第 17 章 项目

本章包含了一系列项目，它们都以本书介绍的内容为基础，并对早期的章节进行了一定程度的扩充。
与以前经历过的项目相比，这儿的大多数项目都明显要复杂得多，它们充分演示了新技术以及类库的运用。

17.1 文字处理

如果您有 C 或 C++的经验，那么最开始可能会对 Java 控制文本的能力感到怀疑。事实上，我们最害怕的就是速度特别慢，这可能妨碍我们创造能力的发挥。然而，Java 对应的工具（特别是 String 类）具有很强的功能，就象本节的例子展示的那样（而且性能也有一定程度的提升）。
正如大家即将看到的那样，建立这些例子的目的都是为了解决本书编制过程中遇到的一些问题。但是，它们的能力并非仅止于此。通过简单的改造，即可让它们在其他场合大显身手。除此以外，它们还揭示出了本书以前没有强调过的一项 Java 特性。

17.1.1 提取代码列表

对于本书每一个完整的代码列表（不是代码段），大家无疑会注意到它们都用特殊的注释记号起始与结束（'//:'和'///:~'）。之所以要包括这种标志信息，是为了能将代码从本书自动提取到兼容的源码文件中。在我的前一本书里，我设计了一个系统，可将测试过的代码文件自动合并到书中。但对于这本书，我发现一种更简便的做法是一旦通过了最初的测试，就把代码粘贴到书中。而且由于很难第一次就编译通过，所以我在书的内部编辑代码。但如何提取并测试代码呢？这个程序就是关键。如果你打算解决一个文字处理的问题，那么它也很有利用

价值。该例也演示了 String 类的许多特性。
我首先将整本书都以 ASCII 文本格式保存成一个独立的文件。CodePackager 程序有两种运行模式（在 usageString 有相应的描述）：如果使用-p 标志，程序就会检查一个包含了 ASCII 文本（即本书的内容）的一个输入文件。它会遍历这个文件，按照注释记号提取出代码，并用位于第一行的文件名来决定创建文件使用什么名字。除此以外，在需要将文件置入一个特殊目录的时候，它还会检查 package 语句（根据由 package 语句指定的路径选择）。
但这样还不够。程序还要对包（package）名进行跟踪，从而监视章内发生的变化。由于每一章使用的所有包都以 c02，c03，c04 等等起头，用于标记它们所属的是哪一章（除那些以 com 起头的以外，它们在对不同的章进行跟踪的时候会被忽略）——只要每一章的第一个代码列表包含了一个 package，所以 CodePackager 程序能知道每一章发生的变化，并将后续的文件放到新的子目录里。
每个文件提取出来时，都会置入一个 SourceCodeFile 对象，随后再将那个对象置入一个集合（后面还会详尽讲述这个过程）。这些 SourceCodeFile 对象可以简单地保存在文件中，那正是本项目的第二个用途。如果直接调用 CodePackager，不添加-p 标志，它就会将一个“打包”文件作为输入。那个文件随后会被提取（释放）进入单独的文件。所以-p 标志的意思就是提取出来的文件已被“打包”（packed）进入这个单一的文件。
但为什么还要如此麻烦地使用打包文件呢？这是由于不同的计算机平台用不同的方式在文件里保存文本信息。其中最大的问题是换行字符的表示方法；当然，还有可能存在另一些问题。然而，Java 有一种特殊类型的 IO 数据流——DataOutputStream——它可以保证“无论数据来自何种机器，只要使用一个 DataInputStream 收取这些数据，就可用本机正确的格式保存它们”。也就是说，Java 负责控制与不同平台有关的所有细节，而这正是 Java 最具魅力的一点。所以-p 标志能将所有东西都保存到单一的文件里，并采用通用的格式。用户可从 Web 下载这个文件以及 Java 程序，然后对这个文件运行 CodePackager，同时不指定-p 标志，文件便会释放到系统中正确的场所（亦可指定另一个子目录；否则就在当前目录创建子目录）。为确保不会留下与特定平台有关的格式，凡是需要描述一个文件或路径的时候，我们就使用 File 对象。除此以外，还有一项特别的安全措施：在每个子目录里都放入一个空文件；那个文件的名字指出在那个子目录里应找到多少个文件。
下面是完整的代码，后面会对它进行详细的说明：

