# 1.String的基本特性

String：字符串，使用一对""引起来表示。

## 1.1String的两种创建方式

1. String sl = "hello"; //字面量的定义方式；
2. String s2 = new String("hello"); //new对象的方式

通过字面量的方式（不同于new）给一个String变量赋值，此时的字符串值声明在字符串常量池中。

## 1.2 String类的特点

* String类是final的,不可被继承
* String实现了Serializable接口,支持序列化
* 实现了Comparable接口,可以比较大小
* String在jdk8及以前内部存储结构定义为“private final **char value[]**”用于存储字符串数据。jdk9时改为**byte[]**,**目的是节省空间**。

jdk1.9的String结构示例：

|  |
| --- |
| public final class String implements java.io.Serializable, Comparable<String>,CharSequence {  @Stable  private final byte[] value;  } |

* 为什么改为 byte[] 存储？

String类的当前实现将字符存储在char数组中，每个字符使用两个字节(16位)。

从许多不同的应用程序收集的数据表明，字符串是堆使用的主要组成部分，而且大多数字符串对象只包含拉丁字符。这些字符只需要一个字节的存储空间，因此这些字符串对象的内部char数组中有一半的空间将不会使用。

之前 String 类使用 UTF-16 的 char[] 数组存储，现在改为 byte[] 数组 外加一个编码标志位存储，该编码标志将指定 String 类中 byte[] 数组的编码方式

结论：String再也不用char[] 来存储了，改成了byte [] 加上编码标记，节约了一些空间

同时基于String的数据结构，例如StringBuffer和StringBuilder也同样做了修改

## 1.3 String不可变的字符序列

简称：不可变性。

当对字符串重新赋值时，需要重新指定内存区域赋值，不能使用原有的value进行赋值。

比如：字符串进行连接操作、String的replace()方法都是重新指定内存区域赋值

* 通过字面量的方式（不同于new）给一个String变量赋值，此时的字符串值声明在字符串常量池中

|  |
| --- |
| /\*\*  \* String的基本使用:体现String的不可变性  \*/public class StringTest1 {  @Test  public void test1() {  String s1 = "abc";//字面量定义的方式，"abc"存储在字符串常量池中  String s2 = "abc";//s1,s2指向同一个abc  s1 = "hello";//s1指向字符串常量池新开辟了的hello，不影响原有的abc  System.out.println(s1 == s2);//判断地址： false  System.out.println(s1);//hello  System.out.println(s2);//abc  } |
| 0 ldc #2 <abc>  2 astore\_1  3 ldc #2 <abc>  5 astore\_2  6 ldc #3 <hello>  8 astore\_1  9 getstatic #4 <java/lang/System.out>  12 aload\_1  13 aload\_2  14 if\_acmpne 21 (+7)  17 iconst\_1  18 goto 22 (+4)  21 iconst\_0  22 invokevirtual #5 <java/io/PrintStream.println>  25 getstatic #4 <java/lang/System.out>  28 aload\_1  29 invokevirtual #6 <java/io/PrintStream.println>  32 getstatic #4 <java/lang/System.out>  35 aload\_2  36 invokevirtual #6 <java/io/PrintStream.println>  39 return  取字符串 “abc” 时，使用的是同一个符号引用：#2  取字符串 “hello” 时，使用的是另一个符号引用：#3 |

* String的拼接操作通过 StringBuilder 的 append() 方法完成

|  |
| --- |
| @Test  public void test2() {  String s1 = "abc";  String s2 = "abc";  s2 += "def";//s2 = s2+"def"  System.out.println(s2);//abcdef  System.out.println(s1);//abc  } |
| 0 ldc #2 <abc>  2 astore\_1  3 ldc #2 <abc>  5 astore\_2  6 new #7 <java/lang/StringBuilder>  9 dup  10 invokespecial #8 <java/lang/StringBuilder.<init>>  13 aload\_2  14 invokevirtual #9 <java/lang/StringBuilder.append>  17 ldc #10 <def>  19 invokevirtual #9 <java/lang/StringBuilder.append>  22 invokevirtual #11 <java/lang/StringBuilder.toString>  25 astore\_2  26 getstatic #4 <java/lang/System.out>  29 aload\_2  30 invokevirtual #6 <java/io/PrintStream.println>  33 getstatic #4 <java/lang/System.out>  36 aload\_1  37 invokevirtual #6 <java/io/PrintStream.println>  40 return |

|  |
| --- |
| @Test  public void test3() {  String s1 = "abc";  String s2 = s1.replace('a', 'm');  System.out.println(s1);//abc  System.out.println(s2);//mbc  } |

test2和test3的目的是说明 字符串在做链接和replace操作的时候是不会改变原本的字符串，而是生成一个新的字符串并返回。

|  |
| --- |
| package StringTable;  */\*\*  \* @author shkstart shkstart@126.com  \* @create 2020 23:44  \*/* public class StringExer {  String str = new String("good");  char[] ch = {'t', 'e', 's', 't'};   public void change(String str, char ch[]) {  str = "test ok";  ch[0] = 'b';  }   public void change(int a , int b){  int tmp = a;  a = b;  b = tmp;  System.*out*.println("a="+a+";b="+b);  }    public static void main(String[] args) {  StringExer ex = new StringExer();  ex.change(ex.str, ex.ch);  System.*out*.println(ex.str);*//good* System.*out*.println(ex.ch);*//best* int a = 5;  int b = 10;  ex.change(5,10);*//a=10;b=5* System.*out*.println("a="+a+";b="+b);*//a=5;b=10* } }  ex.str 传入 ex.change() 函数时，将 ex.str 引用String对象“good”的地址值传给了 change() 函数的str参数，此时 str 参数和 ex.str 都指向了 String 对象 ”good“；之后 str 参数指向了字面量”test ok“，而 ex.str 依旧保存着指向 String 类 ”good“ 的地址，故 ex.str 依旧输出 “good” 。 |

