

String的基本特性

- String：字符串，使用一对“”引起来表示。
- String声明为final的，不可被继承
- String实现了Serializable接口：表示字符串是支持序列化的。
实现了Comparable接口：表示String可以比较大小
- String在jdk 8及以前内部定义了final char[] value用于存储字符串数据。jdk 9时改为byte[]

更换的原因：

Motivation

<http://openjdk.java.net/jeps/254>

The current implementation of the String class stores characters in a char array, using two bytes (sixteen bits) for each character. Data gathered from many different applications indicates that strings are a major component of heap usage and, moreover, **that most String objects contain only Latin-1 characters. Such characters require only one byte of storage, hence half of the space in the internal char arrays of such String objects is going unused.**

Description

We propose to **change the internal representation of the String class from a UTF-16 char array to a byte array plus an encoding-flag field.** The new String class will store characters encoded either as ISO-8859-1/Latin-1 (one byte per character), or as UTF-16 (two bytes per character), based upon the contents of the string. The encoding flag will indicate which encoding is used.

经过大量数据显示，通常只存拉丁文，也就是 byte就够了，用 char 就有一般空间浪费了
所以，修改为了 byte[] 并且加入了字符码标识，如果是 UTF-16 就继续用 2个 byte

既然在 java9之后，String进行了修改，那么 StringBuffer 和 StringBuilder 做出了同样的修改

String：代表不可变的字符序列。简称：不可变性。常量池中存在且唯一

- 当对字符串重新赋值时，需要重写指定内存区域赋值，不能使用原有的value进行赋值。
- 当对现有的字符串进行连接操作时，也需要重新指定内存区域赋值，不能使用原有的value进行赋值。
- 当调用String的replace() 方法修改指定字符或字符串时， 也需要重新指定内存区域赋值， 不能使用原有的value进行赋值。

通过字面量的方式(区别于new) 给一个字符串赋值， 此时的字符串值声明在字符串常量池中。

```
1 import org.junit.Test;
2
3 /**
4  * String的基本使用:体现String的不可变性
5  *
6  * @author shkstart shkstart@126.com
7  * @create 2020 23:42
8  */
9 public class StringTest1 {
10     @Test
11     public void test1() {
12         String s1 = "abc";//字面量定义的方式, "abc"存储在字符串常量池中
13         String s2 = "abc";
14         s1 = "hello";
15
16         System.out.println(s1 == s2);//判断地址: true --> false
17
18         System.out.println(s1);//
19         System.out.println(s2);//abc
20
21     }
22
23     @Test
24     public void test2() {
25         String s1 = "abc";
26         String s2 = "abc";
27         s2 += "def";
28         System.out.println(s2);//abcdef
29         System.out.println(s1);//abc
```

```

30     }
31
32     @Test
33     public void test3() {
34         String s1 = "abc";
35         String s2 = s1.replace('a', 'm');
36         System.out.println(s1); //abc
37         System.out.println(s2); //mbc
38     }
39 }

```

练习题:

```

1  /**
2   * @author shkstart  shkstart@126.com
3   * @create 2020  23:44
4   */
5  public class StringExer {
6      String str = new String("good");
7      char[] ch = {'t', 'e', 's', 't'};
8
9      public void change(String str, char ch[]) {
10         str = "test ok";
11         ch[0] = 'b';
12     }
13
14     public static void main(String[] args) {
15         StringExer ex = new StringExer();
16         ex.change(ex.str, ex.ch);
17         System.out.println(ex.str); //good
18         System.out.println(ex.ch); //best
19     }
20
21 }

```

字符串常量池中是不会存储相同内容的字符串的。

- String的String Pool是一个固定大小的Hashtable，默认值大小长度是1009。如果放进String Pool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 **-XX: StringTableSize**可设置StringTable的长度
- 在jdk 6中String Table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。String Table Size设置没有要求
- 在jdk 7中，String Table的长度默认值是60013;
- jdk 8中 1009是可设置的最小值。

```
D:\developer_tools\jdk\jdk1.8.0_131\bin\java.exe ...
```

```
Error: Could not create the Java Virtual Machine.
```

```
Error: A fatal exception has occurred. Program will exit.
```

```
StringTable size of 1000 is invalid; must be between 1009 and 2305843009213693951
```

代码示例:

```
1 import java.io.FileWriter;
2 import java.io.IOException;
3
4 /**
5  * 产生10万个长度不超过10的字符串，包含a-z,A-Z
6  * @author shkstart shkstart@126.com
7  * @create 2020 23:58
8  */
9 public class GenerateString {
10     public static void main(String[] args) throws IOException {
11         FileWriter fw = new FileWriter("words.txt");
12
13         for (int i = 0; i < 100000; i++) {
14             //1 - 10
15             int length = (int)(Math.random() * (10 - 1 + 1) + 1);
16             fw.write(getString(length) + "\n");
17         }
18
19         fw.close();
20     }
21
22     public static String getString(int length){
```

```

23     String str = "";
24     for (int i = 0; i < length; i++) {
25         //65 - 90, 97-122
26         int num = (int)(Math.random() * (90 - 65 + 1) + 65) + (int)(Math.ra
27         str += (char)num;
28     }
29     return str;
30 }
31 }
32

```

上面生成字符串, 下面开始使用;

证明长度小, 链表长, 性能低

```

1  import java.io.BufferedReader;
2  import java.io.FileReader;
3  import java.io.IOException;
4
5  /**
6   * -XX:StringTableSize=1009
7   * @author shkstart shkstart@126.com
8   * @create 2020 23:53
9   */
10 public class StringTest2 {
11     public static void main(String[] args) {
12         //测试StringTableSize参数
13         // System.out.println("我来打个酱油");
14         // try {
15         //     Thread.sleep(1000000);
16         // } catch (InterruptedException e) {
17         //     e.printStackTrace();
18         // }
19
20         BufferedReader br = null;
21         try {
22             br = new BufferedReader(new FileReader("words.txt"));
23             long start = System.currentTimeMillis();
24             String data;

