9:用异常来处理错误

Java 的基本哲学是“糟糕的代码根本就得不到执行”。

捕捉错误的最佳时机应该是在编译的时候，也就是程序能运行之前。但是，不是所有的错误都能在编译的时候被发现。有些问题只能到程序运行的时候才能得到处理。它们要通过某种方式，让引发问题的代码将适当的信息传给那些知道该怎样正确处理这些问题的程序。

C 以及其它早期语言通常都有多个错误处理的方案，一般来说它们都是建立在约定的基础上的，其本身并不属于语言。最常见的做法是，返回一个特殊的值或设定一个标志，然后希望接收方会看到这个值或标记，并以此判断是不是出了什么问题。但是，随着时间的推移，人们发现，那些使用着类库的程序员们通常都认为自己是刀枪不入的——他们的心态可以归纳为“错误，那时别人的事，我的代码绝不可能有错。”所以，也就不奇怪他们为什么不检查错误条件了(况且有些错误也实在是蠢了点[40]，以至于你都想不到还要去检查)。而且，如果你**真的**每次调用方法的时候，都作一遍完整的错误检查的话，那代码也就没法读了。由于程序员们还能用这些语言凑出一些系统，因此他们一直拒绝承认这样一个事实：这种错误处理的方法已经成为创建大型的、强壮的、可维护的程序的巨大障碍了。

解决的方案就是，强化错误处理的规范性屏弃其原有的随意性。实际上，这种做法已经有很长的历史了。因为异常处理(*exception handling*)的实现最早可以追溯到 1960 年代的操作系统，甚至是 BASIC 的“**on error goto**”。不过 C++的异常处理源于 Ada，而 Java 的则主要建立在 C++的基础之上(尽管看上去更像 Object Pascal 的)。

“exception”这个词的意思有“我对此表示反对”的意味。问题发生的时候，可能你不知道该如何处理，但是你知道不能再没心没肺的运行下去了；你必须停下来，自然会有人知道该如何处理，而处理方案也应该藏在什么地方。但是在现有的条件下，你还不知道问题的详细情况，因此没法解决这个问题。所以你应该把问题交上去，那里会有人来作决定的(就像是行政命令，一级管一级)。

异常还有一个非常重要的好处，就是让错误处理代码显得更有条理。与原先“程序要在多个地方检查同一个错误，并就地作处理”相比，现在，你无需再在方法调用的时候作检查了(因为异常肯定能被捕获)。而且处理问题的代码也集中到一个被称作“异常处理程序(*exception handler*)”地方了。这种做法能为你省下不少代码，而且还把“讲述要作什么的代码”，同“出错的时候该执行的代码”分开来了。总之，无论是写代码还是调试程序，使用异常都比用老式的错误处理更有条理。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

由于“异常处理”是 Java 唯一的正式报告错误的方式，而且 Java 编译器还对此作了强制要求，因此即使不学异常处理，也可以写出很多本书中的例程。本章会介绍如何正确地处理异常以及，如何定义你自己的异常，以供方法出问题的时候使用。

基本异常

“异常条件(*exceptional condition*)”是一种能阻止正在运行的方法或其某一部分继续运行下去的问题。把异常条件同普通问题区分开来，这点很重要。遇到普通问题的时候，你在当前的运行环境下有足够的信息来处理这个困难。对于异常条件，由于你得不到足够的用以处理这个问题的信息，因此不能在“当前的运行环境下(*in the current context*)”继续运行下去。你只能跳出当前的运行环境，并且把问题交到上层的运行环境。这就是抛出异常的时所发生的事情。

除法就是一个例子。如果知道会除以零的话，还是应该去测试一下的。但是被除数是零又代表什么呢？或许你知道。你可以根据问题的上下文，决定如何处理零作分母的情况。但是如果这不是一个预料中的值，因此你不知道该如何处理的话，那就必须抛出一个异常，而不是顺着执行的路径继续下去了。

当你抛出异常的时候，有几件事会随之发生。首先要像创建其它 Java 对象那样，创建一个异常对象：在堆里，用 **new**。然后停下当前的执行路径(这条路是不能再继续下去了)，再将异常对象的 reference 从当前运行环境当中弹出去。现在异常处理机制开始接管程序了，它会去找到一个合适的地方来继续执行这个程序。这个地方就是“异常处理程序(*exception handler*)”，其功能就是将程序从问题中恢复过来，于是程序或者是尝试去换一条运行路径，或者是继续运行下去。

举一个抛异常的简单例子，看看 **t** 这个 reference。它可能还没有经过初始化，因此你应该先检查一下再用它去调用方法。你可以创建一个表示错误信息的对象，并把这个对象“抛出”当前的运行环境，这样就能把出错的信息传递到更高层的运行环境中了。这被称为“抛出一个异常(*throwing an exception*)。”就像这里看到的：

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

这样就抛出了异常，于是你就——在当前运行环境下——无需再为这个问题操心了。自然会有人来处理的。我们这就把他介绍给你。

异常的参数

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

就像 Java 中的其它对象，你也可以用 **new** 在堆中创建异常，而 **new** 会调用它的构造函数并为它分配内存。所有的标准异常都有两个构造函数；第一个是默认的构造函数，第二个是要拿一个字符串当参数的，因此你可以在异常中放入一些相关的信息：

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

就像你将看到的，将来这个字符串可以用各种方法提取出来。

关键词 **throw** 会引发许多很神奇的事情。通常情况下，你会先用 **new** 来创建一个表示错误条件的对象。 然后再把这个对象的 reference 交给 **throw**。虽然这个对象不是方法设计要返回的那种对象，但实际上，方法还是返回这个对象。有一种理解异常处理的简化思路，就是把它想成一种不同的返回机制。但是这种想法别走得太远，否则就有麻烦了。你也可以用“抛异常”的方法从方法的作用域里退出。总之，它会返回了一个值，并且退出了方法或作用域。

异常处理与“从方法中正常返回”的相同点就到此为止了，因为它所返回的地点同正常的方法调用所返回的地点是完全不同的。(它最终是要在某个异常处理程序里面得到解决的，而这个程序可能离异常发生的地方很远——与“调用栈(call stack)”隔着很多层。)

此外，你还可以抛出任何 **Throwable** 对象(这是异常的根类)。通常情况下，你得根据不同的错误抛出不同的异常。错误信息由保存在异常对象中信息，以及异常类的名字表示。于是上层的运行环境就可以用这个异常来决定该怎么干了。(通常，异常类型的名字是唯一的信息，异常对象不会保存什么有用的东西。)

捕捉异常

如果方法抛出了异常，那么必须要有能“捕捉”这个异常，并且处理这个异常的程序。异常处理有一个好处，就是它能让你集中精力在一个地方解决问题，然后将处理错误的代码分开来放在另一个地方。

要想理解异常是怎样被捕捉到的，你必须首先懂得“守护区域(*guarded region*)”的概念。这是一段可能会产生异常的代码，并且后面还跟着要处理这些异常的代码。

Try 区块

如果你从方法里面抛出了一个异常(或是在这个方法调用的另一个方法里面抛出一个异常)，那么抛出异常的同时，这个方法会退出运行。如果你不想被 **throw** 出方法，那么你可以在这个方法的内部建一个特殊的区块

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

来捕获这个异常。这就被称为“*try* 区块(*try block*)”，因为你在这里“try”各种方法调用。try 区块是跟在 **try** 关键词后面的程序块。

```
try {
  // Code that might generate exceptions
}
```

如果你想在不支持异常处理的编程语言里仔细地检查错误，那么每次调用方法的时候，你都得在调用代码的外围放上测试错误条件的代码，即便是在重复调用同一个方法的时候也得这么做。使用异常处理的时候，你把所有的东西放进 try 区块，然后在一个地方捕获所有的异常。这就是说，完成任务的代码和处理错误的代码不会再搅在一起了，因此代码变得更易读易写了。

异常处理程序

当然，抛出来的异常必须要在什么地方得到处理。这个“地方”就是“异常处理程序(*exception handler*)”。此外，你想捉一个异常就要准备一段异常处理程序。异常处理程序会直接跟在 try 区块后面，用 **catch** 关键词表示：

```
try {
  // Code that might generate exceptions
} catch(Type1 id1) {
  // Handle exceptions of Type1
} catch(Type2 id2) {
  // Handle exceptions of Type2
} catch(Type3 id3) {
  // Handle exceptions of Type3
}

// etc...
```

每条 catch 子句(异常处理程序)都像一个微型的，并且有一个且仅有一个特定类型的参数的方法。异常处理程序可以像使用方法的参数那样使用这些标识符(**id1**，**id2** 等等)。有时，由于异常的类型已经给出了足够的信息，因此你根本就用不到这些标识符，但是它还必须搁在那里。

异常处理程序必须直接跟在 try 区块后面。如果程序抛出了异常，异常处理机制就会进来寻找其参数与异常类型相匹配的那些异常处理程序中的第一个。然后进入那条 catch 子句，于是它就认为异常已经得到处理了。一旦 catch 子句结束，寻找异常处理程序的任务就会停下。相匹配的 catch 子句会得到执行；这点不像 **switch** 语句，它得在每个 **case** 后面加一个 **break**，否则后面指令也会被执行。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

注意，在 try 区块中，有可能会碰到“调用多个不同的方法会产生同一种异常的”情况，但是这时你只需要一个异常处理程序。

“中止”还是“继续”

理论上将异常处理划分成两种基本模型。中止模型(*temination*，也就是 Java 和 C++所采纳的那种)假定错误是如此的严重，以致于你没办法再回到错误发生的地方。也就是说，这段程序经过判断认为，它已经没有办法再挽回这个局势了，于是只能抛出异常，并且希望这个错误别再回来。

还有一种被称为“继续(*resumption*)”。它的意思是，异常处理程序应该能作些什么以修补当前的运行环境，然后重新尝试上次出错的那个方法，它假设第二次能获得成功。“继续”的意思是，处理完异常之后，你仍然希望能继续运行当前的指令。在这种情况下，异常更像是在调用方法——如果你想在 Java 中得到类似的效果，可以用这个办法来设置运行环境。(也就是别抛出异常了；调用一个方法来解决这个问题。)此外，还可以把 **try** 区块放到 **while** 循环里面，这样就会重复的运行 **try** 区块，直到你得到满意的结果。

