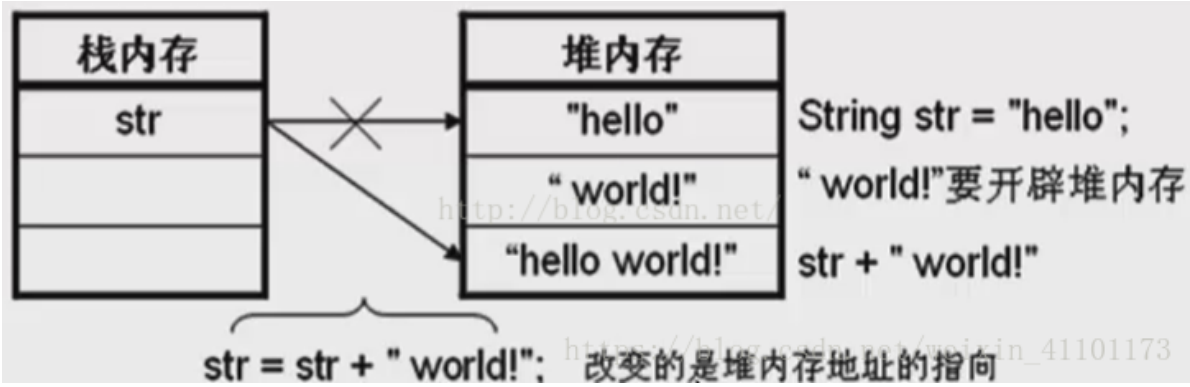
String、StringBuffer和StringBuilder之间的区别和联系

Java String 类——String字符串常量

需要注意的是，String的值是**不可变**的，这就导致每次对String的操作都会生成**新的String对象**，这样不仅效率低下，而且大量浪费有限的内存空间。我们来看一下这张对String操作时内存变化的图：



我们可以看到，初始String值为“hello”，然后在这个字符串后面加上新的字符串“world”，这个过程是需要重新在栈堆内存中开辟内存空间的，最终得到了“hello world”字符串也相应的需要开辟内存空间，这样短短的两个字符串，却需要开辟三次内存空间，不得不说这是对内存空间的**极大浪费**。为了应对经常性的字符串相关的操作，就需要使用Java提供的其他两个操作字符串的类——StringBuffer类和StringBuild类来对此种变化字符串进行处理。

为什么String设计为不可变类呢？

> String设计为不可变类主要考虑到：效率和安全。

- 效率：

  1.在早期的JVM实现版本中，被final修饰的方法会被转为内嵌调用以提升执行效率。而从Java SE5/6开始，就渐渐摈弃这种方式了。因此在现在的Java SE版本中，不需要考虑用final去提升方法调用效率。只有在确定不想让该方法被覆盖时，才将方法设置为final。