959-968 页程序

我们注意到 package 语句已经作为注释标志出来了。由于这是本章的第一个程序，所以 package 语句是必需的，用它告诉 CodePackager 已改换到另一章。但是把它放入包里却会成为一个问题。当我们创建一个包的时候，需要将结果程序同一个特定的目录结构联系在一起，这一做法对本书的大多数例子都是适用的。但在这里，CodePackager 程序必须在一个专用的目录里编译和运行，所以 package 语句作为注释标记出去。但对 CodePackager 来说，它“看起来”依然象一个普通的 package 语句，因为程序还不是特别复杂，不能侦查到多行注释（没有必要做得这么复杂，这里只要求方便就行）。
头两个类是“支持／工具”类，作用是使程序剩余的部分在编写时更加连贯，也更便于阅读。第一个是 Pr，它类似 ANSI C 的 perror 库，两者都能打印出一条错误提示消息（但同时也会退出程序）。第二个类将文件的创建过程封装在内，这个过程已在第 10 章介绍过了；大家已经知道，这样做很快就会变得非常累赘和麻烦。为解决这个问题，第 10 章提供的方案致力于新类的创建，但这儿的“静态”方法已经使用过了。在那些方法中，正常的违例会被捕获，并相应地进行处理。这些方法使剩余的代码显得更加清爽，更易阅读。

帮助解决问题的第一个类是 SourceCodeFile（源码文件），它代表本书一个源码文件包含的所有信息（内容、文件名以及目录）。它同时还包含了一系列 String 常数，分别代表一个文件的开始与结束；在打包文件内使用的一个标记；当前系统的换行符；文件路径分隔符（注意要用 System.getProperty()侦查本地版本是什么）；以及一大段版权声明，它是从下面这个 Copyright.txt 文件里提取出来的：

969-967 页程序

从一个打包文件中提取文件时，当初所用系统的文件分隔符也会标注出来，以便用本地系统适用的符号替换它。
当前章的子目录保存在 chapter 字段中，它初始化成 c02（大家可注意一下第 2 章的列表正好没有包含一个打包语句）。只有在当前文件里发现一个 package（打包）语句时，chapter 字段才会发生改变。

1. 构建一个打包文件
第一个构建器用于从本书的 ASCII 文本版里提取出一个文件。发出调用的代码（在列表里较深的地方）会读入并检查每一行，直到找到与一个列表的开头相符的为止。在这个时候，它就会新建一个 SourceCodeFile 对象，将第一行的内容（已经由调用代码读入了）传递给它，同时还要传递 BufferedReader 对象，以便在这个缓冲区中提取源码列表剩余的内容。
从这时起，大家会发现 String 方法被频繁运用。为提取出文件名，需调用 substring()的过载版本，令其从一个起始偏移开始，一直读到字串的末尾，从而形成一个“子串”。为算出这个起始索引，先要用 length()得出 startMarker 的总长，再用 trim()删除字串头尾多余的空格。第一行在文件名后也可能有一些字符；它们是用 indexOf()侦测出来的。若没有发现找到我们想寻找的字符，就返回-1；若找到那些字符，就返回它们第一次出现的位置。注意这也是 indexOf()的一个过载版本，采用一个字串作为参数，而非一个字符。
解析出并保存好文件名后，第一行会被置入字串 contents 中（该字串用于保存源码清单的完整正文）。随后，将剩余的代码行读入，并合并进入 contents 字串。当然事情并没有想象的那么简单，因为特定的情况需加以特别的控制。一种情况是错误检查：若直接遇到一个 startMarker（起始标记），表明当前操作的这个代码列表没有设置一个结束标记。这属于一个出错条件，需要退出程序。
另一种特殊情况与 package 关键字有关。尽管 Java 是一种自由形式的语言，但这个程序要求 package 关键字必须位于行首。若发现 package 关键字，就通过检查位于开头的空格以及位于末尾的分号，从而提取出包名（注意亦可一次单独的操作实现，方法是使用过载的 substring()，令其同时检查起始和结束索引位置）。随后，将包名中的点号替换成特定的文件分隔符——当然，这里要假设文件分隔符仅有一个字符的长度。尽管这个假设可能对目前的所有系统都是适用的，但一旦遇到问题，一定不要忘了检查一下这里。
默认操作是将每一行都连接到 contents 里，同时还有换行字符，直到遇到一个 endMarker（结束标记）为止。该标记指出构建器应当停止了。若在 endMarker 之前遇到了文件结尾，就认为存在一个错误。

