

常量池梳理及相关练习

详情见 CSDN

一、常量池的分类及含义

(1) 静态常量池

所谓静态常量池，即*.class 文件中的常量池（Constant pool 部分），占用 class 文件绝大部分空间，这种常量池主要用于存放两大类常量：字面量(Literal)和符号引用量(Symbolic References)，

- 字面量包括：文本字符串、被声明为 final 的常量值，基本数据类型，通常表现为等号的右值
如：Integer a = 2;
2 就是字面量
- 符号引用量包括：类和结构的完全限定名、字段名称和描述符、方法名称和描述符
如：Integer a = 2;
Integer 和 a 就是符号引用
下面以简单类进行讲解

```
package com.jvm;

public class JVMmodel {
    public static void main(String[] args) {
        A a = new A();
        System.out.println(a.width);
        System.out.println(a.str);
    }
}

class A{
    public static final String str = "helloword";
    public static int width = 100;
    static {
        System.out.println("静态初始化类A");
        width = 300;
    }
    public A(){
        System.out.println("创建A类的对象");
    }
}
```

字面量（其实就是一些变量的具体值）	文本字符串	Helloword（因为在 A 类中是用 final 限定的，所以编译阶段直接进入 JVMmodel 的常量池）
	被声明为 final 的常量值	
	基本数据类型	无
符号引用量（其实就是名字：类名，方法名，变量名）	类和结构的完全限定名	包名+类名 如上图中 import 之后 JVMmodel、A、a、System
	字段名称	Width、str
	方法名称	Out、println
	描述符	Public、static、

注意：Helloworld 因为是用 final 限定的，所以在编译阶段就会进入到 JVMmodel 的常量池中

```
Constant pool:
#1 = Methodref      #10.#19      // java/lang/Object.<init>():V
#2 = Class           #20           // com/jvm/A
#3 = Methodref      #2.#19        // com/jvm/A.<init>():V
#4 = Fieldref        #21.#22      // java/lang/System.out:Ljava/io/PrintStream;
#5 = Fieldref        #2.#23        // com/jvm/A.width:I
#6 = Methodref      #24.#25      // java/io/PrintStream.println:(I)V
#7 = String          #26           // helloworld
#8 = Methodref      #24.#27      // java/io/PrintStream.println:(Ljava/lang/String;)V
#9 = Class           #28           // com/jvm/JVMmodel
#10 = Class          #29           // java/lang/Object
#11 = Utf8           <init>
#12 = Utf8           ()V
#13 = Utf8           Code
#14 = Utf8           LineNumberTable
#15 = Utf8           main
#16 = Utf8           ([Ljava/lang/String;)V
#17 = Utf8           SourceFile
#18 = Utf8           JVMmodel.java
#19 = NameAndType    #11:#12      // <init>():V
#20 = Utf8           com/jvm/A
#21 = Class          #30           // java/lang/System
#22 = NameAndType    #31:#32      // out:Ljava/io/PrintStream;
#23 = NameAndType    #33:#34      // width:I
#24 = Class          #35           // java/io/PrintStream
#25 = NameAndType    #36:#37      // println:(I)V
#26 = Utf8           helloworld
#27 = NameAndType    #38:#38      // println:(Ljava/lang/String;)V
#28 = Utf8           com/jvm/JVMmodel
#29 = Utf8           java/lang/Object
#30 = Utf8           java/lang/System
#31 = Utf8           out
#32 = Utf8           Ljava/io/PrintStream;
#33 = Utf8           width
#34 = Utf8           I
#35 = Utf8           java/io/PrintStream
#36 = Utf8           println
#37 = Utf8           (I)V
#38 = Utf8           (Ljava/lang/String;)V
{
  public static void main(String[] args) {
    ...
  }
}
```

同理，A 的常量池为

字面量（其实就是一些变量的具体值）	文本字符串	helloworld	静态初始化类 A、创建 A 类的对象
	被声明为 final 的常量值		
	基本数据类型	100、300	
符号引用量（其实就是名字：类名，方法名，变量名）	类和结构的完全限定名	包名+类名 如上图中 import 之后 A、System	
	字段名称	Width、str	
	方法名称	Out、println	
	描述符	Public、static、final	

```

flags: ACC_SUPER
Constant pool:
#1 = Methodref      #8.#22          // java/lang/Object.<init>():V
#2 = Fieldref       #23.#24          // java/lang/System.out:Ljava/io/PrintStream;
#3 = String         #25              // 创建A类的对象
#4 = Methodref      #26.#27          // java/io/PrintStream.println:(Ljava/lang/String;)V
#5 = Fieldref       #7.#28           // com/jvm/A.width:I
#6 = String         #29              // 静态初始化类A
#7 = Class          #30              // com/jvm/A
#8 = Class          #31              // java/lang/Object
#9 = Utf8           str
#10 = Utf8          Ljava/lang/String;
#11 = Utf8          ConstantValue
#12 = String        #32              // helloworld
#13 = Utf8          width
#14 = Utf8          I
#15 = Utf8          <init>
#16 = Utf8          ()V
#17 = Utf8          Code
#18 = Utf8          LineNumberTable
#19 = Utf8          <clinit>
#20 = Utf8          SourceFile
#21 = Utf8          JVMModel.java
#22 = NameAndType   #15:#16          // <init>():V
#23 = Class         #33              // java/lang/System
#24 = NameAndType   #34:#35          // out:Ljava/io/PrintStream;
#25 = Utf8          创建A类的对象
#26 = Class         #36              // java/io/PrintStream
#27 = NameAndType   #37:#38          // println:(Ljava/lang/String;)V
#28 = NameAndType   #13:#14          // width:I
#29 = Utf8          静态初始化类A
#30 = Utf8          com/jvm/A
#31 = Utf8          java/lang/Object
#32 = Utf8          helloworld
#33 = Utf8          java/lang/System
#34 = Utf8          out
#35 = Utf8          Ljava/io/PrintStream;
#36 = Utf8          java/io/PrintStream
#37 = Utf8          println
#38 = Utf8          (Ljava/lang/String;)V

```

Class 文件在加载到 JVM 中后，会变成运行常量池和字符串常量池

(2) 运行时常量池和字符串常量池

① 概念

运行时常量池（Runtime Constant Pool）是方法区的一部分。Class 文件常量池里的内容将在类加载后存放方法区的运行时常量池中，并在类加载的解析阶段，将符号引用转换为直接引用，指向真正的内存地址

