Java 异常处理机制的研究*)

杨厚群　陈　静
(海南大学信息科学技术学院　海口 570228)

摘　要　异常处理是 Java 语言的重要语言机制，正确、合理地处理异常对程序的可靠性、健壮性是十分重要的。在分析了异常处理技术的概念和思想后，对异常处理提出了指导原则，并针对检查型异常和非检查型异常的差异，探讨了对应的解决措施。

关键词　Java，Java 异常，异常处理

Research for Exception Handling Mechanism of Java

YANG Hou-Qun　CHEN Jing
(College of Information Science & Technology, Hainan University, Haikou 570228)

Abstract　Exception Handling is an important language mechanism of Java. Handling exceptions correctly and logically will help to write a reliable and robust program. The conception and thought of exception have been reviewed. An exception handling principle is presented. And how to handle checked and unchecked exceptions of Java is discussed.

Keywords　Java, Java exception, Exception handling

1　引言

异常(Exception)，又称为例外，是特殊的运行错误对象，对应着 Java 语言特定的运行错误处理机制，这种固定的机制用于识别和处理错误。高效率的异常处理机制能使程序更加健壮和更容易纠错。

在运行过程中，应用程序可能会遇到各种严重程度不同的错误，例如：当调用对象的方法后，对象可以发现内部状态的问题(变量值不一致)，检测对象或它操纵的数据(如文件或网络地址)的错误，判定是否和基本的约定冲突(如从已经关闭的流中读入数据)，等等。由于 Java 程序是在网络环境中运行的，安全已经成为需要首先考虑的重要因素之一。分析数据也表明，对异常不合适的处理会引起系统崩溃[1]。为了能够及时有效地处理程序中的运行错误，Java 语言中引入了异常概念和异常类，其最终目的就是要解决以下三个问题：发生了什么异常；在哪里出现异常；为什么会出现异常。

2　Java 异常处理机制

Java 异常的层次结构如下：

Throwable 是所有可以通过 throw 抛出或 catch 捕获错误的类的基类。它分为错误和异常。Error 表示编译时错误和系统错误，这些错误不需要客户程序去捕捉(特殊情况例外)，而 Exception 是所有异常的基类，这些异常包括了在调用 API 方法、用户自定义方法时，或者程序运行时发生意外所抛出的各类异常。

当 Java 程序违反 Java 语言的语义约束时，Java VM 把这个错误当作异常通知程序。比如违反语义：试图在数组的边界外对数组进行索引。某些程序设计语言及它们的应用程序对这种错误的反应就是强行终止程序的运行；还有一些语言则允许其应用程序以一种任意或不确定的方式来处理。这些方式都与 Java 语言的设计目的：提供可移植性和健壮性相悖。Java 提供了另一种不同的处理方式：当发生违反语义约束时，Java 将抛出异常，并做非局部控制转移，从异常的发生点转移到程序指定的异常处理点[2]。在发生点，称作异常被抛出；而在控制转移点，则称作异常被捕获。

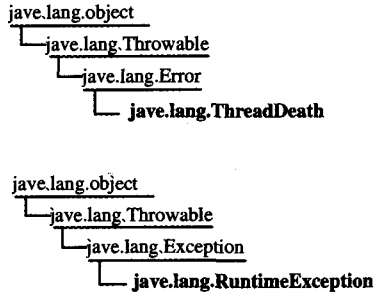

图 1　异常层次结构

所有的异常都由 Throwable 或者其子类的一个对象来表示，这种对象可用于把信息从异常发生点传递到捕捉点的处理程序中。异常句柄由 try 语句块中的 catch 子句建立。在处理异常的过程中，Java VM 把当前线程中已经开始运行但尚未结束的表达式、语句、方法、构造方法调用、静态初始化和域初始化表达式连续终止掉。这个过程一直继续下去，直到发现了一个异常句柄，该句柄通过指定异常的类或异常类的超类来声明它能处理该异常。如果未发现这样的句柄，就调用当前线程的父线程 ThreadGroup 的方法 uncaughtException，从而尽可能避免异常逃过处理。

