深入理解JVM实战篇-String类

- 深入理解JVM实战篇-String类
  - Question
  - 分析
    - 字面量和运行时常量
    - new String() 创建了几个对象?
    - 运行时常量池的动态扩展
  - 总结
  - 参考

Question

Q1：`String str = new String("hello");`定义了几个对象?

Q2: 如何理解String的intern方法?

A1：若常量池中已经存在"hello"，则直接引用，也就是此时只会创建一个对象，如果常量池中不存在"hello"，则先创建后引用，也就是有两个。

A2：当一个String实例调 用`intern()`方法时，Java查找常量池中是否有相同`Unicode`的字符串常量，如果有，则返回其的引用，如果没有，则在常量池中增加一个`Unicode`等于String实例的字符串并返回它的引用;

两个答案看上去没有任何问题，但是，仔细想想好像哪里不对呀。按照上面的两个面试题的回答，就是说new String也会检查常量池，如果有的话就直接引用，如果不存在就要在常量池创建一个，那么还要intern干啥？难道以下代码是没有意义的吗?

```
String s = new String("Hello").intern();
```

分析

思考以下代码运行结果：

```
String s1 = "Hello";
String s2 = new String("Hello");
String s3 = new String("Hello").intern();

System.out.println(s1 == s2);
System.out.println(s1 == s3);
```

运行结果：

```
false  true
```

字面量和运行时常量

`JVM`为了提高性能和减少内存开销，在实例化字符串常量的时候进行了一些优化。为了减少在`JVM`中创建的字符串的数量，字符串类维护了一个字符串常量池。

在`JVM`运行时区域的方法区中，有一块区域是运行时常量池，主要用来存储编译期生成的各种字面量和符号引用。

在`java`代码被`javac`编译之后，文件结构中是包含一部分`Constant pool`的。比如以下代码：

```
public class TestString {

    public static void main(String[] args){
        String str = "Hello";

        System.out.println(" str = "+str);
    }
}
```

先对上述代码 使用 `javac` 编译成 `.class` 文件 在使用 `javap` 命令进行将 `.class` 文件反编译会得到以下信息。

```
javap -v TestString.class
```

反编译出以下信息：

```
Classfile /Users/xingzhezuomeng/codes/idea_codes/build/TestString.class
  Last modified Dec 12, 2019; size 601 bytes
  MD5 checksum 6e3eeb075e2a2a9effa6920f7997b3dc
  Compiled from "TestString.java"
public class TestString
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:  // 常量池
   #1 = Methodref          #11.#20        // java/lang/Object."<init>":()V
   #2 = String             #21            // Hello
   #3 = Fieldref           #22.#23        //
java/lang/System.out:Ljava/io/PrintStream;
   #4 = Class              #24            // java/lang/StringBuilder
   #5 = Methodref          #4.#20         // java/lang/StringBuilder."
<init>":()V
   #6 = String             #25            //  str =
   #7 = Methodref          #4.#26         //
java/lang/StringBuilder.append:
(Ljava/lang/String;)Ljava/lang/StringBuilder;
   #8 = Methodref          #4.#27         //
java/lang/StringBuilder.toString:()Ljava/lang/String;
   #9 = Methodref          #28.#29        // java/io/PrintStream.println:
(Ljava/lang/String;)V
  #10 = Class              #30            // TestString
  #11 = Class              #31            // java/lang/Object
  #12 = Utf8               <init>
  #13 = Utf8               ()V
  #14 = Utf8               Code
  #15 = Utf8               LineNumberTable
  #16 = Utf8               main
  #17 = Utf8               ([Ljava/lang/String;)V
  #18 = Utf8               SourceFile
  #19 = Utf8               TestString.java
  #20 = NameAndType        #12:#13        // "<init>":()V
  #21 = Utf8               Hello     // 字面量
  #22 = Class              #32            // java/lang/System
  #23 = NameAndType        #33:#34        // out:Ljava/io/PrintStream;
  #24 = Utf8               java/lang/StringBuilder
  #25 = Utf8                str =
  #26 = NameAndType        #35:#36        // append:
(Ljava/lang/String;)Ljava/lang/StringBuilder;
  #27 = NameAndType        #37:#38        // toString:()Ljava/lang/String;
  #28 = Class              #39            // java/io/PrintStream
  #29 = NameAndType        #40:#41        // println:(Ljava/lang/String;)V
```

```
  #30 = Utf8               TestString
  #31 = Utf8               java/lang/Object

  #32 = Utf8               java/lang/System
  #33 = Utf8               out
  #34 = Utf8               Ljava/io/PrintStream;
  #35 = Utf8               append
  #36 = Utf8               (Ljava/lang/String;)Ljava/lang/StringBuilder;
  #37 = Utf8               toString
  #38 = Utf8               ()Ljava/lang/String;
  #39 = Utf8               java/io/PrintStream
  #40 = Utf8               println
  #41 = Utf8               (Ljava/lang/String;)V
{
  public TestString();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."
<init>":()V
         4: return
      LineNumberTable:
        line 1: 0

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=3, locals=2, args_size=1
         0: ldc           #2                  // String Hello  将常量
Hello压入栈
         2: astore_1
         3: getstatic     #3                  // Field
java/lang/System.out:Ljava/io/PrintStream;
         6: new           #4                  // class
java/lang/StringBuilder
         9: dup
        10: invokespecial #5                  // Method
java/lang/StringBuilder."<init>":()V
        13: ldc           #6                  // String  str =
        15: invokevirtual #7                  // Method
java/lang/StringBuilder.append:
(Ljava/lang/String;)Ljava/lang/StringBuilder;
        18: aload_1
        19: invokevirtual #7                  // Method
java/lang/StringBuilder.append:
(Ljava/lang/String;)Ljava/lang/StringBuilder;
```

```
        22: invokevirtual #8                  // Method
java/lang/StringBuilder.toString:()Ljava/lang/String;
        25: invokevirtual #9                  // Method
java/io/PrintStream.println:(Ljava/lang/String;)V
        28: return
      LineNumberTable:
        line 4: 0
        line 6: 3
        line 7: 28
}
SourceFile: "TestString.java"
```

