第 9 章 违例差错控制

Java 的基本原理就是“形式错误的代码不会运行”。
与 C++类似，捕获错误最理想的是在编译期间，最好在试图运行程序以前。然而，并非所有错误都能在编译期间侦测到。有些问题必须在运行期间解决，让错误的缔结者通过一些手续向接收者传递一些适当的信息，使其知道该如何正确地处理遇到的问题。
在 C++和其他早期语言中，可通过几种手续来达到这个目的。而且它们通常是作为一种规定建立起来的，而非作为程序设计语言的一部分。典型地，我们需要返回一个值或设置一个标志（位），接收者会检查这些值或标志，判断具体发生了什么事情。然而，随着时间的流逝，终于发现这种做法会助长那些使用一个库的程序员的麻痹情绪。他们往往会这样想：“是的，错误可能会在其他人的代码中出现，但不会在我的代码中”。这样的后果便是他们一般不检查是否出现了错误（有时出错条件确实显得太愚蠢，不值得检验；注释①）。另一方面，若每次调用一个方法时都进行全面、细致的错误检查，那么代码的可读性也可能大幅度降低。由于程序员可能仍然在用这些语言维护自己的系统，所以他们应该对此有着深刻的体会：若按这种方式控制错误，那么在创建大型、健壮、易于维护的程序时，肯定会遇到不小的阻挠。

①：C 程序员研究一下 printf()的返回值便知端详。

解决的方法是在错误控制中排除所有偶然性，强制格式的正确。这种方法实际已有很长的历史，因为早在 60 年代便在操作系统里采用了“违例控制”手段；甚至可以追溯到 BASIC 语言的 on error goto 语句。但 C++的违例控制建立在 Ada 的基础上，而 Java 又主要建立在 C++的基础上（尽管它看起来更象 Object Pascal）。
“违例”（Exception）这个词表达的是一种“例外”情况，亦即正常情况之外的一种“异常”。

在问题发生的时候，我们可能不知具体该如何解决，但肯定知道已不能不顾一切地继续下去。此时，必须坚决地停下来，并由某人、某地指出发生了什么事情，以及该采取何种对策。但为了真正解决问题，当地可能并没有足够多的信息。因此，我们需要将其移交给更级的负责人，令其作出正确的决定（类似一个命令链）。

违例机制的另一项好处就是能够简化错误控制代码。我们再也不用检查一个特定的错误，然后在程序的多处地方对其进行控制。此外，也不需要在方法调用的时候检查错误（因为保证有人能捕获这里的错误）。我们只需要在一个地方处理问题：“违例控制模块”或者“违例控制器”。这样可有效减少代码量，并将那些用于描述具体操作的代码与专门纠正错误的代码分隔开。一般情况下，用于读取、写入以及调试的代码会变得更富有条理。

由于违例控制是由 Java 编译器强行实施的，所以毋需深入学习违例控制，便可正确使用本书编写的大量例子。本章向大家介绍了用于正确控制违例所需的代码，以及在某个方法遇到麻烦的时候，该如何生成自己的违例。

9.1 基本违例

“违例条件”表示在出现什么问题的时候应中止方法或作用域的继续。为了将违例条件与普通问题区分开，违例条件是非常重要的一个因素。在普通问题的情况下，我们在当地已拥有足够的信息，可在某种程度上解决碰到的问题。而在违例条件的情况下，却无法继续下去，因为当地没有提供解决问题所需的足够多的信息。此时，我们能做的唯一事情就是跳出当地环境，将那个问题委托给一个更高级的负责人。这便是出现违例时出现的情况。

一个简单的例子是“除法”。如可能被零除，就有必要进行检查，确保程序不会冒进，并在那种情况下执行除法。但具体通过什么知道分母是零呢？在那个特定的方法里，在我们试图解决的那个问题的环境中，我们或许知道该如何对待一个零分母。但假如它是一个没有预料到的值，就不能对其进行处理，所以必须产生一个违例，而非不顾一切地继续执行下去。

产生一个违例时，会发生几件事情。首先，按照与创建 Java 对象一样的方法创建违例对象：在内存“堆”里，使用 new 来创建。随后，停止当前执行路径（记住不可沿这条路径继续下去），然后从当前的环境中释放出违例对象的句柄。此时，违例控制机制会接管一切，并开始查找一个恰当的地方，用于继续程序的执行。这个恰当的地方便是“违例控制器”，它的职责是从问题中恢复，使程序要么尝试另一条执行路径，要么简单地继续。

作为产生违例的一个简单示例，大家可思考一个名为 t 的对象句柄。有些时候，程序可能传递一个尚未初始化的句柄。所以在用那个对象句柄调用一个方法之前，最好进行一番检查。可将与错误有关的信息发送到一个更大的场景中，方法是创建一个特殊的对象，用它代表我们的信息，并将其“掷”（Throw）出我们当前的场景之外。这就叫作“产生一个违例”或者“掷出一个违例”。下面是它的大概形式：

```
if(t == null)
throw new NullPointerException();
```

