一个优秀[**Java**](http://lib.csdn.net/base/javaee)程序员，必须了解Java内存模型、GC工作原理，以及如何优化GC的性能、与GC进行有限的交互，有一些应用程序对性能要求较高，例如嵌入式系统、实时系统等，只有全面提升内存的管理效率，才能提高整个应用程序的性能。

本文将从JVM内存模型、GC工作原理，以及GC的几个关键问题进行探讨，从GC角度提高Java程序的性能。

# 一、Java内存模型

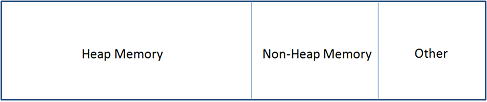
按照官方的说法：Java 虚拟机具有一个堆，堆是运行时数据区域，所有类实例和数组的内存均从此处分配。

JVM主要管理两种类型内存：堆和非堆，堆内存（Heap Memory）是在 Java 虚拟机启动时创建，非堆内存(Non-heap Memory)是在JVM堆之外的内存。

简单来说，堆是Java代码可及的内存，留给开发人员使用的；非堆是JVM留给自己用的，包含方法区、JVM内部处理或优化所需的内存（如 [JIT](http://baike.baidu.com/view/112978.htm#10)Compiler，Just-in-time Compiler，即时编译后的代码缓存）、每个类结构（如运行时常数池、字段和方法数据）以及方法和构造方法的代码。

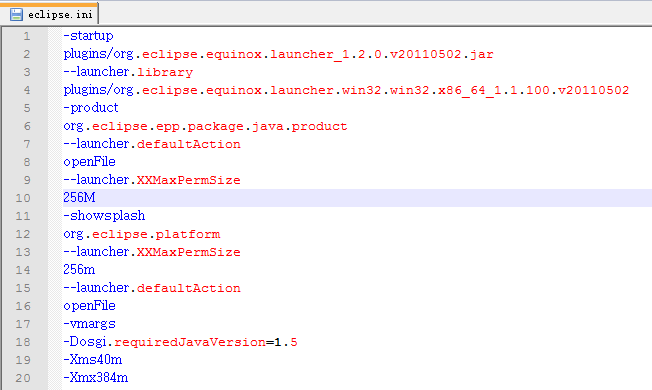
**JVM 内存包含如下几个部分：**

* 堆内存（Heap Memory）： 存放Java对象
* 非堆内存（Non-Heap Memory）： 存放类加载信息和其它meta-data
* 其它（Other）： 存放JVM 自身代码等



在JVM启动时，就已经保留了固定的内存空间给Heap内存，这部分内存并不一定都会被JVM使用，但是可以确定的是这部分保留的内存不会被其他进程使用，这部分内存大小由-Xmx 参数指定。而另一部分内存在JVM启动时就分配给JVM，作为JVM的初始Heap内存使用，这部分内存是由 -Xms 参数指定。

详细配置文件目录：eclipse/eclipse.ini



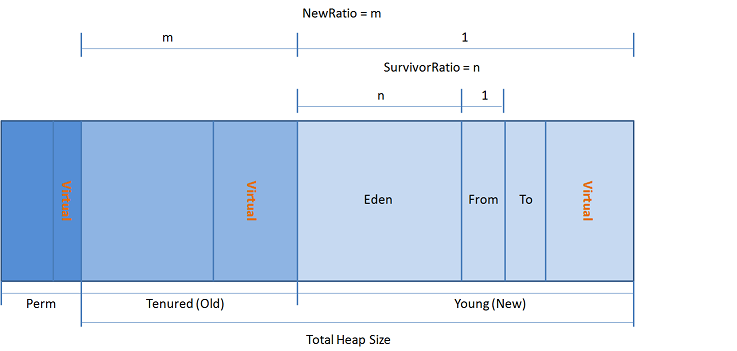
默认空余堆内存小于40%时，JVM 就会增大堆直到-Xmx 的最大限制，可以由 -XX:MinHeapFreeRatio 指定。

默认空余堆内存大于70%时，JVM 会减少堆直到-Xms的最小限制，可以由 -XX:MaxHeapFreeRatio 指定，[详见](http://hi.baidu.com/sunboy_2050/blog/item/b94352ec22e520dbb3fb951b.html)

