

第11章 异常、日志、断言和调试

处理异常

捕获异常

使用异常机制的技巧

使用断言

日志

调试技巧

使用调试器

在理想状态下，用户输入数据的格式永远都是正确的，选择打开的文件也一定存在，并且永远不会出现bug。迄今为止，本书呈现给大家的代码似乎都处在这样一个理想境界中。然而，在现实世界中却充满了不良的数据和带有问题的代码，现在是讨论Java程序设计语言处理这些问题的机制的时候了。

人们在遇到错误时会感觉不爽。如果一个用户在运行程序期间，由于程序的错误或一些外部环境的影响造成用户数据的丢失，用户就有可能不再使用这个程序了。为了避免这类事情的发生，至少应该做到以下几点：

- 向用户通告错误；
- 保存所有的操作结果；
- 允许用户以适当的形式退出程序。

对于异常情况，例如，可能造成程序崩溃的错误输入，Java使用一种称为异常处理（exception handling）的错误捕获机制处理。Java中的异常处理与C++或Delphi中的异常处理十分类似。本章的第1部分先介绍Java的异常。

在测试期间，需要进行大量的检测以验证程序操作的正确性。然而，这些检测可能非常耗时，在测试完成后也不必保留它们，因此，可以将这些检测删掉，并在其他测试需要时将它们粘贴回来，这是一件很乏味的事情。本章的第2部分将介绍如何使用断言来选择检测行为。

当程序出现错误时，并不总是能够与用户或终端进行沟通。此时，可能希望记录下出现的问题，以备日后进行分析。本章的第3部分将讨论Java SE的日志工具。

最后，介绍如何获得一个正在运行的Java应用程序的有用信息，以及如何使用IDE中的调试器的技巧。

11.1 处理异常

假设在一个Java程序运行期间出现了一个错误。这个错误可能是由于文件包含了错误信息，或者网络连接出现问题造成的，也有可能是因为使用无效的数组下标，或者试图使用一个没有被赋值的对象引用而造成的。用户期望在出现错误时，程序能够采用一些理智的行为。如果由于出现错误而使得某些操作没有完成，程序应该：

- 返回到一种安全状态，并能够让用户执行一些其他的命令；或者
- 允许用户保存所有操作的结果，并以适当的方式终止程序。

要做到这些并不是一件很容易的事情。其原因是检测（或引发）错误条件的代码通常离那些能够让数据恢复到安全状态，或者能够保存用户的操作结果，并正常地退出程序的代码很远。异常处理的任务就是将控制权从错误产生的地方转移给能够处理这种情况的错误处理器。为了能够在程序中处理异常情况，必须研究程序中可能会出现错误和问题，以及哪类问题需要关注。

1. 用户输入错误

除了那些不可避免的打字录入外，有些用户喜欢各行其是，不遵守程序的要求。例如，假设有一个用户请求连接一个URL，而语法却不正确。在程序代码中应该对此进行检查，如果没有检查，网络数据包就会给出警告。

2. 设备错误

硬件并不总是让它做什么，它就做什么。打印机可能被关掉了。网页可能临时性地不能浏览。在一个任务的处理过程中，硬件经常出现问题。例如，打印机在打印过程中可能没有纸了。

3. 物理限制

磁盘满了，可用存储空间已被用完。

4. 代码错误

程序方法有可能无法正确的执行。例如，方法可能返回了一个错误的答案，或者错误地调用了其他的方法。使用了一个无效的数组下标，试图查找一个在散列表中不存在的数据项以及试图对一个空栈进行退栈操作。

对于方法中出现的错误，传统的处理方式是返回一个特定的错误编码，调用这个方法的方法对其进行分析。例如，对于一个从文件中读取信息的方法来说，如果返回值不是标准字符，而是一个 - 1，则表示文件结束。这种处理方式对于很多异常状况都是可行的。还有一种表示错误状况的常用返回值是null引用。在第10章中，将可以看到一个使用Applet类中的getParameter方法的示例。当希望查询的参数不存在时，这个方法就会返回null。

遗憾的是，并不是在任何情况下都能够返回一个错误编码。有可能无法明确地将有效数据与无效数据加以区分。一个返回整型的方法就不能简单地通过返回 - 1表示错误，因为 - 1很可能是一个完全合法的结果。

正如第5章中所叙述的那样，在Java中，如果某个方法不能够采用正常的途径完整它的任务，就可以通过另外一个路径退出方法。在这种情况下，方法并不返回任何值，而是抛出（throw）一个封装了错误信息的对象。需要注意的是，这个方法将会立刻退出，并不返回任何值。此外，调用这个方法的代码也将无法继续执行，而是，异常处理机制开始搜索能够处理这种异常状况的异常处理器（exception handler）。

异常具有自己的语法和特定的继承结构。下面首先介绍一下语法，然后再给出有效地使用这种语言功能的技巧。

11.1.1 异常分类

在Java程序设计语言中，异常对象都是派生于Throwable类的一个实例。稍后还可以看到，如果Java中内置的异常类不能够满足需求，用户可以创建自己的异常类。

图11-1是Java异常层次结构的一个简化示意图。

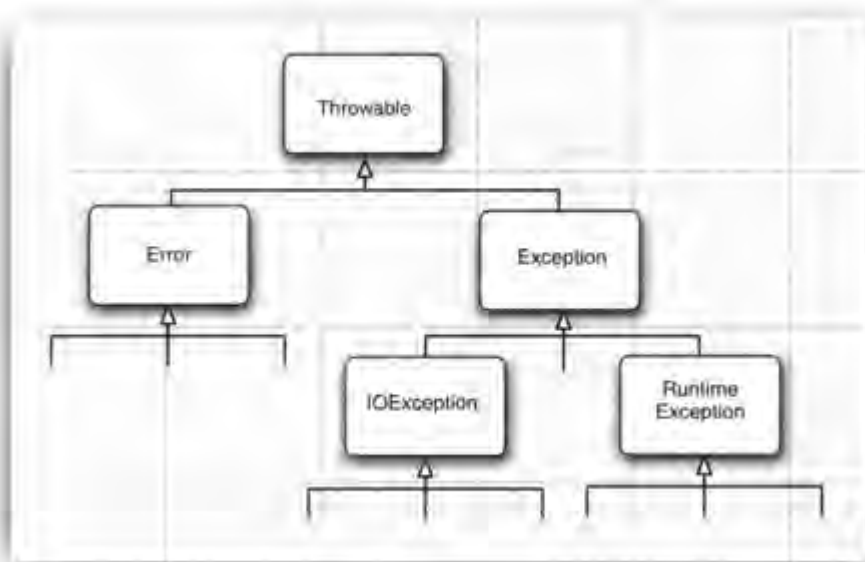


图11-1 Java中的异常层次结构

需要注意的是，所有的异常都是由Throwable继承而来，但在下一层立即分解为两个分支：Error和Exception。

Error类层次结构描述了Java运行时系统的内部错误和资源耗尽错误。应用程序不应该抛出这种类型的对象。如果出现了这样的内部错误，除了通告给用户，并尽力使程序安全地终止之外，再也无能为力了。这种情况很少出现。

在设计Java程序时，需要关注Exception层次结构。这个层次结构又分解为两个分支：一个分支派生于RuntimeException；另一个分支包含其他异常。划分两个分支的规则是：由程序错误导致的异常属于RuntimeException；而程序本身没有问题，但由于像I/O错误这类问题导致的异常属于其他异常。

派生于RuntimeException的异常包含下面几种情况：

- 错误的类型转换。
- 数组访问越界。
- 访问空指针。

不是派生于RuntimeException的异常包括

- 试图在文件尾部后面读取数据。
- 试图打开一个错误格式的URL。
- 试图根据给定的字符串查找Class对象，而这个字符串表示的类并不存在。

“如果出现RuntimeException异常，那么就一定是你的问题”是一条相当有道理的规则。应

该通过检测数组下标是否越界来避免`ArrayIndexOutOfBoundsException`异常；应该通过在使用变量之前检测是否为空来杜绝`NullPointerException`异常的发生。

如何处理错误格式的URL呢？在使用URL之前，是否也需要尽可能地判断是否“具有错误格式”呢？事实上，不同的浏览器可以处理不同类别的URL。例如，Netscape可以处理mailto:URL格式，而applet查看器就不能处理这种格式。因此，“具有错误格式”取决于具体的环境，而不仅仅是程序代码。

Java语言规范将派生于`Error`类或`RuntimeException`类的所有异常称为未检查（unchecked）异常，所有其他的异常称为已检查（checked）异常。这是两个很有用的术语，在后面还会用到。编译器将核查是否为所有的已检查异常提供了异常处理器。



注释：`RuntimeException`这个名字很容易让人混淆。实际上，现在讨论的所有错误都发生在运行时刻。



C++注释：如果熟悉标准C++类库中的异常层次结构，就一定会感到有些困惑。C++有两个基本的异常类，一个是`runtime_error`；另一个是`logic_error`。`logic_error`类相当于Java中的`RuntimeException`，它表示程序中的逻辑错误；`runtime_error`类是所有由于不可预测的原因所引发的异常的基类。它相当于Java中的非`RuntimeException`异常。

11.1.2 声明已检查异常

如果遇到了无法处理的情况，那么Java的方法可以抛出一个异常。这个道理很简单：一个方法不仅需要告诉编译器将要返回什么值，还要告诉编译器有可能发生什么错误。例如，一段读取文件的代码知道有可能读取的文件不存在，或者内容为空，因此，试图处理文件信息的代码就需要通告编译器可能会抛出`IOException`类的异常。

方法应该在其首部声明所有可能抛出的异常。这样可以从首部反映出这个方法可能抛出哪类已检查异常。例如，下面是标准类库中提供的`FileInputStream`类的一个构造器的声明（有关的更多信息请参看第12章。）

```
public FileInputStream(String name) throws FileNotFoundException
```

这个声明表示这个构造器将根据给定的String参数产生一个`FileInputStream`对象，但也有可能抛出一个`FileNotFoundException`异常。如果发生了这种糟糕情况，构造器将不会初始化一个新的`FileInputStream`对象，而是抛出一个`FileNotFoundException`类对象。如果这个方法真的抛出了这样一个异常对象，运行时系统就会开始搜索异常处理器，以便知道如何处理`FileNotFoundException`对象。

在自己编写方法时，不必将所有可能抛出的异常都进行声明。至于什么时候需要在方法中用throws子句声明异常，什么异常必须使用throws子句声明，需要记住在遇到下面4种情况时应该抛出异常：

- 1) 调用一个抛出已检查异常的方法，例如，`FileInputStream`构造器。

- 2) 程序运行过程中发现错误，并且利用throw语句抛出一个已检查异常（下一节将详细地介绍throw语句）。

3) 程序出现错误, 例如, `a[-1]=0` 会抛出一个 `ArrayIndexOutOfBoundsException` 这样的未检查异常。

4) Java 虚拟机和运行时库出现的内部异常。

如果出现前两种情况之一, 则必须告诉调用这个方法的程序员有可能抛出异常。因为任何一个抛出异常的方法都有可能是一个死亡陷阱。如果没有处理器捕获这个异常, 当前执行的线程就会结束。

对于那些可能被他人使用的 Java 方法, 应该根据异常规范 (exception specification), 在方法的首部声明这个方法可能抛出的异常。

```
class MyAnimation
{
    . . .
    public Image loadImage(String s) throws IOException
    {
        . . .
    }
}
```

如果一个方法有可能抛出多个已检查异常, 那么就必须在方法的首部列出所有的异常类。每个异常类之间用逗号隔开。如下面这个例子所示:

```
class MyAnimation
{
    . . .
    public Image loadImage(String s) throws EOFException, MalformedURLException
    {
        . . .
    }
}
```

但是, 不需要声明 Java 的内部错误, 即从 `Error` 继承的错误。任何程序代码都具有抛出那些异常的潜能, 而我们对其没有任何控制能力。

同样, 也不应该声明从 `RuntimeException` 继承的那些未检查异常。

```
class MyAnimation
{
    . . .
    void drawImage(int i) throws ArrayIndexOutOfBoundsException // bad style
    {
        . . .
    }
}
```

这些运行时错误完全在我们的控制之下。如果特别关注数组下标引发的错误, 就应该将更多的时间花费在修正程序中的错误上, 而不是说明这些错误发生的可能性上。

总之, 一个方法必须声明所有可能抛出的已检查异常, 而未检查异常要么不可控制 (`Error`), 要么就应该避免发生 (`RuntimeException`)。如果方法没有声明所有可能发生的已检查异常, 编译器就会给出一个错误消息。

当然, 从前面的示例中可以知道: 除了声明异常之外, 还可以捕获异常。这样会使异常不被抛到方法之外, 也不需要 `throws` 规范。稍后, 将会讨论如何决定一个异常是被捕获, 还是被

抛出让其他的处理器进行处理。



警告：如果在子类中覆盖了超类的一个方法，子类方法中声明的已检查异常不能超过超类方法中声明的异常范围（也就是说，子类方法中抛出的异常范围更加小，或者根本不抛出任何异常）。特别需要说明的是，如果超类方法没有抛出任何已检查异常，子类也不能抛出任何已检查异常。例如，如果覆盖JComponent.paintComponent方法，由于超类中这个方法没有抛出任何异常，所以，自定义的paintComponent也不能抛出任何已检查异常。

如果类中的一个方法声明将会抛出一个异常，而这个异常是某个特定类的实例时，则这个方法就有可能抛出一个这个类的异常，或者这个类的任意一个子类的异常。例如，FileInputStream构造器声明将有可能抛出一个IOException异常，然而并不知道具体是哪一种IOException异常。它既可能是IOException异常，也可能是其子类的异常，例如，FileNotFoundException。



C++注释：Java中的throws说明符与C++中的throw说明符基本类似，但有一点重要的区别。在C++中，throw说明符在运行时执行，而不是在编译时执行。也就是说，C++编译器将不处理任何异常说明符。但是，如果函数抛出的异常没有出现在throw列表中，就会调用unexpected函数，这个函数的默认处理方式是终止程序的执行。

