : -x->sign;
  }
```

减法可以从类似的分析中获益。下表列出了各种情况。

	y<0	y≥0
x<0	-(|x|-|y|) 如果 |x| > |y| |y|-|x| 如果 |x|≤|y|	-(|x|+y)
x≥0	x+|y|	x-y 如果 x>y -(y-x) 如果x≤y

340

这里，非对角线上的情况比较容易处理。两者都可以通过计算|x|+|y|并将结果的符号设为x的符号得到结果。

```
⟨functions 335⟩+≡
  T AP_sub(T x, T y) {
      T z;

      assert(x);
      assert(y);
      if (!⟨x and y have the same sign 338⟩) {
          z = add(mk(maxdigits(x,y) + 1), x, y);
          z->sign = iszero(z) ? 1 : x->sign;
      } else
          ⟨set z to x-y when x and y have the same sign 341⟩
      return z;
  }
```

对角线上的情况取决于x和y的相对值。|x|>|y|时，x-y的值就是|x|-|y|，符号与x相同；如果|x|≤|y|，那么x-y的值就是|y|-|x|，符号与x相反。

```
⟨set z to x-y when x and y have the same sign 341⟩≡
  if (cmp(x, y) > 0) {
      z = sub(mk(x->ndigits), x, y);
      z->sign = iszero(z) ? 1 : x->sign;
  } else {
      z = sub(mk(y->ndigits), y, x);
      z->sign = iszero(z) ? 1 : -x->sign;
  }
```

与add一样，sub调用XP函数实现减法；y永不大于x。

```
⟨static functions 335⟩+≡
  static T sub(T z, T x, T y) {
      int borrow, n = y->ndigits;

      borrow = XP_sub(n, z->digits, x->digits,
          y->digits, 0);
341   if (x->ndigits > n)
          borrow = XP_diff(x->ndigits - n, &z->digits[n],
              &x->digits[n], borrow);
      assert(borrow == 0);
      return normalize(z, z->size);
  }
```

如果x长于y，调用XP_sub计算z->digits[0..n-1]并返回borrow。这个borrow和x->digits[n..x->ndigits-1]之间的差就是z->digits[n..x->ndigits-1]。最后的borrow是0，因为所有sub的调用中|x|≥|y|。如果x和y的位数相同，XP_sub就像前面那样计算这n位数的差，但是没有borrow传递。

18.3.3 除法

除法与乘法一样，只是因截取规则而变得复杂。如果x和y符号相同，商即为|x|/|y|，且

为正，余数为|x| mod |y|；如果x和y符号相反，则商为负，且如果|x| mod |y|为零，则商取值|x|/|y|，如果|x| mod |y|非零，则商取值|x|/|y|+1。如果值为零，余数就是|x| mod |y|；如果|x| mod |y|非零，则余数为|y|-(|x| mod |y|)。余数总为正。商和余数分别与x和y的位数相同。

```
⟨functions 335⟩+≡
  T AP_div(T x, T y) {
      T q, r;

      ⟨q ← x/y, r ← x mod y 343⟩
      if (!⟨x and y have the same sign 338⟩ && !iszero(r)) {
          int carry = XP_sum(q->size, q->digits,
              q->digits, 1);
          assert(carry == 0);
          normalize(q, q->size);
      }
      AP_free(&r);
      return q;
  }
```

342

```
⟨q ← x/y, r ← x mod y 343⟩≡
  assert(x);
  assert(y);
  assert(!iszero(y));
  q = mk(x->ndigits);
  r = mk(y->ndigits);
  {
      XP_T tmp = ALLOC(x->ndigits + y->ndigits + 2);
      XP_div(x->ndigits, q->digits, x->digits,
          y->ndigits, y->digits, r->digits, tmp);
      FREE(tmp);
  }
  normalize(q, q->size);
  normalize(r, r->size);
  q->sign = iszero(q)
      || ⟨x and y have the same sign 338⟩ ? 1 : -1;
```

x和y的符号不同时，AP_div不影响余数调整，因为它舍弃了余数。AP_mod则刚好相反：它仅调整余数而舍弃商。

```
⟨functions 335⟩+≡
  T AP_mod(T x, T y) {
      T q, r;

      ⟨q ← x/y, r ← x mod y 343⟩
      if (!⟨x and y have the same sign 338⟩ && !iszero(r)) {
          int borrow = XP_sub(r->size, r->digits,
              y->digits, r->digits, 0);
          assert(borrow == 0);
          normalize(r, r->size);
      }
      AP_free(&q);
      return r;
  }
```

18.3.4 求幂

343 第三个参数p为空时，AP_pow返回x^y。如果p非空，AP_pow返回（x^y）mod p。

```
⟨functions 335⟩+≡
  T AP_pow(T x, T y, T p) {
      T z;

      assert(x);
      assert(y);
      assert(y->sign == 1);
      assert(!p || p->sign==1 && !iszero(p) && !isone(p));
      ⟨special cases 344⟩
      if (p)
          ⟨z ← x^y mod p 346⟩
      else
          ⟨z ← x^y 345⟩
      return z;
  }
```

```
⟨macros 337⟩+≡
  #define isone(x) ((x)->ndigits==1 && (x)->digits[0]==1)
```

为计算$z=x^y$，可以把z设置为1再乘以x，乘y次。问题是如果y比较大，比方是200位十进制数，这个过程花费的时间就会比宇宙存在的年代还要长。一些数学规则可以简化这个计算：

$$z = \begin{cases} (x^{y/2})^2 = (x^{y/2})(x^{y/2}) & \text{如果} x \text{为偶数} \\ x \cdot x^{y-1} = (x^{y/2})(x^{y/2})x & \text{如果} x \text{不为偶数} \end{cases}$$

这些规则允许递归调用AP_pow、结果相乘及乘方来计算x^y。递归的深度（以及由此运行的次数）与lg y成比例。x或y为0或1时，由于$0^y=0$、$1^y=1$、$x^0=1$及$x^1=x$，所以递归到达最低点开始返回。这些特殊情况中的前三种可以这样处理：

```
⟨special cases 344⟩≡
  if (iszero(x))
      return AP_new(0);
  if (iszero(y))
      return AP_new(1);
  if (isone(x))
      return AP_new(⟨y is even 345⟩ ? 1 : x->sign);
```

344

```
⟨y is even 345⟩≡
  (((y)->digits[0]&1) == 0)
```

递归实现了第四种特殊情况以及上述等式描述的两种情况：

```
⟨z ← x^y 345⟩≡
  if (isone(y))
      z = AP_addi(x, 0);
  else {
      T y2 = AP_rshift(y, 1), t = AP_pow(x, y2, NULL);
      z = AP_mul(t, t);
```

```
        AP_free(&y2);
        AP_free(&t);
        if (!⟨y is even 345⟩) {
            z = AP_mul(x, t = z);
            AP_free(&t);
        }
    }
```

y为正，因此右移一位计算y/2。中间结果——y/2，$x^{y/2}$以及（$x^{y/2}$）（$x^{y/2}$）——就会被释放以避免产生无法访问的存储器区域。

p非空时，AP_pow计算x^y mod p。如果p>1，实际上我们不能计算x^y，因为它可能太大；举个例子，如果x是个10位十进制数且y为200，那么x^y的位数比宇宙中的原子还要多；但x^y mod p是个很小的数。下列关于模数乘法的数学规则可用于避免生成的数太大：

$$(x \cdot y) \bmod p = ((x \bmod p) \cdot (y \bmod p)) \bmod p.$$

AP_mod及静态函数mulmod共同实现了此规则。mulmod 使用AP_mod和AP_mul实现$x \cdot y$ mod p，请注意，要释放临时结果$x \cdot y$。

```
⟨static functions 335⟩+≡
  static T mulmod(T x, T y, T p) {
      T z, xy = AP_mul(x, y);
      z = AP_mod(xy, p);
      AP_free(&xy);
      return z;
  }
```

345

p非空时，AP_pow代码除了mulmod用于乘法、p传递给递归调用AP_pow以及y为奇数时x则减到 mod p，几乎与p为空时的较简单情况相同。

```
⟨z ← x^y mod p 346⟩≡
  if (isone(y))
      z = AP_mod(x, p);
  else {
      T y2 = AP_rshift(y, 1), t = AP_pow(x, y2, p);
      z = mulmod(t, t, p);
      AP_free(&y2);
      AP_free(&t);
      if (!⟨y is even 345⟩) {
          z = mulmod(y2 = AP_mod(x, p), t = z, p);
          AP_free(&y2);
          AP_free(&t);
      }
  }
```

18.3.5 比较

x和y的比较结果取决于它们的符号和数值。x<y、x=y或x>y时，AP_cmp返回一个小于0、

等于0或大于0的值。如果x与y符号不同，AP_cmp可以仅返回x的符号；否则，必须比较它们的数值：

```
⟨functions 335⟩+≡
  int AP_cmp(T x, T y) {
      assert(x);
      assert(y);
      if (!⟨x and y have the same sign 338⟩)
          return x->sign;
      else if (x->sign == 1)
346       return cmp(x, y);
      else
          return cmp(y, x);
  }
```

x和y都为正时，如果|x|<|y|，则x<y．诸如此类。然而，x和y都为负时，如果|x|>|y|，则x<y，第二次调用cmp，参数顺序都颠倒过来了。cmp核对不同长度的操作数之后，XP_cmp执行实际的比较。

```
⟨static functions 335⟩+≡
  static int cmp(T x, T y) {
      if (x->ndigits != y->ndigits)
          return x->ndigits - y->ndigits;
      else
          return XP_cmp(x->ndigits, x->digits, y->digits);
  }

⟨prototypes 336⟩+≡
  static int cmp(T x, T y);
```

18.3.6 简易函数

这6个简易函数把AP_T作为第一个参数，把一个有符号长整型作为第二个参数。每个函数都传递给set一个长整型来初始化一个临时的AP_T，然后调用一些更常用的操作。AP_addi说明了这种方法。

```
⟨functions 335⟩+≡
  T AP_addi(T x, long int y) {
      ⟨declare and initialize t 347⟩
      return AP_add(x, set(&t, y));
  }

⟨declare and initialize t 347⟩≡
  unsigned char d[sizeof (unsigned long)];
  struct T t;
  t.size = sizeof d;
  t.digits = d;
```

347 上述第二段代码声明了适当的局部变量，在栈中分配了临时的AP_T及其相关的digits数组，

这是因为digits数组的大小受限于无符号长整型数的字节数。

其余四个简易函数模式相同：

```
⟨functions⟩+≡
  T AP_subi(T x, long int y) {
      ⟨declare and initialize t 347⟩
      return AP_sub(x, set(&t, y));
  }

  T AP_muli(T x, long int y) {
      ⟨declare and initialize t 347⟩
      return AP_mul(x, set(&t, y));
  }

  T AP_divi(T x, long int y) {
      ⟨declare and initialize t 347⟩
      return AP_div(x, set(&t, y));
  }

  int AP_cmpi(T x, long int y) {
      ⟨declare and initialize t 347⟩
      return AP_cmp(x, set(&t, y));
  }
```

AP_modi比较古怪，因为它返回一个长整型，而不是一个AP_T或整型，而且它必须放弃AP_mod返回的AP_T。

```
⟨functions 335⟩+≡
  long int AP_modi(T x, long int y) {
      long int rem;
      T r;

      ⟨declare and initialize t 347⟩
      r = AP_mod(x, set(&t, y));
      rem = XP_toint(r->ndigits, r->digits);
      AP_free(&r);
      return rem;
  }
```

348

18.3.7 移位

两个移位函数都调用它们的XP相关函数移动它们的操作数。对于AP_lshift，结果比操作数多$\lceil s/8 \rceil$位数，且与操作数符号相同。

```
⟨functions 335⟩+≡
  T AP_lshift(T x, int s) {
      T z;

      assert(x);
      assert(s >= 0);
      z = mk(x->ndigits + ((s+7)&~7)/8);
```

```
        XP_lshift(z->size, z->digits, x->ndigits,
            x->digits, s, 0);
        z->sign = x->sign;
        return normalize(z, z->size);
    }
```

对于AP_rshift，结果字节数减少$\lfloor s/8 \rfloor$，而且可能结果为零，这种情况下它的符号必须为正。

```
T AP_rshift(T x, int s) {
    assert(x);
    assert(s >= 0);
    if (s >= 8*x->ndigits)
        return AP_new(0);
    else {
        T z = mk(x->ndigits - s/8);
        XP_rshift(z->size, z->digits, x->ndigits,
            x->digits, s, 0);
        normalize(z, z->size);
        z->sign = iszero(z) ? 1 : x->sign;
        return z;
    }
}
```

349 if语句来处理s指定的移位位数大于或等于x的位数的情况。

18.3.8 字符串和整数转换

AP_toint（x）返回一个与x符号相同、值等于|x| mod (LONG_MAX+1)的长整型。

```
⟨functions 335⟩+≡
  long int AP_toint(T x) {
      unsigned long u;

      assert(x);
      u = XP_toint(x->ndigits, x->digits)%(LONG_MAX + 1UL);
      if (x->sign == -1)
          return -(long)u;
      else
          return  (long)u;
  }
```

其他的AP函数把AP_Ts转换成字符串或把字符串转换成AP_T。AP_fromstr把一个字符串转换成AP_T；它接收一个具有如下语法的有符号数：

number = { *white* } [- | +] { *white* } *digit* { *digit* }

这里white表示空格字符，digit是特定底数下的一个数字字符，必须在2到36之间。对于超出10的底数，用字母说明那些大于9的数字。AP_fromstr调用XP_fromstr，而且遇到非法字符或空字符时就停止扫描它的字符串参数。

```
⟨functions 335⟩+≡
  T AP_fromstr(const char *str, int base, char **end) {
```

```
    T z;
    const char *p = str;
    char *endp, sign = '\0';
    int carry;

    assert(p);
    assert(base >= 2 && base <= 36);
    while (*p && isspace(*p))
        p++;
    if (*p == '-' || *p == '+')                                    350
        sign = *p++;
    ⟨z ← 0 351⟩
    carry = XP_fromstr(z->size, z->digits, p,
        base, &endp);
    assert(carry == 0);
    normalize(z, z->size);
    if (endp == p) {
        endp = (char *)str;
        z = AP_new(0);
    } else
        z->sign = iszero(z) || sign != '-' ? 1 : -1;
    if (end)
        *end = (char *)endp;
    return z;
}
```

AP_fromstr把endp的地址传递给XP_fromstr，因为它需要知道是什么终止了扫描，这样才能检查非法输入。如果end非空，AP_fromstr就把*end设置成endp。

z中的位数是n · lg base，这里n是字符串中数字个数，因而z的XP_T必须有一个至少m=(n · lg base)/8字节的digits数组。假设base是2^k；m=n · lg (2^k)/8=k · n/8。这样，如果我们选择k，使得2^k为等于或大于base二次幂的最小，那么z需要[k · n/8]位数。k是对以base为底数的每个数字位数的保守估计。例如，base为10时，每个数字带有lg 10 ≈3.32位，k为4。base为2时，k从1开始；base为36时k达到了6。

```
⟨z ← 0 351⟩≡
  {
      const char *start;
      int k, n = 0;
      for ( ; *p == '0' && p[1] == '0'; p++)
          ;
      start = p;
      for ( ; ⟨*p is a digit in base 352⟩; p++)
          n++;
      for (k = 1; (1<<k) < base; k++)
          ;
      z = mk(((k*n + 7)&~7)/8);                                    351
      p = start;
  }
```

```
⟨*p is a digit in base 352⟩≡
  (  '0' <= *p && *p <= '9' && *p < '0' + base
  || 'a' <= *p && *p <= 'z' && *p < 'a' + base - 10
  || 'A' <= *p && *p <= 'Z' && *p < 'A' + base - 10)
```

<z←0 351>中第一个循环略过了前面的零

AP_tostr可以使用类似技巧通近以字符串形式表示base中的x时，所需的字符个数n。x的digits数组中数字的个数是m=(n · lg base)/8。如果选择k使2^k为小于或等于base的两个指数中的最大指数，那么m=n · lg (2^k)/8=k · n/8，且n为$\lceil 8 \cdot m/k \rceil$。由于有终止null字符还要加1。这里，k向下低估每个数字以base为底数进行表示的bit数，这样n就是所需数字位数的一个保守估计。例如，base为10时，x中每个数字都会生成8/lg 10≈2.41十进制数位，且k为3，，因此x的每个数字都分配了$\lceil 8/3 \rceil = 3$个十进制数位的空间。base为36时k从5开始；base为2则达到1。

```
⟨size ← number of characters in str 352⟩≡
  {
      int k;
      for (k = 5; (1<<k) > base; k--)
          ;
      size = (8*x->ndigits)/k + 1 + 1;
      if (x->sign == 1)
          size++;
  }
```

AP_tostr让XP_tostr计算x的字符表示：

```
⟨functions 335⟩+≡
  char *AP_tostr(char *str, int size, int base, T x) {
      XP_T q;

      assert(x);
      assert(base >= 2 && base <= 36);
      assert(str == NULL || size > 1);
352   if (str == NULL) {
          ⟨size ← number of characters in str 352⟩
          str = ALLOC(size);
      }
      q = ALLOC(x->ndigits);
      memcpy(q, x->digits, x->ndigits);
      if (x->sign == -1) {
          str[0] = '-';
          XP_tostr(str + 1, size - 1, base, x->ndigits, q);
      } else
          XP_tostr(str, size, base, x->ndigits, q);
      FREE(q);
      return str;
  }
```

最后一个AP函数是AP_fmt，这是用于打印AP_T的Fmt样式转换函数。它使用AP_tostr以十进制格式化数值，并调用Fmt_putd进行打印。

```
⟨functions 335⟩+≡
  void AP_fmt(int code, va_list *app,
       int put(int c, void *cl), void *cl,
       unsigned char flags[], int width, int precision) {
       T x;
       char *buf;

       assert(app && flags);
       x = va_arg(*app, T);
       assert(x);
       buf = AP_tostr(NULL, 0, 10, x);
       Fmt_putd(buf, strlen(buf), put, cl, flags,
           width, precision);
       FREE(buf);
  }
```

参考书目浅析

AP_T类似于一些编程语言中的“大数”。例如，最近版本的Icon只有一种整数类型，但可以根据需要，使用任意精度的算法来表示计算得到的值。程序员不需要区分机器整数和任意精度整数。 353

任意精度算法的工具经常作为一个标准库或包提供。例如，LISP系统有包括大数包的长整型，还有类似的ML包。

大多数符号算法系统都执行任意精度算法，因为那就是它们的目的。例如，Mathematica（Wolfram 1988）提供任意长度的整数和有理数，且有理数的分子和分母都是任意精度的整数。另一个符号计算系统Maple V（Char etal. 1992）也有类似的工具。

练习

18.1 每次调用AP_div和AP_mod的时候，它们都会分配释放临时空间。修改这两个函数，让它们共享tmp，只分配一次tmp，跟踪它的大小，并根据需要进行扩充。

18.2 有一种与使用重复乘方和相乘计算$z = x^y$相类似的迭代算法（参见Knuth 1981的4.6.3节），AP_pow中使用的递归算法等价：

$z \leftarrow x, u \leftarrow 1$
while $y > 1$ do
  if y is odd then $u \leftarrow u \cdot z$
  $z \leftarrow z^2$
  $y \leftarrow y/2$
$z \leftarrow u \cdot z$

迭代通常比递归快一些，但是这种方法的真正优点是它为中间结果值分配的空间更少一些。使用这种算法重新实现AP_pow，并度量改进的时间和空间。要明显地看出这个算法优于递归算法，x和y必须取多大值？

18.3 实现AP_ceil（AP_T x，AP_T y）及AP_floor（AP_T x，AP_T y），返回x/y的最高限度和最低限度。一定要说明x和y符号不同时的情况。

18.4 AP接口非常“嘈杂”——有大量的参数，而且很容易弄混输入和输出参数。设计
354 并实现一个新的接口，将Seq_T用作堆栈，函数都从中取出参数、存储结果。特别注意尽可能地使接口整洁，但不要遗漏重要的功能。

18.5 实现一个AP函数生成随机数，均匀分布于特定范围。

18.6 设计一个接口，其功能实现以任意n为模的计算，并由此接收和返回0到$n-1$之间整数集合中的值。要注意除法：仅当该集合为有限域，即n为素数时才有定义。

18.7 两个n位数相乘要花费的时间与n^2成比例（参见17.2.2节）。A.Karatsuba（在1962年）说明了如何实现乘法时间与$n^{1.58}$成比例（参见Geddes、Czapor与Labahn 1992的4.3节以及Knuth 1981的4.3.3节）。一个n位数x可以分成最高有效及最低有效的$n/2$位数之和；即$x = aB^{n/2}+b$。因此，乘积xy可以写作

$$xy = (aB^{n/2} + b)(cB^{n/2} + d) = acB^n + (ad + bc)B^{n/2} + bd,$$

需要四次乘法和一次加法。中间项的系数可以重新写成：

$$ad + bc = ac + bd + (a - b)(d - c).$$

乘积xy仅需要三次乘法（ac、bd及$(a-b)(d-c)$）、两次减法及两次加法。如果n很大，可以中间结果占用的空间为代价，保存一个$n/2$位乘法，来减少乘法的执行时
355 间。使用Karatsuba的算法实现一个递归的AP_mul，确定n为何值时会明显快于以前那个通用算法。使用XP_mul完成中间计算。

第19章　多精度算法

这三种精度算法接口的最后一种，MP，导出了用于实现无符号整数和二进制补码整数多精度算法的函数。与XP相同，MP展现了它对于n位整数的表示形式，且MP函数处理的是给定大小的整数。与XP不同的地方在于MP整数的长度是以位（bit）给出的，而且MP的函数既实现了带符号算法也实现了无符号算法。和AP一样，MP函数推行了一组常用的可检查的运行期错误。

MP计划用于需要扩展精度算法但想灵活控制内存分配、或者既需要无符号操作又需要有符号操作、抑或必须模拟二进制补码的n位数算法的应用程序。范例包括使用加密的编译器和应用程序。一些现代加密算法涉及到了数百位固定精度整数的处理。

一些编译器必须使用多精度整数。交叉编译器运行在*X*平台并在*Y*平台生成代码。如果*Y*的整数比*X*大，编译器就会使用MP处理*Y*大小的整数。同样，编译器必须使用多精度算法把浮点常数转换成与它们所指定的最接近的浮点值。

19.1　接口

MP接口内容很多——包括49个函数及两个异常——因为它导出了n位有符号整数及无符号整数的一组完整算法函数。

```
⟨mp.h⟩≡
  #ifndef MP_INCLUDED
  #define MP_INCLUDED
  #include <stdarg.h>
  #include <stddef.h>
  #include "except.h"

  #define T MP_T
  typedef unsigned char *T;

  ⟨exported exceptions 359⟩
  ⟨exported functions 358⟩

  #undef T
  #endif
```

与XP相同，MP表明*n*位整数需要用$\lceil n/8 \rceil$字节来表示，这些字节首先存储最低有效字节。MP使用二进制补码形式表示有符号整数，第$n-1$位是符号位。

与XP不同的是，MP函数实现了常用的可检查的运行期错误；例如，向该接口的任一函数传递空MP_T即为一个可检查的运行期错误。但是，如果传递的MP_T太小而不能容纳*n*位

整数，这就是个不可检查的运行期错误。

MP自动初始化以在32位整数上执行算法，调用：

```
⟨exported functions 358⟩≡
  extern int MP_set(int n);
```

修改MP，这样可以让后面的调用执行n位算法。MP_set返回前面的大小，如果n小于2则为可检查的运行期错误。一经初始化，大多数应用程序都使用只有一种大小的扩展整数。举个
358 例子，交叉编译器可能会使用128位的算法处理常数。这种设计迎合这些类型的应用程序；它简化了其他MP函数的使用，还简化了它们的参数列表。省略n就是一个非常显而易见的简化，但是更重要的一个简化是，对源参数和目的参数都没有了限制：相同的MP_T可以一直作为源和目的出现。由于一些函数所需要的临时空间仅依赖于n，这样可以由MP_set一次分配，因此消除这些限制成为现实。

这种设计也避免了内存分配。MP_set可以引发Mem_Failed，但其他48个MP函数中仅有4个执行内存分配。其中之一是

```
⟨exported functions 358⟩+≡
  extern T MP_new(unsigned long u);
```

这个函数分配一个适当大小的MP_T，并初始化为u，然后返回。

```
⟨exported functions 358⟩+≡
  extern T MP_fromint (T z, long v);
  extern T MP_fromintu(T z, unsigned long u);
```

把z设置成v或u然后返回z。如果n位数u或v不匹配，MP_new、MP_fromint及MP_fromintu就会引发

```
⟨exported exceptions 359⟩≡
  extern const Except_T MP_Overflow;
```

u超出2^n-1时，MP_new和MP_fromintu引发MP_Overflow；v小于-2^{n-1}或大于$2^{n-1}-1$时，MP_fromint引发MP_Overflow。

所有的MP函数都在引发异常之前计算它们的结果，只不过是抛弃了附加位。例如，

```
MP_T z;
MP_set(8);
z = MP_new(0);
MP_fromintu(z, 0xFFF);
```

把z设置为0xFF并引发MP_Overflow。客户调用程序可以在适当的时候，用一个TRY-
359 EXCEPT语句忽略这个异常。例如，

```
MP_T z;
MP_set(8);
z = MP_new(0);
TRY
    MP_fromintu(z, 0xFFF);
EXCEPT(MP_Overflow) ;
END_TRY;
```

把z设为0xFF并丢弃了这个溢出异常。

这个转换并不适用于

```
⟨exported functions 358⟩+≡
  extern unsigned long MP_tointu(T x);
  extern          long MP_toint (T x);
```

这个函数将x的值作为有符号长整数或无符号长整数返回。x不符合返回类型时这些函数，会引发MP_Overflow，而且异常产生时不能捕捉结果。客户调用程序可以使用

```
⟨exported functions 358⟩+≡
  extern T MP_cvt (int m, T z, T x);
  extern T MP_cvtu(int m, T z, T x);
```

把x转换成适当大小的MP_T。MP_cvt和MP_cvtu把x转换成一个m位有符号或无符号MP_T并存于z，然后返回z。m位目的单元格不匹配x时，这些函数会引发MP_Overflow，但是之前会设置z。因而

```
unsigned char z[sizeof (unsigned)];
TRY
    MP_cvtu(8*sizeof (unsigned), z, x);
EXCEPT(MP_Overflow) ;
END_TRY;
```

把z设置成x的8 · sizeof(unsigned)位最低有效数，而不管x的大小。

m超出x的位数时，MP_cvtu用0扩展结果，MP_cvt用x的符号位扩展结果。m小于2即为一个可检查的运行期错误；如果z太小不能容纳m位的整数，就是一个不可检查的运行期错误。

算法函数是

360

```
⟨exported functions 358⟩+≡
  extern T MP_add (T z, T x, T y);
  extern T MP_sub (T z, T x, T y);
  extern T MP_mul (T z, T x, T y);
  extern T MP_div (T z, T x, T y);
  extern T MP_mod (T z, T x, T y);
  extern T MP_neg (T z, T x);

  extern T MP_addu(T z, T x, T y);
  extern T MP_subu(T z, T x, T y);
  extern T MP_mulu(T z, T x, T y);
  extern T MP_divu(T z, T x, T y);
  extern T MP_modu(T z, T x, T y);
```

名字以u结束的那些函数完成无符号算法；其他完成二进制有符号算法。无符号操作与有符号操作的惟一区别就是溢出语义。下面将详细介绍。MP_add、MP_sub、MP_mul、MP_div、MP_mod以及相应的无符号函数分别计算$z = x+y$、$z = x-y$、$z = x \cdot y$、$z = x/y$及$z = x \bmod y$，并返回z。斜体表示x、y及z的值。MP_neg把z设成x的负数，然后返回z。如果x和y符号不同，MP_div和MP_mod向负无穷取整；这样x mod y就总为正。

如果结果不匹配，所有这些函数，除了MP_divu和MP_modu，都会引发MP_Overflow。x<y时MP_subu引发MP_Overflow；x和y符号不同且结果的符号与x的符号不同时MP_sub引发MP_Overflow。y为0时MP_div、MP_divu、MP_mod及MP_modu引发

```
⟨exported exceptions 359⟩+≡
  extern const Except_T MP_Dividebyzero;
```

when *y* is zero.