# 2.字符串常量池底层结构

* String的String Pool (常量池)是一个固定大小的Hashtable,默认值大小长度是1009。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。

str.intern()：如果字符串常量池中没有对应的 str 字符串，则在常量池中生成。

* 使用-XX：StringTableSize可设置StringTable的长度
* 在jdk6中StringTable是固定的，就是1009的长度，所以如果常量池中的字符串过多就会导致效率下降很快。StringTableSize设 置没有要求
* 在JDK7中，StringTable的长度默认值是60013，StringTablesize设置没有要求
* 在JDK8中，StringTable的长度默认值是60013，StringTable可以设置的最小值为1009
* 查看字符串常量池大小

|  |
| --- |
| /\*\*  \* -XX:StringTableSize=1009  \*  \* @author shkstart shkstart@126.com  \* @create 2020 23:53  \*/  public class StringTest2 {  public static void main(String[] args) {  // 测试StringTableSize参数  System.out.println("我来打个酱油");  try {  Thread.sleep(1000000);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  }  JVM 参数:  -XX:StringTableSize=6666  jinfo 查看变量值  jps  jinfo -flag StringTableSize 进程id |

* 测试不同 StringTable 长度下，程序的性能

|  |
| --- |
| /\*\*  \* -XX:StringTableSize=1009  \*  \* @author shkstart shkstart@126.com  \* @create 2020 23:53  \*/  public class StringTest2 {  public static void main(String[] args) {  BufferedReader br = null;  try {  br = new BufferedReader(new FileReader("words.txt"));  long start = System.currentTimeMillis();  String data;  while ((data = br.readLine()) != null) {  //如果字符串常量池中没有对应data的字符串的话，则在常量池中生成  data.intern();  }  long end = System.currentTimeMillis();  System.out.println("花费的时间为：" + (end - start));//1009:143ms 100009:47ms  } catch (IOException e) {  e.printStackTrace();  } finally {  if (br != null) {  try {  br.close();  } catch (IOException e) {  e.printStackTrace();  }  }  }  }  }  -XX:StringTableSize=1009 ：程序耗时 143ms  -XX:StringTableSize=100009 ：程序耗时 47ms |

# 3.String的内存分配

## 常量池的概念

在Java语言中有8种基本数据类型和一种比较特殊的类型String。这些类型为了使它们在运行过程中速度更快、更节省内存，都提供了一种常量池的概念。

常量池就类似一个Java系统级别提供的缓存。8种基本数据类型的常量池都是系统协调的，String类型的常量池比较特殊。

## 让String对象放在字符串常量池的两种方式

1. 直接使用双引号声明出来的String对象会直接存储在常量池中。

比如： String info = "abc";

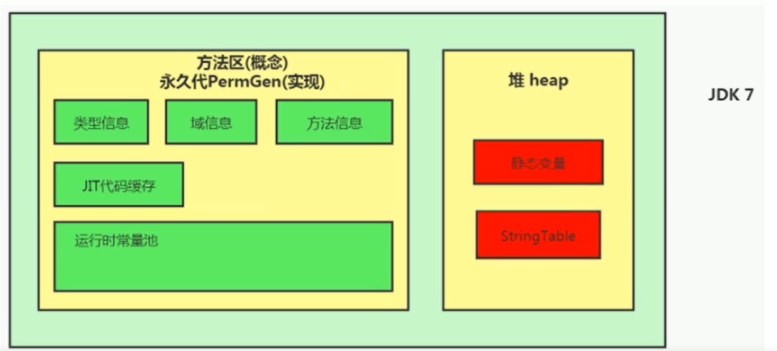
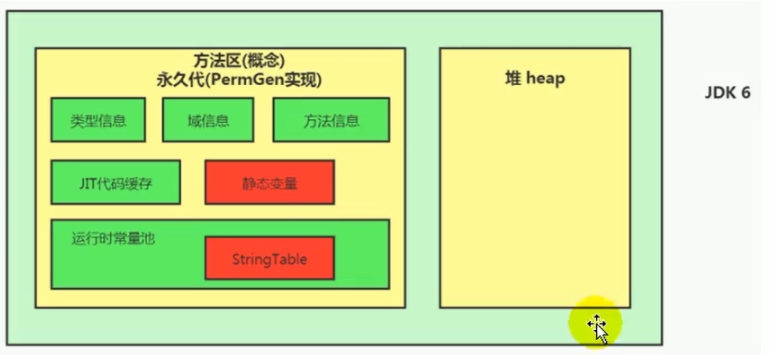
1. 如果不是用双引号声明的String对象，可以使用String提供的intern()方法，让String对象放进字符串常量池中。

## 字符串常量池的存储位置

* Java 6及以前，字符串常量池存放在永久代。
* Java 7中Oracle的工程师对字符串常量池的逻辑做了很大的改变，即将字符串常量池的位置调整到Java堆内。

1. 所有的字符串都保存在堆（Heap）中，和其他普通对象一样，这样可以让你在进行调优应用时仅需要调整堆大小就可以了。
2. 字符串常量池概念原本使用得比较多，但是这个改动使得我们有足够的理由让我们重新考虑在Java 7中使用String.intern()。

* Java8元空间，字符串常量在堆



* **为什么要调整位置？**

1. 永久代的默认比较小
2. 永久代垃圾回收频率低
3. 堆中空间足够大，字符串可被及时回收
4. 在JDK 7中，interned字符串不再在Java堆的永久代中分配，而是在Java堆的主要部分（称为年轻代和年老代）中分配，与应用程序创建的其他对象一起分配。此更改将导致驻留在主Java堆中的数据更多，驻留在永久生成中的数据更少，因此可能需要调整堆大小。