  2.缓存hashcode，String不可变，所以hashcode不变，这样缓存才有意义，不必重新计算。

- 安全：String常被作为网络连接，文件操作等参数类型，倘若可改变，会出现意想不到的结果。

String 的创建方式

，String的创建方式有两种：

直接赋值

- 在**方法区**中字符串常量池中创建对象

```
String str = "flyapi";
```

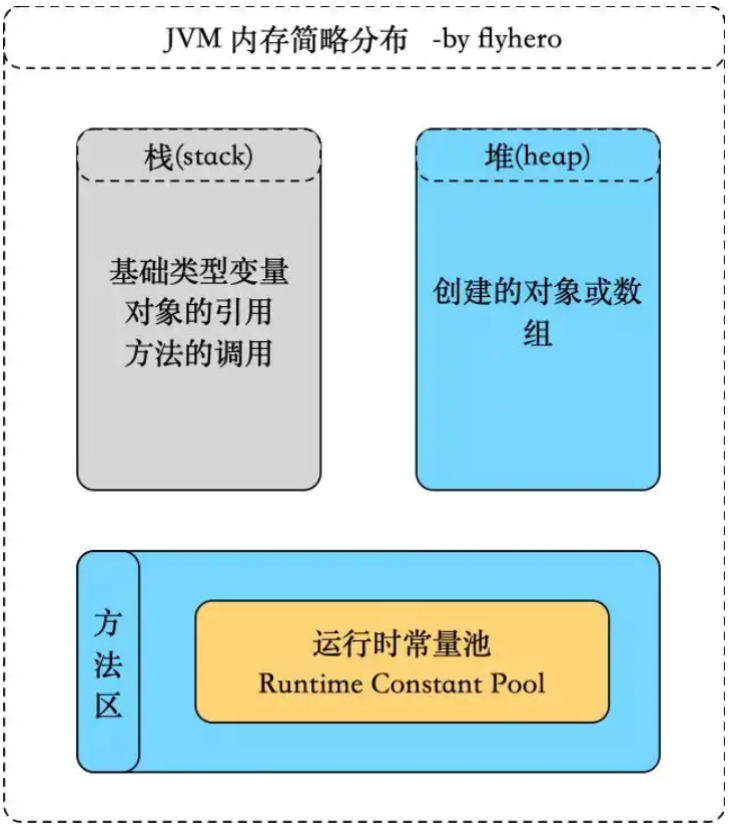
构造函数

- 在**堆内存**创建对象

```
String str = new String();
```

分析

要理解String，那么要了解JVM内存中的栈(stack)、堆(heap)和方法区。简要图如下：



常量池(constant pool)指的是在编译期被确定，并被保存在已编译的.class文件中的一些数据。它包括了关于类、方法、接口等中的常量，**也包括字符串常量。**

- **str1 == str2**

```
String str1 = "HelloFlyapi";
String str2 = "HelloFlyapi";

System.out.println(str1 == str2); // true
```

当执行第一句时，JVM会先去常量池中查找是否存在HelloFlyapi，**当存在时直接返回常量池里的引用；当不存在时，会在字符创常量池中创建一个对象并返回引用。**

当执行第二句时，由于第一句已经在常量池中创建了，所以**直接返回上句创建的对象在常量池中的引用。**

- **str1 == str3**

```
String str1 = "HelloFlyapi";
String str3 = new String("HelloFlyapi");

System.out.println(str1 == str3); // false
```

执行第一句，同上第一句。

执行第二句时，会**在堆（heap）中创建一个对象**(因为使用了new调用构造函数)，**当字符创常量池中没有'HelloFlyapi'时，会在常量池中也创建一个对象；当常量池中已经存在了，就不会创建新的了。**

- **str1 == str6**

```
String str1 = "HelloFlyapi";
String str6 = "Hello" + "Flyapi";

System.out.println(str1 == str6); // true
```

由于"Hello"和"Flyapi"都是常量，编译时，**第二句会被自动编译为'String str6 = "HelloFlyapi";'**

- **str1 == str7**

```
String str1 = "HelloFlyapi";
String str4 = "Hello";
String str5 = "Flyapi";
String str7 = str4 + str5;

System.out.println(str1 == str7); // false
```

其中前三句变量存储的是常量池中的引用地址。

第四句执行时，JVM**会在堆（heap）中创建一个以str4为基础的一个StringBuilder对象，然后调用StringBuilder的append()方法完成与str5的合并，之后会调用toString()方法在堆（heap）中创建一个String对象，并把这个String对象的引用赋给str7。**

String.intern() 方法：

再补充介绍一点：存在于.class文件中的常量池，在运行期被JVM装载，**并且可以扩充**。String的intern()方法就是**扩充常量池的一个方法**；当一个String实例str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str的字符串并返回它的引用**；

```
String s0= “kvill”;
String s1=new String(”kvill”);
String s2=new String(“kvill”);
System.out.println( s0==s1 );  // false
System.out.println( “**********” );
s1.intern();
s2=s2.intern(); //把常量池中“kvill”的引用赋给s2
System.out.println( s0==s1);  // false
System.out.println( s0==s1.intern() ); //true
System.out.println( s0==s2 );//true
```

