Java程序设计

第6章 异常处理

第6章 异常处理

- 本章介绍Java语言中的异常处理。
- 6.1 异常处理
- 6.2 自定义异常
- 6.3 断言及程序的测试
- 6.4 程序的调试

异常处理

异常

- 异常（exception）又称为例外、差错、违例
- 对应着Java运行错误处理机制

- 基本写法
  - try{
  - 语句组
  - }catch(Exception ex){
  - 异常处理语句组；
  - }
- 示例： ExceptionForNum.java

传统的语言如何处理

- 在一些传统的语言（如C语言中）
  - if语句来判断是否出现了例外
  - 全程变量ErrNo
- 但这有几个缺点
  - 正常处理与异常处理的代码同样处理
    - 可读性（readability）差
  - 每次调用一个方法时都进行错误检查
    - 可维护性（ maintainability ）差
  - 错误由谁处理不请
    - 职责不清

Java中的异常处理

- Java中处理异常
  - 抛出(throw)异常
  - 运行时系统在调用栈中查找
    - 从生成异常的方法开始进行回溯，直到找到：
  - 捕获(catch) 异常的代码

相关的语句

- 抛出异常
  - throw 异常对象；

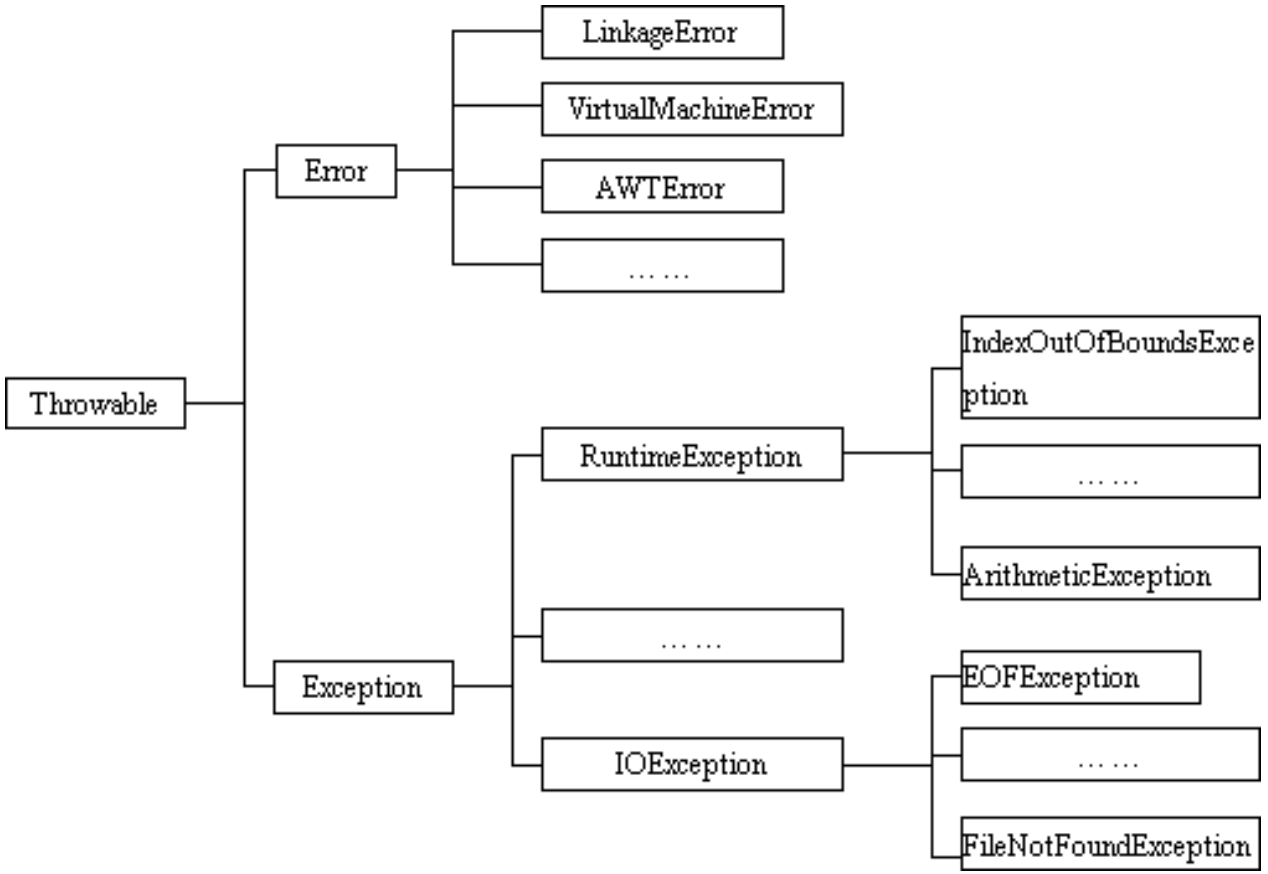
- 捕获异常
-

```
try{
    语句组
}catch(异常类名  异常形式参数名){
    异常处理语句组；
}catch(异常类名  异常形式参数名){
    异常处理语句组；
}finally{
    异常处理语句组；
}
```

