Java 常量池技术

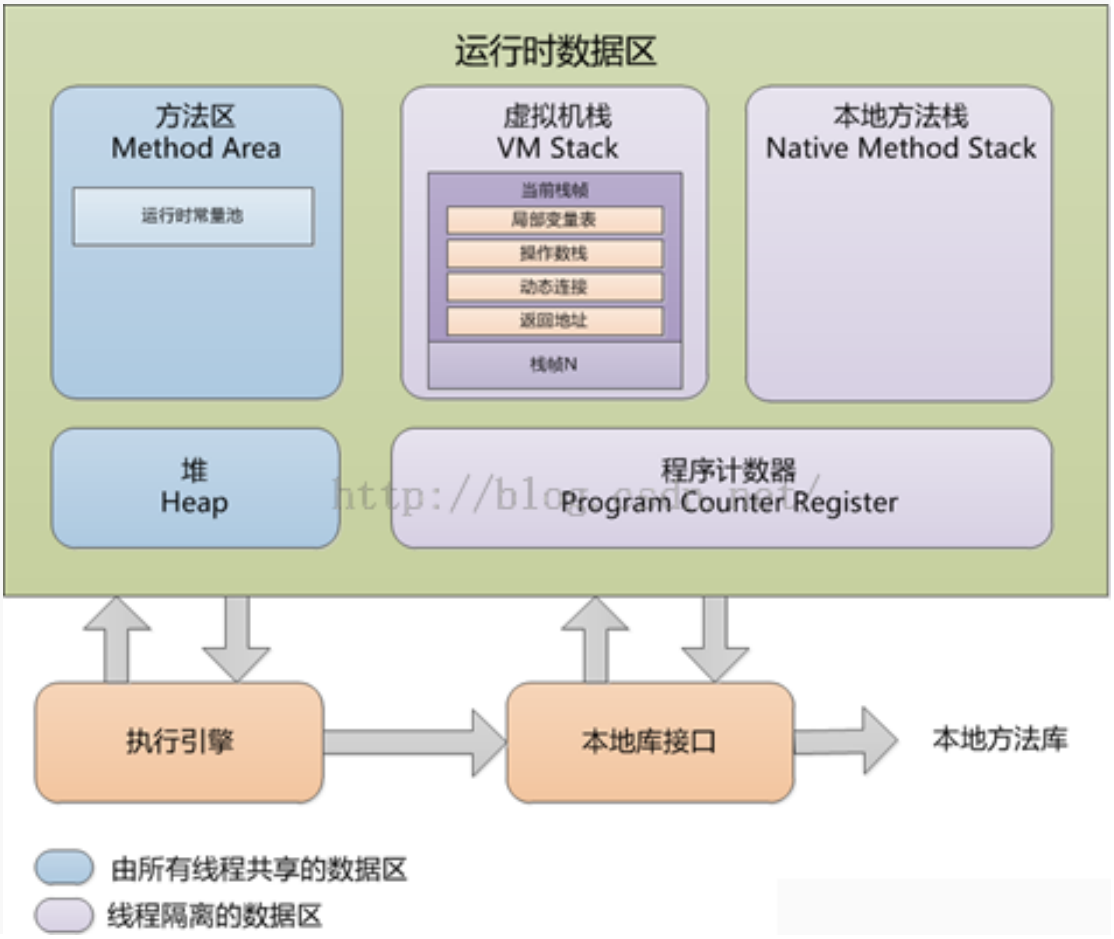
常量池是 JVM 的一块特殊的内存空间，用于保存在编译期已确定的，已编译的 class 文件中的一份数据。它包括了关于类，方法，接口等中的常量；

