[**请别再拿“String s = new String("xyz");创建了多少个String实例”来面试了吧**](http://rednaxelafx.iteye.com/blog/774673)

**博客分类：**

* [Java](http://rednaxelafx.iteye.com/category/22617)
* [Virtual Machine](http://rednaxelafx.iteye.com/category/61298)
* [HotSpot VM](http://rednaxelafx.iteye.com/category/162318)

[面试](http://www.iteye.com/blogs/tag/%E9%9D%A2%E8%AF%95)[Java](http://www.iteye.com/blogs/tag/Java)[JVM](http://www.iteye.com/blogs/tag/JVM)[虚拟机](http://www.iteye.com/blogs/tag/%E8%99%9A%E6%8B%9F%E6%9C%BA)[SUN](http://www.iteye.com/blogs/tag/SUN)

这帖是用来回复[高级语言虚拟机](http://hllvm.group.iteye.com/)圈子里的一个问题，[一道Java笔试题](http://hllvm.group.iteye.com/group/topic/21761)的。   
本来因为见得太多已经吐槽无力，但这次实在忍不住了就又爆发了一把。写得太长干脆单独开了一帖。   
  
顺带广告：对JVM感兴趣的同学们同志们请多多支持[高级语言虚拟机](http://hllvm.group.iteye.com/)圈子 http://rednaxelafx.iteye.com/images/smiles/icon_lol.gif   
  
以下是回复内容。文中的“楼主”是针对原问题帖而言。   
  
===============================================================   
  
楼主是看各种宝典了么……以后我面试人的时候就要专找宝典答案是错的来问，方便筛人orz   
  
楼主要注意了：这题或类似的题虽然经常见，但使用这个描述方式实际上没有任何意义：

**引用**

问题：

**Java代码  [收藏代码](javascript:void())**

1. String s = **new** String("xyz");

创建了几个String Object？

这个问题自身就没有合理的答案，楼主所引用的“标准答案”自然也就不准确了：

**引用**

答案：两个(一个是“xyz”,一个是指向“xyz”的引用对象s)

（好吧这个答案的吐槽点很多……大家慢慢来）   
  
这问题的毛病是什么呢？它并没有定义“创建了”的意义。   
什么叫“创建了”？什么时候创建了什么？   
而且这段Java代码片段实际运行的时候真的会“创建两个String实例”么？   
  
如果这道是面试题，那么可以当面让面试官澄清“创建了”的定义，然后再对应的回答。这种时候面试官多半会让被面试者自己解释，那就好办了，好好晒给面试官看。   
  
如果是笔试题就没有提问要求澄清的机会了。不过会出这种题目的地方水平多半也不怎么样。说不定出题的人就是从各种宝典上把题抄来的，那就按照宝典把那不大对劲的答案写上去就能混过去了http://rednaxelafx.iteye.com/images/smiles/icon_lol.gif   
  
===============================================================   
  
先换成另一个问题来问：

**引用**

问题：

**Java代码  [收藏代码](javascript:void())**

1. String s = **new** String("xyz");

在运行时涉及几个String实例？

一种合理的解答是：

**引用**

答案：两个，一个是字符串字面量"xyz"所对应的、驻留（intern）在一个全局共享的字符串常量池中的实例，另一个是通过new String(String)创建并初始化的、内容与"xyz"相同的实例

这是根据Java语言规范相关规定可以给出的合理答案。考虑到Java语言规范中明确指出了：

**The Java Language Specification, Third Edition 写道**

The Java programming language is normally compiled to the bytecoded instruction set and binary format defined in *The Java Virtual Machine Specification, Second Edition* (Addison-Wesley, 1999).

