

有相应的工具，通过连接列名和在表单中填入信息，让用户能够以某种简单的方式来表示其各种查询。这种工具通常称为实例查询（Query by Example, QBE）工具。而使用 SQL 的查询则是利用 SQL 语法以文本方式编写的。例如，

```
SELECT Books, Books.Publisher_Id, Books.Price, Publishers.Name, Publishers.URL
FROM Books, Publishers
WHERE Books.Publisher_Id = Publishers.Publisher_Id
```

在本节的余下部分中，我们将介绍如何编写这样的查询语句。如果你已经熟悉 SQL 了，就可以跳过这部分内容。

按照惯例，SQL 关键字全部使用大写字母。当然，也可以不这样做。

SELECT 语句相当灵活。仅使用下面这个查询语句，就可以查出 Books 表中的所有记录：

```
SELECT * FROM Books
```

在每一个 SQL 的 SELECT 语句中，FROM 子句都是必不可少的。FROM 子句用于告知数据库应该在哪个表上查询数据。

我们还可以选择所需要的列：

```
SELECT ISBN, Price, Title
FROM Books
```

并且还可以在查询语句中使用 WHERE 子句来限定所要选择的行：

```
SELECT ISBN, Price, Title
FROM Books
WHERE Price <= 29.95
```

请小心使用“相等”这个比较操作。与 Java 编程语言不同，SQL 使用 = 和 <> 而非 == 和 != 来进行相等性比较。

注释：有些数据库供应商的产品支持在进行不等于比较时使用 !=。这不符合标准 SQL 的语法，所以我们建议不要使用这种方法。

WHERE 子句也可以使用 LIKE 操作符来实现模式匹配。不过，这里的通配符并不是通常使用的 * 和 ?，而是用 % 表示 0 或多个字符，用下划线表示单个字符。例如，

```
SELECT ISBN, Price, Title
FROM Books
WHERE Title NOT LIKE '%n_x%'
```

这条语句排除了所有书名中包含 UNIX 或者 Linux 的图书。

请注意，字符串都是用单引号括起来的，而非双引号。字符串中的单引号则需要用一对单引号代替。例如，

```
SELECT Title
FROM Books
WHERE Title LIKE '%''%'
```

上述语句会返回所有包含单引号的书名。

你也可以从多个表中选取数据：

```
SELECT * FROM Books, Publishers
```

如果没有 WHERE 子句，上述查询语句就意义不大了，它只是罗列了两个表中所有记录的组合。在我们这个例子中，Books 表有 20 行记录，Publishers 表有 8 行记录，合并的结果将产生 20×8 条记录，其中不乏大量重复数据。实际上我们需要对查询结果进行限制，只对那些图书与出版社相匹配的数据感兴趣。

```
SELECT * FROM Books, Publishers
WHERE Books.Publisher_Id = Publishers.Publisher_Id
```

这条语句的查询结果共有 20 行记录，每一条记录对应于一本书，因为每本书都在 Publishers 表中只对应一个出版社。

每当查询语句涉及多个表时，相同的列名可能会出现在两个不同的地方。在我们的例子中也存在这种情况，Books 表和 Publishers 表都拥有一个列名为 PublisherId 的列。当出现歧义时，可以在每个列名前添加它所在表的表名作为前缀，比如 Books/Publisher_Id。

也可以使用 SQL 来改变数据库中的数据。例如，假设现在要将所有书名中包含“C++”的图书降价 5 美元，可以执行以下语句：

```
UPDATE Books
SET Price = Price - 5.00
WHERE Title LIKE '%C++%'
```

类似地，要删除所有的 C++ 图书，可以使用下面的 DELETE 查询：

```
DELETE FROM Books
WHERE Title LIKE '%C++%'
```

此外，SQL 中还有许多内置函数，用于对某一列计算平均值、查找最大值和最小值以及其他许多功能。在此我们就不讨论了。

典型情况下，可以使用 INSERT 语句向表中插入值：

```
INSERT INTO Books
VALUES ('A Guide to the SQL Standard', '0-201-96426-0', '0201', 47.95)
```

我们必须为每一条插入到表中的记录使用一次 INSERT 语句。

当然，在查询、修改和插入数据之前，必须要有存储数据的位置。可以使用 CREATE TABLE 语句创建一个新表，还可以为每一列指定列名和数据类型。

```
CREATE TABLE Books
(
    Title CHAR(60),
    ISBN CHAR(13),
    Publisher_Id CHAR(6),
    Price DECIMAL(10,2)
)
```

表 5-5 给出了最常见的 SQL 数据类型。

表 5-5 SQL 数据类型

数据类型	说明
INTEGER 或 INT	通常为 32 位的整数
SMALLINT	通常为 16 位的整数

(续)

数据类型	说明
NUMERIC(<i>m,n</i>)、DECIMAL(<i>m,n</i>) 或 DEC(<i>m,n</i>)	<i>m</i> 位长的定点十进制数，其中小数点后为 <i>n</i> 位
FLOAT(<i>n</i>)	运算精度为 <i>n</i> 位二进制数的浮点数
REAL	通常为 32 位浮点数
DOUBLE	通常为 64 位浮点数
CHARACTER(<i>n</i>) 或 CHAR(<i>n</i>)	固定长度为 <i>n</i> 的字符串
VARCHAR(<i>n</i>)	最大长度为 <i>n</i> 的可变长字符串
BOOLEAN	布尔值
DATE	日历日期（与具体的实现相关）
TIME	当前时间（与具体的实现相关）
TIMESTAMP	当前日期和时间（与具体的实现相关）
BLOB	二进制大对象
CLOB	字符大对象

在本书中，我们不再介绍更多的子句，比如可以应用于 CREATE TABLE 语句的主键子句和约束子句。

5.3 JDBC 配置

当然，你需要有一个可获得其 JDBC 驱动程序的数据库程序。目前这方面有许多出色的程序可供选择，比如 IBM DB2、Microsoft SQL Server、MySQL、Oracle 和 PostgreSQL。

为了练习本部分内容，你还需要创建一个数据库，我们假定你将这个数据库命名为 COREJAVA。你要自己创建，或者让数据库管理员创建这个数据库，并让你拥有适当权限，因为你需要拥有对这个数据库进行创建、更新和删除表的权限。

如果你以前从未安装过采用客户端 / 服务器模式的数据库，那么就会发现配置这样一个数据库会稍显复杂并且难于诊断故障的原因。如果安装的数据库无法正常运行，那么最好请专家来帮忙。

如果第一次接触数据库，我们建议使用 Apache Derby，它可以从 <http://db.apache.org/> derby 处下载到，在某些 JDK 版本中也包含了它。

在编写第一个数据库程序之前，你需要收集大量的信息和文件，下面将讨论这些内容。

5.3.1 数据库 URL

在连接数据库时，我们必须使用各种与数据库类型相关的参数，例如主机名、端口号和数据库名。

JDBC 使用了一种与普通 URL 相类似的语法来描述数据源。下面是这种语法的两个实例：

```
jdbc:derby://localhost:1527/COREJAVA;create=true
jdbc:postgresql:COREJAVA
```

上述 JDBC URL 指定了名为 COREJAVA 的一个 Derby 数据库和一个 PostgreSQL 数据库。JDBC URL 的一般语法为：

```
jdbc:subprotocol:other stuff
```

其中，*subprotocol* 用于选择连接到数据库的具体驱动程序。

other stuff 参数的格式随所使用的 *subprotocol* 不同而不同。如果要了解具体格式，你需要查阅数据库供应商提供的相关文档。

5.3.2 驱动程序 JAR 文件

你需要获得包含了你所使用的数据库的驱动程序的 JAR 文件。如果你使用的是 Derby，那么就需要 *derbyclient.jar*；如果你使用的是其他的数据库，那么就需要去寻找恰当的驱动程序。例如，PostgreSQL 的驱动程序可以在 <http://jdbc.postgresql.org> 处找到。

在运行访问数据库的程序时，需要将驱动程序的 JAR 文件包括到类路径中（编译时并不需要这个 JAR 文件）。

在从命令行启动程序时，只需要使用下面的命令：

```
java -classpath driverPath:. ProgramName
```

在 Windows 上，可以使用分号将当前路径（即由 . 字符表示的路径）与驱动程序 JAR 文件分隔开。

5.3.3 启动数据库

数据库服务器在连接之前需要先启动，启动的细节取决于所使用的数据库。

在使用 Derby 数据库时，需要遵循下面的步骤：

1. 打开命令 shell，并转到将来存放数据库文件的目录中。

2. 定位 *derbyrun.jar*。对于某些 JDK 版本，它包含在 *jdk/db/lib* 目录中，如果没有包含，那就安装 Apache Derby，并定位安装目录的 JAR 文件。我们用 *derby* 来表示包含 *lib/derbyrun.jar* 的目录。

3. 运行下面的命令：

```
java -jar derby/lib/derbyrun.jar server start
```

4. 仔细检查数据库是否正确工作了。然后创建一个名为 *ij.properties* 并包含下面各行的文件：

```
ij.driver=org.apache.derby.jdbc.ClientDriver
ij.protocol=jdbc:derby://localhost:1527/
ij.database=COREJAVA;create=true
```

在另一个命令 shell 中，通过执行下面的命令来运行 Derby 的交互式脚本执行工具（称为 *ij*）：

```
java -jar derby/lib/derbyrun.jar ij -p ij.properties
```

现在，可以发布像下面这样的 SQL 命令了：

```

CREATE TABLE Greetings (Message CHAR(20));
INSERT INTO Greetings VALUES ('Hello, World!');
SELECT * FROM Greetings;
DROP TABLE Greetings;

```

注意，每条命令都需要以分号结尾。要退出编辑器，可以键入

EXIT;

5. 在使用完数据库之后，可以用下面的命令关闭服务器：

```
java -jar derby/lib/derbyrun.jar server shutdown
```

如果使用其他的数据库，则需要查看文档，以了解如何启动和关闭数据库服务器，以及如何连接到数据库和发布 SQL 命令。

5.3.4 注册驱动器类

许多 JDBC 的 JAR 文件（例如 Derby 驱动程序）会自动注册驱动器类，在这种情况下，可以跳过本节所描述的手动注册步骤。包含 META-INF/services/java.sql.Driver 文件的 JAR 文件可以自动注册驱动器类，解压缩驱动程序 JAR 文件就可以检查其是否包含该文件。

如果驱动程序 JAR 文件不支持自动注册，那就需要找出数据库提供商使用的 JDBC 驱动器类的名字。典型的驱动器名字如下：

```

org.apache.derby.jdbc.ClientDriver
org.postgresql.Driver

```

通过使用 DriverManager，可以用两种方式来注册驱动器。一种方式是在 Java 程序中加载驱动器类，例如：

```
Class.forName("org.postgresql.Driver"); // force loading of driver class
```

这条语句将使得驱动器类被加载，由此将执行可以注册驱动器的静态初始化器。

另一种方式是设置 jdbc.drivers 属性。可以用命令行参数来指定这个属性，例如：

```
java -Djdbc.drivers=org.postgresql.Driver ProgramName
```

或者在应用中用下面这样的调用来设置系统属性

```
System.setProperty("jdbc.drivers", "org.postgresql.Driver");
```

在这种方式中可以提供多个驱动器，用冒号将它们分隔开，例如

```
org.postgresql.Driver:org.apache.derby.jdbc.ClientDriver
```

5.3.5 连接到数据库

在 Java 程序中，我们可以用下面这样的代码打开一个数据库连接：

```

String url = "jdbc:postgresql:COREJAVA";
String username = "dbuser";
String password = "secret";
Connection conn = DriverManager.getConnection(url, username, password);

```

驱动管理器会遍历所有注册过的驱动程序，以便找到一个能够使用数据库 URL 中指定的子协议的驱动程序。

`getConnection` 方法返回一个 `Connection` 对象。在下一节中，我们将详细介绍如何使用 `Connection` 对象来执行 SQL 语句。

要连接到数据库，我们还需要知道数据库的名字和密码。

注释：在默认情况下，Derby 允许我们使用任何用户名进行连接，并且不检查密码。它会为每个用户生成一个单独的表集合，而默认的用户名是 `app`。

程序清单 5-1 中的测试程序将所有这些步骤放到了一起：它从名为 `database.properties` 的文件中加载连接参数，并连接到数据库。示例代码中提供的 `database.properties` 文件包含的是关于 Derby 数据库的连接信息，如果使用其他的数据库，则需要将与数据库相关的连接信息放到这个文件中。下面是一个用于连接到 PostgreSQL 数据库的示例：

```
jdbc.drivers=org.postgresql.Driver
jdbc.url=jdbc:postgresql:COREJAVA
jdbc.username=dbuser
jdbc.password=secret
```

在连接到数据库之后，这个测试程序执行了下面的 SQL 语句：

```
CREATE TABLE Greetings (Message CHAR(20))
INSERT INTO Greetings VALUES ('Hello, World!')
SELECT * FROM Greetings
```

`SELECT` 语句的结果将被打印出来，你应该可以看到如下的输出：

```
Hello, World!
```

然后，通过执行下面的语句移除这张表：

```
DROP TABLE Greetings
```

要运行这个测试程序，需要按照前面所描述的方式启动数据库，并像下面这样启动这个程序：

```
java -classpath .:driverJAR test.TestDB
```

(Windows 用户需要注意，用 ; 代替 : 来分隔路径元素。)

提示：调试与 JDBC 相关的问题时，有种方法是启用 JDBC 的跟踪机制。调用 `DriverManager.setLogWriter` 方法可以将跟踪信息发送给 `PrintWriter`，而 `PrintWriter` 将输出 JDBC 活动的详细列表。大多数 JDBC 驱动程序的实现都提供了用于跟踪的附加机制，例如，在使用 Derby 时，可以在 JDBC 的 URL 中添加 `traceFile` 选项，如 `jdbc:derby://localhost:1527/ COREJAVA;create=true;traceFile=trace.out`。

程序清单 5-1 test/TestDB.java

```
1 package test;
2
```

```
3 import java.nio.file.*;
4 import java.sql.*;
5 import java.io.*;
6 import java.util.*;
7
8 /**
9  * This program tests that the database and the JDBC driver are correctly configured.
10 * @version 1.03 2018-05-01
11 * @author Cay Horstmann
12 */
13 public class TestDB
14 {
15     public static void main(String args[]) throws IOException
16     {
17         try
18         {
19             runTest();
20         }
21         catch (SQLException ex)
22         {
23             for (Throwable t : ex)
24                 t.printStackTrace();
25         }
26     }
27
28 /**
29  * Runs a test by creating a table, adding a value, showing the table contents, and
30  * removing the table.
31 */
32 public static void runTest() throws SQLException, IOException
33 {
34     try (Connection conn = getConnection();
35          Statement stat = conn.createStatement())
36     {
37         stat.executeUpdate("CREATE TABLE Greetings (Message CHAR(20))");
38         stat.executeUpdate("INSERT INTO Greetings VALUES ('Hello, World!')");
39
40         try (ResultSet result = stat.executeQuery("SELECT * FROM Greetings"))
41         {
42             if (result.next())
43                 System.out.println(result.getString(1));
44         }
45         stat.executeUpdate("DROP TABLE Greetings");
46     }
47 }
48
49 /**
50  * Gets a connection from the properties specified in the file database.properties.
51  * @return the database connection
52 */
53 public static Connection getConnection() throws SQLException, IOException
54 {
55     var props = new Properties();
56     try (InputStream in = Files.newInputStream(Paths.get("database.properties")))
57     {
```

```

57     {
58         props.load(in);
59     }
60     String drivers = props.getProperty("jdbc.drivers");
61     if (drivers != null) System.setProperty("jdbc.drivers", drivers);
62     String url = props.getProperty("jdbc.url");
63     String username = props.getProperty("jdbc.username");
64     String password = props.getProperty("jdbc.password");
65
66     return DriverManager.getConnection(url, username, password);
67 }
68 }
```

API `java.sql.DriverManager 1.1`

- `static Connection getConnection(String url, String user, String password)`

建立一个到指定数据库的连接，并返回一个 `Connection` 对象。

5.4 使用 JDBC 语句

在下面各节中，你将会看到如何使用 JDBC `Statement` 来执行 SQL 语句，获得执行结果，以及处理错误。然后，我们将向你展示一个操作数据库的简单示例。

5.4.1 执行 SQL 语句

在执行 SQL 语句之前，首先需要创建一个 `Statement` 对象。要创建 `Statement` 对象，需要使用调用 `DriverManager.getConnection` 方法所获得的 `Connection` 对象。

```
Statement stat = conn.createStatement();
```

接着，把要执行的 SQL 语句放入字符串中，例如：

```
String command = "UPDATE Books"
    + " SET Price = Price - 5.00"
    + " WHERE Title NOT LIKE '%Introduction%'";
```

然后，调用 `Statement` 接口中的 `executeUpdate` 方法：

```
stat.executeUpdate(command);
```

`executeUpdate` 方法将返回受 SQL 语句影响的行数，或者对不返回行数的语句返回 0。例如，在先前的例子中调用 `executeUpdate` 方法将返回那些降价 5 美元的行数。

`executeUpdate` 方法既可以执行诸如 `INSERT`、`UPDATE` 和 `DELETE` 之类的操作，也可以执行诸如 `CREATE TABLE` 和 `DROP TABLE` 之类的数据定义语句。但是，执行 `SELECT` 查询时必须使用 `executeQuery` 方法。另外还有一个 `execute` 语句可以执行任意的 SQL 语句，此方法通常只用于由用户提供 的交互式查询。

当我们执行查询操作时，通常感兴趣的是查询结果。`executeQuery` 方法会返回一个 `ResultSet` 类型的对象，可以通过它来每次一行地迭代遍历所有查询结果。

```
ResultSet rs = stat.executeQuery("SELECT * FROM Books");
```

分析结果集时通常可以使用类似如下的循环语句代码：

```
while (rs.next())
{
    look at a row of the result set
}
```

! 警告：ResultSet 接口的迭代协议与 java.util.Iterator 接口稍有不同。对于 ResultSet 接口，迭代器初始化时被设定在第一行之前的位置，必须调用 next 方法将它移动到第一行。另外，它没有 hasNext 方法，我们需要不断地调用 next，直至该方法返回 false。

结果集中行的顺序是任意排列的。除非使用 ORDER BY 子句指定行的顺序，否则不能为行序强加任何意义。

查看每一行时，可能希望知道其中每一列的内容，有许多访问器（accessor）方法可以用于获取这些信息。

```
String isbn = rs.getString(1);
double price = rs.getDouble("Price");
```

不同的数据类型有不同的访问器，比如 getString 和 getDouble。每个访问器都有两种形式，一种接受数字型参数，另一种接受字符串参数。当使用数字型参数时，我们指的是该数字所对应的列。例如，rs.getString(1) 返回的是当前行中第一列的值。

! 警告：与数组的索引不同，数据库的列序号是从 1 开始计算的。

当使用字符串参数时，指的是结果集中以该字符串为列名的列。例如，rs.getDouble("Price") 返回列名为 Price 的列所对应的值。使用数字型参数效率更高一些，但是使用字符串参数可以使代码易于阅读和维护。

当 get 方法的类型和列的数据类型不一致时，每个 get 方法都会进行合理的类型转换。例如，调用 rs.getString("Price") 时，该方法会将 Price 列的浮点值转换成字符串。

API **java.sql.Connection 1.1**

- Statement createStatement()

创建一个 Statement 对象，用以执行不带参数的 SQL 查询和更新。

- void close()

立即关闭当前的连接，并释放由它所创建的 JDBC 资源。

API **java.sql.Statement 1.1**

- ResultSet executeQuery(String sqlQuery)

执行给定字符串中的 SQL 语句，并返回一个用于查看查询结果的 ResultSet 对象。

- int executeUpdate(String sqlStatement)

- long executeLargeUpdate(String sqlStatement) 8

执行字符串中指定的 INSERT、UPDATE 或 DELETE 等 SQL 语句。还可以执行数据定义语言 (Data Definition Language, DDL) 的语句，如 CREATE TABLE。返回受影响的行数，如果是没有更新计数的语句，则返回 0。

- `boolean execute(String sqlStatement)`

执行字符串中指定的 SQL 语句。可能会产生多个结果集和更新计数。如果第一个执行结果是结果集，则返回 `true`；反之，返回 `false`。调用 `getResultSet` 或 `getUpdateCount` 方法可以得到第一个执行结果。请参见 5.5.4 节中关于处理多结果集的详细信息。

- `ResultSet getResultSet()`

返回前一条查询语句的结果集。如果前一条语句未产生结果集，则返回 `null` 值。对于每一条执行过的语句，该方法只能被调用一次。

- `int getUpdateCount()`

- `long getLargeUpdateCount() 8`

返回受前一条更新语句影响的行数。如果前一条语句未更新数据库，则返回 -1。对于每一条执行过的语句，该方法只能被调用一次。

- `void close()`

关闭该语句对象以及它所对应的结果集。

- `boolean isClosed() 6`

如果该语句被关闭，则返回 `true`。

- `void closeOnCompletion() 7`

一旦该语句的所有结果集都被关闭，则关闭该语句。

API `java.sql.ResultSet 1.1`

- `boolean next()`

将结果集中的当前行向前移动一行。如果已经到达最后一行的后面，则返回 `false`。注意，初始情况下必须调用该方法才能转到第一行。

- `Xxx getXxx(int columnNumber)`

- `Xxx getXxx(String columnLabel)`

(`Xxx` 指数据类型，例如 `int`、`double`、`String` 和 `Date` 等。)

- `<T> T getObject(int columnIndex, Class<T> type) 7`

- `<T> T getObject(String columnLabel, Class<T> type) 7`

- `void updateObject(int columnIndex, Object x, SQLType targetSqlType) 8`

- `void updateObject(String columnLabel, Object x, SQLType targetSqlType) 8`

用给定的列序号或列标签返回或更新该列的值，并将值转换成指定的类型。列标签是 SQL 的 AS 子句中指定的标签，在没有使用 AS 时，它就是列名。

- `int findColumn(String columnName)`

根据给定的列名，返回该列的序号。

- void close()
立即关闭当前的结果集。
- boolean isClosed() 6
如果该语句被关闭，则返回 true。

5.4.2 管理连接、语句和结果集

每个 Connection 对象都可以创建一个或多个 Statement 对象。同一个 Statement 对象可以用于多个不相关的命令和查询。但是，一个 Statement 对象最多只能有一个打开的结果集。如果需要执行多个查询操作，且需要同时分析查询结果，那么必须创建多个 Statement 对象。

需要说明的是，每个链接上的语句数是有限制的。使用 DatabaseMetaData 接口中的 getMaxStatements 方法可以获取 JDBC 驱动程序支持的同时打开的语句对象的总数。

实际上，我们通常并不需要同时处理多个结果集。如果结果集相互关联，我们可以使用组合查询，这样就只需要分析一个结果。对数据库进行组合查询比使用 Java 程序遍历多个结果集要高效得多。

我们应该确保在一个 Statement 对象上触发新的查询或更新语句之前结束对所有结果集的处理，因为前序查询的所有结果集都会被自动关闭。

使用完 ResultSet、Statement 或 Connection 对象后，应立即调用 close 方法。这些对象都使用了规模较大的数据结构，它们会占用数据库存服务器上的有限资源。

Statement 对象的 close 方法将自动关闭所有与其相关联的结果集。同样地，调用 Connection 类的 close 方法将关闭该连接上的所有语句。

反过来的情况是，可以在 Statement 上调用 closeOnCompletion 方法，在其所有结果集都被关闭后，该语句会立即被自动关闭。

如果所用连接都是短时的，那么无须操心语句和结果集的关闭。只需将 close 语句放在带资源的 try 语句中，以便确保连接对象不可能继续保持打开状态。

```
try (Connection conn = . . .)
{
    Statement stat = conn.createStatement();
    ResultSet result = stat.executeQuery(queryString);
    process query result
}
```

5.4.3 分析 SQL 异常

每个 SQLException 都有一个由多个 SQLException 对象构成的链，这些对象可以通过 getNextException 方法获取。这个异常链是每个异常都具有的由 Throwable 对象构成的“成因”链之外的异常链（请参见卷 I 第 7 章以了解 Java 异常的详细信息），因此，我们需要用两个嵌套的循环来完整枚举所有的异常。幸运的是，SQLException 类得到了增强，实现了 Iterable<Throwable> 接口，其 iterator() 方法可以产生一个 Iterator<Throwable>，这个迭代器可以迭代这两个链，首先迭代第一个 SQLException 的成因链，然后迭代下一个 SQLException，以此类推。我们可以

直接使用下面这个改进的 for 循环：

```
for (Throwable t : sqlException)
{
    do something with t
}
```

可以在 SQLException 上调用 getSQLState 和 getErrorCode 方法来进一步分析它，其中第一个方法将产生符合 X/Open 或 SQL:2003 标准的字符串（调用 DatabaseMetaData 接口的 getSQLStateType 方法可以查出驱动程序所使用的标准）。而错误代码是与具体的提供商相关的。

SQL 异常按照层次结构树的方式组织到了一起（如图 5-5 所示），这使得我们可以按照与提供商无关的方式来捕获具体的错误类型。

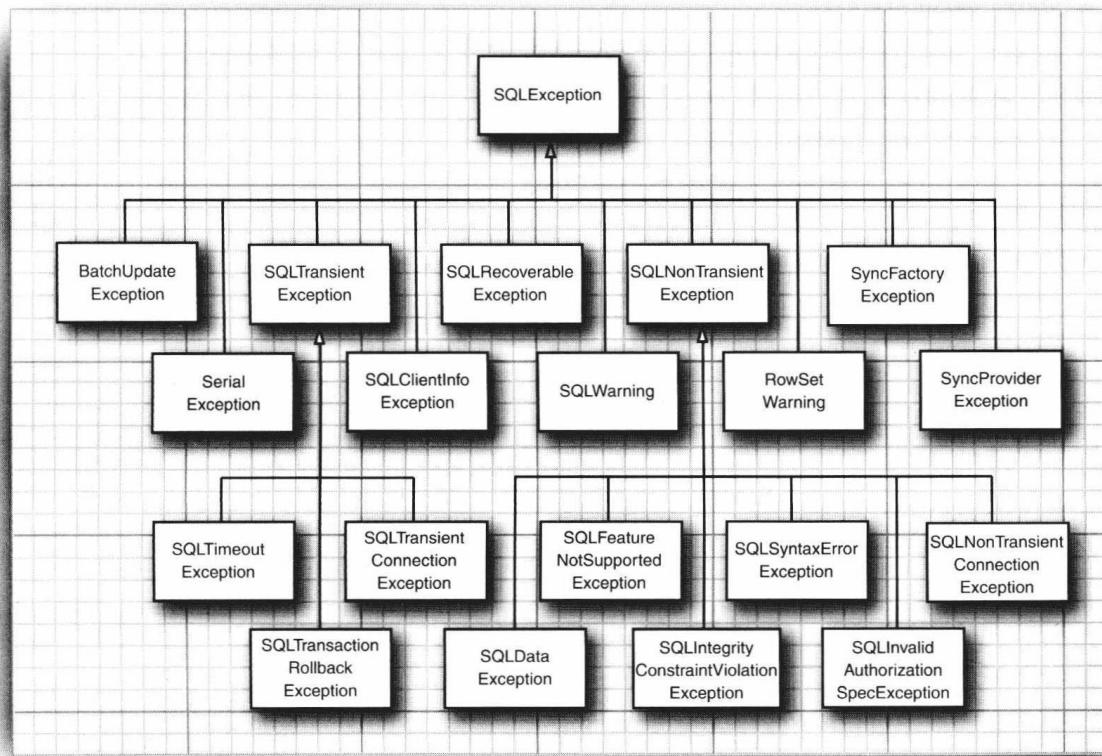


图 5-5 SQL 异常类型

另外，数据库驱动程序可以将非致命问题作为警告报告，我们可以从连接、语句和结果集中获取这些警告。SQLWarning 类是 SQLException 的子类（尽管 SQLWarning 不会被当作异常抛出），我们可以调用 getSQLState 和 getErrorCode 来获取有关警告的更多信息。与 SQL 异常类似，警告也是串成链的。要获得所有的警告，可以使用下面的循环：

```
SQLWarning w = stat.getWarning();
while (w != null)
{
```

```

do something with w
w = w.nextWarning();
}

```

当数据从数据库中读出并意外被截断时，SQLWarning 的 DataTruncation 子类就派上用场了。如果数据截断发生在更新语句中，那么 DataTruncation 对象将会被当作异常抛出。

API `java.sql.SQLException` 1.1

- `SQLException getNextException()`
返回链接到该 SQL 异常的下一个 SQL 异常，或者在到达链尾时返回 null。
- `Iterator<Throwable> iterator()` 6
获取迭代器，可以迭代链接的 SQL 异常和它们的成因。
- `String getSQLState()`
获取“SQL 状态”，即标准化的错误代码。
- `int getErrorCode()`
获取提供商相关的错误代码。

API `java.sql.SQLWarning` 1.1

- `SQLWarning getNextWarning()`
返回链接到该警告的下一个警告，或者在到达链尾时返回 null。

API `java.sql.Connection` 1.1

`java.sql.Statement` 1.1

`java.sql.ResultSet` 1.1

- `SQLWarning getWarnings()`
返回未处理警告中的第一个，或者在没有未处理警告时返回 null。

API `java.sql.DataTruncation` 1.1

- `boolean getParameter()`
如果在参数上进行了数据截断，则返回 true；如果在列上进行了数据截断，则返回 false。
- `int getIndex()`
返回被截断的参数或列的索引。
- `int getDataSize()`
返回应该被传输的字节数量，或者在该值未知的情况下返回 -1。
- `int getTransferSize()`
返回实际被传输的字节数量，或者在该值未知的情况下返回 -1。

5.4.4 组装数据库

至此，大家也许都迫不及待地想编写一个真正实用的 JDBC 程序了。如果我们可以编写

一段程序来执行之前所介绍的那些巧妙的查询，那当然很好。不过，在此之前我们还有一个问题没有解决：目前数据库中还没有数据。我们需要组装数据库，并且也确实存在一种简单方法可以实现此目的：用一系列的 SQL 指令来创建数据表并向其中插入数据。大多数数据库程序都可以处理来自文本文件中的一系列 SQL 指令，但是在语句终止符和其他一些文法问题上，这些数据库程序之间存在着令人讨厌的差异。

正是由于这个原因，我们使用 JDBC 创建了一个简单的程序，它从文件中读取 SQL 指令，其中一条指令占据一行，然后执行它们。

该程序专门用于从下列格式的文本文件中读取数据：

```
CREATE TABLE Publishers (Publisher_Id CHAR(6), Name CHAR(30), URL CHAR(80));
INSERT INTO Publishers VALUES ('0201', 'Addison-Wesley', 'www.aw-bc.com');
INSERT INTO Publishers VALUES ('0471', 'John Wiley & Sons', 'www.wiley.com');
...

```

程序清单 5-2 是用来读取 SQL 语句文件以及执行这些语句的程序代码。通读这些代码并不重要，我们在这里只是提供了这样的程序，使你能够组装数据库并运行本章剩余部分的代码。

请确认你的数据库服务器是在运行的，然后可以使用如下方法运行该程序：

```
java -classpath driverPath:. exec.ExecSQL Books.sql
java -classpath driverPath:. exec.ExecSQL Authors.sql
java -classpath driverPath:. exec.ExecSQL Publishers.sql
java -classpath driverPath:. exec.ExecSQL BooksAuthors.sql
```

在运行程序之前，请检查一下 `database.properties` 文件是否已经针对你的运行环境进行了正确设置。请查看 5.3.5 节。

 **注释：**你的数据库可能也包含直接读取 SQL 文件的工具，例如，在使用 Derby 时，可以运行下面的命令：

```
java -jar derby/lib/derbyrun.jar ij -p ij.properties Books.sql
```

(`ij.properties` 文件在 5.3.3 节中描述过。)

在用于 ExecSQL 命令的数据格式中，我们允许每行的结尾都可以有一个可选的分号，因为大多数数据库工具都希望使用这种格式。

下面将简要介绍一下 ExecSQL 程序的操作步骤。

1. 连接数据库。`getConnection` 方法读取 `database.properties` 文件中的属性信息，并将属性 `jdbc.drivers` 添加到系统属性中。驱动程序管理器使用属性 `jdbc.drivers` 加载相应的驱动程序。`getConnection` 方法使用 `jdbc.url`、`jdbc.username` 和 `jdbc.password` 等属性打开数据库连接。
2. 打开包含 SQL 语句的文件。如果未提供任何文件名，则在控制台中提示用户输入语句。
3. 使用泛化的 `execute` 方法执行每条语句。如果它返回 `true`，则说明该语句产生了一个结果集。我们为图书数据库提供的 4 个 SQL 文件都以一条 `SELECT *` 语句结束，这样就可以看到数据是否已成功插入到了数据库中。
4. 如果产生了结果集，则打印出结果。因为这是一个泛化的结果集，所以我们必须使用

元数据来确定该结果的列数。更多的信息请查看 5.8 节。

5. 如果运行过程中出现 SQL 异常，则打印出这个异常以及所有可能包含在其中的与其链接在一起的相关异常。

6. 关闭数据库连接。

程序清单 5-2 给出了该程序的代码。

程序清单 5-2 exec/ExecSQL.java

```

1 package exec;
2
3 import java.io.*;
4 import java.nio.charset.*;
5 import java.nio.file.*;
6 import java.util.*;
7 import java.sql.*;
8
9 /**
10  * Executes all SQL statements in a file. Call this program as <br>
11  * java -classpath driverPath:. ExecSQL commandFile
12  *
13  * @version 1.33 2018-05-01
14  * @author Cay Horstmann
15  */
16 class ExecSQL
17 {
18     public static void main(String args[]) throws IOException
19     {
20         try (Scanner in = args.length == 0 ? new Scanner(System.in)
21              : new Scanner(Paths.get(args[0]), StandardCharsets.UTF_8))
22         {
23             try (Connection conn = getConnection();
24                  Statement stat = conn.createStatement())
25             {
26                 while (true)
27                 {
28                     if (args.length == 0) System.out.println("Enter command or EXIT to exit:");
29
30                     if (!in.hasNextLine()) return;
31
32                     String line = in.nextLine().trim();
33                     if (line.equalsIgnoreCase("EXIT")) return;
34                     if (line.endsWith(";")) // remove trailing semicolon
35                         line = line.substring(0, line.length() - 1);
36                     try
37                     {
38                         boolean isResult = stat.execute(line);
39                         if (isResult)
40                         {
41                             try (ResultSet rs = stat.getResultSet())
42                             {
43                                 showResultSet(rs);
44                             }
45                         }
46                     }
47                 }
48             }
49         }
50     }
51 }
```

```
45         }
46     else
47     {
48         int updateCount = stat.getUpdateCount();
49         System.out.println(updateCount + " rows updated");
50     }
51 }
52 catch (SQLException e)
53 {
54     for (Throwable t : e)
55         t.printStackTrace();
56 }
57 }
58 }
59 catch (SQLException e)
60 {
61     for (Throwable t : e)
62         t.printStackTrace();
63 }
64 }
65 }

/**
 * Gets a connection from the properties specified in the file database.properties
 * @return the database connection
 */
public static Connection getConnection() throws SQLException, IOException
{
    var props = new Properties();
    try (InputStream in = Files.newInputStream(Paths.get("database.properties")))
    {
        props.load(in);
    }
    String drivers = props.getProperty("jdbc.drivers");
    if (drivers != null) System.setProperty("jdbc.drivers", drivers);

    String url = props.getProperty("jdbc.url");
    String username = props.getProperty("jdbc.username");
    String password = props.getProperty("jdbc.password");

    return DriverManager.getConnection(url, username, password);
}

/**
 * Prints a result set.
 * @param result the result set to be printed
 */
public static void showResultSet(ResultSet result) throws SQLException
{
    ResultSetMetaData metaData = result.getMetaData();
    int columnCount = metaData.getColumnCount();

    for (int i = 1; i <= columnCount; i++)
    {
```

```

99         if (i > 1) System.out.print(", ");
100        System.out.print(metaData.getColumnLabel(i));
101    }
102    System.out.println();
103
104   while (result.next())
105   {
106     for (int i = 1; i <= columnCount; i++)
107     {
108       if (i > 1) System.out.print(", ");
109       System.out.print(result.getString(i));
110     }
111     System.out.println();
112   }
113 }
114 }
```

5.5 执行查询操作

在这一节中，我们将编写一段用于对 COREJAVA 数据库执行查询操作的程序。为了使程序可以正常运行，必须按照上一节中的说明用表组装 COREJAVA 数据库。

在查询数据库时，可以选择作者和出版社，或者将这两项中的一项设置为“Any”。

还可以修改数据库中的数据。选择一家出版社，然后输入金额。该出版社对应的所有价格都将按照填入的金额进行调整，同时程序将显示被修改的行数。修改完价格以后，可以运行一个查询操作，以核实新的价格。

5.5.1 预备语句

在这个程序中，我们使用了一个新的特性，即预备语句（prepared statement）。如果我们要查询某个出版社的所有图书而不考虑具体的作者，那么该查询的 SQL 语句如下：

```

SELECT Books.Price, Books
FROM Books, Publishers
WHERE Books.Publisher_Id = Publishers.Publisher_Id
AND Publishers.Name = the name from the list box
```

我们没有必要在每次触发一个这样的查询时都建立新的查询语句，而是可以准备一个带有宿主变量的查询语句，每次查询时只需为该变量填入不同的字符串就可以反复多次地使用该语句。这一技术改进了查询性能，每当数据库执行一个查询时，它总是首先通过计算来确定查询策略，以便高效地执行查询操作。通过事先准备好查询并多次重用它，我们就可以确保查询所需的准备步骤只被执行一次。

在预备查询语句中，每个宿主变量都用“?”来表示。如果存在一个以上的变量，那么在设置变量值时必须注意“?”的位置。例如，如果我们的预备查询为如下形式：

```

String publisherQuery
= "SELECT Books.Price, Books"
```

```
+ " FROM Books, Publishers"
+ " WHERE Books.Publisher_Id = Publishers.Publisher_Id AND Publishers.Name = ?";
PreparedStatement stat = conn.prepareStatement(publisherQuery);
```

在执行预备语句之前，必须使用 `set` 方法将变量绑定到实际的值上。和 `ResultSet` 接口中的 `get` 方法类似，针对不同的数据类型也有不同的 `set` 方法。在本例中，我们为出版社名称设置了一个字符串值。

```
stat.setString(1, publisher);
```

第一个参数指的是需要设置的宿主变量的位置，位置 1 表示第一个“?”。第二个参数指的是赋予宿主变量的值。

如果想要重用已经执行过的预备查询语句，那么除非使用 `set` 方法或调用 `clearParameters` 方法，否则所有宿主变量的绑定都不会改变。这就意味着，在从一个查询到另一个查询的过程中，只需使用 `setXxx` 方法重新绑定那些需要改变的变量即可。

一旦为所有变量都绑定了具体的值，就可以执行预备语句了：

```
ResultSet rs = stat.executeQuery();
```

 **提示：**通过连接字符串来手动构建查询显得非常枯燥乏味，而且存在潜在的危险。你必须注意像引号这样的特殊字符，而且如果查询中涉及用户的输入，那就还需要警惕注入攻击。因此，只要查询涉及变量，就应该使用预备语句。

价格更新操作可以由 `UPDATE` 语句实现。请注意，我们调用的是 `executeUpdate` 方法，而非 `executeQuery` 方法，因为 `UPDATE` 语句不返回结果集。`executeUpdate` 的返回值为被修改过的行数。

```
int r = stat.executeUpdate();
System.out.println(r + " rows updated");
```

 **注释：**在相关的 `Connection` 对象关闭之后，`PreparedStatement` 对象也就变得无效了。不过，许多数据库通常都会自动缓存预备语句。如果相同的查询被预备两次，数据库通常会直接重用查询策略。因此，无须过多考虑调用 `prepareStatement` 的开销。

下面简要说明示例程序的结构：

- 通过执行两个查询得到数据库中所有的作者和出版社名称，作者和出版社数组列表由此组装而成。
- 涉及作者的查询比较复杂。因为一本书可能有多个作者，`BooksAuthors` 表给出了作者和图书之间的对应关系。例如，ISBN 号为 0-201-96426-0 的图书有两个作者，其代号为：DATE 和 DARW。以下为 `BooksAuthors` 表中的两行记录：

```
0-201-96426-0, DATE, 1
0-201-96426-0, DARW, 2
```

`BooksAuthors` 表中第三列指的是作者的顺序（我们不能只使用表中行的位置，在关系表中没有固定的行顺序）。因此，查询时需要连接 `Books` 表、`BooksAuthors` 表和 `Authors` 表，以便和用户所选的作者名进行比较。

```

SELECT Books.Price, Books FROM Books, BooksAuthors, Authors, Publishers
WHERE Authors.Author_Id = BooksAuthors.Author_Id AND BooksAuthors.ISBN = Books.ISBN
AND Books.Publisher_Id = Publishers.Publisher_Id AND Authors.Name = ?
AND Publishers.Name = ?

```

 提示：许多程序员都不喜欢使用如此复杂的 SQL 语句。比较常见的方法是使用大量的 Java 代码来迭代多个结果集，但是这种方法效率非常低。通常，使用数据库的查询代码要比使用 Java 程序好得多——这是数据库的核心竞争力之一。一般而言，可以使用 SQL 解决的问题，就不要使用 Java 程序。

- change Prices 方法执行了一条 UPDATE 语句。注意，UPDATE 语句中的 WHERE 子句需要使用出版社代码，而我们只知道出版社名称。这个问题可以使用嵌套子查询来解决。

```

UPDATE Books
SET Price = Price + ?
WHERE Books.Publisher_Id = (SELECT Publisher_Id FROM Publishers WHERE Name = ?)

```

程序清单 5-3 给出了程序的完整代码。

程序清单 5-3 query/QueryTest.java

```

1 package query;
2
3 import java.io.*;
4 import java.nio.file.*;
5 import java.sql.*;
6 import java.util.*;
7
8 /**
9  * This program demonstrates several complex database queries.
10 * @version 1.31 2018-05-01
11 * @author Cay Horstmann
12 */
13 public class QueryTest
14 {
15     private static final String allQuery = "SELECT Books.Price, Books.Title FROM Books";
16
17     private static final String authorPublisherQuery = "SELECT Books.Price, Books.Title"
18         + " FROM Books, BooksAuthors, Authors, Publishers"
19         + " WHERE Authors.Author_Id = BooksAuthors.Author_Id AND BooksAuthors.ISBN = Books.ISBN"
20         + " AND Books.Publisher_Id = Publishers.Publisher_Id AND Authors.Name = ?"
21         + " AND Publishers.Name = ?";
22
23     private static final String authorQuery
24         = "SELECT Books.Price, Books.Title FROM Books, BooksAuthors, Authors"
25         + " WHERE Authors.Author_Id = BooksAuthors.Author_Id"
26         + " AND BooksAuthors.ISBN = Books.ISBN"
27         + " AND Authors.Name = ?";
28
29     private static final String publisherQuery
30         = "SELECT Books.Price, Books.Title FROM Books, Publishers"
31         + " WHERE Books.Publisher_Id = Publishers.Publisher_Id AND Publishers.Name = ?";
32

```

```
33 private static final String priceUpdate = "UPDATE Books SET Price = Price + ? "
34     + " WHERE Books.Publisher_Id = (SELECT Publisher_Id FROM Publishers WHERE Name = ?)";
35
36 private static Scanner in;
37 private static ArrayList<String> authors = new ArrayList<>();
38 private static ArrayList<String> publishers = new ArrayList<>();
39
40 public static void main(String[] args) throws IOException
41 {
42     try (Connection conn = getConnection())
43     {
44         in = new Scanner(System.in);
45         authors.add("Any");
46         publishers.add("Any");
47         try (Statement stat = conn.createStatement())
48         {
49             // Fill the authors array list
50             var query = "SELECT Name FROM Authors";
51             try (ResultSet rs = stat.executeQuery(query))
52             {
53                 while (rs.next())
54                     authors.add(rs.getString(1));
55             }
56
57             // Fill the publishers array list
58             query = "SELECT Name FROM Publishers";
59             try (ResultSet rs = stat.executeQuery(query))
60             {
61                 while (rs.next())
62                     publishers.add(rs.getString(1));
63             }
64         }
65         var done = false;
66         while (!done)
67         {
68             System.out.print("Q)uery C)hange prices E)xit: ");
69             String input = in.nextLine().toUpperCase();
70             if (input.equals("Q"))
71                 executeQuery(conn);
72             else if (input.equals("C"))
73                 changePrices(conn);
74             else
75                 done = true;
76         }
77     }
78     catch (SQLException e)
79     {
80         for (Throwable t : e)
81             System.out.println(t.getMessage());
82     }
83 }
84
85 /**
86 * Executes the selected query.
```

```

87     * @param conn the database connection
88     */
89     private static void executeQuery(Connection conn) throws SQLException
90     {
91         String author = select("Authors:", authors);
92         String publisher = select("Publishers:", publishers);
93         PreparedStatement stat;
94         if (!author.equals("Any") && !publisher.equals("Any"))
95         {
96             stat = conn.prepareStatement(authorPublisherQuery);
97             stat.setString(1, author);
98             stat.setString(2, publisher);
99         }
100        else if (!author.equals("Any") && publisher.equals("Any"))
101        {
102            stat = conn.prepareStatement(authorQuery);
103            stat.setString(1, author);
104        }
105        else if (author.equals("Any") && !publisher.equals("Any"))
106        {
107            stat = conn.prepareStatement(publisherQuery);
108            stat.setString(1, publisher);
109        }
110        else
111            stat = conn.prepareStatement(allQuery);
112
113        try (ResultSet rs = stat.executeQuery())
114        {
115            while (rs.next())
116                System.out.println(rs.getString(1) + ", " + rs.getString(2));
117        }
118    }
119
120 /**
121 * Executes an update statement to change prices.
122 * @param conn the database connection
123 */
124 public static void changePrices(Connection conn) throws SQLException
125 {
126     String publisher = select("Publishers:", publishers.subList(1, publishers.size()));
127     System.out.print("Change prices by: ");
128     double priceChange = in.nextDouble();
129     PreparedStatement stat = conn.prepareStatement(priceUpdate);
130     stat.setDouble(1, priceChange);
131     stat.setString(2, publisher);
132     int r = stat.executeUpdate();
133     System.out.println(r + " records updated.");
134 }
135
136 /**
137 * Asks the user to select a string.
138 * @param prompt the prompt to display
139 * @param options the options from which the user can choose
140 * @return the option that the user chose

```

```

141     */
142     public static String select(String prompt, List<String> options)
143     {
144         while (true)
145         {
146             System.out.println(prompt);
147             for (int i = 0; i < options.size(); i++)
148                 System.out.printf("%2d) %s%n", i + 1, options.get(i));
149             int sel = in.nextInt();
150             if (sel > 0 && sel <= options.size())
151                 return options.get(sel - 1);
152         }
153     }
154
155     /**
156      * Gets a connection from the properties specified in the file database.properties.
157      * @return the database connection
158      */
159     public static Connection getConnection() throws SQLException, IOException
160     {
161         var props = new Properties();
162         try (InputStream in = Files.newInputStream(Paths.get("database.properties")))
163         {
164             props.load(in);
165         }
166
167         String drivers = props.getProperty("jdbc.drivers");
168         if (drivers != null) System.setProperty("jdbc.drivers", drivers);
169
170         String url = props.getProperty("jdbc.url");
171         String username = props.getProperty("jdbc.username");
172         String password = props.getProperty("jdbc.password");
173
174         return DriverManager.getConnection(url, username, password);
175     }
176 }

```

API ***java.sql.Connection*** 1.1

- `PreparedStatement prepareStatement(String sql)`

返回一个含预编译语句的 `PreparedStatement` 对象。字符串 `sql` 代表一个 SQL 语句，该语句可以包含一个或多个由 ? 字符指明的参数占位符。

API ***java.sql.PreparedStatement*** 1.1

- `void setXxx(int n, Xxx x)`
(`Xxx` 指 `int`、`double`、`String`、`Date` 之类的数据类型) 设置第 `n` 个参数值为 `x`。
- `void clearParameters()`
清除预备语句中的所有当前参数。
- `ResultSet executeQuery()`

执行预备 SQL 查询，并返回一个 `ResultSet` 对象。

- `int executeUpdate()`

执行预备 SQL 语句 `INSERT`、`UPDATE` 或 `DELETE`，这些语句由 `PreparedStatement` 对象表示。该方法返回在执行上述语句过程中所有受影响的记录总数。如果执行的是数据定义语言（DDL）中的语句，如 `CREATE TABLE`，则该方法返回 0。

5.5.2 读写 LOB

除了数字、字符串和日期之外，许多数据库还可以存储大对象，例如图片或其他数据。在 SQL 中，二进制大对象称为 `BLOB`，字符型大对象称为 `CLOB`。

要读取 `LOB`，需要执行 `SELECT` 语句，然后在 `ResultSet` 上调用 `getBlob` 或 `getBlob` 方法，这样就可以获得 `Blob` 或 `Clob` 类型的对象。要从 `Blob` 中获取二进制数据，可以调用 `getBytes` 或 `getBinaryStream`。例如，如果你有一张保存图书封面图像的表，那么就可以像下面这样获取一张图像：

```
PreparedStatement stat = conn.prepareStatement("SELECT Cover FROM BookCovers WHERE ISBN=?");
...
stat.setInt(1, isbn);
try (ResultSet result = stat.executeQuery())
{
    if (result.next())
    {
        Blob coverBlob = result.getBlob(1);
        Image coverImage = ImageIO.read(coverBlob.getBinaryStream());
    }
}
```

类似地，如果获取了 `Clob` 对象，那么就可以通过调用 `getSubString` 或 `getCharacterStream` 方法来获取其中的字符数据。

要将 `LOB` 置于数据库中，需要在 `Connection` 对象上调用 `createBlob` 或 `createClob`，然后获取一个用于该 `LOB` 的输出流或写出器，写出数据，并将该对象存储到数据库中。例如，下面展示了如何存储一张图像：

```
Blob coverBlob = connection.createBlob();
int offset = 0;
OutputStream out = coverBlob.setBinaryStream(offset);
ImageIO.write(coverImage, "PNG", out);
PreparedStatement stat = conn.prepareStatement("INSERT INTO Cover VALUES (?, ?)");
stat.setInt(1, isbn);
stat.setBinaryStream(2, coverBlob);
stat.executeUpdate();
```

API `java.sql.ResultSet` 1.1

- `Blob getBlob(int columnIndex)` 1.2
- `Blob getBlob(String columnLabel)` 1.2
- `Clob getBlob(int columnIndex)` 1.2

- `Clob getBlob(String columnLabel)` 1.2

获取给定列的 BLOB 或 CLOB。

API `java.sql.Blob` 1.2

- `long length()`
获取该 BLOB 的长度。
- `byte[] getBytes(long startPosition, long length)`
获取该 BLOB 中给定范围的数据。
- `InputStream getBinaryStream()`
- `InputStream getBinaryStream(long startPosition, long length)`
返回一个输入流，用于读取该 BLOB 中全部或给定范围的数据。
- `OutputStream setBinaryStream(long startPosition)` 1.4
返回一个输出流，用于从给定位置开始写入该 BLOB。

API `java.sql.Clob` 1.4

- `long length()`
获取该 CLOB 中的字符总数。
- `String getSubString(long startPosition, long length)`
获取该 CLOB 中给定范围的字符。
- `Reader getCharacterStream()`
- `Reader getCharacterStream(long startPosition, long length)`
返回一个读入器（而不是流），用于读取 CLOB 中全部或给定范围的数据。
- `Writer setCharacterStream(long startPosition)` 1.4
返回一个写出器（而不是流），用于从给定位置开始写入该 CLOB。

API `java.sql.Connection` 1.1

- `Blob createBlob()` 6
- `Clob createClob()` 6
创建一个空的 BLOB 或 CLOB。

5.5.3 SQL 转义

“转义”语法是各种数据库普遍支持的特性，但是数据库使用的是与数据库相关的语法变体，因此，将转义语法转译为特定数据库的语法是 JDBC 驱动程序的任务之一。

转义主要用于下列场景：

- 日期和时间字面常量
- 调用标量函数
- 调用存储过程

- 外连接
- 在 LIKE 子句中的转义字符

日期和时间字面常量随数据库的不同而变化很大。要嵌入日期或时间字面常量，需要按照 ISO 8601 格式 (<http://www.cl.cam.ac.uk/~mgk25/iso-time.html>) 指定它的值，之后驱动程序会将其转译为本地格式。应该使用 d、t、ts 来表示 DATE、TIME 和 TIMESTAMP 值：

```
{d '2008-01-24'}
{t '23:59:59'}
{ts '2008-01-24 23:59:59.999'}
```

标量函数（scalar function）是指仅返回单个值的函数。在数据库中包含大量的函数，但是不同的数据库中这些函数名存在着差异。JDBC 规范提供了标准的名字，并将其转译为数据库相关的名字。要调用函数，需要像下面这样嵌入标准的函数名和参数：

```
{fn left(?, 20)}
{fn user()}
```

在 JDBC 规范中可以找到它支持的函数名的完整列表。

存储过程（stored procedure）是在数据库中执行的用数据库相关的语言编写的过程。要调用存储过程，需要使用 call 转义命令，在存储过程没有任何参数时，可以不用加上括号。另外，应该用 = 来捕获存储过程的返回值：

```
{call PROC1(?, ?)}
{call PROC2}
{call ? = PROC3(?)}
```

两个表的外连接（outer join）并不要求每个表的所有行都要根据连接条件进行匹配，例如，假设有如下的查询：

```
SELECT * FROM {oj Books LEFT OUTER JOIN Publishers
ON Books.Publisher_Id = Publisher.Publisher_Id}
```

这个查询的执行结果中将包含有 Publisher_Id 在 Publishers 表中没有任何匹配的书，其中，Publisher_ID 为 NULL 值的行，就表示不存在任何匹配。如果应该使用 RIGHT OUTER JOIN，就可以囊括没有任何匹配图书的出版商，而使用 FULL OUTER JOIN 可以同时返回这两类没有任何匹配的信息。由于并非所有的数据库对于这些连接都使用标准的写法，因此需要使用转义语法。

最后一种情况，_ 和 % 字符在 LIKE 子句中具有特殊含义，用来匹配一个字符或一个字符序列。目前并不存在任何在字面上使用它们的标准方式，所以如果想要匹配所有包含 _ 字符的字符串，就必须使用下面的结构：

```
... WHERE ? LIKE %!_% {escape '!'}
```

这里我们将 ! 定义为转义字符，而 !_ 组合表示字面常量下划线。

5.5.4 多结果集

在执行存储过程，或者在使用允许在单个查询中提交多个 SELECT 语句的数据库时，一个

查询有可能会返回多个结果集。下面是获取所有结果集的步骤：

1. 使用 execute 方法来执行 SQL 语句。
2. 获取第一个结果集或更新计数。
3. 重复调用 getMoreResults 方法以移动到下一个结果集。
4. 当不存在更多的结果集或更新计数时，完成操作。

如果由多结果集构成的链中的下一项是结果集，execute 和 getMoreResults 方法将返回 true，而如果在链中的下一项不是更新计数，getUpdateCount 方法将返回 -1。

下面的循环可以遍历所有的结果：

```
boolean isResult = stat.execute(command);
boolean done = false;
while (!done)
{
    if (isResult)
    {
        ResultSet result = stat.getResultSet();
        do something with result
    }
    else
    {
        int updateCount = stat.getUpdateCount();
        if (updateCount >= 0)
            do something with updateCount
        else
            done = true;
    }
    if (!done) isResult = stat.getMoreResults();
}
```

java.sql.Statement 1.1

- boolean getMoreResults()
- boolean getMoreResults(int current) 6

获取该语句的下一个结果集，Current 参数是 CLOSE_CURRENT_RESULT (默认值)，KEEP_CURRENT_RESULT 或 CLOSE_ALL_RESULTS 之一。如果存在下一个结果集，并且它确实是一个结果集，则返回 true。

5.5.5 获取自动生成的键

大多数数据库都支持某种在数据库中对行自动编号的机制。但是，不同的提供商所提供的机制之间存在着很大的差异，而这些自动编号的值经常用作主键。尽管 JDBC 没有提供独立于提供商的自动生成键的解决方案，但是它提供了获取自动生成键的有效途径。当我们向数据表中插入一个新行，且其键自动生成时，可以用下面的代码来获取这个键：

```
stat.executeUpdate(insertStatement, Statement.RETURN_GENERATED_KEYS);
ResultSet rs = stat.getGeneratedKeys();
if (rs.next())
```

```

{
    int key = rs.getInt(1);
    ...
}

```

API **java.sql.Statement 1.1**

- `boolean execute(String statement, int autogenerated)` 1.4
- `int executeUpdate(String statement, int autogenerated)` 1.4

像前面描述的那样执行给定的 SQL 语句，如果 `autogenerated` 被设置为 `Statement.RETURN_GENERATED_KEYS`，并且该语句是一条 `INSERT` 语句，那么第一列中就是自动生成的键。

5.6 可滚动和可更新的结果集

我们前面已经介绍过，使用 `ResultSet` 接口中的 `next` 方法可以迭代遍历结果集中的所有行。对于一个只需要分析数据的程序来说，这显然已经足够了。但是，如果是用于展示一张表或查询结果的可视化数据显示（参见图 5-4），我们通常会希望用户可以在结果集上前后移动。对于可滚动结果集而言，我们可以在其中向前或向后移动，甚至可以跳到任意位置。

另外，一旦向用户显示了结果集中的内容，他们就可能希望编辑这些内容。在可更新的结果集中，可以以编程方式来更新其中的项，使得数据库可以自动更新数据。我们将在下面的小节中讨论这些功能。

5.6.1 可滚动的结果集

默认情况下，结果集是不可滚动和不可更新的。为了从查询中获取可滚动的结果集，必须使用下面的方法得到一个不同的 `Statement` 对象：

```
Statement stat = conn.createStatement(type, concurrency);
```

如果要获得预备语句，请调用下面的方法：

```
PreparedStatement stat = conn.prepareStatement(command, type, concurrency);
```

表 5-6 和表 5-7 列出了 `type` 和 `concurrency` 的所有可能值，可以有以下几种选择：

- 是否希望结果集是可滚动的？如果不需要，则使用 `ResultSet.TYPE_FORWARD_ONLY`。
- 如果结果集是可滚动的，且数据库在查询生成结果集之后发生了变化，那么是否希望结果集反映出这些变化？（在我们的讨论中，我们假设将可滚动的结果集设置为 `ResultSet.TYPE_SCROLL_INSENSITIVE`。这个设置将使结果集“感应”不到查询结束后出现的数据库变化。）
- 是否希望通过编辑结果集就可以更新数据库？（详细说明请参见下一节内容。）

表 5-6 `ResultSet` 类的 `type` 值

值	解释
<code>TYPE_FORWARD_ONLY</code>	结果集不能滚动（默认值）

(续)

值	解释
TYPE_SCROLL_INSENSITIVE	结果集可以滚动，但对数据库变化不敏感
TYPE_SCROLL_SENSITIVE	结果集可以滚动，且对数据库变化敏感

表 5-7 ResultSet 类的 Concurrency 值

值	解释
CONCUR_READ_ONLY	结果集不能用于更新数据库（默认值）
CONCUR_UPDATABLE	结果集可以用于更新数据库

例如，如果只想滚动遍历结果集，而不想编辑它的数据，那么可以使用以下语句：

```
Statement stat = conn.createStatement()
    ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
```

现在，通过调用以下方法获得的所有结果集都将 是可滚动的。

```
ResultSet rs = stat.executeQuery(query);
```

可滚动的结果集有一个游标，用以指示当前位置。

注释：并非所有的数据库驱动程序都支持可滚动和可更新的结果集。（使用 DatabaseMetaData 接口中的 supportsResultSetType 和 supportsResultSetConcurrency 方法，我们可以获知在使用特定的驱动程序时，某个数据库究竟支持哪些结果集类型以及哪些并发模式。）即便是数据库支持所有的结果集模式，某个特定的查询也可能无法产生带有所要求的所有属性的结果集。（例如，一个复杂查询的结果集就有可能是不可更新的结果集。）在这种情况下，executeQuery 方法将返回一个功能较少的 ResultSet 对象，并添加一个 SQLWarning 到连接对象中。（参见 5.4.3 节有关如何获取警告信息的内容）或者，也可以使用 ResultSet 接口中的 getType 和 getConcurrency 方法查看结果集实际支持的模式。如果不检查结果集的功能就发起一个不支持的操作，比如对不可滚动的结果集调用 previous 方法，那么该操作将抛出一个 SQLException 异常。

