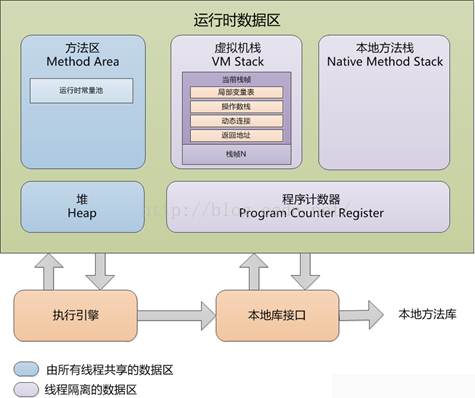
### Java 常量池技术



（Jvm内存模型）

## 1.1 程序计数器

《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

### 1.1.1 什么是程序计数器？

程序计数器是一块较小的内存空间，可以把它看作当前线程正在执行的字节码的行号指示器。也就是说，程序计数器里面记录的是当前线程正在执行的那一条字节码指令的地址。

**注意**：如果当前线程正在执行的是一个本地方法，那么此时程序计数器为空。

### 1.1.2程序计数器的作用

程序计数器有两个作用：

1).字节码解释器通过改变程序计数器来依次读取指令，从而实现代码的流程控制，如：顺序执行、选择、循环、异常处理。

2).在多线程的情况下，程序计数器用于记录当前线程执行的位置，从而当线程被切换回来的时候能够知道该线程上次运行到哪儿了。

### 1.1.3 程序计数器的特点

1).是一块较小的存储空间

2).线程私有。每条线程都有一个程序计数器。

3).是唯一一个不会出现OutOfMemoryError的内存区域。

4).生命周期随着线程的创建而创建，随着线程的结束而死亡。

## 1.2 Java虚拟机栈(JVM Stack)

### 1.2.1什么是Java虚拟机栈？

Java虚拟机栈是描述Java方法运行过程的内存模型。

Java虚拟机栈会为每一个即将运行的Java方法创建一块叫做“栈帧”的区域，这块区域用于存储该方法在运行过程中所需要的一些信息，这些信息包括：

1).局部变量表、存放基本数据类型变量、引用类型的变量、returnAddress类型的变量

2).操作数栈

3).动态链接

4).方法出口信息

当一个方法即将被运行时，Java虚拟机栈首先会在Java虚拟机栈中为该方法创建一块“栈帧”，栈帧中包含局部变量表、操作数栈、动态链接、方法出口信息等。

当方法在运行过程中需要创建局部变量时，就将局部变量的值存入栈帧的局部变量表中。

当这个方法执行完毕后，这个方法所对应的栈帧将会出栈，并释放内存空间。

**注意：**人们常说，Java的内存空间分为“栈”和“堆”，栈中存放局部变量，堆中存放对象。

这句话不完全正确！这里的“堆”可以这么理解，但这里的“栈”只代表了Java虚拟机栈中的局部变量表部分。真正的Java虚拟机栈是由一个个栈帧组成，而每个栈帧中都拥有：局部变量表、操作数栈、动态链接、方法出口信息。

### 1.2.2 Java虚拟机栈的特点

1.局部变量表的创建是在方法被执行的时候，随着栈帧的创建而创建。而且，局部变量表的大小在编译时期就确定下来了，在创建的时候只需分配事先规定好的大小即可。此外，在方法运行的过程中局部变量表的大小是不会发生改变的。

2.Java虚拟机栈会出现两种异常：StackOverFlowError和OutOfMemoryError。

2.1) StackOverFlowError：

若Java虚拟机栈的内存大小不允许动态扩展，那么当线程请求栈的深度超过当前Java虚拟机栈的最大深度的时候，就抛出StackOverFlowError异常。

2.2) OutOfMemoryError：

若Java虚拟机栈的内存大小允许动态扩展，且当线程请求栈时内存用完了，无法再动态扩展了，此时抛出OutOfMemoryError异常。

3.Java虚拟机栈也是线程私有的，每个线程都有各自的Java虚拟机栈，而且随着线程的创建而创建，随着线程的死亡而死亡。

**注意**：StackOverFlowError和OutOfMemoryError的异同？

StackOverFlowError表示当前线程申请的栈超过了事先定好的栈的最大深度，但内存空间可能还有很多。 而OutOfMemoryError是指当线程申请栈时发现栈已经满了，而且内存也全都用光了。

## 1.3 本地方法栈

本地方法栈和Java虚拟机栈实现的功能类似，只不过本地方法区是本地方法运行的内存模型。本地方法被执行的时候，在本地方法栈也会创建一个栈帧，用于存放该本地方法的局部变量表、操作数栈、动态链接、出口信息。方法执行完毕后相应的栈帧也会出栈并释放内存空间。也会抛出StackOverFlowError和OutOfMemoryError异常。