也就是规定了Java语言一般是编译为Java虚拟机规范所定义的Class文件，但并没有规定“一定”（must），留有不使用JVM来实现Java语言的余地。   
考虑上Java虚拟机规范，确实在这段代码里涉及的常量种类为[CONSTANT\_String\_info](http://java.sun.com/docs/books/jvms/second_edition/html/ClassFile.doc.html#29297)的字符串常量也只有"xyz"一个。CONSTANT\_String\_info是用来表示Java语言中String类型的常量表达式的值（包括字符串字面量）用的常量种类，只在这个层面上考虑的话，这个解答也没问题。   
所以这种解答可以认为是合理的。   
  
值得注意的是问题中“在运行时”既包括类加载阶段，也包括这个代码片段自身执行的时候。下文会再讨论这个细节与楼主原本给出的问题的关系。   
  
碰到这种问题首先应该想到去查阅相关的规范，这里具体是[Java语言规范](http://java.sun.com/docs/books/jls/)与[Java虚拟机规范](http://java.sun.com/docs/books/jvms/)，以及一些相关API的JavaDoc。很多人喜欢把“按道理说”当作口头禅，规范就是用来定义各种“道理”的——“为什么XXX是YYY的意思？”“因为规范里是这样定义的！”——无敌了。   
  
在Java虚拟机规范中相关的定义有下面一些：

**The Java Virtual Machine Specification, Second Edition 写道**

**2.3 Literals**   
  
A literal is the source code representation of a value of a primitive type [(§2.4.1)](http://java.sun.com/docs/books/jvms/second_edition/html/Concepts.doc.html#19511), the String type [(§2.4.8)](http://java.sun.com/docs/books/jvms/second_edition/html/Concepts.doc.html#25486), or the null type [(§2.4)](http://java.sun.com/docs/books/jvms/second_edition/html/Concepts.doc.html#22930). String literals and, more generally, strings that are the values of constant expressions are "interned" so as to share unique instances, using the method String.intern.   
  
The null type has one value, the null reference, denoted by the literal null. The boolean type has two values, denoted by the literals true and false.   
  
**2.4.8 The Class String**   
  
Instances of class String represent sequences of Unicode characters [(§2.1)](http://java.sun.com/docs/books/jvms/second_edition/html/Concepts.doc.html#25310). A String object has a constant, unchanging value. String literals [(§2.3)](http://java.sun.com/docs/books/jvms/second_edition/html/Concepts.doc.html#20359) are references to instances of class String.   
  
**2.17.6 Creation of New Class Instances**   
  
A new class instance is explicitly created when one of the following situations occurs:

* Evaluation of a class instance creation expression creates a new instance of the class whose name appears in the expression.
* Invocation of the newInstance method of class Class creates a new instance of the class represented by the Class object for which the method was invoked.

A new class instance may be implicitly created in the following situations:

* Loading of a class or interface that contains a String literal may create a new String object [(§2.4.8)](http://java.sun.com/docs/books/jvms/second_edition/html/Concepts.doc.html#25486) to represent that literal. This may not occur if the a String object has already been created to represent a previous occurrence of that literal, or if the String.intern method has been invoked on a String object representing the same string as the literal.
* Execution of a string concatenation operator that is not part of a constant expression sometimes creates a new String object to represent the result. String concatenation operators may also create temporary wrapper objects for a value of a primitive type [(§2.4.1)](http://java.sun.com/docs/books/jvms/second_edition/html/Concepts.doc.html#19511).

Each of these situations identifies a particular constructor to be called with specified arguments (possibly none) as part of the class instance creation process.   
  
**5.1 The Runtime Constant Pool**   
  
...   
  
● A string literal [(§2.3)](http://java.sun.com/docs/books/jvms/second_edition/html/Concepts.doc.html#20359) is derived from a CONSTANT\_String\_info structure [(§4.4.3)](http://java.sun.com/docs/books/jvms/second_edition/html/ClassFile.doc.html#29297) in the binary representation of a class or interface. The CONSTANT\_String\_info structure gives the sequence of Unicode characters constituting the string literal.   
  
● The Java programming language requires that identical string literals (that is, literals that contain the same sequence of characters) must refer to the same instance of class String. In addition, if the method String.intern is called on any string, the result is a reference to the same class instance that would be returned if that string appeared as a literal. Thus,

**Java代码  [收藏代码](javascript:void())**

1. ("a" + "b" + "c").intern() == "abc"

must have the value true.   
  
● To derive a string literal, the Java virtual machine examines the sequence of characters given by the CONSTANT\_String\_info structure.   
  
