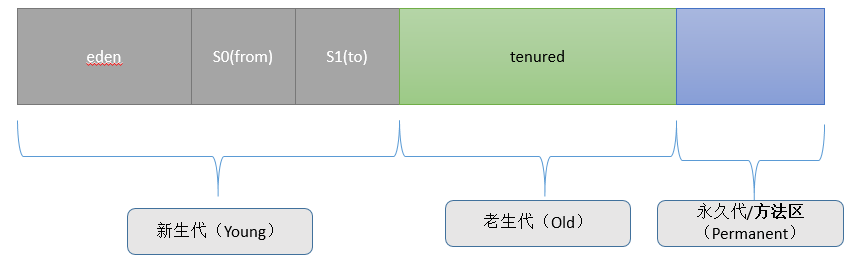
1. 栈

每一个Java虚拟机线程都有一个私有的Java栈。一个线程的Java栈在线程创建时被创建，在线程结束时被销毁。栈中保存着帧信息，栈中保存着局部变量（包括局部的对象的变量）、方法参数

1. Hotspot JVM堆结构（永久代应该是不属于堆，有些人说是属于，有些不属于，但是从JVM的配置来看-Xms和-Xmx是设置堆内存的初始值和最大值，但是permSize是另外设置的来看，应该是不属于堆结构的，而且从jdk1.8即java 8.X后永久代不存在取而代之是元数据区来看应该是不属于堆）



**方法区（method area）**只是**JVM规范**中定义的一个概念，用于存储类信息、常量池、静态变量、JIT编译后的代码等数据，具体放在哪里，不同的实现可以放在不同的地方。而**永久代**是**Hotspot**虚拟机特有的概念，是方法区的一种实现，别的JVM都没有这个东西。  
  
在Java 6中，方法区中包含的数据，除了JIT编译生成的代码存放在native memory的CodeCache区域，其他都存放在永久代；  
在Java 7中，Symbol的存储从PermGen移动到了native memory，并且把静态变量从instanceKlass末尾（位于PermGen内）移动到了java.lang.Class对象的末尾（位于普通Java heap内）；  
在Java 8中，永久代被彻底移除，取而代之的是另一块与堆不相连的本地内存——元空间（**Metaspace**）,‑XX:MaxPermSize 参数失去了意义，取而代之的是-XX:MaxMetaspaceSize  
  
在绝大部分情况下，对象首先分配到新生代的eden区，在一次新生代回收后，如果对象还存活，则会进入到s0或s1，之后，每经过一次新生代回收，对象如果存活，它的年龄就会加1。当对象的年龄达到一定条件后，就会被认为是老年对象，从而进入老年代。

1. 例子说明Java堆、方法区和Java栈之间的关系

**public** **class** SimpleHeap {

**private** **int** id;

**public** SimpleHeap(**int** id){

**this**.id = id;

}

**public** **void** show(){

System.***out***.println("My Id is"+id);

}

**public** **static** **void** main(String[] args){

SimpleHeap s1 = **new** SimpleHeap(1);

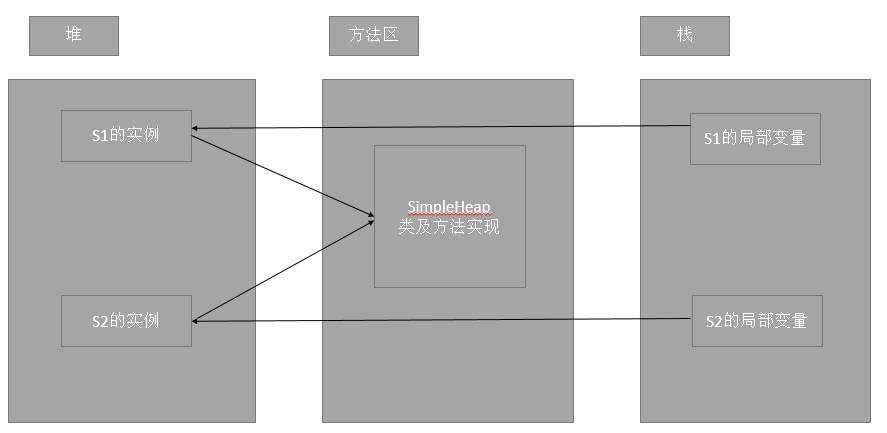
SimpleHeap s2 = **new** SimpleHeap(2);

s1.show();

s2.show();

}

}



1. 局部变量表

局部变量表是栈帧的重要组成部分之一。它保存函数的参数以及局部变量。局部变量表中的变量只有在当前的函数调用中有效，当函数调用结束后，随着函数栈帧的销毁，局部变量表也会随之销毁。

**public** **class** GcTest{

**public** **void** localvarGc1(){

**byte**[] a = **new** **byte**[6\*1024\*1024];

System.*gc*();

}

**public** **void** localvarGc2(){

**byte**[] a = **new** **byte**[6\*1024\*1024];

a = **null**;

System.*gc*();

}

**public** **void** localvarGc3(){

{

**byte**[] a = **new** **byte**[6\*1024\*1024];

}

System.*gc*();

}

**public** **void** localvarGc4(){

{

**byte**[] a = **new** **byte**[6\*1024\*1024];

}

**int** b =10;

System.*gc*();

}

**public** **void** localvarGc5(){

localvarGc1();

System.*gc*();

}

**public** **static** **void** main(String[] args){

//每个方法最初都定义了一个byte数组a,大小为6\*1024\*1024=6M

GcTest t = **new** GcTest();

t.localvarGc1();//在申请空间后即调用回收，byte数组被a引用无法加回收

t.localvarGc2();//申请空间后，将a=null，byte数组引用失去，会被回收

t.localvarGc3();//局部变量a在作用域外失效，虽然出了其作用域，但是变量a仍然在变量表中，也指向byte数组，所以不能被回收

t.localvarGc4();//局部变量a在作用域外失效，申请变量b使其复用a的空间，a被销毁，byte数组没有被引用，所以被销毁

//先调用localvarGc1(),在localvarGc1()中变量a没有失效，所以没被回收，但是执行完后栈帧被销毁，局部变量表也被销毁，byte数组失去引用后被销毁

t.localvarGc5();

}

}

1. 操作数栈

是栈帧的重要内容之一，它主要用于保存计算过程的中间结果，同时作为计算过程中变量临时的存储空间。

也是先进后出的结构，只支持入栈和出栈操作。如下图，iadd指令，它会在操作栈将两个整数出栈并进行加法后，计算结果会入栈



1. 帧数据区

除了局部变量表和操作数栈外，栈帧还需要一些数据来支持常量池的解析、正常方法返回和异常处理。

1. 栈上分配和逃逸分析

栈上分配是Java虚拟机提供的一项优化技术，它的基本思想是，对于那些线程私有的对象（这里指不可能被其他线程访问的对象），可以将它们打散分配在栈上，而不是分配在堆上，分配在栈上的好处是可以在函数调用结束后自行销毁，而不需要GC的介入，从而提高了系统的性能。

栈上分配的一个技术基础是进行逃逸分析，判断对象的作用域是否有可能逃逸。

以下对象u发生了逃逸，可以被其他线程（方法）访问到

**private** **static** User u;

**public** **static** **void** alloc(){

u = **new** User();

}

以下方法没有发生逃逸，因为User以局部变量存在且没有被返回

**public** **static** **void** alloc(){

User u = **new** User();

}

以下User发生了逃逸，虽然定义是局部变量但是被返回，即可被其他线程访问到

**public** **static** **User** alloc(){

User u = **new** User();

return u;

}