

笔记来源：尚硅谷 JVM 全套教程，百万播放，全网巅峰（宋红康详解 java 虚拟机）

同步更新：https://gitee.com/vectorx/NOTE_JVM

https://codechina.csdn.net/qq_35925558/NOTE_JVM

https://github.com/uxiahnan/NOTE_JVM

10. StringTable

10.1. String 的基本特性

10.1.1. String 在 jdk9 中存储结构变更

10.1.2. String 的基本特性

10.2. String 的内存分配

10.3. String 的基本操作

10.4. 字符串拼接操作

10.5. intern()的使用

10.5.1. intern 的使用：JDK6 vs JDK7/8

10.5.2. intern 的效率测试：空间角度

10.6. StringTable 的垃圾回收

10.7. G1 中的 String 去重操作

迷思Chat

10. StringTable

10.1. 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。
- String 实现了 Comparable 接口：表示 string 可以比较大小
- String 在 jdk8 及以前内部定义了 final char[] value 用于存储字符串数据。JDK9 时改为 byte[]

10.1.1. String 在 jdk9 中存储结构变更

官网地址：[JEP 254: Compact Strings \(java.net\)](https://openjdk.org/jeps/254)

Motivation

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.

String-related classes such as `AbstractStringBuilder`, `StringBuilder`, and `StringBuffer` will be updated to use the same representation, as will the HotSpot VM's intrinsic string operations.

This is purely an implementation change, with no changes to existing public interfaces. There are no plans to add any new public APIs or other interfaces.

The prototyping work done to date confirms the expected reduction in memory footprint, substantial reductions of GC activity, and minor performance regressions in some corner cases.

动机

目前 `String` 类的实现将字符存储在一个 `char` 数组中，每个字符使用两个字节（16 位）。从许多不同的应用中收集到的数据表明，字符串是堆使用的主要组成部分，此外，大多数字符串对象只包含 Latin-1 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字符数组中有一半的空间没有被使用。

说明

我们建议将 `String` 类的内部表示方法从 UTF-16 字符数组改为字节数组加编码标志域。新的 `String` 类将根据字符串的内容，以 ISO-8859-1/Latin-1（每个字符一个字节）或 UTF-16（每个字符两个字节）的方式存储字符编码。编码标志将表明使用的是哪种编码。

与字符串相关的类，如 `AbstractStringBuilder`、`StringBuilder` 和 `StringBuffer` 将被更新以使用相同的表示方法，HotSpot VM 的内在字符串操作也是如此。

这纯粹是一个实现上的变化，对现有的公共接口没有变化。目前没有计划增加任何新的公共 API 或其他接口。

迄今为止所做的原型设计工作证实了内存占用的预期减少，GC 活动的大幅减少，以及在某些角落情况下的轻微性能倒退。

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

```
1 public final class String implements java.io.Serializable, Comparable<String>,
  CharSequence {
2     @Stable
3     private final byte[] value;
4 }
```

10.1.2. String 的基本特性

`String`：代表不可变的字符序列。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。

字符串常量池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 `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，设置 `StringTable` 长度的话，1009 是可以设置的最小值

10.2. 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。

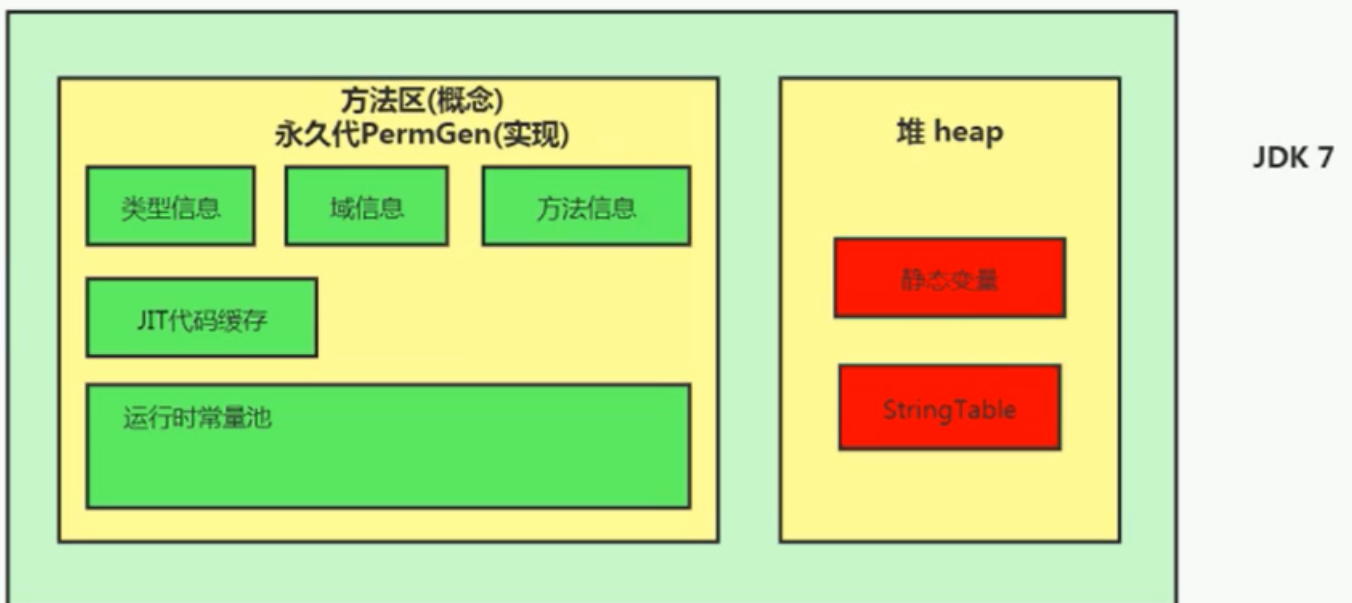
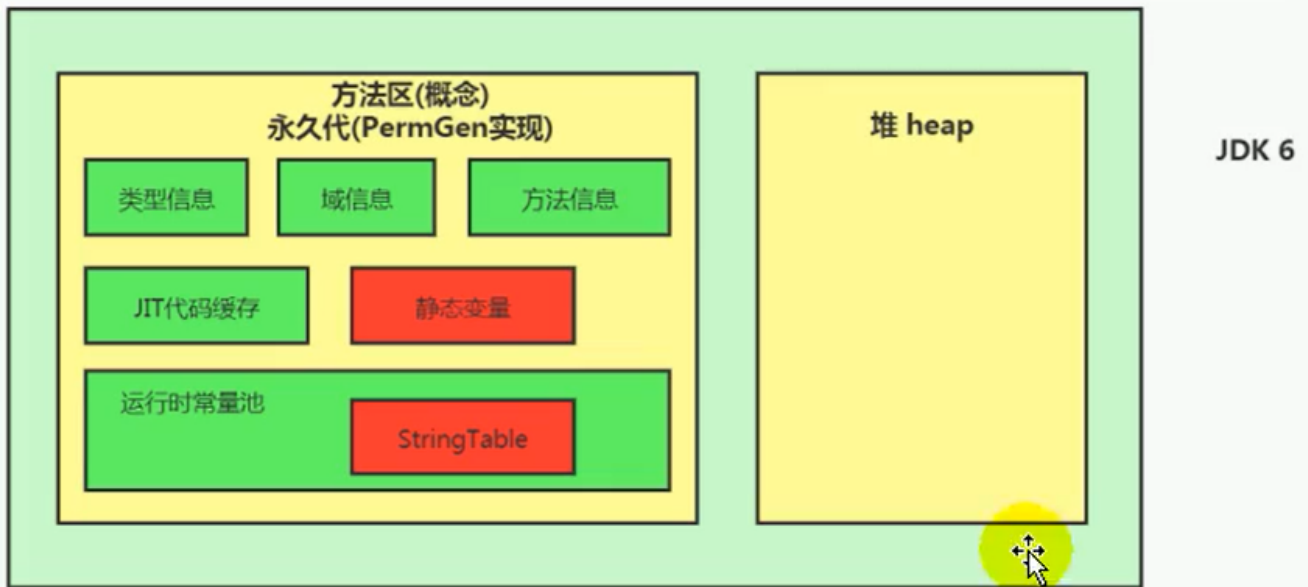
- 直接使用双引号声明出来的 `String` 对象会直接存储在常量池中。
- 如果不是用双引号声明的 `String` 对象，可以使用 `String` 提供的 `intern()` 方法，显式地让该字符串在字符串常量池放一份。这个后面重点谈

