Java 笔试中，经常会考 try catch finally执行顺序和返回值的问题，大部分都只在书里面看过，说 finally 一定会执行。其背后的原因值得深究，看看try catch finally这个语法糖背后的实现原理

0x01 try catch 字节码分析

```
public class TryCatchFinallyDemo {
    public void foo() {
        try {
            tryItOut1();
        } catch (MyException1 e) {
            handleException(e);
        }
    }
}
```

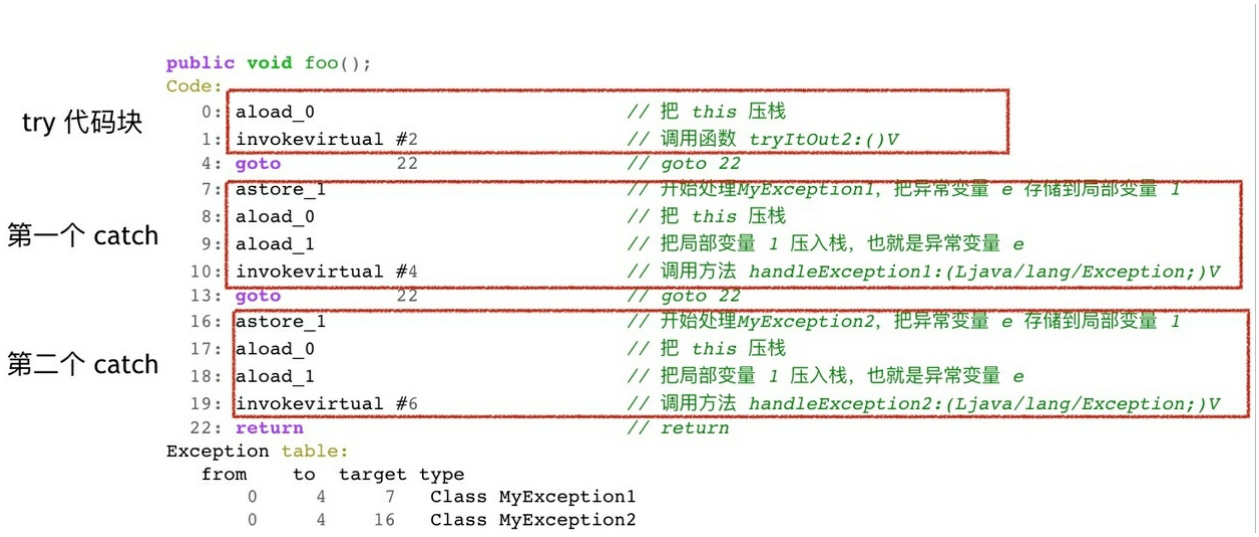
```
public void foo();
Code:
   0: aload_0                           // 加载this指针
   1: invokevirtual #2                  // 调用 tryItOut1:()V 方法
   4: goto          13                  // goto 13
   7: astore_1                          // 把异常变量 e 存储在局部变量 1
   8: aload_0                           // 把 this 指针压入栈
   9: aload_1                           // 把局部变量 1 压入栈
  10: invokevirtual #4                  // 调用方法 handleException:(Ljava/lang/Exception;)V
  13: return                            // return
Exception table:
   from    to  target type
       0     4     7   Class MyException1
```

在编译后字节码中，每个方法都附带一个异常表（Exception table），异常表里的每一行表示一个异常处理器，由 from 指针、to 指针、target 指针、所捕获的异常类型 type 组成。这些指针的值是字节码索引，用于定位字节码
其含义是在[from, to)字节码范围内，抛出了异常类型为type的异常，就会跳转到target表示的字节码处。
比如，上面的例子异常表表示：在0到4中间（不包含 4）如果抛出了MyException1的异常，就跳转到7执行。

当有多个的catch的情况下，又会是怎样?

```
public void foo() {
    try {
        tryItOut2();
    } catch (MyException1 e) {
        handleException1(e);
    } catch (MyException2 e) {
        handleException2(e);
    }
}
```