字符串常量池里的内容是在类加载之后，代码执行在堆中生成字符串对象实例，然后将该字符串对象实例的引用值存到 string pool 中，被所有的类共享。

需要注意的是运行时常量池是静态常量池加载到 JVM 后形成的，但是字符串常量池是代码运行过程中产生的字符串示例对象；

② 位置

在 JDK 6 或更早之前常量池都是分配在永久代中



JDK7 及以前使用永久代代替方法区，JDK8 使用元空间来代替方法区，不管怎样 **JDK7 及以后运行时常量池存储在方法区，字符串常量池存储在堆中**

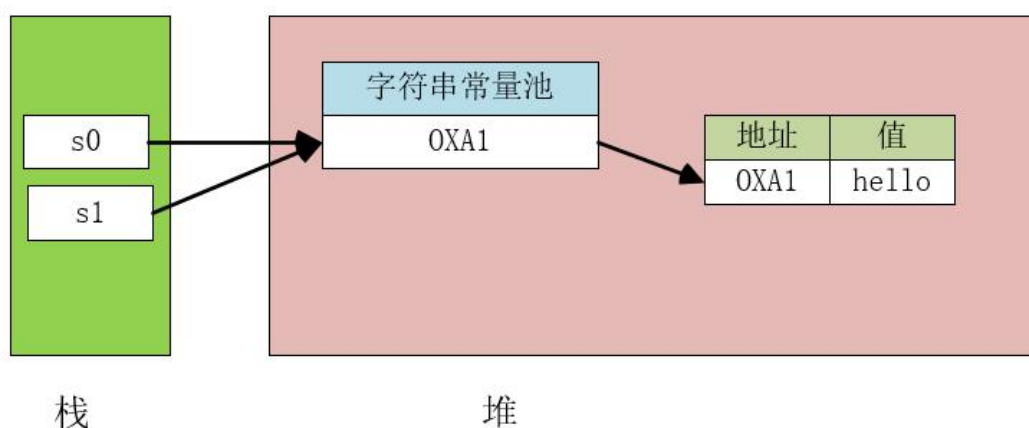


(3) 字符串常量区详解

① 直接赋值方式

- 使用字面量赋值，只会直接分配到字符串常量池中。如果字符串常量池中存在则直接返回，如果不存在则先创建，然后返回。
- 这里需要注意字符串常量池中存储的是引用值而不是实例对象，具体的实例对象存储在堆中。

```
String s0="hello";  
String s1="hello";  
System.out.println( s0==s1 ); //true
```



- 如果是字面量或被声明为 **final** 的符号引用之间相加，会被认为是字符串常量池中的字符串相互拼接
- **注意：这种情况下，final 修饰的符号引用量必须被字面量初始化！！，否则不适用如：**

```

public static final String A; // 常量A
public static final String B; // 常量B

static {
    A = "he";
    B = "llo";
}

private static void example11() {
    String s1 = A + B;
    String s2 = "hello";
    System.out.println(s1 == s2); // false
}

```

没有被初始化

静态块也不行

所以是

✗

```

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

private static String getBB()
{
    return "b";
}

```

静态方法更不行

所以是 false

✗

```

String a = "hello";
final String bb = "llo";
String b = "he" + bb;
System.out.println(a == b); // true

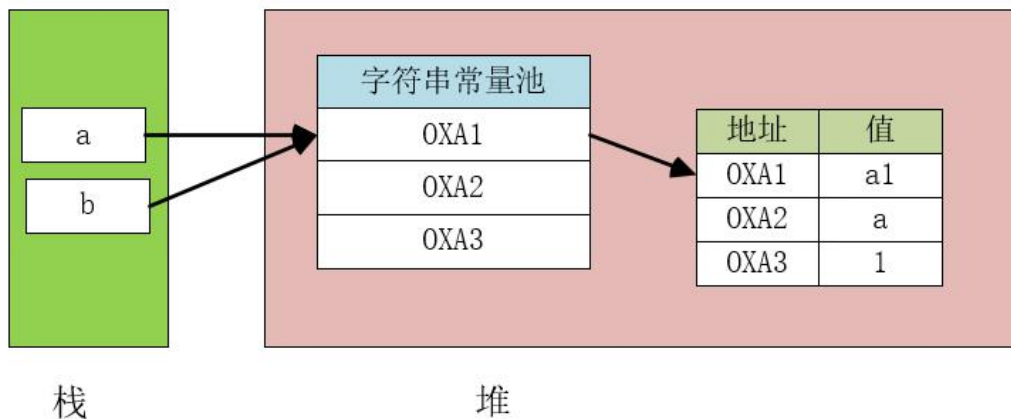
```

初始化

对的，所以为true

✓

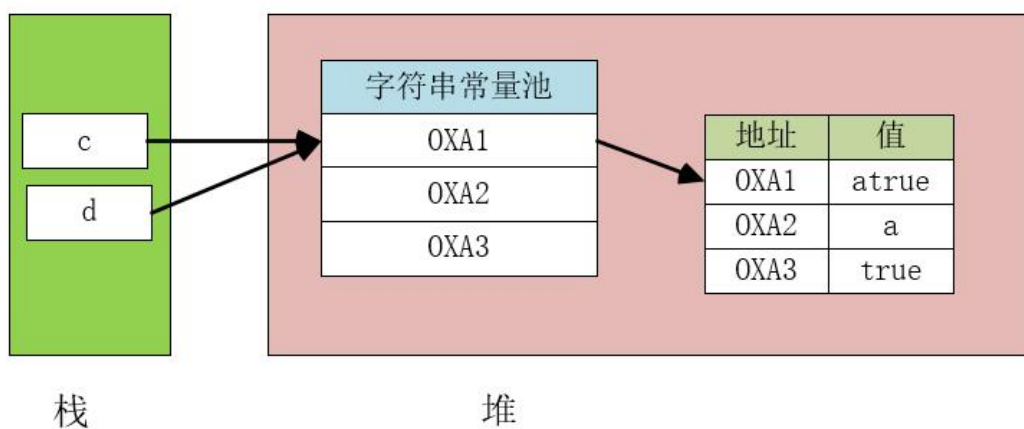

```
String a = "a1";
String b = "a" + 1;
System.out.println(a == b); // true
```



