Atitit 反模式 黑名单 异常处理

**反模式（antipatterns）**

目录

[1.1. 记录并抛出（log and throw） 1](#_Toc9192)

[1.2. 抛出异常基类（Throwing Exception） 2](#_Toc21168)

[1.3. Throwing the Kitchen Sink（太多性质类似的异常 太细节化） 2](#_Toc10243)

[1.4. 捕获异常基类（Catching Exception） 2](#_Toc31797)

[1.5. 破坏性的包装 丢失异常链 2](#_Toc22417)

[1.6. 记录并抛出Null（Log and Return Null） 3](#_Toc89)

[1.7. 捕获然后忽略（Catch and Ignore） 3](#_Toc19800)

[1.8. 在finally中抛出异常 3](#_Toc26755)

[1.9. 一条消息分多行进行记录（Multi-Line Log Messages） 4](#_Toc18429)

[1.10. 本应抛出UnsupportedOperation异常却抛出null（Unsupported Operation Returning Null） 4](#_Toc12366)

[1.11. 忽略InterruptedException （Ignoring InterruptedException ） 4](#_Toc22027)

[1.12. 依靠getCause()函数（Relying on getCause()） 5](#_Toc12276)

## 记录并抛出（log and throw）

　　例如

|  |
| --- |
| catch (NoSuchMethodException e) {   LOG.error("Blah", e);   throw e; } |

　　或者

|  |
| --- |
| catch (NoSuchMethodException e) {   LOG.error("Blah", e);   throw new MyServiceException("Blah", e); } |

　　或者

|  |
| --- |
| catch (NoSuchMethodException e) {   e.printStackTrace();   throw new MyServiceException("Blah", e); } |

　　这三种方式都是错误的。这类方式是最讨人厌的错误处理反模式。要么记录一个异常，要么抛出一个异常，但不要同时进行“抛出”和“记录”两种操作。同时进行这两类操作会对同一个问题产生多种log消息，这会给运维人员分析日志带来麻烦。

## ****抛出异常基类（Throwing Exception）****

　　看下面这个例子：

　　public void foo() throws Exception {

　　这样做是草率的，它完全违背了使用checked异常的目的。它告诉调用你代码的人“您现在调用的函数可能会出错哦”，虽然这有一些作用的，但千万别这么做。应该准确声明你的方法有可能会抛出的异常的类型。如果要抛出的异常有很多种，那么可以将它们包装到你定义的自定义异常中。（详见下文的"Throwing the Kitchen Sink"）

## ****Throwing the Kitchen Sink（太多性质类似的异常 太细节化）****

　　例如：

|  |
| --- |
| public void foo() throws MyException,     AnotherException, SomeOtherException,     YetAnotherException { |

　　抛出多个checked类型的异常是可以的，只要函数调用者能针对不同的异常提供不同的处理方法即可。如果你抛出的几个checked异常对调用者而已差不多是同样的性质，那么应该将它们包装成一类单独的checked异常。

## ****捕获异常基类（Catching Exception）****

　　例如：

|  |
| --- |
| try {   foo(); } catch (Exception e) {   LOG.error("Foo failed", e); } |

　　这通常是错误的和草率的。这种方式下捕获了原本应该被抛出的异常。捕获异常基类的问题在于，如果你随后要调用别的函数，而这个函数含有一个checked类型的异常（函数开发者希望你处理这个特定的checked异常），那么由于你之间捕获了Exception基类（甚至是Throwable类），那么你或许永远不知道你的代码里有本应该处理但却没有处理异常，这样一来你的代码是错误的而你却无从知晓（IDE不会提示，因为Exception基类被捕获了）。

## ****破坏性的包装 丢失异常链****

　　例子：

|  |
| --- |
| catch (NoSuchMethodException e) {   throw new MyServiceException("Blah: " +       e.getMessage()); } |

　　这种方式破坏了原本的异常对象e的追踪堆栈，使用这种包装方式你将无法追踪这个异常之前的传递路径。

## ****记录并抛出Null（Log and Return Null）****

　　例子：

|  |
| --- |
| catch (NoSuchMethodException e) {   LOG.error("Blah", e);   return null; } |

