Java堆、栈和常量池详解

博客分类： Java综合

一：在JAVA中，有六个不同的地方可以存储数据：

1. 寄存器（register）。 这是最快的存储区，因为它位于不同于其他存储区的地方——处理器内部。但是寄存器的数量极其有限，所以寄存器由编译器根据需求进行分配。你不能直接控制，也不能在程序中感觉到寄存器存在的任何迹象。

------最快的存储区, 由编译器根据需求进行分配,我们在程序中无法控制.

2. 堆栈（stack）。位于通用RAM中，但通过它的“堆栈指针”可以从处理器哪里获得支持。堆栈指针若向下移动，则分配新的内存；若向上移动，则释放那些 内存。这是一种快速有效的分配存储方法，仅次于寄存器。创建程序时候，JAVA编译器必须知道存储在堆栈内所有数据的确切大小和生命周期，因为它必须生成 相应的代码，以便上下移动堆栈指针。这一约束限制了程序的灵活性，所以虽然某些JAVA数据存储在堆栈中——特别是对象引用，但是JAVA对象不存储其 中。

------存放基本类型的变量数据和对象，数组的引用，但对象本身不存放在栈中，而是存放在堆（new 出来的对象）或者常量池中（字符串常量对象存放在常量池中）

3. 堆（heap）。一种通用性的内存池（也存在于RAM中），用于存放所以的JAVA对象。堆不同于堆栈的好处是：编译器不需要知道要从堆里分配多少存储区 域，也不必知道存储的数据在堆里存活多长时间。因此，在堆里分配存储有很大的灵活性。当你需要创建一个对象的时候，只需要new写一行简单的代码，当执行 这行代码时，会自动在堆里进行存储分配。当然，为这种灵活性必须要付出相应的代码。用堆进行存储分配比用堆栈进行存储存储需要更多的时间。

------存放所有new出来的对象。

4. 静态存储（static storage）。这里的“静态”是指“在固定的位置”。静态存储里存放程序运行时一直存在的数据。你可用关键字static来标识一个对象的特定元素是静态的，但JAVA对象本身从来不会存放在静态存储空间里。

------存放静态成员（static定义的）

5. 常量存储（constant storage）。常量值通常直接存放在程序代码内部，这样做是安全的，因为它们永远不会被改变。有时，在嵌入式系统中，常量本身会和其他部分分割离开，所以在这种情况下，可以选择将其放在ROM中

------存放字符串常量和基本类型常量（public static final）

6. 非RAM存储。如果数据完全存活于程序之外，那么它可以不受程序的任何控制，在程序没有运行时也可以存在。

------硬盘等永久存储空间

就速度来说，有如下关系：

寄存器 < 堆栈 < 堆 < 其他

这里我们主要关心栈，堆和常量池，对于栈和常量池中的对象可以共享，对于堆中的对象不可以共享。栈中的数据大小和生命周期是可以确定的，当没有引用指向数据时，这个数据就会消失。堆中的对象的由垃圾回收器负责回收，因此大小和生命周期不需要确定，具有很大的灵活性。

对于字符串：其对象的引用都是存储在栈中的，如果是编译期已经创建好(直接用双引号定义的)的就存储在常量池中，如果是运行期（new出来的）才能确定的就存储在堆中。对于equals相等的字符串，在常量池中永远只有一份，在堆中有多份。

如以下代码：

Java代码

String s1 = "china";

String s2 = "china";

String s3 = "china";

String ss1 = new String("china");

String ss2 = new String("china");

String ss3 = new String("china");

这里解释一下黄色这 3 个箭头，对于通过 new 产生一个字符串（假设为 ”china” ）时，会先去常量池中查找是否已经有了 ”china” 对象，如果没有则在常量池中创建一个此字符串对象，然后堆中再创建一个常量池中此 ”china” 对象的拷贝对象。这也就是有道面试题： String s = new String(“xyz”); 产生几个对象？一个或两个，如果常量池中原来没有 ”xyz”, 就是两个。

