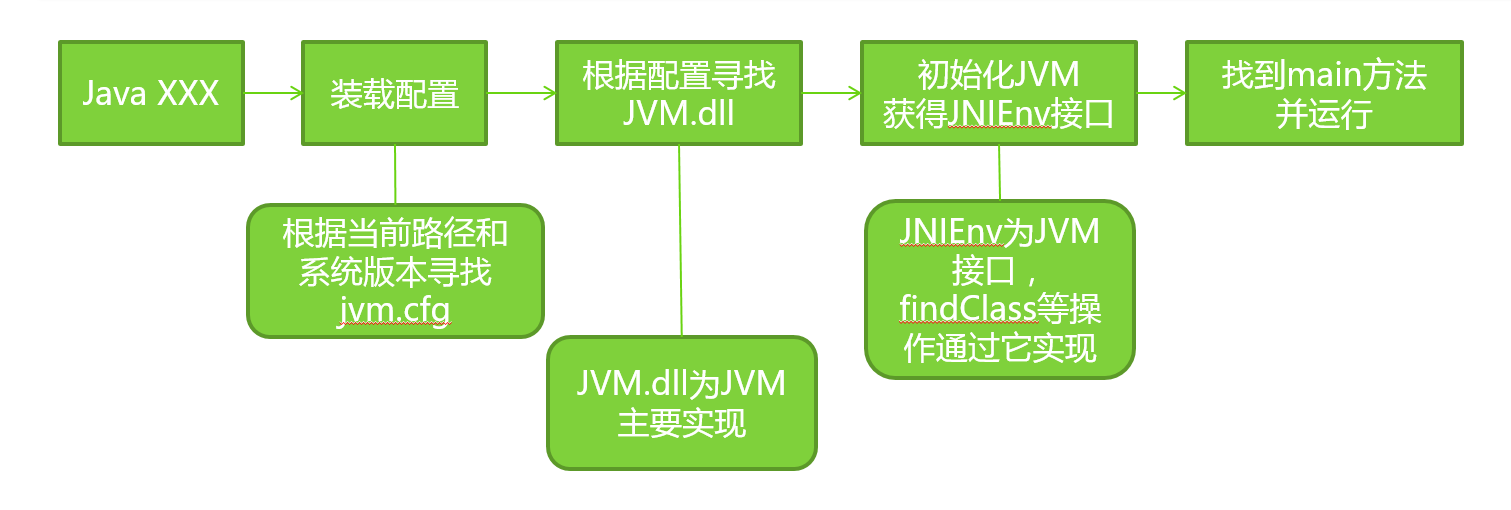
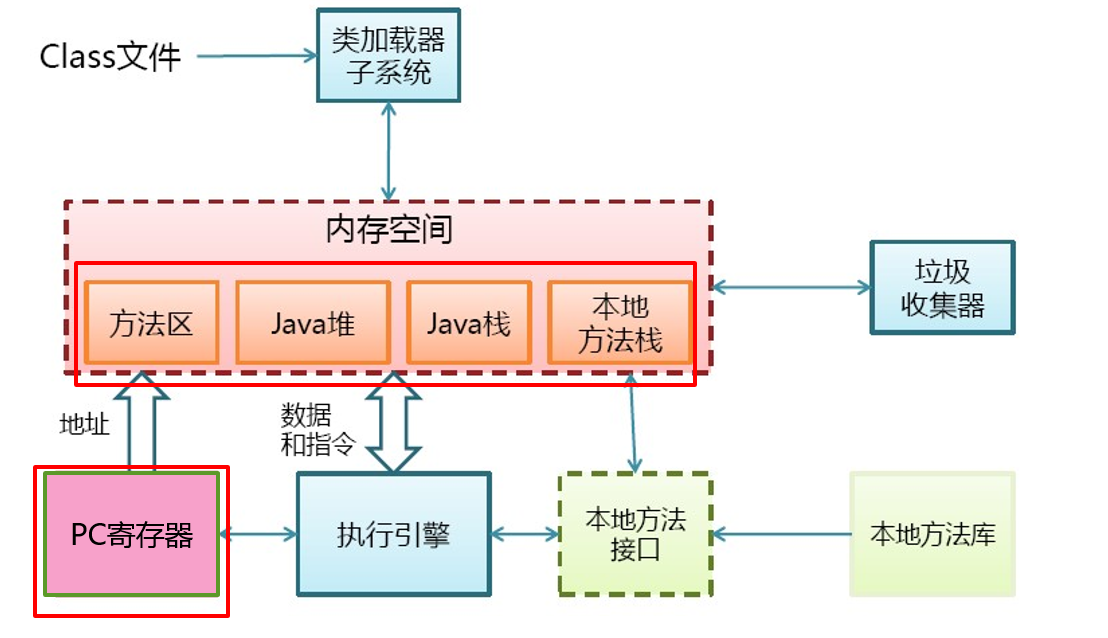
## JVM启动流程