Java 中八种基本类型的包装类的大部分都实现了常量池技术，它们是 Byte、Short、Integer、Long、Character、Boolean（两种浮点数类型的包装类 Float、Double 则没有实现）(String 类也实现了常量池的技术)。另外 Byte, Short, Integer, Long, Character 这 5 种整型的包装类仅在对应值在-128 到 127 时才可使用对象池。

java 中的常量池技术，是为了方便快捷地创建某些对象而出现的，当需要一个对象时，就可以从池中取一个出来（如果池中没有则创建一个），这样在需要重复创建相等变量时节省了很多时间。常量池其实也就是一个内存空间，不同于使用 new 关键字创建的对象所在的堆空间。

先了解一下 JVM 运行时数据区的内存模型。《深入 JAVA 虚拟机》书中是这样描述的：JVM 运行时数据区的内存模型由五部分组成：

【1】方法区 【2】堆 【3】JAVA 栈 【4】PC 寄存器 【5】本地方法栈



- 对于 String s = "test" ,它的虚拟机指令:

```
0:  ldc    #16; //String test
2:  astore_1
3:  return
```

- 对于上面虚拟机指令，其各自的指令流程在《深入 JAVA 虚拟机》这样描述到(结合上面实例):
  - ldc 指令格式: ldc,index
  - ldc 指令过程: 要执行 ldc 指令，JVM 首先查找 index 所指定的常量池入口，在 index 指向的常量池入口，JVM 将会查找 CONSTANT_Integer_info，CONSTANT_Float_info 和 CONSTANT_String_info 入口。如果还没有这些入口，JVM 会解析它们。而对于上面的 hahaJVM 会找到 CONSTANT_String_info 入口，同时，将把指向被拘留 String 对象（由解析该入口的进程产生）的引用压入操作数栈。
  - astore_1 指令格式: astore_1
  - astore_1 指令过程: 要执行 astore_1 指令，JVM 从操作数栈顶部弹出一个引用类型或者 returnAddress 类型值，然后将该值存入由索引 1 指定的局部变量中,即将引用类型或者 returnAddress 类型值存入局部变量 1。
  - return 指令的过程: 从方法中返回，返回值为 void。

从上面的 ldc 指令的执行过程可以得出: s 的值是来自被拘留 String 对象（由解析该入口的进程产生）的引用，即可以理解为是从被拘留 String 对象的引用复制而来的，故我个人的理解是 s 的值是存在栈当中。上面是对于 s 值得分析，接着是对于"haha"值的分析,我们知道，对于 String s = "test" 其中 "test"值在 JAVA 程序编译期就确定下来了的。简单一点说，就是 test 的值在程序编译成 class 文件后，就在 class 文件中生成了（可以用文本编辑工具在打开 class 文件后的字节码文件中看到这个 test 值）。执行 JAVA 程序的过程中，第一步是 class 文件生成，然后被 JVM 装载到内存执行。那么 JVM 装载这个 class 到内存中，其中的 test 这个值，在内存中是怎么为其开辟空间并存储在哪个区域中呢？

说到这里，我们不妨先来了解一下 JVM 常量池这个结构，《深入 JAVA 虚拟机》书中有这样的描述:

- **常量池**:虚拟机必须为每个被装载的类型维护一个常量池。常量池就是该类型所用到常量的一个有序集和，包括直接常量（string,integer 和 floating point 常量）和对其他类型，字段和方法的符号引用。对于 String 常量，它的值是在常量池中的。而 JVM 中的常量池在内存当中是以表的形式存在的，对于 String 类型，有一张固定长度的 CONSTANT_String_info 表用来存储文字字符串值，注意：该表只存储文字字符串值，不存储符号引用。说到这里，对常量池中的字符串值的存储位置应该有一个比较明了的理解了。

- **具体结构**:在 Java 程序中，有很多的东西是永恒的，不会在运行过程中变化。比如一个类的名字，一个类字段的名字/所属类型，一个类方法的名字/返回类型/参数名与所属类型，一个常量，还有在程序中出现的大量的字面值。 每一个都是常量池中的一个常量表(常量项)。而这些常量表之间又有不同，class 文件共有 11 种常量表，如下所示：