```
⟨exported functions 358⟩+≡
  extern T MP_mul2u(T z, T x, T y);
  extern T MP_mul2 (T z, T x, T y);
```

361 返回双倍长度的乘积：它们都计算$z = x \cdot y$，因此z有2n位，然后返回z。这样，结果不会溢出。z太小而不能容纳2n位数时产生一个不可检查的运行期错误。请注意，既然z必须容纳2n位，它就不能由MP_new进行分配。

简易函数接收一个无符号长整型或整型立即数作为它们第二个操作数：

```
⟨exported functions 358⟩+≡
  extern T MP_addi (T z, T x, long y);
  extern T MP_subi (T z, T x, long y);
  extern T MP_muli (T z, T x, long y);
  extern T MP_divi (T z, T x, long y);

  extern T MP_addui(T z, T x, unsigned long y);
  extern T MP_subui(T z, T x, unsigned long y);
  extern T MP_mului(T z, T x, unsigned long y);
  extern T MP_divui(T z, T x, unsigned long y);

  extern          long MP_modi (T x,          long y);
  extern unsigned long MP_modui(T x, unsigned long y);
```

当这些函数的第二个操作数初始化为y时，就等价于它们相应的更通用的函数了，它们也引发类似的异常。例如，

```
MP_T z, x;
long y;
MP_muli(z, x, y);
```

等价于

```
MP_T z, x;
long y;
{
    MP_T t = MP_new(0);
    int overflow = 0;
    TRY
        MP_fromint(t, y);
    EXCEPT(MP_Overflow)
        overflow = 1;
```

```
    END_TRY;
    MP_mul(z, x, t);
    if (overflow)
        RAISE(MP_Overflow);
}
```

362

但是，简易函数不进行内存分配。请注意，如果y太大，这些简易函数，包括MP_divui和MP_modui，会引发MP_Overflow，但是它们在计算完z之后引发。

```
⟨exported functions 358⟩+≡
  extern int MP_cmp  (T x, T y);
  extern int MP_cmpi (T x, long y);

  extern int MP_cmpu (T x, T y);
  extern int MP_cmpui(T x, unsigned long y);
```

比较x和y，而且在$x<y$、$x=y$或$x>y$时依次返回小于0、等于0或大于0的一个值。MP_cmpi和MP_cmpui并不一定要y匹配MP_T；它们只是比较x和y。

下列函数把它们的输入MP_T作为n位的字符串：

```
⟨exported functions 358⟩+≡
  extern T MP_and (T z, T x, T y);
  extern T MP_or  (T z, T x, T y);
  extern T MP_xor (T z, T x, T y);
  extern T MP_not (T z, T x);

  extern T MP_andi(T z, T x, unsigned long y);
  extern T MP_ori (T z, T x, unsigned long y);
  extern T MP_xori(T z, T x, unsigned long y);
```

MP_and、MP_or、MP_xor及它们的直接对应函数将z设置成为x和y按位与、内含OR及排他OR的结果并返回z。MP_not把z设为x和二进制反码，并返回z。这些函数从不引发异常，且简易变量忽略y太大时通常会产生的溢出。例如，

```
MP_T z, x;
unsigned long y;
MP_andi(z, x, y);
```

等价于

363

```
MP_T z, x;
unsigned long y;
{
    MP_T t = MP_new(0);
    TRY
        MP_fromintu(t, y);
    EXCEPT(MP_Overflow) ;
    END_TRY;
    MP_and(z, x, t);
}
```

但是，这些函数中没有一个执行任何内存分配。

三个移位函数

```
⟨exported functions 358⟩+≡
  extern T MP_lshift(T z, T x, int s);
  extern T MP_rshift(T z, T x, int s);
  extern T MP_ashift(T z, T x, int s);
```

实现逻辑和算术移位。MP_lshift将左移s位的x赋予z；MP_rshift将右移s位的x赋予z。这两个函数都用0填充空出的位，然后返回z。MP_ashift跟MP_rshift一样，只是空出的位是用x的符号位填充的。s为负则产生一个可检查的运行期错误。

下列函数实现MP_T与字符串之间的转换。

```
⟨exported functions 358⟩+≡
  extern T    MP_fromstr(T z, const char *str,
      int base, char **end);
  extern char *MP_tostr  (char *str, int size,
      int base, T x);
  extern void  MP_fmt    (int code, va_list *app,
      int put(int c, void *cl), void *cl,
      unsigned char flags[], int width, int precision);
  extern void  MP_fmtu   (int code, va_list *app,
      int put(int c, void *cl), void *cl,
      unsigned char flags[], int width, int precision);
```

MP_fromstr把str中的字符串解释为一个以base为底数的无符号整数，并将这个整数赋予z，
364 然后返回z。它忽略前面的空格，base为一位或多位数。对于大于10的base，则用小写字母和大写字母说明9以外的数字。MP_fromstr和strtoul一样：如果end非空，MP_fromstr把*end设置为终止扫描的那个字符的地址。如果str指的不是一个有效整数，而end为空，MP_fromstr就把*end设置成str，然后返回空。如果str中的字符串所指定的整数太大，MP_fromstr就会引发MP_Overflow。str为空，或者base小于2或大于36都会产生可检查的运行期错误。

MP_tostr用一个以base为底数表示x的非终止字符串填充str[0..size-1]，并返回str。如果str为空，MP_tostr就忽略size，并分配必要的字符串；客户调用程序应当负责释放这个字符串。如果str非空、size太小而不能容纳这个非终止结果、或者base小于2或大于36，都会产生一个可检查的运行期错误。str为空时，MP_tostr会引发Mem_Failed。

MP_fmt和MP_fmtu是用于打印的Fmt样式转换函数。这两个函数都使用一个MP_T及一个base；MP_fmt使用与printf的%d转换的相同规则把一个有符号MP_T转换成一个字符串；MP_fmtu使用printf的%u转换规则转换无符号MP_T。两个函数都可以引发Mem_Failed。如果app或flags为空，则为一个可检查的运行期错误。

19.2 示例：另一计算器

mpcalc与calc一样，只是它要在n位整数上进行有符号和无符号计算。这个示例说明的是

MP接口的使用，与calc相同，mpcalc使用波兰后缀符号表示法：值推入堆栈顶端；操作符从栈中弹出它们的操作数并将结果推入堆栈。值是基于当前输入的底数的一个或多个相邻数字；操作符如下：

~	相反数	&	与
+	加法	|	按位或
-	减法	^	异或
*	乘法	<	左移
/	除法	>	右移
%	取余	!	取非
i	设置输入底数	o	设置输入底数
k	设置精度	c	清空堆栈
d	复制栈顶数值		
p	打印栈顶值		
q	退出		

365

空格字符用于分隔数值但本身会被忽略不记；其他字符都声明为未被识别的运算符。堆栈的大小仅受限于可用存储器，但是诊断程序会声明堆栈下溢。

命令nk，指定mpcalc处理的整数大小，其中n至少为2；默认为32。执行操作符k时堆栈必须为空。操作符i和o指定输入和输出底数；默认情况下这两个底数都是10。当输入底数大于10时，数值的前导数字必须在0与9之间。

如果输出底数是2、8或16，操作符+ 、=、 *、 / 与%就执行无符号算法，而且操作符p和f打印无符号数值。操作符~总是执行有符号算法，操作符& | ^ ! < 与> 总是将它们的操作数解释为无符号数。

溢出和零除数产生时mpcalc会进行声明。对于溢出，这种情况下的结果就是数值中最低n位有效数。而对于零除数，结果就是0.

mpcalc的整体结构很像calc的结构：解释输入、计算数值以及管理堆栈。

⟨*mpcalc.c*⟩≡

```
#include <ctype.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <limits.h>
#include "mem.h"
#include "seq.h"
#include "fmt.h"
#include "mp.h"

⟨mpcalc data 367⟩
⟨mpcalc functions 367⟩
```

正如seq.h的包含文件所表明的那样，mpcalc的堆栈使用了一个序列：

366

```
⟨mpcalc data 367⟩≡
  Seq_T sp;

⟨initialization 367⟩≡
  sp = Seq_new(0);
```

值都是调用Seq_addhi压入堆栈，再调用Seq_remhi弹出堆栈的。序列为空时mpcalc肯定不能调用Seq_remhi，因此它把所有的弹出操作封装在一个检查下溢的函数中：

```
⟨mpcalc functions 367⟩≡
  MP_T pop(void) {
      if (Seq_length(sp) > 0)
          return Seq_remhi(sp);
      else {
          Fmt_fprint(stderr, "?stack underflow\n");
          return MP_new(0);
      }
  }
```

与calc的pop一样，mpcalc的pop也总是返回一个MP_T，即使堆栈为空，因为这样简化了错误检查。

因为要处理MP的异常，mpcalc的主循环变得比calc的主循环复杂一些。mpcalc的主循环与calc的主循环一样，都要读取下一数值或操作符，并根据读取内容进行相应操作。但是它还要为操作数和结果建立一些MP_T，而且还使用TRY-EXCEPT语句捕捉异常。

```
⟨mpcalc functions 367⟩+≡
  int main(int argc, char *argv[]) {
      int c;

      ⟨initialization 367⟩
      while ((c = getchar()) != EOF) {
          MP_T x = NULL, y = NULL, z = NULL;
          TRY
              switch (c) {
              ⟨cases 368⟩
              }
          EXCEPT(MP_Overflow)
              Fmt_fprint(stderr, "?overflow\n");
          EXCEPT(MP_Dividebyzero)
              Fmt_fprint(stderr, "?divide by 0\n");
          END_TRY;
          if (z)
              Seq_addhi(sp, z);
          FREE(x);
          FREE(y);
      }
      ⟨clean up and exit 368⟩
  }

⟨clean up and exit 368⟩≡
```

```
⟨clear the stack 368⟩
Seq_free(&sp);
return EXIT_SUCCESS;
```

x和y用作操作数；z用作结果。如果选中一个操作符之后x和y非空，它们就保存着从栈中弹出的操作数，因此必须释放x和y的内存空间。如果z非空，它就是保存着结果，这个结果必须压入堆栈。这种方法只允许出现一次TRY-EXCEPT语句，而不是在每个操作符的代码周围都可以出现。

输入字符可以是空格、数值的首位数字、一个操作符或其他错误的输入内容。下面是简单的情况：

```
⟨cases 368⟩≡
  default:
      if (isprint(c))
          Fmt_fprint(stderr, "?'%c'", c);
      else
          Fmt_fprint(stderr, "?'\\%03o'", c);
      Fmt_fprint(stderr, " is unimplemented\n");
      break;
  case ' ': case '\t': case '\n': case '\f': case '\r':
      break;
  case 'c': ⟨clear the stack 368⟩ break;
  case 'q': ⟨clean up and exit 368⟩
```

```
⟨clear the stack 368⟩≡
  while (Seq_length(sp) > 0) {
      MP_T x = Seq_remhi(sp);
      FREE(x);
  }
```

368

数值都是从一个数字开始；calc收集数字并调用MP_fromstr把这些数字转换成一个MP_T。ibase是当前的输入底数。

```
⟨cases 368⟩≡
  case '0': case '1': case '2': case '3': case '4':
  case '5': case '6': case '7': case '8': case '9': {
      char buf[512];
      z = MP_new(0);
      ⟨gather up digits into buf 369⟩
      MP_fromstr(z, buf, ibase, NULL);
      break;
  }
```

```
⟨gather up digits into buf 369⟩≡
  {
      int i = 0;
      for ( ; ⟨c is a digit in ibase 369⟩; c = getchar(), i++)
          if (i < (int)sizeof (buf) - 1)
              buf[i] = c;
```

```
        if (i > (int)sizeof (buf) - 1) {
            i = (int)sizeof (buf) - 1;
            Fmt_fprint(stderr,
                "?integer constant exceeds %d digits\n", i);
        }
        buf[i] = '\0';
        if (c != EOF)
            ungetc(c, stdin);
    }
```

太长的数值将被声明并截短。如果

```
⟨c is a digit in ibase 369⟩≡
    strchr(&"zyxwvutsrqponmlkjihgfedcba9876543210"[36-ibase],
        tolower(c))
```

369

非空，字符就是个以ibase为底数的数字。

大多数算法操作符的有相同的形式：

```
⟨cases 368⟩+≡
  case '+': ⟨pop x & y, set z 370⟩ (*f->add)(z, x, y); break;
  case '-': ⟨pop x & y, set z 370⟩ (*f->sub)(z, x, y); break;
  case '*': ⟨pop x & y, set z 370⟩ (*f->mul)(z, x, y); break;
  case '/': ⟨pop x & y, set z 370⟩ (*f->div)(z, x, y); break;
  case '%': ⟨pop x & y, set z 370⟩ (*f->mod)(z, x, y); break;
  case '&': ⟨pop x & y, set z 370⟩    MP_and(z, x, y); break;
  case '|': ⟨pop x & y, set z 370⟩    MP_or (z, x, y); break;
  case '^': ⟨pop x & y, set z 370⟩    MP_xor(z, x, y); break;

  case '!': z = pop(); MP_not(z, z); break;
  case '~': z = pop(); MP_neg(z, z); break;
```

```
⟨pop x & y, set z 370⟩≡
  y = pop(); x = pop();
  z = MP_new(0);
```

f指向一个包含了函数指针的结构，这些函数的操作都取决于mpcalc是执行有符号算法还是无符号算法。

```
⟨mpcalc data 367⟩+≡
  int ibase = 10;
  int obase = 10;
  struct {
      char *fmt;
      MP_T (*add)(MP_T, MP_T, MP_T);
      MP_T (*sub)(MP_T, MP_T, MP_T);
      MP_T (*mul)(MP_T, MP_T, MP_T);
      MP_T (*div)(MP_T, MP_T, MP_T);
      MP_T (*mod)(MP_T, MP_T, MP_T);
  } s = { "%D\n",
      MP_add,  MP_sub,  MP_mul,  MP_div,  MP_mod  },
    u = { "%U\n",
      MP_addu, MP_subu, MP_mulu, MP_divu, MP_modu },
   *f = &s;
```

370

obase是输出底数。最初，底数都是10，且f指向s，s保存着指向MP有符号算法函数的指针。操作符i修改ibase，操作符o修改obase；这两个操作符都让f重新指向u或者s：

```
⟨cases 369⟩+≡
  case 'i': case 'o': {
      long n;
      x = pop();
      n = MP_toint(x);
      if (n < 2 || n > 36)
          Fmt_fprint(stderr, "?%d is an illegal base\n",n);
      else if (c == 'i')
          ibase = n;
      else
          obase = n;
      if (obase == 2 || obase == 8 || obase == 16)
          f = &u;
      else
          f = &s;
      break;
      }
```

如果y不能转换成一个长整数（也就是说，如果MP_toint引发MP_Overflow），或者结果整数不是一个合法底数，就不能修改底数。

s和u结构也有一个Fmt样式格式的字符串，用于打印MP_T。mpcalc用%D注册MP_fmt，用%U注册MP_fmtu：

```
⟨initialization 367⟩+≡
  Fmt_register('D', MP_fmt);
  Fmt_register('U', MP_fmtu);
```

这样，f->fmt访问适当的格式字符串，操作符p和f用于打印MP_T。请注意，p弹出它的操作数并赋予z——主循环中的代码把那个数值再压回到栈顶。

```
⟨cases 369⟩+≡
  case 'p':
      Fmt_print(f->fmt, z = pop(), obase);
      break;
  case 'f': {
      int n = Seq_length(sp);
      while (--n > 0)
          Fmt_print(f->fmt, Seq_get(sp, n), obase);
      break;
  }
```

371

把f情况下的代码与18.2节中的calc的代码相比较。用Seq_T表示时，很容易打印堆栈的所有值。

移位操作符防止不合法的移位量，并按顺序移动它们的操作数：

```
⟨cases 369⟩+≡
  case '<': { ⟨get s & z 372⟩; MP_lshift(z, z, s); break; }
  case '>': { ⟨get s & z 372⟩; MP_rshift(z, z, s); break; }
```

```
⟨get s & z 372⟩≡
  long s;
  y = pop();
  z = pop();
  s = MP_toint(y);
  if (s < 0 || s > INT_MAX) {
      Fmt_fprint(stderr,
          "?%d is an illegal shift amount\n", s);
      break;
  }
```

如果MP_toint引发MP_Overflow，或者s为负或超出了最大的整型整数，就仅将操作数z压回栈顶。

其余的情况都关于操作符k和d：

```
⟨cases 369⟩+≡
  case 'k': {
      long n;
      x = pop();
      n = MP_toint(x);
      if (n < 2 || n > INT_MAX)
          Fmt_fprint(stderr,
              "?%d is an illegal precision\n", n);
      else if (Seq_length(sp) > 0)
          Fmt_fprint(stderr, "?nonempty stack\n");
      else
          MP_set(n);
      break;
      }
  case 'd': {
      MP_T x = pop();
      z = MP_new(0);
      Seq_addhi(sp, x);
      MP_addui(z, x, 0);
      break;
      }
```

372

同样，设置z会让主循环中的代码将其值压入堆栈。

19.3 实现

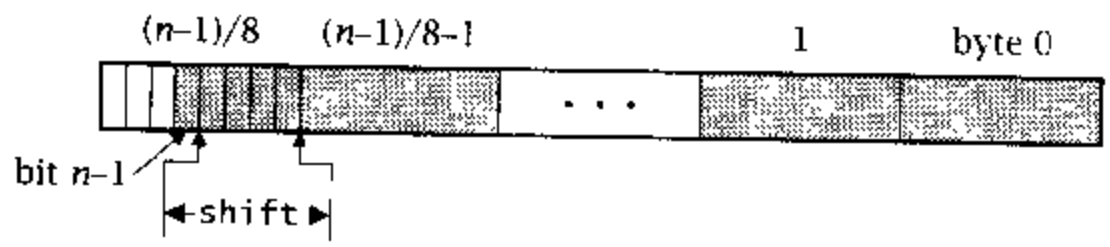
```
⟨mp.c⟩≡
  #include <ctype.h>
  #include <string.h>
  #include <stdio.h>
  #include <stdlib.h>
  #include <limits.h>
  #include "assert.h"
  #include "fmt.h"
  #include "mem.h"
```

```
#include "xp.h"
#include "mp.h"

#define T MP_T

⟨macros 374⟩
⟨data 373⟩
⟨static functions 389⟩
⟨functions 374⟩
```

```
⟨data 373⟩≡
  const Except_T MP_Dividebyzero = { "Division by zero" };
  const Except_T MP_Overflow     = { "Overflow" };
```

373

XP把一个n位数表示成$\lceil n/8 \rceil$=(n-1)/8+1字节，最低有效字节居先（n总为正）。下图说明了MP如何解释这些字节。最低有效字节在右边，地址从右向左递增。



符号位是第$n-1$位，也即$(n-1)/8$字节中的$(n-1) \bmod 8$位。给定n，MP除了把n保存为nbits，还要计算三个重要的数值：nbytes——保存n位所需的字节数；shift——最高有效字节必须右移的位数，以隔开符号位；msb——shift+1位的掩码，用于检测溢出。n为32时这些值是：

```
⟨data 373⟩+≡
  static int nbits  =  32;
  static int nbytes = (32-1)/8 + 1;
  static int shift  = (32-1)%8;
  static unsigned char msb = 0xFF;
```

如上所示，MP使用nbytes和shift访问符号位：

```
⟨macros 374⟩≡
  #define sign(x) ((x)[nbytes-1]>>shift)
```

MP_set修改这些值：

```
⟨functions 374⟩≡
  int MP_set(int n) {
      int prev = nbits;

      assert(n > 1);
      ⟨initialize 375⟩
      return prev;
  }
```

374

```
⟨initialize 375⟩≡
  nbits  = n;
```

```
nbytes = (n-1)/8 + 1;
shift  = (n-1)%8;
msb    = ones(n);
```

⟨*macros* 374⟩+≡
```
#define ones(n) (~(~0UL<<(((n)-1)%8+1)))
```

将~0取反，左移(n-1)%8-1位，形成跟随(n-1) mod 8 +1个0的掩码；取其补码在最低有效位中，生成(n-1) mod 8 + 1个1。如此定义是因为除了传递给MP_set的值，这些1还用于n的其他值。

和MP_div一样，MP_set也分配一些临时空间用于算法函数。在MP_set中只执行一次内存分配，而不是重复在算法函数中执行。MP_set为一个2 · nbyte+2大小的临时变量和一个nbyte大小的临时变量分配足够的空间。

⟨*data* 373⟩+≡
```
static unsigned char temp[16 + 16 + 16 + 2*16+2];
static T tmp[] = {temp, temp+1*16, temp+2*16, temp+3*16};
```

⟨*initialize* 375⟩+≡
```
if (tmp[0] != temp)
    FREE(tmp[0]);
if (nbytes <= 16)
    tmp[0] = temp;
else
    tmp[0] = ALLOC(3*nbytes + 2*nbytes + 2);
tmp[1] = tmp[0] + 1*nbytes;
tmp[2] = tmp[0] + 2*nbytes;
tmp[3] = tmp[0] + 3*nbytes;
```

nbytes没有超过16或n不超过128时，MP_set可以使用静态分配的temp。否则，它必须为临时变量分配足够的空间。temp是必需的，因为MP必须初始化，就好像已经执行过了MP_set（32）。

大多数MP函数调用XP函数执行nbyte数的实际算法，然后检查结果是否超出nbits位。MP_new和MP_ fromintu阐明了这个方案。

375

⟨*functions* 374⟩+≡
```
T MP_new(unsigned long u) {
    return MP_fromintu(ALLOC(nbytes), u);
}

T MP_fromintu(T z, unsigned long u) {
    unsigned long carry;

    assert(z);
    ⟨set z to u 376⟩
    ⟨test for unsigned overflow 376⟩
    return z;
}
```

⟨*set z to u* 376⟩≡
```
carry = XP_fromint(nbytes, z, u);
```

```
carry |= z[nbytes-1]&~msb;
z[nbytes-1] &= msb;
```

如果XP_fromint返回非零的carry，nbytes就装不下u；如果carry为零，nbytes就能装下。但是u可能无法匹配nbits位。MP_fromintu必须确保z的最高有效字节中的8-(shift+1)位最低有效位都是0。MP_set已经为msb做了准备以容纳有shift+1个1的掩码，因此~msb将那些在舍弃前与carry进行OR运算的期望位隔离开。无符号溢出测试仅仅检测carry：

```
⟨test for unsigned overflow 376⟩≡
  if (carry)
      RAISE(MP_Overflow);
```

请注意，MP_fromintu在检测溢出之前设置z；正如这个接口所指定的，所有MP函数必须在引发异常之前设置它们的结果。

检测有符号溢出更加复杂一点，因为它还依赖涉及到的操作。MP_fromint介绍了一个简单情况。

```
⟨functions 374⟩+≡
  T MP_fromint(T z, long v) {
      assert(z);
      ⟨set z to v 377⟩
      if (⟨v is too big 377⟩)
          RAISE(MP_Overflow);
      return z;
  }
```

376

首先，MP_fromint把z初始化成v的值，注意仅向XP_fromint传递正数：

```
⟨set z to v 377⟩≡
  if (v == LONG_MIN) {
      XP_fromint(nbytes, z, LONG_MAX + 1UL);
      XP_neg(nbytes, z, z, 1);
  } else if (v < 0) {
      XP_fromint(nbytes, z, -v);
      XP_neg(nbytes, z, z, 1);
  } else
      XP_fromint(nbytes, z, v);
  z[nbytes-1] &= msb;
```

前两个if语句处理负数：z设置为v的绝对值，然后将1作为第四个参数传递给XP_neg，将z设成它的二进制补码。MP_fromint必须专门处理值最小的那个整数，因为它不能取反。如果v为负，z的最高有效位将是1，而且必须舍弃额外的位。许多MP函数都习惯使用上面显示的z[nbytes-1] &=msb语句舍弃z中超出最高有效字节的那些位。

对于MP_fromint，nbits小于长整数的位数且v超出了z的范围时，会产生有符号溢出。

```
⟨v is too big 377⟩≡
  (nbits < 8*(int)sizeof (v) &&
      (v < -(1L<<(nbits-1)) || v >= (1L<<(nbits-1))))
```

这两个移位表达式计算最小及最大的n位长有符号整数。

19.3.1 转换

377 MP_toint和MP_cvt介绍了检查有符号溢出的另一情况：

```
⟨functions 374⟩+≡
  long MP_toint(T x) {
      unsigned char d[sizeof (unsigned long)];

      assert(x);
      MP_cvt(8*sizeof d, d, x);
      return XP_toint(sizeof d, d);
  }
```

如果d不能容纳x，MP_cvt就会产生MP_Overflow；如果d可以容纳x，XP_toint返回期望值。

MP_cvt完成两种转换：把一个MP_T转换成位数稍多或稍少的另一个MP_T。

```
⟨functions 374⟩+≡
  T MP_cvt(int m, T z, T x) {
      int fill, i, mbytes = (m - 1)/8 + 1;

      assert(m > 1);
      ⟨checked runtime errors for unary functions 378⟩
      fill = sign(x) ? 0xFF : 0;
      if (m < nbits) {
          ⟨narrow signed x 379⟩
      } else {
          ⟨widen signed x 379⟩
      }
      return z;
  }
```

```
⟨checked runtime errors for unary functions 378⟩≡
  assert(x); assert(z);
```

如果m小于nbits，MP_cvt就“缩小”x的值并将其赋予z。这种情况必须检查有符号溢出。如果x中m位到nbits−1位都是0或都是1，也就是说，如果将x作为一个m位整数处理的时候，x中超出的位都等于x的符号位，m位就适合x。下面的程序块中，如果x为负或零，则fill为FF，

378 因此如果位x[m..nbits−1]都是1或都是0，x[i]^fill应为0。

```
⟨narrow signed x 379⟩≡
  int carry = (x[mbytes-1]^fill)&~ones(m);
  for (i = mbytes; i < nbytes; i++)
      carry |= x[i]^fill;
  memcpy(z, x, mbytes);
  z[mbytes-1] &= ones(m);
  if (carry)
      RAISE(MP_Overflow);
```

如果x处于范围内，carry将为0；否则，carry的一些位将为1。carry的初始值忽略了将成为z

的部分非符号位的那些位。

如果m至少为nbits，MP_cvt“扩大”x的值并将它赋予z。这种情况不会产生溢出，但是MP_cvt必须传递由fill给出的x符号位。

```
⟨widen signed x 379⟩≡
  memcpy(z, x, nbytes);
  z[nbytes-1] |= fill&~msb;
  for (i = nbytes; i < mbytes; i++)
      z[i] = fill;
  z[mbytes-1] &= ones(m);
```

MP_tointu使用了类似的方法：它调用MP_cvtu把x转换成一个具有无符号长整型整数位数的MP_T，然后调用XP_toint返回这个值。

```
⟨functions 374⟩+≡
  unsigned long MP_tointu(T x) {
      unsigned char d[sizeof (unsigned long)];

      assert(x);
      MP_cvtu(8*sizeof d, d, x);
      return XP_toint(sizeof d, d);
  }
```

同样，MP_cvtu缩小或者扩大x的值，并将其赋予z。

```
⟨functions 374⟩+≡
  T MP_cvtu(int m, T z, T x) {
      int i, mbytes = (m - 1)/8 + 1;

      assert(m > 1);
      ⟨checked runtime errors for unary functions 378⟩
      if (m < nbits) {
          ⟨narrow unsigned x 380⟩
      } else {
          ⟨widen unsigned x 380⟩
      }
      return z;
  }
```

m小于nbits时，如果x中m到nbits−1位中的任一位为1则产生溢出，这是由与MP_cvt中代码类似、但相比起来更简单一些的代码进行校验检查的。

379

```
⟨narrow unsigned x 380⟩≡
  int carry = x[mbytes-1]&~ones(m);
  for (i = mbytes; i < nbytes; i++)
      carry |= x[i];
  memcpy(z, x, mbytes);
  z[mbytes-1] &= ones(m);
  ⟨test for unsigned overflow 376⟩
```

m大于等于nbits时不会产生溢出，而且z中超出的位将设置为0:

```
⟨widen unsigned x 380⟩≡
  memcpy(z, x, nbytes);
  for (i = nbytes; i < mbytes; i++)
      z[i] = 0;
```

19.3.2 无符号算法

正如代码MP_cvtu和MP_cvt所表明的，无符号算法函数比相应的有符号函数要更容易实现，因为它们不需要处理符号，而且溢出检测也更简单。无符号加法说明了一种简单情况；
380 XP_add完成了所有的工作。

```
⟨functions 374⟩+≡
  T MP_addu(T z, T x, T y) {
      int carry;

      ⟨checked runtime errors for binary functions 381⟩
      carry = XP_add(nbytes, z, x, y, 0);
      carry |= z[nbytes-1]&~msb;
      z[nbytes-1] &= msb;
      ⟨test for unsigned overflow 376⟩
      return z;
  }

⟨checked runtime errors for binary functions 381⟩≡
  assert(x); assert(y); assert(z);
```

减法也一样的容易，但是存在未处理的borrow时会引发MP_Overflow：

```
⟨functions 374⟩+≡
  T MP_subu(T z, T x, T y) {
      int borrow;

      ⟨checked runtime errors for binary functions 381⟩
      borrow = XP_sub(nbytes, z, x, y, 0);
      borrow |= z[nbytes-1]&~msb;
      z[nbytes-1] &= msb;
      ⟨test for unsigned underflow 381⟩
      return z;
  }

⟨test for unsigned underflow 381⟩≡
  if (borrow)
      RAISE(MP_Overflow);
```

MP_mul2u是最简单的乘法函数，因为不会产生溢出。

```
⟨functions 374⟩+≡
  T MP_mul2u(T z, T x, T y) {
      ⟨checked runtime errors for binary functions 381⟩
381   memset(tmp[3], '\0', 2*nbytes);
      XP_mul(tmp[3], nbytes, x, nbytes, y);
```

```
        memcpy(z, tmp[3], (2*nbits - 1)/8 + 1);
        return z;
    }
```

MP_mul2u计算结果将其值赋予tmp[3]，并把tmp[3]复制到z，这样x或y都可以用作z。如果MP_mul2u将计算的结果直接赋予z，这样就不起作用了。在MP_set中分配临时空间不仅将分配隔离开来，也避免了x与y的限制。

MP_mul也调用XP_mul计算tmp[3]中的双倍长度结果，然后把那个结果缩小成nbits，并赋予z。

```
⟨functions 374⟩+≡
  T MP_mulu(T z, T x, T y) {
      ⟨checked runtime errors for binary functions 381⟩
      memset(tmp[3], '\0', 2*nbytes);
      XP_mul(tmp[3], nbytes, x, nbytes, y);
      memcpy(z, tmp[3], nbytes);
      z[nbytes-1] &= msb;
      ⟨test for unsigned multiplication overflow 382⟩
      return z;
  }
```

如果tmp[3]中的nbits到2 · nbits–1位是1乘积就会溢出。采用检测MP_cvtu中类似条件的方法就可以检测这种情况：

```
⟨test for unsigned multiplication overflow 382⟩≡
  {
      int i;
      if (tmp[3][nbytes-1]&~msb)
          RAISE(MP_Overflow);
      for (i = 0; i < nbytes; i++)
          if (tmp[3][i+nbytes] != 0)
              RAISE(MP_Overflow);
  }
```

MP_divu将y复制到临时空间，从而避免了XP_div对其参数的限制： 382

```
⟨functions 374⟩+≡
  T MP_divu(T z, T x, T y) {
      ⟨checked runtime errors for binary functions 381⟩
      ⟨copy y to a temporary 383⟩
      if (!XP_div(nbytes, z, x, nbytes, y, tmp[2], tmp[3]))
          RAISE(MP_Dividebyzero);
      return z;
  }

⟨copy y to a temporary 383⟩≡
  {
      memcpy(tmp[1], y, nbytes);
      y = tmp[1];
  }
```

tmp[2]保存舍弃的余数；tmp[1]保存y、且y重新指向tmp[1]。tmp[3]是XP_div所需的

2 · nbyte +2位临时空间。MP_modu非常相似，只是它用tmp[2]保存商值：

```
⟨functions 374⟩+≡
  T MP_modu(T z, T x, T y) {
      ⟨checked runtime errors for binary functions 381⟩
      ⟨copy y to a temporary 383⟩
      if (!XP_div(nbytes, tmp[2], x, nbytes, y, z, tmp[3]))
          RAISE(MP_Dividebyzero);
      return z;
  }
```

19.3.3 有符号算法

AP的有符号数值表示使AP_add必须考虑x和y的符号。二进制补码表示法的特性允许MP_add避免这种分析，而且仅仅调用XP_add而不管x和y的符号。因而，有符号加法几乎与无符号加法一样，惟一的重要区别就是溢出的检测。

```
⟨functions 374⟩+≡
  T MP_add(T z, T x, T y) {
      int sx, sy;
      ⟨checked runtime errors for binary functions 381⟩
      sx = sign(x);
      sy = sign(y);
      XP_add(nbytes, z, x, y, 0);
      z[nbytes-1] &= msb;
      ⟨test for signed overflow 384⟩
      return z;
  }
```

383

加法中x和y符号相同时产生溢出。当和溢出时，它的符号不同于x和y的符号：

```
⟨test for signed overflow 384⟩≡
  if (sx == sy && sy != sign(z))
      RAISE(MP_Overflow);
```

有符号减法与加法形式相同，但是溢出的检测不同。

```
⟨functions 374⟩+≡
  T MP_sub(T z, T x, T y) {
      int sx, sy;

      ⟨checked runtime errors for binary functions 381⟩
      sx = sign(x);
      sy = sign(y);
      XP_sub(nbytes, z, x, y, 0);
      z[nbytes-1] &= msb;
      ⟨test for signed underflow 384⟩
      return z;
  }
```

对于减法，x和y的符号不同时发生溢出。x为正且y为负时，结果应为正；x为负且y为正时，结果应为负。因此，x和y符号不同时，结果与y符号相同，产生下溢。

```
⟨test for signed underflow 384⟩=
  if (sx != sy && sy == sign(z))
      RAISE(MP_Overflow);
```

384

x取相反数等价于用0减去x，仅当x为负时产生溢出，而且结果溢出时仍旧为负。

```
⟨functions 374⟩+≡
  T MP_neg(T z, T x) {
      int sx;

      ⟨checked runtime errors for unary functions 378⟩
      sx = sign(x);
      XP_neg(nbytes, z, x, 1);
      z[nbytes-1] &= msb;
      if (sx && sx == sign(z))
          RAISE(MP_Overflow);
      return z;
  }
```

MP_neg必须清除z最高有效字节中超出的那些位，因为x为正时，它们都是1.

实现有符号乘法的最简单方法是将负操作数取反，执行一个无符号乘法，操作数符号不同时再将结果取反。对于MP mul2，由于它计算双倍长度的结果，因此永远不会产生溢出，而且很容易填写细节：