Java XXX（Java命令）



## JVM基本结构



### PC寄存器

* + 每个线程拥有一个PC寄存器
  + 在线程创建时 创建
  + 指向下一条指令的地址
  + 执行本地方法时，PC的值为undefined

### 方法区

* + 保存装载的类信息（保存类的元信息对类进行描述）
    - 类型的常量池
    - 字段，方法信息
    - 方法字节码
  + 通常和永久区(Perm)关联在一起

保存一些相对静止的数据，虽然会发生变动，但是相对来说是稳定的。

不是一成不变的

JDK6时，String等常量信息置于方法区

JDK7时，已经移动到了堆

### Java堆

* + 和程序开发密切相关
  + 应用系统对象都保存在Java堆中
  + 所有线程共享Java堆（全局共享）
  + 对分代GC来说，堆也是分代的
  + GC的主要工作区间



### Java栈

* + 线程私有
  + 栈由一系列帧组成（因此Java栈也叫做帧栈）
  + 帧保存一个方法的局部变量、操作数栈、常量池指针
  + 每一次方法调用创建一个帧，并压栈

#### Java栈 – 局部变量表 包含参数和局部变量

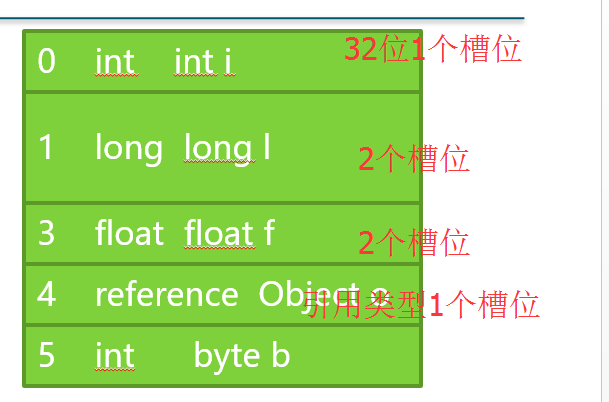
槽位可以容纳32位的数据类型

public class StackDemo {

public static int runStatic(int i,long l,float f,Object o ,byte b){

return 0;

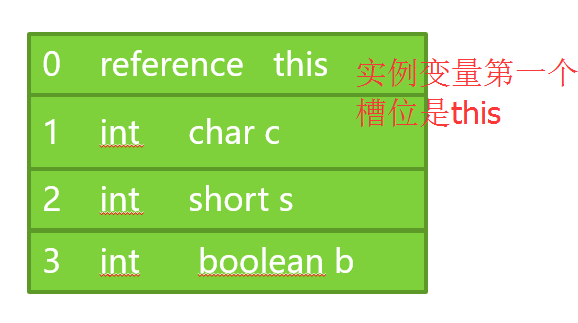
}



public int runInstance(char c,short s,boolean b){

return 0;