常量表类型	标志值(占1 byte)	描述
CONSTANT_Utf8	1	UTF-8编码的Unicode字符串
CONSTANT_Integer	3	int类型的字面值
CONSTANT_Float	4	float类型的字面值
CONSTANT_Long	5	long类型的字面值
CONSTANT_Double	6	double类型的字面值
CONSTANT_Class	7	对一个类或接口的符号引用
CONSTANT_String	8	String类型字面值的引用
CONSTANT_Fieldref	9	对一个字段的符号引用
CONSTANT_Methodref	10	对一个类中方法的符号引用
CONSTANT_InterfaceMethodref	11	对一个接口中方法的符号引用
CONSTANT_NameAndType	12	对一个字段或方法的部分符号引用

- (1)CONSTANT_Utf8 用 UTF-8 编码方式来表示程序中所有的重要常量字符串。这些字符串包括：
  - ①类或接口的全限定名， ②超类的全限定名，③父接口的全限定名，④类字段名和所属类型名，⑤类方法名和返回类型名、以及参数名和所属类型名。⑥字符串字面值
  - 表格式： tag(标志 1：占 1byte) length(字符串所占字节的长度，占 2byte) bytes(字符串字节序列)

- (2)CONSTANT_Integer、 CONSTANT_Float、 CONSTANT_Long、 CONSTANT_Double 所有基本数据类型的字面值。比如在程序中出现的 1 用 CONSTANT_Integer 表示。3.1415926F 用 CONSTANT_Float 表示。
  - 表格式： tag bytes(基本数据类型所需使用的字节序列)
- (3)CONSTANT_Class 使用符号引用来表示类或接口。我们知道所有类名都以 CONSTANT_Utf8 表的形式存储。但是我们并不知道 CONSTANT_Utf8 表中哪些字符串是类名，那些是方法名。因此我们必须用一个指向类名字符串的符号引用常量来表明。
  - 表格式： tag name_index(给出表示类或接口名的 CONSTANT_Utf8 表的索引)
- (4)CONSTANT_String 同 CONSTANT_Class，指向包含字符串字面值的 CONSTANT_Utf8 表。
  - 表格式： tag string_index(给出表示字符串字面值的 CONSTANT_Utf8 表的索引)
- (5)CONSTANT_Fieldref 、 CONSTANT_Methodref、 CONSTANT_InterfaceMethodref 指向包含该字段或方法所属类名的 CONSTANT_Utf8 表，以及指向包含该字段或方法的名字和描述符的 CONSTANT_NameAndType 表。
  - 表格式： tag class _index(给出包含所属类名的 CONSTANT_Utf8 表的索引) name_and_type_index(包含字段名或方法名以及描述符的 CONSTANT_NameAndType 表 的索引)
- (6)CONSTANT_NameAndType 指向包含字段名或方法名以及描述符的 CONSTANT_Utf8 表。
  - 表格式： tag name_index(给出表示字段名或方法名的 CONSTANT_Utf8 表的索引) type_index(给出表示描述符的 CONSTANT_Utf8 表的索引)

在 Java 源代码中的每一个字面值字符串，都会在编译成 class 文件阶段，形成标志号为 8(CONSTANT_String_info)的常量表 。当 JVM 加载 class 文件的时候，会为对应的常量池建立一个内存数据结构，并存放在方法区中。同时 JVM 会自动为 CONSTANT_String_info 常量表中的字符串常量的字面值在堆中创建新的 String 对象(intern 字符串对象 ，又叫拘留字符串对象)。然后把 CONSTANT_String_info 常量表的入口地址转变成这个堆中 String 对象的直接地址(常量池解析)。

- **拘留字符串对象：**源代码中所有相同字面值的字符串常量只可能建立唯一一个拘留字符串对象。 实际上 JVM 是通过一个记录了拘留字符串引用的内部数据结构来维持这一特性的。在 Java 程序中，可以调用 String 的 intern()方法来使得一个常规字符串对象成为拘留字符串对象。