另外，在C++中，如果没有给出throw说明，函数可能会抛出任何异常。而在Java中，没有throws说明符的方法将不能抛出任何已检查异常。

11.1.3 如何抛出异常

假设在程序代码中发生了一些很糟糕的事情。一个名为readData的方法正在读取一个首部具有下列信息的文件：

```
Content-length: 1024
```

然而，读到733个字符之后文件就结束了。我们认为这是一种不正常的情况，希望抛出一个异常。

首先要决定应该抛出什么类型的异常。将上述异常归结为IOException是一种很好的选择。仔细地阅读Java API文档之后会发现：EOFException异常描述的是“在输入过程中，遇到了一个未预期的EOF后的信号”。这正是我们要抛出的异常。下面是抛出这个异常的语句：

```
throw new EOFException();
```

或者

```
EOFException e = new EOFException();  
throw e;
```

下面将这些代码放在一起：

```
String readData(Scanner in) throws EOFException  
{  
    ...  
    while (...)  
    {  
        if (!in.hasNext()) // EOF encountered
```

```

    {
        if (n < len)
            throw new EOFException();
        }
        . . .
    }
    return s;
}

```

EOFException类还有一个含有一个字符串型参数的构造器。这个构造器可以更加细致的描述异常出现的情况。

```

String gripe = "Content-length: " + len + ", Received: " + n;
throw new EOFException(gripe);

```

在前面已经看到，对于一个已经存在的异常类，将其抛出非常容易。在这种情况下：

- 1) 找到一个合适的异常类。
- 2) 创建这个类的一个对象。
- 3) 将对象抛出。

一旦方法抛出了异常，这个方法就不可能返回到调用者。也就是说，不必为返回的默认值或错误代码担忧。



C++注释：在C++与Java中，抛出异常的过程基本相同，只有一点微小的差别。在Java中，只能抛出Throwable子类的对象，而在C++中，却可以抛出任何类型的值。

11.1.4 创建异常类

在程序中，可能会遇到任何标准异常类都没有能够充分地描述清楚的问题。在这种情况下，创建自己的异常类就是一件顺理成章的事情了。我们需要做的只是定义一个派生于Exception的类，或者派生于Exception子类的类。例如，定义一个派生于IOException的类。习惯上，定义的类应该包含两个构造器，一个是默认的构造器；另一个是带有详细描述信息的构造器（超类Throwable的toString方法将会打印出这些详细信息，这在调试中非常有用）。

```

class FileFormatException extends IOException
{
    public FileFormatException() {}
    public FileFormatException(String gripe)
    {
        super(gripe);
    }
}

```

现在，就可以抛出自己定义的异常类型了。

```

String readData(BufferedReader in) throws FileFormatException
{
    . . .
    while (. . .)
    {
        if (ch == -1) // EOF encountered
        {
            if (n < len)

```

```
        throw new FileFormatException();
    }
    ...
}
return s;
}
```

API java.lang.Throwable 1.0

- `Throwable()`
构造一个新的`Throwable`对象，这个对象没有详细的描述信息。
- `Throwable(String message)`
构造一个新的`throwable`对象，这个对象带有特定的详细描述信息。习惯上，所有派生的异常类都支持一个默认的构造器和一个带有详细描述信息的构造器。
- `String getMessage()`
获得`Throwable`对象的详细描述信息。

11.2 捕获异常

到目前为止，已经知道如何抛出一个异常。这个过程十分容易。只要将其抛出就不用理睬了。当然，有些代码必须捕获异常。捕获异常需要进行周密的计划。

如果某个异常发生的时候没有在任何地方进行捕获，那程序就会终止执行，并在控制台上打印出异常信息，其中包括异常的类型和堆栈的内容。对于图形界面程序（`applet`和`application`应用程序），在捕获异常之后，也会打印出堆栈的信息，但程序将返回到用户界面的处理循环中（在调试基于图形界面的程序时，最好保证控制台窗口可见，并且没有被极小化）。

要想捕获一个异常，必须设置`try/catch`语句块。最简单的`try`语句块如下所示：

```
try
{
    code
    more code
    more code
}
catch (ExceptionType e)
{
    handler for this type
}
```

如果在`try`语句块中的任何代码抛出了一个在`catch`子句中说明的异常类，那么

- 1) 程序将跳过`try`语句块的其余代码。
- 2) 程序将执行`catch`子句中的处理器代码。

如果在`try`语句块中的代码没有抛出任何异常，那么程序将跳过`catch`子句。

如果方法中的任何代码抛出了一个在`catch`子句中没声明的异常类型，那么这个方法就会立刻退出（期待调用者为这种类型的异常设计了`catch`子句）。

为了演示捕获异常的处理过程，下面给出一个读取文本的典型程序代码：

```
public void read(String filename)
```



```
{
    try
    {
        InputStream in = new FileInputStream(filename);
        int b;
        while ((b = in.read()) != -1)
        {
            process input
        }
    }
    catch (IOException exception)
    {
        exception.printStackTrace();
    }
}
```

需要注意，try语句中的大多数代码都很容易理解：读取并处理文本行，直到遇到文件结束符为止。正如在Java API中看到的那样，read方法有可能抛出一个IOException异常。在这种情况下，将跳出整个while循环，进入catch子句，并输出堆栈情况。对于一个普通的程序来说，这样处理异常基本上合乎情理。还有其他的选择吗？

通常，最好的选择是什么也不做，而是将异常传递给调用者。如果read方法出现了错误，就让read方法的调用者去操心！如果采用这种处理方式，就必须声明这个方法可能会抛出一个IOException。

```
public void read(String filename) throws IOException
{
    InputStream in = new FileInputStream(filename);
    int b;
    while ((b = in.read()) != -1)
    {
        process input
    }
}
```

请记住，编译器严格地执行throws说明符。如果调用了一个抛出已检查异常的方法，就必须对它进行处理，或者将它传递出去。

哪种方法更好呢？通常，应该捕获那些知道如何处理的异常，而将那些不知道怎样处理的异常传递出去。如果想将异常传递出去，就必须在方法的首部添加一个throws说明符，以便告知调用者这个方法可能会抛出异常。

仔细阅读一下Java API文档，以便知道每个方法可能会抛出哪种异常，然后再决定是自己处理，还是添加到throws列表中。对于后一种情况，也不必犹豫。将异常直接交给能够胜任的处理器进行处理要比压制对它的处理更好。

同时请记住，这个规则也有一个例外。前面曾经提到过：如果编写一个覆盖超类的方法，而这个方法又没有抛出异常（例如，JComponent 中的paintComponent），那么这个方法就必须捕获方法代码中出现的每一个已检查异常。不允许在子类的throws说明符中出现超过超类方法所列出的异常类范围。



C++注释：在Java与C++中，捕获异常的方式基本相同。严格地说，下列代码

```
catch (Exception e) // Java
```

与

```
catch (Exception& e) // C++
```

是一样的。

在Java中，没有与C++中catch() 对应的东西。由于Java中的所有异常类都派生于一个公共的超类，所以，没有必要使用这种机制。

11.2.1 捕获多个异常

在一个try语句块中可以捕获多个异常类型，并对不同类型的异常做出不同的处理。可以按照下列方式为每个异常类型使用一个单独的catch子句：

```
try
{
    code that might throw exceptions
}
catch (MalformedURLException e1)
{
    emergency action for malformed URLs
}
catch (UnknownHostException e2)
{
    emergency action for unknown hosts
}
catch (IOException e3)
{
    emergency action for all other I/O problems
}
```

异常对象 (e1,e2,e3) 可能包含与异常本身有关的信息。要想获得对象的更多信息，可以试着使用

```
e3.getMessage()
```

得到详细的错误信息 (如果有的话)，或者使用

```
e3.getClass().getName()
```

得到异常对象的实际类型。

11.2.2 再次抛出异常与异常链

在catch子句中可以抛出一个异常，这样做的目的是改变异常的类型。如果开发了一个供其他程序员使用的子系统，那么，用于表示子系统故障的异常类型可能会产生多种解释。ServletException就是这样一个异常类型的例子。执行servlet的代码可能不想知道发生错误的细节原因，但希望明确地知道servlet是否有故障。

下面给出了捕获异常并将它再次抛出的基本方法：

```
try
{
    access the database
}
```

```
catch (SQLException e)
{
    throw new ServletException("database error: " + e.getMessage());
}
```

这里，`ServleException`用带有异常信息文本的构造器来构造。在Java SE 1.4中，可以有一种更好的处理方法，并且将原始异常设置为新异常的“诱饵”：

```
try
{
    access the database
}
catch (SQLException e)
{
    Throwable se = new ServletException("database error");
    se.initCause(e);
    throw se;
}
```

当捕获到异常时，就可以使用下面这条语句重新得到原始异常：

```
Throwable e = se.getCause();
```

强烈地建议使用这种包装技术。这样可以让用户抛出子系统中的高级异常，而不会丢失原始异常的细节。



提示：如果在一个方法中发生了一个已检查异常，而不允许抛出它，那么包装技术就十分有用。我们可以捕获这个已检查异常，并将它包装成一个运行时异常。



注释：有些异常类，例如，`ClassNotFoundException`、`InvocationTargetException`和`RuntimeException`，拥有它们自己的异常链方案。在Java SE1.4中，这些已经引入“诱饵”机制。然而，仍然可以利用原来的方式，或者调用`getCause`得到异常链。

11.2.3 Finally子句

当代码抛出一个异常时，就会终止方法中剩余代码的处理，并退出这个方法的执行。如果方法获得了一些本地资源，并且只有这个方法自己知道，又如果这些资源在退出方法之前必须被回收，那么就会产生资源回收问题。一种解决方案是捕获并重新抛出所有的异常。但是，这种解决方案比较乏味，这是因为需要在两个地方清除所分配的资源。一个在正常的代码中；另一个在异常代码中。

Java有一种更好的解决方案，这就是`finally`子句。下面将介绍如何恰当地释放`Graphics`对象。如果使用Java编写数据库程序，就需要使用这个技术关闭与数据库的连接。在卷II的第4章中可以看到更加详细的介绍。当发生异常时，恰当地关闭所有数据库的连接是非常重要的。

不管是否有异常被捕获，`finally`子句中的代码都被执行。在下面的示例中，程序将释放所有环境中的图形设备文本。

```
Graphics g = image.getGraphics();
try
{
    // 1
```

```
    code that might throw exceptions
    // 2
}
catch (IOException e)
{
    // 3
    show error dialog
    // 4
}
finally
{
    // 5
    g.dispose();
}
// 6
```

在上面这段代码中，有下列三种情况会执行finally子句：

1) 代码没有抛出异常。在这种情况下，程序首先执行try语句块中的全部代码，然后执行finally子句中的代码。随后，继续执行try语句块之后的第一条语句。也就是说，执行标注的1、2、5、6处。

2) 抛出一个在catch子句中捕获的异常。在上面的示例中就是IOException异常。在这种情况下，程序将执行try语句块中的所有代码，直到发生异常为止。此时，将跳过try语句块中的剩余代码，转去执行与该异常匹配的catch子句中的代码，最后执行finally子句中的代码。

如果catch子句没有抛出异常，程序将执行try语句块之后的第一条语句。在这里，执行标注1、3、4、5、6处的语句。

如果catch子句抛出了一个异常，异常将被抛回这个方法的调用者。在这里，执行标注1、3、5处的语句。

3) 代码抛出了一个异常，但这个异常不是由catch子句捕获的。在这种情况下，程序将执行try语句块中的所有语句，直到有异常被抛出为止。此时，将跳过try语句块中的剩余代码，然后执行finally子句中的语句，并将异常抛给这个方法的调用者。在这里，执行标注1、5处的语句。

try语句可以只有finally子句，而没有catch子句。例如，下面这条try语句：

```
InputStream in = ...;
try
{
    code that might throw exceptions
}
finally
{
    in.close();
}
```

无论在try语句块中是否遇到异常，finally子句中的in.close() 语句都会被执行。当然，如果真的遇到一个异常，这个异常将会被重新抛出，并且必须由另一个catch子句捕获。

事实上，我们认为在需要关闭资源时，用这种方式使用finally子句是一种不错的选择。下面的提示将给予具体的解释。



提示：这里，强烈建议独立使用try/catch和try/finally语句块。这样可以提高代码的清晰度。例如，

```
InputStream in = ...;
try
{
    try
    {
        code that might throw exceptions
    }
    finally
    {
        in.close();
    }
}
catch (IOException e)
{
    show error dialog
}
```

内层的try语句块只有一个职责，就是确保关闭输入流。外层的try语句块也只有一个职责，就是确保报告出现的错误。这种设计方式不仅清楚，而且还具有一个功能，就是将会报告finally子句中出现的错误。



警告：当finally子句包含return语句时，将会出现一种意想不到的结果。假设利用return语句从try语句块中退出。在方法返回前，finally子句的内容将被执行。如果finally子句中也有一个return语句，这个返回值将会覆盖原始的返回值。请看一个例子：

```
public static int f(int n)
{
    try
    {
        int r = n * n;
        return r;
    }
    finally
    {
        if (n == 2) return 0;
    }
}
```

如果调用f(2)，那么try语句块的计算结果为r = 4，并执行return语句。然而，在方法真正返回前，还要执行finally子句。finally子句将使得方法返回0，这个返回值覆盖了原始的返回值4。

有时候，finally子句也会带来麻烦。例如，回收资源的方法也有可能抛出异常。一个比较典型的例子是关闭流（有关流的详细内容将在第12章介绍）。假设希望能够确保在流处理代码中遇到异常时将流关闭。

```
InputStream in = ...;
try
{
    code that might throw exceptions
}
```

```
}  
finally  
{  
    in.close();  
}
```

现在，假设在try语句块中的代码抛出了一些非IOException的异常，这些异常只有这个方法的调用者才能够给予处理。执行finally语句块，并调用close方法。而close方法本身也有可能抛出IOException异常。当出现这种情况时，原始的异常将会丢失，转而抛出IOException异常。这并不是异常处理机制所希望的结果。

解决这个问题的最好方法是在用户调用的finally子句中执行清除操作时不要抛出异常，例如，dispose、close等等，但是InputStream类的设计者并没有这样做。



