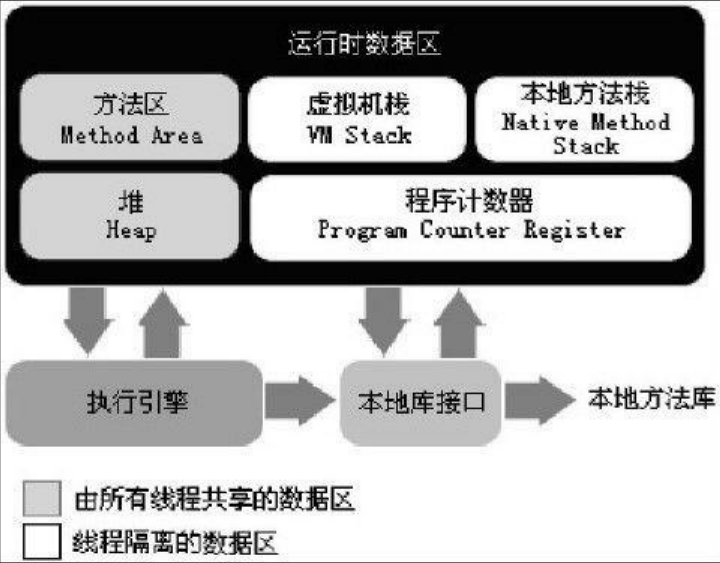
**Java虚拟机**

***1.Java虚拟机的五个内存区域***



***1.1程序计数器***

占用的内存大小较小，几乎可以忽略不计。是线程私有的，是程序运行的信号指示器。字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。

***1.2虚拟机栈***

每个方法在执行的同时都会创建一个栈帧（Stack Frame [1] ）用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每一个方法从调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。

***1.3本地方法栈***

执行的是本地的方法

***1.4堆***

掌管着所有java实例的生命周期。

逻辑划分可以划分为:新生代和老年代

如果在细分，可以分为:eden,from survivor和to survivor,而eden,from survivor和to survivor是新生代的划分，跟老年代没有关系。

每个对象在堆中由三种格式构成，对象头:存储对象相关信息，类型指针，锁等。属性：对象本身的属性值，占位符：用来空位不缺，因为一个对象要占8字节的整数倍

***1.5方法区(非堆)***

它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。

运行时常量池也包含在里面

***2. 对象在内存中的管理***

一种是有序，一种是无序的，排列的规则由jvm的垃圾收集器定

有序的会有一个指针，当创建新的对象时，指针会根据对象本身的大小移动，进行计算，得出已占空间等信息。

因为对象会频繁创建，所以这种指针移动是非线程安全的，有可能对象创建了，指针还没来得及移动就被其他对象创建修改了。所以这里面会有两种解决方式，一种是同步方式，创建失败后会试图重新创建；另一种是使用线程缓冲区进行空间的预占,这个需要额外的配置去设置

***3. Eclipse插件分析内存溢出***

可以在Eclipse的市场上搜索Memory Analysis插件去分析内存溢出文件

***4. 垃圾收集怎么判定一个对象已死?***

一般认为是使用计数算法，就是如果对象存在引用，那么就+1，一些非主流的JVM是这么实现的。但是这存在一个问题，就是两个对象如果相互间存在引用的话，那么垃圾收集器将不会回收这两个对象。

大部分虚拟机是使用可达性分析去计算对象是否需要被回收，就是如果对象在GCROOT上面没有可达的路径，那么可以判断这个对象需要回收。

***5. 方法区的垃圾回收***

对于方法区的垃圾回收一般是运行时常量池和类的定义

对于运行时常量池比较容易判定，比如String.intern()方法生成的常量.

类的定义就不容易被回收，因为它需要判断类的类型是否在堆上存在引用，对于像大量使用反射、动态代理GCLIB和jsp生成的的类就不容易被虚拟机计算并回收

如果调试的话可以使用-verbose:class、-XX:+TraceClassLoading、-XX:+TraceClassUnLoading来查看类的加载以及卸载信息

***6. 垃圾收集算法***

主要是针对堆(Heap)进行垃圾收集

***6.1 标记-清除***

目前好像只有老年带的CMS垃圾收集器才会使用这个算法，过程就是将需要垃圾收集的对象标记出来，下一步就是直接清理掉这部分对象，这个有一个弱点就是这种算法会产生大量的磁盘碎片