```

```

25         while((data = br.readLine()) != null){
26             data.intern(); //如果字符串常量池中没有对应data的字符串的话，则在常
27         }
28
29         long end = System.currentTimeMillis();
30
31         System.out.println("花费的时间为: " + (end - start)); //1009:143ms 1
32     } catch (IOException e) {
33         e.printStackTrace();
34     } finally {
35         if(br != null){
36             try {
37                 br.close();
38             } catch (IOException e) {
39                 e.printStackTrace();
40             }
41
42         }
43     }
44 }
45 }

```

String的内存分配

- 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。它的主要使用方法有两种。
- 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="haha";
- 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**。这个后面重点谈

存放位置变化 (可以参考 方法区笔记 6,7,8 运行时数据区的变化，方法区的实现不同)

- 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**。

>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。

>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Java7中使用String.intern() 。

- Java 8元空间， 字符串常量在堆

调整位置原因

- PermSize默认比较小
- 永久代垃圾回收频率低

代码示例

会爆出 堆 oom

```
1 import java.util.HashSet;
2 import java.util.Set;
3
4 /**
5  * jdk6中:
6  * -XX:PermSize=6m -XX:MaxPermSize=6m -Xms6m -Xmx6m
7  *
8  * jdk8中:
9  * -XX:MetaspaceSize=6m -XX:MaxMetaspaceSize=6m -Xms6m -Xmx6m
10 * @author shkstart shkstart@126.com
11 * @create 2020 0:36
12 */
13 public class StringTest3 {
14     public static void main(String[] args) {
15         //使用Set保持着常量池引用，避免full gc回收常量池行为
16         Set<String> set = new HashSet<String>();
17         //在short可以取值的范围内足以让6MB的PermSize或heap产生OOM了。
18         short i = 0;
19         while(true){
20             set.add(String.valueOf(i++).intern());
21         }
```

```
22     }
23 }
24
```

String的基本操作

Java语言规范里要求完全相同的字符串字面量，应该包含同样的Unicode字符序列(包含同一份码点序列的常量)，并且必须是指向同一个String类实例。

证明不会重复:

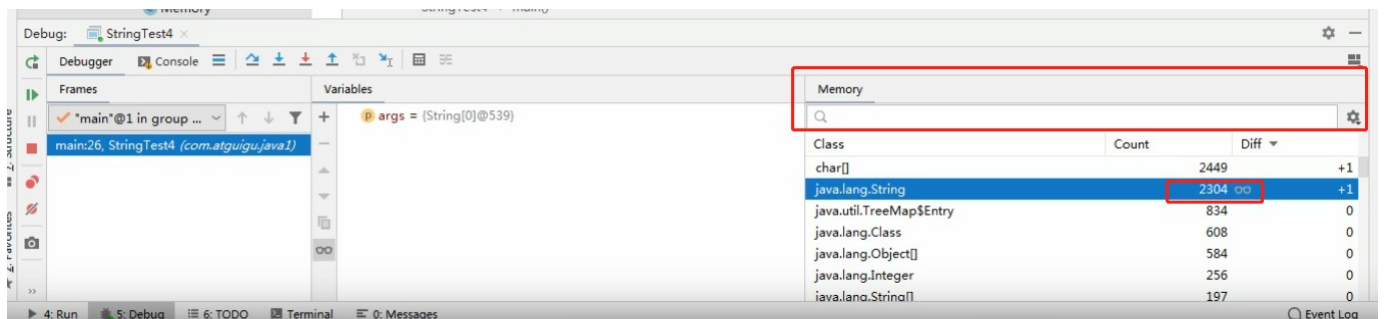
```
1  import org.junit.Test;
2
3  import javax.annotation.processing.SupportedSourceVersion;
4  import java.sql.SQLOutput;
5
6  /**
7   * @author shkstart shkstart@126.com
8   * @create 2020 0:49
9   */
10 public class StringTest4 {
11     public static void main(String[] args) {
12         System.out.println();//2293 换行操作
13         System.out.println("1");//2294
14         System.out.println("2");
15         System.out.println("3");
16         System.out.println("4");
17         System.out.println("5");
18         System.out.println("6");
19         System.out.println("7");
20         System.out.println("8");
21         System.out.println("9");
22         System.out.println("10");//2303
23         //如下的字符串"1" 到 "10"不会再次加载
24         System.out.println("1");//2304
```



```

25     System.out.println("2");//2304
26     System.out.println("3");
27     System.out.println("4");
28     System.out.println("5");
29     System.out.println("6");
30     System.out.println("7");
31     System.out.println("8");
32     System.out.println("9");
33     System.out.println("10");//2304
34 }
35 }
36

```



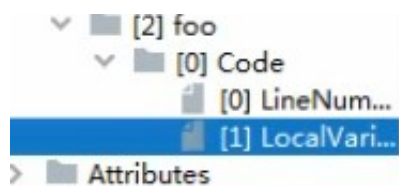
同样证明不会重复:

```

1  class Memory {
2      public static void main(String[] args) { //line 1
3          int i = 1; //line 2
4          Object obj = new Object(); //line 3
5          Memory mem = new Memory(); //line 4
6          mem.foo(obj); //line 5
7      } //line 9
8
9      private void foo(Object param) { //line 6
10         String str = param.toString(); //line 7
11         System.out.println(str);
12     } //line 8
13 }
14

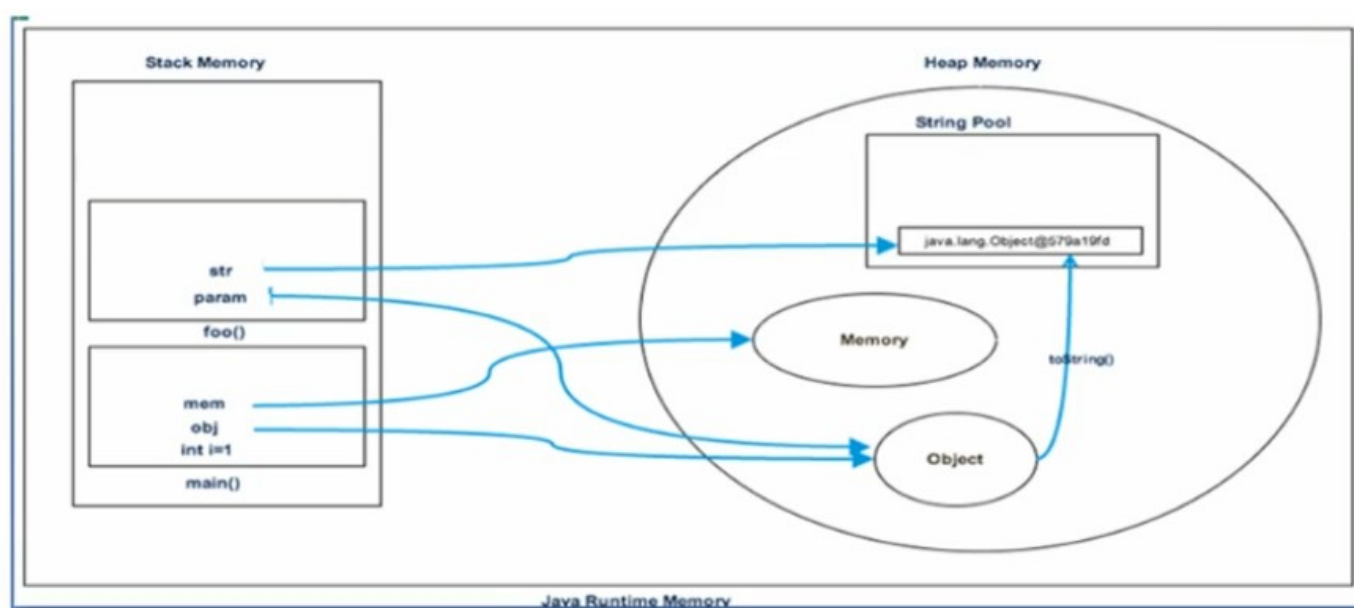
```

