**一 java内存分配**

这里只是在网上找的一些资料;   
[Java](http://lib.csdn.net/base/javase) 中的数据类型分为   
1. 基本类型（原始数据类型） byte short int long float double char boolean   
基本类型的变量持有原始值。   
2. 符合数据类型（引用类型），引用类型持有引用值（即对某个对象的引用，而非对象本身）。

一般Java在内存分配时会涉及到以下区域：   
1. 寄存器：我们在程序中无法控制   
2. 栈：存放**基本类型**的数据和对象的**引用**，**但对象本身不存放在栈中，而是存放在堆中**   
3. 堆：存放用new产生的数据   
4. 静态域：存放在对象中用static定义的静态成员   
5. 常量池：存放常量   
6. 非RAM存储：硬盘等永久存储空间   
其中主要是堆，栈的存储。

**堆，栈**

1. 函数中定义的一些**基本类型的数据变量**和**对象的引用变量**都在函数的栈内存中分配。   
   栈的**优势**是存取速度比堆要快，仅次于直接位于CPU 的寄存器，而且数据可以共享。但是存在栈中的数据大小与生存周期必须是确定的。   
   当在一段代码块定义一个变量时，Java就在**栈中** 为这个变量分配内存空间，当该变量退出该作用域后，Java会自动释放掉为该变量所分配的内存空间，该内存空间可以立即被另作他用。
2. **堆内存**用来存放由new创建的对象和数组。 在堆中分配的内存，由Java虚拟机的自动垃圾回收器来管理。   
   *在堆中产生了一个数组或对象后，还可以 在栈中定义一个特殊的变量，让栈中这个变量的取值等于数组或对象在堆内存中的首地址，栈中的这个变量就成了数组或对象的引用变量。*   
   *引用变量是普通的变量，定义时在栈中分配，引用变量在程序运行到其作用域之外后被释放。而数组和对象本身在堆中分配，即使程序 运行到使用 new 产生数组或者对象的语句所在的代码块之外，数组和对象本身占据的内存不会被释放，数组和对象在没有引用变量指向它的时候，才变为垃圾，不能在被使用，但仍 然占据内存空间不放，在随后的一个不确定的时间被垃圾回收器收走（释放掉）。*

**二 基本数据类型**

这种类型是通过诸如 int a=7; 的形式来定义的，称为自动变量。这里自动变量是字面值。**不是类的实例，即不是类的引用**，这里并没有类的存在。a 是指向一个 int 类型的引用，指向 7 这个字面值。由于其大小确定生存期可知（这些定义在某个程序块中，程序块退出后，字段值就消失），因此存在**栈**中.   
由于栈的数据可以共享，因此 int a=3; int b=3; 这段代码，编译器首先处理 int a =3; ,先会在栈中创建一个变量为 a 的引用，然后查找有没有字面值为 3的地址，没有找到，就开辟一个存放 3 这个字面值的地址，然后将a 指向 3 的地址。接下来处理int b =3; 在创建完 b 这个引用变量后，由于栈中已经有 3 这个字面值，便将 b 指向 3 的地址。【定义变量，给变量赋值】

**三 包装类数据**

Java中的基本类型不是面向对象的，它们只是纯粹的数据，除了数值本身的信息之外，基本类型数据不带有其他信息或者可操作方法。这在实际使用中存在很多不足，为了解决这个不足，*\* 对每个基本类型都对应了一个引用的类型\**，称为装箱基本类型。 

**1，拆箱，装箱**

装箱：根据数据创建对应的包装对象。

Integer i = new Integer (3);

Integer j = 4;//jdk1.5 之后可以通过这种方式自动装箱

拆箱：将包装类型转换为基本数据类型。

int index2 = j.intValue();

int index1 = i;//自动拆箱

JDK1.5 为Integer 增加了一个全新的方法：public static Integer valueOf(int i) 在**自动装箱**过程时，编译器调用的是**static Integer valueOf(int i)这个方法** 于是Integer a=3; ==> Integer a=Integer.valueOf(3);。

此方法与new Integer(i)的不同处在于:   
方法一调用类方法返回一个表示 指定的 int 值的 Integer 实例。方法二产生一个新的Integer 对象。

**2，缓冲机制**

**JDK API文档中对这个新的valueOf方法有明确的解释：**   
如果不需要新的 Integer 实例，则通常应优先使用该方法，而不是构造方法 Integer(int)，因为该方法有可能通过**缓存经常请求的值**而显著提高空间和时间性能 .

