## 一、String类

想要了解一个类，最好的办法就是看这个类的实现源代码，来看一下String类的源码：

```
public final class String implements java.io.Serializable, Comparable, CharSequence
{
 /**
 * The value is used for character storage.
 */
 private final char value[];
 /**
 * The offset is the first index of the storage that is used.
 */
  private final int offset;
  /**
  * The count is the number of characters in the String.
  */
  private final int count;
  /**
  * Cache the hash code for the string
  */
  private int hash; // Default to 0/** use serialVersionUID from JDK 1.0.2 for interoperability */privatestaticfinallong serialVersionUID = -6849794470754667710L;
 // ........
 }
```

从上面可以看出几点：

1）String类是final类，也即意味着String类不能被继承，并且它的成员方法都默认为final方法。在Java中，被final修饰的类是不允许被继承的，并且该类中的成员方法都默认为final方法。

2）上面列举出了String类中所有的成员属性，从上面可以看出String类其实是通过char数组来保存字符串的。

下面再继续看String类的一些方法实现：

```
public String substring(int beginIndex, int endIndex)
{
 if (beginIndex < 0)
 {
 throw new StringIndexOutOfBoundsException(beginIndex);
 }
 if (endIndex > count)
 {
```

```
 throw new StringIndexOutOfBoundsException(endIndex);
 }
 if (beginIndex > endIndex)
 {
 throw new StringIndexOutOfBoundsException(endIndex - beginIndex);
 }
 return ((beginIndex == 0) && (endIndex == count)) ?
 this :
 new String(offset + beginIndex, endIndex - beginIndex, value);
}

public String concat(String str)
{
 int otherLen = str.length();
 if (otherLen == 0)
 {
 return this;
 }
 char buf[] = new char[count + otherLen];
 getChars(0, count, buf, 0);
 str.getChars(0, otherLen, buf, count);
 return new String(0, count + otherLen, buf);
}

public String replace(char oldChar, char newChar)
{
 if (oldChar != newChar)
 {
 int len = count;
 int i = -1;
 char[] val = value; /* avoid getfield opcode */
 int off = offset; /* avoid getfield opcode */
 while (++i < len)
 {
 if (val[off + i] == oldChar)
 {
 break;
 }
 }
 if (i < len)
```

```
{
char buf[] = newchar[len];
for (int j = 0; j < i; j++)
{
buf[j] = val[off + j];
}
while (i < len)
{
char c = val[off + i];
buf[i] = (c == oldChar) ? newChar : c;
i++;
}
retur nnew String(0, len, buf);
}
} return this;
}
```

从上面的三个方法可以看出，无论是sub操、concat还是replace操作都不是在原有的字符串上进行的，而是重新生成了一个新的字符串对象。也就是说进行这些操作后，最原始的字符串并没有被改变。

在这里要永远记住一点：**“String对象一旦被创建就是固定不变的了，对String对象的任何改变都不影响到原对象，相关的任何change操作都会生成新的对象”**。

**二、字符串常量池**

我们知道字符串的分配和其他对象分配一样，是需要消耗高昂的时间和空间的，而且字符串我们使用的非常多。JVM为了提高性能和减少内存的开销，在实例化字符串的时候进行了一些优化：**使用字符串常量池。每当我们创建字符串常量时，JVM会首先检查字符串常量池，如果该字符串已经存在常量池中，那么就直接返回常量池中的实例引用。如果字符串不存在常量池中，就会实例化该字符串并且将其放到常量池中。由于String字符串的不可变性我们可以十分肯定常量池中一定不存在两个相同的字符串**（这点对理解上面至关重要）。

Java中的常量池，实际上分为两种形态：**静态常量池**和**运行时常量池。**

所谓**静态常量池**，即*.class文件中的常量池，class文件中的常量池不仅仅包含字符串(数字)字面量，还包含类、方法的信息，占用class文件绝大部分空间。

而**运行时常量池**，则是jvm虚拟机在完成类装载操作后，将class文件中的常量池载入到内存中，并保存在方法区中，我们常说的常量池，就是指方法区中的运行时常量池。