局部变量表:



Nr.	Start PC	Length	Index	
0	0	13	0	cp info #14 this
1	0	13	1	cp info #27 param
2	5	8	2	cp info #28 str

内存存储简视图:



A string is created in line 7, it goes in the **String Pool** in the heap space and a reference is created in the **foo()** stack space for it.

一个字符串在第七行被创建, 放入了字符串池, 并且有一个引用在 **foo()** 栈中指向;

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
2. 常量池中不会存在相同内容的常量。
3. 只要其中有一个是变量, 结果就在堆中。变量拼接的原理是 **StringBuilder**
4. 如果拼接的结果调用 **intern()** 方法, 则主动将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。

常见面试题

代码示例

```
1 import org.junit.Test;
2
3 /**
4  * 字符串拼接操作
5  * @author shkstart shkstart@126.com
6  * @create 2020 0:59
7  */
8 public class StringTest5 {
9     @Test
10     public void test1(){
11         String s1 = "a" + "b" + "c";//编译期优化: 等同于"abc"
12         String s2 = "abc"; //"abc"一定是放在字符串常量池中, 将此地址赋给s2
13         /*
14          * 最终.java编译成.class,再执行.class
15          * String s1 = "abc";
16          * String s2 = "abc"
17          */
18         System.out.println(s1 == s2); //true
19         System.out.println(s1.equals(s2)); //true
20     }
21
22     @Test
23     public void test2(){
24         String s1 = "javaEE";
25         String s2 = "hadoop";
26
27         String s3 = "javaEEhadoop";
28         String s4 = "javaEE" + "hadoop";//编译期优化
29         //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
30         String s5 = s1 + "hadoop";
31         String s6 = "javaEE" + s2;
32         String s7 = s1 + s2;
33
34         System.out.println(s3 == s4);//true
35         System.out.println(s3 == s5);//false
```

```

36     System.out.println(s3 == s6);//false
37     System.out.println(s3 == s7);//false
38     System.out.println(s5 == s6);//false
39     System.out.println(s5 == s7);//false
40     System.out.println(s6 == s7);//false
41     //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
42     //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
43     String s8 = s6.intern();
44     System.out.println(s3 == s8);//true
45 }
46
47 @Test
48 public void test3(){
49     String s1 = "a";
50     String s2 = "b";
51     String s3 = "ab";
52     /*
53     如下的s1 + s2 的执行细节：（变量s是我临时定义的）
54     ① StringBuilder s = new StringBuilder();
55     ② s.append("a")
56     ③ s.append("b")
57     ④ s.toString() --> 约等于 new String("ab")
58
59     补充：在jdk5.0之后使用的是StringBuilder,在jdk5.0之前使用的是StringBuffer
60     */
61     String s4 = s1 + s2;//
62     System.out.println(s3 == s4);//false
63 }
64 /*
65 1. 字符串拼接操作不一定使用的是StringBuilder!
66     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String
67 2. 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
68     */
69 @Test
70 public void test4(){
71     final String s1 = "a";
72     final String s2 = "b";
73     String s3 = "ab";
74     String s4 = s1 + s2;
75     System.out.println(s3 == s4);//true

```

```

76     }
77     //练习:
78     @Test
79     public void test5(){
80         String s1 = "javaEehadoop";
81         String s2 = "javaEE";
82         String s3 = s2 + "hadoop";
83         System.out.println(s1 == s3);//false
84
85         final String s4 = "javaEE";//s4:常量
86         String s5 = s4 + "hadoop";
87         System.out.println(s1 == s5);//true
88
89     }
90
91     /*
92     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
93     详情: ① StringBuilder的append()的方式: 自始至终中只创建过一个StringBuilder的对象
94           使用String的字符串拼接方式: 创建过多个StringBuilder和String的对象
95           ② 使用String的字符串拼接方式: 内存中由于创建了较多的StringBuilder和String
96
97     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
98           StringBuilder s = new StringBuilder(highLevel);//new char[highLevel]
99     */
100    @Test
101    public void test6(){
102
103        long start = System.currentTimeMillis();
104
105        //        method1(100000);//4014
106        method2(100000);//7
107
108        long end = System.currentTimeMillis();
109
110        System.out.println("花费的时间为: " + (end - start));
111    }
112
113    public void method1(int highLevel){
114        String src = "";
115        for(int i = 0;i < highLevel;i++){

```

```

116         src = src + "a";//每次循环都会创建一个StringBuilder、String
117     }
118     //        System.out.println(src);
119
120 }
121
122 public void method2(int highLevel){
123     //只需要创建一个StringBuilder
124     StringBuilder src = new StringBuilder();
125     for (int i = 0; i < highLevel; i++) {
126         src.append("a");
127     }
128     //        System.out.println(src);
129 }
130 }
131
132 如有理解困难，参考视频：
133 https://www.bilibili.com/video/BV1PJ411n7xZ?p=122

```

字符串变量拼接操作的底层原理

The screenshot displays the source code and the corresponding bytecode for a Java class named `StringTest5`.