这样便“掷”出了一个违例。在当前场景中，它使我们能放弃进一步解决该问题的企图。该问题会被转移到其他更恰当的地方解决。准确地说，那个地方不久就会显露出来。

9.1.1 违例自变量

和 Java 的其他任何对象一样，需要用 new 在内存堆里创建违例，并需调用一个构建器。在所有标准违例中，存在着两个构建器：第一个是默认构建器，第二个则需使用一个字串自变量，使我们能在违例里置入相关信息：

```
if(t == null)
throw new NullPointerException("t = null");
```

稍后，字串可用各种方法提取出来，就象稍后会展示的那样。
在这儿，关键字 throw 会象变戏法一样做出一系列不可思议的事情。它首先执行 new 表达式，创建一个不在程序常规执行范围之内的对象。而且理所当然，会为那个对象调用构建器。随后，对象实际会从方法中返回——尽管对象的类型通常并不是方法设计为返回的类型。为深入理解违例控制，可将其想象成另一种返回机制——但是不要在这个问题上深究，否则会遇到麻烦。通过“掷”出一个违例，亦可从原来的作用域中退出。但是会先返回一个值，再退出方法或作用域。
但是，与普通方法返回的相似性到此便全部结束了，因为我们返回的地方与从普通方法调用中返回的地方是迥然有异的（我们结束于一个恰当的违例控制器，它距离违例“掷”出的地方可能相当遥远——在调用堆栈中要低上许多级）。
此外，我们可根据需要掷出任何类型的“可掷”对象。典型情况下，我们要为每种不同类型的错误“掷”出一类不同的违例。我们的思路是在违例对象以及挑选的违例对象类型中保存信息，所以在更大场景中的某个人可知道如何对待我们的违例（通常，唯一的信息是违例对象的类型，而违例对象中保存的没什么意义）。

9.2 违例的捕获

若某个方法产生一个违例，必须保证该违例能被捕获，并获得正确对待。对于 Java 的违例控制机制，它的一个好处就是允许我们在一个地方将精力集中在要解决的问题上，然后在另一个地方对待来自那个代码内部的错误。
为理解违例是如何捕获的，首先必须掌握“警戒区”的概念。它代表一个特殊的代码区域，有可能产生违例，并在后面跟随用于控制那些违例的代码。

9.2.1 try 块

若位于一个方法内部，并“掷”出一个违例（或在这个方法内部调用的另一个方法产生了违例），那个方法就会在违例产生过程中退出。若不想一个 throw 离开方法，可在那个方法内部设置一个特殊的代码块，用它捕获违例。这就叫作“try 块”，因为要在这个地方“尝试”各种方法调用。try 块属于一种普通的作用域，用一个 try 关键字开头：

```
try {
// 可能产生违例的代码
}
```

若用一种不支持违例控制的编程语言全面检查错误，必须用设置和错误检测代码将每个方法都包围起来——即便多次调用相同的方法。而在使用了违例控制技术后，可将所有东西都置入一个 try 块内，在同一地点捕获所有违例。这样便可极大简化我们的代码，并使其更易辨读，因为代码本身要达到的目标再也不会与繁复的错误检查混淆。

9.2.2 违例控制器

当然，生成的违例必须在某个地方中止。这个“地方”便是违例控制器或者违例控制模块。而且针对想捕获的每种违例类型，都必须有一个相应的违例控制器。违例控制器紧接在 try 块后面，且用 catch（捕获）关键字标记。如下所示：

407-408 页程序

每个 catch 从句——即违例控制器——都类似一个小型方法，它需要采用一个（而且只有一个）特定类型的自变量。可在控制器内部使用标识符（id1，id2 等等），就象一个普通的方法自变量那样。我们有时也根本不使用标识符，因为违例类型已提供了足够的信息，可有效处理违例。但即使不用，标识符也必须就位。
控制器必须“紧接”在 try 块后面。若“掷”出一个违例，违例控制机制就会搜寻自变量与违例类型相符的第一个控制器。随后，它会进入那个 catch 从句，并认为违例已得到控制（一旦 catch 从句结束，对控制器的搜索也会停止）。只有相符的 catch 从句才会得到执行；它与 switch 语句不同，后者在每个 case 后都需要一个 break 命令，防止误执行其他语句。
在 try 块内部，请注意大量不同的方法调用可能生成相同的违例，但只需要一个控制器。

1. 中断与恢复
在违例控制理论中，共存在两种基本方法。在“中断”方法中（Java 和 C++提供了对这种方法的支持），我们假定错误非常关键，没有办法返回违例发生的地方。无论谁只要“掷”出一个违例，就表明没有办法补救错误，而且也不希望再回来。
另一种方法叫作“恢复”。它意味着违例控制器有责任来纠正当前的状况，然后取得出错的方法，假定下一次会成功执行。若使用恢复，意味着在违例得到控制以后仍然想继续执行。在这种情况下，我们的违例更象一个方法调用——我们用它在 Java 中设置各种各样特殊的环境，产生类似于“恢复”的行为（换言之，此时不是“掷”出一个违例，而是调用一个用于解决问题的方法）。另外，也可以将自己的 try 块置入一个 while 循环里，用它不断进入 try 块，直到结果满意时为止。
从历史的角度看，若程序员使用的操作系统支持可恢复的违例控制，最终都会用到类似于中断的代码，并跳过恢复进程。所以尽管“恢复”表面上十分不错，但在实际应用中却显得困难重重。其中决定性的原因可能是：我们的控制模块必须随时留意是否产生了违例，以及是否包含了由产生位置专用的代码。这便使代码很难编写和维护——大型系统尤其如此，因为违例可能在多个位置产生。