  ○ If the method String.intern has previously been called on an instance of class String containing a sequence of Unicode characters identical to that given by the CONSTANT\_String\_info structure, then the result of string literal derivation is a reference to that same instance of class String.   
  
  ○ Otherwise, a new instance of class String is created containing the sequence of Unicode characters given by the CONSTANT\_String\_info structure; that class instance is the result of string literal derivation. Finally, the intern method of the new String instance is invoked.   
  
...   
  
The remaining structures in the constant\_pool table of the binary representation of a class or interface, the CONSTANT\_NameAndType\_info [(§4.4.6)](http://java.sun.com/docs/books/jvms/second_edition/html/ClassFile.doc.html#1327) and CONSTANT\_Utf8\_info [(§4.4.7)](http://java.sun.com/docs/books/jvms/second_edition/html/ClassFile.doc.html#7963) structures are only used indirectly when deriving symbolic references to classes, interfaces, methods, and fields, and when deriving string literals.

把Sun的JDK看作参考实现（reference implementation, RI），其中[String.intern()的JavaDoc](http://download.oracle.com/javase/6/docs/api/java/lang/String.html#intern())为：

**JavaDoc 写道**

public String intern()   
  
    Returns a canonical representation for the string object.   
  
    A pool of strings, initially empty, is maintained privately by the class String.   
  
    When the intern method is invoked, if the pool already contains a string equal to this String object as determined by the equals(Object) method, then the string from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is returned.   
  
    It follows that for any two strings s and t, s.intern() == t.intern() is true if and only if s.equals(t) is true.   
  
    All literal strings and string-valued constant expressions are interned. String literals are defined in §3.10.5 of the Java Language Specification   
  
    **Returns:**   
        a string that has the same contents as this string, but is guaranteed to be from a pool of unique strings.

===============================================================   
  
再换一个问题来问：

**引用**

问题：

**Java代码  [收藏代码](javascript:void())**

1. String s = **new** String("xyz");

涉及用户声明的几个String类型的变量？

答案也很简单：

**引用**

答案：一个，就是String s。

把问题换成下面这个版本，答案也一样：

**引用**

问题：

**Java代码  [收藏代码](javascript:void())**

1. String s = **null**;

涉及用户声明的几个String类型的变量？

Java里变量就是变量，引用类型的变量只是对某个对象实例或者null的引用，不是实例本身。声明变量的个数跟创建实例的个数没有必然关系，像是说：

**Java代码  [收藏代码](javascript:void())**

1. String s1 = "a";
2. String s2 = s1.concat("");
3. String s3 = **null**;
4. **new** String(s1);

这段代码会涉及3个String类型的变量，   
1、s1，指向下面String实例的1   
2、s2，指向与s1相同   
3、s3，值为null，不指向任何实例   
  
以及3个String实例，   
1、"a"字面量对应的驻留的字符串常量的String实例   
2、""字面量对应的驻留的字符串常量的String实例   
（[String.concat()](http://download.oracle.com/javase/6/docs/api/java/lang/String.html#concat(java.lang.String))是个有趣的方法，当发现传入的参数是空字符串时会返回this，所以这里不会额外创建新的String实例）   
3、通过new String(String)创建的新String实例；没有任何变量指向它。   
  
===============================================================   
  
回到楼主开头引用的问题与“标准答案”

**引用**

问题：

**Java代码  [收藏代码](javascript:void())**

1. String s = **new** String("xyz");

创建了几个String Object？   
答案：两个(一个是“xyz”,一个是指向“xyz”的引用对象s)

用归谬法论证。假定问题问的是“在执行这段代码片段时创建了几个String实例”。如果“标准答案”是正确的，那么下面的代码片段在执行时就应该创建4个String实例了：

**Java代码  [收藏代码](javascript:void())**

1. String s1 = **new** String("xyz");
2. String s2 = **new** String("xyz");

马上就会有人跳出来说上下两个"xyz"字面量都是引用了同一个String对象，所以不应该是创建了4个对象。   
  
那么应该是多少个？   
  
运行时的类加载过程与实际执行某个代码片段，两者必须分开讨论才有那么点意义。   
  
为了执行问题中的代码片段，其所在的类必然要先被加载，而且同一个类最多只会被加载一次（要注意对JVM来说“同一个类”并不是类的全限定名相同就足够了，而是<类全限定名, 定义类加载器>一对都相同才行）。   
  