可以通过 -XX:MaxPermSize 设置Non-Heap大小，详细参见我的[百度博客](http://hi.baidu.com/sunboy_2050/blog/item/b7ec14328ca6aeb9a8018e1d.html?timeStamp=1336206805693)

# 二、Java内存分配

Java的内存管理实际上就是变量和对象的管理，其中包括对象的分配和释放。



JVM内存申请过程如下：

1. JVM 会试图为相关Java对象在Eden中初始化一块内存区域
2. 当Eden空间足够时，内存申请结束；否则到下一步
3. JVM 试图释放在Eden中所有不活跃的对象（这属于1或更高级的垃圾回收）,释放后若Eden空间仍然不足以放入新对象，则试图将部分Eden中活跃对象放入Survivor区
4. Survivor区被用来作为Eden及OLD的中间交换区域，当OLD区空间足够时，Survivor区的对象会被移到Old区，否则会被保留在Survivor区
5. 当OLD区空间不够时，JVM 会在OLD区进行完全的垃圾收集（0级）
6. 完全垃圾收集后，若Survivor及OLD区仍然无法存放从Eden复制过来的部分对象，导致JVM无法在Eden区为新对象创建内存区域，则出现”out of memory”错误

# 三、GC基本原理

GC（Garbage Collection)，是JAVA/.NET中的垃圾收集器。

Java是由C++发展来的，它摈弃了C++中一些繁琐容易出错的东西，引入了计数器的概念，其中有一条就是这个GC机制（C#借鉴了JAVA）

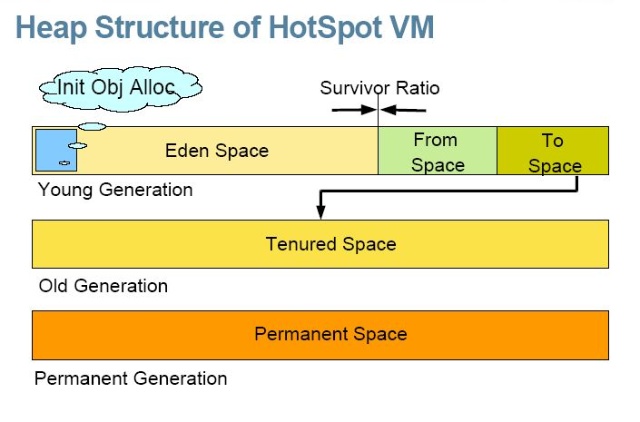
编程人员容易出现问题的地方，忘记或者错误的内存回收会导致程序或系统的不稳定甚至崩溃，Java提供的GC功能可以自动监测对象是否超过作用域从而达到自动回收内存的目的，Java语言没有提供释放已分配内存的显示操作方法。所以，Java的内存管理实际上就是对象的管理，其中包括对象的分配和释放。

对于程序员来说，分配对象使用new关键字；释放对象时，只要将对象所有引用赋值为null，让程序不能够再访问到这个对象，我们称该对象为"不可达的".GC将负责回收所有"不可达"对象的内存空间。

对于GC来说，当程序员创建对象时，GC就开始监控这个对象的地址、大小以及使用情况。通常，GC采用有向图的方式记录和管理堆（heap）中的所有对象。通过这种方式确定哪些对象是"可达的"，哪些对象是"不可达的".当GC确定一些对象为"不可达"时，GC就有责任回收这些内存空间。但是，为了保证 GC能够在不同平台实现的问题，Java规范对GC的很多行为都没有进行严格的规定。例如，对于采用什么类型的回收[**算法**](http://lib.csdn.net/base/datastructure)、什么时候进行回收等重要问题都没有明确的规定。因此，不同的JVM的实现者往往有不同的实现算法。这也给Java程序员的开发带来行多不确定性。本文研究了几个与GC工作相关的问题，努力减少这种不确定性给Java程序带来的负面影响。

# 四、GC分代划分

JVM内存模型中Heap区分两大块，一块是 Young Generation，另一块是Old Generation



1） 在Young Generation中，有一个叫Eden Space的空间，主要是用来存放新生的对象，还有两个Survivor Spaces（from、to），它们的大小总是一样，它们用来存放每次垃圾回收后存活下来的对象。