对应字节码如下：



可以看到多一个异常，会在异常表（Exception table 里面多一条记录）。

当程序出现异常时，Java 虚拟机会从上至下遍历异常表中所有的条目。当触发异常的字节码索引值在某个异常条目的[from, to)范围内，则会判断抛出的异常与该条目想捕获的异常是否匹配。

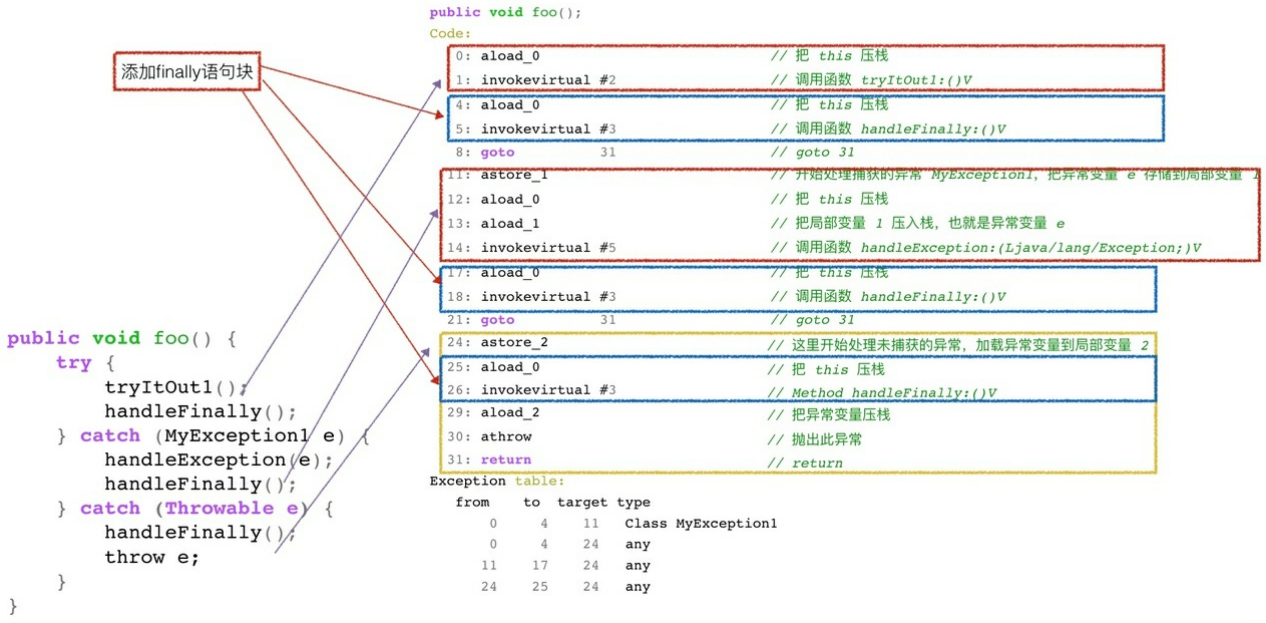
- 如果匹配，Java 虚拟机会将控制流跳转到 target 指向的字节码；如果不匹配则继续遍历异常表
- 如果遍历完所有的异常表，还未匹配到异常处理器，那么该异常将蔓延到调用方（caller）中重复上述的操作。最坏的情况下虚拟机需要遍历该线程 Java 栈上所有方法的异常表

0x02 finally 字节码分析

finally 的字节码分析最为有意思，之前我一直以为 finally 的实现是用简单的跳转来实现的，实际上并非如此。比如下面的代码

```
public void foo() {
    try {
        tryItOut1();
    } catch (MyException1 e) {
        handleException(e);
    } finally {
        handleFinally();
    }
}
```

