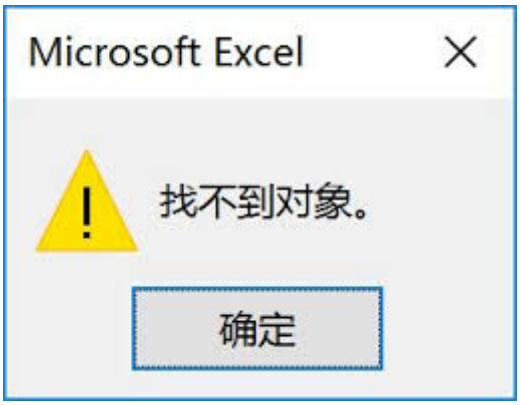
03 异常处理

程序运行的时候，经常会发生各种错误。

比如，使用Excel的时候，它有时候会报错：



本章我们讨论如何在Java程序中处理各种异常情况。

Java的异常

在计算机程序运行的过程中，总是会出现各种各样的错误。

有一些错误是用户造成的，比如，希望用户输入一个`int`类型的年龄，但是用户的输入是`abc`：

```
// 假设用户输入了abc：
String s = "abc";
int n = Integer.parseInt(s); // NumberFormatException!
```

程序想要读写某个文件的内容，但是用户已经把它删除了：

```
// 用户删除了该文件：
String t = readFile("C:\\abc.txt"); //
FileNotFoundException!
```

还有一些错误是随机出现，并且永远不可能避免的。比如：

- 网络突然断了，连接不到远程服务器；
- 内存耗尽，程序崩溃了；
- 用户点“打印”，但根本没有打印机；
- ……

所以，一个健壮的程序必须处理各种各样的错误。

所谓错误，就是程序调用某个函数的时候，如果失败了，就表示出错。

调用方如何获知调用失败的信息？有两种方法：

方法一：约定返回错误码。

例如，处理一个文件，如果返回`0`，表示成功，返回其他整数，表示约定的错误码：

```
int code = processFile("C:\\test.txt");
if (code == 0) {
    // ok:
} else {
    // error:
    switch (code) {
    case 1:
        // file not found:
    case 2:
        // no read permission:
    default:
        // unknown error:
    }
}
```

因为使用`int`类型的错误码，想要处理就非常麻烦。这种方式常见于底层C函数。

方法二：在语言层面上提供一个异常处理机制。

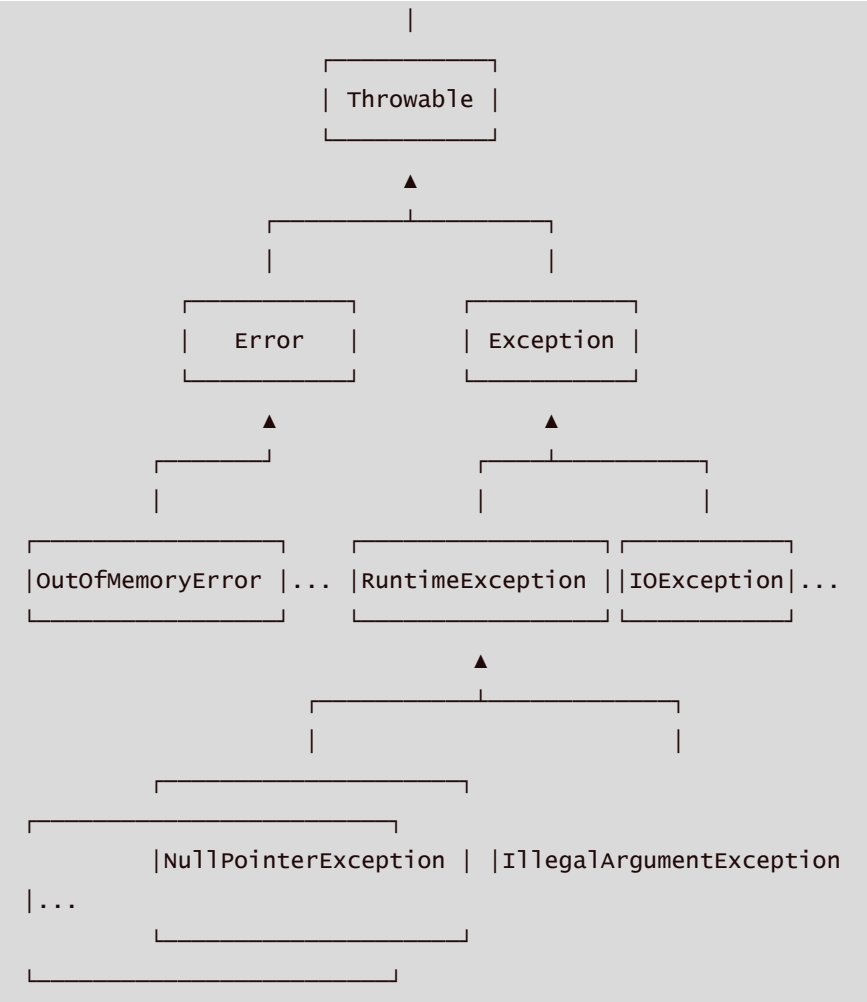
Java内置了一套异常处理机制，总是使用异常来表示错误。

异常是一种`class`，因此它本身带有类型信息。异常可以在任何地方抛出，但只需要在上层捕获，这样就和方法调用分离了：

```
try {
    String s = processFile(“C:\\test.txt”);
    // ok:
} catch (FileNotFoundException e) {
    // file not found:
} catch (SecurityException e) {
    // no read permission:
} catch (IOException e) {
    // io error:
} catch (Exception e) {
    // other error:
}
```

因为Java的异常是`class`，它的继承关系如下：

```
                ┌────────────┐
                │  Object    │
                └────────────┘
                      ▲
```



从继承关系可知：`Throwable`是异常体系的根，它继承自`Object`。`Throwable`有两个体系：`Error`和`Exception`，`Error`表示严重的错误，程序对此一般无能为力，例如：