来看下面的程序：

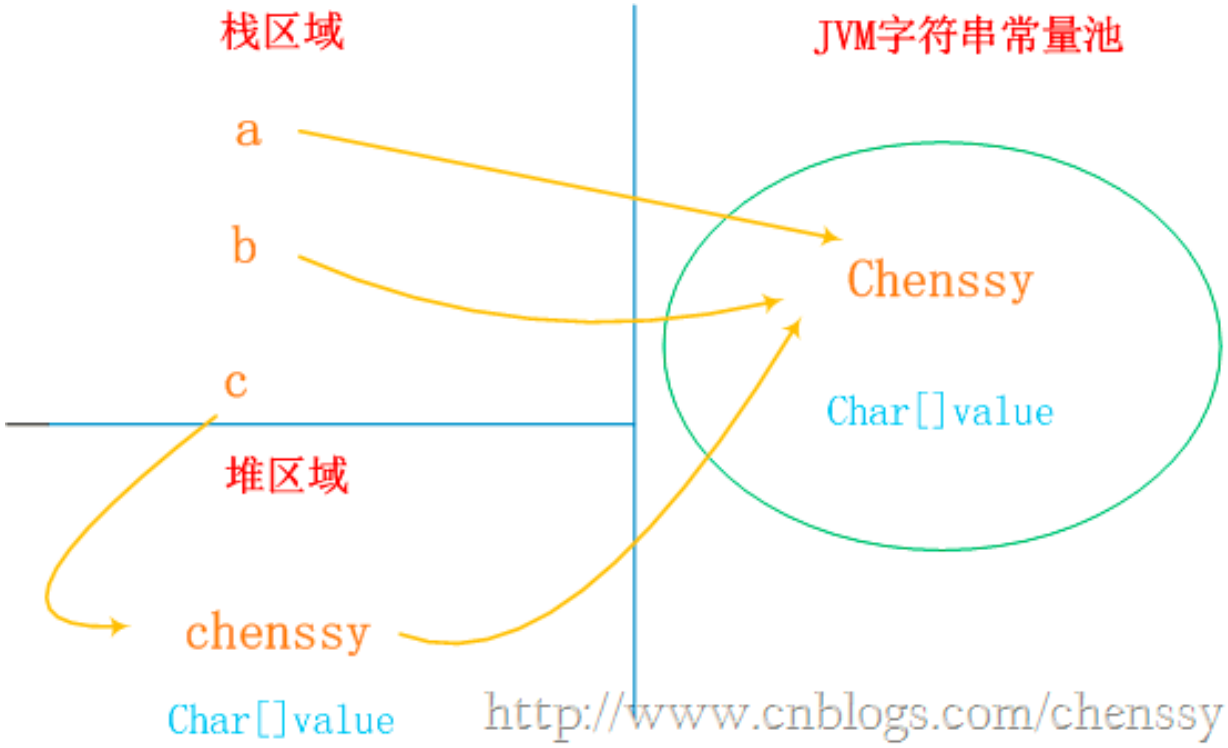
```
String a = "chenssy";String b = "chenssy";
```

a、b和字面上的chenssy都是指向JVM字符串常量池中的"chenssy"对象，他们指向同一个对象。

```
String c = new String("chenssy");
```

new关键字一定会产生一个对象chenssy（注意这个chenssy和上面的chenssy不同），同时这个对象是存储在堆中。所以上面应该产生了两个对象：保存在栈中的c和保存

堆中chenssy。但是在Java中根本就不存在两个完全一模一样的字符串对象。故堆中的chenssy应该是引用字符串常量池中chenssy。所以c、chenssy、池chenssy的关系应该是：c--->chenssy--->池chenssy。整个关系如下：




通过上面的图我们可以非常清晰的认识他们之间的关系。所以我们修改内存中的值，他变化的是所有。

**总结：** 虽然a、b、c、chenssy是不同的对象，但是从String的内部结构我们是可以理解上面的。String c = new String("chenssy");虽然c的内容是创建在堆中，但是他的内部value还是指向JVM常量池的chenssy的value，它构造chenssy时所用的参数依然是chenssy字符串常量。

下面再来看几个例子：

例子1：

```
/**
 * 采用字面值的方式赋值
 */
public void test1()
{
 String str1 = "aaa";
 String str2 = "aaa";
 System.out.println("===========test1============");
 System.out.println(str1 == str2);//true 可以看出str1跟str2是指向同一个对象
}
}
```

执行上述代码，结果为：true。

**分析**：当执行String str1="aaa"时，JVM首先会去字符串池中查找是否存在"aaa"这个对象，如果不存在，则在字符串池中创建"aaa"这个对象，然后将池中"aaa"这个对象的引用

地址返回给字符串常量str1，这样str1会指向池中"aaa"这个字符串对象；如果存在，则不创建任何对象，直接将中"aaa"这个对象的地址返回，赋给字符串常量。当创建字符串对象str2时，字符串池中已经存在"aaa"这个对象，直接把对象"aaa"的引用地址返回给str2，这样str2指向了池中"aaa"这个对象，也就是说str1和str2指向了同一个对象，因此语句System.out.println(str1 == str2)输出：true。

例子2：

```
/**
 * 采用new关键字新建一个字符串对象
 */
public void test2()
{
 String str3 = new String("aaa");
 String str4 = new String("aaa");
 System.out.println("===========test2============");
 System.out.println(str3 == str4);//false 可以看出用new的方式是生成不同的对象 }
 }
```

