JVM是如何处理异常的?

版本：2018/9/11-1(19:00)

基础(13)

1、异常处理涉及到哪两方面的内容?

> 1. 抛出异常
> 2. 处理异常

2、抛出异常的两种方式

> 1. 显式抛出异常，在程序同使用 `throw` 关键字，手动将异常实例抛出。
> 2. 隐式抛出异常，抛出的主体是JVM，在运行时遇到无法处理的异常情况。

3、处理异常涉及的三种代码块?

> 1. try
> 2. catch
> 3. finally

4、try代码块的作用?

> 用于 标记 需要进行异常监控的代码

5、catch代码块的作用?

> 1. 声明了需要捕获的异常类型
> 2. 定义了异常类型所对应的 异常处理器

6、catch代码块中异常处理器的匹配规则?

> 1. JVM按顺序进行匹配
> 2. 因此异常类型范围，需要从小到大，不然会导致覆盖。
> 3. 编译器也会报错

7、finally代码块的作用?

> 1. 用来声明一段必须要执行的代码
> 2. 目的是进行一些清理工作。

8、try-catch-finally中，在try里面出现异常，但是没有去捕获该异常类型，此时会发生什么情况?

> 1. finally中代码会正常运行
> 2. finally运行后，抛出try中遇到的异常。

9、如果catch代码块中出现了异常，finally的代码是否会执行?

> 1. finally代码块依旧会执行
> 2. 执行完毕后，抛出catch中的异常。

10、如果finally中出现异常，会怎么样?

> 1. 会中断finally的执行
> 2. 然后抛出异常。

11、Java语言规范中，所有的异常都是哪个类的子类?

> Throwable

12、Throwable有哪些子类?

> 1. Error： 不应该捕获的异常，此时程序已经无法恢复，需要终止线程，或者终止虚拟机。
> 2. Exception: 可能需要捕获，并且处理的异常。

13、Exception的子类

> 1. RuntimeException： 运行时异常，非检查型异常(Error也是非检查型异常)
> 2. 其他Exception子类：检查型异常，需要代码中显式捕获。或者方法用 `throws` 标记

异常的创建(6)

1、异常实例构造的性能损耗

> 1. 异常实例的构造具有比较高的性能开销
> 2. 在构造时，JVM需要生成该异常的 栈轨迹-stack trece
> 3. 该操作会逐一访问当前线程的Java 栈帧
> 4. 并记录下各种调试信息：栈帧所指向方法的名字、方法所在类名、方法类所在文件名、以及在代码中的第几行会触发该异常。

2、Throwable.fillInStackTrace()的作用?

> 1. 来装填当前线程的栈帧信息

3、JVM生成栈轨迹的注意点

> 1. JVM会忽略掉异常构造器，以及填充栈帧的Java方法(Throwable.fillInStackTrace),直接从创建异常的位置开始算起。
> 2. JVM会忽略掉标记为不可见的Java方法栈帧

4、异常实例的构造有很高的开销，为什么不去缓存异常实例呢?

> 1. 语法上讲，是可以缓存异常实例，并且在需要用到的时候直接抛出。
> 2. 但是该异常对应的栈轨迹，并不是 throw语句 的位置，而是创建该异常的位置。
> 3. 会导致开发者定位到错误位置，因此实践中都是新建异常，并且抛出。

5、Throwable.fillInStackTrace()会影响即使编译器(JIT)的优化吗?

> 不会
>
> 1. 即使编译器生成的代码会保存原始的栈信息，以便去优化时能复原。
> 2. fillInStackTrace()也会去读取这些信息，不需要优化好后，再进行 fillInStackTrace()-装填当前线程的栈帧信息

6、抛异常操作本身的性能问题

> 1. 抛异常的操作本身会导致额外的执行路径
> 2. 但是如果能将异常处理器也编译进去，就不会有太大的影响。
> 3. 因此抛异常不太会影响JIT的优化

异常的捕获(8)

1、JVM是如何捕获异常的?

> 1. 编译而成的字节码中，每个方法都附带一个异常表。

2. 异常表中每一个条目代表一个 异常处理器
3. 触发异常时，JVM会遍历异常表，比较 触发异常的字节码的索引值 是否在异常处理器的 from指针 到 to指针 的范围内。
4. 范围匹配后，会去比较 异常类型 和 异常处理器 中的 type 是否相同。
5. 类型匹配后，会跳转到 target指针 所指向的字节码(catch代码块的开始位置)
6. 如果没有匹配到异常处理器，会弹出当前方法对应的Java栈帧，并对调用者重复上述操作。

2、什么是异常表?

