第9章　异 常 处 理

尽管人人都希望所处理的事情能顺利执行，所操作的机器能正常运转，但在现实生活中总会遇到各种异常情况。例如职工小王开车去上班，在正常情况下，小王会准时到达单位。但是天有不测风云，在小王去上班时，可能会遇到一些异常情况：比如小王的车子出了故障，小王只能改为步行，结果上班迟到；或者遭遇车祸而丧命。

异常情况会改变正常的流程，导致恶劣的后果。为了减少损失，应该事先充分预计所有可能出现的异常，然后采取以下解决措施：

（1）首先考虑避免异常，彻底杜绝异常的发生；如果不能完全避免，则尽可能地减少异常发生的几率。例如，车祸是无法完全避免的，但是通过建立完善的交通规则，并且强化驾驶员的安全意识，可以减少发生车祸的几率。

（2）如果有些异常不可避免，那么应该预先准备好处理异常的措施，从而降低或弥补异常造成的损失，或者恢复正常的流程。例如，车子里备有安全带和安全气囊，当车子遭到猛烈撞击时，这些安全设备能够尽可能地保护当事人的安全；再例如，大楼里备有灭火器，当火灾发生时，可立刻用来灭火。

（3）对于某个系统遇到的异常，有些异常单靠系统本身就能处理，有些异常需要系统本身及其他系统共同来处理。例如，大楼里发生严重火灾，单靠大楼里的灭火器无法完全灭火，还需要消防队的参与才能灭火；还有些异常系统本身不能处理，完全依靠其他系统来处理。

（4）对于某个系统遇到的异常，系统本身应该尽可能地处理异常，实在没办法处理，才求助于其他系统来处理。因为一般来说，异常处理得越早，损失就越小。否则，如果异常传播到多个系统，会引起连锁反应，从而造成更大的损失。例如，船上撞了个洞，由于没有处理这个异常，结果海水渗入船内，最后船沉没了，船上所有的人都丧命，许多遇难者的家属因为悲痛过度而病倒，负责赔偿的一家小型保险公司宣布破产。

程序运行时也会遇到各种异常情况，异常处理的原则和现实生活中异常处理的原则相似，首先应该预计到所有可能出现的异常，然后考虑能否完全避免异常，如果不能完全避免，再考虑异常发生时的具体处理办法。

Java 语言提供了一套完善的异常处理机制。正确运用这套机制，有助于提高程序的健壮性。所谓程序的健壮性，就是指程序在多数情况下能够正常运行，返回预期的正确结果；如果偶而遇到异常情况，程序也能采取周到的解决措施。而不健壮的程序则没有事先充分预计到可能出现的异常，或者没有提供强有力的异常解决措施，导致程序在运行时，经常莫名其妙地终止，或者返回错误的运行结果，而且难以检测出现异常的原因。

9.1 Java 异常处理机制概述

要在程序中处理异常，主要考虑两个问题：（1）如何表示异常情况？（2）如何控制处理异常的流程？本节首先把传统的异常处理方式与 Java 异常处理机制做了比较，分析了后者的优点；接着通过介绍 Java 虚拟机的方法调用栈，来剖析 Java 的异常处理的基本原理；最后分析了异常处理对程序运行性能的影响。

9.1.1 Java 异常处理机制的优点

在一些传统的编程语言，如 C 语言中，并没有专门处理异常的机制，程序员通常用方法的特定返回值来表示异常情况，并且程序的正常流程和异常流程都采用同样的流程控制语句。

例程 9-1（Car.java）和例程 9-2（Worker.java）演示了传统的异常处理方式。为了简化起见，仅仅考虑了职工开车上班时车子出故障的异常情况。

例程 9-1　Car.java

```
public class Car{
   public static final int OK=1;                //正常情况
   public static final int WRONG=2;             //异常情况

   public int run(){
      if(车子没出故障)                           //正常流程
         return OK;
      else   //异常流程
         return WRONG;
   }
}
```

例程 9-2　Worker.java

```
public class Worker{
   private Car car;

   public static final int IN_TIME=1;           //正常情况，准时到达单位
   public static final int LATE=2;              //异常情况，上班迟到

   public Worker(Car car){this.car=car;}

   /**  开车去上班 */
   public int gotoWork(){
      if(car.run()==Car.OK)                     //正常流程
         return IN_TIME;
      else{                                     //异常流程
         walk();
         return LATE;
      }
   }
```

```
    /** 步行去上班 */
    public void walk(){}
}
```

以上传统的异常处理方式尽管是有效的，但存在以下缺点：

- 表示异常情况的能力有限，单靠方法的返回值难以表达异常情况包含的所有信息。例如，对于上班迟到这种异常，相关的信息包括：迟到的具体时间和迟到的原因等。
- 异常流程的代码和正常流程的代码混合在一起，影响程序的可读性，容易增加程序结构的复杂性。
- 随着系统规模的不断扩大，这种处理方式已经成为创建大型可维护应用程序的障碍。

Java 语言按照面向对象的思想来处理异常，使得程序具有更好的可维护性。Java 异常处理机制具有以下优点：

- 把各种不同类型的异常情况进行分类，用 Java 类来表示异常情况，这种类被称为异常类。把异常情况表示成异常类，可以充分发挥类的可扩展和可重用的优势。
- 异常流程的代码和正常流程的代码分离，提高了程序的可读性，简化了程序的结构。
- 可以灵活地处理异常，如果当前方法有能力处理异常，就捕获并处理它，否则只需抛出异常，由方法调用者来处理它。

例程 9-3（CarWrongException.java）、例程 9-4（LateException.java）、例程 9-5（Car.java）和例程 9-6（Worker.java）演示了 Java 异常处理机制。CarWrongException 和 LateException 类为异常类，分别表示车子出故障和上班迟到这两种异常情况。在 Car 类的 run()方法中有可能抛出 CarWrongException 异常，在 Worker 类的 gotoWork()方法中有可能抛出 LateException 异常。

例程 9-3　CarWrongException.java

```
/** 表示车子出故障的异常情况 */
public class CarWrongException extends Exception{
    public CarWrongException(){}
    public CarWrongException(String msg){super(msg);}
}
```

例程 9-4　LateException.java

```
import java.util.Date;

/** 表示上班迟到的异常情况 */
public class LateException extends Exception{
   private Date arriveTime;   //迟到的时间
   private String reason;   //迟到的原因

   public LateException(Date arriveTime,String reason){
      this.arriveTime=arriveTime;
```

```
        this.reason=reason;
    }

    public Date getArriveTime(){return arriveTime;}
    public String getReason(){return reason;}
}
```

例程 9-5　Car.java

```
public class Car{
    public void run()throws CarWrongException{
        //如果车子出故障，就创建一个 CarWrongException 对象，并将其抛出
        if(车子无法刹车)throw new CarWrongException("车子无法刹车");
        if(发动机无法启动)throw new CarWrongException("发动机无法启动");
    }
}
```

例程 9-6　Worker.java

```
public class Worker{
    private Car car;
    public Worker(Car car){this.car=car;}

    public void gotoWork()throws LateException{
        try{
            car.run();
        }catch(CarWrongException e){ //处理车子出故障的异常
            walk();
            Date date=new Date(System.currentTimeMillis());
            String reason=e.getMessage();
            throw new LateException(date,reason); //创建一个 LateException 对象，并将其抛出
        }
    }

    public void walk(){}
}
```

9.1.2　Java 虚拟机的方法调用栈

Java 虚拟机用方法调用栈（method invocation stack）来跟踪每个线程中一系列的方法调用过程，关于线程的知识参见第 13 章（多线程与并发）。该堆栈保存了每个调用方法的本地信息（比如方法的局部变量）。每个线程都有一个独立的方法调用栈。对于 Java 应用程序的主线程，堆栈底部是程序的入口方法 main()。当一个新方法被调用时，Java 虚拟机把描述该方法的栈结构置入栈顶，位于栈顶的方法为正在执行的方法。图 9-1 描述了方法调用栈的结构。在图 9-1 中，方法的调用顺序为：main()方法调用 methodB()方法，methodB()方法调用 methodA()方法。

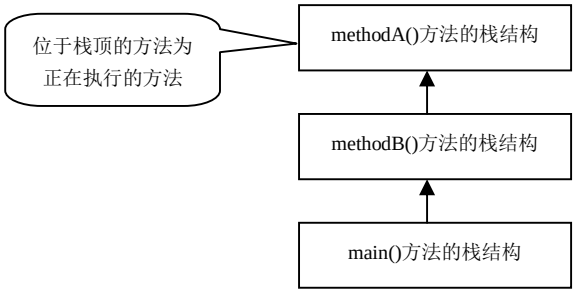


图 9-1　Java 虚拟机的方法调用栈

Tips

当 methodB()方法调用 methodA()方法时，为了叙述方便，本书有时把 methodB()称为 methodA()的方法调用者。

如果方法中的代码块可能抛出异常，有两种处理办法：

（1）在当前方法中通过 try-catch 语句捕获并处理异常，例如：

```
public void methodA(int money){
   try{
         //以下代码可能会抛出 SpecialException
         if(--money<=0) throw new SpecialException("Out of money");
   }catch(SpecialException e){
      处理异常
   }
}
```

（2）在方法的声明处通过 throws 语句声明抛出异常，例如：

```
public void methodA(int money) throws SpecialException{
   //以下代码可能会抛出 SpecialException
   if(--money<=0) throw new SpecialException("Out of money");
}
```

当一个方法正常执行完毕后，Java 虚拟机会从调用栈中弹出该方法的栈结构，然后继续处理前一个方法。如果在执行方法的过程中抛出异常，Java 虚拟机必须找到能捕获该异常的 catch 代码块。它首先查看当前方法是否存在这样的 catch 代码块，如果存在，就执行该 catch 代码块；否则，Java 虚拟机会从调用栈中弹出该方法的栈结构，继续到前一个方法中查找合适的 catch 代码块。

例如，当 methodA()方法抛出 SpecialException 时，如果在该方法中提供了捕获 SpecialException 的 catch 代码块，就执行这个异常处理代码块。如果 methodA()方法未捕获该异常，而是采用第二种方式声明抛出 SpecialException，那么 Java 虚拟机的处理流程将退回到上层调用方法 methodB()，再查看 methodB()方法有没有捕获 SpecialException。如果在 methodB()方法中存在捕获该异常的 catch 代码块，就执行这个 catch 代码块，此时 methodB()方法的定义如下：

```
public void methodB(int money) {
   try{
      methodA(money);
```

```
    }catch(SpecialException e){
      处理异常
    }
  }
```

