- Java异常架构与异常关键字
  - Java异常简介
  - Java异常架构
    - 1. Throwable
    - 2. Error（错误）
    - 3. Exception（异常）
      - 运行时异常
      - 编译时异常
    - 4. 受检异常与非受检异常
      - 受检异常
      - 非受检异常
  - Java异常关键字
- Java异常处理
  - 声明异常
  - 抛出异常
  - 捕获异常
  - 如何选择异常类型
  - 常见异常处理方式
    - 直接抛出异常
    - 封装异常再抛出

Java异常架构与异常关键字

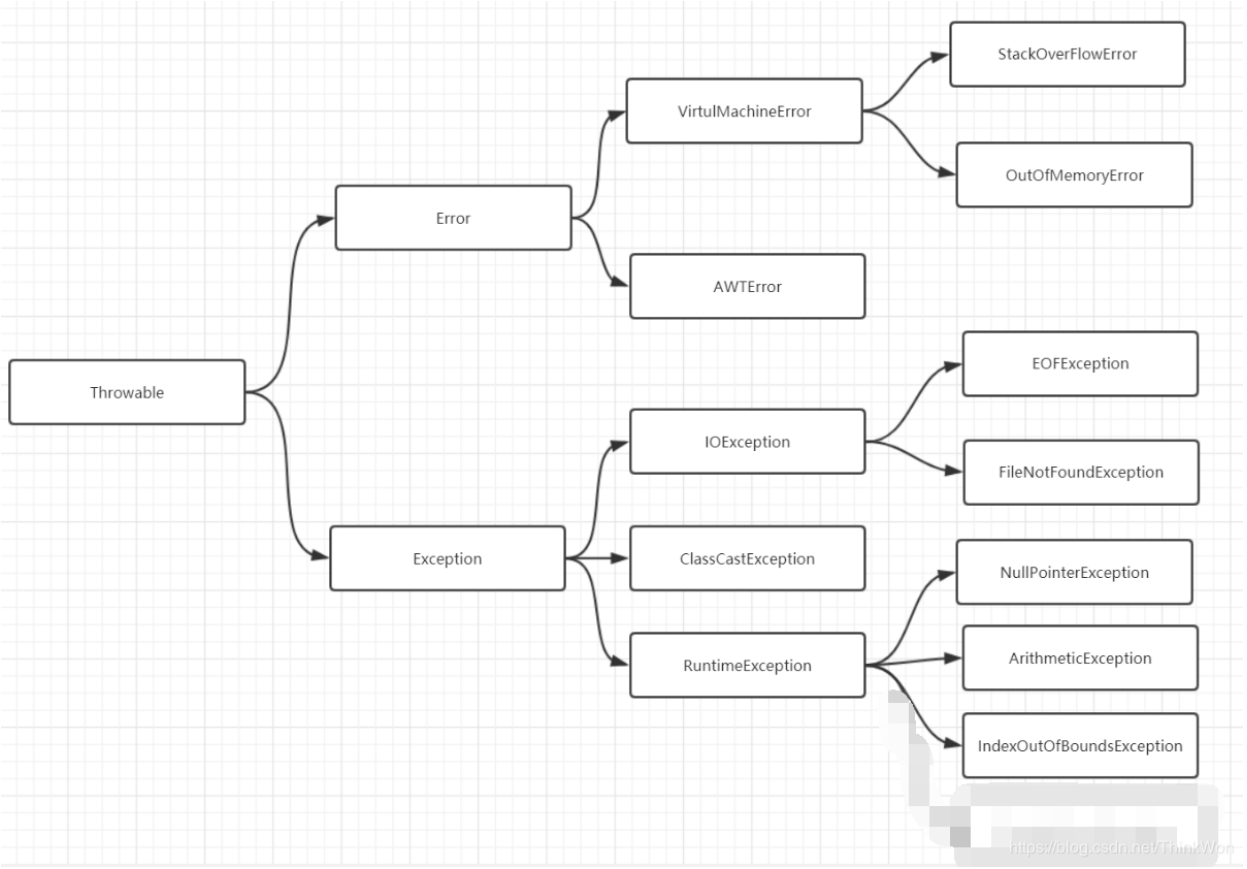
Java异常简介

Java异常是Java提供的一种识别及响应错误的一致性机制。

Java异常机制可以使程序中异常处理代码和正常业务代码分离，保证程序代码更加优雅，并提高程序健壮性。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：异常类型回答了“什么”被抛出，异常堆栈跟踪回答了“在哪”抛出，异常信息回答了“为什么”会抛出。

Java异常架构



1. Throwable

Throwable 是 Java 语言中所有错误与异常的超类。

Throwable 包含两个子类：Error（错误）和 Exception（异常），它们通常用于指示发生了异常情况。

Throwable 包含了其线程创建时线程执行堆栈的快照，它提供了 printStackTrace() 等接口用于获取堆栈跟踪数据等信息。

2. Error（错误）

定义：Error 类及其子类。程序中无法处理的错误，表示运行应用程序中出现了严重的错误。

特点：此类错误一般表示代码运行时 JVM 出现问题。通常有 Virtual MachineError（虚拟机运行错误）、NoClassDefFoundError（类定义错误）等。比如 OutOfMemoryError：内存不足错误；StackOverflowError：栈溢出错误。此类错误发生时，JVM 将终止线程。