}



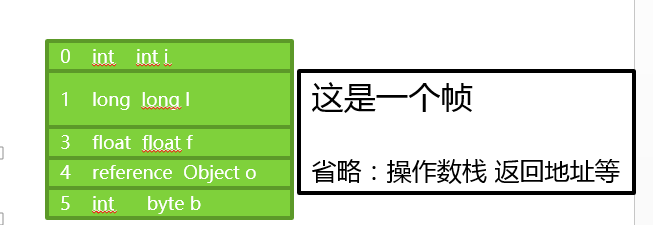
}

#### Java栈 – 函数调用组成帧栈

public static int runStatic(int i,long l,float f,Object o ,byte b){

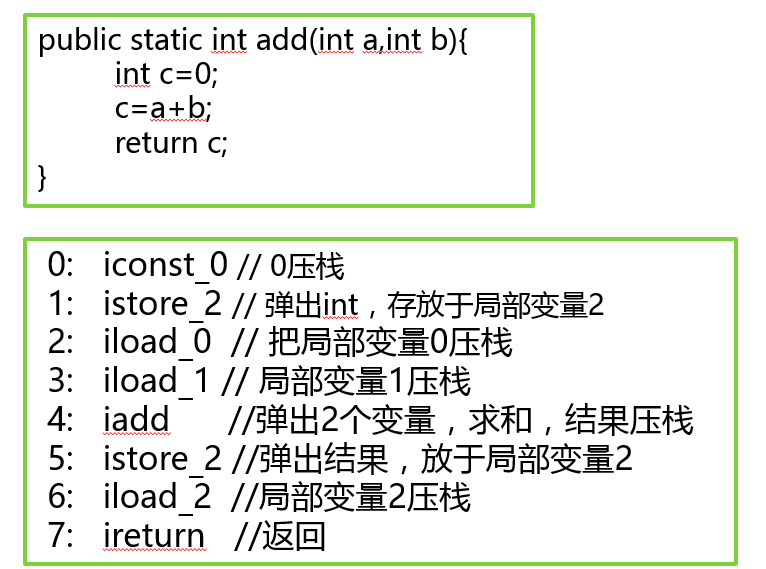
return runStatic(i,l,f,o,b);