由于一个 try 子句可能产生多种不同的异常，这就要求定义多个 catch 子句来实现多异常处理机制，每一个 catch 子句接收和处理一个异常句柄，至于一个异常能否被一个 catch 子句所接收取决于异常与该子句的异常参数匹配情况，它必

*)基金项目：海南省教育厅高校科研项目 Hjkj200603。**杨厚群**　硕士，副教授，主要研究方向为面向对象技术，数据仓库与数据挖掘。**陈　静**　硕士研究生，主要研究方向为人工智能。

须满足以下三个条件之一：

(1)异常与参数属于相同的异常类；

(2)异常属于参数异常类的子类；

(3)异常实现参数所定义的接口。

如果 try 子句产生的异常被第一个 catch 子句所接收，则程序的流程将直接跳转到这个 catch 子句中，语句块执行完毕后就退出调用的方法，try 子句中尚未执行的语句和其他的 catch 子句将被忽略；如果 try 子句产生的异常与第一个 catch 子句不匹配，系统自动转到第二个 catch 子句进行匹配，如果仍未匹配，就转向第三个、第四个…… 直到找到一个可以接收该异常句柄的 catch 子句，完成流程的跳转。

如果所有的 catch 子句都不能与抛出的异常匹配，说明当前方法不能处理这个异常句柄，程序流程将回溯到调用该方法的上层方法，如果这个上层方法中定义了与所产生的异常句柄相匹配的 catch 子句，流程将跳转到这个 catch 子句；否则继续回溯更上层的方法，如果所有的方法中都找不到可以匹配的 catch 子句，则有 Java 运行系统来处理这个异常。此时通常会中止程序的执行，退出虚拟机返回操作系统，在标准输出上打印相关的异常信息。

一个完全相反的情况就是假设 try 子句中所有语句的执行都没有引发异常，则所有的 catch 子句都会被忽略而不予执行。

Java 异常主要属于检查型异常，编译时，Java 语言对方法或构造函数的执行所能产生的检查型异常进行分析，检查程序是否包含检查异常的句柄。对于所有可能发生的检查异常，方法或构造函数的 throw 子句必须声明异常的类或异常的类的父类。标准的运行时异常和错误继承了 RuntimeException 和 Error 类，从而构成非检查型异常。用户创建的异常必须是继承 Exception 的检查型异常。

3 异常处理的原则

异常处理的大原则首先应该要树立起来：遇到每一种非正常情况抛出不同的异常；需要抛出异常时，应先选择一个已经定义的异常，否则自己创建一个；在异常类中，应该说明清楚引起异常的问题的类型；异常类不存在域或方法；异常必须是在要被抛出的 Java 包里创建。

3.1 什么时候抛出异常

一般来说，程序设计人员应该在方法或构造函数的设计中明确：一但碰到无法处理的不正常情况，就抛出异常。但如何区分不正常情况，这需要遵循这样一个原则：应该避免使用异常去处理本来方法可以自己解决的或正常功能之内的问题。从这个原则上可以清楚地得知，所谓不正常情况指的就是指方法的功能设计中不能解决的问题。比如下面的例子：

```
class Example1{
    public static void main(String[]args)
        throws IOException{
        if (args. length==0){
            System. out. println("Must give filename as first arg. ");
            return;
        }
        FileInputStream in;
        try{
            in=new
FileInputStream(args[0]);
        }
        catch (FileNotFoundException e){
            System. out. println("Can't find file:"+args[0]);
            return;
        }
        int ch;
        while((ch=in. read())! =-1){
            System. out. print((char) ch);
            }
            System. out. println();
            in. close();
        }
}
```