```
⟨functions 374⟩+≡
  T MP_mul2(T z, T x, T y) {
      int sx, sy;

      ⟨checked runtime errors for binary functions 381⟩
      ⟨tmp[3] ← x·y 385⟩
      if (sx != sy)
          XP_neg((2*nbits - 1)/8 + 1, z, tmp[3], 1);
      else
          memcpy(z, tmp[3], (2*nbits - 1)/8 + 1);
      return z;
  }
```

```
⟨tmp[3] ← x·y 385⟩≡
  sx = sign(x);
  sy = sign(y);
  ⟨if x < 0, negate x 386⟩
  ⟨if y < 0, negate y 386⟩
  memset(tmp[3], '\0', 2*nbytes);
  XP_mul(tmp[3], nbytes, x, nbytes, y);
```

385

乘积具有2·nbits位，仅需要z的(2·nbits−1)/8+1字节。必要的时候在临时空间形成x和y的取反值，再重新指向x或y，以实现x和y取反

```
⟨if x < 0, negate x 386⟩≡
  if (sx) {
      XP_neg(nbytes, tmp[0], x, 1);
      x = tmp[0];
```

```
        x[nbytes-1] &= msb;
    }

⟨if y < 0, negate y 386⟩≡
    if (sy) {
        XP_neg(nbytes, tmp[1], y, 1);
        y = tmp[1];
        y[nbytes-1] &= msb;
    }
```

通常，必要的时候调用MP函数将x和y取反或复制，并保存在tmp[0]和tmp[1]中。

MP_mul类似于MP_mul2，只是仅拷贝2·nbit位结果中最低有效nbits位，而且，结果不能够放入nbits位，或者操作数符号相同且结果为负时，会发生溢出。

```
⟨functions 374⟩+≡
  T MP_mul(T z, T x, T y) {
      int sx, sy;

      ⟨checked runtime errors for binary functions 381⟩
      ⟨tmp[3] ← x·y 385⟩
      if (sx != sy)
          XP_neg(nbytes, z, tmp[3], 1);
      else
          memcpy(z, tmp[3], nbytes);
      z[nbytes-1] &= msb;
      ⟨test for unsigned multiplication overflow 382⟩
      if (sx == sy && sign(z))
          RAISE(MP_Overflow);
      return z;
  }
```

386

操作数符号相同时有符号除法与无符号除法非常相像，因为它们的商和余数都非负。仅当被除数是最小的n位负数且除数为-1时产生溢出；这种情况下，商为负。

```
⟨functions 374⟩+≡
  T MP_div(T z, T x, T y) {
      int sx, sy;

      ⟨checked runtime errors for binary functions 381⟩
      sx = sign(x);
      sy = sign(y);
      ⟨if x < 0, negate x 386⟩
      ⟨if y < 0, negate y 386⟩ else ⟨copy y to a temporary 383⟩
      if (!XP_div(nbytes, z, x, nbytes, y, tmp[2], tmp[3]))
          RAISE(MP_Dividebyzero);
      if (sx != sy) {
          ⟨adjust the quotient 388⟩
      } else if (sx && sign(z))
          RAISE(MP_Overflow);
      return z;
  }
```

MP_div会将y取反置入临时空间，或者将y复制到那里，因为y和z可能是相同的MP_T，而且用tmp[2]保存余数。

操作数符号不同时有符号除法和取余会复杂一些。这种情况下，商为负，但必须趋向于负无穷截取，且余数为正。所需的调整与AP_div和AP_mod相同：商取反，且如果余数非零则减少商。同样，如果无符号余数非零，y减去那个余数才是正确数值。 387

```
⟨adjust the quotient 388⟩≡
  XP_neg(nbytes, z, z, 1);
  if (!iszero(tmp[2]))
      XP_diff(nbytes, z, z, 1);
  z[nbytes-1] &= msb;
```

```
⟨macros 374⟩+≡
  #define iszero(x) (XP_length(nbytes,(x))==1 && (x)[0]==0)
```

由于MP_div舍弃了余数，因此它不影响余数的调整。MP_mod恰好相反：它仅调整余数，并用tmp[2]保存余数。

```
⟨functions 374⟩+≡
  T MP_mod(T z, T x, T y) {
      int sx, sy;

      ⟨checked runtime errors for binary functions 381⟩
      sx = sign(x);
      sy = sign(y);
      ⟨if x < 0, negate x 386⟩
      ⟨if y < 0, negate y 386⟩ else ⟨copy y to a temporary 383⟩
      if (!XP_div(nbytes, tmp[2], x, nbytes, y, z, tmp[3]))
          RAISE(MP_Dividebyzero);
      if (sx != sy) {
          if (!iszero(z))
              XP_sub(nbytes, z, y, z, 0);
      } else if (sx && sign(tmp[2]))
          RAISE(MP_Overflow);
      return z;
  }
```

19.3.4 简易函数

算术简易函数接收一个长整型或无无符号长整型直接操作数，根据需要将其转换成一个MP_T，并执行相应算法操作。y是以2^8为底数的一位数，这些函数可以使用XP导出的一位数函数。但是有两种情况会产生溢出：y太大以及操作数本身溢出。如果y太大，这些函数必须在引发异常之前完成操作及z的赋值。MP_addui描述了所有简易函数所使用的这种方法： 388

```
⟨functions 374⟩+≡
  T MP_addui(T z, T x, unsigned long y) {
      ⟨checked runtime errors for unary functions 378⟩
      if (y < BASE) {
```

```
            int carry = XP_sum(nbytes, z, x, y);
            carry |= z[nbytes-1]&~msb;
            z[nbytes-1] &= msb;
            ⟨test for unsigned overflow 376⟩
        } else if (applyu(MP_addu, z, x, y))
            RAISE(MP_Overflow);
        return z;
    }
```

```
⟨macros 374⟩+≡
  #define BASE (1<<8)
```

如果y是一位数，XP_sum可以计算x+y。nbits小于8且y太大时这段代码还会检测溢出，因为对任一x总和都会太大。否则，MP_addui调用applyu将y转换成MP_T，并应用更常用的函数MP_addu。如果y太大，applyu返回一个1，但仅在它计算完z之后：

```
⟨static functions 389⟩≡
  static int applyu(T op(T, T, T), T z, T x,
      unsigned long u) {
      unsigned long carry;

      { T z = tmp[2]; ⟨set z to u 376⟩ }
      op(z, x, tmp[2]);
      return carry != 0;
  }
```

applyu使用MP_fromintu的代码转换这个无符号长整型操作，并放入tmp[2]。它还保存转换中的carry，因为转换可能会溢出。然后调用它第一个参数指定的函数；而且，如果保存的carry非零，则返回1，否则返回0。函数op也会引发异常，但也仅在设置完z后。

389

无符号减法和乘法的简易函数非常相似。y小于2^8时，MP_subui调用MP_diff。

```
⟨functions 381⟩+≡
  T MP_subui(T z, T x, unsigned long y) {
      ⟨checked runtime errors for unary functions 381⟩
      if (y < BASE) {
          int borrow = XP_diff(nbytes, z, x, y);
          borrow |= z[nbytes-1]&~msb;
          z[nbytes-1] &= msb;
          ⟨test for unsigned underflow 381⟩
      } else if (applyu(MP_subu, z, x, y))
          RAISE(MP_Overflow);
      return z;
  }
```

如果y太大，不管x为何值x–y都下溢，因此MP_subui调用XP_diff之前不需要检查y是否太大。

MP_mului调用MP_product，但是nbits小于8时MP_mului必须显式检查y是否太大，因为x为零时XP_product不会捕捉那个错误。这个检查也是在计算完z之后执行的。

```
T MP_mului(T z, T x, unsigned long y) {
    ⟨checked runtime errors for unary functions 381⟩
    if (y < BASE) {
```

```
            int carry = XP_product(nbytes, z, x, y);
            carry |= z[nbytes-1]&~msb;
            z[nbytes-1] &= msb;
            ⟨test for unsigned overflow 376⟩
            ⟨check if unsigned y is too big 390⟩
        } else if (applyu(MP_mulu, z, x, y))
            RAISE(MP_Overflow);
        return z;
    }
```

⟨*check if unsigned y is too big* 390⟩≡

```
    if (nbits < 8 && y >= (1U<<nbits))
        RAISE(MP_Overflow);
```

390

MP_divui和MP_modui使用XP_quotient，但是它们必须检测自身的除数是否为零（因为XP_quotient只接受非零的一位除数），而且nbits小于8且y太大时它们必须检测溢出。

⟨*functions* 381⟩+≡

```
T MP_divui(T z, T x, unsigned long y) {
    ⟨checked runtime errors for unary functions 381⟩
    if (y == 0)
        RAISE(MP_Dividebyzero);
    else if (y < BASE) {
        XP_quotient(nbytes, z, x, y);
        ⟨check if unsigned y is too big 390⟩
    } else if (applyu(MP_divu, z, x, y))
        RAISE(MP_Overflow);
    return z;
}
```

MP_modui调用XP_quotient，但是仅用以计算余数。它舍弃保存在tmp[2]中的余数：

⟨*functions* 381⟩+≡

```
unsigned long MP_modui(T x, unsigned long y) {
    assert(x);
    if (y == 0)
        RAISE(MP_Dividebyzero);
    else if (y < BASE) {
        int r = XP_quotient(nbytes, tmp[2], x, y);
        ⟨check if unsigned y is too big 390⟩
        return r;
    } else if (applyu(MP_modu, tmp[2], x, y))
        RAISE(MP_Overflow);
    return XP_toint(nbytes, tmp[2]);
}
```

有符号算法简易函数使用相同的方法，但是调用不同的apply函数，这个函数使用MP_fromint代码将tmp[2]中的长整数转换成有符号MP_T，还调用期望函数，而且，如果直接操作数太大则返回1，否则返回0。

391

⟨*static functions* 389⟩+≡

```
static int apply(T op(T, T, T), T z, T x, long v) {
```

```
        { T z = tmp[2]; ⟨set z to v 377⟩ }
        op(z, x, tmp[2]);
        return ⟨v is too big 377⟩;
    }
```

|y|小于2^8时，有符号简易函数要完成的工作比它们对应的无符号简易函数要稍多一点，因为它们必须处理有符号操作数。一位数XP函数只接受正的一位操作数，因此有符号简易函数必须使用它们操作数的符号确定调用哪个函数。这个分析类似于AP函数所做的分析（参见前面相关章节），但是MP的二进制补码表示形式简化了细节。下表是加法的情形。

	$y<0$	$y\geq 0$
$x<0$	$-(\|x\|+\|y\|) = x-\|y\|$	$-(\|x\|-\|y\|) = x+\|y\|$
$x\geq 0$	$\|x\|-\|y\| = x-y\|$	$\|x\|+\|y\| = x+\|y\|$

y为负时，对任意x，x+y都等于x-|y|，因此MP_addi可以使用XP_diff计算总和；y非负时可以使用XP_sum。

```
⟨functions 381⟩+≡
  T MP_addi(T z, T x, long y) {
      ⟨checked runtime errors for unary functions 381⟩
      if (-BASE < y && y < BASE) {
          int sx = sign(x), sy = y < 0;
          if (sy)
              XP_diff(nbytes, z, x, -y);
          else
              XP_sum (nbytes, z, x,  y);
          z[nbytes-1] &= msb;
          ⟨test for signed overflow 384⟩
          ⟨check if signed y is too big 393⟩
      } else if (apply(MP_add, z, x, y))
          RAISE(MP_Overflow);
      return z;
  }
```

392

```
⟨check if signed y is too big 393⟩≡
  if (nbits < 8
  && (y < -(1<<(nbits-1)) || y >= (1<<(nbits-1))))
      RAISE(MP_Overflow);
```

有符号减法的情形与加法正好相反（参见前面章节AP_sub分析）：

	$y<0$	$y\geq 0$
$x<0$	$-(\|x\|-\|y\|) = x+\|y\|$	$-(\|x\|+\|y\|) = x-\|y\|$
$x\geq 0$	$\|x\|+\|y\| = x+\|y\|$	$\|x\|-\|y\| = x-\|y\|$

因此，y为负时MP_subi调用XP_sum将|y|与x相加，y非负时调用XP_diff。

```
⟨functions 381⟩+≡
  T MP_subi(T z, T x, long y) {
```

```
    ⟨checked runtime errors for unary functions 381⟩
    if (-BASE < y && y < BASE) {
        int sx = sign(x), sy = y < 0;
        if (sy)
            XP_sum (nbytes, z, x, -y);
        else
            XP_diff(nbytes, z, x,  y);
        z[nbytes-1] &= msb;
        ⟨test for signed underflow 384⟩
        ⟨check if signed y is too big 393⟩
    } else if (apply(MP_sub, z, x, y))
        RAISE(MP_Overflow);
    return z;
}
```

MP_muli使用MP_mul的策略：它将负的操作数取反，调用XP_product计算乘积，而且如果操作数符号不同还要将乘积取反。

```
⟨functions 381⟩+≡
  T MP_muli(T z, T x, long y) {
      ⟨checked runtime errors for unary functions 381⟩
      if (-BASE < y && y < BASE) {
          int sx = sign(x), sy = y < 0;
          ⟨if x < 0, negate x 386⟩
          XP_product(nbytes, z, x, sy ? -y : y);
          if (sx != sy)
              XP_neg(nbytes, z, x, 1);
          z[nbytes-1] &= msb;
          if (sx == sy && sign(z))
              RAISE(MP_Overflow);
          ⟨check if signed y is too big 393⟩
      } else if (apply(MP_mul, z, x, y))
          RAISE(MP_Overflow);
      return z;
  }
```

393

MP_divi和MP_modi必须检查除数是否为零，因为它们调用XP_quotient计算商和余数。MP_divi舍弃余数，而MP_modi舍弃商：

```
⟨functions⟩+≡
  T MP_divi(T z, T x, long y) {
      ⟨checked runtime errors for unary functions 381⟩
      if (y == 0)
          RAISE(MP_Dividebyzero);
      else if (-BASE < y && y < BASE) {
          int r;
          ⟨z ← x/y, r ← x mod y 395⟩
          ⟨check if signed y is too big 393⟩
      } else if (apply(MP_div, z, x, y))
          RAISE(MP_Overflow);
      return z;
  }
```

```
long MP_modi(T x, long y) {
    assert(x);
    if (y == 0)
        RAISE(MP_Dividebyzero);
    else if (-BASE < y && y < BASE) {
        T z = tmp[2];
        int r;
        ⟨z ← x/y, r ← x mod y 395⟩
        ⟨check if signed y is too big 393⟩
        return r;
    } else if (apply(MP_mod, tmp[2], x, y))
        RAISE(MP_Overflow);
    return MP_toint(tmp[2]);
}
```

394

MP_modi调用MP_toint而非XP_toint以确定符号得到正确地扩展。

MP_divi和MP_modi共有的代码块计算商和余数，并在x和y符号不同且余数非零时调整商和余数。

```
⟨z ← x/y, r ← x mod y 395⟩≡
  int sx = sign(x), sy = y < 0;
  ⟨if x < 0, negate x 386⟩
  r = XP_quotient(nbytes, z, x, sy ? -y : y);
  if (sx != sy) {
      XP_neg(nbytes, z, z, 1);
      if (r != 0) {
          XP_diff(nbytes, z, z, 1);
          r = y - r;
      }
      z[nbytes-1] &= msb;
  } else if (sx && sign(z))
      RAISE(MP_Overflow);
```

19.3.5 比较和逻辑操作

无符号比较很容易——MP_cmp就可以调用XP_cmp：

```
⟨functions 381⟩+≡
  int MP_cmpu(T x, T y) {
      assert(x);
      assert(y);
      return XP_cmp(nbytes, x, y);
  }
```

395 x和y符号不同时，MP_cmp(x, y)仅返回y与x的符号差别：

```
⟨functions 381⟩+≡
  int MP_cmp(T x, T y) {
      int sx, sy;

      assert(x);
```

```
        assert(y);
        sx = sign(x);
        sy = sign(y);
        if (sx != sy)
            return sy - sx;
        else
            return XP_cmp(nbytes, x, y);
    }
```

x和y符号相同时，MP_cmp可以把它们作为无符号数处理，然后调用XP_cmp进行比较。

比较简易函数不能使用applyu和apply，因为它们计算整数结果，而且它们并不要求它们的长整型和无符号长整型操作数匹配MP_T。这些函数仅仅是将MP_T与一个立即数进行比较；这个值太大时，就会反映在比较结果中 一个无符号长整数的位数至少为nbits时，MP_cmpui就把MP_T转换成一个无符号整数，并使用通常的C比较函数。否则，它就把立即数转换为MP_T保存在tmp[2]中，然后调用XP_cmp。

```
⟨functions 381⟩+≡
  int MP_cmpui(T x, unsigned long y) {
      assert(x);
      if ((int)sizeof y >= nbytes) {
          unsigned long v = XP_toint(nbytes, x);
          ⟨return -1, 0, +1, if v < y, v = y, v > y 396⟩
      } else {
          XP_fromint(nbytes, tmp[2], y);
          return XP_cmp(nbytes, x, tmp[2]);
      }
  }
```

```
⟨return -1, 0, +1, if v < y, v = y, v > y 396⟩≡
  if (v < y)
      return -1;
  else if (v > y)
      return 1;
  else
      return 0;
```

396

MP_cmpui调用XP_fromint之后不必检查溢出，因为那个调用仅在y的位数小于MP_T时执行。

x和y符号不同时MP_cmpi可以不用全部都进行比较，它采用了MP_cmpui的方法：如果立即数的位数不少于MP_T，就可以用C的比较函数完成比较。

```
⟨functions 374⟩+≡
  int MP_cmpi(T x, long y) {
      int sx, sy = y < 0;

      assert(x);
      sx = sign(x);
      if (sx != sy)
          return sy - sx;
```

```
        else if ((int)sizeof y >= nbytes) {
            long v = MP_toint(x);
            ⟨return -1, 0, +1, if v < y, v = y, v > y 396⟩
        } else {
            MP_fromint(tmp[2], y);
            return XP_cmp(nbytes, x, tmp[2]);
        }
    }
```

x和y符号相同且y的位数小于MP_T时，MP_cmpi可以安全地把y转换成MP_T并保存在tmp[2]中，然后调用XP_cmp比较x和tmp[2]　MP_cmpi调用MP_fromint而不是XP_fromint是为了正确处理负值的y。

二进制逻辑函数——MP_and、MP_or及MP_xor——是最容易实现的MP函数，因为结果的每个字节都是操作数相应字节商的按位操作。

```
⟨macros 374⟩+≡
  #define bitop(op) \
      int i; assert(z); assert(x); assert(y); \
      for (i = 0; i < nbytes; i++) z[i] = x[i] op y[i]; \
      return z
```

397

```
  ⟨functions 374⟩+≡
    T MP_and(T z, T x, T y) { bitop(&); }
    T MP_or (T z, T x, T y) { bitop(|); }
    T MP_xor(T z, T x, T y) { bitop(^); }
```

MP_not比较古怪，它不匹配bitop的模式：

```
⟨functions 374⟩+≡
  T MP_not(T z, T x) {
      int i;

      ⟨checked runtime errors for unary functions 378⟩
      for (i = 0; i < nbytes; i++)
          z[i] = ~x[i];
      z[nbytes-1] &= msb;
      return z;
  }
```

为三个逻辑简易函数的单数字操作数专门编写代码几乎没有什么收获，而且这些函数的直接操作数不会引发异常。仍旧可以使用applyu；它的返回值就被忽略不计了。

```
⟨macros 374⟩+≡
  #define bitopi(op) assert(z); assert(x); \
      applyu(op, z, x, y); \
      return z

⟨functions 374⟩+≡
  T MP_andi(T z, T x, unsigned long y) { bitopi(MP_and); }
  T MP_ori (T z, T x, unsigned long y) { bitopi(MP_or);  }
  T MP_xori(T z, T x, unsigned long y) { bitopi(MP_xor); }
```

这三个移位函数先执行可检查的运行期错误，并检查s大于或等于nbits时的简单情况，这种情况下，结果都是0或都是1，然后再调用XP_lshift或XP_rshift。XP_ashift用1进行填充，然后实现一个算术右移。

```
⟨macros 374⟩+≡
  #define shft(fill, op) \
      assert(x); assert(z); assert(s >= 0); \
      if (s >= nbits) memset(z, fill, nbytes); \
      else op(nbytes, z, nbytes, x, s, fill); \
      z[nbytes-1] &= msb; \
      return z
```

398

```
⟨functions 374⟩+≡
  T MP_lshift(T z, T x, int s) { shft(0, XP_lshift); }
  T MP_rshift(T z, T x, int s) { shft(0, XP_rshift); }
  T MP_ashift(T z, T x, int s) { shft(sign(x),XP_rshift); }
```

19.3.6 字符串转换

最后四个函数在字符串和MP_T之间进行转换。MP_fromstr很像strtoul；它把字符串解释成底数在2到36之间的一个无符号数。字母表示底数大于10时的9以上数字。

```
⟨functions 374⟩+≡
  T MP_fromstr(T z, const char *str, int base, char **end){
      int carry;
      assert(z);
      memset(z, '\0', nbytes);
      carry = XP_fromstr(nbytes, z, str, base, end);
      carry |= z[nbytes-1]&~msb;
      z[nbytes-1] &= msb;
      ⟨test for unsigned overflow 376⟩
      return z;
  }
```

XP_fromstr执行这个转换，而且，如果end非空则把*end设置成终止转换过程的那个字符地址。由于XP_fromint把转换得到的值添加到z，因此z被初始化为零。

MP_tostr完成相反的转换：它接收一个MP_T，并将MP_T值表示成以2到36之间的数为底数的字符串形式，用以填充一个字符串。

```
⟨functions 374⟩+≡
  char *MP_tostr(char *str, int size, int base, T x) {
      assert(x);
      assert(base >= 2 && base <= 36);
      assert(str == NULL || size > 1);
      if (str == NULL) {
          ⟨size ← number of characters to represent x in base 400⟩
          str = ALLOC(size);
      }
```

399

```
        memcpy(tmp[1], x, nbytes);
        XP_tostr(str, size, base, nbytes, tmp[1]);
        return str;
    }
```

如果str为空，MP_tostr就分配一个足够长的字符串保存x以base为底数的表示形式。MP_tostr使用AP_tostr的技巧计算这个字符串的大小：str至少必须有⌈nbits/k⌉个字符，这里k要使2^k是小于或等于base的两个值中的最大指数（参见第18章字符串和整数转换内容），而且由于终止空字符的存在还要加1。

```
⟨size ← number of characters to represent x in base 400⟩≡
  {
      int k;
      for (k = 5; (1<<k) > base; k--)
          ;
      size = nbits/k + 1 + 1;
  }
```

Fmt样式的转换函数格式化无符号或有符号MP_T。每个函数都需要两个参数：一个MP_T和一个位于2到36之间的底数。MP_fmtu调用MP_tostr转换它的MP_T，并调用Fmt_putd打印转换结果。回调Fmt_putd以printf的%d转换方式打印数值。

```
⟨functions 374⟩+≡
  void MP_fmtu(int code, va_list *app,
      int put(int c, void *cl), void *cl,
      unsigned char flags[], int width, int precision) {
      T x;
      char *buf;

      assert(app && flags);
      x = va_arg(*app, T);
400   assert(x);
      buf = MP_tostr(NULL, 0, va_arg(*app, int), x);
      Fmt_putd(buf, strlen(buf), put, cl, flags,
          width, precision);
      FREE(buf);
  }
```

MP_fmt要完成的工作稍多一点，因为它把MP_T解释成一个有符号数，但是MP_tostr只接受无符号MP_T。因而，MP_fmt自己分配缓冲区，这样如果需要的话它就能包含一个前导符号。

```
⟨functions 374⟩+≡
  void MP_fmt(int code, va_list *app,
      int put(int c, void *cl), void *cl,
      unsigned char flags[], int width, int precision) {
      T x;
      int base, size, sx;
      char *buf;

      assert(app && flags);
```

```
        x = va_arg(*app, T);
        assert(x);
        base = va_arg(*app, int);
        assert(base >= 2 && base <= 36);
        sx = sign(x);
        ⟨if x < 0, negate x 386⟩
        ⟨size ← number of characters to represent x in base 400⟩
        buf = ALLOC(size+1);
        if (sx) {
            buf[0] = '-';
            MP_tostr(buf + 1, size, base, x);
        } else
            MP_tostr(buf, size + 1, base, x);
        Fmt_putd(buf, strlen(buf), put, cl, flags,
            width, precision);
        FREE(buf);
    }
```

401

参考书目浅析

多精度算法经常用于编译器，而且有时还必须使用。例如，Clinger（1990）表明，把浮点的直接量转换成相应的IEEE浮点表示法有时需要多精度算法以获取最佳的准确度。

Schneier（1996）是关于密码学的一次广泛调查。此书非常实用，而且还用C实现了其中一些算法。这本书还有详尽的参考书目，对继续深入调查是个不错的起点。

正如第17章乘法一节内容所示，两个n位数相乘需要的时间与n^2成正比。Press et al.（1992）中的20.6节介绍了如何使用快速Fourier（傅立叶）转换在与n lgn lg lgn成比例的时间内实现乘法。它还通过计算1/y的倒数再乘以x实现了x/y。这种方法用于小数部分时需要多精度数值。

练习

19.1 当nbits是8的倍数时，MP函数做了大量不必要的工作。你是否能够修改MP的实现避免nbits mod 8为零时出现这种情况？实现你的方案并估量一下它的优点或代价。

19.2 对许多应用程序而言，一旦选好nbits，就不再改变。实现一个产生器，给定nbits的某个值，生成一个接口，并实现nbits位算法MP_nbits，其他与MP相同。

19.3 设计并实现一个定点多精度数算法的接口，也就是说，这些数都有整数部分和小数部分。客户调用程序应能够指定数值两部分的数字。一定要指定取舍的细节。Press et al.（1992）20.6节涉及了这道习题的一些有用算法。

19.4 设计并实现一个浮点数算法的接口，其中客户调用程序可以指定指数及有效位的位数。考虑这道题之前阅读一下Goldberg（1991）。

402

19.5 XP和MP函数没有使用const限定词参数，第17章接口一节中详细说明了其中原因。但是，XP_T和MP_T的其他定义可以正确地处理常数。例如，如果这样定义T：

```
typedef unsigned char T[];
```

“const T”就是指“无符号常字符型的数组”；又例如，MP_add可以这样声明：

```
unsigned char *MP_add(T z, const T x, const T y);
```

在MP_add中，x和y具有“指向无符号常值字符”的类型，因为形参中的数组类型“衰变”到相应的指针类型。当然，常数不能防止偶然失真，因为可能会这样，比方说相同的数组可能会既传递给z又传递给x。MP_add的这个声明说明了把T定义为数组类型的一个缺点：T不能用作返回类型，而且客户调用程序不能声明T类型的变量。这种数组类型仅对参数有用。将T定义为unsigned char的typedef可以避免这个问题：

```
typedef unsigned char T;
```

这样定义，MP_add的声明就可以是

```
T *MP_add(T z[], const T x[], const T y[]);
T *MP_add(T *z, T *x, T *y);
```

中的任何一种。

使用这两种T的定义重新实现XP及其客户调用程序、AP、MP、calc及mpcalc。再把结果的可读性与原始数据进行比较。

403

第20章 线 程

一般的C程序都是顺序执行程序或单线程程序，即程序中只有一条控制轨迹。由程序的单元计数器给出每条指令执行时的地址。大多数情况下存储单元都是顺序向前移动，一次一条指令。单元计数器偶尔会因跳转指令或调用指令而变为跳转的目的位置或被调用函数的地址。存储单元计数器的值描绘出了贯穿程序的一条描述程序执行过程的路径；这条路径看上去像一个贯穿程序的线程。

并行或多线程程序具有多个线程，而且最一般的情况下，这些线程都是同时执行的，至少理论上是这样。正是并行执行使得多线程应用程序的编写比单线程应用程序的编写更加复杂，因为线程相互之间进行交互时具有潜在的不确定性。本章的这三个接口导出了用以创建和管理线程、同步协作线程的行为以及在线程之间进行通信的函数。

对于具有并行行为这种内在特性的应用程序来说，线程非常有用。图形用户接口（graphical user interface，GUI）是个最好的例子；键盘输入、鼠标移动和点击以及显示输出内容全部是同时进行。在多线程系统中，线程可以专门致力于这些行为中的每一个，而不用考虑其他内容。这种方法有助于简化用户接口的实现，因为每个线程都可以像是一个顺序程序一样来设计并编写，除非它们必须与其他线程通信或同步。

在多处理器的计算机上，线程能够改进那些可以容易地分解成相对独立子任务的应用程序的性能。每个子任务都运行在单独的线程中，而且它们全部并行运行，这样完成起来就比顺序执行这些子任务要快一些。20.2节介绍了一个使用这种方法的排序程序。

线程对构造顺序程序也有帮助，因为它们具有状态：线程包含足够的相关信息，供其停止之后在停下的地方重新开始。例如，典型的UNIX C 编译器由一个单独的预处理程序和一个汇编程序组成。预处理程序读取源代码、包含头文件、扩展宏并给出合成的源代码；编译器读取并解析展开的源代码，生成代码以及汇编语言代码；汇编程序读取汇编语言生成目标代码。这些过程通常都是通过读写临时文件相互进行通信。排除临时文件以及读、写和删除这些文件的系统开销，使用线程，每个阶段都可以作为单个应用程序里的独立线程运行。编译器本身可能也在词法分析器和解析器上使用独立的线程。20.2节用计算素数的一个管道举例说明了线程的这种用法。

一些系统并不是针对多线程应用程序进行设计的，这会限制线程的效用。例如，大多数UNIX系统都具有阻塞I/O原语，也就是说，当一个线程发出读请求时，UNIX进程及其所有线程都在等待这个请求的完成。在这些系统上面，线程不能与I/O共用有用的计算结果。信号处理也非常类似。大多数UNIX系统是把信号和信号处理器与进程关联起来，而不是与进程中的个别线程关联。

线程系统支持用户级或核心级的线程，或者两种都支持。用户级线程完全以用户模式实

现．无需操作系统的帮助。用户级的线程包经常有上述的一些缺点；但另一方面，用户级线程的作用也非常大。下一节的Thread接口介绍了用户级线程。

核心级线程使用操作系统工具提供服务，例如，非阻塞I/O和每线程信号处理。较新的操作系统拥有核心级的线程，并使用这些线程提供线程接口。这些接口的一些操作需要系统
406 调用，但是，通常这样会比用户级线程里的类似操作代价更高。

即使在具有核心级线程的系统上，标准库可能不是可再次进入或者安全线程的。一个可再次进入的函数只能修改局部变量和参数。修改全局变量或使用静态变量来保存中间结果的函数是不可再次进入的。标准C库中一些函数的典型实现就是不可再次进入的。如果同时激活两个或更多的不可再次进入函数，它们就会以不可预知的方式修改这些中间值。在单线程程序中，由于直接和间接的递归，可以同时激活多个函数。在多线程程序中，因为不同的线程可以同时调用同一函数，因此可以激活多个函数。这样，同时调用一个不可再次进入函数的两个线程会修改未定义结果的相同存储空间。

线程安全函数使用同步机制管理共享数据的访问，并因而可以或者不可再次进入。这种函数可由多个线程同时调用，而不用考虑同步问题．这样多线程客户调用程序使用起来就更容易一些，但同时也产生了同步的代价，即使是对单线程客户调用程序。

标准C不要求库函数可再次进入或为线程安全的，因此程序员必须设想最坏情况，并使用同步原语确保某一时刻仅有一个线程在执行一个不可再次进入的库函数。

本书中的大多数函数都不是线程安全的，但是可再次进入．有些函数与Text_map一样都是不可再次进入的，且多线程客户调用程序必须调度它们自身的同步问题。例如，如果几个线程共享一个Table_T，它们必须确保每次只有其中一个在调用Table接口中具有Table_T的函数，就像下文中说明的那样。

一些线程接口既设计用作用户级线程，又用作核心级线程。UNIX、Open VMS、OS/2、Windows NT及Windows 95的大多数变体都具有开放软件基金会（Open Software Foundation）的分布式计算环境（Distributed Computing Environment，DCE），简称DCE。一般地，如果主机的操作系统支持，DCE线程就使用核心级线程；否则，DCE线程就被实现为用户级线程。DCE线程接口有50多个函数，比本章的其他三个接口要大很多，但是，DCE
407 接口的功能也更多。例如，它的实现支持线程级信号，保护对标准库函数调用的适当同步。

Sun Microsystem的Solaris 2 操作系统具有一个两级线程工具。核心级线程被称为轻量进程或LWP；每个UNIX“重量”进程至少有一个LWP。Solaris运行一个或多个LWP以运行一个UNIX进程。对LWP的核心支持包括非阻塞I/O及每LWP信号。用户级线程由一个类似于Thread但比Thread大的接口提供，一个LWP可以服务于一个或更多用户级线程。Solaris在LWP之间多路复用处理器，并像这样在用户级线程之间复用自身。

POSIX（Portable Operating Systems Interface，便携式操作系统接口）线程接口——简称pthread——是作为重要的标准线程接口出现的。大多数商家现在都提供pthread的实现，可能是基于他们自己的线程接口。例如，Sun Microsystems使用Solaris 2 LWP实现pthread。pthread工具是Thread和Sem导出的那些函数的父集。更大一些的POSIX接口可以处理每线程信号，包含几种同步机制，以及指定哪个标准C库函数必须是线程安全的。

20.1 接口

本章中的这三个接口每个都很小；它们被分成单独的接口，因为每个接口都有一个相关但截然不同的意图。

理论上，所有运行着的线程都是并行运行，但实际上，线程通常比实际的处理器多。因而处理器会根据一个调度策略在运行的线程之间进行多路复用；采用无优先权调度，运行的线程执行一个函数可能会将其阻塞，或相反，释放其处理器。采用有优先权调度，运行的线程隐式地放弃它的处理器。这种策略通常用时钟中断实现定期的中断，并将其处理器给另一个运行中的线程。份额是线程在取得优先权运行之前的时间量，线程取得优先权时上下文切换（context switch）就挂起当前线程，重新启动另一（可能相同）运行线程。如果一个运行的线程被阻塞，也会发生无优先权调度的上下文切换。如果Thread接口的实现支持，它就使用优先权。

原子行为的执行没有优先权，启动执行原子行为的线程将完成该行为，不会被另一线程中断。如果线程调用一个原子函数，该调用的执行就不会被中断。本章介绍的大多数函数必须是原子函数，这样才可以预测它们的结果和作用，但是原子函数可能会死锁，Sem接口中的同步函数就是个例子。 408

正如最后两段内容所述，并行编程有它自己的专业术语，不同术语经常用于相同概念。例如，线程可能称为轻量进程、任务、子任务或微任务；同步机制可能称为事件、条件变量、同步资源以及消息。

20.1.1 Thread

Thread接口导出一个异常及支持线程创建的函数。

⟨*thread.h*⟩≡