C++注释：在异常处理方面，C++与Java存在重要区别。Java没有析构器，因此，没有C++的自动回收功能。这就意味着Java程序员必须手工地将回收资源的代码写入finally子句中。当然，由于Java内置了垃圾回收机制，所以，只有极少数的资源需要人工回收。

11.2.4 分析堆栈跟踪元素

堆栈跟踪（stack trace）是一个方法调用过程的列表，它包含了程序执行过程中方法调用的特定位置。前面已经看到过这种列表，当Java程序正常终止，而没有捕获异常时，这个列表就会显示出来。

在Java SE 1.4以前的版本中，可以调用Throwable类的printStackTrace方法访问堆栈跟踪的文本描述信息。现在，可以调用getStackTrace方法获得一个StackTraceElement对象的数组，并在程序中对它进行分析。例如，

```
Throwable t = new Throwable();  
StackTraceElement[] frames = t.getStackTrace();  
for (StackTraceElement frame : frames)  
    analyze frame
```

StackTraceElement类含有能够获得文件名和当前执行的代码行号的方法，同时，还含有能够获得类名和方法名的方法。toString方法将产生一个格式化的字符串，其中包含所获得的信息。

在Java SE 5.0中，增加了一个静态的Thread.getAllStackTraces方法，它可以产生所有线程的堆栈跟踪。下面给出使用这个方法的具体方式：

```
Map<Thread, StackTraceElement[]> map = Thread.getAllStackTraces();  
for (Thread t : map.keySet())  
{  
    StackTraceElement[] frames = map.get(t);  
    analyze frames  
}
```

有关Map接口与线程的更加详细的信息请参看第13章和第14章。

例11-1打印了递归阶乘的堆栈情况。例如，如果计算factorial(3)，将会打印下列内容：

```
factorial(3):  
StackTraceTest.factorial(StackTraceTest.java:18)  
StackTraceTest.main(StackTraceTest.java:34)  
factorial(2):
```

```

StackTraceTest.factorial(StackTraceTest.java:18)
StackTraceTest.factorial(StackTraceTest.java:24)
StackTraceTest.main(StackTraceTest.java:34)
factorial(1):
StackTraceTest.factorial(StackTraceTest.java:18)
StackTraceTest.factorial(StackTraceTest.java:24)
StackTraceTest.factorial(StackTraceTest.java:24)
StackTraceTest.main(StackTraceTest.java:34)
return 1
return 2
return 6

```

例 11-1 StackTraceTest.java

```

1. import java.util.*;
2.
3. /**
4.  * A program that displays a trace feature of a recursive method call.
5.  * @version 1.01 2004-05-10
6.  * @author Cay Horstmann
7.  */
8. public class StackTraceTest
9. {
10.     /**
11.      * Computes the factorial of a number
12.      * @param n a nonnegative integer
13.      * @return n! = 1 * 2 * . . . * n
14.      */
15.     public static int factorial(int n)
16.     {
17.         System.out.println("factorial(" + n + "):");
18.         Throwable t = new Throwable();
19.         StackTraceElement[] frames = t.getStackTrace();
20.         for (StackTraceElement f : frames)
21.             System.out.println(f);
22.         int r;
23.         if (n <= 1) r = 1;
24.         else r = n * factorial(n - 1);
25.         System.out.println("return " + r);
26.         return r;
27.     }
28.
29.     public static void main(String[] args)
30.     {
31.         Scanner in = new Scanner(System.in);
32.         System.out.print("Enter n: ");
33.         int n = in.nextInt();
34.         factorial(n);
35.     }
36. }

```

API java.lang.Throwable 1.0

- Throwable(Throwable cause) 1.4
- Throwable(String message, Throwable cause) 1.4

用给定的“诱饵”构造一个Throwable对象。

- Throwable initCause(Throwable cause) 1.4

将这个对象设置为“诱饵”。如果这个对象已经被设置为“诱饵”，则抛出一个异常。返回this引用。

- Throwable getCause() 1.4

获得设置为这个对象的“诱饵”的异常对象。如果没有设置“诱饵”，则返回null。

- StackTraceElement[] getStackTrace() 1.4

获得构造这个对象时调用堆栈的跟踪。

API java.lang.Exception 1.0

- Exception(Throwable cause) 1.4
- Exception(String message, Throwable cause)

用给定的“诱饵”构造一个异常对象。

API java.lang.RuntimeException 1.0

- RuntimeException(Throwable cause) 1.4
- RuntimeException(String message, Throwable cause) 1.4

用给定的“诱饵”构造一个RuntimeException对象。

API java.lang.StackTraceElement 1.4

- String getFileName()
返回这个元素运行时对应的源文件名。如果这个信息不存在，则返回null。
- int getLineNumber()
返回这个元素运行时对应的源文件行数。如果这个信息不存在，则返回-1。
- String getClassName()
返回这个元素运行时对应的类的全名。
- String getMethodName()
返回这个元素运行时对应的方法名。构造器名是<int>；静态初始化器名是<clinit>。这里无法区分同名的重载方法。
- boolean isNativeMethod()
如果这个元素运行时在一个本地方法中，则返回true。
- String toString()
如果存在的话，返回一个包含类名、方法名、文件名和行数的格式化字符串。

11.3 使用异常机制的建议

目前，存在着大量有关如何恰当地使用异常机制的争论。有些程序员认为所有的已检查异常都很令人厌恶；还有一些程序员认为能够抛出的异常量不够。我们认为异常机制（甚至是已

检查异常)有其用武之地。下面给出使用异常机制的几点建议。

1. 异常处理不能代替简单的测试

作为一个示例,在这里编写了一段应用内置stack类的代码,试着上百万次地对一个空栈进行退栈操作。在实施退栈操作之前,首先要查看栈是否为空。

```
if (!s.empty()) s.pop();
```

接下来,强行进行退栈操作。然后,捕获EmptyStackException异常来告知我们不能这样做。

```
try()
{
    s.pop();
}
catch (EmptyStackException e)
{
}
```

在测试的机器上,得到了表11-1中的时间数据。

可以看出,与执行简单的测试相比,捕获异常所花费的时间大大超过了前者,因此使用异常的基本规则是:只在异常情况下使用异常机制。

表11-1 时间数据

Test	Throw/Catch
646 milliseconds	21,739 milliseconds

2. 不要过分地细化异常

很多程序员习惯将每一条语句都分装在一个独立的try语句块中。

```
OutputStream out;
Stack s;

for (i = 0; i < 100; i++)
{
    try
    {
        n = s.pop();
    }
    catch (EmptyStackException s)
    {
        // stack was empty
    }
    try
    {
        out.writeInt(n);
    }
    catch (IOException e)
    {
        // problem writing to file
    }
}
```

这种编程方式将导致代码量的急剧膨胀。首先看一下这段代码所完成的任务。在这里,希望从栈中弹出100个数值,然后将它们存入一个文件中。如果栈是空的,则不会变成非空状态;如果文件出现错误,则也很难给予排除。出现上述问题后,这种编程方式无能为力。因此,有必要将整个任务包装在一个try语句块中,这样,当任何一个操作出现问题时,整个任务都可

以被取消。

```
try
{
    for (i = 0; i < 100; i++)
    {
        n = s.pop();
        out.writeInt(n);
    }
}
catch (IOException e)
{
    // problem writing to file
}
catch (EmptyStackException s)
{
    // stack was empty
}
```

这段代码看起来清晰多了。这样也满足了异常处理机制的其中一个目标，将正常处理与错误处理分开。

3. 利用异常层次结构

不要只抛出`RuntimeException`异常。应该寻找更加适当的子类或创建自己的异常类。

不要只捕获`Throwable`异常，否则，会使程序代码更难读、更难维护。

研究一下已检查异常与未检查异常的区别。已检查异常本来就很庞大，这里不会抛出由逻辑错误造成的异常。例如，反射库发生了错误，而调用者却经常需要捕获那些早已知道不可能发生的异常。

将一种异常转换成另一种更加适合的异常时不要犹豫。例如，在解析某个文件中的一个整数时，捕获`NumberFormatException`异常，然后将它转换成`IOException` 或`MySubsystemException`的子类。

4. 不要压制异常

在Java中，存在着强烈的关闭异常的倾向。如果编写了一个调用一个方法的方法，而这个方法有可能100年才抛出一个异常，那么，编译器会因为没有将这个异常列在`throws`表中产生抱怨。而没有将这个异常列在`throws`表中主要出于编译器将会对所有调用这个方法的方法进行异常处理的考虑。因此，应该将这个异常关闭：

```
public Image loadImage(String s)
{
    try
    {
        code that threatens to throw checked exceptions
    }
    catch (Exception e)
    {} // so there
}
```

现在，这段代码就可以通过编译了。除非发生异常，否则它将可以正常地运行。即使发生了异常也会被忽略。如果认为异常非常重要，就应该对它们进行处理。

5. 在检测错误时，“苛刻”要比放任更好

当检测到错误的时候，有些程序员担心抛出异常。在用无效的参数调用一个方法时，返回一个虚拟的数值，还是抛出一个异常，哪种处理方式更好？例如，当栈空时，`Stack.pop`是返回一个`null`，还是抛出一个异常？我们认为：在出错的地方抛出一个`EmptyStackException`异常要比在后面抛出一个`NullPointerException`异常更好。

6. 不要羞于传递异常

很多程序员都感觉应该捕获抛出的全部异常。如果调用了一个抛出异常的方法，例如，`FileInputStream` 构造器或`readLine`方法，这些方法就会本能地捕获这些可能产生的异常。其实，传递异常要比捕获这些异常更好：

```
public void readStuff(String filename) throws IOException // not a sign of shame!
{
    InputStream in = new FileInputStream(filename);
    ...
}
```

让高层次的方法通告用户发生了错误，或者放弃不成功的命令更加适宜。



注释：规则5、8可以归纳为“早抛出，晚捕获”。

11.4 断言

在一个具有自我保护能力的程序中，断言是一个常用的习语。假设确信某个属性符合要求，并且代码的执行依赖于这个属性。例如，需要计算

```
double y = Math.sqrt(x);
```

我们确信，这里的`x`是一个非负数值。原因是：`x`是另外一个计算的结果，而这个结果不可能是负值；或者`x`是一个方法的参数，而这个方法要求它的调用者只能提供一个正整数。然而，还是希望进行检查，以避免让错误的数值参与计算操作。当然，也可以抛出一个异常：

```
if (x < 0) throw new IllegalArgumentException("x < 0");
```

但是这段代码会一直保留在程序中，即使测试完毕也不会自动地删除。如果在程序中含有大量的这种检查，程序运行起来会相当慢。

断言机制允许在测试期间向代码中插入一些检查语句。当代码发布时，这些插入的检测语句将会被自动地移走。

在Java SE 1.4中，Java语言引入了关键字`assert`。这个关键字有两种形式：

```
assert 条件；
```

和

```
assert 条件：表达式；
```

这两种形式都会对条件进行检测，如果结果为`false`，则抛出一个`AssertionError`异常。在第二种形式中，表达式将被传入`AssertionError`的构造器，并转换成一个消息字符串。



注释：“表达式”部分的唯一目的是产生一个消息字符串。AssertionError对象并不存储表达式的值，因此，不可能在以后得到它。正如JDK文档所描述的那样：如果使用表达式的值，就会鼓励程序员试图从断言中恢复程序的运行，这不符合断言机制的初衷。

要想断言 x 是一个非负数值，只需要简单地使用下面这条语句

```
assert x >= 0;
```

或者将 x 的实际值传递给AssertionError对象，从而可以在后面显示出来。

```
assert x >= 0 : x;
```



C++注释：C语言中的assert宏将断言中的条件转换成一个字符串。当断言失败时，这个字符串将会被打印出来。例如，若assert($x \geq 0$)失败，那么将打印出失败条件“ $x \geq 0$ ”。在Java中，条件并不会自动地成为错误报告中的一部分。如果希望看到这个条件，就必须将它以字符串的形式传递给AssertionError对象：assert $x \geq 0$: “ $x \geq 0$ ”。

11.4.1 启用和禁用断言

在默认情况下，断言被禁用。可以在运行程序时用-enableassertions或-ea选项启用它：

```
java -enableassertions MyApp
```

需要注意：在启用或禁用断言时不必重新编译程序。启用或禁用断言是类加载器（class loader）的功能。当断言被禁用时，类加载器将跳过断言代码，因此，不会降低程序运行的速度。

也可以在某个类或某个包中使用断言。例如，

```
java -ea:MyClass -ea:com.mycompany.mylib... MyApp
```

这条命令将开启MyClass类以及在com.mycompany.mylib包和它的子包中的所有类的断言。选项-ea将开启默认包中的所有类的断言。

也可以用选项-disableassertions或-da禁用某个特定类和包的断言：

```
java -ea:... -da:MyClass MyApp
```

有些类不是由类加载器加载，而是直接由虚拟机加载。可以使用这些开关有选择地启用或禁用那些类中的断言。然而，启用和禁用所有断言的-ea和-da开关不能应用到那些没有类加载器的“系统类”上。对于这些系统类来说，需要使用-enablesystemassertions/-esa开关启用断言。

在程序中也可以控制类加载器的断言状态。有关这方面的内容请参看本节末尾的API注解。

11.4.2 使用断言的建议

在Java语言中，给出了三种处理系统错误的机制：

- 抛出一个异常
- 日志
- 使用断言

什么时候应该选择使用断言呢？请记住下面几点：

- 断言失败是致命的、不可恢复的错误。
- 断言检查只用于开发和测试阶段（这种做法有时候被戏称为“在靠近海岸时穿上救生衣，但在海中央时就把救生衣抛入水中吧”）。

因此，不应该使用断言向程序的其他部分通告发生了可恢复性的错误，或者，不应该作为程序向用户通告问题的手段。断言只应该用于在测试阶段确定程序内部的错误位置。

下面看一个十分常见的例子：检查方法的参数。是否应该使用断言来检查非法的下标值或 null 引用呢？要想回答这个问题，首先阅读一下这个方法的文档。假设实现一个排序方法。

```
/**
 * Sorts the specified range of the specified array into ascending numerical order.
 * The range to be sorted extends from fromIndex, inclusive, to toIndex, exclusive.
 * @param a the array to be sorted.
 * @param fromIndex the index of the first element (inclusive) to be sorted.
 * @param toIndex the index of the last element (exclusive) to be sorted.
 * @throws IllegalArgumentException if fromIndex > toIndex
 * @throws ArrayIndexOutOfBoundsException if fromIndex < 0 or toIndex > a.length
 */
static void sort(int[] a, int fromIndex, int toIndex)
```