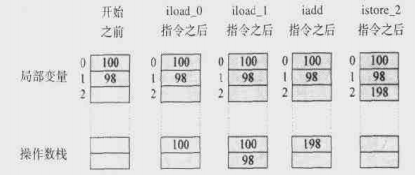
}



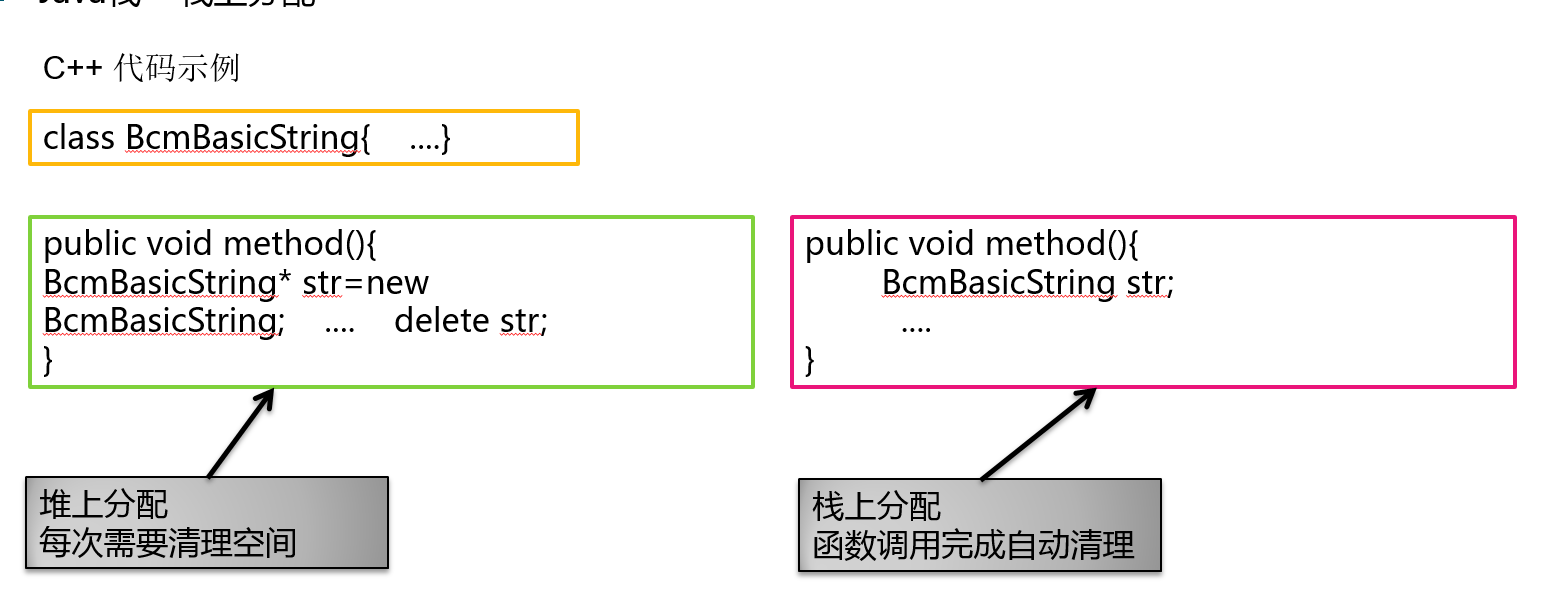
#### Java栈 – 操作数栈

* + Java没有寄存器，所有参数传递使用操作数栈

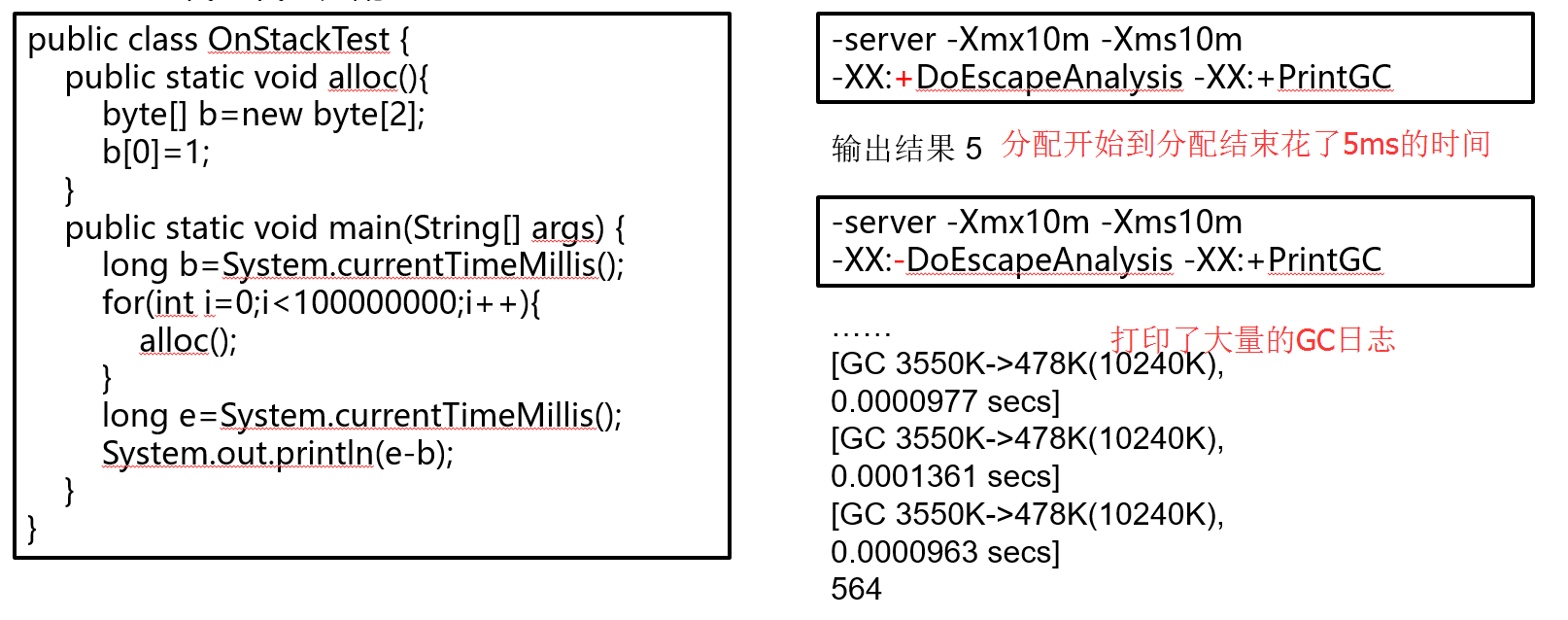