```
#ifndef THREAD_INCLUDED
#define THREAD_INCLUDED
#include "except.h"

#define T Thread_T
typedef struct T *T;

extern const Except_T Thread_Failed;
extern const Except_T Thread_Alerted;

extern int  Thread_init (int preempt, ...);
extern T    Thread_new  (int apply(void *),
                         void *args, int nbytes, ...);
extern void Thread_exit (int code);
extern void Thread_alert(T t);
extern T    Thread_self (void);
extern int  Thread_join (T t);
extern void Thread_pause(void);

#undef T
#endif
```

409 对此接口中所有函数的调用都具有原子性。

Thread_init初始化线程系统；必须在调用其他任何函数之前调用该函数，在Thread_init之前调用此接口或Sem与Chan接口的其他任何函数，或多次调用Thread_init都会引发可检查的运行期错误。

如果preempt为0，Thread_init就把线程系统初始化为仅支持无优先权的调度，并返回1。如果preempt为1，线程系统就初始化为有优先权调度。如果支持优先级，Thread_init则返回1；否则，系统初始化为无优先权调度，且Thread_init返回0。

一般客户调用程序在main中初始化线程系统。例如，对于需要优先权的客户调用程序，main通常具有如下的形式。

```
int main(int argc, char *argv[]) {
    int preempt;
    ...
    preempt = Thread_init(1, NULL);
    assert(preempt == 1);
    ...
    Thread_exit(EXIT_SUCCESS);
    return EXIT_SUCCESS;
}
```

Thread_init也可以接受依赖于实现的其他参数，经常出名称－数值对指定。例如，对于支持优先权的实现，

```
preempt = Thread_init(1, "priorities", 4, NULL);
```

会把线程系统初始化为四个优先级。未知的可选参数通常都被忽略掉了。使用这种方法的实现通常都期望以空指针作为终止参数。

正如上述代码模板所表明的，线程必须调用Thread_exit结束运行。整数参数是个退出码，更像传递给标准库中exit函数的那个参数。可能在等待调用线程消亡的其他线程可以获得这个值，参见下文说明。如果系统中只有一个线程，调用Thread_exit就等价于调用exit。

Thread_new创建新线程并返回它的线程句柄，这是一个隐式指针；线程句柄被传递给

410 Thread_self。新线程独立于它的创建线程而运行。当新线程开始执行，它相当于执行

```
void *p = ALLOC(nbytes);
memcpy(p, args, nbytes);
Thread_exit(apply(p));
```

即，apply调用的参数为args所指向的nbytes字节的副本，这里args假定为指向新线程的参数数据。args经常是个指向结构的指针，结构的字段保存apply的参数，nbytes是那个结构的大小。新线程开始执行时异常堆栈为空：它并不继承其调用线程中TRY-EXCEPT语句所建立的异常状态。异常是针对线程的；一个线程里执行的TRY-EXCEPT语句不能影响另一线程中的异常。

如果args非空且nbytes为零，新线程的执行就相当于执行

```
Thread_exit(apply(args));
```

即，args传递给未修改的apply。如果args为空，新线程就等价于执行

```
Thread_exit(apply(NULL));
```

apply为空、或args为空且nbytes为负时引发一个可检查的运行期错误。如果args为空，则忽略nbytes。

与Thread_init相同，Thread_new会接受针对于实现的其他参数，常由名称-数值对指定。

```
Thread_T t;
t = Thread_new(apply, args, nbytes, "priority", 2, NULL);
```

就是个例子，它创建了一个优先级为2的新线程。就像这个例子所说明的，可选参数应当终止于一个空指针。

线程的创建是同步的：Thread_new在新线程创建并接收到其参数后、但可能在新线程开始执行之前返回。如果Thread_new由于资源限制而不能创建新线程则会引发Thread_Failed。例如，实现可能会限制能够同时存在的线程数；当超过这个限制时，Thread_new就会引发Thread_Failed。 411

Thread_exit(code)结束调用线程的执行。等待调用线程终止的线程（依靠Thread_join）可以重新开始，同时code的值作为每个重新开始的线程里Thread_join调用结果返回。当最后一个线程调用Thread_exit，整个程序就通过调用exit(code)结束。

Thread_join(t)使调用线程挂起直到线程t调用Thread_exit结束。当线程t结束，调用线程就重新开始，而且Thread_join返回由t传递给Thread_exit的整数。如果t给不存在的线程命名，Thread_join立即返回-1。有种特殊情况，Thread_join(NULL)的调用会等待所有线程结束，包括可能是由其他线程创建的那些线程。这种情况下，Thread_join返回0。非空的t给调用线程命名，或多个线程指定一个空t，都是一个可检查的运行期错误。Thread_join可以引发Thread_Alerted。

Thread_self返回调用线程的线程句柄。

Thread_pause让调用线程放弃处理器，如果有线程已经准备好运行，就供另一准备好的线程运行。Thread_pause主要用于无优先权的调度；优先权调度不需要调用Thread_pause。

线程有三种状态：运行、阻塞及消亡；新线程从运行开始。如果它调用Thread_join，就变成阻塞，等待另一线程结束。当一个线程调用Thread_exit，它就消亡了。线程调用Chan导出的通信函数或Sem导出的同步函数时也会阻塞；而没有运行中的线程就会引发可检查的运行期错误。

Thread_alert（t）设置t的“未决警示”标志。如果t阻塞，Thread_alert使t可运行，并为它做准备以清空它的未决警示标志，在它下次运行的时候引发Thread_Alerted。如果t已经运行，Thread_alert调度t清空它的标志，并在下次它调用Thread_join或者阻塞通信或同步函数的时候引发Thread_Alerted。向Thread_alert传递空句柄或不存在的线程句柄将引发可检查的运行期错误。

运行的线程无法终止；线程必须自行结束，或者通过调用Thread_exit，或者通过响应Thread_Alerted。如果线程没有捕捉Thread_Alerted，整个程序将终止于一个未捕获的异常错误。对警告的最常见响应就是终止线程，具有如下一般形式的apply函数可以终止线程。 412

```
int apply(void *p) {
    TRY
        ...
    EXCEPT(Thread_Alerted)
        Thread_exit(EXIT_FAILURE);
    END_TRY;
    Thread_exit(EXIT_SUCCESS);
}
```

TRY-EXCEPT语句必须由线程本身执行。诸如

```
Thread_T t;
TRY
    t = Thread_new(...);
EXCEPT(Thread_Alerted)
    Thread_exit(EXIT_FAILURE);
END_TRY;
Thread_exit(EXIT_SUCCESS);
```

的代码不正确，因为TRY-EXCEPT应用于调用线程，而不是新建线程。

20.1.2 一般信号量

一般信号量或底数信号量是低级同步原语。理论上，信号量是个可以原子性地增加和减少的受保护整数。关于信号量s的两个操作是wait和signal。signal（s）逻辑上等价于原子性的增加；wait（s）等待s为正，然后再进行原子性的消减：

```
while (s <= 0)
    ;
s = s - 1;
```

413

当然，实际的实现会阻塞调用线程；它们不像这里解释的那样循环。

Sem接口将计数器封装在一个结构中，并导出一个初始化函数以及两个同步函数：

⟨*sem.h*⟩≡

```
#ifndef SEM_INCLUDED
#define SEM_INCLUDED

#define T Sem_T
typedef struct T {
    int count;
    void *queue;
} T;

⟨exported macros 416⟩

extern void Sem_init  (T *s, int count);
extern T   *Sem_new   (int count);
extern void Sem_wait  (T *s);
extern void Sem_signal(T *s);

#undef T
#endif
```

信号量是指向Sem_T结构一个实例的指针。这个接口揭示了Sem_T的内部结构，而且还可以进行静态分配或者嵌入到其他结构中。客户调用程序必须将Sem_T当作一种隐式类型，且仅能通过该接口中的函数访问其值所在的字段；间接访问Sem_T的字段是个不可检查的运行期错误。给该接口的任意函数传递空Sem_T指针，将会引发可检查的运行期错误。

Sem_init接受一个指向Sem_T的指针及其计数器的初值；然后初始化信号量的数据结构，并将其计数器设为特定的初值。一旦初始化，指向Sem_T的指针可以传递给那两个同步函数。同一信号量上多次调用Sem_init是一个不可检查的运行期错误。

Sem_new是

414

```
Sem_T *s;
NEW(s);
Sem_init(s, count);
```

的等价原子函数，它可以引发Mem_Failed。

Sem_wait接受一个指向Sem_T的指针，等待它的计数器变为正数，然后按1递减计数器，并返回。这个操作是原子操作。如果设置了调用线程的未决警示标志，Sem_wait就会立即引发Thread_Alerted，且不减低计数器。如果线程阻塞的时候设置未决警示标志，线程就不再等待，并引发Thread_Alerted，且不减低计数器。调用Thread_init之前调用Sem_wait是个可检查的运行期错误。

Sem_signal接受一个指向Sem_T的指针并原子性增加计数器。如果其他线程在等待计数器变为正数，而Sem_signal使它为正，那些线程的其中一个就会完成Sem_wait的调用。调用Thread_init之前调用Sem_wait是个可检查的运行期错误。

向Sem_wait或Sem_signal传递未初始化的信号量是个不可检查的运行期错误。

Sem_wait和Sem_signal操作中隐含的排队过程是先入先出；这是公平的。也就是说，阻塞在信号量s的线程将在t之后调用Sem_wait（&s）的其他线程之前重新开始。

二进制信号量互斥体是个一般信号量，它的计数器是0或1。互斥体用于互斥现象。举个例子，

```
Sem_T mutex;
Sem_init(&mutex, 1);
...
Sem_wait(&mutex);
statements
Sem_signal(&mutex);
```

创建并初始化一个二进制信号量，并用它确保某一时刻只有一个线程在执行*statement*语句，这是临界区的一个例子。这个术语是如此常用，以至于Sem为它导出了实现具有如下语法的LOCK-END_LOCK语句的宏：

415

```
LOCK(mutex)
    statements
END_LOCK
```

这里mutex是个初始为1的二进制信号量。LOCK语句有助于避免出现忘记在临界区末尾调用Sem_signal以及调用具有错误信号量的Sem_signal这样的常见的灾难性的错误。

```
⟨exported macros 416⟩≡
  #define LOCK(mutex) do { Sem_T *_yymutex = &(mutex); \
      Sem_wait(_yymutex);
  #define END_LOCK Sem_signal(_yymutex); } while (0)
```

如果*statement*语句可以引发异常，那么一定不能使用LOCK-END_LOCK，因为如果发生异常，mutex就不会释放了。这种情况下，正确的术语是

```
TRY
    Sem_wait(&mutex);
    statements
FINALLY
    Sem_signal(&mutex);
END_TRY;
```

FINALLY语句确保了不管是否有异常发生都释放mutex。另一个合理的方法是在LOCK和END_LOCK的定义中结合这种术语，但是这之后每次使用LOCK-END_LOCK都会招致TRY-FINALLY语句的系统开销。

互斥信号量mutex经常嵌在ADT中保证ADT的访问是线程安全的。例如，

```
typedef struct {
    Sem_T mutex;
    Table_T table;
} Protected_Table_T;
```

416 将mutex与table相关联。代码

```
Protected_Table_T tab;
tab.table = Table_new(...);
Sem_init(&tab.mutex, 1);
```

创建了一个受保护的表；

```
LOCK(tab.mutex)
    value = Table_get(tab.table, key);
END_LOCK;
```

原子性地获取与key相关的值。请注意，LOCK接收mutex而非其地址。因为Table_put可以引发Mem_Failed，所以tab的加法应当用如下的代码实现：

```
TRY
    Sem_wait(&tab.mutex);
    Table_put(tab.table, key, value);
FINALLY
    Sem_signal(&tab.mutex);
END_TRY;
```

20.1.3 同步通信通道

Chan接口提供可用于线程之间传递数据的同步通信通道。

```
⟨chan.h⟩≡
  #ifndef CHAN_INCLUDED
  #define CHAN_INCLUDED
```

```
#define T Chan_T
typedef struct T *T;

extern T   Chan_new    (void);
extern int Chan_send   (T c, const void *ptr, int size);
extern int Chan_receive(T c,       void *ptr, int size);

#undef T
#endif
```

417

Chan_new创建、初始化及返回一个新通道，这是个指针。Chan_new可以引发Mem_Failed。

Chan_send接受一个通道，也即一个指向保存发送数据及缓冲区容纳字节数的缓冲区的指针，调用线程一直等待，直到另一线程调用具有同一通道的Chan_receive；当发生这种会合时，就从发送方处将数据拷贝到接收方，然后两个函数都返回。Chan_send返回接收方接收的字节数。

Chan_receive接受一个通道，也即一个指向保存接收数据及其可容纳的最大字节数的缓冲区指针。调用者一直等待直到另一线程调用具有同一通道的Chan_send；当发生这种会合时，数据就从发送方处拷贝到接收方，然后两个函数都返回。如果发送方提供的字节多于size，多出的字节就被舍弃了。Chan_receive返回接受的字节数。

Chan_send和Chan_receive都接受0 size。向任一函数传递空Chan_T、空ptr或负size都会引发可检查的运行期错误。如果设置了调用线程的未决警示标志，Chan_send和Chan_receive立刻就会引发Thread_Alerted。如果线程阻塞的时候设置未决警示标志，线程就不再等待，并引发Thread_Alerted。这种情况下，数据可能传输过去也可能没有传输。

调用Thread_init之前调用这个接口中的任一函数都是可检查的运行期错误。

20.2　示例

本节的三个程序接受了线程及通道的简单使用，以及互斥现象信号量的使用。将在下节详细介绍的Chan实现是同步信号量的使用示例。

20.2.1　并行排序

如果有优先权，线程都会并行执行，至少理论上是这样。一组协同操作的线程可以处理问题的各个独立部分。在多处理器的系统上，这种方法使用并发性减少整体的执行时间。当然，在单处理器系统上，这个程序实际上运行得会慢一点，因为在线程间切换需要系统开销。但是，这种方法的确说明了Thread接口的使用。 418

排序这个问题很容易可以分解成独立的子部分。sort生成特定数量的随机整数，进行并行排序，然后检查结果排序的情况：

```
⟨sort.c⟩≡
  #include <stdlib.h>
  #include <stdio.h>
  #include <time.h>
  #include "assert.h"
  #include "fmt.h"
```

```
#include "thread.h"
#include "mem.h"

⟨sort types 421⟩
⟨sort data 422⟩
⟨sort functions 420⟩

main(int argc, char *argv[]) {
    int i, n = 100000, *x, preempt;

    preempt = Thread_init(1, NULL);
    assert(preempt == 1);
    if (argc >= 2)
        n = atoi(argv[1]);
    x = CALLOC(n, sizeof (int));
    srand(time(NULL));
    for (i = 0; i < n; i++)
        x[i] = rand();
    sort(x, n, argc, argv);
    for (i = 1; i < n; i++)
        if (x[i] < x[i-1])
            break;
    assert(i == n);
    Thread_exit(EXIT_SUCCESS);
    return EXIT_SUCCESS;
}
```

time、srand和rand是标准C库函数。time返回日历时间一些必要编码，而srand将使用这些编码设置用于生成伪随机数序列的种子。接下来调用rand返回这个序列中的随机数。sort一开始以n个随机数填充x[0..n-1]。

419

函数sort是快速排序的一个实现。教科书上用“中心”值把这个数组分成两个子数组，然后递归调用自身排序每个子数组。子数组为空时递归到达最底端。

```
void quick(int a[], int lb, int ub) {
    if (lb < ub) {
        int k = partition(a, lb, ub);
        quick(a, lb, k - 1);
        quick(a, k + 1, ub);
    }
}

void sort(int *x, int n, int argc, char *argv[]) {
    quick(x, 0, n - 1);
}
```

partition[a, i, j]任意挑选a[i]作为中心值，然后重新安排a[i..j]的位置，让a[i..k-1]中的所有值都小于或等于这个中心值v；a[k+1..j]中的所有值都大于v；且a[k]就是v。

```
⟨sort functions 420⟩≡
  int partition(int a[], int i, int j) {
      int v, k, t;
```

```
        j++;
        k = i;
        v = a[k];
        while (i < j) {
            i++; while (a[i] < v && i < j) i++;
            j--; while (a[j] > v         ) j--;
            if (i < j) { t = a[i]; a[i] = a[j]; a[j] = t; }
        }
        t = a[k]; a[k] = a[j]; a[j] = t;
        return j;
    }
```

420

partition的最后一次交换把v留在a[k]中，partition返回k。

对quick的递归调用可由单独的线程并行执行。首先，quick的参数必须打包在一个结构中，这样quick可以传递到Thread_new：

```
⟨sort types 421⟩≡
  struct args {
      int *a;
      int lb, ub;
  };
```

```
⟨sort functions 420⟩+≡
  int quick(void *cl) {
      struct args *p = cl;
      int lb = p->lb, ub = p->ub;
      if (lb < ub) {
          int k = partition(p->a, lb, ub);
          ⟨quick 421⟩
      }
      return EXIT_SUCCESS;
  }
```

递归调用在单个线程中执行，而且就像子数组里有足够的元素值得这样做。例如，a[lb..k-1]这样排序：

```
⟨quick 421⟩≡
  p->lb = lb;
  p->ub = k - 1;
  if (k - lb > cutoff) {
      Thread_T t;
      t = Thread_new(quick, p, sizeof *p, NULL);
      Fmt_print("thread %p sorted %d..%d\n", t, lb, k - 1);
  } else
      quick(p);
```

这里cutoff给出单个线程里排序所需的最少元素数。类似地，a[k+1..ub]这样排序：

421

```
⟨quick 421⟩+≡
  p->lb = k + 1;
  p->ub = ub;
  if (ub - k > cutoff) {
```

```
        Thread_T t;
        t = Thread_new(quick, p, sizeof *p, NULL);
        Fmt_print("thread %p sorted %d..%d\n", t, k + 1, ub);
    } else
        quick(p);
```

sort最先调用quick，它在排序进展过程中产生很多线程；然后调用Thread_join等待所有这些线程结束：

```
⟨sort data 422⟩≡
  int cutoff = 10000;

⟨sort functions 420⟩+≡
  void sort(int *x, int n, int argc, char *argv[]) {
      struct args args;

      if (argc >= 3)
          cutoff = atoi(argv[2]);
      args.a = x;
      args.lb = 0;
      args.ub = n - 1;
      quick(&args);
      Thread_join(NULL);
  }
```

按n和cutoff的默认值100 000和10 000执行sort，产生18个线程：

```
% sort
thread 69f08 sorted 0..51162
thread 6dfe0 sorted 51164..99999
thread 72028 sorted 51164..73326
thread 76070 sorted 73328..99999
thread 6dfe0 sorted 51593..73326
thread 72028 sorted 73328..91415
422 thread 7a0b8 sorted 51593..69678
thread 7e100 sorted 73328..83741
thread 82148 sorted 3280..51162
thread 69f08 sorted 73328..83614
thread 7e100 sorted 51593..67132
thread 6dfe0 sorted 7931..51162
thread 69f08 sorted 14687..51162
thread 6dfe0 sorted 14687..37814
thread 72028 sorted 37816..51162
thread 69f08 sorted 15696..37814
thread 6dfe0 sorted 15696..26140
thread 76070 sorted 26142..37814
```

不同的执行给不同的值排序，因此quick每次执行所创建的线程数量及打印的记录数量也会不同。

sort有个重要的漏洞：它不能包含quick中的Fmt_print调用，不能确保Fmt_print可以再次进入，而且C库中的许多例程都是不可再次进入的，什么也不能保证printf或其他任何库程序中断以后再重新开始仍将正确运行。

20.2.2 临界区

在有优先权的系统中任何被多个线程访问的数据都必须得到保护。访问必须限制在一个临界区中，这个区域中一次只允许运行一个线程。spin是个简单的范例，说明了访问共享数据的正确方式和错误方式。

```
⟨spin.c⟩≡
  #include <stdio.h>
  #include <stdlib.h>
  #include "assert.h"
  #include "fmt.h"
  #include "thread.h"
  #include "sem.h"

  #define NBUMP 30000

  ⟨spin types 425⟩
  ⟨spin functions 424⟩

  int n;
  int main(int argc, char *argv[]) {
      int m = 5, preempt;

      preempt = Thread_init(1, NULL);
      assert(preempt == 1);
      if (argc >= 2)
          m = atoi(argv[1]);
      n = 0;
      ⟨increment n unsafely 424⟩
      Fmt_print("%d == %d\n", n, NBUMP*m);
      n = 0;
      ⟨increment n safely 425⟩
      Fmt_print("%d == %d\n", n, NBUMP*m);
      Thread_exit(EXIT_SUCCESS);
      return EXIT_SUCCESS;
  }
```

spin生成m个线程，每个都递增nNBUMP次。首先生成的m个线程不能确保n是原子性地递增：

```
⟨increment n unsafely 424⟩≡
  {
      int i;
      for (i = 0; i < m; i++)
          Thread_new(unsafe, &n, 0, NULL);
      Thread_join(NULL);
  }
```

main创建这m个线程，每个都调用unsafe，并用指针指向n的指针：

```
⟨spin functions 424⟩≡
  int unsafe(void *cl) {
      int i, *ip = cl;
```

423

```
        for (i = 0; i < NBUMP; i++)
            *ip = *ip + 1;
        return EXIT_SUCCESS;
424     }
```

unsafe是不正确的，因为*ip=*ip+1的执行可能会被中断。如果恰好在获取*ip之后被中断，而且其他线程的*ip在递增，那么赋予*ip的值就会出错。

第二批生成的m个线程调用

```
⟨spin types 425⟩≡
  struct args {
      Sem_T *mutex;
      int *ip;
  };

⟨spin functions 424⟩+≡
  int safe(void *cl) {
      struct args *p = cl;
      int i;

      for (i = 0; i < NBUMP; i++)
          LOCK(*p->mutex)
              *p->ip = *p->ip + 1;
          END_LOCK;
      return EXIT_SUCCESS;
  }
```

safe确保每次只有一个线程在执行临界区，也就是语句*ip=*ip+1，main初始化所有线程都用以在safe中输入临界区的一个二进制信号量：

```
⟨increment n safely 425⟩≡
  {
      int i;
      struct args args;
      Sem_T mutex;
      Sem_init(&mutex, 1);
      args.mutex = &mutex;
      args.ip = &n;
      for (i = 0; i < m; i++)
          Thread_new(safe, &args, sizeof args, NULL);
      Thread_join(NULL);
425   }
```

任何时候都会发生抢占，因此使用unsafe的线程每次执行spin都会产生不同的结果：

```
% spin
87102 == 150000
150000 == 150000
% spin
148864 == 150000
150000 == 150000
```

20.2.3 生成素数

最后一个例子是说明由通信通道实现的管道。sieve *N*计算并打印小于或等于*N*的素数。例如：

```
% sieve 100
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73
79 83 89 97
```

sieve实现了计算素数的著名埃拉托色尼筛，其中每个“筛子”都是一个舍弃了它的素数倍数的线程。通道把这些线程连接起来形成一个管道，如图20-1所示。源线程（白框）生成2及2以后的奇数，并将它们送入管道。source和sink（暗灰色框）之间的过滤器（浅灰色框）去掉它们的素数倍数，并把其他数传递出去。sink也会过滤出它的素数，但是如果一个数是通过sink的过滤器得到的，那它就是个素数。图20-1中的每个方框都是个线程；框中的数也都是与那个线程相关的素数。方框之间构成管道的直线就是通道。

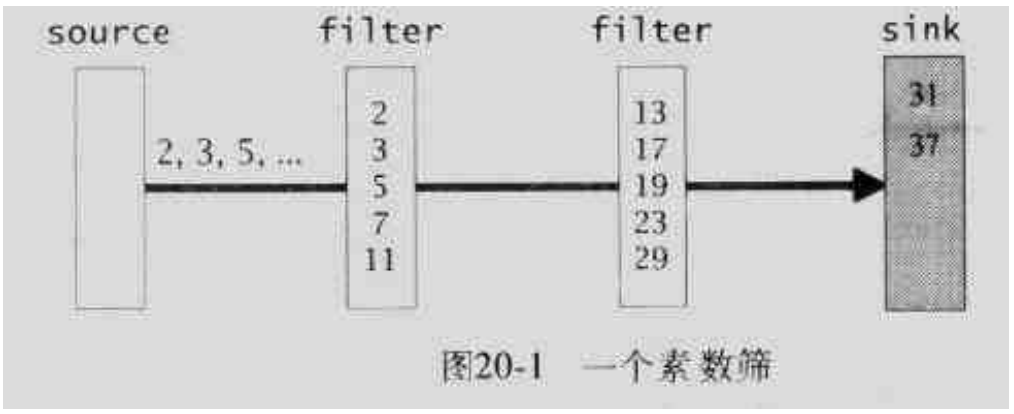


图20-1　一个素数筛

sink和每个filter里放有n个素数。当sink收集到n个素数——图20-1中为5个——它就生成自身的一个副本并转换成filter。图20-2说明了sieve计算100以内包括100的素数时，如何进行扩展。

sieve初始化线程系统后，就为source和sink生成线程，并用新的通道将它们连接起来，然后退出：

```
⟨sieve.c⟩≡
  #include <stdio.h>
  #include <stdlib.h>
  #include "assert.h"
  #include "fmt.h"
  #include "thread.h"
  #include "chan.h"

  struct args {
      Chan_T c;
      int n, last;
  };

  ⟨sieve functions 429⟩

  int main(int argc, char *argv[]) {
      struct args args;

      Thread_init(1, NULL);
      args.c = Chan_new();
```

```
        Thread_new(source, &args, sizeof args, NULL);
        args.n    = argc > 2 ? atoi(argv[2]) : 5;
        args.last = argc > 1 ? atoi(argv[1]) : 1000;
        Thread_new(sink,   &args, sizeof args, NULL);
        Thread_exit(EXIT_SUCCESS);
        return EXIT_SUCCESS;
    }
```

source把整数发射到它的“输出”通道，在args结构的c字段——也是source需要的惟一字段中传递：

426 ~ 427

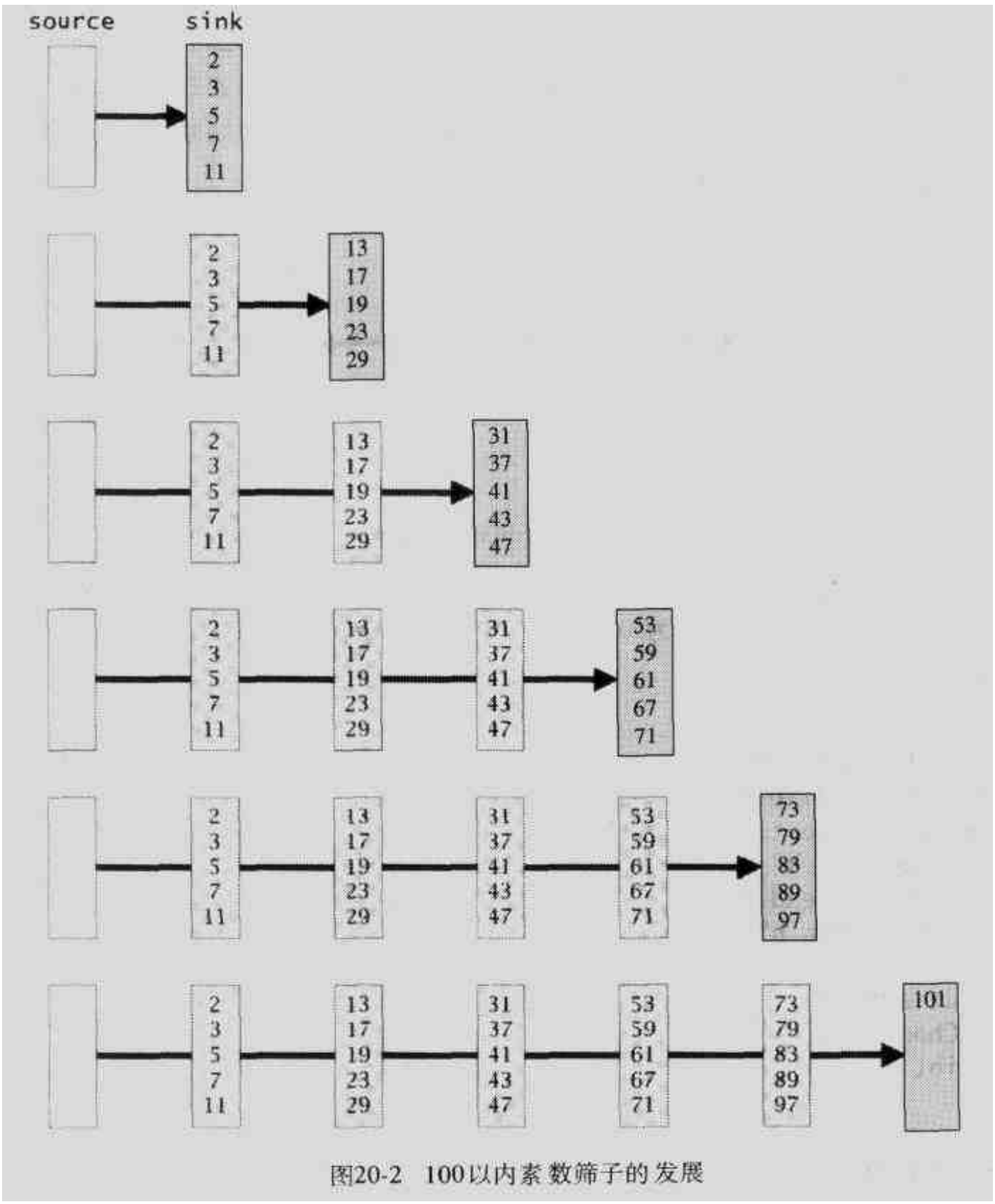


图20-2 100以内素数筛子的发展

428