文档说明，如果方法中使用了错误的下标值，那么就会抛出一个异常。这是方法与调用者之间约定的处理行为。如果实现这个方法，那就必须要遵守这个约定，并抛出表示下标值有误的异常。因此，这里使用断言不太适宜。

是否应该断言 `a` 而不是 `null` 吗？这也不太适宜。当 `a` 是 `null` 时，这个方法的文档没有指出应该采取什么行动。在这种情况下，调用者可以认为这个方法将会成功地返回，而不会抛出一个断言错误。

然而，假设对这个方法的约定做一点微小的改动：

```
@param a the array to be sorted. (Must not be null)
```

现在，这个方法的调用者就必须注意：不允许用 `null` 数组调用这个方法，并在这个方法的开头使用断言

```
assert a != null;
```

计算机科学家将这种约定称为前提条件 (Precondition)。最初的方法对参数没有前提条件，即承诺在任何条件下都能够给予正确的执行。修订后的方法有一个前提条件，即 `a` 非空。如果调用者在调用这个方法时没有提供满足这个前提条件的参数，所有的断言都会失败，并且这个方法可以执行它想做的任何操作。事实上，由于可以使用断言，当方法被非法调用时，将会出现难以预料的结果。有时候会抛出一个断言错误，有时候会产生一个 `null` 指针异常，这完全取决于类加载器的配置。

11.4.3 为文档使用断言

很多程序员使用注释说明假设条件。看一下 <http://java.sun.com/javase/6/docs/technotes/guides/language/assert.html> 上的一个示例：

```
if (i % 3 == 0)
    . . .
else if (i % 3 == 1)
    . . .
else // (i % 3 == 2)
    . . .
```

在这个示例中，使用断言会更好一些。

```
if (i % 3 == 0)
    . . .
else if (i % 3 == 1)
    . . .
else
{
    assert i % 3 == 2;
    . . .
}
```

当然，如果再仔细地考虑一下这个问题会发现一个更有意思的内容。 $i \% 3$ 会产生什么结果？如果 i 是正值，那余数肯定是0、1或2。如果 i 是负值，则余数则可以是-1和-2。然而，实际上都认为 i 是非负值，因此，最好在 if 语句之前使用下列断言：

```
assert i >= 0;
```

无论如何，这个示例说明了程序员如何使用断言来进行自我检查。前面已经知道，断言是一种测试和调试阶段所使用的战术性工具；而日志记录是一种在程序的整个生命周期都可以使用的策略性工具。稍后将介绍日志的相关知识。



java.lang.ClassLoader 1.0

- void setDefaultAssertionStatus(boolean b) 1.4
对于通过类加载器加载的所有类来说，如果没有显式地说明类或包的断言状态，就启用或禁用断言。
- void setClassAssertionStatus(String className, boolean b) 1.4
对于给定的类和它的内部类，启用或禁用断言。
- void setPackageAssertionStatus(String packageName, boolean b) 1.4
对于给定包和其子包中的所有类，启用或禁用断言。
- void clearAssertionStatus() 1.4
移去所有类和包的显式断言状态设置，并禁用所有通过这个类加载器加载的类的断言。

11.5 记录日志

每个Java程序员都很熟悉在有问题的代码中插入一些调用`System.out.println`方法的语句来帮助观察程序运行的操作过程。当然，一旦发现问题的根源，就要将这些语句从代码中删去。如果接下来又出现了问题，就需要再插入几个调用`System.out.println`方法的语句。记录日志API就是为了解决这个问题而设计的。下面先讨论一下这些API的优点。

- 可以很容易地取消全部日志记录，或者仅仅取消某个级别的日志，而且打开和关闭这个操作也很容易。
- 可以很简单地禁止日志记录的输出，因此，将这些日志代码留在程序中所付出的代价很小。
- 日志记录可以被定向到不同的处理器，用于在控制台中显示，用于存储在文件中等等。
- 日志记录器和处理器都可以对记录进行过滤。过滤器可以根据过滤器制定的标准丢

弃那些无用的记录项。

- 日志记录可以采用不同的方式格式化，例如，纯文本或XML。
- 应用程序可以使用多个日志记录器，它们使用类似包名的这种具有层次结构的名称，例如，com.mycompany.myapp。
- 在默认情况下，日志系统的配置由配置文件控制。如果需要的话，应用程序可以替换这个配置。

11.5.1 基本日志

下面从一个最简单的例子开始。日志系统管理着一个名为Logger.global的默认日志记录器，可以用System.out替换它，并通过调用info方法记录日志信息：

```
Logger.global.info("File->Open menu item selected");
```

在默认情况下，这条记录将会显示出如下所示的内容：

```
May 10, 2004 10:12:15 PM LoggingImageViewer fileOpen  
INFO: File->Open menu item selected
```

注意：自动包含了时间、调用的类名和方法名。但是，如果在恰当的地方（例如，main开始）调用

```
Logger.global.setLevel(Level.OFF);
```

将会取消所有的日志。

11.5.2 高级日志

从前面已经看到“虚拟日志”，下面继续看一下企业级（industrial-strength）日志。在一个专业的应用程序中，不要将所有的日志都记录到一个全局日志记录器中，而是可以自定义日志记录器。

当第一次请求一个具有给定名称的日志记录器时，就会创建这个记录器。

```
Logger myLogger = Logger.getLogger("com.mycompany.myapp");
```

如果后面使用同一个名称调用日志记录器就会产生相同的日志记录器对象。

与包名类似，日志记录器名也具有层次结构。事实上，与包名相比，日志记录器的层次性更强。对于包来说，一个包的名称与其父包的名称之间没有语义关系，但是日志记录器的父与子之间将共享某些属性。例如，如果对com.mycompany日志记录器设置了日志级别，它的子记录器也会继承这个级别。

通常，有以下7个日志记录器级别：

- SEVERE
- WARNING
- INFO
- CONFIG
- FINE
- FINER

- FINEST

在默认情况下，只记录前三个级别。也可以设置其他的级别。例如，

```
logger.setLevel(Level.FINE);
```

现在，FINE和更高级别的记录都可以记录下来。

另外，还可以使用Level.ALL开启所有级别的记录，或者使用Level.OFF关闭所有级别的记录。

对于所有的级别有下面几种记录方法：

```
logger.warning(message);  
logger.fine(message);
```

同时，还可以使用log方法指定级别，例如，

```
logger.log(Level.FINE, message);
```



提示：默认的日志配置记录了INFO或更高级别的所有记录，因此，应该使用CONFIG、FINE、FINER和FINEST级别来记录那些有助于诊断，但对于程序员又没有太大意义的调试信息。



警告：如果将记录级别设计为INFO或者更低，则需要修改日志处理器的配置。默认的日志处理器不会处理低于INFO级别的信息。有关更加详细的内容请参看下一节。

默认的日志记录将显示包含日志调用的类名和方法名，如同堆栈所显示的那样。但是，如果虚拟机对执行过程进行了优化，就得不到准确的调用信息。此时，可以调用logp方法获得调用类和方法的确切位置，这个方法的签名为：

```
void logp(Level l, String className, String methodName, String message)
```

下面有一些用来跟踪执行流程的方法：

```
void entering(String className, String methodName)  
void entering(String className, String methodName, Object param)  
void entering(String className, String methodName, Object[] params)  
void exiting(String className, String methodName)  
void exiting(String className, String methodName, Object result)
```

例如：

```
int read(String file, String pattern)  
{  
    logger.entering("com.mycompany.mylib.Reader", "read",  
        new Object[] { file, pattern });  
    . . .  
    logger.exiting("com.mycompany.mylib.Reader", "read", count);  
    return count;  
}
```

这些调用将生成FINER级别和以字符串“ENTRY”和“RETURN”开始的日志记录。



注释：在未来，带Object[]参数的日志记录方法可能会被重写，以便支持变量参数列表（“varargs”）。此后就可以用logger.entering（“com.mycompany.mylib.Reader”，“read”，file, pattern）格式调用这个方法了。

记录日志的常见用途是记录那些不可预料的异常。可以使用下面两个方法提供日志记录中

包含的异常描述内容。

```
void throwing(String className, String methodName, Throwable t)
void log(Level l, String message, Throwable t)
```

典型的用法是：

```
if (...)
{
    IOException exception = new IOException("...");
    logger.throwing("com.mycompany.mylib.Reader", "read", exception);
    throw exception;
}
```

还有

```
try
{
    ...
}
catch (IOException e)
{
    Logger.getLogger("com.mycompany.myapp").log(Level.WARNING, "Reading image", e);
}
```

调用throwing可以记录一条FINER级别的记录和一条以THROW开始的信息。

11.5.3 修改日志管理器配置

可以通过编辑配置文件来修改日志系统的各种属性。在默认情况下，配置文件存在于：

```
jre/lib/logging.properties
```

要想使用另一个配置文件，就要将java.util.logging.config.file特性设置为配置文件的存储位置，并用下列命令启动应用程序：

```
java -Djava.util.logging.config.file=configFile MainClass
```



警告：在main方法中调用 System.setProperty(“java.util.logging.config.file”，file)没有任何影响，其原因是日志管理器在VM启动过程中被初始化，它会在main之前执行。

要想修改默认的日志记录级别，就需要编辑配置文件，并修改以下命令行

```
.level=INFO
```

可以通过添加以下内容来指定自己的日志记录级别

```
com.mycompany.myapp.level=FINE
```


也就是说，在日志记录名后面添加后缀.level。


在稍后可以看到，日志记录并不将消息发送到控制台上，这是处理器的任务。另外，处理器也有级别。要想在控制台上看到FINE级别的消息，就需要进行下列设置

```
java.util.logging.ConsoleHandler.level=FINE
```



警告：在日志管理器配置的属性设置不是系统属性，因此，用-Dcom.mycompany.myapp.level= FINE启动应用程序不会对日志记录产生任何影响。

 警告：截止到Java SE 6，Logmanager类的API文档主张通过Preferences API设置java.util.logging.config.class和java.util.logging.config.file属性。这是不正确的，有关信息请参看http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4691587。

 注释：日志属性文件由java.util.logging.LogManager类处理。可以通过将java.util.logging.manager系统属性设置为某个子类的名字来指定一个不同的日志管理器。另外，在保存标准日志管理器的同时，还可以从日志属性文件跳过初始化。还有一种方式是将java.util.logging.config.class系统属性设置为某个类名，该类再通过其他方式设定日志管理器属性。有关LogManager类的详细内容请参看API文档。

在运行的程序中，使用jconsole程序也可以改变日志记录的级别。有关信息请参看<http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html#LoggingControl>。

11.5.4 本地化

我们可能希望将日志消息本地化，以便让全球的用户都可以阅读它。应用程序的国际化问题将在卷II的第5章中讨论。下面简要地说明一下在本地化日志消息时需要牢记的一些要点。

本地化的应用程序包含资源包（resource bundle）中的本地说明信息。资源包由各个地区（例如，美国、德国）的映射集合组成。例如，某个资源包可能将字符串“readingFile”映射成英文的“Reading file”或者德文的“Achtung! Datei wird eingelesen”。

一个程序可以包含多个资源包，一个用于菜单；其他用于日志消息。每个资源包都有一个名字（例如，com.mycompany.logmessages）。要想将映射添加到一个资源包中，需要为每个地区创建一个文件。英文消息映射位于com/mycompany/logmessages_en.properties文件中；德文消息映射位于com/mycompany/logmessages_de.properties文件中。（en、de是语言编码）。可以将这些文件与应用程序的类文件放在一起，以便ResourceBundle类自动地对它们进行定位。这些文件都是纯文本文件，其组成项如下所示：

```
readingFile=Achtung! Datei wird eingelesen
renamingFile=Datei wird umbenannt
...
```

在请求日志记录器时，可以指定一个资源包：

```
Logger logger = Logger.getLogger(loggerName, "com.mycompany.logmessages");
```

然后，为日志消息指定资源包的关键字，而不是实际的日志消息字符串。

```
logger.info("readingFile");
```

通常需要在本地化的消息中增加一些参数，因此，消息应该包括占位符{0}、{1}等等。例如，要想在日志消息中包含文件名，就应该用下列方式包括占位符：

```
Reading file {0}.
Achtung! Datei {0} wird eingelesen.
```

然后，通过调用下面的一个方法向占位符传递具体的值：

```
logger.log(Level.INFO, "readingFile", fileName);
logger.log(Level.INFO, "renamingFile", new Object[] { oldName, newName });
```

11.5.5 处理器

在默认情况下，日志记录器将记录发送到ConsoleHandler中，并由它输出到System.err 流中。特别是，日志记录器还会将记录发送到父处理器中，而最终的处理器（命名为“ ”）有一个ConsoleHandler。

与日志记录器一样，处理器也有日志记录级别。对于一个要被记录的日志记录，它的日志记录级别必须高于日志记录器和处理器的阈值。日志管理器配置文件设置的默认控制台处理器的日志记录级别为

```
java.util.logging.ConsoleHandler.level=INFO
```

要想记录FINE级别的日志，就必须修改配置文件中的默认日志记录级别和处理器级别。另外，还可以绕过配置文件，安装自己的处理器。

```
Logger logger = Logger.getLogger("com.mycompany.myapp");
logger.setLevel(Level.FINE);
logger.setUseParentHandlers(false);
Handler handler = new ConsoleHandler();
handler.setLevel(Level.FINE);
logger.addHandler(handler);
```

在默认情况下，日志记录器将记录发送到自己的处理器和父处理器。我们的日志记录器是原始日志记录器（命名为“ ”）的子类，而原始日志记录器将会把所有等于或高于INFO级别的记录发送到控制台。然而，我们并不想两次看到这些记录。鉴于这个原因，应该将useParentHandlers属性设置为false。