#### Java栈 – 栈上分配



如果内存不够大，分配了大量的内存之后一定会发送GC。

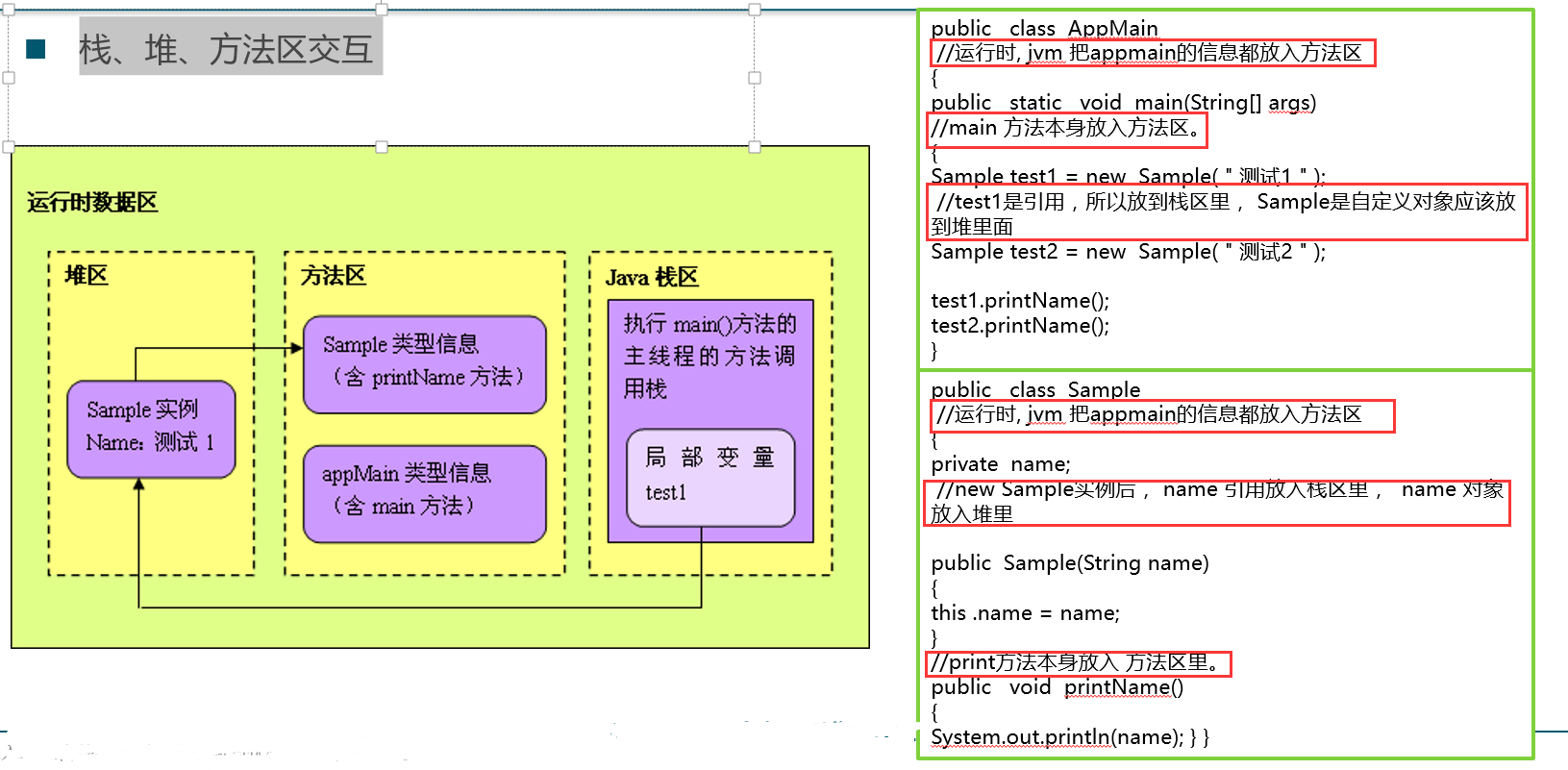


在栈上分配会减轻GC的压力

* + 小对象（一般几十个bytes），在没有逃逸的情况下，可以直接分配在栈上
  + 直接分配在栈上，可以自动回收，减轻GC压力
  + 大对象或者逃逸对象无法栈上分配