上面代码里发现以下重要信息：

```
#21 = Utf8               Hello
```

在编译期间 将 Hello 字面量已经加入了字符串常量池中。

所以我们利用反编译的方式证明了 字符串在编译期间会加入到 字符串常量池中。

new String() 创建了几个对象?

我们下面接着分析下面代码：

```
String s = new String("Hello");
```

这段代码中，我们可以知道的是，在编译期，字面量Hollis会被加入到Class文件的常量池中。

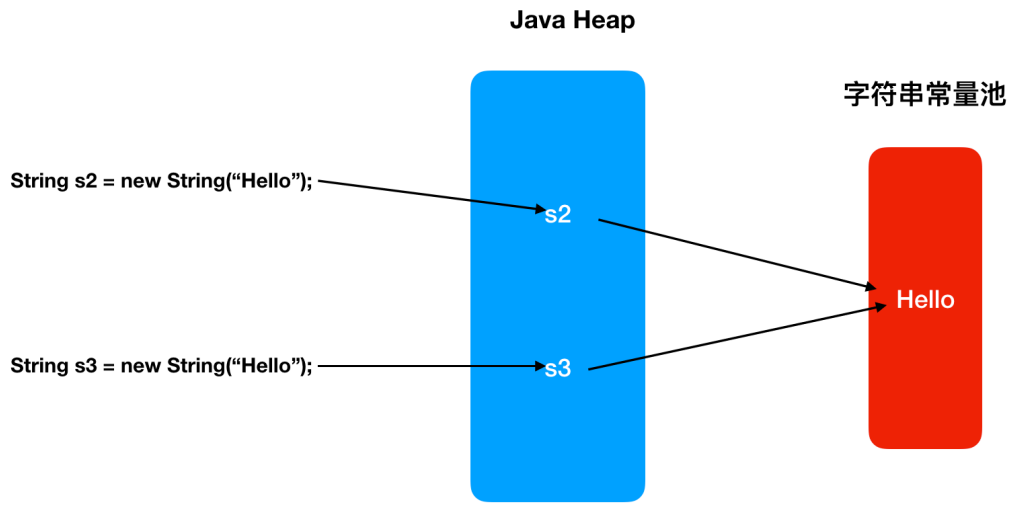
但是，这个“进入”阶段，并不会直接把所有类中定义的常量全部都加载进来，而是会做个比较，如果需要加到字符串常量池中的字符串已经存在，那么就不需要再把字符串字面量加载进来了。

所以，当我们说【若常量池中已经存在"hello"，则直接引用，也就是此时只会创建一个对象】说的就是这个字符串字面量在字符串池中被创建的过程。

在运行期，`new String("Hello");` 执行到的时候，是要在`Java`堆中创建一个字符串对象的，而这个对象所对应的字符串字面量是保存在字符串常量池中的。但是，`String s = new String("Hello");`，对象的符号引用`s`是保存在`Java`虚拟机栈上的，他保存的是堆中刚刚创建出来的的字符串对象的引用。

```
String s2 = new String("Hello");
String s3 = new String("Hello");
```

因为，==比较的是s1和s2在堆中创建的对象的地址，当然不同了。但是如果使用equals，那么比较的就是字面量的内容了，那就会得到true。



常量池中的“对象”是在编译期就确定好了的，在类被加载的时候创建的，如果类加载时，该字符串常量在常量池中已经有了，那这一步就省略了。堆中的对象是在运行期才确定的，在代码执行到new的时候创建的。