`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 `Java 7` 中使用 `String.intern()`。

`Java8` 元空间，字符串常量在堆中



StringTable 为什么要调整?

①permsize默认比较小 ②永久代垃圾回收频率低

官网地址: [Java SE 7 Features and Enhancements \(oracle.com\)](http://java.oracle.com/javase/7/features-and-enhancements/)

Synopsis: In JDK 7, interned strings are no longer allocated in the permanent generation of the Java heap, but are instead allocated in the main part of the Java heap (known as the young and old generations), along with the other objects created by the application. This change will result in more data residing in the main Java heap, and less data in the permanent generation, and thus may require heap sizes to be adjusted. Most applications will see only relatively small differences in heap usage due to this change, but larger applications that load many classes or make heavy use of the `String.intern()` method will see more significant differences.

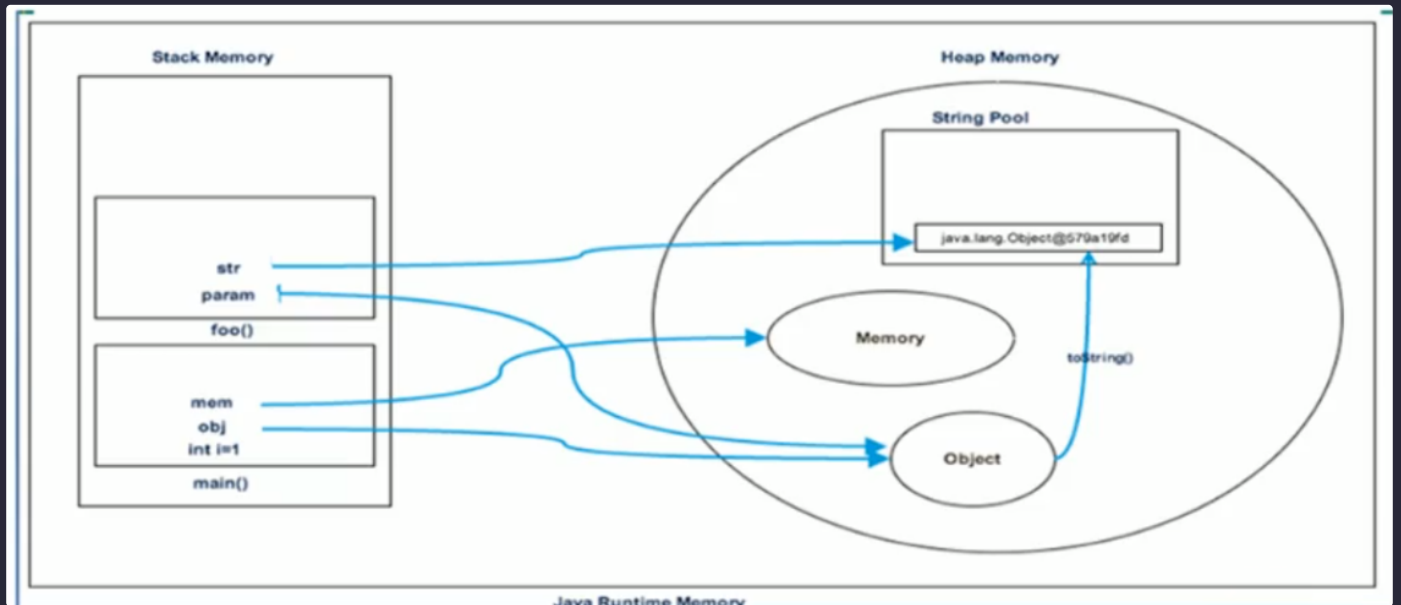
简介：在 JDK 7 中，内部字符串不再分配在 Java 堆的永久代中，而是分配在 Java 堆的主要部分（称为年轻代和老年代），与应用程序创建的其他对象一起。这种变化将导致更多的数据驻留在主 Java 堆中，而更少的数据在永久代中，因此可能需要调整堆的大小。大多数应用程序将看到由于这一变化而导致的堆使用的相对较小的差异，但 [加载许多类或大量使用 `String.intern\(\)` 方法的大型应用程序将看到更明显的差异](#)。

