字符串常量池中是不会存储相同内容的字符串的。

字符串常量池是一个固定大小的Hashtable，如果放进StringPool的String非常多， 就会造成Hash冲突严重，从而导致链表会很长，而链表长了后直接会造成的影响就是当调用String. intern时性能会大幅下降。
使用一XX： StringTableSize可设置StringTable的长度
在jdk6中StringTable是固定的，就是1009的长度，对StringTableSize的大小设 置没有要求
在jdk7中，StringTable的长度默认值是60013
jdk8开始,1009是StringTable长度可设置的最小值

constant_pool（常量池），里面主要存放两大类常量：字面量（Literal）和符号引用（Symbolic References）。字面量如：文本字符串、final常量值等，符号引用包含下面三类：

- 类和接口的全限定名
- 字段的名称和描述符
- 方法的名称和描述符

我们写的每一个Java类被编译后，就会形成一份class文件（每个class文件都有一个class常量池）。 class文件中除了包含类的版本、字段、方法、接口等描述信息外，还有一项信息就是常量池(constant pool table)，用于存放编译器生成的各种字面量(Literal)和符号引用(Symbolic References)。

- 字面量包括：1.文本字符串 2.八种基本类型的值 3.被声明为final的常量等;
- 符号引用包括：1.类和方法的全限定名 2.字段的名称和描述符 3.方法的名称和描述符。

当JVM运行时，需要从常量池获得对应的符号引用，再在创建时或运行时解析、翻译到具体的内存地址之中；字面量就是普通的常量，一般在JVM运行时用作方法的参数。

下面是常量池中的项目类型：

类　型	标　志	描　述
CONSTANT_Utf8_info	1	UTF-8 编码的字符串
CONSTANT_Integer_info	3	整型字面量
CONSTANT_Float_info	4	浮点型字面量
CONSTANT_Long_info	5	长整型字面量
CONSTANT_Double_info	6	双精度浮点型字面量
CONSTANT_Class_info	7	类或接口的符号引用
CONSTANT_String_info	8	字符串类型字面量
CONSTANT_Fieldref_info	9	字段的符号引用
CONSTANT_Methodref_info	10	类中方法的符号引用
CONSTANT_InterfaceMethodref_info	11	接口中方法的符号引用
CONSTANT_NameAndType_info	12	字段或方法的部分符号引用
CONSTANT_MethodHandle_info	15	表示方法句柄
CONSTANT_MethodType_info	16	标识方法类型
CONSTANT_InvokeDynamic_info	18	表示一个动态方法调用点

运行时常量池

根据 The Java SE 8 Virtual Machine Specification，运行时常量池是 class文件中每一个类或接口 的**常量池表**的运行时表示形式，每一个运行时常量池都在Java虚拟机的方法区中分配，在加载类和接口到虚拟机后，就创建对应的运行时常量池。每一个class都有自己的运行时常量池。

这里有一点，就是在Java8中宣布取消永久代（方法区是堆的逻辑部分，真正实现的是永久代），用元空间（metaspace）来代替永久代，变化有以下几点：

- 移除了永久代（PermGen），替换为元空间（Metaspace）
- 永久代中的 class metadata 转移到了 native memory（本地内存，而不是虚拟机）
- **永久代中的 interned Strings 和 class static variables 转移到了 Java heap**
- 永久代参数 （PermSize MaxPermSize）—> 元空间参数（MetaspaceSize MaxMetaspaceSize）

我也挺疑惑的，既然取消了永久代，那方法区就不应该在jdk8官方文档中提到了吧，但是官方文档依然按照jdk7的方式介绍了方法区，而且两者的介绍竟然如出一辙。那按照官方说法，运行时常量池依然是在方法区中，现在方法区的实现被取消了，那究竟是在哪块内存区域呢？

jdk7：方法区==永久代；jdk8：方法区==元空间(把字符串常量池和静态变量移到了堆中)。方法区只是一个概念，jdk7以前的实现是永久代，jdk8的实现是元空间

字符串常量池

从jdk7开始，方法区中的字符串常量池移到了Java堆中，这也意味着字符串常量池是所有线程共享的。字符串常量池，顾名思义，就是用来存储String对象的pool，不过不像堆中可以存在多个相等值的String对象，字符串常量池中的不同值的String对象唯一，这算是JVM的一项优化。

举个例子，对于下面代码：

```
String s1 = new String("abc");

String s2 = "abc";

String s3 = "abc";

s1 == s2            // 值为false

s2 == s3            // 值为true
```

- 第一行创建了两个对象，**一个是new的String对象，放在了堆中的主要区域，另外一个也是String对象，放在了字符串常量池中。**第二行创建了一个**对象**，放在了运行时常量池中。
- 对于String s2 = "abc";会先在字符串常量池中判断有没有和s2通过equals方法返回true的String对象，有的话返回这个对象的引用给s2，没有则将这个对象添加到字符串常量池中，并返回该对象的引用给s2。