在结果集上滚动是非常简单的，可以使用

```
if (rs.previous()) . . .;
```

向后滚动。如果游标位于一个实际的行上，那么该方法将返回 true；如果游标位于第一行之前，那么返回 false。

可以使用以下调用将游标向后或向前移动多行：

```
rs.relative(n);
```

如果 n 为正数，游标将向前移动。如果 n 为负数，游标将向后移动。如果 n 为 0，那么调用该方法将不起任何作用。如果试图将游标移动到当前行集的范围之外，即根据 n 值的正负号，游标需要被设置在最后一行之后或第一行之前，那么，该方法将返回 false，且不移动

游标。如果游标位于一个实际的行上，那么该方法将返回 true。

或者，还可以将游标设置到指定的行号上：

```
rs.absolute(n);
```

调用以下方法将返回当前行的行号：

```
int currentRow = rs.getRow();
```

结果集中第一行的行号为 1。如果返回值为 0，那么游标当前不在任何行上，它要么位于第一行之前，要么位于最后一行之后。

`first`、`last`、`beforeFirst` 和 `afterLast` 这些简便方法用于将游标移动到第一行、最后一行、第一行之前或最后一行之后。

最后，`isFirst`、`isLast`、`isBeforeFirst` 和 `isAfterLast` 用于测试游标是否位于这些特殊位置上。

使用可滚动的结果集是非常简单的，将查询数据放入缓存中的复杂工作是由数据库驱动程序在后台完成的。

5.6.2 可更新的结果集

如果希望编辑结果集中的数据，并且将结果集上的数据变更自动反映到数据库中，那么就必须使用可更新的结果集。可更新的结果集并非必须是可滚动的，但如果将数据提供给用户去编辑，那么通常也会希望结果集是可滚动的。

如果要获得可更新的结果集，应该使用以下方法创建一条语句：

```
Statement stat = conn.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);
```

这样，调用 `executeQuery` 方法返回的结果集就将是可更新的结果集。

注释：并非所有的查询都会返回可更新的结果集。如果查询涉及多个表的连接操作，那么它所产生的结果集将是不可更新的。如果查询只涉及一个表，或者在查询时是使用主键连接多个表的，那么它所产生的结果集将是可更新的结果集。可以调用 `ResultSet` 接口中的 `getConcurrency` 方法来确定结果集是否是可更新的。

例如，假设想提高某些图书的价格，但是在执行 `UPDATE` 语句时又没有一个简单的提价标准。此时，就可以根据任意设定的条件，迭代遍历所有的图书并更新它们的价格。

```
String query = "SELECT * FROM Books";
ResultSet rs = stat.executeQuery(query);
while (rs.next())
{
    if (...)

    {
        double increase = ...;
        double price = rs.getDouble("Price");
        rs.updateDouble("Price", price + increase);
        rs.updateRow(); // make sure to call updateRow after updating fields
    }
}
```

所有对应于 SQL 类型的数据类型都配有 updateXxx 方法，比如 updateDouble、updateString 等。与 getXxx 方法相同，在使用 updateXxx 方法时必须指定列的名称或序号，然后给该字段设置新的值。

注释：在使用第一个参数为列序号的 updateXxx 方法时，请注意这里的列序号指的是该列在结果集中的序号。它的值可以与数据库中的列序号不同。

updateXxx 方法改变的只是结果集中的行值，而非数据库中的值。当更新完行中的字段值后，必须调用 updateRow 方法，这个方法将当前行中的所有更新信息发送给数据库。如果没有调用 updateRow 方法就将游标移动到其他行上，那么对此行所做的所有更新都将被丢弃，而且永远也不会被传递给数据库。还可以调用 cancelRowUpdates 方法来取消对当前行的更新。

我们在前面的例子中已经介绍过如何修改一个现有的行。如果想在数据库中添加一条新的记录，首先需要使用 moveToInsertRow 方法将游标移动到特定的位置，我们称之为插入行 (insert row)。然后，调用 updateXxx 方法在插入行的位置上创建一个新的行。在上述操作全部完成之后，还需要调用 insertRow 方法将新建的行发送给数据库。完成插入操作后，再调用 moveToCurrentRow 方法将游标移回到调用 moveToInsertRow 方法之前的位置。下面是一段示例程序：

```
rs.moveToInsertRow();
rs.updateString("Title", title);
rs.updateString("ISBN", isbn);
rs.updateString("Publisher_Id", pubid);
rs.updateDouble("Price", price);
rs.insertRow();
rs.moveToCurrentRow();
```

请注意，你无法控制在结果集或数据库中添加新数据的位置。

对于在插入行中没有指定值的列，将被设置为 SQL 的 NULL。但是，如果这个列有 NOT NULL 约束，那么将会抛出异常，而这一行也无法插入。

最后需要说明的是，你可以使用以下方法删除游标所指的行。

```
rs.deleteRow();
```

deleteRow 方法会立即将该行从结果集和数据库中删除。

ResultSet 接口中的 updateRow、insertRow 和 deleteRow 方法的执行效果等同于 SQL 命令中的 UPDATE、INSERT 和 DELETE。不过，习惯于 Java 编程语言的程序员通常会觉得使用结果集来操控数据库要比使用 SQL 语句自然得多。

警告：如果不小心处理的话，就很有可能在使用可更新的结果集时编写出非常低效的代码。执行 UPDATE 语句，要比建立一个查询，然后一边遍历一边修改数据显得高效得多。对于用户能够任意修改数据的交互式程序来说，使用可更新的结果集是非常有意义的。但是相对于大多数通过程序进行修改的情况，使用 SQL 的 UPDATE 语句更合适一些。

注释: JDBC 2 对结果集做了进一步的改进, 例如, 如果数据被其他的并发数据库连接所修改, 那么它可以用最新的数据来更新结果集。JDBC 3 添加了另一种优化, 可以指定结果集在事务提交时的行为。但是, 这些高级特性超出了本章的范围。我们推荐你参考 Maydene Fisher、Jon Ellis 和 Jonathan Bruce 所著的 *JDBC API Tutorial and Reference, Third Edition* (Addison-Wesley 出版社 2003 年出版) 和 JDBC 规范, 以了解更多的信息。

API **java.sql.Connection 1.1**

- `Statement createStatement(int type, int concurrency)` 1.2
- `PreparedStatement prepareStatement(String command, int type, int concurrency)` 1.2

创建一个语句或预备语句, 且该语句可以产生指定类型和并发模式的结果集。`type` 参数是 `ResultSet` 接口中的下述常量之一: `TYPE_FORWARD_ONLY`、`TYPE_SCROLL_INSENSITIVE` 或者 `TYPE_SCROLL_SENSITIVE`, `concurrency` 参数是 `ResultSet` 接口中的下述常量之一: `CONCUR_READ_ONLY` 或者 `CONCUR_UPDATABLE`

API **java.sql.ResultSet 1.1**

- `int getType()` 1.2
返回结果集的类型。返回值为以下常量之一: `TYPE_FORWARD_ONLY`、`TYPE_SCROLL_INSENSITIVE` 或 `TYPE_SCROLL_SENSITIVE`。
- `int getConcurrency()` 1.2
返回结果集的并发设置。返回值为以下常量之一: `CONCUR_READ_ONLY` 或 `CONCUR_UPDATABLE`
- `boolean previous()` 1.2
将游标移动到前一行。如果游标位于某一行上, 则返回 `true`; 如果游标位于第一行之前的位置, 则返回 `false`。
- `int getRow()` 1.2
得到当前行的序号。所有行从 1 开始编号。
- `boolean absolute(int r)` 1.2
移动游标到第 `r` 行。如果游标位于某一行上, 则返回 `true`。
- `boolean relative(int d)` 1.2
将游标移动 `d` 行。如果 `d` 为负数, 则游标向后移动。如果游标位于某一行上, 则返回 `true`。
- `boolean first()` 1.2
- `boolean last()` 1.2
移动游标到第一行或最后一行。如果游标位于某一行上, 则返回 `true`。
- `void beforeFirst()` 1.2
- `void afterLast()` 1.2
移动游标到第一行之前或最后一行之后的位置。

- `boolean isFirst() 1.2`
- `boolean isLast() 1.2`
测试游标是否在第一行或最后一行。
- `boolean isBeforeFirst() 1.2`
- `boolean isAfterLast() 1.2`
测试游标是否在第一行之前或最后一行之后的位置。
- `void moveToInsertRow() 1.2`
移动游标到插入行。插入行是一个特殊的行，可以在该行上使用 `updateXxx` 和 `insertRow` 方法来插入新数据。
- `void moveToCurrentRow() 1.2`
将游标从插入行移回到调用 `moveToInsertRow` 方法之前它所在的那一行。
- `void insertRow() 1.2`
将插入行上的内容插入到数据库和结果集中。
- `void deleteRow() 1.2`
从数据库和结果集中删除当前行。
- `void updateXxx(int column, Xxx data) 1.2`
- `void updateXxx(String columnName, Xxx data) 1.2`
(`Xxx` 指数据类型，比如 `int`、`double`、`String`、`Date` 等) 更新结果集中当前行上的某个字段值。
- `void updateRow() 1.2`
将当前行的更新信息发送到数据库。
- `void cancelRowUpdates() 1.2`
撤销对当前行的更新。

API `java.sql.DatabaseMetaData 1.1`

- `boolean supportsResultSetType(int type) 1.2`
如果数据库支持给定类型的结果集，则返回 `true`。`type` 是 `ResultSet` 接口中的常量之一：`TYPE_FORWARD_ONLY`、`TYPE_SCROLL_INSENSITIVE` 或者 `TYPE_SCROLL_SENSITIVE`。
- `boolean supportsResultSetConcurrency(int type, int concurrency) 1.2`
如果数据库支持给定类型和并发模式的结果集，则返回 `true`。`type` 参数是 `ResultSet` 接口中的下述常量之一：`TYPE_FORWARD_ONLY`、`TYPE_SCROLL_INSENSITIVE` 或者 `TYPE_SCROLL_SENSITIVE`。`concurrency` 是 `ResultSet` 接口中的下述常量之一：`CONCUR_READ_ONLY` 或者 `CONCUR_UPDATABLE`。

5.7 行集

可滚动的结果集虽然功能强大，却有一个重要的缺陷：在与用户的整个交互过程中，必

须始终与数据库保持连接。用户也许会离开电脑旁很长一段时间，而在此期间却始终占有着数据库连接。这种方式存在很大的问题，因为数据库连接属于稀有资源。在这种情况下，我们可以使用行集。`RowSet` 接口扩展自 `ResultSet` 接口，却无须始终保持与数据库的连接。

行集还适用于将查询结果移动到复杂应用的其他层，或者是诸如手机之类的其他设备中。你可能永远都不会考虑移动一个结果集，因为它的数据结构可能非常庞大，且依赖于数据库连接。

5.7.1 构建行集

以下为 `javax.sql.rowset` 包提供的接口，它们都扩展了 `RowSet` 接口：

- `CachedRowSet` 允许在断开连接的状态下执行相关操作。关于被缓存的行集我们将在下一节中讨论。
- `WebRowSet` 对象代表了一个被缓存的行集，该行集可以保存为 XML 文件。该文件可以移动到 Web 应用的其他层中，只要在该层中使用另一个 `WebRowSet` 对象重新打开该文件即可。
- `FilteredRowSet` 和 `JoinRowSet` 接口支持对行集的轻量级操作，它们等同于 SQL 中的 `SELECT` 和 `JOIN` 操作。这两个接口的操作对象是存储在行集中的数据，因此运行时无须建立数据库连接。
- `JdbcRowSet` 是 `ResultSet` 接口的一个瘦包装器。它在 `RowSet` 接口中添加了一些有用的方法。

在 Java 7 中，有一种获取行集的标准方式：

```
RowSetFactory factory = RowSetProvider.newFactory();
CachedRowSet crs = factory.createCachedRowSet();
```

获取其他行集类型的对象也有类似的方法。

5.7.2 被缓存的行集

一个被缓存的行集中包含了一个结果集中所有的数据。`CachedRowSet` 是 `ResultSet` 接口的子接口，所以你完全可以像使用结果集一样来使用被缓存的行集。被缓存的行集有一个非常重要的优点：断开数据库连接后仍然可以使用行集。你将在程序清单 5-4 的示例程序中看到，这种做法大大简化了交互式应用的实现。在执行每个用户命令时，我们只需打开数据库连接、执行查询操作、将查询结果放入被缓存的行集，然后关闭数据库连接即可。

我们甚至可以修改被缓存的行集中的数据。当然，这些修改不会立即反馈到数据库中。相反，必须发起一个显式的请求，以便让数据库真正接受所有修改。此时 `CachedRowSet` 类会重新连接到数据库，并通过执行 SQL 语句向数据库中写入所有修改后的数据。

可以使用一个结果集来填充 `CachedRowSet` 对象：

```
ResultSet result = . . .;
RowSetFactory factory = RowSetProvider.newFactory();
CachedRowSet crs = factory.createCachedRowSet();
crs.populate(result);
conn.close(); // now OK to close the database connection
```

或者，也可以让 `CachedRowSet` 对象自动建立一个数据库连接。首先，设置数据库参数：

```
crs.setURL("jdbc:derby://localhost:1527/COREJAVA");
crs.setUsername("dbuser");
crs.setPassword("secret");
```

然后，设置查询语句和所有参数。

```
crs.setCommand("SELECT * FROM Books WHERE Publisher_ID = ?");
crs.setString(1, publisherId);
```

最后，将查询结果填充到行集中：

```
crs.execute();
```

这个方法调用会建立数据库连接、执行查询操作、填充行集，最后断开连接。

如果查询结果非常大，那我们肯定不想将其全部放入行集中。毕竟，用户可能只是想浏览其中的几行而已。在这种情况下，可以指定每一页的尺寸：

```
CachedRowSet crs = . . .;
crs.setCommand(command);
crs.setPageSize(20);
. . .
crs.execute();
```

现在就能只获得 20 行了。要获取下一批数据，可以调用：

```
crs.nextPage();
```

可以使用与结果集中相同的方法来查看和修改行集中的数据。如果修改了行集中的内容，那么必须调用以下方法将修改写回到数据库中：

```
crs.acceptChanges(conn);
```

或

```
crs.acceptChanges();
```

只有在行集中设置了连接数据库所需的信息（如 URL、用户名和密码）时，上述第二个方法调用才会有效。

在 5.6.2 节中，我们曾经介绍过，并非所有的结果集都是可更新的。同样，如果一个行集包含的是复杂查询的查询结果，那么我们就无法将对该行集数据的修改写回到数据库中。不过，如果行集上的数据都来自同一张数据库表，我们就可以安全地写回数据。

! **警告：**如果是使用结果集来填充行集，那么行集就无从获知需要更新数据的数据库表名。此时，必须调用 `setTable` 方法来设置表名称。

另一个导致问题复杂化的情况是：在填充了行集之后，数据库中的数据发生了改变，这显然容易产生数据不一致性。为了解决这个问题，参考实现会首先检查行集中的原始值（即，修改前的值）是否与数据库中的当前值一致。如果一致，那么修改后的值将覆盖数据库中的当前值。否则，将抛出 `SyncProviderException` 异常，且不向数据库写回任何值。在实现行集接口时其他实现也可以采用不同的同步策略。

API `javax.sql.RowSet 1.4`

- `String getURL()`
获取或设置数据库的 URL。
- `void setUsername(String username)`
获取或设置连接数据库所需的用户名。
- `void setPassword(String password)`
获取或设置连接数据库所需的密码。
- `String getCommand()`
获取或设置向行集中填充数据时需要执行的命令。
- `void execute()`
通过执行使用 `setCommand` 方法设置的语句集来填充行集。为了使驱动管理器可以获得连接，必须事先设定 URL、用户名和密码。

API `javax.sql.rowset.CachedRowSet 5.0`

- `void execute(Connection conn)`
通过执行使用 `setCommand` 方法设置的语句集来填充行集。该方法使用给定的连接，并负责关闭它。
- `void populate(ResultSet result)`
将指定的结果集中的数据填充到被缓存的行集中。
- `String getTableName()`
获取或设置数据库表名称，填充被缓存的行集时所需的数据来自该表。
- `void setTableName(String tableName)`
获取和设置页的尺寸。
- `int getPageSize()`
加载下一页或上一页，如果要加载的页存在，则返回 `true`。
- `void acceptChanges()`
重新连接数据库，并写回行集中修改过的数据。如果因为数据库中的数据已经被修改而导致无法写回行集中的数据，该方法可能会抛出 `SyncProviderException` 异常。

API **javax.sql.rowset.RowSetProvider** 7

- static RowSetFactory newFactory()

创建一个行集工厂。

API **javax.sql.rowset.RowSetFactory** 7

- CachedRowSet createCachedRowSet()
- FilteredRowSet createFilteredRowSet()
- JdbcRowSet createJdbcRowSet()
- JoinRowSet createJoinRowSet()
- WebRowSet createWebRowSet()

创建一个指定类型的行集。

5.8 元数据

在前几节中，我们介绍了如何填充、查询和更新数据库表。其实，JDBC 还可以提供关于数据库及其表结构的详细信息。例如，可以获取某个数据库的所有表的列表，也可以获得某个表中所有列的名称及其数据类型。如果是在开发业务应用时使用事先定义好的数据库，那么数据库结构和表信息就不是非常有用了。毕竟，在设计数据库表时，就已经知道了它们的结构。但是，对于那些编写数据库工具的程序员来说，数据库的结构信息却是极其有用的。

在 SQL 中，描述数据库或其组成部分的数据称为元数据（区别于那些存在数据库中的实际数据）。我们可以获得三类元数据：关于数据库的元数据、关于结果集的元数据以及关于预备语句参数的元数据。

如果要了解数据库的更多信息，可以从数据库连接中获取一个 DatabaseMetaData 对象。

```
DatabaseMetaData meta = conn.getMetaData();
```

现在就可以获取某些元数据了。例如，调用

```
ResultSet mrs = meta.getTables(null, null, null, new String[] { "TABLE" });
```

将返回一个包含所有数据库表信息的结果集（如果要了解该方法的其他参数，请参见本节末尾的 API 说明）。

该结果集中的每一行都包含了数据库中一张表的详细信息，其中，第三列是表的名称。（同样，如果要了解其他列的信息，请参阅 API 说明。）下面的循环可以获取所有的表名：

```
while (mrs.next())
    tableNames.addItem(mrs.getString(3));
```

数据库元数据还有第二个重要应用。数据库是非常复杂的，SQL 标准为数据库的多样性提供了很大的空间。DatabaseMetaData 接口中有上百个方法可以用于查询数据库的相关信息，包括一些使用奇特的名字进行调用的方法，如：

```
meta.supportsCatalogsInPrivilegeDefinitions()
```

和

```
meta.nullPlusNonNullIsNull()
```

显然，这些方法主要是针对有特殊要求的高级用户的，尤其是那些需要编写涉及多个数据库且具有高可移植性的代码的编程人员。

`DatabaseMetaData` 接口用于提供有关数据库的数据，第二个元数据接口 `ResultSetMetaData` 则用于提供结果集的相关信息。每当通过查询得到一个结果集时，我们都可以获取该结果集的列数以及每一列的名称、类型和字段宽度。下面是一个典型的循环：

```
ResultSet rs = stat.executeQuery("SELECT * FROM " + tableName);
ResultSetMetaData meta = rs.getMetaData();
for (int i = 1; i <= meta.getColumnCount(); i++)
{
    String columnName = meta.getColumnLabel(i);
    int columnWidth = meta getColumnDisplaySize(i);
    ...
}
```

在这一节中，我们将介绍如何编写一个简单的数据库工具，程序清单 5-4 中的程序通过使用元数据来浏览数据库中的所有表，该程序还展示了如何使用缓存的行集。

程序清单 5-4 view/ViewDB.java

```
1 package view;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.io.*;
6 import java.nio.file.*;
7 import java.sql.*;
8 import java.util.*;
9
10 import javax.sql.*;
11 import javax.sql.rowset.*;
12 import javax.swing.*;
13
14 /**
15  * This program uses metadata to display arbitrary tables in a database.
16  * @version 1.34 2018-05-01
17  * @author Cay Horstmann
18  */
19 public class ViewDB
20 {
21     public static void main(String[] args)
22     {
23         EventQueue.invokeLater(() ->
24         {
25             var frame = new ViewDBFrame();
26             frame.setTitle("ViewDB");
27             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
28             frame.setVisible(true);
29         });
30     }
31 }
```

```
29         });
30     }
31 }
32 /**
33 * The frame that holds the data panel and the navigation buttons.
34 */
35 class ViewDBFrame extends JFrame
36 {
37     private JButton previousButton;
38     private JButton nextButton;
39     private JButton deleteButton;
40     private JButton saveButton;
41     private JPanel dataPanel;
42     private Component scrollPane;
43     private JComboBox<String> tableNames;
44     private Properties props;
45     private CachedRowSet crs;
46     private Connection conn;
47
48     public ViewDBFrame()
49     {
50         tableNames = new JComboBox<String>();
51
52         try
53         {
54             readDatabaseProperties();
55             conn = getConnection();
56             DatabaseMetaData meta = conn.getMetaData();
57             try (ResultSet mrs = meta.getTables(null, null, null, new String[] { "TABLE" }))
58             {
59                 while (mrs.next())
60                     tableNames.addItem(mrs.getString(3));
61             }
62         }
63     }
64     catch (SQLException ex)
65     {
66         for (Throwable t : ex)
67             t.printStackTrace();
68     }
69     catch (IOException ex)
70     {
71         ex.printStackTrace();
72     }
73
74     tableNames.addActionListener(
75         event -> showTable((String) tableNames.getSelectedItem(), conn));
76     add(tableNames, BorderLayout.NORTH);
77     addWindowListener(new WindowAdapter()
78     {
79         public void windowClosing(WindowEvent event)
80         {
81             try
82             {
```

```

83             if (conn != null) conn.close();
84         }
85     catch (SQLException ex)
86     {
87         for (Throwable t : ex)
88             t.printStackTrace();
89     }
90 }
91 );
92
93 var buttonPanel = new JPanel();
94 add(buttonPanel, BorderLayout.SOUTH);
95
96 previousButton = new JButton("Previous");
97 previousButton.addActionListener(event -> showPreviousRow());
98 buttonPanel.add(previousButton);
99
100 nextButton = new JButton("Next");
101 nextButton.addActionListener(event -> showNextRow());
102 buttonPanel.add(nextButton);
103
104 deleteButton = new JButton("Delete");
105 deleteButton.addActionListener(event -> deleteRow());
106 buttonPanel.add(deleteButton);
107
108 saveButton = new JButton("Save");
109 saveButton.addActionListener(event -> saveChanges());
110 buttonPanel.add(saveButton);
111 if (tableNames.getItemCount() > 0)
112     showTable(tableNames.getItemAt(0), conn);
113 }
114
115 /**
116 * Prepares the text fields for showing a new table, and shows the first row.
117 * @param tableName the name of the table to display
118 * @param conn the database connection
119 */
120 public void showTable(String tableName, Connection conn)
121 {
122     try (Statement stat = conn.createStatement();
123          ResultSet result = stat.executeQuery("SELECT * FROM " + tableName))
124     {
125         // get result set
126
127         // copy into cached row set
128         RowSetFactory factory = RowSetProvider.newFactory();
129         crs = factory.createCachedRowSet();
130         crs.setTableName(tableName);
131         crs.populate(result);
132
133         if (scrollPane != null) remove(scrollPane);
134         dataPanel = new DataPanel(crs);
135         scrollPane = new JScrollPane(dataPanel);
136         add(scrollPane, BorderLayout.CENTER);

```

```
137     pack();
138     showNextRow();
139 }
140 catch (SQLException ex)
141 {
142     for (Throwable t : ex)
143         t.printStackTrace();
144 }
145 }
146
147 /**
148 * Moves to the previous table row.
149 */
150 public void showPreviousRow()
151 {
152     try
153     {
154         if (crs == null || crs.isFirst()) return;
155         crs.previous();
156         dataPanel.showRow(crs);
157     }
158     catch (SQLException ex)
159     {
160         for (Throwable t : ex)
161             t.printStackTrace();
162     }
163 }
164
165 /**
166 * Moves to the next table row.
167 */
168 public void showNextRow()
169 {
170     try
171     {
172         if (crs == null || crs.isLast()) return;
173         crs.next();
174         dataPanel.showRow(crs);
175     }
176     catch (SQLException ex)
177     {
178         for (Throwable t : ex)
179             t.printStackTrace();
180     }
181 }
182
183 /**
184 * Deletes current table row.
185 */
186 public void deleteRow()
187 {
188     if (crs == null) return;
189     new SwingWorker<Void, Void>()
190 {
```

```

191     public Void doInBackground() throws SQLException
192     {
193         crs.deleteRow();
194         crs.acceptChanges(conn);
195         if (crs.isAfterLast())
196             if (!crs.last()) crs = null;
197         return null;
198     }
199     public void done()
200     {
201         dataPanel.showRow(crs);
202     }
203     }.execute();
204 }
205 /**
206 * Saves all changes.
207 */
208 public void saveChanges()
209 {
210     if (crs == null) return;
211     new SwingWorker<Void, Void>()
212     {
213         public Void doInBackground() throws SQLException
214         {
215             dataPanel.setRow(crs);
216             crs.acceptChanges(conn);
217             return null;
218         }
219         }.execute();
220     }
221
222 private void readDatabaseProperties() throws IOException
223 {
224     props = new Properties();
225     try (InputStream in = Files.newInputStream(Paths.get("database.properties")))
226     {
227         props.load(in);
228     }
229     String drivers = props.getProperty("jdbc.drivers");
230     if (drivers != null) System.setProperty("jdbc.drivers", drivers);
231 }
232
233 /**
234 * Gets a connection from the properties specified in the file database.properties.
235 * @return the database connection
236 */
237 private Connection getConnection() throws SQLException
238 {
239     String url = props.getProperty("jdbc.url");
240     String username = props.getProperty("jdbc.username");
241     String password = props.getProperty("jdbc.password");
242
243     return DriverManager.getConnection(url, username, password);
244 }

```

```
245 }
246
247 /**
248 * This panel displays the contents of a result set.
249 */
250 class DataPanel extends JPanel
251 {
252     private java.util.List<JTextField> fields;
253
254     /**
255      * Constructs the data panel.
256      * @param rs the result set whose contents this panel displays
257      */
258     public DataPanel(ResultSet rs) throws SQLException
259     {
260         fields = new ArrayList<>();
261         setLayout(new GridBagLayout());
262         var gbc = new GridBagConstraints();
263         gbc.gridwidth = 1;
264         gbc.gridheight = 1;
265
266         ResultSetMetaData rsmd = rs.getMetaData();
267         for (int i = 1; i <= rsmd.getColumnCount(); i++)
268         {
269             gbc.gridx = i - 1;
270
271             String columnName = rsmd.getColumnName(i);
272             gbc.gridx = 0;
273             gbc.anchor = GridBagConstraints.EAST;
274             add(new JLabel(columnName), gbc);
275
276             int columnWidth = rsmd getColumnDisplaySize(i);
277             var tb = new JTextField(columnWidth);
278             if (!rsmd.getColumnClassName(i).equals("java.lang.String"))
279                 tb.setEditable(false);
280
281             fields.add(tb);
282
283             gbc.gridx = 1;
284             gbc.anchor = GridBagConstraints.WEST;
285             add(tb, gbc);
286         }
287     }
288
289     /**
290      * Shows a database row by populating all text fields with the column values.
291      */
292     public void showRow(ResultSet rs)
293     {
294         try
295         {
296             if (rs == null) return;
297             for (int i = 1; i <= fields.size(); i++)
298             {
```

```

299         String field = rs == null ? "" : rs.getString(i);
300         JTextField tb = fields.get(i - 1);
301         tb.setText(field);
302     }
303 }
304 catch (SQLException ex)
305 {
306     for (Throwable t : ex)
307         t.printStackTrace();
308 }
309 }
310
311 /**
312 * Updates changed data into the current row of the row set.
313 */
314 public void setRow(ResultSet rs) throws SQLException
315 {
316     for (int i = 1; i <= fields.size(); i++)
317     {
318         String field = rs.getString(i);
319         JTextField tb = fields.get(i - 1);
320         if (!field.equals(tb.getText()))
321             rs.updateString(i, tb.getText());
322     }
323     rs.updateRow();
324 }
325 }

```

顶部的组合框用于显示数据库中的所有表。选中其中一个表，框中央就会显示出该表的所有字段名及其第一条记录的值，见图 5-6。点击 Next 和 Previous 按钮可以滚动遍历表中的所有记录，还可以删除一行或编辑行的值，点击 Save 按钮可以将各种修改保存到数据库中。

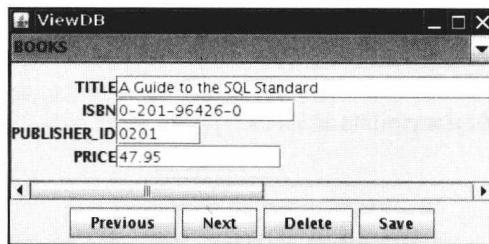


图 5-6 ViewDB 应用程序

注释：许多数据库都配有复杂得多的工具，用于查看和编辑数据库表。如果你使用的数据库没有这样的工具，那么可以求助于 DBeaver (<https://dbeaver.io>) 或者 SQuirreL (<http://squirrel-sql.sourceforge.net>)。这些程序可以查看任何 JDBC 数据库中的表。我们编写示例程序并非为了取代这些工具，而是为了向你演示如何编写工具来处理任意的数据库表。

API ***java.sql.Connection*** 1.1

- `DatabaseMetaData getMetaData()`

返回一个 `DatabaseMetaData` 对象，该对象封装了有关数据库连接的元数据。

API ***java.sql.DatabaseMetaData*** 1.1

- `ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String types[])`

返回某个目录（catalog）中所有表的描述，该目录必须匹配给定的模式（schema）、表名字模式以及类型标准。（模式用于描述一组相关的表和访问权限，而目录描述的是一组相关的模式，这些概念对组织大型数据库非常重要。）

`catalog` 和 `schema` 参数可以为 ""，用于检索那些没有目录或模式的表。如果不考虑目录和模式，也可以将上述参数设为 `null`。

`types` 数组包含了所需的表类型的名称，通常表类型有 TABLE、VIEW、SYSTEM TABLE、GLOBAL TEMPORARY、LOCAL TEMPORARY、ALIAS 和 SYNONYM。如果 `types` 为 `null`，则返回所有类型的表。

返回的结果集共有 5 列，均为 `String` 类型。

行	名称	解释
1	TABLE_CAT	表目录（可以为 <code>null</code> ）
2	TABLE_SCHEM	表模式（可以为 <code>null</code> ）
3	TABLE_NAME	表名称
4	TABLE_TYPE	表类型
5	REMARKS	关于表的注释

- `int getJDBCMajorVersion() 1.4`
- `int getJDBCMinorVersion() 1.4`

返回建立数据库连接的 JDBC 驱动程序的主版本号和次版本号。例如，一个 JDBC 3.0 的驱动程序有一个主版本号 3 和一个次版本号 0。

- `int getMaxConnections()`

返回可同时连接到数据库的最大并发连接数。

- `int getMaxStatements()`

返回单个数据库连接允许同时打开的最大并发语句数。如果对允许打开的语句数目没有限制或者不可知，则返回 0。

API ***java.sql.ResultSet*** 1.1

- `ResultSetMetaData getMetaData()`

返回与当前 `ResultSet` 对象中的列相关的元数据。

API ***java.sql.ResultSetMetaData*** 1.1

- `int getColumnCount()`

返回当前 `ResultSet` 对象中的列数。

- `int getColumnDisplaySize(int column)`

返回给定列序号的列的最大宽度。

- `String getColumnLabel(int column)`

返回该列所建议的名称。

- `String getColumnName(int column)`

返回指定的列序号所对应的列名。

5.9 事务

我们可以将一组语句构建成一个事务 (transaction)。当所有语句都顺利执行之后，事务可以被提交 (commit)。否则，如果其中某个语句遇到错误，那么事务将被回滚，就好像没有任何语句被执行过一样。

将多个语句组合成事务的主要原因是为了确保数据库完整性 (database integrity)。例如，假设我们需要将钱从一个银行账号转账到另一个账号。此时，一个非常重要的问题就是我们必须同时将钱从一个账号取出并且存入另一个账号。如果在将钱存入其他账号之前系统发生崩溃，那么我们必须撤销取款操作。

如果将更新语句组合成一个事务，那么事务要么成功地执行所有操作并提交，要么在中间某个位置发生失败。在这种情况下，可以执行回滚 (rollback) 操作，则数据库将自动撤销自上次提交事务以来的所有更新操作产生的影响。

5.9.1 用 JDBC 对事务编程

默认情况下，数据库连接处于自动提交模式 (autocommit mode)，即每个 SQL 语句一旦被执行便被提交给数据库。一旦命令被提交，就无法对它进行回滚操作。在使用事务时，需要关闭这个默认值：

```
conn.setAutoCommit(false);
```

现在可以按照通常的方式创建一个语句对象：

```
Statement stat = conn.createStatement();
```

然后任意多次地调用 `executeUpdate` 方法：

```
stat.executeUpdate(command1);
stat.executeUpdate(command2);
stat.executeUpdate(command3);
...
```

如果执行了所有命令之后没有出错，则调用 `commit` 方法：

```
conn.commit();
```

如果出现错误，则调用：

```
conn.rollback();
```

此时，程序将自动撤销自上次提交以来的所有语句。当事务被 `SQLException` 异常中断时，典型的方法就是发起回滚操作。

5.9.2 保存点

在使用某些驱动程序时，使用保存点（`save point`）可以更细粒度地控制回滚操作。创建一个保存点意味着稍后只需返回到这个点，而非放弃整个事务。例如，

```
Statement stat = conn.createStatement(); // start transaction; rollback() goes here
stat.executeUpdate(command1);
Savepoint svpt = conn.setSavepoint(); // set savepoint; rollback(svpt) goes here
stat.executeUpdate(command2);

if (...) conn.rollback(svpt); // undo effect of command2
...
conn.commit();
```

当不再需要保存点时，应该释放它：

```
conn.releaseSavepoint(svpt);
```

5.9.3 批量更新

假设有一个程序需要执行许多 `INSERT` 语句，以便将数据填入数据库表中，此时可以使用批量更新的方法来提高程序性能。在使用批量更新（`batch update`）时，一个语句序列作为一批操作将同时被收集和提交。

注释： 使用 `DatabaseMetaData` 接口中的 `supportsBatchUpdates` 方法可以获知数据库是否支持这种特性。

处于同一批中的语句可以是 `INSERT`、`UPDATE` 和 `DELETE` 等操作，也可以是数据库定义语句，如 `CREATE TABLE` 和 `DROP TABLE`。但是，在批量处理中添加 `SELECT` 语句会抛出异常（从概念上讲，批量处理中的 `SELECT` 语句没有意义，因为它会返回结果集，而并不更新数据库）。

为了执行批量处理，首先必须使用通常的办法创建一个 `Statement` 对象：

```
Statement stat = conn.createStatement();
```

现在，应该调用 `addBatch` 方法，而非 `executeUpdate` 方法：

```
String command = "CREATE TABLE . . ."
stat.addBatch(command);

while (. . .)
{
    command = "INSERT INTO . . . VALUES (" + . . . + ")";
    stat.addBatch(command);
}
```

最后，提交整个批量更新语句：

```
int[] counts = stat.executeBatch();
```

调用 `executeBatch` 方法将为所有已提交的语句返回一个记录数的数组。

为了在批量模式下正确地处理错误，必须将批量执行的操作视为单个事务。如果批量更新在执行过程中失败，那么必须将它回滚到批量操作开始之前的状态。

首先，关闭自动提交模式，然后收集批量操作，执行并提交该操作，最后恢复最初的自动提交模式：

```
boolean autoCommit = conn.getAutoCommit();
conn.setAutoCommit(false);
Statement stat = conn.createStatement();

...
// keep calling stat.addBatch(. . .);

...
stat.executeBatch();
conn.commit();
conn.setAutoCommit(autoCommit);
```

API `java.sql.Connection 1.1`

- `boolean getAutoCommit()`
- `void setAutoCommit(boolean b)`

获取该连接中的自动提交模式，或将其设置为 `b`。如果自动更新为 `true`，那么所有语句将在执行结束后立刻被提交。

- `void commit()`
- 提交自上次提交以来所有执行过的语句。

- `void rollback()`
- 撤销自上次提交以来所有执行过的语句所产生的影响。

- `Savepoint setSavepoint() 1.4`
- `Savepoint setSavepoint(String name) 1.4`

设置一个匿名或具名的保存点。

- `void rollback(Savepoint svpt) 1.4`
- 回滚到给定保存点。
- `void releaseSavepoint(Savepoint svpt) 1.4`
- 释放给定的保存点。

API `java.sql.Savepoint 1.4`

- `int getSavepointId()`

获取该匿名保存点的 ID 号。如果该保存点具有名字，则抛出一个 `SQLException` 异常。

- `String getSavepointName()`

获取该保存点的名称。如果该对象为匿名保存点，则抛出一个 `SQLException` 异常。

API `java.sql.Statement 1.1`

- `void addBatch(String command) 1.2`

添加命令到该语句当前的批量命令中。

- `int[] executeBatch()` 1.2
- `long[] executeLargeBatch()` 8

执行当前批量更新中的所有命令。返回一个记录数的数组，其中每一个元素都对应一条语句，如果其值非负，则表示受该语句影响的行数；如果其值为 `SUCCESS_NO_INFO`，则表示该语句成功执行了，但没有提供任何行数；如果其值为 `EXECUTE_FAILED`，则表示该语句执行失败了。

API `java.sql.DatabaseMetaData` 1.1

- `boolean supportsBatchUpdates()` 1.2

如果驱动程序支持批量更新，则返回 `true`。

5.9.4 高级 SQL 类型

表 5-8 列举了 JDBC 支持的 SQL 数据类型以及它们在 Java 语言中对应的数据类型。

表 5-8 SQL 数据类型及其对应的 Java 类型

SQL 数据类型	Java 数据类型
INTEGER 或 INT	<code>int</code>
SMALLINT	<code>short</code>
NUMERIC(m,n), DECIMAL(m,n) 或 DEC(m,n)	<code>java.math.BigDecimal</code>
FLOAT(n)	<code>double</code>
REAL	<code>float</code>
DOUBLE	<code>double</code>
CHARACTER(n) 或 CHAR(n)	<code>String</code>
VARCHAR(n), LONG VARCHAR	<code>String</code>
BOOLEAN	<code>boolean</code>
DATE	<code>java.sql.Date</code>
TIME	<code>java.sql.Time</code>
TIMESTAMP	<code>java.sql.Timestamp</code>
BLOB	<code>java.sql.Blob</code>
CLOB	<code>java.sql.Clob</code>
ARRAY	<code>java.sql.Array</code>
ROWID	<code>java.sql.RowId</code>
NCHAR(n), NVARCHAR(n), LONG NVARCHAR	<code>String</code>
NCLOB	<code>java.sql.NClob</code>
SQLXML	<code>java.sql.SQLXML</code>

SQL ARRAY (SQL 数组) 指的是值的序列。例如，`Student` 表中通常都会有一个 `Scores` 列，这个列就应该是 `ARRAY OF INTEGER` (整数数组)。`getArray` 方法返回一个接口类型为 `java.sql.Array` 的对象，该接口中有许多方法可以用来获取数据值。

从数据库中获得一个 LOB 或数组并不等于获取了它的实际内容，只有在访问具体的值时它们才会从数据库中被读取出来。这对改善性能非常有好处，因为通常这些数据的数据量都非常大。

某些数据库支持描述行位置的 `ROWID` 值，这样就可以非常快捷地获取某一行值。JDBC 4 引入了 `java.sql.RowId` 接口，并提供了用于在查询中提供行 ID，以及从结果中获取该值的方法。

国家属性字符串（`NCHAR` 及其变体）按照本地字符编码机制存储字符串，并使用本地排序惯例对这些字符串进行排序。JDBC 4 提供了方法，用于在查询和结果中进行 Java 的 `String` 对象和国家属性字符串之间的双向转换。

有些数据库可以存储用户自定义的结构化类型。JDBC 3 提供了一种机制用于将 SQL 结构化类型自动映射成 Java 对象。

有些数据库提供用于 XML 数据的本地存储。JDBC 4 引入了 `SQLXML` 接口，它可以在内部的 XML 表示和 DOM 的 `Source/Result` 接口或二进制流之间起到中介作用。请查看 `SQLXML` 类的 API 文档以了解详细信息。

我们不再更深入地讨论这些高级 SQL 类型了，你可以在 *JDBC API Tutorial and Reference* 和 JDBC 4 的规范中找到更多有关这些主题的信息。

5.10 Web 与企业应用中的连接管理

我们在前面几节中曾经介绍过，使用 `database.properties` 文件可以对数据库连接进行非常简单的设置。这种方法适用于小型的测试程序，但是不适用于规模较大的应用。

在 Web 或企业环境中部署 JDBC 应用时，数据库连接管理与 Java 名字和目录接口（JNDI）是集成在一起的。遍布企业的数据源的属性可以存储在一个目录中，采用这种方式使得我们可以集中管理用户名、密码、数据库名和 JDBC URL。

在这样的环境中，可以使用下列代码创建数据库连接：

```
var jndiContext = new InitialContext();
var source = (DataSource) jndiContext.lookup("java:comp/env/jdbc/corejava");
Connection conn = source.getConnection();
```

请注意，我们不再使用 `DriverManager`，而是使用 JNDI 服务来定位数据源。数据源是一个能够提供简单的 JDBC 连接和更多高级服务的接口，比如执行涉及多个数据库的分布式事务。`javax.sql` 标准扩展包定义了 `DataSource` 接口。

注释：在 Java EE 的容器中，甚至不必编程进行 JNDI 查找，只需在 `DataSource` 域上使用 `Resource` 注解，当加载应用时，这个数据源引用将被设置：

```
@Resource(name="jdbc/corejava")
private DataSource source;
```

当然，我们必须在某个地方配置数据源。如果你编写的数据库程序将在 Servlet 容器中运

行，比如 Apache Tomcat，或在应用服务器中运行，比如 GlassFish，那么必须将数据库配置信息（包括 JNDI 名字、JDBC URL、用户名和密码）放置在配置文件中，或者在管理员 GUI 中进行设置。

用户名管理和登录管理只是众多需要特别关注的问题之一。另一个重要问题则涉及建立数据库连接所需的开销。我们的示例数据库程序使用了两种策略来获取数据库连接：程序清单 5-3 中的 `QueryDB` 程序在程序的开头建立了到数据库的单个连接，并在程序结尾处关闭了它，而程序清单 5-4 中的 `ViewDB` 程序在每次需要时都会打开一个新连接。

但是，这两种方式都不令人满意：因为数据库连接是有限的资源，如果用户要离开应用一段时间，那么他占用的连接就不应该保持打开状态；另一方面，每次查询都获取连接并在随后关闭它的代价也是相当高的。

解决上述问题的方法是建立数据库连接池（pool）。这意味着数据库连接在物理上并未被关闭，而是保留在一个队列中并被反复重用。连接池是一种非常重要的服务，JDBC 规范为实现者提供了用以实现连接池服务的手段。不过，JDK 本身并未实现这项服务，数据库供应商提供的 JDBC 驱动程序中通常也不包含这项服务。相反，Web 容器和应用服务器的开发商通常会提供连接池服务的实现。

连接池的使用对程序员来说是完全透明的，可以通过获取数据源并调用 `getConnection` 方法来得到连接池中的连接。使用完连接后，需要调用 `close` 方法。该方法并不会在物理上关闭连接，而只是告诉连接池已经使用完该连接。连接池通常还会将池机制作用于预备语句上。

至此，你已经学会了 JDBC 的基本知识，并且已经知道如何实现简单的数据库应用。然而，正如我们在本章的开头所强调的那样，数据库的相关技术非常复杂；本章属于介绍性章节，相当多的高级话题已经超出了本章的范围。如果要全面了解 JDBC 的高级功能，请参阅 *JDBC API Tutorial and Reference* 或 JDBC 规范。

在本章中，我们学习了如何用 Java 操作关系型数据库。下一章将讨论 Java 8 的日期和时间库。

第6章 日期和时间 API

- ▲ 时间线
- ▲ 本地日期
- ▲ 日期调整器
- ▲ 本地时间

- ▲ 时区时间
- ▲ 格式化和解析
- ▲ 与遗留代码的互操作

光阴似箭，我们可以很容易地设置一个起点，然后向前和向后以秒来计时。那为什么处理时间还会显得如此之难呢？问题出在人类自身上。如果我们只需告诉对方：“1523793600 时来见我，别迟到！”那么一切都会很简单。但是我们希望时间能够和朝夕与季节挂钩，这就使事情变得复杂了。Java 1.0 有一个 `Date` 类，事后证明它过于简单了，当 Java 1.1 引入 `Calendar` 类之后，`Date` 类中的大部分方法就被弃用了。但是，`Calendar` 的 API 还不够给力，它的实例是可修改的，并且它没有处理诸如闰秒这样的问题。第 3 次升级很吸引人，那就是 Java SE 8 中引入的 `java.time` API，它修正了过去的缺陷，并且应该会服役相当长的一段时间。在本章中，你将会学习是什么使时间计算变得如此烦人，以及日期和时间 API 是如何解决这些问题的。

6.1 时间线

在历史上，基本的时间单位“秒”是从地球的自转中推导出来的。地球自转一周需要 24 个小时，即 $24 \times 60 \times 60 = 86\,400$ 秒，因此，看起来这好像只是一个有关如何精确定义 1 秒的天文度量问题。遗憾的是，地球有轻微的摄动，所以需要更加精确的定义。1967 年，人们根据铯 133 原子内在的特性推导出了与历史定义相匹配的秒的新的精确定义。自那以后，一直由一个原子钟网络来维护着官方时间。

官方时间的维护器时常需要将绝对时间与地球自转进行同步。首先，官方的秒需要稍作调整，从 1972 年开始，偶尔需要插入“闰秒”。（在理论上，偶尔也需要移除 1 秒，但是这还从来没发生过。）这又是有关修改系统时间的话题。很明显，闰秒是个痛点，许多计算机系统使用“平滑”方式来人为地在紧邻闰秒之前让时间变慢或变快，以保证每天都是 86.400 秒。这种做法可以奏效，因为计算机上的本地时间并非那么精确，而计算机也惯于将自身时间与外部的时间服务进行同步。

Java 的 Date 和 Time API 规范要求 Java 使用的时间尺度为：

- 每天 86 400 秒
- 每天正午与官方时间精确匹配
- 在其他时间点上，以精确定义的方式与官方时间接近匹配

这赋予了 Java 很大的灵活性，使其可以进行调整，以适应官方时间未来的变化。

在 Java 中，`Instant` 表示时间线上的某个点。被称为“新纪元”的时间线原点被设置为穿过伦敦格林尼治皇家天文台的本初子午线所处时区的 1970 年 1 月 1 日的午夜。这与 UNIX/POSIX 时间中使用的惯例相同。从该原点开始，时间按照每天 86 400 秒向前或向回度量，精确到纳秒。`Instant` 的值往回可追溯 10 亿年 (`Instant.MIN`)。这对于表示宇宙年龄（大约 135 亿年）来说还差得远，但是对于所有实际应用来说，应该足够了。毕竟，10 亿年前，地球表面还覆盖着冰层，只有当今植物和动物的微生物祖先在繁殖生衍。最大的值 `Instant.MAX` 是公元 1 000 000 000 年的 12 月 31 日。

静态方法调用 `Instant.now()` 会给出当前的时刻。你可以按照常用的方式，用 `equals` 和 `compareTo` 方法来比较两个 `Instant` 对象，因此你可以将 `Instant` 对象用作时间戳。

为了得到两个时刻之间的时间差，可以使用静态方法 `Duration.between`。例如，下面的代码展示了如何度量算法的运行时间：

```
Instant start = Instant.now();
runAlgorithm();
Instant end = Instant.now();
Duration timeElapsed = Duration.between(start, end);
long millis = timeElapsed.toMillis();
```

`Duration` 是两个时刻之间的时间量。你可以通过调用 `toNanos`、`toMillis`、`getSeconds`、`toMinutes`、`toHours` 和 `toDays` 来获得 `Duration` 按照传统单位度量的时间长度。

注释：在 Java 8 中，必须调用 `getSeconds` 而不是 `toSeconds`。

如果想要让计算精确到纳秒级，那么就需要当心上溢问题。`long` 值可以存储大约 300 年时间对应的纳秒数。如果你需要的 `Duration` 短于这个时间，那么可以直接将其转换为纳秒数。你可以使用更长的 `Duration`，即让 `Duration` 对象用一个 `long` 来存储秒数，用另外一个 `int` 来存储纳秒数。`Duration` 接口包含了大量在本节末尾展示的用于执行算术运算的方法。

例如，如果想要检查某个算法是否至少比另一个算法快 10 倍，那么你可以执行如下的计算：

```
Duration timeElapsed2 = Duration.between(start2, end2);
boolean overTenTimesFaster
    = timeElapsed.multipliedBy(10).minus(timeElapsed2).isNegative();
```

这里只展示了语法。因为算法不会运行数百年，所以可以直接使用下面的方法：

```
boolean overTenTimesFaster = timeElapsed.toNanos() * 10 < timeElapsed2.toNanos();
```

注释：`Instant` 和 `Duration` 类都是不可修改的类，所以诸如 `multipliedBy` 和 `minus` 这样的方法都会返回一个新的实例。

在程序清单 6-1 的示例程序中，可以看到如何使用 `Instant` 和 `Duration` 类来对两个算法计时。

程序清单 6-1 timeline/TimeLine.java

```
1 package timeline;
2
3 /**
```

```

4  * @version 1.0 2016-05-10
5  * @author Cay Horstmann
6  */
7
8 import java.time.*;
9 import java.util.*;
10 import java.util.stream.*;
11
12 public class Timeline
13 {
14     public static void main(String[] args)
15     {
16         Instant start = Instant.now();
17         runAlgorithm();
18         Instant end = Instant.now();
19         Duration timeElapsed = Duration.between(start, end);
20         long millis = timeElapsed.toMillis();
21         System.out.printf("%d milliseconds\n", millis);
22
23         Instant start2 = Instant.now();
24         runAlgorithm2();
25         Instant end2 = Instant.now();
26         Duration timeElapsed2 = Duration.between(start2, end2);
27         System.out.printf("%d milliseconds\n", timeElapsed2.toMillis());
28         boolean overTenTimesFaster = timeElapsed.multipliedBy(10)
29             .minus(timeElapsed2).isNegative();
30         System.out.printf("The first algorithm is %smore than ten times faster",
31             overTenTimesFaster ? "" : "not ");
32     }
33
34     public static void runAlgorithm()
35     {
36         int size = 10;
37         List<Integer> list = new Random().ints().map(i -> i % 100).limit(size)
38             .boxed().collect(Collectors.toList());
39         Collections.sort(list);
40         System.out.println(list);
41     }
42
43     public static void runAlgorithm2()
44     {
45         int size = 10;
46         List<Integer> list = new Random().ints().map(i -> i % 100).limit(size)
47             .boxed().collect(Collectors.toList());
48         while (!IntStream.range(1, list.size())
49             .allMatch(i -> list.get(i - 1).compareTo(list.get(i)) <= 0))
50             Collections.shuffle(list);
51         System.out.println(list);
52     }
53 }

```

API `java.time.Instant` 8

- `static Instant now()`

从最佳的可用系统时钟中获取当前的时刻。

- Instant plus(TemporalAmount amountToAdd)
 - Instant minus(TemporalAmount amountToSubtract)
- 产生一个时刻，该时刻与当前时刻距离给定的时间量。Duration 和 Period (参阅 6.2 节) 实现了 TemporalAmount 接口。
- Instant (plus|minus)(Nanos|Millis|Seconds)(long number)
- 产生一个时刻，该时刻与当前时刻距离给定数量的纳秒、微秒或秒。

API `java.time.Duration` 8

- static Duration of(Nanos|Millis|Seconds|Minutes|Hours|Days)(long number)
产生一个给定数量的指定时间单位的时间间隔。
 - static Duration between(Temporal startInclusive, Temporal endExclusive)
产生一个在给定时间点之间的 Duration 对象。Instant 类实现了 Temporal 接口，LocalDate/LocalDateTime/LocalTime (参阅 6.4 节) 和 ZonedDateTime (参阅 6.5 节) 也实现了该接口。
 - long toNanos()
● long toMillis()
● long toSeconds() 9
● long toMinutes()
● long toHours()
● long toSeconds()
● long toDays()
获取当前时长按照方法名中的时间单位度量的数量。
 - int to(Nanos|Millis|Seconds|Minutes|Hours)Part() 9
● long to(Days|Hours|Minutes|Seconds|Millis|Nanos)Part() 9
当前时长中给定时间单位的部分。例如，在 100 秒的时间间隔中，分钟的部分是 1，秒的部分是 40。
 - Instant plus(TemporalAmount amountToAdd)
● Instant minus(TemporalAmount amountToSubtract)
- 产生一个时刻，该时刻与当前时刻距离给定的时间量。Duration 和 Period 类 (参阅 6.2 节) 实现了 TemporalAmount 接口。
- Duration multipliedBy(long multiplicand)
 - Duration dividedBy(long divisor)
 - Duration negated()
- 产生一个时长，该时长是通过当前时刻乘以或除以给定的量或 -1 得到的。
- boolean isZero()

- `boolean isNegative()`

如果当前 Duration 对象是 0 或负数，则返回 true。

- `Duration plus|minus(Nanos|Millis|Seconds|Minutes|Hours|Days)(long number)`

产生一个时长，该时长是通过当前时刻加上或减去给定的数量的指定时间单位而得到的。

6.2 本地日期

现在，让我们从绝对时间转向人类时间。在 Java API 中有两种人类时间，本地日期 / 时间和时区时间。本地日期 / 时间包含日期和当天的时间，但是与时区信息没有任何关联。1903 年 6 月 14 日就是一个本地日期的示例（lambda 演算的发明者 Alonzo Church 在这一天诞生）。因为这个日期既没有当天的时间，也没有时区信息，因此它并不对应精确的时刻。与之相反的是，1969 年 7 月 16 日 09:32:00 EDT（阿波罗 11 号发射的时刻）是一个时区日期 / 时间，表示的是时间线上的一个精确的时刻。

有许多计算并不需要时区，在某些情况下，时区甚至是一种障碍。假设你安排每周 10:00 开一次会。如果你加 7 天（即 $7 \times 24 \times 60 \times 60$ 秒）到最后一次会议的时区时间上，那么你可能会碰巧跨越了夏令时的时间调整边界，这次会议可能会早一小时或晚一小时！

正是考虑到这个原因，API 的设计者们推荐程序员不要使用时区时间，除非确实想要表示绝对时间的实例。生日、假日、计划时间等通常最好都表示成本地日期和时间。

`LocalDate` 是带有年、月、日的日期。为了构建 `LocalDate` 对象，可以使用 `now` 或 `of` 静态方法：

```
LocalDate today = LocalDate.now(); // Today's date
LocalDate alonzosBirthday = LocalDate.of(1903, 6, 14);
alonzoBirthday = LocalDate.of(1903, Month.JUNE, 14);
// Uses the Month enumeration
```

与 UNIX 和 `java.util.Date` 中使用的月从 0 开始计算而年从 1900 开始计算的不规则的惯用法不同，你需要提供通常使用的月份的数字。或者，你可以使用 `Month` 枚举。

本节末尾展示了最有用的操作 `LocalDate` 对象的方法。

例如，程序员日是每年的第 256 天。下面展示了如何很容易地计算出它：

```
LocalDate programmersDay = LocalDate.of(2014, 1, 1).plusDays(255);
// September 13, but in a leap year it would be September 12
```

回忆一下，两个 `Instant` 之间的时长是 `Duration`，而用于本地日期的等价物是 `Period`，它表示的是流逝的年、月或日的数量。可以调用 `birthday.plus(Period.ofYears(1))` 来获取下一年的生日。当然，也可以直接调用 `birthday.plusYears(1)`。但是 `birthday.plus(Duration.ofDays(365))` 在闰年是不会产生正确结果的。

`util` 方法会产生两个本地日期之间的时长。例如，

```
independenceDay.until(christmas)
```

会产生 5 个月 21 天的一段时长。这实际上并不是很有用，因为每个月的天数不尽相同。为

了确定到底有多少天，可以使用：

```
independenceDay.until(christmas, ChronoUnit.DAYS) // 174 days
```

! 警告：`LocalDate` API 中的有些方法可能会创建出并不存在的日期。例如，在 1 月 31 日上加上 1 个月不应该产生 2 月 31 日。这些方法并不会抛出异常，而是会返回该月有效的最后一天。例如，

```
LocalDate.of(2016, 1, 31).plusMonths(1)
```

和

```
LocalDate.of(2016, 3, 31).minusMonths(1)
```

都将产生 2016 年 2 月 29 日。

`getDayOfWeek` 会产生星期日期，即 `DayOfWeek` 枚举的某个值。`DayOfWeek.MONDAY` 的枚举值为 1，而 `DayOfWeek.SUNDAY` 的枚举值为 7。例如，

```
LocalDate.of(1900, 1, 1).getDayOfWeek().getValue()
```

会产生 1。`DayOfWeek` 枚举具有便捷方法 `plus` 和 `minus`，以 7 为模计算星期日期。例如，`DayOfWeek.SATURDAY.plus(3)` 会产生 `DayOfWeek.TUESDAY`。

! 注释：周末实际上在每周的末尾。这与 `java.util.Calendar` 有所差异，在后者中，星期日的值为 1，而星期六的值为 7。

Java 9 添加了两个有用的 `datesUntil` 方法，它们会产生 `LocalDate` 对象流。

```
LocalDate start = LocalDate.of(2000, 1, 1);
LocalDate endExclusive = LocalDate.now();
Stream<LocalDate> allDays = start.datesUntil(endExclusive);
Stream<LocalDate> firstDaysInMonth = start.datesUntil(endExclusive, Period.ofMonths(1));
```

除了 `LocalDate` 之外，还有 `MonthDay`、`YearMonth` 和 `Year` 类可以描述部分日期。例如，12 月 25 日（没有指定年份）可以表示成一个 `MonthDay` 对象。

程序清单 6-2 中的示例程序展示了如何使用 `LocalDate` 类。

程序清单 6-2 localdates/LocalDates.java

```
1 package localdates;
2
3 /**
4  * @version 1.0 2016-05-10
5  * @author Cay Horstmann
6 */
7 import java.time.*;
8 import java.time.temporal.*;
9 import java.util.stream.*;
10
11 public class LocalDates
12 {
13     public static void main(String[] args)
```

```

14  {
15      LocalDate today = LocalDate.now(); // Today's date
16      System.out.println("today: " + today);
17
18      LocalDate alonzosBirthday = LocalDate.of(1903, 6, 14);
19      alonzosBirthday = LocalDate.of(1903, Month.JUNE, 14);
20      // Uses the Month enumeration
21      System.out.println("alonzosBirthday: " + alonzosBirthday);
22
23      LocalDate programmersDay = LocalDate.of(2018, 1, 1).plusDays(255);
24      // September 13, but in a leap year it would be September 12
25      System.out.println("programmersDay: " + programmersDay);
26
27      LocalDate independenceDay = LocalDate.of(2018, Month.JULY, 4);
28      LocalDate christmas = LocalDate.of(2018, Month.DECEMBER, 25);
29
30      System.out.println("Until christmas: " + independenceDay.until(christmas));
31      System.out.println("Until christmas: "
32          + independenceDay.until(christmas, ChronoUnit.DAYS));
33
34      System.out.println(LocalDate.of(2016, 1, 31).plusMonths(1));
35      System.out.println(LocalDate.of(2016, 3, 31).minusMonths(1));
36
37      DayOfWeek startOfLastMillennium = LocalDate.of(1900, 1, 1).getDayOfWeek();
38      System.out.println("startOfLastMillennium: " + startOfLastMillennium);
39      System.out.println(startOfLastMillennium.getValue());
40      System.out.println(DayOfWeek.SATURDAY.plus(3));
41
42      LocalDate start = LocalDate.of(2000, 1, 1);
43      LocalDate endExclusive = LocalDate.now();
44      Stream<LocalDate> firstDaysInMonth = start.datesUntil(endExclusive, Period.ofMonths(1));
45      System.out.println("firstDaysInMonth: "
46          + firstDaysInMonth.collect(Collectors.toList()));
47  }
48 }

