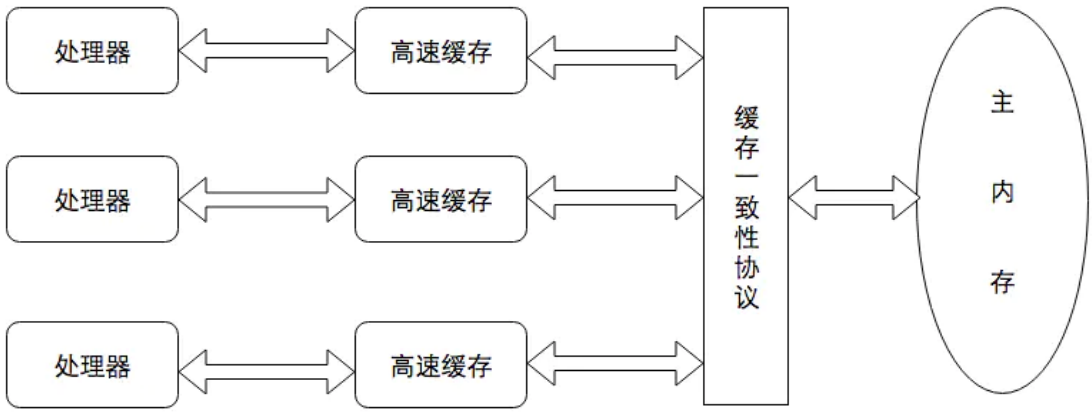
java内存模型(Java Memory Model，JMM)是java虚拟机规范定义的，用来屏蔽掉java程序在各种不同的硬件和操作系统对内存的访问的差异，这样就可以实现java程序在各种不同的平台上都能达到内存访问的一致性。可以避免像c++等直接使用物理硬件和操作系统的内存模型在不同操作系统和硬件平台下表现不同，比如有些c/c++程序可能在windows平台运行正常，而在linux平台却运行有问题。

物理硬件和内存

首先，在单核电脑中，处理问题要简单的多。对内存和硬件的要求，各种方面的考虑没有在多核的情况下复杂。电脑中，CPU的运行计算速度是非常快的，而其他硬件比如IO，网络、内存读取等等，跟cpu的速度比起来是差几个数量级的。而不管任何操作，几乎是不可能都在cpu中完成而不借助于任何其他硬件操作。所以协调cpu和各个硬件之间的速度差异是非常重要的，要不然cpu就一直在等待，浪费资源。而在多核中，不仅面临如上问题，还有如果多个核用到了同一个数据，如何保证数据的一致性、正确性等问题，也是必须要解决的。 目前基于高速缓存的存储交互很好的解决了cpu和内存等其他硬件之间的速度矛盾，多核情况下各个处理器(核)都要遵循一定的诸如MSI、MESI等协议来保证内存的各个处理器高速缓存和主内存的数据的一致性。



处理器、高速缓存、主内存之间遵循一致性协议交互关系

image.png

除了增加高速缓存，为了使处理器内部运算单元尽可能被充分利用，处理器还会对输入的代码进行乱序执行(Out-Of-Order Execution)优化，处理器会在乱序执行之后的结果进行重组，保证结果的正确性，也就是保证结果与顺序执行的结果一致。但是在真正的执行过程中，代码执行的顺序并不一定按照代码的书写顺序来执行，可能和代码的书写顺序不同。

