Managing Projects with GNU Make

第三版
完全修订版

GNU Make

项目管理

Robert Mecklenburg 著
*O'Reilly Taiwan*公司 编译

O'REILLY®
東南大學出版社

Unix Programming

O'REILLY®

GNU Make项目管理

*make*是Unix和其他操作系统上最持久的工具之一。自1970年问世以来，*make*至今仍旧是大多数程序开发项目的核心工具，它甚至被用来编译Linux内核。阅读本书，读者将可以了解，尽管出现了许多新兴的竞争者为何*make*仍旧是开发项目中编译软件的首选工具。

简单就是*make*欲达成的目标：在你变更源代码文件之后，想要重编译你的程序或其他输出文件之际，*make*会检查时间戳，找出被变更的文件并进行必要的重编译动作，因此不会浪费时间去重编译其他文件。为了达到这个目标，*make*提供了许多选项让你能够操作多个目录，为不同的平台编译不同版本的程序以及自定义编译方法。

本书第三版的重点是GNU *make*，这个版本的*make*已经成为行业标准。本书将会探索GNU *make*所提供的强大扩充功能。GNU *make*之所以广受欢迎是因为它是一个自由软件，并且几乎可以在包括微软Windows（作为Cygwin项目的一部分）的每个平台上使用。

Robert Mecklenburg是本书第三版的作者，他对多种平台和语言使用*make*已经有数十年的经验。在本书中，他会很热心地告诉你如何提升编译工作的效率、降低维护工作的困难度、避免错误以及让你彻底了解*make*在做什么。他还在论述C++和Java的章节中为采用这些语言的项目提供经过优化的*makefile*设定项目，他甚至还会讨论到用来制作本书的*makefile*。

O'Reilly Media, Inc. 授权东南大学出版社出版

ISBN 7-5641-0352-3

定价：37.00元

有关此电子图书的说明

本人由于一些便利条件，可以帮您提供各种中文电子图书资料，且质量均为清晰的 PDF 图片格式，质量要高于网上大量传播的一些超星 PDG 的图书。方便阅读和携带。只要图书不是太新，文学、法律、计算机、人文、经济、医学、工业、学术等方面的图书，我都可以帮您找到电子版本。所以，当你想要看什么图书时，可以联系我。我的 QQ 是：85013855，大家可以在 QQ 上联系我。

此 PDF 文件为本人亲自制作，请各位爱书之人尊重个人劳动，敬请您不要修改此 PDF 文件。因为这些图书都是有版权的，请各位怜惜电子图书资源，不要随意传播，否则，这些资源更难以得到。

GNU Make 项目管理

第三版

Robert Mecklenburg 著

O'Reilly Taiwan 公司 编译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

O'Reilly Media, Inc. 授权东南大学出版社出版

东南大学出版社

图书在版编目（CIP）数据

GNU Make 项目管理：第3版 /（美）梅克伦伯格（Mecklenburg, R.），著；O'Reilly Taiwan 公司编译．—南京：东南大学出版社，2006.7

书名原文：Managing Projects with GNU Make,Third Edition

ISBN 7-5641-0352-3

I. G... II. ①梅 ... ② O... III. 操作系统（软件），GNU Make IV. TP316.7

中国版本图书馆 CIP 数据核字（2006）第 043509 号

江苏省版权局著作权合同登记

图字：10-2005-288 号

©2004 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Southeast University Press, 2005. Authorized translation of the English edition, 2004 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O'Reilly Media, Inc. 出版 2004。

简体中文版由东南大学出版社出版 2005。英文原版的翻译得到 O'Reilly Media, Inc. 的授权，此简体中文版的出版和销售得到出版权和销售权的所有者 —— O'Reilly Media, Inc. 的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

书　　名 / GNU Make 项目管理 第三版
书　　号 / ISBN 7-5641-0352-3
责任编辑 / 张烨
封面设计 / Edie Freedman，张健
出版发行 / 东南大学出版社（press.seu.edu.cn）
地　　址 / 南京四牌楼 2 号（邮编 210096）
印　　刷 / 扬中市印刷有限公司
开　　本 / 787 毫米 × 980 毫米　16 开本　18.75 印张　315 千字
版　　次 / 2006 年 7 月第 1 版　2006 年 7 月第 1 次印刷
印　　数 / 0001-3000 册
定　　价 / 37.00 元（册）

O'Reilly Media, Inc. 介绍

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

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

从最畅销的《The Whole Internet User's Guide & Catalog》（被纽约公共图书馆评为二十世纪最重要的50本书之一）到GNN（最早的Internet门户和商业网站），再到WebSite（第一个桌面PC的Web服务器软件），O'Reilly Media, Inc. 一直处于Internet发展的最前沿。

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

目录

序 1

前言 3

第一部分 基本概念

第一章 如何编写一个简单的 makefile 11

工作目标与必要条件 12

检查依存关系 14

尽量减少重新编译的工作量 15

调用 make 16

Makefile 的基本语法 17

第二章 规则 19

具体规则 20

变量 25

以 VPATH 和 vpath 来查找文件 27
模式规则 31
隐含规则 35
特殊工作目标 40
自动产生依存关系 41
管理程序库 45

第三章 变量与宏 52
变量的用途 53
变量的类型 54
宏 56
何时扩展变量 58
工作目标与模式的专属变量 61
变量来自何处 62
条件指令与引入指令的处理 65
标准的 make 变量 69

第四章 函数 72
用户自定义函数 72
内置函数 75
高级的用户自定义函数 92

第五章 命令 100
解析命令 100
使用哪个 shell 109
空命令 110
命令环境 110
对命令脚本求值 111
命令行的长度限制 112

第二部分 高级与特别的议题

第六章 大型项目的管理 119

递归式 make 120
非递归式 make 129
大型系统的组件 136
文件系统的布局 138
自动编译与测试 140

第七章 具可移植性的 makefile 141

可移植性的若干内容 142
Cygwin 143
管理程序和文件 146
使用不具可移植性的工具 149
automake 151

第八章 C 与 C++ 153

分开源文件与二进制文件 153
只读的源文件树 161
产生依存关系 161
支持多个二进制文件树 166
部分的源文件树 168
引用编译结果、程序库以及安装程序 169

第九章 Java 171

make 的替代方案 172
一个通用的 Java makefile 175
编译 Java 179

管理 jar 187
引用树与来自第三方的 jar 文件 189
Enterprise JavaBeans 190

第十章 改进 make 的效能 194
基准测试 194
找出瓶颈与处理瓶颈 199
并行式 make 202
分布式 make 206

第十一章 makefile 实例 208
本书的 makefile 208
Linux 内核的 makefile 229

第十二章 makefile 的调试 241
make 的调试功能 241
编写用于调试的代码 248
常见的错误信息 254

第三部分 附录

附录一 运行 make 261

附录二 越过 make 的极限 264

索引 275

序

make 实用程序是一个令人满意的仆人，它总是随伺在侧，给人方便，对你而言就像是不可或缺的伙伴。make 起初是一个未能充分发挥潜力的职员，你将一些临时的工作丢给它做，然后它渐渐掌控了整个企业（这是许多小说和电影中常见的情节）。

就在每个项目都被我改成以make 为基础之际，我的老板 —— 本书第一版的作者Steve Talbott注意到了我的狂热行为，邀请我为本书编写第二版。在我的人生历练中这的确是关键性的成长（也是相当大的冒险），我因此而进入O'Reilly的美妙世界，但是我们实在不知道第二版能在市场上存活多久。一版能存活 13 年吗？

下面的概述列出了自本书第二版发行以来 make 的发展：

- 本书第二版发行时GNU 版的make 已经是大多数程序员的选择，俨然成为业界的标准。
- GNU/Linux 的兴起让 GNU 编译器工具链的使用更为广泛，其中包括GNU 版的make。举例来说，Linux 内核本身就相当依赖 GNU 版make 所提供的扩展功能，正如本书第十一章所描述的那样。
- 以BSD 的变体（Darwin）为核心（core）的 Mac OS X 继续向 GNU 工具链以及GNU 版的 make 迈进。
- 越来越多的诀窍被发现，让你能够健全、无错误、可移植及灵活地使用make。对于大型项目上常见的问题，已经在社群中逐渐形成了标准的解决方案。已经到了将这些解决方案立言成书的时候了，这就是本书要做的事情。
- 尤其是出现了将make 应用在C++和Java语言中的新需求，make 开发出来时这些语言尚不存在。以下的例子可以说明make 出现的年代：最初的make 具有两项特

殊的功能，一个是支持FORTRAN的两个变体（这个功能还在！），一个是与SCCS的不怎么有用的集成。

- 即使有诸多限制，make仍旧是所有计算机开发项目中最重要的工具，恐怕当年对make的批评者或洞察者都没预料到这一点。这13年来，有意取代make的新工具如雨后春笋般不断推陈出新，它们都想超越make在设计上的限制，其中也不乏众多值得赞赏的巧思。不过，make的简单易用仍使它立于不败之地。

当我观察到这些趋势之后，十年前为本书编写新版的想法又涌上心头。不过我意识到自己的经验浅薄并不足以担负此重责大任。最后，我找到了Robert Mecklenburg，他的专业能力得到O'Reilly所有同仁的肯定。能将这本书交由他全权负责实在太好了，我则退居幕后成为本书的编辑，这让我的名字又可以出现在本书的版权页上。

Robert对他的博士学位保持低调，不过他思考的深度和精确度却在本书中展露无遗。或许更重要的是他把焦点放在实用性上。他会告诉你如何执行得更有效率，以及让你知道如何进行调试。

这是一个重大的时刻：O'Reilly最早期和最持久的一本书出新版了。坐下来，了解一下，一个保守的小工具何以有此能耐让几乎每个项目都要使用它。不要安于老旧而无法令人满意的*makefile* —— 开发你的潜力就在今朝。

— Andy Oram
Editor, O'Reilly Media
August 19, 2004

前言

迈向第三版

1979年首次遇到make的时候我还是Berkeley的大学生。当时我正为能够使用“最新的”设备感到兴奋不已：一台DEC PDP 11/70（具有128 kilobytes的RAM）、一台ADM 3a（具有屏幕的终端机）、Berkeley Unix以及另外20个同时上线的用户！记得有一次，大家在赶作业的时候从我键入账号名称到我看到命令提示符一共花了5分钟的登录时间。

毕业之后，当我再次使用Unix的时候已经是1984年了，这次我是美国太空总署Ames研究中心的程序员。我们买了第一部以微型计算机为基础的Unix系统，它的配置包含一个68000（不是68010或20）微处理器、1 megabyte的RAM以及Unix Version 7——只能有6个同时上线的用户。我所参与的最后一个项目就是使用C程序语言以及yacc/lex命令语言（当然还包括make）来实现出一个交互式卫星数据分析系统。

1988年，我返回学校并且参与构建“曲线几何建模”（spline-based geometric modeler）的项目。这个系统使用了大约120000行的C程序，涵盖了20个左右的可执行文件。这个系统的编译方法就是使用一个手工打造的工具genmakefile（它的功能类似于imake）将*makefile*模板展开成一般的*makefile*。这个工具会进行简易的文件引入、条件编译以及使用自定义的逻辑来管理源文件树和二进制文件树。当时普遍认为，make必须使用此类封装程序（wrapper）才算是完整的编译工具。直到几年前我发现GNU项目以及GNU make，我才了解到大概不再需要封装程序了。我重新建立了不使用模板或产生器的编译系统。这个编译系统被移植到了5种Unix版本中，并且包括独立的源文件树和二进制文件树、每夜自动编译、以及以编译系统填补短缺的二进制文件的方式来支持开发人员进行部分调出的动作。

下一个重要的make使用经验是在1996年。这是一个商用CAD系统，我的工作是将200万行的C++（以及40万行Lisp）程序从Unix移植到Windows NT，使用Microsoft C++编译器进行编译的工作。我就是在那个时候发现Cygwin项目的。这个编译系统还支持独立的源文件树和二进制文件树、多种Unix、几种图形功能、每夜自动编译和测试、以引用编译结果让开发人员进行部分调出的动作。

2000年，我的工作是以Java编写实验室信息管理系统。这是我工作了那么多年之后首次遇到的完全不同的开发环境之一。参与项目的程序员大部分来自于Windows背景，而且Java似乎是他们所使用的第一个程序语言。这个编译环境几乎是由一个商用Java集成开发环境（Integrated Development Environment，简称IDE）所产生的项目文件构成的。尽管项目文件已经可以使用，但是它却很少被马上拿来使用，程序员们通常会坐在彼此的屏幕前处理许多编译问题。

当然，我开始使用make来编写编译系统，但是一个奇特的事情发生了：许多开发人员根本不愿意使用任何命令行工具。此外，许多人无法准确理解环境变量、命令行选项之类的概念，也不知道这些工具如何用来编译程序。IDE将这些问题都藏起来了。为解决这些问题，我所编写的编译系统变得更加复杂。我开始加入更好的错误信息、先决条件的检查、开发人员机器配置的管理以及对IDE的支持。

于是，GNU make使用手册被我读过不少于10次。当我在寻找更多资料的时候，我发现到了本书的第二版，它提供了许多有用的数据，不过缺少GNU make方面的细节。这并不令人感到惊讶，想想看它的出版时间。这是一本经得起时间考验的书，不过到了2003年是需要更新了。本书第三版的重点是GNU make。诚如Paul Smith（GNU make的维护者）所说："编写具可移植性的'*makefile*'是在自找麻烦，使用具可移植性的*make*吧！"

第三版有哪些新的内容

本书几乎所有内容都是新的。我将这些内容划分成三个部分：

第一部分 基本概念。适度说明GNU make的功能以及这些功能的使用方法。

第一章 如何编写简单的makefile。以简单但完整的范例来简介make。这一章将会说明make的基本概念，如工作目标以及必要条件，并解释*makefile*的语法。这应该可以让你具备编写*makefile*的能力。

第二章 规则。将会探讨规则的结构和语法。除了旧式的后缀规则（suffix rule），这一章还会非常详细地说明具体规则（explicit rule）和模式规则（pattern rule）。特殊的工作目标以及依存关系的产生也会在此处被讨论到。

第三章 变量与宏。将会说明简单变量与递归变量。这一章还会探讨当变量被展开时*makefile*是如何被解析的以及条件指令的处理。

第四章 函数。将会查看GNU make所支持的各种内置函数。此处也会以各种范例，包含一般的与深层次的概念来说明用户自定义函数。

第五章 命令。将会说明脚本的细节，内容涵盖脚本的解析与求值。此处也会探讨命令修饰符、命令结束状态的检查以及环境变量。我们还会探索命令行长度限制的问题，以及解决这些问题的若干方法。此刻你已经能够了解本书所要探讨的所有GNU make的功能。

第二部分 深入与特别的议题。包含了比较多的议题，如将make应用在大型项目上、可移植性以及调试等。

第六章 大型项目的管理。将会探讨以make编译大型项目时可能会遇到的许多问题。第一个议题是如何进行make的递归调用，以及如何使用单一非递归的*makefile*来实现前者所用到的许多*makefile*。此外，我们还会探讨大型系统的其他议题，像文件系统的配置、项目组件的管理以及自动化编译与测试。

第七章 具可移植性的makefile。将会探讨*makefile*在各种Unix操作系统与Windows系统间的可移植性。此处还会讨论Cygwin的Unix模拟环境，以及不具可移植性的文件系统功能与工具所引发的问题。

第八章 C与C++。将会举几个"如何分开源文件树和二进制文件树以及如何建立仅供读取的源文件树"的特例。此处会再次提到依存关系分析，不过这次将会强调与程序语言有关的解决方案。这一章与下一章将会探讨第六章所提到的许多延伸议题。

第九章 Java。将会说明如何把make应用在以Java为基础的开发环境中。此处还会提到管理CLASSPATH变量、编译大量文件、创建jar以及构造Enterprise JavaBeans的技术。

第十章 改进make的性能。首先会回顾若干make操作的性能特性，以作为如何编写具有效率的*makefile*的立论基础。此处还会探讨如何找出和解决瓶颈的技术，以及GNU make的并行编译功能。

第十一章 makefile实例。将会提供两个复杂的*makefile*实例。第一个实例是用来建立本书的*makefile*。这是个值得一看的例子，部分是由于这是对自动化的相当极端的应用，部分是由于它将make应用在非传统的领域。另一个实例摘录自Linux 2.6 kbuild系统。

第十二章 makefile的调试。我们将会钻研修复*makefile*的魔法。这一章将会介绍“如何发现make背地里在做什么以及如何减轻开发期痛苦”的技术。

第三部分 附录。包含了补充资料。

附录一 运行make。提供了GNU make命令行选项的参考指导。

附录二 越过make的极限。将会探索GNU make两个不太可能被用到的功能：管理数据结构以及进行算数运算。

排版约定

斜体字（*Italic*）

用来表示新项目、网址、电子邮件地址、文件名、文件扩展名、路径名称以及目录。

等宽字（Constant Width）

用来表示源代码命令、命令行选项、文件的内容或是命令的输出。

等宽黑体字（**Constant Width Bold**）

用来表示应该由用户逐字键入的命令或其他文字。

等宽斜体字（*Constant Width Italic*）

其所标示的文字应该被替换成用户所提供的值。

范例程序代码的使用办法

这本书可以协助你把工作做好。一般而言，你可以在自己的应用程序和说明文件中使用本书的程序代码。除非你要重制重要的程序代码，否则不必取得我们的许可。例如，你使用本书的程序代码片段写了一个应用程序，并不需要取得我们的许可；但是，把O'Reilly书籍的程序范例制作成光盘贩卖或散布，就需要取得授权。引用本书的文字和范例程序代码来回答问题，不需要取得许可；但把本书大量的程序范例整合到你的产品的说明文件中，则需要取得授权。

虽然不是必要，但若能注明来源我们会很感谢。注明来源通常包括书名、作者、出版商以及ISBN。例如：Managing Projects with GNU Make, Third Edition, by Robert Mecklenburg. Copyright 2005 O'Reilly Media, Inc., 0-596-00610-1。

如果你对书中程序范例的使用情况有别于上述情况，不用客气，尽管和我们联络：*permissions@oreilly.com*。

建议与评论

本书的内容都经过测试，尽管我们做了最大的努力，但错误和疏忽仍然是在所难免的。如果你发现有什么错误，或者是对将来的版本有什么建议，请通过下面的地址告诉我们：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

100080 北京市海淀区知春路 49 号希格玛公寓 B 座 809 室
奥莱理软件（北京）有限公司

询问技术问题或对本书的评论，请发电子邮件到：

info@mail.oreilly.com.cn

与本书有关的在线信息（包括勘误、范例程序、相关链接）：

原文书

http://www.oreilly.com/catalog/make3/index.html

中文书

http://www.oreilly.com.cn/book.php?bn=7-5641-0352-3

最后，您可以在 WWW 上找到我们：

http://www.oreilly.com
http://www.oreilly.com.cn

致谢

我要感谢 Richard Stallman 所编织的梦想以及对美梦终能成真的信心。当然，没有 Paul Smith，GNU make 不会有今日的表现，谢谢你。

我也要感谢我的编辑 —— Andy Oram，对我始终如一的支持和热诚。

应该感谢 Cimarron Software 为我提供一个环境，让我得以开始此计划。还应该感谢 Realm Systems 为我提供一个环境，让我得以完成此计划。尤其要感谢 Doug Adamson、Cathy Anderson 和 Peter Bookman 等人。

谢谢本书的评阅者们，Simon Gerraty、John Macdonald 和 Paul Smith，提供了许多见解深刻的意见，并且修正了许多令我难为情的错误。

应该感谢的人还有为本书付出贡献的：Steve Bayer、Richard Bogart、Beth Cobb、Julie Daily、David Johnson、Andrew Morton、Richard Pimentel、Brian Stevens 以及 Linus Torvalds。

还要感谢为我在暴风之海提供安全避风港的如下集体：Christine Delaney，Tony Di Sera、John Major 和 Daniel Reading。

最后，感谢我的妻子 Maggie Kasten 以及我的两个孩子 William 与 James，在最后这 16 个月期间对我的支持、鼓励以及爱。

第一部分

基本概念

在第一个部分里，我们会将重点放在make的功能上：它们能够做什么以及如何正确地使用它们。我们首先会做个简介，并且告诉你如何创建你的第一个*makefile*。这部分的内容涵盖了make的规则、变量、函数以及脚本。

看完第一部分之后，你将会获得相当完整的GNU make操作知识，并且掌握许多高级的用法。

第一章

如何编写一个简单的 makefile

程序设计的技术通常遵循着一个极为简单的惯例：编辑源代码文件、将源代码编译成可执行的形式以及对成果进行调试。尽管将源代码转换成可执行文件被视为惯例，但如果程序员的做法有误也可能会浪费大量的时间在问题的调试上。大多数开发者都经历过这样的挫败：修改好一个函数之后，运行新的程序代码时却发现缺陷并未被修正，接着发现再也无法执行这个经过修改过的函数，因为其中某个程序有误，像无法重新编译源代码、重新链接可执行文件或是重新建造 jar 文件。此外，当程序变得越来越复杂时，这些例行工作可能会因为需要（针对其他平台或程序库的其他版本等）为程序开发不同的版本而变得越来越容易发生错误。

make 程序可让“将源代码转换成可执行文件”之类的例行工作自动化。相较于脚本，make 的优点是：你可以把程序中各元素之间的关系告诉 make，然后 make 会根据这些关系和时间戳（timestamp）（译注 1）判断应该重新进行哪些步骤，以产生你所需要的程序。有了这个信息，make 还可以优化编译过程，跳过非必要的步骤。

GNU make（以及 make 的其他变体）可以准确完成此事。make 定义了一种语言，可用来描述源文件、中间文件以及可执行文件之间的关系。它还提供了一些功能，可用来管理各种候选配置、实现可重用程序库的细节以及让用户以自定义宏将过程参数化。简言之，make 常被视为开发过程的核心，因为它为应用程序的组件以及这些组件的搭配方式提供了一个可依循的准则。

make 一般会将工作细节存放在一个名为 *makefile* 的文件中。下面是一个可用来编译传统“Hello, World”程序的 *makefile*：

译注 1：Unix 文件具有三种时间属性：atime（最近被读取的时间）、ctime（模式被改变的时间）以及 mtime（最近被写入的时间）。文件的时间戳（timestamp）就是指 mtime。

```
hello: hello.c
        gcc hello.c -o hello
```

要编译此程序，你可以在命令行提示符之后键入：

```
$ make
```

以便执行make。这将会使得make程序读入*makefile*文件，并且编译它在该处所找到的第一个工作目标：

```
$ make
gcc hello.c -o hello
```

如果将某个工作目标（target）指定成命令行参数（command-line argument），make就会特别针对该工作目标进行更新的动作；如果命令行上未指定任何工作目标，make就会采用*makefile*文件中的第一个工作目标，称为默认目标（default goal）。

在大多数*makefile*文件中，默认的目标一般就是编译程序，这通常涉及许多步骤。程序的源代码经常是不完整的，而且必须使用flex或bison之类的工具来产生源代码。接着，源代码必须被编译成二进制目标文件（binary object file）——*.o*文件用于C/C++、*.class*文件用于Java等。然后，对C/C++而言，链接器（通常调用自gcc编译器）会将这些目标文件链接在一起形成一个可执行文件。

修改源文件中的任何内容并重新调用make，将会使得这些步骤中的某些（通常不是全部）被重复进行，因此源代码中的变更会被适当地并入可执行文件。这个规范说明书文件（specification file）或*makefile*文件中，描述了源代码文件、中间文件以及可执行文件之间的关系，使得make能够以最少的工作量来完成更新可执行文件的工作。

所以，make的主要价值在于它有能力完成编译应用程序时所需要的一系列复杂步骤，以及当有可能缩短"编辑－编译－调试"（edit-compile-debug）周期时对这些步骤进行优化的动作。此外，make极具灵活性，你可以在任何具有文件依存关系的地方使用它，范围从C/C++到Java、TEX、数据库管理等。

工作目标与必要条件

基本上，*makefile*文件中包含了一组用来编译应用程序的规则。make所看到的第一项规则，会被作为默认规则（default rule）使用。一项规则可分成三个部分：工作目标（target）、它的必要条件（prerequisite）以及所要执行的命令（command）：

```
target: prereq1 prereq2
    commands
```

工作目标（target）是一个必须建造的文件或进行的事情，必要条件（prerequisite）或依存对象（dependent）是工作目标得以被成功创建之前，必须事先存在的那些文件；而所要执行的命令（command）则是必要条件成立时将会创建工作目标的那些shell命令。

下面这项规则是用来将一个C文件*foo.c*编译成一个目标文件*foo.o*：

```
foo.o: foo.c foo.h
        gcc -c foo.c
```

工作目标*foo.o*出现在冒号之前；必要条件*foo.c*和*foo.h*出现在冒号之后，命令脚本（command script）通常出现在后续的文本行上，而且会放在跳格符（tab character）之后。

当make被要求处理某项规则时，它首先会找出必要条件和工作目标中所指定的文件。如果必要条件中存在关联到其他规则的文件，则make会先完成相应规则的更新动作，然后才会考虑到工作目标。如果必要条件中存在时间戳在工作目标的时间戳之后的文件（译注2），则make会执行命令以便重新建立工作目标。脚本会被传递给shell并在其subshell中运行。如果其中的任何命令发生错误，则make会终止工作目标的建立动作并结束执行。

下面这个程序会在它的输入中计算fee、fie、foe和fum等词汇（单词）出现的次数。这个程序（文件名为*count_words.c*）并不难，因为它使用了一个flex扫描程序：

```
#include <stdio.h>

extern int fee_count, fie_count, foe_count, fum_count;
extern int yylex( void );

int main( int argc, char ** argv )
{
    yylex();
    printf( "%d %d %d %d\n", fee_count, fie_count, foe_count, fum_count );
    exit( 0 );
}
```

这个扫描程序（文件名为*lexer.l*）相当简单：

```
        int fee_count = 0;
        int fie_count = 0;
        int foe_count = 0;
        int fum_count = 0;
%%
fee     fee_count++;
fie      fie_count++;
```

译注2：也就是说，必要条件中最近有文件遭到修改，但是工作目标尚未进行更新的动作。

```
foe     foe_count++;
fum     fum_count++;
```

用来编译这个程序的*makefile*也很简单：

```
count_words: count_words.o lexer.o -lfl
        gcc count_words.o lexer.o -lfl -o count_words
count_words.o: count_words.c
        gcc -c count_words.c
lexer.o: lexer.c
        gcc -c lexer.c
lexer.c: lexer.l
        flex -t lexer.l > lexer.c
```

当这个*makefile*首次被执行时，我们会看到：

```
$ make
gcc -c count_words.c
flex -t lexer.l > lexer.c
gcc -c lexer.c
gcc count_words.o lexer.o -lfl -o count_words
```

现在我们已经编译好了一个可执行的程序。当然，此处所举的例子有点简化，因为实际的程序通常是由多个模块所构成。此外，看过后面的章节你就会知道，这个*makefile*并未用到make大部分的特性，所以显得有点冗长。不过，它仍不失为一个实用的*makefile*。举例来说，在这个范例的编写期间，为了测试程序，我执行了这个*makefile*不少于10次。

当这个*makefile*范例在执行时，你可能会发现make执行命令的顺序几乎和它们出现在*makefile*中的顺序相反。这种"从上而下"（top-down）的风格是*makefile*文件中常见的手法。一般来说，通用形式的工作目标会先在*makefile*文件中被指定，而细节则会跟在后面。make程序对此风格的支持有许多方式，其中以make的两阶段执行模型（two-phase execution model）以及递归变量（recursive variable）最为重要。我们将会在稍后的章节中深入探讨相关细节。

检查依存关系

make如何知道自己该做什么事呢？让我们继续探讨前一个范例。

make首先注意到命令行上并未指定任何工作目标，所以会想要建立默认目标*count_words*。当make检查其必要条件时看到了三个项目：*count_words.o*、*lexer.o*以及-lfl。现在make会想要编译*count_words.o*并看到相应的规则。接着，make会再次检查必要条件并注意到*count_words.c*并未关联到任何规则，但存在*count_words.c*这个文件，所以会执行相应的命令把*count_words.c*转换成*count_words.o*：

```
gcc -c count_words.c
```

这种“从工作目标到必要条件，从必要条件到工作目标，再从工作目标到必要条件”的链接（chaining）机制就是 make 分析 *makefile* 决定要执行哪些命令的典型做法。

必要条件的下一个项目会让 make 想要编译 *lexer.o*。规则链会将 make 导向 *lexer.c*，但这次 *lexer.c* 并不存在。make 会从 *lexer.l* 找到产生 *lexer.c* 的规则，所以 make 会运行 flex 程序。现在 *lexer.c* 存在了，make 会接着执行 gcc 命令。

最后，make 看到 -lfl，其中 -l 是选项，用来要求 gcc 必须将其所指定的系统程序库链接进应用程序。此处指定了 fl 这个参数，代表实际的程序库名称为 *libfl.a*。GNU make 对这个语法提供了特别的支持：当 -l<NAME> 形式的必要条件被发现时，make 会搜索 *libNAME.so* 形式的文件；如果找不到相符的文件，make 接着会搜索 *libNAME.a* 形式的文件。在此例中，make 会找到 */usr/lib/libfl.a*，而且会进行最后的动作 —— 链接。

尽量减少重新编译的工作量

运行这个程序时，我们发现它除了会输出 fee、fie、foe 和 fum 等单词出现的次数，还会输出来自输入文件的其他文本。这并非我们想要的结果。问题出在我们忽略了词汇分析器（lexical analyzer）的一些规则，而且 flex 会将未被认出的文本送往输出。我们只要加入一条“any character”（任何字符）规则以及一条 newline（换行字符）规则就可以解决这个问题：

```
            int fee_count = 0;
            int fie_count = 0;
            int foe_count = 0;
            int fum_count = 0;
%%
fee         fee_count++;
fie          fie_count++;
foe         foe_count++;
fum         fum_count++;
.
\n
```

编辑这个文件之后，还需要重新编译应用程序以便测试我们所做的修正：

```
$ make
flex -t lexer.l > lexer.c
gcc -c lexer.c
gcc count_words.o lexer.o -lfl -ocount_words
```

请注意，这次 *count_words.c* 文件并未被重新编译。分析规则的时候，make 发现 *count_words.o* 已存在，而且该文件的时间戳在其必要条件 *count_words.c* 的之后，所以

不需要采取任何更新的动作。不过，分析*lexer.c*的时候，make发现必要条件*lexer.l*的时间戳在其工作目标*lexer.c*的之后，所以make必须更新*lexer.c*。这会依次引起*lexer.o*、*count_words*的更新。运行这个重新编译的程序，你会看到如下的结果：

```
$ count_words < lexer.l
3 3 3 3
```

调用 make

前面的范例做了以下假设：

- 项目的所有程序代码以及make描述文件全都被放在单一目录中。
- make描述文件的文件名为*makefile*、*Makefile*或*GNUMakefile*。
- 执行make命令时，*makefile*就放在用户的当前目录（current directory）中。

当make在上述情况下被调用时，make会自动编译其所找到的第一个工作目标。要更新另一个不同的工作目标（或多个工作目标），请在命令行上指定工作目标的名称：

```
$ make lexer.c
```

当make被执行时，它会读取描述文件以及找出所要更新的工作目标。如果工作目标或其必要条件中的任一文件尚未更新（或不存在），则会（以一次一个命令的方式）执行相应规则的命令脚本中的shell命令。这些命令被执行之后，make会假设工作目标已完成更新的动作，于是移往下一个工作目标或是结束执行。

如果你所指定的工作目标已经更新（up to date），则make除了告诉你此状况并立即结束运行外，其他什么事也不做：

```
$ make lexer.c
make: `lexer.c' is up to date.
```

如果你所指定的工作目标并未出现在*makefile*文件中，也不存在与之相应的隐含规则（implicit rule），则make将会作出如下的响应：

```
$ make non-existent-target
make: *** No rule to make target `non-existent-target'.  Stop.
```

make提供了许多命令行选项。其中最有用的选项之一是--just-print（或-n），用来要求make显示它将为特定工作目标执行的命令，但不要实际执行它们。当你编写*makefile*时，这个功能特别有用。你甚至还可以在命令行上设定几乎所有的*makefile*变量，来改写默认值或*makefile*文件中所设定的值。

Makefile 的基本语法

对make有了基本的认识之后，现在你差不多可以编写自己的*makefile*了。这一节我们将会介绍*makefile*的基本语法和结构，让你得以开始使用make。

*makefile*文件中一般采用“从上而下”(top-down)的结构，所以默认会更新最上层的工作目标（通常叫做all）。下层工作目标用来让上层工作目标保持在最新的状态，例如，用来删除无用的临时文件的clean工作目标应该放在最下层。正如你所猜测的，工作目标的名称并不一定非得是真实的文件名称不可，你可以使用任何名称。

在前面的范例中我们所看到的是经过简化的规则。下面是较完整（但可能仍然不够完整）的规则：

```
target1 target2 target3 : prerequisite1 prerequisite2
    command1
    command2
    command3
```

冒号的左边可以出现一个或多个工作目标，而冒号的右边可以出现零个或多个必要条件。如果冒号的右边没有指定必要条件，那么只有在工作目标所代表的文件不存在时才会进行更新的动作。更新工作目标所要执行的那组命令有时会被称为命令脚本（command script），不过通常只被称为命令（command）。

每个命令必须以跳格符（tab character）开头，这个（隐含的）语法用来要求make将紧跟在跳格符之后的内容传给subshell来执行。如果你不经意地在非命令行（noncommand line）的第一个字符前插入了一个跳格符，则在大多数情况下，make将会把其后的文字作为命令来解释。如果你很幸运，这个误入歧途的跳格符将被视为语法错误，那么你会因此收到如下的信息：

```
$ make
Makefile:6: *** commands commence before first target.  Stop.
```

我们将会在第二章“规则”中讨论错综复杂的跳格符。

make会将井号（#）视为注释字符（comment character），从井号开始到该行结束之间的所有文字都会被make忽略。你可以对作为注释的文本行进行缩排或前置空格。注释字符#并不会在代表命令的文本行中引入make的注释功能，这一整行（包括#及其后的字符）会被传给shell来执行。这行文字的处理方式取决于你所使用的shell。

你可以使用标准的Unix转义字符（escape character）——反斜线（\），来延续过长的文本行。反斜线一般用来延续过长的命令，也可用来延续必要条件。稍后我们将会探讨处理过长必要条件的其他方法。

你现在已经有能力编写简单的*makefile*了。接下来我们将会在第二章探讨规则的细节，在第三章探讨make变量以及在第五章探讨命令。所以现在你应该避免使用变量、宏以及多行命令。

第二章

规则

前一章中，我们编写了若干规则，用来编译与链接我们的单词计数（word counting）程序。我们为每个规则定义了一个工作目标，也就是一个需要更新的文件。每个工作目标依存于一组必要条件，这组必要条件也都是文件。当你要求更新某个工作目标时，如果必要条件中存在时间戳在工作目标的时间戳之后的文件，make就会执行相应规则里的命令脚本。因为某个规则的工作目标可以是另一个规则的必要条件，所以这样的工作目标和必要条件将会形成依存图（dependency graph）。建立及处理依存图，并据此更新特定的工作目标，就是make所要做的事情。

规则对make而言十分重要，make允许你使用各种类型的规则。具体规则（explicit rule），就像我们在上一章所编写的规则，用来指定需要更新的工作目标：如果必要条件中存在时间戳在此工作目标的时间戳之后的文件，make就会对它进行更新的动作。这将会是你最常使用的规则类型。模式规则（pattern rule）中所使用的是通配符（wildcard）而不是明确的文件名称，这让make得以对与模式相符的工作目标应用该规则，进行必要的更新动作。隐含规则（implicit rule）可以是模式规则，也可以是内置于make的后缀规则（suffix rule）。有了这些内置于make的规则可以让*makefile*的编写变得更为容易，因为对于工作目标的更新，make已经知道许多常见文件类型、后缀以及更新工作目标的程序。至于静态模式规则（static pattern rule），它就像正规模式规则一样，只不过它们只能应用在一串特定的工作目标文件中。

GNU make可作为许多其他版本的make的替代品，它特别针对兼容性提供了若干功能。后缀规则最初是make用来编写通则（general rule）的方法。尽管GNU make也支持后缀规则，不过为了更完整及更一般化，它考虑以模式规则来替换。

具体规则

你编写的规则多半会是具体规则，以特定的文件作为工作目标和必要条件。每个规则都可以有多个工作目标。这意味着，每个工作目标所具备的必要条件可以跟其他工作目标的一样。如果这些工作目标尚未被更新，则make将会为它们执行同一组更新动作。例如：

```
vpath.o variable.o: make.h config.h getopt.h gettext.h dep.h
```

这代表 *vpath.o* 和 *variable.o* 与同一组 C 头文件具有依存关系。这一行等效于：

```
vpath.o: make.h config.h getopt.h gettext.h dep.h
variable.o: make.h config.h getopt.h gettext.h dep.h
```

这两个工作目标将会被分开处理。只要有任何一个目标文件尚未被更新（也就是说，任何一个头文件的时间戳在该目标文件的之后），则make将会执行规则中所指定的命令以便更新该目标文件。

你不必将规则一次定义完全（all at once）。每当 make 看到一个工作目标，就会将该工作目标与其必要条件加入依存图。如果make所看到的工作目标已经存在于依存图中，则任何额外的必要条件都会被附加到该工作目标在依存图中的项目里。对较简单的应用来说，这个特性可用来断开太长的规则以增进 *makefile* 的可读性：

```
vpath.o: vpath.c make.h config.h getopt.h gettext.h dep.h
vpath.o: filedef.h hash.h job.h commands.h variable.h vpath.h
```

对较复杂的应用来说，必要条件可以组成自看似无关的文件：

```
# 确定 vpath.c 被编译之前 lexer.c 已经创建好了
vpath.o: lexer.c
...
# 以特殊的标记来编译 vpath.c
vpath.o: vpath.c
        $(COMPILE.c) $(RULE_FLAGS) $(OUTPUT_OPTION) $<
...
# 引入另一个程序所产生的依存关系
include auto-generated-dependencies.d
```

第一个规则指出，每当 *lexer.c* 被更新后，*vpath.o* 就必须被更新（这或许是因为产生 *lexer.c* 的过程中会有其他副作用）。这个规则还可用来确保必要条件的更新动作总是在工作目标之前被实施（注意规则的双向作用。就其正向作用而言，此规则指出，若 *lexer.c* 已经被更新，则需要对 *vpath.o* 执行更新的动作；就其反向作用而言，此规则指出，如果我们需要建立或使用 *vpath.o*，首先必须确定 *lexer.c* 已经更新）。这个规则应该就近放在 *lexer.c* 处理规则的旁边，好让开发人员能够注意到这个微妙的关系。稍后，*vpath.o* 的编译规则会被放到其他编译规则之中。此规则的命令用到了三个make变量。你将会看

到更多的make变量，不过现在你只需要知道，一个变量可以是一个美元符号后面跟着单一字符（character），也可以是一个美元符号后面跟着一个加圆括号的单词（稍后，我将会在本章做进一步的说明，并且会在第三章做更多的说明）。最后，*.o* 文件和 *.h* 文件的依存关系是从另一个文件（这个文件产生自外部程序）引入到 *makefile* 的。

通配符

当你有一长串文件要指定时，为了简化此过程，make提供了通配符（wildcard），此功能也被称为文件名模式匹配（globbing）。make的通配符如同 Bourne shell 的 ~、*、?、[...] 和 [^...]。举例来说，*.* 会被扩展（expand）成文件名中包含点号的所有文件。一个问号代表任何单一字符，而[...]代表一个字符集（character class）。若要取得字符集的"补集"则可以使用[^...]。

此外，"~"符号（译注1）可以用来代表当前用户的主目录（home directory）。一个"~"符号之后若跟着用户的名称则代表该用户的主目录。

每当make在工作目标、必要条件或命令脚本等语境（context）中看到通配符，就会自动扩展通配符。在其他语境中，你可以通过函数的调用手动扩展通配符。如果你想创建适应能力较强的 *makefile*，通配符非常有用。举例来说，如果不想手动列出一个程序里的所有文件，你就可以使用通配符（注1）：

```
prog: *.c
        $(CC) -o $@ $^
```

不过通配符的使用务必谨慎为之，因为一不小心就会有误用的危险。比如：

```
*.o: constants.h
```

这个规则的意图很明显：所有的目标文件皆依存于头文件 *constants.h*。不过，如果工作目录中当前并未包含任何目标文件，则通配符扩展后会变成下面这样：

```
: constants.h
```

译注1："~"符号又称为tilde。若"~"符号出现在西班牙文的字母上，代表它的发音是上颚的鼻音，此时tilde可以译成"颚化符号"。若"~"符号出现在C语言中，代表按位的逻辑运算符（bitwise logical operator），其功能为"取1的补码"。也有人将"~"符号译成"波浪符号"，也就是将tilde作为tidal（潮汐）来看，事实上它的形状还挺像的。

注1：在顾虑比较多的环境中，使用通配符来选取程序里的文件被视为一种坏习惯，因为这可能会将毫无关系的源文件意外地链接进该程序。

这是一个合法的make表达式，而且它本身并不会产生错误信息。实现此规则的正确方法，就是针对源文件使用通配符（因为它们总是存在的）以及将之转换成一串目标文件。当我们在第四章讨论到make的函数时将会提到这个技巧。

最后，值得注意的是，当模式出现在工作目标或必要条件中时，是由make进行通配符的扩展。然而，当模式出现在命令中时，是由subshell进行扩展的动作。区分这两种情况有时会变得很重要，因为make会在读取*makefile*的时候立即扩展通配符，但是shell只会在执行命令的时候扩展通配符。当有许多复杂的文件操作需要进行时，这两种文件扩展动作将会有很大的差别。

假想工作目标

到目前为止，我们所提到的工作目标以及必要条件都会进行文件的创建和更新的动作。尽管这是典型的用法，但是以工作目标充当标签（label）来代表命令脚本，通常会有些用处。举例来说，稍早我们曾提到在许多*makefile*中，默认的首先要处理的标准工作目标称为all。任何不代表文件的工作目标就叫作假想工作目标（phony target）。另一个标准的假想工作目标称为clean：

```
clean:
        rm -f *.o lexer.c
```

通常，make总是会执行假想工作目标，因为对应于该规则的命令并不会创建以该工作目标为名称的文件。

切记，make无法区分文件形式的工作目标与假想工作目标。如果当前目录中刚好出现与假想工作目标同名的文件，make将会在它的相依图中建立该文件与假想工作目标的关系。举例来说，如果你运行make clean时，工作目录中刚好存在*clean*这个文件，那么将会产生令人困惑的信息：

```
$ make clean
make: `clean' is up to date.
```

因为大多数的假想工作目标并未指定必要条件，clean工作目标总是会被视为已经更新，所以相应的命令永远不会被执行。

为了避免这个问题，GNU make提供了一个特殊的工作目标——.PHONY，用来告诉make，该工作目标不是一个真正的文件。当你要声明假想工作目标时，只要将该工作目标指定成.PHONY的一个必要条件即可：

```
.PHONY: clean
clean:
        rm -f *.o lexer.c
```

现在，即使当前目录中存在名为*clean*的文件，make还是会执行对应于clean的命令。除了总是将工作目标标记为尚未更新、将一个工作目标声明为"假"之外，还会让make知道，不应该像处理一般规则那样，从源文件来建立以工作目标为名的文件。因此，make可以优化它的一般规则搜索程序以提高性能。

以"假想工作目标"作为"实际文件"的一个"必要条件"似乎不太有意义，因为假想工作目标总是尚未更新，这总会使得该实际文件（工作目标）被重新建立。然而，以"假想工作目标"作为"假想工作目标"的必要条件通常会有些用处。举例来说，all工作目标常会被用来指定所要编译的一串程序：

```
.PHONY: all
all: bash bashbug
```

其中，all工作目标将会创建bash（一个shell程序）以及bashbug（一个错误报告工具）。

你还可以将假想工作目标作为内置在*makefile*里的shell脚本来用。以假想工作目标作为另一个工作目标的必要条件，可让make在进行实际工作目标之前调用假想工作目标所代表的脚本。假如我们很在意磁盘空间的使用情况，因而在进行磁盘密集的工作之前，我们会想要显示磁盘尚有多少空间可供使用，我们可能会这么做：

```
.PHONY: make-documentation
make-documentation:
 df -k . | awk 'NR == 2 { printf( "%d available\n", $$4 ) }'
 javadoc ...
```

这么做的问题是，我们最后可能会在不同的工作目标下多次指定df和awk命令，这会造成一个维护上的问题，因为如果我们在另一个系统上遇到了输出格式不同的df命令，那么我们必须到指定df和awk命令的每一处进行修改。此时，我们可以把df那一行放在它自己的假想工作目标里：

```
.PHONY: make-documentation
make-documentation: df
        javadoc ...
.PHONY: df
df:
        df -k . | awk 'NR == 2 { printf( "%d available\n", $$4 ) }'
```

以df作为make-documentation的一个必要条件，可让make在产生文件之前先调用我们的df工作目标。可以这么做是因为make-documentation也是一个假想工作目标。现在，即使我们在其他工作目标中重复使用df，也不会造成什么维护上的问题。

假想工作目标还有许多其他的好用处。

make的输出常会把想要进行阅读以及调试的人搞糊涂。这是因为：尽管*makefile*的编写

是采用从上而下（top-down）的形式，不过make执行命令的方式却是采用从下而上（bottom-up）的形式；此外，你根本无法判断当前正在执行哪个规则。如果能够在make的输出中为主要工作目标加上注释，那么make的输出就会变得很容易阅读。这就是假想工作目标可以派上用场的地方。如下所示的例子摘录自bash的*makefile*：

```
$(Program): build_msg $(OBJECTS) $(BUILTINS_DEP) $(LIBDEP)
        $(RM) $@
        $(CC) $(LDFLAGS) -o $(Program) $(OBJECTS) $(LIBS)
        ls -l $(Program)
        size $(Program)

.PHONY: build_msg
build_msg:
        @printf "#\n# Building $(Program)\n#\n"
```

因为printf位于假想工作目标之中，所以在任何必要条件被更新之前会立即输出信息。如果以build_msg作为$(Program)命令脚本的第一个命令，那么在所有编译结果和依存关系都产生之后才会执行该命令。切记，因为假想工作目标总是尚未更新，所以假想工作目标build_msg会导致$(Program)被重建——即使它已经被更新。这么做似乎是明智的抉择，所有的计算工作在编译目标文件的时候大多已经完成，因此只有最后的链接工作一定会被执行。

假想工作目标还可用来改善*makefile*的“用户接口”。工作目标通常是包含目录路径元素、额外文件名成分（比如版本编号）以及标准扩展名的复合字符串，这使得“在命令行上指定工作目标的文件名”成为一种挑战。你只要加入一个简单的假想工作目标，并以实际文件作为它的必要条件，就可以避免这个问题。

许多*makefile*多少都会包含一组标准的假想工作目标。表2-1列出了这些标准的假想工作目标。

表2-1：标准的假想工作目标

工作目标	功能
all	执行编译应用程序的所有工作
install	从已编译的二进制文件进行应用程序的安装
clean	将产生自源代码的二进制文件删除
distclean	删除编译过程中所产生的任何文件（除了二进制文件，也包含configure所产生的Makefile）
TAGS	建立可供编辑器使用的标记表
info	从Texinfo源代码来创建GNU info文件
check	执行与应用程序相关的任何测试

工作目标TAGS实际上不是一个假想工作目标，因为ctags和etags程序的输出就是名为*TAGS*的文件。此处之所以提到它，是因为就我们所知，它是绝无仅有的、标准的非假想工作目标（nonphony target）。

空工作目标

空工作目标（empty target）如同假想工作目标一样，可用来发挥make的潜在能力。假想工作目标总是尚未更新，所以它们总是会被执行，并且总是会使得它们的“依存对象”（工作目标所关联到的必要条件）被重建。但假设我们有若干命令，它们不会输出任何文件，只是偶尔需要被执行一下，而且我们并不想让我们的依存对象（dependent）被更新，该这么办？此时，我们可以建立一个规则，它的工作目标是一个空文件（有时称为cookie）：

```
prog: size prog.o
        $(CC) $(LDFLAGS) -o $@ $^

size: prog.o
        size $^
        touch size
```

请注意，size规则在执行完之后，会使用touch创建一个名为*size*的空文件。这个空文件可作为它的时间戳，因此make只在*prog.o*被更新之后才会执行size规则。此外，*prog*的必要条件size将不会导致*prog*的更新，除非它的目标文件的时间戳也在工作目标（的时间戳）之后。

与自动变量$?并用时，空文件特别有用。我们将会在“自动变量”一节探讨自动变量，不过事先了解一下这个变量应该不会有什么问题。对每个规则的命令脚本部分来说，make会将变量$?替换成一组必要条件，这组必要条件的时间戳在工作目标的时间戳之后。例如，下面这个规则将会输出自从上次执行make print之后，变更过的所有文件：

```
print: *.[hc]
        lpr $?
        touch $@
```

通常，空文件可用来标明最近发生了一个特殊的事件。

变量

现在让我们来查看曾在范例中出现的若干变量。其中最简单的变量具有如下的语法：

```
$(variable-name)
```

这代表我们想要扩展（expand）名为*variable-name*的变量。任何文字都可以包含在变量之中，而且大多数字符（包括标点符号）都可以用在变量名称上。例如，内含C编译命令的变量可以取名为COMPILE.c。一般来说，你必须以$()或$()将变量名称括住，这样make才会认得。有一个例外：变量名称若是单一字符则不需要为它加上圆括号。

通常*makefile*文件中都会定义许多变量，不过其中有许多特殊变量是make自动定义的。这些变量中的若干变量可供用户用来控制make的行为，其余变量则是供make用来跟用户的*makefile*文件沟通。

自动变量

当规则相符时，make会设定自动变量（automatic variable）。通过它们，你可以取用工作目标以及必要条件中的元素，所以你不必指明任何文件名称。要避免重复，自动变量就相当有用，它们也是定义较一般的模式规则时不可少的项目（稍后会讨论到）。

下面是7个“核心”的自动变量：

$@ 工作目标的文件名。

$% 档案文件成员（archive member）结构中的文件名元素。（译注2）

$< 第一个必要条件的文件名。

$? 时间戳在工作目标(的时间戳)之后的所有必要条件，并以空格隔开这些必要条件。

$^ 所有必要条件的文件名，并以空格隔开这些文件名。这份列表已删掉重复的文件名，因为对大多数的应用而言，比如编译、复制等，并不会用到重复的文件名。

$+ 如同$^，代表所有必要条件的文件名，并以空格隔开这些文件名。不过，$+包含重复的文件名。此变量会在特殊的状况下被创建，比如将自变量传递给链接器（linker）时重复的值是有意义的。

$* 工作目标的主文件名。一个文件名称是由两个部分组成：主文件名（stem）和扩展名（suffix）。稍后我们将会在“模式规则”一节中探讨主文件名的处理方式。请不要在模式规则以外使用此变量。

译注2：档案文件（archive file）中个别的成员可作为工作目标或必要条件。你可以通过archive(member)这样的语法在档案文件archive中指定名为member的成员。举例来说，若工作目标是foo.a(bar.o)，则$%是bar.o而$@是foo.a。当工作目标不是一个档案文件成员时，$%就是空的。

此外，为了跟其他版本的make兼容，以上这六个变量都具有两个变体。其中一个变体只会返回值的目录部分，它的指定方式就是在原有的符号之后附加D这个字母，例如$(@D)、$(<D)等，另一个变体只会返回值的文件部分，它的指定方式就是在原有的符号之后附加F这个字母，比如$(@F)、$(<F)等。请注意，这些变体名称的字符长度超过一个，所以必须加上圆括号。GNU make还以dir和notdir函数提供较具可读性的替代方案。我们将会在第四章进行函数的探讨。

make会在规则与它的工作目标和必要条件相符之后设定自动变量，所以变量只能应用在规则中的命令脚本部分。

现在，我们可以将之前明确指定文件名的*makefile*替换成适当的自动变量。

```
count_words: count_words.o counter.o lexer.o -lfl
        gcc $^ -o $@

count_words.o: count_words.c
        gcc -c $<

counter.o: counter.c
        gcc -c $<

lexer.o: lexer.c
        gcc -c $<

lexer.c: lexer.l
        flex -t $< > $@
```

以VPATH和vpath来查找文件

到目前为止我们所举的例子都相当简单：*makefile*与所有的源文件都存放在同一个目录下。真实世界的程序比较复杂（请问，你上一次开发只有一个目录的项目是在什么时候？）。现在让我们重构（refactor）先前的范例，进行较实际的文件布局。我们可以通过将main重构成一个名为counter的函数来修改我们的单词计数程序。

```
#include <lexer.h>
#include <counter.h>
void counter( int counts[4] )
{
    while ( yylex() )
        ;

    counts[0] = fee_count;
    counts[1] = fie_count;
    counts[2] = foe_count;
    counts[3] = fum_count;
}
```

一个可重复使用的程序库函数（library function），在头文件（header file）中应该要有一个声明（declaration），所以让我们创建 *counter.h* 头文件来包含此声明：

```
#ifndef COUNTER_H_
#define COUNTER_H_

extern void
counter( int counts[4] );

#endif
```

我们还可以把 *lexer.l* 的声明放在 *lexer.h* 头文件中：

```
#ifndef LEXER_H_
#define LEXER_H_

extern int fee_count, fie_count, foe_count, fum_count;
extern int yylex( void );

#endif
```

按源代码树（source tree）的布局惯例，头文件会被放在 *include* 目录中，而源文件会被放在*src*目录里。我们也会这样做，并把*makefile*放在它们的上层目录（parent directory）。现在范例程序的布局如图 2-1 所示。

图 2-1：范例源代码树的布局

既然现在我们的源文件中包含了头文件，这些新产生的依存关系就应该记录在我们的 *makefile* 文件中，这样，当我们的头文件有所变动时，才会更新相应的目标文件。

```
count_words: count_words.o counter.o lexer.o -lfl
        gcc $^ -o $@
```

```
count_words.o: count_words.c include/counter.h
        gcc -c $<

counter.o: counter.c include/counter.h include/lexer.h
        gcc -c $<

lexer.o: lexer.c include/lexer.h
        gcc -c $<

lexer.c: lexer.l
        flex -t $< > $@
```

现在运行make将会看到如下的结果：

```
$ make
make: *** No rule to make target `count_words.c', needed by `count_words.o'.  Stop.
```

咦，发生了什么事？*makefile*想要更新*count_words.c*，不过那是一个源文件！让我们来"扮演make的角色"。我们的第一个必要条件是*count_words.o*。我们并未看到这个文件，所以我们会去查找一个规则以便创建此文件。用来创建*count_words.o*的具体规则(explicit rule)指向*count_words.c*，但为何make找不到这个源文件？因为这个源文件并非位于当前目录中，而是被放在*src*目录里。除非你告诉make，否则它只会在当前目录中找寻工作目标以及必要条件。我们要怎么做才有能让make到*src*目录找寻到源文件？也就是说，要如何告诉make我们的源代码放在哪里？

你可以使用VPATH和vpath来告诉make到不同的目录去查找源文件。要解决我们眼前的问题，可以在*makefile*文件中对VPATH进行如下的赋值动作：

```
VPATH = src
```

这表示，如果make所需要的文件并未放在当前目录中，就应该到*src*目录去找。为了让make的输出更为明确，*makefile*本身也做了调整（黑体字部分），此时*makefile*会像下面这样：

```
VPATH = src
count_words: count_words.o counter.o lexer.o -lfl
        gcc $^ -o $@
count_words.o: count_words.c include/counter.h
        gcc -c $< -o $@
counter.o: counter.c include/counter.h include/lexer.h
        gcc -c $< -o $@
lexer.o: lexer.c include/lexer.h
        gcc -c $< -o $@
lexer.c: lexer.l
        flex -t $< > $@
```

现在运行make将会看到如下的结果：

```
$ make
gcc -c src/count_words.c -o count_words.o
src/count_words.c:2:21: counter.h: No such file or directory
make: *** [count_words.o] Error 1
```

请注意，现在make可以编译第一个文件了，因为它会为该文件正确填入相对路径。使用自动变量的另一个理由是：如果你写出具体的文件名，make将无法为该文件填上正确的路径。可惜并未编译成功，因为gcc无法找到引入文件（include file）。我们只要使用正确的-I选项来“自定义”（customizing）隐含编译规则（implicit compilation rule）就可以解决这个问题了：

```
CPPFLAGS = -I include
```

请注意，由于头文件被放在include目录中，所以还必须调整VPATH（黑体字部分）：

```
VPATH = src include
```

现在我们可以顺利完成编译的工作了：

```
$ make
gcc -I include -c src/count_words.c -o count_words.o
gcc -I include -c src/counter.c -o counter.o
flex -t src/lexer.l > lexer.c
gcc -I include -c lexer.c -o lexer.o
gcc count_words.o counter.o lexer.o /lib/libfl.a -o count_words
```

VPATH变量的内容是一份目录列表，可供make搜索其所需要的文件。这份目录列表可用来搜索工作目标以及必要条件，但不包括脚本中所提及的文件。这份目录列表的分隔符在Unix上可以是空格或冒号，在Windows上可以是空格或分号。我喜欢使用空格，因为它可以在任何系统上都通行无阻，这样我们就可以避开冒号与分号的纠结；此外，以空格为分隔符将会使得目录较容易阅读。

虽然VPATH变量可以解决以上的搜索问题，但是也有限制。make将会为它所需要的任何文件搜索VPATH列表中的每个目录。如果在多个目录中出现同名的文件，则make只会攫取第一个被找到的文件。有时这可能会造成问题。

此时可以使用vpath指令（directive）。这个指令的语法如下所示：

```
vpath pattern directory-list
```

所以，之前所使用的VPATH变量可以改写成：

```
vpath %.l %.c src
vpath %.h include
```

现在，我们告诉了make应该在*src*目录中搜索.c文件，我们还告诉它，应该在*include*

目录中搜索.*h*文件（所以我们可以从头文件必要条件中移除*include/*字样）。在较复杂的应用程序中，这项控制功能可省去许多头痛和调试的时间。

我们在此处使用vpath来解决“源文件散布在多个目录中”的问题。这个问题与“源文件放在源代码树（source tree）而目标文件放在二进制代码树（binary tree）时，要如何编译应用程序”的问题，虽然相关但却是不相同的。尽管适当地使用vpath也可以解决这个新问题，不过整个工作很快就会复杂到单靠vpath无法处理的地步。我们将会在稍后详细探讨这个问题。

模式规则

我们现在所看到的*makefile*范例已经有点长了。如果这是一个仅包含十几个或更少文件的小型程序，我们可能并不担心；但是如果这是一个包含成百上千个文件的大型程序，手动指定每个工作目标、必要条件以及命令脚本将会变得不切实际。此外，在我们的*makefile*中，这些命令脚本代表着重复的程序代码。如果这些命令脚本包含了一个缺陷或曾经被修改过，那么我们必须更新所有相关的规则。这将会给维护带来困难，而且会成为各种缺陷的源头。

许多程序在读取文件以及输出文件时都会依照惯例。例如，所有C编译器都会假设，文件若是以.*c*为扩展名，其所包含的就是C源代码，把扩展名从.*c*替换成.*o*（对Windows上的某些编译器来说是.*obj*）就可以得到目标文件的文件名。在前一章中，我们可以看到flex的输入文件使用了.*l*这个扩展名，它的输出使用了.*c*这个扩展名。

这些惯例让make可以通过文件名模式的匹配来简化规则的建立，以及提供内置规则来处理它们。举例来说，通过这些内置的规则，我们可以把之前的这17行（含空白行）的*makefile*缩减成7行：

```
VPATH    = src include
CPPFLAGS = -I include

count_words: counter.o lexer.o -lfl
count_words.o: counter.h
counter.o: counter.h lexer.h
lexer.o: lexer.h
```

所有内置规则（built-in rule）都是模式规则（pattern rule）的实例。一个模式规则看起来就像之前你所见过的一般规则，只是主文件名（就是扩展名之前的部分）会被表示成%字符。上面这个*makefile*之所以可行是因为make里存在三项内置规则。第一项规则描述了如何从一个.*c*文件编译出一个.*o*文件：

```
%.o: %.c
        $(COMPILE.c) $(OUTPUT_OPTION) $<
```

第二项规则描述了如何从*.l*文件产生一个*.c*文件：

```
%.c: %.l
        @$(RM) $@
        $(LEX.l) $< > $@
```

最后是一项特殊的规则，描述了如何从*.c*文件产生出一个不具扩展名（经常是一个可以执行）的文件：

```
%: %.c
        $(LINK.c) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

我们将会进一步探讨这个语法的细节，不过首先让我们查看make的输出，看看它是如何应用这些内置规则的。

当我们对这7行*makefile*运行make时，会看到如下的输出：

```
$ make
gcc  -I include  -c -o count_words.o src/count_words.c
gcc  -I include  -c -o counter.o src/counter.c
flex  -t src/lexer.l > lexer.c
gcc  -I include  -c -o lexer.o lexer.c
gcc   count_words.o counter.o lexer.o /lib/libfl.a   -o count_words
rm lexer.c
```

首先，make会读取*makefile*，并且将默认目标（default goal）设置成*count_words*，因为命令行上并未指定任何工作目标。查看默认目标时，make发现了四个必要条件：*count_words.o*（*makefile*并未指定这个必要条件，它是由隐含规则提供的）、*counter.o*、*lexer.o*以及-lfl。接着，make会试着依次更新每个必要条件。

当make检查第一个必要条件*count_words.o*时，并未发现可以处理它的具体规则（explicit rule），不过却找到了隐含规则（implicit rule）。查看当前目录，make并未找到源文件，所以它开始搜索VPATH，而且在*src*目录中找到了一个相符的源文件。因为*src/count_words.c*没有其他必要条件，make可以自由更新*count_words.o*，所以它会执行这个隐含规则。*counter.o*也是类似的情况，当make检查*lexer.o*的时候，并未找到相应的源文件（即使在*src*目录中），所以make会假设这（不存在的源文件）是一个中间文件，而且会查找“从其他源文件产生*lexer.c*文件”的方法。make找到了一个“从*.l*文件产生*.c*文件”的规则，并且注意到*lexer.l*文件的存在。因为不需要进行*lexer.l*的更新，所以make前往用来更新*lexer.c*的命令，这会产生flex命令行。接着，make会从C源文件来更新目标文件。像这样使用一连串的规则来更新一个工作目标的动作称为规则链接（rule chaining）。

接下来，make会检查程序库规范-lfl，它会搜索系统的标准程序库，并且找到*/lib/libfl.a*。

现在make已经找到更新*count_words*时所需要的每个必要条件，所以它会执行最后一个*gcc*命令。最后，make发现自己创建了一个不必保存的中间文件，所以会对它进行清除的操作。

正如所见，在*makefile*文件中使用规则，可以略过许多细节。这些规则经过复杂的交互之后可产生极为强大的功能。尤其是，使用这些内置规则可大量简化*makefile*的规范工作。

你可以通过在脚本中更改变量的值来自定义内置规则。一个典型的规则包含一群变量，以所要执行的程序开头，并且包括用来设定主要命令行选项（比如输出文件、进行优化、进行调试等）的变量。你可以通过运行make --print-data-base列出make具有哪些默认规则（和变量）。

模式

模式规则中的百分比字符（%）大体上等效于Unix shell中的星号（*），它可以代表任意多个字符。百分比字符可以放在模式中的任何地方，不过只能出现一次。百分比字符的正确用法如下所示：

```
%.v
%%.o
wrapper_%
```

在文件名中，百分比以外的字符会按照字面进行匹配。一个模式可以包含一个前缀（prefix）或一个后缀（suffix），或是这两者同时存在。当make搜索所要使用的模式规则时，它首先会查找相符的模式规则工作目标（pattern rule target）。模式规则工作目标必须以前缀开头并且以后缀结尾（如果它们存在的话）。如果找到相符的模式规则工作目标，则前缀与后缀之间的字符会被作为文件名的词干（stem）。接着make会通过将词干替换到必要条件模式（prerequisite pattern）中来检查该模式规则的必要条件。如果所产生的文件名存在，或是可以应用另一项规则进行产生的工作，则会进行比较以及应用规则的动作。词干必须至少包含一个字符。

事实上，你还有可能用到只有一个百分比字符的模式。此模式最常被用来编译Unix可执行程序。例如，下面就是GNU make用来编译程序的若干模式规则：

```
%: %.mod
        $(COMPILE.mod) -o $@ -e $@ $^

%: %.cpp
        $(LINK.cpp) $^ $(LOADLIBES) $(LDLIBS) -o $@

%: %.sh
        cat $< >$@
```

```
        chmod a+x $@
```

这些模式会依次被用来从 Modula 源文件，经过预处理的 C 源文件和 Bourne shell 脚本产生出可执行文件。我们将会在“隐含规则库”一节中看到更多的隐含规则。

静态模式规则

静态模式规则（static pattern rule）只能应用在特定的工作目标上。

```
$(OBJECTS): %.o: %.c
        $(CC) -c $(CFLAGS) $< -o $@
```

此规则与一般模式规则的唯一差别是开头的$(OBJECTS):规范。这将使得该项规则只能应用在$(OBJECTS)变量中所列举的文件上。

此规则与模式规则十分类似。%.o 模式会匹配$(OBJECTS)中所列举的每个目标文件并且取出其词干。然后该词干会被替换进%.c 模式，以产生工作目标的必要条件。如果工作目标模式不存在，则make会发出警告。

如果明确列出工作目标文件比较容易进行扩展名或其他模式的匹配，请使用静态模式规则。

后缀规则

后缀规则（suffix rule）是用来定义隐含规则的最初（也是过时的）方法。旧版的make可能不支持GNU make的模式规则语法，因此你仍然会在许多*makefile*文件中看到后缀规则，所以你最好能够了解它的语法。尽管为目标系统（target system）编译GNU make可以解决*makefile*的可移植性问题，但是在一些罕见的情况下你可能仍旧需要使用后缀规则。

后缀规则中的工作目标，可以是一个扩展名或两个被衔接在一起的扩展名：

```
.c.o:
        $(COMPILE.c) $(OUTPUT_OPTION) $<
```

这令人有些疑惑，因为必要条件的扩展名被摆到开头，而工作目标退居第二位。这个规则所匹配的工作目标以及必要条件跟下面的规则一样：

```
%.o: %.c
        $(COMPILE.c) $(OUTPUT_OPTION) $<
```

后缀规则会通过移除工作目标的扩展名以形成文件的主文件名，以及通过将工作目标的扩展名替换成必要条件的扩展名以形成必要条件。make只会在这两个（被衔接在一起的）扩展名都列在已知扩展名列表中时，才将之视为后缀规则。

上面这个后缀规则就是所谓的双后缀规则(double-suffix rule)，因为它包含了两个扩展名。你还可以使用单后缀规则（single-suffix rule）。没错，单后缀规则只包含一个扩展名，也就是源文件的扩展名。这个规则可用来创建可执行文件，因为Unix上的可执行文件不需要扩展名：

```
.p:
        $(LINK.p) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

这个规则将会从Pascal源文件产生出可执行图像（executable image）。这个规则的作用等效于下面这个模式规则：

```
%: %.p
        $(LINK.p) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

已知扩展名列表（known suffix list）是语法中最奇特的部分，你可以使用 .SUFFIXES 这个特别的工作目标来设定已知的扩展名。下面是 .SUFFIXES 默认值的第一个部分：

```
.SUFFIXES: .out .a .ln .o .c .cc .C .cpp .p .f .F .r .y .l
```

你只要在 *makefile* 文件中加入 .SUFFIXES 规则，就可以将自己的扩展名加入此列表。

```
.SUFFIXES: .pdf .fo .html .xml
```

如果要删除所有已知的扩展名（因为它们的存在干扰到了你的扩展名），你只要在加入 .SUFFIXES 规则时不指定必要条件就行了：

```
.SUFFIXES:
```

你还可以使用命令行选项 --no-builtin-rules（或 -r）。

我们不会在本书其余部分使用这个旧语法，因为make的模式规则较明显也较一般化。

隐含规则

GNU make 3.80版具有90个隐含规则。隐含规则不是模式规则的形式就是后缀规则的形式。这些内置的模式规则可应用于C、C++、Pascal、FORTRAN、ratfor、Modula、Texinfo、TEX（包括Tangle和Weave）、Emacs Lisp、RCS以及SCCS。此外，有些规则是应用在这些语言的支持程序上的，比如cpp、as、yacc、lex、tangle、weave以及dvi工具。

如果你使用到这些工具，你很有可能会发现，内置规则中已经有你所需要的东西了。如果你用到了未受支持的语言，比如Java或XML，那么你就必须编写自己的规则。但不必担心，通常你只需要若干规则就可以支持一种语言了，而且此类规则相当容易编写。

要查看make内置了哪些规则，可使用 --print-data-base（或-p）命令行选项。这将会产生1000行左右的输出。在版本与版权信息之后，make会输出变量的定义，而且会为每个变量前置一行注释以说明其用途。举例来说，一个变量可以是环境变量、默认值、自动变量等。变量之后，make会输出规则。这个规则若是GNU make内置的，则会使用如下的格式：

```
%: %.C
# commands to execute (built-in):
        $(LINK.C) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

这个规则若是在*makefile*中定义的，则会使用如下的格式（注释中将会包含文件名和行号）：

```
%.html: %.xml
# commands to execute (from `Makefile', line 168):
        $(XMLTO) $(XMLTO_FLAGS) html-nochunks $<
```

隐含规则的使用

当make检查一个工作目标时，如果找不到可以更新它的具体规则，就会使用隐含规则。隐含规则的使用很容易：当你将工作目标加入*makefile*时，只要不指定脚本就行了。这会使得make搜索隐含规则以满足工作目标的需要。通常这就是你所要的结果，但是在极少的状况下，你的开发环境可能会引发问题。举例来说，假设你的语言环境掺杂了Lisp和C源代码，如果文件*editor.l*和*editor.c*存在于同一个目录中（假设其中的一个是另一个所使用的低级实现），则make将会把Lisp文件作为flex文件（因为flex文件是以*.l*为扩展名）并把C源文件作为flex命令的输出。如果工作目标是*editor.o*，而且*editor.l*的时间戳在*editor.c*的之后，则make将会试图以flex的输出更新C源文件，结果你的源代码被覆盖掉了。

要解决这个问题，你可以从内置规则库（built-in rule base）中删除这两个规则：

```
%.o: %.l
%.c: %.l
```

一个没有指定脚本的模式规则将会从make的规则库删除相应的规则。尽管你很难遇到此类状况，不过有一件事请铭记在心：内置规则库中所包含的规则与你的*makefile*之间可能存在让你意外的交互。

我们已经看过make在尝试更新工作目标时将规则链接在一起的几个例子。这样做可能会提高复杂度，我们会在此处查看一番。当make试着更新一个工作目标时，它会搜索隐含规则，试图找到与工作目标（target）相符的工作目标模式（target pattern）。对每个与工作目标相符的工作目标模式来说，make会查找相符的必要条件。也就是说，匹配工

作目标模式之后，make会立即查找必要条件（“源”文件）。如果找到了相符的必要条件，就会使用相应的规则。有些工作目标模式具有多个可能的源文件。举例来说，一个*.o*文件可能产生自*.c*、*.cc*、*.cpp*、*.p*、*.f*、*.r*、*.s*以及*.mod*等文件。但如果搜索过所有可能的规则之后还是找不到源文件，结果会怎样？此时，make会再搜索规则一次，不过这一次会认为源文件的匹配是在更新一个新的工作目标。通过递归地进行这样的搜索动作，make可以找到一串用来更新工作目标的规则链。我们可以在这个*lexer.o*范例中看到这个现象：即使*.c*这个中间文件不存在，通过调用从*.l*到*.c*的规则以及从*.c*到*.o*的规则，make仍旧能够更新*lexer.o*这个工作目标。

make可以从它的规则库中产生出卓越的处理程序。让我们做个试验：首先，创建一个空的yacc源文件以及使用ci向RCS登记（也就是说，我们需要一个有版本控制的yacc源文件）：

```
$ touch foo.y
$ ci foo.y
foo.y,v  <--  foo.y
:
initial revision: 1.1
done
```

现在，我们想问make要如何创建一个可执行的*foo*。我们可以用 --just-print（或-n）选项要求make汇报它将会采取哪些行动，但不要实际执行它们。请注意，此刻并没有*makefile*也没有“源”文件，只存在一个RCS文件：

```
$ make -n foo
co  foo.y,v foo.y
foo.y,v  --> foo.y
revision 1.1
done
bison -y  foo.y
mv -f y.tab.c foo.c
gcc    -c -o foo.o foo.c
gcc   foo.o   -o foo
rm foo.c foo.o foo.y
```