2. 从打包文件中提取
第二个构建器用于将源码文件从打包文件中恢复（提取）出来。在这儿，作为调用者的方法不必担心会跳过一些中间文本。打包文件包含了所有源码文件，它们相互间紧密地靠在一起。需要传递给该构建器的仅仅是一个 BufferedReader，它代表着“信息源”。构建器会从中提

取出自己需要的信息。但在每个代码列表开始的地方还有一些配置信息，它们的身份是用 packMarker（打包标记）指出的。若 packMarker 不存在，意味着调用者试图用错误的方法来使用这个构建器。
一旦发现 packMarker，就会将其剥离出来，并提取出目录名（用一个'#'结尾）以及文件名（直到行末）。不管在哪种情况下，旧分隔符都会被替换成本地适用的一个分隔符，这是用 String replace()方法实现的。老的分隔符被置于打包文件的开头，在代码列表稍靠后的一部分即可看到是如何把它提取出来的。
构建器剩下的部分就非常简单了。它读入每一行，把它合并到 contents 里，直到遇见 endMarker 为止。

3. 程序列表的存取
接下来的一系列方法是简单的访问器：directory()、filename()（注意方法可能与字段有相同的拼写和大小写形式）和 contents()。而 hasFile()用于指出这个对象是否包含了一个文件（很快就会知道为什么需要这个）。
最后三个方法致力于将这个代码列表写进一个文件——要么通过 writePacked()写入一个打包文件，要么通过 writeFile()写入一个 Java 源码文件。writePacked()需要的唯一东西就是 DataOutputStream，它是在别的地方打开的，代表着准备写入的文件。它先把头信息置入第一行，再调用 writeBytes()将 contents（内容）写成一种“通用”格式。
准备写 Java 源码文件时，必须先把文件建好。这是用 IO.psOpen()实现的。我们需要向它传递一个 File 对象，其中不仅包含了文件名，也包含了路径信息。但现在的问题是：这个路径实际存在吗？用户可能决定将所有源码目录都置入一个完全不同的子目录，那个目录可能是尚不存在的。所以在正式写每个文件之前，都要调用 File.mkdirs()方法，建好我们想向其中写入文件的目录路径。它可一次性建好整个路径。

4. 整套列表的包容
以子目录的形式组织代码列表是非常方便的，尽管这要求先在内存中建好整套列表。之所以要这样做，还有另一个很有说服力的原因：为了构建更“健康”的系统。也就是说，在创建代码列表的每个子目录时，都会加入一个额外的文件，它的名字包含了那个目录内应有的文件数目。
DirMap 类可帮助我们实现这一效果，并有效地演示了一个“多重映射”的概述。这是通过一个散列表（Hashtable）实现的，它的“键”是准备创建的子目录，而“值”是包含了那个特定目录中的 SourceCodeFile 对象的 Vector 对象。所以，我们在这儿并不是将一个“键”映射（或对应）到一个值，而是通过对应的 Vector，将一个键“多重映射”到一系列值。尽管这听起来似乎很复杂，但具体实现时却是非常简单和直接的。大家可以看到，DirMap 类的大多数代码都与向文件中的写入有关，而非与“多重映射”有关。与它有关的代码仅极少数而已。
可通过两种方式建立一个 DirMap（目录映射或对应）关系：默认构建器假定我们希望目录从当前位置向下展开，而另一个构建器让我们为起始目录指定一个备用的“绝对”路径。
add()方法是一个采取的行动比较密集的场所。首先将 directory()从我们想添加的 SourceCodeFile 里提取出来，然后检查散列表（Hashtable），看看其中是否已经包含了那个键。如果没有，就向散列表加入一个新的 Vector，并将它同那个键关联到一起。到这时，不管采取的是什么途径，Vector 都已经就位了，可以将它提取出来，以便添加 SourceCodeFile。由于 Vector 可象这样同散列表方便地合并到一起，所以我们从两方面都能感觉得非常方便。
写一个打包文件时，需打开一个准备写入的文件（当作 DataOutputStream 打开，使数据具有