- `OutOfMemoryError`：内存耗尽
- `NoClassDefFoundError`：无法加载某个Class
- `StackOverflowError`：栈溢出

而`Exception`则是运行时的错误，它可以被捕获并处理。

某些异常是应用程序逻辑处理的一部分，应该捕获并处理。例如：

- `NumberFormatException`：数值类型的格式错误
- `FileNotFoundException`：未找到文件
- `SocketException`：读取网络失败

还有一些异常是程序逻辑编写不对造成的，应该修复程序本身。例如：

- `NullPointerException`：对某个`null`的对象调用方法或字段
- `IndexOutOfBoundsException`：数组索引越界

`Exception`又分为两大类：

1. `RuntimeException`以及它的子类；

2. 非`RuntimeException`（包括`IOException`、`ReflectiveOperationException`等等）

Java规定：

- 必须捕获的异常，包括`Exception`及其子类，但不包括`RuntimeException`及其子类，这种类型的异常称为Checked Exception。
- 不需要捕获的异常，包括`Error`及其子类，`RuntimeException`及其子类。

捕获异常

捕获异常使用`try...catch`语句，把可能发生异常的代码放到`try {...}`中，然后使用`catch`捕获对应的`Exception`及其子类：

```
// try...catch
import java.io.UnsupportedEncodingException;
import java.util.Arrays;
public class Main {
    public static void main(String[] args) {
        byte[] bs = toGBK("中文");
        System.out.println(Arrays.toString(bs));
    }

    static byte[] toGBK(String s) {
        try {
            // 用指定编码转换String为byte[]:
            return s.getBytes("GBK");
        } catch (UnsupportedEncodingException e) {
            // 如果系统不支持GBK编码，会捕获到
UnsupportedEncodingException:
            System.out.println(e); // 打印异常信息
            return s.getBytes(); // 尝试使用用默认编码
        }
    }
}
```

如果我们不捕获`UnsupportedEncodingException`，会出现编译失败的问题：

```
// try...catch
import java.io.UnsupportedEncodingException;
import java.util.Arrays;
public class Main {
    public static void main(String[] args) {
        byte[] bs = toGBK("中文");
        System.out.println(Arrays.toString(bs));
    }

    static byte[] toGBK(String s) {
        return s.getBytes("GBK");
    }
}
```

编译器会报错，错误信息类似：`unreported exception UnsupportedEncodingException; must be caught or declared to be thrown`，并且准确地指出需要捕获的语句是`return s.getBytes("GBK");`。意思是说，像`UnsupportedEncodingException`这样的Checked Exception，必须被捕获。

这是因为`String.getBytes(String)`方法定义是：

```
public byte[] getBytes(String charsetName) throws
UnsupportedEncodingException {
    // ...
}
```

在方法定义的时候，使用`throws Xxx`表示该方法可能抛出的异常类型。调用方在调用的时候，必须强制捕获这些异常，否则编译器会报错。

在`toGBK()`方法中，因为调用了`String.getBytes(String)`方法，就必须捕获`UnsupportedEncodingException`。我们也可以不捕获它，而是在方法定义处用throws表示`toGBK()`方法可能会抛出`UnsupportedEncodingException`，就可以让`toGBK()`方法通过编译器检查：

```
// try...catch
import java.io.UnsupportedEncodingException;
import java.util.Arrays;
public class Main {
    public static void main(String[] args) {
        byte[] bs = toGBK("中文");
        System.out.println(Arrays.toString(bs));
    }

    static byte[] toGBK(String s) throws
UnsupportedEncodingException {
        return s.getBytes("GBK");
    }
}
```

上述代码仍然会得到编译错误，但这一次，编译器提示的不是调用`return s.getBytes("GBK");`的问题，而是`byte[] bs = toGBK("中文");`。因为在`main()`方法中，调用`toGBK()`，没有捕获它声明的可能抛出的`UnsupportedEncodingException`。

修复方法是在`main()`方法中捕获异常并处理：

```
// try...catch
import java.io.UnsupportedEncodingException;
import java.util.Arrays;
public class Main {
    public static void main(String[] args) {
        try {
            byte[] bs = toGBK("中文");
            System.out.println(Arrays.toString(bs));
        } catch (UnsupportedEncodingException e) {
            System.out.println(e);
        }
    }

    static byte[] toGBK(String s) throws
UnsupportedEncodingException {
        // 用指定编码转换String为byte[]:
        return s.getBytes("GBK");
    }
}
```

可见，只要是方法声明的Checked Exception，不在调用层捕获，也必须在更高的调用层捕获。所有未捕获的异常，最终也必须在`main()`方法中捕获，不会出现漏写`try`的情况。这是由编译器保证的。`main()`方法也是最后捕获`Exception`的机会。

如果是测试代码，上面的写法就略显麻烦。如果不想写任何`try`代码，可以直接把`main()`方法定义为`throws Exception`：

```
// try...catch
import java.io.UnsupportedEncodingException;
import java.util.Arrays;
public class Main {
    public static void main(String[] args) throws
Exception {
        byte[] bs = toGBK("中文");
        System.out.println(Arrays.toString(bs));
    }

    static byte[] toGBK(String s) throws
UnsupportedEncodingException {
        // 用指定编码转换String为byte[]:
        return s.getBytes("GBK");
    }
}
```

因为`main()`方法声明了可能抛出`Exception`，也就声明了可能抛出所有的`Exception`，因此在内部就无需捕获了。代价就是一旦发生异常，程序会立刻退出。

还有一些童鞋喜欢在`toGBK()`内部“消化”异常：

```
static byte[] toGBK(String s) {
    try {
        return s.getBytes("GBK");
    } catch (UnsupportedEncodingException e) {
        // 什么也不干
    }
    return null;
```