找到了隐含规则链之后，make可以作出如下的决定：如果目标文件*foo.o*存在，就可以创建可执行文件*foo*；如果C源文件*foo.c*存在，就可以创建*foo.o*；如果yacc源文件*foo.y*存在，就可以创建*foo.c*。最后，make发现它可以通过从RCS文件*foo.y,v*中调出（check out）文件*foo.y*来创建该文件。一旦make将此计划公式化之后，就会以co调出*foo.y*，以bison将之转换成*foo.c*，以gcc将之编译成*foo.o*，最后再次以gcc将之链接成*foo*。以上这些步骤全都产生自隐含规则库。酷极了！

链接规则的过程中所产生的文件称为中间文件（intermediate file），make会对它们进行特别的处理。首先，因为中间文件不会在工作目标中出现（否则它们就不是中间文件

了），所以make不会更新中间文件；其次，make创建中间文件本身就有更新工作目标的副作用，所以make在结束运行之前会删除这些中间文件。

规则的结构

内置规则具有标准的结构，好让它们容易自定义。现在让我们来查看此结构，并探讨有关“自定义”（customization）这方面的议题。下面是（现在我们所熟悉的）从C源文件来更新目标文件的规则：

```
%.o: %.c
        $(COMPILE.c) $(OUTPUT_OPTION) $<
```

这个规则的“自定义”完全取决于其所用到的变量。我们在此处看到了两个变量，其中的COMPILE.c是由多个其他变量所定义而成的：

```
COMPILE.c = $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
CC = gcc
OUTPUT_OPTION = -o $@
```

你只要变更CC变量的设定值就可以更换C编译器。此外还包括用来设定编译选项的变量（CFLAGS）、用来设定预处理器选项的变量（CPPFLAGS）以及用来设定结构选项的变量（TARGET_ARCH）。

在内置规则中使用变量的目的，就是让规则的自定义尽可能简单。因此，当你在*makefile*文件中设定这些变量时，务必谨慎。如果设定这些变量时随意为之，将会破坏终端用户自定义它们的能力。例如，你在*makefile*文件中做了如下的赋值动作：

```
CPPFLAGS = -I project/include
```

如果用户想在命令行上加入CPP的定义，他们一般会像这样来调用make：

```
$ make CPPFLAGS= -DDEBUG
```

如果他们真的这么做了，将会意外删除（想必是）编译时所需要的-I选项。命令行上所设定的变量将会覆盖掉该变量原有的值（参见“变量来自何处”一节中关于命令行赋值的详细细节）。所以不当地在*makefile*中设定CPPFLAGS将会破坏大多数用户预设的自定义结果。为了避免此问题，你可以使用如下的方式重新定义编译变量：

```
COMPILE.c = $(CC) $(CFLAGS) $(INCLUDES) $(CPPFLAGS) $(TARGET_ARCH) -c
INCLUDES = -I project/include
```

或者你可以使用附加形式的赋值动作，我们将会在“其他形式的赋值动作”一节中讨论此做法。

支持源代码控制系统的隐含规则

make的隐含规则还针对两种源代码控制系统RCS和SCCS提供了支持。可惜make心有余而力不足，未能跟上源代码控制系统及现代软件工程日新月异的脚步。我从未发现有人使用make所支持的源代码控制功能，也未曾看到过任何的源代码控制软件使用make的这个功能。基于以下几个理由，建议各位不要使用make的这个功能。

首先，make对源代码控制系统RCS和SCCS所提供的支持尽管在当时相当有用，然而现在RCS和SCCS几乎已经完全被Concurrent Version System（简称CVS）或其他商用工具所取代。事实上，CVS的内部是使用RCS来管理单独的文件。然而，如果你直接使用RCS，则会在项目跨越多个目录或具有多个开发人员时引发极大的问题。尤其是，CVS原先就是设计来填补RCS在这方面的缺陷。make目前尚不支持CVS，这或许不是一件坏事（注2）。

你不难发现，软件的生命周期变得复杂了，应用程序很难从一个版本顺利地前进到下一个版本。较典型的例子是：一个应用程序的一个或多个版本同时在进行开发，一个或多个版本被发布出供大家使用（而且有缺陷需要修正）。CVS所提供的强大功能有助于软件的多个版本并行开发时的管理工作。不过这也意味着，开发者必须知道他当前所操作的是程序代码的哪个版本。让*makefile*于编译期间自动调出源文件，必须注意所调出的是哪个版本的源文件，以及它是否兼容于开发者工作目录中已经存在的源文件。在许多实际的开发环境中，开发者一日之内就可以对同一个应用程序的三个以上的不同版本进行开发。在如此复杂的环境里，很难用软件自动更新你的源代码树。

此外，CVS有一个很有用的功能，那就是允许你访问远程的仓库（repository）。在许多实际的开发环境中，CVS的仓库（受控文件所构成的数据库）并非放在开发者自己的机器上，而是放在服务器上。尽管现在网络的访问速度已经很快了（尤其是在局域网络中），但是在搜索源文件的时候以make来搜索网络服务器并不是一个好主意。

所以，尽管有可能使用隐含规则来连接RCS和SCCS，但却没有任何规则可用来访问CVS以让你聚集源文件或*makefile*。我也不认为这么做有多大意义。从另一方面来说，在*makefile*中使用CVS却是相当明智的做法。举例来说，这么做可确保当前的源文件被正确签入（check in）以及确保版本编号或测试结果的正确性。*makefile*的作者只需在意CVS的用法，而不必去管CVS与make要如何整合在一起。

注2： CVS接着又被其他较新的工具所取代。尽管CVS目前是使用得最广泛的源代码控制系统，不过subversion（*http://subversion.tigris.org*）看起来有希望成为下一个新浪潮。

一个简单的help命令

大型的*makefile*文件包含了许多工作目标，这让用户很难搞清楚。减少此类问题的方法之一，就是以一个简洁的help命令为默认目标。然而，手动维护帮助文本（help text）总是很麻烦。要避免此问题，可直接从make的规则库中收集可用的命令。接下来的help工作目标将会把可用的make工作目标显示成一份经过排序的四个字段的列表：

```
# help - The default goal
.PHONY: help
help:
	@ $(MAKE) --print-data-base --question |               \
	  $(AWK) '/^[^.%][-A-Za-z0-9_]*:/                      \
	         { print substr($$1, 1, length($$1)-1) }' |    \
	  $(SORT) |                                            \
	  $(PR) --omit-pagination --width=80 --columns=4
```

此规则的命令脚本组成自一条管道（pipeline）。使用 --print-data-base选项可输出make的规则库，使用--question选项可避免make执行其中的任何命令。此规则库接着会被送往一个简单的awk过滤器，以便收集任何并非以百分比号和点号开头（分别代表模式规则和后缀规则）的工作目标以及删掉额外的信息。最后，工作目标列表会经过排序，并且以四个字段的格式输出。

另一种做法就是对*makefile*文件本身使用awk命令。这需要对被引入（included）的*makefile*文件做特殊的处理（参见“include指令”一节），而且不能处理其所产生的规则。通过让make处理这些元素以及汇报其所产生的规则，就能自动完成所有的工作。

特殊工作目标

特殊工作目标（special target）是一个内置的假想工作目标（phony target），用来变更make的默认行为。例如，.PHONY这个特殊工作目标用来声明它的必要条件并不代表一个实际的文件，而且应该被视为尚未更新。.PHONY将会是最常见的特殊工作目标，但是你还会看到其他的特殊工作目标。

特殊工作目标的语法跟一般工作目标的语法没有不同，也就是*target: prerequisite*，但是target并非文件而是一个假想工作目标。它们实际上比较像是用来修改make内部算法的指令。

目前共有12个特殊工作目标，可分成三类：第一类用来在更新工作目标时修改make的行为；第二类的动作就好像是make的全局标记，用来忽略相应的工作目标；最后是.SUFFIXES这个工作目标，用来指定旧式的后缀规则（参见“后缀规则”一节）。

下面列出（除了 .PHONY 之外的）最有用的工作目标修饰符（target modifier）：

.INTERMEDIATE

这个特殊工作目标的必要条件会被视为中间文件。如果make在更新另一个工作目标期间创建了该文件，则该文件将会在make运行结束时被自动删除。如果在make想要更新该文件之际该文件已经存在了，则该文件不会被删除。

当你要自定义规则链时，这会非常有用。举例来说，大多数Java工具都可以接受Windows形式的文件列表。自定义规则以建立文件列表并把它们的输出文件视为中间文件，可让make清除这些临时性的文件。

.SECONDARY

这个特殊工作目标的必要条件会被视为中间文件，但不会被自动删除。.SECONDARY最常用来标示存储在程序库（library）里的目标文件（object file）。按照惯例，这些目标文件一旦被加入档案库后（archive）就会被删除。在项目开发期间保存这些目标文件，但仍使用make进行程序库的更新，有时会比较方便。

.PRECIOUS

当make在运行期间被中断时，如果自make启动以来该文件被修改过，make将会删除它正在更新的工作目标文件。因此make不会在编译树（build tree）中留下尚未编译完成（可能已经走样）的文件。但有些时候你却不希望make这么做，特别是在该文件很大而且编译的代价很高时。如果该文件极为珍贵（precious），你就该用 .PRECIOUS 来标示它，这样make才不会在自己被中断时删除该文件。

你很少会用到 .PRECIOUS，但是当有此需要时，它通常是一个救生员（life saver）的角色。请注意，如果与规则相应的命令在运行时发生错误，make将不会执行自动删除的动作。make只有在自己被信号中断时才会这么做。

.DELETE_ON_ERROR

.DELETE_ON_ERROR的作用与 .PRECIOUS 相反。将工作目标标示成 .DELETE_ON_ERROR，表示如果与规则相应的任何命令在运行时发生错误的话，就应该删除该工作目标文件。make通常只有在自己被信号中断时才会删除该工作目标文件。

其他的特殊工作目标将会在适当的时候加以介绍。我们将会在第三章探讨 .EXPORT_ALL_VARIABLES，在第十章探讨涉及并行执行（parallel execution）的工作目标。

自动产生依存关系

当我们将单词计数程序重构为使用头文件时，有个棘手的小问题会找上我们。尽管就此

例而言，我们可以轻易地在*makefile*文件中手动加入目标文件与C头文件的依存关系，但是在正常的程序（而不是玩具程序）里这是一个烦人以及动辄得咎的工作。事实上，在大多数程序中，这几乎是不可能的事，因为大多数的头文件还会包含其他头文件所形成的复杂树状结构。举例来说，在我的系统上，头文件*stdio.h*（这是C语言中最常被引用的头文件）会被扩展成包含15个其他的头文件。以手动方式解析这些关系是一个令人绝望的工作。但如果这些文件的重新编译失败，可能会导致数小时的调试，或者更糟的是因此产生出一个具有缺陷的程序。到底该这么办呢？

计算机擅长于搜索以及模式匹配。让我们使用一个程序来找出这些文件之间的关系，我们甚至可以使用此程序以*makefile*的语法编写出这些依存关系。正如你的猜测，此类程序已经存在，至少对C/C++而言是这样。在gcc中这是一个选项，许多其他的C/C++编译器也都会读进源文件并写出*makefile*的依存关系。例如，下面是我为*stdio.h*寻找依存关系的方式：

```
$ echo "#include <stdio.h>" > stdio.c
$ gcc -M stdio.c
stdio.o: stdio.c /usr/include/stdio.h /usr/include/_ansi.h \
  /usr/include/newlib.h /usr/include/sys/config.h \
  /usr/include/machine/ieeefp.h /usr/include/cygwin/config.h \
  /usr/lib/gcc-lib/i686-pc-cygwin/3.2/include/stddef.h \
  /usr/lib/gcc-lib/i686-pc-cygwin/3.2/include/stdarg.h \
  /usr/include/sys/reent.h /usr/include/sys/_types.h \
  /usr/include/sys/types.h /usr/include/machine/types.h \
  /usr/include/sys/features.h /usr/include/cygwin/types.h \
  /usr/include/sys/sysmacros.h /usr/include/stdint.h \
  /usr/include/sys/stdio.h
```

不错吧。你可能会嚷着："现在我必须先运行gcc，再使用编辑器将-M的结果贴到我的*makefile*中。真是麻烦！"如果这是最后的答案，你说得没错。传统上有两种方法可用来将自动产生的依存关系纳入*makefile*。第一种也是最古老的方法，就是在*makefile*结尾加入如下一行内容：

```
# Automatically generated dependencies follow - Do Not Edit
```

然后编写一个脚本以便加入这些自动产生的脚本。这么做当然比手动加入要好，但不是很漂亮。第二种方法就是为make加入一个include指令。如今大多数的make版本都支持include指令，GNU make当然一定可以这么做。

因此，诀窍就是编写一个*makefile*工作目标，此工作目标的动作就是以-M选项对所有源文件执行gcc，并将结果存入一个依存文件中（dependency file），然后重新执行make以便把刚才所产生的依存文件引入*makefile*，这样就可以触发我们所需要的更新（即加入）动作。在GNU make中，你可以使用如下的规则来实现此目的：

```
depend: count_words.c lexer.c counter.c
        $(CC) -M $(CPPFLAGS) $^ > $@
include depend
```

运行make以编译程序之前，你首先应该执行make depend以产生依存关系。这么做虽然不错，但是当人们对源文件加入或移除依存关系时，通常不会重新产生*depend*文件。这会造成无法重新编译源文件，整个工作又会变得一团乱。

在GNU make中，你可以使用一个很酷的功能以及一个简单的算法来解决此问题。首先介绍这个简单的算法。如果我们为每个源文件产生依存关系，将之存入相应的依存文件（一个扩展名为*.d*的文件）并以该*.d*文件为工作目标来加入此依存规则（dependency rule），这样，当源文件被改变时，make就会知道需要更新该*.d*文件（以及目标文件）：

```
counter.o counter.d: src/counter.c include/counter.h include/lexer.h
```

你可以使用如下的模式规则以及（不容易看得懂的）命令脚本（摘录自GNU make的使用手册）来产生这项规则：（注3）

```
%.d: %.c
        $(CC) -M $(CPPFLAGS) $< > $@.$$$$;                    \
        sed 's,\($*\)\.o[ :]*,\1.o $@ : ,g' < $@.$$$$ > $@; \
        rm -f $@.$$$$
```

现在介绍这个很酷的功能。make会把include指令所指定的任何文件视为一个需要更新的工作目标。所以，如果我们表明要引入*.d*文件，则make会在读进*makefile*文件时自动创建这些*.d*文件。我们的*makefile*加入了自动产生依存关系的功能之后会变成下面这样：

```
VPATH    = src include
CPPFLAGS = -I include

SOURCES  = count_words.c \
```

注3： 这是一个令人印象深刻的小型命令脚本，不过我认为需要做些说明。首先，我们会以-M选项来使用C编译器，以便创建一个包含此工作目标的依存关系的临时文件。该临时文件的名称由工作目标$@以及具唯一性的数字扩展名$$$$所组成。在shell中，变量$$会返回当前所运行的shell的进程编号。因为进程编号具有唯一性，所以这么做将可以产生一个独一无二的文件名。然后我们会使用sed以*.d*文件为工作目标加入此规则。sed表达式由搜索部分\($*\)\.o[:]*以及替换部分\1.o $@ :（以逗号为分隔符）组成。搜索表达式的开头是工作目标的主文件名$*，被括在正则表达式（regular expression，简称RE）的分组\(\)中，后面跟着扩展名\.o。工作目标的文件名之后，将会出现零个或多个空格或冒号，即[:]*。替换部分会通过引用第一个RE分组来恢复最初的工作目标并且附加上扩展名，即\1.o，然后加入依存文件工作目标$@。

```
            lexer.c        \
            counter.c

count_words: counter.o lexer.o -lfl
count_words.o: counter.h
counter.o: counter.h lexer.h
lexer.o: lexer.h

include $(subst .c,.d,$(SOURCES))

%.d: %.c
        $(CC) -M $(CPPFLAGS) $< > $@.$$$$;                    \
        sed 's,,(S*\)\.o[ :]*,\1.o $@ : ,g' < $@.$$$$ > $@;    \
        rm -f $@.$$$$
```

include指令总是应该放在手动编写的依存关系的后面，这样默认目标才不会被某个依存文件抢走。include指令可用来指定一串文件（文件名中可以包含通配符）。我们在此处使用了make函数subst来将一串源文件的文件名转换成一串依存文件的文件名。我们将会在"字符串函数"一节中探讨subst的细节，现在你只要知道subst可用来将$(SOURCES)里的文本从*.c*字符串转换成*.d*字符串。

如果针对此*makefile*以--just-print选项来运行make，则会得到如下的结果：

```
$ make --just-print
Makefile:13: count_words.d: No such file or directory
Makefile:13: counter.d: No such file or directory
gcc -M -I include src/counter.c > counter.d.$$;          \
sed 's,\(counter\)\.o[ :]*,\1.o counter.d : ,g' < counter.d.$$ >
counter.d; \
rm -f counter.d.$$
flex -t src/lexer.l > lexer.c
gcc -M -I include lexer.c > lexer.d.$$;          \
sed 's,\(lexer\)\.o[ :]*,\1.o lexer.d : ,g' < lexer.d.$$ > lexer.d;
\
rm -f lexer.d.$$
gcc -M -I include src/count_words.c > count_words.d.$$;
\
sed 's,\(count_words\)\.o[ :]*,\1.o count_words.d : ,g' < count_words.d.
$$
count_words.d;   \
rm -f count_words.d.$$
rm lexer.c
gcc -I include -c -o count_words.o src/count_words.c
gcc -I include -c -o counter.o src/counter.c
gcc -I include -c -o lexer.o lexer.c
gcc  count_words.o counter.o lexer.o /lib/libfl.a  -o count_words
```

一开始，make的响应看起来如同一个make的错误信息。不过不必担心，这只是一个警告而已。起先make会搜索引入文件，但是找不到它们，所以make会在搜索"用来创建

这些文件的规则”之前发出“No such file or directory”（没有这样的文件或目录）这样的警告。若不想看到这个警告信息，只要为include指令前置一个连接号（-）即可。警告信息之后可以看到make以-M选项调用gcc以及执行sed命令的动作。请注意，make必须调用flex以便创建*lexer.c*，然后在开始满足默认目标之前删除*lexer.c*这个临时文件。

这一节只介绍了自动产生依存关系的功能，还有许多没有谈到，像是如何为其他语言产生依存关系或是编译树的布局。我们将会在本书的第二部分深入探讨这方面的议题。

管理程序库

程序库（archive library，通常简称为library或archive）是一个特殊的文件，该文件内含其他被称为成员（member）的文件。程序库可用来将相关的目标文件聚集成较容易操作的单元。例如，C的标准程序库*libc.a*就包含了许多低级的C函数。因为程序库如此常见，所以make对它们的创建、维护以及引用提供了特别的支持。程序库的建立及修改可通过ar程序来进行。

现在来看一个例子。我们可以把单词计数程序中可重复使用的部分放到一个可以重复使用的程序库里。这个程序库由两个文件组成：*counter.o*和*lexer.o*。我们可以使用ar命令来创建此程序库：

```
$ ar rv libcounter.a counter.o lexer.o
a - counter.o
a - lexer.o
```

选项r代表我们想要以指定的目标文件来替换（replace）程序库里的成员，而选项v代表ar必须详细地（verbosely）告诉我们，它做了哪些动作。即使该程序库原本就不存在，我们还是可以使用r选项。rv选项之后的第一个参数是程序库的文件名，接着是一串目标文件（如果该程序库不存在，则有些版本的ar必须指定c选项，才会进行创建（create）程序库的动作，不过对于GNU ar不必这么做）。执行ar命令之后所显示的信息里，你将会看到它以“a”来表示目标文件已被加入程序库里了。

以r选项来使用ar命令，可让我们立即创建或更新一个程序库：

```
$ ar rv libcounter.a counter.o
r - counter.o
$ ar rv libcounter.a lexer.o
r - lexer.o
```

执行ar命令之后所显示的信息里，你将会看到它以“r”来表示目标文件已被替换到程序库中。

一个程序库被链接到一个可执行文件的方法有好几种，最简单的方法就是在命令行上直接指定该程序库。编译器或链接器将会以文件的扩展名来判断命令行上特定文件的类型并做正确的事情：

```
cc count_words.o libcounter.a /lib/libfl.a -o count_words
```

此处，cc 将会把 *libcounter.a* 和 */lib/libfl.a* 这两个文件视为程序库，并对它们搜索未定义的符号。在命令行上引用程序库的另一个方法就是使用 -l 选项：

```
cc count_words.o -lcounter -lfl -o count_words
```

如你所见，使用这个选项可以省略程序库文件名的前缀（prefix）以及扩展名（suffix）。-l 选项可让命令行更加紧凑并且较容易阅读，不过它还具有极为有用的功能：当 cc 看到 -l 选项时，就会在系统的标准程序库目录中搜索相应的程序库。这样，程序员就不必知道程序库的确切位置，而且可以让其所使用的命令行更具可移植性。此外，在支持共享程序库（在 Unix 系统上就是以 *.so* 为扩展名的程序库）的系统上，链接器在搜索程序库（archive library）之前，会先搜索共享程序库（shared library）。这让程序在未指明的状况下也能得益于共享程序库。这是GNU的链接器/编译器默认的行为模式。较旧版的链接器/编译器并不会进行此优化动作。

若要变更编译器所使用的搜索路径，你可以使用 -L 选项来指定所要搜索的目录以及搜索的次序。这些目录应该被加在系统程序库之前，并且会被应用在命令行中的所有 -l 选项上。事实上，前面的那个例子会链接失败，因为当前目录（current directory）并未被列在 cc 的程序库搜索路径之中。我们只要以如下的方式加入当前目录就可以修正此错误：

```
cc count_words.o -L. -lcounter -lfl -o count_words
```

程序库为一个程序的编译过程增添了些许的复杂性。make如何协助我们简化此状态呢？GNU make 为程序库的创建以及链接提供了特别的支持，让我们来看看它的做法。

创建与更新程序库

在 *makefile* 里，指定程序库的方式跟指定任何其他文件没有不同，也就是指出它的文件名。下面就是一个用来创建程序库的简单规则：

```
libcounter.a: counter.o lexer.o
        $(AR) $(ARFLAGS) $@ $^
```

此处使用了AR变量中的对于ar程序的内置定义，以及ARFLAGS变量中的标准选项rv。这个程序库的文件名会被自动设定到 $@ 里，而必要条件会被自动设定到 $^ 里。

现在，如果你以*libcounter.a*作为*count_words*的一个必要条件，则make在链接可执行文件之前会更新该程序库。然而，这么做会有一个问题：程序库里的所有成员每次都会被替换掉，即使它们并未被修改。宝贵的时间就这样浪费了，不过我们可以做得更好：

```
libcounter.a: counter.o lexer.o
        $(AR) $(ARFLGS) $@ $?
```

如果你将$^替换成$?，则make只会把时间戳在工作目标（的时间戳）之后的目标文件传递给ar。

我们还可以做得更好？或许可以，或许不行。尽管make可让我们更新程序库里的个别文件以及为每个目标文件成员执行一个ar命令，不过在我们探索相关细节之前，这种构建程序库的做法中的几点值得我们加以注意。make的主要目标之一，就是只更新过时(out of date)的文件，好让处理器的使用更有效率。可惜，这种为每个过时的成员调用一次ar的做法，很快就会让处理器陷入泥潭。如果程序库所包含的文件超过10个，此时为每个更新动作调用一次ar，其代价将会比语法是否“精致”还大。通过在一个具体规则中使用前面的简单方法以及调用ar，我们可以为所有文件执行一次ar编译并省掉许多的fork/exec调用。此外，以r选项来执行ar在许多系统上是个效率相当差的工作。例如，在1.9 GHz Pentium 4的机器上，从头开始编译一个大型的程序库（包含14216个成员，总计55 MB）耗时4分24秒。然而，程序库建立之后，若以ar r来更新单一目标文件，则需要28秒的时间。所以，如果我们所要替换的文件超过10个，从头开始编译程序库反而比较快。在这样的情况下，使用自动变量$?对每个被修改的目标文件进行程序库的更新，或许应该更谨慎才对。对于较小型的程序库以及速度较快的处理器来说，你不必为了效能的因素去采用前面的简单做法，而舍弃后面较精致的语法。此时，使用特殊的程序库支持是比较好的做法。

在GNU make中，你可以使用如下的符号来引用程序库里的成员：

```
libgraphics.a(bitblt.o): bitblt.o
        $(AR) $(ARFLAGS) $@ $<
```

此处，程序库的文件名是*libgraphics.a*，而成员的文件名是*bitblt.o*（用于bit block transfer）。语法*libname*.a(*module*.o)可用来引用特定程序库里的特定成员。这个工作目标的必要条件就是目标文件本身，而所要执行的命令就是将该目标文件加入程序库。在命令中使用自动变量$<可取得第一个必要条件。事实上，内置的模式规则就是在做这件事。

最后，我们的*makefile*看起来会像下面这样：

```
VPATH    = src include
CPPFLAGS = -I include
```

```
count_words: libcounter.a /lib/libfl.a

libcounter.a: libcounter.a(lexer.o) libcounter.a(counter.o)

libcounter.a(lexer.o): lexer.o
        $(AR) $(ARFLAGS) $@ $<

libcounter.a(counter.o): counter.o
        $(AR) $(ARFLAGS) $@ $<

count_words.o: counter.h
counter.o: counter.h lexer.h
lexer.o: lexer.h
```

对此*makefile*执行make将会产生如下的输出结果：

```
$ make
gcc  -I include  -c -o count_words.o src/count_words.c
flex  -t src/lexer.l> lexer.c
gcc  -I include  -c -o lexer.o lexer.c
ar rv libcounter.a lexer.o
ar: creating libcounter.a
a - lexer.o
gcc  -I include  -c  o counter.o srt/counter.c
ar rv libcounter.a counter.o
a - counter.o
gcc   count_words.o libcounter.a /lib/libfl.a   -o count_words
rm lexer.c
```

注意程序库更的新规则。自动变量$@会被扩展成程序库的文件名称，即使该工作目标在*makefile*里是*libcounter.a(lexer.o)*。

最后，还有一件事应该提到：程序库会为它所包含的符号提供索引。较新版的ar程序（像GNU ar）会在新的成员加入程序库的时候，自动管理此索引。然而，许多较旧版的ar并不会这么做，此时你就必须使用另一个程序（像ranlib）来创建或更新程序库的索引。在这些系统上，隐含规则将无法更新程序库。对于这些系统，你必须使用如下的规则：

```
libcounter.a: libcounter.a(lexer.o) libcounter.a(counter.o)
        $(RANLIB) $@
```

对于大型的程序库，你可以使用如下的规则：

```
libcounter.a: counter.o lexer.o
        $(RM) $@
        $(AR) $(ARFLGS) $@ $^
        $(RANLIB) $@
```

当然，这个用来管理程序库成员的语法也可以使用在隐含规则中。GNU make随附了一个用来更新程序库的内置规则。如果使用这个规则，我们的*makefile*就会变成下面这样：

```
VPATH    = src include
CPPFLAGS = -I include
count_words: libcounter.a -lfl
libcounter.a: libcounter.a(lexer.o) libcounter.a(counter.o)
count_words.o: counter.h
counter.o: counter.h lexer.h
lexer.o: lexer.h
```

以程序库为必要条件

当程序库作为必要条件时，可以使用标准的文件名语法或 -l 语法来引用它们。使用文件名语法时：

```
xpong: $(OBJECTS) /lib/X11/libX11.a /lib/X11/libXaw.a
        $(LINK) $^ -o $@
```

链接器将只会读取命令行上所列出的程序库文件，以及按正常的方式来处理它们。使用 -l 语法时，必要条件并非真正的文件名称：

```
xpong: $(OBJECTS) -lX11 -lXaw
        $(LINK) $^ -o $@
```

当 -l 的语法出现在必要条件上时，make 将会搜索相应的程序库（而且会先搜索共享程序库）以及将它的值（以绝对路径的形式）替换到变量 $^ 和 $? 里。第二种语法的最大优点是，它让你可以使用“优先搜索共享程序库”的功能，即使系统的链接器无法进行这些工作。它的另一个优点是，因为你可以自定义 make 的搜索路径，所以 make 可以找到应用程序的程序库以及系统程序库。总之，第一种语法将会忽略共享程序库并使用链接行（link line）上所指定的程序库。第二种语法会使得 make 优先选择共享程序库，也就是说，在 make 决定使用非共享版本（archive version）的 *X11* 程序库之前，会先搜索共享版本（shared version）的 *X11* 程序库。供 -l 语法辨别程序库文件名格式的模式就存放在 .LIBPATTERNS 变量里，你可以通过它来自定义其他程序库的文件名格式。

可惜，有个小问题。如果 *makefile* 已经将程序库文件指定为工作目标，它就不能在必要条件里对该文件使用 -l 选项。举例来说，对于下面的 *makefile*：

```
count_words: count_words.o -lcounter -lfl
        $(CC) $^ -o $@

libcounter.a: libcounter.a(lexer.o) libcounter.a(counter.o)
```

运行 make，将会显示如下的错误信息：

```
No rule to make target '-lcounter', needed by 'count_words'
```

这是因为make不会把-lcounter扩展成*libcounter.a*并去搜索工作目标，make只会去搜索程序库。所以，如果要在*makefile*里进行程序库的编译工作，必须使用文件名的语法。

要让复杂程序的链接工作没有错误，可能需要使用一些手段。链接器会依次搜索命令行上所指定的程序库。所以，如果程序库A包含了一个未定义的符号，例如open，而且该符号定义在程序库B中，那么你就必须在链接命令行（link command line）上于B之前指定A（也就是A需要B）。否则，一旦链接器读进A并且看到未定义的符号open，再回头来读取B就太迟了，链接器并不会回头来读取前面的程序库。如你所见，程序库在命令行上的顺序相当重要。

一旦工作目标的必要条件存放在变量^和?之后，它们的顺序就会被保存下来。所以，如果在前面的例子里使用$^，将会扩展成顺序跟必要条件完全一样的文件列表，即使必要条件跨越多项规则也是如此。也就是说，每项规则的必要条件会依照它们被看到的顺序被依次附加到该工作目标的必要条件列表中。

一个比较相关的问题就是程序库之间的相互引用（mutual reference），这通常称为循环引用（circular reference，或简称circularity）。假设程序有所变动，使得程序库B现在引用了程序库A中所定义的符号。我们知道A必须放在B的前面，不过现在B必须放在A的前面。这个问题的解决方案，就是在B之前与之后使用*A*：*-lA -lB -lA*。在大型且复杂的程序中，通常需要对程序库重复进行此步骤，有时会超过两次。

这么做将会对make造成一个小问题，因为自动变量通常会丢弃重复的部分。举例来说，假如我们必须重复使用一个程序库必要条件以满足程序库循环引用的需要：

```
xpong: xpong.o libui.a libdynamics.a libui.a -lX11
        $(CC) $^ -o $@
```

这个必要条件列表将会被处理成如下的链接命令：

```
gcc xpong.o libui.a libdynamics.a /usr/lib/X11R6/libX11.a -o xpong
```

为解决$^的这种行为，make另外提供了$+变量。此变量如同$^，不过它会保留重复的必要条件。所以，如果使用$+：

```
xpong: xpong.o libui.a libdynamics.a libui.a -lX11
        $(CC) $+ -o $@
```

那么这个必要条件列表将会被处理成如下的链接命令：

```
gcc xpong.o libui.a libdynamics.a libui.a /usr/lib/X11R6/libX11.a -o xpong
```

双冒号规则

双冒号规则（double-colon rule）是一个模糊的功能，它会依据必要条件的时间戳是否在工作目标（的时间戳）之后，以不同的命令来更新同一个工作目标。通常，当一个工作目标多次出现时，所有的必要条件会被衔接成一份列表，而且只让一个命令脚本进行更新的动作。然而，对双冒号规则而言，相同的工作目标每出现一次就会被视为一个独立的实体，必须进行独立的处理。这意味着，对同一个工作目标来说，所有的规则必须是同一个类型，也就是说，它们若非全都是双冒号规则，就应该全都是单冒号规则。

实际上，我们很难为这个功能找到有用的范例（这就是为何我会称它是一个模糊的功能），下面是一个假造的例子：

```
file-list:: generate-list-script
        chmod +x $<
        generate-list-script $(files) > file-list

file-list:: $(files)
        generate-list-script $(files) > file-list
```

我们可以通过两种方式重新产生*file-list*工作目标。如果产生脚本（generating script）被更新，我们会将该脚本设为可执行，接着加以运行，如果是源文件被变更，我们只会运行该脚本。尽管这个例子有点牵强，不过你可借此感觉一下如何应用这个功能。

到目前为止，我们只是将焦点放在特定的语法以及功能的行为上，不包括如何将它们应用在较复杂的情况下——这是本书第二部分的重点。介绍过规则的大部分功能（这是make的基础）之后，接下来要讲的是变量和命令。

第三章

变量与宏

到目前为止，我们已经看过了*makefile*的变量以及将它们应用在内置和具体规则中的许多范例。不过这些都只是粗浅的例子。变量和宏越复杂，GNU make的功能就越强大。

在我们继续任何探讨之前，最好能先了解make所包含的两种语言。第一种语言用来描述工作目标与必要条件所组成的依存图（此语言的相关说明可参考第二章）。第二种语言是宏语言，用来进行文字的替换。你可能已经熟悉其他的宏语言，像C预处理器、m4、TEX以及宏汇编器（macro assembler）。如同这些其他的宏语言，make允许你为较长的字符序列定义简写并在你的程序中使用该简写，宏处理器将会认出你的简写并将它们替换成展开后的形式。虽然你可以把*makefile*的变量想成传统程序语言的变量，不过宏变量和传统变量之间是有差别的。宏变量会被“就地”扩展，其所产生的文本字符串还可以做进一步的扩展。继续读下去，这个差别将会变得更加明显。

一个变量名称几乎可以由任何字符组成，包括大部分的标点符号。即使空格也可以使用，但如果你自认精神正常，就应该避免这么做。事实上只有:、#和=等字符不允许使用在变量名称中。

变量名称是区分大小写的，所以cc和CC所指的是不同的变量。要取得某个变量的值，请用$()括住该变量的名称。有一个特例：变量名称若为单一字母（letter）则可以省略圆括号，所以请直接使用$letter。这就是为何自动变量的指定不必使用圆括号。一个原则是：指定变量名称时应该加上圆括号，避免使用单一字母的变量名称。

你还可以使用花括号来扩展变量，例如${CC}。事实上，你将会经常看到这种做法，尤其是在比较旧的*makefile*文件中。很难说使用哪个会比较好，所以请择一使用，选定后务必维持其一致性。有些人的做法会像shell那样，将花括号用于变量的引用，将圆括号用于函数的调用。现代化的*makefile*多半会使用圆括号，这也是本书将采用的方法。

当变量用来表示用户在命令行上或环境中所自定义的常数时，习惯上会全部以大写来编写其名称，单词之间以下划线符号（_）隔开。至于只在*makefile*文件中出现的变量，则会全部以小写来编写其名称，单词之间以下划线符号隔开。最后，在本书中，内含用户自定义函数的变量以及宏都会以小写来编写其名称，单词之间以破折号（-）隔开。其他的命名习惯将会适时加以说明（接下来的范例用到了我们尚未说明的功能。我准备用它们来解说变量的命名习惯，读者现在不必太在意赋值符号右边的部分）。

```
# 常数
CC    := gcc
MKDIR := mkdir -p

# 内部变量
sources = *.c
objects = $(subst .c,.o,$(sources))

# 函数
maybe-make-dir  = $(if $(wildcard $1),,$(MKDIR) $1)
assert-not-null = $(if $1,,$(error Illegal null value.))
```

一个变量的值由赋值符号（assignment symbol）右边已删除前导空格（leading space）的所有字组成。跟在所有字之后的空格（trailing spaces）则不会被删除。这有时会导致问题，举例来说，如果变量的值包含了跟在后面的空格，而且随后被使用在命令脚本中：

```
LIBRARY = libio.a # LIBRARY 的值包含了一个跟在后面的空格
missing_file:
        touch $(LIBRARY)
        ls -l | grep '$(LIBRARY)'
```

变量的值后面跟着一个空格，加上注释符号之后，这会变得显而易见（即使未加上注释符号，也并不代表这个跟在后面的空格不存在）。对这个*makefile*运行make，将会得到如下的结果：

```
$ make
touch libio.a
ls -l | grep 'libio.a '
make: *** [missing_file] Error 1
```

因为grep的搜索字符串也包含了跟在后面的空格，所以无法在ls的输出中找到该文件的名称。稍后我们将会深入探讨空格（whitespace）的相关细节。现在让我们进一步讨论变量。

变量的用途

一般来说，以变量来代表外部程序是一个不错的主意，这让*makefile*的用户较容易针对他们特有的环境来改写*makefile*。举例来说，一个系统上常常会包含awk的各种版本：

awk、nawk、gawk。这个时候，你可以建立一个AWK变量来保存awk程序的名称，让*makefile*较容易使用。此外，如果你的环境以安全为主，那么你最好通过绝对路径来取用外部程序，以避免用户搜索路径所带来的安全问题。如果特洛伊木马（trojan horse）版本的系统程序被安装在用户搜索路径中的某处，这么做还可以减少可能的安全问题。当然，绝对路经也会同时降低*makefile*的可移植性。你可以依照需求自己决定要怎么做。

变量可用来保存简单的常数，也可用来存放用户自定义的命令序列。例如，下面的设定可用来汇报尚未使用的磁盘空间：（注1）

```
DF  = df
AWK = awk
free-space := $(DF) . | $(AWK) 'NR == 2 { print $$4 }'
```

变量的用途不止这两种，还有很多，稍后我们就会看到。

变量的类型

make的变量有两种类型：经简单扩展的变量（simply expanded variable）以及经递归扩展的变量（recursively expanded variable）。你可以用:=赋值运算符来定义一个经简单扩展的变量（或称简单变量）：

```
MAKE_DEPEND := $(CC) -M
```

之所以称此变量为“经简单扩展”是因为，一旦make从*makefile*读进该变量的定义语句，赋值运算符的右边部分会立刻被扩展。赋值运算符的右边部分只要出现make变量的引用就会被扩展，而扩展后所产生的文本则会被存储成该变量的值。此行为跟大多数的程序和脚本语言相同。举例来说，此变量被扩展之后一般会变成下面这样：

```
gcc -M
```

然而，如果上面的CC变量尚未定义，则此变量被扩展后一般会变成这样：

```
<space>-M
```

$(CC)会被扩展成它的值（并未包含任何字符），也就是空无一物（即空值）。变量没有定义并不算错误。事实上，此特性相当有用。大多数的隐含命令（implicit command）都

注1：df命令会返回每个已挂载的文件系统的统计信息，包括文件系统的容量以及用量。通过第一个参数让df输出特定文件系统的统计信息。此输出的第一行是一串栏标题（column title）。接着由awk读取此输出，只查看第二行。此输出的第四行就是尚有多少空间可用的统计信息（以块为单位）。

会包含未定义的变量，以作为用户自定义变量的占位符（place holder）。如果用户并未自定义该变量，它就会变成空无一物。现在注意前导的空格。make首先会分析赋值运算符右边的部分，也就是$(CC) -M这个字符串。当变量引用被扩展成空无一物时，make不会重新扫描该值以及删除前导的空格，于是前导的空格就被保留了下来。

第二种变量类型称为经递归扩展的变量。你可以用=赋值运算符来定义一个经递归扩展的变量（或称递归变量）：

```
MAKE_DEPEND = $(CC) -M
```

之所以称此变量为"经递归扩展"是因为，make只会读进赋值运算符右边的部分，并将之存储成该变量的值，但不会进行任何扩展的动作，扩展的动作会被延迟到该变量被使用的时候才进行。将此变量称为延后扩展的变量（lazily expanded variable）或许比较恰当，因为扩展的动作会延迟到该变量实际被使用的时候才进行。这种扩展方式将会导致令人意外的结果，即变量的值可能会变得混乱：

```
MAKE_DEPEND = $(CC) -M
...
# 稍后
CC = gcc
```

这样，当MAKE_DEPEND被使用的时候，即使CC并未定义，MAKE_DEPEND在脚本中的值也会被扩展成gcc -M。

事实上，对递归变量所进行的并非真的是延后赋值的动作（至少不是一般的延后赋值动作）。每当递归变量被使用时，make就会对它的右边部分进行重新求值的动作。如果变量被定义成简单的常数，比如前面的MAKE_DEPEND，做此区别是毫无意义的，因为右边部分的变量也都是简单的常数。但试想，如果右边部分的某个变量被用来代表一个所要运行的程序，例如date，那么每当递归变量被扩展，date程序就会被运行，而且每次变量扩展后所产生的值也不一样（假设date的每次运行前后间隔至少一秒）。有的时候这个特性可能非常有用，有的时候这个特性可能非常烦人！

其他的赋值类型

我们在前面的范例中看到了两种赋值类型，其中=用来创建递归变量，而:=用来创建简单变量。此外，make还另外提供了两种赋值运算符：?=和+=。

?=运算符称为附带条件的变量赋值运算符（conditional variable assignment operator）。这相当冗长难念，所以我们会把它简称为条件赋值（conditional assignment）。此运算符只会在变量的值尚不存在的状况下进行变量要求赋值的动作。

```
# 将所产生的每个文件放到 $(PROJECT_DIR)/out 目录中。
OUTPUT_DIR ?= $(PROJECT_DIR)/out
```

此处，我们只会在输出目录变量OUTPUT_DIR的值尚不存在的状况下对它进行赋值的动作。这个功能可以跟环境变量有很好的交互。稍后我们将会在“变量来自何处”一节中探讨这个议题。

+=运算符通常被称为附加运算符（append operator）。正如其名，此运算符会将文本附加到变量里。这似乎没有什么特别的，但是当递归变量被使用时，它却是一个重要的特征。尤其是，赋值运算符右边部分的值会在“不影响变量中原有值的状况下”被附加到变量里。你可能会说：“这有什么大不了的，'附加'的功能不就是在做这件事吗？”。没错，不过少安毋躁，事实会有细微的差异。

对简单变量进行附加的动作，事情就会变得更加明显。+=运算符可以被实现成这样：

```
simple := $(simple) new stuff
```

因为简单变量中的值会被立即扩展，所以make会扩展$(simple)，附加因此而产生的文本，最后进行赋值的动作。但是递归变量会导致一个问题。如果将+=运算符实现成下面这样，是不允许的。

```
recursive = $(recursive) new stuff
```

这是一个错误，因为make没有办法妥善地加以处理。如果make存储recursive当前的定义加上new stuff，则make就不能在运行时再次扩展它。此外，试图扩展一个自我引用的递归变量将会产生一个无限循环。

```
$ make
makefile:2: *** Recursive variable `recursive' references itself (eventually). Stop.
```

所以，+=被特别实现成可将文本附加到递归变量中并做正确的事。此运算符对于想将所收集到的值递增给变量的人来说特别有用。

宏

变量适合用来存储单行形式的值，可是对于多行形式的值，例如命令脚本，如果我们想在不同的地方执行它，该怎么办？例如，下面这个从Java类文件创建Java程序库（Java archive或简称*jar*）的命令序列：

```
echo Creating $@...
$(RM) $(TMP_JAR_DIR)
$(MKDIR) $(TMP_JAR_DIR)
$(CP) -r $^ $(TMP_JAR_DIR)
cd $(TMP_JAR_DIR) && $(JAR) $(JARFLAGS) $@
```

```
$(JAR) -ufm $@ $(MANIFEST)
$(RM) $(TMP_JAR_DIR)
```

像这样冗长的命令序列，我们都喜欢在开始的地方输出一个简短的信息，这样可让make的输出较容易阅读。这个信息之后，我们会把类文件收集到一个新的临时目录中。所以，如果该临时目录原本就存在的话就会被我们删除（注2），然后我们会创建一个新的临时目录。接下来我们会把必要条件文件（以及它们的所有子目录）复制到该临时目录中并切换到该临时目录，再创建jar并以工作目标作为其文件名。我们还会把清单文件（manifest file）加入jar并且在最后做清理的工作。毫无疑问，我们并不想让这个命令序列重复出现在*makefile*文件中，因为这会给将来的维护带来问题。虽然我们可以把这个命令序列全都塞进一个递归变量，不过这会给维护造成麻烦，而且当make输出命令行（整个命令序列会被输出成一大串文本行）时，这会造成make的输出难以阅读。

在GNU make中，我们可以通过define指令以创建"封装命令序列"（canned sequence）的方式来解决此问题。使用"封装命令序列"这个术语或许大家会有些不习惯，所以我们将会称它为宏（macro）。在make中，宏只是用来定义变量的另一种方式，此变量还可以包含内置的换行符号（embedded newlines）！GNU make的在线使用手册似乎把变量（variable）和宏（macro）这两个词混用了。在本书中，我们将会使用"宏"这个词来指称由define指令所定义的变量，而"变量"这个词仅用来指称由赋值运算符所定义的变量。

```
define create-jar
 @echo Creating $@...
 $(RM) $(TMP_JAR_DIR)
 $(MKDIR) $(TMP_JAR_DIR)
 $(CP) -r $^ $(TMP_JAR_DIR)
 cd $(TMP_JAR_DIR) && $(JAR) $(JARFLAGS) $@ .
 $(JAR) -ufm $@ $(MANIFEST)
 $(RM) $(TMP_JAR_DIR)
endef
```

define指令后面跟着变量名称以及一个换行符号。变量的主体包含了所有的命令序列（每一行命令都必须前置一个跳格符号）直到endef关键字出现为止，endef关键字必须自成一行。一个由define创建的变量，就像任何其他的变量一样，会被扩展许多次，除非它被使用在命令脚本的语境中。下面是宏的使用范例：

```
$(UI_JAR): $(UI_CLASSES)
        $(create-jar)
```

请注意，我们为echo命令前置了一个@字符。当执行命令脚本时，前置@字符的命令

注2：为了得到最好的结果，应该将RM变量定义成rm -rf。事实上，RM变量的默认值是rm -f，虽然比较安全可是不好使用。此外，MKDIR应该定义成mkdir -p。

行不会被make输出。因此，当我们运行make时，它不会输出echo命令本身，只会输出该命令的输出。如果在宏内部使用@前缀，这个前缀字符（prefix character）只会影响使用到它的命令行。然而，如果将这个前缀字符用在宏引用（macro reference）上，则整个宏主体（macro body）都会被隐藏起来：

```
$(UI_JAR): $(UI_CLASSES)
        @$(create-jar)
```

当make运行时只会显示：

```
$ make
Creating ui.jar...
```

稍后我们会在“命令修饰符”一节中对@的使用做更进一步的讨论。

何时扩展变量

稍早我们看到了变量扩展过程的细微差异，这多半取决于变量之前的定义方式以及定义的位置。即使make无法找到任何错误，获得预期以外的结果仍是常有的事。所以，你可能想知道，扩展变量的规则是什么？这些规则真的有用吗？

当make运行时，它会以两个阶段来完成它的工作。第一个阶段，make会读进*makefile*以及被引入的任何其他*makefile*。这个时候，其中所定义的变量和规则会被加载进make的内部数据库，而且依存图也会被建立起来。第二个阶段，make会分析依存图并且判断需要更新的工作目标，然后执行脚本以完成所需要的更新动作。

当make在处理递归变量或define指令的时候，会将变量里的每一行或宏的主体存储起来，包括换行符号，但不会予以扩展。宏定义里的最后一个换行符号并不会被存储成宏的一部分；否则，宏被扩展时make会读进一个额外的换行符号。

当宏被扩展时，make会立即扫描被扩展的文本中是否存在宏或变量的引用，如果存在就予以扩展，如此递归进行下去。如果宏是在命令脚本的语境中被扩展的，则宏主体的每一行都会被插入一个前导的跳格符。

下面是用来处理“*makefile*中的元素何时被扩展”的准则：

- 对于变量赋值（variable assignment），make会在第一阶段读进该行时，立即扩展赋值运算符左边的部分。
- =和?=的右边部分会被延后到它们被使用的时候扩展，并且在第二阶段进行。
- :=的右边部分会被立即扩展。

- 如果+=的左边部分原本被定义成一个简单变量，+=的右边部分就会被立即扩展，否则，它的求值动作会被延后。
- 对于宏定义（使用define指令），宏的变量名称会被立即扩展，宏的主体会被延后到被使用的时候扩展。
- 对于规则，工作目标和必要条件总是会被立即扩展，然而命令总是会延后扩展。

表3-1列出了当变量被扩展时会发生什么事。

表3-1：立即和延后扩展的规则

定义	何时扩展a	何时扩展b
a = b	立即	延后
a ?= b	立即	延后
a := b	立即	立即
a += b	立即	延后或立即
define a b... b... b... endef	立即	延后

一个通则是：总是先定义变量和宏，然后再使用它们。尤其是，在工作目标或必要条件中使用变量时，就需要在使用变量之前先予以定义。

举例说明，你会更清楚。假设我们要定义一个free-space宏。接下来我们会一次说明一个部分，最后再把它们组合在一起。

```
BIN    := /usr/bin
PRINTF := $(BIN)/printf
DF     := $(BIN)/df
AWK    := $(BIN)/awk
```

我们定义了三个变量，用来保存宏中所用到的程序的名称。为了避免重复，我们把*bin*目录抽离而成为第四个变量。当make读进这四个变量的定义时，它们的右边部分都会被立即扩展，因为它们都是简单变量。BIN变量会被定义在其他三个变量之前，所以它的值会被塞进其他三个变量的值里。

接着，我们定义了free-space宏。

```
define free-space
  $(PRINTF) "Free disk space "
  $(DF)    | $(AWK) 'NR == 2 { print $$4 }'
endef
```

紧跟在define指令之后的变量名称会被立即扩展，但就此例而言，并不需要进行扩展的动作。当make读进宏的主体时会予以储存，但不会将之扩展。

最后，我们会在一个规则中使用free-space这个宏。

```
OUTPUT_DIR := /tmp
$(OUTPUT_DIR)/very_big_file:
        $(free-space)
```

当*$(OUTPUT_DIR)/very_big_file*规则被读取时，工作目标和必要条件中所用到所有变量都会被立即扩展。其中，$(OUTPUT_DIR)会被扩展成*/tmp*，所以整个工作目标会变成*/tmp/very_big_file*。接着，make会读取这个工作目标的命令脚本，它会将前置跳格符的文本行视为命令行，加以读取并将之存储起来，但是不会进行扩展的动作。

下面就是以上所提到的*makefile*的完整范例。此处刻意将*makefile*的构成元素的次序打乱，以展示make的求值算法（evaluation algorithm）。

```
OUTPUT_DIR := /tmp

$(OUTPUT_DIR)/very_big_file:
        $(free-space)
define free-space
  $(PRINTF) "Free disk space "
  $(DF) . | $(AWK) 'NR == 2 { print $$4 }'
endef

BIN    := /usr/bin
PRINTF := $(BIN)/printf
DF     := $(BIN)/df
AWK    := $(BIN)/awk
```

请注意，尽管*makefile*中各个构成元素的次序似乎搞反了，不过运行起来却毫无问题。这就是递归变量出人意料的效果之一。虽然这相当有用，但这同时也令人相当困惑。此*makefile*之所以能够运作无误，是因为脚本和宏主体的扩展动作将会被延后到它们实际被使用的时候。因此，它们出现在文件中的先后次序对*makefile*的运行毫无影响。

第二阶段进行的时候，也就是make读进*makefile*之后，make会针对每项规则寻找工作目标、进行依存分析以及执行动作。此处只找到$(OUTPUT_DIR)/very_big_file这个工作目标，因为此工作目标并未依存于任何必要条件，所以make会直接执行相应的动作（假定工作目标所代表的文件不存在）。make所要执行的动作就是$(free-space)这个命令脚本。所以make会将之扩展，整个规则会变成下面这样：

```
/tmp/very_big_file:
        /usr/bin/printf "Free disk space "
        /usr/bin/df . | /usr/bin/awk 'NR == 2 { print $$4 }'
```

一旦所有的变量都被扩展之后，make会每次执行命令脚本里的一个命令。

事实上，*makefile* 文件中有两处的次序很重要。正如稍早所说，工作目标 *$(OUTPUT_DIR)/very_big_file* 会被立即扩展。如果变量OUTPUT_DIR的定义被放在规则之后，那么工作目标扩展之后会变成*/very_big_file*，这或许不是用户想要的结果。同样地，如果BIN的定义被放在AWK的后面，那么另三个变量将依次会被扩展成*/printf*、*/df*和*/awk*，因为:= 会使得make对赋值运算符的右边部分立即进行求值的动作。然而，这个时候，我们可以通过将:= 替换成 = 的方式，将PRINTF、DF和AWK变更为递归变量来避免此问题。

最后，请注意，将OUTPUT_DIR和BIN的定义变更为递归变量，并不会对前面的次序问题有任何的影响。重点在于 *$(OUTPUT_DIR)/very_big_file* 工作目标以及PRINTF、DF和AWK的右边部分是在何时被扩展的，因为它们会被立即扩展，所以它们所引用的变量必须在事先被定义好。

工作目标与模式的专属变量

在 *makefile* 运行期间，变量通常只有一个值。对需要经过两个处理阶段的 *makefile* 来说是这样没错。第一个阶段，make读进 *makefile* 之后，会对变量进行赋值和扩展的动作并建立依存图。第二阶段，make会分析以及遍历依存图。所以，等到make执行命令脚本的时候，所有变量都已经处理完毕了。但是如果我们想为特定的规则或模式重新定义变量，该怎么办？

现在，我们想要编译一个需要额外命令行选项 -DUSE_NEW_MALLOC=1 的文件，但是其他的编译项目并不需要这个额外的命令行选项：

```
gui.o: gui.h
        $(COMPILE.c) -DUSE_NEW_MALLOC=1 $(OUTPUT_OPTION) $<
```

如上所示，我们解决此问题的办法是复制编译命令脚本以及为它加入这个必要的选项。如果这个规则有任何变动，或是我们选择以自定义的模式规则来取代这个内置规则，那么我们就必须对这个部分进行更新，不过我们可能会忘掉此事。其次，如果有许多文件需要经过类似的特别处理（假设有100个文件），整个工作马上就会变得冗长乏味、易于出错。

为解决此类问题，make提供了工作目标的专属变量。这些变量的定义会附加在工作目标之上，且只有在该工作目标以及相应的任何必要条件被处理的时候，它们才会发生作用。通过使用此功能，我们可以把前面的例子改写成这个样子：

```
gui.o: CPPFLAGS += -DUSE_NEW_MALLOC=1
gui.o: gui.h
        $(COMPILE.c) $(OUTPUT_OPTION) $<
```

变量CPPFLAGS内置在默认的C编译规则里，用来保存供C预处理器（preprocessor）使用的选项。通过使用+=附加运算符，我们可以把这个新的选项附加到任何已存在的值里。现在可以把编译命令脚本整个移除：

```
gui.o: CPPFLAGS += -DUSE_NEW_MALLOC=1
gui.o: gui.h
```

当make处理*gui.o*这个工作目标的时候，CPPFLAGS的值除了包含它原有的内容，还会包含-DUSE_NEW_MALLOC=1。当make处理完*gui.o*这个工作目标之后，CPPFLAGS的值将会恢复它原有的内容。

工作目标的专属变量的语法如下所示：

```
target...: variable = value
target...: variable := value
target...: variable += value
target...: variable ?= value
```

如你所见，以上的语法只能用来定义工作目标的专属变量。此类变量在被赋值之前，并不需要事先存在。

此外，这类变量的赋值动作会延后到开始处理工作目标的时候进行。所以赋值运算符右边部分的值，可由另一个工作目标的专属变量来设定。同样地，此变量只有在必要条件的处理期间，才会发生作用。

变量来自何处

到目前为止，我们所看到的大部分变量都会被明确地定义在*makefile*文件中。其实变量可以更复杂。举例来说，我们曾看到过，变量被定义在make命令行上。事实上，make的变量可以有以下几个来源：

文件

当然，变量可以被定义在*makefile*中，或是被*makefile*引入（稍后我们将会说明include指令）。

命令行

你可以直接在make命令行上定义或重新定义变量：

```
$ make CFLAGS=-g CPPFLAGS='-DBSD -DDEBUG'
```

每个命令行参数中所包含的等号（=），都是一个变量赋运值算符。在命令行上，每个变量赋值运算符的右边部分必须是一个单独的shell参数。如果变量的值（或变量本身）包含空格，则必须为参数加上括号或是规避空格。

命令行上变量的赋值结果将会覆盖掉环境变量以及*makefile*文件中的赋值结果。你可以使用:=或=赋值运算符将命令行参数设定成简单或递归变量。此外，如果使用override指令，你还可以要求make采用*makefile*的赋值结果，而不要采用命令行的赋值结果。

```
# 使用big-endian对象，否则程序会无法运行!
override LDFLAGS = -EB
```

当然，你只应该在非常紧迫的状况下，忽略用户所要求的赋值动作（除非你想惹怒用户）。

环境

当make启动时，所有来自环境的变量都会被自动定义成make的变量。这些变量具有非常低的优先级，所以*makefile*文件或命令行参数的赋值结果将会覆盖掉环境变量的值。不过，你可以使用--environment-overrides（或-e）命令行选项，让环境变量覆盖掉相应的*makefile*变量。

当make被递归调用时，有若干来自上层make的变量会通过环境传递给下层的make。默认情况下，只有原先就来自环境的变量会被导出到下层的环境之中。不过，你只要使用export指令就可以让任何变量被导出到环境之中：

```
export CLASSPATH := $(HOME)/classes:$(PROJECT)/classes
SHELLOPTS = -x
export SHELLOPTS
```

要将所有变量全部导出，可以这么做：

```
export
```

请注意，即使这些变量的名称包含了无效的shell变量字符，make也会进行导出的动作。例如：

```
export valid-variable-in-make = Neat!
show-vars:
        env | grep '^valid-'
        valid_variable_in_shell=Great
        invalid-variable-in-shell=Sorry

$ make
env | grep '^valid-'
valid-variable-in-make=Neat!
valid_variable_in_shell=Great
invalid-variable-in-shell=Sorry
/bin/sh: line 1: invalid-variable-in-shell=Sorry: command not found
make: *** [show-vars] Error 127
```

通过导出valid-variable-in-make这个变量，一个“无效的”变量被创建了。虽然无法通过一般的shell语法来访问valid-variable-in-make变量，但是你可以耍些小手段，像是对环境变量运行grep。不过，这个变量将会被任何下层的make所继承。在该处，此变量不仅有效而且可供访问。我们将会在第二部分说明make的“递归”用法。

此外，你还可以通过如下的方式避免环境变量被导出到子进程：

```
unexport DISPLAY
```

指令export、unexport的作用和sh命令中的export、unset一样。

条件赋值运算符与环境变量的交互良好。假如你已经在*makefile*文件中设定了一个默认的输出目录，但是你希望用户轻易就能改写此默认值，此时条件赋值将会是最佳解决方案：

```
# 假设输出目录为 $(PROJECT_DIR)/out
OUTPUT_DIR ?= $(PROJECT_DIR)/out
```

这样，make只会在OUTPUT_DIR尚未定义的状况下进行赋值的动作。此外，使用如下的较冗长的方式也可以得到几乎一样的结果：

```
ifndef OUTPUT_DIR
  # 假设输出目录为$(PROJECT_DIR)/out
  OUTPUT_DIR = $(PROJECT_DIR)/out
Endif
```

其中的差别在于，如果变量的值已经设定，那么即使是空值，条件赋值运算符也会跳过赋值的动作，而运算符ifdef和ifndef只会测试“非空值”。因此，我们会使用条件运算符而不会使用ifdef来对OUTPUT_DIR=赋值。

切记，过多使用环境变量将会大大降低*makefile*的可移植性，因为其他用户不太可能会设定跟你完全一样的环境变量。事实上，我不太使用此功能就是因为这个原因。

自动创建

最后，make会在执行一个规则的命令脚本之前立刻创建自动变量。

传统上，环境变量可协助开发者管理机器之间的差异。常见的做法就是根据*makefile*文件中所引用的环境变量来建立开发环境（源文件树、二进制输出文件以及程序库）。*makefile*将会以环境变量指向每个目录树的根目录。如果能够从PROJECT_SRC变量引用源文件树、从PROJECT_BIN引用二进制输出文件以及从PROJECT_LIB引用程序库，那么开发者就可以依照需要将这些目录树放到适当的地方。

这么做会有一个潜在的问题：如果这些指向根目录的变量的值并未设定，将会出现错误。一个解决方案就是在*makefile*文件中以条件赋值运算符?=提供默认值：

```
PROJECT_SRC ?= /dev/$(USER)/src
PROJECT_BIN ?= $(patsubst %/src,%/bin,$(PROJECT_SRC))
PROJECT_LIB ?= /net/server/project/lib
```

使用这些变量来访问项目里的组件，就能够建立出一个可以适应不同机器的开发环境（我们将会在本书第二个部分看到更多例子）。然而，请小心，不要过多使用环境变量。通常，*makefile* 应该只需从开发者的环境中获得最起码的支持就能够运行，因此你只要提供合理的默认值以及检查重要的组件是否存在就行了。

条件指令与引入指令的处理

当make所读进的 *makefile* 使用了条件处理指令时，*makefile* 文件中有些部分会被省略，有些部分会被挑选出来。用来控制是否选择的条件具有各种形式，比如“是否已定义”或“是否等于”。下面是一个例子：

```
# COMSPEC只会在Windows上被定义
ifdef COMSPEC
  PATH_SEP := ;
  EXE_EXT  := .exe
else
  PATH_SEP := :
  EXE_EXT  :=
Endif
```

如果COMSPEC已定义，则会选择条件指令的第一个分支。

条件指令的基本语法如下所示：

```
if-condition
 text if the condition is true
endif
```

或：

```
if-condition
  text if the condition is true
else
  text if the condition is false
endif
```

if-condition 可以是以下之一：

```
ifdef  variable-name
ifndef variable-name
ifeq  test
ifneq test
```

进行ifdef/ifndef的测试时，不应该以$()括住variable-name。最后，test可以表示成下面这样：

```
"a" "b"
```

或：

```
(a,b)
```

其中，单引号或双引号可以交替使用（但是引号必须成对出现）。

条件处理指令可用在宏定义和命令脚本中，还可以放在*makefile*的顶层：

```
libGui.a: $(gui_objects)
        $(AR) $(ARFLAGS) $@ $<
    ifdef RANLIB
        $(RANLIB) $@
    endif
```

我喜欢缩排我的条件指令，但是草率的缩排动作可能会导致错误。在前面的例子中，条件指令被缩排了四个空格，而且其所括住的命令具有一个前导的跳格符（tab）。如果其所括住的命令并非以一个跳格符开头，make将不会把它视为命令；如果条件指令具有一个前导的跳格符，make会误以为“条件指令”就是“命令”而将之传递给subshell。

ifeq和ifneq条件指令可用来测试其参数是否相等。条件指令里空格的处理有些微妙。举例来说，如果参数采用小括号的形式，那么逗号之后的空格将会被忽略，除此之外所有其他的空格都是有意义的：

```
ifeq (a, a)
  # These are equal
endif

ifeq ( b, b )
  # These are not equal - ' b' != 'b '
endif
```

我比较喜欢使用等效的引号形式：

```
ifeq "a" "a"
  # These are equal
endif

ifeq 'b' 'b'
  # So are these
endif
```

即使如此，还是经常会发生“变量扩展后包含了非预期的空格符号”的状况。这可能引发一些问题，因为进行匹配时会将所有字符纳入考虑。为了创建更稳定的*makefile*，我会使用strip函数：

```
ifeq "$(strip $(OPTIONS)) "-d"
  COMPILATION_FLAGS += -DDEBUG
Endif
```

include 指令

我们已经在“自动产生依存关系”一节中介绍过 include 指令，现在让我们对它进行更深入的探讨。

一个 *makefile* 可以引入（include）其他文件。此功能常用来引入 make 头文件中所存放的共同的make的定义，或是用来自动产生依存关系的信息。include指令的用法如下所示：

```
include definitions.mk
```

你可以为这个指令提供任意多个文件、shell 通配符以及 make 变量。

引入文件与依存关系

当make看到 include 指令时，会事先对通配符以及变量引用进行扩展的动作，然后试着读进引入文件（include file）。如果这个文件存在，则整个处理过程会继续下去；然而，如果这个文件不存在，则make会汇报此问题并且继续读取其余的 *makefile*。当所有的读取动作皆已完成之后，make 会从规则数据库中找出任何可用来更新引入文件的规则。如果找到了一个相符的规则，make就会按照正常的步骤来更新工作目标。如果任何一个引入文件被规则更新，make接着会清除它的内部数据库并且重新读进整个*makefile*。如果完成读取、更新和重新读取的过程之后，仍有 include 指令因为文件不存在而执行失败，那么 make就会显示错误状态并终止执行。

我们可以在接下来的例子中看到此过程。使用warning这个内置函数，我们可以从make输出简单的信息（我们将会第四章详细说明此函数与其他函数）。下面就是我们的 *makefile* 范例：

```
# 这个简单的maketile引入了一个自动产生的文件
include foo.mk
$(warning Finished include)

foo.mk: bar.mk
        m4 --define=FILENAME=$@ bar.mk > $@
```

bar.mk 是这个引入文件的来源：

```
# bar.mk——当我在读取的时候汇报信息
$(warning Reading FILENAME)
```

对这个*makefile*运行make，将会看到如下的结果：

```
$ make
Makefile:2: foo.mk: No such file or directory
Makefile:3: Finished include
m4 --define=FILENAME=foo.mk bar.mk > foo.mk
foo.mk:2: Reading foo.mk
Makefile:3: Finished include
make: `foo.mk' is up to date.
```

第一行显示make并未找到引入文件，不过第二行显示make会继续读取以及执行*makefile*。完成读取动作后，make找到了一个用来创建引入文件*foo.mk*的规则，所以它就会执行此规则。然后make会重新开始整个过程，这次读取引入文件不会遇到任何困难。

现在是让你知道"make也可以把*makefile*本身作为一个可能的工作目标"这件事的好时机。当make读进整个*makefile*之后，make将会试着寻找可用来重建当前所执行的*makefile*的规则。如果找到了，make将会处理此规则，然后检查*makefile*是否已经更新。如果已经更新，make将会清除它的内部状态，重新读进此*makefile*，重新完成整个分析动作。你可以在下面这个无聊的例子里看到一个基于此行为的无限循环：

```
.PHONY: dummy
makefile: dummy
        touch $@
```

当make执行此*makefile*时，它将会看到*makefile*尚未更新（因为.PHONY工作目标*dummy*尚未更新），所以它会执行touch命令，这么做会更新*makefile*的时间戳。然后make会重新读进*makefile*，并且发现*makefile*尚未更新……了解了吧。

make会到何处寻找引入文件？显然，如果include的参数是一个绝对的文件引用，则make会直接读进该文件，如果这是一个相对的文件引用，则make会先到当前的工作目录中查找该文件。如果make无法找到该文件，它接着会到你在命令行上以--include-dir（或-I）选项所指定的目录继续查找。如果还找不到，make会到自己被编译时所使用的搜索路径进行查找，比如*/usr/local/include*、*/usr/gnu/include*、*/usr/include*。你的搜索路径可能跟此处所提到的不同，这取决于make的编译方式。

如果make没有找到引入文件，而且没有找到可用来创建该引入文件的规则，make将会汇报错误信息并且结束执行。如果想让make忽略无法加载的引入文件，可以为include指令前置一个破折号：

```
-include i-may-not-exist.mk
```

为了兼容于其他版本的make，GNU make为-include提供了sinclude这个别名。

标准的 make 变量

除了自动变量，make还会为“自己的状态以及内置规则的定义”提供变量，以便对外提供相关信息：

MAKE_VERSION

GNU make的版本编号。编写本书时，此值为3.80，如果是CVS仓库中的版本，此值为3.81rc1。

make前一个版本的编号是3.79.1，这个版本不支持eval和value函数（以及其他东西），不过仍旧十分常见。所以，当我在编写需要此功能的*makefile*时，我都会使用此变量来查看现在所运行的make的版本编号。稍后我们将会在“流程控制”一节中看到一个例子。

CURDIR

正在执行make进程的当前工作目录（current working directory，简称cwd）。此变量的值将会是shell变量PWD的值（代表你是从哪个目录运行make程序的），除非make在运行时用到了 --directory（或-C）选项。--directory选项会使得make在搜索任何*makefile*之前变更到不同的目录。这个选项的完整形式为

--directory=*directory-name*或-C *directory-name*。如果你使用的是--directory的形式，CURDIR将会包含--include-dir的目录参数。

编写程序代码的时候，我通常会从emacs来调用make。例如，我有一个项目是用Java写成的，而且在顶层目录使用了一个*makefile*（此目录不一定要包含程序代码）。此时，如果使用--directory选项，不管我是从源代码树中的哪个目录来调用make的，我仍然能够访问此*makefile*。在*makefile*文件中，所有路径都应该被设定成相对于*makefile*所在的目录。需要使用绝对路径的时候则可以通过CURDIR进行访问。

MAKEFILE_LIST

make所读进的各个*makefile*文件的名称所构成的列表，包括默认的*makefile*以及命令行或include指令所指定的*makefile*。在每个*makefile*被读进make之前，其文件名会被附加到MAKEFILE_LIST变量里。所以，任何一个*makefile*总是可以查看此列表的最后一项来判断自己的文件名。

MAKECMDGOALS

对当前所运行的make而言，make运行时命令行上指定了哪些工作目标。此变量并不包含命令行选项或变量的赋值。例如：

```
$ make -f- FOO=bar -k goal <<< 'goal:;# $(MAKECMDGOALS)'
# goal
```

这个例子要了一个花招，让make以-f（或--file）选项从*stdin*（标准输入）读进*makefile*。此处还使用了bash的当前字符串的<<<语法（注3），将一个命令行字符串重定向至*stdin*。此*makefile*包含了默认目标goal，命令脚本被放在同一行，不过以分号隔开工作目标与命令脚本。这个命令脚本的内容只有一行：

```
# $(MAKECMDGOALS)
```

当工作目标需要特别的处理时，通常会使用MAKECMDGOALS。常见的例子是"clean"工作目标。当用户以"clean"调用make时，make不应该像平常那样进行由include指令所触发的依存文件产生动作（参见"自动产生依存关系"一节）。要避免此事，可使用ifneq和MAKECMDGOALS：

```
ifneq "$(MAKECMDGOALS)" "clean"
  -include $(subst .xml,.d,$(xml_srcl))
endif
```

.VARIABLES

到目前为止，make从各个*makefile*文件所读进的变量的名称所构成的列表，不含工作目标的专属变量。此变量仅供读取，对它所进行的任何赋值动作都会被忽略掉。

```
list:
        @echo "$(.VARIABLES)" | tr ' ' '\015' | grep MAKE
$ make
MAKEFLAGS
MAKEFILE_LIST
MAKEFILES
```

如你所见，变量还可用来自定义make的隐含规则（implicit rule）。C/C++的隐含规则是将变量应用在程序语言中的典型形式。从图3-1中可以看到文件类型转换过程中所用到的变量。

这些变量的基本形式为*ACTION.suffix*。*ACTION*的值可以是COMPILE（表示创建目标文件）、LINK（表示创建可执行文件）或是其他特殊的操作，像PREPROCESS、YACC或LEX（分别表示运行C预处理器、yacc或lex）。至于*suffix*则代表源文件的类型。

对C++而言，使用这些变量的标准路径会用到两个规则：第一个是将C++源文件编译成目标文件，另一个是将目标文件链接成可执行文件。

```
%.o: %.C
        $(COMPILE.C) $(OUTPUT_OPTION) $<

%: %.o
        $(LINK.o) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

注3：想要在bash以外的shell中运行这个例子的人，可以这么做：

```
$ echo 'goal:;# $(MAKECMDGOALS)' | make -f- FOO=bar -k goal
```

图 3-1：c/c++编译过程中所用到的变量

第一个规则会用到以下变量定义：

```
COMPILE.C      = $(COMPILE.cc)
COMPILE.cc     = $(CXX) $(CXXFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
CXX            = g++
OUTPUT_OPTION = -o $@
```

GNU make允许你使用扩展名.C或.cc来代表C++源文件。CXX变量用来指定所要使用的C++编译器，默认值为g++。变量CXXFLAGS、CPPFLAGS以及TARGET_ARCH并不具备默认值，它们主要是供终端用户用来自定义编译过程的。这三个变量分别用来保存C++编译器标记，C预处理器标记以及结构特有的编译选项，而OUTPUT_OPTION则包含了输出文件选项。

链接规则比较简单：