说明:

- 首先，字面量为: a1,a,1
- 在编译阶段，“a” + 1 时，会自动的将 1 转化为字符串常量，放置在字符串常量池中
- 所以会检测字符串常量池中是否存在字面量: a1,a,1，没有则添加
- 然后将栈中的 a 指向字符串常量池中的 a1
- 然后将字符串常量池中的 a 和 1 进行拼接，为 a1，并检测字符串常量池中是否有 a1，没有则添加
- 然后将栈中的 b 指向字符串常量池中的 a1

```
String c = "atrue";
String d = "a" + true;
System.out.println(c == d); // true
```

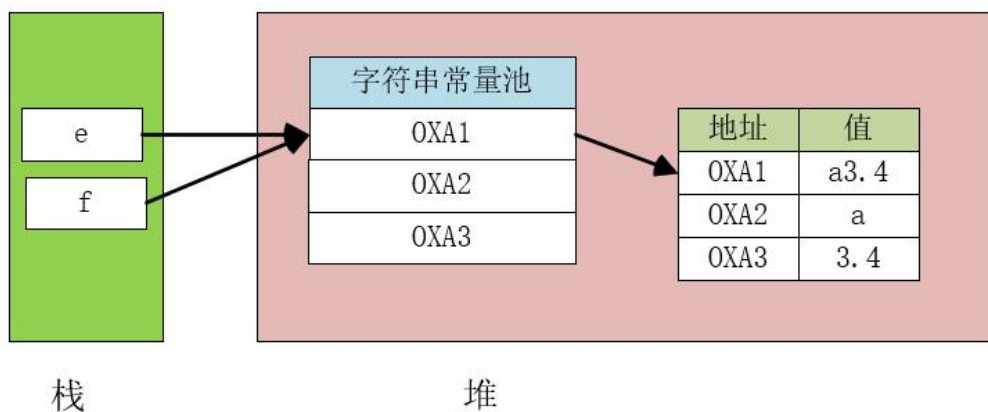


说明:

- 同理，字面量是: atruer、a、true

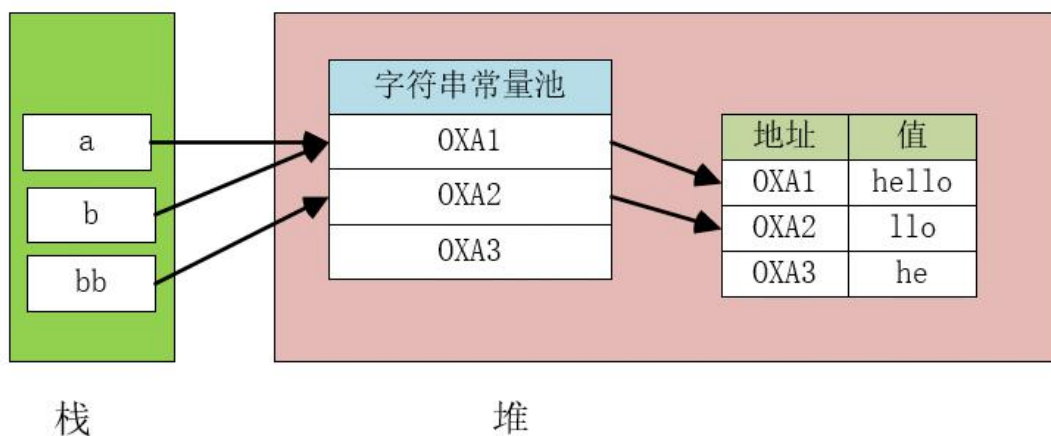
- 在编译阶段, "a" + true,会自动的将 true 转化为字符串常量, 放置在字符串常量池中
- 所以会检测字符串常量池中是否存在字面量: atrue、a、true, 没有则添加
- 然后将栈中的 c 指向字符串常量池中的 atrue
- 然后将字符串常量池中的 a 和 true 进行拼接, 为 atrue, 并检测字符串常量池中是否有 atrue, 没有则添加
- 然后将栈中的 c 指向字符串常量池中的 atrue

```
String e = "a3.4";
String f = "a" + 3.4;
System.out.println(e == f); // true
```



说明: 以上同理

```
String a = "hello";
final String bb = "llo";
String b = "he" + bb;
System.out.println(a == b); // true
```



说明:

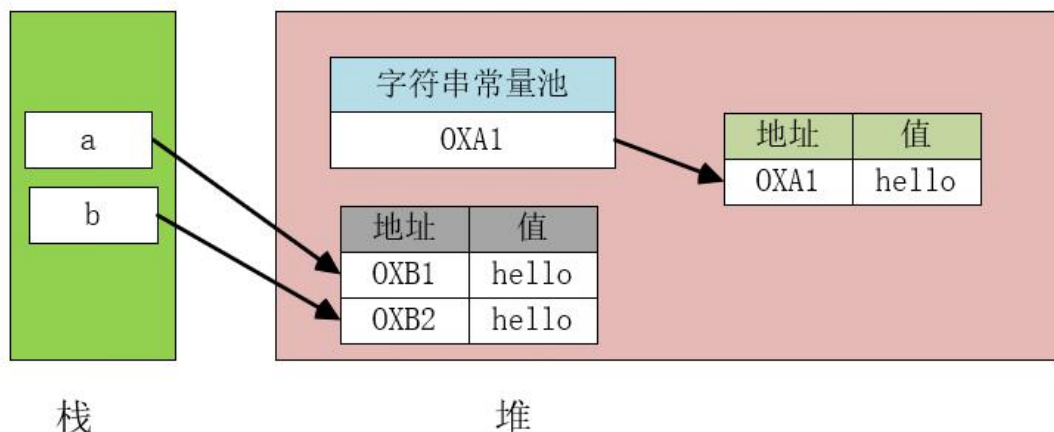
- 与被 final 修饰的符号引用量相加, 可以认为是直接和字面量相加, 所以具体过程如下:

