



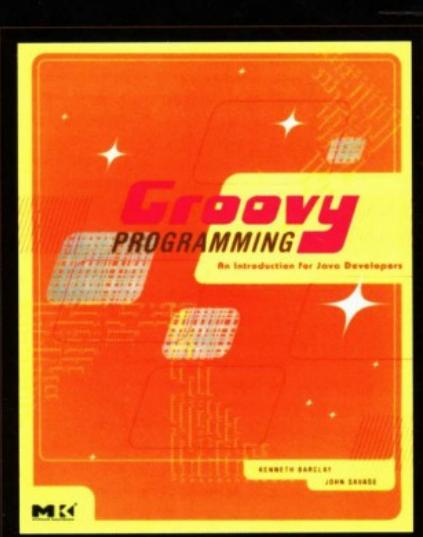
华章程序员书库



Groovy 入门经典

Groovy Programming

An Introduction for Java Developers



(英) Kenneth Barclay John Savage 著
龚波 张平 陈蓓 王琦 等译

“如果想全面了解有关脚本、面向对象或者动态语言等相关知识，本书是最适合的！”

Andrew Glover, President, Stelligent Incorporated



机械工业出版社
China Machine Press

Groovy 入门经典

Groovy Programming An Introduction for Java Developers

适合Java开发者的Groovy入门书

Groovy是唯一能够扩展Java平台的脚本语言。Groovy提供类似于Java的语法结构，本地化支持映射和列表、方法、类、闭包和构造器等结构。由于具有动态弱类型，以及无缝访问Java API等特性，Groovy语言非常适合于开发中型规模的应用程序。

相对于Java语言，Groovy语言的表达性更强，抽象程度更高。它使得应用程序开发更加快捷，提升了程序员生产力。Groovy语言可以用作应用程序的“黏合剂”，而不必实现复杂的数据结构和算法。

与Java代码比较起来，Groovy语言的另一个主要优势是，Groovy代码编写量相对小些。在一般情况下，Java代码过于复杂，难于理解和维护。这是因为Java需要大量的模板文件或者转换代码，而Groovy却不需要。

本书是有关Groovy的第一本正式出版物，作者Kenneth Barclay和John Savage介绍了Groovy开发的所有主要领域，并解释了这种创新性的编程语言给Java平台赋予的动态特性。阅读本书只要求具备Java编程的一般性知识。不管你是经验丰富的Java开发者，还是脚本语言的新手，都会得到如何充分利用Groovy语言的专家性指导。

本书特点

- 第一本全面讲解Groovy编程的图书，演示如何快速和轻松地编写适用于Java平台的应用程序和脚本。
- 本书由高水平软件工程师精心编写，并得到计算机老师和专家的高度赞扬。
- 提供大量编程范例、代码范例、详细的案例分析、习题，非常适合自学。本书还有一个支持网站，并提供一个基于Windows的Groovy编辑器。

关于作者

Kenneth Barclay和John Savage都是Napier大学计算机学院（Edinburgh, Scotland）的教师。他们的专长是面向对象软件工程开发和编程。他们是《Object-Oriented Design with UML and Java》（Butterworth-Heinemann, 2003）一书的合著者。



本书译自原版Groovy Programming: An Introduction for Java Developers，并由Elsevier授权出版

上架指导：计算机/程序设计

ISBN 978-7-111-22493-8



9 787111 224938

ISBN 978-7-111-22493-8

定价：49.00 元

投稿热线：(010) 88379604

购书热线：(010) 68995259, 68995264

读者信箱：hzjsj@hzbook.com

华章网站 <http://www.hzbook.com>

网上购书：www.china-pub.com

封面设计 王建放



华章程序员书库



Groovy 入门经典

Groovy Programming

An Introduction for Java Developers

(英) Kenneth Barclay John Savage 著
龚波 张平 陈蓓 王琦 等译



机械工业出版社
China Machine Press

本书详细介绍脚本语言Groovy，首先介绍Groovy语言的基本特性，包括讨论Groovy方法、程序闭包、列表、映射以及对类和继承的支持，然后介绍如何使用Groovy创建更加高级的应用程序，如使用Spring框架和Cloudscape/Derby关系型数据库管理系统来实现持久性，最后讨论模板和Web应用程序。

本书内容全面详尽，浅显易懂，易于选择性阅读。可以作为对Groovy语言感兴趣的计算机软件开发人员的参考书。

Groovy Programming: An Introduction for Java Developers

Kenneth Barclay, John Savage

ISBN: 0-12-372507-0

Copyright © 2007 by Elsevier Inc. All rights reserved.

Authorized Simplified Chinese translation edition published by the Proprietor.

ISBN: 981-259-991-6

Copyright © 2007 by Elsevier (Singapore) Pte Ltd. All rights reserved.

Printed in China by China Machine Press under special arrangement with Elsevier (Singapore) Pte Ltd. This edition is authorized for sale in China only, excluding Hong Kong SAR and Taiwan. Unauthorized export of this edition is a violation of the Copyright Act. Violation of this Law is subject to Civil and Criminal Penalties.

本书简体中文版由机械工业出版社与Elsevier(Singapore)Pte Ltd.在中国大陆境内合作出版。本版仅限在中国境内（不包括中国香港特别行政区及中国台湾地区）出版及标价销售。未经许可之出口，视为违反著作权法，将受法律之制裁。

版权所有、侵权必究。

本书法律顾问 北京市展达律师事务所

本书版权登记号：图字：01-2007-4489

图书在版编目（CIP）数据

Groovy入门经典/（英）巴克莱（Barclay, K.），（英）萨维奇（Savage, J.）著；龚波等译. —北京：机械工业出版社，2007.10

（华章程序员书库）

书名原文：Groovy Programming an Introduction for Java Developers

ISBN 978-7-111-22493-8

I . G… II . ①巴… ②萨… ③龚… III . 程序语言—程序设计 IV . TP312

中国版本图书馆CIP数据核字（2007）第154315号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）

责任编辑：王春华

北京牛山世兴印刷厂印刷·新华书店北京发行所发行

2008年1月第1版第1次印刷

186mm×240mm · 22.75印张

定价：49.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：(010) 68326294



专业成就人生
立体服务大众

www.hzbook.com

填写读者调查表 加入华章书友会
获赠精彩技术书 参与活动和抽奖

尊敬的读者：

感谢您选择华章图书。为了聆听您的意见，以便我们能够为您提供更优秀的图书产品，敬请您抽出宝贵的时间填写本表，并按底部的地址邮寄给我们（您也可通过www.hzbook.com填写本表）。您将加入我们的“华章书友会”，及时获得新书资讯，免费参加书友会活动。我们将定期选出若干名热心读者，免费赠送我们出版的图书。请一定填写书名书号并留全您的联系信息，以便我们联络您，谢谢！

书名： 书号： 7-111-()

姓名：	性别： <input type="checkbox"/> 男 <input type="checkbox"/> 女	年龄：	职业：
通信地址：		E-mail：	
电话：	手机：	邮编：	

1. 您是如何获知本书的：

朋友推荐 书店 图书目录 杂志、报纸、网络等 其他

2. 您从哪里购买本书：

新华书店 计算机专业书店 网上书店 其他

3. 您对本书的评价是：

技术内容	<input type="checkbox"/> 很好	<input type="checkbox"/> 一般	<input type="checkbox"/> 较差	<input type="checkbox"/> 理由_____
文字质量	<input type="checkbox"/> 很好	<input type="checkbox"/> 一般	<input type="checkbox"/> 较差	<input type="checkbox"/> 理由_____
版式封面	<input type="checkbox"/> 很好	<input type="checkbox"/> 一般	<input type="checkbox"/> 较差	<input type="checkbox"/> 理由_____
印装质量	<input type="checkbox"/> 很好	<input type="checkbox"/> 一般	<input type="checkbox"/> 较差	<input type="checkbox"/> 理由_____
图书定价	<input type="checkbox"/> 太高	<input type="checkbox"/> 合适	<input type="checkbox"/> 较低	<input type="checkbox"/> 理由_____

4. 您希望我们的图书在哪些方面进行改进？

5. 您最希望我们出版哪方面的图书？如果有英文版请写出书名。

6. 您有没有写作或翻译技术图书的想法？

是，我的计划是_____ 否

7. 您希望获取图书信息的形式：

邮件 信函 短信 其他_____

请寄：北京市西城区百万庄南街1号 机械工业出版社 华章公司 计算机图书策划部收

邮编：100037 电话：(010) 88379512 传真：(010) 68311602 E-mail: hzjsj@hzbook.com

序 言

脚本语言不是新生事物。通常，脚本语言运行于Linux和UNIX操作系统环境下，可以完成很多shell脚本任务，比如软件自动安装和配置，平台定制，使用Python语言建立科学的应用程序原型，以及使用bash脚本完成一次性的命令行任务。有的脚本语言（如PHP）已经广泛地用于开发许多网站。事实证明，脚本语言适合于实现关键的业务应用程序。

在通常情况下，脚本语言是独立的平台，一般情况下不与其他平台进行交互。尽管在桥接（bridge）其他系统时也许存在绑定现象，但是这种集成未必都非常直观或者自然。Groovy试图解决这个问题，它是一种创新的语言，能够自然地通过相同虚拟机的Java环境进行交互。

Groovy使用简练的、易于理解的类Java语法，这样可以降低Java程序员学习Groovy语言的难度。除了语法类似于Java语言之外，Groovy给常见的JDK应用程序编程接口（Application Programming Interface）提供包装器API，进一步拓展Groovy的应用领域。借助Groovy，可以简化常见任务的实现，并集成元编程能力，以开发功能强大的新语言结构，或者轻松地处理已有的语言结构。

Groovy可以用于多种情况，比如作为shell脚本语言完成数据处理和文件操作任务，或者试验使用新的API。它也适合于创建强大的小型或者中型应用程序，能够充分利用Java库和组件。除此之外，另一个重要用法是在Java或者Java EE应用程序中嵌入Groovy，实现Java和Groovy的集成。这样做有助于编写和集中处理经常变化的业务逻辑，或者给应用程序架构提供可编程的配置管理能力。

尽管前两种用法很常见，但是我认为嵌入式应用是最吸引人的，也是最有前途的。现在，开发者始终使用模板引擎来定制和重构我们的视图，或者使用业务逻辑引擎来实现逻辑的外部化和集中化。对于一般的语言来说，除了有限的函数集外，程序员通常很难获得更多的开发支持。幸运的是，宿主于平台的脚本语言，比如Groovy语言，有助于解决这种功能需求障碍。Groovy及其后代Grails（一个功能强大的模型-视图-控制器Web框架）的成功就很能说明这个问题。Sun公司也认可这种向应用程序添加动态性的替代方法，在Java 6中包含新的Java 规范请求（Specification Request）：javax.script.* API能够把任何脚本或者动态语言无缝地嵌入到具有一致的编程API的Java应用程序中。

脚本语言的发展已经达到一定的成熟度，能够突破标准化主流平台的限制。当集成这些语言和平台时，你将会很快得到意想不到的欣喜。

Ken Barclay和John Savage都是值得尊敬的讲师，完全有资格向有经验的开发者和开发新手讲解Groovy。他们以清晰有序的方式介绍如何使用Groovy增加Java平台的魅力，以及如何充分利用Groovy的新功能。本书浅显易懂，即使开发新手也能使用，内容全面详尽，涉及Groovy语言的方方面面。

本书的组织结构是：前面几章着重介绍Groovy语言的基本知识，后面的章节分门别类讨论与该语言相关的高级概念。除此之外，本书还提供很多内容丰富的附录，让读者可以了解相关主题的更详细信息。

本书每章的篇幅较小，易于选择性阅读；每章提供大量完整的代码范例、练习和解决方案。为尽可能清晰地展示如何使用Groovy语言，本书提供一个学习案例，随着每章的学习进度，这个学习案例逐渐演化，功能越来越复杂和完善。除此之外，增量开发和单元测试也是本书的一个重要主题，这是为了支持Groovy的动态特性。本书作者也认为Groovy是一种多模式语言。

作者基于多年的行业开发经验，认为Groovy语言适合作为学校语言课程，也适合作为有经验开发者的常备开发工具。

衷心希望你通过阅读本书能够很好地掌握Groovy，相信你不会感到遗憾的！

Guillaume Laforge
Groovy Project Manager
JSR-241 Specification Lead

前　　言

本书是脚本语言Groovy的入门书。对于Java开发者来说，Groovy语言会使得编写Java平台的脚本和应用程序变得既快又容易。Groovy语言包含很多在其他脚本语言（诸如Python、Ruby和Smalltalk）中可见的语言特性。因为Groovy是一种基于Java的语言，Groovy语言编写的应用程序可以完全使用Java应用程序编程接口（API）。这意味着，Groovy可以与使用Java语言所编写的框架和组件完美集成。

脚本语言Groovy和系统编程语言Java相互补充，综合使用两者可以加速程序开发过程。比如，可以使用Java语言编写框架和组件，而把Groovy语言用作框架和组件的“黏合剂”。Groovy语言的易用性有利于大大扩展Groovy的应用范围。现在，组件架构、图形化用户界面(Graphical User Interface, GUI)、数据库访问以及因特网应用等日趋重要，这有利于Groovy脚本语言的应用和发展。

Groovy开发者可以充分利用脚本语言所特有的快速应用程序开发特性，Groovy适合处理涉及大量数据或者文件操作的任务，应用程序测试，或者在小型或者中型项目中替代Java语言。

Groovy的语法类似于Java编程语言的语法。这样可以大大缩短Java程序员学习Groovy语言的时间。Java平台的其他脚本语言通常都是基于早期的预处理器概念，这样就存在先天性的发展障碍。但是，Groovy语言“就是”Java语言，提供与Java平台更自然、更无缝地集成。

本书组织方式

本书的内容组织形式注重引导读者快速了解Groovy语言的基本元素，并且假定阅读本书的读者已经阅读过至少一本Java图书。对于本书最后几章，如果读者具有Swing、标准查询语言(Standard Query Language, SQL)、Spring、XML、Ant以及Web应用程序构建等方面的经验，则学习起来会更加自如。作者尽量保证每章目标的单一性和简要性。有些读者也许希望在一章中就可以学习到特定Groovy特性的全部内容，希望本书的这种组织方式能够实现读者的这个目标。同样，每章内容的简短性也有利于读者进行选择性学习，直接跳转到自己关注的内容章节。

本书中很多章都有对应的附录，以更加详细地介绍所讨论的主题。比如，第7章介绍定义和使用Groovy方法的基本概念。附录G更加深入地讨论这个概念，诸如方法的重载和递归，这些更加高级的话题却不是正文的核心内容。同样，这样做也有利于正文中内容简短单一。

本书中很多章也包含很多简短易懂的范例，以说明相应的语言概念。这些范例是完整的，建议读者在学习过程中尝试和试验这些范例。每章最后也有对应的练习，读者应该尝试做这些练习。本书的网站提供每章中范例的源代码，以及练习的解决方案。

本书的一个特色是提供一个不断演化的学习案例——管理和维护一个电子图书馆的借阅库。在本书中，每一章都会逐步完善这个学习案例，使用刚讨论的Groovy新特性来增强这个学习案例的功能。比如，在第11章中，学习案例增加了以前章所介绍的方法、程序闭包(closure)，

以及文件处理等内容。

本书第1部分包括第1~16章，介绍Groovy语言的基本特性。比如，讨论Groovy方法、程序
闭包、列表、映射，以及对类和继承的支持。

自动化单元测试也是本书的重要讨论主题。Groovy的快速创建-运行循环使之非常适合于
开发单元测试。Groovy充分利用工业标准JUnit框架，来简化单元测试的创建和使用过程。单
元测试在Groovy中的应用集成了动态类型语言的灵活性和静态类型语言的安全性。为加强说明
这一点，本书把单元测试作为大多数学习案例中不可或缺的一部分。

本书的第2部分包括第17~24章，讨论如何使用Groovy创建更加高级的应用程序。比如，
使用Spring框架和Cloudscape/Derby关系型数据库管理系统来实现持久性。Groovy提供独特的
生成器符号，以支持XML和GUI应用程序。本书最后讨论模板和Web应用程序。

本书约定

本书提供很多与所介绍主题相关的网站和参考书目。

一般情况下不区分“程序”和“脚本”的概念，两个术语可以替换使用。但是，在本书中，
这两个概念始终专指Groovy脚本。

软件发布

作者搭建本书的一个支持性网站——<http://www.dcs.napier.ac.uk/~kab/groovy/groovy.html>，
其中包含本书中所有范例和学习案例的脚本，也提供每章最后练习的答案。

Groovy在不断发展变化着。作者希望通过这个网站让读者有机会了解最新的重要变更。因
此，建议读者经常浏览这个网站，了解最新的信息。

感谢

本书作者非常感谢参与Groovy概念设计和开发，以及Java Specification Request (JSR-241)
(<http://www.dcs.napier.ac.uk/~cs05/groovy/groovy.html>) 的人们。本书的目的是宣扬Groovy语
言。我们感谢始终改进Groovy的Guillaume Laforge (Groovy Project Manager)，感谢Andrew
Glover (CTO, Vanward Technologies) 在IBM Developer网站发表有关Groovy的文章。同样，非
常感谢Jon Kerridge教授 (School of Computing, Napier University, Edinburgh)，他始终鼓励和
支持这个项目，并引导我们关注以前没有注意到的领域和挑战。

本书作者也感谢Denise Penrose、Tim Cox、Mary James、Christine Brandt以及他们在
Morgan Kaufmann的同事，感谢Elsevier在本书出版过程中给予的帮助。最后，感谢技术评审专
家Andrew Glover和Sean Burke，他们提供大量有价值的建议。但是本书中出现的任何错误都是
本书作者的责任。

目 录

序言	
前言	
第1章 Groovy	1
1.1 为什么使用脚本语言	1
1.2 为什么使用Groovy	2
第2章 数值和表达式	3
2.1 数值	3
2.2 表达式	3
2.3 运算符优先级	5
2.4 赋值	5
2.5 自增和自减运算符	6
2.6 对象引用	6
2.7 关系运算符和等于运算符	7
2.8 习题	8
第3章 字符串和正则表达式	10
3.1 字符串字面值	10
3.2 字符串索引和索引段	10
3.3 基本操作	11
3.4 字符串方法	11
3.5 比较字符串	14
3.6 正则表达式	15
3.7 习题	16
第4章 列表、映射和范围	18
4.1 列表	18
4.2 列表方法	19
4.3 映射	21
4.4 映射方法	22
4.5 范围	23
4.6 习题	24
第5章 基本输入输出	26
5.1 基本输出	26
5.2 格式化输出	27
5.3 基本输入	28
5.4 习题	30
第6章 学习案例：图书馆应用	
程序（建模）	31
6.1 迭代1：需求规范和列表实现	31
6.2 迭代2：映射实现	32
6.3 习题	34
第7章 方法	35
7.1 方法	35
7.2 方法参数	37
7.3 默认参数	37
7.4 方法返回值	38
7.5 参数传递	39
7.6 作用域	40
7.7 集合作为参数和返回值	42
7.8 习题	42
第8章 流程控制	44
8.1 while语句	44
8.2 for语句	46
8.3 if语句	47
8.4 switch语句	49
8.5 break语句	53
8.6 continue语句	53
8.7 习题	54
第9章 闭包	57
9.1 闭包	57
9.2 闭包、集合和字符串	60
9.3 闭包的其他特性	65
9.4 习题	68
第10章 文件	71
10.1 命令行参数	71
10.2 File类	71

10.3 习题	77
第11章 学习案例：图书馆应用程序 (方法、闭包)	79
11.1 迭代1：需求规范和映射实现	79
11.2 迭代2：基于文本的用户交互界面的 实现	83
11.3 迭代3：使用闭包实现	85
11.4 习题	88
第12章 类	89
12.1 类	89
12.2 复合方法	95
12.3 习题	97
第13章 学习案例：图书馆应用 程序（对象）	99
13.1 需求规范	99
13.2 迭代1：最初的模型	99
13.3 迭代2：模型完善	101
13.4 迭代3：用户界面	106
13.5 习题	111
第14章 继承	113
14.1 继承	113
14.2 继承方法	115
14.3 方法重定义	117
14.4 多态性	117
14.5 抽象类	120
14.6 接口类	123
14.7 习题	126
第15章 单元测试（JUNIT）	130
15.1 单元测试	130
15.2 GroovyTestCase类和JUnit TestCase类	131
15.3 GroovyTestSuite类和JUnit TestSuite类	136
15.4 单元测试的角色	138
15.5 习题	141
第16章 学习案例：图书馆应用 程序（继承）	142
16.1 需求规范	142
16.2 迭代1：多态性	143
16.3 迭代2：功能性需求演示	145
16.4 迭代3：提供用户反馈	149
16.5 迭代4：强制性约束	157
16.6 习题	160
第17章 持久性	162
17.1 简单查询	162
17.2 关系	164
17.3 更新数据库	165
17.4 表的对象	170
17.5 继承	172
17.6 Spring框架	173
17.7 习题	176
第18章 学习案例：图书馆应用程 序（持久性）	177
18.1 迭代1：域模型的持久化	177
18.2 迭代2：持久性的影响	186
18.3 习题	192
第19章 XML构造器和解析器	193
19.1 Groovy标记	193
19.2 MarkupBuilder	194
19.3 XML解析	196
19.4 习题	207
第20章 GUI构造器	208
20.1 SwingBuilder	208
20.2 列表框和表格	215
20.3 Box类和BoxLayout类	220
20.4 习题	221
第21章 模板引擎	224
21.1 字符串	224
21.2 模板	224
21.3 习题	228
第22章 学习案例：图书馆应用 程序（GUI）	229
22.1 迭代1：GUI原型	229
22.2 迭代2：处理器的实现	231
22.3 习题	238
第23章 服务器端编程	239
23.1 Servlets	239
23.2 Groovlets	240

23.3 GSP页面	245
23.4 习题	249
第24章 学习案例：图书馆应用 程序（WEB）	250
24.1 迭代1：Web实现	250
24.2 习题	253
第25章 后记	254
附录A 软件发布	255
A.1 Java开发工具	255
A.2 Groovy开发工具	255
A.3 ANT	255
A.4 Derby/Cloudscape数据库	256
A.5 Spring框架	256
A.6 Tomcat服务器	256
A.7 Eclipse IDE	256
A.8 本书源文件	256
附录B Groovy简介	258
B.1 简洁和优雅	258
B.2 方法	259
B.3 列表	260
B.4 类	260
B.5 多态性	261
B.6 闭包	262
B.7 异常	263
附录C 关于数值和表达式的更多信息	264
C.1 类	264
C.2 表达式	264
C.3 运算符结合性	264
C.4 定义变量	265
C.5 复合赋值运算符	266
C.6 逻辑运算符	266
C.7 条件运算符	267
C.8 数字字面值的分类	268
C.9 转换	268
C.10 静态类型	269
C.11 测试	270
附录D 关于字符串和正则表达式的 更多信息	272
D.1 正则表达式	272
D.2 单字符匹配	273
D.3 匹配开始部分	273
D.4 匹配结尾部分	273
D.5 匹配零次或者多次	273
D.6 匹配一次或者多次	273
D.7 匹配零次或者一次	274
D.8 次数匹配	274
D.9 字符类型	274
D.10 选择	275
D.11 辅助符号	275
D.12 组合	276
附录E 关于列表、映射和范围的 更多信息	278
E.1 类	278
E.2 列表	279
E.3 范围	279
E.4 展开操作符	280
E.5 测试	280
附录F 关于基本输入输出的更多信息	283
F.1 格式化输出	283
F.2 类Console	285
附录G 关于方法的更多信息	287
G.1 递归方法	287
G.2 静态类型	289
G.3 实参协议	290
G.4 方法重载	290
G.5 默认参数值的不确定性	291
G.6 参数和返回值类型为集合的方法	292
附录H 关于闭包的更多信息	295
H.1 闭包和不明确性	295
H.2 闭包和方法	295
H.3 默认参数	296
H.4 闭包和作用域	296
H.5 递归闭包	297
H.6 状态类型	297
H.7 有关实参的约定	298
H.8 闭包、集合和范围	298

H.9 Return语句	299	J.3 组合	319
H.10 测试	300	J.4 计算模式	320
附录I 关于类的更多信息	303	J.5 业务规则	322
I.1 属性和可见性	303	J.6 打包	324
I.2 对象导航	307	J.7 列表简化	330
I.3 静态成员	310	J.8 习题	332
I.4 操作符重载	312	附录K 关于构造器的更多信息	334
I.5 调用方法	313	K.1 AntBuilder	334
I.6 习题	315	K.2 专用的构造器	341
附录J 高级闭包	316	附录L 关于GUI构造器的更多信息	345
J.1 简单闭包	316	L.1 菜单和工具条	345
J.2 部分应用	318	L.2 对话框	350

第1章 Groovy

本章简要介绍Groovy。Groovy是增强Java平台的唯一的脚本语言。它提供类似于Java的语言，内置映射（Map）、列表（List）、方法、类、闭包（closure）以及生成器。Groovy语言具有动态的弱类型，以及对Java应用程序接口（API）的无缝访问，因此非常适合用于开发小型和中型应用程序。

1.1 为什么使用脚本语言

一般来说，和Java这样的系统编程语言相比，脚本语言（比如Groovy）具有更好的表示能力，能够提供更高的抽象等级。这通常会提供更快捷的应用程序开发能力，以及更高的编程生产力。但是，脚本语言和系统编程语言的目标是不同的。脚本语言用于把应用程序集成起来，而不是实现复杂的数据结构和算法。因此，为了保证实用性，脚本语言必须能够访问不同类型组件。

通常，脚本语言不会替代系统编程语言，它们相互补充（Ousterhout, 1998）。一般来说，系统编程语言应该用于如下目的：

- 开发复杂的算法或者数据结构。
- 实现计算密集型应用。
- 操作大型数据集。
- 实现良好定义的、缓慢变更的需求。
- 是大型项目的一部分。

而脚本语言应该用于如下目的：

- 连接已有的组件。
- 处理经常变化的多种类型的实体。
- 具有图形化用户界面。
- 拥有快速变化的功能。
- 是小型或者中型项目的一部分。

相对于系统编程语言，脚本语言的主要长处是所需的编码工作量相对少。通常，系统编程语言的代码看起来非常复杂，难以维护。这是因为系统编程语言的代码需要大量的模板或者转换代码。

系统编程语言是强类型的，能够确保代码的安全性和健壮性。在强类型语言中，变量必须指定为一种类型，只能按照固定方式使用。尽管强类型特性使得大型程序的可管理性更好，并且允许编译器（静态地）检测特定类型的错误，但可能有时候起不到类型安全保护作用。比如，当事先很难或者不可能决定变量的类型时，强类型是没有用处的。当连接组件时，这种情况会经常发生。

为简化组件连接任务，脚本语言被设计成弱类型。这意味着，在不同环境下，变量可以以多种方式使用。但是，当代码被实际执行时，才会检测变量是否被非法使用。比如，尽管

Groovy在编译时（静态地）检查程序的语法，（动态地）检测方法调用是否正确发生在运行时。最终结果是，正确编译的Groovy脚本在运行时也许会抛出异常，甚至导致非正常结束。

弱类型并不意味着代码是不安全的，或者不健壮。极限编程（Beck, 2004）已经成为一种软件开发方法。这个方法注重测试，使用全面的单元测试方案（Link, 2003）来驱动开发过程。通过在不同环境下执行所编写的代码，就可以保证代码的安全性和健壮性。当开发Groovy脚本时，单元测试应该是基础的开发过程。实际上，开发经验已经证明，在弱类型语言中，综合运用弱类型和单元测试通常比传统系统编程语言的强类型检测更好（请参考<http://www.mindview.net/WebLog/log-0025>的相关文档）。这样的话，就同时拥有弱类型的灵活性和单元测试的全面保障。

1.2 为什么使用Groovy

Java编译器会产生可以在Java虚拟机上运行的字节码。Groovy类和Java是二进制兼容的。这意味着，Groovy编译器产生的字节码与Java编译器产生的字节码是完全一样的。因此，对JVM而言，Groovy和Java是完全一样的。这就等于说，Groovy能够完全使用各种Java API，诸如用于数据库开发的JDBC（Fisher et al., 2003），以及用于开发GUI应用程序的Swing（Topley, 1998）。

Groovy的目标是把大量开发者需要做的工作让语言本身来实现。比如，当往GUI添加一个按钮时，只需要提供当按钮被单击时要执行的代码，而无需给这个按钮添加一个事件处理器作为实现特定接口的类的实例。Groovy就是这样做的。

Groovy是一种面向对象的脚本语言，其中涉及的所有事物都是对象，这一点不像Java语言。这样就可以实现语言语法的一致性。Groovy也是动态类型语言，类型标记存在于对象中，而不是由引用它的变量来决定。这样做的结果是，Groovy不要求声明变量的类型、方法的参数或者方法的返回值。这样一来，就可以大大缩短代码规模，并允许程序员把类型决定时间推迟到代码运行时。

通过提供概念“属性”（property），Groovy也尝试统一类中的实例字段和方法。属性概念可以消除实例字段（attribute）和方法之间的差别。结果是，客户端可以把一个属性认为是实例字段及其获取器/设置器（getter/setter）方法的组合。

重要的数据结构，比如Map和List，都是Groovy语言内置的。可以使用Groovy脚本直接表示一个List对象或者Map对象。对于开发新手来说，直接实现List和Map对象会让编程任务更加简单。List和Map对象都提供interator（迭代器）方法，比如each，可以简化处理这些集合中每个元素的过程。可以使用一个closure（闭包）来声明处理过程，闭包是表示一个代码块的对象。这是个非常有价值的结构，可以被变量引用，带参数，被作为参数传入方法或者其他闭包，也可以是类的实例字段。在Groovy编程中，闭包具有举足轻重的地位。

层次性数据结构，比如XML，也可以直接使用Groovy生成器所生成的Groovy脚本来表示。借助于XPath（<http://www.w3.org/TR/xpath20/>）中的标记，Groovy可以快速地表示这些结构的路径，以及引用不同部分的方法。同样，迭代器和闭包提供处理它们的机制。

通常，Groovy生成器适用于任何被嵌套的树型结构。比如，它们可以用于描述使用多种组件组装而成的图形化应用程序。闭包可以充当组件（比如菜单项和按钮）的事件处理器。标准查询语言（SQL）的处理过程也是规范统一的。迭代器方法（比如eachRow）与闭包一起可以用来表示如何处理数据库表中的数据行。

第2章 数值和表达式

本章将集中讨论如何使用Groovy来处理基本的数值类型。在这个过程中，尤为重要的是，必须认识到Groovy是一门面向对象的语言。这也就是说，Groovy中每一个事物最终都会被当作某些类的一个实例对象。举例来说，在Groovy中，大家非常熟悉的整数123实际上是Integer类的一个实例。为使用一个对象来实现相关功能，必须调用所属类声明的某个方法。因此，在Groovy环境下，可以使用123.abs() 表达式调用abs方法来获得整数对象123的绝对值。同样，如要获得123的下一个整型值(124)的话，可以调用获取后续值的next方法，如123.next()所示。

正因为如此，在Groovy环境中，如果想得到整数123和整数456的算术之和的话，可以通过调用Integer类的“+”方法来实现，比如123.(+)456表达式。Integer对象456是该方法的参数。当然，和在学校学到的算术技巧相比，这种做法一点也不直观。幸运的是，Groovy同样也支持运算符重载（参见附录I）。这样，+方法就能够作为一个二元操作符出现在其两个参数之间，表达式123+(+)456将更符合人们的习惯。然而，对象调用方法通常最好的做法是，将另外一个对象作为该方法的参数。事实上，在这个例子中，Groovy环境下更为实用的方法就是像123.plus(456)一样调用plus方法。

本章所涉及的算术操作都力求使用简单直观的风格。然而，在学习中，用恰当的形式来描述才是最重要的，本书将在附录C中更详细论述这点。

2.1 数值

Groovy支持整数和浮点数。整型值没有分数部分。浮点数是包含十进制小数部分的十进制数。

整数可能是正数、负数或者零。12345、-44和0是常见的整数形式。就像前面说到的那样，它们都是Integer类的实例。

有小数部分的数值是BigDecimal类的实例。浮点数的例子如1.23，-3.1415926。请注意，浮点数必须避免以小数点开始，防止出现混淆，比如必须使用0.25，而不是.25来表示浮点数0.25。同样，它的负数也必须使用-0.25来表示。

附录C会更深入讨论这些简单数值的类。

2.2 表达式

Groovy提供大量适用于数值类型的运算符，包括常见的算术运算符、比较运算符、位运算符，以及其他各种类型的运算符。表达式（expression）通常用来描述某些计算行为，由运算符和操作数组成。算术运算符（arithmetic operator）包括加法（+）、减法（-）、乘法（*）和除法（/）。Groovy同样也支持取模运算符，用百分号表示（%）；取模运算用来计算两个整数相除的余数。表2-1列举适用于整数的各种算术运算符。

表2-1 整数运算符

表达式	调用方法	结果	表达式	调用方法	结果
5 + 3	5.plus(3)	8	5 / 3	5.divide(3)	1.6666666667
5 - 3	5.minus(3)	2	5 % 3	5.mod(3)	2
5 * 3	5.multiply(3)	15			

请注意，两个整数的除法运算通常会产生一个浮点数，即使其结果可能是一个整数。举例来说，表达式6/3的结果是浮点数2.0，而不是整数2。

上面的这些运算符也适用于两个浮点数的运算，对应结果如表2-2所示。接下来的表会说明两个浮点数的取模操作。

表2-2 浮点数运算符

表达式	调用方法	结果	表达式	调用方法	结果
5.0 + 3.0	5.0.plus(3.0)	8.0	5.0 * 3.0	5.0.multiply(3.0)	15.0
5.0 - 3.0	5.0.minus(3.0)	2.0	5.0 / 3.0	5.0.divide(3.0)	1.6666666667

同上，除了取模运算符之外，Groovy的算术运算也适用于整数和浮点数的组合。表2-3给出一些范例。

表2-3 混合运算

表达式	调用方法	结果	表达式	调用方法	结果
5 + 3.2	5.plus(3.2)	8.2	5 * 3.2	5.multiply(3.2)	16.0
5.6 + 3	5.6.plus(3)	8.6	5.6 * 3	5.6.multiply(3)	16.8
5 - 3.2	5.minus(3.2)	1.8	5 / 3.2	5.divide(3.2)	1.5625
5.6 - 3	5.6.minus(3)	2.6	5.6 / 3	5.6.divide(3)	1.8666666667

如上述几个表所示，不管整数组合，还是整数和浮点数组合，除法运算通常都会得到相同的结果。下面所有组合的结果都是2.6：

```
13.0 / 5
13 / 5.0
13 / 5
```

为了获得两个整型值相除的整数部分，必须调用intdiv方法：

```
13.intdiv(5)
```

该表达式的结果是整数2。

使用取模运算符(%)可以得到两个整数操作数相除的余数，因此，

```
13%5    结果值为3
15%5    结果值为0
```

请注意，对一个浮点数求模，或者对一个含有浮点数参数的整数求模都是非法的。因此，表达式13.0%5.0、13.0%5、13%5.0都会提示mod方法被错误地调用。

2.3 运算符优先级

像通常一样，Groovy表达式也是根据运算符的优先级来计算的。运算符的运算次序或者优先级决定了算术表达式的计算次序。表2-4列举基本算术运算符的运算顺序（如想获得全部的列表，以及有关运算符结合性方面的更深入知识，请参见附录C）。

从表2-4可以看出，乘法、除法和取模运算符具有相同的最高优先级，而加法和减法

运算符具有相同的最低优先级。在同时含有这些运算符的表达式中，将首先执行所有的乘法、除法以及取模运算符，然后才执行余下的加和减运算。因此，表达式

```
2 +3 * 4
```

的结果是14，这是因为要首先执行3乘4，得到12，然后再加上2，最后的结果是14。

附录C详细描述了如何使用结合性来决定表达式（比如 $2+3*4+5$ ）的运算顺序。目前来说，为了保证加法运算能够在乘法运算之前先执行，就必须在表达式 $(2+3)*(4+5)$ 中使用圆括号，表达式的值为54。

表2-4 算术运算符

分类	运算符	范例	结合方向
乘法	$*/\%$	$x*y$	自左至右
加法	$+/-$	$x+y$	自左至右

2.4 赋值

赋值运算符（assignment operator）允许将某个数值赋给一个程序变量。赋值语句的最简单形式是：

```
variable =expression
```

赋值运算符（=）的作用是，先求赋值表达式右边的值，然后将计算的结果赋给它左边的变量。下面是赋值语句的一些例子：

```
interest = principal * rate * time /100
speed = distance /time
totalMinutes = 60 * hours +minutes
count = count + 1
```

最上面的例子是用来计算，在特定时间（time）内，按照特定利率（rate），本金（principal）的利息。第二个例子是，通过传输距离和消耗的时间来计算某个物体的移动速度。第三个例子是，把以小时和分钟表示的时间转换为分钟总数。最后一个例子是，让count变量在当前值的基础上加1。

当在脚本中第一次使用某个变量时，需要使用Groovy关键字def，其用途是声明变量。但是，在后续的赋值过程中再次使用这个变量时，就不需要再次使用这个关键字了。附录C将更深入论述。举个例子，

```
def count = 0          // 声明并初始化变量
count = count + 1     // 在当前值的基础上加1
```

给变量赋予一个变量名后，就可以使用变量名来引用这个变量。这些被程序员创建的变量

名称通常被称为标识符。在Groovy中，标识符的命名规则如下。

标识符必须由字母和数字组成，对大小写敏感，标识符的首字符必须是字母。下划线（_）允许出现在标识符中，以字母看待。标识符绝对不允许是Groovy关键词（参见附录C）。

请注意，当看到age = 25赋值语句时，这说明在前面已经使用def关键字声明了age变量。此时便可以肯定25已经被正确地赋值给变量age。后续的赋值也只不过是简单的改变其值而已。在这里，可以理解为age指向一个整型值（参见2.6节）。

2.5 自增和自减运算符

Groovy也支持两个一元运算符，以实现对某个数字变量的加一或者减一操作。一元运算符是指应用于一个操作数的运算符。通常，它们就是所说的自增运算符++和自减运算符--。比如：

```
value = value +
```

可以写成：

```
value++
```

同样，也可以使用

```
value--
```

来替代

```
value = value - 1
```

严格地说，把自增或者自减运算符放在变量的前面，也经常被称为前置自增或者前置自减运算法，表示对变量先执行加一或者减一运算；如果把自增或者自减运算符放在变量的后面，经常被称作后置自增或者后置自减运算法，表示后执行加一或者减一运算。前置自增运算（preincrement）是首先对该变量执行加一运算，然后将该变量的新值用于其所在的表达式；后置自增运算（postincrement）是出现该变量表达式中使用该变量值执行完该表达式后，再对该变量执行加一运算。前置自减（predecrement）和后置自减（postdecrement）运算与此类似。

举例来说，x++将原来x的值用于其表达式中，然后再执行加一运算。同理，++x将首先执行加一运算，然后在表达式运算中使用其新值。因此，

```
def x = 10
def y = x++ // x的值为11；而y的值却为10
```

和

```
def p = 20
def q = ++p // p的值为21；q的值也为21
```

这两个自增运算符都可以通过调用next方法来实现。因此，x++和x.next()的功能是一样的。同样，自减运算符可以通过调用previous方法来实现。

2.6 对象引用

在赋值语句age = 25中，使用Groovy的动态类型（dynamic typing）功能。age变量所引用值的类型是在其运行过程中，而不是在编译时确定的。动态类型通常会使程序变得更为简洁，这也是Groovy之所以简洁与灵活的主要原因。执行此赋值语句时会创建一个Integer对象，并

将25赋值给它，变量age的引用关系如图2-1所示。

变量与对象之间的关联关系被称为引用。变量引用了该对象所使用的内存部分。任何使用该变量的情况，如在表达式age+22中，将使用引用来获得与该对象相关的对象值。

请考虑下面的表达式：

```
def age = 25
def number = age
```

在Groovy中，变量通常都和对象相关联。因此，第二个赋值语句的作用是，让number变量引用age变量所引用的同一个对象。图2-2是一个用来描述共享（或者别名）的范例。在这个范例中，两个变量均引用相同的对象。

在后续的代码中，如果给age变量赋一个新值，其效果如图2-3所示。在这里，可以发现age变量现在引用了一个不同的对象，而number变量却仍然引用被age赋值首次创建的对象。

最后，设想一下给number变量赋新值的效果。图2-4说明表示值25的那个对象现在不再被任何变量引用。因此，在后面的代码中将不能再次使用它。这是一个内存垃圾（garbage）范例，也就是一个没法被引用的对象。在Groovy中，垃圾收集器最终将收回这些对象所占用的内存空间，并将其内存空间分配给其他对象循环使用。



图2-1 变量与对象引用

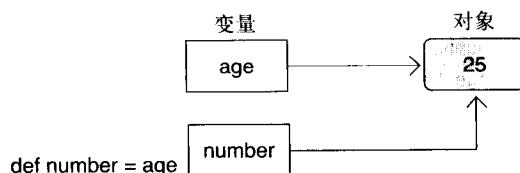


图2-2 共享

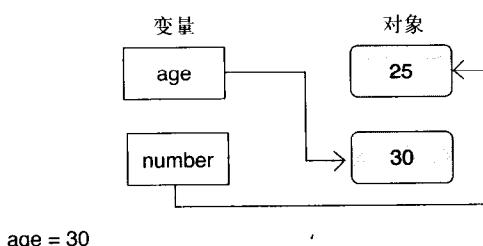


图2-3 重新赋值

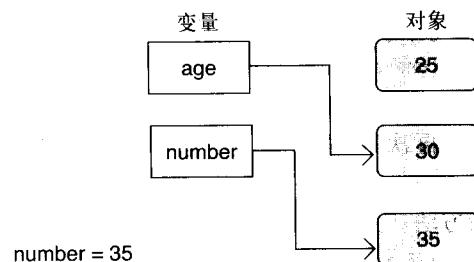


图2-4 内存垃圾

上述这些图说明了，在程序代码的任何地方，可以非常自由地给变量赋新值。更进一步来说，这些新值可以通过已经存在的值而拥有不同的数据类型。这既是Groovy的优点，同时也是它的弱点。举例来说，用户可以自由地将一个String值赋给前面引用Integer的变量，其危险在于：在程序代码中可能不会意识到此问题，并导致其他难以预测的后果。

2.7 关系运算符和等于运算符

有些Groovy控制语句，如if和while控制语句（参见第8章），通常会使用条件（condition）来进行判断。条件决定表达式的值为真或者假。关系运算符、等于运算符和逻辑运算通常被用

于构造条件表达式（附录C会详细讨论逻辑运算符）。

关系运算符如表2-5所示。四种运算符均为二元运算符，每种运算符都使用两个算术表达式作为操作数，并且得出布尔型结果true或者false。true和false都是Boolean类的实例。这些运算符都是通过调用compareTo（表2-5）方法来实现的。举例来说，`a < b`是通过`a.compareTo(b) < 0`来实现的。在a小于b的情况下，`compareTo`方法返回-1；在a大于b的情况下返回+1；在两者相等的情况下返回0。这个方法通常被用做值排序的基础。

下面是一些使用关系运算符的表达式例子：

```
number < 0           // number是否是负数?
age >= 65            // 判断是否是老年人?
index <= limit-1     // 是否达到上限?
```

由于关系运算符的优先级低于算术运算符（参看附录C），最后一个例子应该被解释为`index <= (limit-1)`。

表2-6提到了等于运算符“==”和“!=”。再次说明一下，它们也都是二元运算符，其运算所得出的结果值也是布尔值true或者布尔值false。这两个运算符都是通过equal方法实现的。`compareTo`运算符被`<=>`所替代，其优先级和前两个运算符相同。

一些范例：

```
def forename = "Ken"
def surname = "Barclay"
forename == "Ken"          //结果为true
surname != "Barkley"       //结果为true
```

再次声明一下，这些等于运算最终都是通过调用某个方法实现的。例如，条件语句`forename == "Ken"`实际上是通过`forename.equals("Ken")`来实现的。`equals`方法是String类的方法，其作用是判断两个值是否相同。类似的，对于下面两个赋值语句：

```
def age = 25
def number = 25
```

条件语句：

```
age == number
```

其结果是布尔值true。在这里，`age.equals(number)`是通过Integer类的equals方法来求值的。

表2-5 关系运算符

表达式	调用方法	结果
<code>5 < 3</code>	<code>5.compareTo(3) < 0</code>	假
<code>5 <= 3</code>	<code>5.compareTo(3) <= 0</code>	假
<code>5 > 3</code>	<code>5.compareTo(3) > 0</code>	真
<code>5 >= 3</code>	<code>5.compareTo(3) >= 0</code>	真

表2-6 等于运算符

表达式	调用方法	结果
<code>5 == 3</code>	<code>5.equals(3)</code>	假
<code>5 != 3</code>	<code>! 5.equals(3)</code> //参见附录C	真
<code>5 <=> 3</code>	<code>5.compareTo(3)</code>	+1

2.8 习题

1. 在Groovy中，下面哪些值是合法的？

- (a) -123 (b) .123 (c) 0.123 (d) 10.0e4 (e) 10E4

如果是合法的值，请说明其类名。

2. 请根据优先级和结合性规则，计算下面的值：

(a) `def m = 5`

`12*m`

(b) `m = 5`

`def j = 2`

`12*m/j`

(c) `def f = 1.2`

`(f+10)*20`

(d) `def g = 3.4`

`f = 1.2`

`12*(g-f)`

3. 下面哪些是合法的Groovy标识符，请说明原因。

(a) June

(b) a\$

(c) b

(d) _Z

(e) name1

(f) public

4. 用表2-1的形式设计四个图来说明下面的结果：

```
def value = 42
def anotherValue = value
value = 99
anotherValue = 50
```

在这个上下文中，垃圾收集器起到怎样的作用？

5. 请根据优先级和结合性规则，计算下面的值：

(a) `def x = 12`

`def y = 2`

`x + 3 <= y*10`

(b) `x = 20`

`y = 2`

`x + 3 <= y*10`

(c) `x = 7`

`y = 1`

`x + 3! = y*10`

(d) `x = 17`

`y = 2`

`x + 3 == y*10`

(e) `x = 100`

`y = 5`

`x + 3 > y*10`

第3章 字符串和正则表达式

前面的章节集中讨论了数值以及适用于数值的基本算术运算。本章将学习String（字符串）——用来表示文本信息的字符序列。这种字符序列可能表示某人或者某文件的名称。同样，它也可能代表一个银行账号，或者某程序语言的名字。附录D会更详细地讨论如何使用String和正则表达式。

3.1 字符串字面值

通过使用引号将字符串文本封装起来，就可以很容易获得String字面值。Groovy为表示字符串提供了数种不同的方式。举例来说，在Groovy中，可以使用单引号（'）双引号（"）或者三引号（"""）来封装字符串。而且，在Groovy中使用三引号表示的字符串可以包含多行文本。表3-1列出部分字符串。

请注意，在第二个范例中双引号嵌套在单引号中的用法。同样，在第三个范例中，单引号被嵌套在双引号之中。其实在这两个范例中，都可以使用转义字符反斜线（\）来进行转义。如，'He said \"Hello\"!'。最后一个范例是使用三引号包含多行文本的字符串范例，包含其封装范围内所有文本行的所有字符。

使用单引号封装的字符串的值就是所列出的字符序列本身。而另外两种形式的字符串的值有可能会进一步被解释。任何包含在解释型字符串中的\${expression}都将被求值，其结果是字符串的一部分。下面的范例说明了其效果：

```
def age =25
'My age is ${age}'           //My age is $age
"My age is ${age}"           //My age is 25
"""My age is ${age}""        //My age is 25
"My age is \$age"            //My age is $age
```

从上面的范例可以看到，在第一个范例由于使用了单引号，\${age}没有被解释。同样，在美元符号（\$）前使用了反斜线（\）转义字符后，也没有解释\${age}。通用的原则是，只在字符串需要被解释的时候使用双引号，在其他情况下字符串使用单引号。

3.2 字符串索引和索引段

字符串是顺序排列的字符集合，因此可以通过单个字符在字符串中的位置来获取该字符，

表3-1 字符串

字面值	描述
"	空字符串
'He said "Hello"!'	单引号（嵌套双引号）
"He said 'Hello'! "	双引号（嵌套单引号）
'''one two three'''	三引号
''' "Spread over four lines'''	多行文本使用三引号

这就是所说的索引 (index) 位置。请注意，可以使用索引来指定单个字符或指定字符集合的位置；使用任一方式都能返回一个字符串值。字符串的索引从零开始，止于该字符串的长度值减一。Groovy同样也支持负数索引，但是其顺序是从字符串的末尾开始往前计算。子字符串通常可以用索引段 (slicing) 来表示。使用索引段可以提取字符串的子集。

下面是字符串对象greeting的索引和索引段范例：

```
def greeting ='Hello world'
greeting [4 ]           //o index from start
greeting [-1 ]          //d index from end
greeting [1..2 ]         //e1 slice with inclusive range (see Chapter 4)
greeting [1..<3 ]        //e1 slice with exclusive range (see Chapter 4)
greeting [4..2 ]         //o11 backward slice
greeting [4,1..6 ]       //oew selective slicing
```

请注意，1..2和1..<3这两种索引段表示方法，这种表示方法也被称为范围 (range)，将在第4章中更深入讨论。1..2表示索引范围从1开始到2结束，包括边界，也就是2；1..<3是不包含边界的索引范围表示方法，表示索引范围从1开始，到小于3的整数结束。

3.3 基本操作

基本的字符串操作包括两个字符串的合并、字符串的复制，以及计算字符串的长度。`minus`方法（或者重载运算符`-`）的作用是删除首次出现的子字符串。`count`方法统计某个子字符串出现的次数，而`contains`方法则用来判断某个字符串是否包含指定的子字符串。下面是一些范例：

```
def greeting ='Hello world'
'Hello' +'world'           //Hello world      concatenate
>Hello'*3                  //HelloHelloHello    repeat
greeting -'o world'        //Hell             remove first occurrence
greeting.size()            //11              synonymous with length
greeting.length()          //11              synonymous with size
greeting.count('o')         //2
greeting.contains('ell')   //true
```

请注意，前三个范例演示了前一章节所述的运算符重载方法。举例来说，`'Hello' + 'world'` 表示`"Hello".plus("world")`。`"Hello"`字符串对象调用了`plus`方法，其参数为字符串`"word"`。

Groovy字符串是不可变的，它们在任何地方都不可能改变。可以通过索引、索引段以及合并其他字符串的方式创建一个新`String`对象。因此，`greeting-'o world'`表达式演示了新`String`对象`"Hell"`的产生过程，但字符串对象`greeting`本身仍没有变化。

3.4 字符串方法

`String`类有许多实用的`String`对象处理方法。表3-2列出了一些常用的方法。函数原型/说明栏列出该方法的名称、数量、参数类型、返回值的类型，以及该方法的功能简要描述。

Groovy通过包含附加方法的方式扩展了Java开发工具（JDK）中的类，附录B说明了扩展

的具体过程。Java类String包含如concat、endsWith以及length（参见<http://java.sun.com/j2se/1.5.0/docs/api/index.html>）等在内的一些方法。Groovy开发工具（GDK）详细说明了String类的一些新增方法，如center、getAt、leftShift等（参见<http://groovy.codehaus.org/groovy-jdk.html>）。这些新增的功能在表3-2中用“*”号标注。

表3-2 String方法

方法名称	函数原型/说明
center *	String center(Number numberOfChars) 返回一个长度为numberOfChars、其左边和右边均使用空格填充的新字符串
center *	String center(Number numberOfChars, String padding) 返回一个长度为numberOfChars、其左边和右边均使用padding填充的新字符串
compare- ToIgnoreCase	int compareToIgnoreCase(String str) 按词典顺序比较两个字符串，忽略大小写
concat	String concat(String str) 在当前字符串后加上str字符串
eachMatch *	void eachMatch(String regex, Closure clos) 判断给定字符串的子字符串是否与正则表达式regex（参见下一节）匹配。传递给闭包（参见第9章）的对象是一个匹配成功的字符串数组
endsWith	Boolean endsWith(String suffix) 测试字符串是否以给定的后缀结尾
equalsIgnore- Case	Boolean equalsIgnoreCase(String str) 将当前字符串和另一个字符串相比较，忽略大小写
getAt *	String getAt(int index) String getAt(IntRange range) String getAt(Range range) 字符串的下标运算。
indexOf	Int indexOf(String str) 返回给定子字符串在当前字符串中首次出现的索引值
leftShift *	StringBuffer leftShift(Object value) 重载leftShift操作符，以提供一种将多个字符串对象相加，并返回一个新字符串的更为简单的方法
length	int length() 返回字符串的长度
matches	Boolean matches (String regex) 告诉字符串是否匹配给定正则表达式
minus *	String minus(Object value) 删除字符串中的value部分
next *	String next() 此方法被String类的++操作符调用，它用来增加给定字符串的最末位字符
padLeft *	String padLeft(Number numberOfCharacters) 在字符串的左侧使用空格字符填充
padLeft *	String padLeft(Number numberOfCharacters, String padding) 在字符串的左侧使用padding填充
padRight *	String padRight(Number numberOfCharacters) 在字符串的右侧使用空格字符填充
padRight *	String padRight (Number numberOfCharacters, String padding) 在字符串的右侧使用padding填充

(续)

方法名称	函数原型/说明
plus *	String plus(Object value) 字符串相加
previous *	String previous() 此方法被String类的--操作符调用, 它用来删除给定字符串的最末位字符
replaceAll	void replaceAll(String regex, Closure clos) 替换所有与正则表达式相匹配的闭包中的文本值
reverse *	String reverse() 创建当前字符串的逆序字符串
size *	int size() 返回字符串长度
split *	String[] split(String regex) 使用与给定的正则表达式相匹配的子字符串将字符串分隔为多个字符串
substring	String substring(int beginIndex) 返回一个值为当前字符串的子字符串的新字符串
substring	String substring(int beginIndex, int endIndex) 返回一个值为当前字符串的子字符串的新字符串
toCharacter *	Character toCharacter() 字符串类型转换
toDouble *	Double toDouble()
toFloat *	Float toFloat()
toInteger *	Integer toInteger()
toLong *	Long toLong()
toList *	List toList() 将给定的字符串转换成一个由单个字符组成的字符串列表
~toLowerCase	String toLowerCase() 将当前字符串对象的所有字符转换为小写
toUpperCase	String toUpperCase() 将当前字符串对象的所有字符转换为大写
tokenize *	List tokenize() 使用空格作为字符串的分隔符
tokenize *	List tokenize(String token) 使用给定的token参数作为字符串的分隔符

下面是一些例子 (空格字符使用□符号表示):

```
'Hello'.compareToIgnoreCase('hello')           //0
'Hello'.concat('world')                      //Hello world
'Hello'.endsWith('lo')                       //true
'Hello'.equalsIgnoreCase('hello')             //true
'Hello'.indexOf('lo')                        //3
'Hello world'.indexOf('o',6)                 //7
'Hello'.matches('Hello')                     //true
'Hello'.matches('He')                        //false
'Hello'.replaceAll('l','L')                  //HeLLo
'Hello world'.split('l')                     //'He','o wor','d'
```

```

'Hello'.substring(1)           //ello
'Hello'.substring(1,4)         //ell
'Hello'.toUpperCase()          //HELLO
def message ='Hello'
message.center(11)             //□□□Hello□□□
message.center(3)              //Hello
message.center(11,'#')         //###Hello###
message.eachMatch('.'){ch ->
    println ch }                on separate lines
message.getValueAt(0)          //H
message.getValueAt(0..<3)       //He
message.getValueAt([0,2,4 ])     //Ho
message.leftShift('world')      //Hello world
message <<'world'            //Hello world
message.minus('ell')           //Ho
message -'ell'                 //Ho
message.padLeft(4)             //Hello
message.padLeft(11)             //□□□□Hello
message.padLeft(11,'#')         //#####Hello
message.padRight(4)             //Hello
message.padRight(11)             //Hello□□□□□
message.padRight(11,'#')        //Hello#####
message.plus('world')          //Hello world
message +'world'               //Hello world
message.replaceAll('[a-z ]'){ch ->
    ch.toUpperCase()}
message.reverse()               //olleH
message.toList()                //['H','e','l','l','o']
def message ='Hello world'
message.tokenize()               //['Hello','world']
message.tokenize(' ')            //['He','o wor','d']

```

请注意，左移运算符“`<<`”重载了`leftShift`方法。第2章已经简要介绍过操作符重载，本书将在附录I.4节中详细介绍其用法。

`Tokenize`方法能将某个字符串分隔为一个字符串列表（参见第4章）。该方法的第一个版本使用空格符作为分隔符，第二个版本使用给定的`String`参数作为分隔符。`Split`方法基于与正则表达式（参见3.6节和附录D）的匹配情况，对字符串进行分割，并返回一个字符串数组。

3.5 比较字符串

Groovy支持`String`对象之间的比较方法。如前面章节所述，该操作是指定方法的重载版本。这样，在比较两个`String`对象时，就可以使用`str1==str2`，这比使用`str1.equals(str2)`要方便得多。同样，`str1<=>str2`就代表`str1.compareTo(str2)`。如果`str1`在`str2`之前，该方法返回值为`-1`，如果`str1`在`str2`之后，则返回值为`+1`；如果`str1`和`str2`相同，则返回值为`0`。这个方法通常被用做`String`对象排序的基础。下面是一些`String`比较的范例：

```
'ken'<=>'ken'           //0 same
'ken'<=>'kenneth'        //-1 before
'ken'<=>'Ken'            //1 after
'ken'.compareTo('Ken')    //>0 after
```

字符串比较采用词典顺序，大写字母排在小写字母之前。因此，如最后两个范例所示，'ken'在'Ken'之后，这是由于按照字典顺序，'K'在'k'之前。Groovy采用Unicode（参见<http://www.unicode.org/>）字符集，这使得Groovy程序更容易国际化。

3.6 正则表达式

正则表达式（regular expression）是在文本中寻找子字符串的一种模式。从表3-2可以看出，String类提供多个允许使用正则表达式对String对象执行操作的方法。如果给定的正则表达式与String对象匹配，则matches方法返回true。举例来说，"abc".matches("abc")返回true，而"abc".matches("bc")则返回false。replaceAll方法（参见前面部分内容）用指定的闭包（参见第9章）替换String对象中所有与regex正则表达式匹配的值。

Groovy支持使用~"regex"来定义正则表达式。双引号中的文本表示正则表达式。通过下面的方法可以创建一个正则表达式：

```
def regex =~'cheese'
```

在if语句或者while语句（参见第8章）中，当Groovy操作符"=~"作为一个谓词（表达式返回一个布尔值）出现时，左边的操作数String对象将和右边的正则表达式匹配。下面所有表达式的结果均为true：

```
'cheesecake' =~ 'cheese'
!('cheesecake' =~ 'fromage')
'cheesecake' =~ regex
```

精确匹配符“==~”需要精确匹配。因此，下面表达式结果为false：

```
'cheesecake' ==~ 'cheese'
```

在正则表达式中，有两个特殊的位置标识符（positional characters），它们是脱字符号（^）和美元符号（\$），分别表示某行的开始和结尾：

```
def rhyme ='Humpty Dumpty sat on a wall'
rhyme =~ ^'Humpty'           //true
rhyme =~ '^wall$'           //true
```

正则表达式同样也包含数量（quantifiers）。加符号（+）表示在表达式中位于它前面的字符出现一次或者多次，星号（*）表示出现零次或者多次。“{”和“}”用来匹配位于“{”符号之前指定次数的字符，下列表达式的结果均为true：

```
'aaaaab' =~ 'a * b'
'b' =~ 'a * b'
'aaacd' =~ 'a * c?d'
'aaad' =~ 'a * c?d'
'aaaaab' =~ 'a{5}b'
!( 'aab' =~ 'a{5}b' )
```

在正则表达式中，点符号（.）能代表任意字符，因而称其为通配符（wildcard character）。这样，在需要匹配一个实际的点字符的时候，事情就会变得非常复杂。下列表达式的结果均为true：

```
rhyme =~'.all'
'3.14' =~'3.14'           //pattern:3 followed by any character followed by 14
'3X14' =~'3.14'
'3.14' =~'3 \\.14'        //pattern:3.14 literally!
!('3X14' =~'3 \\.14')
```

使用反斜杠字符时要加倍小心。在普通的String对象中出现时，通常被当作转义字符，因此“\\\"表示单个的反斜杠字符。这样，在正则表达式中需要使用“\\\\\\\"来表示一个反斜杠字符。使用四个反斜杠字符来表示一个反斜杠字符的方法通常会使人困惑不解。

正则表达式还可以包含字符类（character classes）。一组字符集可以通过使用[和]元字符（metacharacters）来表示，如[aeiou]。对于字符和数字，可以使用破折号分隔符，如使用[a-z]或者[a-zA-M]表示。字符类的余数可以通过脱字符号来表示，如[^a-z]表示除所指定字符之外的任意字符。下列表达式的结果均为true：

```
rhyme =~'[HD]umpty'
!(rhyme =~'[hd]umpty')
!(rhyme =~'[ ^ HD]umpty')
```

最后，可以通过组合正则表达式，生成更复杂的正则表达式。组合方式使用“（”和“）”元字符（metacharacters）。正则表达式“(ab)*”表示ab出现任意次。同样，也可以使用选择符（用|表示）来组合一个或者多个可能的正则表达式。正则表达式“(alb)*”表示a、b或者ab混合出现任意次。下列表达式的结果均为true：

```
'ababab'==~'(ab)*'
!('ababa'==~'(ab)*')
'ababc'==~'(ab)* c'
'aaac'==~'(a|b)* c'
'bbbc'==~'(a|b)* c'
'ababc'==~'(a|b)* c'
```

Groovy同样也支持使用“/”分隔符的模式，因此不需要重复所有的反斜线符号。举例来说，表达式：

```
def matcher ="\$abc."=~\\\$(.*)\\.
```

也可以使用下面的表达式表示：

```
def matcher ="\$abc."=~/\$(.*)\./
```

3.7 习题

1. 计算下列表达式的值：

- (a) "Hello"+ "world"
- (b) "12" +"34"
- (c) "1" + "0"

2. 计算下列表达式的值：

- (a) "Hello".length()
- (b) "".length()

3. 假定将一个String变量str定义为：

```
def str = "Hello world"
```

计算下列表达式的值：

- (a) str.indexOf("or")
- (b) str.indexOf("Or")
- (c) str.lastIndexOf("o")
- (d) str.lastIndexOf("or")

4. 假定有如下定义：

```
def str = "Groovy, Groovy, Groovy"
```

计算下列表达式的值：

- (a) str.length()
- (b) str.indexOf("o")
- (c) str.lastIndexOf("o")
- (d) str.indexOf("o", 5)
- (e) str.lastIndexOf("o", 5)
- (f) str.indexOf("ov", str.length() - 10)
- (g) str.lastIndexOf("ov", str.length() - 4)
- (h) str.indexOf("o", str.indexOf("ro"))

5. 假定有如下定义：

```
def str = "Groovy programming"
```

计算下列表达式的值：

- (a) str.length()
- (b) str.substring(7, 14)
- (c) str.substring(1, str.length() - 1)
- (d) str.endsWith("ming")

6. 计算下列表达式的值：

- (a) 'Groovy' =~ 'Groovy'
- (b) 'Groovy' =~ 'oo'
- (c) 'Groovy' =~ '^Groovy'
- (d) 'Groovy' =~ 'oo'
- (e) 'Groovy' =~ '^G'
- (f) 'Groovy' =~ 'G\$'
- (g) 'Groovy' =~ 'Gro*vy'
- (h) 'Groovy' =~ 'Gro{2}vy'

第4章 列表、映射和范围

本章将介绍列表（list）、映射（map）和范围（range）。它们都是引用其他对象的集合。列表和映射能够引用不同类型的对象，范围则表示整型值的集合。列表和映射能够动态地增长。列表中的每个对象都由一个整型值索引来指定；而映射集合则能够通过任意类型的值进行索引。由于这些集合所包含对象的类型是随意的，列表中的元素有可能是一个映射，映射中的元素也有可能是一个列表。这样，就可以构造出任意复杂的数据结构，本书将在第6章和后续章节中详细说明。

4.1 列表

列表是一种用来存储数据项集合的数据结构。在Groovy中，列表可以有序地保存所引用的对象。列表中对象所占用的位置也是有序的，并且能够通过一个整数索引来标识。列表字面值是一系列包含在方括号中的对象集合，这些对象用逗号分隔。表4-1是一些简单的列表字面值范例。

为了处理列表中的数据，必须能够访问列表中的单个元素。Groovy列表使用索引操作符[]来标识元素值。列表的索引从0开始，它指向列表的第一个元素。下面是名为numbers的列表对象和访问列表元素的一些简单范例：

```
def numbers = [11, 12, 13, 14]      // 列表含有四个元素
numbers [0]                         // 第0个元素的值为11
numbers [3]                         // 第3个元素的值为14
```

如果整型索引为负数，则其引用的元素从列表末端开始向前移动，因此，

```
numbers [-1]                      // 值为14
numbers [-2]                      // 值为13
```

索引段同样也可以用于列表，比如：

```
[11, 12, 13, 14][2]              // 值为13
```

再次强调指出，“[]”操作符就是List类中定义的getAt方法（参见4.2节）。因此，当使用numbers[3]引用列表元素时，必须认识到，实际上是List对象numbers调用了getAt方法，其参数为3，即numbers.getAt(3)。

另外，也可以通过索引范围（在本章后面进一步说明）来操作列表。包含边界的索引范围如start..end所示，它将产生一个新的列表对象，其初始值从原始列表对象的start索引位置开始，

表4-1 列表字面值

范例	说 明
[11, 12, 13, 14]	整型值列表
['Ken', 'John', 'Andrew']	字符串列表
[1, 2, [3, 4], 5]	嵌套列表
['Ken', 21, 1.69]	含有不同类型对象的列表
[]	空列表

到原始列表对象的end索引位置结束。而不包含边界的索引范围如start..<end所示，其值不包含列表中end索引位置的元素。一些使用范围索引的范例如下所示：

```
numbers [0..2]           // 返回列表[11, 12, 13]
numbers [1..<3]          // 返回列表[12, 13]
```

使用列表的索引操作符同样也能够为列表赋新值。赋值语句右侧的值将替换赋值语句左侧给定的索引位置的元素值，索引值只能是一个整型表达式；如果赋值语句右侧本身就是一个列表对象，则使用列表对象替代左侧的索引元素值。

```
numbers [1] = 22          // 返回列表[11, 22, 13, 14]
numbers [1] = [33, 44]     // 返回列表[11, [33, 44], 13, 14]
```

`putAt`方法（参见表4-2）提供了赋值语句的功能。

通过`<<`运算符（即`leftShift`方法），可以把一个新元素值追加到列表的最后。如：

```
numbers << 15            // 返回列表[11, [33, 44], 13, 14, 15]
```

同样，使用`+`操作符（即`plus`方法）可以连接两个列表：

```
numbers = [11, 12, 13, 14]    // 列表含有四个元素
numbers + [15, 16]           // 返回列表[11, 12, 13, 14, 15, 16]
```

`-`操作符（即`minus`方法）从列表中删除元素：

```
numbers = [11, 12, 13, 14]    // 列表含有四个元素
numbers -[13]                 // 返回列表[11, 12, 14]
```

4.2 列表方法

在Groovy中，借助于List类提供的方法，可以使列表运算变得较为容易。使用List类可以大大减少操作列表的应用程序编程量。表4-2列出一些常用的List方法。请注意，带有星号的方法是GDK新增的方法。

表4-2 List方法

方法名称	函数原型/说明
add	boolean add(Object value) 在当前列表末尾追加一个新值
add	void add(int index, Object value) 在当前列表给定索引位置插入一个新值
addAll	boolean addAll(Collection values) 在当前列表末尾追加values值
contains	boolean contains(Object value) 如果该列表包含给定值则返回true。
flatten *	List flatten () 使当前列表元素形式一致，并返回一个新列表
get	Object get (int index) 返回当前列表中指定位置的元素值
getAt *	Object getAt (int index) 返回当前列表中指定位置的元素值

(续)

方法名称	函数原型/说明
getAt *	List getAt (Range range) 返回一个新列表，其值为当前列表中给定范围的子列表
getAt *	List getAt(Collection indices) 返回一个新列表，其值为当前列表中给定索引值的子列表
intersect *	List intersect (Collection collection) 返回一个新列表，其值为原始列表和输入列表的所有元素
isEmpty	boolean isEmpty() 如果当前列表没有任何元素则返回true
leftShift *	Collection leftShift (Object value) 重载左移运算符，提供一个向列表添加新值的简单方法
minus *	List minus (Collection collection) 生成一个新列表，其元素由原始列表中那些不包含在collection参数中的值组成
plus *	List plus (Object value) 生成一个新列表，其元素由原始列表中的元素和value参数值组成
plus *	List plus (Collection collection) 生成一个新列表，其元素由原始列表中的元素和给定的collection参数值组成
pop *	Object pop() 删除当前列表中最后一个值元素
putAt *	void putAt (int index, Object value) 赋值语句的左侧支持下标运算符
remove	Object remove (int index) 删除当前列表中给定位置的元素
remove	boolean remove (Object value) 删除在当前列表中首次出现的给定元素
reverse *	List reverse() 生成一个新列表，其值为原始列表的元素的倒序
size	int size() 获得当前列表的元素个数
sort *	List sort() 返回原始列表所有元素的排序副本

下面的范例演示了这些List方法的使用方法及其效果：

```
[11, 12, 13, 14].add(15)                                // 返回值为 [11, 12, 13, 14, 15]
[11, 12, 13, 14].add(2, 15)                            // 返回值为[11, 12, 15, 13, 14]
[11, 12, 13, 14].add([15, 16])                          // 返回值为[11, 12, 13, 14, 15, 16]
[11, 12, 13, 14].get(1)                                // 返回值为12
[11, 12, 13, 14].isEmpty()                             // 返回值为false
[14, 13, 12, 11].size()                               // 返回值为4
[11, 12, [13, 14]].flatten()                           // 返回值为[11, 12, 13, 14]
[11, 12, 13, 14].getAt(1)                             // 返回值为12
[11, 12, 13, 14].getAt(1..2)                          // 返回值为[12, 13]
[11, 12, 13, 14].getAt([2, 3])                        // 返回值为[13, 14]
[11, 12, 13, 14].intersect([13, 14, 15])           // 返回值为[13, 14]
[11, 12, 13, 14].pop()                                // 返回值为14
```

```
[11, 12, 13, 14].reverse()          // 返回值为[14, 13, 12, 11]
[14, 13, 12, 11].sort()            // 返回值为[11, 12, 13, 14]
```

请注意下面的代码：

```
def numbers = [11, 12, 13, 14]
numbers.remove(3)
numbers.remove(13)
```

第一个remove语句调用remove(int index)方法寻找和删除索引位置为3的元素值。该语句能够实现其想要的结果。然而，在调用第二个remove方法时，程序设计者原本想通过使用remove(Object value)方法删除值为13的元素，但程序仍尝试着调用第一个remove方法来删除数据，这样将出现越界(out-of-bound)异常，并导致调用失败。如下面的代码所示，使用字符串参数的方式来删除列表中所包含的字符串，则能够实现所期望的效果，Groovy能够正确地识别所需调用的方法，并删除数据。

```
def names = ['Ken', 'John', 'Sally', 'Jon']
names.remove(3)
names.remove('Ken')
```

4.3 映射

映射（也就是大家熟知的组合数组、词典、表格和散列）是一种引用对象的无序集合。映射中的所有元素都可以通过关键字访问，它使用的关键字可以是任意类型的。当向一个映射插入值时，需要同时提供两个参数：关键字及其相应的值。在映射中，使用索引关键字即可检索出相应的值。表4-3列出一些映射字面值范例，把关键字（值对）放在方括号中，并使用逗号分割。

表4-3 映射

范例	说明
['Ken' : 'Barclay', 'John' : 'Savage']	名/姓集合
[4 : [2], 6 : [3, 2], 12 : [6, 4, 3, 2]]	整型关键字及其约数列表
[:]	空映射

在映射字面值中，如果某个元素的关键字是一个变量名，那么它将被解释成一个String值。在下面的范例中：

```
def x = 1
def y = 2
def m = [x : y, y : x]
```

于是，m就是映射：

```
m = ['x' : 2, 'y' : 1]
```

映射的单个元素一般通过下标运算符（通过getAt方法实现，参见4.4节）访问。在这里，索引值可以是任意类型的对象，并且代表某个关键字，其返回值为与关键字相对应的值或者空值。下面是names和divisors映射对象及其简单索引方法的一些范例：

```

def names = ['Ken' : 'Barclay', 'John' : 'Savage']
def divisors = [4 : [2], 6 : [3, 2], 12 : [6, 4, 3, 2]]
names['Ken']           // 'Barclay'
names.Ken              // 'Barclay'
names['Jessie']        // null
divisors[6]            // [3, 2]

```

和列表一样，该索引同样也是通过getAt方法（参见下一节）实现的。同样，putAt方法也支持在赋值语句左侧使用索引，如：

```

divisors[6] = [6, 3, 2, 1]           // [4 : [2], 6 : [6, 3, 2, 1],
                                     // 12 : [6, 4, 3, 2]]

```

必须注意到那些映射的关键字也都是对象。如前面给出的names映射，其关键字也都是String对象。同样，divisors映射的关键字也都是Integer对象。因此，使用映射实现其功能是完全可行的：

```

def careful = [ 1 : 'Ken', '1' : 'Barclay']
careful[1]           // Ken
careful['1']         // Barclay

```

上面映射范例的第一个项的关键字是Integer类型，值为1；而第二项的关键字是String类型，值为'1'。在接收用户输入时，可以通过将第一个关键字作为整数，将第二个关键字作为字符串对象处理的方式达到此目的。但如果这不是我们的真正意图，就必须格外谨慎了。

4.4 映射方法

Map类所支持的许多方法使映射对象的处理过程变得非常简单，Map类大大减少了那些各组对象之间存在相关联系的应用程序的编程工作。表4-4列出并说明了一些较为常用的映射方法。那些使用了星号标记的方法能够在GDK中使用。

表4-4 Map方法

方法名称	函数原型/说明
containsKey	boolean containsKey(Object key) 当前映射是否包含key关键字
get	Object get(Object key) 在当前映射中查询key关键字，并返回与之相对应的值。如果在当前映射中没有与该关键字相对应的值则返回空
get *	Object get(Object key, Object defaultValue) 在当前映射中，查询key关键字，并返回与之相应的值。如果在当前映射中没有与该关键字相对应的值，则返回defaultValue默认值
getAt *	Object getAt(Object key) 支持使用下标操作符的方法
keySet	Set keySet() 获取当前映射的一组关键字
put	Object put(Object key, Object value) 在当前映射中，使给定的value值和给定的key关键字产生对应关系。如果在该映射中先前已经包含当前关键字的映射，新值将替换旧映射值

(续)

方法名称	函数原型/说明
putAt *	Object putAt(Object key, Object value) 允许映射使用下标操作符
size	int size() 返回当前映射中关键字与值映射的数目
values	Collection values() 返回包含在当前映射中所有值的集合

下面的范例演示了这些Map方法的使用方法及其功能：

```
def mp = ['Ken' : 2745, 'John': 2746, 'Sally' : 2742]
mp.put('Bob', 2713)           // [Bob:2713, Ken:2745, Sally:2742, John:2746]
mp.containsKey('Ken')         // true
mp.get('David', 9999)         // 9999
mp.get('Sally')               // 2742
mp.get('Billy')               // null
mp.keySet()                  // [David, Bob, Ken, Sally, John]
mp.size()                     // 4
mp['Ken']                     // 2745
```

请注意values方法返回包含在映射中值的一个集合，通常将其作为一个列表看待比较好。使用下面的代码可以非常容易地实现其功能：

```
mp.values().asList()
```

4.5 范围

范围是表达特定序列值的一种简略方法。它通过序列中第一个值和最后一个值表示，范围还具有包含和不包含关系。一个具有包含关系的范围，将包含从第一个值开始到最后一个值为止的所有值；而一个具有不包含关系的范围，则包含除了最后一个值之外的所有值。下面是一些包含范围的范例：

```
1900..1999                   // 20世纪(包含边界)
2000..<2100                    // 21世纪(不包含边界)
'A'..'D'                      // A, B, C和 D
10..1                          // 10, 9, ..., 1
'Z'..'X'                      // Z, Y和 X
```

请注意，包含边界的范围通过“..”表示，而不包含边界的范围则在下边界和上边界之间使用“..<”。范围可以通过字符串或者整型值表示。如上所示，范围也可以通过升序或者降序表示。

范围的上标和下标同样也可以通过整型表达式表示，如：

```
def start = 10
def finish = 20
start..finish + 1              // [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]
```

Groovy也为范围定义了一些方法，如表4-5所示。

表4-5 范围方法

方法名称	函数原型/说明
contains	boolean contains(Object obj) 如果当前范围包含给定的元素则返回true
get	Object get(int index) 返回当前范围内给定位置的元素值
getFrom	Comparable getFrom() 获取当前范围内下标最小的元素值
getTo	Comparable getTo() 获取当前范围内下标最大的元素值
isReverse	boolean isReverse() 当前范围是否为逆序
size	int size() 返回当前范围内元素个数
subList	List subList(int fromIndex, int toIndex) 返回当前范围内给定的fromIndex（包含）到toIndex（不包含）之间的所有值

下面的范例演示了这些范围方法的使用方法及其功能：

```
def twentiethCentury = 1900..1999      // Range literal
def reversedTen = 10..1                 // Reversed Range
twentiethCentury.size()                // 100
twentiethCentury.get(0)                // 1900
twentiethCentury.getFrom()             // 1900
twentiethCentury.getTo()               // 1999
twentiethCentury.contains(2000)        // false
twentiethCentury.subList(0, 5)          // 1900..1904
reversedTen[2]                         // 8
reversedTen.isReverse()                // true
```

欲获得有关列表、映射以及范围等更深入的知识，请参考附录E。

4.6 习题

1. 如何以逆序的方式快速获得列表[14, 12, 13, 11]的所有元素？
2. 求表达式[1, [2, [3, 4]]].flatten()的值，并说明flatten方法是否在嵌套列表中被多次执行？
3. 给定两个列表[11, 12, 13, 14]和[13, 14, 15]，如何获得在第一个列表中出现，而没有在第二个列表中出现的元素值，即[11, 12]？
4. 堆栈是一种后进先出的数据结构，所有操作都在堆栈顶部发生。堆栈中的操作包括（a）PUSH：在堆栈顶部设置一个新值；（b）POP：删除堆栈栈顶值；（c）TOP：提取堆栈栈顶值。
如果使用列表方式实现堆栈，并且堆栈栈顶是列表最后一个元素，请说明对堆栈使用何种操作以实现和列表操作类似的功能。
5. 给定某个列表对象table，请说明下列语句引用了哪个元素值：
 - (a) table[0]
 - (b) table[table.size() - 1]

- (c) `table[table.size().intdiv(2)]` (i) 当当前列表的元素个数为奇数时；
(ii) 当当前列表的元素个数为偶数时。

6. 请说明下面两个表达式的不同之处：

- (a) `table.sort().reverse()`
(b) `table.reverse().sort()`

7. 映射`names = ['Ken' : 'Barclay', 'John' : 'Savage', 'Ken' : 'Chisholm']` 在Groovy中合法吗？表达式`names.ken`的值是什么？

8. 根据4.3节divisors映射值，求下列表达式的值：

- (a) `divisors.containsKey(8)`
(b) `divisors[6].sort()`
(c) `divisors[6].intersect(divisors[12])`

9. 一家软件机构能够开发Groovy、Java和C#项目，每个项目组都由一个或多个程序员组成，其中的每个程序员可能参与多个项目。举例来说，下列代码说明Ken、John和Jon同属于Groovy项目：

```
def softwareHouse = ['Groovy' : ['Ken', 'John', 'Jon'],
                     'Java' : ['Ken', 'John'],
                     'C#' : ['Andrew']
]
```

