

以下内容节选自 Java 私塾自编经典教材：

Java 的内存分配一直是初学 Java 的一个重难点，下面就一起来分析一下。

七：再谈 Java 内存分配

Java 程序运行时的内存结构分成：方法区、栈内存、堆内存、本地方法栈几种。

栈和堆都是数据结构的知识，如果不清楚，没有关系，就当成一个不同的名字就好了，下面的讲解不需要用到它们具体的知识。

1：方法区

方法区存放装载的类数据信息包括：

(1)：基本信息：

- 1) 每个类的全限定名
- 2) 每个类的直接超类的全限定名(可约束类型转换)
- 3) 该类是类还是接口
- 4) 该类型的访问修饰符
- 5) 直接超接口的全限定名的有序列表

(2)：每个已装载类的详细信息：

1) 运行时常量池：

存放该类型所用的一切常量(直接常量和对其它类型、字段、方法的符号引用)，它们以数组形式通过索引被访问，是外部调用与类联系及类型对象化的桥梁。它是类文件(字节码)常量池的运行时表示。(还有一种静态常量池，在字节码文件中)。

2) 字段信息：

类中声明的每一个字段的信息(名，类型，修饰符)。

3) 方法信息：

类中声明的每一个方法的信息(名，返回类型，参数类型，修饰符，方法的字节码和异常表)。

4) 静态变量

5) 到类 classloader 的引用：即到该类的类装载器的引用。

6) 到类 class 的引用：

虚拟机为每一个被装载的类型创建一个 class 实例，用来代表这个被装载的类。

2：栈内存

Java 栈内存以帧的形式存放本地方法的调用状态(包括方法调用的参数，局部变量，中间结果等)。每调用一个方法就将对应该方法的方法帧压入 Java 栈，成为当前方法帧。当调用结束(返回)时，就弹出该帧。

编译器将源代码编译成字节码(.class)时,就已经将各种类型的方法的局部变量,操作数栈大小确定并放在字节码中,随着类一并装载入方法区。当调用方法时,通过访问方法区中的类的信息,得到局部变量以及操作数栈的大小。

也就是说:在方法中定义的一些基本类型的变量和对象的引用变量都在方法的栈内存中分配。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,当超过变量的作用域后,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它用。

栈内存的构成:

Java 栈内存由局部变量区、操作数栈、帧数据区组成。

(1): 局部变量区为一个以字为单位的数组,每个数组元素对应一个局部变量的值。调用方法时,将方法的局部变量组成一个数组,通过索引来访问。若为非静态方法,则加入一个隐含的引用参数 this,该参数指向调用这个方法的对象。而静态方法则没有 this 参数。因此,对象无法调用静态方法。

(2): 操作数栈也是一个数组,但是通过栈操作来访问。所谓操作数是那些被指令操作的数据。当需要对参数操作时如 $a=b+c$,就将即将被操作的参数压栈,如将 b 和 c 压栈,然后由操作指令将它们弹出,并执行操作。虚拟机将操作数栈作为工作区。

(3): 帧数据区处理常量池解析,异常处理等

3: 堆内存

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。引用变量就相当于为数组或对象起的一个名称,以后就可以在程序中使用栈中的引用变量来访问堆中的数组或对象。

栈内存和堆内存比较

栈与堆都是 Java 用来在内存中存放数据的地方。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。

Java 的堆是一个运行时数据区,对象从中分配空间。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,因为它是在运行时动态分配内存的,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。

栈的优势是,存取速度比堆要快,仅次于寄存器,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。栈中主要存放一些基本类型的变量(int, short, long, byte, float, double, boolean, char)和对象句柄。

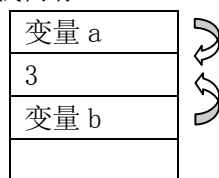
栈有一个很重要的特殊性,就是存在栈中的数据可以共享。假设我们同时定义:

```
int a = 3;
```

```
int b = 3;
```

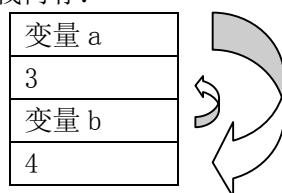
编译器先处理 `int a = 3;` 首先它会在栈中创建一个变量为 a 的引用,然后查找栈中是否有 3 这个值,如果没找到,就将 3 存放进来,然后将 a 指向 3。接着处理 `int b = 3;` 在创建完 b 的引用变量后,因为在栈中已经有 3 这个值,便将 b 直接指向 3。这样,就出现了 a 与 b 同时均指向 3 的情况。内存示意图如下:

