堆栈

对比分析

### 堆栈空间分配

栈（操作系统）：由操作系统自动分配释放 ，存放函数的[参数值](https://baike.baidu.com/item/%E5%8F%82%E6%95%B0%E5%80%BC)，[局部变量](https://baike.baidu.com/item/%E5%B1%80%E9%83%A8%E5%8F%98%E9%87%8F)的值等。其操作方式类似于数据结构中的栈。

堆（操作系统）： 一般由程序员分配释放， 若程序员不释放，程序结束时可能由OS回收，分配方式倒是类似于链表。

### 堆栈缓存方式

栈使用的是[一级缓存](https://baike.baidu.com/item/%E4%B8%80%E7%BA%A7%E7%BC%93%E5%AD%98" \t "_blank)， 他们通常都是被调用时处于存储空间中，调用完毕立即释放。

堆则是存放在[二级缓存](https://baike.baidu.com/item/%E4%BA%8C%E7%BA%A7%E7%BC%93%E5%AD%98)中，生命周期由虚拟机的垃圾回收算法来决定（并不是一旦成为孤儿对象就能被回收）。所以调用这些对象的速度要相对来得低一些。

## 区别介绍

### java

1. 栈(stack)与堆(heap)都是[Java](https://baike.baidu.com/item/Java/85979)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](https://baike.baidu.com/item/%E5%AF%84%E5%AD%98%E5%99%A8" \t "_blank)。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，栈数据在多个线程或者多个栈之间是不可以共享的，但是在栈内部多个值相等的变量是可以指向一个地址的，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(primitivetypes), 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](https://baike.baidu.com/item/%E8%87%AA%E5%8A%A8%E5%8F%98%E9%87%8F" \t "_blank)。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：

|  |  |
| --- | --- |
| 1  2 | 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的内存空间，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。 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JDK5.0中，这种表达式是可以的！因为[编译器](https://baike.baidu.com/item/%E7%BC%96%E8%AF%91%E5%99%A8)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. 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：【String str = "abc"，String str不要连着】

(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。】【上文说数据时存放在堆中，此文说数据存放在栈中】[因为此处不是通过new（）创建的啊]

(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。   
　　复制内容到剪贴板代码:

|  |  |
| --- | --- |
| 1  2  3 | 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](https://baike.baidu.com/item/JVM" \t "_blank)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：   
　　复制内容到剪贴板代码:

|  |  |
| --- | --- |
| 1  2  3  4  5 | 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：   
　　复制内容到剪贴板代码:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  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 |

我们再接着看以下的代码。   
　　复制内容到剪贴板代码:   
　　String str1 = new String("abc");   
　　String str2 = "abc";   
　　System.out.println(str1==str2); //false

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Stringstr = 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

## 理论知识

### 申请方式

stack:

由系统自动分配。 例如，声明在函数中一个[局部变量](https://baike.baidu.com/item/%E5%B1%80%E9%83%A8%E5%8F%98%E9%87%8F" \t "_blank)int b; 系统自动在栈中为b开辟空间

heap:

需要程序员自己申请，并指明大小，在c中[malloc函数](https://baike.baidu.com/item/malloc%E5%87%BD%E6%95%B0" \t "_blank)

如p1 = (char \*)malloc(10);

在C++中用new运算符

如p2 = new char[10];//(char \*)malloc(10);

但是注意p1、p2本身是在栈中的。

### 申请响应

栈：只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常提示[栈溢出](https://baike.baidu.com/item/%E6%A0%88%E6%BA%A2%E5%87%BA" \t "_blank)。

堆：首先应该知道操作系统有一个记录空闲[内存地址](https://baike.baidu.com/item/%E5%86%85%E5%AD%98%E5%9C%B0%E5%9D%80" \t "_blank)的[链表](https://baike.baidu.com/item/%E9%93%BE%E8%A1%A8)，当系统收到程序的申请时，会遍历该链表，寻 找第一个空间大于所申请空间的堆结点，然后将**该结点从空闲结点链表中删除**，并将该结点的空间分配给程序，另外，对于**大多数系统**，会在这块内存空间中的**首地址处记录本次分配的大小**，这样，代码中的delete语句才能正确的释放本内存空间。另外，由于找到的堆结点的大小不一定正好等于申请的大小，系统会自动的将多余的那部分重新放入空闲[链表](https://baike.baidu.com/item/%E9%93%BE%E8%A1%A8" \t "_blank)中。