Source Code:

```

@Test
public void test3(){
    String s1 = "a";
    String s2 = "b";
    String s3 = "ab";
    String s4 = s1 + s2;//
    System.out.println(s3 == s4);//
}

@Test
public void test4(){
    final String s1 = "a";
    final String s2 = "b";
    String s3 = "ab";
    String s4 = s1 + s2;
}

```

Bytecode View (for test3):

```

1  0 ldc #14 <a>
2  2 astore_1
3  3 ldc #15 <b>
4  5 astore_2
5  6 ldc #16 <ab>
6  8 astore_3
7  9 new #9 <java/lang/StringBuilder>
8 12 dup
9 13 invokespecial #10 <java/lang/StringBuilder.<init>>
10 16 aload_1
11 17 invokevirtual #11 <java/lang/StringBuilder.append>
12 20 aload_2
13 21 invokevirtual #11 <java/lang/StringBuilder.append>
14 24 invokevirtual #12 <java/lang/StringBuilder.toString>
15 27 astore_4
16 29 getstatic #3 <java/lang/System.out>
17 32 aload_3
18 33 aload_4
19 35 if_acmpne 42 (+7)

```

就是当有变量拼接的时候，就是 `new StringBuilder`，然后分别 `append`；jdk 5.0 之前用 `StringBuffer`

当时也并不一定，例如 `test4`，`String` 变量添加了 `final`，变量的形式，但是相当于常量了

拼接操作和append操作的效率

上述的 test5() 方法

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！

详情：① StringBuilder的append()的方式：自始至终中只创建过一个StringBuilder的对象

使用String的字符串拼接方式：创建过多个StringBuilder和String的对象

② 使用String的字符串拼接方式：内存中由于创建了较多的StringBuilder和String的对象，内存占用更大；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：

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

intern()的使用

```
java.lang.String  
@NotNull  
public String intern()
```

Returns a canonical representation for the string object.

A pool of strings, initially empty, is maintained privately by the class String.

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.

All literal strings and string-valued constant expressions are interned. String literals are defined in section 3.10.5 of the *The Java™ Language Specification*.

Returns: a string that has the same contents as this string, but is guaranteed to be from a pool of unique strings.

External annotations: `@org.jetbrains.annotations.NotNull`

 < 1.8 >

3167

@

```
@NotNull public native String intern();
```


- 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：
intern方法会从字符串常量池中查询当前字符串是否存在， 若不存在就会将当前字符串放入常量池中。

比如： String myInfo=new String("I love u") .intern() ;

- 也就是说， 如果在任意字符串上调用string.intern方法， 那么其返回结果所指向的那个类实例， 必须和直接以常量形式出现的字符串实例完全相同。

因此， 下列表达式的值必定是true：

("a" +"b"+"c") .intern()=="abc"

- 通俗点讲， InternedString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 InternPool) 。