“通用”性），并在第一行写入与老的分隔符有关的头信息。接着产生对 Hashtable 键的一个 Enumeration（枚举），并遍历其中，选择每一个目录，并取得与那个目录有关的 Vector，使那个 Vector 中的每个 SourceCodeFile 都能写入打包文件中。
用 write()将 Java 源码文件写入它们对应的目录时，采用的方法几乎与 writePackedFile()完全一致，因为两个方法都只需简单调用 SourceCodeFile 中适当的方法。但在这里，根路径会传递给 SourceCodeFile.writeFile()。所有文件都写好后，名字中指定了已写文件数量的那个附加文件也会被写入。

5. 主程序
前面介绍的那些类都要在 CodePackager 中用到。大家首先看到的是用法字串。一旦最终用户不正确地调用了程序，就会打印出介绍正确用法的这个字串。调用这个字串的是 usage()方法，同时还要退出程序。main()唯一的任务就是判断我们希望创建一个打包文件，还是希望从一个打包文件中提取什么东西。随后，它负责保证使用的是正确的参数，并调用适当的方法。
创建一个打包文件时，它默认位于当前目录，所以我们用默认构建器创建 DirMap。打开文件后，其中的每一行都会读入，并检查是否符合特殊的条件：
(1) 若行首是一个用于源码列表的起始标记，就新建一个 SourceCodeFile 对象。构建器会读入源码列表剩下的所有内容。结果产生的句柄将直接加入 DirMap。
(2) 若行首是一个用于源码列表的结束标记，表明某个地方出现错误，因为结束标记应当只能由 SourceCodeFile 构建器发现。

提取／释放一个打包文件时，提取出来的内容可进入当前目录，亦可进入另一个备用目录。所以需要相应地创建 DirMap 对象。打开文件，并将第一行读入。老的文件路径分隔符信息将从这一行中提取出来。随后根据输入来创建第一个 SourceCodeFile 对象，它会加入 DirMap。只要包含了一个文件，新的 SourceCodeFile 对象就会创建并加入（创建的最后一个用光输入内容后，会简单地返回，然后 hasFile()会返回一个错误）。

17.1.2 检查大小写样式
尽管对涉及文字处理的一些项目来说，前例显得比较方便，但下面要介绍的项目却能立即发挥作用，因为它执行的是一个样式检查，以确保我们的大小写形式符合“事实上”的 Java 样式标准。它会在当前目录中打开每个.java 文件，并提取出所有类名以及标识符。若发现有不符合 Java 样式的情况，就向我们提出报告。
为了让这个程序正确运行，首先必须构建一个类名，将它作为一个“仓库”，负责容纳标准 Java 库中的所有类名。为达到这个目的，需遍历用于标准 Java 库的所有源码子目录，并在每个子目录都运行 ClassScanner。至于参数，则提供仓库文件的名字（每次都用相同的路径和名字）和命令行开关-a，指出类名应当添加到该仓库文件中。
为了用程序检查自己的代码，需要运行它，并向它传递要使用的仓库文件的路径与名字。它会检查当前目录中的所有类和标识符，并告诉我们哪些没有遵守典型的 Java 大写写规范。
要注意这个程序并不是十全十美的。有些时候，它可能报告自己查到一个问题。但当我们仔细检查代码的时候，却发现没有什么需要更改的。尽管这有点儿烦人，但仍比自己动手检查代码中的所有错误强得多。
下面列出源代码，后面有详细的解释：

974-980 页程序

MultiStringMap 类是个特殊的工具，允许我们将一组字串与每个键项对应（映射）起来。和前例一样，这里也使用了一个散列表（Hashtable），不过这次设置了继承。该散列表将键作为映射成为 Vector 值的单一的字串对待。add()方法的作用很简单，负责检查散列表里是否存在一个键。如果不存在，就在其中放置一个。getVector()方法为一个特定的键产生一个 Vector；而 printValues()将所有值逐个 Vector 地打印出来，这对程序的调试非常有用。

为简化程序，来自标准 Java 库的类名全都置入一个 Properties（属性）对象中（来自标准 Java 库）。记住 Properties 对象实际是个散列表，其中只容纳了用于键和值项的 String 对象。然而仅需一次方法调用，我们即可把它保存到磁盘，或者从磁盘中恢复。实际上，我们只需要一个名字列表，所以为键和值都使用了相同的对象。