10.3. String 的基本操作

```
1  @Test
2  public void test1() {
3      System.out.println("1"); //2321
4      System.out.println("2");
5      System.out.println("3");
6      System.out.println("4");
7      System.out.println("5");
8      System.out.println("6");
9      System.out.println("7");
10     System.out.println("8");
11     System.out.println("9");
12     System.out.println("10"); //2330
13     System.out.println("1"); //2321
14     System.out.println("2"); //2322
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"); //2330
23 }
```

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。

```
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      private void foo(Object param) { //line 6
9          String str = param.toString(); //line 7
10         System.out.println(str);
11     } //Line 8
12 }
```



10.4. 字符串拼接操作

- 常量与常量的拼接结果在(字符串)常量池，原理是编译期优化
- 常量池中不会存在相同内容的变量
- 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`
- 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

举例 1

```

1  public static void test1() {
2      // 都是常量，前端编译期会进行代码优化
3      // 通过idea直接看对应的反编译的class文件，会显示 String s1 = "abc"; 说明做了代码优化
4      String s1 = "a" + "b" + "c";
5      String s2 = "abc";
6
7      // true，有上述可知，s1和s2实际上指向字符串常量池中的同一个值
8      System.out.println(s1 == s2);
9  }

```

举例 2

```

1  public static void test5() {
2      String s1 = "javaEE";
3      String s2 = "hadoop";
4
5      String s3 = "javaEEhadoop";
6      String s4 = "javaEE" + "hadoop";
7      String s5 = s1 + "hadoop";
8      String s6 = "javaEE" + s2;
9      String s7 = s1 + s2;
10 // 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。
11 System.out.println(s3 == s4); // true 编译期优化
12 System.out.println(s3 == s5); // false s1是变量，不能编译期优化

```

```

13     System.out.println(s3 == s6); // false s2是变量，不能编译期优化
14     System.out.println(s3 == s7); // false s1、s2都是变量
15     System.out.println(s5 == s6); // false s5、s6 不同的对象实例
16     System.out.println(s5 == s7); // false s5、s7 不同的对象实例
17     System.out.println(s6 == s7); // false s6、s7 不同的对象实例
18 //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；
19 //如果字符串常量池中不存在javaEEhadoop，则在常量池中加载一份javaEEhadoop，并返回此对象的地址。
20     String s8 = s6.intern();
21     System.out.println(s3 == s8); // true intern之后，s8和s3一样，指向字符串常量池中
    的"javaEEhadoop"
22 }

```

举例 3

```

1 public void test6(){
2     String s0 = "beijing";
3     String s1 = "bei";
4     String s2 = "jing";
5     String s3 = s1 + s2;
6     System.out.println(s0 == s3); // false    s3指向对象实例，s0指向字符串常量池中
    的"beijing"
7     String s7 = "shanxi";
8
9     final String s4 = "shan";
10    final String s5 = "xi";
11    String s6 = s4 + s5;
12    System.out.println(s6 == s7); // true    s4和s5是final修饰的，编译期就能确定s6的值了
13 }

```

- 不使用 final 修饰，即为变量。如 s3 行的 s1 和 s2，会通过 new StringBuilder 进行拼接
- 使用 final 修饰，即为常量。会在编译器进行代码优化。 (在实际开发中，能够使用 final 的，尽量使用)
- 字符串拼接操作不一定使用的是stringBuilder!
如果拼接符号左右两边都是字符串常量 或 常量引用 (final) ，则仍然使用编译期优化，即非stringBuilder的方式。
- final修饰的，在编译时就初始化好了，初始化得更早

举例 4

```

1 public void test3() {
2     String s1 = "a";
3     String s2 = "b";
4     String s3 = "ab";
5     String s4 = s1 + s2;
6     System.out.println(s3==s4);
7 }

```

```

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

```

I

字节码

我们拿例 4 的字节码进行查看, 可以发现 `s1 + s2` 实际上是 new 了一个 `StringBuilder` 对象, 并使用了 `append` 方法将 `s1` 和 `s2` 添加进来, 最后调用了 `toString` 方法赋给 `s4`

```

1  0 ldc #2 <a>
2  2 astore_1
3  3 ldc #3 <b>
4  5 astore_2
5  6 ldc #4 <ab>
6  8 astore_3
7  9 new #5 <java/lang/StringBuilder>
8 12 dup
9 13 invokespecial #6 <java/lang/StringBuilder.<init>>
10 16 aload_1
11 17 invokevirtual #7 <java/lang/StringBuilder.append>
12 20 aload_2
13 21 invokevirtual #7 <java/lang/StringBuilder.append>
14 24 invokevirtual #8 <java/lang/StringBuilder.toString>
15 27 astore 4
16 29 getstatic #9 <java/lang/System.out>
17 32 aload_3
18 33 aload 4
19 35 if_acmpne 42 (+7)
20 38 iconst_1
21 39 goto 43 (+4)
22 42 iconst_0
23 43 invokevirtual #10 <java/io/PrintStream.println>
24 46 return

```


字符串拼接操作性能对比

```
1 public class Test{
2     public static void main(String[] args) {
3         int times = 50000;
4         // String
5         long start = System.currentTimeMillis();
6         testString(times);
7         long end = System.currentTimeMillis();
8         System.out.println("String: " + (end-start) + "ms");
9         // StringBuilder
10        start = System.currentTimeMillis();
11        testStringBuilder(times);
12        end = System.currentTimeMillis();
13        System.out.println("StringBuilder: " + (end-start) + "ms");
14        // StringBuffer
15        start = System.currentTimeMillis();
16        testStringBuffer(times);
17        end = System.currentTimeMillis();
18        System.out.println("StringBuffer: " + (end-start) + "ms");
19    }
20    public static void testString(int times) {
21        String str = "";
22        for (int i = 0; i < times; i++) {
23            str += "test";
24        }
25    }
26    public static void testStringBuilder(int times) {
27        StringBuilder sb = new StringBuilder();
28        for (int i = 0; i < times; i++) {
29            sb.append("test");
30        }
31    }
32    public static void testStringBuffer(int times) {
33        StringBuffer sb = new StringBuffer();
34        for (int i = 0; i < times; i++) {
35            sb.append("test");
36        }
37    }
38 }
39 // 结果String: 7963ms
40 //StringBuilder: 1ms
41 //StringBuffer: 4ms
42
43 /*
44 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！
45 详情：① stringBuilder的append () 的方式：自始至终中只创建过一个stringBuilder的对象
46        使用string的字符串拼接方式：创建过多个stringBuilder和String的对象
```

```
47      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。
48      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(int highLevel); //new
    char[highLevel]
49      */
```