9.2.3 违例规范
在 Java 中，对那些要调用方法的客户程序员，我们要通知他们可能从自己的方法里“掷”出违例。这是一种有礼貌的做法，只有它才能使客户程序员准确地知道要编写什么代码来捕获所有潜在的违例。当然，若你同时提供了源码，客户程序员甚至能全盘检查代码，找出相应的 throw 语句。但尽管如此，通常并不随同源码提供库。为解决这个问题，Java 提供了一种特殊的语法格式（并强迫我们采用），以便礼貌地告诉客户程序员该方法会“掷”出什么违例，令对方方便地加以控制。这便是我们在这里要讲述的“违例规范”，它属于方法声明的一部分，位于自变量（参数）列表的后面。
违例规范采用了一个额外的关键字：throws；后面跟随全部潜在的违例类型。因此，我们的方法定义看起来应象下面这个样子：
void f() throws tooBig, tooSmall, divZero { //...
若使用下述代码：
void f() [// ...
它意味着不会从方法里“掷”出违例（除类型为 RuntimeException 的违例以外，它可能从任何地方掷出——稍后还会详细讲述）。
但不能完全依赖违例规范——假若方法造成了一个违例，但没有对其进行控制，编译器会侦测到这个情况，并告诉我们必须控制违例，或者指出应该从方法里“掷”出一个违例规范。通过坚持从顶部到底部排列违例规范，Java 可在编译期保证违例的正确性（注释②）。

②：这是在 C++违例控制基础上一个显著的进步，后者除非到运行期，否则不会捕获不符合违例规范的错误。这使得 C++的违例控制机制显得用处不大。

我们在这个地方可采取欺骗手段：要求“掷”出一个并没有发生的违例。编译器能理解我们的要求，并强迫使用这个方法的用户当作真的产生了那个违例处理。在实际应用中，可将其作为那个违例的一个“占位符”使用。这样一来，以后可以方便地产生实际的违例，毋需修改现有的代码。

9.2.4 捕获所有违例

我们可创建一个控制器，令其捕获所有类型的违例。具体的做法是捕获基础类违例类型 Exception（也存在其他类型的基础违例，但 Exception 是适用于几乎所有编程活动的基础）。如下所示：

```
catch(Exception e) {
System.out.println("caught an exception");
}
```

这段代码能捕获任何违例，所以在实际使用时最好将其置于控制器列表的末尾，防止跟随在后面的任何特殊违例控制器失效。

对于程序员常用的所有违例类来说，由于 Exception 类是它们的基础，所以我们不会获得关于违例太多的信息，但可调用来自它的基础类 Throwable 的方法：

String getMessage()
获得详细的消息。

String toString()
返回对 Throwable 的一段简要说明，其中包括详细的消息（如果有的话）。

void printStackTrace()
void printStackTrace(PrintStream)
打印出 Throwable 和 Throwable 的调用堆栈路径。调用堆栈显示出将我们带到违例发生地点的方法调用的顺序。

第一个版本会打印出标准错误，第二个则打印出我们的选择流程。若在 Windows 下工作，就不能重定向标准错误。因此，我们一般愿意使用第二个版本，并将结果送给 System.out；这样一来，输出就可重定向到我们希望的任何路径。

除此以外，我们还可从 Throwable 的基础类 Object（所有对象的基础类型）获得另外一些方法。对于违例控制来说，其中一个可能有用的是 getClass()，它的作用是返回一个对象，用它代表这个对象的类。我们可依次用 getName()或 toString()查询这个 Class 类的名字。亦可对 Class 对象进行一些复杂的操作，尽管那些操作在违例控制中是不必要的。本章稍后还会详细讲述 Class 对象。

下面是一个特殊的例子，它展示了 Exception 方法的使用（若执行该程序遇到困难，请参考第 3 章 3.1.2 小节“赋值”）：

411 页中程序

该程序输出如下：

411 页下程序

可以看到，该方法连续提供了大量信息——每类信息都是前一类信息的一个子集。

9.2.5 重新“掷”出违例

在某些情况下，我们想重新掷出刚才产生过的违例，特别是在用 Exception 捕获所有可能的违例时。由于我们已拥有当前违例的句柄，所以只需简单地重新掷出那个句柄即可。下面是一个例子：

```
catch(Exception e) {
System.out.println("一个违例已经产生");
throw e;
}
```

重新“掷”出一个违例导致违例进入更高一级环境的违例控制器中。用于同一个 try 块的任何更进一步的 catch 从句仍然会被忽略。此外，与违例对象有关的所有东西都会得到保留，所以用于捕获特定违例类型的更高一级的控制器可以从那个对象里提取出所有信息。