上面的例子表明 FileInputStream 的 read()方法并没有通过抛出异常来处理读到文件尾的问题，而是通过返回值－1来进行判断处理。类似这种读到文件尾的情况应该是属于正常范围的，无须通过抛出异常来处理。下面的这个有关 DataInputStream 的范例则采取了不同的处理方式：

```
class Example2{
    public static void main(String[]args)
throws IOException {
        if(args. length==0){
            System. out. println("Must give filename as first arg. ");
            return;
        }
        FileInputStream fin;
        try{
            fin=new
FileInputStream(args[0]);
        }catch (FileNotFoundException e)
{
            System. out. println("Can't find file:"+args[0]);
            return;
            }
            DataInputStream din=new
DataInputStream(fin);
            try{
              int i;
              for(;;){
                i=din. readInt();
                System. out. println(i);
              }
            }catch (EOFException e){
              }
            fin. close();
    }
}
```

每当 readint()方法被调用时，就从 stream 中读 4 个字节并转换成整型数据，一旦碰到文件尾，readint()就抛出异常。之所以这么做，原因是它无法像前一个例子那样得到一个特殊的返回值(如－1)提示到了文件尾部，而且最后读取的字节数也无法保证一定是 4 个，碰到这样的情况，只能抛出异常，并且应该是检查型异常，客户端程序必须处理这个异常。

3.2 抛出什么样的异常

抛出异常的关键是抛出异常类的选择，我们可以抛出 Java API 中定义好的异常，也可以抛出我们自定义的异常。那么究竟抛出什么异常，这跟我们程序设计的要求有关，由于我们无法全面考虑程序运行可能涉及到的所有异常，而且客户端对异常的处理要求也不一致，这就需要对异常的设计确定原则：第一，异常必须分层次；第二，注意区别异常和错误；第三，合理使用可检查和非检查型异常；第四，必须使用 finally 子句恢复释放内存之外的资源设置。

在异常设计的层次处理上，注意不要把解决特定情况的检查型异常笼统地提到更高的异常解决层次[4]，比如，我们定义 SQLException 为 database access 异常类，定义 BatchUpdateException 类为 SQLException 类的子类，定义一个 getUpdateCounts 的方法，一般说来，会声明它抛出一个 BatchUpdateException，但如果写成 SQLException 类或 Exception 类，这样客户端很可能无法判断异常是因为什么原因抛出的。

相反，如果客户端不关心具体的异常，或者没有必要了解异常的细节，客户端程序仅仅希望运行过程中不受抛出异常的影响，那么，正确的做法应该是将 BatchUpdateException 异常转换成其它的检查型异常或者转换成一个非检查型异常；然而，大多数情况下，客户端几乎不会针对类似下面的例子这样的异常做进一步处理：

```
public void dataAccessCode(){
    try{
```

```
        .. some code that throws BatchUpdateException
}catch
(BatchUpdateException ex){
        ex. printStacktrace();
    }
}
```

可以看出，catch 模块几乎没做异常处理，仅仅输出了异常及原因，所以在异常设计时候，应毫不犹豫将这样的异常转换成非检查型异常，使整个代码更为清晰易懂。

```
public void dataAccessCode(){
    try{
        .. some code that throws
BatchUpdateException
    }catch(BatchUpdateException ex){
        throw new RuntimeException(ex);
    }
}
```

BatchUpdateException 异常转换成了 RuntimeException 异常，如果 BatchUpdateException 异常被抛出，catch 模块将会抛出一个新的 RuntimeException 异常，此时执行线程就会被挂起，系统报告异常。通常情况下，如果程序不需要处理 BatchUpdateException 异常就没必要让异常处理中断客户端进程，除非客户端设计能够解决 BatchUpdateException 异常，把这种异常转换成更有意义的检查型异常或抛出类似 RuntimeException 非检查型异常由系统负责处理更为合理。

一般来说，程序抛出应该是异常而不是错误，错误 Error 类是 Throwable 类的子类，用于严重的错误，比如 OutofMemoryError，它们都是由 Java VM 负责报告。但也有类似 java. awt. AWTError 的错误可以被 Java API 抛出。在程序中必须严格限制，抛出的应该是 Exception 类下的异常，错误则交给系统处理。