这些错误是不受检异常，非代码性错误。因此，当此类错误发生时，应用程序不应该去处理此类错误。按照Java惯例，我们是不应该实现任何新的Error子类的！

3. Exception（异常）

程序本身可以捕获并且可以处理的异常。Exception 这种异常又分为两类：运行时异常和编译时异常。

运行时异常

定义：RuntimeException 类及其子类，表示 JVM 在运行期间可能出现的异常。

特点：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，倘若既"没有通过throws声明抛出它"，也"没有用try-catch语句捕获它"，还是会编译通过。比如NullPointerException空指针异常、ArrayIndexOutBoundException数组下标越界异常、ClassCastException类型转换异常、ArithmeticExecption算术异常。此类异常属于不受检异常，一般是由程序逻辑错误引起的，在程序中可以选择捕获处理，也可以不处理。虽然 Java 编译器不会检查运行时异常，但是我们也可以通过 throws 进行声明抛出，也可以通过 try-catch 对它进行捕获处理。如果产生运行时异常，则需要通过修改代码来进行避免。例如，若会发生除数为零的情况，则需要通过代码避免该情况的发生！

RuntimeException 异常会由 Java 虚拟机自动抛出并自动捕获（**就算我们没写异常捕获语句运行时也会抛出错误**！！），此类异常的出现绝大数情况是代码本身有问题应该从逻辑上去解决并改进代码。

编译时异常

定义: Exception 中除 RuntimeException 及其子类之外的异常。

特点: Java 编译器会检查它。如果程序中出现此类异常，比如ClassNotFoundException（没有找到指定的类异常），IOException（IO流异常），要么通过throws进行声明抛出，要么通过try-catch进行捕获处理，否则不能通过编译。在程序中，通常不会自定义该类异常，而是直接使用系统提供的异常类。**该异常我们必须手动在代码里添加捕获语句来处理该异常**。

4. 受检异常与非受检异常

Java 的所有异常可以分为受检异常（checked exception）和非受检异常（unchecked exception）。

受检异常

编译器要求必须处理的异常。正确的程序在运行过程中，经常容易出现的、符合预期的异常情况。一旦发生此类异常，就必须采用某种方式进行处理。**除 RuntimeException 及其子类外，其他的 Exception 异常都属于受检异常**。编译器会检查此类异常，也就是说当编译器检查到应用中的某处可能会此类异常时，将会提示你处理本异常——要么使用try-catch捕获，要么使用方法签名中用 throws 关键字抛出，否则编译不通过。

非受检异常

编译器不会进行检查并且不要求必须处理的异常，也就说当程序中出现此类异常时，即使我们没有try-catch捕获它，也没有使用throws抛出该异常，编译也会正常通过。**该类异常包括运行时异常（RuntimeException极其子类）和错误（Error）。**

Java异常关键字

• **try** – 用于监听。将要被监听的代码(可能抛出异常的代码)放在try语句块之内，当try语句块内发生异常时，异常就被抛出。

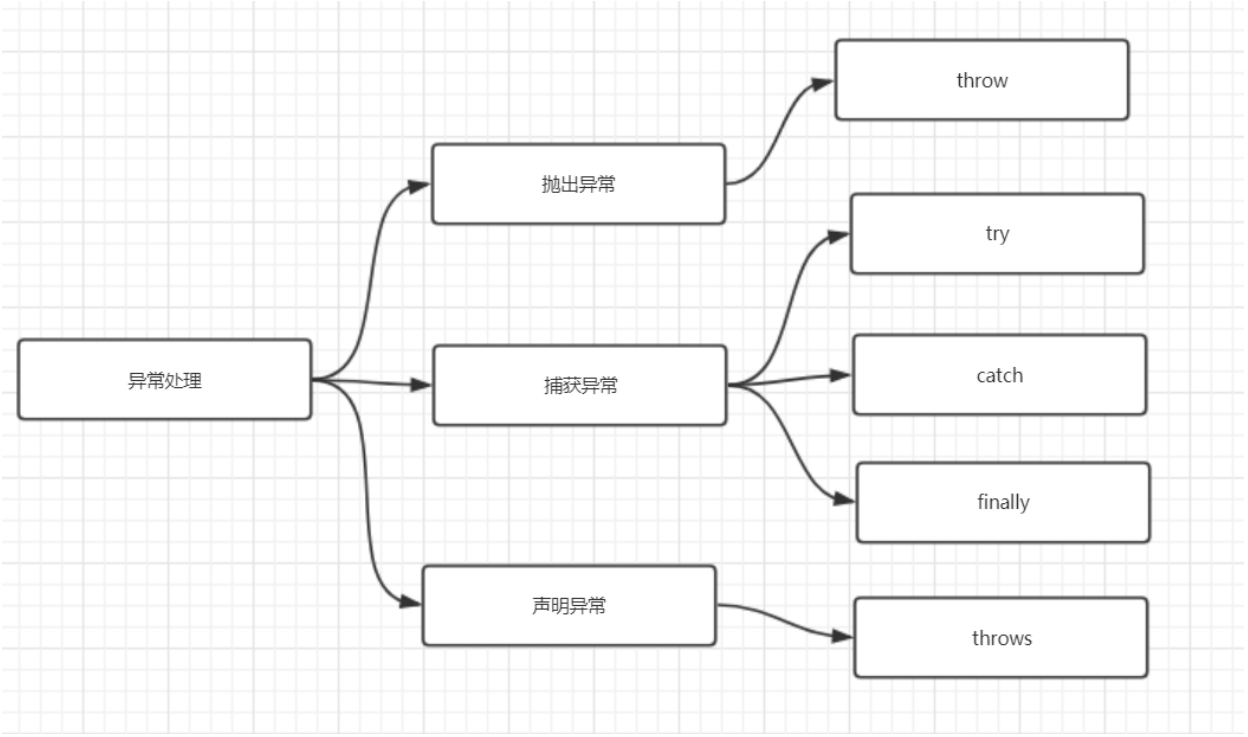
• **catch** – 用于捕获异常。catch用来捕获try语句块中发生的异常。

• **finally** – finally语句块总是会被执行。它主要用于回收在try块里打开的物力资源(如数据库连接、网络连接和磁盘文件)。只有finally块，执行完成之后，才会回来执行try或者catch块中的return或者throw语句，如果finally中使用了return或者throw等终止方法的语句，则就不会跳回执行，直接停止。