对于基础类型的变量和常量：变量和引用存储在栈中，常量存储在常量池中。

如以下代码：

Java代码

int i1 = 9;

int i2 = 9;

int i3 = 9;

public static final int INT1 = 9;

public static final int INT2 = 9;

public static final int INT3 = 9;

对于成员变量和局部变量：成员变量就是方法外部，类的内部定义的变量；局部变量就是方法或语句块内部定义的变量。局部变量必须初始化。

形式参数是局部变量，局部变量的数据存在于栈内存中。栈内存中的局部变量随着方法的消失而消失。

成员变量存储在堆中的对象里面，由垃圾回收器负责回收。

如以下代码：

Java代码

class BirthDate {

private int day;

private int month;

private int year;

public BirthDate(int d, int m, int y) {

day = d;

month = m;

year = y;

}

省略get,set方法………

}

public class Test{

public static void main(String args[]){

int date = 9;

Test test = new Test();

test.change(date);

BirthDate d1= new BirthDate(7,7,1970);

}

public void change1(int i){

i = 1234;

}

对于以上这段代码，date为局部变量，i,d,m,y都是形参为局部变量，day，month，year为成员变量。下面分析一下代码执行时候的变化：

1. main方法开始执行：int date = 9;

date局部变量，基础类型，引用和值都存在栈中。

2. Test test = new Test();

test为对象引用，存在栈中，对象(new Test())存在堆中。

3. test.change(date);

i为局部变量，引用和值存在栈中。当方法change执行完成后，i就会从栈中消失。

4. BirthDate d1= new BirthDate(7,7,1970);

d1 为对象引用，存在栈中，对象(new BirthDate())存在堆中，其中d，m，y为局部变量存储在栈中，且它们的类型为基础类型，因此它们的数据也存储在栈中。 day,month,year为成员变量，它们存储在堆中(new BirthDate()里面)。当BirthDate构造方法执行完之后，d,m,y将从栈中消失。

5.main方法执行完之后，date变量，test，d1引用将从栈中消失，new Test(),new BirthDate()将等待垃圾回收。

Java堆、栈和常量池详解（二）

博客分类： Java综合

1. 栈(stack)与堆(heap)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。

2. 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。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。另外，栈数据可以共 享，详见第3点。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。

3. Java中的数据类型有两种。

一种是基本类型(primitive types), 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的引用变量后，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。

特 别注意的是，这种字面值的引用与类对象的引用不同。假定两个类对象的引用同时指向一个对象，如果一个对象引用变量修改了这个对象的内部状态，那么另一个对 象引用变量也即刻反映出这个变化。相反，通过字面值的引用来修改其值，不会导致另一个指向此字面值的引用的值也跟着改变的情况。如上例，我们定义完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a=4；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。

另一种是包装类数据，如Integer, String, Double等将相应的基本数据类型包装起来的类。这些类数据全部存在于堆中，Java用new()语句来显示地告诉编译器，在运行时才根据需要动态创建，因此比较灵活，但缺点是要占用更多的时间。

举例如下：

Java代码

public class Test {

public static void main(String[] args)

{ int a1=1;

int b1=1;

int c1=2;

int d1=a1+b1;

Integer a = 1;

Integer b = 2;

Integer c = 3;

Integer d = 3;

Integer e = 321;

Integer f = 321;

Long g = 3L;

System.out.println(a1==b1); //true 结果1

System.out.println(c1==d1); //true 结果2

System.out.println(c==d); //true 结果3

System.out.println(e==f); //false 结果4

}

}

分析：结果1：a1==b1如上面所述,会在栈 中开辟存储空间存放数据。

结果2：首先它会在栈 中创建一个变量为c1的引用，然后查找有没有字面值为2的地址，没找到，就开辟一个存放2这个字面值的地址，然后将c1指向2的地址,d1为两个字面值相加也为2， 由于在栈中已经有2这个字面值，便将d1直接指向2的地址。这样，就出现了c1与d1同时均指向3的情况。