若只是简单地重新掷出当前违例，我们打印出来的、与 printStackTrace()内的那个违例有关的信息会与违例的起源地对应，而不是与重新掷出它的地点对应。若想安装新的堆栈跟踪信息，可调用 fillInStackTrace()，它会返回一个特殊的违例对象。这个违例的创建过程如下：将当前堆栈的信息填充到原来的违例对象里。下面列出它的形式：

412-413 页程序

其中最重要的行号在注释内标记出来。注意第 17 行没有设为注释行。它的输出结果如下：

413 页中程序

因此，违例堆栈路径无论如何都会记住它的真正起点，无论自己被重复“掷”了好几次。

若将第 17 行标注（变成注释行），而撤消对第 18 行的标注，就会换用 fillInStackTrace()，结果如下：

413 页下程序

由于使用的是 fillInStackTrace()，第 18 行成为违例的新起点。

针对 g()和 main()，Throwable 类必须在违例规格中出现，因为 fillInStackTrace()会生成一个 Throwable 对象的句柄。由于 Throwable 是 Exception 的一个基础类，所以有可能获得一个能够“掷”出的对象（具有 Throwable 属性），但却并非一个 Exception（违例）。因此，在 main()中用于 Exception 的句柄可能丢失自己的目标。为保证所有东西均井然有序，编译器强制 Throwable 使用一个违例规范。举个例子来说，下述程序的违例便不会在 main()中被捕获到：

414 页上程序

也有可能从一个已经捕获的违例重新“掷”出一个不同的违例。但假如这样做，会得到与使

用 fillInStackTrace()类似的效果：与违例起源地有关的信息会全部丢失，我们留下的是与新的 throw 有关的信息。如下所示：

414-415 页程序

输出如下：

415 页程序

最后一个违例只知道自己来自 main()，而非来自 f()。注意 Throwable 在任何违例规范中都不是必需的。
永远不必关心如何清除前一个违例，或者与之有关的其他任何违例。它们都属于用 new 创建的、以内存堆为基础的对象，所以垃圾收集器会自动将其清除。

9.3 标准 Java 违例

Java 包含了一个名为 Throwable 的类，它对可以作为违例“掷”出的所有东西进行了描述。Throwable 对象有两种常规类型（亦即“从 Throwable 继承”）。其中，Error 代表编译期和系统错误，我们一般不必特意捕获它们（除在特殊情况以外）。Exception 是可以从任何标准 Java 库的类方法中“掷”出的基本类型。此外，它们亦可从我们自己的方法以及运行期偶发事件中“掷”出。
为获得违例的一个综合概念，最好的方法是阅读由 http://java.sun.com 提供的联机 Java 文档（当然，首先下载它们更好）。为了对各种违例有一个大概的印象，这个工作是相当有价值的。但大家不久就会发现，除名字外，一个违例和下一个违例之间并不存在任何特殊的地方。此外，Java 提供的违例数量正在日益增多；从本质上说，把它们印到一本书里是没有意义的。大家从其他地方获得的任何新库可能也提供了它们自己的违例。我们最需要掌握的是基本概念，以及用这些违例能够做什么。
java.lang.Exception
这是程序能捕获的基本违例。其他违例都是从它衍生出去的。这里要注意的是违例的名字代表发生的问题，而且违例名通常都是精心挑选的，可以很清楚地说明到底发生了什么事情。违例并不全是在 java.lang 中定义的；有些是为了提供对其他库的支持，如 util，net 以及 io 等——我们可以从它们的完整类名中看出这一点，或者观察它们从什么继承。例如，所有 IO 违例都是从 java.io.IOException 继承的。

9.3.1 RuntimeException 的特殊情况

本章的第一个例子是：

```
if(t == null)
throw new NullPointerException();
```

看起来似乎在传递进入一个方法的每个句柄中都必须检查 null（因为不知道调用者是否已传递了一个有效的句柄），这无疑是相当可怕的。但幸运的是，我们根本不必这样做——它属于 Java 进行的标准运行期检查的一部分。若对一个空句柄发出了调用，Java 会自动产生一个 NullPointerException 违例。所以上述代码在任何情况下都是多余的。
这个类别里含有一系列违例类型。它们全部由 Java 自动生成，毋需我们亲自动手把它们包含到自己的违例规范里。最方便的是，通过将它们置入单独一个名为 RuntimeException 的基础类下面，它们全部组合到一起。这是一个很好的继承例子：它建立了一系列具有某种共通

性的类型，都具有某些共通的特征与行为。此外，我们没必要专门写一个违例规范，指出一个方法可能会“掷”出一个 RuntimeException，因为已经假定可能出现那种情况。由于它们用于指出编程中的错误，所以几乎永远不必专门捕获一个“运行期违例”——RuntimeException——它在默认情况下会自动得到处理。若必须检查 RuntimeException，我们的代码就会变得相当繁复。在我们自己的包里，可选择“掷”出一部分 RuntimeException。如果不捕获这些违例，又会出现什么情况呢？由于编译器并不强制违例规范捕获它们，所以假如不捕获的话，一个 RuntimeException 可能过滤掉我们到达 main()方法的所有途径。为体会此时发生的事情，请试试下面这个例子：