执行上述代码，结果为：false。

**分析**： 采用new关键字新建一个字符串对象时，JVM首先在字符串池中查找有没有"aaa"这个字符串对象，如果有，则不在池中再去创建"aaa"这个对象了，直接在堆中创建一个"aaa"字符串对象，然后将堆中的这个"aaa"对象的地址返回赋给引用str3，这样，str3就指向了堆中创建的这个"aaa"字符串对象；如果没有，则首先在字符串池中创建一个"aaa"字符串对象，然后再在堆中创建一个"aaa"字符串对象，然后将堆中这个"aaa"字符串对象的地址返回赋给str3引用，这样，str3指向了堆中创建的这个"aaa"字符串对象。当执行String str4=new String("aaa")时， 因为采用new关键字创建对象时，每次new出来的都是一个新的对象，也即是说引用str3和str4指向的是两个不同的对象，因此语句System.out.println(str3 == str4)输出：false。

例子3：

```
/**
 * 编译期确定
 */
public void test3()
{
 String s0 = "helloworld";
 String s1 = "helloworld";
 String s2 = "hello" + "world";
 System.out.println("===========test3============");
  System.out.println(s0 == s1); //true
  // 可以看出s0跟s1是指向同一个对象 System.out.println(s0==s2);
  // true 可以看出s0跟s2是指向同一个对象
```

```
}
```

执行上述代码，结果为：true、true。

**分析**：因为例子中的s0和s1中的"helloworld”都是字符串常量，它们在编译期就被确定了，所以s0==s1为true；而"hello”和"world”也都是字符串常量，当一个字符串由多个字符串常量连接而成时，它自己肯定也是字符串常量，所以s2也同样在编译期就被解析为一个字符串常量，所以s2也是常量池中"helloworld”的一个引用。所以我们得出s0==s1==s2。

例子4：

```
/**
 * 编译期无法确定
 */
public void test4()
{
 String s0 = "helloworld";
 String s1 = new String("helloworld");
 String s2 = "hello" + new String("world");
 System.out.println("===========test4============");
  System.out.println( s0==s1 );// false
  System.out.println( s0==s2 );// false
  System.out.println( s1==s2 );// false
}
```

执行上述代码，结果为：false、false、false。

**分析**：用new String() 创建的字符串不是常量，不能在编译期就确定，所以new String() 创建的字符串不放入常量池中，它们有自己的地址空间。

s0还是常量池中"helloworld”的引用，s1因为无法在编译期确定，所以是运行时创建的新对象"helloworld”的引用，s2因为有后半部分new String(”world”)所以也无法在编译期确定，所以也是一个新创建对象"helloworld”的引用。

例子5：

```
/**
 * 继续-编译期无法确定
 */
public void test5()
{
 String str1 = "abc";
 String str2 = "def";
 String str3 = str1 + str2;
 System.out.println("===========test5============");
  System.out.println(str3 == "abcdef");
  //false
}
```

执行上述代码，结果为：false。

**分析**：因为str3指向堆中的"abcdef"对象，而"abcdef"是字符串池中的对象，所以结果为false。JVM对String str="abc"对象放在常量池中是在编译时做的，而String str3=str1+str2是在运行时刻才能知道的。new对象也是在运行时才做的。而这段代码总共创建了5个对象，字符串池中两个、堆中三个。+运算符会在堆中建立来两个String对象，这两个对象的值分别是"abc"和"def"，也就是说从字符串池中复制这两个值，然后在堆中创建两个对象，然后再建立对象str3,然后将"abcdef"的堆地址赋给str3。

步骤：

1)栈中开辟一块中间存放引用str1，str1指向池中String常量"abc"。

2)栈中开辟一块中间存放引用str2，str2指向池中String常量"def"。

3)栈中开辟一块中间存放引用str3。

4)str1 + str2通过StringBuilder的最后一步toString()方法还原一个新的String对象"abcdef"，因此堆中开辟一块空间存放此对象。

5)引用str3指向堆中(str1 + str2)所还原的新String对象。

6)str3指向的对象在堆中，而常量"abcdef"在池中，输出为false。

例子6：

```
/**
 * 编译期优化
 */
public void test6()
{
 String s0 = "a1";
 String s1 = "a" + 1;
 System.out.println("===========test6============");
 System.out.println((s0 == s1)); //result = true
  String s2 = "atrue";
  String s3 = "a" + "true";
  System.out.println((s2 == s3)); //result = true
  String s4 = "a3.4";
  String s5 = "a" + 3.4;
  System.out.println((s4 == s5)); //result = true
 }
```

执行上述代码，结果为：true、true、true。

**分析**：在程序编译期，JVM就将常量字符串的"+"连接优化为连接后的值，拿"a" + 1来说，经编译器优化后在class中就已经是a1。在编译期其字符串常量的值就确定下来，故上面程序最终的结果都为true。

例子7：

```
/**
 * 编译期无法确定
 */
```

```
public void test7()
{
 String s0 = "ab";
 String s1 = "b";
 String s2 = "a" + s1;
 System.out.println("===========test7============");
  System.out.println((s0 == s2)); //result = false}
 }
```

执行上述代码，结果为：false。

**分析**：JVM对于字符串引用，由于在字符串的"+"连接中，有字符串引用存在，而引用的值在程序编译期是无法确定的，即"a" + s1无法被编译器优化，只有在程序运行期来动态分配并将连接后的新地址赋给s2。所以上面程序的结果也就为false。