对应的字节码如下：



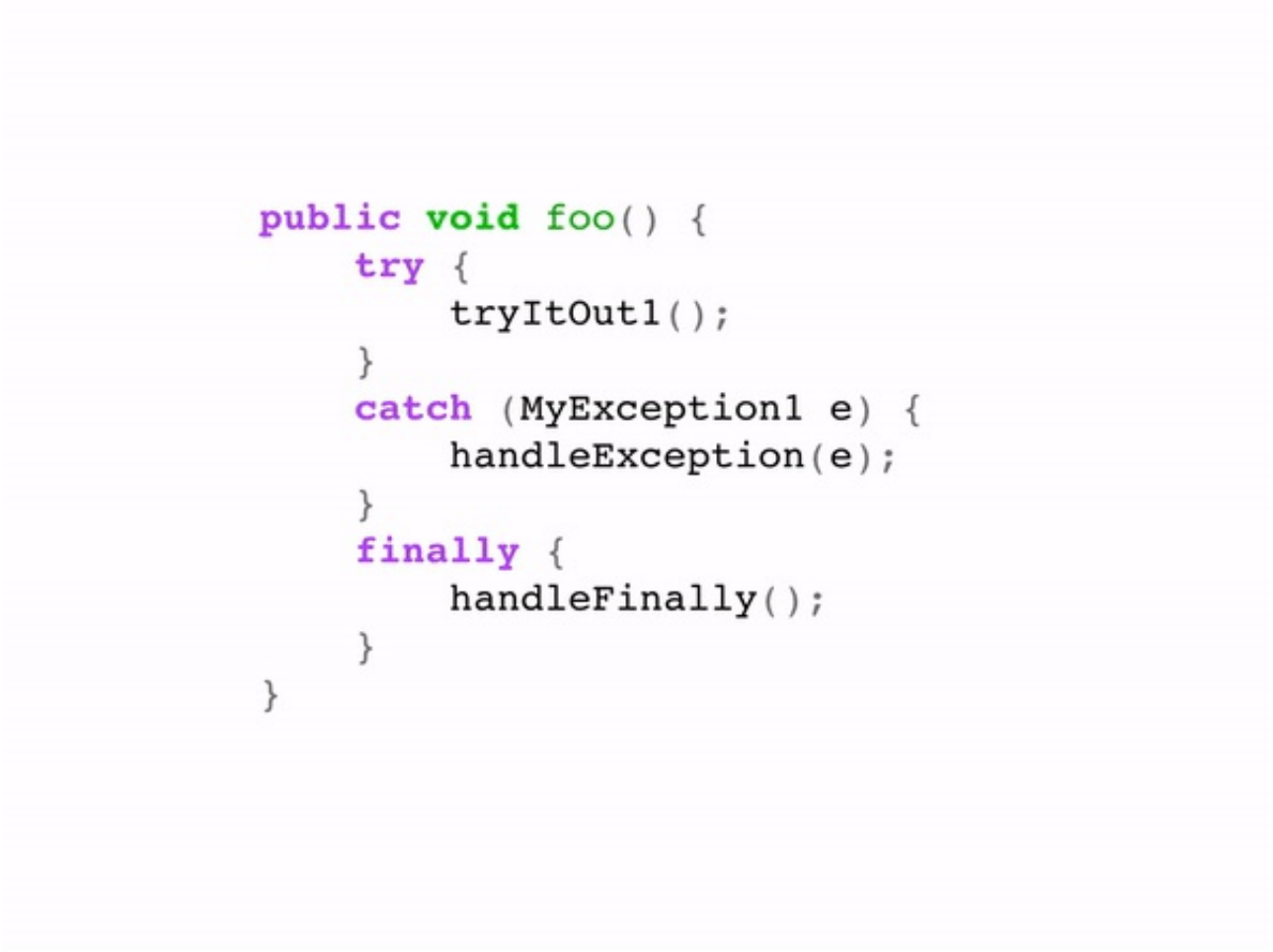
可以看到，字节码中包含了三份 finally 语句块，都在程序正常 return 和异常 throw 之前。其中两处在 try 和 catch 调用 return 之前，一处是在异常 throw 之前。

Java 采用方式是复制 finally 代码块的内容，分别放在 try catch 代码块所有正常 return 和 异常 throw 之前。

相当于如下的代码：

```
public void foo() {
    try {
        tryItOut1();
        handleFinally();
    } catch (MyException1 e) {
        handleException(e);
        handleFinally();
    } catch (Throwable e) {
        handleFinally();
        throw e;
    }
}
```

