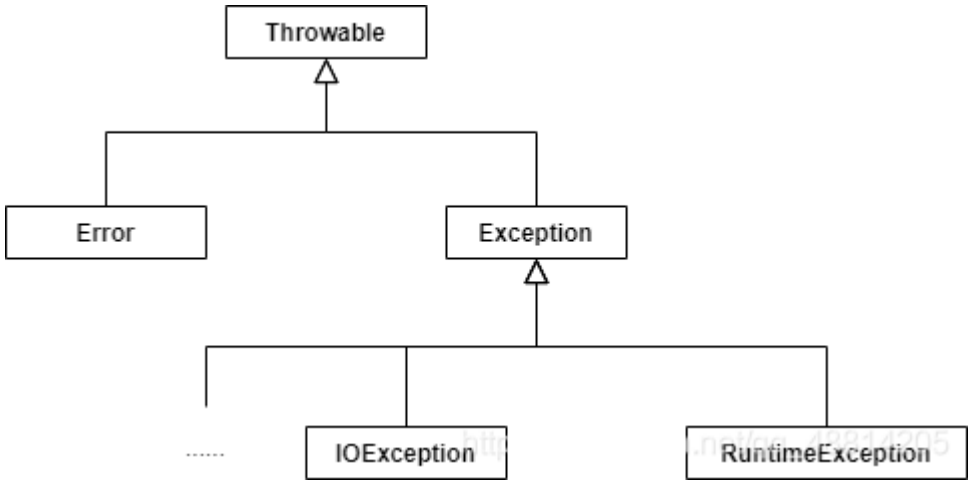
1 处理错误

1.1 异常分类

在 Java 中，异常对象都是派生于`Throwable`类的一个类实例。下图是 Java 异常层次结构的简化示意图：



`Error`类层次结构描述了 Java 运行时系统的内部错误和资源耗尽错误。如果出现了这样的内部错误，除了通知用户并尽力妥善地终止程序之外，几乎无能为力。这种情况很少出现。

在设计 Java 程序时，要重点关注`Exception`层次结构。这个层次结构又分解为两个分支：一个分支派生于`RuntimeException`，另一个分支包含其他异常。一般规则是：由编程错误导致的异常属于`RuntimeException`；如果程序本身没有问题，但由于像 I/O 错误这类问题导致的异常属于其他异常。

派生于`Error`类或`RuntimeException`类的所有异常称为非检查型异常，所有其他的异常称为检查型异常。编译器将检查是否为所有的检查型异常提供了异常处理器。

1.2 声明检查型异常

如果遇到了无法处理的情况，Java 方法可以抛出一个异常。要在方法的首部指出这个方法可能抛出的检查型异常。例如：

```
public FileInputStream(String name) throws FileNotFoundException
```

这是一个构造器，根据给定的`String`参数构造一个`FileInputStream`对象。如果出错，构造器将不会初始化一个新的`FileInputStream`对象，而是抛出一个`FileNotFoundException`类对象。

如果一个方法有可能抛出多个检查型异常类型，就必须在方法的首部列出所有的异常类，每个异常类之间用逗号隔开。例如：

```
public Image loadImage(String s) throws FileNotFoundException, EOFException
```

不需要声明从`Error`类和`RuntimeException`类继承的异常。从`Error`类继承的异常在控制范围之外，从`RuntimeException`类继承的异常应该在编程时避免。

如果一个方法声明它会抛出一个异常，而这个异常是某个特定类的实例，那么这个方法抛出的异常可能属于这个类，也可能属于这个类的任意一个子类。

如果在子类中覆盖了超类的一个方法，子类方法中声明的检查型异常不能比超类方法中声明的异常更通用。子类方法可以抛出更特定的异常，或者不抛出任何异常。如果超类方法没有抛出任何检查型异常，子类方法也不能抛出任何检查型异常。