- `String a = "hello"` : 先检测字符串常量池中是否有"hello"这个字符串, 没有则添加
- 然后将栈中的 `a` 指向字符串常量池中的 `hello`
- `final String bb = "llo"` : 先检测字符串常量值中是否有"llo"这个字符串, 没有则添加
- 然后将栈中的 `bb` 指向字符串常量池中的 `llo`
- `String b = "he" + bb` : 如果符号引用量 `bb` 不是被 `final` 修饰, 就需要按照下面的讲解, 但是被 `final` 修饰, 就可以直接认为是和字面量"llo"相加, 为 `hello`
- 然后将栈中的 `b` 指向字符串常量池中的 `hello`

② `new String("xxxx")`创建方式

- 首先会在字符串常量池中, 检测 `new String("xxx")`中的字面量是否存在, 没有则创建
- 然后会在堆中开辟一块空间, 创建 `String` 对象, 并将地址返回给栈

```
String a = new String("hello");
String b = new String("hello");
System.out.println(a == b); // false
```



- 这里需要特别说一下, 涉及非 `final` 声明的符号引用量, 以及 `new String("xxx")` 的相加, 不管它两其中一个或者任意一个, 只要位于 "+" 号的左边或者右边, 底层实际使用的是 `StringBuilder.append` 进行拼接, 然后再使用 `StringBuilder.toString` 返回, 我们可以使用工具来进行观察

```
private static void example5() {
    String s0 = "hello";
    String s1 = new String( original: "hello");
    String s2 = "he" + new String( original: "llo");
    System.out.println(s0 == s1); // false
    System.out.println(s0 == s2); // false
    System.out.println(s1 == s2); // false
}
```

```

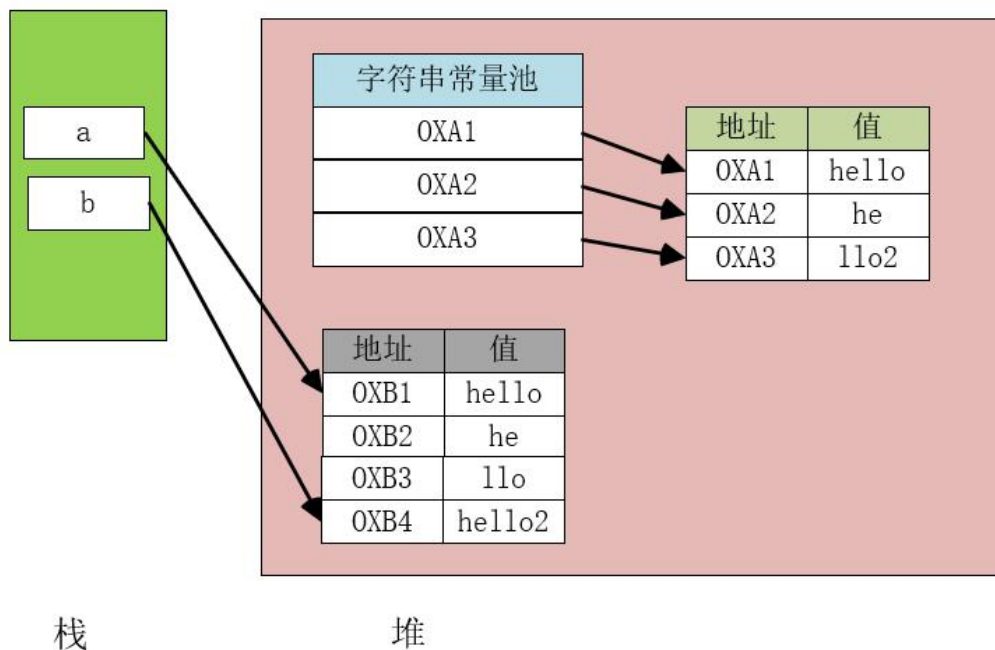
17 invokespecial #12 <java/lang/StringBuilder.<init> : ()V>
20 ldc #13 <he>
22 invokevirtual #14 <java/lang/StringBuilder.append : (Ljava/lang/String;)Ljava/lang/StringBu
25 new #9 <java/lang/String>
28 dup
29 ldc #15 <llo>
31 invokespecial #10 <java/lang/String.<init> : (Ljava/lang/String;)V>
34 invokevirtual #14 <java/lang/StringBuilder.append : (Ljava/lang/String;)Ljava/lang/StringBu
37 invokevirtual #16 <java/lang/StringBuilder.toString : ()Ljava/lang/String;>
40 astore_2
41 getstatic #4 <java/lang/System.out : Ljava/io/PrintStream;>
```


- 再来看下 `StringBuilder` 类的 `toString` 源码，底层还是一个 `new String` 操作，注意这个 `new String()` 是不产生字面量的。

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

- 所以，涉及非 `final` 声明的符号引用量或者 `new String()` 的相加就等于 `new String()` 的效果
- 1) `new String("xxx")` 和 `new String("yyy")` 相加

```
String a = new String("hello");
String b = new String("he") + new String("llo2");
System.out.println(a == b); // false
```