由此可见，在回溯过程中，如果 Java 虚拟机在某个方法中找到了处理该异常的代码块，则该方法的栈结构将成为栈顶元素，程序流程将转到该方法的异常处理代码部分继续执行。

如果 methodB()方法也没有捕获 SpecialException，而是声明抛出该异常，Java 虚拟机的处理流程将退回到 main()方法，此时 methodB()方法的定义如下：

```
public void methodB(int money) throws SpecialException{
  methodA(money);
}
```

当 Java 虚拟机追溯到调用栈的最底部的方法时，如果仍然没有找到处理该异常的代码块，将按以下步骤处理：

（1）调用异常对象的 printStackTrace()方法，打印来自方法调用栈的异常信息。例如运行例程 9-7（Sample.java），将打印如下异常信息：

```
Exception in thread "main" SpecialException: Out of money
        at Sample.methodA(Sample.java:3)
        at Sample.methodB(Sample.java:7)
        at Sample.main(Sample.java:11)
```

（2）如果该线程不是主线程，那么终止这个线程，其他线程继续正常运行。如果该线程是主线程（即方法调用栈的最底部为 main()方法），那么整个应用程序被终止。

例程 9-7　Sample.java

```
public class Sample {
  public void methodA(int money)throws SpecialException{
    if(--money<=0) throw new SpecialException("Out of money");
    System.out.println("methodA");
  }
  public void methodB(int money) throws SpecialException{
    methodA(money);
    System.out.println("methodB");
  }
  public static void main(String args[])throws SpecialException{
    new Sample ().methodB(1);
  }
```

SpecialException 类表示某种异常，例程 9-8 是它的源程序。

例程 9-8　SpecialException.java

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

```
    }
```

9.1.3　异常处理对性能的影响

一般说来，在 Java 程序中使用 try-catch 语句不会对应用的性能造成很大的影响。仅当异常发生时，Java 虚拟机需要执行额外的操作，来定位处理异常的代码块，这时会对性能产生负面影响。如果抛出异常的代码块和捕获异常的代码块位于同一个方法中，那么这种影响就会小一些；如果 Java 虚拟机必须搜索方法调用栈来寻找异常处理代码块，对性能的影响就比较大了。尤其是当异常处理代码块位于调用栈的最底部时，Java 虚拟机定位异常处理代码块需要大量的工作。

所以不应该使用异常处理机制来控制程序的正常流程，而应该确保仅仅在程序中可能出现异常的地方使用 try-catch 语句。此外，应该使异常处理代码块位于适当的层次，如果当前方法具备处理某种异常的能力，就尽量自行处理，不要把自己可以处理的异常推给方法调用者去处理。

9.2　运用 Java 异常处理机制

上一节介绍了 Java 异常处理机制，本节介绍如何在应用程序中运用这种机制，来处理实际的异常情况。

9.2.1　try-catch 语句：捕获异常

在 Java 语言中，用 try-catch 语句来捕获异常。格式如下：

```
try {
    可能会出现异常情况的代码
} catch (SQLExcetion e) {
    处理操作数据库出现的异常
} catch (IOException e) {
   处理操作输入流和输出流出现的异常
}
```

在例程 9-9（MainCatcher.java）中，当 methodA()方法抛出 SpecialException 时，流程退回到 methodB()方法，由于 methodB()方法未捕获该异常，流程继续退回到 main()方法，main()方法提供了处理该异常的 catch 代码块，因此 main()方法的正常流程被中断，Java 虚拟机跳转到该 catch 代码块，执行处理异常的代码。

例程 9-9　MainCatcher.java

```
public class MainCatcher{
   public void methodA(int money)throws SpecialException{
      if(--money<=0) throw new SpecialException("Out of money");
      System.out.println("methodA");
   }
   public void methodB(int money) throws SpecialException{
```

```
        methodA(money);
        System.out.println("methodB");
    }
    public static void main(String args[]){
        try{
            new MainCatcher().methodB(1);
            System.out.println("main");
        }catch(SpecialException e){
            System.out.println("Wrong");
        }
    }
}
```

以上程序的打印结果为：Wrong。如果把 main()方法中的 methodB(1)改为 methodB(2)，就会按正常流程执行，程序的打印结果为：

```
methodA
methodB
main
```

9.2.2 finally 语句：任何情况下必须执行的代码

由于异常会强制中断正常流程，这会使得某些不管在任何情况下都必须执行的步骤被忽略，从而影响程序的健壮性。例如，小王开了一家小店，在店里上班的正常流程为：打开店门、工作 8 个小时、关门。异常流程为：小王在工作时突然犯病，因而提前下班。以下 work()方法表示小王的上班行为：

```
public void work()throws LeaveEarlyException {
    try{
        开门
        工作 8 个小时              //可能会抛出 DiseaseException 异常
        关门
    }catch(DiseaseException e){
        throw new LeaveEarlyException();
    }
}
```

假如，小王在工作时突然犯病，那么流程会跳转到 catch 代码块，这意味着关门的操作不会被执行，这样的流程显然是不安全的，必须确保关门的操作在任何情况下都会被执行。在程序中，应该确保占用的资源被释放，比如及时关闭数据库连接，关闭输入流，或者关闭输出流。finally 代码块能保证特定的操作总是会被执行，它的形式如下：

```
public void work()throws LeaveEarlyException {
    try{
        开门
        工作 8 个小时              //可能会抛出 DiseaseException 异常
    }catch(DiseaseException e){
        throw new LeaveEarlyException();
    }finally{
        关门
    }
```

```
    }
```

不管 try 代码块中是否出现异常，都会执行 finally 代码块。例程 9-10（WithFinally.java）的 main()方法的 try 代码块后面跟了 finally 代码块：

例程 9-10　WithFinally.java

```
public class WithFinally {
   public void methodA(int money)throws SpecialException{
      if(--money<=0) throw new SpecialException("Out of money");
      System.out.println("methodA");
   }
   public static void main(String args[]){
       try{
           new WithFinally().methodA(1);   //抛出 SpecialException 异常
           System.out.println("main");
       }catch(SpecialException e){
           System.out.println("Wrong");
       }finally{
          System.out.println("Finally");
       }
   }
}
```

以上程序的打印结果为：

```
Wrong
Finally
```

如果把 main()方法中的“methodA(1)”改为“methodA(2)”，程序将正常运行，打印结果为：

```
methodA
main
Finally
```

在以下程序代码中，把打印“Finally”的操作放在 try-catch 语句的后面，这也能保证这个操作被执行：

```
public static void main(String args[]){
   try{
       new WithFinally().methodA(1);   //抛出 SpecialException 异常
       System.out.println("main");
   }catch(SpecialException e){
       System.out.println("Wrong");
   }

   System.out.println("Finally");
}
```

以上处理方式尽管在某些情况下是可行的，但不值得推荐，因为它有两个缺点：

（1）把与 try 代码块相关的操作孤立开来，使程序结构松散，可读性差。

（2）影响程序的健壮性。假如 catch 代码块继续抛出异常，就不会执行打印“Finally”的操作，例如以下 catch 代码块继续抛出异常，这将导致 main()方法异常终止，程序的

打印结果为“java.lang.Exception: Wrong”：

```
public static void main(String args[])throws Exception{
   try{
      new WithFinally().methodA(1);   //抛出 SpecialException 异常
      System.out.println("main");
   }catch(SpecialException e){
      throw new Exception("Wrong");
   }
   System.out.println("Finally");   //不会被执行
}
```

9.2.3 throws 子句：声明可能会出现的异常

如果一个方法可能会出现异常，但没有能力处理这种异常，可以在方法声明处用 throws 子句来声明抛出异常，例如，汽车在运行时可能会出现故障，汽车本身没办法处理这个故障，因此 Car 类的 run()方法声明抛出 CarWrongException：

```
public void run()throws CarWrongException{
   if(车子无法刹车)throw new CarWrongException("车子无法刹车");
   if(发动机无法启动)throw new CarWrongException("发动机无法启动");
}
```

Worker 类的 gotoWork()方法调用以上 run()方法，gotoWork()方法捕获并处理 CarWrongException 异常，在异常处理过程中，又生成了新的异常 LateException，gotoWork()方法本身不会再处理 LateExeption，而是声明抛出 LateExeption：

```
public void gotoWork()throws LateException{
   try{
      car.run();
   }catch(CarWrongException e){   //处理车子出故障的异常
      walk();
      Date date=new Date(System.currentTimeMillis());
      String reason=e.getMessage();
      throw new LateException(date,reason); //创建一个 LateException 对象，并将其抛出
   }
}
```

Tips

谁会来处理 Worker 类的 LateException 呢？显然是职工的老板，如果某职工上班迟到，老板就会批评他，甚至扣他的工资。

一个方法可能会出现多种异常，throws 子句允许声明抛出多个异常，例如：

```
public void method() throws SQLException,IOException{…}
```

异常声明是接口的一部分，在 JavaDoc 文档中应描述方法可能会抛出某种异常的条件。根据异常声明，方法调用者了解到被调用方法可能抛出的异常，从而采取相应的措施：捕获并处理异常，或者声明继续抛出异常。

9.2.4　throw 语句：抛出异常

throw 语句用于抛出异常，例如，以下代码表明汽车在运行时会出现故障：

```
public void run()throws CarWrongException{
   if(车子无法刹车)
      throw new CarWrongException("车子无法刹车");
   if(发动机无法启动)
      throw new CarWrongException("发动机无法启动");
}
```

再例如，游泳馆的救生人员负责保护游泳者的安全，如果出现有人溺水的异常，就先进行急救，比如人工呼吸，假如溺水者立刻恢复正常，那么异常处理完毕，否则只能继续抛出该异常，由医院来处理它：

```
try{
   巡察是否有人溺水
}catch(DrownException e){
   人工呼吸
   if(溺水者未恢复正常)
      throw e;   //继续抛出溺水异常
}
```

值得注意的是，由 throw 语句抛出的对象必须是 java.lang.Throwable 类或者其子类的实例。以下代码是不合法的：

```
throw new String("有人溺水啦，救命啊!");  //编译错误，String 类不是异常类型
```

