几个关于Class文件的问题分析

> 日常工作中，我们直接接触Class文件的时间可能不多，但这不代表了解了Class文件就用处不大。本文将试图回答三个问题，*Class文件中字符串的最大长度是多少*、*Java存在尾递归调用优化吗？*、*类的初始化顺序是怎样的？*。与直接给出答案不同，我们试图从Class文件中找出这个答案背后的道理。我们一一来看一下。

Class文件中字符串的最大长度是多少？

在上篇文章中我们提到，在class文件中，字符串是被存储在常量池中，更进一步来讲，它使用一种UTF-8格式的变体来存储一个常量字符，其存储结构如下：

```
1.  CONSTANT_Utf8_info {
2.      u1 tag;//值为CONSTANT_Utf8_info（1）
3.      u2 length;//字节的长度
4.      u1 bytes[length]//内容
5.  }
```

可以看到CONSTANT_Utf8_info中使用了u2类型来表示长度，当我最开始接触到这里的时候，就在想一个问题，如果我声明了一个超过u2长度（65536）的字符串，是不是就无法编译了。我们来做个实现。

字符串太长就不贴出来，直接贴出在终端上使用javac命令编译后的结果：

```
⇒  javac LongString.java
LongString.java:5: 错误: 对于常量池来说，字符串 "I spend a lot of tim..." 的 UTF8 表示过长
public class LongString {
       ^
1 个错误
```

果然，编译报错了，看来class文件的确无法存储超过65536字节的字符串。

如果事情到这里为止，并没有太大意思了，但后来我发现了一个有趣的事情。下面的这段代码在eclipse中是可以编译过的：

```
1.  public class LongString {
2.      public static void main(String[] args){
3.        String s = a long long string...;
4.        System.out.println(s);
5.      }
6.  }
```

这不科学，更不符合我们的认知。eclipse搞了什么名堂？我们拖出class文件看一看：

```
1.    public static void main(java.lang.String[]);
2.      descriptor: ([Ljava/lang/String;)V
3.      flags: ACC_PUBLIC, ACC_STATIC
4.      Code:
5.        stack=3, locals=2, args_size=1
6.           0: new           #16                 // class java/lang/StringBuilder
7.           3: dup
8.           4: ldc           #18
9.           6: invokespecial #20                 // Method java/lang/StringBuilder."<init>":(Ljava/lang/String;)V
10.          9: ldc           #23                 // String
11.         11: invokevirtual #25                 // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/l
12.         14: invokevirtual #29                 // Method java/lang/StringBuilder.toString:()Ljava/lang/String;
```

```
          17: invokevirtual #33                 // Method java/lang/String.intern:()Ljava/lang/String;
          20: astore_1
          21: getstatic     #38                 // Field java/lang/System.out:Ljava/io/PrintStream;
          24: aload_1
          25: invokevirtual #44                 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
          28: return
        LineNumberTable:
          line 10: 0
          line 3212: 21
          line 3213: 28
        LocalVariableTable:
          Start  Length  Slot  Name   Signature
              0      29     0  args   [Ljava/lang/String;
             21       8     1   STR   Ljava/lang/String;
```

可以看到，上面的超长字符串被eclipse截成两半，#18和#23， 然后通过StringBuilder拼接成完整的字符串。awesome！

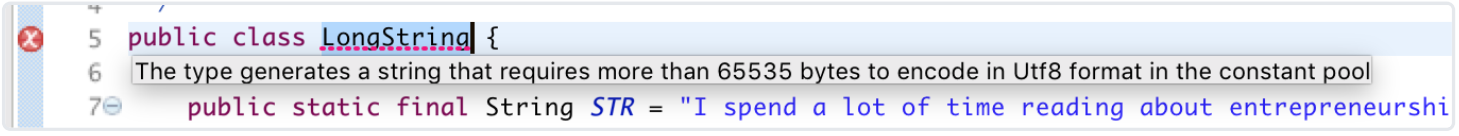
但是，如果我们不是在函数中声明了一个巨长的字符串，而是在类中直接声明：

```
public class LongString {

    public static final String STR = a long long string...;

}
```

Eclipse会直接进行错误提示：



具体关于在上面两个字符串的初始化时机我们会在第三点里进行阐述，但理论上在类中直接声明也是可以像在普通函数中一样进行优化。具体的原因我们就不得而知了。不过这提醒我们的是在Class文件中，和字符串长度类似的还有类中继承接口的个数、方法数、字段数等等，它们都是存在个数由上限的。