整个过程如下图所示



这样就解释了我们一直以来被灌输的观点，finally语句一定会执行

0x03 面试题解析

这里有一个笔试中特别喜欢考，但是实际用处不大的场景：在 finally 中有 return 的情况。有了上述分析，就比较简单了，如果 finally 中有 return，因为它先于其它的执行，会覆盖其它的返回（包括异常）

```
题目1:
public static int func() {
    try {
        return 0;
```

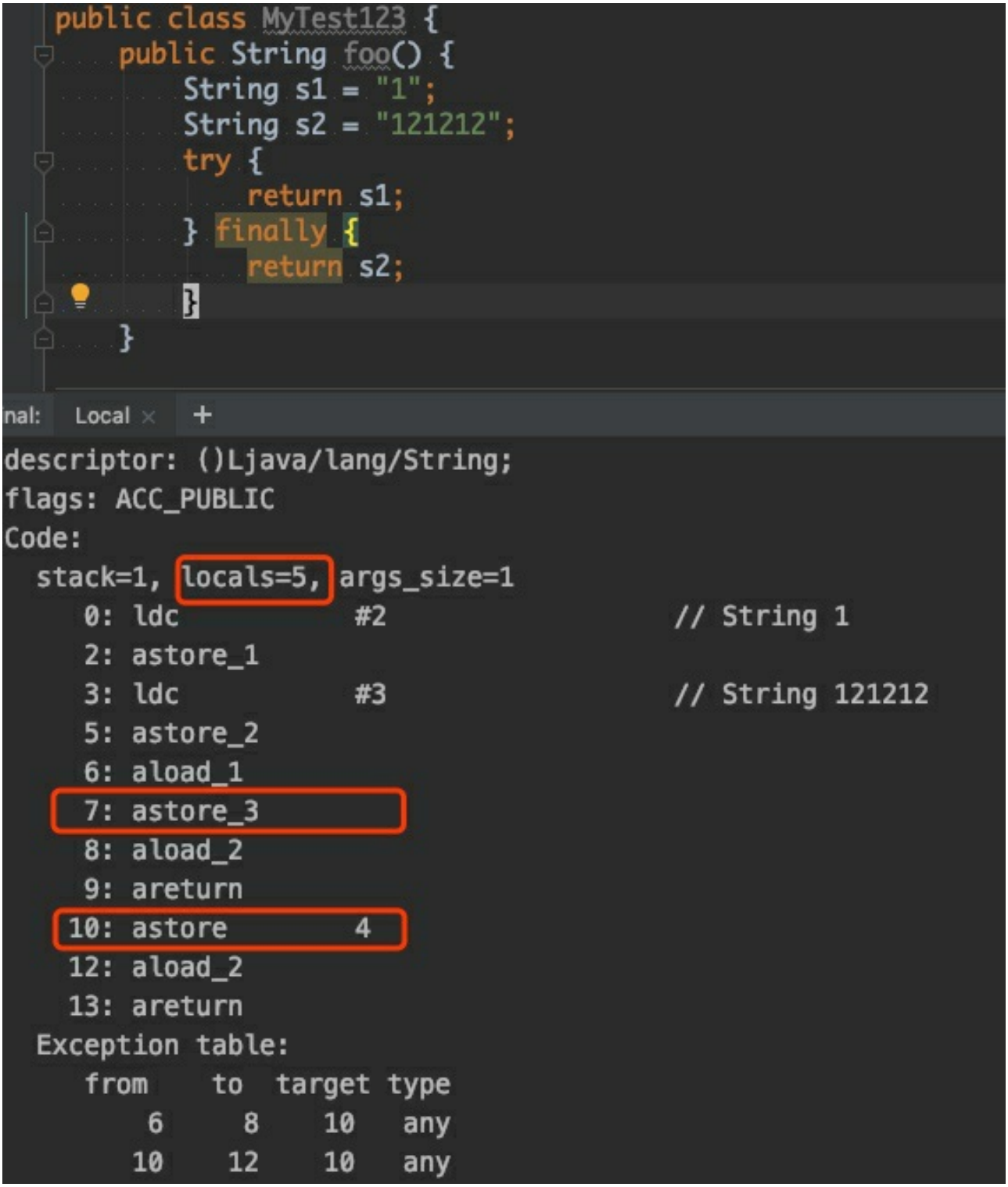
```
    } catch (Exception e){
        return 1;
    } finally {
        return 2;
    }
}
返回 2

题目2:
public static int func() {
    try {
        int a = 1 / 0;
        return 0;
    } catch (Exception e) {
        return 1;
    } finally {
        return 2;
    }
}
返回 2

题目3:
public static int func() {
    try {
        int a = 1 / 0;
        return 0;
    } catch (Exception e) {
        int b = 1 / 0;
    } finally {
        return 2;
    }
}
返回 2
```