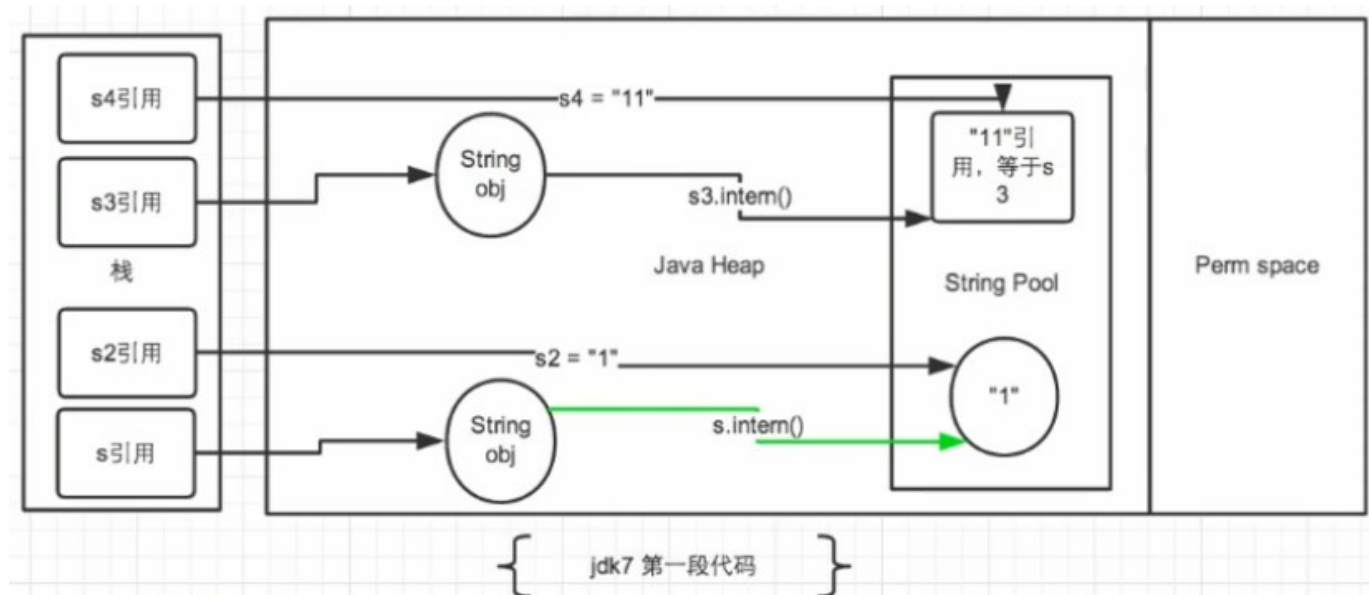
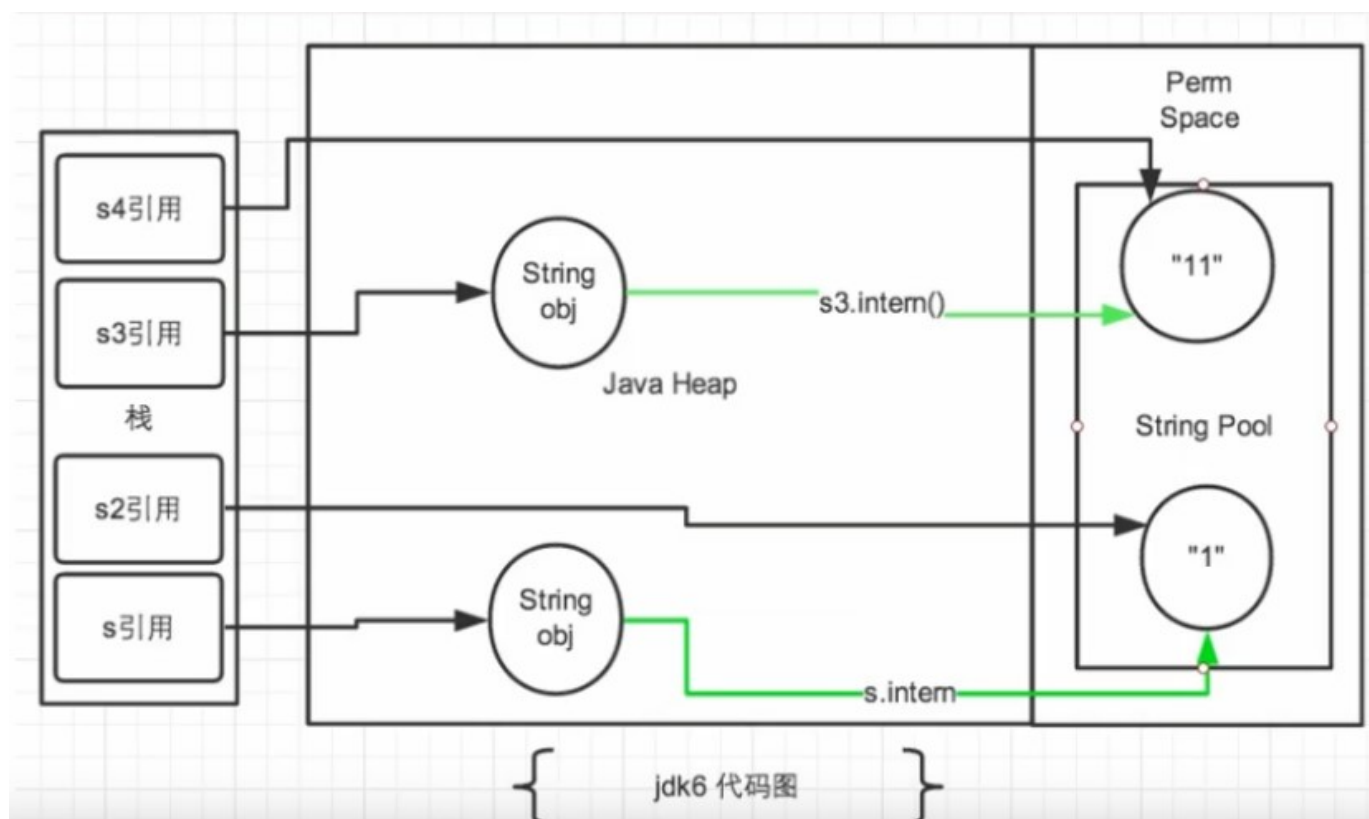
```
1  /**
2   * 如何保证变量s指向的是字符串常量池中的数据呢？
3   * 有两种方式：
4   * 方式一： String s = "shkstart";//字面量定义的方式
5   * 方式二： 调用intern()
6   *          String s = new String("shkstart").intern();
7   *          String s = new StringBuilder("shkstart").toString().intern();
8   *
9   * @author shkstart shkstart@126.com
10  * @create 2020 18:49
11  */
12 public class StringIntern {
13     public static void main(String[] args) {
14
15         String s = new String("1");
16         s.intern();//调用此方法之前，字符串常量池中已经存在了"1"
17         String s2 = "1";
18         System.out.println(s == s2);//jdk6: false   jdk7/8: false
19
20
21         String s3 = new String("1") + new String("1");//s3变量记录的地址为： new String("11")
22         //执行完上一行代码以后，字符串常量池中，是否存在"11"呢？答案：不存在！！
23         s3.intern();//在字符串常量池中生成"11"。如何理解：jdk6:创建了一个新的对象"11"
24         //                                jdk7:此时常量中并没有创建"11"
25         // 如果在此之前 new String("11"), 则 jdk 7/8返回 false; 否则返回 true
```



```

26 // 已经在常量池里了，intern方法也就没有用了；即使取返回值，
27 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
28 System.out.println(s3 == s4);//jdk6: false  jdk7/8: true
29 }
30
31
32 }

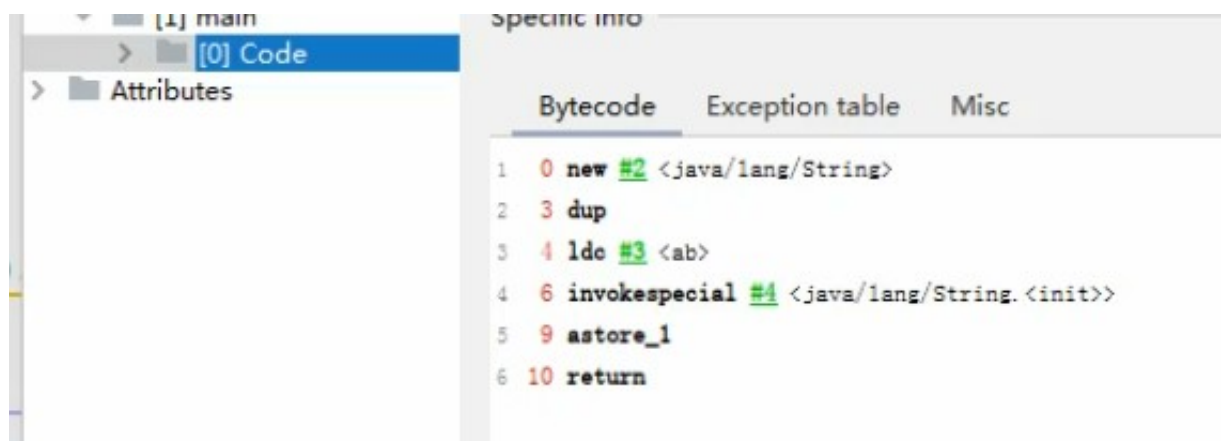
```