Tips

throws 和 throw 关键字尽管只有一个字母之差，却有着不同的用途，注意不要将两者混淆。

9.2.5　异常处理语句的语法规则

异常处理语句主要涉及 try、catch、finally、throw 和 throws 关键字，要正确使用它们，就必须遵守必要的语法规则。

（1）try 代码块后面可以有零个或多个 catch 代码块，还可以有零个或至多一个 finally 代码块。如果 catch 代码块和 finally 代码块并存，finally 代码块必须在 catch 代码块后面。

（2）try 代码块后面可以只跟 finally 代码块，例如：

```
public static void main(String args[])throws SpecialException{
   try{
      new Sample().methodA(1);
      System.out.println("main");
   }finally{
      System.out.println("Finally");
   }
}
```

（3）在 try 代码块中定义的变量的作用域为 try 代码块，在 catch 代码块和 finally

代码块中不能访问该变量。例如：

```
try{
   int a=1;
   new Sample().methodA(a);
}catch(SpecialException e){
   a=0;  //编译错误
}finally{
   a++;  //编译错误
}
```

如果希望在 catch 代码块和 finally 代码块中访问变量 *a*，必须把变量 *a* 定义在 try 代码块的外面：

```
int a=1;
try{
   new Sample().methodA(a);
}catch(SpecialException e){
   a=0;  //合法
}finally{
   a++;  //合法
}
```

（4）当 try 代码块后面有多个 catch 代码块时，Java 虚拟机会把实际抛出的异常对象依次和各个 catch 代码块声明的异常类型匹配，如果异常对象为某个异常类型或其子类的实例，就执行这个 catch 代码块，不会再执行其他的 catch 代码块。在以下代码中， code1 语句抛出 FileNotFoundException 异常，FileNotFoundException 类是 IOException 类的子类，而 IOException 类是 Exception 的子类。Java 虚拟机先把 FileNotFoundException 对象与 IOException 类匹配，因此当出现 FileNotFoundException 时，程序的打印结果为“IOException”：

```
try{
   code1;  //可能抛出 FileNotFoundException
}catch(SQLException e){
   System.out.println("SQLException");
}catch(IOException e){
   System.out.println("IOException");
}catch(Exception e){
   System.out.println("Exception");
}
```

在以下程序中，如果出现 FileNotFoundException，打印结果为“Exception”，因为 FileNotFoundException 对象与 Exception 类匹配：

```
try{
   code1;  //可能抛出 FileNotFoundException
}catch(SQLException e){
   System.out.println("SQLException");
}catch(Exception e){
   System.out.println("Exception");
}
```

以下程序将导致编译错误，因为如果 code1 语句抛出 FileNotFoundException 异常，

将执行 catch(Exception e)代码块。catch(IOException e)代码块永远不会被执行：

```
try{
   code1;  //可能抛出 FileNotFoundException
}catch(SQLException e){
   System.out.println("SQLException");
}catch(Exception e){
   System.out.println("Exception");
}catch(IOException e){   //编译错误，这个 catch 代码块永远不会被执行
   System.out.println("IOException");
}
```

（5）为了简化编程，从 JDK7 开始，允许在一个 catch 子句中同时捕获多个不同类型的异常，用符号“|”来分割，例如：

```
void method(){
   try{
      //do something...
   }catch(FileNotFoundException | InterruptedIOException ex2){
      //deal with Exception ....
   }
}
```

（6）如果一个方法可能出现受检查异常，要么用 try-catch 语句捕获，要么用 throws 子句声明将它抛出，否则会导致编译错误。关于受检查异常的概念参见本章第 9.3.2 节（受检查异常）。以下 method1()方法声明抛出 IOException，它是受检查异常，其他方法调用 method1()方法：

```
void method1() throws IOException{}   //合法

//编译错误，必须捕获或声明抛出 IOException
void method2(){
   method1();
}

//合法，声明抛出 IOException
void method3()throws IOException {
   method1();
}

//合法，声明抛出 Exception，IOException 是 Exception 的子类
void method4()throws Exception {
   method1();
}

//合法，捕获 IOException
void method5(){
   try{
      method1();
   }catch(IOException e){…}
}

//编译错误，必须捕获或声明抛出 Exception
void method6(){
```

```
    try{
      method1();
    }catch(IOException e){throw new Exception();}
  }

  //合法，声明抛出 Exception
  void method7()throws Exception{
    try{
      method1();
    }catch(IOException e){throw new Exception();}
  }
```

判断一个方法可能会出现异常的依据如下：

- 方法中有 throw 语句。例如以上 method7()方法的 catch 代码块有 throw 语句。
- 调用了其他方法，其他方法用 throws 子句声明抛出某种异常。例如 method3()方法调用了 method1()方法，method1()方法声明抛出 IOException，因此在 method3()方法中可能会出现 IOException。

（7）针对前面一条语法规则，从 JDK7 开始，如果在 catch 子句中捕获的异常被声明为 final 类型，那么当 catch 代码块中继续抛出该异常时，可以不用在定义方法时用 throws 子句声明将它抛出。例如以下是合法的方法定义：

```
void method(){                          //无须声明 cath 代码块中抛出的用 final 修饰的异常
  try{
    //do something...
  }catch(final Throwable ex){           //被捕获的异常用 final 修饰
    //deal with Exception ....
    throw ex; //继续抛出异常
  }
}
```

（8）throw 语句后面不允许紧跟其他语句，因为这些语句永远不会被执行，例如：

```
public void method(int money) throws Exception{
  try{
    if(--money<=0){
      throw new SpecialException("Out of money");
      money=1; //编译错误
    }
  }catch(Exception e){
    throw e;
    money=0; //编译错误
  }finally{
    System.out.println("Finally");
  }
}
```

9.2.6 异常流程的运行过程

异常流程由 try-catch-finally 语句来控制。如果程序中还包含 return 和 System.exit()语句，会使流程变得更加复杂。本节结合具体例子来说明异常流程的运行过程。

（1）finally 语句不被执行的唯一情况是先执行了用于终止程序的 System.exit()方

法。java.lang.System 类的静态方法 exit()用于终止当前的 Java 虚拟机进程，Java 虚拟机所执行的 Java 程序也随之终止。exit()方法的定义如下：

```
public static void exit(int status)
```

exit()方法的参数 status 表示程序终止时的状态码，按照编程惯例，零表示正常终止，非零数字表示异常终止。

Tips

如果在执行 try 代码块时，突然拔掉计算机的电源开关，所有进程都终止运行，当然也不会执行 finally 语句。

以下 try 代码块调用了 System 类的 exit()方法，因此 finally 代码块及 try-finally 语句后面的代码都不会被执行：

```
public static void main(String args[]){
   try{
      System.out.println("Begin");
      System.exit(0);
   }finally{
      System.out.println("Finally");
   }
   System.out.println("End");
}
```

以上程序的打印结果为：Begin。以下程序代码在执行“methodA(1)”语句时出现异常，流程跳转到 catch 代码块。catch 代码块打印“Wrong”后继续抛出异常，main()方法将会异常终止，但在终止之前仍然会执行 finally 代码块：

```
public static void main(String args[])throws Exception{
   try{
      System.out.println("Begin");
      new Sample().methodA(1);   //出现异常
      System.exit(0);
   }catch(Exception e){
      System.out.println("Wrong");
      throw e;          //如果把此行注释掉，将得到不同的运行结果
   }finally{
      System.out.println("Finally");
   }
   System.out.println("End");
}
```

以上程序的粗体字部分表示运行时会执行的代码，程序的打印结果为：

```
Begin
Wrong
Finally
java.lang.SpecialException
…
```

如果把 catch 代码块中的“throw e”语句注释掉，那么在执行完 finally 代码块后还会执行 try-catch-finally 语句后面的代码，程序的打印结果为：

```
Begin
Wrong
Finally
End
```

（2）return 语句用于退出本方法。在执行 try 或 catch 代码块中的 return 语句时，假如有 finally 代码块，会先执行 finally 代码块。例如例程 9-11（WithReturn.java）的 methodB()方法中，在 try 和 catch 代码块中都有 return 语句，其中粗体字部分表示运行时会执行的代码：

例程 9-11　WithReturn.java

```
public class WithReturn{
   public int methodA(int money)throws SpecialException{
      if(--money<=0) throw new SpecialException("Out of money");
      return money;
   }

   public int methodB(int money){
      try{
         System.out.println("Begin");
         int result=methodA(money);   //可能抛出异常
         return result;
      }catch(SpecialException e){
         System.out.println(e.getMessage());
         return -100;
      }finally{
         System.out.println("Finally");
      }
   }

   public static void main(String args[]){
      System.out.println(new WithReturn().methodB(1));
   }
}
```

以上程序的打印结果为：

```
Begin
Out of money
Finally
-100
```

（3）finally 代码块虽然在 return 语句之前被执行，但 finally 代码块不能通过重新给变量赋值的方式来改变 return 语句的返回值。例如：

```
public static int test(){
   int a=0;
   try{
      return a;
   }finally{
      a=1;
   }
}
```

```
    public static void main(String args[])throws Exception{
      System.out.println(test());
    }
```

以上程序在 finally 代码块中把变量 *a* 的值改为 1，但是 test()方法的返回值为 0，因此程序的打印结果为 0。

（4）建议不要在 finally 代码块中使用 return 语句，因为它会导致以下两种潜在的错误。第一种错误是覆盖 try 或 catch 代码块的 return 语句，例程 9-12（FinallyReturn.java）就会导致这种错误。

例程 9-12　FinallyReturn.java

```
public class FinallyReturn {
  public int methodA(int money)throws SpecialException{
    if(--money<=0) throw new SpecialException("Out of money");
    return money;
  }

  public int methodB(int money){
    try{
      return methodA(money);   //可能抛出异常
    }catch(SpecialException e){
      return -100;
    }finally{
      return 100;   //会覆盖 try 和 catch 代码块的 return 语句
    }
  }

  public static void main(String args[]){
    FinallyReturn s=new FinallyReturn ();
    System.out.println(s.methodB(1));   //打印 100
    System.out.println(s.methodB(2));   //打印 100
  }
}
```

s.methodB(1)和 s.methodB(2)的返回值都是 100，由此可见，finally 代码块的 return 语句把 try 和 catch 代码块的 return 语句覆盖了。