针对特定目录中的文件，为找出相应的类与标识符，我们使用了两个 MultiStringMap：classMap 以及 identMap。此外在程序启动的时候，它会将标准类名仓库装载到名为 classes 的 Properties 对象中。一旦在本地目录发现了一个新类名，也会将其加入 classes 以及 classMap。这样一来，classMap 就可用于在本地目录的所有类间遍历，而且可用 classes 检查当前标记是不是一个类名（它标记着对象或方法定义的开始，所以收集接下去的记号——直到碰到一个分号——并将它们都置入 identMap）。

ClassScanner 的默认构建器会创建一个由文件名构成的列表（采用 FilenameFilter 的 JavaFilter 实现形式，参见第 10 章）。随后会为每个文件名都调用 scanListing()。

在 scanListing()内部，会打开源码文件，并将其转换成一个 StreamTokenizer。根据 Java 帮助文档，将 true 传递给 slashStartComments()和 slashSlashComments()的本意应当是剥除那些注释内容，但这样做似乎有些问题（在 Java 1.0 中几乎无效）。所以相反，那些行被当作注释标记出去，并用另一个方法来提取注释。为达到这个目的，'/'必须作为一个原始字符捕获，而不是让 StreamTokeinzer 将其当作注释的一部分对待。此时要用 ordinaryChar()方法指示 StreamTokenizer 采取正确的操作。同样的道理也适用于点号（'.'），因为我们希望让方法调用分离出单独的标识符。但对下划线来说，它最初是被 StreamTokenizer 当作一个单独的字符对待的，但此时应把它留作标识符的一部分，因为它在 static final 值中以 TT_EOF 等等形式使用。当然，这一点只对目前这个特殊的程序成立。wordChars()方法需要取得我们想添加的一系列字符，把它们留在作为一个单词看待的记号中。最后，在解析单行注释或者放弃一行的时候，我们需要知道一个换行动作什么时候发生。所以通过调用 eollsSignificant(true)，换行符（EOL）会被显示出来，而不是被 StreamTokenizer 吸收。

scanListing()剩余的部分将读入和检查记号，直至文件尾。一旦 nextToken()返回一个 final static 值——StreamTokenizer.TT_EOF，就标志着已经抵达文件尾部。

若记号是个'/'，意味着它可能是个注释，所以就调用 eatComments()，对这种情况进行处理。我们在这儿唯一感兴趣的其他情况是它是否为一个单词，当然还可能存在另一些特殊情况。如果单词是 class（类）或 interface（接口），那么接着的记号就应当代表一个类或接口名字，并将其置入 classes 和 classMap。若单词是 import 或者 package，那么我们对这一行剩下的东西就没什么兴趣了。其他所有东西肯定是一个标识符（这是我们感兴趣的），或者是一个关键字（对此不感兴趣，但它们采用的肯定是小写形式，所以不必兴师动众地检查它们）。它们将加入到 identMap。

discardLine()方法是一个简单的工具，用于查找行末位置。注意每次得到一个新记号时，都必须检查行末。

只要在主解析循环中碰到一个正斜杠，就会调用 eatComments()方法。然而，这并不表示肯定遇到了一条注释，所以必须将接着的记号提取出来，检查它是一个正斜杠（那么这一行会被丢弃），还是一个星号。但假如两者都不是，意味着必须在主解析循环中将刚才取出的记