　　或

|  |
| --- |
| catch (NoSuchMethodException e) {   e.printStackTrace();   return null; }  // Man I hate this one |

　　并不是所有情况下这样处理都是错的，但通常它是不正确的处理方式。相比于返回null，抛出异常让该函数的调用者来处理会更好一些。只有在正常的情况下（非异常处理）才应该有返回null这样的语句出现。例如，当查找的字串不存在时返回null。

## ****捕获然后忽略（Catch and Ignore）****

　　例子：

|  |
| --- |
| catch (NoSuchMethodException e) {   return null; } |

　　这种方式是阴险的，它不但不做任何处理而是返回null，并且还吞掉了原本的异常对象，使它丧失了所有的信息！！

## ****在finally中抛出异常****

　　例子：

|  |
| --- |
| try {   blah(); } finally {   cleanUp(); } |

　　如果 cleanUp()不会抛出任何异常，那么这样写是没问题的。在上例中，如果blah()函数抛出了一个异常，然后在finally 语句块中cleanUp()又抛出一个异常，那么第二个异常将会被抛出，而第一个异常则完全消失了。如果finally 语句块中调用的函数会抛出异常，那么要么处理它，要么记录它，千万不要让它逃出finally 语句块的范围。

## ****一条消息分多行进行记录（Multi-Line Log Messages）****

　　例子：

|  |
| --- |
| LOG.debug("Using cache policy A"); LOG.debug("Using retry policy B"); |

　　不管在那个代码层次上，都应该尝试将消息组织到一起，对于上面这个例子，正确的编码方式是：

　　LOG.debug("Using cache policy A, using retry policy B");

　　将统一组的日志记录到两个调用语句中，在测试用例的测试下或许看起来没什么问题。但是在多线程（假设有500个线程）的系统中，信息将喷涌般地被记录到log文件中，而讲一条语句拆做两条写可能会让这两条语句中间相差十万八千里，而它们本应该同时输出的。

## ****本应抛出UnsupportedOperation异常却抛出null（Unsupported Operation Returning Null）****

　　例子：

|  |
| --- |
| public String foo() {   // Not supported in this implementation.   return null; } |

　　如果上述代码是用在一个抽象基类中，用来提供钩子（hooks）以供子类在重写的话，那么是可以的。若非如此，则应该抛出一个UnsupportedOperationException 而不是返回一个null。对于方法的调用者而已，如果你抛出了一个UnsupportedOperationException，那么他们会更容易知道自己的方法为什么没有正常工作。如果你是抛出null的话，函数的调用者可能就会接收到莫名其妙的NullPointerException了。

## ****忽略InterruptedException （Ignoring InterruptedException ）****

　　例子：

|  |
| --- |
| while (true) {   try {     Thread.sleep(100000);   } catch (InterruptedException e) {}   doSomethingCool(); } |

　　InterruptedException 是一个提示，用来告知代码不管现在在做什么，都停下。一个线程被中断的情况通常出现在事务处理时间耗尽或线程池被关闭。相比于忽略InterruptedException，代码中更应该做的是赶快完成现在在做的工作，并结束当前线程。所以，正确的写法应该是：

|  |
| --- |
| while (true) {   try {     Thread.sleep(100000);   } catch (InterruptedException e) {     break;   }   doSomethingCool(); } |

## ****依靠getCause()函数（Relying on getCause()）****

　　例子：

|  |
| --- |
| catch (MyException e) {   if (e.getCause() instanceof FooException) {     ... |

　　依赖于getCause()函数的结果会让你的代码变得脆弱。如果你调用的函数或者你所依赖的代码改变了它的底层实现，换了一种异常封装，而你却依赖之前的异常类型来进行判断，怎么办？其实你本意上是想判断这个异常最初的根源是什么，也就是cause's cause。现在Apache的 commons-lang提供了一个ExceptionUtils.getRootCause() 方法来轻松获得异常源。

**结论**

　　好的异常处理是搭建具有鲁棒性和可靠性系统的关键。避免出现上文中提出的反模式可以帮助你搭建一个可维护的、可适应变化的，且能与其他系统共同和谐工作的系统。