

# 认识异常

## 本章目标

- 了解异常的背景
- 掌握异常的基本用法
- 认识Java异常体系
- 学会自定义异常类

## 1. 异常的背景

### 初识异常

我们曾经的代码中已经接触了一些 "异常" 了. 例如:

#### 除以 0

```
System.out.println(10 / 0);

// 执行结果
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

#### 数组下标越界

```
int[] arr = {1, 2, 3};
System.out.println(arr[100]);

// 执行结果
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 100
```

#### 访问 null 对象

```
public class Test {
    public int num = 10;
    public static void main(String[] args) {
        Test t = null;
        System.out.println(t.num);
    }
}

// 执行结果
Exception in thread "main" java.lang.NullPointerException
```

所谓异常指的就是程序在 **运行时** 出现错误时通知调用者的一种机制.

关键字 "运行时"

有些错误是这样的, 例如将 `System.out.println` 拼写错了, 写成了 `system.out.println`. 此时编译过程中就会出错, 这是 "编译期" 出错.

而运行时指的是程序已经编译通过得到 `class` 文件了, 再由 JVM 执行过程中出现的错误.

异常的种类有很多, 不同种类的异常具有不同的含义, 也有不同的处理方式.

## 防御式编程

错误在代码中是客观存在的. 因此我们要让程序出现问题的时候及时通知程序猿. 我们有两种主要的方式

**LBYL:** Look Before You Leap. 在操作之前就做充分的检查.

**EAFP:** It's Easier to Ask Forgiveness than Permission. "事后获取原谅比事前获取许可更容易". 也就是先操作, 遇到问题再处理.

**注意!!!!** 上面这两个概念千万不要背!

其实很好理解, 举个栗子~~

比如汤老湿年轻的时候, 和你们师娘刚开始谈对象. 我们都知道, 谈对象需要有一些亲密的动作, 比如 "拉小手" 这种. emmmmm 问题来了, 汤老湿去拉师娘的小手有两种方式:

- a) 老湿说, 妹子, 我拉你小手可以嘛? 获取妹子的同意后, 再拉手(这就是 LBYL).
- b) 老湿趁妹子不备, 直接拉住. 大不了妹子生气了给老湿一巴掌, 老湿再道歉就是(这就是 EAFP).

异常的核心思想就是 EAFP.

## 异常的好处

例如, 我们用伪代码演示一下开始一局王者荣耀的过程.

**LBYL 风格的代码(不使用异常)**

```
boolean ret = false;
ret = 登陆游戏();
if (!ret) {
    处理登陆游戏错误;
    return;
}
ret = 开始匹配();
if (!ret) {
    处理匹配错误;
    return;
}
ret = 游戏确认();
if (!ret) {
    处理游戏确认错误;
    return;
}
ret = 选择英雄();
if (!ret) {
    处理选择英雄错误;
    return;
}
ret = 载入游戏画面();
if (!ret) {
    处理载入游戏错误;
    return;
}
```

```
}  
.....
```

### EAFP 风格的代码(使用异常)

```
try {  
    登陆游戏();  
    开始匹配();  
    游戏确认();  
    选择英雄();  
    载入游戏画面();  
    ...  
} catch (登陆游戏异常) {  
    处理登陆游戏异常;  
}  
} catch (开始匹配异常) {  
    处理开始匹配异常;  
}  
} catch (游戏确认异常) {  
    处理游戏确认异常;  
}  
} catch (选择英雄异常) {  
    处理选择英雄异常;  
}  
} catch (载入游戏画面异常) {  
    处理载入游戏画面异常;  
}  
}  
.....
```

对比两种不同风格的代码, 我们可以发现, 使用第一种方式, 正常流程和错误处理流程代码混在一起, 代码整体显的比较混乱. 而第二种方式正常流程和错误流程是分离开的, 更容易理解代码.

## 2. 异常的基本用法

### 捕获异常

#### 基本语法

```
try{  
    有可能出现异常的语句 ;  
}[catch (异常类型 异常对象) {  
} ... ]  
[finally {  
    异常的出口  
}]
```

- try 代码块中放的是可能出现异常的代码.
- catch 代码块中放的是出现异常后的处理行为.
- finally 代码块中的代码用于处理善后工作, 会在最后执行.
- 其中 catch 和 finally 都可以根据情况选择加或者不加.

#### 代码示例1 不处理异常

```
int[] arr = {1, 2, 3};
System.out.println("before");
System.out.println(arr[100]);
System.out.println("after");

// 执行结果
before
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 100
```

我们发现一旦出现异常, 程序就终止了. after 没有正确输出.

#### 代码示例2 使用 try catch 后的程序执行过程