• **throw** – 用于抛出异常。

• **throws** – 用在方法签名中，用于声明该方法可能抛出的异常。

Java异常处理



Java 通过面向对象的方法进行异常处理，一旦方法抛出异常，系统自动根据该异常对象寻找合适异常处理器（Exception Handler）来处理该异常，把各种不同的异常进行分类，并提供了良好的接口。在 Java 中，每个异常都是一个对象，它是 Throwable 类或其子类的实例。当一个方法出现异常后便抛出一个异常对象，该对象中包含有异常信息，调用这个对象的方法可以捕获到这个异常并可以对其进行处理。Java 的异常处理是通过 5 个关键词来实现的：try、catch、throw、throws 和 finally。

在Java应用中，异常的处理机制分为声明异常，抛出异常和捕获异常。

声明异常

通常，应该捕获那些知道如何处理的异常，将不知道如何处理的异常继续传递下去。传递异常可以在方法签名处使用 **throws** 关键字声明可能会抛出的异常。

注意

- 非检查异常（Error、RuntimeException 或它们的子类）不可使用 throws 关键字来声明要抛出的异常。
- 一个方法出现编译时异常，就需要 try-catch/ throws 处理，否则会导致编译错误。

抛出异常

如果你觉得解决不了某些异常问题，且不需要调用者处理，那么你可以抛出异常。

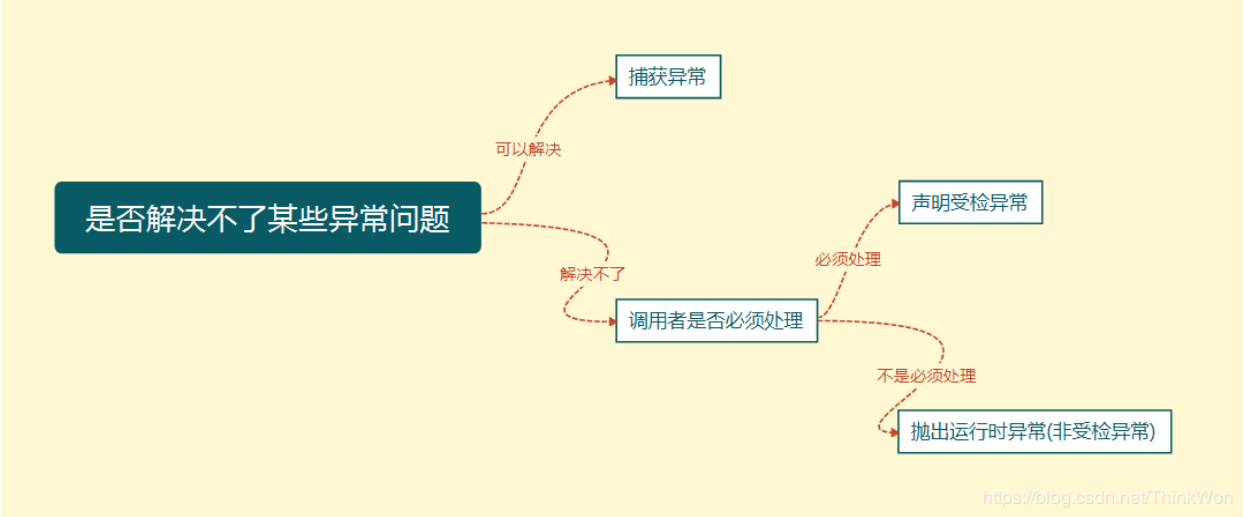
throw关键字作用是在方法内部抛出一个Throwable类型的异常。任何Java代码都可以通过throw语句抛出异常。

捕获异常

程序通常在运行之前不报错，但是运行后可能会出现某些未知的错误，但是还不想直接抛出到上一级，那么就需要通过try...catch...的形式进行异常捕获，之后根据不同的异常情况来进行相应的处理。

如何选择异常类型

可以根据下图来选择是捕获异常，声明异常还是抛出异常



在这里插入图片描述

常见异常处理方式

直接抛出异常

通常，应该捕获那些知道如何处理的异常，将不知道如何处理的异常继续传递下去。传递异常可以在方法签名处使用 **throws** 关键字声明可能会抛出的异常。

```
1 private static void readFile(String filePath) throws IOException {
2  File file = new File(filePath);
3  String result;
4  BufferedReader reader = new BufferedReader(new FileReader(file));
5  while((result = reader.readLine())!=null) {
6  System.out.println(result);
7  }
8  reader.close();
9 }
```

封装异常再抛出

有时我们会从 catch 中抛出一个异常，目的是为了改变异常的类型。多用于在多系统集成时，当某个子系统故障，异常类型可能有多种，可以用统一的异常类型向外暴露，不需暴露太多内部异常细节。

```
private static void readFile(String filePath) throws MyException {
 try {
 // code
 } catch (IOException e) {
 MyException ex = new MyException("read file failed.");
 ex.initCause(e);
 throw ex;
 }
}
```

捕获异常

在一个 try-catch 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

```
private static void readFile(String filePath) {
 try {
 // code
 } catch (FileNotFoundException e) {
 // handle FileNotFoundException
 } catch (IOException e){
 // handle IOException
 }
}
```

同一个 catch 也可以捕获多种类型异常，用 | 隔开

```
private static void readFile(String filePath) {
 try {
 // code
 } catch (FileNotFoundException | UnknownHostException e) {
 // handle FileNotFoundException or UnknownHostException
 } catch (IOException e){
 // handle IOException
 }
}
```

