74 为什么 String 被设计为是不可变的?

本课时我们主要讲解为什么 String 被设计为是不可变的？这样设计有什么好处?

String 是不可变的

我们先来介绍一下“String 是不可变的”这件事。在 Java 中，**字符串是一个常量**，我们一旦创建了一个 String 对象，就无法改变它的值，它的内容也就不可能发生变化（不考虑反射这种特殊行为）。

举个例子，比如我们给字符串 s 赋值为“lagou”，然后再尝试给它赋一个新值，正如下面这段代码所示：

```
String s = "lagou";

s = "la";
```

看上去好像是改变了字符串的值，但其背后实际上是新建了一个新的字符串“la”，并且把 s 的引用指向这个新创建出来的字符串“la”，原来的字符串对象“lagou”保持不变。

同样，如果我们调用 String 的 subString() 或 replace() 等方法，同时把 s 的引用指向这个新创建出来的字符串，这样都没有改变原有字符串对象的内容，因为这些方法只不过是**建了一个新的字符串而已。**例如下面这个例子：

```
String lagou = "lagou";

lagou = lagou.subString(0, 4);
```

代码中，利用 lagou.subString(0, 4) 会建立一个新的字符串“lago”这四个字母，比原来少了一个字母，但是这并不会影响到原有的“lagou”这个五个字母的字符串，也就是说，现在内存中**同时存在“lagou”和“lago”这两个对象。**

那这背后是什么原因呢？我们来看下 String 类的部分重要源码：

```
public final class String

    implements Java.io.Serializable, Comparable<String>, CharSequence {

    /** The value is used for character storage. */

    private final char value[];

        //...

}
```

首先，可以看到这里面有个非常重要的属性，即 **private final 的 char 数组**，数组名字叫 value。它存储着字符串的每一位字符，同时 value 数组是被 final 修饰的，也就是说，这个 value 一旦被赋值，引用就不能修改了；并且在 String 的源码中可以发现，除了构造函数之外，**并没有任何其他方法会修改 value 数组里面的内容**，而且 value 的权限是 private，外部的类也访问不到，所以最终使得 value 是不可变的。

那么有没有可能存在这种情况：其他类继承了 String 类，然后重写相关的方法，就可以修改 value 的值呢？这样的话它不就是可变的了吗？

这个问题很好，不过这一点也不用担心，因为 String 类是被 final 修饰的，所以这个 String 类是不会被继承的，因此**没有任何人可以通过扩展或者覆盖行为来破坏 String 类的不变性。**

这就是 String 具备不变性的原因。

String 不可变的好处

那我们就考虑一下，为什么当时的 Java 语言设计者会把它设计成这样？当然我们不是 String 的设计者本人，也无从考究他们当时的真实想法。不过我们可以思考一下，如果把 String 设计为不可变的，会带来哪些好处呢？我经过总结，主要有以下这**四个好处。**