## 1.4 堆

### 1.4.1什么是堆

堆是用来存放对象的内存空间，几乎所有的对象都存储在堆中。

### 1.4.2堆的特点

1).线程共享

整个Java虚拟机只有一个堆，所有的线程都访问同一个堆。而程序计数器、Java虚拟机栈、本地方法栈都是一个线程对应一个的。

2).在虚拟机启动时创建

3).垃圾回收的主要场所。

4).可以进一步细分为：新生代、老年代。

新生代又可被分为：Eden、From Survior、To Survior。

不同的区域存放具有不同生命周期的对象。这样可以根据不同的区域使用不同的垃圾回收算法，从而更具有针对性，从而更高效。

5).堆的大小既可以固定也可以扩展，但主流的虚拟机堆的大小是可扩展的。

因此当线程请求分配内存，但堆已满，且内存已满无法再扩展时，就抛出OutOfMemoryError。

## 1.5 方法区

### 1.4.1什么是方法区

Java虚拟机规范中定义方法区是堆的一个逻辑部分。

方法区中存放已经被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等。

### 1.4.2方法区特点

1).线程共享

方法区是堆的一个逻辑部分，因此和堆一样，都是线程共享的。整个虚拟机中只有一个方法区。

2).永久代

方法区中的信息一般需要长期存在，而且它又是堆的逻辑分区，因此用堆的划分方法，我们把方法区称为老年代。

3).内存回收效率低

方法区中的信息一般需要长期存在，回收一遍内存之后可能只有少量信息无效。

对方法区的内存回收的主要目标是：对常量池的回收 和 对类型的卸载。

4). Java虚拟机规范对方法区的要求比较宽松。

和堆一样，允许固定大小，也允许可扩展的大小，还允许不实现垃圾回收。

## 综述

1).Java虚拟机的内存模型中一共有两个“栈”，分别是：Java虚拟机栈和本地方法栈。

2).两个“栈”的功能类似，都是方法运行过程的内存模型。并且两个“栈”内部构造相同，都是线程私有。

只不过Java虚拟机栈描述的是Java方法运行过程的内存模型，而本地方法栈是描述Java本地方法运行过程的内存模型。

3).Java虚拟机的内存模型中一共有两个“堆”，一个是原本的堆，一个是方法区。方法区本质上是属于堆的一个逻辑部分。堆中存放对象，方法区中存放类信息、常量、静态变量、即时编译器编译的代码。

堆是Java虚拟机中最大的一块内存区域，也是垃圾收集器主要的工作区域。

4).程序计数器、Java虚拟机栈、本地方法栈是线程私有的，即每个线程都拥有各自的程序计数器、Java虚拟机栈、本地方法区。并且他们的生命周期和所属的线程一样。

5).而堆、方法区是线程共享的，在Java虚拟机中只有一个堆、一个方法栈。并在JVM启动的时候就创建，JVM停止才销毁。

## 1.6 运行时常量池

**1**).方法区中存放三种数据：类信息、常量、静态变量，以及即时编译器编译后的代码。

其中常量存储在运行时常量池中。如果一个类中通过public static final来声明一个常量。这个类被编译后生成class文件，这个类的所有信息都存储在class文件中。

当这个类被Java虚拟机加载后，class文件中的常量就存放在方法区的运行时常量池中。而且在运行期间，可以向常量池中添加新的常量。如：String类的intern()方法就能在运行期间向常量池中添加字符串常量。

当运行时常量池中的某些常量没有被对象引用，同时也没有被变量引用，那么就需要垃圾收集器回收

**2**).常量池用于保存在编译期已确定的、已编译的 class 文件中关于类，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。

**3**).Java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。

## 1.7 查看常量池虚拟机指令

对于String str = "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 (将int, float或String型常量值从常量池中推送至栈顶)
* 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：**str**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**str**的值是存在栈当中。上面是对于**str**值得分析，接着是对于"**Test**"值的分析,我们知道，对于String str = "**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？

### 1.7.1 JVM常量池

关于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。

### 1.7.2 JVM常量池结构

* **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12种常量表，如下所示：

|  |  |  |
| --- | --- | --- |
| 类型 | 标志值(1byte) | 描述 |
| CONSTANT\_utf8\_info | 1 | UTF-8编码的字符串 |
| CONSTANT\_Integer\_info | 3 | 整形字面量 |
| 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 | 字段或方法的符号引用 |
| CONSTANT\_MethodHandle\_info | 15 | 表示方法句柄 |
| CONSTANT\_MothodType\_info | 16 | 标志方法类型 |
| CONSTANT\_InvokeDynamic\_info | 18 | 表示一个动态方法调用点 |

* **(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)。

### 1.7.3 拘留字符串对象

* **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

}

}

### 1.7.4 Integer对象的代码补充

* **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。

### 1.7.5 String方面的代码补充

* **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 str = "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;//Methodjava/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：

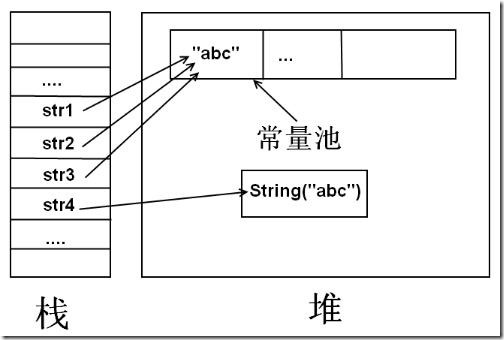
要执行astore\_1指令，JVM从操作数栈顶部弹出一个引用类型或者returnAddress类型值，然后将该值存入由索引1指定的局部变量中,即将引用类型或者returnAddress类型值存入局部变量1。

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：



### 1.7.7 String相关常见问题

**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。