本实验进行 5 万次循环，String 拼接方式的时间是 StringBuilder.append 方式的约 8000 倍，StringBuffer.append()方式的时间是 StringBuilder.append()方式的约 4 倍

可以看到，通过 StringBuilder 的 append 方式的速度，要比直接对 String 使用“+”拼接的方式 **快的不是一点半点**

那么，在实际开发中，对于需要 **多次或大量拼接** 的操作，在不考虑线程安全问题时，我们就应该尽可能 **使用 StringBuilder 进行 append 操作**

除此之外，还有那些操作能够帮助我们提高字符串方面的运行效率呢？

StringBuilder 空参构造器的初始化大小为 16。那么，如果提前知道需要拼接 String 的个数，就应该直接使用 **带参构造器** 指定 capacity，以减少扩容的次数（扩容的逻辑可以自行查看源代码）

```
1  /** * Constructs a string builder with no characters in it and an * initial
    capacity of 16 characters. */
2  public StringBuilder() {
3      super(16);
4  }
5  /**
6   * Constructs a string builder with no characters in it and an
7   * initial capacity specified by the {@code capacity} argument.
8   * * @param
9   capacity the initial capacity.
10  * @throws NegativeArraySizeException
11  if the {@code capacity} *
12  argument is less than {@code 0}. */
13  public StringBuilder(
14      int capacity) {
15      super(capacity);
16  }
```

10.5. intern()的使用

官方 API 文档中的解释

```
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.

当调用 `intern` 方法时，如果池子里已经包含了一个与这个 `String` 对象相等的字符串，正如 `equals(Object)` 方法所确定的，那么池子里的字符串会被返回。否则，这个 `String` 对象被添加到池中，并返回这个 `String` 对象的引用。

由此可见，对于任何两个字符串 `s` 和 `t`，当且仅当 `s.equals(t)` 为真时，`s.intern() == t.intern()` 为真。

所有字面字符串和以字符串为值的常量表达式都是 `interned`。

返回一个与此字符串内容相同的字符串，但保证是来自一个唯一的字符串池。

`intern` 是一个 `native` 方法，调用的是底层 `C` 的方法

```
1 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，它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。

```
1 String myInfo = new String("I love atguigu").intern();
```

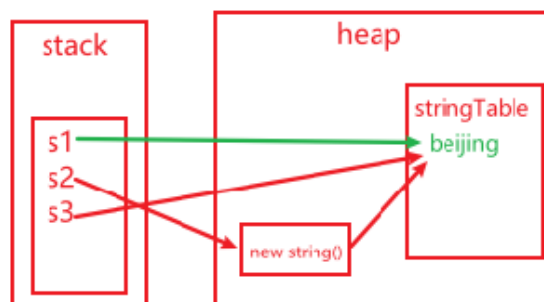
也就是说，如果在任意字符串上调用 `String.intern` 方法，那么其返回结果所指向的那个类实例，必须和直接以常量形式出现的字符串实例完全相同。因此，下列表达式的值必定是 `true`