请回答下列问题：

- (a) Groovy项目组共有多少个成员？
(b) 哪些成员同时在Groovy和Java项目中工作？
(c) 哪些成员在Groovy项目中工作，却不在Java项目中工作？

10. 很多大学都由多个系组成，每个系负责完成一门或多门专业的教学任务。举例来说，下列代码说明了计算机系由计算机与信息系统两个专业组成，它们分别有600和300名学生。

```
def university = ['Computing' : ['Computing' : 600, 'Information Systems' : 300],
                  'Engineering' : ['Civil' : 200, 'Mechanical' : 100],
                  'Management' : ['Management' : 800]
]
```

请回答下列问题：

- (a) 该校共有多少个系？
(b) 计算机系共有多少个专业？
(c) 土木工程（Civil）专业共有多少个学生？

第5章 基本输入输出

严格地说，输入输出工具并不是Groovy语言的一部分，然而却与程序的运行环境有很大关系。在Groovy中，为了实现基本的文本输出功能，可以使用下面形式的语句：

```
print xxx          print(xxx)  
println xxx       println(xxx)
```

print和println方法通常用来输出相关的值（用xxx表示），该值可能表示一个字符串字面值、变量或者表达式，也有可能是一个解释型字符串。print方法用来输出它的值，并且其后续输出也在同一行上显示。println方法在显示值之后，将在下一行换行输出。

5.1 基本输出

范例01演示了print和println方法输出字符串字面值的过程。

范例01 简单输出

```
print "My name is"  
print("Ken")  
println()  
  
println "My first program"  
println("This is fun")
```

执行上面的Groovy脚本，其输出结果为：

```
My name is Ken  
My first program  
This is fun
```

前两个程序语句的执行结果输出在第一行中。print方法所带的参数将在控制台上显示，其余的输出将在同一行相继输出。简单的调用println()方法即可输出一个换行符。请注意，圆括号内的参数可以是任意类型的值。

根据第3章所讨论的内容，我们知道包含在双引号内的String对象将被解释，类似于\${expression}的所有语句都将会被求值，其值将作为字符串的一部分。范例02演示了它在print语句中的用法。

范例02 输出一个值

```
def age = 25  
print "My age is: "  
println age  
println "My age is: ${age}"
```

执行此脚本将输出：

```
My age is: 25  
My age is: 25
```

请注意，`println age`语句输出变量age的方法。

同样，我们也可以输出列表或者映射的内容。范例03演示了输出列表和映射值的方法。

范例03 输出列表和映射值

```
def numbers =[11,12,13,14 ]  
def staffTel =[ 'Ken' ::2745,'John' ::2746,'Jessie' ::2772 ]  
  
println "Numbers:${numbers}"  
println "Staff telephones:${staffTel}"
```

其输出结果为：

```
Numbers: [11, 12, 13, 14]  
Staff telephones: ["Jessie":2772, "John":2746, "Ken":2745]
```

5.2 格式化输出

格式化输出通过调用`printf`方法实现，`printf`方法的调用形式为：

```
printf(String format, List values)
```

调用此方法将在控制台上显示值。`values`参数表示那些即将输出的任意表达式，这些值的显示样式是由格式化字符串控制的。格式化字符串包含两类字符：简单拷贝至输出的普通字符，用来控制格式转换与输出的格式字符。

范例04演示了`printf`的两个简单用法。在这两个范例中，将要输出的列表值为空，格式化字符串由简单字符组成。调用第一个`printf`方法将显示格式化字符串，下一条输出紧跟在同一行上输出。第二个范例包含一个表示新行的换行符`\n`，在其后的所有输出都将换行输出。

范例04 简单的格式化输出

```
printf('My name is Ken', [])  
printf('My name is Ken\n', [])
```

下面的范例中含有一个列表值，其格式化字符串包含每个值的格式字符。格式说明由百分号（%）开头，在第一个范例中，`%d`表示按整型数据输出；在第二个范例中，`%f`格式字符表示按浮点数输出。

范例05 转换规范

```
def a =10  
def b =15  
printf('The sum of %d and %d is %d \n',[a,b,a +b ])  
  
def x =1.234  
def y =56.78  
printf('%f from %f gives %f \n',[y,x,x -y ])
```

其输出结果为：

```
The sum of 10 and 15 is 25
56.780000 from 1.234000 gives - 55.546000
```

欲获得转换规范方面更深入的论述，请参见附录F。范例06演示了使用%s格式字符输出字符串的范例。在这三个范例中，其输出字符串都包含在“[”和“]”中，以比较其不同的转换效果。%s仅仅按原样输出字符串，格式字符%20s在右侧输出字符串，并且宽度为20。格式字符%-20s则在左侧输出字符串。

范例06 字符串输出宽度和格式

```
printf('[%s]\n', ["Hello there"])
printf('[%20s]\n', ["Hello there"])
printf(['%-20s']\n', ["Hello there"])
```

其输出结果为：

```
[Hello there]
[      Hello there]
[Hello there      ]
```

5.3 基本输入

在Java中，System类定义的对象in即是标准输入，它是InputStream类的一个对象。该类含有将单行输入读作一个字符串的readLine方法。因此，调用System.in.readLine()方法通常用来获取用户的单行输入。范例07演示了在某个程序中读取用户姓名的方法。

范例07 简单输入

```
print "Please enter your name: "
def name = System.in.readLine()
println "My name is: ${name}"
```

readLine方法返回一个字符串值，我们可以使用toInteger方法将其转换成一个整型值，或者使用toDouble将其转换成一个浮点数值。范例08演示了可能用来读取不同输入值的三种Groovy方法（参见第7章）。

范例08 多种类型数据的输入

```
def readString(){
    return System.in.readLine()
}

def readInteger(){
    return System.in.readLine().toInteger()
}

def readDouble(){
    return System.in.readLine().toDouble()
}
```

```
print 'Please enter your name:'
def name =readString()
println "My name is:${name}"

print 'Please enter your age:'
def age =readInteger()
println "My age is:${age}"
```

使用System.in.readLine()方法就意味着输入的每个值都将被作为单个文本行看待。toInteger方法和toDouble方法只能接受没有非法字符，并且前后没有空格的格式正确的字符串。

readString、readInteger等方法在其他的脚本语言中也经常出现，可以将其归入一组Console类的静态方法，以便引入到其他应用程序中。Console类可以在console包中找到。关于Console类的更多信息请参考附录F。

```
package console

class Console {
    def static readString(){...}
    def static readLine(){...}
    def static readInteger(){...}
    def static readDouble(){...}
    def static readBoolean(){...}
}
```

非常重要的是，Console类的方法能够标记用户输入，如，readInteger方法将会忽略整型值前的空格。范例09演示了这些方法的使用方法。

范例09 Console类的使用

```
import console.*

print 'Please enter your name:'
def name =Console.readString()
println "My name is:${name}"

print 'Please enter your age:'
def age =Console.readInteger()
println "My age is:${age}"
```

执行上述脚本将得到如下结果（用户输入使用粗斜体表示）：

```
Please enter your name: Ken
My name is: Ken
Please enter your age: 25
My age is: 25
```

Console类能够缓冲和标记输入，因此可能会有多个值在同一行输入。范例10演示了它的具体用法。

范例10 缓冲和标记输入

```
import console.*  
  
print 'Please enter your name and age:'  
def name = Console.readString()  
def age = Console.readInteger()  
println "Name: ${name}, and age: ${age}"
```

执行此脚本将得到如下结果：

```
Please enter your name and age: Ken 25  
Name: Ken, and age: 25
```

5.4 习题

1. 请编写Groovy程序以输出如下两行内容：

```
Programming in Groovy  
is fab
```

请使用(a)两条输出语句；(b)一条输出语句。

2. 在Groovy脚本中，声明变量staffNumber，并将其初始化为123，声明变量staffSalary并将其初始化为456.78，请编写输出语句输出如下结果（请注意数值和文本同时输出的方法）：

```
STAFF      PAY  
123        456.78
```

3. 请编写Groovy脚本读取货币总量值123.45，并将其值增加10%，然后输出新的货币总量。
4. 请编写程序计算从午夜开始到现在所经历时间的秒数，并将其以小时、分钟和秒数表示，如以hh:mm:ss的形式表示。
5. 请编写程序读取物体的长度（用英寸表示），并将其以码数、英尺数（12英寸为1英尺，3英尺为1码）表示。
6. 请编写程序计算一个长度和宽度由用户输入的矩形的周长和面积。

第6章 学习案例：图书馆应用 程序（建模）

本章将演示Groovy列表和映射在实践中的用途。在第一个学习案例中，我们首先构建了一个简单的图书馆图书借阅模型。欲获得更为真实的范例，请参见本书那些重新回顾这一问题以及构建更精细的实现的章节。

为了与用户习惯保持一致，我们将使用迭代方式开发图书馆应用程序，这样的话，我们可以非常容易控制这个案例的实现变更。第一个迭代实现将演示列表和映射的使用方法。

6.1 迭代1：需求规范和列表实现

图书馆需要维护借阅者及其所借图书的记录。在这个非常简单的解决方案中，每本图书都通过书名来标记，借阅者都通过他或者她的姓名来标记。图书馆的图书借阅数据库可以通过多种方式实现，为了简单起见，我们选择Groovy自身支持的两个重要数据结构——列表和映射实现。

解决方案之一就是使用列表表示借阅数据库。在第4章中曾提到列表可以存储大量引用的对象，并且列表的每个元素值都可以通过整型值索引的方式访问。列表同样也能够动态地增长或者缩短，并且列表的每个元素也可以是任意类型的对象。这就意味着，我们可以通过列表的方式实现任意复杂的数据结构。在第一个解决方案中，用来表示图书馆数据库的列表的每个元素自身也是一个包含两个元素的列表，即借阅者姓名及其所借书籍的书名。

下面是借阅数据库的初始化方法：

```
def library =[ ['Ken','Groovy'],
              ['Ken','UML'],
              ['John','Java']
            ]
```

上面的代码给列表赋予了三个元素值。其中，每个元素都是一个含有两个元素的列表。第一个语句说明借阅者Ken借出了名为Groovy的书籍，第二个语句说明他同样也借出了名为UML的书籍，第三个语句用来记录John借出了名为Java的书籍。

根据第4章的内容，我们知道List类支持大量的、可以使列表操作更为简单的方法。举例来说，新增一条借阅记录可以通过调用add方法实现，也可使用与其功能一致的“<<”操作符来实现：

```
library <<['John','OOD']
library.add(['Sally','Basic'])
```

可以通过下列语句输出列表值：

```
println "Library:${library}"
```

这为检查列表是否包含所期望的元素值（参见其随后实际输出的文本）提供了一个可视化方法。

将上述所有的代码集中起来，可以得到下面的Groovy脚本：首先初始化该借阅数据库，然

后增加两条新记录，随后输出库中的内容。

范例01 一个列表实现程序

```
//initialize the loans database
def library =[ ['Ken','Groovy'],
              ['Ken','UML'],
              ['John','Java']
            ]

//add two new loans
library <<['John','OOD']
library.add(['Sally','Basic'])

//print the loan details
println "Library:${library}"
```

其输出结果为：

```
Library:[["Ken","Groovy"], ["Ken", "UML"], ["John", "Java"], ["John", "OOD"], ["Sally", "Basic"]]
```

虽然其输出结果的格式不太好，但作为第一个学习案例已经够用了。

为了更深入演示其案例的功能，我们可以通过下列语句判断Ken是否借阅了书名为UML的图书：

```
library.contains(['Ken', 'UML'])//true
```

通过下列语句可以获取所有已经借出的书籍的总数目：

```
library.size()
```

更新的脚本代码如下所示：

```
// as for the previous script
// ...

// determine if Ken has borrowed UML
println "Ken has borrowed UML? ${library.contains(['Ken', 'UML'])}"

// print the number of books on loan
Println "Number of books on loan: ${library.size()}"
```

其输出结果为：

```
Library:[["Ken","Groovy"], ["Ken", "UML"], ["John", "Java"], ["John", "OOD"], ["Sally", "Basic"]]
Ken has borrowed UML?true
Number of books on loan:5
```

请注意，即使在如此简单的范例中，我们也可以在List类中使用GDK方法，如leftShift(“<<”操作符)，就像使用JDK方法（如contains）一样高效。这么做不仅会大大简化了我们的工作，而且还能得到一个非常健壮的Groovy程序。

6.2 迭代2：映射实现

另一个替代方案就是使用映射方式实现借阅数据库。我们回想一下，第4章中曾提到映射

是一个无序的对象引用的集合，它也能动态地增长和缩短，映射中的每个值可以通过关键字进行访问。在向映射插入值时，必须同时提供一组通常称为映射项的关键字或值。由于关键字和值都可以是任意类型的对象，因此和列表实现方式一样，我们也能够通过映射构造出任意复杂的数据结构。

举例来说，我们可以使用列表把某个被借出的书名与其借阅者关联起来，其关键字就是借阅者姓名，与之相对应的值则为由该借阅者所借书名组成的列表。可以使用如下的方法初始化该借阅数据库：

```
def library = ['Ken' : ['Groovy', 'UML'],
              'John': ['Java']]
]
```

上述代码的作用是给映射加入两个条目。第一个语句将Ken作为关键字，与之相对应的值则是由Groovy和UML两个元素组成的列表。它们都是Ken所借阅图书的书名。同样，第二个语句用来记录John借阅了一本名为Java的书籍。

可以通过下列语句，向映射增加一个新的借阅者及她所借图书的条目：

```
library['Sally'] = ['Basic']
```

由于映射不允许使用相同的关键字，因此在更新一条已经存在的借阅者的书名列表时必须格外小心。任何尝试着向映射加入重复关键字的做法都必将导致原来的值丢失。因此，我们必须使用下面的方法向John增加一条新的借阅记录：

```
library['John'] = library['John'] << 'OOD'
```

这个语句将生成一个新的列表以替代映射中原来存在的条目。

和列表方法一样，可以通过下面的语句判断Ken是否借阅了指定的书籍：

```
library['Ken'].contains('UML')
```

使用如下方法可以统计借阅者个数：

```
library.size()
```

和列表方法一样，我们可以同时使用GDK和JDK中已经存在的Map类的方法。举个更为深入的例子，如果想以词典顺序对借阅者姓名进行排序，并且想知道Ken所借出的书籍数目，通过下列语句可以非常容易实现这个功能：

```
library.keySet().sort()
```

和

```
library['Ken'].size()
```

将上述代码集中起来，可以得到下面的Groovy脚本：

范例02 映射实现

```
//initialize the loans database
def library =['Ken' :['Groovy', 'UML'],
             'John' :['Java']]
]

//add a new borrower
```

```
library ['Sally']=['Basic']
//update an existing borrower
library ['John']=library ['John']<<'OOD'

//display the data in various ways
println "Library:${library}"

println "Ken has borrowed UML? ${library ['Ken'].contains('UML')}"

println "Number of borrowers in the library: ${library.size()}"  
println "Library:${library.keySet().sort()}"

println "Number of books borrowed by Ken: ${library ['Ken'].size()}"
```

该脚本的输入结果为：

```
Library: ["Sally": ["Basic"], "John": ["Java", "OOD"], "Ken": ["Groovy", "UML"]]
Ken has borrowed UML? true
Number of borrowers in the library: 3
Library: ["John", "Ken", "Sally"]
Number of books borrowed by Ken: 2
```

如前所述，虽然输出结果同样也没有进行格式化处理，但对于实现我们的目标来说已经足够了。最重要的一点是：借助Groovy自身所带的列表和映射语法，可以大大简化编程工作量。本书将在下面的章节中更深入讨论其特性。

6.3 习题

1. 请使用迭代1中所述的列表方法，编写获取某借阅者所借阅书籍数目的代码。
2. 请考虑如下情形：同一个借阅者可能是两家不同图书馆的借阅者，请使用列表方法获取哪些借阅者同时是两家不同图书馆的借阅者。
3. 使用迭代2中所述的映射方法重新实现该案例，并将书名作为关键字，借阅者姓名列表作为其值（在这里假定图书馆中有数本相同书籍）。
4. 比较参与两个独立项目的员工名字列表。一个员工可能同时参与一个或者两个项目。请编写代码，首先获取同时参与两个项目的员工姓名，然后获取那些仅参与一个项目的员工姓名。
5. 构造一个映射，其关键字是每个人的姓名，其值是每个人的年龄。请问，如何编写代码以获取John的年龄？如何编写代码以实现从映射中删除一个条目？如何如何编写代码以获取年龄最小的那个人的年龄？

第7章 方 法

方法就是用来表示一段程序代码的名字，它在程序中可以被执行或者被调用任意多次。调用方法时可能还需要提供输入参数；一个方法在多次被调用时可能使用不同的参数，这些不同的参数会决定方法被执行的实际效果。

在Groovy中，使用方法有利于把大而复杂的程序分割为小而易于处理的程序单元，因而使用方法可以简化编程工作。应用程序中的每个方法都能够实现某个特定的功能，一个方法可以执行或者调用其他方法，因此一个方法所表示的任务通常可以分割为多个子任务，由其他子方法分别实现。一个程序的方法也可以在其他应用程序中使用，因而可以避免重写该代码。

从概念上讲，本章所述的Groovy方法和其他程序语言的功能、过程以及子程序是等价的。

7.1 方法

方法通常需要用关键字def声明，最简单的不含参数的方法的声明形式如下所示：

```
def methodName() {  
    // 方法代码从这里开始  
}
```

方法名通常作为程序标识符（参见第2章）。如果一个方法不带参数，需要使用()，并且括号不能省略。

范例01 一个简单的方法定义

```
def greetings() {  
    println 'Hello and welcome'  
}  
greetings()
```

在这里，该方法被命名为greetings。该方法的代码包括向用户输出一条简单问候语功能，随后可以使用greetings()的方式调用该方法。程序输出结果为：

```
Hello and welcome
```

该方法同样也可以使用范例02所示的代码：

范例02 使用三个语句的方法

```
def greetings() {  
    print 'Hello'  
    print ' and '  
    println 'welcome'  
}  
greetings()
```

在这里，greetings方法含有三条语句。每一个语句都分别使用了输出语句，在这里每个语

句都在不同的代码行中。使用这种形式的代码大大增强了程序的可阅读性，因此本书的所有代码都使用这种方式。如果需要在同一行中使用两个或多个语句，则必须使用分号隔开，如：

范例03 在一行中使用多个语句

```
def greetings() {
    print 'Hello'; print' and '
    println 'welcome'
}
greetings()
```

请思考含有多个变量的方法。假定某个程序需要读取两个整型值，并以倒序的方式输出其结果。为了实现该功能，方法必须用两个变量存储数据的值。

范例04 方法变量

```
import console.*

def reverse() {
    print 'Enter the two integer values:'
    def first = Console.readInteger()
    def second = Console.readInteger()
    println "Reversed values: ${second} and ${first}"
}
reverse() // 调用方法
```

运行该脚本，其输出结果如下所示（用户输入使用粗斜体表示）：

```
Enter the two integer values: 12
34
Reversed values: 34 and 12
```

下一个范例和最后一个范例较为类似。它首先也读取某些数据，然后处理并输出其运算结果，其处理过程包含一些数学运算。该程序读入三个整型值分别用来表示24小时制的小时、分钟和秒数，然后将该时间以秒为单位表示。

范例05 转换时钟时间

```
import console.*

def processTime(){
    print 'Enter the time to be converted:'
    def hours =Console.readInteger()
    def minutes =Console.readInteger()
    def seconds =Console.readInteger()
    def totalSeconds =(60 *hours +minutes)*60 +seconds
    println "The original time of:${hours}hours,${minutes}minutes
    and ${seconds}seconds "
    println "Converts to:${totalSeconds}seconds "
}

processTime() //now call it
```

运行该程序，其输出结果如下所示：

```
Enter the time to be converted: 1
2
3
The original time of: 1 hours, 2 minutes and 3 seconds
Converts to: 3723 seconds
```

7.2 方法参数

含有一个或者多个参数的方法的用途更为广泛。我们可以通过使用方法参数的方式将参数传递给所调用的方法。一个含有三个参数的方法形如

```
def methodName(para1, para2, para3) {
    // 方法代码从这里开始
}
```

方法的参数以形参列表的方式紧跟在方法名后面的括号中，各参数名必须互不相同。

举例来说，让我们回过头来看greetings方法。第一个版本只是简单的输出一行固定的消息。如果能够提供一个表示被欢迎人的姓名作为参数，将能使该方法实现个性化输出。下面是其新版本代码。

范例06 方法参数

```
def greetings(name){
    println "Hello and welcome, ${name}"
}

greetings('John')
```

运行该程序，其输出结果如下所示：

```
Hello and welcome, John
```

实参'John'初始化了形参name，于是输出上述结果。

7.3 默认参数

方法中的形参可以指定为默认参数。一旦给定默认值，在调用该值的时候就不需要显式地传递其值。默认参数可以通过赋值语句来指定。当某个方法的声明含有默认参数时，默认参数仅能出现在非默认参数之后。也就是说，默认参数只能出现在形参列表的最后面。默认参数和非默认参数不能混杂，举例来说，在下面的方法中

```
def someMethod(para1, para2 = 0, para3 = 0) {
    // Method code goes here
}
```

第二个和第三个参数都已经被赋予了默认值。

someMethod方法可以通过一个、两个或者三个实参调用。如果仅仅提供一个实参，则其余两个参数均为0。如果使用了两个实参，则最后一个参数为0。调用该方法最少必须提供一个实

参，最多提供三个实参。范例07演示了默认参数的使用方法。

范例07 默认参数

```
def greetings(salutation, name = 'Ken') {
    println "${salutation} ${name}"
}

greetings('Hello', 'John')           // Hello John
greetings('Welcome')                // Welcome Ken
```

执行该脚本时，可以发现在第二次调用greetings方法时，name参数已经拥有默认值'Ken'：

```
Hello John
Welcome Ken
```

7.4 方法返回值

方法同样也能为其调用者返回一个值，通过return语句的形式可以实现此功能：

```
return expression
```

此语句表示调用该方法后立即返回表达式的值，并且其返回值能被调用者直接使用。该值也可以通过适当的赋值语句获取。

范例08演示了return语句的使用方法。hmsToSeconds方法能够通过其参数获得时间值，然后将其转换为秒数。在这里，方法返回其计算的结果给调用者。调用该方法以及输出返回值的代码如下所示：

范例08 方法返回值

```
import console.*

def hmsToSeconds(h,m,s){
    return (60 *h+m)*60+s
}

//Get the input from the user.
print 'Enter hours to convert:'
def hours =Console.readInteger()
print 'Enter minutes to convert:'
def minutes =Console.readInteger()
print 'Enter seconds to convert:'
def seconds =Console.readInteger()

//Now call the method.
def total =hmsToSeconds(hours,minutes,seconds)
println "Total number of seconds =${total}"
```

在该程序上运行某个会话，其结果为：

```
Enter hours to convert: 1
Enter minutes to convert: 2
Enter seconds to convert: 3
Total number of seconds = 3723
```

最后，我们必须注意return关键字是可选的。如果省略return关键字，则代码的最后一条语句的值就是方法返回值。范例09修正了hmsToSeconds方法，并重新实现了前面的范例。

范例09 隐式的返回值

```
import console.*

def hmsToSeconds(h,m,s){
    def totalSeconds =(60 *h+m)*60+s
    totalSeconds
}

//Get the input from the user.
print 'Enter hours to convert:'
def hours =Console.readInteger()
print 'Enter minutes to convert:'
def minutes =Console.readInteger()
print 'Enter seconds to convert:'
def seconds =Console.readInteger()
//Now call the method.
def total =hmsToSeconds(hours,minutes,seconds)
println "Total number of seconds =${total}"
```

7.5 参数传递

在Groovy中，方法的参数传递策略为传值方式。这意味着实参的值通常用来初始化形参的值。举例来说，在前一个范例中，程序在调用hmsToSeconds方法时，使用实参hours的值初始化形参h。其余两个形参的传值方式也使用类似的方法。

在第2章（第2.6节）中，我们讨论了变量引用对象的方法。变量是指一个对象所占用的内存空间，图2-1至2-4演示了这些概念。这就是说，使用实参初始化某个方法的形参时，实际上是通过其别名实现的，图2-2演示了该效果。因此，范例09在调用hmsToSeconds方法时，形参h实际上是实参hours的一个别名。

这种方法就意味着，给方法内部的形参赋值时必须为该形参创建一个新的对象以供其引用，因此，与其相应的实参实际上不受此影响，范例10演示了这种效果。

范例10 参数别名

```
def printName(name){
    println "Name (at entry):${name}"
    name ='John'
    println "Name (after assignment):${name}"
}

def tutor ='Ken'
printName(tutor)

println "Tutor:${tutor}"
```

运行该程序，其结果为：

```
Name (at entry): Ken  
Name (after assignment): John  
Tutor: Ken
```

printName方法定义了一个形参name，该方法首先输出传递给它的值，这就会给该形参变量创建一个新的字符串对象。其结果如图2-2所示，name参数于是引用值为'John'的新字符串对象。方法中的最后一条print语句验证了它实际上并没有拥有该值。在此代码中，变量tutor以实参形式引用对象('Ken')调用printName方法。由于在方法的入口处，它同样也被name参数所引用，因此第一个输出语句的输出结果为'Ken'。当从printName方法返回后，程序在输出tutor值后终止。可以看到形参值的改变实际上对该方法并没有什么影响。

swap方法是一种以别名形式交换形参值和实参值的方法，如范例11所示，它实际上并没有达到所期望的结果。所声明的swap方法本来期望交换参数x和参数y的值。这种结果在执行该方法时能够做到，但像前面所解释那样，这种交换并不会反应到相应的实参上。请执行该程序并仔细观察其结果。

范例11 交换值的方法

```
Enter the first value:12  
Enter the second value:34  
First:12  
Second:34  
  
import console.*  
def swap(x,y){  
    def temp =x  
    x=y  
    y =temp  
}  
  
print 'Enter the first value:'  
def first =Console.readInteger()  
print 'Enter the second value:'  
def second =Console.readInteger()  
  
//Now call the swap method  
swap(first,second)  
println "First:${first}"  
println "Second:${second}"
```

7.6 作用域

范例05中的processTime方法含有hours、minutes、seconds以及totalSeconds四个变量。由于它们都在该方法内部定义，因而被称作局部变量。局部变量只能在方法体中有效，方法体是这些变量的作用域。这就意味着它们只能在其作用域内引用，在其作用域之外无效，因此，无法在方法之外的代码中使用它们。

前面曾提到方法的参数可以紧跟在方法名后，在括号中以形参列表的形式出现。这些参数名必须互不相同，并且这些形参名都是该方法的局部变量。当方法被调用时，这些形参都将使用与其相应的实参初始化，形参同样也作为方法的一个局部变量，仅在方法内部有效。

在方法外部声明变量也使用相同的机制，如范例11中所示的变量first和变量second。附录B详细论述了run方法将Groovy脚本编译成Java类的过程。使用关键字def且在方法之外声明的变量对run方法来说是有效的局部变量；但是，所有别的Groovy方法却不能引用它（参见范例12）。

范例12包含printName方法和声明的变量tutor。从前面部分可知，变量tutor是run方法的局部变量，不能通过方法printName引用（参见该方法中的注释行部分）。

范例12 变量的作用域

```
def printName(name){  
    println "Name (at entry): ${name}"  
    //name =tutor  
    name ='Ken'  
    println "Name (after assignment): ${name}"  
}  
  
def tutor ='Ken'  
  
printName('John')  
  
//println "Name:${name}"      //ERROR:No such property
```

运行该程序，其输出为：

```
Name (at entry): John  
Name (after assignment): Ken
```

这两行输出显示了形参name最初被实参值'John'初始化，然后将它赋值为字符串'Ken'。

注意最后一行代码的注释，参数name就像printName方法中所定义的所有变量一样，都将在方法内部作为其作用域。因此，这些变量都只能在该方法内部被引用。任何尝试在其他地方引用name变量的代码都将会产生如上所示的错误。

范例13再次讨论了作用域的范围。当指向的变量与方法不相关时，同样也不能在方法内部引用。

范例13 变量和方法具有相同的作用域

```
def tutor ='Ken'  
  
def printName(name){  
    println "Name:${name}"  
    //println "Tutor:${tutor}"  
}  
  
printName('John')
```

该程序的输出如所期望的那样，printName方法并不能访问变量tutor（参见方法中的注释行）。

```
Name: John
```

7.7 集合作为参数和返回值

Groovy中的方法也支持集合参数，如列表，并返回一个集合值。在范例14中，sort方法将为一个列表值排序。如果第二个参数是布尔值true，该列表将以升序排列；如果该布尔值是false，则该列表以降序形式排列。

范例14 List参数及其返回值

```
def sort(list,ascending =true){  
    list.sort()  
    if(ascending ==false)  
        list =list.reverse()  
    return list  
}  
  
def numbers =[10,5,3,6 ]  
  
assert(sort(numbers,false)==[10,6,5,3 ])
```

在这里，上述程序不显示其结果，使用assert关键字给sort方法的返回值定义一个断言。由于该方法实际输出列表为[10, 6, 5, 3]，因此该断言为真，并且程序没有输出值。如果该断言为假，则会抛出AssertionError消息。第15章将对断言进行更深入的论述。

关于方法的更多知识，如递归方法和静态类型方法的参数及其返回值，将在附录G中详细论述。

7.8 习题

在完成下列习题之前，请读者参考附录G中的帮助信息。

1. 请编写并测试一个名为square的方法，其参数为给定的正方形边长值，其返回值为正方形的面积。
2. 英国在使用十进位货币制度之前使用便士、先令和英镑等货币单位，其中，1先令等于12便士，20先令等于1英镑。请编写方法分别求两个货币值之和及两个货币值之差。这两种方法都需要使用六个参数，其中前三个参数表示第一个货币总值，余下的三个表示另一个货币总值。每个方法都以便士数目作为其返回值。
3. 请编写并测试一个方法，以确认在指定一天中某个时间是否在另一个时间之前。每个时间参数以三个一组的形式给定，如11, 59, AM或者1, 15, PM。
4. 值 1, 2, 4, 8, 16, …是2的幂值。首先请编写一个方法isEven，以判断某个整型参数是否是偶数值。然后使用isEven方法编写递归方法isPowerOfTwo以确认某个参数是否是2的幂数。
5. 请只使用head和tail方法（参加附录G中的范例02），编写length方法以计算指定列表参数的元素个数。
6. 请只使用head、tail和cons方法，编写reverse方法以获取指定列表参数的元素的倒序列表。
7. 请使用递归方法实现习题5中length方法。
8. 编写递归方法maxList以获取一个整型列表中的最大值。
9. 假定列表[50, 20, 10, 5, 2, 1] 和列表[25, 10, 5, 1] 分别表示英国和美国的流通硬币值。请编写changeUK和changeUS方法，获取一个与给定货币值（其值为1至99（含）之间）数量相等的硬币列表。
10. 编写一个名为intToRoman的方法（与前个习题类似），将一个整数以罗马数字形式表示；该整型值以

参数形式给定，罗马数字值以一个字符串列表形式返回。简单来说，如，值1984可以用MDCCCLXXXIII表示。

11. 有向图列表的每个元素都由一对列表值表示。列表graph = [['a', 'b'], ['a', 'c'], ['a', 'd'], ['b', 'e'], ['c', 'f'], ['d', 'e'], ['e', 'f'], ['e', 'g']]表示：在一个有向图中，节点a指向b、c和d，节点b指向e等。请编写方法successors(node, graph)以返回graph中node节点的后续节点列表。
12. 许多图论的算法都是通过边实现的，保证一个节点至多只能被遍历一次。在深度优先搜索算法中，在其他节点被遍历之前，当前节点所能到达的子图中节点会首先被全部遍历。请编写depthFirst(node, graph)方法，以返回从graph中node节点开始，执行深度优先算法所遍历的节点列表。
13. 编写一个名为explode的方法，将一个字符串转换成一个长度为1的字符串列表。该方法形如：

```
def explode(str) { ... }
```

请完成implode方法，其接受一系列字符串值并将其联结成一个字符串：

```
def implode(strList) { ... }
```

使用上述两个方法，编写reverseString方法将字符串参数中字符以倒序方式输出：

```
def reverseString(str) { ... }
```

最后，请编写isPalindrome方法，在字符串参数是回文（palindromic）时返回布尔值true，回文是指顺读和倒读都一样的文本。

```
def isPalindrome(str) { ... }
```

14. 使用习题13中的explode和implode方法，请编写remove方法以删除字符串中出现的所有指定字符：

```
def remove(ch, str) { ... }
```

15. 两个整型数的最大公约数可以通过欧几里得算法求出，用递归的方法描述如下所示：

```
gcd(n, m) = n           if n == m
gcd(n, m) = gcd(n, m - n) if n < m
gcd(n, m) = gcd(n - m, m) otherwise
```

请编写一个方法实现该功能，并验证gcd(18,27)的值为9。

16. 阿克曼算法以递归的形式可以定义为：

```
ackermann(n, m)=1+m           if n= =0
ackermann(n, m) = ackermann(n - 1, 1)   if m = = 0
ackermann(n, m) = ackermann(n - 1, ackermann(n, m - 1)) otherwise
```

请编写一个方法实现该功能，并验证ackermann(3,3)的值为61。

17. 请实现Quicksort算法进行列表值排序，并使用google搜索功能看看其实现结果。

第8章 流程控制

程序语句的执行必然会导致某个操作的执行。前面所编写的程序都是以顺序执行的方式执行完某条语句后，再执行下一个语句。由于这种语句的执行顺序关系，我们可以将此程序在逻辑上看作是顺序的。同样也可以在方法的定义中创建一个抽象行为，并将它们也看作是一些通过这些方法所调用的简单语句。

另外，Groovy提供了一些用来改变程序逻辑的流程控制语句，可以将它们分成三种流程控制结构：

- 顺序
- 选择
- 迭代

8.1 while语句

最基本的迭代子句是while语句。while语句的语法如下所示：

```
while(condition) {  
    statement #1  
    statement #2  
    ...  
}
```

在执行while语句时，首先会计算condition表达式的值（一个布尔值）。如果该值为真，则执行while语句中内嵌的语句。整个过程都重复进行，接下来重新计算条件语句的值来决定是否开始下一个循环。循环将一直持续，直到该条件表达式的值为false时循环终止。程序将继续执行紧跟在while语句之后的语句。几条程序语句的组合通常被称为复合语句或者代码块。

在while循环语句中，如果仅有一条内嵌语句，则单个语句可以用下面的形式表示：

```
while(condition)  
    statement
```

范例01输出从1到10之间的值。每个循环过程都通过循环语句输出变量count的当前值，然后将其值加一。count变量初始值为1，while语句中的条件语句用来指定变量count的值在不超过LIMIT值的情况下，循环将继续执行。

范例01 while语句

```
//Set limit and counter  
def LIMIT =10  
def count =1  
  
println 'Start'
```

```
while(count <=LIMIT){  
    println "count:$!count!"  
    count++  
}  
  
println'Done'
```

程序输出值为：

```
Start  
count: 1  
count: 2  
count: 3  
count: 4  
count: 5  
count: 6  
count: 7  
count: 8  
count: 9  
count: 10  
Done
```

按照惯例，我们在表示拥有固定值的变量时，变量名的首字母是大写的。拥有固定值的变量通常也称为常量。在声明常量时，通常将一个给定的值赋给其变量名。变量的声明在代码中仅出现一次，并且仅使用一个语句更改其值。

while语句的一个典型应用是，按照不确定的次数，循环处理一段连续的程序代码行。在循环语句中，通常使用条件语句来控制循环次数。范例02演示了一个用来读取未知个数的正整数，并且计算这些值之和的程序，当用户输入任意负数值时循环终止。

范例02 对输入的一系列正整数求和

```
import console.*  
  
//Running total  
def sum =0  
  
print 'Enter first value:'  
def data =Console.readInteger()  
while(data >=0){  
    sum +=data  
    print 'Enter next value:'  
    data =Console.readInteger()  
}  
  
println "The sum is:$!sum"
```

该程序的一个简单会话如下所示：

```
Enter first value:1  
Enter next value:2
```

```
Enter next value:3
Enter next value:4
Enter next value:-1
The sum is:10
```

请注意在此范例中，如果第一个输入数为负数，由于不符合循环条件，程序将立即终止，并且其和为0。正因为如此，while语句也可以当作控制循环体内语句循环处理0次或者多次的控制语句。

8.2 for语句

在groovy中，for语句通常用于循环处理某个范围、集合（列表、映射或者数组；参见第4章以及附录E）或者字符串。

```
for(variable in range) | for(variable in collection) | for(variable in string) |
    statement #1           statement #1           statement #1
    statement #2           statement #2           statement #2
    ...
    ...
}
```

范例03重新实现本章的第一个范例。当循环次数在逻辑上已知时，使用for语句控制循环通常是一种非常合适的做法。

范例03 for语句

```
def LIMIT = 10
println 'Start'
for(count in 1..LIMIT)
    println "count: ${count}"
println 'Done'
```

下一个范例说明了for语句在列表处理过程中的用法。

范例04 循环处理列表

```
// List
println 'Start'
for(count in [11, 12, 13, 14])
    println "count: ${count}"
println 'Done'
```

该程序的输出结果为：

```
Start
count: 11
count: 12
count: 13
count: 14
Done#
```

同样，也可以使用循环语句来遍历处理映射的每个元素。在范例05中，所有员工年龄之和记录在某个映射中。在这里要特别注意循环变量staffEntry。由于循环处理映射中包含所有条目，因而每个条目都是同时包括关键字及其值的一个Map.Entry对象（参见JDK帮助文档）。这样，在循环中就可以通过staffEntry.value获得每个员工的年龄。

范例05 循环处理映射

```
// Staff name and age
def staff = ['Ken' : 21, 'John' : 25, 'Sally' : 22]

def totalAge = 0
for(staffEntry in staff)
    totalAge += staffEntry.value

println "Total staff age: ${totalAge}"
```

该程序的输出结果为：

```
Total staff age: 68
```

最后，我们将演示遍历组成某个字符串的所有字符的方法。在范例06中，name将会分割成一个接一个的字符对象，并插入到一个列表中。

范例06 循环处理字符串

```
def name = 'Kenneth'
def listOfCharacters = []
.

for(letter in name)
    listOfCharacters << letter

println "listOfCharacters: ${listOfCharacters}"
```

输出结果为：

```
listOfCharacters: ["K", "e", "n", "n", "e", "t", "h"]
```

8.3 if语句

if语句的形式通常为：

```
if(condition) {
    statement #1a
    statement #1b
    ...
} else {
    statement #2a
    statement #2b
    ...
}
```

在这里，if和else都是保留字。如果表达式condition的计算结果为布尔值true，则复合语句将从statement #1a开始执行，并且其控制权限将传递给if语句的下一个语句。如果该条件值为

假，复合语句将从statement #2a开始执行，并且其控制权限将传递给if语句的下一个语句。如前所述，两个复合语句也可以是单条语句。

在程序中，if语句提供了从两个直接逻辑分支路径中选择一条路径的方法。有些时候，我们希望选择是否执行某段代码，可以使用if语句的简化版本实现：

```
if(condition) {
    statement #1
    statement #2
    ...
}
```

如果上述条件表达式的计算结果为true，程序将执行内嵌的复合语句，并且会继续执行if语句后面的语句。如果该条件值为false，则会忽略其内嵌的复合语句，而直接执行if语句后面的语句。如前所述，两个复合语句也可以是单条语句。

在范例07中，程序将读入两个整型值，并将它们以升序方式输出。这可以通过一组if-else语句选择合适的print语句实现：

范例07 一个简单的if语句

```
import console.*

print 'Enter first value: '
def first = Console.readInteger()
print 'Enter second value: '
def second = Console.readInteger()

if(first < second)
    println "${first} and ${second}"
else
    println "${second} and ${first}"
```

运行该程序的某个交互会话，输出结果如下：

```
Enter first value: 34
Enter second value: 12
12 and 34
```

范例08重新实现此范例。这次程序使用if语句的简化版本，如果条件语句判断出第一个值比第二个值大时，则其值相互交换。

范例08 两个值交换

```
import console.*

print 'Enter first value: '
def first = Console.readInteger()
print 'Enter second value: '
def second = Console.readInteger()

// Exchange the order
```

```

if(first > second) {
    def temp = first
    first = second
    second = temp
}

println "${first} and ${second}"

```

执行该程序将输出如下结果：

```

Enter first value: 34
Enter second value: 12
12 and 34

```

if语句允许多种结合方式。举例来说，与else语句相关联的可能是另一条if语句，这可能会嵌套许多次。这样的结构通常用于从多个逻辑分支路径选择一条路径。为了演示此效果，请思考一个读取考试成绩（包含0至100在内的任意值），并指定相应的字母等级的程序片断。其等级划分方法如下所示：

score	grade	score	grade
70-100	A	40-49	D
60-69	B	0-39	E
50-59	C		

实现此处理过程需要一连串的if-else语句：

```

if(score >= 70)
    grade = 'A'
else if(score >= 60)
    grade = 'B'
else if(score >= 50)
    grade = 'C'
else if(score >= 40)
    grade = 'D'
else
    grade = 'E'

```

8.4 switch语句

在上一节中，一连串频繁出现的if-else语句，可以通过一个已经存在的专用语句来实现此功能，它就是switch语句，其一般形式如下：

```

switch(expression) {
    case expression #1:
        statement #1a
        statement #1b
        ...
    case expression #2:
        statement #2a
}

```

```

statement #2b
...
...
case expression #N:
    statement #Na
    statement #Nb
...
default:
    statement #Da
    statement #Db
...
}

```

在上述语法中，switch、case和default均是Groovy关键字。default子句和嵌在其内的语句是可选的。圆括号中的控制条件表达式可以被求值。这些值将轮流与每个case表达式相比较。如果它与某个case表达式匹配，将执行从该case子句开始到switch结束的所有语句。如果没有匹配项，但有default子句，则执行default子语。范例09演示了switch语句的基本用法。

范例09 基本的switch行为

```

def n = 2
switch(n) {
    case 1: println 'One'
    case 2: println 'Two'
    case 3: println 'Three'
    case 4: println 'Four'
    default: println 'Default'
}
println 'End of switch'

```

在上述代码中，条件表达式仅仅是变量n的值。在求值后，它将依次与这些case表达式的值相比较。它在与case 2比较时匹配，程序的输出结果为：

```

Two
Three
Four
Default
End of switch

```

一般来说，case标签的语句必须互不相同。一旦所选择的值与case语句匹配，通常都希望执行与之相应的语句，然后执行紧跟在switch语句后面的程序。在switch语句上下文中，break语句能够立即终止switch语句的执行，转而执行switch之后的语句，因此可以实现此功能。范例10演示了此效果。

范例10 Switch和break语句

```

def n =2
switch(n){
    case 1:

```

```
    println 'One'
    break
case 2:
    println 'Two'
    break
case 3:
    println 'Three'
    break
case 4:
    println 'Four'
    break
default:
    println 'Default'
    break
}
println 'End of switch'
```

执行此程序，其输出结果为：

```
Two
End of switch
```

switch语句也能够替代前一节后面所述的一连串的if语句。范例11中的代码描述了基于考试成绩进行评价等级的switch语句。每个case子句都与代表考试成绩级别的分数范围匹配。在这里并没有用到default子句。

范例11 Switch和范围

```
import console.*

print 'Enter examination score:'
def score =Console.readInteger()
def grade

switch(score){
    case 70..100:
        grade ='A'
        break
    case 60..69:
        grade ='B'
        break
    case 50..59:
        grade ='C'
        break
    case 40..49:
        grade ='D'
        break
    case 0..39:
        grade ='E'
```

```

        break
    }

    println "Score:${score};grade:${grade}"
}

```

执行此程序，其输出结果为：

```

Enter examination score: 50
Score: 50; grade: C

```

正如程序所演示那样，case表达式既可以表示某个整型值，也可以表示整型值的范围。实际上，case表达式还可以是一个字符串、列表、正则表达式或者某些类的对象（参见第12章）。在范例12所演示的switch语句中，case表达式就是一个列表。如果控制表达式的值是某个集合中的一个成员值的话，那么我们就能获得与之相匹配的值。

范例12 List和case表达式

```

def number = 32

switch(number) {
    case [21, 22, 23, 24] :
        println 'number is a twenty something'
        break
    case [31, 32, 33, 34] :
        println 'number is a thirty something'
        break
    default :
        println 'number type is unknown'
        break
}

```

其输出为：

```
number is a thirty something
```

范例13演示了一个case表达式为正则表达式的switch语句，同样，其值将与所给定的模式相匹配。

范例13 表示case语句的正则表达式

```

def number ='1234'

switch(number){
    case ~'[0-9]{3}-[0-9]{4}' :
        println 'number is a telephone number'
        break
    case ~'[0-9]{4}' :
        println 'number is a 4-digit sequence'
        break
    default :
        println 'number type is unknown'
        break
}

```

其输出为：

```
number is a 4-digit sequence
```

8.5 break语句

break语句通常用来改变循环语句和switch语句中的控制流程。我们在前面已经看到，break语句在switch语句中的作用，break语句同样也可以用在while语句和for语句中。在循环结构体内执行break语句时，将立即终止循环体的最内层循环。

范例14演示了这种效果。此程序最多可求100个正整型值之和，用户输入的值作为此程序的输入。用户一旦输入某个负数，for循环将立即终止，程序输出用户提供的这些值之和。

范例14 for循环和break语句

```
import console.*  
  
def MAX =100  
def sum =0  
  
for(k in 1..MAX){  
    print 'Enter next value:'  
    def value =Console.readInteger()  
    if(value <0)  
        break  
    sum +=value  
}  
println "sum:${sum}"
```

运行此程序，输出结果为：

```
Enter next value: 11  
Enter next value: 12  
Enter next value: 13  
Enter next value: 14  
Enter next value: -1  
sum: 50
```

8.6 continue语句

continue语句是break语句的补充，仅限于在while和for循环中使用。当执行continue语句时，将结束本次循环，并跳转到离它最近的条件判断语句，以确认是否执行下一个循环。在含有continue语句的循环中，程序将跳过循环体内下面所有尚未执行的语句。

在范例15中，程序将计算用户输入的10个整型值之和。所有输入的负数值都无效。程序的功能也是统计所有输入的正整型值之和。

范例15 for循环和continue语句

```
import console.*  
  
def MAX = 10
```

```

def sum = 0

for(k in 1..MAX) {
    print 'Enter next value: '
    def value = Console.readInteger()
    if(value < 0)
        continue
    sum += value
}

println "sum: ${sum}"

```

程序的执行结果如下所示：

```

Enter next value: 1
Enter next value: 2
Enter next value: 3
Enter next value: 4
Enter next value: -5
Enter next value: -6
Enter next value: -7
Enter next value: 8
Enter next value: 9
Enter next value: 10
sum: 37

```

8.7 习题

1. 请只使用加法和减法运算，编写一个名为quotient的方法，以计算两个正整数之商。
2. 编写一个程序以读入某个正整型值，将它分隔成单独的数字，并用英文表示该数字。比如，当输入值是932时，输出为：

932: nine three two

3. 编写一个程序以接收一个以小时数、分钟数和秒数表示的时间值（24小时制），并用下面所示方式描述这些值：

09:10:00	ten past nine
10:45:00	quarter to eleven
11:15:00	quarter past eleven
17:30:00	half past five
19:50:00	ten to eight
06:12:29	just after ten past six
06:12:30	just before quarter past six
00:17:29	just after quarter past midnight

4. 斐波纳契数列遵循如下规则：数列中的前两个值都为1，其他的值都是前两个整型值之和。如果fib(b)表示序列中的第n个值，于是，

```

fib(1) = 1
fib(2) = 1
fib(n) = fib(n-1) + fib(n-2)

```

请编写程序以实现fib(n)，然后输出斐波纳契数列的前十个值。

5. 请使用简单循环的方式重新实现附录G中的阶乘算法。

6. 使用一个给定的映射表示公司员工及其电话号码，如：

```
def staff = ['Ken' : 2745, 'John' : 2746, 'Sally' : 2742]
```

请编写程序，输出一个以字母顺序排序的员工姓名及其所对应的电话号码的列表。

7. Zeller的一致性法则可以用来计算某个给定日期的星期值。该算法的计算结果为0~6之间任意合法的整型值，其中0代表星期天，1代表星期一，依次类推。其公式为：

$$Z = \left[\frac{26m - 2}{10} + k + D + \frac{D}{4} + \frac{C}{4} - 2C + 77 \right] \bmod 7$$

在上述公式中：D为在某个世纪的年份数，C为世纪数，k为在某个月的日期数，m为月数，其中需要将一月和二月看作是前一年的第11个月和第12个月，因此，三月的月数为1，四月的月数为2，……，十月的月数为10。

因此，日期16/11/2005的D值为5，C值为20，k值为16，m值为9。

请编写一个程序，以DD/MM/YYYY的形式读入日期值，并以如下形式表示该日期，如，日期16/11/2005的输出结果为：星期三，16November2005。

8. 扑克牌可以以简写的形式表示，如QS表示黑桃皇后，10S表示黑桃10。其排列依次为A、2…10、J、Q或者K。与之相应的花色为D、C、H或者S。请编写程序读入这些简写值，如QS，然后将其以普通形式表示，如黑桃皇后。

9. 在第7章的习题10中，我们实现一个用来将阿拉伯数字转换成罗马数字的方法intToRoman，请编写其逆过程方法romanToInt。

10. 请使用intToRoman和romanToInt方法编写addRoman和subtractRoman方法。每个方法都接收表示罗马数字值的两个字符串，并返回一个表示结果的字符串（同样也是罗马数字）。请编写一个应用程序，其输入形式如VIII + II所示。

11. 复活节(Easter Sunday)是3月21日或者其后的第一个星期天，可以通过如下公式计算（整数算法）：

```

c = year/100
n = year - 19 * (year /19)
k = (c - 17)/25
i = c - c /4 - (c - k)/3 +19 * n +15
i = i - 30 * (i /30)
i = i - (i /28)*(1 - i /28)*(29 /(i +1))*((21 - n)/11)
j = year +year /4 +i +2 - c +c /4
j = j - 7 * (j /7)
l = i - j
m = 3 + (l +40)/44

```

$$d = 1 + 28 - 31 * m / 4$$

如果d的值大于20，则复活节出现在三月，否则出现在四月。

12. 儒略日 (Julian date) 是指从某个远古纪元日期开始到现在所经过的天数。举例来说，如果今天的儒略值是100000，则明天的值将是100001。计算某个日期儒略值的算法如下所示：

```
def julian(day,month,year){
    def mm =month
    def yy =year

    if(mm >2)
        mm -=3
    else {
        mm +=9
        yy --
    }

    def cent =yy.intdiv(100)
    def yr =yy %100
    return ((146097 * cent).intdiv(4))+((1461 * yr).intdiv(4))
    +(153 * mm +2).intdiv(5)+day +1721119
}
```

请编写一个程序，以MM/YYYY的形式读入日期值，程序将输出该月的日历。举例来说，输入11/2005，其输出结果将为：

November 2005

S	M	T	W	T	F	S
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30			

13. 现在，Groovy语言并不支持Java语言中的do语句，请编写Groovy程序以实现do语句。

第9章 闭包

Groovy闭包是一种表示可执行代码块的方法。闭包也是对象，可以像方法一样传递参数。由于闭包是代码块，因此也可以在需要时执行。像方法一样，在定义的过程中，闭包也可以使用一个或者多个参数。闭包的一个重要的特有属性就是，它们可以访问属性信息。这就意味着在声明闭包之后，闭包可以使用并修改其作用域内的所有变量值。

闭包最常见的用途是处理集合。举例来说，可以遍历某个集合的所有元素，并将闭包应用到其这些元素上。Groovy的闭包是简化脚本开发的一个重要原因。

本章将介绍闭包的一般概念，附录H将详细介绍有关闭包的更多特性，附录J则说明了其更高级的用法。

9.1 闭包

闭包的语法如下所示：

```
{comma-separated-formal-parameter-list -> statement-list}
```

如果不需要形参，则可以省略参数List和->间隔符。下面是一个不带参数的闭包范例。

范例01 闭包及其调用方法

```
def clos = {println 'Hello world'}
clos.call()
```

在这里，闭包不带参数，并且仅由一个println语句组成。使用标识符clos引用这个闭包。如范例01所示，可以使用call语句来执行这个标识符所引用的代码块。程序的输出结果为：

```
Hello world
```

就像方法一样，在闭包声明中引入形参可使这些闭包更加实用。下面是一个相同的闭包，它接收每个人的名字作为祝贺信息的参数：

范例02 参数化的闭包

```
def clos = {param -> println "Hello ${param}"}

clos.call('world') // actual argument is 'world'
clos.call('again') // actual argument is 'again'
clos('shortcut') // abbreviated form
```

执行上面的脚本，输出结果为：

```
Hello world
Hello again
Hello shortcut
```

请注意第三个语句已经省略了call。

下面的演示范例重新实现前一个的范例，并且输出结果和前一个范例的输出结果相同，但是它演示了闭包使用隐参数it的方法。

范例03 单个隐参数

```
def clos = {println "Hello ${it}"}

clos.call('world')
clos.call('again')

clos('shortcut')
```

在绪论中已经提到，用户可以通过闭包访问属性值。闭包在定义后就能引用不同的对象。范例04中的变量greeting定义了欢迎语句。该变量在闭包clos定义之前定义，因此，它的值能在闭包调用时使用。变量的初始化值为'Hello'。

范例04 闭包和作用范围

```
def greeting = 'Hello'
def clos = {param -> println "${greeting} ${param}"}
clos.call('world')

// Now show that changes to this variable change the closure.
greeting = 'Welcome'
clos.call('world')
```

在第二次调用闭包之前，变量greeting的值已经改变，运行程序则会影响到输出结果：

```
Hello world
Welcome world
```

下一个范例将在前面的代码中增加一个名为demo的方法，它接收一个表示闭包的参数clo。demo方法通过参数'Ken'来调用闭包。demo方法同样也定义了一个新变量greeting，并将变量的值限定为'Bonjour'。调用新增加的方法，结果如下：

```
Welcome Ken
```

只有在闭包被定义且存在，而不是在被调用时，可以访问其状态值。下面是其演示代码：

范例05 作用范围

```
def greeting ='Hello'
def clos ={param ->println "${greeting}${param}"}
clos.call('world')

//Now show that changes to this variable change the closure.
greeting ='Welcome'
clos.call('world')

def demo(clo){
    def greeting ='Bonjour'           //does not affect closure
    clo.call('Ken')
}
```

```
demo(clos)
```

其输出结果为：

```
Hello world  
Welcome world  
Welcome Ken
```

如最后一个参数所示，Groovy为方法调用闭包提供了一个简化方案，这使得代码相对简单易懂。最后一个范例调用了方法demo，其实参为一个闭包。由于demo调用的最后一个参数是闭包，因此可以将它从实参列表中删除，并立即放到其后的括号中去。这样，demo方法的调用可以以下列两种方式出现：

```
demo(clos)      // 闭包参数在括号中  
demo() clos    // 闭包参数从括号中移出来
```

下一个范例将演示此用法。在调用方法demo时，闭包已经从实参中移出来，因此可以删除其空参数列表。

范例06 闭包在实参列表外部

```
def greeting ='Hello'  
def clos =|param ->println "${greeting}${param}"|  
  
def demo(clo)|  
    def greeting ='Bonjour'           //does not affect closure  
    clo.call('Ken')  
}  
  
//demo()clos                      //1:closure reference;include parentheses  
demo(){param ->println "Welcome ${param}"} //2:closure literal;include parentheses  
  
demo clos                         //3:closure reference;omit parentheses  
demo {param ->println "Welcome ${param}"} //4:closure literal;omit parentheses
```

其输出为：

```
Welcome Ken  
Hello Ken  
Welcome Ken
```

最后两行程序语句（注释行中标记为3和标记为4的代码行）调用了demo方法，并将闭包作为实参。在这两个方法中，第一个方法调用了闭包对象的引用，而第二个方法则调用了闭包字面值。这两种演示方法都省略了方法调用所使用的括号。

请仔细观察注释行中标记为1和标记为2的语句，第二条语句使用闭包字面值的方法在Groovy中是可行的。然而，由于第一条语句使用了一个闭包引用，因此不能作为代码的一部分。这将导致程序执行出错，并抛出传递给所调用的方法参数为空的错误信息。

闭包也常用在集合中（参见9.2节）。使用闭包可以更高效地遍历集合中的所有元素，并将闭包应用到每个元素中。举例来说，所有的数字类型值都支持upto方法，该方法的原型如下所示：

```
void upto(Number to, Closure closure)
```

使用如下代码可以调用该方法：

```
1.upto(10) { ... }
```

上面的代码将调用闭包10次。如果闭包有一个形参p：

```
1.upto(10) { p -> ... }
```

则每个迭代的参数值为1, 2, ..., 10。

upto方法在每次调用闭包时，其迭代都从它接收的数字值(1)开始，止于给定的参数值(10)。这种方法为计算值的阶乘提供了一个行之有效的途径。我们可以使用upto方法产生一连串的整型值1, 2, 3, ...一直到所给定的限定范围值为止。对于每个值来说，我们可以计算出部分的阶乘结果，到数列循环结束时即可计算出阶乘值。下面是其代码：

范例07 使用闭包求阶乘

```
def factorial = 1
1.upto(5) { num -> factorial *= num }
println "Factorial(5): ${factorial}"
```

执行该脚本，输出为：

```
Factorial(5): 120
```

9.2 闭包、集合和字符串

许多列表、映射和字符串方法都接收闭包参数（参加附录H）。这种闭包和集合相结合的方式，能够为常见的编程问题提供非常完美的Groovy解决方案。举例来说，each方法的原型为：

```
void each(Closure closure)
```

它常常用于列表、映射和字符串，以遍历每个元素，并将闭包应用于每个元素。范例08演示了each方法和闭包的一些简单应用范例。

范例08 一个使用each方法和闭包的范例

```
[1, 2, 3, 4].each {println it}
```

```
['Ken' : 21, 'John' : 22, 'Sally' : 25].each {println it}
['Ken' : 21, 'John' : 22, 'Sally' : 25].each {println "${it.key} maps to: ${it.value}"}
'Ken'.each {println it}
```

第一个范例在不同的行中输出值1、2、3和4。最后一个范例在不同的行中输出姓名的每个字母。在第二个演示范例中，映射中的关键字及其值以Ken=21的形式输出。第三个范例遍历映射中每个元素的关键字及其对应的值，然后以类似于Ken maps to: 21的形式输出映射中所有元素。其输出结果为：

```
1
2
3
4
Sally = 25
John = 22
```

```
Ken=21
Sally maps to: 25
John maps to: 22
Ken maps to: 21
K
e
n
```

通常，用户可能希望迭代处理集合的所有元素，并且在元素符合某个条件时，应用某些逻辑关系。在闭包中使用条件语句可以很容易地实现此功能。

范例09 条件元素

```
//even values only
[1,2,3,4 ].each {num ->if(num %2 ==0)println num}

//staff at least 25 years old
['Ken' :21,'John' :22,'Sally' :25 ].each {staff ->
    if(staff.value >=25)println staff.key
}
['Ken' :21,'John' :22,'Sally' :25 ].each {staffName,staffAge ->
    if(staffAge >=25)println staffName
}

//only lowercase letters
'Ken'.each {letter ->if(letter >='a' &&letter <='z')println letter}
```

脚本的输出结果为：

```
2
4
Sally
Sally
e
n
```

请仔细观察上面的两个范例，并找出那些年龄不小于25岁的元素。在这两个范例中，我们用迭代处理映射，且对每个映射元素都应用一个闭包。第一个范例的闭包参数staff是包含关键字及其对应值的Map.Entry。因此，可以在布尔表达式中使用staff.value以检查员工的年龄。在第二个范例中，闭包则有两个表示Map.Entry元素的参数，它们是关键字(staffName)及其对应值(staffAge)。

find方法返回集合中符合某个判断标准的第一个值。在闭包中，集合元素使用的判断条件必须是布尔表达式。当存在符合条件的值时，find方法将返回第一个符合条件的值，如果不存在符合条件的元素，则返回null。find方法原型如下所示：

```
Object find(Closure closure)
```

范例10 find方法和闭包功能演示

```
//locate the value 7
```

```

def value =[1,3,5,7,9 ].find {element ->element >6}
println "Found:${value}"

//locate no value (null)
value =[1,3,5,7,9 ].find {element ->element >10}
println "Found:${value}"

//first staff member over 21
value =['Ken' :21,'John' :22,'Sally' :25 ].find {staff ->staff.value >21}
println "Found:${value}"

```

运行此脚本，输出结果为：

```

Found: 7
Found: null
Found: Sally = 25

```

请注意，在映射上使用find方法时，其返回的对象为Map.Entry。在这种情况下，我们不能使用一组参数值表示关键字及其对应的值，如：

```
value=['Ken' : 21, 'John' : 22, 'Sally' : 25].find {key, value -> value > 21}
```

这将导致不确定所返回的值到底是关键字，还是与之相对应的值。

find方法在其需要匹配值的集合中定位第一个条目（如果有的话）；而findAll方法将遍历所有元素，并返回一个符合条件的列表。findAll方法的原型如下所示：

```
List findAll(Closure closure)
```

该方法将在使用它的对象中找出所有匹配闭包条件的值。范例11给出使用findAll方法的一些范例。第二个范例证实：闭包组合可以简化复杂算法的实现。这种方式的特点在于，对每个表达式来说使用闭包都相对简单。

范例11 findAll方法和闭包功能演示

```

// Find all items that exceed the value 6
def values = [1, 3, 5, 7, 9].findAll {element -> element > 6}
values.each {println it}

// Combine closures by piping the result of findAll
// through to each
[1, 3, 5, 7, 9].findAll {element -> element > 6}.each {println it}

// Apply a findAll to a Map finding all staff over the age of 24
values = ['Ken' : 21, 'John' : 22, 'Sally' : 25].findAll {staff -> staff.value > 24}
values.each {println it}

```

再次强调，在映射中使用findAll方法将生成一个Map.Entry元素列表。脚本的最后一行代码输出结果，演示了其效果：

```

7
9
7

```

```
9  
Sally = 25
```

另外两个使用闭包作为参数的方法是any和every。any方法将遍历检查集合的每个元素，以确认：对至少一个元素来说，某个布尔断言是否合法，该断言由闭包提供。every方法则用来检查：对集合的所有元素来说，某个断言（其返回值为true或者false的一个闭包）是否合法；如果合法则返回true，否则返回false。这两个方法的原型如下所示：

```
boolean any(Closure closure)  
boolean every(Closure closure)
```

范例12演示了一些有代表性的范例：

范例12 any和every方法

```
// Any number over 12?  
def anyElement = [11, 12, 13, 14].any {element -> element > 12}  
println "anyElement: ${anyElement}"  
  
// Are all values over 10?  
def allElements = [11, 12, 13, 14].every {element -> element > 10}  
println "allElements: ${allElements}"  
  
// Any staff member over the age of 30?  
def anyStaff = ['Ken' : 21, 'John' : 22, 'Sally' : 25].any {staff -> staff.value > 30}  
println "anyStaff: ${anyStaff}"
```

运行此脚本，其输出为：

```
anyElement: true  
allElements: true  
anyStaff: false
```

下面将要讨论的两个方法是collect和inject。这两个方法同样也使用闭包作为参数。方法collect将遍历某个集合，并使用闭包中的变换方法，将集合中的每个元素转换为一个新值。collect方法返回一个由转换后的值所组成的列表。其原型如下所示：

```
List collect(Closure closure)
```

范例13演示了该方法的最简单用法。

范例13 collect方法的简单用法

```
//Square of the values  
def list =[1,2,3,4 ].collect {element ->return element *element}  
println "list:${list}"  
  
//Square of the values (no explicit return)  
list =[1,2,3,4 ].collect {element ->element *element}  
println "list:${list}"  
  
//Double of the values (no explicit return)
```

```

list =(0..<5).collect {element ->2 *element}
println "list:${list}"

//Age by one year
def staff =['Ken' :21,'John' :22,'Sally' :25 ]
list =staff.collect {entry ->++entry.value}
def olderStaff =staff.collect {entry ->++entry.value;return entry}
println "staff:${staff}"
println "list:${list}"
println "olderStaff:${olderStaff}"

```

执行此脚本，其输出结果为：

```

list: [1, 4, 9, 16]
list: [1, 4, 9, 16]
list: [0, 2, 4, 6, 8]
staff: [Sally:27, John:24, Ken:23]
list: [26, 23, 22]
olderStaff: [Sally = 27, John = 24, Ken = 23]

```

有关collect方法的第三个范例是其在范围（Range）上的应用。因为范围（Range）的接口扩展了列表（List）接口，所以这种方法是可行的。也正是由于这个原因，它也常常用来替代列表。仔细观察上面的演示范例，每次迭代处理staff集合时，年龄值都将加1，其返回值为一个来自于映射中年龄值的新列表。映射容器对象和staff一样，也可以通过闭包改变其值。最后一个范例给oldStaff赋值，并构建一个Map.Entry列表，其对应的年龄值将再次加1。

范例14将更深入演示collect方法的作用。请注意map方法，它通过list参数调用使用闭包参数的collect方法。借助于map方法，可以计算每个元素的两倍值、三倍值，或者从一个整数列表中寻找偶数元素。我们在后面详细介绍map的算法（参见附录J）。

范例14 collect的高级范例

```

//A series of closures
def doubles ={item ->2 *item}
def triples ={item ->3 *item}
def isEven ={item ->(item %2 ==0)}

//A method to apply a closure to a list
def map(clos,list){
    return list.collect(clos)
}

//Uses:
println "Doubling:${map(doubles,[1,2,3,4 ])}"
println "Tripling:${map(triples,[1,2,3,4 ])}"
println "Evens:${map(isEven,[1,2,3,4 ])}"

```

该脚本的输出结果为：

```
Doubling: [2, 4, 6, 8]
```

```
Tripling: [3, 6, 9, 12]
Evens: [false, true, false, true]
```

在本节中，我们将要讲述的最后一个方法是inject。该方法可用于遍历集合，首先将需要传递的值和集合项目传给闭包，此时其传递的值将作为处理结果，然后再和下一个集合项目一起传给闭包，依此类推，其方法原型为：

```
Object inject(Object value, Closure closure)
```

下面是三个求5的阶乘的范例：

范例15 求5的阶乘值

```
//Direct usage
def factorial =[2,3,4,5 ].inject(1){previous,element ->previous *element}
println "Factorial(5): ${factorial}"

//Equivalence
def fact =1
[2,3,4,5 ].each {number ->fact *=number}
println "fact:${fact}"

//Named list
def list =[2,3,4,5 ]
factorial =list.inject(1){previous,element ->previous *element}
println "Factorial(5): ${factorial}"

//Named list and closure
list =[2,3,4,5 ]
def closure ={previous,element ->previous *element}
factorial =list.inject(1,closure)
println "Factorial(5): ${factorial}"
```

其输出结果为：

```
Factorial(5): 120
Fact: 120
Factorial(5): 120
Factorial(5): 120
```

在上述代码块中使用变量fact，主要用于演示inject方法是否能够达到使用each迭代方法的效果。首先，变量fact被分配给inject方法的第一个参数值（这里是1），然后遍历列表的每个元素。对第一个值（number = 2）来说，闭包计算fact*number的值，也就是fact=1*2=2。对第二个值（number = 3）来说，闭包再次计算fact*number的值，也就是fact=2*3=6，依此类推。

9.3 闭包的其他特性

由于闭包也是一个对象，因此它也可以作为方法的参数。在范例16中，非常简单的filter方法需要两个参数：它们分别是一个列表和一个闭包。该方法在列表中查找所有满足闭包指定条件的元素，当然，这需要通过findAll方法实现。

范例16 闭包作为方法的参数

```
//Find those items that qualify
def filter(list,predicate){
    return list.findAll(predicate)
}

//Two predicate closure
def isEven ={x ->return (x %2 ==0)}
def isOdd ={x ->return !isEven(x)}

def table =[11,12,13,14 ]

//Apply filter
def evens =filter(table,isEven)
println "evens:${evens}"

def odds =filter(table,isOdd)
println "odds:${odds}"
```

其输出结果说明变量evens的值是table中全部偶数值的列表。

```
evens: [12, 14]
odds: [11, 13]
```

闭包同样也可以被用作其他闭包的参数。在范例17中，引入了一个takeWhile闭包，它传递那些从原始List列表中查找出的，符合闭包参数所指定标准的值。

范例17 闭包作为另一个闭包的参数

```
//Find initial list that conforms to predicate
def takeWhile ={predicate,list ->
    def result =[]
    for(element in list){
        if(predicate(element)){
            result <<element
        }else
            return result
    }
    return result
}

//Two predicate closures
def isEven ={x ->return (x %2 ==0)}
def isOdd ={x ->return !isEven(x)}

def table1 =[12,14,15,18 ]
def table2 =[11,13,15,16,18 ]

//Apply takeWhile
```

```
def evens =takeWhile.call(isEven,table1)
println "evens:${evens}"
def odds =takeWhile(isOdd,table2)
println "odds:${odds}"
```

输出结果表明：变量evens拥有来自table1的偶数值列表。

```
evens: [12, 14]
odds: [11, 13, 15]
```

范例18中定义了multiply方法，它接收一个整型参数，并返回一个闭包。该闭包用来计算两个值的乘积，其中一个闭包参数为multiply方法提供的参数。变量twice是一个闭包，它返回某个参数的两倍值。根据同样的方式，闭包multiplication也接收一个整型参数，并返回一个闭包。和multiply一样，所返回的闭包会把其参数乘以某个指定的值。闭包quadruple会把参数乘以4。

范例18 闭包作为返回值

```
// Method returning a closure
def multiply(x) {
    return {y -> return x * y}
}

def twice = multiply(2)

println "twice(4): ${twice(4)}"

// Closure returning a closure
def multiplication = {x -> return {y -> return x * y} }

def quadruple = multiplication(4)

println "quadruple(3): ${quadruple(3)}"
```

输出结果也证明：twice闭包的返回值确实是其参数值的两倍值，闭包quadruple则返回其参数的四倍值。

```
twice(4): 8
quadruple(3): 12
```

本章的最后一个范例演示了闭包可以包含嵌套闭包的情形。在范例19中，selectionSort闭包对某个列表以升序方式排序。实现此闭包首先需要查找未排序列表的最小值，并且需要将它向前移动。将该值移动到队列前面的过程，实际上就是将最小值和其前面的值交换的过程。因此，实现selectionSort闭包需要minimunPosition和swap两个局部闭包，其中，前者用来在列表中查找列表中最小的元素值。

范例19 选择性排序

```
def selectionSort ={list ->

    def swap ={sList,p,q ->
        def temp =sList [p ]
        sList [p ]=sList [q ]
        sList [q ]=temp
    }

    def minimumPosition ={list ->
        var minIndex =0
        for (var i =1;i <list.length;i ++){
            if (list [i ]<list [minIndex ]) minIndex =i
        }
        minIndex
    }

    var list =list
    for (var i =0;i <list.length;i ++){
        var minIndex =minimumPosition .call(list)
        swap .call(list,minIndex ,i )
    }
}
```

```

    sList [q ]=temp
}

def minimumPosition =|pList,from ->
    def mPos =from
    def nextFrom =1 +from
    for(j in nextFrom..<pList.size()){
        if(pList [j ] <pList [mPos ])
            mPos =j
    }
    return mPos
}

def size =list.size()-1
for(k in 0..<size){
    def minPos =minimumPosition(list,k)
    swap(list,minPos,k)
}

return list
}

```

def table =[13,14,12,11,14]

def sorted =selectionSort(table)

println "sorted:\${sorted}"

执行此程序，其输出结果为：

sorted: [11, 12, 13, 14, 14]

在使用闭包时，必须特别注意变量和参数的作用域。有关闭包的更深入讨论请参见附录H和附录J。

9.4 习题

1. 请编写方法intersect，该方法有两个列表参数，其返回值为这两个列表的公共元素。
2. 请编写方法union，该方法有两个列表参数，分别返回第一个列表的所有元素值、第二个列表的所有元素值，以及这两个列表的所有元素值。
3. 请编写方法subtract，该方法有两个列表参数，其返回值为那些在第一个列表中出现，而第二个列表中没有出现的元素。
4. 使用闭包修改第6章中学习案例的代码，以格式化处理这些脚本的输出。
5. 假定有一个表示员工及其管理者的映射，如staff= ['Ken' : ['John', 'Peter'], 'Jon' : ['Ken', 'Jessie'], 'Jessie': ['Jim', 'Tom']]所示。这个映射说明了Ken是John和Peter的管理者，而Jessie的管理者则是Jon。请编写方法findManagerof(name, staff)通过员工姓名求其管理者的姓名，然后使用此方法编写方法findNoManager(staff)，以获取那些没有直接上司的员工列表。
6. 酒店模型通常可以使用一个内嵌映射的映射描述。最外层的映射由一个表示楼层的整型值索引，而与

每个楼层关键字相对应的值是一个内部映射，每个内部映射由一个表示房间号的整型值索引，而与每个房间号关键字相对应的值是酒店房间的具体类型列表。假定一个酒店映射的范例为：hotel = [1 : [1 : ['Bedroom', 2], 2 : ['Bedroom', 4], 3 : ['Studyroom', 10]], 2 : [1 : ['Bedroom', 4], 2 : ['Bedroom', 4]], 3 : [1 : ['Bedroom', 4], 2 : ['Conferenceroom', 25, 'Balmoral']]]。在这里，我们可以看出：一楼有三个房间，其中有两个卧室，一个书房。在这两个卧室中，有一个卧室是双人间，而另一个卧室则是四人间。书房能容纳10个人。酒店的顶层3层有一个能容纳25人，名为Balmoral的会议室。请编写方法printAllRooms(hotel)输出所有楼层及其房间号。同样，请编写方法printRoomsOnFloor(hotel, floorNumber)，输出给定楼层房间的使用情况。

7. 请说明闭包lSubtract和闭包rSubtract的用途：

```
def lSubtract = {x -> return {y -> return x - y}}
def rSubtract = {y -> return {x -> return x - y}}
```

对象p和q定义如下：

```
def p = lSubtract(100)
def q = rSubtract(1)
```

请说明下列语句的用途：

```
println "p(25): ${p(25)} "
println "q(9): ${q(9)} "
```

闭包comp定义如下：

```
def comp = {f, g -> return {x -> return f(g(x))}}
```

请详细解释其用法，并说明调用闭包产生的结果：

```
def r = comp(p, q)
def s = comp(q, p)
```

说明下列语句的执行结果：

```
println "r(10): ${r(10)} "
println "s(10): ${s(10)} "
```

8. 一个软件机构能够开发Groovy、Java和C#项目。每个项目都由一个或者多个程序员完成，其中，一个程序员可能会同时参与多个软件项目。举例来说，下列语句说明Ken、John和Jon都参与Groovy项目：

```
def softwareHouse = ['Groovy' : ['Ken', 'John', 'Jon'],
                    'Java' : ['Ken', 'John'],
                    'C#' : ['Andrew']]
]
```

请说明下列每个语句的作用：

- softwareHouse.each {key, value -> if(value.size() >= 2) println "\${value}"}
- softwareHouse['Groovy'].each {g ->
 softwareHouse['Java'].each {j ->

```
    if(g == j) println "${g}"
}
}
```

- 9.一所大学由多个部系组成，每个部系负责一个或者多个学科的教学任务。举例来说，下列语句说明计算机系由计算机专业和信息系统两个专业组成，它们分别有600和300个学生。

```
def university = ['Computing' : ['Computing' : 600, 'Information Systems' : 300],
                  'Engineering' : ['Civil' : 200, 'Mechanical' : 100],
                  'Management' : ['Management' : 800]
]
```

请说明下列语句的作用：

```
university.each {k, v ->
  v.each {ke, va ->
    if(va >= 300) println "${k}: ${ke}"
  }
}
```

- 10.开发闭包explode、implode和reverseString，作为闭包isPalindrome的局部闭包，基于此重新实现第7章中习题14的方法。

第10章 文 件

前面我们已经学习了很多程序，这些程序都有输出结果，同样也接收用户输入。这些输入输出都通过标准的输入设备（键盘）、输出设备（屏幕）实现。然而，这些简单程序对很多应用程序来说，都不具有代表性。在程序实践过程中，大部分程序都将数据永久存放在计算机的某个文件中。本章将讨论文件的处理方法。

10.1 命令行参数

Groovy程序通常存在于操作系统所建立的环境中。当执行程序时，系统支持给应用程序传递命令行参数的方法。如范例01所示，在Groovy脚本中，这些命令行参数是一个字符串队列，可以通过args变量访问。

范例01 命令行参数

```
println "args: ${args}"
println "size: ${args.size()}"
println "First arg: ${args[0]}"
```

如果通过如下命令执行此脚本，

```
groovy example01.groovy aaa bbb ccc
```

其输出结果为：

```
args: {"aaa", "bbb", "ccc"}
size: 3
First arg: aaa
```

通过上例，我们可以看出args变量仅包含这些命令行参数。使用方法size可以获得args参数列表的参数个数，其每个参数都可以按照常用的列表方法来引用。

10.2 File类

操作系统使用系统相关的路径字符串为文件和目录命名。File类可以提供抽象的、独立于系统的分级路径名视图。一个抽象路径名可以是零个或者多个字符串名称的序列。除了最后一个字符串，其余的字符串都表示目录层次。最后一个字符串可能表示一个文件名或者目录名。在一个抽象路径中，分隔符字符用来分隔每个字符串名。一些路径名范例如下所示：

```
myfile.txt          // simple file
docs/report.doc    // file in docs subdirectory
src/groovy/example01.groovy // file in nested subdirectories
src/groovy         // directory
c:/windows        // MS Windows directory and disk drive specifier
```

一个文件（File）对象能够表示某个文件或者某个目录。借助于File类的方法，我们可以以判断某个文件对象是否存在、是否表示文件或者目录、文件是可读还是可写，以及该文件的长度等。更进一步来说，Groovy已经通过接收闭包作为方法参数的方式扩充了File类。这些方法对于处理文件或者目录的路径来说特别有效。表10-1列举了一些常用的文件方法。在这里再次强调，使用了星号标记的方法是GDK的新增方法。举例来说，方法eachLine一行行地迭代处理文本文件，并应用闭包。

范例02是一个将文件内容制成表格的程序。闭包的行参数表示文件中的下一行值。行的最后一个字符不是本行字符的一部分，因此，需要使用println在单独的行输出，以分割文件的每行内容。范例所示的代码行创建一个新的文件对象，处理它，并在程序结束时关闭它。由于没有使用其他方式引用这个文件对象，因此并不需要通过一个变量来引用它。

表10-1 常用的文件处理方法

方法名称	函数原型/描述
append *	void append(String text) 将字符串text追加到文件末尾
createNewFile	Boolean createNewFile() 创建一个新空文档，当且仅当不存在使用这个文件名的文件时以抽象路径名命名
delete	Boolean delete() 删除由抽象路径名所表示的文件或者目录名
eachFile *	void eachFile(Closure closure) 为指定目录中的每个文件应用闭包
eachFileRecurse *	void eachFileRecurse(Closure closure) 为指定目录中的每个文件应用闭包，且对每个子目录使用递归方法
eachLine *	void eachLine(Closure closure) 逐行遍历指定的文档
exists	Boolean exists() 判断是否存在以此抽象路径名表示的文件或者目录
getPath	String getPath() 将抽象路径名转换成一个路径名字符串
getText *	String getText() 读取文件内容并将它作为一个字符串返回
isDirectory	Boolean isDirectory() 判断抽象路径名所表示的对象是否是一个目录
mkdir	Boolean mkdir() 创建一个抽象路径命名的目录名
withPrintWriter *	void withPrintWriter(Closure closure) 为文件创建一个新的PrintWriter辅助方法，并将它传递给闭包，并再次确认文件是否被关闭

范例02 以一次一行的方式读取并显示文件

```
import java.io.File

if(args.size() != 1)
    println 'Usage: example02 filename'
```

```
else {
    // Print each line of the file
    new File(args[0]).eachLine { line ->
        println "Line: ${line}"
    }
}
```

请注意，如果命令行参数指定的文件并不存在，则调用eachLine方法时会抛出一个异常。如果文件包含以下内容：

```
This is the first line
This is the second line
This is the third line
This is the fourth line
```

运行上述程序，其输出结果为：

```
Line: This is the first line
Line: This is the second line
Line: This is the third line
Line: This is the fourth line
```

在Unix操作系统中，有一个名为"wc"的实用程序。它可以扫描整个文本文档，并获得文件中字符总数、单词总数，以及文本行总数。在Groovy中，使用如下代码可以非常容易实现这个应用程序。

范例03 WC实用程序

```
import java.io.File

//Counters
def chars =0
def words =0
def lines =0

if(args.size()!=1)
    println 'Usage:example03 filename'
else {
    //Process the file
    new File(args [0 ]).eachLine{line ->
        chars +=1 +line.length()
        words +=line.tokenize().size()
        lines++
    }
    //Print the outcome
    println "chars:${chars};words:${words};lines:${lines}"
}
```

使用前一个范例所使用的文件将输出：

```
chars: 94; words: 20; lines: 4
```

File类同样也提供eachFile方法。通常用来表示某个目录的文件对象，它接收闭包作为其参数，且对该目录中每个文件调用此闭包。在下面的范例中，printDir方法接收目录名，并将它作为方法的参数。eachFile方法只不过是调用了系统所支持的listDir方法，该方法接收一个文件对象作为其第一个参数，一个整型参数作为第二个参数，在这里，文件对象表示一个目录。方法listDir调用文件对象的eachFile方法，闭包可以输出此目录下的所有文件名。如果在这些对象中含有子目录，则listDir方法将进行递归调用，其整型参数用于指定列表所需的缩进数，每次递归调用，该参数值都将增加。

范例04 目录列表

```
import java.io.File

        //List the content of a directory File
def listDir(dirFile,indent){
    dirFile.eachFile {file ->
        (0..<indent).each {print " "}
        println "${file.getName()}"
        if(file.isDirectory())
            listDir(file,2 + indent)
    }
}

        //Print the content of a named directory
def printDir(dirName){
    listDir(new File(dirName),0)
}

if(args.size()!=1 || new File(args [0 ]).isDirectory()==false)
    println 'Usage:example04 directory'
else {
    //Print the current directory
    printDir(args [0 ])
}
```

File类同样也支持eachFileRecurse方法。如其名所示，它可以遍历某个目录中的所有文件，并且对子目录上使用递归方法遍历。举例来说，可以使用如范例05所示的方法标识那些大于某个长度值的文件。

范例05 目录的递归访问

```
import java.io.File

        //List those files exceeding a given size
def printDir(dirName,size){
    new File(dirName).eachFileRecurse {file ->
        if(file.length()>size)
            println "${file.getName()}"
```

```
}

if(args.size()!=2 ||new File(args [0 ]).isDirectory()==false)
    println'Usage:example05 directory'
else {
    //List from the current directory
    printDir(args [0 ],args [1 ].toInteger())
}
```

在对象PrintWriter的帮助下，我们可以将一个文件的内容拷贝至另一个文件中。PrintWriter类通常用来格式化输出文件的对象。PrintWriter与File类和eachLine结合使用能获得强大功能。首先，它需要检查是否存在目的文件，如果存在则删除它。File类中提供newPrintWriter方法，可以为给定的目的文件传递PrintWriter对象，这样便可从源文件中拷贝每个代码行至目标文件。

范例06 文件拷贝

```
import java.io.*

if(args.size()!=2)
    println 'Usage:example06 filename filename'
else {
    //Write to a destination file
    def outFile =new File(args [1 ])
    if(outFile.exists())
        outFile.delete()

    //Create a PrintWriter
    def printWriter =outFile.newPrintWriter()

    //Copy each line of the file
    new File(args [0 ]).eachLine {line ->
        printWriter.println(line)
    }
    //Close up
    printWriter.flush()
    printWriter.close()
}
```

同样，File类也提供了大量的支持输入/输出(参见GDK文档)的辅助方法。举例来说，方法withPrintWriter首先为文件创建了一个新的PrintWriter对象，然后将该对象传递给闭包，并确保随后关闭这个对象。其他的辅助方法包括withInputStream、withOutputStream、withReader以及withWriter。范例07使用PrintWriter重新实现了上一个范例。

范例07 使用PrintWriter方法的文件拷贝

```
import java.io.*

if(args.size()!=2)
```

```

    println 'Usage:example07 filename filename'
else {
    //Write to a destination file
    new File(args [1]).withPrintWriter {printWriter ->

        //Copy each line of the file
        new File(args [0]).eachLine {line ->
            printWriter.println(line)
        }
    }
}

```

对文本文件进行排序是非常常见的功能。对于小型乃至中型文件来说，在Groovy中实现排序是比较简单的，这是因为列表提供sort方法。我们可以将文件的每行数据放到一个列表中，并执行列表的内部排序，然后将排序后结果写回到同一个文件中，其实现方法如范例08所示。

范例08 对文件排序

```

import java.io.*

if(args.size()!=1)
    println 'Usage:example08 filename'
else {
    def lines =[]

    //Read from the text file
    new File(args [0]).eachLine {line ->
        lines <<line
    }

    //Sort the text
    lines.sort()

    //Write back to text file
    new File(args [0]).withPrintWriter {printWriter ->
        lines.each {line ->
            printWriter.println(line)
        }
    }
}

```

最后，请思考如下形式的数据文档：

```

John 2:30PM
Jon 10:30AM
// ...