```
⟨sieve functions 429⟩≡
  int source(void *cl) {
      struct args *p = cl;
      int i = 2;
```

```
        if (Chan_send(p->c, &i, sizeof i))
            for (i = 3; Chan_send(p->c, &i, sizeof i); )
                i += 2;
        return EXIT_SUCCESS;
    }
```

只要接收方接受，source就将2及随后的奇数发送过来。一旦sink打印出所有的素数，就从它的输入通道读取0字节内容，给上流过滤器发送信号通知它作业已经完成且终止了。每个filter的工作都相同，直到source监听到它的接收方读取0字节内容并终止执行。

filter从它的输入通道读取整数并向它的输出通道写入潜在的素数，直到接收这些可能素数的线程不能再接收为止：

```
⟨sieve functions 429⟩+≡
  void filter(int primes[], Chan_T input, Chan_T output) {
      int j, x;

      for (;;) {
          Chan_receive(input, &x, sizeof x);
          ⟨x is a multiple of primes[0...] 429⟩
          if (primes[j] == 0)
              if (Chan_send(output, &x, sizeof x) == 0)
                  break;
      }
      Chan_receive(input, &x, 0);
  }
```

primes[0..n-1]保存有关filter的素数。这个数组以0结束，因此搜索循环向下遍历primes，直到它确定x不是素数或遇到结束符号：

```
⟨x is a multiple of primes[0...] 429⟩≡
  for (j = 0; primes[j] != 0 && x%primes[j] != 0; j++)
      ;
```

429

正如上述代码所表明的，搜索结束于终止符0时就失败了。这种情况下，x可能是个素数，因此还要把它送入输出通道发送给另一filter或发送给sink。

所有的行为都在sink中；args的c字段保存了sink的输入通道；n字段给出了每个filter的素数数量；last字符保存N，这是期望的素数范围；sink初始化它的primes数组并监听它的输入：

```
⟨sieve functions 429⟩+≡
  int sink(void *cl) {
      struct args *p = cl;
      Chan_T input = p->c;
      int i = 0, j, x, primes[256];

      primes[0] = 0;
      for (;;) {
          Chan_receive(input, &x, sizeof x);
          ⟨x is a multiple of primes[0...] 429⟩
          if (primes[j] == 0) {
              ⟨x is prime 430⟩
          }
```

```
        }
        Fmt_print("\n");
        Chan_receive(input, &x, 0);
        return EXIT_SUCCESS;
    }
```

如果x不是primes中一个非零数值的倍数，那它就是个素数，然后sink就将它打印出来并添加到primes中。

```
⟨x is prime 430⟩≡
  if (x > p->last)
      break;
  Fmt_print(" %d", x);
  primes[i++] = x;
  primes[i] = 0;
  if (i == p->n)
      ⟨spawn a new sink and call filter 431⟩
```

430 x大于p->last时，所有期望的素数都被打印出来了，且sink可以结束了。这之前，它会等待来自于其输入通道中的多个整数，但是读取0字节，即给上流线程发信号通知计算完成。

sink收集到n个素数后就把自身克隆一下，并变成还需要一个新通道的filter：

```
⟨spawn a new sink and call filter 431⟩≡
  {
      p->c = Chan_new();
      Thread_new(sink, p, sizeof *p, NULL);
      filter(primes, input, p->c);
      return EXIT_SUCCESS;
  }
```

新通道变成副本的输入通道及filter的输入通道。sink的输入通道即filter的输入通道。filter返回时它的线程就退出了。

sieve中线程之间的所有切换都发生在Chan_send和Chan_receive中，而且总是至少有一个线程可以运行。这样，sieve可以采用有优先权调度运行，也可以采用无优先权调度；它是将线程主要用于构造一个应用程序的简单范例。无优先权线程常称为协同程序。

20.3 实现

Chan的实现完全可以紧接着Sem的实现建立，因此它是独立于机器的。Sem也是独立于机器的，但它还依靠Thread实现的内部结构．因此Thread也实现了Sem。单处理器的Thread实现很大程度上可以既独立于主机也独立于其操作系统。如下详细介绍，机器和操作系统的相关性开始蔓延到仅用于上下文切换和优先权的代码。

20.3.1 同步通信通道

Chan_T是指向一个具有三个信号量、一个消息指针及一个字节数的结构的指针：

```
⟨chan.c⟩≡
  #include <string.h>
  #include "assert.h"
```

431

```
#include "mem.h"
#include "chan.h"
#include "sem.h"

#define T Chan_T
struct T {
    const void *ptr;
    int *size;
    Sem_T send, recv, sync;
};
```

⟨*chan functions* 432⟩

创建新线程的时候，ptr和size字段还未定义，信号量send、recv和sync的计数器分别被初始化为1、0和0：

```
⟨chan functions 432⟩≡
  T Chan_new(void) {
      T c;

      NEW(c);
      Sem_init(&c->send, 1);
      Sem_init(&c->recv, 0);
      Sem_init(&c->sync, 0);
      return c;
  }
```

信号量send和recv控制对ptr和size和访问，信号量sync确保消息传递按Chan接口所指定的那样同步。线程通过填充ptr和size字段发发送消息，但仅在这样做很安全时这样做。发送方可以设置ptr和size时send为1；否则为0——例如，在接收方接收消息之前。类似地，ptr和size具有指向一条消息及其大小的有效指针时recv为1；否则为0——例如，在发送方设置了ptr和size之前。send和recv持续周期性地波动：recv为0时send为1；反之亦然。接收方成功将消息拷贝到它的专用缓冲区后，sync为1。

Chan_send通过等待send、填充ptr和size、给recv发信号以及等待sync发送消息：

```
⟨chan functions 432⟩+≡
  int Chan_send(Chan_T c, const void *ptr, int size) {
      assert(c);
      assert(ptr);
      assert(size >= 0);
      Sem_wait(&c->send);
      c->ptr = ptr;
      c->size = &size;
      Sem_signal(&c->recv);
      Sem_wait(&c->sync);
      return size;
  }
```

c->size保存指向字节数的指针，这样接收方可以修改那个计数，由此通知发送方传递了多少字节。Chan_receive执行的这三个步骤补充了Chan_send完成的那些内容。Chan_receive通过等待recv、将消息复制到它的参数缓冲区并修改字节数以及给sync和send发送消息：

```
⟨chan functions 432⟩+≡
  int Chan_receive(Chan_T c, void *ptr, int size) {
      int n;

      assert(c);
      assert(ptr);
      assert(size >= 0);
      Sem_wait(&c->recv);
      n = *c->size;
      if (size < n)
          n = size;
      *c->size = n;
      if (n > 0)
          memcpy(ptr, c->ptr, n);
      Sem_signal(&c->sync);
      Sem_signal(&c->send);
      return n;
  }
```

433 n是实际接收到的字节数，可能是0。这段代码处理所有三种情况：发送方的size超出接收方的size、两方的size相等以及接收方的size超出了发送方的size。

20.3.2 线程

Thread实现，thread.c，实现了Thread和Sem接口：

```
⟨thread.c⟩≡
  #include <stdio.h>
  #include <stdlib.h>
  #include <string.h>
  #include </usr/include/signal.h>
  #include <sys/time.h>
  #include "assert.h"
  #include "mem.h"
  #include "thread.h"
  #include "sem.h"

  void _MONITOR(void) {}
  extern void _ENDMONITOR(void);

  #define T Thread_T
  ⟨macros 436⟩
  ⟨types 435⟩
  ⟨data 435⟩
  ⟨prototypes 439⟩
  ⟨static functions 436⟩
  ⟨thread functions 438⟩
  #undef T

  #define T Sem_T
  ⟨sem functions 457⟩
  #undef T
```

空函数_MONITOR和外部函数_ENDMONITOR仅用于它们的地址。如下所述，这些地址包括临界区——千万不能中断的线程代码。这段代码中有少量是用汇编语言编写的，而且在这个汇编语言文件的末尾定义了_ENDMONITOR，这样临界区也包括这段汇编代码。它的名称以下划线开始，因为那是这里使用的实现定义汇编语言名称的惯例。 434

线程句柄是指向Thread_T结构的一个隐式指针，包含了确定线程状态的所有必要信息。这个结构常称为线程控制模块。

```
⟨types 435⟩≡
  struct T {
      unsigned long *sp;        /* must be first */
      ⟨fields 435⟩
  };
```

最初的字段保存依赖机器和操作系统的数值。这些字段首先出现在Thread_T结构中，因为是汇编语言代码访问它们。首先放置它们可以更容易地访问这些字段，而且还可以添加新字段而不用改变现有汇编语言代码。只有一个字段sp，大多数机器都需要，它保存了线程的堆栈指针。

大多数线程操作都围绕着将线程排入队列及从队列中移出。Thread和Sem接口设计用于维护一个简单的不变量：线程没在队列中或只在一个队列中，这种设计可以避免为队列入口分配任何空间。比方说，不使用Seq_T表示队列，相反用Thread_T结构的循环链接列表表示队列。就绪队列就是个例子，它保存没有获得处理器的运行线程：

```
⟨data 435⟩≡
  static T ready = NULL;
```

```
⟨fields 435⟩≡
  T link;
  T *inqueue;
```

图20-3显示了就绪队列中按A、B和C的顺序排列的三个线程。ready指向队列中的最后一个线程C；队列通过link字段链接在一起。每个Thread_T结构的inqueue字段指向队列变量——这里是ready——并用于删除队列中的线程。队列变量为空时队列为空，正如ready的初值所表明以及宏 435

```
⟨macros 436⟩≡
  #define isempty(q) ((q) == NULL)
```

测试的那样。

如果线程t位于队列中，那么t->link和t->inqueue就不为空；否则，这两个字段为空。下面的队列函数使用了包含link和inqueue的断言以确保上述不变量有效。例如，put把一个线程添加到一个空队列或非空队列的末尾：

```
⟨static functions 436⟩≡
  static void put(T t, T *q) {
      assert(t);
      assert(t->inqueue == NULL && t->link == NULL);
      if (*q) {
          t->link = (*q)->link;
          (*q)->link = t;
```

```
        } else
            t->link = t;
        *q = t;
        t->inqueue = q;
    }
```

这样，put(t, &ready)把t添加到就绪队列末尾。put接收队列变量的地址，这样它就可以进行修改：调用put(t, &q)后，q等于t且t->inqueue等于&q。

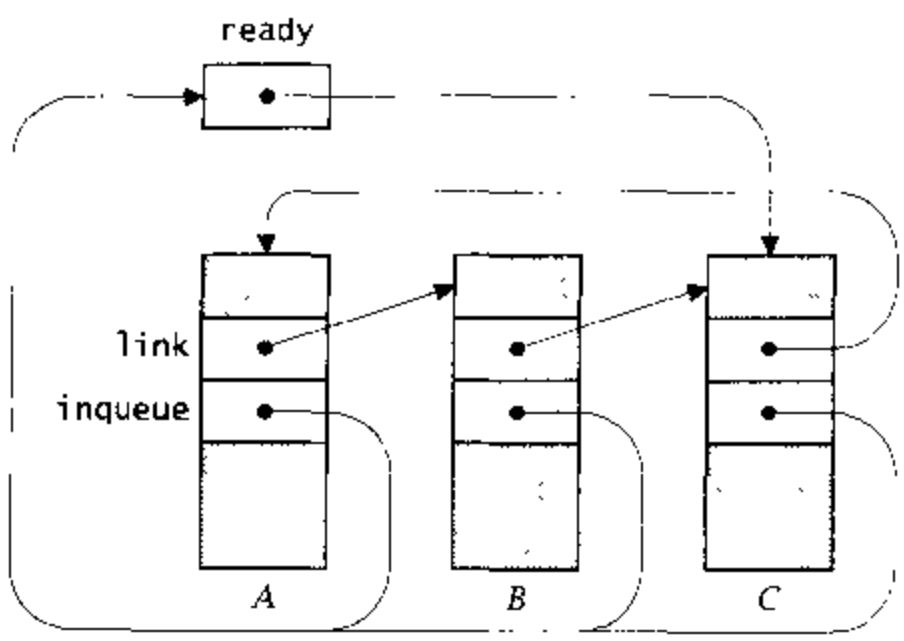


图20-3 就绪队列中的三个线程

get从给定队列中删除第一个元素：

```
⟨static functions 436⟩+≡
  static T get(T *q) {
      T t;

      assert(!isempty(*q));
      t = (*q)->link;
      if (t == *q)
          *q = NULL;
      else
          (*q)->link = t->link;
      assert(t->inqueue == q);
      t->link = NULL;
      t->inqueue = NULL;
      return t;
  }
```

这段代码使用inqueue字段确保线程的确在q中，而且它清除了link和inqueue字段将线程标记为不在任何队列。

第三个也是最后一个队列函数从队列中线程出现的位置删除该线程：

```
⟨static functions 436⟩+≡
  static void delete(T t, T *q) {
      T p;

      assert(t->link && t->inqueue == q);
      assert(!isempty(*q));
      for (p = *q; p->link != t; p = p->link)
```

```
        ;
    if (p == t)
        *q = NULL;
    else {
        p->link = t->link;
        if (*q == t)
            *q = p;
    }
    t->link = NULL;
    t->inqueue = NULL;
}
```

436 ~ 437

第一个断言确保t在q中；第二个确保队列非空，t在队列中后队列必须非空。if语句处理t是q中惟一线程的情况。

Thread_init创建“根”线程（根线程的Thread_T结构是静态分配的）：

```
⟨thread functions 438⟩≡
  int Thread_init(int preempt, ...) {
      assert(preempt == 0 || preempt == 1);
      assert(current == NULL);
      root.handle = &root;
      current = &root;
      nthreads = 1;
      if (preempt) {
          ⟨initialize preemptive scheduling 454⟩
      }
      return 1;
  }
```

```
⟨data 435⟩+≡
  static T current;
  static int nthreads;
  static struct Thread_T root;
```

```
⟨fields 435⟩+≡
  T handle;
```

current是当前占据处理器的线程；nthreads是已有线程数。Thread_new逐1递增nthreads；Thread_exit逐1递减。handle字段仅指向线程句柄并帮助检查句柄的有效性：仅当t等于t->handle时t确定一个已有线程。

如果current为空，就还没有调用Thread_init，因此，如上所示，对空current的检测实现了Thread_init必须只调用一次的可检查的运行期错误。检查其他Thread和Sem函数中的非空current实现了Thread_init必须在其他任何Thread、Sem和Chan函数之前调用的可检查的运行期错误。Thread_self是个范例，它仅返回current：

438

```
⟨thread functions 438⟩+≡
  T Thread_self(void) {
      assert(current);
      return current;
  }
```

在线程之间切换需要一个依赖于机器的代码，原因有很多，例如每个线程都有它自己的堆栈和异常状态。上下文开关原语有众多可能的设计，所有这些都相对很简单，因为它们都是全部或部分用汇编语言编写的。Thread实现使用了具体于实现的单一原语

```
⟨prototypes 439⟩≡
  extern void _swtch(T from, T to);
```

将上下文从从线程from切换到线程to，这里from和to是指向Thread_T结构的指针。_swtch与setjmp和longjmp一样：当线程A调用_swtch时，假定控制转移给线程B。当B调用_swtch让A重新开始时，A对_swtch的调用就返回了。这样，A和B都把_swtch完全当作另一函数调用。这个简单的设计也是利用了机器的调用顺序，这非常有帮助，例如它有助于在A切换到B时保存A的状态。惟一的不足就是创建的新线程必须具有一个状态，看上去好像调用了_swtch，因为第一次它运行的时候会被作为_swtch中return的结果。

_swtch只能在一个地方调用，即静态函数run：

```
⟨static functions 436⟩+≡
  static void run(void) {
      T t = current;

      current = get(&ready);
      t->estack = Except_stack;
      Except_stack = current->estack;
      _swtch(t, current);
  }
```

439

```
⟨fields 435⟩+≡
  Except_Frame *estack;
```

run从当前执行的线程切换到就绪队列的顶部线程。它将最上面的线程从ready中删除，设置current，并切换到新线程。estack字段保存着指向位于线程异常堆栈顶部的异常结构的指针，而且run注意更新Except的全局Except_stack，4.2节页介绍了这个变量。

可以引起上下文切换的所有Thread和Sem函数都调用run，而且它们在调用run之前将当前线程置入ready或另一适当的队列。Thread_pause是个最简单的例子：它将current置入ready，然后调用run。

```
⟨thread functions 438⟩+≡
  void Thread_pause(void) {
      assert(current);
      put(current, &ready);
      run();
  }
```

如果只有一个线程在运行，Thread_pause就把它置入ready，而run将其删除并切换到这个线程。这样，_swtch(t, t)必须正常工作。图20-4描述了执行如下调用的线程A、B与C之间的上下文开关，假定A最先占有处理器且ready按顺序保存B和C。

A	*B*	*C*
Thread_pause()	Thread_pause()	Thread_pause()
Thread_join(C)	Thread_exit(0)	Thread_exit(0)
Thread_exit(0)		

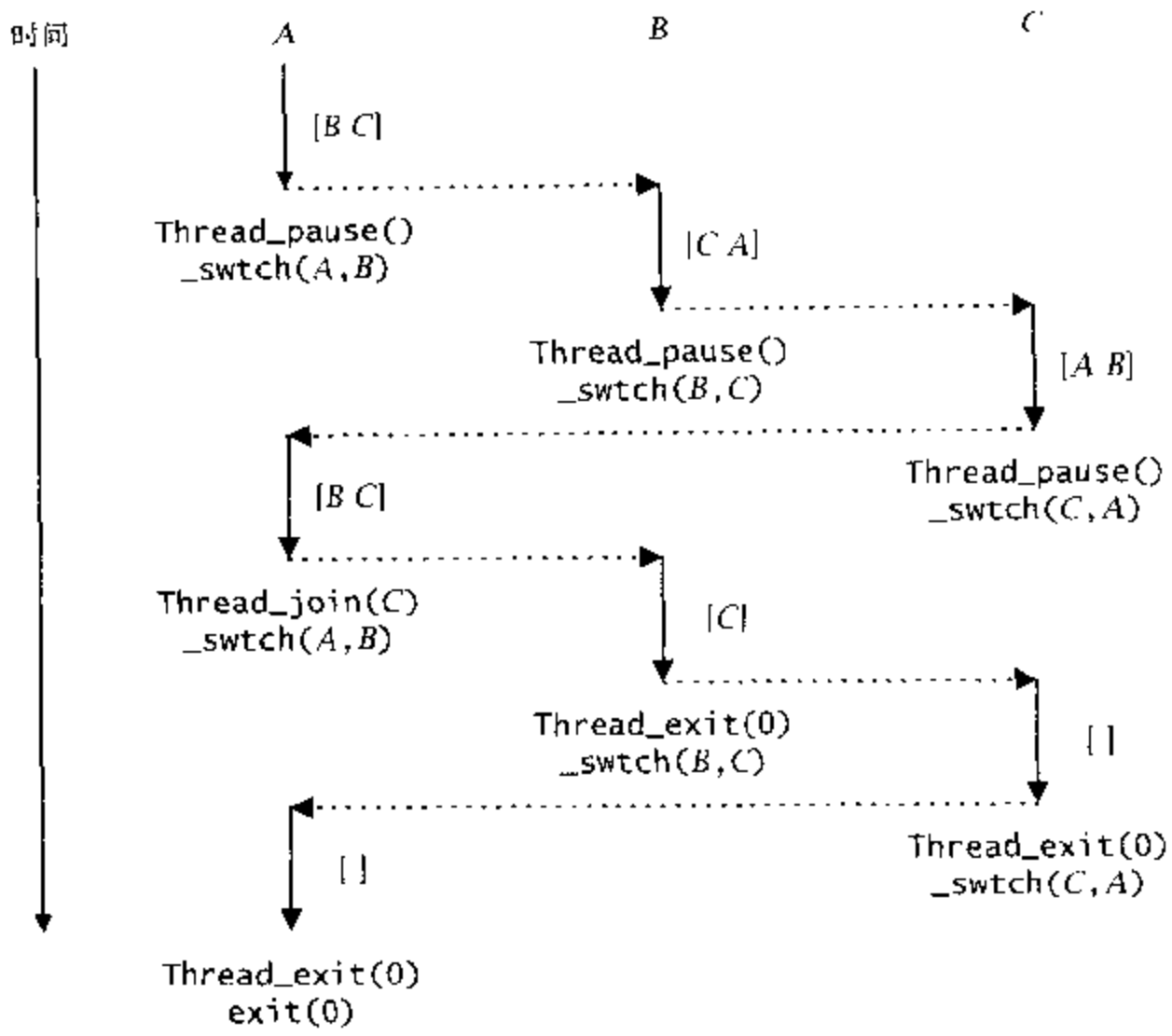


图20-4　三个线程之间的上下文切换

图20-4中的垂直实线箭头表示的是每个线程都拥有处理器时的情况，水平虚线箭头是上下文开关；就绪队列显示在实线箭头旁边的括号里。Thread函数及其引起的_swtch调用出现每个上下文开关的下面。

当A调用Thread_pause时，就将它放入ready，并删除B，然后B获得处理器。当B运行的时候，ready保存C、A。当B调用Thread_pause时，就从ready中删除C，然后C获得处理器，ready保存A、B。C调用Thread_pause之后，ready再次保存B、C，而A正在运行。当A调用Thread_join(C)，它因C的结束而阻塞，因此处理器交付给ready中的第一位线程B。

此时，ready中只有C，因为A位于C的相关队列中。当B调用Thread_exit时，run切换到C且ready为空。C调用Thread_exit而结束运行，这使得A作为C的结束结果被放回到ready中。这样，当Thread_exit调用run时，A获得处理器。但是，A调用Thread_exit不会引起上下文转换：A是系统中的惟一线程，隐藏Thread_exit调用exit。

当ready为空且调用了run时发生死锁；也就是说，没有运行中的线程。死锁是个可检查的运行期错误，当空的就绪队列调用get时能够检测到死锁。

440
↓
441

Thread_join和Thread_exit描述了包含“加入队列”的队列操作和就绪队列。有两种样式的Thread_join：Thread_join(t)等待线程t结束并返回t的退出码——传递给Thread_exit的值t；t一定不能是调用线程。Thread_join(NULL)等待所有线程结束并返回0；只有一个线程可以调用Thread_join(NULL)。

```
⟨thread functions 438⟩+≡
  int Thread_join(T t) {
```

```
        assert(current && t != current);
        testalert();
        if (t) {
            ⟨wait for thread t to terminate 442⟩
        } else {
            ⟨wait for all threads to terminate 443⟩
            return 0;
        }
    }
```

如下所述，如果调用线程被警告，testalert就会引发Thread_Alerted。当t非空并且指的是一个已有线程，调用线程就把自身放入t的合并队列中等待它的消亡；否则，Thread_join立刻返回-1。

```
⟨wait for thread t to terminate 442⟩≡
  if (t->handle == t) {
      put(current, &t->join);
      run();
      testalert();
      return current->code;
  } else
      return -1;

⟨fields 435⟩+≡
  int code;
  T join;
```

只有t->handle等于t时t才表示已有线程。如下所示，当线程结束时Thread_exit清除handle字段。当t结束时，Thread_exit在把那些线程移到就绪队列的同时，将它的参数保存在t->join中每个Thread_T的code字段中。这样，当那些线程再次执行的时候，就可以方便地获得那个退出码了。

442

当t为空时，调用线程放入join0，join0保存着等待其他所有线程结束的那个惟一线程：

```
⟨wait for all threads to terminate 443⟩≡
  assert(isempty(join0));
  if (nthreads > 1) {
      put(current, &join0);
      run();
      testalert();
  }

⟨data 435⟩+≡
  static T join0;
```

下次调用线程运行的时候，它就是惟一存在的线程。这段代码也处理调用线程是系统中惟一线程时的情况，nthreads为1时发生这种情况。

Thread_exit需要完成大量的工作：它必须释放与调用线程相关的资源，重新运行等待调用线程结束的线程并安排它们获取退出码，以及检查调用线程是否为系统的倒数第二个或最后一个线程。

```
⟨thread functions 438⟩+≡
  void Thread_exit(int code) {
      assert(current);
      release();
```

```
        if (current != &root) {
            current->next = freelist;
            freelist = current;
        }
        current->handle = NULL;
        ⟨resume threads waiting for current's termination 444⟩
        ⟨run another thread or exit 444⟩
    }

⟨fields 435⟩+≡
    T next;
⟨data 435⟩+≡
    static T freelist;
```

443

对release的调用及把current附加到freelist末尾的那段代码共同合作释放调用线程的资源，下面将详细叙述。如果调用线程为根线程，它的存储空间一定不能释放，因为那部分空间是静态分配的。

清除handle字段就把线程标记为不存在的，而且那些等待消亡的线程现在可以重新开始了：

```
⟨resume threads waiting for current's termination 444⟩≡
    while (!isempty(current->join)) {
        T t = get(&current->join);
        t->code = code;
        put(t, &ready);
    }
```

调用线程的退出码被复制到等待线程中Thread_T结构的code字段，这样就可以释放current了。

如果仅有两个线程且其中一个在join0中，那么那个等待线程现在可以重新开始了。

```
⟨resume threads waiting for current's termination 444⟩+≡
    if (!isempty(join0) && nthreads == 2) {
        assert(isempty(ready));
        put(get(&join0), &ready);
    }
```

断言有助于检测维护nthreads和ready过程中的错误：如果join0非空且nthreads为2，那么ready必须为空，因为两个现存线程中的一个位于join0，且另一个在执行Thrcad_exit。

Thread_exit以nthreads递减并调用库函数exit或运行另一线程作为结束：

```
⟨run another thread or exit 444⟩≡
    if (--nthreads == 0)
        exit(code);
    else
        run();
```

444

Thread_alert在线程的Thread_T结构中设置标志，而且，如果线程位于队列中，就从队列中删除它，从而将线程标记为“alerted”。

```
⟨thread functions 438⟩+≡
    void Thread_alert(T t) {
        assert(current);
        assert(t && t->handle == t);
        t->alerted = 1;
```

```
        if (t->inqueue) {
            delete(t, t->inqueue);
            put(t, &ready);
        }
    }
```

⟨*fields* 435⟩+≡
```
    int alerted;
```

Thread_alert本身不能引发Thread_Alerted，因为调用线程的状态与t不同。线程必须引发Thread_Alerted并自己解决。这就是testalert的目的：

⟨*static functions* 436⟩+≡
```
    static void testalert(void) {
        if (current->alerted) {
            current->alerted = 0;
            RAISE(Thread_Alerted);
        }
    }
```

⟨*data* 435⟩+≡
```
    const Except_T Thread_Alerted = { "Thread alerted" };
```

无论何时，只要线程被阻塞或阻塞之后重新开始，就调用testalert。已述Thread_join开始的调用testalert说明了前面那种情况；后面一种情况常发生在调用run之后，代码块<*wait for thread t to terminate 442*>和<*wait for all threads to terminate 443*>中的testalert调用对这种情况进行了说明。Sem_wait和Sem_signal中出现了类似的用法；参见20.3.5节。

445

20.3.3 线程创建与上下文转换

最后一个Thread函数是Thread_new。一些Thread_new是依赖于机器的，因为它与_swtch交互，但是大多数几乎都是独立于机器的。Thread_new有四项任务：为新线程分配资源；初始化新线程的状态，这样它就可以通过_swtch的返回开始运行；递增nthreads以及将新线程附加到ready末尾。

⟨*thread functions* 438⟩+≡
```
T Thread_new(int apply(void *), void *args,
    int nbytes, ...) {
    T t;

    assert(current);
    assert(apply);
    assert(args && nbytes >= 0 || args == NULL);
    if (args == NULL)
        nbytes = 0;
    ⟨allocate resources for a new thread 446⟩
    t->handle = t;
    ⟨initialize t's state 449⟩
    nthreads++;
    put(t, &ready);
    return t;
}
```

在这个Thread的单处理器实现中，一个线程需要的惟一资源就是Thread_T结构和一个堆栈。Thread_T结构和一个16K字节的堆栈都是在一次调用Mem的ALLOC中分配的：

```
⟨allocate resources for a new thread 446⟩≡
  {
      int stacksize = (16*1024+sizeof (*t)+nbytes+15)&~15;
      release();
      ⟨begin critical region 447⟩
      TRY
          t = ALLOC(stacksize);
          memset(t, '\0', sizeof *t);
      EXCEPT(Mem_Failed)
          t = NULL;
      END_TRY;
      ⟨end critical region 447⟩
      if (t == NULL)
          RAISE(Thread_Failed);
      ⟨initialize t's stack pointer 448⟩
  }

⟨data 435⟩+≡
  const Except_T Thread_Failed =
      { "Thread creation failed" };
```

446

这段代码比较复杂，因为它必须维护几个不变量，其中最重要的是一定不能中断Thread函数的调用。两种机制共同合作维护这个不变量：一个如下所述，处理控制位于Thread函数中时的中断；另一个机制处理当控制位于Thread函数调用的例程时的中断，对ALLOC和memset的调用就说明了这种情况。这些各种各样的调用都被通过递增和递减critical确定临界区的代码块括在一起：

```
⟨begin critical region 447⟩≡
  do { critical++;

⟨end critical region 447⟩≡
  critical--; } while (0);

⟨data 435⟩+≡
  static int critical;
```

如20.3.4节中所示，忽略critical非零时发生的中断。

Thread_new必须捕捉Mem_Failed，并在完成临界区之后引发它的异常Thread_Failed。如果它没有捕捉这个异常，就会将控制传递给调用者的异常管理者，并将critical设为永不递减的一个正数。

Thread_new假定堆栈向低端地址发展，并像图20-5中所示那样初始化sp字段；顶部的阴影框是Thread_T结构，底部的那个是args的副本以及初始帧，如下所示。

447