自定义异常

习惯上，定义一个异常类应包含两个构造函数，一个无参构造函数和一个带有详细描述信息的构造函数（Throwable 的 toString 方法会打印这些详细信息，调试时很有用）

```
public class MyException extends Exception {
 public MyException(){ }
 public MyException(String msg){
 super(msg);
 }
 // ...
}
```

try-catch-finally

当方法中发生异常，异常处之后的代码不会再执行，如果之前获取了一些本地资源需要释放，则需要在方法正常结束时和 catch 语句中都调用释放本地资源的代码，显得代码比较繁琐，finally 语句可以解决这个问题。

```
private static void readFile(String filePath) throws MyException {
 File file = new File(filePath);
 String result;
 BufferedReader reader = null;
 try {
 reader = new BufferedReader(new FileReader(file));
 while((result = reader.readLine())!=null) {
 System.out.println(result);
 }
  } catch (IOException e) {
  System.out.println("readFile method catch block.");
  MyException ex = new MyException("read file failed.");
  ex.initCause(e);
  throw ex;
  } finally {
  System.out.println("readFile method finally block.");
  if (null != reader) {
  try {
  reader.close();
  } catch (IOException e) {
  e.printStackTrace();
  }
  }
```

```
24    }
25  }
```

调用该方法时，读取文件时若发生异常，代码会进入 catch 代码块，之后进入 finally 代码块；若读取文件时未发生异常，则会跳过 catch 代码块直接进入 finally 代码块。所以无论代码中是否发生异常，fianlly 中的代码都会执行。

若 catch 代码块中包含 return 语句，finally 中的代码还会执行吗？将以上代码中的 catch 子句修改如下：

```
1  catch (IOException e) {
2   System.out.println("readFile method catch block.");
3   return;
4  }
```

调用 readFile 方法，观察当 catch 子句中调用 return 语句时，finally 子句是否执行

```
1  readFile method catch block.
2  readFile method finally block.
```

可见，即使 catch 中包含了 return 语句，finally 子句依然会执行。若 finally 中也包含 return 语句，finally 中的 return 会覆盖前面的 return.

try-with-resource

上面例子中，finally 中的 close 方法也可能抛出 IOException, 从而覆盖了原始异常。JAVA 7 提供了更优雅的方式来实现资源的自动释放，自动释放的资源需要是实现了 AutoCloseable 接口的类。

```
1  private static void tryWithResourceTest(){
2   try (Scanner scanner = new Scanner(new FileInputStream("c:/abc"),"UTF-
8")){
3   // code
4   } catch (IOException e){
5   // handle exception
6   }
7  }
```

try 代码块退出时，会自动调用 scanner.close 方法，和把 scanner.close 方法放在 finally 代码块中不同的是，若 scanner.close 抛出异常，则会被抑制，抛出的仍然为原始异常。被抑制的异常会由 addSusppressed 方法添加到原来的异常，如果想要获取被抑制的异常列表，可以调用 getSuppressed 方法来获取。

Java异常常见面试题

1. Error 和 Exception 区别是什么？

Error 类型的错误通常为虚拟机相关错误，如系统崩溃，内存不足，堆栈溢出等，编译器不会对这类错误进行检测，JAVA 应用程序也不应对这类错误进行捕获，一旦这类错误发生，通常应用程序会被终止，仅靠应用程序本身无法恢复；Exception 类的错误是可以在应用程序中进行捕获并处理的，通常遇到这种错误，应对其进行处理，使应用程序可以继续正常运行。

2. 运行时异常和一般异常(受检异常)区别是什么？

运行时异常包括 RuntimeException 类及其子类，表示 JVM 在运行期间可能出现的异常。 Java 编译器不会检查运行时异常。

受检异常是Exception 中除 RuntimeException 及其子类之外的异常。 Java 编译器会检查受检异常。

RuntimeException异常和受检异常之间的区别：是否强制要求调用者必须处理此异常，如果强制要求调用者必须进行处理，那么就使用受检异常，否则就选择非受检异常(RuntimeException)。一般来讲，如果没有特殊的要求，我们建议使用RuntimeException异常。

3. JVM 是如何处理异常的？

在一个方法中如果发生异常，这个方法会创建一个异常对象，并转交给 JVM，该异常对象包含异常名称，异常描述以及异常发生时应用程序的状态。创建异常对象并转交给 JVM 的过程称为抛出异常。可能有一系列的方法调用，最终才进入抛出异常的方法，这一系列方法调用的有序列表叫做调用栈。

JVM 会顺着调用栈去查找看是否有可以处理异常的代码，如果有，则调用异常处理代码。当 JVM 发现可以处理异常的代码时，会把发生的异常传递给它。如果 JVM 没有找到可以处理该异常的代码块，JVM 就会将该异常转交给默认的异常处理器（默认处理器为 JVM 的一部分），默认异常处理器打印出异常信息并终止应用程序。

4. throw 和 throws 的区别是什么？

Java 中的异常处理除了包括捕获异常和处理异常之外，还包括声明异常和抛出异常，可以通过 throws 关键字在方法上声明该方法要抛出的异常，或者在方法内部通过 throw 抛出异常对象。

throws 关键字和 throw 关键字在使用上的几点区别如下：

- throw 关键字用在方法内部，只能用于抛出一种异常，用来抛出方法或代码块中的异常，受查异常和非受查异常都可以被抛出。
- throws 关键字用在方法声明上，可以抛出多个异常，用来标识该方法可能抛出的异常列表。一个方法用 throws 标识了可能抛出的异常列表，调用该方法的方法中必须包含可处理异常的代码，否则也要在方法签名中用 throws 关键字声明相应的异常。

5. final、finally、finalize 有什么区别？

- final可以修饰类、变量、方法，修饰类表示该类不能被继承、修饰方法表示该方法不能被重写、修饰变量表示该变量是一个常量不能被重新赋值。
- finally一般作用在try-catch代码块中，在处理异常的时候，通常我们将一定要执行的代码方法finally代码块中，表示不管是否出现异常，该代码块都会执行，一般用来存放一些关闭资源的代码。
- finalize是一个方法，属于Object类的一个方法，而Object类是所有类的父类，Java 中允许使用 finalize()方法在垃圾收集器将对象从内存中清除出去之前做必要的清理工作。