2） 在Old Generation中，主要存放应用程序中生命周期长的内存对象。

3） 在Young Generation块中，垃圾回收一般用Copying的算法，速度快。每次GC的时候，存活下来的对象首先由Eden拷贝到某个SurvivorSpace，当Survivor Space空间满了后，剩下的live对象就被直接拷贝到OldGeneration中去。因此，每次GC后，Eden内存块会被清空。

4） 在Old Generation块中，垃圾回收一般用mark-compact的算法，速度慢些，但减少内存要求。

5） 垃圾回收分多级，0级为全部(Full)的垃圾回收，会回收OLD段中的垃圾；1级或以上为部分垃圾回收，只会回收Young中的垃圾，内存溢出通常发生于OLD段或Perm段垃圾回收后，仍然无内存空间容纳新的Java对象的情况。

# 五、增量式GC

增量式GC（Incremental GC），是GC在JVM中通常是由一个或一组进程来实现的，它本身也和用户程序一样占用heap空间，运行时也占用CPU。

当GC进程运行时，应用程序停止运行。因此，当GC运行时间较长时，用户能够感到Java程序的停顿，另外一方面，如果GC运行时间太短，则可能对象回收率太低，这意味着还有很多应该回收的对象没有被回收，仍然占用大量内存。因此，在设计GC的时候，就必须在停顿时间和回收率之间进行权衡。一个好的GC实现允许用户定义自己所需要的设置，例如有些内存有限的设备，对内存的使用量非常敏感，希望GC能够准确的回收内存，它并不在意程序速度的快慢。另外一些实时网络游戏，就不能够允许程序有长时间的中断。

增量式GC就是通过一定的回收算法，把一个长时间的中断，划分为很多个小的中断，通过这种方式减少GC对用户程序的影响。虽然，增量式GC在整体性能上可能不如普通GC的效率高，但是它能够减少程序的最长停顿时间。

Sun JDK提供的HotSpot JVM就能支持增量式GC。HotSpot JVM缺省GC方式为不使用增量GC，为了启动增量GC，我们必须在运行Java程序时增加-Xincgc的参数。

HotSpot JVM增量式GC的实现是采用Train GC算法，它的基本想法就是：将堆中的所有对象按照创建和使用情况进行分组（分层），将使用频繁高和具有相关性的对象放在一队中，随着程序的运行，不断对组进行调整。当GC运行时，它总是先回收最老的（最近很少访问的）的对象，如果整组都为可回收对象，GC将整组回收。这样，每次GC运行只回收一定比例的不可达对象，保证程序的顺畅运行。

# 六、详解函数finalize

finalize 是位于Object类的一个方法，详见我的开源项目：[src-jdk1.7.0\_02](http://code.google.com/p/jdk-java/source/browse/trunk/src-jdk1.7.0_02/java/lang/Object.java)

protected void finalize() throws Throwable { }

该方法的访问修饰符为protected，由于所有类为Object的子类，因此用户类很容易访问到这个方法。

由于，finalize函数没有自动实现链式调用，我们必须手动的实现，因此finalize函数的最后一个语句通常是 super.finalize（）。通过这种方式，我们可以实现从下到上实现finalize的调用，即先释放自己的资源，然后再释放父类的资源。根据Java语言规范，JVM保证调用finalize函数之前，这个对象是不可达的，但是JVM不保证这个函数一定会被调用。另外，规范还保证finalize函数最多运行一次。

很多Java初学者会认为这个方法类似与C++中的析构函数，将很多对象、资源的释放都放在这一函数里面。其实，这不是一种很好的方式，原因有三：

其一、GC为了能够支持finalize函数，要对覆盖这个函数的对象作很多附加的工作。

其二、在finalize运行完成之后，该对象可能变成可达的，GC还要再检查一次该对象是否是可达的。因此，使用 finalize会降低GC的运行性能。

其三、由于GC调用finalize的时间是不确定的，因此通过这种方式释放资源也是不确定的。

通常，finalize用于一些不容易控制、并且非常重要资源的释放，例如一些I/O的操作，数据的连接。这些资源的释放对整个应用程序是非常关键的。在这种情况下，程序员应该以通过程序本身管理（包括释放）这些资源为主，以finalize函数释放资源方式为辅，形成一种双保险的管理机制，而不应该仅仅依靠finalize来释放资源。