```

上面这个数据文档是用来保存日志记录信息。假定用户希望以时间顺序产生每天所发生事件的报告。可以将文件的每个记录都放到一个列表中，则可以基于时间进行排序。sort方法可

以接收一个用来比较值大小的闭包作为参数。使用正则表达式可以从每个行记录中提取出时间值(参见第3章和附录D)，其中，正则表达式的形式如下：

(\\d{1,2}):(\\d{2})([AP]M)

通过数字和后缀的组合可以提取出单个元素的时间值以进行比较。完整的日志数据正则表达式如下所示：

(\\w*)\\s((\\d{1,2}):(\\d{2})([AP]M))

上述代码在范例09中给出，请注意使用的compareTo方法和“<=>”操作符效果相同。

范例09 日志输出

```
if(args.size()!=1)
    println 'Usage:groovy9 filename'
else {
    def TIME_PATTERN ='(\\w *)\\s((\\d{1,2}):(\\d{2})([AP ]M))'
    def diary =[]

    //read the file
    new File(args [0 ]).eachLine {entry ->
        diary <<entry
    }

    //sort the entries
    diary.sort {entry1,entry2 ->
        def matcher1 =entry1 =~TIME_PATTERN
        def matcher2 =entry2 =~TIME_PATTERN
        matcher1.matches()
        matcher2.matches()

        def cmpMeridian =matcher1 [0 ][5 ]<=>matcher2 [0 ][5 ]
        def cmpHour =matcher1 [0 ][3 ].toInteger()<=>matcher2 [0 ][3 ].toInteger()
        def cmpMinute =matcher1 [0 ][4 ].toInteger()<=>matcher2 [0 ][4 ].toInteger()
        return ((cmpMeridian !=0)?cmpMeridian :(cmpHour !=0)?cmpHour :cmpMinute)
    }

    println 'Diary events'
    diary.each {entry ->println "${entry}"}
}
```

10.3 习题

1. 重新实现范例07中的文件拷贝功能，并将文本行距设置为两行。
2. 编写把一个文本文件的内容复制到另一个文本文件的程序，并删除源文本文件中的空行。文件名是通过命令行参数指定的。
3. 编写一个给文本文件加入行号的程序，其中，输入文件名通过命令行参数指定，程序输出到标准输出

设备。

4. 编写一个用来合并指定的输入文件，并输出到标准输出设备的程序。命令行参数可能会提供一个或者多个文件名。
5. 编写一个功能类似于Unix grep实用程序的程序。该程序接收由某个模式和某个文本文件名组成的命令行参数。程序将输出那些符合给定模式的代码行。
6. 为范例08的实现程序增加一个命令行可选项-r，以逆序排序。

第11章 学习案例：图书馆应用 程序（方法、闭包）

本章将通过构建第6章引入的小型学习案例的解决方案，演示Groovy方法和闭包的超级功能。和以往一样，我们首先将为维护图书馆借阅数据建立一个简单模型，图书馆模型主要负责维护库存图书记录，以及已经借阅一本或者数本图书的借阅者姓名。

在第6章学习案例的开发过程中，我们使用了迭代方式开发图书馆应用程序。这样，我们向应用程序添加功能就处于可控状态，同时也能确保尽可能早地实现一个（局部）解决方案。迭代1将演示我们需要完成的功能，迭代2则实现一个基于文本的、简单的命令行用户交互界面，迭代3将简化迭代2的实现代码。我们的最终目标是使实现代码具有可读性和可维护性。

11.1 迭代1：需求规范和映射实现

需求说明是管理和维护图书馆的借阅记录，我们必须实现如下几个方面的功能：

- 对借阅数据库的操作，比如添加/删除书籍。
- 某本书的借阅和归还记录。
- 显示当前借阅数据库的详细信息。
- 显示某借阅者所借出的图书数目。
- 显示某本书籍的借阅者数量。

一旦以用户用例建立应用程序的外部视图，就可以使用多种方法进行建模。在第6章中，我们介绍了两种可行的解决方案，即使用列表和映射数据结构。在迭代1中，我们将使用映射表示图书馆，原因之一是映射是一种适合高效信息存储和检索的解决方案。可以预见使用此方法将对图书馆应用程序的实现非常有帮助。

首先，我们假定映射的关键字为每本书的书名，与之相对应的值是其借阅者的姓名列表。在这里假设所有书籍的每个借阅者姓名都会存储在同一个列表中。映射的另一个实用特征是其关键字的唯一性，这样就消除了图书馆数据库中重复输入同一个书名的可能性。

映射中每个关键字所对应的值都是由零个或者多个字符串元素组成的列表，在这里，字符串用来表示借阅者姓名。请注意，列表中可能包含相同的元素，这是因为在前面的假设中，已经假定同一个借阅者可以借阅多本书籍。列表允许为空。当列表为空时，书名是其关键字，表示当前该书籍没有被借出。一个简单的借阅数据库初始化方法为：

```
def library = ['Groovy':['Ken','John'], 'OOD':[['Ken']], 'Java':[['John','Sally']], 'UML': ['Sally'], 'Basic':[]]
```

图11-1 是最终的映射数据结构。该图说明Ken和John都借阅了Groovy，Ken还借阅了OOD，John和Sally都借阅了Java，Sally借阅了UML图书。请注意书名为Basic的书籍没有借出，其对应的值是一个空列表。

由于这个应用程序的功能需求较为简单，因此，使用Groovy方法实现起来非常容易。每个方法将分别实现前面确定的相关功能需求。由于其中的某些功能过于复杂，因此我们可以将其分解为若干个简单方法。请注意，这里虽然使用了术语“方法”，但由于此学习案例的实现采取高效的过程式开发方法（Deitel, 2003），其实际上和上下文中的术语“过程”功能相似。在很多脚本应用程序中，只需要使用一些简单的过程性代码即可实现解决方案。

为了访问与某个指定书名相关的借阅者姓名列表，需要使用映射的索引操作符“[]”（参见 GDK），在这里，我们使用书名做为索引值，这在一定程度上可以简化代码编写量。举例来说，添加一本新书可以使用如下代码：

```
def addBook(library, bookTitle) {
    library[bookTitle] = []
}
```

欲获取某本书的借阅者数量，可以使用下面的代码：

```
def readNumberBorrowers(library, bookTitle) {
    return library[bookTitle].size()
}
```

如Library01的部分代码所示，程序所使用的大部分方法都相当简单。为清晰起见，在这里故意使用一些简单方法，本书将会在后面的章节中修改这些方法。

LIBRARY 01 数据库应用程序——方法

```
def addBook(library,bookTitle){
    library [bookTitle ]= []
}

def removeBook(library,bookTitle){
    library.remove(bookTitle)
}

def lendBook(library,bookTitle,borrowerName){
    library [bookTitle ]<<borrowerName
}

def returnBook(library,bookTitle,borrowerName){
    library [bookTitle ].remove(borrowerName)
}

def displayLoanStock(library){
    println "Library stock:${library}\n"
}

def readNumberBorrowedBooks(library,borrowerName){
```

Key	Value
Groovy	Ken, John
OOD	Ken
Java	John, Sally
UML	Sally
Basic	

图11-1 数据库应用程序的映射数据结构范例

```

    //
    //get a List of each List of the borrower names from the library
    def borrowerNames =library.values().asList()
    //
    //create a single List of the borrower names
    borrowerNames =borrowerNames.flatten()
    //
    //return the number of borrower names in the List
    return borrowerNames.count(borrowerName)
}

def readNumberBorrowers(library,bookTitle){
    return library [bookTitle ].size()
}

//More code follows ...

```

请注意，所有的方法都将library作为形参，这种程式应用程序开发方法会操作某个主要的数据结构。在本案例的实现过程中，图书馆数据库使用映射实现。

为了测试所编写的程序是否能够实现期望的结果，需要为每个用户用例开发一个测试用例。在此案例的简化版本中，我们仅使用了首先输入假定的理想数据，然后根据程序的可视化输出检测其功能的方式。举例来说，为了测试借阅数据库，仅需要在屏幕控制台上输出相关的映射信息，然后将可视化的输出结果与所期望的映射初始值进行比较。更明确地说，这种方法能够完成相应功能，但不是那么实用。本书稍后将在第15章更详细地讨论测试方法。不论使用何种方法，对目前来说，这种方法都已经够用了。

和应用程序的迭代开发一样，逐步引入测试用例的方法通常都很凑效，换句话说，也就是逐次添加一个测试用例。这将减轻测试所引起的、可能超过用户忍受限度的风险，并建立用户测试程序的信心。举例来说，下面的脚本演示了测试用例的基本使用方法。

LIBRARY 01 数据库应用程序——第一个测试用例

```

//methods as shown previously
//...

//Initialize the loan stock
def library =[ 'Groovy':['Ken','John'], 'OOD':['Ken'], 'Java':['John','Sally'], 'UML':['Sally'],
'Basic':[]]

//Test Case:Display loan stock
println 'Test Case:Display loan stock'
displayLoanStock(library)

```

令人欣慰的是，测试程序输出了所期望的结果：

```

Test Case: Display loan stock
Library stock: ["Groovy":["Ken", "John"], "UML": ["Sally"], "Java": ["John", "Sally"], "OOD": ["Ken"],
"Basic": []]

```

下面将增加另一个测试用例。

LIBRARY 01 数据库应用程序——第二个测试用例

```
// methods, initialization and test case as shown previously
// ...

// Test Case: Add a new book
println 'Test Case: Add a new book'
addBook(library, 'C#')
displayLoanStock(library)
```

测试用例再次输出了所期望的结果：

```
Test Case: Display loan stock
Library stock: ["Groovy":["Ken", "John"], "UML":["Sally"], "Java":["John", "Sally"], "OOD":["Ken"], "Basic":[]]
Test Case: Add a new book
Library stock: ["Groovy":["Ken", "John"], "UML":["Sally"], "Java":["John", "Sally"], "OOD":["Ken"], "Basic":[], "C#":[]]
```

在向应用程序添加测试用例的过程中，我们将一直采用这种逐次添加测试用例方式，以检查和修正必要的代码。

```
//methods, initialization and test case as shown previously
//...

//Test Case:Remove a book
println 'Test Case:Remove a book'
removeBook(library,'UML')
displayLoanStock(library)

//Test Case:Record a book loan to a borrower
lendBook(library,'Java','Ken')
println 'Test Case:Record a book loan to a borrower'
displayLoanStock(library)

//Test Case:Record a book return by a borrower
returnBook(library,'Java','Sally')
println 'Test Case:Record a book return by a borrower'
displayLoanStock(library)

//Test Case:Display the number of books on loan to a borrower
println 'Test Case:Display the number of books on loan to a borrower'
println "Number of books on loan to Ken:${readNumberBorrowedBooks(library,'Ken')}\n"

//Test Case:Display the number of borrowers of a book
println 'Test Case:Display the number of borrowers of a book'
println "Number of borrowers of Java:${readNumberBorrowers(library,'Java')}\\n"
```

最后的输出结果为：

```
Test Case:Display loan stock
```

```
Library stock:[{"Groovy": ["Ken", "John"], "UML": ["Sally"], "Java": ["John", "Sally"], "OOD": ["Ken"], "Basic": []}]  
  
Test Case: Add a new book  
Library stock:[{"Groovy": ["Ken", "John"], "UML": ["Sally"], "Java": ["John", "Sally"], "OOD": ["Ken"], "Basic": [], "C#": []}]  
  
Test Case: Remove a book  
Library stock:[{"Groovy": ["Ken", "John"], "Java": ["John", "Sally"], "OOD": ["Ken"], "Basic": [], "C#": []}]  
Test Case: Record a book loan to a borrower  
Library stock:[{"Groovy": ["Ken", "John"], "Java": ["John", "Sally", "Ken"], "OOD": ["Ken"], "Basic": [], "C#": []}]  
  
Test Case: Record a book return by a borrower  
Library stock:[{"Groovy": ["Ken", "John"], "Java": ["John", "Ken"], "OOD": ["Ken"], "Basic": [], "C#": []}]  
  
Test Case: Display the number of books on loan to a borrower  
Number of books on loan to Ken:3  
  
Test Case: Display the number of borrowers of a book  
Number of borrowers of Java:2
```

案例的第一次迭代实现到此结束，其脚本代码可以在本书的相关web站点获得。

11.2 迭代2：基于文本的用户交互界面的实现

前面已经证实了图书馆应用程序的执行结果和预期相同，现在把焦点转到用户如何与该程序进行交互。毫无疑问，用户交互界面的实现方法很多。但是，我们仍然选择最简单的方法，基于文本的、命令行的交互方式。在本书的后续章节中，我们将介绍更多更详细的替代方法。

在迭代2中，我们需要给用户提供一个基于文本的选项菜单。用户一旦选择了某个选项，它就将执行相关操作，并且该菜单仍将再次呈现，以便于用户选择其他选项。程序将一直运行，直到用户选择Quit选项为止。

为了获得更好的效果，学习案例使用了第8章中讨论的程序流程控制结构。比如，使用while循环语句控制菜单的反复显示，使用if...else语句选择和处理用户所选的选项。

同样，在必要时也可以引入一些新方法。比如，我们必须能够获取书名、借阅者姓名，以及用户所选的选项。这些需求分别通过readBookTitle、readBorrowerName和readMenuSelection方法实现。部分代码如Library 02所示。欲获得完整代码，请访问本书相关站点。

LIBRARY 02 图书馆应用程序——其他方法和程序控制结构

```
import console.*  
  
//methods as shown previously  
//...  
  
def readBookTitle(){  
    print('\tEnter book title:')
```

```
        return Console.readLine()
    }

def readBorrowerName(){
    print('\tEnter borrower name:')
    return Console.readLine()
}

def readMenuSelection(){
    println()
    println('0:Quit')
    println('1:Add new book')
    println('2:Remove book')
    println('3:Lend a book')
    println('4:Return a book')
    println('5:Display loan stock')
    println('6:Display number of books on loan to a borrower')
    println('7:Display number of borrowers of a book')

    print('\n \tEnter choice:')
    return Console.readLine()
}

def library =['Groovy':['Ken', 'John'], 'OOD':['Ken'], 'Java':['John', 'Sally'], 'UML':['Sally'],
'Basic' :[]]

def choice =readMenuSelection()

while(choice !='0'){

    if(choice =='1')
        addBook(library,readBookTitle())
    else if(choice =='2')
        removeBook(library,readBookTitle())
    else if(choice =='3')
        lendBook(library,readBookTitle(),readBorrowerName())
    else if(choice =='4')
        returnBook(library,readBookTitle(),readBorrowerName())
    else if(choice =='5')
        displayLoanStock(library)
    else if(choice =='6'){
        def count =getNumberBorrowedBooks(library,readBorrowerName())
        println "\nNumber of books borrowed:$count\n"
    }else if(choice =='7'){
        def count =getNumberBorrowers(library,readBookTitle())
        println "\nNumber of borrowers:$count\n"
    }else
        println ('\nUnknown selection \n')
}
```

```
//next selection  
choice =readMenuSelection()  
}  
  
println('\nSystem closing \n')
```

请注意，程序并没有改动数据库初始化方法行业的其他支持方法，在这里并不需要编写测试用例了。为了测试Library 02是否到达预期效果，可以通过选择每个按钮选项，以输出数据库具体内容的方式检查其结果。一个典型的用户交互过程如下所示，用户输入用黑斜体表示：

```
0:Quit  
1:Add new book  
2:Remove book  
3:Lend a book  
4:Return a book  
5:Display loan stock  
6:Display number of books on loan to a borrower  
7:Display number of borrowers of a book
```

```
Enter choice: 3  
Enter book title: Java  
Enter borrower name: Ken
```

```
0:Quit  
1:Add new book  
2:Remove book  
3:Lend a book  
4:Return a book  
5:Display loan stock  
6:Display number of books on loan to a borrower  
7:Display number of borrowers of a book
```

```
Enter choice: 5  
Library stock:[ "Groovy":[ "Ken", "John"], "UML":[ "Sally"], "Java":[ "John", "Sally", "Ken"], "OOD":[ "Ken"],  
"Basic":[] ]
```

由于程序已经实现了预期的输出结果，学习案例的第二次迭代实现到此结束。

11.3 迭代3：使用闭包实现

迭代3没有新的功能性需求，其主要目的是重新编写迭代2中的代码，以增强其可阅读性和可维护性。Library 02代码存在的一个潜在问题是，通过if...else语句控制代码的执行流程，在增加更多可选择项的同时，可能会增加代码的复杂度。虽然可以使用switch语句替换if...else语句，但由于switch语句也只能简单地改变语法结构，并不能真正起到有效的作用。

Groovy语言的一个非常强大的特征是第9章所讨论的闭包。它所表示的可执行代码块同样也是一个对象。正因为如此，它也可以是映射中的一个值。如果映射中关键字是一个用户选项，那就可以确定和执行与之相关的闭包，这样程序就不需要任何复杂的流程控制代码。

举例来说，doAddBook闭包就可以实现迭代2中addBook方法。

```
def doAddBook = { addBook(library, readBookTitle()) }
```

如果为每个可选择项都编写一个类似的闭包，那么就可以得到一个以用户选项为关键字、与之相应的值为闭包的映射。

```
def menu = ['1' : doAddBook,
           '2' : doRemoveBook,
           '3' : doLendBook,
           '4' : doReturnBook,
           '5' : doDisplayLoanStock,
           '6' : doDisplayNumberBooksOnLoanToBorrower,
           '7' : doDisplayNumberBorrowersOfBook
         ]
```

上述结构通常称作是查找表(lookup table)或者分配表(dispatch table)。这种方式非常实用，可以使用一个查找表替换复杂的程序控制代码，如下所示：

```
def choice = readMenuSelection()

while(choice != '0'){
    menu[choice].call()
    choice = readMenuSelection()
}
```

Library 03的部分代码集成了这些思想。和前面的范例一样，欲获得完整代码请访问本书相关站点。

LIBRARY 03 图书馆应用程序——重构的版本

```
//methods and initialization as shown previously
//...

def doAddBook ={addBook(library,readBookTitle())}

def doRemoveBook ={removeBook(library,readBookTitle())}

def doLendBook ={lendBook(library,readBookTitle(),readBorrowerName())}

def doReturnBook ={returnBook(library,readBookTitle(),readBorrowerName())}

def doDisplayLoanStock ={displayLoanStock(library)}

def doDisplayNumberBooksOnLoanToBorrower ={ 
    def count =getNumberBorrowedBooks(library,readBorrowerName())
    println "\nNumber of books borrowed:${count}\n"
}

def doDisplayNumberBorrowersOfBook ={ 
    def count =getNumberBorrowers(library,readBookTitle())
    println "\nNumber of borrowers:${count}\n"
```

```

}

def menu =['1' :doAddBook,
           '2' :doRemoveBook,
           '3' :doLendBook,
           '4' :doReturnBook,
           '5' :doDisplayLoanStock,
           '6' :doDisplayNumberBooksOnLoanToBorrower,
           '7' :doDisplayNumberBorrowersOfBook
]

def choice =readMenuSelection()
while(choice !='0'){
    menu [choice ].call()
    choice =readMenuSelection()
}

println('\nSystem closing \n')

```

如所期望的那样，其执行结果和Library 02的执行结果相同。但是，通过闭包的方式重新编写相关代码，向程序添加附加功能就会变得非常容易。举例来说，程序可能需要增加一个选项，这个选项能按字母对给定书籍的借阅者姓名列表进行排序。在这里，所有需要做的工作仅仅是编写一个能够调用合适方法的闭包：

```

//method
def getBorrowers(library,bookTitle){
    return library [bookTitle ]
}

//closure
def doDisplayBorrowersOfBook ={
    def borrowerNames =getBorrowers(library,readBookTitle())
    println "\nBorrowers:${borrowerNames.sort()}\\n"
}

```

将闭包及其适当的关键字添加到映射中去：

```
menu['8'] = doDisplayBorrowersOfBook
```

然后更新readMenuSelection方法：

```

def readMenuSelection(){

    // As shown previously
    // ...

    println('8: Display borrowers of a book')
    // ...
}


```

最重要的一点是必须认识到，我们并没有改动程序的其他部分代码。特别地，这不会使现有代码更加复杂。案例的第三次迭代实现到此结束。

11.4 习题

1. 第3章3.1节讨论了在Groovy中多行文本字符串的使用方法。请重新编写Library 02中的readMenuSelection方法以实现其功能，并分别说明两种实现方式的优缺点。
2. 修改习题1中的方法，能够提供一个新的选型，使它支持以字母顺序显示给定书籍的借阅者姓名列表。
3. JDK有一个HashSet类，和ArrayList一样，它可以用来初始化列表。但是，它不支持重复元素。使用HashSet修改前面习题中的方法，使它实现一个新的选项，以字母顺序显示所有已经借出，并且不含重复书籍名的列表。
4. 修改前面习题中的方法，使它支持一个新的选项，以字母顺序显示图书馆中所有书籍列表。
5. 在案例实现迭代3中，每个闭包都调用一个方法。请重新编写Library 03的代码，使每个闭包都调用一个内嵌闭包（参见9.3节），以替换这个方法。并说明这种方法的优点。
6. 假定有两个列表，分别表示参与两个独立项目的开发人员姓名，其中，一个员工有可能只参与一个项目，也可能同时参与两个项目。请编写代码，以获取同时参与两个项目的员工，以及只参与一个项目的员工。然后，再编写方法，以获取只参与第一个项目，而不参与第二个项目的员工。
7. 请思考第7章习题05的时间处理程序。在执行过程中，程序将分别创建hours、minutes、seconds和totalSeconds变量。当执行完前两条语句后，Groovy程序将为变量hours和变量minutes创建对象。

这些对象一旦被创建，Groovy运行环境就将一直保留这些对象。可以使用变量name及其对应值组成的一个映射为此运行环境建模。在执行完前两条语句后，我们将得到如下结果：

```
environment = ['hours' : 1, 'minutes' : 2]
```

请编写add、getVariables和getValue方法，分别用来：

- (a) 向environment增加一个新变量及其相应的值
- (b) 获取当前环境中所有变量
- (c) 获取与给定变量相对应的值

8. 请思考附录G习题12所示的单词一致性比较程序。该程序的运行环境需要管理程序块结构，程序的执行从定义变量doc开始，然后调用concordance方法。concordance方法的前两行代码用来引入变量lineNumber和concord。这两个局部变量仅在concordance方法执行时才存在。一旦程序从该方法返回，这些变量将从运行环境中删除，随后添加concord变量到整个程序的运行环境中。

和前面的习题一样，请使用一个条目为映射的列表为此应用程序建立数据模型。当输入一个新的代码块时，列表中将会增加一个新的映射；当退出代码块时，将会删除该映射。作为引入到环境中的变量来说，这些变量都会记录在列表的最近映射中。

请编写程序实现以下方法：

- (a) addBlock：向运行环境中添加一个新的代码块
- (b) removeBlock：模拟退出代码块时的情形
- (c) add：在运行环境中引入一个新变量
- (d) lookup：获取给定变量的值

第12章 类

在Groovy中，类将数据和操作这些数据的方法封装在一起。类中的数据和方法通常用来描述现实世界中的某类事物。举例来说，如果要开发一个银行应用程序，可能会需要一些类来表示账户对象、银行对象，甚至客户对象。类似地，在一个高校学生记录系统中，也可能会需要一些类来表示学生对象、课程和模块对象，以及表示教学计划程序的对象。

请仔细观察，这些现实世界中的对象在应用程序中可能是以物理形式存在，或者以一些非常容易理解的抽象实体概念存在。在本章的范例中，学生对象肯定真实存在，而在教学计划程序中确不会实际存在。

12.1 类

Groovy中的类被用于表示问题域的抽象概念。如前所述，类的定义就是定义类的状态（数据）和行为。因此，Groovy类同时描述该类的实例字段和方法。属性定义了该类对象所具有的属性，方法则定义了类的行为。

下面是一个简单的Groovy脚本范例，它定义了一个用来简单描述银行账户的类，以及创建实例和显示其属性值的代码。

范例01 一个简单的Groovy类

```
class Account {  
    def number          //account number  
    def balance         //current balance  
  
    //create a new Account instance  
    def acc =new Account(number :'ABC123',balance :1200)  
  
    //display its state values  
    println "Account ${acc.number} has balance ${acc.balance}"  
}
```

该脚本的输出结果为：

```
Account ABC123 has balance 1200
```

类名要紧跟在Groovy关键字class之后，在这里也就是Account，它有两个公有属性，但没有定义任何方法。使用new操作可以创建一个Accout类的实例，如下所示：

```
def acc = new Account(number : 'ABC123', balance : 1200)
```

要创建对象的类名必须紧跟在new关键字之后，余下的代码由一个命名属性的列表组成，说明会被初始化的属性。在这里，使用一个名为acc的Account对象，拥有两个属性：银行账户ABC123和金额1200。请仔细观察print语句引用实例属性的方法，表达式acc.number用来访问

Account对象acc的number属性。

看起来如此简单的类，其中也隐含着大量信息。首先，类Account所引入的两个属性拥有公共访问权限，即在代码的任何地方是否可以用它们来获取类Account的某个实例的具体属性值。这就是print语句能够访问acc对象属性的原因。

其次，该简单范例演示了Groovy获取实例属性和方法的方式。属性消除了某个实例字段（有时也称为特性）与其方法之间的差别。从客观的角度看Groovy类，属性不仅和实例字段，而且和其getter/setter方法都非常相似。比如，acc.number所示的属性引用方法实际上是通过acc.getNumber()实现的。

类是创建对象实例的模板。在下一个范例中，我们将创建两个Account对象，并输出它们的值。

范例02 两个对象实例

```
class Account {
    def number          //account number
    def balance         //current balance
}

//create two instances
def acc1 = new Account(number : 'ABC123',balance :1200)
def acc2 = new Account(number : 'XYZ888',balance :400)

//report on both
println "Account ${acc1.number} has balance ${acc1.balance}"
println "Account ${acc2.number} has balance ${acc2.balance}"
```

运行该程序，结果如下：

```
Account ABC123 has balance 1200
Account XYZ888 has balance 400
```

我们注意到，每个Groovy类都有与之相应的Java等价类。这也就是说，方法getter和方法setter都是Groovy类的隐含方法。因此，可以混合使用它们，如下例所示。

范例03 隐式调用getter和setter方法

```
class Account {
    def number          //account number
    def balance         //current balance
}

//create two instances
def acc1 = new Account(number : 'ABC123',balance :1200)
def acc2 = new Account(number : 'XYZ888',balance :400)

//access the state using properties
println "Account ${acc1.number} has balance ${acc1.balance}"

//access the state using getters
```

```
println "Account ${acc2.getNumber()} has balance ${acc2.getBalance()}"  
  
        //modify the state using a property  
acc1.balance = 200  
println "Account ${acc1.getNumber()} has balance ${acc1.getBalance()}"  
  
        //modify the state using a setter  
acc2.setBalance(600)  
println "Account ${acc2.number} has balance ${acc2.balance}"
```

其输出结果为：

```
Account ABC123 has balance 1200  
Account XYZ888 has balance 400  
Account ABC123 has balance 200  
Account XYZ888 has balance 600
```

在很多情况下，类的一些方法也可以定义实例的独特行为。在类Account中，我们可能会希望它拥有支持存款和提款操作的方法。为实现此目的，我们向Account类增加了credit和debit方法。另外，还增加了一个显示Account对象属性值的方法，如范例04所示。

范例04 类方法

```
class Account {  
    def number          //account number  
    def balance         //current balance  
  
    def credit(amount){  
        balance += amount  
    }  
  
    def debit(amount){      //only if there are sufficient funds  
        if(balance >= amount)  
            balance -= amount  
    }  
  
    def display(){  
        println "Account:${number} with balance:${balance}"  
    }  
}  
  
//create a new instance  
def acc = new Account(number : 'ABC123', balance : 1200)  
acc.display()  
  
        //credit transaction  
acc.credit(200)           //balance now 1400  
acc.display()  
  
        //other transactions
```

```

acc.debit(900)           //balance now 500
acc.debit(700)           //balance remains unchanged at 500
acc.display()

```

其输出结果为：

```

Account: ABC123 with balance: 1200
Account: ABC123 with balance: 1400
Account: ABC123 with balance: 500

```

因为Account是个对象，Account类的实例对象也可以被用作列表元素。在范例05中，我们创建Account类的三个实例，并将它们存放在一个列表中，然后分别显示其值。请注意在此版本中，Account类已经使用toString方法代替了display方法，该替代方法会把实例的属性值作为一个字符串返回。

范例05 account对象列表

```

class Account {
    def number           //account number
    def balance          //current balance

    def credit(amount){
        balance +=amount
    }

    def debit(amount){   //only if there are sufficient funds
        if(balance >=amount)
            balance -=amount
    }

    def toString(){      //see also next example
        return "Account:${number}with balance:${balance}"
    }
}

//create some instances
def acc1 = new Account(number : 'ABC123',balance :1200)
def acc2 = new Account(number : 'PQR456',balance :200)
def acc3 = new Account(number : 'XYZ789',balance :123)

//populate a list with the instances
def accounts =[acc1,acc2,acc3 ]

//now display each
accounts.each {acc ->
    println acc.toString()
}

```

其输出结果为：

```
Account: ABC123 with balance: 1200  
Account: PQR456 with balance: 200  
Account: XYZ789 with balance: 123
```

为了表示Account对象的实际含义，我们在程序中定义了toString方法，然后把它作为print语句的一部分用在表达式acc.toString()中。为了避免显式地调用toString方法，程序重新定义（参见第14章）了toString方法，这需要重新编写toString方法，并让其返回一个字符串值。其另一个版本如范例06所示。

范例06 重新编写的toString方法

```
class Account {  
    def number          //account number  
    def balance         //current balance  
  
    def credit(amount){  
        balance +=amount  
    }  
  
    def debit(amount){           //only if there are sufficient funds  
        if(balance >=amount)  
            balance -=amount  
    }  
  
    String toString(){          //redefinition  
        return "Account:${number}with balance:${balance}"  
    }  
}  
  
//populate a list with the instances  
def accounts =[new Account(number :'ABC123',balance :1200),  
              new Account(number :'PQR456',balance :200),  
              new Account(number :'XYZ789',balance :123)]  
  
//now display each  
accounts.each {acc ->  
    println acc           //automatically call toString  
}
```

该范例的输出结果和上一个范例的输出结果相同。

在Java中，Account类通常包含初始化类对象的构造器方法。在Groovy中并不需要做此工作，只需使用new操作符和指定的参数即可替代。当然，在Groovy中同样也有构造器方法。为了显式地提供一个构造器方法，通常希望在类的定义中包含参数化的构造器以初始化类的属性。构造器和方法的最大区别在于构造器名和类名相同。如范例07所示。

范例07 一个构造器方法

```

class Account {

    def Account(number,balance){      //constructor method
        this.number =number
        this.balance =balance
    }

    def credit(amount){
        balance +=amount
    }

    def debit(amount){                //only if there are sufficient funds
        if(balance >=amount)
            balance -=amount
    }

    String toString(){               //redefinition
        return "Account:${number}with balance:${balance}"
    }

    def number                      //account number
    def balance                      //current balance
}

//populate a list with the instances
def accounts = [new Account('ABC123',1200),
               new Account('PQR456',200),
               new Account('XYZ789',123)]

//now display each
accounts.each {acc ->
    println acc                  //automatically call toString
}

//def acc =new Account(number :'ABC123',balance :1200)      //No matching constructor

```

请仔细观察为Account类创建实例的方式。通过调用构造器方法，传递两个分别用来表示账户和初始账户金额的实参来新建Account实例。在这里，我们使用了位置参数（positional parameters）。另外，请注意构造器方法的定义方式，其形参和其属性名相同。为了在方法内区分它们，在属性的前面使用了this关键字。于是，表达式this.number =number可以解释为“把number赋值给该对象的number属性”。

请注意最后一个代码行注释。当类包含一个用户自定义的构造器后，程序并不会自动生成默认的构造器。最后一条语句试图以这个常用的方式来创建一个新的Account对象。但是，由于这需要默认的构造器，系统会提示报错：用户没有与属性相匹配的构造器可以使用（参见后

面的进一步说明)。

在此版本的Account类中，最后一个代码行注释说明在类定义的最后部分列出类的属性。在Groovy中，这是完全可行的，甚至在引入这些属性之前，也可以使用相关方法来引用它们。这样做的原因是使用户认识到：使用这样的类开发客户程序，对类支持服务比其实现更受关注。

在使用new操作符时，我们宁愿使用命名参数模式。这种方法的操作过程非常值得思考。类的属性将会导致自动生成getter方法和setter方法。更进一步来说，在没有定义构造器时，编译器将创建默认的构造器。

```
Account() {  
}
```

对象创建代码：

```
def acc = new Account(number : 'ABC123', balance : 1200)
```

会被其等效的代码替换(参见附录B)：

```
def acc = new Account()           // 默认的构造器  
acc.setNumber('ABC123')         // 隐式调用setter方法  
acc.setBalance(1200)
```

12.2 复合方法

范例05和范例06演示了Account对象作为集合组成元素的相关处理方法。我们可以使用此技术为银行应用程序建模。在此模型中，从创建account对象开始，在这些账户上所做的交易都通过这家银行实现。此应用程序的结构是一个一对多的关系：即一个银行对应着多个账户。列表和映射集合可以轻易处理这种一对多的关系。由于Bank类的某些方法需要通过账户号码来标识某个特定账户，我们可以使用映射，把账户号码作为关键字，其值为Account对象。含有两个Account对象的映射如下所示：

```
['ABC123' : new Account('ABC123', 1200), 'DEF456' : new Account('DEF456', 1000)]
```

在这里，我们使用构造器语句new Account('ABC123', 1200)创建一个新的Account对象。

我们可以使用图12-1所示的类图作为模型。图12-1定义了Bank类和Account类之间的聚合关系。多样性(multiplicity)标识符“*”说明了一个Bank对象与零个或者多个Account对象相关。角色(role)账户说明了Bank以何种方式指向这些Account对象。角色名可以作为Bank类的一个属性。

在这个应用程序中，我们希望能够在银行新开一个账户，并在特定账户上使用credit和debit方法，以获取某个指定账户的金额数，并获取该银行的总资产数(所有账户的金额之和)。应用程序的解决方案如范例08所示。请注意范例中两个类的定义方法，特别是Bank类，它将account属性初始化为一个空映射，方法openAccount生成一个以账户号码为关键字、Account对象作为其值的映射。方法findAccount和方法getTotalAssets使用闭包高效地实现了其功能。

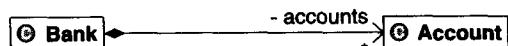


图12-1 类图

范例08 银行范例

```
class Account {  
  
    def credit(amount){  
        balance +=amount  
    }  
  
    def debit(amount){  
        if(balance >=amount)  
            balance -=amount  
    }  
  
    String toString(){  
        //redefinition  
        return "Account:${number}with balance:${balance}"  
    }  
  
//-----properties -----  
  
    def number  
    def balance  
}  
  
class Bank {  
  
    def openAccount(number,balance){  
        def acc =new Account(number :number,balance :balance)  
        accounts [number ]=acc  
    }  
  
    def creditAccount(number,amount){  
        def acc =this.findAccount(number)  
        if(acc !=null)  
            acc.credit(amount)  
    }  
  
    def debitAccount(number,amount){  
        def acc =this.findAccount(number)  
        if(acc !=null)  
            acc.debit(amount)  
    }  
  
    def getAccountBalance(number){  
        def acc =this.findAccount(number)  
        return (acc ==null)?null :acc.balance  
    }  
  
    def getTotalAssets(){  
}
```

```
def total =0
accounts.each {number,account ->total +=account.balance }
return total
}

def findAccount(number){
    def acc =accounts.find {entry ->entry.key ==number }
    return (acc ==null)?null :acc.value
}

//----properties -------

def name                      //name of bank
def accounts =[ :]           //accounts opened with the bank
{

    //open new bank
def bk =new Bank(name :'Community')

    //Open new accounts
bk.openAccount('ABC123',1200)
bk.openAccount('DEF456',1000)
bk.openAccount('GHI789',2000)

    //Perform transactions on a particular account
bk.creditAccount('ABC123',200)      //balance now 1400
bk.debitAccount('ABC123',900)        //balance now 500
bk.debitAccount('ABC123',700)        //balance remains unchanged at 500

    //Display details of this account
println "Balance for account ABC123 is:${bk.getAccountBalance('ABC123')}"

    //Calculate total bank assets
println "Total assets:${bk.getTotalAssets()}"
```

执行该程序，输出结果为：

```
Balance for account ABC123 is: 500
Total assets: 3500
```

12.3 习题

1. 请编写表示公司员工的类Employee，其中，每个员工都有员工编号、姓名以及工资三个属性。随后，请编写程序创建一个员工列表，并计算这些员工的总工资值。
2. 请编写表示二维空间上点的类Point，其中，每个点都具有x和y属性。在类的定义中请包含moveBy方法，用来将点从x和y位置移动到方法参数所指定的位置。
3. 在上一个习题Point类的基础上编写类Line，并定义其起始点和终止点。在类的定义中请包含方法

moveBy，它用来将线移动到指定的位置。它还包含isHorizontal和isVertical方法以判断线的特性。

4. 在先前编写的Point类的基础上编写类Rectangle，并定义其左上顶点的位置、宽度和高度。该类还包含moveBy、getArea和getPerimeter方法。其中，方法moveBy用来将矩形移动到指定位置，getArea和getPerimeter方法分别用来计算矩形的面积和周长。
5. 使用习题1中的Employee类编写含有一定数量员工的类Company。在Company类中定义hire(employee)、display()和getTotalSalaries()方法。方法hire将那些新员工加入到Company中。display方法输出一个包含所有员工对象的列表，而getTotalSalaries方法则用来计算公司的总工资值。
6. 完善习题5中的类Company，使得在Company中有一定数量的部门，而每个部门都有一定数量的员工。类Department有一个name属性，并且有add、display和getTotalSalaries方法。其中，add方法用来将新员工分配到相关部门，Display方法和getTotalSalaries方法分别用来显示某个部门中的所有员工和获取某个部门的总工资值。而类Company则有open(department)、hire(deptName, employee)、display()和getTotalSalaries()方法。
7. 一家新闻机构需要维护一个包含顾客姓名的列表。对于每个顾客来说，该新闻机构有一个需要投递到顾客家中的报纸列表。开发一个系统以列出每种报纸及其所需数量。

第13章 学习案例：图书馆 应用程序（对象）

图书馆应用范例首次出现在第6章中，该章演示了综合使用列表和映射来实现复杂的数据结构，以操作图书馆借阅数据库。在这些集合中，需要维护的都是一些简单的字符串。本书在第11章引入了一些用来增强系统性能的程序代码，重新实现了应用程序的功能。举例来说，程序编写了用来获取某个借阅者所借书籍和记录某个借阅者借阅书籍的方法。

本章将使用第12章所讨论的面向对象的方法，并实现与之相同的功能。在该应用程序中，使用对象分别表示图书馆、借阅者及其书名，这种做法要远胜于使用简单的字符串的表示方式。一旦使用了对象，它们就拥有更有意义的属性和行为。如应用程序的前两个版本所示，我们将使用容器来模拟对象之间的复杂关系。

13.1 需求规范

假定我们对如下描述的图书馆运转流程已经非常熟悉：

图书馆拥有一个名字属性，保存着大量的书籍，其中每本书籍都有一个书名、作者名及其唯一标识码属性。对于已经注册过的借阅者来说，每个借阅者都有其姓名和一个唯一会员号属性。每个借阅者都能借阅或者归还书籍。但是，每本书籍的借阅归还过程都必须通过图书管理员登记。图书管理员可以注册新的借阅者，增加新的书籍，能够显示整个图书馆的借阅信息，能够显示那些可供外借的书名，能够显示已经被借出的书名，还能够显示每个已经注册的借阅者的详细信息。

这些需求可以通过如下功能模块实现：

- 向借阅数据库添加一本新书
- 记录每本书的借出和归还信息
- 显示当前借阅数据库的详细信息
- 注册一个新借阅者
- 显示借阅者的详细信息

由于所要开发的图书馆系统较为复杂，因此非常适合使用迭代开发方式。前两次迭代实现的目标是：使程序模型能够尽可能真实地反映问题。如果不这样做，后续开发容易陷入困境。迭代3在应用程序中引入了一个基于文本的简单用户交互界面。为了防止陷入需求泥潭，每次迭代都有明确的实现目标。

13.2 迭代1：最初的模型

在详细需求部分提到过书籍都是从图书馆借出的，这说明这个应用程序的类图和第12章中的银行与其多个账户之间的关系非常类似。最初的类图如图13-1所示。Book类

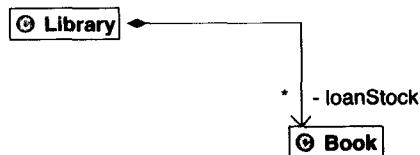


图13-1 最初的类图

表示可以从图书馆借出的书籍，是一个具体的类，它包含所有借出书籍的属性和行为，也就是说，它包含图书分类号、书名以及作者名等属性。Library类和Book类（表示借阅数据库）之间存在聚合关系。这个关系可以使用映射来实现，图书分类号作为关键字，其对应值为Book对象。

在第12章的银行范例中，我们首先创建了一些对象，然后把这些对象应用到应用程序结构中，最后通过调用各种方法以确保我们所做工作的完整性，成功地演示了这些类的基本功能。在这里，我们也采用相同的做法。程序首先将创建一个Library对象和一些Book对象，然后把它们添加到图书馆的借阅数据库中。最后，应用程序将显示借阅数据库的所有信息。这些代码如Library 01所示。

LIBRARY 01 最初的对象配置

```
class Book {  
  
    String toString() { //redefinition  
        return "Book:${catalogNumber}:${title}by:${author}"  
    }  
  
    //-----properties-----  
  
    def catalogNumber  
    def title  
    def author  
}  
  
class Library {  
  
    def addBook(bk){  
        loanStock [bk.catalogNumber ]=bk  
    }  
  
    def displayStock(){  
        println "Library:${name}"  
        println '====='  
  
        loanStock.each {catalogNumber,book ->println " ${book}" }  
    }  
  
    //-----properties-----  
  
    def name  
    def loanStock =[ :]  
}  
  
//Create a library object  
def lib = new Library(name :'Dunning')  
  
//Create some books...
```

```

def bk1 =new Book(catalogNumber :'111',title :'Groovy',author :'Ken')
def bk2 =new Book(catalogNumber :'222',title :'OOD',author :'Ken')
def bk3 =new Book(catalogNumber :'333',title :'UML',author :'John')

    //...add them to the loan stock
lib.addBook(bk1)
lib.addBook(bk2)
lib.addBook(bk3)

    //See stock
lib.displayStock()

```

请注意，在闭包中使用了两个形参（映射的关键字及其所对应的值），如

```
loanStock.each { catalogNumber, book -> println "${book.value}" }
```

尽管我们只使用一个形参，但这将使我们更加明确，loanStock引用映射而不是列表。因此把这种方法作为标准做法。

执行此Groovy脚本，结果如下所示：

```

Library: Dunning
=====
Book: 111: Groovy by: Ken
Book: 222: OOD by: Ken
Book: 333: UML by: John

```

执行结果表明我们已经建立起了正确的体系结构，并且正确地实现了两个类的行为。因此，迭代1到此结束。

13.3 迭代2：模型完善

接下来，需要在模型中引入注册的借阅者概念。在详细需求部分，我们已经说明：注册的借阅者可以通过唯一的会员号来识别。注册的借阅者能够借阅和归还图书馆的图书。这种需求可以使用图13-2所示的类图表示。Borrower对象通过Library注册，且通过它们维护所借图书的集合。

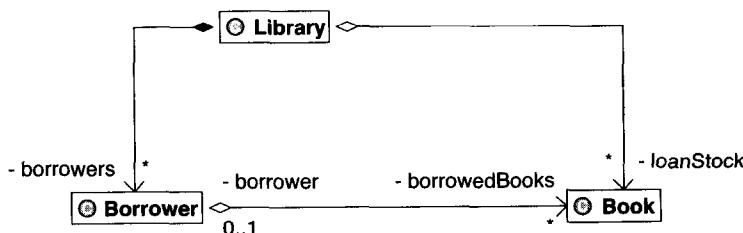


图13-2 引入借阅者概念

为实现图13-2的功能需求，需要引入Borrower类，会员号和会员名是该类的属性。Borrower类同样也有向借阅者所借图书的集合中添加或者删除一个Book对象的方法。请注意Book对象引用借阅某本书籍的Borrower对象（使用角色名borrower）的方式。如果值为空，则说明这个

Book对象没被借出；如果值不为空，则说明这个值所引用的Borrower对象就是借阅本书的借阅者。

Library类还增加了一个用来注册新的Borrower对象的方法，以及用来显示每个Borrower对象所借图书信息的方法。Library类也包含借出和归还Book对象的操作。其代码如Library 02所示。

LIBRARY 02 Borrowers

```
class Book {

    def attachBorrower(borrower){
        this.borrower =borrower
    }

    def detachBorrower(){
        borrower =null
    }

    String toString(){           //redefinition
        return "Book:${catalogNumber}:${title}by:${author}"
    }

    //-----properties-----

    def catalogNumber
    def title
    def author
    def borrower =null
}

class Borrower {

    def attachBook(bk){
        borrowedBooks [bk.catalogNumber ]=bk
        bk.attachBorrower(this)
    }

    def detachBook(bk){
        borrowedBooks.remove(bk.catalogNumber)
        bk.detachBorrower()
    }

    String toString(){
        return "Borrower:${membershipNumber};${name}"
    }

    //-----properties-----

    def membershipNumber
```

```
def name
def borrowedBooks =[ :]
}

class Library {

    def addBook(bk){
        loanStock [bk.catalogNumber]=bk
    }

    def displayStock(){
        println "\n \nlibrary:${name}"
        println '====='

        loanStock.each {catalogNumber,book ->println " ${book}" }
    }

    def displayBooksAvailableForLoan(){
        println "\n \nLibrary:${name}:Available for loan "
        println '====='

        loanStock.each {catalogNumber,book ->if(book.borrower ==null)println " ${book}" }
    }

    def displayBooksOnLoan(){
        println "\n \nLibrary:${name}:On loan "
        println '====='

        loanStock.each {catalogNumber,book ->if(book.borrower !=null)println " ${book}" }
    }

    def registerBorrower(borrower){
        borrowers [borrower.membershipNumber ]=borrower
    }

    def displayBorrowers(){
        println "\n \nLibrary:${name}:Borrower details "
        println '====='

        borrowers.each {membershipNumber,borrower ->
            println borrower
            borrower.borrowedBooks.each {catalogNumber,book ->println " ${book}" }
        }
    }

    def lendBook(catalogNumber,membershipNumber){
        def loanStockEntry =loanStock.find {entry ->entry.key ==catalogNumber }
        def borrowersEntry =borrowers.find {entry ->entry.key ==membershipNumber }
    }
}
```

```
    borrowersEntry.value.attachBook(loanStockEntry.value)
}

def returnBook(catalogNumber){
    def loanStockEntry =loanStock.find {entry ->entry.key ==catalogNumber }
    def bor =loanStockEntry.value.borrower
    bor.detachBook(loanStockEntry.value)
}

//----properties ----

def name
def loanStock =[ :]
def borrowers =[ :]
}

//Create a library object
def lib = new Library(name :'Dunning')

//Create some books...
def bk1 = new Book(catalogNumber :'111',title :'Groovy',author :'Ken')
def bk2 = new Book(catalogNumber :'222',title :'OOD',author :'Ken')
def bk3 = new Book(catalogNumber :'333',title :'UML',author :'John')

//...add them to the loan stock
lib.addBook(bk1)
lib.addBook(bk2)
lib.addBook(bk3)

//See stock
lib.displayStock()

//Now introduce some borrowers
bo1 = new Borrower(membershipNumber :'1234',name :'Jessie')
bo2 = new Borrower(membershipNumber :'5678',name :'Sally')

lib.registerBorrower(bo1)
lib.registerBorrower(bo2)

//See borrowers
lib.displayBorrowers()

//Finally,make some transactions
lib.displayBooksAvailableForLoan()

lib.lendBook('111','1234')

lib.displayBooksAvailableForLoan()
```

```
lib.displayBooksOnLoan()  
lib.displayBorrowers()  
  
lib.returnBook('111')  
  
lib.displayBooksAvailableForLoan()  
lib.displayBooksOnLoan()  
lib.displayBorrowers()
```

如上一次迭代所示，运行此脚本时，所输出的结果说明类的行为已经达到了预期目的。

```
Library:Dunning  
=====  
Book:111:Groovy by:Ken  
Book:222:OOD by:Ken  
Book:333:UML by:John
```

```
Library:Dunning :Borrower details  
=====  
Borrower:1234;Jessie  
Borrower:5678;Sally
```

```
Library:Dunning :Available for loan  
=====  
Book:111:Groovy by:Ken  
Book:222:OOD by:Ken  
Book:333:UML by:John
```

```
Library:Dunning :Available for loan  
=====  
Book:222:OOD by:Ken  
Book:333:UML by:John
```

```
Library:Dunning :On loan  
=====  
Book:111:Groovy by:Ken
```

```
Library:Dunning :Borrower details  
=====  
Borrower:1234;Jessie  
Book:111:Groovy by:Ken  
Borrower:5678;Sally
```

```
Library:Dunning :Available for loan  
=====  
Book:111:Groovy by:Ken  
Book:222:OOD by:Ken  
Book:333:UML by:John
```

```

Library:Dunning :On loan
=====
Library:Dunning :Borrower details
=====
Borrower:1234;Jessie
Borrower:5678;Sally

```

13.4 迭代3：用户界面

前面的练习都是通过程序代码硬编码方式实现的。迭代3将实现基于文本的用户交互界面，通过类似于第11章中迭代2所使用的简单菜单选项来控制应用程序的流程。由于方法的执行行为都是由用户选项决定的，因此这种菜单方式将使应用程序的运行更加灵活。

菜单可以使用某些简单的过程性代码来实现。readMenuSelection方法向用户提供应用程序菜单，接收用户的输入选项，然后将选项值返回给调用函数。while循环语句将确保菜单可以循环使用，直到用户需要退出程序为止。通过一连串的if…else语句判断用户的输入选项，然后执行所需的功能。

虽然这个用户交互界面实现起来不是特别困难，但在后面的程序中，我们可能会需要为此应用程序开发图形化用户交互界面或者web交互界面。如果仔细思考迭代2所开发的模型，就会发现其设计上的缺陷非常明显，也就是说，Library类的很多方法都是直接将结果输出到控制台上。不幸的是，用户交互界面的改变需要对此做很大改动，并且其他的类也可能同样需要改动。

为了尽量避免大的改动，没有给类Book、Borrower和Library提供输入输出功能，并将它们称作域模型（domain model）。这种方法可以确保表示这些域模型的类，不需要做任何修改就能适应用户交互界面的改变。

一个实用的方法是让某个对象专门负责和域模型的交互，并处理应用程序的输入输出事件。出于此想法，我们在程序中引入了Action类，它有一组分别用来处理应用程序中每个用户用例的方法。Action类和域模型类的关系如图13-3所示。

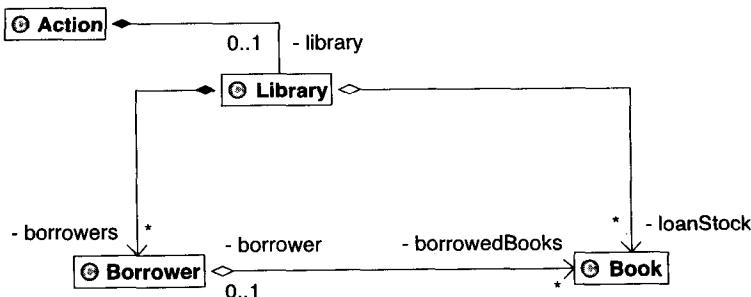


图13-3 集成Action类

程序的最终代码如Library 03所示。其中，主应用程序代码负责显示用户菜单，以及处理用户选项。用户选项随后被转发给Action类的方法。举例来说，Action类的displayStock方法将处理所有来自Library域模型的输出数据。同样，Action类的registerBorrower方法用来登记借阅

者的信息，构造一个Borrower对象，然后把Borrower对象注册到Library。

LIBRARY 03 基于文本的菜单

```
import console.*

//The Book and Borrower classes are unchanged from the previous iteration

class Library {

    def addBook(bk){
        loanStock [bk.catalogNumber ]=bk
    }

    def registerBorrower(borrower){
        borrowers [borrower.membershipNumber ]=borrower
    }

    def lendBook(catalogNumber,membershipNumber){
        def loanStockEntry =loanStock.find {entry ->entry.key ==catalogNumber }
        def borrowersEntry =borrowers.find {entry ->entry.key ==membershipNumber }
        borrowersEntry.value.attachBook(loanStockEntry.value)
    }

    def returnBook(catalogNumber){
        def loanStockEntry =loanStock.find {entry ->entry.key ==catalogNumber }
        def bor =loanStockEntry.value.borrower
        bor.detachBook(loanStockEntry.value)
    }

//-----properties -----
    def name
    def loanStock =[ :]
    def borrowers =[ :]
}

class Action {

    def addBook(){
        print('\nEnter book catalog number:')
        def catalogNumber =Console.readLine()
        print('Enter book title:')
        def title =Console.readLine()
        print('Enter book author:')
        def author =Console.readLine()

        def bk = new Book(catalogNumber :catalogNumber,title :title,author :author)
    }
}
```

```
    library.addBook(bk)
}

def displayStock(){
    println "\n \nLibrary:${library.name}"
    println '====='

    library.loanStock.each {catalogNumber,book ->println " ${book}" }
}

def displayBooksAvailableForLoan(){
    println "\n \nLibrary:${library.name}:Available for loan "
    println '====='

    library.loanStock.each {catalogNumber,book ->if(book.borrower ==null)println "${book}" }
}

def displayBooksOnLoan(){
    println "\n \nLibrary:${library.name}:On loan "
    println '====='

    library.loanStock.each {catalogNumber,book ->if(book.borrower !=null)println " ${book}" }
}

def registerBorrower(){
    print('\nEnter borrower membership number:')
    def membershipNumber =Console.readLine()
    print('Enter borrower name:')
    def name =Console.readLine()

    def bor =new Borrower(membershipNumber :membershipNumber,name :name)

    library.registerBorrower(bor)
}

def displayBorrowers(){
    println "\n \nLibrary:${library.name}:Borrower details "
    println '====='

    library.borrowers.each {membershipNumber,borrower ->
        println borrower
        borrower.borrowedBooks.each {catalogNumber,book ->println " ${book}" }
    }
}

def lendBook(){
    print('\nEnter book catalog number:')
```

```
def catalogNumber =Console.readLine()
print('Enter borrower membership number:')
def membershipNumber =Console.readLine()

    library.lendBook(catalogNumber,membershipNumber)
}

def returnBook(){
    print('\nEnter book catalog number:')
    def catalogNumber =Console.readLine()

        library.returnBook(catalogNumber)
}

//-----properties -----
def library
}

def readMenuSelection(){
    println()
    println('0:Quit')
    println('1:Add new book')
    println('2:Display stock')
    println('3:Display books available for loan')
    println('4:Display books on loan')
    println('5:Register new borrower')
    println('6:Display borrowers')
    println('7:Lend one book')
    println('8:Return one book')

    print('\n \tEnter choice>>>')
    return Console.readString()
}

//make the Action object
def action = new Action(library :new Library(name :'Dunning'))

//make first selection
def choice = readMenuSelection()
while(choice !='0'){

    if(choice =='1'){                      //Add new book
        action.addBook()
    }else if(choice =='2'){                  //Display stock
        action.displayStock()
    }else if(choice =='3'){                  //Display books available for loan
        action.displayBooksAvailableForLoan()
    }
}
```

```

        }else if(choice =='4'){           //Display books on loan
            action.displayBooksOnLoan()
        }else if(choice =='5'){
            action.registerBorrower()
        }else if(choice =='6'){
            action.displayBorrowers()
        }else if(choice =='7'){
            action.lendBook()
        }else if(choice =='8'){
            action.returnBook()
        }else {
            println("Unknown selection ")
        }

        //next selection
        choice =readMenuSelection()
    }
    println('System closing')
}

```

最后，当然要测试程序是否实现了如前所述的功能需求，从菜单中进行选择，并执行与之相应硬编码指令，可以通过比较输出结果的方式进行测试。比如，用户的输入用黑斜体表示，其结果为：

```

0:Quit
1:Add new book
2:Display stock
3:Display books available for loan
4:Display books on loan
5:Register new borrower
6:Display borrowers
7:Lend one book
8:Return one book

```

Enter choice>>>**1**

Enter book catalog number:**111**
 Enter book title:**Groovy**
 Enter book author:**Ken**

//...
 //Present the menu to the user

Enter choice>>>**2**

Library:Dunning
 ======
 Book:333:UML by:John
 Book:111:Groovy by:Ken

```
Book:222:OOD by:Ken  
  
//...  
//Present the menu to the user  
  
Enter choice>>>6  
  
Library:Dunning :Borrower details  
=====  
Borrower:1234;Jessie  
Book:111:Groovy by:Ken  
Borrower:5678;Sally  
  
//...  
//Present the menu to the user  
  
Enter choice>>>0  
  
System closing
```

由于得到了相应的输出结果，迭代3到此结束。

13.5 习题

1. 在Library 01所示的第一次迭代中，Library通过addBook(bk)方法增加了一个新书对象，其中参数bk表示应用程序客户端所创建的Book对象。为此方法引入更多的描述书籍具体属性的参数，如addBook(catalogNumber, title, author)方法所示。
2. Library 01所示的应用程序脚本创建了一个Library对象和一些Book对象，并将这些Book对象添加到Library借阅数据库中，然后再显示借阅数据库中信息。请说明在创建Library对象后立即调用displayStock方法的原因。
3. 迭代3实现与用户交互的菜单、判断用户输入，以及执行选项所对应操作的代码是可重复使用的。为了与第11章中11.3节中的讨论一致，请使用Groovy闭包生成一个lookup表替代这部分代码。
4. 软件开发者常常发现他们所开发的系统具有某些相似性，尽管许多细节部分可能不同，但整个设计和实现过程通常都非常相似。在我们的学习案例中，一个图书馆会有许多书籍与之对应。同样，一个医生也对应着许多病人，或者一所大学对应着许多学生。在很多时候，这种设计模式都是基本相同的。

以本章的学习案例为指南，思考如下几个问题：

- 某个汽车租赁机构拥有数台小汽车。每台小汽车都有唯一牌照号、车型名和注册年份属性，这些小汽车只能租给那些在公司注册过的客户，每个客户都以唯一的客户号和姓名表示。请编写程序使公司所有者能够记录那些已经出租给客户的小汽车信息。
- 一家音像店有大量的故事片供出租。每个故事片都用唯一片名，每个客户都用唯一的注册号表示。请编写程序记录已经租出的故事片及其客户的信息。
- 某家医院有许多医生和病人。每个医生和病人都用姓名和唯一编号表示，医生能够照顾许多病人，但病人却只能有一个医生。请开发一个病人监护系统，以记录病人和照顾他的特定医生的信息。

- 一所大学有许多学生，每个学生都用唯一的入学编号、姓名及学科表示。请编写程序记录每个学生信息，并显示所有选修指定课程的所有学生信息。

请基于以上一个或者所有范例，编写设计及其实现方案。

5. 请开发一个酒店管理软件，酒店管理的主要特征如下：

- 酒店总共有三层，分别用数字1、2和3表示，每层的房间数各不相同
- 不是每层都有相同数量的房间
- 每个房间都有一个房间号，如，201表示二楼的第一个房间
- 每个房间都有人数限制

酒店员工能够获取每个楼层每个房间的具体使用信息，为了简单起见，软件只需要实现以下功能：

- 显示所有楼层的所有房间信息
- 显示指定楼层的所有房间信息
- 显示指定楼层的指定房间信息

此外，还需要增加一个房间废弃操作：

- 删除指定楼层的指定房间信息

6. 某家软件开发机构有很多程序员，每个程序员都精通某种语言，如Groovy、C++和Java。所有程序员的基本月工资都在1000英镑左右。但是，程序员和程序员之间的总工资数可能会大不相同。

根据规定，公司将给精通Groovy的程序员每个月多发基本工资10%的奖金。但是，程序员也可以改变他所精通的语言，其薪水将也会相应地增加，如，C++程序员在经过一段实习期后就能变一个Groovy程序员。

新增一个程序员后，公司会为他指定一个经验丰富的程序员指导他的工作，当然这两个员工都需要精通相同的语言。这种做法的目的是使公司的新员工能够方便向经验丰富的老员工请教。对老员工来说，这是额外的工作，因此公司会给老员工发5%的奖励工资。当新员工不再需要指导后，其指导者的薪水也会随之发生变化。

为了管理方便，每个员工都使用姓名和唯一薪水账号表示。

请编写一个公司管理软件。为了简单起见，软件只需要生成每个程序员的详细报告以及为公司生成全部的月工资清单。

报告应该显示每个程序员的如下信息：

- 薪水账号及其姓名
- 程序员所精通的语言及其当前的月工资
- 任意程序员的指导者和被指导者详细信息

第14章 继承

本章将介绍两个类之间可能存在的继承关系。继承 (inheritance) 广泛用在面向对象应用程序中，使设计和程序具有面向对象的重要特征。

继承（也称为特化， specialization）是使用已经定义的类派生新类的一种方法。派生出来的类通常被称作派生类（derived classes），它能继承基类（base class）的属性和方法。在本书中，术语“父类”（parent class）和“子类”（child class）与“超类”（superclass）和“子类”（subclass）一起使用，意思相同。继承的目标就是在很少或者不修改代码的情况下，重复使用已经存在的代码。

继承同样也可以称作泛化(generalization)。例如，账户就是活期账户（经常账户）和有息存款（储蓄存款账户）的泛化。常说的账户实际上就是活期账户和有息存款等账户的一个抽象概念。反过来说，由于活期账户也是账户，即，一个活期账户也就是一个账户，它们继承了所有账户的公有属性，如账户号和账户余额等属性。

14.1 继承

假定我们要为银行的客户新开几个不同的活期账户，每个活期账户都分配一个唯一的账号，也有账户余额和信用卡透支额度等属性。综合前两章所学的知识，可以通过范例01所示的代码实现CurrentAccount类。

范例01 CurrentAccount类

```
class CurrentAccount {  
  
    String toString(){  
        return "Current Account:${number};balance:${balance};overdraft :${overdraftLimit}"  
    }  
  
    //----properties -----  
  
    def number  
    def balance  
    def overdraftLimit  
  
    //populate a list with the instances  
    def accounts =[new CurrentAccount(number :'AAA111',balance :1000,overdraftLimit :400),  
                 new CurrentAccount(number :'BBB222',balance :2500,overdraftLimit :800)]
```

```
//now display each
accounts.each {acc ->
    println acc           //automatically call toString
}
```

运行此程序，我们可以发现其输出结果和预期结果相同。其中，列表变量account中的两个Account对象使用toString方法输出值。

```
Current Account: AAA111; balance: 1000; overdraft : 400
Current Account: BBB222; balance: 2500; overdraft : 800
```

尽管类CurrentAccount本身并没有任何错误，但其代码有很多可改进之处。举例来说，银行同样也为客户提供有息存款业务。如果程序中的银行需要建立有息存款业务的账户，则应该实现表示此类账户的类DepositAccount。类DepositAccount同样也有唯一账号和账户余额属性，但是它却并不需要透支额度属性。无论如何，DepositAccount账户也应该有利息。这两个类型的账户共享其公有属性，但每个账户类型都有额外的属性。

我们可以把CurrentAccount（经常账户）和DepositAccount（储蓄存款账户）当作一种特殊类型的账户。Account类拥有CurrentAccount类和DepositAccount类都具有的公有属性，也就是账户号和余额属性。这样，CurrentAccount类和Account类就通过继承建立了联系，DepositAccount类同样也通过继承和Account类建立了联系。其中，Account类通常被称作超类，而CurrentAccount类（和DepositAccount类）则被称作子类。

图14-1所示的类图描述了这些类之间的关系。继承关系（有向箭头表示）使子类CurrentAccount和DepositAccount与超类Account建立了联系。

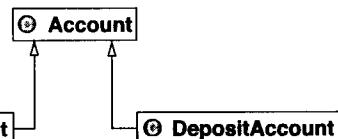


图14-1 继承

Groovy保留字extends说明某个类是从另一个类继承过来的，如范例02中代码所示。下面我们将引入DepositAccount类。

范例02 类继承

```
class Account {

    String toString(){
        return "${number};${balance}"           //redefinition
    }

    //----properties -------

    def number
    def balance
}

class CurrentAccount extends Account {

    String toString(){

    }
}
```

```
    return 'Current Account:' +super.toString() +";${overdraftLimit}"  
}  
  
//-----properties -----  
  
def overdraftLimit  
{  
  
    //populate a list with the instances  
def accounts = [new Account(number : 'AAA111',balance :1000),  
               new CurrentAccount(number : 'BBB222',balance :2000,overdraftLimit :400),  
               new CurrentAccount(number : 'CCC333',balance :3000,overdraftLimit :800)]  
    //now display each  
accounts.each {acc ->  
    println acc  
    //automatically call toString  
}
```

执行此程序其输出结果为

```
AAA111; 1000  
Current Account: BBB222; 2000; 400  
Current Account: CCC333; 3000; 800
```

在这里，我们可以发现第一行的输出和其他两行的输出结果形式不一致。这是因为，accounts变量中的第一个对象是Account对象，而其他两个则是CurrentAccount对象。由于列表中的第一个对象是Account对象，在Account类中实现的toString方法负责显示输出。toString方法也会把相同的消息发送给另外两个CurrentAccount对象。但是，在CurrentAccount类中，方法toString已经被重新定义，这就是出现最后两行输出结果的原因。

CurrentAccount类现在仅拥有特殊属性。同样，Account类也仅仅拥有与普通账户相关的属性。这样，CurrentAccount类就变得非常容易开发。认识到Account类在此应用程序，或者其他的应用程序中能够重复使用是非常重要的。本书稍后将用到的DepositAccount类就是从它继承过来的。

请注意，子类CurrentAccount已经重定义了toString方法。这是因为，它需要改进超类Account中toString方法输出的结果。因此，类CurrentAccount中定义的方法通过表达式super.toString()调用了定义在超类中的方法。关键字Super用来调用超类中方法，如果没有该关键字，类CurrentAccount中的toString方法就将以递归方式调用它自己（参见附录G）。

14.2 继承方法

在Groovy中，子类可以继承超类中定义的所有特征。这就意味着类CurrentAccount（参考前面的说明）仅需要定义它自己所需要的方法和属性，在这里，类CurrentAccount有一个额外的overdraflimit属性（我们将简要的介绍toString方法）。在一些更为复杂的范例中，这将会极大地节省开发成本。范例03的代码如下所示。

范例03 特征的继承

```
class Account {
```

```

String toString() { //redefinition
    return "${number};${balance}"
}

//----properties ----

def number
def balance
}

class CurrentAccount extends Account {

    String toString(){
        return 'Current Account:' +super.toString() +";${overdraftLimit}"
    }

//----properties ----

def overdraftLimit
}

//populate a list with the instances
def accounts =[new Account(number :'AAA111',balance :1000),
    new CurrentAccount(number :'BBB222',balance :2000,overdraftLimit :400),
    new CurrentAccount(number :'CCC333',balance :3000,overdraftLimit :800)]

//now display each
accounts.each {acc ->
    println acc //automatically call toString
}

def ca =new CurrentAccount(number :'DDD444',balance :4000,overdraftLimit :1200)

//use methods and inherited methods
println "Overdraft:${ca.overdraftLimit}"
println "Number:${ca.number}"

def ac =new Account(number :'EEE555',balance :1234)

println "Number:${ac.number}" //OK
//println "Overdraft:${ac.overdraftLimit}" //ERROR:no such property

```

运行此程序的输出结果如下所示。输出结果的前三行代码和前一个范例的输出结果相同，接下来一行信息是对象ca的信用额度值。由于ca是类CurrentAccount的对象，因此getOverdraftLimit消息就定义在CurrentAccount类中。第五行是发送消息getNumber给同一个CurrentAccount对象所返回的结果。由于该类并没有定义此方法，系统将执行通过继承得到的超类Account的方法。

```
AAA111;1000
Current Account:BBB222;2000;400
Current Account:CCC333;3000;800
Overdraft:1200
Number:DDD444
Number:EEE555
```

请注意代码清单末尾附近的代码行，当创建CurrentAccount对象ca后，它的overdraftLimit属性和number属性随即将被访问。overdraftLimit属性定义在CurrentAccount类中，而number属性却是从Account类继承过来的。代码中余下的三行显示了Account对象访问number属性（定义在Account类中）的方法，但是在此程序中，Account对象并不能引用overdraftLimit，这是因为该类并没有定义此属性。

14.3 方法重定义

在Groovy中，子类可以继承超类的所有属性和方法。这就意味着在CurrentAccount类中，如果没有定义toString方法，那么CurrentAccount类就将继承其超类Account所定义的toString方法，该方法可被所有的CurrentAccount对象调用。在Groovy中，子类可以重新定义所继承的方法，以实现不同的行为。对于CurrentAccount类所需的toString方法来说，一个显而易见的策略是利用超类Account中的toString方法，并增加相应的逻辑功能，以实现用户所想要的功能。在范例03中，类CurrentAccount中toString方法如下所示：

```
String toString() {
    return 'Current Account: ' + super.toString() + " by: ${overdraftLimit}"
}
```

请再次注意保留关键字super的用法。在这里，它用来调用超类中的toString方法，返回值作为当前方法输出的一部分。

14.4 多态性

面向对象系统的一个重要特性是多态性 (polymorphic effect)。在面向对象系统中，一条发往某个类对象的消息可以像平常一样接收。但是，该类的派生类对象同样也能够接收到此消息。如果这两类对象都有自身定义的处理消息的方法，那么就可能会出现不同的结果。使用多态性的确会使系统看上去更为简单，但同样也会使执行模型变得更加复杂。

范例03演示了多态性的用法，其部分代码如下：

```
def accounts = [new Account(number : 'AAA111', balance : 1000),
               new CurrentAccount(number : 'BBB222', balance : 2000, overdraftLimit : 400),
               new CurrentAccount(number : 'CCC333', balance : 3000, overdraftLimit : 800)]

// now display each
accounts.each { acc ->
    println acc                                // automatically call toString
}
```

其输出结果为：

```
AAA111; 1000
Current Account: BBB222; 2000; 400
Current Account: CCC333; 3000; 800
```

`print`语句向每个`acc`对象都发送了`toString`消息（隐含地）。最先接收到消息的是`Account`对象，其输出结果如第一行所示。当然，它的输出结果是执行`Account`类中所定义的`toString`方法得来的。输出的另外两行和第一行输出结果不同，即使它们所接收到的消息相同。这是由于接收消息的是`CurrentAccount`对象，而在`CurrentAccount`子类中已经重新定义了`toString`方法。

范例04展示了多态性的效果。该应用程序使用的Bank模型如类图14-2所示。

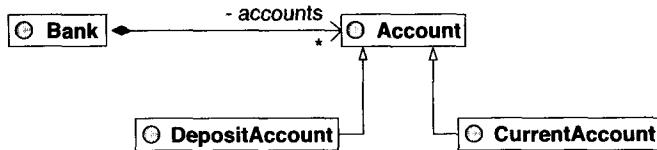


图14-2 Bank应用程序

Bank账户由`CurrentAccount`账户或者`DepositAccount`账户组成。Bank提供新账户开户、获得Bank的报告及其账户信息的方法。

范例04 Bank范例

```

class Account {

    String toString(){                               //redefinition
        return "${number};${balance}";
    }

    //----properties ----

    def number
    def balance
}

class CurrentAccount extends Account {

    String toString(){
        return 'Current Account:' +super.toString() +";${overdraftLimit}"
    }

    //----properties ----

    def overdraftLimit
}

class DepositAccount extends Account {
}
  
```

```
String toString(){
    return 'Deposit Account:' +super.toString() +";${interestRate}"
}

//-----properties ----

def interestRate
}

class Bank {

    def openAccount(account){
        accounts [account.number] =account
    }

//-----properties ----

    def name
    def accounts =[ :]
}

def displayBank(bk){
    println "Bank:${bk.name}"
    println '====='

    bk.accounts.each {number,account ->println " ${account}" }
}

//create a new Bank object
def bk = new Bank(name :'Barclay')

//create some accounts
def ca1 = new CurrentAccount(number :'AAA111',balance :2000,overdraftLimit :400)
def ca2 = new CurrentAccount(number :'BBB222',balance :3000,overdraftLimit :800)
def da1 = new DepositAccount(number :'CCC333',balance :4000,interestRate :4)

//add them to the bank
bk.openAccount(ca1)
bk.openAccount(ca2)
bk.openAccount(da1)

//now display everything
displayBank(bk)
```

该程序的输出结果为：

Bank: Barclay

```
=====
Deposit Account: CCC333; 4000; 4
Current Account: BBB222; 3000; 800
Current Account: AAA111; 2000; 400
```

该范例演示了替代原理 (principle of substitution)。这表明，在应用程序代码中，希望超类对象执行的地方，子类的对象也同样可以执行。Bank类中的openAccount方法有一个用来表示某种Account对象的参数。应用程序将此方法发送给Bank对象bk，同时也发送给CurrentAccount对象和DepositAccount对象。根据继承原理，在调用基类的Account对象时，只有该类所拥有的方法才能被当作参数使用，因此上面的方法是可行的。由于子类对象能够自动地继承这些行为，这样也能确保正确的行为。

最后需要指出，Bank类是一个域模型类，由于前面所述的原因，我们将使该类不包含任何显示方法。

14.5 抽象类

定义一个只用来创建其他类的基础类的做法，通常来说是非常实用的。我们并不需要为这种类创建实例，因为它只是其继承类共享的公有特性的一种方法。这种类型的类通常被称作抽象类。

在范例04所示的bank应用程序中，我们假定没有Account实例。也就是说，有CurrentAccounts实例或者DepositAccounts实例，但没有Account实例。我们还希望所有的Bank账户都共享公有特征，如number和balance。因此，我们可以把Account类定义成一个不需要改动的抽象类。在图14-3中，Account类被标识为“A”，就是为了强调它是一个抽象类。

抽象类的定义必须使用关键字abstract，如范例05中类Account所示。如果不这么做，类的余下部分功能就将和其子类完全一样。我们必须注意的一个关键点是，Groovy支持抽象概念，但它并不会去尝试创建抽象类的实例。

范例05 抽象类不能被实例化

```
abstract class Account {

    String toString() { //redefinition
        return "${number};${balance}"
    }

    //----properties -----

    def number
    def balance
}
```