栈内存：



这时，如果再令 $a=4$ ；那么编译器 会重新搜索栈中是否有 4 值，如果没有，则将 4 存放进来，并令 a 指向 4；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。要注意这种数据的共享与两个对象的引用同时指向一个对象的这种共享是不同的，因为这种情况 a 的修改并不会影响到 b ，它是由编译器完成的，它有利于节省空间。此时的内存分配示意图如下：

栈内存：



而一个对象引用变量修改了这个对象的内部状态，会影响到另一个对象引用变量。

4: 本地方法栈内存

与调用的本地方法的语言相关，如调用的是一个 c 语言方法则为一个 c 栈。本地方法可以回调 java 方法。若有 java 方法调用本地方法，虚拟机就运行这个本地方法。

在虚拟机看来运行这个本地方法就是执行这个 java 方法，如果本地方法抛出异常，虚拟机就认为是这个 java 方法抛出异常。

Java 通过 Java 本地接口 JNI (Java Native Interface) 来调用其它语言编写的程序，在 Java 里面用 native 修饰符来描述一个方法是本地方法。这个了解一下就好了，在我们的课程中不会涉及到。

5: String 的内存分配

String 是一个特殊的包装类数据。可以用：

```
String str = new String("abc");  
String str = "abc";
```

两种的形式来创建，第一种是用 new() 来新建对象的，它会在存放于堆中。每调用一次就会创建一个新的对象。

而第二种是先在栈中创建一个对 String 类的对象引用变量 str，然后查找栈中有没有存放 "abc"，如果没有，则将 "abc" 存放进栈，并令 str 指向 "abc"，如果已经有 "abc" 则直接令 str 指向 "abc"。

比较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 ==，下面用例子说明上面的理论。

```
String str1 = "abc";  
String str2 = "abc";  
System.out.println(str1==str2); //true  
可以看出 str1 和 str2 是指向同一个对象的。
```

```
String str1 = new String ("abc");
String str2 = new String ("abc");
System.out.println(str1==str2); // false
用 new 的方式是生成不同的对象。每一次生成一个。
```

因此用第一种方式创建多个"abc"字符串，在内存中其实只存在一个对象而已。这种写法有利于节省内存空间。同时它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。

另一方面，要注意：我们在使用诸如 String str = "abc"; 的格式时，总是想当然地认为，创建了 String 类的对象 str。担心陷阱！对象可能并没有被创建！而可能只是指向一个先前已经创建的对象。只有通过 new() 方法才能保证每次都创建一个新的对象。

由于 String 类的值不可变性 (immutable)，当 String 变量需要经常变换其值时，应该考虑使用 StringBuffer 或 StringBuilder 类，以提高程序效率。

6: static 属性的内存分配

一个类中，一个 static 变量只会有一个内存空间，虽然有多个类实例，但这些类实例中的这个 static 变量会共享同一个内存空间。示例如下：

```
public class Test{
    static UserModel um = new UserModel();
    public static void main(String[] args) {
        Test t1 = new Test();
        t1.um.userName = "张三";
        Test t2 = new Test();
        t2.um.userName = "李四";

        System.out.println("t1.um.userName==" + t1.um.userName);
        System.out.println("t2.um.userName==" + t2.um.userName);
    }
}

class UserModel{
    public String userName = "";
}
```

运行结果：

```
t1.um.userName==李四
t2.um.userName==李四
```

为什么会是一样的值呢？就是因为多个实例中的静态变量 um 是共享同一内存空间，t1.um 和 t2.um 其实指向的都是同一个内存空间，所以就得到上面的结果了。

要想看看是不是 static 导致这样的结果，你可以尝试去掉 UserModel 前面的 static，然后

再试一试，看看结果，应该如下：

```
t1.um.userName==张三  
t2.um.userName==李四
```

还有一点也很重要：**static** 的变量是在类装载的时候就会被初始化。也就是说，只要类被装载，不管你是否使用了这个 **static** 变量，它都会被初始化。

小结一下：类变量（**class variables**）用关键字 **static** 修饰，在类加载的时候，分配类变量的内存，以后在生成类的实例对象时，将共享这块内存（类变量），任何一个对象对类变量的修改，都会影响其它对象。外部有两种访问方式：通过对象来访问或通过类名来访问。