0x04 答读者问

读者群有读者提了一个问题，我觉得可以分享一下。代码如下



```
public class MyTest123 {
    public String foo() {
        String s1 = "1";
        String s2 = "121212";
        try {
            return s1;
        } finally {
            return s2;
        }
    }
}
```

```
nal:  Local  +
descriptor: ()Ljava/lang/String;
flags: ACC_PUBLIC
Code:
  stack=1, locals=5, args_size=1
     0: ldc           #2                  // String 1
     2: astore_1
     3: ldc           #3                  // String 121212
     5: astore_2
     6: aload_1
     7: astore_3
     8: aload_2
     9: areturn
    10: astore        4
    12: aload_2
    13: areturn
  Exception table:
     from    to  target type
         6     8    10   any
        10    12    10   any
```

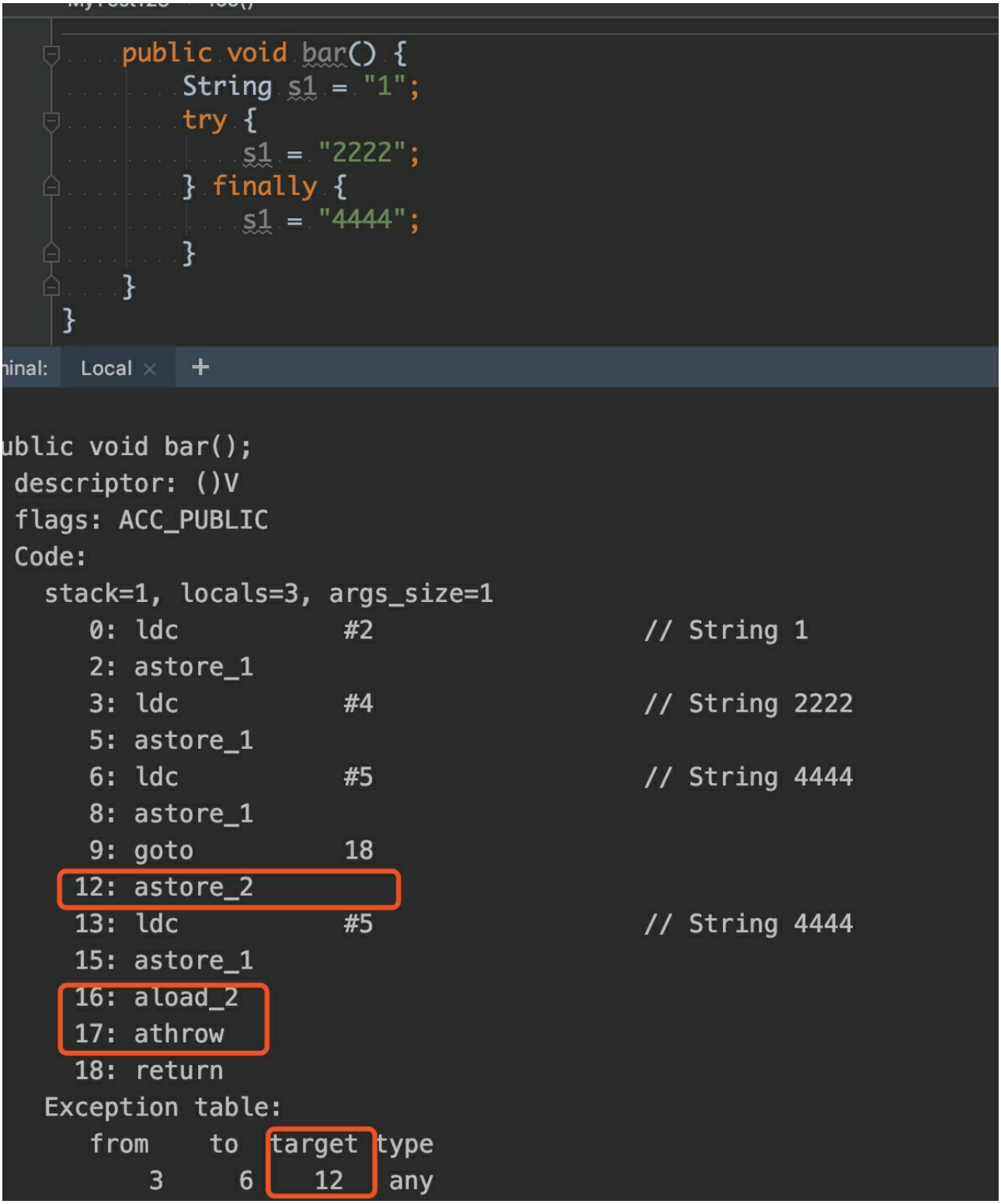
问题是：字节码中的第 10 行 astore 4 是什么意思，栈上不都是空的吗？还有为什么显示局部变量 locals 个数等于 5，不是只有