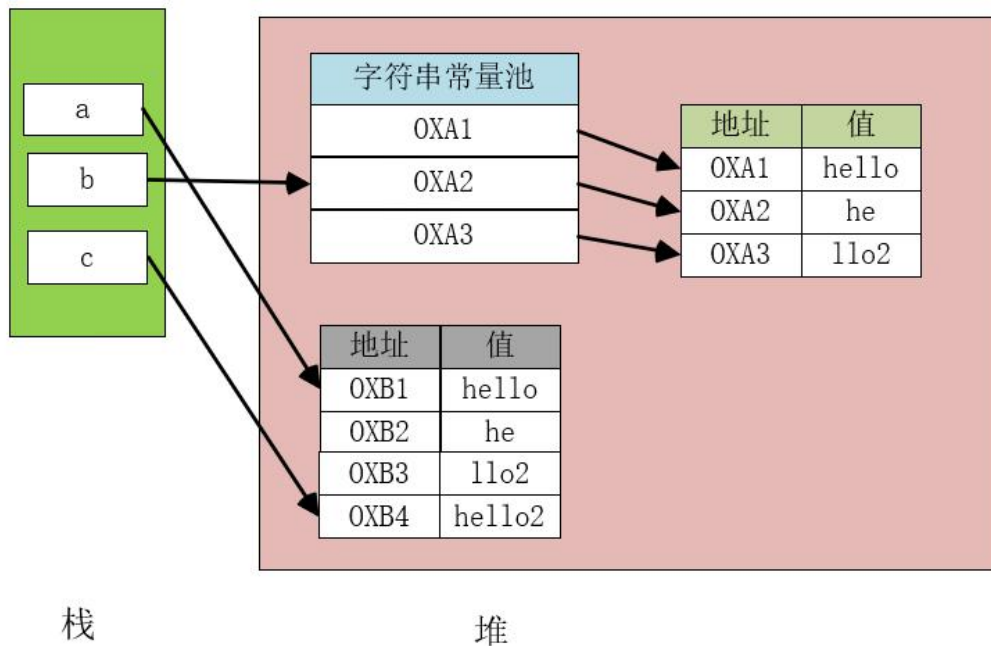


说明：

- `String a = new String("hello");`
- 字面量: `hello`, 首先会检测字符串常量池中是否有 `hello`，没有则添加到字符串常量池中
- 然后在堆中创建对象 `new String("hello")`，并将地址返回到栈中
- `String b = new String("he") + new String("llo2");`
- 字面量: `he` 和 `llo2`, 首先会检测字符串常量池中是否有 `he` 和 `llo2`，没有则添加到字符串常量池中
- 然后会在堆中创建 `new String("he")` 对象和 `new String("llo2")` 对象
- 然后就是 `new String("he") + new String("llo2")`，根据上述说明，是使用 `new String()` 拼接，但是不会产生拼接后的字面量 `hello2`，直接在堆中创建 `new String("hello2")`，将引用返回到栈中；

2) new String("xxx")和符号引用量相加

```
String a = new String("hello");
String b = "he";
String c = b + new String("llo2");
System.out.println(a == c); // false
```

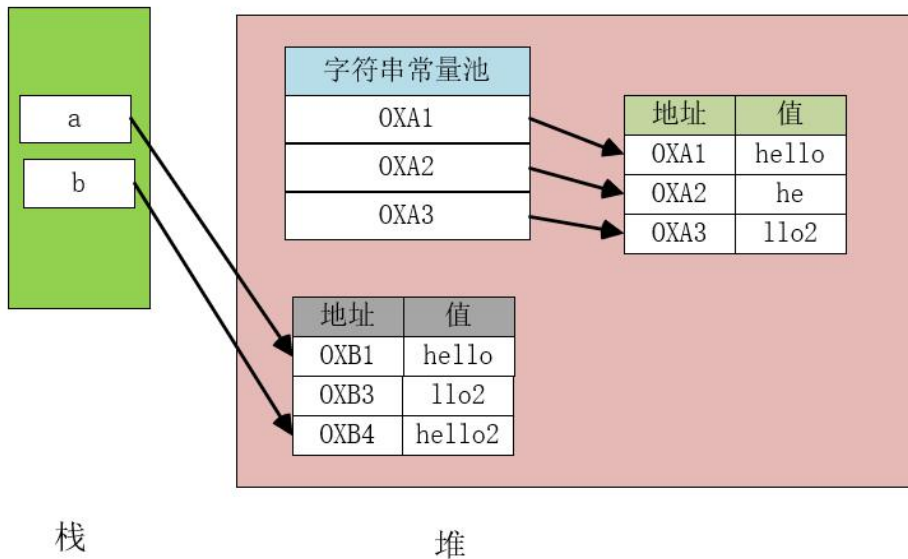


说明:

- `String a = new String("hello");`
- 字面量: `hello`, 首先会检测字符串常量池中是否有 `hello`, 没有则添加到字符串常量池中
- 然后在堆中创建对象 `new String("hello")`, 并将地址返回到栈中
- `String b = "he";`
- 字面量: `he`, 首先会检测字符串常量池中是否有 `he`, 没有则添加到字符串常量池中
- 由于采用的是直接赋值, 所以会直接将字符串常量池中的地址返回到栈中
- `String c = b + new String("llo2");`
- 字面量: `llo2`, 首先会检测字符串常量池中是否有 `llo2`, 没有则添加到字符串常量池中
- 然后在堆中创建对象 `new String("llo2")`
- 然后就是 `b + new String("llo2")`, 根据上述说明, 是使用 `new String()` 拼接, 但是不会产生拼接后的字面量 `hello2`, 直接在堆中创建 `new String("hello2")`, 将引用返回到栈中;

3) new String("xxx")和字面量相加

```
String a = new String("hello");
String b = "he" + new String("llo2");
System.out.println(a == b); // false
```

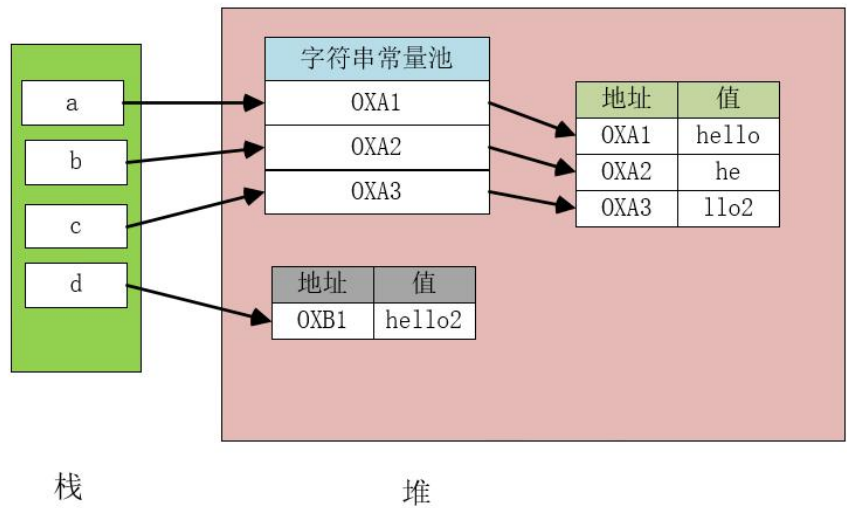


说明：

- `String a = new String("hello");`
- 字面量: `hello`, 首先会检测字符串常量池中是否有 `hello`, 没有则添加到字符串常量池中
- 然后在堆中创建对象 `new String("hello")`, 并将地址返回到栈中
- `String b = "he" + new String("llo2");`
- 字面量: `he` 和 `llo2`, 首先会检测字符串常量池中是否有 `he` 和 `llo2`, 没有则添加到字符串常量池中
- 然后在堆中创建对象 `new String("llo2")`
- 然后就是 `"he" + new String("llo2")`, 根据上述说明, 是使用 `new String()` 拼接, 但是不会产生拼接后的字面量 `hello2`, 直接在堆中创建 `new String("hello2")`, 将引用返回到栈中