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

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）

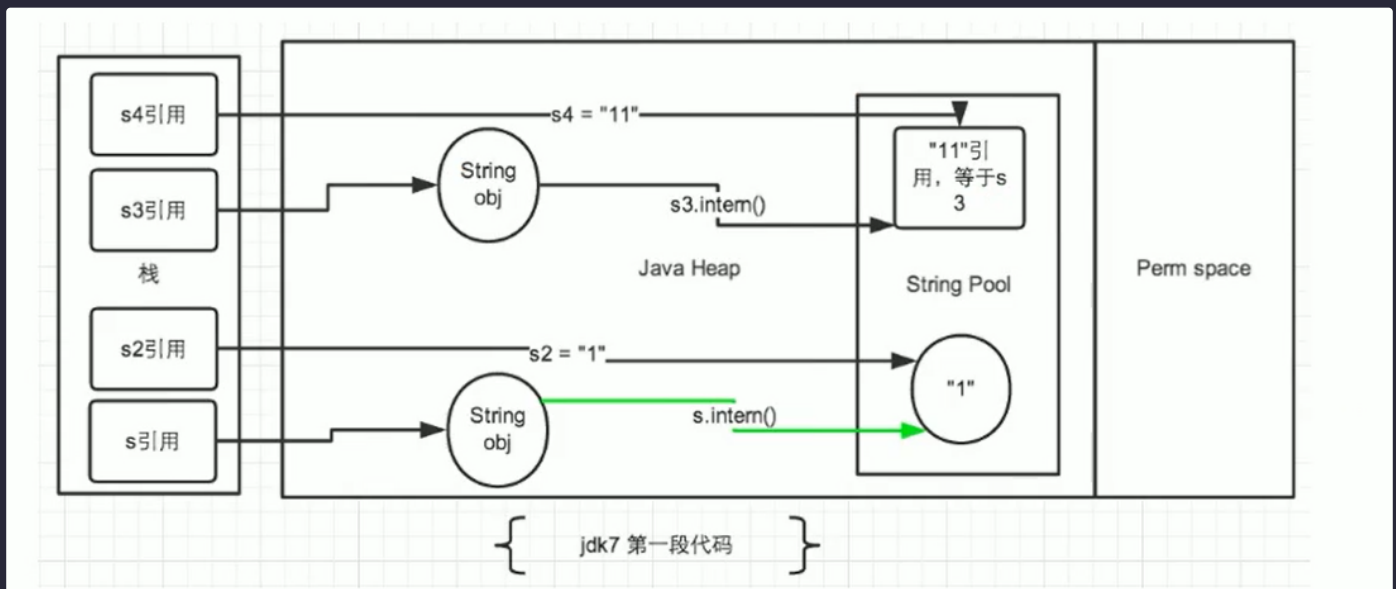
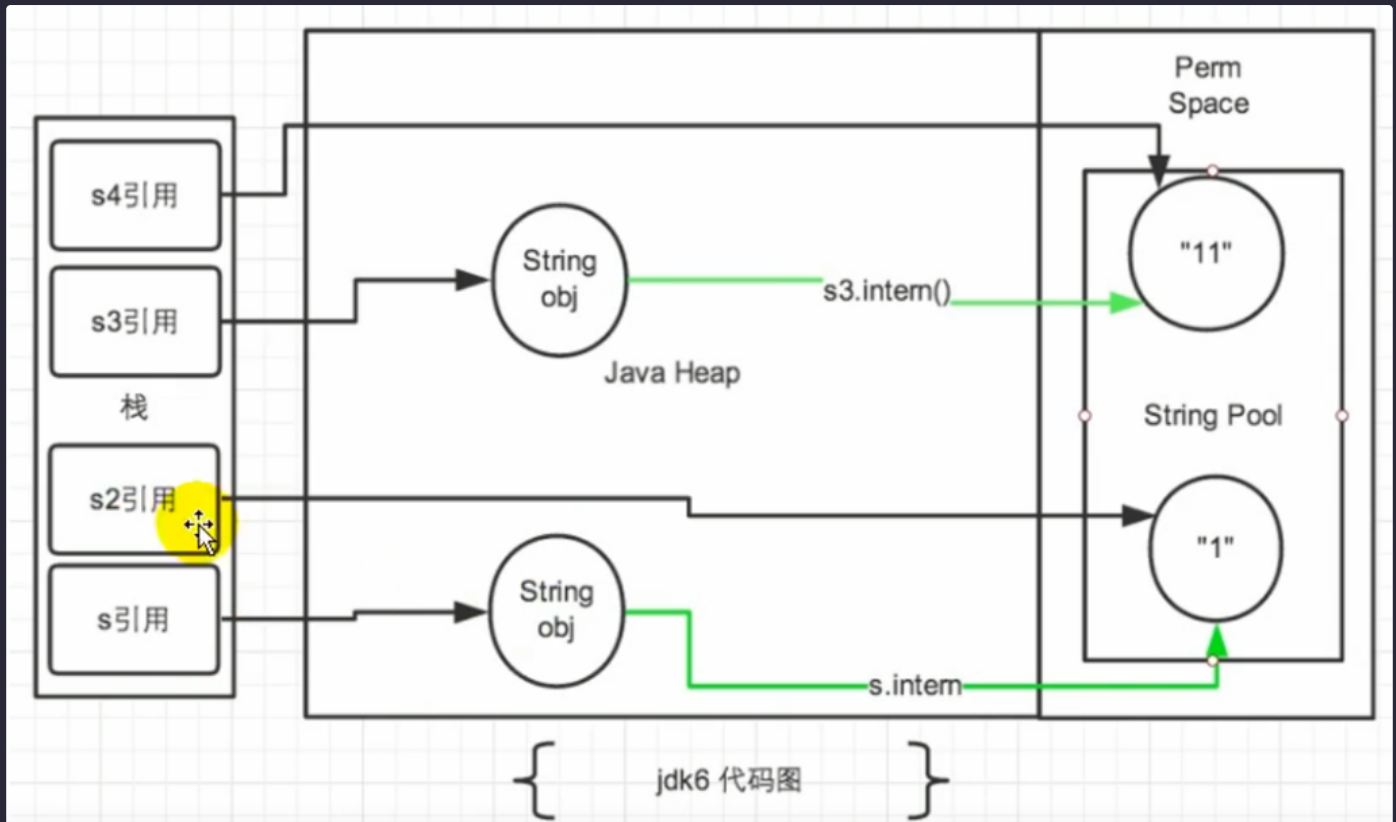
```
@test
public void test8(){
    String s1 = "beijing";
    String s2 = new String( original: "beijing");
    String s3 = s2.intern();

    System.out.println(s1 == s2); false
    System.out.println(s1 == s3); true
}
```



10.5.1. intern 的使用: JDK6 vs JDK7/8

```
1 /** * ① String s = new String("1") * 创建了两个对象 * 堆空间中一个new对象 * 字符串常量池中一个字符串常量"1" (注意: 此时字符串常量池中已有"1") * ② s.intern()由于字符串常量池中已存在"1" * * s 指向的是堆空间中的对象地址 * s2 指向的是堆空间中常量池中"1"的地址 * 所以不相等 */String s = new String("1");s.intern();String s2 = "1";System.out.println(s==s2); // jdk1.6 false jdk7/8 false/* * ① String s3 = new String("1") + new String("1") * 等价于new String ("11") , 但是, 常量池中并不生成字符串"11"; * * ② s3.intern() * 由于此时常量池中并无"11", 所以把s3中记录的对象的地址存入常量池 * 所以s3 和 s4 指向的都是一个地址*/String s3 = new String("1") + new String("1");s3.intern();String s4 = "11";System.out.println(s3==s4); //jdk1.6 false jdk7/8 true
```



总结 String 的 intern()的使用:

