# Java中的8大基本类型& 内存中占有的字节 & 初始值？



Float 1 8 23

Double 1 11 52

# 带符号数（正数/负数）在计算机中存储方式？

带符号数的存储方式是以补码的形式存储

原码：原码就是符号位加上真值的绝对值, 即用第一位表示符号, 其余位表示值。

补码：正数的补码就是其本身。负数的补码是取反+1.

补码是计算机存储带符号数的方式，可以解决0的符号问题以及两个编码问题。

1能够统一+0和-0的表示

2对于有符号整数的运算能够把符号位同数值位一起处理

3能够简化运算规则（把减法运算转换为加法运算）

# 为什么byte类型（8位二进制表示）的取值范围为-128~127？

0用[0000 0000]表示，-128用[1000 0000]表示。byte类型1个字节，8位二进制，范围为[1000 0000]~[0111 1111]

# Int强转为Byte的过程

1、int转为2进制并求补码

2、补码转为byte类型精度丢失（最高位为符号位）

3、再将补码转回原码即为byte的值

byte b=(byte)178;

此时b的值变成了-78，为什么呢？

首先，Java中变量，或者说计算机底层是以补码保存，int类型178强转byte流程应该是这样：

int类型178二进制表示：

补码=原码/反码：0(符号位)000，0000，0000，0000，0000，0000，1011，0010（先变为补码再转换类型）

转为byte类型后，因精度丢失，补码表示为1(符号位)011，0010。

对整数的补码再求一次补码即为整数原码；

二进制数1001110转换为十进制是78，符号位1表示负数，结果即为-78。

# 浮点数（小数）在计算机中存储方式？

1、将小数转换为二进制科学记数法，得到符号位、指数和小数部分

2、将指数部分加上偏移量并转换为二进制

3、将3个部分的二进制连起来，剩余部分补0

1.小数（浮点数）的二进制转换

78.375 的整数部分：



小数部分：



所以，78.375 的二进制形式就是 1001110.011

然后，使用二进制科学记数法，有



注意，转换后用二进制科学记数法表示的这个数，有底有指数有小数部分，这个就叫做浮点数。

2.浮点数在计算机中存储

在计算机中，保存这个数使用的是浮点表示法，分为三大部分：

第一部分用来存储符号位（sign），用来区分正负，这里是 0，表示正数

第二部分用来存储指数（exponent），这里的指数是十进制的 6

第三部分用来存储小数（fraction），这里的小数部分是 001110011



指数位决定了大小范围，因为指数位越大则能表示的数越大，而小数位决定了计算精度，因为小数位越大能计算的精度越大。

float类型是32位，是单精度浮点表示法：

符号位占用1位，指数位占用 8 位，小数位占用 23 位。

float 的小数位只有 23 位，即二进制的 23 位，能表示的最大的十进制数为 2 的 23 次方，即 8388608，即十进制的 7 位，严格点，精度只能百分百保证十进制的 6 位运算。

double 类型是 64 位，是双精度浮点表示法：

符号位占用 1 位，指数位占用 11 位，小数位占用 52 位。

double 的小数位有 52 位，对应十进制最大值为 4 503 599 627 370 496，这个数有 16 位，所以计算精度只能百分百保证十进制的 15 位运算。

# 指数位是如何储存的

将指数加上偏移量转化为无符号整数存储，float是127，double是1023

float 的指数部分是 8 位，则指数的取值范围是 -126 到 +127（指数不允许使用0和128），为了消除负数带来的实际计算上的影响（比如比较大小，加减法等），可以在实际存储的时候，需要把指数转换为无符号整数，即给指数做一个简单的映射，加上一个偏移量，比如float的指数偏移量为 127，这样就不会有负数出现了。比如：指数如果是 6，则实际存储的是 6+127=133，即把 133 转换为二进制之后再存储。

对应的 double 类型，存储的时候指数偏移量是 1023。

举例求78.375浮点数表示

所以用float类型来保存十进制小数78.375的话，需要先转换成浮点数，得到符号位和指数和小数部分。符号位是0，指数位是6+127=133，二进制表示为10 000 101，小数部分是001110011，不足部分请自动补0。

