

01.快速了解异常

1.1 什么是异常

1.2 为何要有Java中的异常体系

1. 异常的由来与传统错误处理方式的局限性
2. 为什么需要异常处理机制？
3. Java的异常机制标准模式

1.3 异常思考

02.Java异常体系

2.1 继承体系介绍

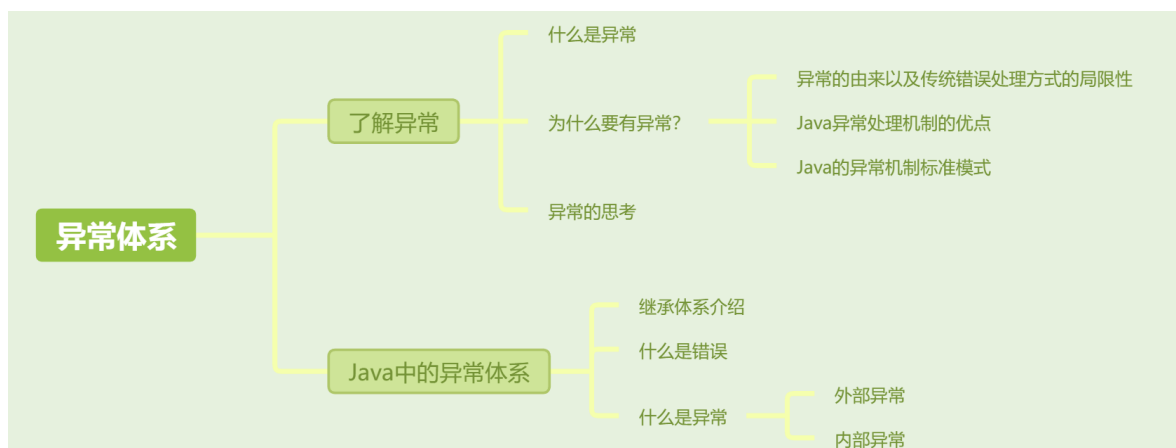
2.2 什么是Error(错误)

2.3 什么是Exception

2.4 编译时异常和运行时异常

其他介绍

01.关于我的博客



01.快速了解异常

1.1 什么是异常

异常 (Exception) 是指程序没有按照期望的正确结果执行，途中出现错误的情况。

例如，常见的异常情况包括：

- **算术运算异常**：例如，除数为零时会触发 `ArithmeticException`。
- **数组下标越界**：如果访问数组时使用了非法的索引（超出数组的范围），会抛出 `ArrayIndexOutOfBoundsException`。
- **空指针异常**：试图调用 `null` 对象的方法或访问它的成员变量时，会抛出 `NullPointerException`。

1.2 为何要有Java中的异常体系

1. 异常的由来与传统错误处理方式的局限性

在早期编程语言（如 C 语言）中，错误处理通常依赖于约定俗成的方法，例如：

- 函数返回特殊的错误码，例如 `-1` 或 `NULL`，以表明出现了错误。
- 使用全局的错误号：`errno`，来指示错误的具体类型。比如：内核中`errno`是多少就表示对应的什么错误。

这种方式的问题在于：以打开文件，读取数据为例。

- **每一步都要检查错误**：程序员需要在代码的每一步操作后检查是否出现错误，导致逻辑繁琐且难以维护。
- **错误检查容易遗漏**：如果某个地方忘记检查返回值或错误码，可能会导致程序在出现错误时继续运行，最终导致难以预料的行为。
- **不利于大型系统**：当系统规模变大时，随着函数调用的嵌套和模块的增多，使用返回值层次往上手动传递，传递过程中是否会出现失真？

举例：

```
FILE *file = fopen("example.txt", "r");
if (file == NULL) {
    printf("打开文件失败\n");
    return -1;
}

char buffer[256];
if (fgets(buffer, sizeof(buffer), file) == NULL) {
    printf("读取文件失败\n");
    fclose(file);
    return -1;
}

if (fclose(file) != 0) {
    printf("关闭文件失败\n");
    return -1;
}

// 业务逻辑：处理读取的内容
```

2. 为什么需要异常处理机制？

异常处理机制的引入是为了解决传统错误处理方式的局限性，尤其是在构建大型、健壮的系统时，它具有以下优势：

1. **分离错误处理与正常逻辑**：异常处理机制将程序的正常执行路径与错误处理路径分离开来，把“出现问题时怎么办”的处理逻辑集中放在一个地方，减少了大量的错误检查代码，避免每一步都要检查错误，也减少了错误检查的遗漏。
2. **自动传递与传播**：当某个代码块发生异常时，它可以自动被传递给调用它的上层代码，直到有一个地方对它进行处理。这种传播机制避免了需要手动返回错误码层层传递的繁琐过程。

怎么理解：分离错误处理与正常逻辑

"分离错误处理与正常逻辑" 是异常机制的一个核心概念，指的是在程序中将正常的业务逻辑（程序应该执行的主要功能）与处理错误的逻辑分开，从而使代码更加清晰、易读、易维护。以下是对这个概念的详细解释和示例。