抛出检查型异常或非检查型异常本身就是一个见仁见智的问题，一个检查型异常来源于 Exception 类的一些子类（或者是 Exception 类本身），而不是 RuntimeException 类或其子类。非检查型异常则来源于 RuntimeException 类或其子类，错误 Error 类和其子类也属于非检查型异常。程序在需要抛出异常的时候，开发人员应该决定是抛出类似 RuntimeException 的非检查型异常还是抛出 Exception 的检查型异常。

如果程序需要抛出一个检查型异常，那么程序在定义方法时要声明这个异常。客户端程序在调用这个方法时就需要做好捕捉和处理这个异常的准备，当然也可以在方法的异常列表中声明这个异常。之所以决定抛出检查型异常，用意就在于强迫客户端程序必须做好处理异常的准备。

如果程序抛出的非检查型异常，客户端程序可以决定是否去捕捉或是忽略这个异常，如同处理一个检查型异常。对于非检查型异常来说，编译器并不会强迫客户端程序必须去捕捉它，或是在异常列表中声明它。实际上，客户端程序甚至无须知道异常是否会抛出。也就是说，客户端程序几乎可以不用考虑如何处理非检查型异常，这点不同于处理检查型异常。

如果抛出一个异常只是指明类的不合理调用，那么应该使用非检查型异常。例如 String's charAt() 方法抛出的 StringIndexOutOfBoundsException 异常是非检查型异常，String 类设计的本意并非要强迫客户端程序在调用 charAt (int index) 方法时都要准备去处理因为无效的 index 参数而抛出的异常。

而 java. io. FileInputStream 类的方法 read() 抛出的异常 IOException 是一个检查型异常，它的作用就是指明在读取文件时引发异常的原因，也就是告知 read() 方法无法满足从文件读取下一个字节的要求，而不是试图表明客户端程序不正确使用了 FileInputStream 类。FileInputStream 类设计已经考虑到类似这样的异常情况会经常发生而且非常重要，因此要求客户端程序必须去处理它。

从上述的分析可以得出这样一个结论：由于某些原因导致程序抛出异常，如果程序被认定必须处理这样的异常，那么设计抛出的应该是检查型的异常，否则就抛出非检查型异常。

3.3 善用 finally 子句

Java 资源中，内存的释放和回收是通过系统的垃圾回收机制自动完成的。但是，内存以外的资源，比如一个打开的文件，网络的连接或者是屏幕上图画等，如果在程序运行过程中由于可能抛出的异常没有被捕获，就有可能无法被正常释放。然而，使用 finally 子句，可以把所有的释放资源的处理写在其中，这样无论发生了什么情况，都能确保资源正确释放，因为 finally 子句能够在回溯机制发生作用前得到执行[3]。例如下面的程控开关程序：

```
public static void main(String[]args){
  try{
    sw. on();
    f();
    sw. off();
  } catch(OnOffException1 e){
    System. err. println("OnOffException1");
    sw. off();
  } catch(OnOffException2 e) {
    System. err. println("OnOffException2");
    sw. off();
  }
}
```

程序的目的是确保 main() 方法结束的时候，开关处于关闭状态，所以 sw. off() 被置于 try 子句以及每个异常处理程序的末尾。但程序仍然有可能会抛出一个没被捕获的异常，所以 sw. off() 还是有可能被漏掉。然而使用了 finally 之后，就可以把 try 子句里面的清理代码全部集中到一个地方，finally 块确保 sw. off() 得到执行，即便是所有异常都没有被这组 catch 子句所捕获，finally 子句也会在回溯机制发生作用前得到执行：

```
public static void main(String[]args) {
  try{
    sw. on();
    f();
  } catch(OnOffException1 e) {
    System. err. println("OnOffException1");
  } catch(OnOffException2 e) {
    System. err. println("OnOffException2");
  } finally {
      sw. off();
  }
}
```