this、s1、s2 这 3 个吗?

先来看astore_3这个字节码，其实是把 s1 的引用存储到局部变量表 slot 为 3 的位置上。第 10 行的astore 4是什么呢？从异常表（Exception table）可以看到第 10 行开始是异常处理的逻辑，这个时候栈顶并非是空的，栈顶元素就是抛出的异常，astore 4将这个异常放到局部变量表中 slot 为 4 的位置。因此最后一个局部变量也清楚了。局部变量表列表如下：

- 0：this
- 1：s1
- 2：s2
- 3：tmp_s1
- 4：exception

如果上面的例子还不够清楚直接，可以再来一段代码



```
    public void bar() {
        String s1 = "1";
        try {
            s1 = "2222";
        } finally {
            s1 = "4444";
        }
    }
}
```

```
inal:  Local  +

ublic void bar();
 descriptor: ()V
 flags: ACC_PUBLIC
 Code:
   stack=1, locals=3, args_size=1
      0: ldc           #2                  // String 1
      2: astore_1
      3: ldc           #4                  // String 2222
      5: astore_1
      6: ldc           #5                  // String 4444
      8: astore_1
      9: goto          18
     12: astore_2
     13: ldc           #5                  // String 4444
     15: astore_1
     16: aload_2
     17: athrow
     18: return
   Exception table:
      from    to  target type
          3     6    12   any
```

字节码中 12 行开始是异常处理的逻辑，字节码16：aload_2 17：athrow，通过 athrow 可以知道局部变量表中位置为 2 的变量是一个异常。与上面的例子是一样的。

0x05 小结

这篇文章我们讲了 try-catch-finally 语句块的底层字节码实现，一起来回顾一下要点：

- 第一，JVM 采用异常表的方式来处理 try-catch 的跳转逻辑；
- 第二，finally 的实现采用拷贝 finally 语句块的方式来实现 finally 一定会执行的语义逻辑；
- 第三，讲解了面试喜欢考的在 finally 中有 return 语句或者 抛异常的情况。

0x06 作业题

最后，给你留两道作业题。

1、下面代码输出什么，原因是什么

```
public static int foo() {
    int x = 0;
    try {
        return x;
    } finally {
        ++x;
    }
}

public static void main(String[] args) {
    int res = foo();
    System.out.println(res);
}
```

2、低版本的 JDK 采用 jsr/ret 来实现 finally 语句块，你可以去了解一下这两个指令的作用，实现一下 finally 语义吗?

欢迎你在留言区留言，和我一起讨论。