### 申请限制

栈：在Windows下,**栈**是向**低地址扩展**的数据结构，是一块连续的内存的区域。这句话的意思是栈顶的地址和栈的最大容量是系统预先规定好的，在 WINDOWS下，栈的大小是2M（也有的说是1M，总之是一个编译时就确定的常数），如果申请的空间超过栈的剩余空间时，将提示overflow。因此，能从栈获得的空间较小。

堆：堆是向**高地址扩展**的数据结构，是不连续的内存区域。这是由于系统是用[链表](https://baike.baidu.com/item/%E9%93%BE%E8%A1%A8)来存储的空闲[内存地址](https://baike.baidu.com/item/%E5%86%85%E5%AD%98%E5%9C%B0%E5%9D%80)的，自然是不连续的，而**链表的遍历方向是由低地址向高地址**。堆的大小受限于计算机系统中有效的[虚拟内存](https://baike.baidu.com/item/%E8%99%9A%E6%8B%9F%E5%86%85%E5%AD%98)。由此可见，堆获得的空间比较灵活，也比较大。

### 效率比较

栈由系统自动分配，速度较快。但程序员是无法控制的。

堆是由new分配的内存，一般速度比较慢，而且容易产生[内存碎片](https://baike.baidu.com/item/%E5%86%85%E5%AD%98%E7%A2%8E%E7%89%87" \t "_blank),不过用起来最方便.

另外，在WINDOWS下，最好的方式是用VirtualAlloc分配内存，他不是在堆，也不是在栈,而是直接在进程的[地址空间](https://baike.baidu.com/item/%E5%9C%B0%E5%9D%80%E7%A9%BA%E9%97%B4" \t "_blank)中保留一块内存，虽然用起来最不方便。但是速度快，也最灵活

### 存储内容

栈： 在函数调用时，在大多数的C[编译器](https://baike.baidu.com/item/%E7%BC%96%E8%AF%91%E5%99%A8)中，参数是由**右往左入**栈的，然后是函数中的[局部变量](https://baike.baidu.com/item/%E5%B1%80%E9%83%A8%E5%8F%98%E9%87%8F)。注意[静态变量](https://baike.baidu.com/item/%E9%9D%99%E6%80%81%E5%8F%98%E9%87%8F)是不入栈的。

当本次[函数调用](https://baike.baidu.com/item/%E5%87%BD%E6%95%B0%E8%B0%83%E7%94%A8" \t "_blank)结束后，局部变量先[出栈](https://baike.baidu.com/item/%E5%87%BA%E6%A0%88)，然后是参数，最后栈顶[指针](https://baike.baidu.com/item/%E6%8C%87%E9%92%88)指向函数的返回地址，也就是[主函数](https://baike.baidu.com/item/%E4%B8%BB%E5%87%BD%E6%95%B0)中的下一条指令的地址，程序由该点继续运行。

堆：一般是在堆的头部用一个字节存放堆的大小。堆中的具体内容由程序员安排。

### 存取比较

char s1[] = "aaaaaaaaaaaaaaa";

char \*s2 = "bbbbbbbbbbbbbbbbb";

aaaaaaaaaaa是在运行时刻赋值的；

而bbbbbbbbbbb是在编译时就确定的；

但是，在以后的存取中，在栈上的[数组](https://baike.baidu.com/item/%E6%95%B0%E7%BB%84)比[指针](https://baike.baidu.com/item/%E6%8C%87%E9%92%88)所指向的字符串(例如堆)快。

比如：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | #include  void main()  {  char a = 1;  char c[] = "1234567890";  char \*p ="1234567890";  a = c[1];  a = p[1];  return;  } |

对应的汇编代码

10: a = c[1];

00401067 8A 4D F1 mov cl,byte ptr [ebp-0Fh]

0040106A 88 4D FC mov byte ptr [ebp-4],cl

11: a = p[1];