- **八种基本类型的包装类和对象池：**Java 中基本类型的包装类的大部分都实现了常量池技术，这些类是 Byte,Short,Integer,Long,Character,Boolean,另外两种浮点数类型的包装类则没有实现。另外 Byte,Short,Integer,Long,Character 这 5 种整型的包装类也只是在对应值小于等于 127 时才可使用对象池，也即对象不负责创建和管理大于 127 的这些类的对象。一些对应的测试代码：

```
public class Test{
   public static void main(String[] args){
      // 5 种整形的包装类 Byte,Short,Integer,Long,Character 的对象，
      // 在值小于 127 时可以使用常量池
      Integer i1=127;
      Integer i2=127;
      System.out.println(i1==i2); //输出 true
      // 值大于 127 时，不会从常量池中取对象
      Integer i3=128;
      Integer i4=128;
      System.out.println(i3==i4); //输出 false
      // Boolean 类也实现了常量池技术
      Boolean bool1=true;
      Boolean bool2=true;
      System.out.println(bool1==bool2); //输出 true
      // 浮点类型的包装类没有实现常量池技术
      Double d1=1.0;
      Double d2=1.0;
      System.out.println(d1==d2); //输出 false
   }
}
```

- **对 Integer 对象的代码补充**

```
private static final Integer[] SMALL_VALUES = new Integer[256];

            static {
                  for (int i = -128; i < 128; i++) {
                        SMALL_VALUES[i + 128] = new Integer(i);
```

```
                }
            }

            public static Integer valueOf(int i) {
               assert IntegerCache.high >= 127;
               if (i >= IntegerCache.low && i <= IntegerCache.high)
                  return IntegerCache.cache[i + (-IntegerCache.low)];
               return new Integer(i);
            }
            //Integer 缓存内部类，应该是 jdk1.7 版本
            private static class IntegerCache {
                  static final int low = -128;
                  static final int high;
                  static final Integer cache[];
                  static {
                  // high value may be configured by property
                  int h = 127;
                  String integerCacheHighPropValue =
 sun.misc.VM.getSavedProperty("java.lang.Integer.IntegerCache.high");
                  if (integerCacheHighPropValue != null) {
                      int i = parseInt(integerCacheHighPropValue);
                      i = Math.max(i, 127);
                  // Maximum array size is Integer.MAX_VALUE
                     h = Math.min(i, Integer.MAX_VALUE - (-low) - 1);
                  }
                        high = h;
                        cache = new Integer[(high - low) + 1];
                        int j = low;
                        for (int k = 0; k < cache.length; k++)
                              cache[k] = new Integer(j++);
                  }
                  private IntegerCache() {}
            }
      }
```

当你直接给一个 Integer 对象一个 int 值的时候，其实它调用了 valueOf 方法，然后你赋的这个值很特别，是 128，那么没有进行 cache 方法，相当于 new 了两个新对象。

可以看出在创建小于 127 的对象时，相比较 new Integer(127)，使用 Integer.valueOf(127) 可以使用系统缓存，既减少可能的内存占用，也省去了频繁创建对象的开销。

所以问题中定义 a、b 的两句代码就类似于：

```
                  Integer a = new Integer(128);
                  Integer b = new Integer(128);
```

这个时候再问你，输出结果是什么？你就知道是 false 了。如果把这个数换成 127，再执行：

```
Integer a = 127;
Integer b = 127;
System.out.println(a == b);
```

结果就是：true

进行对象比较时最好还是使用 equals，便于按照自己的目的进行控制。这里引出 equals()和==，equals 比较的是字符串字面值即比较内容，==比较引用。

看一下 IntegerCache(1.6.0_13 版本)这个类里面的内容：

```
private static class IntegerCache {
    private IntegerCache() {}
    static final Integer cache[] = new Integer[-(-128) + 127 + 1];
    static {
       for (int i = 0; i < cache.length; i++)
           cache[i] = new Integer(i - 128);
    }
}
```