**只要这个引用有可能会被多个线程所引用，这就发生了逃逸**。

#### 栈、堆、方法区交互



对象本身是在堆中，栈中存放的是引用；

类的信息的描述，以及包括类当中的方法的字节码是在方法区的；

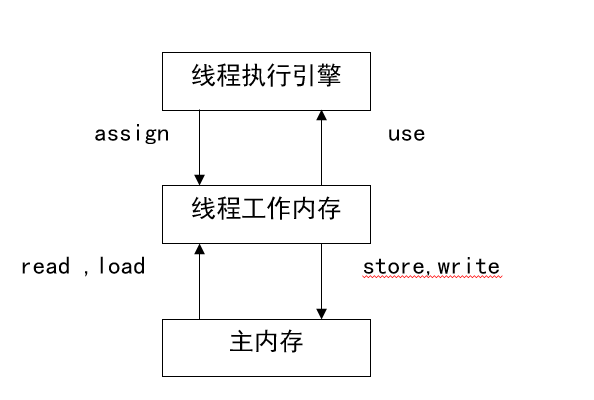
堆中保存的类的实例；

为了能让递归函数调用的次数更多一些，应该怎么做呢？（递归函数比较容易造成栈溢出）

### 内存模型

每一个线程有一个工作内存和主存独立

工作内存存放主存中变量的值的拷贝



当数据从主内存复制到工作存储时，必须出现两个动作：

1. 由主内存执行的读（read）操作；
2. 由工作内存执行的相应的load操作；

当数据从工作内存拷贝到主内存时，也出现两个操作：

第一，由工作内存执行的存储（store）操作；

第二，由主内存执行的相应的写（write）操作

每一个操作都是原子的，即执行期间不会被中断