Java存在尾递归调用优化吗？

回答这个问题之前，我们需要了解什么是尾递归呢？借用维基百科中的回答：

> - 调用自身函数(Self-called)；
> - 计算仅占用常量栈空间(Stack Space)

用更容易理解的话来讲，尾递归调用就是函数最后的语句是调用自身，但调用自己的时候，已经不再需要上一个函数的环境了。所以并非所有的递归都属于尾递归，它需要通过上述的规则来编写递归代码。和普通的递归相比，尾递归即使递归调用数万次，它的函数栈也仅为常数，不会出现`Stack Overflow`异常。

那么java中存在尾递归优化吗？这个回答现在是否定的，到目前的Java8为止，Java仍然是不支持尾递归的。

但最近class家族的一位成员`kotlin`是号称支持尾递归调用的，那么它是怎么实现的呢？我们通过递归实现一个功能来对比`Java`与`Kotlin`之间生成的字节码的差别。

我们来实现一个对两个整数的开区间内所有整数求和的功能。函数声明如下：

```
int sum(int start, int end , int acc)
```

参数start为起始值，参数end为结束值，参数acc为累加值（调用时传入0，用于递归使用）。如sum(2,4,0)会返回9。我们分别用`Java`与`Kotlin`来实现这个函数。

Java:

```java
public static int sum(int start, int end , int acc){
    if(start > end){
       return acc;
    }else{
       return sum(start + 1, end, start + acc);
    }
}
```

Koklin:

```kotlin
tailrec fun sum(start: Int, end: Int, acc: Int): Int{
    if (start > end){
        return acc
    } else{
        return  sum(start+1, end, start + acc)
    }
}
```

我们对这两个文件编译生成的class文件中的sum函数进行分析：

Java生成的sum函数字节码如下：

```
public static int sum(int, int, int);
  descriptor: (III)I
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=4, locals=3, args_size=3
       0: iload_0
       1: iload_1
       2: if_icmple     7
       5: iload_2
       6: ireturn
       7: iload_0
       8: iconst_1
       9: iadd
      10: iload_1
      11: iload_0
      12: iload_2
      13: iadd
      14: invokestatic  #5                  // Method sum:(III)I
      17: ireturn
    LineNumberTable:
      line 8: 0
      line 9: 5
      line 11: 7
    StackMapTable: number_of_entries = 1
      frame_type = 7 /* same */
```

我们提取主要信息，在第14个命令上，sum函数又递归的调用了sum函数自己。此时，还没有调用到第17条命令ireturn来退出函数，所以，函数栈会进行累加，如果递归次数过多，就难免不会发生`Stack Overflow`异常了。

我们再来看一下`Kotlin`中sum函数的字节码是怎样的：

```
public static final int sum(int, int, int);
  descriptor: (III)I
  flags: ACC_PUBLIC, ACC_STATIC, ACC_FINAL
  Code:
    stack=4, locals=3, args_size=3
       0: iload_0
       1: iload_1
       2: if_icmple     7
       5: iload_2
       6: ireturn
       7: iload_0
       8: iconst_1
       9: iadd
      10: iload_1
      11: iload_0
      12: iload_2
      13: iadd
      14: istore_2
      15: pop
      16: istore_0
      17: goto          0
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0      20     0 start   I
          0      20     1   end   I
          0      20     2 result   I
    LineNumberTable:
      line 2: 0
      line 3: 5
      line 5: 7
    StackMapTable: number_of_entries = 2
      frame_type = 0 /* same */
      frame_type = 6 /* same */
```

可以看到，在上面的sum函数中并没有存在对sum自身的调用，而取而代之的是，是第17条的goto命令。所以，`Kotlin`尾递归背后的黑魔法就是将递归改成循环结构。上面的代码翻译成我们容易理解的代码就是如下形式：

```
1.  public int sum(int start, int end , int acc){
2.      for(;;){
3.          if(start > end){
4.              return acc;
5.          }else{
6.              acc = start + acc;
7.              start = start + 1;
8.          }
9.      }
10.  }
```

通过上述的分析我们可以看到，递归是通过转化为循环来降低内存的占用。但这并不意味着写递归就是很差的编程习惯。在`Java`这种面向对象的语言中我们更倾向于将递归改成循环，而在`Haskell`这类函数式编程语言中是将循环都改为了递归。在思想上并没有优劣之分，只是解决问题的思维上的差异而已，具体表现就是落实到具体语言上对这两种方法的支持程度不同而已（Java没有尾递归，Haskell没有for、while循环）。