例子8：

```
/**
 * 比较字符串常量的“+”和字符串引用的“+”的区别
 */
public void test8()
{
 String test = "javalanguagespecification";
 String str = "java";
 String str1 = "language";
 String str2 = "specification";
  System.out.println("===========test8============");
  System.out.println(test == "java" + "language" + "specification");
  System.out.println(test == str + str1 + str2);
 }
```

执行上述代码，结果为：true、false。

**分析**：为什么出现上面的结果呢？这是因为，字符串字面量拼接操作是在Java编译器编译期间就执行了，也就是说编译器编译时，直接把"java"、"language"和"specification"这三个字面量进行"+"操作得到一个"javalanguagespecification" 常量，并且直接将这个常量放入字符串池中，这样做实际上是一种优化，将3个字面量合成一个，避免了创建多余的字符串对象。而字符串引用的"+"运算是在Java运行期间执行的，即str + str2 + str3在程序执行期间才会进行计算，它会在堆内存中重新创建一个拼接后的字符串对象。总结来说就是：字面量"+"拼接是在编译期间进行的，拼接后的字符串存放在字符串池中；而字符串引用的"+"拼接运算实在运行时进行的，新创建的字符串存放在堆中。

对于直接相加字符串，效率很高，因为在编译器便确定了它的值，也就是说形如"I"+"love"+"java"; 的字符串相加，在编译期间便被优化成了"Ilovejava"。对于间接相加（即包含字符串引用），形如s1+s2+s3; 效率要比直接相加低，因为在编译器不会对引用变量进行优化。

例子9：

```java
/**
 * 编译期确定
 */
public void test9()
{
 String s0 = "ab";
 final String s1 = "b";
 String s2 = "a" + s1;
 System.out.println("===========test9===========");
  System.out.println((s0 == s2)); //result = true
 }
```

执行上述代码，结果为：true。

**分析**：和例子7中唯一不同的是s1字符串加了final修饰，对于final修饰的变量，它在编译时被解析为常量值的一个本地拷贝存储到自己的常量池中或嵌入到它的字节码流中。所以此时的"a" + s1和"a" + "b"效果是一样的。故上面程序的结果为true。

例子10：

```java
/**
 * 编译期无法确定
 */
public void test10()
{
 String s0 = "ab";
 final String s1 = getS1();
 String s2 = "a" + s1;
 System.out.println("===========test10===========");
  System.out.println((s0 == s2)); //result = false
 }

private static String getS1()
{
  return "b";
 }
```

执行上述代码，结果为：false。

**分析**：这里面虽然将s1用final修饰了，但是由于其赋值是通过方法调用返回的，那么它的值只能在运行期间确定，因此s0和s2指向的不是同一个对象，故上面程序的结果为false。

## 三、总结

### 1.String类初始化后是不可变的(immutable)

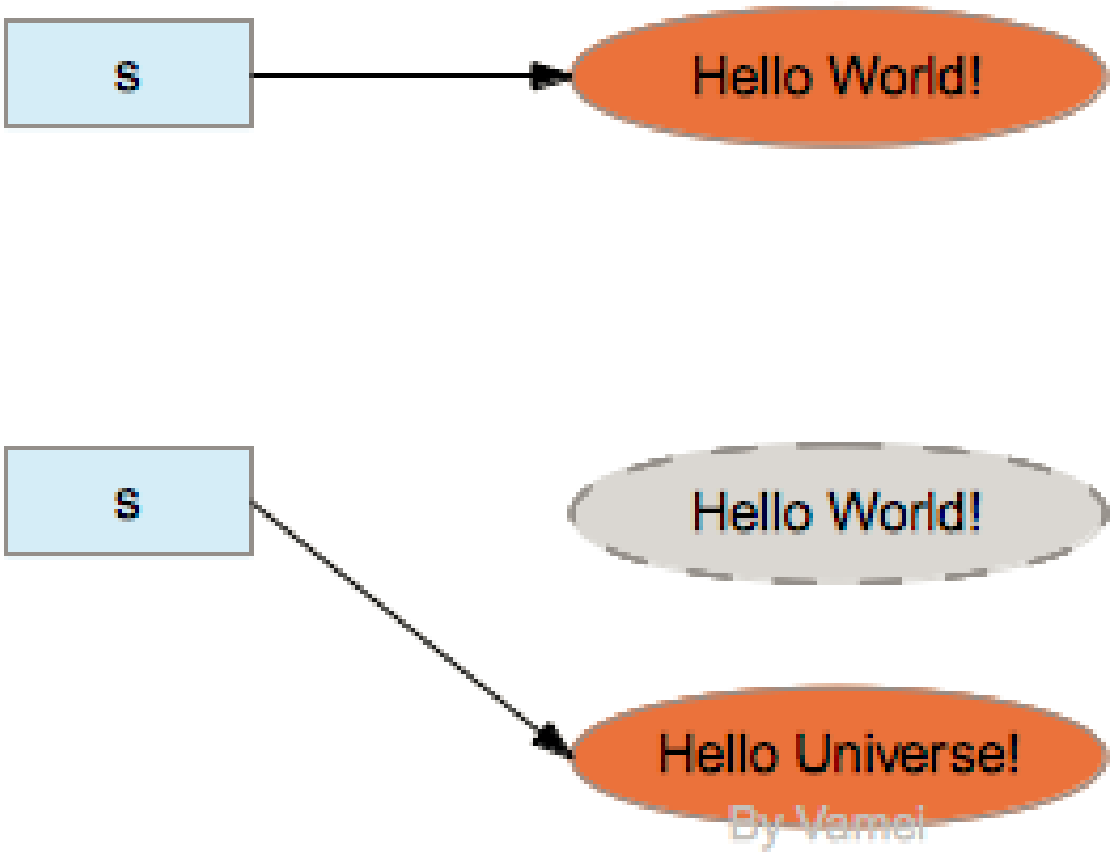
String使用private final char value[]来实现字符串的存储，也就是说String对象创建之后，就不能再修改此对象中存储的字符串内容，就是因为如此，才说String类型是不可变