对于普通变量，一个线程中更新的值，不能马上反应在其他变量中（从内存模型的图中可以发现，工作内存和主内存之间进行同步需要一定的时间，因此有一个时间差）

如果需要在其他线程中立即可见，需要使用 volatile 关键字

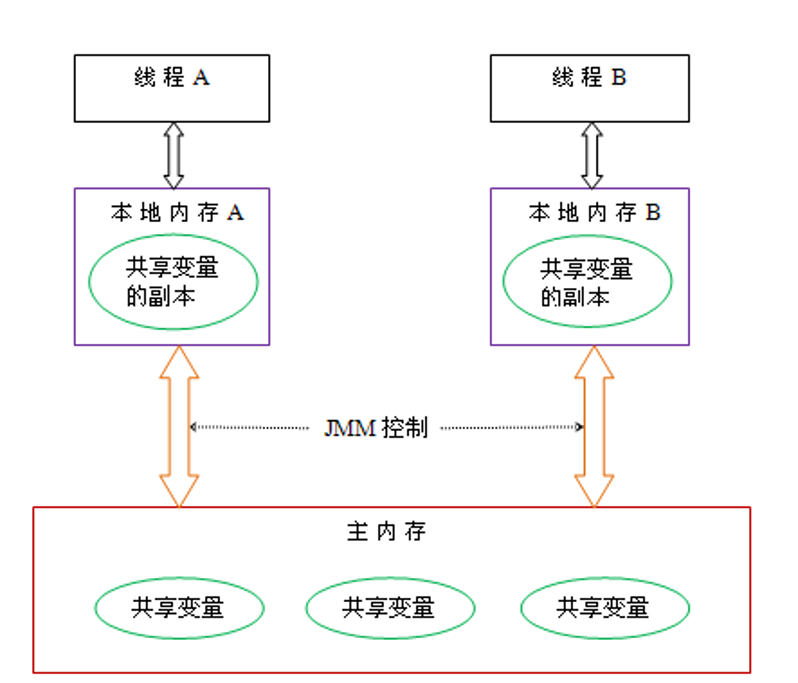
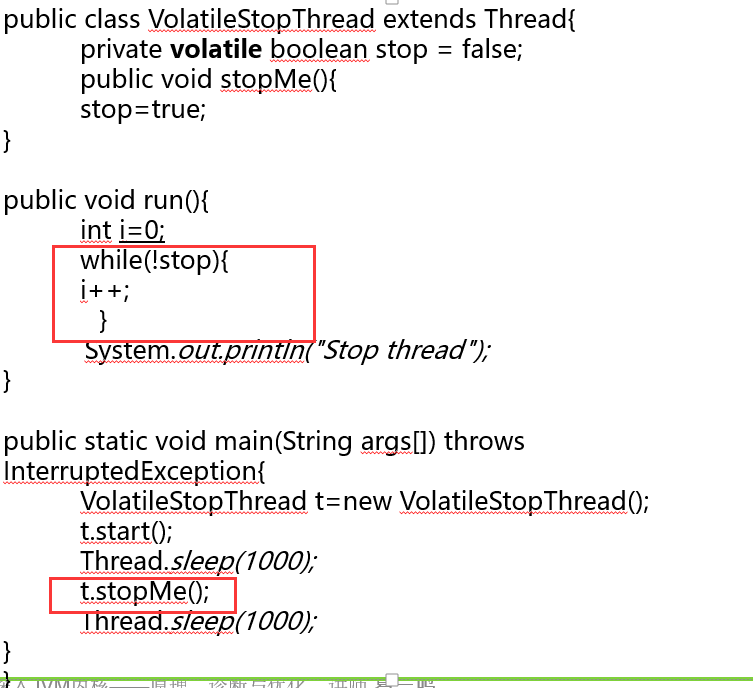


图-时间差的原因

#### volatile 不能代替锁

一般认为volatile 比锁性能好（不绝对）；选择使用volatile的条件是：语义是否满足应用



没有volatile -server 运行 无法停止

### 可见性

一个线程修改了变量，其他线程可以立即知道

保证可见性的方法：

volatile

synchronized （unlock之前，写变量值回主存）

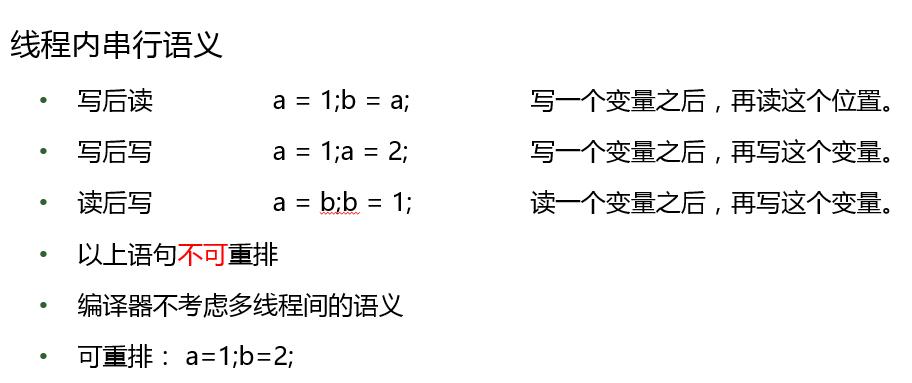
final(一旦初始化完成，其他线程就可见)