连起来用 float 表示，加粗部分是指数位，最左边是符号位 0，代表正数：

0 **10000101** 001110011 00000 00000 0000

# 知道float和double类型为什么会出现精度丢失的情况吗？

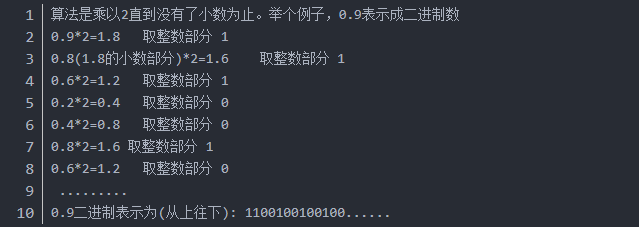
因为存储的尾部位数有限，且二进制的表示有时也是不可能精确的，比如无法表示1/10。

浮点型数据精度丢失的原因（乘2算法）

将十进制浮点数转换为二进制浮点数时，小数的二进制有时也是不可能精确的。

就如同十进制不能准确表示1/3，二进制也无法准确表示1/10，而double类型存储尾数部分最多只能存储52位，于是，计算机在存储该浮点型数据时，便出现了精度丢失。

例：十进制小数如何转化为二进制数（乘2算法）



注意：上面的计算过程循环了，也就是说\*2永远不可能消灭小数部分，这样算法将无限下去。很显然，小数的二进制表示有时是不可能精确的 。

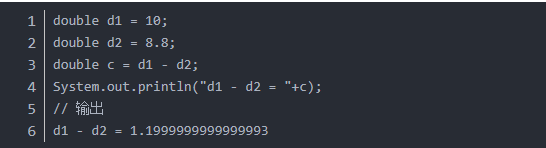
因此将11.9化为二进制后大约是” 1011. 1110011001100110011001100…”。

# **浮点型数据精度丢失的解决方法**

1可以将数据库表的price实体属性单位精确到分，并将字段类型设置为int，用来表示包括小数点后的所有数字，后续再由程序员手动转换。

2在浮点数计算时使用java的BigDecimal类

商业运算中应用场景：例如某用户有10块钱，买了一件商品花了8.8，理应剩下1.2元。但却无法继续购买价格为1.2元的商品。



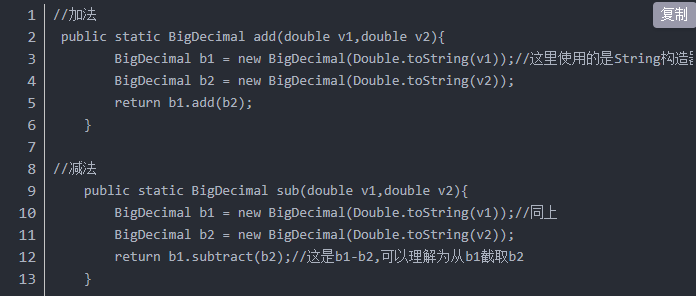
解决方法1

在设计数据库表的时候可以将price字段类型设置为int(oracle应设置为number)类型，而在实体中对应的属性单位应该表示为分（即精确到0.00）或者角(即0.0),但一般情况下money会精确到分。

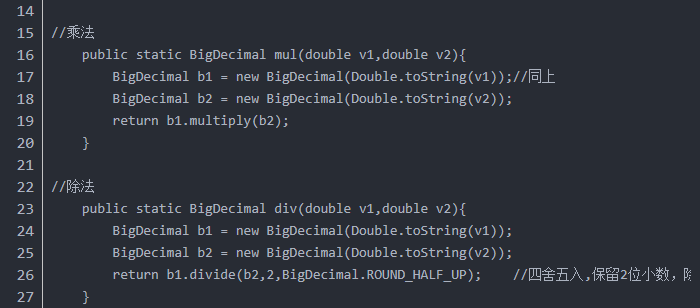
如：商品的价格为12.53元(精确到分)，在数据库中price字段对应的数据为应该为1253。使用这种方法需要编程人员自己在程序中收懂转换，当然也可以封装为一个工具类。

解决方法2

使用java提供的BigDecimal类。该类封装在java.math.BigDecimal中。该类的构造器有很多，但在使用浮点类型计算时一定要使用String构造器来实例BigDecimal对象。



3 //这里使用的是String构造器，将double转换为String类型



26 //四舍五入,保留2位小数，除不尽的情况

# JAVA基本数据类型与封装类型的区别？

封装类（如Integer）是基本数据类型（如int）的包装类。



# 什么是拆箱 & 装箱，能给我举栗子吗？

封装类（如Integer）是基本数据类型（如int）的包装类。装箱就是 自动将基本数据类型转换为包装器类型；拆箱就是 自动将包装类型转换为基本数据类型。

装箱（基本数据类型->封装类）

Integer i = 10; // 实际上执行Integer.valueOf(10);