查看Integer的valueOf方法的：

public static Integer valueOf(int i) {

assert IntegerCache.high >= 127;

//static final int low = -128;

//当-128=<i<=127的时候，就直接在缓存中取出 i de Integer 类型对象

if (i >= IntegerCache.low && i <= IntegerCache.high)

return IntegerCache.cache[i + (-IntegerCache.low)];

//否则就在堆内存中创建

return new Integer(i);

}

看出对于范围 [-128,127] 的整数，valueOf 方法做了特殊处理。采用IntegerCache.cache[i + (-IntegerCache.low)]; 这个方法。

查看 IntegerCache 类的实现为：

private static class IntegerCache {

static final int low = -128; //最小值是固定的

static final int high;

static final Integer cache[];//cache 缓存是一个存放Integer类型的数组

static { //初始化，最大值可以配置

// 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]; //初始化数组

int j = low;

//缓存区间数据

for(int k = 0; k < cache.length; k++)

//将-128~127包装成256个对象存入缓存

cache[k] = new Integer(j++);

}

private IntegerCache() {}

}

IntegerCache初始化后内存中就有Integer缓冲区cache[]了，-128~127区间的int值有其对应的的包装对象。**这就是 valueOf 方法真正的优化方法**，当-128=

public class ZhuangXaing {

public static void main(String[] args) {

Integer i= new Integer(12);

Integer j=12;

Integer k=Integer.valueOf(12);

Integer l= new Integer(232);

Integer m=232;

Integer n=232;

Double q = 232.0;

System.out.println("use ==.......");

System.out.println(i==12);

System.out.println(i==j);

System.out.println(j==k);

System.out.println(l==232);

System.out.println(l==m);

System.out.println(m==n);

System.out.println("use equals.....");

System.out.println(m.equals(n));

System.out.println(m.equals(q));

}

}

输出结果:

use ==.......

true

false

true

true

false

false

use equals.....

true

false

Integer i= new Integer(12); 是指明了在堆内存中创建对象；   
Integer j=12; 是自动装箱，调用valueOf 方法，返回return IntegerCache.cache[12 + 128]， 得到的是Integer 缓冲池中的对象。Integer k=Integer.valueOf(12); 与Integer j=12; 本质上相同，指向缓冲池中同一对象。**包装对象与数值比较，自动拆箱**。   
而对于大于127 的数值，执行的都是return new Integer(i) 都在堆内存中，但是地址不同。

对于equals 方法比较的是数值大小：

public boolean equals(Object obj) {

if (obj instanceof Integer) {

return value == ((Integer)obj).intValue();

}

return false;

}

可以看出比较的 obj 如果是 Integer 的实例，则比较拆箱后数值的是否相等。否则返回false。

2，下面这段代码输出结果是什么：

public class Main {

public static void main(String[] args) {

Double i1 = 100.0;

Double i2 = 100.0;

Double i3 = 200.0;

Double i4 = 200.0;

System.out.println(i1==i2);

System.out.println(i3==i4);

}

//false

//false

因为Double类的valueOf方法会采用与Integer类的valueOf方法不同的实现。很简单：在某个范围内的整型数值的个数是有限的，而浮点数却不是。

其他的包装器:   
Boolean： (全部缓存)   
Byte： (全部缓存)

Character ( <=127 缓存)   
Short (-128~127 缓存)   
Long (-128~127 缓存)

Float (没有缓存)   
Doulbe (没有缓存)

3，下面这段代码输出结果是什么：

public class Main {

public static void main(String[] args) {

Boolean i1 = false;

Boolean i2 = false;

Boolean i3 = true;

Boolean i4 = true;

System.out.println(i1==i2);

System.out.println(i3==i4);

}

}

先看Boolean 类的源码 ，valueOf 方法的实现：

public static Boolean valueOf(boolean b) {

return (b ? TRUE : FALSE);

}

而其中的 TRUE 和FALSE又是什么呢？在Boolean中定义了2个静态成员属性：

public static final Boolean TRUE = new Boolean(true);

/\*\*

\* The <code>Boolean</code> object corresponding to the primitive

\* value <code>false</code>.