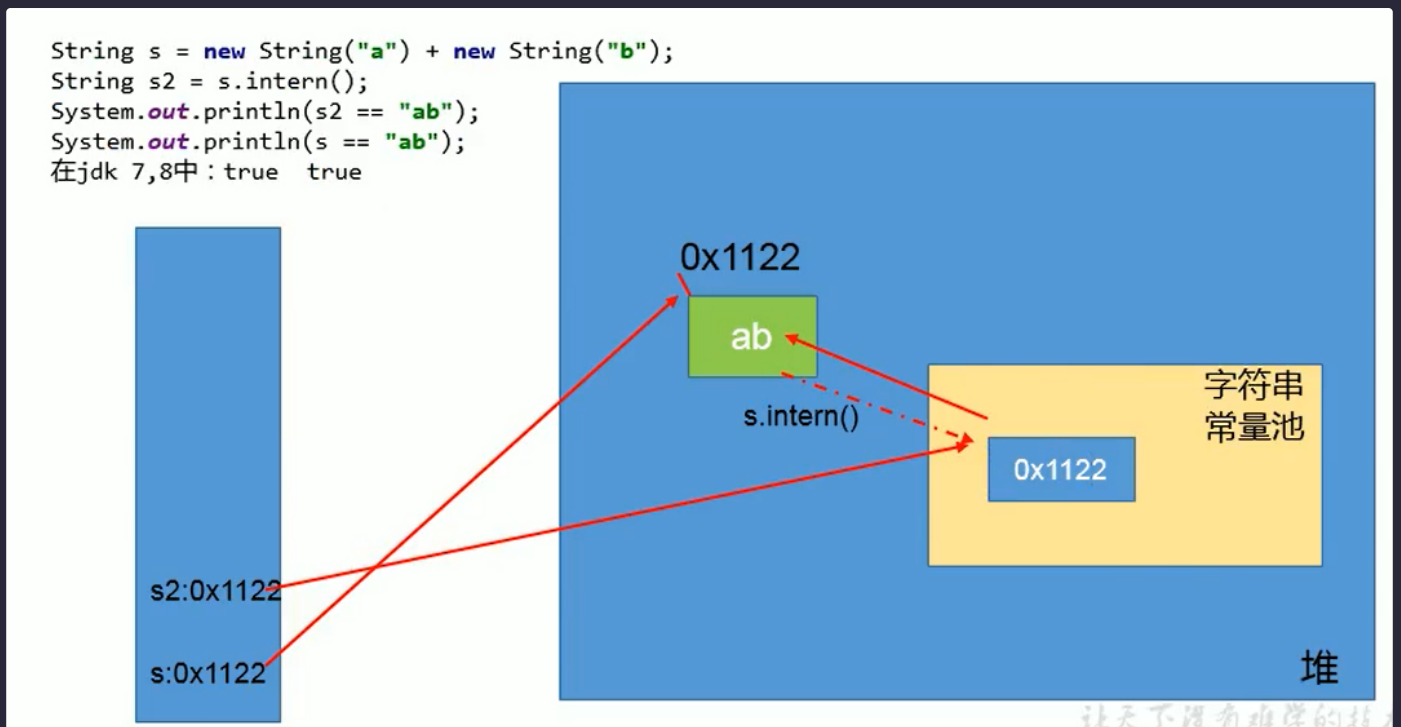
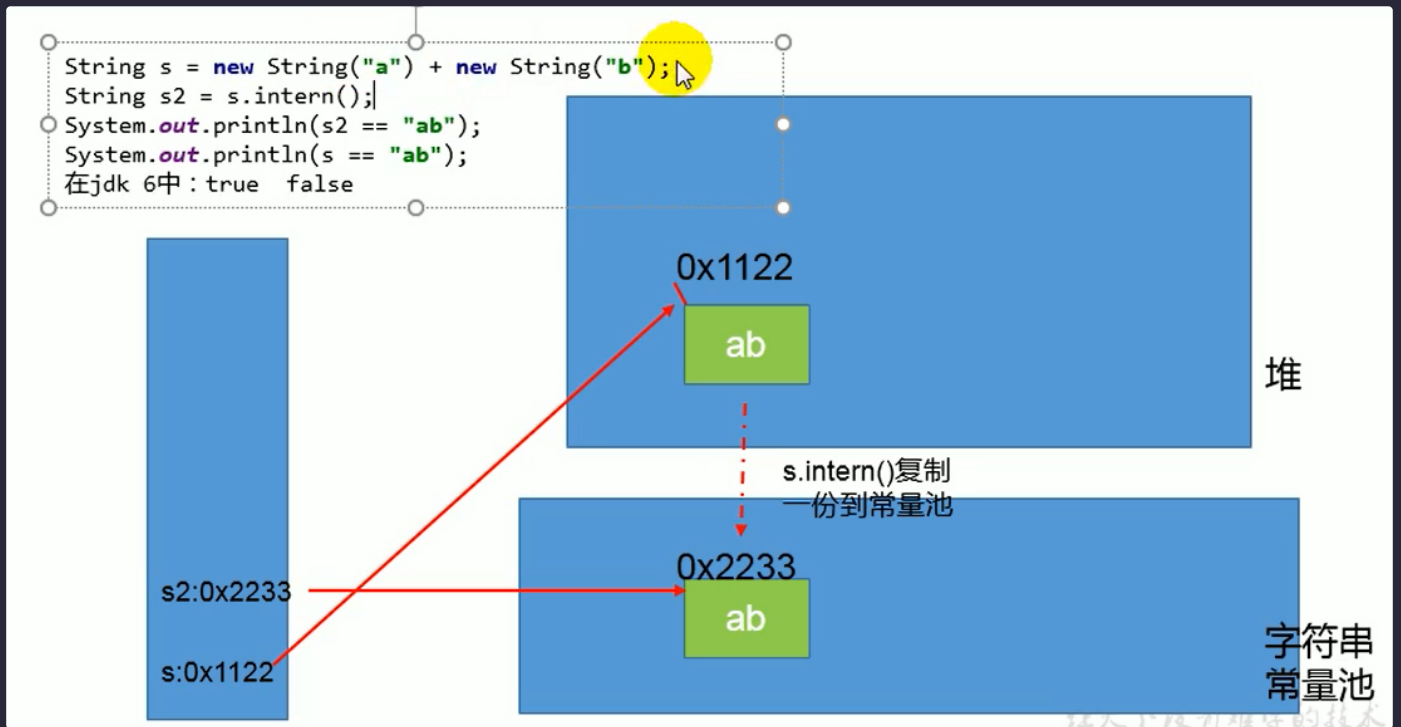
JDK1.6 中，将这个字符串对象尝试放入串池。