```
LINK.o = $(CC) $(LDFLAGS) $(TARGET_ARCH)
CC     = gcc
```

此规则将会使用C编译器将目标文件组合成可执行文件。C编译器的默认值为gcc。LDFLAGS和TARGET_ARCH并不具备默认值。LDFLAGS用来保存链接选项，像-L标记，而LOADLIBES和LDLIBS变量中则包含了所要链接的程序库列表。这两个变量被同时纳入主要是为了具有较好的可移植性。

以上就是make变量的快速浏览（quick tour）。事实上还有很多，不过这些内容已经能够让你大致了解变量与规则的整合方式。另一群变量与TEX和它的一组规则有关。递归式make是变量所提供的另一种功能，我们将会在第六章探讨此议题。

第四章

函数

GNU make 支持内置函数以及用户自定义函数。函数调用（function invocation）看起来非常像变量引用(variable reference)，不过前者包含了一个或多个被逗号隔开的参数。大部分内置数扩展后多半会被赋值给一个变量或是传递给一个subshell。用户自定义函数则存储在变量或宏中，而且会接收调用者（caller）传来的一个或多个参数。

用户自定义函数

用户自定义函数能够将命令序列存储在变量里，让我们得以在*makefile*中使用各种应用程序。在下面的例子中我们可以看到一个用来终止进程的宏（注1）：

```
AWK  := awk
KILL := kill

# $(kill-acroread)
define kill-acroread
  @ ps -W |
  $(AWK) 'BEGIN       { FIELDWIDTHS = "9 47 100" }      \
          /AcroRd32/ {                                   \
                       print "Killing " $$3;             \
                       system( "$(KILL) -f " $$1 )       \
                     }'
endef
```

注1：　你可能会觉得奇怪：为什么要在*makefile*里做这件事？嗯，在Windows上，一个文件被打开时会被锁住以避免其他的进程对它进行写入的动作。在本书编写期间，PDF文件常会被Acrobat Reader锁住以避免我的*makefile*更新该PDF文件。所以我才会为一些工作目标加入此命令，以便在我想要更新被锁住的文件之前，先终止运行中的Acrobat Reader。

(这个宏在编写时使用的是Cygwin工具（注2），因此我们所查找的程序名称以及ps和kill的选项都不是标准Unix的形式。）为了终止一个进程，我们把ps的输出使用管道重定向至awk。接着，awk脚本会以Acrobat Reader在Windows上的程序名称来查找它，如果它正在运行，就会终止相应的进程。我们在awk脚本中使用了FIELDWIDTHS这个功能，好让程序名称以及它的所有参数能够被作为单一字段来处理。这可让你正确输出完整的程序名称和参数，即使其中包含内置的空格。在awk中，字段引用会被依次写成$1，$2等。如果我们没有为awk脚本加上引号，那么这些东西将会被视为make变量。我们可以通过“以额外的美元符号规避$n中的美元符号（也就是$$n）”的方式要求make将$n引用传递给awk，而不要将它本身扩展。make将会看到两个美元符号，然后将之缩减成一个美元符号并把它传递给subshell。

宏很好用对于经常要用的脚本，可以使用define指令以避免重复。但是这么做并不完美：如果我们要终止Acrobat Reader以外的进程，该怎么办？我们还需要用重复的脚本来定义另一个宏？不！

你可以传递参数给变量和宏，这样每次的扩展结果都可以不一样。宏的参数在宏定义的主体中可依次以$1、$2等进行引用。要让我们的kill-acroread函数使用参数，只需要加入一个搜索参数就行了：

```
AWK          := awk
KILL         := kill
KILL_FLAGS   := -f
PS           := ps
PS_FLAGS     := -W
PS_FIELDS    := "9 47 100"

# $(call kill-program,awk-pattern)
define kill-program
  @ $(PS) $(PS_FLAGS) |                                    \
  $(AWK) 'BEGIN { FIELDWIDTHS = $(PS_FIELDS) }             \
          /$1/  {                                          \
                  print "Killing " $$3;                    \
                  system( "$(KILL) $(KILL_FLAGS) " $$1 ) \
                }'
endef
```

我们使用了一个参数引用$1来取代awk的搜索模式/AcroRd32/。注意宏参数$1与awk字段参数$$1之间的细微差异，记住“哪个程序是变量引用的接受者”是一件非常

注2：Cygwin工具包含了许多可以在Windows上使用的标准GNU和Linux程序，像编译器套件（compiler suite）、X11R6、ssh甚至是inetd。之所以能够这样，是因为在它的兼容程序库里包含了由Win32 API函数实现出来的Unix系统调用。这是一项惊人的成就，建议使用看看。你可以到http://www.cygwin.com下载此工具。

重要的事。这个函数还可以更完美，我们可以为它取个适当的名字，并使用变量来取代Cygwin特有的一些值。现在这个用来终止程序的宏应该具有很好的可移植性。

所以让我在如下的过程中使用此函数：

```
FOP         := org.apache.fop.apps.Fop
FOP_FLAGS   := -q
FOP_OUTPUT  := > /dev/null
%.pdf: %.fo
        $(call kill-program,AcroRd32)
        $(JAVA) $(FOP) $(FOP_FLAGS) $< $@ $(FOP_OUTPUT)
```

这个模式规则首先会终止*Acrobat*进程，如果它正在运行的话，会使用Fop处理器（*http://xml.apache.org/fop*）把*fo*文件转换成*pdf*文件。其中用来扩展变量或宏的语法如下所示：

```
$(call macro-name[, param1 ...])
```

call是一个内置于make的函数，call会扩展它的第一个参数并把其余参数依次替换到出现$1，$2……的地方（事实上，单就控制转换的意义来说，call并不会真的去"调用"它的宏参数，它所进行的是一种特殊的宏扩展动作）。macro-name可以是任何宏或变量（不要忘了，宏只是一个允许内嵌换行符（embedded newline）的变量）的名称。宏或变量的值中甚至不必包含任何的$n引用，如果是这样的话，使用call几乎没什么意义。macro-name之后是宏的参数，并以逗号为分隔符。

请注意，call的第一个参数是一个非扩展式变量名称（即它并非以一个美元符号开始），这种状况并不常见。origin是另一个可以接受非扩展式变量名称的内置函数。如果你为call的第一个参数加上一个美元符号以及圆括号，该参数就会像变量一样被扩展，它的值会被传递给call。

call的参数检查机制非常简单。你可以为call指定任意多个参数。如果一个宏引用了一个参数$n，但是调用它的call实例并未指定相应的参数，那么该变量就会变成空值。如果call实例所指定的参数比宏的$n引用还多，那么在宏中并不会扩展额外的参数。

如果你是从一个宏来调用另一个宏的，最好能够知道make 3.80的一个奇怪特性：call函数所定义的参数，在扩展动作进行期间，会被视为一般的make变量。如果一个宏调用了另一个宏，被调用者（child）将可以在它的宏被扩展期间看到调用者（parent）的参数：

```
define parent
  echo "parent has two parameters: $1, $2"
  $(call child,$1)
endef
```

```
define child
  echo "child has one parameter: $1"
  echo "but child can also see parent's second parameter: $2!"
endef

scoping_issue:
        @$(call parent,one,two)
```

对此 *makefile* 运行 make 就可以看到这个宏实现具有可见性的问题（scoping issue）。

```
$ make
parent has two parameters: one, two
child has one parameter: one
but child can also see parent's second parameter: two!
```

这个问题在 3.81 版中已经解决，所以 child 里的 $2 会被扩展成空值。

本书接下来将会以很大的篇幅来探讨用户自定义函数，不过在此之前我们必须具备更多的背景知识。

内置函数

当你开始以 make 变量取代简单的常数之后，你将会发现自己越来越想要以复杂的方式来操作变量以及它们的内容。嗯，你当然可以。GNU make 提供了不少内置函数可让你用来操作变量和它们的内容。make的函数可分类成：字符串操作、文件名操作、流程控制、用户自定义函数以及若干（重要的）杂项函数。

但首先你应该多知道一点函数的语法。所有的函数都会具有如下的形式：

```
$(function-name arg1[, argn])
```

$(之后是内置函数的名称，接着是函数的参数。第一个参数的前导空格会被删除，但是后续的任何参数若包含前导（当然也包括内置的和结尾的）空格则都会被保留下来。

函数的参数是以逗号为分隔符，所以只有一个参数的函数并不需要使用逗号，具有两个参数的函数需要使用一个逗号，以此类推。许多只接受一个参数的函数会把它的参数视为一串以空格隔开的单词。对这些函数而言，它们会以空格作为分隔符。

我喜欢使用空格，因为它可让我的程序代码具有较好的可读性并且较容易维护。所以只要"可以这么做"，我就会以空格为分隔符。然而，有时空格在参数列表或变量定义中可能会让make无法正确解读我们所做的描述。当此事发生时，你没有多少选择，你只能删除有问题的空格。我们已经在第三章看过一个例子，在该例子中，一个结尾的空格被意外插入到 grep 命令的搜索模式里。后续还会看到更多的例子，届时我们会指出空格所引发的问题。

make有许多函数允许你以模式为参数，此模式的语法如同模式规则中所使用的模式（参见“模式规则”一节）。模式中可以包含一个%符号，以及前导或跟在后面的字符（或是同时包含两者）。%字符代表零个或多个任何类型的字符。进行工作目标字符串匹配时，此模式必须匹配整个字符串，而不只是字符串的子集。稍后我们将会举例加以说明。在模式中，%字符是一个选项，通常你可以适时予以省略。

字符串函数

make的内置函数大部分都可以操作两种形式的文本，不过有些函数具有特别强的字符串处理能力，这就是此处所要探讨的内容。

在make中常见的字符串操作就是从一份文件列表中选出一组文件，shell脚本中之所以常会用到grep就是这个原因。在make中我们有filter、filter-out和findstring等函数可用。

$(filter pattern ...,text)

filter函数会将text视为一系列被空格隔开的单词，与pattern比较之后，接着会返回相符者。举例来说，为了建立用户界面（user-interface）的程序库，我们可能只想从*ui*子目录中选出目标文件。接下来，我们将会从文件名列表中取出开头为*ui/*而结尾为*.o*的文件名。%字符将会与其中任意多个字符匹配：

```
$(ui_library): $(filter ui/%.o,$(objects))
        $(AR) $(ARFLAGS) $@ $^
```

filter还可以接受多个（被空格隔开的）格式。正如之前所说，格式必须比较整个单词，才可以将相符的单词放到输出列表中。所以，如下的例子：

```
words := he the hen other the%
get-the:
        @echo he matches: $(filter he, $(words))
        @echo %he matches: $(filter %he, $(words))
        @echo he% matches: $(filter he%, $(words))
        @echo %he% matches: $(filter %he%, $(words))
```

运行之后会产生这样的输出：

```
$ make
he matches: he
%he matches: he the
he% matches: he hen
%he% matches: the%
```

如你所见，第一个模式只会匹配于单词he，因为该模式必须匹配整个单词而不是单词的一部分。其他的模式会匹配于he和其他单词（he在整个单词中的右边、左边或中间）。

模式中只可以包含一个%字符。如果模式包含了额外的%字符，那么第一个%字符除外，其余%字符都会被视为文字字符（literal character）。

filter无法在单词中匹配子字符串而且只接受一个通配符，这看起来或许有些奇怪。当你想要使用的功能不存在时，你可能有得忙了。不过，你可以通过“循环”和“条件测试”来实现类似的功能。稍后我们将会示范给你看。

$(filter-out pattern...,text)

filter-out函数所做的事刚好跟filter相反，用来选出与模式不相符的每个单词。所以，下面的例子可以从文件名列表中选出所有非C头文件的文件。

```
all_source := count_words.c counter.c lexer.l counter.h lexer.h
to_compile := $(filter-out %.h, $(all_source))
```

$(findstring string...,text)

此函数将会在text里搜索string。如果该字符串被找到了，此函数就会返回string；否则，它会返回空值。

乍看之下，此函数有点像子字符串搜索函数grep，就像当初我们也以为filter就是这样，但并非如此。首先，而且最重要的是，此函数所返回的只是“搜索字符串”，而不是它所找到的包含该“搜索字符串”的单词。其次，“搜索字符串”无法包含通配符（就算你在“搜索字符串”中使用%字符，也是按照字面进行匹配）。此函数通常会跟稍后将会讨论的if函数一起使用。不过，我发现有一个状况适合单独使用findstring。

假设你现在有几个平行的目录树，像引用源文件（reference source）、沙箱源文件（sandbox source）、调试二进制文件（debugging binary）以及经优化的二进制文件（optimized binary），你想要试着从当前目录（current directory）中找出你现在位于哪个目录树。下面就是作此判断的*makefile*结构：

```
find-tree:
        # PWD = $(PWD)
        # $(findstring /test/book/admin,$(PWD))
        # $(findstring /test/book/bin,$(PWD))
        # $(findstring /test/book/dblite_0.5,$(PWD))
        # $(findstring /test/book/examples,$(PWD))
        # $(findstring /test/book/out,$(PWD))
        # $(findstring /test/book/text,$(PWD))
```

（命令脚本中每行命令的开头是一个跳格符与一个shell的注释字符，所以每行命令就像其他命令那样，将会在自己的subshell中被执行。bash以及许多其他与Bourne类似的shell都会直接忽略这些命令。与@echo相比，以此方法将简单的make结构的扩展动作显示出来较为方便。事实上，使用可移植性较好的shell运算符：也可以达到几乎一样的效果，不过：运算符可能会进行重定向（redirection）的动作。

因此，一行包含 > word 字样的命令将会产生创建 *word* 文件的副作用。）对此 *makefile* 运行 make，你将会看到如下的结果：

```
$ make
# PWD = /test/book/out/ch03-findstring-1
#
#
#
#
# /test/book/out
#
```

如你所见，每项针对 $(PWD) 的测试几乎都没有返回任何东西，直到我们测试到相应的根目录才会返回根目录本身。请注意，你所看到的只是 findstring 的范例。事实上，你可以自己编写一个函数来返回当前目录树的根目录。

此外还有两个“搜索和替换”函数：

$(subst search-string,replace-string,text)

这是一个不具通配符能力的“搜索和替换”函数。它最常被用来在文件名列表中将一个扩展名替换成另一个扩展名：

```
sources := count_words.c counter.c lexer.c
objects := $(subst .c,.o,$(sources))
```

这可以在 $(sources) 中将所有出现 *.c* 字样的地方都替换成 *.o*，或者较一般的说法是，以“替换字符串”取代所有出现“搜索字符串”的地方。

接下来的例子，你常可以在说明空格对函数调用的参数有何影响的范例中看到。请注意，逗号之后不可以有空格。如果我们将前面的代码做如下的改变：

```
sources := count_words.c counter.c lexer.c
objects := $(subst .c, .o, $(sources))
```

（注意每个逗号之后的空格）$(objects) 的值将会变成这样：

```
count_words .o counter .o lexer .o
```

这并不是我们想要的结果。问题出在 *.o* 参数之前的空格是“替换字符串”的一部分，所以会被放在输出字符串中。*.c* 参数之前的空格没有问题，因为第一个参数之前的任何空白符号都会被 make 移除。事实上，$(sources) 之前的空格也还好，因为 $(objects) 最有可能被作为一个简单的命令行参数，此时前导的空格是没有问题的。然而，我从来不会在一个函数调用中，在逗号之后以不一致的方式加上空格，即使这么做的结果是正确的：

```
# 这个函数调用加空格的方式实在令人难以捉摸
objects := $(subst .c,.o, $(source))
```

请注意，subst 并不知道什么文件名或扩展名，它只知道字符所构成的字符串，所以 $(source) 中只要出现 .c 字样就会被替换掉。例如，文件名 *car.cdr.c* 将会被转换成 *car.odr.o*。或许这并不是你想要的。

你可以在"自动产生依存关系"一节的最后一个 *makefile* 范例中看到它使用 subst 这个函数：

```
VPATH    = src include
CPPFLAGS = -I include
SOURCES  = count_words.c \
           lexer.c       \
           counter.c
count_words: counter.o lexer.o -lfl
count_words.o: counter.h
counter.o: counter.h lexer.h
lexer.o: lexer.h
include $(subst .c,.d,$(SOURCES))
%.d: %.c
        $(CC) -M $(CPPFLAGS) $< > $@.$$$$;                    \
        sed 's,\($*\)\.o[ :]*,\1.o $@ : ,g' < $@.$$$$ > $@;   \
        rm -f $@.$$$$
```

此处，subst 函数被用来将源文件列表转换成依存文件列表。因为这些依存文件将会成为 include 的参数，所以 make 会把它们作为必要条件，并且使用 %.d 规则来更新它们。

$(patsubst search-pattern,replace-pattern,text)

这是一个具通配符能力的"搜索和替换"函数。照例，此处的模式只可以包含一个 % 字符。replace-pattern 中的百分比符号会被扩展成与模式相符的文字。切记，search-pattern 必须与 text 的整个值进行匹配。例如，下面的例子只会删除 text 结尾的斜线符号，而不是 text 中的每个斜线符号：

```
strip-trailing-slash = $(patsubst %/,%,$(directory path))
```

对于相同的替换操作来说，替换引用（substitution reference）是个具可移植性的做法。替换引用的语法如下：

```
$(variable:search=replace)
```

其中，*search* 可以是一个简单的字符串，如果是这样的话，只要该字符串出现在一个单词的结尾（即后面接着空格或变量值的结尾），就会被替换成 *replace*。此外，*search* 可以包含一个代表通配字符的 %，如果是这样的话，make 会依照 patsubst 的规则进行搜索和替换的操作。我觉得这个语法含糊不清，与 patsubst 相比较难读懂。

如我们先前所见，变量通常会包含一串单词。接下来我们会看到可以从一份列表中选出所需单词的函数、计算列表长度的函数等。如同所有其他的 make 函数，单词列表中是以空格为分隔符。

$(words text)

此函数会返回 text 中单词的数量。

```
CURRENT_PATH := $(subst /, ,$(HOME))
words:
        @echo My HOME path has $(words $(CURRENT_PATH)) directories.
```

本书将会多次使用此函数，稍后我们将会看到，不过它通常需要跟其他函数一起使用。

$(words n,text)

此函数会返回text中的第*n*个单词，第一个单词的编号是1。如果*n*的值大于text中单词的个数，则此函数将会返回空值。

```
version_list  := $(subst .,  ,$(MAKE_VERSION))
minor_version := $(word 2, $(version_list))
```

其中，MAKE_VERSION 是一个内置变量（参见“标准的 make 变量”一节）。

你总是能够以如下的方式取得列表中的最后一个单词：

```
current := $(word $(words $(MAKEFILE_LIST)), $(MAKEFILE_LIST))
```

这将会返回最近所读取的 *makefile* 的文件名。

$(firstword text)

此函数会返回 text 中的第一个单词。此功能等效于 $(word 1,text)。

```
version_list := $(subst .,  ,$(MAKE_VERSION))
major_version := $(firstword $(version_list))
```

$(wordlist start,end,text)

此函数会返回 text 中范围从 start（含）到 end（含）的单词。如同 word 函数，第一个单词的编号是 1。如果 start 的值大于单词的个数，则函数所返回的是空值；如果 end 的值大于单词的个数，则函数将会自 start 开始返回所有单词。

```
# $(call uid_gid, user-name)
uid_gid = $(wordlist 3, 4, \
            $(subst :, ,    \
              $(shell grep "^$1:" /etc/passwd)))
```

重要的杂项函数

在我们使用这些函数来管理文件名之前，让我们先来了解两个非常有用的函数：sort 和 shell。

$(sort list)

sort 函数会排序它的 list 参数并且移除重复的项目。此函数运行之后会返回按照字典编纂顺序排列的不重复的单词列表，并以空格作为分隔符。此外，sort 函数还会删除前导以及结尾的空格。

```
$ make -f- <<< 'x:;@echo =$(sort    d  b s   d      t   )='
=b d s t=
```

因为sort函数是由make直接实现，所以它并不支持sort程序所提供的任何选项。此函数的参数通常是一个变量或是另一个make函数的返回值。

$(shell command)

shell函数的参数会被扩展（就像所有其他的参数）并且传递给subshell来执行。然后make会读取command的标准输出，并将之返回成函数的值。输出中所出现的一系列换行符号会被缩减成单一空格符号，任何接在后面的换行符号都会被删除。标准错误以及任何程序的结束状态都不会被返回。

```
stdout := $(shell echo normal message)
stderr := $(shell echo error message 1>&2)
shell-value:
        # $(stdout)
        # $(stderr)
```

如你所见，*stderr*的信息就像往常那样被送往终端机，所以shell函数的输出中并不会包含*stderr*的信息：

```
$ make
error message
# normal message
#
```

你可以在下面看到一个用来创建一组目录的循环：

```
REQUIRED_DIRS = ...
_MKDIRS := $(shell for d in $(REQUIRED_DIRS); \
            do                                \
              [[ -d $$d ]] || mkdir -p $$d;   \
            done)
```

通常，若能确保在任何命令脚本运行之前，必要的输出目录已经存在，则*makefile*的编写会比较容易。这个make变量（即_MKDIRS）会创建必要的目录，它会使用bash shell的for循环来确保这些必要的目录已经存在。双中括号是用来进行bash的条件测试，它的语法跟test程序类似，但是它不会进行单词的分隔以及路径名称的扩展。因此，如果变量所包含的文件名里有内置的空格符号，则测试的动作仍会正确地进行（而且不用加上引号）。在*makefile*文件中，这个make变量的赋值动作应该提早进行，以确保命令脚本或变量在使用输出目录之前就已经存在。_MKDIRS的值无关紧要，而且make实际上并不会用到_MKDIRS变量本身。

因为shell函数可用来调用任何外部的程序，所以使用时应该小心以对。特别是，你应该考虑简单变量与递归变量之间的差异。

```
START_TIME   := $(shell date)
CURRENT_TIME = $(shell date)
```

START_TIME变量会在定义的时候执行一次date命令。在*makefile*文件中，CURRENT_TIME变量每被使用一次就会执行一次date命令。

现在我们的工具箱已经能够编写出极为有用的函数。如下的函数可用来测试一个值是否包含重复的内容：

```
# $(call has-duplicates, word-list)
has-duplicates = $(filter                    \
                   $(words $1)               \
                   $(words $(sort $1)))
```

这个函数将会分别对单词列表（word-list）的两个版本（一个内容可能重复，另一个内容绝不会重复）计算单词的数量，然后比较这两个数字。然而，make函数并不了解数字，只了解字符串，因此我们必须把数字作为字符串来比较。最简单的方法就是使用filter函数。所以我们会在其他的数字中搜索一个数字。如果这两个数字相同，has-duplicates函数将会返回一个非空值（non-null）。

使用如下的方法，你可以通过时间戳轻易产生出文件名来：

```
RELEASE_TAR := mpwm-$(shell date +%F).tar.gz
```

这会产生如下的文件名：

```
mpwm-2003-11-11.tar.gz
```

让date多做一点事，我们也可以产生同样的文件名：

```
RELEASE_TAR := $(shell date +mpwm-%F.tar.gz)
```

接下来的函数可用来把相对路径（例如相对于*com*的目录）转换成完整的Java类名：

```
# $(call file-to-class-name, file-name)
file-to-class-name := $(subst /,.,$(patsubst %.java,%,$1))
```

使用两个subst函数也可以达到相同的目的：

```
# $(call file-to-class-name, file-name)
file-to-class-name := $(subst /,.,$(subst .java,,$1))
```

接着，我们可以通过此函数来调用Java类：

```
CALIBRATE_ELEVATOR := com/wonka/CalibrateElevator.java
calibrate:
        $(JAVA) $(call file-to-class-name,$(CALIBRATE_ELEVATOR))
```

如果你在$(sources)里的文件名发现com之前具有多个上层目录组件，你可以使用如下的函数，并以目录树的根目录作为第一个参数来移除它们（注3）：

```
# $(call file-to-class-name, root-dir, file-name)
file-to-class-name := $(subst /,.,                    \
                        $(subst .java,,               \
                          $(subst $1/,,$2)))
```

当你看到嵌套函数调用的时候，通常采取“由内而外”的方式最容易理解。让我们从最里面的subst开始，此函数首先会移除$1/字符串，接着会移除*.java*字符串，最后会把所有的斜线符号全都转换成点号。

文件名函数

*makefile*的编写者通常会花许多时间在文件的处理上。难怪有很多make函数被提供来协助此工作。

$(wildcard pattern...)

第二章曾提到过，通配符可被用于工作目标、必要条件以及命令脚本等语境中。但如果我们想将此功能用在其他语境中，例如变量定义，该怎么办？使用shell函数，我们可以通过subshell来扩展模式，但如果我们需要经常这么做，运行起来将会非常地慢。此时，我们可以使用wildcard函数：

```
sources := $(wildcard *.c *.h)
```

wildcard函数的参数是一份模式列表，它会对列表中的每个模式进行扩展的动作。（注4）如果被扩展的模式找不到相符的文件，则会返回空字符串。如同工作目标和必要条件中的通配符扩展一样，wildcard函数也支持shell的文件名匹配字符（globbing character）：~、*、?、[...]和[^...]。

wildcard的另一个用法，就是在条件语句中测试文件是否存在。与if函数（稍后会提到）并用时，你常会看到wildcard函数调用的参数中并未包含任何的通配字符。例如：

```
dot-emacs-exists := $(wildcard ~/.emacs)
```

将会返回空字符串，如果用户的主目录中并未包含*.emacs*这个文件。

注3：在Java中，包所声明的类名，建议（以颠倒的形式）使用开发者的完整网络域名。此外，目录结构通常会按照包结构来摆放。因此，许多源代码树看起来会像这样：*root-dir/com/company-name/dir*。

注4：make 3.80的在线使用手册所举的例子都只有一种模式。

$(dir list...)

dir函数会返回list中每个单词的目录部分。下面的用户自定义函数会返回包含C源文件的每个子目录：

```
source-dirs := $(sort                          \
                 $(dir                         \
                   $(shell find . -name '*.c')))
```

find命令会返回当前目录中所有C源文件的文件路径，接着由dir函数删掉文件名的保留目录的部分，最后由sort移除重复的项目。请注意，为避免每次此变量被使用时会重新执行find命令（假设在*makefile*运行期间，源文件会自发性地出现和不见），此处所定义的变量是一个简单变量。接着举一个需要定义递归变量的例子：

```
# $(call source-dirs, dir-list)
source-dirs = $(sort                           \
                 $(dir                         \
                   $(shell find $1 -name '*.c')))
```

这个版本的source-dirs变量将会使用目录列表（以空格为分隔符）作为它的第一个参数。这个参数包含了find命令所要搜索的一个或多个目录。find命令将会以第一个“破折号”作为目录列表的终止符（我用了几十年，居然没发现这个find功能！）。

$(notdir name...)

notdir函数会返回文件路径的文件名部分。例如，下面的用户自定义函数会从一个Java源文件返回Java的类名：

```
# $(call get-java-class-name, file-name)
get-java-class-name = $(notdir $(subst .java,,$1))
```

我们经常可以看到dir和notdir会被一起使用来产生必要的输出。举例来说，假设你必须在输出文件所在的目录中执行自定义shell脚本以产生输出文件。

```
$(OUT)/myfile.out: $(SRC)/source1.in $(SRC)/source2.in
        cd $(dir $@); \
        generate-myfile $^ > $(notdir $@)
```

自动变量$@代表工作目标，可以被分解成目录和文件两个独立的值。事实上，如果OUT是一个绝对路径，此处就不必使用notdir函数了，不过这么做会使得输出较具可读性。

在命令脚本中，分解文件路径的另一个方法就是使用$(@D)和$(@F)（参见“自动变量”一节）。

下面是用来新增和删除扩展名、基本文件名等的函数。

$(suffix name...)

suffix函数会返回它的参数中每个单词的后缀(即文件名称的扩展名)。例如，下面的用户自定义函数将会测试列表中所有单词是否具有相同的后缀：

```
# $(call same-suffix, file-list)
same-suffix = $(filter 1 $(words $(sort $(suffix $1))))
```

条件语句里的suffix函数常会跟findstring一起使用。

$(basename name...)

basename是suffix的反函数。basename所返回的是文件名称中不含后缀的部分。调用basename之后，任何前导的路径组件都会被原封不动地保留下来。我们可以使用basename来改写前面所举的file-to-class-name和get-java-class-name函数范例：

```
# $(call file-to-class-name, root-directory, file-name)
file-to-class-name  := $(subst /,.,                  \
                         $(basename                  \
                           $(subst $1/,,$2)))
# $(call get-java-class-name, file-name)
get-java-class-name =  $(notdir $(basename $1))
```

$(addsuffix suffix,name...)

addsuffix函数会将你所指定的suffix附加到name中所包含的每个单词后面。suffix可以是任何值。下面的用户自定义函数会从PATH中找出所有相符的文件：

```
# $(call find-program, filter-pattern)
find-program = $(filter $1,                          \
                 $(wildcard                          \
                   $(addsuffix /*,                   \
                     $(sort                          \
                       $(subst :, ,                  \
                         $(subst ::,:.:,             \
                           $(patsubst :%,.:%,        \
                             $(patsubst %:,%:.,$(PATH)))))))))
find:
        @echo $(words $(call find-program, %))
```

最内层的三次替换动作是用来处理shell语法中的特例。一个空的路径组件代表当前目录。为了将这个特殊的语法正规化，我们会依次搜索一个空的结尾路径组件、一个空的前导路径组件以及一个位于中间的空路径组件。任何匹配的组件都会被替换成点号。然后我们会把路径分隔符（path separator）替换成空格以便区分出单词，再使用sort函数移除重复的路径组件。接着把文件名匹配字符附加在每个单词后面，再调用wildcard以便扩展各个文件名匹配表达式（globbing expression）。最后使用filter取出我们所需要的模式。

尽管看起来这个函数的运行似乎会非常慢（而且可以在许多系统上运行），但在我的1.9 GHz P4的（具有512 MB内存）机器上，运行这个函数只需要0.20秒，找

到了4335个程序。如果把$1参数向内移往wildcard函数，还可以提高效能。所以让我们去除filter函数，并且让addsuffix函数使用调用者的参数：

```
# $(call find-program,wildcard-pattern)
find-program = $(wildcard                          \
                 $(addsuffix /$1,                  \
                   $(sort                          \
                     $(subst :, ,                  \
                       $(subst ::,:.:,             \
                         $(patsubst :%,.:%,        \
                           $(patsubst %:,%:.,$(PATH))))))))
find:
        @echo $(words $(call find-program,*))
```

运行这个版本的函数需要0.17秒。这个版本运行得比较快是因为wildcard不再需要返回每个文件，只是为了让此函数稍后使用filter来丢掉它们。你可以在GNU make的在线使用手册看到类似的例子。此外，请注意，第一个版本所使用的是filter风格的文件名匹配模式，第二个版本所使用的是wildcard风格的文件名匹配模式（-、*、?、[...]和[^...]）。

$(addprefix prefix,name ...)

addprefix是addsuffix的反函数。下面的用户自定义函数可用来测试一组文件是否存在而且不是空的：

```
# $(call valid-files, file-list)
valid-files = test -s . $(addprefix -a -s ,$1)
```

此函数的定义方式跟之前大多数的范例不一样，因为它将会在命令脚本中运行。此函数是以shell的test程序与-s选项（如果文件存在而且不是空的则返回真值）进行测试的动作。当有多个文件要测试时，test命令需要在各文件名之间使用-a（表示and）选项，所以此函数会使用addprefix为每个文件名前置-a。这串"and"链开头的第一个文件就是"点号"，它的测试结果总是为"真"。

$(join prefix-list,suffix-list)

join是dir和notdir的反函数。此函数的参数是两个列表：prefix-list和suffix-list，它会把prefix-list的第一个元素与suffix-list的第一个元素衔接在一起，然后把prefix-list的第二个元素与suffix-list的第二个元素衔接在一起，依此类推。此函数可用来重建被dir和notdir分解的列表。

流程控制

因为到目前为止我们所看到的函数有很多被实现成针对串行（列表）来进行处理的，所以即使不使用循环结构，它们也能够运作得很好。然而，如果不提供实际的循环运算符以及某种条件处理能力，make的宏语言将会受到非常大的限制。还好，make支持以上

所提到的这两种功能。这一节还会谈到“无可挽回的”error函数，此函数显然是流程控制最极端的一种形式！

$(if condition,then-part,else-part)

if函数（不要跟第三章所提到的条件指令ifeq、ifne、ifdef和ifndef搞混了）会根据条件表达式（conditional expression）的求值结果，从两个宏中选一个出来，进行扩展的动作。如果condition扩展之后包含任何字符（即使是空格），那么它的求值结果为“真”，于是会对then-part进行扩展的动作，否则，如果condition扩展之后空无一物，那么它的求值结果为“假”，于是会对else-part进行扩展的动作（注5）。

想要测试*makefile*是否在Windows上运行，方法非常简单。查看COMSPEC环境变量就行了，因为只有Windows会定义此环境变量：

```
PATH_SEP := $(if $(COMSPEC),;,:)
```

make对condition求值时，首先会移除前导和接在后面的空格，然后对条件表达式进行扩展的动作。如果扩展之后包含任何字符（包括空格），则表达式的求值结果为“真”。现在不论*makefile*是在Windows还是在Unix上运行，PATH_SEP所包含的是可在路径中使用的正确分隔符。

上一章我曾提到过，如果你要使用最新的功能（像eval），应该检查make的版本是否支持。这种测试字符串值的工作常会用到if和filter函数：

```
$(if $(filter $(MAKE_VERSION),3.80),,\
  $(error This makefile requires GNU make version 3.80.))
```

现在，当make的后续版本被发布时，你还可以将其他可接受的版本编号加入条件表达式中：

```
$(if $(filter $(MAKE_VERSION),3.80 3.81 3.90 3.92),,\
  $(error This makefile requires one of GNU make version -.))
```

这个方法的缺点是，当make有新的版本被发布时，你就必须对此代码进行更新的动作。不过这不会常常发生就是了。例如，3.80版从2002年10月发布至今，尚未有新的版本出现。在*makefile*中，如上的测试动作可被添加到*makefike*以作为最顶层的表达式。因为，如果表达式的求值结果为“真”，if就会被缩减成空无一物，否则error会终止make的运行状态。

$(error text)

error函数可用来输出“无可挽回的”错误信息（fatal error message）。在此函

注5：我在第三章曾提到宏语言跟其他程序语言是不同的。宏语言的运作方式是通过宏的定义和扩展，把源文本转换成输出文本。当我们看过if函数的运作方式之后，你会更清楚其间的差异。

数输出信息之后，make将会以2这个结束状态终止运行。输出中包含当前*makefile*的名称、当前的行号以及消息正文。接下来让我们为make实现常见的assert编程结构：

```
# $(call assert,condition,message)
define assert
  $(if $1,,$(error Assertion failed: $2))
endef
# $(call assert-file-exists,wildcard-pattern)
define assert-file-exists
  $(call assert,$(wildcard $1),$1 does not exist)
endef
# $(call assert-not-null,make-variable)
define assert-not-null
  $(call assert,$($1),The variable "$1" is null)
endef
error-exit:
        $(call assert-not-null,NON_EXISTENT)
```

第一个函数assert只会测试它的第一个参数，如果该参数是空的，则会输出用户的错误信息。第二个函数建立在第一个函数之上，它会以通配模式（wildcard pattern）来测试文件是否存在。请注意，它的参数可以包含任意多个文件名匹配模式（globbing pattern）。

第三个函数是一个非常有用的断言（assert），这是一个基于“经求值的变量”（computed variable）的函数。一个make变量中可以包含任何内容，包括另一个make变量的名称。但如果一个变量包含了另一个变量的名称，要如何取得另一个变量的值？嗯，非常简单，只要扩展该变量两次就行了：

```
NO_SPACE_MSG := No space left on device.
NO_FILE_MSG  := File not found.
...
STATUS_MSG   := NO_SPACE_MSG
$(error $($(STATUS_MSG)))
```

为了保持简单性，这个例子显得有些不自然，它通过存储错误信息变量名称的方式，将STATUS_MSG设定成多个错误信息中的一个。当它要输出信息的时候，首先会扩展STATUS_MSG以便取得错误信息变量名称$(STATUS_MSG)，接着会扩展$(STATUS_MSG)以便取得错误信息$($(STATUS_MSG)。在assert-not-null函数中，我们假设该函数的参数为make变量的名称。我们首先会扩展参数$1以便取得变量名称，接着会扩展$($1)以便检查它是否有值。如果它是个空值，我们就会在错误信息中使用$1里的变量名称。

```
$ make
Makefile:14: *** Assertion failed: The variable "NON_EXISTENT" is null.  Stop.
```

此外，warning函数（参见“较不重要的杂项函数”一节）也会输出格式跟error一样的信息，但是它不会终止make的运行状态。

$(foreach variable,list,body)

这个函数可让你在反复扩展文本的时候，将不同的值替换进去。请注意，这跟你使用不同的参数反复执行函数的状况是不一样的(尽管这么做也能达到相同的目的)。例如：

```
letters := $(foreach letter,a b c d,$(letter))
show-words:
        # letters has $(words $(letters)) words: '$(letters)'
$ make
# letters has 4 words: 'a b c d'
```

当这个foreach函数被执行时，它会反复扩展循环主体$(letter)，并且将循环控制变量letter的值依次设定成a、b、c、d。每次扩展所得到的文本会被累积起来，并以空格为分隔符。

如下的用户自定义函数可用来测试一组变量是否定义过：

```
VARIABLE_LIST := SOURCES OBJECTS HOME
$(foreach i,$(VARIABLE_LIST), \
  $(if $($i),,                \
    $(shell echo $i has no value > /dev/stderr)))
```

(要在shell函数里使用假文件*/dev/stderr*，必须将SHELL设定成bash。)这个循环会把i依次设定成VARIABLE_LIST中的每个单词。if里的测试表达式首先会对$i求值以便取得变量名称，然后再对此结果求值一次以便检查它是否不为空。如果表达式的求值结果不是空的，则*then*的部分什么事也不做；否则，*else*的部分会输出警告信息。请注意，如果我们删除echo命令的重定向符号，shell函数的输出将会被替换到*makefile*里，这会导致语法错误。如你所见，整个foreach循环会被扩展成空无一物。

稍早曾提到，我们要示范如何以循环和条件测试在单词中搜索子字符串，下面这个函数将会从单词列表中找出包含特定子字符串的所有单词：

```
# $(call grep-string, search-string, word-list)
define grep-string
$(strip                                  \
  $(foreach w, $2,                       \
    $(if $(findstring $1, $w),           \
      $w)))
endef
words := count_words.c counter.c lexer.l lexer.h counter.h
find-words:
        @echo $(call grep-string,un,$(words))
```

尽管此函数不接受模式，不过它可以查找简单的子字符串：

```
$ make
count_words.c counter.c counter.h
```

变量何时应该使用圆括号

稍早曾提到，单一字符形式的make变量不需要使用圆括号。例如，所有基本的自动变量的名称都是采用单一字符的形式。就像你在GNU make在线手册所看到的，所有的自动变量都没有被加上圆括号。不过，make在线手册会为所有其他变量使用圆括号，即使是单一字符形式的变量，并且强烈要求用户这么做。这凸显了make变量的特色，因为所有具有“美元符号变量”的其他程序（像shell、perl、awk、yacc等）都不需要使用圆括号。较常见的一个make编程错误就是忘了使用圆括号。下面是使用foreach函数的时候常见的错误：

```
INCLUDE_DIRS := ,.,
INCLUDES := $(foreach i,$INCLUDE_DIRS,-I $i)
# INCLUDES 现在的值为 "-I NCLUDE_DIRS"
```

然而，我发现通过明智地使用单一字符变量以及省略非必要的圆括号，宏将会变得较容易阅读。例如，我认为，如果省略非必要的圆括号，has-duplicates函数将会变得较容易阅读。请比较：

```
# $(call has-duplicates, word-list)
has-duplicates = $(filter                    \
                   $(words $1)               \
                   $(words $(sort $1)))
```

和：

```
# $(call has-duplicates, word-list)
has-duplicates = $(filter                    \
                   $(words $(1))             \
                   $(words $(sort $(1))))
```

然而，kill-program函数可能需要使用圆括号才会容易阅读，因为圆括号可以协助我们区分make变量与shell变量或其他程序所使用的变量：

```
define kill-program
  @ $(PS) $(PS_FLAGS) |                                  \
  $(AWK) 'BEGIN { FIELDWIDTHS = $(PS_FIELDS) }           \
          /$(1)/{                                        \
                 print "Killing " $$3;                   \
                 system( "$(KILL) $(KILLFLAGS) " $$1 )  \
                }'
endef
```

搜索字符串包含了宏的第一个参数$(1)，而$$3和$$1则是awk变量。

只有这么做会较具可读性时，我才会使用单一字符变量并省略圆括号。我通常会对宏的参数以及foreach循环中的控制变量这么做，你应该采用符合自己实际情况的做法。如果你对*makefile*的可维护性有任何疑问，请遵照make在线手册的建议，充分使用圆括

号。别忘了，make程序主要是用来减轻与“软件维护”有关的问题。当你在编写*makefile*的时候，若能将此事铭记在心，则可以避免许多问题。

较不重要的杂项函数

最后，让我们来看一些杂项函数。尽管它们的使用不如foreach或call频繁，不过你将发现自己会经常用到它们。

$(strip text)

strip函数将会从text中移除所有前导和接在后面的空格，并以单一空格符号来替换内部所有的空格。此函数常用来清理“条件表达式”中所使用的变量。

当变量和宏的定义跨越多行时，我常会通过此函数从中移除非必要的空格。不过，如果函数会受参数的前导空格的影响，为函数的参数$1、$2等加上strip也是个不错的主意。通常程序员并不知道，在call的参数列表中，make会于逗号之后附加一个空格。

$(origin variable)

origin函数将会返回描述变量来自何处的字符串。这个变量可以协助你决定如何使用一个变量的值。举例来说，如果一个变量来自环境，或许你会想要忽略该变量的值；如果该变量来自命令行，你就不会这么做。让我们来看一个比较具体的范例。下面是一个新的断言函数，可用来测试一个变量是否有定义：

```
# $(call assert-defined,variable-name)
define assert-defined
  $(call assert,                          \
    $(filter-out undefined,$(origin $1)), \
    '$1' is undefined)
endef
```

origin的可能返回值包括：

undefined

变量尚未定义。

default

变量的定义来自make的内置数据库。如果你改变了内置变量的值，origin所返回的是最近一次的定义来自何处。

environment

变量的定义来自环境（而且并未使用--environment-overrides选项）。

environment override

变量的定义来自环境（而且使用了--environment-overrides选项）。

file

变量的定义来自 *makefile*。

command line

变量的定义来自命令行。

override

变量的定义来自 override 指令。

automatic

这个变量是 make 所定义的自动变量。

$(warning text)

warning 函数类似于 error 函数，不过 warning 不会导致 make 结束运行。如同 error 函数，warning 的输出中包含了当前 *makefile* 的文件名、当前的行号以及所要显示的信息内容。warning 函数扩展之后会变成空字符串，所以它几乎可以使用在任何地方。

```
$(if $(wildcard $(JAVAC)),,                                  \
  $(warning The java compiler variable, JAVAC ($(JAVAC)), \
            is not properly set.))
```

高级的用户自定义函数

我们将会花费大量的时间在宏函数的编写上。可惜，make 并未提供多少可以协助我们进行调试的功能。让我们试着编写一个简单的调试追踪函数以协助我们摆脱此困境。

正如稍早所说，call 函数将会把它的每个参数依次绑定到 $1、$2 等编号变量。你可以为 call 函数指定任意多个参数，你还可以通过 $0 来访问当前所执行的函数的名称（即变量名称）。使用这个信息，我们可以编写一对调试函数来追踪宏的扩展过程：

```
# $(debug-enter)
debug-enter = $(if $(debug_trace),\
                $(warning Entering $0($(echo-args))))

# $(debug-leave)
debug-leave = $(if $(debug_trace),$(warning Leaving $0))

comma := ,
echo-args   = $(subst ' ','$(comma) ',\
                $(foreach a,1 2 3 4 5 6 7 8 9,'$($a)'))
```

如果我们想要查看函数 a 和 b 是如何被调用的，我们可以这么做：

```
debug_trace = 1
```

```
define a
  $(debug-enter)
  @echo $1 $2 $3
  $(debug-leave)
endef

define b
  $(debug-enter)
  $(call a,$1,$2,hi)
  $(debug-leave)
endef

trace-macro:
        $(call b,5,$(MAKE))
```

通过在函数的开头和结尾摆放debug-enter和debug-leave变量，你可以追踪函数的扩展过程。这些函数相当简单。echo-args函数只会输出前9个参数，更糟的是，它无法决定调用中实际参数的个数（当然，make也没办法！）。然而，我还是在调试的时候使用了这些函数。对这个 *makefile* 执行make之后会产生如下的追踪输出：

```
$ make
makefile:14: Entering b( '5', 'make', '', '', '', '', '', '', '')
makefile:14: Entering a( '5', 'make', 'hi', '', '', '', '', '', '')
makefile:14: Leaving a
makefile:14: Leaving b
5 make hi
```

最近曾听到有人这么说："以前，我根本不会把make作为是一种程序语言。" GNU make已非老祖母昔日所使用的make!

eval 与 value 函数

eval是个与所有其他内置函数完全不同的函数，它的用途是将文本直接放入make解析器。例如：

```
$(eval sources := foo.c bar.c)
```

首先make会扫描eval的参数中是否有变量以便进行扩展的动作（就像对任何函数的任何参数所做的那样），然后make会解析文本以及进行求值的动作，就好像它是来自输入文件的一样。此处所举的例子似乎太简单，或许你会觉得奇怪，为何我要自找麻烦地使用这个函数。让我们再来看一个比较有意义的例子。假设你有一个用来编译许多程序的 *makefile*，而且你想要为每个程序定义若干变量，例如sources、headers和objects。此时，你不必反复地以手动的方式为每个程序定义这些变量：

```
ls_sources := ls.c glob.c
ls_headers := ls.h glob.h
```

```
ls_objects := ls.o glob.o
...
```

你可以定义宏，让它帮你做这个工作：

```
# $(call program-variables, variable-prefix, file-list)
define program-variables
  $1_sources = $(filter %.c,$2)
  $1_headers = $(filter %.h,$2)
  $1_objects = $(subst .c,.o,$(filter %.c,$2))
endef

$(call program-variables, ls, ls.c ls.h glob.c glob.h)

show-variables:
        # $(ls_sources)
        # $(ls_headers)
        # $(ls_objects)
```

program-variables宏有两个自变量：一个是这三个变量的前缀，另一个是一份文件列表，宏可从中选出文件以便设定这三个变量。但是，当我们使用这个宏的时候，却得到如下的错误信息：

```
$ make
Makefile:7: *** missing separator.  Stop.
```

正如我们所预料的，这么做是行不通的。这跟make解析器的运作方式有关。一个（位于解析顶层的）宏被扩展成多行文本是不合法的，这会导致语法错误。此时，解析器会以为call这一行是一个规则或是命令行的一部分，但是找不到分隔符记号（separator token）。这是一个令人相当困惑的错误信息，eval函数可用来解决这个问题。如果对call这一行做如下的修改：

```
$(eval $(call program-variables, ls, ls.c ls.h glob.c glob.h))
```

就可以获得我们预期的结果：

```
$ make
# ls.c glob.c
# ls.h glob.h
# ls.o glob.o
```

eval可以解决解析的问题，因为eval能够处理多行形式的宏的扩展动作，而且它本身会被扩展成零行。

现在，我们可以非常简单地使用这个宏来定义三个变量。注意宏中赋值表达式里的变量名称，它是由一个可变的前缀（来自函数的第一个参数）和一个固定的后缀所构成，例如$1_sources。准确地说，这些变量并非前面所提到的“经求值的变量”，不过它们非常相似。

接着，我们还想在这个宏里引入我们的规则：

```
# $(call program-variables,variable-prefix,file-list)
define program-variables
  $1_sources = $(filter %.c,$2)
  $1_headers = $(filter %.h,$2)
  $1_objects = $(subst .c,.o,$(filter %.c,$2))

  $($1_objects): $($1_headers)
endef

ls: $(ls_objects)

$(eval $(call program-variables,ls,ls.c ls.h glob.c glob.h))
```

program-variables的这两个版本，可让我们看到函数参数中的空格所造成的问题。在前一个版本中，函数所使用的两个参数并不受前导空格的影响。也就是说，不管 $1 或 $ 2 是否有前导空格，其行为都一样。然而，新的版本使用了“经求值的变量”$($1_objects)和$($1_headers)。现在为函数的第一个参数（ls）加上一个前导的空格，好让“经求值的变量”能够以一个前导的空格开头，这样它会被扩展成空无一物，因为没有一个变量被定义成以前导空格开头的。这是一个相当难以诊断的问题。

当这个*makefile*被运行之后，我们发现不知为什么，*.h*必要条件会被make忽略掉。为了诊断此问题，我们决定使用 --print-data-base 选项来运行make，以便查看make的内部数据库。我们看到了一些奇怪的地方：

```
$ make --print-data-base | grep ^ls
ls_headers = ls.h glob.h
ls_sources = ls.c glob.c
ls_objects = ls.o glob.o
ls.c:
ls.o: ls.c
ls: ls.o
```

*ls.o*的必要条件*.h*不见了！这个使用“经求值的变量”的规则有点问题。

当make解析eval函数调用时，它首先会扩展用户自定义函数program-variables。这个宏的第一行会被扩展成：

```
ls_sources = ls.c glob.c
```

请注意，宏中的每一行会如预期般地立即被扩展。其他的变量赋值动作也是以同样的方式来处理。然后我们会来到描述规则的地方：

```
$($1_objects): $($1_headers)
```

“经求值的变量”里的变量名称会先被扩展：

```
$(ls_objects): $(ls_headers)
```

然后会进行外部的变量扩展，这次会产生如下的结果：

```
:
```

等一下！我们的变量跑哪里去了？答案是，虽然make会扩展前面那三个赋值表达式，但是却不会对它们求值。让我们继续看下去。一旦对 program-variables 的调用被扩展后，make 会看到如下的结果：

```
$(eval   ls_sources = ls.c glob.c
ls_headers = ls.h glob.h
ls_objects = ls.o glob.o

:)
```

接着，eval 函数会执行并定义这三个变量。所以，答案是，在规则中的变量被实际定义之前 make 就已经将它们扩展了。

要解决这个问题，我们可以把“经求值的变量”的扩展动作延后到这三个变量定义后进行。方法就是为“经求值的变量”加上美元符号：

```
$$($1_objects): $$($1_headers)
```

这一次 make 的数据库会显示我们所预期的必要条件：

```
$ make -p | grep ^ls
ls_headers = ls.h glob.h
ls_sources = ls.c glob.c
ls_objects = ls.o glob.o
ls.c:
ls.h:
ls.o: ls.c ls.h glob.h
ls: ls.o
```

结果，eval 的参数会被扩展两次：第一次是在 make 为 eval 准备参数列表的时候，第二次是 eval 自己进行的。

我们解决最后一个问题的方法是延后“经求值的变量”的求值动作。处理这个问题的另一个方法就是以 eval 函数封装（wrapping）每个变量赋值表达式，迫使其提早进行求值的动作：

```
# $(call program-variables,variable-prefix,file-list)
define program-variables
  $(eval $1_sources = $(filter %.c,$2))
  $(eval $1_headers = $(filter %.h,$2))
  $(eval $1_objects = $(subst .c,.o,$(filter %.c,$2)))
```

```
  $($1_objects): $($1_headers)
endef
ls: $(ls_objects)
$(eval $(call program-variables,ls,ls.c ls.h glob.c glob.h))
```

通过把变量赋值表达式封装在它们自己的eval调用中，可让make在扩展program-variables宏的时候将它们放入内置数据库中。于是它们就可以立即被使用在宏中。

加强这个*makefile*的同时，我们觉得应该为宏加入另一个规则。任何一个我们所要编译的程序都应该依存于它自己的目标文件。所以，为了让我们的*makefile*能够参数化，我们会加入一个最顶层的*all*工作目标，以及使用一个变量来保存我们所要编译的每个程序：

```
#$(call program-variables,variable-prefix,file-list)
define program-variables
  $(eval $1_sources = $(filter %.c,$2))
  $(eval $1_headers = $(filter %.h,$2))
  $(eval $1_objects = $(subst .c,.o,$(filter %.c,$2)))

  programs += $1

  $1: $($1_objects)

  $($1_objects): $($1_headers)
endef

# 将all工作目标放在此处，所以它就是默认目标
all:

$(eval $(call program-variables,ls,ls.c ls.h glob.c glob.h))
$(eval $(call program-variables,cp,...))
$(eval $(call program-variables,mv,...))
$(eval $(call program-variables,ln,...))
$(eval $(call program-variables,rm,...))

# 将 programs 必要条件放在此处，所以all工作目标被定义在这里
all: $(programs)
```

注意all工作目标与其必要条件的摆放位置。programs变量必须等到那五个eval调用被扩展之后才会有正确的定义，但是我们想让all工作目标成为默认目标。此时，我们可以先为*makefile*加入all工作目标，稍后再加入它的必要条件。

program-variables函数所发生的问题，起因于有些变量的求值动作太早进行。实际上，make所提供的value函数可以协助我们解决此问题。value函数会返回它的variable参数未扩展的值。这个未扩展的值可以传给eval函数做进一步的处理。通过传递未扩展的值，我们可以避免必须为宏中的变量引用加上引号的问题。

可惜，program-variables宏无法使用此函数。因为value是一个“非全有即全无”（all-or-nothing）的函数，如果加以使用，value将不会扩展宏中的任何变量。此外，value不接受任何参数（就算存在任何参数也会加以忽略），所以我们的程序名称及文件名列表参数都不会被扩展。

因为存在以上这些限制，所以本书很少会提到value这个函数。

函数挂钩

用户自定义函数只是一个用来存放文本的变量。如果变量名称中存在$1、$2等引用，call函数将会予以扩展。如果函数中不存在任何变量引用，call并不在意。事实上，如果变量中不存在任何文本，call也不会在意。所以你看不到任何错误或警告信息。如果你不小心拼错了函数的名称，这个特性可能会让你十分沮丧。不过这个特性也可能非常有用。

你可以把需要重复使用的描述全都放在函数里。你越常重复使用一个函数，就越值得花时间把它写好。要让函数更具重用性，可以对它加入挂钩（hook）。挂钩是一种函数引用，用户可以重新加以定义，以便进行自己所自定义的工作。

假设你想在*makefile*文件中建立许多程序库。在某些系统上，你会想要运行ranlib，在某些系统上，你可能还会想要运行chmod。这个时候，为这些操作编写明确的命令并非是你的唯一选择，你可以选择编写一个函数以及加入一个挂钩：

```
# $(call build-library, object-files)
define build-library
  $(AR) $(ARFLAGS) $@ $1
  $(call build-library-hook,$@)
endef
```

为了使用此挂钩，我们会定义build-library-hook函数：

```
$(foo_lib): build-library-hook = $(RANLIB) $1
$(foo_lib): $(foo_objects)
        $(call build-library,$^)

$(bar_lib): build-library-hook = $(CHMOD) 444 $1
$(bar_lib): $(bar_objects)
        $(call build-library,$^)
```

传递参数

一个函数可以从四种来源取得它的数据：使用call所传入的参数、全局变量、自动变

量以及工作目标专属变量。其中，以通过参数为最模块化的选择，因为参数的使用可让函数中的变动与全局数据无关，但有时这并不是最重要的评断标准。

假设我们有若干项目共享一组make函数。我们将会通过变量前缀（variable prefix），例如PROJECT1_，来区分每个项目，而且项目里的重要变量都会使用具“跨项目后缀”（cross-project suffix）的前缀。之前范例中所用到的PROJECT_SRC等变量，将会变成PROJECT1_SRC、PROJECT1_BIN和PROJECT1_LIB。这样，我们就不必编写用来读取这三个变量的函数，我们可以使用“经求值的变量”以及传递单一参数（也就是变量前缀）：

```
# $(call process-xml,project-prefix,file-name)
define process-xml
  $($1_LIB)/xmlto -o $($1_BIN)/xml/$2 $($1_SRC)/xml/$2
endef
```

传递参数的另一个方式就是使用工作目标专属变量。当大部分的调用所使用的都是标准的值，仅有少数需要特别的处理时，这个方式特别有用。当规则被定义在一个引入文件中，但我们想从定义变量的*makefile*来调用该规则时，工作目标专属变量还可以为我们提供相当的灵活性。

```
release: MAKING_RELEASE = 1
release: libraries executables
...
$(foo_lib):
        $(call build-library,$^)
...
# $(call build-library, file-list)
define build-library
  $(AR) $(ARFLAGS) $@              \
    $(if $(MAKING_RELEASE),        \
      $(filter-out debug/%,$1),    \
      $1)
endef
```

这段代码会在运行发行版本的时候设定工作目标专属变量，以便指示此状况。在此状况下，程序库编译（build-library）函数将会从程序库删除任何调试模块。

第五章

命令

到目前为止，我们已经看过了make命令的许多基本元素，不过为了让所有人都能够具备阅读本章的背景知识，让我们先复习一下前面提到过的内容。

make命令实质上是一个单行shell脚本。实际上，make会获取每行命令并将它传递给subshell去执行。事实上，如果make能够保证省略shell不会影响程序的行为，它就可以优化这个（相对而言）代价昂贵的fork/exec算法。make会通过在每个命令行中扫描shell特殊字符（比如通配符以及I/O重定向）来进行此项检查。如果不存在任何shell特殊字符，make就会直接执行此命令，而不会将此命令传递给subshell去执行。

make默认会使用*/bin/sh*这个shell。之所以会使用*/bin/sh*，并非自环境继承而来，而是由SHELL这个make变量所控制的。当make启动时，它会从用户的环境中导入所有变量以作为make变量，但不包含SHELL。这是因为用户对shell的选择不应该导致*makefile*（可能包含在某个你所下载的软件包里）运行失败。如果用户真的要变更make默认要使用的shell，他可以在*makefile*文件中明确设定SHELL变量。稍后我们将会在“要使用哪个shell”一节中探讨这个问题。

解析命令

在工作目标之后，凡是第一个字符为跳格符（tab）的文本行一律会被视为命令（除非前一行的结尾是一个反斜线符号）。GNU make在处理其他语境中的跳格符时会变得比较精明。举例来说，当不可能出现意义不明的状况时，注释、变量赋值以及include指令都可以使用跳格符来作为它们的第一个字符。如果make所读到的命令行并非立即跟在工作目标之后，就会显示如下的错误信息：

```
makefile:20: *** commands commence before first target.  Stop.
```

这个信息的用词有点奇怪，因为它通常会发生在 *makefile* 的中间部分，与“第一个”(first)工作目标所指定之处相去甚远。不过现在我们要理解这个信息并不会有太大的困难。这个信息的用词如果能像“encountered a command outside the context of a target”（发现命令位于工作目标的语境之外）这样或许会比较好。

解析器所看到的命令位于合法的语境时，它会切换到“命令解析”模式，以一次一行的方式建立脚本。当解析器所遇到的文本行不可能成为命令脚本的一部分时，它就会停止追加到脚本的动作。此处就代表脚本的结尾。以下列出可能会出现在脚本中的内容：

- 以跳格符开头的文本行就是将会被subshell执行的命令。进入“命令解析”模式时，即使会被解译成make结构的文本行（比如ifdef，注释以及include指令），也会被视为命令。
- 空行会被忽略掉，所以不会被 subshell 执行。
- 以#开头的文本行 —— 或许会有前导的空格（不是跳格符！），就是 *makefile* 的注释，会被忽略掉。
- 条件处理指令，像 ifdef 和 ifeq，会在脚本中被认出以及处理。

内置的make函数将会终止“命令解析”模式，除非它前置了一个跳格符。这意味着，它们必须被扩展成有效的shell命令，否则就会变成空值。例如函数warning和eval就会被扩展成空值。

“命令脚本中可以使用空行以及make的注释”这件事，可能会让你感到意外。你可以在如下的范例中看到它们的处理方式：

```
long-command:
        @echo Line 2: A blank line follows

        @echo Line 4: A shell comment follows
        # A shell comment (leading tab)
        @echo Line 6: A make comment follows
# A make comment, at the beginning of a line
        @echo Line 8: Indented make comments follow
  # A make comment, indented with leading spaces
        # Another make comment, indented with leading spaces
        @echo Line 11: A conditional follows
    ifdef COMSPEC
        @echo Running Windows
    endif
        @echo Line 15: A warning "command" follows
        $(warning A warning)
        @echo Line 17: An eval "command" follows
        $(eval $(shell echo Shell echo 1>&2))
```

请注意，第5行和第10行看起来一样，但是执行结果却有很大的差别。第5行是shell

的注释，以一个前导的跳格符来表示；而第10行则是make的注释，被内缩了8个空格。显然，我并不建议以此方法来编排make的注释（除非你打算参加晦涩*makefile*大赛）。正如你在接下来的输出结果中所见，make的注释并不会被执行，而且即使它们出现在命令脚本的语境中，你也不会在输出结果中发现它们的踪迹：

```
$ make
makefile:2: A warning
Shell echo
Line 2: A blank line follows
Line 4: A shell comment follows
# A shell comment (leading tab)
Line 6: A make comment follows
Line 8: Indented make comments follow
Line 11: A conditional follows
Running Windows
Line 15: A warning command follows
Line 17: An eval command follows
```

函数warning和eval的输出顺序似乎有问题，不用担心，没有问题（稍后我们将会在"对脚本求值"一节中探讨求值顺序的问题）。"脚本中可以包含任意多个空行和注释"这件事将会让你很容易遇到错误。假设你不经意地加入了一个具有前导跳格符的文本行，如果它的前面存在一个工作目标（可能具备也可能不具备脚本），而且只有脚本或空行介于它们之间，则make将会把你不经意加入的"具前导跳格符的文本行"视为前面那个工作目标的命令。如你所见，这完全合法而且不会产生警告或错误信息，除非该工作目标的规则就位于*makefile*（或引入文件）中的某处。

运气好的话，你不经意加入的"具前导跳格符的文本行"跟前面的命令脚本之间，可能会包含非空行、非注释的文本行。此时，你将会看到"commands commence before first target"这个信息。

现在是介绍一些软件工具的好机会。我想每个人都会同意，用前导的跳格符来表示命令行是一个不幸的决定，但是现在要改似乎有点来不及了。具有语法检查能力的现代化编辑器将会通过"标示可疑的语法结构"的方式来协助我们避免一些潜在的问题。

GNU emacs为*makefile*的编辑提供了非常好的模式。这个模式会进行语法的标示以及查找简单的语法错误，比如续行符号之后有空格以及混用前导的空格与跳格符。我将会在稍后进一步说明如何并用emacs和make。

持续很长的命令

因为每个命令会在它自己的shell中执行（至少看起来是这样），所以若要让一系列shell命令一起执行，则必须经过特别的处理。举例来说，假如我需要产生一个文件，以便用

来保存文件列表。Java编译器可以读取此类文件以一次编译多个源文件。我编写了如下的命令脚本：

```
.INTERMEDIATE: file_list
file_list:
        for d in logic ui
        do
          echo $d/*.java
        done > $@
```

显然这么做是行不通的，它会产生如下的错误：

```
$ make
for d in logic ui
/bin/sh: -c: line 2: syntax error: unexpected end of file
make: *** [file_list] Error 2
```

于是我会为命令脚本中的每一行（最后一行除外）添加延续字符：

```
.INTERMEDIATE: file_list
file_list:
        for d in logic ui        \
        do                       \
          echo $d/*.java         \
        done > $@
```

这次它会产生如下的错误：

```
$ make
for d in logic ui        \
do                       \
  echo /*.java  \
done > file_list
/bin/sh: -c: line 1: syntax error near unexpected token `>'
/bin/sh: -c: line 1: 'for d in logic ui  do                    echo /*.java
make: *** [file_list] Error 2
```

怎么了？又跑出了两个问题。首先，对于循环控制变量的引用d必须加以规避，其次，因为for循环将会以单行的形式传给subshell，所以我们必须于文件列表及for循环语句之后加上“分号”分隔符：

```
.INTERMEDIATE: file_list
file_list:
        for d in logic ui;       \
        do                       \
          echo $$d/*.java;       \
        done > $@
```

现在我们可以产生预期的文件了。这个工作目标被声明为.INTERMEDIATE，所以在编译工作完成之后，make将会删除这个临时的工作目标。

一个比较实际可行的做法，就是将目录列表存放在一个make变量里。如果确定文件的数量不会太多，我们可以通过make函数来达到相同的目的而不必使用for循环：

```
.INTERMEDIATE: file_list
file_list:
        echo $(addsuffix /*.java,$(COMPILATION_DIRS)) > $@
```

如果预期的目录列表会与时俱增（grow with time）的话，for循环的做法不太可能导致命令行过长的问题。

make脚本中另一个常见的问题是如何切换目录。显然，如下的简单脚本：

```
TAGS:
        cd src
        ctags -recurse
```

将无法在src子目录中运行ctags程序。为了获得我们想要的结果，我们必须将这两个命令放在同一行上，或是以反斜线规避换行符号（以及使用分号隔开这两个命令）：

```
TAGS:
        cd src;          \
        ctags -recurse
```

一个更好的做法就是在运行ctags程序之前检查cd的状态：

```
TAGS:
        cd src &&
        ctags -recurse
```

请注意，在某些情况下，省略分号并不会让make或shell产生错误信息：

```
disk-free = echo "Checking free disk space ..." \
            df . | awk '{ print $$4 }'
```

这个例子会先输出一段简单的信息，接着输出当前磁盘的可用块的数目。它会这么做吗？我们不经意地省略了echo命令之后的分号，因此df程序根本不会运行，只会将如下的内容送往awk：

```
Checking free disk space ... df .
```

于是awk便会忠实地输出第四个字段space...。

当你用define指令建立多行形式的命令序列时，也可能会遇到问题。这个问题跟前面的问题不太一样。当一个多行形式的宏被扩展时，宏主体中的每一行会被插入脚本并具有前导的跳格符，make会认为这样的每一行都是各自独立的命令，于是它们并不会在单一的subshell中执行。所以你也应该注意宏中命令行延续的问题。

命令修饰符

一个命令可以通过若干前缀加以修饰。我们已经多次看到“安静模式”前缀（@）被使用在前面的范例中，接下来我们会完整列出所有可用的前缀（修饰符），并提供详细的说明：

@ 不要输出命令。当你想将某个工作目标的所有命令全都隐藏起来的时候，如果考虑到旧版的兼容性，你可以把该工作目标设成特殊工作目标.SILENT的一个必要条件。不过，最好能够使用@，因为它可以应用在脚本中个别的命令上。如果你将这个修饰符应用在所有的工作目标上（尽管很难想象为什么要这么做），你可以使用--silent（或-s）选项。

隐藏命令可让make的输出较容易阅读，不过这么做可能会使得命令的调试较为困难。如果你发现自己常常需要移除和恢复@修饰符，你可以创建一个内含@修饰符的变量，例如QUIET，并将它使用在命令上：

```
QUIET = @
hairy_script:
        $(QUIET) complex script ...
```

往后，如果你在make运行复杂脚本（complex script）的时候需要看到脚本本身，只需要通过命令行重新设定QUIET变量就行了：

```
$ make QUIET= hairy_script
complex script ...
```

- 破折号前缀（dash prefix）用来指示make应该忽略命令中的错误。默认的情况下，当make执行一个命令的时候，它会检查程序或管道的结束状态，如果所返回的是非零（失败）的结束状态，make将会终止命令脚本接下来的执行动作，并且结束执行。这个修饰符会指示make忽略被修饰那行（modified line）的结束状态，并且继续执行下去，就好像没有发生错误一样。我们将会在下一节更深入地探讨这个内容。

如果考虑到旧版的兼容性，你可以通过将工作目标设成.IGNORE特殊工作目标的一个必要条件，让make忽略某部分的命令脚本。如果你想要忽略整个*makefile*中的所有错误，你可以使用--ignore-errors（或-i）选项。同样地，这么做似乎不太有用。

+ 加号修饰符（plus modifier）用来要求make执行命令，就算用户是以--just-print（或-n）命令行选项来执行make的。当你要编写递归形式的*makefile*时，就会用到这个功能。我们将会在“递归式make”一节中更深入地探讨这个议题。

这些修饰符都可以使用在单行命令上。显然，在命令执行之前，这些修饰符都会被去除。

错误与中断

make每执行一个命令就会返回一个状态码。值为零的状态码代表命令执行成功，值为非零的状态码代表发生了某种错误。某些程序会以状态码来指示更具意义的内容，而不仅是代表“错误”。举例来说，grep会返回0来代表“找到相符的模式”，返回1来代表“没有找到相符的模式”，返回2来代表“发生了某种错误”。

通常，当有一个程序运行失败（也就是，返回非零值的结束状态）时，make会停止执行命令的动作，并以错误的状态结束执行。有时你会想让make继续执行下去，以便尽量完成其余的工作目标。举例来说，你可能会想要尽量编译完所有文件，好让你只需执行一次就能看到所有的编译错误。这个时候你可以使用--keep-going（或-k）选项。

虽然-修饰符可让make忽略个别命令所发生的错误，不过我会尽量避免使用这个功能。这是因为，此功能会让自动的错误处理机制复杂化，而且让人有不一致的感觉。

当make忽略错误时，它会在相应工作目标的名称（放在方括号里）之后输出警告信息。例如，当rm试着要删除一个不存在的文件时，会输出如下的内容：

```
rm non-existent-file
rm: cannot remove 'non-existent-file': No such file or directory
make: [clean] Error 1 (ignored)
```

有些命令，比如rm，本身具有选项可用来抑制错误结束状态。抑制错误信息的同时，你还可以使用-f选项迫使rm返回成功结束状态。使用此类选项比依靠-修饰符要好。

有时，如果命令执行成功，你会希望它执行失败，以便取得错误结束状态。这个时候，你应该将程序的结束状态取反（negate）：

```
# 确认程序代码里已不包含调试语句。
.PHONY: no_debug_printf
no_debug_printf: $(sources)
        ! grep --line-number '"debug:' $^
```

可惜make 3.80有缺陷（bug），所以你无法直接这么做。make并不知道!字符是要给shell处理的，它会自己执行该行命令，结果会导致错误。这个问题有一个简单的解决方案，就是为该行命令加入一个shell特殊字符，以作为make判断的依据：

```
# 确认程序代码里已不包含调试语句。
.PHONY: no_debug_printf
no_debug_printf: $(sources)
        ! grep --line-number '"debug:' $^ > /dev/null
```

另一个常见的非预期的命令错误，就是使用shell的if结构时忘了使用else。

```
$(config): $(config_template)
        if [ ! -d $(dir $@) ];      \
        then                        \
          $(MKDIR) $(dir $@);       \
        fi
        $(M4) $^ > $@
```

第一个命令用来测试输出目录是否存在，如果不存在就调用mkdir来创建它。不过，如果该目录存在，if命令将会返回失败的结束状态（此项测试的结束状态），并终止脚本的运行。一个解决方案就是加入else子句：

```
$(config): $(config_template)
        if [ ! -d $(dir $@) ];      \
        then                        \
          $(MKDIR) $(dir $@);       \
        else                        \
          true;                     \
        fi
        $(M4) $^ > $@
```

在shell中，冒号（:）是个no-op命令，它总是会返回“真值”，所以可用来取代true。通过下面的替代方案也可以达到相同的目的：

```
$(config): $(config_template)
        [[ -d $(dir $@) ]] || $(MKDIR) $(dir $@)
        $(M4) $^ > $@
```

现在，只要目录存在或者mkdir执行成功，第一条语句就为“真值”。另一个替代方案就是使用mkdir -p，这样即使目录已经存在，mkdir也会执行成功。即使目录已经存在，以上所提到的各种做法都会在subshell中执行某个动作。如果不想在目录存在的时候执行任何动作，则可以使用wildcard函数。

```
# $(call make-dir, directory)
make-dir = $(if $(wildcard $1),,$(MKDIR) -p $1)

$(config): $(config_template)
        $(call make-dir, $(dir $@))
        $(M4) $^ > $@
```

因为每个命令会在它自己的shell中被执行，所以你常会看到多行命令中的每个命令组件被分号隔开，好让它们在同一个shell中被执行。这样即使其中有命令组件发生错误，也不会让脚本终止运行：

```
target:
        rm fm-fails; echo But the next command executes anyway
```

命令脚本的长度越短越好，这样make才会有机会为你处理结束状态以及终止运行。例如：

```
path-fixup = -e "s;[a-zA-Z:/]*/src/;$(SOURCE_DIR)/;g" \
             -e "s;[a-zA-Z:/]*/bin/;$(OUTPUT_DIR)/;g"