417 页上程序

大家已经看到，一个 RuntimeException（或者从它继承的任何东西）属于一种特殊情况，因为编译器不要求为这些类型指定违例规范。
输出如下：

```
java.lang.RuntimeException: From f()
at NeverCaught.f(NeverCaught.java:9)
at NeverCaught.g(NeverCaught.java:12)
at NeverCaught.main(NeverCaught.java:15)
```

所以答案就是：假若一个 RuntimeException 获得到达 main()的所有途径，同时不被捕获，那么当程序退出时，会为那个违例调用 printStackTrace()。
注意也许能在自己的代码中仅忽略 RuntimeException，因为编译器已正确实行了其他所有控制。因为 RuntimeException 在此时代表一个编程错误：
(1) 一个我们不能捕获的错误（例如，由客户程序员接收传递给自己方法的一个空句柄）。
(2) 作为一名程序员，一个应在自己的代码中检查的错误（如 ArrayIndexOutOfBoundException，此时应注意数组的大小）。
可以看出，最好的做法是在这种情况下违例，因为它们有助于程序的调试。
另外一个有趣的地方是，我们不可将 Java 违例划分为单一用途的工具。的确，它们设计用于控制那些讨厌的运行期错误——由代码控制范围之外的其他力量产生。但是，它也特别有助于调试某些特殊类型的编程错误，那些是编译器侦测不到的。

9.4 创建自己的违例

并不一定非要使用 Java 违例。这一点必须掌握，因为经常都需要创建自己的违例，以便指出自己的库可能生成的一个特殊错误——但创建 Java 分级结构的时候，这个错误是无法预知的。
为创建自己的违例类，必须从一个现有的违例类型继承——最好在含义上与新违例近似。继承一个违例相当简单：

418-419 页程序

继承在创建新类时发生：

419 页上程序

这里的关键是“extends Exception”，它的意思是：除包括一个 Exception 的全部含义以外，还有更多的含义。增加的代码数量非常少——实际只添加了两个构建器，对 MyException 的创建方式进行了定义。请记住，假如我们不明确调用一个基础类构建器，编译器会自动调用基础类默认构建器。在第二个构建器中，通过使用 super 关键字，明确调用了带有一个 String 参数的基础类构建器。
该程序输出结果如下：

419 页下程序

可以看到，在从 f()“掷”出的 MyException 违例中，缺乏详细的消息。
创建自己的违例时，还可以采取更多的操作。我们可添加额外的构建器及成员：

419-421 页程序

此时添加了一个数据成员 i；同时添加了一个特殊的方法，用它读取那个值；也添加了一个额外的构建器，用它设置那个值。输出结果如下：

421 页上程序

由于违例不过是另一种形式的对象，所以可以继续这个进程，进一步增强违例类的能力。但要注意，对使用自己这个包的客户程序员来说，他们可能错过所有这些增强。因为他们可能只是简单地寻找准备生成的违例，除此以外不做任何事情——这是大多数 Java 库违例的标准用法。若出现这种情况，有可能创建一个新违例类型，其中几乎不包含任何代码：

```
//: SimpleException.java
class SimpleException extends Exception {
} ///:~
```

它要依赖编译器来创建默认构建器（会自动调用基础类的默认构建器）。当然，在这种情况下，我们不会得到一个 SimpleException(String)构建器，但它实际上也不会经常用到。

9.5 违例的限制

覆盖一个方法时，只能产生已在方法的基础类版本中定义的违例。这是一个重要的限制，因为它意味着与基础类协同工作的代码也会自动应用于从基础类衍生的任何对象（当然，这属于基本的 OOP 概念），其中包括违例。
下面这个例子演示了强加在违例身上的限制类型（在编译期）：

422-423 页程序

在 Inning 中，可以看到无论构建器还是 event()方法都指出自己会“掷”出一个违例，但它们实际上没有那样做。这是合法的，因为它允许我们强迫用户捕获可能在覆盖过的 event()版本里添加的任何违例。同样的道理也适用于 abstract 方法，就象在 atBat()里展示的那样。
“interface Storm”非常有趣，因为它包含了在 Incoming 中定义的一个方法——event()，以及不是在其中定义的一个方法。这两个方法都会“掷”出一个新的违例类型：RainedOut。当执行到“StormyInning extends”和“implements Storm”的时候，可以看到 Storm 中的 event()

