> 介绍了异常处理相关的Throwable是什么？Eception是什么？Error是什么？try-catch-finally的使用和优缺点等各方面内容，可以参考目录，
> 鸣谢：《Think in java》

Java 异常(41)

版本号：2918/8/28-1(16:24)

面试题(4)

1、Execption和Error的异同？

> 1. 都继承自 `Throwable`

2. Error一般是系统级错误、JMV的错误，如 `OutOfMemoryError` ，都是无法解决的问题。
3. Exception分为检查型异常和非检查型异常
4. 检查型异常是直接继承自Exception的异常，需要 显式捕获、抛出和处理
5. 非检查型异常是继承自 `RuntimeException` 的异常，是否捕获处理根据实际情况，一般是逻辑错误。需要修改代码。

2、运行时异常和一般异常有什么区别?

1. 运行时异常就是 `RuntimeException` 及其子类，不要求一定要显示处理。
2. 一般异常是直接继承自 `Exception` 的异常，需要 `try-catch` 或者 `throw`
3. RuntimeException(非检查型异常)的抛出，只需要throw

```
private void runTimeException(){
    throw new RuntimeException();
}
```

1. 一般的Exception(检查型异常)，不仅需要throw，还需要throws

```
private void exception() throws Exception {
    throw new Exception();
}
```

3、NoClassDefFoundError和ClassNotFoundException的区别?

1. 前者是Error(LinkageError)，后者是Exception(RuntimeException)
2. NoClassDefFoundError发生在JVM在动态运行时，在classpath中没有找到对应的类。
3. Exception还是可以尝试去恢复，Error一般没有办法恢复。
4. ClassNotFoundException: 出现在类的加载阶段，从外部存储器找不到class。
5. NoClassDefFoundError: 出现在类的连接阶段，从内存找不到class。

4、throw和throws的区别?

1. throws: 声明一个方法可能抛出的异常信息，是一种异常声明。
2. throw: 抛出具体的异常类型。
3. throws：抛出检查型异常的方法一定要使用 `throws` 进行声明。抛出非检查型异常的方法不需要该声明。

基础(9)

1、为什么需要异常处理?

工程师最理想状态是在编译时就能发现所有错误，但是不是所有的错误都可以被找到。这些错误需要在运行时被处理，这就需要异常处理机制。

2、 异常处理的优点

> 1.提高系统的稳健性。
> 2.减少处理错误的代码的复杂度。
> 3.不需要再在方法调用中去处理，而是直接在exception handler去处理。

3、 throw exception的原因

> 当前异常的环境无法处理该异常，就需要将问题提交给更高层去处理。
> 抛出异常也可以看做一种return机制，当抛出异常时，其所处的方法就会退出。

4、 Exception arguments参数

> 异常对象构造器有两种，一种是默认的，一种是带String参数的，用于提供更多的错误信息。

5、 try-catch和switch区别

> try-catch由上至下匹配异常类型，一旦匹配到就会进入相应的catch进行处理，处理完成后不会继续向下匹配。而switch必须通过break来停止流程。

6、 Termination vs resumption(终止和继续)

> Java最支持的就是终止-该错误严重到无法继续执行。如果你想能继续执行，就不能单纯的throw exception，而是需要将try-catch块放置于while循环中，直到结果满意为止。

7、 resumption的缺点

> 继续执行其实实际用途中没什么用，因为resumption会导致handler需要关注异常产生在哪里，并且要包含关于异常抛出点的非通用代码。这样使得系统非常复杂。

抛出异常的流程

8、抛出异常时的流程

> 1. exception对象会被创建，如同其他java对象被new创建一样，创建在堆heap中。
> 2. 当前异常的路径会停止，并且异常对象的引用会被当前context抛出。
> 3. 此时，异常处理机制(exception-handling mechanism)会接管，并且开始寻找合适的地方去继续执行程序。
> 4. 合适的地方, 就是catch中，会进行问题修复。

异常链

9、 catch中再次抛出该异常

```
catch(exception e){
  throw e;
}
```

> 这里再次抛出的异常e，本身类型是其初始的类型，而不会变成Exception类型。这种特性被称为-exception chaining,异常链

Throwable(14)

1、Throwable是什么?