# A good version.
define fix-project-paths
  sed $(path-fixup) $1 > $2.fixed && \
  mv $2.fixed $2
endef

# A better version.
define fix-project-paths
  sed $(path-fixup) $1 > $2.fixed
  mv $2.fixed $2
endef
```

这个宏可让你为特定的源文件树和输出文件树，将DOS风格的路径名称（但以斜线为分隔符）转换成目标路径。这个宏的参数有两个：输入文件的名称和输出文件的名称。只有在sed命令完全正确的状况下，你才需要担心输出文件是否会被覆盖掉。第一个版本（good version）的做法就是使用&&将sed和mv连接在一起，这样它们就可以在同一个shell中执行。第二个版本（better version）的做法就是将它们作为两个独立的命令来执行，让make可以在sed执行失败的时候，终止脚本的运行。第二个版本的代价并不高（mv可以直接执行，并不需要通过shell），也比较容易阅读，而且在错误发生时还能提供较多的信息（因为make会指出是哪个命令执行失败）。

请注意，下面所要谈的并非cd的常见问题：

```
TAGS:
        cd src && \
        ctags --recurse
```

由于脚本中的那两条语句必须在同一个subshell中执行，所以我们必须使用某种语句连接符（像;或&&）来隔开它们。

删除与保存工作目标文件

如果有错误发生，make会假设相应的工作目标无法被重新建立。作为当前工作目标的必要条件的任何其他工作目标也无法被重新建立，所以make不会这么做，也不会执行其脚本中的任何一部分。如果执行make的时候用到了--keep-going（或-k）选项，则make会试图进行下一个工作目标；否则make就会结束执行。假设当前的工作目标是一个文件，如果在它的工作完成之前命令提早结束执行，则这个文件可能会遭到破坏。遗憾的是，为了与旧版兼容，make会把这个可能已遭破坏的文件留在磁盘上。因为该文件的时间戳已经更新，所以make随后的执行动作并不会以正确的数据来更新它。为了避免此问题，只需将该工作目标文件设成.DELETE_ON_ERROR的一个必要条件，这样当有错误发生时，make会删除有问题的文件。当你使用.DELETE_ON_ERROR的时

候，如果没有为它指定任何必要条件，那么不管是哪个工作目标文件的编译过程发生错误，都会使得make删除该工作目标文件。

当make的执行被信号（像Ctrl+C）中断时，会产生相反的问题。这个时候，如果文件被更改，make就会删除当前的工作目标文件。有些时候，删除文件可能不是你想要的结果，或许因为创建这个文件的代价非常高，能够保存部分的内容比什么都没有强，也或许是这个文件必须存在，这样方能继续进行其他部分的编译工作。这个时候，你可以通过将该文件设成特殊工作目标 .PRECIOUS的一个必要条件来保护它。

使用哪个 shell

当make需要传递一个命令给subshell时，默认情况下会使用*/bin/sh*。你可以通过设定SHELL这个make变量来加以变更，但请三思而行。通常make的用途就是为开发者社群提供一个工具，使他们可以从源文件来编译一个系统。如果能依照这个目的来使用工具，*makefile*的创建应该相当容易，不过对开发者社群里的某些人来说目的并非如此。然而，对任何（通过匿名ftp或开放式cvs）广泛发布的应用程序来说，使用*/bin/sh*以外的任何shell被认为是最糟糕的做法。我们将会在第七章更深入地探讨可移植性的相关议题。

不过，make也可能使用在另一种环境之中。通常在封闭的开发环境里，你会看到一群经许可的开发者在一组有限的机器与操作系统上进行开发的工作。事实上，我常常发现自己就身处在这样的环境之中。这个时候，根据make预期要运行的环境来进行环境的自定义最有意义。开发者必须知道如何设置自己的环境好让编译工作能够正确进行，以及让自己的日子好过一点。

在这样的环境之下，我比较喜欢在“最前面”对可移植性做些让步。我相信这样可让整个开发过程更加顺利。我对此所做的让步是将SHELL变量设成*/usr/bin/bash*。bash是一个具可移植性、与POSIX兼容的shell（因此它的功能涵盖sh），也是GNU/Linux所采用的标准shell。*makefile*里许多的移植问题皆肇因于命令脚本中使用了不具可移植性的语法结构。解决这个问题的方法很简单，使用标准的shell就行了，不必刻意使用具可移植性的sh子集。GNU make的维护者Paul Smith在他的Paul's Rules of Makefiles（*http://make.paulandlesley.org/rules.html*）网页上提到：“编写具可移植性的*makefile*是在自找麻烦，为何不使用具可移植性的make！”我也会这么说：“编写具可移植性的命令脚本是在自找麻烦，只要有可能，请使用具可移植性的shell（bash）。”bash shell可以在大多数的操作系统上运行，包括几乎所有的Unix变体、Windows、BeOS、Amiga以及OS/2。

在本书其余的部分，当命令脚本用到bash特有的功能时，我将会加以说明。

空命令

空命令（empty command）就是一个什么事都不做的命令。

```
header.h: ;
```

先前提到过工作目标的必要条件列表可以跟着一个分号和命令。此处，分号之后空无一物，这表示命令并不存在。当然你也可以在工作目标之后指定只包含一个跳格符的空白行，不过这么做根本无法阅读。空命令最常用来避免模式规则匹配特定的工作目标，而执行你不想要的命令的情况。

请注意，在make的其他版本中，空工作目标（empty target）有时会被作为假想工作目标（phony target）来用。在GNU make中，你可以使用.PHONY这个特殊工作目标，因为这么做较安全也较清楚。

命令环境

make执行命令时会从make本身继承其处理环境。此环境包含当前的工作目标、文件描述符以及由make传递的环境变量。

当一个subshell被创建时，make会将若干变量加入环境：

```
MAKEFLAGS
MFLAGS
MAKELEVEL
```

MAKEFLAGS变量包含了你传递给make的命令行选项。MFLAGS变量的内容是MAKEFLAGS的镜像副本，存在的理由是为了旧版的兼容性。MAKELEVEL变量的内容代表嵌套的make调用的次数。也就是说，当make递归调用make时，MAKELEVEL变量的值就会加1。对make而言，具单一父进程的子进程将会具有一个值为1的MAKELEVEL变量。这些变量通常会被用来管理递归式make(recursive make)。请参考"递归式make"一节。

当然，用户可以通过export指令将任何变量加入子进程的环境之中。

make用来执行命令的当前工作目录就是上层make的工作目录。这个目录通常就是你用来执行make程序的目录，不过你可以通过--directory=directory（或-C）命令行选项加以变更。请注意，仅使用--file来指定不同的*makefile*并无法变更当前目录，这只会影响所读取的*makefile*。

make所衍生（spawn）的每个子进程都会继承三个标准的文件描述符：*stdin*、*stdout*和*stderr*。这并没有任何值得特别注意的地方，除了命令脚本可以读取它的*stdin*这件事，这是“合理”并且可行的。一旦命令脚本完成它的读取动作之后，其余命令会如预期般地被执行。不过用户一般不会通过这种交互的方式来运行*makefile*。用户通常会希望这么做：启动make，“走完”每个步骤，返回稍后要检查的结果。当然，能够读取*stdin*将有助于与“基于cron的自动编译过程”进行交互。

*makefile*中一个常见的错误就是意外地读取了*stdin*：

```
$(DATA_FILE): $(RAW_DATA)
        grep pattern $(RAW_DATA_FILES) > $@
```

此处，当我们以变量来为grep指定输入文件的时候，误用了变量名称。如果变量扩展之后空无一物，grep将会读取*stdin*，但不会显示提示符或指示为何make“死掉了”。这个问题的一个简单的解决方案就是，总是在命令行上使用*/dev/null*作为额外的“文件”：

```
$(DATA_FILE): $(RAW_DATA)
        grep pattern $(RAW_DATA_FILES) /dev/null > $@
```

现在，这个grep命令一定不会去读取*stdin*。当然，如果要对*makefile*调试，这么做也很恰当！

对命令脚本求值

命令脚本的处理过程历经四个步骤：读取程序代码、扩展变量、对make表达式求值以及执行命令。现在让我们来看看，如何将这些步骤应用在复杂的命令脚本上。以下面这个（有点不自然的）*makefile*为例。当目标文件被链接成一个应用程序之后，我们可以选择是否要删掉符号，以及是否使用upx这个可执行文件封装程序（executable packer）进行压缩的动作：

```
# $(call strip-program, file)
define strip-program
  strip $1
endef

complex_script:
        $(CC) $^ -o $@
    ifdef STRIP
        $(call strip-program, $@)
    endif
        $(if $(PACK), upx -best $@)
        $(warning Final size: $(shell ls -s $@))
```

命令脚本的求值动作将会延后到它们被执行的时候进行，不过ifdef指令的处理将会在它们被读进make的时候立即进行。因此，make在读取命令脚本的时候会忽略并存储其所读到的每一行，直到它读进ifdef STRIP这一行。它会对条件表达式进行求值的动作，如果STRIP未定义，make会读取并丢弃接下来的所有文本，直到它读进作为结束的endif。然后make会读取并存储命令脚本的其余部分。

当命令脚本被执行时，make首先会扫描命令脚本中是否存在需要被扩展和求值的make语法结构。当宏被扩展时，make会为其中的每一行添加一个前导的跳格符。如果你不打算这么做，那么在任何命令执行之前所进行的扩展及求值动作，可能会导致非预期的执行顺序。在我们的例子中，命令脚本的最后一行是错误的。在应用程序被链接之前，shell和warning已经先被执行了。因此，在文件的检查动作被更新之前，ls命令将会被执行。这说明了为何我们在“解析命令”一节中所看到的输出顺序会“乱掉”。

此外，请注意，make会在读取这个*makefile*的时候对ifdef STRIP那一行进行求值动作，不过它会在complex_script的命令被执行之前立即对$(if...)那一行进行求值动作。使用if函数可让你的代码较具灵活性，因为通过变量的定义，你可以获得更多的控制机会，不过它非常不适合用来管理大型的文本块。

如此例所示，我们必须随时注意当前是哪个程序在对表达式进行求值动作（例如make或shell），以及何时会进行求值动作：

```
$(LINK.c) $(shell find $(if $(ALL),$(wildcard core ext*),core) -name '*.o')
```

这个回旋式的命令脚本可用来链接一组目标文件。以下列出这个命令脚本的求值顺序，并在圆括号中提示执行此动作的程序：

1. 扩展$ALL (make)。
2. 对if求值 (make)。
3. 对wildcard求值，假设ALL并非空值 (make)。
4. 对shell求值 (make)。
5. 执行find (sh)。
6. 完成make语法结构的扩展和求值动作之后，执行链接命令 (sh)。

命令行的长度限制

开发大型项目的时候，你偶尔会遇到make所要执行的命令过长的问题。命令行的长度限制因操作系统的不同有很大的差异。Red Hat 9 GNU/Linux的长度限制大约为128K个

字符，而Windows XP则具有32K的限制，因此所产生的错误信息也不相同。在Windows上使用Cygwin的人，如果为ls指定过长的参数列表，将会看到如下的信息：

```
C:\usr\cygwin\bi- -ash: /usr/bin/ls: Invalid argument
```

使用Red Hat 9的人则会看到如下的信息：

```
/bin/ls: argument list too long
```

虽然32K的限制听起来好像比较长的，但是当你的项目在100个子目录中包含了3000个文件，而你想要对它们进行操作时，就会超过此限制。

有两项基本的操作会让你陷入此泥潭：使用shell工具扩展某个基本的值，或是使用make本身为一个变量设定很长的值。举例来说，假设我们想在单一命令行上编译所有的源文件：

```
compile_all:
        $(JAVAC) $(wildcard $(addsuffix /*.java,$(source_dirs)))
```

make变量source_dirs可能只包含了几百个单词，但是为Java文件添加通配符以及使用wildcard扩展它之后，此文件列表很容易就会超过系统的命令行长度限制。顺便说一下，make本身并没这样的限制，只要有足够的存储空间，make就会让你使用你会用到的任何数据量。

当发现自己身处于此状况时，你会觉得仿佛坠入了迷雾中。举例来说，你可能会想要使用xargs来解决上面的问题，因为xargs可以根据系统的长度限制来分割其参数：

```
        echo $(wildcard $(addsuffix /*.java,$(source_dirs))) | \
        xargs $(JAVAC)
```

可惜，这么做只是将命令行的长度限制问题从javac命令行转移到echo命令行。同样地，我们也无法使用echo或printf将数据写入一个文件（假定编译器可以从一个文件读取文件列表）。

处理此状况的方法就是避免一次将文件列表定全。我们可以使用shell一次处理一个目录：

```
compile_all:
        for d in $(source_dirs); \
        do                       \
            $(JAVAC) $$d/*.java; \
        done
```

我们还可以将文件列表使用管道转至xargs，以较少的执行次数来完成相同的工作：

```
compile_all:
        for d in $(source_dirs); \
        do                       \
            echo $$d/*.java;     \
        done |                   \
        xargs $(JAVAC)
```

可惜，这些命令脚本也无法在编译期间正确处理错误。一个比较好的做法就是存储完整的文件列表并将它提供给编译器，如果编译器可以从一个文件读取它的参数的话。Java编译器支持此功能：

```
compile_all: $(FILE_LIST)
        $(JAVA) @$<

.INTERMEDIATE: $(FILE_LIST)
$(FILE_LIST):
        for d in $(source_dirs); \
        do                       \
            echo $$d/*.java;     \
        done > $@
```

注意for循环中的微妙错误。如果目录列表中的任何目录并未包含Java文件，字符串*.java将会被包含在文件列表中，于是Java编译器就会产生“File not found”的错误信息。我们可以通过设定nullglob选项的方式，让bash将不匹配的文件名模式扩展成空字符串。

```
compile_all: $(FILE_LIST)
        $(JAVA) @$<

.INTERMEDIATE: $(FILE_LIST)
$(FILE_LIST):
        shopt -s nullglob;       \
        for d in $(source_dirs); \
        do
            echo $$d/*.java;
        done > $@
```

许多项目都必须建立文件列表。下面的宏中包含了一个用来产生文件列表的bash脚本。这个宏的第一个参数就是所要切换的根目录，列表中的所有文件都将会相对于这个根目录。第二个参数就是用来搜索匹配文件的目录列表。第三和第四个参数则是选项，代表文件的扩展名。

```
# $(call collect-names, root-dir, dir-list, suffix1-opt, suffix2-opt)
define collect-names
  echo Making $@ from directory list...
  cd $1;                                                    \
  shopt -s nullglob;                                        \
  for f in $(foreach file,$2,'$(file)'); do                 \
    files=( $$f$(if $3,/*.{$3$(if $4,$(comma)$4)}) );       \
```

```
    if (( $${#files[@]} > 0 ));                                  \
    then                                                         \
      printf '"%s"\n' $${files[@]};                              \
    else :; fi;                                                  \
  done
endef
```

下面是用来创建图像文件列表的模式规则：

```
%.imagec:
        @$(call collect-names,$(SOURCE_DIR),$^,gif,jpeg) > $@
```

这个宏的执行过程会被隐藏起来，因为此脚本太长了，而且需要转贴此内容的机会并不多。目录列表是由必要条件提供。切换至根目录后，此脚本允许空文件的匹配（null globbing）。其余的是用来处理我们想要搜索的每个目录的 for 循环。文件搜索表达式就是参数 $2 所传入的单词列表。此脚本将会使用单引号来保护文件列表中的单词，因为其中可能会包含shell特殊字符。尤其是，有些语言（比如Java）的文件名可能会包含美元符号：

```
for E in $(foreach file,$2,'$(file)'); do
```

我们将会通过将文件名匹配的结果填入 files 数组来搜索一个目录。如果 files 数组中包含了任何的元素，我们就会使用printf为其所输出的每个单词添加一个换行符号。使用数组的好处是，可让宏正确处理内置空格的路径。这也是为何printf需要为文件名加上双引号的原因。

文件列表由下面这一行所产生：

```
files=( $$f$(if $3,/*.{$3$(if $4,$(comma)$4)}) );
```

$$f 就是此宏的目录或文件参数。接下来的表达式是make的 if 函数，用来测试第三个参数是否为非空值，这就是你用来实现可选用参数的方法。如果第三个参数是空的，它就会假设第四个参数也是这样。在此情况下，用户所传递的文件应该就包含在文件列表中。这让此宏可以在通配模式不相符的时候，为所有文件建立列表。如果提供了第三个参数，if 函数会为根文件添加 /*.{$3}，如果提供了第四个参数，它会在 $3 之后添加 ,$4。注意我们为通配模式插入逗号的方法：我们会使用comma这个make变量来规避解析器的处理，否则该逗号将会被解译成if函数的参数的分隔符。comma变量的定义很简单：

```
comma := ,
```

前面所有的for循环也都会受到命令行长度的限制，因为它们都进行了通配符扩展的动作。差别在于，针对单一目录的内容进行通配符扩展的动作不太可能会超过此限制。

你可能会问：如果有一个make变量包含了很长的文件列表，要如何处理？嗯，这实在很麻烦。我只找到了两种可以把一个内容很长的make变量传递给subshell的方法。第一种方法就是过滤变量的内容，只将变量内容的子集传递给任何一次的subshell调用。

```
compile_all:
        $(JAVAC) $(wordlist 1, 499, $(all-source-files))
        $(JAVAC) $(wordlist 500, 999, $(all-source-files))
        $(JAVAC) $(wordlist 1000, 1499, $(all-source-files))
```

你也可以使用filter函数，不过会有比较多的不确定性，因为这么做所选出的文件数目与你所选用的模式空间的分布有关。我们可以根据英文字母来选择模式：

```
compile_all:
        $(JAVAC) $(filter a%, $(all-source-files))
        $(JAVAC) $(filter b%, $(all-source-files))
```

或是根据文件名本身的特征来选取模式。

请注意，要进一步将此做法自动化并不容易。我们试着把这个以英文字母为模式的做法封装在foreach循环中：

```
compile_all:
        $(foreach l,a b c d e ;,                        \
          $(if $(filter $l%, $(all-source-files)),      \
            $(JAVAC) $(filter $l%, $(all-source-files));))
```

但这么做并不可行。make会将它扩展成单行文本，因此会掺杂行长度限制的问题。我们可以改用eval：

```
compile_all:
        $(foreach l,a b c d e ;,                        \
          $(if $(filter $l%, $(all-source-files)),      \
            $(eval                                      \
              $(shell                                   \
                $(JAVAC) $(filter $l%, $(all-source-files));))))
```

这么做可行是因为eval会立即执行shell命令，而且会被扩展成空值。所以foreach循环会被扩展成空值。它的问题是错误报告在此语境中是无意义的，所以编译错误将不会被正确传送至make。

wordlist的做法更糟。由于make的数值能力有限，所以你无法将wordlist封装在循环中。一般来说，对于无限长的文件列表，几乎找不到能够处理它的方法。

第二部分

高级与特别的议题

在第二部分中，我们将以面向问题的观点来审视make。将make应用在实际问题（例如多个目录的编译、新的程序语言、移植与效能、调试）中的方法通常不是很直观。接下来我们将会探讨这些问题并且提供若干复杂的范例。

第六章

大型项目的管理

何谓大型项目？嗯，就是需要一组开发人员，可以在多种架构上运行，有若干已发行的版本需要维护的项目。当然，并非一定要这样才叫做大型项目。一个只能在单一平台上运行、包含了100万行C++程序代码的抢先版（prerelease）软件仍属大型项目。不过软件极少永远停留在抢先版的阶段。而且，如果软件成功发行了，迟早会有人要求能够在另一个平台上运行该软件，所以大多数的大型软件系统在不久以后看起来都会非常类似。

我们通常会对大型软件项目进行简化的动作：将它们划分成几个主要组件，通常会被归纳成不同的程序模块、程序库或是此两者皆有。这些组件往往会被存放在它们自己的目录下，并由它们自己的*makefile*来管理。为整个系统编译所有组件的一个方法是，通过顶层的*makefile*以适当的顺序调用每个组件的*makefile*。此做法称为递归式（recursive）make，因为顶层的*makefile*会对每个组件的*makefile*递归地调用make。递归式make技术常用来处理逐组件（component-wise）的编译工作。Peter Miller于1998年建议，应该使用单一*makefile*从每个组件目录引入信息，以避免递归make所导致的许多问题（注1）。

一旦项目编译好它的组件之后，它最后会在结构上遇到比较大的问题。这包括对项目的多个版本进行开发、支持多个平台、对源文件和二进制文件提供有效的访问以及进行自动化的编译工作。我们将会在本章后半部探讨这些问题。

注1：　参见Miller, P.A.于1998年在AUUG公司的AUUGN杂志上所发表的"Recursive Make Considered Harmful"（递归式make被认为是有害的）。实际的内容可到*http://aegis.sourceforge.net/auug97.pdf*上获取。

递归式 make

递归make（recursive make）背后的动机很简单：make在单一目录（或一小群目录）中可以运作得非常好，但是当目录的数量增长时，事情会变得比较复杂。因此，当我们要使用make来编译一个大型项目时，我们可以为每个目录编写一个简单的、各自独立的*makefile*，然后分别地执行它们。虽然我们可以使用脚本工具来完成此事，不过还是使用make比较有效，因为在较高的层次上还涉及了依存关系。

例如，假设我有一个mp3 player应用程序。逻辑上，它可以被划分成若干组件：用户界面、编解码器（codec）以及数据库管理。它们分别可用三个程序库来表示：*libui.a*、*libcodec.a*和*libdb.a*。将这些组件紧凑地放一起就可以组成这个应用程序。图6-1展示了将这些组件直接对应到一个文件结构的样子。

图6-1：MP3播放器的文件布局图

一个比较传统的布局，就是将应用程序的main函数放在顶层目录，而不要放在子目录*app/player*中。为了在顶层建立较明确的布局，我倾向于把应用程序代码放在它自己的

目录中，这样我就可以使用额外的模块来扩充系统。举例来说，如果我们稍后选择加入独立的编目应用程序（cataloging application），你就可以明确地将它放在 *app/catalog* 之下。

如果 *lib/db*、*lib/codec*、*lib/ui* 和 *app/player* 等目录中都包含了一个 *makefile*，那么顶层 *makefile* 的工作就是去调用它们。

```
lib_codec := lib/codec
lib_db    := lib/db
lib_ui    := lib/ui
libraries := $(lib_ui) $(lib_db) $(lib_codec)
player    := app/player

.PHONY: all $(player) $(libraries)
all: $(player)

$(player) $(libraries):
        $(MAKE) --directory=$@
```

在顶层 *makefile* 中，你会看到一个规则在工作目标上列出了所有的子目录，它的动作就是对每个子目录调用 make：

```
$(player) $(libraries):
        $(MAKE) --directory=$@
```

在 *makefile* 文件中，MAKE 变量应该总是用来调用 make 程序。make 程序一看到 MAKE 变量就会把它设成 make 的实际路径，所以递归调用中的每次调用都会使用同一个可执行文件。此外，当命令行选项 --touch（-t）、--just-print（-n）和 --question（-q）被使用时，包含 MAKE 变量的每一行将会受到特别的处理。稍后我们将会在“命令行选项”一节中进一步探讨此议题。

由于这些“工作目标目录”被设成 .PHONY 的必要条件，所以即使工作目标已经更新，此规则仍旧会进行更新的动作。使用 --directory（-C）选项的目的是要让 make 在读取 *makefile* 之前先切换到相应的“工作目标目录”。

尽管这个规则有些难以捉摸，不过它可以避免下面这个较简单的命令脚本所发生的若干问题：

```
all:
        for d in $(player) $(libraries); \
        do                                \
          $(MAKE) --directory=$$d;        \
        done
```

这个命令脚本无法正确地将错误传送到上层的 make，它也无法让 make 以并行的方式对任何子目录进行编译的动作。我们将会在第十章探讨 make 的这个特性。

当make在建立依存图的时候找不到程序库与app/player工作目标之间的依存关系时，这意味着，建立任何程序库之前，make将会先执行app/player目录中的*makefile*。显然，这将会导致失败的结果，因为应用程序的链接需要程序库。为解决此问题，我们会提供额外的依存信息。

```
$(player): $(libraries)
$(lib_ui): $(lib_db) $(lib_codec)
```

我们在此处做了如下的描述：运行*app/player*目录中的*makefile*之前必须先运行程序库子目录中的*makefile*。此外，编译*lib/ui*目录中的程序代码之前必须先编译*lib/db*和*lib/codec*目录中的程序库。这么做可确保任何自动产生的程序代码（像*yacc/lex*文件）在*lib/ui*目录中的程序代码被编译之前就已经产生出来了。

更新必要条件的时候，会引发微妙的次序问题。如同所有的依存关系，更新的次序取决于依存图的分析结果，但是当工作目标的必要条件出现在同一行时，GNU make将会以从左至右的次序进行更新。例如：

```
all: a b c
all: d e f
```

如果不存在其他依存关系，这6个必要条件的更新动作可以是任何次序（例如"d b a c e f"），不过GNU make将会以从左至右的次序来更新出现在同一行的必要条件，这会产生如下的更新次序："a b c d e f"或"d e f a b c"。但是不要因为之前这么做更新的次序是对的，就以为每次这么做都是对的，而忘了提供完整的依存信息。最后，依存分析可能会产生不同的次序而引发一些问题。所以，如果有一组工作目标需要以特定的次序进行更新时，你就必须提供适当的必要条件来实现正确的次序。

当顶层的*makefile*被运行时，我们将会看到如下的结果：

```
$ make
make --directory=lib/db
make[1]: Entering directory `/test/book/out/ch06-simple/lib/db'
Update db library...
make[1]: Leaving directory `/test/book/out/ch06-simple/lib/db'
make --directory=lib/codec
make[1]: Entering directory `/test/book/out/ch06-simple/lib/codec'
Update codec library...
make[1]: Leaving directory `/test/book/out/ch06-simple/lib/codec'
make --directory=lib/ui
make[1]: Entering directory `/test/book/out/ch06-simple/lib/ui'
Update ui library...
make[1]: Leaving directory `/test/book/out/ch06-simple/lib/ui'
make --directory=app/player
make[1]: Entering directory `/test/book/out/ch06-simple/app/player'
Update player application...
make[1]: Leaving directory `/test/book/out/ch06-simple/app/player'
```

当make发觉它正在递归调用另一个make时，它会启用--print-directory（-w）选项，这会使得make输出Entering directory（进入目录）和Leaving directory（离开目录）的信息。当--directory（-C）选项被使用时，也会启用这个选项。你还可以在每一行看到，MAKELEVEL这个make变量的值加上方括号之后被一起输出。在这个简单的例子里，每个组件的*makefile*只会输出组件正在更新的信息，而不会真的去更新该组件。

命令行选项

递归式make最初只是一个简单的构想，但很快变得非常复杂。最完美的递归式make方式是：尽管系统中有许多*makefile*，但是它的行为就如同只有一个*makefile*一样。要达到这样的程度几乎不可能，所以妥协是免不了的。当我们知道命令行选项的处理方式之后，这个微妙的问题将会变得较为明确。

假设我们为mp3 player的头文件加入了注释。我们不想让make因为头文件被修改而重新编译所有的源文件，我们发现可以执行make --touch来让相应文件的时间戳变成已更新的。当我们对顶层的*makefile*执行make --touch时，我们也希望make能够通过下层的make来让所有应该处理的文件变成已更新的。让我们来看看这是怎么办到的。

通常，当你在命令行上指定--touch选项时，make将会暂停规则的正常处理过程，它会去搜索依存图，从中选出未被.PHONY标注过的工作目标以及必要条件，然后对工作目标执行touch，好让它们变成已更新的。因为我们的子目录都被.PHONY标注过，所以它们都会被忽略掉（像一般文件那样变更它们的时间戳没有任何意义）。但是我们并不想让那些工作目标被忽略掉，我们想要运行它们的命令脚本。为了做对的事情，make将会自动以+修饰符来标示包含MAKE的任何行，这会使得make在运行下层的make时忽略--touch选项。

当make运行下层的make时，还必须将--touch标记传递给子进程。它可以通过MAKEFLAGS变量来完成此事。当make启动时，它会自动将大部分的命令行选项追加到MAKEFLAGS中，例外是--directory（-C）、--file（-f）、--old-file（-o）和--new-file（-W）等选项。然后MAKEFLAGS变量会被导出到环境中，让下层的make启动时能够读取。

有了这样的特别支持，下层make的行为几乎就是你想要的那样。$(MAKE)的递归执行以及MAKEFLAGS的特别处理将会被应用在--touch（-t）选项上，也会被应用在选项--just-print（-n）和--question（-q）上。

传递变量

正如之前提到的，变量可以通过环境传递给下层的make，而且可以使用export和unexport指令加以控制。通过环境传递的变量将会成为默认值，不过对此类变量所进行的任何赋值动作将会覆盖掉此默认值。使用--environment-overrides(-e)选项可让环境变量覆盖掉当时的赋值动作。你也可以明确地针对特定的赋值动作使用override指令来覆盖掉环境变量，即使当时使用了--environment-overrides选项：

```
override TMPDIR = ~/tmp
```

命令行上所定义的变量，如果使用了正确的shell语法，将会被自动导出到环境中。一个变量的名称中如果只使用了字母、数字和下划线，将会被视为语法正确。命令行上所进行的变量赋值动作会跟着命令行选项一起被存放在MAKEFLAGS变量里。

错误处理

当递归式make发生错误时，要如何处理呢？事实上，跟一般的编译方式差别不会太大。当make收到错误状态时，会终止它的处理过程并返回值为2的结束状态。然后上层的make会结束执行，错误状态会遍及整个“递归式make”进程树（process tree）。如果顶层的make使用了--keep-going(-k)选项，这个选项将会如往常般被传递到下层的make。下层make的处理方式如同往常那样，它会跳过当前的工作目标，前进到下一个工作目标（条件是此工作目标并未以发生错误的工作目标作为它的必要条件）继续处理。

举例来说，如果我们的mp3 player程序在lib/db组件中遇到了编译错误，lib/db的编译工作将会因此而结束，并将值为2的状态返回顶层的*makefile*。如果我们使用了--keep-going(-k)选项，顶层的*makefile*将会前进到下一个无关的工作目标lib/codec继续处理。完成该工作目标之后会忽略它的结束状态，make会返回值为2的结束状态——由于lib/db的失败已经没有其他工作目标可供处理了。

--question(-q)选项的行为非常类似。如果某工作目标尚未更新，此选项会使得make返回值为1的结束状态，否则会返回0。当你将它应用在具有树状结构的*makefile*上时，make会开始递归地运行*makefile*，直到它可以判断该项目是否已经更新。一发现尚未更新的文件，make会终止当前所运行的make，并从递归操作返回。

建立其他工作目标

基本的“建立工作目标”（build target）是任何编译系统不可或缺的，不过我们还需要其他工作目标的支持，像clean、install、print等。因为这些都属于.PHONY工作目标，之前曾提到此技术的运作不是很好。

例如，下面这几种是有问题的做法，像：

```
clean: $(player) $(libraries)
        $(MAKE) --directory=$@ clean
```

或是：

```
$(player) $(libraries):
        $(MAKE) --directory=$@ clean
```

第一种做法的问题出在，它的必要条件将会触发$(player)和$(libraries)所在的*makefile*的默认目标进行编译动作，而不是clean工作目标的编译动作。第二种做法的问题出在，已经存在的工作目标具有不同的命令脚本。

还有一种做法会用到shell的for循环：

```
clean:
        for d in $(player) $(libraries); \
        do                               \
          $(MAKE) --directory=$$f clean; \
        done
```

这个for循环也无法完全解决稍早所提到的问题，不过它（以及之前有问题的例子）却让我们找到了如下这个解决方案

```
$(player) $(libraries):
        $(MAKE) --directory=$@ $(TARGET)
```

对递归式make的那一行加上$(TARGET)变量，并且在make命令行上设定TARGET变量，我们就可以随意为下层的make加上默认目标：

```
$ make TARGET=clean
```

可惜，这么做并无法在顶层的*makefile*上调用$(TARGET)。通常并不需要这么做，因为顶层的*makefile*并没有什么事可以做，但如果有此需要的话，我们可以加入另一个受到if保护的make调用：

```
$(player) $(libraries):
        $(MAKE) --directory=$@ $(TARGET)
        $(if $(TARGET), $(MAKE) $(TARGET))
```

现在只要在命令行上设定TARGET变量，我们就可以调用clean工作目标（或是任何其他的工作目标）。

跨越makefile的依存关系

make对命令行选项的特殊支持以及把环境变量作为沟通管道，意味着递归式make技术可以协助我们把事情做好。但是我们为什么要担心事情会变得更复杂呢？

这些独立的*makefile*将会被递归的$(MAKE)链接在一起，但它只记录了最表面的顶层链接。可惜，这些微妙的依存关系通常会被隐藏在某些目录中。

举例来说，假设*db*模块引入了一个以yacc为基础的解析器，以便导入或导出音乐数据。如果*ui*模块*ui.c*引入了自动产生的yacc标头，那么这两个模块之间便存在着依存关系。如果我们正确建立此依存关系，那么每当语法标头被更新时，make应该就会重新编译我们的*ui*模块。使用稍早所提到的自动产生依存关系的技术，要完成此事并不难。但如果yacc文件本身被修改呢？在此状况下，当*ui*的*makefile*在运行时，一个正确的*makefile*应该知道，编译*ui.c*之前必须先运行yacc，以便产生解析器和标头。经过递归式make的分解之后，并不会发生此事，因为用来运行yacc的规则和依存关系被存放在*db*的*makefile*中，而不是放在*ui*的*makefile*里。

在此状况下，我们所能做的只是在*ui*的*makefile*运行之前，总是先运行*db*的*makefile*。这个较高级的依存关系必须以手动编码。尽管我们的能力足以在*makefile*的第一个版本中手动完成此事，但是一般而言，这会存在非常难以维护的问题。当该代码被编写和修改之后，顶层的*makefile*将无法正确记录模块间的依存关系。

继续这个例子，如果*db*中的yacc语法被更新，而且*ui*的*makefile*在*db*的*makefile*之前被运行（因为你并非通过顶层的*makefile*而是直接运行它），那么*ui*的*makefile*根本不可能知道*db*的*makefile*中有依存关系尚未被满足（必须运行yacc以便更新头文件）。在此状况下，*ui*的*makefile*将会以旧版的yacc标头来编译它的程序。如果源文件中定义并引用了新的符号，那么就会发生编译错误的状态。因此，递归式make本来就比单一*makefile*还容易出错。

当程序代码产生器被大量使用时，问题会变得更糟。假设*ui*中使用了RPC stub产生器，并且在*db*中引用了标头。现在我们遇到了交互引用的问题。要解决此问题，make可能需要先进入*db*以产生yacc标头，然后进入*ui*以产生RPC stub，接着进入*db*来编译文件，最后进入*ui*以完成编译的程序。为了完成项目中源文件的建立和编译而反复进入各组件的次数，取决于程序代码的结构以及用来创建它的工具。这类相互引用的问题常见于复杂的系统中。

在真实世界中的*makefile*通常没有那么复杂。为了确保所有文件都会被更新，当顶层的*makefile*中有命令被执行时，所有下层的*makefile*都应该被执行。请注意，这正好就是mp3 player的*makefile*的做法：当顶层的*makefile*运行时，所有下层的*makefile*都会无条件被运行。在比较复杂的状况下，*makefile*会被反复运行以便在编译之前先完成所有程序代码的产生动作。通常这种反复运行的做法完全是在浪费时间，不过偶尔还是有此需要。

避免重复的代码

这个应用程序的目录结构包含了三个程序库，这些程序库的*makefile*都非常类似。这是合理的，因为尽管这三个程序库在应用程序中的功能各不相同，但是它们皆产生自类似的命令。这种将大型项目分解成各个组件的典型做法，将会导致许多类似的*makefile*以及大量的重复代码。

跟重复的程序代码一样，出现重复的*makefile*代码也是很糟糕的。这会增加软件的维护成本而且会引发更多缺陷，也会让人更难了解其中的算法以及找到其中的细微变动。所以我们会尽量避免重复的代码出现在我们的*makefile*中。达到此目标的最简单的方法就是将*makefile*里共同的部分移到一个共同的引入文件中。

例如，*codec*的*makefile*中包含如下的内容：

```
lib_codec    := libcodec.a
sources      := codec.c
objects      := $(subst .c,.o,$(sources))
dependencies := $(subst .c,.d,$(sources))

include_dirs := . ../../include
CPPFLAGS     += $(addprefix -I ,$(include_dirs))
vpath %.h $(include_dirs)

all: $(lib_codec)

$(lib_codec): $(objects)
        $(AR) $(ARFLAGS) $@ $^

.PHONY: clean
clean:
        $(RM) $(lib_codec) $(objects) $(dependencies)

ifneq "$(MAKECMDGOALS)" "clean"
  include $(dependencies)
endif

%.d: %.c
        $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -M $< |
        sed 's,\($*\.o\) *:,\1 $@: ,' > $@.tmp
        mv $@.tmp $@
```

你可以在*db*和*ui*的*makefile*中看到几乎重复的代码，只有程序库本身的名称以及程序库所包含的源文件不同而已。当重复的代码被移往*common.mk*之后，我们就可以把*codec*的*makefile*缩减成下面这样：

```
library := libcodec.a
sources := codec.c

include ../../common.mk
```

下面就是我们移往*common.mk*这个共享引入文件的内容：

```
MV            := mv -f
RM            := rm -f
SED           := sed

objects       := $(subst .c,.o,$(sources))
dependencies  := $(subst .c,.d,$(sources))
include_dirs  := . ../../include
CPPFLAGS      += $(addprefix -I ,$(include_dirs))

vpath %.h $(include_dirs)
.PHONY: library
library: $(library)

$(library): $(objects)
        $(AR) $(ARFLAGS) $@ $^

.PHONY: clean
clean:
        $(RM) $(objects) $(program) $(library) $(dependencies)
$(extra_clean)

ifneq "$(MAKECMDGOALS)" "clean"
  -include $(dependencies)
endif

%.c %.h: %.y
        $(YACC.y) --defines $<
        $(MV) y.tab.c $*.c
        $(MV) y.tab.h $*.h

%.d: %.c
        $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -M $< |        \
        $(SED) 's,\(($*\.o\) *:,\1 $@: ,' > $@.tmp
        $(MV) $@.tmp $@
```

include_dirs变量的值，原本在每个*makefile*中皆不同，现在在每个*makefile*中都一样了。这是因为我们对路径做了变更，让所有的程序库使用相同的引入路径。

*common.mk*文件甚至为程序库的引入文件使用了默认目标（default goal）。原本的*makefile*使用的是all这个默认目标。对那些需要为它们的默认目标指定不同必要条件的非程序库的*makefile*来说，这将会引发某些问题。所以*common.mk*会使用library这个默认目标。

请注意，因为*common.mk*包含了工作目标，所以在非程序库的*makefile*中，它必须在默认目标之后被引入。还请注意，clean的脚本引用了program、library和extra_clean等变量。在程序库的*makefile*中，program变量是空的，在程序的*makefile*中，library变量是空的。至于extra_clean变量则是为*db*的*makefile*特别加进来的：

```
library     := libdb.a
sources     := scanner.c playlist.c
extra_clean := $(sources) playlist.h

.SECONDARY: playlist.c playlist.h scanner.c

include ../../common.mk
```

使用以上所提到的技术，将可让重复的代码减到最少。当有更多的*makefile*代码被移到共同的*makefile*中时，它将会逐渐成为整个项目通用的*makefile*。当你要针对每个目录修改这个通用的*makefile*时，make 变量以及用户自定义函数可作为你的自定义点(customization point)。

非递归式 make

就算不使用递归式 make 的技术也可以管理具有多个目录的项目。此处的差别在于*makefile*所操作的源文件存放在多个目录中。为了适应这样的布局方式，当引用到子目录里的文件时，文件名称中必须包含（绝对或相对）路径。

通常，用来管理大型项目的*makefile*具有许多工作目标，项目中的每个模块各一个。以我们的mp3 player为例，每个程序库以及每个应用程序都需要一个工作目标。对于模块的集合，比如所有程序库的集合，使用假想工作目标也很有用。*makefile*的默认目标通常就是建立所有的工作目标，以及建立说明文件与运行测试程序。

非递归式(nonrecursive) make的最直接用法，就是在单一的*makefile*中包含工作目标、目标文件引用以及依存关系。这么做通常无法满足熟悉递归式make的开发者，因为目录里文件的相关信息被集中放在单一文件中，然而源文件本身却分布在文件系统各处。为了解决这个问题，Miller的文章对非递归式make提供了以下建议：为每个目录使用一个内含文件列表和模块专属规则的make引入文件，并在顶层*makefile*中引入这些下层的*makefile*。

在例6-1中可以看到mp3 player的*makefile*从每个子目录引入模块层的*makefile*。在例6-2中可以看到一个模块层的引入文件。

例6-1：一个非递归的makefile

```
# 到每个模块收集这四个变量的信息
# 初始设定这四个变量为简单变量
programs     :=
sources      :=
libraries    :=
extra_clean  :=

objects      = $(subst .c,.o,$(sources))
dependencies = $(subst .c,.d,$(sources))
```

```
include_dirs := lib include
CPPFLAGS     += $(addprefix -I ,$(include_dirs))
vpath %.h $(include_dirs)

MV  := mv -f
RM  := rm -f
SED := sed

all:

include lib/codec/module.mk
include lib/db/module.mk
include lib/ui/module.mk
include app/player/module.mk

.PHONY: all
all: $(programs)

.PHONY: libraries
libraries: $(libraries)

.PHONY: clean
clean:
        $(RM) $(objects) $(programs) $(libraries) \
              $(dependencies) $(extra_clean)

ifneq "$(MAKECMDGOALS)" "clean"
  include $(dependencies)
endif

%.c %.h: %.y
        $(YACC.y) --defines $<
        $(MV) y.tab.c $*.c
        $(MV) y.tab.h $*.h

%.d: %.c
        $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -M $< | \
        $(SED) 's,\($(notdir $*)\.o\) *:,$(dir $@)\1 $@: ,' > $@.tmp
        $(MV) $@.tmp $@
```

例 6-2：非递归的 makefile 的 lib/codec 引入文件

```
local_dir  := lib/codec
local_lib  := $(local_dir)/libcodec.a
local_src  := $(addprefix $(local_dir)/,codec.c)
local_objs := $(subst .c,.o,$(local_src))

libraries  += $(local_lib)
sources    += $(local_src)

$(local_lib): $(local_objs)
        $(AR) $(ARFLAGS) $@ $^
```

因此，模块特有的信息会被包含在模块目录中的引入文件里。顶层的 *makefile* 只会包含模块列表以及 include 指令。让我们进一步研究 *makefile* 和 *module.mk*。

每个*module.mk*引入文件会将本地的程序库名称添加到libraries变量中，以及将本地的源文件添加到sources变量中。具local_前缀的变量用来保存常数或用来避免重复求值动作。请注意，每个引入文件都会重复使用具local_前缀的相同变量名称，因此，它会使用简单变量（以:=赋值的变量）而不会使用递归变量。这样，当这些*makefile*被组合在一起时，才不会对同名的变量造成影响。程序库名称以及源文件会使用相对路径，正如稍早所说的。最后，引入文件定义了一个用来更新本地程序库的规则。在这个规则中使用具local_前缀的变量一点问题也没有，因为规则中的工作目标部分以及必要条件部分都会立即进行求值动作。

在顶层的*makefile*中，前四行所定义的变量是用来收集每个模块专属的文件信息。这些变量必须是简单变量，因为每个模块将会以相同的本地变量名称对它们进行添加的动作：

```
local_src  := $(addprefix $(local_dir)/,codec.c)
...
sources    += $(local_src)
```

如果有一个递归变量被用于sources，那么它最终的值只会是local_src反复出现的最后的值。请注意，即使要将这些简单变量初始化为空值，也要明确地进行赋值动作，因为任何未赋值的变量在默认情况下会被视为递归的形式。

下一个部分，是从sources变量中找出目标文件列表objects以及依存文件列表。请注意，这两个变量是递归的形式。因为当make读到该处时，sources变量是空的，直到稍后make读进引入文件后sources变量才会有值。在这个*makefile*中，虽然把这些变量的定义移到引入文件的后面并且变为简单变量是绝对可行的，不过若能集中存放这些基本的文件列表（例如sources、libraries、objects），可让*makefile*变得易于了解以及符合实际需要。然而，变量之间的交互引用则需要使用递归变量。

接下来，为了处理C语言的引入文件，我们会设定CPPFLAGS。这让编译器能够找到头文件。设定CPPFLAGS的时候我们会使用附加运算符，因为我们并不知道变量的值是否真的是空的，它可能会被设定成命令行选项、环境变量或是其他的make结构。vpath指令可让make找到放在其他目录里的头文件。include_dirs变量可用来避免引入文件列表中出现重复的项目。

为mv、rm和sed定义变量可避免将程序的路径名称固定在*makefile*里。注意变量的大小写。我们将会依照make在线手册所建议的惯例：在*makefile*内部设定的变量使用小写，能够由命令行来设定的变量使用大写。

事情越来越有趣了。接着，我们想要以具体规则all作为默认目标。麻烦的是，all的必要条件是programs变量，make会立即对这个变量进行求值的动作，不过它的值却是在读取模块引入文件的时候设定的。所以，我们必须在make读取引入文件之前先定义

all工作目标。可惜，被引入的模块包含了工作目标，其中第一个工作目标将会被视为值的目标。解决此问题的方法是先不要为all工作目标指定必要条件，等到取得引入文件之后，再为all指定必要条件。

这个*makefile*的其余部分就是你在前一个例子所看到的内容，不过你现在应该注意make应用隐含规则的方法。我们的源文件现在存放在各个子目录里，当make想要应用标准的%.o:%.c规则时，必要条件将会是一个具有相对路径的文件名，例如*lib/ui/ui.c*。make会自动将此相对路径传播至工作目标文件，而且会试图更新*lib/ui/ui.o*。因此，make会自动地做正确的事。

最后还有一个小问题：尽管make会正确处理文件名路径，不过并非*makefile*所用到任何工具都是如此。尤其是当你使用gcc时，自动产生的依存文件将不会包含工作目标目标文件（target object file）的相对路径。也就是说，gcc -M的输出将会像这样：

```
ui.o: lib/ui/ui.c include/ui/ui.h lib/db/playlist.h
```

而不是我们所预期的：

```
lib/ui/ui.o: lib/ui/ui.c include/ui/ui.h lib/db/playlist.h
```

这将会中断头文件必要条件（header file prerequisite）的处理。为了修正此问题，我们会修改sed命令以便加入相对路径信息：

```
$(SED) 's,\($(notdir $*)\.o\) *:,$(dir $@)\1 $@: ,'
```

"调整*makefile*以便处理各种工具的异常行为"是使用make的时候一般会考虑到的内容。具可移植性的*makefile*通常很复杂，这是因为其中必须用到各种工具的异常行为。

现在我们拥有了一个合用的、非递归的*makefile*，不过它却存在维护的问题。各个*module.mk*引入文件几乎都一样。当其中有一个引入文件需要变更时，很可能所有引入文件都需要变更。对于小型项目来说，像我们的mp3 player，这将会令人厌烦不已。对于具有数百个引入文件的大型项目来说，这可能会让开发工作无法进行下去。这个时候若能使用一致的变量名称以及调整引入文件的内容，将可协助我们修正这些问题。下面是重构之后*lib/codec*引入文件的样子：

```
local_src := $(wildcard $(subdirectory)/*.c)

$(eval $(call make_library, $(subdirectory)/libcodec.a, $(local_src)))
```

此处不会以名称来指定源文件，我们会假设自己要重新编译目录中所有的*.c*文件。我们会使用make-library函数为每个引入文件来完成大量的工作。此函数将会被定义在项目的顶层*makefile*中：

```
# $(call make-library, library-name, source-file-list)
define make-library
  libraries += $1
  sources   += $2

  $1: $(call source-to-object,$2)
    $(AR) $(ARFLAGS) $$@ $$^
endef
```

这个函数会把程序库和源文件添加到相应的变量里去，然后定义具体规则以便建立程序库。注意，自动变量是如何以另一个美元符号来将$@和$^的求值动作延后到此规则执行的时候进行的。source-to-object 函数用来将源文件列表转换成相应的目标文件：

```
source-to-object = $(subst .c,.o,$(filter %.c,$1)) \
                   $(subst .y,.o,$(filter %.y,$1)) \
                   $(subst .l,.o,$(filter %.l,$1))
```

在这个 *makefile* 的前一个版本中，我们并未指出实际的解析器和扫描器源文件是 *playlist.y* 和 *scanner.l*。我们只是将这些源文件列成它们所产生的 *.c* 文件。这迫使我们必须明确列出它们以及使用一个额外的变量 extra_clean。为了修正这个问题，我们会直接在 sources 变量中使用 *.y* 和 *.l* 文件，并且让 source-to-object 函数为我们进行转换的工作。

除了修改 source-to-object，我们还需要另一个函数为我们处理 yacc 和 lex 的输出文件，好让 clean 工作目标能够进行适当的清理工作。generated-source 函数的参数是一份源文件列表，它的输出是一份中间文件列表：

```
# $(call generated-source, source-file-list)
generated-source = $(subst .y,.c,$(filter %.y,$1)) \
                   $(subst .y,.h,$(filter %.y,$1)) \
                   $(subst .l,.c,$(filter %.l,$1))
```

有了 subdirectory 函数的协助，可让我们省略 local_dir 变量。

```
subdirectory = $(patsubst %/makefile,%,                          \
                 $(word                                          \
                 $(words $(MAKEFILE_LIST)),$(MAKEFILE_LIST)))
```

正如“字符串函数”一节所说，我们可以从 MAKEFILE_LIST 中取出当前 *makefile* 的名称。只要使用 patsubst，我们就可以从最后被读进的 *makefile* 的名称中取出相对路径。这让我们能够省略 local_dir 变量，并且减少引入文件之间的差异。

最后一个优化动作（至少是对此例而言），就是使用 wildcard 来取得源文件列表。这个方法适合使用在源文件树（source tree）未受污染的大多数环境中。然而，我们所工作的项目并未提供这样的环境。为了“以防万一”，我们会将旧的程序代码存放在源文件树中。这让程序员必须在时间和精力上付出实际的代价，因为当程序员以全局搜索找到

很旧、不用的程序代码并将它放回原处之后，程序员将会对它进行维护的工作，而新进的程序员（或是老的程序员但对某个模块不熟悉）可能会想要对从未使用的程序代码进行编译或调试的动作。如果你使用的是现代化的源代码控制系统，比如CVS，根本不需要把不用的程序代码存放在源文件树中（因为它已经被存放在仓库中了），这个时候使用wildcard就没什么问题了。

include指令还可以进行以下的优化：

```
modules := lib/codec lib/db lib/ui app/player
. . .
include $(addsuffix /module.mk,$(modules))
```

对于较大型的项目而言，当模块列表增长到成百上千个模块时，即使这么做也可能会导致维护困难的问题。在此情况下，最好将modules模块定义成find命令：

```
modules := $(subst /module.mk,,$(shell find . -name module.mk))
. . .
include $(addsuffix /module.mk,$(modules))
```

这样我们就可以从find的输出中去除文件名的部分，所以modules变量会变得比模块列表还具通用性。如果不需要这么做，我们当然会省略subst和addsuffix，并直接把find的输出存入modules变量。例6-3完整地列出了这个非递归*makefile*的第二个版本。

例6-3：非递归makefile的第二个版本

```
# $(call source-to-object, source-file-list)
source-to-object = $(subst .c,.o,$(filter %.c,$1)) \
                   $(subst .y,.o,$(filter %.y,$1)) \
                   $(subst .l,.o,$(filter %.l,$1))

# $(subdirectory)
subdirectory = $(patsubst %/module.mk,%,                          \
                 $(word                                           \
                   $(words $(MAKEFILE_LIST)),$(MAKEFILE_LIST)))

# $(call make-library, library-name, source-file-list)
define make-library
  libraries += $1
  sources   += $2

  $1: $(call source-to-object,$2)
        $(AR) $(ARFLAGS) $$@ $$^
endef

# $(call generated-source, source-file-list)
generated-source = $(subst .y,.c,$(filter %.y,$1))        \
                   $(subst .y,.h,$(filter %.y,$1))        \
                   $(subst .l,.c,$(filter %.l,$1))
```

```
# 在这四个变量中收集来自每个模块的信息
# 于此处将它们初始设定为简单变量。
modules       := lib/codec lib/db lib/ui app/player
programs      :=
libraries     :=
sources       :=

objects       =  $(call source-to-object,$(sources))
dependencies =  $(subst .o,.d,$(objects))

include_dirs := lib include
CPPFLAGS      += $(addprefix -I ,$(include_dirs))
vpath %.h $(include_dirs)

MV  := mv -f
RM  := rm -f
SED := sed

all:

include $(addsuffix /module.mk,$(modules))

.PHONY: all
all: $(programs)

.PHONY: libraries
libraries: $(libraries)

.PHONY: clean
clean:
        $(RM) $(objects) $(programs) $(libraries) $(dependencies)
              $(call generated-source, $(sources))

ifneq "$(MAKECMDGOALS)" "clean"
  include $(dependencies)
endif

%.c %.h: %.y
        $(YACC.y) --defines $<
        $(MV) y.tab.c $*.c
        $(MV) y.tab.h $*.h

%.d: %.c
        $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -M $< | \
        $(SED) 's,\($(notdir $*)\.o\) *:,$(dir $@)\1 $@: ,' > $@.tmp
        $(MV) $@.tmp $@
```

为每个模块使用一个引入文件是相当可行的做法，而且具有某些优点，但是我并不确定是否值得这么做。就我个人的经验来说，对于一个大型的Java项目，在单一的顶层*makefile*中直接插入所有的*module.mk*文件，将能提供一个可行的解决方案。此项目包含了997个独立的模块，大约有两打的程序库以及半打的应用程序。分组后的程序代码具备若干*makefile*，这些*makefile*的长度大约有2500行。一个共享的引入文件包含了全局变量、用户自定义函数以及模式规则 —— 这是另一个长度为2500行的*makefile*。

无论你选择的是单一的*makefile*还是将模块信息取出放到各个引入文件，非递归式make对大型项目的编译而言，是一个可行的解决方案。它还可以用来解决你在递归式make中所遇到的许多典型问题。我所知道的唯一缺点是，当开发人员想要进行递归式make时，需要转移自己的思考模式（paradigm shift）。

大型系统的组件

今日有两种广受欢迎的开发风格：自由软件开发模式以及商业开发模式。

在自由软件开发模式中，每个开发者多半是靠自己。在此模式中，每个项目仅有一个*makefile*文件以及一个*README*文件，开发者自觉只会获得非常少的协助。而项目的负责人则想要把事情做好，以及想要从社群中获得大众的帮助，不过他们最想获得的是技术上的帮助以及让他们持续下去的动力。这并不是批评。就此观点来看，软件应该会被写得很好，而且不需要排定计划。

在商业开发模式中，存在着不同技术层次的开发者，他们都必须具备开发软件的能力，以便对开发成果作出贡献。如果有任何开发者不知道怎样完成工作，或是系统无法正确编译或运行，或是整个开发团队无所事事，都得付出昂贵的代价。要避免这些问题，应该由一个工程支持团队来管理开发的过程，这个团队必须协调编译程序、软件工具的配置、新的开发计划和维护工作以及版本的管理。在这样的开发环境中，效率是整个开发过程的重点所在。

商业开发模式通常会建立出复杂的编译系统。这主要是“为了降低软件开发成本必须增加程序员效率”的压力所导致的结果。这应该会让利润有所增加。然而，此处所提到的技术也同样适用于自由软件开发模式，只要它们有此需要。

这一节提到了许多功能，但没有太详细的讨论，也没有列举任何实例。那是因为其中有太多的功能与你所使用的语言和操作环境有关。我将会在第八章和第九章以实际的例子说明实现这些功能的方法。

需求

当然，每个项目以及每个工作环境的需求都有所不同。这一节我们将会广泛地说明在许多商业开发环境中常被考虑到的重要需求。

开发团队最常见的需求就是将源代码（source code）与二进制码（binary code）分开。也就是说，编译后所产生的目标文件应该放在独立的二进制文件树（binary tree）中。这能够让许多其他的功能被加入。独立的二进制文件树可以提供许多好处：

- 当大型的二进制文件树的位置可以被指定时，就会容易管理磁盘资源。
- 可以平行管理二进制文件树的各个版本。例如，一个源文件树可以产生经过优化（optimized）、含调试信息（debug）以及输出剖析信息（profiling）的二进制文件版本。
- 可以同时支持多个平台。一个适当实现的源文件树可同时为许多平台编译二进制文件。
- 开发人员可以调出（check out）部分的源文件树，并且让编译系统自动从引用源文件和二进制文件树（reference source and binary tree）“填入”短缺的文件。这不一定需要你将源文件与二进制文件分开，但是如果不分开的话，开发人员在编译系统的时候，很有可能会搞不清楚二进制文件被存放在哪里。
- 你可以用只供读取的访问权限来保护源文件树。这么做还可以提供额外的保证，因为编译结果可以真实反映仓库里的源代码。
- 某些工作目标，像 `clean`，如果能够将文件树作为一个单元来处理，而不用在文件树中搜索所要处理的文件，实现起来将会很容易（而且可以执行得非常快）。

以上各点，大部分本身就是重要的编译功能，也可能是项目的需求。

能够维护项目的引用编译结果（reference builds）通常是一个重要的系统功能。它的概念是：源文件的调出和编译只在夜间进行，这通常由一个 `cron` 任务来完成。因为这样所产生的源文件和二进制文件树就CVS上的源文件而言，是没有被修改过的，所以我会将它视为“引用源文件和二进制文件树”。此类文件树具有许多用处。

首先，需要检查源文件的程序人员和管理人员可以使用“引用源文件树”。这似乎没什么，但是当文件和发行版本的数量越来越多时，只是为了查看一个文件而去CVS仓库调出整个源文件树，可能极为不方便也相当不合理。此外，尽管CVS仓库浏览工具相当常见，不过它们通常并未提供可以轻易搜索整个源文件树的功能。这个时候，使用标记或是 `find/grep`（或者 `grep -R`）才是比较恰当的做法。

其次，而且是最重要的，“引用二进制文件树”是个未受污染的源文件编译结果（clean source builds）。开发人员每天早上开始工作的时候，他们就可以知道系统是否正常。如果有面向批处理的测试结构（batch-oriented testing framework）可用，你就可以对这个未受污染的编译结果进行自动的测试。开发人员每天都可以检查测试报告，判断系统的健康状态，而不用浪费自己的时间去执行测试的动作。如果开发人员只拥有源文件经修改过的版本，因为他不想浪费额外的时间去进行未受污染的调出和编译动作，那么所节省的成本将会打折扣。最后，开发人员可以对“引用编译结果”进行测试并比较特定组件的功能性。

"引用编译结果"还可以使用在其他地方。对于由许多程序库组成的项目而言，程序设计人员可以从来自"每夜编译结果"（nightly build）的预编译程序库（precompiled library）中把他们没有修改过的程序库链接到自己的应用程序上。这让他们能够通过"从自己本地的编译动作略过源文件树里的多数内容"来缩短开发周期。当然，如果开发人员需要查看程序代码但并未拥有被完整调出的源文件树，能够在本地文件服务器上轻易取得项目的源文件将会很方便。

有这么多不同的用法，使得确认"引用源文件和二进制文件树"是否完整变得较为重要。要提高可靠度，一个简单而有效的方法就是将源文件树设定成只读（read-only）。这样可确保"引用源文件树"能够准确反映当时被调出仓库的状态。你可以在需要特别注意的地方这么做，因为许多编译动作有可能会想要写入源文件树，尤其是在产生源代码或写出临时文件的时候。将源文件树设定成只读还可以避免漫不经心的用户意外地破坏源文件树，这是一个最常出现的错误。

项目编译系统的另一个常见的需求，就是具备轻易处理不同编译、链接和部署配置的能力。编译系统通常必须能够管理项目的各个版本（可能是源文件仓库中的各个分支）。

多数大型项目相当倚重第三方软件（不是可链接的程序库就是工具的形式）。如果没有其他工具可用来管理软件的配置（通常就是没有），使用*makefile*和编译系统来做此管理通常是一个合理的选择。

最后，当软件要发布给客户时，它通常会从开发的形式重新封装成发行的形式。这可以很复杂，像为Windows制作*setup.exe*文件；也可以很简单，像编排HTML文件以及使用jar来封装它。有时安装程序（installer）的编译动作会并入到正常的编译程序中。我比较喜欢将编译和安装分成两个阶段，因为它们使用的似乎是两种完全不同的过程。无论如何，这两项操作都会对编译系统造成影响。

文件系统的布局

一旦你选择支持多个二进制文件树（binary tree），就会引发文件系统布局的问题。在需要多个二进制文件树的环境之中，通常会存在着许多二进制文件树。要维持这些树的正确性，需要动些脑筋。

一个组织此类数据的常见方法，就是指定一个大型的磁盘以作为二进制文件树的"布局空间"（farm）。在这个磁盘的顶层（或附近）为每个二进制文件树配置一个目录。一个可行的布局方案就是每个目录的名称应该包含厂商、硬件平台、操作系统以及二进制文件树的编译参数：

```
$ ls
hp-386-windows-optimized
hp-386-windows-debug
sgi-irix-optimzed
sgi-irix-debug
sun-solaris8-profiled
sun-solaris8-debug
```

当有需要保存不同时间的编译结果时，通常，最好的做法就是为目录名称加上日期戳（甚至是时间戳）。其格式可以是yymmdd或yymmddhhmm：

```
$ ls
hp-386-windows-optimized-040123
hp-386-windows-debug-040123
sgi-irix-optimzed-040127
sgi-irix-debug-040127
sun-solaris8-profiled-040127
sun-solaris8-debug-040127
```

当然，此处的文件名组件次序可能跟你的不一样。这些二进制文件树的顶层目录是存放*makefile*和测试记录的好地方。

这样的布局适合用来存放各个平行开发人员的编译结果。如果现在有一个开发团队负责编译发行版本，也许是给内部的客户使用的，你可以考虑新增一个额外的“发行版布局空间”（release farm）来针对一组产品进行布局，每个产品可以包含版本编号和时间戳，如图6-2所示。

图6-2：发行树的布局范例

此处的产品可能是某个开发团队所编译的、可供其他开发人员使用的程序库。当然，它们也可能是传统上所谓的产品。

无论你的文件布局或环境为何，许多实现的准则都是一样的：每个文件树必须容易区分、清理的动作应该快速且明确，应该有助于文件树的搬移和保存。此外，文件系统的布局应该尽量配合组织的处理结构。程序员以外的人员（像管理人员、质量保证人员和技术文件编写人员）应该能够轻易了解“文件树布局空间”（tree farm）。

自动编译与测试

尽量能够让编译的过程自动化通常很重要。这让“引用树”（reference tree）的编译能够在夜间进行，从而节省了开发人员白天的时间；这也让开发人员能在自己所拥有的无人照料的机器上进行编译的工作。

进入生产阶段的软件，通常会有为不同的产品编译不同版本的需求。对于负责满足这些需求的人来说，启动若干编译过程之后“离开”，通常是让他们保持清醒以及满足需求的不可或缺的能力。

自动测试是另一项议题。许多非图形的应用程序可以使用简单的脚本来管理测试的过程。GNU 工具 dejaGnu 还可以用来测试需要交互的非图形实用程序。当然，JUnit（*http://www.junit.org*）之类的测试结构也能够为非图形单元的测试提供支持。

至于图形应用程序的测试则另当别论。在以X11为基础的系统上，我已经可以使用Xvfb（virtual frame buffer）以 cron 进行自动的测试。在 Windows 系统上，我还找不到一个满意的解决方案。自动测试进行的时候，必须维持测试账号登录的状态以及取消屏保的状态。

第七章

具可移植性的 makefile

何谓具可移植性的*makefile*？举一个极端的例子，就是我们想让一个*makefile*不必经过修改就能够在 GNU make 所运行的任何系统上运行。不过这几乎是不可能的，因为不同的操作系统间有极大的差异。一个较合理的说法是，具可移植性的*makefile*经过简单的修改就可以在每个新的平台上运行。一个重要的额外限制是：不可以因为移植到新的系统而中断对之前平台的支持。

如同传统的程序设计，我们也可以使用封装（encapsulation）和抽象化（abstraction）等技术让*makefile*达到这样的可移植性水平。通过变量和用户自定义函数的使用，我们可以封装应用程序和算法；通过为命令行选项和参数定义变量，我们可以从常数元素中抽取出具平台差异性的元素。

然后你必须决定每个平台应该提供哪些工具来让你的工作能够顺利完成，以及每个平台应该使用什么方法。只使用你感兴趣的、所有平台上都提供的工具和功能，可让你获得最大程度的可移植性。这就是所谓的“最小公分母”（least common denominator）法，显然你所使用的都应该是非常基本的功能。

“最小公分母”法的另一个版本，就是使用一组功能强大的工具，并且确定你可以把它们带到每个平台，这样可确保你在*makefile*中所调用的命令在任何平台上都可以正确执行。就管理来说，这可能难以实现，而且用户的系统可能无法配合。不过这也可能会成功，稍后我将会举一个在 Windows 上使用 Cygwin 包的例子。正如你将会看到的，工具标准化无法解决每个问题，操作系统之间总是存在某些差别。

最后，你会接受系统之间存在着差异性以及通过慎选宏和函数来跨越这些差异。本章稍后也会示范这个方法。

所以，通过明智而审慎地使用变量和用户自定义函数，以及尽量使用标准工具以避免特殊的功能，我们就可以提高*makefile*的可移植性。正如稍早所说，可移植性不可能达到完美的地步，所以我们的工作就是让编译结果与可移植性达到平衡。不过在我们探索具体的技术之前，让我们了解具可移植性的*makefile*的若干内容。

可移植性的若干内容

可移植性的问题可能难以具体描述，因为它涵盖了极广的范围，大到完全转移自己的思考模式（比如传统的Mac OS与System V Unix），小到缺陷的修复（比如“修复”程序的错误结束状态中的某个缺陷）。然而，下面还是列出了每个*makefile*迟早必须面对的若干常见的可移植性问题：

程序名称

不同平台上相当常见的问题，就是对相同或类似的程序使用不同的名称。最常见的就是C或C++编译器的名称（例如cc、xlc）。另一个常见的状况是，当GNU版的程序被安装到非GNU的系统上时会对程序的名称前置字母g（例如gmake、gawk）。

路径

平台不同，程序和文件的位置通常也会不同。例如，在Solaris系统上，X窗口的路径为*/usr/X*，然而在许多其他的系统上，X窗口的路径为*/usr/X11R6*。此外，当你从一个系统转移到另一个系统时，*/bin*、*/usr/bin*、*/sbin*和*/usr/sbin*之间的差别通常会有些模糊不清。

选项

程序不同，命令行选项也会不同，特别是当你使用的是替代品时。此外，如果一个平台缺少了某个实用程序或是提供了某个有问题的版本，你可能需要将该实用程序替换成另一个使用不同命令行选项的实用程序。

shell的功能

make默认会以*/bin/sh*来运行脚本，但是不同的sh实现之间的功能差异很大。特别是非POSIX的shell缺少了许多功能，无法接受新式shell所能处理的语法。

Open Group针对System V shell和POSIX shell的差异提供了一份非常有用的白皮书。你可以在*http://www.unix-systems.org/whitepapers/shdiffs.html*找到这份文件。想要取得进一步信息，可以在*http://www.opengroup.org/onlinepubs/007904975/utilities/xcu_chap02.html*找到POSIX的shell's command language规范说明书。

程序的行为

具可移植性的*makefile*还必须全力处理行为完全不同的程序。当有不同的厂商调整或插入缺陷以及新增功能时，这是极为常见的问题。实用程序是否升级到厂商的最新版本也是一个问题。例如，awk 程序于 1987 年大改版，几乎 20 年之后，仍有若干系统并未把它们的 awk 程序升级到这个版本。

操作系统

最后，完全不同的操作系统之间，比如 Windows 与 Unix 或 Linux 与 VMS，也存在着可移植性的问题。

Cygwin

虽然make已经有固有的Win32移植版本可用，但是仍存在着一小部分的Windows移植问题，因为这个固有的移植版本所使用的shell是*cmd.exe*（或*command.exe*）。再加上缺少其他 Unix 工具，让“跨平台移植”成为令人却步的工作。幸好，Cygwin 项目（*http://www.cygwin.com*）为 Windows 建立了一个与 Linux 兼容的程序库，这让许多程序（注 1）能够被移植过来。对想要获得 Linux 兼容性或取用 GNU 工具的 Windows 开发人员来说，我不相信他们还可以找到更好的工具。

我将 Cygwin 使用在各种项目上，从 C++/Lisp 的 CAD 应用程序到纯 Java 的工作流程管理系统，已经有 10 年以上的经验。Cygwin 工具集包含了许多程序语言的编译器和解释器。然而，就算应用程序本身被实现的时候使用的是非 Cygwin 的编译器和解释器，Cygwin的使用也能让你获益。Cygwin工具集只是用来让开发过程和编译过程协同一致。换句话说，你无需编写 Cygwin 应用程序或者使用 Cygwin 语言工具，就可以从 Cygwin 环境获得好处。

然而，Linux 并非 Windows（谢天谢地！），当你将 Cygwin 工具使用在原生的 Windows 应用程序上时，将会遇到一些问题。这些问题几乎都在“文件中所使用的行结束符号”以及“Cygwin 与 Windows 之间所传递的路径的形式”上打转。

行终止符

Windows操作系统的文本文件中将会以CRLF，即carriage return（回行首）与line feed（换行），这个双字符序列作为行终止符（line terminator）。POSIX 系统则会使用单字符 LF作为行终止符。这个差异有时会导致令人混淆的状况，例如程序汇报语法错误或是在

注 1： 我的 Cygwin 操作环境的 */bin* 目录中目前包含了 1343 个可执行文件。

数据文件中找错位置。当Cygwin被安装（或是mount命令被使用）的时候，你可以选择是否应该让Cygwin以CRLF结束符来转译文件。如果你选择使用DOS文件格式，那么当文本文件被读入的时候，Cygwin会把CRLF转译成LF，而当文本文件被写出的时候，Cygwin会进行相反的动作。因此，以Unix为基础的程序可以正确处理DOS文本文件。如果你打算使用原生的语言工具，比如Visual C++或Sun的Java SDK，请选择DOS文件格式；如果你准备使用Cygwin编译器，请选择Unix（你可以随时改变你的选择）。

此外，你也可以使用Cygwin所提供的工具手动进行文件的转译工作。如果有需要，你可以使用dos2unix和unix2dos来转换文件的行结束符号。

文件系统

Cygwin所提供的是Windows文件系统的POSIX观点。POSIX文件系统的根目录是/，刚好对应到Cygwin被安装的目录。至于Windows磁盘驱动器则可通过假目录*/cygdrive/letter*来访问。所以，如果Cygwin被安装在*C:\usr\cygwin*（我喜欢安装在这个位置）下，你可以在表7-1中看到相应的目录映射关系。

表7-1：默认的Cygwin目录映射关系

原生的Windows路径	Cygwin路径	替代的Cygwin路径
c:\usr\cygwin	*/*	*/cygdrive/c/usr/cygwin*
c:\Program Files	*/cygdrive/c/Program Files*	
c:\usr\cygwin\bin	*/bin*	*/cygdrive/c/usr/cygwin/bin*

乍看之下，这可能会令人有些混淆，但是这并不会对工具造成任何问题。Cygwin还提供了mount命令，这让访问文件和目录的用户能够更加方便。这个mount命令具有一个选项，--change-cygdrive-prefix，可让你变更前缀符号。我发现将前缀符号变更为/特别有用，因为磁盘驱动器盘符（drive letter）的使用将会变得更加自然：

```
$ mount --change-cygdrive-prefix /
$ ls /c
AUTOEXEC.BAT            Home          Program Files              hp
BOOT.INI                I386          RECYCLER                   ntldr
CD                      IO.SYS        System Volume Information  pagefile.sys
CONFIG.SYS              MSDOS.SYS     Temp                       tmp
C_DILLA                 NTDETECT.COM  WINDOWS                    usr
Documents and Settings  PERSIST       WUTemp                     work
```

做此变更之后，前面的目录映射关系将会如表7-2所示。

表 7-2：修改后的 Cygwin 目录映射关系

原生的 Windows 路径	Cygwin 路径	替代的 Cygwin 路径
c:\usr\cygwin	*/*	*/c/usr/cygwin*
c:\Program Files	*/c/Program Files*	
c:\usr\cygwin\bin	*/bin*	*/c/usr/cygwin/bin*

如果你需要将文件名传递给一个 Windows 程序，比如 Visual C++ 编译器，你通常可以使用 POSIX 风格的分隔符（斜线）传递文件的相对路径。斜线和反斜线对 Win32 API 而言并无不同，可惜，有些实用程序在解析它们自己的命令行参数时会将斜线作为命令选项。DOS 的 print 命令就是这样的实用程序，net 命令是另一个这样的实用程序。

如果使用绝对路径，磁盘驱动器盘符的语法总是一个问题。尽管 Windows 程序通常可以接受斜线，但是它们完全无法了解 */c* 的语法。磁盘驱动器盘符总是必须转换回 *c:*。为了解决此问题以及进行斜线/反斜线的转换，Cygwin 提供了 cygpath 实用程序，让我们能够来回转换 POSIX 路径与 Windows 路径。