***6.2 复制算法***

新生代一般都使用这个算法，就是将不需要回收的对象诺到内存的一边，然后一次性清理需要回收的那部分内存

比如新生代的Eden,from survivor和to survivor。在垃圾收集时一般会将存活的对象复制到from survivor内存区域中，然后清理掉Eden区剩下的对象。

这个弱点就是会浪费一定一块内存区域，这个收集算法比较适合用在朝生夕死的新生代当中

Eden,from survivor和to survivor一般设置为8:1:1

***6.3 复制整理算法***

将存活的对象挪到内存的一边，将边界外的垃圾进行清除。

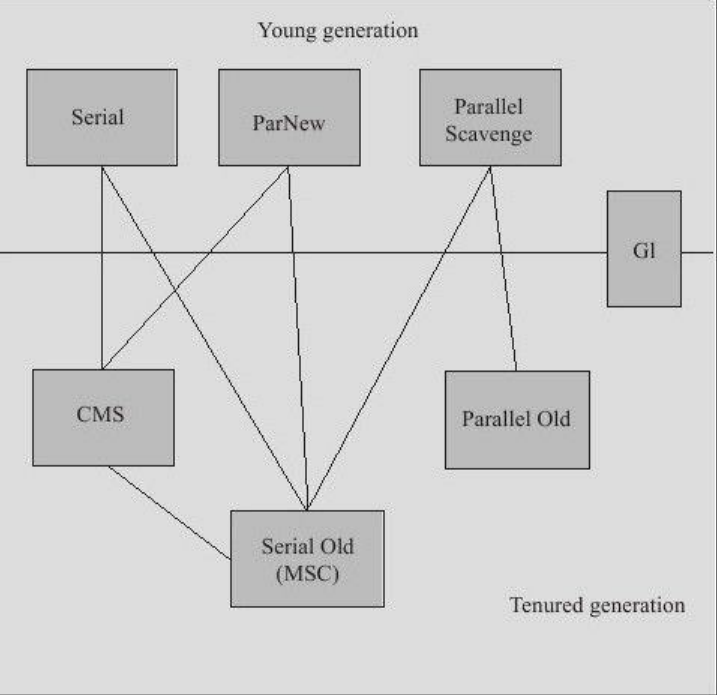
***6.4 分代收集算法***

其实就是新生代和老年带按照不同的垃圾收集算法去进行垃圾收集。

***7. finalize()方法***

在gc时执行的方法，只会执行一次。对象可以在这里将自己指向到其他引用进行自救。但是注意，这个方法只能执行一次。

***8. 垃圾收集器***



新生代:Serial，ParNew，Parallel Scavenge

老年代:CMS，Serial Old，Parallel Old

横跨新生代老年代收集器:G1(1.7之后才出现)

***8.1 新生代对应的老年代收集器***

Serial->(CMS,Serial Old)

ParNew->(CMS,Serial Old)

Prallel Scavenge(Serial Old, Parallel Old)

***8.2 各种收集器的原理***

***Serial***:单线程垃圾收集器，比较适合在虚拟机在Client模式下。

***ParNew***:Serial的多线程版本，启动多个线程同时进行新生代的垃圾收集，只不过如果在CPU不超过4个的情况下，效率不一定比Serial高，因为他有线程切换的开销

***Prallel Scavenge***: 也是多线程垃圾收集器，那么他跟ParNew有什么不一样呢？

ParNew:专注的是低停顿，就是以启用多个线程的方式尽量缩短垃圾收集的停顿时间.比较适合web应用交互类型的应用，提升用户体验

Prallel Scavenge:专注的是吞吐量，就是垃圾收集时间占总的CPU运行时间的一个比例，比较适合没有太多交互的后台应用。

***Serial Old***: 老年代的单线程垃圾收集器

***CMS***: 多线程垃圾收集器，只不过他使用的是标记清除算法，会导致大量的磁盘碎片，如果清理后空间仍然不够，那么会触发一次Full GC,使用Serial Old再次进行垃圾清理

***Parallel Old***: 也是老年代多线程垃圾收集器，目前只能跟Prallel Scavenge配合使用。

***9. 常用的垃圾收集器设置***

-XX:+UseSerialGC 使用(Serial + Serial Old)组合

-XX:+UseParNewGC 使用(ParNew + Serial Old)组合

-XX:+UseConcMarkSweepGC 使用(ParNew + CMS + Serial Old)组合，而Serial Old则在Concurrent Mode Failure失败之后使用