长久以来，程序员所使用的操作系统都是用“继续模式”来处理异常的，但是最终他们写代码的时候，都会跳开“继续模式”而采用“中止模式”。所以尽管初听上去，“继续模式”很具吸引力，但是实际上并不是那么实用。最主要的原因恐怕还是它所导致的耦合(*coupling*)；通常异常处理程序必须知道异常是从哪里抛出的，并且还要包括专门针对异常抛出位置的非泛型代码。这使得代码非常难写，也无法维护。碰上会随时产生异常的大型问题的时候，更是如此。

创建你自己的异常

Java 没有限定你只能使用它提供的异常。JDK 的异常体系不可能预见到所有你要报告的异常，所以你可以用你自己创建的异常来表示你的类库可能会遇到的问题。

要想创建你自己的异常，你必须继承已有的异常，最好是意思上同你要创建的那个新的异常比较相近的那个(虽然不是经常能做到这一点)。创建一个新的异常类的最简单的做法是直接让编译器为你生成一个默认的构造函数，这样做几乎不需要写任何代码：

```
//: c09:SimpleExceptionDemo.java
// Inheriting your own exceptions.
import com.bruceeckel.simpletest.*;

class SimpleException extends Exception {}

public class SimpleExceptionDemo {
  private static Test monitor = new Test();
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
  public void f() throws SimpleException {
    System.out.println("Throw SimpleException from
f()");
    throw new SimpleException();
  }
  public static void main(String[] args) {
    SimpleExceptionDemo sed = new
SimpleExceptionDemo();
    try {
      sed.f();
    } catch(SimpleException e) {
      System.err.println("Caught it!");
    }
    monitor.expect(new String[] {
      "Throw SimpleException from f()",
      "Caught it!"
    });
  }
} ///:~
```

编译器创建了一个默认的构造函数，它会自动地(并且用一种你看不到的方式)调用基类的默认构造函数。当然，在这个例子中**SimpleException(String)**的构造函数就没有了，但实际上它用的也不多。正如你将看到的，对异常来说，最重要的是这个类的名字，所以绝大多数时候这个异常也就够用了。

这里，会用 **System.err** 把结果打印到控制台的标准错误流(*standard error*)。对于错误信息来说，送到这里通常会比送到 **System.out** 要好，因为后者可能会被重定向。如果你将输出送到 **System.err**，那么即使是在 **System.out** 被重定向的时候，它还是会被打印在屏幕上的，这样就能更好地引起用户的注意。

你也可以在你创建的的异常类里定义以 **String** 作参数的构造函数。

```
//: c09:FullConstructors.java
import com.bruceeckel.simpletest.*;

class MyException extends Exception {
  public MyException() {}
  public MyException(String msg) { super(msg); }
}

public class FullConstructors {
  private static Test monitor = new Test();
  public static void f() throws MyException {
    System.out.println("Throwing MyException from
f()");
    throw new MyException();
  }
  public static void g() throws MyException {
    System.out.println("Throwing MyException from
g()");
    throw new MyException("Originated in g()");
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
  }
  public static void main(String[] args) {
    try {
      f();
    } catch(MyException e) {
      e.printStackTrace();
    }
    try {
      g();
    } catch(MyException e) {
      e.printStackTrace();
    }
    monitor.expect(new String[] {
      "Throwing MyException from f()",
      "MyException",
      "%% \tat FullConstructors.f\\(.*\\)",
      "%% \tat FullConstructors.main\\(.*\\)",
      "Throwing MyException from g()",
      "MyException: Originated in g()",
      "%% \tat FullConstructors.g\\(.*\\)",
      "%% \tat FullConstructors.main\\(.*\\)"
    });
  }
} ///:~
```

新增的代码很少：**MyException** 有两个构造函数。第二个通过 **super** 关键词，明确地用一个 **String** 参数调用了基类的构造函数。

处理程序调用了 **Throwable** 类(**Exception** 是从它那里继承的)的 **printStackTrace( )**方法。这个方法会返回“被调用的方法是经过怎样一个顺序到达异常发生地点”的信息。缺省情况下，这些信息会送到标准错误流，但是这个方法的重载版也允许你将结果送到其它流。

创建自定义的异常还能更进一步。你还可以添进额外的构造函数和成员：

```
//: c09:ExtraFeatures.java
// Further embellishment of exception classes.
import com.bruceeckel.simpletest.*;

class MyException2 extends Exception {
  private int x;
  public MyException2() {}
  public MyException2(String msg) { super(msg); }
  public MyException2(String msg, int x) {
    super(msg);
    this.x = x;
  }
  public int val() { return x; }
  public String getMessage() {
    return "Detail Message: "+ x + " "+
super.getMessage();
  }
}
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
public class ExtraFeatures {
  private static Test monitor = new Test();
  public static void f() throws MyException2 {
    System.out.println("Throwing MyException2 from
f()");
    throw new MyException2();
  }
  public static void g() throws MyException2 {
    System.out.println("Throwing MyException2 from
g()");
    throw new MyException2("Originated in g()");
  }
  public static void h() throws MyException2 {
    System.out.println("Throwing MyException2 from
h()");
    throw new MyException2("Originated in h()", 47);
  }
  public static void main(String[] args) {
    try {
      f();
    } catch(MyException2 e) {
      e.printStackTrace();
    }
    try {
      g();
    } catch(MyException2 e) {
      e.printStackTrace();
    }
    try {
      h();
    } catch(MyException2 e) {
      e.printStackTrace();
      System.err.println("e.val() = " + e.val());
    }
    monitor.expect(new String[] {
      "Throwing MyException2 from f()",
      "MyException2: Detail Message: 0 null",
      "%% \tat ExtraFeatures.f\\(.*\\)",
      "%% \tat ExtraFeatures.main\\(.*\\)",
      "Throwing MyException2 from g()",
      "MyException2: Detail Message: 0 Originated in
g()",
      "%% \tat ExtraFeatures.g\\(.*\\)",
      "%% \tat ExtraFeatures.main\\(.*\\)",
      "Throwing MyException2 from h()",
      "MyException2: Detail Message: 47 Originated
in h()",
      "%% \tat ExtraFeatures.h\\(.*\\)",
      "%% \tat ExtraFeatures.main\\(.*\\)",
      "e.val() = 47"
    });
  }
} ///:~
```

这个异常加了一个数据成员 **i**，一个能读取这个数据的方法和能设定这个数据的构造函数。此外为了能产生一些更有意思的详细信息，它还覆写了

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

Throwable.getMessage( )方法。对于异常类来说，**getMessage( )**有点像 **toString( )**。

由于异常同其它对象没什么两样，所以你可以不断地用这种方法来增强这种自定义的异常类。但是要记住，当你把这些 package 交给客户程序员的时候，他们很可能会把这些“浇头”全给扔掉，因为他们可能只是单纯地找到这个抛出的异常，其它什么都不管。(绝大多数的 Java 类库中的异常都是这么用的。)

异常说明

Java 鼓励你告诉那些调用你方法的客户程序员们，你定义的方法可能会抛出哪种异常。这是一种很友好的做法，因为这么一来，调用方法的人就能知道应该写什么样的代码来捕捉潜在的异常了。当然，如果可以得到源代码的话，客户程序员可以通过查找 **throw** 语句来发现这点，但通常情况下，他们是得不到类库的源代码的。为了应付这种情况，Java 在语法上提供了支持(同时也是强制你必须使用这种语法)，能让你很礼貌地告诉客户程序员们，这个方法能抛出什么异常，这样客户程序员就能自行处理了。这就是“异常说明(*exception specification*)”，它属于方法声明的一部分，要列在在参数表的后面。

异常说明要用 **throws** 关键词，后面再跟上所有可能抛出的异常类型的清单。所以方法的定义可能会是这样：

```
void f() throws TooBig, TooSmall, DivZero { //...
```

如果是

```
void f() { // ...
```

这就意味着这个方法不会抛出任何异常(除非是继承自 **RuntimeException** 的异常，这类异常不需要异常说明就可以从任何地方抛出——这一点我们以后会讲)。

你不能对异常说明撒谎。如果你的方法产生一个异常但又不作处理，那么编译器就会提醒你，要么处理这个异常，要么在异常说明部分指明这个方法会抛出这个异常。通过强化这种从顶层到底层的异常说明，Java 提供了一定程度的在编译时纠正异常的保障。

不过还是有个能撒谎的地方：你可以声称方法会抛出一个实际上并不会抛出的异常。编译器会当真的，它会要求客户程序员像真的会抛出异常那样

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

使用这个方法。这么作的好处就是，它能为异常先占个位子，以后你就能真的抛出异常而不用去改动已有的代码了。而且对于其实现会真的抛出异常的 **abstract** 基类和 **interface** 来说，这是相当重要的。

会在编译时进行检查并且强制得到处理的异常被称为 *checked exception*。

捕捉任意类型的异常

要想创建一个“能处理任意类型的异常的”异常处理程序是完全可能的。要做到这点，你就必须去捕捉异常的基类 **Exception** 了 (还有一些其它类型的基类，但实际上只有 **Exception** 是同编程活动相关的基类)：

```
catch(Exception e) {
  System.err.println("Caught an exception");
}
```

这样就能捕获所有的异常了，所以如果你想这么写的话，就应该把它放到最后，这样它就不会抢在其它处理程序前头把异常给劫走了。

由于 **Exception** 只是那些跟程序员有关的异常类的基类，因此你不能从它那里得到更多的关于异常的具体信息，不过你也可以调用 **Exception** 的基类 **Throwable** 的方法：

String getMessage( )
String getLocalizedMessage( )
获取详细信息，或者是用当地语言表示的信息。

String toString( )
返回包括详细说明在内的，这个 **Throwable** 的简短说明。如果有的话。

void printStackTrace( )
void printStackTrace(PrintStream)
void printStackTrace(java.io.PrintWriter)
打印 **Throwable** 以及 **Throwable** 的调用栈轨迹(call stack trace)。 调用栈显示了“将你带到异常发生地的”方法调用的顺序。第一个版本会打印到标准错误上，第二和第三个则会打印到你选择的流里(到第 12 章，你就会知道为什么要有这两种流)。

Throwable fillInStackTrace( )
在 **Throwable** 对象里面记录“栈帧(stack frame)”的当前状态。等应用程序要重抛错误或异常的时候就要用这些信息了(过一会再作详细讲解)。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