```
$ cygpath --windows /c/work/src/lib/foo.c
c:\work\src\lib\foo.c
$ cygpath --mixed /c/work/src/lib/foo.c
c:/work/src/lib/foo.c
$ cygpath --mixed --path "/c/work/src:/c/work/include"
c:/work/src;c:/work/include
```

--windows 选项可让你将命令上所指定的 POSIX 路径转译成 Windows 路径（或者你可以使用适当的选项进行相反的动作）。如果 Windows 可以接受的话，我比较喜欢使用 --mixed 选项来产生 Windows 路径，不过是以斜线而不是以反斜线为分隔符。这么做对 Cygwin shell 的运作会比较好，因为反斜线在该处是个转义字符（escape character）。cygpath 实用程序具有许多选项，其中的部分可为重要的 Windows 路径提供相应的 POSIX 路径：

```
$ cygpath --desktop
/c/Documents and Settings/Owner/Desktop
$ cygpath --homeroot
/c/Documents and Settings
$ cygpath --smprograms
/c/Documents and Settings/Owner/Start Menu/Programs
$ cygpath --sysdir
/c/WINDOWS/SYSTEM32
$ cygpath --windir
/c/WINDOWS
```

如果你正在 Windows/Unix 混合的环境中使用 cygpath，你将会在具可移植性的函数中纳入这些调用：

```
ifdef COMSPEC
  cygpath-mixed          = $(shell cygpath -m "$1")
  cygpath-unix           = $(shell cygpath -u "$1")
  drive-letter-to-slash = /$(subst :,,$1)
else
  cygpath-mixed          = $1
  cygpath-unix           = $1
  drive-letter-to-slash = $1
endif
```

如果你只需要把*c:*这个磁盘驱动器盘符语法映射至POSIX的形式，drive-letter-to-slash函数的速度将会比运行cygpath程序还快。

最后，Cygwin无法了解Windows中所有的"怪癖"。在Windows中无效的文件名，在Cygwin中也属无效。因此，你无法在POSIX路径中使用*aux.h*、*com1*和*prn*之类的文件名，即使有扩展名也不行。

程序名称相抵触时

有些Windows程序的名称跟Unix程序的一样。当然，此类Windows程序的命令行参数和行为跟同名的Unix程序是不一样的。如果你意外调用了Windows版的程序，通常会造成非常混乱的结果。最麻烦的似乎是find、sort、ftp和telnet等程序，当你在Unix、Windows和Cygwin之间移植此类程序时，若想获得最大的兼容性，则应该对它们使用完整的路径。

如果你对Cygwin有绝对的信心，而且不需要使用原生的Windows支持工具，那么在你的PATH变量中，就应该把Cygwin的*/bin*目录放在Windows路径的前面。这样可确保Cygwin工具会被优先采用。

如果你的*makefile*用到Java工具，别忘了Cygwin所提供的GNU jar程序无法处理标准的Sun *jar*文件格式。因此，在你的PATH变量中，就应该把Java jdk的*/bin*目录放在Cygwin的*/bin*目录的前面，以避免用到Cygwin的jar程序。

管理程序和文件

管理程序最常见的方式，就是对有可能发生变化的程序名称或路径使用变量。此类变量可以定义在简单块中，正如我们之前看到的：

```
MV ?= mv -f
RM ?= rm -f
```

或是定义在条件块中：

```
ifdef COMSPEC
  MV ?= move
  RM ?= del
else
  MV ?= mv -f
  RM ?= rm -f
endif
```

当变量定义在简单块时，如果要变更它们的值，可以在命令行上重新设定它们、编辑 *makefile* 或是设定环境变量（因为我们使用的是条件赋值运算符?=）。正如之前所提到的，测试 Windows 平台的一个方法，就是检查所有 Windows 操作系统都会使用的 COMSPEC 变量。有些时候只有一个路径需要变更：

```
ifdef COMSPEC
  OUTPUT_ROOT := d:
  GCC_HOME    := c:/gnu/usr/bin
else
  OUTPUT_ROOT := $(HOME)
  GCC_HOME    := /usr/bin
endif

OUTPUT_DIR := $(OUTPUT_ROOT)/work/binaries
CC := $(GCC_HOME)/gcc
```

这样你就可以在 *makefile* 中通过 make 变量调用大部分的程序。不过在你习惯之前，你可能会觉得 *makefile* 有点难读。然而，不管怎样，在 *makefile* 中使用变量通常会比较方便，因为相比较于逐字指定的程序名称，变量名称通常会短很多，特别是当你必须使用完整路径的时候。

相同的技术可用来管理不同的命令选项。例如，内置的编译规则便包含了一个名为 TARGET_ARCH 的变量，这个变量可用来指定平台专属的标记：

```
ifeq "$(MACHINE)" "hpux-hppa"
  TARGET_ARCH := -mdisable-fpregs
Endif
```

当你要定义自己的程序变量的时候，可能需要用到类似的方法：

```
MV := mv $(MV_FLAGS)

ifeq "$(MACHINE)" "solaris-sparc"
  MV_FLAGS := -f
Endif
```

如果要移植到许多平台上，这么做就会出现一连串的 ifdef 块，这会使得 *makefile* 变得既难阅读又不好维护。这个时候，你可以把平台专属变量存放在它自己的文件中，并在文件名中包含平台指示符。举例来说，如果你可以通过平台的 uname 参数来指定一个平台，那么你就可以使用下面的方式选出适当的 make 引入文件：

```
MACHINE := $(shell uname -smo | sed 's/ /-/g')
include $(MACHINE)-defines.mk
```

文件名中若包含空格，将会对make造成问题。“解析进行期间以空格为分隔符”这个假设对make而言十分重要。许多内置函数，比如word、filter、wildcard等，都会假设它们的参数是以空格为分隔符。不过，有些诀窍或许可以在某些小地方协助你。第一个诀窍（参考“支持多个二进制文件树”一节）就是使用subst来以另一个字符取代空格：

```
space = $(empty) $(empty)

# $(call space-to-question,file-name)
space-to-question = $(subst $(space),?,$1)
```

space-to-question函数将会以文件名匹配通配符?来取代所有空格。现在，我们可以实现能够处理空格的wildcard和file-exists函数：

```
# $(call wildcard-spaces,file-name)
wildcard-spaces = $(wildcard $(call space-to-question,$1))

# $(call file-exists
file-exists = $(strip                                          \
                $(if $1,,$(warning $1 has no value))           \
                $(call wildcard-spaces,$1))
```

我们可以使用space-to-question实现wildcard-spaces函数，让*makefile*能够在包含空格的模式上进行通配符的处理。我们还可以使用wildcard-spaces实现file-exists函数。当然，问号的使用可能会让wildcard-spaces返回不匹配的文件（例如，“my document.doc”和“my-document.doc”），但是我们最多只能做到这个程度。

space-to-question函数还可用来转换工作目标和必要条件中包含空格的文件名，因为你可以在工作目标和必要条件中使用文件名匹配模式（globbing pattern）。

```
space := $(empty) $(empty)

# $(call space-to-question,file-name)
space-to-question = $(subst $(space),?,$1)

# $(call question-to-space,file-name)
question-to-space = $(subst ?,$(space),$1)

$(call space-to-question,foo bar): $(call space-to-question,bar baz)
        touch "$(call question-to-space,$@)"
```

假设文件“*bar baz*”存在，那么当这个*makefile*首次被运行的时候就会找到必要条件，因为文件名匹配模式已经被求过值了。但是工作目标的文件名匹配模式会求值失败，因为该工作目标尚不存在，所以$@的值将会是foo?bar。然后，命令脚本会使用question-

to-space将$@转换回我们真正想要的，包含空格的文件名。当这个*makefile*第二次被运行时会找到这个工作目标，因为文件名匹配模式会找到具有空格的工作目标。尽管有点儿不容易看得懂，不过我发现这些诀窍在实际的*makefile*中很有用。

源文件树的布局

可移植性的另一个方面就是"允许开发人员在他们认为有需要的时候自由管理自己的开发环境"的能力。如果编译系统要求开发人员必须把他们的源文件、二进制文件、程序库以及支持工具存放在同一个目录下，或是存放在同一个Windows磁盘上，最后，在磁盘空间快用光时，开发人员将会面临必须分开这些文件的状况。

此时，有意义的*makefile*实现方式，就是使用变量来引用这些被分开的文件以及设定合理的默认值。此外，因为能够通过变量引用每个支持程序库和工具，所以开发人员可以在发现有需要的时候，自定义文件的位置。你可以对最有可能需要自定义的变量使用条件赋值运算符，让开发人员能够轻易地使用环境变量来改写*makefile*。

另外，"能够轻易地为源文件和二进制文件树支持多个副本"也是一项对开发人员有益的能力。即使不必支持不同的平台或编译选项，开发人员通常也会发现自己还是会用到源文件的多个副本，可能是基于调试的目的或是因为他们要同时开发多个项目。我们已经讨论过支持此功能的两种方法：使用一个"顶层"环境变量来区分源文件和二进制文件树的根目录，或是使用*makefile*的目录以及一个固定的相对路径来找到二进制文件树。这两种方法都可以让开发人员获得支持多个文件树的灵活性。

使用不具可移植性的工具

正如之前所提到的，"最小公分母"法的替代方案就是采用若干标准工具。当然，你必须确定这些标准工具的可移植性至少跟你所要编译的应用程序一样。显然，来自GNU项目的程序是具可移植性工具的最佳选择。不过具可移植性工具的来源很广泛，Perl和Python是我马上可以想到的两个具可移植性的工具。

在你找不到具可移植性工具的时候，使用make函数来封装不具可移植性的工具，有时会很有用。例如，支持Enterprise JavaBeans的各式各样的编译器（每个编译器的调用语法皆有细微的差异），此时我们可以编写一个基本的函数来编译一个EJB jar并将它参数化以便插入不同的编译器。

```
EJB_TMP_JAR = $(TMPDIR)/temp.jar

# $(call compile-generic-bean, bean-type, jar-name,
#                               bean-files-wildcard, manifest-name-opt )
```

```
define compile-generic-bean
  $(RM) $(dir $(META_INF))
  $(MKDIR) $(META_INF)
  $(if $(filter %.xml %.xmi, $3),                    \
    cp $(filter %.xml %.xmi, $3) $(META_INF))
  $(call compile-$1-bean-hook,$2)
  cd $(OUTPUT_DIR) &&                                \
  $(JAR) -cf0 $(EJB_TMP_JAR)                         \
         $(call jar-file-arg,$(META_INF))            \
         $(call bean-classes,$3)
  $(call $1-compile-command,$2)
  $(call create-manifest,$(if $4,$4,$2),,)
endef
```

这个通用的EJB编译函数的第一个参数用来指定我们所要使用的bean编译器类型，比如Weblogic、Websphere等；其余的参数分别用来指定jar的名称、jar中所包含的文件（包括配置文件）以及一个非必需的manifest文件。这个模板函数首先会以“删除任何旧的临时目录并且予以重建”的方式来建立一个干净的暂存区。接着，此函数会将出现在必要条件中的*xml*或*xmi*文件复制到$(META_INF)目录下。这个时候，我们可能需要执行自定义的操作（custom operation）以便清理*META-INF*中的文件或是准备*.class*文件。为了支持这些操作，我们预备了一个挂钩函数compile-$1-bean-hook，用户可以在有需要的时候自行定义。举例来说，如果Websphere编译器需要一个额外的控制文件，比如*xsl*文件，我们就可以编写下面这个挂钩：

```
# $(call compile-websphere-bean-hook, file-list)
define compile-websphere-bean-hook
  cp $(filter %.xsl, $1) $(META_INF)
endef
```

只要定义过这个函数，compile-generic-bean中的这个调用将会被扩展得恰如其分。如果我们选择不编写这个挂钩函数，compile-generic-bean中的这个调用将会被扩展成空无一物。

接着，我们的通用函数会创建jar。辅助函数jar-file-arg会把一个普通的文件路径分解成一个 -C 选项与一个相对路径：

```
# $(call jar-file-arg, file-name)
define jar-file-arg
  -C "$(patsubst %/,%,$(dir $1))" $(notdir $1)
endef
```

辅助函数bean-classes可以从一个源文件列表中取出恰当的类文件（jar文件只需要interface和home类）：

```
# $(call bean-classes, bean files-list)
define bean-classes
```

```
    $(subst $(SOURCE_DIR)/,,                    \
      $(filter %Interface.class %Home.class, \
        $(subst .java,.class,$1)))
endef
```

然后，通用函数会以$(call $1-compile-command,$2)来调用我们所选择的编译器：

```
define weblogic-compile-command
  cd $(TMPDIR) && \
  $(JVM) weblogic.ejbc -compiler $(EJB_JAVAC) $(EJB_TMP_JAR) $1
endef
```

最后，我们的通用函数会加入 manifest。

定义好 compile-generic-bean 之后，我们可以将它封装在想要支持的每个环境的 compiler-specific-bean 函数里。

```
# $(call compile-weblogic-bean, jar-name,
#                               bean files-wildcard, manifest-name-opt )
define compile-weblogic-bean
  $(call compile-generic-bean,weblogic,$1,$2,$3)
endef
```

标准的 shell

有一件事值得一提再提，一个令人厌烦的不兼容问题，那就是当你从一个系统移往另一个系统时发现 */bin/sh*（make 默认使用的 shell）的能力有问题。如果你发现自己需要修改 *makefile* 里的命令脚本，就应该考虑让你的 shell 标准化。当然，让 *makefile* 运行在不受控制的环境里，这对典型的开放源码计划来说，并不恰当。然而，让 *makefile* 运行在受控制的设定之中，以及固定使用一组经过特别设定的机器，那就相当恰当了。

此外，为了避免 shell 的不兼容，许多的 shell 都会提供“可以避免使用其他小型实用程序”的功能。例如，bash shell 所提供的经强化的 shell 变量展开功能，比如 %% 和 ##，可以协助我们避免其他 shell 实用程序，比如 sed 和 expr 的使用。

automake

本章的重点放在如何有效地使用 GNU make 与支持工具以便实现一个具可移植性的编译系统上。即使这些都是适当的需求，然而有的时候还是会超出其能力范围。如果你无法使用 GNU make 所提供的经强化的功能，并被迫依赖一组具最小公分母性质的功能，此时你应该考虑使用 automake 工具，参见 *http://www.gnu.org/software/automake/automake.html*。

automake工具的输入是一个经过格式化的*makefile*，其输出是一个具可移植性的旧式*makefile*。automake产生自一组m4宏，这让你能够在输入文件（称为*makefile.am*）中使用非常精简的符号。通常，automake会跟autoconf（对C/C++程序而言这是一个具可移植性的支持包）一起使用，不过你不一定要使用autoconf。

对于需要具备最大可移植性的编译系统而言，尽管automake是一个不错的解决方案，但是它所产生的*makefile*并不能使用GNU make的任何高级功能，附加运算符（+=）除外，因为automake对它提供特别的支持。此外，automake所接受的输入文件有点像一般的*makefile*。因此，使用automake（但不使用autoconf）跟使用最小公分母法没有太大的差别。

第八章

C与C++

本章将会延伸第六章所提到的议题与技术并将之应用在C与C++的项目上。我们将会继续以“在非递归的*makefile*上编译mp3 player”来举例说明。

分开源文件与二进制文件

如果我们想要以单一源文件树支持多个平台，以及为每个平台编译多个版本，将源文件树和二进制文件树分开是必要的，所以我们该这么做呢？make程序最初是设计来处理放在单一目录里的文件的。尽管情况已经有显著的改变，但make并未把它的根本忘掉，当它所要更新的文件被存放在当前目录（或它的子目录）中时，make也能将多个目录管理得很好。

简单的办法

最简单的办法，就是让make将二进制文件存放在与源文件不同的目录里，并从二进制文件目录来启动make程序。此时，输出文件的访问可通过相对路径，而输入文件的位置则必须明确指定其路径或是通过对vpath的搜索。不管是哪种状况，我们都需要在不同的地方引用到源文件目录，所以我们会在开头使用一个变量来保存它：

```
SOURCE_DIR := ../mp3_player
```

让我们来看例6-3的非递归*makefile*在这个办法下有何变化：source-to-object没有改变，不过subdirectory函数现在需要考虑到源文件的相对路径。

```
# $(call source-to-object, source-file-list)
source-to-object = $(subst .c,.o,$(filter %.c,$1)) \
                   $(subst .y,.o,$(filter %.y,$1)) \
                   $(subst .l,.o,$(filter %.l,$1))
```

```
# $(subdirectory)
subdirectory = $(patsubst $(SOURCE_DIR)/%/module.mk,%,   \
                  $(word                                  \
                    $(words $(MAKEFILE_LIST)),$(MAKEFILE_LIST)))
```

在我们现在的*makefile*中，MAKEFILE_LIST里所列出的文件将会包含源文件的相对路径。所以，如果要从中取出各模块目录的相对路径，我们就必须去除前缀以及*module.mk*后缀。

接着，为了协助make找到源文件，我们将会使用vpath的功能：

```
vpath %.y $(SOURCE_DIR)
vpath %.l $(SOURCE_DIR)
vpath %.c $(SOURCE_DIR)
```

这让我们能够为自己的源文件以及输出文件使用简单的相对路径。当make需要一个源文件时，如果它在输出文件树的当前目录中找不到该文件，它将会搜索SOURCE_DIR。然后，我们必须更新include_dirs变量：

```
include_dirs := lib $(SOURCE_DIR)/lib $(SOURCE_DIR)/include
```

除了源文件目录，此变量现在还包含了二进制文件树的*lib*目录，因为自动产生的yacc和lex头文件将会存放在那里。

make的include指令必须被更新为从它们的源文件目录来访问*module.mk*文件，因为make并不会使用vpath来查找引入文件：

```
include $(patsubst %,$(SOURCE_DIR)/%/module.mk,$(modules))
```

最后，我们会创建输出目录本身：

```
create-output-directories :=                          \
        $(shell for f in $(modules);                  \
                do                                    \
                  $(TEST) -d $$f || $(MKDIR) $$f;     \
                done)
```

这个赋值动作将会创建一个虚拟变量，它的值远永不会被用到。但因为这是个简单的赋值动作，所以我们可以确定，在make执行任何其他工作之前这些目录已经被创建好了。我们必须“手动”创建这些目录，因为yacc、lex以及其他依存文件的产生并不会创建输出目录本身。

确定这些目录是否被创建的另一个方法，就是将这些目录设定成依存文件（*.d*文件）的必要条件。这并不是一个好主意，因为目录其实无法作为必要条件。yacc、lex或其他依存文件与目录的内容并无依存关系，也不应该只是因为目录的时间戳已经更新而去重

新产生它们。事实上，当某个输出目录中的一个文件被加入或移除时，如果项目会被重新编译，这将会是一个非常没有效率的方法。

module.mk 文件的修改甚至更为简单：

```
local_src := $(addpretix $(subdirectory)/,playlist.y scanner.l)

$(eval $(call make-library, $(subdirectory)/libdb.a, $(local_src)))

.SECONDARY: $(call generated-source, $(local_src))

$(subdirectory)/scanner.d: $(subdirectory)/playlist.d
```

原来的 *makefile* 提供了以通配符寻找源文件的功能。要恢复此功能相当简单，留给读者自己进行练习。这个版本有一个毛病，看来似乎是原本的 *makefile* 的一个缺陷。当此范例被运行时，我发现依存文件 *scanner.d* 会在它所依存的 *playlist.h* 被产生之前先被产生。这项依存关系在原本的 *makefile* 中就漏列了，只是碰巧没有出问题而已。要让所有的依存关系都正确是一项困难的任务，即使是在小型的项目中。

假设源文件被存放在 *mp3_player* 子目录中，下面是我们以这个新版的 *makefile* 编译项目的方法：

```
$ mkdir mp3_player_out
$ cd mp3_player_out
$ make --file=../mp3_player/makefile
```

尽管这是一个正确无误可以正常运行的 *makefile*，不过却相当麻烦。首先你必须将目录切换至输出目录，然后必须加上 `--file`（或 `-f`）选项。我们可以使用如下的 shell 脚本来解决这个问题：

```
#! /bin/bash
if [[ ! -d $OUTPUT_DIR ]]
then
  if ! mkdir -p $OUTPUT_DIR
  then
    echo "Cannot create output directory" > /dev/stderr
    exit 1
  fi
fi

cd $OUTPUT_DIR
make --file=$SOURCE_DIR/makefile "$@"
```

这个脚本会假设源文件和输出文件目录被分别存放在环境变量 `SOURCE_DIR` 和 `OUTPUT_DIR` 中。这是一个标准实现方式，可以让开发人员在不用频繁地键入路径的状况下轻易切换文件树。

最后，有一点必须注意：make 或 *makefile* 并无法避免开发人员从源文件树来运行 *makefile*，即使它应该从二进制文件树来运行。这是一个常见的问题，而且某些命令脚本的行为可能会让问题雪上加霜。举例来说，如下的 clean 工作目标：

```
.PHONY: clean
clean:
        $(RM) -r *
```

会把用户的源文件树全部删除掉！似乎应该在最上层的*makefile*中对此进行检查以防万一：

```
$(if $(filter $(notdir $(SOURCE_DIR)),$(notdir $(CURDIR))),\
  $(error Please run the makefile from the binary tree.))
```

这段代码会测试当前的工作目录（$(notdir $(CURDIR)））与源文件目录（$(notdir $(SOURCE_DIR)））是否相同。如果一样，则会输出错误信息并且结束运行。因为 if 和 error 等函数会被扩展成空值，所以我们可以将这两行直接放在SOURCE_DIR定义之后。

麻烦的办法

有些开发人员觉得每次都必须切换到二进制文件树实在太麻烦了，所以他们不遗余力地想避免这么做，或者，*makefile*的维护人员的工作环境可能不适合使用shell脚本的封装（wrapper）或别名。不管是哪一种状况，*makefile* 都可以被修改成从源文件树中运行 make，以及通过为所有的输出文件名前置路径，来把二进制文件存放在独立的输出文件树中。这个时候，我通常会使用绝对路径，因为这样可提供较大的灵活性，然而这将会让命令行的长度问题恶化。输入文件仍继续使用相对于 *makefile* 所在目录的路径。

例 8-1 所示的 *makefile* 已经被修改成让 make 能够从源文件树运行，以及将二进制文件写出至二进制文件树中。

例 8-1：*把源文件和二进制文件分开的 makefile 允许从源文件树来运行*

```
SOURCE_DIR := /test/book/examples/ch07-separate-binaries-1
BINARY_DIR := /test/book/out/mp3_player_out

# $(call source-dir-to-binary-dir, directory-list)
source-dir-to-binary-dir = $(addprefix $(BINARY_DIR)/, $1)

# $(call source-to-object, source-file-list)
source-to-object = $(call source-dir-to-binary-dir,     \
                    $(subst .c,.o,$(filter %.c,$1))     \
                    $(subst .y,.o,$(filter %.y,$1))     \
                    $(subst .l,.o,$(filter %.l,$1)))

# $(subdirectory)
subdirectory = $(patsubst %/module.mk,%,                \
```

```
                $(word                                  \
                  $(words $(MAKEFILE_LIST)),$(MAKEFILE_LIST)))

# $(call make-library, library-name, source-file-list)
define make-library
  libraries += $(BINARY_DIR)/$1
  sources   += $2

  $(BINARY_DIR)/$1: $(call source-dir-to-binary-dir,    \
                      $(subst .c,.o,$(filter %.c,$2))   \
                      $(subst .y,.o,$(filter %.y,$2))   \
                      $(subst .l,.o,$(filter %.l,$2)))
        $(AR) $(ARFLAGS) $$@ $$^
endef

# $(call generated-source, source-file-list)
generated-source = $(call source-dir-to-binary-dir,    \
                     $(subst .y,.c,$(filter %.y,$1))    \
                     $(subst .y,.h,$(filter %.y,$1))    \
                     $(subst .l,.c,$(filter %.l,$1)))   \
                   $(filter %.c,$1)

# $(compile-rules)
define compile-rules
  $(foreach f, $(local_src),\
    $(call one-compile-rule,$(call source-to-object,$f),$f))
endef

# $(call one-compile-rule, binary-file, source-files)
define one-compile-rule
  $1: $(call generated-source,$2)
        $(COMPILE.c) -o $$@ $$<

  $(subst .o,.d,$1): $(call generated-source,$2)
        $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -M $$< | \
        $(SED) 's,\($$(notdir $$*)\.o\) *:,$$(dir $$@)\1 $$@: ,' > $$@.tmp
        $(MV) $$@.tmp $$@

endef

modules      := lib/codec lib/db lib/ui app/player
programs     :=
libraries    :=
sources      :=

objects      = $(call source-to-object,$(sources))
dependencies = $(subst .o,.d,$(objects))

include_dirs := $(BINARY_DIR)/lib lib include
CPPFLAGS     += $(addprefix -I ,$(include_dirs))
vpath %.h $(include_dirs)

MKDIR := mkdir -p
MV    := mv -f
RM    := rm -f
```

```
SED   := sed
TEST  := test

create-output-directories :=                                          \
        $(shell for f in $(call source-dir-to-binary-dir,$(modules));  \
                do                                                    \
                  $(TEST) -d $$f || $(MKDIR) $$f;                     \
                done)

all:

include $(addsuffix /module.mk,$(modules))

.PHONY: all
all: $(programs)

.PHONY: libraries
libraries: $(libraries)

.PHONY: clean
clean:
        $(RM) -r $(BINARY_DIR)

ifneq "$(MAKECMDGOALS)" "clean"
  include $(dependencies)
endif
```

在这个版本的*makefile*中，source-to-object已经被修改成“为目录列表前置二进制文件树的路径”。这个前置路径的动作将会进行许多次，所以我们会把它编写成函数：

```
SOURCE_DIR := /test/book/examples/ch07-separate-binaries-1
BINARY_DIR := /test/book/out/mp3_player_out

# $(call source-dir-to-binary-dir, directory-list)
source-dir-to-binary-dir = $(addprefix $(BINARY_DIR)/, $1)

# $(call source-to-object, source-file-list)
source-to-object = $(call source-dir-to-binary-dir,     \
                     $(subst .c,.o,$(filter %.c,$1))    \
                     $(subst .y,.o,$(filter %.y,$1))    \
                     $(subst .l,.o,$(filter %.l,$1)))
```

make-library函数同样也会被改成为输出文件前置BINARY_DIR。subdirectory函数将会恢复到它前一个版本的样子，因为引入文件的路径又变成了相对路径。有一个令人意外的小障碍：make 3.80中有一个缺陷会使得它无法在新版的make-library中调用source-to-object。这个缺陷在3.81版中已经被修复，我们可以手动扩展source-to-object函数以越过此缺陷。

现在我们要来研究较复杂的部分。当输出文件无法直接通过相对于*makefile*的路径来访问时，就不会再执行隐含规则了。例如，当源文件与二进制文件位于相同的目录时，或

者C源文件位于子目录中（比如*lib/codec/codec.c*），基本的编译规则%.o: %.c将可以正常运作。当源文件位于独立的目录时，我们可以通过vpath功能，指示make到何处搜索源文件。但是当目标文件位于独立的目录中时，make根本无法判断目标文件位于何处，“工作目标/必要条件”链（chain）将会因此而中断。

让make知道输出文件位于何处的唯一方法，就是以具体规则来链接源文件和目标文件：

```
$(BINARY_DIR)/lib/codec/codec.o: lib/codec/codec.c
```

而且必须为每个目标文件都这么做。

更糟的是，这个“工作目标/必要条件”对（pair）无法使用隐含规则%.o: %.c。这意味着，我们还必须提供命令脚本，这使得我们必须重复描述内置数据库中已有的内容，也可能必须多次重复描述此脚本。这个问题同样也会出现在我们曾经用过的自动依存关系产生规则中。如果必须为每个目标文件加入两项具体规则，*makefile*的维护工作将很难以手动进行。然而，如果能通过函数来自动产生这些规则，将可让代码的重复性以及维护的困难度降至最低：

```
# $(call one-compile-rule, binary-file, source-files)
define one-compile-rule
  $1: $(call generated-source,$2)
        $(COMPILE.c) $$@ $$<

  $(subst .o,.d,$1): $(call generated-source,$2)
        $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -M $$< | \
        $(SED) 's,\($$(notdir $$*)\):o\) *:,$$(dir $$@)\1 $$@: ,' > $$@.tmp
        $(MV) $$@.tmp $$@
endef
```

此函数的前两行就是用来处理object-to-source依存关系的具体规则。这个规则的必要条件必须使用我们在第六章所写的generated-source函数来进行求值的动作，因为源文件中的部分是yacc和lex文件，当它们出现在命令脚本中（例如，来自$^的扩展结果）时，这将会导致编译失败。自动变量被加上了引号（美元符号），所以它们的扩展动作会延后到脚本运行的时候进行，而不是用户自定义函数被eval求值的时候。generated-source函数已经被修改成返回未经转换的C源文件，以及为yacc和lex产生源文件：

```
# $(call generated-source, source-file-list)
generated-source = $(call source-dir-to-binary-dir,     \
                     $(subst .y,.c,$(filter %.y,$1))    \
                     $(subst .y,.h,$(filter %.y,$1))    \
                     $(subst .l,.c,$(filter %.l,$1)))   \
                   $(filter %.c,$1)
```

经过这样的修改之后，此函数现在会产生如下的输出：

参数	结果
lib/db/playlist.y	/c/mp3_player_out/lib/db/playlist.c /c/mp3_player_out/lib/db/playlist.h
lib/db/scanner.l	/c/mp3_player_out/lib/db/scanner.c
app/player/play_mp3.c	app/player/play_mp3.c

这与产生依存关系的具体规则类似。同样地，注意脚本需要使用额外的引号（双美元符号）。

我们现在必须为模块中的每个源文件扩展这个新函数：

```
# $(compile-rules)
define compile-rules
  $(foreach f, $(local_src),\
    $(call one-compile-rule,$(call source-to-object,$f),$f))
endef
```

此函数将用来处理各个*module.mk*文件所使用的全局变量local_src。一个较通用的办法就是将这个文件列表作为参数传递。不过对此项目来说这似乎是多余的，这些函数轻易就能加入到各个*module.mk*文件中：

```
local_src := $(subdirectory)/codec.c

$(eval $(call make-library,$(subdirectory)/libcodec.a,$(local_src)))

$(eval $(compile-rules))
```

我们必须使用eval，因为compile-rules函数会被扩展成多行的make代码。

最后还有一个复杂的地方。如果标准的C编译模式规则无法找到匹配的二进制文件输出路径，则lex的隐含规则以及我们的yacc模式规则也会匹配失败。我们可以手动更新这些规则。因为它们再也无法应用到其他的lex或yacc文件中，所以我们可以把它们移往*lib/db/module.mk*：

```
local_dir := $(BINARY_DIR)/$(subdirectory)
local_src := $(addprefix $(subdirectory)/,playlist.y scanner.l)

$(eval $(call make-library,$(subdirectory)/libdb.a,$(local_src)))

$(eval $(compile-rules))

.SECONDARY: $(call generated-source, $(local_src))

$(local_dir)/scanner.d: $(local_dir)/playlist.d

$(local_dir)/%.c $(local_dir)/%.h: $(subdirectory)/%.y
        $(YACC.y) --defines $<
        $(MV) y.tab.c $(dir $@)$*.c
        $(MV) y.tab.h $(dir $@)$*.h

$(local_dir)/scanner.c: $(subdirectory)/scanner.l
        @$(RM) $@
        $(LEX.l) $< > $@
```

lex规则现在被实现成一般的具体规则，不过yacc规则却被实现成一个模式规则。为什么？因为yacc规则将会被用来建立两个工作目标：一个C源文件与一个头文件。如果我们将yacc规则实现成一般的具体规则，make将会运行两次脚本：一次用来创建C源文件，一次用来创建头文件。但是make会假设一个模式具有多个工作目标，所以如果我们将yacc规则实现成模式规则，make只需运行一次就能同时更新两个工作目标。

如果可能，我将会使用较简单的方法从二进制文件树来进行编译的工作，而不会使用这一节所展示的*makefile*。如你所见，当我们试着从源文件树来进行编译的工作，事情将会立即复杂化（而且似乎会越来越恶化）。

只读的源文件树

一旦源文件树和二进制文件树分开之后，如果编译结果只会产生存放在源文件树中的二进制文件，我们通常可以自由地将“引用源文件树”设置成只读。

在较简单的“从二进制文件树进行编译”的办法中，其所产生的文件将会被自动写入二进制文件树，因为yacc和lex程序运行自二进制文件树。在“从源文件树进行编译”的办法中，我们必须为源文件和工作目标文件提供明确的路径，因此为二进制文件树中的文件指定路径并非额外的工作，除非我们必须强迫自己记住这件事。

将“引用源文件树”设置成只读的其他障碍通常是自己强加的。前人所遗留下来的编译系统通常包含了在源文件树中创建文件的动作，这是因为原来的作者并未考虑到使用只读的源文件树有哪些优点。这些障碍包含了在源文件树中所产生的文件、记录文件以及临时文件。要将这些文件移往输出文件树有时可能会很麻烦，但如果从单一源文件树来建立多个二进制文件树是必要的，可行的替代方案就是维护多个相同的源文件树并且让它们保持同步的状态。

产生依存关系

我们曾在“自动产生依存关系”一节中介绍过如何产生依存关系，不过当时尚留有若干问题未解决。因此，这一节将会为已经提到的简单解决方案，提供若干替代方案（注1）。特别是，稍早所提到的简单办法以及GNU make在线手册中所提到的办法将会遇到以下问题：

注1： 这一节的内容有许多是GNU automake工具的作者Tom Tromey（*tromey@cygnus.com*）创造的，以及取材自Paul Smith（GNU make的维护者）的网站*http://make.paulandlesley.org*上所提供的数据。

- 没有效率。当make发现某个依存文件不存在或尚未更新时，它就会更新.d文件并且重新启动自己。如果在*makefile*的读取期间make会进行许多工作并分析依存图，则重新读取 *makefile* 完全没有效率可言。
- 第一次建立工作目标，以及每当你加入新的源文件的时候，make都会产生警告信息。以上的状况都会让依存文件关联到尚不存在的新源文件上，所以当make试图读取依存文件的时候，会在产生依存文件之前先产生警告信息。这并不是无可挽回的错误，不过相当烦人。
- 如果你移除了一个源文件，make将会在随后的编译过程中发生无可挽回的错误并且停止运行。在此状况下会存在一个依存文件，它的必要条件包含了已移除的文件。因为make无法找到已移除的文件，而且不知道如何编译它，所以make将会输出如下的信息：

  ```
  make: *** No rule to make target foo.h, needed by foo.d.  Stop.
  ```

  此外，make也无法重建依存文件，因为已经发生如上的错误了。要解决此问题，只能靠手动移除该依存文件，但因为通常很难找到这些文件，所以用户一般会删除所有的依存文件以及进行清理的编译工作。当有文件被更名时，也会发生此错误。

  请注意，将.c文件移除或更名所导致的问题并不如将头文件移除或更名所导致的问题显著。这是因为.c文件将会自动从依存文件列表中移除，而且不会在编译过程中发生错误。

Tromey 的办法

让我们逐一探讨这些问题，

如何避免重新启动make？

经过深思熟虑之后，我们发现没有必要重新启动make（即重新读取*makefile*）。如果我们说有一个依存文件需要更新，这表示它的必要条件中至少有一个已被变更，也就是说我们必须更新工作目标。我们还发现这次执行make的时候也不需这么做，因为依存信息并不会改变make的行为。但是我们需要更新依存文件，这样在make下次执行的时候将会拥有完整的依存信息。

因为在这次执行make的时候，我们还不需要依存文件，所以我们可以在更新工作目标的同时产生依存文件。我们可以将编译规则改写成同时更新依存文件来完成此事。

```
# $(call make-depend,source-file,object-file,depend-file)
define make-depend
  $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -M $1 | \
  $(SED) 's,\($$(notdir $2)\) *:.$$(dir $2) $3: ,' > $3.tmp
```

```
  $(MV) $3.tmp $3
endef

%.o: %.c
        $(call make-depend,$<,$@,$(subst .o,.d,$@))
        $(COMPILE.c) -o $@ $<
```

我们将会使用make-depend这个（以源文件、目标文件和依存文件的名称为参数的）函数来实现依存关系产生功能。如果稍后我们需要在不同的语境中重用此函数，这将能够提供最大的灵活性。当我们以这个方式来修改编译规则时，必须删除我们所编写的%.d: %.c模式规则，以避免产生两次依存文件。

现在，目标文件和依存文件在逻辑上应该是链接在一起的：如果一个存在，另一个就必须存在。因此，我们实际上并不在乎是否缺少某个依存文件。如果某个依存文件不存在，这代表相应的目标文件也不存在，而且会在下一个编译过程更新它们。所以我们现在可以忽略因为缺少 *.d* 文件所产生的任何警告信息。

在“引入文件与依存关系”一节中，我们提到过 include 指令的替代形式 -include（或 sinclude）将会忽略错误而且不会产生警告信息：

```
ifneq "$(MAKECMDGOALS)" "clean"
   -include $(dependencies)
endif
```

这可以解决第二个问题：依存文件尚不存在的时候会产生烦人的信息。

最后，当发现缺少必要条件的时候，有个小诀窍可避免产生警告信息。这个诀窍就是为缺少的文件建立一个没有必要条件、没有命令的工作目标。例如，假设我们的依存文件产生器建立了如下的依存关系：

```
target.o target.d: header.h
```

现在假设：由于程序代码重构，*header.h* 不再存在。当我们再次运行 *makefile* 的时候，我们将会得到如下的错误：

```
make: *** No rule to make target header.h, needed by target.d.  Stop.
```

但如果我们为 *header.h* 新增了一个没有命令的工作目标，这个错误就不会发生了：

```
target.o target.d: header.h
header.h:
```

这是因为，如果 *header.h* 不存在，它只会被视为尚未更新，任何以它为必要条件的工作目标将会被更新。所以，即使没有 *header.h* 也可以产生依存文件。如果 *header.h* 存在，则 make将会把它视为已经更新并且继续运行。所以我们只需让每个必要条件关联到一

个空规则就行了。你可能还记得，我们曾在“假想工作目标”一节中首次遇到此类规则。下面是加入新工作目标的make-depend版本：

```
# $(call make-depend,source-file,object-file,depend-file)
define make-depend
  $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -M $1 |        \
  $(SED) 's,\($$(notdir $2)\) *:,$$(dir $2) $3: ,' > $3.tmp
  $(SED) -e 's/#.*//'                                       \
         -e 's/^[^:]*: *//'                                 \
         -e 's/ *\\$$$$//'                                  \
         -e '/^$$$$/ d'                                     \
         -e 's/$$$$/ :/' $3.tmp >> $3.tmp
  $(MV) $3.tmp $3
endef
```

我们将会对依存文件执行新的sed命令，以便产生额外的规则。这段sed程序代码将会进行以下五种转换：

1. 删除注释
2. 删除工作目标文件以及随后的空格
3. 删除结尾的空格
4. 删除空行
5. 为每行的结尾加上一个冒号

（GNU sed可以在单一命令行中对文件进行读取和追加的动作，这使得我们不必使用第二个临时文件。此功能或许无法使用在其他系统上。）新的sed命令将会读取如下的输入：

```
# any comments
target.o target.d: prereq1 prereq2 prereq3 \
     prereq4
```

并且将它转换成：

```
prereq1 prereq2 prereq3:
prereq4:
```

所以，make-depend会把这个新的输出附加到原来的依存文件中。这样就不会发生“No rule to make target”的错误了。

makedepend 程序

到目前为止，大多数编译器所提供的-M选项已足够我们使用了，但如果此选项不存在呢？还有比-M更好的选项可用吗？

近来，大部分的C编译器皆已对“从源文件来产生make依存关系”提供若干支持，但是不久之前的情况并非如此。X Window System项目进行初期，他们实现了一个名为makedepend的工具，此工具可用来从一组C或C++源文件推断出依存关系。你可以通过Internet免费取得此工具。makedepend有点难用，因为它会把输出添加到*makefile*中，但这并不是我们想要的结果。makedepend的输出会假设目标文件与源文件位于同一个目录下，这意味着我们的sed表达式必须做如下的改变：

```
# $(call make-depend,source-file,object-file,depend-file)
define make-depend
  $(MAKEDEPEND) -f- $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) $1 | \
  $(SED) 's,^.*/\([^/]*\.o\) *:,$(dir $2)\1 $3: ,' > $3.tmp
  $(SED) -e 's/#.*//'                                          \
         -e 's/^[^:]*: *//'                                    \
         -e 's/ *\\$$$//'                                      \
         -e '/^$$$$/ d'                                        \
         -e 's/$$$$/ :/' $3.tmp >> $3.tmp
  $(MV) $3.tmp $3
endef
```

-f- 选项用来要求makedepend把它的依存信息写到标准输出。

gcc是makedepend或原生编译器的一个替代方案。gcc为依存信息的产生提供了一组令人混乱的选项。对我们目前的需要来说，下面的做法似乎最恰当：

```
ifneq "$(MAKECMDGOALS)" "clean"
  -include $(dependencies)
endif

# $(call make-depend,source-file,object-file,depend-file)
define make-depend
  $(GCC) -MM              \
         -MF $3           \
         -MP              \
         -MT $2           \
         $(CFLAGS)        \
         $(CPPFLAGS)      \
         $(TARGET_ARCH)   \
         $1
endef

%.o: %.c
        $(call make-depend,$<,$@,$(subst .o,.d,$@))
        $(COMPILE.c) $(OUTPUT_OPTION) $<
```

-MM选项会让gcc从必要条件列表中省略“系统”头文件。这么做有用是因为这些文件难得改变，而且当编译系统越来越复杂时，这样可减少因系统头文件所导致的混乱。最初可能是基于效能的原因这么做。对今日的处理器来说，效能的差异几乎无法量度。

-MF选项用来指定依存文件，也就是把目标文件的.o扩展名替换成.d。还有另一个gcc选项-MD或-MMD，会使用类似的替换方式自动产生输出文件名。理论上，我们比较喜欢使用这个选项，但是这种替换方式无法适当地加入目标文件目录的相对路径，只会把.d文件存放在当前目录中。所以，我们不得不使用-MF来完成此事。

-MP选项用来指示gcc为每个必要条件加入假想工作目标。这样，我们就不必在make-depend函数中加入那个令人混乱的，由五个部分组成的sed表达式。看来似乎是因为automake的开发人员发明了假想工作目标的技术，才使得这个选项被加入gcc。

最后，-MT选项所指定的字符串将会作为依存文件中的工作目标。同样地，如果没有使用这个选项，gcc将无法加入目标文件输出目录的相对路径。

通过gcc的使用，我们不再需要像先前那样以四个命令来产生依存关系。就算你使用的是专属编译器，你还是有可能使用gcc进行依存关系的管理。

支持多个二进制文件树

一旦*makefile*被修改成将二进制文件写到不同的文件树，多个文件树的支持将会变得相当简单。对于交互式编译或开发人员手动编译来说，将会由一个开发人员从键盘来开始编译的动作，这可能需要若干或不需要准备工作。开发人员将会创建输出目录，切换到输出目录并且运行*makefile*。

```
$ mkdir -p ~/work/mp3_player_out
$ cd ~/work/mp3_player_out
$ make -f ~/work/mp3_player/makefile
```

如果整个过程不止这些，那么以shell脚本来封装此过程通常是最佳解决方案。这个shell脚本还可以分析当前目录，以及设定可供*makefile*使用的环境变量，像BINARY_DIR。

```
# !/bin/bash

# 假定我们位于源文件目录下。
curr=$PWD
export SOURCE_DIR=$curr
while [[ $SOURCE_DIR ]]
do
  if [[ -e $SOURCE_DIR/[Mm]akefile ]]
  then
    break;
  fi
  SOURCE_DIR=${SOURCE_DIR%/*}
done

# 如果找不到makefile，就输出错误信息。
if [[ ! $SOURCE_DIR ]]
```

```
then
  printf "run-make: Cannot find a makefile" > /dev/stderr
  exit 1
fi

# 将输出目录设成默认值，如果没有设定的话
if [[ ! $BINARY_DIR ]]
then
  BINARY_DIR=${SOURCE_DIR}_out
fi

# 创建输出目录
mkdir --parents $BINARY_DIR

# 运行make
make --directory="$BINARY_DIR" "$@"
```

这个特别的脚本有点花哨。它首先会在当前目录搜索*makefile*，然后会往上层目录查找，直到找出*makefile*或是输出错误信息。然后，它会检查二进制文件树的变量是否被设定，如果没有设定，就会把它设定成源文件目录附加"_out"。最后，它会创建输出目录并且运行make。

如果这个编译过程将会在不同的平台上进行，那么你需要使用某个方法来区分不同的平台。最简单的做法，就是要求开发人员为每种平台设定一个环境变量，并且在*makefile*中加入条件语句来判断此变量。较好的做法，就是根据uname的输出自动设定平台的类型。

```
space := $(empty) $(empty)
export MACHINE := $(subst $(space),-,$(shell uname -smo))
```

如果编译过程是由cron自动调用的，我发现通过辅助shell脚本会比让cron调用make本身还好。辅助shell脚本可以对初始设定、错误恢复以及自动编译的结束工作提供较佳的支持。此脚本也是一个适合用来设定变量和命令行参数的地方。

最后，如果你的项目支持一组固定的文件树和平台，你可以使用目录名称自动区分当前的编译过程。例如

```
ALL_TREES := /builds/hp-386-windows-optimized \
             /builds/hp-386-windows-debug     \
             /builds/sgi-irix-optimzed        \
             /builds/sgi-irix-debug           \
             /builds/sun-solaris8-profiled    \
             /builds/sun-solaris8-debug

BINARY_DIR := $(foreach t,$(ALL_TREES),\
                $(filter $(ALL_TREES)/%,$(CURDIR)))

BUILD_TYPE := $(notdir $(subst -,/,$(BINARY_DIR)))
```

```
MACHINE_TYPE := $(strip                                  \
                  $(subst /,-,                           \
                    $(patsubst %/,%,                     \
                      $(dir                              \
                        $(subst -,/,                     \
                          $(notdir $(BINARY_DIR)))))))
```

ALL_TREES变量用来保存一份有效的二进制文件树列表。foreach循环用来匹配当前目录与每个有效的二进制文件树，只有一个会匹配成功。一旦区分出二进制文件树，我们就可以从编译目录名称中取出编译类型（例如optimized、debug或profiled）。我们可以通过将破折号分隔符转换成斜线分隔符的方式以及使用notdir取出最后一个单词，来获取目录名称中的最后一个部分。同样地，我们可以通过取出最后一个单词以及使用相同的技术移除最后一个破折号来获取机器类型。

部分的源文件树

在相当大的项目中，只是调出与维护源代码就可能成为开发人员沉重的负担。如果一个项目由许多模块组成，而且有某个开发人员修改了部分的源代码，那么调出与编译整个项目可能会成为一个时间黑洞。这个时候若能够在夜间进行一个集中管理的编译过程，将可填补开发人员的源文件树和二进制文件树中的漏洞。

要这么做，需要进行两种搜索。首先，当编译器所需要的头文件缺少时，你必须指示make搜索“引用源文件树”；其次，当*makefile*所需要的程序库缺少时，你必须指示make搜索“引用二进制文件树”。为了协助编译器找到源文件，我们会在为当前目录指定-I选项后加上额外的-I选项；为了协助make找到程序库，我们会为vpath加上额外的目录。

```
SOURCE_DIR      := ../mp3_player
REF_SOURCE_DIR := /reftree/src/mp3_player
REF_BINARY_DIR := /binaries/mp3_player

include_dirs := lib $(SOURCE_DIR)/lib $(SOURCE_DIR)/include
CPPFLAGS     += $(addprefix -I ,$(include_dirs))                    \
                $(addprefix -I $(REF_SOURCE_DIR)/,$(include_dirs))
vpath %.h       $(include_dirs)                                     \
                $(addprefix $(REF_SOURCE_DIR)/,$(include_dirs))

vpath %.a       $(addprefix $(REF_BINARY_DIR)/lib/, codec db ui)
```

这个做法是假设CVS调出动作的“粒度”（大小范围）为一个程序库或程序模块。在此状况下，make可能会跳过短缺的程序库或程序模块目录，如果开发人员选择不调出它们的话。但是当make需要使用这些程序库时，搜索路径会自动填入缺少的文件。

在这个*makefile*中，我们会以modules变量列出可用来找到*module.mk*文件的目录列表。如果其中的某个目录不会被调出，你就必须编辑此列表将该子目录移除。你也可以使用wildcard来设定*modules*变量：

```
modules := $(dir $(wildcard lib/*/module.mk))
```

这个表达式将会找到包含了 *module.mk* 的所有子目录，并且返回目录列表。请注意，经过 dir 函数的处理，每个目录都会包含一个结尾的斜线符号。

当然，make 也可以在单独的文件层上管理部分的源文件树，通过从开发人员当前的文件树中收集二进制文件以及从引用树中收集缺少的文件来建立程序库。然而，就我们的经验来说，这是个相当麻烦的工作，而且开发人员也不喜欢这么做。

引用编译结果、程序库以及安装程序

此刻，实现引用编译结果所需要的每样东西我们已经介绍得非常多了。自定义一个顶层的 *makefile* 来支持此功能一点都不难，我们只需要以?= 运算符对 SOURCE_DIR 和 BINARY_DIR进行赋值的动作就行了。通过cron运行的脚本基本上应该进行以下动作：

1. 将输出重定向以及设定多个记录文件
2. 清理旧版的编译结果以及清理引用源文件树
3. 调出新的源文件
4. 设定源文件和二进制文件目录变量
5. 调用make
6. 在记录文件中查找错误
7. 产生标记文件，并且有可能要更新locate 数据库（注 2）
8. 发送关于编译成功或失败的信息

在“引用编译结果”模型中，若能维护一组旧的编译结果，就不怕有人不小心毁坏了文件树。我通常会保存一周或两周的夜间编译结果。当然，夜间编译脚本会把它的输出记录到文件里，存储在编译结果的附近，而且脚本会清除旧的编译结果和记录文件。在记录文件中搜索错误通常会使用awk脚本来进行。为了判断编译结果是否有效，我会为每个 *makefile* 加入一个 validate 工作目标。此工作目标将会对被编译的其他工作目标进行简单的确认。

```
.PHONY: validate_build
validate_build:
        test $(foreach f,$(RELEASE_FILES),-s $f -a) -e .
```

注 2： locate 数据库中包含了文件系统上所有文件名的索引。当你要以名称来查找文件时，这是比较快的方法。我发现这个数据库非常适合用来管理大型的源文件树，而且当编译结果完成之后，你会想要在夜间对它进行更新的动作。

此脚本只会测试一组你所需要的文件是否存在而且非空白。当然，这并不是一个正式的检查，这只是在对编译的结果做简单的“健康检查”（sanity check）。如果此项测试返回失败的结果，make就会返回失败的结果，于是夜间编译脚本（nightly build script）最后所留下的符号链接将会指向旧的编译结果。

第三方程序库（third-party library）总是有点难管理。我同意一般人所认为的，将大型的二进制文件存入CVS并不恰当。这是因为CVS无法存储二进制文件的前后版本差异之处，所以底层的RCS文件可能会变得非常庞大。非常大型的文件在CVS仓库中会使许多常用的CVS操作变慢，因此会对项目的开发造成影响。

如果第三方程序库并未存入CVS，那么你必须使用其他方法来管理它们。我们目前偏好的做法就是在引用树中为程序库创建一个目录，以及在目录名称里记录程序库的版本编号，如图8-1所示。

图8-1：第三方程序库的目录布局

*makefile*将会引用这些目录名称：

```
ORACLE_9011_DIR ?= /reftree/third_party/oracle-9.0.1.1/Ora90
ORACLE_9011_JAR ?= $(ORACLE_9011_DIR)/jdbc/lib/classes12.jar
```

当厂商更新其程序库时，我们就会在引用树中创建一个新的目录并在*makefile*里声明新的变量。这样，*makefile*在标记（tag）和分支（branch）被适当维护的状况下，总是可以明确反映其所用的版本。

安装程序也很麻烦。我相信，把基本的编译过程与创建“安装程序图像”（installer image）的过程分开，是件好事。当前的安装程序工具既复杂又容易出问题，把它们存放到（同常也是既复杂又容易出问题的）编译系统中将会产生难以维护的系统。倒不如让基本的编译过程将它的结果写入一个“发行版”的目录中，使得这个目录包含安装程序编译工具需要用到的所有（或大部分）数据。而且这个工具可能会通过它自己的*makefile*的驱动，最后产生出可执行的“安装程序图像”。

第九章

Java

许多Java开发人员喜欢使用集成开发环境（Integrated Development Environment，简称IDE），比如Eclipse。读者可能会问，既然已经有Java IDE和Ant可用，还需要在Java项目上使用make吗？本章将会探索在此类状况下使用make有什么好处，特别是，读者会看到一个通用的*makefile*稍加修改几乎就可以使用在任何的Java项目上，而且可以完成所有标准的重编译工作。

使用make来处理Java将会引发某些问题，也带来了若干机会。这么做，主要是考虑到三项因素：Java编译器javac的运行速度非常快；标准的Java编译器支持@filename语法，可用来从一个文件中读取"命令行参数"；如果用到Java包，必须提供相应的*.class*文件的路径名称。

标准的Java编译器的运行速度非常快，这主要与import指令的工作方式有关。如同C语言的#include，这个指令可用来访问外部所定义的符号。然而，Java只会直接读取类文件，而不会重新读取源代码，因为这会需要进行重新解析与分析的动作。由于编译期间，类文件中的符号并不会有所改变，所以编译器会将类文件隐藏起来。这意味着，相比较于C语言，在中型项目里，Java编译器会避免逐字重新读取、剖析和分析上百万行程序代码。大部分Java编译器所进行的是最基本的优化动作，这对效能的改进并不大。事实上，Java的效能主要是靠Java虚拟机（Java virtual machine，简称JVM）本身所进行的较为复杂的即时（just-in-time，简称JIT）优化动作。

多数大型Java项目都会广泛使用Java的包功能。一个类的声明会被封装在一个包（package）里，而包可用来限制相应文件所定义的符号的有效范围。层次式的包名一经定义就等于定义了相应的目录结构。例如，包的名称被定义为a.b.c就等于定义了*a/b/c*的目录结构，而a.b.c包中所声明的程序代码将会被编译成存放在*a/b/c*目录里的类文件。这意味着，make用来关联二进制文件与其源文件的标准算法在此处派不上用

场。不过，这也意味着，不必使用 -o 选项来指出输出文件应该存放在何处，指出输出文件树的根目录（对所有文件而言都一样）就够了。这表示，相同的命令行调用方式就可以编译来自不同目录的源文件。

标准的Java编译器都支持 @filename的语法，这使得命令行参数可以从一个文件中被读取。与包的功能并用时，这么做相当有用。因为这意味着，你只要运行一次Java编译器就可以编译项目中所有的Java源代码。这会使得效能有较大的提升，因为加载和运行编译器所花的时间基本上就是编译时间。

总结如下：通过适当的命令行，在2.5-GHz Pentium 4的处理器上编译400000行的Java源代码，大约需要3分钟的时间，编译等效的C++应用程序却需要数小时。

make的替代方案

正如之前提到的，Java开发者社群非常喜欢采用新的技术。接下来让我们对其中两个——Ant和IDE，来跟make进行比较。

Ant

Java社群非常活跃，新工具和新API的产生速度令人印象深刻。Ant便是这些新工具中的一个，这个编译工具企图在Java开发过程中取代make的地位。如同make一样，Ant会使用一个描述文件来指出项目的工作目标与必要条件，与make不同的是，Ant是用Java写成的，而且Ant编译文件是用XML写成的。

让我们来感觉一下XML格式的编译文件的样子，如下的内容是摘录自Ant编译文件：

```
<target name="build"
        depends="prepare, check_for_optional_packages"
        description="--> compiles the source code">
  <mkdir dir="${build.dir}"/>
  <mkdir dir="${build.classes}"/>
  <mkdir dir="${build.lib}"/>

  <javac srcdir="${java.dir}"
         destdir="${build.classes}"
         debug="${debug}"
         deprecation="${deprecation}"
         target="${javac.target}"
         optimize="${optimize}" >
    <classpath refid="classpath"/>
  </javac>

...
```

```
  <copy todir="${build.classes}">
    <fileset dir="${java.dir}">
      <include name="**/*.properties"/>
      <include name="**/*.dtd"/>
    </fileset>
  </copy>
</target>
```

如你所见，工作目标是由<target>这个XML标记定义的。每个工作目标都具有一个名称和依存列表，分别是由属性<name>和<depends>定义的。实际的编译动作是由Ant的任务（task）执行的。Ant的每项任务是由Java写成，而且会被绑定到一个XML标记。例如，创建目录的任务是由<mkdir>标记来指定的，这会触发Java方法Mkdir.execute的执行动作，最后会调用File.mkdir。Ant会尽量以Java API来实现所有任务。

一个供make使用的等效编译文件将会像下面这样：

```
# 编译源代码
build: $(all_javas) prepare check_for_optional_packages
        $(MKDIR) -p $(build.dir) $(build.classes) $(build.lib)
        $(JAVAC) -sourcepath $(java.dir)                    \
                 -d $(build.classes)                        \
                 $(debug)                                   \
                 $(deprecation)                             \
                 -target $(javac.target)                    \
                 $(optimize)                                \
                 -classpath $(classpath)                    \
                 @$<

...
$(FIND) . \( -name '*.properties' -o -name '*.dtd' \) | \
$(TAR) -c -f - -T - | $(TAR) -C $(build.classes) -x -f -
```

这个*makefile*片段使用了本书尚未介绍的技术，简单地说，就是在必要条件*all_javas*中包含需要编译的*.java*文件列表。<mkdir>、<javac>和<copy>等Ant任务还可以用来进行依存关系检查。也就是说，如果目录已经存在，mkdir并不会被执行。同样地，如果Java类文件的时间戳在源文件（的时间戳）之后，就不会对源文件进行编译的动作。然而，基本上，make的脚本所执行的是相同的功能。Ant还提供了一个通用的任务，称为<exec>，可用来运行一个本地程序。

Ant是一个精明且新颖的编译工具，然而，有几个相关的议题值得我们加以探讨：

- 尽管Ant在Java社群中广受欢迎，但它在其他地方却默默无闻。此外，如果说它普及的程度会远超过Java似乎颇令人怀疑（基于此处所列的理由）。另一方面，make一贯被广泛应用在各种领域中，包括软件开发、文件处理与排版以及网站与工作站的维护等。需要在不同的软件系统上工作的人都知道make是个重要的工具。

- 尽管以Java为基础的工具适合选择XML作为描述语言，但是(对许多人来说)XML的读写并不会特别容易。要找到令人满意的XML编译器并不容易，而且通常无法跟现有的工具有很好的集成(如果我的集成开发环境没有提供令人满意的XML编译器，我就得放弃自己的IDE，另外找一个独立的工具)。正如你在之前的例子中所看到的，相比较于make和shell的语法，XML和它的Ant方言似乎太冗长了。
- 编写Ant编译文件的时候，你还必须应付另一个间接的层次。Ant的<mkdir>任务并不会为你的系统调用底层的mkdir程序。事实上，它会执行java.io.File类中的mkdir()方法。这可能是也可能不是你想要的结果。基本上，一个使用过一般工具的程序设计人员应该会对此感到怀疑，而且会去查看Ant文件、Java文件或者Ant的源代码。此外，为了调用Java编译器，我不得不查看几十个或更多个陌生的XML属性，比如<srcdir>、<debug>等，这些在编译器的手册中都没有提到。相对来说，make脚本完全透明，即为了查看命令实际的行为，我通常会将命令直接键入在一个shell中。
- 毫无疑问地，Ant具有可移植性，make也是如此。如第七章所示，编写具可移植性的*makefile*，就如同编写具可移植性的Ant描述文件一样，需要具备相关的经验和知识。程序员编写具可移植性的*makefile*已经有20年的历史了。此外，Ant的说明文件提到了一些可移植性的问题：Unix上的符号链接，Windows上的长文件名，Apple操作系统方面只支持MacOSX，而且不保证会支持其他的平台。而且，一些基本的操作，像为文件设定执行位，无法通过Java API来进行，必须使用一个外部的程序。可移植性对Ant来说并不容易也不完全。
- Ant工具无法准确地说明它正在做什么。因为Ant的任务通常不是通过运行shell脚本来实现的，所以要Ant工具显示它的动作很困难。通常，其所显示的信息组成自任务的作者在各个print语句中所加入的自然语言，用户无法通过shell来执行这些被显示出来的语句。相对而言，make所输出的每一行通常就是命令行，用户可将它转贴到一个shell来重新执行。这意味着，Ant的编译动作对想要了解编译过程和工具的开发人员来说没有什么作用。而且，开发人员也不可能随意地通过键盘再利用一个任务的某些部分。
- 最后而且是最重要的，Ant需要你将自己对编译方面的思考模式从脚本式的程序语言转移至非脚本式的程序语言。Ant的任务是用Java写成的，如果某个任务不存在或者不是你想要的，那么你不是必须用Java编写自己的任务，就是必须使用<exec>这个任务。(当然，如果你经常需要用到<exec>这个任务，使用make并利用它的宏、函数以及更简洁的语法反而会比较好。)

  另一方面，脚本语言之所以大行其道，就是因为它解决了此类问题。make已经存活了几乎30年，而且即使不加以扩充也能使用在最复杂的情况下。当然，在这30年间它已经做了不少扩充，其中有许多都是GNU make所构想和实现的。

尽管Ant是一个广受Java社群欢迎的工具，不过在你着手新项目之前应该谨慎考虑Ant是否适合使用在你的开发环境里。希望这一章能够向你证明，make也能符合你的Java编译系统的需要。

IDE

大多数的Java开发人员都会使用集成开发环境（Integrated Development Environment，简称IDE），它（通常）会将编辑器、编译器、调试器以及程序代码浏览器集成到一个图形环境里。实际的例子有开放源代码的Eclipse（*http://www.eclipse.org*）、Emacs JDEE（*http://jdee.sunsite.dk*）以及商业性质的Sun Java Studio（*http://wwws.sun.com/software/sundev/jde*）和JBuilder（*http://www.borland.com/jbuilder*）。这些环境通常知道"编译必要文件以及让应用程序能够运行"的项目编译过程。

如果IDE能够支持以上这些功能，为何我们应该考虑使用make？最明显的理由就是可移植性。如果需要在另一个平台上编译项目，当我们实际移植到新的平台上时，可能会导致编译失败的结果。尽管Java本身具跨平台的可移植性，不过它的支持工具通常做不到。举例来说，如果项目的配置文件可能包含Unix或Windows风格的路径，那么当你在其他的操作系统上运行编译程序的时候，这些路径可能会产生错误。使用make的第二个理由是它支持自动编译的功能。有些IDE支持批编译的功能，有些则不支持；支持此功能的程度也各不相同。最后，IDE对编译功能的支持通常很有限。如果你希望自定义发行版目录结构、从外部的应用程序集成辅助说明文件、支持自动测试功能以及处理分支和平行的开发线，你可能会发现IDE无法满足你的需要。

以我自己的经验来说，我发现IDE非常适合小规模或局部的开发工作，不过make可以提供产品编译工作所需要的比较复杂的功能。我通常会使用一个IDE来编写和调试程序代码，并为产品的编译和发行编写一个*makefile*。开发期间我会使用该IDE将项目编译成适合调试的状态。但如果源代码产生器的输入文件有许多遭到变更或修改，我就会运行*makefile*。就我使用过的IDE来说，它们对外部的源代码产生工具的支持都不是很好。通常，一个IDE的编译结果并不适合发布给内部或外部的客户使用。要进行此类工作时我都会使用make。

一个通用的Java makefile

例9-1可以看到一个通用的Java *makefile*，本章稍后将会说明它的每一个部分。

例9-1：一个通用的Java makefile

```
# Java项目的通用makefile

VERSION_NUMBER := 1.0
```

```
# 文件树的位置
SOURCE_DIR  := src
OUTPUT_DIR  := classes

# Unix工具
AWK         := awk
FIND        := /bin/find
MKDIR       := mkdir -p
RM          := rm -rf
SHELL       := /bin/bash

# 支持工具的路径
JAVA_HOME   := /opt/j2sdk1.4.2_03
AXIS_HOME   := /opt/axis-1_1
TOMCAT_HOME := /opt/jakarta-tomcat-5.0.18
XERCES_HOME := /opt/xerces-1_4_4
JUNIT_HOME  := /opt/junit3.8.1

# Java工具
JAVA        := $(JAVA_HOME)/bin/java
JAVAC       := $(JAVA_HOME)/bin/javac

JFLAGS      := -sourcepath $(SOURCE_DIR)        \
               -d $(OUTPUT_DIR)                 \
               -source 1.4

JVMFLAGS    := -ea                              \
               -esa                             \
               -Xfuture

JVM         := $(JAVA) $(JVMFLAGS)

JAR         := $(JAVA_HOME)/bin/jar
JARFLAGS    := cf

JAVADOC     := $(JAVA_HOME)/bin/javadoc
JDFLAGS     := -sourcepath $(SOURCE_DIR)        \
               -d $(OUTPUT_DIR)                 \
               -link http://java.sun.com/products/jdk/1.4/docs/api

# Jars
COMMONS_LOGGING_JAR   := $(AXIS_HOME)/lib/commons-logging.jar
LOG4J_JAR             := $(AXIS_HOME)/lib/log4j-1.2.8.jar
XERCES_JAR            := $(XERCES_HOME)/xerces.jar
JUNIT_JAR             := $(JUNIT_HOME)/junit.jar

# 设定Java classpath
class_path := OUTPUT_DIR                  \
              XERCES_JAR                  \
              COMMONS_LOGGING_JAR         \
              LOG4J_JAR                   \
              JUNIT_JAR

# space - A blank space
space := $(empty) $(empty)
```

```
# $(call build-classpath, variable-list)
define build-classpath
$(strip                                          \
  $(patsubst :%,%,                               \
    $(subst : ,:,                                \
      $(strip                                    \
        $(foreach j,$1,$(call get-file,$j):)))))
endef

# $(call get-file, variable-name)
define get-file
  $(strip                                        \
    $($1)                                        \
    $(if $(call file-exists-eval,$1),,           \
      $(warning The file referenced by variable  \
                '$1' ($($1)) cannot be found)))
endef

# $(call file-exists-eval, variable-name)
define file-exists-eval
  $(strip                                        \
    $(if $($1),,$(warning '$1' has no value))    \
    $(wildcard $($1)))

# $(call brief-help, makefile)
define brief-help
  $(AWK) '$$1 ~ /^[^.][-A-Za-z0-9]*:/                          \
          { print substr($$1, 1, length($$1)-1) }' $1 |        \
  sort |                                                       \
  pr -T -w 80 -4
endef

# $(call file-exists, wildcard-pattern)
file-exists = $(wildcard $1)

# $(call check-file, file-list)
define check-file
  $(foreach f, $1,                                             \
    $(if $(call file-exists, $($f)),,                          \
      $(warning $f ($($f)) is missing)))
endef

# #(call make-temp-dir, root-opt)
define make-temp-dir
  mktemp -t $(if $1,$1,make).XXXXXXXXXX
endef

# MANIFEST_TEMPLATE——宏处理器的manifest输入
MANIFEST_TEMPLATE := src/manifest/manifest.mf
TMP_JAR_DIR       := $(call make-temp-dir)
TMP_MANIFEST      := $(TMP_JAR_DIR)/manifest.mf

# $(call add-manifest, jar, jar-name, manifest-file-opt)
define add-manifest
  $(RM) $(dir $(TMP_MANIFEST))
```

```
  $(MKDIR) $(dir $(TMP_MANIFEST))
  m4 --define=NAME="$(notdir $2)"                          \
     --define=IMPL_VERSION=$(VERSION_NUMBER)               \
     --define=SPEC_VERSION=$(VERSION_NUMBER)               \
     $(if $3,$3,$(MANIFEST_TEMPLATE))                      \
     > $(TMP_MANIFEST)
  $(JAR) -utm $1 $(TMP_MANIFEST)
  $(RM) $(dir $(TMP_MANIFEST))
endef

# $(call make-jar,jar-variable-prefix)
define make-jar
  .PHONY: $1 $$($1_name)
  $1: $($1_name)
  $$($1_name):
        cd $(OUTPUT_DIR); \
        $(JAR) $(JARFLAGS) $$(notdir $$@) $$($1_packages)
        $$(call add-manifest, $$@, $$($1_name), $$($1_manifest))
endef

# 设定CLASSPATH
export CLASSPATH := $(call build-classpath, $(class_path))

# make directories——确定输出目录的存在
make-directories := $(shell $(MKDIR) $(OUTPUT_DIR))
# help——默认目标
.PHONY: help
help:
        @$(call brief-help, $(CURDIR)/Makefile)

# all——为整个编译过程完成所有任务
.PHONY: all
all: compile jars javadoc

# all_javas——用来保存源文件列表的临时文件
all_javas := $(OUTPUT_DIR)/all.javas

# compile——编译源文件
.PHONY: compile
compile: $(all_javas)
        $(JAVAC) $(JFLAGS) @$<

# all_javas——收集源文件列表
.INTERMEDIATE: $(all_javas)
$(all_javas):
        $(FIND) $(SOURCE_DIR) -name '*.java' > $@

# jar_list——jar文件列表
jar_list := server_jar ui_jar

# jars——创建所有的jar文件
.PHONY: jars
jars: $(jar_list)
```

```
# server_jar——创建$(server_jar)
server_jar_name     := $(OUTPUT_DIR)/lib/a.jar
server_jar_manifest := src/com/company/manifest/foo.mf
server_jar_packages := com/company/m com/company/n

# ui_jar——创建$(ui_jar)
ui_jar_name     := $(OUTPUT_DIR)/lib/b.jar
ui_jar_manifest := src/com/company/manifest/bar.mf
ui_jar_packages := com/company/o com/company/p

# 为每个jar文件自定义一个规则
# $(foreach j, $(jar_list), $(eval $(call make-jar,$j)))
$(eval $(call make-jar,server_jar))
$(eval $(call make-jar,ui_jar))

# javadoc——从源文件产生Java doc
.PHONY: javadoc
javadoc: $(all_javas)
        $(JAVADOC) $(JDFLAGS) @$<

.PHONY: clean
clean:
        $(RM) $(OUTPUT_DIR)

.PHONY: classpath
classpath:
        @echo CLASSPATH='$(CLASSPATH)'

.PHONY: check-config
check-config:
        @echo Checking configuration...
        $(call check-file, $(class_path) JAVA_HOME)

.PHONY: print
print:
        $(foreach v, $(V), \
          $(warning $v = $($v)))
```

编译 Java

使用make来编译Java可以有两种方式：传统方式，就是为每个源文件执行一次javac；快速方式，就是之前所说的使用@filename。

快速方式：一次做好编译

让我们从快速方式开始。正如你在这个通用的*makefile*中所看到的：

```
# all_javas——用来保存源文件列表的临时文件
all_javas := $(OUTPUT_DIR)/all.javas
```

```
# compile——编译源文件
.PHONY: compile
compile: $(all_javas)
        $(JAVAC) $(JFLAGS) @$<

# all_javas——收集源文件列表
.INTERMEDIATE: $(all_javas)
$(all_javas):
        $(FIND) $(SOURCE_DIR) -name '*.java' > $@
```

假想工作目标compile将会对项目中的所有源文件调用一次javac。

$(all_javas)必要条件是一个文件，这个文件名为*all.javas*，内容为一份Java文件列表，每行一个文件名。尽管不需要每个文件名自成一行，但是这么做之后，如果需要使用grep -v的话，文件的过滤会容易许多。用来创建*all.javas*文件的规则被声明为.INTERMEDIATE，所以每次运行之后make便会移除*all.javas*文件，也因此在每次编译之前make就会创建一个新的*all.javas*文件。用来创建*all.javas*文件的命令脚本相当简单。为了获得最高的可维护性，我们会使用find命令在源文件树中取出所有的*.java*文件。虽然这个命令执行起来有点慢，不过当源文件树有所变动时，保证几乎不用修改就能正常工作。

如果你的*makefile*中已经准备了一份源文件目录列表，就可以使用运行速度较快的命令脚本来创建*all.javas*文件。如果这是一份中等长度的源文件目录列表，那么命令行的长度应该不会超过操作系统的限制。下面就是这个运行速度较快的脚本：