这种捕获后不处理的方式是非常不好的，即使真的什么也做不了，也要先把异常记录下来：

```
static byte[] toGBK(String s) {
    try {
        return s.getBytes("GBK");
    } catch (UnsupportedEncodingException e) {
        // 先记下来再说:
        e.printStackTrace();
    }
    return null;
```

所有异常都可以调用`printStackTrace()`方法打印异常栈，这是一个简单有用的快速打印异常的方法。

小结

- Java使用异常来表示错误，并通过`try ... catch`捕获异常；
- Java的异常是`class`，并且从`Throwable`继承；
- `Error`是无需捕获的严重错误，`Exception`是应该捕获的可处理的错误；
- `RuntimeException`无需强制捕获，非`RuntimeException`（Checked Exception）需强制捕获，或者用`throws`声明；
- 不推荐捕获了异常但不进行任何处理。

捕获异常

在Java中，凡是可能抛出异常的语句，都可以用`try ... catch`捕获。把可能发生异常的语句放在`try { ... }`中，然后使用`catch`捕获对应的`Exception`及其子类。

多catch语句

可以使用多个`catch`语句，每个`catch`分别捕获对应的`Exception`及其子类。JVM在捕获到异常后，会从上到下匹配`catch`语句，匹配到某个`catch`后，执行`catch`代码块，然后*不再*继续匹配。

简单地说就是：多个`catch`语句只有一个能被执行。例如：

```
public static void main(String[] args) {
    try {
        process1();
        process2();
        process3();
    } catch (IOException e) {
        System.out.println(e);
    } catch (NumberFormatException e) {
        System.out.println(e);
    }
}
```

存在多个`catch`的时候，`catch`的顺序非常重要：子类必须写在前面。例如：

```
public static void main(String[] args) {
    try {
        process1();
        process2();
        process3();
    } catch (IOException e) {
        System.out.println("IO error");
    } catch (UnsupportedEncodingException e) { // 永远捕获
不到
        System.out.println("Bad encoding");
    }
}
```

对于上面的代码，`UnsupportedEncodingException`异常是永远捕获不到的，因为它是`IOException`的子类。当抛出`UnsupportedEncodingException`异常时，会被`catch (IOException e) { ... }`捕获并执行。

因此，正确的写法是把子类放到前面：

```
public static void main(String[] args) {
    try {
        process1();
        process2();
        process3();
    } catch (UnsupportedEncodingException e) {
        System.out.println("Bad encoding");
    } catch (IOException e) {
        System.out.println("IO error");
    }
}
```

finally语句

无论是否有异常发生，如果我们都希望执行一些语句，例如清理工作，怎么写？

可以把执行语句写若干遍：正常执行的放到`try`中，每个`catch`再写一遍。例如：

```
public static void main(String[] args) {
    try {
        process1();
        process2();
        process3();
        System.out.println("END");
    } catch (UnsupportedEncodingException e) {
        System.out.println("Bad encoding");
        System.out.println("END");
    } catch (IOException e) {
        System.out.println("IO error");
        System.out.println("END");
    }
}
```

上述代码无论是否发生异常，都会执行`System.out.println("END");`这条语句。

那么如何消除这些重复的代码？Java的`try ... catch`机制还提供了`finally`语句，`finally`语句块保证有无错误都会执行。上述代码可以改写如下：

```
public static void main(String[] args) {
    try {
        process1();
        process2();
        process3();
    } catch (UnsupportedEncodingException e) {
        System.out.println("Bad encoding");
    } catch (IOException e) {
        System.out.println("IO error");
    } finally {
        System.out.println("END");
    }
}
```

注意`finally`有几个特点：

1. `finally`语句不是必须的，可写可不写；
2. `finally`总是最后执行。

如果没有发生异常，就正常执行`try { ... }`语句块，然后执行`finally`。如果发生了异常，就中断执行`try { ... }`语句块，然后跳转执行匹配的`catch`语句块，最后执行`finally`。

可见，`finally`是用来保证一些代码必须执行的。

某些情况下，可以没有`catch`，只使用`try ... finally`结构。例如：

```
void process(String file) throws IOException {
    try {
        ...
    } finally {
        System.out.println("END");
    }
}
```

因为方法声明了可能抛出的异常，所以可以不写`catch`。

捕获多种异常

如果某些异常的处理逻辑相同，但是异常本身不存在继承关系，那么就得编写多条`catch`子句：

```
public static void main(String[] args) {
    try {
        process1();
        process2();
        process3();
    } catch (IOException e) {
        System.out.println("Bad input");
    } catch (NumberFormatException e) {
        System.out.println("Bad input");
    } catch (Exception e) {
        System.out.println("Unknown error");
    }
}
```

因为处理`IOException`和`NumberFormatException`的代码是相同的，所以我们可以把它两用`|`合并到一起：

```
public static void main(String[] args) {
    try {
        process1();
        process2();
        process3();
    } catch (IOException | NumberFormatException e) { //
IOException或NumberFormatException
        System.out.println("Bad input");
    } catch (Exception e) {
        System.out.println("Unknown error");
    }
}
```

练习

用`try ... catch`捕获异常并处理。

下载练习：捕获异常练习 （推荐使用IDE练习插件快速下载）

小结

使用`try ... catch ... finally`时：

- 多个`catch`语句的匹配顺序非常重要，子类必须放在前面；
- `finally`语句保证了有无异常都会执行，它是可选的；
- 一个`catch`语句也可以匹配多个非继承关系的异常。

抛出异常

异常的传播

当某个方法抛出了异常时，如果当前方法没有捕获异常，异常就会被抛到上层调用方法，直到遇到某个`try ... catch`被捕获为止：

```
public class Main {
    public static void main(String[] args) {
        try {
            process1();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    static void process1() {
        process2();
    }

    static void process2() {
        Integer.parseInt(null); // 会抛出
NumberFormatException
    }
}
```