- 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

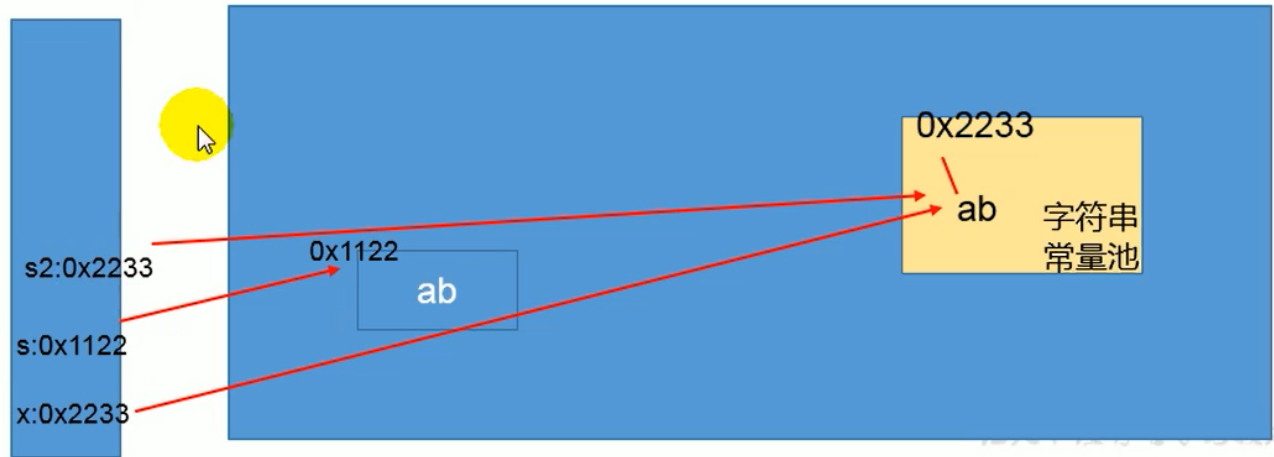
练习 1



练习 2

```
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



10.5.2. intern 的效率测试：空间角度

我们通过测试一下，使用了 intern 和不使用的时候，其实相差还挺多的

```
1 public class StringIntern2 {
2     static final int MAX_COUNT = 1000 * 10000;
3     static final String[] arr = new String[MAX_COUNT];
4     public static void main(String[] args) {
5         Integer [] data = new Integer[]{1,2,3,4,5,6,7,8,9,10};
6         long start = System.currentTimeMillis();
7         for (int i = 0; i < MAX_COUNT; i++) {
8             // arr[i] = new String(String.valueOf(data[i%data.length]));
9             arr[i] = new String(String.valueOf(data[i%data.length])).intern();
10        }
11        long end = System.currentTimeMillis();
12        System.out.println("花费的时间为: " + (end - start));
13        try {
14            Thread.sleep(1000000);
15        } catch (Exception e) {
16            e.printStackTrace();
17        }
18    }
19 }
20 // 运行结果不使用intern: 7256ms使用intern: 1395ms
```

结论：对于程序中大量使用存在的字符串时，尤其存在很多已经重复的字符串时，使用 intern() 方法能够节省内存空间。

降低内存占用的原因是 之前的堆区创建的字符串由于无人指向，然后被GC清理掉了。

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，就会很明显降低内存的大小。

10.6. StringTable 的垃圾回收

```
1 public class StringGCTest {
2     /**
3      * -Xms15m -Xmx15m -XX:+PrintGCDetails
4      */
5     public static void main(String[] args) {
6         for (int i = 0; i < 100000; i++) {
7             String.valueOf(i).intern();
8         }
9     }
10 }
```

运行结果

```
1 [GC (Allocation Failure) [PSYoungGen: 4096K→504K(4608K)] 4096K→1689K(15872K),
0.0581583 secs] [Times: user=0.00 sys=0.00, real=0.06 secs] [GC (Allocation Failure)
[PSYoungGen: 4600K→504K(4608K)] 5785K→2310K(15872K), 0.0015621 secs] [Times:
user=0.00 sys=0.00, real=0.00 secs] [GC (Allocation Failure) [PSYoungGen: 4600K-
>504K(4608K)] 6406K→2350K(15872K), 0.0034849 secs] [Times: user=0.00 sys=0.00,
real=0.00 secs] Heap PSYoungGen      total 4608K, used 1919K [0x00000000ffb00000,
0x0000000010000000, 0x0000000010000000) eden space 4096K, 34% used
[0x00000000ffb00000,0x00000000ffc61d30,0x00000000fff00000) from space 512K, 98%
used [0x00000000fff00000,0x00000000fff7e010,0x00000000fff80000) to space 512K, 0%
used [0x00000000fff80000,0x00000000fff80000,0x0000000010000000) ParOldGen
total 11264K, used 1846K [0x00000000ff000000, 0x00000000ffb00000,
0x00000000ffb00000) object space 11264K, 16% used
[0x00000000ff000000,0x00000000ff1cd9b0,0x00000000ffb00000) Metaspace      used
3378K, capacity 4496K, committed 4864K, reserved 1056768K class space      used 361K,
capacity 388K, committed 512K, reserved 1048576K
```

10.7. G1 中的 String 去重操作

官网地址: [JEP 192: String Deduplication in G1 \(java.net\)](https://openjdk.org/jeps/192)

Motivation

Many large-scale Java applications are currently bottlenecked on memory. Measurements have shown that roughly 25% of the Java heap live data set in these types of applications is consumed by `String` objects. Further, roughly half of those `String` objects are duplicates, where duplicates means `string1.equals(string2)` is true. Having duplicate `String` objects on the heap is, essentially, just a waste of memory. This project will implement automatic and continuous `String` deduplication in the G1 garbage collector to avoid wasting memory and reduce the memory footprint.

目前，许多大规模的 Java 应用程序在内存上遇到了瓶颈。测量表明，在这些类型的应用程序中，大约 25%的 Java 堆实时数据集被 String'对象所消耗。此外，这些 "String "对象中大约有一半是重复的，其中重复意味着 "string1.equals(string2) "是真的。在堆上有重复的 String'对象，从本质上讲，只是一种内存的浪费。这个项目将在 G1 垃圾收集器中实现自动和持续的`String'重复数据删除，以避免浪费内存，减少内存占用。

注意这里说的重复，指的是在堆中的数据，而不是常量池中的，因为常量池中的本身就不会重复

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 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：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。

实现

1. 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
2. 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。
3. 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。
4. 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。
5. 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