```

API `java.time.LocalDate` 8

- `static LocalDate now()`
获取当前的 `LocalDate`。
- `static LocalDate of(int year, int month, int dayOfMonth)`
- `static LocalDate of(int year, Month month, int dayOfMonth)`
用给定的年、月（1 到 12 之间的整数或者 `Month` 枚举的值）和日（1 到 31 之间）产生一个本地日期。
- `LocalDate (plus|minus)(Days|Weeks|Months|Years)(long number)`
产生一个 `LocalDate`，该对象是通过在当前对象上加上或减去给定数量的时间单位获得的。
- `LocalDate plus(TemporalAmount amountToAdd)`
- `LocalDate minus(TemporalAmount amountToSubtract)`
产生一个时刻，该时刻与当前时刻距离给定的时间量。`Duration` 和 `Period` 类实现了

TemporalAmount 接口。

- `LocalDate withDayOfMonth(int dayOfMonth)`
- `LocalDate withDayOfYear(int dayOfYear)`
- `LocalDate withMonth(int month)`
- `LocalDate withYear(int year)`

返回一个新的 LocalDate，将月份日期、年日期、月或年修改为给定值。

- `int getDayOfMonth()`

获取月份日期（1 到 31 之间）。

- `int getDayOfYear()`

获取年日期（1 到 366 之间）。

- `DayOfWeek getDayOfWeek()`

获取星期日期，返回某个 DayOfWeek 枚举值。

- `Month getMonth()`

- `int getMonthValue()`

获取用 Month 枚举值表示的月份，或者用 1 到 12 之间的数字表示的月份。

- `int getYear()`

获取年份，在 -999,999,999 到 999,999,999 之间。

- `Period until(ChronoLocalDate endDateExclusive)`

获取直到给定终止日期的 period。LocalDate 和 date 类针对非公历实现了 ChronoLocalDate 接口。

- `boolean isBefore(ChronoLocalDate other)`

- `boolean isAfter(ChronoLocalDate other)`

如果该日期在给定日期之前或之后，则返回 true。

- `boolean isLeapYear()`

如果当前是闰年，则返回 true。即，该年份能够被 4 整除，但是不能被 100 整除，或者能够被 400 整除。该算法应该可以应用于所有已经过去的年份，尽管在历史上它并不准确（闰年是在公元前 46 年发明出来的，而涉及整除 100 和 400 的规则是在 1582 年的公历改革中引入的。这场改革经历了 300 年才被广泛接受）。

- `Stream<LocalDate> datesUntil(LocalDate endExclusive) 9`

- `Stream<LocalDate> datesUntil(LocalDate endExclusive, Period step) 9`

产生一个日期流，从当前的 LocalDate 对象直至参数 endExclusive 指定的日期，其中步长尺寸为 1，或者是给定的 period。

API `java.time.Period` 8

- `static Period of(int years, int months, int days)`
- `Period of(Days|Weeks|Months|Years)(int number)`

用给定数量的时间单位产生一个 `Period` 对象。

- `int get(Days | Months | Years)()`

获取当前 `Period` 对象的日、月或年。

- `Period (plus | minus)(Days | Months | Years)(long number)`

产生一个 `LocalDate`, 该对象是通过在当前对象上加上或减去给定数量的时间单位获得的。

- `Period plus(TemporalAmount amountToAdd)`

- `Period minus(TemporalAmount amountToSubtract)`

产生一个时刻, 该时刻与当前时刻距离给定的时间量。`Duration` 和 `Period` 类实现了 `TemporalAmount` 接口。

- `Period with(Days | Months | Years)(int number)`

返回一个新的 `Period`, 将日、月、年修改为给定值。

6.3 日期调整器

对于日程安排应用来说, 经常需要计算诸如“每个月的第一个星期二”这样的日期。`TemporalAdjusters` 类提供了大量用于常见调整的静态方法。你可以将调整方法的结果传递给 `with` 方法。例如, 某个月的第一个星期二可以像下面这样计算:

```
LocalDate firstTuesday = LocalDate.of(year, month, 1).with(
    TemporalAdjusters.nextOrSame(DayOfWeek.TUESDAY));
```

一如既往, `with` 方法会返回一个新的 `LocalDate` 对象, 而不会修改原来的对象。本节末尾展示了有关可用的调整器的 API 说明。

还可以通过实现 `TemporalAdjuster` 接口来创建自己的调整器。下面是用于计算下一个工作日的调整器。

```
TemporalAdjuster NEXT_WORKDAY = w ->
{
    var result = (LocalDate) w;
    do
    {
        result = result.plusDays(1);
    }
    while (result.getDayOfWeek().getValue() >= 6);
    return result;
};
```

```
LocalDate backToWork = today.with(NEXT_WORKDAY);
```

注意, `lambda` 表达式的参数类型为 `Temporal`, 它必须被强制转型为 `LocalDate`。你可以用 `ofDateAdjuster` 方法来避免这种强制转型, 该方法期望得到的参数是类型为 `UnaryOperator<LocalDate>` 的 `lambda` 表达式。

```
TemporalAdjuster NEXT_WORKDAY = TemporalAdjusters.ofDateAdjuster(w ->
{
    LocalDate result = w; // No cast
```

```

do
{
    result = result.plusDays(1);
}
while (result.getDayOfWeek().getValue() >= 6);
return result;
});

```

API **java.time.LocalDate 9**

- `LocalDate with(TemporalAdjuster adjuster)`

返回该日期通过给定的调整器调整后的结果。

API **java.time.temporal.TemporalAdjusters 9**

- `static TemporalAdjuster next(DayOfWeek dayOfWeek)`
- `static TemporalAdjuster nextOrSame(DayOfWeek dayOfWeek)`
- `static TemporalAdjuster previous(DayOfWeek dayOfWeek)`
- `static TemporalAdjuster previousOrSame(DayOfWeek dayOfWeek)`

返回一个调整器，用于将日期调整为给定的星期日期。

- `static TemporalAdjuster dayOfWeekInMonth(int n, DayOfWeek dayOfWeek)`
- `static TemporalAdjuster lastInMonth(DayOfWeek dayOfWeek)`

返回一个调整器，用于将日期调整为月份中第 n 个或最后一个给定的星期日期。

- `static TemporalAdjuster firstDayOfMonth()`
- `static TemporalAdjuster firstDayOfNextMonth()`
- `static TemporalAdjuster firstDayOfYear()`
- `static TemporalAdjuster firstDayOfNextYear()`
- `static TemporalAdjuster lastDayOfMonth()`
- `static TemporalAdjuster lastDayOfYear()`

返回一个调整器，用于将日期调整为月份或年份中给定的日期。

6.4 本地时间

`LocalTime` 表示当日时刻，例如 15:30:00。可以用 `now` 或 `of` 方法创建其实例：

```

LocalTime rightNow = LocalTime.now();
LocalTime bedtime = LocalTime.of(22, 30); // or LocalTime.of(22, 30, 0)

```

API 说明展示了常见的对本地时间的操作。`plus` 和 `minus` 操作是按照一天 24 小时循环操作的。例如，

```
LocalTime wakeup = bedtime.plusHours(8); // wakeup is 6:30:00
```

 **注释：**`LocalTime` 自身并不关心 AM/PM。这种愚蠢的设计将问题抛给格式器去解决，请参见 6.6 节。

还有一个表示日期和时间的 `LocalDateTime` 类。这个类适合存储固定时区的时间点，例如，用于排课或排程。但是，如果你的计算需要跨越夏令时，或者需要处理不同时区的用户，那么就应该使用接下来要讨论的 `ZonedDateTime` 类。

API `java.time.LocalTime` 8

- `static LocalTime now()`
获取当前的 `LocalTime`。
- `static LocalTime of(int hour, int minute)`
- `static LocalTime of(int hour, int minute, int second)`
- `static LocalTime of(int hour, int minute, int second, int nanoOfSecond)`
产生一个 `LocalTime`，它具有给定的小时（0 到 23 之间）、分钟、秒（0 到 59 之间）和纳秒（0 到 999,999,999 之间）。
- `LocalTime (plus|minus)(Hours|Minutes|Seconds|Nanos)(long number)`
产生一个 `LocalTime`，该对象是通过在当前对象上加上或减去给定数量的时间单位获得的。
- `LocalTime plus(TemporalAmount amountToAdd)`
- `LocalTime minus(TemporalAmount amountToSubtract)`
产生一个时刻，该时刻与当前时刻距离给定的时间量。
- `LocalTime with(Hour|Minute|Second|Nano)(int value)`
返回一个新的 `LocalTime`，将小时、分钟、秒或纳秒修改为给定值。
- `int getHour()`
获取小时（0 到 23 之间）。
- `int getMinute()`
- `int getSecond()`
获取分钟或秒（0 到 59 之间）。
- `int getNano()`
获取纳秒（0 到 999,999,999 之间）。
- `int toSecondOfDay()`
- `long toNanoOfDay()`
产生自午夜到当前 `LocalTime` 的秒或纳秒数。
- `boolean isBefore(LocalTime other)`
- `boolean isAfter(LocalTime other)`
如果当期日期在给定日期之前或之后，则返回 `true`。

6.5 时区时间

时区问题比较复杂。在理性的世界中，我们都会遵循格林尼治时间，有些人在 02:00

吃午饭，而有些人却在 22:00 吃午饭。中国横跨了 4 个时区，但是使用了同一个时间。在其他地方，时区显得并不规则，并且还有国际日期变更线，而夏令时则使事情变得更复杂了。

尽管时区显得变化繁多，但这就是无法回避的现实生活。在实现日历应用时，它需要能够为坐飞机在不同国家之间穿梭的人们提供服务。如果你有个 10:00 在纽约召开的电话会议，但是碰巧你人在柏林，那么你肯定希望该应用能够在正确的本地时间点上发出提醒。

互联网编码分配管理机构（Internet Assigned Numbers Authority, IANA）保存着一个数据库，里面存储着世界上所有已知的时区 (www.iana.org/time-zones)，它每年会更新数次，而批量更新会处理夏令时的变更规则。Java 使用了 IANA 数据库。

每个时区都有一个 ID，例如 America/New_York 和 Europe/Berlin。要想找出所有可用的时区，可以调用 ZoneId.getAvailableZoneIds。在本书撰写之时，有将近 600 个 ID。

给定一个时区 ID，静态方法 ZoneId.of(id) 可以产生一个 ZoneId 对象。可以通过调用 local.atZone(zoneId) 用这个对象将 LocalDateTime 对象转换为 ZonedDateTime 对象，或者可以通过调用静态方法 ZonedDateTime.of(year, month, day, hour, minute, second, nano, zoneId) 来构造一个 ZonedDateTime 对象。例如，

```
ZonedDateTime apollo11launch = ZonedDateTime.of(1969, 7, 16, 9, 32, 0, 0,
    ZoneId.of("America/New_York"));
// 1969-07-16T09:32:00[America/New_York]
```

这是一个具体的时刻，调用 apollo11launch.toInstant 可以获得对应的 Instant 对象。反过来，如果你有一个时刻对象，调用 instant.atZone(ZoneId.of("UTC")) 可以获得格林尼治皇家天文台的 ZonedDateTime 对象，或者使用其他的 ZoneId 获得地球上其他地方的 ZoneId。

 **注释：** UTC 代表“协调世界时”，这是英文“Coordinated Universal Time”和法文“Temps Universel Coordonné”首字母缩写的折中，它与这两种语言中的缩写都不一致。UTC 是不考虑夏令时的格林尼治皇家天文台时间。

ZonedDateTime 的许多方法都与 LocalDateTime 的方法相同（参见本节末尾的 API 说明），它们大多数都很直观，但是夏令时带来了一些复杂性。

当夏令时开始时，时钟要向前拨快一小时。当你构建的时间对象正好落入了这跳过去的一个小时内时，会发生什么？例如，在 2013 年，中欧地区在 3 月 31 日 2:00 切换到夏令时，如果你试图构建的时间是不存在的 3 月 31 日 2:30，那么你实际上得到的是 3:30。

```
ZonedDateTime skipped = ZonedDateTime.of(
    LocalDate.of(2013, 3, 31),
    LocalTime.of(2, 30),
    ZoneId.of("Europe/Berlin"));
// Constructs March 31 3:30
```

反过来，当夏令时结束时，时钟要向回拨慢一小时，这样同一个本地时间就会有出现两次。当你构建位于这个时间段内的时间对象时，就会得到这两个时刻中较早的一个：

```
ZonedDateTime ambiguous = ZonedDateTime.of(
    LocalDate.of(2013, 10, 27), // End of daylight savings time
    LocalTime.of(2, 30),
    ZoneId.of("Europe/Berlin"));
// 2013-10-27T02:30+02:00[Europe/Berlin]
ZonedDateTime anHourLater = ambiguous.plusHours(1);
// 2013-10-27T02:30+01:00[Europe/Berlin]
```

一个小时后的时间会具有相同的小时和分钟，但是时区的偏移量会发生变化。

你还需要在调整跨越夏令时边界的日期时特别注意。例如，如果你将会议设置在下个星期，不要直接加上一个 7 天的 Duration：

```
ZonedDateTime nextMeeting = meeting.plus(Duration.ofDays(7));
// Caution! Won't work with daylight savings time
```

而是应该使用 Period 类。

```
ZonedDateTime nextMeeting = meeting.plus(Period.ofDays(7)); // OK
```

！ 警告：还有一个 OffsetDateTime 类，它表示与 UTC 具有偏移量的时间，但是没有时区规则的束缚。这个类被设计用于专用应用，这些应用特别需要剔除这些规则的约束，例如某些网络协议。对于人类时间，还是应该使用 ZonedDateTime。

程序清单 6-3 中的示例程序演示了 ZonedDateTime 类的用法。

程序清单 6-3 zonedtimes/ZonedDateTime.java

```
1 package zonedtimes;
2
3 /**
4  * @version 1.0 2016-05-10
5  * @author Cay Horstmann
6 */
7
8 import java.time.*;
9
10 public class ZonedDateTime
11 {
12     public static void main(String[] args)
13     {
14         ZonedDateTime apollo11Launch = ZonedDateTime.of(1969, 7, 16, 9, 32, 0, 0,
15             ZoneId.of("America/New_York")); // 1969-07-16T09:32-04:00[America/New_York]
16         System.out.println("apollo11Launch: " + apollo11Launch);
17
18         Instant instant = apollo11Launch.toInstant();
19         System.out.println("instant: " + instant);
20
21         ZonedDateTime zonedDateTime = instant.atZone(ZoneId.of("UTC"));
22         System.out.println("zonedDateTime: " + zonedDateTime);
23
24         ZonedDateTime skipped = ZonedDateTime.of(LocalDate.of(2013, 3, 31),
25             LocalTime.of(2, 30), ZoneId.of("Europe/Berlin")); // Constructs March 31 3:30
26         System.out.println("skipped: " + skipped);
```

```

27
28     ZonedDateTime ambiguous = ZonedDateTime.of(
29         LocalDate.of(2013, 10, 27), // End of daylight savings time
30         LocalTime.of(2, 30), ZoneId.of("Europe/Berlin"));
31         // 2013-10-27T02:30+02:00[Europe/Berlin]
32     ZonedDateTime anHourLater = ambiguous.plusHours(1);
33         // 2013-10-27T02:30+01:00[Europe/Berlin]
34     System.out.println("ambiguous: " + ambiguous);
35     System.out.println("anHourLater: " + anHourLater);
36
37
38     ZonedDateTime meeting = ZonedDateTime.of(LocalDate.of(2013, 10, 31),
39         LocalTime.of(14, 30), ZoneId.of("America/Los_Angeles"));
40     System.out.println("meeting: " + meeting);
41     ZonedDateTime nextMeeting = meeting.plus(Duration.ofDays(7));
42         // Caution! Won't work with daylight savings time
43     System.out.println("nextMeeting: " + nextMeeting);
44     nextMeeting = meeting.plus(Period.ofDays(7)); // OK
45     System.out.println("nextMeeting: " + nextMeeting);
46 }

```

API `java.time.ZonedDateTime` 8

- `static ZonedDateTime now()`
获取当前的 `ZonedDateTime`。
- `static ZonedDateTime of(int year, int month, int dayOfMonth, int hour, int minute, int second, int nanoOfSecond, ZoneId zone)`
- `static ZonedDateTime of(LocalDate date, LocalTime time, ZoneId zone)`
- `static ZonedDateTime of(LocalDateTime localDateTime, ZoneId zone)`
- `static ZonedDateTime ofInstant(Instant instant, ZoneId zone)`
用给定的参数和时区产生一个 `ZonedDateTime`。
- `ZonedDateTime (plus|minus)(Days|Weeks|Months|Years|Hours|Minutes|Seconds|Nanos)(long number)`
产生一个 `ZonedDateTime`, 该对象是通过在当前对象上加上或减去给定数量的时间单位获得的。
- `ZonedDateTime plus(TemporalAmount amountToAdd)`
- `ZonedDateTime minus(TemporalAmount amountToSubtract)`
产生一个时刻, 该时刻与当前时刻距离给定的时间量。
- `ZonedDateTime with(DayOfMonth|DayOfYear|Month|Year|Hour|Minute|Second|Nano)(int value)`
返回一个新的 `ZonedDateTime`, 用给定的值替换给定的时间单位。
- `ZonedDateTime withZoneSameInstant(ZoneId zone)`
- `ZonedDateTime withZoneSameLocal(ZoneId zone)`
返回一个新的 `ZonedDateTime`, 位于给定的时区, 它与当前对象要么表示相同的时刻, 要么表示相同的本地时间。

- `int getDayOfMonth()`
获取月份日期（1 到 31 之间）。
- `int getDayOfYear()`
获取年份日期（1 到 366 之间）。
- `DayOfWeek getDayOfWeek()`
获取星期日期，返回 `DayOfWeek` 枚举的值。
- `Month getMonth()`
- `int getMonthValue()`
获取用 `Month` 枚举值表示的月份，或者用 1 到 12 之间的数字表示的月份。
- `int getYear()`
获取年份，在 -999 999 999 到 999 999999 之间。
- `int getHour()`
获取小时（0 到 23 之间）。
- `int getMinute()`
- `int getSecond()`
获取分钟到秒（0 到 59 之间）。
- `int getNano()`
获取纳秒（0 到 999 999 999 之间）。
- `public ZoneOffset getOffset()`
获取与 UTC 的时间差距。差距可在 -12:00 ~ +14:00 变化。有些时区还有小数时间差。时间差会随着夏令时变化。
- `LocalDate toLocalDate()`
- `LocalTime toLocalTime()`
- `LocalDateTime toLocalDateTime()`
- `Instant toInstant()`
生成当地日期、时间，或日期 / 时间，或相应的瞬间。
- `boolean isBefore(ChronoZonedDateTime other)`
- `boolean isAfter(ChronoZonedDateTime other)`
如果这个时区日期 / 时间在给定的时区日期 / 时间之前或之后，则返回 `true`。

6.6 格式化和解析

`DateTimeFormatter` 类提供了三种用于打印日期 / 时间值的格式器：

- 预定义的格式器（参见表 6-1）
- `locale` 相关的格式器
- 带有定制模式的格式器

表 6-1 预定义的格式器

格式器	描述	示例
BASIC_ISO_DATE	年、月、日、时区偏移量，中间没有分隔符	19690716-0500
ISO_LOCAL_DATE, ISO_LOCAL_TIME, ISO_LOCAL_DATE_TIME	分隔符为 -、:、T	1969-07-16, 09:32:00, 1969-07-16T09:32:00
ISO_OFFSET_DATE, ISO_OFFSET_TIME, ISO_OFFSET_DATE_TIME	类似 ISO_LOCAL_XXX，但是有时区偏移量	1969-07-16-05:00, 09:32:00-05:00, 1969-07-16T09:32:00-05:00
ISO_ZONED_DATE_TIME	有时区偏移量和时区 ID	1969-07-16T09:32:00-05:00[America/New_York]
ISO_INSTANT	在 UTC 中，用 Z 时区 ID 来表示	1969-07-16T14:32:00Z
ISO_DATE, ISO_TIME, ISO_DATE_TIME	类似 ISO_OFFSET_DATE、ISO_OFFSET_TIME 和 ISO_ZONED_DATE_TIME，但是时区信息是可选的	1969-07-16-05:00, 09:32:00-05:00, 1969-07-16T09:32:00-05:00[America/New_York]
ISO_ORDINAL_DATE	LocalDate 的年和年日期	1969-197
ISO_WEEK_DATE	LocalDate 的年、星期和星期日期	1969-W29-3
RFC_1123_DATE_TIME	用于邮件时间戳的标准，编纂于 RFC822，并在 RFC1123 中将年份更新到 4 位	Wed, 16 Jul 1969 09:32:00 -0500

要使用标准的格式器，可以直接调用其 `format` 方法：

```
String formatted = DateTimeFormatter.ISO_OFFSET_DATE_TIME.format(apollo11launch);
// 1969-07-16T09:32:00-04:00"
```

标准格式器主要是为了机器可读的时间戳而设计的。为了向人类读者表示日期和时间，可以使用 `locale` 相关的格式器。对于日期和时间而言，有 4 种与 `locale` 相关的格式化风格，即 `SHORT`、`MEDIUM`、`LONG` 和 `FULL`，参见表 6-2。

表 6-2 locale 相关的格式化风格

风格	日期	时间
SHORT	7/16/69	9:32 AM
MEDIUM	Jul 16, 1969	9:32:00 AM
LONG	July 16, 1969	9:32:00 AM EDT
FULL	Wednesday, July 16, 1969	9:32:00 AM EDT

静态方法 `ofLocalizedDate`、`ofLocalizedTime` 和 `ofLocalizedDateTime` 可以创建这种格式器。例如：

```
DateTimeFormatter formatter = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.LONG);
String formatted = formatter.format(apollo11launch);
// July 16, 1969 9:32:00 AM EDT
```

这些方法使用了默认的 `locale`。为了切换到不同的 `locale`，可以直接使用 `withLocale` 方法。

```
formatted = formatter.withLocale(Locale.FRENCH).format(apollo11launch);
// 16 juillet 1969 09:32:00 EDT
```

`DayOfWeek` 和 `Month` 枚举都有 `getDisplayName` 方法，可以按照不同的 `locale` 和格式给出星期日期和月份的名字。

```

for (DayOfWeek w : DayOfWeek.values())
    System.out.print(w.getDisplayName(TextStyle.SHORT, Locale.ENGLISH) + " ");
// Prints Mon Tue Wed Thu Fri Sat Sun

```

请查看第 7 章以了解更多有关 locale 的信息。

 **注释：**java.time.format.DateTimeFormatter 类被设计用来替代 java.util.DateFormat。如果你为了向后兼容性而需要后者的实例，那么可以调用 formatter.toFormat()。

最后，可以通过指定模式来定制自己的日期格式。例如，

```
formatter = DateTimeFormatter.ofPattern("E yyyy-MM-dd HH:mm");
```

会将日期格式化为 Wed 1969-07-16 09:32 的形式。按照人们日积月累而制定的显得有些晦涩的规则，每个字母都表示一个不同的时间域，而字母重复的次数对应于所选择的特定格式。表 6-3 展示了最有用的模式元素。

表 6-3 常用的日期 / 时间格式的格式化符号

时间域或目的	示例
ERA	G: AD, GGGG: Anno Domini, GGGGG: A
YEAR_OF_ERA	yy: 69, yyyy: 1969
MONTH_OF_YEAR	M: 7, MM: 07, MMMM: Jul, MMMMM: July, MMMMMM: J
DAY_OF_MONTH	d: 6, dd: 06
DAY_OF_WEEK	e: 3, E: Wed, EEEE: Wednesday, EEEEE: W
HOUR_OF_DAY	H: 9, HH: 09
CLOCK_HOUR_OF_AM_PM	K: 9, KK: 09
AMPM_OF_DAY	a: AM
MINUTE_OF_HOUR	mm: 02
SECOND_OF_MINUTE	ss: 00
NANO_OF_SECOND	nnnnnn: 000000
时区 ID	VV: America/New_York
时区名	z: EDT, zzzz: Eastern Daylight Time V:ET, VVV:Eastern time
时区偏移量	x: -04, xx: -0400, xxx: -04:00, XXX: 与 xxx 相同，但是 Z 表示 0
本地化的时区偏移量	O: GMT-4, OOOO: GMT-04:00
修改后的儒略日	g:58243

为了解析字符串中的日期 / 时间值，可以使用众多的静态 parse 方法之一。例如，

```

LocalDate churchsBirthday = LocalDate.parse("1903-06-14");
ZonedDateTime apollo11Launch =
    ZonedDateTime.parse("1969-07-16 03:32:00-0400",
        DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ssxx"));

```

第一个调用使用了标准的 ISO_LOCAL_DATE 格式器，而第二个调用使用的是一个定制的格式器。

程序清单 6-4 中的程序展示了如何格式化和解析日期与时间。

程序清单 6-4 formatting/Formatting.java

```

1 package formatting;
2
3 /**
4  * @version 1.0 2016-05-10
5  * @author Cay Horstmann
6 */
7
8 import java.time.*;
9 import java.time.format.*;
10 import java.util.*;
11
12 public class Formatting
13 {
14     public static void main(String[] args)
15     {
16         ZonedDateTime apollo11launch = ZonedDateTime.of(1969, 7, 16, 9, 32, 0, 0,
17             ZoneId.of("America/New_York"));
18
19         String formatted = DateTimeFormatter.ISO_OFFSET_DATE_TIME.format(apollo11launch);
20         // 1969-07-16T09:32:00-04:00
21         System.out.println(formatted);
22
23         DateTimeFormatter formatter = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.LONG);
24         formatted = formatter.format(apollo11launch);
25         // July 16, 1969 9:32:00 AM EDT
26         System.out.println(formatted);
27         formatted = formatter.withLocale(Locale.FRENCH).format(apollo11launch);
28         // 16 juillet 1969 09:32:00 EDT
29         System.out.println(formatted);
30
31         formatter = DateTimeFormatter.ofPattern("E yyyy-MM-dd HH:mm");
32         formatted = formatter.format(apollo11launch);
33         System.out.println(formatted);
34
35         LocalDate churchsBirthday = LocalDate.parse("1903-06-14");
36         System.out.println("churchsBirthday: " + churchsBirthday);
37         apollo11launch = ZonedDateTime.parse("1969-07-16 03:32:00-0400",
38             DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ssxx"));
39         System.out.println("apollo11launch: " + apollo11launch);
40
41         for (DayOfWeek w : DayOfWeek.values())
42             System.out.print(w.getDisplayName(TextStyle.SHORT, Locale.ENGLISH) + " ");
43     }
44 }
```

API `java.time.format.DateTimeFormatter` 8

- `String format(TemporalAccessor temporal)`

格式化给定值。`Instant`、`LocalDate`、`LocalTime`、`LocalDateTime` 和 `ZonedDateTime`，以及许多其他类，都实现了 `TemporalAccessor` 接口。

- static DateTimeFormatter ofLocalizedDate(FormatStyle dateStyle)
- static DateTimeFormatter ofLocalizedTime(FormatStyle timeStyle)
- static DateTimeFormatter ofLocalizedDateTime(FormatStyle dateTimeStyle)
- static DateTimeFormatter ofLocalizedDateTime(FormatStyle dateStyle, FormatStyle timeStyle)
产生一个用于给定风格的格式器。FormatStyle 枚举的值包括 SHORT、MEDIUM、LONG 和 FULL。
- DateTimeFormatter withLocale(Locale locale)
用给定的地点产生一个等价于当前格式器的格式器。
- static DateTimeFormatter ofPattern(String pattern)
- static DateTimeFormatter ofPattern(String pattern, Locale locale)
用给定的模式和地点产生一个格式器。参阅表 6-3 有关模式的语法。

API **java.time.LocalDate 8**

- static LocalDate parse(CharSequence text)
- static LocalDate parse(CharSequence text, DateTimeFormatter formatter)
用默认的格式器或给定的格式器产生一个 LocalDate。

API **java.time.ZonedDateTime 8**

- static ZonedDateTime parse(CharSequence text)
- static ZonedDateTime parse(CharSequence text, DateTimeFormatter formatter)
用默认的格式器或给定的格式器产生一个 ZonedDateTime。

6.7 与遗留代码的互操作

作为全新的创造，Java Date 和 Time API 必须能够与已有类之间进行互操作，特别是无处不在的 `java.util.Date`、`java.util.GregorianCalendar` 和 `java.sql.Date/Time/Timestamp`。

`Instant` 类近似于 `java.util.Date`。在 Java 8 中，这个类有两个额外的方法：将 `Date` 转换为 `Instant` 的 `toInstant` 方法，以及反方向转换的静态的 `from` 方法。

类似地，`ZonedDateTime` 近似于 `java.util.GregorianCalendar`，在 Java 8 中，这个类有细粒度的转换方法。`toZonedDateTime` 方法可以将 `GregorianCalendar` 转换为 `ZonedDateTime`，而静态的 `from` 方法可以执行反方向的转换。

另一个可用于日期和时间类的转换集位于 `java.sql` 包中。你还可以传递一个 `DateTimeFormatter` 给使用 `java.text.Format` 的遗留代码。表 6-4 对这些转换进行了总结。

表 6-4 java.time 类与遗留类之间的转换

类	转换到遗留类	转换自遗留类
Instant ↔ java.util.Date	Date.from(instant)	date.toInstant()

(续)

类	转换到遗留类	转换自遗留类
ZonedDateTime ↔ java.util.GregorianCalendar	GregorianCalendar. from(zonedDateTime)	cal.toZonedDateTime()
Instant ↔ java.sql.Timestamp	TimeStamp.from(instant)	timestamp.toInstant()
LocalDateTime ↔ java.sql.Timestamp	Timestamp.valueOf(localDateTime)	timeStamp.toLocalDateTime()
LocalDate ↔ java.sql.Date	Date.valueOf(localDate)	date.toLocalDate()
LocalTime ↔ java.sql.Time	Time.valueOf(localTime)	time.toLocalTime()
DateTimeFormatter ↔ java.text.DateFormat	formatter.toFormat()	无
java.util.TimeZone ↔ ZoneId	Timezone.getTimeZone(id)	timeZone.toZoneId()
java.nio.file.attribute.FileTime ↔ Instant	FileTime.from(instant)	fileTime.toInstant()

你现在知道如何使用 Java 8 的日期和时间库来操作全世界的日期和时间值了。下一章将进一步讨论如何为国际受众编程。你将会看到如何以对客户而言有意义的方式来格式化程序的消息、数字和货币，无论这些客户身处世界的何处。

第7章 国际化

- ▲ locale
- ▲ 数字格式
- ▲ 日期和时间
- ▲ 排序和规范化

- ▲ 消息格式化
- ▲ 文本输入和输出
- ▲ 资源包
- ▲ 一个完整的例子

世界丰富多彩，我们希望大部分居民都能对你的软件感兴趣。一方面，因特网早已为我们打破了国家之间的界限。另一方面，如果你不去关注国际用户，你的产品的应用情况就会受到限制。

Java 编程语言是第一种设计成为全面支持国际化的语言。从一开始，它就具备了进行有效的国际化所必需的一个重要特性：使用 Unicode 来处理所有字符串。由于支持 Unicode，在 Java 编程语言中编写程序来操作多种语言的字符串变得异常方便。

多数程序员认为将程序进行国际化需要做的所有事情就是支持 Unicode 并在用户接口中对消息进行翻译。但是，在本章你将会看到，国际化一个程序所要做的事情绝不仅仅是提供 Unicode 支持。在世界的不同地方，日期、时间、货币甚至数字的格式都不相同。你需要用一种简单的方法来为不同的语言配置菜单与按钮的名字、消息字符串和快捷键。

在本章中，我们将演示如何编写国际化的 Java 应用程序以及如何将日期、时间、数字、文本和图形用户界面本地化，还将演示 Java 提供的编写国际化程序的工具。最后以一个完整的例子来作为本章的结束，它是一个退休金计算器，带有英语、德语和中文用户界面。

7.1 locale

当你看到一个面向国际市场的应用软件时，它与其他软件最明显的区别就是语言。其实如果以这种外在的不同来判断是不是真正的国际化就太片面了：不同的国家可以使用相同的语言，但是为了使两个国家的用户都满意，你还有很多工作要做。就像 Oscar Wilde 所说的那样，“我们现在真的是每件东西都和美国一样，当然，语言除外。”

7.1.1 为什么需要 locale

当你提供程序的国际化版本时，所有程序消息都需要转换为本地语言。当然，直接翻译用户界面的文本是不够的，还有许多更细微的差异，例如，数字在英语和德语中格式很不相同。对于德国用户，数字

123,456.78

应该显示为

123.456,78

小数点和十进制数的逗号分隔符的角色是相反的！在日期的显示上也有相似的变化。在美国，日期显示为月 / 日 / 年，这有些不合理。德国使用的是更合理的顺序，即日 / 月 / 年，而在在中国，则使用年 / 月 / 日。因此，对于德国用户，日期

3/22/61

应该被表示为

22.03.1961

当然，如果月份的名称被显式地写了出来，那么语言之间的不同就显而易见了。英语

March 22, 1961

在德国应该被表示成

22. März 1961

在中国则是

1961年3月22日

locale 捕获了像上面这类偏好特征。无论何时，只要你表示数字、日期、货币值以及其他格式会随语言或地点发生变化的项，都需要使用 locale 感知的 API。

7.1.2 指定 locale

locale 由多达 5 个部分构成：

1. 一种语言，由 2 个或 3 个小写字母表示，例如 en（英语）、de（德语）和 zh（中文）。表 7-1 展示了常用的代码。

表 7-1 常见的 ISO-639-1 语言代码

语言	代码	语言	代码
Chinese	zh	Italian	it
Danish	da	Japanese	ja
Dutch	nl	Korean	ko
English	en	Norwegian	no
French	fr	Portuguese	pt
Finnish	fi	Spanish	es
German	de	Swedish	sv
Greek	el	Turkish	tr

2. 可选的一段脚本，由首字母大写的四个字母表示，例如 Latn（拉丁文）、Cyril（西里尔文）和 Hant（繁体中文）。这个部分很有用，因为有些语言，例如塞尔维亚语，可以用拉丁文或西里尔文书写，而有些中文读者更喜欢阅读繁体中文而不是简体中文。

3. 可选的一个国家或地区，由 2 个大写字母或 3 个数字表示，例如 US（美国）和 CH（瑞士）。表 7-2 展示了常用的代码。

表 7-2 常见的 ISO-3166-1 国家代码

国家	代码	国家	代码
Austria	AT	Japan	JP
Belgium	BE	Korea	KR
Canada	CA	The Netherlands	NL
China	CN	Norway	NO
Denmark	DK	Portugal	PT
Finland	FI	Spain	ES
Germany	DE	Sweden	SE
Great Britain	GB	Switzerland	CH
Ireland	IE	Turkey	TR
Italy	IT	United States	US

4. 可选的一个变体，用于指定各种杂项特性，例如方言和拼写规则。变体现在已经很少使用了。过去曾经有一种挪威语的变体“尼诺斯克语”，但是它现在已经用另一种不同的代码 nn 来表示了。过去曾经用于日本帝国历和泰语数字的变体现在也都被表示成了扩展（请参见下一条）。

5. 可选的一个扩展。扩展描述了日历（例如日本历）和数字（替代西方数字的泰语数字）等内容的本地偏好。Unicode 标准规范了其中的某些扩展，这些扩展应该以 u- 和两个字母的代码开头，这两个字母的代码指定了该扩展处理的是日历 (ca) 还是数字 (nu)，或者是其他内容。例如，扩展 u-nu-thai 表示使用泰语数字。其他扩展是完全任意的，并且以 x- 开头，例如 x-java。

locale 的规则在 Internet Engineering Task Force 的“Best Current Practices”备忘录 BCP 47 (<http://tools.ietf.org/html/bcp47>) 中进行了明确阐述。你可以在 www.w3.org/International/articles/language-tags 处找到更容易理解的总结。

语言和国家的代码看起来有点乱，因为它们中的有些是从本地语言导出的。德语在德语中是 Deutsch，中文在中文里是 zhongwen，因此它们分别是 de 和 zh。瑞士是 CH，这是从瑞士联邦的拉丁语 Confoederatio Helvetica 中导出的。

locale 是用标签描述的，标签是由 locale 的各个元素通过连字符连接起来的字符串，例如 en-US。

在德国，你可以使用 de-DE。瑞士有 4 种官方语言（德语、法语、意大利语和里托罗曼斯语）。在瑞士讲德语的人希望使用的 locale 是 de-CH。这个 locale 会使用德语的规则，但是货币值会表示成瑞士法郎而不是欧元。

如果只指定了语言，例如 de，那么该 locale 就不能用于与国家相关的场景，例如货币。

我们可以像下面这样用标签字符串来构建 Locale 对象：

```
Locale usEnglish = Locale.forLanguageTag("en-US");
```

toLanguageTag 方法可以生成给定 locale 的语言标签。例如，Local.US.toLanguageTag() 生成的字符串是 "en-US"。

为方便起见，有许多为各个国家预定义的 Locale 对象：

```
Locale.CANADA
Locale.CANADA_FRENCH
Locale.CHINA
Locale.FRANCE
Locale.GERMANY
Locale.ITALY
Locale.JAPAN
Locale.KOREA
Locale.PRC
Locale.UK
Locale.US
```

还有许多预定义的语言 Locale，它们只设定了语言而没有设定位置：

```
Locale.CHINESE
Locale.ENGLISH
Locale.FRENCH
Locale.GERMAN
Locale.ITALIAN
Locale.JAPANESE
Locale.KOREAN
Locale.SIMPLIFIED_CHINESE
Locale.TRADITIONAL_CHINESE
```

最后，静态的 `getAvailableLocales` 方法会返回由 Java 虚拟机能够识别的所有 locale 构成的数组。

 **注释：**可以用 `Locale.getISOLanguages()` 获取所有语言代码，用 `Locale.getISOCountries()` 获取所有国家代码。

7.1.3 默认 locale

`Locale` 类的静态 `getDefault` 方法可以获得作为本地操作系统的一部分而存放的默认 `locale`。可以调用 `setDefault` 来改变默认的 Java `locale`，但是，这种改变只对你的程序有效，不会对操作系统产生影响。

有些操作系统允许用户为显示消息和格式化指定不同的 `locale`。例如，生活在美国的说法语的人菜单是法语的，但是货币值是用美元来表示的。

要想获取这些偏好，可以调用

```
Locale displayLocale = Locale.getDefault(Locale.Category.DISPLAY);
Locale formatLocale = Locale.getDefault(Locale.Category.FORMAT);
```

 **注释：**在 UNIX 中，可以为数字、货币和日期分别设置 `LC_NUMERIC`、`LC_MONETARY` 和 `LC_TIME` 环境变量来指定不同的 `locale`。但是 Java 并不会关注这些设置。

 **提示：**为了测试，你也许希望改变你的程序的默认 `locale`，可以在启动程序时提供语言和地域特性。比如，下面的语句将默认的 `locale` 设为 `de-CH`：

```
java -Duser.language=de -Duser.region=CH MyProgram
```

7.1.4 显示名字

一旦有了一个 locale，你能用它做什么呢？答案是它所能做的事情很有限。Locale 类中唯一有用的是那些识别语言和国家代码的方法，其中最重要的一个是 `getDisplayName`，它返回一个描述 locale 的字符串。这个字符串并不包含前面所说的由两个字母组成的代码，而是以一种面向用户的形式来表现，比如

```
German (Switzerland)
```

事实上，这里有一个问题，显示的名字是以默认的 locale 来表示的，这可能不太恰当。如果你的用户已经选择了德语作为首选的语言，那么你可能希望将字符串显示成德语。通过将 German locale 作为参数传递就可以做到这一点：代码

```
var loc = new Locale("de", "CH");
System.out.println(loc.getDisplayName(Locale.GERMAN));
```

将打印出

```
Deutsch (Schweiz)
```

这个例子说明了为什么需要 Locale 对象。你把它传给 locale 感知的那些方法，这些方法将根据不同的地域产生不同形式的文本。在后面各节中你可以见到大量的例子。

◆ 警告：即使是像把字符串中的字母全部转换为小写或大写这样简单的操作，也可能是与 locale 相关的。例如，在土耳其 locale 中，字母 I 的小写是不带点的 I。那些试图通过将字符串存储为小写格式来正则化字符串的程序对于土耳其客户来说就会显得很失败，因为 I 和带点的 i 没有相同的小写格式。一种好的做法是总是使用 `toUpperCase` 和 `toLowerCase` 的变体，这种变体会接受一个 Locale 参数。例如，试试下面的代码：

```
String cmd = "QUIT".toLowerCase(Locale.forLanguageTag("tr"));
// "quit" with a dotless i
```

当然，在土耳其，`Locale.getDefault()` 产生的就是那里的 locale，“QUIT”.`toLowerCase()` 与 “quit” 不同。

如果想要将英语字符串正则化为小写形式，那么就应该将英语的 locale 传递给 `toLowerCase` 方法。

■ 注释：你可以显式地指定输入 / 输出操作的 locale。

- 当从 Scanner 读入数字时，可以用 `useLocale` 方法设置它的 locale。
- `String.format` 和 `PrintWriter.printf` 方法也可以接受一个 Locale 参数。

API `java.util.Locale 1.1`

- `Locale(String language)`
- `Locale(String language, String country)`
- `Locale(String language, String country, String variant)`

用给定的语言、国家和变量创建一个 locale。在新代码中不要使用变体，应该使用

IETF BCP 47 语言标签。

- static Locale forLanguageTag(String languageTag) 7

构建与给定的语言标签相对应的 locale。

- static Locale getDefault()

返回默认的 locale。

- static void setDefault(Locale loc)

设定默认的 locale。

- String getDisplayName()

返回一个在当前的 locale 中所表示的用来描述 locale 的名字。

- String getDisplayName(Locale loc)

返回一个在给定的 locale 中所表示的用来描述 locale 的名字。

- String getLanguage()

返回语言代码，它是两个小写字母组成的 ISO 639 代码。

- String getDisplayLanguage()

返回在当前 locale 中所表示的语言名称。

- String getDisplayLanguage(Locale loc)

返回在给定 locale 中所表示的语言名称。

- String getCountry()

返回国家代码，它是由两个大写字母组成的 ISO 3166 代码。

- static String[] getISOCountries()

- static Set<String> getISOCountries(Locale.IsoCountryCode type) 9

获取所有两字母的国家代码，或者所有 2、3、4 个字母的国家代码。type 参数是枚举常量 PART1_ALPHA2、PART1_ALPHA3 和 PART3 之一。

- String getDisplayCountry()

返回在当前 locale 中所表示的国家名。

- String getDisplayCountry(Locale loc)

返回在给定 locale 中所表示的国家名。

- String toLanguageTag() 7

返回该 locale 的语言标签，例如 "de-CH"。

- String toString()

返回 locale 的描述，包括语言和国家，用下划线分隔（比如，"de_CH"）。应该只在调试时使用该方法。

7.2 数字格式

我们已经提到了数字和货币的格式是高度依赖于 locale 的。Java 类库提供了一个格式器

(formatter) 对象的集合，它可以对 java.text 包中的数字值进行格式化和解析。

7.2.1 格式化数字值

可以通过下面的步骤对特定 locale 的数字进行格式化：

1. 使用上一节的方法，得到 Locale 对象。
2. 使用一个“工厂方法”得到一个格式器对象。
3. 使用这个格式器对象来完成格式化和解析工作。

工厂方法是 NumberFormat 类的静态方法，它们接受一个 Locale 类型的参数。总共有 3 个工厂方法 getNumberInstance、getCurrencyInstance 和 getPercentInstance，这些方法返回的对象可以分别对数字、货币量和百分比进行格式化和解析。例如，下面显示了如何对德语中的货币值进行格式化。

```
Locale loc = Locale.GERMAN;
NumberFormat currFmt = NumberFormat.getCurrencyInstance(loc);
double amt = 123456.78;
String result = currFmt.format(amt);
```

结果是

123.456,78 €

请注意，货币符号是 €，而且位于字符串的最后。同时还要注意到小数点和十进制分隔符与其他语言中的情况是相反的。

相反，如果想读取一个按照某个 locale 的惯用法而输入或存储的数字，那么就需要使用 parse 方法。比如，下面的代码解析了用户输入到文本框中的值。parse 方法能够处理小数点和分隔符以及其他语言中的数字。

```
TextField inputField;
...
NumberFormat fmt = NumberFormat.getNumberInstance();
// get the number formatter for default locale
Number input = fmt.parse(inputField.getText().trim());
double x = input.doubleValue();
```

parse 的返回类型是抽象类型 Number。返回的对象是一个 Double 或 Long 的包装器对象，这取决于被解析的数字是否是浮点数。如果不关心两者的差异，可以直接使用 Number 类的 doubleValue 方法来读取被包装的数字。

！ 警告：Number 类型的对象并不能自动转换成相关的基本类型，因此，不能直接将一个 Number 对象赋给一个基本类型，而应该使用 doubleValue 或 intValue 方法。

如果数字文本的格式不正确，该方法会抛出一个 ParseException 异常。例如，字符串以空白字符开头是不允许的（可以调用 trim 方法来去掉它）。但是，任何跟在数字之后的字符都将被忽略，所以这些跟在后面的字符是不会抛出异常的。

请注意，由 getXxxInstance 工厂方法返回的类并非是 NumberFormat 类型的。NumberFormat 类

型是一个抽象类，而我们实际上得到的格式器是它的一个子类。工厂方法只知道如何定位属于特定 locale 的对象。

可以用静态的 `getAvailableLocales` 方法得到一个当前支持的 locale 列表。这个方法返回一个 locale 数组，从中可以获得针对它们的数字格式器对象。

本节的示例程序让你体会到了数字格式器的用法（参见图 7-1）。图上方的组合框包含所有带数字格式器的 locale，可以在数字、货币和百分率格式器之间进行选择。每次改变选择，文本框中的数字就会被重新格式化。在尝试了几种 locale 后，你就会对有这么多种方式来格式化数字和货币值而感到吃惊。也可以输入不同的数字并点击 Parse 按钮来调用 `parse` 方法，这个方法会尝试解析你输入的内容。如果解析成功，`format` 方法就会将结果显示出来。如果解析失败，文本框中会显示“Parse error”消息。

程序清单 7-1 给出了它的代码，非常直观。在构造器中，我们调用 `NumberFormat.getAvailableLocales`。对每一个 locale，我们调用 `getDisplayName`，并把返回的结果字符串填入组合框（字符串没有被排序，在 7.4 节中我们将深入研究排序问题）。一旦用户选择了另一个 locale 或点击了单选按钮，就创建一个新的格式器对象并更新文本框。当用户点击 Parse 按钮后，调用 `Parse` 方法来基于选中的 locale 进行实际的解析操作。

注释：可以使用 `Scanner` 来读取本地化的整数和浮点数。可以调用 `useLocale` 方法来设置 locale。

程序清单 7-1 numberFormat/NumberFormatTest.java

```

1 package numberFormat;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.text.*;
6 import java.util.*;
7
8 import javax.swing.*;
9
10 /**
11 * This program demonstrates formatting numbers under various locales.
12 * @version 1.15 2018-05-01
13 * @author Cay Horstmann
14 */
15 public class NumberFormatTest
16 {
17     public static void main(String[] args)
18     {
19         EventQueue.invokeLater(() ->
20         {
21             var frame = new NumberFormatFrame();
22             frame.setTitle("NumberFormatTest");

```

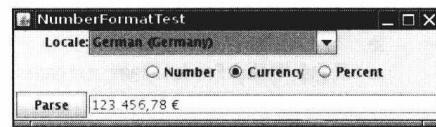


图 7-1 NumberFormatTest 程序

```

23         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24         frame.setVisible(true);
25     });
26 }
27 }
28
29 /**
30  * This frame contains radio buttons to select a number format, a combo box to pick a locale,
31  * a text field to display a formatted number, and a button to parse the text field contents.
32 */
33 class NumberFormatFrame extends JFrame
34 {
35     private Locale[] locales;
36     private double currentNumber;
37     private JComboBox<String> localeCombo = new JComboBox<>();
38     private JButton parseButton = new JButton("Parse");
39     private JTextField numberText = new JTextField(30);
40     private JRadioButton numberRadioButton = new JRadioButton("Number");
41     private JRadioButton currencyRadioButton = new JRadioButton("Currency");
42     private JRadioButton percentRadioButton = new JRadioButton("Percent");
43     private ButtonGroup rbGroup = new ButtonGroup();
44     private NumberFormat currentNumberFormat;
45
46     public NumberFormatFrame()
47     {
48         setLayout(new GridBagLayout());
49
50         ActionListener listener = event -> updateDisplay();
51
52         var p = new JPanel();
53         addRadioButton(p, numberRadioButton, rbGroup, listener);
54         addRadioButton(p, currencyRadioButton, rbGroup, listener);
55         addRadioButton(p, percentRadioButton, rbGroup, listener);
56
57         add(new JLabel("Locale:"), new GBC(0, 0).setAnchor(GBC.EAST));
58         add(p, new GBC(1, 1));
59         add(parseButton, new GBC(0, 2).setInsets(2));
60         add(localeCombo, new GBC(1, 0).setAnchor(GBC.WEST));
61         add(numberText, new GBC(1, 2).setFill(GBC.HORIZONTAL));
62         locales = (Locale[]) NumberFormat.getAvailableLocales().clone();
63         Arrays.sort(locales, Comparator.comparing(Locale::getDisplayName()));
64         for (Locale loc : locales)
65             localeCombo.addItem(loc.getDisplayName());
66         localeCombo.setSelectedItem(Locale.getDefault().getDisplayName());
67         currentNumber = 123456.78;
68         updateDisplay();
69
70         localeCombo.addActionListener(listener);
71
72         parseButton.addActionListener(event ->
73         {
74             String s = numberText.getText().trim();
75             try
76             {

```

```

77         Number n = currentNumberFormat.parse(s);
78         currentNumber = n.doubleValue();
79         updateDisplay();
80     }
81     catch (ParseException e)
82     {
83         numberText.setText(e.getMessage());
84     }
85 }
86 pack();
87 }
88
89 /**
90 * Adds a radio button to a container.
91 * @param p the container into which to place the button
92 * @param b the button
93 * @param g the button group
94 * @param listener the button listener
95 */
96 public void addRadioButton(Container p, JRadioButton b, ButtonGroup g,
97     ActionListener listener)
98 {
99     b.setSelected(g.getButtonCount() == 0);
100    b.addActionListener(listener);
101    g.add(b);
102    p.add(b);
103 }
104
105 /**
106 * Updates the display and formats the number according to the user settings.
107 */
108 public void updateDisplay()
109 {
110     Locale currentLocale = locales[localeCombo.getSelectedIndex()];
111     currentNumberFormat = null;
112     if (numberRadioButton.isSelected())
113         currentNumberFormat = NumberFormat.getNumberInstance(currentLocale);
114     else if (currencyRadioButton.isSelected())
115         currentNumberFormat = NumberFormat.getCurrencyInstance(currentLocale);
116     else if (percentRadioButton.isSelected())
117         currentNumberFormat = NumberFormat.getPercentInstance(currentLocale);
118     String formatted = currentNumberFormat.format(currentNumber);
119     numberText.setText(formatted);
120 }
121 }

```

API **java.text.NumberFormat 1.1**

- static Locale[] getAvailableLocales()

返回一个 Locale 对象的数组，其成员为可用的 NumberFormat 格式器。

- static NumberFormat getInstance()
- static NumberFormat getInstance(Locale l)

- static NumberFormat getCurrencyInstance()
- static NumberFormat getCurrencyInstance(Locale l)
- static NumberFormat getPercentInstance()
- static NumberFormat getPercentInstance(Locale l)

为当前或给定的 locale 提供处理数字、货币量或百分比的格式器。

- String format(double x)

对给定的浮点数或整数进行格式化并以字符串的形式返回结果。

- Number parse(String s)

解析给定的字符串并返回数字值，如果输入字符串描述了一个浮点数，返回类型就是 Double，否则返回类型就是 Long。字符串必须以一个数字开头，以空白字符开头是不允许的。数字之后可以跟随其他字符，但它们都将被忽略。解析失败时抛出 ParseException 异常。

- void setParseIntegerOnly(boolean b)

- boolean isParseIntegerOnly()

设置或获取一个标志，该标志指示这个格式器是否应该只解析整数值。

- void setGroupingUsed(boolean b)

- boolean isGroupingUsed()

设置或获取一个标志，该标志指示这个格式器是否会添加和识别十进制分隔符（比如，100,000）。

- void setMinimumIntegerDigits(int n)

- int getMinimumIntegerDigits()

- void setMaximumIntegerDigits(int n)

- int getMaximumIntegerDigits()

- void setMinimumFractionDigits(int n)

- int getMinimumFractionDigits()

- void setMaximumFractionDigits(int n)

- int getMaximumFractionDigits()

设置或获取整数或小数部分所允许的最大或最小位数。

7.2.2 货币

为了格式化货币值，可以使用 NumberFormat.getCurrencyInstance 方法。但是，这个方法的灵活性不好，它返回的是一个只针对一种货币的格式器。假设你为一个美国客户准备了一张货物单，货物单中有些货物的金额是用美元表示的，有些是用欧元表示的，此时，你不能只是使用两种格式器：

```
NumberFormat dollarFormatter = NumberFormat.getCurrencyInstance(Locale.US);
NumberFormat euroFormatter = NumberFormat.getCurrencyInstance(Locale.GERMANY);
```

这是因为，这样一来，你的发票看起来非常奇怪，有些金额的格式像 \$100,000，另一些则像 100.000 €（注意，欧元值使用小数点而不是逗号作为分隔符）。

处理这样的情况，应该使用 `Currency` 类来控制被格式器处理的货币。可以通过将一个货币标识符传给静态的 `Currency.getInstance` 方法来得到一个 `Currency` 对象，然后对每一个格式器都调用 `setCurrency` 方法。下面展示了如何为你的美国客户设置欧元的格式：

```
NumberFormat euroFormatter = NumberFormat.getCurrencyInstance(Locale.US);
euroFormatter.setCurrency(Currency.getInstance("EUR"));
```

货币标识符由 ISO 4217 定义，可参考 <https://www.iso.org/iso-4217-currency-codes.html>。表 7-3 提供了其中的一部分。

表 7-3 货币标识符

货币值	标识符	货币代号	货币值	标识符	货币代号
U.S. Dollar	USD	840	Chinese Renminbi (Yuan)	CNY	156
Euro	EUR	978	Indian Rupee	INR	356
British Pound	GBP	826	Russian Ruble	RUB	643
Japanese Yen	JPY	392			

API `java.util.Currency` 1.4

- `static Currency getInstance(String currencyCode)`
- `static Currency getInstance(Locale locale)`

返回与给定的 ISO 4217 货币代号或给定的 `locale` 中的国家相对应的 `Currency` 对象。

- `String toString()`
- `String getCurrencyCode()`
- `String getNumericCode() 7`
- `String getNumericCodeAsString() 9`

获取该货币的 ISO 4217 代码。

- `String getSymbol()`
- `String getSymbol(Locale locale)`

根据默认或给定的 `locale` 得到该货币的格式化符号。比如美元的格式化符号可能是 "\$" 或 "US\$"，具体是哪种形式取决于 `locale`。

- `int getDefaultFractionDigits()`
- 获取该货币小数点后的默认位数。
- `static Set<Currency> getAvailableCurrencies() 7`
- 获取所有可用的货币。

7.3 日期和时间

当格式化日期和时间时，需要考虑 4 个与 `locale` 相关的问题：

- 月份和星期应该用本地语言来表示。
- 年、月、日的顺序要符合本地习惯。
- 公历可能不是本地首选的日期表示方法。
- 必须要考虑本地的时区。

`java.time` 包中的 `DateTimeFormatter` 类可以处理这些问题。首先挑选表 7-4 中所示的一种格式风格，然后获取一个格式器：

```
FormatStyle style = . . .; // One of FormatStyle.SHORT, FormatStyle.MEDIUM, . . .
DateTimeFormatter dateFormatter = DateTimeFormatter.ofLocalizedDate(style);
DateTimeFormatter timeFormatter = DateTimeFormatter.ofLocalTime(style);
DateTimeFormatter dateTimeFormatter = DateTimeFormatter.ofLocalDateTime(style);
// or DateTimeFormatter.ofLocalizedDateTime(style1, style2)
```

表 7-4 日期和时间的格式化风格

风格	日期	时间
SHORT	7/16/69	9:32 AM
MEDIUM	Jul 16, 1969	9:32:00 AM
LONG	July 16, 1969	9:32:00 AM EDT in en-US, 9:32:00 MSZ in de-DE (只用于 ZonedDateTime)
FULL	Wednesday, July 16, 1969	9:32:00 AM EDT in en-US, 9:32 Uhr MSZ in de-DE (只用于 ZonedDateTime)

这些格式器都会使用当前的 `locale`。为了使用不同的 `locale`，需要使用 `withLocale` 方法：

```
DateTimeFormatter dateFormatter =
    DateTimeFormatter.ofLocalizedDate(style).withLocale(locale);
```

现在你可以格式化 `LocalDate`、`LocalDateTime`、`LocalTime` 和 `ZonedDateTime` 了：

```
ZonedDateTime appointment = . . .;
String formatted = formatter.format(appointment);
```

注释：这里我们使用的是 `java.time` 包中的 `DateTimeFormatter`。还有一种来自 Java 1.1 的遗留的 `java.text.SimpleDateFormat` 类，它可以操作 `Date` 和 `Calendar` 对象。

可以使用 `LocalDate`、`LocalDateTime`、`LocalTime` 和 `ZonedDateTime` 的静态 `parse` 方法之一来解析字符串中的日期和时间：

```
LocalTime time = LocalTime.parse("9:32 AM", formatter);
```

这些方法不适合解析人类的输入，至少不适合解析未做预处理的人类输入。例如，用于美国的短时间格式器可以解析 "9:32 AM"，但是解析不了 "9:32AM" 和 "9:32 am"。

警告：日期格式器可以解析不存在的日期，例如 November 31，它会将这种日期调整为给定月份的最后一天。

有时，你需要显示星期和月份的名字，例如在日历应用中。此时可以调用 `DayOfWeek` 和 `Month` 枚举的 `getDisplayName` 方法：

```
for (Month m : Month.values())
    System.out.println(m.getDisplayName(textStyle, locale) + " ");
```

表 7-5 展示了文本风格，其中 STANDALONE 版本用于格式化日期之外的显示。例如，在芬兰语中，一月在日期中是“tammikuuta”，但是单独显示时是“tammikuu”。

表 7-5 java.time.format.TextStyle 枚举

风格	示例
FULL / FULL_STANDALONE	January
SHORT / SHORT_STANDALONE	Jan
NARROW / NARROW_STANDALONE	J

注释：星期的第一天可以是星期六、星期日或星期一，这取决于 `locale`。你可以像下面这样获取星期的第一天：

```
DayOfWeek first = WeekFields.of(locale).getFirstDayOfWeek();
```