1. 每个方法都附带一个异常表
2. 异常表中每一个条目, 就是一个异常处理器
3. 异常表如下:

from	to	target	type
0	3	6	IOException
0	3	11	Exception

3、什么是异常处理器？其组成部分有哪些?

1. 异常处理器由 from指针、to指针、target指针 ，以及所捕获的异常类型所构成(type)。
2. 这些指针的数值就是字节码的索引(bytecode index, bci),可以直接去定位字节码。
3. from指针和to指针，标识了该异常处理器所监控的返回
4. target指针，指向异常处理器的起始位置。如 catch代码块的起始位置
5. type：捕获的异常类型，如Exception

4、如果在方法的异常表中没有匹配到异常处理器，会怎么样?

1. 会弹出当前方法对应的Java栈帧
2. 在调用者上重复异常匹配的流程。
3. 最坏情况下，JVM需要编译当前线程Java栈上所有方法的异常表

5、finally代码块是如何去实现的?

1. 在编译阶段对finally代码块进行处理
2. 当前版本Java编译器的做法，是复制finally代码块的内容，分别放到所有正常执行路径，以及异常执行路径的出口中。

6、finally代码块实例

1. 有三分finally代码块
    1. 第一份复制的finally代码块，位于try代码后。try代码块出现异常跳转到catch代码块，如果catch无法捕获，会跳转到最后一份finally代码块。
    2. 第二份复制的finally代码块，位于catch代码后。catch代码块出现异常会跳转到最后一份finally代码块。

> 3. 最后一份finally代码块，位于异常执行路径，运行完后，抛出any异常。
> 2. try和catch代码块中的finally代码块，如果出现了异常，也会直接抛出any异常。
> 3. javap中用any代指所有种类的异常。
>
> Java代码

```
public class Test {
    // 便于查看字节码
    private int tryBlock;
    private int catchBlock;
    private int finallyBlock;
    private int methodExit;

    public void test(){
        try {
            tryBlock = 0;
        }catch (Exception e){
            catchBlock = 1;
        }finally {
            finallyBlock = 2;
        }
        methodExit = 3;
    }
}
```

> 字节码

```
public void test();
  descriptor: ()V
  flags: ACC_PUBLIC
  Code:
    stack=2, locals=3, args_size=1
       0: aload_0
       1: iconst_0
       2: putfield      #2                  // Field tryBlock:I
       5: aload_0
       6: iconst_2
       7: putfield      #3                  // Field finallyBlock:I
      10: goto          35
      13: astore_1
      14: aload_0
      15: iconst_1
      16: putfield      #5                  // Field catchBlock:I
      19: aload_0
      20: iconst_2
      21: putfield      #3                  // Field finallyBlock:I
      24: goto          35
      27: astore_2
      28: aload_0
      29: iconst_2
      30: putfield      #3                  // Field finallyBlock:I
      33: aload_2
      34: athrow
      35: aload_0
      36: iconst_3
      37: putfield      #6                  // Field methodExit:I
      40: return
    Exception table:
       from    to  target type
           0     5    13   Class java/lang/Exception
           0     5    27   any
          13    19    27   any
```

7、try中抛出异常A，catch中出现了异常B，最终抛出的异常是哪一个?

> 异常B

8、为什么catch抛出的异常B会导致原来的异常A丢失?

> 1. catch中出现了异常后，会到异常表中去查找。
> 2. 最终从异常处理器中，根据target指针找到目标方法，也就是finally代码块。
> 3. finally代码块执行好后，会将捕获的异常B，抛出。

Supressed异常(5)

1、Supressed异常是什么?

1. Java7引入，目标是解决异常丢失问题。
2. 该新特性允许讲一个异常附加到另一个异常之上。
3. 让一个异常可以附带多个异常的信息。

2、Supressed异常的问题

1. finally代码块缺少能指向所捕获异常的引用，导致使用Suppressed非常繁琐

3、Java7还允许在同一个catch代码中捕获多种异常

```
try {
    tryBlock = 0;
}catch (IOException | FileNotFoundException e){
    catchBlock = 1;
}
```

try-with-resources

4、Java7中专门提供了try-with-reources语法糖解决该问题

1. 在字节码层面自动使用 Supressed异常
2. 其主要目的是，精简资源打开和关闭的方式。
3. 每个资源的打开和关闭都需要try-finally，如果有多个资源，会导致非常繁琐。

```
try {
    //打开资源A
    try {
        //打开资源B
        try {
            //打开资源C
        }finally {
            //关闭资源C
        }
    }finally {
        //关闭资源B
    }
}finally {
    //关闭资源A
}
```

5、try-with-resources的使用

1. 实现AutoCloseable接口
2. 在try中初始化实例

```
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");
    }
}
```

知识扩展

栈帧

1、栈帧是什么?