由于 cache[]在 IntegerCache 类中是静态数组，也就是只需要初始化一次，即 static{.....}部分，所以，如果 Integer 对象初始化时是-128~127 的范围，就不需要再重新定义申请空间，都是同一个对象---在 IntegerCache.cache 中，这样可以在一定程度上提高效率。

- 针对 String 方面的补充

  - 在同包同类下,引用自同一 String 对象.
  - 在同包不同类下,引用自同一 String 对象.
  - 在不同包不同类下,依然引用自同一 String 对象.
  - 在编译成.class 时能够识别为同一字符串的,自动优化成常量,所以也引用自同一 String 对象.
  - 在运行时创建的字符串具有独立的内存地址,所以不引用自同一 String 对象.
  - String 的 intern()方法会查找在常量池中是否存在一份 equal 相等的字符串。

    - 如果有则返回一个引用,没有则添加自己的字符串进入常量池，注意：只是字符串部分。 所以这时会存在 2 份拷贝，常量池的部分被 String 类私有并管理，自己的那份按对象生命周期继续使用。

String s = "test"

在介绍完 JVM 常量池的相关概念后，接着谈开始提到的" test "的值的内存分布的位置。对于 test 的值，实际上是在 class 文件被 JVM 装载到内存当中并被引擎在解析 ldc 指令并执行 ldc 指令之前，JVM 就已经为 test 这个字符串在常量池的 CONSTANT_String_info 表中分配了空间来存储 test 这个值。既然 test 这个字符串常量存储在常量池中，根据《深入 JAVA 虚拟机》书中描述：常量池是属于类型信息的一部分，类型信息也就是每一个被转载的类型，这个类型反映到 JVM 内存模型中是对应存在于 JVM 内存模型的方法区中，也就是这个类型信息中的常量池概念是存在于在方法区中，而方法区是在 JVM 内存模型中的堆中由 JVM 来分配的。所以，test 的值是应该是存在堆空间中的。

而对于 String s = new String("test ") ,它的 JVM 指令：

```
0:  new    #16; //class String
3:  dup
4:  ldc    #18; //String test
6:  invokespecial  #20; //Method
java/lang/String."":(Ljava/lang/String;)V
9:  astore_1
10: return
```

对于上面虚拟机指令，其各自的指令流程在《深入 JAVA 虚拟机》这样描述到(结合上面实例)：

new 指令格式：new indexbyte1,indexbyte2

new 指令过程：

要执行 new 指令，Jvm 通过计算(indextype1<<8)|indextype2 生成一个指向常量池的无符号 16 位索引。然后 JVM 根据计算出的索引查找常量池入口。该索引所指向的常量池入口必须为 CONSTANT_Class_info。如果该入口尚不存在，那么 JVM 将解析这个常量池入口，该入口类型必须是类。JVM 从堆中为新对象映像分配足够大的空间，并将对象的实例变量设为默认值。最后 JVM 将指向新对象的引用 objectref 压入操作数栈。

dup 指令格式：dup

dup 指令过程：

要执行 dup 指令，JVM 复制了操作数栈顶部一个字长的内容，然后再将复制内容压入栈。本指令能够从操作数栈顶部复制任何单位字长的值。但绝对不要使用它来复制操作数栈顶部任何两个字长(long 型或 double 型)中的一个字长。上面例中，即复制引用 objectref，这时在操作数栈存在 2 个引用。

ldc 指令格式：ldc,index

ldc 指令过程：

要执行 ldc 指令，JVM 首先查找 index 所指定的常量池入口，在 index 指向的常量池入口，JVM 将会查找 CONSTANT_Integer_info，CONSTANT_Float_info 和 CONSTANT_String_info 入口。如果还没有这些入口，JVM 会解析它们。而对于上面的 haha,JVM 会找到 CONSTANT_String_info 入口，同时，将把指向被拘留 String 对象（由解析该入口的进程产生）的引用压入操作数栈。