要想将日志记录发送到其他地方，就要添加其他的处理器。日志API为此提供了两个很有用的处理器，一个是FileHandler；另一个是SocketHandler。SocketHandler将记录发送到特定的主机和端口。而更令人感兴趣的是FileHandler，它可以收集文件中的记录。

可以像下面这样直接将记录发送到默认文件的处理器：

```
FileHandler handler = new FileHandler();
logger.addHandler(handler);
```

这些记录被发送到用户主目录的javan.log文件中，n是文件名的惟一编号。如果用户系统没有主目录（例如，在Windows95/98/Me），文件就存储在C:\Window这样的默认位置上。在默认情况下，记录被格式化为XML。下面是一个典型的日志记录的形式：

```
<record>
  <date>2002-02-04T07:45:15</date>
  <millis>1012837515710</millis>
  <sequence>1</sequence>
  <logger>com.mycompany.myapp</logger>
  <level>INFO</level>
  <class>com.mycompany.mylib.Reader</class>
  <method>read</method>
  <thread>10</thread>
  <message>Reading file corejava.gif</message>
</record>
```

可以通过设置日志管理器配置文件中的不同参数（请参看表11-2），或者利用其他的构造

器（请参看本节后面给出的API注释）来修改文件处理器的默认行为。

也有可能不想使用默认的日志记录文件名，因此，应该使用另一种模式，例如，`%h/myapp.log`（有关模式变量的解释请参看表11-3）。

如果多个应用程序（或者同一个应用程序的多个拷贝）使用同一个日志文件，就应该开启append标志。另外，应该在文件名模式中使用`%u`，以便每个应用程序创建日志的惟一拷贝。

开启文件循环功能也是一个不错的主意。日志文件以`myapp.log.0`，`myapp.log.1`，`myapp.log.2`，这种循环序列的形式出现。只要文件超出了大小限制，最旧的文件就会被删除，其他的文件将重新命名，同时创建一个新文件，其编号为0。

表11-2 文件处理器配置参数

配置属性	描 述	默 认
<code>java.util.logging.FileHandler.level</code>	处理器级别	<code>Level.ALL</code>
<code>java.util.logging.FileHandler.append</code>	控制处理器应该追加到一个已经存在的文件尾部；还是应该为每个运行的程序打开一个新文件	<code>false</code>
<code>java.util.logging.FileHandler.limit</code>	在打开另一个文件之前允许写入一个文件的近似最大字节数（0表示无限制）	在FileHandler类中为0（表示无限制）；在默认的日志管理器配置文件中为50000。
<code>java.util.logging.FileHandler.pattern</code>	日志文件名的模式。参看表11-3中的模式变量	<code>%h/java%u.log</code>
<code>java.util.logging.FileHandler.count</code>	在循环序列中的日志记录数量	1（不循环）
<code>java.util.logging.FileHandler.filter</code>	使用的过滤器类	没有使用过滤器
<code>java.util.logging.FileHandler.encoding</code>	使用的字符编码	平台的编码
<code>java.util.logging.FileHandler.formatter</code>	记录格式器	<code>java.util.logging.XMLFormatter</code>



提示：很多程序员将日志记录作为辅助文档提供给技术支持员工。如果程序的行为有误，用户就可以返回查看日志文件以找到错误的原因。在这种情况下，应该开启“append”标志，或使用循环日志，也可以两个功能同时使用。

表11-3 日志记录文件模式变量

变 量	描 述
<code>%h</code>	系统属性 <code>user.home</code> 的值
<code>%t</code>	系统临时目录
<code>%u</code>	用于解决冲突的惟一性编号
<code>%g</code>	为循环日志记录生成的数值。（当使用循环功能且模式不包括 <code>%g</code> 时，使用后缀 <code>%g</code> ）
<code>%%</code>	%字符

还可以通过扩展Handler类或StreamHandler类自定义处理器。在本节结尾的示例程序中就定义了这样一个处理器。这个处理器将在窗口中显示日志记录（如图11-2所示）。



图11-2 在窗口中显示记录的日志处理器

这个处理器扩展于StreamHandler类，并安装了一个流。这个流的write方法将流显示输出到文本框中。

```
class WindowHandler extends StreamHandler
{
    public WindowHandler()
    {
        ...
        final JTextArea output = new JTextArea();
        setOutputStream(new
            OutputStream()
            {
                public void write(int b) {} // not called
                public void write(byte[] b, int off, int len)
                {
                    output.append(new String(b, off, len));
                }
            });
    }
    ...
}
```

使用这种方式只存在一个问题，这就是处理器会缓存记录，并且只有在缓存满的时候才将它们写入流中，因此，需要覆盖public方法，以便在处理器获得每个记录之后刷新缓冲区。

```
class WindowHandler extends StreamHandler
{
    ...
    public void publish(LogRecord record)
    {
        super.publish(record);
        flush();
    }
}
```

如果希望编写更加复杂的流处理器，就应该扩展Handler类，并自定义public、flush和close方法。

11.5.6 过滤器

在默认情况下，过滤器根据日志记录的级别进行过滤。每个日志记录器和处理器都可以有一个可选的过滤器来完成附加的过滤。另外，可以通过实现Filter接口并定义下列方法来自定义过滤器。

```
boolean isLoggable(LogRecord record)
```

在这个方法中，可以利用自己喜欢的标准，对日志记录进行分析，返回true表示这些记录应该包含在日志中。例如，某个过滤器可能只对entering方法和exiting方法产生的消息感兴趣，这个过滤器可以调用 record.getMessage() 方法，并查看这个消息是否用ENTRY或RETURN开头。

要想将一个过滤器安装到一个日志记录器或处理器中，只需要调用setFilter方法就可以了。注意，同一时刻最多只能有一个过滤器。

11.5.7 格式化器

ConsoleHandler类和FileHandler类可以生成文本和XML格式的日志记录。但是，也可以自定义格式。这需要扩展Formatter类并覆盖下面这个方法：

```
String format(LogRecord record)
```

可以根据自己的愿望对记录中的信息进行格式化，并返回结果字符串。在format方法中，有可能会调用下面这个方法

```
String formatMessage(LogRecord record)
```

这个方法对记录中的部分消息进行格式化、参数替换和本地化应用操作。

很多文件格式（例如XML）需要在已格式化的记录的前后加上一个头部和尾部。在这个例子中，要覆盖下面两个方法：

```
String getHead(Handler h)  
String getTail(Handler h)
```

最后，调用setFormatter方法将格式化器安装到处理器中。

11.5.8 日志记录说明

面对日志记录如此之多的可选项，很容易让人忘记最基本的东西。下面的“日志说明书”总结了一些最常用的操作。

1) 为一个简单的应用程序，选择一个日志记录器，并把日志记录器命名为与主应用程序包一样的名字，例如，com.mycompany.myprog，这是一种好的编程习惯。另外，可以通过调用下列方法得到日志记录器。

```
Logger logger = Logger.getLogger("com.mycompany.myprog");
```

为了方便起见，可能希望利用一些日志操作将下面的静态域添加到类中：

```
private static final Logger logger = Logger.getLogger("com.mycompany.myprog");
```

2) 默认的日志配置将级别等于或高于INFO级别的所有消息记录到控制台。用户可以覆盖默认的配置文件。但是正如前面所述，改变配置需要做相当多的工作。因此，最好在应用程序中安装一个更加适宜的默认配置。

下列代码确保将所有的消息记录到应用程序特定的文件中。可以将这段代码放置在应用程序的main方法中。

```
if (System.getProperty("java.util.logging.config.class") == null  
    && System.getProperty("java.util.logging.config.file") == null)
```

```

{
    try
    {
        Logger.getLogger("").setLevel(Level.ALL);
        final int LOG_ROTATION_COUNT = 10;
        Handler handler = new FileHandler("%h/myapp.log", 0, LOG_ROTATION_COUNT);
        Logger.getLogger("").addHandler(handler);
    }
    catch (IOException e)
    {
        logger.log(Level.SEVERE, "Can't create log file handler", e);
    }
}

```

3) 现在, 可以记录自己想要的内容了。但需要牢记: 所有级别为INFO、WARNING和SEVERE的消息都将显示到控制台上。因此, 最好只将对程序用户有意义的消息设置为这几个级别。将程序员想要的日志记录, 设定为FINE是一个很好的选择。

当调用System.out.println时, 实际上生成了下面的日志消息:

```
logger.fine("File open dialog canceled");
```

记录那些不可预料的异常也是一个不错的想法。例如,

```

try
{
    . . .
}
catch (SomeException e)
{
    logger.log(Level.FINE, "explanation", e);
}

```

例11-2利用上述说明可实现: 日志记录消息也显示在日志窗口中。

例11-2 LoggingImageViewer.java

```

1. import java.awt.*;
2. import java.awt.event.*;
3. import java.io.*;
4. import java.util.logging.*;
5. import javax.swing.*;
6.
7. /**
8.  * A modification of the image viewer program that logs various events.
9.  * @version 1.02 2007-05-31
10.  * @author Cay Horstmann
11.  */
12. public class LoggingImageViewer
13. {
14.     public static void main(String[] args)
15.     {
16.         if (System.getProperty("java.util.logging.config.class") == null
17.             && System.getProperty("java.util.logging.config.file") == null)
18.         {
19.             try
20.             {
21.                 Logger.getLogger("com.horstmann.corejava").setLevel(Level.ALL);

```

```
22.         final int LOG_ROTATION_COUNT = 10;
23.         Handler handler = new FileHandler("%h/LoggingImageViewer.log", 0, LOG_ROTATION_COUNT);
24.         Logger.getLogger("com.horstmann.corejava").addHandler(handler);
25.     }
26.     catch (IOException e)
27.     {
28.         Logger.getLogger("com.horstmann.corejava").log(Level.SEVERE,
29.             "Can't create log file handler", e);
30.     }
31. }
32.
33. EventQueue.invokeLater(new Runnable()
34. {
35.     public void run()
36.     {
37.         Handler windowHandler = new WindowHandler();
38.         windowHandler.setLevel(Level.ALL);
39.         Logger.getLogger("com.horstmann.corejava").addHandler(windowHandler);
40.
41.         JFrame frame = new ImageViewerFrame();
42.         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
43.
44.         Logger.getLogger("com.horstmann.corejava").fine("Showing frame");
45.         frame.setVisible(true);
46.     }
47. });
48. }
49. }
50.
51. /**
52.  * The frame that shows the image.
53.  */
54. class ImageViewerFrame extends JFrame
55. {
56.     public ImageViewerFrame()
57.     {
58.         logger.entering("ImageViewerFrame", "<init>");
59.         setTitle("LoggingImageViewer");
60.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
61.
62.         // set up menu bar
63.         JMenuBar menuBar = new JMenuBar();
64.         setJMenuBar(menuBar);
65.
66.         JMenu menu = new JMenu("File");
67.         menuBar.add(menu);
68.
69.         JMenuItem openItem = new JMenuItem("Open");
70.         menu.add(openItem);
71.         openItem.addActionListener(new FileOpenListener());
72.
73.         JMenuItem exitItem = new JMenuItem("Exit");
74.         menu.add(exitItem);
75.         exitItem.addActionListener(new ActionListener()
76.         {
77.             public void actionPerformed(ActionEvent event)
78.             {
```



```

79.         logger.fine("Exiting.");
80.         System.exit(0);
81.     }
82. });
83.
84. // use a label to display the images
85. label = new JLabel();
86. add(label);
87. logger.exiting("ImageViewerFrame", "<init>");
88. }
89.
90. private class FileOpenListener implements ActionListener
91. {
92.     public void actionPerformed(ActionEvent event)
93.     {
94.         logger.entering("ImageViewerFrame.FileOpenListener", "actionPerformed", event);
95.
96.         // set up file chooser
97.         JFileChooser chooser = new JFileChooser();
98.         chooser.setCurrentDirectory(new File("."));
99.
100.        // accept all files ending with .gif
101.        chooser.setFileFilter(new javax.swing.filechooser.FileFilter()
102.        {
103.            public boolean accept(File f)
104.            {
105.                return f.getName().toLowerCase().endsWith(".gif") || f.isDirectory();
106.            }
107.
108.            public String getDescription()
109.            {
110.                return "GIF Images";
111.            }
112.        });
113.
114.        // show file chooser dialog
115.        int r = chooser.showOpenDialog(ImageViewerFrame.this);
116.
117.        // if image file accepted, set it as icon of the label
118.        if (r == JFileChooser.APPROVE_OPTION)
119.        {
120.            String name = chooser.getSelectedFile().getPath();
121.            logger.log(Level.FINE, "Reading file {0}", name);
122.            label.setIcon(new ImageIcon(name));
123.        }
124.        else logger.fine("File open dialog canceled.");
125.        logger.exiting("ImageViewerFrame.FileOpenListener", "actionPerformed");
126.    }
127. }
128.
129. private JLabel label;
130. private static Logger logger = Logger.getLogger("com.horstmann.corejava");
131. private static final int DEFAULT_WIDTH = 300;
132. private static final int DEFAULT_HEIGHT = 400;
133. }
134.
135. /**

```

```
136. * A handler for displaying log records in a window.
137. */
138. class WindowHandler extends StreamHandler
139. {
140.     public WindowHandler()
141.     {
142.         frame = new JFrame();
143.         final JTextArea output = new JTextArea();
144.         output.setEditable(false);
145.         frame.setSize(200, 200);
146.         frame.add(new JScrollPane(output));
147.         frame.setFocusableWindowState(false);
148.         frame.setVisible(true);
149.         setOutputStream(new OutputStream()
150.             {
151.                 public void write(int b)
152.                 {
153.                     } // not called
154.
155.                 public void write(byte[] b, int off, int len)
156.                 {
157.                     output.append(new String(b, off, len));
158.                 }
159.             });
160.     }
161.
162.     public void publish(LogRecord record)
163.     {
164.         if (!frame.isVisible()) return;
165.         super.publish(record);
166.         flush();
167.     }
168.
169.     private JFrame frame;
170. }
```