通过`printStackTrace()`可以打印出方法的调用栈，类似：

```
java.lang.NumberFormatException: null
    at
java.base/java.lang.Integer.parseInt(Integer.java:614)
    at
java.base/java.lang.Integer.parseInt(Integer.java:770)
    at Main.process2(Main.java:16)
    at Main.process1(Main.java:12)
    at Main.main(Main.java:5)
```

`printStackTrace()`对于调试错误非常有用，上述信息表示：`NumberFormatException`是在`java.lang.Integer.parseInt`方法中被抛出的，调用层次从上到下依次是：

1. `main()`调用`process1()`；
2. `process1()`调用`process2()`；
3. `process2()`调用`Integer.parseInt(String)`；
4. `Integer.parseInt(String)`调用`Integer.parseInt(String, int)`。

查看`Integer.java`源码可知，抛出异常的方法代码如下：

```
public static int parseInt(String s, int radix) throws
NumberFormatException {
    if (s == null) {
        throw new NumberFormatException("null");
    }
    ...
}
```

并且，每层调用均给出了源代码的行号，可直接定位。

抛出异常

当发生错误时，例如，用户输入了非法的字符，我们就可以抛出异常。

如何抛出异常？参考`Integer.parseInt()`方法，抛出异常分两步：

1. 创建某个`Exception`的实例；
2. 用`throw`语句抛出。

下面是一个例子：

```
void process2(String s) {
    if (s==null) {
        NullPointerException e = new
NullPointerException();
        throw e;
    }
}
```

实际上，绝大部分抛出异常的代码都会合并写成一行：

```
void process2(String s) {
    if (s==null) {
        throw new NullPointerException();
    }
}
```

如果一个方法捕获了某个异常后，又在`catch`子句中抛出新的异常，就相当于把抛出的异常类型“转换”了：

```
void process1(String s) {
    try {
        process2();
    } catch (NullPointerException e) {
        throw new IllegalArgumentException();
    }
}

void process2(String s) {
    if (s==null) {
        throw new NullPointerException();
    }
}
```

当`process2()`抛出`NullPointerException`后，被`process1()`捕获，然后抛出`IllegalArgumentException()`。

如果在`main()`中捕获`IllegalArgumentException`，我们看看打印的异常栈：

```
public class Main {
    public static void main(String[] args) {
        try {
            process1();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    static void process1() {
        try {
            process2();
        } catch (NullPointerException e) {
            throw new IllegalArgumentException();
        }
    }

    static void process2() {
        throw new NullPointerException();
    }
}
```

打印出的异常栈类似：

```
java.lang.IllegalArgumentException
    at Main.process1(Main.java:15)
    at Main.main(Main.java:5)
```

这说明新的异常丢失了原始异常信息，我们已经看不到原始异常`NullPointerException`的信息了。

为了能追踪到完整的异常栈，在构造异常的时候，把原始的`Exception`实例传进去，新的`Exception`就可以持有原始`Exception`信息。对上述代码改进如下：

```
public class Main {
    public static void main(String[] args) {
        try {
            process1();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    static void process1() {
        try {
            process2();
        } catch (NullPointerException e) {
            throw new IllegalArgumentException(e);
        }
    }

    static void process2() {
        throw new NullPointerException();
    }
}
```

运行上述代码，打印出的异常栈类似：

```
java.lang.IllegalArgumentException:
java.lang.NullPointerException
    at Main.process1(Main.java:15)
    at Main.main(Main.java:5)
Caused by: java.lang.NullPointerException
    at Main.process2(Main.java:20)
    at Main.process1(Main.java:13)
```

注意到`Caused by: Xxx`，说明捕获的`IllegalArgumentException`并不是造成问题的根源，根源在于`NullPointerException`，是在`Main.process2()`方法抛出的。

在代码中获取原始异常可以使用`Throwable.getCause()`方法。如果返回`null`，说明已经是“根异常”了。

有了完整的异常栈的信息，我们才能快速定位并修复代码的问题。

如果我们在`try`或者`catch`语句块中抛出异常，`finally`语句是否会执行？例如：

```
public class Main {
    public static void main(String[] args) {
        try {
            Integer.parseInt("abc");
        } catch (Exception e) {
            System.out.println("catched");
            throw new RuntimeException(e);
        } finally {
            System.out.println("finally");
        }
    }
}
```

上述代码执行结果如下：

```
catched
finally
Exception in thread "main" java.lang.RuntimeException:
java.lang.NumberFormatException: For input string: "abc"
    at Main.main(Main.java:8)
Caused by: java.lang.NumberFormatException: For input
string: "abc"
    at ...
```

第一行打印了`catched`，说明进入了`catch`语句块。第二行打印了`finally`，说明执行了`finally`语句块。

因此，在`catch`中抛出异常，不会影响`finally`的执行。JVM会先执行`finally`，然后抛出异常。

异常屏蔽

如果在执行`finally`语句时抛出异常，那么，`catch`语句的异常还能否继续抛出？例如：

```
public class Main {
    public static void main(String[] args) {
        try {
            Integer.parseInt("abc");
        } catch (Exception e) {
            System.out.println("catched");
            throw new RuntimeException(e);
        } finally {
            System.out.println("finally");
            throw new IllegalArgumentException();
        }
    }
}
```

执行上述代码，发现异常信息如下：

```
catched
finally
Exception in thread "main"
java.lang.IllegalArgumentException
    at Main.main(Main.java:11)
```

这说明`finally`抛出异常后，原来在`catch`中准备抛出的异常就“消失”了，因为只能抛出一个异常。没有被抛出的异常称为“被屏蔽”的异常（Suppressed Exception）。

在极少数的情况下，我们需要获知所有的异常。如何保存所有的异常信息？方法是先用`origin`变量保存原始异常，然后调用`Throwable.addSuppressed()`，把原始异常添加进来，最后在`finally`抛出：

```
public class Main {
    public static void main(String[] args) throws
Exception {
        Exception origin = null;
        try {
            System.out.println(Integer.parseInt("abc"));
        } catch (Exception e) {
            origin = e;
            throw e;
        } finally {
            Exception e = new IllegalArgumentException();
            if (origin != null) {
                e.addSuppressed(origin);
            }
            throw e;
        }
    }
}
```