方法不能改变 Inning 中的 event()的违例接口。同样地，这种设计是十分合理的；否则的话，当我们操作基础类时，便根本无法知道自己捕获的是否正确的东西。当然，假如 interface 中定义的一个方法不在基础类里，比如 rainHard()，它产生违例时就没什么问题。
对违例的限制并不适用于构建器。在 StormyInning 中，我们可看到一个构建器能够“掷”出它希望的任何东西，无论基础类构建器“掷”出什么。然而，由于必须坚持按某种方式调用基础类构建器（在这里，会自动调用默认构建器），所以衍生类构建器必须在自己的违例规范中声明所有基础类构建器违例。
StormyInning.walk()不会编译的原因是它“掷”出了一个违例，而 Inning.walk()却不会“掷”出。若允许这种情况发生，就可让自己的代码调用 Inning.walk()，而且它不必控制任何违例。但在以后替换从 Inning 衍生的一个类的对象时，违例就会“掷”出，造成代码执行的中断。通过强迫衍生类方法遵守基础类方法的违例规范，对象的替换可保持连贯性。
覆盖过的 event()方法向我们显示出一个方法的衍生类版本可以不产生任何违例——即便基础类版本要产生违例。同样地，这样做是必要的，因为它不会中断那些已假定基础类版本会产生违例的代码。差不多的道理亦适用于 atBat()，它会“掷”出 PopFoul——从 Foul 衍生出来的一个违例，而 Foul 违例是由 atBat()的基础类版本产生的。这样一来，假如有人在自己的代码里操作 Inning，同时调用了 atBat()，就必须捕获 Foul 违例。由于 PopFoul 是从 Foul 衍生的，所以违例控制器（模块）也会捕获 PopFoul。
最后一个有趣的地方在 main()内部。在这个地方，假如我们明确操作一个 StormyInning 对象，编译器就会强迫我们只捕获特定于那个类的违例。但假如我们上溯造型到基础类型，编译器就会强迫我们捕获针对基础类的违例。通过所有这些限制，违例控制代码的“健壮”程度获得了大幅度改善（注释③）。

③：ANSI/ISO C++施加了类似的限制，要求衍生方法违例与基础类方法掷出的违例相同，或者从后者衍生。在这种情况下，C++实际上能够在编译期间检查违例规范。

我们必须认识到这一点：尽管违例规范是由编译器在继承期间强行遵守的，但违例规范并不属于方法类型的一部分，后者仅包括了方法名以及自变量类型。因此，我们不可在违例规范的基础上覆盖方法。除此以外，尽管违例规范存在于一个方法的基础类版本中，但并不表示它必须在方法的衍生类版本中存在。这与方法的“继承”颇有不同（进行继承时，基础类中的方法也必须在衍生类中存在）。换言之，用于一个特定方法的“违例规范接口”可能在继承和覆盖时变得更“窄”，但它不会变得更“宽”——这与继承时的类接口规则是正好相反的。

9.6 用 finally 清除
无论一个违例是否在 try 块中发生，我们经常都想执行一些特定的代码。对一些特定的操作，经常都会遇到这种情况，但在恢复内存时一般都不需要（因为垃圾收集器会自动照料一切）。为达到这个目的，可在所有违例控制器的末尾使用一个 finally 从句（注释④）。所以完整的违例控制小节象下面这个样子：

```
try {
// 要保卫的区域：
// 可能“掷”出 A,B,或 C 的危险情况
} catch (A a1) {
// 控制器 A
```

```
} catch (B b1) {
// 控制器 B
} catch (C c1) {
// 控制器 C
} finally {
// 每次都会发生的情况
}
```

④：C++违例控制未提供 finally 从句，因为它依赖构建器来达到这种清除效果。

为演示 finally 从句，请试验下面这个程序：

426 页上程序

通过该程序，我们亦可知道如何应付 Java 违例（类似 C++的违例）不允许我们恢复至违例产生地方的这一事实。若将自己的 try 块置入一个循环内，就可建立一个条件，它必须在继续程序之前满足。亦可添加一个 static 计数器或者另一些设备，允许循环在放弃以前尝试数种不同的方法。这样一来，我们的程序可以变得更加“健壮”。
输出如下：

426 页下程序

无论是否“掷”出一个违例，finally 从句都会执行。

9.6.1 用 finally 做什么
在没有“垃圾收集”以及“自动调用破坏器”机制的一种语言中（注释⑤），finally 显得特别重要，因为程序员可用它担保内存的正确释放——无论在 try 块内部发生了什么状况。但 Java 提供了垃圾收集机制，所以内存的释放几乎绝对不会成为问题。另外，它也没有构建器可供调用。既然如此，Java 里何时才会用到 finally 呢？

⑤：“破坏器”（Destructor）是“构建器”（Constructor）的反义词。它代表一个特殊的函数，一旦某个对象失去用处，通常就会调用它。我们肯定知道在哪里以及何时调用破坏器。C++提供了自动的破坏器调用机制，但 Delphi 的 Object Pascal 版本 1 及 2 却不具备这一能力（在这种语言中，破坏器的含义与用法都发生了变化）。

除将内存设回原始状态以外，若要设置另一些东西，finally 就是必需的。例如，我们有时需要打开一个文件或者建立一个网络连接，或者在屏幕上画一些东西，甚至设置外部世界的一个开关，等等。如下例所示：

427-428 页程序

这里的目标是保证 main()完成时开关处于关闭状态，所以将 sw.off()置于 try 块以及每个违例控制器的末尾。但产生的一个违例有可能不是在这里捕获的，这便会错过 sw.off()。然而，利用 finally，我们可以将来自 try 块的关闭代码只置于一个地方：