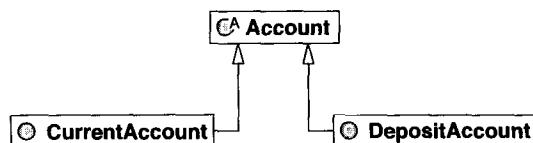


图14-3 抽象类

```
class CurrentAccount extends Account {...}

class DepositAccount extends Account {...}

class Bank {...}

def displayBank(bk){...}

    //create a new Bank object
def bk = new Bank(name : 'Barclay')

    //create some accounts
def ca1 = new CurrentAccount(number : 'AAA111',balance : 2000,overdraftLimit : 400)
def ca2 = new CurrentAccount(number : 'BBB222',balance : 3000,overdraftLimit : 800)
def da1 = new DepositAccount(number : 'CCC333',balance : 4000,interestRate : 4)

    //add them to the bank
bk.openAccount(ca1)
bk.openAccount(ca2)
bk.openAccount(da1)

    //now display everything
displayBank(bk)

//acc =new Account(number : 'DDD444',balance : 1234)      //ERROR
```

该程序的输出结果和前一范例相同。范例05中的最后一行代码也证实：抽象类Account不允许被实例化。

抽象类的常见做法是提供延迟实现的方法（deferred method），也就是说抽象类所定义的方法没有方法体代码。出现这种情况的原因是，由于类过于抽象，而不清楚该方法究竟需要执行什么操作。抽象类的延迟实现方法推断：如果这些抽象类的子类表示具体的类，并且需要创建子类的实例，就会提供自己的方法实现。实际上，延迟实现方法强加给子类的一个协议是：必须考虑是否需要一个具体的类。在Groovy中，延迟实现的方法通常被称为抽象方法（abstract method），并且使用关键字abstract来定义。范例06中的抽象类Account包含一个名为isOverdrawn的抽象方法，其定义如下。

```
//class Account
def abstract isOverdrawn()

范例06 抽象方法
abstract class Account {

    String toString()           //redefinition
        return "${number};${balance}"
    }

    def abstract isOverdrawn()      //deferred method

//----properties -----
```

```
def number
def balance
}

class CurrentAccount extends Account {

    String toString(){
        return 'Current Account:' +super.toString() +";${overdraftLimit}"
    }

    def isOverdrawn(){           //redefinition
        return balance <-overdraftLimit
    }

//-----properties -----
    def overdraftLimit
}

class DepositAccount extends Account {

    String toString(){
        return 'Deposit Account:' +super.toString() +";${interestRate}"
    }

    def isOverdrawn(){           //redefinition
        return balance <0
    }

//-----properties -----
    def interestRate
}

class Bank {

    def openAccount(account){
        accounts [account.number] = account
    }

//-----properties -----
    def name
    def accounts =[ :]
}

def displayBank(bk){
```

```
println "Bank:${bk.name}"
println '====='

bk.accounts.each {number,account ->println " ${account}"}

//create a new Bank object
def bk = new Bank(name :'Barclay')

//create some accounts
def ca1 = new CurrentAccount(number :'AAA111',balance :2000,overdraftLimit :400)
def ca2 = new CurrentAccount(number :'BBB222',balance :3000,overdraftLimit :800)
def da1 = new DepositAccount(number :'CCC333',balance :4000,interestRate :4)

//add them to the bank
bk.openAccount(ca1)
bk.openAccount(ca2)
bk.openAccount(da1)

//now display everything
displayBank(bk)

//check status of some accounts
println "Current account:${ca1.number};overdrawn?:${ca1.isOverdrawn()}"
println "Deposit account:${da1.number};overdrawn?:${da1.isOverdrawn()}"
```

该程序的输出结果和前两个程序的输出结果相同。请注意CurrentAccounts类和DepositAccounts类中定义的isOverdrawn方法。在CurrentAccounts类中，该方法用来检查账户余额与透支额度的关系；而在DepositAccounts类中，则用来检查账户余额是否为负数。

14.6 接口类

抽象类可能没有定义任何一个方法，这些方法都需要子类根据自身需求实现。这样的类通常被称作接口类（interface class）。由于接口类并没有真正定义方法，因而接口仅仅表示接口类所有行为的规范。接口非常重要，它作为一个或者多个子类必须遵守的协议，要求子类必须提供所有方法的定义。

在Groovy中，接口类的定义必须使用interface关键字。虽然接口类和没有定义方法的抽象类较为相似，但认识这两种类之间的差异是非常重要的。实现接口的类需要为其以后的操作提供方法实现，并不需要处于相同的类层次。虽然它们可能会实现其他的方法，并拥有不同的超类，但如果它们通过接口实现这些操作，就可以取代接口的功能。这使得接口的功能非常强大，并且使用起来也非常方便。和抽象类相比，它能给设计者提供更大的灵活性。

请思考银行及其客户之间存在的关系。首先我们假定使用者能够获取任意账户的透支信息。很明显，一个account对象所属的类必须能够实现isOverdrawn操作。然而，这并不需要让每个类都处于相同的继承层次。对设计者来说，这也是区分它们的一个重要标识。Bank所要做的所

有事情就是给它的每个account对象发送isOverdrawn消息。它也可能会发送其他的消息，但对于由Bank开户的account来说，仅需要isOverdrawn操作。

这种情形可以使用如图14-4所示的Groovy接口建模。连接AccountAB和AccountIF的虚箭头表示抽象类AccountAB实现了接口类AccountIF。根据UML规则，我们使用《interface》和/或者I来修饰AccountIF类。

范例07实现了此模型，其中AccountIF为接口类，它定义了一个或者多个抽象方法，但并不定义具体

的实现方法。抽象类AccountAB则实现了类AccountIF。请注意，类AccountAB为isOverdrawn方法提供了简单的实现。我们必须在CurrentAccount类中显式地重定义isOverdrawn方法。

范例07 接口类

```
interface AccountIF {
    def abstract isOverdrawn() //deferred method
}

abstract class AccountAB implements AccountIF {
    String toString(){ //redefinition
        return "${number};${balance}"
    }

    def isOverdrawn(){ //redefinition
        return balance < 0
    }
    //----properties -----
    def number
    def balance
}

class CurrentAccount extends AccountAB {
    String toString(){
        return 'Current Account:' +super.toString() +";${overdraftLimit}"
    }

    def isOverdrawn(){ //redefinition
        return balance < -overdraftLimit
    }
}
```

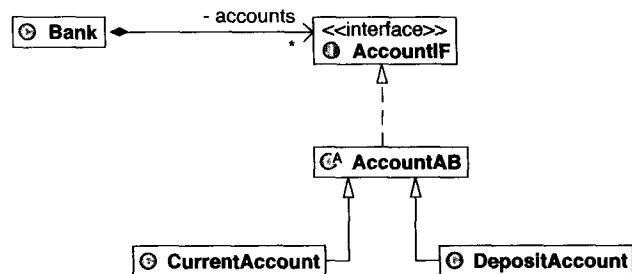


图14-4 接口类

```
        return balance <-overdraftLimit
    }
//-----properties -----
def overdraftLimit
}

class DepositAccount extends AccountAB {

    String toString(){
        return 'Deposit Account:' +super.toString() +";${interestRate}"
    }

//-----properties -----
def interestRate
}

class Bank {

    def openAccount(account){
        accounts [account.number] =account
    }

//-----properties -----
def name
def accounts =[ :]
}

def displayBank(bk){
    println "Bank:${bk.name}"
    println '====='
    bk.accounts.each {number,account ->println " ${account}"}
}

//create a new Bank object
def bk = new Bank(name :'Barclay')

//create some accounts
def ca1 = new CurrentAccount(number :'AAA111',balance :2000,overdraftLimit :400)
def ca2 = new CurrentAccount(number :'BBB222',balance :3000,overdraftLimit :800)
def da1 = new DepositAccount(number :'CCC333',balance :4000,interestRate :4)

//add them to the bank
bk.openAccount(ca1)
```

```

bk.openAccount(ca2)
bk.openAccount(da1)
//now display everything
displayBank(bk)

//check status of some accounts
println "Current account:${ca1.number};overdrawn?:${ca1.isOverdrawn()}"
println "Deposit account:${da1.number};overdrawn?:${da1.isOverdrawn()}"

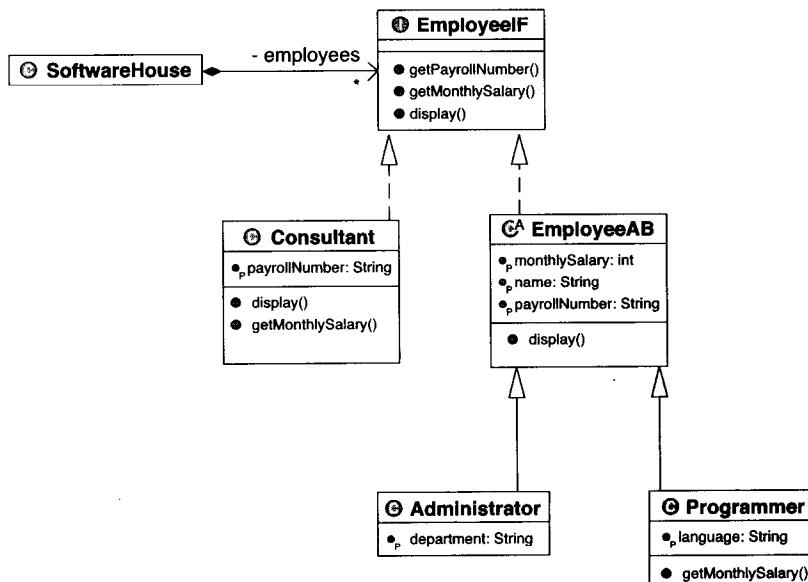
```

程序的输出结果和前一个程序的输出结果相同。

附录B-5简要讨论了在Groovy中接口的价值。由于Groovy支持动态类型，接口概念的价值并不大。在Groovy中，多态性只不过是匹配方法名和原型的一种方式。但是和Java一样，接口则提供了所有具体子类必须遵循的协议。更进一步来说，如果有必要的话，包含接口的Groovy代码可以很容易地转换成Java代码。

14.7 习题

1. 在范例07中，实现类Account的display方法使其子类CurrentAccount和子类DepositAccount不需重新实现此方法。
2. 以习题1为原型，开发类StudentAB、类Undergraduate和类Postgraduate。其中，类StudentAB是一个抽象类，它表示与学生相关的一般属性，也就是他们的姓名及其学籍号。另外两个类都是具体的、特殊的StudentAB类。类Undergraduate拥有专业名称和学习年份信息，而Postgraduate则拥有课题名称信息。建立研究生和大学生列表，并输出他们的具体信息。
3. 下面的类图描述了被一家软件公司雇佣的员工的个人信息。其中顾问是临时员工，其月工资为500英镑，精通Groovy的程序员工资比基本工资高10%。



4. 假定需要开发一套酒店管理软件。其中，酒店的主要特征如下：

- 酒店共有1、2和3层，每层有5个房间。
- 大部分房间都是标准间，但也有一些会议室。
- 会议室可能会有与之相关的学习室。
- 学习室是由卧室简单改装而成。
- 并非所有的房间都有与之相关的会议室。
- 并非所有楼层的房间组成结构都相同。
- 每个房间都有房间号，如201表示2楼的第一个房间。
- 每个房间都有人数限制，也就是能够使用该房间的人数有限。
- 会议室同样也有一个名称。

酒店员工能够获得每个楼层每个房间的具体使用信息，为了简单起见，软件只需要实现以下功能：

- 显示所有楼层的所有房间信息
- 显示指定楼层的所有房间信息
- 显示指定楼层的指定房间信息

还需要增加一个房间废弃操作：

- 删除指定楼层的指定房间信息

如果该房间是卧室，则需要显示其房间号和最大使用人数信息。如果是一个会议室，则必须提供与之相关的每个学习室的名称和房间号。学习室则显示和卧室相同的信息。

5. 在下面的代码中，类Point表示一个二维空间上的点。根据接口类QuadrilateralIF完善它所确立的类层次关系。Rectangle对象的位置通过左上角位置、宽度和高度表示。

```
class Point {  
  
    def moveBy(deltaX,deltaY){  
        x +=deltaX  
        y +=deltaY  
    }  
  
    //----properties -----  
  
    def x  
    def y  
}  
  
interface QuadrilateralIF {  
    def abstract getArea()  
    def abstract getPerimeter()  
    def abstract moveBy(deltaX,deltaY)  
}  
  
class Rectangle implements QuadrilateralIF {  
    //----properties -----
```

```
def upperLeft
def width
def height
}

class Square extends Rectangle {
}

def rect =new Rectangle(upperLeft :new Point(x :0,y :10),width :10,height :5)
rect.moveBy(2,4)

println "rect:${rect.getArea()},${rect.getPerimeter()}"           //output:50,30

def sq =new Square(upperLeft :new Point(x :0,y :10),width :10,height :10)

println "sq:${sq.getArea()},${sq.getPerimeter()}"           //output:100,40
```

6. 编写类SalariedEmployee和HourlyEmployee。一个有薪水的员工，他的薪水是月付的。一个小时工则有固定的工资率及每月工作的小时数信息。为这两个子类实现computeMonthlyPay方法，然后显示整个机构的月工资清单，总值为3300英镑。

```
interface EmployeeIF {
    def abstract getName()
    def abstract getPayrollNumber()
}

abstract class EmployeeAB implements EmployeeIF {
    def abstract computeMonthlyPay()

//----properties -----
    def name
    def payrollNumber
}

class SalariedEmployee extends EmployeeAB {
//----properties -----
    def salary
}

class HourlyEmployee extends EmployeeAB {
//----properties -----
    def payRate          //per hour
```

```
def hoursWorked      //per month
}

class Company {
    def hire(employee){
        employees [employee.payrollNumber] = employee
    }

    def getMonthlySalaryBill(){
        def total =0
        employees.each {number,employee ->
            total +=employee.computeMonthlyPay()
        }
        return total
    }

//-----properties -----
    def name
    def employees =[ :]
}

def co =new Company(name :'Napier')

def sel =new SalariedEmployee(name :'Ken',payrollNumber :1111,salary :12000)
def se2 =new SalariedEmployee(name :'John',payrollNumber :2222,salary :18000)

def hel = new HourlyEmployee(name :'Sally',payrollNumber :3333,
                             payRate :5,hoursWorked :160)

co.hire(sel)
co.hire(se2)
co.hire(hel)

println "Total monthly bill:${co.getMonthlySalaryBill()}"      //output:3300
```

第15章 单元测试 (JUNIT)

本章将介绍JUnit测试框架在Groovy环境中的使用方法。为了演示GroovyTestCase类实现单元测试的整个过程，我们使用了第13章的学习案例。本章随后将展示如何把多个GroovyTestCase类组合为一个GroovyTestSuite。最后，我们思考了单元测试在迭代、递增的应用程序开发过程中的作用。在整个讨论过程中，我们会一直强调Groovy单元测试非常简单方便。

15.1 单元测试

在面向对象系统中，类是最基本的组成单元。因此，一个显而易见的方案是：让类作为单元测试的基本单元。这种测试方式首先为需要测试的类创建一个对象，并通过它来检查所选定方法的执行结果是否和预期结果一致。一般来说，我们并不需要测试所有的方法，这是因为测试所有的方法通常是不可能的，而且我们也不必要测试所有的方法。测试的目标只是发现和纠正正在应用程序中部署类时可能出现的问题。

单元测试是一种编程行为，因此每个单元测试过程都会涉及所测试类的编码细节。这就是所说的“白盒测试”(white box testing)——深入类的内部机制来进行测试。还有一种测试方式是“黑盒测试”(black box testing)，如其名所示，测试人员不能了解类的内部机制。黑盒测试的目的是，在不知道内部代码方式的条件下，测试所实现方法的所有功能。在第11章和第13章中，我们使用的测试用例（功能性）都是黑盒测试方式。

实现单元测试的最显而易见的方法应该是编写用来输出期望结果的测试脚本。对于第13章范例01中的Book类和Library类，可以编写如下测试脚本，并保存为runBookTest.groovy文件。

```
class Book {  
  
    def String toString(){  
        return "Book:${catalogNumber}:${title} by ${author}"  
    }  
  
    //----properties -----  
  
    def catalogNumber  
    def title  
    def author  
}  
  
//create the Book under test  
def bk1 = new Book(catalogNumber : '111', title : 'Groovy', author: 'Ken')  
  
//test the method toString  
println bk1
```

执行此脚本，输出结果为：

```
Book: 111: Groovy by: Ken
```

测试人员可以明确地比对实际输出值和期望输出值之间的差异。

不幸的是，随着测试代码增加，测试人员的工作量也随之增加。需要测试的类代码一旦改动，就必须运行测试脚本，以确认程序没有引入任何新问题。在我们的脑海中，如果存在大量的此类测试，那么可以认为这种测试方法是不合适的。这种测试方法过于浪费时间，毫无疑问，你会厌烦的。

当然也有其他的一些替代方案，但是大部分方案都存在着某种缺陷。举例来说，向调试器增加测试代码并不比编写测试脚本的方式更佳。这样的测试方法通常需要最初的程序开发人员解释其功能，这就意味着随着时间的流逝，它们并不能保留其优点。当找不到最初的程序开发人员时，进行此类测试就非常困难，甚至根本无法实施。另外一个替代方法是使用第7章范例14类似的断言，但是，使用断言检查方式通常会不必要的提升代码的复杂度，对程序执行速度会起到负面影响。

使用商用的测试工具同样也是可行的，但这些测试工具通常都非常昂贵。为了高效地使用这些工具，往往还需要投入相当大的时间和精力。对单元测试来说，这些工具往往太过“完美”，更适合那些测试要求较高的测试环境，如作为大型应用程序的用户用例（功能性）测试。

15.2 GroovyTestCase类和JUnit TestCase类

JUnit是一个开源的测试框架，符合Java代码的自动化单元测试工业标准（参见<http://www.junit.org>）。最初由Erich Gamma和Ken Beck共同开发，其主要目标是编写一个程序员能真正使用的单元测试框架。第二个目标是鼓励大家编写随着时间流逝仍具有价值的测试程序，并通过这些现有的测试程序开发新的测试程序。

幸运的是，JUnit框架可以非常容易地用于测试Groovy类。所有要做的事情就是扩展GroovyTestCase类，它是基于JUnit TestCase类开发的标准Groovy环境的一部分。

通常，一个单元测试用例由多个测试方法组成，每个测试方法测试目标类的一个方法。比如，可以使用下面的代码替代前一个测试脚本：

```
import groovy.util.GroovyTestCase

class BookTest extends GroovyTestCase {

    /**
     * Test that the expected String is returned from toString
     */
    def void testToString() {
        def bk1 = new Book(catalogNumber : '111', title : 'Groovy', author : 'Ken')
        def expected = 'Book: 111: Groovy by: Ken'

        assertEquals(expected, bk1.toString())
    }
}
```

在这里构建了GroovyTestCase类，可以最大程度地减少用户需要做的工作。举例来说，在BookTest类中，每个冠以test前缀的方法，都将像通常的Groovy脚本一样被编译和执行。它适用于扩展GroovyTestCase类的所有类。

在测试方法中，我们也可以对代码的状态使用断言。在测试方法的执行期间，如果一个断言为假，则表明有问题存在且测试失败。Groovy有一套专门用来报告故障位置和特性的自动机制。当断言为真时，则表明通过测试。在程序中可以使用多个不同的断言，但由于前面所述的原因，在程序中通常很少使用断言。欲获得有关断言的更详细信息，有兴趣的读者可以访问Groovy相关web站点。

举例来说，在testToString中，为了测试方法是否实现预期功能，我们可以通过bk1引用Book类中的toString方法，并使用下面的断言实现测试功能：

```
assertToString(bk1, expected)
```

如果从bk1.toString()所返回的字符串值与期望值相同，则断言为真，测试通过。否则断言为假，测试失败。

执行BookTest，测试报告如下：

```
Time: 0.05
```

```
OK (1 test)
```

请注意，BookTest测试已经通过，其返回信息很少。更确切的说，点表示测试与执行时间在很短的时间内完成。在这里我们是故意这样做的，这是因为我们并不需要更多的测试通过信息。但是，如果断言失败，会提供给我们更为详细的报告。

请思考第13章Example01中的类Library，其单元测试更为有趣。假设使用如下测试代码：

```
import groovy.util.GroovyTestCase

class LibraryTest extends GroovyTestCase {

    /**
     * Set up the fixture
     */
    void setUp(){
        lib = new Library(name : 'Dunning')

        bk1 = new Book(catalogNumber : '111', title : 'Groovy', author : 'Ken')
        bk2 = new Book(catalogNumber : '222', title : 'OOD', author : 'Ken')
    }

    /**
     * Test that addition of a Book to an empty Library results in one more Book
     * in the Library
     */
    void testAddBook_1(){

    }
}
```

```

def pre =lib.loanStock.size()
lib.addBook(bk1)
def post =lib.loanStock.size()

assertTrue('one less book than expected',post ==pre +1)
}

/**
 * Test that the addition of two Books with different catalog numbers
 * to an empty Library results in two Books in the Library
 */
void testAddBook_2(){
    lib.addBook(bk1)
    lib.addBook(bk2)
    def expected =2
    def actual =lib.loanStock.size()

    assertTrue('unexpected number of books',expected ==actual)
}

//----properties -----
def lib

def bk1
def bk2
}

```

请注意，我们在程序中使用编码方式来注释说明测试方法。当然，可以使用更有意义的方法名，如testAddPublicationWithDifferentCatalogNumber，这将是更受欢迎的选择。

setUp方法用来为测试方法建立测试环境（上下文）。测试环境通常也被称作测试工具（test fixture），它必须在执行测试方法时初始化。这样做可以确保测试之间不存在冲突，而且这些测试能够以任意次序执行。在执行每个测试方法之前，Groovy将自动执行setUp方法。

在类LibraryTest中，测试环境是lib引用的Library对象，以及两个分别被bk1和bk2引用的Book对象。它们都在LibraryTest类中作为属性定义，并且都通过setUp方法初始化。

在每个测试方法中，我们声称断言都为真。如果结果不是如此的话，测试将失败且会报告适当的提示消息。举例来说，所使用断言是，在testAddBook_1方法执行完后，Library中将会有个或者更多的Book对象：

```
assertTrue('one less book than expected', post == pre + 1)
```

如果条件表达式post == pre + 1的结果为真，则断言为真。否则，断言为假，抛出的错误信息如下：

```
one less book than expected
```

结合使用系统报告的提示信息和我们自定义的提示信息，有助于标识问题产生的原因。

在testAddBook_2中，我们也使用了类似的断言，在测试中将判断：向一个空Library添加

两个不同分类号的Book对象，是否会导致Library中有两个Book对象。执行LibraryTest输出结果为：

```
Time: 0.741
```

```
OK (2 tests)
```

现在，我们可以看出这两个测试都已经通过。虽然显示的信息比较简单，但这些测试能够证明Library类的实际行为和设计行为一致。在这里并不需要使用更复杂的单元测试。实际上，一般来说，测试用例并不是越多越好，并且每个需要测试的方法也仅需要一个合乎逻辑的测试方法即可。由于它们能够自动编译、执行并检查结果，因而并不会带来额外的负担。随着测试工作量的不断增加，我们对程序代码的信心也不断增加。

单元测试就是使用程序员的经验来发现和纠正代码中可能存在的错误。为了举例说明，请思考以下问题：假定library中已经存在一个Book对象，当试图向该library添加一个分类号与之相同的Book对象时会产生怎样的结果呢？

用户可能会不确认，并对该对象是否已经加入到Library中心存疑虑。因此，我们需要在LibraryTest中加入如下测试代码：

```
// class LibraryTest
/**
 * Set up the fixture
 */
void setUp(){
    /**
     * ...
     */
    bk3 =new Book(catalogNumber : '222',title : 'UML',author : 'John')
}

/**
 * Test that addition of a Book with the same catalog number
 * as one already present in the Library results in no change
 * to the number of Books in the Library
 */
void testAddBook_3(){
    lib.addBook(bk1)
    lib.addBook(bk2)
    def pre =lib.loanStock.size()
    lib.addBook(bk3)
    def post =lib.loanStock.size()

    assertTrue('one more book than expected',post ==pre)
}
```

执行LibraryTest，结果如下：

```
Time: 0.772
```

```
OK (3 tests)
```

然而，这里出现了一个新问题，library中的对象到底是最初的Book对象，还是新加入的Book呢？

假定我们希望library中的对象为最初的Book对象，那么可以在类LibraryTest中加入另外一个测试方法：

```
/*
 * Test that addition of a Book with the same catalog number
 * as one already present in the Library results in no change
 * to the loan stock
 */
void testAddBook_4(){
    lib.addBook(bk2)
    lib.addBook(bk3)
    def expected ='Book:222:OOD by:Ken'
    def actual =lib.loanStock ['222']

    assertEquals(actual,expected)
}
```

执行LibraryTest，结果如下：

```
....F
Time: 0.901
There was 1 failure:
1) testAddBook_4(LibraryTest)junit.framework.AssertionFailedError:
toString() on value: Book: 222: UML by: John expected:<Book: 222: OOD
by: Ken> but was:<Book: 222: UML by: John>

FAILURES!!!
Tests run: 4, Failures: 1, Errors: 0
```

现在，测试报告表明，第四个测试方法失败（因而其结果为四个点和一个F）。系统随后提供了更详细的出错信息。虽然在这里我们不需要关心提示内容，但请注意：如果此时出现了一个非预期的异常，则报告的内容将是错误，而不是失败。

在类Library的addBook方法中已经发现了此问题，现在将其方法重新编写为：

```
// class Library
def addBook(bk) {
    if(!loanStock.containsKey(bk.catalogNumber))
        loanStock[bk.catalogNumber] = bk
}
```

执行LibraryTest结果为：

```
...
Time: 0.882
```

```
OK (4 tests)
```

幸运的是，所有四个测试均已通过，这更增强了我们对程序代码的信心。请注意，这里的第四个测试仅在前一个无效的测试上做了少许改动。

15.3 GroovyTestSuite类和JUnit TestSuite类

用户可能希望在应用程序中编写一个测试用例类，以测试所有类的方法。如果我们将上面四个测试用例结合起来作为一个实体使用的话，测试用例类就很容易实现。在前一节中，我们发现使用GroovyTestCase类可以非常方便地编写、编译以及执行某个独立的JUnit TestCase。使用同样的形式，GroovyTestSuite类使我们非常容易使用JUnit TestSuite类（设计用来管理多个JUnit TestCase）。

请思考下面的runAllTests.groovy Groovy脚本：

```
import groovy.util.GroovyTestSuite
import junit.framework.Test
import junit.textui.TestRunner

class AllTests {

    static Test suite(){
        def allTests =new GroovyTestSuite()

        allTests.addTestSuite(BookTest.class)
        allTests.addTestSuite(LibraryTest.class)

        return allTests
    }

}

TestRunner.run(AllTests.suite())
```

AllTests类有一个静态方法suite，它返回一个被AllTests引用的GroovyTestSuite实例。对于引用它的对象类来说，所有的类方法中都已经含有GroovyTestSuite方法。请注意Test是一个接口，它是通过GroovyTestSuite实现的。它的作用仅仅是确认是否能够正常运行GroovyTestSuite对象。

执行此脚本时，类TestRunner的静态方法Run将会被调用。Run方法使用的实参是类AllTests的suite方法返回的Test对象。run方法将自动执行GroovyTestSuite的所有GroovyTestCase。此测试用例使用了前面编写的BookTest类和LibraryTest类，其具体实现过程在这里并不需要我们关心。欲获得更详尽的信息，有兴趣的读者请访问JUnit web站点（参见<http://www.junit.org>）。

正如GroovyTestCase一样，正常编译和执行runAllTests的测试报告如下所示：

```
Time: 0.861
OK (5 tests)
```

和以往一样，所有的五个测试（一个BookTest和四个LibraryTest）都正常通过。请注意，TestRunner的执行过程是隐式的。在这里，我们可以发现将它显式的表示也非常容易。有兴趣

的读者可以访问Groovy web(参见<http://groovy.codehaus.org>)站点，了解其他替代方案。

为了使大家真正感受到Groovy的单元测试是多么的有用，请在Library类的addBook方法的返回值中加入如下代码，然后我们就可以轻松地判断出添加一个Book对象是成功还是失败。重新编写后的代码如下：

```
// class Library

def addBook(bk) {
    if(!loanStock.containsKey(bk.catalogNumber)){
        loanStock[bk.catalogNumber] = bk
        return true
    } else
        return false
}
```

在LibraryTest类中加入两个新的测试方法：

```
//class LibraryTest

/**
 * Test that successfully adding a Book to the Library
 * is detected
 */
void testAddBook_5(){
    def success = lib.addBook(bk2)

    assertTrue('addition expected',success)
}

/**
 * Test that unsuccessfully attempting to add a Book with the same
 * catalog number as one already present in the Library is detected
 */
void testAddBook_6(){
    lib.addBook(bk2)
    def success = lib.addBook(bk3)

    assertFalse('no addition expected',success)
}
```

现在所要做的所有事情就是执行runAllTests以生成测试报告：

.....

Time: 1.01

OK (7 tests)

请注意，当条件表达式为false时，assertFalse返回true。这与其等价表达式相比要方便得多：

```
assertTrue('no addition expected', success == false)
```

由于前面所有的测试都已通过，我们有理由相信：在后面的代码改动过程中，并不会对程序产生有害的影响。我们只需要很小的付出就能达到目的，这就是单元测试为什么是如此强大的工具的原因之一。

15.4 单元测试的角色

单元测试是迭代、递增式软件开发过程的一个组成部分。在第13章学习案例的迭代2中，用户通常需要开发对Borrower类的方法进行单元测试的BorrowerTest类。举例来说，对于某个指定的Book对象来说，绝对只能允许一个Borrower对象被借阅一次，其BorrowerTest类代码如下：

```
import groovy.util.GroovyTestCase

class BorrowerTest extends GroovyTestCase {

    /**
     * Set up the fixture
     */
    void setUp(){
        bor1 = new Borrower(membershipNumber :'1234',name :'Jessie')

        bk1 = new Book(catalogNumber :'111',title :'Groovy',author :'Ken')
        bk2 = new Book(catalogNumber :'222',title :'OOD',author :'Ken')
        bk3 = new Book(catalogNumber :'222',title :'UML',author :'John')
    }

    /**
     * Test that a Borrower with no Books on loan can borrow a Book
     */
    void testAttachBook_1(){
        def pre =bor1.borrowedBooks.size()
        bor1.attachBook(bk1)
        def post =bor1.borrowedBooks.size()

        assertTrue('one less book than expected',post ==pre +1)
    }

    /**
     * Test that a Borrower with no Books on loan can borrow two Books
     * with different catalog numbers
     */
    void testAttachBook_2(){
        bor1.attachBook(bk1)
        bor1.attachBook(bk2)
        def expected =2
        def actual =bor1.borrowedBooks.size()

        assertTrue('unexpected number of books',expected ==actual)
    }
}
```

```
/*
 * Test that an attempt to borrow a Book with the same catalog number
 * as one already borrowed results in no change to the number of
 * Books borrowed
 */
void testAttachBook_3(){
    bor1.attachBook(bk2)
    def pre =bor1.borrowedBooks.size()
    bor1.attachBook(bk3)
    def post =bor1.borrowedBooks.size()

    assertTrue('one more book than expected',post ==pre)
}

/*
 * Test that an attempt to borrow a Book with the same catalog number
 * as one already borrowed results in no change to the borrowed books
 */
void testAttachBook_4(){
    bor1.attachBook(bk2)
    bor1.attachBook(bk3)
    def expected ='Book:222:00D by:Ken'
    def actual =bor1.borrowedBooks ['222']

    assertEquals(expected,actual)
}

//----properties -----
def bor1

def bk1
def bk2
def bk3
}
```

为了使Borrower类通过测试，必须重新编写attachBook方法的代码：

```
//class Borrower

def attachBook(bk){
    if(!borrowedBooks.containsKey(bk.catalogNumber)){
        borrowedBooks [bk.catalogNumber ]=bk
        bk.attachBorrower(this)
        return true
    }
    else
        return false
}
```

重新编写代码后，我们将BorrowerTest.class添加到AllTests的GroovyTestSuite中：

```
//...
class AllTests {
    static Test suite(){
        def allTests =new GroovyTestSuite()
        //...
        allTests.addTestSuite(BorrowerTest.class)
        return allTests
    }
}
//...
```

执行runAllTests，其结果为：

```
.....
Time: 1.152
```

```
OK (11 tests)
```

作为迭代2的一部分，同样需要更新类LibraryTest和类BookTest的测试方法，以测试对类Library和类Book的改动。如：

```
//class LibraryTest
/**
 * Set up the fixture
 */
void setUp(){
    //...
    bor1 = new Borrower(membershipNumber :'1234',name :'Jessie')
}

//...

/**
 * Test that registering a Borrower with an empty Library results
 * in one more Borrower in the Library
 */
void testRegisterBorrower_1(){
    def pre =lib.borrowers.size()
    lib.registerBorrower(bor1)
    def post =lib.borrowers.size()

    assertTrue('one less borrower than expected',post ==pre +1)
}
```

```
//...
def bor1
```

```
//...
```

执行runAllTests，其结果为：

```
.....
Time: 1.412
```

```
OK (12 tests)
```

随着代码的增加，我们向应用程序添加方法和测试用例通常都使用这种方式。使用Groovy的单元测试，可使软件开发过程中的单元测试更加高效，产生非常大的效益。

在本书的网站上提供了本章开发的GroovyTestCase类和GroovyTestSuite类的完整代码。

15.5 习题

使用第13章迭代2中的Book类、Borrower类及Library类，为本章开发的LibraryTest类编写三个新的测试方法。

1. 通过不同的会员号向一个空Library对象注册两个Borrower对象，测试在Library中是否存在两个Borrower对象。
2. 试图向Library注册一个Borrower对象，其会员号与Library中某个Borrower对象的会员号相同，测试Library中Borrower对象的数量是否变化。
3. 试图向Library注册一个Borrower对象，其会员号与Library中一个Borrower对象的会员号相同，测试已经注册的Borrower对象是否变化。
4. GroovyTestSuite的一个替代方法是使用如Ant(参见<http://ant.apache.org>)和Maven(参见<http://maven.apache.org>)这样的工具。可惜的是，如此强大的工具不在本书所讨论的范围之内。但是，有兴趣的读者在访问Groovy主站点获得更多的有关Groovy的AntBuilder知识之前，可以从第19章中所讨论的Groovy的XMLBuilder类中学到很多知识。

第16章 学习案例：图书馆应用程序（继承）

图书馆应用程序首次出现在第6章中。在第6章中，我们通过列表和映射相结合的方式，演示了管理图书馆借阅数据库所需的数据结构，在这些集合中所维护的数据都是一些简单的字符串。在第11章中，我们通过使用程序代码和闭包增强了系统性能，为了支持用户交互，还引入了基于文本的菜单。第13章使用了更令人感兴趣的属性和行为对象表示图书馆、借阅者和书籍，并删除了所有的输入/输出功能，且重新创建一个类专门用来处理输入/输出交互事件。

本章的前两个迭代部分将使用类的继承模型重新实现此学习案例，这样我们表示的对象就不仅仅是书和杂志，而且可以是所有公开发行的出版物。在早期版本中，我们使用容器辅助表示各个对象之间所建立的关系模型。迭代3将解决检测到的错误及用户反馈信息，以增强系统功能。最后一个迭代将演示如何使用Groovy处理模型上的限制。

16.1 需求规范

和第13章的学习案例一样，首先假定我们对图书馆的运作已经非常熟悉，且理解应用程序需要完成下列功能。

library拥有一个name属性，并且能容纳大量的Book对象或者Journal对象。Book和Journal都有一个书名属性和唯一分类号属性。但是，每个Book对象还有作者属性，而每个Journal对象则有出版社属性。应用程序必须能够显示可供借阅的图书信息，以及已经借出的图书信息。在未来的某个时间，library还将拥有其他的存储载体，如视频和压缩光盘。

在library中有很多注册过的Borrower对象，每个Borrower对象都用姓名和唯一会员号表示。一个Borrower可以借阅和归还Book对象或者Journal对象。系统必须记录每个图书馆事务。借阅记录必须包含Book或者Journal信息、借阅者会员号信息，以及出版物的分类号信息。归还记录则只需要包含分类号信息。

系统也必须能够显示那些已经被Borrower借出的书籍条目信息。

我们需要应用程序实现图书馆管理员的相关功能，这些需求可以通过一组用例表示。唯一的不同是现在我们需要实现从“/”到library的添加、借阅以及归还杂志操作。其需求列表如表16-1所示。

表16-1 图书馆应用程序的用户用例

- | | |
|-----------------|-----------------|
| • 添加一个Book对象 | • 显示Borrower信息 |
| • 添加一个Journal对象 | • 借阅一个Book对象 |
| • 显示库存记录 | • 借阅一个Journal对象 |
| • 显示可供借阅的库存记录 | • 归还一个Book对象 |
| • 显示已经借出的记录 | • 归还一个Journal对象 |
| • 注册新的Borrower | |

和前面的学习案例相同，本章也使用迭代方式，所有迭代的目标都成功实现预期的功能。

16.2 迭代1：多态性

需求规范列表提到library中有两种类型的资料：书和杂志。在将来，图书馆还将保存视频和CD。为了存储不同类型的资料，我们需要使用类层次表示这些库存条目。我们可以水平地扩展类以包含新的条目类型，垂直地扩展条目的特性。

最初的类图如图16-1所示。Publication类表示可以从library借出的所有对象。它是一个抽象类，并且拥有所有外借对象的公有属性和行为：条目分类号以及题名，这些属性和它们的字面意义一样。两个子类表示当前图书馆库存中可用的条目类型，除了继承父类Publication的分类号和题名属性外，Book子类还有作者属性，Journal则有编辑属性。

这就需要在Publication.groovy、Book.groovy和Journal.groovy中分别编写Groovy类Publication、Book和Journal。

```
abstract class Publication {

    String toString() { //redfinition
        return "${catalogNumber}:${title}"
    }

    //----properties -----
    def catalogNumber
    def title
}

class Book extends Publication {

    String toString(){
        return 'Book:' +super.toString() + " by::${author}"
    }

    //----properties -----
    def author
}

class Journal extends Publication {

    String toString(){
        return 'Journal:' +super.toString() + " edited by::${editor}"
    }
}
```

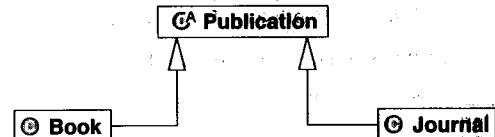


图16-1 最初的类层次

```
//----properties -----
```

```
def editor  
|
```

当展开类的层次结构时，我们需要确认是否已经正确地初始化了对象、所有的多态行为是否如期望的那样运行，这就是迭代的目标。由于所有的三个类都重新定义了toString方法（Publication在Object中重定义了toString方法，而Book和Journal则在Publication中重新定义了toString方法），因此需要确认是否能够实现所期望的多态行为。

下面将讨论前一章节中为了对Book类和Journal类进行单元测试，而创建的类GroovyTestCases、BookTest和JournalTest。在BookTest类中有如下代码：

```
import groovy.util.GroovyTestCase  
  
class BookTest extends GroovyTestCase {  
  
    /*  
     * Test that the expected String is returned  
     */  
    void testToString(){  
        def bk1 =new Book(catalogNumber :'111',title :'Groovy',author :'K Barclay')  
        def expected ='Book:111:Groovy by:K Barclay'  
  
        assertToString(bk1,expected)  
    }  
}
```

JournalTest类也与之相似。请注意单元测试同样也可以测试构造器是否正常运行，这是因为toString方法能够使用所有对象属性。

由于用户也希望在其他类中使用单元测试，因此需要编写运行一个GroovyTestSuite的runAllTests脚本，其代码如第15章所示。

```
import groovy.util.GroovyTestSuite  
import junit.framework.Test  
import junit.textui.TestRunner  
  
class AllTests {  
  
    static Test suite(){  
        def allTests =new GroovyTestSuite()  
  
        allTests.addTestSuite(BookTest.class)  
        allTests.addTestSuite(JournalTest.class)  
  
        return allTests  
    }  
}  
  
TestRunner.run(AllTests.suite())
```

以后，我们也可以方便地添加其他的GroovyTestCase。

运行该代码将产生如下所示的测试报告：

```
Time: 0.39
```

```
OK (2 tests)
```

测试报告证实，程序已经正确地实现了对象的初始化和多态行为，因此，我们已经成功实现本次迭代的目标。

16.3 迭代2：功能性需求演示

在前一次迭代中，我们使用了多态效应，本次迭代的目标是：演示16.1节所描述系统功能需求用例的实现过程。

假定我们引入了抽象类Publication，其属性和行为是所有能够外借对象的公有特征。因此，我们需要调整图13-3所示的类图来反映此关系，如图16-2所示。

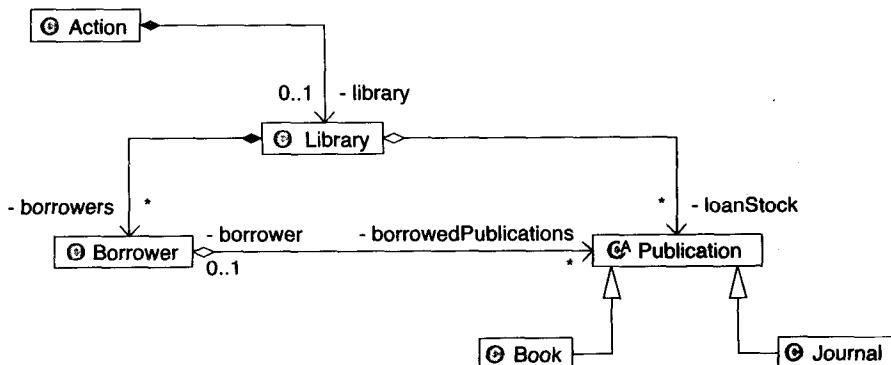


图16-2 类层次

很明显的是，我们可以基于第13章中学习案例的类实现图16-2中的类。当然，我们也可以合并Library类和Borrower类的变动，并将它作为第15章中测试单元的结果。幸运的是，我们同样也可以保留Library类和Borrower类的单元测试，只需要向GroovyTestSuite中runAllTests添加LibraryTest和BorrowerTest方法即可。

由于Groovy具有动态性，因此Library类和Borrower类维护Book集合或者Publication集合并没有任何本质上的区别。所有要做的事就是，消息一定要发送给定义在容器对象的类或者超类中的方法。因此，在这里所做的工作就是对Library、Borrower以及GroovyTestCase类的名称做较小的改动。如：

```
// class Library
def addPublication(publication) {
    if(!loanStock.containsKey(publication.catalogNumber)) {
        loanStock[publication.catalogNumber] = publication
    }
    return true
}
```

```

    |
else

    return false
}

和

// class Borrower ...
def attachPublication(publication) {
    if(!borrowedPublications.containsKey(publication.catalogNumber)) {
        borrowedPublications[publication.catalogNumber] = publication
        publication.attachBorrower(this)
        return true
    }
    else
        return false
}
和

/** 
 * Test that addition of a Book to an empty Library results in one
 * more Publication in the Library
 */
void testAddPublication_1(){
    def pre =lib.loanStock.size()
    lib.addPublication(bk1)
    def post =lib.loanStock.size()

    assertTrue('one less publication than expected',post =pre +1)
}

```

当然，类Action同样也需要做较小的改动。举例来说，读取一个新Book对象详细信息的方法，和读取一个新Journal对象详细信息的方法类似。

```

// class Action
def addJournal() {
    print('\nEnter journal catalog number: ')
    def catalogNumber = Console.readLine()
    print('Enter journal title: ')
    def title = Console.readLine()
    print('Enter journal editor: ')
    def editor = Console.readLine()

    def jo = new Journal(catalogNumber : catalogNumber, title : title, editor : editor)

    library.addPublication(jo)
}

```

最后，我们也修改了呈现用户菜单和执行用户选项的Groovy脚本代码，如Library 01所示。

LIBRARY 01 含有Book对象和Journal对象的Library实例

```
import console.*\n\n\ndef readMenuSelection():\n    println()\n    println('0:Quit')\n    println('1:Add new book')\n    println('2:Add new journal')\n    println('3:Display stock')\n    println('4:Display publications available for loan')\n    println('5:Display publications on loan')\n    println('6:Register new borrower')\n    println('7:Display borrowers')\n    println('8:Lend one publication')\n    println('9:Return one publication')\n\n    print('\n \tEnter choice>>>')\n    return Console.readString()\n}\n\n//make the Action object\n\ndef action = new Action(library :new Library(name :'Dunning'))\n\n//make first selection\n\ndef choice = readMenuSelection()\nwhile(choice !='0'){\n\n    if(choice =='1'){//Add new book\n        action.addBook()\n    }else if(choice =='2'){//Add new journal\n        action.addJournal()\n    }else if(choice =='3'){//Display stock\n        action.displayStock()\n    }else if(choice =='4'){//Display publications available for loan\n        action.displayPublicationsAvailableForLoan()\n    }else if(choice =='5'){//Display publications on loan\n        action.displayPublicationsOnLoan()\n    }else if(choice =='6'){//Register new borrower\n        action.registerBorrower()\n    }else if(choice =='7'){//Display borrowers\n        action.displayBorrowers()\n    }else if(choice =='8'){//Lend one publication\n        action.lendPublication()\n    }else if(choice =='9'){//Return one publication\n        action.returnPublication()\n    }else {\n        println("Unknown selection ")
```

```
    //next selection  
choice = readMenuSelection()  
}  
println('\nSystem closing')
```

完整的脚本及其所支持的类的代码在本书的Web站点中已经给出。

为完成此迭代过程，请运行单元测试。幸运的是，它们都能通过测试。下面我们将使用菜单进行功能性测试。一个显而易见的策略是：让选项（通过不同的显示选项辅助）与其先前的用户用例关联起来。一个演示性会话（用户输入使用粗斜体表示）如下所示：

```
0:Quit  
1:Add new book  
2:Add new journal  
3:Display stock  
4:Display publications available for loan  
5:Display publications on loan  
6:Register new borrower  
7:Display borrowers  
8:Lend one publication  
9:Return one publication
```

Enter choice>>>1

Enter book catalog number:**111**

Enter book title:**Groovy**

Enter book author:**K Barclay**

//Present menu to the user

Enter choice>>>2

Enter journal catalog number:**333**

Enter journal title:**JOOP**

Enter journal editor:**S Smith**

//Present menu to the user

Enter choice>>>3

Library:Dunning

=====

Book:111:Groovy by:K Barclay

Journal:333:JOOP edited by:S Smith

//Present menu to the user

Enter choice>>>0

System closing

由于没有碰到任何问题，此节到此结束。

16.4 迭代3：提供用户反馈

在前面的演示范例中，图书管理员需要从系统获得更多的反馈信息，并且希望处理常见的错误。他同样也希望系统实现以下功能：

- 删除某个Publication对象
- 显示Publication对象的详细信息
- 显示所选定的Publication对象
- 显示特定Borrower对象的详细信息
- 显示所选定的Borrower对象

本次迭代的目标是发现错误，给用户提供反馈信息，并实现这些附加的用户用例。

我们将从解决用户错误输入开始。图书管理员已经考虑到用户可能会做如下尝试：

- 添加一个重复的Publication
- 删除一个不存在Publication
- 注册重复的Borrower
- 删除一个不存在Borrower
- 借阅一个不存在的Publication
- 借阅一个已经被借出的Publication
- 借给一个不存在的Borrower
- 归还一个不存在的Publication
- 归还一个尚未借出的Publication
- 显示一个不存在的Publication信息
- 显示一个不存在的Borrower信息

很明显地，我们必须设想出现这些假定的输入，并做相应的处理，然后通知用户出错信息。在Library类上执行大部分检测操作是非常合理的，这是因为它和注册一个新Borrower的方法一样，也有添加、删除和归还Publication的方法。

为了给用户显示友好的消息，还需要Library类把合适的文本信息发送到Action类。基本思路是，Library类中添加、删除、借阅或者归还Publication对象的方法应该返回一个字符串形式的返回值，以说明操作结果。Library中用来注册Borrower对象的方法也应该这样做。Library的最终代码如下所示：

```
class Library {  
  
    def addPublication(publication){  
        def message  
        if(loanStock.containsKey(publication.catalogNumber)==false){  
            loanStock [publication.catalogNumber ]=publication  
            message ='Publication added'  
        }  
    }  
}
```

```
else

    message ='Cannot add:publication already present'
    return message
}

def removePublication(catalogNumber){
    def message
    if(loanStock.containsKey(catalogNumber)==true){
        def publication =loanStock [catalogNumber ]
        //
        //note:use of safe navigation
        publication.borrower?.detachPublication(publication)
        publication.borrower =null
        loanStock.remove(catalogNumber)
        message ='Publication removed'
    }
    else
        message ='Cannot remove:publication not present'

    return message
}
def registerBorrower(borrower){
    def message
    if(borrowers.containsKey(borrower.membershipNumber)==false){
        borrowes [borrower.membershipNumber ]=borrower
        message ='Borrower registered'
    }
    else
        message ='Cannot register:borrower already registered'

    return message
}
def lendPublication(catalogNumber,membershipNumber){
    def message
    if(loanStock.containsKey(catalogNumber)==true){
        def publication =loanStock [catalogNumber ]
        if(publication.borrower ==null){
            if(borrowers.containsKey(membershipNumber)==true){
                def borrower =borrowers [membershipNumber ]
                borrower.attachPublication(publication)
                message ='Publication loaned'
            }
            else
                message ='Cannot lend:borrower not registered'
        }
        else
    }
}
```

```
        message ='Cannot lend:publication already on loan'
    }
    else
        message ='Cannot lend:publication not present'

    return message
}

def returnPublication(catalogNumber):
    def message
        if(loanStock.containsKey(catalogNumber)==true)
            def publication =loanStock [catalogNumber ]
            if(publication.borrower !=null){
                publication.borrower.detachPublication(publication)
                message ='Publication returned'
            }
            else
                message ='Cannot return:publication not on loan'
        }
        else
            message ='Cannot return:publication not present'
    return message
}

//-----properties -----
def name
def loanStock =[ :]
def borrowers =[ :]
}
```

和以往一样，我们也构造了一些单元测试，以确认所有的方法是否都正常运行。在 LibraryTest类中，典型测试方法范例如下所示：

```
//class LibraryTest
/**
 * Test that the Library has one less Publication after removal of
 * a Publication known to be in the Library
 */
void testRemovePublication_1(){
    //
    //bk1 is created in the fixture
    lib.addPublication(bk1)
    def pre =lib.loanStock.size()
    lib.removePublication(bk1.catalogNumber)
    def post =lib.loanStock.size()

    assertTrue('one more publication than expected',post ==pre -1)
```

```

}

/**
 * Test that the correct message is available to a client
 */
void testRemovePublication_2(){
    //
    //bk1 is created in the fixture
    lib.addPublication(bk1)
    def actual =lib.removePublication(bk1.catalogNumber)
    def expected ='Publication removed'

    assertTrue('unexpected message',actual ==expected)
}
/**
 * Test that the correct message is available to a client
 */
void testRemovePublication_3(){
    def actual =lib.removePublication(bk1.catalogNumber)
    def expected ='Cannot remove: publication not present'

    assertTrue('unexpected message',actual ==expected)
}

```

请注意，removePublication方法使用了安全导航，这意味着我们并不需要显式地检查Publication对象在借出后是否已经被删除。如果其Borrower属性为null，则不会发送detachPublication消息，也不会抛出null异常。

Action类应该在试图显示Publication对象或者Borrower对象之前，检查所指定的Publication对象或者Borrower对象是否存在。它应该通知用户所碰到问题的特性。由于Action对象能和用户交互，因此这么做是合理可行的。

为了实现余下的新用例，程序引入了两个更为灵活的显示方法。两个方法都使用了第3章所讨论的正则表达式的字符串对象。displaySelectedStock方法将显示分类号从用户输入的String开始的所有Publication对象。第二个方法与此相似，它显示会员号从用户输入的String开始的所有Borrower对象。更新后的Action类的大致要点如下：

```

import console.*

class Action {
    ...
    def removePublication(){
        print('\nEnter publication catalog number:')
        def catalogNumber =Console.readLine()
        def message =library.removePublication(catalogNumber)
        println "\nResult:$message\n"
    }
}

```

```
//...

def displayOnePublication(){
    print('\nEnter publication catalog number:')
    def catalogNumber =Console.readLine()

    def publication =library.loanStock [catalogNumber ]
    if(publication !=null){
        this.printHeader('One publication display')
        println publication
    }
    else {
        println '\nCannot print:No such publication \n'
    }
}

//...

def displaySelectedStock(){
    print('\nEnter start of catalog numbers:')
    def pattern =Console.readLine()
    pattern ='^' +pattern +'*'
    def found =false

    this.printHeader('Selected publications display')
    library.loanStock.each {catalogNumber,publication ->if(catalogNumber =~pattern){
        found =true
        println " ${publication}"
    }

    if(found == false)
        println '\nCannot print:No such publications \n'
}

//...

def displayOneBorrower(){
    print('\nEnter borrower membership number:')
    def membershipNumber =Console.readLine()

    def bor =library.borrowers [membershipNumber ]
    if(bor !=null){
        this.printHeader('One borrower display')
        println bor
        def publications =bor.borrowedPublications
        publications.each {catalogNumber,publication ->println " ${publication}"}
    }
}
```

```

    println '\nCannot print:No such borrower \n'
}

//...

def displaySelectedBorrowers(){
    print('\nEnter start of membership numbers:')
    def pattern =Console.readLine()
    pattern ='^'+pattern +'.*'
    def found =false

    this.printHeader('Selected borrowers display')
    library.borrowers.each {membershipNumber,borrower ->
        if(membershipNumber =~pattern){
            found = true
            println borrower
            def publications = borrower.borrowedPublications
            publications.each {catalogNumber,publication ->println "${publication}"}
        }
    }

    if (found == false)
        println '\nCannot print:No such borrowers \n'
}

//...

private printHeader(detail){
    println "\nLibrary:${library.name}: ${detail}"
    println '=====\\n'
}

//----properties -------

private library
}

```

在这里，请注意私有方法printHeader。在迭代开发过程中，类似的改动方式非常普遍。如果能够提供改动文档以及测试结果的话，就再好不过了。

现在所剩下的工作就是修改以前的Groovy脚本，向用户呈现菜单，并执行用户选项，部分代码如Library 02所示。

LIBRARY 02 由图书和杂志组成的数据库应用程序，提供错误检测和用户反馈

```

import console.*

def readMenuSelection(){

//...

```

```
println('3:Remove a publication \n')

//...
println('5:Display selected publications')
println('6:Display one publication')

//...
println('11:Display selected borrowers')
println('12:Display one borrower \n')

//...
print('\n \tEnter choice>>>')
return Console.readString()
}

//make the Action object
def action = new Action(library :new Library(name :'Dunning'))

//make first selection
def choice = readMenuSelection()
while(choice !='0'){
if(choice =='1'){
//...
}else if(choice =='3'){
    action.removePublication()           //Remove a publication
    //...
}else if(choice =='5'){
    action.displaySelectedStock()        //Display selected stock
}else if(choice =='6'){
    action.displayOnePublication()       //Display one publication
    //...
}else if(choice =='11'){
    action.displaySelectedBorrowers()    //Display selected borrowers
}else if(choice =='12'){
    action.displayOneBorrower()          //Display one borrower
    //...

//next selection
choice =readMenuSelection()
}
println('\nSystem closing \n')
```

现在我们需要执行基于用例的功能测试，一个典型会话摘录如下，其中用户数据输入用黑斜体表示：

```
//...
0:Quit
1:Add new book
```

```
2:Add new journal  
3:Remove a publication  
  
4:Display stock  
5:Display selected publications  
6:Display one publication  
7:Display publications available for loan  
8:Display publications on loan  
  
9:Register new borrower  
10:Display all borrowers  
11:Display selected borrowers  
12:Display one borrower  
  
13:Lend one publication  
14:Return one publication
```

Enter choice>>>2

```
Enter journal catalog number: 124  
Enter journal title: JOOP  
Enter journal editor: S Smith
```

Result:Publication added

//Present menu to the user

Enter choice>>>4

```
Library:Dunning:All publications display  
=====  
Book:111:Groovy by:K Barclay  
Journal:124:JOOP edited by:S Smith  
Book:123:OOD by:J Savage
```

//Present menu to the user

Enter choice>>>5

```
Enter start of catalog numbers: 12
```

```
Library:Dunning:Selected publications display  
=====  
Journal:124:JOOP edited by:S Smith  
Book:123:OOD by:J Savage
```

//Present menu to the user

Enter choice>>>0

System closing

本次迭代将在此结束。

16.5 迭代4：强制性约束

图形化符号，如UML，常常很难标记系统需求规范的具体细节。本次迭代实现的目标是演示Groovy如何帮助用户实现此功能。

我们可以通过给模型元素添加文本注释的方式为模型添加断言。举例来说，图16-3所示的类图显示了对Borrower类的约束，诸如一个Borrower对象不能借阅超出指定数量的Publication对象。

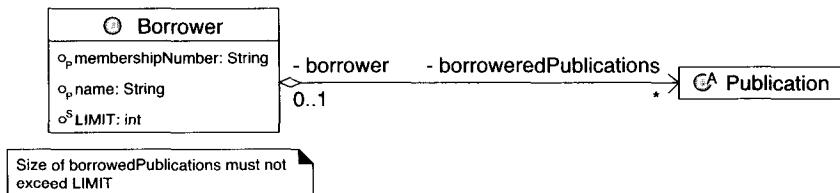


图16-3 显示为文本注释的约束条件

注释中的文本描述了这些约束。它可能使用非正式的英语，如本案例所示，也可能使用更为正式的定义。无论如何，我们在实现过程中必须确认这些约束的有效性。为了实现此功能，程序在Borrower类中加入了一个公有的静态属性LIMIT，并将其初始化为可供借阅的Publication对象的最大数目：

```

class Borrower {
    // ...
    // -----properties -----
    def membershipNumber
    def name
    static public final LIMIT = 4
    private borrowedPublications = [ : ]
}
  
```

在Library类的方法中可以加入检查操作，判断是否超过限制。lendPublication方法的典型检查方法为：

```

//class:Library
def lendPublication(catalogNumber, membershipNumber) {
    def message
    if(loanStock.containsKey(catalogNumber)==true){
        def publication = loanStock[catalogNumber]
        if(publication.borrower ==null){
            if(borrowers.containsKey(membershipNumber)==true){
                def borrower = borrowers[membershipNumber]
                if(borrower.borrowedPublications.size()<Borrower.LIMIT){
                    borrower.attachPublication(publication)
                    this.checkPublicationBorrowerLoopInvariant
                }
            }
        }
    }
}
  
```

```

        ('Library.lendPublication')
        message ='Publication loaned'
    }
    else
        message ='Cannot lend:borrower over limit'
    }
    else
        message ='Cannot lend:borrower not registered'
    }
    else
        message ='Cannot lend:publication already on loan'
}
else
    message ='Cannot lend:publication not present'

return message
}

```

和以前一样，应该更新单元测试，以确认这些代码的执行结果是否和期望值相同。如：

```

//class:LibraryTest
/**
 * Test that the correct message is available to a client
 */
void testLendPublication_7(){
    def bk4 =new Book(catalogNumber :'444',title :'C++',author :'S Smith')
    def bk5 =new Book(catalogNumber :'555',title :'C',author :'A Cumming')
    def bk6 =new Book(catalogNumber :'666',title :'C#',author :'I Smith')
    //
    //bk1 and bk2 are created in the fixture
    def publicationList =[bk1,bk2,bk4,bk5,bk6 ]

    lib.registerBorrower(bor1)

    def actual
    publicationList.each{publication ->
        lib.addPublication(publication)
        actual = lib.lendPublication(publication.catalogNumber, bor1.membershipNumber)
    }

    def expected ='Cannot lend:borrower over limit'

    assertTrue('unexpected message',actual ==expected)
}

```

Groovy测试系统非常容易使用，因而用户可以进行更多的测试。举例来说，用户可以对对象之间关系做出约束，而不是仅限于某个独立的对象。这些强加的约束关系从某个对象开始，在使用测试之前遵循与其他对象之间的结构关系。比如，将任一待借阅的Publication对象借阅给某个Borrower对象，那么该Borrower对象的borrowedPublication属性必须包含对该Publication

对象的引用。换句话说，一个待借阅的Publication对象和它的Borrower对象必须保持一致。

下面是一个闭环（loop invariant）的例子。虽然它不是这里关注的重点，但闭环被广泛地应用于需要验证程序正确性的软件开发过程中，它是一种非常重要的形式化方法。在这里，我们只需要演示从某个对象开始，依次访问与之相关的对象，并最终返回到起始对象的情形。其对象图如16-4所示。

该图表明：如果从给定的Book对象开始，随后通过其Borrower对象，应该在Borrower对象已经借阅的映射中找到以Book对象分类号为关键字的出版物。为了与此模型一致，与该关键字相关的值必须是循环开始的Book对象。使用Groovy编写的闭环检查代码如下所示：

```
//class:Library
private checkPublicationBorrowerLoopInvariant(methodName){
    def publications =loanStock.values().asList()

    def onLoanPublications =publications.findAll{publication ->
        publication.borrower.borrowedPublications.containsKey(publication.catalogNumber)
    }
    if(allOK ==false ){
        throw new Exception("${methodName}:Invariant failed ")
    }
}
```

如果违反闭环，则说明出现了严重的错误，应该抛出含有适当信息的异常消息，并终止系统运行。请注意，并没有说是这个方法抛出异常（参见附录B）。

和以往一样，在这里仅检查那些可能违反闭环的方法。在此案例中，仅需要检查lendPublication方法。

```
//class:Library
def lendPublication(catalogNumber,membershipNumber){
    //...
    if(borrower.borrowedPublications.size()<Borrower.LIMIT){
        borrower.attachPublication(publication)
        this.checkPublicationBorrowerLoopInvariant('Library.lendPublication')
        message ='Publication loaned'
    }else
        message ='Cannot lend:borrower over limit'
    //...
}
```

首先，必须创建至少一个单元测试以检查期望的异常是否抛出。此处可能存在问题，因为

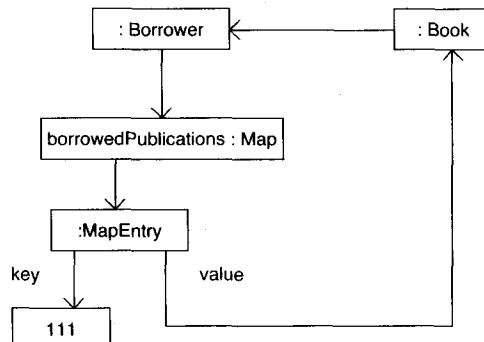


图16-4 Publication-Borrower闭环

已经实现Borrower类中attachPublication方法，以确保不会违反闭环原则。

一个解决方案是创建MockBorrower子类，并重新定义attachPublication方法以处理异常行为：

```
class MockBorrower extends Borrower {

    def attachPublication(publication) {
        //
        // Normal behavior is commented out
        // borrowedPublications[publication.catalogNumber] = publication
        publication.attachBorrower(this)
    }
}
```

在期望Borrower对象正常运行的地方，创建一个MockBorrower对象。

```
//class LibraryTest {
void testCheckPublicationBorrowerLoopInvariant(){
    def mockBorrower =new MockBorrower(membershipNumber :'1234',name :'P Thompson')
    lib.registerBorrower(mockBorrower)
    lib.addPublication(bk1)
    lib.addPublication(bk2)

    try {
        lib.lendPublication(bk1.catalogNumber, mockBorrower.membershipNumber)
        fail('Expected:Library.testPublicationBorrowerLoop:Invariant failed')
    }catch(Exception e){}
}
//...
}
```

请注意，方法fail仅在没有抛出异常的时候报告出错信息。类MockBorrower是mock对象（mock object）测试设计模式（Massol, 2003）的一个实例。它能够避免异常的行为对正常代码的影响。

令人高兴的是，runAllTest脚本的所有测试都能通过。从这一点上来说，执行前一次迭代的Groovy脚本就可以完成功能性测试，如所期望的那样没有出现任何问题，因此本次迭代实现到此结束。

16.6 习题

1. 用户输入的合法性检查是所有交互式系统的一个重要组成部分。请修改最后一次迭代中的Action类以执行如下合法性检查：
 - (a) Borrower的姓名只能由字母表中的字符组成，并且姓和名的首字母必须是大写的，如，K Barclay。
 - (b) Borrower的会员号只能由数字组成，如，1234。
 - (c) Publication的分类号必须由四个数字后面紧跟一个小写字母表示，如，0012a。
2. 学习案例的需求规范说明指出，在未来某个时候，图书馆会保存视频和CD等资料。
 - (a) 修改图16.4所示的类图，使之包含视频和CD等资料。

- (b) 使用最后一次迭代中的Groovy脚本及其所支持的类，编写一个迭代器遍历图书馆数据库中所有Book、Journal和Video对象。假定Video对象有分类号、标题和持续时间属性，其中持续时间以分钟为单位。
3. 请讨论，如果采用本学习案例中所使用的方法或者途径，习题2是更容易还是更难完成？
4. (a) 请实现如下约束条件：每个可供Borrower借阅的Publication对象都在Library的借阅数据库中记录，并且Borrower必须已经注册。
(b) 为上一习题中实现的约束条件设计一个适当的单元测试。
5. (a) 迭代4演示了Publication-Borrower闭环的实现过程。请编写一个Publication-Borrower闭环，它从Borrower开始，导航至每个它所借阅的Publication对象，并检查它是否是与每个Publication相对应的Borrower。
(b) 为上一习题中实现的约束条件设计一个适当的单元测试。
6. 本章在约束条件被打破的时候创建并抛出了一个异常。但是，在其他一些场合中抛出异常也有可能是非常有用的。
(a) 为Borrower姓名、会员号以及Publication分类号加入符合习题1要求的约束条件。
(b) 通过抛出异常实现这些约束条件。
(c) 请说明本题和习题1中所使用方法的优缺点。

第17章 持久性

本章将集中讲解在关系型数据库中持久存储数据的方法（假定读者对结构化查询语言（SQL，Structured Query Language）（Beaulieu, 2005；Molinaro, 2006）已经非常熟悉）。对Java开发者来说，这将涉及到Java数据库连接（JDBC，Java Database Connectivity）API编程。而对于Groovy开发者来说，JDBC API的大部分编程工作已经转移到Groovy框架，极大地减少从数据库中提取数据的相关处理工作。比如，借助于迭代器和闭包，程序员可以轻易地遍历数据库表中数据。

17.1 简单查询

假定某个数据库仅有一个表存储大量银行账户的详细资料，表名为accounts，其结构如图17-1所示。Accounts表的一行表示一个个人账户，列表示账户号码以及当前账户余额。

groovy.sql包提供使用SQL的数据访问能力，该API广泛应用迭代器和闭包，以处理SQL查询返回的结果。范例01演示了查询accounts表中所有行和显示详细信息的方法。欲获取有关创建数据库方面的知识，参见附录A或者请访问本书的web站点。

范例01 简单的SQL查询

```
import groovy.sql.*  
  
def DB = 'jdbc:derby:accountDB'  
def USER = ''  
def PASSWORD = ''  
def DRIVER = 'org.apache.derby.jdbc.EmbeddedDriver'  
  
//Connect to database  
def sql = Sql.newInstance(DB,USER,PASSWORD,DRIVER)  
  
//Iterate over the result set  
println 'Accounts'  
println ' Number Balance '  
println '+-----+-----+'  
sql.eachRow('select * from accounts'){acc ->  
    printf('|%-8s |%-8d |\n',[acc.number,acc.balance ])  
}  
println '+-----+-----+'
```

accounts	
number	balance
ABC123	1200
DEF456	400
...	...

图17-1 Accounts表

Sql类包含newInstance方法，用于连接到所需的数据库。在本案例中，我们创建一个Sql实例，指向本地的Cloudscape数据库（参见<http://db.apache.org/derby/>）（参见附录A）。数据库标识是JDBC链接URL地址jdbc:derby:accountDB。数据库驱动器的类名为org.apache.derby.

jdbc.EmbeddedDriver。更多有用的代码如下所示。迭代器方法eachRow接收两个参数，也就是表示SQL查询语句的字符串，以及处理查询结果的数据行的闭包。在这里，闭包只负责输出账户号码及其当前余额信息。

该程序的输出结果为：

Accounts	
Number	Balance
ABC123	1200
DEF456	400

下一个范例将对同一个数据库表执行相同的查询操作。使用查询结果创建Account对象，并将这些对象添加到列表中。最后，简单地显示所有对象属性。

范例02 创建对象

```
import groovy.sql.*  
class Account {  
    String toString(){  
        return "Account:${number}${balance}"  
    }  
  
    //----properties -----  
  
    def number  
    def balance  
}  
def DB ='jdbc:derby:accountDB'  
def USER =''  
def PASSWORD =''  
def DRIVER ='org.apache.derby.jdbc.EmbeddedDriver'  
  
    //Collection of Account objects  
def accounts =[]  
  
    //Connect to database and make query  
def sql =Sql.newInstance(DB,USER,PASSWORD,DRIVER)  
sql.eachRow('select * from accounts'){acc ->  
    accounts <<new Account(number :acc.number,balance :acc.balance)}  
}  
  
    //Display accounts  
accounts.each {acc ->  
    println "${acc}"  
}
```

输出结果证实已经成功地创建Account对象：

```
Account: ABC123 1200
Account: DEF456 400
```

17.2 关系

假设，数据库中有多个用来存储银行和账户信息的表。其中，每个账户都有账户号码和余额属性。为了标识账户的所属银行，accounts表也应该包含一个外部关键字（foreign key）。这个外部关键字唯一地标识账号所属的银行，也存在与银行表中，作为后者的主关键字。遵循这种模式，同样也给每个账户分配一个唯一的标识。表的结构如图17-2所示。banks表将id列作为主关键字。accounts表同样也将id作为主关键字。在accounts表中，bankID列是外部关键字，标识当前账户所属的银行。

范例03演示了数据库的某些处理过程。前两个查询语句仅仅以列表的形式显示两个表的数据。最后一个查询则包含一个where子句，只检索银行ID为RBS的账户。

范例03 | 关系

```
import groovy.sql.*

def DB ='jdbc:derby:bankDB'
def USER =''
def PASSWORD =''
def DRIVER ='org.apache.derby.jdbc.EmbeddedDriver'

//Connect to database
def sql = Sql.newInstance(DB,USER,PASSWORD,DRIVER)

//Query the bank table
println 'Banks'
println ' Name '
println '+-----+'
sql.eachRow('select * from banks'){bk ->
    printf('|%-30s |\n',[bk.name ])
}
println '+-----+'
println()

//Query the accounts table
println 'Accounts'
```

banks

id	name
RBS	Rich Bank of Scotland
BOS	Banque of Scotland
...	...

accounts

id	bankID	number	balance
1	RBS	ABC123	1200
2	RBS	DEF456	400
3	BOS	GHI789	600
...

图17-2 关系

```

println ' Number Balance Bank '
println '+-----+-----+'
sql.eachRow('select * from accounts'){acc ->
    printf('|%-8s |%-8d |%-4s |\n',[acc.number,acc.balance,acc.bankID ])
}
println '+-----+-----+'
println()

        //Find the RBS accounts
println 'RBS accounts'
println ' Number Balance '
println '+-----+'
sql.eachRow('select * from accounts where bankID =?',['RBS']){acc ->
    printf('| %-8s| %-8d|\n',[acc.number,acc.balance ])
}
println '+-----+'
println()

```

请看如下查询语句：

```
sql.eachRow('select * from accounts where bankID = ? ', ['RBS']) ...
```

在这里，where子句用来选择那些bankID为RBS的accounts信息。在此范例中，“?”符号表示用列表中的值替代。如果where子句包含多个“?”符号，则它们被依次列表中的值替代。

此应用程序的输出结果如下面的文字所示。可以发现有两个bank，三个accounts，并且两个accounts从属于RBS银行。

Banks		
Name		
Rich Bank of Scotland		
Banque of Scotland		

Accounts		
Number	Balance	Bank
ABC123	1200	RBS
DEF456	400	RBS
GHI789	600	BOS

RBS accounts	
Number	Balance
ABC123	1200
DEF456	400

17.3 更新数据库

数据库中的表能够通过SQL的insert语句更新，它将为指定的表添加新行。SQL delete语句

将以同样的方式从指定的表中删除数据行。这两个语句将在范例04中演示。在这里使用了前三个范例所使用的账户数据库。在示例代码中，为accounts表加入了两个新行，随后立即将它们删除，并退出数据库，数据库实际上并没有变化。在数据处理每一个阶段都输出accounts表的具体内容，以监视其效果。

范例04 更新数据库记录

```
import groovy.sql.*

def DB ='jdbc:derby:accountDB'
def USER =''
def PASSWORD =''
def DRIVER ='org.apache.derby.jdbc.EmbeddedDriver'

def displayAccounts(banner,sql){
    println banner
    sql.eachRow('select * from accounts'){acc ->
        println " Account::${acc.number}|${acc.balance}"
    }
    println()
}

//Connect to database
def sql = Sql.newInstance(DB,USER,PASSWORD,DRIVER)

//Iterate over the result set
displayAccounts('Initial content',sql)

//Now insert a new row...
sql.execute("insert into accounts(number,balance)values('GHI789',600)")

//...and another
def newNumber ='AAA111'
def newBalance =1600
sql.execute("insert into accounts(number,balance)values(${newNumber}, ${newBalance})")

//Now see what we have
displayAccounts('After inserts',sql)

//Restore original
['GHI789','AAA111'].each {accNumber ->
    sql.execute('delete from accounts where number =?',[accNumber ])
}

//Now see that they have gone
displayAccounts('After deletes',sql)
```

输出结果为：

```
Initial content
Account: ABC123 1200
Account: DEF456 400
```

```
After inserts
Account: ABC123 1200
Account: DEF456 400
Account: GHI789 600
Account: AAA111 1600
```

```
After deletes
Account: ABC123 1200
Account: DEF456 400
```

groovysql包中也包含DataSet类。DataSet类是Sql类的一个扩展，表示数据库表的对象。程序员可以通过DataSet的each方法遍历所有数据行，通过add方法添加一个新行，并能执行从超类Sql继承的方法，如execute。DataSet类的用法如范例05所示，重复范例04所示的逻辑关系。

范例05 使用DataSet更新数据库记录

```
import groovy.sql.*

def DB ='jdbc:derby:accountDB'
def USER =''
def PASSWORD =''
def DRIVER ='org.apache.derby.jdbc.EmbeddedDriver'

def displayAccounts(banner,dSet){
    println banner
    dSet.each {acc ->
        println " Account:${acc.number}${acc.balance}"
    }
    println()
}

//Connect to database
def sql = Sql.newInstance(DB,USER,PASSWORD,DRIVER)
def accounts =sql.dataSet('accounts')

//Iterate over the data set
displayAccounts('Initial content',accounts)

//Now insert a new row...
accounts.add(number :'GHI789',balance :600)

//...and another
def newNumber ='AAA111'
def newBalance =1600
```

```

accounts.add(number :newNumber,balance :newBalance)

    //Now see what we have
displayAccounts('After inserts',accounts)

    //Restore original
['GHI789','AAA111'].each {accNumber ->
    accounts.execute('delete from accounts where number =?',[accNumber ])
}

    //Now see that they have gone
displayAccounts('After deletes',accounts)

```

本节的最后一个范例同时操作数据库中的Banks表和Accounts表。如前例所示，首先加入了一个新的bank对象，然后加入与该bank对象关联的一个新的account对象，随后取消了更新，并退出数据库，数据库实际上也没有变化。

范例06 关系更新

```

import groovy.sql.*

def DB ='jdbc:derby:bankDB'
def USER =''
def PASSWORD =''
def DRIVER ='org.apache.derby.jdbc.EmbeddedDriver'

def displayBanks(banner,dSet){
    println banner
    dSet.each {bk ->
        println " Bank:::${bk.id}${bk.name}"
    }
    println()
}

def displayAccounts(banner,dSet){
    println banner
    dSet.each {acc ->
        println " Account:::${acc.id}${acc.bankID}${acc.number}${acc.balance})"
    }
    println()
}

//Connect to database
def sql = Sql.newInstance(DB,USER,PASSWORD,DRIVER)
def banks = sql.dataSet('banks')
def accounts = sql.dataSet('accounts')

//Query the bank table

```

```
displayBanks('Banks',banks)

    //Query the account table
displayAccounts('Accounts',accounts)

    //Now add a new bank...
banks.add(id :'CB',name :'Clydebank')

    //...and check it is there
displayBanks('Banks (after add)',banks)

    //Now add a new account to this new bank...
accounts.add(bankID :'CB',number :'AAA111',balance :1600)

    //...and check it is there...
displayAccounts('Accounts (after add)',accounts)

    //...then remove it...
accounts.execute('delete from accounts where number =?',['AAA111'])

    //...and check it is gone
displayAccounts('Accounts (after delete)',accounts)

    //Now remove the new bank...
banks.execute('delete from banks where id =?',['CB'])

    //...and check it is gone
displayBanks('Banks (after delete)',banks)
```

其输出结果为：

Banks

```
Bank:RBS Rich Bank of Scotland
Bank:BOS Banque of Scotland
```

Accounts

```
Account:1 RBS ABC123 1200
Account:2 RBS DEF456 400
Account:3 BOS GHI789 600
```

Banks (after add)

```
Bank:RBS Rich Bank of Scotland
Bank:BOS Banque of Scotland
Bank:CB Clydebank
```

Accounts (after add)

```
Account:1 RBS ABC123 1200
Account:2 RBS DEF456 400
Account:3 BOS GHI789 600
```

```
Account:5 CB AAA111 1600

Accounts (after delete)
Account:1 RBS ABC123 1200
Account:2 RBS DEF456 400
Account:3 BOS GHI789 600

Banks (after delete)
Bank:RBS Rich Bank of Scotland
Bank:BOS Banque of Scotland
```

17.4 表的对象

Groovy是一门面向对象的脚本语言。如何存储那些在以后会被同一个或者其他应用程序使用的对象是一个难题。使用关系型数据库来持久保存数据是一个非常自然和明显的选择。但是，这种做法也会引发一个新问题，就是如何把关系型表中数据映射到对象。在本节和下一节中，我们将实现一个简单映射模式。在最后一节中，我们将使用Groovy的“粘合（glue）”技术，并为此功能开发一个框架。

请思考如何在本章的前两个范例所使用的accountDB数据库的accounts表中创建Account对象。读者可能会简单地认为，使用eachRow迭代器，并从表的每个行中创建一个Account对象即可达到目的。然而，指定的数据库中可能包含多个不同结构的表，这样，从表中创建对象就需要更全面的解决方案。

请注意下面列出的抽象类SqlQuery，它的两个属性分别为Sql对象，以及在表中检索所有元素的SQL查询字符串。SqlQuery类使用模板方法（template method）（Gamma et al., 1995）mapRow，从数据库表中的单个行中传递所需类型的对象。execute方法使用mapRow传递从表的所有列中构建出来的对象列表。调用sql对象的rows方法，并将查询字符串作为参数传递可以实现此功能。查询将返回一个结果集列表，我们随后会处理每个条目，以将每行数据转换成对象。

```
abstract class SqlQuery {

    def SqlQuery(sql,query){
        this.sql =sql
        this.query =query
    }

    def execute(){
        def rowsList =sql.rows(query)
        def results =[]
        def size =rowsList.size()
        0.upto(size -1){index ->
            results <<this.mapRow(rowsList [index ])
        }
        return results
    }

    def abstract mapRow(row)
```

```
//----properties -----
```

```
def sql  
def query  
|
```

现在，我们可以固化这个解决方案，专门用来从accounts表的数据行中检索由Account对象所组成的列表，这也就是类AccountQuery所要实现的功能：

```
class AccountQuery extends SqlQuery |  
  
    def AccountQuery(sql){  
        super(sql,'select * from accounts')  
    }  
  
    def mapRow(row){  
        def acc =new Account(number :row.getProperty('number'),  
                            balance :row.getProperty('balance'))  
        return acc  
    }  
|
```

mapRow方法只有一个GroovyRowResult (参见GDK)参数，用来表示表中的一行数据。可以通过名字或者使用索引来访问该行的每个列，与之相应的方法分别是getProperty 和getAt。在这里，使用getProperty方法访问每个列，并初始化Account对象。

最后，范例07演示了从accounts表的数据中重新创建Account对象的方法，程序随后显示了这些对象的属性。

范例07 表中的对象

```
import groovy.sql.*  
  
def DB ='jdbc:derby:accountDB'  
def USER =''  
def PASSWORD =''  
def DRIVER ='org.apache.derby.jdbc.EmbeddedDriver'  
  
    //Connect to database  
def sql = Sql.newInstance(DB,USER,PASSWORD,DRIVER)  
  
    //Prepare the query object  
def accQuery = new AccountQuery(sql)  
  
    //Get the Accounts  
def accs =accQuery.execute()  
  
accs.each {acc ->  
    println "${acc}"  
}|
```

17.5 继承

在包含继承的应用程序模型中，需要能够表示数据库的每种具体类型。假定bank类和account类的层次关系如第14章中范例04所示，并且希望持久化保存类CurrentAccount和类DepositAccount。为了处理两种类型的账户，一个解决方案是新建一个数据库表，包含所有子类的属性。它还应该包含账户号码、余额、透支额度和利率等属性。为了分清两种账户类型，还应该包含一个表示账户类型的列（图17-3）。范例08演示了数据库读取、创建对象，以及显示它们的方法。

accounts				
type	number	balance	overdraftlimit	interestrade
CURRENT	AAA111	2000	400	null
CURRENT	BBB222	3000	800	null
DEPOSIT	CCC333	4000	null	4
...