-XX:+UseParallelGC 使用(Parallel Scavenge + Serial Old)组合

-XX:+UseParallelOldGC 使用(Parallel Scavenge + Parallel Old)组合

-XX:SurvivorRatio=8 代表Eden:survivor比例是8:1

-XX:ParallelGCThreads 设置垃圾收集时的线程数

***10. Minor GC和Major GC(Full GC)区别***

Minor GC:是新生代的垃圾回收动作，会比较频繁

Major GC(Full GC):是指老年代的垃圾回收动作

***11. 对象从新生代进入老年代***

一种是新生代对象没有空间存放大对象时，大对象直接进入老年代

另一种是对象在新生代中熬过一次Minor GC年龄就+1

可以通过-XX:MaxTenuringThreshold=15来设置新生代对象多大年龄后进入老年代

也可以动态判定，每次进入老年代的对象平均年龄，如果对象大于这个数值，则进入老年代

最后一种是通过担保，如果新生代Minro GC后仍然没有空间，那么可以通过担保进入老年代，如果老年代也没有空间那么会触发一次Full GC

***12. jps工具***

jps工具跟linux的ps命令差不多，前面的数字跟ps一样，是jvm进程的id，跟系统进程id一致

jps -v包含java程序启动时的jvm参数信息

***13. jstat工具***

jstat可以监控远程或者本地的jvm。

首先通过ps或者jps找出LVMID，然后组装需要操作的option。比如下面

查询本机某一个jvm程序的新生代gc情况:jstat -gcnew 13275

还有可以查看类的加载情况:jstat -class 13275

其他参数用的时候再了解下就可以了

***14. jinfo工具***

实时查看和调整虚拟机的启动参数，他跟jps -v不一样的是，他可以查看一些指定未被显示的参数，用-flag

比如显示CMS收集器在老年代被占用多少后触发gc，默认为68%

jinfo -flag CMSInitiatingOccupancyFraction 13275

另外还可以用-sysprops选项去查看System.getProperties的参数,另外还可以进行修改(不详述，需要用的时候再查阅)

jinfo -sysprops 13275

***15 jmap工具***

可用于生成堆的转存快照，也就是我们长说的dump文件。个人认为这是个学习jvm的好工具

这个命令有很多值得参考的地方，下面列举一下

查看堆的内存设置和使用信息:jmap -heap 1609

查看待收集的对象个数:jmap -finalizerinfo 1609

查看虚拟机已加载的类，实例个数等信息(因为一般程序都会比较多，所以建议输出到一个文本文件查阅)

jmap -histo 1609 > objects.cc

查看方法区的内存状态(仅linux下有效，会计算比较慢):jmap -permstat 1609

dump文件(文件可以放到eclipse上面去使用Memory Analysis插件分析):jmap -dump:format=b,file=dumpfile\_1609.bin 1609

***16. jhat工具***

JVM Heap Analysis Tool，与jmap搭配使用，用来分析jmap生成的转存快照

书上说没什么用，原因是一般不会在目标机器直接分析快照文件，因为会占用资源。不过个人觉得，在实验阶段还是挺好的，开发机器也能做这个事情.

比如刚dump出来的文件dumpfile\_1609.bin,直接运行jhat dumpfile\_1609.bin，待文件分析完成后，浏览器访问该机器的7000端口<http://192.168.33.101:7000/>