1. 传统错误处理方法 vs 异常机制

在没有异常机制的语言（如 C 语言）中，程序员通常需要在每一步操作后进行错误检查。例如，打开文件、处理输入、执行运算等，都可能出错。这种方式会导致业务逻辑与错误处理混在一起，使得代码显得冗杂和复杂。

示例：

```
FILE *file = fopen("example.txt", "r");
```

```

if (file == NULL) {
    printf("打开文件失败\n");
    return -1;
}

char buffer[256];
if (fgets(buffer, sizeof(buffer), file) == NULL) {
    printf("读取文件失败\n");
    fclose(file);
    return -1;
}

if (fclose(file) != 0) {
    printf("关闭文件失败\n");
    return -1;
}

// 业务逻辑：处理读取的内容

```

在这个例子中，你看到每一个操作（打开文件、读取文件、关闭文件）之后都有错误检查代码。这样做的结果是，业务逻辑和错误处理逻辑交织在一起，代码显得冗长、复杂、不易读，而且容易出错。

2. 异常机制中的分离

程序员不需要在每一步操作之后都进行错误检查，只需在一个集中的地方捕获和处理异常。

```

try {
    FileReader reader = new FileReader("example.txt");//打开文件
    BufferedReader bufferedReader = new BufferedReader(reader);//将读取的字符先存入
    一个内部缓冲区

    String line = bufferedReader.readLine(); // 读取缓冲区中文本

    bufferedReader.close();//关闭 BufferedReader 以及它包装的底层 FileReader。
} catch (IOException e) {
    System.out.println("文件处理出错: " + e.getMessage()); // 错误处理逻辑集中在这里
}

```

怎么理解：自动传播与分离？

1. 传统的手动传播

```

#include <stdio.h>

int functionB() {
    printf("Inside functionB\n");
    // 返回错误码 -1 表示错误
    return -1;
}

int functionA() {
    printf("Inside functionA\n");
    // 调用 functionB，手动检查错误码并传递错误
    int result = functionB();
    if (result == -1) {
        return -1; // 向上传递错误码
    }
}

```

```

        return 0;
    }

    int main() {
        printf("Inside main\n");
        // 调用 functionA, 手动检查错误码
        int result = functionA();
        if (result == -1) {
            printf("Error occurred in functionB\n");
        }
        printf("Program continues after error handling.\n");
        return 0;
    }

```

2. Java中的自动传播

```

public class ExceptionPropagationExample {

    // functionB 抛出异常
    public static void functionB() throws Exception {
        System.out.println("Inside functionB");
        // 人为抛出异常
        throw new Exception("An error occurred in functionB");
    }

    // functionA 调用 functionB, 不处理异常, 继续传播
    public static void functionA() throws Exception {
        System.out.println("Inside functionA");
        functionB(); // 调用 functionB, 异常将向上传递
    }

    // main 函数, 最终处理异常
    public static void main(String[] args) {
        try {
            System.out.println("Inside main");
            functionA(); // 调用 functionA
        } catch (Exception e) {
            // 捕获异常并处理
            System.out.println("Exception caught in main: " +
                e.getMessage());
        }
        System.out.println("Program continues after exception handling.");
    }
}

```

3. Java的异常机制标准模式

Java中常见的异常处理结构包括 `try-catch-finally` 语句, 以及可以使用 `throw` 关键字手动抛出异常。

示例:

```
try {
    int result = 10 / 0; // 可能会抛出 ArithmeticException
} catch (ArithmeticException e) {
    System.out.println("发生了算术异常: " + e.getMessage());
} finally {
    System.out.println("无论异常是否发生, 都会执行finally块");
}
```

在这个例子中, `try` 块中是我们的工作代码执行流, 它可能会发生异常。 `catch` 块是异常的捕捉和处理流, 当 `try` 中出现异常时会执行 `catch` 中代码, 而 `finally` 块则始终会执行, 确保资源被正确释放或其他必要的操作得以执行, 只有在程序因调用 `System.exit()` 终止时, `finally` 块不会执行。即使 `try` 或 `catch` 中有 `return` 语句, `finally` 块也会在返回值传递之前执行。如果 `finally` 中包含 `return` 语句, 它将覆盖 `try` 或 `catch` 中的 `return`。

1.3 异常思考

什么是一个有效的异常?