```
int[] arr = {1, 2, 3};

try {
    System.out.println("before");
    System.out.println(arr[100]);
    System.out.println("after");
} catch (ArrayIndexOutOfBoundsException e) {
    // 打印出现异常的调用栈
    e.printStackTrace();
}
System.out.println("after try catch");

// 执行结果
before
java.lang.ArrayIndexOutOfBoundsException: 100
    at demo02.Test.main(Test.java:10)
after try catch
```

我们发现, 一旦 try 中出现异常, 那么 try 代码块中的程序就不会继续执行, 而是交给 catch 中的代码来执行. catch 执行完毕会继续往下执行.

#### 关于异常的处理方式

异常的种类有很多, 我们要根据不同的业务场景来决定.

对于比较严重的问题(例如和算钱相关的场景), 应该让程序直接崩溃, 防止造成更严重的后果

对于不太严重的问题(大多数场景), 可以记录错误日志, 并通过监控报警程序及时通知程序员

对于可能会恢复的问题(和网络相关的场景), 可以尝试进行重试.

在我们当前的代码中采取的是经过简化的第二种方式. 我们记录的错误日志是出现异常的方法调用信息, 能很快速的让我们找到出现异常的位置. 以后在实际工作中我们会采取更完备的方式来记录异常信息.

#### 关于 "调用栈"

方法之间是存在相互调用关系的, 这种调用关系我们可以用 "调用栈" 来描述. 在 JVM 中有一块内存空间称为 "虚拟机栈" 专门存储方法之间的调用关系. 当代码中出现异常的时候, 我们就可以使用 `e.printStackTrace();` 的方式查看出现异常代码的调用栈.

### 代码示例3 catch 只能处理对应种类的异常

我们修改了代码, 让代码抛出的是空指针异常.

```
int[] arr = {1, 2, 3};

try {
    System.out.println("before");
    arr = null;
    System.out.println(arr[100]);
    System.out.println("after");
} catch (ArrayIndexOutOfBoundsException e) {
    e.printStackTrace();
}
System.out.println("after try catch");

// 执行结果
before
Exception in thread "main" java.lang.NullPointerException
    at demo02.Test.main(Test.java:11)
```

此时, catch 语句不能捕获到刚才的空指针异常. 因为异常类型不匹配.

### 代码示例4 catch 可以有多个

```
int[] arr = {1, 2, 3};

try {
    System.out.println("before");
    arr = null;
    System.out.println(arr[100]);
    System.out.println("after");
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("这是个数组下标越界异常");
    e.printStackTrace();
} catch (NullPointerException e) {
    System.out.println("这是个空指针异常");
    e.printStackTrace();
}
System.out.println("after try catch");

// 执行结果
before
这是个空指针异常
java.lang.NullPointerException
    at demo02.Test.main(Test.java:12)
after try catch
```

一段代码可能会抛出多种不同的异常, 不同的异常有不同的处理方式. 因此可以搭配多个 catch 代码块.

如果多个异常的处理方式是完全相同, 也可以写成这样

```
catch (ArrayIndexOutOfBoundsException | NullPointerException e) {
    ...
}
```

**代码示例5** 也可以用 `catch` 捕获所有异常(不推荐)

```
int[] arr = {1, 2, 3};

try {
    System.out.println("before");
    arr = null;
    System.out.println(arr[100]);
    System.out.println("after");
} catch (Exception e) {
    e.printStackTrace();
}
System.out.println("after try catch");

// 执行结果
before
java.lang.NullPointerException
    at demo02.Test.main(Test.java:12)
after try catch
```

由于 `Exception` 类是所有异常类的父类, 因此可以用这个类型表示捕捉所有异常.

备注: `catch` 进行类型匹配的时候, 不光会匹配相同类型的异常对象, 也会捕捉目标异常类型的子类对象.

如刚才的代码, `NullPointerException` 和 `ArrayIndexOutOfBoundsException` 都是 `Exception` 的子类, 因此都能被捕获到.

**代码示例6** `finally` 表示最后的善后工作, 例如释放资源

```
int[] arr = {1, 2, 3};

try {
    System.out.println("before");
    arr = null;
    System.out.println(arr[100]);
    System.out.println("after");
} catch (Exception e) {
    e.printStackTrace();
} finally {
    System.out.println("finally code");
}