根据上文引用的规范的内容，符合规范的JVM实现应该在类加载的过程中创建并驻留一个String实例作为常量来对应"xyz"字面量；具体是在类加载的resolve阶段进行的。这个常量是全局共享的，只在先前尚未有内容相同的字符串驻留过的前提下才需要创建新的String实例。   
  
等到真正执行原问题中的代码片段时，JVM需要执行的字节码类似这样：

**Java bytecode代码  [收藏代码](javascript:void())**

1. 0: new  #2; //class java/lang/String
2. 3: dup
3. 4: ldc  #3; //String xyz
4. 6: invokespecial    #4; //Method java/lang/String."<init>":(Ljava/lang/String;)V
5. 9: astore\_1

这之中出现过多少次new java/lang/String就是创建了多少个String对象。也就是说原问题中的代码在每执行一次只会新创建一个String实例。   
这里，ldc指令只是把先前在类加载过程中已经创建好的一个String对象（"xyz"）的一个引用压到操作数栈顶而已，并不新创建String对象。   
  
所以刚才用于归谬的代码片段：

**Java代码  [收藏代码](javascript:void())**

1. String s1 = **new** String("xyz");
2. String s2 = **new** String("xyz");

每执行一次只会新创建2个String实例。   
  
---------------------------------------------------------------   
  
为了避免一些同学犯糊涂，再强调一次：   
  
在Java语言里，“new”表达式是负责创建实例的，其中会调用构造器去对实例做初始化；构造器自身的返回值类型是void，并不是“构造器返回了新创建的对象的引用”，而是new表达式的值是新创建的对象的引用。   
  
对应的，在JVM里，“new”字节码指令只负责把实例创建出来（包括分配空间、设定类型、所有字段设置默认值等工作），并且把指向新创建对象的引用压到操作数栈顶。此时该引用还不能直接使用，处于未初始化状态（uninitialized）；如果某方法a含有代码试图通过未初始化状态的引用来调用任何实例方法，那么方法a会通不过JVM的字节码校验，从而被JVM拒绝执行。   
能对未初始化状态的引用做的唯一一种事情就是通过它调用实例构造器，在Class文件层面表现为特殊初始化方法“<init>”。实际调用的指令是invokespecial，而在实际调用前要把需要的参数按顺序压到操作数栈上。在上面的字节码例子中，压参数的指令包括dup和ldc两条，分别把隐藏参数（新创建的实例的引用，对于实例构造器来说就是“this”）与显式声明的第一个实际参数（"xyz"常量的引用）压到操作数栈上。   
在构造器返回之后，新创建的实例的引用就可以正常使用了。   
  
关于构造器的讨论，可以参考我之前的一帖，[实例构造器是不是静态方法？](http://rednaxelafx.iteye.com/blog/652719)   
  
===============================================================   
  
以上讨论都只是针对规范所定义的Java语言与Java虚拟机而言。概念上是如此，但实际的JVM实现可以做得更优化，原问题中的代码片段有可能在实际执行的时候一个String实例也不会完整创建（没有分配空间）。   
  