第二种错误是丢失异常，例程 9-13（ExLoss.java）就会导致这种错误。

例程 9-13　ExLoss.java

```
public class ExLoss{
  public int methodA(int money)throws SpecialException{
       if(--money<=0) throw new SpecialException("Out of money");
       return money;
  }

  public int methodB(int money)throws Exception{
    try{
      return methodA(money);   //可能抛出异常
    }catch(SpecialException e){
      throw new Exception("Wrong");
    }finally{
```

```
            return 100;
        }
    }

    public static void main(String args[]){
        try{
            System.out.println(new ExLoss().methodB(1));   //打印 100
            System.out.println("No Exception");
        }catch(Exception e){
            System.out.println(e.getMessage());
        }
    }
}
```

methodB()方法的 catch 代码块继续抛出异常，按理说 main()方法的 catch 代码块应该捕获并处理该异常，但由于 methodB()方法的 finally 代码块有返回值，异常被丢失了，main()方法没有捕获到 methodB()方法的异常。以上程序的打印结果为：

```
100
No Exception
```

9.2.7 跟踪丢失的异常

9.2.6 节的例程 9-13 的 ExLoss 类演示了异常丢失的情况。此外，在 try-catch-finally 语句中，如果 try-catch 语句和 finally 语句都抛出异常，那么 try-catch 语句中抛出的异常就会丢失。例如，在下面的例程 9-14 的 ExTester1 类的 show()方法中，catch 代码块和 finally 代码块都抛出异常，那么 catch 代码块抛出的异常会丢失。

例程 9-14　ExTester1.java

```
public class ExTester1 {

    public void show() throws Exception {
        try{
            Integer.parseInt("Hello");
        }catch (NumberFormatException e1) {
            throw new Exception("无效的数字",e1);   //chatc
        } finally {
            try{
                int result = 2 / 0;
            }catch (ArithmeticException e2) {
                throw new Exception("数学运算出错",e2);
            }
        }
    }

    public static void main(String[] args) throws Exception {
        ExTester1 t = new ExTester1();
        t.show();
    }
}
```

为了解决这一问题，在 JDK7 中，Throwable 接口中增加了两个已经提供默认实现的方法：

```
public final void addSuppressed(Throwable exception)
public final Throwable[] getSuppressed()
```

以上 addSuppressed()方法把差点丢失的异常保存了下来，getSuppressed()方法能返回所有保存下来的差点丢失的异常。在以下例程 9-15 的 ExTester2 类的 show()方法的 finally 代码块中，调用异常对象的 addSuppressed()方法，把 catch 代码块中抛出的差点丢失的异常保存下来。在 main()方法中，调用当前异常对象的 getSuppressed()方法，就能得到所有差点丢失的异常。

例程 9-15　ExTester2.java

```
public class ExTester2 {
   public void show() throws Exception {
      Exception myException=null;  //引用当前异常，并且能保存差点丢失的异常
      try{
         Integer.parseInt("Hello");
      }catch (NumberFormatException e1) {
         myException=e1;
      }finally {
         try{
            int result = 2 / 0;
         }catch (ArithmeticException e2) {
            if(myException==null)
              myException =e2;
            else
               myException.addSuppressed(e2);   //保存差点被丢失的异常
         }
         throw   myException;
      }
   }

   public static void main(String[] args){
      ExTester2 t = new ExTester2();
      try{
         t.show();
      }catch (Exception ex) {
         System.out.println("当前异常信息："+ex.getMessage());

         Throwable[] exs=ex.getSuppressed(); //  获得差点丢失的异常
         for(Throwable e:exs)
           System.out.println("差点丢失的异常信息："+e.getMessage());
      }
   }
}
```

以上程序的运行结果如下：

```
当前异常信息：For input string: "Hello"
差点丢失的异常信息：/ by zero
```

9.3 Java 异常类

在程序运行中，任何中断正常流程的因素都被认为是异常。按照面向对象的思想，Java 语言用 Java 类来描述异常。所有异常类的祖先类为 java.lang.Throwable 类，它的实例表示具体的异常对象，可以通过 throw 语句抛出。Throwable 类提供了访问异常信息的一些方法，常用的方法包括：

- getMessage()：返回 String 类型的异常信息。
- printStackTrace()：打印跟踪方法调用栈而获得的详细异常信息。在程序调试阶段，此方法可用于跟踪错误。

例程 9-16（ExTrace.java）演示了 getMessage()和 printStackTrace()方法的用法。

例程 9-16　ExTrace.java

```
public class ExTrace{
   public void methodA(int money)throws SpecialException{
      if(--money<=0) throw new SpecialException("Out of money");
   }
   public void methodB(int money)throws Exception{
      try{
         methodA(1);
      }catch(SpecialException e){
         System.out.println("---Output of methodB()---");
         System.out.println(e.getMessage());
         throw new Exception("Wrong");
      }
   }
   public static void main(String args[]){
      try{
         new ExTrace().methodB(1);
      }catch(Exception e){
         System.out.println("---Output of main()---");
         e.printStackTrace();
      }
   }
}
```

以上程序的打印结果为：

```
---Output of methodB()---
Out of money
---Output of main()---
java.lang.Exception: Wrong
        at ExTrace.methodB(ExTrace.java:11)
        at ExTrace.main(ExTrace.java:16)
```

Throwable 类有两个直接子类：

（1）Error 类：表示单靠程序本身无法恢复的严重错误，比如内存空间不足，或者 Java 虚拟机的方法调用栈溢出。在大多数情况下，遇到这样的错误时，建议让程序

终止。

（2）Exception 类：表示程序本身可以处理的异常，本章所有例子都针对这类异常。当程序运行时出现这类异常，应该尽可能地处理异常，并且使程序恢复运行，而不应该随意终止程序。

JDK 中预定义了一些具体的异常，如图 9-2 所示为常见异常类的类框图。

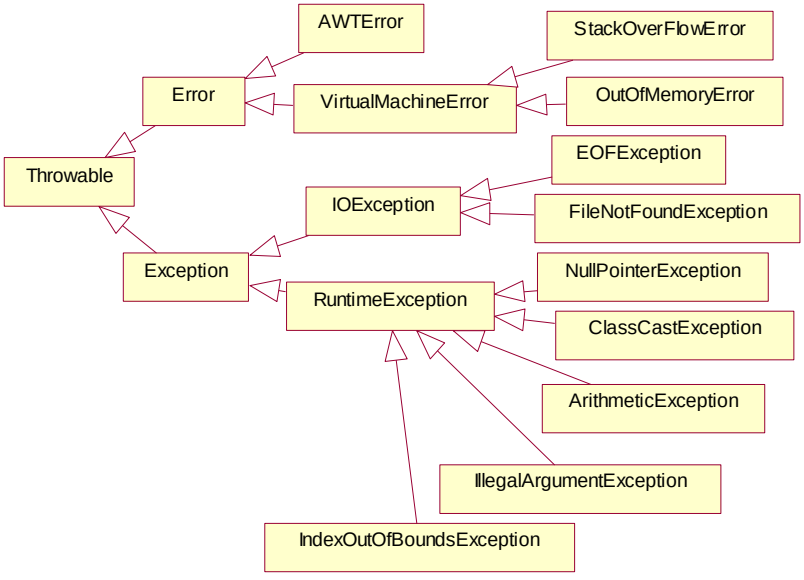


图 9-2　主要 Java 异常类的类框图

下面对一些常见的异常做简要的介绍。

（1）IOException：操作输入流和输出流时可能出现的异常。

（2）ArithmeticException：数学异常。如果把整数除以 0，就会出现这种异常，例如：

```
int a=12 / 0;   //抛出 ArithmeticException
```

（3）NullPointerException：空指针异常。当引用变量为 null 时，试图访问对象的属性或方法，就会出现这种异常，例如：

```
Date d= null;
System.out.println(d.toString());   //抛出 NullPointerException
```

（4）IndexOutOfBoundsException：下标越界异常，它的子类 ArrayIndexOutOfBoundsException 表示数组下标越界异常，以下代码会导致这种异常：

```
int[] array=new int[4];
array[0]=1;
array[7]=1;   //抛出 ArrayIndexOutOfBoundsException
```

（5）ClassCastException：类型转换异常，参见第 6 章的 6.6 节（多态）。

（6）IllegalArgumentException：非法参数异常，可用来检查方法的参数是否合法，例如：

```
public void setName(String name){
   if(name==null)throw new IllegalArgumentException("姓名不能为空");
   this.name=name;
}
```

Exception 类还可分为两种：运行时异常和受检查异常，下面分别介绍。

9.3.1 运行时异常

RuntimeException 类及其子类都称为运行时异常，这种异常的特点是 Java 编译器不会检查它，也就是说，当程序中可能出现这类异常时，即使没有用 try-catch 语句捕获它，也没有用 throws 子句声明抛出它，也会编译通过。例如，当以下 divide()方法的参数 b 为 0，执行“a/b”操作时会出现 ArrithmeticException 异常，它属于运行时异常，Java 编译器不会检查它：

```
public int divide(int a,int b){
   return a/b;   //当参数 b 为 0，抛出 ArrithmeticException
}
```

再例如，下面例程 9-17（WithRuntimeEx.java）中的 IllegalArgumentException 也是运行时异常，divide()方法既没有捕获它，也没有声明抛出它。

例程 9-17　WithRuntimeEx.java

```
public class WithRuntimeEx {
   public int divide(int a,int b){
      if(b==0)throw new IllegalArgumentException("除数不能为 0");
      return a/b;
   }
   public static void main(String args[]){
      new WithRuntimeEx ().divide(1,0);
      System.out.println("End");
   }
}
```

由于程序代码不会处理运行时异常，因此当程序在运行时出现了这种异常时，就会导致程序异常终止，以上程序的打印结果为：

```
Exception in thread "main" java.lang.IllegalArgumentException: 除数不能为 0
        at WithRuntimeEx.divide(WithRuntimeEx.java:3)
        at WithRuntimeEx.main(WithRuntimeEx.java:7)
```

9.3.2 受检查异常（Checked Exception）

除了 RuntimeException 及其子类以外，其他的 Exception 类及其子类都属于受检查异常。这种异常的特点是 Java 编译器会检查它，也就是说，当程序中可能出现这类异常时，要么用 try-catch 语句捕获它，要么用 throws 子句声明抛出它，否则编译不会通过，参见本章 9.2.5 节（异常处理语句的语法规则）。