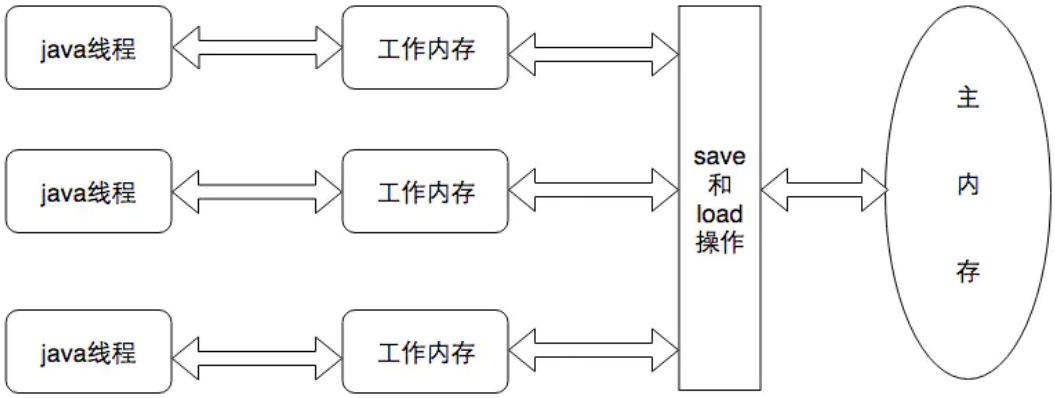
java内存模型

虽然java程序所有的运行都是在虚拟机中，涉及到的内存等信息都是虚拟机的一部分，但实际也是物理机的，只不过是虚拟机作为最外层的容器统一做了处理。虚拟机的内存模型，以及多线程的场景下与物理机的情况是很相似的，可以类比参考。 Java内存模型的主要目标是定义程序中变量的访问规则。即在虚拟机中将变量存储到主内存或者将变量从主内存取出这样的底层细节。需要注意的是这里的变量跟我们写java程序中的变量不是完全等同的。这里的变量是指实例字段，静态字段，构成数组对象的元素，但是不包括局部变量和方法参数(因为这是线程私有的)。这里可以简单的认为主内存是java虚拟机内存

区域中的堆，局部变量和方法参数是在虚拟机栈中定义的。但是在堆中的变量如果在多线程中都使用，就涉及到了堆和不同虚拟机栈中变量的值的一致性问题了。 Java内存模型中涉及到的概念有：

- 主内存：java虚拟机规定所有的变量(不是程序中的变量)都必须在主内存中产生，为了方便理解，可以认为是堆区。可以与前面说的物理机的主内存相比，只不过物理机的主内存是整个机器的内存，而虚拟机的主内存是虚拟机内存中的一部分。
- 工作内存：java虚拟机中每个线程都有自己的工作内存，该内存是线程私有的为了方便理解，可以认为是虚拟机栈。可以与前面说的高速缓存相比。线程的工作内存保存了线程需要的变量在主内存中的副本。虚拟机规定，线程对主内存变量的修改必须在线程的工作内存中进行，不能直接读写主内存中的变量。不同的线程之间也不能相互访问对方的工作内存。如果线程之间需要传递变量的值，必须通过主内存来作为中介进行传递。

  这里需要说明一下：主内存、工作内存与java内存区域中的java堆、虚拟机栈、方法区并不是一个层次的内存划分。这两者是基本上是没有关系的，上文只是为了便于理解，做的类比



线程、工作内存、主内存之间的交互关系

image.png

工作内存与主内存交互

物理机高速缓存和主内存之间的交互有协议，同样的，java内存中线程的工作内存和主内存的交互是由java虚拟机定义了如下的8种操作来完成的，每种操作必须是原子性的(double和long类型在某些平台有例外，参考volatile详解和非原子性协定) java虚拟机中主内存和工作内存交互，就是一个变量如何从主内存传输到工作内存中，如何把修改后的变量从工作内存同步回主内存。

- **lock(锁定)**:作用于主内存的变量，一个变量在同一时间只能一个线程锁定，该操作表示这条线成独占这个变量
- **unlock(解锁)**:作用于主内存的变量，表示这个变量的状态由处于锁定状态被释放，这样其他线程才能对该变量进行锁定
- **read(读取)**:作用于主内存变量，表示把一个主内存变量的值传输到线程的工作内存，以便随后的load操作使用
- **load(载入)**:作用于线程的工作内存的变量，表示把read操作从主内存中读取的变量的值放到工作内存的变量副本中(副本是相对于主内存的变量而言的)
- **use(使用)**:作用于线程的工作内存中的变量，表示把工作内存中的一个变量的值传递给执行引擎，每当虚拟机遇到一个需要使用变量的值的字节码指令时就会执行该操作
- **assign(赋值)**:作用于线程的工作内存的变量，表示把执行引擎返回的结果赋值给工作内存中的变量，每当虚拟机遇到一个给变量赋值的字节码指令时就会执行该操作
- **store(存储)**:作用于线程的工作内存中的变量，把工作内存中的一个变量的值传递给主内存，以便随后的write操作使用
- **write(写入)**:作用于主内存的变量，把store操作从工作内存中得到的变量的值放入主内存的变量中

如果要把一个变量从主内存传输到工作内存，那就要顺序的执行read和load操作，如果要把一个变量从工作内存回写到主内存，就要顺序的执行store和write操作。对于普通变量，虚拟机只是要求顺序的执行，并没有要求连续的执行，所以如下也是正确的。对于两个线程，分别从主内存中读取变量a和b的值，并不一样要read a; load a; read b; load b; 也会出现如下执行顺序：read a; read b; load b; load a; (对于volatile修饰的变量会有一些其他规则,后边会详细列出)，对于这8中操作，虚拟机也规定了一系列规则，在执行这8中操作的时候必须遵循如下的规则：