```
$(all_javas):
        shopt -s nullglob; \
        printf "%s\n" $(addsuffix /*.java,$(PACKAGE_DIRS)) > $@
```

这个脚本将会使用shell通配符来确定每个目录里的*.java*文件列表。然而，如果某个目录并未包含任何*.java*文件，我们会想让通配符产生空字符串，而不是原本的文件名匹配模式（这是许多shell默认的行为）。为了产生此结果，我们会使用bash选项-s nullglob，其他的shell大部分都会提供类似的选项。最后，我们会使用文件名匹配（globbing）和printf（而不会使用ls -l），因为它们都是bash内置的功能，所以我们的命令脚本只会运行一个程序，不论包有多少个目录。

此外，如果不想使用shell的文件名匹配功能，那么可以使用wildcard：

```
$(all_javas):
        print "%s\n" $(wildcard \
                        $(addsuffix /*.java,$(PACKAGE_DIRS))) > $@
```

如果你有非常多的源文件目录（或相当长的路径），如上的脚本可能会超过操作系统的命令行长度限制。这个时候，使用下面的脚本可能会比较好：

```
.INTERMEDIATE: $(all_javas)
$(all_javas):
        shopt -s nullglob;              \
        for f in $(PACKAGE_DIRS);       \
        do                              \
          printf "%s\n" $$f/*.java;     \
        done > $@
```

请注意，compile工作目标与相关的支持规则所依照的是非递归的编译方法，不管有多少子目录，我们只会有一个*makefile*，而且只运行一次编译器。如果你想要编译所有的源文件，这是最快的方式。

此外，我们完全没有用到任何依存信息。因为我们所自定义的这些规则中，make既不知道也不在乎哪个文件的时间戳在哪个文件（的时间戳）之后，它只会在每次被调用时编译所有源文件。这么做有个好处，我们可以从源文件树来运行*makefile*，而不必从二进制文件树来运行。如果考虑到make管理依存关系的能力，这看起来是个没有什么用的编写这个*makefile*的方法，但是考虑以下几点：

- 有一个替代方案（稍后将会讨论到）就是采用标准的依存关系做法。它会为每个文件调用一个新的javac进程，因此增加了许多开销。但是，如果项目规模小，编译所有源文件所花的时间与编译少数几个文件所花的时间并不会相差太多，因为javac编译器的运行速度很快而进程的建立通常很慢。任何编译工作所花的时间如果少于15秒，基本上说明了跟它做多少工作无关。例如，在我的机器（CPU为1.8-GHz Pentium 4、RAM为512 MB）上，编译500个左右的源文件（来自Ant发行包）需要14秒的时间，编译一个文件需要5秒的时间。
- 多数开发人员都将会使用可以为个别文件提供快速编译功能的某种开发环境。不过，如果变动的范围较大、需要进行全面的重编译工作或是需要进行自动的编译工作，他们最可能想要使用的就是*makefile*。
- 正如我们所看到的，实现和维护依存关系所获得的成果，如同我们为C/C++项目进行分离源文件树和二进制文件树的编译工作（参见第八章）一样，没有任何一项工作应该被轻视。

正如我们在前面最后几个例子所看到的，它们使用的是PACKAGE_DIRS变量而不是去创建*all.javas*文件。但是这个变量的维护，可能是一个劳动密集并具有潜在困难的工作。对于较小型的项目来说，以手动方式维护*makefile*里的目录列表不会有什么问题，但是当目录的数量增长到超过100个的时候，手动维护将会变得容易出错而且令人厌烦。这个时候，应该使用find来扫描这些目录：

```
# $(call find-compilation-dirs, root-directory)
find-compilation-dirs =                         \
```

```
    $(patsubst %/,%,                                    \
      $(sort                                            \
        $(dir                                           \
          $(shell $(FIND) $1 -name '*.java'))))

PACKAGE_DIRS := $(call find-compilation-dirs, $(SOURCE_DIR))
```

find命令会返回一份文件列表，dir函数会丢掉文件只留下目录，sort函数会从列表中移除重复的部分，而patsubst会删除结尾的斜线。请注意，find-compilation-dirs函数将会找出需要编译的源文件列表，只不过它会去掉文件名部分，然后由*all_javas*规则以通配符取回文件名。这似乎是在浪费时间，不过我发现能够准备好包（内含源代码）列表，这对编译过程的其他部分来说通常很有用，例如EJB配置文件的扫描。如果你的情况不需要包列表，那么你就可以使用先前所提到的较简单的方法来建立*all.javas*文件。

编译时考虑依存关系

编译时若要进行完整的依存关系检查，你首先需要使用一个工具从Java源文件中取出依存信息，这有点类似cc -M。Jikes（*http://jikes.sourceforge.net/*）是一个开放源代码的Java编译器，它针对此功能提供了--makefile或+M选项。Jikes不太适合在源文件与二进制文件分开的状态下进行编译的工作，因为它总是会将依存文件写入源文件所在的目录，不过要这么做也行。从好的方面来看，它会在编译的同时产生依存文件，不用分成两个阶段。

下面是一个依存关系处理函数以及一个使用此函数的规则：

```
%.class: %.java
        $(JAVAC) $(JFLAGS) +M $<
        $(call java-process-depend,$<,$@)

# $(call java-process-depend, source-file, object-file)
define java-process-depend
  $(SED) -e 's/^.*\.class *:/$2 $(subst .class,.d,$2):/'   \
         $(subst .java,.u,$1) > $(subst .class,.tmp,$2)
  $(SED) -e 's/#.*//'                                      \
         -e 's/^[^:]*: *//'                                \
         -e 's/ *\\$$$$//'                                 \
         -e '/^$$$$/ d'                                    \
         -e 's/$$$$/ :/' $(subst .class,.tmp,$2)           \
         >> $(subst .class,.tmp,$2)
  $(MV) $(subst .class,.tmp,$2) tmp  $(subst .class,.d,$2)
endef
```

这需要你从二进制文件树来运行此*makefile*，并且将vpath设定成可以找到源文件。如果你只想将Jikes编译器用在依存关系的产生上，且使用不同的编译器来产生实际的二进制码，你可以使用+B选项来避免Jikes产生bytecode。

如果我们以先前所提到的快速编译方式，对223个 *.java* 文件调用一次Java编译器，在我的机器上需要9.9秒的时间。相同的这223个文件，如果为每个文件调用一次Java编译器，则需要411.6秒的时间，是前者的41.5倍。此外，在源文件与二进制文件分开的状态下进行编译，任何的编译动作都需要多编译4个文件，所以会比对所有源文件调用一次Java编译器还慢。如果依存关系的产生和编译是由不同的程序进行的，则会增加不一致性。

当然，开发环境各有不同，重点在于你想达成什么样的目标。将所编译的文件数量最小化，并不一定能够将编译一个系统所花的时间最小化。对Java而言尤其是这样，一般的程序开发环境并不需要进行完整的依存关系检查以及将所编译的文件数量最小化。

设定 CLASSPATH

当你以Java开发软件时，最重要的议题之一就是正确设定CLASSPATH变量。当进行类引用的解析时，将会根据此变量来决定应该加载哪个程序代码。要正确编译Java应用程序，*makefile* 中必须包含正确的CLASSPATH。当有许多Java包、API和支持工具被加入系统时，CLASSPATH马上就会变得既冗长且复杂。如果CLASSPATH很难正确设定，最好只在一处设定它。

我发现如下的技术非常适合在*makefile*中用来为它自己和其他程序设定CLASSPATH。例如，工作目标classpath将会把CLASSPATH返回给调用如下 *makefile* 的shell：

```
.PHONY: classpath
classpath:
        @echo "export CLASSPATH='$(CLASSPATH)'"
```

开发人员可以用如下的方式来设定他们的CLASSPATH（如果他们使用的是bash）：

```
$ eval $(make classpath)
```

CLASSPATH 在 Windows 环境中可以用如下的方式来设定：

```
.PHONY: windows_classpath
windows_classpath:
        regtool set /user/Environment/CLASSPATH "$(subst /,\\,$(CLASSPATH))"
        control sysdm.cpl,@1,4 &
        @echo "Now click Environment Variables, then OK, then OK again."
```

regtool 程序是Cygwin开发系统中的一个实用程序，可用来操作Windows Registry。然而，只是设定Registry并不会使得Windows读取新的设定值。解决此事的一个方法就是查看“Environment Variable”（环境变量）对话框，然后按“OK”（确定）按钮两次。

命令脚本的第二行将会使Windows显示“System Properties”（系统内容）对话框并且选定“Advanced”（高级）页面。可惜，这个命令无法显示“Environment Variable”（环境变量）对话框并为我们按下“OK”（确定）按钮两次，所以需要最后一行来提醒用户完成此工作。

将CLASSPATH导出到其他程序，比如Emacs JDEE或Jbuilder的项目文件（project file）中，并不困难。

CLASSPATH本身的设定也可以通过make来管理。以如下明确的方式来设定CLASSPATH变量是绝对没问题的：

```
CLASSPATH = /third_party/toplink-2.5/Toplink.jar:/third_party/...
```

为了方便维护，使用变量会更好：

```
CLASSPATH = $(TOPLINK_25_JAR):$(TOPLINKX_25_JAR):...
```

但是我们还可以做得更好。正如你在通用的*makefile*中所看到的，CLASSPATH的建立可以分成两个阶段：首先将路径中各个元素列成make变量，然后将这些变量转换成环境变量的字符串值：

```
# 设定Java classpath
class_path := OUTPUT_DIR
              XERCES_JAR
              COMMONS_LOGGING_JAR
              LOG4J_JAR
              JUNIT_JAR

...
# 设定CLASSPATH
export CLASSPATH := $(call build-classpath, $(class_path))
```

（例9-1里的CLASSPATH的示范性质高于实用性。）一个好的build-classpath函数应该可以解决以下几个问题：

- 可以轻易地将各个片段组合成一个CLASSPATH。举例来说，如果使用不同的应用服务器，CLASSPATH可能会有所不同。那么你就可以把不同版本的CLASSPATH放在各个ifdef区段中，并通过make变量的设定加以选择。
- *makefile*的临时维护人员不必担心内置的空格、换行符号或续行符号，因为build-classpath函数可以处理这些问题。
- build-classpath函数可以自动选择路径分隔符（path separator），因此不管你是在Unix或Windows上运行，它都是正确的。

- build-classpath函数可以确认路径中各元素的正确性。特别是，make有一个麻烦的问题，就是未定义的变量会被扩展成空字符串而且不会产生错误。在大多数的情况下这个特性非常有用，不过有时候这会妨碍到你。在此情况下，它会闷不吭声地为CLASSPATH变量产生错误的设定值（注1）。我们可以通过build-classpath函数来检查具空值的元素并提出警告来解决这个问题。
- 最后，一个用来处理CLASSPATH的挂钩将有助于实现高级的功能，像协助调整路径名称中内置的空格以及搜索路径。

如下的build-classpath实现可以解决前三个问题：

```
# $(call build-classpath, variable-list)
define build-classpath
$(strip                                          \
  $(patsubst %:,%,                               \
    $(subst ¦ ,:,                                \
      $(strip                                    \
        $(foreach c,$1,$(call get-file,$c)¦)))))
endef

# $(call get-file, variable-name)
define get-file
  $(strip                                        \
    $($1)                                        \
    $(if $(call file-exists-eval,$1),,           \
      $(warning The file referenced by variable  \
                '$1' ($($1)) cannot be found)))
endef

# $(call file-exists-eval, variable-name)
define file-exists-eval
  $(strip                                        \
    $(if $($1),,$(warning '$1' has no value))    \
    $(wildcard $($1)))
endef
```

build-classpath函数将会反复处理它的参数中所包含的每个单词，确认每个元素并且为他们衔接上路径分隔符（此例为:）。要自动选择路径分隔符现在很容易。然后，此函数会删除由get-file函数与foreach循环所加上的空格。接着，它会删除由foreach循环所加上的最后一个分隔符。最后，就是将整个东西封装在strip中，这将会移除续行符所带进来的多余空格。

get-file函数将会返回它的文件名参数，然后测试变量所引用的文件是否存在。如果文件不存在，它会产生一个警告信息。然而，不管文件是否存在，它都会返回变量的值，

注1：我们将会使用 --warn-undefined-variables选项来检查此情况，不过这同时会影响到其他需要使用此特性的地方。

因为这个值对调用者可能有用。有时，get-file可能会被用来测试即将产生但尚不存在的文件。

最后一个函数file-exists-eval的参数是一个变量名称（内含文件引用）。如果这个变量的值是空的，则会产生警告信息；否则，wildcard函数会将此值解析成一个文件（或一份文件列表）。

当build-classpath函数处理到某些有问题的值时，我们将会看到如下的错误信息：

```
Makefile:37: The file referenced by variable 'TOPLINKX_25_JAR'
             (/usr/java/toplink-2.5/TopLinkX.jar) cannot be found
...
Makefile:37: 'XERCES_142_JAR' has no value
Makefile:37: The file referenced by variable
             'XERCES_142_JAR' () cannot be found
```

相较于闷不吭声的简单做法，这是一个很大的改进。

使用get-file函数来检查文件是否存在，倒不如将输入文件的搜索一般化。

```
# $(call get-jar, variable-name)
define get-jar
  $(strip                                                    \
    $(if $($1),,$(warning '$1' is empty))                    \
    $(if $(JAR_PATH),,$(warning JAR_PATH is empty))          \
    $(foreach d, $(dir $($1)) $(JAR_PATH),                   \
      $(if $(wildcard $d/$(notdir $($1))),                   \
        $(if $(get-jar-return),,                             \
          $(eval get-jar-return := $d/$(notdir $($1))))))    \
    $(if $(get-jar-return),                                  \
      $(get-jar-return)                                      \
      $(eval get-jar-return :=),                             \
      $($1)                                                  \
      $(warning get-jar: File not found '$1' in $(JAR_PATH))))
endef
```

此处，我们将JAR_PATH变量定义成内含文件的搜索路径，第一个被找到的文件将会被返回。此函数的参数是一个变量名称（内含一个jar文件的路径）。我们想要先在变量所指定的路径中查找jar文件，然后在JAR_PATH中查找。为完成此事，foreach循环中的目录列表将会由来自变量的目录组成，后面跟着JAR_PATH。另外，此参数将会在两次notdir调用中使用到，所以jar的名称可以由此列表中的路径组成。请注意，不能马上结束每个foreach循环，应该使用eval来设定get-jar-return变量，以便记住我们所找到的第一个文件。处理完循环之后，我们不是返回这个临时变量的值，就是显示警告信息（如果没有找到任何文件的话）。我们必须记得在终止此宏之前重新设定我们的返回值。

这本质上是在设定 CLASSPATH 的语境中重新实现 vpath 的功能。让我们复习一下：vpath 是make内部所使用的搜索路径，用来查找以相对路径在当前目录中找不到的必要条件。在此情况下，make会到vpath查找必要条件文件，以及将完整的路径插入$^、$?和 $+ 等自动变量中。为了设定 CLASSPATH，我们会让 make 为每个 jar 文件搜索路径并将完整的路径插入 CLASSPATH变量。因为make对此没有内置的支持，所以我们必须自己提供。当然，你可以使用适当的jar 文件名来扩展jar 路径变量，让Java 进行搜索的动作，但是 CLASSPATH 马上就会变得很长。在某些操作系统上，环境变量的空间有限，CLASSPATH的内容太长将有可能被截断。在 Windows XP 上，一个环境变量的长度限制为 1023 个字符。此外，即使 CLASSPATH的内容没有被截断，Java 虚拟机必须在加载类的时候搜索 CLASSPATH，因此也会拖慢应用程序的运行速度。

管理 jar

在Java 中建立和管理jar 所引发的问题跟 C/C++的程序库的不一样。理由有三个：首先，jar中的成员包含了相对路径，所以必须小心处理传递给jar程序的文件名，其次，在Java 中有一个趋势，就是将各个jar 合并成一个 jar以利于发行，最后，jar 中不止类文件，它还包含了 manifest、property 和 XML 等文件。

在 GNU make 中用来创建 jar 的基本命令为：

```
JAR      := jar
JARFLAGS := -cf

$(FOO_JAR): prerequisites...
        $(JAR) $(JARFLAGS) $@ $^
```

jar程序所接受的是目录而不是文件名，在此情况下，jar中将会包含目录树里的所有文件。这将会非常方便，特别是当你想要使用 -C 选项来变更目录的时候：

```
JAR      := jar
JARFLAGS :=  cf

.PHONY: $(FOO_JAR)
$(FOO_JAR):
        $(JAR) $(JARFLAGS) $@ -C $(OUTPUT_DIR) com
```

此处我们把jar本身声明为 .PHONY，否则，下次运行*makefile*的时候将不会重编译该文件，因为它没有必要条件。如同前面章节所介绍的ar 命令，这似乎有点像在使用更新标记-u，因为它需要花相同的时间或是较长的时间(如果是从头开始重新建立jar的话)，至少对大部分的更新动作而言是这样。

一个jar通常会包含一个*manifest*文件，内含厂商、API和jar版本编号等信息。一个简单的manifest文件可能会像这样：

```
Name: JAR_NAME
Specification-Title: SPEC_NAME
Implementation-Version: IMPL_VERSION
Specification-Vendor: Generic Innovative Company, Inc.
```

这个manifest文件具有三个占位符：JAR_NAME、SPEC_NAME和IMPL_VERSION，这让make可以在建立jar的时候，使用sed、m4或任何其他流编辑器，将它们替换成实际的值。你可以使用如下的函数来处理manifest文件：

```
MANIFEST_TEMPLATE := src/manifests/default.mf
TMP_JAR_DIR       := $(call make-temp-dir)
TMP_MANIFEST      := $(TMP_JAR_DIR)/manifest.mf

# $(call add-manifest, jar, jar-name, manifest-file-opt)
define add-manifest
  $(RM) $(dir $(TMP_MANIFEST))
  $(MKDIR) $(dir $(TMP_MANIFEST))
  m4 --define=NAME="$(notdir $2)"                    \
     --define=IMPL_VERSION=$(VERSION_NUMBER)         \
     --define=SPEC_VERSION=$(VERSION_NUMBER)         \
     $(if $3,$3,$(MANIFEST_TEMPLATE))                \
     > $(TMP_MANIFEST)
  $(JAR) -ufm $1 $(TMP_MANIFEST)
  $(RM) $(dir $(TMP_MANIFEST))
endef
```

add-manifest函数所操作的manifest文件如同之前你所看到的：此函数首先会创建一个临时目录，接着会扩展manifest实例，然后它会更新jar，而且最后会删除临时目录。请注意，此函数的最后一个参数是一个选项。如果manifest文件路径是个空值，则此函数将会使用MANIFEST_TEMPLATE的值。

这个通用的*makefile*将以上这些操作纳入到了一个通用的函数中，利用此函数来为jar的建立编写具体规则：

```
# $(call make-jar,jar-variable-prefix)
define make-jar
  .PHONY: $1 $$($1_name)
  $1: $($1_name)
  $$($1_name):
        cd $(OUTPUT_DIR); \
        $(JAR) $(JARFLAGS) $$(notdir $$@) $$($1_packages)
        $$(call add-manifest, $$@, $$($1_name), $$($1_manifest))
endef
```

此函数的参数只有一个，也就是make变量的前缀，这样可确定一组变量来描述以下四

个jar参数：工作目标的名称、jar的名称、jar中所包含的包以及jar的manifest文件。举例来说，如果要建立一个名为*ui.jar*的文件，我们可以这么做：

```
ui_jar_name     := $(OUTPUT_DIR)/lib/ui.jar
ui_jar_manifest := src/com/company/ui/manifest.mf
ui_jar_packages := src/com/company/ui \
                   src/com/company/lib

$(eval $(call make-jar,ui_jar))
```

通过组合变量的名称，不仅可以缩短函数的调用过程，也可以让函数的实现有很多的灵活性。

如果许多jar文件要创建，我们可以通过把这些jar文件的名称存放在一个变量中，让创建的工作自动进行：

```
jar_list := server_jar ui_jar

.PHONY: jars $(jar_list)
jars: $(jar_list)

$(foreach j, $(jar_list),\
  $(eval $(call make-jar,$j)))
```

有时，我们可能需要将一个jar文件扩展并放到一个临时目录中。如下的简单函数可以为我们完成此事：

```
# $(call burst-jar, jar-file, target-directory)
define burst-jar
  $(call make dir,$2)
  cd $2; $(JAR) -xf $1
endef
```

引用树与来自第三方的jar文件

如果要使用一个共享的引用树来为开发人员提供部分的源文件树，只要让每夜编译动作为项目建立jar文件并且将那些jar文件的路径加入Java编译器的CLASSPATH中就行了。开发人员可以从源文件树中调出自己所需要的部分并且进行编译（假定原始列表是由find之类的程序自动建立的）。当Java编译器需要某个短缺的源文件里的某些符号时，它将会搜索CLASSPATH并且在jar文件中相应的*.class*文件里找到。

要从一个引用树中选出第三方所提供的jar文件也很简单，将该jar文件的路径放入CLASSPATH就行了。正如之前所提到的，*makefile*非常适合用来管理此过程。当然，get-file函数只要通过JAR_PATH变量的设定，就可以被用来自动选出测试版或稳定版、本地或远程的jar文件。

Enterprise JavaBeans

Enterprise JavaBeans是一个功能强大的技术，可用来在远程函数调用（remote method invocation）的结构中封装和重用商业逻辑。EJB所建立的Java类会被用来实现成供远程客户端程序使用的服务器端API。这些对象和服务的配置是通过以XML为基础的控制文件进行的。一旦编写好Java类和XML控制文件之后，它们必须被一起封装到一个jar文件中，接着使用一个特殊的EJB编译器来建立stub和tie以实现RPC支持程序代码。

将以下的代码插入例9-1将可提供通用的EJB支持：

```
EJB_TMP_JAR = $(EJB_TMP_DIR)/temp.jar
META_INF    = $(EJB_TMP_DIR)/META-INF

# $(call compile-bean, jar-name,
#                      bean-files-wildcard, manifest-name-opt)
define compile-bean
  $(eval EJB_TMP_DIR := $(shell mktemp -d $(TMPDIR)/compile-bean.XXXXXXXX))
  $(MKDIR) $(META_INF)
  $(if $(filter %.xml, $2),cp $(filter %.xml, $2) $(META_INF))
  cd $(OUTPUT_DIR) &&                                  \
  $(JAR) -cf0 $(EJB_TMP_JAR)                           \
         $(call jar-file-arg,$(META_INF))              \
         $(filter-out %.xml, $2)
  $(JVM) weblogic.ejbc $(EJB_TMP_JAR) $1
  $(call add-manifest,$(if $3,$3,$1),.)
  $(RM) $(EJB_TMP_DIR)
endef

# $(call jar-file-arg, jar-file)
jar-file-arg = -C "$(patsubst %/,%,$(dir $1))" $(notdir $1)
```

compile-bean函数可以接受三个参数：jar的名称、jar中文件的列表以及一个可有可无的manifest文件。这个函数首先会使用mktemp函数创建一个干净的临时目录，并且将目录名称存储在变量EJB_TMP_DIR里。通过将赋值动作嵌入eval函数中，可确保每次compile-bean被扩展时，EJB_TMP_DIR都会被重新设定为一个新的临时目录。因为compile-bean将会被用在某个规则的命令脚本中，所以此函数只会在命令脚本运行时被扩展。然后，此函数会把bean文件列表中所包含的所有XML文件复制到*META-INF*目录下，这就是EJB配置文件存放的位置。接着，此函数会建立一个临时的jar以作为EJB编译器的输入。jar-file-arg函数会把dir1/dir2/dir3形式的文件名转换成-C dir1/dir2 dir3，这样，文件在jar中的相对路径才是正确的。当我们为jar命令指定*META-INF*目录时必须使用这种格式。bean文件列表中所包含的*.xml*文件已经被复制到了*META-INF*目录下，所以我们可以把这些文件删除。建立临时的jar之后，调用

WebLogic EJB 编译器以产生我们所要的jar。然后，一个manifest文件（如果被指定的话）会被加入编译好的jar中。最后，移除我们的临时目录。

这个新函数的用法很简单：

```
bean_files = com/company/bean/FooInterface.class        \
             com/company/bean/FooHome.class             \
             src/com/company/bean/ejb-jar.xml           \
             src/com/company/bean/weblogic-ejb-jar.xml

.PHONY: ejb_jar $(EJB_JAR)
ejb_jar: $(EJB_JAR)
$(EJB_JAR):
        $(call compile-bean, $@, $(bean_files), weblogic.mf)
```

bean_files列表令人有些混淆：它所引用的*.class*文件被访问时是相对于类目录，而*.xml*文件被访问时是相对于*makefile*所在的目录。

这么做没什么问题，但如果bean jar中包含了许多bean文件，我们可以自动建立文件列表吗？当然可以：

```
src_dirs := $(SOURCE_DIR)/com/company/...

bean_files =                                            \
  $(patsubst $(SOURCE_DIR)/%,%,                         \
    $(addsuffix /*.class,                               \
      $(sort                                            \
        $(dir                                           \
          $(wildcard                                    \
            $(addsuffix /*Home.java,$(src_dirs))))))))

.PHONY: ejb_jar $(EJB_JAR)
ejb_jar: $(EJB_JAR)
$(EJB_JAR):
        $(call compile-bean, $@, $(bean_files), weblogic.mf)
```

这是假设具有EJB源文件的所有目录都包含在src_dirs变量中（也可能存在并未包含EJB源文件的目录），而且任何文件名以*Home.java*结尾的文件，代表是一个包含EJB程序代码的包。这个用来设定bean_files变量的表达式，首先会为目录加上通配符后缀，然后调用wildcard函数来收集*Home.java*文件列表。文件名的部分会被删掉但保留目录的部分，然后调用sort移除重复的部分，接着加入通配符后缀/*.class。这样，shell将会把列表扩展成实际的类文件。最后，源文件目录前缀（无法使用在类文件树中）会被移除。之所以会使用shell通配符进行扩展的动作，而不使用make的wildcard函数，是因为在类文件完成编译之后，我们无法通过make来进行它的扩展动作。如果make对wildcard函数的求值动作太早进行，它将会找不到文件，而且目录隐藏功能将

会使得它不再重新搜索文件。在源文件树中使用wildcard将会相当安全，因为（我们假设）在make运行的时候将不会有任何源文件被加进来。

当我们的bean jar数量不多时，如上的代码可以应付自如。另一种开发方式是将每个EJB放在它自己的jar中。大型的项目可能包含数10个jar。为了自动处理此状况，我们需要为每个EJB jar产生一个具体规则。在这个例子中，EJB源代码自给自足：每个EJB与相关的XML文件将会被放在单一目录中。你可以通过文件结尾的*Session.java*来找出每个EJB目录。

基本的做法就是在源文件树中搜索EJB，然后为每个EJB建立一个具体规则，将这些规则写入一个文件。EJB规则的建立动作将会受到"make本身对引入文件的依存处理动作"的触发。

```
# session_jars——每个EJB jar以及它们相对源文件的路径
session_jars =
  $(subst .java,.jar,                                  \
    $(wildcard                                         \
      $(addsuffix /*Session.java, $(COMPILATION_DIRS))))

# EJBS——EJB jar列表
EJBS = $(addprefix $(TMP_DIR)/,$(notdir $(session_jars)))

# ejbs——创建所有的EJB jar文件
.PHONY: ejbs
ejbs: $(EJBS)
$(EJBS):
        $(call compile-bean,$@,$^)
```

我们会通过对所有编译目录调用wildcard来找出各个*Session.java*文件。在这个例子中，jar文件就是Session文件的名称加上*.jar*后缀。这些jar文件本身会被放到临时的二进制文件目录中。EJBS变量中所包含的是jar文件列表与它们的二进制文件目录路径。这些EJB jar都是我们相要更新的工作目录，而实际的命令脚本就是我们的compile-bean函数。为每个jar文件将文件列表记录在必要条件中，需要一点技巧。让我们来看它们是如何被建立的。

```
-include $(OUTPUT_DIR)/ejb.d

# $(call ejb-rule, ejb-name)
ejb-rule = $(TMP_DIR)/$(notdir $1):                    \
           $(addprefix $(OUTPUT_DIR)/,                 \
             $(subst .java,.class,                     \
               $(wildcard $(dir $1)*.java))) \
           $(wildcard $(dir $1)*.xml)

# ejb.d - EJB依存文件
$(OUTPUT_DIR)/ejb.d: Makefile
        @echo Computing ejb dependencies...
```

```
@for f in $(session_jars);              \
do                                      \
  echo "\$$(call ejb-rule,$$f)";        \
done > $@
```

每个EJB jar的依存关系会被记录到一个独立的文件*ejb.d*中，这个文件会被*makefile*引入。make首次查找这个文件的时候，它并不存在，所以make会调用更新引入文件的规则。我们为每个EJB编写了一行规则，如下所示：

```
$(call ejb-rule,src/com/company/foo/FooSession.jar)
```

ejb-rule函数将会被扩展成相应的jar工作目标与它的必要条件列表，如下所示：

```
classes/lib/FooSession.jar: classes/com/company/foo/FooHome.jar \
            classes/com/company/foo/FooInterface.jar            \
            classes/com/company/foo/FooSession.jar              \
            src/com/company/foo/ejb-jar.xml                     \
            src/com/company/foo/ejb-weblogic-jar.xml
```

这样，就连大量的jar也可以在make中被管理，而且不必面对必须手动维护一组具体规则所带来的麻烦。

第十章

改进 make 的效能

make 在开发过程中扮演着关键性的角色。它可以将一个项目的各个要素组合起来以建立一个应用程序，而且可以让开发人员避免因为意外省略某些编译步骤所导致的难以捉摸的错误。然而，如果开发人员不想使用 make，因为他们觉得 *makefile* 太慢了，无法获得以上所提到的 make 的所有好处。因此，重点在于如何尽可能地让 *makefile* 有效率地运行。

效能的问题总是难以处理，再加上用户的看法以及不同的描述方式就更难处理了。并非 *makefile* 的每个工作目标都值得你花时间进行优化的动作，即使是基本的优化动作也可能不值得你做，这取决于你的环境。例如，将一项操作从 90 分钟缩短为 45 分钟可能无关痛痒，因为即使速度变快了，也等于是吃一顿午餐的时间。从另一方面来说，即使将一项工作从 2 分钟缩短为 1 分钟，也可能获得喝彩 —— 如果开发人员闲得无聊的话。

当你要编写能够有效运行的 *makefile* 时，必须先了解各种操作的成本以及目前所进行的是哪些操作。在接下来的各节中，我们将会进行若干简单的基准测试（benchmarking）以便将我们的说明量化，并介绍可以协助我们找出瓶颈的技术。

一个可以协助我们改进效能的方法是利用并行和局域网络的技术。如果能够在同一时间运行多个命令脚本（即使机器上只有一个处理器），将可缩短编译时间。

基准测试

这一节我们将会测量 make 中若干基本操作的效能。从表 10-1 中可以看到测量的结果。我们将针对每种测试作出说明，并且提示它们会如何影响你所编写的 *makefile*。

表 10-1：各种操作的成本

操作	执行次数	每次执行所需秒数（Windows）	每秒执行次数（Windows）	每次执行所需秒数（Linux）	每秒执行次数（Linux）
make (bash)	1000	0.0436	22	0.0162	61
make (ash)	1000	0.0413	24	0.0151	66
make (sh)	1000	0.0452	22	0.0159	62
赋值操作	10000	0.0001	8130	0.0001	10,989
subst（短字符串）	10000	0.0003	3891	0.0003	3846
subst（长字符串）	10000	0.0018	547	0.0014	704
sed (bash)	1000	0.0910	10	0.0342	29
sed (ash)	1000	0.0699	14	0.0069	144
sed (sh)	1000	0.0911	10	0.0139	71
shell (bash)	1000	0.0398	25	0.0261	38
shell (ash)	1000	0.0253	39	0.0018	555
shell (sh)	1000	0.0399	25	0.0050	198

Windows上的测试是在CPU为1.9-GHz Pentium 4（大约3578 BogoMips）（注1）、RAM为512 MB、OS为Windows XP的机器上进行的。进行测试的make 3.80使用的是Cygwin的版本，而且是从rxvt窗口启动的。Linux上的测试是在CPU为450-MHz Pentium 2（891 BogoMips）、RAM为256 MB、OS为RedHat 9的机器上进行的。

make所使用的subshell对*makefile*的整体效能将会有显著的影响。bash是一个复杂、功能齐全的shell，因此是一个大型程序。ash shell的功能较少，所以是一个较小型的程序，不过足够满足大部分工作所需。如果bash调用自文件名*/bin/sh*，bash会显著改变自己的行为以便表现得比较像标准的shell，这会使得事情复杂化。在大部分的Linux系统上，文件*/bin/sh*是一个指向bash的符号链接，然而在Cygwin中，*/bin/sh*实际上是ash的副本。为了评估这些差异，有部分测试会进行三次，每一次使用一种shell。我们会将测试时所使用的shell标注在圆括号里。当(sh)出现时，意味着bash被链接到名为*/bin/sh*的文件。

前三个测试标注为make，用来指示当make运行时如果什么事也不做需要耗费多少成本。测试所使用的*makefile*内容如下：

注1： 想要进一步了解BogoMips，可参考*http://www.hobby.nl/~clifton/bogomips.html*的说明。

```
SHELL := /bin/bash
.PHONY: x
x:
        $(MAKE) --no-print-directory --silent --question make-bash.mk; \
        将此命令重复 99 次
```

请将 bash 字样替换成适当的 shell 名称。

为了消除非必要的求值动作影响计时测试的结果，以及避免在计时输出值中掺杂不适当的文字，此处使用了 `--no-print-directory` 和 `--silent` 命令行选项。`--question` 选项用来告诉 make 只需检查依存关系而不必执行任何命令，并且在文件已更新的时候返回值为零的结束状态。这让 make 能够尽量少做事。没有任何命令会被这个 *makefile* 执行，而且只有一个 .PHONY 工作目标存在依存关系。命令脚本会执行 make 100 次。这个名为 *make-bash.mk* 的 *makefile* 将会被上层的 *makefile* 以下面的方式运行 10 次：

```
define ten-times
  TESTS += $1
  .PHONY: $1
  $1:
        @echo $(MAKE) --no-print-directory --silent $2;\
        time $(MAKE) --no-print-directory --silent $2; \
        time $(MAKE) --no-print-directory --silent $2; \
        time $(MAKE) --no-print-directory --silent $2; \
        time $(MAKE) --no-print-directory --silent $2; \
        time $(MAKE) --no-print-directory --silent $2; \
        time $(MAKE) --no-print-directory --silent $2; \
        time $(MAKE) --no-print-directory --silent $2; \
        time $(MAKE) --no-print-directory --silent $2
endef

.PHONY: all
all:

$(eval $(call ten-times, make-bash, -f make-bash.mk))
all: $(TESTS)
```

记录运行 1000 次所花的时间，然后求平均值。

如表 10-1 所示，Cygwin 版的 make 大约每秒运行 22 次，也就是每次要花 0.044 秒的时间，而 Linux 的版本（即使在相当慢的 CPU 上）大约每秒运行 61 次，也就是每次要花 0.016 秒的时间。为了确认这些结果，我们还对原生的 Windows 版的 make 做了测试，但是速度并没有多大的提升。结论：Cygwin 版的 make 创建进程的速度比原生的 Windows 版的 make 稍慢，不过两者都比 Linux 版的慢很多。这同时意味着，在 Windows 平台上递归式 make 的使用将会比在 Linux 上慢很多。

正如你所预期的那样，运行时间不会因为shell的不同而有太大的差异。因为命令脚本中并未包含 SHELL 的特殊字符，所以 shell 根本不会被调用，make 会直接执行命令。要确认此事，你可以将shell变量设定成完全错误的值，并注意你所进行的测试是否仍然能够正确运行。这三种 shell 在效能上的不同必然是因为正常的系统差异。

接下来的基准测试用来测量变量赋值的速度。我们将会以此作为make的大多数基本操作的基础。这个 *makefile* 名为 *assign.mk*：

```
# 10000次赋值动作
z := 10
重复10000次
.PHONY: x
x: ;
```

我们将会以上层 *makefile* 里的 ten-times 函数来运行这个 *makefile*。

赋值操作的执行速度显然非常快。Cygwin 版的 make 每秒将会执行 8130 次赋值操作，而 Linux 系统高达 10989 次。我相信对大部分操作来说，Windows 的效能实际上会比测试的结果好，因为创建make进程10次的成本无法从测试结果中排除。结论：因为一般 *makefile* 不太可能执行 10000 次赋值操作，所以在一般的 *makefile* 中可以忽略变量赋值的成本。

接下来的两项基准测试用来测量 subst 函数调用的成本。首先是对10个字符长的字符串进行3次替换：

```
# 对10个字符长的字符串进行10000次subst操作
dir := ab/cd/ef/g
x := $(subst /, ,$(dir))
重复10000次
.PHONY: x
x: ;
```

此操作所花的时间大约是一般赋值操作的两倍，即在 Windows 上每秒可以进行 3891 次操作。同样地，Linux 系统的效能还是比 Windows 系统的好（别忘了，运行 Linux 系统的机器的时钟速度是运行 Windows 系统的机器的四分之一。）

然后是对 1000 个字符长的字符串进行大约 100 次替换：

```
# 10个字符长的文件名称
dir := ab/cd/ef/g
# 1000个字符长的路径
p100 := $(dir);$(dir);$(dir);$(dir);$(dir);...
p1000 := $(p100)$(p100)$(p100)$(p100)$(p100)...

# 对1000个字符长的字符串进行10000次subst操作
x := $(subst :, ,$(p1000))
```

```
重复10000次
.PHONY: x
x: ;
```

接下来的三项基准测试，用来测量以sed进行相同替换操作的成本：

```
# 以bash进行100次sed操作
SHELL := /bin/bash
.PHONY: sed-bash
sed-bash:
        echo '$(p1000)' | sed 's/;/ /g' > /dev/null
        重复100次
```

照例，我们将会以上层*makefile*里的ten-times函数来运行这个*makefile*。在Windows上，sed的执行速度大约比subst函数慢50倍，在我们的Linux系统上，大约只慢24倍。

考虑到shell的成本因素时，我们发现ash在Windows上会有提升速度的效果。使用ash，sed只比subst慢39倍！在Linux系统上，使用shell将会有更显著的影响：使用ash，sed只比subst慢5倍。我们还注意到，将bash更名为sh所产生的奇怪结果。在Cygwin上，把bash取名为/bin/bash以及取名为/bin/sh并没有什么差别。但是，在Linux系统上，把bash链接至/bin/sh将会有比较好的效能。

最后的基准测试只是调用make的shell命令来评估运行一个subshell的成本：

```
# 以bash进行100次$(shell)操作
SHELL := /bin/bash
x := $(shell :)
重复100次
.PHONY: x
x: ;
```

毫无疑问，在Windows系统上会比在Linux系统上慢，ash的效能比bash好。较明确地说，ash大约快50%。在Linux系统上，ash效能最好而bash（当取名为bash时）最慢。

然而，基准测试是一个永无止境的工作，测量结果可以提供给我们若干有用的信息。你可以创建任何必要的变量，如果这些变量有助于理清*makefile*结构的话，因为它们基本上是免费的。如果调用内置的make函数以及执行命令就可以达成你的目的，应该优先采用前者，即使你的*makefile*的结构需要一再重新执行make函数。避免递归式make或是在Windows上创建非必要的进程。在Linux系统上，如果你要创建许多进程，最好使用ash。

最后，别忘了，对大多数*makefile*来说，运行*makefile*所花的时间几乎完全取决于所运行程序的成本，而与make或*makefile*的结构无关。通常，减少所运行程序的数目，对提高*makefile*运行的速度最有帮助。

找出瓶颈与处理瓶颈

makefile 中一些非必要的延迟来自：不良的 *makefile* 结构、不良的依存分析以及 make 函数和变量使用不当。这些问题可能会被 make 函数所掩盖，比如调用命令的 shell 没有显示出它们，这让你难以找到延迟的原因。

依存分析有如双刃剑。一方面，如果进行完整的依存分析，分析动作本身就可能导致明显的延迟。没有编译器特别的支持，比如 gcc 或 jikes，依存文件的创建将会需要运行另一个程序，这差不多是两倍的编译时间（注 2）。另一方面，完整依存分析的好处是可以让 make 执行较少的编译工作。可惜，有些开发人员可能不相信有这种好事，所以编写 *makefile* 的时候所提供的依存信息并不完整。这种打折的做法几乎总是会导致更多的开发问题，而且会使得其他开发人员对打折的结果付出更多的代价。

为了将依存分析策略公式化，现在让我们来了解项目的依存关系本质。一旦完全理解依存信息之后，你可以选择在 *makefile* 中呈现多少信息（通过求值或直接表示）以及编译期间可以采取哪些快捷方式。尽管以上所提到的这些都不是很简单，但是很直观。

一旦决定好 *makefile* 的结构以及必要的依存关系，"实现出一个有效的 makefile" 这件事，通常只是在避免某些简单的陷阱。

简单变量与递归变量

最常见的与效能相关的问题之一，就是使用递归变量而不使用简单变量。例如，因为如下的代码使用的是 = 运算符而不是 :=，所以它会在每次 DATE 变量被使用时执行 date 命令：

```
DATE = $(shell date +%F)
```

+%F 选项用来指示 date 以 yyyy-mm-dd 的格式返回日期，所以大多数用户可能都不会注意到 date 会被重复执行。当然，在深夜工作的开发人员可能会吓一跳！

因为 make 不会显示出 shell 函数所执行的命令，所以很难判断当前正在执行什么命令。通过把 shell 变量重新设定为 /bin/sh -x，可让 make 显示出它所执行的每一个命令。

如下的 *makefile* 会在任何动作执行之前创建它的输出目录。输出目录的名称由 "out" 这个单词和日期组成：

注 2：实际上，编译时间与输入文件的大小呈线性增长，而且此时间总是受到磁盘 I/O 的支配。同样地，以 -M 选项找出依存关系所花的时间也是呈线性增长并且受磁盘 I/O 的支配。

```
DATE = $(shell date +%F)
OUTPUT_DIR = out-$(DATE)

make-directories := $(shell { -d $(OUTPUT_DIR) } || mkdir -p $(OUTPUT_DIR))

all: ;
```

让我们使用进入调试模式的shell来运行此*makefile*：

```
$ make SHELL='/bin/sh -x'
+ date +%F
+ date +%F
+ '{' -d out-2004-03-30 '}'
+ mkdir -p out-2004-03-30
make: all is up to date.
```

显然，date命令被执行了两次。如果你时常需要进行此类的shell追踪，这么做会比较容易：

```
ifdef DEBUG_SHELL
  SHELL = /bin/sh -x
endif
```

禁用 @

另一个隐藏命令的方式就是使用安静模式命令修饰符@。有些时候，禁用此功能将会很有用。为了方便进行此事，你可以定义一个变量，比如QUIET，来保存@符号，并且在所要执行的命令中使用此变量：

```
ifndef VERBOSE
  QUIET := @
endif
...
target:
        $(QUIET) echo Building target..
```

当需要查看安静模式修饰符所隐藏的命令时，只要在命令行上定义VERBOSE就行了：

```
$ make VERBOSE=1
echo Building target...
Building target...
```

延迟初始化

当简单变量被用来关联shell函数时，make会在它读进*makefile*的时候对所有的shell函数调用进行求值的动作。如果简单变量中有许多这样的情况，或是需要进行成本昂贵的计算，make的反应可能会变得很迟钝。要测量make的反应能力（responsiveness），可以用一个不存在的工作目标来调用make，并对它做计时的动作：

```
$ time make no-such-target
make: *** No rule to make target no-such-target.  Stop.
real    0m0.058s
user    0m0.062s
sys     0m0.015s
```

这就是 make 执行任何命令（即使是不重要或错误的命令）时的基本开销。

因为递归变量会在它们被扩展的时候，重新对它们的右边部分进行求值的动作，所以一个趋势就是将复杂的计算表示成简单变量。然而，这么做却会降低 make 对所有工作目标的反应力。我们似乎需要使用另一种变量，这种变量的右边部分只会在变量首次被求值而不是在这之前，进行求值的动作。

“快速方式：一次做好编译”一节中所提到的 find-compilation-dirs 函数，就需要进行这样的初始化设定工作：

```
# $(call find-compilation-dirs, root-directory)
find-compilation-dirs =                          \
  $(patsubst %/,%,                               \
    $(sort                                       \
      $(dir                                      \
        $(shell $(FIND) $1 -name '*.java'))))

PACKAGE_DIRS := $(call find-compilation-dirs, $(SOURCE_DIR))
```

理想情况下，我们想要在每次执行的时候只进行一次的 find 操作，但是这只是在 PACKAGE_DIRS 被实际使用的时候。这称为延迟初始化（lazy initialization）。我们可以使用 eval 来建立此类变量，例如：

```
PACKAGE_DIRS = $(redefine-package-dirs) $(PACKAGE_DIRS)

redefine-package-dirs = \
   $(eval PACKAGE_DIRS := $(call find-compilation-dirs, $(SOURCE_DIR)))
```

基本的做法就是先将 PACKAGE_DIRS 定义成递归变量。当它被扩展的时候，会对成本昂贵的函数（此处为 find-compilation-dirs）进行求值的动作，并且将它重新定义为简单变量。最后，该变量（现在是简单变量）的值会从原本的递归变量定义中被返回。

让我们再仔细地说明一遍：

1. 当make读进这些变量时，make 只会记录它们的右边部分，因为它们是递归变量。
2. 当 PACKAGE_DIRS 变量首次被使用时，make会取出它的右边部分并且扩展第一个变量 redefine-package-dirs。
3. redefine-package-dirs 的值只是一个 eval 函数调用。

4. eval的动作是将递归变量PACKAGE_DIRS重新定义为简单变量，这个简单变量的值就是find-compilation-dirs所返回的目录列表。现在PACKAGE_DIRS可以被初始化为这份目录列表。
5. redefine-package-dirs变量会被扩展成空字符串（因为eval会被扩展成空字符串）。
6. make会继续扩展PACKAGE_DIRS原本的右边部分。现在只剩下PACKAGE_DIRS变量需要扩展。make会查询该变量的值，发现它是一个简单变量，然后返回它的值。

这段代码所用到的技巧是，make对递归变量的右边部分求值时将会从左至右。举例来说，如果make决定在$(redefine-package-dirs)之前先对$(PACKAGE_DIRS)进行求值的动作，这段代码就会失败。

以上的步骤可以重构成一个lazy-init函数：

```
# $(call lazy-init,variable-name,value)
define lazy-init
  $1 = $$(redefine-$1) $$($1)
  redefine-$1 = $$(eval $1 := $2)
endef

# PACKAGE_DIRS —— 一份延后展开的目录列表
$(eval                                  \
  $(call lazy-init,PACKAGE_DIRS, \
    $$(call find-compilation-dirs,$(SOURCE_DIRS))))
```

并行式make

改进编译效能的另一种做法就是利用*makefile*解决问题时能够同时进行的本质。大部分*makefile*所进行的许多工作很容易以并行的方式来完成，像是将C源文件编译成二进制文件，或是使用二进制文件来建立程序库。此外，一个完善的*makefile*应该为并发进程（concurrent process）的自动控制提供所有必要的信息。

例10-1展示了我们以jobs选项--jobs=2（或-j 2）运行make来编译mp3_player程序的过程。从图10-1中可以看到相应的UML顺序图（sequence diagram）。--jobs=2选项用来告诉make尽可能同时更新两个工作目标。当make以并行的方式更新工作目标时，它会依照命令的执行顺序送出这些命令，所以你会在输出中看到它们交替出现。这会使得并行式make的输出较难阅读。让我们小心谨慎地查看此输出。

例10-1：当--jobs = 2时make的输出

```
$ make -f ../ch07-separate-binaries/makefile --jobs=2

1    bison -y  - defines ../ch07-separate-binaries/lib/db/playlist.y
```

```
2    flex  -t ../ch07-separate-binaries/lib/db/scanner.l > lib/db/scanner.c

3    gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-binaries/include
     -M ../ch07-separate-binaries/app/player/play_mp3.c | \
     sed 's,\(play_mp3\.o\) *:,app/player/\1 app/player/play_mp3.d: ,' >
     app/player/play_mp3.d.tmp

4    mv -f y.tab.c lib/db/playlist.c

5    mv -f y.tab.h lib/db/playlist.h

6    gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-binaries/include
     -M ../ch07-separate-binaries/lib/codec/codec.c | \
     sed 's,\(codec\.o\) *:,lib/codec/\1 lib/codec/codec.d: ,' > lib/codec/codec.d.tmp

7    mv -f app/player/play_mp3.d.tmp app/player/play_mp3.d

8    gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-binaries/include
     -M lib/db/playlist.c | \
     sed 's,\(playlist\.o\) *:,lib/db/\1 lib/db/playlist.d: ,' > lib/db/playlist.d.tmp

9    mv -f lib/codec/codec.d.tmp lib/codec/codec.d
10   gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-binaries/include
     -M ../ch07-separate-binaries/lib/ui/ui.c | \
     sed 's,\(ui\.o\) *:,lib/ui/\1 lib/ui/ui.d: ,' > lib/ui/ui.d.tmp

11   mv -f lib/db/playlist.d.tmp lib/db/playlist.d

12   gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-binaries/include
     -M lib/db/scanner.c | \
     sed 's,\(scanner\.o\) *:,lib/db/\1 lib/db/scanner.d: ,' > lib/db/scanner.d.tmp

13   mv -f lib/ui/ui.d.tmp lib/ui/ui.d

14   mv -f lib/db/scanner.d.tmp lib/db/scanner.d

15   gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-
     binaries/include  -c -o app/player/play_mp3.o ../ch07-separate-
     binaries/app/player/play_mp3.c

16   gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-
     binaries/include  -c -o lib/codec/codec.o ../ch07-separate-
     binaries/lib/codec/codec.c

17   gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-
     binaries/include  -c -o lib/db/playlist.o lib/db/playlist.c

18   gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-
     binaries/include  -c -o lib/db/scanner.o lib/db/scanner.c
  ../ch07-separate-binaries/lib/db/scanner.l: In function yylex:
  ../ch07-separate-binaries/lib/db/scanner.l:9: warning: return makes
     integer from pointer without a cast

19   gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-
     binaries/include  -c -o lib/ui/ui.o ../ch07-separate-binaries/lib/ui/ui.c

20   ar rv lib/codec/libcodec.a lib/codec/codec.o
```

```
   ar: creating lib/codec/libcodec.a
   a - lib/codec/codec.o

21    ar rv lib/db/libdb.a lib/db/playlist.o lib/db/scanner.o
   ar: creating lib/db/libdb.a
   a - lib/db/playlist.o
   a - lib/db/scanner.o

22    ar rv lib/ui/libui.a lib/ui/ui.o
   ar: creating lib/ui/libui.a
   a - lib/ui/ui.o

23    gcc   app/player/play_mp3.o lib/codec/libcodec.a lib/db/libdb.a
      lib/ui/libui.a   -o app/player/play_mp3
```

图 10-1：当 --jobs=2 时 make 的运行过程

首先，make 必须产生源文件和依存文件。有两个自动产生的源文件来自 yacc 和 lex 的输出，也就是 make 所执行的命令 1 和命令 2。命令 3 是为 *play_mp3.c* 产生的依存文件，而且显然是在 make 为 *playlist.c* 或 *scanner.c* 完成依存文件（命令 4、5、8、9、12、14）之前就开始了。因此，这个 make 同时进行了三项工作，即使我们在命令行选项上只要求它同时进行两项工作。

命令 4 和命令 5 这两个 mv 命令完成了命令 1 所开始的 *playlist.c* 源文件产生的动作。命

令 6 开始了另一个依存文件的产生动作。每个脚本总是由单一 make 所运行，但是每个工作目标和必要条件会形成一项独立的工作。因此，命令 7，也就是依存文件产生脚本的第二个命令，将会被之前执行命令 3 的 make 进程所执行。尽管命令 6 有可能被“执行命令 1、4、5（用来处理 yacc 的语法）的 make 完成工作之后立即衍生出来的 make”所执行，但是必须在命令 8 里的依存文件产生之前进行。

依存关系产生动作将会以如上的方式继续进行到命令 14 为止。所有依存文件都必须在“make 移往下一个处理阶段、重读 *makefile*”之前完成产生的动作。这形成了一个自然的同步点，让 make 能够自动遵循。

重新读进具备依存信息的 *makefile* 之后，make 将会再次以并行的方式继续进行编译的过程。这次 make 会选择在每个程序库建立之前先编译所有目标文件。这个次序是“非决定性的”，也就是说，如果 *makefile* 再次被运行，它可能会在 *playlist.c* 被编译之前先建立 *libcodec.a* 程序库，因为该程序库不需要 *codec.o* 以外的任何目标文件。因此，这个例子所展示的只是许多执行顺序中的一个可能的执行顺序。

最后，你会看到链接程序的动作。对这个 *makefile* 来说，链接阶段也是一个自然的同步点，而且总是最后进行。然而，如果你的目的不是产生一个程序，而是产生许多程序或程序库，你最后所执行的命令将会不一样。

在多处理器的系统上同时执行多项工作是理所当然的，不过在单处理器的系统上同时执行一项以上的工作也非常有用。这是因为磁盘的输入/输出有延迟的现象，而且大多数的系统都会提供大量的高速缓存空间。举例来说，如果有一个进程，比如 gcc，为了等待磁盘的输入/输出而闲置一旁，可能是另一项工作（比如 mv、yacc 或 ar）的数据目前放在内存中。这个时候最好先进行有数据可处理的工作。一般来说，让 make 在单处理器的系统上同时执行两项工作，几乎会比执行一项工作还快，而且同时执行三或四项工作比同时执行两项工作还快，这也不是什么不寻常的事。

使用 --jobs 选项的时候可以不用指定数字。如果是这样的话，make 将会衍生出跟所要更新的工作目标同样数目的工作来。这通常不是一个好主意，因为大量的工作通常会让一个处理器陷入泥潭，这可能会让执行速度变得比执行单一工作的时候还慢。

管理多重工作的另一个方法就是以系统的负载平均数作为指针。负载平均数（load average）就是以可运行的进程数目对某个时间周期（通常是 1 分钟、5 分钟或 15 分钟）求平均值。负载平均数会被表示成一个浮点数。--load-average（或 -l）选项可用来为 make 提供一个临界值，负载平均数超过此值之后便不再衍生新的工作。例如，下面这个命令：

```
% make --load-average=3.5
```

就是用来告诉make，只有在负载平均数小于或等于3.5时才可以衍生新的工作。如果负载平均数大于此值，make会一直等到平均数低于此值的时候或是完成了所有其他工作之后，才会衍生新的工作。

编写以并行的方式运行的*makefile*时，注意必要条件的设置是否恰当甚至更为重要。如先前所提到的，当--jobs的值为1时，必要条件列表的求值动作通常是由左至右，当--jobs的值大于1时，这些必要条件的求值动作将会以并行的方式进行。因此，任何以从左至右这样的默认求值顺序自动处理的依存关系，必须在你以并行的方式运行*makefile*时明确地加以指定。

并行式make还会遇到共享中间文件的问题。举例来说，如果一个目录包含*foo.y*和*bar.y*，以并行的方式运行yacc两次，可能会造成其中的一个要取得另一个的*y.tab.c*或*y.tab.h*实例，或者它们都想要将之移往自己的*.c*或*.h*文件中。任何程序，如果它用来暂存信息的临时文件使用的是固定的名称，都会让你面临类似的问题。

以shell的for循环进行递归式make也会对并行执行造成问题：

```
dir:
        for d in $(SUBDIRS);          \
        do                            \
          $(MAKE) --directory=$$d;    \
        done
```

正如“递归式make”一节所说，make并不会以并行的方式来执行这些递归调用。如果要达到并行执行的目的，你可以把这些目录声明为.PHONY，并且让它们成为工作目标：

```
.PHONY: $(SUBDIRS)
$(SUBDIRS):
        $(MAKE) --directory=$@
```

分布式make

GNU make提供了一个鲜为人知（而且只经过少许测试）的编译选项，可用来管理经网络使用多个系统的编译工作。此功能必须使用pmake所发布的自定义程序库（Customs library）。pmake大约是1989年Adam de Boor为Sprite操作系统所编写的make替代品（而且Andreas Stolcke自从那个时候就在维护这个程序了）。

自定义程序库可让make的运行以并行的方式分布到许多主机上进行。GNU make自3.77版开始纳入分布式make的自定义程序库的功能。

要启用自定义程序库的功能，你必须从源文件来重新建立make。相关细节请参考make发行包随附的README.customs文件。首先，你必须下载并建立pmake发行包（网址在README文件中提到），然后以--with-customs选项来建立make。

每个加入分布式make网络的主机都必须运行自定义服务器（customs daemon），自定义服务器是自定义程序库的核心。这些主机对文件系统的观点必须一致，比如NFS。这些自定义服务器的实例中会有一个被选定为主控服务器。主控服务器将会监控“参与主机列表”中所列出的主机，并且为每个成员分配工作。当make运行时，如果 --jobs标记的值大于1，make将会联络主控服务器，在网络中可用的各个主机上衍生出多个工作来。

自定义程序库具有广泛的功能：它可以让主机按照硬件结构分组以及按照效能分级，它可以根据属性和布尔运算符的组合，将任意属性指派给这些主机，并将工作分配给这些主机，此外，处理工作的时候，它还可以将闲置时间、可用的磁盘空间、可用的交换空间以及当前的平均负载等主机状态纳入考虑。

如果你的项目是以C、C++或Objective-C实现的，而你想把编译的工作分布到多台机器上进行，你还应该考虑distcc（*http://distcc.samba.org*）。distcc的作者为Martin Pool，目的在于加快Samba的编译速度。对以C、C++或Objective-C编写而成的项目来说，distcc是一个健全而完整的解决方案。要使用这个工具，只要把C编译器替换成distcc程序即可：

```
$ make --jobs=8 CC=distcc
```

对每一个编译工作来说，distcc会使用本地编译器对输出进行预处理，然后将已扩展的源代码送往一个可利用的远程机器进行编译。最后，远程主机会将所产生的目标文件返回主控服务器。这个做法可移除使用共享文件系统的必要性，这大大简化了安装与配置的工作。

“参与主机列表”的指定方式有好几种。最简单的方式，就是在你启动distcc之前，先在环境变量中列出参与工作的主机：

```
$ export DISTCC_HOSTS='localhost wasatch oops'
```

distcc相当好设定，它具有选项可用来操作主机列表、集成固有的编译器、管理压缩、搜索路径以及处理失败和恢复。

ccache是另一个可以改进编译效能的工具，它的作者是Samba项目负责人Andrew Tridgel。它的做法很简单，就是把先前的编译结果放入高速缓存，在执行编译工作之前，检查高速缓存中是否包含先前所产生的目标文件。这并不需要多台主机或通过网络。作者说这么做可让一般的编译工作加快5到10倍。使用ccache的最简单方法，就是为你的编译器命令前缀ccache：

```
$ make CC='ccache gcc'
```

ccache与distcc并用，对效能将会有更大的提高。此外，你可以在Cygwin的工具列表中找到这两个工具。

第十一章

makefile 实例

本书之前所提供的都是*makefile*的工业级技巧，相当适合应用在你的大多数的高端需要上。不过，仍然值得我们花些时间来查看若干*makefile*实例，看看在提交压力的状况下，人们会使用make做哪些事情。接下来，我们将会探讨若干*makefile*实例的细节。第一个实例是用来编译本书的*makefile*，第二个实例就是用来编译2.6.7版Linux内核的*makefile*。

本书的 makefile

在编译系统中，编写一本关于程序设计的书，本身就是一个有趣的练习。一本书的内容由许多文件组成，每个文件都需要各种的预处理步骤。例如，应该运行哪些程序以及对它们的输出进行收集、后处理、纳入正文的动作（所以你不必对它们进行剪贴的动作，也不会因为一时疏忽而发生错误）。写作期间，能够以不同的格式来查看正文，对你会有莫大的帮助。最后，必须把所要提交的东西封装起来。当然，以上这些动作都必须可以重复进行，而且必须很容易维护。

这听起来就像是一个make可以处理的工作！这是make极佳的一个用途。make可应用在各种让人意想不到的问题上。本书在编写时采用的是DocBook（即XML）格式。将make应用在T_EX、L_AT_EX或troff等工具上是一个标准程序。

例11-1所示为本书的完整*makefile*，大约有440行。这个*makefile*可以划分成以下几个基本工作：

- 管理范例
- 对XML进行预处理

- 产生各种输出格式
- 确认源文件
- 进行基本的维护

例 11-1：用来编译本书的 makefile

```
# 编译本书!
#
# 以下是这个文件中的主要工作目标:
#
# show_pdf        产生pdf以及启动浏览器
# pdf             产生pdf
# print           输出pdf
# show_html       产生htm以及启动浏览器
# html            产生html
# xml             产生xml
# release         制作发行版tarball
# clean           清除所产生的文件
#

BOOK_DIR      := /test/book
SOURCE_DIR    := text
OUTPUT_DIR    := out
EXAMPLES_DIR  := examples

QUIET             = @

SHELL             =  bash
AWK               := awk
CP                := cp
EGREP             := egrep
HTML_VIEWER       := cygstart
KILL              := /bin/kill
M4                := m4
MV                := mv
PDF_VIEWER        := cygstart
RM                := rm -f
MKDIR             := mkdir -p
LNDIR             := lndir
SED               := sed
SORT              := sort
TOUCH             := touch
XMLTO             := xmlto
XMLTO_FLAGS       =  -o $(OUTPUT_DIR) $(XML_VERBOSE)
process-pgm       := bin/process-includes
make-depend       := bin/make-depend

m4-macros         := text/macros.m4

# $(call process-includes, input-file, output-file)
#    移除跳格符、扩展宏以及处理引入文件
define process-includes
```

```
  expand $1 |                                                    \
  $(M4) --prefix-builtins --include=text $(m4-macros) - |        \
  $(process-pgm) > $2
endef

# $(call file-exists, file-name)
#   如果文件存在，则返回非空值
file-exists = $(wildcard $1)

# $(call maybe-mkdir, directory-name-opt)
#   如果目录不存在，则创建该目录
#   如果省略directory-name-opt，则以$@为目录名称
maybe-mkdir = $(if $(call file-exists,          \
                    $(if $1,$1,$(dir $@))),,    \
                $(MKDIR) $(if $1,$1,$(dir $@)))

# $(kill-acroread)
#   终止acrobat阅读程序
define kill-acroread
  $(QUIET) ps -W |                                  \
  $(AWK) 'BEGIN { FIELDWIDTHS = "9 47 100" }        \
          /AcroRd32/ {                              \
                      print "Killing " $$3;         \
                      system( "$(KILL) -f " $$1 )   \
                    }'
endef

# $(call source-to-output, file-name)
#   将"源文件树引用"转换成"输出文件树引用"
define source-to-output
$(subst $(SOURCE_DIR),$(OUTPUT_DIR),$1)
endef

# $(call run-script-example, script-name, output-file)
#   运行一个范例makefile。
define run-script-example
  ( cd $(dir $1);                                   \
    $(notdir $1) 2>&1 |                             \
    if $(EGREP) --silent '\$$\(MAKE\)' |mM|akefile; \
    then                                            \
      $(SED) -e 's/^++*/$$/';                       \
    else                                            \
      $(SED) -e 's/^++*/$$/'                        \
             -e '/ing directory /d'                 \
             -e 's/\[[0-9]\]//';                    \
    fi )                                            \
  > $(TMP)/out.$$$$ &                               \
  $(MV) $(TMP)/out.$$$$ $2
endef

# $(call generic-program-example,example-directory)
#   建立规则以便编译一个通用范例
define generic-program-example
  $(eval $1_dir       := $(OUTPUT_DIR)/$1)
```

```
  $(eval $1_make_out := $($1_dir)/make.out)
  $(eval $1_run_out  := $($1_dir)/run.out)
  $(eval $1_clean    := $($1_dir)/clean)
  $(eval $1_run_make := $($1_dir)/run-make)
  $(eval $1_run_run  := $($1_dir)/run-run)
  $(eval $1_sources  := $(filter-out %/CVS, $(wildcard $(EXAMPLES_DIR)/$1/*)))
  $($1_run_out): $($1_make_out) $($1_run_run)
        $$(call run-script-example, $($1_run_run), $$@)

  $($1_make_out): $($1_clean) $($1_run_make)
        $$(call run-script-example, $($1_run_make), $$@)

  $($1_clean): $($1_sources) Makefile
        $(RM) -r $($1_dir)
        $(MKDIR) $($1_dir)
        $(LNDIR) -silent ../../$(EXAMPLES_DIR)/$1 $($1_dir)
        $(TOUCH) $$@

  $($1_run_make):
        printf "#! /bin/bash -x\nmake\n" > $$@
endef


# 本书的输出格式
BOOK_XML_OUT      := $(OUTPUT_DIR)/book.xml
BOOK_HTML_OUT     := $(subst xml,html,$(BOOK_XML_OUT))
BOOK_FO_OUT       := $(subst xml,fo,$(BOOK_XML_OUT))
BOOK_PDF_OUT      := $(subst xml,pdf,$(BOOK_XML_OUT))
ALL_XML_SRC       := $(wildcard $(SOURCE_DIR)/*.xml)
ALL_XML_OUT       := $(call source-to-output,$(ALL_XML_SRC))
DEPENDENCY_FILES := $(call source-to-output,$(subst .xml,.d,$(ALL_XML_SRC)))


# xml/html/pdf——为本书产生所需要的输出格式
.PHONY: xml html pdf
xml:  $(OUTPUT_DIR)/validate
html: $(BOOK_HTML_OUT)
pdf:  $(BOOK_PDF_OUT)

# show_pdf——产生一个 pdf 文件并加以显示
.PHONY: show_pdf show_html print
show_pdf: $(BOOK_PDF_OUT)
        $(kill-acroread)
        $(PDF_VIEWER) $(BOOK_PDF_OUT)

# show_html——产生一个 html 文件并加以显示
show_html: $(BOOK_HTML_OUT)
        $(HTML_VIEWER) $(BOOK_HTML_OUT)

# print——输出本书特定的页面
print: $(BOOK_FO_OUT)
        $(kill-acroread)
        java -Dstart=15 -Dend=15 $(FOP) $< -print > /dev/null
```

```
# $(BOOK_PDF_OUT) ——产生pdf文件
$(BOOK_PDF_OUT): $(BOOK_FO_OUT) Makefile

# $(BOOK_HTML_OUT) ——产生html文件
$(BOOK_HTML_OUT): $(ALL_XML_OUT) $(OUTPUT_DIR)/validate Makefile

# $(BOOK_FO_OUT) ——产生fo中间输出文件
.INTERMEDIATE: $(BOOK_FO_OUT)
$(BOOK_FO_OUT): $(ALL_XML_OUT) $(OUTPUT_DIR)/validate Makefile

# $(BOOK_XML_OUT) ——处理所有的xml输入文件
$(BOOK_XML_OUT): Makefile

##################################################################
# 对FOP的支持
#
FOP := org.apache.fop.apps.Fop

# DEBUG_FOP——如果定义了，则可以看到fop处理器的输出
ifndef DEBUG_FOP
  FOP_FLAGS  := -q
  FOP_OUTPUT := | $(SED) -e '/not implemented/d'          \
                         -e '/relative-align/d'           \
                         -e '/xsl-footnote-separator/d'
endif

# CLASSPATH——为fop定义适当的CLASSPATH
export CLASSPATH
CLASSPATH = $(patsubst %;,%,                                \
              $(subst ; ,;,                                 \
                $(addprefix c:/usr/xslt-process-2.2/java/,  \
                  $(addsuffix .jar;,                        \
                    xalan                                   \
                    xercesImpl                              \
                    batik                                   \
                    fop                                     \
                    jimi-1.0                                \
                    avalon-framework-cvs-20020315))))

# %.pdf——从fo输入产生pdf输出的模式规则
%.pdf: %.fo
        $(kill-acroread)
        java -Xmx128M $(FOP) $(FOP_FLAGS) $< $@ $(FOP_OUTPUT)

# %.fo—— 从xml输入产生fo输出的模式规则
PAPER_SIZE := letter
%.fo: %.xml
        XSLT_FLAGS="--stringparam paper.type $(PAPER_SIZE)" \
        $(XMLTO) $(XMLTO_FLAGS) fo $<

# %.html——从xml输入产生html输出的模式规则
%.html: %.xml
        $(XMLTO) $(XMLTO_FLAGS) html-nochunks $<

# fop_help——显示fop处理器的说明文字
.PHONY: fop_help
```

```
fop_help:
        -java org.apache.fop.apps.Fop -help
        -java org.apache.fop.apps.Fop -print help

#####################################################################
# release——为本书产生一个可供发行的 tarball
#
RELEASE_TAR   := mpwm-$(shell date +%F).tar.gz
RELEASE_FILES := README Makefile *.pdf bin examples out text
.PHONY: release
release: $(BOOK_PDF_OUT)
        ln -sf $(BOOK_PDF_OUT) .
        tar --create                          \
            --gzip                            \
            --file=$(RELEASE_TAR)             \
            --exclude=CVS                     \
            --exclude=semantic.cache          \
            --exclude=*~                      \
            $(RELEASE_FILES)
        ls -l $(RELEASE_TAR)

#####################################################################
# 为各章产生范例的规则
#

# 下面是每个范例所摆放的目录
EXAMPLES :=                                   \
                ch01-bogus-tab                \
                ch01-cw1                      \
                ch01-hello                    \
                ch01-cw2                      \
                ch01-cw2a                     \
                ch02-cw3                      \
                ch02-cw4                      \
                ch02-cw4a                     \
                ch02-cw5                      \
                ch02-cw5a                     \
                ch02-cw5b                     \
                ch02-cw6                      \
                ch02-make-clean               \
                ch03-assert-not-null          \
                ch03-debug-trace              \
                ch03-debug-trace-1            \
                ch03-debug-trace-2            \
                ch03-filter-failure           \
                ch03-find-program-1           \
                ch03-find-program-2           \
                ch03-findstring-1             \
                ch03-grep                     \
                ch03-include                  \
                ch03-invalid-variable         \
                ch03-kill-acroread            \
                ch03-kill-program             \
                ch03-letters                  \
```

```
                ch03-program-variables-1        \
                ch03-program-variables-2        \
                ch03-program-variables-3        \
                ch03-program-variables-5        \
                ch03-scoping-issue              \
                ch03-shell                      \
                ch03-trailing-space             \
                ch04-extent                     \
                ch04-for-loop-1                 \
                ch04-for-loop-2                 \
                ch04-for-loop-3                 \
                ch06-simple                     \
                appb-defstruct                  \
                appb-arithmetic

# 实际上，我想使用如下的 foreach 循环，但是因为make 3.80 有一个缺陷，
# 所以会产生一个无法挽回的错误
#$(foreach e,$(EXAMPLES),$(eval $(call generic-program-example,$e)))

# 我只好自己手动扩展这个 foreach 循环
$(eval $(call generic-program-example,ch01-bogus-tab))
$(eval $(call generic-program-example,ch01-cw1))
$(eval $(call generic-program-example,ch01-hello))
$(eval $(call generic-program-example,ch01-cw2))
$(eval $(call generic-program-example,ch01-cw2a))
$(eval $(call generic-program-example,ch02-cw3))
$(eval $(call generic-program-example,ch02-cw4))
$(eval $(call generic-program-example,ch02-cw4a))
$(eval $(call generic-program-example,ch02-cw5))
$(eval $(call generic-program-example,ch02-cw5a))
$(eval $(call generic-program-example,ch02-cw5b))
$(eval $(call generic-program-example,ch02-cw6))
$(eval $(call generic-program-example,ch02-make-clean))
$(eval $(call generic-program-example,ch03-assert-not-null))
$(eval $(call generic-program-example,ch03-debug-trace))
$(eval $(call generic-program-example,ch03-debug-trace-1))
$(eval $(call generic-program-example,ch03-debug-trace-2))
$(eval $(call generic-program-example,ch03-filter-failure))
$(eval $(call generic-program-example,ch03-find-program-1))
$(eval $(call generic-program-example,ch03-find-program-2))
$(eval $(call generic-program-example,ch03-findstring-1))
$(eval $(call generic-program-example,ch03-grep))
$(eval $(call generic-program-example,ch03-include))
$(eval $(call generic-program-example,ch03-invalid-variable))
$(eval $(call generic-program-example,ch03-kill-acroread))
$(eval $(call generic-program-example,ch03-kill-program))
$(eval $(call generic-program-example,ch03-letters))
$(eval $(call generic-program-example,ch03-program-variables-1))
$(eval $(call generic-program-example,ch03-program-variables-2))
$(eval $(call generic-program-example,ch03-program-variables-3))
$(eval $(call generic-program-example,ch03-program-variables-5))
$(eval $(call generic-program-example,ch03-scoping-issue))
$(eval $(call generic-program-example,ch03-shell))
$(eval $(call generic-program-example,ch03-trailing-space))
```