看了一下那个文件，确实比较简陋

***17. jstack工具***

线程分析工具，用来分析线程死锁原因，在等待什么资源。

jstack -l 1609

***18. jconsole工具***

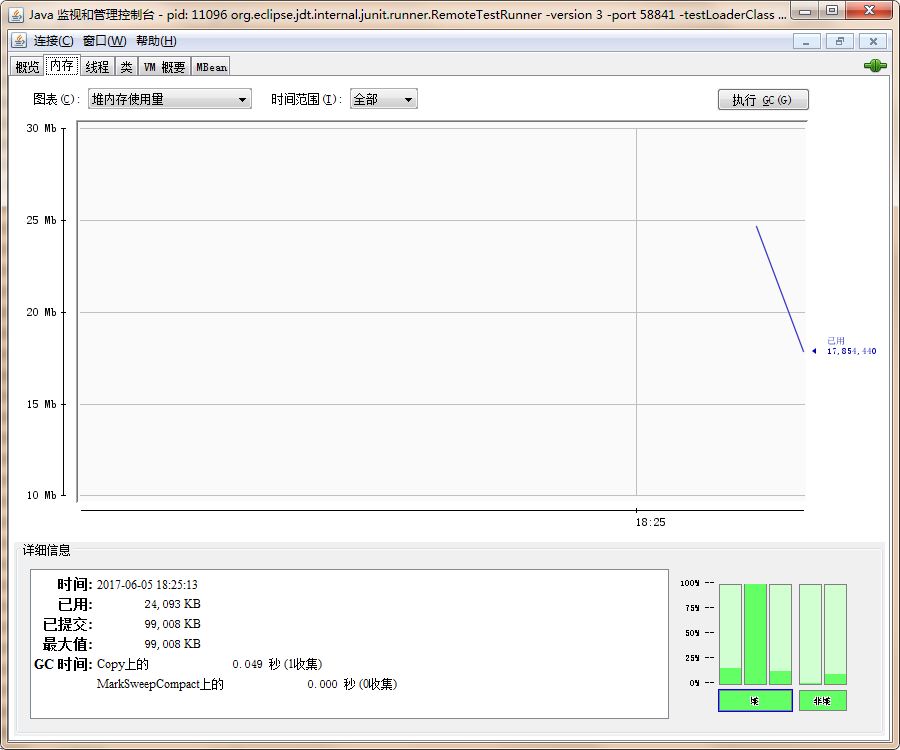
是一个可视化工具，可以监测内存的使用状况，包括新生代，老年代，而且可以看到新生代Eden区的运行状况

可以监测线程停顿状况，线程停顿引发的状况包括等待外部资源，死循环，锁等待等。

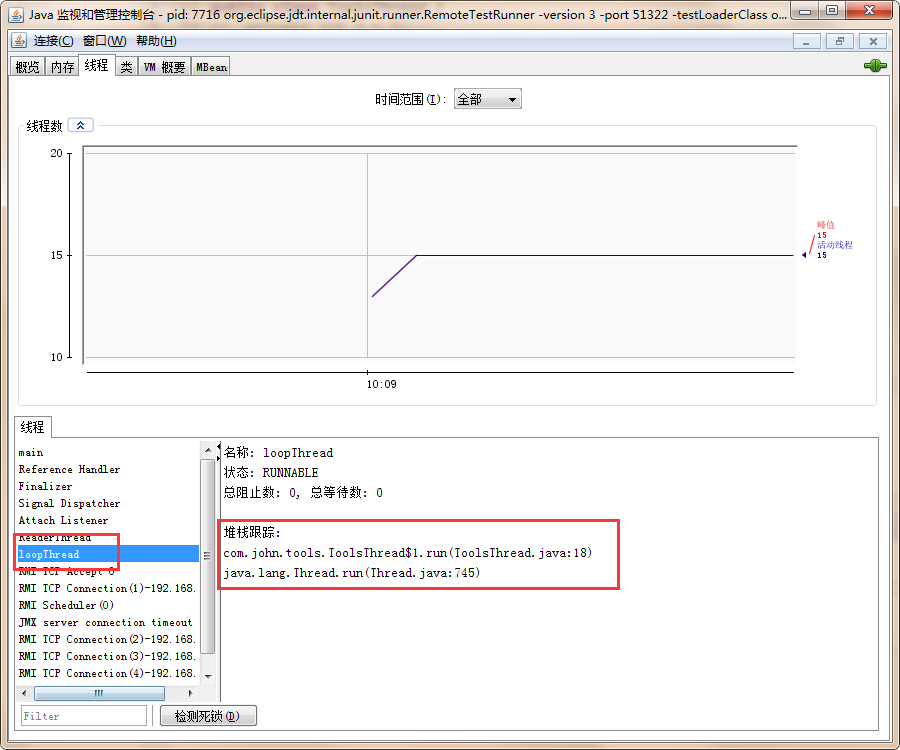
jconsole可以通过jmx链接远程的jvm程序进行监测

***18.1 小程序监测***

第一个设定好堆相关大小之后，可以通过jconsole看到Eden,survivor和Old内存之间的变化。



第二个程序主要是体现在线程上，程序运行当中的线程都会展示在上面，如果死循环，还可以看到线程停顿对应的方法和具体代码行



***19. VisualVM多合一工具***

囊括上面所有的jps，jstat，jinfo等功能，所以叫All-in-one。这个是官方主打的工具，而且有很多功能强大的插件可以安装。

***20. jmx配置(tomcat)***

这里不作描述。如有需要请查看源文件(myProgram\note\基础\jvm)