的(immutable)。程序员不能对已有的不可变对象进行修改。我们自己也可以创建不可变对象，只要在接口中不提供修改数据的方法就可以。

然而，String类对象确实有编辑字符串的功能，比如replace()。这些编辑功能是通过创建一个新的对象来实现的，而不是对原有对象进行修改。比如:

```
1  s = s.replace("World", "Universe");
```

上面对s.replace()的调用将创建一个新的字符串"Hello Universe!"，并返回该对象的引用。通过赋值，引用s将指向该新的字符串。如果没有其他引用指向原有字符串"Hello World!"，原字符串对象将被垃圾回收。




**2.引用变量与对象**

A aa;

这个语句声明一个类A的引用变量aa[我们常常称之为句柄]，而对象一般通过new创建。所以aa仅仅是一个引用变量，它不是对象。

**3.创建字符串的方式**

创建字符串的方式归纳起来有两类:

（1）使用""引号创建字符串;

（2）使用new关键字创建字符串。

结合上面例子，总结如下:

（1）单独使用""引号创建的字符串都是常量,编译期就已经确定存储到String Pool中;

（2）使用new String("")创建的对象会存储到heap中,是运行期新创建的;

**new创建字符串时首先查看池中是否有相同值的字符串，如果有，则拷贝一份到堆中，然后返回堆中的地址；如果池中没有，则在堆中创建一份，然后返回堆中的地址（注意，此时不需要从堆中复制到池中，否则，将使得堆中的字符串永远是池中的子集，导致浪费池的空间）！**

（3）使用只包含常量的字符串连接符如"aa" + "aa"创建的也是常量,编译期就能确定,已经确定存储到String Pool中；

（4）使用包含变量的字符串连接符如"aa" + s1创建的对象是运行期才创建的,存储在heap中；

**4.使用String不一定创建对象**

在执行到双引号包含字符串的语句时，如String a = "123"，JVM会先到常量池里查找，如果有的话返回常量池里的这个实例的引用，否则的话创建一个新实例并置入常量池里。所以，当我们在使用诸如String str = "abc"；的格式定义对象时，总是想当然地认为，创建了String类的对象str。**担心陷阱！对象可能并没有被创建！而可能只是指向一个先前已经创建的对象。**只有通过new()方法才能保证每次都创建一个新的对象。

**5.使用new String，一定创建对象**

在执行String a = new String("123")的时候，首先走常量池的路线取到一个实例的引用，然后在堆上创建一个新的String实例，走以下构造函数给value属性赋值，然后把实例引用赋值给a：

```
1  public String(String original)
2  {
3   int size = original.count;
4   char[] originalValue = original.value;
5   char[] v;
6   if (originalValue.length > size)
7   { // The array representing the String is bigger than the new
8   // String itself. Perhaps this constructor is being called
9   // in order to trim the baggage, so make a copy of the array.int off = o
riginal.offset;
10
11   v = Arrays.copyOfRange(originalValue, off, off + size);
12   }
13   else
14   {
15   // The array representing the String is the same
16   // size as the String, so no point in making a copy.
17   v = originalValue;
18   }
19   this.offset = 0;
20   this.count = size;
21   this.value = v;
22  }
```

从中我们可以看到，虽然是新创建了一个String的实例，但是value是等于常量池中的实例的value，即是说没有new一个新的字符数组来存放"123"。

**6.关于String.intern()**

**intern方法使用：**一个初始为空的字符串池，它由类String独自维护。当调用 intern方法时，如果池已经包含一个等于此String对象的字符串（用equals(oject)方法确定），则返回池中的字符串。否则，将此String对象添加到池中，并返回此String对象的引用。

**它遵循以下规则：对于任意两个字符串 s 和 t，当且仅当 s.equals(t) 为 true 时，s.intern() == t.intern() 才为 true。**

String.intern();