图17-3 Accounts表

范例08 继承

```
import groovy.sql.*

def DB ='jdbc:derby:specialDB'
def USER =''
def PASSWORD =''
def DRIVER ='org.apache.derby.jdbc.EmbeddedDriver'

//Connect to database
def sql = Sql.newInstance(DB,USER,PASSWORD,DRIVER)

//Prepare the query object
def accQuery =new SpecialAccountQuery(sql)

//Get the Accounts
def accs =accQuery.execute()

accs.each {acc ->
    println "${acc}"
}
```

我们再一次继承了SqlQuery类。在这里，SpecialAccountQuery类中的mapRow方法需要识别所创建Account对象的类型，在表中使用type列表示。

```
class SpecialAccountQuery extends SqlQuery {

    def SpecialAccountQuery(sql){
        super(sql,'select * from accounts')
    }
}
```

```

def mapRow(row){
    def acc =null

    if(row.getProperty('type')=='CURRENT')
        acc = new CurrentAccount(number :row.getProperty('number'),
                                balance :row.getProperty('balance'),overdraftLimit :row.getProperty('overdraftlimit'))
    else
        acc = new DepositAccount(number :row.getProperty('number'),
                                balance :row.getProperty('balance'),interestRate :row.getProperty ('interestrate'))
    return acc
}
}

```

17.6 Spring框架

Spring框架 (Johnson et al., 2005; Wall et al., 2004) 是一个重要的开源应用程序开发框架, 其原意是使Java/J2EE开发更加容易和高效。Spring将目标定为辅助构建完善的应用程序。关于Spring的详细讨论已经超出了本书范畴, 读者可以查阅相关的参考资料。无论如何, 当用户开发大型复杂应用程序时, 非常值得考虑使用Spring框架。

使用Spring的一个好处是, 能够从数据存储对象 (DAO) 设计模式中获利。DAO模式的最初意图是从普通应用程序类和应用程序逻辑中分离持久性。和以前所讨论的MVC模式一样, DAO模式从代码中分离数据库技术知识。

图17-4所示的类图描述了问题的模型。Bank和Accounts之间存在一对多的关系。Account类含有与getter和setter相匹配的number和balance属性。

图17-5描述了维护不同类型账户数据的accounts表。通过类似于前一节所述的方式进行组织, accounts表包含在accountDB数据库中。

Spring使用一个DataSource对象连接数据库。在此处, 我们使用DriverManagerDataSource实现与数据库的连接, 对于测试环境或者独立环境来说, 这是一个非常实用的做法。创建实例的代码如下所示:

```
ds = new DriverManagerDataSource(driverClassName : 'org.apache.derby.jdbc.EmbeddedDriver',
                               url : 'jdbc:derby:accountDB', username : '', password : '')
```

为了执行SQL查询, 并将结果映射到Groovy类, Spring在org.springframework.jdbc.object包中提供了一组类。比如, MappingSqlQuery类执行查询, 并从查询结果中获取对象。下面是AccountQuery类扩展MappingSqlQuery类的代码:

```

import java.sql.*
import org.springframework.jdbc.object.*

class AccountQuery extends MappingSqlQuery {

```

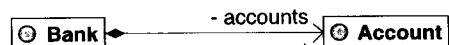


图17-4 银行应用程序

accounts

number	balance
AAA111	2000
BBB222	3000
CCC333	4000
...	...

图17-5 Accounts表

```

def AccountQuery(ds){
    super(ds,'select * from accounts')
    this.compile()
}
protected Object mapRow(ResultSet rs,int rowNumber){
    def acc =new Account(number :rs.getString('number'),balance :rs.getInt('balance'))
    return acc
}
}

```

超类MappingSqlQuery的核心思想是，让用户指定通过execute方法执行的SQL查询，这个查询在子类AccountQuery的构造器中指定。子类必须包含（protected；参见附录I）mapRow方法的实现，把查询结果的每个行映射为一个对象（表示通过查询获取的实体）。因此，execute方法将返回Account对象的一个列表。

使用同样的方式，类AccountInsert可以在数据库表中插入一个新行：

```

import java.sql.*
import org.springframework.jdbc.object.*
import org.springframework.jdbc.core.*

class AccountInsert extends SqlUpdate {

    def AccountInsert(ds){
        super(ds,'insert into accounts(number,balance)values(?,?)')
        this.declareParameter(new SqlParameter(Types.VARCHAR))
        this.declareParameter(new SqlParameter(Types.INTEGER))
        this.compile()
    }
}

```

现在可以创建一个描述DAO功能性需求的接口。在本范例中，接口用来检索数据库中所有的Accounts对象，并向数据库添加新的Account对象。

```

interface BankDaoIF {

    def abstract getAccounts()
    def abstract addAccount(acc)
}

```

类BankDaoJdbc是实现此接口的JDBC。使用类AccountQuery和类AccountInsert可以非常容易地实现类BankDaoJdbc。举例来说，方法getAccounts只是调用AccountQuery类实例的execute方法，并返回Accounts列表。

```

import org.springframework.jdbc.object.*
import org.springframework.jdbc.core.*

class BankDaoJdbc implements BankDaoIF {
    def getAccounts()
}

```

```
def aQuery =new AccountQuery(dataSource)
return aQuery.execute()
}

def addAccount(acc){
    def params =[acc.number,acc.balance ]
    def aInsert =new AccountInsert(dataSource)
    aInsert.update(params as Object [])
}

//----properties -----


def dataSource
```

用户只需要简单地改动Bank类代码，就可利用数据库的持久性。我们在代码中还加入了通过Bank构造器初始化的BankDaoIF引用。构造器同样也调用了DAO的getAccounts方法，并且将accounts属性初始化为一个列表。作为一个简单的演示，在这里也包含openAccount方法。它通过DAO中的addAccount方法实现，随后将新建的账户加入到accounts集合中。

```
class Bank {

    def Bank(name,dao){
        this.name =name
        this.dao =dao

        accounts =dao.getAccounts()
    }

    def openAccount(account){
        dao.addAccount(account)
        accounts <<account
    }

//----properties -----


    def name
    def accounts

    def dao
}
```

最后，我们开发了一个简单的演示脚本。在这里，程序首先创建了一个Bank对象，并使用数据库中的内容初始化Bank对象以新开一个账户，然后列出Bank中的所有账户。如果运行这个脚本仅仅为了更新数据库的话，请运行另一个用来演示的脚本版本，从中看到如何实现持久性。

范例09 Spring框架

```
import groovy.sql.*
import org.springframework.jdbc.datasource.*
```

```

def DB ='jdbc:derby:accountDB'
def USER =''
def PASSWORD =''
def DRIVER ='org.apache.derby.jdbc.EmbeddedDriver'

def displayBank(bk){
    println "Bank:${bk.name}"
    println '====='
    bk.accounts.each {account ->println " ${account}"}
    println()
}

def ds =new DriverManagerDataSource(driverClassName :DRIVER,url :DB,
                                   username :USER,password :PASSWORD)
def dao =new BankDaoJdbc(dataSource :ds)
def bk =new Bank('Napier',dao)

def da =new Account(number :'DDD444',balance :5000)
bk.openAccount(da)

//now display everything
displayBank(bk)

```

输出结果为：

```

Bank: Napier
=====
Account: AAA111; 2000
Account: BBB222; 3000
Account: CCC333; 4000
Account: DDD444; 5000

```

17.7 习题

1. 以范例01为模板，计算那些余额超过1000的账户信息列表。在eachRow迭代器中使用简单的if语句。
2. 使用SQL的select语句检索习题1中期望得到的账户信息列表。
3. eachRow方法包含一个queryString参数和表示操作的闭包参数。使用另一个版本eachRow(query, params, closure)重载此方法，其中，params是范例03中的参数值列表。使用新的eachRow方法列出那些账户余额超过某个指定值的账户信息列表。
4. 继承的一个替代方案是为每个类创建一个数据库表。因此，17.5节中的范例可由三个表组成：accounts、currentaccounts和depositaccounts。每个表都有与其相对应的类相同的属性信息。当然，必须使用表currentaccounts和表depositaccounts的外部关键字，以关联accounts表中的数据。开发一个程序，读取这个数据库，并且列表显示活期账户和有息存款账户信息。
5. 完善17.6节的范例，使其类层次和第14章的类层次相似。

第18章 学习案例：图书馆应用 程序（持久性）

本章将完善第16章最后一次迭代所开发的应用程序范例，图书馆、借阅者和出版物（域模型）等信息都将持久性存储在数据库中。以前，我们曾用使用Action对象实现模型-视图-控制器（MVC）框架，使得域模型独立于用户交互代码。本学习案例将引入一个数据访问对象（DAO，Data Access Object），使域模型独立于数据库持久性代码。这个目标可以通过Spring框架和Cloudscape DBMS实现。

在这里，类Publication、Book和Journal都不需要改动，类Borrower也只需要做很小的改动。但是类Library和运行应用程序的主Groovy脚本改动很大，两者都依赖于使用Spring框架的DAO实现。迭代1将详细说明所需的改动。

迭代2将讨论持久性对单元测试的影响。令人欣慰的是，以前所有单元测试的运行都没有出现大的问题。我们同样也可以发现，新建一个测试持久性的单元测试也是惊人的简单。最后，我们反思了自动单元测试和Groovy的作用。

18.1 迭代1：域模型的持久化

本学习案例使用了第16章最后一次迭代中学习案例的功能性需求，但是需要修改应用程序，以使域模型能够持久性存储在数据库中。为了与第17章的讨论保持一致，本章使用了Cloudscape的关系型DBMS和Spring框架。Cloudscape是一个流行的、基于Java的DBMS，Spring则是能够实现DAO规则的极为优秀的框架。借助于Groovy，我们可以轻松地使用这两个工具。

本次迭代目标是，演示我们能够在数据库中持久性管理应用程序。首先，创建包含两个表的数据库，其中，borrowers表示图书馆的借阅者，publications表示图书馆的出版物。两个表如图18-1所示。每个表都有如membershipNumber和catalogNumber所示的主关键字，它们用来标识不同的条目。在publications表中有一个与该表无关的关键字，如borrowerID，用来联结borrowers表，在将一个publication对象借阅给borrower对象时使用。最后，publications表使用了17.5节所述的方式处理继承关系，也就是说使用一个表来表示类Publication、Book和Journal之间的关系。

请注意，publications表中有两个book对象（type=BOOK）和一个Journal对象（type=JOURNAL）。为保持一致性，范例仅使用大写字母表示每个类型的publication。此外还可以看出，Groovy book对象已经被Jessie借出（borrowerID=1234）。

下面将修改前面我们开发的类。令人欣慰的是，类Publication、Book和Journal在这里并不需要改动。在Borrower类中，我们把用于保存已借阅的Publication对象的映射声明为属性（property），而不是私有特性（attribute）。这将避免需要显式地调用getter和setter方法。类似地，

Library类中所用的两个映射（用于维护borrower和publication集合）被声明为属性。

borrowers

membershipNumber	name
1234	Jessie
...	...

publications

catalogNumber	title	author	editor	type	borrowerID
111	Groovy	Ken Barclay		BOOK	1234
222	UML	John Savage		BOOK	null
333	OOD		Jon Kerridge	JOURNAL	null
...

图18-1 Library应用程序的数据库表

第一个主要改动是，在Library中引入了DAO。和第17.6节所讨论的Bank类一样，也使用DAO处理数据库访问需求。采用DAO设计模式，Library就不需要了解底层数据库的细节。请注意，一个好的设计方案就是把主要功能层面分离开。在这里我们已经分离了业务逻辑，也就是模型和数据库持久性代码。

程序所需要的DAO功能被封装在接口LibraryDaoIF中：

```
interface LibraryDaoIF {
    def abstract getBorrowers()
    def abstract getPublications(borrowers)

    def abstract addPublication(publication)
    def abstract removePublication(publication)

    def abstract registerBorrower(borrower)

    def abstract lendPublication(catalogNumber, membershipNumber)
    def abstract returnPublication(catalogNumber)
}
```

接口仅指定了前面Action类访问和更新Library数据库所需的方法。当然，该数据库仅仅是两个简单的映射而已。现在，它则是功能全面的关系型数据库。

由于程序试图通过JDBC API连接数据库，因此我们将实现接口的类命名为LibraryDaoJdbc。具体实现代码如下所示：

```
import org.springframework.jdbc.object.*
import org.springframework.jdbc.core.*

class LibraryDaoJdbc implements LibraryDaoIF {

    def getBorrowers(){
        def bQuery =new BorrowerQuery(dataSource)
        return bQuery.execute()
    }
}
```

```
}

def getPublications(borrowers){
    def pQuery =new PublicationQuery(dataSource,borrowers)
    return pQuery.execute()
}

def addPublication(publication){
    def params =null
    if(publication instanceof Book)
        params =[publication.catalogNumber,publication.title,publication.author,'','BOOK',null ]
    else
        params =[publication.catalogNumber,publication.title,'',publication.editor,'JOURNAL',null ]
    def pInsert =new PublicationInsert(dataSource)
    pInsert.update(params as Object [])
}

def removePublication(publication){
    def params =[publication.catalogNumber ]
    def pRemove = new PublicationRemove(dataSource)
    pRemove.update(params as Object [])
}

def registerBorrower(borrower){
    def params =[borrower.membershipNumber,borrower.name ]

    def bInsert =new BorrowerInsert(dataSource)
    bInsert.update(params as Object [])
}

def lendPublication(catalogNumber,membershipNumber){
    def params =[membershipNumber,catalogNumber ]

    def pUpdate =new PublicationUpdate(dataSource)
    pUpdate.update(params as Object [])
}

def returnPublication(catalogNumber){
    def params =[null,catalogNumber ]

    def pUpdate =new PublicationUpdate(dataSource)
    pUpdate.update(params as Object [])
}

//----properties -------

def dataSource
}
```

在方法getBorrowers和getPublications的实现过程中，我们引入了类BorrowerQuery和类PublicationQuery。它们都是从Spring框架中引出的MappingSqlQuery类的子类，其行为和第17.6节所讨论的类AccountQuery的行为较为相似。

这两个类的功能都非常强大，它们允许用户查询数据库（在构造器中），并基于查询结果建立域模型（在重新定义的mapRow方法中）。当然，Spring框架负责与数据库通信的细节，代码如下所示：

```
//class:BorrowerQuery
import java.sql.*
import org.springframework.jdbc.object.*

class BorrowerQuery extends MappingSqlQuery {

    def BorrowerQuery(ds){
        super(ds,'select * from borrowers')
        this.compile()
    }

    protected Object mapRow(ResultSet rs,int rowNumber){
        def bor =new Borrower(membershipNumber :rs.getString('membershipNumber'),
                             name :rs.getString('name'))
        return bor
    }
}

//class:PublicationQuery
import java.sql.*
import org.springframework.jdbc.object.*

class PublicationQuery extends MappingSqlQuery {

    def PublicationQuery(ds,borrowers){
        super(ds,'select * from publications')
        this.compile()

        this.borrowers =borrowers           //used by mapRow to update the model
    }

    protected Object mapRow(ResultSet rs,int rowNumber){
        def pub =null

        if(rs.getString('type')=='BOOK')
            pub =new Book(catalogNumber :rs.getString('catalogNumber'),
                           title :rs.getString('title'),author :rs.getString('author'))
        else
            pub =new Journal(catalogNumber :rs.getString('catalogNumber'),
                              title :rs.getString('title'),editor :rs.getString('editor'))
    }
}
```

```
def borID =rs.getString('borrowerID')

if(borID !=null){
    def bor =borrowers [borID ]
    if(bor !=null)
        bor.attachPublication(pub)

}

return pub
}

//----properties ----

def borrowers           //an alias for borrowers in the Library
{
```

请注意，PublicationQuery的方法mapRow关联Publication对象和Borrower对象关联，以及把Borrower对象关联到自身的方式。在这里调用Borrower的attachPublication方法模拟了前面学习案例中所使用的方法。这为工作的可延续性带来了强大的契机。

在此方法中，通过使用membershipNumber（在数据库中为borrowerID），我们像以往一样把borrowers定位到一个映射。不幸的是，由于borrowers不能作为形参，因此必须重新定义Spring中的mapRow方法。这样，在PublicationQuery的构造器中，把borrowers的别名当作属性集看待，目的是让Library在调用DAOgetPublications方法时将它作为一个实参传递。

类BorrowerInsert、PublicationInsert、PublicationUpdate和PublicationRemove都是从Spring框架引入的类SqlUpdate的子类。它们模仿了第17.6节所讨论的AccountInsert类的实现方法。再次强调，这些类非常实用，使用它们可以从域模型中轻易地更新数据库，而不需要关心与数据库通信的具体细节。

```
//class:BorrowerInsert
import java.sql.*
import org.springframework.jdbc.object.*

import org.springframework.jdbc.core.*

class BorrowerInsert extends SqlUpdate {

    def BorrowerInsert(ds){
        super(ds,'insert into borrowers(membershipNumber,name)values(?,?)')
        this.declareParameter(new SqlParameter(Types.VARCHAR))
        this.declareParameter(new SqlParameter(Types.VARCHAR))
        this.compile()
    }
}

//class:PublicationInsert
```

```
import java.sql.*  
import org.springframework.jdbc.object.*  
import org.springframework.jdbc.core.*  
  
class BorrowerInsert extends SqlUpdate {  
  
    def BorrowerInsert(ds){  
        super(ds,'insert into borrowers(membershipNumber,name)values(?,?)')  
        this.declareParameter(new SqlParameter(Types.VARCHAR))  
        this.declareParameter(new SqlParameter(Types.VARCHAR))  
        this.compile()  
    }  
}  
  
//class:PublicationUpdate  
import java.sql.*  
import org.springframework.jdbc.object.*  
import org.springframework.jdbc.core.*  
  
class PublicationUpdate extends SqlUpdate {  
  
    def PublicationUpdate(ds){  
        super(ds,'update publications set borrowerID =?where catalogNumber =?')  
        this.declareParameter(new SqlParameter(Types.VARCHAR))  
        this.declareParameter(new SqlParameter(Types.VARCHAR))  
        this.compile()  
    }  
}  
  
//class:PublicationRemove  
import java.sql.*  
import org.springframework.jdbc.object.*  
import org.springframework.jdbc.core.*  
  
class PublicationRemove extends SqlUpdate {  
  
    def PublicationRemove(ds){  
        super(ds,'delete from publications where catalogNumber =?')  
        this.declareParameter(new SqlParameter(Types.VARCHAR))  
        this.compile()  
    }  
}
```

请注意在PublicationUpdate中，在Borrower借出Publication后，publications表中的borrowerID域已经改变。

令人高兴的是，Library类只需要很少改动，并且容易实现。Library继续负责维护两个映射

中的Borrowers集合和Publications集合（分别被borrowers和loanStock引用）。当然，以前开发的所有错误检查代码在引入数据库后并不需要变动。

第一个变动是，在两个映射发生变化时，Library应该调用合适的DAO，这可以确保在应用程序运行期间，域模型和数据库之间的一致性。第二个变动是，Library的构造器应该初始化DAO，并且使用它来初始化两个映射，这可以确保在应用程序启动时域模型和数据库之间的一致性。Library的大致代码如下所示：

```
class Library {  
  
    def Library(name,dao){  
        this.name =name  
        this.dao =dao  
  
        def bors =dao.getBorrowers()  
        bors.each {bor ->  
            borrowers [bor.membershipNumber ]=bor  
        }  
  
        def pubs =dao.getPublications(borrowers)  
        pubs.each {pub ->  
            loanStock [pub.catalogNumber ]=pub  
        }  
    }  
  
    def addPublication(publication){  
        def message  
        if(loanStock.containsKey(publication.catalogNumber)==false){  
            //  
            //update database  
            dao.addPublication(publication)  
            //  
            //update model  
            loanStock [publication.catalogNumber ]=publication  
            message ='Publication added'  
        }  
        else  
            message ='Cannot add: publication already present'  
  
        return message  
    }  
  
    //As for iteration 4 of Chapter 16  
  
    //...  
  
    //-----properties -----  
  
    def name  
    def loanStock =[ :]
```

```

def borrowers =[ :]
def dao
}

请注意，Groovy和Spring框架的结合使得底层数据库已经非常透明。如：

```

```

def bors = dao.getBorrowers()
bors.each { bor ->
    borrowers[bor.membershipNumber] = bor
}

def pubs = dao.getPublications(borrowers)
pubs.each { pub ->
    loanStock[pub.catalogNumber] = pub
}

```

上述代码非常简洁高效。同样的，

```
dao.addPublication(publication)
```

在addPublication方法中，更新底层数据库的代码已经简单得不能再简单了。

由于功能性需求并没有变化，Action类只需继续完成相同的功能即可。比如，方法displayStock通过指定某个Library对象，生成一个包含图书馆所有出版物的列表。类似地，registerBorrower方法与图书管理员通过基于文本的菜单进行交互。提示管理员和以前一样完整地输入借阅者的会员号和姓名。幸运的是，这意味着不需要对Action类做任何改动。

最后一个任务是，编写运行此应用程序的Groovy脚本。脚本的大部分代码都是负责菜单呈现的控制逻辑，获取用户输入选项，以及调用Action类中适当的方法。这和前一版本没有变化。

但是，此版本和第16章所开发应用程序的一个最大区别是，使用Spring框架所支持的反向控制设计模式（IoC，inversion of control）（参见<http://www.martinfowler.com/articles/injection.html>）。它的目标就是消除类之间可能存在的、会导致程序过于复杂的耦合。

以前的学习案例都是创建并直接组合对象。比如，在创建Library对象和Action对象之后，直接将Library对象加入到Action对象中作为一个集合组件。但是经验告诉我们，当开发更为复杂的应用程序架构时，这种方式是不适合的。幸运的是，Spring提供了一个“轻型容器”（lightweight container），专门用来处理这种交错纷杂的组件。

借助于Spring，类ClassPathXmlApplicationContext被用于创建和组合这些相互关联的对象。它的构造器接收一个配置文件，该配置文件用来定义所需对象，以及描述存在于这些对象之间的相互关系。特别地，我们有如下配置文件config.xml：

```

<?xml version="1.0" encoding=="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
    <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <constructor-arg index="0"><value>org.apache.derby.jdbc.EmbeddedDriver</value></constructor-arg>
        <constructor-arg index="1"><value>jdbc:derby:libraryDB</value></constructor-arg>
        <constructor-arg index="2"><value></value></constructor-arg>
        <constructor-arg index="3"><value></value></constructor-arg>
    
```

```
</bean>
<bean id="libDao" class=="LibraryDaoJdbc">
    <property name="dataSource">
        <ref local="dataSource"/>
    </property>
</bean>

<bean id="lib" class=="Library">
    <constructor-arg index="0"><value>Napier</value></constructor-arg>
    <constructor-arg index="1"><ref local="libDao"/></constructor-arg>
</bean>

<bean id="act" class="Action">
    <property name="library">
        <ref local="lib"/>
    </property>
</bean>
</beans>
```

如第17章所述，在此XML文件中，标签bean用来引入通过id属性标识的bean元素。它的类通过class定义，属性通过property name来定义。属性constructor-arg index用来指定构造器参数。

举例来说，标识为libDao的bean和类LibraryDaoJdbc都有一个dataSource属性，它引用了另外一个标识也为dataSource的本地bean。这就意味着，类LibraryDaoJdbc需要方法setDataSource来初始化其关系属性。

XML同样也告诉用户构造器，是否使用标识为dataSource的对象。有疑问的对象是riverManagerDataSource，它的类是从Spring框架引入的。这个构造器参数允许Library与某个本地Cloudscape数据库通信。

一个需要用户理解的关键点是，所有需要的代码都是自动生成的，因此前面给出的有关类LibraryDaoJdbc的说明都非常简单。举例来说，用户并不需要编写setDataSource方法。这也使得该类的使用更为简单，因为用户不必直接创建和配置 Library Dao Jdbc 及 DriverManagerDataSource对象。

创建Action bean和Library bean的方法相似，它们也都是自动创建和配置的。真正区别在于，对于指定的配置文件，Spring框架允许用户使用关系型数据库。

为了完成主Groovy脚本，我们创建了一个合适的应用程序上下文，并从中获得一个Action对象，所需的代码简洁易懂：

```
def applicationContext = new ClassPathXmlApplicationContext('config.xml')
def action = applicationContext.getBean('act')
```

此次迭代实现的简要代码如Library 01所示。

LIBRARY 01 主脚本

```
import org.springframework.context.support.*
import console.*

def readMenuSelection()
```

```

    println()
    println('0:Quit \n')
    println('1:Add new book')

    //As for iteration 4 of Chapter 16
    //...

    print('\n \tEnter choice>>>')
    return Console.readString()
}

def applicationContext =new ClassPathXmlApplicationContext('config.xml')
def action =applicationContext.getBean('act')

//make first selection
def choice =readMenuSelection()
while(choice !='0'){

    if(choice =='1'){
        action.addBook()           //Add new book
    }
    //As for iteration 4 of Chapter 16
    //...
    else {
        println("\nUnknown selection")
    }
    //next selection
    choice =readMenuSelection()
}
println('\nSystem closing \n')

```

为了结束本次迭代，我们必须演示本次迭代实现以前次迭代拥有的功能。对于使用基于文本的菜单来验证每个用户用例的功能测试，本次迭代已经完全实现。现在，可以重复第16章迭代4中执行的功能测试。令人欣慰的是，它们的测试结果相同。

我们同样也必须演示，对域模型的改变被持久存储在数据库中。这个过程也非常容易实现，只需要选择如下选项来验证域模型的状态即可：

- Display stock (显示库存记录)
- Display publications available for loan (显示可供借阅的publication)
- Display publications on loan (显示已经借出的publication)
- Display all borrowers (显示所有的借阅者信息)

随后请关闭该应用程序，然后重启该程序。和以前一样，可以发现四个显示选项的结果都和原来的结果相同。由于已经实现了其目标，迭代1在此结束。

18.2 迭代2：持久性的影响

在前一个学习案例中，单元测试已经成为我们软件开发的一个主要部分。在迭代2中，我

们将考虑持久性的影响。非常明显地，通过引入数据库，软件已经有了重大变化。因此，我们需要验证前面所有的单元测试是否能够无故障地运行；另外，我们还需要开发与持久性相关的新的单元测试。

对于类Book、Journal和Borrower，它们并没有任何问题。它们各自的GroovyTestCase类，也就是BookTest、JournalTest和BorrowerTest，在执行过程中都不会出现任何问题或者错误。因此，可以把它们的类文件直接加入到runAllTests脚本中。这种方法是可靠的。由于这些类和数据库无关，因而结果和预期会完全一样。

对于类Library来说，由于它和数据库的交互是通过DAO实现的，因此情况就不一样了。另外，它是通过Spring框架创建的，也是Action对象的一部分。在运行前一个Library单元测试之前，必须确认完全清空了数据库中的数据，以避免不同测试之间相互影响。为了实现此功能，我们使用clearAllMethod方法来更新LibraryDaoIF接口。

```
interface LibraryDaoIF {  
  
    // ...  
    def abstract clearAll()  
}
```

然后在LibraryDaoJdbc中实现，实现代码为：

```
// class: LibraryDaoJdbc  
def clearAll() {  
    def pClear = new PublicationsClear(dataSource)  
    pClear.update()  
    def bClear = new BorrowersClear(dataSource)  
    bClear.update()  
}
```

类PublicationsClear和类BorrowersClear的代码为：

```
import java.sql.*  
import org.springframework.jdbc.object.*  
import org.springframework.jdbc.core.*  
  
class BorrowersClear extends SqlUpdate {  
  
    def BorrowersClear(ds) {  
        super(ds, 'delete from borrowers')  
        this.compile()  
    }  
}
```

和

```
import java.sql.*  
import org.springframework.jdbc.object.*  
import org.springframework.jdbc.core.*  
  
class PublicationsClear extends SqlUpdate {
```

```

def PublicationsClear(ds) {
    super(ds,'delete from publications')
    this.compile()
}
}

```

下面，更新前面的LibraryTest类的setUp方法：

```

//class:LibraryTest
void setUp(){
    //Create the Action object
    def applicationContext =new ClassPathXmlApplicationContext ('config.xml')
    action =applicationContext.getBean('act')

    //Clear the model
    action.library.loanStock =[:]
    action.library.borrowers =[:]

    //Clear the database
    action.library.dao.clearAll()

    //As for iteration 4 of Chapter 16
    //...
}

```

请注意，lib属性已经被action属性取代。如其名字所表示的那样，它引用了通过Spring框架创建的Action对象。这样，应该通过它来访问Spring框架创建的Library。举例来说，使用如下代码，来确保每个测试的测试环境都是空白的Library数据库：

```

action.library.loanStock = [:]
action.library.borrowers = [:]

```

因此，Spring框架所做的所有初始化工作都被取消。

类似的，在启动测试之前，也可以使用如下语句来清空图书馆数据库：

```
action.library.dao.clearAll()
```

使用action.library的方法，在测试方法中调用Library，比如：

```

void testAddPublication_1() {
    def pre = action.library.loanStock.size()
    action.library.addPublication(bk1)
    def post = action.library.loanStock.size()

    assertTrue('one less publication than expected', post == pre + 1)
}

```

还有，LibraryTest类并没有改变。令人欣慰的是，所有的测试都顺利通过，这样可以将LibraryTest类文件加入到runAllTests脚本中。这说明类Book、Journal、Borrower和Library并不受向Library引入数据库，以及通过Spring框架构建Library的影响。就这一点来说，可以确信带来的破坏不大。

如果将注意力转移到对使用数据库的这些类进行单元测试方面，LibraryDaoJdbc类是唯一合适的选择，LibraryDaoJdbc类的简要代码为：

```
import groovy.util.GroovyTestCase
import org.springframework.context.support.*
class LibraryDaoJdbcTest extends GroovyTestCase {

    /**
     * Set up the fixture
     */
    void setUp(){
        action =this.getActionObject()

        action.library.loanStock =[:]
        action.library.borrowers =[:]

        action.library.dao.clearAll()
        bk1 =new Book(catalogNumber :'111',title :'Groovy',author :'Ken')
        jol =new Journal(catalogNumber :'333',title :'JOO嫵',editor :'Sally')
        borl =new Borrower(membershipNumber :'1234',name :'Jessie')
    }

    /**
     * Test that the addition of a Book is stored in the database
     */
    void testAddPublication_1(){
        //update the model and the database
        action.library.addPublication(bk1)
        //
        //reset the model
        action.library.loanStock =[:]
        //
        //restore the Action object from the database
        action =this.getActionObject()

        def expected =1
        def actual =action.library.loanStock.size()
        def book =action.library.loanStock [bk1.catalogNumber ]

        assertTrue('unexpected number of publications',actual ==expected)
        assertNotNull('book not present',book)
    }

    /**
     * Test that the addition of a Journal is stored in the database
     */
    void testAddPublication_2(){
        //...
    }
}
```

```
/*
 * Test that the removal of a Publication is stored in the database
 */
void testRemovePublication(){
    //...
}

/*
 * Test that a new Borrower is stored in the database
 */
void testRegisterBorrower(){
    //update the model and the database
    action.library.registerBorrower(bor1)
    //
    //reset the model
    action.library.borrowers =[:]
    //
    //restore the Action object from the database
    action =this.getActionObject()

    def expected =1
    def actual =action.library.borrowers.size()
    def borrower =action.library.borrowers [bor1.membershipNumber ]

    assertTrue('unexpected number of borrowers',actual ==expected)
    assertNotNull('borrower not present',borrower)
}

/*
 * Test that the lending of a Publication to a Borrower is stored in the database
 */
void testLendPublication(){
    //...
}

/*
 * Test that the return of a Publication is stored in the database
 */
void testReturnPublication(){
    //...
}

/*
 * Test that the database tables,borrowers and publications are empty
 */
void testClearAll(){
    //update the model and the database
    action.library.addPublication(bk1)
    action.library.addPublication(jo1)
```

```
    action.library.registerBorrower(bor1)
    //
    //reset the model and the database
    action.library.loanStock =[:]
    action.library.borrowers =[:]
    action.library.dao.clearAll()
    //
    //restore the Action object from the database
    action =this.getActionObject()

    def actual =(action.library.loanStock.size()==0)&&
                (action.library.borrowers.size()==0)
    assertTrue('unexpected Publications or Borrowers',actual ==true)
}

/**
 * Get an Action object composed of a Library with its Borrowers and Publications
 * updated from the database
 */
private getActionObject(){
    def applicationContext =new ClassPathXmlApplicationContext('config.xml')
    return applicationContext.getBean('act')
}

//----properties -------

def action
def bk1
def jol
def bor1
|
```

当将相应的类文件加入到runAllTests脚本中，并执行它时，所有（38个）测试都能通过。虽然今后我们可能会加入更多的测试，但就这一点来说，我们有理由相信数据库的行为和预期一致。所有的代码在本书web站点上列出。

请注意，子类的方法setUp继承自GroovyTestCase类，同时还继承了tearDown方法。tearDown方法在每个测试方法完成后执行。它重新定义了申请或者回收测试所使用主要资源的方法，在其他方面并没有任何作用。关闭数据库连接是它的一个非常明显的作用，但是在[DbUnit](http://dbunit.sourceforge.net)（参见<http://dbunit.sourceforge.net>）非常适合测试数据库应用程序。不管使用何种方法，我们都可以说Cloudscape数据库已经通过了严格的测试。

在结束本次迭代之前，我们将思考单元测试和Groovy之间的关系。对所有的开发者来说，单元测试无疑是一个非常重要的活动。但是，对于Groovy程序员来说，单元测试不仅重要，而且是必需的。这是因为Groovy是一种动态语言，意味着编译器不能自动实现全部的类型检查，这一点与静态语言不同。对于编译正常的Groovy脚本，并不能保证运行时一定没有问题。例如，

可能由于某些对象不能执行特定的方法，或者没有特定的属性，会导致应用程序异常结束。

一个显而易见的解决方法是，在程序正式部署之前，尽可能频繁和严格地运行和测试Groovy代码，这样就有可能及时处理潜在的问题。当然，手工完成这种测试任务是不切实际的，但通过自动化单元测试极有可能实现。举例来说，当执行runAllTests脚本时，我们可以确保这些（38个）测试代码不仅能够编译成功，而且执行时也不会出现任何问题。

最后要强调的一点是，单元测试通常不仅仅只能做编译器所做的检查工作，还能检查对象是否有实际意义。因此现在有着观点认为，相对于传统的编译式程序设计语言，有些动态语言，如Groovy，和单元测试的结合更加实用(参见<http://www.mindview.net/WebLog/log-0025>)。举例来说，编译器可能会提示某个方法应该返回一个字符串，而不是一个int值。Groovy就不会出现这种问题。不管怎么说，编译器都不可能去检查返回的String是否有意义。

我们可以轻易地通过一个单元测试实现此功能，比如，LibraryTest类中有如下代码：

```
// class: LibraryTest
void testRemovePublication_2() {
    action.library.addPublication(bk1)
    def actual = action.library.removePublication(bk1.catalogNumber)
    def expected ='Publication removed'

    assertTrue('unexpected message', actual == expected)
}
```

该测试可以检测String是否能够正确地告知我们已经删除Publication对象，而不仅仅是String是否可用。因此强烈推荐使用单元测试，特别是在Groovy中，实现起来一点都不困难。正相反，还非常有趣。

18.3 习题

1. borrowers有时会要求图书管理员注销自己，因此，软件必须提供注销功能，在Library类中引入一个方法：

```
removeBorrower(membershipNumber)
```

以实现从Library中删除一个Borrower对象。请更新学习案例中最后一个迭代，以增加注销选项。

2. 设计和实现removeBorrower方法的测试过程。比如，测试Borrower在注销时是否有未归还的Publication？是否有与它相关联的Publication对象？同样也应该测试数据库是否已经及时更新。最后还应检查可能错误的用户操作，比如删除一个尚未注册的Borrower对象。
3. 请说明本章以及前些章节中设计方案是减轻还是增加了改动的难度。
4. 请说明Groovy是减轻还是增加了改动的难度。
5. 在学习案例中有一个表示Book类和Journal类的数据库表publications。请说明这种不是面向对象的方案是否实用，另请提出一个替代方法，并给出大致的实现框架。
6. 学习案例中的数据库是本地的，请说明如何使用DAO设计模式访问远程服务器上的数据库？并给出一个大致的实现框架。

第19章 XML构造器和解析器

XML是一种广泛应用于各个应用领域，能够快速构建它本身的技术。像简单的XML标记，也能表示数据及其结构。范例01的输出结果演示了此效果。由于XML具有如此强大的适应能力，因此它也可应用于对象配置（参见第18章和第22章）、GUI架构（参考第20章），以及应用程序的构建和部署等（参考附录K）等领域。

Groovy支持基于树形结构的标记生成器(markup generator)。BuilderSupport能实现各种树形对象的表示方式。一般来说，这些构造器可用来表示XML标记、HTML标记，或者下一章将要用到的Swing用户界面。Groovy的标记生成器可以捕获伪方法的调用，并将它们转换成元素或者树形结构的节点（node）。这些伪方法的参数将被当作节点的属性看待。方法调用的闭包可被视为所生成树节点的嵌套内容。

这些伪方法演示了附录I部分I.5节所描述的元对象协议（MOP，meta-object protocol）。类BuilderSupport是从具体的构造器类派生出来的，它包含了MOP方法invokeMethod的实现。invokeMethod方法能够将这些伪方法转换成结果树的节点。

19.1 Groovy标记

无论使用何种类型的构造器对象，Groovy标记的语法都是相同的。举例来说，请思考下面的代码：

```
staffBuilder = ...           //创建一个构造器对象
staffBuilder.staff(department : 'Computing', campus : 'Merchiston') {
    academic(name : 'Ken Barclay', office : 'C48', telephone : '2745') {
        module(number : 'CO12002', name : 'Software Development 1')
        module(number : 'CO12005', name : 'Software Development 2')
    }
    academic(name : 'John Savage', office : 'C48', telephone : '2746') {
        module(number : 'CO22002', name : 'Software Development 3')
        module(number : 'CO32005', name : 'Design Patterns')
    }
}
```

在这里，树形结构由嵌套的元素表示。这两个academic元素都含有两个截然不同的module元素。staff元素由两个academic元素组成。图19-1为该数据的树形表示法。

我们认为，staffBuilder对象通过两个参数调用了伪方法staff。第一个参数是映射对象[department : 'Computing', campus :

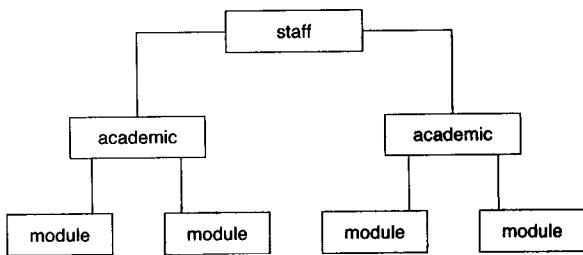


图19-1 构造器的树型表示方式

'Merchiston']，第二个参数是表示嵌套子元素的闭包对象。调用闭包时，此闭包将两次调用academic伪方法。这些伪方法的调用重复了相同的模式。比如说，第一次调用academic方法时，使用的映射参数为：[name : 'Ken Barclay', office : 'C48', telephone : '2745']。

认识到用来表示任意嵌套的标记都是Groovy自身的语法，这一点是非常重要的。由于这是Groovy自身所带的标记，因此同样也可以和其他的Groovy结构混杂在一起，如变量、流程控制，如分支结构，或者方法调用等。在本章中，将讨论构造XML结构的构造器。在后续章节中，将会看到构造其他类型的树形结构的构造器，包括GUI对象。

19.2 MarkupBuilder

我们的第一个范例将演示使用MarkupBuilder构造一个XML文档，以表示具有书名、作者、出版社和ISBN号等信息的图书。伪方法author('Ken Barclay')把参数变成XML元素的内容：

```
<author>Ken Barclay</author>
```

如果此参数作为命名参数出现，那么它将被转化为某个XML元素的一个属性。举例来说，isbn(number : '1234567890')将输出：

```
<isbn number='1234567890' />
```

范例01 第一个范例

```
import groovy.xml.MarkupBuilder

//Create a builder
def mB = new MarkupBuilder()

//Compose the builder
mB.book(){
    author('Ken Barclay')           //producing <author>Ken Barclay</author>
    title('Groovy')
    publisher('Elsevier')
    isbn(number : '1234567890')     //producing <isbn number='1234567890' />
}
```

执行此脚本，其输出结果将直接输出到标准输出流中：

```
<book>
    <author>Ken Barclay</author>
    <title>Groovy</title>
    <publisher>Elsevier</publisher>
    <isbn number='1234567890' />
</book>
```

在这里，MarkupBuilder类被用来构建应用程序。构造器对象mB被伪方法book调用，并创建了一个<book>元素。由于被调用的方法没有参数，因此只是为其内容指定了某个元素。调用闭包所包含的伪方法将输出book内容，如<author>元素所示。

调用伪方法book时使用构造器对象mB，更进一步地，嵌套元素author、title等也都被视为应用到构造器对象mB的伪方法。

由于类MarkupBuilder的默认构造器已经被初始化，因而生成的XML将被发送到标准输出流。我们可以使用参数化的构造器调用，以指定发送XML的目的地文件，如范例02所示。在这里，MarkupBuilder构造器的参数为从File对象中获取的PrintWriter（参见JDK）。输出到文件book.xml的内容和前面提到的XML相同。

范例02 文件输出

```
import groovy.xml.MarkupBuilder
import java.io.*

//Create a builder
def mB = new MarkupBuilder(new File('book.xml').newPrintWriter())

//Compose the builder
mB.book(){
    author('Ken Barclay')           //producing <author>Ken Barclay</author>
    title('Groovy')
    publisher('Elsevier')
    isbn(number : '1234567890')     //producing <isbn number='1234567890' />
}
```

现在，有信心创建更复杂的XML文档。假定某个映射提供了用来填充XML文件的数据。该映射的每个关键字表示书的ISBN号，与之相应的值为包含书籍其他详细资料的列表对象。范例03的代码如下所示。

范例03 含有Book对象的Library范例

```
import groovy.xml.MarkupBuilder
import java.io.*

def data =[ '1111111111' :['Groovy',                      'Ken Barclay', 'Elsevier'],
           '2222222222' :['Object Oriented Design',   'John Savage', 'Elsevier'],
           '3333333333' :['C Programming',             'Ken Barclay', 'Prentice Hall']
         ]

//Create a builder
def mB = new MarkupBuilder(new File('library.xml').newPrintWriter())

//Compose the builder
def lib = mB.library()
data.each {bk ->
    mB.book(){
        title(bk.value[0])
        author(bk.value[1])
        publisher(bk.value[2])
```

```
    isbn(number :bk.key)  
}
```

请注意，在此范例中，必须重申，在mB.book()中一定要使用mB前缀，以分辨出这是更深层次的标记，而不是作为闭包的一部分Groovy脚本。如果没有此修饰符，使用book() { ... } 将会导致Groovy脚本编译出错。程序的输出已经被写入文件中，其内容如下：

```
<library>
  <book>
    <title>Object Oriented Design</title>
    <author>John Savage</author>
    <publisher>Elsevier</publisher>
    <isbn number='2222222222' />
  </book>
  <book>
    <title>C Programming</title>
    <author>Ken Barclay</author>
    <publisher>Prentice Hall</publisher>
    <isbn number='3333333333' />
  </book>
  <book>
    <title>Groovy</title>
    <author>Ken Barclay</author>
    <publisher>Elsevier</publisher>
    <isbn number='1111111111' />
  </book>
</library>
```

请注意，这里的book对象和定义在映射中的book对象的输出顺序并不相同。当然，这是由于映射是无序的关键字/值对的集合。如果需要以相同的顺序输出这些内容，请参见习题5。

19.3 XML解析

Groovy类XmlParser使用了一个简单模型将XML文档解析成树的节点实例（参见GDK文档）。解析器将忽略XML文档中的所有注释和执行指令，并将XML文档中所有元素都转换成节点。每个节点都有XML元素的名称、元素的属性，以及引用的所有子节点信息。这种模型对于大多数简单的XML事务处理来说已经够用了。

使用附录I中引入的对象导航方案能够遍历结果树的所有节点对象。假定doc表示前面所给出的范例<library>中的根节点，那么doc.book将可以选择<library>中的所有<book>元素。<book>列表中的元素将被视为一个表示<book>元素的节点对象列表传递。同样地，doc.book[0]将选择<library>中的第一个<book>元素。在<library>中，<book>元素含有一个<title>元素。然而，由于在某个<book>元素中可能会包含多个<title>元素，同样地，某个<library>元素中也可能会包含多个<book>元素，这样doc.book[0].title[0]将得到第一个<book>元素中的第一个<title>元素。

范例04演示了类XmlParser和XML文档的导航功能。现在请思考：定义在Node类中的text方法是如何获取来自<title>元素中的String值的呢？

范例04 XML的解析和导航功能

```
import groovy.util.*  
  
def parser =new XmlParser()  
def doc =parser.parse('library.xml')  
  
println "${doc.book[0].title[0].text()}"
```

程序的输出结果和期望的结果相同：

Object Oriented Design

由于doc.book传递了一个节点列表，因此我们可以使用迭代器方法和闭包处理<library>中的所有<book>元素。在范例05中，使用each迭代器输出每本图书的书名。

范例05 迭代XML内容

```
import groovy.util.*  
  
def parser =new XmlParser()  
def doc =parser.parse('library.xml')  
  
doc.book.each {bk ->  
    println "${bk.title[0].text()}"  
}
```

再一次地，期望的输出结果如下所示：

Object Oriented Design
C Programming
Groovy

前一个范例可以利用doc.book.title在所有书名上的导航功能。范例06简化了前面的代码：

范例06 通过导航器简化代码

```
import groovy.util.*  
  
def parser =new XmlParser()  
def doc =parser.parse('library.xml')  
  
doc.book.title.each {title ->  
    println "${title.text()}"  
}
```

符号“['@number']”可以用在某个<isbn>元素中，以获取其number属性。下面的XML文件列出教师和所指导学生的成绩。其中，对于老师，只记录名字；对于学生，则记录他们的姓名和成绩。此文件包含如下代码：

```

<staff>
    <lecturer name='Ken Barclay'>
        <student name='David' grade='55' />
        <student name='Angus' grade='75' />
    </lecturer>
    <lecturer name='John Savage'>
        <student name='Jack' grade='60' />
        <student name='Todd' grade='44' />
        <student name='Mary' grade='62' />
    </lecturer>
    <lecturer name='Jessie Kennedy'>
        <student name='Mike' grade='50' />
        <student name='Ruth' grade='70' />
    </lecturer>
</staff>

```

范例07演示了基于不同标准从该文件中选择数据项的效果。

范例07 属性

```

import groovy.util.*

def parser =new XmlParser()
def doc =parser.parse('staff.xml')

println doc.lecturer.student ['@name']

println doc.lecturer.student.findAll {stu ->
    stu ['@grade'].toInteger()>=65
} ['@name']

doc.lecturer.student.each {stu ->
    if(stu ['@grade'].toInteger()>=65)
        println stu ['@name']
}

```

第一个输出语句输出所有学生的姓名列表：

[David, Angus, Jack, Todd, Mary, Mike, Ruth]

第二个输出语句用来获取那些成绩不低于65分的学生列表，对于每个符合条件的学生，输出他们的姓名。

[Angus, Ruth]

上述代码中最后的each迭代器成功地实现与上一范例相同的功能。虽然这次并没有生成列表对象，但那些符合条件的学生姓名被逐行输出：

Angus

Ruth

请思考一下大型数据库应用程序的开发过程。在整个应用程序开发期间，可能创建大量的

相互关联的数据库表，以获取问题域中实体以及这些实体之间存在的关系。使用SQL指令创建这些数据库表，可能会被证实是一个昂贵且浪费时间的方案。在下一个范例中，将演示使用XML文档描述数据库表以及这些数据库表之间存在的关系。从XML文档中将这些关系转换成SQL指令，实在是一件相当简单的任务。

XML文档（在tables.xml文件中）包含如下代码：

```
<?xml version="1.0" encoding="UTF-8"?>

<tables>
    <table name="Book">
        <field name="title" type="text"/>
        <field name="isbn" type="text"/>
        <field name="price" type="integer"/>
        <field name="author" type="id"/>
        <field name="publisher" type="id"/>
    </table>
    <table name="Author">
        <field name="surname" type="text"/>
        <field name="forename" type="text"/>
    </table>
    <table name="Publisher">
        <field name="name" type="text"/>
        <field name="url" type="text"/>
    </table>
</tables>
```

其中，每个`<table>`元素都描述了一个关系型数据库表。`<field>`子元素通过其`name`和`type`表示表的字段。这个简单范例所支持的基本类型包括`text`、`integer`和`id`。`id`类型表示与其他的表之间的关系，并且是另外一个表的外部关键字。范例08给出了处理此信息的方法。

范例08 将XML转换成SQL指令

```
import groovy.util.*

def typeToSQL =[ 'text' : 'TEXT NOT NULL',
                'id'   : 'INTEGER NOT NULL',
                'integer' : 'INTEGER NOT NULL'
            ]
def parser =new XmlParser()
def doc =parser.parse('tables.xml')
doc.table.each {tab ->
    println "DROP TABLE IF EXISTS ${tab ['@name']};" 
    println "CREATE TABLE ${tab ['@name']}(" 
    println " ${tab ['@name']}_ID ${typeToSQL ['id']},"
    tab.field.each {col ->
        println " ${col ['@name']} ${typeToSQL [col ['@type']]},"
    }
    println " PRIMARY KEY ((${tab ['@name']}_ID))"
```

```
    println ");"  
}
```

当我们基于tables.xml中的数据执行此程序时，程序生成在数据库中创建数据库表的SQL指令：

```
DROP TABLE IF EXISTS Book;  
CREATE TABLE Book(  
    Book_ID INTEGER NOT NULL,  
    title TEXT NOT NULL,  
    isbn TEXT NOT NULL,  
    price INTEGER NOT NULL,  
    author INTEGER NOT NULL,  
    publisher INTEGER NOT NULL,  
    PRIMARY KEY (Book_ID)  
);  
DROP TABLE IF EXISTS Author;  
CREATE TABLE Author(  
    Author_ID INTEGER NOT NULL,  
    surname TEXT NOT NULL,  
    forename TEXT NOT NULL,  
    PRIMARY KEY (Author_ID)  
);  
DROP TABLE IF EXISTS Publisher;  
CREATE TABLE Publisher(  
    Publisher_ID INTEGER NOT NULL,  
    name TEXT NOT NULL,  
    url TEXT NOT NULL,  
    PRIMARY KEY (Publisher_ID)  
);
```

把XML解析和基于MarkupBuilder的XML结构导航功能相结合，能为我们提供了一种机制，对某些XML输入实施某种转换，并产生新的输出，新的输出可能是XML形式，或者其他形式。这种类型的转换通常是XSLT（Fitzgerald, 2003; Tidwell, 2001）遗留下来的。不管怎样，大部分的转换在Groovy中都是非常容易实现的。范例09演示了此效果，在这里我们使用了天气文件（所有温度都采用华氏温度表示，所有日期都使用MM/DD/YYYY的形式）：

```
<weather>  
    <temperatures city="Paris">  
        <temperature date="01/21/2001">67</temperature>  
        <temperature date="01/22/2001">70</temperature>  
        <temperature date="01/23/2001">72</temperature>  
        <temperature date="01/24/2001">62</temperature>  
        <temperature date="01/25/2001">65</temperature>  
        <temperature date="01/26/2001">65</temperature>  
        <temperature date="01/27/2001">66</temperature>  
        <temperature date="01/28/2001">78</temperature>  
    </temperatures>  
    <temperatures city="London">  
        <temperature date="01/21/2001">42</temperature>
```

```
<temperature date="01/22/2001">41</temperature>
<temperature date="01/23/2001">45</temperature>
<temperature date="01/24/2001">50</temperature>
<temperature date="01/25/2001">31</temperature>
<temperature date="01/26/2001">40</temperature>
<temperature date="01/27/2001">42</temperature>
<temperature date="01/28/2001">47</temperature>
</temperatures>
<temperatures city="Edinburgh">
<temperature date="01/21/2001">22</temperature>
<temperature date="01/22/2001">24</temperature>
<temperature date="01/23/2001">23</temperature>
<temperature date="01/24/2001">30</temperature>
<temperature date="01/25/2001">12</temperature>
<temperature date="01/26/2001">10</temperature>
<temperature date="01/27/2001">28</temperature>
<temperature date="01/28/2001">22</temperature>
</temperatures>
</weather>
```

生成每个城市的温度表，并且记录每个城市的最低温度。我们的下一个任务，虽然描述起来很简单，但在XSLT中表示却不容易。然而，如果使用Groovy的对象导航功能，这非常容易实现。

范例09 XML转换

```
import groovy.util.*

def parser =new XmlParser()
def doc =parser.parse('weather.xml')
doc.temperatures.each {temps ->
    def lowest =200
    println "City:${temps ['@city']}"

    println'-----+---+
    temps.temperature.each {temp ->
        def tmp =temp.text().toInteger()
        printf('[%10s %2d |n',[temp ['@date'],tmp ])
        if(tmp <lowest)
            lowest =tmp
    }
    println'-----+---+
    println "Lowest recorded temperature is:${lowest}"
    println()
}
```

当我们运行此脚本时，其输出结果如下所示：

```
City:Paris
+-----+--+
|01/21/2001 | 67 |
|01/22/2001 | 70 |
|01/23/2001 | 72 |
|01/24/2001 | 62 |
|01/25/2001 | 65 |
|01/26/2001 | 65 |
|01/27/2001 | 66 |
|01/28/2001 | 78 |
+-----+--+
Lowest recorded temperature is:62
```

```
City:London
+-----+--+
|01/21/2001| 42 |
|01/22/2001| 41 |
|01/23/2001| 45 |
|01/24/2001| 50 |
|01/25/2001| 31 |
|01/26/2001| 40 |
|01/27/2001| 42 |
|01/28/2001| 47 |
+-----+--+
Lowest recorded temperature is:31
```

```
City:Edinburgh
+-----+--+
|01/21/2001| 22 |
|01/22/2001| 24 |
|01/23/2001| 23 |
|01/24/2001| 30 |
|01/25/2001| 12 |
|01/26/2001| 10 |
|01/27/2001| 28 |
|01/28/2001| 22 |
+-----+--+
Lowest recorded temperature is:10
```

有些XSLT转换非常难以表示(参见http://www.oracle.com/technology/pub/articles/wang_xslt.html, <http://www.javaworld.com/javaworld/jw-12-2001/jw-1221-xslt.html>)。对于这些转换来说,还必须求助于非常不简便的XSLT扩展。Groovy支持使用类似于XPath的符号转换复杂的结构,因而实现起来通常要比使用XSLT简单得多。

举例来说,请思考下面用来描述CD目录的XML文档:

```
<catalog>
  <cd>
    <title>Empire Burlesque</title>
```

```
<artist>Bob Dylan</artist>
<country>USA</country>
<company>Columbia</company>
<price>10.90</price>
<year>1985</year>
</cd>
<cd>
    <title>Hide your heart</title>
    <artist>Bonnie Tyler</artist>
    <country>UK</country>
    <company>CBS Records</company>
    <price>9.90</price>
    <year>1988</year>
</cd>
<cd>
    <title>Still got the blues</title>
    <artist>Gary More</artist>
    <country>UK</country>
    <company>Virgin Records</company>
    <price>10.20</price>
    <year>1990</year>
</cd>
<cd>
    <title>This is US</title>
    <artist>Gary Lee</artist>
    <country>UK</country>
    <company>Virgin Records</company>
    <price>12.20</price>
    <year>1990</year>
</cd>
</catalog>
```

我们计划以这些CD的原产地国家为标准分组，然后通过发行年份进一步分组，并发布这些数据。我们寻求的最终形式如：

```
<grouping>
    <country name='UK'>
        <year year='1988'>
            <title>Hide your heart</title>
        </year>
        <year year='1990'>
            <title>Still got the blues</title>
            <title>This is US</title>
        </year>
    </country>
    <country name='USA'>
        <year year='1985'>
            <title>Empire Burlesque</title>
```

```

</year>
</country>
</grouping>
```

开发此范例的目的是，演示我们能够转换XML结构。更进一步地，使用Groovy语言本身支持的列表和映射来完成所需的转换。将此XML转换成以原产地国家为关键字的映射：

```

['UK' : ...,
 'USA' : ...
 ]
```

每个关键字相应的值是另外一个映射，其关键字为发行年份：

```

['UK' : [1988 : ..., 1990 : ...],
 'USA' : [1985 : ...]
 ]
```

最后，内置映射的值是CD名的列表：

```

['UK' : [1988 : ['Hide your heart'], 1990 : ['Still got the blues', 'This is US']],
 'USA' : [1985 : ['Empire Burlesque']]
 ]
```

我们可以轻易地将此结构中的数据转换成所需的XML。范例10是实现这个功能的脚本。范例使用countryGrouping方法实现了XML到嵌套映射之间的转换。

范例10 分组

```

import groovy.util.*
import groovy.xml.*

def countryGrouping(catalog){
    countryMap =[:]

    catalog.cd.each {cd ->
        if(countryMap.containsKey(cd.country [0].text())){
            def yearMap =countryMap [cd.country [0].text()]
            if(yearMap.containsKey(cd.year [0].text())){
                yearMap [cd.year [0].text()]=[cd.title [0 ].text()]
            }else
                .yearMap [cd.year [0 ].text()]=[cd.title [0 ].text()]
            }else {
                countryMap [cd.country [0 ].text()]=[{cd.year [0].text():[cd.title [0].text()]}]
            }
        }

        return countryMap
    }

    def parser = new XmlParser()
    def doc = parser.parse('catalog.xml')
```

```
//Create a builder
def mB = new MarkupBuilder(new File('catalog.countries.xml').newPrintWriter())
def groupings =countryGrouping(doc)
mB.grouping(){
    groupings.each {country,yearMap ->
        mB.country(name :country){
            yearMap.each {year,titleList ->
                mB.year(year :year){
                    titleList.each {title ->
                        mB.title(title)
                    }
                }
            }
        }
    }
}
```

作为最后一个XML文件转换范例，请考虑提供一个有关顾客订单的报表，最初的文件内容可能为：

```
<orderinfo>
    <customer group="exclusive">
        <id>234</id>
        <serviceorders>
            <order>
                <productid>1231</productid>
                <price>100</price>
                <timestamp>2004-06-05:14:40:05</timestamp>
            </order>
            <order>
                <productid>2001</productid>
                <price>20</price>
                <timestamp>2004-06-12:15:00:44</timestamp>
            </order>
        </serviceorders>
    </customer>
    <customer group="regular">
        <id>111</id>
        <serviceorders>
            <order>
                <productid>1001</productid>
                <price>10</price>
                <timestamp>2004-06-07:10:00:56</timestamp>
            </order>
            <order>
                <productid>1231</productid>
                <price>10</price>
                <timestamp>2004-06-01:09:42:15</timestamp>
            </order>
        </serviceorders>
    </customer>
</orderinfo>
```

```
</order>
<order>
<productid>2001</productid>
<price>20</price>
<timestamp>2004-06-16:22:11:19</timestamp>
</order>
</serviceorders>
</customer>
<customer group="regular">
<id>112</id>
<serviceorders/>
</customer>
</orderinfo>
```

在这里，每个顾客都以<id>元素进行标识，如<id>234</id>；每个订单则通过商品标识来表示，如<productid>1231</productid>。一个可能的需求是：在XML的转换过程中，如果某个<timestamp>元素被删除，那么顾客和商品的标识符都要被替换为它们在数据库中的相应名字。再次强调指出，XSLT是不能实现这种转换的。范例11是实现此转换的简洁的Groovy脚本。

范例11 id替换

```
import groovy.sql.*
import groovy.util.*
import groovy.xml.*

def DB = 'jdbc:derby:orderinfoDB'
def USER = ''
def PASSWORD = ''
def DRIVER ='org.apache.derby.jdbc.EmbeddedDriver'

        //Connect to database
def sql = Sql.newInstance(DB,USER,PASSWORD,DRIVER)

def parser = new XmlParser()
def doc = parser.parse('orderinfo.xml')

        //Create a builder
def mB = new MarkupBuilder(new File('orderinfo.details.xml').newPrintWriter())

mB.orderinfo(){
    doc.customer.each {cust ->
        mB.customer(group :cust ['@group']){
            def customer =sql.firstRow('select * from customers'+
                'where id =?',[cust.id [0].text()])
            mB.id(customer.name)
            mB.serviceorders(){
                cust.serviceorders.order.each {order ->
                    mB.order()
```

```
def product =sql.firstRow('select * from products  
    where id =?', [order.productid[0].text()])  
    mB.productid(product.name)  
    mB.price(order.price[0].text())  
}  
}  
}  
}
```

19.4 习题

1. 使用MarkupBuilder实现范例09所示的天气文件。
2. 使用范例04、05和06中的library.xml文件，生成Elsevier出版社所出版图书的图书名和ISBN号列表。
3. 修改范例09，以将XML转换为借阅者能够调整的HTML。
4. 增强范例10的功能，使之能够处理第18章末尾所示的配置文件。
5. 修改范例03，使输出结果按图书ISBN号排序。

第20章 GUI构造器

前一章讲述了使用Groovy标记组织XML结构的方法。图形化应用程序是一些Swing组件的集合，组件可以使用层次方式嵌套在另一个组件中。举例来说，为了构建一个用户交互界面，可能会在面板中嵌套其他的面板。同样地，也可能会在某个用户对话框中加入文本域和按钮。因此，Groovy标记自带的语法也能在Swing应用程序中使用。

20.1 SwingBuilder

Swing框架 (Eckstein et al., 2002; Topley, 1998) 可用来开发图形化应用程序。它是一个大而复杂的库，由超过300个类和接口组成。在开发过程中，软件工程师使用了最前沿的技术，如设计模式 (Gamma et al., 1995; Grand, 2002)，这使其用法变得更为复杂。

借助于类SwingBuilder，使用伪方法就可以表示Swing组件。在大多数情况下，这些伪方法的命名方式都不使用Swing类的前缀“J”，并且规定其首字母必须为小写。使用伪方法frame就能构造一个JFrame部件，使用textField就能构造一个 JTextField部件。伪方法参数用来初始化这些组件，闭包则用来定义子组件。

我们的第一个演示范例是一个显示文本“Hello world”的图形化应用程序。类JLabel的对象表示在Swing中固定的文本元素。类JFrame的对象则表示某个应用程序的顶层窗口。在第一个应用程序中，使用一个封闭的JLabel对象填充JFrame对象。这就使我们立即联想到Groovy的标记frame和label。范例01的代码如下所示：

范例01 第一个框架

```
import groovy.swing.SwingBuilder
import javax.swing.*

//Create a builder
def sB = new SwingBuilder()

//Compose the builder
def frame = sB.frame(title :'Example01',location :[100,100],
    size :[400,300],defaultCloseOperation :WindowConstants.EXIT_ON_CLOSE){
    label(text :'Hello world')
}

//Now show it
frame.pack()
frame.setVisible(true)
```

在这里，使用SwingBuilder类构造这个应用程序。伪方法frame调用构造器对象sB，以创建

一个JFrame对象。方法中命名参数详细说明了标题栏的标题、窗口的左上角位置、窗口的大小，以及当用户单击窗口的关闭按钮时应用程序退出Java运行时等。闭包则只包含一个名为label的伪方法，用来创建包含指定文本的JLabel对象。图20-1显示了应用程序的运行状态。

我们很快就雄心勃勃地在窗口面板中添加了六个组件。标签和文本域组合被用来要求用户提供他们的全名。面板使用了GridLayout管理器以管理这些子组件的位置，将这些子组件按照 3×2 栅格进行组织（每个栅格之间的间距为5个像素点）。代码如范例02所示。

范例02 使用layout管理器

```
import groovy.swing.SwingBuilder
import javax.swing.*
import java.awt.*

//Create a builder
def sB = new SwingBuilder()

//Compose the builder
def frame = sB.frame(title : 'Example02', location : [100,100],
    size : [400,300], defaultCloseOperation : WindowConstants.EXIT_ON_CLOSE){
    panel(layout :new GridLayout(3,2,5,5)){
        label(text :'Last Name:',horizontalAlignment :JLabel.RIGHT)
        textField(text :'',columns :10)
        label(text :'Middle Name:',horizontalAlignment :JLabel.RIGHT)
        textField(text :'',columns :10)
        label(text :'First Name:',horizontalAlignment :JLabel.RIGHT)
        textField(text :'',columns :10)
    }
}

//Now show it
frame.pack()
frame.setVisible(true)
```

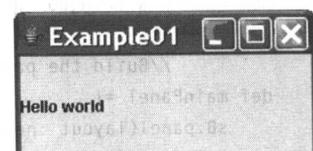


图20-1 第一个框架

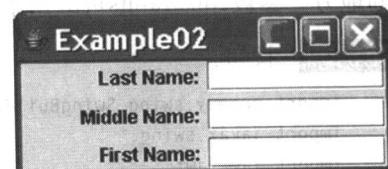


图20-2 GridLayout管理器

图20-2显示了代码的执行结果。请注意，在这里，这六个组件被分成了三行，每行两个组件。每个行都有一个标签和一个文本域。

持续这种风格将会导致深层次的嵌套结构，使得构建和维护这种结构都非常困难。最好的做法通常是构建一个独立的子结构，然后将它们组合成更大的结构。范例03演示了此效果，它重现了前一个演示范例。现在请注意单独定义各个子面板，以及将它们组合成框架的方法。

范例03 递增式组合

```
import groovy.swing.SwingBuilder
import javax.swing.*
import java.awt.*
```

```

//Create a builder
def sB = new SwingBuilder()

    //Build the panel...
def mainPanel ={
    sB.panel(layout :new GridLayout(3,2,5,5)){
        label(text :'Last name:',horizontalAlignment :JLabel.RIGHT)
        textField(text:'',columns :10)
        label(text :'Middle name:',horizontalAlignment :JLabel.RIGHT)
        textField(text:'',columns :10)
        label(text :'First name:',horizontalAlignment :JLabel.RIGHT)
        textField(text:'',columns :10)
    }
}

//...and the frame
def frame =sB.frame(title :'Example03',location :[100,100],
    size :[400,300],defaultCloseOperation :WindowConstants.EXIT_ON_CLOSE){
    mainPanel()
}

//Now show it
frame.pack()
frame.setVisible(true)

```

现在，向应用程序面板添加更多Swing组件将较为容易。范例04引入了两个按钮。请注意主面板使用BorderLayout管理器的方式。BorderLayout管理器利用NORTH、EAST、WEST、SOUTH和CENTER，最多可以安排五个组件的位置。在这种情况下，必须指定子组件应该放置的地方。参数constraints可以用来指定这些组件的位置信息。

范例04 按钮

```

import groovy.swing.SwingBuilder
import javax.swing.*
import java.awt.*

    //Create a builder
def sB = new SwingBuilder()

    //Build the button panel...
def buttonPanel ={
    sB.panel(constraints :BorderLayout.SOUTH){
        button(text :'OK')
        button(text :'Cancel')
    }
}

//...then the main panel...