***21. 调优方案***

***21.1 优化服务器慢的问题***

背景是一个15W/PV的小型网站，但是会频繁阅读本地的资源

原来是因为垃圾收集太频繁，所以导致程序比较慢。因为原来用的是32位的机器，在linux下顶多能使用4个G的内存,所以现场调整为64位的机器，并且堆空间设置为12个G。问题更加严重，12个G的垃圾回收一下要用10多秒。而且频繁回收的现象没有得到改善。

本身64位系统由于指针膨胀等原因，导致内存消耗比32位的更大，所以推荐在32位系统上部署多个一样的程序，在前段进行反向代理。不过这个要注意IO等资源竞争的问题，另外程序大量使用HashMap作为K/V缓存会浪费内存，所以推荐使用集中式缓存机制。

***21.2 堆内存正常回收，但是出现OOM***

这种情况有可能是比如NIO使用了直接内存(Direct Memory)，这部分内存不在堆中直接管理，只是老年代满了的时候进行full gc才顺带会收集那部分内存，因为32位的程序可以使用的内存是固定的，所以有可能堆上占用了大部分内存，而导致直接内存不足。

鉴于这种情况，在设置堆内存的时候需要考虑留有足够的空间给直接内存使用。

***21. 3 java.net.SocketException：Connection reset***

这种问题是可能由于集成了其他系统，而另外的系统返回时间较长，导致等待的Socket链接越来越多，最终虚拟机承受不了压力后崩溃。

鉴于这种问题，可以使用队列进行异步推送消息。

***22. 这里补充下X进制的算法***

2进制：0，1

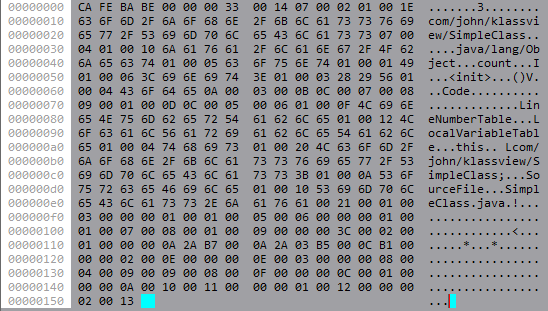
8进制：0，1，2，3，4，5，6，7

10进制：0，1，2，3，4，5，6，7，8，9

16进制：0，1，2，3，4，5，6，7，8，9，A，B，C，D，E，F

所以16进制的0x0032代表的是50，是怎么算的呢？因为已经进位3次了，所以是3\*16+2=50。

***23. class文件构成***



***23.1 jdk版本***

demo:CA FE BA BE 00 00 00 33 00 14 07 00 02 01 00 1E

class文件是一个无符号数的集合，无符号数指的是数字，索引引用，数值，或者字符串值

class文件可以用editplus打开,用hex view编码格式打开

可以看到第一个无符号数是魔数(magic)：0xCAFEBABE。魔数是用来区分文件类型的，像jpg，gif等文件都会有各自的魔数，而java的魔数就是0xCAFEBABE

第二个无符号数是第5和第6个数字，代表java小版本号

第三个无符号数是第7和第8个数字，代表java大版本号

java版本号从jdk1.1开始,十进制是45表示，后面每个大版本就是往后+1，比如说jdk 1.2就是46。这个是第7和第8为无符号数的16进制演算出来的10进制值。

比如jdk 1.2 16进制是00 00 00 2E 转换成10进制就是2\*16+14=46

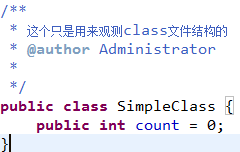
其他无符号数，大致包含:常量池，本类，父类，域，方法等描述信息

***23.2 常量池***

第8，第9位无符号数代表的是constant\_pool\_count，常量个数。不过这个容器的计数不是从0开始计算，而是从1开始计算，比如下面

这边很简单的一个类，值包含了一个int的定义，常量个数就有00 14个，也就是有19个常量，如果再把那个int变量删除，就是00 10只有15个常量。可以得出结论，int占定义占了4个常量个数

java提供了一个工具类分析class文件:javap -verbose SimpleClass,注意这个SimpleClass是.class文件





通过上面的输出得出，class文件的常量池其实就是字面量和符号引用，包括里面的变量，类，方法的描述，仅仅是一些描述，具体要在虚拟机加载这个class文件的时候才知道具体将各个部分分配到那块内存区域。

