Java 中堆与栈的区别

简单的说：

Java 把内存划分成两种：一种是栈内存，一种是堆内存。

在函数中定义的一些基本类型的变量和对象的引用变量都在函数的栈内存中分配。

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他用。

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。

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()语句来显式地告诉编译器，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。

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。

5. 关于 String str = "abc"的内部工作。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(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。

我们再来更进一步，将以上代码改成：

```
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 发现在栈中没有存

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。

再修改原来代码：

```
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。

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。

6. 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。

7. 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。

java 中内存分配策略及堆和栈的比较

内存分配策略

按照编译原理的观点,程序运行时的内存分配有三种策略,分别是静态的,栈式的,和堆式的.

静态存储分配是指在编译时就能确定每个数据目标在运行时刻的存储空间需求,因而在

编译时就可以给他们分配固定的内存空间.这种分配策略要求程序代码中不允许有可变数据结构(比如可变数组)的存在,也不允许有嵌套或者递归的结构出现,因为它们都会导致编译程序无法计算准确的存储空间需求.

栈式存储分配也可称为动态存储分配,是由一个类似于堆栈的运行栈来实现的.和静态存储分配相反,在栈式存储方案中,程序对数据区的需求在编译时是完全未知的,只有到运行的时候才能够知道,但是规定在运行中进入一个程序模块时,必须知道该程序模块所需的数据区大小才能够为其分配内存.和我们在数据结构所熟知的栈一样,栈式存储分配按照先进后出的原则进行分配。

静态存储分配要求在编译时能知道所有变量的存储要求,栈式存储分配要求在过程的入口处必须知道所有的存储要求,而堆式存储分配则专门负责在编译时或运行时模块入口处都无法确定存储要求的数据结构的内存分配,比如可变长度串和对象实例.堆由大片的可利用块或空闲块组成,堆中的内存可以按照任意顺序分配和释放.

堆和栈的比较

上面的定义从编译原理的教材中总结而来,除静态存储分配之外,都显得很呆板和难以理解,下面撇开静态存储分配,集中比较堆和栈:

从堆和栈的功能和作用来通俗的比较,堆主要用来存放对象的，栈主要是用来执行程序的.而这种不同又主要是由于堆和栈的特点决定的:

在编程中，例如C/C++中，所有的方法调用都是通过栈来进行的,所有的局部变量,形式参数都是从栈中分配内存空间的。实际上也不是什么分配,只是从栈顶向上用就行,就好像工厂中的传送带(conveyor belt)一样,Stack Pointer 会自动指引你到放东西的位置,你所要做的只是把东西放下来就行.退出函数的时候,修改栈指针就可以把栈中的内容销毁.这样的模式速度最快,当然要用来运行程序了.需要注意的是,在分配的时候,比如为一个即将要调用的程序模块分配数据区时,应事先知道这个数据区的大小,也就说是**虽然分配是在程序运行时进行的,但是分配的大小多少是确定的,不变的,而这个"大小多少"是在编译时确定的,不是在运行时.**

堆是应用程序在运行的时候请求操作系统分配给自己内存，由于从操作系统管理的内存分配,所以在分配和销毁时都要占用时间，因此用堆的效率非常低.但是堆的优点在于,编译器不必知道要从堆里分配多少存储空间,也不必知道存储的数据要在堆里停留多长的时间,

因此,用堆保存数据时会得到更大的灵活性。事实上,面向对象的多态性,堆内存分配是必不可少的,因为多态变量所需的存储空间只有在运行时创建了对象之后才能确定.在 C++中，要求创建一个对象时，只需用 new 命令编制相关的代码即可。执行这些代码时，会在堆里自动进行数据的保存.当然，为达到这种灵活性，必然会付出一定的代价:在堆里分配存储空间时会花掉更长的时间！这也正是导致我们刚才所说的效率低的原因,看来列宁同志说的好,人的优点往往也是人的缺点,人的缺点往往也是人的优点(晕~).

JVM 中的堆和栈

JVM 是基于堆栈的虚拟机.JVM 为每个新创建的线程都分配一个堆栈.也就是说,对于一个 Java 程序来说，它的运行就是通过对堆栈的操作来完成的。堆栈以帧为单位保存线程的状态。JVM 对堆栈只进行两种操作:以帧为单位的压栈和出栈操作。

我们知道,某个线程正在执行的方法称为此线程的当前方法.我们可能不知道,当前方法使用的帧称为当前帧。当线程激活一个 Java 方法,JVM 就会在线程的 Java 堆栈里新压入一个帧。这个帧自然成为了当前帧.在此方法执行期间,这个帧将用来保存参数,局部变量,中间计算过程和其他数据.这个帧在这里和编译原理中的活动纪录的概念是差不多的.