* 测试代码：验证jdk8中将字符串常量池放在了堆中

|  |
| --- |
| import java.util.HashSet; import java.util.Set;  */\*\*  \* jdk6中：  \* -XX:PermSize=6m -XX:MaxPermSize=6m -Xms6m -Xmx6m  \*  \* jdk8中：  \* -XX:MetaspaceSize=6m -XX:MaxMetaspaceSize=6m -Xms6m -Xmx6m  \* @author shkstart shkstart@126.com  \* @create 2020 0:36  \*/* public class StringTest3 {  public static void main(String[] args) {  *//使用Set保持着常量池引用，避免full gc回收常量池行为* Set<String> set = new HashSet<String>();  *//在short可以取值的范围内足以让6MB的PermSize或heap产生OOM了。* short i = 0;  while(true){  set.add(String.*valueOf*(i++).intern());  }  } } |
| 异常日志说：我真没骗你，字符串真的在堆中（JDK8） |

# 4.字符串基操

## 1.验证 字符串字面量在常量池中是唯一的

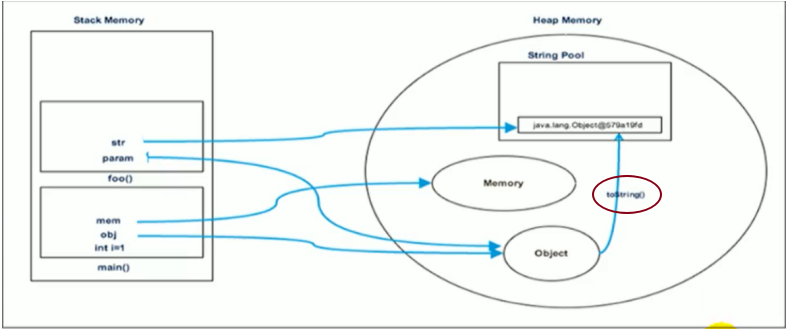
|  |
| --- |
| package StringTable;  */\*\*  \* @author shkstart shkstart@126.com  \* @create 2020 0:49  \*/* public class StringTest4 {  public static void main(String[] args) {  System.*out*.println();*//2330* System.*out*.println("1");*//2331* System.*out*.println("2");  System.*out*.println("3");  System.*out*.println("4");  System.*out*.println("5");  System.*out*.println("6");  System.*out*.println("7");  System.*out*.println("8");  System.*out*.println("9");  System.*out*.println("10");*//2340   //如下的字符串"1" 到 "10"不会再次加载* System.*out*.println("1");*//2341* System.*out*.println("2");*//2341* System.*out*.println("3");  System.*out*.println("4");  System.*out*.println("5");  System.*out*.println("6");  System.*out*.println("7");  System.*out*.println("8");  System.*out*.println("9");  System.*out*.println("10");*//2341* } } |
|  |
|  |
|  |

如上面代码和图所示，一个新的string字面量会在内存中多一个String类；但是如果这个String字面量加载过了，就不会再加载了！

## 2.Object的toString()方法

|  |
| --- |
| class Memory {  public static void main(String[] args) {  int i = 1;  Object obj = new Object();  Memory mem = new Memory();  mem.foo(obj);  }  private void foo(Object param) {  String str = param.toString();  System.out.println(str); //java.lang.Object@27c170f0  }  } |

显示的调用toString()方法在字符串常量池当中生成调用者对应的字符串,并且返回常量池当中的地址。



# 5.字符串拼接操作☆☆☆

1. **常量与常量的拼接结果在常量池**

原理是**编译期优化**。

编译生成的字节码文件中，已经将类似于字面量的赋值操作s1="a"+"b"+"c"替换为"abc"，如果先前常量池当中保存有s2="abc"，则s1、s2指向同一个字符串常量池地址

1. **常量池中不会存在相同内容的常量。**
2. **如果拼接的结果调用intern()方法，则主动将字符串常量池中还没有的字符串对象放入池中，并返回此对象地址。**
   1. **如果存在，则返回字符串在常量池中的地址**
   2. **如果字符串常量池中不存在该字符串，则在常量池中创建一份，并返回此对象的地址**
3. **只要其中有一个是变量（final修饰的常量除外），结果就在堆中（非字符串常量池）。变量拼接的原理是StringBuilder**

## 字符串**常量**拼接：编译期优化

示例代码1：

|  |
| --- |
| @Test  public void test1(){  String s1 = "a" + "b" + "c";//编译期优化：等同于"abc"  String s2 = "abc"; //"abc"一定是放在字符串常量池中，将此地址赋给s2  /\*  \* 最终.java编译成.class,再执行.class  \* String s1 = "abc";  \* String s2 = "abc"  \*/  System.out.println(s1 == s2); //true  System.out.println(s1.equals(s2)); //true  } |
| @Test public void test2(){  String s1 = new String("abc");   String s2 = new String("abc");     System.*out*.println(s1 == s2); *//false* System.*out*.println(s1.equals(s2)); *//true* } |
|  |

字符串常量分为两种：

1. 字符串字面量
2. final修饰的字符串变量

## 字符串**变量**拼接：结果在堆中

* 拼接前后，只要其中有一个是变量，结果就在堆中

调用 intern() 方法，则主动将字符串对象存入字符串常量池中，并将其地址返回

|  |
| --- |
| @Test  public void test2(){  String s1 = "javaEE";  String s2 = "hadoop";  String s3 = "javaEEhadoop";  String s4 = "javaEE" + "hadoop";//编译期优化  //如果拼接符号的前后出现了变量，则相当于在堆空间中new String()，具体的内容为拼接的结果：javaEEhadoop  String s5 = s1 + "hadoop";  String s6 = "javaEE" + s2;  String s7 = s1 + s2;  System.out.println(s3 == s4);//true  System.out.println(s3 == s5);//false  System.out.println(s3 == s6);//false  System.out.println(s3 == s7);//false  System.out.println(s5 == s6);//false  System.out.println(s5 == s7);//false  System.out.println(s6 == s7);//false    //intern():判断字符串常量池中是否存在javaEEhadoop值，如果存在，则返回常量池中javaEEhadoop的地址；  //如果字符串常量池中不存在javaEEhadoop，则在常量池中加载一份javaEEhadoop，并常量池中对象的地址。  String s8 = s6.intern();  System.out.println(s3 == s8);//true  } |