// 执行结果
before
java.lang.NullPointerException
    at demo02.Test.main(Test.java:12)
finally code
```

无论是否存在异常, `finally` 中的代码一定会执行到. 保证最终一定会执行到 `Scanner` 的 `close` 方法.

**代码示例7** 使用 `try` 负责回收资源

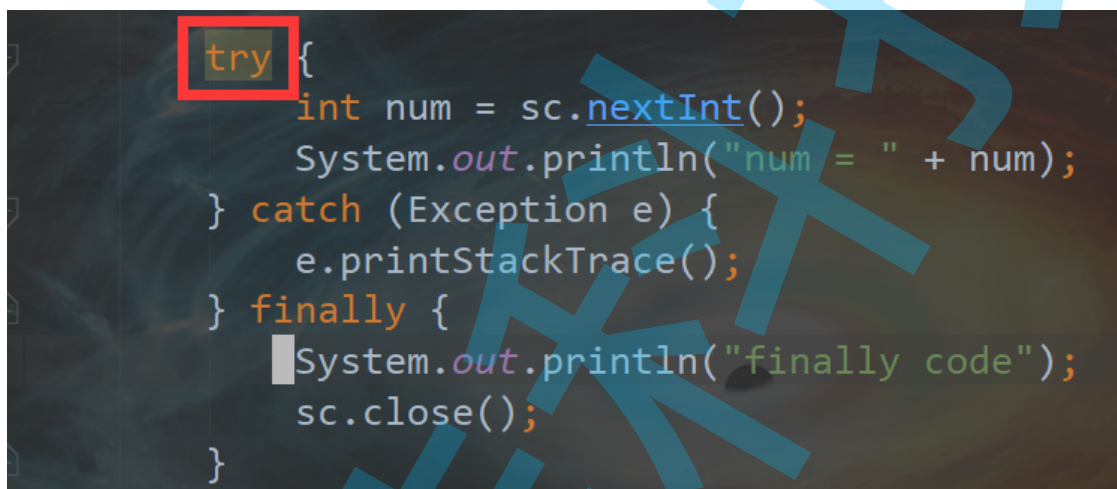
刚才的代码可以有一种等价写法, 将 Scanner 对象在 try 的 () 中创建, 就能保证在 try 执行完毕后自动调用 Scanner 的 close 方法.

```
try (Scanner sc = new Scanner(System.in)) {  
    int num = sc.nextInt();  
    System.out.println("num = " + num);  
} catch (Exception e) {  
    e.printStackTrace();  
}
```

### 小技巧

IDEA 能自动检查我们的代码风格, 并给出一些更好的建议.

如我们之前写的代码, 在 try 上有一个"加深底色", 这时 IDEA 针对我们的代码提出了一些更好的建议.



```
try {  
    int num = sc.nextInt();  
    System.out.println("num = " + num);  
} catch (Exception e) {  
    e.printStackTrace();  
} finally {  
    System.out.println("finally code");  
    sc.close();  
}
```

此时把光标放在 try 上悬停, 会给出原因. 按下 alt + enter, 会弹出一个改进方案的弹窗. 我们选择其中的



此时我们的代码就自动被 IDEA 调整成上面的 代码示例7 的模样

**代码示例8** 如果本方法中没有合适的处理异常的方式, 就会沿着调用栈向上传递

```
public static void main(String[] args) {  
    try {  
        func();  
    } catch (ArrayIndexOutOfBoundsException e) {  
        e.printStackTrace();  
    }  
    System.out.println("after try catch");  
}  
  
public static void func() {  
    int[] arr = {1, 2, 3};  
    System.out.println(arr[100]);  
}
```

// 直接结果

java.lang.ArrayIndexOutOfBoundsException: 100

```
at demo02.Test.func(Test.java:18)
at demo02.Test.main(Test.java:9)
after try catch
```

**代码示例9** 如果向上一级传递都没有合适的方法处理异常, 最终就会交给 JVM 处理, 程序就会异常终止(和我们最开始未使用 try catch 时是一样的).

```
public static void main(String[] args) {
    func();
    System.out.println("after try catch");
}

public static void func() {
    int[] arr = {1, 2, 3};
    System.out.println(arr[100]);
}

// 执行结果
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 100
at demo02.Test.func(Test.java:14)
at demo02.Test.main(Test.java:8)
```

可以看到, 程序已经异常终止了, 没有执行到 `System.out.println("after try catch");` 这一行.

## 异常处理流程

- 程序先执行 try 中的代码
- 如果 try 中的代码出现异常, 就会结束 try 中的代码, 看和 catch 中的异常类型是否匹配.
- 如果找到匹配的异常类型, 就会执行 catch 中的代码
- 如果没有找到匹配的异常类型, 就会将异常向上传递到上层调用者.
- 无论是否找到匹配的异常类型, finally 中的代码都会被执行到(在该方法结束之前执行).
- 如果上层调用者也没有处理的了异常, 就继续向上传递.
- 一直到 main 方法也没有合适的代码处理异常, 就会交给 JVM 来进行处理, 此时程序就会异常终止.

## 抛出异常

除了 Java 内置的类会抛出一些异常之外, 程序员也可以手动抛出某个异常. 使用 throw 关键字完成这个操作.