1.3 如何抛出异常

抛出异常的步骤如下：

1. 找到一个合适的异常类。
2. 创建这个类的一个对象。
3. 使用 `throw` 关键字，将这个异常类对象抛出。

例如，用 `readData()` 方法读取一个长度为 1024 个字符的文件，然而读到 733 个字符之后文件就结束了。在这种情况下， `EOFException` 异常满足要求，它指示输入过程中意外遇到了 EOF。抛出这个异常的代码如下：

```
String readData(Scanner in) throws EOFException // 声明异常
{
    // ...
    while (...)
    {
        if (!in.hasNext())
        {
            if (n < len)
                throw new EOFException(); // 抛出异常
        }
        // ...
    }
    return s;
}
```

1.4 创建异常类

如果已有的异常类无法满足要求，可以创建一个新的异常类。习惯做法是，自定义的这个类应该包含两个构造器，一个是默认的构造器，另一个是包含详细信息描述的构造器。例如：

```
class FileFormatException extends IOException
{
    public FileFormatException() {}

    public FileFormatException(String gripe)
    {
        super(gripe);
    }
}
```

2 捕获异常

2.1 捕获异常

如果发生了异常，但没有捕获这个异常，程序就会终止，并在控制台上打印一个消息，其中包括这个异常的类型和堆栈轨迹。

用`try/catch`语句块捕获异常。最简单的`try`语句块如下所示：

```
try
{
    // 正常执行的代码
}
catch (ExceptionType e)
{
    // 处理异常
}
```

如果`try`语句块中的任何代码抛出了`catch`子句中指定的一个异常类，程序将跳过`try`语句块的其余代码，执行`catch`子句中的处理器代码。如果`try`语句块中的代码没有抛出任何异常，程序将跳过`catch`子句。如果`try`语句块中的任何代码抛出了`catch`子句中没有声明的异常类型，`try/catch`语句块所在的方法就会立即退出，将异常传递给这个方法的调用者。

捕获异常的例子：

```
public void read(String filename)
{
    try
    {
        FileInputStream in = new FileInputStream(filename);
        int b;
        while ((b = in.read()) != -1)
        {
            // 输入过程
        }
    }
    catch (IOException exception)
    {
        exception.printStackTrace(); // 打印堆栈轨迹
    }
}
```

也可以不处理这个异常，而是将异常传递给调用者，由调用者去处理。例如：

```
public void read(String filename) throws IOException
{
    FileInputStream in = new FileInputStream(filename);
    int b;
    while ((b = in.read()) != -1)
    {
        // 输入过程
    }
}
```

一般经验是，要捕获那些知道如何处理的异常，传播那些不知道如何处理的异常。要传播一个异常，就必须在方法的首部添加 `throws` 子句，提醒调用者这个方法可能会抛出异常。

如果子类方法覆盖超类方法，而超类方法没有抛出异常，这时子类方法不能抛出异常，就必须在子类方法中捕获并处理异常。

2.2 捕获多个异常

在一个 `try` 语句块中可以捕获多个异常类型，并对不同类型的异常做出不同的处理，每个异常类型使用一个单独的 `catch` 子句。例如：

```
try
{
    // 可能抛出异常的代码
}
catch (FileNotFoundException e)
{
    // 处理缺少文件异常
}
catch (UnknownHostException e)
{
    // 处理未知主机异常
}
catch (IOException e)
{
    // 处理 I/O 异常
}
```

异常对象可能包含有关异常性质的信息。要想获得这个对象的更多信息，可以使用 `e.getMessage()` 得到详细的错误信息，或者使用 `e.getClass().getName()` 得到异常对象的实际类型。

在 Java 7 中，同一个 `catch` 子句可以捕获多个异常类型。如果两个异常类型的处理操作是一样的，就可以合并 `catch` 子句。例如：

```
try
{
    // 可能抛出异常的代码
}
catch (FileNotFoundException | UnknownHostException e)
{
    // 处理缺少文件异常或未知主机异常
}
catch (IOException e)
{
    // 处理 I/O 异常
}
```