6. NoClassDefFoundError 和 ClassNotFoundException 区别？

NoClassDefFoundError 是一个 Error 类型的异常，是由 JVM 引起的，不应该尝试捕获这个异常。

引起该异常的原因是 JVM 或 ClassLoader 尝试加载某类时在内存中找不到该类的定义，该动作发生在运行期间，即编译时该类存在，但是在运行时却找不到了，可能是变异后被删除了等原因导致；

ClassNotFoundException 是一个受查异常，需要显式地使用 try-catch 对其进行捕获和处理，或在方法签名中用 throws 关键字进行声明。当使用 Class.forName, ClassLoader.loadClass 或 ClassLoader.findSystemClass 动态加载类到内存的时候，通过传入的类路径参数没有找到该类，就会抛出该异

常；另一种抛出该异常的可能原因是某个类已经由一个类加载器加载至内存中，另一个加载器又尝试去加载它。

7. try-catch-finally 中哪个部分可以省略？

答：catch 可以省略

原因

更为严格的说法其实是：try只适合处理运行时异常，try+catch适合处理运行时异常+普通异常。也就是说，如果你只用try去处理普通异常却不加以catch处理，编译是通不过的，因为编译器硬性规定，普通异常如果选择捕获，则必须用catch显示声明以便进一步处理。而运行时异常在编译时没有如此规定，所以catch可以省略，你加上catch编译器也觉得无可厚非。

理论上，编译器看任何代码都不顺眼，都觉得可能有潜在的问题，所以你即使对所有代码加上try，代码在运行期时也只不过是在正常运行的基础上加一层皮。但是你一旦对一段代码加上try，就等于显示地承诺编译器，对这段代码可能抛出的异常进行捕获而非向上抛出处理。如果是普通异常，编译器要求必须用catch捕获以便进一步处理；如果运行时异常，捕获然后丢弃并且+finally扫尾处理，或者加上catch捕获以便进一步处理。

至于加上finally，则是在不管有没捕获异常，都要进行的“扫尾”处理。

8. try-catch-finally 中，如果 catch 中 return 了，finally 还会执行吗？

答：会执行，在 return 前执行。

注意：在 finally 中改变返回值的做法是不好的，因为如果存在 finally 代码块，try中的 return 语句不会立马返回调用者，而是记录下返回值待 finally 代码块执行完毕之后再向调用者返回其值，然后如果在 finally 中修改了返回值，就会返回修改后的值。显然，在 finally 中返回或者修改返回值会对程序造成很大的困扰，C#中直接用编译错误的方式来阻止程序员干这种龌龊的事情，Java 中也可以通过提升编译器的语法检查级别来产生警告或错误。

代码示例1：

```
1 public static int getInt() {
2  int a = 10;
3  try {
```

```
System.out.println(a / 0);
a = 20;
} catch (ArithmeticException e) {
a = 30;
return a;
/*
 * return a 在程序执行到这一步的时候，这里不是return a 而是 return 30；这个返回路径就形成了
 * 但是呢，它发现后面还有finally，所以继续执行finally的内容，a=40
 * 再次回到以前的路径,继续走return 30，形成返回路径之后，这里的a就不是a变量了，而是常量30
 */
 } finally {
 a = 40;
 }
 return a;
}
```

执行结果：30

代码示例2：

```
public static int getInt() {
 int a = 10;
 try {
 System.out.println(a / 0);
 a = 20;
 } catch (ArithmeticException e) {
 a = 30;
 return a;
 } finally {
 a = 40;
 //如果这样，就又重新形成了一条返回路径，由于只能通过1个return返回，所以这里直接返回40
 return a;
 }

}
```

执行结果：40

9. 类 ExampleA 继承 Exception，类 ExampleB 继承 ExampleA。

有如下代码片断：

```
try {
 throw new ExampleB("b")
} catch（ExampleA e）{
 System.out.println("ExampleA");
} catch（Exception e）{
 System.out.println("Exception");
}
```

请问执行此段代码的输出是什么？

答：

输出：ExampleA。（根据里氏代换原则[能使用父类型的地方一定能使用子类型]，抓取 ExampleA 类型异常的 catch 块能够抓住 try 块中抛出的 ExampleB 类型的异常）

面试题 - 说出下面代码的运行结果。（此题的出处是《Java 编程思想》一书）

```
class Annoyance extends Exception {
}
class Sneeze extends Annoyance {
}
class Human {
 public static void main(String[] args)
 throws Exception {
 try {
 try {
  throw new Sneeze();
  } catch ( Annoyance a ) {
  System.out.println("Caught Annoyance");
  throw a;
  }
  } catch ( Sneeze s ) {
  System.out.println("Caught Sneeze");
  return ;
  } finally {
  System.out.println("Hello World!");
  }
  }
}
```

结果

```
Caught Annoyance
Caught Sneeze
Hello World!
```

10. 常见的 RuntimeException 有哪些？

- ClassCastException(类转换异常)
- IndexOutOfBoundsException(数组越界)
- NullPointerException(空指针)
- ArrayStoreException(数据存储异常，操作数组时类型不一致)
- 还有IO操作的BufferOverflowException异常

11. Java常见异常有哪些

java.lang.IllegalAccessError：违法访问错误。当一个应用试图访问、修改某个类的域（Field）或者调用其方法，但是又违反域或方法的可见性声明，则抛出该异常。

java.lang.InstantiationError：实例化错误。当一个应用试图通过Java的new操作符构造一个抽象类或者接口时抛出该异常.