```

```

def mainPanel ={
    sB.panel(layout :new BorderLayout()){
        label(text :'Is this OK?',horizontalAlignment :JLabel.CENTER,constraints BorderLayout.CENTER)
        buttonPanel()
    }
}

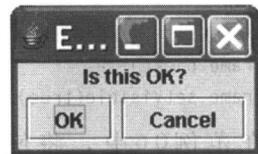
//...and the frame
def frame = sB.frame(title :'Example04',location :[100,100],
    size :[400,300],defaultCloseOperation :WindowConstants.EXIT_ON_CLOSE){
    mainPanel()
}

//Now show it
frame.pack()
frame.setVisible(true)

```

图20-3是范例运行结果。文本处在区域中间（使用默认的FlowLayout），而按钮位置则被SOUTH限定。

图20-3 BorderLayout管理器和按钮



在下一个范例中，将给按钮添加事件处理器（event handler）。事件处理器表示按钮被单击之后将要执行的动作。每个button伪方法都有参数actionPerformed，通常表示为闭包的语句块。在这两个范例中，参数都是一个简单的println语句。举例来说，当单击OK按钮之后，文本“OK press”将出现在控制台中。

范例05 事件处理

```

import groovy.swing.SwingBuilder
import javax.swing.*
import java.awt.*

def sB =new SwingBuilder()

    //Build the button panel...
def buttonPanel ={
    sB.panel(constraints :BorderLayout.SOUTH){
        button(text :'OK',actionPerformed :{
            println 'OK pressed'
        })
        button(text :'Cancel',actionPerformed :{
            println 'Cancel pressed'
        })
    }
}

    //...then the main panel...
def mainPanel ={
    sB.panel(layout :new BorderLayout()){

```

```
label(text : 'Is this OK?',horizontalAlignment : JLabel.CENTER,
      constraints : BorderLayout.CENTER)
buttonPanel()
}
}

//...and the frame
def frame = sB.frame(title : 'Example05',location : [100,100],
size : [400,300],defaultCloseOperation : WindowConstants.EXIT_ON_CLOSE){
mainPanel()
}

//Now show it
frame.pack()
frame.setVisible(true)
```

在范例06中，将使用闭包处理这个事件。将代码转移到处理器中有助于简化actionPerformed参数。在这里，提供了两个处理器闭包，每个闭包处理一个按钮事件。这两个按钮根据列表对象中的数据组合。buttons中的两个子列表中的文本元素用来修饰按钮，闭包对象则充当按钮的事件处理器。

范例06 事件处理器方法

```
import groovy.swing.SwingBuilder
import javax.swing.*
import java.awt.*

//Handlers
def okHandler ={
    println 'OK pressed'
}

def cancelHandler ={
    println 'Cancel pressed'
}

//Buttons
def buttons =[['OK',okHandler],['Cancel',cancelHandler]]

//Create a builder
def sB = new SwingBuilder()

//Build the button panel...
def buttonPanel ={
    sB.panel(constraints : BorderLayout.SOUTH){
        buttons.each {but ->
            sB.button(text :but [0],actionPerformed :but [1])
        }
    }
}
```

```
        }

        //...then the main panel...
def mainPanel ={
    sB.panel(layout :new BorderLayout()){
        label(text :'Is this OK?',horizontalAlignment :JLabel.CENTER,
              constraints :BorderLayout.CENTER)
        buttonPanel()
    }
}

//...and the frame
def frame =sB.frame(title :'Example06',location :[100,100],
                     size :[400,300],defaultCloseOperation :WindowConstants.EXIT_ON_CLOSE){
    mainPanel()
}

//Now show it
frame.pack()
frame.setVisible(true)
```

请注意，buttonPanel使用each迭代器，基于buttons对象（一个元素为列表的列表）来组织按钮。在这里，综合使用Groovy标准代码和构造器标记。each方法使用的闭包可以包含更多的Groovy语句，或者其他构造器标记。对于后者来说，必须引用构造器对象sB以消除它和Groovy代码之间存在的歧义。

在下一个范例中，开发一个简单的应用程序，将某个以英寸表示的测量距离转换成以厘米表示的等价距离。应用程序需要两个文本域和一个按钮。将英寸数输入到一个文本域后，再单击按钮，这样转换后的数据将显示在另一个文本域中。范例07列出代码清单。

范例07 英制长度单位和米制单位之间的转换

```
import groovy.swing.SwingBuilder
import javax.swing.*
import java.awt.*

//Create a builder
def sB = new SwingBuilder()

//properties
def inputText = null
def outputText = null

//Handlers
def doConvert ={
    def text =inputText.getText()
    def inches =text.toInteger()
    def centimetres =2.54 * inches
```

```

        outputText.setText(centimeters.toString())
    }

        //Build the input panel...
def inputPanel ={
    sB.panel(){
        label(text :'Input the length in inches:',horizontalAlignment :JLabel.RIGHT)
        inputText =textField(text :'',columns :10)
    }
}

        //...then the output panel...
def outputPanel ={
    sB.panel(){
        label(text :'Converted length in centimeters:',horizontalAlignment :JLabel.RIGHT)
        outputText =textField(text :'',columns :10,enabled :false)
        button(text :'Convert',actionPerformed :doConvert)
    }
}

        //...and now the main panel
def mainPanel ={
    sB.panel(layout :new GridLayout(2,3,5,5)){
        inputPanel()
        outputPanel()
    }
}

        //...and the frame
def frame =sB.frame(title :'Example07',location :[100,100],
    size :[400,300],defaultCloseOperation :WindowConstants.EXIT_ON_CLOSE){
    mainPanel()
}

        //Now show it
frame.pack()
frame.setVisible(true)

```

请注意，在inputPanel中，变量inputText引用textField的方法。类似地，变量outputText也引用了用来转换值的文本域。button方法的actionPerformed参数调用事件处理器闭包doConvert访问这两个值。这些值是两个JTextField对象，表示数据输入源和数据输出地。闭包doConvert使用getText方法从输入JTextField对象中提取字符串，执行字符到数字的转换工作，并使用setText方法把结果字符串发送给输出JTextField对象。图20-4演示了程序的执行结果。

同样非常值得思考（参见附录B）的是：Java类是怎样封装这些脚本的呢？在脚本中定义的变量，如inputText，是封装类的属性。而在脚本中定义的方法，如doConvert，则是封装类的方法。类的方法仍然能够引用定义在同一个类中的属性。因此，方法doConvert可以引用变量

inputText和outputText。

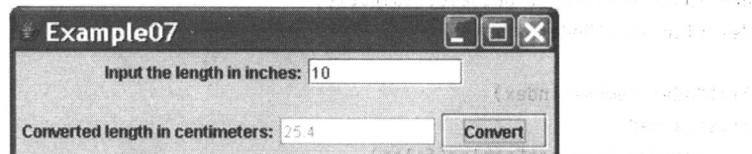


图20-4 英制长度单位和米制单位之间的转换

20.2 列表框和表格

图形化应用程序经常使用列表框和表格显示数据。Swing类JList表示从一组选项中选择一个或者多个项目的组件。列表框中的内容是动态的，可以向列表框添加或者删除列表项。使用JList类时，有两个方面的问题需要注意，一方面是使用一个数据模型表示列表数据，另一方面是使用一个选择模型以确定可以从列表中选定的项目数目。我们向列表添加或者删除项目，实际上是通过底层的数据模型完成的。

下一个范例的演示效果如图20-5所示。JList组件经过 JScrollPane的包装，减小了列表的显示范围。在此情形中，Remove按钮处于激活状态，而Add按钮处于禁用状态。单击Remove按钮后，所选定的项目将从列表中删除。如果在文本域中输入数据，则Add按钮处于可用状态，单击它后，文本域中的内容将被添加到列表中。

范例08给出了相关代码。请注意doRemove和doAdd事件处理器，它们的作用就是从列表中删除选定的项目，或者向列表添加一个项目。这两个操作实际上是通过数据模型，也就是类DefaultListModel的一个对象，进行数据删除或者添加操作的。列表框中的内容最初来自于Groovy列表staffList中的数据。

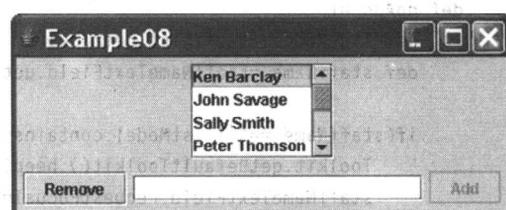


图20-5 列表框

范例08 列表组件

```
import groovy.swing.SwingBuilder
import javax.swing.*
import javax.swing.event.*
import javax.swing.text.*
import java.awt.*

//properties
def staffList = null
def removeButton = null
def staffNameTextField = null

//Event handler for the Remove button
def doRemove = {
    if (staffList != null && staffList.selectedItem) {
        staffList.remove(staffList.selectedIndex)
    }
}

//Event handler for the Add button
def doAdd = {
    if (staffNameTextField != null && staffNameTextField.text) {
        staffList.addElement(staffNameTextField.text)
        staffNameTextField.text = ''
    }
}
```

```
def listModel =staffList.getModel()
def index =staffList.getSelectedIndex()
def size =listModel.size()

listModel.remove(index)
if(size ==0)
    removeButton.setEnabled(false)
else {
    if(index ==listModel.getSize())
        index -

    staffList.setSelectedIndex(index)
    staffList.ensureIndexIsVisible(index)
}

//Event handler for the Add button
def doAdd ={
    def listModel =staffList.getModel()
    def staffName =staffNameTextField.getText()

    if(staffName =="||listModel.contains(staffName)){
        Toolkit.getDefaultToolkit().beep()
        staffNameTextField.requestFocusInWindow()
        staffNameTextField.selectAll()
        return
    }

    def index =staffList.getSelectedIndex()
    index =(index ==-1)?0 :1 +index

    listModel.insertElementAt(staffName,index)

    staffNameTextField.requestFocusInWindow()
    staffNameTextField.setText('')

    staffList.setSelectedIndex(index)
    staffList.ensureIndexIsVisible(index)
}

//-----
//Implementation for an observer to register to receive
//notifications of changes to a text document.
class StaffDocumentListener implements DocumentListener {

    void changedUpdate(DocumentEvent event){
        if(event.document.length <=0)
```

```
        button.setEnabled(false)
    }

    void insertUpdate(DocumentEvent event){
        button.setEnabled(true)
    }

    void removeUpdate(DocumentEvent event){
        if(event.getDocument.length <=0)
            button.setEnabled(false)
    }

//----properties -------

def button

}

//-----
//Specialized DefaultListModel with a parameterized
//constructor
class StaffListModel extends DefaultListModel {

    StaffListModel(list){
        super()
        list.each {item ->this.addElement(item)}
    }
}

//Create a builder
def sB = new SwingBuilder()

//Panel carrying the staff list
def listPanel ={
    sB.panel(constraints :BorderLayout.CENTER){
        scrollPane(){

            def sList =[ 'Ken Barclay',          'John Savage',
                       'Sally Smith',           'Peter Thomson',
                       'John Owens',            'Neil Urquhart',
                       'Jessie Kennedy',         'Jon Kerridge'
            ]
            staffList =list(model :new StaffListModel(sList),
                            selectionMode :ListSelectionModel.SINGLE_SELECTION,
                            selectedIndex :0,visibleRowCount :4)
        }
    }
}
```

```

        }

        //Add/Remove buttons and text field
def buttonPanel ={
    sB.panel(constraints :BorderLayout.SOUTH){
        removeButton =button(text :'Remove',actionPerformed :doRemove)
        def plainDocument =new PlainDocument()
        staffNameTextField =textField(text :'',columns :20,
            document :plainDocument,actionPerformed :doAdd)
        def addButton =button(text :'Add',enabled :false,actionPerformed :doAdd)
        def documentListener =new StaffDocumentListener(button :addButton)
        plainDocument.addDocumentListener(documentListener)
    }
}

//Now the main panel...
def mainPanel ={
    sB.panel(layout :new BorderLayout()){
        listPanel()
        buttonPanel()
    }
}

//...and the frame
def frame =sB.frame(title :'Example08',location :[100,100],
    size :[400,300],defaultCloseOperation :WindowConstants.EXIT_ON_CLOSE){
    mainPanel()
}

//Now show it
frame.pack()
frame.setVisible(true)

```

使用数据模型支持图形化组件是一个常用的模式，通常被称为模型-视图-控制器MVC，Model-View-Controller)框架。其中，模型部分表示为组件提供的数据，视图部分则是组件的可视化外观，而控制则表示用户和组件之间的交互。MVC框架同样也可以用在文本域中，其模型部分是指引用的文档。在这里，通过类PlainDocumnt的一个对象初始化staffNameTextField。将类StaffDocumentListener的一个对象注册到此文档中，以监听文档的改动事件。特别地，当文本域中输入文本后，Add按钮将立即被启用。当清空文本后，Add按钮重新被禁用。

下一个范例将演示JTable组件的用法。JTable通常以二维表格的形式显示数据。和JList组件一样，JTable的操作也依赖其他类的支持。再次强调，在这里也使用了MVC框架。类DefaultTableModel给JTable提供了数据。实际上，为了达到此目的，SwingBuilder对象使用groovy.model.DefaultTableModel的定制版本。在范例09中，可以看到其初始化时使用一个名为staffList的映射列表。JTable同样也需要列标签，在这里列标签由元素closureColumn提供。闭包的列值将被添加到指定的表格模型中。闭包的列值指定列名和从列中访问数据的方式。在这里，闭包只是简单地根据关键字从行中提取映射的值。图20-6演示了范例09的执行结果。

First name	Last name	Room	Tel extension
Ken	Barclay	C48	2745
John	Savage	C48	2746
Sally	Smith	C46	2742
Peter	Thomson	D51	2781
John	Owens	C47	2744
Neil	Urquhart	C66	2655
Jessie	Kennedy	C50	2772
Jon	Kerridge	C36	2777

图20-6 JTable组件

范例09 表格组件

```

import groovy.swing.SwingBuilder
import javax.swing.*
import javax.swing.table.*
import java.awt.*

//Create a builder
def sB = new SwingBuilder()

//Panel carrying the staff list
def tablePanel ={
    sB.panel(constraints :BorderLayout.CENTER){
        scrollPane(){
            table(selectionMode :ListSelectionMode.SINGLE_SELECTION){
                def staffList =[[forename :'Ken',surname :'Barclay',room :'C48',telephone :2745],
                               [forename :'John',surname :'Savage',room :'C48',telephone :2746],
                               [forename :'Sally',surname :'Smith',room :'C46',[telephone :2742],
                               [forename :'Peter',surname :'Thomson',room :'D51',telephone :2781],
                               [forename :'John',surname :'Owens',room :'C47',telephone :2744],
                               [forename :'Neil',surname :'Urquhart',room :'C66',telephone :2655],
                               [forename :'Jessie',surname :'Kennedy',room :'C50',telephone :2772],
                               [forename :'Jon',surname :'Kerridge',room :'C36',telephone :2777]
                               ]

                tableModel(list :staffList){
                    closureColumn(header :'First name',read :{row ->return row.forename})
                    closureColumn(header :'Last name',read :{row ->return row.surname})
                    closureColumn(header :'Room',read :{row ->return row.room})
                    closureColumn(header :'Tel extension',read :{row ->return row.telephone})
                }
            }
        }
    }
}

```

```

        //Now the main panel...
def mainPanel ={
    sB.panel(layout :new BorderLayout()){
        tablePanel()
    }
}
//...and the frame
def frame =sB.frame(title :'Example09',location :[100,100],
    size :[400,300],defaultCloseOperation :WindowConstants.EXIT_ON_CLOSE){
    mainPanel()
}

//Now show it
frame.pack()
frame.setVisible(true)

```

20.3 Box类和BoxLayout类

类BoxLayout是一个管理组件的行或者列位置的布局管理器，特别适合管理带状按钮。类Box则是为BoxLayout管理器准备的一个轻量级容器，为管理盒状布局内组件提供了诸多方便。使用Box创建面板要比使用BoxLayout管理器控制面板布局方便得多。伪方法hbox和vbox分别用来创建水平组件和垂直组件的Box结构。图20-7的垂直Box结构包含带状按钮组件。

在范例10的代码清单中，类FixedButton(FixedTextArea)被用来创建大小固定的定制JButton(JTextArea)。请注意，如何创建这些类的实例，并使用伪方法widget来进行包装。widget方法通常用来创建定制的Swing组件。

范例10 按钮和文本域

```

import groovy.swing.SwingBuilder
import javax.swing.*
import java.awt.*

//Button of set size
class FixedButton extends JButton {
    Dimension getMinimumSize(){return BUTTONSIZE }
    Dimension getMaximumSize(){return BUTTONSIZE }
    Dimension getPreferredSize(){return BUTTONSIZE }

    private static final BUTTONSIZE =new Dimension(80,30)
}

```

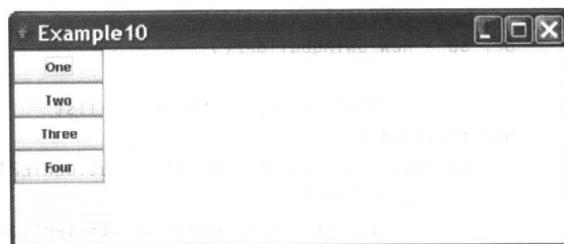


图20-7 带按钮的Box

```
//Text area of set size
class FixedTextArea extends JTextArea {

    Dimension getMinimumSize(){return TEXTAREASIZE }
    Dimension getMaximumSize(){return TEXTAREASIZE }
    Dimension getPreferredSize(){return TEXTAREASIZE }

    private static final TEXTAREASIZE =new Dimension(400,400)
}

//Create a builder
def sB = new SwingBuilder()

//Now the main panel...
def mainPanel ={
    sB.panel(layout :new BorderLayout()){
        vbox(constraints :BorderLayout.WEST){
            def buttons =['One ','Two ','Three ','Four ']
            buttons.each {but ->
                sB.widget(new FixedButton(text :but))
            }
        }
        panel(constraints :BorderLayout.CENTER){
            widget(new FixedTextArea(enabled :false))
        }
    }
}

//...and finally the frame
def frame =sB.frame(title :'Example10 ',location :[100,100],
    size :[400,300],defaultCloseOperation :WindowConstants.EXIT_ON_CLOSE){
    mainPanel()
}

//Now show it
frame.pack()
frame.setVisible(true)
```

此图同样也包含了一个多行的文本域。期望按钮提供一些应用程序功能方法的信息，并将其输出到该区域中。范例中的文本域是不可用的，只是起到显示面板的作用。

附录L讨论使用SwingBuilder创建其他Swing组件的方法，进一步展示利用Groovy的SwingBuilder创建图形化应用程序是非常简单的。

20.4 习题

下面的习题将被视为某个小型项目的用户用例。在做这些习题之前，读者需要学习附录L中有关SwingBuilder类的更多知识。

1. 使用本章和附录L中的知识，开发一个图形化应用程序，并将它作为集成开发环境（IDE）以实现编辑、编译和运行Groovy脚本的功能。IDE界面如图20-8所示。在第一个版本中，只需要在每个事件处理器中使用简单的输出语句，如范例06所示。使用System.exit(0)作为Exit事件处理器的代码（参见附录L中范例01），以成功地关闭应用程序。

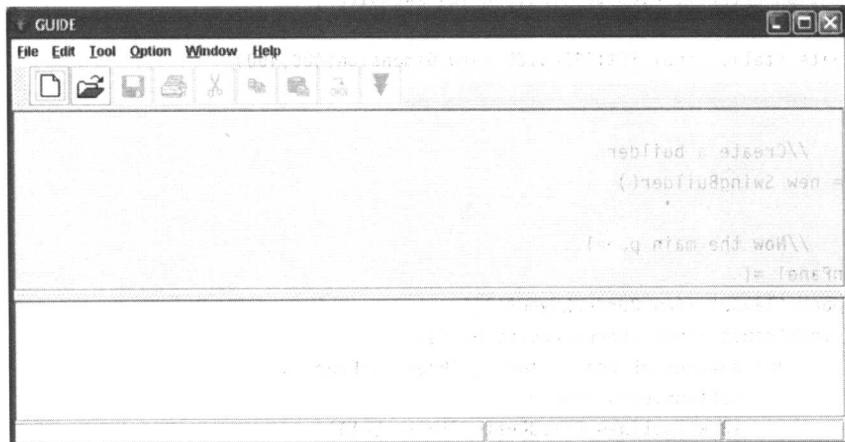


图20-8 最初的IDE

主面板应该被隔离开。面板的上部分作为编辑器面板，设计成有标签的面板，每个标签分别标识等待编辑、编译或者执行的Groovy源文件。面板的下部分则是控制台面板，用来实现运行脚本与用户的交互。

2. 请实现菜单File+New的处理器。用来显示Groovy脚本的文本域应该是一个可滚动的编辑器面板。在刚开始，标签名称分别是Edit1、Edit2等。为编辑窗口添加一个定制的DocumentListener，一旦文本文档的内容被修改，则相应的标签使用一个“*”符号修饰，如*Edit1所示。最终效果图请参考图20-9。

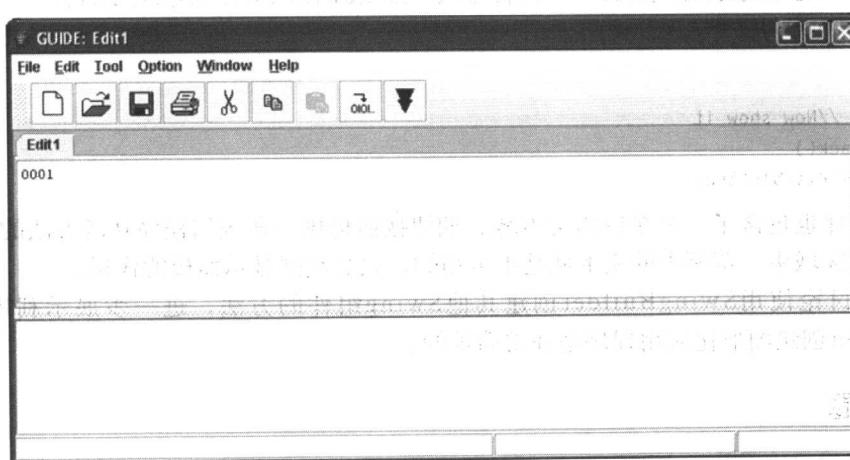


图20-9 编辑器标签

下面实现File+Open的处理器。使用JfileChooser辅助用户选择希望打开的*.groovy文件。文件将在一个可滚动的编辑器窗口中显示，并且窗口的标签名就是文件名。再次声明，结合使用DocumentListener可以新建一个新的文档。

下面继续实现File+SaveAs和File+Save菜单。如果标签以Edit或者*Edit为标签前缀的话，File+Save的处理器将调用File+SaveAs的处理器。再次声明，使用JfileChooser可以帮助用户选择文件名和定位文件的位置。如果标签是Edit或者*Edit的话，该组件将被执行；否则，将使用标签中的文件名更新当前的文件。

3. 假定需要在标题栏中显示应用程序的文件名及其位置信息。给标签窗口添加一个ChangeListener监听器，它的功能是在用户选择不同的标签时更新标题栏，以反映那个被激活的新文件。另外，为标签窗口添加一个MouseListener监听器，它的功能是在用户把鼠标移到某个标签，并单击右键时，弹出一个邀请用户关闭标签及其文件的按钮。
4. 实现Edit + Cut, Edit + Copy 和Edit + Paste菜单项的处理器。
5. 实现类DropTarget的一个专用版本，使得我们能够把*.groovy文件拖放到IDE。

此项目的最终版本在src\guide目录中给出。

第21章 模板引擎

文字处理软件中的邮件合并功能通常用来合并套用信封，包括源自邮件列表中收件人姓名和地址信息。套用信封由静态文本，如信函体，以及那些将要被替代的占位符组成。通常可能包含收件人的姓名和地址信息。数据源则包含每个收信人的占位符的值。此类合并工具最大程度上减轻了用户负担，特别是在数据总量很大，并且由多种元素组成的时候。

Groovy的模板引擎（template engine）和邮件合并功能非常相似，但更具有一般性。本质上来说，它并没有限制表单文档或者被合并数据源的特性。

21.1 字符串

根据第3章的讨论可知，包含在单引号内的字符串对象将按字面值看待，而双引号中的字符串对象则会被解释。因此，下面的注释语句演示了两个print语句的输出结果：

```
def name = "Ken"  
println 'My name is: ${name}'           // My name is: ${name}  
println" My name is: ${name}"          // My name is: Ken
```

双引号字符串的解释功能类似于模板引擎的动作。在这里，表达式中\${name}会被name变量的实际值替代。

21.2 模板

请思考出现在文件book.template中的简单模板。此文件包含描述某本书籍的XML元素的形式：

```
<book>  
    <author>${author}</author>  
    <title>${title}</title>  
    <publisher>${publisher}</publisher>  
    <isbn number="${isbn}" />  
</book>
```

和以往一样，被替换的值使用“\${ }”符号来标识。使用绑定（binding）和SimpleTemplateEngine，可以很容易把占位符映射为实际的值。绑定是一个映射，其中占位符是关键字，以替换值为其值。范例01演示了它的用法。

范例01 映射一个简单模板的值

```
import groovy.text.*  
import java.io.*  
  
def file = new File('book.template')  
def binding = ['author' : 'Ken Barclay',
```

```
'title'          : 'Groovy',
'publisher'      : 'Elsevier',
'isbn'           : '1234567890'
]

def engine =new SimpleTemplateEngine()
def template =engine.createTemplate(file)
def writable =template.make(binding)
println writable
```

当执行此程序时，用值Ken Barclay替换模板中的\${author}。应用程序的最终输出结果如下所示：

```
<book>
<author>Ken Barclay</author>
<title>Groovy</title>
<publisher>Elsevier</publisher>
<isbn number="1234567890"/>
</book>
```

对此范例做稍许改动，以将合并后的结果发送到某个文件。在范例01中，template是类Writable的一个对象。在输出语句中，为了显示此对象的值，显式地调用了toString消息。如范例02所示，同样也可以发送消息writeTo以把结果持久性地保存到某个文件中。

范例02 持久性保存合并后的模板

```
import groovy.text.*
import java.io.*

def file =new File('book.template')
def binding =[ 'author'      : 'Ken Barclay',
              'title'        : 'Groovy',
              'publisher'    : 'Elsevier',
              'isbn'         : '1234567890'
            ]
def writable =new SimpleTemplateEngine().createTemplate(file).make(binding)
def destination =new FileWriter('book.xml')
writable.writeTo(destination)
destination.flush()
destination.close()
```

同样，也可以通过Java服务器端页面（JSP，JavaServer Pages）(Bergsten, 2003)的脚本语法进行调用。可以使用的两个脚本元素为JSP *scriptlets*（使用<%...%>表示）和JSP *expressions*（使用<% = ...%>表示）。scriptlet通常用来添加代码块，包括流程控制语句、输出语句以及变量等。请思考下面的文件library.template：

```
<library>
<%for(bk in books){%>
<book>
<author>$[bk.value [0]]</author>
```

```

<title>${bk.value [1]}</title>
<publisher>${bk.value [2]}</publisher>
<isbn number='${bk.key}' />
</book>
<%|%>
</library>

```

在这里，scriptlet包含一个迭代处理图书集合的循环。它是一个映射集合，对于每个关键字/值条目来说，关键字是图书的ISBN号，其对应的值为依次包含作者、书名和出版社等元素的列表。

为了绑定值，需要提供一个如范例03所示的映射。在这里，请注意将'books'绑定到变量books所引用映射的方法。

范例03 集合合并

```

import groovy.text.*
import java.io.*

def file =new File('library.template')
def books = ['1234567890' :['Ken Barclay','Groovy','Elsevier'],
            '0750660989' :['John Savage','OOD with UML and JAVA','Elsevier'],
            '0130373265' :['Ken Barclay','C Programming','Prentice Hall']
        ]
def writable = new SimpleTemplateEngine().createTemplate(file).make(['books' :books])
println writable

```

当基于这个模板文件执行此程序时，输出结果如下所示：

```

<library>
    <book>
        <author>Ken Barclay</author>
        <title>C Programming</title>
        <publisher>Prentice Hall</publisher>
        <isbn number='0130373265' />
    </book>
    <book>
        <author>John Savage</author>
        <title>OOD with UML and JAVA</title>
        <publisher>Elsevier</publisher>
        <isbn number='0750660989' />
    </book>
    <book>
        <author>Ken Barclay</author>
        <title>Groovy</title>
        <publisher>Elsevier</publisher>
        <isbn number='1234567890' />
    </book>
</library>

```

对前一个范例做简单的修改，以实现与数据库的绑定。booksDB数据库中books表的数据行包含图书ISBN号、作者、书名和出版社等信息。另外，在文件library.html.template中，有一个用来生成HTML的模板：

```
<html>
  <head>
    <title>Library</title>
  </head>
  <body>
    <table border="1">
      <%sql.eachRow('select * from books') {bk ->%>
        <tr>
          <td>${bk.author}</td>
          <td>${bk.title}</td>
          <td>${bk.publisher}</td>
          <td>${bk.isbn}</td>
        </tr>
      <%}>
    </table>
  </body>
</html>
```

修改后的代码为：

范例04 使用数据库实例化的模板

```
import groovy.sql.*
import groovy.text.*

import java.io.*

def DB ='jdbc:derby:booksDB'
def USER =''
def PASSWORD =''
def DRIVER ='org.apache.derby.jdbc.EmbeddedDriver'

//Connect to database
def sql = Sql.newInstance(DB,USER,PASSWORD,DRIVER)

//Create the template
def file = new File('library.html.template')
def writable =new SimpleTemplateEngine().createTemplate(file).make(['sql' :sql])
println writable
```

类Sql没有合适的构造器进行实例化，包括来自JDBC连接的URL、用户名、密码和驱动器类名。在这里，需要使用帮助器方法newInstance。我们得到的输出结果如下所示：

```
<html>
  <head>
```

```
<title>Library</title>
</head>
<body>
<table border="1">
<tr>
<td>Ken Barclay</td>
<td>C Programming</td>
<td>Prentice Hall</td>
<td>0130373265</td>
</tr>
<tr>
<td>John Savage</td>
<td>OOD with UML and JAVA</td>
<td>Elsevier</td>
<td>0750660989</td>
</tr>
<tr>
<td>Ken Barclay</td>
<td>Groovy</td>
<td>Elsevier</td>
<td>1234567890</td>
</tr>
</table>
</body>
</html>
```

21.3 习题

1. 如下面的范例所示，请修改范例01中的代码以注释掉对作者的绑定。现在，请执行该程序，并解释它的作用。

```
def binding = [//'author' : 'Ken Barclay',
```

2. 使用范例01的原始代码，将数据文件修改为下面的范例所示的形式。现在，请执行该程序并解释它的作用。

```
<author>${authorname}</author>
```

第22章 学习案例：图书馆 应用程序（GUI）

本章将使用第20章和附录L的知识，重新编写第18章中图书馆应用程序学习案例。我们的目标是，通过加入一个GUI，使应用程序具有流行的外观。幸运的是，汲取了MVC框架的优势，不需要对类模型做任何改动。类似地，采用DAO也就意味着不需要修改访问数据库所需的类。最后，Groovy的SwingBuilder也使GUI的构建工作变得相对容易。

22.1 迭代1：GUI原型

本次迭代的目标是，使用GUI替换第18章学习案例的基于文本的菜单。我们的意图是，让它真实地反映出第20章中范例10所示的GUI。因此，使用含有合适标签的FixedButtons替换前面学习案例中的菜单选项，并使用事件处理器替换类Action的方法。

模型类（Library、Publication、Book、Journal和Borrower）和DAO的实现（Library-DaoJdbc）在这里都不需要改动，这使得编码任务相对简单。生成GUI的代码如Library01所示。

LIBRARY 01 用户界面

```
import groovy.swing.*  
import groovy.sql.*  
import javax.swing.*  
import java.awt.*  
  
class FixedButton extends JButton {  
  
    Dimension getMaximumSize(){return BUTTONSIZE}  
    Dimension getMinimumSize(){return BUTTONSIZE}  
    Dimension getPreferredSize(){return BUTTONSIZE}  
  
    private static final BUTTONSIZE =new Dimension(250,30)  
}  
  
class FixedTextArea extends JTextArea {  
  
    Dimension getMaximumSize(){return TEXTAREASIZE}  
    Dimension getMinimumSize(){return TEXTAREASIZE}  
    Dimension getPreferredSize(){return TEXTAREASIZE}  
  
    private static final TEXTAREASIZE =new Dimension(600,400)  
}
```

```
//Event handlers
def doExit = {
    System.exit(0)
}

//Create the builder
def sB = new SwingBuilder()

//Create the button panel
def buttonPanel ={
    sB.panel(constraints :BorderLayout.WEST){
        vbox(){
            widget(new FixedButton(text :'Exit'),actionPerformed :doExit )
            widget(new FixedButton(text :'Add new publication'))
            widget(new FixedButton(text :'Display stock'))
            widget(new FixedButton(text :'Display publications available for loan'))
            widget(new FixedButton(text :'Display publications on loan'))
            widget(new FixedButton(text :'Register new borrower'))
            widget(new FixedButton(text :'Display borrowers'))
            widget(new FixedButton(text :'Lend one publication'))
            widget(new FixedButton(text :'Return one publication'))
        }
    }
}

//Create display panel
def displayPanel = {
    sB.panel(constraints :BorderLayout.CENTER){
        widget(new FixedTextArea(enabled :false))
    }
}

//Create status panel
def statusPanel ={
    sB.panel(constraints :BorderLayout.SOUTH){
        label(text :'Status')
        textField(text:'',columns :60,enabled :false)
    }
}

//Assemble main panel
def mainPanel ={
    sB.panel(layout :new BorderLayout()){
        sB.panel(layout :new BorderLayout(),constraints :BorderLayout.CENTER){
            buttonPanel()
            displayPanel()
        }
    }
}
```

```
statusPanel()  
}  
  
//Main frame  
def frame = sB.frame(title : 'Library application', location : [50,50], size : [800,500],  
    defaultCloseOperation :WindowConstants.EXIT_ON_CLOSE){  
    mainPanel()  
}  
  
frame.pack()  
frame.setVisible(true)
```

请注意，通过使用面板来组装各种组件，以及使用BorderLayout管理器来设置这些组件的位置，可以大大简化复杂的GUI构建工作。

执行此脚本时，可以看到，GUI由一组垂直的带状按钮组成，对应到前面学习案例中的简单文本界面的菜单项。窗口右侧的文本区域用来显示程序的输出结果。窗口下部区域是状态面板，用来显示用户信息。图22-1演示这个图形化应用程序的外观。

为完成此次迭代，为Exit按钮设置了一个简单的事件处理器。幸运的是，其行为和预期一致，在下一次迭代中我们将提供更多的事件处理器。

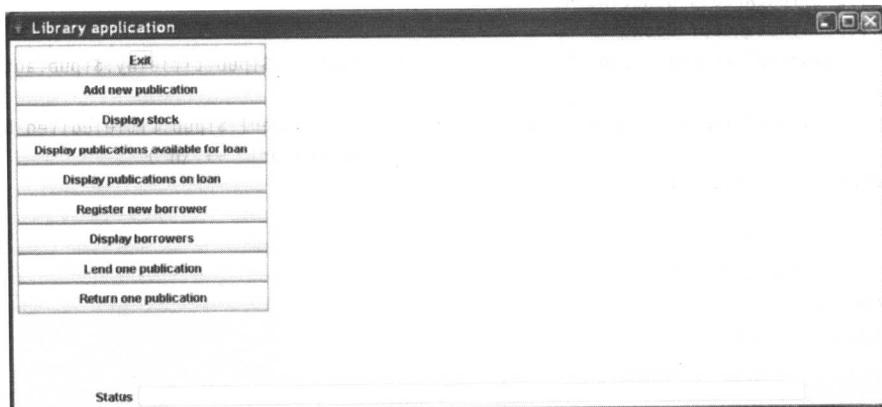


图22-1 可视化界面

22.2 迭代2：处理器的实现

现在，应用程序需要使用事件处理器以提供必需的功能。本次迭代的目的是，在应用程序合适的地方添加大部分的事件处理器，没有添加的事件处理器将作为习题留给读者。同样地，在状态栏中显示来自Library的错误消息，但是把GUI的错误检测的功能实现代码作为习题留给读者（参见22.3节）。

在前一次迭代中，我们为成功地给Exit按钮添加事件处理器而感到高兴。当然，其他按钮也应该实现相应的用户需求。我们期望和前一个中学习案例的Action对象一样，按钮的事件处

理器也能发送消息给Library。类Action的方法逻辑上也就等效地演变成应用程序的事件处理器代码。Library 02的代码如下所示。

LIBRARY 02 事件处理器

```
import groovy.swing.*  
import groovy.sql.*  
import javax.swing.*  
import java.awt.*  
  
import org.springframework.context.support.*  
  
def applicationContext =new ClassPathXmlApplicationContext ('config.xml')  
def library =applicationContext.getBean('lib')  
  
//properties  
def statusTextField = null  
def displayTextArea = null  
  
//helper closure  
  
def displayPublication = (indent,pub ->  
    displayTextArea.append(indent)  
    if(pub instanceof Book)  
        displayTextArea.append("Book:${pub.catalogNumber}:${pub.title}by:${pub.author}"+'\n')  
    else  
        displayTextArea.append("Journal :${pub.catalogNumber}: ${pub.title}edited by:" +  
            "${pub.editor}" +'\n')  
    statusTextField.setText('')  
)  
  
//Event handlers  
def doExit = {  
    System.exit(0)  
}  
  
def doAddNewPublication ={  
    def message  
    def pubType = JOptionPane.showInputDialog(null,'Add a book (B)or journal (J)',  
        'Add new publication',JOptionPane.QUESTION_MESSAGE)  
    if(pubType =='B'||pubType =='b'){  
        def catalogNumber = JOptionPane.showInputDialog(null,'Enter book catalog number',  
            'Add new publication',JOptionPane.QUESTION_MESSAGE)  
        def title = JOptionPane.showInputDialog(null,'Enter book title','Add new publication',  
            JOptionPane.QUESTION_MESSAGE)  
        def author = JOptionPane.showInputDialog(null,'Enter book author','Add new publication',  
            JOptionPane.QUESTION_MESSAGE)  
        message =library.addPublication(new Book(catalogNumber :catalogNumber,title :title,  
            author :author))  
    }  
}
```

```
statusTextField.setText(message)

else if(pubType =='J'||pubType =='j'){
    def catalogNumber = JOptionPane.showInputDialog(null,'Enter journal catalog number',
                                                    'Add new publication',JOptionPane.QUESTION_MESSAGE)
    def title = JOptionPane.showInputDialog(null,'Enter journal title','Add new publication',
                                            JOptionPane.QUESTION_MESSAGE)
    def editor = JOptionPane.showInputDialog(null,'Enter journal editor','Add new publication',
                                              JOptionPane.QUESTION_MESSAGE)
    message =library.addPublication(new Journal(catalogNumber :catalogNumber,title:title,
                                                editor :editor))

    statusTextField.setText(message)

} else
    JOptionPane.showMessageDialog(null,'Incorrect response (B or J)','Add new publication',
                                JOptionPane.ERROR_MESSAGE)
}

def doDisplayStock ={
    def stock =library.loanStock
    displayTextArea.append('Library (full stock inventory):' +'\n')
    displayTextArea.append('======' +'\n')

    stock.each {catNo, pub ->
        displayPublication('',pub)
    }

    statusTextField.setText('')
}

def doDisplayPublicationsAvailableForLoan ={
    def stock =library.loanStock
    displayTextArea.append('Library (publications available for loan):' +'\n')
    displayTextArea.append('======' +'\n')

    stock.each {catNo, pub ->
        if(pub.borrower !=null){
            displayPublication('',pub)
        }
    }

    statusTextField.setText('')
}

def doDisplayPublicationsOnLoan ={
    def stock = library.loanStock
    displayTextArea.append('Library (publications on loan):' +'\n')
```

```
displayTextArea.append('======' +'\n')

stock.each {catNo,pub ->
    if(pub.borrower !=null){
        displayPublication('',pub)
    }
}

statusTextField.setText('')
}

def doRegisterNewBorrower ={
    def message
    def membershipNumber =JOptionPane.showInputDialog(null,
            'Enter borrower membership number','Register new borrower',
            JOptionPane.QUESTION_MESSAGE)
    def name = JOptionPane.showInputDialog(null,'Enter borrower name','Register new borrower',
            JOptionPane.QUESTION_MESSAGE)
    message = library.registerBorrower(new Borrower(membershipNumber:membershipNumber,name :name))

    statusTextField.setText(message)
}

def doDisplayBorrowers ={
    def stock =library.loanStock
    def borrowers =library.borrowers
    displayTextArea.append('Library (all borrowers):' +'\n')
    displayTextArea.append('===== +'\n')

    borrowers.each {memNo,bor ->
        if(bor.membershipNumber != 0){
            displayTextArea.append("Borrower:${bor.membershipNumber};${bor.name}" +'\n')

            def displayed =false
            stock.each {catNo,pub ->
                if(pub.borrower ==bor)
                    displayPublication('',pub)
                    displayed =true
            }
            if(displayed ==false)
                displayTextArea.append(' None')
        }
    }

    statusTextField.setText('')
}

def doLendPublication ={
```

```
def message
def catalogNumber = JOptionPane.showInputDialog(null,'Enter publication catalog number',
                                                'Lend publication',JOptionPane.QUESTION_MESSAGE)
def membershipNumber = JOptionPane.showInputDialog(null,
                                                'Enter borrower membership number','Lend publication',
                                                JOptionPane.QUESTION_MESSAGE)

message =library.lendPublication(catalogNumber,membershipNumber)

statusTextField.setText(message)
}

def doReturnPublication ={
    def message
    def catalogNumber = JOptionPane.showInputDialog(null,'Enter publication catalog number',
                                                    'Return publication',JOptionPane.QUESTION_MESSAGE)

    library.returnPublication(catalogNumber)

    statusTextField.setText(message)
}

//Create the builder
def sB = new SwingBuilder()

//Create the button panel
def buttonPanel ={
    sB.panel(constraints :BorderLayout.WEST){
        vbox(){
            widget(new FixedButton(text :'Exit'),actionPerformed :doExit)
            widget(new FixedButton(text :'Add new publication'),
                  actionPerformed :doAddNewPublication)
            widget(new FixedButton(text :'Display stock'),
                  actionPerformed :doDisplayStock)
            widget(new FixedButton(text :'Display publications available for loan'),
                  actionPerformed :doDisplayPublicationsAvailableForLoan)
            widget(new FixedButton(text :'Display publications on loan'),
                  actionPerformed :doDisplayPublicationsOnLoan)
            widget(new FixedButton(text :'Register new borrower'),
                  actionPerformed :doRegisterNewBorrower)
            widget(new FixedButton(text :'Display borrowers'),
                  actionPerformed :doDisplayBorrowers)
            widget(new FixedButton(text :'Lend one publication'),
                  actionPerformed :doLendPublication)
            widget(new FixedButton(text :'Return one publication'),
                  actionPerformed :doReturnPublication)
        }
    }
}
```

```
        }

    //Create display panel
def displayPanel ={
    sB.panel(constraints :BorderLayout.CENTER)
    sB.scrollPane()
    def displayTextArea =new FixedTextArea(enabled :false)
    sB.widget(new JScrollPane(displayTextArea))
}

//Create status panel
def statusPanel ={
    sB.panel(constraints :BorderLayout.SOUTH){
        label(text :'Status')
        ef statusTextField =textField(text :'',columns :60,enabled :false)
    }
}

//Assemble main panel
def mainPanel ={
    sB.panel(layout :new BorderLayout()){
        sB.panel(layout :new BorderLayout(),constraints :BorderLayout.CENTER){
            buttonPanel()
            displayPanel()
        }
        statusPanel()
    }
}

//Main frame
def frame = sB.frame(title :'Library application',location :[50,50],size :[800,500],
                      defaultCloseOperation :WindowConstants.EXIT_ON_CLOSE){
    mainPanel()
}

frame.pack()

frame.setVisible(true)
```

请注意，在这里我们使用“Add new publication”按钮，而不使用“添加图书”或者“添加期刊”之类的按钮。这只不过是一个更注重实效的做法，可以减少GUI中按钮数量，与底层软件没有关系。不管使用何种方法，都必须修改相关的配置文件，以说明我们不再使用Action对象（使用事件处理器替代）。从文件中删除以下代码即可实现此功能：

```
<bean id="act" class="Action">
```

```

<property name="library">
    <ref local="lib"/>
</property>
</bean>

```

但是，这样做并没有改变第18章中学习案例的逻辑关系。某个典型的输出结果如图22-2所示。

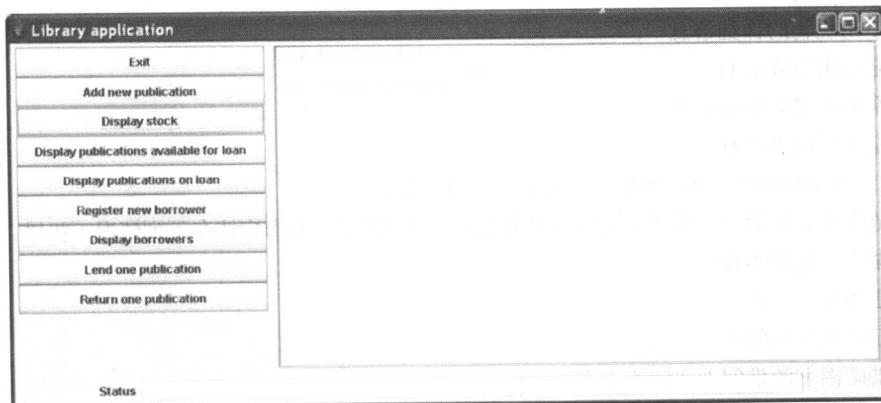


图22-2 含有合适处理器的GUI

和以往一样，最后一个任务也是测试应用程序。用户用例的功能性测试和以前一样，也就是说，通过可视化的方法，比对各种输出结果是否与期望的结果一致。不管使用何种方法，单元测试在这里都会出现少量问题。在前一个学习案例中，使用Action对象访问Library，现在我们则可以直接访问Library（Action对象已经不再存在）。这只需要对类LibraryTest和类LibraryDaoJdbcTest的代码进行小幅修改即可。修改后的代码为：

```

//class:LibraryTest
void setUp(){
    def applicationContext =new ClassPathXmlApplicationContext ('config.xml')
    library =applicationContext.getBean('lib')

    library.loanStock =[:]
    library.borrowers =[:]

    library.dao.clearAll()
    //...
}

```

和

```

//class:LibraryDaoJdbcTest
private getLibraryObject(){
    def applicationContext =new ClassPathXmlApplicationContext ('config.xml')
    return applicationContext.getBean('lib')
}

```

幸运的是，所有测试都能通过，因为我们认为此应用程序已经完成。欲获取Groovy脚本和

类的全部代码，请访问本书的Web站点。

22.3 习题

1. 在第二次迭代中，省略了一些重要的用户用例。举例来说，GUI应该支持以下功能：

- 删除某个出版物
- 显示某个出版物的详细信息
- 显示选定的出版物信息
- 显示某个借阅者的详细信息
- 显示选定的借阅者信息

参照第18章类Action的代码，实现一个或者多个用户用例。

2. 在第二次迭代中，省略了一些重要的错误检测代码。举例来说，GUI应该在下列情况下报错：

- 显示的某个出版物不存在
- 所选的出版物不存在
- 正在显示的某个借阅者不存在
- 所选已借阅图书的借阅者实际不存在

参照第18章类Action的代码，实现一个或者多个错误检测功能。

3. 请思考我们为更新程序所做的努力，并说出Groovy在此上下文中为什么如此优秀的至少四点原因。

4. 在Library 01中，使用如下代码创建按钮面板：

```
def buttonPanel = {
    sB.panel(constraints :BorderLayout.WEST){
        vbox(){
            widget(new FixedButton(text :'Exit'),actionPerformed :doExit)
            widget(new FixedButton(text :'Add new publication'))
            widget(new FixedButton(text :'Display stock'))
            widget(new FixedButton(text :'Display publications available for loan'))
            widget(new FixedButton(text :'Display publications on loan'))
            widget(new FixedButton(text :'Register new borrower'))
            widget(new FixedButton(text :'Display borrowers'))

            widget(new FixedButton(text :'Lend one publication'))
            widget(new FixedButton(text :'Return one publication'))
        }
    }
}
```

虽然这段代码使我们的意图更加明显，但是重复性太强。请使用第29章习题06所示的方法重新编写代码，新方法采用的数据结构是一个以列表为元素的列表，每个列表元素包含某个表示按钮文本的字符串对象和一个作为按钮事件处理器的闭包对象。

第23章 服务器端编程

Java servlet是Java服务器端程序开发的核心技术之一。Servlet是Web服务器的小型可插拔扩展程序，能增强服务器功能。Servlet通常用来创建页面的动态内容，也可以用来创建Web应用程序。许多公司都使用servlet技术重新部署了它们的商用软件。

Servlet是由Java类提供的服务器扩展，可以在Web服务器上动态加载。迄今为止，所有的主流Web服务器都支持servlet。因此，Servlet能够在所有Web服务器上移植，和运行环境一样，应归功于Java在所有操作系统的通用性。

Java服务器端页面（JSP）和servlet联系非常紧密。JSP页面是一种包含JSP元素和静态标记的页面；针对不同请求，JSP元素会产生不同的内容。当请求一个JSP页面时，静态内容将和JSP元素生成的动态内容结合在一起。结果随后返回浏览器。支持JSP的Web服务器首先将JSP页面转换成servlet，这就是翻译阶段。所有的静态内容会保持不变。所有的JSP元素都被转换为提供动态行为的Java代码。

本章我们将讨论Groovy中与Java servlet和JSP等效的Groolet和GSP。同时将谈论Groovy如何简化servlet和JSP的方法。

23.1 Servlets

Servlet是使用Java编写的，因而它们适用于不同的操作系统，而且能在不同的服务器上执行。Servlet包含JAVA API核心的所有特征，如网络和数据库连接。Servlet也同样支持Web应用程序的高效实现。Servlet一旦被加载，就以实例对象的形式存在。服务器的进一步请求实际上就是对这个对象的方法调用。通过它的API类，Servlet代码提供了优雅的面向对象实现。

当客户端连接至服务器，并执行HTTP请求时，它将被视为GET或者POST请求。GET用来获取信息（比如文档和数据库请求等）。POST则用来返回信息（比如信用卡卡号、存储在数据库中信息等）。如图23-1所示。

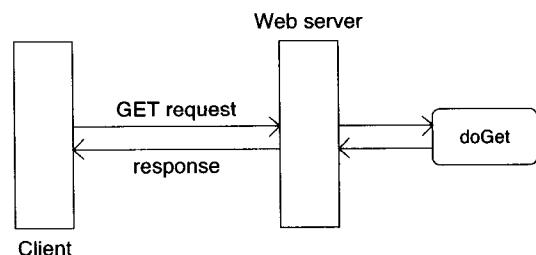


图23-1 使用Servlet处理GET请求

从客户端发出的GET请求将被Web服务器转发给servlet对象。服务器对象将遵循Java接口，重新定义doGet方法。此方法将作为GET请求的处理器。PUT请求也采用类似的策略，由servlet对象的doPut方法处理。

编写一个Java servlet，创建javax.servlet.http.HttpServlet的一个子类，以重载doGet和“/”或者doPut方法（参见图23-1）。这两个方法使用的参数都是HttpServletRequest和HttpServletResponse。如其名所示，它们分别表示客户端请求数据和servlet答复。

23.2 Groovlets

如前面指出的那样，开发servlet需要非常熟练的Java编程技能。但对于Groovy开发者来说，事情就简单多了。Groovlet框架为创建Web应用程序提供了一个优雅而且简化的平台。简化是通过GroovyServlet类实现的。运行Groovy脚本的servlet被称作Groovlets。

通过Groovlet，将不需要HttpServlet子类，也不需要重新定义doGet或者doPut方法。实际上，甚至都不需要开发类。Groovlet仅仅用于转发给客户端的答复。范例01列出了那些供浏览器显示的HTML代码。

范例01 Hello world

```
println """
<html>
    <head>
        <title>Hello world groovlet</title>
        <link rel="stylesheet" type="text/css" href="groovy.css"/>
    </head>
    <body>
        <p class="redarial20">Hello world groovlet</p>
    </body>
</html>
"""
```

此Web应用程序的执行结果如图23-2所示。根据部署描述符文件web.xml，Groovlet框架把所选模式的所有URL映射到指定的servlet。<servlet>元素在它的类中注册servlet名。<servlet-mapping>条目指出GroovyServlet将负责处理所有的*.groovy请求。web.xml文件的部分代码为：

```
<?xml version="1.0"?>

<web-app>

    <servlet>
        <servlet-name>GroovyServlet</servlet-name>
        <servlet-class>groovy.servlet.GroovyServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>GroovyServlet</servlet-name>
        <url-pattern>*.groovy</url-pattern>
    </servlet-mapping>

    ...
</web-app>
```

在Apache Tomcat (Brittain和Darwin, 2003) servlet容器中部署Web应用程序，通常需要编译应用程序代码文件，以及发布Java档案文件、HTML文件等。很明显，这应该是构造工具应该完成的事情，如Ant (Holzner, 2005)。同样，也可能考虑使用附录K.1所述的定制工具AntBuilder。



图23-2 浏览器中的Hello world

我们在前面提到过，Java servlet的doGet和doPut方法用来传递请求和答复消息。Java servlet可以使用的这些对象和其他另外一些对象，Groovlet也可以隐式地使用。在Groovlet中可以使用的变量如表23-1所示。

表23-1 Groovlet的隐性变量

变 量 名	绑 定 到	变 量 名	绑 定 到
request	ServletRequest	session	request.getSession(true)
response	ServletResponse	out	response.getWriter()
context	ServletContext	sout	response.getOutputStream()
application	ServletContext	html	new MarkupBuilder(out)

范例02使用了其中某些变量，其输出如图23-3所示。

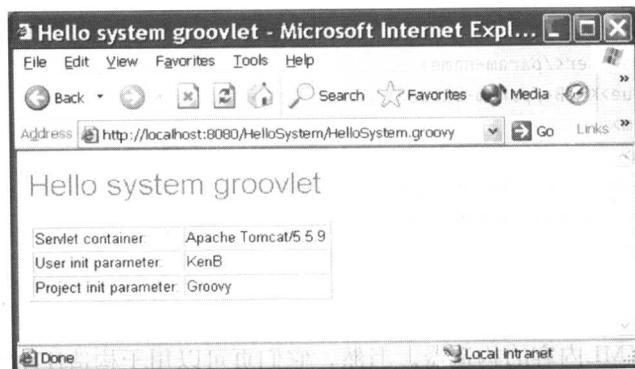


图23-3 信息访问

范例02 隐性变量

```
println ""
<html>
<head>
<title>Hello system groovlet</title>
```

```

<link rel="stylesheet" type="text/css" href="groovy.css"/>
</head>
<body>
<p class="redaria120">Hello system groovlet</p>

<table border="1" class="arial10">
<tr>
<td>Servlet container:</td>
<td>${application.getServerInfo()}</td>
</tr>
<tr>
<td>User init parameter:</td>
<td>${application.getInitParameter("user")}</td>
</tr>
<tr>
<td>Project init parameter:</td>
<td>${application.getInitParameter("project")}</td>
</tr>
</table>
</body>
</html>
"""

```

getInitParameter方法用来获取所谓的Init参数的值。这些初始值都在web.xml文件中指定：

```

<web-app>
    ...
<context-param>
    <param-name>user</param-name>
    <param-value>KenB</param-value>
</context-param>
<context-param>
    <param-name>project</param-name>
    <param-value>Groovy</param-value>
</context-param>
</web-app>

```

本书第19章引入XML内容的构造器。当然，它们也可以用于构造HTML内容。实例03重复了上一个范例演示的结果。请注意，这里使用了预定义的变量html。它是一个MarkupBuilder对象（参见第19章），包含通过out变量发送的，服务器端生成的HTML答复。

范例03 使用构造器生成HTML内容

```

import groovy.xml.MarkupBuilder

html.html(){
    head(){}

```

```
title("Hello system groovlet")
link(rel :"stylesheet",type :"text/css",href :"groovy.css")
|
body(){
    div(class :"redarial20"){
        p("Hello system groovlet")
    }

    table(border :"1",class :"arial10"){
        tr(){
            td("Servlet container:")
            td("${application.getServerInfo()}")
        }
        tr(){
            td("User init parameter:")
            td("${application.getInitParameter('user')}")
        }
        tr(){
            td("Project init parameter:")
            td("${application.getInitParameter('project')}")
        }
    }
}
|
```

下一个应用程序将演示表单数据的处理方法。静态页面Introduction.html是一个请求用户名的表单。action属性是Groovlet文件Hello.groovy。表单页面的代码如下：

```
<html>
<head>
    <title>Introduction</title>
    <link rel="stylesheet" type="text/css" href="groovy.css"/>
</head>
<body>
    <form method="get" action="Hello.groovy">

        <div class="redarial20">Please enter your name<br/></div>
        <table class="arial10">
            <tr>
                <td>Firstname:</td>
                <td><input type="text" name="firstname"/></td>
            </tr>
            <tr>
                <td>Surname:</td>
                <td><input type="text" name="surname"/></td>
            </tr>
            <tr>
                <td><input type="submit"/></td>
```

```

        </tr>
    </table>

</form>
</body>
</html>
```

范例04列出Groovlet代码。

范例04 请求参数

```

import groovy.xml.MarkupBuilder

html.html(){
    head(){
        title("Hello groovlet")
        link(rel :"stylesheet",type :"text/css",href :"groovy.css")
    }
    body(){
        div(class :"redarial20"){
            p("Hello groovlet")
        }

        div(class :"arial10"){
            p("Hello,${request.getParameter('firstname')}${request.getParameter('surname')}")
        }
    }
}
```

在本节的最后一个范例中，将演示Groovlet访问数据库（图23-4）的方法。我们使用标记构造器生成一个用来显示银行账户具体信息的表格。对每个账户来说，列出账户号码和当前余额。其代码如范例05所示。



图23-4 访问数据库

范例05 数据库访问

```
import groovy.xml.MarkupBuilder
```

```
import groovy.sql.*

def DB ="jdbc:derby:C:/Books/groovy/src/Chapter23.Groovlets/Example05/accountDB"
def USER =""
def PASSWORD =""
def DRIVER ="org.apache.derby.jdbc.EmbeddedDriver"

        //Connect to database
def sql = Sql.newInstance(DB,USER,PASSWORD,DRIVER)
html.html(){
    head(){
        title("Account groovlet")
        link(rel :"stylesheet",type :"text/css",href :"groovy.css")
    }
    body(){
        div(class :"redarial20"){
            p("Account groovlet")
        }

        table(border :"1",class :"arial10"){
            sql.eachRow("select * from account"){acc ->
                html.tr(){
                    td("${acc.number}")
                    td("${acc.balance}")
                }
            }
        }
    }
}
```

23.3 GSP页面

JSP页面最初用来显示具体内容，当然，它同时含有静态内容和动态内容。JSP宣称的一个功能是能够实现动态数据和静态数据的分离。但是，由于JSP页面的功能特性，与其说实现分离，不如说加强关联更好。

从表面上看，GroovyServer页面（GSP）和JSP页面非常相似，它们之间的主要区别在于GSP框架是真正意义的模板引擎（参见第21章）。这样，GSP就决不比对应的JSP功能少；从某种意义上来说，甚至更适合静态和动态内容的简单结合。

GSP非常适合表示Web内容。由于GSP框架采用了模板技术，其角色更集中于MVC架构的视图部分。本节后面的范例将混合使用Groovlet和GSP，其中前者更注重业务逻辑，后者负责显示信息内容。

GSP页面是一种常见的Web页面，它含有相互交错的动态内容。GSP允许在其他的静态HTML文件中包含Groovlet代码。每个代码块（通常称为scriptlet）被嵌套在“`<%`”和“`%>`”之间。和Groovlet一样，我们也可以引用servlet对象，比如会话（session）。范例06列出了一段

简单的GSP代码，它三次显示Hello。请注意scriptlet处理循环的方法：

范例06 一个简单的GSP

```
<html>
    <head>
        <title>Hello GSP</title>
        <link rel="stylesheet" type="text/css" href="groovy.css"/>
    </head>
    <body>
        <p class="redarial20">Hello GSP</p>

        <p class="arial10">
            <%3.times(){%>
                Hello
            <%}%>
        </p>
    </body>
</html>
```

浏览器载入此代码后，结果如图23-5所示。



图23-5 一个简单的GSP

下面，重新回到范例04上来。在这里，HTML表单接收用户名，然后值将输出到另一个页面，通过设置表单的action元素的方式引用Groovlet实现此功能：

```
def dispatcher = request.getRequestDispatcher("Hello.gsp")
dispatcher.forward(request, response)
```

在这里，Groovlet只是发送请求给GSP，GSP再负责处理结果的显示。Groovlet将请求参数传递给GSP，其代码如范例07所示。

范例07 GSP中的请求参数

```
<html>
    <head>
        <title>Hello GSP</title>
```

```
<link rel="stylesheet" type="text/css" href="groovy.css"/>
</head>
<body>
    <p class="redarial20">Hello GSP</p>

    <p class="arial10">
        Hello <%print "${request.getParameter('firstname')}";%>
        print "${request.getParameter('surname')}" %>
    </p>
</body>
</html>
```

程序执行结果如图23-6所示。

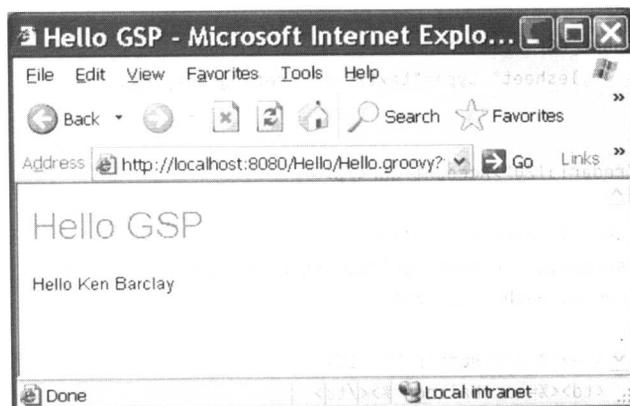


图23-6 请求参数

最后一个范例演示了Groovlet负责应用程序处理，GSP只负责显示数据。在此使用银行账户数据库。Groovlet负责打开和访问account表。在代码中，使用setAttribute方法将account数据集绑定到请求对象。这实际上也就是在姓名和account对象之间的绑定。

```
import groovy.xml.MarkupBuilder
import groovy.sql.*

//Forwards to specified page
def forward(page,req,res){
    def dispatcher = req.getRequestDispatcher(page)
    dispatcher.forward(req,res)
}

def DB ="jdbc:derby:C:/Books/groovy/src/Chapter23.Groovlets/Example08/accountDB"
def USER =""
def PASSWORD =""
def DRIVER ="org.apache.derby.jdbc.EmbeddedDriver"

//Connect to database
```

```

def sql = Sql.newInstance(DB,USER,PASSWORD,DRIVER)
def accounts =sql.dataSet("account")

request.setAttribute("accounts",accounts)

forward("AccountDB.gsp",request,response)

```

Groovlet将继续发送请求给GSP，GSP访问表中accounts对象，使用迭代器和闭包在表格中显示每个账户的号码和当前余额。请注意表达式中的“`<%=>`”和“`%>`”标记。程序将计算这些标记之间表达式的值，计算结果将作为页面的一部分。

范例08 使用GSP显示数据库表

```

<html>
    <head>
        <title>Account GSP</title>
        <link rel="stylesheet" type="text/css" href="groovy.css"/>
    </head>
    <body>
        <p class="redarial20">Account GSP</p>

        <table border="1" class="arial10">
            <%def accounts = request.getAttribute('accounts')>
            accounts.each {acc ->%
                <tr>
                    <td><%=acc.number %></td>
                    <td><%=acc.balance %></td>
                </tr>
            <%}%>
        </table>
    </body>
</html>

```

图23-7是GSP页面的屏幕截图。



图23-7 数据库视图

23.4 习题

1. 使用范例03所示的类MarkupBuilder重新实现范例02的功能。
2. 如范例01所示，请仅使用Groovlet重新实现范例07的功能。
3. 增强范例04的功能，以检查用户输入的姓名是否为空。如果为空，返回输入表单，并向用户提供适当的错误提示信息。
4. 编写一个简单的GSP，并给用户显示适当的欢迎词，如“早上好”、“下午好”或者“晚上好”。

第24章 学习案例：图书馆 应用程序（WEB）

本章学习案例是图书馆应用程序学习案例开发过程中，思考的最后一个问題。在这里，使用Web浏览器界面替换第22章所开发的图形用户界面（GUI），使应用程序变成了一个Web应用程序。根据前面章节所学到的知识，将综合使用Groovlet和GSP技术。

如学习案例的早期版本所述，将使用MVC框架分离逻辑关系。在第13章中，我们采用了此框架，成功地实现基于文本的用户界面类与域模型类的业务逻辑相分离。第22章沿用了这种结构，这就意味着使用Swing类替换基于文本的用户界面类，以生成应用程序GUI，是一个相对简单的任务。

以往经历充分证明，建立一个MVC框架，我们能够轻易地替代新的用户界面。GSP非常适合显示页面内容。由于GSP框架是一种模板技术，其任务集中于MVC架构的视图部分。Groovlet实现应用程序所需的控制器逻辑，比如，添加一个新的出版物，或者允许某个借阅者借阅出版物。最后，仍旧不改动域模型，使用数据访问对象与数据库进行交互。

24.1 迭代1：Web实现

此应用程序的图形化界面将真实地再现我们在第20章所开发的界面。其可视化效果如图24-1所示。它由一组垂直的带状按钮组成，每个按钮的左侧都有一个解释型标签。

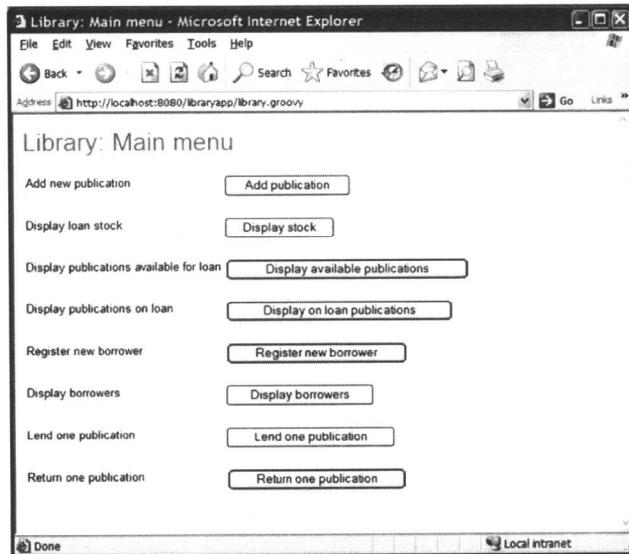


图24-1 可视化界面

如第23章范例08所示，Groovlet被用来启动应用程序。我们仍旧使用Spring框架来配置应用程序bean，随后把lib变量引用的Library对象绑定到'lib'上，这样，就可在应用程序的任何地方引用此对象。实现此功能的代码位于library.groovy中，如下所示：

```
import java.io.*  
import org.springframework.context.support.*  
  
def applicationContext = new FileSystemXmlApplicationContext(context.getRealPath('/') + 'config.xml')  
def lib = applicationContext.getBean('lib')  
  
application.setAttribute('lib', lib)  
  
Utility.forward('mainmenu.gsp', request, response)
```

由于第23章范例08引入的forward方法已经成为类Utility的静态方法，因此我们在应用程序的任何地方使用这个方法。

GSP mainmenu.gsp呈现的菜单如图24-1所示，它使用了一个含有两个列的HTML表格，其左侧的列是用户用例的注释，右边的列是选定此功能的按钮。部分代码如下所示：

```
<html>  
  <head>  
    <title>Library:Main menu</title>  
    <link rel="stylesheet" type="text/css" href="groovy.css"/>  
  </head>  
  
  <body>  
    <p class="redarial20">Library:Main menu</p>  
  
    <table>  
  
      <tr>  
        <td class="arial10" valign="top">Add new publication</td>  
        <td>  
          <form action="addpublication.gsp">  
            <input type="submit" value="Add publication"/>  
          </form>  
        </td>  
      </tr>  
  
      <tr>  
        <td class="arial10" valign="top">Display loan stock</td>  
        <td>  
          <form action="displaystock.gsp">  
            <input type="submit" value="Display stock"/>  
          </form>  
        </td>  
      </tr>  
  
    //...  
    </table>  
  </body>  
</html>
```

为显示library的所有库存信息，这个用户用例的动作将调用GSP displaystock.gsp。输出结果类似于图24-2。在Groovlet启动时，GSP引用了Library对象集，随后将使用each迭代器迭代处理loanStock的每个出版物。与之相关的闭包则负责处理表中数据行，并有选择地输出图书或者杂志。

Catalog number	Title	Author	Editor
JOURNAL	333	OOD	JonK
BOOK	111	Groovy	KenB
BOOK	222	UML	JohnS

图24-2 显示图书馆数据库的内容

本项目的许多其他用户用例将由GSP和相应的Groovlet实现。举例来说，归还某个出版物时，需要提供此出版物的分类号，可以通过下面的GSP实现：

```
<html>
  <head>
    <title>Library:Return one publication</title>
    <link rel="stylesheet" type="text/css" href="groovy.css" />
  </head>

  <body>
    <p class="redarial20">Library:Return one publication</p>

    <form action="returnpublication.groovy">
      <table>
        <tr>
          <td class="arial10">Catalog number:</td>
          <td><input type="text" name="catalognumber"/></td>
        </tr>
      </table>
      <input type="submit" value="Submit" />
    </form>
  </body>
</html>
```

用户输入的分类号随即将被转换成实现此功能的Groovlet请求参数：

```
def lib = application.getAttribute('lib')

def catalogNumber request.getParameter('catalognumber')
lib.returnPublication(catalogNumber)

Utility.forward('mainmenu.gsp', request, response)
```

24.2 习题

1. 请思考mainmenu.gsp提供的主菜单。所有的应用程序用户用例都在一个表格中显示，其中左侧的列描述用户用例，右侧的列则是激活相应功能的按钮。请思考使用MakrupBuilder组织HTML的方法，并重新编写代码实现此功能。

第25章 后记

本章将是我们Groovy编程之旅的最后一站。我们坚信这也预示着Groovy时代即将来临，Groovy必将为Java平台做出巨大贡献。我们期望看到Groovy用户群茁壮成长，就像Java当初宣称的那样。Groovy是一种灵活的、敏捷的脚本语言，是Java语言的有益补充。

有人认为，Groovy是一种不成熟的技术，这种说法都只是部分正确。不可否认，目前来说，参考实现（JSP06）有很多地方需要纠正。尽管如此，它的健壮性依然是大家对官方发布版本的共识。全面支持Groovy技术的IDE构造器正在完善中（参见<http://groovy.codehaus.org/Eclipse+Plugin>，<http://groovy.codehaus.org/IntelliJ+IDEA+Plugin>）。

如果觉得Groovy的大部分功能都起源于Java API扩展集合，就认为Groovy不成熟的话，我认为是不合适的。Java也是一门小型的、简单的编程语言，也必须依靠API增强其功能，这一点与Groovy有什么差别呢？Groovy提供一种敏捷的环境，能够充分利用已有的代码，从这一点来讲，Groovy更胜一筹。Groovy和Java两者是相互补充的关系，其中，Java是开发框架和架构的系统编程语言，Groovy则将不同架构组合到应用程序中。

随着Groovy不断发展，毋庸置疑，现有产品和为Groovy开发的新产品会进一步整合。在本书中，我们使用了Spring框架的部分内容。我们期望Groovy在其他的某些地方也能发现Spring的踪影，如单元测试、bean定义，甚至是利用Groovy模板的SpringMVC视图中。一个名为Grail（参见<http://grails.codehaus.org/>）独立开发项目致力于在Rails平台上再现Ruby的成功（参见<http://www.rubyonrails.org/>）。Rails是一个开源的Ruby框架，用来开发数据库支持的Web应用。

Groovy可能做出巨大贡献的另一个领域是Web服务(Erl, 2004; Topley, 2003)。除了能够简化对服务的访问之外，Groovy还能实现Web服务使用的业务过程执行语言(BPEL, Business Process Execution Language)（参见<http://www-128.ibm.com/developerworks/webservices/library/ws-bpelwp/>）。BPEL大量使用XML，同时Groovy能够完美支持XML。更进一步来说，像有人所说的那样（参见<http://www.martinfowler.com/articles/languageWorkbench.html>），Groovy的构造器也可能是BPEL的域描述语言(DSL, domain-specific language)的基石。

Groovy也为我们提供了新的学习机会。计算机和软件工程专业的学生应该会发现Groovy的诱人之处，这是因为Groovy使编写类似于Java应用软件的起点变得很低，而不需要陷于复杂问题的泥潭中。尽管这些学生以后几乎都不可避免地会使用Java，但是他们应该能发现，在今后的职业生涯中，Groovy的粘合能力对大部分费事的应用程序来说都将非常实用。

Groovy同样也能够帮助基础理论研究学者潜心于软件开发研究工作。Groovy所能触及的地方都将会发出耀眼的光芒，即使在错综复杂的Java编程过程中也不会黯然失色。

让我们从Groovy开始吧！

附录A 软件发布

本书使用的素材都具有不同的开源许可，其二进制文件和源代码可以免费获得，同时这些软件在软件开发团体的不断更新下得到了飞速的发展。这些素材的质量都非常之高，而且用户可以免费使用。

本书的案例都基于MS Windows平台开发，其相关描述信息将在这里给出，因而适合于Windows开发环境。不管使用何种方法，讨论的大部分话题都具有一般性，并且能快速地移植到其他平台。

下面的章节都非常简短，并且可能会经常变动。无论如何，我建议读者经常浏览本书的Web站点和这些软件的Web站点的内容，以获取更多的知识，并了解最新的信息。

A.1 Java开发工具

Java开发工具（Java Development Kit，JDK）可以从Web站点<http://java.sun.com/j2se/1.5.0/download.jsp>获得。它是一个压缩包文件，能在不同的平台上使用。在此Web站点上，还可以下载一个包含技术文档的ZIP文件。在MS Windows中，请双击可执行文件（比方说，`jdk-1_5_04-windowsi586-p.exe`）启动解压缩过程。JDK的默认路径是C:\Program Files\文件夹。文件名中的间隔通常是一些更新小bug后的源文件，在为目的文件夹命名时，这种方式非常值得推荐（比方说，`C:\jdk-1_5_04`）。

将ZIP文档解压缩到同一个路径中，使JDK的所有文件都在同一个目录下。

A.2 Groovy开发工具

Groovy开发工具（GDK，Groovy Development Kit）可以从<http://groovy.codehaus.org/>站点获得。它以ZIP文件的形式下载，是一个包含工具和文档在内的完整包。请在合适的路径下解压缩此包。再次强调，请注意文件夹名中嵌入的间隔符。分发的GDK含有大量的子文件夹，如bin（包含可执行文件、批处理文件等等）和包含Groovy JAR文件的lib文件夹。

A.3 ANT

Apache ANT是一个基于Java的构造工具，可在<http://ant.apache.org/bindownload.cgi>站点获得。它以ZIP文件的形式下载，是一个包含工具和文档在内的完整包。请在合适的路径下解压缩此包，再次强调，请注意文件夹名中嵌入的间隔符。分发的Apache ANT含有大量的子文件夹，如bin（包含可执行文件、批处理文件等等）和包含JAR文件的lib文件夹。

分发的docs文件夹包含index.html欢迎页面。从这里开始，遵循手册的链接，可以了解如何安装、使用和运行Ant；其中还包含不同Ant任务的说明文档。

A.4 Derby/Cloudscape数据库

Cloudscape数据库最初由IBM开发，现在已经捐献给开源协会，并将它重新命名为Derby，可以从<http://db.apache.org/derby/>站点获得。请在合适的路径下解压缩此包。分发的Derby数据库包含的lib文件夹，此文件夹含有它支持的JAR文件。

doc/pdf文件夹包含大量的文档文件。“getting started”文件对那些不熟悉DBMS工具的用户来说是没有价值的。开发者向导将讨论Cloudscape的安装和部署方法，参考文档和工具文档则说明SQL和Cloudscape工具的使用方法，比如， JDBC脚本交互工具ij。

A.5 Spring框架

Spring框架可以从Web站点<http://www.springframework.org/>获得。它是一个完整的ZIP文件，包含相关的支持文档。请在合适的路径下解压缩此包。分发的Spring框架包含的dist文件夹，此文件夹含有它支持的JAR文件。

docs/reference文件夹含有大量的Spring参考书籍，以PDF文档的形式发布。docs/api文件夹包含Spring提供的类文件参考文档。

A.6 Tomcat服务器

Tomcat服务器可以从Web站点<http://tomcat.apache.org/>获得，提供很多常见的格式。请下载二进制版本，并将它解压缩到合适的目录下。分发的Tomcat服务器包含bin文件夹，bin文件夹中含有可执行文件、批处理文件等。提供的文档（在webs\tomcat-dpcs目录下）包含对批处理文件bin\startup.bat和bin\shutdown.bat（启动和停止Tomcat服务器）的说明。webapps文件夹是我们部署Web应用的文件夹，其子文件夹tomcat-docs包含Tomcat用户指南，覆盖如何安装和部署服务器的信息。

A.7 Eclipse IDE

Eclipse IDE是一个开源的集成开发环境（参见<http://www.eclipse.org/>）。基于插件的独特架构使它能够轻易地创建、集成和使用软件工具。Eclipse IDE的发展势头迅猛，现在已经有非常广泛的工具集。本书使用的类图都是使用Omondo UML工具（参见<http://www.eclipseplugins.info/eclipse/plugins.jsp>；遵循下面的分类和UML链接）开发的。Groovy插件现在已经能在Eclipse IDE中使用（参见<http://groovy.codehaus.org/Eclipse+Plugin>）。

A.8 本书源文件

本书的Web站点提供了所有范例代码和习题答案。范例代码在src文件夹中，习题答案在solutions文件夹中。在这两个文件夹中，代码文件都位于以这些文件所属章节和附录名为目录名的子目录中。如，src文件夹中的子目录AppendixI.Classes包含附录I中*.groovy范例代码文件。solutions文件夹中子目录Chapter12.Classes则包含第12章中*.groovy习题答案。src文件夹的结构如下所示：

```
src
  AppendixB.Groovy
  AppendixD.Strings
  ...
  Chapter07.Methods
    example01.groovy
    example02.groovy
  ...
  Chapter08.Control.Flow
  ...
  guide
  lib
  utils
  setgroovy.bat
```

在所有的案例中，表示类声明的文件遵循Java命名惯例。这样，包含类Account的文件可以在Account.groovy中找到。那些表示脚本的文件名则为example01.groovy，或者exercise02.groovy。

在src文件夹中，utils子目录包含类Console（第5章和附录F）、Build（附录K）和Functor（附录J）的源代码。可以使用Ant构造文件并编译此代码，随后创建utils.jar，并把它部署到src\lib子目录中。guide子目录中包含所有的GUIDE工具源文件。再次强调，提供的Ant构造文件用来编译和代码存档。

src文件夹同样也包含setgroovy.bat文件。此MS Windows批处理文件用来设置Groovy在此平台上的运行环境。当然，它需要编辑以反映读者的本地设置。使用和模板类似的配置文件可以将代码移植到其他的平台。

新增功能会定期定期发布到这个Web站点，因此建议读者经常浏览这个网站，以了解最新的信息。

附录B Groovy简介

Groovy语言的语法是基于Java编程语言的，这有助于缩短Java开发者的学习时间。Groovy语言使得编写Java平台的脚本和应用程序变得既快又容易。它包含很多存在于Python、Ruby和Smalltalk的语言特性。但是，其语法对于Java开发者来说，显得更为自然。由于Groovy是一种基于Java的语言，因而使用Groovy语言编写的应用程序完全可以使用Java API，这样Groovy就可以与使用Java语言编写的框架和组件实现无缝结合。

本附录将演示如何使用Groovy实现Java能够实现的各种特征。我们希望演示Groovy开发者如何使用更优雅和简洁的纯Java设计方法。这样，和纯Java一样，它们也完全能够使用Java API。

B.1 简洁和优雅

Groovy的创建者曾寻求开发一门类似于Java的语言，其主要目标就是引入一门语言，对Java开发者来说，他们不会对此语言感到惊奇。举例来说，Groovy类可以由普通的Java代码表示。从语法上来说，Groovy类看起来就像是一个Java类，它可以编译成标准的Java字节码，而且符合Java虚拟机规范（JVM）。对Java虚拟机来说，Groovy语言编译的类文件与Java编程语言编译的类文件并没有任何差别。

根据惯例，我们的第一个应用程序将是“hello world”程序（Kernighan和Ritchie，1988）。此程序将只在标准输出设备上输出文本hello world。其Groovy代码如下所示：

```
println 'hello, world'
```

Java程序员可能会期望看到一个包含main方法的类。main方法包含输出所需的代码。稍微有点令人惊讶的是，在字节码层面上来说，Groovy代码是纯Java代码！如果我们使用Groovy编译器将代码编译成一个.class文件，然后反编译此.class文件的字节码，就会得到所期望的Java代码。

如果此Groovy代码放置在Groovy源文件Hello.groovy中，当研究Groovy编译器编译生成的字节码时，有如下（更简洁但不完全的）等效代码：

```
public class Hello ...{  
  
    public static void main(String [] args){  
        Hello h =new Hello();  
        h.run(args);  
    }  
  
    public void run(String [] args){  
        this.println('hello,world');  
    }  
}
```

这次我们有一个名为Hello的类，这个类含有期望的启动方法main。在main方法中，创建了类Hello的一个对象，并且调用了它的run方法。run方法定义在这个类中，它调用了println方法，它传递的值是我们希望显示的值。当然，println方法并不在类Hello中定义。它是从类Object中继承过来的，类Hello间接地扩展了这个方法。

此解释让我们推断出在Java类Object中定义了println方法，但是事实上却并不是这样。更进一步来说，类Hello中并没有任何Groovy变体类，以扩展Object，并且包含println方法。事实上，这是由于Groovy解释器中途截获了此调用，并自己实现了此功能。这样就让人产生类Object中确实存在println方法的错觉。

Groovy将这些扩充的类作为Groovy开发工具箱（GDK，Groovy Development Kit）的一部分。这些文档中的类（参见<http://groovy.codehaus.org/groovy-jdk.html>）和JDK文档中的类功能相同，但包含了一些额外的方法，因而需要关注Groovy编译器的角色。值得注意的是，JDK并没有变化。

B.2 方法

所有的方法从根本上来说都属于某个类。对于那些在Groovy脚本（参见第7章和附录G）中定义（最顶层）的方法来说，同样如此。如所演示的那样，脚本的代码也封装在某个类中。最顶层的方法也是类的方法。因此，Groovy脚本（源自文件Demo.groovy）：

```
def times(x,y){  
    return x * y  
}  
  
def p =times(3,4)  
println p
```

代码被封装到一个类中，在某种意义上等价于：

```
public class Demo ...{  
  
    public static void main(String [] args){  
        Demo h =new Demo();  
        h.run(args);  
    }  
  
    public void run(String [] args){  
        Object p = this.times(new Integer(3),new Integer(4));  
        this.println(p);  
    }  
  
    public Object times(Object x,Object y){  
        return x.multiply(y);  
    }  
}
```

首先，请注意方法times的代码。由于大部分Groovy代码都使用了动态类型，这样参数的类

型和返回值的类型都是Object。方法本身通过它引用的对象x调用方法multiply实现，其中y为方法multiply的参数。当然，multiply是“*”操作符的实现方法（参见第2章、附录C和附录G）。

在方法run中，变量p表示调用times方法的返回值。实参为整型值，使用整型字面值初始化。请注意p作为方法run的局部变量的方法。

B.3 列表

Groovy天生就支持列表和映射（参见第4章和附录E）。举例来说，一个含有三个整型值的列表为：

```
def numbers = [11, 12, 13]
```

如上所示，Groovy使用[]直接创建列表。我们知道这是通过createList方法实现的。它接收我们传给它的所有Object，然后返回一个包含这些对象的新ArrayList（参见JDK）。

```
public class Demo ...{

    public static void main(String [] args){
        Demo d =new Demo();
        d.run(args);
    }

    public Object run(String [] args){
        Object numbers =createList(new Object [] {new Integer(11),new Integer(12),new Integer(13)});
        return numbers;
    }
}
```

B.4 类

Groovy类（参见第12章和附录I）的一个重要特点是简化了它的Java等价类。使用关键字def以寻求统一属性（实例字段）和方法的概念。在Groovy中，属性等价于实例字段和它的getter/setter方法。

```
class Account {

    //----properties ----

    def number
    def balance
}
```

这个类的Java版本如下所示：

```
public class Account ...{
    public Account(){...}

    public Object getNumber(){return number;}
    public void setNumber(number){this.number =number;}
```

```
public Object getBalance (){return balance;}
public void setBalance(Object balance){this.balance =balance; }

//-----properties -----
Object number;
Object balance;
}
```

每个属性都包含默认的构造器和setter方法。这样，在创建对象时，它就支持Groovy使用命名参数，比如：

```
def acc = new Account(number : 'ABC123', balance : 1200)
```

是下列代码的简化版本：

```
Account acc = new Account();
acc.setNumber('ABC123');
acc.setBalance(new Integer(1200));
```

B.5 多态性

尽管Groovy支持接口（参见第14章），但由于它的动态类型，因而实际上Groovy并不真正需要接口。接口通常用来规定其他类的实现协议。在对象的类实现接口后，就能调用这些接口中定义的所有方法。

在Groovy中，多态性简化了方法名之间的匹配方式。两个属于没有任何关系的两个类的对象也能发送同样的消息，只要在各自的类中定义了相应的处理方法。下面的范例演示了此效果：

```
class Account {

    def display(){
        println "Account:${number}|${balance}"
    }

    def number
    def balance
}

class Student {

    def display(){
        println "Student:${registrationNumber}|${name}"
    }

    def registrationNumber
    def name
}

def group = [new Account(number :'ABC123',balance :1200),
            new Student(registrationNumber :'2006.1234',name :'Ken Barclay')
        ]
group.each {item ->item.display()}
```

类Account和类Student并不共享公共的超类，也不实现相同的接口。当我们迭代处理集合时，可以调用group中所有引用对象的display方法。

B.6 闭包

使用内部类实现Groovy闭包（参见第9章和附录H）（Eckel，2003）。请思考下面的脚本，一个名为times的参数化闭包定义了计算两个值乘积结果的方法。调用闭包后，结果将被赋值给变量z。

```
def times = { x, y ->
    return x * y
}
def z = times(3, 4)
```

一个本地内部类定义了run方法，在这里它被随意地指定为TimesClosure，是Groovy类Closure的扩展类。本地类可以使用封装类的属性和方法。更进一步来说，本地内部类的代码可以使用定义在类方法中的局部变量和参数。这就是闭包作用域的初衷。随后，即可创建这个类的实例，并传递两个整型字面值作为Integer类的实例来调用闭包。

```
public class Demo ... {
    public static void main(String [] args){
        Demo d =new Demo();
        d.run(args);
    }

    public Object run(String [] args){
        class TimesClosure extends Closure {...}
        TimesClosure clos = new TimesClosure(this);
        Object z = invokeClosure(clos,new Object [] {new Integer(3),new Integer(4)});
        return z;
    }
}
```

内部类的定义方式如下所示。对闭包调用，实际上是调用了它的call方法。在这里，方法体就是闭包的定义。

```
class TimesClosure ... {

    public TimesClosure(Object obj){
        super(obj);
        owner = Demo.this;
    }

    public Object call(Object obj1,Object obj2){
        return invokeMethod(this,"multiply ",new Object [] {obj1,obj2})
    }
}
```

B.7 异常

Java编程语言使用异常为程序提供错误处理功能。异常就是在程序执行过程期间出现的，破坏程序正常执行流程的事件。Java运行时系统需要捕捉异常或者指定所有已检查异常的方法，方法可以抛出这些异常。方法能够通过为某类异常提供的异常处理器捕捉异常，或者在方法的定义中使用throws子句抛出异常。

扩展JDKException类的类通常被认为是已检查异常（checked exception）。Java编译器将使用这些类检查程序中是否出现了以下两类事情：

- 抛出已检查异常的所有方法必须在它的方法定义中提供throws子句。
- 调用检查出异常的所有方法必须处理这些异常（使用try和catch），或者使用自己的throws子句抛出异常。

有一些可能出现的错误，如内存被耗尽，已经超出了程序员的控制范围。这将阻止Java虚拟机执行指定的任务。由于不可能计划处理这种类型的错误，因此必须在任何地方捕捉这些错误。这违背了维护代码整齐性的原则。因此，这些错误都是未检查异常（unchecked exception），这也就意味着，此类异常不需要包含在throws子句中。

由于Groovy不区分已检查异常和未检查异常，这样，方法体不支持throws子句。因此，Groovy编译器不会强制实施前面所述的这些规则。Groovy编译器将所有异常默认为未检查异常，除非程序员明确指出属于已检查异常。

附录C 关于数值和表达式的更多信息

Groovy解释器在计算表达式的值时发挥着重要作用。举例来说，在第2章中，我们讨论了操作符在计算表达式值上下文中的作用，如`123+456`，前提条件是可以向`Integer`对象发送`plus`消息。由于Groovy环境并不存在包含此方法的类，因而严格的说，这种做法是不正确的。Groovy解释器认可这种两个整型值相加的形式，所以看起来好像是在执行调用的方法`123.plus(456)`。这种方式的魅力在于Java的核心类，如`Integer`，在Groovy中构建时不仅不需要改动，还为我们提供了类似于Groovy的功能。

C.1 类

在前面的讨论中，曾经说过Groovy中的所有事物都是对象。这一点也不令人惊讶，因此，某个整型值就是类`Integer`的一个实例，某个浮点数值也就是类`BigDecimal`的一个实例。实际上，某个Groovy定义，比如：

```
def age = 25
```

等价于：

```
Integer age = new Integer(25)
```

其中，变量`age`引用了一个`Integer`对象实例。

在第2章中，我们引入了关系运算符和等于运算符。这些运算的结果为布尔值`false`或者`true`。如：

```
def age1 = 25
def age2 = 35
def isYounger = age1 < age2
```

变量`isYounger`引用了一个`Boolean`对象，它的值为`true`。

C.2 表达式

我们一再强调，算术表达式`123+456`，在Groovy中实际上是以`123.plus(456)`的形式调用`plus`方法实现的。这两个表达式在Groovy中都是合法的。根据前面章节所述的内容，也可以使用如下表达式，`new Integer(123).plus(new Integer(456))`。虽然在Groovy中也合法，或许这也不是一个明智的选择，但它演示了构建这些对象的等效方法。

值得注意的是，我们可以将多个赋值语句串接起来，下列语句在Groovy中完全正确。

```
def p = 10
def q = 20
p = q = 30 // p is 30, q is 30
```