> 1. 继承自 `Object`
> 2. 是所有异常和错误的超类。
> 3. Java中只有 `Throwable` 类型的实例才可以被 `throw` 和 `catch`
> 4. 有两个子类 `Exception` 和 `Error`

Exception

2、Exception的分类:

> 1. 检查型异常：直接继承自 `Exception` ， 如 `IOException`
> 2. 非检查型异常：继承自 `RunTimeException` -继承自 `Exception`

3、抛出Exception的限制

> 1. 子类重写的方法，抛出的异常不能比父类该方法抛出的异常范围大。
> 2. 子类重写的方法，抛出的异常要在父类该方法抛出的异常范围之内。
> 3. 子类构造器中，可以抛出任何范围的异常，而不会受到限制。

```
情况1:
public String getDescription() throws RuntimeException{
                return "SuperClass";
}
public String getDescription() throws Exception{ ! 错误! 不可以比超类范围大
                return "子类";
}
情况2:
public String getDescription() throws CarException{
                return "SuperClass";
}
public String getDescription() throws FoodException{ !错误!超类抛出异常和子类抛出异常不是同类异常
                return "子类";
}
```

4、不知道如何处理某些异常：将其转换为系统处理的异常

> 1. 有个异常在当前范围内不知道如何处理，可以直接作为 `RuntimeException` 抛出
> 2. 之后处理throw出的异常处，可以通过e.getCause()去判断出原来的异常类型并且作相应处理。

```
try{
         // ... to do something useful
} catch(IDontKnowWhatToDoWithThisCheckedException e) {
        throw new RuntimeException(e); //不知道如何处理，直接转换为Runtime异常抛出。
}
//之后处理throw出的异常处，可以通过e.getCause()去判断出原来的异常类型并且作相应处理。
```

RuntimeException

5、RuntimeException是什么?

> 1. 继承自 `Exception`
> 2. RuntimeException和子类都属于 非检查型异常
> 3. Java自动抛出，不需要手动去抛出。
> 4. 非检查型异常可以根据情况决定是否去try-catch
> 5. 常见子
>
>   类： `NullPointerException`、`ClassNotFoundException`、`ArrayIndexOutOfBoundsException`

ClassNotFoundException

6、ClassNotFoundException导致的原因?

> 1. 找不到指定的class
> 2. 一个类已经被某个类加载器加载到内存中，此时另一个类加载器尝试动态地加载这个类。
> 3. 该错误出现在 类的加载阶段

7、常见场景

> 1. 调用Class.forName(): 找不到指定的类
> 2. 调用ClassLoader.findSystemClass()： 找不到指定的类
> 3. 调用ClassLoader.loadClass(): 找不到指定的类

Error

8、Error是什么?

> 1. 继承自 `Throwable`
> 2. 系统级、JVM级别的异常。
> 3. 子类： `LinkageError`、`VirtualMachineError`、

VirtualMachineError

9、VirtualMachineError是什么?

> 1. 继承自 `Error`
> 2. 表示：JVM损坏或者耗尽了必要资源。
> 3. 典型子类： `OutOfMemoryError` -大名鼎鼎的OOM

LinkageError

10、LinkageError是什么?

> 1. 继承自 `Error`
> 2. 表示：一个类依赖于其他的类，但是在该类编译好后，依赖的那些类发生了不兼容性的改变。
> 3. 出现场景：如Jar包重复。
> 4. 典型子类： `NoClassDefFoundError`

NoClassDefFoundError

11、NoClassDefFoundError是什么?

> 1. JVM在编译时找到了合适的类，但是在运行时不能找到合适的类。会导致该错误。
> 2. 运行时 抛出该异常。
> 3. 出现在 类的链接阶段 ，从内存找不到需要的class

12、产生的原因

> 1. 缺少jar文件，或者jar文件没有添加到classpath，或者jar的文件名发生变更会导致java.lang.NoClassDefFoundError的错误。
> 2. 如果你工作在J2EE的环境，有多个不同的类加载器，也可能导致NoClassDefFoundError

13、常见场景

> 1. 类依赖的class或者jar不存在
> 2. 类文件存在，但是不处于classpath中，直接调用也找不到。
> 3. javac编译不区分大小写，编译出来的class和需要的不一致。

自定义异常

14、 创建自己的异常

> 1. 继承自Exception： `class SimpleException extends Exception{}`
> 2. 一般都不需要自己实现内容，直接继承Exception即可。
> 3. 一个异常最重要的部分就是class name。
> 4. 需要考虑是否定义成 检查型异常 还是 非检查型异常
> 5. 要注意信息安全，不要将重要私密信息显示出来。
> 6. 可以创建包含String的构造器 `public SimpleException(String msg){ super(msg);}` 。
> 7. 在catch中 `e.printStackTrace(System.out);` 将信息送到System.out流中，也可以用默认参数，显示到标准错误流中。

try-catch(6)

finally

1、try-catch-finally

> 1. 执行无论异常与否都必须执行的代码。
> 2. 可以用于文件、网络连接等内容的清理工作。
> 3. 在涉及到break和continue的部分，finally也一定会执行(在java中都可以消除goto的需求)