9.3.3　区分运行时异常和受检查异常

受检查异常表示程序可以处理的异常，如果抛出异常的方法本身不能处理它，那么方法调用者应该去处理它，从而使程序恢复运行，不至于终止程序。例如喷墨打印机在打印文件时，如果纸用完或者墨水用完，就会暂停打印，等待用户添加打印纸或更换墨盒，如果用户添加了打印纸或更换了墨盒，就能继续打印。可以用 OutOfPaperException 类和 OutOfInkException 类来表示纸张用完和墨水用完这两种异常情况，由于这些异常是可修复的，因此是受检查异常，可以把它们定义为 Exception 类的子类：

```
public class OutOfPaperException extends Exception{…}
public class OutOfInkException extends Exception{…}
```

以下是打印机的 print()方法：

```
public void print(){
   while(文件未打印完){
      try{
         打印一行
      }catch(OutOfInkException e){
         do{
            等待用户更换墨盒
         }while(用户没有更换墨盒)
      }catch(OutOfPaperException e){
         do{
            等待用户添加打印纸
         }while(用户没有添加打印纸)
      }
   }
}
```

运行时异常表示无法让程序恢复运行的异常，导致这种异常的原因通常是执行了错误操作。一旦出现了错误操作，建议终止程序，因此 Java 编译器不检查这种异常。

如果程序代码中有错误，就可能导致运行时异常，例如以下 for 语句的循环条件不正确，会导致 ArrayIndexOutOfBoundsException：

```
public void method(int[] array){
   for(int i=0;i<=array.length;i++)
      array[i]=1;   //当 i 的取值为 array.length 时，将抛出 ArrayIndexOutOfBoundsException
}
```

只要对程序代码做适当修改，就能避免数组下标越界异常：

```
public void method(int[] array){
   for(int i=0;i<array.length;i++)
      array[i]=1;   //不会抛出 ArrayIndexOutOfBoundsException
}
```

再例如，对于以下代码，如果 *person* 变量为 null，访问“person.name”会导致 NullPointerException 异常：

```
if(person!=null & person.name.equals("Linda")){…}
```

只要对程序代码做适当修改，就能避免 NullPointerException 异常：

```
if(person!=null && person.name.equals("Linda")){…}
```

Tips

本书第 21 章的 21.10 节（用 Optional 类避免空指针异常）还介绍了用 Optional 类来避免 NullPointerException 异常的方法。

当系统 A 访问了系统 B 的接口时，如果运行时出现了运行时异常，有一种可能是系统 A 访问系统 B 的接口的方式错误。例如，在人（相当于系统 A）用微波炉（相当于系统 B）加热食物的问题领域中，人会调用微波炉的 heaten()方法：

```
public void heaten(Food food){
    //如果是罐装食物，会导致爆炸
    if(food instanceof CannedFood)
        throw new ExplosionException();   //ExplosionException 类是 RuntimeException 类的子类
    …
}
```

以下代码演示某人用微波炉来加热一盒罐装水果，结果引起爆炸：

```
Food fruit=new CannedFood();       //一盒罐装水果
microwaveOven.heaten(fruit);       //出现 ExplosionException
```

之所以出现以上异常，是因为传给 heaten()方法的参数是违规的。谁也不希望微波炉屡屡发生爆炸，正确的做法是避免把罐装食物放到微波炉里加热，这相当于在系统 A 的程序代码中，不要把 CannedFood 类型的参数传给 heaten()方法。

由此可见，运行时异常是应该尽量避免的，在程序调试阶段，遇到这种异常时，正确的做法是改进程序的设计和实现方式，修改程序中的错误，从而避免这种异常。捕获它并且使程序恢复运行并不是明智的办法，这主要有两方面的原因：

（1）一旦发生这种异常，损失严重，比如微波炉发生爆炸。

（2）即使程序恢复运行，也可能会导致程序的业务逻辑错乱，从而导致更严重的异常，或者得到错误的运行结果。

4．区分运行时异常和错误

Error 类及其子类表示程序本身无法修复的错误，它和运行时异常的相同之处是：Java 编译器都不会检查它们，当程序运行时出现它们，都会终止程序。

两者的不同之处在于：Error 类及其子类表示的错误通常是由 Java 虚拟机抛出的，在 JDK 中预定义了一些错误类，比如 OutOfMemoryError 和 StackOutofMemoryError。在应用程序中，一般不会扩展 Error 类，来创建用户自定义的错误类。而 RuntimeException 类表示程序代码中的错误，它是可以扩展的，用户可以根据特定的问题领域，来创建相关的运行时异常类。

9.4　用户定义异常

在特定的问题领域，可以通过扩展 Exception 类或 RuntimeException 类来创建自定义的异常，异常类包含和异常相关的信息，这有助于负责捕获异常的 catch 代码块正确地分析并处理异常。以下代码定义了一个服务器超时异常：

```
public class ServerTimedOutException extends Exception {
   private String reason; //异常原因
   private int port;   //服务器端口
   public ServerTimedOutException (String reason,int port){
      this.reason = reason;
      this.port = port;
   }
   public String getReason() {
      return reason;
   }
   public int getPort() {
      return port;
   }
}
```

以下代码使用 throw 语句来抛出上述异常：

```
//不能连接 80 端口
throw new ServerTimedOutException("Could not connect", 80);
```

9.4.1　异常转译和异常链

在分层的软件结构中，会存在自上而下的依赖关系，也就是说上层的子系统会访问下层子系统的 API。如图 9-3 显示了一个典型的分层结构。

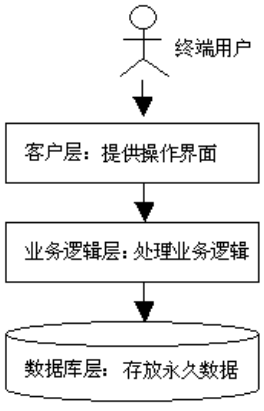


图 9-3　分层的软件系统

在图 9-3 中，客户层访问业务逻辑层，而业务逻辑层访问数据库层。数据库层把异常抛给业务逻辑层，业务逻辑层把异常抛给客户层，客户层则把异常抛给终端用户。

当位于最上层的子系统不需要关心来自底层的异常的细节时，常见的做法是捕获原始的异常，把它转换为一个新的不同类型的异常，再抛出新的异常，这种处理异常的办法称为异常转译。例如，假设终端用户通过客户界面把一个图像文件上传到数据库中，客户层调用业务逻辑层的 uploadImageFile()方法：

```
public void uploadImageFile( String imagePath ) throws UploadException{
  try{
    //上传图像文件
    …
  }catch(IOException e){
    //把原始异常信息记录到日志中，便于排错
    …
    throw new UploadException();
  }catch(SQLException){
    //把原始异常信息记录到日志中，便于排错
    …
    throw new UploadException();
  }
}
```

uploadImageFile()方法执行上传图像文件操作时，可能会捕获 IOException 或者 SQLException。但是用户没有必要关心异常的底层细节，他们仅需要知道上传图像失败，具体的调试和排错由系统管理员或者软件开发人员来处理。因此，uploadImageFile()方法捕获到原始的异常后，在 catch 代码块中先把原始的异常信息记入日志，然后向用户抛出 UploadException 异常。

从面向对象的角度来理解，异常转译使得异常类型与抛出异常的对象的类型位于相同的抽象层。例如，车子运行时能出现故障异常，而职工开车上班会出现迟到异常，车子的故障异常是导致职工迟到异常的原因。以下职工的 gotoWork()方法直接抛出车子故障异常：

```
public void gotoWork()throws CarWrongExceptio n{
  try{
    car.run();
  }catch(CarWrongException e){   //处理车子出故障的异常
    walk();
    throw e;
  }
}
```

以上代码意味着车子故障是在职工身上发生的，这显然是不合理的，正确的做法是把 CarWrongException 转译为 LateException，参见本章 9.1.1 节的例程 9-6（Worker.java）。

JDK1.4 以上版本中的 Throwable 类支持异常链机制。所谓异常链就是把原始异常包装为新的异常类，也就是说在新的异常类中封装了原始异常类，这有助于查找产生异常的根本原因。此外，如果使用 JDK1.4 以下的版本，可以由开发者自行设计支持异常链的异常类，例程 9-18（BaseException.java）提供了一种实现方案，JDK1.4 以上版本中的 Throwable 类的实现机制与它很相似。

例程 9-18　支持异常链的异常类 BaseException.java

```
import java.io.*;
public class BaseException extends Exception {
   protected Throwable cause = null;

   public BaseException(){}

   public BaseException(String msg){super(msg);}

   public BaseException( Throwable cause ) {                    //参数 cause 指定原始的异常
      this.cause =cause;
   }
   public BaseException(String msg,Throwable cause){            //参数 cause 指定原始的异常
      super(msg);
      this.cause = cause;
   }

   public Throwable initCause(Throwable cause) {
      this.cause =cause;
      return this;
   }

   public Throwable getCause() {
      return cause;
   }

   public void printStackTrace() {
      printStackTrace(System.err);
   }

   public void printStackTrace(PrintStream outStream) {
      printStackTrace(new PrintWriter(outStream));
   }

   public void printStackTrace(PrintWriter writer) {
      super.printStackTrace(writer);

      if ( getCause() != null ) {
         getCause().printStackTrace(writer);
      }
      writer.flush();
   }
}
```

在 BaseException 中定义了 Throwable 类型的 cause 变量，它用于保存原始的 Java 异常。假定 UploadException 类扩展了 BaseException 类：

```
public class UploadException extends BaseException{
   public UploadException(Throwable cause){super(cause);}
   public UploadException(String msg){super(msg);}
}
```

以下是把 IOException 包装为 UploadException 的代码：

```
try{
```

```
        //上传图像文件
        …
    }catch(IOException e){
        //把原始异常信息记录到日志中，便于排错
        …
        //把原始异常包装为 UploadException
        UploadException ue= new UploadException(e);
        throw ue;
    }
```

9.4.2 处理多样化异常

和异常链相近的一个概念是多样化异常。在实际应用中，有时需要一个方法同时抛出多个异常。例如，用户提交的 HTML 表单上有多个字段域，业务规则要求每个字段域的值都符合特定规则，如果不符合规则，就抛出相应的异常。