API java.util.logging.Logger 1.4

- Logger getLogger(String loggerName)
- Logger getLogger(String loggerName, String bundleName)
获得给定名字的日志记录器。如果这个日志记录器不存在，创建一个日志记录器。
参数：loggerName 具有层次结构的日志记录器名。例如， com.mycompany.myapp
 bundleName 用来查看本地消息的资源包名。
- void severe(String message)
- void warning(String message)
- void info(String message)
- void config(String message)
- void fine(String message)
- void finer(String message)
- void finest(String message)

记录一个由方法名和给定消息指示级别的日志记录。

- void entering(String className, String methodName)
- void entering(String className, String methodName, Object param)
- void entering(String className, String methodName, Object[] param)
- void exiting(String className, String methodName)
- void exiting(String className, String methodName, Object result)

记录一个描述进入/退出方法的日志记录，其中应该包括给定参数和返回值。

- void throwing(String className, String methodName, Throwable t)

记录一个描述抛出给定异常对象的日志记录。

- void log(Level level, String message)
- void log(Level level, String message, Object obj)
- void log(Level level, String message, Object[] objs)
- void log(Level level, String message, Throwable t)

记录一个给定级别和消息的日志记录，其中可以包括对象或者可抛出对象。要想包括对象，消息中必须包含格式化占位符{0}、{1}等。

- void logp(Level level, String className, String methodName, String message)
- void logp(Level level, String className, String methodName, String message, Object obj)
- void logp(Level level, String className, String methodName, String message, Object[] objs)
- void logp(Level level, String className, String methodName, String message, Throwable t)

记录一个给定级别、准确的调用者信息和消息的日志记录，其中可以包括对象或可抛出对象。

- void logrb(Level level, String className, String methodName, String bundleName, String message)
- void logrb(Level level, String className, String methodName, String bundleName, String message, Object obj)
- void logrb(Level level, String className, String methodName, String bundleName, String message, Object[] objs)
- void logrb(Level level, String className, String methodName, String bundleName, String message, Throwable t)

记录一个给定级别、准确的调用者信息、资源包名和消息的日志记录，其中可以包括对象或可抛出对象。

- Level getLevel()
- void setLevel(Level l)

获得和设置这个日志记录器的级别。

- `Logger getParent()`
- `void setParent(Logger l)`
获得和设置这个日志记录器的父日志记录器。
- `Handler[] getHandlers()`
获得这个日志记录器的所有处理器。
- `void addHandler(Handler h)`
- `void removeHandler(Handler h)`
增加或删除这个日志记录器中的一个处理器。
- `boolean getUseParentHandlers()`
- `void setUseParentHandlers(boolean b)`
获得和设置“use parent handler”属性。如果这个属性是true，则日志记录器会将全部的日志记录转发给它的父处理器。
- `Filter getFilter()`
- `void setFilter(Filter f)`
获得和设置这个日志记录器的过滤器。

API `java.util.logging.Handler 1.4`

- `abstract void publish(LogRecord record)`
将日志记录发送到希望的目的地。
- `abstract void flush()`
刷新所有已缓冲的数据。
- `abstract void close()`
刷新所有已缓冲的数据，并释放所有相关的资源。
- `Filter getFilter()`
- `void setFilter(Filter f)`
获得和设置这个处理器的过滤器。
- `Formatter getFormatter()`
- `void setFormatter(Formatter f)`
获得和设置这个处理器的格式化器。
- `Level getLevel()`
- `void setLevel(Level l)`
获得和设置这个处理器的级别。

API `java.util.logging.ConsoleHandler 1.4`

- `ConsoleHandler()`
构造一个新的控制台处理器。

API java.util.logging.FileHandler 1.4

- FileHandler(String pattern)
- FileHandler(String pattern, boolean append)
- FileHandler(String pattern, int limit, int count)
- FileHandler(String pattern, int limit, int count, boolean append)

构造一个文件处理器。

参数：pattern	构造日志文件名的模式。参见表11-3列出的模式变量
limit	在打开一个新日志文件之前，日志文件可以包含的近似最大字节数
count	循环序列的文件数量
append	新构造的文件处理器对象应该追加在一个已存在的日志文件尾部，则为true

API java.util.logging.LogRecord 1.4

- Level getLevel()
获得这个日志记录的记录级别。
- String getLoggerName()
获得正在记录这个日志记录的日志记录器的名字。
- ResourceBundle getresourceBundle()
- String getresourceBundleName()
获得用于本地化消息的资源包或资源包的名字。如果没有获得，则返回null。
- String getMessage()
获得本地化和格式化之前的原始消息。
- Object[] getParameters()
获得参数对象。如果没有获得，则返回null。
- Throwable getThrown()
获得被抛出的对象。如果不存在，则返回null。
- String getSourceClassName()
- String getSourceMethodName()
获得记录这个日志记录的代码区域。这个信息有可能是由日志记录代码提供的，也有可能是自动从运行时堆栈推测出来的。如果日志记录代码提供的值有误，或者运行时代码由于被优化而无法推测出确切的位置，这两个方法的返回值就有可能不准确。
- long getMillis()
获得创建时间。以毫秒为单位（从1970年开始）。
- long getSequenceNumber()
获得这个日志记录的惟一序列序号。
- int getThreadID()

获得创建这个日志记录的线程的惟一ID。这些ID是由LogRecord类分配的，并且与其他线程的ID无关。

API java.util.logging.Filter 1.4

- boolean isLoggable(LogRecord record)
如果给定日志记录需要记录，则返回true。

API java.util.logging.Formatter 1.4

- abstract String format(LogRecord record)
返回对日志记录格式化后得到的字符串。
- String getHead(Handler h)
- String getTail(Handler h)
返回应该出现在包含日志记录的文档的开头和结尾的字符串。超类Formatter定义了这些方法，它们只返回空字符串。如果必要的话，可以对它们进行重载。
- String formatMessage(LogRecord record)
返回经过本地化和格式化后的日志记录的消息内容。

11.6 调试技术

假设编写了一个程序，并对所有的异常进行了捕获和恰当的处理，然后，运行这个程序，但还是出现问题，现在该怎么办呢（如果从来没有遇到过这种情况，可以跳过本章的剩余部分）？

当然，如果有一个方便且功能强大的调试器就太好了。调试器是Eclipse、NetBeans这类专业集成开发环境的一部分。本章稍后将讨论调试器。本节，在启动调试器之前，先给出一些有价值的建议。

1) 可以用下面的方法打印或记录任意变量的值：

```
System.out.println(x);
```

或者

```
logger.global().info(x);
```

如果x是一个数值，则会被转换成等价的字符串。如果x是一个对象，那么Java就会调用这个对象的toString方法。要想获得隐式参数对象的状态，就可以打印这个对象的状态。

```
logger.global().info("Obj = " + obj);
```

Java类库中的绝大多数类都覆盖了toString方法，以便能够提供有用的类信息。这样会使调试更加便捷。在自定义的类中，也应该这样做。

2) 一个不太为人所知，但却非常有效的技巧是在每一个类中放置一个main方法。这样就可以对每一个类进行单元测试。

```
public class MyClass
```

```
{
    // methods and fields
}
```

```

public static void main(String[] args)
{
    test();
}

```

利用这种技巧，只需要创建少量的对象，调用所有的方法，并检测每个方法是否能够正确地运行就可以了。另外，可以为每个类保留一个main方法，然后分别为每个文件调用Java虚拟机进行运行测试。在运行applet应用程序的时候，这些main方法不会被调用，而在运行应用程序的时候，Java虚拟机只调用启动类的main方法。

3) 如果喜欢使用前面所讲述的技巧，就应该到<http://junit.org>网站上查看一下JUnit。JUnit是一个非常常见的单元测试框架，利用它可以很容易地组织几套测试用例。只要修改类，就需要运行测试。在发现bug时，还要补充一些其他的测试用例。

4) 日志代理 (logging proxy) 是一个子类的对象，它可以窃取方法调用，并进行日志记录，然后调用超类中的方法。例如，如果在调用一个面板的 setBackground方法时出现了问题，就可以按照下面的方式，以匿名子类实例的形式创建一个代理对象：

```

JPanel proxy = new
    JPanel()
{
    public void setBackground(Color c)
    {
        logger.global.info("setBackground()");
        super.setBackground(c);
    }
}

```

当调用setBackground方法时，就会产生一个日志消息。要想知道谁调用了这个方法，就要生成一个堆栈跟踪。

5) 利用Throwable类提供的printStackTrace方法，可以从任何一个异常对象中获得堆栈情况。下面的代码将捕获任何异常，打印异常对象和堆栈跟踪，然后，重新抛出异常，以便能够找到相应的处理器。

```

try
{
    ...
}
catch (Throwable t)
{
    t.printStackTrace();
    throw t;
}

```

不一定要通过捕获异常来生成堆栈跟踪。只要在代码的任何位置插入下面这条语句就可以获得堆栈跟踪：

```
Thread.dumpStack();
```

6) 一般来说，堆栈跟踪显示在System.err上。也可以利用printStackTrace(PrintWriter s)方法将它发送到一个文件中。另外，如果想记录或显示堆栈跟踪，就可以采用下面的方式，将它

捕获到一个字符串中：

```
StringWriter out = new StringWriter();
new Throwable().printStackTrace(new PrintWriter(out));
String trace = out.toString();
```

有关PrintWriter和StringWriter两个类的详细情况，请参看卷II第1章。

7) 通常，将一个程序中的错误信息保存在一个文件中是非常有用的。然而，错误信息被发送到System.err中，而不是System.out中。因此，不能够通过运行下面的语句获取它们：

```
java MyProgram > errors.txt
```

而是采用下面的方式捕获错误流：

```
java MyProgram 2> errors.txt
```

要想在同一个文件中同时捕获System.err和System.out，需要使用下面这条命令

```
java MyProgram >& errors.txt
```

这条命令将工作在Windows外壳中。

8) 让非捕获异常的堆栈跟踪出现在System.err中并不是一个很理想的方法。如果在客户端偶然看到这些消息，则会感到迷惑，并且在需要的时候也无法实现诊断目的。比较好的方式是将这些内容记录到一个文件中。在Java SE5.0中，可以调用静态的Thread.setDefaultUncaughtExceptionHandler方法改变非捕获异常的处理器：

```
Thread.setDefaultUncaughtExceptionHandler(
    new Thread.UncaughtExceptionHandler()
    {
        public void uncaughtException(Thread t, Throwable e)
        {
            save information in log file
        }
    });
```

9) 要想观察类的加载过程，可以用-verbose标志启动Java虚拟机。这样就可以看到如下所示的输出结果：

```
[Opened /usr/local/jdk5.0/jre/lib/rt.jar]
[Opened /usr/local/jdk5.0/jre/lib/jsse.jar]
[Opened /usr/local/jdk5.0/jre/lib/jce.jar]
[Opened /usr/local/jdk5.0/jre/lib/charsets.jar]
[Loaded java.lang.Object from shared objects file]
[Loaded java.io.Serializable from shared objects file]
[Loaded java.lang.Comparable from shared objects file]
[Loaded java.lang.CharSequence from shared objects file]
[Loaded java.lang.String from shared objects file]
[Loaded java.lang.reflect.GenericDeclaration from shared objects file]
[Loaded java.lang.reflect.Type from shared objects file]
[Loaded java.lang.reflect.AnnotatedElement from shared objects file]
[Loaded java.lang.Class from shared objects file]
[Loaded java.lang.Cloneable from shared objects file]
...
```

有时候，这种方法有助于诊断由于类路径引发的问题。

10) 如果曾经看到过Swing窗口，并对设计者能够采用某种管理手段将所有的组件排列整齐而感到好奇，就可以监视一下有关的信息。按下CTRL+SHIFT+F1，将会按照层次结构打印出所有组件的信息：

```
FontDialog[frame0,0,0,300x200,layout=java.awt.BorderLayout,...
javax.swing.JRootPane[,4,23,292x173,layout=javax.swing.JRootPane$RootLayout,...
javax.swing.JPanel[null.glassPane,0,0,292x173,hidden,layout=java.awt.FlowLayout,...
javax.swing.JLayeredPane[null.layeredPane,0,0,292x173,...
javax.swing.JPanel[null.contentPane,0,0,292x173,layout=java.awt.GridBagLayout,...
javax.swing.JList[,0,0,73x152,alignmentX=null,alignmentY=null],...
    javax.swing.CellRendererPane[,0,0,0x0,hidden]
        javax.swing.DefaultListCellRenderer$UIResource[, -73, -19, 0x0, ...
javax.swing.JCheckBox[,157,13,50x25,layout=javax.swing.OverlayLayout,...
javax.swing.JCheckBox[,156,65,52x25,layout=javax.swing.OverlayLayout,...
javax.swing.JLabel[,114,119,30x17,alignmentX=0.0,alignmentY=null],...
javax.swing.JTextField[,186,117,105x21,alignmentX=null,alignmentY=null],...
javax.swing.JTextField[,0,152,291x21,alignmentX=null,alignmentY=null],...
```

11) 如果自定义Swing组件，却不能正确地显示，则会发现Swing图形调试器（Swing graphics debugger）是一个不错的工具。即使自己不编写组件类，亲眼看看组件的内容如何被绘制出来也非常有意义且有趣。要想对一个Swing组件进行调试，可以调用JComponent 类的setDebugGraphicsOptions方法。下面是调用这个方法的可选项：

DebugGraphics.FLASH_OPTION	在绘制每条线段、每个矩形、每个文本之前，用红色闪烁显示
DebugGraphics.LOG_OPTION	打印每次绘制操作的信息
DebugGraphics.BUFFERED_OPTION	显示在显示区域之外执行的缓冲操作
DebugGraphics.NONE_OPTION	关闭图形调试

我们已经发现：要使闪烁选项能够工作，就必须禁用“双重缓冲”。这是Swing为缓解更新窗口时屏幕抖动现象所采取的一种策略。下面是用来开启闪烁选项的代码：

```
RepaintManager.currentManager(getRootPane()).setDoubleBufferingEnabled(false);
((JComponent) getContentPane()).setDebugGraphicsOptions(DebugGraphics.FLASH_OPTION);
```