捕获多个异常时，异常变量隐含为`final`变量，不能在`catch`子句中改变异常变量的值。

2.3 再次抛出异常与异常链

可以在`catch`子句中抛出一个异常。通常，希望改变异常的类型时会这样做。例如：

```
try
{
    // 访问数据库
}
catch (SQLException e)
{
    throw new ServletException("database error: " + e.getMessage());
}
```

还有一种更好的处理方法，可以把原始异常设置为新异常的原因。例如：

```
try
{
    // 访问数据库
}
catch (SQLException original)
{
    ServletException e = new ServletException("database error");
    e.initCause(origonal); // 将 original 设置为 e 的原因
    throw e;
}
```

捕获到新异常后，可以使用下面这条语句获取原始异常：

```
Throwable original = caughtException.getCause();
```

如果在一个方法中发生了一个检查型异常，但这个方法不允许抛出检查型异常，那么包装技术很有用。可以捕获这个检查型异常，并将它包装成一个运行时异常。

2.4 `finally` 子句

`finally` 语句放在 `try/catch` 语句块之后，不管是否有异常被捕获， `finally` 子句中的代码都会执行。例如：

```
FileInputStream in = new FileInputStream(...);
try
{
    // 1
    // 可能抛出异常的代码
    // 2
}
catch (IOException e)
{
    // 3
    // 处理异常
    // 4
}
finally
{
    // 5
    in.close(); // 关闭输入流
}
// 6
```

这段代码的执行情况分析如下：

1. 如果 `try` 语句块中没有抛出异常，执行顺序是 1、2、5、6。
2. 如果 `try` 语句块中抛出异常，并在 `catch` 子句中捕获。
   （1）如果 `catch` 子句没有抛出异常，执行顺序是 1、3、4、5、6。
   （2）如果 `catch` 子句抛出异常，执行顺序是 1、3、5。
3. 如果 `try` 语句块中抛出异常，但没有任何 `catch` 子句捕获这个异常，执行顺序是 1、5。

`try` 语句可以只有 `finally` 子句，而没有 `catch` 子句。例如：

```
InputStream in = ...;
try
{
    // 可能抛出异常的代码
}
finally
{
    in.close();
}
```

如果`finally`子句中也有可能抛出异常，可以将这个`try-catch-finally`语句块放在另一个`try`语句块内部。例如：

```
InputStream in = ...;
try
{
    try
    {
        // 可能抛出异常的代码
    }
    finally
    {
        in.close();
    }
}
catch (IOException e)
{
    // 处理异常
}
```

内层的`try`语句块确保关闭输入流，外层的`try`语句块确保报告出现的错误，既实现了原有的功能，又能处理`finally`子句中的异常。

`finally`子句的体要用于清理资源，不要把改变控制流的语句（`return`，`throw`，`break`，`continue`）放在`finally`子句中。如果`finally`子句中有`return`语句，这个返回值会遮蔽原来的返回值。例如：

```
public static int parseInt(String s)
{
    try
    {
        return Integer.parseInt(s);
    }
    finally
    {
        return 0; // 错误
    }
}
```

如果 `try` 块正常执行，会在方法返回前执行 `finally` 子句，这就使得方法最后返回 0，而忽略原先的返回值。如果 `try` 块抛出异常，仍然会执行 `finally` 子句，这时 `return` 语句甚至会遮蔽这个异常。因此 `finally` 子句中不能有 `return` 语句。其他改变控制流的语句同理。

2.5 try-with-Resources语句

`finally` 子句用于清理资源。在 Java 7 中，可以使用 try-with-Resources 语句清理资源。try-with-Resources 语句的最简形式为：

```
try (Resource res = ...)
{
    // 使用 res 完成操作
}
```