如果应用程序不支持在一个方法中同时抛出多个异常，用户每次只能看到针对一个字段域的验证错误。当改正了一个错误后，重新提交表单，又收到针对另一个字段域的验证错误，这会令用户很烦恼。

有效的做法是每次当用户提交表单后，都验证所有的字段域，然后向用户显示所有的验证错误信息。不幸的是，在 Java 方法中一次只能抛出一个异常对象。因此需要开发者自行设计支持多样化异常的异常类。例程 9-19 提供了一种实现方案。

例程 9-19　支持多样化异常的异常类 BaseException.java

```
package multiex;
import java.util.List;
import java.util.ArrayList;
import java.io.PrintStream;
import java.io.PrintWriter;

public class BaseException extends Exception{

   protected Throwable cause = null;
   private List <Throwable> exceptions = new ArrayList<Throwable>();

         public BaseException(){}

   public BaseException(String msg){super(msg);}

   public BaseException( Throwable cause ) {
      this.cause = cause;
   }

   public BaseException(String msg,Throwable cause){
      super(msg);
      this.cause = cause;
   }

   public List getExceptions() {
      return exceptions;
   }
```

```
    public void addException( BaseException ex ){
        exceptions.add( ex );
    }

    public Throwable initCause(Throwable cause) {
        this.cause = cause;
        return this;
    }

    public Throwable getCause() {    //返回原始的异常
        return cause;
    }

    public void printStackTrace() {
        printStackTrace(System.err);
    }

    public void printStackTrace(PrintStream outStream) {
        printStackTrace(new PrintWriter(outStream));
    }

    public void printStackTrace(PrintWriter writer) {
        super.printStackTrace(writer);

        if ( getCause() != null ) {
            getCause().printStackTrace(writer);
        }
        writer.flush();
    }
}
```

BaseException 类包含一个 List 类型的 exceptions 变量，用来存放其他的 Exception。以下代码显示了 BaseException 的用法：

```
public void check() throws BaseException{
    BaseException be=new BaseException();
    try{
        checkField1();
    }catch(Field1Exception e){be.addException(e);}

    try{
        checkField2();
    }catch(Field2Exception e){be.addException(e);}

    if(be.getExceptions().size>0)throw be;
}
```

9.5　异常处理原则

本章从优化程序的角度出发，介绍了正确运用异常处理机制的原则。遵守这些原

则，可以提高程序的健壮性，并且有利于排除程序代码中的错误。

9.5.1 异常只能用于非正常情况

异常只能用于非正常情况，不能用异常来控制程序的正常流程。以下程序代码用抛出异常的手段来结束正常的循环流程：

```
public static void initArray(int[] array){
   try{
      int i=0;
      while(true){
         array[i++]=1;
      }
   }catch(ArrayIndexOutOfBoundsException e){}
}
```

这种处理方式有以下缺点：

（1）滥用异常流程会降低程序的性能。

（2）用异常类来表示正常情况，违背了异常处理机制的初衷。在遍历 array 数组时，当访问到最后一个元素时，应该正常结束循环，而不是抛出异常。

（3）模糊了程序代码的意图，影响可读性。如果把 initArray()方法改为以下实现方式，就会使程序代码一目了然：

```
public static void initArray(int[] array){
   for(int i=0;i<array.length;i++)
      array[i]=1;
}
```

（4）容易掩盖程序代码中的错误，增加调试的复杂性。例如，以下程序的本意是找出数组中的最大值，把它赋给最后一个元素。由于编程人员误以为 Java 数组的下标范围为：1 ~ array.length，因此把数组最后一个元素表示为 array[array.length]。当执行第一次循环时，就会抛出 ArrayIndexOutOfBoundsException 异常，从而结束循环。由于这个异常被捕获，程序在运行时不会异常终止，这使得编程人员难以发现程序代码中的错误：

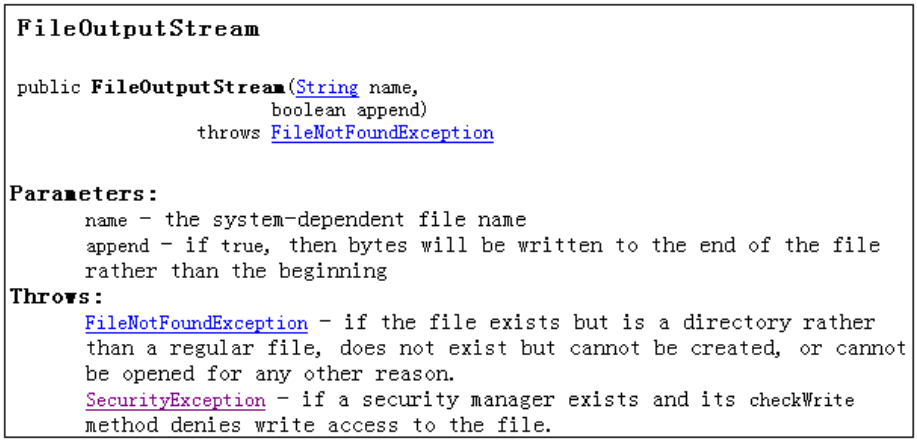
```
public static void changeArray(int[] array){
   try{
      int i=1;
      while(true){
         if(array[i]>array[array.length]){
            array[array.length]=array[i];
         }
         i++;
      }
   }catch(ArrayIndexOutOfBoundsException e){}
}
```

9.5.2 为异常提供说明文档

在 JavaDoc 文档中应该为方法可能抛出的所有异常提供说明文档。无论是受检查

异常，还是运行时异常，都应该通过 JavaDoc 的@throws 标签来描述产生异常的条件。关于@throws 标签的用法参见第 2 章的 2.3.1 节（JavaDoc 标记）。如图 9-4 所示是 java.io.FileOutputStream 类的一个构造方法的部分 JavaDoc 文档。通过这份文档，使用者了解到这个方法可能会抛出 FileNotFoundException 和 SecurityException，前者是受检查异常，因此在构造方法中用 throws 子句声明抛出它，后者是运行时异常，无须用 throws 子句加以声明。

完整的异常文档可以帮助方法调用者正确地调用方法，提供合理的参数，尽可能地避免异常或者能方便地找到产生异常的原因。



FileOutputStream

public **FileOutputStream**(String name,
boolean append)
throws FileNotFoundException

Parameters:
name - the system-dependent file name
append - if true, then bytes will be written to the end of the file rather than the beginning

Throws:
FileNotFoundException - if the file exists but is a directory rather than a regular file, does not exist but cannot be created, or cannot be opened for any other reason.
SecurityException - if a security manager exists and its checkWrite method denies write access to the file.

图 9-4　FileOutputStream 类的部分 JavaDoc 文档

9.5.3　尽可能地避免异常

应该尽可能地避免异常，尤其是运行时异常。避免异常通常有两种办法：

（1）许多运行时异常是由于程序代码中的错误引起的，只要修改了程序代码的错误，或者改进了程序的实现方式，就能避免这种错误。

（2）提供状态测试方法。有些异常是由于当对象处于某种状态下，不适合某种操作造成的。例如，当高压锅内的水蒸气的压力很大时，突然打开锅盖，会导致爆炸。为了避免这类事故，高压锅应该提供状态测试功能，让使用者在打开锅盖前，能够判断锅内的高压蒸汽是否排放完。下面用 ExplosionException 类表示爆炸异常，它是一种后果严重的应该避免的异常，定义为 RuntimeException 类的子类。PressureTank 类表示高压锅，它的 isSafeForOpen()方法为状态测试方法，参见例程 9-20。

例程 9-20　PressureTank.java

```
public class PressureTank {
   private int pressure;   //当前压力
   private static final int SAFE_PRESSURE=2;    //当锅内压力低于两个大气压，可打开锅盖
   private static final int CRITICAL_POINT=3;   //爆炸时的压力临界点为 3 个大气压

   public boolean isSafeForOpen(){
      return pressure<=SAFE_PRESSURE;
```

```
    }

    public void exhaust(){       //排气
      pressure=SAFE_PRESSURE;
    }

    public void open(){          //打开锅盖
      if(pressure>=CRITICAL_POINT)throw new ExplosionException();
    }

    public void cook(){
      pressure=CRITICAL_POINT;
    }
  }
```

以下代码演示使用者用高压锅来安全地烧饭的过程：

```
pressureTank.cook();     //烧饭
pressureTank.exhaust();  //排气
if(pressureTank.isSafeForOpen())  //先进行状态测试，避免爆炸
  pressureTank.open();   //打开锅盖
```

Tips

高压锅有专门的排气阀门，打开阀门，蒸汽就能逐渐释放出去，如果不再有蒸汽排放出去，就表明压力恢复正常，此时可以安全地打开锅盖。

9.5.4 保持异常的原子性

应该尽力保持异常的原子性。异常的原子性是指当异常发生后，各个对象的状态能够恢复到异常发生前的初始状态，而不至于停留在某个不合理的中间状态。对象的状态是否合理，是由特定问题领域的业务逻辑决定的。例如，假设一开始张三和李四的银行账户上都有 1000 元钱，张三把 100 元钱转到李四的账户上。以下程序代码演示了转账过程：

```
//转账操作
public void transfer(Account accountFrom,Account accountTo){
  //从转出账户中取出 100 元
  accountFrom.setBalance(accountFrom.getBalance()-100);
  //向转入账户中存入 100 元
  accountTo.setBalance(accountTo.getBalance()+100);
}
```

以下代码调用 transfer()方法来实现张三和李四之间的转账：

```
transfer(zhangsanAccount,lisiAccount);
```

假如以上操作成功，那么张三和李四的银行账户的余额分别为 900 和 1100 元，如图 9-5 所示。

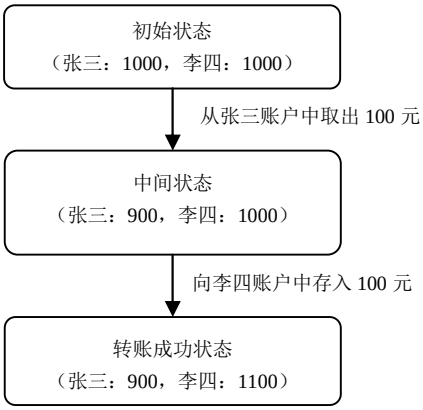


图 9-5　转账操作正常执行时的状态转换图

Account 类代表银行账户，它的 setBalance()方法可能会抛出 IllegalStateException 异常，该异常是运行时异常：

```
public setBalance(int balance){
   if(isClosed()) throw new IllegalStateException("账户已关闭");
   this.balance=balance;
}
```

