JAVA基础：突破Java异常处理规则 PDF转换可能丢失图片或格式，建议阅读原文 https://www.100test.com/kao_ti2020/145/2021_2022_JAVA_E5_9F_BA_E7_A1_80_c104_145251.htm 问题: 我在我的应用程序中调用了外部方法并且想捕获它可能抛出的异常。我能捕获java.lang.Exception吗？ 答案: 通过一个给定的方法去处理所有运行时和检测异常对于预防外部错误是不充分的。 你可以去读目前 JavaWorld文章 “ Java Tip 134: When Catching Exception, Don ’ t Cast Your Net Too Wide ”。这篇文章警告了捕获java.lang.Exception和java.lang.Throable是不好的。捕获你能指定的异常对于代码的可维护性是十分重要的。然而这个规则依赖于特殊的环境。如果你不打算你的程序崩溃并且保留你的数据结构的安全异常，那么你必须捕获被抛出的真正的异常。 举个例子，想象你有一个加载了这个接口的服务器应用： public interface IFoo { /** * This method cant throw any checked exceptions...or can it? */ void bar (). } // End of interface 对于给出参数的理由是让我们通知你这样的服务在什么地方，并且不同的IFoo实现能够从外部资源加载上。你写如下代码： try { IFoo foo = ... // get an IFoo implementation foo.bar (). } catch (RuntimeException ioe) { // Handle ioe ... } catch (Error e) { // Handle or re-throw e ... } 并且你在这个里处理了所有可能的异常。你不需要在这里加上任何捕获java.io.IOException的异常，因为IFoo实现没有从IFoo.bar()中抛出它，对吗？（事实上，如果你加上了捕获java.io.IOException异常块，编译器可能会把它作为不可到达的异常而丢弃） 错误。在我写的EvilFoo类中bar()方法证明了将抛出你传递给类构造器的任何异常：

public void bar () { EvilThrow.throwThrowable (m_throwthis). } 运行Main方法： public class Main { public static void main (final String[] args) { // This try/catch block appears to intercept all exceptions that // IFoo.bar() can throw. however, this is not true try { IFoo foo = new EvilFoo (new java.io.IOException ("SURPRISE!")). foo.bar (). } catch (RuntimeException ioe) { // Ignore ioe } catch (Error e) { // Ignore e } } } // End of class 你将看到从bar()方法抛出的java.io.IOException异常实例并且没有任何捕获块： >java -cp classes Main Exception in thread "main" java.io.IOException: SURPRISE! at Main.main(Main.java:23) 在这里发生了什么？ 主要的观察是通常针对检测异常的Java规则仅仅在编译的时候被执行。在运行的时候，一个JVM不能保证被一个方法抛出的异常是否和在这个方法中声明的抛出异常相匹配。因为调用方法的职责是捕获和处理所有从调用方法抛出的异常。任何没有被调用方法声明的异常将不予理睬并且拒绝调用栈。 如果正常行为是编译器执行，那么我怎么创建EvilFoo的？至少有两个方法可以去创建抛出没有声明的异常的Java方法：Thread.stop(Throwable)在一些时候不被赞成使用，但是它仍然被使用并且传递一个Throwable给被调用的Thread。 分别编译：你能在编译EvilFoo时候不去编译真正声明bar()方法抛出检测异常的IFoo临时版本。 我用后一种选择：我编译开始定义的EvilThrow类： public abstract class EvilThrow { public static void throwThrowable (Throwable throwable) throws Throwable { throw throwable. } } 接下来，我用Byte Code Engineering Library(BCEL)的JasminVisitor分解结果，在汇编代码中删除throwThrowable()方法Throwable的声明，并且用Jasmin

assembler 编译新的版本。 如果你编写捕获异常的构造器，那么它应该总是捕获java.lang.Throwable而不仅仅只捕获java.lang.Exception。这个规则适合你开发管理运行时的应用程序和必须执行可能包含错误甚至恶意代码的外部组件。你要确保捕获Throwable并且过滤掉错误信息。 下面示例说明了如果你没有遵循这个建议将发生什么。 Example: Breaking SwingUtilities.invokeAndWait() javax.swing.SwingUtilities.invokeAndWait()是在AWT上执行一个线程的有用方法。当一个应用程序线程必须更新图形用户接口并且服从所有Swing线程规则的时候这个方法将被调用。一个没有捕获Runnable.run()抛出的异常将被捕获并且被封装在一个InvocationTragetException中重新抛出。 Sun的J2SE1.4.1假设这样一个未捕获的异常仅仅是java.lang.Exception的子类。这里是一个SwingUtilities.invokeAndWait()调用java.awt.event.InvocationEvent的一个分析： public void dispatch() { if (catchExceptions) { try { runnable.run(). } catch (Exception e) { exception = e. } } else { runnable.run(). } if (notifier != null) { synchronized (notifier) { notifier.notifyAll(). } } } 这段代码的问题是如果runnable.run()抛出一个Throwable，捕获块又没有并且notifier.notifyAll()从来不会被执行。然后调用应用线程将等待在java.awt.EventQueue.invokeAndWait()里的一个非公共锁对象（lock.wait()将从未执行）： public static void invokeAndWait(Runnable runnable) throws InterruptedException, InvocationTargetException { class AWTInvocationLock {} Object lock = new AWTInvocationLock(). InvocationEvent event =new InvocationEvent(Toolkit.getDefaultToolkit(), runnable, lock,true).

synchronized (lock) { Toolkit.getEventQueue().postEvent(event). lock.wait(). } Exception eventException = event.getException(). if (eventException != null) { throw new InvocationTargetException(eventException). } } 让EvilFoo实现Runnable接口： public void run () { bar (). } 然后，在Main中调用它： SwingUtilities.invokeAndWait (new EvilFoo (new Throwable ("SURPRISE!"))). 正如你看到的，未受信任代码使你的代码进入你没有准备处理的执行路径中的异常被保护起来。