所以直接看class文件太费劲了，可以看javap分析出来的信息

***23.3 类访问标志***

常量池结束后，紧接着两个字节描述访问标志(access\_flag),例如我们的类是public的，那么这个access\_flag就是ACC\_PUBLIC

还有其他的，比如ACC\_FINAL,ACC\_SUPER,ACC\_INTERFACE,ACC\_ABSTRACT,ACC\_SYNTHETIC,ACC\_ANNOTATION,ACC\_ENUM

***23.4 类索引，父类索引，接口索引结合等***

比如一个类的索引表示

this\_class(index=2)-->CONSTANT\_Class\_info(index2)-->CONSTANT\_Uft8\_info(length:29,bytes:com/john/test/SimpleClass)

父类索引只有一个，这个是java的规范，只能集成一个父类

接口是集合，因为可以实现多个接口

***24.5 字段表集合***

不像上面的常量池，这里面包含了字段的变量名，修饰符等。

***24.6 方法表集合***

也是跟类和字段一样，有修饰符，字面量。

只不过方法体不在这里面。方法经过编译器处理会，方法体编程字节码存储在Code属性内

***25. 操作数***

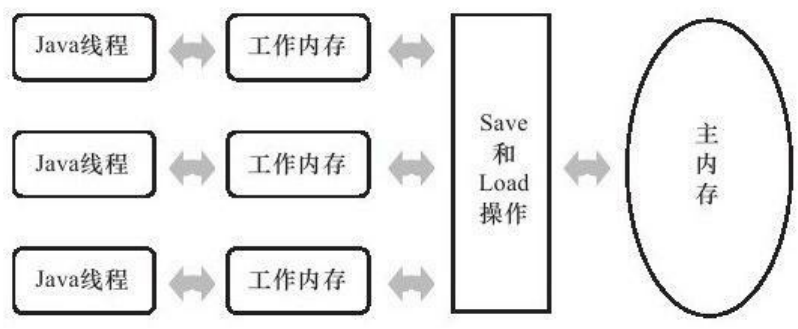
操作某个方法指令所需的参数，称之为操作数。

***26. TPS概念***

Transactions Per Second,服务端每秒处理事务数

***27. 主内存和工作内存***

每条线程都有自己的工作内存，主内存存储所有的的变量。java内存模型规定了，所有对变量的操作都必须在工作内存中进行，而不能直接在主内存中操作。



***28. 内存的操作***

**lock(锁定)**:作用于主内存的变量，它把一个变量标识为一条线程独占的状态。

**unlock(解锁)**:作用于主内存的变量，将一个变量从锁定状态中解锁出来。

**read(读取)**:作用于主内存的变量，将一个变量从主内存读取到工作内存当中。

**load(载入)**:作用于工作内存的变量，将read操作得到的变量值放入工作内存的变量副本中。

**use(使用)**:作用于工作内存的变量，将工作内存的一个变量值传给执行引擎操作。

**assign(赋值)**:作用于工作内存的变量，将从执行引擎接受到的值赋给工作内存的变量。

**store(存储)**:作用于工作内存的变量，将工作内存中的一个变量传到主内存中，以便后续的write操作。

**write(写入)**:作用于主内存的变量，将store传过来的变量值写入到主内存当中。

***29. volatile***

volatile关键字可以让所有线程能看到变量修改后最新的值，但是这并不能适用于高并发。因为volatile仅仅能让线程都可以拿到最新的值，但是线程并发修改volatile这个值放回去时，仍然有可能会产生值的覆盖。

volatile的另一个特性是禁止指令重排序优化；普通变量仅仅能保证所有依赖的地方都能拿到正确的结果，但是并不能保证变量赋值顺序跟程序代码顺序保持一致。

volatile变量的读操作性能消耗跟一般的变量差别不大，但是写操作会慢一些，因为他需要在本地代码中插入内存屏障指令保证处理器的执行顺序不会乱。