程序清单 7-2 展示了如何在实际中使用 `DateFormat` 类，用户可以选择一个 `locale` 并看看日期和时间在世界上的不同地区是如何格式化的。

程序清单 7-2 dateFormat/DateTimeFormatterTest.java

```
1 package dateFormat;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.time.*;
6 import java.time.format.*;
7 import java.util.*;
8
9 import javax.swing.*;
10
11 /**
12 * This program demonstrates formatting dates under various locales.
13 * @version 1.01 2018-05-01
14 * @author Cay Horstmann
15 */
16 public class DateTimeFormatterTest
17 {
18     public static void main(String[] args)
19     {
20         EventQueue.invokeLater(() ->
21         {
22             var frame = new DateTimeFormatterFrame();
23             frame.setTitle("DateFormatTest");
24             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
25             frame.setVisible(true);
26         });
27     }
28 }
```

```

30  /**
31  * This frame contains combo boxes to pick a locale, date and time formats, text fields to
32  * display formatted date and time, buttons to parse the text field contents, and a "lenient"
33  * check box.
34 */
35 class DateTimeFormatterFrame extends JFrame
36 {
37     private Locale[] locales;
38     private LocalDate currentDate;
39     private LocalTime currentTime;
40     private ZonedDateTime currentTime;
41     private DateTimeFormatter currentDateFormat;
42     private DateTimeFormatter currentTimeFormat;
43     private DateTimeFormatter currentDateTimeFormat;
44     private JComboBox<String> localeCombo = new JComboBox<>();
45     private JButton dateParseButton = new JButton("Parse");
46     private JButton timeParseButton = new JButton("Parse");
47     private JButton dateTimeParseButton = new JButton("Parse");
48     private JTextField dateText = new JTextField(30);
49     private JTextField timeText = new JTextField(30);
50     private JTextField dateTimeText = new JTextField(30);
51     private EnumCombo<FormatStyle> dateStyleCombo = new EnumCombo<>(FormatStyle.class,
52         "Short", "Medium", "Long", "Full");
53     private EnumCombo<FormatStyle> timeStyleCombo = new EnumCombo<>(FormatStyle.class,
54         "Short", "Medium");
55     private EnumCombo<FormatStyle> dateTimeStyleCombo = new EnumCombo<>(FormatStyle.class,
56         "Short", "Medium", "Long", "Full");
57
58     public DateTimeFormatterFrame()
59     {
60         setLayout(new GridBagLayout());
61         add(new JLabel("Locale"), new GBC(0, 0).setAnchor(GBC.EAST));
62         add(localeCombo, new GBC(1, 0, 2, 1).setAnchor(GBC.WEST));
63
64         add(new JLabel("Date"), new GBC(0, 1).setAnchor(GBC.EAST));
65         add(dateStyleCombo, new GBC(1, 1).setAnchor(GBC.WEST));
66         add(dateText, new GBC(2, 1, 2, 1).setFill(GBC.HORIZONTAL));
67         add(dateParseButton, new GBC(4, 1).setAnchor(GBC.WEST));
68
69         add(new JLabel("Time"), new GBC(0, 2).setAnchor(GBC.EAST));
70         add(timeStyleCombo, new GBC(1, 2).setAnchor(GBC.WEST));
71         add(timeText, new GBC(2, 2, 2, 1).setFill(GBC.HORIZONTAL));
72         add(timeParseButton, new GBC(4, 2).setAnchor(GBC.WEST));
73
74         add(new JLabel("Date and time"), new GBC(0, 3).setAnchor(GBC.EAST));
75         add(dateTimeStyleCombo, new GBC(1, 3).setAnchor(GBC.WEST));
76         add(dateTimeText, new GBC(2, 3, 2, 1).setFill(GBC.HORIZONTAL));
77         add(dateTimeParseButton, new GBC(4, 3).setAnchor(GBC.WEST));
78
79         locales = (Locale[]) Locale.getAvailableLocales().clone();
80         Arrays.sort(locales, Comparator.comparing(Locale::getDisplayName));
81         for (Locale loc : locales)
82             localeCombo.addItem(loc.getDisplayName());
83         localeCombo.setSelectedItem(Locale.getDefault().getDisplayName());

```

```
84     currentDate = LocalDate.now();
85     currentTime = LocalTime.now();
86     currentDateTime = ZonedDateTime.now();
87     updateDisplay();
88
89     ActionListener listener = event -> updateDisplay();
90     localeCombo.addActionListener(listener);
91     dateStyleCombo.addActionListener(listener);
92     timeStyleCombo.addActionListener(listener);
93     dateTimeStyleCombo.addActionListener(listener);
94
95     addAction(dateParseButton, () ->
96     {
97         currentDate = LocalDate.parse(dateText.getText().trim(), currentDateFormat);
98     });
99     addAction(timeParseButton, () ->
100    {
101        currentTime = LocalTime.parse(timeText.getText().trim(), currentTimeFormat);
102    });
103    addAction(dateTimeParseButton, () ->
104    {
105        currentDateTime = ZonedDateTime.parse(
106            dateTimeText.getText().trim(), currentDateTimeFormat);
107    });
108
109    pack();
110 }
111
112 /**
113 * Adds the given action to the button and updates the display upon completion.
114 * @param button the button to which to add the action
115 * @param action the action to carry out when the button is clicked
116 */
117 public void addAction(JButton button, Runnable action)
118 {
119     button.addActionListener(event ->
120     {
121         try
122         {
123             action.run();
124             updateDisplay();
125         }
126         catch (Exception e)
127         {
128             JOptionPane.showMessageDialog(null, e.getMessage());
129         }
130     });
131 }
132
133 /**
134 * Updates the display and formats the date according to the user settings.
135 */
136 public void updateDisplay()
137 {
```

```

138     Locale currentLocale = locales[localeCombo.getSelectedIndex()];
139     FormatStyle dateStyle = dateStyleCombo.getValue();
140     currentDateFormat = DateTimeFormatter.ofLocalizedDate(
141         dateStyle).withLocale(currentLocale);
142     dateText.setText(currentDateFormat.format(currentDate));
143     FormatStyle timeStyle = timeStyleCombo.getValue();
144     currentTimeFormat = DateTimeFormatter.ofLocalizedTime(
145         timeStyle).withLocale(currentLocale);
146     timeText.setText(currentTimeFormat.format(currentTime));
147     FormatStyle dateTimeStyle = dateTimeStyleCombo.getValue();
148     currentDateTimeFormat = DateTimeFormatter.ofLocalizedDateTime(
149         dateTimeStyle).withLocale(currentLocale);
150     dateTimeText.setText(currentDateTimeFormat.format(currentDateTime));
151 }
152 }

```

图 7-2 显示了该程序（已安装中文字体）。就像你看到的那样，输出能够正确显示。

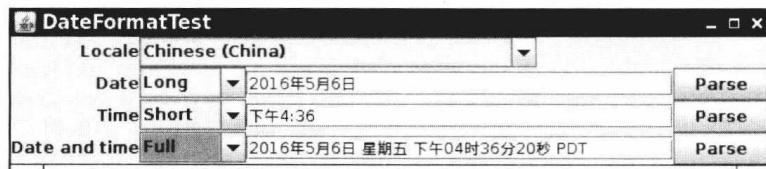


图 7-2 DateFormatTest 程序

也可以对解析进行试验。输入一个日期或时间，或同时输入日期和时间，然后点击 Parse 按钮。

我们使用了辅助类 `EnumCombo` 来解决一个技术问题（参见程序清单 7-3）。我们想用 `Short`、`Medium` 和 `Long` 等值来填充一个组合框（combo），然后自动将用户的选择转换成整数值 `FormatStyle.SHORT`、`FormatStyle.MEDIUM` 和 `FormatStyle.LONG`。我们并没有编写重复的代码，而是使用了反射：我们将用户的选择转换成大写字母，所有空格都用下划线替换，然后找到使用这个名字的静态域的值。（更多关于反射的内容参见卷 I 第 5 章。）

程序清单 7-3 dateFormat/EnumCombo.java

```

1 package dateFormat;
2
3 import java.util.*;
4 import javax.swing.*;
5
6 /**
7  * A combo box that lets users choose from among static field
8  * values whose names are given in the constructor.
9  * @version 1.15 2016-05-06
10 * @author Cay Horstmann
11 */
12 public class EnumCombo<T> extends JComboBox<String>
13 {
14     private Map<String, T> table = new TreeMap<>();

```

```

15
16  /**
17  * Constructs an EnumCombo yielding values of type T.
18  * @param cl a class
19  * @param labels an array of strings describing static field names
20  * of cl that have type T
21  */
22 public EnumCombo(Class<?> cl, String... labels)
23 {
24     for (String label : labels)
25     {
26         String name = label.toUpperCase().replace(' ', '_');
27         try
28         {
29             java.lang.reflect.Field f = cl.getField(name);
30             @SuppressWarnings("unchecked") T value = (T) f.get(cl);
31             table.put(label, value);
32         }
33         catch (Exception e)
34         {
35             label = "(" + label + ")";
36             table.put(label, null);
37         }
38         addItem(label);
39     }
40     setSelectedItem(labels[0]);
41 }
42
43 /**
44 * Returns the value of the field that the user selected.
45 * @return the static field value
46 */
47 public T getValue()
48 {
49     return table.get(getSelectedItem());
50 }
51 }

```

API `java.time.format.DateTimeFormatter` 8

- `static DateTimeFormatter ofLocalizedDate(FormatStyle dateStyle)`
- `static DateTimeFormatter ofLocalizedTime(FormatStyle dateStyle)`
- `static DateTimeFormatter ofLocalizedDateTime(FormatStyle dateTimeStyle)`
- `static DateTimeFormatter ofLocalizedDate(FormatStyle dateStyle, FormatStyle timeStyle)`
返回用指定的风格格式化日期、时间或日期和时间的 `DateTimeFormatter` 实例。
- `DateTimeFormatter withLocale(Locale locale)`
返回当前格式器的具有给定 `locale` 的副本。
- `String format(TemporalAccessor temporal)`
返回格式化给定日期 / 时间所产生的字符串。

API	<code>java.time.LocalDate 8</code>
	<code>java.time.LocalTime 8</code>
	<code>java.time.LocalDateTime 8</code>
	<code>java.time.ZonedDateTime 8</code>

- static Xxx parse(CharSequence text, DateTimeFormatter formatter)

解析给定的字符串并返回其中描述的 LocalDate、LocalTime、LocalDateTime 或 ZonedDateTime。如果解析不成功，则抛出 DateTimeParseException 异常。

7.4 排序和规范化

大多数程序员都知道如何使用 String 类中的 compareTo 方法对字符串进行比较。但是，当与人类用户交互时，这个方法就不是很有用了。compareTo 方法使用的是字符串的 UTF-16 编码值，这会导致很荒唐的结果，即使在英文比较中也是如此。比如，下面的 5 个字符串进行排序的结果为：

```
America
Zulu
able
zebra
Ångström
```

按照字典中的顺序，你希望将大写和小写看作是等价的。对于一个说英语的读者来说，期望的排序结果应该是：

```
able
America
Ångström
zebra
Zulu
```

但是，这种顺序对于瑞典用户是不可接受的。在瑞典语中，字母 Å 和字母 A 是不同的，它应该排在字母 Z 之后！就是说，瑞典用户希望排序的结果是：

```
able
America
zebra
Zulu
Ångström
```

为了获得 locale 敏感的比较器，可以调用静态的 Collator.getInstance 方法：

```
Collator coll = Collator.getInstance(locale);
words.sort(coll); // Collator implements Comparator<Object>
```

因为 Collator 类实现了 Comparator 接口，因此，可以传递一个 Collator 对象给 list.sort(Comparator) 方法来对一组字符串进行排序。

排序器有几个高级设置项。你可以设置排序器的强度来选择不同的排序行为。字符间的差别可以被分为首要的（primary）、其次的（secondary）和再次的（tertiary）。比如，在英语

中，“A”和“Z”之间的差别被归为首要的，而“A”和“Å”之间的差别是其次的，“A”和“a”之间的差别是再次的。

如果将排序器的强度设置成 `Collator.PRIMARY`，那么排序器将只关注 primary 级的差别。如果设置成 `Collator.SECONDARY`，排序器将把 secondary 级的差别也考虑进去。就是说，两个字符串在“secondary”或“tertiary”强度下更容易被区分开来，如表 7-6 所示。

表 7-6 不同强度下的排序（英语 locale）

首要	其次	再次
<code>Angstrom = Ångström</code>	<code>Angstrom ≠ Ångström</code>	<code>Angstrom ≠ Ångström</code>
<code>Able = able</code>	<code>Able = able</code>	<code>Able ≠ able</code>

如果强度被设置为 `Collator.IDENTICAL`，则不允许有任何差别。这种设置在与排序器的另一种具有相当技术性的设置即分解模式（decomposition mode）联合使用时，显得非常有用。我们接下来将讨论分解模式。

偶尔我们会碰到一个字符或字符序列在被描述成 Unicode 时有多种方式的情况。例如，“Å”可以是 Unicode 字符 U+00C5，或者可以表示成普通的 A (U+0065) 后跟°（“上方组合环”，U+030A）。也许让你更吃惊的是，字母序列“ffi”可以用代码 U+FB03 描述成单个字符“拉丁小连字 ffi”。（有人会说这是表示方法的不同，不应该因此产生不同的 Unicode 字符，但规则不是我们定的。）

Unicode 标准对字符串定义了四种规范化形式（normalization form）：D、KD、C 和 KC。请查看 <http://www.unicode.org/unicode/reports/tr15/tr15-23.html> 以了解详细信息。在规范化形式 C 中，重音符号总是组合的。例如，A 和上方组合环° 被组合成了单个字符 Å。在规范化形式 D 中，重音字符被分解为基字符和组合重音符。例如，Å 就被转换成由字母 A 和上方组合环° 构成的序列。规范化形式 KC 和 KD 也会分解字符，例如连字或商标符号。

我们可以选择排序器所使用的规范化程度：`Collator.NO_DECOMPOSITION` 表示不对字符串做任何规范化，这个选项处理速度较快，但是对于以多种形式表示字符的文本就不适用了；默认值 `Collator.CANONICAL_DECOMPOSITION` 使用规范化形式 D，这对于包含重音但不包含连字的文本是非常有用的形式；最后是使用规范化形式 KD 的“完全分解”。请参见表 7-7 中的示例。

表 7-7 分解模式之间的差异

不分解	规范分解	完全分解
<code>Å ≠ A°</code>	<code>Å=A°</code>	<code>Å=A°</code>
<code>TM ≠ TM</code>	<code>TM ≠ TM</code>	<code>TM=TM</code>

让排序器去多次分解一个字符串是很浪费的。如果一个字符串要和其他字符串进行多次比较，可以将分解的结果保存在一个排序键对象中。`getCollationKey` 方法返回一个 `CollationKey` 对象，可以用它来进行更进一步、更快速的比较操作。下面是一个例子：

```

String a = . . .;
CollationKey aKey = coll.getCollationKey(a);
if(aKey.compareTo(coll.getCollationKey(b)) == 0) // fast comparison
    . . .

```

最后，有可能在你不需要进行排序时，也希望将字符串转换成其规范化形式。例如，在将字符串存储到数据库中，或与其他程序进行通信时。`java.text.Normalizer` 类实现了对规范化的处理。例如：

```

String name = "Ångström";
String normalized = Normalizer.normalize(name, Normalizer.Form.NFD); // uses normalization
// form D

```

上面的字符串规范化后包含 10 个字符，其中“Å”和“ö”被替换成“A°”和“o°”序列。

但是，这种形式通常并不是用于存储或传输的最佳形式。规范化形式 C 首先进行分解，然后将重音按照标准化的顺序组合在后面。根据 W3C 的标准，这是用于在因特网上进行数据传输的推荐模式。

程序清单 7-4 中的程序让你体验了一下比较排序。你可以向文本框中输入一个词然后点击 Add 按钮把它添加到一个单词列表中。每当添加一个单词，或选择 locale、强度或分解模式时，列表中的单词就会被重新排列。`=` 号表示这两个词被认为是等同的（参见图 7-3）。

组合框中 `locale` 名字的显示顺序，是用默认 `locale` 的排序器进行排序而产生的顺序。如果用美国英语 `locale` 运行这个程序，即使逗号的 Unicode 值比右括号的 Unicode 值大，“Norwegian (Norway, Nynorsk)” 也会显示在 “Norwegian (Norway)” 的前面。



图 7-3 CollationTest 程序

程序清单 7-4 collation/CollationTest.java

```

1 package collation;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.text.*;
6 import java.util.*;
7 import java.util.List;
8
9 import javax.swing.*;
10
11 /**
12 * This program demonstrates collating strings under various locales.
13 * @version 1.16 2018-05-01
14 * @author Cay Horstmann
15 */

```

```
16 public class CollationTest
17 {
18     public static void main(String[] args)
19     {
20         EventQueue.invokeLater(() ->
21         {
22             var frame = new CollationFrame();
23             frame.setTitle("CollationTest");
24             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
25             frame.setVisible(true);
26         });
27     }
28 }
29
30 /**
31 * This frame contains combo boxes to pick a locale, collation strength and decomposition
32 * rules, a text field and button to add new strings, and a text area to list the collated
33 * strings.
34 */
35 class CollationFrame extends JFrame
36 {
37     private Collator collator = Collator.getInstance(Locale.getDefault());
38     private List<String> strings = new ArrayList<>();
39     private Collator currentCollator;
40     private Locale[] locales;
41     private JComboBox<String> localeCombo = new JComboBox<>();
42     private JTextField newWord = new JTextField(20);
43     private JTextArea sortedWords = new JTextArea(20, 20);
44     private JButton addButton = new JButton("Add");
45     private EnumCombo<Integer> strengthCombo = new EnumCombo<>(Collator.class, "Primary",
46         "Secondary", "Tertiary", "Identical");
47     private EnumCombo<Integer> decompositionCombo = new EnumCombo<>(Collator.class,
48         "Canonical Decomposition", "Full Decomposition", "No Decomposition");
49
50     public CollationFrame()
51     {
52         setLayout(new GridBagLayout());
53         add(new JLabel("Locale"), new GBC(0, 0).setAnchor(GBC.EAST));
54         add(new JLabel("Strength"), new GBC(0, 1).setAnchor(GBC.EAST));
55         add(new JLabel("Decomposition"), new GBC(0, 2).setAnchor(GBC.EAST));
56         add(addButton, new GBC(0, 3).setAnchor(GBC.EAST));
57         add(localeCombo, new GBC(1, 0).setAnchor(GBC.WEST));
58         add(strengthCombo, new GBC(1, 1).setAnchor(GBC.WEST));
59         add(decompositionCombo, new GBC(1, 2).setAnchor(GBC.WEST));
60         add(newWord, new GBC(1, 3).setFill(GBC.HORIZONTAL));
61         add(new JScrollPane(sortedWords), new GBC(0, 4, 2, 1).setFill(GBC.BOTH));
62
63         locales = (Locale[]) Collator.getAvailableLocales().clone();
64         Arrays.sort(locales,
65             (l1, l2) -> collator.compare(l1.getDisplayName(), l2.getDisplayName()));
66         for (Locale loc : locales)
67             localeCombo.addItem(loc.getDisplayName());
68         localeCombo.setSelectedItem(Locale.getDefault().getDisplayName());
69 }
```

```
70     strings.add("America");
71     strings.add("able");
72     strings.add("Zulu");
73     strings.add("zebra");
74     strings.add("\u00C5ngstr\u00F6m");
75     strings.add("A\u030angstro\u0308m");
76     strings.add("Angstrom");
77     strings.add("Able");
78     strings.add("office");
79     strings.add("o\uFB03ce");
80     strings.add("Java\u2122");
81     strings.add("JavaTM");
82     updateDisplay();
83
84     addButton.addActionListener(event ->
85     {
86         strings.add(newWord.getText());
87         updateDisplay();
88     });
89
90     ActionListener listener = event -> updateDisplay();
91
92     localeCombo.addActionListener(listener);
93     strengthCombo.addActionListener(listener);
94     decompositionCombo.addActionListener(listener);
95     pack();
96 }
97
98 /**
99  * Updates the display and collates the strings according to the user settings.
100 */
101 public void updateDisplay()
102 {
103     Locale currentLocale = locales[localeCombo.getSelectedIndex()];
104     localeCombo.setLocale(currentLocale);
105
106     currentCollator = Collator.getInstance(currentLocale);
107     currentCollator.setStrength(strengthCombo.getValue());
108     currentCollator.setDecomposition(decompositionCombo.getValue());
109
110     strings.sort(currentCollator);
111
112     sortedWords.setText("");
113     for (int i = 0; i < strings.size(); i++)
114     {
115         String s = strings.get(i);
116         if (i > 0 && currentCollator.compare(s, strings.get(i - 1)) == 0)
117             sortedWords.append(" = ");
118         sortedWords.append(s + "\n");
119     }
120     pack();
121 }
122 }
```

API **java.text.Collator 1.1**

- **static Locale[] getAvailableLocales()**
返回 Locale 对象的一个数组，该 Collator 对象可用于这些对象。
- **static Collator getInstance()**
- **static Collator getInstance(Locale l)**
为默认或给定的 locale 返回一个排序器。
- **int compare(String a, String b)**
如果 a 在 b 之前，则返回负值；如果它们相等，则返回 0；否则返回正值。
- **boolean equals(String a, String b)**
如果 a 和 b 相等，则返回 true，否则返回 false。
- **void setStrength(int strength)**
- **int getStrength()**
设置或获取排序器的强度。更强的排序器可以区分更多的词。强度的值可以是 Collator.PRIMARY、Collator.SECONDARY 和 Collator.TERTIARY。
- **void setDecomposition(int decomp)**
- **int getDecompositon()**
设置或获取排序器的分解模式。分解越细，判断两个字符串是否相等时就越严格。分解的等级值可以是 Collator.NO_DECOMPOSITION、Collator.CANONICAL_DECOMPOSITION 和 Collator.FULL_DECOMPOSITION。
- **CollationKey getCollationKey(String a)**
返回一个排序器键，这个键包含一个对一组字符按特定格式分解的结果，可以快速地和其他排序器键进行比较。

API **java.text.CollationKey 1.1**

- **int compareTo(CollationKey b)**
如果这个键在 b 之前，则返回一个负值；如果两者相等，则返回 0，否则返回正值。

API **java.text.Normalizer 6**

- **static String normalize(CharSequence str, Normalizer.Form form)**
返回 str 的规范化形式，form 的值是 ND、NKD、NC 或 NKC 之一。

7.5 消息格式化

Java 类库中有一个用来对包含变量部分的文本进行格式化的 `MessageFormat` 类，它的格式化方式与用 `printf` 方法进行格式化很类似，但是它支持 `locale`，并且可以对数字和日期进行格式化。我们将在以下各节中审视这种机制。

7.5.1 格式化数字和日期

下面是一个典型的消息格式化字符串：

```
"On {2}, a {0} destroyed {1} houses and caused {3} of damage."
```

括号中的数字是占位符，可以用实际的名字和值来替换它们。使用静态方法 `MessageFormat.format` 可以用实际的值来替换这些占位符。它是一个“varargs”方法，所以可以通过下面的方法提供参数：

```
String msg
= MessageFormat.format("On {2}, a {0} destroyed {1} houses and caused {3} of damage.",
    "hurricane", 99, new GregorianCalendar(1999, 0, 1).getTime(), 10.0E8);
```

在这个例子中，占位符 {0} 被 "hurricane" 替换，{1} 被 99 替换，等等。

上述例子的结果是下面的字符串：

```
On 1/1/99 12:00 AM, a hurricane destroyed 99 houses and caused 100,000,000 of damage.
```

这只是开始，离完美还有距离。我们不想将时间显示为“12:00 AM”，而且我们想将造成的损失量打印成货币值。通过为占位符提供可选的格式，就可以做到这一点：

```
"On {2,date,long}, a {0} destroyed {1} houses and caused {3,number,currency} of damage."
```

这段示例代码将打印出：

```
On January 1, 1999, a hurricane destroyed 99 houses and caused $100,000,000 of damage.
```

一般来说，占位符索引后面可以跟一个类型（type）和一个风格（style），它们之间用逗号隔开。类型可以是：

```
number
time
date
choice
```

如果类型是 `number`，那么风格可以是

```
integer
currency
percent
```

或者是数字格式模式，如 `$,##0`。（关于格式的更多信息，可参见 `DecimalFormat` 类的文档。）

如果类型是 `time` 或 `date`，那么风格可以是

```
short
medium
long
full
```

或者是一个日期格式模式，如 `yyyy-MM-dd`。（关于格式的更多信息，可参见 `SimpleDateFormat` 类的文档。）

◆ 警告： 静态的 `MessageFormat.format` 方法使用当前的 `locale` 对值进行格式化。要想用任

意的 locale 进行格式化，还有一些工作要做，因为这个类还没有提供任何可以使用的“varargs”方法。你需要把将要格式化的值置于 Object[] 数组中，就像下面这样：

```
var mf = new MessageFormat(pattern, loc);
String msg = mf.format(new Object[] { values });
```

API **java.text.MessageFormat 1.1**

- `MessageFormat(String pattern)`
- `MessageFormat(String pattern, Locale loc)`
用给定的模式和 locale 构建一个消息格式对象。
- `void applyPattern(String pattern)`
给消息格式对象设置特定的模式。
- `void setLocale(Locale loc)`
- `Locale getLocale()`

设置或获取消息中占位符所使用的 locale。这个 locale 仅仅被通过调用 `applyPattern` 方法所设置的后续模式使用。

- `static String format(String pattern, Object... args)`
通过使用 `args[i]` 作为占位符 {i} 的输入来格式化 pattern 字符串。
- `StringBuffer format(Object args, StringBuffer result, FieldPosition pos)`
格式化 `MessageFormat` 的模式。`args` 参数必须是一个对象数组。被格式化的字符串会被附加到 `result` 末尾，并返回 `result`。如果 `pos` 等于 `new FieldPosition(MessageFormat.Field.ARGUMENT)`，就用它的 `beginIndex` 和 `endIndex` 属性值来设置替换占位符 {1} 的文本位置。如果不关心位置信息，可以将它设为 `null`。

API **java.text.Format 1.1**

- `String format(Object obj)`
按照格式器的规则格式化给定的对象，这个方法将调用 `format(obj,new StringBuffer(),new FieldPosition(1)).toString()`。

7.5.2 选择格式

让我们仔细地看看前面一节所提到的模式：

"On {2}, a {0} destroyed {1} houses and caused {3} of damage."

如果我们用 "earthquake" 来替换代表灾难的占位符 {0}，那么，在英语中，这句话的语法就不正确了。

On January 1, 1999, a earthquake destroyed . . .

这说明，我们真正希望的是将冠词“a”集成到占位符中去：

"On {2}, {0} destroyed {1} houses and caused {3} of damage."

这样我们就应该用 "a hurricane" 或 "an earthquake" 来替换 {0}。当消息需要被翻译成某种语言，而该语言中的词会随词性的变化而变化时，这种替换方式特别适用。比如，在德语中，模式可能是：

```
"{0} zerstörte am {2} {1} Häuser und richtete einen Schaden von {3} an."
```

这样，占位符将被正确地替换成冠词和名词的组合，比如 "Ein Wirbelsturm" 或 "Eine Naturkatastrophe"。

让我们来看看参数 {1}。如果灾难的后果不严重，{1} 的替换值可能是数字 1，消息就变成：

```
On January 1, 1999, a mudslide destroyed 1 houses and . . .
```

我们当然希望消息能够随占位符的值而变化，这样就能根据具体的值形成

```
no houses
one house
2 houses
. . .
```

`choice` 格式化选项就是为了这个目的而设计的。

一个选择格式是由一个序列对构成的，每一个对包括：

- 一个下限 (lower limit)
- 一个格式字符串 (format string)

下限和格式字符串由一个 # 符号分隔，对与对之间由符号 | 分隔。

例如，

```
{1,choice,0#no houses|1#one house|2#{1} houses}
```

表 7-8 显示了格式字符串对 {1} 的不同值产生的作用。

表 7-8 由选择格式进行格式化的字符串

{1}	结果	{1}	结果
0	"no houses"	3	"3 houses"
1	"one house"	-1	"no houses"

为什么在格式字符串中两次用到了 {1}？当消息格式将选择格式应用于占位符 {1} 而且替換值是 2 时，选择格式会返回 "{1} houses"。这个字符串由消息格式再次格式化，并将这次的结果和上一次的叠加。

注释：这个例子说明选择格式的设计者有些糊涂了。如果你有 3 个格式字符串，就需要两个下限来分隔它们。一般来说，你需要的下限数目比格式字符串数目少 1。就像你在表 7-8 中见到的，`MessageFormat` 类将忽略第一个下限。

如果这个类的设计者意识到下限只在两个选择之间出现，那么语法就要清楚得多，比如，

```
no houses|1|one house|2|{1} houses // not the actual format
```

可以使用 < 符号来表示如果替换值严格小于下限，则选中这个选择项。

也可以使用 \leq (Unicode 中的代码是 \u2264) 来实现和 # 相同的效果。如果愿意的话，甚至可以将第一个下限的值定义为 $-\infty$ (Unicode 代码是 -\u221E)。

例如，

```
-\infty<no houses|0<one house|2\leq{1} houses
```

或者使用 Unicode 转义字符，

```
-\u221E<no houses|0<one house|2\u2264{1} houses
```

让我们来结束自然灾害的场景。如果我们将选择字符串放到原始消息字符串中，那么会得到下面的格式化指令：

```
String pattern = "On {2,date,long}, {0} destroyed {1,choice,0#no houses|1#one house|2#{1} houses}" + "and caused {3,number,currency} of damage.;"
```

在德语中，即

```
String pattern
= "{0} zerstörte am {2,date,long} {1,choice,0#kein Haus|1#ein Haus|2#{1} Häuser}"
+ "und richtete einen Schaden von {3,number,currency} an.;"
```

请注意，在德语中词的顺序和英语中是不同的，但是你传给 `format` 方法的对象数组是相同的。可以用格式字符串中占位符的顺序来处理单词顺序的改变。

7.6 文本输入和输出

众所周知，Java 编程语言自身是完全基于 Unicode 的。但是，Windows 和 Mac OS X 仍旧支持遗留的字符编码机制，例如西欧国家的 Windows-1252 和 Mac Roman。因此，与用户通过文本沟通并非看上去那么简单。下面各节将讨论你可能会碰到的各种复杂情况。

7.6.1 文本文件

当今最好是使用 UTF-8 来存储和加载文本文件，但是你可能需要操作遗留文件。如果你知道遗留文件所希望使用的字符编码机制，那么可以在读写文本文件时指定它：

```
var out = new PrintWriter(filename, "Windows-1252");
```

如果想要获得可用的最佳编码机制，可以通过下面的调用来获得“平台的编码机制”：

```
Charset platformEncoding = Charset.defaultCharset();
```

7.6.2 行结束符

这不是 `locale` 的问题，而是平台的问题。在 Windows 中，文本文件希望在每行末尾使用 \r\n，而基于 UNIX 的系统只需要一个 \n 字符。当今，大多数 Windows 程序都可以处理只有一个 \n 的情况，一个重要的例外是记事本。如果“用户可以在你的应用所产生的文本文件上

双击并在记事本中浏览它”对你来说非常重要，那么你就要确保该文本文件使用了正确的行结束符。

任何用 `println` 方法写入的行都将是被正确终止的。唯一的问题是你是否打印了包含 `\n` 字符的行。它们不会被自动修改为平台的行结束符。

与在字符串中使用 `\n` 不同，可以使用 `printf` 和 `%n` 格式说明符来产生平台相关的行结束符。例如，

```
out.printf("Hello%nWorld%n");
```

会在 Windows 上产生

```
Hello\r\nWorld\r\n
```

而在其他所有平台上产生

```
Hello\nWorld\n
```

7.6.3 控制台

如果你编写的程序是通过 `System.in/System.out` 或 `System.console()` 与用户交互的，那么就不得不面对控制台使用的字符编码机制与 `Charset.defaultCharset()` 报告的平台编码机制有差异的可能性。当使用 Windows 上的 cmd 工具时，这个问题尤其需要注意。在美国版本的 Windows 10 中，命令行 Shell 使用的是陈旧的 IBM437 编码机制，它源自 1982 年 IBM 的个人计算机。没有任何官方的 API 可以揭示该信息。`Charset.defaultCharset()` 方法将返回 Windows-1252 字符集，它与 IBM437 完全不同。例如，在 Windows-1252 中有欧元符号 €，但是在 IBM437 中没有。如果调用

```
System.out.println("100 €");
```

控制台会显示

```
100 ?
```

你可以建议用户切换控制台的字符编码机制。在 Windows 中，这可以通过 chcp 命令实现。例如：

```
chcp 1252
```

会将控制台变换为 Windows-1252 编码页。

当然，理想情况下你的用户应该将控制台切换到 UTF-8。在 Windows 中，该命令为

```
chcp 65001
```

遗憾的是，这种命令还不足以让 Java 在控制台中使用 UTF-8，我们还必须使用非官方的 `file.encoding` 系统属性来设置平台的编码机制：

```
java -Dfile.encoding=UTF-8 MyProg
```

7.6.4 日志文件

当来自 `java.util.logging` 库的日志消息被发送到控制台时，它们会用控制台的编码机制来

书写。在上一节中你看到了如何进行控制。但是，文件中的日志消息会使用 `FileHandler` 来处理，它在默认情况下使用平台的编码机制。

要想将编码机制修改为 UTF-8，需要修改日志管理器的设置。具体做法是在日志配置文件中做如下设置：

```
java.util.logging.FileHandler.encoding=UTF-8
```

7.6.5 UTF-8 字节顺序标志

正如我们已经提到的，尽可能地让文本文件使用 UTF-8 是一个好的做法。如果你的应用必须读取其他程序创建的 UTF-8 文本文件，那么你可能会碰到另一个问题。在文件中添加一个“字节顺序标志”字符 U+FEFF 作为文件的第一个字符，是一种完全合法的做法。在 UTF-16 编码机制中，每个码元都是一个两字节的数字，字节顺序标志可以告诉读入器该文件使用的是“高字节在前”还是“低字节在前”的字节顺序。UTF-8 是一种单字节编码机制，因此不需要指定字节的顺序。但是如果一个文件以字节 0xEF 0xBB 0xBF (U+FEFF 的 UTF-8 编码) 开头，那么这就是一个强烈暗示，表示该文件使用了 UTF-8。正是这个原因，Unicode 标准鼓励这种实践方式。任何读入器都被认为会丢弃最前面的字节顺序标志。

还有一个美中不足的瑕疵。Oracle 的 Java 实现很固执地因潜在的兼容性问题而拒绝遵循 Unicode 标准。作为程序员，这对你而言意味着必须去执行平台并不会执行的操作。在读入文本文件时，如果开头碰到了 U+FEFF，那就需要忽略它。

! **警告：**遗憾的是，JDK 的实现没有遵循这项建议。在向 `javac` 编译器传递有效的以字节顺序标志开头的 UTF-8 源文件时，编译会以产生错误消息 “illegal character: \65279” 而失败。

7.6.6 源文件的字符编码

作为程序员，要牢记你需要与 Java 编译器交互，这种交互需要通过本地系统的工具来完成。例如，可以使用中文版的记事本来写你的 Java 源代码文件。但这样写出来的源码不是随处可用的，因为它们使用的是本地的字符编码。只有编译后的 `class` 文件才能随处使用，因为它们会自动地使用“modified UTF-8”编码来处理标识符和字符串。这意味着即使在程序编译和运行时，也涉及 3 种字符编码：

- 源文件：平台编码
- 类文件：modified UTF-8
- 虚拟机：UTF-16

关于 modified UTF-8 和 UTF-16 格式的定义，参见第 1 章。

! **提示：**可以用 `-encoding` 标记来设定源文件的字符编码，例如：

```
javac -encoding UTF-8 Myfile.java
```

7.7 资源包

当本地化一个应用时，可能会有大量的消息字符串、按钮标签和其他的东西需要被翻译。为了能灵活地完成这项任务，你肯定希望在外部定义消息字符串，这些消息字符串通常被称为资源（resource）。这样，翻译人员不需要接触程序源代码就可以很容易地编辑资源文件。

在 Java 中，要使用属性文件来设定字符串资源，并为其他类型的资源实现相应的类。

注释：Java 技术资源与 Windows 或 Macintosh 资源不同。Macintosh 或 Windows 可执行文件在程序代码以外的地方存储类似菜单、对话框、图标和消息这样的资源。资源编辑器能够在不影响程序代码的情况下检查并更新这些资源。

注释：卷 I 第 5 章描述了 JAR 文件资源的概念，以及为何数据文件、声音和图片可以存放在 JAR 文件中。Class 类的 `getResource` 方法可以找到相应的文件，打开它并返回资源的 URL。通过将文件放置到 JAR 文件中，将查找这些资源文件的工作留给了类的加载器去处理，加载器知道如何定位 JAR 文件中的项。但是，这种机制不支持 `locale`。

7.7.1 定位资源包

当本地化一个应用时，会产生很多资源包（resource bundle）。每一个包都是一个属性文件或者是一个描述了与 `locale` 相关的项的类（比如消息、标签等）。对于每一个包，都要为所有你想要支持的 `locale` 提供相应的版本。

需要对这些包使用一种统一的命名规则。例如，为德国定义的资源放在一个名为“`baseName_de_DE`”的文件中，而所有说德语的国家所共享的资源则放在名为“`baseName_de`”的文件中。一般来说，使用

`baseName_language_country`

来命名所有和国家相关的资源，使用

`baseName_language`

来命名所有和语言相关的资源。最后，作为后备，可以把默认资源放到一个没有后缀的文件中。

可以用下面的命令加载一个包：

```
 ResourceBundle currentResources = ResourceBundle.getBundle(baseName, currentLocale);
```

`getBundle` 方法试图加载匹配当前 `locale` 定义的语言和国家的包。如果失败，通过依次放弃国家和语言来继续进行查找，然后同样的查找被应用于默认的 `locale`，最后，如果还不行的话就去查看默认的包文件，如果这也失败了，则抛出一个 `MissingResourceException` 异常。

这就是说，`getBundle` 方法会试图加载以下包：

```
baseName_currentLocaleLanguage_currentLocaleCountry
baseName_currentLocaleLanguage
baseName_currentLocaleLanguage_defaultLocaleCountry
baseName_defaultLocaleLanguage
baseName
```

一旦 `getBundle` 方法定位了一个包，比如，`baseName_de_DE`，它还会继续查找 `baseName_de` 和 `baseName` 这两个包。如果这些包也存在，它们在资源层次中就成为 `baseName_de_DE` 的父包。以后，当查找一个资源时，如果在当前包中没有找到，就去查找其父包。就是说，如果一个特定的资源在当前包中没有找到，比如，某个特定资源在 `baseName_de_DE` 中没有找到，那么就会去查找 `baseName_de` 和 `baseName`。

这是一项非常有用的服务，如果手工来编写将会非常麻烦。Java 编程语言的资源包机制会自动定位与给定的 locale 匹配得最好的项。可以很容易地把越来越多的本地化信息加到已有的程序中：你需要做的只是增加额外的资源包。

 **注释：**我们简化了对资源包查找的讨论。如果 `locale` 中包含脚本或变体，那么查找就会复杂得多。可以查看 `ResourceBundle.Control.getCandidateLocales` 方法的文档以了解其细节。

 **提示：**不需要把你的程序的所有资源都放到同一个包中。可以用一个包来存放按钮标签，用另一个包存放错误消息等。

7.7.2 属性文件

对字符串进行国际化是很直接的，可以把所有字符串放到一个属性文件中，比如 `MyProgramStrings.properties`，这是一个每行存放一个键 - 值对的文本文件。典型的属性文件看起来像下面这样：

```
computeButton=Rechnen
colorName=black
defaultPaperSize=210×297
```

然后像上一节描述的那样命名属性文件，例如，

```
MyProgramStrings.properties
MyProgramStrings_en.properties
MyProgramStrings_de_DE.properties
```

可以加载包，例如：

```
 ResourceBundle bundle = ResourceBundle.getBundle("MyProgramStrings", locale);
```

要查找一个具体的字符串，可以调用

```
String computeButtonLabel = bundle.getString("computeButton");
```

 **警告：**在 Java 9 之前，存储属性的文件都是 ASCII 文件。如果你使用的是旧版本的 Java，并且需要将 Unicode 字符放到属性文件中，那么请用 \uxxxx 编码方式对它们进

行编码。比如，要设定 "colorName=Grün"，可以使用

```
colorName=Gr\u00fcn
```

你可以使用 native2ascii 工具来产生这些文件。

7.7.3 包类

为了提供字符串以外的资源，需要定义类，它必须扩展自 `ResourceBundle` 类。应该使用标准的命名规则来命名你的类，比如

```
MyProgramResources.java  
MyProgramResources_en.java  
MyProgramResources_de_DE.java
```

可以使用与加载属性文件相同的 `getBundle` 方法来加载这个类：

```
 ResourceBundle bundle = ResourceBundle.getBundle("MyProgramResources", locale);
```

◆ 警告：当搜索包时，如果类中的包和属性文件中的包都存在匹配，则优先选择类中的包。

每一个资源包类都实现了一个查询表。你需要为每一个你想定位的设置提供一个关键字字符串，使用这个字符串来提取相应的设置。例如，

```
var backgroundColor = (Color) bundle.getObject("backgroundColor");
double[] paperSize = (double[]) bundle.getObject("defaultPaperSize");
```

实现资源包类的最简单方法就是继承 `ListResourceBundle` 类。`ListResourceBundle` 类让你把所有资源都放到一个对象数组中并提供查找功能。要遵循以下的代码框架：

```
public class baseName_language_country extends ListResourceBundle
{
    private static final Object[][] contents =
    {
        { key1, value2 },
        { key2, value2 },
        ...
    }
    public Object[][] getContents() { return contents; }
}
```

例如，

```
public class ProgramResources_de extends ListResourceBundle
{
    private static final Object[][] contents =
    {
        { "backgroundColor", Color.black },
        { "defaultPaperSize", new double[] { 210, 297 } }
    }
    public Object[][] getContents() { return contents; }
}
```

```

public class ProgramResources_en_US extends ListResourceBundle
{
    private static final Object[][] contents =
    {
        { "backgroundColor", Color.blue },
        { "defaultPageSize", new double[] { 216, 279 } }
    }
    public Object[][] getContents() { return contents; }
}

```

注释：纸的尺寸是以毫米为单位给出的。在世界上，除了加拿大和美国，其他地区都使用 ISO 216 规格的纸。更多信息见 <http://www.cl.cam.ac.uk/~mgk25/iso-paper.html>。

或者，你的资源包类可以继承 ResourceBundle 类。然后需要实现两个方法，一是枚举所有键，二是用给定的键查找相应的值：

```

Enumeration<String> getKeys()
Object handleGetObject(String key)

```

ResourceBundle 类的 getObject 方法会调用你提供的 handleGetObject 方法。

API java.util.ResourceBundle 1.1

- static ResourceBundle getBundle(String baseName, Locale loc)
- static ResourceBundle getBundle(String baseName)

在给定或默认的 locale 下以给定的名字加载资源包类和它的父类。如果资源包类位于一个 Java 包中，那么类的名字必须包含完整的包名，例如 "intl.ProgramResources"。资源包类必须是 public 的，这样 getBundle 方法才能访问它们。

- Object getObject(String name)

从资源包或它的父包中查找一个对象。

- String getString(String name)

从资源包或它的父包中查找一个对象并把它转型成字符串。

- String[] getStringArray(String name)

从资源包或它的父包中查找一个对象并把它转型成字符串数组。

- Enumeration<String> getKeys()

返回一个枚举对象，枚举出资源包中的所有键，也包括父包中的键。

- Object handleGetObject(String key)

如果你要定义自己的资源查找机制，那么这个方法就需要被覆写，用来查找与给定的键相关联的资源的值。

7.8 一个完整的例子

在这一节中，我们使用本章中的内容来对退休金计算器小程序进行本地化，这个小程序

可以计算你是否为退休存够了钱。你需要输入年龄，每个月存多少钱等信息（参见图 7-4）。

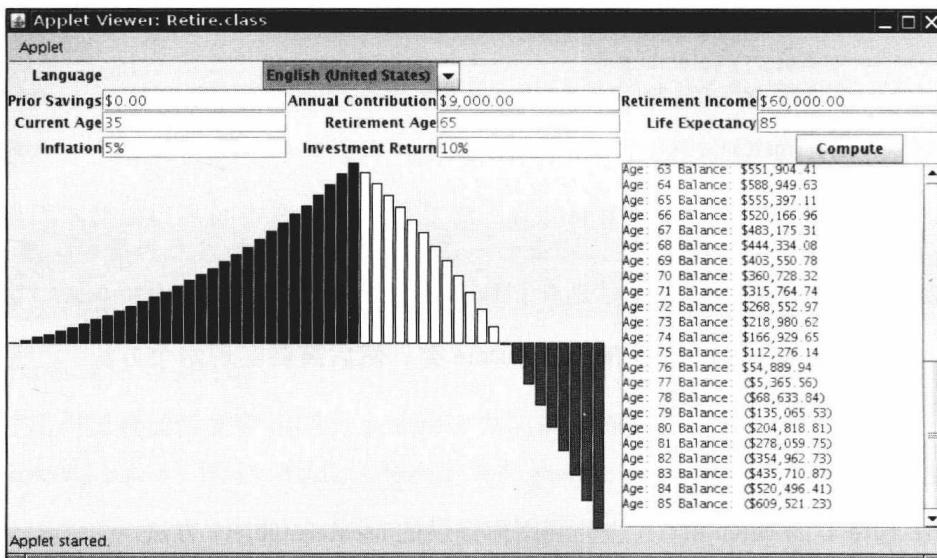


图 7-4 使用英语的退休金计算器

文本域和图表显示每年退休金账户中的余额。如果你后半生的退休金余额变成负数，并且表中的数据条在 x 轴以下，你就需要做些什么了。例如，存更多的钱、推迟退休等。

这个退休金计算器可以在三种 locale（英语、德语和中文）下工作。下面是进行国际化时的一些要点：

- 标签、按钮和消息被翻译成德语和中文。你可以在 RetireResources_de 和 RetireResources_zh 中找到它们。英语作为后备，见 RetireResources 文件。
- 当 locale 改变时，我们重置标签并格式化文本域中的内容。
- 文本域以本地格式处理数字、货币值和百分数。
- 计算域使用了 MessageFormat。格式字符串被存储在每种语言的资源包中。
- 为了使展示的确可行，我们按照用户选择的语言为条形图使用不同的颜色。

程序清单 7-5 到程序清单 7-8 展示了代码，而程序清单 7-9 到程序清单 7-11 是本地化的字符串的属性文件。图 7-5 和图 7-6 分别显示了在德语和中文下的输出。为了显示中文，请确认你已经在 Java 运行环境中安装并配置了中文字体。否则，所有的中文字符将会显示“missing character”图标。

程序清单 7-5 retire/Retire.java

```

1 package retire;
2
3 import java.awt.*;
4 import java.awt.geom.*;
5 import java.text.*;

```

```
6 import java.util.*;
7
8 import javax.swing.*;
9
10 /**
11 * This program shows a retirement calculator. The UI is displayed in English, German, and
12 * Chinese.
13 * @version 1.25 2018-05-01
14 * @author Cay Horstmann
15 */
16 public class Retire
17 {
18     public static void main(String[] args)
19     {
20         EventQueue.invokeLater(() ->
21         {
22             var frame = new RetireFrame();
23             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24             frame.setVisible(true);
25         });
26     }
27 }
28
29 class RetireFrame extends JFrame
30 {
31     private JTextField savingsField = new JTextField(10);
32     private JTextField contribField = new JTextField(10);
33     private JTextField incomeField = new JTextField(10);
34     private JTextField currentAgeField = new JTextField(4);
35     private JTextField retireAgeField = new JTextField(4);
36     private JTextField deathAgeField = new JTextField(4);
37     private JTextField inflationPercentField = new JTextField(6);
38     private JTextField investPercentField = new JTextField(6);
39     private JTextArea retireText = new JTextArea(10, 25);
40     private RetireComponent retireCanvas = new RetireComponent();
41     private JButton computeButton = new JButton();
42     private JLabel languageLabel = new JLabel();
43     private JLabel savingsLabel = new JLabel();
44     private JLabel contribLabel = new JLabel();
45     private JLabel incomeLabel = new JLabel();
46     private JLabel currentAgeLabel = new JLabel();
47     private JLabel retireAgeLabel = new JLabel();
48     private JLabel deathAgeLabel = new JLabel();
49     private JLabel inflationPercentLabel = new JLabel();
50     private JLabel investPercentLabel = new JLabel();
51     private RetireInfo info = new RetireInfo();
52     private Locale[] locales = { Locale.US, Locale.CHINA, Locale.GERMANY };
53     private Locale currentLocale;
54     private JComboBox<Locale> localeCombo = new LocaleCombo(locales);
55     private ResourceBundle res;
56     private ResourceBundle resStrings;
57     private NumberFormat currencyFmt;
58     private NumberFormat numberFmt;
59     private NumberFormat percentFmt;
```

```

60
61     public RetireFrame()
62     {
63         setLayout(new GridBagLayout());
64         add(languageLabel, new GBC(0, 0).setAnchor(GBC.EAST));
65         add(savingsLabel, new GBC(0, 1).setAnchor(GBC.EAST));
66         add(contribLabel, new GBC(2, 1).setAnchor(GBC.EAST));
67         add(incomeLabel, new GBC(4, 1).setAnchor(GBC.EAST));
68         add(currentAgeLabel, new GBC(0, 2).setAnchor(GBC.EAST));
69         add(retireAgeLabel, new GBC(2, 2).setAnchor(GBC.EAST));
70         add(deathAgeLabel, new GBC(4, 2).setAnchor(GBC.EAST));
71         add(inflationPercentLabel, new GBC(0, 3).setAnchor(GBC.EAST));
72         add(investPercentLabel, new GBC(2, 3).setAnchor(GBC.EAST));
73         add(localeCombo, new GBC(1, 0, 3, 1));
74         add(savingsField, new GBC(1, 1).setWeight(100, 0).setFill(GBC.HORIZONTAL));
75         add(contribField, new GBC(3, 1).setWeight(100, 0).setFill(GBC.HORIZONTAL));
76         add(incomeField, new GBC(5, 1).setWeight(100, 0).setFill(GBC.HORIZONTAL));
77         add(currentAgeField, new GBC(1, 2).setWeight(100, 0).setFill(GBC.HORIZONTAL));
78         add(retireAgeField, new GBC(3, 2).setWeight(100, 0).setFill(GBC.HORIZONTAL));
79         add(deathAgeField, new GBC(5, 2).setWeight(100, 0).setFill(GBC.HORIZONTAL));
80         add(inflationPercentField, new GBC(1, 3).setWeight(100, 0).setFill(GBC.HORIZONTAL));
81         add(investPercentField, new GBC(3, 3).setWeight(100, 0).setFill(GBC.HORIZONTAL));
82         add(retireCanvas, new GBC(0, 4, 4, 1).setWeight(100, 100).setFill(GBC.BOTH));
83         add(new JScrollPane(retireText),
84             new GBC(4, 4, 2, 1).setWeight(0, 100).setFill(GBC.BOTH));
85
86         computeButton.setName("computeButton");
87         computeButton.addActionListener(event ->
88         {
89             getInfo();
90             updateData();
91             updateGraph();
92         });
93         add(computeButton, new GBC(5, 3));
94
95         retireText.setEditable(false);
96         retireText.setFont(new Font("Monospaced", Font.PLAIN, 10));
97
98         info.setSavings(0);
99         info.setContrib(9000);
100        info.setIncome(60000);
101        info.setCurrentAge(35);
102        info.setRetireAge(65);
103        info.setDeathAge(85);
104        info.setInvestPercent(0.1);
105        info.setInflationPercent(0.05);
106
107        int localeIndex = 0; // US locale is default selection
108        for (int i = 0; i < locales.length; i++)
109            // if current locale one of the choices, select it
110            if (getLocale().equals(locales[i])) localeIndex = i;
111        setCurrentLocale(locales[localeIndex]);
112
113        localeCombo.addActionListener(event ->
114        {

```

```
115         setCurrentLocale((Locale) localeCombo.getSelectedItem());
116         validate();
117     });
118     pack();
119 }
120
121 /**
122 * Sets the current locale.
123 * @param locale the desired locale
124 */
125 public void setCurrentLocale(Locale locale)
126 {
127     currentLocale = locale;
128     localeCombo.setLocale(currentLocale);
129     localeCombo.setSelectedItem(currentLocale);
130
131     res = ResourceBundle.getBundle("retire.RetireResources", currentLocale);
132     resStrings = ResourceBundle.getBundle("retire.RetireStrings", currentLocale);
133     currencyFmt = NumberFormat.getCurrencyInstance(currentLocale);
134     numberFmt = NumberFormat.getNumberInstance(currentLocale);
135     percentFmt = NumberFormat.getPercentInstance(currentLocale);
136
137     updateDisplay();
138     updateInfo();
139     updateData();
140     updateGraph();
141 }
142
143 /**
144 * Updates all labels in the display.
145 */
146 public void updateDisplay()
147 {
148     languageLabel.setText(resStrings.getString("language"));
149     savingsLabel.setText(resStrings.getString("savings"));
150     contribLabel.setText(resStrings.getString("contrib"));
151     incomeLabel.setText(resStrings.getString("income"));
152     currentAgeLabel.setText(resStrings.getString("currentAge"));
153     retireAgeLabel.setText(resStrings.getString("retireAge"));
154     deathAgeLabel.setText(resStrings.getString("deathAge"));
155     inflationPercentLabel.setText(resStrings.getString("inflationPercent"));
156     investPercentLabel.setText(resStrings.getString("investPercent"));
157     computeButton.setText(resStrings.getString("computeButton"));
158 }
159
160 /**
161 * Updates the information in the text fields.
162 */
163 public void updateInfo()
164 {
165     savingsField.setText(currencyFmt.format(info.getSavings()));
166     contribField.setText(currencyFmt.format(info.getContrib()));
167     incomeField.setText(currencyFmt.format(info.getIncome()));
168     currentAgeField.setText(numberFmt.format(info.getCurrentAge()));
```

```

169     retireAgeField.setText(numberFmt.format(info.getRetireAge()));
170     deathAgeField.setText(numberFmt.format(info.getDeathAge()));
171     investPercentField.setText(percentFmt.format(info.getInvestPercent()));
172     inflationPercentField.setText(percentFmt.format(info.getInflationPercent()));
173 }
174
175 /**
176 * Updates the data displayed in the text area.
177 */
178 public void updateData()
179 {
180     retireText.setText("");
181     var retireMsg = new MessageFormat("");
182     retireMsg.setLocale(currentLocale);
183     retireMsg.applyPattern(resStrings.getString("retire"));
184
185     for (int i = info.getCurrentAge(); i <= info.getDeathAge(); i++)
186     {
187         Object[] args = { i, info.getBalance(i) };
188         retireText.append(retireMsg.format(args) + "\n");
189     }
190 }
191
192 /**
193 * Updates the graph.
194 */
195 public void updateGraph()
196 {
197     retireCanvas.setColorPre((Color) res.getObject("colorPre"));
198     retireCanvas.setColorGain((Color) res.getObject("colorGain"));
199     retireCanvas.setColorLoss((Color) res.getObject("colorLoss"));
200     retireCanvas.setInfo(info);
201     repaint();
202 }
203
204 /**
205 * Reads the user input from the text fields.
206 */
207 public void getInfo()
208 {
209     try
210     {
211         info.setSavings(currencyFmt.parse(savingsField.getText()).doubleValue());
212         info.setContrib(currencyFmt.parse(contribField.getText()).doubleValue());
213         info.setIncome(currencyFmt.parse(incomeField.getText()).doubleValue());
214         info.setCurrentAge(numberFmt.parse(currentAgeField.getText()).intValue());
215         info.setRetireAge(numberFmt.parse(retireAgeField.getText()).intValue());
216         info.setDeathAge(numberFmt.parse(deathAgeField.getText()).intValue());
217         info.setInvestPercent(percentFmt.parse(investPercentField.getText()).doubleValue());
218         info.setInflationPercent(
219             percentFmt.parse(inflationPercentField.getText()).doubleValue());
220     }
221     catch (ParseException ex)
222     {

```

```
223         ex.printStackTrace();
224     }
225 }
226 }
227 /**
228 * The information required to compute retirement income data.
229 */
230
231 class RetireInfo
232 {
233     private double savings;
234     private double contrib;
235     private double income;
236     private int currentAge;
237     private int retireAge;
238     private int deathAge;
239     private double inflationPercent;
240     private double investPercent;
241     private int age;
242     private double balance;
243
244 /**
245 * Gets the available balance for a given year.
246 * @param year the year for which to compute the balance
247 * @return the amount of money available (or required) in that year
248 */
249 public double getBalance(int year)
250 {
251     if (year < currentAge) return 0;
252     else if (year == currentAge)
253     {
254         age = year;
255         balance = savings;
256         return balance;
257     }
258     else if (year == age) return balance;
259     if (year != age + 1) getBalance(year - 1);
260     age = year;
261     if (age < retireAge) balance += contrib;
262     else balance -= income;
263     balance = balance * (1 + (investPercent - inflationPercent));
264     return balance;
265 }
266
267 /**
268 * Gets the amount of prior savings.
269 * @return the savings amount
270 */
271 public double getSavings()
272 {
273     return savings;
274 }
275
276 /**
```

```
277     * Sets the amount of prior savings.  
278     * @param newValue the savings amount  
279     */  
280     public void setSavings(double newValue)  
281     {  
282         savings = newValue;  
283     }  
284  
285     /**  
286     * Gets the annual contribution to the retirement account.  
287     * @return the contribution amount  
288     */  
289     public double getContrib()  
290     {  
291         return contrib;  
292     }  
293  
294     /**  
295     * Sets the annual contribution to the retirement account.  
296     * @param newValue the contribution amount  
297     */  
298     public void setContrib(double newValue)  
299     {  
300         contrib = newValue;  
301     }  
302  
303     /**  
304     * Gets the annual income.  
305     * @return the income amount  
306     */  
307     public double getIncome()  
308     {  
309         return income;  
310     }  
311  
312     /**  
313     * Sets the annual income.  
314     * @param newValue the income amount  
315     */  
316     public void setIncome(double newValue)  
317     {  
318         income = newValue;  
319     }  
320  
321     /**  
322     * Gets the current age.  
323     * @return the age  
324     */  
325     public int getCurrentAge()  
326     {  
327         return currentAge;  
328     }  
329  
330     /**
```

```
331     * Sets the current age.  
332     * @param newValue the age  
333     */  
334     public void setCurrentAge(int newValue)  
335     {  
336         currentAge = newValue;  
337     }  
338  
339     /**  
340      * Gets the desired retirement age.  
341      * @return the age  
342      */  
343     public int getRetireAge()  
344     {  
345         return retireAge;  
346     }  
347  
348     /**  
349      * Sets the desired retirement age.  
350      * @param newValue the age  
351      */  
352     public void setRetireAge(int newValue)  
353     {  
354         retireAge = newValue;  
355     }  
356  
357     /**  
358      * Gets the expected age of death.  
359      * @return the age  
360      */  
361     public int getDeathAge()  
362     {  
363         return deathAge;  
364     }  
365  
366     /**  
367      * Sets the expected age of death.  
368      * @param newValue the age  
369      */  
370     public void setDeathAge(int newValue)  
371     {  
372         deathAge = newValue;  
373     }  
374  
375     /**  
376      * Gets the estimated percentage of inflation.  
377      * @return the percentage  
378      */  
379     public double getInflationPercent()  
380     {  
381         return inflationPercent;  
382     }  
383  
384     /**  
385      * Sets the estimated percentage of inflation.
```

```
386     * @param newValue the percentage
387     */
388     public void setInflationPercent(double newValue)
389     {
390         inflationPercent = newValue;
391     }
392
393     /**
394      * Gets the estimated yield of the investment.
395      * @return the percentage
396      */
397     public double getInvestPercent()
398     {
399         return investPercent;
400     }
401
402     /**
403      * Sets the estimated yield of the investment.
404      * @param newValue the percentage
405      */
406     public void setInvestPercent(double newValue)
407     {
408         investPercent = newValue;
409     }
410 }
411
412 /**
413  * This component draws a graph of the investment result.
414 */
415 class RetireComponent extends JComponent
416 {
417     private static final int PANEL_WIDTH = 400;
418     private static final int PANEL_HEIGHT = 200;
419     private static final Dimension PREFERRED_SIZE = new Dimension(800, 600);
420     private RetireInfo info = null;
421     private Color colorPre;
422     private Color colorGain;
423     private Color colorLoss;
424
425     public RetireComponent()
426     {
427         setSize(PANEL_WIDTH, PANEL_HEIGHT);
428     }
429
430     /**
431      * Sets the retirement information to be plotted.
432      * @param newInfo the new retirement info.
433      */
434     public void setInfo(RetireInfo newInfo)
435     {
436         info = newInfo;
437         repaint();
438     }
439
440     public void paintComponent(Graphics g)
```

```
441  {
442      var g2 = (Graphics2D) g;
443      if (info == null) return;
444
445      double minValue = 0;
446      double maxValue = 0;
447      int i;
448      for (i = info.getCurrentAge(); i <= info.getDeathAge(); i++)
449      {
450          double v = info.getBalance(i);
451          if (minValue > v) minValue = v;
452          if (maxValue < v) maxValue = v;
453      }
454      if (maxValue == minValue) return;
455
456      int barWidth = getWidth() / (info.getDeathAge() - info.getCurrentAge() + 1);
457      double scale = getHeight() / (maxValue - minValue);
458
459      for (i = info.getCurrentAge(); i <= info.getDeathAge(); i++)
460      {
461          int x1 = (i - info.getCurrentAge()) * barWidth + 1;
462          int y1;
463          double v = info.getBalance(i);
464          int height;
465          int y0origin = (int) (maxValue * scale);
466
467          if (v >= 0)
468          {
469              y1 = (int) ((maxValue - v) * scale);
470              height = y0origin - y1;
471          }
472          else
473          {
474              y1 = y0origin;
475              height = (int) (-v * scale);
476          }
477
478          if (i < info.getRetireAge()) g2.setPaint(colorPre);
479          else if (v >= 0) g2.setPaint(colorGain);
480          else g2.setPaint(colorLoss);
481          var bar = new Rectangle2D.Double(x1, y1, barWidth - 2, height);
482          g2.fill(bar);
483          g2.setPaint(Color.black);
484          g2.draw(bar);
485      }
486  }
487
488 /**
489 * Sets the color to be used before retirement.
490 * @param color the desired color
491 */
492 public void setColorPre(Color color)
493 {
494     colorPre = color;
495     repaint();
}
```

```

496    }
497
498    /**
499     * Sets the color to be used after retirement while the account balance is positive.
500     * @param color the desired color
501     */
502    public void setColorGain(Color color)
503    {
504        colorGain = color;
505        repaint();
506    }
507
508    /**
509     * Sets the color to be used after retirement when the account balance is negative.
510     * @param color the desired color
511     */
512    public void setColorLoss(Color color)
513    {
514        colorLoss = color;
515        repaint();
516    }
517
518    public Dimension getPreferredSize() { return PREFERRED_SIZE; }
519 }