号送回去！幸运的是，pushBack()方法允许我们将当前记号“压回”输入数据流。所以在主解析循环调用 nextToken()的时候，它能正确地得到刚才送回的东西。
为方便起见，classNames()方法产生了一个数组，其中包含了 classes 集合中的所有名字。这个方法未在程序中使用，但对代码的调试非常有用。
接下来的两个方法是实际进行检查的地方。在 checkClassNames()中，类名从 classMap 提取出来（请记住，classMap 只包含了这个目录内的名字，它们按文件名组织，所以文件名可能伴随错误的类名打印出来）。为做到这一点，需要取出每个关联的 Vector，并遍历其中，检查第一个字符是否为小写。若确实为小写，则打印出相应的出错提示消息。
在 checkIdentNames()中，我们采用了一种类似的方法：每个标识符名字都从 identMap 中提取出来。如果名字不在 classes 列表中，就认为它是一个标识符或者关键字。此时会检查一种特殊情况：如果标识符的长度等于 3 或者更长，而且所有字符都是大写的，则忽略此标识符，因为它可能是一个 static final 值，比如 TT_EOF。当然，这并不是一种完美的算法，但它假定我们最终会注意到任何全大写标识符都是不合适的。
这个方法并不是报告每一个以大写字符开头的标识符，而是跟踪那些已在一个名为 reportSet()的 Vector 中报告过的。它将 Vector 当作一个“集合”对待，告诉我们一个项目是否已在那个集合中。该项目是通过将文件名和标识符连接起来生成的。若元素不在集合中，就加入它，然后产生报告。
程序列表剩下的部分由 main()构成，它负责控制命令行参数，并判断我们是准备在标准 Java 库的基础上构建由一系列类名构成的“仓库”，还是想检查已写好的那些代码的正确性。不管在哪种情况下，都会创建一个 ClassScanner 对象。
无论准备构建一个“仓库”，还是准备使用一个现成的，都必须尝试打开现有仓库。通过创建一个 File 对象并测试是否存在，就可决定是否打开文件并在 ClassScanner 中装载 classes 这个 Properties 列表（使用 load()）。来自仓库的类将追加到由 ClassScanner 构建器发现的类后面，而不是将其覆盖。如果仅提供一个命令行参数，就意味着自己想对类名和标识符名字进行一次检查。但假如提供两个参数（第二个是"-a"），就表明自己想构成一个类名仓库。在这种情况下，需要打开一个输出文件，并用 Properties.save()方法将列表写入一个文件，同时用一个字串提供文件头信息。

17.2 方法查找工具

第 11 章介绍了 Java 1.1 新的“反射”概念，并利用这个概念查询一个特定类的方法——要么是由所有方法构成的一个完整列表，要么是这个列表的一个子集（名字与我们指定的关键字相符）。那个例子最大的好处就是能自动显示出所有方法，不强迫我们在继承结构中遍历，检查每一级的基础类。所以，它实际是我们节省编程时间的一个有效工具：因为大多数 Java 方法的名字都规定得非常全面和详尽，所以能有效地找出那些包含了一个特殊关键字的方法名。若找到符合标准的一个名字，便可根据它直接查阅联机帮助文档。
但第 11 的那个例子也有缺陷，它没有使用 AWT，仅是一个纯命令行的应用。在这儿，我们准备制作一个改进的 GUI 版本，能在我们键入字符的时候自动刷新输出，也允许我们在输出结果中进行剪切和粘贴操作：

983-987 页程序

程序中的有些东西已在以前见识过了。和本书的许多 GUI 程序一样，这既可作为一个独立的应用程序使用，亦可作为一个程序片（Applet）使用。此外，StripQualifiers 类与它在第 11 章的表现是完全一样的。

GUI 包含了一个名为 name 的“文本字段”（TextField），或在其中输入想查找的类名；还包含了另一个文本字段，名为 searchFor，可选择性地在其中输入一定的文字，希望在方法列表中查找那些文字。Checkbox（复选框）允许我们指出最终希望在输出中使用完整的名字，还是将前面的各种限定信息删去。最后，结果显示于一个“文本区域”（TextArea）中。

大家会注意到这个程序未使用任何按钮或其他组件，不能用它们开始一次搜索。这是由于无论文本字段还是复选框都会受到它们的“侦听者（Listener）对象的监视。只要作出一项改变，结果列表便会立即更新。若改变了 name 字段中的文字，新的文字就会在 NameL 类中捕获。若文字不为空，则在 Class.forName()中用于尝试查找类。当然，在文字键入期间，名字可能会变得不完整，而 Class.forName()会失败，这意味着它会“掷”出一个违例。该违例会被捕获，TextArea 会随之设为“Nomatch”（没有相符）。但只要键入了一个正确的名字（大小写也算在内），Class.forName()就会成功，而 getMethods()和 getConstructors()会分别返回由 Method 和 Constructor 对象构成的一个数组。这些数组中的每个对象都会通过 toString()转变成一个字串（这样便产生了完整的方法或构建器签名），而且两个列表都会合并到 n 中——一个独立的字串数组。数组 n 属于 DisplayMethods 类的一名成员，并在调用 reDisplay()时用于显示的更新。

若改变了 Checkbox 或 searchFor 组件，它们的“侦听者”会简单地调用 reDisplay()。reDisplay()会创建一个临时数组，其中包含了名为 rs 的字串（rs 代表“结果集”——Result Set）。结果集要么直接从 n 复制（没有 find 关键字），要么选择性地从包含了 find 关键字的 n 中的字串复制。最后会检查 strip Checkbox，看看用户是不是希望将名字中多余的部分删除（默认为“是”）。若答案是肯定的，则用 StripQualifiers.strip()做这件事情；反之，就将列表简单地显示出来。