0040106D 8B 55 EC mov edx,dword ptr [ebp-14h]

00401070 8A 42 01 mov al,byte ptr [edx+1]

00401073 88 45 FC mov byte ptr [ebp-4],al

第一种在读取时直接就把字符串中的元素读到[寄存器](https://baike.baidu.com/item/%E5%AF%84%E5%AD%98%E5%99%A8" \t "_blank)cl中，而第二种则要先把[指针](https://baike.baidu.com/item/%E6%8C%87%E9%92%88" \t "_blank)值读到edx中，再根据edx读取字符，显然慢了。

### 小结

堆和栈的区别可以用如下的比喻来看出：

使用栈就象我们去饭馆里吃饭，只管点菜（发出申请）、付钱、和吃（使用），吃饱了就走，不必理会切菜、洗菜等准备工作和洗碗、刷锅等扫尾工作，他的好处是快捷，但是自由度小。

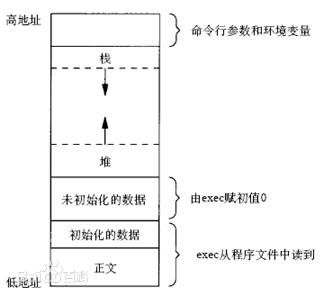
使用堆就象是自己动手做喜欢吃的菜肴，比较麻烦，但是比较符合自己的口味，而且自由度大。

## 主要分别

操作系统方面的堆和栈，如上面说的那些。

还有就是数据结构方面的堆和栈，这些都是不同的概念。这里的堆实际上指的就是（满足堆性质的）优先队列的一种数据结构，第1个元素有最高的优先权；栈实际上就是满足后进先出的性质的数学或数据结构。

虽然堆栈，堆栈的说法是连起来叫，但是他们还是有很大区别的，连着叫只是由于历史的原因。

堆与栈的分布

## 补充说明

堆栈是一种存储部件，即数据的写入跟读出不需要提供地址，而是根据写入的顺序决定读出的顺序。

形象来说，栈就是一条流水线，而流水线中加工的就是方法的主要程序，在分配栈时，由于程序是自上而下[顺序执行](https://baike.baidu.com/item/%E9%A1%BA%E5%BA%8F%E6%89%A7%E8%A1%8C" \t "_blank)，就将程序指令一条一条压入栈中，就像流水线一样。而堆上站着的就是工作人员，他们加工流水线中的商品，由程序员分配：何时加工，如何加工。而我们通常使用new[运算符](https://baike.baidu.com/item/%E8%BF%90%E7%AE%97%E7%AC%A6)为对象在堆上分配内存（C#,Java），堆上寻找对象的任务交给[句柄](https://baike.baidu.com/item/%E5%8F%A5%E6%9F%84)，而栈中由栈[指针](https://baike.baidu.com/item/%E6%8C%87%E9%92%88)管理



编程语言的书籍解释了值的类型是在堆栈上创建的，参考类型是在堆上创建的，而不解释这两件事情是什么。 我没有读到这个清楚的解释。 我明白什么是堆栈，但是他们在什么地方（真实的计算机内存）？

它们在多大程度上由操作系统或语言运行时控制？

他们的范围是什么？

什么决定了他们每个的大小？

是什么让一个更快？

堆栈是作为执行线程的暂存空间留出的内存。当一个函数被调用时，一个块被保留在堆栈的顶部，用于局部变量和一些簿记数据。当该函数返回时，该块将不被使用，并且可以在下一次调用函数时使用。堆栈始终以LIFO（后进先出）顺序保留;最近保留的块总是下一个要被释放的块。这使得跟踪堆栈变得非常简单。从堆栈释放一个块只不过是调整一个指针。

堆是为动态分配留出的内存。与堆栈不同，没有强制的模式来分配和释放堆中的块;你可以随时分配一个块，随时释放它。这使得在任何给定的时间跟踪堆的哪些部分被分配或释放变得更加复杂;有许多自定义堆分配器可用于调整不同使用模式的堆性能。