下面给出一个例子说明，finalize函数被调用以后，仍然可能是可达的，同时也可说明一个对象的finalize只可能运行一次。

**[java]** [view plain](http://blog.csdn.net/ithomer/article/details/6252552) [copy](http://blog.csdn.net/ithomer/article/details/6252552)

[print?](http://blog.csdn.net/ithomer/article/details/6252552)

1. **class** MyObject {
2. Test main;      // 记录Test对象，在finalize中时用于恢复可达性
4. **public** MyObject(Test t) {
5. main = t;   // 保存Test 对象
6. }
8. **protected** **void** finalize() {
9. main.ref = **this**;    // 恢复本对象，让本对象可达
10. System.out.println("This is finalize");     // 用于测试finalize只运行一次
11. }
12. }
14. **class** Test {
15. MyObject ref;
17. **public** **static** **void** main(String[] args) {
18. Test test = **new** Test();
19. test.ref = **new** MyObject(test);
20. test.ref = **null**;    // MyObject对象为不可达对象，finalize将被调用
21. System.gc();
22. **if** (test.ref != **null**)
23. System.out.println("My Object还活着");
24. }
25. }

**运行结果：**

　　This is finalize

　　My Object还活着

此例子中需要注意，虽然MyObject对象在finalize中变成可达对象，但是下次回收时候，finalize却不再被调用，因为finalize函数最多只调用一次。

# 七、GC程序交互

程序如何与GC进行交互呢？ Java2增强了内存管理功能，增加了一个java.lang.ref包，详见我的开源项目：[src-jdk1.7.0\_02](http://code.google.com/p/jdk-java/source/browse/trunk/#trunk%2Fsrc-jdk1.7.0_02%2Fjava%2Flang%2Fref)

其中定义了三种引用类。这三种引用类分别为：SoftReference、 WeakReference、 PhantomReference

通过使用这些引用类，程序员可以在一定程度与GC进行交互，以便改善GC的工作效率，这些引用类的引用强度介于可达对象和不可达对象之间。

创建一个引用对象也非常容易，例如：如果你需要创建一个Soft Reference对象，那么首先创建一个对象，并采用普通引用方式（可达对象）；然后再创建一个SoftReference引用该对象；最后将普通引用设置为null。通过这种方式，这个对象就只有一个Soft Reference引用。同时，我们称这个对象为Soft Reference 对象。

Soft Reference的主要特点是据有较强的引用功能。只有当内存不够的时候，才进行回收这类内存，因此在内存足够的时候，它们通常不被回收。另外，这些引用对象还能保证在Java抛出OutOfMemory 异常之前，被设置为null。它可以用于实现一些常用图片的缓存，实现Cache的功能，保证最大限度的使用内存而不引起OutOfMemory。以下给出这种引用类型的使用伪代码：

**[java]** [view plain](http://blog.csdn.net/ithomer/article/details/6252552) [copy](http://blog.csdn.net/ithomer/article/details/6252552)

[print?](http://blog.csdn.net/ithomer/article/details/6252552)

1. // 申请一个图像对象
2. Image image=**new** Image();       // 创建Image对象
3. …
4. // 使用 image
5. …
6. // 使用完了image，将它设置为soft 引用类型，并且释放强引用；
7. SoftReference sr=**new** SoftReference(image);
8. image=**null**;
9. …
10. // 下次使用时
11. **if** (sr!=**null**)
12. image=sr.get();
13. **else**{
14. image=**new** Image();  //由于GC由于低内存，已释放image，因此需要重新装载；
15. sr=**new** SoftReference(image);
16. }

Weak引用对象与Soft引用对象的最大不同就在于：GC在进行回收时，需要通过算法检查是否回收Soft引用对象，而对于Weak引用对象，GC总是进行回收。Weak引用对象更容易、更快被GC回收。虽然，GC在运行时一定回收Weak对象，但是复杂关系的Weak对象群常常需要好几次GC的运行才能完成。Weak引用对象常常用于Map结构中，引用数据量较大的对象，一旦该对象的强引用为null时，GC能够快速地回收该对象空间。

Phantom引用的用途较少，主要用于辅助finalize函数的使用。Phantom对象指一些对象，它们执行完了finalize函数，并为不可达对象，但是它们还没有被GC回收。这种对象可以辅助finalize进行一些后期的回收工作，我们通过覆盖Reference的clear()方法，增强资源回收机制的灵活性。