2、 finally和return

> 1. 在finally之前的任何return都不会影响finally块的执行。

缺点

3、 finally缺陷:会丢失异常

```
try {
    try {
            //抛出异常a
    } finally {
            //抛出异常b
    }
} catch (Exception e) {
     //获得异常b，而异常a丢失
}
```

```
try {
      throw new RuntimeException();
} finally {
      //在finally中return，会导致本该抛出的异常却丢失了
   return;
}
```

try-with-resources

4、try-with-resources

> 1. java.lang包中。
> 2. try-catch-finally因为容易丢失异常，在Java 7中推出该特性。
> 3. 需要对象实现了 `AutoCloseable` 接口，并在close()方法中执行清理操作。
> 4. 不再需要Finally

```
try(Resource resource = new Resource();){
    //做一些工作
} catch (IOException e) {
    e.printStackTrace();
}
```

```
public class Resource implements AutoCloseable{
    @Override
    public void close() throws IOException {
        //清理工作
        System.out.println("清理工作");
    }
}
```

5、AutoCloseable、Closeable和Flushable接口的区别

> 1. AutoCloseable Java 7，继承并且实现其 `close` 方法，就可以用于 `try-with-resources`
> 2. Closeable Java 5提出，Java 7后已经继承了AutoCloseable。
> 3. Flushable接口的类的对象，可以强制将缓存的输出写入到与对象关联的流中。该接口定义了flush()方法

性能开销

6、异常处理的性能开销

> 1. try-catch: 该代码段往往会影响到JVM对代码的优化，建议只去捕获必要的代码段，禁止大面积try-catch.
> 2. Java实例化Exception时会对栈进行快照：这是重量级操作，如果发生的非常频繁，会影响性能。

异常使用规范(8)

1、禁止捕获通用的Exception
2、禁止生吞异常

```
try {
    Thread.sleep(1000L);
}catch (Exception e){
    // 不作任何工作
}
```

> 1. 不应该采用通用的Exception。
> 2. 不应该生吞异常。

3、不要捕获Throwable或者Error

> 因为出现了OOM这种Error是很难进行恢复的。

4、不要使用e.printStackTrace()

> 1. 该方法是将异常和调用栈输出到 标准错误流 中。
> 2. 只适合调试阶段，不适合上线的产品。应该输入到错误日志中。

```
try {
    Thread.sleep(1000L);
}catch (InterruptedException e) {
    e.printStackTrace();
}
```

5、尽早抛出异常，避免难以定位问题。

> 反例：会抛出文件没有找到的异常，难以定位。

```
public void readFile(String filename) throws FileNotFoundException {
    InputStream inputStream = new FileInputStream(filename);
}
```

> 正确：两种情况分开抛出异常，方便定位。

```
public void readFile2(String filename) throws FileNotFoundException {
    // 如果为NULL会抛出异常
    Objects.requireNonNull(filename);
    // 正常使用，会抛出FileNotFoundException异常
    InputStream inputStream = new FileInputStream(filename);
}
```

6、自定义异常要保证信息的安全性。

> 1. `java.net.ConnectionException` 在连接错误时，只会显示错误，而不会去显示机器名、ip、端口等信息。减少安全风险。

构造方法中的异常

7、Constructors(构造器)中的异常

```
/**-------------------------------------------------
 *  问题：在构造器中产生了异常，如何关闭需要清理的资源，如文件？
 *  解析：绝对不能在finally中清理资源。
 *            1.这样会导致无论异常与否资源都会被清理
 *            2.而且前面的异常可能会导致需要被清理的资源本质上也没创建。
 * -------------------------------------------------*/
public class InputFile {
          private BufferedReader in;
          public InputFile(String fname) throws Exception {
            try {
              in = new BufferedReader(new FileReader(fname));
              // 其他可能抛出异常的代码
            } catch(FileNotFoundException e) {
              System.out.println("Could not open " + fname);
              // 没有打开,就不需要关闭
              throw e;
            } catch(Exception e) {
              // 其他异常,必须要关闭文件
              try {
                in.close();
              } catch(IOException e2) {
                System.out.println("in.close() unsuccessful");
              }
              throw e; // Rethrow(再次抛出)
            } finally {
              // Don’t close it here!!!
            }
          }
          //其他功能...
}
```

指导方案

8、 Exception Guidelines要点

> - 一言以蔽之：在合适层级修复问题，以完成预期功能，并提高系统的健壮性、安全性。

```
// Java编程思想中的原文
 1.  Handle problems at the appropriate level. (Avoid catching exceptions unless you know what
 2.  Fix the problem and call the method that caused the exception again.
 3.  Patch things up and continue without retrying the method.
 4.  Calculate some alternative result instead of what the method was supposed to produce.
 5.  Do whatever you can in the current context and rethrow the same exception to a
higher context.
 6.  Do whatever you can in the current context and throw a different exception to a higher cor
 7.  Terminate the program.
 8.  Simplify. (If your exception scheme makes things more complicated, then it is painful and
 9. Make your library and program safer. (This is a short-term investment for debugging, and a
```

参考资料

1. java.lang.ClassNotFoundException与java.lang.NoClassDefFoundError的区别
2. ClassNotFoundException和NoClassDefFoundError的区别
3. io中的AutoCloseable，Closeable和Flushable接口