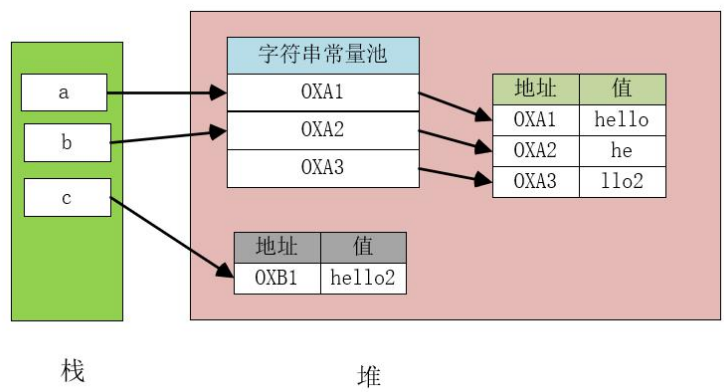
4) 符号引用量和符号引用相加

```
String a = "hello";
String b = "he";
String c = "llo2";
String d = b + c;
System.out.println(a == d); // false
```



- 说明:
- String a = "hello";String b = "he";String c = "llo2";
 - 字面量: hello, he, llo2, 首先会检测字符串常量池中是否有 hello, he, llo2, 没有则添加到字符串常量池中
 - 由于是直接赋值方式, 会将字符串常量池中的地址直接返回到栈中
 - String d = b + c;
 - 根据上述说明, 是使用 new String() 拼接, 但是不会产生拼接后的字面量 hello2, 直接在堆中创建 new String("hello2"), 将引用返回到栈中
- 5) 符号引用量和字面量相加

```
String a = "hello";
String b = "he";
String c = b+ "llo2";
System.out.println(a == c); // false
```



说明:

- `String a = "hello";String b = "he";`
- 字面量: `hello, he`, 首先会检测字符串常量池中是否有 `hello, he`, 没有则添加到字符串常量池中
- `String c = b+ "llo2";`
- 字面量: `llo2`, 首先会检测字符串常量池中是否有 `llo2`, 没有则添加到字符串常量池中
- 然后 `b+ "llo2"`: 根据上述说明, 是使用 `new String()` 拼接, 但是不会产生拼接后的字面量 `hello2`, 直接在堆中创建 `new String("hello2")`, 将引用返回到栈中

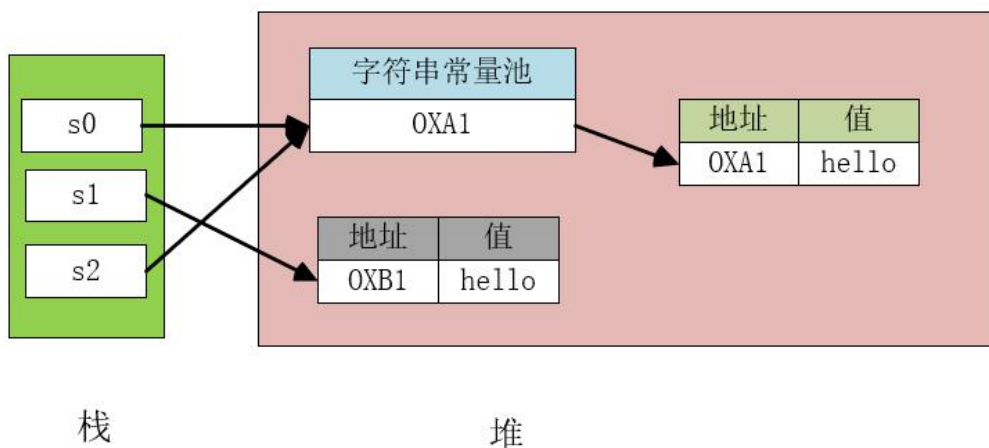
6) 字面量和字面量相加

见上

③ Intern 方法创建

`intern` 方法, 返回 `String` 对象所对应的字面量在常量池中的引用, 如果常量池中没有, 则添加到常量池中, 并返回常量池中的引用;

```
String s0 = "hello";
String s1 = new String("hello");
String s2 = s1.intern();
System.out.println(s0 == s1); //false
System.out.println(s1 == s2); //false
```



说明:

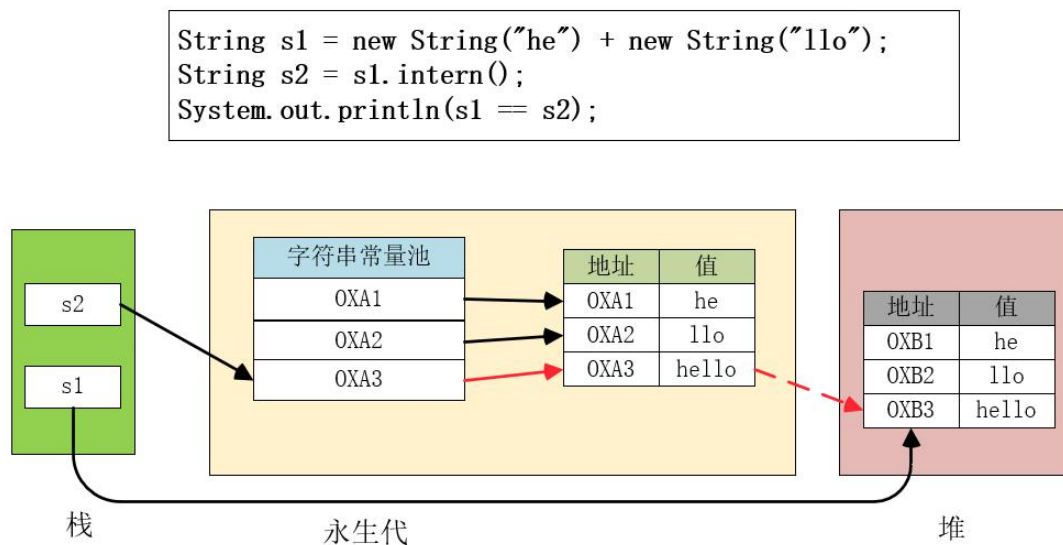
- `String s0 = "hello";`
- 字面量: `hello`, 首先会检测字符串常量池中是否有 `hello`, 没有则添加到字符串常量池中
- `String s1 = new String("hello");`
- 字面量: `hello`, 首先会检测字符串常量池中是否有 `hello`, 没有则添加到字符串常量池中
- 然后在堆中创建对象 `new String("hello")`, 并将地址返回到栈中