当`catch`和`finally`都抛出了异常时，虽然`catch`的异常被屏蔽了，但是，`finally`抛出的异常仍然包含了它：

```
Exception in thread "main"
java.lang.IllegalArgumentException
    at Main.main(Main.java:11)
Suppressed: java.lang.NumberFormatException: For input
string: "abc"
    at
java.base/java.lang.NumberFormatException.forInputString(N
umberFormatException.java:65)
    at
java.base/java.lang.Integer.parseInt(Integer.java:652)
    at
java.base/java.lang.Integer.parseInt(Integer.java:770)
    at Main.main(Main.java:6)
```

通过`Throwable.getSuppressed()`可以获取所有的`Suppressed Exception`。

绝大多数情况下，在`finally`中不要抛出异常。因此，我们通常不需要关心`Suppressed Exception`。

练习

如果传入的参数为负，则抛出IllegalArgumentException。

下载练习：抛出异常练习 （推荐使用IDE练习插件快速下载）

小结

- 调用`printStackTrace()`可以打印异常的传播栈，对于调试非常有用；
- 捕获异常并再次抛出新的异常时，应该持有原始异常信息；
- 通常不要在`finally`中抛出异常。如果在`finally`中抛出异常，应该原始异常加入到原有异常中。调用方可通过`Throwable.getSuppressed()`获取所有添加的`Suppressed Exception`。

自定义异常

Java标准库定义的常用异常包括：

```
Exception
|
├─ RuntimeException
|  |
|  ├─ NullPointerException
|  |
|  ├─ IndexOutOfBoundsException
|  |
|  ├─ SecurityException
|  |
|  └─ IllegalArgumentException
```

```
|     |
|     └─ NumberFormatException
|
├─ IOException
|  |
|  ├─ UnsupportedCharsetException
|  |
|  ├─ FileNotFoundException
|  |
|  └─ SocketException
|
├─ ParseException
|
├─ GeneralSecurityException
|
├─ SQLException
|
└─ TimeoutException
```

当我们在代码中需要抛出异常时，尽量使用JDK已定义的异常类型。例如，参数检查不合法，应该抛出`IllegalArgumentException`：

```
static void process1(int age) {
    if (age <= 0) {
        throw new IllegalArgumentException();
    }
}
```

在一个大型项目中，可以自定义新的异常类型，但是，保持一个合理的异常继承体系是非常重要的。

一个常见的做法是自定义一个`BaseException`作为“根异常”，然后，派生出各种业务类型的异常。

`BaseException`需要从一个适合的`Exception`派生，通常建议从`RuntimeException`派生：

```
public class BaseException extends RuntimeException {
}
```

其他业务类型的异常就可以从`BaseException`派生：

```
public class UserNotFoundException extends BaseException {
}

public class LoginFailedException extends BaseException {
}

// ...
```

自定义的`BaseException`应该提供多个构造方法：

```
public class BaseException extends RuntimeException {
    public BaseException() {
        super();
    }

    public BaseException(String message, Throwable cause)
{
        super(message, cause);
    }

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

    public BaseException(Throwable cause) {
        super(cause);
    }
}
```

上述构造方法实际上都是原样照抄`RuntimeException`。这样，抛出异常的时候，就可以选择合适的构造方法。通过IDE可以根据父类快速生成子类的构造方法。

练习

下载练习：从BaseException派生自定义异常 （推荐使用IDE练习插件快速下载）

小结

- 抛出异常时，尽量复用JDK已定义的异常类型；
- 自定义异常体系时，推荐从`RuntimeException`派生“根异常”，再派生出业务异常；
- 自定义异常时，应该提供多种构造方法。

使用断言

断言（Assertion）是一种调试程序的方式。在Java中，使用`assert`关键字来实现断言。

我们先看一个例子：

```
public static void main(String[] args) {
    double x = Math.abs(-123.45);
    assert x >= 0;
    System.out.println(x);
}
```

语句`assert x >= 0;`即为断言，断言条件`x >= 0`预期为`true`。如果计算结果为`false`，则断言失败，抛出`AssertionError`。

使用`assert`语句时，还可以添加一个可选的断言消息：

```
assert x >= 0 : "x must >= 0";
```

这样，断言失败的时候，`AssertionError`会带上消息`x must >= 0`，更加便于调试。

Java断言的特点是：断言失败时会抛出`AssertionError`，导致程序结束退出。因此，断言不能用于可恢复的程序错误，只应该用于开发和测试阶段。

对于可恢复的程序错误，不应该使用断言。例如：

```
void sort(int[] arr) {
    assert arr != null;
}
```

应该抛出异常并在上层捕获：

```
void sort(int[] arr) {
    if (x == null) {
        throw new IllegalArgumentException("array cannot
be null");
    }
}
```

当我们在程序中使用`assert`时，例如，一个简单的断言：

`// assert` Run

断言`x`必须大于`0`，实际上`x`为`-1`，断言肯定失败。执行上述代码，发现程序并未抛出`AssertionError`，而是正常打印了`x`的值。

这是怎么肥四？为什么`assert`语句不起作用？

这是因为JVM默认关闭断言指令，即遇到`assert`语句就自动忽略了，不执行。

要执行`assert`语句，必须给Java虚拟机传递`-enableassertions`（可简写为`-ea`）参数启用断言。所以，上述程序必须在命令行下运行才有效果：

```
$ java -ea Main.java
Exception in thread "main" java.lang.AssertionError
    at Main.main(Main.java:5)
```

还可以有选择地对特定地类启用断言，命令行参数是：`-ea:com.itranswarp.sample.Main`，表示只对`com.itranswarp.sample.Main`这个类启用断言。