再补充介绍一点：存在于.class文件中的常量池，在运行期间被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。

```
/**
 * 关于String.intern()
 */
public void test11()
{
 String s0 = "kvill";
 String s1 = new String("kvill");
 String s2 = new String("kvill");
 System.out.println("===========test11============");
  System.out.println(s0 == s1); //false
  System.out.println("**********");
  s1.intern();
  //虽然执行了s1.intern(),但它的返回值没有赋给s1
  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//说明s1.intern()返回的是常量池中"kvill"的引用
  System.out.println(s0 == s2); //true
 }
```

运行结果：false、false、true、true。

**7.关于equals和==**

（1）对于==，如果作用于基本数据类型的变量（byte,short,char,int,long,float,double,boolean ），则直接比较其存储的"值"是否相等；如果作用于引用类型的变量（String），则比较的是所指向的对象的地址（即是否指向同一个对象）。

（2）equals方法是基类Object中的方法，因此对于所有的继承于Object的类都会有该方法。在Object类中，equals方法是用来比较两个对象的引用是否相等，即是否指向同一

个对象。

（3）对于equals方法，注意：equals方法不能作用于基本数据类型的变量。如果没有对equals方法进行重写，则比较的是引用类型的变量所指向的对象的地址；而String类对equals方法进行了重写，用来比较指向的字符串对象所存储的字符串是否相等。其他的一些类诸如Double，Date，Integer等，都对equals方法进行了重写用来比较指向的对象所存储的内容是否相等。

```
/**
 * 关于equals和==
 */
public void test12()
{
 String s1 = "hello";
 String s2 = "hello";
 String s3 = new String("hello");
 System.out.println("===========test12============");
  System.out.println(s1 == s2); //true,表示s1和s2指向同一对象，它们都指向常量池中的"hello"对象
  // flase,表示s1和s3的地址不同，即它们分别指向的是不同的对象,s1指向常量池中的地址，s3指向堆中的地址
  System.out.println(s1 == s3);
  System.out.println(s1.equals(s3)); //true,表示s1和s3所指向对象的内容相同
 }
```

**8.String相关的+：**

String中的 + 常用于字符串的连接。看下面一个简单的例子：

```
/**
 * String相关的+
 */
public void test13()
{
 String a = "aa";
 String b = "bb";
 String c = "xx" + "yy " + a + "zz" + "mm" + b;
 System.out.println("===========test13============");
  System.out.println(c);
 }
```

编译运行后，主要字节码部分如下：

```
 public static main([Ljava/lang/String;)V L0 LINENUMBER 5
 L0 LDC "aa" ASTORE 1
 L1 LINENUMBER 6
 L1 LDC "bb" ASTORE 2
```

```
5   L2 LINENUMBER 7
6   L2 NEW
7   java/lang/
8   StringBuilder DUP
9   LDC "xxyy "
10   INVOKESPECIAL java/lang/StringBuilder.(Ljava/lang/String;)
11   V ALOAD 1
12   INVOKEVIRTUAL java/lang/StringBuilder.append(Ljava/lang/String;)Ljava/l
ang/StringBuilder;LDC "zz"
13   INVOKEVIRTUAL java/lang/StringBuilder.append(Ljava/lang/String;)Ljava/l
ang/StringBuilder;LDC "mm"
14   INVOKEVIRTUAL java/lang/StringBuilder.append(Ljava/lang/String;)Ljava/l
ang/StringBuilder;ALOAD 2
15   INVOKEVIRTUAL java/lang/StringBuilder.append(Ljava/lang/String;)Ljava/l
ang/StringBuilder;
16   INVOKEVIRTUAL java/lang/StringBuilder.toString()Ljava/lang/String;ASTOR
E 3
17   L3 LINENUMBER 8
18   L3 GETSTATIC
19   java/lang/System.out :Ljava/io/PrintStream;ALOAD 3
20   INVOKEVIRTUAL java/io/PrintStream.println(Ljava/lang/String;)
21   V L4
22   LINENUMBER 9
23   L4 RETURN
24   L5 LOCALVARIABLE
25   args [Ljava/lang/String;
26   L0 L5 0
27   LOCALVARIABLE a
28   Ljava/lang/String;
29   L1 L5 1
30   LOCALVARIABLE b
31   Ljava/lang/String;
32   L2 L5 2
33   LOCALVARIABLE c
34   Ljava/lang/String;
35   L3 L5 3MAXSTACK =3MAXLOCALS =4
36  }
```

显然，通过字节码我们可以得出如下几点结论：

(1).String中使用 + 字符串连接符进行字符串连接时，连接操作最开始时如果都是字符串常量，编译后将尽可能多的直接将字符串常量连接起来，形成新的字符串常量参与后续连接（通过反编译工具jd-gui也可以方便的直接看出）；