428 页中程序

在这儿，sw.off()已移至一个地方。无论发生什么事情，都肯定会运行它。
即使违例不在当前的 catch 从句集里捕获，finally 都会在违例控制机制转到更高级别搜索一个控制器之前得以执行。如下所示：

428-429 页程序

该程序的输出展示了具体发生的事情：

429 页下程序

若调用了 break 和 continue 语句，finally 语句也会得以执行。请注意，与作上标签的 break 和 continue 一道，finally 排除了 Java 对 goto 跳转语句的需求。

9.6.2 缺点：丢失的违例
一般情况下，Java 的违例实施方案都显得十分出色。不幸的是，它依然存在一个缺点。尽管违例指出程序里存在一个危机，而且绝不应忽略，但一个违例仍有可能简单地“丢失”。在采用 finally 从句的一种特殊配置下，便有可能发生这种情况：

430 页上程序

输出如下：

430 页下程序

可以看到，这里不存在 VeryImportantException（非常重要的违例）的迹象，它只是简单地被 finally 从句中的 HoHumException 代替了。
这是一项相当严重的缺陷，因为它意味着一个违例可能完全丢失。而且就象前例演示的那样，这种丢失显得非常“自然”，很难被人查出蛛丝马迹。而与此相反，C++里如果第二个违例在第一个违例得到控制前产生，就会被当作一个严重的编程错误处理。或许 Java 以后的版本会纠正这个问题（上述结果是用 Java 1.1 生成的）。

9.7 构建器
为违例编写代码时，我们经常要解决的一个问题是：“一旦产生违例，会正确地进行清除吗？”大多数时候都会非常安全，但在构建器中却是一个大问题。构建器将对象置于一个安全的起始状态，但它可能执行一些操作——如打开一个文件。除非用户完成对象的使用，并调用一个特殊的清除方法，否则那些操作不会得到正确的清除。若从一个构建器内部“掷”出一个违例，这些清除行为也可能不会正确地发生。所有这些都意味着在编写构建器时，我们必须特别加以留意。
由于前面刚学了 finally，所以大家可能认为它是一种合适的方案。但事情并没有这么简单，因为 finally 每次都会执行清除代码——即使我们在清除方法运行之前不想执行清除代码。因此，假如真的用 finally 进行清除，必须在构建器正常结束时设置某种形式的标志。而且

只要设置了标志，就不要执行 finally 块内的任何东西。由于这种做法并不完美（需要将一个地方的代码同另一个地方的结合起来），所以除非特别需要，否则一般不要尝试在 finally 中进行这种形式的清除。
在下面这个例子里，我们创建了一个名为 InputFile 的类。它的作用是打开一个文件，然后每次读取它的一行内容（转换为一个字串）。它利用了由 Java 标准 IO 库提供的 FileReader 以及 BufferedReader 类（将于第 10 章讨论）。这两个类都非常简单，大家现在可以毫无困难地掌握它们的基本用法：

431-433 页程序

该例使用了 Java 1.1 IO 类。
用于 InputFile 的构建器采用了一个 String（字串）参数，它代表我们想打开的那个文件的名字。在一个 try 块内部，它用该文件名创建了一个 FileReader。对 FileReader 来说，除非转移并用它创建一个能够实际与之“交谈”的 BufferedReader，否则便没什么用处。注意 InputFile 的一个好处就是它同时合并了这两种行动。
若 FileReader 构建器不成功，就会产生一个 FileNotFoundException（文件未找到违例）。必须单独捕获这个违例——这属于我们不想关闭文件的一种特殊情况，因为文件尚未成功打开。其他任何捕获从句（catch）都必须关闭文件，因为文件已在进入那些捕获从句时打开（当然，如果多个方法都能产生一个 FileNotFoundException 违例，就需要稍微用一些技巧。此时，我们可将不同的情况分隔到数个 try 块内）。close()方法会掷出一个尝试过的违例。即使它在另一个 catch 从句的代码块内，该违例也会得以捕获——对 Java 编译器来说，那个 catch 从句不过是另一对花括号而已。执行完本地操作后，违例会被重新“掷”出。这样做是必要的，因为这个构建器的执行已经失败，我们不希望调用方法来假设对象已正确创建以及有效。
在这个例子中，没有采用前述的标志技术，finally 从句显然不是关闭文件的正确地方，因为这可能在每次构建器结束的时候关闭它。由于我们希望文件在 InputFile 对象处于活动状态时一直保持打开状态，所以这样做并不恰当。
getLine()方法会返回一个字串，其中包含了文件中下一行的内容。它调用了 readLine()，后者可能产生一个违例，但那个违例会被捕获，使 getLine()不会再产生任何违例。对违例来说，一项特别的设计问题是决定在这一级完全控制一个违例，还是进行部分控制，并传递相同（或不同）的违例，或者只是简单地传递它。在适当的时候，简单地传递可极大简化我们的编码工作。getLine()方法会变成：

```
String getLine() throws IOException {
return in.readLine();
}
```