# 八、Java编程建议

根据GC的工作原理，我们可以通过一些技巧和方式，让GC运行更加有效率，更加符合应用程序的要求。一些关于程序设计的几点建议：

1）最基本的建议就是尽早释放无用对象的引用。大多数程序员在使用临时变量的时候，都是让引用变量在退出活动域（scope）后，自动设置为 null.我们在使用这种方式时候，必须特别注意一些复杂的对象图，例如数组，队列，树，图等，这些对象之间有相互引用关系较为复杂。对于这类对象，GC 回收它们一般效率较低。如果程序允许，尽早将不用的引用对象赋为null，这样可以加速GC的工作。

2）尽量少用finalize函数。finalize函数是Java提供给程序员一个释放对象或资源的机会。但是，它会加大GC的工作量，因此尽量少采用finalize方式回收资源。

3）如果需要使用经常使用的图片，可以使用soft应用类型。它可以尽可能将图片保存在内存中，供程序调用，而不引起OutOfMemory.

4）注意集合数据类型，包括数组，树，图，链表等[**数据结构**](http://lib.csdn.net/base/datastructure)，这些数据结构对GC来说，回收更为复杂。另外，注意一些全局的变量，以及一些静态变量。这些变量往往容易引起悬挂对象（dangling reference），造成内存浪费。

5）当程序有一定的等待时间，程序员可以手动执行System.gc()，通知GC运行，但是Java语言规范并不保证GC一定会执行。使用增量式GC可以缩短Java程序的暂停时间。

**参考推荐：**

[Java内存模型及GC原理](http://blog.sina.com.cn/s/blog_4e90b3ba0100m5cb.html)

[一个优秀的Java程序员必须了解的GC机制](http://inotgaoshou.iteye.com/blog/757470)

[**Android**](http://lib.csdn.net/base/android) 智能指针原理（推荐）

[Java虚拟机规范](http://www.189works.com/article-42350-1.html)

[Java虚拟机参数](http://hi.baidu.com/qust409xls/blog/item/28dd9e26605e6213908f9d0b.html)

[Java内存模型](http://www.cnblogs.com/tiechui/archive/2010/12/07/1898774.html)

[Java系列教程](http://blog.csdn.net/silentbalanceyh)（推荐）

[Java垃圾回收原理](http://www.360doc.com/content/11/0911/15/18042_147476260.shtml)（360doc）

[Java内存模型及GC原理](http://blog.csdn.net/zeo112140/article/details/7225277)（图解）

[Java的内存结构和垃圾收集](http://blog.csdn.net/autofei/article/details/7456213)（图解）

[JDK5.0中JVM堆模型、GC垃圾收集详细解析](http://blog.csdn.net/sfdev/article/details/4483442)（图解）

[Java内存泄露的理解与解决](http://hi.baidu.com/sunboy_2050/blog/item/81c381800ccfe48cf703a6a6.html)

[Java gc的调用机制和编程规则](http://blog.csdn.net/remote_roamer/article/details/5539975)

[Java 内存泄漏实例及解决方案研究](http://blog.csdn.net/bulinner/article/details/715752)

[JVM 优点与缺点的深入分析](http://news.csdn.net/a/20091222/215973.html) [草稿]

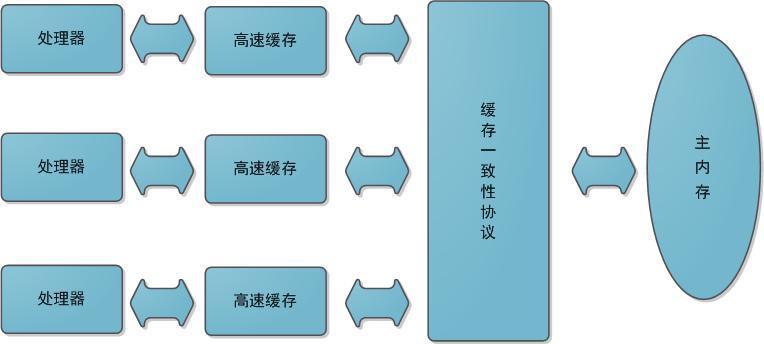
# 九，其他说法

## 1. 概述

　　多任务和高并发是衡量一台计算机处理器的能力重要指标之一。一般衡量一个服务器性能的高低好坏，使用每秒事务处理数（Transactions Per Second，TPS）这个指标比较能说明问题，它代表着一秒内服务器平均能响应的请求数，而TPS值与程序的并发能力有着非常密切的关系。在讨论Java内存模型和线程之前，先简单介绍一下硬件的效率与一致性。