C.3 运算符结合性

当表达式包含多个同一优先级的运算符时，运算次序由运算符的结合方向决定。运算符结

合性定义了同一优先级运算符的执行次序。举例来说，表达式：

```
2 + 3 * 4 + 5
```

上面表达式的计算次序将遵循下面的规则，在这三个运算符中，乘法具有最高的优先级，因此将首先执行乘法运算。这样，表达式现在就变为 $2 + 12 + 5$ ，这两个加法运算符具有相同的优先级。由于结合方向为自左至右，因此，将首先执行 $2 + 12$ 运算，其结果为14，最后计算14和5之和，其结果为19，这样最终的结果为19。

如果在表达式 $2 + 3 * 4 + 5$ 中，两个加法运算需要在乘法运算之前执行，就必须在这两个子表达式中使用圆括号。表达式现在的形式为 $(2 + 3) * (4 + 5)$ ，它的结果为5*9或者45。

运算符优先级和结合性的完整表格由表C-1列出。

表C-1 运算符优先级和结合性

分 类	运 算 符	范 例	结合方向
数组下标运算符	[]	a[2]	自左至右
成员运算符	.	a.b()	
后置自增（减）运算符	expr++ expr--	x++	自右至左
单目运算符	++expr --expr + - ~ !	-x	自右至左
乘法运算符	* / %	x*y	自左至右
加法运算符	+ -	x+y	自左至右
移位运算符	<< >> >>>	x << y	自左至右
关系运算符	< <= > >= instanceof	x <= y	自左至右
等于运算符	= = != <=>	x != y	自左至右
按位与运算符	&	x & y	自左至右
按位异或运算符	^	x ^ y	自左至右
按位或运算符		x y	自左至右
逻辑与运算符	&&	x && y	自左至右
逻辑或运算符		x y	自左至右
条件运算符	: ?	a < b ? x : y	自左至右
赋值运算符	= += -= *= /= %= &=	x += y	自右至左
	^= = <=> >=>=		

C.4 定义变量

定义变量时需要引入一个变量名，可以为变量指定初始值，也可以确定变量能被引用的作用域或者作用范围。这些定义变量的代码块出现在定义变量的地方。下列变量的定义是合法的：

```
def a = 10
def b = 20, c = 30
```

请注意，在第二个范例中，同时定义并初始化了两个变量的方法。

定义一个变量，但不初始化它也是可行的。在那种情况下，由于变量可能要被某个对象引用，因而它被隐式地初始化为null。下列代码的效果都相同：

```
def d = null           //explicitly null
def e,f               //both implicitly null
```

在Groovy中，在同一个代码块内两次定义同一个变量的做法是非法的。这样，下面的代码

将会引发一个错误，并提示用户变量p已经被定义：

```
def p =20
def p =30          //error:p already defined
```

如果在变量未定义之前就在表达式中使用变量，同样也会引发一个错误，并提示用户变量p没有定义：

```
def pp =qq        //error:qq not defined
```

本书2.4节详细说明了在Groovy中创建合法标识符的规则。标识符不得使用Groovy语言使用的保留关键字。Groovy关键字在表C-2中列出。

表C-2 Groovy关键字

abstract	any	as	assert	boolean
break	byte	case	catch	char
class	continue	def	default	do
double	else	enum	extends	false
final	finally	float	for	if
implements	import	in	instanceof	int
interface	long	native	new	null
package	private	protected	public	return
short	static	strictfp	super	switch
synchronized	this	threadsafe	throw	throws
transient	true	try	void	volatile
while	with			

C.5 复合赋值运算符

Groovy支持复合赋值运算符以简化某些赋值表达式。其一般语句形式如：

```
variable = variable operator expression
```

也可以写成：

```
variable operator = expression
```

假设这里的operator是表C-1所示的二元赋值运算符。一些范例如：

```
balance +=15           //balance =balance +15
balance -=15           //balance =balance -15
interest *=1.5         //interest =interest * 1.5
interest /=2.5         //interest =interest /2.5
value %=4              //value =value %4
```

在这些范例中，由于变量需要更新，它们应该在前面就已经被定义和初始化。

C.6 逻辑运算符

逻辑运算符由表C-3给出。逻辑与运算符（用&&表示）和逻辑或运算符（用||表示）都是二元运算符。它们被应用到一对布尔值中，运算结果也是布尔值。逻辑非运算符（!）是一元运算符，只要求有一个布尔值，运算结果也是布尔值。

表C-3 逻辑运算符

运算符	描述	结合方向
&&	逻辑与	自左至右
	逻辑或	自左至右
!	逻辑非	自右至左

这些运算符的作用结果如表C-4所示。

表C-4 逻辑运算的真值表

P	Q	P&Q	P Q	S	!S
false	false	false	false	false	true
false	true	false	true	true	false
true	false	false	true		
true	true	true	true		

在逻辑与运算符和逻辑或运算符中有一个微妙的地方值得注意。当子表达式中`&&`和`||`两边的操作数需要计算值时，如果整个表达式的结果一旦确定，程序就会立即停止计算余下操作数的值。举例来说，假设`expr1`和`expr2`是子表达式的操作数，那么，在如下表达式中：

```
expr1 & expr2
```

如果`expr1`的值确定为`false`，则程序就不会计算`expr2`的值。请思考下面的语句：

```
def a = 10
def b = 20
def c = 30
def d = (b < a) && (c = 40)
```

由于子表达式`b < a`的值为`false`，这样变量`c`就不会被赋予新值`40`。

同样地，如果`expr1`为`true`，那么在：

```
expr1 || expr2
```

中，程序就不会计算`expr2`的值，这是因为整个逻辑表达式的值已经确认为`true`。

C.7 条件运算符

条件运算符（也被称为三元运算符）将产生一个值，它的一般形式为：

```
expr ? expr#1 : expr#2
```

如果表达式`expr`的计算值为布尔值`true`，则表达式`expr#1`的计算值就是整个表达式的值。如果表达式`expr`的计算值为布尔值`false`，则求表达式`expr#2`的计算值就是整个表达式的值。一般来说，可以使用这个运算符产生用来赋值的值或者返回值。下面是条件表达式的一些范例：

```
def positive = x < 0 ? -x : x
```

```
def min(x, y) |
    return x < y ? x : y
|
```

C.8 数字字面值的分类

到目前为止，已经使用了默认的十进制记数法表示整型字面值。但是，也可以使用分别以0或者0x开头的八进制或者十六进制的整型字面值。举例来说，可能会使用0xAB和0177分别表示十进制数171和127。另外，也可以为某个整型字面值加上整型后缀(I、L或者G)。这个整型值就分别变成了类Integer、Long或者BigInteger的实例，这些类表示的范围值也越来越大。如果没有指定后缀，则会把数字字面值当作合适它的类的实例。

浮点数字面值默认认为是类BigDecimal的实例。通常我们都使用十进制记数法，但也有可能使用科学记数法。在这种情况下，浮点数字面值就会在可选字母E(或者e)后面追加一个带符号的整型值。举例来说，可能会使用123E2和1.23e-2分别表示12300.0和0.0123。和整型字面值一样，也可以将后缀F、D或者G应用到浮点数字面值中，以限定它们分别为类Float、Double或者BigDecimal的实例。

下面是一些合法的范例：

```
123L      //Long
456G      //BigInteger
1.23F     //Float
4.56D     //Double
7.89G     //BigDecimal
```

C.9 转换

在Groovy中引入变量时，必须使用关键字def定义。我们也可以显式地指定基本数据类型，以引入一个静态变量(参见第10节)。在下面的代码片断中，变量temperature被静态地指定为Double类型，age变量却没有被指定，但是它的值在运行时将被指定为Integer类型。

```
double temperature = 32.0
def age = 25
```

请注意temperature实际上是一个Double实例。我们将简要地讨论此用法。在本书中使用的其他基本类型还包括int和boolean。举例来说，有如下代码：

```
int age = 25
```

和

```
boolean result = age > 21
```

现在，age被静态地指定为Integer类型，result则被静态地指定为Boolean类型。

根据我们前面的讨论可知，在Groovy中所有的事物都是对象。因此，每个类型为int、double和boolean的值最终都会变成Java类Integer、Double和Boolean的实例，就一点也不令人惊讶了。这样，temperature就变成类Double的一个实例，age变成类Integer的一个实例，result变成类Boolean的一个实例。实际上，Groovy定义，比如：

```
int age = 25
```

等价于：

```
Integer age = new Integer(25)
```

无论如何，在Groovy中计算表达式的值比执行这些转换要复杂得多。我们知道，整型值可

以精确地存储其值。但是，这些值的有效取值范围是有限的，就如应用到基本类型int上的封装器类Integer一样。由于浮点数值是以二进制表示的，因而通常也不能在计算机中精确的存储浮点数值，尽管现在值的有效范围已经大大增加。应用到基本类型float和double的封装器类Float和Double也存在这个问题。

一个可选的解决方案是，使用一些允许存储任意精度和取值范围数值的其他表示方法。Java API提供了类BigInteger和BigDecimal以解决这个问题。虽然我们不关心这种表示方法的具体细节，但我们应该意识到，当把这些数据与其等价的二进制数据比较时，通常会发现它们效率极其低下。因此，它们之间的计算将耗费更长的时间和更多的计算机内存。

当然，Groovy提供了访问所有Java math的类和操作的方法。不管使用何种方法，都要尽可能使Groovy脚本更直观地表示数字字面值，现在已经采用了一种“最不令人惊讶”的方法。这样，在Groovy中，字面值含有小数点的值将默认认为是BigDecimal类型，而不是Float或者Double类型。如果有必要，Float和Double类型值可以显式地创建，或者通过它的后缀（参见第8节）创建。当然，如果有必要也可以采用科学记数法表示。

类似地，一个整型字面值（不含后缀）将默认认为是符合该值的最小数据类型。也就是说，是Integer、Long或者BigInteger。和前面一样，可以显式地或者使用后缀创建Integer和Long类型值。对于十六进制数和八进制数来说也同样适用。

另一个“最不令人惊讶”方法的结果是，groovy从不使用数据类型自动提升功能将某个二进制浮点数提升为BigDecimal值。这样做就意味着由于精度等级不同不能保证浮点数的转换精度。使用二进制表示的数执行效率较高，这也是事实，因此一旦引入了二进制数，最好就一直保持。这样，包含BigDecimals、BigIntegers、Doubles、Floats、Longs和Integers的二元运算就依据表C-5对参数进行自动转换。唯一例外的就是下面将要讨论的除法。

表C-5 参数转换矩阵

	<i>BigDecimal</i>	<i>BigInteger</i>	<i>Double</i>	<i>Float</i>	<i>Long</i>	<i>Integer</i>
<i>BigDecimal</i>	<i>BigDecimal</i>	<i>BigDecimal</i>	<i>Double</i>	<i>Double</i>	<i>BigDecimal</i>	<i>BigDecimal</i>
<i>BigInteger</i>	<i>BigDecimal</i>	<i>BigInteger</i>	<i>Double</i>	<i>Double</i>	<i>BigInteger</i>	<i>BigInteger</i>
<i>Double</i>	<i>Double</i>	<i>Double</i>	<i>Double</i>	<i>Double</i>	<i>Double</i>	<i>Double</i>
<i>Float</i>	<i>Double</i>	<i>Double</i>	<i>Double</i>	<i>Double</i>	<i>Double</i>	<i>Double</i>
<i>Long</i>	<i>BigDecimal</i>	<i>BigInteger</i>	<i>Double</i>	<i>Double</i>	<i>Long</i>	<i>Long</i>
<i>Integer</i>	<i>BigDecimal</i>	<i>BigInteger</i>	<i>Double</i>	<i>Double</i>	<i>Long</i>	<i>Integer</i>

如果操作数是Float或者Double的话，除法运算符/和/=的结果将是一个Double值。否则，其结果将是BigDecimal。换句话说，这是在正常情况下的结果。然而，如果是整型值除法的话，可以通过强制取整抛弃结果值的小数部分，如：

```
(int)(3/2)           //gives a result of 1
```

或者使用intdiv操作，如

```
3.intdiv(2)          //gives a result of 1
```

C.10 静态类型

在Groovy中也允许使用静态类型变量。我们使用基本类型名，如int或者double，或者使用

用户自定义的类名替代了关键词def。这样，可能有如下代码：

```
int age = 25
double temperature = 98.4
boolean isAdult = true
```

静态类型变量随后将检查它们是否被正确地使用。举例来说，在前面提到的变量定义上下文中，下面的代码将会抛出异常，因为用户试图将一个字符串值赋给int变量：

```
age = "Ken"
```

C.11 测试

数字字面值的表示方法和计算算术表达式的值是Groovy的重要组成部分。不管使用何种方法，都可能会非常困难，而且容易让人混淆。因此，一个非常明智的做法是，检查Groovy的行为是否和我们期望的行为一致。下面列出的类GroovyMathTest的部分代码含有一些有代表性的单元测试（参见第15章）代码，可以帮助我们实现测试的目标。这些测试都基于断言，可以从Groovy主站点获得（参见<http://groovy.codehaus.org>）。有兴趣的读者可以添加其他合适的代码。

```
import groovy.util.GroovyTestCase

class GroovyMathTest extends GroovyTestCase {

    void testExactLiteralDefaultCalculations(){
        assertTrue(1.1 + 0.1 == 1.2)
    }

    void testOctal(){
        assertTrue(0177 == 127)
    }

    void testHexadecimal(){
        assertTrue(0xAB == 171)
    }

    void testSuffixes(){
        assertTrue("I suffix", 42I == new Integer("42"))
        assertTrue("L suffix", 123L == new Long("123"))
        assertTrue("Long type", 2147483648 == new Long("2147483648"))
        assertTrue("G suffix #1", 456G == new java.math.BigInteger("456"))
        assertTrue("G suffix #2", 123.45G == new java.math.BigDecimal("123.45"))
        assertTrue("G suffix #3", 123.45G == new Double("123.45"))
        assertTrue("Default BigDecimal type", 123.45 == new java.math.BigDecimal("123.45"))
        assertTrue("D suffix", 1.200065D == new Double("1.200065"))
        assertTrue("F suffix", 1.234F == new Float("1.234"))
        assertTrue("E suffix", 1.23E23D == new Double("1.23E23"))
        assertTrue("e suffix", 1.23e23D == new Double("1.23e23"))
    }
}
```

```
void testDivision(){
    assertTrue("Trailing zeros removed",1/2 ==new java.math.BigDecimal("0.5"))
    assertTrue("Normalized to 10 places #1",1/3 ==new java.math.BigDecimal("0.3333333333"))
    assertTrue("Normalized to 10 places #2",2/3 ==new java.math.BigDecimal("0.6666666667"))
}

void testIntegerDivisionByCasting(){
    assertTrue((int)(3/2)==1I )
}

void testIntegerDivisionByIntdiv(){
    assertTrue(3.intdiv(2)==1I )
}

void testBinaryOperationConversionMatrix(){
    //Many more assertions could be made.
    //
    assertTrue("#1 G +G #1", (123.45 +67.8).getClass().getName() == "java.math.BigDecimal" )
    assertTrue("#1 G +G #2", (123.45 +67G).getClass().getName() == "java.math.BigDecimal" )
    assertTrue("#1 G +D", (123.45 +67.8D).getClass().getName() == "java.lang.Double" )
    assertTrue("#1 G +F", (123.45 +67.8F).getClass().getName() == "java.lang.Double" )
    assertTrue("#1 G +L", (123.45 +67L).getClass().getName() == "java.math.BigDecimal" )
    assertTrue("#1 G +I", (123.45 +67I).getClass().getName() == "java.math.BigDecimal" )
    //
    assertTrue("#2 G +G #1", (123 +67.8).getClass().getName() == "java.math.BigDecimal" )
    assertTrue("#2 G +G #2", (123 +67G).getClass().getName() == "java.math.BigInteger" )
    assertTrue("#2 G +D", (123 +67.8D).getClass().getName() == "java.lang.Double" )
    assertTrue("#2 G +F", (123 +67.8F).getClass().getName() == "java.lang.Double" )
    assertTrue("#2 G +L", (123G +67L).getClass().getName() == "java.math.BigInteger" )
    assertTrue("#2 G +I", (123G +67I).getClass().getName() == "java.math.BigInteger" )
    //
    assertTrue("#3 L +L", (123 +67L).getClass().getName() == "java.lang.Long" )
    assertTrue("#3 I +I", (123 +67I).getClass().getName() == "java.lang.Integer" )
}
```

附录D 关于字符串和正则表达式的更多信息

正则表达式是字符串和模式之间的匹配规则。它们为描述这些模式提供了强有力的机制，比如依据模式分隔字符串，或者依据模式修改周围的字符串。由于它们复杂的结构，乍看上去使它们显得有点让人畏惧。然而，只需很少的实践即可轻松地掌握其本质。

D.1 正则表达式

本质上来说，正则表达式是Groovy中一种特殊的程序语言。它们提供字符串是否与某些模式匹配的确认方式。举例来说，可能希望确认某个字符串是否与社会保险号码匹配。我们也可以使用正则表达式以各种方式分隔字符串或者修改字符串。

正则表达式以字符串的形式表示。然而，由于有些字符被指定了特殊的用途，因而被称为元字符。本附录的许多内容都致力于在正则表达式中演示它们的功能。我们将要讨论的元字符包括：

. ^ \$ * +?[] {} \ | ()

表D-1给出了它们所代表的意义的简要解释。

表D-1 元字符

元字符	描述
.	匹配任意单个字符。如，正则表达式b.t将匹配字符串bat、bet，但是不匹配beat
^	匹配某个行的开始部分。如，正则表达式^Mat将匹配字符串Matches的开始部分，但是不匹配Football Matches
\$	匹配某个行的结尾部分。如，正则表达式ball\$将匹配字符串Football的结尾部分，但是不匹配Football Matches
*	匹配位于*符号前面的字符或者正则表达式出现零次或者多次。如，正则表达式A*匹配出现任意次数的A，也就是说，A、AA、AAA等等
+	匹配位于+符号前面的字符或者正则表达式出现一次或者多次。如，正则表达式A+匹配出现任意次数的A，也就是说，A、AA、AAA等等
?	匹配位于?符号前面的字符或者正则表达式出现零次或者一次
[]	匹配方括号中列出的字符集中的任一字符。如，正则表达式B[aeu]t将匹配Bat、Bet和But
{}	匹配位于{}符号之前、某个字符集范围之内字符的特定次数，如，正则表达式[0-9]{3}将精确匹配三个数字
\	转义字符，用来使其后的字符表示它本身，哪怕它是一个元字符。如，\\$表示美元符号
	选择符，同时包含两个表达式。如，正则表达式(He She) said同时匹配He said和She said
()	将封装的表达式组合起来，保留匹配的字符

D.2 单字符匹配

在正则表达式中的点字符（.）可以匹配任意单个的字符。范例01所示的一组断言都为真。

范例01 单字符匹配

```
assert 'Bat'=='B.t'  
assert 'Bet'=='B.t'  
assert 'But'=='B.t'  
assert 'Batter'=='B.tt.r'  
assert 'Better'=='B.tt.r'  
assert 'Butter'=='B.tt.r'  
  
assert 'B.t'=='B \\.t'  
assert !( 'Bat'=='B \\.t' )
```

D.3 匹配开始部分

在正则表达式中的脱字符号（^）只匹配字符串的开始部分。范例02所示的一组断言都为真。

范例02 匹配开始部分

```
assert 'Batter'=='^Bat'  
assert 'Batter'=='^Batt'  
assert !( 'batter'=='^Bat' )
```

D.4 匹配结尾部分

在正则表达式中的美元符号（\$）只匹配字符串的结尾部分。范例03所示的一组断言都为真。

范例03 匹配结尾部分

```
assert 'Football'=='ball$'  
assert 'Mothball'=='ball$'  
assert !( 'Dancehall'=='ball$' )
```

D.5 匹配零次或者多次

正则表达式中的星号符号（*）用来匹配位于“*”符号前面的字符或者正则表达式，出现零次或者多次。范例04所示的一组断言都为真。

范例04 匹配零次或者多次

```
assert 'Ft'=='Fo * t'  
assert 'Fot'=='Fo * t'  
assert 'Foot'=='Fo * t'  
assert 'Foooot'=='Fo * t'
```

D.6 匹配一次或者多次

正则表达式中的加符号（+）用来匹配位于“+”符号前面的字符或者正则表达式，出现一

次或者多次。范例05所示的一组断言都为真。

范例05 匹配一次或者多次

```
assert !('Ft'='~'Foot')
assert 'Foot'='~'Foot'
assert 'Foot'='~'Foot'
assert 'Foot'='~'Foot'
```

D.7 匹配零次或者一次

正则表达式中的问号符号（?）用来匹配位于“?”符号前面的字符或者正则表达式，出现零次或者一次。范例06所示的一组断言都为真。

范例06 匹配零次或者一次

```
assert 'Shoe-shine'='~'Shoe-?shine'
assert 'Shoeshine'='~'Shoe-?shine'
```

D.8 次数匹配

正则表达式中的重复次数修饰符{m, n}用来匹配位于“{”前面的字符或者正则表达式，至少出现m次但至多出现n次。m和n都为正整型值。如果n被省略，但前面的逗号存在，则可认为至少出现m次。如果仅给定一个值n，则为精确匹配出现n次。范例07所示的一组断言都为真。

范例07 次数匹配

```
assert 'abc'='~'ab{1,3}c'
assert 'abbc'='~'ab{1,3}c'
assert 'abbcc'='~'ab{1,3}c'

assert !('ac'='~'ab{1,3}c')
assert !('abbbbc'='~'ab{1,3}c')

assert 'abbbc'='~'ab{3}c'
assert !('abbc'='~'ab{3}c')

assert 'abbbc'='~'ab{3,}c'
assert 'abbbb'='~'ab{3,}c'
assert !('abbc'='~'ab{3,}c')
```

请注意，“*”修饰部分等价于{0, }、“+”修饰部分等价于{1, }、“?”修饰部分等价于{0, 1}。

D.9 字符类型

字符类型用来从一组字符集中匹配单个字符。字符集在方括号“[”和“]”之间列出。单个字符集可以以如下方式给出[aeiou]（元音字母），或者以范围的形式给出，如[a-z]（小写字母）。也可以使用[a-zA-Z]（字符）的方式表示多个范围。为了匹配那些不包含在范围中的字符，

可以使用补足字符的方式表示，如`[^0-9]`（也就是除了数字之外的任意字符）。如范例08所示。

范例08 字符类型

```
assert '0'=='[0-9]{1,3}'
assert '01'=='[0-9]{1,3}'
assert '012'=='[0-9]{1,3}'
assert !('0123' == '[0-9]{1,3}')
assert !('A45'=='[0-9]{3}')

assert 'tan'=='t [aeiou]n'
assert 'ten'=='t [aeiou]n'
assert 'tin'=='t [aeiou]n'
assert 'ton'=='t [aeiou]n'
assert 'tun'=='t [aeiou]n'
assert !('Tan'=='t [aeiou]n')
```

如果想在字符集中匹配破折号（-），则必须把它放在正则表达式的第一位位置，如`[-ab]`所示，这样，它就不会被解释为指定范围的字符。

D.10 选择

选择模式通过元字符|表示。选择通常使用圆括号分组。如范例09所示。

范例09 选择

```
assert 'tan'=='t(a|e|i|o|u)n'
assert 'ten'=='t(a|e|i|o|u)n'
assert 'tin'=='t(a|e|i|o|u)n'
assert 'ton'=='t(a|e|i|o|u)n'
assert 'tun'=='t(a|e|i|o|u)n'
assert 'toon'=='t(a|e|i|oo|u)n'
```

D.11 辅助符号

Groovy中有一些表示通用正则表达式的简写符号。表D-2列举了部分简写符号。

表D-2 简写符号

简写字符	等价符号	描述	简写字符	等价符号	描述
\d	[0-9]	数字	\W	[^a-zA-Z0-9]	非word字符
\D	[^0-9]	非数字	\s	[\t\n\f\r\v]	空白字符
\w	[a-zA-Z0-9]	Word字符	\S	[^\s]	非空白字符

正则表达式使用反斜杠字符“\”取消对特殊字符的转义。不幸的是，Groovy在字符串字面值中，把它当作转义字符使用时会发生冲突。这样，就必须使用更多的反斜杠字符表示任意个数的反斜杠字符。下列代码演示了此效果。

范例10 辅助符号

```
assert '12:34' =~ '\d{2}:\d{2}'
assert !('12:34' =~ '\d{2}:\d{2}')

assert 'Hello world' =~ '\w* \s* \w*'
assert !('Hello world Groovy' =~ '\w* \s* \w*')
```

D.12 组合

正则表达式中的括号可用来建立那些随后从匹配字符串中检索子字符串的组合。也就是说，正则表达式可以将字符串分割成多个子组合。举例来说，U.S.社会保险号码的形式为999-99-9999，也就是三个数字，随后紧跟一个连字符，再接两个数字，之后再紧跟另一个连字符，最后接四个数字。范例11中的正则表达式可以实现此功能，它能够检索出三个数字子组合。

范例11 U.S.社会保险号码

```
def SSNPATTERN = '([0-9]{3})-([0-9]{2})-([0-9]{4})'
def ssN = '123-45-6789'

def matcher =ssN =~SSNPATTERN
matcher.matches()

println "matcher [0]:<\$matcher [0]>"
println "matcher [0][0]:<\$matcher [0][0]>"
println "matcher [0][1]:<\$matcher [0][1]>"
println "matcher [0][2]:<\$matcher [0][2]>"
println "matcher [0][3]:<\$matcher [0][3]>"
```

变量matcher是类Matcher（参见JDK文档）的一个对象。它可以执行不同类型的匹配操作。举例来说，方法matches试图在整个文本中匹配那些和模式相反的文本。GDK为这个类添加了getAt方法，以支持索引操作符。举例来说，matcher[0][1]表示所有匹配模式中的第一个匹配模式中的第一个组合。应用程序的输出结果如下所示：

```
matcher[0]: <["123-45-6789", "123", "45", "6789"]>
matcher[0][0]: <123-45-6789>
matcher[0][1]: <123>
matcher[0][2]: <45>
matcher[0][3]: <6789>
```

范例12演示了使用组合获取英国小汽车牌照（执照）号码部分内容的方法。小汽车牌照的号码形式为XX99XXX，其中，这两个数字用来确认牌照的注册日期。举例来说，01表示2001年的四月份，51表示2001年的九月份，02表示2002年的四月份等。

范例12 U.K.车牌号

```
def UKPATTERN = '([A-Z]{2})([0-9]{2})\s([A-Z]{3})'
def ukPlate ='SK51 PIQ'

def matcher =ukPlate =~UKPATTERN
```

```
matcher.matches()

println"matcher [0]:<${matcher [0]}>"  
println"matcher [0][0]:<${matcher [0][0]}>"  
println"matcher [0][1]:<${matcher [0][1]}>"  
println"matcher [0][2]:<${matcher [0][2]}>"  
println"matcher [0][3]:<${matcher [0][3]}>"
```

输出结果如下所示：

```
matcher[0]: <["SK51 PIQ", "SK", "51", "PIQ"]>  
matcher[0][0]: <SK51 PIQ>  
matcher[0][1]: <SK>  
matcher[0][2]: <51>  
matcher[0][3]: <PIQ>
```

最后一个范例是以MMM DD, YYYY的形式表示日期值，如NOV 28, 2005所示。范例13的输出结果如下所示。

范例13 日期值

```
matcher [0]:<["NOV 28,2005 ", "NOV ", "28 ", "2005 "]>  
matcher [0][0]:<NOV 28,2005>  
matcher [0][1]:<NOV>  
matcher [0][2]:<28>  
matcher [0][3]:<2005>

def DATEPATTERN ='([A-Z]{3})\\s([0-9]{1,2}),\\s([0-9]{4})'  
def date ='NOV 28,2005'

def matcher =date =~DATEPATTERN  
matcher.matches()

println"matcher [0]:<${matcher [0]}>"  
println"matcher [0][0]:<${matcher [0][0]}>"  
println"matcher [0][1]:<${matcher [0][1]}>"  
println"matcher [0][2]:<${matcher [0][2]}>"  
println"matcher [0][3]:<${matcher [0][3]}>"
```

附录E 关于列表、映射和范围的更多信息

当我们在Groovy脚本中定义的对象上调用方法时，Groovy解释器就会起到重要的作用。从本质上来说，解释器截获了这些方法的调用，因而能够添加那些并没有定义在Java核心类中的方法。以这样的方式，Groovy设计者能够扩展Java核心类（JDK）的功能，使得它们看起来更像Groovy。不管使用何种方法，都要知道JDK本身并没有改动，这一点非常重要。举例来说，如果我们在某个列表上调用each（参见第9章）方法，如：

```
def numbers = [11, 12, 13, 14]
numbers.each {it -> println it * 2}
```

随后我们将会察觉到，JDK的所有类中都不存在这个方法。解释器使它看起来更像是List对象中有此方法，即使它并不在JVM层中存在。Map类也以类似的方式增强了其功能。我们将把这些新增的功能看作Groovy GDK方法。

我们也应该知道，当定义某个列表时，如：

```
def numbers = [11, 12, 13, 14]
```

Groovy的动态类型（dynamic typing）将开始发挥作用。当定义好某个列表字面值之后，numbers即可在运行时引用ArrayList。

然而，即使它是一个普通的JDK ArrayList，它也能借助Groovy GDK方法响应这些新增的方法，比如each。类似地，当我们使用如下代码定义某个映射时：

```
def names = ['Ken' : 'Barclay', 'John' : 'Savage']
```

names即可在运行时引用HashMap。和以前一样，它的行为也是通过增强Groovy GDK方法实现的。

E.1 类

我们提到的列表是指某个行为遵循List接口的对象，并不关心它的具体类型。通过增强Groovy GDK中与列表有关的方法，更进一步地限定了最后一个声明。我们已经注意到默认的实现方法是ArrayList：

```
def numbers =[1,2,3]
println "numbers:${numbers.getClass().getName()}"      //java.util.ArrayList
```

如果有必要，可以使用as子句的不同表示方法：

```
def numbers [] as LinkedList
numbers.addAll([1,2,3])
println "numbers:${numbers.getClass().getName()}"      //java.util.LinkedList
```

无论如何，都必须非常小心，假定有如下所示的代码：

```
def numbers =[] as LinkedList
numbers =numbers +[1,2,3]
println "numbers:${numbers.getClass().getName()}" //java.util.ArrayList
```

在这里，程序通过默认的ArrayList给变量numbers赋值。在前面的演示中，变量numbers保留了LinkedList，只不过是由于我们需要在它上面调用addAll方法而已。

as子句只能用于空列表中，如果我们使用：

```
def numbers =[1, 2, 3] as LinkedList
```

将会抛出异常，这是因为我们试图把ArrayList对象强加给LinkedList对象。由于类LinkedList并没有ArrayList子类，因而这是不允许的。

当某个方法（如JDK方法）需要使用数组值作为它的实参时，使用as子句就非常有效。假设代码中有一个含有值的列表传递给那个方法，可以使用下面的代码：

```
def names ='['Ken','John','Jessie']'
def someMethod(String [] args){...}
someMethod(names as String []) //convert to required type
```

E.2 列表

在第4章中，把列表方法getAt和putAt分别看作是表达式左边和右边的索引操作符。因此：

```
def numbers =[1, 2, 3]
def x = numbers[1] // x = numbers.getAt(1)
numbers[1] = 22 // numbers.putAt(1, 22)
```

如果使用索引操作符给超出当前列表长度的索引赋值的话，则其他的索引值将自动为null。在下面的范例中，变量numbers的初始长度为3。赋值语句将它的长度增加到10，则索引3和索引8之间的所有值都为null。

```
def numbers =[0, 1, 2]
println "numbers: ${numbers}" // numbers: [0, 1, 2]
numbers[9] = 9
println "numbers: ${numbers}" // numbers: [0, 1, 2, null, null, null, null, null, null, null, 9]
```

另外，超出列表长度的索引值也为null：

```
println "numbers[20]: ${numbers[20]} " // numbers[20]: null
```

E.3 范围

范围的字面值1..10是IntRange类的一个实例。IntRange类实现了Range接口，并增强了类AbstractList。Range接口引入了方法getFrom、getTo和isReverse。增强AbstractList也就意味着范围也支持列表方法。下面的代码片段演示了此用法：

```
def rng = 1..10
println "rng: ${rng.getClass().getName()}" // groovy.lang.IntRange
```

```
println "to, from: ${rng.getFrom()} ${rng.getTo()}"           // to, from: 1 10
println "get: ${rng.get(0)}"                                // get: 1
```

对范围来说，并不是所有的列表方法都适用，因此范围不能使用这些方法。举例来说，`rng.set(0, 99)`将会抛出不支持此操作的异常。

这里有一些特殊的案例需要注意，如下所示：

```
println "1..1: ${1..1}"      // 1..1: [1]
println "1..0: ${1..0}"      // 1..0: [1, 0]
println "1..<2: ${1..<2}"    // 1..<2: [1]
println "1..<1: ${1..<1}"    // 1..<1: []
```

第一个和第三个范例中的范围都只有一个值。第二个范围则有两个以倒序的方式排列的值。由于最后一个范例的范围没有值，因此它有潜在的缺陷。

E.4 展开操作符

展开操作符用来将某个列表的元素展开成另一个列表，或者将某个映射的元素展开成另一个映射。列表中的展开操作符用“*”符号表示。如：

```
def x = [2, 3]
def y = [0, 1, *x, 4, *[5, 6, 7]]
println "y: ${y}"           // y: [0, 1, 2, 3, 4, 5, 6, 7]
```

变量y表示将列表中使用变量x和字面值为[5, 6, 7]表示的元素展开成列表y。

同样，映射中的展开操作符用“*：“符号表示。在下面的代码中，映射x和字面值为[6 : 'f', 7 : 'g'] 的映射被合并为映射y。

```
def x = [3 : 'c', 4 : 'd']
def y = [1 : 'a', 2 : 'b', *:x, 5 : 'e', *:[6 : 'f', 7 : 'g']]
println "y: ${y}"
// y: [2: "b", 4: "d", 6: "f", 1: "a", 3: "c", 7: "g", 5: "e"]
```

E.5 测试

非常值得为程序引入一些简单的测试，哪怕它们仅能确认我们是否正确理解了Groovy的部分细节。下面的GroovyTestCase是这些类型测试的一个范例，其他的测试可以很容易加进去。

```
import groovy.util.GroovyTestCase

class GroovyJDKTests extends GroovyTestCase {

    //Establish the fixture
    def obj
    def table
    def count

    void setUp(){
        obj = new Object();
```

```
table =[11,12,13,14]
count =0
}

//*****
//Object tests
//*****

//Show that we have an Object
void testObjectClassName(){
    def className =obj.getClass().getName()
    assertTrue(className =="java.lang.Object ")
}

//Show that we have a toString method
void testObjectToStringMethodName(){
    def methods =obj.getClass().getMethods()
    def methodNames =methods.collect{element ->return element.getName()}
    assertTrue(methodNames.contains("toString "))
}

//Show that we have the correct number of public methods from JDK Object
void testObjectMethodNumber(){
    def methods =obj.getClass().getMethods()
    assertTrue(methods.length ==9 )
}

//Show that we don't have an each method
void testObjectEachMethodName(){
    def methods =obj.getClass().getMethods()
    def methodNames =methods.collect{element ->return element.getName()}
    assertFalse(methodNames.contains("each "))
}

//Show that we can send the each message
void testEachMethodCallForAnObject(){
    obj.each {element ->count++}
    assertTrue(count ==1 )
}

//*****
//List tests
//*****


//Show that we have an ArrayList
void testListClassName(){
```

```
def className =table.getClass().getName()
assertTrue(className == "java.util.ArrayList" )
}

//Show that we have a size method
void testListSizeMethodName(){
    def methods =table.getClass().getMethods()
    def methodNames = methods.collect{element ->return element.getName()}
    assertTrue(methodNames.contains("size "))
}

//Show that we have the correct number of public methods from JDK ArrayList
void testArrayListMethodNumber(){
    def methods =table.getClass().getMethods()
    assertTrue(methods.length ==35 )
}

//Show that we don't have an each method
void testListEachMethodName(){
    def methods =obj.getClass().getMethods()
    def methodNames =methods.collect{element ->return element.getName()}
    assertFalse(methodNames.contains("each "))
}

//Show that we can send the each message
void testEachMethodCallForAList(){
    table.each {element ->count++}
    assertTrue(count ==4 )
}
}
```

附录F 关于基本输入输出的更多信息

在本附录中，我们将进一步讨论格式化输出，并关注第5章引入的输入类Console。

F.1 格式化输出

格式字符串中的转换规范由百分号（%）引入。这样，就有一系列转换可供选择。格式规范以单字符表示的转换操作结束。转换规范的一般定义如下：

```
%[index$][flags][width][.precision]conversion
```

在这里index是可选的，是一个无符号整数，用来指示参数在参数列表中的位置。第一个参数用1\$引用，第二个用2\$引用，依此类推。我们将不演示此选项的用法。可选参数flags是一组用来更改格式输出的字符。可选参数width是一个非负的十进制整型值，用来指示输出字符的最小宽度。可选参数precision也是一个非负的十进制整型值，用来限制字符的个数。具体的行為由转换决定。最后，必需的参数conversion用来指示参数将使用的格式。参数类型决定了给定转换参数的合法性。

本附录并不是要讨论所有可能的组合，毕竟数量太多了。在这里，提供了几个常见范例的解释。欲获取完整描述的读者，请参考Formatter类（JDK）的说明文档。

在我们的所有范例中，实际输出值都使用“[”和“]”，以强调实际输出是经过转换的结果。实际输出效果由printf语句旁边的注释给出。

范例01演示了使用%d转换规范输出整型值的效果。转换规范%d的作用是将输出的整型值转换为有符号十进制数输出。请仔细观察，-标识表示输出结果靠左侧输出，+标识表示在值前面使用符号，0标识表示在输出域内，前面的字符使用0补充。

范例01 %d转换控制

```
def j =45
def k =-123
def jj =123456780

printf("[%d]\n",[k]) //[-123]
printf("[%4d]\n",[jj])      // [ 45]
printf(["%-5d"]\n,[j])      // [45]
printf(["%05d"]\n,[j])      // [00045]
printf(["%2d"]\n,[k])      // [-123]
printf(["%+4d"]\n,[j])      // [+45]
printf(["%+05d"]\n,[j])      // [+045]
printf(["%d"],[jj]) // [123456789]
```

范例02使用%f作为有符号十进制浮点数值的转换控制字符。包含转换规范的范例，比如

10.2，它表示域的宽度总共为10个字符，并且精确到2位小数。

范例02 %f转换控制

```
def x =12.345
def y =-678.9

printf("[%f]\n",[y])           //[-678.900000]
printf(["%-10.2f"]\n,[x])      //[12.35]
printf(["%+8.1f"]\n,[x])       // [+12.3]
printf(["%.2f"]\n,[y])          // [-678.90]
printf(["%08.2f"]\n,[x])        // [00012.35]
printf(["%+06.1f"]\n,[x])       // [+012.3]
```

%e和%E转换字符通常以科学记数法表示浮点数值。其一般形式为d.ddddde+dd (用%e控制)和d.ddddddE+dd (用%E控制)。可选参数precision指出小数位数 (默认为6位)。范例03演示了部分用法。

范例03 %e和%E转换控制

```
def x =12.345
def y =-678.9

printf(["%e"]\n,[x])           // [1.234500e+01]
printf(["%E"]\n,[x])           // [1.234500E+01]
printf(["%14.4e"]\n,[x])       // [ 1.2345e++01]
printf(["%-.12.1E"]\n,[y])     // [-6.8E+02]
printf(["%+.12.2E"]\n,[x])     // [+1.23E+01]
printf(["%.2e"]\n,[y])          // [-6.79e+02]
printf(["%10.0e"]\n,[x])        // [ 1e+01]
```

在范例04中，演示了使用%s转换字符格式化输出字符串的方法。在这里需要注意，可选参数precision用来表示字符串的输出字符个数。

范例04 %s转换控制

```
def message ="Hello"

printf(["%s"]\n,[message])      // [Hello]
printf(["%8s"]\n,[message])      // [ Hello]
printf(["% -8s"]\n,[message])    // [Hello]
printf(["%6.2s"]\n,[message])    // [ He]
printf(["% -10.6s"]\n,[message]) // [Hello]
```

本节的最后一个范例将演示格式字符串中值过多或者过少时的输出情况。在第二个演示范例中，多出的值将被忽略。在最后一个演示范例中，当提供的值数量不足时将抛出异常。

范例05 值过多或者过少时的格式输出

```
def x = 21
def y = 22
```

```
def z =23

printf('First:%d,second:%d \n',[x,y])           //ok
printf('First:%d,second:%d \n',[x,y,z])         //extras ignored
//printf('First:%d,second:%d \n',[x])            //raise exception
```

F.2 类Console

第5章引入了类Console，以从用户的控制台中读取不同类型的值。从本质上来说，这些获取某个行缓冲器中下一个值的方法都是静态方法。当行缓冲器被耗尽时，需要使用新的用户输入刷新缓冲器。Console类的代码如下：

```
package console

class Console {

    def static readLine(){
        return getNextLine()
    }

    def static readString(){
        return getNextToken()
    }

    def static readInteger(){
        return getNextToken().toInteger()
    }

    def static readDouble(){
        return getNextToken().toDouble()
    }

    def static readBoolean(){
        return (getNextToken() == "true")
    }

    private static String getNextToken(){
        if(inputLine == null)
            readInputLine()

        while(inputIndex == number_of_tokens)
            readInputLine()

        return inputTokens [inputIndex++]
    }

    private static String getNextLine(){
        if(inputLine == null)
            readInputLine()
```

```

while(inputIndex ==numberOfTokens)
    readInputLine()

def line =inputTokens [inputIndex..<numberOfTokens].join(' ')
inputIndex =numberOfTokens
return line
}

private static void readInputLine(){
    inputLine =System.in.readLine()
    inputTokens =inputLine.tokenize()
    numberOfTokens =inputTokens.size()
    inputIndex =0
}

//----properties -------

private static String inputLine      =null
private static List inputTokens     =null
private static int numberOfTokens   =0
private static int inputIndex       =-1
}

```

很显然，这个代码本应是更复杂的，只是故意的简化了。下面的范例演示了它的用法：

```

import console.*

def lastName =Console.readString()    //input:Barclay
println "lastName:${lastName}"        //lastName:Barclay

def name =Console.readString()        //input:Ken Barclay
println "name:${name}"               //name:Ken

name =Console.readLine()             //input:NONE!!!
println "name:${name}"               //name:Barclay

```

运行此程序时，结果如下：

```

Barclay
lastName:Barclay
Ken Barclay
name:Ken
name:Barclay

```

第一行是用户的输入值，`readString`方法的第一次调用把它作为参数。值由变量`lastName`存储，随后在第二行中输出。第二次调用`readString`时，缓冲器的输入如第三行所示。在这里，它将`Ken`作为输入，并赋值给变量`name`，并把它输出到第四行。最后，调用`readLine`方法时，并不需要用户输入新值，这是由于缓冲器中仍然保留着第三行输入的余下部分。`name`变量的值如最后一行所示。

附录G 关于方法的更多信息

能够调用方法本身的方法通常称为递归方法。在方法体中直接调用方法的过程，称之为直接递归。如果某个方法调用了另外一个方法，而在另一个方法中又要依次调用该方法本身的过程，称之为间接递归。递归是一种表示许多优秀程序解决方案的强有力的方法。

在本附录中，演示了递归方法和方法的其他方面信息，比如，使用静态类型参数和返回值。

G.1 递归方法

方法可以调用任何其他的方法。程序员经常利用这种方式构建分层程序，以在方法中调用子方法执行相关辅助任务。这些子方法也可能继续调用更多的子方法，以执行更为简单的任务等。

这并不是使用方法的唯一途径。特别地，方法可以调用它本身，这样的方法被称为递归。许多程序的解决方案都可以以直接或者间接递归的方式表述。将这些解决方案映射到递归方法中，实现起来通常会更优雅、更自然。

为了演示其效果，请思考计算某个数的阶乘的方法。正整型值的阶乘用 $n!$ 表示，它的结果为从1到n之间所有连续正整型值的乘积。0的阶乘是个特例，它被定义为1。这样：

$n! = n * (n - 1) * (n - 2) * \dots * 3 * 2 * 1$

且

$0! = 1$

这样可以推导出：

$n! = n * ((n - 1) * (n - 2) * \dots * 3 * 2 * 1)$

于是：

$n! = n * (n - 1)!$

范例01中的factorial方法实现了此功能。请注意观察if语句依据参数n的值选择返回值的方法。

范例01 递归方法factorial

```
def factorial(n){\n    if(n ==0)\n        return 1\n    else\n        return n *factorial(n-1)\n}\n\ndef fact5 =factorial(5)\nprintln "factorial(5):${fact5}"
```

其输出结果如下所示：

```
factorial(5): 120
```

对许多列表处理程序来说，它们的公有方法包括head、tail和cons。它们分别返回列表中的第一个项目（如果列表为空，则返回null），当第一个条目被删除后返回一个新列表，以及在它的头部增加一个新项目后返回一个新列表。范例02演示了这些方法的使用效果。随后这些方法用来开发upTo和prod方法。方法upTo返回一列整型值作为指定方法的参数。方法prod用来计算列表中所有整型值的乘积。upTo和prod都是递归方法。最后，upto和prod可用来实现另一个版本的factorial方法。

范例02 列表处理方法

```
def head(list){  
    return (list.size()==0)?null:list[0]  
}  
  
def tail(list){  
    def size =list.size()  
    return (size ==0)?[] :list[1..<size]  
}  
  
def cons(item,list){  
    def copy =list  
    copy.add(0,item)  
    return copy  
}  
  
def upTo(m,n){  
    if(m >n)  
        return []  
    else  
        return cons(m,upTo(m +1,n))  
}  
  
def prod(list){  
    if(head(list)==null)  
        return 1  
    else  
        return head(list) * prod(tail(list))  
}  
  
def factorial(n){  
    return prod(upTo(1,n))  
}  
  
.println "factorial(5):${factorial(5)}"
```

这个程序的输出结果和前一个范例的输出结果相同。

G.2 静态类型

在方法的定义中，已经使用了动态类型的参数和返回值。这个特征使方法更具一般性，如范例03所示。在这里，方法times返回的结果是它的两个参数之积。程序通过不同的实参类型三次调用了这个方法。由于“*”操作符可以在所有给定的情形中使用，因此所有的执行结果都和期望值一致。

范例03 动态类型方法

```
def times(x,y){  
    return x * y  
}  
  
println "times(3,4):${times(3,4)}"  
println "times(3.1,4.2):${times(3.1,4.2)}"  
println "times('Hello',4):${times('Hello',4)}"
```

程序的输出结果如下所示。

```
times(3,4):12  
times(3.1,4.2):13.02  
times('Hello',4):HelloHelloHelloHello
```

方法的返回值类型和形参类型也可以静态的指定。当我们希望重新定义来自Java超类的方法时需要使用这种方法。在这种情形下，必须指定相关的实际类型。这样的静态类型方法也引入了安全元素，这是因为这些方法在没有所需类型的实参时不能被调用。

范例04重新实现了前一个范例，但是这次为方法times静态地定义了类型。由于实参的类型不正确，因此最后的两次调用出现错误。

范例04 静态类型方法

```
int times(int x,int y){  
    return x * y  
}  
  
println "times(3,4):${times(3,4)}"  
//println "times(3.1,4.2):${times(3.1,4.2)}"           //missing method exception  
//println "times('Hello',4):${times('Hello',4)}"       //missing method exception
```

如果我们希望允许方法times接收数值型值参数，那么我们需要放宽对整型值参数和整型值返回值的限制，并使用Number类替换整型值参数和返回值，类Number为所有数字值类型的超类。范例05更新了前一个范例。

范例05 指定类型的方法

```
Number times(Number x,Number y){  
    return x * y  
}  
  
println "times(3,4):${times(3,4)}"  
println "times(3.1,4.2):${times(3.1,4.2)}"
```

随后我们将构造一个重新定义范例01引入的factorial方法的强制性案例，新方法使用单个整型值参数且返回一个整型值结果。如范例06所示，在这里，如果调用factorial(5.1)的话，程序将报告，没有名为factorial的方法定义了BigDecimal类型的形参。

范例06 指定静态类型的factorial方法

```
int factorial(int n){
    if(n == 0)
        return 1
    else
        return n * factorial(n-1)
}

println "factorial(5):${factorial(5)}"
println "factorial(5.1):${factorial(5.1)}" //missing method exception
```

G.3 实参协议

调用方法中的实参必须精确匹配方法定义中给定的形参数数。否则，Groovy将抛出方法缺少参数的异常。范例07中的最后两个方法调用将抛出此错误。这两个方法中，第一个方法调用的实参数目过少，而第二个方法调用的参数数目过多。

范例07 缺少参数的方法

```
def meth(a,b,c ){
    println "meth(${a},${b},${c})"
}

meth(1,2,3)
meth(1,2)           //missing method
meth(1,2,3,4)       //missing method
```

G.4 方法重载

Groovy支持方法重载。重载的方法是指在同一作用域内，和另一个方法的方法名相同，但是参数的个数不同的方法。当调用这种类型的方法时，程序通过参数匹配的方式调用正确的方法。通常使用实参的个数确定调用的方法。范例08演示了此用法。

范例08 方法重载

```
def times(x,y){
    return x * y
}

def times(x){
    return 2 * x
}
```

```
println "times(3,4):${times(3,4)}"
println "times(3):${times(3)}"
```

在这里，重载后的方法times含有两个或者一个参数。第一个调用提供了两个实参，因此将调用第一个实现方法。第二个调用使用了第二个定义。此应用程序的输出结果证实了这个行为：

```
times(3, 4): 12
times(3): 6
```

G.5 默认参数值的不确定性

方法的默认参数值和方法的重载一样可能会导致不确定性。请思考范例09中的代码。

范例09 默认参数值的不确定性

```
def times(x,y =4){
    return x * y
}

def times(x){
    return 2 * x
}

println "times(3,5):${times(3,5)}"
println "times(3):${times(3)}"
```

在这个范例中，方法times已经被重载，因此所调用的方法将由实参的个数决定。另外，此方法的第一个版本的第二个参数有默认值。这将潜在地和此方法的第二个定义版本产生冲突。第一个调用，times(3,5)将调用方法的第一个版本。第二个调用，times(3)可能会被解释为第二个参数为默认参数值的第一个版本的方法，也可能会被解释为含有单个参数的第二个版本。Groovy通过优先选择精确匹配参数个数的方法解决这种潜在的不确定性，因此程序的输出结果如下所示：

```
times(3, 5): 15
times(3): 6
```

当然，这种类型的不确定性能够通过使用静态类型参数来解决。请思考范例10中的代码：

范例10 指定静态参数类型的方法

```
def times(String str,num =1){
    return str * num
}

def times(x){
    return 2 * x
}

println "times('Hello',3):${times('Hello',3)}"
println "times('Hello'): ${times('Hello')}"
```

```
println "times(3):${times(3)}"
println "times(1.2):${times(1.2)}"
```

执行此脚本时，输出结果如下所示：

```
times('Hello', 3): HelloHelloHello
times('Hello'): Hello
times(3): 6
times(1.2): 2.4
```

请注意，第二行中的输出说明在只有一个String参数的地方也并不存在不确定性。

G.6 参数和返回值类型为集合的方法

下一个范例将引入接收文本行列表为输入，并将每个行分隔成独立的单词的方法split。程序随后将把这些单词按词典顺序排序。

范例11 将行列表分隔为单词

```
def split(lines){
    def words = []
    lines.each {line ->
        def wordsInLine = line.tokenize()
        words.addAll(wordsInLine)
    }
    return words
}

def doc = ['This is the first line','This is the second line','This is the third line']
def words = split(doc).sort()

words.each {word ->
    println "${word}"
}
```

运行此程序的输出结果如下所示：

```
This
This
This
first
is
is
is
line
line
line
second
the
the
the
third
```

单词索引是将来自某篇文本的单词按字母顺序排列的列表。文本被当作是一系列包含单个单词的行。索引列表中的每个单词将列出单词出现的行序号。

范例12 单词索引

```
import java.util.*

def concordance(lines){
    def lineNumber =1
    def concord =[ :]
    lines.each {line ->
        def wordsInLine =line.tokenize()
        wordsInLine.each {word ->
            if(concord [word ]==null)
                concord [word ]=[lineNumber ]
            else
                concord [word ] <<<lineNumber
        }
        lineNumber++
    }
    return concord
}

def printConcordance(concordance){
    def words =concordance.keySet().sort()
    words.each {word ->
        print "${word}"
        concordance [word ].each {lineNumber ->print "${lineNumber}"}
        println()
    }
}

def doc =[ "This is the first line ", "This is the second line ", "This is the third line "]
def concord =concordance(doc)
printConcordance(concord)
```

concordance方法将在每个行文本中操作。每个行将使用tokenize方法分隔成单个的单词。如果单词不在索引中，则此单词和它的行号将放入索引映射中。如果单词已经存在，则将行号追加到表示此单词的已经存在的行序号列表中。当整个索引过程结束后，方法printConcordance将安排以词典顺序输出。我们通过映射中的关键字为它们排序，随后通过已经排序的关键字访问映射。程序的输出结果如下所示：

```
This:1 2 3
first:1
is:1 2 3
line:1 2 3
second:2
the:1 2 3
third:3
```

在下一个范例中，数据值（可能从数据文件中获取）表示所有成员根据权利要求得到的个人费用。数据以下面的文本表示，它们分别表示提出要求人的姓名、要求得到的金额以及支出这笔费用的原因。

```
John 45.00 Train
Ken 102.20 Air
etc
```

处理此数据和获取所有成员要求得到费用总和的方法如范例13所示。

范例13 费用

```
def totalExpenses(expenseLines){
    def total =0
    expenseLines.each {expenseLine ->
        def expense =expenseLine.tokenize()
        total +=expense [1 ].toDouble()
    }
    return total
}

def expensesData =  ['John'      45.00   'Train',
                     'Ken'       102.20  'Air',
                     'Sally'     22.20   'Supplies'
]
println "Total expenses:${totalExpenses(expensesData)}"
```

我们可能会考虑将支出的数量分开，如将102.20分成两个独立的值102和20，它们分别表示美元数和美分数，这样所有的数值运算都使用整型值参数。在这种情形下，将首先把每个行分隔为三个组成部分，然后使用正则表达式获取表示美元和美分值的数字组。代码如下所示。

范例14 费用

```
def totalExpenses(expenseLines){
    def total =0
    expenseLines.each {expenseLine ->
        def expense =expenseLine.tokenize()
        def matcher =expense [1 ]=~'([0-9 ]* )\\.( [0-9 ]*)'
        matcher.matches()
        total +=100 * matcher [0 ][1 ].toInteger()+matcher [0 ][2 ].toInteger()
    }
    return total /100
}

def expensesData = ['John'      45.00   'Train',
                     'Ken'       102.20  'Air',
                     'Sally'     22.20   'Supplies'
]
println "Total expenses:${totalExpenses(expensesData)}"
```

附录H 关于闭包的更多信息

本书第9章介绍了闭包，以及当迭代处理集合时闭包的重要性。同时，还讨论了闭包的一般特性，包括闭包属性。在本附录中，会进一步讨论闭包的细节内容，包括默认参数、闭包和方法的差别，以及闭包和作用域规则。我们还列出List、Map和Range类中把闭包作为参数的方法。

H.1 闭包和不明确性

第9章的范例06展示，当闭包作为某方法调用的最后一个实参时，可以从实参列表中删除，并把它直接放在方法调用右括号的后面。如果meth表示具有三个参数的一个方法，如下所示：

```
def meth(a, b, c) {...}
```

其中，clos是一个闭包变量，{...}是闭包体。假设x和y是两个随意设置的值，则查看如下不同的方法调用语句：

```
meth(x, y, {...})           // OK
meth(x, y) {...}            // OK
meth(x, y, clos)            // OK
meth(x, y) clos             // ERROR: no such method
```

第2行代码把闭包体直接放在方法调用参数的后面。第4行代码说明，同样的做法也许不适合闭包变量。Groovy解释器无法认定clos标识符是该方法调用的一部分。Groovy解释器会报错，说明找不到只具有两个参数的meth方法。

如果meth表示具有一个参数的方法：

```
def meth(c) {...}
```

则clos是个闭包变量，{...}是闭包体。请查看如下方法调用语句。

```
meth(...)                  //OK
meth(){...}                 //OK
meth {...}                 //OK
meth(clos)                 //OK
meth()clos                //ERROR:null pointer exception
meth clos                  //OK
```

在第5行代码中，调用meth方法时不使用任何实参。在该方法体中，形参将被初始化为null，在这个方法中使用闭包变量将会导致系统报错。但是有点奇怪的是，最后一个范例可以正常运行。

H.2 闭包和方法

对于如下两个Groovy结构，必须非常清楚：

```
def double = {n -> return 2 * n}
def double(n) {return 2 * n}
```

前者定义一个闭包，后者是个方法定义。本书7.6部分引入作用域的概念。任何闭包引用肯定有固定的作用域。下面两个闭包不可能同时出现：

```
def divide = {x, y -> return x / y}
def divide = {x -> return 1 / x}
```

Groovy解释器会提示divide已经被定义，不能重复定义。相反地，知道Groovy支持方法重载，如下语句是能够运行的：

```
def multiply(x, y) {return x * y}
def multiply(x) {return 2 * x}
```

H.3 默认参数

像方法一样，闭包的形参也许会被分配默认值，下面就是一个简单的例子：

```
def greeting = {message, name = "world" -> println "${message} ${name}"}

greeting("Hello", "world")
greeting("Hello")
```

上述两个闭包调用会产生输出“Hello world”。

H.4 闭包和作用域

本书9.1节介绍了闭包的作用域，并在范例04到07中进行展示。第9章中的范例19使用一个局部闭包来实现一个选择性排序算法。下面范例01的bubbleSort闭包采取类似的方式。

范例01 闭包BubbleSort

```
def bubbleSort = {list ->

    def swap = {values,j,k ->
        def temp = values [j]
        values [j] = values [k]
        values [k] = temp
    }

    def size = list.size()
    def numberSorted = 0

    while(numberSorted < size){
        for(index in 1..<(size -numberSorted)){
            if(list [index ] < list [index -1 ])
                swap(list,index,index -1 )
        }
        numberSorted ++
    }

    return list
}
```

```
def numbers =[13,14,11,12,14 ]
println "Sorted numbers:${bubbleSort(numbers)}"
```

在相同的作用域中，一个变量不能被两次定义。在下面的代码中，不能定义局部闭包swap，原因是它的形参list会在当前作用域（bubbleSort闭包的闭包体）中引入一个具有相同名称的变量。

```
def swap = {list, j, k ->
    def temp = list[j]
    list[j] = list[k]
    list[k] = temp
}
```

但是，虽然存在相同作用域规则，但是如下代码允许我们定义swap。这时候，使用两个参数来描述swap的行为，并且list变量被定义在封闭的作用域中。

```
def swap = {j, k ->
    def temp = list[j]
    list[j] = list[k]
    list[k] = temp
}
```

最后一点需要注意的是，闭包只能由语句组成。这意味着，局部闭包swap不能使用方法定义来代替。

H.5 递归闭包

附录G（参见G.1）引入递归方法，也就是一个方法可以调用自己。出现这种情况的原因是方法的方法体可以引用自己。Groovy并不支持在闭包定义中引用自己。但是，当我们认识到一个闭包是类Closure的一个对象时，则闭包体可以使用this关键字来引用自己。这样做的结果是能够实现递归闭包，如范例02所示，其中factorial是个闭包。

范例02 递归的factorial闭包

```
def factorial = { n ->
    return (n == 0) ? 1 : n * this.call(n - 1)
}

println "Factorial(5): ${factorial(5)}"
```

输出结果是：

Factorial(5): 120

因为在闭包体中不允许引用自己，这意味着在Groovy中如下代码是非法的：

```
def factorial = { n ->
    return (n == 0) ? 1 : n * factorial(n - 1)
}
```

H.6 状态类型

可以使用参数和返回值的动态类型来定义闭包。这个特性使得闭包更加通用，如范例03所

示。在该范例中，闭包times返回两个参数相乘的结果。对闭包的三次调用使用不同的实参类型。因为“*”操作符适用于所有指定的环境，所以可以允许正常。

范例03 闭包的动态类型

```
def times = {x, y ->
    return x * y
}

println "times(3, 4): ${times(3, 4)}"
println "times(3.1, 4.2): ${times(3.1, 4.2)}"
println "times('Hello', 4): ${times('Hello', 4)}"
```

也可以静态地指定闭包形参的类型和闭包的返回类型。这种静态指定类型的闭包会引入安全问题，原因是如果实参的类型不是指定的类型或者子类型的话，就不能调用闭包。

范例04与其一个范例类似，唯一区别是times闭包的形参类型是静态指定的。运行上述程序，第三个调用就会报错，原因就是实参的类型不匹配。

范例04 静态指定参数类型的闭包

```
def times = {Number x, Number y ->
    return x * y
}

println "times(3, 4): ${times(3, 4)}"
println "times(3.1, 4.2): ${times(3.1, 4.2)}"
//println "times('Hello', 4): ${times('Hello', 4)}"
```

H.7 有关实参的约定

在调用闭包时，实参的数量必须严格匹配闭包定义时的形参的数量。否则，Groovy解释器也会报错，指示参数不正确。范例05中最后两个闭包调用语句就会报错，会指示第一个闭包调用的实参太少，第二个闭包调用的实参太多。

范例05 实参数量不正确的闭包调用

```
def clos = {a, b, c ->
    "clos(${a}, ${b}, ${c})"
}

clos(1, 2, 3)
//clos(1, 2)           // missing closure
//clos(1, 2, 3, 4)     // missing closure
```

H.8 闭包、集合和范围

列表、映射和范围等类提供很多使用闭包作为参数的方法，这有利于迭代处理集合或者范围中的每个元素，并执行指定的任务。表H-1列出常见的一些方法。Inject和reverseEach方法只

适用于列表和范围。H-1列表中的星号 (*) 说明对应的方法是新增的GDK方法。

表H-1 迭代器方法

方法名称	方法原型/说明
any*	boolean any(Closure clos) 迭代处理集合的每个元素，并检查clos标识的谓词是否适用于至少其中一个元素
collect*	List collect(Closure clos) 迭代处理这个集合，并使用clos转为转换器，把每个元素转换为一个新的值，并返回转换后的值列表
collect *	List collect(Collection collection, Closure clos) 迭代处理这个集合，并使用clos转为转换器，把每个元素转换为一个新的值，接着把转换后的值添加到这个集合中，最后返回最终的集合
each *	Void each(Closure clos) 迭代处理这个集合，并把闭包clos应用到每个元素
every *	boolean every(Closure clos) 迭代处理集合中每个元素，并检查闭包clos标识的谓词是否适合于所有元素
find *	object find (Closure clos) 从集合中找到匹配close闭包所标识谓词的第一个元素
findAll *	List findAll(Closure clos) 从集合中找到匹配close闭包所标识谓词的所有元素
findIndexOf *	int findIndexOf(Closure clos) 迭代处理集合中每个元素，并返回符合闭包clos所指定条件的第一个元素的索引
inject *	Object inject (Object value, Closure clos) 迭代处理这个集合，并把最初的值和第一个被迭代的元素传递给闭包clos，并把结果传入接下来的迭代
reverseEach *	void reverseEach (Closure clos) 反向迭代处理这个集合，并把闭包clos应用到每个元素
sort *	List sort (Closure clos) 把close闭包作为参照物，对这个集合进行排序

H.9 Return语句

我们需要注意，在闭包体中所使用return语句的语法。比如，比如，为决定某项数据是否是某列表的成员，通常会使用find方法。但是，应该考虑范例06中isMember方法两个版本所存在的问题。第一个版本isMemberA使用一个简单的for循环来迭代处理列表中的所有元素。如果发现一个匹配的元素，该方法会立即退出，并返回true值。如果遍历完，仍旧未发现匹配的元素，则该方法返回false值。下面的方法实现运行结果是正确的，通过前两个print语句的输出结果就可以看到。

范例06 闭包中的return语句

```
def isMemberA(item,list){
    def size =list.size()
    for(index in 0..<size){
        if(list [index ]==item)
            return true
    }
}
```

```

    }
    return false
}

def isMemberB(item,list){
    list.each {element ->
        //println "searching:${element}"
        if(element ==item)
            return true
    }
    return false
}

def numbers =[11,12,13,14 ]
println "isMemberA(15,numbers):${isMemberA(15,numbers)}" ///OK:false
println "isMemberA(13,numbers):${isMemberA(13,numbers)}" ///OK:true

println "isMemberB(15,numbers):${isMemberB(15,numbers)}" ///OK:false
println "isMemberB(13,numbers):${isMemberB(13,numbers)}" ///ERROR:false

```

我们也许想采用更像Groovy做法的方式——使用闭包作为each迭代器方法的参数来执行搜索。isMemberB方法就是这种形式的方法实现。但是，正如前两个print语句输出的那样，这个版本的方法返回false。如想知道为什么出现这种情况，就需要了解each方法的实现方式。正如如下代码所示，迭代器被用于检索列表中对对象的引用，并调用闭包clos（把每个obj作为实参）。

```

def each(list, clos) {
    def iter = list.iterator()
    while(iter.hasNext()) {
        def obj = iter.next()
        clos.call(obj)
    }
}

```

在我们的范例中，作为实参的闭包包含一个return语句。闭包返回true，就会导致while循环继续处理列表中下一个对象。如果不把方法isMemberB中print语句注释掉，就会看到具体的执行过程。你会发现，循环会继续执行，直到处理完列表中最后一个元素，这时isMemberB方法才会结束，一般情况下就会返回false。

H.10 测试

通常，经常会执行一些简单的测试，即使代码看起来非常简单，也非常符合Groovy规范。下面的GroovyTestCase就是经常见到的测试范例。基于这个测试范例，你可以增加更多。在此之前，应该确保自己完全了解闭包和集合。

```

import groovy.util.*
import java.util.regex.*

class GroovyIteratorTests extends GroovyTestCase {

    void setUp(){}

```

```
numbers =[11,12,13,14 ]
staffTelephones =['Ken' :2745,'John' :2746,'Sally' :2742 ]
century =2000..2099
}

//method any
void testAny(){
    assertTrue('One even value',numbers.any {element ->return (element %2 ==0)})
    assertTrue('Ken is staff member',staffTelephones.any {entry ->return (entry.key =='Ken')})
    assertTrue('This century all correct',century.any {element ->return
                                                (element >=2000 &&element <2100)})
}

void testCollect(){
    assertTrue('Doubled numbers',
               [22,24,26,28 ]==numbers.collect {element ->return 2 *element})
    assertTrue('Incremented telephone number',
               [2746,2747,2743 ].containsAll(staffTelephones.collect {entry ->
                                                return ++entry.value
                                            }))
    assertTrue('Next century',
               (2100..2199).containsAll(century.collect {element ->return 100 +element}))
}

void testEach(){
    def numbersResult =""
    numbers.each {element ->numbersResult =numbersResult +"+"+element}
    assertTrue('Numbers +',Pattern.compile ('(\\"+[0-9 ][0-9 ])').matcher(numbersResult).find())

    def staffTelephonesResult =""
    staffTelephones.each {entry ->
        staffTelephonesResult =staffTelephonesResult +"+"+entry.key
    }

    assertTrue('Names +',
               Pattern.compile ('(\\"+[A-Z ][a-z ]* )').matcher(staffTelephonesResult).find())

    def centuryResult =""
    century.each {element ->centuryResult =centuryResult +"+"+element}
    assertTrue('Numbers +',Pattern.compile ('(\\"+[0-9 ][0-9 ])').matcher(centuryResult).find())
}

void testEvery(){
    assertTrue('Every number 11..14',
               numbers.every {element ->return (element >=11 &&element <=14)})
    assertTrue('',staffTelephones.every {entry ->
        return [2745,2746,2742 ].contains(entry.value)})
    assertTrue('This century',
               century.every {element ->return (element >=2000 &&element <2100)})
}
```

```
void testFind(){}
    assertTrue('First is 11',11 ==numbers.find {element ->return element >10})
    assertTrue('Ken at 2745',
        2745 ==(staffTelephones.find {entry ->return (entry.key =='Ken')}).value)
    assertTrue('Last year',2099 ==century.find {element ->return (element ==2099)})
}

void testFindAll(){
    assertTrue('Last two',[13,14 ].containsAll (numbers.findAll {element ->
        return (element >12)}))
    assertTrue('Ken at 2745',
        1 ==(staffTelephones.findAll {entry ->return (entry.key =='Ken')}).size())
    assertTrue('',50 ==(century.findAll {element ->return (element >=2050)}).size())
}

void testFinIndexOf(){
    assertTrue('Position of 13',2 ==numbers.findIndexOf {element ->return (element >12)})
    assertTrue('Map indexing',
        staffTelephones.size()>staffTelephones.findIndexOf {entry ->return
            (entry.key =='Ken')})
    assertTrue('',99 ==century.findIndexOf {element ->return (element ==2099)})
}

void testInject(){
    assertTrue('Adding numbers',
        50 ==numbers.inject(0){previous,element ->return previous +element})
    assertTrue('All in century',
        century.inject(true){previous,element ->
            return (previous &&(2000 <=element &&element <=2099)})
    )
}

void testReverseEach(){
    def numbersResult =""
    numbers.reverseEach {element ->numbersResult =numbersResult +"+"+element}
    assertTrue('Numbers +',Pattern.compile (''(\\"+[0-9 ][0-9 ])*'').matcher(numbersResult).find())

    def centuryResult =""
    century.reverseEach {element ->centuryResult =centuryResult +"+"+element}
    assertTrue('Numbers +',Pattern.compile (''(\\"+[0-9 ][0-9 ])*'').matcher(centuryResult).find())
}

//-----properties -----
def numbers
def staffTelephones
def century
}
```

附录I 关于类的更多信息

本书在第12章引入类的概念，本附录会更加详细介绍那一章没有涉及的信息。我们将讨论属性的可见性、setter和getter方法，以及演示哪些方法名被预留以作为标准操作符的方法定义。同样，还会讨论Groovy提供的支持，以便于遍历对象网络。

I.1 属性和可见性

在前面章节中，已经见识到，借助于类可以大大缩减代码规模。Groovy尝试通过使用属性来统一实例字段和方法的概念。而且，还知道，如果类包含公共属性，就会自动生成相应的公共getter和setter方法。当使用命名属性创建一个类的实例时，就会自动调用这些方法，这样的话在定义类时就不必要定义参数化的构造器。

当重新阅读第12章引入的Account类时，就会看到上述所说的情况。下面的范例01重复实现上述过程。在如下代码中，创建Account类的一个实例，更改实例的状态，并通过访问其状态来获取属性值。上述方法的实现过程都是自动的。

范例01 Account类

```
class Account {  
  
    def credit(amount){  
        balance +=amount  
    }  
  
    def debit(amount){  
        //only if there are sufficient funds  
        if(balance >=amount)  
            balance -=amount  
    }  
  
    //----properties -----  
  
    def number  
    def balance  
}  
  
//Create an instance  
def acc = new Account(number :'ABC123',balance :1200)  
  
//Change state with the automatic setters  
acc.number = 'DEF456'
```

```
acc.balance =1500

//Now use the automatic getters
println "Account:${acc.number};balance:${acc.balance}"
```

运行范例01，就会得到如下输出：

```
Account: DEF456; balance: 1500
```

并演示说明Account对象被正确地初始化。

类的实例字段可以被定义为public（公开）、protected（保护）或者private（私有）等。正如字段修饰符表面意义那样，客户端代码可以直接访问public类型的实例字段。private类型的实例字段只能被定义它的类所访问。protected类型的实例字段可以被定义它的类和该类的子类所访问，其他的客户端代码不能访问。

使用这些可见性修饰符（visibility qualifiers），可以代码对实例字段的访问控制。把实例字段定义为公开的，客户端代码可以直接修改这些实例字段的值。当选择类的可见性特性时，通常我们会隐藏实现细节，只把那些支持类所表示抽象能力的特性公开化。这意味着，实例字段一般被定义为私有或者保护类型，而类的操作被定义为公开的。

如果Account类的实例字段被定义保护类型的，同样也会自动创建setter方法。这表示可以按照范例02的思路实现上一个范例。

范例02 保护类型的属性

```
class Account {

    //...

    def display(){
        println "Account number:${number}balance:${balance}"
    }

    //----properties ----

    protected number          //account number
    protected balance         //current balance
}

//Create an instance
//Since the setters are protected then the object instance
//is not correctly initialized
def acc = new Account(number : 'ABC123',balance :1200)
acc.display()

//Change things with the automatic setters
//acc.number ='DEF456'           //ERROR:Cannot access protected member
//acc.balance =1500              //ERROR:Cannot access protected member
```

运行范例02，产生如下输出：

```
Account number: null balance: null
```

我们已经创建一个Account对象，但是由于两个属性的getter和setter方法被设置为保护类型的，这个对象不能被正确地初始化。在默认情况下，没有被初始化的属性会被分配null值。因此产生上述输出。

请注意分析为什么不能访问实例字段。如果不注释掉更改账户编号和余额的代码，Groovy环境会报错，错误信息是“Cannot access protected member”。

不能给私有的或者保护的实例字段定义显性的getter和setter方法。如果只指定一个（公开的）getter方法，则实际上说明这个属性是只读的。如果只定义一个setter方法，则说明这个属性是只写的。范例01已经演示这种情况。

在范例03中，定义两个私有的实例字段。给balance实例字段提供一个getter方法，以便于实现对这个字段的只读访问。但是，正如代码所演示的那样，使用访问器方法（accessor）仍然会得到null值。其中原因不在于getter方法，而是缺失当创建这个对象时，与默认构造器一起工作的公开的setter方法。当运行如下脚本时，就会发现这个问题。

范例03 私有属性

```
Account balance:null

class Account {

    def getBalance(){
        return balance
    }
    //...

//-----properties -----
private number          //account number
private balance          //current balance
}

//Create an instance
//Since the setters are protected then the object instance
//is not correctly initialized
def acc = new Account(number :'ABC123',balance :1200)

//However, the balance property is read-only through the public getter
//But this produces the output:
//Account balance:null
println "Account balance:${acc.getBalance()}"
println "Account balance:${acc.balance}"
```

下面是一种解决思路。首先，给Account类提供一个参数化的构造器。这样就可以保证私有的实例字段被正确地初始化，访问器方法也能够抽取正确的值。解决方案见范例04。

范例04 参数化的构造器

```
class Account {
```

```
def Account(number,balance){  
    this.number =number  
    this.balance =balance  
}  
  
def getBalance(){  
    return balance  
}  
  
//...  
  
//-----properties -----  
  
private number          //account number  
private balance         //current balance  
  
//Create an instance  
def acc = new Account('ABC123',1200)  
  
//However, the balance property is read-only through the following  
println "Account balance:${acc.balance}"
```

运行范例04产生如下输出：

```
Account balance: 1200
```

范例05是另外一种解决思路。给Account类提供两个显性的setter方法。范例05会产生与范例04相同的输出。

范例05 公开的setter方法

```
class Account {  
  
    def getBalance(){  
        return balance  
    }  
  
    def setNumber(number){  
        this.number =number  
    }  
  
    def setBalance(balance){  
        this.balance =balance  
    }  
  
    //...  
  
    //-----properties -----  
  
    private number  //account number  
    private balance //current balance  
  
    //Create an instance
```

```

def acc = new Account(number : 'ABC123',balance :1200)

//However, the balance property is read-only through the public getter
println "Account balance:${acc.balance}"

```

I.2 对象导航

把属性与实例字段和方法集成起来会提供额外的特性。面向对象应用程序是相互交错的对象网络。对象实例形成关系，方法执行会覆盖整个的对象系统。对象之间的关系会产生类似于图论的结构。对于这种结构，一般需要采取某种遍历技术才能找到所需的对象，并调用所需的方法。

可以采用类似于Xpath的语法结构（诸如`http://www.w3.org/TR/xpath`）来遍历对象图。使用点来标识遍历步骤。下面举个例子，假设一个Bank被组织为Customer的一个集合，每个Customer都有多个Account（如图I-1所示）。我们也许希望产生与某Customer相关的所有Account对象的一个报告。很显然，需要引用在这个Bank中注册的该Customer所有Account对象。实现代码如下所示：

```
customers[customerNumber].accounts.each { number, account -> println "$account" }
```



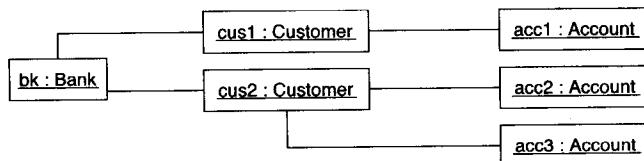
图I-1 Bank模型

使用上述导航形式的一个危险是，当遍历到一个null值时，程序会抛出`NullPointerException`异常。为避免出现这种情况，可以使用“`?.`”操作符提供的安全导航模式。上述代码重写如下：

```
customers[customerNumber]?.accounts.each { number, account -> println "$account" }
```

`customerNumber`被用于对Customer对象的Map进行索引，并获取所需的Customer。这个Customer对象的属性`accounts`引用这个Customer的所有Accounts。使用`each`方法迭代处理每个Account对象，应用于每个Account对象的闭包就是输出每个对象。

在开发类图之前，对象图(object diagram)是一种非常有用的数据分析工具。对象图可以展示在系统执行特定点时的对象网络。同样，页可以显示每个对象的状态。图I-2就是这样的一个对象图。在此，仅仅先是一个Bank对象，两个Customer对象，以及三个Account对象。这个对象图说明第一个Customer有一个Account，第二个Customer有两个Account对象。



图I-2 对象图

范例06进一步展示细节内容。请注意，在三个显示方法中对象导航的方式。

范例06 对象导航

```
class Account {  
    //...  
  
    String toString(){  
        return "Account:${number}${balance}"  
    }  
  
    //-----properties -----  
  
    def number          //account number  
    def balance         //current balance  
}  
  
class Customer {  
  
    def openAccount(number,balance){  
        def acc =new Account(number :number,balance :balance)  
        accounts [number ]=acc  
    }  
  
    String toString(){  
        return "Customer:${number}${name}"  
    }  
  
    //-----properties -----  
  
    def number          //account number  
    def name           //current balance  
    def accounts =[ :]  
}  
  
class Bank {  
    def registerCustomer(cust){  
        customers [cust.number ]=cust  
    }  
  
    String toString(){  
        return "Bank:${name}"  
    }  
  
    //-----properties -----  
    def name           //current balance
```

```
def customers =[ :]
}

def displayAllAccounts(bank){
    println "Bank:${bank.name}(all accounts)"
    println '====='

    bank?.customers.each {customerNumber, customer ->
        println customer
        customer?.accounts.each {accountNumber, account ->println " ${account}" }
    }
    println()
}

def displayAllAccountsForCustomer(bank, customerNumber){
    println "Bank:${bank.name}(customer accounts)"
    println '====='
    def customer =bank?.customers [customerNumber ]
    println customer
    customer?.accounts.each {number,account ->println " ${account}" }
    println()
}

def displayAccountForCustomer(bank, customerNumber, accountNumber){
    println "Bank:${bank.name}(customer account)"
    println '====='

    def customer =bank?.customers [customerNumber ]
    def account =customer?.accounts [accountNumber ]
    println " ${account}"
    println()
}

//create a Bank...
def bk = new Bank(name :'Barclay')

//...and some customers with accounts
def cust1 = new Customer(number :111,name :'Savage')
cust1.openAccount(1111,1200)
cust1.openAccount(1112,400)
cust1.openAccount(1113,800)

def cust2 =new Customer(number :222,name :'Kennedy')
cust2.openAccount(2221,1000)
cust2.openAccount(2222,1400)

//now register customers with bank
bk.registerCustomer(cust1)
```

```

bk.registerCustomer(cust2)

    //print some reports
displayAllAccounts(bk)
displayAllAccountsForCustomer(bk,111)
displayAccountForCustomer(bk,222,2222)
执行范例06，就会得到如下输出：
Bank:Barclay (all accounts)
=====
Customer:111 Savage
    Account:1111 1200
    Account:1112 400
    Account:1113 800
Customer:222 Kennedy
    Account:2221 1000
    Account:2222 1400
Bank:Barclay (customer accounts)
=====
Customer:111 Savage
    Account:1111 1200
    Account:1112 400
    Account:1113 800

Bank:Barclay (customer account)
=====
    Account:2222 1400

```

I.3 静态成员

实例化和初始化对象，对于我们来说应该是非常熟悉的过程。其实也相当于说，对象开始拥有一组有实际意义的属性，属性的值定义了这个对象的状态。对于属性而言，还有可能是类数据，充当被所有对象实例共享的值。这样的类成员（class member）被声明为static（静态的），并且必须与类名一起被访问。在范例07的Point类中，定义一个static成员ORIGIN，同时也是被限定为public和final。final关键字说明ORIGIN属性的值在这个程序中不能被修改。

范例07 静态的数据成员

```

class Point {

    def move(deltaX,deltaY){
        x +=deltaX
        y +=deltaY
    }
    //----properties -----
    def x
}

```

```
def y

public final static ORIGIN = new Point(x :0.0,y :0.0)
}

def p = new Point(x :2.0,y :3.0)
p.move(1.0,1.0)
println "p:${p.x}, ${p.y}"

println "Origin:${Point.ORIGIN.x}, ${Point.ORIGIN.y}"
```

请注意，在最后的输出语句中，ORIGIN成员被引用的方式。在引用时必须加上所属类的类名。

static修饰符也可以修饰类的方法。同样，这样的方法不属于特定的对象实例。这种方法的行为完全由输入参数描述，与类当前的状态无关。JDK Math类中的算术方法sqrt就是这样一个例子，如下所示：

```
public class Math {

    public static double sqrt(double x) { ... }

    // ... others
}
```

范例08演示如何在类LineSegment的getLength方法定义中使用sqrt方法。同样，在使用sqrt方法时必须加上类名修饰符Math。

范例08 静态方法

```
class Point {
    //...
}

class LineSegment {

    def move(deltaX,deltaY){
        start.move(deltaX,deltaY)
        end.move(deltaX,deltaY)
    }

    def getLength(){
        def xDiff =start.x -end.x
        def yDiff =start.y -end.y
        return Math.sqrt(xDiff * xDiff +yDiff * yDiff)
    }

    //-----properties -----
    def start
    def end
}
```

```

def p = new Point(x :3.0,y :4.0)
def q = new Point(x :4.0,y :5.0)

def line = new LineSegment(start :p,end :q)
println "Line length:${line.getLength()}"

```

1.4 操作符重载

本书第2章曾经说明，在Groovy中所有事物都是一个对象。比如，整数字面值123实际上就是类Integer的一个实例。这个对象实例可以用于诸如123+456这样的表达式中，就像传统的算术运算一样。实际上，这个表达式是方法调用123.plus(456)的简写方式。上述算术表达式中第一个操作数是方法调用的接收者，而第二个操作数是传递给plus方法的参数。

对于预定义的操作符，Groovy支持操作符重载（operator overloading）。每个操作符被映射到一个特定的方法名。正如前面演示的那样，“+”操作符被映射到一个名为plus的方法。通过在我们的类中实现这些方法，就可以重载对应的方法，应用到我们的类中的对象。

在范例09中，在类Vector中演示了操作符重载，类Vector被用于表示数字值的一维数组。方法plus实现矢量的加法运算，方法multiply实现矢量的乘法运算，如下所示：

```

[a1, a2, a3, ...] + [b1, b2, b3, ...] = [a1 + b1, a2 + b2, a3 + b3, ...]
[a1, a2, a3, ...] * [b1, b2, b3, ...] = a1 * b1 + a2 * b2 + a3 * b3 + ...