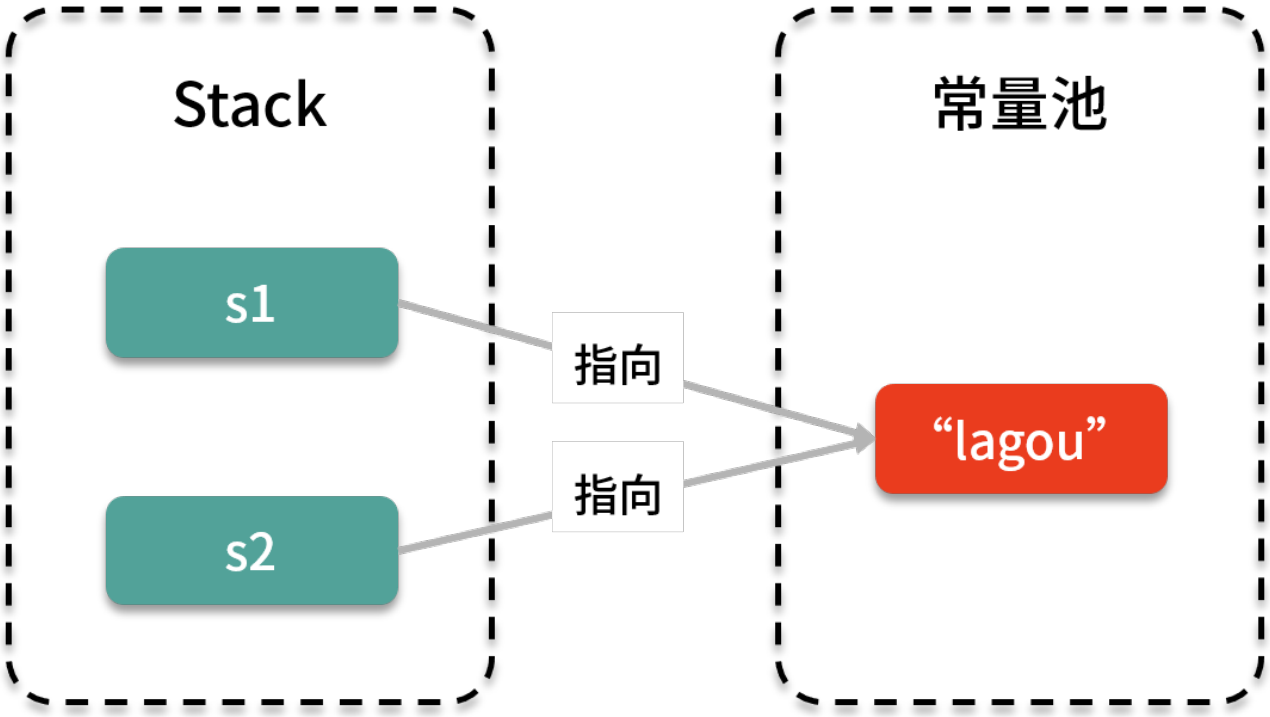
字符串常量池

String 不可变的第一个好处是可以使用**字符串常量池。**在 Java 中有字符串常量池的概念，比如两个字符串变量的内容一样，那么就会指向同一个对象，而不需创建第二个同样内容的新对象，例如：

```
String s1 = "lagou";

String s2 = "lagou";
```

其实 s1 和 s2 背后指向的都是常量池中的同一个“lagou”，如下图所示：



在图中可以看到，左边这两个引用都指向常量池中的同一个“lagou”，正是因为这样的机制，再加上 String 在程序中的应用是如此广泛，我们就可以**节省大量的内存空间**。

如果想利用常量池这个特性，这就要求 String 必须具备不可变的性质，否则的话会出问题，我们来看下面这个例子：

```
String s1 = "lagou";

String s2 = "lagou";

s1 = "LAGOU";

System.out.println(s2);
```

我们想一下，假设 String 对象是可变的，那么把 s1 指向的对象从小写的“lagou”修改为大写的“LAGOU”之后，s2 理应跟着变化，那么此时打印出来的 s2 也会是大写的：

```
LAGOU
```

这就和我们预期不符了，同样也就没办法实现字符串常量池的功能了，因为对象内容可能会不停变化，没办法再实现复用了。假设这个小写的“lagou”对象已经被许多变量引用了，如果使用其中任何一个引用更改了对象值，那么其他的引用指向的内容是不应该受到影响的。实际上，由于 String 具备不可变的性质，所以上面的程序依然会打印出小写的“lagou”，不

变性使得不同的字符串之间不会相互影响，符合我们预期。

用作 HashMap 的 key

String 不可变的第二个好处就是它可以很方便地用作 **HashMap （或者 HashSet） 的 key**。通常建议把**不可变对象作为 HashMap的 key**，比如 String 就很合适作为 HashMap 的 key。

对于 key 来说，最重要的要求就是它是不可变的，这样我们才能利用它去检索存储在 HashMap 里面的 value。由于 HashMap 的工作原理是 Hash，也就是散列，所以需要对象始终拥有相同的 Hash 值才能正常运行。如果 String 是可变的，这会带来很大的风险，因为一旦 String 对象里面的内容变了，那么 Hash 码自然就应该跟着变了，若再用这个 key 去查找的话，就找不回之前那个 value 了。

缓存 HashCode

String 不可变的第三个好处就是**缓存 HashCode**。

在 Java 中经常会用到字符串的 HashCode，在 String 类中有一个 hash 属性，代码如下：

```
/** Cache the hash code for the String */
private int hash;
```

这是一个成员变量，保存的是 String 对象的 HashCode。因为 String 是不可变的，所以对象一旦被创建之后，HashCode 的值也就不可能变化了，我们就可以把 HashCode 缓存起来。这样的话，以后每次想要用到 HashCode 的时候，**不需要重新计算，直接返回缓存过的 hash 的值就可以了**，因为它不会变，这样可以提高效率，所以这就使得字符串非常适合用作 HashMap 的 key。

而对于其他的不具备不变性的普通类的对象而言，如果想要去获取它的 HashCode ，就必须每次都重新算一遍，相比之下，效率就低了。

线程安全

String 不可变的第四个好处就是**线程安全**，因为具备**不变性的对象一定是线程安全的**，我们不需要对其采取任何额外的措施，就可以天然保证线程安全。

由于 String 是不可变的，所以它就可以非常安全地被多个线程所共享，这对于多线程编程而言非常重要，避免了很多不必要的同步操作。

总结

在本课时，我们先介绍了 String 是不可变的，然后介绍了 String 具备不可变性会带来的好处，分别是可以使用字符串常量池、适合作为 HashMap 的 key、缓存 HashCode 以及线程安全。

上一页 下一页