此外你还可以用 **Throwable** 的基类，也就是 **Object**(所有对象的基类)的方法。对异常来说有一个很有用的方法，**getClass( )**。它会返回一个“表示这个对象是属于哪种类型的”**Class** 对象。接下来，你可以用 **getName( )**查询这个 **Class** 对象的名字。你还可以用 **Class** 对象作更复杂的事情，不过对异常处理来说，就没这个必要了。

下面这段程序演示了这些基本的 **Exception** 方法的用法：

```
//: c09:ExceptionMethods.java
// Demonstrating the Exception Methods.
import com.bruceeckel.simpletest.*;

public class ExceptionMethods {
  private static Test monitor = new Test();
  public static void main(String[] args) {
    try {
      throw new Exception("My Exception");
    } catch(Exception e) {
      System.err.println("Caught Exception");
      System.err.println("getMessage():" +
e.getMessage());
      System.err.println("getLocalizedMessage():" +
        e.getLocalizedMessage());
      System.err.println("toString():" + e);
      System.err.println("printStackTrace():");
      e.printStackTrace();
    }
    monitor.expect(new String[] {
      "Caught Exception",
      "getMessage():My Exception",
      "getLocalizedMessage():My Exception",
      "toString():java.lang.Exception: My Exception",
      "printStackTrace():",
      "java.lang.Exception: My Exception",
      "%% \tat ExceptionMethods.main\\(.*\\)"
    });
  }
} ///:~
```

你能看到这些方法一个比一个提供更多的信息——实际上它们每个都是前一个的超集。

重抛异常

有时你需要重新抛出那个刚捕捉到的，用 **Exception** 捕捉到的异常。由于你已经有了当前异常的 reference，因此你可以直接将那个 reference 重抛出来：

```
catch(Exception e) {
  System.err.println("An exception was thrown");
  throw e;
}
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

重抛会把异常送到更高一层的异常处理程序去。同一个 **try** 区块的其它 **catch** 子句都将被忽略。此外，它还会保留异常对象里的所有信息，这样捕获这个异常的上一层的异常处理程序就能够提取这个对象中的所有信息了。

如果你直接重抛当前的异常，则 **printStackTrace( )**所打印出来的那些保存在异常对象里的信息，还会指向异常发生的地方，它们不会被指到你重抛异常的地点。如果你要装载新的栈轨迹信息，你可以调用 **fillInStackTrace( )**，这个方法会将当前栈的信息塞进旧的异常对象中，并返回一个 **Throwable** 对象。下面就是它的工作方式：

```
//: c09:Rethrowing.java
// Demonstrating fillInStackTrace()
import com.bruceeckel.simpletest.*;

public class Rethrowing {
  private static Test monitor = new Test();
  public static void f() throws Exception {
    System.out.println("originating the exception in
f()");
    throw new Exception("thrown from f()");
  }
  public static void g() throws Throwable {
    try {
      f();
    } catch(Exception e) {
      System.err.println("Inside
g(),e.printStackTrace()");
      e.printStackTrace();
      throw e; // 17
      // throw e.fillInStackTrace(); // 18
    }
  }
  public static void
  main(String[] args) throws Throwable {
    try {
      g();
    } catch(Exception e) {
      System.err.println(
        "Caught in main, e.printStackTrace()");
      e.printStackTrace();
    }
    monitor.expect(new String[] {
      "originating the exception in f()",
      "Inside g(),e.printStackTrace()",
      "java.lang.Exception: thrown from f()",
      "%% \tat Rethrowing.f(.*?)",
      "%% \tat Rethrowing.g(.*?)",
      "%% \tat Rethrowing.main(.*?)",
      "Caught in main, e.printStackTrace()",
      "java.lang.Exception: thrown from f()",
      "%% \tat Rethrowing.f(.*?)",
      "%% \tat Rethrowing.g(.*?)",
      "%% \tat Rethrowing.main(.*?)"
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
    });
  }
} ///:~
```

最重要的几行都已经标出来了。如果第 17 行没有注释掉(也就是现在的情况)，那么无论这个异常被重抛了多少次，异常的栈轨迹都会一直记得它是从最初那个抛出的。

如果把第 17 行注释掉，并且把第 18 行注释回来，那么就会用到 **fillInStackTrace( )**，结果是：

```
originating the exception in f()
Inside g(),e.printStackTrace()
java.lang.Exception: thrown from f()
        at Rethrowing.f(Rethrowing.java:9)
        at Rethrowing.g(Rethrowing.java:12)
        at Rethrowing.main(Rethrowing.java:23)
Caught in main, e.printStackTrace()
java.lang.Exception: thrown from f()
        at Rethrowing.g(Rethrowing.java:18)
        at Rethrowing.main(Rethrowing.java:23)
```

(此外还有一些 **Text.expect( )**输出的出错信息。)由于 **fillInStackTrace( )**，第 18 行成了异常的新的发生地了。

方法 **g( )**和 **main( )**的异常说明中必须包括 **Throwable**，这是因为 **fillInStackTrace( )**方法返回的是 **Throwable** 对象的 reference。而由于 **Throwable** 是 **Exception** 的基类，所以有可能会得到一个 **Throwable** 而不是 **Exception**，因此 **main( )**里面的 **Exception** 的处理程序就有可能会捕捉不到这个重抛出来的对象。为了确保一切都能正常运行，编译器会强制要求在异常说明中使用 **Throwable**。例如，在下面这段程序中的 **main( )**就没能捉到抛出来的异常：

```
//: c09:ThrowOut.java
// {ThrowsException}
public class ThrowOut {
  public static void
  main(String[] args) throws Throwable {
    try {
      throw new Throwable();
    } catch(Exception e) {
      System.err.println("Caught in main()");
    }
  }
} ///:~
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

你也可以抛出一个与你捕捉到的异常不同的异常。这么做的效果同使用**fillInStackTrace( )**的差不多——异常最初在哪里发生的信息被仍了，现在里面保存的是抛出新异常的地点。

```
//: c09:RethrowNew.java
// Rethrow a different object from the one that was
caught.
// {ThrowsException}
import com.bruceeckel.simpletest.*;

class OneException extends Exception {
  public OneException(String s) { super(s); }
}

class TwoException extends Exception {
  public TwoException(String s) { super(s); }
}

public class RethrowNew {
  private static Test monitor = new Test();
  public static void f() throws OneException {
    System.out.println("originating the exception in
f()");
    throw new OneException("thrown from f()");
  }
  public static void
  main(String[] args) throws TwoException {
    try {
      f();
    } catch(OneException e) {
      System.err.println(
        "Caught in main, e.printStackTrace()");
      e.printStackTrace();
      throw new TwoException("from main()");
    }
    monitor.expect(new String[] {
      "originating the exception in f()",
      "Caught in main, e.printStackTrace()",
      "OneException: thrown from f()",
      "\tat RethrowNew.f(RethrowNew.java:18)",
      "\tat RethrowNew.main(RethrowNew.java:22)",
      "Exception in thread \"main\" " +
      "TwoException: from main()",
      "\tat RethrowNew.main(RethrowNew.java:28)"
    });
  }
} ///:~
```

最后那个异常只知道它是从**main( )**而不是**f( )**来的。

你永远也不需要为清理前一个异常或是其它异常而担心。它们都是用**new**在堆里创建的，因此垃圾收集器会把它们自动清理干净。

异常链

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

你经常会碰到在“捉到一个异常并且抛出另一个异常的时候，还要保存前一个异常的信息”的情况——这就是所谓的异常链(*exception chaining*)。在 JDK 1.4 以前，程序员们必须自己写代码来保存的前一个异常的信息，但现在所有的 **Throwable** 的子类都有一个能接受 *cause*(原因)对象的构造函数。这个 *cause* 就是用来保存前一个异常的，这样通过一级一级的传递，即便你在创建并抛出了新的异常，它仍然能维系一个“能追踪到异常的第一现场的”栈轨迹。

有趣的是，在 **Throwable** 的子类中，只有三种基本的异常类提供了带 *cause* 参数的构造函数，它们是 **Error** (供 JVM 报告系统错误之用)，**Exception** 和 **RuntimeException**。如果你要链接其它异常，那就不能用构造函数，而只能用 **initCause( )**方法了。

下面这段程序能让你在运行时动态的向 **DynamicField** 对象里加入数据：

```
//: c09:DynamicFields.java
// A Class that dynamically adds fields to itself.
// Demonstrates exception chaining.
// {ThrowsException}
import com.bruceeckel.simpletest.*;

class DynamicFieldsException extends Exception {}

public class DynamicFields {
  private static Test monitor = new Test();
  private Object[][] fields;
  public DynamicFields(int initialSize) {
    fields = new Object[initialSize][2];
    for(int i = 0; i < initialSize; i++)
      fields[i] = new Object[] { null, null };
  }
  public String toString() {
    StringBuffer result = new StringBuffer();
    for(int i = 0; i < fields.length; i++) {
      result.append(fields[i][0]);
      result.append(": ");
      result.append(fields[i][1]);
      result.append("\n");
    }
    return result.toString();
  }
  private int hasField(String id) {
    for(int i = 0; i < fields.length; i++)
      if(id.equals(fields[i][0]))
        return i;
    return -1;
  }
  private int
  getFieldNumber(String id) throws
NoSuchFieldException {
    int fieldNum = hasField(id);
    if(fieldNum == -1)
      throw new NoSuchFieldException();
    return fieldNum;
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
  }
  private int makeField(String id) {
    for(int i = 0; i < fields.length; i++)
      if(fields[i][0] == null) {
        fields[i][0] = id;
        return i;
      }
    // No empty fields. Add one:
    Object[][]tmp = new Object[fields.length + 1][2];
    for(int i = 0; i < fields.length; i++)
      tmp[i] = fields[i];
    for(int i = fields.length; i < tmp.length; i++)
      tmp[i] = new Object[] { null, null };
    fields = tmp;
    // Reursive call with expanded fields:
    return makeField(id);
  }
  public Object
  getField(String id) throws NoSuchFieldException {
    return fields[getFieldNumber(id)][1];
  }
  public Object setField(String id, Object value)
  throws DynamicFieldsException {
    if(value == null) {
      // Most exceptions don't have a "cause"
constructor.
      // In these cases you must use initCause(),
      // available in all Throwable subclasses.
      DynamicFieldsException dfe =
        new DynamicFieldsException();
      dfe.initCause(new NullPointerException());
      throw dfe;
    }
    int fieldNumber = hasField(id);
    if(fieldNumber == -1)
      fieldNumber = makeField(id);
    Object result = null;
    try {
      result = getField(id); // Get old value
    } catch(NoSuchFieldException e) {
      // Use constructor that takes "cause":
      throw new RuntimeException(e);
    }
    fields[fieldNumber][1] = value;
    return result;
  }
  public static void main(String[] args) {
    DynamicFields df = new DynamicFields(3);
    System.out.println(df);
    try {
      df.setField("d", "A value for d");
      df.setField("number", new Integer(47));
      df.setField("number2", new Integer(48));
      System.out.println(df);
      df.setField("d", "A new value for d");
      df.setField("number3", new Integer(11));
      System.out.println(df);
      System.out.println(df.getField("d"));
      Object field = df.getField("a3"); // Exception
    } catch(NoSuchFieldException e) {
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
      throw new RuntimeException(e);
    } catch(DynamicFieldsException e) {
      throw new RuntimeException(e);
    }
    monitor.expect(new String[] {
      "null: null",
      "null: null",
      "null: null",
      "",
      "d: A value for d",
      "number: 47",
      "number2: 48",
      "",
      "d: A new value for d",
      "number: 47",
      "number2: 48",
      "number3: 11",
      "",
      "A value for d",
      "Exception in thread \"main\" " +
      "java.lang.RuntimeException: " +
      "java.lang.NoSuchFieldException",
      "\tat
DynamicFields.main(DynamicFields.java:98)",
      "Caused by: java.lang.NoSuchFieldException",
      "\tat DynamicFields.getFieldNumber(" +
      "DynamicFields.java:37)",
      "\tat
DynamicFields.getField(DynamicFields.java:58)",
      "\tat
DynamicFields.main(DynamicFields.java:96)"
    });
  }
} ///:~
```