- **不允许read和load、store和write操作之一单独出现**，也就是不允许从主内存读取了变量的值但是工作内存不接收的情况，或者不允许从工作内存将变量的值回写到主内存但是主内存不接收的情况
- **不允许一个线程丢弃最近的assign操作**，也就是不允许线程在自己的工作线程中修改了变量的值却不同步/回写到主内存
- **不允许一个线程回写没有修改的变量到主内存**，也就是如果线程工作内存中变量没有发生过任何assign操作，是不允许将该变量的值回写到主内存
- **变量只能在主内存中产生**，不允许在工作内存中直接使用一个未被初始化的变量，也就是没有执行load或者assign操作。也就是说在执行use、store之前必须对相同的变量执行了load、assign操作
- **一个变量在同一时刻只能被一个线程对其进行lock操作**，也就是说一个线程一旦对一个变量加锁后，在该线程没有释放掉锁之前，其他线程是不能对其加锁的，但是同一个线程对一个变量加锁后，可以继续加锁，同时在释放锁的时候释放锁次数必须和加锁次数相同。
- **对变量执行lock操作，就会清空工作空间该变量的值**，执行引擎使用这个变量之前，需要重新load或者assign操作初始化变量的值
- **不允许对没有lock的变量执行unlock操作**，如果一个变量没有被lock操作，那也不能对其执行unlock操作，当然一个线程也不能对被其他线程lock的变量执行unlock操作
- **对一个变量执行unlock之前，必须先把变量同步回主内存中**，也就是执行store和write操作

当然，最重要的还是如开始所说，这8个动作必须是原子的，不可分割的。 针对volatile修饰的变量，会有一些特殊规定。

volatile修饰的变量的特殊规则

关键字volatile可以说是java虚拟机中提供的最轻量级的同步机制。java内存模型对volatile专门定义了一些特殊的访问规则。这些规则有些晦涩拗口，先列出规则，然后用更加通俗易懂的语言来解释： 假定T表示一个线程，V和W分别表示两个volatile修饰的变量，那么在进行read、load、use、assign、store和write操作的时候需要满足如下规则：

- **只有当线程T对变量V执行的前一个动作是load，线程T对变量V才能执行use动作；同时只有当线程T对变量V执行的后一个动作是use的时候线程T对变量V才能执行load操作。**所以，线程T对变量V的use动作和线程T对变量V的read、load动作相关联，必须是连续一起出现。也就是在线程T的工作内存中，每次使用变量V之前必须从主内存去重新获取最新的值，用于保证线程T能看得见其他线程对变量V的最新的修改后的值。
- **只有当线程T对变量V执行的前一个动作是assign的时候，线程T对变量V才能执行store动作；同时只有当线程T对变量V执行的后一个动作是store的时候，线程T对变量V才能执行assign动作。**所以，线程T对变量V的assign操作和线程T对变量V的store、write动作相关联，必须一起连续出现。也即是在线程T的工作内存中，每次修改变量V之后必须立刻同步回主内存，用于保证线程T对变量V的修改能立刻被其他线程看到。
- **假定动作A是线程T对变量V实施的use或assign动作，动作F是和动作A相关联的load或store动作，动作P是和动作F相对应的对变量V的read或write动作；类似的，假定动作B是线程T对变量W实施的use或assign动作，动作G是和动作B相关联的load或store动作，动作Q是和动作G相对应的对变量W的read或write动作。如果动作A先于B，那么P先于Q。**也就是说在同一个线程内部，被volatile修饰的变量不会被指令重排序，保证代码的执行顺序和程序的顺序相同。