## 2.硬件的效率与一致性

　　由于计算机的存储设备与处理器的运算能力之间有几个数量级的差距，所以现代计算机系统都不得不加入一层读写速度尽可能接近处理器运算速度的高速缓存（cache）来作为内存与处理器之间的缓冲：将运算需要使用到的数据复制到缓存中，让运算能快速进行，当运算结束后再从缓存同步回内存之中没这样处理器就无需等待缓慢的内存读写了。  
　　基于高速缓存的存储交互很好地解决了处理器与内存的速度矛盾，但是引入了一个新的问题：缓存一致性（Cache Coherence）。在多处理器系统中，每个处理器都有自己的高速缓存，而他们又共享同一主存，如下图所示：多个处理器运算任务都涉及同一块主存，需要一种协议可以保障数据的一致性，这类协议有MSI、MESI、MOSI及Dragon Protocol等。Java虚拟机内存模型中定义的内存访问操作与硬件的缓存访问操作是具有可比性的，后续将介绍Java内存模型。



　　除此之外，为了使得处理器内部的运算单元能竟可能被充分利用，处理器可能会对输入代码进行乱起执行（Out-Of-Order Execution）优化，处理器会在计算之后将对乱序执行的代码进行结果重组，保证结果准确性。与处理器的乱序执行优化类似，Java虚拟机的即时编译器中也有类似的指令重排序（Instruction Recorder）优化。

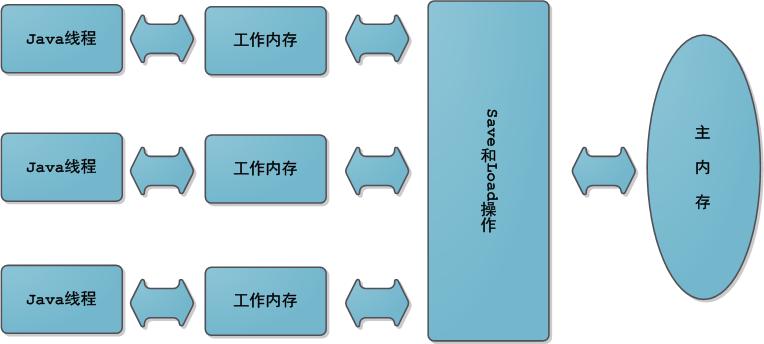
## 3.Java内存模型

　　定义Java内存模型并不是一件容易的事情，这个模型必须定义得足够严谨，才能让Java的并发操作不会产生歧义；但是，也必须得足够宽松，使得虚拟机的实现能有足够的自由空间去利用硬件的各种特性（寄存器、高速缓存等）来获取更好的执行速度。经过长时间的验证和修补，在JDK1.5发布后，Java内存模型就已经成熟和完善起来了。

### 3.1 主内存与工作内存

　　Java内存模型的主要目标是定义程序中各个变量的访问规则，即在虚拟机中将变量存储到内存和从内存中取出变量这样底层细节。此处的变量与Java编程时所说的变量不一样，指包括了实例字段、静态字段和构成数组对象的元素，但是不包括局部变量与方法参数，后者是线程私有的，不会被共享。

　　Java内存模型中规定了所有的变量都存储在主内存中，每条线程还有自己的工作内存（可以与前面将的处理器的高速缓存类比），线程的工作内存中保存了该线程使用到的变量到主内存副本拷贝，线程对变量的所有操作（读取、赋值）都必须在工作内存中进行，而不能直接读写主内存中的变量。不同线程之间无法直接访问对方工作内存中的变量，线程间变量值的传递均需要在主内存来完成，线程、主内存和工作内存的交互关系如下图所示，和上图很类似。



这里的主内存、工作内存与Java内存区域的Java堆、栈、方法区不是同一层次内存划分。

### 3.2 内存间交互操作

　　关于主内存与工作内存之间的具体交互协议，即一个变量如何从主内存拷贝到工作内存、如何从工作内存同步到主内存之间的实现细节，Java内存模型定义了以下八种操作来完成：