每个 **DynamicFields** 对象里面都有一个 **Object-Object** 对的数组。第一个对象是成员数据的标识符(一个 **String**)，而第二个则是这个成员数据的值，这个值可以是除 primitive 之外的任何东西。当你创建这个对象的时候，你要估计一下它大概会有多少成员。当你调用 **setField( )**的时候，它要么根据你提供的名字重新设置那个数据，要么创建一个新的成员并把你设置的值放进去。如果空间用完了，它会这样扩充空间：先创建一个新的比原先那个数组多一个元素的数组，然后把原先那个数组中的元素全部拷贝进新的数组。如果你试图放入一个 **null**，那它就会创建并抛出一个 **DynamicFieldsException**，然后使用 **initCause( )**将 **NullPointerException** 当作 cause 插进去。

setField( )还会调用 **getField( )**提取并返回成员数据的值，而 **getField( )**有可能会抛出一个 **NoSuchFieldException**。如果客户程序员调用 **getField( )**，那么他必须负责处理 **NoSuchFieldException**，但是如果这个异常是从 **setField( )**抛出

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

的，那这就是一个编程错误了，所以它会用拿 cause 当参数的构造函数将 **NoSuchFieldException** 转化成 **RuntimeException**。

标准 Java 异常

Java 的 **Throwable** 类概括了所有能被当作异常抛出来的东西。**Throwable** 主要分成两类(分成两类的意思就是从它那里继承两个类)。**Error** 表示编译时错误和系统错误，这些错误都不需要你去费心捕捉(除非是特殊情况)。**Exception** 是“所有 Java 标准类库的方法、你自定义的方法、以及程序运行发生意外时能抛出的异常”的基类。因此 Java 程序员们最感兴趣的还是 **Exception**。

要想对异常有个大致的了解，最好的办法是浏览 Java 的 HTML 文档。看一遍还是值得的，这样就会对各种异常有个感性认识，但是次数多了你就会发现，异常同异常之间除了名字之外没什么不同。此外，Java 中异常的数量仍然在增长之中；在书里罗列这些异常也是毫无意义的。而且你拿到的第三方的新类库里，大概也有它自己的一套异常。对异常来说，最重要的就是理解它的概念，以及应该怎样运用异常。

基本意思是用异常的名字来表示出了什么问题，所以异常的名字又都多少可以自己解释自己。异常并不是都在 **java.lang** 里面定义的；有些异常是专门为其它类库，像 **util**，**net**，和 **io**，服务的。这些异常可以从它们的类名全称或是其继承体系里面看出端倪。比如所有的 I/O 异常都是继承自 **java.io.IOException**。

RuntimeException 的特例

本章的第一个例子就是

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

如果你必须检查每一个传给方法的 reference，判断它是不是 **null** 的(由于你不知道调用方法的那一方传给你的是不是一个有效的 reference)，那可真有点吓人了。幸运的是，你不必这么做——Java 会用它的标准运行时检查系统为你搞定这类问题，如果有人用 **null** 的 reference 进行了调用，那么 Java 会自动抛出一个 **NullPointerException**。所以上述的代码实际上是多余的。

RuntimeException 下面包括了好多种异常。它们都是由 Java 自动抛出的，因此你无需在异常说明中把它们列出来。它们都是继承 **RuntimeException** 这个基类的，因此用起来很方便，而且这也是一个绝好的运用继承的例子；它创建了一族具有相同特征和行为的类。而

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

且，由于它们都是 *unchecked exception*，因此你也无需在异常说明中列出 **RuntimeException**(或者是任何继承 **RuntimeException** 的异常)。由于它们表示 bug，因此通常情况下，你用不着去捕捉 **RuntimeException**——它会自动得到处理。如果要让你去检查 **RuntimeException** 的话，那代码就会变得太乱了。虽然你一般不会去捕捉 **RuntimeException**，但当你自己创建 package 的时候，可能还是会抛出一些 **RuntimeException** 的。

你不捕捉这类异常的时候又发生了些什么呢？由于没有异常说明，编译器也不强制检查，所以 **RuntimeException** 应该会不受阻挡地一路闯到 **main( )**。如果想看看这种情况下都发生了些什么，那么就试试下面这段程序：

```
//: c09:NeverCaught.java
// Ignoring RuntimeExceptions.
// {ThrowsException}
import com.bruceeckel.simpletest.*;

public class NeverCaught {
  private static Test monitor = new Test();
  static void f() {
    throw new RuntimeException("From f()");
  }
  static void g() {
    f();
  }
  public static void main(String[] args) {
    g();
    monitor.expect(new String[] {
      "Exception in thread \"main\" " +
      "java.lang.RuntimeException: From f()",
      "         at NeverCaught.f(NeverCaught.java:7)",
      "         at
NeverCaught.g(NeverCaught.java:10)",
      "         at
NeverCaught.main(NeverCaught.java:13)"
    });
  }
} ///:~
```

你也看到了 **RuntimeException**(或是由它派生的)都是一些特例，因为编译器不会要求你对这些类型的异常作说明的。

所以答案就是：如果 **RuntimeException** 不受阻拦的冲到 **main( )**，它就会在程序退出的时候，调用 **printStackTrace( )**。

记住，由于编程的时候，编译器会强制要求你写出所有其它的处理程序，所以你只能忽略 **RuntimeException**(及其子类)。原因就是 **RuntimeException** 表示编程错误：

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

1. 一种你无法预料的错误。比如不在你控制之内的 **null** reference。
2. 一种由于“程序员忘了检查它应该检查的错误条件”而造成的错误(比如 **ArrayIndexOutOfBoundsException**，你访问数组的时候应该对数组的大小做一个检查**)**。第一种情况下发生的异常，经常会演变成第二种情况下的问题。

你会看到这种处理异常的好处的，在调试过程中，它能帮上很大的忙。

值得提请你注意的是，你不能把 Java 的异常处理当作一种功能单一的工具。是的，它的设计初衷是用来处理那些“不在你控制范围内的因素所造成的”，非常麻烦的运行时错误，但是对那些编译器无法发觉的编程错误，它也是非常重要的。

用 **finally** 进行清理

你会时常碰到一些“无论 **try** 区块有没有抛出异常，程序都必须执行的代码”。一般来说这些代码都是进行恢复内存之外的操作的(因为恢复内存是垃圾回收器的活)。要达到这个效果，你可以在异常处理程序之后使用 **finally** 子句[41]。于是异常处理的完整的句法就是：

```
try {
  // The guarded region: Dangerous activities
  // that might throw A, B, or C
} catch(A a1) {
  // Handler for situation A
} catch(B b1) {
  // Handler for situation B
} catch(C c1) {
  // Handler for situation C
} finally {
  // Activities that happen every time
}
```

要想证明 **finally** 子句总是会得到执行，请运行下面这段程序：

```
//: c09:FinallyWorks.java
// The finally clause is always executed.
import com.bruceeckel.simpletest.*;

class ThreeException extends Exception {}

public class FinallyWorks {
  private static Test monitor = new Test();
  static int count = 0;
  public static void main(String[] args) {
    while(true) {
      try {
        // Post-increment is zero first time:
        if(count++ == 0)
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
            throw new ThreeException();
          System.out.println("No exception");
        } catch(ThreeException e) {
          System.err.println("ThreeException");
        } finally {
          System.err.println("In finally clause");
          if(count == 2) break; // out of "while"
        }
      }
      monitor.expect(new String[] {
        "ThreeException",
        "In finally clause",
        "No exception",
        "In finally clause"
      });
    }
} ///:~
```

你可以从程序的输出看出，无论有没有抛出异常，**finally** 子句总会得到执行。

这个程序也启发我们应该如何处理我们前面讲到过的，Java 异常(和 C++的异常一样)会阻止程序回到异常抛出的地方恢复执行，这个问题。如果你把 **try** 区块放进一个循环，你就能构建一个程序运行之前必须满足的条件了。你也可以在循环里加上 **static** 的计数器，或其它什么东西，让它退出之前多试几种方法。这样你就能把程序的强壮性就能更上一个台阶。

finally 是用来干什么的?