在分析下面结果以前让我们先对Java自动拆箱和装箱做个了结：在自动装箱时，把int变成Integer的时候，是有规则的，当你的int的值在-128-IntegerCache.high(127) 时，返回的不是一个新new出来的Integer对象，而是一个已经缓存在堆 中的Integer对象，（我们可以这样理解，系统已经把-128到127之 间的Integer缓存到一个Integer数组中去了，如果你要把一个int变成一个Integer对象，首先去缓存中找，找到的话直接返回引用给你就 行了，不必再新new一个），如果不在-128-IntegerCache.high(127) 时会返回一个新new出来的Integer对象。

结果3：由于3是在范围内所以是从缓存中取数据的，c和d指向同一个对象，结果为true;

结果4：由于321不是在范围内所以不是从缓存中取数据的而是单独有new对象，e和f并没有指向同一个对象，结果为false;

4. String是一个特殊的包装类数据。即可以用String str = new String("abc");的形式来创建，也可以用String str = "abc"；的形式来创建(作为对比，在JDK 5.0之前，你从未见过Integer i = 3;的表达式，因为类与字面值是不能通用的，除了String。而在JDK 5.0中，这种表达式是可以的！因为编译器在后台进行Integer i = new Integer(3)的转换)。前者是规范的类的创建过程，即在Java中，一切都是对象，而对象是类的实例，全部通过new()的形式来创建。Java 中的有些类，如DateFormat类，可以通过该类的getInstance()方法来返回一个新创建的类，似乎违反了此原则。其实不然。该类运用了单 例模式来返回类的实例，只不过这个实例是在该类内部通过new()来创建的，而getInstance()向外部隐藏了此细节。那为什么在String str = "abc"；中，并没有通过new()来创建实例，是不是违反了上述原则？其实没有。

4(1)String str = "abc"创建对象的过程

1 首先在常量池中查找是否存在内容为"abc"字符串对象

2 如果不存在则在常量池中创建"abc"，并让str引用该对象

3 如果存在则直接让str引用该对象

至 于"abc"是怎么保存，保存在哪？常量池属于类信息的一部分，而类信息反映到JVM内存模型中是对应存在于JVM内存模型的方法区，也就是说这个类信息 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"abc"可以说存在于堆中（而有些资料，为了把方法区的 堆区别于JVM的堆，把方法区称为栈）。一般这种情况下，"abc"在编译时就被写入字节码中，所以class被加载时，JVM就为"abc"在常量池中 分配内存，所以和静态区差不多。

4(2)String str = new String("abc")创建实例的过程

1 首先在堆中（不是常量池）创建一个指定的对象"abc"，并让str引用指向该对象

2 在字符串常量池中查看，是否存在内容为"abc"字符串对象

3 若存在，则将new出来的字符串对象与字符串常量池中的对象联系起来

4 若不存在，则在字符串常量池中创建一个内容为"abc"的字符串对象，并将堆中的对象与之联系起来

intern 方法可以返回该字符串在常量池中的对象的引用，可以通过下面代码简单的测试

Java代码

class StringTest {

public static void main(String[] args) {

String str1 = "abc";

String str2 = new String("abc").intern();

System.out.println(str1==str2);

}

}

一个初始为空的字符串池，它由类 String 私有地维护。

当调用 intern 方法时，如果池已经包含一个等于此 String 对象的字符串（用 equals(Object) 方法确定），则返回池中的字符串。否则，将此 String 对象添加到池中，并返回此 String 对象的引用。

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 。

所以String str1 = "abc"，str1引用的是常量池（方法区）的对象，而String str2 = new String("abc")，str2引用的是堆中的对象，所以内存地址不一样，但是内容一样，所以==为false，而equals是true

4(3)String str1 = "abc"; String str2 = "ab" + "c"; str1==str2是ture

是因为String str2 = "ab" + "c"会查找常量池中时候存在内容为"abc"字符串对象，如存在则直接让str2引用该对象，显然String str1 = "abc"的时候，上面说了，会在常量池中创建"abc"对象，所以str1引用该对象，str2也引用该对象，所以str1==str2