* lock（锁定）：作用于主内存的变量，把一个变量标识为一条线程独占状态。
* unlock（解锁）：作用于主内存变量，把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定。
* read（读取）：作用于主内存变量，把一个变量值从主内存传输到线程的工作内存中，以便随后的load动作使用
* load（载入）：作用于工作内存的变量，它把read操作从主内存中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，把工作内存中的一个变量值传递给执行引擎，每当虚拟机遇到一个需要使用变量的值的字节码指令时将会执行这个操作。
* assign（赋值）：作用于工作内存的变量，它把一个从执行引擎接收到的值赋值给工作内存的变量，每当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作。
* store（存储）：作用于工作内存的变量，把工作内存中的一个变量的值传送到主内存中，以便随后的write的操作。
* write（写入）：作用于主内存的变量，它把store操作从工作内存中一个变量的值传送到主内存的变量中。

　　如果要把一个变量从主内存中复制到工作内存，就需要按顺寻地执行read和load操作，如果把变量从工作内存中同步回主内存中，就要按顺序地执行store和write操作。Java内存模型只要求上述操作必须按顺序执行，而没有保证必须是连续执行。也就是read和load之间，store和write之间是可以插入其他指令的，如对主内存中的变量a、b进行访问时，可能的顺序是read a，read b，load b， load a。Java内存模型还规定了在执行上述八种基本操作时，必须满足如下规则：

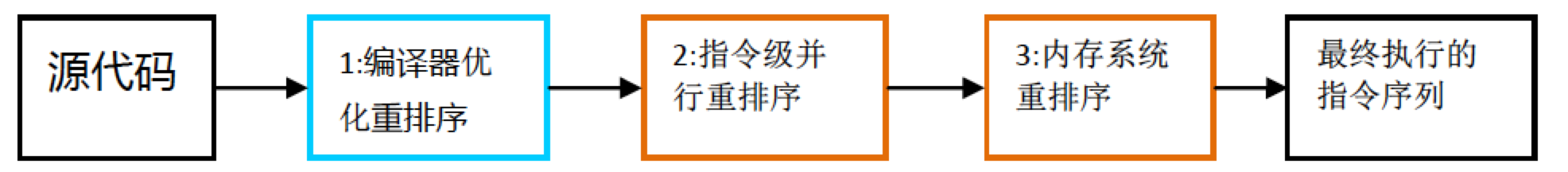
* 不允许read和load、store和write操作之一单独出现
* 不允许一个线程丢弃它的最近assign的操作，即变量在工作内存中改变了之后必须同步到主内存中。
* 不允许一个线程无原因地（没有发生过任何assign操作）把数据从工作内存同步回主内存中。
* 一个新的变量只能在主内存中诞生，不允许在工作内存中直接使用一个未被初始化（load或assign）的变量。即就是对一个变量实施use和store操作之前，必须先执行过了assign和load操作。
* 一个变量在同一时刻只允许一条线程对其进行lock操作，lock和unlock必须成对出现
* 如果对一个变量执行lock操作，将会清空工作内存中此变量的值，在执行引擎使用这个变量前需要重新执行load或assign操作初始化变量的值
* 如果一个变量事先没有被lock操作锁定，则不允许对它执行unlock操作；也不允许去unlock一个被其他线程锁定的变量。
* 对一个变量执行unlock操作之前，必须先把此变量同步到主内存中（执行store和write操作）。

### 3.3 重排序

　　在执行程序时为了提高性能，编译器和处理器经常会对指令进行重排序。重排序分成三种类型：

1. 编译器优化的重排序。编译器在不改变单线程程序语义放入前提下，可以重新安排语句的执行顺序。
2. 指令级并行的重排序。现代处理器采用了指令级并行技术来将多条指令重叠执行。如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。
3. 内存系统的重排序。由于处理器使用缓存和读写缓冲区，这使得加载和存储操作看上去可能是在乱序执行。

从Java源代码到最终实际执行的指令序列，会经过下面三种重排序：



为了保证内存的可见性，Java编译器在生成指令序列的适当位置会插入内存屏障指令来禁止特定类型的处理器重排序。Java内存模型把内存屏障分为LoadLoad、LoadStore、StoreLoad和StoreStore四种：



### 3.4 同步机制

介绍volatile、synchronized和final

### 3.5 原子性、可见性与有序性

介绍三个特性