\*/

public static final Boolean FALSE = new Boolean(false);

由此可知上面代码输出都为true .

、

java的各类型数据在内存中分配情况详解**\_**

|  |
| --- |
| 1.        有这样一种说法，如今争锋于IT战场的两大势力，MS一族偏重于底层实现，Java一族偏重于系统架构。说法根据无从考证，但从两大势力各自的社区力量和图书市场已有佳作不难看出，此说法不虚，但掌握Java的底层实现对Java程序员来说是至关重要的，本文介绍了Java中的数据在内存中的存储。      2 内存中的堆(stack)与栈(heap)   Java程序运行时有6个地方可以存储数据，它们分别是寄存器、栈、堆、静态存储、常量存储和非RAM存储，主要是堆与栈的存储。  【随机存储器 ：Random Access Memory 】    栈与堆都是Java用来在RAM中存放数据的地方。与C++不同，Java自动管理栈和堆，程序员不能直接地设置栈或堆。栈的优势是，存取速度比堆要快，仅次于直接位于CPU中的寄存器。另外，栈数据可以共享。但缺点是，存在栈中的数据大小与生存期必须是确定的，缺乏灵活性。堆的优势是可以动态地分配内存大小，生存期也不必事先告诉编译器，Java的垃圾收集器会自动收走这些不再使用的数据。但缺点是，由于要在运行时动态分配内存，存取速度较慢。   【 寄存器位于CPU中 】    3 Java中数据在内存中的存储   3.1基本数据类型的存储   Java的基本数据类型共有8种，即int, short, long, byte, float, double, boolean, char(注意，并没有string的基本类型)。这种类型的定义是通过诸如int a = 3； long b = 255L；的形式来定义的，称为自动变量。值得注意的是：自动变量存的是字面值，不是类的实例，即不是类的引用，这里并没有类的存在。如int a = 3；这里的a是一个指向int类型的引用，指向3这个字面值。这些字面值的数据，由于大小可知，生存期可知(这些字面值固定定义在某个程序块里面，程序块退出后，字段值就消失了)，出于追求速度的原因，就存在于栈中。   另外，栈有一个很重要的特殊性，就是存在栈中的数据可以共享。     假设我们同时定义：    int a = 3；     int b=3；    编译器先处理int a = 3；首先它会在栈中创建一个变量为a的引用，然后查找有没有字面值为3的地址，没找到，就开辟一个存放3这个字面值的地址，然后将a指向3的地址。接着处理int b = 3；在创建完b这个引用变量后，由于在栈中已经有3这个字面值，便将b直接指向3的地址。这样，就出现了a与b同时均指向3的情况。   【上文提到了"引用+数值+内存地址"这三个名词,其中变量名就是引用，给变量赋的值就是数值，       而所提到的内存是抽象的内容，让引用指向的不是数值，而是存取数值的那块内存地址】     定义完a与b的值后，再令a = 4；那么，b不会等于4，还是等于3。在编译器内部，遇到时，它就会重新搜索栈中是否有4的字面值，如果没有，重新开辟地址存放4的值；如果已经有了，则直接将a指向这个地址。因此a值的改变不会影响到b的值。     【定义变量，给变量赋值，然后在编译的过程中就可以将其保存在内存中了】    3.2对象的内存模型   在Java中，创建一个对象包括对象的声明和实例化两步，下面用一个例题来说明对象的内存模型。    假设有类Rectangle定义如下：     【**Rectangle：矩形**】    class Rectangle{    double width,height；    Rectangle(double w,double h){    width=w；height=h； }}   (1)声明对象时的内存模型   用Rectangle rect；声明一个对象rect时，将在栈内存为对象的引用变量rect分配内存空间，但Rectangle的值为空，称rect是一个空对象。空对象不能使用，因为它还没有引用任何“实体”。   (2)对象实例化时的内存模型   当执行rect=new Rectangle(3,5)；时，会做两件事：   在堆内存中为类的成员变量width,height分配内存，并将其初始化为各数据类型的默认值；接着进行显式初始化（类定义时的初始化值）；最后调用构造方法，为成员变量赋值。    