或者对特定地包启用断言，命令行参数是：`-ea:com.itranswarp.sample...`（注意结尾有3个`.`），表示对`com.itranswarp.sample`这个包启动断言。

实际开发中，很少使用断言。更好的方法是编写单元测试，后续我们会讲解`JUnit`的使用。

小结

- 断言是一种调试方式，断言失败会抛出`AssertionError`，只能在开发和测试阶段启用断言；
- 对可恢复的错误不能使用断言，而应该抛出异常；
- 断言很少被使用，更好的方法是编写单元测试。

使用JDK Logging

在编写程序的过程中，发现程序运行结果与预期不符，怎么办？当然是用`System.out.println()`打印出执行过程中的某些变量，观察每一步的结果与代码逻辑是否符合，然后有针对性地修改代码。

代码改好了怎么办？当然是删除没有用的`System.out.println()`语句了。

如果改代码又改出问题怎么办？再加上`System.out.println()`。

反复这么搞几次，很快大家就发现使用`System.out.println()`非常麻烦。

怎么办？

解决方法是使用日志。

那什么是日志？日志就是Logging，它的目的是为了取代`System.out.println()`。

输出日志，而不是用`System.out.println()`，有以下几个好处：

1. 可以设置输出样式，避免自己每次都写`"ERROR: " + var`；
2. 可以设置输出级别，禁止某些级别输出。例如，只输出错误日志；
3. 可以被重定向到文件，这样可以在程序运行结束后查看日志；
4. 可以按包名控制日志级别，只输出某些包打的日志；
5. 可以……

总之就是好处很多啦。

那如何使用日志？

因为Java标准库内置了日志包`java.util.logging`，我们可以直接用。先看一个简单的例子：

```
// logging
import java.util.logging.Level;
import java.util.logging.Logger;
public class Hello {
    public static void main(String[] args) {
        Logger logger = Logger.getGlobal();
        logger.info("start process...");
        logger.warning("memory is running out...");
        logger.fine("ignored.");
        logger.severe("process will be terminated...");
    }
}
```

运行上述代码，得到类似如下的输出：

```
Mar 02, 2019 6:32:13 PM Hello main
INFO: start process...
Mar 02, 2019 6:32:13 PM Hello main
WARNING: memory is running out...
Mar 02, 2019 6:32:13 PM Hello main
SEVERE: process will be terminated...
```

对比可见，使用日志最大的好处是，它自动打印了时间、调用类、调用方法等很多有用的信息。

再仔细观察发现，4条日志，只打印了3条，`logger.fine()`没有打印。这是因为，日志的输出可以设定级别。JDK的Logging定义了7个日志级别，从严重到普通：

- SEVERE
- WARNING
- INFO
- CONFIG
- FINE
- FINER
- FINEST

因为默认级别是INFO，因此，INFO级别以下的日志，不会被打印出来。使用日志级别的好处在于，调整级别，就可以屏蔽掉很多调试相关的日志输出。

使用Java标准库内置的Logging有以下局限：

Logging系统在JVM启动时读取配置文件并完成初始化，一旦开始运行`main()`方法，就无法修改配置；

配置不太方便，需要在JVM启动时传递参数`-Djava.util.logging.config.file=`。

因此，Java标准库内置的Logging使用并不是非常广泛。更方便的日志系统我们稍后介绍。

练习

使用logger.severe()打印异常：

```
import java.io.UnsupportedEncodingException;
import java.util.logging.Logger;

public class Main {
    public static void main(String[] args) {
        Logger logger =
Logger.getLogger(Main.class.getName());
        logger.info("Start process...");
        try {
            "".getBytes("invalidCharsetName");
        } catch (UnsupportedEncodingException e) {
            // TODO: 使用logger.severe()打印异常
        }
        logger.info("Process end.");
    }
}
```

下载练习：打印异常 （推荐使用IDE练习插件快速下载）

小结

- 日志是为了替代`System.out.println()`，可以定义格式，重定向到文件等；
- 日志可以存档，便于追踪问题；
- 日志记录可以按级别分类，便于打开或关闭某些级别；
- 可以根据配置文件调整日志，无需修改代码；
- Java标准库提供了`java.util.logging`来实现日志功能。

使用Commons Logging

和Java标准库提供的日志不同，Commons Logging是一个第三方日志库，它是由Apache创建的日志模块。

Commons Logging的特色是，它可以挂接不同的日志系统，并通过配置文件指定挂接的日志系统。默认情况下，Commons Loggin自动搜索并使用Log4j（Log4j是另一个流行的日志系统），如果没有找到Log4j，再使用JDK Logging。

使用Commons Logging只需要和两个类打交道，并且只有两步：

第一步，通过`LogFactory`获取`Log`类的实例； 第二步，使用`Log`实例的方法打日志。

示例代码如下：

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
public class Main {
    public static void main(String[] args) {
        Log log = LogFactory.getLog(Main.class);
        log.info("start...");
        log.warn("end.");
    }
}
```

运行上述代码，肯定会得到编译错误，类似`error: package org.apache.commons.logging does not exist`（找不到`org.apache.commons.logging`这个包）。因为Commons Logging是一个第三方提供的库，所以，必须先把它下载下来。下载后，解压，找到`commons-logging-1.2.jar`这个文件，再把Java源码`Main.java`放到一个目录下，例如`work`目录：

```
work
|
├─ commons-logging-1.2.jar
|
└─ Main.java
```

然后用`javac`编译`Main.java`，编译的时候要指定`classpath`，不然编译器找不到我们引用的`org.apache.commons.logging`包。编译命令如下：

```
javac -cp commons-logging-1.2.jar Main.java
```

如果编译成功，那么当前目录下就会多出一个`Main.class`文件：

```
work
|
├─ commons-logging-1.2.jar
|
├─ Main.java
|
└─ Main.class
```

现在可以执行这个`Main.class`，使用`java`命令，也必须指定`classpath`，命令如下：