其中，catch语句可以0至多个，可以没有finally语句

异常的分类

- Throwable
  - Error：JVM的错误
  - Exception：异常
- 一般所说的异常
- 是指Exception及其子类



Exception类

- Exception类
  - 构造方法
    - public Exception() ;
    - public Exception(String message) ;
    - Exception(String message, Throwable cause) ;
  - 方法
    - getMessage()
    - getCause()
    - printStackTrace()

多异常的处理

- 多异常的处理
  - 子类异常要排在父类异常的前面
- **finally语句**
  - 无论是否有异常都要执行
    - 即使其中有break,return等语句
    - 在编译时，finally部分代码生成了多遍
  - 例 TestTryFinally.java

受检的异常

- Exception分两种
  - RuntimeException及其子类，可以不明确处理
  - 否则，称为受检的异常（checked Exception）
- 受检的异常，要求明确进行语法处理
  - 要么捕（catch）
  - 要么抛（throws）：在方法的签名后面用throws xxxx来声明
    - 在子类中，如果要覆盖父类的一个方法，若父类中的方法声明了throws异常，则子类的方法也可以throws异常
    - 可以抛出子类异常（更具体的异常），但不能抛出更一般的异常
- 示例： ExceptionTrowsToOther.java

再谈try...with...resource

- try(类型 变量名 = new 类型（ ） ）{
  - 。。。
- }

- 自动添加了finally{ 变量.close(); }
  - 不论是否出现异常，都会执行
- 示例： TryWithResourcesTest.java

自定义异常类

创建用户自定义异常类

- 创建用户自定义异常时
- (1) 继承自Exception类或某个子Exception类
- (2) 定义属性和方法，或重载父类的方法
-

重抛异常及异常链接

- 对于异常，不仅要进行捕获处理，有时候还需要将此异常进一步传递给调用者，以便让调用者也能感受到这种异常。这时可以在catch语句块或finally语句块中采取以下三种方式：
- （1）将当前捕获的异常再次抛出：
  - throw e;
- （2）重新生成一个异常，并抛出，如：
  - throw new Exception("some message");
- （3）重新生成并抛出一个新异常，该异常中包含了当前异常的信息，如：
  - throw new Exception("some message"，e);
  - 可用e.getCause() 来得到内部异常
- 例：**ExceptionCause.java**

断言及程序的测试

断言

- 断言（assertion）
- assert的格式是：
  - assert 表达式;
  - assert 表达式：信息;
- 在调试程序时
  - 如果表达式不为true，则程序会产生异常，并输出相关的错误信息
- 示例：Assertion.java

Assert的编译及运行

- 编译
  - 只有在JDK1.4及以上的版本中才可以使用断言。
  - 具体地说，在早期的JDK版本(1.4)中编译时，要通过-source选项来指明版本，如：
  - javac -deprecation -source 1.4 -classpath . Assertion.java
- 运行
  - 在运行时，要使assert起作用，则在java命令中，使用选项(-ea，即-enableassertions)。如：
  - java -ea -classpath . Assertion

程序的测试及JUnit

- 程序的修改是经常要进行的过程，必须保证程序在修改后其结果仍然是正确的。
- 在编写程序代码的同时，还编写测试代码来判断这些程序是否正确。
- 这个过程称为“测试驱动”的开发过程。
- 从而保证了代码的质量，减少了后期的查错与调试的时间，所以实际上它提高了程序的开发效率。

JUnit

- 在Java的测试过程，经常使用JUnit框架
  - 参见http://www.junit.org。
- 现在大多数Java集成开发工具都提供了对JUnit的支持。
- 在Eclipse中
  - 项目右键—New— Junit Test Case
  - 项目右键—Run as — Junit Test
    - 测试通过则为绿色，不通过显示红色
- 在NetBeans中
  - 项目右键—新建— Junit测试
  - 运行—测试，或者直接按Alt+F6即可

测试函数

- @Test来标注测试函数
- 在测试中常用的语句如下：
  - fail(信息); //表示程序出错
  - assertEqauls(参数1，参数2); //表示程序要保证两个参数要相等
  - assertNull(参数); //表示参数要为null

```
@Test
public void testSum2() {
    HelloWorld a = new HelloWorld();
    assertEquals(a.sum(0, 100), 100);
    // fail("Not yet implemented");
}
```

程序的调试

程序中的错误

- 程序中的错误通常可以分成三大类
  - 语法错误（Syntax error）
    - 编辑、编译器发现
  - 运行错误（Runtime error）
    - 异常处理机制
  - 逻辑错误（Logic error）
    - 调试（debug）、单元测试（unit test）

程序调试

- 程序的调试（debug）
  - 在IDE中，项目上点右键 ，debug as...
  - 进入到调试视图(debug perspective)

调试的手段

- 调试的三种手段
  - 断点 (breakpoint)
  - 跟踪 (trace)
  - 监视 (watch)

断点

- 切换断点（toggle breakpoint）
  - 用鼠标单击（或右击）编辑器左边条
  - 或者
    - Eclipse　　Ctrl+Shift+B
    - NetBeans　Ctrl+F8
  -

跟踪

	Eclipse	NetBeans
逐语句执行	F5	F7
逐过程执行	F6	F8
跳出函数	F7	Ctrl+F7
运行到光标处	Ctrl+R	F4

监视

- 即时监视
  - 鼠标指向变量
- 快速监视
  - 点右键，Inspector
- 添加监视
  - 点右键，Watch
- 还可以看：调用堆栈等等