从 Java 的这种分配机制来看,堆栈又可以这样理解:堆栈(Stack)是操作系统在建立某个进程时或者线程(在支持多线程的操作系统中是线程)为这个线程建立的存储区域,该区域具有先进后出的特性。

每一个 Java 应用都唯一对应一个 JVM 实例，每一个实例唯一对应一个堆。应用程序在运行中所创建的所有类实例或数组都放在这个堆中,并由应用所有的线程 共享.跟 C/C++不同，Java 中分配堆内存是自动初始化的。**Java 中所有对象的存储空间都是在堆中分配的，但是这个对象的引用却是在堆栈中分配,也就是说在建立一个对象时从两个地方都分配内存，在堆中分配的内存实际建立这个对象，而在堆栈中分配的内存只是一个指向这个堆对象的指针(引用)而已。**

GC 的思考

Java 为什么慢?JVM 的存在当然是一个原因,但有人说,在 Java 中,除了简单类型(int,char 等)的数据结构,其它都是在堆中分配内存(所以说 Java 的一切都是对象)，这也

是程序慢的原因之一。

我的想法是(应该说代表 TIJ 的观点),如果没有 Garbage Collector(GC),上面的说法就是成立的.堆不象栈是连续的空间,没有办法指望堆本身的内存分配能够象堆栈一样拥有传送带般的速度,因为,谁会 为你整理庞大的堆空间,让你几乎没有延迟的从堆中获取新的空间呢?

这个时候,GC 站出来解决问题.我们都知道 GC 用来清除内存垃圾,为堆腾出空间供程序使用,但GC同时也担负了另外一个重要的任务,就是要让Java中堆的内存分配和其他语言中堆栈的内存分配一样快,因为速度的问题几乎是众口一词的对 Java 的诟病.要达到这样的目的,就必须使堆的分配也能够做到象传送带一样,不用自己操心去找空闲空间.这样,GC 除了负责清除Garbage外,还要负责整理堆中的对象,把它们转移到一个远离Garbage的纯净空间中无间隔的排列起来,就象堆栈中一样紧凑,这样 Heap Pointer 就可以方便的指向传送带的起始位置,或者说一个未使用的空间,为下一个需要分配内存的对象"指引方向".因此可以这样说,垃圾收集影响了对象的创建速度,听起来很怪,对不对?

那 GC 怎样在堆中找到所有存活的对象呢?前面说了,在建立一个对象时， 在堆中分配实际建立这个对象的内存,而在堆栈中分配一个指向这个堆对象的指针(引 用),那么只要在堆栈(也有可能在静态存储区)找到这个引用,就可以跟踪到所有存活的对象.找到之后,GC 将它们从一个堆的块中移到另外一个堆的块中,并 将它们一个挨一个的排列起来,就象我们上面说的那样,模拟出了一个栈的结构,但又不是先进后出的分配,而是可以任意分配的,在速度可以保证的情况下, Isn't it great?

但是， 列宁同志说了,人的优点往往也是人的缺点,人的缺点往往也是人的优点(再晕~~).GC()的运行要占用一个线程,这本身就是一个降低程序运行性能 的缺陷,更何况这个线程还要在堆中把内存翻来覆去的折腾.不仅如此,如上面所说,堆中存活的对象被搬移了位置,那么所有对这些对象的引用都要重新赋值.这 些开销都会导致性能的降低.

基础数据类型直接在栈空间分配，方法的形式参数，直接在栈空间分配，当方法调用完成后从栈空间回收。引用数据类型，需要用 new 来创建，既在栈空间 分配一个地址空间，又在堆空间分配对象的类变量 。方法的引用参数，在栈空间分配一个地址空间，并指向堆空间的对象区，当方法调用完成后从栈空间回收。局部变量 new 出来时，在栈空间和堆空间中分配空 间，当局部变量生命周期结束后，栈空间立刻被回收，堆空间区域等待 GC 回收。方法调用时传入的 literal 参数，先在栈空间分配，在方法调用完成后从栈 空间分配。字

符串常量在 DATA 区域分配，this 在堆空间分配。数组既在栈空间分配数组名称，又在堆空间分配数组实际的大小！

JVM 中的堆和栈

JVM 是基于堆栈的虚拟机。JVM 为每个新创建的线程都分配一个堆栈。也就是说，对于一个 Java 程序来说，它的运行就是通过对堆栈的操作来完成的。堆栈以帧为单位保存线程的状态。JVM 对堆栈只进行两种操作:以帧为单位的压栈和出栈操作。
我们知道，某个线程正在执行的方法称为此线程的当前方法。我们可能不知道，当前方法使用的帧称为当前帧。当线程激活一个 Java 方法，JVM 就会在 线程的 Java 堆栈里新压入一个帧。这个帧自然成为了当前帧.在此方法执行期间，这个帧将用来保存参数，局部变量，中间计算过程和其他数据。这个帧在这里 和编译原理中的活动纪录的概念是差不多的。
从 Java 的这种分配机制来看,堆栈又可以这样理解：堆栈(Stack)是操作系统在建立某个进程时或者线程(在支持多线程的操作系统中是线程)为这个线程建立的存储区域,该区域具有先进后出的特性。

每一个 Java 应用都唯一对应一个 JVM 实例，每一个实例唯一对应一个堆。应用程序在运行中所创建的所有类实例或数组都放在这个堆中，并由应用所有 的线程共享。跟 C/C++不同，Java 中分配堆内存是自动初始化的。Java 中所有对象的存储空间都是在堆中分配的，但是这个对象的引用却是在堆栈中分 配,也就是说在建立一个对象时从两个地方都分配内存，在堆中分配的内存实际建立这个对象，而在堆栈中分配的内存只是一个指向这个堆对象的指针(引用)而已。