## 字符串变量拼接原理

|  |
| --- |
| @Test public void test3(){  String s1 = "a";  String s2 = "b";  String s3 = "ab"; */\* 如下的s1 + s2 的执行细节：(变量s是我临时定义的） ① StringBuilder s = new StringBuilder(); ② s.append("a") ③ s.append("b") ④ s.toString() --> 约等于 new String("ab")  补充：在jdk5.0之后使用的是StringBuilder,在jdk5.0之前使用的是StringBuffer  \*/* String s4 = s1 + s2;*//"ab"* System.*out*.println(s3 == s4);*//false* } |



**字符串变量有两种**：

1. 存储的是字面量的变量 如：String s1 = “a”
2. 存储的是字符串对象 如：String s2 = new String(“a”)

字符串变量相加，首先创建一个空的StringBuilder对象，然后用append方法挨个拼接，拼接完后用toString方法返回一个新的String对象。

* **字符串变量拼接的结果不会在字符串常量池中生成 该字符串。仅仅在堆中生成一个String对象。**

## StringBuilder的toString()方法

|  |
| --- |
| @Override public String toString() {  *// Create a copy, don't share the array* return new String(value, 0, count); } |

每次调用toString方法，会重新new一个String出来。(value是StringBuilder中的char[])

* 观察String类的构造方法

|  |
| --- |
| public String(char value[], int offset, int count) {  if (offset < 0) {  throw new StringIndexOutOfBoundsException(offset);  }  if (count <= 0) {  if (count < 0) {  throw new StringIndexOutOfBoundsException(count);  }  if (offset <= value.length) {  this.value = "".value;  return;  }  }  *// Note: offset or count might be near -1>>>1.* if (offset > value.length - count) {  throw new StringIndexOutOfBoundsException(offset + count);  }  this.value = Arrays.*copyOfRange*(value, offset, offset+count); } |
| public String(char value[]) {  this.value = Arrays.*copyOf*(value, value.length); } |
| public String(String original) {  this.value = original.value;  this.hash = original.hash; } |
| public String() {  this.value = "".value; } |

在java.lang.String的代码中,看最后一行，value并不是直接指向的,而是通过系统拷贝函数进行的内存操作，value 是个 char[] 数组;所以toString方法始终没有涉及方法区。

**总结：StringBuilder在调用toString()的时候，不会在常量池中生成字面量对象。**

## final常量拼接

* 如果拼接符号左右两边都是字符串常量（也就是字面量的意思）或常量引用(final修饰的变量也称为--常量)，则仍然使用编译期优化，即非StringBuilder的方式。

/\*

1. 字符串拼接操作不一定使用的是StringBuilder!

2.如果拼接符号左右两边都是字符串常量或常量引用，则仍然使用**编译期优化**，即非StringBuilder的方式。

3. 针对于final修饰类、方法、基本数据类型、引用数据类型的量的结构时，能使用上final的时候建议使用上。

\*/

@Test

public void test4(){

final String s1 = "a";

final String s2 = "b";

String s3 = "ab";

String s4 = s1 + s2;

System.out.println(s3 == s4);//true

}

//练习： @Test

public void test5(){

String s1 = "javaEEhadoop";

String s2 = "javaEE";

String s3 = s2 + "hadoop";

System.out.println(s1 == s3);//false

final String s4 = "javaEE";//s4:常量

String s5 = s4 + "hadoop";

System.out.println(s1 == s5);//true

}

## 拼接操作与append的效率对比

|  |
| --- |
| @Test public void test6(){  long start = System.*currentTimeMillis*();  method1(100000);*//3580  //method2(100000);//3* long end = System.*currentTimeMillis*();   System.*out*.println("花费的时间为：" + (end - start)); }  public void method1(int highLevel) {  String src = "";  for (int i = 0; i < highLevel; i++) {  src = src + "a";*//每次循环都会创建一个StringBuilder、String* } } public void method2(int highLevel){  *//只需要创建一个StringBuilder* StringBuilder src = new StringBuilder();  for (int i = 0; i < highLevel; i++) {  src.append("a");  } } |

体会执行效率：通过StringBuilder的append()的方式添加字符串的效率要远高于使用String的字符串拼接方式！

分析原因：

1. StringBuilder的append()的方式：
2. 自始至终中只创建过一个StringBuilder的对象
3. 使用String的字符串拼接方式：创建过多个StringBuilder和String的对象
4. 使用String的字符串拼接方式：
   1. 内存中由于创建了较多的StringBuilder和String的对象，内存占用更大；
   2. 如果进行GC，需要花费额外的时间。

改进的空间：

在实际开发中，如果基本确定要前前后后添加的字符串长度不高于某个限定值highLevel的情况下，建议使用构造器实例化：防止超过默认容量大小16而扩容，扩容会将原有的字符串赋值到新的容器当中，原有的容器变成垃圾需要回收。

StringBuilder s = new StringBuilder(highLevel); //new char[highLevel]

# 6.new String() 的说明

public class StringNewTest {

public static void main(String[] args) {

String str = new String("ab");

String str = new String("a") + new String("b");

}

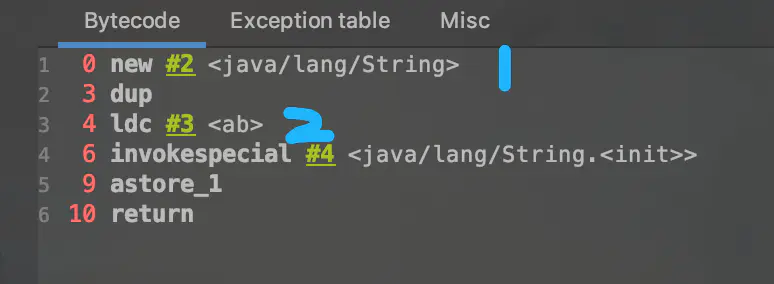
}

## new String("ab")会创建几个对象？

看字节码，就知道是两个：

1. 一个对象是：new关键字在堆空间创建的
2. 另一个对象是：字符串常量池中的对象"ab"。 **字节码指令：ldc 在常量池创建**

但注意：str此时指向的是堆空间当中的对象



## new String("a") + new String("b")呢？

对象1：new StringBuilder()，**变量拼接“+”操作肯定有StringBuilder**

对象2： new String("a")