```
java -cp .;commons-logging-1.2.jar Main
```

注意到传入的`classpath`有两部分：一个是`.`，一个是`commons-logging-1.2.jar`，用`;`分割。`.`表示当前目录，如果没有这个`.`，JVM不会在当前目录搜索`Main.class`，就会报错。

如果在Linux或macOS下运行，注意`classpath`的分隔符不是`;`，而是`:`：

```
java -cp .:commons-logging-1.2.jar Main
```

运行结果如下：

```
Mar 02, 2019 7:15:31 PM Main main
INFO: start...
Mar 02, 2019 7:15:31 PM Main main
WARNING: end.
```

Commons Logging定义了6个日志级别：

- FATAL
- ERROR
- WARNING
- INFO
- DEBUG
- TRACE

默认级别是`INFO`。

使用Commons Logging时，如果在静态方法中引用`Log`，通常直接定义一个静态类型变量：

```
// 在静态方法中引用Log:
public class Main {
    static final Log log = LogFactory.getLog(Main.class);

    static void foo() {
        log.info("foo");
    }
}
```

在实例方法中引用`Log`，通常定义一个实例变量：

```
// 在实例方法中引用Log:
public class Person {
    protected final Log log =
LogFactory.getLog(getClass());

    void foo() {
        log.info("foo");
    }
}
```

注意到实例变量log的获取方式是`LogFactory.getLog(getClass())`，虽然也可以用`LogFactory.getLog(Person.class)`，但是前一种方式有个非常大的好处，就是子类可以直接使用该`log`实例。例如：

```
// 在子类中使用父类实例化的log:
public class Student extends Person {
    void bar() {
        log.info("bar");
    }
}
```

由于Java类的动态特性，子类获取的`log`字段实际上相当于`LogFactory.getLog(Student.class)`，但却是从父类继承而来，并且无需改动代码。

此外，Commons Logging的日志方法，例如`info()`，除了标准的`info(String)`外，还提供了一个非常有用的重载方法：`info(String, Throwable)`，这使得记录异常更加简单：

```
try {
    // ...
} catch (Exception e) {
    log.error("got exception!", e);
}
```

练习

使用`log.error(String, Throwable)`打印异常。

下载练习：Commons Logging练习 （推荐使用IDE练习插件快速下载）

小结

- Commons Logging是使用最广泛的日志模块；
- Commons Logging的API非常简单；
- Commons Logging可以自动检测并使用其他日志模块。

使用Log4j

前面介绍了Commons Logging，可以作为“日志接口”来使用。而真正的“日志实现”可以使用Log4j。

Log4j是一种非常流行的日志框架，最新版本是2.x。

Log4j是一个组件化设计的日志系统，它的架构大致如下：

```
log.info("User signed in.");
 |
 |   ┌──────────┐    ┌──────────┐    ┌──────────┐    ┌──────────┐
 ├──>| Appender |──->|  Filter  |──->|  Layout  |──->| Console  |
 |   └──────────┘    └──────────┘    └──────────┘    └──────────┘
 |
 |   ┌──────────┐    ┌──────────┐    ┌──────────┐    ┌──────────┐
 ├──>| Appender |──->|  Filter  |──->|  Layout  |──->| File     |
 |   └──────────┘    └──────────┘    └──────────┘    └──────────┘
 |
 |   ┌──────────┐    ┌──────────┐    ┌──────────┐    ┌──────────┐
 └──>| Appender |──->|  Filter  |──->|  Layout  |──->| Socket   |
     └──────────┘    └──────────┘    └──────────┘    └──────────┘
```

当我们使用Log4j输出一条日志时，Log4j自动通过不同的Appender把同一条日志输出到不同的目的地。例如：

- console：输出到屏幕；
- file：输出到文件；
- socket：通过网络输出到远程计算机；
- jdbc：输出到数据库

在输出日志的过程中，通过Filter来过滤哪些log需要被输出，哪些log不需要被输出。例如，仅输出`ERROR`级别的日志。

最后，通过Layout来格式化日志信息，例如，自动添加日期、时间、方法名称等信息。

上述结构虽然复杂，但我们在实际使用的时候，并不需要关心Log4j的API，而是通过配置文件来配置它。

以XML配置为例，使用Log4j的时候，我们把一个`log4j2.xml`的文件放到`classpath`下就可以让Log4j读取配置文件并按照我们的配置来输出日志。下面是一个配置文件的例子：

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
    <Properties>
        <!-- 定义日志格式 -->
        <Property name="log.pattern">%d{MM-dd
HH:mm:ss.SSS} [%t] %-5level
%logger{36}%n%msg%n%n</Property>
        <!-- 定义文件名变量 -->
```

```
        <Property
name="file.err.filename">log/err.log</Property>
        <Property
name="file.err.pattern">log/err.%i.log.gz</Property>
    </Properties>
    <!-- 定义Appender，即目的地 -->
    <Appenders>
        <!-- 定义输出到屏幕 -->
        <Console name="console" target="SYSTEM_OUT">
            <!-- 日志格式引用上面定义的log.pattern -->
            <PatternLayout pattern="${log.pattern}" />
        </Console>
        <!-- 定义输出到文件,文件名引用上面定义的
file.err.filename -->
        <RollingFile name="err" bufferedIO="true"
fileName="${file.err.filename}"
filePattern="${file.err.pattern}">
            <PatternLayout pattern="${log.pattern}" />
            <Policies>
                <!-- 根据文件大小自动切割日志 -->
                <SizeBasedTriggeringPolicy size="1 MB" />
            </Policies>
            <!-- 保留最近10份 -->
            <DefaultRolloverStrategy max="10" />
        </RollingFile>
    </Appenders>
    <Loggers>
        <Root level="info">
            <!-- 对info级别的日志，输出到console -->
            <AppenderRef ref="console" level="info" />
            <!-- 对error级别的日志，输出到err，即上面定义的
RollingFile -->
            <AppenderRef ref="err" level="error" />
        </Root>
    </Loggers>
</Configuration>
```