但是当然，调用者现在需要对可能产生的任何 IOException 进行控制。
用户使用完毕 InputFile 对象后，必须调用 cleanup()方法，以便释放由 BufferedReader 以及／或者 FileReader 占用的系统资源（如文件句柄）——注释⑥。除非 InputFile 对象使用完毕，而且到了需要弃之不用的时候，否则不应进行清除。大家可能想把这样的机制置入一个 finalize()方法内，但正如第 4 章指出的那样，并非总能保证 finalize()获得正确的调用（即便确定它会调用，也不知道何时开始）。这属于 Java 的一项缺陷——除内存清除之外的所有清除都不会自动进行，所以必须知会客户程序员，告诉他们有责任用 finalize()保证清除工作的正确进行。

⑥：在 C++里，“破坏器”可帮我们控制这一局面。

在 Cleanup.java 中，我们创建了一个 InputFile，用它打开用于创建程序的相同的源文件。同时一次读取该文件的一行内容，而且添加相应的行号。所有违例都会在 main()中被捕获——尽管我们可选择更大的可靠性。
这个示例也向大家展示了为何在本书的这个地方引入违例的概念。违例与 Java 的编程具有很高的集成度，这主要是由于编译器会强制它们。只有知道了如何操作那些违例，才可更进一步地掌握编译器的知识。

9.8 违例匹配
“掷”出一个违例后，违例控制系统会按当初编写的顺序搜索“最接近”的控制器。一旦找到相符的控制器，就认为违例已得到控制，不再进行更多的搜索工作。
在违例和它的控制器之间，并不需要非常精确的匹配。一个衍生类对象可与基础类的一个控制器相配，如下例所示：

435 页上程序

Sneeze 违例会被相符的第一个 catch 从句捕获。当然，这只是第一个。然而，假如我们删除第一个 catch 从句：

435 页下程序

那么剩下的 catch 从句依然能够工作，因为它捕获的是 Sneeze 的基础类。换言之，catch(Annoyance e)能捕获一个 Annoyance 以及从它衍生的任何类。这一点非常重要，因为一旦我们决定为一个方法添加更多的违例，而且它们都是从相同的基础类继承的，那么客户程序员的代码就不需要更改。至少能够假定它们捕获的是基础类。
若将基础类捕获从句置于第一位，试图“屏蔽”衍生类违例，就象下面这样：

436 页上程序

则编译器会产生一条出错消息，因为它发现永远不可能抵达 Sneeze 捕获从句。

9.8.1 违例准则
用违例做下面这些事情：
(1) 解决问题并再次调用造成违例的方法。
(2) 平息事态的发展，并在不重新尝试方法的前提下继续。
(3) 计算另一些结果，而不是希望方法产生的结果。
(4) 在当前环境中尽可能解决问题，以及将相同的违例重新“掷”出一个更高级的环境。
(5) 在当前环境中尽可能解决问题，以及将不同的违例重新“掷”出一个更高级的环境。
(6) 中止程序执行。
(7) 简化编码。若违例方案使事情变得更加复杂，那就会令人非常烦恼，不如不用。
(8) 使自己的库和程序变得更加安全。这既是一种“短期投资”（便于调试），也是一种“长期投资”（改善应用程序的健壮性）

9.9 总结

通过先进的错误纠正与恢复机制，我们可以有效地增强代码的健壮程度。对我们编写的每个程序来说，错误恢复都属于一个基本的考虑目标。它在 Java 中显得尤为重要，因为该语言的一个目标就是创建不同的程序组件，以便其他用户（客户程序员）使用。为构建一套健壮的系统，每个组件都必须非常健壮。

在 Java 里，违例控制的目的是使用尽可能精简的代码创建大型、可靠的应用程序，同时排除程序里那些不能控制的错误。

违例的概念很难掌握。但只有很好地运用它，才可使自己的项目立即获得显著的收益。Java 强迫遵守违例所有方面的问题，所以无论库设计者还是客户程序员，都能够连续一致地使用它。

9.10 练习

(1) 用 main()创建一个类，令其掷出 try 块内的 Exception 类的一个对象。为 Exception 的构建器赋予一个字串参数。在 catch 从句内捕获违例，并打印出字串参数。添加一个 finally 从句，并打印一条消息，证明自己真正到达那里。

(2) 用 extends 关键字创建自己的违例类。为这个类写一个构建器，令其采用 String 参数，并随同 String 句柄把它保存到对象内。写一个方法，令其打印出保存下来的 String。创建一个 try-catch 从句，练习实际操作新违例。

(3) 写一个类，并令一个方法掷出在练习 2 中创建的类型的一个违例。试着在没有违例规范的前提下编译它，观察编译器会报告什么。接着添加适当的违例规范。在一个 try-catch 从句中尝试自己的类以及它的违例。

(4) 在第 5 章，找到调用了 Assert.java 的两个程序，并修改它们，令其掷出自己的违例类型，而不是打印到 System.err。该违例应是扩展了 RuntimeException 的一个内部类。