```
⟨initialize t's stack pointer 448⟩≡
  t->sp = (void *)((char *)t + stacksize);
  while (((unsigned long)t->sp)&15)
      t->sp--;
```

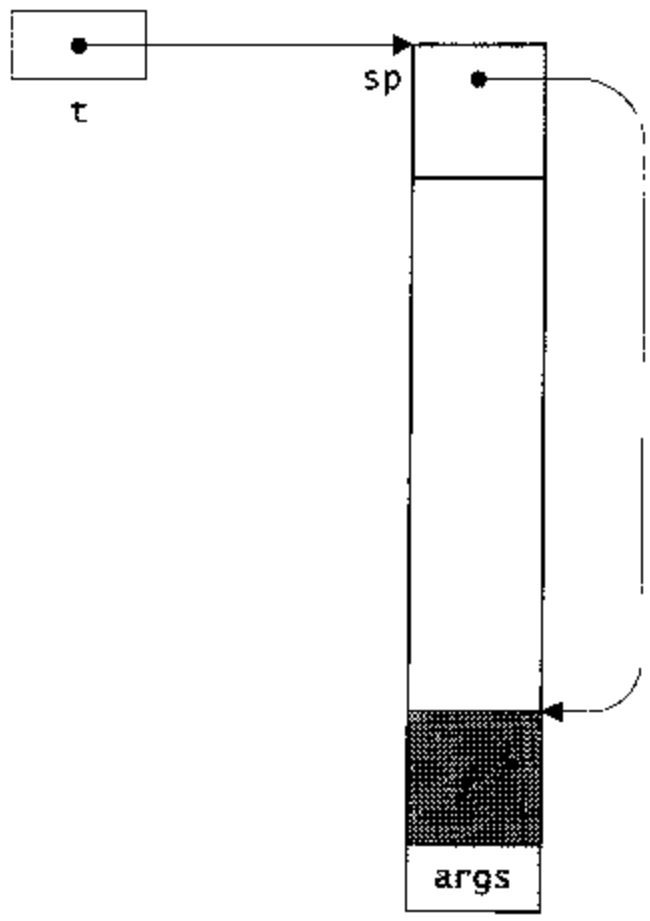


图20-5 Thread_T结构与堆栈的分配

赋予stacksize的值以及这段代码表明，Thread_new初始化堆栈指针，将它的序列界限设置为16字节，这对大多数平台都适用。大多数机器都需要4字节或8字节的堆栈序列，但是DEC ALPHA需要16字节的序列。

Thread_new从调用release开始，Thread_exit也调用这个函数。Thread_exit不能释放当前线程的堆栈，因为它正在使用这个堆栈。因此，它把这个线程句柄添加到freelist，并将释放工作延迟到下次调用release的时候：

```
⟨static functions 436⟩+≡
  static void release(void) {
448       T t;
      ⟨begin critical region 447⟩
      while ((t = freelist) != NULL) {
          freelist = t->next;
          FREE(t);
      }
      ⟨end critical region 447⟩
  }
```

release的使用更常见，不只是在需要的时候：freelist仅有一个元素，因为Thread_exit和Thread_new都要调用release。如果仅有Thread_new调用过release，死亡的Thread_T就会在freelist堆积。release使用了临界区，因为它要调用Mem的FREE。

其次，Thread_new初始化新线程的堆栈以保存自args开始的nbytes字节副本以及要将线程显示成好像已经调用过_swtch所需要的结构帧；后者的初始化依赖于机器。

```
⟨initialize t's state 449⟩≡
  if (nbytes > 0) {
      t->sp -= ((nbytes + 15U)&~15)/sizeof (*t->sp);
      ⟨begin critical region 447⟩
```

```
        memcpy(t->sp, args, nbytes);
        ⟨end critical region 447⟩
        args = t->sp;
    }
    #if alpha
    { ⟨initialize an ALPHA stack 463⟩ }
    #elif mips
    { ⟨initialize a MIPS stack 461⟩ }
    #elif sparc
    { ⟨initialize a SPARC stack 452⟩ }
    #else
    Unsupported platform
    #endif
```

图20-5所示的堆栈底部描述了这些初始化的结构：较暗的阴影表示的是依赖于机器的结构帧；较浅的阴影是args的副本。thread.c和swtch.s是本书中使用条件编译的惟一模块。 449

融会贯通_swtch的汇编语言实现之后，堆栈的初始化就更易于理解了：

```
⟨swtch.s⟩≡
  #if alpha
  ⟨ALPHA swtch 462⟩
  ⟨ALPHA startup 463⟩
  #elif sparc
  ⟨SPARC swtch 450⟩
  ⟨SPARC startup 452⟩
  #elif mips
  ⟨MIPS swtch 460⟩
  ⟨MIPS startup 461⟩
  #else
  Unsupported platform
  #endif
```

_swtch（from, to）必须保存from的状态，恢复to的状态，并从to的最近一次调用返回到_swtch以继续执行to。调用规范保存了大部分状态，因为通常它们规定必须保存调用中的一些寄存器值，但不用保存条件码这样一些机器状态信息。因此，_swtch仅保存了它需要但不受调用规范支持的状态——例如返回地址——而且它可以在调用线程的堆栈中保存这些值。

SPARC_swtch可能是最容易的，因为SPARC调用规范要给每个函数提供它自己的“寄存器窗口”以保存所有的寄存器；它惟一必须保存的寄存器是帧指针及返回地址。

```
⟨SPARC swtch 450⟩≡
        .global __swtch
        .align  4
        .proc   4
1   __swtch:save    %sp,-(8+64),%sp
2           st      %fp,[%sp+64+0]  ! save from's frame pointer
3           st      %i7,[%sp+64+4]  ! save from's return address
4           ta      3               ! flush from's registers
5           st      %sp,[%i0]       ! save from's stack pointer
6           ld      [%i1],%sp       ! load to's stack pointer
7           ld      [%sp+64+0],%fp  ! restore to's frame pointer
```

450

```
8           ld      [%sp+64+4],%i7  ! restore to's return address
9           ret                     ! continue execution of to
10          restore
```

上面的行编号用于确定下面具体说明所针对的非样本行，它们不属于汇编语言代码。根据管理，汇编语言名称都以下划线作为前缀，因此在SPARC的汇编语言中_swtch就称为_swtch。

图20-6 显示了_swtch的帧结构；所有的SPARC帧顶部至少有64字节供操作系统在必要的时候存储寄存器窗口。_swtch72字节帧中的另两个字存放的是已有的帧指针和返回地址。

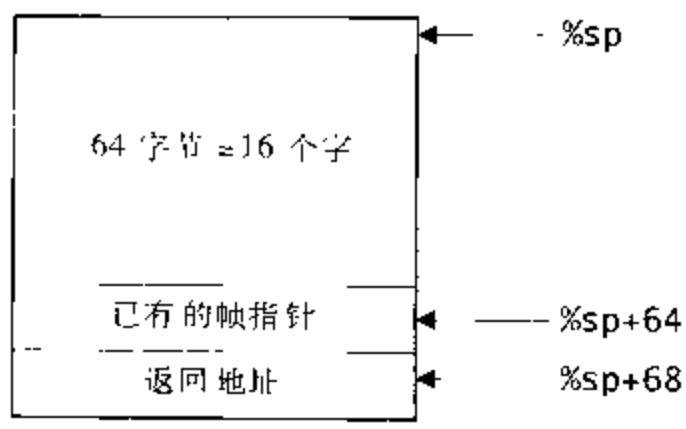


图20-6 _swtch堆栈帧的布局

_swtch中行1为_swtch分配了堆栈帧。行2和行3保存from的帧指针（%fp）并返回新帧中第17和18个32位字（偏移量位64和68）中的地址（%i7）。行4执行了一个系统调用将from的寄存器窗口“冲刷”到堆栈，为了能够继续运行to的寄存器窗口这是必要的。这个调用效果不好：用户级线程几个假定的优点之一就是上下文切换不需要核心干预。但是在SPARC是只有核心可以刷洗寄存器窗口。

行5将from的堆栈指针保存在它Thread_T结构的sp字段中。该指令说明了为什么那个字段要在第一位：这段代码独立于Thread_T的大小及其他字段的位置。行6是斜体字因为它是实际的上下文开关。这条指令把to的堆栈指针装载到%sp，堆栈指针寄存器。然后，_swtch在to
451 的堆栈上执行。行7和行8恢复to的帧指针并返回地址，因为%sp现在指向to堆栈的顶端。行9和行10包含了正常的函数返回顺序，且控制继续位于to最后一次调用_swtch时保存的地址。

Thread_new必须为_swtch创建帧，这样其他一些对_swtch的调用可以正确返回并开始运行新的线程；该运行必须调用apply。图20-7显示了Thread_new所建立的内容：_swtch的帧在堆栈顶端；帧下面就是如下的启动代码。

```
⟨SPARC startup 452⟩≡
        .global __start
        .align  4
        .proc   4
1   __start:ld      [%sp+64+4],%o0
2           ld      [%sp+64],%o1
3           call    %o1; nop
4           call    _Thread_exit; nop
5           unimp   0
        .global __ENDMONITOR
    __ENDMONITOR:
```

_swtch帧中的返回地址指向_start，启动帧存放着apply和args，如图20-7所示。_swtch第

次返回时，控制到达_start（这是汇编代码中的_start）。启动代码的行1将args装载到%o0，根据SPARC调用规范，%o0用于传递第一个参数。行2把apply的地址装载到%o1，其他时候都不使用%o1；行3间接调用了apply。如果apply返回，它的退出码就会在%o0中，这样其中的值就会传给从不返回的Thread_exit。行5应当永不执行；如果执行就会引起一个错误。下面就_ENDMONITOR进行了说明。

_swtch和_start中的这15行汇编语言是SPARC上需要的全部内容；如图20-7中所示那样初始化新线程的堆栈可以完全用C实现。如下所示，这两个帧都是从下往上建立的。

⟨*initialize a SPARC stack* 452⟩≡

```
1   int i; void *fp; extern void _start(void);
2   for (i = 0; i < 8; i++)
3       *--t->sp = 0;
4   *--t->sp = (unsigned long)args;
5   *--t->sp = (unsigned long)apply;
6   t->sp -= 64/4;
7   fp = t->sp;
8   *--t->sp = (unsigned long)_start - 8;
9   *--t->sp = (unsigned long)fp;
10  t->sp -= 64/4;
```

452

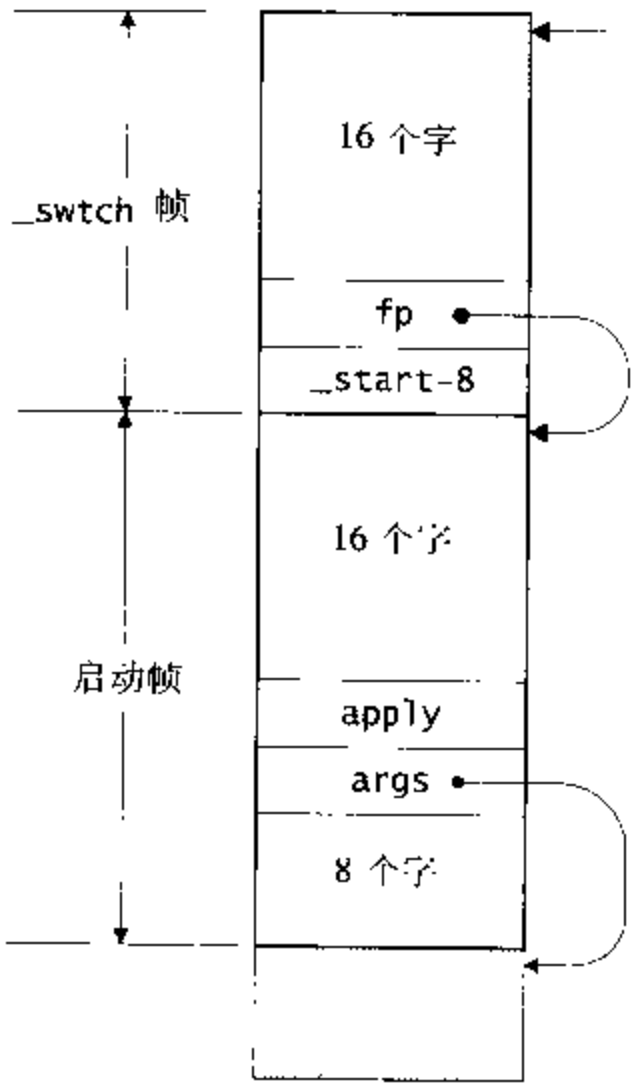


图20-7　SPARC上的启动和初始_swtch帧

行2和行3在启动帧底部创建了8个字。行4和行5把args和apply的值推入堆栈；行6在启动帧的顶部分配了64字节。这个地方的堆栈指针是_swtch必须恢复的帧指针，因此行7把这个值保存在fp中。行8将返回地址——%i7中保存的值推入堆栈。返回地址是_start前面的8个字节，

453 因为SPARC ret指令在返回时向%i7中的地址加了8。行9将%fp中保存的值推入堆栈；行10以_swtch帧顶端的64个字节作为结束。

如果apply是个接收不定量参数的函数，它的入口顺序就将%o0至%o5中的值保存到堆栈调用方帧即启动帧的偏移64至88的位置。行2和行3用于分配这段空间及另外的8个字节，这样堆栈指针保持按8字节的界限排列，参见SPARC硬件说明。

MIPS和ALPHA版本的_swtch和_start将在20.3.6中介绍。

20.3.4 抢占

抢占相当于定期隐式调用Thread_pause。依赖于UNIX的Thread抢占实现安排了一个“虚拟的”定时器每50毫秒中断一次，中断管理器就执行等价于Thread_pause的代码。定时器是虚拟的，因为它只在执行进程的时候才工作。Thread_init使用UNIX信号工具初始化定时器的中断信号。第一步把中断管理器与虚拟时钟信号SIGVTALRM关联起来：

```
⟨initialize preemptive scheduling 454⟩≡
  {
      struct sigaction sa;
      memset(&sa, '\0', sizeof sa);
      sa.sa_handler = (void (*)())interrupt;
      if (sigaction(SIGVTALRM, &sa, NULL) < 0)
          return 0;
  }
```

sigaction结构有三个字段：sa_handler是产生SIGVTALRM信号时所调用函数的地址；sa_mask是一个信号集，用于指定处理中断时应当阻塞的其他信号，SIGVTALRM除外；sa_flags提供具体于信号的选项。如下所述，Thread_init把sa_handler设为interrupt，并清空其他字段。

sigaction函数是用于关联管理器与信号的POSIX标准函数。POSIX标准得到了大多数UNIX变体及Windows NT这样的其他一些操作系统的支持。这三个参数都作为符号名，分别表示信
454 号数、指向修改该信号行为的sigaction结构的指针以及指向另一个没有用该信号前一行为进行填充的sigaction结构的指针。当第三个参数为空时，就不再返回前一行为的相关信息了。

如果信号的行为被修改为第二个参数所指定的那样，sigaction函数就返回零；否则返回-1。如果sigaction返回-1，Thread_init则返回零，表明线程系统不能支持有优先权的调度。

一旦信号管理器就位，虚拟定时器就得到了初始化：

```
⟨initialize preemptive scheduling 454⟩+≡
  {
      struct itimerval it;
      it.it_value.tv_sec     =  0;
      it.it_value.tv_usec    = 50;
      it.it_interval.tv_sec  =  0;
      it.it_interval.tv_usec = 50;
      if (setitimer(ITIMER_VIRTUAL, &it, NULL) < 0)
          return 0;
  }
```

itimerval结构的it_value字段以秒（tv_sec）和毫秒（tv_msec）指定下次定时器中断的时间长度。it_interval字段中的数值用于定时器过期时重新设置it_value字段。Thread_init安排定时器中断每50毫秒发生一次。

setitimer函数很像sigaction函数：它的第一个参数指定对哪个定时器的行为起作用（也有实时的定时器）；第二个参数是个指针，指向存放新定时器值的itimerval结构；第三个参数也是个指针，指向获取前一定时器值的itimerval结构，但如果不需要前面的这个值就为空。如果成功设置定时器，setitimer就返回零；否则返回-1。

虚拟定时器过期的时候就调用信号管理器interrupt。interrupt返回的时候就会解除中断，这时定时器就重新开始。interrupt运行相当于Thread_pause的函数，除非当前线程位于临界区中或者位于Thread或Sem函数中的某个地方。

```
⟨static functions 436⟩+≡
  static int interrupt(int sig, int code,
      struct sigcontext *scp) {
      if (critical ||
         scp->sc_pc >= (unsigned long)_MONITOR
      && scp->sc_pc <= (unsigned long)_ENDMONITOR)
          return 0;
      put(current, &ready);
      sigsetmask(scp->sc_mask);
      run();
      return 0;
  }
```

455

sig参数存放信号数；code为一些信号提供另外数据。scp参数是个执行sigcontext结构的指针，这个结构在sc_pc字段中包含了中断时候的单元计数器。thread.c开始于空函数_MONITOR；swtch.s中的汇编语言代码以全局符号_ENDMONITOR的定义作为结束。如果目标文件装载到程序，那么swtch.s的目标代码就紧随thread.c的目标代码，然后如果中断线程的单元计数器在_MONITOR和_ENDMONITOR之间，它就执行一个Thread或Sem函数。这样，如果critical非0，或scp->sc_pc位于_MONITOR和_ENDMONITOR之间，interrupt就返回并忽略这次定时器中断。否则，中断就将当前线程推入ready并运行另一线程。

调用sigsetmask恢复中断禁用的信号集scp->sc_mask中提供的信号；这个集合通常仅存放SIGVTALRM信号。这个调用很必要，因为下个运行的线程可能还没有被中断挂起。例如，假设线程A明确调用了Thread_pause，执行进展到线程B。发生定时器中断时，控制到达interrupt，并禁用了SIGVTALRM信号。B重新启用了SIGVTALRM并将处理器释放给A。

如果省略了sigsetmask的调用，A再次运行的时候SIGVTALRM仍会是禁用的。A被Thread_pause而不是interrupt挂起。下次定时器中断发生时，A被挂起而B继续。这种情况下，调用sigsetmask是多余的，因为B解除了这个中断，恢复了信号掩码。Thread_T结构中的一个标志可用于避免对sigsetmask的不必要调用。

中断管理器的第二个及其后的参数是受系统影响的。大多数UNIX变体支持上述的code和scp参数，但是其他POSIX兼容系统可能会向管理器提供不同的参数。

456

20.3.5 一般信号量

创建和初始化信号量是四个Sem函数中较容易的两个：

```
⟨sem functions 457⟩≡
  T *Sem_new(int count) {
      T *s;

      NEW(s);
      Sem_init(s, count);
      return s;
  }

  void Sem_init(T *s, int count) {
      assert(current);
      assert(s);
      s->count = count;
      s->queue = NULL;
  }
```

Sem_wait和Sem_signal很短小，但是要编写既正确又合理的实现还需要慎重对待。信号量操作在语义上等价于：

```
Sem_wait(s):    while (s->count <= 0)
                    ;
                --s->count;

Sem_signal(s):  ++s->count;
```

这些语法使如下所示的实现非常简明而且正确，但不合理；这些实现也忽略了警告和可检查的运行期错误。

```
void Sem_wait(T *s) {
    while (s->count <= 0) {
        put(current, &s->queue);
        run();
    }
    --s->count;
}

void Sem_signal(T *s) {
    if (++s->count > 0 && !isempty(s->queue))
        put(get(&s->queue), &ready);
}
```

457

这些实现之所以不合理是因为它们允许“资源不足”。假设s初始化为1且线程A和B都执行

```
for (;;) {
    Sem_wait(s);
    ...
    Sem_signal(s);
}
```

假设A位于省略号所表示的临界区，B在s->queue中。当A调用Sem_signal时，B被移到就绪

队列。如果接下来运行B，它对Sem_wait的调用将会返回，且B将进入临界区。但是A可能先调用Sem_wait而独占临界区。如果A在临界区中被取代，B重新开始但发现s->count为零，又回到s->queue。没有某种干预，B会在ready和s->queue之间无限期地循环，而且，更多线程竞争s会更有可能发生饿死。

一种解决方法就是确保线程从s->queue移到ready时得到信号量。当s->count要从0递增到1但还没有真正增加时将线程从s->queue移到ready就可以实现这种方案。类似地，阻塞的线程在Sem_wait中重新开始时不减少s->count。

```
⟨sem functions 457⟩+≡
  void Sem_wait(T *s) {
      assert(current);
      assert(s);
      testalert();
      if (s->count <= 0) {
          put(current, (Thread_T *)&s->queue);
          run();
          testalert();
      } else
          --s->count;
  }

  void Sem_signal(T *s) {
      assert(current);
      assert(s);
      if (s->count == 0 && !isempty(s->queue)) {
          Thread_T t = get((Thread_T *)&s->queue);
          assert(!t->alerted);
          put(t, &ready);
      } else
          ++s->count;
  }
```

458

当s->count为零且线程C移到就绪队列中时，要确保C得到信号量，因为调用Sem_wait的其他线程都因s->count为零而阻塞。但是对于一般信号量，C不会首先得到信号量：如果D在C再次运行之前调用Sem_signal，就为另一线程在C之前获取信号量打开了大门，尽管C也会得到信号量。

警告让Sem_wait很难于理解。如果s中阻塞的线程被警告，它在Sem_wait中对run的调用就会返回，同时还设置了它的alerted标志。这种情况下，Thread_Alert而不是Sem_signal将线程移到ready，这样，它的恢复就与s->count的值无关。这个线程一定不能干扰s，必须清除它的alerted标志，并引发Thread_Alerted。

20.3.6 MIPS和ALPHA上的上下文转换

MIPS与ALPHA版本的_swtch和_start在SPARC版本的设计方面非常类似，但是细节上也有不同的地方。

MIPS版本的_swtch如下所示。帧大小为88字节。$31,48+36($sp)中的存储指令保存“已

有的调用方”浮点及整数寄存器；寄存器31存放返回地址。斜体指令通过装载to的准栈指针
459 切换上下文，而且其后的装载指令恢复to的已有调用方寄存器。

```
⟨MIPS swtch 460⟩≡
    .text
    .globl  _swtch
    .align  2
    .ent    _swtch
    .set    reorder
    _swtch: .frame  $sp,88,$31
            subu    $sp,88
            .fmask  0xfff00000,-48
            s.d     $f20,0($sp)
            s.d     $f22,8($sp)
            s.d     $f24,16($sp)
            s.d     $f26,24($sp)
            s.d     $f28,32($sp)
            s.d     $f30,40($sp)
            .mask   0xc0ff0000,-4
            sw      $16,48+0($sp)
            sw      $17,48+4($sp)
            sw      $18,48+8($sp)
            sw      $19,48+12($sp)
            sw      $20,48+16($sp)
            sw      $21,48+20($sp)
            sw      $22,48+24($sp)
            sw      $23,48+28($sp)
            sw      $30,48+32($sp)
            sw      $31,48+36($sp)
            sw      $sp,0($4)
            lw      $sp,0($5)
            l.d     $f20,0($sp)
            l.d     $f22,8($sp)
            l.d     $f24,16($sp)
            l.d     $f26,24($sp)
            l.d     $f28,32($sp)
            l.d     $f30,40($sp)
            lw      $16,48+0($sp)
            lw      $17,48+4($sp)
            lw      $18,48+8($sp)
            lw      $19,48+12($sp)
            lw      $20,48+16($sp)
            lw      $21,48+20($sp)
460         lw      $22,48+24($sp)
            lw      $23,48+28($sp)
            lw      $30,48+32($sp)
            lw      $31,48+36($sp)
            addu    $sp,88
            j       $31
```

这里是MIPS的启动代码：

```
⟨MIPS startup 461⟩≡
  .globl  _start
  _start: move    $4,$23      # register 23 holds args
          move    $25,$30     # register 30 holds apply
          jal     $25
          move    $4,$2       # Thread_exit(apply(p))
          move    $25,$21     # register 21 holds Thread_exit
          jal     $25
          syscall
  .end    _swtch
  .globl  _ENDMONITOR
  _ENDMONITOR:
```

这段代码与Thread_new中受MIPS影响的部分代码共同合作，并通过将Thread_exit、args及apply存储在帧中的适当位置而安排它们分别显示在寄存器21、23和30中。apply的第一个参数传递给寄存器4，并将结果返回到寄存器2。启动代码不需要一个帧，因此Thread_new仅建立一个_swtch帧，但是它在堆栈中那个帧下面分配四个字，以防apply接收不定量的参数。

```
⟨initialize a MIPS stack 461⟩≡
  extern void _start(void);
  t->sp -= 16/4;
  t->sp -= 88/4;
  t->sp[(48+20)/4] = (unsigned long)Thread_exit;
  t->sp[(48+28)/4] = (unsigned long)args;
  t->sp[(48+32)/4] = (unsigned long)apply;
  t->sp[(48+36)/4] = (unsigned long)_start;
```

由于MIPS的启动代码必须独立于位置，Thread_exit的地址就在寄存器21中传递。启动代码在调用（jal指令）之前将args的地址复制到寄存器4，并将apply和Thread_exit的地址复制到寄存器25，因为那是MIPS独立于位置的调用顺序所要求的。 461

ALPHA代码块与相应的MIPS代码块类似。

```
⟨ALPHA swtch 462⟩≡
  .globl  _swtch
  .ent    _swtch
  _swtch: lda     $sp,-112($sp)    # allocate _swtch's frame
          .frame  $sp,112,$26
          .fmask  0x3f0000,-112
          stt     $f21,0($sp)      # save from's registers
          stt     $f20,8($sp)
          stt     $f19,16($sp)
          stt     $f18,24($sp)
          stt     $f17,32($sp)
          stt     $f16,40($sp)
          .mask   0x400fe00,-64
          stq     $26,48+0($sp)
          stq     $15,48+8($sp)
          stq     $14,48+16($sp)
```

```
        stq     $13,48+24($sp)
        stq     $12,48+32($sp)
        stq     $11,48+40($sp)
        stq     $10,48+48($sp)
        stq     $9,48+56($sp)
        .prologue 0
        stq     $sp,0($16)      # save from's stack pointer
        ldq     $sp,0($17)      # restore to's stack pointer
        ldt     $f21,0($sp)     # restore to's registers
        ldt     $f20,8($sp)
        ldt     $f19,16($sp)
        ldt     $f18,24($sp)
        ldt     $f17,32($sp)
        ldt     $f16,40($sp)
        ldq     $26,48+0($sp)
        ldq     $15,48+8($sp)
        ldq     $14,48+16($sp)
        ldq     $13,48+24($sp)
462     ldq     $12,48+32($sp)
        ldq     $11,48+40($sp)
        ldq     $10,48+48($sp)
        ldq     $9,48+56($sp)
        lda     $sp,112($sp)    # deallocate frame
        ret     $31,($26)
.end    _swtch
```

⟨*ALPHA startup* 463⟩≡

```
.globl  _start
.ent    _start
_start: .frame  $sp,0,$26
        .mask   0x0,0
        .prologue 0
        mov     $14,$16         # register 14 holds args
        mov     $15,$27         # register 15 holds apply
        jsr     $26,($27)       # call apply
        ldgp    $26,0($26)      # reload the global pointer
        mov     $0,$16          # Thread_exit(apply(args))
        mov     $13,$27         # register 13 has Thread_exit
        jsr     $26,($27)
        call_pal0
.end    _start
.globl  _ENDMONITOR
_ENDMONITOR:
```

⟨*initialize an ALPHA stack* 463⟩≡

```
extern void _start(void);
t->sp -= 112/8;
t->sp[(48+24)/8] = (unsigned long)Thread_exit;
t->sp[(48+16)/8] = (unsigned long)args;
t->sp[(48+ 8)/8] = (unsigned long)apply;
t->sp[(48+ 0)/8] = (unsigned long)_start;
```

参考书目浅析

Andrews（1991）是关于并行编程的综合文章。它介绍了大部分针对于编写并行系统的问题及解决方案，包括同步机制、消息传递系统和远程过程调用。它也介绍了四种编程语言专为并行编程所设计的特点。

463

Thread是建立在Modula-3之上的线程接口，这个接口来自于使用Digital系统研究中心的Modula-2+线程工具的经历。Andrew Birrell 所著的Nelson（1991）第4章介绍了如何用线程编程；任何人编写基于线程的应用程序都能从这篇文章中获益。大多数现代操作系统中的线程工具都在某种方式上基于SRC接口。

Tanenbaum（1995）调查了用户级和核心级线程的设计问题，并概括了它们的实现方案。他的个案研究介绍了三种操作系统（Amoeba、Chorus和Mach）中的线程包，以及开放软件基金会的分布式计算环境中的线程。最初，DCE就像这种环境为人们了解那样，是运行在UNIX的OSF/1变体上，但是现在大多数操作系统上都拥有这种环境，包括OpenVMS、OS/2、Windows NT及Windows 95。

Kleiman、Shah和Smaalders（1996）详细介绍了POSIX线程（电气和电子工程协会1995）和Solaris 2线程。这本实用性的书有一章是关于线程交互作用和库的，以及使用线程并行化算法的大量范例，包括排序和列表、队列和散列表的线程安全实现。

sieve改编自一个类似的例子，McIlroy（1968）曾用来介绍如何使用与无优先权线程一样的协同程序进行编程。几种语言中都出现了协同程序、有时名称不同而已。Icon的协同表达就是个例子（Wampler和Griswold 1983）。Marlin（1980）调查了许多原始的协同程序提案，并介绍了Pascal变体的模型实现。

通道基于CSP——通信顺序进程（Hoare 1978）。线程和通道都出现在Newsqueak中，这是一个应用的并行语言。CSP和Newsqueak中的通道比Chan所提供的更强大，因为这两种语言都有工具可以在多个通道不确定地等待。Pike（1990）浏览了Newsqueak的一种解释程序实现的最重要部分，并介绍了使用随机数改变抢占频率，使线程调度变得不确定（但合理）。McIlroy（1990）详细叙述了一个将幂级数作为数据流处理的Newsqueak程序；他的方法非常类似于筛法的精髓。

Newsqueak现已用于实现窗口系统，它举例说明了从线程中获得最大益处的各种交互应用。NeWS窗口系统（Gosling、Rosenthal和Arden 1989）是用具有线程的语言所编写的另一个窗口系统范例。NeWS系统的核心是个处理文本和图象的PostScript解释程序。大多数NeWS窗口系统本身就是用它包括无优先权线程扩展部分的PostScript变体编写的。

464

函数式语言Concurrent ML（Reppy 1997）支持线程和同步通道的方式更像Chan一些。通常用非命令性语言比基于堆栈的命令性语言更容易实现线程。例如，在Standard ML中就没有堆栈，因为激活过程会它们的调用方存在的时间更长。因此，Concurrent ML全部采用Standard ML实现。

使用_MONITOR和_ENDMONITOR函数定义Thread和Sem实现中的代码来自于Cormack（1988），其中介绍了UNIX线程的一个类似但稍有不同的接口。Stevens（1992）的第10章全面论述了信号和信号处理过程；它还介绍了UNIX变体和POSIX标准之间的区别。

练习

20.1 二进制信号量——通常称为锁或互斥体——是最普遍的一种信号量。为锁设计一个单独的接口，它的实现要比一般信号量的接口简单。要注意警告。

20.2 假设线程*A*锁定*x*，然后试图锁住*y*，同时*B*锁定*y*，然后试图锁住*x*。这些线程就死锁了：*A*不能继续执行除非*B*解开*y*；而*B*也不能运行，除非*A*解开*x*。扩展你在上题中锁的实现以检测这些各种各样的简单死锁。

20.3 不使用信号量重新实现thread.c中的Chan接口。为通道设计一个何时的表示法，并直接使用这种内部队列和线程函数，而非信号量函数。要注意警告。设计一套测试方法估计一下这些推测起来可能效率更高一些的实现的好处。量化对这个改进的实现应用程序必须具有的消息行为的级别。

465

20.4 为异步带缓冲通信设计并实现一个接口——一个线程间消息工具，其发送方不用等待消息被接收，同时消息一直缓冲到被接收为止。你的设计应当允许消息存在的时间长于它们的发送线程所存在的时间；也就是说，线程发送出消息之后在消息被接收之前退出。异步通信比Chan的同步通信更加复杂，因为它必须处理缓冲消息的存储管理以及更多错误条件，例如，为线程提供一种方式确定是否已经接收了一条消息。

20.5 Modula-3支持条件变量。条件变量c与锁m相关。原子操作sleep(m, c)使调用线程解开m的锁并等待c。调用线程必须锁定m。wakeup(c)使一个或多个线程等待c重新开始运行；其中之一解开m并从它的调用返回到sleep。broadcast(c)与wakeup(c)一样，但是c上所有睡眠中的线程重新开始运行。警告对阻塞在条件变量上的线程不起作用，除非它们调用了alertsleep而不是sleep。当一个调用了alertsleep的线程得到警告的时候，它就锁定m并引发Thread_Alerted。设计并实现一个支持条件变量的接口；使用你在练习20.1中的锁。

20.6 如果系统支持非阻塞I/O系统调用，就用它们建立一个C标准I/O库的线程安全实现。也就是说，如果一个线程调用fgetc，其他线程可以在那个线程等待输入的时候运行。

20.7 设计一种方法不使用_MONITOR和_ENDMONITOR让Thread和Sem函数具有原子性。提示：单个全局临界标志是不够的。每个线程都会需要一个临界标志，而且汇编语言代码要修改这个标志。要注意——使用这种方法你会无法相信那么容易就犯下了一个细小的错误。

466

20.8 扩展Thread_new，让它接受指定堆栈大小的可选参数。例如，