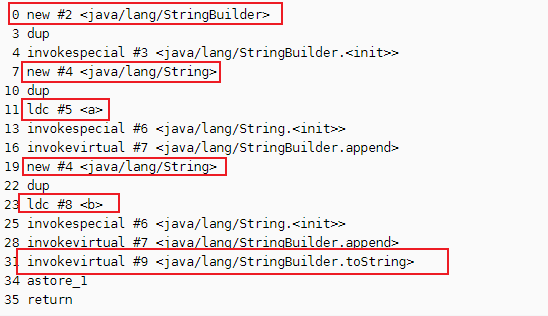
对象3： 常量池中的"a"

对象4： new String("b")

对象5： 常量池中的"b"

对象6 ：StringBuilder的toString生成的String("ab")

[StringBuilder的toString方法也是new String，但是不会在常量池中生成字符串](#_StringBuilder的toString()方法)(点击查看解释)



可以看出来在常量池中创建了 a和b，但是并没有ab

toString()的隐式调用(在字节码中StringBuilder拼接new String("a") + new String("b")之后调用)，表明其属于变量的拼接，结果ab存储在堆区(因为没有出现字节码指令lcd，所以在字符串常量池中并没有生成"ab")

# intern()的使用

|  |
| --- |
| 先来点逼格，看看官方文档  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 |

* intern()返回的就是字符串常量池中字符串字面量的地址。

## 关于 intern() 方法的说明

1. intern是一个native方法，调用的是底层C的方法
2. 字符串池最初是空的，由String类私有地维护。
3. 当一个堆中的String对象调用intern()方法的时候，intern方法会从字符串常量池中查询当前字符串对象所对应的字符串字面量是否存在

* 若不存在就会将当前字符串放入常量池中。
* 若存在就啥也不干

无论字符串常量池是否存在，intern()返回值都是常量池中字符串字面量的地址。

通俗点讲，Interned String就是确保字符串在内存里只有一份拷贝，这样可以节约内存空间，加快字符串操作任务的执行速度。注意，这个值会被存放在字符串内部池（String Intern Pool）。

**问：** 如何保证变量s指向的是字符串常量池中的数据呢？

有两种方式：

方式一： String s = "shkstart";//字面量定义的方式

方式二： 调用intern() 返回String类型的地址值就是常量池中的地址

String s = new String("shkstart").intern();

String s = new StringBuilder("shkstart").toString().intern();

## String.intern()的面试题☆☆☆☆

☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆

public class StringIntern {

public static void main(String[] args) {

|  |
| --- |
| **测试1：**  String s = new String("1");  s.intern();  String s2 = "1";  System.out.println(s == s2);//jdk6：false jdk7/8：false  **分析：**  String s = new String("1");  创建了两个对象，一个在堆中，一个是堆中的字符串常量池中的字面量"1".  s.intern();  此方法执行之前，常量池中已经有了字面量"1",那就啥也不干,s仍然指向堆空间字符串对象的的内存地址  String s2 = "1";  s2 指向字符串常量池已存在的"1"的内存地址  因此，无论是jdk6还是jdk1.7之后，s都不可能等于s2 |

|  |
| --- |
| 测试2：  String s = **new** String("1");  String s1 = s.intern(); //s指向堆空间"1"的内存地址 s1指向字符串常量池中  String s2 = "1";  //s2 指向字符串常量池已存在的"1"的内存地址  System.***out***.println(s1 == s2); //true |

☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆

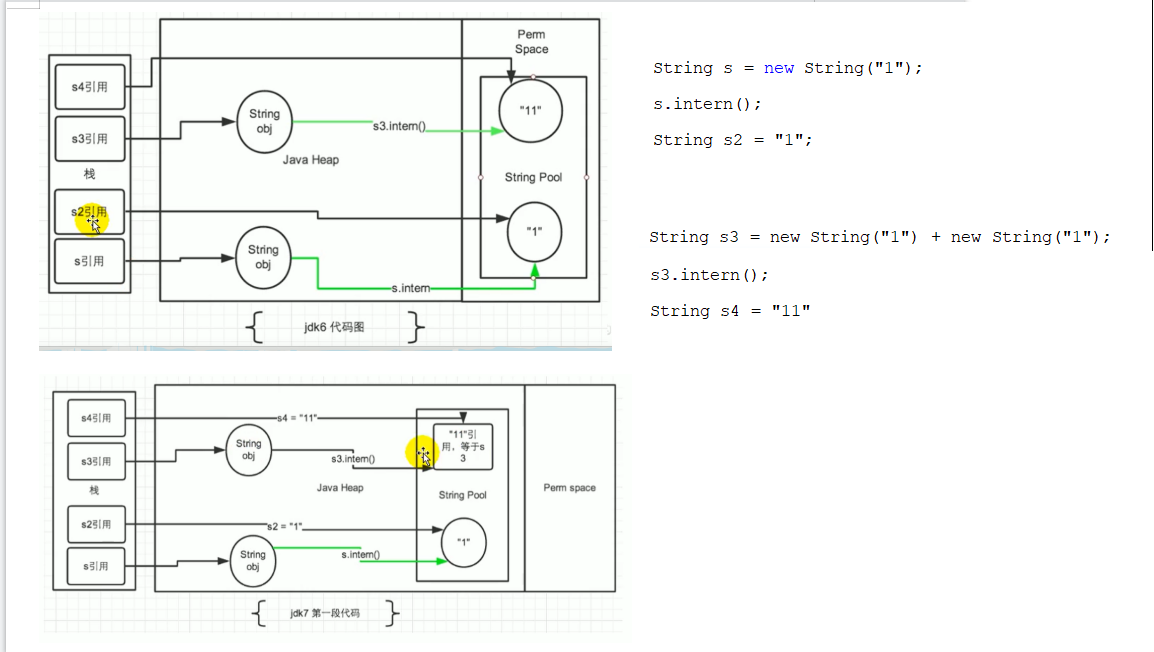
|  |
| --- |
| public class StringIntern {  public static void main(String[] args) {  String s = new String("1");  s.intern();//这方法其实没啥屌用，调用此方法之前，字符串常量池中已经存在"1"  String s2 = "1";  /\*  jdk6：false jdk7/8：false  因为 s 指向堆空间中的 "1" ，s2 指向字符创常量池中的 "1"  \*/  System.out.println(s == s2);//false  // 执行完下一行代码以后，字符串常量池中，是否存在"11"呢？答案：不存在！！  String s3 = new String("1") + new String("1");//s3变量记录的地址为：new String("11")  /\*  如何理解：jdk6:创建了一个新的对象"11",也就有新的地址。  jdk7:此时常量中并没有创建"11",而是在常量池中记录了指向堆空间中new String("11")的地址（节省空间）  \*/  s3.intern(); // 在字符串常量池中生成"11"。  String s4 = "11";//s4变量记录的地址：使用的是上一行代码代码执行时，在常量池中生成的"11"的地址  // jdk6：false jdk7/8：true  System.out.println(s3 == s4);  }  } |