```
$(eval $(call generic-program-example,ch04-extent))
$(eval $(call generic-program-example,ch04-for-loop-1))
$(eval $(call generic-program-example,ch04-for-loop-2))
$(eval $(call generic-program-example,ch04-for-loop-3))
$(eval $(call generic-program-example,ch06-simple))
$(eval $(call generic-program-example,ch10-echo-bash))
$(eval $(call generic-program-example,appb-defstruct))
$(eval $(call generic-program-example,appb-arithmetic))


######################################################################
# 确认
#
# 检查 a) 未扩展的m4宏, b) 跳格符, c) FIXME注释, d)
# RM: 对 Andy 的响应, e) 重复的 m4 宏
#
validation_checks := $(OUTPUT_DIR)/chk_macros_tabs                  \
                     $(OUTPUT_DIR)/chk_fixme                        \
                     $(OUTPUT_DIR)/chk_duplicate_macros             \
                     $(OUTPUT_DIR)/chk_orphaned_examples

.PHONY: validate-only
validate-only: $(OUTPUT_DIR)/validate
$(OUTPUT_DIR)/validate: $(validation_checks)
        $(TOUCH) $@

$(OUTPUT_DIR)/chk_macros_tabs: $(ALL_XML_OUT)
        # 查找宏和跳格符……
        $(QUIET)! $(EGREP) --ignore-case                            \
                           --line-number                            \
                           --regexp='\b(m4_|mp_)'                   \
                           --regexp='\011'                          \
                           $^
        $(TOUCH) $@

$(OUTPUT_DIR)/chk_fixme: $(ALL_XML_OUT)
        # 查找RM:和FIXME……
        $(QUIET)$(AWK)                                              \
                '/FIXME/ { printf "%s:%s: %s\n", FILENAME, NR, $$0 }  \
                 /^ *RM:/  {                                        \
                           if ( $$0 !~ /RM: Done/ )                 \
                           printf "%s:%s: %s\n", FILENAME, NR, $$0  \
                         }' $(subst $(OUTPUT_DIR)/,$(SOURCE_DIR)/,$^)
        $(TOUCH) $@

$(OUTPUT_DIR)/chk_duplicate_macros: $(SOURCE_DIR)/macros.m4
        # 查找重复的宏……
        $(QUIET)! $(EGREP) --only-matching                          \
              "\`[^']+'," $< |                                      \
        $(SORT) |                                                   \
        uniq -c |                                                   \
        $(AWK) '$$1 > 1 { printf "$<:0: %s\n", $$0 }' | \
        $(EGREP) "^"
        $(TOUCH) $@
```

```
ALL_EXAMPLES := $(TMP)/all_examples
$(OUTPUT_DIR)/chk_orphaned_examples: $(ALL_EXAMPLES) $(DEPENDENCY_FILES)
        $(QUIET)$(AWK) -F/ '/(EXAMPLES|OUTPUT)_DIR/ { print $$3 }'        \
                $(filter %.d,$^) |                                         \
        $(SORT) -u |                                                       \
        comm -13 - $(filter-out %.d,$^)
        $(TOUCH) $@

.INTERMEDIATE: $(ALL_EXAMPLES)
$(ALL_EXAMPLES):
        # 查找未使用的范例……
        $(QUIET) ls -p $(EXAMPLES_DIR) |          \
        $(AWK) '/CVS/ { next }                    \
                /\// { print substr($$0, 1, length - 1) }' > $@

#######################################################################
# 清理
#
clean:
        $(kill-acroread)
        $(RM) -r $(OUTPUT_DIR)
        $(RM) $(SOURCE_DIR)/*~ $(SOURCE_DIR)/*.log semantic.cache
        $(RM) book.pdf

#######################################################################
# 依存关系的管理
#
# 如果我们正在进行清理的工作，则不会读取或重建引入文件
#
ifneq "$(MAKECMDGOALS)" "clean"
  -include $(DEPENDENCY_FILES)
endif

vpath %.xml $(SOURCE_DIR)
vpath %.tif $(SOURCE_DIR)
vpath %.eps $(SOURCE_DIR)

$(OUTPUT_DIR)/%.xml: %.xml $(process-pgm) $(m4-macros)
        $(call process-includes, $<, $@)

$(OUTPUT_DIR)/%.tif: %.tif
        $(CP) $< $@

$(OUTPUT_DIR)/%.eps: %.eps
        $(CP) $< $@

$(OUTPUT_DIR)/%.d: %.xml $(make-depend)
        $(make-depend) $< > $@

#######################################################################
# 创建输出目录
#
# 在有需要的时候创建输出目录。
#
DOCBOOK_IMAGES := $(OUTPUT_DIR)/release/images
DRAFT_PNG      := /usr/share/docbook-xsl/images/draft.png
```

```
ifneq "$(MAKECMDGOALS)" "clean"
  _CREATE_OUTPUT_DIR :=                                                  \
    $(shell                                                              \
      $(MKDIR) $(DOCBOOK_IMAGES) &                                       \
      $(CP) $(DRAFT_PNG) $(DOCBOOK_IMAGES);                              \
      if ! [[ $(foreach d,                                               \
                $(notdir                                                 \
                  $(wildcard $(EXAMPLES_DIR)/ch*)),                      \
                -e $(OUTPUT_DIR)/$d &) -e . ]];                          \
      then                                                               \
        echo Linking examples... > /dev/stderr;                          \
        $(LNDIR) $(BOOK_DIR)/$(EXAMPLES_DIR) $(BOOK_DIR)/$(OUTPUT_DIR); \
      fi)
endif
```

这个 *makefile* 是在 Cygwin 下编写而成的，而且不打算移植到 Unix 上。不过，我相信它与 Unix 之间有少数不兼容的地方是无法通过重新定义变量或加入额外变量来解决的。

全局变量区段首先会定义根目录的位置，以及正文、范例、输出目录的相对位置。*makefile* 所使用的每个程序都会被定义成一个变量。

管理范例

第一项工作就是管理范例，这是其中最复杂的工作。每个范例都存放在它自己位于 *book/examples/ch<n>-<title>* 的目录中，都包含了一个 *makefile* 以及所有的支持文件与子目录。为了处理范例，我们首先会创建一个目录符号链接，让它指向输出文件树并且在该处工作，这样运行 *makefile* 所产生的结果就不会留在源文件树中了。此外，大部分的范例都需要把当前工作目录设成 *makefile* 所在的目录，以便产生预期的输出。为源文件创建符号链接之后，我们会运行一个 shell 脚本 `run-make`，以适当的参数来调用 *makefile*。如果源文件树中没有 shell 脚本，我们可以产生默认的版本。`run-make` 脚本的输出会被存入 *make.out*。有些例子会产生一个可执行文件，而且它还必须被运行。要达到此目的，你可以运行 `run-run` 脚本并将它的输出存入 *run.out* 文件。

文件树的符号链接的创建动作是由如下的代码（位于 *makefile* 尾端）进行的：

```
ifneq "$(MAKECMDGOALS)" "clean"
  _CREATE_OUTPUT_DIR :=                                                  \
    $(shell                                                              \
      ...                                                                \
      if ! [[ $(foreach d,                                               \
                $(notdir                                                 \
                  $(wildcard $(EXAMPLES_DIR)/ch*)),                      \
                -e $(OUTPUT_DIR)/$d &&) -e . ]];                         \
      then                                                               \
        echo Linking examples... > /dev/stderr;                          \
        $(LNDIR) $(BOOK_DIR)/$(EXAMPLES_DIR) $(BOOK_DIR)/$(OUTPUT_DIR); \
```

```
        fi)
    endif
```

此代码由一个简单变量的赋值表达式组成，而且此赋值表达式被包裹在ifneq语句中。这个条件语句用来避免make在make clean运行期间建立输出目录结构。_CREATE_OUTPUT_DIR是一个实际存在的变量，但它却是一个虚拟变量，因为它的值永远不会被用到。然而，位于赋值运算符右边部分的shell函数会在make读进*makefile*时立即被执行。shell函数会检查输出树中是否存在每个范例目录。如果有任何缺失的状况，就会调用lndir命令来更新文件树的符号链接。

if语句所进行的测试值得我们做更深入的探讨。这个测试是由对每个范例目录所进行的-e测试所组成的(即这个文件存在吗？)，实际的代码有点像是这样：使用wildcard来确定所有的范例目录，并且用notdir删除路径中目录的部分，然后为每个范例目录产生-e $(OUTPUT_DIR)/*dir* &&这样的文字。现在，将以上所有片段衔接在一起，嵌入bash的[[...]]测试中。最后，将结果取反。一个额外的测试-e .会被纳入，好让foreach循环能够轻易为每个子句加上&&。

这足以确保当新的目录被发现时，总是会被加入编译系统。

下一个步骤就是建立用来更新*make.out*和*run.out*这两个输出文件的规则，方法就是建立一个用户自定义函数来为每个范例的*.out*文件进行此事：

```
# $(call generic-program-example,example-directory)
#   建立规则以便编译一个通用范例
define generic-program-example
  $(eval $1_dir       := $(OUTPUT_DIR)/$1)
  $(eval $1_make_out  := $($1_dir)/make.out)
  $(eval $1_run_out   := $($1_dir)/run.out)
  $(eval $1_clean     := $($1_dir)/clean)
  $(eval $1_run_make  := $($1_dir)/run-make)
  $(eval $1_run_run   := $($1_dir)/run-run)
  $(eval $1_sources   := $(filter-out %/CVS, $(wildcard $(EXAMPLES_DIR)/$1/*)))

  $($1_run_out): $($1_make_out) $($1_run_run)
        $$(call run-script-example, $($1_run_run), $$@)

  $($1_make_out): $($1_clean) $($1_run_make)
        $$(call run-script-example, $($1_run_make), $$@)

  $($1_clean): $($1_sources) Makefile
        $(RM) -r $($1_dir)
        $(MKDIR) $($1_dir)
        $(LNDIR) -silent ../../$(EXAMPLES_DIR)/$1 $($1_dir)
        $(TOUCH) $$@

  $($1_run_make):
        printf "#! /bin/bash -x\nmake\n" > $$@
endef
```

我们必须为每个范例目录调用一次此函数：

```
$(eval $(call generic-program-example,ch01-bogus-tab))
$(eval $(call generic-program-example,ch01-cw1))
$(eval $(call generic-program-example,ch01-hello))
$(eval $(call generic-program-example,ch01-cw2))
```

为了方便性与可读性，变量的定义被放在此函数的开头。而且因为将赋值的动作放在eval里面，所以它们的值可以立即被宏所使用，而不必进行额外的加引号动作。

此函数的重心是两个工作目标：$($1_run_out)和$($1_make_out)，它们分别被用来为每个范例更新*run.out*和*make.out*工作目标。这两个变量的名称由范例目录名称以及后缀*_run_out*或*_make_out*组成。

第一个规则指出*run.out*依存于*make.out*和*run-run*脚本。也就是说，如果make已经被运行或*run-run*控制脚本已经被更新，就返回范例程序。工作目标的更新是通过run-script-example函数来完成：

```
# $(call run-script-example, script-name, output-file)
#   运行一个范例makefile
define run-script-example
  ( cd $(dir $1);                                          \
    $(notdir $1) 2>&1 |                                    \
    if $(EGREP) --silent '\$$\(MAKE\)' [mM]akefile;        \
    then                                                   \
      $(SED) -e 's/^++*/$$/';                              \
    else                                                   \
      $(SED) -e 's/^++*/$$/'                               \
             -e '/ing directory /d'                        \
             -e 's/\([0-9]\)//';                           \
    fi )                                                   \
  > $(TMP)/out.$$$$ &&                                     \
  $(MV) $(TMP)/out.$$$$ $2
endef
```

此函数需要两个参数：脚本的路径以及输出文件的名称。它会切换到脚本所在的目录并且运行脚本，通过一个过滤器将标准输出和错误输出使用管道转至清理它们的地方（注1）。

注1　清理的过程比较复杂。*run-run*和*run-make*脚本通常会使用bash -x，好让实际的make命令行被输出。在输出中，-x选项会把++放每个命令之前，将清理脚本转换成一个简单的$以代表shell提示符。然而，命令并非输出中所出现的唯一信息。因为make正在运行范例，而且最后会启动另一个make，所以简单的*makefile*可能会包含额外的、不需要的输出，比如Entering directory...和Leaving directory...等信息，以及在信息中显示make的层次编号。对于一个不会递归调用make的简单*makefile*，我们会去除make的输出中不恰当的信息，就好像它是从最顶层的shell来运行。

*make.out*工作目标也一样，不过会执行额外的编译动作。如果新的文件被加入一个范例中，我们会想要检测此状态并且重编译范例。_CREATE_OUTPUT_DIR的代码只会在一个新的目录中被发现时重建符号链接，而不是在新的文件被加入的时候。为了检测此状态，我们会在每个范例目录中放入一个时间戳文件，用来指示 lndir 上一次是何时被执行的。$(S1_clean)工作目标将会更新此时间戳文件，而且依存于范例目录中的实际源文件（并非输出目录中的符号链接）。如果make的依存分析发现范例目录中有一个文件比*clean*时间戳文件还新，命令脚本将会删除符号链接版的输出目录，然后重建它并放入一个新的*clean*时间戳文件。当*makefile*本身被修改时也会执行此项动作。

最后，*run-make*这个 shell 脚本会被调用来运行*makefile*，这是一个典型的两行脚本。

```
#! /bin/bash -x
make
```

照本宣科地产生这些脚本马上就会让人感到厌烦，所以$($S1_run_make)工作目标会被加入到$($S1_make_out)的必要条件列表中以便创建它。如果没有这个必要条件，*makefile*会在输出文件树中产生它。

为每个范例目录执行generic-program-example函数时，此函数会为运行范例建立所有规则，并为XML 文件里的内容准备输出。这些规则会被*makefile*中所包含的依存关系的求值动作所触发。例如，下面是 Chapter 1 的依存文件：

```
out/ch01.xml: $(EXAMPLES_DIR)/ch01-hello/Makefile
out/ch01.xml: $(OUTPUT_DIR)/ch01-hello/make.out
out/ch01.xml: $(EXAMPLES_DIR)/ch01-cw1/count_words.c
out/ch01.xml: $(EXAMPLES_DIR)/ch01-cw1/lexer.l
out/ch01.xml: $(EXAMPLES_DIR)/ch01-cw1/Makefile
out/ch01.xml: $(OUTPUT_DIR)/ch01-cw1/make.out
out/ch01.xml: $(EXAMPLES_DIR)/ch01-cw2/lexer.l
out/ch01.xml: $(OUTPUT_DIR)/ch01-cw2/make.out
out/ch01.xml: $(OUTPUT_DIR)/ch01-cw2/run.out
out/ch01.xml: $(OUTPUT_DIR)/ch01-bogus-tab/make.out
```

这些依存关系产生自一个简单的awk脚本，姑且称它为make-depend：

```
#! /bin/awk -f

function generate_dependency( prereq )
{
  filename = FILENAME
  sub( /text/, "out", filename )
  print filename ": " prereq
}

/^ *include-program/ {
  generate_dependency( "$(EXAMPLES_DIR)/" $2 )
}
```

```
/^ *mp_program\(/ {
  match( $0, /\((.*)\)/, names )
  generate_dependency( "$(EXAMPLES_DIR)/" names[1] )
}

/^ *include-output/ {
  generate_dependency( "$(OUTPUT_DIR)/" $2 )
}

/^ *mp_output\(/ {
  match( $0, /\((.*)\)/, names )
  generate_dependency( "$(OUTPUT_DIR)/" names[1] )
}

/graphic fileref/ {
  match( $0, /"(.*)"/, out_file )
  generate_dependency( out_file[1] )
}
```

此脚本会搜索如下的模式：

```
mp_program(ch01-hello/Makefile)
mp_output(ch01-hello/make.out)
```

（mp_program宏使用的是程序列表的形式，而mp_output宏使用的是程序输出的形式。）此脚本会从源文件的名称与文件名参数产生出依存关系来。

最后，依存文件的产生按惯例是受到make的include语句的触发：

```
# $(call source-to-output, file-name)
#   将"源文件树引用"转换成"输出文件树引用"
define source-to-output
$(subst $(SOURCE_DIR),$(OUTPUT_DIR),$1)
endef
 . . .
ALL_XML_SRC       := $(wildcard $(SOURCE_DIR)/*.xml)
DEPENDENCY_FILES := $(call source-to-output,$(subst
.xml,.d,$(ALL_XML_SRC)))
 . . .
ifneq "$(MAKECMDGOALS)" "clean"
  -include $(DEPENDENCY_FILES)
endif

vpath %.xml $(SOURCE_DIR)
 . . .
$(OUTPUT_DIR)/%.d: %.xml $(make-depend)
        $(make-depend) $< > $@
```

以上就是用来处理范例的完整代码。大多数复杂的主文件名，来自*makefile*依需要所实际引入的源文件以及make和范例程序的实际输出。我想make应该会说"要么你去做，要么给我闭嘴"（put up or shut up）。如果我相信make是如此的好用，那它应该能够处理这个复杂的工作，而且，天啊，它的确可以。

XML的预处理

即使让子孙后代认为我是一个俗不可耐的人，我也必须承认，我非常不喜欢XML。我觉得它既笨拙且冗长。所以，当我发现自己的原稿必须以DocBook编写时，我找来了一些能够协助我减轻痛苦的、较传统的工具。m4宏处理器与awk这两个工具对我的帮助非常大。

m4非常适合用来处理DocBook与XML所存在的两个问题：避免XML的冗长语法，以及管理交互引用中的XML标识符。例如，在DocBook中要强调一个单词，你必须这么做：

```
<emphasis>not</emphasis>
```

而有了m4，我只要编写一个简单的宏，就可以这么做：

```
mp_em(not)
```

啊，感觉好多了。此外，我发现许多符号编排风格适合使用此工具，比如mp_variable和mp_target。这让我能够选择一个不重要的格式，比如literal，并且稍后在产品部门有意见的时候可以随时加以变更，而不用进行全局的搜索和替换。

我相信XML迷们可能会寄一堆电子邮件来告诉我，如何使用实体或之类的东西来达到此目的，但是别忘了Unix就是希望你用手边的工具把事情做好，而且正如Larry Wall的口头禅“解决事情的方法不只一种”说得那样。此外，我害怕学太多有关XML的知识会让我的脑袋坏掉？

m4的第二项工作是让XML标识符能够使用在交互引用中。每一章、节、范例以及表格都会标上一个如下的标识符：

```
<sect1 id="MPWM-CH-7-SECT-1">
```

要引用某一章就必须使用这个标识符。从程序设计的观点来看这显然是一个问题。标识符是一个复杂的常数，里头充满了各种编码。此外，标识符符号本身不代表任何意义。我并不知道第七章第一节的内容在讲什么。使用m4，我可以避免重复的复杂文字，而且可以提供较具意义的名字：

```
<sect1 id="mp_se_makedepend">
```

最重要的是，如果章、节有所调整，只要在一个文件中变更几个常数就可以进行更新的动作。当一章里各节的顺序需要重新安排时，此项优点最为显著。如果不使用符号引用的话，像这样的操作，可能需要跨越所有文件进行多次全局的搜索和替换操作。

下面是一些 m4 宏的范例（注 2）：

```
m4_define(`mp_tag',      `<$1>`$2'</$1>')
m4_define(`mp_lit',      `mp_tag(literal, `$1')')

m4_define(`mp_cmd',      `mp_tag(command, `$1')')
m4_define(`mp_target',   `mp_lit($1)')

m4_define(`mp_all',      `mp_target(all)')
m4_define(`mp_bash',     `mp_cmd(bash)')

m4_define(`mp_ch_examples',      'MPWM-CH-11')
m4_define(`mp_se_book',          'MPWM-CH-11,1')
m4_define(`mp_ex_book_makefile', 'MPWM-CH-11-EX-1')
```

其他的预处理工作，就是为了实现出一个引入的功能以加入之前所讨论到的范例文本。范例文本中的跳格需要转换成空格（因为 O'Reilly 的 DocBook 转换程序无法处理跳格，但是 *makefile* 中有许多跳格！），所以必须把它们包裹在 [CDATA[...]] 里以便保护特殊的字符，最后还必须整理范例文本开头和结尾处的额外换行符号。我使用另一个名为 process-includes 的小型 awk 程序来完成此事：

```
#! /usr/bin/awk -f
function expand_cdata( dir )
{
  start_place = match( $1, "include=" )
  if ( start_place > 0 )
  {
    prefix = substr( $1, 1, start_place - 1 )
  }
  else
  {
    print "Bogus include '" $0 "'" > "/dev/stderr"
  }

  end_place = match( $2, "(</(programlisting|screen)>.*)$", tag )
  if ( end_place > 0 )
  {
    file = dir substr( $2, 1, end_place - 1 )
  }
  else
  {
    print "Bogus include '" $0 "'" > "/dev/stderr"
  }

  command = "expand " file

  printf "%s>&33;&91;CDATA[", prefix
  tail = 0
```

注 2　mp 前缀代表 Managing Projects（本书的标题），macro processor（宏处理器）或 make pretty。任君选择。

```
  previous_line = ""
  while ( (command | getline line) > 0 )
  {
    if ( tail )
      print previous_line;

    tail = 1
    previous_line = line
  }

  printf "%s&93;&93;&62;%s\n", previous_line, tag[1]
  close( command )
}

/include-program/ {
  expand_cdata( "examples/" )
  next;
}

/include-output/ {
  expand_cdata( "out/" )
  next;
}

/<(programlisting|screen)> *$/ {
  # Find the current indentation.
  offset = match( $0, "<(programlisting|screen)>" )

  # Strip newline from tag.
  printf $0

  # Read the program...
  tail = 0
  previous_line = ""
  while ( (getline line) > 0 )
  {
    if ( line ~ "</(programlisting|screen)>" )
    {
      gsub( /^ */, "", line )
      break
    }

    if ( tail )
      print previous_line

    tail = 1
    previous_line = substr( line, offset + 1 )
  }

  printf "%s%s\n", previous_line, line

  next
}
```

```
{
  print
}
```

在这个 *makefile* 中，我们将 XML 文件从源文件树复制到输出文件树、转换跳格符、扩展宏以及处理引入文件：

```
process-pgm := bin/process-includes
m4-macros   := text/macros.m4

# $(call process-includes, input-file, output-file)
#   移除跳格符、扩展宏以及处理引入文件
define process-includes
  expand $1 |                                          \
  $(M4) --prefix-builtins --include=text $(m4-macros) - | \
  $(process-pgm) > $2
endef

vpath %.xml $(SOURCE_DIR)

$(OUTPUT_DIR)/%.xml: %.xml $(process-pgm) $(m4-macros)
        $(call process-includes, $<, $@)
```

此处的模式规则指出了将源文件树的XML文件复制到输出文件树的方法。它还指出了，如果宏或引入文件处理器有所变动，应该重新产生输出文件树中的所有 XML 文件。

产生输出

到目前为止，我们尚未说明实际被编排过的任何文本，或者任何可以输出或显示的结果。显然，一个非常重要的功能就是让 *makefile* 进行本书的编排工作。有两种格式是我感兴趣的：HTML 和 PDF。

我首先会说明如何进行编排成HTML格式的工作。有一个好用的小程序xsltproc以及它的辅助脚本 xmlto 可用来完成此工作。使用这两个工具，处理的过程相当简单：

```
# 本书的输出格式
BOOK_XML_OUT      := $(OUTPUT_DIR)/book.xml
BOOK_HTML_OUT     := $(subst xml,html,$(BOOK_XML_OUT))
ALL_XML_SRC       := $(wildcard $(SOURCE_DIR)/*.xml)
ALL_XML_OUT       := $(call source-to-output,$(ALL_XML_SRC))

# html——为本书产生所需要的输出格式
.PHONY: html
html: $(BOOK_HTML_OUT)

# show_html——产生一个 html 文件并加以显示
.PHONY: show_html
show_html: $(BOOK_HTML_OUT)
        $(HTML_VIEWER) $(BOOK_HTML_OUT)
```

```
# $(BOOK_HTML_OUT)——产生 html 文件。
$(BOOK_HTML_OUT): $(ALL_XML_OUT) $(OUTPUT_DIR)/validate Makefile

# %.html——从 xml 输入产生 html 输出的模式规则。
%.html: %.xml
        $(XMLTO) $(XMLTO_FLAGS) html-nochunks $<
```

此处的模式规则主要是用来将XML文件转换成HTML文件。本书的结构就放在一个名为*book.xml*的顶层文件中，用来引入每一章。BOOK_XML_OUT变量代表这个顶层文件，BOOK_HTML_OUT则是这个顶层文件的HTML版本，我们会以它为工作目标，以它所包含的XML文件为必要条件。为了方便起见，我们会设定两个假想工作目标html和show_html，分别用来创建HTML文件以及通过当前的浏览器来显示它。

尽管说起来容易，但是PDF文件的产生却相当复杂。*xsltproc*程序可用来直接产生PDF文件，但是我无法单独使用它。所有的工作是在具有Cygwin的Windows上进行的，而且Cygwin版的xsltproc需要使用POSIX风格的路径。而我所使用的自定义版（custom version）的DocBook以及原稿本身，所包含的是Windows风格的路径。我相信这个差异将会让xsltproc发生我无法处理的问题。所以我选择使用xsltproc来产生XML编排对象（formatting object），然后使用Java程序FOP（*http://xml.apache.org/fop*）来产生PDF文件。

因此，用来产生PDF文件的代码会有点长：

```
# 本书的输出格式
BOOK_XML_OUT      := $(OUTPUT_DIR)/book.xml
BOOK_FO_OUT       := $(subst xml,fo,$(BOOK_XML_OUT))
BOOK_PDF_OUT      := $(subst xml,pdf,$(BOOK_XML_OUT))
ALL_XML_SRC       := $(wildcard $(SOURCE_DIR)/*.xml)
ALL_XML_OUT       := $(call source-to-output,$(ALL_XML_SRC))

# pdf——为本书产生所需要的输出格式
.PHONY: pdf
pdf:  $(BOOK_PDF_OUT)

# show_pdf——产生pdf文件并加以显示
.PHONY: show_pdf
show_pdf: $(BOOK_PDF_OUT)
        $(kill-acroread)
        $(PDF_VIEWER) $(BOOK_PDF_OUT)

# $(BOOK_PDF_OUT) ——产生pdf文件
$(BOOK_PDF_OUT): $(BOOK_FO_OUT) Makefile

# $(BOOK_FO_OUT) ——产生fo中间输出文件
.INTERMEDIATE: $(BOOK_FO_OUT)
$(BOOK_FO_OUT): $(ALL_XML_OUT) $(OUTPUT_DIR)/validate Makefile

# 对FOP的支持
FOP := org.apache.fop.apps.Fop
```

```
# DEBUG_FOP——如果定义了，则可以看到 fop 处理器的输出
ifndef DEBUG_FOP
  FOP_FLAGS  := -q
  FOP_OUTPUT := | $(SED) -e '/not implemented/d'          \
                         -e '/relative-align/d'           \
                         -e '/xsl-footnote-separator/d'
endif

# CLASSPATH——为 fop 定义适当的 CLASSPATH
export CLASSPATH
CLASSPATH = $(patsubst %;,%,                                \
              $(subst ; ,:,                                 \
                $(addprefix c:/usr/xslt-process-2.2/java/,  \
                  $(addsuffix .jar;,                        \
                    xalan                                   \
                    xercesImpl                              \
                    batik                                   \
                    fop                                     \
                    jimi-1.0                                \
                    avalon-framework-cvs-20020315))))

# %.pdf——从 fo 输入产生 pdf 输出的模式规则
%.pdf: %.fo
        $(kill-acroread)
        java -Xmx128M $(FOP) $(FOP_FLAGS) $< $@ $(FOP_OUTPUT)

# %.fo——从 xml 输入产生 fo 输出的模式规则
PAPER_SIZE := letter
%.fo: %.xml
        XSLT_FLAGS="--stringparam paper.type $(PAPER_SIZE)" \
        $(XMLTO) $(XMLTO_FLAGS) fo $<

# fop_help——显示 fop 处理器的说明文字
.PHONY: fop_help
fop_help:
        -java org.apache.fop.apps.Fop -help
         java org.apache.fop.apps.Fop -print help
```

你在此处所看到的两个模式规则，反映出我所使用的两个阶段的处理过程：用来把*.xml*转换成*.fo*的规则将会调用 xmlto；用来把*.fo*转换成*.pdf*的规则首先会终止任何运行中的 Acrobat 阅读程序（因为这个程序会锁住 PDF 文件，避免 PDF 写入该文件），然后运行 FOP。FOP 是一个非常饶舌的程序，它会显示上百行无意义的警告信息，因此我加入了一个简单的 sed 过滤器 FOP_OUTPUT，以便移除这些烦人的信息。然而，有些时候，这些警告信息中可能会包含有意义的数据，所以我加入了一个调试功能 DEBUG_FOP 以便停用此过滤器。最后，如同 HTML 的版本，我加入了两个方便的工作目标 pdf 和 show_pdf，让整个事情能够开始进行。

确认源文件

"跳格符、宏处理器、引入文件以及来自编辑器的注释"是DocBook所厌恶的东西，要确定源文件中的文本是否正确和完整并不容易。为了协助此过程，我实现了四项确认工作目标来检查各种形式和正确性。

```
validation_checks := $(OUTPUT_DIR)/chk_macros_tabs          \
                     $(OUTPUT_DIR)/chk_fixme                 \
                     $(OUTPUT_DIR)/chk_duplicate_macros      \
                     $(OUTPUT_DIR)/chk_orphaned_examples

.PHONY: validate-only
validate-only: $(OUTPUT_DIR)/validate
$(OUTPUT_DIR)/validate: $(validation_checks)
        $(TOUCH) $@
```

每项工作目标会产生一个时间戳文件（timestamp file），而且它们都是顶层时间戳文件*validate*的必要条件。

```
$(OUTPUT_DIR)/chk_macros_tabs: $(ALL_XML_OUT)
        # 查找宏和跳格符……
        $(QUIET)! $(EGREP) --ignore-case            \
                           --line-number            \
                           --regexp='\b(m4_|mp_)'   \
                           --regexp='\011'          \
                           $^
        $(TOUCH) $@
```

第一项检查会搜索预处理期间未扩展的m4宏。这表示不是宏被拼写错，就是宏没有被定义。此项检查还会扫描跳格符。当然，这些状况或许应该都不会发生，不过总是有出差错的时候！命令脚本中有一点值得注意，那就是$(QUIET)之后的惊叹号。其目的在于将egrep的结束状态取反。也就是说，如果egrep找到了其中的一个模式，make应该将此视为命令执行失败。

```
$(OUTPUT_DIR)/chk_fixme: $(ALL_XML_OUT)
        # 查找RM:和FIXME……
        $(QUIET)$(AWK)                                          \
                '/FIXME/ { printf "%s:%s: %s\n", FILENAME, NR, $$0 }   \
                 /^ *RM:/  {                                    \
                           if ( $$0 !~ /RM: Done/ )             \
                           printf "%s:%s: %s\n", FILENAME, NR, $$0   \
                         }' $(subst $(OUTPUT_DIR)/,$(SOURCE_DIR)/,$^)
        $(TOUCH) $@
```

此项检查用来搜索尚未处理的标记。显然，对于任何标有FIXME的文字应该先加以改正，然后删掉标签。此外，任何出现RM:的地方，如果后面没有跟着Done这个字样，应该加以注记。注意printf函数的格式是如何依照编译器的错误信息的标准格式来安排的。这样，认得编译器的错误信息的标准工具才会正确地处理这些警告信息。

```
$(OUTPUT_DIR)/chk_duplicate_macros: $(SOURCE_DIR)/macros.m4
        # 查找重复的宏 ……
        $(QUIET)) $(EGREP) --only-matching                    \
              "\[^]+'," $< |                                  \
        $(SORT) |                                             \
        uniq -c |                                             \
        $(AWK) '$$1 > 1 { printf "$>:0: %s\n", $$0 }' | \
        $(EGREP) "^"
        $(TOUCH) $@
```

此项检查会搜索m4宏文件中重复的宏定义。由于m4处理器并不会把重复定义视为错误，所以我才特别加入此项检查。整个处理流程是这样的：从每个宏取出已定义的符号、排序、计算重复的数目，滤除重复数为1的每一行，最后使用egrep以作为它的结束状态。同样地，注意到结束状态经过取反，所以只有在找到东西时才会产生make的错误。

```
ALL_EXAMPLES := $(TMP)/all_examples

$(OUTPUT_DIR)/chk_orphaned_examples: $(ALL_EXAMPLES) $(DEPENDENCY_FILES)
        $(QUIET)$(AWK)  F/ '/(EXAMPLES|OUTPUT)_DIR/ { print $$3 }' \
                $(filter %.d,$^) |                                  \
        $(SORT) -u |                                                \
        comm -13 - $(filter-out %.d,$^)
        $(TOUCH) $@

.INTERMEDIATE: $(ALL_EXAMPLES)
$(ALL_EXAMPLES):
        # 查找未使用的范例……
        $(QUIET) ls -p $(EXAMPLES_DIR) |          \
        $(AWK) '/CVS/ { next }                    \
                /\// { print substr($$0, 1, length - 1) }' > $@
```

最后一项检查将会搜索尚未被正文引用到的范例。此处的工作目标使用了一个有趣的技巧，它需要两组输入文件：所有的范例目录以及所有的XML依存文件。它会使用filter和filter-out将必要条件列表分成这两组。范例目录列表的产生是通过ls -p（这会为每个目录附加一个斜线符号）以及扫描斜线符号。整个处理流程是这样的：从必要条件列表中取出XML依存文件，输出其中所找到的范例目录，移除任何重复的部分。这些就是在正文中被实际引用到的范例。这份列表会被放入comm的标准输入，然而所有已知的范例目录会被放入第二个文件中。-13选项用来告诉comm应该只输出第二个字段所找到的每一行（即没有被依存文件引用到的目录）。

Linux 内核的 makefile

Linux内核的*makefile*是一个“在复杂的编译环境中使用make”的绝佳范例。尽管说明Linux内核的结构和编译方式已经超出了本书的范围，不过我们可以看看内核编译系统使用make时，一些值得注意的用法。*http://archive.linuxsymposium.org/ols2003/*

*Proceedings/All-Reprints/Reprint-Germaschewski-OLS2003.pdf*上对2.5/2.6版内核的编译过程以及从2.4版以来的变革有更完整的讨论。

因为*makefile*有如此多的方面的功能，我们只会探讨其中适合用在各种应用程序中的功能。首先，我们将会看到如何使用单字母make变量来模拟单字母命令行选项，以及如何分开源文件和二进制文件树来让用户能够从源文件树调用make。接着，我们将会查看*makefile*控制输出细节的方法。然后，我们将会看到一些最值得注意的用户自定义函数，以及这些函数如何降低代码的重复性、改进可读性以及提供封装性。最后，我们将会看到*makefile*实现简易说明措施的方法。

Linux内核的编译过程就是大部分自由软件所依循的模式：设定配置、进行编译、进行安装。尽管许多自由和开放源代码软件包还会用到一个独立的*configure*脚本（通常是由autoconf所建立），不过Linux内核的*makefile*却是以make进行配置的设定，并且间接调用其他脚本与辅助程序。

当配置设定阶段完成之后，只要使用make或make all就可以编译内核本身、所有的模块以及经压缩的内核映像（依次是vmlinux、modules和bzImage等工作目标）。每次编译内核的时候都会把*version.o*文件中所提供的具唯一性的版本编号链接进内核，此编号（以及*version.o*文件）会被*makefile*本身所更新。

其中，你可能会想要在自己的*makefile*中采用的*makefile*功能包括：命令行选项的处理、命令行意图的分析、在各编译过程之间存储编译状态以及管理make的输出。

命令行选项

此*makefile*的第一个部分包括了从命令行来设定编译选项的代码。下面摘录了用来控制verbose标记的代码：

```
# To put more focus on warnings, be less verbose as default
# Use 'make V=1' to see the full commands
ifdef V
  ifeq ("$(origin V)", "command line")
    KBUILD_VERBOSE = $(V)
  endif
endif
ifndef KBUILD_VERBOSE
  KBUILD_VERBOSE = 0
Endif
```

嵌套的一对ifdef/ifeq用来确保，只有当V是在命令行上设定的，才会设定KBUILD_VERBOSE变量。如果V是在环境或*makefile*中设定的，将不会有任何效果。如

果KBUILD_VERBOSE尚未设定，随后的ifndef条件语句将会关闭verbose选项。如果你真的要从环境或*makefile*来设定verbose选项，你必须设定KBUILD_VERBOSE而不是V。

然而，请注意，直接在命令行上设定KBUILD_VERBOSE也会让你获得预期的结果。当你要编写一些脚本（或别名）来调用*makefile*时，这个功能可能会很有用。这些脚本本身就具备说明性，就如同使用GNU的长选项。

其他的命令行选项，sparse检查（C）和外部模块（M），也都需要仔细检查，以避免在*makefile*中设定它们。

*makefile*的下一个区段用来处理输出目录选项（O）。这是一段相当复杂的代码，为了将它的结构强调出来，我们会把摘录内容中某些部分替换成删节符号：

```
# kbuild supports saving output files in a separate directory.
# To locate output files in a separate directory two syntax'es are
supported.
# In both cases the working directory must be the root of the kernel src.
# 1) O=
# Use "make O=dir/to/store/output/files/"
#
# 2) Set KBUILD_OUTPUT
# Set the environment variable KBUILD_OUTPUT to point to the directory
# where the output files shall be placed.
# export KBUILD_OUTPUT=dir/to/store/output/files/
# make
#
# The O= assigment takes precedence over the KBUILD_OUTPUT environment variable.
# KBUILD_SRC is set on invocation of make in OBJ directory
# KBUILD_SRC is not intended to be used by the regular user (for now)
ifeq ($(KBUILD_SRC),)

  # OK, Make called in directory where kernel src resides
  # Do we want to locate output files in a separate directory?
  ifdef O
    ifeq ("$(origin O)", "command line")
      KBUILD_OUTPUT := $(O)
    endif
  endif
  ...
  ifneq ($(KBUILD_OUTPUT),)
    ...
    .PHONY: $(MAKECMDGOALS)

    $(filter-out _all,$(MAKECMDGOALS)) _all:
            $(if $(KBUILD_VERBOSE:1=),@)$(MAKE) -C $(KBUILD_OUTPUT)          \
            KBUILD_SRC=$(CURDIR)          KBUILD_VERBOSE=$(KBUILD_VERBOSE)   \
            KBUILD_CHECK=$(KBUILD_CHECK) KBUILD_EXTMOD="$(KBUILD_EXTMOD)"   \
            -f $(CURDIR)/Makefile $@
    # Leave processing to above invocation of make
    skip-makefile := 1
```

```
  endif # ifneq ($(KBUILD_OUTPUT),)
endif # ifeq ($(KBUILD_SRC),)

# We process the rest of the Makefile if this is the final invocation of make
ifeq ($(skip-makefile),)
  ... 此处是 makefile 的其余部分 ...
endif   # skip-makefile
```

基本上，这段代码是说，如果KBUILD_OUTPUT设定了，就会在KBUILD_OUTPUT所定义的输出目录中递归调用make。KBUILD_SRC会被设定为make原本被执行的目录，*makefile*也是从该处取得的。make看不到*makefile*的其余部分，因为skip-makefile将会被设定。递归式make将会重新读取*makefile*，不过只有这一次运行中KBUILD_SRC才会被设定，所以skip-makefile将不会被定义，于是*makefile*的其余部分将会被make所读取和处理。

命令行选项的处理过程到此结束。ifeq ($(skip-makefile),)区段中包含了*makefile*的其余部分。

配置与编译

此*makefile*包含了配置工作目标（configuration target）以及编译工作目标（build target）。配置工作目标具有menuconfig、defconfig等形式，维护工作目标（maintenance target），比如clean，也会被视为配置工作目标。其他工作目标，像all、vmlinux和modules，则属于编译工作目标。配置工作目标的调用结果主要是两个文件：*.config*和*.config.cmd*。这两个文件会被*makefile*的编译工作目标引入，而不会被配置工作目标引入（因为它们就是配置工作目标设定的）。你还可以在调用make的命令行上同时指定配置工作目标以及编译工作目标，例如：

```
$ make oldconfig all
```

就此例来说，*makefile*会递归调用它自己来个别处理每个工作目标，因此配置工作目标与编译工作目标的处理是分开进行的。

用来控制配置工作目标、编译工作目标以及两者并用的代码的开头如下所示：

```
# To make sure we do not include .config for any of the *config targets
# catch them early, and hand them over to scripts/kconfig/Makefile
# It is allowed to specify more targets when calling make, including
# mixing *config targets and build targets.
# For example 'make oldconfig all'.
# Detect when mixed targets is specified, and make a second invocation
# of make so .config is not included in this case either (for *config).
no-dot-config-targets := clean mrproper distclean \
                         cscope TAGS tags help %docs check%
```

```
config-targets := 0
mixed-targets := 0
dot-config     := 1
```

no-dot-config-targets变量列出了不需要*.config*文件的额外工作目标。这段代码之后会对config-targets、mixed-targets和dot-config等变量进行初始化的动作。如果命令行上指定了任何的编译工作目标，dot-config变量的值会被设成1。最后，如果同时指定配置和编译工作目标，mixed-targets变量的值会被设成1。

用来设定dot-config的代码如下所示：

```
ifneq ($(filter $(no-dot-config-targets), $(MAKECMDGOALS)),)
  ifeq ($(filter-out $(no-dot-config-targets), $(MAKECMDGOALS)),)
    dot-config := 0
  endif
endif
```

如果MAKECMDGOALS中包含配置工作目标，filter表达式的求值结果就是非空值。如果filter表达式的求值结果为非空值，ifneq的部分就为“真”。这段代码很难看得懂，因为它包含了两次否定的意思。如果MAKECMDGOALS中只包含配置工作目标，则ifeq表达式的值为“真”。所以，如果MAKECMDGOALS中包含配置工作目标而且只有配置工作目标，dot-config将会被设成0。如果能够改用如下较冗长的实现方式，将可让这两个条件表达式所代表的意义较为清楚：

```
config-target-list := clean mrproper distclean \
                      cscope TAGS tags help %docs check%

config-target-goal := $(filter $(config-target-list), $(MAKECMDGOALS))
build-target-goal := $(filter-out $(config-target-list), $(MAKECMDGOALS))

ifdef config-target-goal
  ifndef build-target-goal
    dot-config := 0
  endif
endif
```

ifdef形式被用来取代ifneq，因为空值得变量会被视为未定义，不过你必须小心，以避免让一个变量只包含空格所构成的字符串（这会使得它被视为已定义过）。

config-targets和mixed-targets变量被设定在下一段代码中：

```
ifeq ($(KBUILD_EXTMOD),)
  ifneq ($(filter config %config,$(MAKECMDGOALS)),)
    config-targets := 1
    ifneq ($(filter-out config %config,$(MAKECMDGOALS)),)
      mixed-targets := 1
    endif
  endif
endif
```

当所进行的是外部模块的编译动作，而不是一般的编译动作时，KBUILD_EXTMOD将会被设成非空值。如果MAKECMDGOALS所包含的工作目标具有config后缀，则第一个ifneq的求值结果为“真”；如果MAKECMDGOALS中也包含非config的工作目标，则ifneq的求值结果为“真”。

一旦变量被设定之后，它们会被使用在一个“具有4个分支的if-else”链中。这段代码已经过浓缩与缩排，以便把它的结构强调出来：

```
ifeq ($(mixed-targets),1)
  # We're called with mixed targets (*config and build targets),
  # Handle them one by one.
  %:: FORCE
        $(Q)$(MAKE) -C $(srctree) KBUILD_SRC= $@
else
  ifeq ($(config-targets),1)
    # *config targets only - make sure prerequisites are updated, and descend
    # in scripts/kconfig to make the *config target
    %config: scripts_basic FORCE
            $(Q)$(MAKE) $(build)=scripts/kconfig $@
  else
    # Build targets only - this includes vmlinux, arch specific targets, clean
    # targets and others. In general all targets except *config targets.
    ...
    ifeq ($(dot-config),1)
      # In this section, we need .config
      # Read in dependencies to all Kconfig* files, make sure to run
      # oldconfig if changes are detected.
      -include .config.cmd
      include .config

      # If .config needs to be updated, it will be done via the dependency
      # that autoconf has on .config.
      # To avoid any implicit rule to kick in, define an empty command
      .config: ;

      # If .config is newer than include/linux/autoconf.h, someone tinkered
      # with it and forgot to run make oldconfig
      include/linux/autoconf.h: .config
            $(Q)$(MAKE) -f $(srctree)/Makefile silentoldconfig
    else
      # Dummy target needed, because used as prerequisite
      include/linux/autoconf.h: ;
    endif

    include $(srctree)/arch/$(ARCH)/Makefile
    …… 此处有许多make的代码 ……
  endif #ifeq ($(config-targets),1)
endif #ifeq ($(mixed-targets),1)
```

第一个分支ifeq ($(mixed-targets),1)，用来处理混用(mixed)的命令行参数。只有这个分支里的工作目标完全采用通用的模式规则，因此你看不到处理工作目标的特定

规则（这些规则位于另一个条件分支中），每个工作目标只会调用模式规则一次。这就是具有配置工作目标和编译工作目标的命令行如何被分开成较简单的命令行的方法。通用模式规则的命令行将会为每个工作目标递归调用 make，这使得相同的逻辑可以应用在每个工作目标上。FORCE 必要条件用来取代 .PHONY，因为如下的模式规则：

```
%:: FORCE
```

是无法被声明成 .PHONY 的。所以可行方法似乎就是固定使用 FORCE 以维持其一致性。

当命令行上只有一个配置工作目标时，会调用 if-else 链的第二个分支 ifeq($(config-targets),1)。在这个分支中，主要的工作目标是模式规则 %config（其他的工作目标已被省略）。它的脚本会为每个工作目标在 *scripts/kconfig* 子目录中递归地调用 make。奇怪的 $(build) 结构就定义在 *makefile* 的末端：

```
# Shorthand for $(Q)$(MAKE) -f scripts/Makefile.build obj=dir
# Usage:
# $(Q)$(MAKE) $(build)=dir
build := -f $(if $(KBUILD_SRC),$(srctree)/)scripts/Makefile.build obj
```

如果设定了 KBUILD_SRC，-f 选项就会对 *scripts* 组件的 *makefile* 使用完整的路径，否则只会使用相对路径。接着，obj 变量会被设定成等号的右边部分。

第三个分支 ifeq($(dot-config),1)，用来处理需要引入两个自动产生的配置文件（*.config* 和 *.config.cmd*）的编译工作目标。最后一个分支只是用来为 *autoconf.h* 引入一个假想工作目标，让它能够被作为一个必要条件来用，即使它不存在。

makefile 的其余部分（包含了编译内核和模块的代码）大多数是按照第三和第四个分支的指示进行的。

控制命令是否被输出

内核的 *makefile* 使用了一个新颖的技术，用来控制命令被输出的详细程度。每个重要的工作都会有 verbose（详细模式）和 quiet（安静模式）两个版本。verbose 的版本就是让命令以自然的方式执行，并将它存入名为 cmd_*action* 的变量中。quiet 的版本是一个用来描述动作的简短信息，并且将它存入名为 quiet_cmd_*action* 的变量中。举例来说，如下的命令可用来产生 emacs 标记：

```
quiet_cmd_TAGS = MAKE $@
      cmd_TAGS = $(all-sources) | etags -
```

此类命令将会通过 cmd 函数来执行：

```
# If quiet is set, only print short version of command
```

```
cmd = @$(if $($(quiet)cmd_$(1)),\
        echo '  $($(quiet)cmd_$(1))' &&) $(cmd_$(1))
```

如果想要调用编译 emacs 标记的命令，*makefile* 必须包含如下的规则：

```
TAGS:
        $(call cmd,TAGS)
```

注意 cmd 函数开头的 @，所以此函数只会输出来自 echo 命令的信息。在标准模式中，quiet 变量是空的，而且 if 所测试的表达式 $($(quiet)cmd_$(1))，会被扩展成 $(cmd_TAGS)。因为这个变量是空的，所以整个函数会被扩展成：

```
echo '  $(all-sources) | etags -' && $(all-sources) | etags -
```

如果需要使用 quiet 的版本，则 quiet 变量的值为 quiet_，而且函数会被扩展成：

```
echo '  MAKE $@' && $(all-sources) | etags -
```

这个变量还可以被设定成 silent_。因为并不存在 silent_cmd_TAGS 命令，所以此值将会使得 cmd 函数不输出任何内容。

是否输出命令的管理有时会变得比较复杂，特别是如果命令中包含单引号。为解决此类问题，*makefile* 包含了如下的代码：

```
$(if $($(quiet)cmd_$(1)),echo '  $(subst ','\'',$($(quiet)cmd_$(1)))';)
```

此处，echo 命令中包含了一个以“经转义处理的单引号”来取代单引号的替换操作，这让它们能够被正确地输出。

至于较小的命令（没有必要自找麻烦去建立 cmd_ 和 quiet_cmd_ 变量）则会被前置 $(Q)，它的值可能是空的或是 @：

```
ifeq ($(KBUILD_VERBOSE),1)
  quiet =
  Q =
else
  quiet=quiet_
  Q = @
endif

# If the user is running make -s (silent mode), suppress echoing of
# commands

ifneq ($(findstring s,$(MAKEFLAGS)),)
  quiet=silent_
endif
```

用户自定义函数

内核的 *makefile* 中定义了一些函数，我们在此处只讨论最重要的几个。以下所举的例子已经重新编排成较具可读性的形式。

check_gcc 函数（译注 1）用来选择一个 gcc 命令行选项。

```
# $(call check_gcc,preferred-option,alternate-option)
check_gcc =                                                  \
    $(shell if $(CC) $(CFLAGS) $(1) -S -o /dev/null \
               -xc /dev/null > /dev/null 2>&1;                \
            then                                             \
              echo "$(1)";                                     \
            else                                             \
              echo "$(2)";                                     \
            fi ;)
```

此函数的作用，就是以较优选的命令行选项在一个空的输入文件上调用gcc。输出文件、标准输出以及标准错误文件都会被丢弃。如果gcc命令执行成功，这表示较优选的命令行选项对此结构来说是有效的，于是它会被此函数返回，否则，表示此选项是无效的，于是此函数会返回替代的选项。你可以在 *arch/i386/Makefile* 文件中看到如下的例子：

```
# prevent gcc from keeping the stack 16 byte aligned
CFLAGS += $(call check_gcc,-mpreferred-stack-boundary=2,)
```

if_changed_dep 函数用来产生依存信息，它所用到的技术值得我们加以了解。

```
# execute the command and also postprocess generated
# .d dependencies file
if_changed_dep =                                                 \
    $(if                                                         \
      $(strip $?                                                 \
        $(filter-out FORCE $(wildcard $^),$^)                    \
        $(filter-out $(cmd_$(1)),$(cmd_$@))                      \
        $(filter-out $(cmd_$@),$(cmd_$(1))))),                   \
      @set -e;                                                   \
      $(if $($(quiet)cmd_$(1)),                                  \
        echo '  $(subst ','\'',$($(quiet)cmd_$(1)))';)           \
      $(cmd_$(1));                                               \
      scripts/basic/fixdep                                       \
         $(depfile)                                              \
         $@                                                      \
         '$(subst $$,$$$$,$(subst ','\'',$(cmd_$(1))))' \
         > $(@D)/.$(@F).tmp;                                     \
      rm -f $(depfile);                                          \
      mv -f $(@D)/.$(@F).tmp $(@D)/.$(@F).cmd)
```

译注 1：这个函数已经停用，现在使用的是 cc-option。

此函数由一个 if 子句组成。它的测试细节相当难懂，不过很显然，如果依存文件应该被重新产生，它就是非空值。标准的依存信息与文件的时间戳的改变有关。内核编译系统还会为此项工作加入另一个技巧：使用了一个变化极大的编译器选项来控制各个组件的结构和行为。为了确定命令行选项在编译期间适当发生了作用，*makefile* 被实现成：如果用于特定工作目标的命令行选项有所变化，相应的文件就会被重新编译。让我们来看看这是如何办到的。

基本上，用来编译内核中每个文件的命令，会被存放在一个 *.cmd* 文件里。当有一个编译动作被执行时，make会读取这个 *.cmd* 文件，并且比较当前的编译命令与上一次的命令。如果有所不同，就会重新产生 *.cmd* 依存文件，这会使得目标文件被重新编译。*.cmd* 文件通常会包含工作目标文件的实际文件与用来记录命令行选项的单一变量的依存关系。举例来说，文件 *arch/i386/kernel/cpu/mtrr/if.c* 会产生如下的依存文件（经过删减）：

```
cmd_arch/i386/kernel/cpu/mtrr/if.o := gcc -Wp,-MD -; if.c

deps_arch/i386/kernel/cpu/mtrr/if.o := \
  arch/i386/kernel/cpu/mtrr/if.c \
  ...

arch/i386/kernel/cpu/mtrr/if.o: $(deps_arch/i386/kernel/cpu/mtrr/if.o)
$(deps_arch/i386/kernel/cpu/mtrr/if.o):
```

让我们回到 if_changed_dep 函数，送给 strip 的第一个参数，就是时间戳在工作目标（的时间戳）之后的所有必要条件 —— 如果有的话。送给 strip 的第二个参数，就是非文件与空工作目标FORCE的所有必要条件。比较难理解的是最后两个 filter-out 调用：

```
$(filter-out $(cmd_$(1)),$(cmd_$@))
$(filter-out $(cmd_$@),$(cmd_$(1)))
```

如果命令行选项有所改变，这两个调用或其中之一将会被扩展成非空字符串。宏 $(cmd_$(1))是当前的命令，而 $(cmd_$@)将会是之前的命令，例如你刚才所看到的 cmd_arch/i386/kernel/cpu/mtrr/if.o变量。如果新的命令中包含额外的选项，则第一个 filter-out 将会是空的，而第二个将会被扩展成新的选项。如果新的命令中包含了较少的选项，则第一个命令将会包含被删除的函数，而第二个命令将会是个空的。请注意，因为filter-out的参数可以是一串单词（每个单词会被视为独立的模式），所以选项的次序可以改变，而且 filter-out 仍然可以准确区分出被新增或被删除的选项。做得相当漂亮。

命令脚本中的第一条语句会设定一个shell选项，好让它在错误发生时立即结束运行。这样可避免多行脚本在问题发生时破坏文件。对于简单的脚本，你可以使用另一个方法来达到此效果，那就是用 && 而不是分号来连接每条语句。

下一条语句是一个echo命令，它用到了“控制命令是否被输出”一节中所提到的技术，后面跟着依存关系产生命令。此命令会写入$(depfile)文件，此文件会经过scripts/basic/fixdep的转换。fixdep命令行中的嵌套subst函数里，首先会转义单引号，然后转义$$（shell语法中的当前进程编号）。

最后，如果没有错误发生，中间文件$(depfile)会被移除，而所产生的依存文件（具*.cmd*扩展名）会被移往安置处。

下一个函数if_changed_rule使用了跟if_changed_dep一样的比较技术，用来控制一个命令的执行与否：

```
# Usage: $(call if_changed_rule,foo)
# will check if $(cmd_foo) changed, or any of the prequisites changed,
# and if so will execute $(rule_foo)

if_changed_rule =                                          \
    $(if $(strip $?                                        \
            $(filter-out $(cmd_$(1)),$(cmd_$(@F)))         \
            $(filter-out $(cmd_$(@F)),$(cmd_$(1))))),      \
      @$(rule_$(1))
```

在最上层的*makefile*中，此函数会使用下面这些宏来链接内核：

```
# This is a bit tricky: If we need to relink vmlinux, we want
# the version number incremented, which means recompile init/version.o
# and relink init/init.o. However, we cannot do this during the
# normal descending-into-subdirs phase, since at that time
# we cannot yet know if we will need to relink vmlinux.
# So we descend into init/ inside the rule for vmlinux again.
...

quiet_cmd_vmlinux__ = LD $@
define cmd_vmlinux__
  $(LD) $(LDFLAGS) $(LDFLAGS_vmlinux) \
    ...
endef

# set -e makes the rule exit immediately on error

define rule_vmlinux__
  +set -e;                                                 \
  $(if $(filter .tmp_kallsyms%,$^),,                       \
    echo '  GEN     .version';                             \
    . $(srctree)/scripts/mkversion > .tmp_version;         \
    mv -f .tmp_version .version;                           \
    $(MAKE) $(build)=init;)                                \
  $(if $($(quiet)cmd_vmlinux__),                           \
    echo '  $($(quiet)cmd_vmlinux__)' &&)                  \
  $(cmd_vmlinux__);                                        \
  echo 'cmd_$@ := $(cmd_vmlinux__)' > $(@D)/.$(@F).cmd
endef
```

```
define rule_vmlinux
  $(rule_vmlinux__);                    \
  $(NM) $@ |                            \
  grep -v '\(compiled\)\|...' | \
  sort > System.map
endef
```

if_changed_rule函数会被用来调用rule_vmlinux，它会进行链接并且编译最后的*System.map*。正如此*makefile*中的注释所说，rule_vmlinux__函数必须在重新链接*vmlinux*之前，重新产生内核的版本文件以及重新链接*init.o*。这受到rule_vmlinux__中第一个if语句的控制。第二个if语句用来控制链接命令$(cmd_vmlinux__)的是否输出。链接命令之后，实际被执行的命令会被记录到一个*.cmd*文件中，以备下次编译时比较之用。

第十二章

makefile 的调试

makefile 的调试有点像魔法。可惜，并不存在 *makefile* 调试器之类的东西可用来查看特定规则是如何被求值的，或某个变量是如何被扩展的。相反，大部分的调试过程只是在执行输出的动作以及查看 *makefile*。事实上，GNU make 提供了若干可以协助调试的内置函数以及命令行选项。

用来调试 *makefile* 的一个最好方法就是加入调试挂钩以及使用具保护的编程技术，让你能够在事情出错时恢复原状。我将会介绍若干基本的调试技术以及我所发现的最有用的具保护能力的编码习惯。

make 的调试功能

warning 函数非常适合用来调试难以捉摸的 *makefile*。因为 warning 函数会被扩展成空字符串，所以它可以放在 *makefile* 中的任何地方：开始的位置、工作目标或必要条件列表中以及命令脚本中。这让你能够在最方便查看变量的地方输出变量的值。例如：

```
$(warning A top-level warning)

FOO := $(warning Right-hand side of a simple variable)bar
BAZ = $(warning Right-hand side of a recursive variable)boo

$(warning A target)target: $(warning In a prerequisite list)makefile
$(BAZ)
        $(warning In a command script)
        ls
$(BAZ):
```

这会产生如下的输出：

```
$ make
makefile:1: A top-level warning
```

```
makefile:2: Right hand side of a simple variable
makefile:5: A target
makefile:5: In a prerequisite list
makefile:5: Right hand side of a recursive variable
makefile:8: Right-hand side of a recursive variable
makefile:6: In a command script
ls
makefile
```

请注意，warning函数的求值方式是按照make标准的立即和延后求值算法。虽然对BAZ的赋值动作中包含了一个warning函数，但是直到BAZ在必要条件列表中被求值后，这个信息才会被输出来。

"可以在任何地方安插warning调用"的这个特性，让它能够成为一个基本的调试工具。

命令行选项

我找到了三个最适合用来调试的命令行选项：--just-print (-n)、--print-database (-p) 以及 --warn-undefined-variables。

--just-print

在一个新的*makefile*工作目标上，我所做的第一个测试就是以 --just-print (-n) 选项来调用make。这会使得make读进*makefile*并且输出它更新工作目标时将会执行的命令，但是不会真的执行它们。GNU make有一个方便的功能，就是允许你为将被输出的命令标上安静模式修饰符 (@)。

这个选项被假设可以抑制所有命令的执行动作，然而这只在特定的状况下为真。实际上，你必须小心以对。尽管make不会运行命令脚本，但是在立即的语境之中，它会对shell函数调用进行求值动作。例如：

```
REQUIRED_DIRS = ...
_MKDIRS := $(shell for d in $(REQUIRED_DIRS); \
            do                                \
              [[ -d $$d ]] || mkdir -p $$d;   \
            done)

$(objects) : $(sources)
```

正如我们之前所见，_MKDIRS简单变量的目的是触发必要目录的创建动作。如果这个*makefile*是以 --just-print 选项的方式运行的，那么当make读进*makefile*时，shell命令将会一如往常般被执行。然后，make将会输出（但不会执行）更新$(objects)文件列表所需要进行的每个编译命令。

--print-data-base

--print-data-base (-p) 是另一个你常会用到的选项。它会运行*makefile*，显示GNU版权信息以及make所运行的命令，然后输出它的内部数据库。数据库里的数据将会依种类划分成以下几个组：variables、directories、implicit rules、pattern-specific variables、files（explicit rules）以及 vpath search path。如下所示：

```
# GNU Make 3.80
# Copyright (C) 2002  Free Software Foundation, Inc.
# This is free software; see the source for copying conditions.
# There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
# PARTICULAR PURPOSE.
正常的命令将会在此处执行

# Make data base, printed on Thu Apr 29 20:58:13 2004
# Variables
...
# Directories
...
# Implicit Rules
...
# Pattern-specific variable values
...
# Files
...
# VPATH Search Paths
```

让我们更详细地查看以上这几个区段。

变量区段（variable）将会列出每个变量以及具描述性的注释：

```
# automatic
<D = $(patsubst %/,%,$(dir $<))
# environment
EMACS_DIR = C:/usr/emacs-21.3.50.7
# default
CWEAVE = cweave
# makefile (from '../mp3_player/makefile', line 35)
CPPFLAGS = $(addprefix -I ,$(include_dirs))
# makefile (from '../ch07-separate-binaries/makefile', line 44)
RM := rm -f
# makefile (from '../mp3_player/makefile', line 14)
define make-library
  libraries += $1
  sources   += $2

  $1: $(call source-to-object,$2)
        $(AR) $(ARFLAGS) $$@ $$^
endef
```

自动变量不会被显示出来，但是通过它们可以方便变量的获得，像$(<D)。注释所指出的是origin函数所返回的变量类型（参见“较不重要的杂项函数”一节）。如果变量被定义在一个文件中，则会在注释中指出其文件名以及该定义所在的行号。简单变量和递归变量的差别在于赋值运算符。简单变量的值将会被显示成右边部分被求值的形式。

下一个区段标示为Directories，它对make开发人员比对make用户有用。它列出了将会被make检查的目录，包括可能会存在的SCCS和RCS子目录，但它们通常不存在。对每个目录来说，make会显示实现细节，比如设备编号、inode以及文件名模式匹配的统计数据。

接着是Implicit Rules区段。这个区段包含了make数据库中所有的内置的和用户自定义的模式规则。此外，对于那些定义在文件中的规则，它们的注释将会指出文件名以及行号：

```
%.c %.h: %.y
# commands to execute (from `../mp3_player/makefile', line 73):
        $(YACC.y) --defines $<
        $(MV) y.tab.c $*.c
        $(MV) y.tab.h $*.h

%: %.c
#  commands to execute (built-in):
        $(LINK.c) $^ $(LOADLIBES) $(LDLIBS) -o $@

%.o: %.c
#  commands to execute (built-in):
        $(COMPILE.c) $(OUTPUT_OPTION) $<
```

查看这个区段，是让你能够熟悉make内置规则的变化和结构的最佳方法。当然，并非所有的内置规则都会被实现成模式规则。如果你没有找到你想要的规则，可以查看Files区段，旧式后缀规则就列在该处。

下一个区段被标示为Pattern-specific variables，此处所列出的是定义在*makefile*里的模式专属变量。所谓模式专属变量，就是变量定义的有效范围被限定在相关的模式规则执行的时候。例如，模式变量YYLEXFLAG被定义成：

```
%.c %.h: YYLEXFLAG := -d
%.c %.h: %.y
        $(YACC.y) --defines $<
        $(MV) y.tab.c $*.c
        $(MV) y.tab.h $*.h
```

将会被显示成：

```
# Pattern-specific variable values
```

```
%.c :
# makefile (from 'Makefile', line 1)
# YYLEXFLAG := -d
# variable set hash-table stats:
# Load=1/16=6%, Rehash=0, Collisions=0/1=0%
%.h :
# makefile (from 'Makefile', line 1)
# YYLEXFLAG := -d
# variable set hash-table stats:
# Load=1/16=6%, Rehash=0, Collisions=0/1=0%

# 2 pattern-specific variable values
```

接着是 Files 区段，此处所列出的都是与特定文件有关的自定义和后缀规则：

```
# Not a target:
.p.o:
#  Implicit rule search has not been done.
#  Modification time never checked.
#  File has not been updated.
#  commands to execute (built-in):
        $(COMPILE.p) $(OUTPUT_OPTION) $<

lib/ui/libui.a: lib/ui/ui.o
#  Implicit rule search has not been done.
#  Last modified 2004-04-01 22:04:09.515625
#  File has been updated.
#  Successfully updated.
#  commands to execute (from '../mp3_player/lib/ui/module.mk', line 3):
        ar rv $@ $?

lib/codec/codec.o: ../mp3_player/lib/codec/codec.c ../mp3_player/lib/codec/
    codec.c ../mp3_player/include/codec/codec.h
#  Implicit rule search has been done.
#  Implicit/static pattern stem: 'lib/codec/codec'
#  Last modified 2004-04-01 22:04:08.40625
#  File has been updated.
#  Successfully updated.
#  commands to execute (built-in):
        $(COMPILE.c) $(OUTPUT_OPTION) $<
```

中间文件与后缀规则会被标示为 Not a target，其余是工作目标。每个文件将会包含注释，用以指出 make 是如何处理此规则的。被找到的文件在被显示的时候将会通过标准的 vpath 搜索来找出其路径。

最后一个区段被标示为 VPATH Search Paths，列出了 VPATH 的值以及所有的 vpath 模式。

对于大规模使用 eval 以及用户自定义函数来建立复杂的变量和规则的 *makefile* 来说，查看它们的输出结果通常是确认宏是否已被扩展成预期值的唯一方法。

--warn-undefined-variables

这个选项会使得make在未定义的变量被扩展时显示警告信息。因为未定义的变量会被扩展成空字符串，这常见于变量名称打错而且很长一段时间未被发现到。这个选项有个问题，这也是为什么我很少使用这个选项的原因，那就是许多内置规则都会包含未定义的变量以作为用户自定义值的挂钩。所以使用这个选项来运行make必然会产生许多不是错误的警告信息，而且对用户的*makefile*没有什么用处。例如：

```
$ make --warn-undefined-variables -n
makefile:35: warning: undefined variable MAKECMDGOALS
makefile:45: warning: undefined variable CFLAGS
makefile:45: warning: undefined variable TARGET_ARCH
...
makefile:35: warning: undefined variable MAKECMDGOALS
make: warning: undefined variable CFLAGS
make: warning: undefined variable TARGET_ARCH
make: warning: undefined variable CFLAGS
make: warning: undefined variable TARGET_ARCH
...
make: warning: undefined variable LDFLAGS
make: warning: undefined variable TARGET_ARCH
make: warning: undefined variable LOADLIBES
make: warning: undefined variable LDLIBS
```

不过，此命令在需要捕获此类错误的某些场合上可能非常有用。

--debug 选项

当你需要知道make如何分析你的依存图时，可以使用--debug选项。除了运行调试器，这个选项是让你获得最详细信息的另一个方法。你有五个调试选项以及一个修饰符可用，分别是：basic、verbose、implicit、jobs、all以及makefile。

如果调试选项被指定成--debug，就是在进行basic调试，如果调试选项被指定成-d，就是在进行all调试，如果要使用选项的其他组合，则可以使用--debug=*option1*,*option2*这个以逗号为分隔符的列表，此处的选项可以是下面任何一个单词（实际上，make只会查看第一个字母）：

basic

这是所提供的信息最不详细的基本调试功能。启用时，make会输出被发现尚未更新的工作目标并更新动作的状态。它的输出会像下面这样：

```
File all does not exist.
  File app/player/play_mp3 does not exist.
    File app/player/play_mp3.o does not exist.
   Must remake target app/player/play_mp3.o.
```

```
gcc ... ../mp3_player/app/player/play_mp3.c
   Successfully remade target file app/player/play_mp3.o.
```

verbose

这个选项会设定basic选项，以及提供关于“哪些文件被分析、哪些必要条件不需要重建等”的额外信息：

```
File all does not exist.
 Considering target file app/player/play_mp3.
  File app/player/play_mp3 does not exist.
   Considering target file app/player/play_mp3.o.
    File app/player/play_mp3.o does not exist.
     Pruning file ../mp3_player/app/player/play_mp3.c.
     Pruning file ../mp3_player/app/player/play_mp3.c.
     Pruning file ../mp3_player/include/player/play_mp3.h.
    Finished prerequisites of target file app/player/play_mp3.o.
   Must remake target app/player/play_mp3.o.
gcc ... ../mp3_player/app/player/play_mp3.c
   Successfully remade target file app/player/play_mp3.o.
   Pruning file app/player/play_mp3.o.
```

implicit

这个选项会设定basic选项，以及提供关于“为每个工作目标搜索隐含规则”的额外信息：

```
File all does not exist.
  File app/player/play_mp3 does not exist.
  Looking for an implicit rule for app/player/play_mp3.
  Trying pattern rule with stem play_mp3.
  Trying implicit prerequisite app/player/play_mp3.o.
  Found an implicit rule for app/player/play_mp3.
    File app/player/play_mp3.o does not exist.
    Looking for an implicit rule for app/player/play_mp3.o.
    Trying pattern rule with stem play_mp3.
    Trying implicit prerequisite app/player/play_mp3.c.
    Found prerequisite app/player/play_mp3.c as VPATH ../mp3_player/app/
player/play_mp3.c
    Found an implicit rule for app/player/play_mp3.o.
   Must remake target app/player/play_mp3.o.
gcc ... ../mp3_player/app/player/play_mp3.c
   Successfully remade target file app/player/play_mp3.o.
```

jobs

这个选项会输出被make调用的子进程的细节，它不会启用basic选项的功能。

```
Got a SIGCHLD; 1 unreaped children.
gcc ... ../mp3_player/app/player/play_mp3.c
Putting child 0x10033800 (app/player/play_mp3.o) PID 576 on the chain.
Live child 0x10033800 (app/player/play_mp3.o) PID 576
Got a SIGCHLD; 1 unreaped children.
Reaping winning child 0x10033800 PID 576
Removing child 0x10033800 PID 576 from chain.
```

all

这会启用前面的所有选项，当你使用 -d 选项时，默认会启用此功能。

makefile

它不会启用调试信息，直到*makefile*被更新——这包括更新任何的引入文件。如果使用此修饰符，make会在重编译*makefile*以及引入文件的时候，输出被选择的信息。这个选项会启用basic选项，all选项也会启用此选项。

编写用于调试的代码

如你所见，并没有太多的工具可用来调试*makefile*，你只有几个方法可以输出若干可能有用的信息。当这些方法都不管用时，你就得将*makefile*编写成可以尽量减少错误发生的机会，或是可以为自己提供一个舞台来协助你进行调试。

这一节所提供的建议被我（有点随意地）分类成：良好的编码习惯、具保护功能的编码以及调试技术等部分。然而一些特殊的项目，像是检查命令的结束状态，可能会被放在良好的编码习惯中或是具保护功能的编码中，做这样的分类适当地反映出了趋势所在。将焦点好好地放在*makefile*上，尽量避免简单行事。采用具保护的编码以避免*makefile*被非预期的事件和环境状态所影响。最后，当缺陷出现时，使用你可以找到的用来压制它们的每个诀窍。

"简洁就是美"（Keep It Simple）的原则（*http://www.catb.org/~esr/jargon/html/K/KISS-Principle.html*）是所有良好设计的核心所在。正如你在前面几章所看到的，*makefile*马上就会变得很复杂——即使是一般的工作，比如依存关系的产生。要对抗"在你的编译系统中加入越来越多的功能"的潮流，你将会失败，但如果你只是不经思索地加入你所发现的每个功能，失败并不会比你这么做的后果还糟。

良好的编码习惯

以我的经验来说，大部分的程序员都不会把*makefile*作为程序来写，因此，他们不会像编写C++或Java时那样细心。事实上，make语言是一个完整的非程序语言。如果可靠性和可维护性对你的编译系统来说很重要，那么请小心编写你的*makefile*，并且尽量遵守良好的编码习惯。

编码健全的*makefile*的重点之一就是检查命令的返回状态。当然，make将会自动检查简单的命令，但是*makefile*通常会使用可能不会处理失败状态的复合命令：