只要将这几行代码放置在框架构造器的尾部就可以了。当运行程序时，可以看到内容窗格以极慢的速度填充。如果要进行本地化调试，只要为单个组件调用setDebugGraphicsOptions方法即可。对于这种闪烁，可以设置的参数包括持续时间、次数和闪烁的颜色。有关DebugGraphics类更加详细的内容请参看在线文档。

12) Java SE 5.0增加了-Xlint选项，这样，编译器就可以对一些普遍容易出现的代码问题进行检查。例如，如果使用下面这条命令编译：

```
javac -Xlint:fallthrough
```

当switch语句中缺少break语句时，编译器就会给出报告（术语“lint”最初用来描述一种定位C程序中潜在问题的工具，现在通常用于描述查找可疑的、但不违背语法规则的代码问题的工具。）

下面列出了可以使用选项：

-Xlint 或 -Xlint:all	执行所有的检查
-Xlint:deprecation	与-deprecation一样，检查反对使用的方法

-Xlint:fallthrough	检查switch语句中是否缺少break语句
-Xlint:finally	警告finally子句不能正常地执行
-Xlint:none	不执行任何检查
-Xlint:path	检查类路径和源代码路径上的所有目录是否存在
-Xlint:serial	警告没有serialVersionUID的串行化类（请参看卷II第1章）
-Xlint:unchecked	对通用类型与原始类型之间的危险转换给予警告（请参看第12章）

13) Java SE 5.0增加了对Java应用程序进行监控（monitoring）和管理（management）的支持。它允许利用虚拟机中的代理装置跟踪内存消耗、线程使用、类加载等情况。这个功能对于像应用程序服务器这样大型的、长时间运行的Java程序来说特别重要。下面是一个能够展示这种功能的例子：JDK加载了一个称为jconsole的图形工具，可以用于显示虚拟机性能的统计结果，如图11-3所示。找出运行虚拟机的操作系统进程的ID。在UNIX/Linux环境下，运行ps实用工具，在Windows环境下，使用任务管理器。然后运行jconsole程序：

jconsole processID

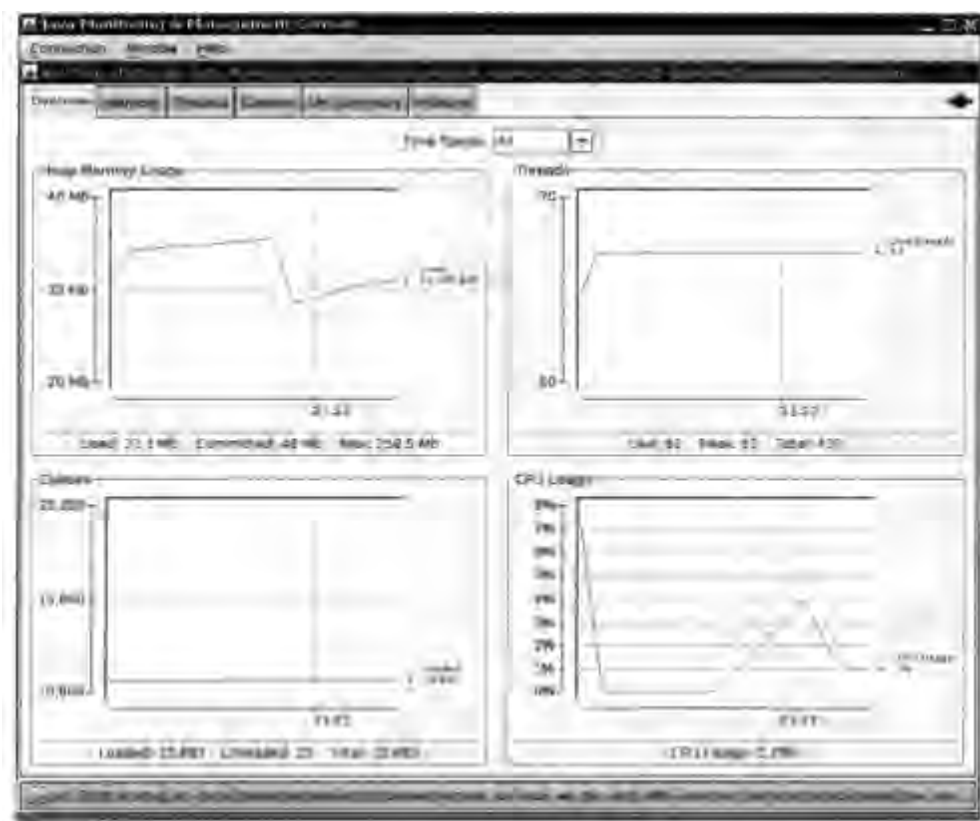


图11-3 jconsole程序

控制台给出了有关运行程序的大量信息。有关更加详细的信息参见<http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html>。



注释：在Java SE 6之前，需要用-Dcom.sun.management.jmxremote option启动程序：


```
java -Dcom.sun.management.jmxremote MyProgram
jconsole processID
```

14) 可以使用jmap实用工具获得一个堆的堆放处，其中显示了堆中的每个对象。使用命令如下：

```
jmap -dump:format=b,file=dumpFileName processID  
jhat dumpFileName
```

然后，通过浏览器进入localhost:7000，将会运行一个网络应用程序，借此探查倾到对象时堆的内容。

15) 如果使用-Xprof标志运行Java虚拟机，就会运行一个基本的剖析器来跟踪那些代码中经常被调用的方法。剖析信息将发送给System.out。输出结果中还会显示哪些方法是由just-in-time编译器编译的。

 警告：编译器的-X选项并没有正式支持，而且在有些JDK版本中并不存在这个选项。可以运行命令java -X得到所有非标准选项的列表。

11.6.1 使用控制台窗口

在调试applet应用程序时可以在窗口中看到错误消息。在Java内置的配置面板中可以选择Show Java Console（请参看第10章）。Java Console窗口有一组滚动条，当消息已经滚动到窗口之外时，可以利用滚动条将它们滚动回来。用户将会发现：使用窗口明显优越于采用System.out和System.err将消息显示在DOS窗口中。

这里提供一个类似的窗口类，以便在调试程序时，能够在具有同样效果的窗口中查看调试消息。图11-4显示了运行ConsoleWindow类的效果。



图11-4 控制台窗口

这个类的用法很简单。只需要进行下面这个调用就可以了：

```
ConsoleWindow.init()
```

然后，用常规的方式打印System.out和System.err。

例11-3列出了ConsoleWindow类的代码清单。可以看到，这个类相当简单。消息将显示在JScrollPane内的JTextArea中。在这个类中，通过调用System.setOut和System.setErr方法将输出流和错误流定向到一个特定的流中，这个流将所有的消息加入到一个文本区中。

例11-3 ConsoleWindow.java

```
1. import javax.swing.*;
2. import java.io.*;
3.
4. /**
5.  A window that displays the bytes sent to System.out and System.err
6.  @version 1.01 2004-05-10
7.  @author Cay Horstmann
8.  */
9. public class ConsoleWindow
10. {
11.     public static void init()
12.     {
13.         JFrame frame = new JFrame();
14.         frame.setTitle("ConsoleWindow");
15.         final JTextArea output = new JTextArea();
16.         output.setEditable(false);
17.         frame.add(new JScrollPane(output));
18.         frame.setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
19.         frame.setLocation(DEFAULT_LEFT, DEFAULT_TOP);
20.         frame.setFocusableWindowState(false);
21.         frame.setVisible(true);
22.
23.         // define a PrintStream that sends its bytes to the output text area
24.         PrintStream consoleStream = new PrintStream(new
25.             OutputStream()
26.             {
27.                 public void write(int b) {} // never called
28.                 public void write(byte[] b, int off, int len)
29.                 {
30.                     output.append(new String(b, off, len));
31.                 }
32.             });
33.
34.         // set both System.out and System.err to that stream
35.         System.setOut(consoleStream);
36.         System.setErr(consoleStream);
37.     }
38.
39.     public static final int DEFAULT_WIDTH = 300;
40.     public static final int DEFAULT_HEIGHT = 200;
41.     public static final int DEFAULT_LEFT = 200;
42.     public static final int DEFAULT_TOP = 200;
43. }
```

11.6.2 跟踪AWT事件

当用Java设计一个奇特的用户界面时，需要知道AWT将什么事件发送到什么组件上。很遗憾，AWT文档对这部分内容讲述的轻描淡写。例如，假设当用户将鼠标移到屏幕的不同位置时，希望在状态栏上显示一条提示信息，此时应该捕获AWT产生的鼠标和焦点事件。

下面给出一个很有用的Eventtrace类，它可以监控这些事件，并打印输出所有事件的处理方法和参数。图11-5显示了被跟踪的事件。



图11-5 The Eventtracer 类的运行结果

要想监测这些信息，需要希望跟踪事件的组件添加到事件跟踪器中：

```
EventTracer tracer = new EventTracer();
tracer.add(frame);
```

这样就会打印出所有事件的文字化描述信息，如下所示：

```
public abstract void java.awt.event.MouseListener.mouseExited(java.awt.event.MouseEvent):
java.awt.event.MouseEvent[MOUSE_EXITED,(408,14),button=0,clickCount=0] on javax.swing.JBut-
ton[0,345,400x25,...]
public abstract void java.awt.event.FocusListener.focusLost(java.awt.event.FocusEvent):
java.awt.event.FocusEvent[FOCUS_LOST,temporary,opposite=null] on javax.swing.JButton[0,345,400x25,...]
```

可能还希望将这些信息输出到一个文件或控制台窗口上。前面已经讲过实现的具体方式。

例11-4是Eventtracer类的程序代码。尽管这个类实现起来非常复杂，但其思路却十分简单。

例11-4 EventTracer.java

```
1. import java.awt.*;
2. import java.beans.*;
3. import java.lang.reflect.*;
4.
5. /**
6.  * @version 1.31 2004-05-10
7.  * @author Cay Horstmann
8.  */
9. public class EventTracer
10. {
11.     public EventTracer()
12.     {
13.         // the handler for all event proxies
14.         handler = new InvocationHandler()
15.         {
16.             public Object invoke(Object proxy, Method method, Object[] args)
17.             {
18.                 System.out.println(method + ":" + args[0]);
19.                 return null;
20.             }
21.         };
22.     }
```

```
23.
24. /**
25.  * Adds event tracers for all events to which this component and its children can listen
26.  * @param c a component
27.  */
28. public void add(Component c)
29. {
30.     try
31.     {
32.         // get all events to which this component can listen
33.         BeanInfo info = Introspector.getBeanInfo(c.getClass());
34.
35.         EventSetDescriptor[] eventSets = info.getEventSetDescriptors();
36.         for (EventSetDescriptor eventSet : eventSets)
37.             addListener(c, eventSet);
38.     }
39.     catch (IntrospectionException e)
40.     {
41.     }
42.     // ok not to add listeners if exception is thrown
43.
44.     if (c instanceof Container)
45.     {
46.         // get all children and call add recursively
47.         for (Component comp : ((Container) c).getComponents())
48.             add(comp);
49.     }
50. }
51.
52. /**
53.  * Add a listener to the given event set
54.  * @param c a component
55.  * @param eventSet a descriptor of a listener interface
56.  */
57. public void addListener(Component c, EventSetDescriptor eventSet)
58. {
59.     // make proxy object for this listener type and route all calls to the handler
60.     Object proxy = Proxy.newProxyInstance(null, new Class[] { eventSet.getListenerType() },
61.         handler);
62.
63.     // add the proxy as a listener to the component
64.     Method addListenerMethod = eventSet.getAddListenerMethod();
65.     try
66.     {
67.         addListenerMethod.invoke(c, proxy);
68.     }
69.     catch (InvocationTargetException e)
70.     {
71.     }
72.     catch (IllegalAccessException e)
73.     {
74.     }
75.     // ok not to add listener if exception is thrown
76. }
77.
78. private InvocationHandler handler;
79. }
```

1) 当用add方法将一个组件添加到事件跟踪器时, JavaBeans自省类就会分析这个组件中形如addXxxListener(XxxEvent)的所有方法(有关JavaBeans的详细介绍请看卷II的第8章)。对于每一个匹配的方法, 都会生成一个EventSetDescriptor对象, 并将它传递给addListener方法。

2) 如果组件是一个容器, 就列举其中的每一个组件, 并为每个组件递归地调用add方法。

3) 调用addListener方法需要提供两个参数: 一个是希望监测其事件的组件; 另一个是事件设置描述符。调用EventSetDescriptor类的getListenerType方法可以返回一个Class对象, 这个对象描述了诸如 ActionListener或ChangeListener这样的事件监听器接口。随后, 要为接口创建一个代理对象。这个代理处理器只打印被调用的事件方法和参数。调用EventSetDescriptor类的getAddListenerMethod方法可以返回一个Method对象, 使用这个对象可以增加一个代理对象来作为组件的事件监听器。

这个程序是体现反射机制的一个很好例子。在这里, 并不需要一定为JButton类设计一个addActionListener方法, 为JSlider类再设计一个 addChangeListener方法。反射机制会发现这些信息。

例11-5测试了事件跟踪器。这个程序显示了一个含有一个按钮, 一个滚动条的窗口框架, 并且能够跟踪这些组件生成的事件。

例11-5 EventTracerTest.java

```
1. import java.awt.*;
2.
3. import javax.swing.*;
4.
5. /**
6.  * @version 1.13 2007-06-12
7.  * @author Cay Horstmann
8.  */
9. public class EventTracerTest
10. {
11.     public static void main(String[] args)
12.     {
13.         EventQueue.invokeLater(new Runnable()
14.         {
15.             public void run()
16.             {
17.                 JFrame frame = new EventTracerFrame();
18.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.                 frame.setVisible(true);
20.             }
21.         });
22.     }
23. }
24.
25. class EventTracerFrame extends JFrame
26. {
27.     public EventTracerFrame()
28.     {
29.         setTitle("EventTracerTest");
30.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
31.     }
32. }
```

```
32.    // add a slider and a button
33.    add(new JSlider(), BorderLayout.NORTH);
34.    add(new JButton("Test"), BorderLayout.SOUTH);
35.
36.    // trap all events of components inside the frame
37.    EventTracer tracer = new EventTracer();
38.    tracer.add(this);
39. }
40.
41. public static final int DEFAULT_WIDTH = 400;
42. public static final int DEFAULT_HEIGHT = 400;
43. }
```