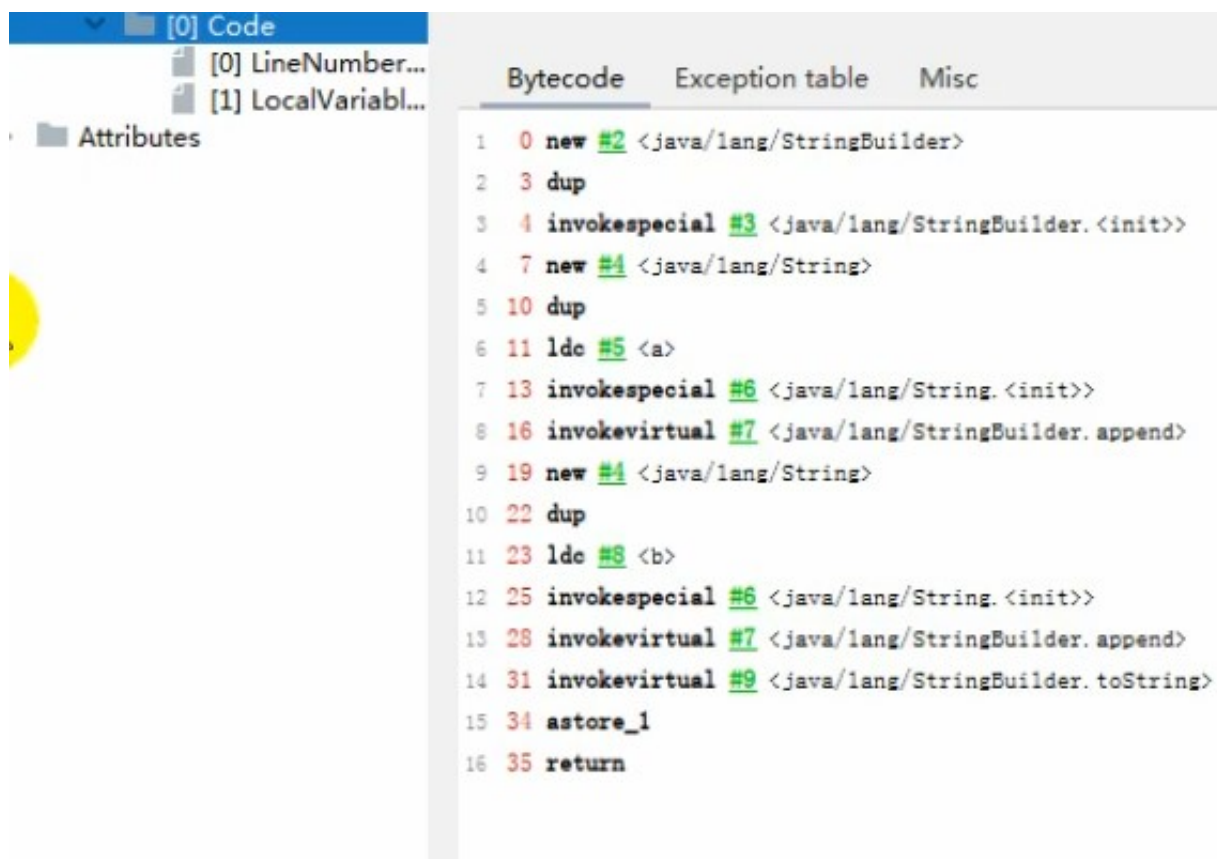
常见面试题

new String 创建了几个对象

```
1
2 /**
3  * 题目:
4  * new String("ab")会创建几个对象? 看字节码, 就知道是两个。
5  *     一个对象是: new关键字在堆空间创建的
6  *     另一个对象是: 字符串常量池中的对象"ab"。 字节码指令: ldc      !!!!!!!!!!!!!!!
7  *
8  *
9  * 思考:
10 * new String("a") + new String("b")呢?
11 * 对象1: new StringBuilder()
12 * 对象2: new String("a")
13 * 对象3: 常量池中的"a"
14 * 对象4: new String("b")
15 * 对象5: 常量池中的"b"
16 *
17 * 深入剖析: StringBuilder的toString():
18 *     对象6 : new String("ab")
19 *     强调一下, toString()的调用, 在字符串常量池中, 没有生成"ab" !!!!!!!!!!!!!!!
20 *
21 * @author shkstart shkstart@126.com
22 * @create 2020 20:38
23 */
24 public class StringNewTest {
25     public static void main(String[] args) {
26         // String str = new String("ab");
27
28         String str = new String("a") + new String("b");
29     }
30 }
31
```



扩展问题



注意, 扩展之后的问题里面 字符串常量池里面并没有 "ab"!

面试题:

上面的拓展的拓展



```

7      //StringIntern.java中练习的拓展:
8      String s3 = new String("1") + new String("1");//new String("11")
9      //执行完上一行代码以后，字符串常量池中，是否存在"11"呢？答案：不存在！！
10     String s4 = "11";//在字符串常量池中生成对象"11"
11     String s5 = s3.intern();
12     System.out.println(s3 == s4);//false
13     System.out.println(s5 == s4);//true
14 }
15 }

```

总结

总结String的intern() 的使用：

- Jdk 1.6 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

- Jdk 1.7 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

原因就是, 1.6之前 永久代, 1.7之后 常量池放到了 堆里面

练习题

1.

```

1  /**
2   * @author shkstart  shkstart@126.com
3   * @create 2020  20:17
4   */