拆箱（封装类->基本数据类型）

Integer i = 10; //装箱

int t = i; //拆箱，实际上执行了 int t = i.intValue();

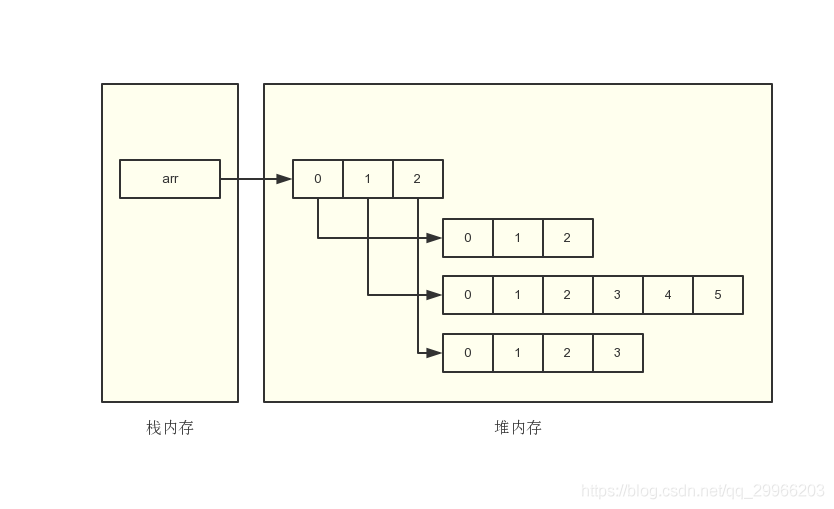
# 能说说多维数组在内存上是怎么存储的吗？

在java中数组也是对象。因此，对象存放在内存中的原理同样适用于数组。

当创建一个数组时，在堆中会为数组对象分配一段内存空间，并返回一个引用。数组对象的引用存放在栈中，实际的数组对象存放在堆中。

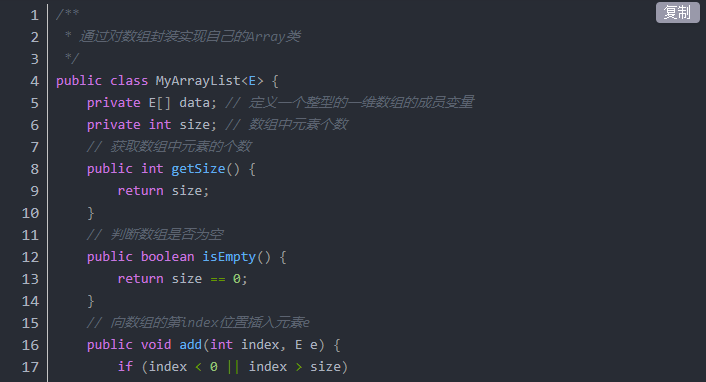
多维数组在内存中存储方式：

数组对象的引用指向多维数组的头部，而每一维数组的头部又指向其对应维的数组。

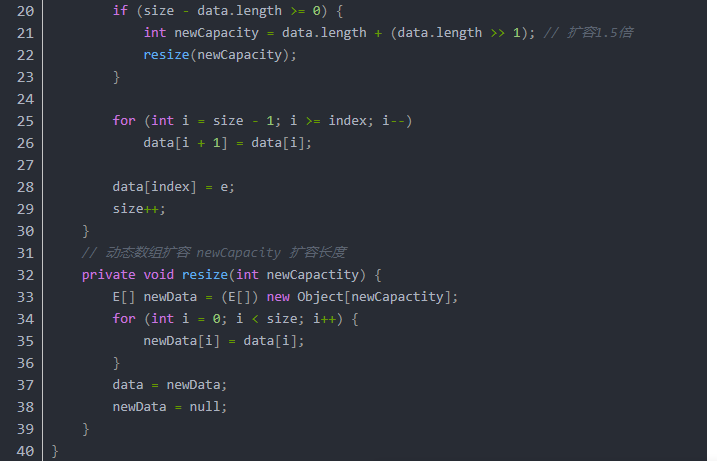


# 你对数组二次封装过吗？说说封装了什么？

使用Java一维数组，仿照ArrayList源码，封装相关构造、获取元素个数、容量大小、判空、增删查改等功能，得到有特定功能的数组。  
[对Java一维数组E[]自定义ArrayList集合](https://blog.csdn.net/u011035026/article/details/83820427)  
下面是部分实现：