> 1. 栈帧-Stack Frame
> 2. 用于支持JVM进行方法调用和方法执行的数据结构
> 3. 是JVM运行时数据区的虚拟机栈(`Virtual Machine Stack`)的栈元素。

2、栈帧的组成部分

> 1. 方法的局部变量表
> 2. 操作数栈
> 3. 动态链接
> 4. 方法返回地址

3、一个方法调用从开始到结束，对应着一个栈帧在虚拟机栈中从入栈到出栈的过程

4、栈帧需要分配多少内存，是何时决定的?

> 1. 内部局部变量表有多大
> 2. 操作数栈有多深
> 3. 栈帧需要多少内存
> 4. 这些都是在编译阶段就已经决定，并且写入到了 方法表 的 `code` 属性中。
> 5. 不会受到运行时的影响。

5、什么是当前栈帧

> 1. 当前栈帧-Current Stack Frame
> 2. 活动的线程中只有虚拟机栈栈顶的栈帧才是有效的。被称为当前栈帧

> 3. 当前栈帧所关联的方法，被称为当前方法(Current Method)

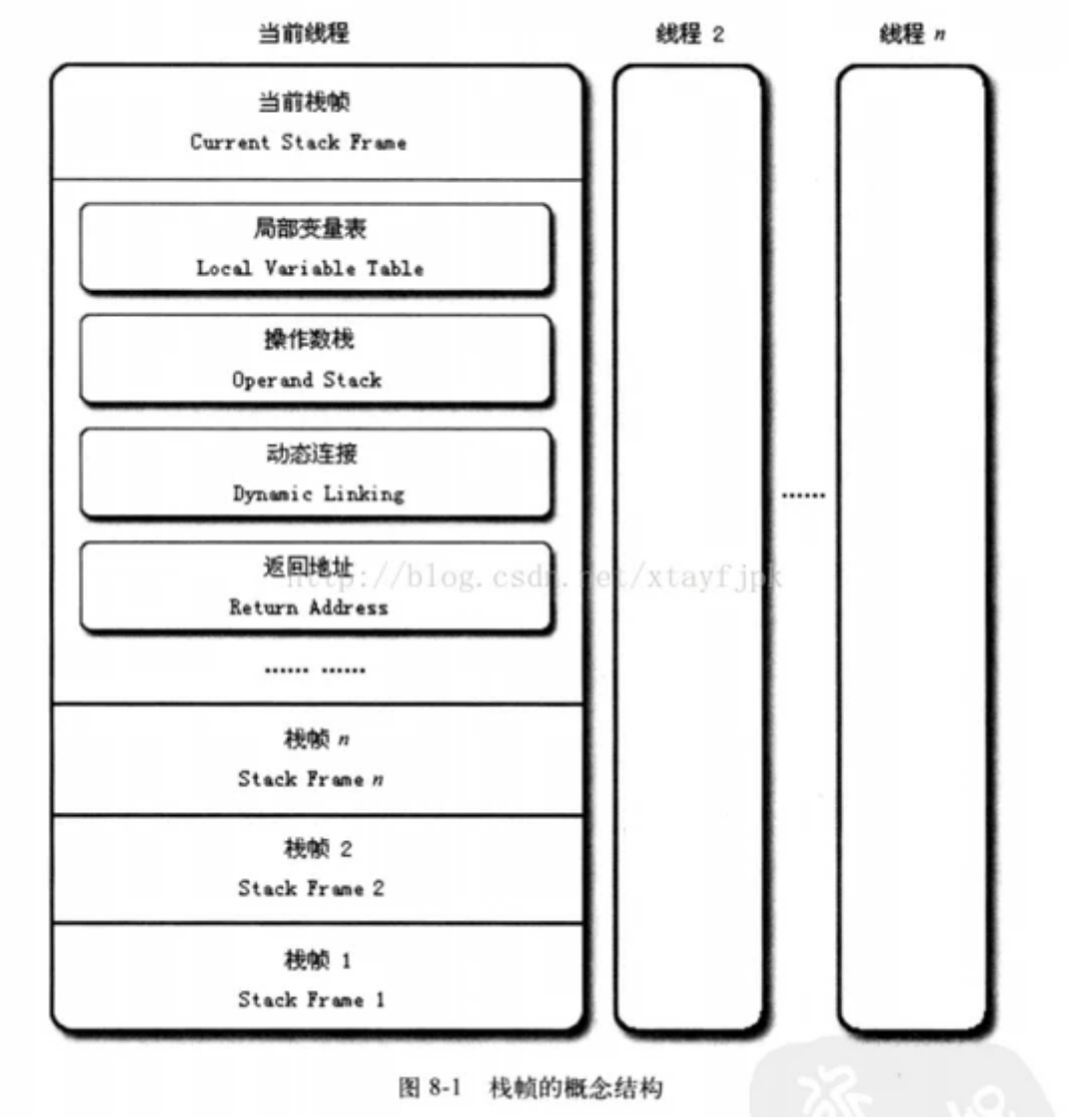

图 8-1　栈帧的概念结构

6、虚拟机栈中的栈帧顺序和方法调用顺序的关系?