每个线程都得到一个堆栈，而应用程序通常只有一个堆（尽管针对不同类型的分配有多个堆并不罕见）。

直接回答你的问题：

它们在多大程度上由操作系统或语言运行时控制？

在创建线程时，OS为每个系统级线程分配堆栈。通常，语言运行时会调用操作系统来为应用程序分配堆。

他们的范围是什么？

堆栈被附加到一个线程，所以当线程退出时，堆栈被回收。堆通常在应用程序启动时由运行时分配，并在应用程序（技术上处理）退出时回收。

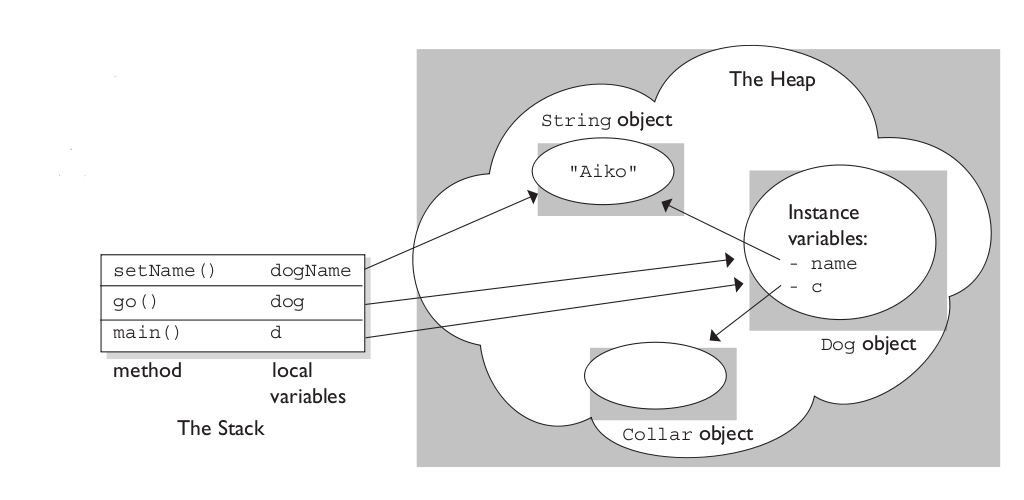
什么决定了他们每个的大小？

创建线程时设置堆栈的大小。堆的大小是在应用程序启动时设置的，但可以随着空间需求而增长（分配器从操作系统请求更多的内存）。

是什么让一个更快？

堆栈更快，因为访问模式使分配和释放内存变得微不足道（指针/整数只是递增或递减），而堆在分配或解除分配中涉及更复杂的簿记。 另外，堆栈中的每个字节都会经常被重复使用，这意味着它往往会被映射到处理器的缓存中，使其非常快速。 堆的另一个性能是，堆主要是一个全局资源，通常必须是多线程安全的，即每个分配和释放需要（通常）与程序中的所有其他堆访问同步。

一个明确的示范：



堆栈：

像堆一样存储在计算机RAM中。

在堆栈上创建的变量将超出范围，并自动释放。

与堆上的变量相比，分配速度要快得多。

用一个实际的堆栈数据结构来实现。

存储本地数据，返回地址，用于参数传递。

如果使用了太多的堆栈（主要来自无限或太深的递归，非常大的分配），可能会发生堆栈溢出。

在堆栈上创建的数据可以在没有指针的情况下使用。

如果您在编译之前确切知道需要分配多少数据，那么您将使用堆栈，并且它不是太大。

程序启动时，通常已经确定了最大尺寸。

堆：

像堆栈一样存储在计算机RAM中。

在C ++中，堆上的变量必须手动销毁，永远不会超出范围。数据被删除，删除[]或释放。

与堆栈中的变量相比，分配速度较慢。

根据需要用来分配一块数据供程序使用。

当有大量的分配和释放时，可以有碎片。

在C ++或C中，在堆上创建的数据将由指针指向并分别用new或malloc分配。

如果请求分配过大的缓冲区，可能会导致分配失败。

如果您不知道在运行时需要多少数据，或者需要分配大量数据，则可以使用堆。

负责内存泄漏。

https://stackoverflow.com/questions/79923/what-and-where-are-the-stack-and-heap