}

}

☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆

图解：



* JDK6
* new String() 即在堆中
* str.intern() 则把字符串放入常量池中
* Jdk 1.7
* new String() 即在堆中
* str.intern() 则把字符串放入常量池中，出于节省空间的目的，如果 str 不存在于字符串常量池中，则将 str 在堆中的引用存储在字符串常量池中，没错，字符串常量池中存的是 str 在堆中的引用，所以 s3 == s4 为 true

Jdk1.6和jdk1.7及以后的区别：

Jdk1.7并没有移除永久代,只是把静态变量和字符串常量池移动到了堆中；

如果常量池中没有字符串字面量，调用S.intern()：

在jdk1.6版本，就会在永久代的字符串常量池中创建一个字面量。

在jdk1.7版本，由于字符串常量池就在堆内，因此没必要再在字符串常量池中创建字面量了，而是直接存储s的引用(地址)。所以不是在字符串常量池创建字面量，而是存储地址。因此s4==s3

|  |
| --- |
| **总结：**  String s = new String("1");//这种方式创建了两个对象，一个是字面量在常量池中，一个是字符串对象在堆中。  s.intern();//调用此方法之前，字符串常量池中**已经存在了"1"**.可理解为此方法没有生效，s保存的仍然是堆中的地址。但是此方法返回值是字面量"1"在常量池中的地址  String s1 = s.intern();  String s2 = "1";  具体表现为：s==s2 //false s1==s2//true  s.intern();//调用此方法之前，字符串常量池中**并不存在"1"**.此方法生效，s此时保存的为字面量"1"在常量池中的地址，返回值也是字面量"1"在常量池中的地址 |

☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆

面试题拓展：

public class StringIntern1 {

public static void main(String[] args) {

String s3 = new String("1") + new String("1");

String s4 = "11";//在字符串常量池中直接生成对象"11"

String s5 = s3.intern();//此时s3.intern全程OB视角，没作为。判断字符串常量池存在11，将s5指向字符串常量池

System.out.println(s3 == s4);//false

System.out.println(s5 == s4);//true }

}

**public** **class** 字符串 {

**public** **static** **void** main(String[] args) {

String s = **new** String("1");

String s1 = s.intern();

String s2 = "1";

System.***out***.println(s == s2);//false

System.***out***.println(s1 == s2);//true

String s3 = **new** String("1") + **new** String("1");

String s5 = s3.intern();

String s4 = "11";

System.***out***.println(s3 == s4);//true

System.***out***.println(s5 == s4);//true

}

## 总结String的intern()的使用

JDK1.6中，将这个字符串对象尝试放入串池。

如果串池中有，则并不会放入。返回已有的串池中的对象的地址

如果没有，会把此**对象**复制一份，放入串池，并返回串池中的对象地址

JDK1.7起，将这个字符串对象尝试放入串池。

如果串池中有，则并不会放入。返回已有的串池中的对象的地址

如果没有，则会把**对象的引用地址**复制一份，放入串池，并返回串池中的引用地址

## intern练习题1：

public class StringExer1 {

public static void main(String[] args) {

//String x = "ab";

String s = new String("a") + new String("b");//new String("ab")

//在上一行代码执行完以后，字符串常量池中并没有"ab"

String s2 = s.intern();//jdk6中：在串池中创建一个字符串"ab"

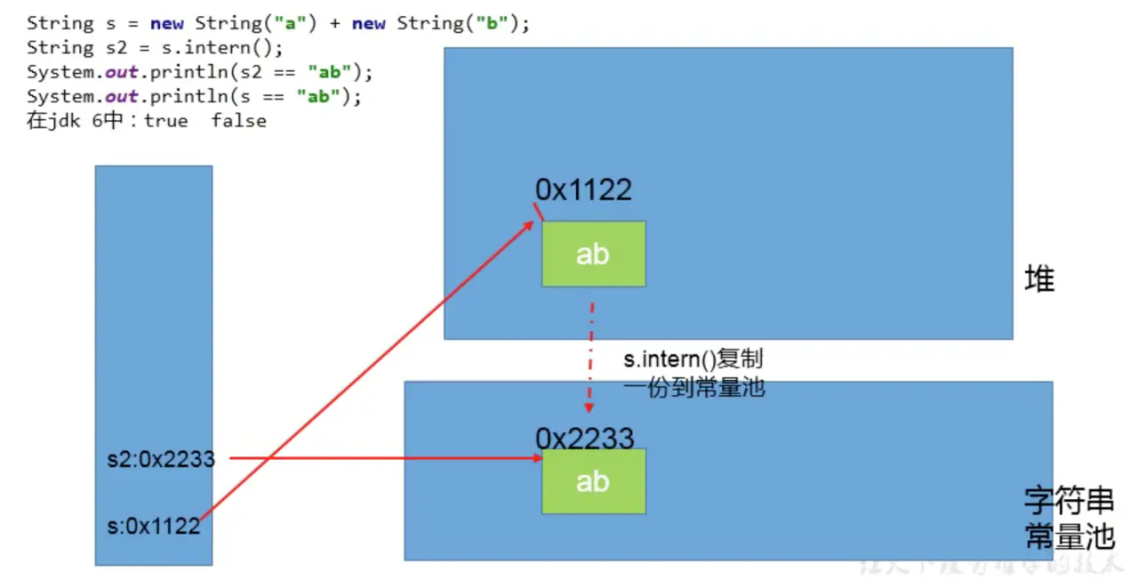
System.out.println(s2 == "ab");//jdk6:true jdk8:true