假设在执行转账操作时，张三的账户为打开状态，而李四的账户为关闭状态，那么转账操作无法正常执行。从张三账户中取出 100 元的操作执行成功，而在执行向李四账户中存入 100 元的操作时，出现 IllegalStateException 异常，该异常使得各个对象的状态永久停留在不合理的中间状态，张三账户上少了 100 元，而李四账户上未增加 100 元，这显然是不合理的。

保持异常的原子性有以下办法：

（1）最常见的办法是先检查方法的参数是否有效，确保当异常发生时还没有改变对象的初始状态。对 transfer()方法可做如下修改：

```
public void transfer(Account accountFrom,Account accountTo){
   if(accountFrom.isClosed())
      throw new IllegalStateException("转出账户已关闭");
   if(accountTo.isClosed())
      throw new IllegalStateException("转入账户已关闭");

   //从转出账户中取出 100 元
   accountFrom.setBalance(accountFrom.getBalance()-100);
   //向转入账户中存入 100 元
   accountTo.setBalance(accountTo.getBalance()+100);
}
```

（2）编写一段恢复代码，由它来解释操作过程中发生的失败，并且使对象状态回滚到初始状态。这种办法不是很常用，主要用于永久性的数据结构，比如数据库系统的事务回滚机制就采取了这种办法。

（3）在对象的临时副本上进行操作，当操作成功后，把临时副本中的内容复制到原来的对象中。

9.5.5 避免过于庞大的 try 代码块

有些编程新手喜欢把大量代码放入单个 try 代码块，这看起来省事，实际上不是好的编程习惯。try 代码块越庞大，出现异常的地方就越多，要分析发生异常的原因就越困难。有效的做法是分割各个可能出现异常的程序段落，把它们分别放在单独的 try 代码块中，从而分别捕获异常。

9.5.6 在 catch 子句中指定具体的异常类型

有些编程新手喜欢用 catch(Exception ex)子句来捕获所有异常，例如，在以下打印机的 print()方法中，用 catch(Exception ex)子句来捕获所有的异常，包括 OutOfInkException 和 OutOfPaperException：

```
public void print(){
  while(文件未打印完){
    try{
      打印一行
    }catch(Exception e){…}
  }
}
```

以上代码看起来省事，但实际上不是好的编程习惯，理由如下：

（1）俗话说“对症下药”，对不同的异常通常有不同的处理方式。以上代码意味着对各种异常采用同样的处理方式，这往往是不现实的。

（2）会捕获本应该抛出的运行时异常，掩盖程序中的错误。

正确的做法是在 catch 子句中指定具体的异常类型：

```
public void print(){
  while(文件未打印完){
    try{
      打印一行
    }catch(OutOfInkException e){
      do{
        等待用户更换墨盒
      }while(用户没有更换墨盒)
    }catch(OutOfPaperException e){
      do{
        等待用户添加打印纸
      }while(用户没有添加打印纸)
    }
  }
}
```

9.5.7 不要在 catch 代码块中忽略被捕获的异常

只要异常发生，就意味着某些地方出了问题，catch 代码块既然捕获了这种异常，就应该提供处理异常的措施，比如：

- 处理异常。针对该异常采取一些行动，比如弥补异常造成的损失或者给出警

告信息等。

- 重新抛出异常。catch 代码块在分析了异常之后，认为自己不能处理它，重新抛出异常。
- 进行异常转译，把原始异常包装为适合于当前抽象的另一种异常，再将其抛出，参见本章 9.4.1 节（异常转译和异常链）。
- 假如在 catch 代码块中不能采取任何措施，那么就不要捕获异常，而是用 throws 子句声明将异常抛出。

以下两种处理异常的方式是应该避免的：

```
try{
   …
}catch(SpecialException e){} //对异常不采取任何操作

或者

try{
   …
}catch(SpecialException e){e.printStackTrace();} //仅仅打印异常信息
```

在 catch 代码块中调用异常类的 printStackTrace()方法，对调试程序有帮助，但程序调试阶段结束之后，printStackTrace()方法就不应再在异常处理代码块中担负主要责任，因为光靠打印异常信息并不能解决实际存在的问题。

Tips

本章有些例子在 catch 代码块中仅仅打印异常信息，这是因为这些例子侧重于演示异常流程的运行过程。

9.6　记录日志

在程序中输出日志总的来说有 3 个作用：

- 监视代码中变量的变化情况，把数据周期性地记录到文件中供其他应用进行统计分析工作。
- 跟踪代码运行时轨迹，作为日后审计的依据。
- 承担集成开发环境中的调试器的作用，向文件或控制台打印代码的调试信息。

要在程序中输出日志，最普通的做法是在代码中嵌入许多打印语句，这些打印语句可以把日志输出到控制台或文件中。比较好的做法是构造一个日志操作类来封装此类操作，而不是让一系列的打印语句充斥代码的主体。

在强调可重用组件开发的今天，可以直接使用 Java 类库的 java.util.logging 日志操作包。这个包中主要有 4 个类：

- Logger 类：负责生成日志，并能够对日志信息进行分级别筛选，通俗地讲，就是决定什么级别的日志信息应该被输出，什么级别的日志信息应该被忽略。

- Handler 类：负责输出日志信息，它有两个子类：ConcoleHandler 类（把日志输出到 DOS 命令行控制台）和 FileHandler 类（把日志输出到文件中）。
- Formatter 类：指定日志信息的输出格式。它有两个子类：SimpleFormatter 类（表示常用的日志格式）和 XMLFormatter 类（表示基于 XML 的日志格式）。
- Level 类：表示日志的各种级别，它的静态常量如 Level.INFO、Level.WARNING 和 Level.CONFIG 等分别表示不同的日志级别。

如图 9-6 显示了这些类之间的关系。

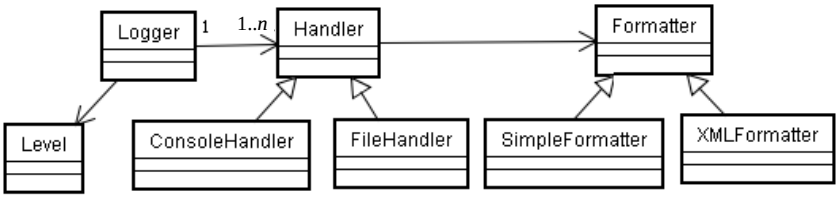


图 9-6　日志操作包中各个类的关系

上述类协同工作，使得开发者能够依据日志信息级别去记录信息，并能够在程序运行期间，控制日志信息的输出格式以及日志存放地点。

9.6.1　创建 Logger 对象及设置日志级别

要记录日志，首先通过 Logger 类的 getLogger(String name)方法获得一个 Logger 对象：

```
Logger myLogger=Logger.getLogger("mylogger");
```

对于以上代码，如果名为“mylogger”的 Logger 对象已经存在，就直接将其引用返回，否则就创建一个名为“mylogger”的 Logger 对象并返回它的引用。

在写程序的时候，为了调试程序，会在很多可能出错的地方输出大量的日志信息。当程序调试完毕后，不需要输出这些日志信息了，那怎么办呢？以前的做法是把每个程序中输出日志信息的代码删除。对于大的应用程序，这种做法既费力又费时，几乎是不现实的。Java 日志操作包采用日志级别机制，简化了控制日志输出的步骤。日志共分为 9 个级别，从低到高依次为：SEVERE、WARNING、INFO、CONFIG、FINE、CONFIG、FINE、FINER 和 FINEST。

在默认情况下，Logger 类只输出 SEVERE、WARNING、INFO 这前 3 个级别。可以通过 Logger 类的 setLevel()方法来设置日志级别，例如：

```
logger.setLevel("Level.FINE");   //把日志级别设为 FINE
logger.setLevel("Level.WARNING"); //把日志级别设为 WARNING
logger.setLevel("Level.ALL"); //开启所有日志级别
logger.setLevel("Level.OFF"); //关闭所有日志级别
```

如果把日志级别设为 WARNING，那么意味着只会输出 WARNING 级别及比它低级别的日志。

9.6.2　生成日志

与日志级别相对应，Logger 类的 severe()、warn()和 info()方法等分别生成各种级别的日志。例程 9-21 的 LoggerTester1 类的 main()方法负责生成并输出一些日志。

例程 9-21　LoggerTester1.java

```
import java.util.logging.*;
class LoggerTester1{
   public    static void main(String[] args) {
      Logger myLogger = Logger.getLogger("mylogger");//得到一个日志记录器对象
      myLogger.setLevel(Level.WARNING);                  //设置 WARNING 日志级别

      myLogger.info("这是一条普通提示信息");             //生成 INFO 级别的日志
      myLogger.warning("这是一条警告信息");              //生成 WARNING 级别的日志
      myLogger.severe("这是一条严重错误信息");           //生成 SEVERE 级别的日志
   }
}
```

以上代码将 myLogger 对象的日志级别设为 WARNING，因此，只有 warning()方法和 server()方法生成的日志会被输出。运行以上程序，将在 DOS 控制台输出以下内容：

```
警告: 这是一条警告信息
严重: 这是一条严重错误信息
```

9.6.3　把日志输出到文件

默认情况下，Logger 类把日志输出到 DOS 控制台，也就是说，Logger 对象与一个 ConcoleHandler 对象关联，例如本章 9.6.2 节的例程 9-21 的 LoggerTester1 类就把日志输出到控制台。如果要把日志输出到文件中，可以采用以下方式：

```
FileHandler fileHandler = new FileHandler("C:\\test.log");
fileHandler.setLevel(Level.INFO); //设定向文件中写日志的级别
myLogger.addHandler(fileHandler); //把 FileHandler 与 Logger 对象关联
```

以上代码创建了一个 FileHandler 对象，它负责向指定的“C:/test.log”文件中写日志。Logger 类的 addHandler()方法会建立 Logger 对象与 FileHandler 对象的关联。例程 9-22 的 LoggerTester2 类是向文件中写日志的范例。