运行时常量池的动态扩展

编译期生成的各种字面量和符号引用是运行时常量池中比较重要的一部分来源，但是并不是全部。那么还有一种情况，可以在运行期像运行时常量池中增加常量。那就是String的 `intern` 方法。

intern()有两个作用:

- 第一个是将字符串字面量放入常量池（如果池没有的话）
- 第二个是返回这个常量的引用。

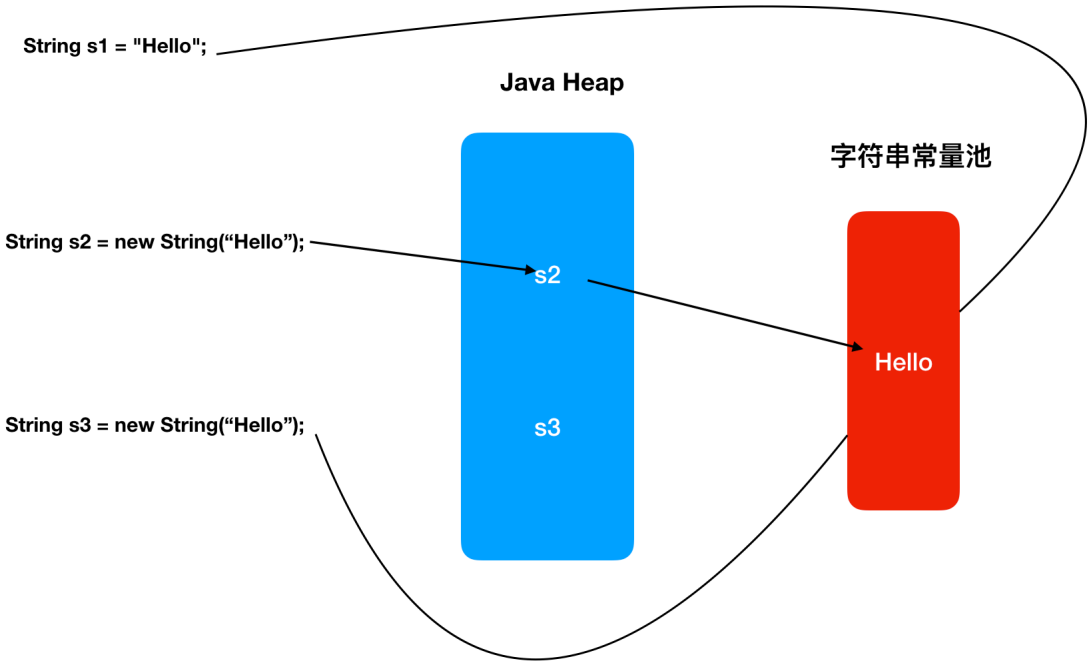
再次讨论开头的例子：

```
String s1 = "Hello";
String s2 = new String("Hello");
String s3 = new String("Hello").intern();

System.out.println(s1 == s2);
System.out.println(s1 == s3);
```

可以简单的理解为String s1 = "Hollo";和 String s3 = new String("Hollis").intern();做的事情是一样的,都是定义一个字符串对象，然后将其字符串字面量保存在常量池中，并把这个字面量的引用返回给定义好的对象引用。

如下图所示：



对于`String s3 = new String("Hollo").intern();`，在不调用`intern`情况，s3指向的是JVM在堆中创建的那个对象的引用的（如图中的s3）。但是当执行了`intern`方法时，s3将指向字符串常量池中的那个字符串常量。

由于s1和s3都是字符串常量池中的字面量的引用，所以s1==s3。但是，s2的引用是堆中的对象，所以s2!=s1。

总结

我们再回到文章开头那个问题：按照上面的两个面试题的回答，就是说`new String()`也会检查常量池，如果有的话就直接引用，如果不存在就要在常量池创建一个，那么还要`intern`干啥？难道以下代码是没有意义的吗？

```
String s = new String("Hollis").intern();
```

而`intern`中说的“如果有的话就直接返回其引用”，指的是会把字面量对象的引用直接返回给定义的对象。这个过程是不会在`Java`堆中再创建一个String对象的。

的确，以上代码的写法其实是使用`intern`是没什么意义的。因为字面量`Hollo`会作为编译期常量被加载到运行时常量池。

参考

Oracle JVM 指令说明

https://docs.oracle.com/javase/specs/

https://www.iteye.com/blog/icyfenix-1256329

https://www.jianshu.com/p/26f95965320e

https://www.jianshu.com/p/6a8997560b05

https://stackoverflow.com/questions/5546280/understanding-javaps-output-for-the-constant-pool

https://juejin.im/post/5c5e604b6fb9a04a102fbe01

https://docs.oracle.com/javase/specs/jls/se8/html/jls-3.html#jls-3.10.5

https://www.zhihu.com/question/55994121