java.lang.OutOfMemoryError：内存不足错误。当可用内存不足以让Java虚拟机分配给一个对象时抛出该错误。

java.lang.StackOverflowError：堆栈溢出错误。当一个应用递归调用的层次太深而导致堆栈溢出或者陷入死循环时抛出该错误。

java.lang.ClassCastException：类造型异常。假设有类A和B（A不是B的父类或子类），O是A的实例，那么当强制将O构造为类B的实例时抛出该异常。该异常经常被称为强制类型转换异常。

java.lang.ClassNotFoundException：找不到类异常。当应用试图根据字符串形式的类名构造类，而在遍历CLASSPAH之后找不到对应名称的class文件时，抛出该异常。

java.lang.ArithmeticException：算术条件异常。譬如：整数除零等。

java.lang.ArrayIndexOutOfBoundsException：数组索引越界异常。当对数组的索引值为负数或大于等于数组大小时抛出。

java.lang.IndexOutOfBoundsException：索引越界异常。当访问某个序列的索引值小于0或大于等于序列大小时，抛出该异常。

java.lang.InstantiationException：实例化异常。当试图通过newInstance()方法创建某个类的实例，而该类是一个抽象类或接口时，抛出该异常。
java.lang.NoSuchFieldException：属性不存在异常。当访问某个类的不存在的属性时抛出该异常。
java.lang.NoSuchMethodException：方法不存在异常。当访问某个类的不存在的方法时抛出该异常。
java.lang.NullPointerException：空指针异常。当应用试图在要求使用对象的地方使用了null时，抛出该异常。譬如：调用null对象的实例方法、访问null对象的属性、计算null对象的长度、使用throw语句抛出null等等。
java.lang.NumberFormatException：数字格式异常。当试图将一个String转换为指定的数字类型，而该字符串确不满足数字类型要求的格式时，抛出该异常。
java.lang.StringIndexOutOfBoundsException：字符串索引越界异常。当使用索引值访问某个字符串中的字符，而该索引值小于0或大于等于序列大小时，抛出该异常。

Java异常处理最佳实践

在 Java 中处理异常并不是一个简单的事情。不仅仅初学者很难理解，即使一些有经验的开发者也需要花费很多时间来思考如何处理异常，包括需要处理哪些异常，怎样处理等等。这也是绝大多数开发团队都会制定一些规则来规范进行异常处理的原因。而团队之间的这些规范往往是截然不同的。
本文给出几个被很多团队使用的异常处理最佳实践。

1. 在 finally 块中清理资源或者使用 try-with-resource 语句

当使用类似InputStream这种需要使用后关闭的资源时，一个常见的错误就是在try块的最后关闭资源。

```
1 public void doNotCloseResourceInTry() {
2  FileInputStream inputStream = null;
3  try {
4  File file = new File("./tmp.txt");
5  inputStream = new FileInputStream(file);
6  // use the inputStream to read a file
```

```
// do NOT do this
inputStream.close();
} catch (FileNotFoundException e) {
 log.error(e);
 } catch (IOException e) {
 log.error(e);
 }
}
```

问题就是，只有没有异常抛出的时候，这段代码才可以正常工作。try 代码块内代码会正常执行，并且资源可以正常关闭。但是，使用 try 代码块是有原因的，一般调用一个或多个可能抛出异常的方法，而且，你自己也可能会抛出一个异常，这意味着代码可能不会执行到 try 代码块的最后部分。结果就是，你并没有关闭资源。

所以，你应该把清理工作的代码放到 finally 里去，或者使用 try-with-resource 特性。

1.1 使用 finally 代码块

与前面几行 try 代码块不同，finally 代码块总是会被执行。不管 try 代码块成功执行之后还是你在 catch 代码块中处理完异常后都会执行。因此，你可以确保你清理了所有打开的资源。

```
public void closeResourceInFinally() {
 FileInputStream inputStream = null;
 try {
 File file = new File("./tmp.txt");
 inputStream = new FileInputStream(file);
 // use the inputStream to read a file
 } catch (FileNotFoundException e) {
 log.error(e);
 } finally {
 if (inputStream != null) {
 try {
 inputStream.close();
 } catch (IOException e) {
 log.error(e);
 }
 }
 }
```

```
}
```

1.2 Java 7 的 try-with-resource 语法

如果你的资源实现了 AutoCloseable 接口，你可以使用这个语法。大多数的 Java 标准资源都继承了这个接口。当你在 try 子句中打开资源，资源会在 try 代码块执行后或异常处理后自动关闭。

```
public void automaticallyCloseResource() {
 File file = new File("./tmp.txt");
 try (FileInputStream inputStream = new FileInputStream(file);) {
 // use the inputStream to read a file
 } catch (FileNotFoundException e) {
 log.error(e);
 } catch (IOException e) {
 log.error(e);
 }
}

```

2. 优先明确的异常

你抛出的异常越明确越好，永远记住，你的同事或者几个月之后的你，将会调用你的方法并且处理异常。

因此需要保证提供给他们尽可能多的信息。这样你的 API 更容易被理解。你的方法的调用者能够更好的处理异常并且避免额外的检查。

因此，总是尝试寻找最适合你的异常事件的类，例如，抛出一个 NumberFormatException 来替换一个 IllegalArgumentException 。避免抛出一个不明确的异常。

```
public void doNotDoThis() throws Exception {
 ...
}
public void doThis() throws NumberFormatException {
 ...
}

```

3. 对异常进行文档说明

当在方法上声明抛出异常时，也需要进行文档说明。目的是为了给调用者提供尽可能多的信息，从而可以更好地避免或处理异常。

在 Javadoc 添加 @throws 声明，并且描述抛出异常的场景。

```
public void doSomething(String input) throws MyBusinessException {
 ...
}
```

4. 使用描述性消息抛出异常

在抛出异常时，需要尽可能精确地描述问题和相关信息，这样无论是打印到日志中还是在监控工具中，都能够更容易被人阅读，从而可以更好地定位具体错误信息、错误的严重程度等。