- `String s2 = s1.intern();`
- 根据上面规则，首先会检测字符串常量池中是否有对应的字面量：“hello”，返回在常量池中的引用

注意，存在一种情况，堆对象的 `intern`，但是堆对象所对应的字面量，并不存在于字符串常量池中，就需要分两种情况来考虑

- JDK1.6 及之前：

在 JDK1.6 及之前，常量池是存在于永生代中的，所以在执行堆对象的 `Intern()` 方法后，会检测字符串常量池是否含有堆对象所对应的字符串常量，没有则会在永生代中创建，并将引用放到字符串常量池中，返回到栈中



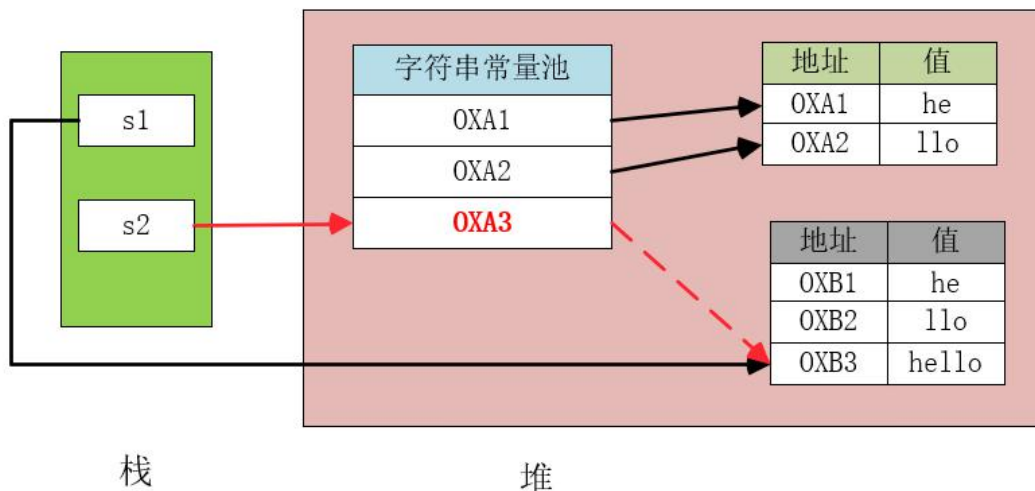
说明：

- `String s1 = new String("he") + new String("llo");`
- 字面量：he 和 llo, 首先会检测字符串常量池中是否有 he 和 llo，没有则添加到字符串常量池中
- 然后会在堆中创建 `new String("he")` 对象和 `new String("llo")` 对象
- 然后就是 `new String("he") + new String("llo")`, 根据上述说明，是使用 `new String()` 拼接，但是不会产生拼接后的字面量 hello，直接在堆中创建 `new String("hello")`, 将引用返回到栈中；
- `String s2 = s1.intern();`
- 首先会检测 s1 对应地字面量“hello”，是否存在于永生代中的字符串常量池中，没有则在永生代中创建，并将引用放到字符串常量池中，返回到栈中

- JDK1.7 及以后

在 JDK1.7 以后，由于取消了永生代概念，并且将常量池放入到了堆中，所以在执行堆对象的 Intern()方法后，会检测字符串常量池是否含有堆对象所对应的字符串常量，没有就会将堆对象的地址放入到字符串常量池中，并将字符串常量池中的引用，返回到栈。

```
String s1 = new String("he") + new String("llo");
String s2 = s1.intern();
System.out.println(s1 == s2);
```



说明：

- `String s1 = new String("he") + new String("llo");`
- 字面量: `he` 和 `llo`, 首先会检测字符串常量池中是否有 `he` 和 `llo`, 没有则添加到字符串常量池中
- 然后会在堆中创建 `new String("he")` 对象和 `new String("llo")` 对象
- 然后就是 `new String("he") + new String("llo")`, 根据上述说明, 是使用 `new String()` 拼接, 但是不会产生拼接后的字面量 `hello`, 直接在堆中创建 `new String("hello")`, 将引用返回到栈中;
- `String s2 = s1.intern();`
- 会检测字符串常量池是否含有堆对象所对应的字符串常量, 没有就会将堆对象的地址放入到字符串常量池中, 并将字符串常量池中的引用, 返回到栈。

(4) 基本数据类型的包装类的池技术

在 8 种基本类型种, 除 `float` 和 `double`, 其他六种基本类型对应的包装类都应用了池技术(`Byte`, `Short`, `Integer`, `Long`, `Character`, `Boolean`)

池技术大致分为三个技术要点: 常量池, 自动装箱以及自动拆箱

- 常量池

在包装类进行 JVM 加载的时候, 会用静态块/静态变量初始化一个包装类数组, 数组内部的数据是该包装类被常用到的数据, 我们称之为常量池或者包装类缓存。

如: `Integer/Long` 类型, 数组范围为: `-128~127`, `Boolean` 类型为 `true/false`

源码截图如下:

```
Integer.java x Long.java x
private static 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) {
            try {
                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);
            } catch (NumberFormatException nfe) {
                // If the property cannot be parsed into an int, ignore it.
            }
        }
        high = h;

        cache = new Integer[(high - low) + 1];
        int j = low;
        for(int k = 0; k < cache.length; k++)
            cache[k] = new Integer(j++);

        // range [-128, 127] must be interned (JLS7 5.1.7)
        assert IntegerCache.high >= 127;
    }
}
```

源码
最小值
包装类数组
最大值
初始化数组
数组加入数据

```
Long.java x LongCache
private LongCache(){}

static final Long cache[] = new Long[-(-128) + 127 + 1];

static {
    for(int i = 0; i < cache.length; i++)
        cache[i] = new Long(value: i - 128);
}
}
```