在没有垃圾收集器**并且**不会自动调用拆构函数(destructor) [42]的语言里，**finally** 是非常重要的，因为这样一来不论 **try** 区块都作了些什么，程序员都能保证释放内存了。但是 Java 有垃圾收集器，所以释放内存实际上不是什么问题，更何况它还没有拆构函数可供调用。所以 Java 为什么还要 **finally** 呢？

当你需要把内存**以外**的东西设置到原先状态的时候，**finally** 就显得很有必要了。它可能是用来清理一个打开的文件或网络连接，画在屏幕上的什么东西或者，甚至是下面这个的程序所模拟的程控开关：

```
//: c09:Switch.java
public class Switch {
  private boolean state = false;
  public boolean read() { return state; }
  public void on() { state = true; }
  public void off() { state = false; }
} ///:~
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
//: c09:OnOffException1.java
public class OnOffException1 extends Exception {}
///:~
```

```
//: c09:OnOffException2.java
public class OnOffException2 extends Exception {}
///:~
```

```
//: c09:OnOffSwitch.java
// Why use finally?

public class OnOffSwitch {
  private static Switch sw = new Switch();
  public static void f()
  throws OnOffException1,OnOffException2 {}
  public static void main(String[] args) {
    try {
      sw.on();
      // Code that can throw exceptions...
      f();
      sw.off();
    } catch(OnOffException1 e) {
      System.err.println("OnOffException1");
      sw.off();
    } catch(OnOffException2 e) {
      System.err.println("OnOffException2");
      sw.off();
    }
  }
} ///:~
```

这里的目标是要确保当 **main( )**结束的时候，开关应该处在关闭状态，所以 **sw.off( )**会被放在 try 区块以及每个异常处理程序的末尾。但是有可能会抛出一个没能捕获的异常，所以 **sw.off( )**还是可能会被漏掉。然而用了 **finally** 之后，你就可以把 try 区块里面的清理代码全都集中到一个地方：

```
//: c09:WithFinally.java
// Finally Guarantees cleanup.

public class WithFinally {
  static Switch sw = new Switch();
  public static void main(String[] args) {
    try {
      sw.on();
      // Code that can throw exceptions...
      OnOffSwitch.f();
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
      } catch(OnOffException1 e) {
        System.err.println("OnOffException1");
      } catch(OnOffException2 e) {
        System.err.println("OnOffException2");
      } finally {
        sw.off();
      }
    }
} ///:~
```

现在 **sw.off( )**都被放在了一个地方，而放在这里的代码，无论发生了什么情况都能确保得到执行。

甚至是在“异常没有被当前这组 **catch** 子句所捕获”的情况下，**finally** 也会在“异常处理机制在更高一层的运行环境中开始寻找处理程序”之前得到执行，

```
//: c09:AlwaysFinally.java
// Finally is always executed.
import com.bruceeckel.simpletest.*;

class FourException extends Exception {}

public class AlwaysFinally {
  private static Test monitor = new Test();
  public static void main(String[] args) {
    System.out.println("Entering first try block");
    try {
      System.out.println("Entering second try
block");
      try {
        throw new FourException();
      } finally {
        System.out.println("finally in 2nd try
block");
      }
    } catch(FourException e) {
      System.err.println(
        "Caught FourException in 1st try block");
    } finally {
      System.err.println("finally in 1st try block");
    }
    monitor.expect(new String[] {
      "Entering first try block",
      "Entering second try block",
      "finally in 2nd try block",
      "Caught FourException in 1st try block",
      "finally in 1st try block"
    });
  }
} ///:~
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

在有 **break** 和 **continue** 语句的情况下，**finally** 语句也会得到执行。注意，有了 **finally** 以及带标签的 **break** 和带标签的 **continue** 之后，Java 已经不再需要 **goto** 语句了。

错误：丢失的异常

不幸的是，Java 的异常处理的实现中有一个错误。异常是程序出错的标志，因此你不应该放过任何异常，但是这里会有可能轻易地漏过一个异常。当你用某些特定方式使用了 **finally** 之后，这种情况就发生了：

```
//: c09:LostMessage.java
// How an exception can be lost.
// {ThrowsException}
import com.bruceeckel.simpletest.*;

class VeryImportantException extends Exception {
  public String toString() {
    return "A very important exception!";
  }
}

class HoHumException extends Exception {
  public String toString() {
    return "A trivial exception";
  }
}

public class LostMessage {
  private static Test monitor = new Test();
  void f() throws VeryImportantException {
    throw new VeryImportantException();
  }
  void dispose() throws HoHumException {
    throw new HoHumException();
  }
  public static void main(String[] args) throws
Exception {
    LostMessage lm = new LostMessage();
    try {
      lm.f();
    } finally {
      lm.dispose();
    }
    monitor.expect(new String[] {
      "Exception in thread \"main\" A trivial
exception",
      "\tat
LostMessage.dispose(LostMessage.java:24)",
      "\tat LostMessage.main(LostMessage.java:31)"
    });  }
} ///:~
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

看到了吗？**VeryImportantException** 消失得无影无踪了，它直接被 **finally** 里面的 **HoHumException** 给取代了。这是一个相当严重的错误，它表示这个异常被彻底丢失了，而在真实环境中，这种情况会远比上述程序更为微妙，更难以察觉。相反，C++会把这种“第一个异常尚未得到处理，就产生了第二个异常”的现象，当成极端严重的编程错误。或许未来的 Java 版本会修复这个问题(另一方面，通常你应该将所有像 **dispose( )**之类的所有能抛出异常的方法，全都装到 **try-catch** 子句中。)

加在异常上面的限制

覆写方法的时候，你只能抛出这个方法在基类中的版本所声明的异常。这是一个有用的限制，因为同基类打交道的代码能自动地同它的派生类，包括它抛出地异常打交道(这是 OOP 的基本概念)。

下面这个例子就演示了这种(会在编译时)加在异常上面的限制：

```
//: c09:StormyInning.java
// Overridden methods may throw only the exceptions
// specified in their base-class versions, or
exceptions
// derived from the base-class exceptions.

class BaseballException extends Exception {}
class Foul extends BaseballException {}
class Strike extends BaseballException {}

abstract class Inning {
  public Inning() throws BaseballException {}
  public void event() throws BaseballException {
    // Doesn't actually have to throw anything
  }
  public abstract void atBat() throws Strike, Foul;
  public void walk() {} // Throws no checked
exceptions
}

class StormException extends Exception {}
class RainedOut extends StormException {}
class PopFoul extends Foul {}

interface Storm {
  public void event() throws RainedOut;
  public void rainHard() throws RainedOut;
}

public class StormyInning extends Inning implements
Storm {
  // OK to add new exceptions for constructors, but
you
  // must deal with the base constructor exceptions:
  public StormyInning()
    throws RainedOut, BaseballException {}
  public StormyInning(String s)
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
    throws Foul, BaseballException {}
  // Regular methods must conform to base class:
//! void walk() throws PopFoul {} //Compile error
  // Interface CANNOT add exceptions to existing
  // methods from the base class:
//! public void event() throws RainedOut {}
  // If the method doesn't already exist in the
  // base class, the exception is OK:
  public void rainHard() throws RainedOut {}
  // You can choose to not throw any exceptions,
  // even if the base version does:
  public void event() {}
  // Overridden methods can throw inherited
exceptions:
  public void atBat() throws PopFoul {}
  public static void main(String[] args) {
    try {
      StormyInning si = new StormyInning();
      si.atBat();
    } catch(PopFoul e) {
      System.err.println("Pop foul");
    } catch(RainedOut e) {
      System.err.println("Rained out");
    } catch(BaseballException e) {
      System.err.println("Generic baseball
exception");
    }
    // Strike not thrown in derived version.
    try {
      // What happens if you upcast?
      Inning i = new StormyInning();
      i.atBat();
      // You must catch the exceptions from the
      // base-class version of the method:
    } catch(Strike e) {
      System.err.println("Strike");
    } catch(Foul e) {
      System.err.println("Foul");
    } catch(RainedOut e) {
      System.err.println("Rained out");
    } catch(BaseballException e) {
      System.err.println("Generic baseball
exception");
    }
  }
} ///:~
```

你可以看到 **Inning** 的构造函数和 **event( )**方法都声称会抛出一个异常，但实际上它们什么异常也不抛。这么做是合法的，其目的就是让你能强制用户去捕捉那些“有可能会在 **event( )**的覆写版里实现的”异常。这种想法也能用于 **abstract** 方法，就像 **atBat( )**。

interface Storm 非常有趣，因为它的两个方法，一个在 **Inning** 里面(**event( )**)，一个不在，而这两个方法都会抛出一种新的异常，**RainedOut**。当 **StormyInning extends Inning** 并且

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

implements Storm 的时候，你就会发现 **Storm** 的 **event( )不能**改变 **Inning** 的 **event( )**的异常接口。这也是有道理的，否则的话，如果你用基类捕捉到这些异常的时候，就不知道该如何处理了。当然，如果方法是在 **interface** 而不是在基类中定义的，比如 **rainHard( )**，那么抛出异常的时候，那就没这个问题了。

这种异常方面的限制对构造函数不起作用。你可以从 **StormyInning** 看出，派生类的构造函数根本不看基类抛出的是什么类型的异常，它只会根据它自己的需要抛出异常。然而，由于派生类的构造函数总是会以这样或那样的形式调用基类的构造函数(这里调用的是默认的构造函数)，派生类的构造函数必须在异常说明中声明，它有可能会抛出其基类构造函数所抛出的异常。注意，派生类的构造函数不能捕获任何由基类构造函数抛出的异常。

StormyInning.walk( )之所以不能编译通过，是因为它会抛出异常，而 **Inning.walk( )**不会。如果编译器允许你这么做得话，你就能在调用 **Inning.walk( )**的时候不用做异常处理了。但是当你把它替换成 **Inning** 派生类的对象时，这个方法就有可能会抛出异常，于是程序就卡壳了。通过强制派生类遵守基类方法的异常说明，对象的可替换性得到了保障。

覆写后的 **event( )** 方法显示了，派生类方法可以不抛出任何异常，即使它是基类所定义的异常。这是因为，即使基类的方法会抛出异常，这样做也不会破坏已有的程序。类似的情况也出现在 **atBat( )**身上，它抛出的是 **PopFoul**，这个异常是继承自“会被基类的 **atBat( )**抛出”的 **Foul**。这样，如果你写的代码是同 **Inning** 打交道的，并且调用了它的 **atBat( )**的话，你就肯定能捕获 **Foul**。而 **PopFoul** 是由 **Foul** 派生出来的，因此异常处理程序也能捕获 **PopFoul**。

最后一个让人感兴趣的地方是 **main( )**。这里你能看到，如果你用的是具体的 **StormyInning** 对象的 reference 的话，编译器只会强制要求你捕获这个类所抛出的异常。但是如果你将它上传到基类，那么编译器就会(很严格地)要求你捕获基类的异常。所有这些限制都是为了能产生更为强壮的代码。[43]