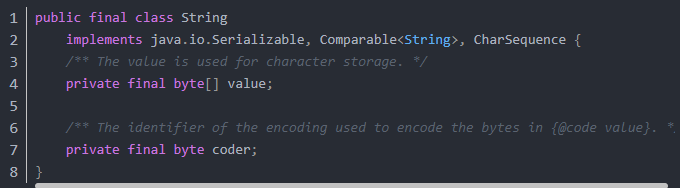
throw new IllegalArgumentException("Add failed. Require index >= 0 and index <= size.");



# String 的实现原理

在 Java 8 中，String 内部使用 char 数组存储数据。

在 Java 9 之后，String 类的实现改用 byte 数组value存储字符串，同时使用 coder 来标识使用了哪种编码。



# String 不可变性的实现原理

1、String被声明为final，因此不可被继承。

2、所有成员都为私有变量且用final修饰，保证成员变量初始化后不被修改。

3、初始化时对传入对象进行深拷贝。

4、不提供setter方法。

5、getter方法返回的是对象的深拷贝。

String对象是不可变的，即对象的状态（成员变量）在对象创建之后不再改变。

（一）不可变性实现

由String内部构造：

（1）String 被声明为 final，因此它不可被继承。(Integer 等包装类也不能被继承）

（2）value 数组被声明为 final，这意味着 value 数组初始化之后就不能再引用其它数组。

（3）String 内部没有改变 value 数组的方法。

可知String是不可变的。

补充：不可变的实现：

1String类被final修饰，保证类不被继承。

2String内部所有成员都设置为私有变量，并且用final修饰符修饰，保证成员变量初始化后不被修改。

3不提供setter方法改变成员变量，即避免外部通过其他接口修改String的值。

4通过构造器初始化所有成员（value[]）时，对传入对象进行深拷贝（deep copy），避免用户在String类以外通过改变这个对象的引用来改变其内部的值。

5在getter方法中，不要直接返回对象引用，而是返回对象的深拷贝，防止对象外泄。

# 深拷贝与浅拷贝（和赋值）的区别

深拷贝：创建新对象，所有的值都相等。

浅拷贝：创建新对象，除引用类型只拷贝地址其他所有的值都相等。

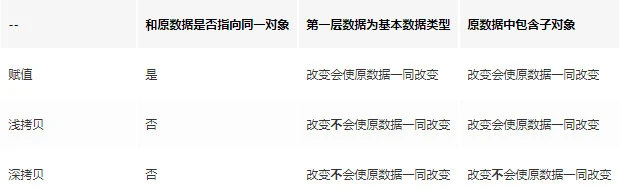
赋值：将一个对象的地址给另一个变量。

深拷贝和浅拷贝是只针对Object和Array这样的引用数据类型的。

赋值将一个对象在栈中的地址给一个新的变量，此时两个变量指向的是同一个存储空间。

浅拷贝会创建一个新对象，并把旧对象的值复制一份给新对象，若旧对象内存在属性值为引用数据类型时，只拷贝地址。

深拷贝会另外创造一个一模一样的对象，新对象跟原对象不共享内存，修改新对象不会改到原对象。



# String不可变的好处

1可以创建字符串常量池String Pool（有助于共享）

2线程安全考虑

3支持hash映射和缓存

1满足字符串常量池的需要（有助于共享）

可以将字符串对象保存在字符串常量池中以供与字面值相同字符串对象共享。

如果一个 String 对象已经被创建过了，那么就会从 String Pool 中取得引用。只有 String 是不可变的，才可能使用 String Pool。

如果String对象是可变的，那就不能这样共享，因为一旦对某一个String类型变量引用的对象值改变，将同时改变一起共享字符串对象的其他 String类型变量所引用的对象的值。

2线程安全考虑

同一个字符串实例可以被多个线程共享。字符串的不变性保证字符串本身便是线程安全的。

3支持hash映射和缓存

因为字符串是不可变的，所以在它创建的时候hashcode就被缓存了，不需要重新计算。这就使得String很适合作为Map中的键，字符串的处理速度要快过其它的键对象。所以HashMap中的键往往都使用字符串。