(2).接下来的字符串连接是从左向右依次进行，对于不同的字符串，首先以最左边的字符串为参数创建StringBuilder对象，然后依次对右边进行append操作，最后将StringBuilder对象通过toString()方法转换成String对象（注意：中间的多个字符串常量不会自动拼接）。

也就是说**String c = "xx" + "yy " + a + "zz" + "mm" + b; 实质上的实现过程是：String c = new StringBuilder("xxyy ").append(a).append("zz").append("mm").append(b).toString();**

**由此得出结论：当使用+进行多个字符串连接时，实际上是产生了一个StringBuilder对象和一个String对象。**

**9.String的不可变性导致字符串变量使用+号的代价：**

```
String s = "a" + "b" + "c";
String s1 = "a";
String s2 = "b";
String s3 = "c";
String s4 = s1 + s2 + s3;
```

**分析**：变量s的创建等价于 String s = "abc"; 由上面例子可知编译器进行了优化，这里只创建了一个对象。由上面的例子也可以知道s4不能在编译期进行优化，其对象创建相当于：

```
StringBuilder temp = new StringBuilder();
temp.append(a).append(b).append(c);
String s = temp.toString();
```

由上面的分析结果，可就不难推断出String 采用连接运算符（+）效率低下原因分析，形如这样的代码：

```
public class Test
{
 public static void main(String args[])
 {
 String s = null;
 for (int i = 0; i < 100; i++)
 {
 s += "a";
 }
  }
 }
```

每做一次 + 就产生个StringBuilder对象，然后append后就扔掉。下次循环再到达时重新产生个StringBuilder对象，然后 append 字符串，如此循环直至结束。 如果我们直接采用 StringBuilder 对象进行 append 的话，我们可以节省 N - 1 次创建和销毁对象的时间。所以对于在循环中要进行字符串连接的应用，一般都是用StringBuffer或StringBulider对象来进行append操作。

**10.String、StringBuffer、StringBuilder的区别**

（1）可变与不可变：String是**不可变字符串对象**，StringBuilder和StringBuffer是**可变字符串对象**（其内部的字符数组长度可变）。

（2）是否多线程安全：String中的对象是不可变的，也就可以理解为常量，显然**线程安全**。StringBuffer 与 StringBuilder 中的方法和功能完全是等价的，只是StringBuffer 中的方法大都采用了synchronized 关键字进行修饰，因此是**线程安全**的，而 StringBuilder 没有这个修饰，可以被认为是**非线程安全**的。

（3）String、StringBuilder、StringBuffer三者的执行效率：StringBuilder > StringBuffer > String 当然这个是相对的，不一定在所有情况下都是这样。比如String str = "hello"+ "world"的效率就比 StringBuilder st = new StringBuilder().append("hello").append("world")要高。因此，这三个类是各有利弊，应当根据不同的情况来进行选择使用：

当字符串相加操作或者改动较少的情况下，建议使用 String str="hello"这种形式；

当字符串相加操作较多的情况下，建议使用StringBuilder，如果采用了多线程，则使用StringBuffer。

**11.String中的final用法和理解**

```
final StringBuffer a = new StringBuffer("111");
final StringBuffer b = new StringBuffer("222");
a=b;//此句编译不通过final
StringBuffer a = new StringBuffer("111");
a.append("222");//编译通过
```

可见，**final只对引用的"值"(即内存地址)有效，它迫使引用只能指向初始指向的那个对象，改变它的指向会导致编译期错误。**至于它所指向的对象的变化，final是不负责的。

**12.关于String str = new String("abc")创建了多少个对象?**

这个问题在很多书籍上都有说到比如《Java程序员面试宝典》，包括很多国内大公司笔试面试题都会遇到，大部分网上流传的以及一些面试书籍上都说是2个对象，这种说法是片面的。

首先必须弄清楚创建对象的含义，创建是什么时候创建的？这段代码在运行期间会创建2个对象么？毫无疑问不可能，用javap -c反编译即可得到JVM执行的字节码内容：

```
E:\Workspace\Test\bin\com\cxh\test1>javap -c Main
Compiled from "Main.java"
public class com.cxh.test1.Main extends java.lang.Object{
public com.cxh.test1.Main();
  Code:
   0:   aload_0
   1:   invokespecial   #8; //Method java/lang/Object."<init>":()V
   4:   return

public static void main(java.lang.String[]);
  Code:
   0:   new     #16; //class java/lang/String
   3:   dup
   4:   ldc     #18; //String abc
   6:   invokespecial   #20; //Method java/lang/String."<init>":(Ljava/lang/Stri
ng;)V
   9:   astore_1
   10:  return

}
```

很显然，new只调用了一次，也就是说只创建了一个对象。而这道题目让人混淆的地方就是这里，这段代码在运行期间确实只创建了一个对象，即在堆上创建了"abc"对象。而为什么大家都在说是2个对象呢，这里面要澄清一个概念，该段代码执行过程和类的加载过程是有区别的。在类加载的过程中，确实在运行时常量池中创建了一个"abc"对象，而在代码执行过程中确实只创建了一个String对象。