```
public static void main(String[] args) {
    System.out.println(divide(10, 0));
}

public static int divide(int x, int y) {
    if (y == 0) {
        throw new ArithmeticException("抛出除 0 异常");
    }
    return x / y;
}
```



```
// 执行结果
```

```
Exception in thread "main" java.lang.ArithmeticException: 抛出除 0 异常
    at demo02.Test.divide(Test.java:14)
    at demo02.Test.main(Test.java:9)
```

在这个代码中, 我们可以根据实际情况来抛出需要的异常. 在构造异常对象同时可以指定一些描述性信息.

## 异常说明

我们在处理异常的时候, 通常希望知道这段代码中究竟会出现哪些可能的异常.

我们可以使用 throws 关键字, 把可能抛出的异常显式的标注在方法定义的位置. 从而提醒调用者要注意捕获这些异常.

```
public static int divide(int x, int y) throws ArithmeticException {
    if (y == 0) {
        throw new ArithmeticException("抛出除 0 异常");
    }
    return x / y;
}
```

## 关于 finally 的注意事项

finally 中的代码保证一定会执行到. 这也会带来一些麻烦.

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

public static int func() {
    try {
        return 10;
    } finally {
        return 20;
    }
}
```

```
// 执行结果
20
```

### 注意:

finally 执行的时机是在方法返回之前(try 或者 catch 中如果有 return 会在这个 return 之前执行 finally). 但是如果 finally 中也存在 return 语句, 那么就会执行 finally 中的 return, 从而不会执行到 try 中原有的 return.

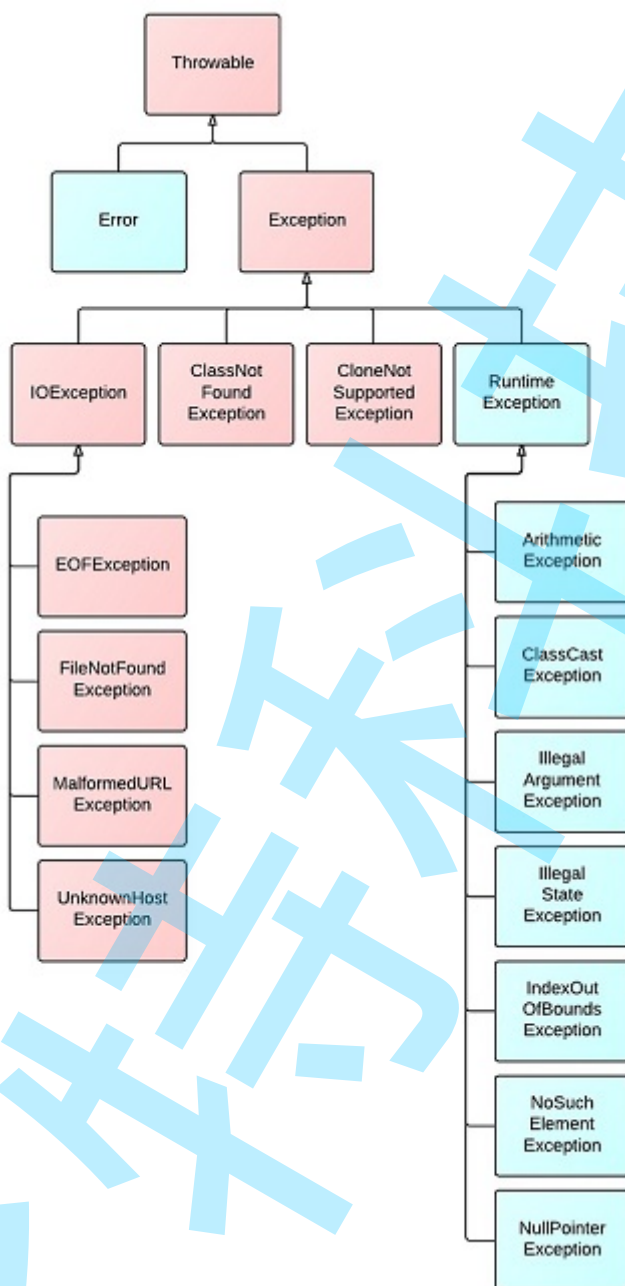
一般我们不建议在 finally 中写 return (被编译器当做一个警告).

```
'return' inside 'finally' block less... (Ctrl+F1)
Inspection info: Reports return statements inside of finally blocks. While occasionally intended, such return statements may mask exceptions thrown, and tremendously complicate debugging.
```

### 3. Java 异常体系

Java 内置了丰富的异常体系, 用来表示不同情况下的异常.