System.out.println(s == "ab");//jdk6:false jdk8:true }

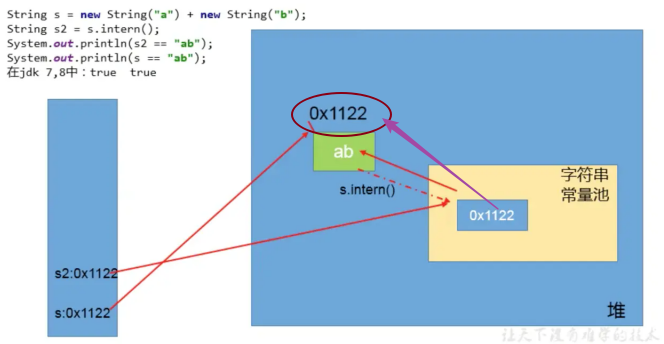
}

**图解：**

JDK6：

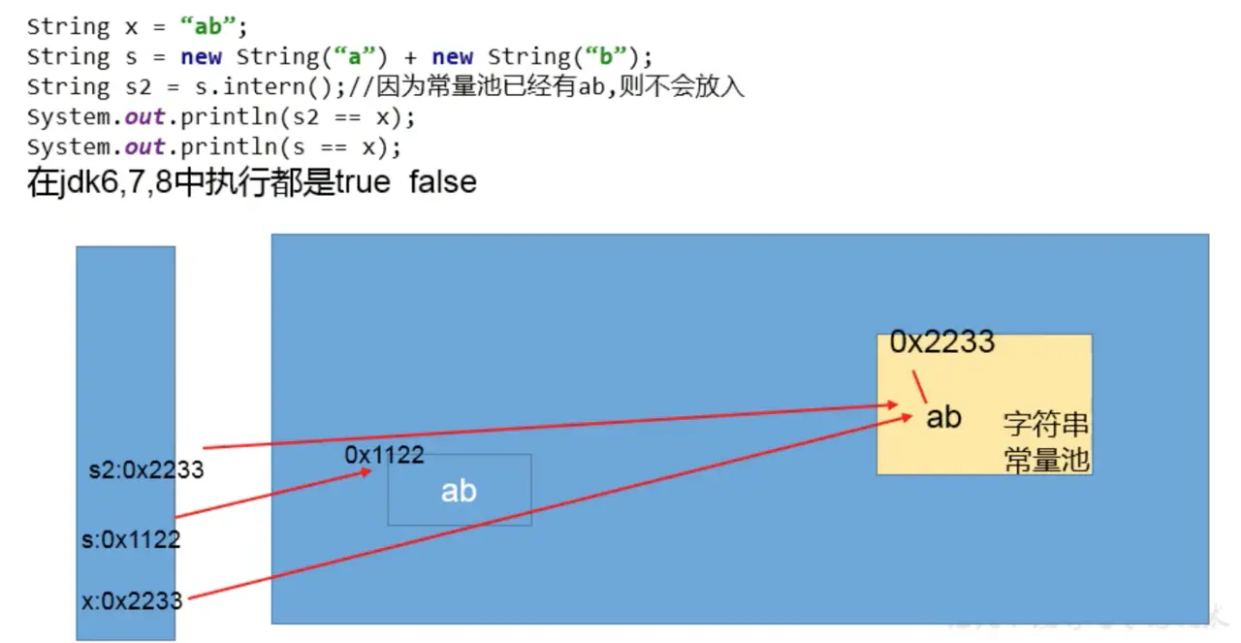


 JDK7/8：



放进常量池后，刷新堆中的值由对象地址变成了字符串常量池的地址。

## intern练习题2：



## intern练习题3：

public class StringExer2 {

public static void main(String[] args) {

String s1 = new String("ab");//执行完以后，会在字符串常量池中会生成"ab"。但是指向堆aa// String s1 = new String("a") + new String("b");////执行完以后，不会在字符串常量池中会生成"ab"

s1.intern();//检查字符串常量池当中有ab,有了就全程OB视角，不作为

String s2 = "ab";//指向字符串常量池已经存在的ab

System.out.println(s1 == s2); //false

}

}

## 验证intern copy的是地址

|  |
| --- |
| public class StringIntern2 {  *// 对象内存地址可以使用System.identityHashCode(object)方法获取* public static void main(String[] args) {  String s1 = new String("a") + new String("b");*//执行完以后，不会在字符串常量池中会生成"ab"* System.*out*.println(System.*identityHashCode*(s1));  s1.intern();  System.*out*.println(System.*identityHashCode*(s1));  String s2 = "ab";  System.*out*.println(System.*identityHashCode*(s2));  System.*out*.println(s1 == s2); *// true* } }  */\* 程序运行结果  21685669  21685669  21685669  true \*/* |

## intern()效率测试

大的网站平台，需要内存中存储大量的字符串。比如社交网站，很多人都存储：北京市、海淀区等信息。这时候如果字符串都调用 intern（）方法，就会明显降低内存的大小（原因是将字符串从堆中转移到了元空间了，元空间不在JVM内存）。