即便在程序中存在 break 和 continue 语句，finally 子句也会得到执行。但是，finally 子句也会带来麻烦[5]。比如，在 finally 块中的回收资源的方法抛出异常怎么办？一个非常典型的例子就是关闭流。假设客户希望在处理流的代码中如果出现异常也可以保证流被安全地关闭：

```
InputStream in;
try{
    .. some code that throws exception
}catch(IOException e){
    .. show error message
}finally{
    in. close();
}
```

假如在 try 块中的代码抛出一个非 IOException 的异常，而调用该代码的调用者会处理该异常，则 finally 块会被执行，因而 close() 会被调用。而 close() 本身也是会抛出异常的。如果出现这种极端的情况，则原始的异常会丢失，而抛出 IOException，而这并非异常处理机制所期望的好结果。解决

的办法惟有在 finally 块中使用清除操作时不抛出异常，如 dispose()，close()。Java 语言没有所谓的“析构函数”(destructor)，因此在 Java 语言中不存在自动资源回收功能。只能使用 finally 子句通过人工设置代码回收极少数必须手工回收的资源。

4 再论检查型异常和非检查型异常

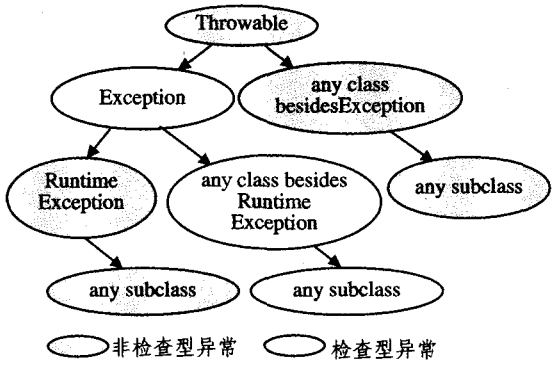

图 2 非检查型异常与检查型异常

异常处理的重要准则就是：如果你不知道如何处理这个异常，就不要去捕捉它。异常处理的目标就是把处理异常的代码同正常流程的代码区别开来，这样程序就不会被异常处理这样的枝节问题纠缠，更易于理解和维护。

但是，Java 的检查型异常强迫客户程序必须在 try 子句后加上 catch 子句捕捉异常，如果仅仅是静态类型的问题检查，检查型异常也许是必要的，可是开发人员出于多一事不如少一事的心理，有时只是简单地打印栈轨迹应付了事，这样编译就通过了，可是异常就完全被忽略掉了。在 C++ 和 Python 语言中，所有异常都是非检查型的，异常一旦被抛出，你可以捕捉它进行处理，也可以根本不用去理睬它，程序似乎也不受影响。

异常的作用应该体现在：(1)提供一个标准统一的机制报告错误；(2)允许客户程序不必关心异常处理。而 Java 提供的检查型异常似乎违反了这样的准则，因为太多的客户程序利用 catch 子句忽略掉了许多的异常，这和检查型异常设置的初衷是相违背的。开发人员习惯于在编译时发现异常，处理异常，并且认为这是可靠、安全的方式，一旦异常出现在运行时，总认为是不可靠的，其实这是个误解。

非检查型异常允许客户程序根据需要决定是否捕捉它，系统总会捕捉非检查型异常，这使得正常工作的代码可以写得更为清晰和有条理，而不是跟异常处理代码纠缠在一起。下面是一个将检查型异常转换成非检查型异常 RuntimeException 的例子：

```
import java. io. * ;
class ExceptionAdapter extends
RuntimeException{
    private final String stackTrace;
    public Exception originalException;
    public ExceptionAdapter(Exception e){
      super(e. toString());
      originalException=e;
      StringWriter sw = new StringWriter();
      e. printStackTrace(new
PrintWriter(sw));
      stackTrace = sw. toString();
    }
    public void printStackTrace(){
      printStackTrace(System. err);
    }
    public void rethrow(){throw
originalException;}
}
    ......
    catch(ExceptionAdapter ea){
      try {
        ea. rethrow();
      } catch(IllegalArgumentException e)
{
        // ...
      } catch(FileNotFoundException e){
        // ...
      }
      // etc.
    }
```