```
do:
        cd i-dont-exist; \
        echo *.c
```

运行时，此*makefile*并不会因为有错误发生而终止运行，尽管这是一个必然会发生的错误：

```
$ make
cd i-dont-exist; \
echo *.c
/bin/sh: line 1: cd: i-dont-exist: No such file or directory
*.c
```

此外，当文件名匹配表达式（globbing expression）找不到任何的.c文件时，它会不动声色地返回文件名匹配表达式。一个比较好的做法，就是在你编码此命令脚本时，使用shell的功能来检查以及防止错误：

```
SHELL = /bin/bash
do:
        cd i-dont-exist && \
        shopt -s nullglob &&
        echo *.c
```

现在cd的错误会被正确传送到make，所以echo命令不会被执行，而且make会因为有错误发生而终止运行。此外，设定bash的nullglob选项，将会使得文件名匹配模式在找不到文件时返回空字符串。（当然，你的应用程序可能比较喜欢文件名匹配模式。）

```
$ make
cd i-dont-exist && \
echo *.c
/bin/sh: line 1: cd: i-dont-exist: No such file or directory
make: *** [do] Error 1
```

另一个良好的编码习惯，就是将你的代码编排成最具可读性的形式。我所看过的*makefile*，多半编排得很差，这必然会造成难以阅读的情况。下面这两段代码哪一个比较容易阅读？

```
_MKDIRS := $(shell for d in $(REQUIRED_DIRS); do [[ -d $$d \
]] || mkdir -p $$d; done)
```

或：

```
_MKDIRS := $(shell                          \
            for d in $(REQUIRED_DIRS);      \
            do                              \
              [[ -d $$d ]] || mkdir -p $$d; \
            done)
```

如果你像大部分人那样，你将会觉得第一段代码比较难分析，不容易找到分号，很难计算有几句语句。这些都是必须注意到的地方。在命令脚本中，你会遇到的语法错误，多半是由于漏掉了分号、反斜线或是其他的分隔符，比如管道（pipe）和逻辑运算符。

此外请注意，并非任何分隔符被漏掉都会产生错误。例如，下面的错误都不会产生shell的语法错误：

```
TAGS:
        cd src \
        ctags --recurse

disk_free:
        echo "Checking free disk space..." \
        df . | awk '{ print $$4 }'
```

把命令编排得具有可读性，将会让以上所提到的错误很容易被发现。编排用户自定义函数的时候可以采用内缩的做法。有时候，宏扩展后的结果中，额外的空格将会造成问题。如果是这样，你可以将它的编排结果封装在strip函数的调用中。编排一长串值时，你可以让每个值自成一行。在每个工作目标的前面加上注释，可以提供简介以及说明参数列表。

下一个良好的编码习惯就是大量使用变量来保存常用的值。如同在程序中一样，过度使用文字值将会造成重复的程序代码，以及导致维护困难与缺陷。变量的另一个优点是在执行期间，你可以基于调试的目的，让make把它们给显示出来。稍后你将会在“调试技术”一节中看到一个不错的命令行界面。

具保护功能的编码

具保护功能的代码，就是如果你的假设或预计有一个是错误的（if测试结果永远为假、assert函数决不会失败或追踪代码）才会执行的代码，这让你能够查看make内部工作的状态。

事实上，你已经在本书其他地方看到过此类代码，不过为了方便起见，此处会重复加以描述。

确认检查就是具保护功能代码的最佳范例。如下的代码范例可用来确认当前所运行的make版本是否为3.80：

```
NEED_VERSION := 3.80
$(if $(filter $(NEED_VERSION),$(MAKE_VERSION)),,
  $(error You must be running make version $(NEED_VERSION).))
```

对Java应用程序来说，它可用来检查CLASSPATH中的文件。

进行确认的代码还可以用来确认某个东西是否为真，比如前一节用来创建目录的代码就是这样。

另一个重要的具保护功能的编码技术，就是使用“流程控制”一节所定义的assert函数。下面是其中的若干版本：

```
# $(call assert,condition,message)
define assert
  $(if $1,,$(error Assertion failed: $2))
endef

# $(call assert-file-exists,wildcard-pattern)
define assert-file-exists
  $(call assert,$(wildcard $1),$1 does not exist)
endef

# $(call assert-not-null,make-variable)
define assert-not-null
  $(call assert,$($1),The variable "$1" is null)
endef
```

我发现在*makefile*中到处声明assert的调用，是找出漏掉和打错的参数以及违反其他假定的既便宜又有效的方法。

我曾在第四章中编写了一对可用来追踪用户自定义函数扩展过程的函数：

```
# $(debug-enter)
debug-enter = $(if $(debug_trace),\
                $(warning Entering $0($(echo-args))))

# $(debug-leave)
debug-leave = $(if $(debug_trace),$(warning Leaving $0))

comma := ,
echo-args   = $(subst ' ','$(comma) ',\
                $(foreach a,1 2 3 4 5 6 7 8 9,'$($a)'))
```

你可以把这些宏调用到自己的函数里，并让它们处在停用状态，直到你需要进行调试。要启用它们时，请将debug_trace设定成任何非空值：

```
$ make debug_trace=1
```

正如第四章所说，这些追踪宏本身存在一些问题，不过仍然可用来追踪缺陷。

最后要介绍的具保护功能的编码技术，就是通过make变量让@命令修饰符的禁用更容易进行：

```
QUIET := @
...
target:
        $(QUIET) some command
```

使用此技术时，如果想看到安静模式命令的执行，你可以在命令行上以如下的方式重新定义QUIET：

```
$ make QUIET=
```

调试技术

这一节将会探讨一般的调试技术与相关主题。最后你会觉得，调试就好像是一个装了各种你需要的东西的幸运袋。这些技术对我来说都很实用，即使是最简单的*makefile*问题，我也是靠着它们来进行调试的，或许它们也能协助你。

3.80版中一个非常恼人的缺陷是，当make汇报*makefile*中的问题时还会包含一个行号，我发现那个行号通常是错的。我并未调查出是否此问题是由于引入文件、多行变量赋值或用户自定义宏的关系，但是它的确是存在的。make所汇报的行号通常会比实际的行号还大，在复杂的*makefile*中，我发现行号差了20行之多。

通常，查看make变量值的最简单方法，就是在工作目标的执行期间输出它。尽管使用*warning*加入输出语句很简单，而为了在长期运行中节省时间你会想要加入通用的debug工作目标，但是必须多费一番工夫。下面是一个简单的debug工作目标：

```
debug:
        $(for v,$(V), \
          $(warning $v = $($v)))
```

要使用此功能，只需要在命令行上将一份需要输出的变量的列表赋值给变量V以及指定debug工作目标：

```
$ make V="USERNAME SHELL" debug
makefile:2: USERNAME = Owner
makefile:2: SHELL = /bin/sh.exe
make: debug is up to date.
```

如果你觉得这样很麻烦，只要使用MAKECMDGOALS就可以避免对变量V进行赋值的动作：

```
debug:
        $(for v,$(V) $(MAKECMDGOALS), \
          $(if $(filter debug,$v),,$(warning $v = $($v))))
```

现在，你只需要在命令行上直接指定需要输出的变量即可。但是我并不建议使用这个技术，因为当make的警告信息指出它不知道如何更新变量时（因为它们是以工作目标的形式出现在命令行上的），你可能会产生混淆：

```
$ make debug PATH SHELL
makefile:2: USERNAME = Owner
```

```
makefile:2: SHELL = /bin/sh.exe
make: debug is up to date.
make: *** No rule to make target USERNAME.  Stop.
```

我在第十章曾简单提到过，使用开启调试功能的shell可协助我们了解make在后台所进行的活动。尽管make在执行命令之前会输出命令脚本中的命令，但是它并不会输出shell函数中所执行的命令。通常这些命令是既微妙且复杂的，尤其是因为它们可能会被立即执行或是延后执行（如果它们出现在递归变量中）。查看这些命令如何执行的一个方法，就是要求subshell启用调试的功能：

```
DATE := $(shell date +%F)
OUTPUT_DIR = out-$(DATE)

make-directories := $(shell [ -d $(OUTPUT_DIR) ] || mkdir -p
$(OUTPUT_DIR))

all: ;
```

如果运行时指定了sh的调试选项，我们将会看到：

```
$ make SHELL="sh -x"
+ date +%F
+ '[' -d out-2004-05-11 ']'
+ mkdir -p out-2004-05-11
```

这么做，你不仅可以看到make的警告信息，也可以看到额外的调试信息，因为开启调试功能的shell还会显示变量和表达式的值。

本书所举过的许多范例都用到了嵌套层极深的表达式，比如下面这个用来在Windows/Cygwin系统上检查PATH变量的表达式：

```
$(if $(findstring /bin/,                                  \
       $(firstword                                        \
         $(wildcard                                       \
           $(addsuffix /sort$(if $(COMSPEC),.exe),        \
             $(subst :, ,$(PATH))))))),,                  \
  $(error Your PATH is wrong, c:/usr/cygwin/bin should \
    precede c:/WINDOWS/system32))
```

要对这些表达式进行调试并没有什么好办法。一个可行的办法就是将它们拆开，输出每个子表达式（subexpression）：

```
$(warning $(subst :, ,$(PATH)))
$(warning /sort$(if $(COMSPEC),.exe))
$(warning $(addsuffix /sort$(if $(COMSPEC),.exe),        \
              $(subst :, ,$(PATH))))
$(warning $(wildcard                                      \
            $(addsuffix /sort$(if $(COMSPEC),.exe),       \
              $(subst :, ,$(PATH)))))
```

尽管这有点烦人，但是在没有调试器可用的状况下，这或许是确定各个子表达式值的最好办法（有时是唯一的办法）。

常见的错误信息

3.81版的GNU make在线使用手册列有make的错误信息以及它们产生的原因。我们在此只会介绍若干最常见的错误。此处所提到的问题中的部分并非完全是make的错误，比如命令脚本中的语法错误，但是它们仍然是开发人员常会遇到的问题。至于完整的make错误列表，请参考make在线使用手册。

make所输出的错误信息具有如下的标准格式：

```
makefile:n: *** message. Stop.
```

或：

```
make:n: *** message. Stop.
```

*makefile*部分是发生错误的*makefile*或引入文件的名称。下一个部分是发生错误的行号，接着是三个星号，最后是错误信息。

请注意，make的工作就是运行其他的程序，如果发生错误，即使问题出在你的*makefile*上，也非常可能会让人觉得错误是来自其他程序。例如，shell发生错误有可能是命令脚本形式不正确的结果，编译器发生错误有可能是因为命令行参数不正确。找出错误信息产生自哪个程序，是你解决此问题时所必须进行的第一项工作。幸好，make的错误信息相当具有说明性。

语法错误

这些通常是打字上的错误：漏掉圆括号、以空格代替跳格等。

make的新用户最常会遇到的一个错误，就是漏掉变量名称的圆括号：

```
foo:
        for f in $SOURCES; \
        do                 \
           ....            \
        done
```

这可能会使得make把$S扩展成空无一物，而且shell只会以值为OURCES的f执行循环一次。你可能会看到如下适当的shell错误信息：

```
OURCES: No such file or directory
```

不过也可能看不到任何信息，这取决于你处理 f 的方式。所以，别忘了为你的make变量加上圆括号。

missing separator

如下的错误信息：

```
makefile:2:missing separator. Stop.
```

或：

```
makefile:2:missing separator (did you mean TAB instead of 8 spaces?). Stop.
```

通常代表你的命令脚本以空格代替了跳格。

以文字来解释的话，就是make想要查找一个make分隔符，比如:、=或一个跳格符，但是找不到。它所找到的是它不了解的东西。

commands commence before first target

跳格符的问题又出现了！

此信息首次出现在“分析命令”一节中。当命令脚本之外的文本行以一个跳格符开头时，此错误似乎通常会出现在*makefile*的中间。make将会尽可能消除此模糊不清的状态，但如果该文本行无法被确定为变量赋值、条件表达式或多行宏定义，make就会认为这代表命令放错地方了。

unterminated variable reference

这是一个简单但常见的错误，代表你没有为变量引用或函数调用加上适当数目的右圆括号。当函数调用和变量引用嵌套很多层时，make文件看起来很像Lisp！使用能够检查圆括号是否完整的编辑器，比如Emacs，是避免此类错误最可靠的方法。

命令脚本中的错误

脚本中有三种常见的错误：在多行命令中漏掉一个分号，一个不完整或不正确的路径变量，或是一个“执行时会遇到问题的”命令。

我们已经在“良好的编码习惯”一节中探讨过漏掉分号的问题，所以此处不再做进一步的说明。

当shell无法找到Foo命令时，将会显示如下的典型错误信息：

```
bash: foo: command not found
```

这表示shell已经搜索过PATH变量中的每个变量，但是找不到相符的可执行文件。要修正此错误，你必须更新你的PATH变量，它通常被放在你的*.profile*文件（Bourne shell）、*.bashrc*文件（bash）或*.cshrc*文件（C shell）中。当然，它也有可能设定在*makefile*文件中的PATH变量里，并且从make导出PATH变量。

最后，当shell命令执行失败的时候，它会以非零的结束状态终止执行。在此状况下，make将会以如下的信息汇报此错误：

```
$ make
touch /foo/bar
touch: creating /foo/bar: No such file or directory
make: *** [all] Error 1
```

此处执行失败的命令是touch，它会输出自己的错误信息以说明此状态。下一行是make的错误摘要。执行失败的*makefile*工作目标会被显示在中括号里，后面还会跟着运行失败的程序的结束值。如果程序结束运行是因为信号的缘故，make将会输出比较详细的信息，而不会只显示非零的结束状态。

并请注意，因为@修饰符而安静执行的命令也会执行失败。在此状况下，所显示的错误信息好像到处都是。

不管是以上哪种状况，错误信息皆来自make所运行的程序，而不是make本身。

No Rule to Make Target

此信息有两种形式：

```
make: *** No rule to make target XXX. Stop.
```

以及：

```
make: *** No rule to make target XXX, needed by YYY. Stop.
```

这代表make判断文件*XXX*需要更新，但是make找不到执行此工作的任何规则。在放弃和输出此信息之前，make将会在它的数据库中搜索所有的隐含和具体规则。

此项错误的理由可能有三个：

- 你的*makefile*漏掉了更新此文件所需要的一个规则。在此状况下，你必须加入描述如何建立此工作目标的规则。
- 在*makefile*中打错了字。不是make找错了文件，就是更新此文件的规则指定了错

误的文件。因为make变量的使用，你很难在*makefile*中发现打错字的问题。有时候，要确定复杂文件名的值是否正确唯有将它输出：你可以直接输出变量，或是查看make的内部数据库。

- 这个文件应该存在，但是make就是找不到它，可能是因为把它漏掉了，或是因为make不知道要到哪里找它。当然，有时make是绝对正确的，文件缺失的原因或许是你忘了将它从CVS调出。较常见的状况是，make找不到源文件只是因为文件放错地方了。有时是因为源文件放在独立的源文件树中，或是文件产生自另一个程序且所产生的文件放在二进制文件树中。

Overriding Commands for Target

make只允许一个工作目标拥有一个命令脚本（双冒号规则除外，但是很少使用）。如果一个工作目标被指定了一个以上的命令脚本，make将会输出如下的警告信息：

```
makefile:5: warning: overriding commands for target foo
```

它也可能会显示如下的警告信息：

```
makefile:2: warning: ignoring old commands for target foo
```

第一个警告信息指出，make在第5行找到了第二个命令脚本；第二个警告信息指出，位于第2行的最初命令脚本被覆盖掉了。

在复杂的*makefile*中，一个工作目标通常会被定义许多次，每一次都会加入它自己的必要条件。这些工作目标中通常会有一个被指定命令脚本，但是在开发或调试期间，你很容易会加入另一个命令脚本而忘记这么做会覆盖掉现有的命令脚本。

例如，我们可能会在一个引入文件中定义一个通用的工作目标：

```
# 建立一个 jar 文件。
$(jar_file):
        $(JAR) $(JARFLAGS) -f $@ $^
```

这使得其他的*makefile*可以加入自己的必要条件。然后我们可能会在某个*makefile*文件中这么做：

```
# 为 jar 的建立设定工作目标并且加入必要条件
jar_file = parser.jar
$(jar_file): $(class_files)
```

如果我们不小心将一个命令脚本加入此*makefile*，make可能会产生overriding的警告信息。

第三部分

附录

最后一个部分所包含的信息并非本书的重点所在，不过在某些特殊的状况下你会发现它很有用。附录一列出了Gnu make的命令行选项。附录二把make作为一般用途的程序语言来看，探讨数据结构与算术运算，你将会发现它很有趣而且可能很有用。

附录一

运行make

GNU make拥有一组令人印象深刻的命令行选项。大部分的命令行选项都包含了短和长两种形式。短式选项就是一个破折号后面跟着一个字母，而长式选项就是双破折号之后跟着被破折号隔开的单词。它们的语法分别如下所示：

```
-o argument
--option-word=argument
```

接下来所要介绍的是make最常用的选项。完整的列表请参考GNU make在线手册，或是键入make --help。

--always-make

-B

假设每个工作目标皆尚未更新，都需要执行更新的动作。

--directory=*directory*

-C *directory*

在搜索*makefile*或执行任何工作之前将当前目录变更为所指定的目录。这也会将CURDIR变量设定为*directory*。

--environment-overrides

-e

当这个选项被使用时，环境变量的值将会覆盖*makefile*文件中同名变量的值。在*makefile*中针对特定的变量使用override指令可覆盖掉这个命令行选项。

--file=*makefile*

-f *makefile*

把所指定的文件（而不是默认的文件名，即*makefile*、*Makefile*或*GNUMakefile*）作为*makefile*来读。

--help

-h

为命令行选项输出一份简单的摘要。

--include-dir=*directory*

-I *directory*

如果在当前目录找不到引入文件，在搜索编译进make里的搜索路径之前，先到所指定的目录中找找看。--include-dir选项可以在命令行上指定任意多次。

--keep-going

-k

即使有一个命令返回错误状态也不要终止make进程，只是跳过当前工作目标的其余部分，并且继续进行其他工作目标的处理。

--just-print

-n

显示make将会执行哪些命令，但不会执行命令脚本中的任何命令。当你想在make实际工作之前先了解它会执行哪些命令，这个选项非常有用。但请注意，这个选项只能避免命令脚本中命令的执行动作，shell函数中的命令就没办法了。

--old-file=*file*

-o *file*

让*file*的时间戳变成在任何文件（的时间戳）之前，以及对需要更新的工作目标采取适当的动作。如果一个文件的时间戳被意外变更或是会影响依存图中的某个必要条件，这个选项将会很有用。这个选项用来对--new-file（-W）选项的不足之处进行补充。

--print-data-base

-p

输出make的内部数据库。

--touch

-t

对每个尚未更新（out-of-date）的工作目标运行touch程序，以便更新它的时间戳。指定这个选项可以让依存图中的所有文件变成已更新的。例如，一个核心头文件的注释被变更，可能会使得make非必要地重新编译大量源代码。要避免重新编译浪费机器的时间，你可以使用--touch选项让所有文件变成已更新的。

--new-file=*file*

-W *file*

让*file*的时间戳变成在任何工作目标（的时间戳）之后。指定这个选项，不必编

辑文件或变更文件的时间戳，就可以迫使工作目标进行更新的动作。这个选项用来对 `--old-file` 选项的不足之处进行补充。

`--warn-undefined-variables`

如果有一个未定义的变量被扩展就会输出一个警告信息。这是一个非常有用的诊断工具，因为未定义的变量会被安静地扩展成空无一物。然而，基于自定义的目的在 *makefile* 中使用空变量也很常见。这个选项同样也会汇报任何未设定的自定义变量。

附录二

越过 make 的极限

如你所见，GNU make 可以做若干令人难以置信的事情，但是能够以eval结构让make 3.80 越过极限的例子并不多见。在接下来的例子中，我们要来看看是否可以越过极限。

数据结构

make的一个极限是，当我们所编写的复杂*makefile*用到了make所欠缺的数据结构能力时，make 偶尔也会变成废物。一个极为有限的做法就是，你可以将变量定义成内嵌的点号（或是 ->，如果你坚持的话）的形式来模拟一个数据结构：

```
file.path = /foo/bar
file.type = unix
file.host = oscar
```

如果通过测试，你甚至还可以通过“经求值的变量”（computed variable）将这个 file 结构传递给一个函数：

```
 define remote-file
  $(if $(filter unix,$($1.type)), \
    /net/$($1.host)/$($1.path), \
    //$($1.host)/$($1.path))
endef
```

然而，这似乎是一个无法令人满意的解决方案，理由如下：

- 你无法轻易地为此“结构”分配一个实例（instance）。创建一个新的实例涉及以下动作：选择新变量的名称，以及对每个元素进行赋值。这也意味着，不保证这些假实例（pseudo-instance）具有相同的字段（称为槽）。
- 此结构只存在于用户的心中，它们是一组不同的make变量，而不是一个具有自己名称的统一实体（unified entity）。而且因为此结构没有名称，你很难建立一个指

向此结构的引用（或指针），所以将它们作为参数来传递或存储到变量中是个愚蠢的行为。

- 没有万无一失的方法可用来访问结构的一个槽。变量名称中任何一个部分的打字错误，将会产生错误的值（或是没有值），而且不会从make获得警告信息。

不过，remote-file函数暗示了一个更广泛的解决方案。假设我们以经求值的变量来实现结构实例。早期的Lisp对象系统（甚至今天的某些系统）使用了类似的技术。一个结构，比如file-info，可以具有以符号名称来表示的实例，比如file_info_1。

另一个实例可能名为file_info_2。此结构的槽可以用经求值的变量来表示：

```
file_info_1_path
file_info_1_type
file_info_1_host
```

因为实例具有符号名称，所以它可以被存入一个或多个变量（通常，程序员的选择是使用递归变量或简单变量）：

```
before_foo = file_info_1
another_foo = $(before_foo)
```

file-info的元素可以使用Lisp-like的取值函数（getter）和设值函数（setter）：

```
path := $(call get-value,before_foo,path)
$(call set-value,before_foo,path,/usr/tmp/bar)
```

我可以进一步地为file-info结构创建一个模板，方便新实例的分配：

```
orig_foo := $(call new,file-info)
$(call set-value,orig_foo,path,/foo/bar)

tmp_foo := $(call new,file-info)
$(call set-value,tmp_foo,path,/tmp/bar)
```

现在，file-info具有两个不同的实例。最后，我们可以为槽加入默认值的概念。所以，我们可以这样来声明file-info结构：

```
$(call defstruct,file-info, \
  $(call defslot,path,/, \
  $(call defslot,type,unix), \
  $(call defslot,host,oscar))
```

defstruct函数的第一个参数是结构的名称，后面跟着一串defslot调用。每个defslot调用中包含了一个槽名与默认值（*name,default value*）的配对。从例B-1中可以看到defstruct以及支持程序代码的实现方式：

例 B-1：make 中的结构定义

```
# $(next-id) —— 返回一个独一无二的编号
next_id_counter :=
define next-id
$(words $(next_id_counter))$(eval next_id_counter += 1)
endef

# all_structs——已定义结构的列表
all_structs :=

value_sep := XxSepxX

# $(call defstruct, struct_name, $(call defslot, slot_name, value), ...)
define defstruct
  $(eval all_structs += $1)                                          \
  $(eval $1_def_slotnames :=)                                        \
  $(foreach v, $2 $3 $4 $5 $6 $7 $8 $9 $(10) $(11),                  \
    $(if $($v_name),                                                 \
      $(eval $1_def_slotnames          += $($v_name))                \
      $(eval $1_def_$($v_name)_default := $($v_value))))
endef

# $(call defslot,slot_name,slot_value)
define defslot
  $(eval tmp_id := $(next_id))
  $(eval $1_$(tmp_id)_name := $1)
  $(eval $1_$(tmp_id)_value := $2)
  $1_$(tmp_id)
endef

# all_instances——将任何结构的所有实例列在此处
all_instances :=

# $(call new, struct_name)
define new
$(strip                                                              \
  $(if $(filter $1,$(all_structs)),,                                 \
    $(error new on unknown struct '$(strip $1)'))                    \
  $(eval instance := $1@$(next-id))                                  \
  $(eval all_instances += $(instance))                               \
  $(foreach v, $($(strip $1)_def_slotnames),                         \
    $(eval $(instance)_$v := $($(strip $1)_def_$v_default)))         \
  $(instance))
endef

# $(call delete, variable)
define delete
$(strip                                                                  \
  $(if $(filter $($(strip $1)),$(all_instances)),,                       \
    $(error Invalid instance '$($(strip $1))'))                          \
  $(eval all_instances := $(filter-out $($(strip $1)),$(all_instances))) \
  $(foreach v, $($(strip $1)_def_slotnames),                             \
    $(eval $(instance)_$v := )))
endef
```

```
# $(call struct-name, instance_id)
define struct-name
$(firstword $(subst @, ,$($(strip $1))))
endef

# $(call check-params, instance_id, slot_name)
define check-params
  $(if $(filter $($(strip $1)),$(all_instances)),,                  \
    $(error Invalid instance '$(strip $1)'))                        \
  $(if $(filter $2,$($(call struct-name,$1)_def_slotnames)),,       \
    $(error Instance '$($(strip $1))' does not have slot '$(strip $2)'))
endef

# $(call get-value, instance_id, slot_name)
define get-value
$(strip                                 \
  $(call check-params,$1,$2)            \
  $($($(strip $1))_$(strip $2)))
endef

# $(call set-value, instance_id, slot_name, value)
define set-value
  $(call check-params,$1,$2) \
  $(eval $($(strip $1))_$(strip $2) := $3)
endef

# $(call dump-struct, struct_name)
define dump-struct
{ $(strip $1)_def_slotnames "$($(strip $1)_def_slotnames)"          \
  $(foreach s,                                                      \
    $($(strip $1)_def_slotnames),$(strip                            \
    $(strip $1)_def_$s_default "$($(strip $1)_def_$s_default)")) }
endef

# $(call print-struct, struct_name)
define print-struct
{ $(foreach s,                                                      \
    $($(strip $1)_def_slotnames),$(strip                            \
    { "$s" "$($(strip $1)_def_$s_default)" })) }
endef

# $(call dump-instance, instance_id)
define dump-instance
{ $(eval tmp_name := $(call struct-name,$1))                        \
  $(foreach s,                                                      \
    $($(tmp_name)_def_slotnames),$(strip                            \
    { $($(strip $1))_$s "$($($(strip $1))_$s)" })) }
endef

# $(call print-instance, instance_id)
define print-instance
{ $(foreach s,                                                      \
    $($(call struct-name,$1)_def_slotnames),"$(strip                \
    $(call get-value,$1,$s))") }
endef
```

让我们一次查看一段程序代码。首先看到的是next-id函数的定义，这是一个简单的计数器：

```
# $(next-id) ——返回一个独一无二的数字
next_id_counter :=
define next_id
$(words $(next_id_counter))$(eval next_id_counter += 1)
endef
```

许多人认为make无法进行算术运算，因为这个语言的限制太大。一般来说，的确是如此，不过像这样的极限测试通常可以求得你想要的结果。此函数会使用eval重新定义一个简单变量的值。此函数包含了两个表达式：第一个表达式会返回next_id_counter中的单词数目；第二个表达式会将另一个单词附加到该变量。它并不是非常有效率，但是当数字在数千时它还是很好的。

下一段是定义defstruct函数本身以及创建支持的数据结构。

```
# all_structs——已定义结构的列表
all_structs :=

value_sep := XxSepxX

# $(call defstruct, struct_name, $(call defslot, slot_name, value), ...)
define defstruct
  $(eval all_structs += $1)                                  \
  $(eval $1_def_slotnames :=)                                \
  $(foreach v, $2 $3 $4 $5 $6 $7 $8 $9 $(10) $(11),          \
    $(if $($v_name),                                         \
      $(eval $1_def_slotnames          += $($v_name))        \
      $(eval $1_def_$($v_name)_default := $($v_value)))))
endef

# $(call defslot,slot_name,slot_value)
define defslot
  $(eval tmp_id := $(next_id))
  $(eval $1_$(tmp_id)_name := $1)
  $(eval $1_$(tmp_id)_value := $2)
  $1_$(tmp_id)
endef
```

all_structs变量是一份已（被defstruct）定义结构的列表。这份列表让new函数能够对它所分配的结构进行类型检查。对每个结构来说，S是defstruct函数所定义的一组变量：

```
S_def_slotnames
S_def_slotn_default
```

第一个变量为一个结构定义了一组槽，第二个变量为每个槽定义默认值。defstruct函数的前两行会对all_structs进行附加的动作以及对槽列表进行初始化的动作。函数的其余部分会反复处理槽，以便建立槽列表并存入默认值。

每个槽的定义由defslot来处理。此函数首先会分配一个id，然后将槽名和值存入两个变量，最后返回前缀。所返回的前缀可让defstruct的参数列表成为一串简单的符号，符号列表中每个元素皆可用来访问每个槽的定义。如果有更多属性被加到槽之后，要将他们并入defslot相当简单。这个技术还让默认值能够比“较简单的替代方案”具有更大的值的范围（包括空格）。

defstruct中的foreach循环决定了槽数的上限。这个版本允许你使用10个槽。foreach的主体处理每个参数的方式，就是将槽名附加到$_def_slotnames中，以及把默认值赋值给一个变量。例如，我们的file-info结构将会被定义成：

```
file-info_def_slotnames := path type host
file-info_def_path_default :=
file-info_def_type_default := unix
file-info_def_host_default := oscar
```

于是就完成了结构的定义。

现在我们已经可以定义结构了，我们还需具备为结构创建实例的能力。new函数就是用来执行此操作的：

```
# $(call new, struct_name)
define new
$(strip
  $(if $(filter $1,$(all_structs)),,
    $(error new on unknown struct '$(strip $1)'))                    \
  $(eval instance := $1@$(next-id))                                  \
  $(eval all_instances += $(instance))                               \
  $(foreach v, $($(strip $1)_def_slotnames),                         \
    $(eval $(instance)_$v := $($(strip $1)_def_$v_default)))         \
  $(instance))
endef
```

首先，函数里的if会检查所指定的名称是否为已知的结构名称。如果在all_structs中找不到相符的名称，就会发出错误信息。接着，我们会为新的实例构造独一无二的id（将结构的名称与独一无二的整数后缀衔接在一起）。我们在结构名称与后缀之间使用了一个@符号，所以稍后我们可以轻易地将它们分开。new函数接着会记录新实例的名称，以便稍后让访问函数（accessor）进行类型检查，然后会使用默认值来初始化结构的槽。这个进行初始化的程序代码值得介绍一下：

```
$(foreach v, $($(strip $1)_def_slotnames),                           \
  $(eval $(instance)_$v := $($(strip $1)_def_$v_default)))
```

foreach循环会反复处理结构的槽名。循环的主体会对结构的名称使用strip，这让用户能够在call new的逗号之后加入空格。别忘了，每个槽可以被表示成：将实例名称与槽名衔接在一起（例如file_info@1_path）。赋值运算符右边部分是默认值（可以从结构名称与槽名求得）。最后，此函数会返回实例的名称。

请注意，尽管我称这些“语法结构”为函数，但是它们实际上是宏。也就是说，解析的时候，new这个符号会被递归地扩展成一段新的文本插入*makefile*。defstruct宏之所以能够完成我们想要的功能，主要是因为它被内置在eval调用中，会被扩展成空无一物。同样地，new宏也会在eval调用里完成它的重要工作。其实将它称为函数也是合理的，因为将它扩展之后逻辑上会产生一个值，就是用来表示新实例的符号。

接下来，我们需要具备在结构中取值和设值的能力。为提供此功能，我们定义了两个新函数：

```
# $(call get-value, instance_id, slot_name)
define get-value
$(strip                                  \
  $(call check-params,$1,$2)             \
  $($($(strip $1))_$(strip $2)))
endef

# $(call set-value, instance_id, slot_name, value)
define set-value
  $(call check-params,$1,$2) \
  $(eval $($(strip $1))_$(strip $2) := $3)
endef
```

要取得槽的值，我们只需对来自“实例id与槽名”的槽变量名称进行求值的动作即可。求值之前我们可以先使用check-params函数来检查实例id和槽名是否为有效字符串以改进安全性。为了提升编排的美感以及确保额外的空格不会破坏槽值，我们会将这些参数封装在strip调用中。

set函数也会在设值之前检查参数。同样地，我们会用strip来处理这两个函数参数，让用户能够随意地在参数列表中加入空格。请注意，我们并没有对槽值使用strip，因为用户可能真的需要用到空格。

```
# $(call check-params, instance_id, slot_name)
define check-params
  $(if $(filter $($(strip $1)),$(all_instances)),,                  \
    $(error Invalid instance '$(strip $1)'))                        \
  $(if $(filter $2,$($(call struct-name,$1)_def_slotnames)),,       \
    $(error Instance '$($(strip $1))' does not have slot '$(strip $2)'))
endef

# $(call struct-name, instance_id)
define struct-name
$(firstword $(subst @, ,$($(strip $1))))
endef
```

check-params函数只是用来检查传递给设值和取值函数的实例id是否被包含在已知的实例列表中。同样地，它会检查槽名是否被包含在隶属此结构的槽列表中。结构的名称

可以从实例名称（一个被@符号隔开的符号）中获得并取出第一个单词。这意味着结构名称中不可以包含@符号。

最后，我们可以在实现中加入一对输出函数（print-instance与print-struct）和转储函数（dump-instance与dump-struct）。输出函数可以把结构定义和结构实例显示成用户看得懂的形式，然而转储函数所显示的是结构定义和结构实例的实现细节。详情参见例B-1。

在下面的例子中，你可以看到我们定义和使用 file-info 结构的方法：

```
include defstruct.mk

$(call defstruct, file-info,     \
  $(call defslot, path,),        \
  $(call defslot, type,unix),    \
  $(call defslot, host,oscar))

before := $(call new, file-info)
$(call set-value, before, path,/etc/password)
$(call set-value, before, host,wasatch)

after := $(call new,file-info)
$(call set-value, after, path,/etc/shadow)
$(call set-value, after, host,wasatch)

demo:
        # before        = $(before)
        # before.path   = $(call get-value, before, path)
        # before.type   = $(call get-value, before, type)
        # before.host   = $(call get-value, before, host)
        # print before = $(call print-instance, before)
        # dump before   = $(call dump-instance, before)
        #
        # all_instances    = $(all_instances)
        # all_structs      = $(all_structs)
        # print file-info = $(call print-struct, file-info)
        # dump file-info   = $(call dump-struct, file-info)
```

下面是它的运行结果：

```
$ make
# before        = file-info@0
# before.path   = /etc/password
# before.type   = unix
# before.host   = wasatch
# print before = { "/etc/password" "unix" "wasatch" }
# dump before   = { { file-info@0_path "/etc/password" } { file-info@0_type "unix" }
{ file-info@0_host "wasatch" } }
#
# all_instances    =  file-info@0 file-info@1
# all_structs      =  file-info
# print file-info = { { "path" "" } { "type" "unix" } { "host" "oscar" } }
# dump file-info   = { file-info_def_slotnames " path type host" file-info_def_path_
default "" file-info_def_type_default "unix" file-info_def_host_default "oscar" }
```

此外，注意非法的结构使用是如何受到限制的：

```
$ cat badstruct.mk
include defstruct.mk
$(call new, no-such-structure)
$ make -f badstruct.mk
badstruct.mk:2: *** new on unknown struct 'no-such-structure'.  Stop.

$ cat badslot.mk
include defstruct.mk
$(call defstruct, foo, defslot(size, 0))
bar := $(call new, foo)
$(call set value, bar, siz, 10)
$ make -f badslot.mk
badslot.mk:4: *** Instance 'foo@0' does not have slot 'siz'.  Stop.
```

当然，例 B-1 还有许多可以改进的地方：

- 对槽赋值进行确认检查。你可以使用一个挂钩函数来实现此功能，该函数必须在赋值动作完成之后产生空值。此挂钩的用法会像这个样子：

```
# $(call set value, instance_id, slot_name, value)
define set-value
  $(call check-params,$1,$2)                                  \
  $(if $(call $(strip $1)_$(strip $2)_hook, value),           \
    $(error set-value hook, $(strip $1)_$(strip $2)_hook, failed)) \
  $(eval $($(strip $1))_$(strip $2) := $3)
endef
```

- 支持继承的功能。defstruct 的参数可以接受另一个 defstruct 名称而成为一个超类（superclass），子类（subclass）可以复制超类的所有成员（member）。
- 对结构引用提供较好的支持。使用当前的实现，一个槽可以保存另一个结构的ID，但是访问很麻烦。新版的 get-value 函数应该能够检查引用（通过查找 *defstruct@number*）以及进行自动解除引用。

算术运算

在前一节我提到过，只使用make的原生功能是无法在make中进行算术运算的。接着，我还提到，如何通过“把单词添加到列表以及返回列表的长度”的方式来实现一个简单的计数器。然后，我找到了递增数字的技巧，Michael Mounteney 为“在 make 中以有限的形式进行整数的加法”提供了一个很酷的技巧。

他的技巧是运用一行数字来计算两个整数（值为1或更大）的总和。工作原理是这样的，假设你有一行这样的数字：

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

现在，请注意（如果我们所使用的索引值没问题的话），我们可以做加法运算，例如4加5，首先从这行数字取出一个子集（范围从第4个元素到最后一个元素），然后从子集中选出第5个元素。我们可以使用原生的make函数来完成此功能：

```
number_line := 2 3 4 5 6 7 8 9 10 11 12 13 14 15
plus = $(word $2, $(wordlist $1, 15, $(number_line)))
four+five = $(call plus, 4, 5)
```

Michael真聪明！注意，这行数字是从2开头而不是从0或1。如果你是以1和1来运行plus函数，你就会发现必须这么做。这两个索引值最后将会取得第一个元素，并且答案必须是2，因此数字列表的第一个元素必须是2。所以，对word和wordlist函数来说，列表中第一个元素的索引值是1不是0（但是我们并不想证明此事）。

现在，有了这行数字之后，我们可以进行加法运算，但是我们如何在不手动键入或使用shell程序的状况下创建这行数字呢？我们可以通过将“在一处的所有可能值”与“在十处的所有可能值”组合在一起来创建从00到99间的所有数字。例如：

```
make -f - <<< '$(warning $(foreach i, 0 1 2, $(addprefix $i, 0 1 2))))'
/c/TEMP/Gm002568:1: 00 01 02 10 11 12 20 21 22
```

只要纳入0到9间的所有数字，我们就可以产生00到99间的所有数字。通过再次组合foreach与这100个数字（译注1），我们可以产生000到999间的所有数字，等等。最后还必须除去数字中前导的零。

我们将Mounteney先生用来产生一行数字的程序代码做了如下的修改，并且定义了plus和gt运算：

```
# combine——将一串数字与另一串数字衔接在一起
combine = $(foreach i, $1, $(addprefix $i, $2))

# stripzero——从每个单词中移除一个前导的零
stripzero = $(patsubst 0%,%,$1)

# generate——从单词列表产生出三个元素的所有的数字排列
generate = $(call stripzero,                          \
             $(call stripzero,                        \
               $(call combine, $1,                    \
                 $(call combine, $1, $1))))

# number_line——创建从0到999间的所有数字
number_line := $(call generate,0 1 2 3 4 5 6 7 8 9)
length      := $(word $(words $(number_line)), $(number_line))
```

译注1：`$ make -f - <<< '$(warning $(foreach j, 0 1 2 3 4 5 6 7 8 9, $(addprefix $j, $(foreach i, 0 1 2 3 4 5 6 7 8 9, $(addprefix $i, 0 1 2 3 4 5 6 7 8 9)))))'`

```
# plus——通过一行数字来进行两个整数的加法运算
plus = $(word $2,                                          \
         $(wordlist $1, $(length),                         \
           $(wordlist 3, $(length), $(number_line))))

# gt——通过一行数字来判断 $1 是否大于 $2
gt = $(filter $1,                                          \
       $(wordlist 3, $(length),                            \
         $(wordlist $2, $(length), $(number_line))))

all:
        @echo $(call plus,4,7)
        @echo $(if $(call gt,4,7),is,is not)
        @echo $(if $(call gt,7,4),is,is not)
        @echo $(if $(call gt,7,7),is,is not)
```

当我们运行此 *makefile* 时将会产生如下的结果：

```
$ make
11
is not
is
is not
```

我们可以做进一步的延伸，像是通过索引一个颠倒的数字列表来纳入减法功能，就像倒着数一样。例如，计算7减4，首先创建范围0到6的数字列表子集，将它颠倒过来，然后取出第4个元素：

```
number_line := 0 1 2 3 4 5 6 7 8 9...
1through6   := 0 1 2 3 4 5 6
reverse_it  := 6 5 4 3 2 1 0
fourth_item := 3
```

让我们以make的语法来表示这个算法：

```
 # backwards——一个颠倒的数字行
backwards := $(call generate, 9 8 7 6 5 4 3 2 1 0)

# reverse——将一串单词颠倒过来
reverse    = $(strip                                       \
               $(foreach f,                                \
                 $(wordlist 1, $(length), $(backwards)), \
                 $(word $f, $1)))

# minus——计算 $1 减 $2
minus      = $(word $2,                                    \
               $(call reverse,                             \
                 $(wordlist 1, $1, $(number_line))))

minus:
        # $(call minus, 7, 4)
```

至于乘法和除法则留给读者自己练习。

索引

前言

本索引让读者方便查找特定词汇在本书里的页码，以快速找到所需的内容。传统上，中文书没有编制索引的惯例，因为中文不像英文那样有一套公认而且大家都知道的排序规则（字母顺序）。然而，这是一本工具书，为了方便读者查阅，索引是不可或缺的。为了兼顾“查阅”与“中文化”，我们决定沿袭英文版的编排格式，也就是依照英文字母顺序排列所有条目，所以，我们保留所有原文，如此读者才能快速找到想找的条目，而中文是以辅助说明的方式出现的。

然而，使用英文字母顺序编排意味着读者必须先知道英文词汇原文才能顺利找到其所在的页码。例如，假设读者想知道本书哪几页提到了“域名服务”，那么，你必须先知道其原文是“Domain Name Service”，或是知道其缩写是“DNS”，然后才能推断此条目应该是编在“D”小节。如果读者觉得这样不方便，我们为此感到抱歉，因为我们还没有找到一套大家都公认的中文排序规则，如果我们像编字典那样使用首字笔划顺序来排列，除了不方便查找之外，还必须面临一词多译的问题。例如，有人习惯将“serial port”翻译成“串行端口”，但也有人将它翻译成“顺序端口”，如果你不知道本书采用哪一种译词，要第一次就顺利找到“serial port”的页码，唯一的办法是碰运气。如果运气不好，你必须同时知道 serial port 的每一种可能译法才有机会找到。

格式说明

所有条目都是依照英文字母顺序排列。举一个实例说明如何使用本篇索引以及索引条目

的编排格式。如果你想知道本书哪几页提到“performance tuning（性能调整）”，则必须先翻到“P”小节，然后你会看到：

performance tuning（性能调整）
    basic step（基本步骤），97
    capacity planning（容量规划），107-111
    External（外部的），102
    Internal（内部的），103

如你所见，我们以缩排格式来表示各项信息。通常，从最左侧逐层往右读，可以得到一个符合文法的完整句子或词汇，不过，并非每次都能这样，有时候你得到的只是特定概念的关键词而已，例如“performance tuning，基本步骤”。所以，这段索引是这样解读的：第97页提到了“性能调整的基本步骤”，在103页提到了“内部的性能调整”。

最后，在不妨碍查阅的前提下，对于第二层与第三层的条目，我们会予以中文化。

希望本节的说明有助于读者使用本篇索引。如果读者对本公司书籍的索引方式有任何意见，请用E-mail告诉我们，好让我们知道如何改进。

符号

[ ] (brackets) wildcard（中括号通配符），21
{ } (curly braces) in variables（用于变量的花括号），52
( ) (parentheses) in variables（用于变量的圆括号），52
+= (append) operator（附加运算符），56
* (asterisk) wildcard（星号通配符），21
$% 自动变量，26
$+ 自动变量，26
$< 自动变量，26
$? 自动变量，26
$@ 自动变量，26
$^ 自动变量，26
$* 自动变量，26
- (dash) command prefix（破折号命令前缀），105
^ in patterns（模式中的^符号），21
:= 运算符，54
?= 运算符
    附带条件的变量赋值运算符，55
    环境变量，63
% (percent) character pattern rules（模式规则中的百分比字符），33
+ (plus) command modifier（加号命令修饰符），105
? (question mark) wildcard（问号通配符），21
@ sign（@符号）
    command prefix, echo and（命令前缀与输出），104
    效能与……，200
~ (tilde) wildcard（~通配符），21

A

add-manifest 函数，jars，188
addprefix 函数，86
addsuffix 函数，85

all 工作目标，24
ALL_TREES 变量，167
--always-make 选项，261
Ant (Java)，172
    build file（编译文件），172
        mkdir 程序，174
    portability（可移植性），174
    task（任务），173
append operator (+=)(附加运算符)，56
ar command, archive library（ar 命令，程序库），45
archive library（程序库），45
archive members, automatic variables and（自动变量与程序库成员），25
arguments, patterns as built-in functions（内置函数以模式为参数），76
arithmetic, performing with make（以 make 进行算术运算），272
arrays, files array and command-line limits（数组，files 数组与命令行长度限制），115
assert 函数，87
automake 工具，151
automatic variable（自动变量），25，64
    程序库成员，25
    空工作目标与……，25
    必要条件，25
    工作目标与……，25
    VPATH 与……，29
    vpath 与……，29

B

basename 函数，84
bash shell benchmarking and（基准测试与 bash shell），195
benchmarking（基准测试），194
    bash shell，195
    Cygwin 与……，196
    subst 函数调用与……，197
    variable assignment speed（变量赋值的速度），197
binary tree（二进制文件树）
    ALL_TREES 变量，167
    文件系统布局与……，138
    多个，166
    搜索，168
    独立的，137
    源文件树的分离，153-161
book makefile（本书的 makefile），208-217
    m4 宏与……，222
    输出的产生，225
    源文件的确认，228
    对 XML 进行预处理，222
bottleneck（瓶颈），198
    dependencies and（依存关系与……），199
build files, XML（编译文件，XML 格式），172
build targets, recursive make and（递归式 make 与建立工作目标），124
build-classpath 函数，185
builds（编译结果）
    automation（自动），140
    Linux 内核的 makefile，230
    testing（测试），140
built-in function（内置函数），75-91
    文件名与……，83
    流程控制与……，86
    以模式为参数，76
    字符串函数，76
    语法，75
built-in rule（内置规则），33
    隐含规则，36
    变量，38
    （另见 rule）

C

-C 目录选项，261
call 函数，74
calling functions, wildcards and（通配符与调用函数），21
canned sequences（见 macros）
case-sensitivity in variables（变量名称中的大小写有别），52
chaining rule（链接规则），33
    intermediate file and（中间文件与……），38
character classes, wildcards and（通配符与字符集），21
characters in variables（变量名称中允许使用的字符），52
check 工作目标，24
circular references, library（循环引用，程序库），50
circularity, library（循环引用，程序库），50
CLASSPATH 变量，Java makefile，183
clean 工作目标，24

code（makefile 的代码）
debugging（调度）
defensive coding（具保护功能的代码），250
writing for（为……编写），248
duplicating, recursive make and（递归式 make 与重复），126
command script（命令脚本），18
error message（错误信息），255
evaluation（求值），111
parsing and（分析与……），101
（另见 script），18
command-line（命令行）
debugging option（调试选项），242
limit（长度限制），112
echo 命令行，113
Linux 内核的 makefile 选项，230
recursive make and（递归式 make 与……），123
subshells，108
targets as arguments（将工作目标指定为参数），12
variable definition and（变量定义与……），62
command（命令）
ar (archive library)，45
awk, phony targets and（假想工作目标与 awk），23
df, phony targets and（假想工作目标与 df），23
empty（空），109
environment（环境），110
file descriptor（文件描述符），110
MAKEFLAGS 变量，110
MAKELEVEL 变量，110
errors in, ignoring（忽略命令中的错误），105
help，40
long（太长的），102
make, execution order（make 的执行顺序），14
makefile, execution order（makefile 的执行顺序），14
multiline（多行），107
no-op，107
option, error and（错误与选项），106
parsing（分析），100
command script and（命令脚本与……），101
editors and（编辑器与……），102
prefix（前缀）
@，104
-（破折号），105
+（加号），105
shell, sequence（shell 序列），102
status code（状态码），105
targets, overriding（工作目标，覆盖掉），256
variable expansion（变量扩展），60
comment（注释），18
commercial development model（商业开发模型），136
compile-bean function (Java)，190
computed variables, assert function and（assert 函数与经求值的变量），87
Concurrent Version Systems（见 CVS）
conditional directive（条件指令）
ifdef，66
ifeq，66
ifndef，66
ifneq，66
syntax（语法），65
conditional processing directive（条件处理指令），65
conditional variable assignment operator (?=)（附带条件的变量赋值运算符），55
configuring Linux kernel makefile（配置 Linux 内核的 makefile），232
constant, user-customized variable（常数，用户自定义变量），53
cookie，24
（另见 empty target）
counter 函数，27
CPPFLAGS 变量，61
CURDIR 变量，68
curly brace ({ }) in variable（用于变量的花括号），52
CVS (Concurrent Version Systems)
binary files, large（大量的二进制文件），169
implicit rules and（内定规则与……），39
cwd (current working directory), CURDIR variable and（CURDIR 变量与当前工作目录），68
Cygwin，73
benchmarking and（基准测试与……），196
conflict（相抵触），146
line termination（行终止符），143

portability and（可移植性与……），143
regtool，184
Windows filesystem and（Windows 文件系统与……），144

D

data structure（数据结构），264
illegal（非法），272
trapping（受到限制），272
--debug 选项，246
debug-enter 变量，92
debugging（调试）
code writing and（makefile 的代码与……），248
command script error message（命令脚本错误信息），255
command-line option（命令行选项），242
defensive coding and（具保护功能的代码与……），250
error message（错误信息），253
introduction（介绍），241
macro and（宏与……），92
make output, phony targets and（make 的输出，假想工作目标与……），23
overriding command, target（覆盖掉命令、工作目标），256
syntax error（语法错误），254
techniuqes for（……的技术），251
warning 函数与……，241
debug-leave 变量，92
declaration, reusable, library functions（声明，可重复使用的，程序库函数），28
deferred expansion（延后扩展），59
define directive, macro（define 指令，宏），57
.DELETE_ON_ERROR 工作目标修饰符，42
dependency（依存关系），125
bottleneck and（瓶颈与……），199
generating（发生），161
automatically（自动地），42
include 指令与……，67
Java makefile，182
make-depend，164
recursion and（递归与……），122
rule（规则），12
dependency checking（检查依存关系），14
development environment, variable（开发环境，变量），64
development requirement（开发需求），136
development style（开发风格），136
dir 函数，83
directive（指令）
conditional processing（条件处理），65
conditional, syntax（条件，语法），65
define，57
include，65，66
dependency and（依存关系与……），67
optimization（优化），133
vpath，30
directory（目录）
~(tilde) and（波浪符号与……），21
multidirectory project, nonrecursive make and（非递归式 make 与多目录项目），129
distclean 工作目标，24
distributed make（分布式 make），206
distribution, performance and（分布式，效能与……），206
double-colon rule（双冒号规则），51

E

echo command line, limit（echo 命令行，长度限制），113
Eclipse，175
EJB (Enterprise Java Beans)，190
ejb-rule 函数，193
EJBS 变量，192
else 关键字，65
Emacs JDEE，175
embedded period, variable definition and（内嵌的点号，变量定义与……），264
empty command（空命令），109
empty target（空工作目标），24
automatic variable and（自动变量与……），25
encapsulation, Java package（封装，Java 包），171
endef 关键字，57
endif 关键字，65
Enterprise JavaBeans (EJB)，190
environment variables, ?= operator（环境变量，?=运算符），63
--environment-overrides 选项，261
environment（环境）
command（命令），110
file descriptor（文件描述符），110
MAKEFLAGS 变量，110
MAKELEVEL 变量，110

variable definition and（变量的定义与……），62
error 函数，87
error handling, recursive make and（错误处理，递归式 make 与……），124
error message（错误信息）
command script（命令脚本），255
debugging and（调试与……），253
fatal, printing（输出无可挽回的），87
error, command（错误，命令）
ignoring（忽略），105
option（选项），106
eval 函数，94
parsing and（分析与……），95
expanding macro（扩展宏）
deferred versus immediate（延后与立即的差异），59
macro invoked from another（从一个宏调用另一个宏），74
expanding text, foreach function and（foreach 函数与扩展文本），88
expanding variable（扩展变量），58
curly braces and（花括号与……），52
deferred（延后），59
immediate（立即），59
explicit rule（具体规则），19，19-25
empty target and（空工作目标与……），24
phony target（假想工作目标），22
wildcard（通配符），21
exporting variable（导出变量），63

F

fatal error messages（无可挽回的错误信息），87
file descriptor（文件描述符），110
file management（文件管理），146
source tree layout（源文件树的布局），149
–file 选项，261
filename（文件名）
built-in functions and（内置函数与……），83
function（函数）
addprefix，86
addsuffix，85
basename（基本名称），84
dir，83
join，86
notdir，84
suffix（扩展名；后缀），84
wildcard 函数，83
pattern（见 pattern rules）
suffix 函数，84
timestamp and（时间戳与……），82
file（文件）
variable definition and（变量的定义与……），62
wildcard（通配符），21
files array, command-line limits and（files 数组，命令行长度限制与……），114
filesystems layout, binary trees and（二进制文件树与文件系统的布局），138
filter 函数，76
filter-out 函数，77
find command, module definition（将模块定义成 find 命令），134
findstring 函数，77
firstword 函数，80
flow control function（流程控制函数），86
assert，87
error，87
foreach，88
if，86
foreach f 函数，88
free software model development（自由软件开发模型），136
function（函数）
add-manifest，188
addprefix，86
addsuffix，85
assert，87
basename，84
build-classpath，185
built-in（内置），75-91
call，74
filename（文件名），83
flow control（流程控制），86
patterns as argument（以模式为参数），76
string function（字符串函数），76
syntax（语法），75
calling, wildcards and（调用，通配符与……），21
compile-bean，190
counter，27
defslot，269

dir，83
ejb-rule，193
error，87
eval，94
filter，76
filter-out，77
findstring，77
firstword，80
flow control（流程控制），86
foreach，88
generated-source，133
hook（挂钩），98
if，86
join，86
library, declaration（程序库，声明），28
make-library，132
miscellaneous（杂项），90，91
notdir，84
origin，90
parameter, passing to（参数，传递到），98
patsubst，79
remote-file，265
search and replace, string（搜索与替换，字符串），78
shell，80
sort，80
source-to-object，133
space-to-question，148
string function（字符串函数），76
strip，90
whitespace removal（移除空格），66
subst，78
suffix，84
user-defined（用户自定义），72-75
advanced（高级的），91-99
Linux 内核的 makefile，236
parameters and（参数与……），73
value，97
variable（变量），53
warning，91
whitespace manipulation（空格的处理），79
wildcard，83
wildcard-spaces，148
word，80
wordlist，80
words，79

G

generated-source 函数，133
generating dependency（产生依存关系），161
globbing（文件名模式匹配），21
grep command, variables and（grep 命令，变量与……），110

H

header file, include directory（头文件，include 目录），28
Hello World makefile 文件，11
help 命令，40
--help 选项，262
home directory, ~ (tilde) and（波浪符号与主目录），21
hook, function（挂钩，函数），98

I

IDE（Integrated Development Environment，集成开发环境），171，175
if 函数，86
ifdef 条件指令，66
ifeq 条件指令，66
ifndef 条件指令，66
ifneq 条件指令，66
immediate expansion（立即扩展），59
implicit rules（隐含规则），19，36
built-in（内置），36
source control and（源码控制与……），39
CVS，39
include 指令，65，66
dependency and（依存关系与……），67
header file（头文件），28
optimization（优化），133
include processing（引入处理），65
--include-dir 选项，262
info 工作目标，24
initialization, performance and（效能，初始化与……），200
input file, text printing（输入文件，文本输出），15
install 工作目标，24
installer, reference builds and（安装程序，引用编译结果与……），169
Integrated Development Environments（见 IDE）

interface, phony targets and（假想工作目标与接口），24
intermediate file, chaining rule and（链接规则与中间文件），38
.INTERMEDIATE 工作目标修饰符，41
invoking make（调用 make），15

J

jar 程序，187
  reference tree（引用树），189
  third-party（第三方），189
jars (Java)，187
Java
  Ant，172
    build file（编译文件），172
    mkdir 程序，174
    portability（可移植性），174
    task（任务），173
  Eclipse，171
  EJB，190
  IDE，175
  jars 与……，187
  make 与……，171
  通用 makefile，176
    CLASSPATH 变量，183
    dependecies and（依存关系与……），182
  package（包），171
Java virtual machine（JVM，Java 虚拟机），171
JBuilder，175
JIT (just-in-time) optimization（即时优化），171
--jobs 选项，202
join 函数，86
just-in-time (JIT) optimization（即时优化），171
--just-print 选项，262
--just-print 选项，调试，242
JVM（Java virtual machine，Java 虚拟机），171

K

--keep-going 选项，262
keyword（关键字）
  else，65
  endif，65
killing processe, user-defined function and（终止进程，用户自定义函数与……），73

L

large project（大型项目），119-140
library（程序库）
  archive library（程序库），45
  circular reference（循环引用），50
  creating（创建），47
  double-colon rule（双冒号规则），51
  make-library 函数，132
  as prerequisite（作为必要条件），49
  recursion and（递归与……），122
  reference builds and（引用编译结果与……），169
  .SECONDARY 工作目标修饰符，41
  updating（更新），47
library function, reusable（可重复使用的程序库函数），28
line termination, Cygwin（行终止符，Cygwin），143
Linux 内核的 makefile，230
  command echo（命令输出），235
  command-line option（命令行选项），230
  configuration versus building（配置与编译），232
  user-defined function（用户自定义函数），236
long command（太长的命令），102

M

m4 macro, book makefile and（本书的 makefile 与 m4 宏），222
macro（宏）
  debugging and（调试与……），92
  define 指令，57
  defining（定义），59
  expanding（扩展），58
  implementing, scoping and（有效范围与实现），75
  introduction（介绍），56
  invoking from another macro（从另一个宏来调用），74
  program-variable，94
  rules in（其中的规则），95
make
  automation and（自动化与……），11

command execution（命令的执行），14
comment（注释），18
dependency checking（检查依存关系），14
escape character（转义字符），18
invoking（调用），15
script and（脚本与……），11
make shell command, benchmarking and（基准测试与 make shell 命令），198
$(MAKE) 变量，121
MAKECMDGOALS 变量，69
makedepend，164
MAKEFILE_LIST 变量，69
makefile
本书的 makefile，208-217
m4 宏，222
output generation（输出的产生），225
source validation（源文件的确认），228
XML preprocessing（对 XML 的预处理），222
command execution（命令的执行），14
Hello World，11
Java, generic for（Java，……的通用），176
Linux kernel，230
command echo and（命令输出与……），235
command-line option（命令行选项），230
configuration versus building（配置与编译），232
user-defined function（用户自定义函数），236
syntax（语法），16
将工作目标指定为命令行参数，12
top-down style（从上而下的风格），14
MAKELEVEL variable, command environment（MAKELEVEL 变量，命令环境），110
make-library 函数，132
MAKE_VERSION 变量，68
matched rules, automatic variable（匹配规则，自动变量），25
members of archive library（程序库的成员），45
miscellaneous function（杂项函数），90
warning，92
module definition, find command（模块定义，find 命令），134
module.mk 引入文件，130
multiline command（多行命令），107

N

--new-file 选项，262
newline rule（换行字符规则），15
nonrecursive make（非递归式 make），129
no-op 命令，107
notdir 函数，84

O

object file, update（目标文件，更新），38
$(OBJECTS) 变量，34
--old-file 选项，262
option（选项）
command, error and（命令，错误与……），106
portability and（可移植性与……），142
origin 函数，90
output, book makefile example（输出，本书的 makefile 范例），225

P

packages, Java（Java 包），171
parallelism（并行式 make）
--jobs 选项，202
performance and（效能与……），202
pmake，206
parameter（参数）
passing to function（传递给函数），98
user-defined function（用户自定义函数），73
parenthese variable and（变量与圆括号），52，89
parsing（分析）
command（命令），100
command scripts and（命令脚本与……），101
editor and（编辑器与……），102
eval function and（eval 函数与……），95
partial source tree（部分的源文件树），168
passing parameter to function（传递参数给函数），98
passing variable, recursive make and（传递变量，递归式 make 与……），123

paths, portability and（可移植性、路径与……），142
patsubst 函数，79
pattern rules（模式规则），19，32
  %（百分比字符），33
  implicit rule and（隐含规则与……），19
  pattern（模式），33
  static pattern rules（静态模式规则），34
  suffix rule（后缀规则），35
    deleting（删除），35
pattern（模式），33
  作为内置函数的参数，76
  filter 函数，76
pattern-specific variable（模式专属变量），61
performance（效能）
  @ 符号与……，200
  benchmarking and（基准测试与……），194
  bottleneck（瓶颈），198
  distribution and（分布式 make 与……），206
  initialization and（初始化设定与……），200
  introduction（介绍），194
  parallelism and（并行式 make 与……），202
  recursive variable（递归变量），199
  simple variable（简单变量），199
.PHONY 工作目标修饰符，22
phony target（假想工作目标），22
  interface and（接口与……），24
  nonrecursive make and（非递归式 make 与……），129
  output（输出）
    debugging（调试），23
    reading（阅读），23
  prerequisite（必要条件），22
  special target（特殊工作目标），41
  standard（标准），24
pmake，206
portability（可移植性），141
  Ant (Java)，174
  Cygwin，143
  nonportable tool（不具可移植性的工具），149
  option and（选项与……），142
  path and（路径与……），142
  program behavior and（程序行为与……），143
  program name and（程序名称与……），142
  shell 与……，142，151
.PRECIOUS 工作目标修饰符，41
prefix（前缀）
  on commands（命令行上）
    @，104
    -（破折号），105
    +（加号），105
  pattern rule（模式规则），34
prerequisite（必要条件）
  automatic variable and（自动变量与……），25
  library as（以程序库为），49
  phony target（假想工作目标），22
  rule and（规则与……），12
  保存，50
  target（工作目标）
    chaining（链接），14
    .INTERMEDIATE 修饰符与……，41
    .SECONDARY 修饰符，41
  update, ordering and（更新，顺序与……），122
--print-data-base 选项，262
  debugging and（调试与……），242
program behavior, portability and（程序的行为，可移植性与……），143
program management（程序的管理），146
program names, portability and（程序的名称，可移植性与……），142
program-variables 宏，94

R

RCS source control, implicit rules and（隐含规则与 RCS 源代码控制），39
read-only source（只读的源文件树），161
rebuilds, minimizing（尽量减少重新编译的工作量），15
recursion（递归），119
recursive make（递归式 make），120
  build target and（建立工作目标与……），124
  code duplication（代码的重复），126
  command-line and（命令行与……），123
  dependency and（依存关系与……），122
  error handling（错误处理），124
  $(MAKE) 变量，121
  variable, passing（变量，传递），123
  （另见 nonrecursive make）

recursive variable（递归变量）
performance and（效能与……），199
shell 函数与……，81
recursively expanded variable（经递归扩展的变量），54
reference builds（引用编译结果）
installers and（安装程序与……），169
library and（程序库与……），169
reference tree, jar program（引用树，jar 程序），189
regtool，184
relative path, converting to Java class name（相对路径，转换成 Java 类名），82
release tree layout（发行树的布局），139
remote-file 函数，265
reusable library function, declaration（可重复使用的程序库函数，声明），28
rule chaining（链接规则），33
rule（规则）
any character（任何字符），15
chaining, intermediate file（链接，中间文件），38
customization, variables and（自定义，变量与……），38
default rule（默认规则），12
dependent（依存关系），12
double-colon（双冒号），51
explicit（具体），19、19-25
empty targets（空工作目标），24
phony targets（假想工作目标），22
implicit（隐含），19，36
built-in（内置），36
source control and（源代码控制与……），39
macro（宏），95
matching, automatic variables and（匹配，自动变量与……），25
newline（换行符），15
pattern（模式），19、32
suffix rule（后缀规则），35
prerequisite and（必要条件与……），12
static pattern（静态模式），19，34
structure（结构），38
suffix（后缀），19
target（工作目标），12
multiple（多个），19
variable, customization and（变量，自定义与……），38
run-make shell script, running book makefile（run-make shell 脚本，运行本书的 makefile），220

S

SCCS source control, implicit rules and（隐含规则与 SCCS 源代码控制），39
scoping, macro implementation and（宏实现与有效范围），75
script（脚本）
command script, parsing and（命令脚本，分析与……），101
make and（make 与……），11
search and replace function, string（搜索与替换函数，字符串），78
substitution reference（替换引用），79
search（搜索）
binary tree（二进制文件树），168
source tree（源代码树），168
VPATH and（VPATH 与……），27
vpath and（vpath 与……），27
.SECONDARY 工作目标修饰符，41
separators, missing（错误信息），254
shell
command line, subshell（命令行，subshell），108
portability and（可移植性与……），142，151
shell command, sequence（shell 命令，序列），102
shell 函数，80
variable（变量），81
simple variable（简单变量）
performance and（效能与……），199
shell 函数与……，81
simply expanded variable（经简单扩展的变量），54
sort 函数，80
source（源文件）
binary tree separation（二进制文件树的分离），153-161
src 目录，28
validation, book makefile（确认，本书的 makefile），228
source tree（源文件树）
layout, file management and（布局，文件管理与……），149
partial（部分的），168

read-only（只读的），161
search（搜索），168
sources of variable（变量的来源），62
source-to-object 函数，133
space-to-question 函数，148
special target（特殊工作目标），41
src directory, source file（src 目录，源文件），28
static pattern rule（静态模式规则），19，34
status code, command（状态码，命令），105
stderr 文件描述符，110
stdin 文件描述符，110
stdout 文件描述符，111
string function（字符串函数），76
filter，76
filter-out，77
findstring，77
firstword，80
patsubst，79
search and replace function（搜索与替换函数），78
subst，78
wordlist，80
words，79
strip 函数，90
whitespace removal（移除空格），66
structure of rules（规则的结构），38
subshell, command line and（subshell，命令行与……），108
subst 函数，78
substitution reference, string function（替换引用，字符串函数），79
suffix 函数，84
suffix rule（后缀规则），35
implicit rule and（隐含规则与……），19
suffix（后缀；扩展名）
filename, function（文件名，函数），84
pattern rule（模式规则），34
deleting（删除），35
target（工作目标），35
syntax（语法）
built-in function（内置函数），75
conditional directive（条件指令），65
editor（编辑器），102
error, debugging and（错误，调试与……），254
makefile，16
target-specific variable（工作目标的专属变量），62

T

tab character, error message（跳格符，错误信息），254
TAGS 工作目标，24
target（工作目标）
automatic variable and（自动变量与……），25
build target, recursive make and（建立工作目标，递归式 make 与……），124
as command-line argument（指定成命令行参数），12
command, overriding（命令，覆盖掉），257
deleting（删除），108
empty（空），24
modifier（修饰符），41
.DELETE_ON_ERROR，42
.INTERMEDIATE，41
.PHONY，22
.PRECIOUS，41
.SECONDARY，41
phony target（假想工作目标），22
special target（特殊工作目标），41
prerequisite（必要条件）
chaining（链接），14
saving（保存），50
rule（规则），12
explicit rule（具体规则），19
multiple（多个），19
special target（特殊工作目标），41
static pattern rule（静态模式规则），34
suffix（扩展名），35
updating, rule chaining and（更新，链接规则与……），33
target-specific variable（工作目标的专属变量），61
syntax（语法），62
tasks（任务）（Ant），173
text expansion, foreach function and（foreach 函数与文本的扩展），88
timestamp（时间戳）
empty file and（空文件与……），24
filenames and（文件名与……），82
top-down style（从上而下的风格），makefile，14
--touch 选项，262

U

update（更新）
library（程序库），47
object file, rule（目标文件，更新），38
prerequisite, ordering and（必要条件，顺序与……），122
target, rule chaining（工作目标，链接规则），33
user-defined function（用户自定义函数），72-75
advanced（高级的），91-99
killing processes and（终止进程与……），73
Linux kernel makefile，236
参数与……，73
变量与……，53

V

validating source, book makefile（确认源文件，本书的makefile），228
value 函数，97
.VARIABLES 变量，70
variable（变量），25-27
ALL_TREES，167
assigning, speed（赋值的速度），197
automatic（自动），25，64
空工作目标与……，25
built-in rule（内置规则），38
case-sensitivity（区分大小写），52
characters allowed（允许使用的字符），52
CLASSPATH, Java makefile，183
经求值的，assert 函数与……，88
constant, user-customized（常数，用户自定义），53
CPPFLAGS，61
CURDIR，68
debug-enter，92
debug-leave，92
development environment（开发环境），64
EJBS，192
error message（错误信息），255
expanding（扩展），58
curly braces and（花括号与……），52
exporting（导出），63
function, user-defined（函数，用户自定义），53
grep 命令，110
introduction（介绍），52
macros and（宏与……），57
MAKE，121
MAKECMDGOALS，69
MAKEFILE_LIST，69
MAKEFLAGS，环境变量与……，110
MAKELEVEL，110
MAKE_VERSION，68
OBJECTS，34
operator（运算符）
:=，54
=，53
+=（附加），56
?=（附带条件的变量赋值运算符），55
origins, origin function（来自何处，origin 函数），91
parentheses and（圆括号与……），52，90
passing, recursive make and（递归式 make 与传递），123
pattern-specific（模式专属的），61
periods embedded（内嵌的点号），264
recursive, performance and（递归的，效能与……），199
recursively expanded（经递归扩展），54
shell 函数，81
simple, performance and（简单的，效能与……），199
simply expanded（经简单扩展），54
sources（来源），62
target-specific（工作目标专属的），61
trailing spaces and（跟在后面的空格与……），53
uses（使用），53
.VARIABLE，70
VARIABLES，70
VPATH（见 VPATH）
VARIABLES 变量，70
VPATH，27
vpath，27
vpath 指令，30

W

warning 函数，91
debugging and（调试与……），241
—warn-undefined-variables 选项，263
debugging and（调试与……），245

whitespace（空格）
functions for manipulating（用来处理的函数），79
removing（移除），75
strip 函数，90
wildcard 函数，83
wildcard（通配符），21
?（问号），21
~（波浪号），21
calling functions and（调用函数与……），21
character classes（字符集），21
expanding（扩展），21
misuse（误用），21
pattern rules and（模式规则与……），19
（另见 globbing）
wildcard-spaces 函数，148
Windows filesystem, Cygwin and（Cygwin 与 Windows 文件系统），144
Windows, using Cygwin and make on（在 Windows 系统上使用 Cygwin 和 make），143
word 函数，80
wordlist 函数，80
words 函数，80

X

XML
build file（编译文件），172
对本书的 makefile 进行预处理，222

有关此电子图书的说明

本人由于一些便利条件，可以帮您提供各种中文电子图书资料，且质量均为清晰的 PDF 图片格式，质量要高于网上大量传播的一些超星 PDG 的图书。方便阅读和携带。只要图书不是太新，文学、法律、计算机、人文、经济、医学、工业、学术等方面的图书，我都可以帮您找到电子版本。所以，当你想要看什么图书时，可以联系我。我的 QQ 是：85013855，大家可以在 QQ 上联系我。

此 PDF 文件为本人亲自制作，请各位爱书之人尊重个人劳动，敬请您不要修改此 PDF 文件。因为这些图书都是有版权的，请各位怜惜电子图书资源，不要随意传播，否则，这些资源更难以得到。

有关此电子图书的说明

本人由于一些便利条件，可以帮您提供各种中文电子图书资料，且质量均为清晰的 PDF 图片格式，质量要高于网上大量传播的一些超星 PDG 的图书。方便阅读和携带。只要图书不是太新，文学、法律、计算机、人文、经济、医学、工业、学术等方面的图书，我都可以帮您找到电子版本。所以，当你想要看什么图书时，可以联系我。我的 QQ 是：85013855，大家可以在 QQ 上联系我。

此 PDF 文件为本人亲自制作，请各位爱书之人尊重个人劳动，敬请您不要修改此 PDF 文件。因为这些图书都是有版权的，请各位怜惜电子图书资源，不要随意传播，否则，这些资源更难以得到。