因此，这个问题如果换成 String str = new String("abc")涉及到几个String对象？合理的解释是2个。个人觉得在面试的时候如果遇到这个问题，可以向面试官询问清楚”是这段代码执行过程中创建了多少个对象还是涉及到多少个对象“再根据具体的来进行回答。

**13.字符串池的优缺点：**

字符串池的优点就是避免了相同内容的字符串的创建，节省了内存，省去了创建相同字符串的时间，同时提升了性能；另一方面，字符串池的缺点就是牺牲了JVM在常量池中遍历对象所需要的时间，不过其时间成本相比而言比较低。

**四、综合实例**

```
1
2  public class StringTest
3  {
4   public static void main(String[] args)
5   {
6   /**
7   * 情景一：字符串池 * JAVA虚拟机(JVM)中存在着一个字符串池，其中保存着很多String对象; * 并且可以被共享使用，因此它提高了效率。 * 由于String类是final的，它的值一经创建就不可改变。 * 字符串池由String类维护，我们可以调用intern()方法来访问字符串池。
8   */
9   String s1 = "abc";
10   //↑ 在字符串池创建了一个对象
11   String s2 = "abc";//↑ 字符串pool已经存在对象“abc”(共享),所以创建0个对象，累计创建一个对象
```

```
System.out.println("s1 == s2 : " + (s1 == s2));
//↑ true 指向同一个对象，
System.out.println("s1.equals(s2) : " + (s1.equals(s2)));
//↑ true 值相等
// ↑-----------------------------------------------------over
/**
* 情景二：关于new String("") *
*/
String s3 = new String("abc");
//↑ 创建了两个对象，一个存放在字符串池中，一个存在与堆区中；
// ↑ 还有一个对象引用s3存放在栈中
String s4 = new String("abc");
//↑ 字符串池中已经存在“abc”对象，所以只在堆中创建了一个对象
System.out.println("s3 == s4 : " + (s3 == s4));
//↑false s3和s4栈区的地址不同，指向堆区的不同地址；
System.out.println("s3.equals(s4) : " + (s3.equals(s4)));
//↑true s3和s4的值相同
System.out.println("s1 == s3 : " + (s1 == s3));
//↑false 存放的地区多不同，一个栈区，一个堆区
System.out.println("s1.equals(s3) : " + (s1.equals(s3)));
//↑true 值相同
// ↑-----------------------------------------------------over
/**
* 情景三：
* 由于常量的值在编译的时候就被确定(优化)了。
* 在这里，"ab"和"cd"都是常量，因此变量str3的值在编译时就可以确定。
* 这行代码编译后的效果等同于：
* String str3 = "abcd";
*/
String str1 = "ab" + "cd";
//1个对象
String str11 = "abcd";
System.out.println("str1 = str11 : " + (str1 == str11));
//↑-----------------------------------------------------over
/**
* 情景四：
* 局部变量str2,str3存储的是存储两个拘留字符串对象(intern字符串对象)的地址。
* 第三行代码原理(str2+str3)：
* 运行期JVM首先会在堆中创建一个StringBuilder类，
* 同时用str2指向的拘留字符串对象完成初始化，
```

```
52   * 然后调用append方法完成对str3所指向的拘留字符串的合并，
53   * 接着调用StringBuilder的toString()方法在堆中创建一个String对象，
54   * 最后将刚生成的String对象的堆地址存放在局部变量str3中。
55   *
56   * 而str5存储的是字符串池中"abcd"所对应的拘留字符串对象的地址。
57   * str4与str5地址当然不一样了。 *
58   * 内存中实际上有五个字符串对象：
59   * 三个拘留字符串对象、一个String对象和一个StringBuilder对象。
60   */
61   String str2 = "ab";
62   //1个对象
63   String str3 = "cd"; //1个对象
64   String str4 = str2 + str3;
65   String str5 = "abcd";
66   System.out.println("str4 = str5 : " + (str4 == str5));// false
67   // ↑---------------------------------------------------over
68   /**
69   * 情景五：
70   * JAVA编译器对string + 基本类型
71   * 常量 是当成常量表达式直接求值来优化的。
72   * 运行期的两个string相加，会产生新的对象的，存储在堆(heap) 中
73   */
74   String str6 = "b";
75   String str7 = "a" + str6;
76   String str67 = "ab";
77   System.out.println("str7 = str67 : " + (str7 == str67));
78   //↑str6为变量，在运行期才会被解析。
79   final String str8 = "b";
80   String str9 = "a" + str8;
81   String str89 = "ab";
82   System.out.println("str9 = str89 : " + (str9 == str89));
83   //↑str8为常量变量，编译期会被优化
84   // ↑---------------------------------------------------over
85   }
86 }
```

运行结果：

s1 == s2 : true

s1.equals(s2) : true

s3 == s4 : false

s3.equals(s4) : true

s1 == s3 : false
s1.equals(s3) : true
str1 = str11 : true
str4 = str5 : false
str7 = str67 : false
str9 = str89 : true