```

程序清单 7-6 retire/RetireResources.java

```

1 package retire;
2
3 import java.awt.*;
4
5 /**
6  * These are the English non-string resources for the retirement calculator.
7  * @version 1.21 2001-08-27
8  * @author Cay Horstmann
9  */
10 public class RetireResources extends java.util.ListResourceBundle
11 {
12     private static final Object[][] contents = {
13         // BEGIN LOCALIZE
14         { "colorPre", Color.blue }, { "colorGain", Color.white }, { "colorLoss", Color.red }
15         // END LOCALIZE
16     };
17
18     public Object[][] getContents()
19     {
20         return contents;
21     }
22 }

```

程序清单 7-7 retire/RetireResources_de.java

```

1 package retire;
2

```

```

3 import java.awt.*;
4
5 /**
6  * These are the German non-string resources for the retirement calculator.
7  * @version 1.21 2001-08-27
8  * @author Cay Horstmann
9  */
10 public class RetireResources_de extends java.util.ListResourceBundle
11 {
12     private static final Object[][][] contents = {
13         // BEGIN LOCALIZE
14             { "colorPre", Color.yellow }, { "colorGain", Color.black }, { "colorLoss", Color.red }
15         // END LOCALIZE
16     };
17
18     public Object[][][] getContents()
19     {
20         return contents;
21     }
22 }

```

程序清单 7-8 retire/RetireResources_zh.java

```

1 package retire;
2
3 import java.awt.*;
4
5 /**
6  * These are the Chinese non-string resources for the retirement calculator.
7  * @version 1.21 2001-08-27
8  * @author Cay Horstmann
9  */
10 public class RetireResources_zh extends java.util.ListResourceBundle
11 {
12     private static final Object[][][] contents = {
13         // BEGIN LOCALIZE
14             { "colorPre", Color.red }, { "colorGain", Color.blue }, { "colorLoss", Color.yellow }
15         // END LOCALIZE
16     };
17
18     public Object[][][] getContents()
19     {
20         return contents;
21     }
22 }

```

程序清单 7-9 retire/RetireStrings.properties

```

1 language=Language
2 computeButton=Compute
3 savings=Prior Savings
4 contrib=Annual Contribution
5 income=Retirement Income
6 currentAge=Current Age

```

```

7 retireAge=Retirement Age
8 deathAge=Life Expectancy
9 inflationPercent=Inflation
10 investPercent=Investment Return
11 retire=Age: {0,number} Balance: {1,number,currency}

```

程序清单 7-10 retire/RetireStrings_de.properties

```

1 language=Sprache
2 computeButton=Rechnen
3 savings=Vorherige Ersparnisse
4 contrib=Jährliche Einzahlung
5 income=Einkommen nach Ruhestand
6 currentAge=Jetziges Alter
7 retireAge=Ruhestandsalter
8 deathAge=Lebenserwartung
9 inflationPercent=Inflation
10 investPercent=Investitionsgewinn
11 retire=Alter: {0,number} Guthaben: {1,number,currency}

```

程序清单 7-11 retire/RetireStrings_zh.properties

```

1 language=语言
2 computeButton=计算
3 savings=既存
4 contrib=每年存金
5 income=退休收入
6 currentAge=现龄
7 retireAge=退休年龄
8 deathAge=预期寿命
9 inflationPercent=通货膨胀
10 investPercent=投资报酬
11 retire=年龄: {0,number} 总结: {1,number,currency}

```

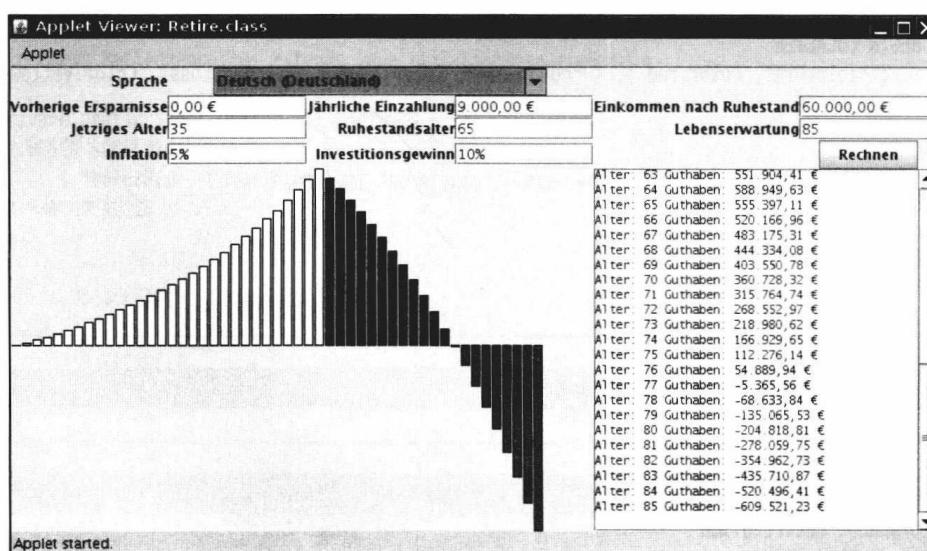


图 7-5 使用德语的退休金计算器

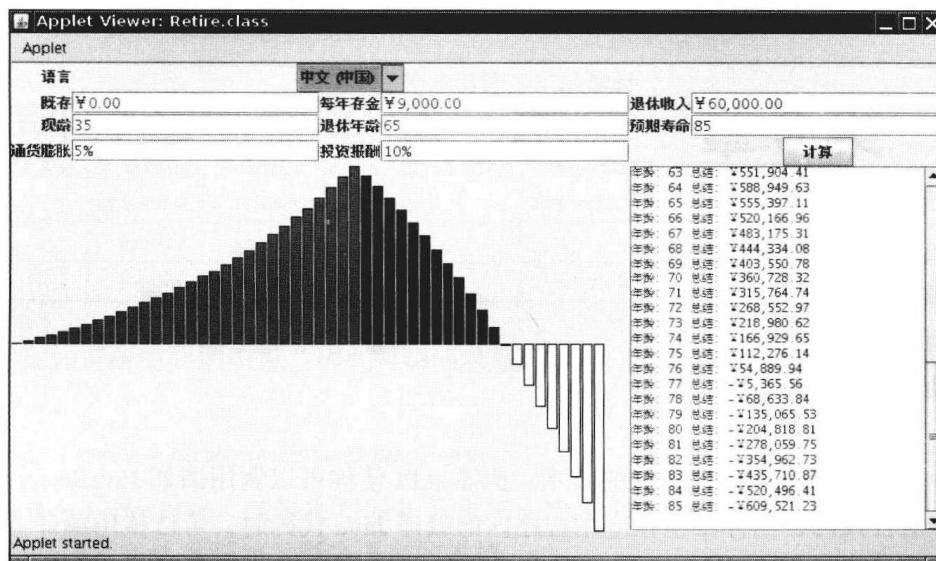


图 7-6 使用中文的退休金计算器

本章进述了如何运用 Java 语言的国际化特性。现在你可以使用资源包来提供多种语言的转换，也可以使用格式器和排序器来处理与 locale 相关的文本了。

下一章将研究脚本编写、编译和注解处理。

第8章 脚本、编译与注解处理

- ▲ Java 平台的脚本机制
- ▲ 编译器 API
- ▲ 使用注解
- ▲ 注解语法

- ▲ 标准注解
- ▲ 源码级注解处理
- ▲ 字节码工程

本章将介绍三种用于处理代码的技术：脚本 API 使你可以调用诸如 JavaScript 和 Groovy 这样的脚本语言代码；当你希望在应用程序内部编译 Java 代码时，可以使用编译器 API；注解处理器可以在包含注解的 Java 源代码和类文件上进行操作。如你所见，有许多应用程序都可以用来处理注解，从简单的诊断到“字节码工程”，后者可以将字节码插入到类文件中，甚至可以插入到运行程序中。

8.1 Java 平台的脚本机制

脚本语言是一种通过在运行时解释程序文本，从而避免使用通常的编辑 / 编译 / 链接 / 运行循环的语言。脚本语言有许多优势：

- 便于快速变更，鼓励不断试验。
- 可以修改运行着的程序的行为。
- 支持程序用户的定制化。

另一方面，大多数脚本语言都缺乏可以使编写复杂应用受益的特性，例如强类型、封装和模块化。

因此人们在尝试将脚本语言和传统语言的优势相结合。脚本 API 使你可以在 Java 平台上实现这个目的，它支持在 Java 程序中对用 JavaScript、Groovy、Ruby，甚至是更奇异的诸如 Scheme 和 Haskell 等语言编写的脚本进行调用。例如，Renjin 项目 (www.renjin.org) 就提供了一个 R 编程语言的 Java 实现和相应的脚本 API 的“引擎”，R 语言被广泛应用于统计编程中。

在下面的小节中，我们将向你展示如何为某种特定的语言选择一个引擎，如何执行脚本，以及如何利用某些脚本引擎提供的先进特性。

8.1.1 获取脚本引擎

脚本引擎是一个可以执行用某种特定语言编写的脚本的类库。当虚拟机启动时，它会发现可用的脚本引擎。为了枚举这些引擎，需要构造一个 `ScriptEngineManager`，并调用 `getEngineFactories` 方法。可以向每个引擎工厂询问它们所支持的引擎名、MIME 类型和文件

扩展名。表 8-1 显示了这些内容的典型值。

表 8-1 脚本引擎工厂的属性

引擎	名字	MIME 类型	文件扩展
Nashorn (包含在 JDK 中)	nashorn, Nashorn, js, JS, JavaScript, javascript, ECMAScript, ecmascript	application/javascript, application/ecmascript, text/javascript, text/ecmascript	.js
Groovy	groovy	无	.groovy
Renjin	Renjin	text/x-R	R, r, S, s

通常，你知道所需要的引擎，因此可以直接通过名字、MIME 类型或文件扩展来请求它，例如：

```
ScriptEngine engine = manager.getEngineByName("nashorn");
```

Java 8 引入了 Nashorn，这是由 Oracle 开发的一个 JavaScript 解释器。可以通过在类路径中提供必要的 JAR 文件来添加对更多语言的支持。

API javax.script.ScriptEngineManager 6

- List<ScriptEngineFactory> getEngineFactories()
获取所有发现的引擎工厂的列表。
- ScriptEngine getEngineByName(String name)
● ScriptEngine getEngineByExtension(String extension)
● ScriptEngine getEngineByMimeType(String mimeType)
获取给定名字、脚本文件扩展名或 MIME 类型的脚本引擎。

API javax.script.ScriptEngineFactory 6

- List<String> getNames()
- List<String> getExtensions()
- List<String> getMimeTypes()

获取该工厂所了解的名字、脚本文件扩展名和 MIME 类型。

8.1.2 脚本计算与绑定

一旦拥有了引擎，就可以通过下面的调用来直接调用脚本：

```
Object result = engine.eval(scriptString);
```

如果脚本存储在文件中，那么需要先打开一个 Reader，然后调用：

```
Object result = engine.eval(reader);
```

可以在同一个引擎上调用多个脚本。如果一个脚本定义了变量、函数或类，那么大多数引擎都会保留这些定义，以供将来使用。例如：

```
engine.eval("n = 1728");
Object result = engine.eval("n + 1");
```

将返回 1729。

注释：要想知道在多个线程中并发执行脚本是否安全，可以调用

```
Object param = factory.getParameter("THREADING");
```

其返回的是下列值之一：

- `null`: 并发执行不安全。
- `"MULTITHREADED"`: 并发执行安全。一个线程的执行效果对另外的线程有可能是可视的。
- `"THREAD-ISOLATED"`: 除了 `"MULTITHREADED"`, 还会为每个线程维护不同的变量绑定。
- `"STATELESS"`: 除了 `"THREAD-ISOLATED"`, 脚本还不会改变变量绑定。

我们经常希望能够向引擎中添加新的变量绑定。绑定由名字及其关联的 Java 对象构成。例如，考虑下面的语句：

```
engine.put("k", 1728);
Object result = engine.eval("k + 1");
```

脚本代码从“引擎作用域”中的绑定里读取 `k` 的定义。这一点非常重要，因为大多数脚本语言都可以访问 Java 对象，通常使用的是比 Java 语法更简单的语法。例如，

```
engine.put("b", new JButton());
engine.eval("b.text = 'Ok'");
```

反过来，也可以获取由脚本语句绑定的变量：

```
engine.eval("n = 1728");
Object result = engine.get("n");
```

除了引擎作用域之外，还有全局作用域。任何添加到 `ScriptEngineManager` 中的绑定对所有引擎都是可视的。

除了向引擎或全局作用域添加绑定之外，还可以将绑定收集到一个类型为 `Bindings` 的对象中，然后将其传递给 `eval` 方法：

```
Bindings scope = engine.createBindings();
scope.put("b", new JButton());
engine.eval(scriptString, scope);
```

如果绑定集不应该为了将来对 `eval` 方法的调用而持久化，那么这么做就很有用。

注释：你可能希望除了引擎作用域和全局作用域之外还有其他的作用域。例如，Web 容器可能需要请求作用域或会话作用域。但是，这需要你自己去解决。你需要实现一个类，它实现了 `ScriptContext` 接口，并管理着一个作用域集合。每个作用域都是由一个整数标识的，而且越小的数字应该越先被搜索。(标准类库提供了 `SimpleScriptContext` 类，但是它只能持有全局作用域和引擎作用域。)

API `javax.script.ScriptEngine` 6

- `Object eval(String script)`
- `Object eval(Reader reader)`

- Object eval(String script, Bindings bindings)
- Object eval(Reader reader, Bindings bindings)
对由字符串或读取器给定的脚本进行计算，并服从给定的绑定。
- Object get(String key)
- void put(String key, Object value)
在引擎作用域内获取或放置一个绑定。
- Bindings createBindings()
创建一个适合该引擎的空 Bindings 对象。

API *javax.script.ScriptEngineManager* 6

- Object get(String key)
- void put(String key, Object value)
在全局作用域内获取或放置一个绑定。

API *javax.script.Bindings* 6

- Object get(String key)
- void put(String key, Object value)
在由该 Bindings 对象表示的作用域内获取或放置一个绑定。

8.1.3 重定向输入和输出

可以通过调用脚本上下文的 `setReader` 和 `setWriter` 方法来重定向脚本的标准输入和输出。例如，

```
var writer = new StringWriter();
engine.getContext().setWriter(new PrintWriter(writer, true));
```

在上例中，任何用 JavaScript 的 `print` 和 `println` 函数产生的输出都会被发送到 `writer`。

`setReader` 和 `setWriter` 方法只会影响脚本引擎的标准输入和输出源。例如，如果执行下面的 JavaScript 代码：

```
println("Hello");
java.lang.System.out.println("World");
```

则只有第一个输出会被重定向。

Nashorn 引擎没有标准输入源的概念，因此调用 `setReader` 没有任何效果。

API *javax.script.ScriptEngine* 6

- ScriptContext getContext()
获得该引擎的默认的脚本上下文。

API *javax.script.ScriptContext* 6

- Reader getReader()

- void setReader(Reader reader)
- Writer getWriter()
- void setWriter(Writer writer)
- Writer getErrorWriter()
- void setErrorWriter(Writer writer)

获取或设置用于输入的读入器或用于正常与错误输出的写出器。

8.1.4 调用脚本的函数和方法

对于许多脚本引擎而言，我们都可以调用脚本语言的函数，而不必对实际的脚本代码进行计算。如果允许用户用他们所选择的脚本语言来实现服务，那么这种机制就很有用了。

提供这种功能的脚本引擎实现了 `Invocable` 接口。特别是，Nashorn 引擎就是实现了 `Invocable` 接口。

要调用一个函数，需要用函数名来调用 `invokeFunction` 方法，函数名后面是函数的参数：

```
// Define greet function in JavaScript
engine.eval("function greet(how, whom) { return how + ', ' + whom + '!' }");

// Call the function with arguments "Hello", "World"
result = ((Invocable) engine).invokeFunction("greet", "Hello", "World");
```

如果脚本语言是面向对象的，那就可以调用 `invokeMethod`：

```
// Define Greeter class in JavaScript
engine.eval("function Greeter(how) { this.how = how }");
engine.eval("Greeter.prototype.welcome =
    + " function(whom) { return this.how + ', ' + whom + '!' }");

// Construct an instance
Object yo = engine.eval("new Greeter('Yo')");

// Call the welcome method on the instance
result = ((Invocable) engine).invokeMethod(yo, "welcome", "World");
```

注释：关于如何用 JavaScript 定义类的更多细节，可以参阅 *JavaScript: The Good Parts*，Douglas Crockford 著 (O'Reilly, 2008)。

注释：即使脚本引擎没有实现 `Invocable` 接口，你也可能仍旧可以以一种独立于语言的方式来调用某个方法。`ScriptEngineFactory` 类的 `getMethodCallSyntax` 方法可以产生一个字符串，你可以将其传递给 `eval` 方法。但是，所有的方法参数必须都与名字绑定，而 `invokeMethod` 方法是可以用任意值调用的。

我们可以更进一步，让脚本引擎去实现一个 Java 接口，然后就可以用 Java 方法调用的语法来调用脚本函数。

其细节依赖于脚本引擎，但是典型情况是我们需要为该接口中的每个方法都提供一个函数。例如，考虑下面的 Java 接口：

```
public interface Greeter
{
    String welcome(String whom);
}
```

如果在 Nashorn 中定义了具有相同名字的函数，那么可通过这个接口来调用它：

```
// Define welcome function in JavaScript
engine.eval("function welcome(whom) { return 'Hello, ' + whom + '!' }");

// Get a Java object and call a Java method
Greeter g = ((Invocable) engine).getInterface(Greeter.class);
result = g.welcome("World");
```

在面向对象的脚本语言中，可以通过相匹配的 Java 接口来访问一个脚本类。例如，下面的代码展示了如何使用 Java 的语法来调用 JavaScript 的 SimpleGreeter 类：

```
Greeter g = ((Invocable) engine).getInterface(yo, Greeter.class);
result = g.welcome("World");
```

总之，如果你希望从 Java 中调用脚本代码，同时又不想因这种脚本语言的语法而受到困扰，那么 Invocable 接口就很有用。

API `javasCript.Invocable`

- `Object invokeFunction(String name, Object... parameters)`
- `Object invokeMethod(Object implicitParameter, String name, Object... explicitParameters)`
用给定的名字调用函数或方法，并传递给定的参数。
- `<T> T getInterface(Class<T> iface)`
返回给定接口的实现，该实现用脚本引擎中的函数实现了接口中的方法。
- `<T> T getInterface(Object implicitParameter, Class<T> iface)`
返回给定接口的实现，该实现用给定对象的方法实现了接口中的方法。

8.1.5 编译脚本

某些脚本引擎出于对执行效率的考虑，可以将脚本代码编译为某种中间格式。这些引擎实现了 Compilable 接口。下面的示例展示了如何编译和计算包含在脚本文件中的代码：

```
var reader = new FileReader("myscript.js");
CompiledScript script = null;
if (engine implements Compilable)
    script = ((Compilable) engine).compile(reader);
```

一旦该脚本被编译，就可以执行它。下面的代码将会在编译成功的情况下执行编译后的脚本，如果引擎不支持编译，则执行原始的脚本。

```
if (script != null)
    script.eval();
else
    engine.eval(reader);
```

当然，只有需要重复执行时，我们才希望编译脚本。

API **javax.script.Compilable** 6

- `CompiledScript compile(String script)`
 - `CompiledScript compile(Reader reader)`
- 编译由字符串或读入器给定的脚本。

API **javax.script.CompiledScript** 6

- `Object eval()`
 - `Object eval(Bindings bindings)`
- 对该脚本计算。

8.1.6 示例：用脚本处理 GUI 事件

为了演示脚本 API，我们将开发一个样例程序，它允许用户指定使用他们所选择的脚本语言编写的事件处理器。

让我们看看程序清单 8-1 中的程序，它可以将脚本添加到任意的框体类中。默认情况下，它会读取程序清单 8-2 中的 `ButtonFrame` 类，`ButtonFrame` 类与卷 I 中介绍的事件处理演示程序类似，但是有两个差异：

- 每个构件都有其自己的 `name` 属性集。
- 没有任何事件处理器。

程序清单 8-1 script/ScriptTest.java

```

1 package script;
2
3 import java.awt.*;
4 import java.beans.*;
5 import java.io.*;
6 import java.lang.reflect.*;
7 import java.util.*;
8 import javax.script.*;
9 import javax.swing.*;
10
11 /**
12  * @version 1.03 2018-05-01
13  * @author Cay Horstmann
14  */
15 public class ScriptTest
16 {
17     public static void main(String[] args)
18     {
19         EventQueue.invokeLater(() ->
20         {
21             try
22             {
23                 var manager = new ScriptEngineManager();
24                 String language;
25                 if (args.length == 0)

```

```

26     {
27         System.out.println("Available factories: ");
28         for (ScriptEngineFactory factory : manager.getEngineFactories())
29             System.out.println(factory.getEngineName());
30
31         language = "nashorn";
32     }
33     else language = args[0];
34
35     final ScriptEngine engine = manager.getEngineByName(language);
36     if (engine == null)
37     {
38         System.err.println("No engine for " + language);
39         System.exit(1);
40     }
41
42     final String frameClassName
43         = args.length < 2 ? "buttons1.ButtonFrame" : args[1];
44
45     var frame
46         = (JFrame) Class.forName(frameClassName).getConstructor().newInstance();
47     InputStream in = frame.getClass().getResourceAsStream("init." + language);
48     if (in != null) engine.eval(new InputStreamReader(in));
49     var components = new HashMap<String, Component>();
50     getComponentBindings(frame, components);
51     components.forEach((name, c) -> engine.put(name, c));
52
53     var events = new Properties();
54     in = frame.getClass().getResourceAsStream(language + ".properties");
55     events.load(in);
56
57     for (Object e : events.keySet())
58     {
59         String[] s = ((String) e).split("\\.");
60         addListener(s[0], s[1], (String) events.get(e), engine, components);
61     }
62     frame.setTitle("ScriptTest");
63     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
64     frame.setVisible(true);
65 }
66 catch (ReflectiveOperationException | IOException
67         | ScriptException | IntrospectionException ex)
68 {
69     ex.printStackTrace();
70 }
71 });
72 }
73
74 /**
75 * Gathers all named components in a container.
76 * @param c the component
77 * @param namedComponents a map into which to enter the component names and components
78 */
79 private static void getComponentBindings(Component c,

```

```

80     Map<String, Component> namedComponents)
81     {
82         String name = c.getName();
83         if (name != null) { namedComponents.put(name, c); }
84         if (c instanceof Container)
85         {
86             for (Component child : ((Container) c).getComponents())
87                 getComponentBindings(child, namedComponents);
88         }
89     }
90
91 /**
92 * Adds a listener to an object whose listener method executes a script.
93 * @param beanName the name of the bean to which the listener should be added
94 * @param eventName the name of the listener type, such as "action" or "change"
95 * @param scriptCode the script code to be executed
96 * @param engine the engine that executes the code
97 * @param bindings the bindings for the execution
98 * @throws IntrospectionException
99 */
100 private static void addListener(String beanName, String eventName, final String scriptCode,
101     ScriptEngine engine, Map<String, Component> components)
102     throws ReflectiveOperationException, IntrospectionException
103 {
104     Object bean = components.get(beanName);
105     EventSetDescriptor descriptor = getEventSetDescriptor(bean, eventName);
106     if (descriptor == null) return;
107     descriptor.getAddListenerMethod().invoke(bean,
108         Proxy.newProxyInstance(null, new Class[] { descriptor.getListenerType() },
109             (proxy, method, args) ->
110             {
111                 engine.eval(scriptCode);
112                 return null;
113             }));
114 }
115
116 private static EventSetDescriptor getEventSetDescriptor(Object bean, String eventName)
117     throws IntrospectionException
118 {
119     for (EventSetDescriptor descriptor : Introspector.getBeanInfo(bean.getClass())
120         .getEventSetDescriptors())
121         if (descriptor.getName().equals(eventName)) return descriptor;
122     return null;
123 }
124 }

```

程序清单 8-2 buttons1/ButtonFrame.java

```

1 package buttons1;
2
3 import javax.swing.*;
4
5 /**
6  * A frame with a button panel.

```

```

7  * @version 1.00 2007-11-02
8  * @author Cay Horstmann
9  */
10 public class ButtonFrame extends JFrame
11 {
12     private static final int DEFAULT_WIDTH = 300;
13     private static final int DEFAULT_HEIGHT = 200;
14
15     private JPanel panel;
16     private JButton yellowButton;
17     private JButton blueButton;
18     private JButton redButton;
19
20     public ButtonFrame()
21     {
22         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
23
24         panel = new JPanel();
25         panel.setName("panel");
26         add(panel);
27
28         yellowButton = new JButton("Yellow");
29         yellowButton.setName("yellowButton");
30         blueButton = new JButton("Blue");
31         blueButton.setName("blueButton");
32         redButton = new JButton("Red");
33         redButton.setName("redButton");
34
35         panel.add(yellowButton);
36         panel.add(blueButton);
37         panel.add(redButton);
38     }
39 }
```

事件处理器是在属性文件中定义的。每个属性定义都具有下面的形式：

componentName.eventName = scriptCode

例如，如果选择使用 JavaScript，那就要在 *js.properties* 文件中提供事件处理器：

```

yellowButton.action=panel.background = java.awt.Color.YELLOW
blueButton.action=panel.background = java.awt.Color.BLUE
redButton.action=panel.background = java.awt.Color.RED
```

本书附带的代码还包括用于 Groovy、R 和 SISC Scheme 的文件。

该程序以加载在命令行中指定的语言所需的引擎开始，如果未指定语言，则使用 JavaScript。然后，我们处理 *init.language* 脚本，如果该文件存在的话。这对 R 语言和 Scheme 语言而言很有用，因为这些语言需要某些麻烦的初始化工作，我们不希望在每个事件处理器的脚本中都包括这部分工作。

接下来，我们递归地遍历所有的子构件，并在构件映射表中添加（名字，对象）绑定，然后，将它们添加到引擎中。

然后，我们读入 `language.properties` 文件。对于每一个属性，都合成其事件处理器代理，使得脚本代码得以执行。其细节有些技术性，如果你希望了解实现的细节，请参阅卷 I 第 6 章有关代理的小节。但是，其精髓部分是每个事件处理器都会调用下面的方法：

```
engine.eval(scriptCode);
```

让我们详细看看 `yellowButton`。当下面一行被处理时，

```
yellowButton.action=panel.background = java.awt.Color.YELLOW
```

我们找到了具有“`yellowButton`”名字的 `JButton` 构件，然后附着一个 `ActionListener`，它拥有 `actionPerformed` 方法，该方法将执行下面的脚本，如果该脚本是用 Nashorn 执行的：

```
panel.background = java.awt.Color.YELLOW
```

引擎包含一个将名字“`panel`”与这个 `JPanel` 对象绑定在一起的绑定。当事件发生时，该面板的 `setBackground` 方法就会执行，并且其颜色也会改变。

只需要执行下面的命令，就可以运行这个带有 JavaScript 事件处理器的程序：

```
java ScriptTest
```

对于 Groovy 处理器，需要使用

```
java -classpath .:groovy/lib/* ScriptTest groovy
```

这里，`groovy` 是 Groovy 的安装目录。

对于 R 的 Renjin 实现，要在类路径中包含 Renjin Studio 的 JAR 文件以及 Renjin 脚本引擎。它们都可以在 www.renjin.org/downloads.html 处获得。

这个应用演示了如何在 Java GUI 编程中使用脚本机制。大家可以更进一步，用 XML 文件来描述 GUI，就像在第 3 章中看到的那样。然后我们的程序就会变成解释器，去解释那些由 XML 文件定义可视化表示以及用脚本语言定义行为的 GUI。请注意这与动态 HTML 页面或动态服务器端脚本环境之间的相似性。

8.2 编译器 API

有许多工具都需要编译 Java 代码。很明显，教授 Java 编程的开发环境和程序就位于其列，测试和自动化构建工具也属于这类工具。另一个例子是 JavaServer Pages 的处理工具，JSP 是一种嵌入了 Java 语句的网页。

8.2.1 调用编译器

调用编译器非常简单，下面是一个示范调用：

```
JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
OutputStream outStream = . . .;
OutputStream errStream = . . .;
int result = compiler.run(null, outStream, errStream,
    "-sourcepath", "src", "Test.java");
```

返回值为 0 表示编译成功。

编译器会向提供给它的流发送输出和错误消息。如果将这些参数设置为 null，编译器就会使用 System.out 和 System.err。run 方法的第一个参数是输入流，由于编译器不会接受任何控制台输入，因此总是应该让其保持为 null。（run 方法是从泛化的 Tool 接口继承而来的，它考虑到某些工具需要读取输入。）

如果在命令行调用 javac，那么 run 方法其余的参数就会作为变量传递给 javac。这些变量是一些选项或文件名。

8.2.2 发起编译任务

可以通过使用 CompilationTask 对象来对编译过程进行更多的控制。如果要从字符串中提供源码，在内存中捕获类文件，或者处理错误和警告消息，这样做就会显得很有用。

要想获取 CompilationTask 对象，需要以前一节中描述的 compiler 对象开始，然后按照下面的方式调用：

```
JavaCompiler.CompilationTask task = compiler.getTask(
    errorWriter, // Uses System.err if null
    fileManager, // Uses the standard file manager if null
    diagnostics, // Uses System.err if null
    options, // null if no options
    classes, // For annotation processing; null if none
    sources);
```

最后三个参数是 Iterable 的实例。例如，选项序列可以像下面这样指定：

```
Iterable<String> options = List.of("-d", "bin");
```

sources 参数是 JavaFileObject 实例的 Iterable。如果想要编译磁盘文件，需要获取一个 StandardJavaFileManager 对象，并调用其 getJavaFileObjects 方法：

```
StandardJavaFileManager fileManager = compiler.getStandardFileManager(null, null, null);
Iterable<JavaFileObject> sources
    = fileManager.getJavaFileObjectsFromStrings(List.of("File1.java", "File2.java"));
JavaCompiler.CompilationTask task = compiler.getTask(
    null, null, null, options, null, sources);
```

注释：classes 参数只用于注解处理。在这种情况下，还需要用一个 Processor 对象的列表来调用 task.processors(annotationProcessors)。请参见 8.6 节中有关注解处理的示例。

getTask 方法会返回任务对象，但是并不会启动编译过程。CompilationTask 类扩展了 Callable<Boolean>，我们可以将其对象传递给 ExecutorService 以并行执行，或者只是做出如下的同步调用：

```
Boolean success = task.call();
```

8.2.3 捕获诊断消息

为了监听错误消息，需要安装一个 DiagnosticListener。这个监听器在编译器报告警告或

错误消息时会收到一个 `Diagnostic` 对象。`DiagnosticCollector` 类实现了这个接口，它将收集所有的诊断信息，使得你可以在编译完成之后遍历这些信息。

```
DiagnosticCollector<JavaFileObject> collector = new DiagnosticCollector<>();
compiler.getTask(null, fileManager, collector, null, null, sources).call();
for (Diagnostic<? extends JavaFileObject> d : collector.getDiagnostics())
{
    System.out.println(d);
}
```

`Diagnostic` 对象包含有关问题位置的信息（包括文件名、行号和列号）以及人类可阅读的描述。

还可以在标准的文件管理器上安装一个 `DiagnosticListener` 对象，这样就可以捕获到有关文件缺失的消息：

```
StandardJavaFileManager fileManager
= compiler.getStandardFileManager(diagnostics, null, null);
```

8.2.4 从内存中读取源文件

如果动态地生成了源代码，那么就可以从内存中获取它来进行编译，而无须在磁盘上保存文件。可以使用下面的类来持有代码：

```
public class StringSource extends SimpleJavaFileObject
{
    private String code;

    StringSource(String name, String code)
    {
        super(URI.create("string:/// " + name.replace('.', '/') + ".java"), Kind.SOURCE);
        this.code = code;
    }

    public CharSequence getCharContent(boolean ignoreEncodingErrors)
    {
        return code;
    }
}
```

然后，生成类的代码，并提交给编译器一个 `StringSource` 对象的列表：

```
List<StringSource> sources = List.of(
    new StringSource(className1, class1CodeString), . . .);
task = compiler.getTask(null, fileManager, diagnostics, null, null, sources);
```

8.2.5 将字节码写出到内存中

如果动态地编译类，那么就无须将类文件写出到硬盘上。可以将它们存储在内存中，并立即加载它们。

首先，要有一个类来持有这些字节：

```
public class ByteArrayClass extends SimpleJavaFileObject
{
```

```

private ByteArrayOutputStream out;

ByteArrayClass(String name)
{
    super(URI.create("bytes://" + name.replace('.','/') + ".class"), Kind.CLASS);
}

public byte[] getCode()
{
    return out.toByteArray();
}

public OutputStream openOutputStream() throws IOException
{
    out = new ByteArrayOutputStream();
    return out;
}
}

```

接下来，需要将文件管理器配置为使用这些类作为输出：

```

List<ByteArrayClass> classes = new ArrayList<>();
StandardJavaFileManager stdFileManager
    = compiler.getStandardFileManager(null, null, null);
JavaFileManager fileManager
    = new ForwardingJavaFileManager<JavaFileManager>(stdFileManager)
{
    public JavaFileObject getJavaFileForOutput(Location location,
        String className, Kind kind, FileObject sibling)
        throws IOException
    {
        if (kind == Kind.CLASS)
        {
            ByteArrayClass outfile = new ByteArrayClass(className);
            classes.add(outfile);
            return outfile;
        }
        else
            return super.getJavaFileForOutput(location, className, kind, sibling);
    }
};

```

为了加载这些类，需要使用类加载器（参见第 10 章）：

```

public class ByteArrayClassLoader extends ClassLoader
{
    private Iterable<ByteArrayClass> classes;

    public ByteArrayClassLoader(Iterable<ByteArrayClass> classes)
    {
        this.classes = classes;
    }

    public Class<?> findClass(String name) throws ClassNotFoundException
    {
        for (ByteArrayClass cl : classes)

```

```
        {
            if (cl.getName().equals("/") + name.replace('.', '/') + ".class"))
            {
                byte[] bytes = cl.getCode();
                return defineClass(name, bytes, 0, bytes.length);
            }
        }
        throw new ClassNotFoundException(name);
    }
}
```

编译完成后，用上面的类加载器调用 `Class.forName` 方法：

```
ByteArrayClassLoader loader = new ByteArrayClassLoader(classes);
Class<?> cl = Class.forName(className, true, loader);
```

8.2.6 示例：动态 Java 代码生成

在用于动态 Web 页面的 JSP 技术中，可以在 HTML 中混杂 Java 代码，例如：

The current date and time is **<%= new java.util.Date() %>**.

JSP 引擎动态地将 Java 代码编译到 Servlet 中。在示例应用中，我们使用了一个更简单的示例，它可以动态生成 Swing 代码。其基本思想是使用 GUI 构建器在窗体中放置构件，并在一个外部文件中指定构件的行为。程序清单 8-4 展示了一个非常简单的窗体类实例，而程序清单 8-5 展示了按钮动作的代码。请注意，窗体类的构造器调用了抽象方法 `addEventHandlers`。我们的代码生成器将产生一个实现了 `addEventHandlers` 方法的子类，并且对 `action.properties` 文件中的每一行都添加了动作监听器。(我们给读者留下了一个典型的练习，即扩展代码的生成功能，使其支持其他的事件类型。)

我们将这个子类置于名字为 `x` 的包中，因为我们不希望在程序的其他地方用到它。所生成的代码有如下形式：

```
package x;
public class Frame extends SuperclassName
{
    protected void addEventHandlers()
    {
        componentName1.addActionListener(event ->
        {
            code for event handler1
        });
        // repeat for the other event handlers . . .
    }
}
```

程序清单 8-3 的程序中的 `buildSource` 方法构建了这些代码，并将它们放到了 `StringBuilder-
JavaSource` 对象中。该对象会传递给 Java 编译器。

如前一节所述，我们使用了一个 `ForwardingJavaFileManager` 对象，它会为每一个编译过的类都构造一个 `ByteArrayClass` 对象，这些对象会捕获 `x.Frame` 类被编译时所生成的类文件。该方法将每个文件对象都添加到了一个列表中，然后将其返回，以使得我们稍后可以定位这些字节码。

编译完成后，我们使用前一节中描述的类加载器来加载存储在这个列表中的所有类。然后，我们构造并显示应用程序的窗体类。

```
var loader = new ByteArrayClassLoader(classFileObjects);
var frame = (JFrame) loader.loadClass("x.Frame").getConstructor().newInstance();
frame.setVisible(true);
```

当点击按钮时，背景色会按照常规方式进行修改。为了查看这些动作是动态编译的，可以更改 `action.properties` 文件中一行，例如，修改成下面这样：

```
yellowButton=panel.setBackground(java.awt.Color.YELLOW); yellowButton.setEnabled(false);
```

再次运行这个程序，现在，黄色按钮在点击之后就变得禁用了。再看看代码目录，你不会发现 x 包中的类的任何源文件和类文件。这个示例向你演示了如何通过内存中的源文件和类文件来使用动态编译。

程序清单 8-3 compiler/CompilerTest.java

```
1 package compiler;
2
3 import java.awt.*;
4 import java.io.*;
5 import java.nio.file.*;
6 import java.util.*;
7 import java.util.List;
8
9 import javax.swing.*;
10 import javax.tools.*;
11 import javax.tools.JavaFileObject.*;
12
13 /**
14 * @version 1.10 2018-05-01
15 * @author Cay Horstmann
16 */
17 public class CompilerTest
18 {
19     public static void main(final String[] args)
20         throws IOException, ReflectiveOperationException
21     {
22         JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
23
24         var classFileObjects = new ArrayList<ByteArrayClass>();
25         var diagnostics = new DiagnosticCollector<JavaFileObject>();
26
27         JavaFileManager fileManager = compiler.getStandardFileManager(diagnostics, null, null);
28         fileManager = new ForwardingJavaFileManager<JavaFileManager>(fileManager)
29         {
30             public JavaFileObject getJavaFileForOutput(Location location,
31                 String className, Kind kind, FileObject sibling) throws IOException
32             {
33                 if (kind == Kind.CLASS)
34                 {
35                     var fileObject = new ByteArrayClass(className);
36                     classFileObjects.add(fileObject);
37                 }
38             }
39         };
40         compiler.getTask(null, fileManager, diagnostics, null, null, null).call();
41     }
42 }
```

```

37         return fileObject;
38     }
39     else return super.getJavaFileForOutput(location, className, kind, sibling);
40   }
41 };
42
43 String frameClassName = args.length == 0 ? "buttons2.ButtonFrame" : args[0];
44 //compiler.run(null, null, null, frameClassName.replace(".", "/") + ".java");
45
46 StandardJavaFileManager fileManager2 = compiler.getStandardFileManager(null, null, null);
47 var sources = new ArrayList<JavaFileObject>();
48 for (JavaFileObject o : fileManager2.getJavaFileObjectsFromStrings(
49     List.of(frameClassName.replace(".", "/") + ".java")))
50   sources.add(o);
51
52 JavaFileObject source = buildSource(frameClassName);
53 JavaCompiler.CompilationTask task = compiler.getTask(null, fileManager, diagnostics,
54   null, null, List.of(source));
55 Boolean result = task.call();
56
57 for (Diagnostic<? extends JavaFileObject> d : diagnostics.getDiagnostics())
58   System.out.println(d.getKind() + ": " + d.getMessage(null));
59 fileManager.close();
60 if (!result)
61 {
62   System.out.println("Compilation failed.");
63   System.exit(1);
64 }
65
66 var loader = new ByteArrayClassLoader(classFileObjects);
67 var frame = (JFrame) loader.loadClass("x.Frame").getConstructor().newInstance();
68
69 EventQueue.invokeLater(() ->
70 {
71   frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
72   frame.setTitle("CompilerTest");
73   frame.setVisible(true);
74 });
75 }
76
77 /**
78 * Builds the source for the subclass that implements the addEventHandlers method.
79 * @return a file object containing the source in a string builder
80 */
81 static JavaFileObject buildSource(String superClassNames)
82   throws IOException, ClassNotFoundException
83 {
84   var builder = new StringBuilder();
85   builder.append("package x;\n\n");
86   builder.append("public class Frame extends " + superClassNames + " {\n");
87   builder.append("protected void addEventHandlers() {\n");
88   var props = new Properties();
89   props.load(Files.newInputStream(Paths.get(
90     superClassNames.replace(".", "/").getParent().resolve("action.properties"))));

```

```

91     for (Map.Entry<Object, Object> e : props.entrySet())
92     {
93         var beanName = (String) e.getKey();
94         var eventCode = (String) e.getValue();
95         builder.append(beanName + ".addActionListener(event -> {\n");
96         builder.append(eventCode);
97         builder.append("\n} );\n");
98     }
99     builder.append("} }\n");
100    return new StringSource("x.Frame", builder.toString());
101 }
102 }
```

程序清单 8-4 buttons2/ButtonFrame.java

```

1 package buttons2;
2 import javax.swing.*;
3
4 /**
5  * A frame with a button panel.
6  * @version 1.00 2007-11-02
7  * @author Cay Horstmann
8 */
9 public abstract class ButtonFrame extends JFrame
10 {
11     public static final int DEFAULT_WIDTH = 300;
12     public static final int DEFAULT_HEIGHT = 200;
13
14     protected JPanel panel;
15     protected JButton yellowButton;
16     protected JButton blueButton;
17     protected JButton redButton;
18
19     protected abstract void addEventHandlers();
20
21     public ButtonFrame()
22     {
23         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
24
25         panel = new JPanel();
26         add(panel);
27
28         yellowButton = new JButton("Yellow");
29         blueButton = new JButton("Blue");
30         redButton = new JButton("Red");
31
32         panel.add(yellowButton);
33         panel.add(blueButton);
34         panel.add(redButton);
35
36         addEventHandlers();
37     }
38 }
```

程序清单 8-5 buttons2/action.properties

```
1 yellowButton=panel.setBackground(java.awt.Color.YELLOW);
2 blueButton=panel.setBackground(java.awt.Color.BLUE);
```

API javax.tools.Tool 6

- int run(InputStream in, OutputStream out, OutputStream err, String... arguments)

用给定的输入、输出、错误流，以及给定的参数来运行该工具。返回值为 0 表示成功，非 0 值表示失败。

API javax.tools.JavaCompiler 6

- StandardJavaFileManager getStandardFileManager(DiagnosticListener<? super JavaFileObject> diagnosticListener, Locale locale, Charset charset)

获取该编译器的标准文件管理器。如果要使用默认的错误报告机制、locale 和字符集等参数，则可以提供 null。

- JavaCompiler.CompilationTask getTask(Writer out, JavaFileManager fileManager, DiagnosticListener<? super JavaFileObject> diagnosticListener, Iterable<String> options, Iterable<String> classesForAnnotationProcessing, Iterable<? extends JavaFileObject> sourceFiles)

获取编译任务，在被调用时，该任务将编译给定的源文件。参见前一节中有关这部分内容的详细讨论。

API javax.tools.StandardJavaFileManager 6

- Iterable<? extends JavaFileObject> getJavaFileObjectsFromStrings(Iterable<String> fileNames)
- Iterable<? extends JavaFileObject> getJavaFileObjectsFromFiles(Iterable<? extends File> files)

将文件名或文件序列转译成一个 JavaFileObject 实例序列。

API javax.tools.JavaCompiler.CompilationTask 6

- Boolean call()

执行编译任务。

API javax.tools.DiagnosticCollector<S> 6

- DiagnosticCollector<S> DiagnosticCollector<S>()

构造一个空收集器。
- List<Diagnostic<? extends S>> getDiagnostics()

获取收集到的诊断信息。

API javax.tools.Diagnostic<S> 6

- S getSource()

获取与该诊断信息相关联的源对象。

- `Diagnostic.Kind getKind()`

获取该诊断信息的类型，返回值为 `ERROR`, `WARNING`, `MANDATORY_WARNING`, `NOTE` 或 `OTHER` 之一。

- `String getMessage(Locale locale)`

获取一条消息，这条消息描述了由该诊断信息所揭示的问题。如果要使用默认的 `locale`，则传递 `null`。

- `long getLineNumber()`

- `long getColumnNumber()`

获取由该诊断信息所揭示的问题的位置。

`javax.tools.SimpleJavaFileObject` 6

- `CharSequence getCharContent(boolean ignoreEncodingErrors)`

对于表示源文件并产生源代码的文件对象，需要覆盖该方法。

- `OutputStream openOutputStream()`

对于表示类文件并产生字节码可写入其中的流的文件对象，需要覆盖该方法。

`javax.tools.ForwardingJavaFileManager<M extends JavaFileManager>` 6

- `protected ForwardingJavaFileManager(M fileManager)`

构造一个 `JavaFileManager`，它将所有的调用都代理给指定的文件管理器。

- `FileObject getFileForOutput(JavaFileManager.Location location, String className, JavaFileObject.Kind kind, FileObject sibling)`

如果希望替换用于写出类文件的文件对象，则需要拦截该调用。`kind` 的值是 `SOURCE`, `CLASS`, `HTML` 或 `OTHER` 之一。

8.3 使用注解

注解是那些插入到源代码中使用其他工具可以对其进行处理的标签。这些工具可以在源码层次上进行操作，或者可以处理编译器在其中放置了注解的类文件。

注解不会改变程序的编译方式。Java 编译器对于包含注解和不包含注解的代码会生成相同的虚拟机指令。

为了能够受益于注解，你需要选择一个处理工具，然后向你的处理工具可以理解的代码中插入注解，之后运用该处理工具处理代码。

注解的使用范围还是很广泛的，并且这种广泛性让人乍一看会觉得有些杂乱无章。下面是关于注解的一些可能的用法：

- 附属文件的自动生成，例如部署描述符或者 bean 信息类。

- 测试、日志、事务语义等代码的自动生成。

8.3.1 注解简介

我们首先介绍基本概念，然后将这些概念运用到一个具体示例中：我们将某些方法标注为 AWT 构件的事件监听器，然后向你展示一个能够分析注解和连接监听器的注解处理器。然后，我们对其语法规则进行详细讨论。最后我们以两个注解处理的高级示例结束本章。其中一个可以处理源代码级别的注解。另外一个使用了 Apache 的字节码工程类库，可以向注解过的方法中添加额外的字节码。

下面是一个简单注解的示例：

```
public class MyClass
{
    ...
    @Test public void checkRandomInsertions()
}
```

注解 `@Test` 用于注解 `checkRandomInsertions` 方法。

在 Java 中，注解是当作一个修饰符来使用的，它被置于被注解项之前，中间没有分号。（修饰符就是诸如 `public` 和 `static` 之类的关键词。）每一个注解的名称前面都加上了 @ 符号，这有点类似于 Javadoc 的注释。然而，Javadoc 注释出现在 `/**...*/` 定界符的内部，而注解是代码的一部分。

`@Test` 注解自身并不会做任何事情，它需要工具支持才会有用。例如，当测试一个类的时候，JUnit4 测试工具（可以从 <http://junit.org> 处获得）可能会调用所有标识为 `@Test` 的方法。另一个工具可能会删除一个类文件中的所有测试方法，以便在对这个类测试完毕后，不会将这些测试方法与程序装载在一起。

注解可以定义成包含元素的形式，例如：

```
@Test(timeout="10000")
```

这些元素可以被读取这些注解的工具去处理。其他形式的元素也是有可能的；我们将会在本章的随后部分进行讨论。

除了方法外，还可以注解类、成员以及局部变量，这些注解可以存在于任何可以放置一个像 `public` 或者 `static` 这样的修饰符的地方。另外，正如在 8.4 节中看到的，你还可以注解包、参数变量、类型参数和类型用法。

每个注解都必须通过一个注解接口进行定义。这些接口中的方法与注解中的元素相对应。例如，JUnit 的注解 `Test` 可以用下面这个接口进行定义：

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Test
{
    long timeout() default 0L;
    ...
}
```

`@interface` 声明创建了一个真正的 Java 接口。处理注解的工具将接收那些实现了这个注

解接口的对象。这类工具可以调用 `timeout` 方法来获取某个特定 `Test` 注解的 `timeout` 元素。

注解 `Target` 和 `Retention` 是元注解。它们注解了 `Test` 注解，即将 `Test` 注解标识成一个只能运用到方法上的注解，并且当类文件载入到虚拟机的时候，它仍可以保留下。我们将在 8.5.3 节详细讨论这些元注解。

你现在已经清楚了程序的元数据和注解这两个概念。在接下来的小节中，我们将深入到一个注解处理的具体示例中继续探讨。

 **注释：**对于注解引人入胜的用法，可以查看 `JCommander` (<http://jcommander.org>) 和 `picocli` (<http://picocli.info>)。这些类库将注解用于命令行参数的处理。

8.3.2 示例：注解事件处理器

在用户界面编程中，一件更令人讨厌的事情就是组装事件源上的监听器。很多监听器是下面这种形式的：

```
myButton.addActionListener(() -> doSomething());
```

在本节，我们设计了一个注解来免除这种苦差事。该注解是在程序清单 8-8 中定义的，其使用方式如下：

```
@ActionListenerFor(source="myButton") void doSomething() { . . . }
```

程序员不再需要去调用 `addActionListener` 了。相反地，每个方法直接用一个注解标记起来。程序清单 8-7 展示了卷 I 第 10 章的 `ButtonFrame` 程序，但是使用上述这类注解重新实现了一遍。

我们还需要定义一个注解接口，代码在程序清单 8-8 中。

程序清单 8-6 runtimeAnnotations/ActionListenerInstaller.java

```
1 package runtimeAnnotations;
2
3 import java.awt.event.*;
4 import java.lang.reflect.*;
5
6 /**
7  * @version 1.00 2004-08-17
8  * @author Cay Horstmann
9  */
10 public class ActionListenerInstaller
11 {
12     /**
13      * Processes all ActionListenerFor annotations in the given object.
14      * @param obj an object whose methods may have ActionListenerFor annotations
15      */
16     public static void processAnnotations(Object obj)
17     {
18         try
19         {
20             Class<?> cl = obj.getClass();
```

```

21     for (Method m : cl.getDeclaredMethods())
22     {
23         ActionListenerFor a = m.getAnnotation(ActionListenerFor.class);
24         if (a != null)
25         {
26             Field f = cl.getDeclaredField(a.source());
27             f.setAccessible(true);
28             addListener(f.get(obj), obj, m);
29         }
30     }
31 }
32 catch (ReflectiveOperationException e)
33 {
34     e.printStackTrace();
35 }
36 }
37 */
38 /**
39 * Adds an action listener that calls a given method.
40 * @param source the event source to which an action listener is added
41 * @param param the implicit parameter of the method that the listener calls
42 * @param m the method that the listener calls
43 */
44 public static void addListener(Object source, final Object param, final Method m)
45     throws ReflectiveOperationException
{
46     var handler = new InvocationHandler()
47     {
48         public Object invoke(Object proxy, Method mm, Object[] args) throws Throwable
49         {
50             return m.invoke(param);
51         }
52     };
53
54     Object listener = Proxy.newProxyInstance(null,
55         new Class[] { java.awt.event.ActionListener.class }, handler);
56     Method adder = source.getClass().getMethod("addActionListener", ActionListener.class);
57     adder.invoke(source, listener);
58 }
59 }
60 }

```

程序清单 8-7 buttons3/ButtonFrame.java

```

1 package buttons3;
2
3 import java.awt.*;
4 import javax.swing.*;
5 import runtimeAnnotations.*;
6
7 /**
8  * A frame with a button panel.
9  * @version 1.00 2004-08-17
10 * @author Cay Horstmann
11 */

```

```

12 public class ButtonFrame extends JFrame
13 {
14     private static final int DEFAULT_WIDTH = 300;
15     private static final int DEFAULT_HEIGHT = 200;
16
17     private JPanel panel;
18     private JButton yellowButton;
19     private JButton blueButton;
20     private JButton redButton;
21
22     public ButtonFrame()
23     {
24         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
25
26         panel = new JPanel();
27         add(panel);
28
29         yellowButton = new JButton("Yellow");
30         blueButton = new JButton("Blue");
31         redButton = new JButton("Red");
32
33         panel.add(yellowButton);
34         panel.add(blueButton);
35         panel.add(redButton);
36
37         ActionListenerInstaller.processAnnotations(this);
38     }
39
40     @ActionListenerFor(source = "yellowButton")
41     public void yellowBackground()
42     {
43         panel.setBackground(Color.YELLOW);
44     }
45
46     @ActionListenerFor(source = "blueButton")
47     public void blueBackground()
48     {
49         panel.setBackground(Color.BLUE);
50     }
51
52     @ActionListenerFor(source = "redButton")
53     public void redBackground()
54     {
55         panel.setBackground(Color.RED);
56     }
57 }
```

程序清单 8-8 runtimeAnnotations/ActionButtonFor.java

```

1 package runtimeAnnotations;
2
3 import java.lang.annotation.*;
4
5 /**
```

```

6  * @version 1.00 2004-08-17
7  * @author Cay Horstmann
8  */
9 @Target(ElementType.METHOD)
10 @Retention(RetentionPolicy.RUNTIME)
11 public @interface ActionListenerFor
12 {
13     String source();
14 }

```

当然，这些注解本身不会做任何事情，它们只是存在于源文件中。编译器将它们置于类文件中，并且虚拟机会将它们载入。我们现在需要的是一个分析注解以及安装行为监听器的机制。这也是类 `ActionListenerInstaller` 的职责所在。`ButtonFrame` 构造器将调用下面的方法：

```
ActionListenerInstaller.processAnnotations(this);
```

静态的 `processAnnotations` 方法可以枚举出某个对象接收到的所有方法。对于每一个方法，它先获取 `ActionListenerFor` 注解对象，然后再对它进行处理。

```

Class<?> cl = obj.getClass();
for (Method m : cl.getDeclaredMethods())
{
    ActionListenerFor a = m.getAnnotation(ActionListenerFor.class);
    if (a != null) . .
}

```

这里，我们使用了定义在 `AnnotatedElement` 接口中的 `getAnnotation` 方法。`Method`、`Constructor`、`Field`、`Class` 和 `Package` 这些类都实现了这个接口。

源成员域的名字是存储在注解对象中的。我们可以通过调用 `source` 方法对它进行检索，然后查找匹配的成员域。

```

String fieldName = a.source();
Field f = cl.getDeclaredField(fieldName);

```

这表明我们的注解有点局限。源元素必须是一个成员域的名字，而不能是局部变量。

代码的剩余部分相当具有技术性。对于每一个被注解的方法，我们构造了一个实现了 `ActionListener` 接口的代理对象，其 `actionPerformed` 方法将调用这个被注解过的方法。（关于代理的更多信息见卷 I 第 6 章。）细节并不重要，关键要知道注解的功能是通过 `processAnnotations` 方法建立起来的。

图 8-1 展示了在本例中注解是如何被处理的。

在这个示例中，注解是在运行时进行处理的。另外也可以在源码级别上对它们进行处理，这样，源代码生成器将产生用于添加监听器的代码。注解也可以在字节码级别上进行处理，字节码编辑器可以将对 `addActionListener` 的调用注入框体构造器中。听起来似乎很复杂，不过可以利用一些类库相对直截了当地实现这项任务。

对于用户界面程序员来说，我们这个示例并不能看作是一个严格意义上的工具。因为，用于添加监听器的实用方法对于程序员来说和添加一条注解一样方便。（实际上，`java.beans.EventHandler` 类试图实现就是这样。通过在这个类中提供一个可以添加事件处理器的方法，

而不只是构建它，就可以很容易地对它进行改进。)

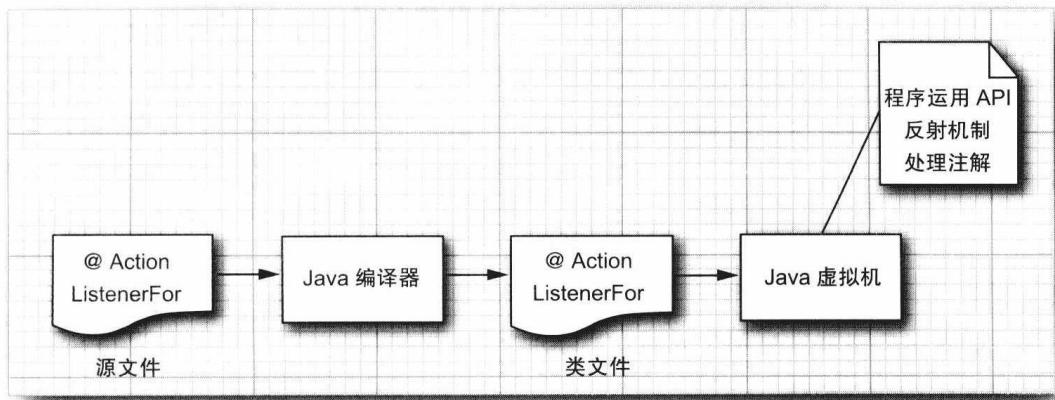


图 8-1 在运行时处理注解

不过，这个示例展示了对一个程序进行注解以及对这些注解进行分析的机制。既然你已经领会了这个具体示例，那么，现在可能已经为后续小节详述注解语法做好了更充分的准备（这也是我们所希望的）。

API `java.lang.reflect.AnnotatedElement 5.0`

- `boolean isAnnotationPresent(Class<? extends Annotation> annotationType)`
如果该项具有给定类型的注解，则返回 `true`。
- `<T extends Annotation> T getAnnotation(Class<T> annotationType)`
获得给定类型的注解，如果该项不具有这样的注解，则返回 `null`。
- `<T extends Annotation> T[] getAnnotationsByType(Class<T> annotationType) 8`
获得某个可重复注解类型的所有注解（查阅 8.5.3 节），或者返回长度为 0 的数组。
- `Annotation[] getAnnotations()`
获得作用于该项的所有注解，包括继承而来的注解。如果没有出现任何注解，那么将返回一个长度为 0 的数组。
- `Annotation[] getDeclaredAnnotations()`
获得为该项声明的所有注解，不包含继承而来的注解。如果没有出现任何注解，那么将返回一个长度为 0 的数组。

8.4 注解语法

本节将介绍你必须了解的注解语法。

8.4.1 注解接口

注解是由注解接口来定义的：

```
modifiers @interface AnnotationName
{
    elementDeclaration_1
    elementDeclaration_2
    ...
}
```

每个元素声明都具有下面这种形式：

```
type elementName();
```

或者

```
type elementName() default value;
```

例如，下面这个注解具有两个元素：`assignedTo` 和 `severity`。

```
public @interface BugReport
{
    String assignedTo() default "[none]";
    int severity();
}
```

所有的注解接口都隐式地扩展自 `java.lang.annotation.Annotation` 接口。这个接口是一个常规接口，不是一个注解接口。请查看本节最后为该接口提供的一些方法所做的 API 注解。

你无法扩展注解接口。换句话说，所有的注解接口都直接扩展自 `java.lang.annotation.Annotation`。

你从来不用为注解接口提供实现类。

注解元素的类型为下列之一：

- 基本类型 (`int`、`short`、`long`、`byte`、`char`、`double`、`float` 或者 `boolean`)。
- `String`。
- `Class` (具有一个可选的类型参数，例如 `Class<? extends MyClass`)。
- `enum` 类型。
- 注解类型。
- 由前面所述类型组成的数组 (由数组组成的数组不是合法的元素类型)。

下面是一些合法的元素声明的例子：