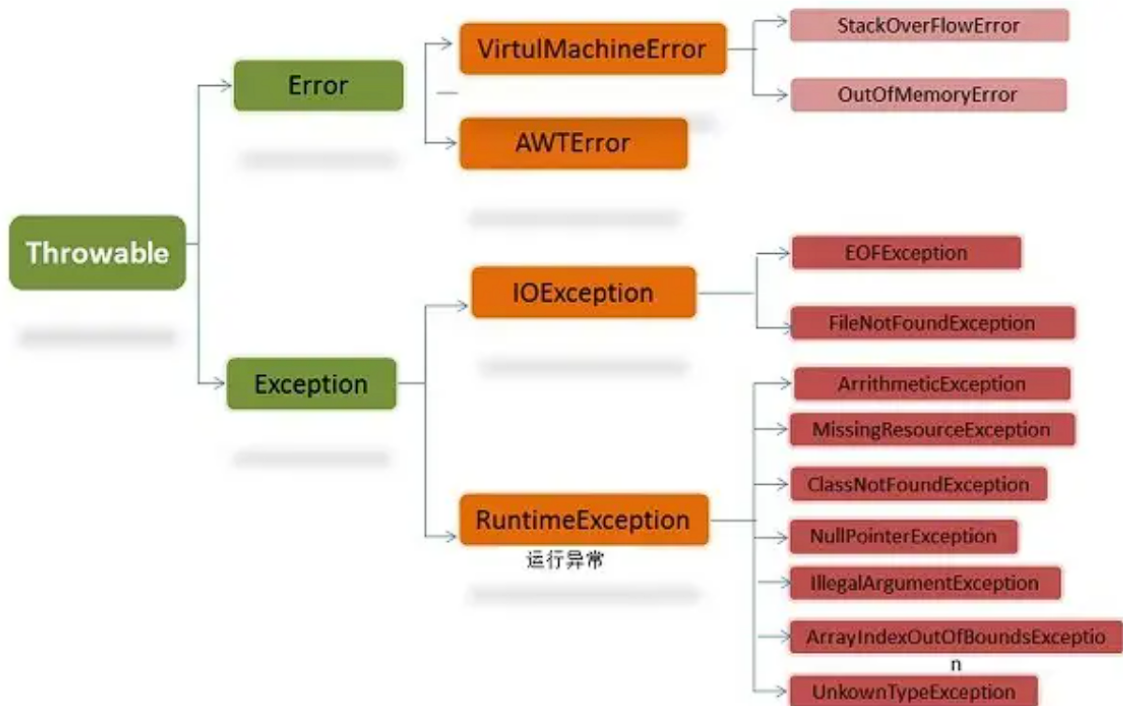
- 在有效使用异常的情况下, 异常能清晰的回答 what, where, why 这3个问题:
 - 异常类型, 会告诉“什么”被抛出
 - 异常堆栈跟踪, 会告诉“在哪”抛出
 - 异常信息, 会告诉“为什么”会抛出。

举例: 看一个异常信息

- 异常类型, 是 `NullPointerException`, 空指针异常
- 异常堆栈跟踪, `ZygoteInit.main ----> RuntimeInit ----> ActivityThread.main ----> Handler.dispatchMessage ----> View.performClick ----> CrashTestActivity.onClick`
- 异常信息, `Attempt to invoke virtual method 'void android.app.Activity.finish()' on a null object reference`

我们可以推算出: 代码试图在一个 `null` 对象上调用 `finish()` 方法, 而 `finish()` 是 `Activity` 类中的方法。可以推断出, `onClick` 方法中的某个 `Activity` 对象为 `null`

02.Java异常体系



2.1 继承体系介绍

父类Object，子类Throwable，往后是Error错误和Exception异常。错误有VirtualMachineError和另一个。异常分为内部异常和外部异常，也就是IOException和RuntimeException。

2.2 什么是Error(错误)

- Error（错误）：很严重的，程序无法处理的错误，大多数错误都不是由程序本身产生的，而表示代码运行时 JVM（Java 虚拟机）出现的问题。
- 例如，Java虚拟机运行错误（Virtual MachineError），当JVM不再有继续执行操作所需的内存资源时，将出现 OutOfMemoryError。这不是你代码编写的错误，你只是正常的申请虚拟机内存资源，但是由于虚拟机内存资源本身不足造成的错误，不是你的程序可以处理的。

2.3 什么是Exception

- Exception（异常）：是程序本身可以处理的异常。
- Exception 这种异常分两大类运行时异常和非运行时异常(编译异常)。程序中应当尽可能去处理这些异常。

2.4 编译时异常和运行时异常

我理解他们为内部异常和外部异常，什么是内部，什么是外部？程序的本质是对数据进行处理。处理数据是内部，数据从外存，网络输入输出叫外部。外部异常是必定会出现的异常，内部异常时可以通过优秀逻辑避免的异常。

外部异常：数据从外部输入时总是会产生错误的，这不是我们程序内部逻辑的问题，因为我们没办法控制外部，与外界的交互总是会面临链接不稳定，或者没法成功通信的问题，因此我们要提前在编译时就必须编写异常处理代码，这些是必定会出现异常的，只是概率问题。如果不编写异常处理，程序将无法编译通过。

内部异常是与程序的内部逻辑有关，比如空指针引用、数组下标越界等。这些问题是可以通过优秀的逻辑避免的。因此运行时异常并不强制要求程序员在编译时处理，它们可以在运行时被抛出。

其他介绍

01.关于我的博客

- github: <https://github.com/jjjjjjava>
- 简书: <http://www.jianshu.com/u/92a2412be53e>
- csdn: http://my.csdn.net/qg_35829566
- 邮箱: 934137388@qq.com
- 掘金: <https://juejin.im/user/499639464759898>