尽管在继承过程中，编译器会对异常说明作强制要求，但异常说明本身并不属于方法的特征(signature)，特征是由方法的名字与参数的类型组成的。理解这一点会非常有用。因此，你不能根据异常说明来重载方法。此外，一个出现在基类方法的异常说明中的异常，不一定会出现在派生类方法的异常说明里。这点同继承的规则有明显不同，在继承中，基类的方法必须出现在继承类里。换一句话说，在继承和覆写的过程中，方法的“异常说明的接口”不是变大而是变小了——这正好和接口在继承时的情形相反。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

构造函数

用异常编程的时候，有一点非常重要，就是你必须时刻提醒自己，“如果这里发生了异常，它会得到清理吗？”绝大多数时候，你都是相当安全的，但是构造函数是个例外。构造函数的任务是把对象设置到一种安全启动的状态，但是它还可能进行的一些其它操作，比如打开一个文件。这都是些用户在“用完那个对象，并且调用特殊的清理方法”之前不会清理的东西。如果构造函数抛出了异常，那么这些清理活动就无法正常开展了。这就意味着你写构造函数的时候，必须特别仔细。

由于你刚学了 **finally**，可能会想可以用它来解决这个问题。但实际上没那么简单，因为 **finally** 会每次都执行清理代码，即便是你不需要它执行的时候它也会执行。因此，如果你真的要用 **finally** 来进行清理的话，你就必须设置一些标志，如果构造函数正常结束就设定标志，于是你就不需要再用 **finally** 来作任何清理了。由于这种方法不是特别优雅(你必须让两个地方的程序配合起来)，因此除非没有别的办法，最好还是避免用这种办法在 **finally** 里进行清理。

下面这段程序创建了一个“能打开并且逐行读取(转换成 **String**)文件”的 **InputFile** 对象。它用到了我们要在第 12 章讲的 Java 标准 I/O 类库中的 **FileReader** 和 **BufferedReader** 类，不过这两个类都比较简单，因此要能理解它们的基本用途大概没什么问题：

```
//: c09:Cleanup.java
// Paying attention to exceptions in constructors.
import com.bruceeckel.simpletest.*;
import java.io.*;

class InputFile {
  private BufferedReader in;
  public InputFile(String fname) throws Exception {
    try {
      in = new BufferedReader(new FileReader(fname));
      // Other code that might throw exceptions
    } catch(FileNotFoundException e) {
      System.err.println("Could not open " + fname);
      // Wasn't open, so don't close it
      throw e;
    } catch(Exception e) {
      // All other exceptions must close it
      try {
        in.close();
      } catch(IOException e2) {
        System.err.println("in.close()
unsuccessful");
      }
      throw e; // Rethrow
    } finally {
      // Don't close it here!!!
    }
  }
  public String getLine() {
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
    String s;
    try {
      s = in.readLine();
    } catch(IOException e) {
      throw new RuntimeException("readLine()
failed");
    }
    return s;
  }
  public void dispose() {
    try {
      in.close();
      System.out.println("dispose() successful");
    } catch(IOException e2) {
      throw new RuntimeException("in.close()
failed");
    }
  }
}

public class Cleanup {
  private static Test monitor = new Test();
  public static void main(String[] args) {
    try {
      InputFile in = new InputFile("Cleanup.java");
      String s;
      int i = 1;
      while((s = in.getLine()) != null)
        ; // Perform line-by-line processing here...
      in.dispose();
    } catch(Exception e) {
      System.err.println("Caught Exception in main");
      e.printStackTrace();
    }
    monitor.expect(new String[] {
      "dispose() successful"
    });
  }
} ///:~
```

InputFile 的构造函数需要一个 **String** 作参数，这个 **String** 就是你要打开的那个文件的文件名。**try** 区块用这个文件名创建了一个 **FileReader**。**FileReader** 并不是特别有用，你只是用它去创建一个真正去进行操作的 **BufferedReader**——注意 **InputFile** 的好处之一就是，它将这两步结合起来了。

如果 **FileReader** 的构造函数运行不成功，它会抛出一个 **FileNotFoundException** 异常，而这个异常必须单独处理。当你捉到这个异常的时候，并不需要关闭文件，因为这个文件还没有打开。而捕捉其它异常的 **catch** 子句必须关闭文件，因为在它们捉到异常之前，文件已经打开了。(当然，如果还有其它方法能抛出 **FileNotFoundException**，这个方法就显得有些投机取巧了。这时，你最好还是将它分成几个 **try** 区块。)**close( )**方法也可能会抛出异常，

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

所以尽管是已经在 **try** 里面了，还要再用一层 **try-catch**。对 Java 编译器而言，这只不过是又多了一对花括号。在本地做完处理之后，异常被重新抛出，对于构造函数而言这么做是很合适的，因为你总不希望去误到调用方，让它认为“这个对象已经创建完毕，可以使用了”吧。

这个例子没用使用前面所讲的设立标志的手段，由于 **finally** 会在每次完成构造函数之后都执行一遍，因此它实在不是 **close( )**文件的地方。我们希望文件在 **InputFile** 对象的整个生命周期内都处于打开状态，所以这么做肯定是不合适的。

getLine( )方法会返回表示文件下一行内容的 **String**。它调用了能抛出异常的 **readLine( )**，但是这个异常已经得到处理了，因此 **getLine( )**不会抛出任何异常。异常设计方面有一个课题，就是应该把异常全部放在这一层处理；还是这里处理一部分，然后再向上层抛相同的(或是新的)异常；还是直接往上面抛。如果情况合适，直接向上面抛当然能简化编程。不过这里，**getLine( )**方法将异常转换为 **RuntimeException**，以表示编程错误。

dispose( )方法要由用户在用完 **InputFile** 对象之后调用。这样就能释放 **BufferedReader** 和/或 **FileReader** 对象所占用的系统资源了(诸如文件句柄)。在你用完 **InputFile** 对象之前是不会调用它的，所以现在你不需要这个。可能你会考虑把上述功能放到 **finalize( )**里面，但是我们已经在第 4 章讲过了，你不知道 **finalize( )**会不会被调用(即使被调用，你也不知道它会在什么时候调用)。这是 Java 的另一面[1]；所有的清理——除了内存清理之外——都不会自动发生，所以你必须告诉客户程序员，他们也是有责任的，或者在使用 **finalize( )**的时候要保证清理活动会发生。

Cleanup.java 所创建的 **InputFile** 打开了这个程序的源文件，然后一行一行的读取文件，再加上行号。虽然我们也可以用更细致一点的方法，但这里所有的异常都由 **main( )**用 **Exception** 捕捉。

这个例子还有一个好处，它向你解释了，为什么本书要在这里介绍异常——如果不懂如何处理异常的话，很多类库(像前面提到的 I/O)是根本没法用的。异常同 Java 结合得如此的紧密，特别是编译器也会作强制要求，因此如果对异常一无所知的话，你就只能到此为止了。

异常的匹配

抛出异常的时候，异常处理系统会按照你写代码的顺序找出“最近”的处理程序。找到匹配的处理程序之后，它就认为异常已经得到处理了，于是查找任务也结束了。

[1] 原文为“downsides to Java”。英语有句谚语“two sides of a coin”，这里用的就是这个意思。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

匹配的时候并不要求抛出的异常同处理程序所要求的异常完全匹配。派生类的对象也可以匹配处理程序中的基类，就像这个程序：

```
//: c09:Human.java
// Catching exception hierarchies.
import com.bruceeckel.simpletest.*;

class Annoyance extends Exception {}
class Sneeze extends Annoyance {}

public class Human {
  private static Test monitor = new Test();
  public static void main(String[] args) {
    try {
      throw new Sneeze();
    } catch(Sneeze s) {
      System.err.println("Caught Sneeze");
    } catch(Annoyance a) {
      System.err.println("Caught Annoyance");
    }
    monitor.expect(new String[] {
      "Caught Sneeze"
    });
  }
} ///:~
```

Sneeze 会被第一个匹配的 **catch** 子句捕获，当然也就是程序里的第一个。然而如果你将这个 **catch** 删掉，只留下：

```
    try {
      throw new Sneeze();
    } catch(Annoyance a) {
      System.err.println("Caught Annoyance");
    }
```

程序仍然能运行，因为这次它抓的是 **Sneeze** 的基类。换言之，**catch(Annoyance e)**会捕获 **Annoyance** 以及所有由它派生出来的异常。这一点非常有用，因为如果你打算往方法里加上更多的派生异常的话，只要客户程序员捕捉的是基类异常，那么他们的代码就无需更动了。

如果你把捕捉基类的 **catch** 子句放在最前面，就会把派生类的异常全给“屏蔽”掉，就像这样：

```
    try {
      throw new Sneeze();
    } catch(Annoyance a) {
      System.err.println("Caught Annoyance");
    } catch(Sneeze s) {
      System.err.println("Caught Sneeze");
    }
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

这样编译器就会发现 **Sneeze** 的 **catch** 子句永远也得不到执行，因此它就给你报一个错误。

其它方法

异常处理系统还是一扇能让你放弃程序正常运行顺序的暗门。当某种“异常情况”出现的时候，这个门就开启了，于是程序再也不能，或者再也不需要按照正常的顺序执行下去了。异常表示的是一些“当前方法无法处理”的情况。之所以要开发异常处理系统，是因为，要对每次函数调用的每个错误都做检查实在是太费力了，所以程序员们根本就不去做。结果就是他们把错误也给忽略了。值得注意的是，开发异常处理的初衷是为了方便程序员。

异常处理的一个重要准则就是“如果你不知道该如何处理这个异常，你就别去捕捉它”。实际上，异常处理有一项很重要的目标，这就是将处理异常的代码从异常发生的地方移开。这样你就能在一个地方集中精力去解决你想解决的问题，然后再到另一地方去处理这些异常问题了。这样程序的主线就不会被异常处理这类枝节问题给搞得肢离破碎了，于是程序也变得更易于理解和维护了。

但是，checked expection 将这种情况变得稍微复杂了一点，这是因为即使你不准备这么做，编译器也会强制要求你在 try 的后面加上了 catch 子句。于是，这就导致了“错误地私吞(harmful if swallowed)”的问题了：