要求资源所属的类必须实现 `AutoCloseable` 接口。 `AutoCloseable` 接口有一个 `close()` 方法，当 `try` 块退出时，无论是否抛出异常，都会自动调用 `close()` 方法关闭资源。例如：

```
try (Scanner in = new Scanner(
        new FileInputStream("/user/share/dict.words"), StandardCharsets.UTF_8))
{
    while (in.hasNext())
    {
        System.out.println(in.next());
    }
} // 自动调用 in.close()
```

还可以指定多个资源。例如：

```
try (Scanner in = new Scanner(
            new FileInputStream("/user/share/dict.words"), StandardCharsets.UTF_8);
     PrintWriter out = new PrintWriter("out.txt", StandardCharsets.UTF_8))
{
    while (in.hasNext())
    {
        out.println(in.next().toUpperCase());
    }
} // 自动调用 in.close() 和 out.close()
```

在 Java 9 中，可以在 `try` 首部的小括号中提供之前声明的事实最终变量。例如：

```
public static void printAll(String[] lines, PrintWriter out)
{
    try (out)
    {
        for (String line : lines)
            out.println(line);
    } // 自动调用 out.close()
}
```

如果 `try` 块抛出一个异常， `close()` 方法也抛出一个异常， `try` 块的异常会重新抛出，而 `close()` 方法抛出的异常会被抑制。这些异常将自动捕获，并由 `addSuppressed()` 方法将 `close()` 方法抛出的异常附加到 `try` 块的异常。如果要得到被抑制的异常，可以调用 `getSuppressed()` 方法，它会返回从 `close()` 方法抛出并被抑制的异常数组。

try-with-Resources 语句也可以有 `catch` 子句和 `finally` 子句， `close()` 方法的调用会在 `try` 块结束之后、 `catch` 子句之前执行。

2.6 分析堆栈轨迹元素

堆栈轨迹是程序执行过程中某个特定点上所有挂起的方法调用的一个列表。当 Java 程序因为一个未捕获的异常而终止时，就会显示堆栈轨迹。可以调用 `Throwable` 类的 `printStackTrace()` 方法访问堆栈轨迹的文本表述信息。例如：

```
Throwable t = new Throwable();
StringWriter out = new StringWriter();
t.printStackTrace(new PrintWriter(out));
String description = out.toString();
```

`printStackTrace()` 方法的签名为：

```
void printStackTrace() // 将堆栈轨迹打印到标准错误流
void printStackTrace(PrintStream s) // 将堆栈轨迹打印到指定的打印流
void printStackTrace(PrintWriter s) // 将堆栈轨迹打印到指定的打印作者
```

一种更灵活的方法是使用`StackWalker`类，它会生成一个`StackWalker.StackFrame`实例流，其中每个实例分别描述一个栈帧。`StackWalker`类和`StackWalker.StackFrame`类的 API 如下：

```
/* java.lang.StackWalker */

// 得到一个 StackWalker 实例。StackWalker.Option 是一个枚举类。
static StackWalker getInstance()
static StackWalker getInstance(StackWalker.Option option)
static StackWalker getInstance(Set<StackWalker.Option> options)

// 在每一个栈帧上完成给定的动作，从最近调用的方法开始
forEach(Consumer<? super StackWalker.StackFrame> action)

// 对一个栈帧流应用给定的函数，返回这个函数的结果
walk(Function<? super Stream<StackWalker.StackFrame>, ? extends T> function)


/* java.lang.StackWalker.StackFrame */

// 得到包含该元素执行点的源文件的文件名，如果这个信息不可用则返回 null
String getFileName()

// 得到包含该元素执行点的源文件的行号，如果这个信息不可用则返回 -1
int getLineNumber()

// 得到方法包含该元素执行点的类的完全限定名
String getClassName()

// 得到方法包含该元素执行点的类的 Class 对象
// 如果这个栈遍历器不是用 RETAIN_CLASS_REFERENCE 选项构造的，则会抛出一个异常
String getDeclaringClass()

// 得到包含该元素执行点的方法的方法名。构造器的方法名为 <init>，静态初始化器的方法名为 <clinit>
// 无法区分同名的重载方法
String getMethodName()

// 如果这个元素的执行点在一个原生方法中，则返回 true
boolean isNativeMethod()

// 返回一个格式化字符串，包含类和方法名、文件名以及行号
String toString()
```