但这里并不是说要对错误信息长篇大论，因为本来 Exception 的类名就能够反映错误的原因，因此只需要用一到两句话描述即可。

如果抛出一个特定的异常，它的类名很可能已经描述了这种错误。所以，你不需要提供很多额外的信息。一个很好的例子是 NumberFormatException 。当你以错误的格式提供 String 时，它将被 java.lang.Long 类的构造函数抛出。

```
try {
 new Long("xyz");
} catch (NumberFormatException e) {
 log.error(e);
}
```

5. 优先捕获最具体的异常

大多数 IDE 都可以帮助你实现这个最佳实践。当你尝试首先捕获较不具体的异常时，它们会报告无法访问的代码块。

但问题在于，只有匹配异常的第一个 catch 块会被执行。 因此，如果首先捕获 IllegalArgumentException ，则永远不会到达应该处理更具体的 NumberFormatException 的 catch 块，因为它是 IllegalArgumentException 的子类。

总是优先捕获最具体的异常类，并将不太具体的 catch 块添加到列表的末尾。

你可以在下面的代码片断中看到这样一个 try-catch 语句的例子。 第一个 catch 块处理所有 NumberFormatException 异常，第二个处理所有非 NumberFormatException 异常的IllegalArgumentException 异常。

```
public void catchMostSpecificExceptionFirst() {
 try {
 doSomething("A message");
```

```
 } catch (NumberFormatException e) {
 log.error(e);
 } catch (IllegalArgumentException e) {
 log.error(e)
 }
}

```

6. 不要捕获 Throwable 类

Throwable 是所有异常和错误的超类。你可以在 catch 子句中使用它，但是你永远不应该这样做！

如果在 catch 子句中使用 Throwable ，它不仅会捕获所有异常，也将捕获所有的错误。JVM 抛出错误，指出不应该由应用程序处理的严重问题。 典型的例子是 OutOfMemoryError 或者 StackOverflowError 。两者都是由应用程序控制之外的情况引起的，无法处理。

所以，最好不要捕获 Throwable ，除非你确定自己处于一种特殊的情况下能够处理错误。

```
public void doNotCatchThrowable() {
 try {
 // do something
 } catch (Throwable t) {
 // don't do this!
 }
}
```

7. 不要忽略异常

很多时候，开发者很有自信不会抛出异常，因此写了一个catch块，但是没有做任何处理或者记录日志。

```
public void doNotIgnoreExceptions() {
 try {
 // do something
 } catch (NumberFormatException e) {
 // this will never happen
 }
}
```

但现实是经常会出现无法预料的异常，或者无法确定这里的代码未来是不是会改动(删除了阻止异常抛出的代码)，而此时由于异常被捕获，使得无法拿到足够的错误信息来定位问题。

合理的做法是至少要记录异常的信息。

```
public void logAnException() {
 try {
 // do something
 } catch (NumberFormatException e) {
 log.error("This should never happen: " + e);
 }
}
```

8. 不要记录并抛出异常

这可能是本文中最常被忽略的最佳实践。可以发现很多代码甚至类库中都会有捕获异常、记录日志并再次抛出的逻辑。如下：

```
try {
 new Long("xyz");
} catch (NumberFormatException e) {
 log.error(e);
 throw e;
}
```

这个处理逻辑看着是合理的。但这经常会给同一个异常输出多条日志。如下：

```
17:44:28,945 ERROR TestExceptionHandling:65 - java.lang.NumberFormatException: For input string: "xyz"
Exception in thread "main" java.lang.NumberFormatException: For input string: "xyz"
 at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
 at java.lang.Long.parseLong(Long.java:589)
 at java.lang.Long.(Long.java:965)
 at com.stackify.example.TestExceptionHandling.logAndThrowException(TestExceptionHandling.java:63)
 at com.stackify.example.TestExceptionHandling.main(TestExceptionHandling.java:8)
```

如上所示，后面的日志也没有附加更有用的信息。如果想要提供更加有用的信息，那么可以将异常包装为自定义异常。

```
public void wrapException(String input) throws MyBusinessException {
 try {
```

```
  // do something
  } catch (NumberFormatException e) {
  throw new MyBusinessException("A message that describes the error.", e);
  }
 }
```

因此，仅仅当想要处理异常时才去捕获，否则只需要在方法签名中声明让调用者去处理。

9. 包装异常时不要抛弃原始的异常

捕获标准异常并包装为自定义异常是一个很常见的做法。这样可以添加更为具体的异常信息并能够做针对的异常处理。

在你这样做时，请确保将原始异常设置为原因（注：参考下方代码 NumberFormatException e 中的原始异常 e ）。Exception 类提供了特殊的构造函数方法，它接受一个 Throwable 作为参数。否则，你将会丢失堆栈跟踪和原始异常的消息，这将会使分析导致异常的异常事件变得困难。

```
public void wrapException(String input) throws MyBusinessException {
 try {
 // do something
 } catch (NumberFormatException e) {
 throw new MyBusinessException("A message that describes the error.", e);
 }
}

```

10. 不要使用异常控制程序的流程

不应该使用异常控制应用的执行流程，例如，本应该使用if语句进行条件判断的情况下，你却使用异常处理，这是非常不好的习惯，会严重影响应用的性能。

11. 使用标准异常

如果使用内建的异常可以解决问题，就不要定义自己的异常。Java API 提供了上百种针对不同情况的异常类型，在开发中首先尽可能使用 Java API 提供的异常，如果标准的异常不能满足你的要求，这时候创建自己的定制异常。尽可能得使用标准异常有利于新加入的开发者看懂项目代码。

12. 异常会影响性能

异常处理的性能成本非常高，每个 Java 程序员在开发时都应牢记这句话。创建一个异常非常慢，抛出一个异常又会消耗1~5ms，当一个异常在应用的多个层级之间传递时，会拖累整个应用的性能。

- 仅在异常情况下使用异常；
- 在可恢复的异常情况下使用异常；

尽管使用异常有利于 Java 开发，但是在应用中最好不要捕获太多的调用栈，因为在很多情况下都不需要打印调用栈就知道哪里出错了。因此，异常消息应该提供恰到好处的信息。