类的初始化顺序是怎样的？

这个问题对于正在找工作的人可能比较有感觉，起码当时我在毕业准备面试题时就遇到了这个问题，并且也机械的记忆了答案。不过我们更期待的是这个答案背后的理论依据是什么。我们尝试从class文件中找到答案。来看这样的一段代码：

```
1.  public class InitialOrderTest {
2.
3.      public static String staticField = "   StaticField";
4.
5.      public String fieldFromMethod = getStrFromMethod();
6.
7.      public String fieldFromInit = "   InitField";
8.
```

```
    static {
        System.out.println( "Call Init Static Code" );
        System.out.println( staticField );
    }

    {
        System.out.println( "Call Init Block Code" );
        System.out.println( fieldFromInit );
        System.out.println( fieldFromMethod );
    }

    public InitialOrderTest()
    {
        System.out.println( "Call Constructor" );
    }

    public String getStrFromMethod(){
        System.out.println("Call getStrFromMethod Method");
        return "   MethodField" ;
    }

    public static void main( String[] args )
    {
        new InitialOrderTest();
    }
}

```

它运行后的结果是什么呢？结果如下：

```
⇒  java InitialOrderTest
Call Init Static Code
   StaticField
Call getStrFromMethod Method
Call Init Block Code
   InitField
   MethodField
Call Constructor
```

我们来一一来看一下它的class文件中的内容，首先是有一个static方法区：

```
static {};
    descriptor: ()V
    flags: ACC_STATIC
    Code:
      stack=2, locals=0, args_size=0
         0: ldc           #14                 // String    StaticField
         2: putstatic     #15                 // Field staticField:Ljava/lang/String;
         5: getstatic     #6                  // Field java/lang/System.out:Ljava/io/PrintStream;
         8: ldc           #16                 // String Call Init Static Code
        10: invokevirtual #8                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
        13: getstatic     #6                  // Field java/lang/System.out:Ljava/io/PrintStream;
        16: getstatic     #15                 // Field staticField:Ljava/lang/String;
        19: invokevirtual #8                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
```

```
14.          22: return
```

Java编译器在编译阶段会将所有static的代码块收集到一起，形成一个特殊的方法，这个方法的名字叫做`<clinit>`,这个名字容易让我们联想到构造函数的名称叫做`<init>`，但与构造函数不同，这个方法在Java层中是调用不到的，并且，这个函数是在这个类被加载时，由虚拟机进行调用。注意的是，是类被加载，而不是类被初始化成实例。所以，静态代码块的加载优先于普通的代码块，也优先于构造函数。这属于虚拟机规定的范畴，我们不做更深入的探讨。

在Class文件中，是没有为普通方法区开辟类似于`<clinit>`这种方法的，而是将所有普通方法区的代码都合并到了构造函数中，我们直接来看构造函数：

```
1.   public InitialOrderTest();
2.       descriptor: ()V
3.       flags: ACC_PUBLIC
4.       Code:
5.         stack=2, locals=1, args_size=1
6.            0: aload_0
7.            1: invokespecial #1                  // Method java/lang/Object."<init>":()V
8.            4: aload_0
9.            5: aload_0
10.           6: invokevirtual #2                  // Method getStr:()Ljava/lang/String;
11.           9: putfield      #3                  // Field field:Ljava/lang/String;
12.          12: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
13.          15: aload_0
14.          16: getfield      #3                  // Field field:Ljava/lang/String;
15.          19: invokevirtual #5                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
16.          22: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
17.          25: ldc           #6                  // String Init Block
18.          27: invokevirtual #5                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
19.          30: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
20.          33: ldc           #7                  // String Constructor
21.          35: invokevirtual #5                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
22.          38: return
```

通过分析构造函数，我们就可以对一个实例初始化的顺序一清二楚，首先，0，1在构造函数中调用了父类的构造函数，接着，4、5、6、9为成员变量进行赋值，25、27在执行实例的代码块，最后，33、35才是执行我们Java文件中编写的构造函数的代码。这样，一个普通类的初始化顺序大致如下：

> 静态代码按照顺序初始化 -> 父类构造函数 -> 变量初始化 -> 实例代码块 -> 自身构造函数