```
public @interface BugReport
{
    enum Status { UNCONFIRMED, CONFIRMED, FIXED, NOTABUG };
    boolean showStopper() default false;
    String assignedTo() default "[none]";
    Class<?> testCase() default Void.class;
    Status status() default Status.UNCONFIRMED;
    Reference ref() default @Reference(); // an annotation type
    String[] reportedBy();
}
```

API `java.lang.annotation.Annotation` 5.0

- `Class<? extends Annotation> annotationType()`

返回 Class 对象，它用于描述该注解对象的注解接口。注意：调用注解对象上的 getClass 方法可以返回真正的类，而不是接口。

- `boolean equals(Object other)`

如果 other 是一个实现了与该注解对象相同的注解接口的对象，并且如果该对象和 other 的所有元素彼此相等。那么返回 True。

- `int hashCode()`

返回一个与 equals 方法兼容、由注解接口名以及元素值衍生而来的散列码。

- `String toString()`

返回一个包含注解接口名以及元素值的字符串表示，例如，`@BugReport (assignedTo=[none], severity=0)`。

8.4.2 注解

每个注解都具有下面这种格式：

```
@AnnotationName(elementName1=value1, elementName2=value2, . . .)
```

例如，

```
@BugReport(assignedTo="Harry", severity=10)
```

元素的顺序无关紧要。下面这个注解和前面那个一样。

```
@BugReport(severity=10, assignedTo="Harry")
```

如果某个元素的值并未指定，那么就使用声明的默认值。例如，考虑一下下面这个注解：

```
@BugReport(severity=10)
```

元素 assignedTo 的值是字符串 "[none]"。

! **警告：**默认值并不是和注解存储在一起的；相反地，它们是动态计算而来的。例如，如果你将元素 assignedTo 的默认值更改为 "[]"，然后重新编译 BugReport 接口，那么注解 @BugReport(severity=10) 将使用这个新的默认值，甚至在那些在默认值修改之前就已经编译过的类文件中也是如此。

有两个特殊的快捷方式可以用来简化注解。

如果没有指定元素，要么是因为注解中没有任何元素，要么是因为所有元素都使用默认值，那么你就不需要使用圆括号了。例如，

```
@BugReport
```

和下面这个注解是一样的

```
@BugReport(assignedTo="[none]", severity=0)
```

这样的注解又称为标记注解。

另外一种快捷方式是单值注解。如果一个元素具有特殊的名字 value，并且没有指定其他元素，那么你就可以忽略掉这个元素名以及等号。例如，既然我们已经在前面将 Action

`ListenerFor` 注解接口定义为如下形式：

```
public @interface ActionListenerFor
{
    String value();
}
```

那么，我们可以将这个注解书写成如下形式：

```
@ActionListenerFor("yellowButton")
```

而不是

```
@ActionListenerFor(value="yellowButton")
```

一个项可以有多个注解：

```
@Test
@BugReport(showStopper=true, reportedBy="Joe")
public void checkRandomInsertions()
```

如果注解的作者将其声明为可重复的，那么你就可以多次重复使用同一个注解：

```
@BugReport(showStopper=true, reportedBy="Joe")
@BugReport(reportedBy={"Harry", "Carl"})
public void checkRandomInsertions()
```

注释：因为注解是由编译器计算而来的，因此，所有元素值必须是编译期常量。例如，

```
@BugReport(showStopper=true, assignedTo="Harry", testCase=MyTestCase.class,
           status=BugReport.Status.CONFIRMED, . . .)
```

警告：一个注解元素永远不能设置为 `null`，甚至不允许其默认值为 `null`。这样在实际应用中会相当不方便。你必须使用其他的默认值，例如 "" 或者 `Void.class`。

如果元素值是一个数组，那么要将它的值用括号括起来，像下面这样：

```
@BugReport(. . ., reportedBy={"Harry", "Carl"})
```

如果该元素具有单值，那么可以忽略这些括号：

```
@BugReport(. . ., reportedBy="Joe") // OK, same as {"Joe"}
```

既然一个注解元素可以是另一个注解，那么就可以创建出任意复杂的注解。例如，

```
@BugReport(ref=@Reference(id="3352627"), . . .)
```

注释：在注解中引入循环依赖是一种错误。例如，因为 `BugReport` 具有一个注解类型为 `Reference` 的元素，所以 `Reference` 就不能再拥有一个类型为 `BugReport` 的元素。

8.4.3 注解各类声明

注解可以出现在许多地方，这些地方可以分为两类：声明和类型用法声明注解可以出现在下列声明处：

- 包

- 类（包括 enum）
- 接口（包括注解接口）
- 方法
- 构造器
- 实例域（包含 enum 常量）
- 局部变量
- 参数变量
- 类型参数

对于类和接口，需要将注解放置在 class 和 interface 关键词的前面：

```
@Entity public class User { . . . }
```

对于变量，需要将它们放置在类型的前面：

```
@SuppressWarnings("unchecked") List<User> users = . . .;
public User getUser(@Param("id") String userId)
```

泛化类或方法中的类型参数可以像下面这样被注解：

```
public class Cache<@Immutable V> { . . . }
```

包是在文件 package-info.java 中注解的，该文件只包含以注解先导的包语句。

```
/**
 * Package-level Javadoc
 */
@GPL(version="3")
package com.horstmann.corejava;
import org.gnu.GPL;
```

注释：对局部变量的注解只能在源码级别上进行处理。类文件并不描述局部变量。因此，所有的局部变量注解在编译完一个类的时候就会被遗弃掉。同样地，对包的注解不能在源码级别之外存在。

8.4.4 注解类型用法

声明注解提供了正在被声明的项的相关信息。例如，在下面的声明中

```
public User getUser(@NotNull String userId)
```

就断言 userId 参数不为空。

注释：@NotNull 注解是 Checker Framework 的一部分 (<http://types.cs.washington.edu/checker-framework>)。通过使用这个框架，可以在程序中包含断言，例如某个参数不为空，或者某个 String 包含一个正则表达式。然后，静态分析工具将检查在给定的源代码段中这些断言是否有效。

现在，假设我们有一个类型为 List<String> 的参数，并且想要表示其中所有的字符串都不为 null。这就是类型用法注解大显身手之处，可以将该注解放置到类型参数之前：List<@

`@NotNull String>`。

类型用法注解可以出现在下面的位置：

- 与泛化类型参数一起使用：`List<@NotNull String>, Comparator.<@NotNull String> reverseOrder()`。
- 数组中的任何位置：`@NotNull String[][] words (words[i][j] 不为 null), String @NotNull [][] words (words 不为 null), String[] @NotNull [] words (words[i] 不为 null)`。
- 与超类和实现接口一起使用：`class Warning extends @Localized Message`。
- 与构造器调用一起使用：`new @Localized String(...)`。
- 与强制转型和 `instanceof` 检查一起使用：`(@Localized String) text, if (text instanceof @Localized String)`。（这些注解只供外部工具使用，它们对强制转型和 `instanceof` 检查不会产生任何影响。）
- 与异常规约一起使用：`public String read() throws @Localized IOException`。
- 与通配符和类型边界一起使用：`List<@Localized ? extends Message>, List<? extends @Localized Message>`。
- 与方法和构造器引用一起使用：`@Localized Message::getText`。

有多种类型位置是不能被注解的：

```
@NotNull String.class // ERROR: Cannot annotate class literal
import java.lang.@NotNull String; // ERROR: Cannot annotate import
```

可以将注解放置到诸如 `private` 和 `static` 这样的其他修饰符的前面或后面。习惯（但不是必需）的做法，是将类型用法注解放置到其他修饰符的后面和将声明注解放置到其他修饰符的前面。例如，

```
private @NotNull String text; // Annotates the type use
@Id private String userId; // Annotates the variable
```

注释：注解的作者需要指定特定的注解可以出现在哪里。如果一个注解可以同时应用于变量和类型用法，并且它确实被应用到了某个变量声明上，那么该变量和类型用法就都被注解了。例如，请考虑

```
public User getUser(@NotNull String userId)
```

如果 `@NotNull` 可以同时应用于参数和类型用法，那么 `userId` 参数就被注解了，而其参数类型是 `@NotNull String`。

8.4.5 注解 this

假设想要将参数注解为在方法中不会被修改。

```
public class Point
{
    public boolean equals(@ReadOnly Object other) { . . . }
}
```

那么，处理这个注解的工具在看到下面的调用时

```
p.equals(q)
```

就会推理出 q 没有被修改过。

但是 p 呢？

当该方法被调用时，this 变量是绑定到 p 的。但是 this 从来都没有被声明过，因此你无法注解它。

实际上，你可以用一种很少使用的语法变体来声明它，这样你就可以添加注解了：

```
public class Point
{
    public boolean equals(@ReadOnly Point this, @ReadOnly Object other) { . . . }
}
```

第一个参数被称为接收器参数，它必须被命名为 this，而它的类型就是要构建的类。

 **注释：**你只能为方法而不能为构造器提供接收器参数。从概念上讲，构造器中的 this 引用在构造器没有执行完之前还不是给定类型的对象。所以，放置在构造器上的注解描述的是被构建的对象的属性。

传递给内部类构造器的是另一个不同的隐藏参数，即对其外围类对象的引用。你也可以让这个参数显式化：

```
public class Sequence
{
    private int from;
    private int to;

    class Iterator implements java.util.Iterator<Integer>
    {
        private int current;
        public Iterator(@ReadOnly Sequence Sequence.this)
        {
            this.current = Sequence.this.from;
        }
        . .
    }
}
```

这个参数的名字必须像引用它时那样，叫做 EnclosingClass.this，其类型为外围类。

8.5 标准注解

Java SE 在 `java.lang`、`java.lang.annotation` 和 `javax.annotation` 包中定义了大量的注解接口。其中四个是元注解，用于描述注解接口的行为属性，其他的三个是规则接口，可以用它们来注解你的源代码中的项。表 8-2 列出了这些注解。我们将会在随后的两个小节中给予详细介绍。

表 8-2 标准注解

注解接口	应用场合	目的
Deprecated	全部	将项标记为过时的
SuppressWarnings	除了包和注解之外的所有情况	阻止某个给定类型的警告信息
SafeVarargs	方法和构造器	断言 varargs 参数可安全使用
Override	方法	检查该方法是否覆盖了某一个超类方法
FunctionalInterface	接口	将接口标记为只有一个抽象方法的函数式接口
PostConstruct PreDestroy	方法	被标记的方法应该在构造之后或移除之前立即被调用
Resource	类、接口、方法、域	在类或接口上：标记为在其他地方要用到的资源。在方法或域上：为“注入”而标记
Resources	类、接口	一个资源数组
Generated	全部	
Target	注解	指明可以应用这个注解的那些项
Retention	注解	指明这个注解可以保留多久
Documented	注解	指明这个注解应该包含在注解项的文档中
Inherited	注解	指明当这个注解应用于一个类的时候，能够自动被它的子类继承
Repeatable	注解	指明这个注解可以在同一个项上应用多次

8.5.1 用于编译的注解

@Deprecated 注解可以被添加到任何不再鼓励使用的项上。所以，当你使用一个已过时的项时，编译器将会发出警告。这个注解与 Javadoc 标签 @deprecated 具有同等功效。但是，该注解会一直持久化到运行时。

 **注释：**jdeprscan 工具可以扫描 JAR 文件集中的过时元素，它是 JDK 的组成部分。

@SuppressWarnings 注解会告知编译器阻止特定类型的警告信息，例如，

```
@SuppressWarnings("unchecked")
```

@Override 这种注解只能应用到方法上。编译器会检查具有这种注解的方法是否真正覆盖了一个来自于超类的方法。例如，如果你声明：

```
public MyClass
{
    @Override public boolean equals(MyClass other);
    ...
}
```

那么编译器会报告一个错误。毕竟，这个 equals 方法没有覆盖 Object 类的 equals 方法。因为那个方法有一个类型为 Object 而不是 MyClass 的参数。

@Generated 注解的目的是供代码生成工具来使用。任何生成的源代码都可以被注解，从而与程序员提供的代码区分开。例如，代码编辑器可以隐藏生成的代码，或者代码生成器可以

移除生成代码的旧版本。每个注解都必须包含一个表示代码生成器的唯一标识符，而日期字符串（ISO8601 格式）和注释字符串是可选的。例如，

```
@Generated("com.horstmann.beanproperty", "2008-01-04T12:08:56.235-0700");
```

8.5.2 用于管理资源的注解

`@PostConstruct` 和 `@PreDestroy` 注解用于控制对象生命周期的环境中，例如 Web 容器和应用服务器。标记了这些注解的方法应该在对象被构建之后，或者在对象被移除之前，紧接着调用。

`@Resource` 注解用于资源注入。例如，考虑一下访问数据库的 Web 应用。当然，数据库访问信息不应该被硬编码到 Web 应用中。而是应该让 Web 容器提供某种用户接口，以便设置连接参数和数据库资源的 JNDI 名字。在这个 Web 应用中，可以像下面这样引用数据源：

```
@Resource(name="jdbc/mydb")
private DataSource source;
```

当包含这个域的对象被构造时，容器会“注入”一个对该数据源的引用。

8.5.3 元注解

`@Target` 元注解可以应用于一个注解，以限制该注解可以应用到哪些项上。例如，

```
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface BugReport
```

表 8-3 显示了所有可能的取值情况，它们属于枚举类型 `ElementType`。可以指定任意数量的元素类型，用括号括起来。

表 8-3 `@Target` 注解的元素类型

元素类型	注解适用场合	元素类型	注解适用场合
ANNOTATION_TYPE	注解类型声明	FIELD	成员域（包括 enum 常量）
PACKAGE	包	PARAMETER	方法或构造器参数
TYPE	类（包括 enum）及接口 （包括注解类型）	LOCAL_VARIABLE	局部变量
METHOD	方法	TYPE_PARAMETER	类型参数
CONSTRUCTOR	构造器	TYPE_USE	类型用法

一条没有 `@Target` 限制的注解可以应用于任何项上。编译器将检查你是否将一条注解只应用到了某个允许的项上。例如，如果将 `@BugReport` 应用于一个成员域上，则会导致一个编译器错误。

`@Retention` 元注解用于指定一条注解应该保留多长时间。只能将其指定为表 8-4 中的任意值，其默认值是 `RetentionPolicy.CLASS`。

表 8-4 用于 @Retention 注解的保留策略

保留规则	描述
SOURCE	不包括在类文件中的注解
CLASS	包括在类文件中的注解，但是虚拟机不需要将它们载入
RUNTIME	包括在类文件中的注解，并由虚拟机载入。通过反射 API 可获得它们

在程序清单 8-8 中，`@ActionListenerFor` 注解声明为具有 `RetentionPolicy.RUNTIME`，因为我们将是使用反射机制进行注解处理的。在随后的两个小节里，你将会看到一些在源码级别和类文件级别上怎样对注解进行处理的示例。

`@Documented` 元注解为像 Javadoc 这样的归档工具提供了一些提示。应该像处理其他修饰符（例如 `protected` 和 `static`）一样来处理归档注解，以实现其归档目的。其他注解的使用并不会纳入归档的范畴。例如，假定我们将 `@ActionListenerFor` 作为一个归档注解来声明：

```
@Documented
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface ActionListenerFor
```

现在每一个被该注解标注过的方法的归档就会含有这条注解，如图 8-2 所示。

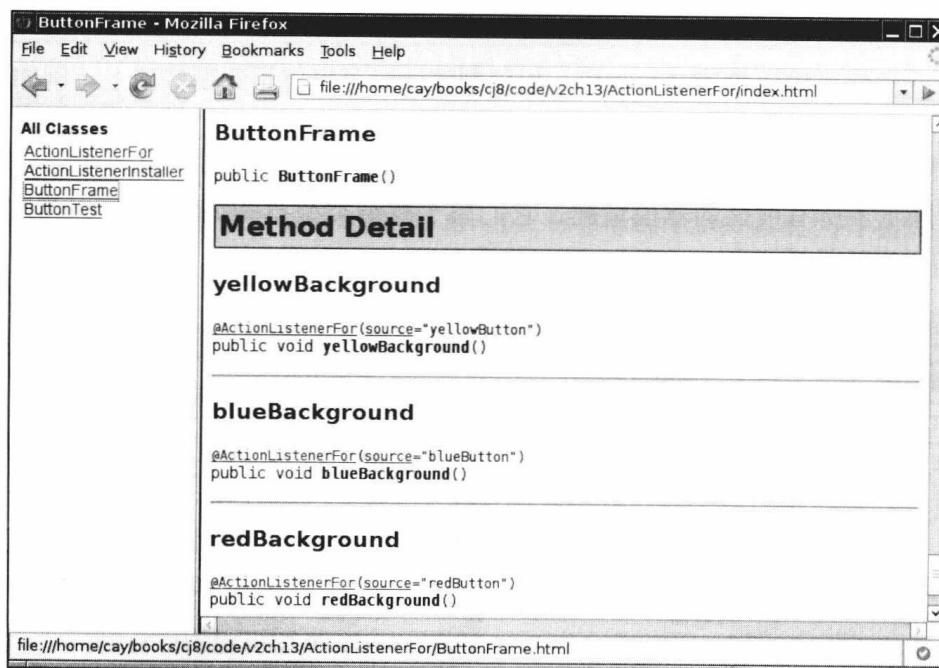


图 8-2 归档注解

如果某个注解是暂时性的（例如 `@BugReport`），那么就不应该对它们的用法进行归档。

注释：将一个注解应用到它自身上是合法的。例如，`@Documented` 注解被它自身注解为 `@Documented`。因此，针对注解的 Javadoc 文档可以表明它们是否可被归档。

@Inherited 元注解只能应用于对类的注解。如果一个类具有继承注解，那么它的所有子类都自动具有同样的注解。这使得创建一个与 Serializable 这样的标记接口具有相同运行方式的注解变得很容易。

实际上，@Serializable 注解应该比没有任何方法的 Serializable 标记接口更适用。一个类之所以可以被序列化，是因为存在着对它的成员域进行读写的运行期支持，而不是因为任何面向对象的设计原则。注解比接口继承更擅长描述这一事实。当然，可序列化接口是在 JDK1.1 中产生的，远比注解出现得早。

假设定义了一个继承注解 @Persistent 来指明一个类的对象可以存储到数据库中，那么该持久类的子类就会自动被注解为是持久性的。

```
@Inherited @interface Persistent { }
@Persistent class Employee { . . . }
class Manager extends Employee { . . . } // also @Persistent
```

在持久化机制去查找存储在数据库中的对象时，它就会同时探测到 Employee 对象以及 Manager 对象。

对于 Java SE 8 来说，将同种类型的注解多次应用于某一项是合法的。为了向后兼容，可重复注解的实现者需要提供一个容器注解，它可以将这些重复注解存储到一个数组中。

下面是如何定义 @TestCase 注解以及它的容器的代码：

```
@Repeatable(TestCases.class)
@interface TestCase
{
    String params();
    String expected();
}

@interface TestCases
{
    TestCase[] value();
}
```

无论何时，只要用户提供了两个或更多个 @TestCase 注解，那么它们就会自动地被包装到一个 @TestCases 注解中。

◆ **警告：**在处理可重复注解时必须非常仔细。如果调用 getAnnotation 来查找某个可重复注解，而该注解又确实重复了，那么就会得到 null。这是因为重复注解被包装到了容器注解中。

在这种情况下，应该调用 getAnnotationsByType。这个调用会“遍历”容器，并给出一个重复注解的数组。如果只有一条注解，那么该数组的长度就为 1。通过使用这个方法，你就不用操心如何处理容器注解了。

8.6 源码级注解处理

在上一节中，你看到了如何分析正在运行的程序中的注解。注解的另一种用法是自动处

理源代码以产生更多的源代码、配置文件、脚本或其他任何我们想要生成的东西。

8.6.1 注解处理器

注解处理已经被集成到了 Java 编译器中。在编译过程中，你可以通过运行下面的命令来调用注解处理器。

```
javac -processor ProcessorClassName1,ProcessorClassName2,... sourceFiles
```

编译器会定位源文件中的注解。每个注解处理器会依次执行，并得到它表示感兴趣的注解。如果某个注解处理器创建了一个新的源文件，那么上述过程将重复执行。如果某次处理循环没有再产生任何新的源文件，那么就编译所有的源文件。

 **注释：**注解处理器只能产生新的源文件，它无法修改已有的源文件。

注解处理器通常通过扩展 `AbstractProcessor` 类而实现 `Processor` 接口。你需要指定你的处理器支持的注解，我们的案例如下：

```
@SupportedAnnotationTypes("com.horstmann.annotations.ToString")
@SupportedSourceVersion(SourceVersion.RELEASE_8)
public class ToStringAnnotationProcessor extends AbstractProcessor
{
    public boolean process(Set<? extends TypeElement> annotations,
                          RoundEnvironment currentRound)
    {
        ...
    }
}
```

处理器可以声明具体的注解类型或诸如“`com.horstmann*`”这样的通配符（`com.horstmann` 包及其所有子包中的注解），甚至是“`*`”（所有注解）。

在每一轮中，`process` 方法都会被调用一次，调用时会传递给由这一轮在所有文件中发现的所有注解构成的集，以及包含了有关当前处理轮次的信息的 `RoundEnvironment` 引用。

8.6.2 语言模型 API

应该使用语言模型 API 来分析源码级的注解。与用来呈现类和方法的虚拟机表示形式的反射 API 不同，语言模型 API 让我们可以根据 Java 语言的规则去分析 Java 程序。

编译器会产生一棵树，其节点是实现了 `javax.lang.model.element.Element` 接口及其 `TypeElement`、`VariableElement`、`ExecutableElement` 等子接口的类的实例。这些节点可以类比于编译时的 `Class`、`Field/Parament` 和 `Method/Constructor` 反射类。

本书并不会详细讨论该 API，但我们要强调的是，你需要知道它是如何处理注解的。

- `RoundEnvironment` 通过调用下面的方法交给你一个由特定注解标注过的所有元素构成的集。

```
Set<? extends Element> getElementsAnnotatedWith(Class<? extends Annotation> a)
```

- 在源码级别上等价于 `AnnotatedElement` 接口的是 `AnnotatedConstruct`。使用下面的方法就

可以获得属于给定注解类的单条注解或重复的注解。

```
A getAnnotation(Class<A> annotationType)
A[] getAnnotationsByType(Class<A> annotationType)
```

- TypeElement 表示一个类或接口，而 getEnclosedElements 方法会产生一个由它的域和方法构成的列表。
- 在 Element 上调用 getSimpleName 或在 TypeElement 上调用 getQualifiedName 会产生一个 Name 对象，它可以用 toString 方法转换为一个字符串。

8.6.3 使用注解来生成源码

作为示例，我们将使用注解来减少实现 `toString` 方法时枯燥的编程工作量。我们不能将这些方法放到原来的类中，因为注解处理器只能产生新的类，而不能修改已有的类。

因此，我们将所有方法添加到工具类 `ToStrings` 中：

```
public class ToStrings
{
    public static String toString(Point obj)
    {
        Generated code
    }
    public static String toString(Rectangle obj)
    {
        Generated code
    }
    ...
    public static String toString(Object obj)
    {
        return Objects.toString(obj);
    }
}
```

我们不想使用反射，因此对访问器方法而不是域进行注解：

```
@ToString
public class Rectangle
{
    ...
    @ToString(includeName=false) public Point getTopLeft() { return topLeft; }
    @ToString public int getWidth() { return width; }
    @ToString public int getHeight() { return height; }
}
```

然后，注解处理器应该生成下面的源码：

```
public static String toString(Rectangle obj)
{
    var result = new StringBuilder();
    result.append("Rectangle");
    result.append("[");
    result.append(toString(obj.getTopLeft()));
    result.append(",");
}
```

```

        result.append("width=");
        result.append(toString(obj.getWidth()));
        result.append(",");
        result.append("height=");
        result.append(toString(obj.getHeight()));
        result.append("]");
    }
}

```

其中，灰色的是“模板”代码。下面的框架所描述的方法可以为具有给定的 TypeElement 的类产生 `toString` 方法：

```

private void writeToStringMethod(PrintWriter out, TypeElement te)
{
    String className = te.getQualifiedName().toString();
    Print method header and declaration of string builder
    ToString ann = te.getAnnotation(ToString.class);
    if (ann.includeName())
        Print code to add class name
    for (Element c : te.getEnclosedElements())
    {
        ann = c.getAnnotation(ToString.class);
        if (ann != null)
        {
            if (ann.includeName()) Print code to add field name
            Print code to append toString(obj.methodName())
        }
    }
    Print code to return string
}

```

而下面给出的是注解处理器的 `process` 方法的框架。它会创建助手类的源文件，并为每个被注解标注的类编写类头和一个 `toString` 方法。

```

public boolean process(Set<? extends TypeElement> annotations,
                      RoundEnvironment currentRound)
{
    if (annotations.size() == 0) return true;
    try
    {
        JavaFileObject sourceFile = processingEnv.getFiler().createSourceFile(
            "com.horstmann.annotations.ToStrings");
        try (var out = new PrintWriter(sourceFile.openWriter()))
        {
            Print code for package and class
            for (Element e : currentRound.getElementsAnnotatedWith(ToString.class))
            {
                if (e instanceof TypeElement)
                {
                    TypeElement te = (TypeElement) e;
                    writeToStringMethod(out, te);
                }
            }
            Print code for toString(Object)
        }
    }
}

```

```

        catch (IOException ex)
        {
            processingEnv.getMessager().printMessage(
                Kind.ERROR, ex.getMessage());
        }
    }
    return true;
}

```

对于具体的那些显得有些冗长的代码，可以去查看本书附带的代码。

注意，`process`方法在后续轮次中是用空的注解列表调用的，然后，它会立即返回，因此它并不会多次创建源文件。

首先，编译注解处理器，然后编译并运行测试程序，就像下面这样：

```

javac sourceAnnotations/ToStringAnnotationProcessor.java
javac -processor sourceAnnotations.ToStringAnnotationProcessor rect/*.java
java rect.SourceLevelAnnotationDemo

```

 提示：要想查看轮次，可以用 `-XprintRounds` 标记来运行 `javac` 命令：

```

Round 1:
  input files: {rect.Point, rect.Rectangle,
    rect.SourceLevelAnnotationDemo}
  annotations: [sourceAnnotations.ToString]
  last round: false
Round 2:
  input files: {sourceAnnotations.ToStrings}
  annotations: []
  last round: false
Round 3:
  input files: {}
  annotations: []
  last round: true

```

这个示例演示了工具可以如何获取源文件注解以产生其他文件。生成的文件并非一定要是源文件。注解处理器可以选择生成 XML 描述符、属性文件、Shell 脚本、HTML 文档等。

 注释：有些人建议使用注解来完成一项更繁重的体力活。如果琐碎的获取器和设置器可以自动生成，那岂不是很好？例如，用下面的注解：

```
@Property private String title;
```

来产生下面的方法：

```

public String getTitle() { return title; }
public void setTitle(String title) { this = title; }

```

但是，这些方法需要被添加到同一个类中。这需要编辑源文件而不是产生另一个文件，而这超出了注解处理器的能力范围。我们可以为实现此目的而构建另一个工具，但是这种工具超出了注解的职责范围。注解被设计为对代码项的描述，而不是添加或修改代码的指令。

8.7 字节码工程

你已经看到了我们是怎样在运行期或者在源码级别上对注解进行处理的。还有第 3 种可能：在字节码级别上进行处理。除非将注解在源码级别上删除，否则它们会一直存在于类文件中。类文件格式是归过档的（参阅 <http://docs.oracle.com/javase/specs/jvms/se10/html>），这种格式相当复杂，并且在没有特殊类库支持的情况下，处理类文件具有很大的挑战性。ASM 库就是这样的特殊类库之一，可以从网站 <http://asm.ow2.org> 上获得。

8.7.1 修改类文件

在本小节，我们使用 ASM 向已注解方法中添加日志信息。如果一个方法被这样注解过：

```
@LogEntry(logger=loggerName)
```

那么，在方法的开头部分，我们将添加下面这条语句的字节码：

```
Logger.getLogger(loggerName).entering(className, methodName);
```

例如，如果对 Item 类的 hashCode 方法做了如下注解：

```
@LogEntry(logger="global") public int hashCode()
```

那么，在任何时候调用该方法，都会报告一条与下面打印出来的消息相似的消息：

```
May 17, 2016 10:57:59 AM Item hashCode
FINER: ENTRY
```

为了实现这项任务，我们需要遵循下面几点：

1. 加载类文件中的字节码。
2. 定位所有的方法。
3. 对于每个方法，检查它是不是有一个 LogEntry 注解。
4. 如果有，在方法开头部分添加下面所列指令的字节码：

```
ldc loggerName
invokestatic
    java/util/logging/Logger.getLogger:(Ljava/lang/String;)Ljava/util/logging/Logger;
ldc className
ldc methodName
invokevirtual
    java/util/logging/Logger.entering:(Ljava/lang/String;Ljava/lang/String;)V
```

插入这些字节码看起来相当棘手，不过 ASM 却使它变得相当简单。我们不会详细描述和分析插入字节码的过程。关键之处是程序清单 8-9 中的程序可以编辑一个类文件，并且在已经用 LogEntry 注解标注过的方法的开头部分插入日志调用。

例如，下面展示了应该怎样向程序清单 8-10 中的 Item.java 文件添加记录日志指令，其中 asm 是安装 ASM 库的目录。

```
javac set/Item.java
javac -classpath .:asm/lib/* bytecodeAnnotations/EntryLogger.java
java -classpath .:asm/lib/* bytecodeAnnotations.EntryLogger set.Item
```

在对 Item 类文件进行修改之前和之后分别试运行一下：

```
javap -c set.Item
```

就可以看到在 hashCode、equals 以及 compareTo 方法的开头部分插入的那些指令。

```
public int hashCode();
Code:
0: ldc #85; // String global
2: invokestatic #80;
   // Method
   // java/util/logging/Logger.getLogger:(Ljava/lang/String;)Ljava/util/logging/Logger;
5: ldc #86; //String Item
7: ldc #88; //String hashCode
9: invokevirtual #84;
   // Method java/util/logging/Logger.entering:(Ljava/lang/String;Ljava/lang/String;)V
12: bipush 13
14: aload_0
15: getfield #2; // Field description:Ljava/lang/String;
18: invokevirtual #15; // Method java/lang/String.hashCode:()I
21: imul
22: bipush 17
24: aload_0
25: getfield #3; // Field partNumber:I
28: imul
29: iadd
30: ireturn
```

程序清单 8-11 中的 SetTest 程序会将 Item 对象插入到一个散列集中。当你用修改过的类文件来运行该程序时，会看到下面的日志记录信息：

```
May 17, 2016 10:57:59 AM Item hashCode
FINER: ENTRY
May 17, 2016 10:57:59 AM Item hashCode
FINER: ENTRY
May 17, 2016 10:57:59 AM Item hashCode
FINER: ENTRY
May 17, 2016 10:57:59 AM Item equals
FINER: ENTRY
[[description=Toaster, partNumber=1729], [description=Microwave, partNumber=4104]]
```

当将同一项插入两次时，请注意一下对 equals 的调用。

这个示例显示了字节码工程的强大之处：注解可以用来向程序中添加一些指示，而字节码编辑工具则可以提取这些指示，然后修改虚拟机指令。

程序清单 8-9 bytecodeAnnotations/EntryLogger.java

```
1 package bytecodeAnnotations;
2
3 import java.io.*;
4 import java.nio.file.*;
5
6 import org.objectweb.asm.*;
7 import org.objectweb.asm.commons.*;
8
```

```

9  /**
10 * Adds "entering" logs to all methods of a class that have the LogEntry annotation.
11 * @version 1.21 2018-05-01
12 * @author Cay Horstmann
13 */
14 public class EntryLogger extends ClassVisitor
15 {
16     private String className;
17
18     /**
19      * Constructs an EntryLogger that inserts logging into annotated methods of a given class.
20      * @param cg the class
21      */
22     public EntryLogger(ClassWriter writer, String className)
23     {
24         super(OpcodesASM5, writer);
25         this.className = className;
26     }
27
28     public MethodVisitor visitMethod(int access, String methodName, String desc,
29                                     String signature, String[] exceptions)
30     {
31         MethodVisitor mv = cv.visitMethod(access, methodName, desc, signature, exceptions);
32         return new AdviceAdapter(OpcodesASM5, mv, access, methodName, desc)
33         {
34             private String loggerName;
35
36             public AnnotationVisitor visitAnnotation(String desc, boolean visible)
37             {
38                 return new AnnotationVisitor(OpcodesASM5)
39                 {
40                     public void visit(String name, Object value)
41                     {
42                         if (desc.equals("LbytecodeAnnotations/LogEntry;"))
43                             && name.equals("logger"))
44                         loggerName = value.toString();
45                     }
46                 };
47             }
48
49             public void onMethodEnter()
50             {
51                 if (loggerName != null)
52                 {
53                     visitLdcInsn(loggerName);
54                     visitMethodInsn(INVOKESTATIC, "java/util/logging/Logger", "getLogger",
55                                 "(Ljava/lang/String;)Ljava/util/logging/Logger;", false);
56                     visitLdcInsn(className);
57                     visitLdcInsn(methodName);
58                     visitMethodInsn(INVOKEVIRTUAL, "java/util/logging/Logger", "entering",
59                                 "(Ljava/lang/String;Ljava/lang/String;)V", false);
60                     loggerName = null;
61                 }
62             }

```

```

63     };
64 }
65
66 /**
67 * Adds entry logging code to the given class.
68 * @param args the name of the class file to patch
69 */
70 public static void main(String[] args) throws IOException
71 {
72     if (args.length == 0)
73     {
74         System.out.println("USAGE: java bytecodeAnnotations.EntryLogger classfile");
75         System.exit(1);
76     }
77     Path path = Paths.get(args[0]);
78     var reader = new ClassReader(Files.newInputStream(path));
79     var writer = new ClassWriter(
80         ClassWriter.COMPUTE_MAXS | ClassWriter.COMPUTE_FRAMES);
81     var entryLogger = new EntryLogger(writer,
82         path.toString().replace(".class", "").replaceAll("[/\\\\\\\\]", "."));
83     reader.accept(entryLogger, ClassReader.EXPAND_FRAMES);
84     Files.write(Paths.get(args[0]), writer.toByteArray());
85 }
86 }
```

程序清单 8-10 set/Item.java

```

1 package set;
2
3 import java.util.*;
4 import bytecodeAnnotations.*;
5
6 /**
7 * An item with a description and a part number.
8 * @version 1.01 2012-01-26
9 * @author Cay Horstmann
10 */
11 public class Item
12 {
13     private String description;
14     private int partNumber;
15
16     /**
17      * Constructs an item.
18      * @param aDescription the item's description
19      * @param aPartNumber the item's part number
20      */
21     public Item(String aDescription, int aPartNumber)
22     {
23         description = aDescription;
24         partNumber = aPartNumber;
25     }
26
27     /**
```

```

28     * Gets the description of this item.
29     * @return the description
30     */
31     public String getDescription()
32     {
33         return description;
34     }
35
36     public String toString()
37     {
38         return "[description=" + description + ", partNumber=" + partNumber + "]";
39     }
40
41     @LogEntry(logger = "com.horstmann")
42     public boolean equals(Object otherObject)
43     {
44         if (this == otherObject) return true;
45         if (otherObject == null) return false;
46         if (getClass() != otherObject.getClass()) return false;
47         var other = (Item) otherObject;
48         return Objects.equals(description, other.description) && partNumber == other.partNumber;
49     }
50
51     @LogEntry(logger = "com.horstmann")
52     public int hashCode()
53     {
54         return Objects.hash(description, partNumber);
55     }
56 }
```

程序清单 8-11 set/SetTest.java

```

1 package set;
2
3 import java.util.*;
4 import java.util.logging.*;
5
6 /**
7  * @version 1.03 2018-05-01
8  * @author Cay Horstmann
9  */
10 public class SetTest
11 {
12     public static void main(String[] args)
13     {
14         Logger.getLogger("com.horstmann").setLevel(Level.FINEST);
15         var handler = new ConsoleHandler();
16         handler.setLevel(Level.FINEST);
17         Logger.getLogger("com.horstmann").addHandler(handler);
18
19         var parts = new HashSet<Item>();
20         parts.add(new Item("Toaster", 1279));
21         parts.add(new Item("Microwave", 4104));
22         parts.add(new Item("Toaster", 1279));
```

```

23     System.out.println(parts);
24 }
25 }
```

8.7.2 在加载时修改字节码

在前一节中，已经看到了一个用于编辑类文件的工具。但是，在把另一个工具添加到程序的构建过程中时，会显得笨重不堪。更吸引人的做法是将字节码工程延迟到载入时，即类加载器加载类的时候。

设备 (*instrumentation*) API 提供了一个安装字节码转换器的挂钩。不过，必须在程序的 `main` 方法调用之前安装这个转换器。通过定义一个代理，即被加载用来按照某种方式监视程序的一个类库，就可以处理这个需求。代理代码可以在 `premain` 方法中执行初始化。

下面是构建一个代理所需的步骤：

1. 实现一个具有下面这个方法的类：

```
public static void premain(String arg, Instrumentation instr)
```

当加载代理时，此方法会被调用。代理可以获取一个单一的命令行参数，该参数是通过 `arg` 参数传递进来的。`instr` 参数可以用来安装各种各样的挂钩。

2. 制作一个清单文件 `EntryLoggingAgent.mf` 来设置 `Premain-Class` 属性。例如：

```
Premain-Class: bytecodeAnnotations.EntryLoggingAgent
```

3. 将代理代码打包，并生成一个 JAR 文件，例如：

```
javac -classpath .:asm/lib/* bytecodeAnnotations/EntryLoggingAgent.java
jar cvfm EntryLoggingAgent.jar bytecodeAnnotations/EntryLoggingAgent.mf \
bytecodeAnnotations/Entry*.class
```

为了运行一个具有该代理的 Java 程序，需要使用下面这个命令行选项：

```
java -javaagent:AgentJARFile=agentArgument . . .
```

例如，运行具有日志代理的 `SetTest` 程序需调用：

```
javac set/SetTest.java
java -javaagent:EntryLoggingAgent.jar=set.Item -classpath .:asm/lib/* set.SetTest
```

`Item` 参数是代理应该修改的类的名称。

程序清单 8-12 展示了这个代理的代码。该代理安装了一个类文件转换器，这个转换器首先检验类名是否与代理参数相匹配。如果匹配，那么它会利用上一节那个 `EntryLogger` 类修改字节码。不过，修改过的字节码并不保存成文件。相反地，转换器只是将它们返回，以加载到虚拟机中（参见图 8-3）。换句话说，这项技术实现的是“即时”(just in time) 字节码修改。

程序清单 8-12 bytecodeAnnotations/EntryLoggingAgent.java

```

1 package bytecodeAnnotations;
2
3 import java.lang.instrument.*;
4
```

```

5 import org.objectweb.asm.*;
6 /**
7 * @version 1.11 2018-05-01
8 * @author Cay Horstmann
9 */
10 public class EntryLoggingAgent
11 {
12     public static void premain(final String arg, Instrumentation instr)
13     {
14         instr.addTransformer((loader, className, cl, pd, data) ->
15         {
16             if (!className.replace("/", ".").equals(arg)) return null;
17             var reader = new ClassReader(data);
18             var writer = new ClassWriter(
19                 ClassWriter.COMPUTE_MAXS | ClassWriter.COMPUTE_FRAMES);
20             var el = new EntryLogger(writer, className);
21             reader.accept(el, ClassReader.EXPAND_FRAMES);
22             return writer.toByteArray();
23         });
24     }
25 }
26 }
```

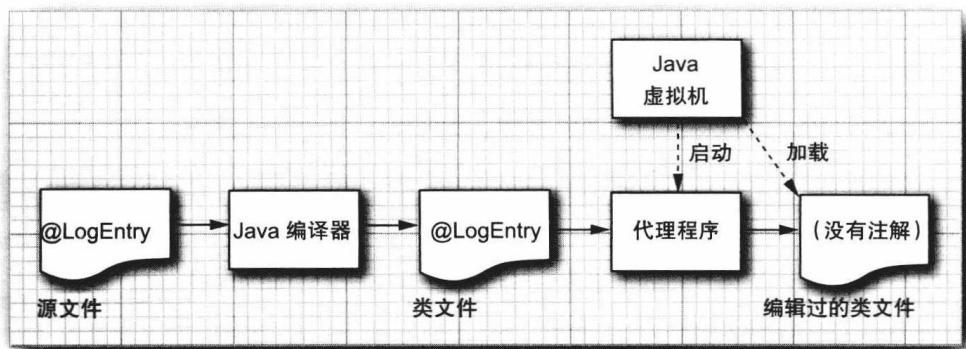


图 8-3 在加载时修改类

在本章，你已经学习到了以下的知识：

- 怎样向 Java 程序中添加注解。
- 怎样设计你自己的注解接口。
- 怎样实现可以利用注解的工具。

你已经看到了三种处理代码的技术：编写脚本、编译 Java 程序和处理注解。前两种技术十分简单。而另一方面，构建注解工具可能会很复杂，但这并非是大多数开发者都需要解决的问题。本章向你介绍了一些背景知识，有助于你去理解可能会碰到的注解工具内部工作机制，但这些背景知识可能会挫伤你自行开发工具的积极性。

下一章将讨论 Java 平台模块系统，它是 Java 9 的关键特性，是促进 Java 平台向前发展的重要动力。

第9章 Java 平台模块系统

- ▲ 模块的概念
- ▲ 对模块命名
- ▲ 模块化的“Hello, World！”程序
- ▲ 对模块的需求
- ▲ 导出包
- ▲ 模块的 JAR
- ▲ 模块和反射式访问
- ▲ 自动模块
- ▲ 不具名模块
- ▲ 用于迁移的命令行标志
- ▲ 传递的需求和静态的需求
- ▲ 限定导出和开放
- ▲ 服务加载
- ▲ 操作模块的工具

封装是面向对象编程的一个重要特性。类的声明由公有接口和私有实现构成，类可以通过只修改实现而不影响其用户的方式而得以演化。模块系统为编程带来了大致相同的益处。模块使类和包可以有选择性地获取，从而使得模块的演化可以受控。

多个现有的 Java 模块系统都依赖于类加载器来实现类之间的隔离。但是，Java 9 引入了一个由 Java 编译器和虚拟机支持的新系统，称为 Java 平台模块系统。它被设计用来模块化基于 Java 平台的大型代码基。如果愿意，也可以使用这个系统来模块化我们自己的应用程序。

无论是否在自己的应用程序中使用 Java 平台模块，都可能会受到模块化的 Java 平台的影响。本章将展示如何声明和使用 Java 平台模块。你还会学习到如何迁移你的应用程序，使其能够与模块化的 Java 平台和第三方模块一起工作。

9.1 模块的概念

在面向对象编程中，基础的构建要素就是类。类提供了封装，私有特性只能被具有明确访问权限的代码访问，即，只能被其所属类中的方法访问，这使得对访问权限的推断成为可能。如果某个私有变量发生了变化，那么我们就会发现一系列可能出错的方法。如果需要修改私有表示，那么就需要知道哪些方法会受到影响。

在 Java 中，包提供了更高一级的组织方式，包是类的集合。包也提供了一种封装级别，具有包访问权限的所有特性（无论是公有的还是私有的）都只能被同一个包中的方法访问。

但是，在大型系统中，这种级别的访问控制仍显不足。所有公有特性（即在包的外部也可以访问的特性）可以从任何地方访问。假设我们想要修改或剔除一个很少使用的特性，如果它是公有的，那么就没有办法推断这个变化所产生的影响。

Java 平台的设计者们面对的就是这种情况。过去 20 年中，JDK 呈跨越式发展，但是有

些特性现在明显过时了。有大家喜欢提到的例子，即 CORBA。你最后一次使用它是什么时候？但是 `org.omg.corba` 包仍旧打包在每一个 JDK 中，直至 Java 10。到了 Java 11，仍旧需要这个包的那些极少量的人就必须将所需的 JAR 文件自己添加到他们的项目中了。

`java.awt` 的情况又如何呢？服务器端的应用程序并不需要它，对吗？但是，`java.awt.DataFlavor` 类在 SOAP 的实现中仍在使用，这是一种基于 XML 的 Web 服务协议。

Java 平台的设计者们在面对规模超大且盘根错节的代码时，认为他们需要一种能够提供更多控制能力的构建机制。他们研究了现有的模块系统（例如 OSGi），发现它们都不适用于他们的问题。于是，他们设计了一个新的系统，称为 Java 平台模块系统，现在成了 Java 语言和虚拟机的一部分。这个系统已经成功地用于 Java API 的模块化，如果愿意，也可以使用这个系统来模块化我们自己的应用程序。

一个 Java 平台模块包含：

- 一个包集合
- 可选地包含资源文件和像本地库这样的其他文件
- 一个有关模块中可访问的包的列表
- 一个有关这个模块依赖的所有其他模块的列表

Java 平台在编译时和在虚拟机中都强制执行封装和依赖。

为什么在我们自己的程序中要考虑使用 Java 平台模块系统而不是传统的使用类路径上的 JAR 文件呢？因为这样做有以下两个优点。

1. 强封装：我们可以控制哪些包是可访问的，并且无须操心去维护那些我们不想开放给公众去访问的代码。
2. 可靠的配置：我们可以避免诸如类重复或丢失这类常见的类路径问题。

还有一些有关 Java 平台模块系统的话题我们没有涉及，例如模块的版本管理。当前还不支持指定要求使用模块的具体版本，或者在同一个程序中使用某个模块的多个版本。这些特性可能正是人们所期望的，但是如果需要用到它们，就必须使用 Java 平台模块系统之外的机制。

9.2 对模块命名

模块是包的集合。模块中的包名无须彼此相关。例如，`java.sql` 模块中就包含了 `java.sql`、`javax.sql` 和 `javax.transaction.xa` 这几个包。并且，正如这个例子所示，模块名和包名相同是完全可行的。

就像路径名一样，模块名是由字母、数字、下划线和句点构成的。而且，和路径名一样，模块之间没有任何层次关系。如果有一个模块是 `com.horstmann`，另一个模块是 `com.horstmann.corejava`，那么就模块系统而言，它们是无关的。

当创建供他人使用的模块时，重要的是要确保它的名字是全局唯一的。我们期望大多数的模块名都遵循“反向域名”惯例，就像包名一样。

命名模块最简单的方式就是按照模块提供的顶级包来命名。例如，SLF4J 日志记录外观有一个 org.slf4j 模块，其中包含的包为 org.slf4j、org.slf4j.spi、org.slf4j.event 和 org.slf4j.helpers。

这个惯例可以防止模块中产生包名冲突，因为任何给定的模块都只能被放到一个模块中。如果模块名是唯一的，并且包名以模块名开头，那么包名也就是唯一的。

我们可以使用更短的模块名来命名不打算给其他程序员使用的模块，例如包含某个应用程序的模块。只是为了展示这样做可行，本章就使用了这种方式，那些貌似应该成为库代码的模块都具有像 com.horstmann.util 这样的名字，而包含程序（具有 main 方法的类）的模块都具有像 v2ch09.hellomod 这样很容易记忆的名字。

 **注释：**模块名只用于模块声明中。在 Java 类的源文件中，永远都不应该引用模块名，而是应该按照一如既往的方式去使用包名。

9.3 模块化的“Hello, World!”程序

让我们把传统的“Hello, World!”程序转换为一个模块。首先，我们需要将这个类放到一个包中，“不具名的”包是不能包含在模块中的。下面是代码：

```
package com.horstmann.hello;

public class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("Hello, Modular World!");
    }
}
```

到目前为止，还没有任何东西有变化。为了创建包含这个包的 v2ch09.hellomod 模块，需要添加一个模块声明，可以将其置于名为 module.info.java 的文件中，该文件位于基目录中（即，与包含 com 目录的目录相同）。按照惯例，基目录的名字与模块名相同。

```
v2ch09.hellomod/
└ module-info.java
com/
└ horstmann/
  └ hello/
    └ HelloWorld.java
```

module-info.java 文件包含模块声明：

```
module v2ch09.hellomod
{ }
```

这个模块声明之所以为空，是因为该模块没有任何可以向其他人提供的内容，它也不需要依赖任何东西。

现在，按照往常一样编译它：

```
javac v2ch09.hellomod/module-info.java v2ch09.hellomod/com/horstmann/hello/HelloWorld.java
```

`module.info.java` 这个文件看起来与 Java 资源文件不同，当然，也不可能存在名为 `module-info` 的类，因为类名不能包含连字符。关键词 `module` 和在下一节将会看到的 `requires`、`exports` 等关键词都是“限定关键词”，即只在模块声明中具有特殊含义。这个文件会以二进制形式编译到包含该模块定义的类文件 `module-info.class` 中。

为了让这个程序作为模块化应用程序来运行，需要指定模块路径，它与类路径相似，但是包含的是模块。还需要以模块名 / 类名的形式指定主类：

```
java --module-path v2ch09.hellomod --module v2ch09.hellomod/com.horstmann.hello.HelloWorld
```

也可以不使用 `--module-path` 和 `-module`，而是使用单字母选项 `-p` 和 `-m`：

```
java -p v2ch09.hellomod -m v2ch09.hellomod/com.horstmann.hello.HelloWorld
```

无论哪种方式，都会显示问候语 “Hello, Module World!”，证明我们成功地模块化了第一个应用程序。

 **注释：**在编译这个模块时，会获得一条警告消息：

```
warning: [module] module name component v2ch09 should avoid terminal digits
```

这条警告意在建议程序员不要给模块名添加版本号。你可以忽略这个警告，或者用注解来抑制它：

```
@SuppressWarnings("module")
module v2ch09.hellomod
{
}
```

在这一点上，`module` 声明就像类声明一样：可以对其进行注解。（注解类型必须具有值为 `ElementType.MODULE` 的 target。）

9.4 对模块的需求

让我们创建一个新的模块 `v2ch09.requiremod`，其中使用一个 `JOptionPane` 对象展示了消息“Hello, Modular World”：

```
package com.horstmann.hello;

import javax.swing.JOptionPane;

public class HelloWorld
{
    public static void main(String[] args)
    {
        JOptionPane.showMessageDialog(null, "Hello, Modular World!");
    }
}
```

现在，编译会失败并报下面的消息：

```
error: package javax.swing is not visible
  (package javax.swing is declared in module java.desktop,
  but module v2ch09.requiremod does not read it)
```

JDK 已经被模块化了，并且 `javax.swing` 包现在包含在 `java.desktop` 模块中。我们的模块需要声明它依赖于这个模块：

```
module v2ch09.requiremod
{
    requires java.desktop;
}
```

模块系统的设计目标之一就是模块需要明确它们的需求，使得虚拟机可以确保在启动程序之前所有的需求都得以满足。

在前一节中，并没有产生明确的需求，因为我们只用到了 `java.lang` 和 `java.io` 包。这些包都包含在默认需要的 `java.base` 模块中。

注意，我们的 `v2ch09.requiremod` 模块只列出了它自己的模块需求。它需要 `java.desktop` 模块，这样它才能使用 `javax.swing` 包。`java.desktop` 模块自身声明了它需要其他三个包，即 `java.datatransfer`、`java.prefs` 和 `java.xml`。

图 9-1 展示了一张模块图，图中的节点是模块，而图中的边，也就是连接节点的箭头，要么声明了需求，要么在没有声明任何需求时表示需要 `java.base`。

在模块图中不能有环，即，一个模块不能直接或间接地对自己产生依赖。

模块不会自动地将访问权限传递给其他模块。在我们的示例中，`java.desktop` 模块声明它需要 `java.prefs`，而 `java.prefs` 模块声明它需要 `java.xml`，但是这并不会赋予 `java.desktop` 使用来自 `java.xml` 模块中的包的权力。按照数学术语描述，`require` 不是“传递性”的。通常，这种行为正是我们想要的，因为它使得需求必须明确化，但是正如你将会在 9.11 节中看到的，在某些情况下，可以放松这条限制。

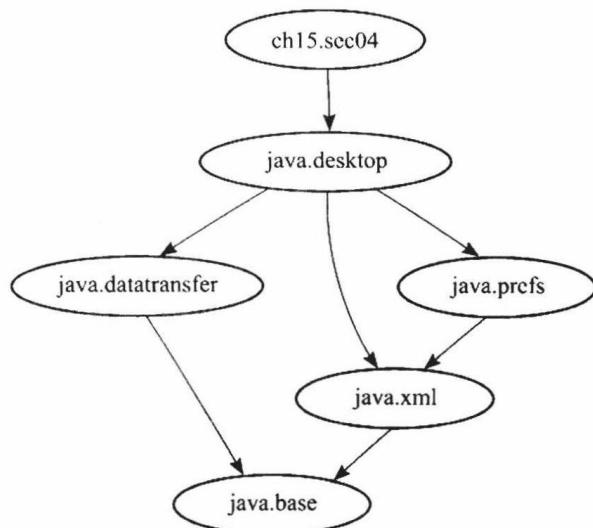


图 9-1 Swing 应用程序“Hello, Modular World!”的模块图

注释：本节开头部分给出的错误消息声明我们的 `v2ch09.requiremod` 模块没有“读入”`java.desktop` 模块。按照 Java 模块系统的用语，模块 *M* 会在下列情况下读入模块 *N*：

1. *M* 需要 *N*
2. *M* 需要某个模块，而该模块传递性地需要 *N*（参阅 9.11 节）
3. *N* 是 *M* 或 `java.base`

9.5 导出包

在前一节中，我们看到一个模块如果想要使用其他模块中的包，就必须声明需要该模块。但是，这并不会自动使得所需模块中所有的包都可用。模块可以用 `exports` 关键词来声明它的哪些包可用。例如，下面是 `java.xml` 模块的模块声明中的一部分：

```
module java.xml
{
    exports javax.xml;
    exports javax.xml.catalog;
    exports javax.xml.datatype;
    exports javax.xml.namespace;
    exports javax.xml.parsers;

    ...
}
```

这个模块让许多包都可用，但是通过不导出其他的包而隐藏了它们（例如 `jdk.xml.internal`）。

当包被导出时，它的 `public` 和 `protected` 的类和接口，以及 `public` 和 `protected` 的成员，在模块的外部也是可以访问的（如往常一样，`protected` 的类型和成员只有在子类中才是可访问的）。

但是，没有导出的包在其自己的模块之外是不可访问的，这与 Java 模块化之前很不相同。在过去，我们可以使用任何包中公有的类，尽管它可能并非公有 API 的一部分。例如，当公有 API 没有提供相对应的适合的功能时，通常会推荐使用像 `sun.misc.BASE64Encoder` 或 `com.sun.rowset.CachedRowSetImpl` 这样的类。

现在，不能再访问 Java 平台 API 中未导出的包了，因为所有的这些包都包含在模块的内部。因此，有些程序不能再用 Java 9 来运行了。当然，从来没有人承诺过会让非公有的 API 一直保持可用，因此大家不应该对此感到震惊。

让我们在一个简单场景中使用导出机制。我们将准备一个 `com.horstmann.greet` 模块，它会导出一个名字也是 `com.horstmann.greet` 的包，这遵循了向他人提供代码的模块应该按照其内部的顶层包来命名的惯例。还有一个名为 `com.horstmann.greet.internal` 的包，我们并不会导出它。

公有的 Greeter 接口在第一个包中。

```
package com.horstmann.greet;

public interface Greeter
{
    static Greeter newInstance()
    {
        return new com.horstmann.greet.internal.GreeterImpl();
    }

    String greet(String subject);
}
```

第二个包有一个实现了该接口的类。这个类是公有的，因为它需要在第一个包中是可访问的：

```
package com.horstmann.greet.internal;
```

```

import com.horstmann.greet.Greeter;

public class GreeterImpl implements Greeter
{
    public String greet(String subject)
    {
        return "Hello, " + subject + "!";
    }
}

```

com.horstmann.greet 模块包含这两个包，但是只会导出第一个包：

```

module com.horstmann.greet
{
    exports com.horstmann.greet;
}

```

第二个包在模块外部是不可访问的。

我们将应用程序放到第二个包中，它需要用到第一个模块：

```

module v2ch09.exportedpkg
{
    requires com.horstmann.greet;
}

```

 **注释：**exports 语句跟在包名后面，而 requires 语句跟在模块名后面。

现在，我们的应用程序将使用 Greeter 来获取问候语：

```

package com.horstmann.hello;

import com.horstmann.greet.Greeter;

public class HelloWorld
{
    public static void main(String[] args)
    {
        Greeter greeter = Greeter.newInstance();
        System.out.println(greeter.greet("Modular World"));
    }
}

```

下面是这两个模块的源文件结构：

```

com.horstmann.greet
├ module-info.java
└ com
  └ horstmann
    └ greet
      └ Greeter.java
      └ internal
        └ GreeterImpl.java
v2ch09.exportedpkg
├ module-info.java
└ com
  └ horstmann
    └ hello
      └ HelloWorld.java

```

为了构建这个应用程序，首先要编译 com.horstmann.greet 模块：

```
javac com.horstmann.greet/module-info.java \
com.horstmann.greet/com/horstmann/greet/Greeter.java \
com.horstmann.greet/com/horstmann/greet/internal/GreeterImpl.java
```

然后，用模块路径上的第一个模块来编译这个应用程序模块：

```
javac -p com.horstmann.greet v2ch09.exportedpkg/module-info.java \
v2ch09.exportedpkg/com/horstmann/hello/HelloWorld.java
```

最后，用模块路径上的这两个模块来运行这个程序：

```
java -p v2ch09.exportedpkg:com.horstmann.greet \
-m v2ch09.exportedpkg/com.horstmann.hello.HelloWorld
```

 提示：如果要用 Eclipse 来构建这个应用程序，需要为每一个模块建立一个单独的工程。在 v2ch09.exportedpkg 项目中，编辑项目属性。在 Projects 配置页上，添加 com.horstmann.greet 模块到模块路径中，参阅图 9-2。

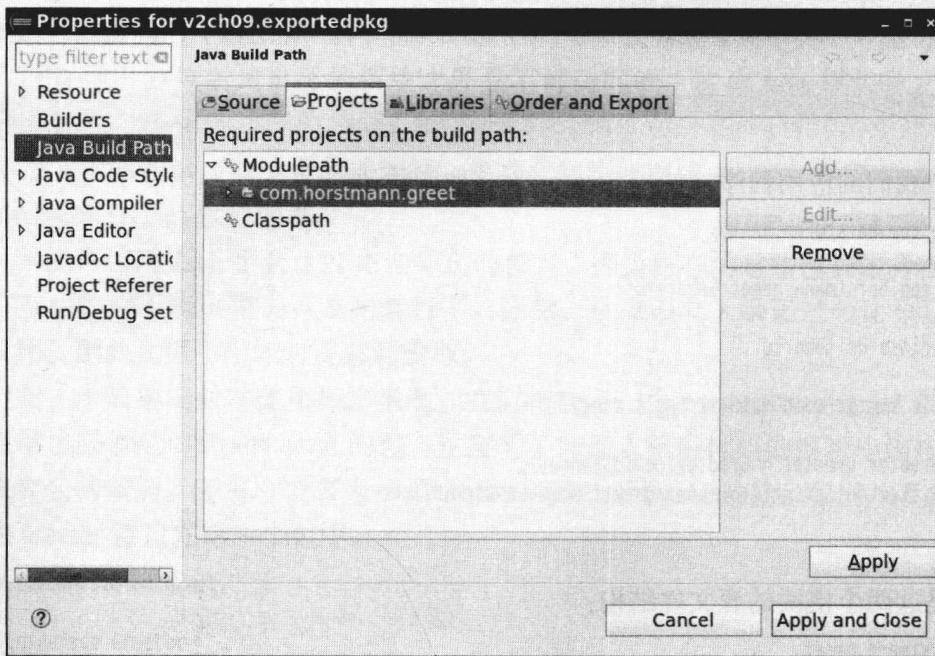


图 9-2 添加依赖的模块到 Eclipse 项目中

现在，你已经看到了构成 Java 平台模块系统基础的 requires 和 exports 语句。正如你所见，模块系统在概念上很简单。模块指定了它们需要哪些模块，以及它们可以向其他模块提供哪些包。9.12 节将展示 exports 语句的一个次要变体。

 警告：模块没有作用域的概念。不能在不同的模块中放置两个具有相同名字的包。即使是隐藏的包（即，不会导出的包），情况也是如此。

9.6 模块化的 JAR

到目前为止，我们直接将模块编译到了源代码的目录树中。很明显，这无法满足部署的要求。模块可以通过将其所有的类都置于一个 JAR 文件中而得以部署，其中 `module-info.class` 在 JAR 文件的根部。这样的 JAR 文件被称为模块的 JAR。

要想创建模块化的 JAR 文件，只需以通常的方式使用 `jar` 工具。如果有多个包，那么最好是用 `-d` 选项来编译，这样可以将类文件置于单独的目录中，如果该目录不存在，则会创建该目录。然后，在收集这些类文件时使用 `-c` 选项的 `jar` 命令来修改该目录。