# String不可变的缺点：

String对象不适用于经常发生修改的场景，会创建大量的String对象。

（三）String 的 “改变”？

public static void main(String[] args) {

String s = "ABCDEF";

System.out.println("s = " + s);

s = "123456";

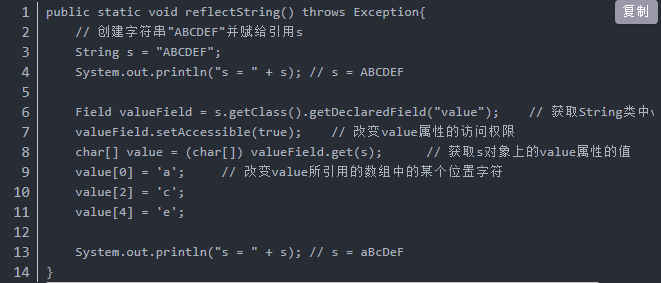
System.out.println("s = " + s);

}

String的改变实际上是创建了一个新的String对象"123456"，并将引用指向了这个新的对象，同时原来的String对象"ABCDEF"并没有发生改变，仍保存在内存中。

# String 的不可变 真的不可变？

通过反射获取value数组直接改变内存数组中的数据是可以修改所谓的"不可变"对象的。



6// 获取String类中value字段

# String && StringBuilder && StringBuffer

* 可变性  
  String 不可变  
  StringBuffer 和 StringBuilder 可变
* 线程安全  
  String 不可变，因此是线程安全的  
  StringBuilder 不是线程安全的  
  StringBuffer 是线程安全的，内部使用 synchronized 进行同步



# String在内存中的存储

对于String，其对象的引用都是存储在栈中的。

字符串常量—String Pool字符串常量池（堆）

字符串对象—非字符串常量池（堆）

java中对String对象特殊对待，所以在heap堆区域分成了两块，一块是字符串常量池(String constant pool)，用于存储java字符串常量对象，另一块用于存储普通对象及字符串对象。

1"abc"字符串常量/s.intern()——StringPool

编译期已经创建好(直接用双引号定义的"abc")的就存储在字符串常量池中。即jvm会在String constant pool中创建对象。字符串常量池（String Pool）保存着所有字符串字面量（literal strings），这些字面量在编译时期就确定。不仅如此，还可以使用 String 的 intern() 方法在运行过程中将字符串添加到 String Pool 中。String Pool用于共享字符串字面量，防止产生大量String对象导致OOM。

jvm会首先在String constant pool 中寻找是否已经存在（equals方法）“abc"常量，如果没有则创建该常量，并且将此常量的引用返回给String a；如果已有"abc” 常量，则直接返回String constant pool 中“abc” 的引用给String a。

当一个字符串调用 intern() 方法时，与上述过程相同。

equals相等（指向同一引用）的字符串在常量池中永远只有一份。

intern() 方法返回字符串对象的规范化表示形式，即一个字符串，内容与此字符串相同，但一定取自具有唯一字符串的池。

它遵循以下规则：对于任意两个字符串 s 和 t，当且仅当 s.equals(t) 为 true 时，s.intern() == t.intern() 才为 true。

2new String(“abc”)

运行期（new出来的 new String(s)）才能确定的就存储在堆中。即jvm会直接在heap中非String constant pool 中创建字符串对象，然后把该对象引用返回给String b（并且不会把"abc” 加入到String constant pool中）。

new就是在堆中创建一个新的String对象，不管"abc"在内存中是否存在，都会在堆中开辟新空间。

equals相等的字符串在堆中可能有多份。

对于 new String(“abc”)，使用这种方式一共会创建两个字符串对象（前提是 String Pool 中还没有 “abc” 字符串对象）。这两个字符串对象指向同一个value数组。