11.6.3 AWT的Robot类

Java SE1.3版本增加了一个Robot类，它用来将敲击键盘和点击鼠标的事件发送给AWT程序，并能够对用户界面进行自动地检测。

要想获得Robot对象，首先要得到一个GraphicsDevice对象。通过下面的一系列调用获得默认的屏幕设备：

```
GraphicsEnvironment environment = GraphicsEnvironment.getLocalGraphicsEnvironment();
GraphicsDevice screen = environment.getDefaultScreenDevice();
```

然后构造一个Robot对象：

```
Robot robot = new Robot(screen);
```

要想发送一个敲击键盘的事件，就需要告诉robot模拟按下键和释放键的动作：

```
robot.keyPress(KeyEvent.VK_TAB);
robot.keyRelease(KeyEvent.VK_TAB);
```

要想发送一个点击鼠标的事件，首先要模拟移动鼠标的事件，然后再模拟按下和释放鼠标的事件：

```
robot.mouseMove(x, y); // x and y are absolute screen pixel coordinates.
robot.mousePress(InputEvent.BUTTON1_MASK);
robot.mouseRelease(InputEvent.BUTTON1_MASK);
```

下面的思路是：模拟键盘和鼠标的输入，然后进行屏幕快照，以便查看应用程序是否实施了预期的操作。可以调用createScreenCapture方法实现捕捉屏幕的操作：

```
Rectangle rect = new Rectangle(x, y, width, height);
BufferedImage image = robot.createScreenCapture(rect);
```

矩形坐标使用的也是绝对像素坐标。

最后，通常希望上面的各条命令之间加上一个短暂的延迟，以保证应用程序能够捕获到各个事件。设置延迟的方法是：调用delay方法并传递一个以毫秒为单位的延迟时间，例如：

```
robot.delay(1000); // delay by 1000 milliseconds
```

例11-6的程序说明了如何使用robot的方法。Rrobot检测了在第8章中已经出现过的按钮测试程序。首先，按下一个空格键来激活最左侧的按钮，然后robot等待2秒，以便能够看到它所做的一切操作。等待一会儿之后，robot将模拟按下TAB键和空格键来点击下一个按钮。最后，


```
27.         GraphicsDevice screen = environment.getDefaultScreenDevice();
28.
29.         try
30.         {
31.             Robot robot = new Robot(screen);
32.             runTest(robot);
33.         }
34.         catch (AWTException e)
35.         {
36.             e.printStackTrace();
37.         }
38.     }
39. });
40. }
41.
42. /**
43.  * Runs a sample test procedure
44.  * @param robot the robot attached to the screen device
45.  */
46. public static void runTest(Robot robot)
47. {
48.     // simulate a space bar press
49.     robot.keyPress(' ');
50.     robot.keyRelease(' ');
51.
52.     // simulate a tab key followed by a space
53.     robot.delay(2000);
54.     robot.keyPress(KeyEvent.VK_TAB);
55.     robot.keyRelease(KeyEvent.VK_TAB);
56.     robot.keyPress(' ');
57.     robot.keyRelease(' ');
58.
59.     // simulate a mouse click over the rightmost button
60.     robot.delay(2000);
61.     robot.mouseMove(200, 50);
62.     robot.mousePress(InputEvent.BUTTON1_MASK);
63.     robot.mouseRelease(InputEvent.BUTTON1_MASK);
64.
65.     // capture the screen and show the resulting image
66.     robot.delay(2000);
67.     BufferedImage image = robot.createScreenCapture(new Rectangle(0, 0, 400, 300));
68.
69.     ImageFrame frame = new ImageFrame(image);
70.     frame.setVisible(true);
71. }
72. }
73.
74. /**
75.  * A frame to display a captured image
76.  */
77. class ImageFrame extends JFrame
78. {
79.     /**
80.      * @param image the image to display
81.      */
82.     public ImageFrame(Image image)
83.     {
```

```
84.     setTitle("Capture");
85.     setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
86.
87.     JLabel label = new JLabel(new ImageIcon(image));
88.     add(label);
89. }
90.
91. public static final int DEFAULT_WIDTH = 450;
92. public static final int DEFAULT_HEIGHT = 350;
93. }
```

API java.awt.GraphicsEnvironment 1.2

- static GraphicsEnvironment getLocalGraphicsEnvironment()
返回本地的图形环境。
- GraphicsDevice getDefaultScreenDevice()
返回默认的屏幕设备。需要注意，使用多台监视器的计算机，每一个屏幕有一个图形设备。通过调用 getScreenDevices方法可以得到一个保存所有屏幕设备的数组。

API java.awt.Robot 1.3

- Robot(GraphicsDevice device)
构造一个能够与给定设备交互的robot对象。
- void keyPress(int key)
- void keyRelease(int key)
模拟按下或释放按键。
参数：key 键码。有关键码更加详细的信息请参看KeyStroke类
- void mouseMove(int x, int y)
模拟移动鼠标。
参数：x, y 用绝对像素坐标表示的鼠标位置
- void mousePress(int eventMask)
- void mouseRelease(int eventMask)
模拟按下或释放鼠标键。
参数：eventMask 描述鼠标键的事件掩码。有关事件掩码更加详细的信息请参看InputEvent
- void delay(int milliseconds)
根据给定毫秒数延迟robot。
- BufferedImage createScreenCapture(Rectangle rect)
截取屏幕的一部分。
参数：rect 用绝对像素坐标表示的所截取的矩形。

11.7 使用调试器

借助于打印语句进行调试并不是一件令人愉快的事情。在这个过程中，需要频繁地增加和删除这些语句，然后，再对程序进行重新编译。使用调试器情况就好多了。调试器会全速运行程序，直到遇到一个断点为止，然后可以查看任何感兴趣的内容。

在这里，有意识地对第8章中的ButtonTest程序进行了一些破坏性的修改，例11-7是ButtonTest程序的错误版本。当点击任何一个按钮时，没有发生任何事情。请看一下源代码，程序原本设计的功能是：点击一下按钮，将背景颜色设置为与按钮名相同的颜色。

例11-7 BuggyButtonTest.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4.
5. /**
6.  * @version 1.22 2007-05-14
7.  * @author Cay Horstmann
8.  */
9. public class BuggyButtonTest
10. {
11.     public static void main(String[] args)
12.     {
13.         EventQueue.invokeLater(new Runnable()
14.         {
15.             public void run()
16.             {
17.                 BuggyButtonFrame frame = new BuggyButtonFrame();
18.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.                 frame.setVisible(true);
20.             }
21.         });
22.     }
23. }
24.
25. class BuggyButtonFrame extends JFrame
26. {
27.     public BuggyButtonFrame()
28.     {
29.         setTitle("BuggyButtonTest");
30.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
31.
32.         // add panel to frame
33.
34.         BuggyButtonPanel panel = new BuggyButtonPanel();
35.         add(panel);
36.     }
37.
38.     public static final int DEFAULT_WIDTH = 300;
39.     public static final int DEFAULT_HEIGHT = 200;
40. }
41.
42. class BuggyButtonPanel extends JPanel
```

```
43. {  
44.     public BuggyButtonPanel()  
45.     {  
46.         ActionListener listener = new ButtonListener();  
47.  
48.         JButton yellowButton = new JButton("Yellow");  
49.         add(yellowButton);  
50.         yellowButton.addActionListener(listener);  
51.  
52.         JButton blueButton = new JButton("Blue");  
53.         add(blueButton);  
54.         blueButton.addActionListener(listener);  
55.  
56.         JButton redButton = new JButton("Red");  
57.         add(redButton);  
58.         redButton.addActionListener(listener);  
59.     }  
60.  
61.     private class ButtonListener implements ActionListener  
62.     {  
63.         public void actionPerformed(ActionEvent event)  
64.         {  
65.             String arg = event.getActionCommand();  
66.             if (arg.equals("yellow")) setBackground(Color.yellow);  
67.             else if (arg.equals("blue")) setBackground(Color.blue);  
68.             else if (arg.equals("red")) setBackground(Color.red);  
69.         }  
70.     }  
71. }
```

在这样一个简短的程序中，通过阅读源代码可以发现bug。假设这是一个非常复杂的程序，要想通过阅读程序查找错误就不太切合实际了。下面将介绍一下如何使用Eclipse调试器定位错误。



注释：如果使用像JSwat (<http://www.bluemarsh.com/java/jswat/>) 这样的独立调试器，或者非常陈旧的jdb，那么就首先必须用-g选项编译程序。例如：

```
javac -g BuggyButtonTest.java
```

在集成环境中，这个操作是自动完成的。

在Eclipse中，可以选择菜单Run→Debug As→Java Application启动调试器。此时，程序就开始运行。

在actionPerformed方法的第一行中设置一个断点：将鼠标移到希望设置断点的那一行，然后在左侧的空白处点击鼠标右键，并选择Toggle Breakpoint。

一旦Java开始处理actionPerformed方法中的代码就会遇到断点。为此，点击Yellow按钮。调试器将在actionPerformed方法的开始处中断。如图11-7所示。

有两个单步跟踪应用程序的命令。“Step Into”命令跟踪到每个方法调用的内部。“Step Over”命令定位到下一行，而并不跟踪到方法调用的内部。Eclipse使用菜单选项Run→Step Into和Run→Step Over，键盘快捷键示F5和F6。发出两次“Step Over”命令，看一下刚表的位置。

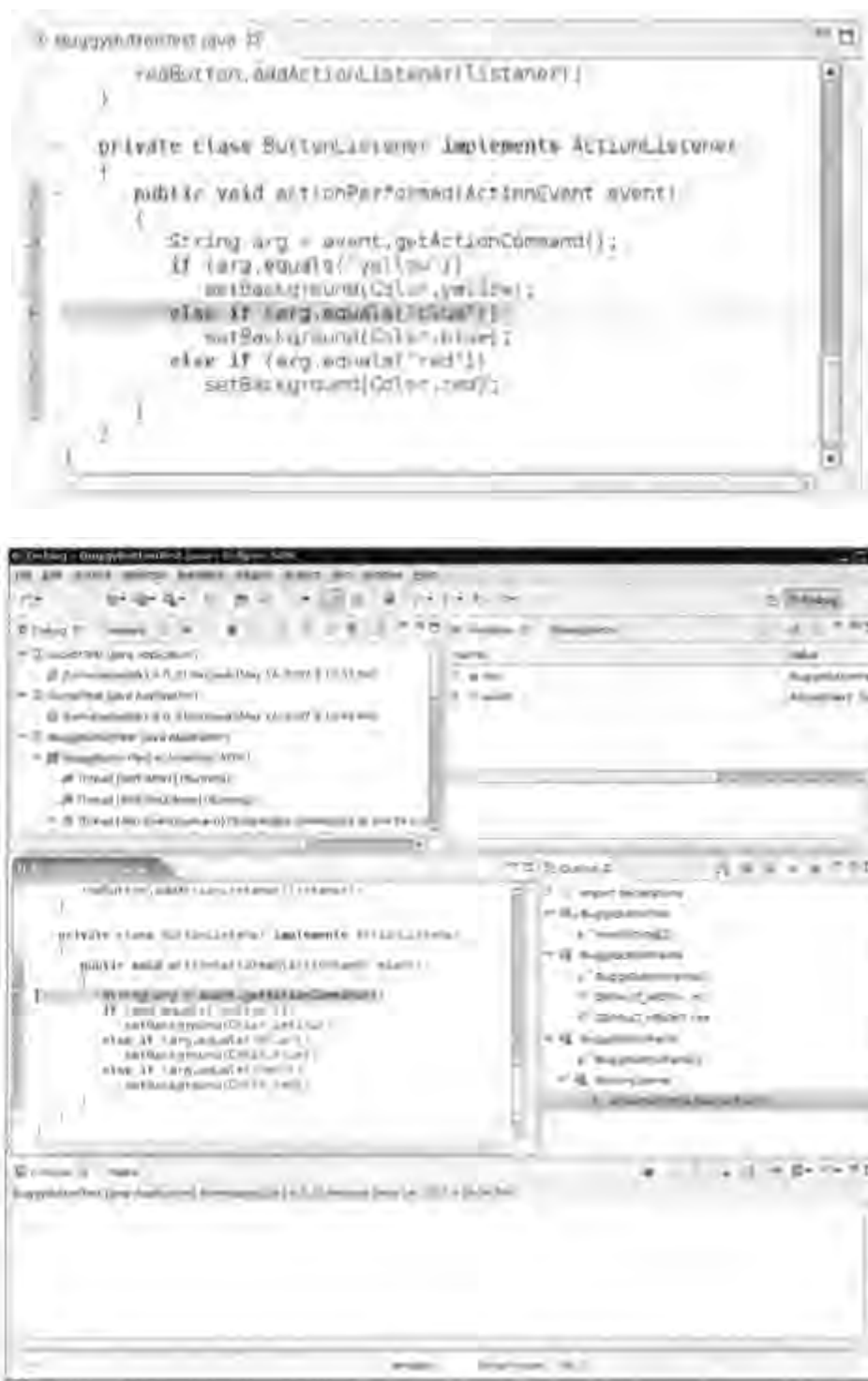


图11-7 停止在断点处

这不是应该发生的事情。假设程序调用`setColor(Color.YELLOW)`，然后退出方法。查看局部变量，并检查`arg`变量的值：



现在，可以看到所发生的变化。arg的值是“Yellow”，Y是大写字母，但是，下面比较测试中的y是小写字母：

```
if (arg.equals("yellow"))
```

神秘般地解决了。

要想退出调试器，需要从菜单中选择Run→Terminate。

在Eclipse中，还有一些高级的调试命令，但是，大都使用刚才看到的简单技术。其他的调试器，如NetBeans调试器页都提供了类似的命令。

本章介绍了异常处理和测试于调试遇到的一些技术。接下来的两章将介绍范型程序设计以及最重要的应用：Java集合框架。