例程 9-22　LoggerTester2.java

```
import java.util.logging.*;
import java.io.IOException;
public class LoggerTester2{
   public    static void main(String[] args) throws IOException {
      Logger myLogger = Logger.getLogger("mylogger"); //得到一个日志记录器对象

      //创建一个 FileHandler 对象，它向指定的文件中写日志
      FileHandler fileHandler = new FileHandler("C:\\ test.log");
      fileHandler.setLevel(Level.INFO);      //设定向文件中写日志的级别
      myLogger.addHandler(fileHandler);    //把 FileHandler 与 Logger 对象关联
```

```
        myLogger.info("这是一条普通提示信息");          //生成 INFO 级别的日志
        myLogger.warning("这是一条警告信息");          //生成 WARNING 级别的日志
        myLogger.severe("这是一条严重错误信息");       //生成 SEVERE 级别的日志
    }
}
```

运行以上程序，将在"C:\test.log"文件中记录以下 XML 格式的日志：

```
<?xml version="1.0" encoding="GBK" standalone="no"?>
<!DOCTYPE log SYSTEM "logger.dtd">
<log>
<record>
  ...
  <message>这是一条普通提示信息</message>
</record>
<record>
  ...
  <message>这是一条警告信息</message>
</record>
<record>
  ...
  <message>这是一条严重错误信息</message>
</record>
</log>
```

9.6.4 设置日志的输出格式

默认情况下，ConcoleHandler 类与 SimpleFormatter 类关联，也就是说，向控制台输出的日志采用普通的格式。默认情况下，FileHandler 类与 XMLFormatter 类关联，也就是说，向文件中输出的日志是基于 XML 格式的。

也可以自定义一个继承 Fomatter 类的子类，然后覆盖它的 format()方法，在该方法中指定客户化的日志输出格式。例如，在例程 9-23 的 LoggerTester3 类中，有一个 MyFormatter 内部类，它就是自定义的日志输出格式类。

例程 9-23　LoggerTester3.java

```
import java.util.logging.*;
import java.io.IOException;
public class LoggerTester3{

  static class MyFormatter extends Formatter {                    //自定义的日志输出格式类
    public String format(LogRecord record) {                      //覆盖 format()方法
      return "<"+record.getLevel() + ">:" + record.getMessage()+"\n";
    }
  }

  public   static void main(String[] args)throws IOException{
    Logger myLogger = Logger.getLogger("mylogger");               //得到一个日志记录器对象

    FileHandler fileHandler = new FileHandler("C:\\test.log");
    fileHandler.setFormatter(new MyFormatter());                  //设置自定义的日志输出格式
```

```
        myLogger.addHandler(fileHandler); //把 FileHandler 与 Logger 对象关联

        myLogger.info("这是一条普通提示信息");     //生成 INFO 级别的日志
        myLogger.warning("这是一条警告信息");     //生成 WARNING 级别的日志
    }
}
```

运行以上程序，在日志文件“C:\test.log”中将输出以下格式的日志：

```
<INFO>:这是一条普通提示信息
<WARNING>:这是一条警告信息
```

9.7　使用断言

在编写代码时，有时会做出一些条件已经成立的假设，例如，以下方法假设其调用者传入的参数 b 不为零：

```
public int divide(int a,int b){
    return a/b;
}
```

但实际上，有可能编写调用该方法的程序员由于粗心，忘了对参数 b 进行检查，结果把值为 0 的参数 b 传给 divide()方法，divide()方法在执行时就会抛出 ArithmeticException 异常。为了提高程序代码的健壮性，使得程序员在开发调试阶段就能及时发现这样的漏洞，从 JDK1.4 开始，引入了断言机制。

可以把断言看作是异常处理的一种高级形式。使用断言的语法有两种形式为：

```
assert 条件表达式
assert 条件表达式 : 包含错误信息的表达式
```

以上 assert 为 Java 关键字。条件表达式的值是一个布尔值。当条件表达式的值为 false 时，就会抛出一个 AssertionError，这是一个错误，而不是一个异常。在第二种形式中，第二个表达式会被转换成作为错误消息的字符串。例如：

```
assert b!=0 : b;
```

例程 9-24 的 AssertTester 类的 divide()方法就使用了断言。

例程 9-24　AssertTester.java

```
public class AssertTester {
    public int divide(int a,int b){
        assert b!=0 : “除数不允许为零”;  //使用断言
            return a/b;
    }

    public static void main(String[] args) {
            AssertTester t=new AssertTester();
            System.out.println(t.divide(3,0));
    }
}
```

当程序运行时，断言在默认情况下是关闭的，这意味着程序中的断言语句不会被执行。在“java”命令中启用断言需要使用-ea 参数，禁用断言使用-da 参数。例如，用命令“java –ea AssertTester”来执行 AssertTester 类，将会输出以下错误信息：

```
Exception in thread “main” java.lang.AssertionError: 除数不允许为零
at AssertTester.divide(AssertTester.java:4)
at AssertTester.main(AssertTester.java:10)
```

以下命令表明在运行 AppMain 类时，启用 MyClass1 类的断言，并且禁用 MyClass2 类的断言：

```
java  –ea : MyClass1  –da : MyClass2  AppMain
```

9.8 小结

Java 的异常处理涉及 5 个关键字 try、catch、throw、throws 和 finally。异常处理流程由 try、catch 和 finally 等 3 个代码块组成。其中 try 代码块包含可能发生异常的程序代码；catch 代码块紧跟在 try 代码块后面，用来捕获并处理异常；finally 代码块用于释放被占用的相关资源。

Exception 类表示程序中出现的异常，可分为受检查异常和运行时异常。受检查异常表示只要通过处理，就可能使程序恢复运行的异常。对于方法中可能出现的受检查异常，要么用 try-catch 语句捕获并处理它，要么用 throws 语句声明抛出它，Java 编译器会对此做检查。运行时异常表示会导致程序终止的异常，Java 编译器不会对此做检查。运行时异常通常是由程序代码中的错误造成的，因此要尽量避免它。

本章最后还总结了一些异常处理的原则，包括：

- 异常只能用于非正常情况。
- 为异常提供说明文档。
- 尽可能地避免异常，尤其是运行时异常。
- 保持异常的原子性。
- 避免过于庞大的 try 代码块。
- 在 catch 子句中指定具体的异常类型。
- 不要在 catch 代码块中忽略被捕获的异常。

Java 日志操作包用于记录程序运行中产生的日志，有助于调试和检测程序的运行。Logger 类是日志记录器，负责生成各种级别的日志，Handler 类负责向控制台或文件输出日志，Formatter 类指定日志的输出格式，Level 类有一系列的静态常量，分别表示各种日志级别。

Java 断言主要在程序调试阶段使用，有利于及时发现程序中的一些缺陷，当程序运行时出现 AssertionError 错误时，程序员可以根据该错误提示来修改程序代码，确保断言中指定的假设条件真正成立。

9.9　思考题

1．throw 和 throws 关键字有什么区别？

2．以下代码能否编译通过？假如能编译通过，运行时将得到什么打印结果？

```
import java.io.*;
class Base{
   public static void amethod()throws FileNotFoundException{}
}

public class ExcepDemo extends Base{
   public static void main(String argv[]){
      ExcepDemo e = new ExcepDemo();
   }
   public static void amethod(){}

   protected ExcepDemo() throws IOException{
      DataInputStream din = new DataInputStream(System.in);
      System.out.println("Pausing");
      din.readChar();
      System.out.println("Continuing");
      this.amethod();
   }
}
```

3．try 代码块后面可以只跟 finally 代码块，这句话对吗？

4．对于以下程序，运行“java Rethrow”，将得到什么打印结果？

```
public class Rethrow {
   public static void g() throws Exception {
      System.out.println("Originates from g()");
      throw new Exception("thrown from g()");
   }
   public static void main(String []args) {
      try {
         g();
    }catch(Exception e) {
         System.out.println("Caught in main");
         e.printStackTrace();
         throw new NullPointerException("from main");
      }
   }
}
```

5．假定方法 f()可能会抛出 Exception，以下哪些代码是合法的？

a)

```
public static void g(){
   try {
      f();
   }catch(Exception e) {
```

```
        System.out.println("Caught in g()");
        throw new Exception("thrown from g()");
    }
}
```

b)

```
public static void g(){
    try {
      f();
    }catch(Exception e) {
        System.out.println("Caught in g()");
        throw new NullPointerException("thrown from g()");
    }
}
```

c)

```
public static void g() throws Throwable{
    try {
        f();
    }catch(Exception e) {
        System.out.println("Inside g()");
        throw e.fillInStackTrace();
    }
}
public static void main(String []args) {
    try {
        g();
    }catch(Exception e) {
        System.out.println("Caught in main");
        e.printStackTrace();
    }
}
```

6．在 Java 中，Throwable 是所有异常类的祖先，这句话对吗？

7．在执行 trythis()方法时，如果 problem()方法抛出 Exception，程序将打印什么结果？

```
public void trythis() {
    try{
        System.out.println("1");
        problem();
    }catch (RuntimeException x) {
        System.out.println("2");
        return;
    }catch (Exception x) {
        System.out.println("3");
        return;
    }finally {
        System.out.println("4");
    }
    System.out.println("5");
}
```

8．以下代码能否编译通过？假如能编译通过，运行“java　MyTest”时将得到什么打印结果？

```
class MyException extends Exception {}
```

```
public class MyTest {
   public void foo() {
      try{
         bar();
      }finally {
         baz();
      }catch (MyException e) {}
   }
   public void bar() throws MyException {
      throw new MyException();
   }
   public void baz() throws RuntimeException {
      throw new RuntimeException();
   }
}
```

9．对于以下代码：

```
import java.io.*;
public class Th{
   public static void main(String argv[]){
      Th t = new Th();
      t.amethod();
   }
   public void amethod(){
      try{
         ioCall();
      }catch(IOException ioe){}
   }
}
```

以下哪些是合理的 ioCall()方法的定义？

a)

```
public void ioCall () throws IOException{
   DataInputStream din = new DataInputStream(System.in);
   din.readChar();
}
```

b)

```
public void ioCall () throw IOException{
   DataInputStream din = new DataInputStream(System.in);
   din.readChar();
}
```

c)

```
public void ioCall (){
   DataInputStream din = new DataInputStream(System.in);
   din.readChar();
}
```

d)

```
public void ioCall throws IOException(){
   DataInputStream din = new DataInputStream(System.in);
   din.readChar();
}
```

读书笔记