13. 总结

综上所述，当你抛出或捕获异常的时候，有很多不同的情况需要考虑，而且大部分事情都是为了改善代码的可读性或者 API 的可用性。

异常不仅仅是一个错误控制机制，也是一个通信媒介。因此，为了和同事更好的合作，一个团队必须要制定出一个最佳实践和规则，只有这样，团队成员才能理解这些通用概念，同时在工作中使用它。

异常处理-阿里巴巴Java开发手册

1. 【强制】Java 类库中定义的可以通过预检查方式规避的RuntimeException异常不应该通过catch 的方式来处理，比如：NullPointerException，IndexOutOfBoundsException等等。 说明：无法通过预检查的异常除外，比如，在解析字符串形式的数字时，可能存在数字格式错误，不得不通过catch NumberFormatException来实现。 正例：if (obj != null) {...} 反例：try { obj.method(); } catch (NullPointerException e) {...}
2. 【强制】异常不要用来做流程控制，条件控制。 说明：异常设计的初衷是解决程序运行中的各种意外情况，且异常的处理效率比条件判断方式要低很多。
3. 【强制】catch时请分清稳定代码和非稳定代码，稳定代码指的是无论如何不会出错的代码。对于非稳定代码的catch尽可能进行区分异常类型，再做对应的异常处理。 说明：对大段代码进行try-catch，使程序无法根据不同的异常做出正确的应激反应，也不利于定位问题，这是一种不负责任的表现。 正例：用户注册的场景中，如果用户输入非法字符，或用户名

称已存在，或用户输入密码过于简单，在程序上作出分门别类的判断，并提示给用户。

4. 【强制】捕获异常是为了处理它，不要捕获了却什么都不处理而抛弃之，如果不想处理它，请将该异常抛给它的调用者。最外层的业务使用者，必须处理异常，将其转化为用户可以理解的内容。

5. 【强制】有try块放到了事务代码中，catch异常后，如果需要回滚事务，一定要注意手动回滚事务。

6. 【强制】finally块必须对资源对象、流对象进行关闭，有异常也要做try-catch。 说明：如果JDK7及以上，可以使用try-with-resources方式。

7. 【强制】不要在finally块中使用return。 说明：try块中的return语句执行成功后，并不马上返回，而是继续执行finally块中的语句，如果此处存在return语句，则在此直接返回，无情丢弃掉try块中的返回点。 反例：

```
private int x = 0;
public int checkReturn() {
 try {
 // x等于1，此处不返回
 return ++x;
 } finally {
 // 返回的结果是2
 return ++x;
 }
}
```

1. 【强制】捕获异常与抛异常，必须是完全匹配，或者捕获异常是抛异常的父类。 说明：如果预期对方抛的是绣球，实际接到的是铅球，就会产生意外情况。

2. 【强制】在调用RPC、二方包、或动态生成类的相关方法时，捕捉异常必须使用Throwable类来进行拦截。 说明：通过反射机制来调用方法，如果找不到方法，抛出NoSuchMethodException。什么情况会抛出NoSuchMethodError呢？二方包在类冲突时，仲裁机制可能导致引入非预期的版本使类的方法签名不匹配，或者在字节码修改框架（比如：ASM）动态创建或修改类时，修改了相应的方法签名。这些情况，即使代码编译期是正确的，但在代码运行期时，会抛出NoSuchMethodError。

3. 【推荐】方法的返回值可以为null，不强制返回空集合，或者空对象等，必须添加注释充分说明什么情况下会返回null值。 说明：本手册明确防止NPE是调用者的责任。即使被调用方法返回空集合或者空对象，对调用者来说，也并非高枕无忧，必须考虑到远程调用失败、序列化失败、运行时异常等场景返回null的情况。

4. 【推荐】防止NPE，是程序员的基本修养，注意NPE产生的场景： 1）返回类型为基本数据类型，return包装数据类型的对象时，自动拆箱有可能产生NPE。 反例：public int f() { return Integer对象}， 如果为null，自动解箱抛NPE。 2） 数据库的查询结果可能为null。 3） 集合里的元素即使isNotEmpty，取出的数据元素也可能为null。 4） 远程调用返回对象时，一律要求进行空指针判断，防止NPE。 5） 对于Session中获取的数据，建议进行NPE检查，避免空指针。 6） 级联调用obj.getA().getB().getC()；一连串调用，易产生NPE。

正例：使用JDK8的Optional类来防止NPE问题。

5. 【推荐】定义时区分unchecked / checked 异常，避免直接抛出new RuntimeException()，更不允许抛出Exception或者Throwable，应使用有业务含义的自定义异常。推荐业界已定义过的自定义异常，如：DAOException / ServiceException等。

6. 【参考】对于公司外的http/api开放接口必须使用“错误码”；而应用内部推荐异常抛出；跨应用间RPC调用优先考虑使用Result方式，封装isSuccess()方法、“错误码”、“错误简短信息”。 说明：关于RPC方法返回方式使用Result方式的理由： 1）使用抛异常返回方式，调用方如果没有捕获到就会产生运行时错误。 2）如果不加栈信息，只是new自定义异常，加入自己的理解的error message，对于调用端解决问题的帮助不会太多。如果加了栈信息，在频繁调用出错的情况下，数据序列化和传输的性能损耗也是问题。

7. 【参考】避免出现重复的代码（Don’t Repeat Yourself），即DRY原则。 说明：随意复制和粘贴代码，必然会导致代码的重复，在以后需要修改时，需要修改所有的副本，容易遗漏。必要时抽取共性方法，或者抽象

公共类，甚至是组件化。 正例：一个类中有多个public方法，都需要进行数行相同的参数校验操作，这个时候请抽取：

private boolean checkParam(DTO dto) {...}