```

其中，[...]代表一个Vector对象。

范例09 操作符重载

```

class Vector {
    def plus(vec){
        def res =[]
        def size =this.values.size()
        def vecSize =vec.values.size()
        if(size ==vecSize){
            for(index in 0..<size){
                res <<(values [index]+vec.values [index])
            }
        }
        return res
    }

    def multiply(vec){
        def prod =0.0
        def size =this.values.size()
        def vecSize =vec.values.size()
        if(size ==vecSize){
            for(index in 0..<size){
                prod +=values [index ]* vec.values [index ]
            }
        }
    }
}

```

```

        return prod
    }

//-----properties -----
def values =[]
}

def vec1 =new Vector(values :[1.0,2.0,3.0,4.0,5.0])
def vec2 =new Vector(values :[6.0,7.0,8.0,9.0,10.0])

println "plus:${vec1 +vec2}"
println "multiply:${vec1 * vec2}"

```

请注意，在最后一个输出语句中Vector对象的算术操作符用法。当运行上述脚本时，输出如下：

```

plus: [7.0, 9.0, 11.0, 13.0, 15.0]
multiply: 130.00

```

表I-1列出被重载的操作符。

表I-1 被重载的操作符

操作符	方法	描述
a+b	a.plus(b)	加法运算
a-b	a.minus(b)	减法运算
a*b	a.multiply(b)	乘法运算
a/b	a.divide(b)	除法运算
a++ ++a	a.increment(b)	前置加一和后置加--
a--- -a	a.decrement(b)	前置减一和后置减--
a == b	a==b	等于运算，比如比较是否是相同的对象
a<= >b	a.compareTo (b)	比较，比如-1 if a<b; 0 if a == b; +1 if a>b
a==b	a.equals(b)	等于运算
a!=b	! a.equals(b)	不等于运算
a<b	a.compareTo(b)<0	小于
a<=b	a.compareTo(b)<=0	小于或者等于
a>b	a.compareTo(b)>0	大于
a>=b	a.compareTo(b)>=0	大于或者等于
a[b]	a.get(b)	使用索引访问数据
a[b]=c	a.put(b,c)	使用索引进行赋值

I.5 调用方法

附录B描述了Groovy脚本如何被实现为一个类。比如，简单脚本println 'hello world'变成一个类（在文件Hello.java中）：

```

public class Hello extends Script {

    public static void main(String[] args) {

```

```

Hello h = new Hello();
h.run(args);
}

public void run(String[] args) {
    this.println('hello, world');
}
}

```

实际上，当使用如下形式调用这个方法时：

```
this.println('hello, world')
```

实际的调用过程是使用从Script类继承的invokeMethod方法。在此，上述语句的实际调用过程是：

```
this.invokeMethod('println', ['hello, world'] as Object[])
```

被调用的方法被指定为一个String参数，并且方法参数被定义为一个Object[]。范例10是一个演示范例，在实例化之后，Account对象的存款是200，借款是900。这时候，通过继承的invokeMethod方法来调用方法credit和debit。

范例10 方法调用

```

class Account {

    //...

    String toString(){
        return "Account:${number}with balance:${balance}"
    }

    //-----properties -----
    def number
    def balance
}

def acc =new Account(number :'AAA111',balance :1200)

acc.invokeMethod('credit',[200] as Object [])
acc.invokeMethod('debit',[900] as Object [])

println "acc:${acc}"

```

如果在Account类中重新定义invokeMethod方法，就可以监控对Account对象的方法调用如何最终执行invokeMethod方法的。在范例11中，创建Account类的一个实例，并调用伪方法credit和debit。

范例11 重新定义的invokeMethod方法

```

class Account {
    public Object invokeMethod(String name,Object params){

```

```

    println "invokeMethod(${name},...)"
    return null
}

String toString(){
    return "Account:${number}with balance:${balance}"
}

//-----properties -----
def number
def balance
}

def acc =new Account(number :'AAA111',balance :1200)

acc.credit(200)
acc.debit(900)

println "acc:$acc"

```

当运行范例11时，输出如下所示：

```

invokeMethod(credit, ...)
invokeMethod(debit, ...)
acc: Account: AAA111 with balance: 1200

```

在此，就可以看到credit和debit方法调用的处理过程。

这种处理模式可以充当元对象协议（meta-object protocol）的基础（请参考<http://www-128.ibm.com/developerworks/java/library/j-pg09205/>和<http://www2.parc.com/csl/groups/sda/projects/mops/default.html>）。当在运行时，借助于这个协议，对象能够做出明确选择来影响自己的状态或者行为。在附录K的Ant构建器部分，提供一个有关这方面的展示。

I.6 习题

1. 请重复实现第12章中的七个练习（参见12.3部分），把属性的可见性设置为私有的。
2. 构造一个Matrix类，以表示数字值的二维数组。提供plus和multiply方法以重载该类所属对象的算术操作符。定义如下：

$$[[a_{00}, a_{01}, a_{02}, \dots], [a_{10}, a_{11}, a_{12}, \dots], \dots]$$

$$+ [[b_{00}, b_{01}, b_{02}, \dots], [b_{10}, b_{11}, b_{12}, \dots], \dots] = [[a_{00} + b_{00}, a_{01} + b_{01}, \dots], [a_{10} + b_{10}, a_{11} + b_{11}, \dots], \dots]$$

和

$$[[a_{00}, a_{01}, a_{02}, \dots], [a_{10}, a_{11}, a_{12}, \dots], \dots]$$

$$\cdot [[b_{00}, b_{01}, b_{02}, \dots], [b_{10}, b_{11}, b_{12}, \dots], \dots] = [[a_{00} * b_{00} + a_{01} * b_{01} + \dots, a_{00} * b_{01} + a_{01} * b_{11} + \dots], [a_{10} * b_{10} + a_{11} * b_{11} + \dots, a_{10} * b_{11} + a_{11} * b_{12} + \dots], \dots]$$

附录J 高级闭包

在第9章中，闭包被作为一个参数化的代码块，可以被引用，也可以被当作方法的参数。关于作为方法参数的典型范例是用于迭代处理集合。我们知道，诸如each和findAll等方法可以遍历集合中所有元素，并执行闭包指定的操作。

本附录将进一步研究闭包的特性，闭包可以从方法中返回，也可以作为其他闭包的返回值。闭包能够作为其他闭包的返回值，这个特性是解决很多问题的强大解决方案。比如，也许想定义应用于每个对象的约束，以及存在于对象或者规则之间的关系（也许应用于不同分类的对象）。通过闭包可以非常容易实现。

假设，Account的属性有账户号码、账户余额，以及账户透支额。我们也许认为透支额应该是个正值，比如透支额是500英镑。而且，应该保证账户余额应该大于或者等于-500英镑，也就是说账户余额是个受限的负值。我们将使用闭包来实现这些约束。

同样，假设某家超市使用特殊的商品价格来吸引新客户，以及维护已有的客户。在一周期内，也许早餐谷类食品的价格优惠1/3；在另一周期内，优惠政策可能变化，化妆品销售按照买一送一的方式进行销售。在这种优惠政策经常变化的环境中，需要使用某种灵活的方式来定义这些“业务规则”，并且能够灵活地修改和调整。在本章将展示如何实现这种灵活性。

此处所说的思路来源于多范式（multiparadigm）方式，这是Groovy的一个重要特性。本附录将演示在Groovy中，如何实现函数编程（functional programming）（Thompson, 1999）思想。借助于更高级函数，函数编程范式综合了多态（polymorphism）、组合（composition），以及计算（computation）模式。现在开始展示如何在Groovy中成功应用这些思想。

J.1 简单闭包

以前，把闭包描述为一个代码块。闭包实现参数化更加有价值，可以被引用，可以被作为参数传入方法，以及可以与call消息一起被调用。范例01包含一个简单的参数化闭包，名为multiply，用于计算两个数值参数的乘积。

范例01 简单闭包

```
def multiply = { x, y -> return x * y |  
  
    println "multiply(3, 4): ${multiply.call(3, 4)}"      // explicit call  
    println "multiply(3.4, 5.6): ${multiply(3.4, 5.6)}"   // implicit call
```

执行上述程序，演示闭包的执行情况，并输出如下结果：

```
multiply(3, 4): 12  
multiply(3.4, 5.6): 19.04
```

本书第9章曾经说明，闭包可以引用对象的状态。在范例02中，在multiply闭包的定义范围

内有个multiplier变量。这个闭包计算一个参数和multiplier的乘积。

范例02 作用域和闭包

```
def multiplier = 2
def multiply = {x -> return x * multiplier}      //second operand from enclosing scope

println "multiply(3):${multiply.call(3)}"
println "multiply(5.6):${multiply(5.6)}"

//Now do it again but with a different multiplier value
multiplier = 3

println "multiply(3):${multiply.call(3)}"
println "multiply(5.6):${multiply(5.6)}"
```

当运行范例02，得到如下输出：

```
multiply(3): 6
multiply(5.6): 11.2
multiply(3): 9
multiply(5.6): 16.8
```

这时候，看到在闭包定义范围内的变量可以从闭包代码内访问。

下一个范例演示闭包如何返回另一个闭包，作为闭包返回值。在范例03中，基于String参数，闭包arithmetic会从四个闭包中选择一个闭包，作为返回值。同时，这个程序演示被返回的闭包接下来可以像平常一样被调用。

范例03 闭包返回值

```
//Various closures
def add = {x,y -> return x + y}
def subtract = {x,y -> return x - y}
def multiply = {x,y -> return x * y}
def divide = {x,y -> return x / y}

//Select a closure
def arithmetic = {arith ->
    switch(arith){
        case 'ADD'      :return add
        case 'SUBTRACT' :return subtract
        case 'MULTIPLY' :return multiply
        case 'DIVIDE'   :return divide
        default         :return add
    }
}

//Get one...
def addOperation = arithmetic('ADD')
def mulOperation = arithmetic('MULTIPLY')
```

```
//...and use it
println "addOperation(3,4):${addOperation(3,4)}"
println "mulOperation(3,4):${mulOperation(3,4)}"

//Get one and use it
println "arithmetic('MULTIPLY')(3,4):${arithmetic.call('MULTIPLY').call(3,4)}"
```

下面是上述程序的输出。这个程序说明，arithmetic闭包的返回值是个闭包，这个闭包接下来可以像其他闭包一样被调用。

```
addOperation(3, 4): 7
mulOperation(3, 4): 12
arithmetic('MULTIPLY')(3, 4): 12
```

J.2 部分应用

在范例02中，multiply闭包计算一个参数和内部变量multiplier的乘积。现在，如果把multiplier作为闭包的参数，这个闭包返回multiply闭包，后者计算自己的参数与multiplier的乘积，这就是闭包部分应用（partial application）的范例。这个范例是常见的包含两个参数的闭包（参见范例04中的multiply），两个参数可以被重新设计，并被部分应用到一个参数。这是组合和合并闭包的强大方式。范例04的代码如下所示。

范例04 部分应用

```
def multiply = { x, y -> return x * y }

        // Now some partial applications...
        // ...both are closures
def triple = multiply.curry(3)
def quadruple = multiply.curry(4)

        // Both are partial applications of multiply
println "triple(4): ${triple(4)}"
println "quadruple(5): ${quadruple(5)}"
```

运行这个程序会得到如下输出：

```
triple(4): 12
quadruple(5): 20
```

请注意，表达式multiply.curry(3)表示把参数乘以3的闭包。然后，被返回的闭包使用一个参数进行调用，诸如triple(4)。在数学家Haskell Curry之后，闭包的部分应用被称为修正（currying）。闭包triple的定义如下所示：

```
def triple = { y -> return 3 * y }
```

其中，第一个参数被删除，所有出现第一个参数的地方使用数字3来代替。

加法和乘法被称为可交换（commutative）操作。这意味着， $A+B=D+A$ 和 $A*B=B*A$ 。但是，减法和除法不是可交换的操作。我们可以实现类似于multiply闭包，如范例05所示。

范例05 实现可交换操作

```
def rSubtract = { y, x -> return x - y }
def lSubtract = { x, y -> return x - y }

def subtract10 = rSubtract.curry(10)
def subtractFrom20 = lSubtract.curry(20)

println "subtract10(22): ${subtract10(22)}"
println "subtractFrom20(14): ${subtractFrom20(14)}"
```

范例05的输出如下所示：

```
subtract10(22): 12
subtractFrom20(14): 6
```

关于修正闭包的一个重点是，curry方法的实参数量绝对不能超过该闭包所需的参数实际数量。比如，如果该闭包有三个参数，则调用curry方法时可以使用零个、一个、两个或者三个实参。

下一节将演示闭包部分应用的一个优势。部分应用可以被认为是一种简化形式，是个复杂任务可以被分解为多个子任务。在multiply（参见范例04）的部分应用中，没有定义两个数值的乘法，而是把它分解，以便于定义任意数量的乘法运算闭包，比如triple或者quadruple。借助于这些更简单的任务，现在考虑如何组合使用它们。

J.3 组合

结构化程序的一种组织方式是顺序执行多个任务。通常而言，每部分任务被分别设计和实现。在此，也许认为，一个闭包表示需要执行的某个简单任务。使用组合（composition）概念进行任务合并，可以非常容易构造出复杂的任务。而且，借助于不同的组合方式，可以较快地创建超市所需的新任务，就像本附录刚开始时说明的案例。

范例06演示了composition闭包。这个闭包有两个参数f和g，两个参数代表闭包，把闭包g应用到x（比如g(x)），把闭包f应用到处理结果（比如f(g(x)))。

范例06 闭包组合

```
// Composition closure
def composition = { f, g, x -> return f(g(x)) }

// Multiply closure and two instances
def multiply = { x, y -> return x * y }

def triple = multiply.curry(3)
def quadruple = multiply.curry(4)

// Construct a new closure by combining two others
def twelveTimes = composition.curry(triple, quadruple)

println "twelveTimes(12): ${twelveTimes(12)}"
```

范例06的输出结果如下所示：

```
twelveTimes(12): 144
```

闭包triple有一个参数，并把这个参数与3相乘。闭包twelveTimes是闭包triple和quadruple的组合。结果是，闭包twelveTimes把其参数乘以4，然后把乘法结果与3相乘，全部过程定义在composition闭包中。

相同的组合可以在任意地方使用，包括应用到集合的元素。范例07就是这样的一个范例。

范例07 组合和集合

```
// Composition closure
def composition = {f, g, x -> return f(g(x))}

// Multiply closure and two instances
def multiply = {x, y -> return x * y}

def triple = multiply.curry(3)
def quadruple = multiply.curry(4)

// Construct a new closure by combining two others
def twelveTimes = composition.curry(triple, quadruple)

def table = [1, 2, 3, 4].collect {element -> return twelveTimes(element)}

println "table: ${table}"
```

上述代码把twelveTimes闭包应用到列表[1,2,3,4]的每个元素，并产生一个新的列表，如下所示：

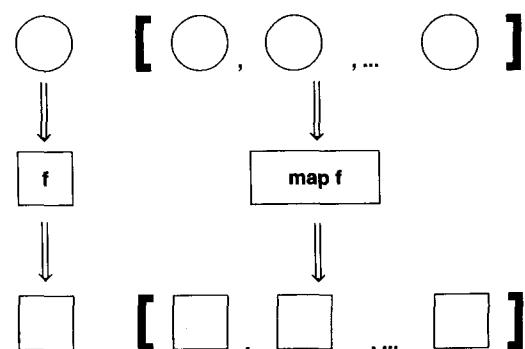
```
table: [12, 24, 36, 48]
```

J.4 计算模式

本节将介绍一种机制，使得闭包包含一种计算模式（pattern of computation）。一个范例是对列表中每个元素按照某种形式进行转换。当然，

Groovy已经给List类定义了collect方法，可以使用这个方法来简化模式的实现。这种转换通常被命名为map。图J-1描述了这种性质。左边是函数f（闭包），当把这个闭包应用到一个参数时（使用一个圆形来表示），会得到一个新的值（就是正方形）。现在，如果拥有值类型为圆形的一个列表，通过应用f，map f会产生值类型为正方形的一个列表。

在范例08中，定义map闭包。map的部分应用将返回操作单个列表的一个闭包。修正map闭包是通过应用于列表中每个元素的闭包来实现。



图J-1 应用map

范例08 映射

```
// map closure
def map = {clos, list -> return list.collect(clos)}

// composition closure
def composition = {f, g, x -> return f(g(x))}

// Multiply closure and two instances
def multiply = {x, y -> return x * y}

def triple = multiply.curry(3)
def quadruple = multiply.curry(4)

// closure to triple the elements in a list
def tripleAll = map.curry(triple)

def table = tripleAll([1, 2, 3, 4])

println "table: ${table}"
```

首先，研究一下map闭包。map闭包需要两个参数，一个闭包和当前闭包将应用的元素列表。定义闭包tripleAll以把triple闭包映射到列表的所有元素。上述程序的输出如下所示。

```
table: [3, 6, 9, 12]
```

关于映射的一个等级模式是，如果把闭包（比如f）映射到列表x，然后把闭包（比如g）应用到前一个闭包的处理结果。那么上述这个过程就等价于把g和f的组合应用到这个列表x。范例09演示了这种等价处理关系。

范例09 等价处理关系

```
// map closure
def map = { clos, list -> return list.collect(clos) }

// composition closure
def composition = { f, g, x -> return f(g(x)) }

// Multiply closure and two instances
def multiply = { x, y -> return x * y }

def triple = multiply.curry(3)
def quadruple = multiply.curry(4)

// composition of two maps...
def composeMapMap = composition.curry(map.curry(triple), map.curry(quadruple))

def tableComposeMapMap = composeMapMap([1, 2, 3, 4])

println "tableComposeMapMap: ${tableComposeMapMap}"

// ...equivalent to the map of a composition
def mapCompose = map.curry(composition.curry(triple, quadruple))
```

```
def tableMapCompose = mapCompose([1, 2, 3, 4])

println "tableMapCompose: ${tableMapCompose}"
```

上述代码的输出说明我们的假设是正确的：

```
tableComposeMapMap: [12, 24, 36, 48]
tableMapCompose: [12, 24, 36, 48]
```

J.5 业务规则

现在我们考虑如何计算一个Book的净价，在计算过程要考虑到商店折扣和政府税金，比如增值税（VAT，value added tax）。VAT是有关居民消费的税金。在商品或者服务提供的商业活动中需要考虑进来。如果打算把这部分逻辑作为Book类的一部分，就很可能导致解决方案被固化了。书店也许会更改折扣比例，或者折扣优惠政策只适用于一部分书籍。同样，政府部门也有可能修改税金标准。

关于这个问题，范例10提供基于闭包的解决方案的源代码。

范例10 净价计算

```
//Book class and instance
class Book {

    def name           //properties
    def author
    def price
    def category

}

def bk = new Book(name : 'Groovy',author : 'KenB',price :25,category : 'CompSci')

//constants
def discountRate = 0.1
def taxRate = 0.17

//basic closures
def rMultiply ={y,x ->return x *y }

def lMultiply ={x,y ->return x *y }

def composition ={f,g,x ->return f(g(x))}

//book closures
def calcDiscountedPrice =rMultiply.curry(1 -discountRate)

def calcTax =rMultiply.curry(1 +taxRate)
def calcNetPrice =composition.curry(calcTax,calcDiscountedPrice)

//now calculate net price
def netPrice = calcNetPrice(bk.price)

println "netPrice:${netPrice}"
```

闭包rMultiply是部分应用的一种候选解决方案，通过使用常量类型的第二个操作符，把二元的乘法修改为一元的闭包。两个有关书籍的闭包calcDiscountedPrice和calcTax都是rMultiply闭包的实例。闭包calcNetPrice是计算净价的算法，首先计算折扣价，然后添加税金，最后，把calcNetPrice应用到所选定图书的定价。输出结果是：

```
netPrice: 26.325
```

范例11的目的是，确保书店的最大折扣在一个上限值之内。因此，必须比较上限值和折扣量，并取最小值来计算折扣价。范例11如下所示。

范例11 有上限的折扣

```
//Book class and instance
class Book {

    def name           //properties
    def author
    def price
    def category

}

def bk =new Book(name :'Groovy',author :'KenB',price :35,category :'CompSci')

//constants
def discountRate = 0.1
def taxRate =0.17
def maxDiscount =3

//basic closures
def rMultiply = {y,x ->return x * y }
def tMultiply = {x,y ->return x * y }

def subtract = {x,y ->return x - y }
def rSubtract = {y,x ->return x - y }
def tSubtract = {x,y ->return x - y }
    //minimum closure
def min ={x,y ->return (x <y)?x :y }

    //identity closure
def id ={x ->return x }

    //composition closures
def composition ={f,g,x ->return f(g(x))}

    //binary composition
def bComposition ={h,f,g,x ->return h(f(x),g(x))}

    //book closures
def calcDiscount =rMultiply.curry(discountRate)
```

```

def calcActualDiscount =bComposition.curry(min,calcDiscount,id)

def calcDiscountedPrice =bComposition.curry(subtract,id,calcActualDiscount)

def calcTax =rMultiply.curry(1 +taxRate)

def calcNetPrice =composition.curry(calcTax,calcDiscountedPrice)

    //now calculate net price
println "bk.price:${bk.price}"

def netPrice =calcNetPrice(bk.price)
println "netPrice:${netPrice}"

```

首先，关注一元闭包id，返回单个参数的值。二元闭包min返回两个参数的最小者。bComposition（二元组合）闭包把一个二元闭包应用到两个一元闭包（应用于相同的值）产生的结果。部分闭包calcDiscount是个一元闭包，把书籍价格和折扣率相乘。calcActualDiscount闭包比较打折后的价格和上限价格。最后，闭包calcDiscountedPrice计算实际的折扣价格。

calcActualDiscount和calcDiscountedPrice都是bComposition闭包的部分应用。在第一个范例中，程序发现最小值；在第二个范例中，程序计算其中的差别。bComposition闭包被定义如下：

```
bComposition = {h, f, g, x -> return h(f(x), g(x))}
```

在此，f和g都是一元闭包，而h是二元闭包。请注意，f和g应用到相同参数x的方式。这个参数的实际值就是本书价格。下面是程序输出：

```

bk.price: 35
netPrice: 36.855

```

J.6 打包

上述范例开发了大量非常有价值的闭包，可以非常灵活地合并使用。因此，有必要把这些闭包打包到一个类中，这样的话更加容易引入到应用程序。这个类的最初版本是：

```

package fp
/**
 * The Functor class contains a series of static closures that
 * support functional programming constructs.
 */

abstract class Functor {

    //arithmetic (binary, left commute, and right commute)
    public static Closure bAdd      ={x,y ->return x + y}
    public static Closure rAdd      ={y,x ->return x + y}
    public static Closure lAdd      ={x,y ->return x + y}

    public static Closure bSubtract ={x,y ->return x - y}
    public static Closure rSubtract ={y,x ->return x - y}
    public static Closure lSubtract ={x,y ->return x - y}
}

```

```

public static Closure bMultiply ={x,y ->return x * y}
public static Closure rMultiply ={y,x ->return x * y}
public static Closure lMultiply ={x,y ->return x * y}

public static Closure bDivide ={x,y ->return x / y}
public static Closure rDivide ={y,x ->return x / y}
public static Closure lDivide ={x,y ->return x / y}

public static Closure bModulus ={x,y ->return x % y}
public static Closure rModulus ={y,x ->return x % y}
public static Closure lModulus ={x,y ->return x % y}

//min/max
public static Closure bMin ={x,y ->return (x <y)?x :y }
public static Closure bMax ={x,y ->return (x <y)?y :x }

//identity
public static Closure id ={x ->return x}
public static Closure konst ={x,y ->return y}

//composition
public static Closure composition ={f,g,x ->return f(g(x))}
public static Closure bComposition ={h,f,g,x ->return h(f(x),g(x))}

//lists
public static Closure head ={list ->return (list.size()==0)?null :list [0 ]}
public static Closure tail ={list ->
    return (list.size()==0)?[] :list [1..<list.size()]
}

public static Closure cons ={item,list ->
    def copy =list.clone()
    copy.add(0,item)
    return copy
}

public static Closure map ={action,list ->return list.collect(action)}
public static Closure apply ={action,list ->list.each(action)}
public static Closure filter ={predicate,list ->return list.findAll(predicate)}
public static Closure forAll ={predicate,list ->
    if(list.size()==0)
        return true
    else if(predicate(list [0 ]))
        return this.call(predicate,list [1..<list.size()])
    else
}

```

```

        return false
    }
    public static Closure thereExists
        = {predicate,list ->
            if(list.size()==0)
                return false
            else if(predicate(list[0]))
                return true
            else
                return this.call(predicate,list[1..<list.size()])
        }
    //others ...
}

```

这个类包含适用于算术、关系和逻辑运算符的闭包。下面继续关于可交换性的讨论。我们已经提供左右位置可交换的操作符的二元版本。闭包head、tail和cons重复实现第9章引入的方法，同时还包含闭包composition、map、filter等等。这个类被定义为抽象的，原因是当我们无意创建这个类的实例，它仅仅充当闭包的符号。

图J-2描述了filter闭包。如果p代表某个谓词函数（也就是返回一个布尔值的闭包），当应用到一个三角形时，它会返回false；当应用到一个正方形时，它返回true。现在，filter p list产生一个新的列表，其中只包含满足这些谓词条件的三角形。

现在，分析范例12的代码相对于以前，进一步简化了应用程序。第一个值得注意的差别是，被要求重新引入call方法；否则，Groovy编译器正常编译这部分代码。其次，bMin、id和bSubtract闭包必须是Functor类的成员。这个程序会产生与范例11相同的输出。

范例12 使用Functor类

```

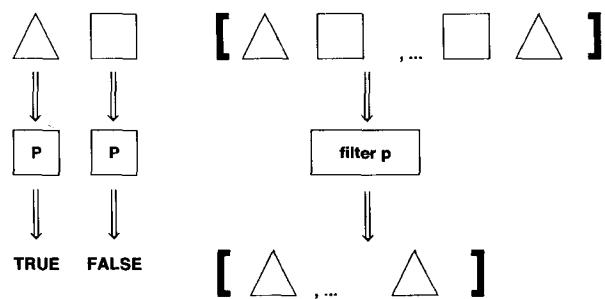
import fp.*

//Book class and instance
class Book {
    def name           //properties
    def author
    def price
    def category
}

def bk =new Book(name : 'Groovy',author : 'KenB',price : 35,category : 'CompSci')

//constants

```



图J-2 Filter闭包

```

def discountRate =0.1
def taxRate =0.17
def maxDiscount =3

    //book closures
def calcDiscount = Functor.rMultiply.curry(discountRate)

def calcActualDiscount = Functor.bComposition.curry(Functor.bMin,calcDiscount,Functor.id)

def calcDiscountedPrice = Functor.bComposition.curry(Functor.bSubtract,
                                                       Functor.id,calcActualDiscount)

def calcTax =Functor.rMultiply.curry(1 +taxRate)

def calcNetPrice =Functor.composition.curry(calcTax,calcDiscountedPrice)

    //now calculate net price
println "bk.price:${bk.price}"

def netPrice =calcNetPrice(bk.price)
println "netPrice:${netPrice}"

```

类Functor也包含一些作为计算模式的闭包，比如，map闭包被用于把一个行为（由一个闭包表示）应用到一个列表。范例13计算单词列表中每个单词的长度，并返回一个新的列表。

范例13 计算单词的长度

```

import fp.*

def size = { text -> return text.length() }

println "map(size, ['Edinburgh', 'Glasgow', 'Perth']): ${Functor.map.call(size,
['Edinburgh','Glasgow', 'Perth'])}"

```

这个范例产生如下输出：

```
map(size, ['Edinburgh', 'Glasgow', 'Perth']): [9, 7, 5]
```

Functor类也包含filter闭包。这个闭包向一个列表应用一个谓词条件（返回布尔值的闭包），并按照原来顺序返回满足指定谓词条件的所有元素的列表。范例14查找长度为3的所有单词，代码输出如下：

```
filter(isSize3, rhyme): [Fee, Fie, Fum]
```

范例14 Filter闭包

```

import fp.*

def isSize3 = {text -> return (text.length() == 3)}

def rhyme = ['Fee', 'Fie', 'Fo', 'Fum']

println "filter(isSize3, rhyme): ${Functor.filter.call(isSize3, rhyme)}"

```

由于修正的作用，使一些闭包可以共同处理列表的列表。范例15演示如何把一个filter映射到列表的列表。就像前一个范例一样，在Lists单词列表中查询长度为3的单词，并返回一个符

合条件的单词列表。

范例15 共同工作

```
import fp.*

def isSize3 = { text -> return (text.length() == 3) }

def rhyme = [['Fee', 'Fie', 'Fo', 'Fum'],
             ['I', 'smell', 'the', 'blood', 'of', 'an', 'Englishman']
            ]
println "map(filter(isSize3), rhyme): ${Functor.map.call(Functor.filter.curry(isSize3), rhyme)}"
```

运行上述程序，得到如下输出：

```
map(filter(isSize3), rhyme): [[Fee, Fie, Fo], [the]]
```

闭包thereExists (forAll) 显示列表中是否存在满足指定谓词条件的元素。它可以被视为列表的量词符。范例16演示了thereExists，范例17演示了forAll。

范例16 使用thereExists

```
import fp.*

def isSize3 = { text -> return (text.length() == 3) }
def rhyme = ['Fee', 'Fie', 'Fo', 'Fum']
println "thereExists(isSize3, rhyme): ${Functor.thereExists.call(isSize3, rhyme)}"
```

下面是这个范例的输出。我们看到至少有一个单词的长度是3。

```
thereExists(isSize3, rhyme): true
```

范例17 使用forAll

```
import fp.*

def isSize3 = { text -> return (text.length() == 3) }
def rhyme = ['Fee', 'Fie', 'Fo', 'Fum']
println "forAll(isSize3, rhyme): ${Functor.forAll.call(isSize3, rhyme)}"
```

运行范例17，会看到并不是所有的单词的长度都是3。

```
forAll(isSize3, rhyme): false
```

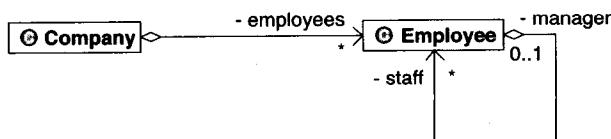
对于这些有关列表的量词，可以使用它们对模型的运作模式进行判定。图J-3是一个组织的类图。这个模型显示出，雇员被分配管理组成员的职责。我们希望实现的一个约束条件是，每个雇员的管理者必须是其他雇员。

在范例18中，雇员JonK负责管理KenB和JohnS。但是，JonK没有管理者。可以使用forAll闭包来确认这种不一致性。

范例18 模型约束

```
import java.util.*
import fp.*
```

```
class Employee |
```



图J-3 类图

```
def String toString(){
    return "Employee:${id}${name}"
}

def addToTeam(employee){
    staff [employee.id ]=employee
    employee.manager =this
}

//-----properties -------

def id
def name
def staff =[ :]
def manager =null
}

class Company {

    def hireEmployee(employee){
        employees [employee.id ]=employee
    }

//-----properties -------

    def name
    def employees =[ :]
}

def displayStaff(co){
    println "Company:${co.name}"
    println "===="
    co?.employees.each {entry ->println "${entry.value}"}
}

def co = new Company(name :'Napier')

def emp1 = new Employee(id :123,name :'KenB')
def emp2 = new Employee(id :456,name :'JohnS')
def emp3 = new Employee(id :789,name :'JonK')

co.hireEmployee(emp1)
co.hireEmployee(emp2)
co.hireEmployee(emp3)

emp3.addToTeam(emp1)
emp3.addToTeam(emp2)

displayStaff(co)
```

```

def hasManager = employee -> return (employee.manager != null)

def staff = co.employees.values().toList()

println "Every employee has a manager?: ${Functor.forAll.call(hasManager, staff)}"

    //Now make JonK a member of own team
emp3.addToTeam(emp3)
println "Every employee has a manager?: ${Functor.forAll.call(hasManager, staff)}"

```

当执行这段程序时，输出如下：

```

Company: Napier
= == == == == == == == == =
Employee: 789 JonK
Employee: 456 JohnS
Employee: 123 KenB
Every employee has a manager?: false
Every employee has a manager?: true

```

从倒数第二行输出，可以看到，没有满足我们设定的条件。谓词闭包hasManager判断指定的雇员已经被分配一个管理者。forAll闭包则是把这个谓词应用到所有的雇员。

J.7 列表简化

Functor类包含两个列表简化（list-reducing）闭包——rFold和lFold。这些闭包是通用计算模式的范例。假设，拥有列表[x₁,x₂, ..., x_n]，并且希望计算x₁+x₂+...+x_n。我们可以把上述表达式视为add(x₁, add(x₂, ..., add(x_n, 0))). 图J-4描述了闭包rFold。在左边是二元闭包f，接收参数是一个圆形值和一个正方形值，并产生一个正方形值。在此，rFold f e list 会产生一个正方形，其中参数e和list分别是一个正方形和圆形列表。图J-4的底下部分说明，闭包是如何从最右边元素开始处理的。

```

rFold = { f, e, list ->
    def size = list.size()
    def res = e
    for(index in 0..<size) {
        res = f(list[size -1 -index], res)
    }
    return res
}

```

掌握列表简化的关键是，参数f代表二元闭包，参数e代表基础值，参数list是要处理的列表。为计算整数列表的总和，会把Functor类中的bAdd方法作为二元闭包，整数字面值0作为基础值。范例19说明了这个过程。

范例19 列表简化

```

import fp._

//Closure sum adds the items in a list
def sum = Functor.rFold.curry(Functor.bAdd, 0)

```

```

println "sum:${sum([11,12,13,14])}"

def append =[list1,list2 ->
    def result =[]
    result.addAll(list1)
    result.addAll(list2)

    return result
]

//Closure flat flattens a list of lists
def flat = Functor.rFold.curry(append,[])

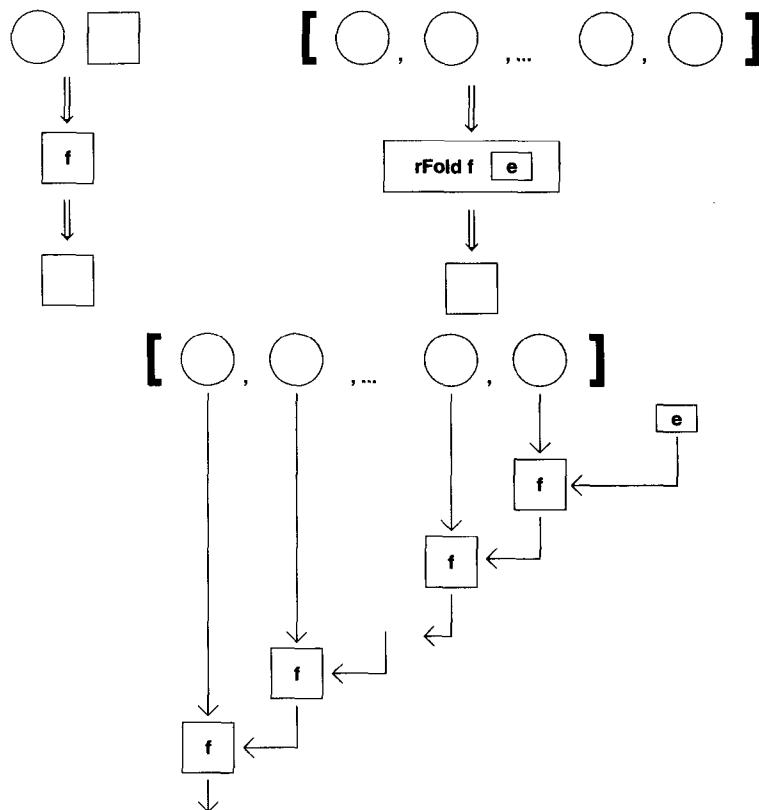
```

请注意，sum闭包是否与以前描述的一样。借助于这个程序定义的append闭包，也可以展开这个列表。在此，基础值是空的列表。输出结果如下所示：

```

sum: 50
flat: [11, 12, 13, 21, 22, 23, 24, 31, 32]

```



图J-4 rFold闭包

J.8 习题

1. 开发一个名为square的简单闭包，以计算单个参数的平方值。
2. 开发一个名为twice的简单闭包，以计算单个参数值的倍数。
3. 开发一个那个位isEven的简单闭包，判断其唯一的整数参数是否是偶数。
4. 使用本附录所开发的Functor类、闭包dec = Functor.rSubtract.curry(1)和inc = Functor.rAdd.curry(1)，以预测Functor.bMultiply.call(inc(4), dec(4))所产生的值。
5. 使用前一个练习中定义的闭包inc和dec，计算Functor.composition.curry(inc, dec).call(4)的值。
6. 对于闭包leftLT = Functor.lLt.curry(3)和rightLT = Functor.rLt.curry(5)，计算Functor.bComposition和curry(Functor.bAnd, leftLT, rightLT).call(4)的值。
7. 对于范例04中定义的multiply闭包，说明p、q和r分别代表什么，并演示如何使用它们。

```
def p = multiply.curry(2)
def q = multiply.curry(3, 4)
def r = multiply.curry()
```

8. 使用本附录提供的Functor类，解释闭包inc = Functor.lAdd.curry(1)。现在，说明由下属语句所定义的闭包的作用

```
Functor.map.curry(Functor.* composition.curry(inc, inc))
```

9. 使用类Functor中定义的head和tail，预测调用如下两个闭包会产生什么结果：

```
def hT = Functor.composition.curry(Functor.head.curry(), Functor.tail.curry())
def tT = Functor.composition.curry(Functor.tail.curry(), Functor.tail.curry())
```

10. 使用Functor类中定义的闭包，并把附录G中范例02的upTo方法重新定义为一个闭包。现在，定义闭包factors，这个闭包返回整数的一个列表，这些整数都是其整数参数的因数。

```
def factors = {n -> ...}
```

定义闭包isEmpty，如果其列表参数是个空列表时返回布尔值true。

```
def isEmpty = {list -> ...}
```

使用修正的（curried）组合，定义闭包prime；如果该闭包的整数参数是素数（只能被1和自己整除的整数），则返回布尔值true。现在使用prime来修正（curry）Functor类中filter闭包，然后获取2和50之间的素数。

11. 闭包insert会往一个已经排序的列表中插入一个新数据项，同时保证元素的既往顺序不会被改变：

```
def insert ={x,list ->
  def res = []
  if(list.size()==0){
    res <<x
  }else {
    def inserted =false
    for(element in list){
```

```
if(inserted ==false &&x <element){
    inserted =true
    res <<x
}
res <<element
}
if(inserted ==false){
    res <<x
}
}
return res
}
```

现在，定义闭包如下：

```
Functor.rFold.curry(insert, [])
```

请判断当把这个闭包应用到整数列表会得到什么输出。

12. 闭包xxx定义如下：

```
def xxx = Functor.rFold.curry(Functor.cons)
```

然后，请判断如下调用的输出值：

```
xxx([11, 12], [13, 14])
```

13. 闭包inc、xComp和xComposition定义如下：

```
def inc = { x -> return 1 + x }
def xComp = { h, f, x, y -> return h(f(x), y) }
def xComposition = xComp.curry(Functor.cons, inc)
```

闭包xxx定义如下：

```
xxx = Functor.rFold.curry(xComposition, [])
```

计算如下表达式的结果：

```
xxx([11, 12, 13, 14])
```

并判断Functor类的那个闭包等价于xxx。

14. 使用Functor类中的闭包head、tail和cons，开发一个闭包insert，并在排序后的列表中插入一个新的数据项：

```
def insert = { x, list -> ... }
```

现在，使用这个闭包以实现闭包insertSort，使用insertsort算法对列表中的值进行排序：

```
def insertSort = { list -> ... }
```

附录K 关于构造器的更多信息

本书的第19章和20章引入Groovy构造器的概念。从根本上讲，借助于Groovy构造器，可以很容易表示嵌套的类树数据结构，比如XML数据。借助于构造器，比如MarkupBuilder，可以容易地构造XML数据。使用SwingBuilder，也可以轻松地使用Swing组件构造一个GUI应用程序。

本附录进一步研究AntBuilder，使用它来构造Ant XML构造文件（Holzner,2005），并在无需关心XML情况下执行它们。同时，也会简单说明如果定制自己的构造器。在阅读本章之前，应该具备Ant的有关知识。

K.1 AntBuilder

Groovy提供AntBuilder类，可以使用它非常容易地构造和执行Ant XML构造文件。甚至在不必直接使用XML的前提下就可以达到这个目标。而且，正如我们前面说明的那样，可以把任何其他Groovy代码和AntBuilder代码集成到一起。范例01使用AntBuilder来创建一个目录，并把自己的文件复制到这个新目录中。

范例01 创建目录，并复制文件

```
import groovy.util.*  
  
def aB = new AntBuilder()  
  
aB.echo(message : 'Start')  
aB.mkdir(dir : 'demo')  
aB.copy(file : 'example01.groovy', todir : 'demo')  
aB.echo(message : 'End')
```

本范例是个简单的Ant构造文件，定义一个Ant项目。这个项目有一个或者多个目标，每个目标由多个Ant任务组成（<http://ant.apache.org/manual/index.html>）。范例01中的代码是默认的任务，并调用Ant核心任务echo、copy和mkdir。第一个任务反馈一个消息，第二个任务把文件复制到一个目录，第三个renweu创建一个新目录。

范例02说明把具有Groovy后缀的所有文件复制到新创建的目录下。首先，请注意，demoDir的Groovy定义代码与构造器代码混杂在一起。伪方法fileSet使用dir参数指定源目录，而include用于限制被复制的文件。

范例02 创建目录，并复制所有的Groovy文件

```
import groovy.util.*  
def aB = new AntBuilder()  
  
aB.echo(message : 'Start')
```

```
def demoDir = 'demo'

aB.mkdir(dir : demoDir)
aB.copy(todir : demoDir) {
    aB.fileSet(dir : '.') {
        aB.include(name : '*.groovy')
    }
}
aB.echo(message : 'End')
```

在范例03中，使用一个扫描器（类FileScanner，参见GDK），以查找新目录下的所有Groovy文件，并输出这些文件的文件名列表。

范例03 文件扫描

```
import groovy.util.*

def aB =new AntBuilder()

aB.echo(message :'Start')

def demoDir ='demo'

def scanner =aB.fileScanner(){
    aB.filesset(dir :demoDir){
        aB.include(name :'* .groovy')
    }
}

println "${demoDir}"
scanner.each {file ->
    println " ${file}"
}

aB.echo(message :'End')
```

如下范例演示了Groovy如何被用于构造Ant构造文件，就像一般的Groovy开发者所做的那样。提供目标以编译Groovy文件，执行脚本，运行单元测试，或者清除环境。在本书的后面几章中，本书作者使用它开发一个更大的应用程序。

这个列表定义Build类。被设计像简单的Ant构造文件一样被使用，可以使用它编译文件，执行脚本，以及完成随后的清除工作。通过重新定义invokeMethod方法，这个Build类使用附录I所引入的元对象协议（参见<http://www-128.ibm.com/developerworks/java/library/j-pg12144.html>）。一旦拥有这个类的实例，就可以调用伪方法compile、clean等等，比如如下代码：

```
def b = new Build()
b.clean()
```

invokeMethod方法会把请求转发到某闭包提供的实现。这个范例会调用clean闭包。从代码中，可以看出它已经删除临时文件和目录，代码如下：

```
def b = new Build()
```

```
b.compile()
```

上述代码调用compile闭包。它的代码依赖于init任务，因此它首先调用init闭包。init闭包会创建一些临时的工作目录，并定义新的Ant任务，基于此我们可以在运行时调用groovy编译器。在此之后，闭包compile会对源目录中所有的*.groovy文件执行Groovy编译器。

File:Build.groovy

```
package build
import groovy.util.*
import java.io.*
import java.util.*
class Build {
    public Object invokeMethod(String name, Object params){
        def target =targets [name]
        if(target !=null)
            target.call(params)
        else
            usage.call(params)
        return null
    }
}

//-----properties -----
def aB =new AntBuilder()
def ENV_CLASSPATH =System.getenv('CLASSPATH')
def ENV_GROOVY_HOME =System.getenv('GROOVY_HOME')

def BASEDIR ='.'
def SRCDIR =BASEDIR
def DESTDIR =BASEDIR +'/classes'
def REPDIR =BASEDIR +'/reports'

def GROOVLETSDIR =BASEDIR +'/src'
def GSPDIR =BASEDIR +'/src'

def COMMONDIR =BASEDIR +'/..common'
def WEBDIR =BASEDIR +'/web'
def BUILDDIR =BASEDIR +'/build'
def DEPLOYDIR =BASEDIR +'/deploy'

def WEBAPPSDIR =ENV_CATALINA_HOME +'/webapps'

def BASIC_CLASSPATH ='basic.classpath'
def basicClasspath =aB.path(id :BASIC_CLASSPATH){
    aB.pathelement(path :"${SRCDIR};${DESTDIR}")
    aB.pathelement(location :"${ENV_CLASSPATH}")
}
```

```
def COMPILE_CLASSPATH           ='compile.classpath'
def compileClasspath           =aB.path(id :COMPILE_CLASSPATH){
    aB.path(refid :BASIC_CLASSPATH)
}

def clean ={params ->
    aB.delete(){
        aB.fileset(dir :"${SRCDIR}",includes : '** /* .bak')
        aB.fileset(dir :"${SRCDIR}",includes : '** /* .BAK')
        aB.fileset(dir :"${SRCDIR}",includes : '** /* .txt')
    }
    aB.delete(dir :"${DESTDIR}")
    aB.delete(dir :"${REPDIR}")
    aB.delete(dir :"${BUILDDIR}")
}

def init ={params ->
    aB.taskdef(name :'groovyc',classname :'org.codehaus.groovy.ant.Groovyc')
    aB.taskdef(name :'groovy',classname :'org.codehaus.groovy.ant.Groovy')

    aB.mkdir(dir :"${DESTDIR}")
    aB.mkdir(dir :"${REPDIR}")
    aB.mkdir(dir :"${BUILDDIR}")
}

def compile ={params ->
    init.call(params)

    aB.groovyc(srkdir :"${SRCDIR}",destdir :"${DESTDIR}",classpath :"${basicClasspath}")
}

def run ={params ->
    compile.call(params)

    aB.groovy(src :params [1]){
        aB.classpath(){
            aB pathelement(path :"${SRCDIR};${DESTDIR}")
            aB pathelement(location :"${ENV_CLASSPATH}")
        }
    }
}

def test ={params ->
    compile.call(params)

    aB.junit(fork :'yes'){
        aB.classpath(){


```

```
aB.pathElement(path :"${SRCDIR};${DESTDIR}")
aB.pathElement(location :"${ENV_CLASSPATH}")
}

aB.formatter(type :'plain')

aB.batchTest(todir :"${REPDIR}" ){
    aB.fileSet(dir :"${DESTDIR}" ){
        aB.include(name : '** /* Test.class')
    }
}
}

def assemble =[params ->
    init.call(params)

    aB.copy(todir :"${BUILDDIR}" ){
        aB.fileSet(dir :"${COMMONDIR}" )
    }

    aB.copy(todir :"${BUILDDIR}" ){
        aB.fileSet(dir :"${GROOVLETSDIR}" ){
            aB.include(name : '** /* .groovy ')
            aB.include(name : '** /* .gsp ')
        }
        aB.fileSet(dir :"${BASEDIR}" ){
            aB.include(name : '** /* .html ')
        }
    }
}

def deploy =[params ->
    assemble.call(params)

    aB.copy(todir :"${WEBAPPSDIR}/${params [1]}" ){
        aB.fileSet(dir :"${BUILDDIR}" )
    }
]

def undeploy =[params ->
    aB.delete(dir :"${WEBAPPSDIR}/${params [1]}")
]

def db =[params ->
    aB.delete(dir :"${params [1]}DB ")
    aB.sql(url :"jdbc:derby:${params [1]}DB;create=true ",userid :"",password :"",

```

```
        driver : 'org.apache.derby.jdbc.EmbeddedDriver',src :"${params [1]}.sql ")
    }

def usage =[params ->
    aB.echo(message : "")
    aB.echo(message : 'Available targets:')
    aB.echo(message : "")
    aB.echo(message : 'clean:      Remove all temporary files/directories')
    aB.echo(message : 'compile:     Compile all source files')
    aB.echo(message : 'deploy:      Deploy the web application as a directory')
    aB.echo(message : 'init:       Prepare working directories')
    aB.echo(message : 'db:          Establish and populate the database')
    aB.echo(message : 'run:         Execute the named script')
    aB.echo(message : 'test:        JUnit tests')
    aB.echo(message : 'usage:       Default target')
    aB.echo(message : '')
]

def targets =[['clean'      :clean,
              'init'       :init,
              'compile'    :compile,
              'run'        :run,
              'test'       :test,
              'assemble'   :assemble,
              'deploy'     :deploy,
              'undeploy'   :undeploy,
              'db'         :db,
              'usage'      :usage
            ]
]
```

在本书第16章，执行过大量的单元测试，其中就用到test闭包。同样，在第24章使用deploy闭包来组装所需的所有文件，并部署到Tomcat服务器。

我们把一个简单的脚本作为驱动器。文件gbuild.groovy如下所示：

File:Build driver

```
/*
 * Usage:
 * groovy gbuild.groovy clean
 * groovy gbuild.groovy init
 * groovy gbuild.groovy compile
 * groovy gbuild.groovy test
 * groovy gbuild.groovy run script-file-name
 * groovy gbuild.groovy deploy project-name
 * groovy gbuild.groovy undeploy project-name
 * groovy gbuild.groovy db database-name
 * groovy gbuild.groovy usage
 *
```

```
* groovy gbuild.groovy default target:usage
*/
def b =new Build()

if(args.size()>0){
    def target =args [0]
    b.invokeMethod(target,args)
}else
    b.usage(args)
```

借助于这个Groovy脚本，可以使用这个Build类。比如使用如下代码：

```
groovy gbuild.groovy clean
```

来执行clean闭包。

本书第18章把数据库处理能力引入图书馆学习案例中。在那个例子中，建议一个数据库，其中有分别表示借阅者和出版物的表。使用典型的值来初始化这些表。文件library.sql列出这些信息：

```
create table borrowers(
    membershipNumber           varchar(10)not null,
    name                         varchar(20),
    primary key(membershipNumber)
);

create table publications(
    catalogNumber           varchar(10)not null,
    title                     varchar(40),
    author                    varchar(20),
    editor                    varchar(20),
    type                      varchar(8),
    borrowerID                varchar(10),
    primary key(catalogNumber),
    foreign key(borrowerID)references borrowers(membershipNumber)
);

insert into borrowers values('1234','Jessie');

insert into publications values('111','Groovy','KenB','','BOOK','1234');
insert into publications values('222','UML','JohnS','','BOOK',null);
insert into publications values('333','OOD','','JonK','JOURNAL',null);
```

db目标把这个文件名作为参数，并创建一个特名为libraryDB的数据库，使用这个文件名和DB文件名后缀。从第18章开始，一直使用如下代码：

```
groovy gbuild.groovy db library
```

来创建libraryDB数据库，并填充初始化数据。

K.2 专用的构造器

为创建一个名为MarkupBuilder或者AntBuilder的构造器，程序员必须实现groovy.util.BuilderSupport的一个子类。这个子类应该实现的方法包括：

```
void      setParent(Object parent, Object child);
Object    createNode(Object name);      //a node without parameter and closure
Object    createNode(Object name, Object value); //a node without parameters, but with closure
Object    createNode(Object name, Map attributes); //a node without closure but with parameters
Object    createNode(Object name, Map attributes, Object value); //a node without parameters, but with
                                                               closure and parameters
```

比如，当Groovy代码包含下述代码时，构造器会调用方法createNode(Object name, Object value)。

```
aB.demo() {
...
}
```

方法createNode的参数name就使用伪方法demo的方法名。参数value提供这个闭包及其内容。

除此之外，BuilderSupport类拥有两个（钩子）方法，子类也许会重新定义这两个方法，以实现定制的行为。方法getName就是一个钩子方法，允许把名称转换为其他某种对象，比如XML构造器中的限定性名称。借助于方法nodeCompleted，可以在处理完依赖性节点后处理这个节点。

```
void      nodeCompleted(Object parent, Object node);
Object    getName(String methodName);
```

范例04定义类MonitorBuilder，它是BuilderSupport的子类。类MonitorBuilder不执行任何实际性工作。但是，它定义一些抽象的方法，并在被调用时显示这些方法的名称和参数。这样的话，这个类就可以监控当处理构造器代码时这些方法在何处被调用。

范例04 MonitorBuilder类

```
class MonitorBuilder extends BuilderSupport {

    protected void setParent(Object parent, Object child){
        println "setParent(${parent}, ${child})"
    }

    protected Object createNode(Object name){
        println "createNode(${name})"
        return name
    }

    protected Object createNode(Object name, Object value){
        println "createNode(${name}, ${value})"
        return name
    }
}
```

```
protected Object createNode(Object name, Map attributes, Object value){  
    println "createNode(${name}, ${attributes}, ${value})"  
    return name  
}  
  
protected Object createNode(Object name, Map attributes){  
    return createNode(name, attributes, null)  
}  
  
protected void nodeCompleted(Object parent, Object node){  
    println "nodeCompleted(${parent}, ${node})"  
}  
}  
  
def mB = new MonitorBuilder()  
  
def monitor = mB.database(name : 'library'){  
    table(name : 'Book'){  
        field(name : 'title', type : 'text')  
        field(name : 'isbn', type : 'text')  
        field(name : 'price', type : 'integer')  
        field(name : 'author', type : 'id')  
        field(name : 'publisher', type : 'id')  
    }  
}
```

当运行这个程序时，会得到如下输出。

```
createNode(database, ["name": "library"], null)  
createNode(table, ["name": "Book"], null)  
setParent(database, table)  
createNode(field, ["name": "title", "type": "text"], null)  
setParent(table, field)  
nodeCompleted(table, field)  
createNode(field, ["name": "isbn", "type": "text"], null)  
setParent(table, field)  
nodeCompleted(table, field)  
createNode(field, ["name": "price", "type": "integer"], null)  
setParent(table, field)  
nodeCompleted(table, field)  
createNode(field, ["name": "author", "type": "id"], null)  
setParent(table, field)  
nodeCompleted(table, field)  
createNode(field, ["name": "publisher", "type": "id"], null)  
setParent(table, field)  
nodeCompleted(table, field)  
nodeCompleted(database, table)  
nodeCompleted(null, database)
```

上述输出的第一行和最后一行显示，被伪方法创建的database节点。在这两个事件之间，其他节点分别被创建和完成。在database节点内，有个table节点，table节点内有五个field节点。输出显示，可以截取createNode和nodeComplemented方法调用，以定制新的行为。比如，MarkupBuilder支持XML/HTML内容。

在第19章，范例08读取一个描述关系型表的XML文件，并把他转换为SQL语句，最终创建这个数据库表。在范例05中，使用定制的构造器来完成相同的任务。

范例05 定制的SQL构造器

```
import groovy.util.*
import java.io.*

class SqlBuilder extends BuilderSupport {

    protected void setParent(Object parent, Object child){
    }

    protected Object createNode(Object name){
        println "createNode(${name})"
        return name
    }

    protected Object createNode(Object name, Object value){
        println "createNode(${name}, ${value})"
        return name
    }

    protected Object createNode(Object name, Map attributes, Object value){
        this.processStartNode(name, attributes, value)
        return name
    }

    protected Object createNode(Object name, Map attributes){
        return createNode(name, attributes, null)
    }

    protected void nodeComplement(Object parent, Object node){
        this.processEndNode(parent, node)
    }

    private void processStartNode(Object name, Map attributes, Object value){
        switch(name){
            case 'database':
                out.println "DROP DATABASE IF EXISTS ${attributes.get('name')};"
                out.println "CREATE DATABASE ${attributes.get('name')};" 
                break
            case 'table'
```

```
        out.println "DROP TABLE IF EXISTS ${attributes.get('name'));"  
        out.println "CREATE TABLE ${attributes.get('name'))("br/>        out.print "${attributes.get('name'))}_ID INTEGER NOT NULL "  
        break  
    case 'field':  
        out.println ","  
        out.print "${attributes.get('name'))}${type To SQL [attributes.get('type'))}"  
        break  
    }  
}  
  
private void processEndNode(Object parent, Object node){  
    switch(node){  
        case 'table':  
            out.println()  
            out.println ');'  
            break  
    }  
}  
  
//-----properties -----  
  
def out  
def typeToSQL = ['text' : 'TEXT NOT NULL',  
                'id' : 'INTEGER NOT NULL',  
                'integer' : 'INTEGER NOT NULL']  
}  
  
def sB =new SqlBuilder(out :new File('db.sql').newPrintWriter())  
  
def sql =sB.database(name :'library'){  
    table(name :'Book'){  
        field(name :'title',type :'text')  
        field(name :'isbn',type :'text')  
        field(name :'price',type :'integer')  
        field(name :'author',type :'id')  
        field(name :'publisher',type :'id')  
    }  
}  
  
sB.out.flush()  
sB.out.close()
```

附录L 关于GUI构造器的更多信息

第20章使用SwingBuilder标记生成器引入了基本的Swing组件。目前来讲，在典型的图形化应用程序中，都能找到很多GUI组件，比如菜单、菜单项、工具条和对话框等。

L.1 菜单和工具条

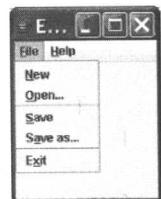
图形化应用程序经常会提供菜单，用户使用菜单可以选择希望执行的程序功能。一个菜单条通常包含多个菜单。每个菜单都使用下拉形式提供多个菜单项，分别表示应用程序能够提供的服务。使用SwingBuilder提供的menuBars、menus和menuItems，就可以非常容易实现上述说明的图形化功能。层次关系（menuBar）包含menu，每个menu包含多个menuItem。每个菜单有一个标识菜单名的文本标签和操作快捷键。menuItem也有文本标签、快捷键，以及表示选中时会执行相应动作的闭包。范例01是产生图L-1所示GUI的Groovy脚本。

范例01 简单的菜单

```
import groovy.swing.SwingBuilder
import javax.swing.*

//Create a builder
def sB = new SwingBuilder()

//Now the frame
def frame = sB.frame(title :'Example01',location :[100,100],
    size :[400,300],defaultCloseOperation :WindowConstants.EXIT_ON_CLOSE){
    menuBar {
        menu(text :'File',mnemonic :'F'){
            menuItem(){
                action(name :'New',mnemonic :'N',closure :{println 'File +New'})
            }
            menuItem(){
                action(name :'Open...',mnemonic :'O',closure :{println 'File +Open...'})
            }
            separator()
            menuItem(){
                action(name :'Save',mnemonic :'S',closure :{println 'File +Save'})
            }
        }
    }
}
```



图L-1 一个菜单

```

menuItem(){
    action(name :'Save as...',mnemonic :'A',closure :{println 'File +Save as...'})
}

separator()
menuItem(){
    action(name :'Exit',mnemonic :'X',closure :{System.exit(0)})
}
}

menu(text :'Help',mnemonic :'H'){
    menuItem(){
        action(name :'About',mnemonic :'A',closure :{println 'Help +About'})
    }
}
}

//Now show it
frame.pack()
frame.setVisible(true)

```

通过预先初始化的Groovy List，要实现菜单会更加容易。范例02使用一个List来构造菜单和菜单项重新实现前面的范例。在范例02代码中，菜单表示一个列表，每个菜单的菜单项也是列表。这些嵌套的列表能够完全表达每个菜单的细节。这些列表包含菜单的名称和快捷键，以及每个菜单项的名称、快捷键和处理程序。迭代器代码menus.each处理嵌套的列表，最终组装成一个完整的菜单条。

范例02 使用列表实现菜单

```

import groovy.swing.SwingBuilder
import javax.swing.*

//Menu handlers
def fileNew ={
    println 'File +New'
}

def fileOpen ={
    println 'File +Open...'
}

def fileSave ={
    println 'File +Save'
}

def fileSaveAs ={
    println 'File +Save as...'
}

```

```
def fileExit ={
    System.exit(0)
}

def helpAbout ={
    println 'Help +About'
}

//Create a builder
def sB = new SwingBuilder()

//Now the frame
def frame = sB.frame(title :'Example02',location :[100,100],
    size :[400,300],defaultCloseOperation :WindowConstants.EXIT_ON_CLOSE){
    menuBar {
        def fileMenu =[[['File',      'F'],
                      ['New',       'N',     fileNew],
                      ['Open...',   'O',     fileOpen],
                      null,
                      ['Save',      'S',     fileSave],
                      ['Save as...', 'A',   fileSaveAs],
                      null,
                      ['Exit',      'X',     fileExit]
                    ]
        def helpMenu =[[['Help',      'H'],
                      ['About',     'A',     helpAbout]
                    ]
        menus =[fileMenu,helpMenu]
        menus.each {mnu ->
            def mnuDetails =mnu [0]
            sB.menu(text :mnuDetails [0],mnemonic :mnuDetails [1]){
                for(k in 1..<mnu.size()){
                    def mnuItem =mnu [k]
                    if(mnuItem ==null)
                        sB.separator()
                    else
                        sB.menuItem(){
                            sB.action(name :mnuItem [0],mnemonic :mnuItem [1],closure :mnuItem [2])
                        }
                }
            }
        }
    }
}

//Now show it
frame.pack()
frame.setVisible(true)
```

在提供菜单条之外，大多数图形化应用程序还提供一个工具条。工具条通常位于应用程序的顶部，在菜单条之下。每个菜单条按钮充当一个菜单项的快捷方式。

图L-2是一个具有传统菜单和工具条的应用程序。用户已经打开Help菜单，其中的菜单项列表已经显示出来。通常而言，用户会从中选择一个菜单项，执行绑定到这个菜单项的动作。请注意，这个工具条拥有New、File和Save按钮。我们的代码仅仅演示菜单和工具条的构造过程，并不会执行任何实际的动作。当然，按照早期的范例要求，肯定会提供合适的处理器程序。

范例03是应用程序的代码清单。请注意菜单条条和工具条的组装方式。菜单条和工具条的内容保存在嵌套的列表中。这样做可以简化菜单和工具条的构建和修改过程。同时请关注，第二个迭代器menus.each使用文本标签和处理器程序创建和初始化工具条的方式。这样的话，就可以把每个处理器程序与每个菜单项和对应的工具条按钮关联起来。

范例03 菜单和工具条

```
import groovy.swing.SwingBuilder
import javax.swing.*
import java.awt.*

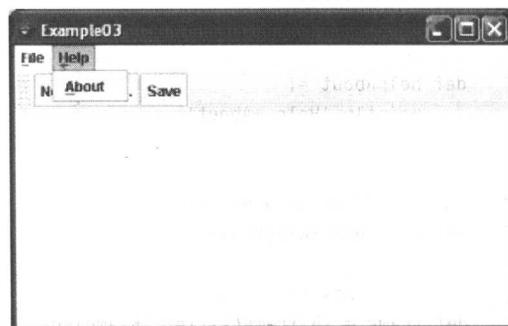
//Text area of set size
class FixedTextArea extends JTextArea {

    Dimension getMinimumSize(){return TEXTAREASIZE }
    Dimension getMaximumSize(){return TEXTAREASIZE }
    Dimension getPreferredSize(){return TEXTAREASIZE }
    private static final TEXTAREASIZE =new Dimension(400,400)
}

//Menu handlers
def fileNew ={
    println 'File +New'
}

def fileOpen ={
    println 'File +Open...'
}

def fileSave ={
    println 'File +Save'
}
```



图L-2 菜单条和工具条

```
def fileSaveAs ={
    println 'File +Save as...'
}

def fileExit ={
    System.exit(0)
}

def helpAbout ={
    println 'Help +About'
}

//Create a builder
def sB = new SwingBuilder()

//Now the frame
def frame = sB.frame(title :'Example03',location :[100,100],
    size :[400,300],defaultCloseOperation :WindowConstants.EXIT_ON_CLOSE){
    def fileMenu = [[['File','F'],
        ['New','N',true,fileNew],
        ['Open...', 'O', true, fileOpen],
        null,
        ['Save', 'S', true, fileSave],
        ['Save_as...', 'A', false, fileSaveAs],
        null,
        ['Exit', 'X', false, fileExit]
    ]]
    def helpMenu = [[['Help','H'],
        ['About', 'A', false, helpAbout]
    ]]

    def menus =[fileMenu,helpMenu]

    menuBar {
        menus.each {mnu ->
            def mnuDetails =mnu [0]
            sB.menu(text :mnuDetails [0],mnemonic :mnuDetails [1]){
                for(k in 1..<mnu.size()){
                    def mnuItem =mnu [k]
                    if(mnuItem ==null)
                        sB.separator()
                    else {
                        sB.menuItem(){
                            sB.action(name:mnuItem [0],mnemonic :mnuItem [1],closure:mnuItem [3])
                        }
                    }
                }
            }
        }
    }
}
```

```

sB.panel(layout :new BorderLayout()){
    toolBar(constraints :BorderLayout.NORTH){
        menus.each {toolMnu ->
            for(k in 1..<toolMnu.size()){
                def toolItem =toolMnu [k]
                if(toolItem !=null &&toolItem [2]==true){
                    def toolText =toolItem [0]
                    def toolAction =toolItem [3]
                    sB.button(text :toolText,actionPerformed :toolAction)
                }
            }
        }
    }
    sB.panel(constraints :BorderLayout.CENTER){
        widget(new FixedTextArea(enabled :false))
    }
}
}

//Now show it
frame.pack()
frame.setVisible(true)

```

L.2 对话框

这个范例展示如何在应用程序中引入一个对话框。在范例04的代码清单中，实现Swing类JDialog的子类，以提供自己定制的对话框。这个对话框被填充两个文本字段，以获取用户名和密码，这个界面在很多应用程序中都可以见到。程序的执行结果见图L-3，范例04是代码清单。

范例04 提供对话框的应用程序代码

```

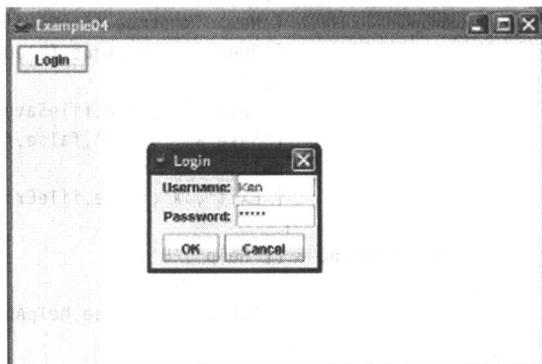
import groovy.swing.SwingBuilder
import javax.swing.*
import java.awt.*

//Text area of set size
class FixedTextArea extends JTextArea {

    Dimension getMinimumSize(){return TEXTAREASIZE }
    Dimension getMaximumSize(){return TEXTAREASIZE }
    Dimension getPreferredSize(){return TEXTAREASIZE }

    private static final TEXTAREASIZE =new Dimension(400,400)
}

```



图L-3 被激活的对话框

```
//Builder
def sB = new SwingBuilder()
def loginDialog =null

    //frame handlers
def loginHandler ={
    loginDialog.setVisible(true)
}

    //Now the main panel...
def mainPanel ={
    sB.panel(layout :new BorderLayout()){
        panel(constraints :BorderLayout.WEST){
            button(text :'Login',actionPerformed :loginHandler)
        }
        panel(constraints :BorderLayout.CENTER){
            widget(new FixedTextArea(enabled :false))
        }
    }
}

//...and finally the frame
def frame = sB.frame(title :'Example04',location :[100,100],
    size :[400,300],defaultCloseOperation :WindowConstants.EXIT_ON_CLOSE){
    mainPanel()
}

    //dialog handlers
def okHandler ={
    loginDialog.setVisible(false)

    def userName =nameField.getText()
    def userPassword =passwordField.getText()

    nameField.setText('')
    passwordField.setText('')

    if(userName =='')
        println 'NO user name given'
    else if(userPassword =='')
        println 'NO password given'
    else
        println 'User:${userName}'
}

def cancelHandler ={
    loginDialog.setVisible(false)
```

```
userName =
userPassword =

nameField.setText('')
passwordField.setText('')
}

def dialogPanel ={
    sB.panel(layout :new BorderLayout()){
        panel(layout :new GridLayout(2,2,5,5),constraints :BorderLayout.CENTER){
            label(text :'Username:',horizontalAlignment :JLabel.RIGHT)
            def nameField =textField(text :'',columns :20)
            label(text :'Password:',horizontalAlignment :JLabel.RIGHT)
            def passwordField =passwordField(text :'',columns :20)
        }
        panel(constraints :BorderLayout.SOUTH){
            button(text :'OK',actionPerformed :okHandler)
            button(text :'Cancel',actionPerformed :cancelHandler)
        }
    }
}

loginDialog =sB.dialog(owner :frame,title :'Login',size :[160,120],modal :true){
    dialogPanel()
}

//Now show it
frame.pack()
frame.setVisible(true)
```

[General Information]

书名 = G R O O V Y 入门经典

作者 = (英) Kenneth Barclay John Savage 著

页数 = 352

S S 号 = 11854354

出版日期 = 2008.1

封面

书名

版权

前言

目录

第1章 Groovy

- 1 . 1 为什么使用脚本语言
- 1 . 2 为什么使用Groovy

第2章 数值和表达式

- 2 . 1 数值
- 2 . 2 表达式
- 2 . 3 运算符优先级
- 2 . 4 赋值
- 2 . 5 自增和自减运算符
- 2 . 6 对象引用
- 2 . 7 关系运算符和等于运算符
- 2 . 8 习题

第3章 字符串和正则表达式

- 3 . 1 字符串字面值
- 3 . 2 字符串索引和索引段
- 3 . 3 基本操作
- 3 . 4 字符串方法
- 3 . 5 比较字符串
- 3 . 6 正则表达式
- 3 . 7 习题

第4章 列表、映射和范围

- 4 . 1 列表
- 4 . 2 列表方法
- 4 . 3 映射
- 4 . 4 映射方法
- 4 . 5 范围
- 4 . 6 习题

第5章 基本输入输出

- 5 . 1 基本输出
- 5 . 2 格式化输出
- 5 . 3 基本输入
- 5 . 4 习题

第6章 学习案例：图书馆应用程序（建模）

- 6 . 1 迭代1：需求规范和列表实现
- 6 . 2 迭代2：映射实现
- 6 . 3 习题

第7章 方法

- 7 . 1 方法
- 7 . 2 方法参数
- 7 . 3 默认参数
- 7 . 4 方法返回值
- 7 . 5 参数传递
- 7 . 6 作用域
- 7 . 7 集合作为参数和返回值
- 7 . 8 习题

第 8 章 流程控制

- 8 . 1 w h i l e 语句
- 8 . 2 f o r 语句
- 8 . 3 i f 语句
- 8 . 4 s w i t c h 语句
- 8 . 5 b r e a k 语句
- 8 . 6 c o n t i n u e 语句
- 8 . 7 习题

第 9 章 闭包

- 9 . 1 闭包
- 9 . 2 闭包、集合和字符串
- 9 . 3 闭包的其他特性
- 9 . 4 习题

第 10 章 文件

- 1 0 . 1 命令行参数
- 1 0 . 2 F i l e 类
- 1 0 . 3 习题

第 11 章 学习案例：图书馆应用程序（方法、闭包）

- 1 1 . 1 迭代 1：需求规范和映射实现
- 1 1 . 2 迭代 2：基于文本的用户交互界面的实现
- 1 1 . 3 迭代 3：使用闭包实现
- 1 1 . 4 习题

第 12 章 类

- 1 2 . 1 类
- 1 2 . 2 复合方法
- 1 2 . 3 习题

第 13 章 学习案例：图书馆应用程序（对象）

- 1 3 . 1 需求规范
- 1 3 . 2 迭代 1：最初的模型
- 1 3 . 3 迭代 2：模型完善
- 1 3 . 4 迭代 3：用户界面
- 1 3 . 5 习题

第 14 章 继承

- 1 4 . 1 继承

- 1 4 . 2 继承方法
- 1 4 . 3 方法重定义
- 1 4 . 4 多态性
- 1 4 . 5 抽象类
- 1 4 . 6 接口类
- 1 4 . 7 习题

第15章 单元测试 (J U N I T)

- 1 5 . 1 单元测试
- 1 5 . 2 GroovyTestCase类和JUnit TestCase类
- 1 5 . 3 GroovyTestSuite类和JUnit TestSuite类
- 1 5 . 4 单元测试的角色
- 1 5 . 5 习题

第16章 学习案例：图书馆应用程序（继承）

- 1 6 . 1 需求规范
- 1 6 . 2 迭代1：多态性
- 1 6 . 3 迭代2：功能性需求演示
- 1 6 . 4 迭代3：提供用户反馈
- 1 6 . 5 迭代4：强制性约束
- 1 6 . 6 习题

第17章 持久性

- 1 7 . 1 简单查询
- 1 7 . 2 关系
- 1 7 . 3 更新数据库
- 1 7 . 4 表的对象
- 1 7 . 5 继承
- 1 7 . 6 Spring框架
- 1 7 . 7 习题

第18章 学习案例：图书馆应用程序（持久性）

- 1 8 . 1 迭代1：域模型的持久化
- 1 8 . 2 迭代2：持久性的影响
- 1 8 . 3 习题

第19章 XML构造器和解析器

- 1 9 . 1 Groovy标记
- 1 9 . 2 MarkupBuilder
- 1 9 . 3 XML解析
- 1 9 . 4 习题

第20章 GUI构造器

- 2 0 . 1 SwingBuilder
- 2 0 . 2 列表框和表格
- 2 0 . 3 Box类和BoxLayout类

20.4 习题

第21章 模板引擎

21.1 字符串

21.2 模板

21.3 习题

第22章 学习案例：图书馆应用程序（G U I）

22.1 迭代1：G U I原型

22.2 迭代2：处理器的实现

22.3 习题

第23章 服务器端编程

23.1 Servlets

23.2 Groovlets

23.3 G S P页面

23.4 习题

第24章 学习案例：图书馆应用程序（W E B）

24.1 迭代1：Web实现

24.2 习题

第25章 后记

附录A 软件发布

A.1 Java开发工具

A.2 Groovy开发工具

A.3 A N T

A.4 Derby / Cloudscape数据库

A.5 Spring框架

A.6 Tomcat服务器

A.7 Eclipse IDE

A.8 本书源文件

附录B Groovy简介

B.1 简洁和优雅

B.2 方法

B.3 列表

B.4 类

B.5 多态性

B.6 闭包

B.7 异常

附录C 关于数值和表达式的更多信息

C.1 类

C.2 表达式

C.3 运算符结合性

C.4 定义变量

C.5 复合赋值运算符

C.6 逻辑运算符

- C . 7 条件运算符
- C . 8 数字字面值的分类
- C . 9 转换
- C . 10 静态类型
- C . 11 测试

附录D 关于字符串和正则表达式的更多信息

- D . 1 正则表达式
- D . 2 单字符匹配
- D . 3 匹配开始部分
- D . 4 匹配结尾部分
- D . 5 匹配零次或者多次
- D . 6 匹配一次或者多次
- D . 7 匹配零次或者一次
- D . 8 次数匹配
- D . 9 字符类型
- D . 10 选择
- D . 11 辅助符号
- D . 12 组合

附录E 关于列表、映射和范围的更多信息

- E . 1 类
- E . 2 列表
- E . 3 范围
- E . 4 展开操作符
- E . 5 测试

附录F 关于基本输入输出的更多信息

- F . 1 格式化输出
- F . 2 类Console

附录G 关于方法的更多信息

- G . 1 递归方法
- G . 2 静态类型
- G . 3 实参协议
- G . 4 方法重载
- G . 5 默认参数值的不确定性
- G . 6 参数和返回值类型为集合的方法

附录H 关于闭包的更多信息

- H . 1 闭包和不明确性
- H . 2 闭包和方法
- H . 3 默认参数
- H . 4 闭包和作用域
- H . 5 递归闭包
- H . 6 状态类型
- H . 7 有关实参的约定

H . 8 闭包、集合和范围
H . 9 R e t u r n语句
H . 1 0 测试

附录 I 关于类的更多信息

I . 1 属性和可见性
I . 2 对象导航
I . 3 静态成员
I . 4 操作符重载
I . 5 调用方法
I . 6 习题

附录 J 高级闭包

J . 1 简单闭包
J . 2 部分应用
J . 3 组合
J . 4 计算模式
J . 5 业务规则
J . 6 打包
J . 7 列表简化
J . 8 习题

附录 K 关于构造器的更多信息

K . 1 A n t B u i l d e r
K . 2 专用的构造器

附录 L 关于G U I 构造器的更多信息

L . 1 菜单和工具条
L . 2 对话框