```

```

5 public class StringExer1 {
6     public static void main(String[] args) {
7         String x = "ab";
8         String s = new String("a") + new String("b");//new String("ab")
9         //在上一行代码执行完以后，字符串常量池中并没有"ab"
10
11         String s2 = s.intern();//jdk6中：在串池中创建一个字符串"ab"
12         //jdk8中：串池中并没有创建字符串"ab",而是创建一个引用,
13
14         System.out.println(s2 == "ab");//jdk6:true   jdk8:true
15         System.out.println(s == "ab");//jdk6:false  jdk8:true
16     }
17 }
18

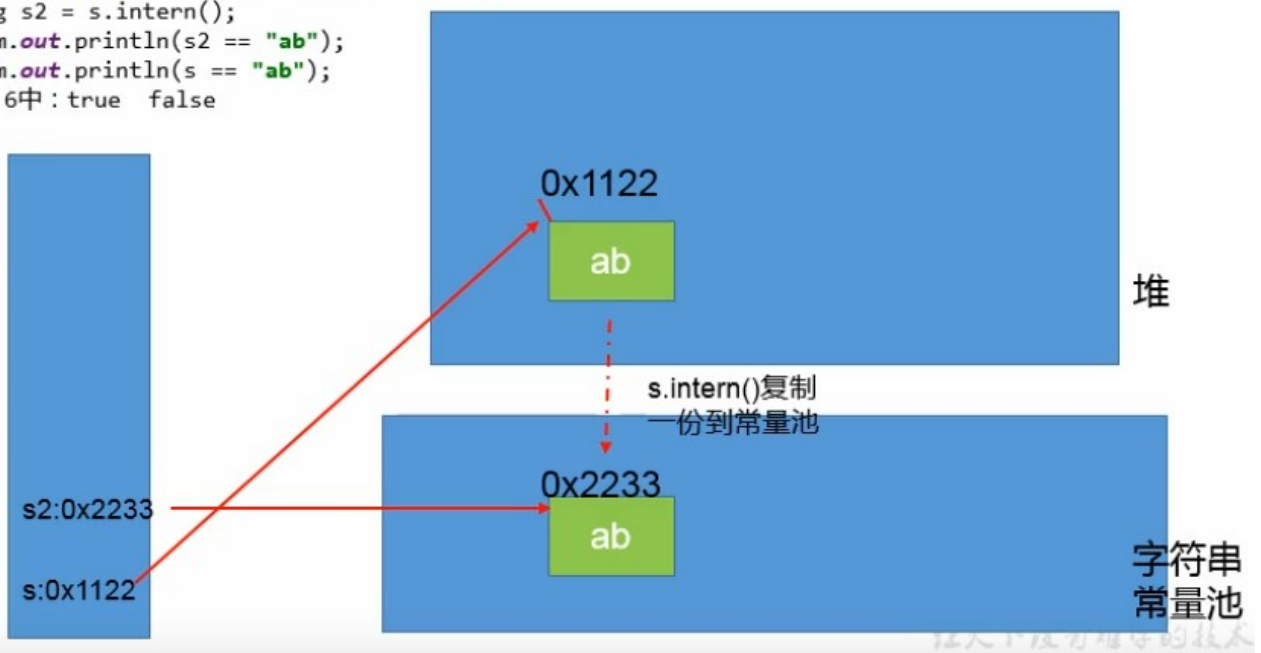
```

jdk6 中的 内存图

```

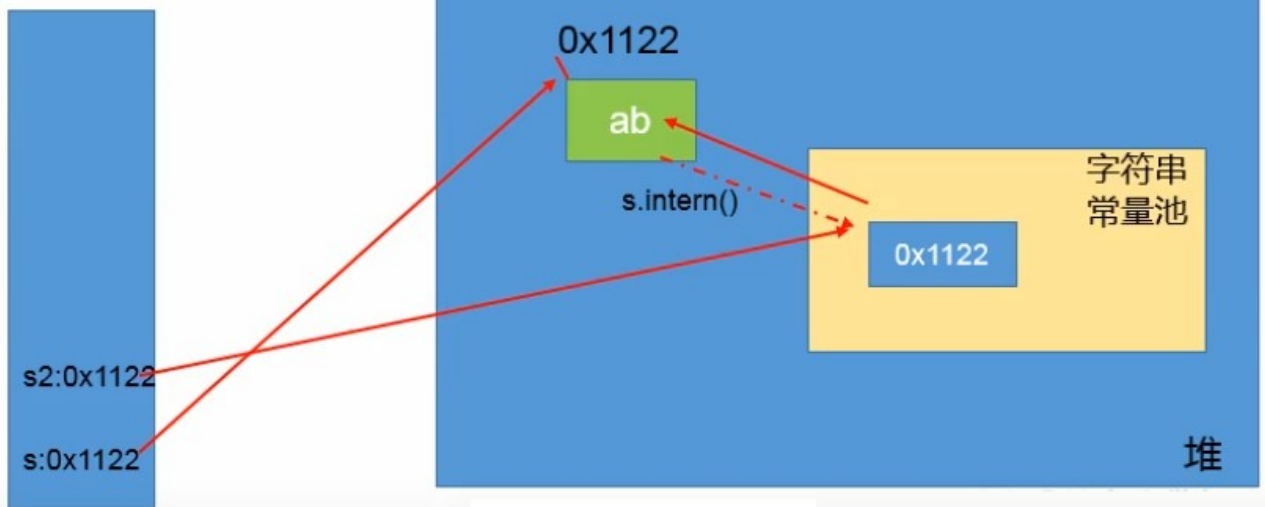
String s = new String("a") + new String("b");
String s2 = s.intern();
System.out.println(s2 == "ab");
System.out.println(s == "ab");
在jdk 6中: true false

```



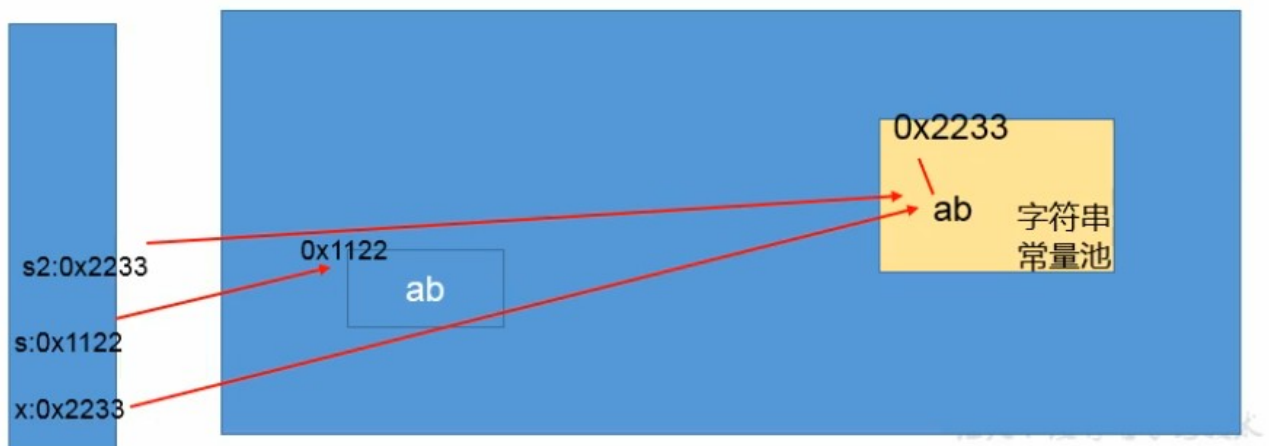
jdk 7/8 中

```
String s = new String("a") + new String("b");
String s2 = s.intern();
System.out.println(s2 == "ab");
System.out.println(s == "ab");
在jdk 7,8中:true true
```