下图表示 Java 内置的异常类之间的继承关系:



- 顶层类 `Throwable` 派生出两个重要的子类, `Error` 和 `Exception`
- 其中 `Error` 指的是 Java 运行时内部错误和资源耗尽错误. **应用程序不抛出此类异常**. 这种内部错误一旦出现, 除了告知用户并使程序终止之外, 再无能为力. 这种情况很少出现.
- `Exception` 是我们程序猿所使用的异常类的父类.
- 其中 `Exception` 有一个子类称为 `RuntimeException`, 这里面又派生出很多我们常见的异常类 `NullPointerException`, `IndexOutOfBoundsException` 等.

Java语言规范将派生于 `Error` 类或 `RuntimeException` 类的所有异常称为 **非受查异常**, 所有的其他异常称为 **受查异常**.

如果一段代码可能抛出 **受查异常**, 那么必须显式进行处理.

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

public static String readFile() {
    // 尝试打开文件，并读其中的一行。
    File file = new File("d:/test.txt");
    // 使用文件对象构造 Scanner 对象。
    Scanner sc = new Scanner(file);
    return sc.nextLine();
}
```

// 编译出错

Error: (13, 22) java: 未报告的异常错误 java.io.FileNotFoundException; 必须对其进行捕获或声明以便抛出

查看 `Scanner` 的构造方法可以发现, 存在 `FileNotFoundException` 这样的异常说明.

```
public Scanner(File source) throws FileNotFoundException {
    ...
}
```

如 `FileNotFoundException` 这样的异常就是受查异常. 如果不显式处理, 编译无法通过.

显式处理的方式有两种:

#### a) 使用 try catch 包裹起来

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

public static String readFile() {
    File file = new File("d:/test.txt");
    Scanner sc = null;
    try {
        sc = new Scanner(file);
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
    return sc.nextLine();
}
```

#### b) 在方法上加上异常说明, 相当于将处理动作交给上级调用者

```
public static void main(String[] args) {
    try {
        System.out.println(readFile());
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
}

public static String readFile() throws FileNotFoundException {
    File file = new File("d:/test.txt");
    Scanner sc = new Scanner(file);
    return sc.nextLine();
}
```

别忘了 IDEA 神奇的 alt + enter, 能够快速修正代码.

## 4. 自定义异常类

Java 中虽然已经内置了丰富的异常类, 但是我们实际场景中可能还有一些情况需要我们对异常类进行扩展, 创建符合我们实际情况的异常.

例如, 我们实现一个用户登陆功能.

```
public class Test {
    private static String userName = "admin";
    private static String password = "123456";

    public static void main(String[] args) {
        login("admin", "123456");
    }

    public static void login(String userName, String password) {
        if (!Test.userName.equals(userName)) {
            // TODO 处理用户名错误
        }
        if (!Test.password.equals(password)) {
            // TODO 处理密码错误
        }
        System.out.println("登陆成功");
    }
}
```

此时我们在处理用户名密码错误的时候可能就需要抛出两种异常. 我们可以基于已有的异常类进行扩展 (继承), 创建和我们业务相关的异常类.

```
class UserError extends Exception {
    public UserError(String message) {
        super(message);
    }
}

class PasswordError extends Exception {
    public PasswordError(String message) {
        super(message);
    }
}
```

此时我们的 login 代码可以改成

```
public static void main(String[] args) {
    try {
        login("admin", "123456");
    } catch (UserError userError) {
        userError.printStackTrace();
    } catch (PasswordError passwordError) {
        passwordError.printStackTrace();
    }
}

public static void login(String userName, String password) throws UserError,
PasswordError {
    if (!Test.userName.equals(userName)) {
        throw new UserError("用户名错误");
    }
    if (!Test.password.equals(password)) {
        throw new PasswordError("密码错误");
    }
    System.out.println("登陆成功");
}
```

### 注意事项

- 自定义异常通常会继承自 `Exception` 或者 `RuntimeException`
- 继承自 `Exception` 的异常默认是受查异常
- 继承自 `RuntimeException` 的异常默认是非受查异常.

## 作业

编写课堂代码, 写博客总结 Java 异常知识点.