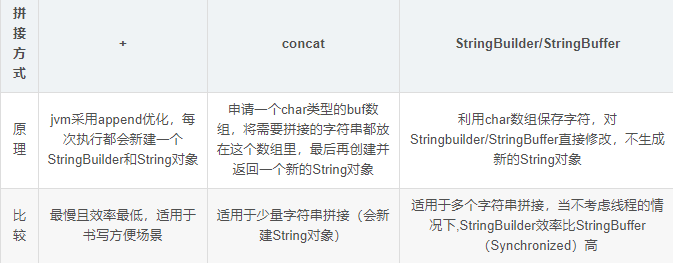
# 字符串拼接方式 & 比较

拼接方式

1、“+”（StringBuilder）

2、concat（char数组）

3、StringBuilder/StringBuffer的append（char数组）



1"+" 拼接

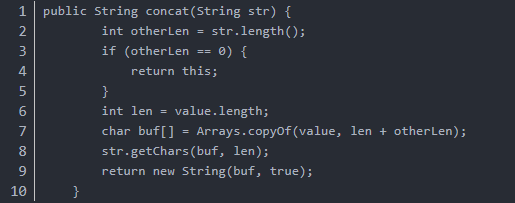
加号拼接字符串jvm底层其实是调用StringBuilder来实现的，也就是说”a” + “b” + "c"等效于下面的代码片。



但并不是说直接用“+”号拼接就可以达到StringBuilder的效率了，因为每次使用 "+"拼接 都会新建一个StringBuilder对象，并且最后toString()方法还会生成一个String对象。在循环拼接十万次的时候，就会生成十万个StringBuilder对象，会产生大量内存消耗

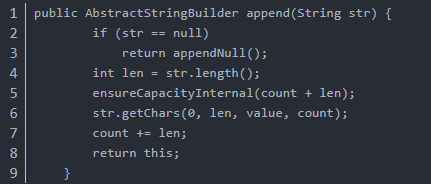
**2concat 拼接**

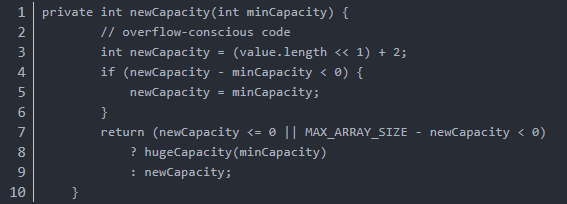
concat其实就是申请一个char类型的buf数组，将需要拼接的字符串都放在这个数组里，最后再创建并返回一个新的String对象



**3StringBuilder/StringBuffer append**

这两个类实现append的方法都是调用父类AbstractStringBuilder的append方法，只不过StringBuffer的append方法加了sychronized关键字，因此是线程安全的。append代码如下，他主要也是利用char数组保存字符，通过ensureCapacityInternal方法来保证数组容量可用还有扩容。



他扩容的方法的代码如下，可见，当容量不够的时候，数组容量左移1位（也就是翻倍）再加2。  


# String a = “a”+“b”+“c”;在内存中创建了几个对象？

1个对象，String pool对象

String a = “a”+“b”+"c"经过编译器优化后得到的效果为String a = “abc”

java编译期会进行常量折叠，全字面量字符串相加是可以折叠为一个字面常量，而且是进入常量池的。

在JAVA虚拟机（JVM）中存在着一个字符串池，其中保存着很多String对象，并且可以被共享使用，因此它提高了效率。由于String类是final的，它的值一经创建就不可改变，因此我们不用担心String对象共享而带来程序的混乱。字符串池由String类维护，我们可以调用intern()方法来访问字符串池。

对于String a=“abc”;，这行代码被执行的时候，JAVA虚拟机首先在字符串池中查找是否已经存在了值为"abc"的这么一个对象，它的判断依据是String类equals(Object obj)方法的返回值。如果有，则不再创建新的对象，直接返回已存在对象的引用；如果没有，则先创建这个对象，然后把它加入到字符串池中，再将它的引用返回。

字符串内部拼接：只有使用引号包含文本的方式创建的String对象之间使用“+”连接产生的新对象才会被加入字符串池中。对于所有包含new方式新建对象（包括null）的“+”连接表达式，它所产生的新对象都不会被加入字符串池中。

# **String s=new String(“abc”)创建了几个对象？**

**2个对象**

new String(“abc”)可看成"abc"（创建String pool对象）和new String(String original)（String构造器，创建String对象）2个对象。

我们正是使用new调用了String类的上面那个构造器方法创建了一个对象，并将它的引用赋值给了str变量。同时我们注意到，被调用的构造器方法接受的参数也是一个String对象，这个对象正是"abc"。

String s = new String("abc")实际上是"abc"本身就是文字池中的一个对象，在运行 new String()时，把文字池即pool中的字符串"abc"复制到堆中，并把这个对象的应用交给s，所以创建了两个String对象，一个在pool 中，一个在堆中。