例如说，在x86、Windows Vista SP2、Sun JDK 6 update 14的fastdebug版上跑下面的测试代码：

**Java代码  [收藏代码](javascript:void())**

1. **public** **class** C2EscapeAnalysisDemo {
2. **private** **static** **void** warmUp() {
3. IFoo[] array = **new** IFoo[] {
4. **new** FooA(), **new** FooB(), **new** FooC(), **new** FooD()
5. };
6. **for** (**int** i = 0; i < 1000000; i++) {
7. array[i % array.length].foo(); // megamorphic callsite to prevent inlining
8. }
9. }
11. **public** **static** **void** main(String[] args) {
12. **while** (**true**) {
13. warmUp();
14. }
15. }
16. }
18. **interface** IFoo {
19. **void** foo();
20. }
22. **class** FooA **implements** IFoo {
23. **public** **void** foo() {
24. String s1 = **new** String("xyz");
25. }
26. }
28. **class** FooB **implements** IFoo {
29. **public** **void** foo() {
30. String s1 = **new** String("xyz");
31. String s2 = **new** String("xyz");
32. }
33. }
35. **class** FooC **implements** IFoo {
36. **public** **void** foo() {
37. String s1 = **new** String("xyz");
38. String s2 = **new** String("xyz");
39. String s3 = **new** String("xyz");
40. }
41. }
43. **class** FooD **implements** IFoo {
44. **public** **void** foo() {
45. String s1 = **new** String("xyz");
46. String s2 = **new** String("xyz");
47. String s3 = **new** String("xyz");
48. String s4 = **new** String("xyz");
49. }
50. }

照常用javac用默认参数编译，然后先用server模式的默认配置来跑，顺带打出GC和JIT编译日志来看

**Command prompt代码  [收藏代码](javascript:void())**

1. java -server -verbose:gc -XX:+PrintCompilation C2EscapeAnalysisDemo

看到的日志的开头一段如下：

**Hotspot log代码  [收藏代码](javascript:void())**

1. 1       java.lang.String::charAt (33 bytes)
2. 2       java.lang.Object::<init> (1 bytes)
3. 3       java.lang.String::<init> (61 bytes)
4. 1%      C2EscapeAnalysisDemo::warmUp @ 47 (71 bytes)
5. 4       FooA::foo (11 bytes)
6. 5       FooB::foo (21 bytes)
7. 6       FooC::foo (31 bytes)
8. 7       FooD::foo (42 bytes)
9. [GC 3072K->168K(32768K), 0.0058325 secs]
10. [GC 3240K->160K(32768K), 0.0104623 secs]
11. [GC 3232K->160K(32768K), 0.0027323 secs]
12. [GC 3232K->160K(35840K), 0.0026220 secs]
13. [GC 6304K->160K(35840K), 0.0173733 secs]
14. [GC 6304K->144K(41664K), 0.0059720 secs]
15. [GC 12432K->144K(41664K), 0.0353320 secs]
16. [GC 12432K->144K(54016K), 0.0139333 secs]
17. 8       C2EscapeAnalysisDemo::warmUp (71 bytes)
18. [GC 24720K->160K(54016K), 0.0697970 secs]
19. [GC 24736K->160K(68800K), 0.0261921 secs]
20. [GC 39520K->160K(68800K), 0.0958433 secs]
21. [GC 39520K->160K(87168K), 0.0433377 secs]
22. [GC 57888K->160K(87168K), 0.0542482 secs]
23. [GC 57888K->148K(87168K), 0.0533140 secs]
24. [GC 57876K->164K(84288K), 0.0533537 secs]
25. [GC 55204K->164K(81728K), 0.0596820 secs]
26. [GC 52644K->164K(79488K), 0.0515090 secs]
27. [GC 50212K->164K(76992K), 0.0491227 secs]
28. [GC 47908K->164K(74944K), 0.0450666 secs]
29. [GC 45668K->164K(72640K), 0.0467671 secs]
30. [GC 43556K->152K(70784K), 0.0420757 secs]
31. [GC 41560K->168K(68736K), 0.0391296 secs]
32. [GC 39656K->168K(67072K), 0.0397539 secs]
33. [GC 37864K->188K(65216K), 0.0360861 secs]

上面的日志中，后面的方法名的行是JIT编译的日志，而以[GC开头的是minor GC的日志。   
程序一直跑，GC的日志还会不断的打出来。这是理所当然的对吧？HotSpot的堆就那么大，而测试代码在不断新创建String对象，肯定得不断触发GC的。   
  
用不同的VM启动参数来跑的话，

**Command prompt代码  [收藏代码](javascript:void())**

1. java -server -verbose:gc -XX:+PrintCompilation -XX:+DoEscapeAnalysis -XX:+EliminateAllocations C2EscapeAnalysisDemo