```
javac -d modules/com.horstmann.greet $(find com.horstmann.greet -name *.java)
jar -cvf com.horstmann.greet.jar -C modules/com.horstmann.greet .
```

如果你使用的是像 Maven、Ant 或 Gradle 这样的构建工具，那么只需按照你惯用的方式来构建 JAR 文件。只要 `module-info.class` 包含在内，就可以得到该模块的 JAR 文件。

然后，在模块路径中包含该模块化的 JAR，该模块就会被加载。

！ 警告：在过去，包中的类有时会分布在多个 JAR 文件中。(这种包被称为“分离包”。)这可能从来就不是一个好主意，对于模块来说也不可能是个好主意。

就像常规的 JAR 文件一样，可以指定模块化的 JAR 中的主类：

```
javac -p com.horstmann.greet.jar \
-d modules/v2ch09.exportedpkg $(find v2ch09.exportedpkg -name *.java)
jar -c -v -f v2ch09.exportedpkg.jar -e com.horstmann.hello.HelloWorld \
-C modules/v2ch09.exportedpkg .
```

当启动该程序时，可以指定包含主类的模块：

```
java -p com.horstmann.greet.jar:v2ch09.exportedpkg.jar -m v2ch09.exportedpkg
```

在创建 JAR 文件时，可以选择指定版本号。使用 `--module-version` 选项，以及在 JAR 文件名上添加 @ 和版本号：

```
jar -c -v -f com.horstmann.greet@1.0.jar --module-version 1.0 -C com.horstmann.greet .
```

正如已经讨论过的，Java 平台模块系统并不会使用版本号来解析模块，但是可以通过其他工具和框架来查询版本号。

注释：可以通过反射 API 找到版本号。在我们的示例中：

```
Optional<String> version = Greeter.class.getModule().getDescriptor().rawVersion();
```

将产生一个包含版本号字符串 "1.0" 的 `Optional`。

注释：等价于类加载器的模块是一个层。Java 平台模块系统会将 JDK 模块和应用程序模块加载到启动层（boot layer）。程序还可以使用分层 API 加载其他模块（本书不会讨论该 API）。这种程序可以选择考虑模块的版本。Java 期望像 Java EE 应用服务器这样的程序的开发者会利用分层 API 来提供对模块的支持。

 提示：如果想要加载模块到 JShell 中，需要将 JAR 包含在模块路径中，并使用 `--add-modules` 选项：

```
jshell --module-path com.horstmann.greet@1.0.jar --add-modules com.horstmann.greet
```

9.7 模块和反射式访问

在前面的章节中，我们看到了模块系统是如何强制执行封装的。模块只能访问显式地由其他包导出的包。在过去，总是可以通过使用反射来克服令人讨厌的访问权限问题。正如在卷 I 第 5 章中看到的，反射可以访问任何类的私有成员。

但是，在模块化的世界中，这条路再也行不通了。如果一个类在某个模块中，那么对非公有成员的反射式访问将失败。特别是，回忆一下我们是如何访问私有域的：

```
Field f = obj.getClass().getDeclaredField("salary");
f.setAccessible(true);
double value = f.getDouble(obj);
f.setDouble(obj, value * 1.1);
```

`f.setAccessible(true)` 调用会成功，除非安全管理器不允许对私有域的访问。但是，使用安全管理器来运行 Java 应用程序并不常见，并且有许多使用反射式访问的库。典型的例子包括像 JPA 这样的对象 - 关系映射器，它们会自动地将对象持久化到数据库中，以及在对象和 XML 或 JSON 之间转换的库中，例如 JAXB 和 JSON-B。

如果使用这种库，并且还想使用模块，那么就必须格外小心。为了演示这个问题，让我们将卷 I 第 5 章中的 `ObjectAnalyzer` 类放到 `com.horstmann.util` 模块中。这个类有一个 `toString` 方法，可以使用反射机制来打印出对象的域。

单独的 `v2ch09.openpkg` 模块包含一个简单 `Country` 类：

```
package com.horstmann.places;

public class Country
{
    private String name;
    private double area;

    public Country(String name, double area)
    {
        this.name = name;
        this.area = area;
    }
    // ...
}
```

下面的短程序演示了如何分析 `Country` 对象：

```
package com.horstmann.places;

import com.horstmann.util.*;
```

```

public class Demo
{
    public static void main(String[] args) throws ReflectiveOperationException
    {
        var belgium = new Country("Belgium", 30510);
        var analyzer = new ObjectAnalyzer();
        System.out.println(analyzer.toString(belgium));
    }
}

```

现在编译模块和 Demo 程序：

```

javac com.horstmann.util/module-info.java \
    com.horstmann.util/com/horstmann/util/ObjectAnalyzer.java
javac -p com.horstmann.util v2ch09.openpkg/module-info.java \
    v2ch09.openpkg/com/horstmann/places/*.java
java -p v2ch09.openpkg:com.horstmann.util -m v2ch09.openpkg/com.horstmann.places.Demo

```

该程序会以下面的异常而失败：

```

Exception in thread "main" java.lang.reflect.InaccessibleObjectException:
  Unable to make field private java.lang.String com.horstmann.places.Country.name
  accessible: module v2ch09.openpkg does not "opens com.horstmann.places" to module
  com.horstmann.util

```

当然，按照纯理论来说，破坏对象的封装并窥视其私有成员是错误的。但是像对象 - 关系映射或 XML/JSON 绑定这样的机制应用非常广泛，使得模块系统必须接纳它们。

通过使用 `opens` 关键词，模块就可以打开包，从而启动对给定包中的类的所有实例进行反射式访问。下面是我们的模块必须执行的操作：

```

module v2ch09.openpkg
{
    requires com.horstmann.util;
    opens com.horstmann.places;
}

```

有了这样的变化，`ObjectAnalyzer` 就可以正确地工作了。

模块可以像下面这样声明为 Open (开放的)：

```

open module v2ch09.openpkg
{
    requires com.horstmann.util;
}

```

开放的模块可以授权对其所有包的运行时访问，就像所有的包都用 `exports` 和 `opens` 声明过一样。但是，在运行时只有显式导出的包是可访问的。开放模块将模块系统编译时的安全性和经典的授权许可的运行时行为结合在一起。

回忆一下卷 I 第 5 章，JAR 文件除了类文件和清单外，还可以包含文件资源，它们可以被 `Class.getResourceAsStream` 方法加载，现在还可以被 `Module.getResourceAsStream` 加载。如果资源存储在匹配模块的某个包的目录中，那么这个包必须对调用者是开放的。在其他目录中的资源，以及类文件和清单，可以被任何人读取。

注释：作为更贴近实际的例子，我们把 Country 对象转换为 XML 或 JSON。Java 9 和 10 中包含用于转换为 XML 的 java.xml.bind 模块。该模块已经从 Java 11 中移除了（同时被移除的模块还有 java.activation、java.corba、java.transaction、java.xml.ws 和 java.xml.ws.annotation）。这些模块包含的包也是 Jakarta EE（之前的 Java EE）规范的一部分，其中的 API 比 Java SE 内涵更广。如果 JDK 中包含有与其冲突的包，那么企业应用服务器不能被模块化。遗憾的是，到本书撰写之时，还没有用于 XML 绑定的模块化替代物出现。

但是，对于 JSON-B 的实现，如果我们从源码构建它，它就会提供模块化的 JAR 文件。可以期望的是，在你阅读这部分内容时，这些 JAR 文件已经进入了 Maven Central。将这些 JAR 文件放到模块路径上，然后运行 com.horstmann.places.Demo2 程序，当 com.horstmann.places 包开放时，向 JSON 的转换就会成功。

注释：未来的库可能会使用变量句柄而不是反射来读写域。VarHandle 类似于 Field。我们可以使用它来读写具体类的任何实例的具体域。但是，为了获得 VarHandle 对象，库代码需要一个 Lookup 对象：

```
public Object getFieldValue(Object obj, String fieldName, Lookup lookup)
    throws NoSuchFieldException, IllegalAccessException
{
    Class<?> cl = obj.getClass();
    Field field = cl.getDeclaredField(fieldName);
    VarHandle handle = MethodHandles.privateLookupIn(cl, lookup)
        .unreflectVarHandle(field);
    return handle.get(obj);
}
```

只要该模块中生成的 Lookup 对象拥有对该域的访问权，这段代码就可以工作。在模块中的某些方法可以直接调用 MethodHandles.lookup()，它会产生一个封装了调用者访问权限的对象。在这种方式下，一个模块可以赋予另一个模块访问私有成员的权限。在实践中，需要解决如何以麻烦最少的方式赋予这些权限的问题。

9.8 自动模块

现在你知道了如何使用 Java 平台模块系统。如果从全新的项目开始，其中所有的代码都由我们自己编写，那么就可以设计模块、声明模块依赖关系，并将应用程序打包成模块化的 JAR 文件。

但是，这是一种非常罕见的场景，几乎所有的项目都依赖于第三方的库。当然，我们可以等到所有库的提供商都将库演化成模块，然后再模块化我们自己的代码。

但是如果等不及怎么办呢？Java 平台模块系统提供了两种机制来填补将当今的前模块化世界与完全模块化应用程序割裂开来的鸿沟：自动化模块和不具名模块。

如果是为了迁移，我们可以通过把任何 JAR 文件置于模块路径的目录而不是类路径的目录中，实现将其转换成一个模块。模块路径上没有 `module-info.class` 文件的 JAR 被称为自动模块。自动模块具有下面的属性：

1. 模块隐式地包含对其他所有模块的 `requires` 子句。
2. 其所有包都被导出，且是开放的。
3. 如果在 JAR 文件清单 `META-INF/MANIFEST.MF` 中具有键为 `Automatic-Module-Name` 的项，那么它的值会变为模块名。
4. 否则，模块名将从 JAR 文件文件名中获得，将文件名中尾部的版本号删除，并将非字母数字的字符替换为句点。

前两条规则表明自动模块中的包的行为和在类路径上一样。使用模块路径的原因是为了让其他模块受益，使得它们可以表示对这个模块的依赖关系。

例如，假设我们正在实现一个处理 CSV 文件的模块，并使用了 Apache Commons CSV 库。我们想要在 `module-info.java` 文件中表示模块需要依赖 Apache Commons CSV。

如果在模块路径中添加 `commons-csv-1.5.jar`，那么我们的模块就可以引用这个模块了。它的名字是 `commons.csv`，因为去掉了尾部版本号 `-1.5`，而非字母数字字符 `-` 被替换成了句点。

这个名字也许算是一个可接受的模块名，因为 Commons CSV 人们耳熟能详，其他人也不太可能会用这个名字来命名其他的模块。但是，如果这个 JAR 文件的维护者同意保留反向域名，使用更好的顶级包名 `org.apache.commons.csv` 作为模块名，那会显得更好。他们只需在 JAR 中的 `META-INF/MANIFEST.MF` 文件里添加一行：

```
Automatic-Module-Name: org.apache.commons.csv
```

最终，我们期望他们能够在 `module-info.java` 中添加保留的模块名将这个 JAR 文件转换成一个真正的模块，而每个用该模块名引用了这个 CSV 模块的模块也都能够继续工作。

注释：模块的迁移计划是一项伟大的社会实验，没有人知道它是否能够顺利实施。在将第三方的 JAR 放到模块路径之前，请检查它们是否是模块化的。如果不是，那它们的清单是否有模块名；如果没有，仍旧需要将这样的 JAR 转换成自动模块，但是要准备好以后更新该模块名。

在撰写本书时，Commons CSV JAR 文件的 1.5 版本还没有模块描述符或自动模块名。尽管如此，它在模块路径上工作良好。我们可以从 <https://commons.apache.org/proper/commons-csv> 处下载这个库，解压并将 `commons-csv-1.5.jar` 放到 `v2ch09.automod` 模块的目录中。这个模块包含了一个很简单的从 CSV 文件中读取国家数据的程序：

```
package com.horstmann.places;

import java.io.*;
import org.apache.commons.csv.*;

public class CSVDemo
{
```

```

public static void main(String[] args) throws IOException
{
    var in = new FileReader("countries.csv");
    Iterable<CSVRecord> records = CSVFormat.EXCEL.withDelimiter(';')
        .withHeader().parse(in);
    for (CSVRecord record : records)
    {
        String name = record.get("Name");
        double area = Double.parseDouble(record.get("Area"));
        System.out.println(name + " has area " + area);
    }
}
}

```

因为我们将 commons-csv-1.5.jar 用作自动模块，所以我们要声明需要它：

```

@SuppressWarnings("module")
module v2ch09.automod
{
    requires commons.csv;
}

```

下面是编译和运行该程序的命令：

```

javac -p v2ch09.automod:commons-csv-1.5.jar \
    v2ch09.automod/com/horstmann/places/CSVDemo.java \
    v2ch09.automod/module-info.java
java -p v2ch09.automod:commons-csv-1.5.jar \
    -m v2ch09.automod/com.horstmann.places.CSVDemo

```

9.9 不具名模块

任何不在模块路径中的类都是不具名模块的一部分。从技术上说，可能会有多个不具名模块，但是它们合起来看就像是单个不具名的模块。与自动模块一样，不具名模块可以访问所有其他的模块，它的所有包都会被导出，并且都是开放的。

但是，没有任何明确模块可以访问不具名的模块。(明确模块是指既不是自动模块也不是不具名模块的模块，即，`module-info.class` 在模块路径上的模块。) 换句话说，明确模块总是可以避免“类路径的坑”。

例如，考虑前一节的程序，假设将 commons-csv-1.5.jar 放到类路径而不是模块路径上：

```

java --module-path v2ch09.automod \
    --class-path commons-csv-1.5.jar \
    -m v2ch09.automod/com.horstmann.places.CSVDemo

```

现在，这个程序将无法启动：

```

Error occurred during initialization of boot layer
java.lang.module.FindException: Module commons.csv not found, required by v2ch09.automod

```

因此，迁移到 Java 平台模块系统必须按照自底向上的方式处理：

1. Java 平台自身被模块化。
2. 接下来，库被模块化，要么通过使用自动模块，要么将它们转换为明确模块。

3. 一旦应用程序使用的所有库都被模块化，就可以将应用程序的代码转换为一个模块。

注释：自动模块可以读取不具名模块，因此它们的依赖关系放在类路径中。

9.10 用于迁移的命令行标识

即使我们的程序没有使用模块，在使用 Java 9 或更新的版本时，我们也无法逃离模块化的世界。即使应用程序的代码位于不具名模块的类路径上，并且所有的包都被导出且开放，它也需要与模块化的 Java 平台交互。

到了 Java 11，默认行为是允许非法的模块访问，但是会在每种违规行为第一次出现时在控制台上显示一条警告消息。在 Java 未来的版本中，默认行为会发生变化，非法访问会被拒绝。为了未雨绸缪地应对这种变化，我们应该用 `--illegal-access` 标志来测试我们的应用程序。下面是 4 种可能的设置：

1. `--illegal-access=permit` 是 Java 9 默认的行为，它会在每一种非法访问第一次出现时打印一条消息。
2. `--illegal-access=warn` 对每次非法访问都打印一条消息。
3. `--illegal-access=debug` 对每次非法访问都打印一条消息和栈轨迹。
4. `--illegal-access=deny` 是未来的默认行为，直接拒绝所有非法访问。

现在是时候用 `--illegal-access=deny` 来测试了，这样我们就可以为这种行为变成默认行为时做好准备。

考虑这样的一个应用程序，它使用了一个不能再继续访问的内部 API，例如 `com.sun.rowset.CachedRowSetImpl`。最好的解决方案就是修改这个实现。（在 Java 7 中，可以从 `RowSetProvider` 中获取一个缓冲的行集。）但是，假设我们不能访问源代码。

在这种情况下，用 `--add-exports` 标志启动该应用程序，指定希望导出的模块和包，以及将包导出到的模块，在我们所举的例子中，包会导出到不具名模块中。

```
java --illegal-access=deny --add-exports java.sql.rowset/com.sun.rowset=ALL_UNNAMED \
-jar MyApp.jar
```

现在，假设我们的应用程序使用反射来访问私有域或方法，那么在不具名模块内的反射是可行的，但是对 Java 平台类的非公有成员的反射式访问就再也不可行了。例如，有些动态生成 Java 类的库会通过反射来调用受保护的 `ClassLoader.defineClass`。如果某个应用程序使用了这样的库，那么需要添加下面的标志

```
--add-opens java.base/java.lang=ALL_UNNAMED
```

当添加这些命令行选项来让遗留应用程序工作时，你可能最终会被这些吓人的命令行吓倒。为了更好地管理多个选项，可以将它们放到一个或多个用 @ 前缀指定的文件中。例如

```
java @options1 @options2 -jar MyProg.java
```

其中文件 options1 和 options2 包含 java 命令的选项。

对于选项文件，有多条相关的语法规则：

- 用空格、制表符和换行符将各个选项分离
- 用双引号将包括空格在内的参数括起来，例如 "Program Files"
- 在一行的末尾用一个 \ 来合并下一行
- 反斜杠必须转义，例如 C:\\Users\\Fred
- 注释以 # 开头

9.11 传递的需求和静态的需求

在 9.4 节中，你已经看到了 `requires` 语句的基本形式。在本节中，你将看到偶尔会用到的它的两种变体。

在某些情况下，对于给定模块的用户而言，声明所有需要的模块会显得很冗长。例如，考虑一下包含像按钮这样的 JavaFX 用户界面元素的 `javafx.controls` 模块。`javafx.controls` 需要 `javafx.base` 模块，因此每个使用 `javafx.controls` 的程序也都需要 `javafx.base` 模块。（如果没有获取 `javafx.base` 模块中的包，那么我们就无法用像按钮这样的用户界面控件做太多的事情。）因为这个原因，`javafx.controls` 模块声明了需要使用 `transitive` 修饰符：

```
module javafx.controls
{
    requires transitive javafx.base;
    ...
}
```

任何声明需要 `javafx.controls` 的模块现在都自动地需要 `javafx.base`。

注释：有些程序员推荐在来自另一个模块的包会在公有 API 中用到时，应该总是使用 `requires transitive`。但是，这并不是 Java 语言的规则。例如，考虑 `java.sql` 模块：

```
module java.sql
{
    requires transitive java.logging;
    ...
}
```

在整个 `java.sql` API 中，唯一用到 `java.logging` 模块中的包的地方，就是 `java.sql.Driver.parentLogger` 方法，它会返回一个 `java.util.logging.Logger` 对象。此时，最可接受的方式是不要将这个模块需求声明成传递性的。然后，那些真正使用这个方法的模块，并且也只有那些模块，需要声明它们需要 `java.logging`。

`requires transitive` 语句的一种很有吸引力的用法是聚集模块，即没有任何包，只有传递性需求的模块。`java.se` 模块就是这样的模块，它被声明成下面的样子：

```
module java.se
{
    requires transitive java.compiler;
    requires transitive java.datatransfer;
```

```

requires transitive java.desktop;
...
requires transitive java.sql;
requires transitive java.sql.rowset;
requires transitive java.xml;
requires transitive java.xml.crypto;
}

```

对细粒度模块依赖不感兴趣的程序员可以直接声明需要 `java.se`，然后获取 Java SE 平台的所有模块。

最终，还有一种不常见的 `requires static` 变体，它声明一个模块必须在编译时出现，而在运行时是可选的。下面是两个用例：

1. 访问在编译时进行处理的注解，而该注解是在不同的模块中声明的。
2. 对于位于不同模块中的类，如果它可用，就使用它，否则就执行其他操作，例如：

```

try
{
    new oracle.jdbc.driver.OracleDriver();
    ...
}
catch (NoClassDefFoundError er)
{
    Do something else
}

```

9.12 限定导出和开放

在本节中，你将会看到 `exports` 和 `opens` 语句的一种变体，将它们的作用域窄化到指定的模块集。例如，`javafx.base` 模块包含下面的语句：

```

exports com.sun.javafx.collections to
    javafx.controls, javafx.graphics, javafx.fxml, javafx.swing;

```

这样的语句被称为限定导出，所列的模块可以访问这个包，但是其他模块不行。

过多地使用限定导出表明模块化结构比较糟糕。尽管如此，在模块化现有代码基时，这种情况还是会发生的。这里，Java 平台的设计者们将 JavaFX 的代码分布到了多个模块中，这是个好主意，因为并非所有的 JavaFX 实现都需要用到 FXML 和 Swing 的交互性。但是，JavaFX 的实现者们可以在他们的代码中不受限制地使用像 `com.sun.javafx.collections.ListListenerHelper` 这样的内部类。在新创建的项目中，人们可以设计更健壮的公有 API。

类似地，可以将 `opens` 语句限制到具体的模块。例如，在 9.7 节中，我们使用了下面这样的限定 `opens` 语句：

```

module v2ch09.openpkg
{
    requires com.horstmann.util;
    opens com.horstmann.places to com.horstmann.util;
}

```

现在，`com.horstmann.places` 包就只对 `com.horstmann.util` 模块开放了。

9.13 服务加载

`ServiceLoader` 类（卷 I 第 6 章）提供了一种轻量级机制，用于将服务接口与实现匹配起来。Java 平台模块系统使得这种机制更易于使用。

下面是对服务加载的一个快速回顾。服务拥有一个接口和一个或多个可能的实现。下面是一个简单的接口示例：

```
public interface GreeterService
{
    String greet(String subject);
    Locale getLocale();
}
```

有一个或多个模块提供了实现，例如

```
public class FrenchGreeter implements GreeterService
{
    public String greet(String subject) { return "Bonjour " + subject; }
    public Locale getLocale() { return Locale.FRENCH; }
}
```

服务消费者必须基于其认为适合的标准在提供的所有实现中选择一个。

```
ServiceLoader<GreeterService> greeterLoader = ServiceLoader.load(GreeterService.class);
GreeterService chosenGreeter;
for (GreeterService greeter : greeterLoader)
{
    if (...)
    {
        chosenGreeter = greeter;
    }
}
```

在过去，实现是通过将文本文件放置到包含实现类的 `META-INF/services` 目录中而提供给服务消费者的。模块系统提供了一种更好的方式，与提供文本文件不同，可以添加语句到模块描述符中。

提供服务实现的模块可以添加一条 `provides` 语句，它列出了服务接口（可能定义在任何模块中），以及实现类（必须是该模块的一部分）。下面是来自 `jdk.security.auth` 模块的一个例子：

```
module jdk.security.auth
{
    ...
    provides javax.security.auth.spi.LoginModule with
        com.sun.security.auth.module.Krb5LoginModule,
        com.sun.security.auth.module.UnixLoginModule,
        com.sun.security.auth.module.JndiLoginModule,
        com.sun.security.auth.module.KeyStoreLoginModule,
        com.sun.security.auth.module.LdapLoginModule,
        com.sun.security.auth.module.NTLoginModule;
}
```

这与 `META-INF/services` 文件等价。

使用它的消费模块包含一条 uses 语句：

```
module java.base
{
    ...
    uses javax.security.auth.spi.LoginModule;
}
```

当消费模块中的代码调用 `ServiceLoader.load(ServiceInterface.class)` 时，匹配的提供者类将被加载，尽管它们可能不在可访问的包中。

在我们的代码示例中，我们为 `com.horstmann.greetsvc.internal` 包中的德语和法语问候者提供了相关的实现。该服务模块导出了 `com.horstmann.greetsvc` 包，但是没有导出包含实现的包。`provides` 语句声明了在未导出包中的服务及其实现类：

```
module com.horstmann.greetsvc
{
    exports com.horstmann.greetsvc;

    provides com.horstmann.greetsvc.GreeterService with
        com.horstmann.greetsvc.internal.FrenchGreeter,
        com.horstmann.greetsvc.internal.GermanGreeterFactory;
}
```

`v2ch09.usesservice` 模块会消费该服务。通过使用 `ServiceLoader` 工具，我们会迭代提供的所有服务，并挑选出匹配所期望语言的服务：

```
package com.horstmann.hello;

import java.util.*;
import com.horstmann.greetsvc.*;

public class HelloWorld
{
    public static void main(String[] args)
    {
        ServiceLoader<GreeterService> greeterLoader
            = ServiceLoader.load(GreeterService.class);
        String desiredLanguage = args.length > 0 ? args[0] : "de";
        GreeterService chosenGreeter = null;
        for (GreeterService greeter : greeterLoader)
        {
            if (greeter.getLocale().getLanguage().equals(desiredLanguage))
                chosenGreeter = greeter;
        }
        if (chosenGreeter == null)
            System.out.println("No suitable greeter.");
        else
            System.out.println(chosenGreeter.greet("Modular World"));
    }
}
```

该模块声明需要服务模块，并声明 `GreeterService` 正在被使用。

```
module v2ch09.usesservice
{
```

```

    requires com.horstmann.greetsvc;
    uses com.horstmann.greetsvc.GreeterService;
}

```

`provides` 和 `uses` 声明的效果，是使得消费该服务的模块允许访问私有实现类。

为了构建并运行该程序，首先要编译服务：

```

javac com.horstmann.greetsvc/module-info.java \
com.horstmann.greetsvc/com/horstmann/greetsvc/GreeterService.java \
com.horstmann.greetsvc/com/horstmann/greetsvc/internal/*.java

```

然后，编译并运行消费模块：

```

javac -p com.horstmann.greetsvc \
v2ch09.usesservice/com/horstmann/hello/HelloWorld.java \
v2ch09.usesservice/module-info.java
java -p com.horstmann.greetsvc:v2ch09.usesservice \
-m v2ch09.usesservice/com.horstmann.hello.HelloWorld

```

9.14 操作模块的工具

`jdeps` 工具可以分析给定的 JAR 文件集之间的依赖关系。例如，假设我们想要模块化 Junit 4。运行

```
jdeps -s junit-4.12.jar hamcrest-core-1.3.jar
```

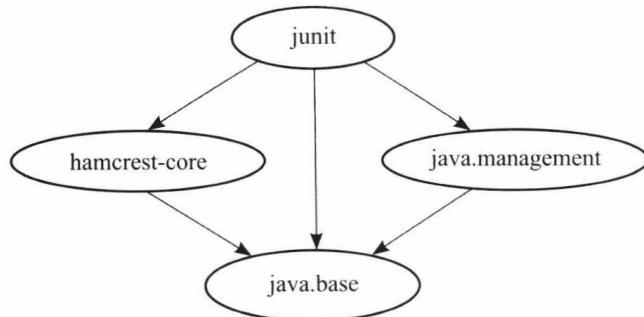
`-s` 标志会产生总结性的输出：

```

hamcrest-core-1.3.jar -> java.base
junit-4.12.jar -> hamcrest-core-1.3.jar
junit-4.12.jar -> java.base
junit-4.12.jar -> java.management

```

它告知了我们下面的模块图：



如果删除 `-s` 标志，那么我们得到的是模块的总结，后面跟着一个映射表，将包映射到所需要的包和模块上。如果添加 `-v` 标志，那么列出的清单会将类映射到所需要的包和模块上。

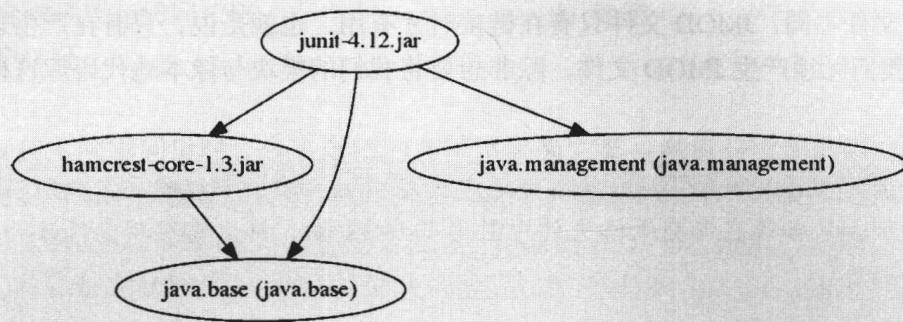
`--generate-module-info` 选项会对每个分析过的模块产生 `module-info` 文件：

```
jdeps --generate-module-info /tmp/junit junit-4.12.jar hamcrest-core-1.3.jar
```

注释：还有一个选项，可以用“dot”语言生成用于描述图的图形化输出。假设我们已经安装了 dot 工具，那么运行下面的命令：

```
jdeps -s -dotoutput /tmp/junit junit-4.12.jar hamcrest-core-1.3.jar
dot -Tpng /tmp/junit/summary.dot > /tmp/junit/summary.png
```

就会得到下面的 summary.png 图：



使用 jlink 工具可以产生执行时无须单独的 Java 运行时环境的应用程序。所产生的镜像比整个 JDK 要小很多。我们可以指定想要包含的模块和输出目录：

```
jlink --module-path com.horstmann.greet.jar:v2ch09.exportedpkg.jar:$JAVA_HOME/jmods \
--add-modules v2ch09.exportedpkg --output /tmp/hello
```

输出目录有一个包含 java 可执行文件的子目录 bin。如果运行

```
bin/java -m v2ch09.exportedpkg
```

那么该模块的主类的 main 方法就会被调用。

jlink 的关键是它将运行应用程序所需的最小的模块集打包在一起。我们可以列出其中包含的所有模块：

```
bin/java --list-modules
```

在这个示例中，输出是

```
v2ch09.exportedpkg
com.horstmann.greet
java.base@9
```

所有模块都包含在运行时镜像文件 lib/modules 中。在我的计算机上，这个文件有 23MB，而所有 JDK 模块的运行时镜像会占据 181MB。整个应用占据 45MB，比 486MB 的 JDK 的 10% 还少。

这可以成为用于打包应用程序的实用工具的基础。我们仍旧需要产生针对多平台的文件集和针对应用程序的脚本。

注释：我们可以用 jimage 命令来审视运行时镜像。但是，其格式对 JVM 来说是内部的，并且运行时镜像并不是为其他工具而生成并供其他工具所使用的。

最后，`jmod` 工具可以构建并审视包含在 JDK 中的模块文件。当查看 JDK 内的 `jmods` 目录时，会发现针对每个模块都有一个扩展名为 `jmod` 的文件。要注意的是，现在再也没有 `rt.jar` 文件了。

与 JAR 文件一样，这些文件也包括类文件。此外，它们还可以包括本地代码库、命令、头文件、配置文件和合法的通知。`JMOD` 文件使用 ZIP 文件格式，可以用任意 ZIP 工具查看它们的内容。

与 JAR 文件不同，`JMOD` 文件只有在链接时才有用，也就是说，只有在产生运行时镜像时才有用。我们无须产生 `JMOD` 文件，除非想要将我们的模块与像本地代码库这样的二进制文件绑定。

注释：因为 `rt.jar` 和 `tools.jar` 文件不再包含在 Java 9 中，所以需要更新所有对它们的引用。例如，如果在安全策略文件中引用了 `tools.jar`，那么需要将它修改为对下面的模块的引用：

```
grant codeBase "jrt:/jdk.compiler"
{
    permission java.security.AllPermission;
};
```

`jrt`: 语法表示 Java 运行时文件。

现在到了该结束 Java 平台模块系统这一章的时候了。下一章将讨论另一个重要的主题：安全。安全已经成为 Java 平台的核心特性之一。我们生活和计算的世界正在变得越来越危险，彻底理解 Java 的安全对许多开发者来说重要性与日俱增。

第 10 章 安 全

▲ 类加载器

▲ 安全管理器与访问权限

▲ 用户认证

▲ 数字签名

▲ 加密

当 Java 技术刚刚问世时，令人激动的并不是因为它是一种设计完美的编程语言，而是因为它能够安全地运行通过因特网传播的各种 applet。很显然，只有当用户确信 applet 的代码不会破坏他的计算机时，用户才会接受在网上传播的可执行的 applet。因此，安全是 Java 技术的设计人员和使用者所关心的一个重大问题。这就意味着，Java 与其他的语言和系统有所不同，在那些语言和系统中安全是在事后才想到要去实现的，或者是对破坏的一种应对措施，而对 Java 来说，安全机制是一个不可分割的组成部分。

Java 技术提供了以下三种确保安全的机制：

- 语言设计特性（对数组的边界进行检查，无不受检查的类型转换，无指针算法等）。
- 访问控制机制，用于控制代码能够执行的操作（比如文件访问，网络访问等）。
- 代码签名，利用该特性，代码的作者就能够用标准的加密算法来认证 Java 代码。这样，该代码的使用者就能够准确地知道谁创建了该代码，以及代码签名后是否被修改过。

首先，我们来讨论类加载器，它可以在将类加载到虚拟机中的时候检查类的完整性。我们将展示这种机制是如何探测类文件中的损坏的。

为了获得最大的安全性，无论是加载类的默认机制，还是自定义的类加载器，都需要与负责控制代码运行的安全管理器类协同工作。后面我们还要详细介绍如何配置 Java 平台的安全性。

最后，我们要介绍 `java.security` 包提供的加密算法，用来进行代码的签名和用户认证。

与我们的一贯宗旨一样，我们将重点介绍应用程序编程人员最感兴趣的话题。如果要深入研究，推荐阅读 Li Gong、Gary Ellison 和 Mary Dageforde 撰写的 *Inside Java 2 Platform Security: Architecture, API Design, and Implementation* 一书，该书由 Prentice Hall 出版社于 2003 年出版。

10.1 类加载器

Java 编译器会为虚拟机转换源指令。虚拟机代码存储在以 `.class` 为扩展名的类文件中，

每个类文件都包含某个类或者接口的定义和实现代码。在以下各节中，你将会看到虚拟机是如何加载这些类文件的。

10.1.1 类加载过程

请注意，虚拟机只加载程序执行时所需要的类文件。例如，假设程序从 `MyProgram.class` 开始运行，下面是虚拟机执行的步骤：

1. 虚拟机有一个用于加载类文件的机制，例如，从磁盘上读取文件或者请求 Web 上的文件，它使用该机制来加载 `MyProgram` 类文件中的内容。
2. 如果 `MyProgram` 类拥有类型为另一个类的域，或者是拥有超类，那么这些类文件也会被加载。（加载某个类所依赖的所有类的过程称为类的解析。）
3. 接着，虚拟机执行 `MyProgram` 中的 `main` 方法（它是静态的，无须创建类的实例）。
4. 如果 `main` 方法或者 `main` 调用的方法要用到更多的类，那么接下来就会加载这些类。

然而，类加载机制并非只使用单个的类加载器。每个 Java 程序至少拥有三个类加载器：

- 引导类加载器
- 平台类加载器
- 系统类加载器（有时也称为应用类加载器）

引导类加载器负责加载包含在下列模块以及大量的 JDK 内部模块中的平台类：

```
java.base
java.datatransfer
java.desktop
java.instrument
java.logging
java.management
java.management.rmi
java.naming
java.prefs
java.rmi
java.security.sasl
java.xml
```

引导类加载器没有对应的 `ClassLoader` 对象，例如，方法

```
StringBuilder.class.getClassLoader()
```

将返回 `null`。

在 Java 9 之前，Java 平台类位于 `rt.jar` 中。如今，Java 平台是模块化的，每个平台模块都包含一个 `JMOD` 文件（参见第 9 章）。平台类加载器会加载引导类加载器没有加载的 Java 平台中的所有类。

系统类加载器会从模块路径和类路径中加载应用类。

注释：在 Java 9 之前，“扩展类加载器”会加载 `jre/lib/ext` 目录中的“标准扩展”，而“授权标准覆盖”机制提供了一种方式，可以用更新的版本覆盖某些平台类（包括 CORBA 和 XML 的实现）。这两种机制都被移除了。

10.1.2 类加载器的层次结构

类加载器有一种父 / 子关系。除了引导类加载器外，每个类加载器都有一个父类加载器。根据规定，类加载器会为它的父类加载器提供一个机会，以便加载任何给定的类，并且只有在其父类加载器加载失败时，它才会加载该给定类。例如，当要求系统类加载器加载一个系统类（比如，`java.lang.StringBuilder`）时，它首先要求平台类加载器进行加载，该加载器则首先要求引导类加载器进行加载。引导类加载器会找到并加载这个类，而无须其他类加载器做更多的搜索。

某些程序具有插件架构，其中代码的某些部分是作为可选的插件打包的。如果插件被打包为 JAR 文件，那就可以直接用 `URLClassLoader` 类的实例去加载插件类。

```
var url = new URL("file:///path/to/plugin.jar");
var pluginLoader = new URLClassLoader(new URL[] { url });
Class<?> cl = pluginLoader.loadClass("mypackage.MyClass");
```

由于在 `URLClassLoader` 构造器中没有指定父类加载器，因此 `pluginLoader` 的父亲就是系统类加载器。图 10-1 展示了这种层次结构。

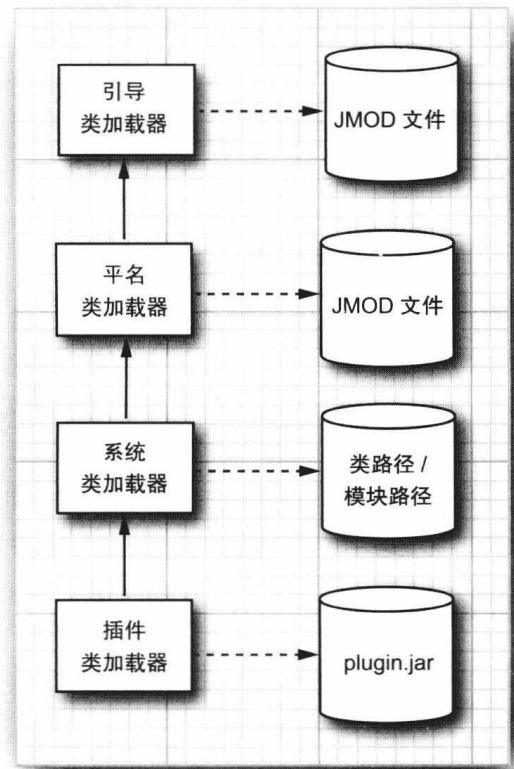


图 10-1 类加载器的层次结构

◆ 警告：在 Java 9 之前，系统类加载器是 `URLClassLoader` 类的实例。有些程序员会使用强制转型来访问其 `getURLs` 方法，或者通过反射机制调用受保护的 `addURLs` 方法将 JAR 文件添加到类路径中。现在无法这样操作了。

大多数时候，你不必操心类加载的层次结构。通常，类是由于其他的类需要它而被加载的，而这个过程对你是透明的。

偶尔，你也会需要干涉和指定类加载器。考虑下面的例子：

- 你的应用的代码包含一个助手方法，它要调用 `Class.forName(classNameString)`。
- 这个方法是从一个插件类中被调用的。
- `classNameString` 指定的正是一个包含在这个插件的 JAR 中的类。

插件的作者期望这个类会被加载。但是，助手方法的类是由系统类加载器加载的，这正是 `Class.forName` 所使用的类加载器。而对于它来说，插件 JAR 中的类是不可见的，这种现象称为类加载器倒置。

要解决这个问题，助手方法需要使用恰当的类加载器，它可以要求类加载器作为其一个参数传递给它。或者，它可以要求将恰当的类加载器设置成为当前线程的上下文类加载器，这种策略在许多框架中都得到了应用（例如 JAXP 和 JNDI）。

每个线程都有一个对类加载器的引用，称为上下文类加载器。主线程的上下文类加载器是系统类加载器。当新线程创建时，它的上下文类加载器会被设置成为创建该线程的上下文类加载器。因此，如果不做任何特殊的操作，那么所有线程就都会将它们的上下文类加载器设置为系统类加载器。

但是，我们也可以通过下面的调用将其设置成为任何类加载器。

```
Thread t = Thread.currentThread();
t.setContextClassLoader(loader);
```

然后助手方法可以获取这个上下文类加载器：

```
Thread t = Thread.currentThread();
ClassLoader loader = t.getContextClassLoader();
Class<?> cl = loader.loadClass(className);
```

 **提示：**如果你编写了一个按名字来加载类的方法，那么让调用者在传递显式的类加载器和使用上下文类加载器之间进行选择是一种好的做法。不要直接使用该方法所属的类的类加载器。

10.1.3 将类加载器用作命名空间

每个 Java 程序员都知道，包的命名是为了消除名字冲突。在标准类库中，有两个名为 `Date` 的类，它们的实际名字分别为 `java.util.Date` 和 `java.sql.Date`。使用简单的名字只是为了方便程序员，它们要求程序包含恰当的 `import` 语句。在一个正在执行的程序中，所有的类名都包含它们的包名。

然而，令人惊奇的是，在同一个虚拟机中，可以有两个类，它们的类名和包名都是相同的。类是由它的全名和类加载器来确定的。这项技术在加载来自多处的代码时很有用。例如，应用服务器会为每一个应用使用单独的类加载器，这使得虚拟机可以区分来自不同应用的类，而无论它们是怎样命名的。图 10-2 展示了一个示例。假设一个应用服务器加载了两个

不同的应用，它们都有一个名为 Util 的类。因为每个类都是由单独的类加载器加载的，所以这些类可以彻底地区分开而不会产生任何冲突。

10.1.4 编写你自己的类加载器

我们可以编写自己的用于特殊目的的类加载器，这使得我们可以在向虚拟机传递字节码之前执行定制的检查。例如，我们可以编写一个类加载器，它可以拒绝加载没有标记为“paid for”的类。

如果要编写自己的类加载器，只需要继承 ClassLoader 类，然后覆盖下面这个方法：

```
findClass(String className)
```

ClassLoader 超类的 loadClass 方法用于将类的加载操作委托给其父类加载器去进行，只有当该类尚未加载并且父类加载器也无法加载该类时，才调用 findClass 方法。

如果要实现该方法，必须做到以下几点：

1. 为来自本地文件系统或者其他来源的类加载其字节码。

2. 调用 ClassLoader 超类的 defineClass 方法，向虚拟机提供字节码。

在程序清单 10-1 中，我们实现了一个类加载器，用于加载加密过的类文件。该程序要求用户输入第一个要加载的类的名字（即包含 main 方法的类）和密钥。然后，使用一个专门的类加载器来加载指定的类并调用 main 方法。该类加载器对指定的类和所有被其引用的非系统类进行解密。最后，该程序会调用已加载类的 main 方法（参见图 10-3）。

为了简单起见，我们忽略了密码学领域 2000 年来所取得的技术进展，而是采用了传统的 Caesar 密码对类文件进行加密。

注释：David Kahn 的佳作 *The Codebreakers*（纽约 Macmillan 出版社 1967 年出版）第 84 页中称 Suetonius 是 Caesar 密码的发明人。Caesar 将罗马字母表的 24 个字母移动了 3 个字母的位置，在那个时代这可以迷惑对手。

第一次撰写本章时，美国政府限制高强度加密方法的出口。因此，我们在实例中使用的是 Caesar 的加密方法，因为该方法的出口显然是合法的。

我们的 Caesar 密码版本使用的密钥是 1 到 255 之间的一个数字，解密时，只需将密钥与每个字节相加，然后对 256 取余。程序清单 10-2 的 Caesar.java 程序就实现了这种加密行为。

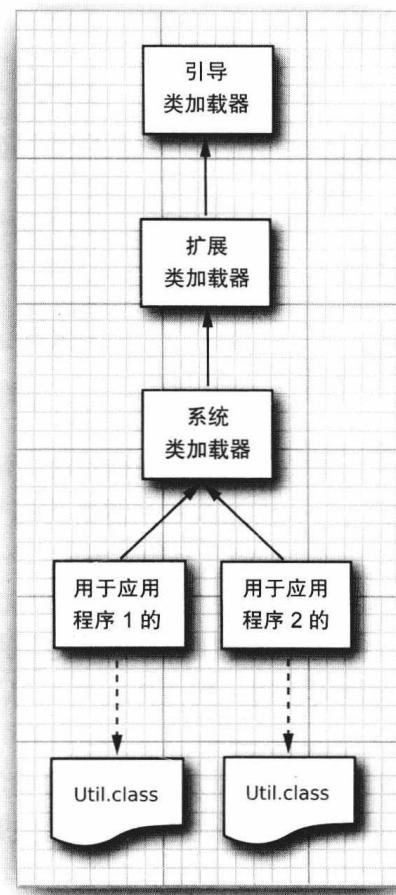


图 10-2 两个类加载器分别加载具有相同名字的两个类

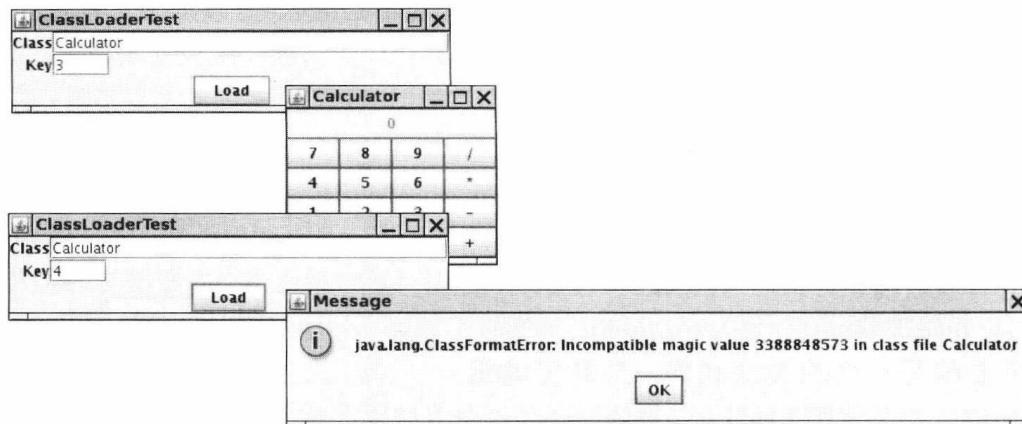


图 10-3 ClassLoaderTest 程序

为了不与常规的类加载器相混淆，我们对加密的类文件使用了不同的扩展名 .caesar。

解密时，类加载器只需要将每个字节减去该密钥即可。在本书的程序代码中，可以找到 4 个类文件，它们都是用“3”这个传统的密钥值进行加密的。为了运行加密程序，需要使用在我们的 ClassLoaderTest 程序中定义的定制类加载器。

对类文件进行加密有很大的用途（当然，使用的密码的强度应该高于 Caesar 密码的），如果没有加密密钥，类文件就毫无用处。它们既不能由标准虚拟机来执行，也不能轻易地被反汇编。

这就是说，可以使用定制的类加载器来认证类用户的身份，或者确保程序在运行之前已经支付了软件费用。当然，加密只是定制类加载器的应用之一。可以使用其他类型的加载器来解决别的问题，例如，将类文件存储到数据库中。

程序清单 10-1 classLoader/ClassLoaderTest.java

```

1 package classLoader;
2
3 import java.io.*;
4 import java.lang.reflect.*;
5 import java.nio.file.*;
6 import java.awt.*;
7 import java.awt.event.*;
8 import javax.swing.*;
9
10 /**
11  * This program demonstrates a custom class loader that decrypts class files.
12  * @version 1.25 2018-05-01
13  * @author Cay Horstmann
14  */
15 public class ClassLoaderTest
16 {
17     public static void main(String[] args)
18     {
19         EventQueue.invokeLater(() ->

```

```
20     {
21         var frame = new ClassLoaderFrame();
22         frame.setTitle("ClassLoaderTest");
23         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24         frame.setVisible(true);
25     });
26 }
27 }
28 /**
29 * This frame contains two text fields for the name of the class to load and the decryption
30 * key.
31 */
32
33 class ClassLoaderFrame extends JFrame
34 {
35     private JTextField keyField = new JTextField("3", 4);
36     private JTextField nameField = new JTextField("Calculator", 30);
37     private static final int DEFAULT_WIDTH = 300;
38     private static final int DEFAULT_HEIGHT = 200;
39
40     public ClassLoaderFrame()
41     {
42         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
43         setLayout(new GridBagLayout());
44         add(new JLabel("Class"), new GBC(0, 0).setAnchor(GBC.EAST));
45         add(nameField, new GBC(1, 0).setWeight(100, 0).setAnchor(GBC.WEST));
46         add(new JLabel("Key"), new GBC(0, 1).setAnchor(GBC.EAST));
47         add(keyField, new GBC(1, 1).setWeight(100, 0).setAnchor(GBC.WEST));
48         var loadButton = new JButton("Load");
49         add(loadButton, new GBC(0, 2, 2, 1));
50         loadButton.addActionListener(event -> runClass(nameField.getText(), keyField.getText()));
51         pack();
52     }
53
54 /**
55 * Runs the main method of a given class.
56 * @param name the class name
57 * @param key the decryption key for the class files
58 */
59 public void runClass(String name, String key)
60 {
61     try
62     {
63         var loader = new CryptoClassLoader(Integer.parseInt(key));
64         Class<?> c = loader.loadClass(name);
65         Method m = c.getMethod("main", String[].class);
66         m.invoke(null, (Object) new String[] {});
67     }
68     catch (Throwable t)
69     {
70         JOptionPane.showMessageDialog(this, t);
71     }
72 }
73 }
```

```

74
75 /**
76 * This class loader loads encrypted class files.
77 */
78 class CryptoClassLoader extends ClassLoader
79 {
80     private int key;
81
82 /**
83 * Constructs a crypto class loader.
84 * @param k the decryption key
85 */
86 public CryptoClassLoader(int k)
87 {
88     key = k;
89 }
90
91 protected Class<?> findClass(String name) throws ClassNotFoundException
92 {
93     try
94     {
95         byte[] classBytes = null;
96         classBytes = loadClassBytes(name);
97         Class<?> cl = defineClass(name, classBytes, 0, classBytes.length);
98         if (cl == null) throw new ClassNotFoundException(name);
99         return cl;
100    }
101   catch (IOException e)
102   {
103       throw new ClassNotFoundException(name);
104   }
105 }
106
107 /**
108 * Loads and decrypt the class file bytes.
109 * @param name the class name
110 * @return an array with the class file bytes
111 */
112 private byte[] loadClassBytes(String name) throws IOException
113 {
114     String cname = name.replace('.', '/') + ".caesar";
115     byte[] bytes = Files.readAllBytes(Paths.get(cname));
116     for (int i = 0; i < bytes.length; i++)
117         bytes[i] = (byte) (bytes[i] - key);
118     return bytes;
119 }
120 }

```

程序清单 10-2 classLoader/Caesar.java

```

1 package classLoader;
2
3 import java.io.*;
4

```

```

5  /**
6   * Encrypts a file using the Caesar cipher.
7   * @version 1.02 2018-05-01
8   * @author Cay Horstmann
9  */
10 public class Caesar
11 {
12     public static void main(String[] args) throws Exception
13     {
14         if (args.length != 3)
15         {
16             System.out.println("USAGE: java classLoader.Caesar in out key");
17             return;
18         }
19
20         try (var in = new FileInputStream(args[0]));
21             var out = new FileOutputStream(args[1]))
22         {
23             int key = Integer.parseInt(args[2]);
24             int ch;
25             while ((ch = in.read()) != -1)
26             {
27                 byte c = (byte) (ch + key);
28                 out.write(c);
29             }
30         }
31     }
32 }

```

API **java.lang.Class 1.0**

- **ClassLoader getClassLoader()**

获取加载该类的类加载器。

API **java.lang.ClassLoader 1.0**

- **ClassLoader getParent() 1.2**

返回父类加载器，如果父类加载器是引导类加载器，则返回 `null`。

- **static ClassLoader getSystemClassLoader() 1.2**

获取系统类加载器，即用于加载第一个应用类的类加载器。

- **protected Class findClass(String name) 1.2**

类加载器应该覆盖该方法，以查找类的字节码，并通过调用 `defineClass` 方法将字节码传给虚拟机。在类的名字中，使用 `.` 作为包名分隔符，并且不使用 `.class` 后缀。

- **Class defineClass(String name, byte[] byteCodeData, int offset, int length)**

将一个新的类添加到虚拟机中，其字节码在给定的数据范围内。

API **java.net.URLClassLoader 1.2**

- **URLClassLoader(URL[] urls)**

- `URLClassLoader(URL[] urls, ClassLoader parent)`

构建一个类加载器，它可以从给定的 URL 处加载类。如果 URL 以 / 结尾，那么它表示的是一个目录，否则，它表示的是一个 JAR 文件。

API `java.lang.Thread` 1.0

- `ClassLoader getContextClassLoader() 1.2`

获取类加载器，该线程的创建者将其指定为执行该线程时最适合使用的类加载器。

- `void setContextClassLoader(ClassLoader loader) 1.2`

为该线程中的代码设置一个类加载器，以获取要加载的类。如果在启动一个线程时没有显式地设置上下文类加载器，则使用父线程的上下文类加载器。

10.1.5 字节码校验

当类加载器将新加载的 Java 平台类的字节码传递给虚拟机时，这些字节码首先要接受校验器（verifier）的校验。校验器负责检查那些指令无法执行的明显有破坏性的操作。除了系统类外，所有的类都要被校验。

下面是校验器执行的一些检查：

- 变量要在使用之前进行初始化。
- 方法调用与对象引用类型之间要匹配。
- 访问私有数据和方法的规则没有被违反。
- 对本地变量的访问都落在运行时堆栈内。
- 运行时堆栈没有溢出。

如果以上这些检查中任何一条没有通过，那么该类就被认为遭到了破坏，并且不予加载。

注释：如果熟悉 Gödel 定理，那么你可能想知道校验器究竟是如何证明某个类文件不存在类型不匹配、变量没有初始化和堆栈溢出等问题的。根据 Gödel 定理，不可能设计出这样的算法：它能够处理程序，确定其是否具有特定的属性（比如不出现堆栈溢出问题）。这是否属于 Oracle 公司的公共关系部门和逻辑法则之间的矛盾呢？不——事实上，校验器并非是一个 Gödel 意义上的决策算法。如果校验器接受了一个程序，那么该程序就确实是安全的。然而，也有许多程序尽管是安全的，但却被校验器拒绝了。（在强制用哑元值来初始化一个变量时，你就会碰到这个问题，因为编译器无法了解这个变量是否可以被正确地初始化。）

这种严格的校验是出于安全上的考虑，有一些偶然性的错误，比如变量没有初始化，如果没有被捕获，就很容易对系统造成严重的破坏。更为重要的是，在因特网这样开放的环境中，你必须保护自己以防恶意的程序员对你实施攻击，因为他们的目的就是要造成恶劣的影响。例如，通过修改运行时堆栈中的值，或者向系统对象的私有数据字段写入数据，某个程序就会突破浏览器的安全防线。

当然，你可能想知道为什么要有一个专门的校验器来检查这些特性。毕竟，编译器绝不会允许你生成一个这样的类文件：该类文件中有未初始化的变量或者可以通过另一个类来访问该类的某个私有数据字段。实际上，用 Java 语言编译器生成的类文件总是可以通过校验的。然而，类文件中使用的字节码格式是文档记录良好的，对于具有汇编程序设计经验并且拥有十六进制编辑器的人来说，要手工地创建一个对 Java 虚拟机来说由合法但是不安全的指令构成的类文件，是一件非常容易的事情。再次提醒你，要记住，校验器总是在防范被故意篡改的类文件，而不只是检查编译器产生的类文件。

下面的例子将展示如何创建一个变动过的类文件。我们从程序清单 10-3 中的程序 VerifierTest.java 开始。这是一个简单的程序，它调用一个方法，并且显示方法的运行结果。该程序既可以在控制台运行，也可以作为一个 applet 程序来运行。其中的 fun 方法本身只是负责计算 $1+2$ 。

```
static int fun()
{
    int m;
    int n;
    m = 1;
    n = 2;
    int r = m + n;
    return r;
}
```

程序清单 10-3 verifier/VerifierTest.java

```
1 package verifier;
2
3 import java.awt.*;
4
5 /**
6  * This application demonstrates the bytecode verifier of the virtual machine. If you use a
7  * hex editor to modify the class file, then the virtual machine should detect the tampering.
8  * @version 1.10 2018-05-05
9  * @author Cay Horstmann
10 */
11 public class VerifierTest
12 {
13     public static void main(String[] args)
14     {
15         System.out.println("1 + 2 == " + fun());
16     }
17
18 /**
19  * A function that computes 1 + 2.
20  * @return 3, if the code has not been corrupted
21  */
22     public static int fun()
23     {
24         int m;
25         int n;
26         m = 1;
```

```

27     n = 2;
28     // use hex editor to change to "m = 2" in class file
29     int r = m + n;
30     return r;
31 }
32 }
```

作为一次实验，请尝试编译下面这个对该程序进行修改后的文件。

```

static int fun()
{
    int m = 1;
    int n;
    m = 1;
    m = 2;
    int r = m + n;
    return r;
}
```

在这种情况下，n 没有被初始化，它可以是任何随机值。当然，编译器能够检测到这个问题并拒绝编译该程序。如果要建立一个不良的类文件，我们必须得多花点工夫。首先，运行 javap 程序，以便知晓编译器是如何翻译 fun 方法的。命令

```
javap -c verifier.VerifierTest
```

用助记 (mnemonic) 格式显示了类文件中的字节码。

```

Method int fun()
  0  iconst_1
  1  istore_0
  2  iconst_2
  3  istore_1
  4  iload_0
  5  iload_1
  6  iadd
  7  istore_2
  8  iload_2
  9  ireturn
```

我们使用一个十六进制编辑器将指令 3 从 istore_1 改为 istore_0，也就是说，局部变量 0 (即 m) 被初始化了两次，而局部变量 1 (即 n) 则根本没有初始化。我们必须知道这些指令的十六进制值，这些值可以从 Java 虚拟机规范中获知：<https://docs.oracle.com/javase/specs/jvms/se11/html/index.html>。

```

0  iconst_1 04
1  istore_0 3B
2  iconst_2 05
3  istore_1 3C
4  iload_0  1A
5  iload_1  1B
6  iadd    60
7  istore_2 3D
8  iload_2  1C
9  ireturn AC
```

可以使用任何十六进制编辑器来执行这种修改。在图 10-4 中，你可以看到类文件 VerifierTest.class 被加载到了 Gnome 编辑器中，fun 方法的字节码已经被选定。

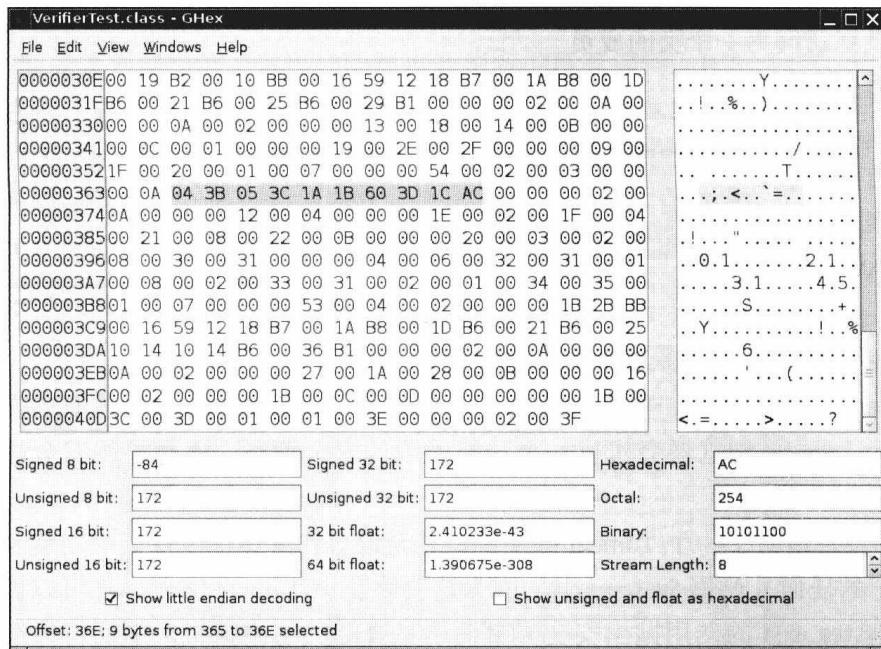


图 10-4 使用十六进制编辑器修改字节码

将 3C 改为 3B 并保存类文件。然后尝试运行 VerifierTest 程序，将会看到下面的出错信息：

```
Exception in thread "main" java.lang.VerifyError: (class: VerifierTest, method:fun signature: ()I) Accessing value from uninitialized register 1
```

这很好——虚拟机发现了我们所做的修改。

现在用 -noverify 选项（或者 -Xverify:none）来运行程序：

```
java -noverify verifier.VerifierTest
```

从表面上看，fun 方法似乎返回了一个随机值。但实际上，该值是 2 与存储在尚未初始化的变量 n 中的值相加得到的结果。下面是典型的输出结果：