### 有序性

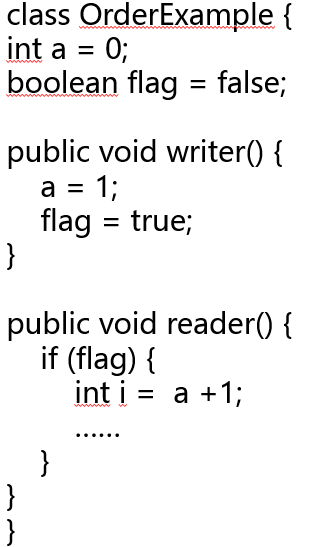
在本线程内，操作都是有序的

在线程外观察，操作都是无序的。（指令重排 或 主内存同步延时）

#### 指令重排



1，指令重排 – 破坏线程间的有序性

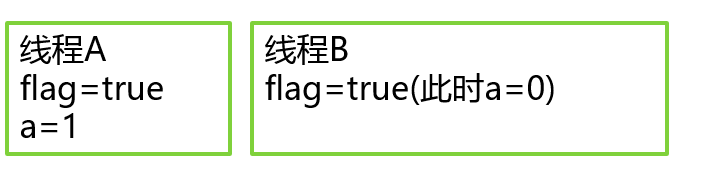


线程A首先执行writer()方法

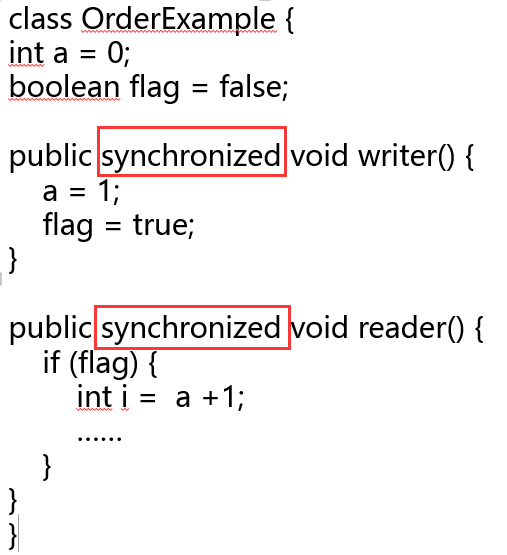
线程B线程接着执行reader()方法

线程B在int i=a+1 是不一定能看到a已经被赋值为1

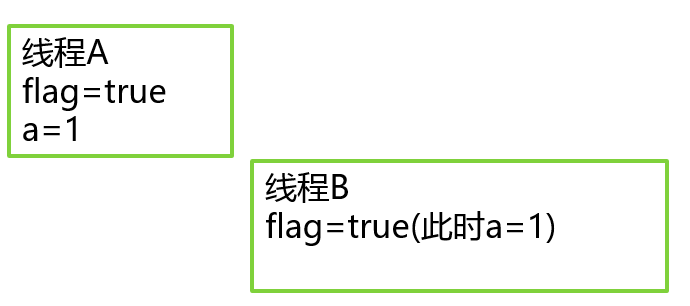
因为在writer中，两句话顺序可能打乱



1. 指令重排 – 保证有序性的方法



同步后，即使做了writer重排，因为互斥的缘故，reader 线程看writer线程也是顺序执行的。



3，指令重排的基本原则

* 程序顺序原则：一个线程内保证语义的串行性（a=4;b=a+4;这样的语句是不能重排的，重排后语义就不一致了）；
* volatile规则：volatile变量的写，先发生于读；
* 锁规则：解锁(unlock)必然发生在随后的加锁(lock)前
* 传递性：A先于B，B先于C 那么A必然先于C
* 线程的start方法先于它的每一个动作
* 线程的所有操作先于线程的终结（Thread.join()）
* 线程的中断（interrupt()）先于被中断线程的代码（保证了中断之后的代码不会被执行）
* 对象的构造函数执行结束先于finalize()方法

### 字节码bytecode运行方式-编译和解释运行的概念

* 解释运行
  + 解释执行以解释方式运行字节码
  + 解释执行的意思是：读一句执行一句
* 编译运行（JIT）
  + 将字节码编译成机器码
  + 直接执行机器码
  + 运行时编译
  + 编译后性能有数量级的提升

JIT的性能远优于解释运行