```
1 # 开启String去重，默认是不开启的，需要手动开启。
2 UseStringDeduplication(bool)
3 # 打印详细的去重统计信息
4 PrintStringDeduplicationStatistics(bool)
5 # 达到这个年龄的String对象被认为是去重的候选对象
6 StringpeDuplicationAgeThreshold(uintx)
```

迷思Chat

java中，如果显式地new一个String对象，它会被放在堆区中。比如String s = new String("Hello");，堆区中会存在一个String对象，此时字符串常量池中是否会存在一个"Hello"字符串？

ChatGPT 说：

ChatGPT

是的，在你显式地使用 `new String("Hello")` 时，堆区会创建一个新的 `String` 对象，而 **字符串常量池** 中是否会有 `"Hello"` 则取决于它是否已经存在。

具体来说：

1. **常量池**：Java 在编译期间会将字符串字面量（比如 `"Hello"`）存储在常量池中，如果 `"Hello"` 已经在常量池中，那么不会重复创建。
2. **`new String()`**：当你使用 `new String("Hello")` 时，会显式地在堆上创建一个新的 `String` 对象，即使常量池中已经有 `"Hello"`，它也会新建一个堆对象。但在 `new String("Hello")` 的构造函数中，引用的仍然是常量池中的 `"Hello"`。

具体过程：

```
1 java
2
3
4 复制代码
5 String s = new String("Hello");
```

- `"Hello"` 是一个字面量，它会首先查找 **字符串常量池**，如果池中没有，它将被加入到常量池中。
- `new String("Hello")` 会在 **堆区** 创建一个新的 `String` 对象，它的值与常量池中的 `"Hello"` 相同，但它是一个不同的对象。

如果你想让堆上的 `String` 对象也引用常量池中的对象，你可以使用 `intern()` 方法：

```
1 java
2
3
4 复制代码
5 String s1 = new String("Hello").intern();
```

这将确保 `s1` 指向常量池中的 `"Hello"`。

因此，在你的示例中：

- **堆区**：会有一个新的 `String` 对象。
- **字符串常量池**：如果之前没有 `"Hello"`，它也会被加入到常量池中。

您说：

是否我可以理解为：只要在源代码中出现过的字符串字面量，包括 `new String("Hello")` 中的 `"Hello"`，都会在字符串常量池中创建一份？

ChatGPT 说：

你可以理解为：只要在源代码中出现过的字符串字面量，都会在编译时尝试放入字符串常量池中。具体来说，`new String("Hello")` 中的 `"Hello"` 是一个字面量，而字面量会按照以下规则处理：

1. 编译时：Java 编译器在编译时会把所有的字符串字面量（例如 `"Hello"`）放入字符串常量池中。如果字符串常量池中已经存在相同的字面量，它不会重复添加。
2. 运行时：
 - 当代码执行 `new String("Hello")` 时，首先会查找字符串常量池中是否已经存在 `"Hello"`。如果存在，它不会再向常量池中添加该字面量。
 - 其次，`new String("Hello")` 会在堆上创建一个新的 `String` 对象，它的内容是从常量池中的 `"Hello"` 复制过来的。

关键点

- 字符串字面量（如 `"Hello"`）在程序中出现时，无论是否包裹在 `new String()` 中，都会被放入或查找字符串常量池。
- `new String()` 语句不会影响常量池中的字面量，它只是在堆上创建了一个新对象，与常量池中的对象无关。

因此，你的理解基本正确：在源代码中出现的字符串字面量都会被存入字符串常量池。即使是在 `new String("Hello")` 中的 `"Hello"`，它依然是一个字面量，会存在于常量池中。

补充

- 如果你使用

```
1 intern()
```

方法，堆上的

```
1 String
```

对象也会指向常量池中的字符串：

```
1 java
2
3
4 复制代码
5 String s = new String("Hello").intern();
```

这样，

```
1 s
```

将引用常量池中的

```
1 "Hello"
```

对象，而不是堆上的新对象。

您说：

如果字符串常量池中已经存在"Hello",但是我又执行了`String s = new String("Hello");`，那么这个s指向的应该是堆区的String对象，而这个堆区的String对象又会引用字符串常量池的"Hello"，对吗？这样可以保证字符串只会被存储一份

ChatGPT 说：

ChatGPT

你理解得大体正确，但有一点需要澄清。确实，当你执行 `String s = new String("Hello");` 时，`s` 会指向堆区的 `String` 对象，而堆区的 `String` 对象的内容和常量池中的 "Hello" 是一样的，但堆区的 `String` 对象并不会直接引用常量池中的 "Hello"。

具体细节如下：

1. 常量池中的 "Hello"：

- "Hello" 作为字面量，首先会被放入字符串常量池中。如果常量池中已经有 "Hello"，不会重复存储。

2. 堆上的新 String 对象：

- 当执行 `new String("Hello")` 时，JVM 会在堆区创建一个新的 `String` 对象。
- 这个新的 `String` 对象会复制常量池中的 "Hello" 字符串的值，而不是直接引用常量池中的 "Hello" 对象。因此，堆上的 `String` 对象和常量池中的 "Hello" 是两个不同的对象，尽管它们的内容相同。

3. 最终情况：

- 堆上的新 `String` 对象是一个独立的对象，内容与常量池中的 "Hello" 一致，但它们不是同一个引用。
- 因此，字符串 "Hello" 的内容确实只会存储一份，但堆上的 `String` 对象是对这个内容的一个拷贝，而不是对常量池对象的直接引用。

总结：

- 堆区对象：`new String("Hello")` 会创建一个新的 `String` 对象，存储在堆中。
- 常量池中的字符串："Hello" 在常量池中只会有一份。
- 引用关系：堆区的 `String` 对象并不会直接引用常量池中的 "Hello"，而是独立存在。虽然内容相同，但它们是两个不同的对象。

如果你想让堆上的 `String` 对象引用常量池中的字符串，可以通过调用 `intern()` 方法来实现：

```
1 java
2
3
4 复制代码
5 String s = new String("Hello").intern();
```