还是同样的Java测试程序，同样的Sun JDK 6 update 14，但打开了逃逸分析和空间分配消除功能，再运行，看到的全部日志如下：

**Hotspot log代码  [收藏代码](javascript:void())**

1. 1       java.lang.String::charAt (33 bytes)
2. 2       java.lang.Object::<init> (1 bytes)
3. 3       java.lang.String::<init> (61 bytes)
4. 1%      C2EscapeAnalysisDemo::warmUp @ 47 (71 bytes)
5. 5       FooB::foo (21 bytes)
6. 4       FooA::foo (11 bytes)
7. 6       FooC::foo (31 bytes)
8. 7       FooD::foo (42 bytes)
9. [GC 3072K->176K(32768K), 0.0056527 secs]
10. 8       C2EscapeAnalysisDemo::warmUp (71 bytes)

继续跑下去也没有再打出GC日志了。难道新创建String对象都不吃内存了么？   
  
实际情况是：经过HotSpot的server模式编译器的优化后，FooA、FooB、FooC、FooD四个版本的foo()实现都不新创建String实例了。这样自然不吃内存，也就不再触发GC了。   
经过的分析和优化笼统说有方法内联（method inlining）、逃逸分析（escape analysis）、标量替换（scalar replacement）、无用代码削除（dead-code elimination）之类。   
  
FooA.foo()最短，就以它举例来大致演示一下优化的过程。   
它其实就是创建并初始化了一个String对象而已。调用的构造器的源码是：

**Java代码  [收藏代码](javascript:void())**

1. **public** String(String original) {
2. **int** size = original.count;
3. **char**[] originalValue = original.value;
4. **char**[] v;
5. **if** (originalValue.length > size) {
6. // The array representing the String is bigger than the new
7. // String itself.  Perhaps this constructor is being called
8. // in order to trim the baggage, so make a copy of the array.
9. **int** off = original.offset;
10. v = Arrays.copyOfRange(originalValue, off, off+size);
11. } **else** {
12. // The array representing the String is the same
13. // size as the String, so no point in making a copy.
14. v = originalValue;
15. }
16. **this**.offset = 0;
17. **this**.count = size;
18. **this**.value = v;
19. }

因为参数是"xyz"，可以确定在我们的测试代码里不会走到构造器的if分支里，下面为了演示方便就省略掉那部分代码（实际代码还是存在的，只是没执行而已）

**Java代码  [收藏代码](javascript:void())**

1. **public** String(String original) {
2. **int** size = original.count;
3. **char**[] originalValue = original.value;
4. **char**[] v;
5. **if** (originalValue.length > size) {
6. // 省略
7. } **else** {
8. v = originalValue;
9. }
10. **this**.offset = 0;
11. **this**.count = size;
12. **this**.value = v;
13. }

那么把构造器内联到FooA.foo()里，

**Java代码  [收藏代码](javascript:void())**

1. **public** FooA **implements** IFoo {
2. **public** **void** foo() {
3. String s = AllocateString(); // 这里虚构一个只分配空间，不调用构造器的函数
4. String original = "xyz";
6. // 下面就是内联进来的构造器内容
7. **int** size = original.count;
8. **char**[] originalValue = original.value;
9. **char**[] v;
10. **if** (originalValue.length > size) {
11. // 省略
12. } **else** {
13. v = originalValue;
14. }
16. s.offset = 0;
17. s.count = size;
18. s.value = v;
19. }
20. }

然后经过逃逸分析与标量替换，

**Java代码  [收藏代码](javascript:void())**

1. **public** FooA **implements** IFoo {
2. **public** **void** foo() {
3. String original = "xyz";
5. // 下面就是内联进来的构造器内容
6. **int** size = original.count;
7. **char**[] originalValue = original.value;
8. **char**[] v;
9. **if** (originalValue.length > size) {
10. // 省略
11. } **else** {
12. v = originalValue;
13. }
15. // 原本s的实例变量被标量替换为foo()的局部变量
16. **int** sOffset = 0;
17. **int** sCount = size;
18. **char**[] sValue = v;
19. }
20. }

注意，到这里就已经把新创建String在堆上分配空间的代码全部削除了，原本新建的String实例的字段变成了FooA.foo()的局部变量。   
最后再经过无用代码削除，把sOffset、sCount和sValue这三个没被读过的局部变量给削除掉，