```
t = Thread_new(..., "stacksize", 4096, NULL);
```

就会创建一个带有4K字节堆栈的线程。

20.9 给20.1节中介绍的Thread实现增加对少量优先权的支持。修改Thread_init和Thread_new，让它们可以接受优先权说明作为可选参数，Tanenbaum（1995）介绍了如何实现一个支持优先权的合理调度策略。

20.10 DCE支持模板，本质上是线程属性的结合表格。当用DCE的pthread_create创建

线程的时候，模板提供诸如堆栈大小及优先权这样的属性。模板不会在线程创建的调用中重复相同的参数，同时不运行在创建的地点指定线程属性。用Table_T为Thread设计一个模板工具，并修改Thread_new让它接受模板作为其可选参数之一。

20.11 在Sequent这样具有共享内存的多处理器上实现Thread和Sem。这种实现比20.3节中详细叙述的实现要更加复杂，因为在多处理器上线程是真正地并行执行。实现原子操作将需要某种形式的低级自旋锁以确保对访问共享数据结构的小临界区的独占访问，就像Thread和Sem函数中的那样。

20.12 在具有许多处理器的大规模并行处理器（MPP）实现Thread、Sem和Chan，例如Cray T3D，它由2^n个DEC ALPHA 处理器构成。在MPP上每个处理器都有它自己的内存，而且一个处理器还要某种低级机制（通常在硬件中实现）以访问其他处理器的内存。这个练习中的难点之一是要决定如何把Thread、Sem及Chan接口喜欢的共享内存模型映射到MPP提供的分布内存模型。

467

20.13 用DCE线程实现Thread、Sem和Chan。一定要指明你的Thread_new实现接受的是哪些依赖于系统的可选参数。

20.14 用Solaris 2上的LWP实现Thread、Sem和Chan，根据需要给Thread_new提供可选参数。

20.15 用POSIX线程实现Thread、Sem和Chan（参见Kleiman、Shah及Smaalders 1996）。

20.16 用Microsoft的Win32线程接口实现Thread、Sem和Chan（参见Richter 1995）。

20.17 如果你有权使用SPARC的C编译器，例如lcc（Fraser和Hanson 1995），请修改这个编译器，让它不要使用SPARC寄存器窗口，它在_swtch中不能执行ta 3系统调用。你还必须重新编译所使用的任意库。估计一下运行期中的改进结果。警告：这个练习是项大工程。

20.18 Thread_new必须分配一个堆栈，因为大多数编译系统都假定程序开始运行时已经分配一个连续的堆栈。少数系统，例如Cray-2，是在运行中按块分配堆栈。如果当前块能够容纳，函数入口序列就在当前块中分配帧；否则，它就分配一个足够大小的新块，并将这个新块链接到当前块。当已有序列的最后一个帧被删除时，它就断开块的链接并释放其存储空间。这种方法不仅简化了线程的创建，也自动检查了堆栈溢出。修改一个C编译器，让它使用这种方法，并估计一下修改后的好处。和前一练习一样，需要重新编译使用的每个库；而且这个练习也是一项大工程。

468

附录 接口概要

下面按字母表顺序列出了接口概要；小节中列举了每个接口，而且如果接口具有基本类型也将其列出。表示法“T是隐式X_T”表示接口X导出了一个隐型指针类型X_T，在描述中缩写为T。如果接口出现了基本类型，就给定了X_T的表示形式。

每个接口的概要都是以字母表顺序列出导出的变量以及函数，其中不包括异常。每个函数的原型后面都有它会引发的异常及一个简明的描述。缩写c.r.e和u.r.e代表可检查或不可检查的运行期错误。

下表分类概括了接口，并给出了其相应概要开始的页面。

AP T是隐式AP_T

给任意AP函数传递空T时产生一个c.r.e.，

T AP_add(T x, T y) Mem_Failed

T AP_addi(T x, long int y) Mem_Failed

返回x+y的和。

int AP_cmp(T x, T y)

int AP_cmpi(T x, long int y)

如果x<y、x=y或x>y则分别返回小于0、等于0或大于0的整数。

T AP_div(T x,T y) Mem_Failed

T AP_divi(T x, long int y) Mem_Failed

返回商x/y；参见Arith_div。y=0是个c.r.e.。

void AP_fmt(int code, va_list *app, int put(int c, void *cl), void *cl,

unsigned char flags[], int width, int precision) Mem_Failed

一个Fmt转换函数：需要一个T，并像printf的%d那样格式化T。app或flags为空时产

生c.r.e.。

void AP_free(T *z)

释放并清除*z。z或*z为空是c.r.e.。

T AP_fromstr(const char *str, int base, char **end) Mem_Failed

把str解释为一个以base为底数的整数，并返回结果T。忽略前导空格并接受一个跟有一个或多个以base为底数的数字的可选符号。对于10<base≤36，小写字母或大写字母解释为大于9的数字。如果end≠null，*end就指向str中终止扫描的那个字符。如果str没有指定base的底数，AP_fromstr就返回空，而且如果end非空就将*end设为str。如果str=null或者base<2或base>36则产生c.r.e.。

T AP_lshift(T x, int s) Mem_Failed

返回x左移s位后的结果：空出的位填零，且结果的符号与x相同。s<0时产生c.r.e.。

T AP_mod(T x, T y) Mem_Failed

long AP_modi(T x, long int y) Mem_Failed

返回x mod y；参见Arith_mod。y = 0时产生c.r.e.。

T AP_mul(T x, T y) Mem_Failed

T AP_muli(T x, long int y) Mem_Failed

470 返回乘积x · y。

T AP_neg(T x) Mem_Failed

返回-x。

T AP_new(long int n) Mem_Failed

分配并返回初始为n的一个新T。

T AP_pow(T x, T y, T p) Mem_Failed

返回x^y mod p。如果p为空则返回x^y。y<0或p非空但P<2时产生c.r.e.。

T AP_rshift(T x, int s) Mem_Failed

返回x右移s位后的结果：空出的位填零，且结果的符号与x相同。s<0时产生c.r.e.。

T AP_sub(T x, T y) Mem_Failed

T AP_subi(T x, long int y) Mem_Failed

返回x - y的差。

long int AP_toint(T x)

返回符号与x相同且值为|x| mod LONG_MAX+1的长整数。

char * AP_tostr(char *str, int size, int base, T x) Mem_Failed

用x的base底数字符表示形式填充str[0..size-1]，并返回str。如果str为空，AP_tostr分配一个str。大写字母用于在base>10时表示大于9的数字。非空的str太小或者base<2或base>36时产生一个c.r.e.。

Arena T是隐式Arena_T

给任意Arena函数传递nbytes≤0或空T时产生一个c.r.e.。

void *Arena_alloc(T arena, long nbytes, const char *file, int line)　Arena_Failed

在arena中分配nbytes个字节，并返回指向第一个字节的指针。字节内容没有初始化。如果Arena_alloc引发Arena_Failed，那么file和line就作为错误源进行报告。

void *Arena_calloc(T arena, long count, long nbytes, const char *file, int line) Arena_Failed

在arena中为count元素数组分配空间，每个元素占nbytes，并返回第一个元素的指针。count≤0时产生一个c.r.e.。元素均未初始化。如果时产生一个c.r.e.。如果Arena_calloc引发Arena_Failed，那么file和line就作为错误源进行报告。 471

void Arena_dispose(T *ap)

释放*ap中的所有空间，释放arena自身，并清空*ap。ap或*ap为空时产生一个c.r.e.。

void Arena_free(T arena)

释放arena中的所有空间——从最后一次调用Arena_free以来分配的所有空间。

T Arena_new(void)　Arena_NewFailed

分配、初始化并返回一个新的arena。

Arith

int Arith_ceiling(int x, int y)

返回不小于x/y实数商值的最小整数。y = 0时产生一个u.r.e。

int Arith_div(int x, int y)

返回x/y，不超过实数z的最大整数，让z·y=x。趋向−∞进行截取；例如，Arith_div(−13, 5)返回−3。y=0时产生一个u.r.e。

int Arith_floor(int x, int y)

返回不超过x/y实数商值的最大整数。y=0时产生一个u.r.e。

int Arith_max(int x, int y)

返回max(x, y)。

int Arith_min(int x, int y)

返回min(x. y)。

int Arith_mod(int x, int y)

返回x-y·Arith_div(x, y)；例如，Arith_mod(-13, 5)返回2。y=0时产生一个u.r.e。

Array　T是隐式Array_T

数组下标从0到*N*−1，这里*N*是数组的长度。空数组没有元素。给任意Array函数传递空T时产生一个c.r.e.。 472

T Array_copy(T array, int length)　Mem_Failed

创建并返回新数组存放array中的前length个元素。如果length大于array的长度，多余的元素就被清空了。

Void Array_free(T *array)

释放并清除*array。array或*array为空时产生一个c.r.e.。

void *Array_get(T array, int i)

返回指向array中第i个元素的指针。i<0或i≥N时产生一个c.r.e，这里N为array长度。

int Array_length(T array)

返回array中的元素个数。

T Array_new(int length, int size) Mem_Failed

分配、初始化及返回具有length个元素的新数组，每个元素为size字节。元素都清空，length<0或size≥0时产生一个c.r.e.。

void * Array_put(T array, int i, void *elem)

从elem复制Array_size(array)个字节到array中的第i个元素，并返回elem。elem为空或者i<0或i≥N时产生一个c.r.e，这里N为array长度。

void Array_resize(T array, int length) Mem_Failed

把array的元素个数改为length。如果length大于原始长度，多余的元素就被清空了。length<0时产生一个c.r.e.。

int Array_size(T array)

返回array中元素的字节数。

ArrayRep T是Array_T