4(4)String str1 = "abc"; String str2 = "ab"; String str3 = str2 + "c"; str1==str3是false

是因为String str3 = str2 + "c"涉及到变量（不全是常量）的相加，所以会生成新的对象，其内部实现是先new一个StringBuilder，然后 append(str2),append("c");然后让str3引用toString()返回的对象

如果想了解更多的细节，可以自己查看反编译的代码，查看反编译代码可以用javap，即

javap -c -verbose 要查看的类文件(.class不要)

比如上面的代码的示例

javac StringTest.java //编译

javap -c -verbose StringTest //反编译

4(5)String str1 = "abc"; String str2 = "abc"; System.out.println(str1==str2); //true

注意，我们这里并不用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。

结果说明，JVM创建了两个引用str1和str2，但只创建了一个对象，而且两个引用都指向了这个对象。

4(6)String str1 = "abc";String str2 = "abc";str1 = "bcd";System.out.println(str1 + "," + str2); //bcd, abc

System.out.println(str1==str2); //false

这就是说，赋值的变化导致了类对象引用的变化，str1指向了另外一个新对象！而str2仍旧指向原来的对象。上例中，当我们将str1的值改为"bcd"时，JVM发现在 常量池中没有存放该值的地址，便开辟了这个地址，并创建了一个新的对象，其字符串的值指向这个地址。

事 实上，String类被设计成为不可改变(immutable)的类。如果你要改变其值，可以，但JVM在运行时根据新值悄悄创建了一个新对象，然后将这 个对象的地址返回给原来类的引用。这个创建过程虽说是完全自动进行的，但它毕竟占用了更多的时间。在对时间要求比较敏感的环境中，会带有一定的不良影响。

4(7)String str1 = "abc"; String str2 = "abc";str1 = "bcd";

String str3 = str1;

System.out.println(str3); //bcd

String str4 = "bcd";

System.out.println(str1 == str4); //true

str3 这个对象的引用直接指向str1所指向的对象(注意，str3并没有创建新对象)。当str1改完其值后，再创建一个String的引用str4，并指向 因str1修改值而创建的新的对象。可以发现，这回str4也没有创建新的对象，从而再次实现栈中数据的共享。

4(8)我们再接着看以下的代码。

String str1 = new String("abc");

String str2 = "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。

String str1 = "abc";

String str2 = new String("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。

5. 数据类型包装类的值不可修改。不仅仅是String类的值不可修改，所有的数据类型包装类都不能更改其内部的值。

6. 结论与建议：

(1) 我们在使用诸如String str = "abc"；的格式定义类时，总是想当然地认为，我们创建了String类的对象str。担心陷阱！对象可能并没有被创建！唯一可以肯定的是，指向 String类的引用被创建了。至于这个引用到底是否指向了一个新的对象，必须根据上下文来考虑，除非你通过new()方法来显要地创建一个新的对象。因 此，更为准确的说法是，我们创建了一个指向String类的对象的引用变量str，这个对象引用变量指向了某个值为"abc"的String类。清醒地认 识到这一点对排除程序中难以发现的bug是很有帮助的。

(2)使用String str = "abc"；的方式，可以在一定程度上提高程序的运行速度，因为JVM会自动根据栈中数据的实际情况来决定是否有必要创建新对象。而对于String str = new String("abc")；的代码，则一概在堆中创建新对象，而不管其字符串值是否相等，是否有必要创建新对象，从而加重了程序的负担。这个思想应该是 享元模式的思想，但JDK的内部在这里实现是否应用了这个模式，不得而知。

(3)当比较包装类里面的数值是否相等时，用equals()方法；当测试两个包装类的引用是否指向同一个对象时，用==。

(4)由于String类的immutable性质，当String变量需要经常变换其值时，应该考虑使用StringBuffer类，以提高程序效率