```
1 + 2 == 15102330
```

10.2 安全管理器与访问权限

一旦某个类被加载到虚拟机中，并由检验器检查过之后，Java 平台的第二种安全机制就会启动，这个机制就是安全管理器。下面几小节将讨论这种机制。

10.2.1 权限检查

安全管理器是一个负责控制具体操作是否允许执行的类。安全管理器负责检查的操作包

括以下内容：

- 创建一个新的类加载器
- 退出虚拟机
- 使用反射访问另一个类的成员
- 访问本地文件
- 打开 socket 连接
- 启动打印作业
- 访问系统剪贴板
- 访问 AWT 事件队列
- 打开一个顶层窗口

整个 Java 类库中还有许多其他类似的检查。

在运行 Java 应用程序时，默认的设置是不安装安全管理器的，这样所有的操作都是允许的。另一方面，applet 浏览器会执行一个功能受限的安全策略。更严格的安全性对其他情况也具有意义。

例如，假设你运行了一个 Tomcat 的实例，并允许合作者或学生在其中安装 Servlet。你并不想让他们中的任何人调用 `System.exit`，因为这会终止该 Tomcat 实例。你可以设置一个安全策略，让对 `System.exit` 的调用抛出安全异常而不是真的关闭虚拟机。下面将详细说明这种情况。`Runtime` 类的 `exit` 方法会调用安全管理器的 `checkExit` 方法，下面是 `exit` 方法的全部代码：

```
public void exit(int status)
{
    SecurityManager security = System.getSecurityManager();
    if (security != null)
        security.checkExit(status);
    exitInternal(status);
}
```

这时安全管理器要检查退出请求是来自浏览器还是单个的 applet 程序。如果安全管理器同意了退出请求，那么 `checkExit` 便直接返回并继续处理下面正常的操作。但是，如果安全管理器不同意退出请求，那么 `checkExit` 方法就会抛出一个 `SecurityException` 异常。

只有当没有任何异常发生时，`exit` 方法才能继续执行。然后它调用本地私有的 `exitInternal` 方法，以真正终止虚拟机的运行。没有其他的方法可以终止虚拟机的运行，因为 `exitInternal` 方法是私有的，任何其他类都不能调用它。因此，任何试图退出虚拟机的代码都必须通过 `exit` 方法，从而在不触发安全异常的情况下，通过 `checkExit` 安全检查。

显然，安全策略的完整性依赖于谨慎的编码。标准类库中系统服务的提供者，在试图继续任何敏感的操作之前，都必须与安全管理器进行协商。

Java 平台的安全管理器，不仅允许系统管理员，而且允许程序员对各个安全访问权限实施细致的控制。我们将在下一节介绍这些特性。首先，我们将介绍 Java 2 平台的安全模型的概况，然后介绍如何使用策略文件对各个权限实施控制。最后，我们要介绍如何来定义你自己的权限类型。

10.2.2 Java 平台安全性

JDK 1.0 具有一个非常简单的安全模型，即本地类拥有所有的权限，而远程类只能在沙盒里运行。就像儿童只能在沙盒里玩沙子一样，远程代码只被允许打印屏幕和与用户进行交互。applet 的安全管理器拒绝了远程代码对本地资源的所有访问。JDK 1.1 对此进行了微小的修改，如果远程代码带有可信赖的实体的签名，将被赋予和本地类相同的访问权限。不过，JDK 1.0 和 1.1 这两个版本提供的都是一种“要么都有，要么都没有”的权限赋予方法。程序要么拥有所有的访问权限，要么必须在沙盒里运行。

从 Java 1.2 开始，Java 平台拥有了更灵活的安全机制，它的安全策略建立了代码来源和访问权限集之间的映射关系（参见图 10-5）。

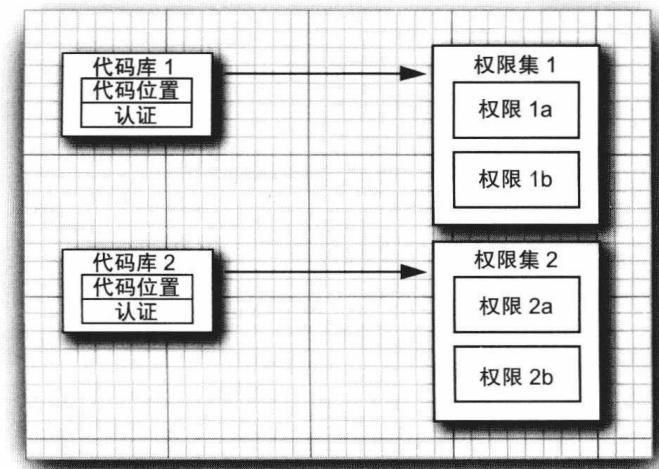


图 10-5 一个安全策略

代码来源（code source）是由一个代码位置和一个证书集指定的。代码位置指定了代码的来源。例如，远程 applet 代码的代码位置是下载 applet 的 HTTP URL，位于 JAR 文件中的代码的代码位置是该文件的 URL。证书的目的是要由某一方来保障代码没有被篡改过。我们将在本章的后面部分讨论证书。

权限（permission）是指由安全管理器负责检查的任何属性。Java 平台支持许多访问权限类，每个类都封装了特定权限的详细信息。例如，下面这个 FilePermission 类的实例表示：允许在 /tmp 目录下读取和写入任何文件。

```
var p = new FilePermission("/tmp/*", "read,write");
```

更为重要的是，Policy 类的默认实现可从访问权限文件中读取权限。在权限文件中，同样的读权限表示为：

```
permission java.io.FilePermission "/tmp/*", "read,write";
```

我们将在下一节介绍权限文件。

图 10-6 显示了 Java 1.2 中提供的权限类的层次结构。JDK 的后续版本添加了更多的权限类。

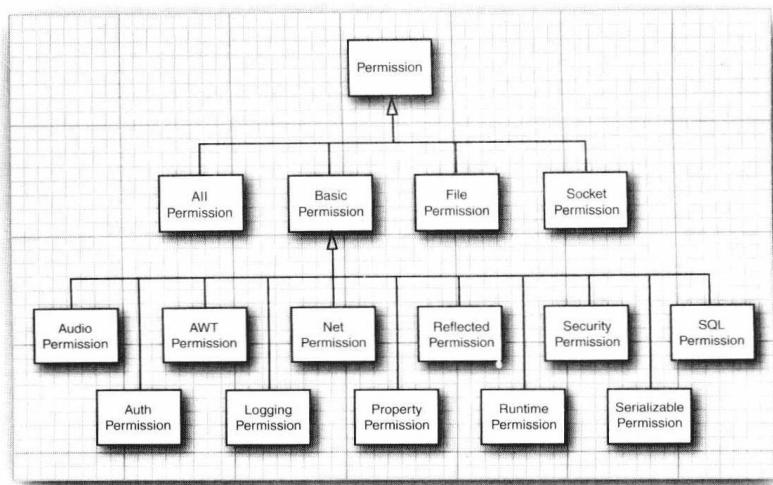


图 10-6 权限类的层次结构

在上一节中，我们看到了 `SecurityManager` 类有许多诸如 `checkExit` 的安全检查方法，这些方法的存在，只是为了程序员的方便和向后的兼容性，它们都已被映射为标准的权限检查，例如，下面是 `checkExit` 方法的源代码：

```

public void checkExit()
{
    checkPermission(new RuntimePermission("exitVM"));
}
  
```

每个类都有一个保护域，它是一个用于封装类的代码来源和权限集合的对象。当 `SecurityManager` 类需要检查某个权限时，它要查看当前位于调用堆栈上的所有方法的类，然后它要获得所有类的保护域，并且询问每个保护域，其权限集合是否允许执行当前正在被检查的操作。如果所有的域都同意，那么检查得以通过。否则，就会抛出一个 `SecurityException` 异常。

为什么在调用堆栈上的所有方法都必须允许某个特定的操作呢？让我们通过一个实例来说明这个问题。假设一个 Servlet 的 `init` 方法想要打开一个文件，它可能会调用下面的语句：

```
var in = new FileReader(name);
```

`FileReader` 构造器调用 `FileInputStream` 构造器，而 `FileInputStream` 构造器调用安全管理器的 `checkRead` 方法，安全管理器最后用 `FilePermission(name, "read")` 对象调用 `checkPermission`。表 10-1 显示了该调用堆栈。

表 10-1 权限检查期间的调用堆栈

类	方法	代码来源	权限
SecurityManager	checkPermission	null	AllPermission
SecurityManager	checkRead	null	AllPermission

(续)

类	方法	代码来源	权限
FileInputStream	Constructor	null	AllPermission
FileReader	Constructor	null	AllPermission
Servlet	init	Servlet 代码来源	TomcatWeb 应用权限

FileInputStream 和 SecurityManager 类都属于系统类，它们的 CodeSource 为 null，它们的权限都是由 AllPermission 类的一个实例组成的，AllPermission 类允许执行所有的操作。显然地，仅仅根据它们的权限是无法确定检查结果的。正如我们所看到的那样，checkPermission 方法必须考虑 applet 类的受限制的权限问题。通过检查整个调用堆栈，安全机制就能够确保一个类决不会要求另一个类代表自己去执行某个敏感的操作。

注释：上面关于如何进行权限检查的简要介绍，向你展示了这方面的基本概念。不过我们在这里省略了对许多技术细节的说明。对于安全性的细节问题，我们建议你阅读 Li Gong 撰写的著作，以便了解更多的内容。有关 Java 平台安全模型的更多重要信息，请查阅 Gary McGraw 和 Ed Felten 撰写的 *Securing Java: Getting Down to Business with Mobile Code* 第 2 版一书，该书由 Wiley 出版社于 1999 年出版。你可以在下面的网站上找到该书的在线版本：<http://www.securingjava.com>。

API **java.lang.SecurityManager 1.0**

- void checkPermission(Permission p) 1.2

检查当前的安全管理器是否授予给定的权限。如果没有授予该权限，本方法抛出一个 SecurityException 异常。

API **java.lang.Class 1.0**

- ProtectionDomain getProtectionDomain() 1.2

获取该类的保护域，如果该类被加载时没有保护域，则返回 null。

API **java.security.ProtectionDomain 1.2**

- ProtectionDomain(CodeSource source, PermissionCollection permissions)

用给定的代码来源和权限构建一个保护域。

- CodeSource getCodeSource()

获取该保护域的代码来源。

- boolean implies(Permission p)

如果该保护域允许给定的权限，则返回 true。

API **java.security.CodeSource 1.2**

- Certificate[] getCertificates()

获取与该代码来源相关联的用于类文件签名的证书链。

- URL getLocation()

获取与该代码来源相关联的类文件代码位置。

10.2.3 安全策略文件

策略管理器要读取相应的策略文件，这些文件包含了将代码来源映射为权限的指令。下面是一个典型的策略文件：

```
grant codeBase "http://www.horstmann.com/classes"
{
    permission java.io.FilePermission "/tmp/*", "read,write";
};
```

该文件给所有下载自 `http://www.horstmann.com/classes` 的代码授予在 `/tmp` 目录下读取和写入文件的权限。

- 可以将策略文件安装在标准位置上。默认情况下，有两个位置可以安装策略文件：
- Java 平台主目录的 `java.policy` 文件。
 - 用户主目录的 `.java.policy` 文件（注意文件名前面的圆点）。

注释：可以在 `jdk/Conf/Security` 目录下 `java.security` 配置文件中修改这些文件的位置，
默认位置设定为：

```
policy.url.1=file:${java.home}/lib/security/java.policy
policy.url.2=file:${user.home}/.java.policy
```

系统管理员可以修改 `java.security` 文件，并可以指定驻留在另外一台服务器上并且用户无法修改的策略 URL。策略文件中允许存放任意数量的策略 URL（这些 URL 带有连续的编号）。所有文件的权限都被组合了在一起。

如果想将策略文件存储到文件系统之外，那么可以去实现 `Policy` 类的一个子类，让其去收集所允许的权限。然后在 `java.security` 配置文件中更改下面这行：

```
policy.provider=sun.security.provider.PolicyFile
```

在测试期间，我们不喜欢经常地修改这些标准文件。因此，我们更愿意为每一个应用程序单独命名策略文件，这样将权限写入一个独立的文件（比如 `MyApp.policy`）中即可。要应用这个策略文件，可以有两个选择。一种是在应用程序的 `main` 方法内部设置系统属性：

```
System.setProperty("java.security.policy", "MyApp.policy");
```

或者，可以像下面这样启动虚拟机：

```
java -Djava.security.policy=MyApp.policy MyApp
```

在这些例子中，`MyApp.policy` 文件被添加到了其他有效的策略中。如果在命令行中添加了第二个等号，比如：

```
java -Djava.security.policy==MyApp.policy MyApp
```

那么应用程序就只使用指定的策略文件，而标准策略文件将被忽略。

! **警告：**在测试期间，一个容易犯的错误是在当前目录中留下了一个 `.java.policy` 文件，该文件授予了许许多多的权限，甚至可能授予了 `AllPermission`。如果发现你的应用程序似乎没有应用策略文件中的规定，就应该检查当前目录下是否留有 `.java.policy` 文件。如果使用的是 UNIX 系统，就更容易犯这样的错误，因为在 UNIX 中，文件名以圆点开头的文件默认是不显示的。

正如前面所说，在默认情况下，Java 应用程序是不安装安全管理器的。因此，在安装安全管理器之前，看不到策略文件的作用。当然，可以将这行代码：

```
System.setSecurityManager(new SecurityManager());
```

添加到 `main` 方法中，或者在启动虚拟机的时候添加命令行选项 `-Djava.security.manager`。

```
java -Djava.security.manager -Djava.security.policy=MyApp.policy MyApp
```

在本节的剩余部分，我们将要详细介绍如何描述策略文件的权限。我们将介绍整个策略文件的格式，不过不包括代码证书部分，代码证书将在本章的后面部分介绍。

一个策略文件包含一系列 `grant` 项。每一项都具有以下的形式：

```
grant codesource
{
    permission1;
    permission2;
    . . .
};
```

代码来源包含一个代码基（如果某一项适用于所有来源的代码，则代码基可以省略）和值得信赖的用户特征（principal）与证书签名者的名字（如果不要求对该项签名，则可以省略）。

代码基可以设定为：

```
codeBase "url"
```

如果 URL 以 “/” 结束，那么它是一个目录。否则，它将被视为一个 JAR 文件的名字。例如：

```
grant codeBase "www.horstmann.com/classes/" { . . . };
grant codeBase "www.horstmann.com/classes/MyApp.jar" { . . . };
```

代码基是一个 URL 并且总是以斜杠作为文件分隔符，即使是 Windows 中的文件 URL，也是如此。例如：

```
grant codeBase "file:C:/myapps/classes/" { . . . };
```

! **注释：**大家都知道 http 格式的 URL 都以双斜杠 (`http://`) 开头的，但是它很容易与 file 格式的 URL 搞混淆，策略文件阅读器接受两种格式的 file URL，即 `file://localFile` 和 `file:localFile`。此外，Windows 驱动器名前面的斜杠是可有可无的。也就是说，下面的各种表示都是可以接受的：

```
file:C:/dir/filename.ext
file:/C:/dir/filename.ext
file://C:/dir/filename.ext
file:///C:/dir/filename.ext
```

实际上，我们的测试结果是 file:///C:/dir/filename.ext 也是允许的，对此我们无法解释。

注释：请考虑编译 Java 代码的应用程序，它需要大量的权限。在 JDK 9 之前，你可以被授权获得对 tools.jar 中的代码的所有权限。这个 JAR 文件现在已经不存在了。因此，需要像下面这样授予对适合的模块进行访问的权限：

```
grant codeBase "jrt:/jdk.compiler"
{
    permission java.security.AllPermission;
};
```

权限采用下面的结构：

```
permission className targetName, actionList;
```

类名是权限类的全称类名（比如 `java.io.FilePermission`）。目标名是个与权限相关的值，例如，文件权限中的目录名或者文件名，或者是 socket 权限中的主机和端口。操作列表同样是与权限相关的，它是一个操作方式的列表，比如 `read` 或者 `connect` 等操作，用逗号分隔。有些权限类并不需要目标名和操作列表。表 10-2 列出了标准的权限和它们执行的操作。

表 10-2 权限及其相关的目标和操作

权限	目标	操作
<code>java.io.FilePermission</code>	文件目标（见正文）	<code>read, write, execute, delete</code>
<code>java.net.SocketPermission</code>	Socket 目标（见正文）	<code>accept, connect, listen, resolve</code>
<code>java.util.PropertyPermission</code>	属性目标（见正文）	<code>read, write</code>
<code>java.lang.RuntimePermission</code>	<code>createClassLoader</code> <code>getClassLoader</code> <code>setContextClassLoader</code> <code>enableContextClassLoaderOverride</code> <code>createSecurityManager</code> <code>setSecurityManager</code> <code>exitVM</code> <code>getenv.variableName</code> <code>shutdownHooks</code> <code>setFactory</code> <code>setIO</code> <code>modifyThread</code> <code>stopThread</code> <code>modifyThreadGroup</code> <code>getProtectionDomain</code> <code>readFileDescriptor</code> <code>writeFileDescriptor</code>	无

(续)

权限	目标	操作
java.lang.RuntimePermission	loadLibrary.libraryName accessClassInPackage.packageName defineClassInPackage.packageName accessDeclaredMembers.className queuePrintJob getStackTrace setDefaultUncaughtExceptionHandler preferences usePolicy	无
java.awt.AWTPermission	showWindowWithoutWarningBanner accessClipboard accessEventQueue createRobot fullScreenExclusive listenToAllAWTEvents readDisplayPixels replaceKeyboardFocusManager watchMousePointer setWindowAlwaysOnTop setAppletStub	无
java.net.NetPermission	setDefaultAuthenticator specifyStreamHandler requestPasswordAuthentication setProxySelector getProxySelector setCookieHandler getCookieHandler setResponseCache getResponseCache	无
java.lang.reflect.ReflectPermission	suppressAccessChecks	无
java.io.SerializablePermission	enableSubclassImplementation enableSubstitution	无
java.security.SecurityPermission	createAccessControlContext getDomainCombiner getPolicy setPolicy getProperty.keyName setProperty.keyName insertProvider.providerName removeProvider.providerName setSystemScope setIdentityPublicKey setIdentityInfo	无

(续)

权限	目标	操作
java.security.SecurityPermission	addIdentityCertificate removeIdentityCertificate printIdentity clearProviderProperties.providerName putProviderProperty.providerName removeProviderProperty.providerName getSignerPrivateKey setSignerKeyPair	无
java.security.AllPermission	无	无
javax.audio.AudioPermission	播放录音	无
javax.security.auth.AuthPermission	doAs doAsPrivileged getSubject getSubjectFromDomainCombiner setReadOnly modifyPrincipals modifyPublicCredentials modifyPrivateCredentials refreshCredential destroyCredential createLoginContext.contextName getLoginConfiguration setLoginConfiguration refreshLoginConfiguration	无
java.util.logging.LoggingPermission	control	无
java.sql.SQLPermission	setLog	无

正如表 10-2 中所示，大部分权限只允许执行某种特定的行为。可以将这些行为视为带有 一个隐含操作“permit”的目标。这些权限类都继承自 BasicPermission 类（参见本章图 10-6）。然而，文件、socket 和属性权限的目标都比较复杂，我们必须对它们进行详细介绍。

文件权限的目标可以有下面几种形式：

<i>file</i>	文件
<i>directory/</i>	目录
<i>directory/*</i>	目录中的所有文件
*	当前目录中的所有文件
<i>directory/-</i>	目录和其子目录中的所有文件
-	当然目录和其子目录中所有文件
<<ALL FILES>>	文件系统中的所有文件

例如，下面的权限项赋予对 /myapp 目录和它的子目录中的所有文件的访问权限。

```
permission java.io.FilePermission "/myapp/-", "read,write,delete";
```

必须使用 \\ 转义字符序列来表示 Window 文件名中的反斜杠。

```
permission java.io.FilePermission "c:\\myapp\\-/-", "read,write,delete";
```

Socket 权限的目标由主机和端口范围组成。对主机的描述具有下面几种形式：

hostname 或 *IPaddress* 单个主机

localhost 或空字符串 本地主机

**.domainSuffix* 以给定后缀结尾的域中所有的主机

*** 所有主机

端口范围是可选的，具有下面几种形式：

:n 单个端口

:n- 编号大于等于 *n* 的所有端口

:-n 编号小于等于 *n* 的所有端口

:n1-n2 位于给定范围内的所有端口

下面是一个权限的实例：

```
permission java.net.SocketPermission "*.*.horstmann.com:8000-8999", "connect";
```

最后，属性权限的目标可以采用下面两种形式之一：

property 一个具体的属性

*propertyPrefix.** 带有给定前缀的所有属性

“java.home” 和 “java.vm.*” 就是这样的例子。

例如，下面的权限项允许程序读取以 *java.vm* 开头的所有属性。

```
permission java.util.PropertyPermission "java.vm.*", "read";
```

可以在策略文件中使用系统属性，其中的 \${*property*} 标记会被属性值替代，例如，\${*user.home*} 会被用户主目录替代。下面是在访问权限项中使用系统属性的典型应用。

```
permission java.io.FilePermission "${user.home}", "read,write";
```

为了创建平台无关的策略文件，使用 *file.separator* 属性而不是使用显式的 / 或者 \\ 分隔符绝对是个好主意。如果要使它更加简单，可以使用符号 \${/} 作为 \${*file.separator*} 的缩写。例如，

```
permission java.io.FilePermission "${user.home}${/}-", "read,write";
```

是一个可在平台之间移植的项，用于授予对在用户的主目录及其子目录中的文件进行读写的权限。

10.2.4 定制权限

在本节中，我们将要介绍如何把自己的权限类提供给用户，以使得他们可以在策略文件中引用这些权限类。

如果要实现自己的权限类，可以继承 *Permission* 类，并提供以下方法：

- 带有两个 String 参数的构造器，这两个参数分别是目标和操作列表
- `String getActions()`
- `boolean equals(Object other)`
- `int hashCode()`
- `boolean implies(Permission other)`

最后一个方法是最重要的。权限有一个排序，其中更加泛化的权限隐含了更加具体的权限。请考虑下面的文件权限：

```
p1 = new FilePermission("/tmp/-", "read, write");
```

该权限允许读写 /tmp 目录以及子目录中的任何文件。

该权限隐含了其他更加具体的权限：

```
p2 = new FilePermission("/tmp/-", "read");
p3 = new FilePermission("/tmp/aFile", "read, write");
p4 = new FilePermission("/tmp/aDirectory/-", "write");
```

换句话说，如果

1. p1 的目标文件集包含 p2 的目标文件集。
2. p1 的操作集包含 p2 的操作集。

那么，文件访问权限 p1 就隐含了另一个文件访问权限 p2。

请考虑下面关于 `implies` 方法的用法举例。当 `InputStream` 构造器想要打开一个文件，以读取该文件时，要检查它是否拥有操作权限。如果要执行这种检查，就应将一个具体的文件权限对象传递给 `checkPermission` 方法：

```
checkPermission(new FilePermission(fileName, "read"));
```

现在安全管理器询问所有适用的权限是否隐含了该权限。如果其中某个隐含了该权限，就通过了检查。

特别地，`AllPermission` 隐含了其他所有的权限。

如果你定义了自己的权限类，那么必须对权限对象定义一个合适的隐含法则。例如，假设你为采用 Java 技术的机顶盒定义一个 `TVPermission`，那么下面这个访问权限

```
new TVPermission("Tommy:2-12:1900-2200", "watch, record")
```

将允许 Tommy 在 19 点到 22 点之间对 2 至 12 频道的电视节目进行观看和录像。必须实现 `implies` 方法，以隐含像下面这样的更具体的权限。

```
new TVPermission("Tommy:4:2000-2100", "watch")
```

10.2.5 实现权限类

在下面这个示例程序中，我们实现了一个新的权限，用于监视将文本插入到文本域的操作。该程序会确保你不能输入“不良单词”，例如 `sex`, `drugs` 以及 `C++` 等。我们使用了一个定制的权限类，以便在策略文件中提供这些不良单词。

下面这个 `JTextArea` 的子类询问安全管理器是否准备好了去添加新文本。

```

class WordCheckTextArea extends JTextArea
{
    public void append(String text)
    {
        var p = new WordCheckPermission(text, "insert");
        SecurityManager manager = System.getSecurityManager();
        if (manager != null) manager.checkPermission(p);
        super.append(text);
    }
}

```

如果安全管理器赋予了 `WordCheckPermission` 权限，那么该文本就可以追加。否则，`checkPermission` 方法就会抛出一个异常。

单词检查权限有两个可能的操作，一个是 `insert`（用于插入具体文本的权限），另一个是 `avoid`（添加不包含某些不良单词的任何文本的权限）。应该用下面的策略文件运行这个程序：

```

grant
{
    permission permissions.WordCheckPermission "sex,drugs,C++", "avoid";
};

```

这个策略文件赋予的权限是可以插入不包含不良单词 `sex`, `drugs` 和 `C++` 的任何文本。

当设计 `WordCheckPermission` 类时，我们必须特别注意 `implies` 方法，下面是控制权限 `p1` 是否隐含 `p2` 的规则：

- 如果 `p1` 有 `avoid` 操作，`p2` 有 `insert` 操作，那么 `p2` 的目标必须避开 `p1` 中的所有单词。例如，下面这个权限：

```
permissions.WordCheckPermission "sex,drugs,C++", "avoid"
```

隐含了下面这个权限：

```
permissions.WordCheckPermission "Mary had a little lamb", "insert"
```

- 如果 `p1` 和 `p2` 都有 `avoid` 操作，那么 `p2` 的单词集合必须包含 `p1` 单词集合中的所有单词。例如，下面这个权限：

```
permissions.WordCheckPermission "sex,drugs", "avoid"
```

隐含了下面这个权限：

```
permissions.WordCheckPermission "sex,drugs,C++", "avoid"
```

- 如果 `p1` 和 `p2` 都有 `insert` 操作，那么 `p1` 的文本必须包含 `p2` 的文本。例如，下面这个权限：

```
permissions.WordCheckPermission "Mary had a little lamb", "insert"
```

包含了下面这个权限：

```
permissions.WordCheckPermission "a little lamb", "insert"
```

可以在程序清单 10-4 中看到该类的具体实现。

请注意，可以用 `Permission` 类中名字容易混淆的 `getName` 方法来获取权限的目标。

由于在策略文件中权限是由一对字符串来表示的，因此，权限类需要准备好解析这些字

字符串。特别地，我们应该使用下面的方法，将用逗号分隔的 avoid 权限的不良单词表转换为一个真正的 Set。

```
public Set<String> badWordSet()
{
    var set = new HashSet<String>();
    set.addAll(List.of(getName().split(",")));
    return set;
}
```

该代码允许我们用 equals 和 containsAll 方法来比较这些集。正如我们在卷 I 第 9 章中所介绍的那样，如果两个集包含任意次序的相同元素，那么集类的 equals 方法可以判定它们相等。例如，由 "sex,drugs,C++" 和 "C++,drugs,sex" 产生的两个集是相等的集。

◆ 警告：务必把你的权限类设为 public。策略文件加载器不能加载具有包可视性的类，并且它会悄悄忽略其无法找到的所有类。

程序清单 10-5 中的程序展示了 WordCheckPermission 类是如何工作的。请在文本框内输入任意文本，然后按下 Insert 按钮。如果文本通过了安全检查，该文本就会被添加到文本区域中。如果没有通过检查，就会弹出一个消息（参见图 10-7）。

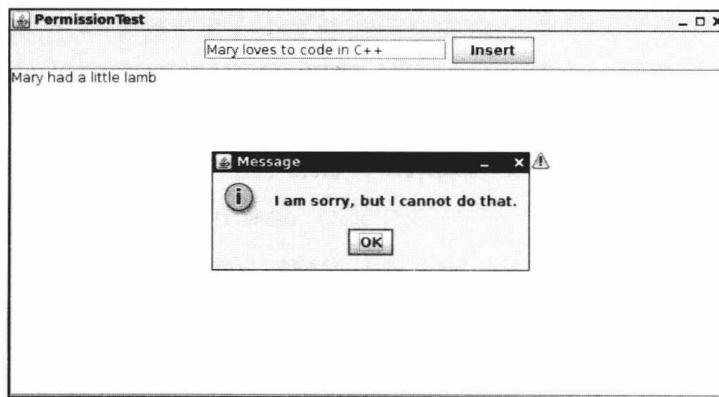


图 10-7 PermissionTest 程序

你现在已经看到应该如何配置 Java 平台的安全性了。更常见的情况是，你只需微调标准的权限集。对于其他额外的控制，你可以定义自制的权限，它们应该可以按照与标准权限相同的方式配置。

程序清单 10-4 permissions/WordCheckPermission.java

```
1 package permissions;
2
3 import java.security.*;
4 import java.util.*;
5
6 /**
7  * A permission that checks for bad words.
8 */
```

```
9 public class WordCheckPermission extends Permission
10 {
11     private String action;
12
13     /**
14      * Constructs a word check permission.
15      * @param target a comma separated word list
16      * @param anAction "insert" or "avoid"
17      */
18     public WordCheckPermission(String target, String anAction)
19     {
20         super(target);
21         action = anAction;
22     }
23
24     public String getActions()
25     {
26         return action;
27     }
28
29     public boolean equals(Object other)
30     {
31         if (other == null) return false;
32         if (!getClass().equals(other.getClass())) return false;
33         var b = (WordCheckPermission) other;
34         if (!Objects.equals(action, b.action)) return false;
35         if ("insert".equals(action)) return Objects.equals(getName(), b.getName());
36         else if ("avoid".equals(action)) return badWordSet().equals(b.badWordSet());
37         else return false;
38     }
39
40     public int hashCode()
41     {
42         return Objects.hash(getName(), action);
43     }
44
45     public boolean implies(Permission other)
46     {
47         if (!(other instanceof WordCheckPermission)) return false;
48         var b = (WordCheckPermission) other;
49         if (action.equals("insert"))
50         {
51             return b.action.equals("insert") && getName().indexOf(b.getName()) >= 0;
52         }
53         else if (action.equals("avoid"))
54         {
55             if (b.action.equals("avoid")) return b.badWordSet().containsAll(badWordSet());
56             else if (b.action.equals("insert"))
57             {
58                 for (String badWord : badWordSet())
59                     if (b.getName().indexOf(badWord) >= 0) return false;
60                 return true;
61             }
62         }
63         else return false;
64     }
65 }
```

```

63     }
64     else return false;
65 }
66
67 /**
68 * Gets the bad words that this permission rule describes.
69 * @return a set of the bad words
70 */
71 public Set<String> badWordSet()
72 {
73     var set = new HashSet<String>();
74     set.addAll(List.of(getName().split(",")));
75     return set;
76 }
77 }
```

程序清单 10-5 permissions/PermissionTest.java

```

1 package permissions;
2
3 import java.awt.*;
4
5 import javax.swing.*;
6
7 /**
8  * This class demonstrates the custom WordCheckPermission.
9  * @version 1.05 2018-05-01
10 * @author Cay Horstmann
11 */
12 public class PermissionTest
13 {
14     public static void main(String[] args)
15     {
16         System.setProperty("java.security.policy", "permissions/PermissionTest.policy");
17         System.setSecurityManager(new SecurityManager());
18         EventQueue.invokeLater(() ->
19         {
20             var frame = new PermissionTestFrame();
21             frame.setTitle("PermissionTest");
22             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23             frame.setVisible(true);
24         });
25     }
26 }
27
28 /**
29  * This frame contains a text field for inserting words into a text area that is protected
30  * from "bad words".
31 */
32 class PermissionTestFrame extends JFrame
33 {
34     private JTextField textField;
35     private WordCheckTextArea textArea;
36     private static final int TEXT_ROWS = 20;
```

```

37     private static final int TEXT_COLUMNS = 60;
38
39     public PermissionTestFrame()
40     {
41         textField = new JTextField(20);
42         var panel = new JPanel();
43         panel.add(textField);
44         var openButton = new JButton("Insert");
45         panel.add(openButton);
46         openButton.addActionListener(event -> insertWords(textField.getText()));
47
48         add(panel, BorderLayout.NORTH);
49
50         textArea = new WordCheckTextArea();
51         textArea.setRows(TEXT_ROWS);
52         textArea.setColumns(TEXT_COLUMNS);
53         add(new JScrollPane(textArea), BorderLayout.CENTER);
54         pack();
55     }
56
57 /**
58 * Tries to insert words into the text area. Displays a dialog if the attempt fails.
59 * @param words the words to insert
60 */
61 public void insertWords(String words)
62 {
63     try
64     {
65         textArea.append(words + "\n");
66     }
67     catch (SecurityException ex)
68     {
69         JOptionPane.showMessageDialog(this, "I am sorry, but I cannot do that.");
70         ex.printStackTrace();
71     }
72 }
73 }
74 /**
75 * A text area whose append method makes a security check to see that no bad words are added.
76 */
77
78 class WordCheckTextArea extends JTextArea
79 {
80     public void append(String text)
81     {
82         var p = new WordCheckPermission(text, "insert");
83         SecurityManager manager = System.getSecurityManager();
84         if (manager != null) manager.checkPermission(p);
85         super.append(text);
86     }
87 }

```

API **java.security.Permission 1.2**

- **Permission(String name)**

用指定的目标名构建一个权限。

- `String getName()`

返回该权限的对象名称。

- `boolean implies(Permission other)`

检查该权限是否隐含了 other 权限。如果 other 权限描述了一个更加具体的条件，而这个具体条件是由该权限所描述的条件所产生的结果，那么该权限就隐含这个 other 权限。

10.3 用户认证

Java API 提供了一个名为 Java 认证和授权服务的框架，它将平台提供的认证与权限管理集成起来。我们将在以下各节中讨论 JAAS 框架。

10.3.1 JAAS 框架

正如其名字所表示的，Java 认证和授权服务（JAAS，Java Authentication and Authorization Service）包含两部分：“认证”部分主要负责确定程序使用者的身份，而“授权”将各个用户映射到相应的权限。

JAAS 是一个可插拔的 API，可以将 Java 应用程序与实现认证的特定技术分离开来。除此之外，JAAS 还支持 UNIX 登录、NT 登录、Kerberos 认证和基于证书的认证。

一旦用户通过认证，就可以为其附加一组权限。例如，这里我们赋予 Harry 一个特定的权限集，而其他用户则没有，它的语法规则如下：

```
grant principal com.sun.security.auth.UnixPrincipal "harry"
{
    permission java.util.PropertyPermission "user.*", "read";
    ...
};
```

`com.sun.security.auth.UnixPrincipal` 类检查运行该程序的 UNIX 用户的名字，它的 `getName` 方法将返回 UNIX 登录名，然后我们就可以检查该名称是否等于 "harry"。

可以使用一个 `LoginContext` 以使得安全管理器能够检查这样的授权语句。下面是登录代码的基本轮廓：

```
try
{
    System.setSecurityManager(new SecurityManager());
    var context = new LoginContext("Login1"); // defined in JAAS configuration file
    context.login();
    // get the authenticated Subject
    Subject subject = context.getSubject();
    ...
    context.logout();
}
```

```

catch (LoginException exception) // thrown if login was not successful
{
    exception.printStackTrace();
}

```

这里，`subject` 是指已经被认证的个体。

`LoginContext` 构造器中的字符串参数 "Login1" 是指 JAAS 配置文件中具有相同名字的项。

下面是一个简单的配置文件：

```

Login1
{
    com.sun.security.auth.module.UnixLoginModule required;
    com.whizzbang.auth.module.RetinaScanModule sufficient;
};

Login2
{
    ...
};

```

当然，JDK 中没有包含任何使用 biometric 的登录模块。JDK 在 `com.sun.security.auth.module` 包中包含以下模块：

```

UnixLoginModule
NTLoginModule
Krb5LoginModule
JndiLoginModule
KeyStoreLoginModule

```

一个登录策略由一个登录模块序列组成，每个模块被标记为 `required`、`sufficient`、`requisite` 或 `optional`。这些关键字的含义在下面的算法中进行了描述：

登录时要对登录的主体（`subject`）进行认证，该主体可以拥有多个特征（`principal`）。特征描述了主体的某些属性，比如用户名、组 ID 或角色等。我们在 `grant` 语句中可以看到，特征管制着各个权限。`com.sun.security.auth.UnixPrincipal` 类描述了 UNIX 登录名，`UnixNumericGroupPrincipal` 类可以用来检测用户是否归属于某个 UNIX 用户组。

使用下面的语法，`grant` 语句可以对一个特征进行测试：

```
grant principalClass "principalName"
```

例如：

```
grant com.sun.security.auth.UnixPrincipal "harry"
```

当用户登录后，就会在独立的访问控制上下文中，运行要求检查用户特征的代码。使用静态的 `doAs` 或 `doAsPrivileged` 方法，启动一个新的 `PrivilegedAction`，其 `run` 方法就会执行这段代码。

这两个方法都可以通过使用主体特征的权限来调用某个对象的 `run` 方法去执行特定操作，而该对象必须是实现了 `PrivilegedAction` 接口的对象。

```

PrivilegedAction<T> action = () ->
{

```

```

// run with permissions of subject principals
. . .
};

T result
= Subject.doAs(subject, action); // or Subject.doAsPrivileged(subject, action, null)

```

如果该操作会抛出受检查的异常，那么必须改为实现 `PrivilegedExceptionAction` 接口。

`doAs` 和 `doAsPrivileged` 方法之间的区别是微小的。`doAs` 方法开始于当前的访问控制上下文，而 `doAsPrivileged` 方法则开始于一个新的上下文。后者允许将登录代码和“业务逻辑”的权限相分离。在我们的示例应用程序中，登录代码有如下权限：

```

permission javax.security.auth.AuthPermission "createLoginContext.Login1";
permission javax.security.auth.AuthPermission "doAsPrivileged";

```

通过认证的用户有一个权限：

```
permission java.util.PropertyPermission "user.*", "read";
```

如果我们用 `doAs` 代替了 `doAsPrivileged`，那么登录代码也需要这个权限！

程序清单 10-6 和程序清单 10-7 的程序展示了如何限制某些用户的权限。`AuthTest` 程序对用户的身份进行了认证，然后运行了一个简单的操作，以获得一个系统属性。

程序清单 10-6 auth/AuthTest.java

```

1 package auth;
2
3 import javax.security.auth.*;
4 import javax.security.auth.login.*;
5
6 /**
7 * This program authenticates a user via a custom login and then executes the SysPropAction
8 * with the user's privileges.
9 * @version 1.02 2018-05-01
10 * @author Cay Horstmann
11 */
12 public class AuthTest
13 {
14     public static void main(final String[] args)
15     {
16         System.setSecurityManager(new SecurityManager());
17         try
18         {
19             var context = new LoginContext("Login1");
20             context.login();
21             System.out.println("Authentication successful.");
22             Subject subject = context.getSubject();
23             System.out.println("subject=" + subject);
24             var action = new SysPropAction("user.home");
25             String result = Subject.doAsPrivileged(subject, action, null);
26             System.out.println(result);
27             context.logout();
28         }
29         catch (LoginException e)

```

```

30     {
31         e.printStackTrace();
32     }
33 }
34 }
```

程序清单 10-7 auth/SysPropAction.java

```

1 package auth;
2
3 import java.security.*;
4
5 /**
6  * This action looks up a system property.
7  * @version 1.01 2007-10-06
8  * @author Cay Horstmann
9 */
10 public class SysPropAction implements PrivilegedAction<String>
11 {
12     private String propertyName;
13
14     /**
15      Constructs an action for looking up a given property.
16      @param propertyName the property name (such as "user.home")
17     */
18     public SysPropAction(String propertyName)
19     {
20         this.propertyName = propertyName;
21     }
22
23     public String run()
24     {
25         return System.getProperty(propertyName);
26     }
27 }
```

要使该例子能够运行，必须将登录类和操作类的代码封装到两个独立的 JAR 文件中：

```

javac auth/*.java
jar cvf login.jar auth/AuthTest.class
jar cvf action.jar auth/SysPropAction.class
```

如果查看程序清单 10-8 中的策略文件，将会看到名为 harry 的 UNIX 用户拥有读取所有文件的权限。将 harry 改为自己的登录名，然后运行下面的命令

```

java -classpath login.jar:action.jar \
-Djava.security.policy=auth/AuthTest.policy \
-Djava.security.auth.login.config=auth/jaas.config \
auth.AuthTest
```

程序清单 10-8 auth/AuthTest.policy

```

1 grant codebase "file:login.jar"
2 {
```

```

3     permission javax.security.auth.AuthPermission "createLoginContext.Login1";
4     permission javax.security.auth.AuthPermission "doAsPrivileged";
5 };
6
7 grant principal com.sun.security.auth.UnixPrincipal "harry"
8 {
9     permission java.util.PropertyPermission "user.*", "read";
10 };

```

程序清单 10-9 展示了登录的配置。

程序清单 10-9 auth/jaas.config

```

1 Login1
2 {
3     com.sun.security.auth.module.UnixLoginModule required;
4 };

```

在 Windows 下运行时，请将 AuthTest.policy 中的 UnixPrincipal 改为 NTUser-Principal，并将 jaas.config 中的 UnixLoginModule 改为 NTLoginModule。运行该程序时，请用分号来分隔各个 JAR 文件：

```
java -classpath login.jar;action.jar . . .
```

AuthTest 程序现在将显示 user.home 属性的值。但是，如果用不同的名字登录，那么就应该抛出一个安全异常，因为你不再拥有必需的权限了。

！ 警告：必须严格按照这些指令来运行。如果对程序进行了一些看上去无关紧要的更改，那就很容易使你的设置出错。

API **javax.security.auth.login.LoginContext 1.4**

- `LoginContext(String name)`

创建一个登录上下文。name 对应于 JAAS 配置文件中的登录描述符。

- `void login()`

建立一个登录操作，如果登录失败，则抛出一个 `LoginException` 异常。它会调用 JAAS 配置文件中的管理器上的 `login` 方法。

- `void logout()`

Subject 退出登录。它会调用 JAAS 配置文件中的管理器上的 `logout` 方法。

- `Subject getSubject()`

返回认证过的 Subject。

API **javax.security.auth.Subject 1.4**

- `Set<Principal> getPrincipals()`

获取该 Subject 的各个 Principal。

- static Object doAs(Subject subject, PrivilegedAction action)
- static Object doAs(Subject subject, PrivilegedExceptionAction action)
- static Object doAsPrivileged(Subject subject, PrivilegedAction action, AccessControlContext context)
- static Object doAsPrivileged(Subject subject, PrivilegedExceptionAction action, AccessControlContext context)

以 subject 的身份执行特许操作。它将返回 run 方法的返回值。doAsPrivileged 方法在给定的访问控制上下文中执行该操作，你可以提供一个在前面调用静态方法 AccessController.getContext() 时所获得的“上下文快照”，或者指定为 null，以便使其在一个新的上下文中执行该代码。

API `java.security.PrivilegedAction` 1.4

- Object run()
- 必须定义该方法，以执行你想要代表某个主体去执行的代码。

API `java.security.PrivilegedExceptionAction` 1.4

- Object run()
- 必须定义该方法，以执行你想要代表某个主体去执行的代码。本方法可以抛出任何受检查的异常。

API `java.security.Principal` 1.1

- String getName()
- 返回该特征的身份标识。

10.3.2 JAAS 登录模块

在本节中，我们将要用一个 JAAS 例子向读者介绍：

- 如何实现你自己的登录模块；
- 如何实现基于角色的认证。

如果登录信息存储在数据库中，那么使用自己的登录模块就非常有用。尽管你可能很喜欢默认的登录模块，但是学习如何定制自己的模块将有助于你理解 JAAS 配置文件的各个选项。

基于角色的认证对于大量用户的管理来说是十分必要的。将所有合法用户的名字都写入策略文件是不切实际的。而登录模块应该将用户映射到诸如“admin”或“HR”等角色，并且权限的赋予也要基于这些角色。

登录模块的工作之一是组装被认证的主体的特征集。如果一个登录模块支持某些角色，该模块就会添加 Principal 对象来描述这些角色。JDK 并没有提供相应的类，所以我们写了自己的类（见程序清单 10-10）。该类直接存储了一个描述 / 值对，例如 role=admin。该类的 getName 方法用于返回该描述 / 值对，因此我们就可以添加基于角色的权限到策略文件中：

```
grant principal SimplePrincipal "role=admin" { . . . }
```

程序清单 10-10 jaas/SimplePrincipal.java

```

1 package jaas;
2
3 import java.security.*;
4 import java.util.*;
5
6 /**
7  * A principal with a named value (such as "role=HR" or "username=harry").
8 */
9 public class SimplePrincipal implements Principal
10 {
11     private String descr;
12     private String value;
13
14     /**
15      * Constructs a SimplePrincipal to hold a description and a value.
16      * @param descr the description
17      * @param value the associated value
18      */
19     public SimplePrincipal(String descr, String value)
20     {
21         this.descr = descr;
22         this.value = value;
23     }
24
25     /**
26      * Returns the role name of this principal.
27      * @return the role name
28      */
29     public String getName()
30     {
31         return descr + "=" + value;
32     }
33
34     public boolean equals(Object otherObject)
35     {
36         if (this == otherObject) return true;
37         if (otherObject == null) return false;
38         if (getClass() != otherObject.getClass()) return false;
39         var other = (SimplePrincipal) otherObject;
40         return Objects.equals(getName(), other.getName());
41     }
42
43     public int hashCode()
44     {
45         return Objects.hashCode(getName());
46     }
47 }
```

我们的登录模块会在包含如下行的文本文件中查找用户、密码和角色：

```
harry|secret|admin
carl|guessme|HR
```

当然，在实际的登录模块中，你可能会将这些信息存储在数据库或者目录中。

在程序清单 10-11 中可以找到 SimpleLoginModule 的代码，其 checkLogin 方法用于检查输入的用户名和密码是否与密码文件中的用户记录相匹配。如果匹配成功，则会添加两个 SimplePrincipal 对象到主体的特征集中。

```
Set<Principal> principals = subject.getPrincipals();
principals.add(new SimplePrincipal("username", username));
principals.add(new SimplePrincipal("role", role));
```

程序清单 10-11 jaas/SimpleLoginModule.java

```
1 package jaas;
2
3 import java.io.*;
4 import java.nio.charset.*;
5 import java.nio.file.*;
6 import java.security.*;
7 import java.util.*;
8 import javax.security.auth.*;
9 import javax.security.auth.callback.*;
10 import javax.security.auth.login.*;
11 import javax.security.auth.spi.*;
12
13 /**
14  * This login module authenticates users by reading usernames, passwords, and roles from
15  * a text file.
16 */
17 public class SimpleLoginModule implements LoginModule
18 {
19     private Subject subject;
20     private CallbackHandler callbackHandler;
21     private Map<String, ?> options;
22
23     public void initialize(Subject subject, CallbackHandler callbackHandler,
24                           Map<String, ?> sharedState, Map<String, ?> options)
25     {
26         this.subject = subject;
27         this.callbackHandler = callbackHandler;
28         this.options = options;
29     }
30
31     public boolean login() throws LoginException
32     {
33         if (callbackHandler == null) throw new LoginException("no handler");
34
35         var nameCall = new NameCallback("username: ");
36         var passCall = new PasswordCallback("password: ", false);
37         try
38         {
39             callbackHandler.handle(new Callback[] { nameCall, passCall });

```

```

40     }
41     catch (UnsupportedCallbackException e)
42     {
43         var e2 = new LoginException("Unsupported callback");
44         e2.initCause(e);
45         throw e2;
46     }
47     catch (IOException e)
48     {
49         var e2 = new LoginException("I/O exception in callback");
50         e2.initCause(e);
51         throw e2;
52     }
53
54     try
55     {
56         return checkLogin(nameCall.getName(), passCall.getPassword());
57     }
58     catch (IOException ex)
59     {
60         var ex2 = new LoginException();
61         ex2.initCause(ex);
62         throw ex2;
63     }
64 }
65
66 /**
67 * Checks whether the authentication information is valid. If it is, the subject acquires
68 * principals for the user name and role.
69 * @param username the user name
70 * @param password a character array containing the password
71 * @return true if the authentication information is valid
72 */
73 private boolean checkLogin(String username, char[] password)
74     throws LoginException, IOException
75 {
76     try (var in = new Scanner(
77         Paths.get("") + options.get("pwfile")), StandardCharsets.UTF_8))
78     {
79         while (in.hasNextLine())
80         {
81             String[] inputs = in.nextLine().split("\\|");
82             if (inputs[0].equals(username)
83                 && Arrays.equals(inputs[1].toCharArray(), password))
84             {
85                 String role = inputs[2];
86                 Set<Principal> principals = subject.getPrincipals();
87                 principals.add(new SimplePrincipal("username", username));
88                 principals.add(new SimplePrincipal("role", role));
89                 return true;
90             }
91         }
92         return false;
93     }

```

```

94 }
95
96 public boolean logout()
97 {
98     return true;
99 }
100
101 public boolean abort()
102 {
103     return true;
104 }
105
106 public boolean commit()
107 {
108     return true;
109 }
110 }

```

SimpleLoginModule 剩余的部分就非常直截了当了。initialize 方法接收下面几个参数：

- 用于认证的 Subject。
- 一个获取登录信息的 handler。
- 一个 sharedState 映射表，它可用于登录模块之间的通信。
- 一个 options 映射表，它包含了登录配置文件中设置的名 / 值对。

例如，我们将模块做如下配置：

```
SimpleLoginModule required pwfile="password.txt";
```

则登录模块可以从 options 映射表中获取 pwfile 设置。

该登录模块并没有收集用户名和密码，这是单独的 handler 需要做的工作。这种功能上的分离有助于在各种情况下使用相同的登录模块，而不用关心登录信息是来自 GUI 对话框、控制台提示符还是配置文件。

handler 是在创建 LoginContext 时指定的。例如，

```
var context = new LoginContext("Login1",
    new com.sun.security.auth.callback.DialogCallbackHandler());
```

DialogCallbackHandler 会弹出一个简单的 GUI 对话框，以获取用户名和密码。而 com.sun.security.auth.callback.TextCallbackHandler 则从控制台获取这些信息。

但是，在我们的应用程序中，是通过自己编写的 GUI 来获得用户名和密码的（参见图 10-8）。我们创建了一个简单的 handler，仅仅用于存储和返回这些信息（见程序清单 10-12）。

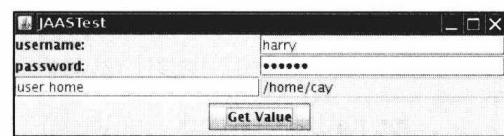


图 10-8 一个定制的登录模块

程序清单 10-12 jaas/SimpleCallbackHandler.java

```

1 package jaas;
2

```

```

3 import javax.security.auth.callback.*;
4
5 /**
6  * This simple callback handler presents the given user name and password.
7 */
8 public class SimpleCallbackHandler implements CallbackHandler
9 {
10     private String username;
11     private char[] password;
12
13     /**
14      * Constructs the callback handler.
15      * @param username the user name
16      * @param password a character array containing the password
17      */
18     public SimpleCallbackHandler(String username, char[] password)
19     {
20         this.username = username;
21         this.password = password;
22     }
23
24     public void handle(Callback[] callbacks)
25     {
26         for (Callback callback : callbacks)
27         {
28             if (callback instanceof NameCallback)
29             {
30                 ((NameCallback) callback).setName(username);
31             }
32             else if (callback instanceof PasswordCallback)
33             {
34                 ((PasswordCallback) callback).setPassword(password);
35             }
36         }
37     }
38 }

```

该 handler 有一个简单的方法 handle，用于处理 Callback 对象数组。有很多预定义类，比如 NameCallback 和 PasswordCallback 等，都实现了 Callback 接口。也可以添加自己的类，比如 RetinaScanCallback 等。下面这段 handler 代码可能有些不雅致，因为它要分析 callback 对象的类型：

```

public void handle(Callback[] callbacks)
{
    for (Callback callback : callbacks)
    {
        if (callback instanceof NameCallback) . . .
        else if (callback instanceof PasswordCallback) . . .
        else . . .
    }
}

```

登录模块提供 callback 数组以满足认证的需要。

```
var nameCall = new NameCallback("username: ");
var passCall = new PasswordCallback("password: ", false);
callbackHandler.handle(new Callback[] { nameCall, passCall });
```

然后它从 callback 中获取所要的信息。

程序清单 10-13 中的程序将显示一个窗体，用于输入登录信息和系统属性名。如果用户通过了认证，属性值会在 PrivilegedAction 中被取出。从程序清单 10-14 的策略文件中可以看到，只有具有 admin 角色的用户才具有对属性的读取权限。

程序清单 10-13 jaas/JAASTest.java

```
1 package jaas;
2
3 import java.awt.*;
4 import javax.swing.*;
5
6 /**
7  * This program authenticates a user via a custom login and then looks up a system property
8  * with the user's privileges.
9  * @version 1.03 2018-05-01
10 * @author Cay Horstmann
11 */
12 public class JAASTest
13 {
14     public static void main(final String[] args)
15     {
16         System.setSecurityManager(new SecurityManager());
17         EventQueue.invokeLater(() ->
18         {
19             var frame = new JAASFrame();
20             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21             frame.setTitle("JAASTest");
22             frame.setVisible(true);
23         });
24     }
25 }
```

程序清单 10-14 jaas/JAASTest.policy

```
1 grant codebase "file:login.jar"
2 {
3     permission java.awt.AWTPermission "showWindowWithoutWarningBanner";
4     permission java.awt.AWTPermission "accessEventQueue";
5     permission javax.security.auth.AuthPermission "createLoginContext.Login1";
6     permission javax.security.auth.AuthPermission "doAsPrivileged";
7     permission javax.security.auth.AuthPermission "modifyPrincipals";
8     permission java.io.FilePermission "jaas/password.txt", "read";
9 };
10
11 grant principal jaas.SimplePrincipal "role=admin"
12 {
13     permission java.util.PropertyPermission "*", "read";
14 };
```

正如前一节中所讲到的，必须将登录和操作代码分开。因此，首先创建两个 JAR 文件：

```
javac *.java
jar cvf login.jar JAAS*.class Simple*.class
jar cvf action.jar SysPropAction.class
```

然后以如下方式运行程序：

```
java -classpath login.jar:action.jar \
-Djava.security.policy=JAATest.policy \
-Djava.security.auth.login.config=jaas.config \
JAATest
```

程序清单 10-15 说明了登录的配置。

注释：有些应用有可能需要支持更复杂的两阶段协议，即只有登录配置文件中的所有模块都认证成功，该登录才会被提交。更多详细信息，请参阅下面地址的登录模块开发指南：<http://docs.oracle.com/javase/8/docs/technotes/guides/security/jaas/JAASLMDevGuide.html>。

程序清单 10-15 jaas/jaas.config

```
1 Login1
2 {
3     jaas.SimpleLoginModule required pwfile="jaas/password.txt" debug=true;
4 };
```

API **javax.security.auth.callback.CallbackHandler 1.4**

- **void handle(Callback[] callbacks)**

处理给定的 callback，如果愿意，可以与用户进行交互，并且将安全信息存储到 callback 对象中。

API **javax.security.auth.callback.NameCallback 1.4**

- **NameCallback(String prompt)**
- **NameCallback(String prompt, String defaultValue)**

用给定的提示符和默认的名字构建一个 NameCallback。

- **String getName()**
 - **void setName(String name)**
- 设置或者获取该 callback 所收集到的名字。
- **String getPrompt()**
- 获取查询该名字时所使用的提示符。
- **String getDefaultName()**
- 获取查询该名字时所使用的默认名字。

API **javax.security.auth.callback.PasswordCallback 1.4**

- **PasswordCallback(String prompt, boolean echoOn)**

用给定提示符和回显标记构建一个 PasswordCallback。

- `char[] getPassword()`
- `void setPassword(char[] password)`

设置或者获取该 callback 所收集到的密码。

- `String getPrompt()`
获取查询该密码时所使用的提示符。
- `boolean isEchoOn()`
获取查询该密码时所使用的回显标记。

API `javax.security.auth.spi.LoginModule 1.4`

- `void initialize(Subject subject, CallbackHandler handler, Map<String,?> sharedState, Map<String,?> options)`

为了认证给定的 subject，初始化该 LoginModule。在登录处理期间，用给定的 handler 来收集登录信息；使用 sharedState 映射表与其他登录模块进行通信；options 映射表包含该模块实例的登录配置中指定的名 / 值对。

- `boolean login()`
执行认证过程，并组装主体的特征集。如果登录成功，则返回 true。
- `boolean commit()`
对于需要两阶段提交的登录场景，当所有的登录模块都成功后，调用该方法。如果操作成功，则返回 true。
- `boolean abort()`
如果某一登录模块失败导致登录过程中断，就调用该方法。如果操作成功，则返回 true。
- `boolean logout()`
注销当前的主体。如果操作成功，则返回 true。

10.4 数字签名

正如我们前面所说，applet 是在 Java 平台上开始流行起来的。实际上，人们发现尽管他们可以编写出像著名的“nervous text”那样栩栩如生的 applet，但是在 JDK 1.0 安全模式下无法发挥其一整套非常有用的作用。例如，由于 JDK 1.0 下的 applet 要受到严密的监管，因此，即使 applet 在公司安全内部网上运行时风险相对较小，applet 也无法在企业内部网上发挥很大的作用。Sun 公司很快就认识到，要使 applet 真正变得非常有用，用户必须可以根据 applet 的来源为其分配不同的安全级别。如果 applet 来自值得信赖的提供商，并且没有被篡改过，那么 applet 的用户就可以决定是否给 applet 授予更多的运行特权。

如果要给予一个 applet 更多的信任，你必须知道下面两件事：

1. 这个 applet 来自哪里?
2. 在传输过程中代码是否被破坏?

在过去的 50 年里, 数学家和计算机科学家已经开发出各种各样成熟的算法, 用于确保数据和电子签名的完整性, 在 `java.security` 包中包含了许多这类算法的实现, 而且幸运的是, 你无须掌握相应的数学基础知识, 就可以使用 `java.security` 包中的算法。在下面几节中, 我们将要介绍消息摘要是如何检测数据文件中的变化的, 以及数字签名是如何证明签名者的身份的。

10.4.1 消息摘要

消息摘要 (message digest) 是数据块的数字指纹。例如, 所谓的 SHA1 (安全散列算法 #1) 可将任何数据块, 无论其数据有多长, 都压缩为 160 位 (20 字节) 的序列。与真实的指纹一样, 人们希望任何两条不同的消息都不会有相同的 SHA1 指纹。当然, 这是不可能的一因为只存在 2^{160} 个 SHA1 指纹, 所以肯定会有某些消息具有相同的指纹。因为 2^{160} 是一个很大的数字, 所以存在重复指纹的可能性微乎其微, 那么这种重复的可能性到底小到什么程度呢? 根据 James Walsh 在他的 *True Odds: How Risks Affect Your Everyday Life* (Merritt Publishing 出版社 1996 年出版) 一书中所叙述的, 人死于雷击的概率为三万分之一。现在, 假设有 9 个人, 比如你不喜欢的 9 个经理或者教授, 你和他们所有的人都死于雷击的概率, 比伪造的消息与原有消息具有相同的 SHA1 指纹的概率还要高。(当然, 可能有你不认识的其他 10 个以上的人会死于雷击, 但这里我们讨论的是你选择的特定的人的死亡概率。)

消息摘要具有两个基本属性:

1. 如果数据的 1 位或者几位改变了, 那么消息摘要也将改变。
2. 拥有给定消息的伪造者无法创建与原消息具有相同摘要的假消息。

当然, 第二个属性又是一个概率问题。让我们来看看下面这位亿万富翁留下的遗嘱:

“我死了之后, 我的财产将由我的孩子平分, 但是, 我的儿子 George 应该拿不到一个子。”这份遗嘱的 SHA1 指纹为:

```
12 5F 09 03 E7 31 30 19 2E A6 E7 E4 90 43 84 B4 38 99 8F 67
```

这位有疑心病的父亲将这份遗嘱交给一位律师保存, 而将指纹交给另一位律师保存。现在, 假设 George 能够贿赂那位保存遗嘱的律师, 他想修改这份遗嘱, 使得 Bill 一无所获。当然, 这需要将原指纹改为下面这样完全不同的位模式:

```
7D F6 AB 08 EB 40 EC CD AB 74 ED E9 86 F9 ED 99 D1 45 B1 57
```

那么 George 能够找到与该指纹相匹配的其他措辞吗? 如果从地球形成之时, 他就很自豪地拥有 10 亿台计算机, 每台计算机每秒钟能处理一百万条信息, 他依然无法找到一个能够替换的遗嘱。

人们已经设计出大量的算法, 用于计算这些消息摘要, 其中最著名的两种算法是 SHA1 和 MD5。SHA1 是由美国国家标准和技术学会开发的加密散列算法, MD5 是由麻省理工学院的 Ronald Rivest 发明的算法。这两种算法都使用了独特巧妙的方法对消息中的各个位进行