**Java代码  [收藏代码](javascript:void())**

1. **public** FooA **implements** IFoo {
2. **public** **void** foo() {
3. String original = "xyz";
5. // 下面就是内联进来的构造器内容
6. **int** size = original.count;
7. **char**[] originalValue = original.value;
8. **char**[] v;
9. **if** (originalValue.length > size) {
10. // 省略
11. } **else** {
12. v = originalValue;
13. }
15. // 几个局部变量也干掉了
16. }
17. }

这就跟FooA.foo()被优化编译后实际执行的代码基本一致了。   
实际执行的x86代码如下：

**X86 asm代码  [收藏代码](javascript:void())**

1. 0x0247aeec: mov    %eax,-0x4000(%esp) ; 检查栈是否溢出（stack bang），若溢出则这条指令会引发异常
2. 0x0247aef3: push   %ebp               ; 保存老的栈帧指针
3. 0x0247aef4: sub    $0x18,%esp         ; 为新栈帧分配空间
4. 0x0247aefa: mov    $0x1027e9a8,%edi   ; String original /\* EDI \*/ = "xyz"
5. 0x0247aeff: mov    0x8(%edi),%ecx     ; char[] originalValue /\* ECX \*/ = original.value;
6. 0x0247af02: mov    0x8(%ecx),%ebx     ; EBX = originalValue.length
7. 0x0247af05: mov    0x10(%edi),%ebp    ; int size /\* EBP \*/ = original.count;
8. 0x0247af08: cmp    %ebp,%ebx          ; 比较originalValue.length与size
9. 0x0247af0a: jg     0x0247af17         ; 如果originalValue.length > size则跳转到0x0247af17
10. ;   实际不会发生跳转（就是不会执行if分支），所以后面代码省略
11. 0x0247af0c: add    $0x18,%esp         ; 撤销栈帧分配的空间
12. 0x0247af0f: pop    %ebp               ; 恢复老的栈帧指针
13. 0x0247af10: test   %eax,0x310000      ; 方法返回前检查是否需要进入safepoint （{poll\_return}）
14. 0x0247af16: ret                       ; 方法返回
15. 0x0247af17: ; 以下是if分支和异常处理器的代码，因为实际不会执行，省略

看，确实没有新创建String对象了。   
  
另外三个版本的foo()实现也是类似，HotSpot成功的把无用的new String("xyz")全部干掉了。   
关于逃逸分析的例子，可以参考我以前一篇帖，[HotSpot 17.0-b12的逃逸分析/标量替换的一个演示](http://rednaxelafx.iteye.com/blog/659108)   
  
再回头看看楼主的原问题，问题中的代码片段执行的时候（对应到FooA.foo()被调用的时候）一个String对象也没有新建。于是那“标准答案”在现实中的指导意义又有多少呢？   
  
===============================================================   
  
另外，楼主还提到了PermGen：

**QM42977 写道**

"xyz"在perm gen应该还会生成一个对象，因为常量("xyz")都会保存在perm gen中

这里也是需要强调一点：永生代（“Perm Gen”）只是Sun JDK的一个实现细节而已，Java语言规范和Java虚拟机规范都没有规定必须有“Permanent Generation”这么一块空间，甚至没规定要用什么GC算法——不用分代式GC算法哪儿来的“永生代”？   
  
HotSpot的PermGen是用来实现Java虚拟机规范中的“**方法区**”（method area）的。如果使用“方法区”这个术语，在讨论概念中的JVM时就安全得多——大家都必须实现出这个表象。   
当然如何实现又是另一回事了。Oracle JRockit没有PermGen，IBM J9也没有，事实上有这么一块空间特别管理的反而是少数吧orz   
  
事实上新版HotSpot VM也在计划去除PermGen，转而使用native memory来实现方法区存储元数据。在JDK8的HotSpot VM中已经实现了这点。   
可以参考这帖：<http://rednaxelafx.iteye.com/blog/905273>   
  
===============================================================   
  
费那么多口舌，最后点题：请别再拿“String s = new String("xyz");创建了多少个String实例”来面试了吧，既没意义又不涨面子。