```
typedef struct T {
  int length; int size; char *array; }*T;
```

修改T中的字段是一个u.r.e。

void ArrayRep_init(T array, int length, int size, void *ary)

用length、size和ary的值初始化array中的字段。length≠0且ary为空、length = 0且ary非空或者size ≤0时产生一个c.r.e.。用其他方法初始化T时会产生一个u.r.e。

473

Assert

assert(e)

如果e为零则引发Assert_Failed。依照句法，assert(e)是个表达式。如果包含assert.h时定义了NDEBUG，就禁用了断言。

Atom

给任意Atom函数传递空str时产生一个c.r.e.。修改一个原子会引发u.r.e。

int Atom_length(const char *str)

返回原子str的长度。str不是一个原子时产生一个c.r.e.。

const char * Atom_new(const char *str, int len) Mem_Failed

为str[0..len-1]返回一个原子，如果需要新建一个。len<0时产生一个c.r.e.。

const char * Atom_string(const char *str) Mem_Failed

返回Atom_new(str, strlen(str))。

const char * Atom_int(long n)　Mem_Failed

为n的十进制字符串表示形式返回一个原子。

Bit　T是隐式Bit_T

位矢量中的位是有限的0到$N-1$，这里N是矢量的长度。给任意Bit函数传递空T时产生一个c.r.e，除了Bit_union、Bit_inter、Bit_minus和Bit_diff。

void Bit_clear(T set, int lo, int hi)

清除set中的位lo..hi。lo>hi、lo<0或lo≥N时产生一个c.r.e.。这里N为set的长度，hi也类似。

int Bit_count(T set)

返回set中1的个数。

474

T Bit_diff(T s, T t)　Mem_Failed

返回s/t的对称差分：s和t的异或。如果s为空或t为空，它就表示空集。s为空且t为空或者s和t长度不同时产生一个c.r.e.。

int Bit_eq(T s, T t)

如果s=t则返回1否则返回0。s和t长度不同时产生一个c.r.e.。

void Bit_free(T *set)

释放并清空*set。set或*set为空时产生一个c.r.e.。

int Bit_get(T set , int n)

返回第n位。n<0或n≥N时产生一个c.r.e，这里N为set的长度。

T Bit_inter(T s, T t)　Mem_Failed

返回s∩t：s和t的逻辑与。其c.r.e参见Bit_diff。

int Bit_length(T set)

返回set的长度。

int Bit_leq(T s, T t)

如果s⊆t则返回1；否则返回0。其c.r.e参见Bit_eq。

int Bit_lt(T s, T t)

如果s⊂t则返回1；否则返回0。其c.r.e参见Bit_eq。

void Bit_map(T set, void apply(int n, int bit, void *cl), void *cl)

set中从第0位到第$N-1$位的每一位都调用apply(n, bit, cl)，这里N为set的长度。通过apply修改set以作用于bit后来的值。

T Bit_minus(T s, T t)　Mem_Failed

返回s-t：s和~t的逻辑与。其c.r.e参见Bit_diff。

T Bit_new(int length)　Mem_Failed

创建并返回length长度的新位矢量0。length<0时产生一个c.r.e.。

void Bit_not(T set, int lo, int hi)

将set中的位lo..hi取反。其c.r.e参见Bit_clear。

int Bit_put(T set, int n, int bit)

将第n位设置为bit，并返回位n以前的值。bit<0或bit>1时或n<0或n≥*N*时产生一个c.r.e.。这里*N*为set的长度。

void Bit_set(T set, int lo, int hi)

设置set中的位lo..hi。其c.r.e参见Bit_clear。

T Bit_union(T s, T t) Mem_Failed

475 返回s∪t：s和t的逻辑或。其c.r.e参见Bit_diff。

Chan T是隐式Chan_T

给任意Chan函数传递空T或调用Thread_init之前调用会产生一个c.r.e.。

T Chan_new(void) Mem Failed

创建、初始化及返回新通道。

int Chan_receive(T c, void *ptr, int size)

等待相应的Chan send，然后从发送方拷贝size之多的字节到ptr，并返回拷贝的字节数。ptr为空或size<0时产生一个c.r.e.。

int Chan_send(T c, const void *ptr, int size) Thread_Alerted

等待相应的Chan_receive，然后从ptr拷贝size之多的字节到接收方，并返回拷贝的字节数。其c.r.e参见Chan_receive。

Except T是Except_T

typedef struct T {char *reason; } T;

TRY语句的语法如下所示：S和e表示语句和异常。ELSE语句可选。

TRY S EXCEPT(e_1) S_1...EXCEPT(e_n) S_n ELSE S_0 END_TRY

TRY S FINALLY S_1 END_TRY

void Except_raise(const T *e, const char *file, int line)

在源坐标file和line引发异常*e。e为空时产生一个c.r.e.。未捕捉的异常将使程序中断。

RAISE(e)

引发e。

RERAISE

再次引发使管理器运行的异常。

RETURN

RETURN 表达式

476 是个用在TRY语句中的返回语句。在TRY语句中使用C返回语句会产生一个u.r.e.。

Fmt T是Fmt_T

typedef void (*T)(int code, va_list *app, int put(int c, void *cl), void *cl,
unsigned char flags[256], int width, int precision)

定义了转换函数的类型，相关转换指示符出现格式串中时由Fmt函数调用。这里，调用put(c, cl)生成每个格式化字符c。表14-1(162页)概括了转换指示符的初始集。向任意Fmt函数传递空put、buf或fmt，或者格式串使用了与转换函数无关的转换指示符。

char *Fmt_flags = "-+0"

指向可以出现在转换指示符中的标志符。

void Fmt_fmt(int put(int c, void *cl), void *cl, const char *fmt, ...)

根据格式串fmt格式化并生成"..."参数。

void Fmt_fprint(FILE *stream, const char *fmt, ...)

void Fmt_print(const char *fmt, ...)

根据fmt格式化并生成"..."参数；Fmt_fprint写到stream，而Fmt_print写到stdout。

void Fmt_putd(const char *str, int len, int put(int c, void *cl), void *cl,
unsigned char flags[256], int width, int precision)

void Fmt_puts(const char *str, int len, int put(int c, void *cl), void *cl,
unsigned char flags[256], int width, int precision)

根据Fmt的默认值（参见162页的表14-1）和flags、width和precision的值格式化并在str[0..len-1]生成转换过来的数字（Fmt_putd）和字符串（Fmt_puts） str为空、len<0或flags为空时产生一个c.r.e.。

T Fmt_register(int code, T cvt)

把cvt与格式符code关联，并返回前一转换函数。code<0或code>255时产生一个c.r.e.。 477

int Fmt_sfmt(char *buf, int size, const char *fmt, ...) Fmt_Overflow

根据fmt格式化"..."参数并放入buf[1..size-1]；在末尾附加一个空字符；以及返回buf的长度。size≤0时产生一个c.r.e. 如果生成了多余size-1个字符则引发Fmt_Overflow。

char *Fmt_string(const char *fmt, ...)

根据fmt将"..."参数格式化成以null结束的字符串，并返回这个字符串。

void Fmt_vfmt(int put(int c, void *cl), void *cl, const char *fmt, va_list ap)

参见Fmt_fmt；从ap列表中接收参数。

int Fmt_vsfmt(char *buf, int size, const char *fmt, va_list ap) Fmt_Overflow

参见Fmt_sfmt；从ap列表中接收参数。

char *Fmt_vstring(const char *fmt, va_list ap)

参见Fmt_string；从列表ap中接收参数。

List T是List_T

typedef struct T *T;

struct T { T rest; void *first; };

所有List函数对任意list参数接受空T并解释为空列表。

T List_append(T list, T tail)

给list末尾添加tail并返回list。如果list为空，List_append就返回tail。

T List_copy(T list) Mem_Failed

创建并返回最高级别的副本。

void List_free(T *list)

释放并清空*list。list为空时产生一个c.r.e.。

int List_length(T list)

返回list中的元素个数。

T List_list(void *x, ...) Mem_Failed

478 创建并返回一个列表，其元素为“...”参数，一直到第一个空指针。

void List_map(T list, void apply(void **x, void *cl), void *cl)

对list中的每个元素p调用apply(&p->first, cl)。如果apply修改list则为一个u.r.e。

T List_pop(T list, void **x)

如果x非空，把list->first赋给*x；释放list并返回list->rest。如果list为空，List_pop返回空且不修改*x。

T List_push(T list, void *x) Mem_Failed

在list的前面添加一个新元素存放x，并返回新列表。

T List_reverse(T list)

将list中的元素颠倒顺序排列，然后返回颠倒过来的列表。

void ** List_toArray(T list, void *end) Mem_Failed

创建一个N+1个元素的数组，其中存放了list中的N个元素，并返回指向这个数组第一个元素的一个指针。数组中的第N个元素是end。

Mem

向任意Mem函数或宏传递nbytes≤0。

ALLOC(nbytes) Mem_Failed

分配nbytes字节并返回指向第一个字节的指针。字节未初始化。参见Mem_alloc。

CALLOC(count, nbytes) Mem_Failed

为一个count元素的数组分配空间，每个元素占nbytes字节，并返回指向第一个元素的指针。count≤0时产生一个c.r.e.。其中元素都被清空。参见Mem_calloc。

FREE(ptr)

如果ptr非空就释放ptr，并且清除ptr。多次计算ptr。参见Mem_free。

void *Mem_alloc(long nbytes), const char *file, int line) Mem_Failed

分配nbytes字节并返回指向第一个字节的指针。字节未初始化。如果Mem_alloc引发
479 Mem_Failed，file和line就作为错误源进行报告。

void *Mem_calloc(long count, long nbytes, const char *file, int line) Mem_Failed

为一个count元素的数组分配空间，每个元素占nbytes字节，并返回指向第一个元素的指针。count≤0时产生一个c.r.e.。其中元素都被清空，不需要将指向空或浮点数值的指针初始化为0.0。如果Mem_calloc引发Mem_Failed，file和line就作为错误源进行报告。

void Mem_free(void *ptr, const char *file, int line)

如果ptr非空则释放ptr。如果ptr是Mem分配函数的前一调用所未返回的指针，则引发一个u.r.e。实现使用file和line报告内存使用错误。

void *Mem_resize(void *ptr, long nbytes, const char *file, int line) Mem_Failed

修改存放nbytes字节的ptr存储块的大小，并返回指向这个新块第一个字节的指针。如果nbytes超出原始块的大小，多余的字节就不初始化。如果nbytes小于原始块的大小，那么仅有nbytes字节出现在新块中。如果Mem_resize引发Mem_Failed，file和line就作为错误源进行报告。ptr为空时产生一个c.r.e；如果ptr是Mem分配函数的前一调用所未返回的指针，则引发一个u.r.e。

NEW(p) Mem_Failed

NEWO(p) Mem_Failed

分配足够大的内存块存放*p，将p设置为块地址，并返回该地址。NEWO清除这些字节内容，NEW则不进行初始化。这两个宏都仅计算一次ptr。

RESIZE(ptr, nbytes) Mem_Failed

修改存放nbytes字节的ptr存储块的大小，并让ptr重新指向大小调整后的存储块，然后返回块地址。ptr要多次计算。参见Mem_resize。

MP T是MP_T

typedef unsigned char *T

MP函数执行n位有符号和无符号算法，这里n最初为32，而且可以通过MP_set修改。函数指定以u或ui结尾的执行无符号算法；其他的执行有符号算法。MP函数在引发MP_DivideByZero或MP_Overflow前计算它们的结果。向任意MP函数传递空T会产生一个c.r.e.。向任意MP函数传递太小的T会产生一个u.r.e。 480

T MP_add(T z, T x, T y) MP_Overflow

T MP_addi(T z, T x, long y) MP_Overflow

T MP_addu(T z, T x, T y) MP_Overflow

T MP_addui(T z, T x, unsigned long y) MP_Overflow

z设为x+y，并返回z。

T MP_and(T z, T x, T y)

T MP_andi(T z, T x, unsigned long y)

z设为x AND y，并返回z。

T MP_ashift(T z, T x, int s)

z设为x右移s位后的值并返回z。空出的位填上x的符号位。s<0时产生一个c.r.e.。

int MP_cmp(T x, T y)

int MP_cmpi(T x, long y)

int MP_cmpu(T x, T y)

int MP_cmpui(T x, unsigned long y)

x<y、x=y或x>y时分别返回小于0、等于0或大于0的整数。

T MP_cvt(int m, T z, T x) MP_Overflow

T MP_cvtu(int m, T z, T x) MP_Overflow

把x缩小或扩大为一个m位的有符号或无符号的整数并放入z，然后返回z。m<2时产生一个c.r.e.。

T MP_div(T z, T x, T y) MP_Overflow, MP_DivideByZero

T MP_divi(T z, T x, long y) MP_Overflow, MP_DivideByZero

T MP_divu(T z, T x, T y) MP_DivideByZero

T MP_divui(T z, T x, unsigned long y) MP_Overflow, MP_DivideByZero

z设为x/y并返回z。有符号函数趋向于$-\infty$截取；参见Arith_div。

void MP_fmt(int code, va_list * app,int put(int c, void *cl), void *cl,
unsigned char flags[], int width, int precision)

void MP_fmtu(int code, va_list * app,int put(int c, void *cl), void *cl,
unsigned char flags[], int width, int precision)

是Fmt转换函数。它们需要一个T及一个底数b，并像printf的%d和%u那样进行格式化。
481 b<2或b>36时和app或flags为空时产生一个c.r.e.。

T MP_fromint(T z, long v) MP_Overflow

T MP_fromintu(T z, unsigned long u) MP_Overflow

z设为v或u并返回z。

T MP_fromstr(T z, const char *str, int base,char **end) MP_Overflow

把str解释为以base为底的一个整数，并把z设为那个整数，然后返回z。参见AP_fromstr。

T MP_lshift(T z, T x, int s)

z设为x左移s位后的值然后返回z。空出的位填上零。s<0时产生一个c.r.e.。

T MP_mod(T z, T x, T y) MP_Overflow, MP_DivideByZero

z设为 x mod y并返回z。趋向于$-\infty$进行截取；参见Arith_mod。

long MP_modi(T x, long y) MP_Overflow, MP_DivideByZero

返回x mod y。趋向于$-\infty$进行截取；参见Arith_mod。

T MP_modu(T z, T x, T y) MP_DivideByZero

z设为 x mod y并返回z。

unsigned long MP_modui(T x, unsigned long y) MP_Overflow, MP_DivideByZero

返回x mod y。

T MP_mul(T z, T x, T y) MP_Overflow

z设为x·y并返回z。

T MP_mul2(T z, T x, T y) MP_Overflow

T MP_mul2u(T z, T x, T y) MP_Overflow

z设为x·y的双倍长度结果并返回具有2n位的z。

T MP_muli(T z, T x, long y) MP_Overflow

T MP_mulu(T z, T x, T y) MP_Overflow

T MP_mului(T z, T x, unsigned long y) MP_Overflow

z设为x·y并返回z。

T MP_neg(T z, T x) MP_Overflow

z设为–x并返回z。

T MP_new(unsigned long u) Mem_Failed, MP_Overflow

创建并返回初始为u的一个T。

T MP_not(T z, T x)

z设为~x并返回z。

T MP_or(T z, T x, T y)

T MP_ori(T z, T x, unsigned long y)

z设为x OR y并返回z。

T MP_rshift(T z, T x, int s)

z设为x右移s位后的值并返回z。空出的位填零。s<0时产生一个c.r.e.。 482

int MP_set(int n) Mem_Failed

重新将MP设置为n位算法。n<2时产生一个c.r.e.。

T MP_sub(T z, T x, T y) MP_Overflow

T MP_subi(T z, T x, long y) MP_Overflow

T MP_subu(T z, T x, T y) MP_Overflow

T MP_subui(T z, T x, unsigned long y) MP_Overflow

z设为x-y并返回z。

long int MP_toint(T x) MP_Overflow

unsigned long MP_tointu(T x) MP_Overflow

x返回位一个长整数或无符号长整数。

char * MP_tostr(char *str, int size, int base, T x) Mem_Failed

str[0..size–1]填以null结尾的x的base底的字符串表示形式，然后返回str。如果str为空，MP_tostr则忽略size并分配这个字符串。参见AP_tostr。

T MP_xor (T z, T x, T y)

T MP_ xori(T z, T x, unsigned long y)

z设为x XOR y并返回z。

Ring T是隐式的Ring_T

环的下标从0到$N-1$，这里N是环的长度。空环没有元素。任何地方都可以添加或删除指针；环字段扩充。旋转一个环改变它的原点。向任意Ring函数传递一个空T会产生一个c.r.e.。

void *Ring_add(T ring, int pos, void *x) Mem_Failed

在ring中的pos位置插入x并返回x。位置指的是元素之间的点；参见Str。pos < $-N$或

pos>*N*+1时产生一个c.r.e，这里*N*为ring的长度。

void *Ring_addhi(T ring, void *x) Mem_Failed

void *Ring_addlo(T ring, void *x) Mem_Failed

向ring的高端（下标*N*−1）或低端（下标0）添加x，并返回x。

void Ring_free(T *ring)

释放并清除*ring。ring或*ring为空时产生一个c.r.e.。

int Ring_length(T ring)

483 返回ring中的元素个数。

void * Ring_get(T ring, int i)

返回环中的第i个元素。i<0或i≥*N*时产生一个c.r.e，这里*N*是ring的长度。

T Ring_new(void) Mem_Failed

创建并返回一个空环。

void *Ring_put(T ring, int i, void *x) Mem_Failed

把ring中的第i个元素改为x并返回以前的值。其c.r.e参见Ring_get。

void *Ring_remhi(T ring)

void *Ring_remlo(T ring)

删除并返回ring高端（下标*N*−1）或低端（下标0）的元素。ring为空时产生一个c.r.e.。

void* Ring_remove(T ring, int i)

从ring删除并返回元素i。i<0或i≥*N*时产生一个c.r.e，这里*N*是ring的长度。

T Ring_ring(void *x, ...) Mem_Failed

创建并返回一个环，其元素为“...”参数直到第一个空指针。

void Ring_rotate(T ring, int n)

把ring的原点向左（n<0）或向右（n≥0）旋转n个元素。|n|<0或|n|>*N*时产生一个c.r.e，这里*N*是ring的长度。

Sem T是隐式的Sem_T

typedef struct T { int count; void *queue; } T;

直接读写T中的字段，或向任意Sem函数传递未初始化的T是个u.r.e。向任意Sem函数传递空T或在调用Thread_init之前调用任意Sem函数都会引发一个c.r.e.。

LOCK语句的句法如下；S和m表示语句及T。

LOCK(m) S END_LOCK

484 m被锁定，执行语句S并解开m。LOCK可以引发Thread_Alerted。

void Sem_init(T *s, int count)

把s->count设为count。在同一T上多次调用Sem_init是个u.r.e。

Sem_T *Sem_new(int count) Mem_Failed

创建并返回一个count字段初始化为count的T。

void Sem_wait(T *s) Thread_Alerted

等待直到s->count>0，然后递减s->count。

void Sem_signal(T *s) Thread_Alerted

递增s->count。

Seq T是隐式的Seq_T

序列下标从0到$N-1$，这里N是序列的长度。空序列内有元素。低端（下标0）或高端（$N-1$）可以添加或删除指针；序列自动扩充。向任意Seq函数传递空T是个c.r.e.。

void *Seq_addhi(T seq, void *x) Mem_Failed

void * Seq_addlo(T seq, void *x) Mem_Failed

向seq的高端或低端添加x并返回x。

void Seq_free(T *seq)

释放并清空*seq。seq或*seq为空时产生一个c.r.e.。

int Seq_length(T seq)

返回seq中的元素个数。

void * Seq_get(T seq, int i)

返回seq中的第i个元素。i<0或i≥N时产生一个c.r.e，这里N为seq的长度。

T Seq_new(int hint) Mem_Failed

创建并返回一个空序列。hint为这个序列最大大小的估计值，hint<0时产生一个c.r.e.。

void * Seq_put(T seq, int i, void *x)

把seq的第i个元素修改为x并返回原有值。其c.r.e参见Seq_get。

void * Seq_remhi(T seq)

void * Seq_remlo(T seq)

删除并返回seq的高端或低端元素。seq为空时产生一个c.r.e.。 485

T Seq_seq(void *x, ...) Mem_Failed

创建并返回一个序列，其元素为“...”参数直到第一个空指针。

Set T是隐式的Set_T

除了Set_diff、Set_inter、Set_minus和Set_union函数把空T解释成空集，向任意Set函数传递空member或T都是个c.r.e.。

T Set_diff(T s, T t) Mem_Failed

返回s/t的对称差分子集：元素仅出现在s或t中的集合。s为空且t为空或具有不同的cmp和hash函数的非空s和t，会产生一个c.r.e.。

void Set_free(T *set)

释放并清空*set。set或*set为空时产生一个c.r.e.。

T Set_inter(T s, T t) Mem_Failed

返回s∩t：元素出现在s和t中的几个集合。其c.r.e. 参见Set_diff。

int Set_length(T set)

返回set中的元素个数。

void Set_map(T set, void apply(const void *member, void *cl), void *cl)

为每个member ∈ set调用apply(member, cl)。apply要修改set则产生一个c.r.e.。

int Set_member(T set, const void *member)

如果member ∈ set则返回1；否则返回0。

T Set_minus(T s, T t) Mem_Failed

返回s-t的差：元素出现在s但不出现在t的集合。其c.r.e. 参见Set_diff。

T Set_new(int hint, int cmp(const void *x, const void *y),
unsigned hash(const void *x)) Mem_Failed

创建、初始化并返回一个空集合。关于hint、cmp和hash参见Table_new的说明。

void Set_put(T set, const void *member) Mem_Failed

486 如果需要，给set添加member。

void * Set_remove(T set, const void *member)

如果member ∈ set，则从set中删除member，并返回删除的这个元素；否则；Set_remove返回空。

void ** Set_toArray(T set, void *end) Mem_Failed

创建一个$N+1$-element元素的数组，以未指定的顺序存放set中的N个元素，并返回第一个元素的指针。第N个元素为end。

T Set_union(T s, T t) Mem_Failed

返回s ∪ t：元素出现s或t的集合；其c.r.e. 参见Set_diff。

Stack T是隐式Stack_T

向任意Stack函数传递空T会产生一个c.r.e.。

int Stack_empty(T stk)

如果stk为空则返回1；否则返回0。

void Stack_free(T *stk)

释放并清空*stk，stk或*stk为空时产生一个c.r.e.。

T Stack_new(void) Mem_Failed

返回一个新的空T。

void * Stack_pop(T stk)

弹出并返回stk的顶端元素。stk为空时产生一个c.r.e.。

void Stack_push(T stk, void *x) Mem_Failed

将x推入堆栈顶端。

Str

Str函数处理以null结尾的字符串。位置确定两个字符之间的点；例如，STRING中的位置是：

$$^{1}_{6}\mathrm{S}^{2}_{-5}\mathrm{T}^{3}_{-4}\mathrm{R}^{4}_{-3}\mathrm{I}^{5}_{-2}\mathrm{N}^{6}_{-1}\mathrm{G}^{7}_{0}$$

可以任一种顺序给出任意两个位置。创建字符串的Str函数为其结果分配空间。在下面的介绍中，s[i:j]表示s中位置i到j之间的子字符串。向任意Str函数传递不存在的位置或空字符指针会产生一个c.r.e.，除了Str_catv和Str_map指明的具体情况。 487

int Str_any(const char *s, int i,const char *set)

如果s[i: i+1]出现在set中，就返回s中那个字符后的正值位置，否则返回0。set为空时产生一个c.r.e.。

char * Str_cat(const char *s1, int i1, int j1, const char *s2, int i2, int j2) Mem_Failed

返回s1[i1: j1]连接上s2[i2: j2]的结果。

char * Str_catv(const char *s, ...) Mem_Failed

返回一个由“...”中的三元组所构成的字符串，一直到一个空指针。每个三元组指定一个s[i: j]。

int Str_chr(const char *s, int i, int j, int c)

返回s[i: j]中c的最左边出现位置之前在s中的位置。

int Str_cmp(const char *s1, int i1, int j1, const char *s2, int i2, int j2)

如果s1[i1: j1]<s2[i2: j2]、s1[i1: j1]=s2[i2: j2]或s1[i1: j1]>s2[i2: j2]则分别返回小于0、等于0或大于0的一个整数。

char * Str_dup(const char *s, int i, int j, int n) Mem_Failed

返回s[i: j]的n个副本。n<0时产生一个c.r.e.。

int Str_find(const char *s, int i, int j, const char *str)

返回s[i: j]中str的最左边出现位置之前在s中的位置。

void Str_fmt(int code, va_list *app, int put(int c, void *cl), void *cl,

unsigned char flags[], int width, int precision)

是个Fmt转换函数。它需要三个参数——一个字符串及两个位置——并以printf的%s风格格式化这个子字符串。app或flags为空时产生一个c.r.e.。

int Str_len(const char *s, int i, int j)

返回s[i: j]的长度。

int Str_many(const char *s, int i, int j, const char *set)

返回s[i: j]的最前面从set开始的一串非空字符之后在s中的正值位置；否则返回0。set为空时产生一个c.r.e.。 488

char *Str_map(const char *s, int i, int j, const char *from, const char *to) Mem_Failed

返回一个根据from和to映射s[i: j]中的字符所得到字符串。s[i: j]中出现在from的每个字符都被映射到to中的相应字符。没有处在from的字符映射成自身。如果from和to都为空，就使用它们以前的值。如果s为空，from和to建立一个默认的映射。from和to中仅有一个为空时，strlen(from)≠strlen(to)时，s、from及to都为空时，或者第一次调用from和to都为空时，产生一个c.r.e.。

int Str_match(const char *s, int i, int j, const char *str)

如果s[i: j]以str开始则返回s中的正值位置；否则返回0。str为空时产生一个c.r.e.。

int Str_pos(const char *s, int i)

返回s[i: j]的相应正值位置；减1生成s[i: i+1]的下标。

int Str_rchr(const char *s, int i, int j, int c)

是Str_chr的最右边变体。

char * Str_reverse(const char *s, int i, int j) Mem_Failed

以s[i: j]中字符的相反顺序生成其副本并返回这个副本。

int Str_rfind(const char *s, int i, int j, const char *str)

是Str_find的最右边变体。

int Str_rmany(const char *s, int i, int j, const char *set)

返回s[i: j]末尾从set开始的一串非空字符之后在s中的正值位置；否则返回0。set为空时产生一个c.r.e.。

int Str_rmatch(const char *s, int i, int j, const char *str)

如果s[i: j]以str结束则返回s中str之前的正值位置；否则返回0。str为空时产生一个c.r.e.。

int Str_rupto(const char *s, int i, int j, const char *set)

是Str_upto的最右边变体。

char * Str_sub(const char *s, int i, int j) Mem_Failed

返回s[i: j]。

int Str_upto(const char *s, int i, int j, const char *str)

返回set中任一字符在s[i: j]的最左边出现位置之前在s中位置；否则返回0。set为空时产生一个c.r.e.。

489

Table T是隐式Table_T

向任意Table函数传递空T或空key会产生一个c.r.e.。

void Table_free(T *table)

释放并清空*table。table或*table为空时产生一个c.r.e.。

void * Table_get(T table, const void *key)

返回table中与key相关联的值；如果table中没有key则返回空。

int Table_length(T table)

返回table中键-值对的个数。

void Table_map(T table,void apply(const void *key, void **value, void *cl), void *cl)

对table中每个键-值对按未指定的顺序调用apply(key, &value, cl)。apply要修改table会产生一个c.r.e.。

T Table_new(int hint, int cmp(const void *x, const void *y),

unsigned hash(const void *key)) Mem_Failed

创建、初始化并返回一个可以存放任意数目键-值对的新的空表。hint是对这种键-值对期望个数的估计。hint<0时产生一个c.r.e.。cmp和hash是比较和散列这些键的函数。对于键x和y，如果x<y、x=y或x>y，cmp(x, y)则必须分别返回小于0、等于0或大于0的整数。如果

cmp(x, y)返回0，那么hash(x, y)必须等于hash(y)。如果cmp为空或hash为空，那么Table_new就使用适合于Atom_T键的一个函数。

void * Table_put(T table,const void *key, void *value)　Mem_Failed

把table中key的关联值改变为value并返回以前的值；或者，如果table中没有key就添加key和value，并返回空。

void * Table_remove(T table, const void *key)

从table中删除key-值对并返回删除的那个值。如果table中没有key，Table_remove就不产生作用并返回空。

void ** Table_toArray(T table, void *end)　Mem_Failed

创建一个2*N*+1-element个元素的数组，以未指定顺序存放table中*N*个键-值对，并返回第一个元素的指针。出现在偶数位置数组元素中的键及其相应的值出现在其后的奇数位置元素中；元素2N是end。 490

Text　T是Text_T

typedef struct T { int len; const char *str; } T;

typedef struct Text_save_T * Text_save_T;

T是个描述符；客户可以读取描述符的字段，但是写这些字段则是一个u.r.e.。Text函数通过value接受并返回描述符；向任意Text函数传递str为空或len<0的描述符，会产生一个c.r.e.。

Text为其恒不变的字符串管理内存；在这部分字符串空间进行写操作或用外部手段释放这部分空间都是个u.r.e.。字符串空间中的字符串可以包含空字符，因此不能以空字符结束。

一些Text函数接受用以确定字符之间的点的位置；参见Str。下面的描述符中，s[i : j]表示s中位置i和j之间的子串。

const T Text_cset　= { 256, "\000\001..\376\377" }

const T Text_ascii　= { 128, "\000\001...\176\177" }

const T Text_ucase = { 26, "ABCDEFGHIJKLMNOPQRSTUVWXYZ" }

const T Text_lcase = { 26, "abcdefhijklmnopqrtuvwxyz" }

const T Text_digits = { 10, "0123456789" }

const T Text_null　= { 0, "" }

是经过如上初始化的静态描述符。

int Text_any(T s, int i, T set)

如果s[i : i+1]出现在set中则返回其后在s中的位置；否则返回0。

T Text_box(const char *str, int len)

为客户分配的长度为len的字符串str建立并返回一个描述符。str为空和len<0时产生一个c.r.e.。

T Text_cat(T s1, T s2)　Mem_Failed

返回s1连接s2后的值。

int Text_chr(T s, int i, int j, int c)

参见Str_chr。

int Text_cmp(T s1, T s2)

491 如果s1<s2、s1=s2或s1>s2则分别返回小于0、等于0或大于0的一个整数。

T Text_dup(T s, int n) Mem_Failed

返回s的n个副本。n<0时产生一个c.r.e.。

int Text_find(T s, int i, int j, T str)

参见Str_find。

void Text_fmt(int code, va_list *app, int put(int c, void *cl), void *cl,
unsigned char flags[], int width, int precision)

是个Fmt转换函数。它需要一个指向描述符的指针，并以printf的%s风格格式化字符串。描述符指针app或flags为空时产生一个c.r.e.。

char * Text_get(char *str, int size, T s)

把s.str[0..str.len-1]拷贝到str[0..size-1]，并在末尾附加一个空字符，然后返回str。如果str为空，Text_get则分配空间。str不为空且size<s.len+1时产生一个c.r.e.。

int Text_many(T s, int i, int j, T set)

参见Str_many。

T Text_map(T s, const T *from, const T *to) Mem_Failed

返回一个根据from和 to映射s中的字符所得到字符串。参见Str_map。如果from和to都为空，就使用它们以前的值。如果s为空，from和to建立一个默认的映射。from和to中仅有一个为空或from->len≠to->len时产生一个c.r.e.。

int Text_match(T s, int i, int j, T str)

参见Str_match。

int Text_pos(T s, int i)

参见Str_pos。

T Text_put(const char *str) Mem_Failed

把以空字符结尾的str拷贝到字符串空间并返回它的描述符。str为空时产生一个c.r.e.。

int Text_rchr(T s, int i, int j, int c)

参见Str_rchr。

void Text_restore(Text_save_T *save)

弹出字符串空间赋予save所表示的点。save为空时产生一个c.r.e.。调用Text_restore之后使用其他表示高于save单元的Text_save_T值会产生一个u.r.e.。

T Text_reverse(T s) Mem_Failed

以s中字符的相反顺序生成其副本并返回这个副本。

int Text_rfind(T s, int i, int j, T str)

参见Str_rfind。

int Text_rmany(T s, int i, int j, T str)

492 参见Str_rmany。

int Text_rmatch(T s, int i, int j, T str)

参见Str_rmatch。

int Text_rupto(T s, int i, int j, T set)

参见Str_rupto。

Text_save_T Text_save(void) Mem_Failed

返回对当前字符串空间顶端进行编码的一个隐式指针。

T Text_sub(T s, int i, int j)

返回s[i : j]。

int Text_upto(T s, int i, int j, T set)

参见Str_upto。

Thread T是隐式Thread_T

调用Thread_init之前调用任何Thread函数都会产生一个c.r.e.。

void Thread_alert(T t)

设置t的未决警告标志并使t可运行。下次t运行时或调用模块化Thread、Sem或Chan原语时清空其标志并引发Thread_Alerted。t为空或给一个不存在的线程命名会产生一个c.r.e.。

void Thread_exit(int code)

终止调用线程并将code传递给任一等待调用线程结束的线程。当最后一个线程调用Thread_exit时，程序就终止于exit(code)。

int Thread_init(int preempt, ...)

将Thread初始化为无优先权（preempt = 0）或有优先权（preempt = 1）调度并返回preempt；如果preempt=1且不支持优先权调度则返回0. Thread_init可以接受实现所定义的其他参数；参数列表必须以空字符结束。多次调用Thread_init会产生一个c.r.e.。

int Thread_join(T t) Thread_Alerted

挂起调用线程直到线程t终止。当t终止时，Thread_join返回t的返回码。如果t为空，调用线程就等待其他所有线程结束，然后返回0。t要给调用线程命名或多个线程传递空t则会产生一个c.r.e.。

493

T Thread_new(int apply(void *), void *args, int nbytes, ...),

Thread_Failed

创建、初始化并启动一个新线程，然后返回其句柄。如果nbytes=0，新线程就执行Thread_exit(apply(args))；否则执行Thread_exit(apply(p))，这里p指向一个从args开始的nbytes内存块副本。新线程开始于它自己的空异常堆栈。Thread_new可以接受实现所定义的其他参数；参数列表必须以空字符结束。apply为空或者args为空且nbytes<0时产生一个c.r.e.。

void Thread_pause(void)

放弃处理器供另一线程使用，可能是调用方。

T Thread_self(void)

返回调用线程的句柄。

XP T是XP_T

typedef unsigned char *T;

一个扩展精度的无符号整数按2^8为底表示成n位数组，最无效数字在第一位。大多数XP函数把n当作与源和目的Ts一起的一个参数；n<1或n不是Ts的相应长度时产生一个u.r.e.。向任一XP函数传递一个空T或太小的T都会产生一个u.r.e.。

int XP_add(int n, T z, T x, Ty, int carry)

将z[0..n-1]设为x+y+carry并返回z[n-1]中的进位。carry必须为0或1。

int XP_cmp(int n, T x, T y)

如果x<y、x=y或x>y则分别返回小于0、等于0或大于0的整数。

int XP_diff(int n, T z, T x, int y)

将z[0..n-1]设为x-y，这里y是一位数字，并返回z[n-1]中的借位。$y>2^8$时产生一个u.r.e.。

int XP_div(int n, T q, T x, int m, T y, T r, T tmp)

将q[0..n-1]设为x[0..n-1]/y[0..m-1]，r[0..m-1]设为x[0..n-1] mod y[0..m-1]，如果$y\neq0$，返回1。如果y=0，XP_div返回0且保持q和r不变。tmp必须至少有n+m+2个数字。q或r是x或
494 y中其一时、q及r都同为T时或者tmp太小时则是个u.r.e.。

unsigned long XP_fromint(int n, T z, unsigned long u)

将z[0..n-1]设为u mod 2^{8n}并返回$u/2^{8n}$。

int XP_fromstr(int n, T z, const char *str, int base, char **end)

将str解释为以base为底的无符号整数，并在转换中把z[0..n-1]用作初始值，然后返回转换的第一个非零结果。如果end不为空，*end指向str中终止扫描或生成非零进位的那个字符。参见AP_fromstr。

int XP_length(int n, T x)

返回x的长度；也即，下标加上x[0..n-1]中的一个最有效非零位。

void XP_lshift(int n, T z, int m, T x, int s, int fill)

将z[0..n-1]设为x[0..m-1]左移s位后的结果，并用fill填充空出的位，fill必须为0或1。s<0时产生一个u.r.e.。

int XP_mul(T z, int n, T x, int m, T y)

把x[0..n-1]·y[0..m-1]加到z[0..n+m-1]并返回z[n+m-1]的结果。如果z=0，XP_mul则计算x·y。z是与x或y相同的T时产生一个u.r.e.。

int XP_neg(int n, T z, T x, int carry)

将z[0..n-1]设为~x+carry，这里carry为0或1，并返回z[n-1]的结果。

int XP_product(int n, T z, T x, int y)

将z[0..n-1]设为x·y，这里y为一位的数字，并返回z[n-1]的结果。$y\geq2^8$时产生一个u.r.e.。

int XP_quotient(int n, T z, T x, int y)

将z[0..n-1]设为x/y，这里y为一位的数字，并返回x mod y。y=0或$y\geq2^8$时产生一个u.r.e.。

void XP_rshift(int n, T z, int m, T x, int s, int fill)

右移；参见XP_lshift。如果n>m，多余的位就当作它们等于fill。

int XP_sub(int n, T z, T x, T y, int borrow)

将z[0..n-1]设为x-y-borrow，并返回z[n-1]中的借位。borrow必须为0或1。

int XP_sum(int n, T z, T x, int y)

将z[0..n-1]设为x+y，这里y为一位的数字，并返回z[n-1]的结果。$y \geqslant 2^8$时产生一个u.r.e.。

unsigned long XP_toint(int n, T x)

返回x mod (ULONG_MAX+1)。 495

char * XP_tostr(char *str, int size, int base, int n, T x)

用x以base为底数的字符表示形式填充str[0..size-1]，并将x设为0，然后返回str。str为空、size太小或者base<2或base>36时产生一个c.r.e.。 496

参考书目

每个文献后面是本书英文原书引用它的页码。

Each entry is followed by the page numbers on which the entry is cited.

Abelson, H., and G. J. Sussman. 1985. *Structure and Interpretation of Computer Programs*. Cambridge, Mass.: MIT Press. (114)

Adobe Systems Inc. 1990. *PostScript Language Reference Manual* (second edition). Reading, Mass.: Addison-Wesley. (133)

Aho, A. V., B. W. Kernighan, and P. J. Weinberger. 1988. *The AWK Programming Language*. Reading, Mass.: Addison-Wesley. (13, 132, 265)

Aho, A. V., R. Sethi, and J. D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools*. Reading, Mass.: Addison-Wesley. (43)

American National Standards Institute. 1990. *American National Standard for Information Systems — Programming Language — C*. ANSI X3.159-1989. New York. (12)

Andrews, G. R. 1991. *Concurrent Programming: Principles and Practice*. Menlo Park, Calif.: Addison-Wesley. (463)

Appel, A. W. 1991. Garbage collection. In P. Lee (ed.), *Topics in Advanced Language Implementation Techniques*, 89-100. Cambridge, Mass.: MIT Press. (100)

Austin, T. M., S. E. Breach, and G. S. Sohi. 1994. Efficient detection of all pointer and array access errors. *Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation, SIGPLAN Notices* 29 (7), 290-301. July. (85)

Barrett, D. A., and B. G. Zorn. 1993. Using lifetime predictors to improve allocation performance. *Proceedings of the SIGPLAN'93 Conference on Programming Language Design and Implementation, SIGPLAN Notices* 28 (6), 187-196. June. (98)

Bentley, J. L. 1982. *Writing Efficient Programs*. Englewood Cliffs, N.J.: Prentice-Hall. (13)

Boehm, H., R. Atkinson, and M. Plass. 1995. Ropes: An alternative to strings. *Software — Practice and Experience* 25 (12), 1315-30. December. (294)

Boehm, H., and M. Weiser. 1988. Garbage collection in an uncooperative environment. *Software — Practice and Experience* 18 (9), 807–20. September. (100)

Briggs, P., and L. Torczon. 1993. An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems* 2 (1–4), 59–69. March–December. (213, 213)

Brinch-Hansen, P. 1994. Multiple-length division revisited: A tour of the minefield. *Software — Practice and Experience* 24 (6), 579–601. June. (321)

Budd, T. A. 1991. *An Introduction to Object-Oriented Programming.* Reading, Mass.: Addison-Wesley. (31)

Char, B. W., K. O. Geddes, G. H. Gonnet, B. L. Leong, M. B. Monagan, and S. M. Watt. 1992. *First Leaves: A Tutorial Introduction to Maple V.* New York: Springer-Verlag. (354)

Clinger, W. D. 1990. How to read floating-point numbers accurately. *Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementation, SIGPLAN Notices* 25 (6), 92–101. June. (239, 402)

Cohen, J. 1981. Garbage collection of linked data structures. *ACM Computing Surveys* 13 (3), 341–67. September. (100)

Cormack, G. V. 1988. A micro-kernel for concurrency in C. *Software — Practice and Experience* 18 (5), 485–91. May. (465)

Ellis, M. A., and B. Stroustrup. 1990. *The Annotated C++ Reference Manual.* Reading, Mass.: Addison-Wesley. (31, 63)

Evans, D. 1996. Static detection of dynamic memory errors. *Proceedings of the SIGPLAN'96 Conference on Programming Language Design and Implementation, SIGPLAN Notices* 31 (5), 44–53. May. (14, 85)

Fraser, C. W., and D. R. Hanson. 1995. *A Retargetable C Compiler: Design and Implementation.* Menlo Park, Calif.: Addison-Wesley. (xi, xiii, 13, 42, 65, 98, 468)

Geddes, K. O., S. R. Czapor, and G. Labahn. 1992. *Algorithms for Computer Algebra.* Boston: Kluwer Academic. (355)

Gimpel, J. F. 1974. The minimization of spatially-multiplexed character sets. *Communications of the ACM* 17 (6), 315–18. June. (213)

Goldberg, D. 1991. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys* 23 (1), 5–48. March. (238, 403)

Gosling, J., D. S. H. Rosenthal, and M. J. Arden. 1989. *The NeWS Book.*

New York: Springer-Verlag. (465)

Griswold, R. E. 1972. *The Macro Implementation of SNOBOL4.* San Francisco: W. H. Freeman (out of print). (42, 132, 293)

———. 1980. Programming techniques using character sets and character mappings in Icon. *The Computer Journal* 23 (2), 107–14. (264)

Griswold, R. E., and M. T. Griswold. 1986. *The Implementation of the Icon Programming Language.* Princeton, N.J.: Princeton University Press. (133, 158)

———. 1990. *The Icon Programming Language* (second edition). Englewood Cliffs, N.J.: Prentice-Hall. (xii, 132, 158, 169, 180, 264, 266, 293)

Grunwald, D., and B. G. Zorn. 1993. CustoMalloc: Efficient synthesized memory allocators. *Software — Practice and Experience* 23 (8), 851–69. August. (85)

Hansen, W. J. 1992. Subsequence references: First-class values for substrings. *ACM Transactions on Programming Languages and Systems* 14 (4), 471–89. October. (294, 295)

Hanson, D. R. 1980. A portable storage management system for the Icon programming language. *Software — Practice and Experience* 10 (6), 489–500. June. (294, 295)

———. 1987. Printing common words. *Communications of the ACM* 30 (7), 594–99. July. (13)

———. 1990. Fast allocation and deallocation of memory based on object lifetimes. *Software — Practice and Experience* 20 (1), 5–12. January. (98)

Harbison, S. P. 1992. *Modula-3.* Englewood Cliffs, N.J.: Prentice-Hall. (31)

Harbison, S. P., and G. L. Steele Jr. 1995. *C: A Reference Manual* (fourth edition). Englewood Cliffs, N.J.: Prentice-Hall. (12)

Hastings, R., and B. Joyce. 1992. Purify: Fast detection of memory leaks and access errors. *Proceedings of the Winter USENIX Technical Conference,* San Francisco, 125–36. January. (85)

Hennessy, J. L., and D. A. Patterson. 1994. *Computer Organization and Design: The Hardware/Software Interface.* San Mateo, Calif.: Morgan Kaufmann. (238, 321, 322)

Hoare, C. A. R. 1978. Communicating sequential processes. *Communications of the ACM* 21 (8), 666–77. August. (464)

Horning, J., B. Kalsow, P. McJones, and G. Nelson. 1993. Some Useful Modula-3 Interfaces. Research Report 113, Palo Alto, Calif.: Systems

Research Center, Digital Equipment Corp. December. (31, 180)

International Organization for Standardization. 1990. *Programming Languages — C.* ISO/IEC 9899:1990. Geneva. (12)

Institute for Electrical and Electronic Engineers. 1995. *Information Technology — Portable Operating Systems Interface (POSIX) — Part 1: System Application Programming Interface (API) — Amendment 2: Threads Extension [C Language].* IEEE Standard 1003.1c-1995. New York. Also ISO/IEC 9945-2:1990c. (464)

Jaeschke, R. 1991. *The Dictionary of Standard C.* Horsham, Penn.: Professional Press Books. (12)

Kernighan, B. W., and R. Pike. 1984. *The UNIX Programming Environment.* Englewood Cliffs, N.J.: Prentice-Hall. (13, 13)

Kernighan, B. W., and P. J. Plauger. 1976. *Software Tools.* Reading, Mass.: Addison-Wesley. (12, 295)

———. 1978. *The Elements of Programming Style* (second edition). Englewood Cliffs, N.J.: Prentice-Hall. (13)

Kernighan, B. W., and D. M. Ritchie. 1988. *The C Programming Language* (second edition). Englewood Cliffs, N.J.: Prentice-Hall. (12, 13, 85, 86)

Kleiman, S., D. Shah, and B. Smaalders. 1996. *Programming with Threads.* Upper Saddle River, N.J.: Prentice-Hall. (464, 468)

Knuth, D. E. 1973a. *The Art of Computer Programming: Volume 1, Fundamental Algorithms* (second edition). Reading, Mass.: Addison-Wesley. (13, 85, 100, 113, 196)

———. 1973b. *The Art of Computer Programming: Volume 3, Searching and Sorting.* Reading, Mass.: Addison-Wesley. (42, 42)

———. 1981. *The Art of Computer Programming: Volume 2, Seminumerical Algorithms* (second edition). Reading, Mass.: Addison-Wesley. (321, 354, 355)

———. 1992. *Literate Programming.* CSLI Lecture Notes Number 27. Stanford, Calif.: Center for the Study of Language and Information, Stanford Univ. (12)

Koenig, A. 1989. *C Traps and Pitfalls.* Reading, Mass.: Addison-Wesley. (13)

Larson, P. 1988. Dynamic hash tables. *Communications of the ACM* 31 (4), 446–57. April. (133, 134)

Ledgard, H. F. 1987. *Professional Software Volume II. Programming Practice.* Reading, Mass.: Addison-Wesley. (13)

Maguire, S. 1993. *Writing Solid Code.* Redmond, Wash.: Microsoft Press. (13, 31, 64, 85, 86)

Marlin, C. D. 1980. *Coroutines: A Programming Methodology, a Language Design and an Implementation.* Lecture Notes in Computer Science 95, New York: Springer-Verlag. (464)

McConnell, S. 1993. *Code Complete: A Practical Handbook of Software Construction.* Redmond, Wash.: Microsoft Press. (13)

McIlroy, M. D. 1968. Coroutines. Unpublished Report, Murray Hill, N.J.: Bell Telephone Laboratories. May. (464)

———. 1990. Squinting at power series. *Software — Practice and Experience* 20 (7), 661–83. July. (464)

McKeeman, W. M., J. J. Horning, and D. B. Wortman. 1970. *A Compiler Generator*, Englewood Cliffs, N.J.: Prentice-Hall (out of print). (294)

Meyer, B. 1992. *Eiffel: The Language.* London: Prentice-Hall International. (31, 63)

Musser, D. R., and A. Saini. 1996. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library.* Reading, Mass.: Addison-Wesley. (31)

Nelson, G. 1991. *Systems Programming with Modula-3.* Englewood Cliffs, N.J.: Prentice-Hall. (31, 63, 169, 464)

Parnas, D. L. 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* 5 (12), 1053–58. December. (30)

Pike, R. 1990. The implementation of Newsqueak. *Software — Practice and Experience* 20 (7), 649–59. July. (464)

Plauger, P. J. 1992. *The Standard C Library.* Englewood Cliffs, N.J.: Prentice-Hall. (12, 30, 238, 264)

Press, W. H., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. 1992. *Numerical Recipes in C: The Art of Scientific Computing* (second edition). Cambridge, England: Cambridge University Press. (402, 402)

Pugh, W. 1990. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM* 33 (6), 668–76. June. (181)

Ramsey, N. 1994. Literate programming simplified. *IEEE Software* 13 (9), 97–105. September. (13)

Reppy, J. H. 1997. *Concurrent Programming in ML.* Cambridge, England: Cambridge University Press. (465)

Richter, J. 1995. *Advanced Windows: The Developer's Guide to the Win32 API for Windows NT 3.5 and Windows 95.* Redmond, Wash.: Microsoft Press. (468)

Roberts, E. S. 1989. Implementing Exceptions in C. Research Report 40, Palo Alto, Calif.: Systems Research Center, Digital Equipment Corp. March. (63, 64)

———. 1995. *The Art and Science of C: An Introduction to Computer Science.* Reading, Mass.: Addison-Wesley. (31, 264)

Schneier, B. 1996. *Applied Cryptography: Protocols, Algorithms, and Source Code in C* (second edition). New York: John Wiley. (402)

Sedgewick, R. 1990. *Algorithms in C.* Reading, Mass.: Addison-Wesley. (xii, xiv, 13, 42, 134, 196)

Sewell, W. 1989. *Weaving a Program: Literate Programming in WEB.* New York: Van Nostrand Reinhold. (13)

Steele, Jr., G. L., and J. L. White. 1990. How to print floating-point numbers accurately. *Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementation, SIGPLAN Notices* 25 (6), 112–26. June. (239, 239)

Stevens, W. R. 1992. *Advanced Programming in the UNIX Environment.* Reading, Mass.: Addison-Wesley. (465)

Tanenbaum, A. S. 1995. *Distributed Operating Systems.* Englewood Cliffs, N.J.: Prentice-Hall. (464, 467)

Ullman, J. D. 1994. *Elements of ML Programming.* Englewood Cliffs, N.J.: Prentice-Hall. (114)

Vo, K. 1996. Vmalloc: A general and efficient memory allocator. *Software — Practice and Experience* 26 (3), 357–74. March. (99)

Wampler, S. B., and R. E. Griswold. 1983. Co-expressions in Icon. *Computer Journal* 26 (1), 72–78. January. (464)

Weinstock, C. B., and W. A. Wulf. 1988. Quick fit: An efficient algorithm for heap storage management. *SIGPLAN Notices* 23 (10), 141–48. October. (85)

Wolfram, S. 1988. *Mathematica: A System for Doing Mathematics by Computer.* Menlo Park, Calif.: Addison-Wesley. (354)

Zorn, B. G. 1993. The measured cost of conservative garbage collection. *Software — Practice and Experience* 23 (7), 733–56. July. (100)

索　引

下面列出的页码标识符分别以粗体、斜体、罗马体，以及下划线表示。粗体页码表示参考页面中包含该标识符的定义；斜体页码表示参考页面中包含该标识符在接口、实现或者示例中的重要使用；罗马字体页码表示参考页面包含该标识符在文本中的使用。带下划线的页码则表示参考页面包含该标识符的接口规格说明。

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

[General Information]
书名=C语言接口与实现
作者=
页数=379
SS号=0
出版日期=