invokespecial 指令格式：invokespecial,indextype1,indextype2

invokespecial 指令过程：对于该类而言，该指令是用来进行实例初始化方法的调用。鉴于该指令篇幅，具体可以查阅《深入 JAVA 虚拟机》中描述。上面例子中，即通过其中一个引用调用 String 类的构造器，初始化对象实例，让另一个相同的引用指向这个被初始化的对象实例，然后前一个引用弹出操作数栈。

astore_1 指令格式：astore_1

astore_1 指令过程：

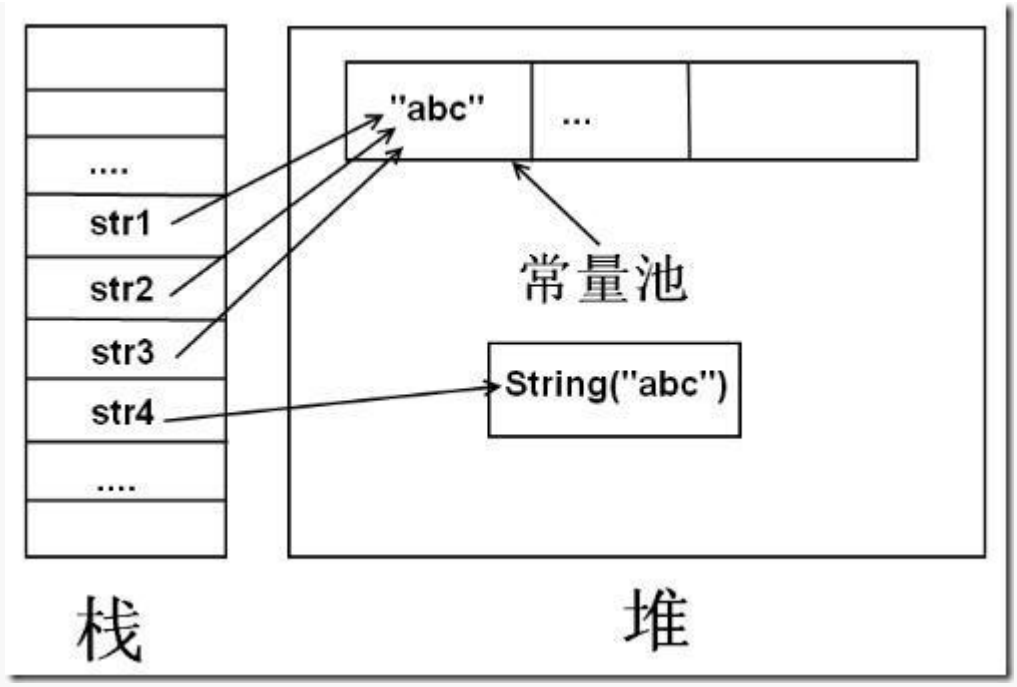
要执行 astore_1 指令，JVM 从操作数栈顶部弹出一个引用类型或者 returnAddress 类型值，然后将该值存入由索引 1 指定的局部变量中,即将引用类型或者 returnAddress 类型值存入局部变量 1。

return 指令的过程：

从方法中返回，返回值为 void。

要执行 astore_1 指令，JVM 从操作数栈顶部弹出一个引用类型或者 returnAddress 类型值，然后将该值存入由索引 1 指定的局部变量中,即将引用类型或者 returnAddress 类型值存入局部变量 1。

通过上面 6 个指令，可以看出，String s = new String("test");中的 test 存储在堆空间中，而 s 则是在操作数栈中。
上面是对 s 和 test 值的内存情况的分析和理解；那对于 String s = new String("test");语句,到底创建了几个对象呢?
我的理解：这里"test"本身就是常量池中的一个对象，而在运行时执行 new String()时，将常量池中的对象复制一份放到堆中，并且把堆中的这个对象的引用交给 s 持有。所以这条语句就创建了 2 个 String 对象。如下图所示：



String 相关的常见问题

String 中的 final 用法和理解:

```
final StringBuffer a = new StringBuffer("111");
final StringBuffer b = new StringBuffer("222");
a=b;//此句编译不通过
```

```
final StringBuffer a = new StringBuffer("111");
a.append("222");//编译通过
```