```
try {
  // ... to do something useful
} catch(ObligatoryException e) {} // Gulp!
```

程序员们(包括我自己，在本书的第一版里) 都挑了最简单的事情做，于是这个异常就被私吞了——通常不是有意要这么作的，但是只有这样才能编译通过，所以除非你记得回来更正这些代码，否则异常就会丢了。异常发生过，但是被你私吞了，因此它已经消失得无影无踪了。既然编译器要求你立刻写出异常处理程序，那么即便这是最糟的选择，要图方便的话，最好还是照办。

当我意识到我犯了这么大一个错的时候，我简直吓了一大跳，接着在第二版中，我用在处理程序里用打印栈痕迹的方法来“修补”这个问题(本章中的很多例子还是用了这个办法，这个做法还是比较合适的)。尽管这样可以跟踪异常的行为，但是你还是不知道该如何处理异常。这一节，我们

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

来看看 checked exception 的问题和它的并发症，以及应该采取什么方法来解决这些问题。

这个话题看起来简单，但实际上还不仅仅是复杂，更重要的是它还非常多变。总有人会顽固地坚持自己的立场，声称正确的(也是他们的)答案是不言自明的。我觉得之所以会有这种论点，是因为我们所使用的工具已经不是“ANSI 标准出台前的 C 那样的”弱类型语言了”(poorly-typed language)，而是像 C＋＋和 Java 这样的“强静态类型语言”(strong, statically-typed language，也就是编译时就作类型检查的语言)，这是前者所无法比拟的。当你刚开始这个转变的时候(就像我一样)，会发现它带来的好处是那样生动，好像强类型检查能一劳永逸地解决所有问题。这里，我想结合我自己的认识过程，告诉你我是怎样从对类型检查的绝对迷信变成怀疑的；当然，很多时候它还是非常有用的，但是当它挡住我们的去路并且成为障碍的时候，我们就得跨过去。只是这条界限并不是很清晰的(我最喜欢的一句格言就是：“所有模型都是错的。但是有些是能用的。”)。

历史

异常处理最早出现在 PL/1 和 Mesa 之类的系统里，后来又出现在 CLU，Smalltalk，Modula-3，Ada，Eiffel，C++，Python，Java 以及 Java 后面的 Ruby 和 C#里面。Java 的设计与 C++的很相似，只是 Java 的设计者们去掉了一些他们认为 C++设计得有问题的东西。

为了能向程序员们提供一个他们更愿意使用的处理和恢复错误的框架，异常处理很晚才被加进“由 C++的设计者，Bjarne Stroustrup 所倡议的”C++的标准化过程里。C++的异常模型主要是效法 CLU 的。然而当时其它语言已经支持异常处理了：Ada，Smalltalk (两者都有异常处理，但是都没有异常说明)，以及 Modula-3 (它既有有异常也有说明)。

Liskov 和 Snyder 在他们的项目报告中[44]指出，用 C 的瞬时风格(transient fashion)报告错误的语言有一个主要缺陷，就是：

“...每次调用的时候都必须测试条件，并判断结果。这使得程序难以阅读，并且还有可能会降低运行效率，因此程序员们都不愿意使用异常信号，也不愿意去处理异常。”

注意，异常处理的初衷就是要取消这种限制，但是我们又从 Java 的 checked exception 看到了这种代码。他们继续道：

“...在调用会引发异常的函数的同时，还要求程序员给出异常处理程序，会降低程序的可读性，使得程序的正常思路被异常处理给破坏了。”

C++异常的设计参照了 CLU 的。Stroustrup 声称其目标是要减少恢复错误所需的代码。我想他这话是说给那些“通常情况下都不写 C 的异常处

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

理”的程序员们听的，因为要把这么多代码放到这么多地方实在不是什么好差使。所以他们写 C 程序的习惯是，忽略所有的错误，然后使用 debugger 来排错。这些程序员们知道，使用异常就意味着他们要写一些通常不用写的“多出来的”代码。因此，要把他们拉到“使用错误处理”的正确轨道上，“多出来的”代码决不能太多。我认为，评价 Java 的 checked expection 的时候，这一点是很重要的。

C++还从 CLU 那里还带过来另一种思想：在方法的特征签名中申明这个方法可能会抛出哪些异常的异常说明。异常说明有两个目的。首先它说“我的代码会产生这种异常，你得处理”。另一层意思是“我忽略了这些异常，它要由你来处理”。学习异常的机制和语法的时候，我们一直在关注“你来处理”这部分，但这里我特别感兴趣的是这么一个事实，我们通常都忽略了异常说明所申明的异常。

C++的异常说明不属于函数的特征。编译时唯一要检查的是这些异常说明是不是前后一致；比如，如果一个函数或方法会抛出某些异常，那么它重载版或派生版也必须也抛出同样的异常。与 Java 不同，它不会在编译时进行检查以确定函数或方法是不是真的抛出异常，或者异常说明是不是完整(也就是说，异常说明有没有精确地描述所有会被抛出的异常)。这种检查还是有的，但只是在运行时。如果抛出的异常不符合异常说明，C++会调用标准类库的 **unexpected( )**方法。

有趣的是，由于使用了模板(template)，C++的标准类库里根本没有使用异常说明。由此看来，异常说明会对 Java 的 *generics* 产生非常重大的影响(Java 版的 C++模板，可望在 JDK 1.5 中出现)。

观点

首先要指出，Java已经在事实上发明了 checked exception (很明显是受 C++的异常说明的启发，同时也因为它发现 C++的程序员们一般不会为此费心)。这还是一个实验，在此之前还没有别的语言采用这种做法。

再者，仅从示意性的例子和小程序来看，checked exception 有很明显的好处。但是当程序开始膨胀的时候，它就会带来一些难度了。当然，程序不是一夜变大的；它是慢慢变大的。不适于大项目的语言会被用于会增长的小项目。这样突然有一天，你会发现原先可以管理的东西，现在已经变得无法管理了。这就是我所说的过多的类型检查，特别是 checked exceptions 所造成的问题。

看来程序的规模是个大课题。由于很多讨论都用小程序来做演示，因此这成了个问题。有个 C#的设计者发现：

“仅从小程序看，你会认为异常说明能增进开发人员的效率，提高代码的质量；但是考察大项目的时候，结论就不同了——开发效率下降了，而代码质量只有微不足道的提高，甚至毫无提高”[45]

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

谈到未被捕获的异常的时候，CLU 的设计师们说道：

“我们觉得要让程序员在不知道该采取什么措施的时候提供处理程序，是不现实的。” [46]

Stroustrup 是这样解释为什么一个没有异常说明方法可以抛出任何异常：

“但是，这样一来几乎所有的函数就都得提供异常说明了，于是就都要重新编译了，而且这样还会妨碍它同用其它语言的合作。这样会逼程序员们去造异常处理机制的反的，他们会写欺骗程序来掩盖异常。这样别人就注意不到这些异常，于是会造成一种虚假的安全感。” [47]

我们已经看到有人在造异常的反了，就在 Java 的 checked exception 上。

Martin Fowler (*UML Distilled*, *Refactoring*, and *Analysis Patterns* 的作者)给我写了下面这段：

“...总体的来说，我觉得异常很不错，但是 Java 的 checked exception 带来的麻烦比好处多。”

我觉得 Java 的当务之急应该是统一其报告错误的模式，这样所有的错误都能通过异常得到报告。C++没有这么做，这是因为它考虑到要向后兼容，要照顾那些直接忽略所有错误的 C 代码。但是如果你一贯用异常来报告错误，那么只要愿意随时可以用异常，如果不是，那么这些错误会被送到到最上层(控制台或其它外层程序 container program)。Java 修改了 C++的模型，因此异常成了报告错误的唯一方式，所以 checked exception 的额外限制就变得不那么必须了。

过去，我曾坚定地认为 checked exception 与强静态类型检查对开发健壮的程序是非常必要。但是，我看到的和我的亲身经历[48]告诉我，这些好处实际上是来自于：

1. 不是编译器会不会强制程序员去处理错误，而是它有一个统一的使用异常的错误报告模式。
2. 不在于什么时候检查，而是要有类型检查。也就是说，必须强制程序使用正确的类型，但至于是在编译的时候还是运行时，那倒无所谓。

此外，减少编译时的限制能显著提高程序员的编程效率。实际上，*reflection* (说到底还有 *generics*)都是用来补偿强静态类型检查所带来的过多限制的，就像你会在下一章以及本书的很多例子中所见的。

已经有人在指责了，他们说，这是异端邪说，这种言论会令我名誉扫地，会让文明陷落，会导致更高比例的项目破产。他们的信念是应该在编译时

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

指出所有错误，这样才能挽救项目，这种信念可以说是无比坚定的；其实更重要的是要理解编译器的能力极限；在第 15 章，我强调了自动编译(automated build process)和单元测试(unit testing)的重要性，比起把所有东西都说成是语法错误，它们的功效可说是事半功倍。下面这段话可说是至理名言：

好的编程语言能帮程序员写出好程序。但是无论哪种语言挡不住你去写坏程序。[49]

不管怎么说，要让 Java 把 checked exception 去掉，这种可能性看来是微乎其微的。对语言来说，这个变化可能太激进了些，况且 Sun 的支持者们也非常强大。Sun 有完全向后兼容的历史和政策——讲个故事给你听听，实际上所有 Sun 的软件都能跑在 Sun 的硬件上，无论它有多古老。但是，如果你发现有些 checked exception 挡住了你的路，或者你发现你不得不去捕捉连你自己都不知道该如何处理的异常，那还是有些办法的。

将异常传到控制台

在一些简单的程序里，就像这本书里的很多例子，要使用异常而又不用写很多代码的最简单的办法就是将它们送出 **main( )**，送到控制台去。比如，如果你想读文件就得把它打开(第 12 章会作详细介绍)，这样就要用到 **FileInputStream**，而它会抛出异常。对一些简单的程序，你可以这样做(你可以在本书的很多地方看到这种用法)：