源码
静态块
初始化数组

```
java x Integer.java x Long.java x Boolean.java x
*/
public final class Boolean implements java.io.Serializable,
    Comparable<Boolean>
{
    /**
     * The {@code Boolean} object corresponding to the primitive
     * value {@code true}.
     */
    public static final Boolean TRUE = new Boolean( value: true);

    /**
     * The {@code Boolean} object corresponding to the primitive
     * value {@code false}.
     */
    public static final Boolean FALSE = new Boolean( value: false);
}
```

- 自动装箱

自动装箱往往发生在用字面量给包装类进行赋值时，底层实际调用的是 `valueOf()` 方法，如果该字面量位于包装类的缓存数组的范围中，则返回该数组中对应的对象；如果该字面量不位于包装类的缓存数组的范围中，则返回堆中的 `new` 对象

```
*/
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
```

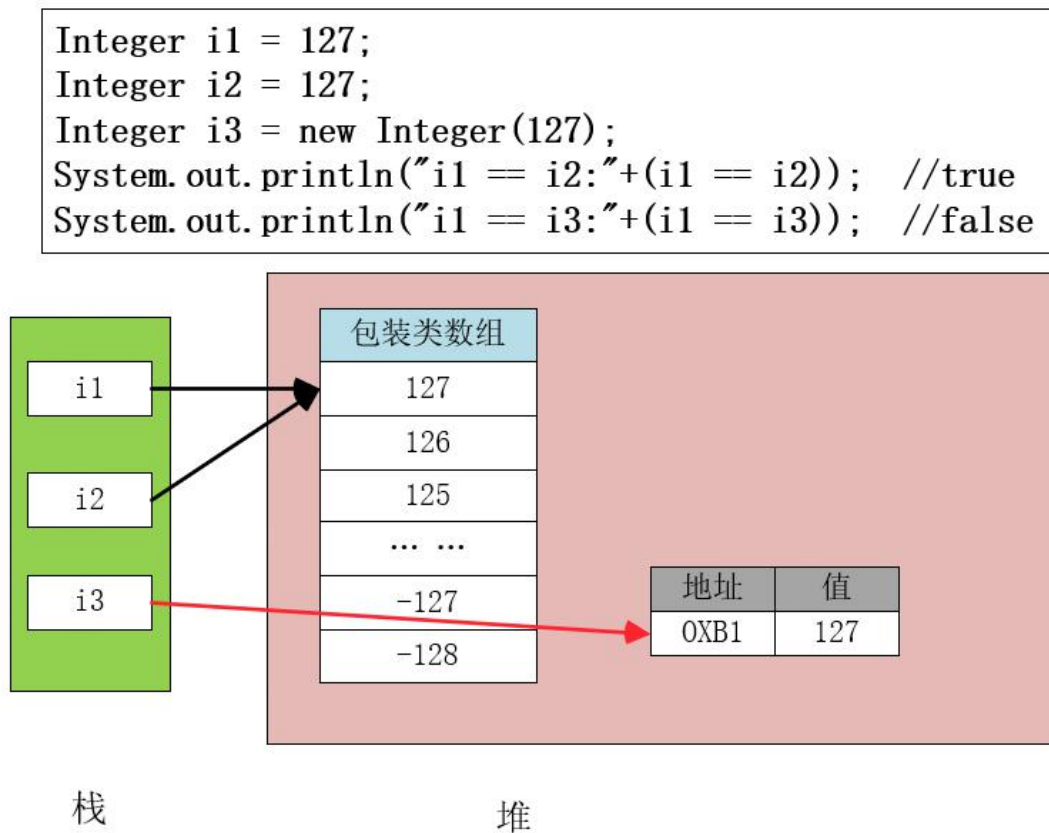
```
// 等效
Integer i1 = 127;
Integer i2 = Integer.valueOf(127);

// 等效
Long l1 = 127L;
Long l2 = Long.valueOf(127L);

// 等效
Boolean b1 = true;
Boolean b2 = Boolean.valueOf(true);

//还有其它六种类型.....
```

当然在使用 new 包装类的方式进行赋值，即使数据范围位于包装类数组缓存的范围，也会返回堆中新创建的对象，而不是数组中的对象。



- 自动拆箱

当包装类型参与数学运算（加、减、乘、除、与、或、非等）（包装类型之间、包装类型和基本类型），包装类型会调用 intValue() 方法自动转为基本类型，然后参与运算，包装类型自动转为基本类型的过程叫自动拆箱；

在计算完后，又会调用 valueOf() 方法，进行自动装箱(见上面的自动装箱理论)



下面是因为超过了自动装箱的范围（Integer -128~127），所以报 false

```
@Test
public void test18(){
    Integer i1 = 128;
    Integer i = 120;
    Integer i2 = 8 + i;
    System.out.println(i1 == i2); //false
}
```

等价于

```
@Test
public void test19(){
    Integer i1 = 127;
    Integer i2 = 7 + 120;
    System.out.println(i1 == i2); //false
}
```

等价于

```
@Test
public void test20(){
    Integer i1 = 127;
    Integer i2 = Integer.valueOf(7 + 120);
    System.out.println(i1 == i2); //false
}
```

当然还有一种自动拆箱发生在基本数据类型和包装类的比较（等于、小于、大于等），包装类会自动转为基本数据类型进行比较

```
@Test
public void test21(){
    Integer i1 = 2000;
    System.out.println(2000 == i1); // true
    System.out.println(1999 < i1); // true
    System.out.println(1999 > i1); // false
}
```

Integer i1 = 2000 会自动转为 int i1 = 2000 进行比较

包装类和包装类的比较，等于(==)不会发生自动拆箱，大于(>)小于(<)会发生自动拆箱，即包装类和包装类进行等于比较的时候，不会转为基本数据类型，但是进行大于小于比较的时候会转为基本数据类型进行比较

```
@Test
public void test22(){
    Integer i1 = 2000;
    Integer i2 = 1999;
    System.out.println(i2 == i1); // false
    System.out.println(i2 < i1); // true
    System.out.println(i2 > i1); // false
}
```

不会自动拆箱

会自动拆箱