可见，final 只对引用的"值"(即内存地址)有效，它迫使引用只能指向初始指向的那个对象，改变它的指向会导致编译期错误。至于它所指向的对象的变化，final 是不负责的。

String 常量池问题的几个例子:

[1]

```
String a = "a1";
String b = "a" + 1;
System.out.println((a == b)); //result = true
String a = "atrue";
String b = "a" + "true";
System.out.println((a == b)); //result = true
String a = "a3.4";
String b = "a" + 3.4;
System.out.println((a == b)); //result = true
```

- 分析：JVM 对于字符串常量的"+"号连接，将程序编译期，JVM 就将常量字符串的"+"连接优化为连接后的值，拿"a" + 1 来说，经编译器优化后在 class 中就已经是 a1。在编译期其字符串常量的值就确定下来，故上面程序最终的结果都为 true。

[2]

```
String a = "ab";
String bb = "b";
String b = "a" + bb;
System.out.println((a == b)); //result = false
```

- 分析：JVM 对于字符串引用，由于在字符串的"+"连接中，有字符串引用存在，而引用的值在程序编译期是无法确定的，即"a" + bb 无法被编译器优化，只有在程序运行期来动态分配并将连接后的新地址赋给 b。所以上面程序的结果也就为 false。

[3]

```
String a = "ab";
final String bb = "b";
String b = "a" + bb;
System.out.println((a == b)); //result = true
```

- 分析：和[3]中唯一不同的是 bb 字符串加了 final 修饰，对于 final 修饰的变量，它在编译时被解析为常量值的一个本地拷贝存储到自己的常量池中或嵌入到它的字节码流中。所以此时的"a" + bb 和"a" + "b"效果是一样的。故上面程序的结果为 true。

[4]

```
String a = "ab";
final String bb = getBB();
String b = "a" + bb;
System.out.println((a == b)); //result = false
private static String getBB() {
   return "b";
}
```

- 分析：JVM 对于字符串引用 bb，它的值在编译期无法确定，只有在程序运行期调用方法后，将方法的返回值和"a"来动态连接并分配地址为 b，故上面程序的结果为 false。

通过上面 4 个例子可以得出得知:

```
String s = "a" + "b" + "c";
```

就等价于 String s = "abc";

```
String a = "a";
String b = "b";
String c = "c";
String s = a + b + c;
```

这个就不一样了，最终结果等于:

```
StringBuffer temp = new StringBuffer();
temp.append(a).append(b).append(c);
String s = temp.toString();
```

由上面的分析结果，可就不难推断出 String 采用连接运算符 (+) 效率低下原因分析，形如这样的代码:

```
public class Test {
public static void main(String args[]) {
    String s = null;
       for(int i = 0; i < 100; i++) {
           s += "a";
       }
    }
}
```

每做一次 + 就产生个 StringBuilder 对象，然后 append 后就扔掉。下次循环再到达时重新产生个 StringBuilder 对象，然后 append 字符串，如此循环直至结束。 如果我们直接采用 StringBuilder 对象进行 append 的话，我们可以节省 N - 1 次创建和销毁对象的时间。所以对于在循环中要进行字符串连接的应用，一般都是用 StringBuffer 或 StringBulider 对象来进行 append 操作。

String 对象的 intern 方法理解和分析:

```
public class Test4 {
  private static String a = "ab";
  public static void main(String[] args){
          String s1 = "a";
          String s2 = "b";
          String s = s1 + s2;
        System.out.println(s == a);//false
        System.out.println(s.intern() == a);//true
     }
  }
```

这里用到 Java 里面是一个常量池的问题。对于 s1+s2 操作，其实是在堆里面重新创建了一个新的对象,s 保存的是这个新对象在堆空间的的内容，所以 s 与 a 的值是不相等的。而当调用 s.intern()方法，却可以返回 s 在常量池中的地址值，因为 a 的值存储在常量池中，故 s.intern 和 a 的值相等