面试题

String str = new String("abc")创建了多少个实例？

这个问题其实是不严谨的，但面试一般会遇到，所以我们要补充来说明。

> 类的加载和执行要分开来讲： 创建了两个
>
> 1. 当加载类时，"abc"被创建并驻留在了字符创常量池中（如果先前加载中没有创建驻留过）。
> 2. 当执行此句时，因为"abc"对应的String实例已经存在于字符串常量池中，所以JVM会将此实例复制到会在堆（heap）中并返回引用地址。

通过字节码我们可以看到：

源码：String str = new String("abc")

字节码：

```
    Code:
       0: new           #2                  // class java/lang/String
       3: dup
       4: ldc           #3                  // String abc
       6: invokespecial #4                  // Method java/lang/String."<init>":
(Ljava/lang/String;)
       9: astore_1
      10: return
复制代码
```

执行时仅(#2)创建了一个对象。所以标准答案为在运行时涉及两个String实例

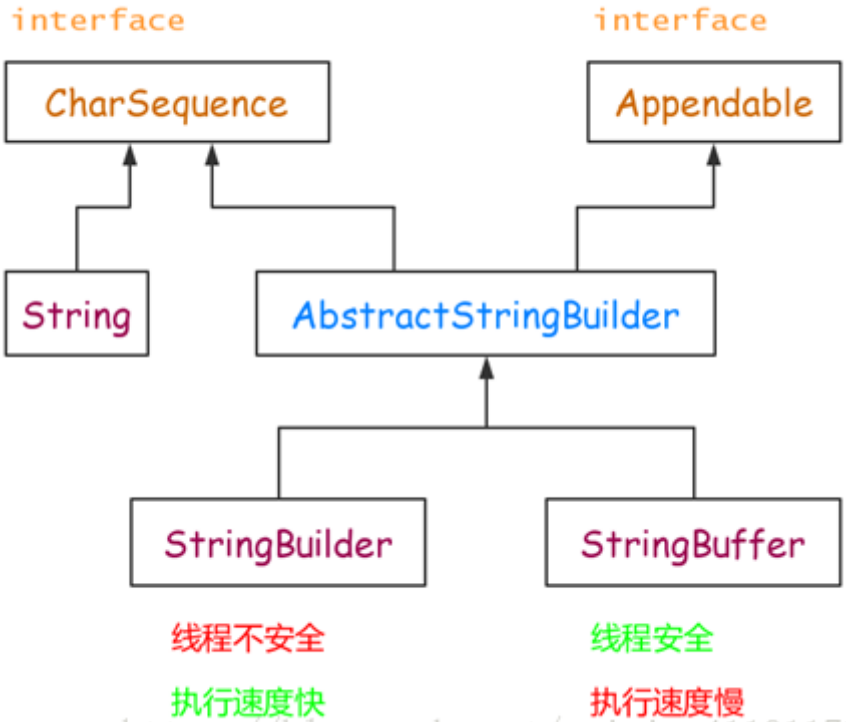
StringBuffer 和 StringBuilder 类——StringBuffer、StringBuilder字符串变量

当***对字符串进行** 修改***的时候，需要使用 StringBuffer 和 StringBuilder 类。

和 String 类不同的是，StringBuffer 和 StringBuilder 类的对象能够**被多次的修改，并且不产生新的未使用对象**。

StringBuilder 类在 Java 5 中被提出，它和 StringBuffer 之间的最大不同在于 StringBuilder 的方法不是线程安全的（不能同步访问）。

由于 StringBuilder 相较于 StringBuffer 有速度优势，***所以多数情况下建议使用 StringBuilder 类***。然而在应用程序要求**线程安全**的情况下，则必须使用 StringBuffer 类。



String	StringBuffer	StringBuilder
String的值是不可变的，这就导致每次对String的操作都会生成新的String对象，不仅效率低下，而且浪费大量优先的内存空间	StringBuffer是可变类，和线程安全的字符串操作类，任何对它指向的字符串的操作都不会产生新的对象。每个StringBuffer对象都有一定的缓冲区容量，当字符串大小没有超过容量时，不会分配新的容量，当字符串大小超过容量时，会自动增加容量	可变类，速度更快
不可变	可变	可变
	线程安全	线程不安全
	多线程操作字符串	单线程操作字符串