扩展!!!!!!

```
String x = "ab";
String s = new String("a") + new String("b");
String s2 = s.intern(); // 因为常量池已经有ab, 则不会放入
System.out.println(s2 == x);
System.out.println(s == x);
在jdk6,7,8中执行都是true false
```



2.

```
1 /**
2  *
3  * @author shkstart shkstart@126.com
```

```

4  * @create 2020  20:26
5  */
6  public class StringExer2 {
7      public static void main(String[] args) {
8          String s1 = new String("ab");//执行完以后，会在字符串常量池中生成"ab"
9  //      String s1 = new String("a") + new String("b");//执行完以后，不会在字
10         s1.intern();
11         String s2 = "ab";
12         System.out.println(s1 == s2);
13     }
14 }
15

```

intern()方法 空间效率

```

1  import java.util.Random;
2
3  /**
4   * 使用intern()测试执行效率：空间使用上
5   *
6   * 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
7   *
8   *
9   * @author shkstart  shkstart@126.com
10  * @create 2020  21:17
11  */
12 public class StringIntern2 {
13     static final int MAX_COUNT = 1000 * 10000;
14     static final String[] arr = new String[MAX_COUNT];
15
16     public static void main(String[] args) {
17         Integer[] data = new Integer[]{1,2,3,4,5,6,7,8,9,10};
18
19         long start = System.currentTimeMillis();
20         for (int i = 0; i < MAX_COUNT; i++) {

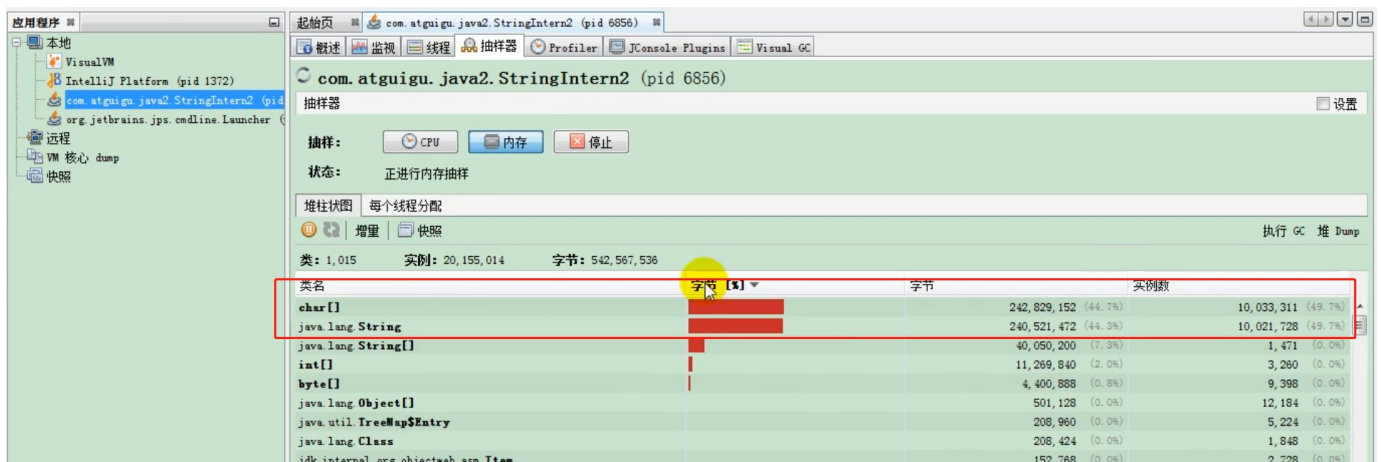
```

```

21 //      arr[i] = new String(String.valueOf(data[i % data.length]));
22      arr[i] = new String(String.valueOf(data[i % data.length])).intern()
23
24  }
25  long end = System.currentTimeMillis();
26  System.out.println("花费的时间为: " + (end - start));
27
28  try {
29      Thread.sleep(1000000);
30  } catch (InterruptedException e) {
31      e.printStackTrace();
32  }
33  System.gc();
34  }
35 }
36

```

不用 intern



用 intern 时间也快了



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。

指向的是对象, 那么对象不能被回收； 第二种指向的是常量池的常量, 所以对象会被回收, 所以省空间!!!!

讲解地址:

<https://www.bilibili.com/video/BV1PJ411n7xZ?p=131>

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。

StringTable的垃圾回收

-XX:+PrintStringTableStatistics

```
1  /**
2   * String的垃圾回收:
3   * -Xms15m -Xmx15m -XX:+PrintStringTableStatistics -XX:+PrintGCDetails
4   *
5   * @author shkstart shkstart@126.com
6   * @create 2020 21:27
7   */
8  public class StringGCTest {
9      public static void main(String[] args) {
10         for (int j = 0; j < 100000; j++) {
11             String.valueOf(j).intern();
12         }
13     }
14 }
```

当对 代码 j 进行 100, 1000, 10000 可以看到 当数特别大的时候, 进行了垃圾回收

G1中的String去重操作

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25%是String对象。

更进一步， 这里面差不多一半string对象是重复的， 重复的意思是说：

`string 1.equals(string 2) =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。

实现

- 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对象会被调整引用那个数组， 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hash table， 这样以后的时候就可以共享这个数组了。

命令行选项

- Use String De duplication (bool) : 开启String去重, 默认是不开启的, 需要手动开启。
- PrintString De duplication Statistics (bool) : 打印详细的去重统计信息
- String De duplication Age Threshold (uint x) : 达到这个年龄的String对象被认为是去重的候选对象