返回堆内存中对象的引用（相当于首地址）给引用变量rect,以后就可以通过rect来引用堆内存中的对象了。    (3)创建多个不同的对象实例    一个类通过使用new运算符可以创建多个不同的对象实例，这些对象实例将在堆中被分配不同的内存空间，改变其中一个对象的状态不会影响其他对象的状态。例如：    Rectangle r1=new Rectangle(3,5)；    Rectangle r2=new Rectangle(4,6)；    此时，将在堆内存中分别为两个对象的成员变量width、height分配内存空间，两个对象在堆内存中占据的空间是互不相同的。如果有    Rectangle r1=new Rectangle(3,5)；    Rectangle r2=r1；    则在堆内存中只创建了一个对象实例，在栈内存中创建了两个对象引用，两个对象引用同时指向一个对象实例。   3.3包装类数据的存储   基本型别都有对应的包装类：如int对应Integer类，double对应Double类等，基本类型的定义都是直接在栈中，如果用包装类来创建对象，就和普通对象一样了。例如：int i=0；i直接存储在栈中。  Integer i（i此时是对象） = new Integer(5)；这样，i对象数据存储在堆中，i的引用存储在栈中，通过栈中的引用来操作对象。    【数据存储在堆中，引用存储在栈中】    3.4 String 类型数据的存储   String是一个特殊的包装类数据。可以用      String str = new String("abc")；的形式来创建； 也可以用    String str = "abc"；的形式来创建。  第一种创建方式，和普通对象的的创建过程一样； 第二种创建方式，Java内部将此语句转化为以下几个步骤：    (1)先定义一个名为str的对String类的对象引用变量：String str；    (2)在栈中查找有没有存放值为“abc”的地址，如果没有，则开辟一个存放字面值为“abc”的地址，接着创建一个新的String类的对象o，并将o的字符串值指向这个地址，而且在栈中这个地址旁边记下这个引用的对象o。如果已经有了值为“abc”的地址，则查找对象o，并返回o的地址。    (3)将str指向对象o的地址。    值得注意的是，一般String类中字符串值都是直接存值的。但像String str = "abc"；这种场合下，其字符串值却是保存了一个指向存在栈中数据的引用。    为了更好地说明这个问题，我们可以通过以下的几个代码进行验证。    String str1=“abc”；    String str2=“abc”；    System.out.println(s1==s2)；//true    注意，这里并不用str1.equals(str2)；的方式，因为这将比较两个字符串的值是否相等。==号，根据JDK的说明，只有在两个引用都指向了同一个对象时才返回真值。而我们在这里要看的是，str1与str2是否都指向了同一个对象。    我们再接着看以下的代码。    Stringstr1=new String(“abc”)；    Stringstr2=“abc”；    System.out.println(str1==str2)；//false    创建了两个引用。创建了两个对象。两个引用分别指向不同的两个对象。    以上两段代码说明，只要是用new()来新建对象的，都会在堆中创建，而且其字符串是单独存值的，即使与栈中的数据相同，也不会与栈中的数据共享。   **3.5数组的内存分配**    当定义一个数组，int x[]；或int []x；时，在栈内存中创建一个数组引用，通过该引用（即数组名）来引用数组。x=new int[3]；将在堆内存中分配3个保存int型数据的空间，堆内存的首地址放到栈内存中，每个数组元素被初始化为0。      4 内存空间的释放   栈上变量的生存时间受限于当前函数的生存时间，函数退出了，变量就不存在了。在堆中分配的对象实例，当不再有任何一个引用变量指向它时，这个对象就可以被垃圾回收机制回收了。      5 总结堆栈  再来看Java的内存，栈内存用来存放一些基本类型的变量和数组及对象的引用变量，而堆内存主要是来放置对象实例的。明白这个就能很好的解释多态、继承、覆盖方面的问题了 |