这里的"abc"就是我们所说的字符串字面量（string literals），(堆中)字符串字面量总是有一个来自字符串常量池的引用。因此字符串字面量不会被垃圾回收。The Java SE 8 Language Specification的3.10.5节专门对string literals进行了介绍：

一个string literal是一个String实例对象的引用，并且总是指向同一个String实例对象。这是为了在字符串常量池中共享唯一的实例而对string literals（更通俗的讲，就是常数表达式构成的字符串）做出的限制（interned），而实现的途径就是String的intern方法。

我们来看看String类中的intern方法：

```
/**
 * Returns a canonical representation for the string object.
 * <p>
 * A pool of strings, initially empty, is maintained privately by the
```

```
 * class {@code String}.
 * <p>
 * When the intern method is invoked, if the pool already contains a
 * string equal to this {@code String} object as determined by
 * the {@link #equals(Object)} method, then the string from the pool is
 * returned. Otherwise, this {@code String} object is added to the
 * pool and a reference to this {@code String} object is returned.
 * <p>
 * It follows that for any two strings {@code s} and {@code t},
 * {@code s.intern() == t.intern()} is {@code true}
 * if and only if {@code s.equals(t)} is {@code true}.
 * <p>
 * All literal strings and string-valued constant expressions are
 * interned. String literals are defined in section 3.10.5 of the
 * <cite>The Java™ Language Specification</cite>.
 *
 * @return  a string that has the same contents as this string, but is
 *          guaranteed to be from a pool of unique strings.
 */
public native String intern();
```

```
// 主程序
package testPackage;
class Test {
    public static void main(String[] args) {
        String hello = "Hello", lo = "lo";
        System.out.print((hello == "Hello") + " ");
        System.out.print((Other.hello == hello) + " ");
        System.out.print((other.Other.hello == hello) + " ");
        System.out.print((hello == ("Hel"+"lo")) + " ");
        System.out.print((hello == ("Hel"+lo)) + " ");
        System.out.println(hello == ("Hel"+lo).intern());
    }
}
class Other { static String hello = "Hello"; }
```

```
// 另一个包
package other;
public class Other { public static String hello = "Hello"; }
```

```
// 运行主程序输出的结果
true true true true false true
```

```
// 运行主程序输出的结果
true true true true false true
```

上述程序运行结果说明了六点：

- 同一个包、同一个类中的string字面量表示同一个String对象的引用
- 同一个包、不同的类中的string字面量表示同一个String对象的引用
- 不同的包、不同的类中的string字面量表示同一个String对象的引用
- 通过常量表达式得到的字符串是在编译期生成的，会被当成字面量
- 在运行期间拼接起来的String对象是新创建的，因此是不同的
- 通过显示的调用生成的字符串的intern方法，得到的结果与之前已存在且内容相同的字面量是同一个字符串对象

显然，"hel"+"lo"在编译时就会将"hel"、"lo"、"hello"三个常量添加到class文件的常量池中；而"hel"+lo，由于编译器不知道此时lo的具体值，所以只有"hel"被添加到class文件的常量池中。那class文件的这些字面量什么时候被加载到字符串常量池中呢？我们来看一个String字面量从java源文件到运行时的生命后期。

- 编译器编译源文件，将String常量放到常量池中，然后存储到生成的class文件中。
- 在类加载时期，常量池被JVM加载。
- JVM会自动调用String字面量的intern方法，常量池中的String对象被加载到字符串常量池中（前提是字符串常量池中没有与它值相同的String对象）

对于下面代码，你觉得在控制台上会打印什么？答案是true，这里c.substring(1)最终会创建一个新的String对象（并不会往字符串常量池中添加"we"，因为方法中没有出现"we"的字面量），所以可以肯定d指向的是堆中的对象，而不是字符串常量池中的。

```
String c = "qwe"
String d = c.substring(1)
d.intern()
String e = "we"
System.out.println(e==d);       // 打印true
```

为什么？因为如果一个String对象的intern方法在String字面量之后被调用，那这个String对象就会被添加到字符串常量池中，如果这个对象是new出来的，那么字符串常量池中存储的就是这个new出来的对象；如果你不手动调用intern方法，字符串字面量就会自动为我们往字符串常量池中添加String对象。这就使得在上面生命周期的第三步之前影响字符串常量池。

```
String c = "qwe";  // 字面量“qwe”进入字符串常量池
String d = c.substring(1); // 在堆中创建"we"对象
d.intern();        // 调用intern方法，因为字符串常量池中没有对应String，所以d进入里面
```

```
String e = "we";   // 现在才看到"we"的字面量，但是字符串常量池中d已经存在，所以返回了d的引用

System.out.println( e == d );  // returns true
```

这里我们将String e = "we"调到d.intern()之前，再看看打印结果：

```
String c = "qwe";
String d = c.substring(1);
String e = "we";   // 字符串常量池中还没有"we"，所以e进入里面
d.intern();        // 因为e已经存在于字符串常量池，所以什么也不做，返回e

System.out.println( e == d );  // returns false
System.out.println( e == d.intern() );  // returns true
```