> 1. 栈顶的栈帧，是最后一个调用的方法。
> 2. 栈底的栈帧，是第一个调用的方法。

局部变量表

7、局部变量表是什么?

> 1. 一组变量值的存储空间，用于存放方法参数、方法内部定义的局部变量
> 2. 局部变量表的大小，在Java编译阶段给定。
> 3. 具体大小存放于方法表的 `Code属性` 的 `max_locals` 数据项中

8、局部变量表的第0位索引的狭槽(Slot)的作用

> 1. 用于传递方法所属对象实例的引用
> 2. 通过 `this` 来访问这个隐含的参数

9、方法执行时，虚拟机使用局部变量表完成参数变量列表的传递过程

> 1. 除了 `this` ，其与参数按照参数列表的顺序排列，从index=1开始
> 2. 分配完毕后，剩余的slot(位置)会按照方法体内部定义的变量顺序和作用域来分配

10、类变量和局部变量的区别

> 1. 类变量存在 `准备阶段` ，具有两次赋值的过程。
> 2. 一次是在准备阶段，赋予系统初始值。
> 3. 一次是在初始化阶段，赋予开发者定义的数值。
> 4. 局部变量必须要赋予初始值才可以使用，类变量不需要。

栈轨迹

1、栈轨迹(Stack Trace)是什么?

> 1. 当前线程的虚拟机栈，从栈顶到栈尾，整个方法的调用轨迹。
> 2. 栈轨迹中，每一个元素，就是一个栈帧。
> 3. `e.getStackTrace()` 会返回栈轨迹中，所有元素所构成的数组。
> 4. index=0的元素，就是栈顶元素。也就是方法调用序列中的最后一个方法调用。
> 5. 尾部的元素，就是栈尾元素。也就是方法调用序列中的第一个方法调用。

问题汇总

> 汇总JVM层面关于异常的所有问题。这些问题你都知道答案吗?
> 一般题目的答案都在文中给出。
> 【☆】标记的题目，会直接给出。属于补充题。

1. 异常处理涉及到哪两方面的内容?
2. 抛出异常的两种方式
3. 处理异常涉及的三种代码块?
4. try代码块的作用?
5. catch代码块的作用?
6. catch代码块中异常处理器的匹配规则?
7. finally代码块的作用?
8. try-catch-finally中，在try里面出现异常，但是没有去捕获该异常类型，此时会发生什么情况?
9. 如果catch代码块中出现了异常，finally的代码是否会执行?
10. 如果finally中出现异常，会怎么样?
11. Java语言规范中，所有的异常都是哪个类的子类?
12. Throwable有哪些子类?
13. Exception的子类
14. 异常实例构造的性能损耗
15. Throwable.fillInStackTrace()的作用?
16. JVM生成栈轨迹的注意点
17. 异常实例的构造有很高的开销，为什么不去缓存异常实例呢?

18. Throwable.fillInStackTrace()会影响即使编译器(JIT)的优化吗?
19. 抛异常操作本身的性能问题
20. 异常实例的构造有很高的开销，为什么不去缓存异常实例呢?
21. JVM是如何捕获异常的?
22. 什么是异常表?
23. 什么是异常处理器？其组成部分有哪些?
24. 如果在方法的异常表中没有匹配到异常处理器，会怎么样?
25. finally代码块是如何去实现的?
26. finally代码块实例
27. try中抛出异常A，catch中出现了异常B，最终抛出的异常是哪一个?
28. 为什么catch抛出的异常B会导致原来的异常A丢失?
29. Supressed异常是什么?
30. Supressed异常的问题
31. Java7还允许在同一个catch代码中捕获多种异常
32. Java7中专门提供了try-with-reources语法糖解决该问题
33. try-with-resources的使用
34. 【☆】finally为什么一定会被执行?
    > 编译阶段进行的处理。在try、catch正常流程和异常流程的出口，复制一份finally代码块。

35. catch代码块中去捕获的自定义异常，这种异常也会出现在当前方法的异常表内吗?
    > 会

36. 方法的异常表是包含这段代码可能抛出的所有异常吗?
    > 不是，只会包含声明的需要被捕获的异常

参考资料

1. Java异常的栈轨迹(Stack Trace)
2. 深入理解Java虚拟机笔记---运行时栈帧结构