下面的例子打印了递归阶乘函数的堆栈轨迹：

```java
import java.util.*;

public class StackTraceTest
{
    /**
     * 计算阶乘
     * @param n 一个非负整数
     * @return n! = 1 * 2 * ... * n
     */
    public static int factorial(int n)
    {
        System.out.println("factorial(" + n +"):");

        StackWalker walker = StackWalker.getInstance();
        walker.forEach(System.out::println);

        int r;
        if (n <= 1)
            r = 1;
        else
            r = n * factorial(n - 1);

        System.out.println("return " + r);
        return r;
    }

    public static void main(String[] args)
    {
        try (Scanner in = new Scanner(System.in))
        {
            System.out.print("Enter n: ");
            int n = in.nextInt();
            factorial(n);
        }
    }
}
```

3 使用异常的技巧

3.1 异常处理不能代替简单的测试

如果在弹栈时栈为空，会抛出`EmptyStackException`异常。在弹栈时，可以先判断栈是否为空来决定是否弹栈，也可以强制要求不论栈是否为空都弹栈，使用`try/catch`语句块捕获可能抛出的异常。经过测试，前者运行速度更快。

因此，应该尽量避免出现异常，只在异常无法避免的地方使用异常处理。

3.2 不要过分地细化异常

一个不好的例子：

```
PrintStream out;
Stack s;

for(i = 0; i < 100; i++)
{
    try
    {
        n = s.pop();
    }
    catch(EmptyStackException e)
    {
        // 处理栈空异常
    }

    try
    {
        out.writeInt(n);
    }
    catch (IOException e)
    {
        // 处理输出异常
    }
}
```

这个例子将每一条语句都分装在一个独立的`try`语句块中，这种编程方式会导致代码量的急剧膨胀。更好的编程方式为：

```
try
{
    for (i = 0; i < 100; i++)
    {
        n = s.pop();
        out.writeInt(n);
    }
}
catch(EmptyStackException e)
{
    // 处理栈空异常
}
catch (IOException e)
{
    // 处理输出异常
}
```

这样代码看起来更清晰，也满足了异常处理的一个承诺：将正常处理与错误处理分开。

3.3 充分利用异常层次结构

不要只抛出`RuntimeException`异常，应该寻找一个合适的子类或创建自己的异常类。

不要只捕获`Throwable`异常，否则，这会使代码更难读、更难维护。

不要为逻辑错误抛出检查型异常。

如果能够将一种异常转换成另一种更合适的异常，尽量完成这种转换。例如，在解析某个文件中的一个整数时，可以捕获`NumberFormatException`异常，然后将它转换成`IOException`的一个子类。

3.4 不要压制异常

`catch`子句什么都不做，只捕获异常而不处理异常，这种做法是不对的。例如：

```
try
{
    // 可能抛出异常的代码
}
catch (Exception e)
{} // 错误
```

这种做法可以通过编译。但是一旦出现异常，这个异常会被忽略，可能造成更大范围的错误。

3.5 早抛出，晚捕获

在出错的地方，不要用一个特殊的返回值代替抛出异常。因为这个特殊的返回值可能会在其他地方引发新的异常，不利于问题的溯源，不如在问题的源头抛出异常。

在某个方法中抛出的异常不一定要就地捕获，不妨继续向上传递这个异常。更高层的方法通常可以更好地通知用户发生了错误，或者放弃不成功的命令。