从例子中开发人员仍然能够捕捉特定的异常，但不必在程序的任何地方设置 try 和 catch 子句捕捉异常，甚至如果忘了捕捉，异常也会被提交到更高层次的场合进行处理。

(下转第 292 页)

(上接第 285 页)

C_d)，如图中的粗直线 u_1T 所示。而角色 r 在 t_3 时刻失效，故而在(t_3，t_4)时段是 $disabled(r, nil.(t_4-t_3).C_d)$；$u_1$ 和 r 之间的指派关系也失效了。从 t_4 开始，角色 r 再次有效 $enable(r)$，u_1 能够再次激活 r；但是 u_1 不能通过条件集 $COND_2$ 的执行(FALSE)来激活角色 r，对于用户 u_1 来说，角色 r 只是有效状态，如图中的粗虚线 u_1F 所示。直到会话 s_1 结束(t_5 时刻)，u_1 都没有机会激活角色 r。

用户 u_2 的会话 s_2 在 t_2 开始，虽然在(t_2，t_3)时段，角色 r 是 $enabled(r, COND_1.(t_3-t_2).C_d)$，但是 u_2 不能通过条件集 $COND_1$ 的执行(FALSE)激活角色 r，如图中的粗虚线 u_2F 所示。同样的角色 r 在(t_3，t_4)时段是 $disabled(r, nil.(t_4-t_3).C_d)$，$u_2$ 没有权利激活角色 r。在(t_4，t_6)时段，角色 r 是 $enabled(r, COND_2.(t_6-t_4).C_d)$，且 u_2 能够通过条件集 $COND_2$ 的执行(TRUE)，因此角色 r 在该会话中的状态是 $s\_actived(r, u_2, s_2, COND_2.(t_6-t_4).C_d)$，如图中的粗直线 u_2T 所示。在 t_6 时刻，角色 r 再次失效，u_2 和 r 之间的指派关系也随之失效，直至该会话 s_2 结束(t_7 时刻)，u_2 都不能激活角色 r。

结束语 本文在研究周期理论和时态 RBAC 模型的基础上，提出了条件周期表达式和条件时态的概念，将模型的时间维控制因子扩展为条件时间平面的控制因子，从而提高了模型控制的灵活性和多样性。通过对条件周期表达式、条件周期事件和条件时态下的角色状态断言论述，形式化地描述了条件时态的原理和控制机制。

参 考 文 献

1 Ferraiolo D, Cugini J, Kuhn D R. Role Based Access Control (RBAC): Features and Motivations. In: Proc. 1995 Computer Security Applications Conference, December 1995. 241～248
2 Ferraiolo D, Sandhu R, Gavrila S, et al. A Proposed Standard for Role Based Access Control. ACM Transactions on Information and System Security, August 2001. 224～274
3 St'evenne J-M. A model-checking approach to temporal reasoning. In: the Second Bar-Ilan Symposium on Foundation of Artificial Intelligence, January 1991
4 Ni'ezette M, St'evenne J-M. An efficient symbolic representation of periodic time. In: International Conference on Information and Knowledge Management, 1992
5 Bertino E, Bonatti P A, Ferrari E. TRBAC: A temporal role-based access control model. ACM Trans. on Information and System Security, 2001, 4(3): 191～233
6 Joshi J B D, Bertino E, Latif U, et al. Generalized Temporal Role-Based Access Control Model. IEEE Transactions on Knowledge and Data Engineering, 2005, 17(1): 4～23
7 Sandhu R, Coyne E, Feinstein H, et al. Role-Based Access Control Models. Computer, 1996, 29(2): 38～47