虽然配置Log4j比较繁琐，但一旦配置完成，使用起来就非常方便。对上面的配置文件，凡是`INFO`级别的日志，会自动输出到屏幕，而`ERROR`级别的日志，不但会输出到屏幕，还会同时输出到文件。并且，一旦日志文件达到指定大小（1MB），Log4j就会自动切割新的日志文件，并最多保留10份。

有了配置文件还不够，因为Log4j也是一个第三方库，我们需要从这里下载Log4j，解压后，把以下3个jar包放到`classpath`中：

- log4j-api-2.x.jar
- log4j-core-2.x.jar
- log4j-jcl-2.x.jar

因为Commons Logging会自动发现并使用Log4j，所以，把上一节下载的`commons-logging-1.2.jar`也放到`classpath`中。

要打印日志，只需要按Commons Logging的写法写，不需要改动任何代码，就可以得到Log4j的日志输出，类似：

```
03-03 12:09:45.880 [main] INFO
com.itranswarp.learnjava.Main
Start process...
```

最佳实践

在开发阶段，始终使用Commons Logging接口来写入日志，并且开发阶段无需引入Log4j。如果需要把日志写入文件， 只需要把正确的配置文件和Log4j相关的jar包放入`classpath`，就可以自动把日志切换成使用Log4j写入，无需修改任何代码。

练习

根据配置文件，观察Log4j写入的日志文件。

下载练习：commons logging + log4j （推荐使用IDE练习插件快速下载）

小结

- 通过Commons Logging实现日志，不需要修改代码即可使用Log4j；
- 使用Log4j只需要把log4j2.xml和相关jar放入classpath；
- 如果要更换Log4j，只需要移除log4j2.xml和相关jar；
  只有扩展Log4j时，才需要引用Log4j的接口（例如，将日志加密写入数据库的功能，需要自己开发）。

使用SLF4J和Logback

前面介绍了Commons Logging和Log4j这一对好基友，它们一个负责充当日志API，一个负责实现日志底层，搭配使用非常便于开发。

有的童鞋可能还听说过SLF4J和Logback。这两个东东看上去也像日志，它们又是啥？

其实SLF4J类似于Commons Logging，也是一个日志接口，而Logback类似于Log4j，是一个日志的实现。

为什么有了Commons Logging和Log4j，又会蹦出来SLF4J和Logback？这是因为Java有着非常悠久的开源历史，不但OpenJDK本身是开源的，而且我们用到的第三方库，几乎全部都是开源的。开源生态丰富的一个特定就是，同一个功能，可以找到若干种互相竞争的开源库。

因为对Commons Logging的接口不满意，有人就搞了SLF4J。因为对Log4j的性能不满意，有人就搞了Logback。

我们先来看看SLF4J对Commons Logging的接口有何改进。在Commons Logging中，我们要打印日志，有时候得这么写：

```
int score = 99;
p.setScore(score);
log.info("Set score " + score + " for Person " +
p.getName() + " ok.");
```

拼字符串是一个非常麻烦的事情，所以SLF4J的日志接口改进成这样了：

```
int score = 99;
p.setScore(score);
logger.info("Set score {} for Person {} ok.", score,
p.getName());
```

我们靠猜也能猜出来，SLF4J的日志接口传入的是一个带占位符的字符串，用后面的变量自动替换占位符，所以看起来更加自然。

如何使用SLF4J？它的接口实际上和Commons Logging几乎一模一样：

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

class Main {
    final Logger logger =
LoggerFactory.getLogger(getClass());
}
```

对比一下Commons Logging和SLF4J的接口：

COMMONS LOGGING	SLF4J
org.apache.commons.logging.Log	org.slf4j.Logger
org.apache.commons.logging.LogFactory	org.slf4j.LoggerFactory

不同之处就是Log变成了Logger，LogFactory变成了LoggerFactory。

使用SLF4J和Logback和前面讲到的使用Commons Logging加Log4j是类似的，先分别下载SLF4J和Logback，然后把以下jar包放到classpath下：

- slf4j-api-1.7.x.jar
- logback-classic-1.2.x.jar
- logback-core-1.2.x.jar

然后使用SLF4J的Logger和LoggerFactory即可。和Log4j类似，我们仍然需要一个Logback的配置文件，把`logback.xml`放到classpath下，配置如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>

    <appender name="CONSOLE"
class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
```

```
            <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level
%logger{36} - %msg%n</pattern>
        </encoder>
    </appender>

    <appender name="FILE"
class="ch.qos.logback.core.rolling.RollingFileAppender">
        <encoder>
            <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level
%logger{36} - %msg%n</pattern>
            <charset>utf-8</charset>
        </encoder>
        <file>log/output.log</file>
        <rollingPolicy
class="ch.qos.logback.core.rolling.FixedWindowRollingPolic
y">

<fileNamePattern>log/output.log.%i</fileNamePattern>
        </rollingPolicy>
        <triggeringPolicy
class="ch.qos.logback.core.rolling.SizeBasedTriggeringPoli
cy">
            <MaxFileSize>1MB</MaxFileSize>
        </triggeringPolicy>
    </appender>

    <root level="INFO">
        <appender-ref ref="CONSOLE" />
        <appender-ref ref="FILE" />
    </root>
</configuration>
```

运行即可获得类似如下的输出：

```
13:15:25.328 [main] INFO  com.itranswarp.learnjava.Main -
Start process...
```

从目前的趋势来看，越来越多的开源项目从Commons Logging加Log4j转向了SLF4J加Logback。

练习

根据配置文件，观察Logback写入的日志文件。

下载练习：slf4j+logback （推荐使用IDE练习插件快速下载）

小结

- SLF4J和Logback可以取代Commons Logging和Log4j；

- 始终使用SLF4J的接口写入日志，使用Logback只需要配置，不需要修改代码。