```
//: c09:MainException.java
import java.io.*;

public class MainException {
  // Pass all exceptions to the console:
  public static void main(String[] args) throws
Exception {
    // Open the file:
    FileInputStream file =
      new FileInputStream("MainException.java");
    // Use the file ...
    // Close the file:
    file.close();
  }
} ///:~
```

注意 **main( )**方法也可以有异常说明，这里抛出的是所有 checked exception 的根 **Exception**。如果异常被传到控制台，你就无需再在 **main( )**的主体里用 **try-catch** 子句了。(不幸的是，文件 I/O 要比这里看到的复杂得多，所以在学完第 12 章之前别高兴的太早)。

将 checked exception 转换成 unchecked exception

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

写 **main( )**的时候，抛异常是很方便的，但这种方法不是很通用。真正要解决的是写普通方法。你调用另一个方法，然后发现“我不知道怎样处理这个异常，但是又不能把它私吞了，或者打印一些乱七八糟的消息”。JDK 1.4 的异常链提供了一种新的思路来解决这个问题。你直接把 checked exception“包”进 **RuntimeException** 里面，就像这样：

```
try {
  // ... to do something useful
} catch(IDontKnowWhatToDoWithThisCheckedException e)
{
  throw new RuntimeException(e);
}
```

如果你想把 checked exception 的功能“关掉”的话，这看上去像是一个好办法——你无需私吞异常，也不必把它放到方法的异常说明里面，而且异常链还能保证你不会丢失任何异常的原始信息。

这种技巧给了你一种选择，你可以不写 **try-catch** 子句和异常说明了，直接忽略异常，让它自己沿着调用栈(call stack)往上跑。同时，你还可以用 **getCause( )**捕捉并处理这个异常的，就像这里：

```
//: c09:TurnOffChecking.java
// "Turning off" Checked exceptions.
import com.bruceeckel.simpletest.*;
import java.io.*;

class WrapCheckedException {
  void throwRuntimeException(int type) {
    try {
      switch(type) {
        case 0: throw new FileNotFoundException();
        case 1: throw new IOException();
        case 2: throw new RuntimeException("Where am
I?");
        default: return;
      }
    } catch(Exception e) { // Adapt to unchecked:
      throw new RuntimeException(e);
    }
  }
}

class SomeOtherException extends Exception {}

public class TurnOffChecking {
  private static Test monitor = new Test();
  public static void main(String[] args) {
    WrapCheckedException wce = new
WrapCheckedException();
    // You can call f() without a try block, and let
    // RuntimeExceptions go out of the method:
    wce.throwRuntimeException(3);
    // Or you can choose to catch exceptions:
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
    for(int i = 0; i < 4; i++)
      try {
        if(i < 3)
          wce.throwRuntimeException(i);
        else
          throw new SomeOtherException();
      } catch(SomeOtherException e) {
        System.out.println("SomeOtherException: "
+ e);
      } catch(RuntimeException re) {
        try {
          throw re.getCause();
        } catch(FileNotFoundException e) {
          System.out.println(
            "FileNotFoundException: " + e);
        } catch(IOException e) {
          System.out.println("IOException: " + e);
        } catch(Throwable e) {
          System.out.println("Throwable: " + e);
        }
      }
    monitor.expect(new String[] {
      "FileNotFoundException: " +
      "java.io.FileNotFoundException",
      "IOException: java.io.IOException",
      "Throwable: java.lang.RuntimeException: Where
am I?",
      "SomeOtherException: SomeOtherException"
    });
  }
} ///:~
```

WrapCheckedException.throwRuntimeException( ) 的代码可以生成其它类型的异常。这些异常被包进了 **RuntimeException** 对象，所以它们成了这些异常的“cause”了。

TrunOffChecking 可以不用 **try** 区块就调用 **throwRuntimeException( )**，因为它抛出的不是 checked exception。但是，当你决定去捕捉异常的时候，你还是可以用 **try** 区块来捕捉任何你想捕捉的异常。你应该先捕捉 try 区块肯定会抛出的异常，这里就是 **SomeOtherException**。**RuntimeException** 要放到最后去捉。然后把 **getCause( )**结果(也就是包在里面的那个异常)抛出来。这样就把原先那个异常给提取出来了，然后就可以用它们自己的 **catch** 子句来处理了。

本书的其余部分会尽量使用这种“用 **RuntimeException** 来包装 checked exception”的技术。

异常运用的原则

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

使用异常来：

1. 在合适的地方处理问题。(避免在自己还不知道该如何处理的情况下去捕捉异常)。
2. 把问题解决掉，然后重新调用那个引起问题的方法。
3. 修正一下问题，然后绕过那个方法再继续下去。
4. 用一些别的，不准备让这个方法返回的数字来进行计算。
5. 把当前运行环境下能做的事情全部做完，然后把**相同**的异常抛到更高层。
6. 把当前运行环境下能做的事情全部做完，然后抛一个**不同**的异常到更高层。
7. 中止程序。
8. 简化。(如果异常结构把事情搞得太复杂了，那用起来会非常痛苦也很烦人。)
9. 把类库和程序做得更安全。(这既是在为调试作短期投资，也是在为程序的健壮性作长期投资。)

总结

更完善的“错误恢复(error recovery)”是一种最能增强程序健壮性的工具。错误恢复对任何程序来说，都是很重要的，但是对 Java 来说，它就更重要了，因为 Java 的首要目标就是要创建供别人使用的程序组件。**而要创建一个强壮的系统，其组件必须强壮**。通过使用异常，Java 提供了一种统一的错误报告模式，这使它能将组件所发生的问题可靠地传递给客户代码。

Java 异常处理的目的就是要让我们能用比现在更少的代码，以一种更简单的方式来开发大型，可靠的程序，并且让你在开发过程中能更自信“你的程序里面没有未经处理地错误”。异常不是特别难学，但是却能给项目带来立竿见影的效果。

练习

只要付很小一笔费用就能从 *www.BruceEckel.com* 下载名为 *The Thinking in Java Annotated Solution Guide* 的电子文档，这上面有一些习题的答案。

1. 创建一个类，在 **main( )**的 **try** 区块里抛出一个 **Exception**。传一个 **String** 参数给 **Exception**。用 **catch** 子句捕捉这个异常，然后再打印这个 **String** 参数。最后加一个 **finally** 子句，让它打印一些信息以证明程到过那里。
2. 用 **extends** 关键词创建一个你自己的异常。为这个类写一个带 **String** 参数的构造函数，并且让对象保存这个 **String**。写一个会把这个

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

String 打印出来的方法。创建一个 **try-catch** 子句，测试一下这个新的异常。

3. 写一个包含“能抛出练习 2 所创建的异常的方法”的类。不用异常说明，看看能否编译通过。加上异常说明。用 **try-catch** 子句测试一下这个类和异常。
4. 定义一个对象的 reference，把它初始化为 **null**。用这个 reference 调用对象的方法。然后用 **try-catch** 子句把这段代码包起来以捕获异常。
5. 创建一个带 **f( )**和 **g( )** 这两个方法的类。在 **g( )**里面抛出一种你新定义的异常。用 **f( )** 调用 **g( )**，捉到异常之后要在 **catch** 子句里抛出另一种异常(你定义的第二种异常)。用 **main( )**作测试。
6. 重复前一个练习，但是在 **catch** 子句里，要用 **RuntimeException** 把 **g( )**的异常包起来。
7. 创建三种新的异常。写一个带“能抛这三种异常的”方法的类。在 **main( )**里面调用这个方法，但是只能有一条 **catch** 子句，而它要能捕获这三种异常。
8. 写一个会引发并且抛出 **ArrayIndexOutOfBoundsException** 的程序。
9. 用 **while** 循环创建一个类似“恢复模型”的异常处理程序，让它重复运行，直到不再抛出异常。
10. 创建一个三层继承的异常体系。然后创建一个带“能抛出基类异常”的方法的类 **A**。用 **B** 继承 **A**，并且覆写这个方法，让它抛出第二层的异常。继续下去，让 **C** 继承 **B**。在 **main( )**里面创建一个 **C** 的对象，上传到 **A**，然后调用这个方法。
11. 验证一下，派生类的构造函数捕捉不到它的基类构造函数所抛出的异常。
12. 演示一下，**try** 区块抛出 **RuntimeException** 之后 **OnOffSwitch.java** 会运行失败。
13. 演示一下，**try** 区块抛出 **RuntimeException** 之后 **WithFinally.java** 不会运行失败。
14. 修改练习 7，加一个 **finally** 子句。验证一下，即便是抛出 **NullPointerException** 的情况下，**finally** 子句也会得到执行。
15. 用“构造函数”这一节的第二段所说方法，创建一个用标记来控制是否调用清理代码的例子。
16. 修改 **StormyInning.java**，加一个 **UmpireArgument** 异常，再加一个会抛出这种异常的方法。测试一下修改后的继承体系。
17. 删除 Human.java 的第一个 catch 子句，验证一下，代码仍然能正常编译和运行。
18. 让 **LostMessage.java** 再多丢失一层异常，用第三个异常来替代 **HoHumException**。
19. 为第八章的 **GreenhouseControls.java** 添加一套合适的异常。
20. 为第八章的 Sequence.java 添加一套合适的异常。
21. 用一个不存在的文件名来替换 **MainException.java** 文件名。运行程序，注意一下运行结果。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

[40] C 的程序员们只要去看看 **prinf( )**的返回值，就会知道我没有夸大其词了。

[41] C++的异常处理没有 finally 子句，因为它要用拆构器(destructor)来完成类似的清理工作。

[42] 拆构函数是一种应该在“对象已经用完”之后调用的函数。你总能明确地知道，是在什么时间，什么地点调用拆构函数的。C++会自动调用拆构器，而 C#(它更像 Java)里面有会进行自动清理的机制。

[43] ISO C++加上了相似的限制，要求派生类的方法所抛出的异常要与基类方法的相同，或者是基类方法抛出的异常的派生类。这是 C++真正能够在编译时进行异常说明检查的地方。

[44] Barbara Liskov 和 Alan Snyder：CLU 的异常处理 (*Exception Handling in CLU*)，IEEE Transactions on Software Engineering，Vol. SE-5， No. 6，1979 年 11 月。这份报告在 Internet 上是找不到的，只有印刷的，所以你只能联系图书馆找一份拷贝了。

[45] http://discuss.develop.com/archives/wa.exe?A2=ind0011A&L=DOTNET&P=R32820

[46] 同上

[47] Bjarne Stroustrup, *The C++ Programming Language, 3rd edition, Addison-Wesley 1997*，第 376 页.

[48] 间接的是从与很多有经验的 Smalltalk 程序员的谈话中得到的；直接的则是得自于 Python(*www.Python.org*)。

[49] (Kees Koster, CDL 语言的设计者，出自 Eiffel 语言的设计者 Bertrand Meyer) http://www.elj.com/elj/v1/n1/bm/right/。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com