在 init()中，大家也许认为在设置布局时需要进行大量繁重的工作。事实上，组件的布置完全可能只需要极少的工作。但象这样使用 BorderLayout 的好处是它允许用户改变窗口的大小，并特别能使 TextArea（文本区域）更大一些，这意味着我们可以改变大小，以便毋需滚动即可看到更长的名字。

编程时，大家会发现特别有必要让这个工具处于运行状态，因为在试图判断要调用什么方法的时候，它提供了最好的方法之一。

17.3 复杂性理论

下面要介绍的程序的前身是由 Larry O'Brien 原创的一些代码，并以由 Craig Reynolds 于 1986 年编制的“Boids”程序为基础，当时是为了演示复杂性理论的一个特殊问题，名为“凸显”（Emergence）。

这儿要达到的目标是通过为每种动物都规定少许简单的规则，从而逼真地再现动物的群聚行为。每个动物都能看到看到整个环境以及环境中的其他动物，但它只与一系列附近的“群聚伙伴”打交道。动物的移动基于三个简单的引导行为：

(1) 分隔：避免本地群聚伙伴过于拥挤。

(2) 方向：遵从本地群聚伙伴的普遍方向。

(3) 聚合：朝本地群聚伙伴组的中心移动。

更复杂的模型甚至可以包括障碍物的因素，动物能预知和避免与障碍冲突的能力，所以它们能围绕环境中的固定物体自由活动。除此以外，动物也可能有自己的特殊目标，这也许会造成群体按特定的路径前进。为简化讨论，避免障碍以及目标搜寻的因素并未包括到这里建立的模型中。

尽管计算机本身比较简陋，而且采用的规则也相当简单，但结果看起来是真实的。也就是说，相当逼真的行为从这个简单的模型中“凸显”出来了。

程序以合成到一起的应用程序／程序片的形式提供：

989-995 页程序

尽管这并非对 Craig Reynold 的“Boids”例子中的行为完美重现，但它却展现出了自己独有的迷人之外。通过对数字进行调整，即可进行全面的修改。至于与这种群聚行为有关的更多的情况，大家可以访问 Craig Reynold 的主页——在那个地方，甚至还提供了 Boids 一个公开的 3D 展示版本：
http://www.hmt.com/cwr/boids.html
为了将这个程序作为一个程序片运行，请在 HTML 文件中设置下述程序片标志：

995 页中程序

17.4 总结
通过本章的学习，大家知道运用 Java 可做到一些较复杂的事情。通过这些例子亦可看出，尽管 Java 必定有自己的局限，但受那些局限影响的主要是性能（比如写好文字处理程序后，会发现 C++的版本要快得多——这部分是由于 IO 库做得不完善造成的；而在你读到本书的时候，情况也许已发生了变化。但 Java 的局限也仅此而已，它在语言表达方面的能力是无以伦比的。利用 Java，几乎可以表达出我们想得到的任何事情。而与此同时，Java 在表达的方便性和易读性上，也做足了功夫。所以在使用 Java 时，一般不会陷入其他语言常见的那种复杂境地。使用那些语言时，会感觉它们象一个爱唠叨的老太婆，哪有 Java 那样清纯、简练！而且通过 Java 1.2 的 JFC/Swing 库，AWT 的表达能力和易用性甚至又得到了进一步的增强。

17.5 练习
(1) （稍微有些难度）改写 FieldOBeasts.java，使它的状态能够保持固定。加上一些按钮，允许用户保存和恢复不同的状态文件，并从它们断掉的地方开始继续运行。请先参考第 10 章的 CADState.java，再决定具体怎样做。
(2) （大作业）以 FieldOBeasts.java 作为起点，构造一个自动化交通仿真系统。
(3) （大作业）以 ClassScanner.java 作为起点，构造一个特殊的工具，用它找出那些虽然定义但从未用过的方法和字段。
(4) （大作业）利用 JDBC，构造一个联络管理程序。让这个程序以一个平面文件数据库为基础，其中包含了名字、地址、电话号码、E-mail 地址等联系资料。应该能向数据库里方便地加入新名字。键入要查找的名字时，请采用在第 15 章的 VLookup.java 里介绍过的那种名字自动填充技术。