示例代码：

/\*\*

\* 使用intern()测试执行效率：空间使用上

\*

\* 结论：对于程序中大量存在存在的字符串，**尤其其中存在很多重复字符串时，使用intern()可以节省内存空间。**

\*

\*/public class StringIntern2 {

static final int MAX\_COUNT = 1000 \* 10000;

static final String[] arr = new String[MAX\_COUNT];

public static void main(String[] args) {

Integer[] data = new Integer[]{1,2,3,4,5,6,7,8,9,10};

long start = System.currentTimeMillis();

for (int i = 0; i < MAX\_COUNT; i++) {// arr[i] = new String(String.valueOf(data[i % data.length]));

arr[i] = new String(String.valueOf(data[i % data.length])).intern();

}

long end = System.currentTimeMillis();

System.out.println("花费的时间为：" + (end - start));

try {

Thread.sleep(1000000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.gc();

}

}

# StrtingTable的垃圾回收

示例代码：

-XX:+PrintStringTableStatistics展示字符串常量池信息

/\*\*

\* String的垃圾回收:

\* -Xms15m -Xmx15m -XX:+PrintStringTableStatistics -XX:+PrintGCDetails

\*

\*/public class StringGCTest {

public static void main(String[] args) {// for (int j = 0; j < 100; j++) {// String.valueOf(j).intern();// }

//发生垃圾回收行为

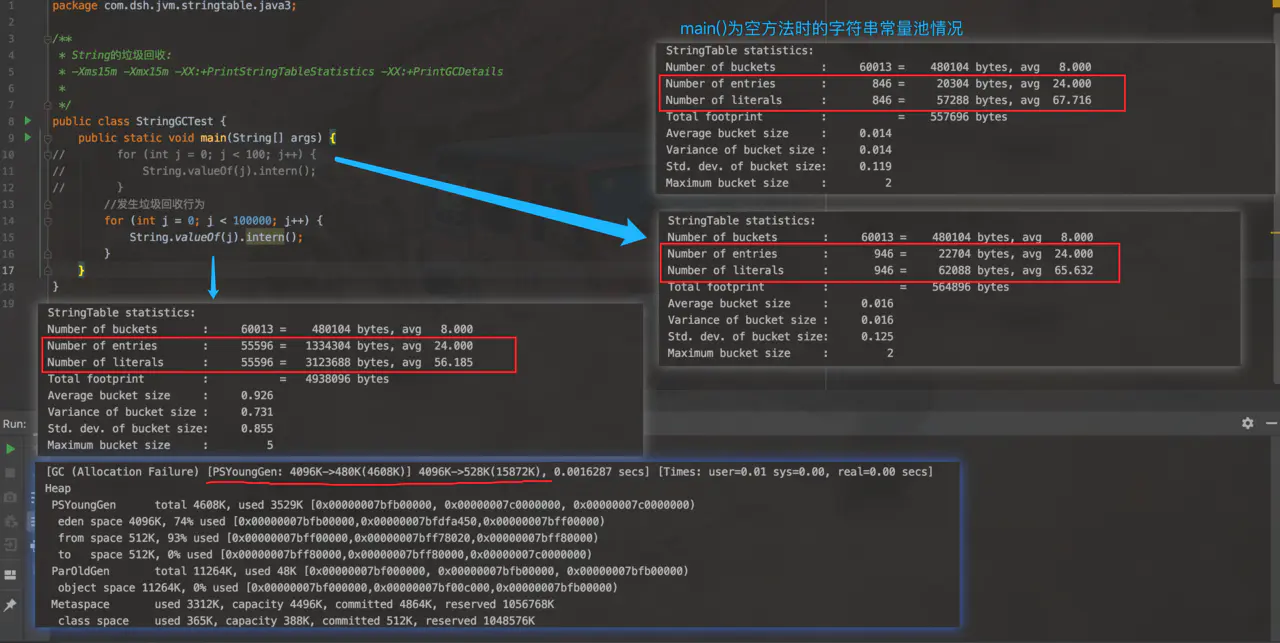
for (int j = 0; j < 100000; j++) {

String.valueOf(j).intern();

}

}

}



# G1中的String去重操作

声明：去重自然是对堆上数据的去重。而字符串常量池中的数据本身就不允许重复。

背景：对许多Java应用（有大的也有小的）做的测试得出以下结果：

➢堆存活数据集合里面String对象占了25%

➢堆存活数据集合里面重复的String对象有13.5%

➢String对象的平均长度是45

许多大规模的Java应用的瓶颈在于内存，测试表明，在这些类型的应用 里面，Java堆中存活的数据集合差不多258是String对象。更进一一步，这里面差不多一半String对象是重复的，重复的意思是说： string1. equals （string2）=true。堆上存在重复的string对象必然是一种内存的浪费。这个项目**将在G1垃圾收集器中实现自动持续对重复的String对象进行去重，这样就能避免浪费内存。**

## 去重实现步骤

➢当垃圾收集器工作的时候，会访问堆上存活的对象。对每一个访问的对象都会检查是否是候选的要去重的String对象。

➢如果是，把这个对象的一个引用插入到队列中等待后续的处理。一个去重的线程在后台运行，处理这个队列。处理队列的一个元素意味着从队列删除这个元素，然后尝试去重它引用的String对象。

➢使用一个hashtable来记录所有的被String对象使用的不重复的char数组。 当去重的时候，会查这个hashtable，来看堆上是否已经存在一个一模一样的char数组。

➢如果存在，String对象会被调整引用此共享数组，释放对原来的数组的引用，最终会被垃圾收集器回收掉。

➢如果查找失败，char数组会被插入到hashtable，这样以后的时候就可以共享这个数组了。

## 命令行选项

➢UseStringDeduplication （bool） ：开启String去重，默认是不开启的，需要手动开启。

➢PrintStringDedupl icationStatistics （bool） ：打印详细的去重统计信息，

➢StringDedupl icationAgeThreshold （uintx） ：达到这个年龄的string对象被认.为是去重的候选对象