总结上面三条规则，前面两条可以概括为：**volatile类型的变量保证对所有线程的可见性。**第三条为：**volatile类型的变量禁止指令重排序优化。**

- ***valatile类型的变量保证对所有线程的可见性*** 可见性是指当一个线程修改了这个变量的值，新值（修改后的值）对于其他线程来说是立即可以得知的。正如上面的前两条规则规定，volatile类型的变量每次值被修改了就立即同步回主内存，每次使用时就需要从主内存重新读取值。返回到前面对普通变量的规则中，并没有要求这一点，所以普通变量的值是不会立即对所有线程可见的。误解：volatile变量对所有线程是立即可见的，所以对volatile变量的所有修改(写操作)都立刻能反应到其他线程中。或者换句话说：volatile变量在各个线程中是一致的，所以基于volatile变量的运算在并发下是线程安全的。 这个观点的论据是正确的，但是根据论据得出的结论是错误的，并不能得出这样的结论。volatile的规则，保证了read、load、use的顺序和连续行，同理assign、store、write也是顺序和连续的。也就是这几个动作是原子性的，但是对变量的修改，或者对变量的运算，却不能保证是原子性的。如果对变量的修改是分为多个步骤的，那么多个线程同时从主内存拿到的值是最新的，但是经过多步运算后回写到主内存的值是有可能存在覆盖情况发生的。如下代码的例子：

```
public class VolatileTest {
  public static volatile int race = 0;
  public static void increase() {
    race++
  }

  private static final int THREADS_COUNT = 20;

  public void static main(String[] args) {
      Thread[] threads = new Thread[THREADS_COUNT];
      for (int = 0; i < THREADS_COUNT; i++) {
          threads[i] = new Thread(new Runnable(){
              @Override
              public void run() {
                  for (int j = 0; j < 10000; j++) {
                     increase();
                  }
              }
          });
          threads[i].start();
      }
      while (Thread.activeCount() > 1) {
         Thread.yield();
      }
      System.out.println(race);
  }
}
```

代码就是对volatile类型的变量启动了20个线程，每个线程对变量执行1w次加1操作，如果volatile变量并发操作没有问题的话，那么结果应该是输出20w，但是结果运行的时候每次都是小于20w，这就是因为`race++`操作不是原子性的，是分多个步骤完成的。假设两个线程a、b同时取到了主内存的值，是0，这是没有问题的，在进行`++`操作的时候假设线程a执行到一半，线程b执行完了，这时线程b立即同步给了主内存，主内存的值为1，而线程a此时也执行完了，同步给了主内存，此时的值仍然是1，线程b的结果被覆盖掉了。

- ***volatile变量禁止指令重排序优化*** 普通的变量仅仅会保证在该方法执行的过程中，所有依赖赋值结果的地方都能获取到正确的结果，但不能保证变量赋值的操作顺序和程序代码的顺序一致。因为在一个线程的方法执行过程中无法感知到这一点，这也就是java内存模型中描述的所谓的“线程内部表现为串行的语义”。 也就是在单线程内部，我们看到的或者感知到的结果和代码顺序是一致的，即使代码的执行顺序和代码顺序不一致，但是在需要赋值的时候结果也是正确的，所以看起来

就是串行的。但实际结果有可能代码的执行顺序和代码顺序是不一致的。这在多线程中就会出现问题。 看下面的伪代码举例：

```
Map configOptions;
char[] configText;
//volatile类型bianliang
volatile boolean initialized = false;

//假设以下代码在线程A中执行
//模拟读取配置信息，读取完成后认为是初始化完成
configOptions = new HashMap();
configText = readConfigFile(fileName);
processConfigOptions(configText, configOptions);
initialized = true;

//假设以下代码在线程B中执行
//等待initialized为true后，读取配置信息进行操作
while ( !initialized) {
  sleep();
}
doSomethingWithConfig();
```

如果initialiezd是普通变量，没有被volatile修饰，那么线程A执行的代码的修改初始化完成的结果`initialized = true`就有可能先于之前的三行代码执行，而此时线程B发现initialized为true了，就执行`doSomethingWithConfig()`方法，但是里面的配置信息都是null的，就会出现问题了。 现在initialized是volatile类型变量，保证禁止代码重排序优化，那么就可以保证`initialized = true`执行的时候，前边的三行代码一定执行完成了，那么线程B读取的配置文件信息就是正确的。

跟其他保证并发安全的工具相比，volatile的性能确实会好一些。在某些情况下，volatile的同步机制性能要优于锁(使用synchronized关键字或者java.util.concurrent包中的锁)。但是现在由于虚拟机对锁的不断优化和实行的许多消除动作，很难有一个量化的比较。 与自己相比，就可以确定一个原则：volatile变量的读操作和普通变量的读操作几乎没有差异，但是写操作会性能差一些，慢一些，因为要在本地代码中插入许多内存屏障指令来禁止指令重排序，保证处理器不发生代码乱序执行行为。

long和double变量的特殊规则

Java内存模型要求对主内存和工作内存交换的八个动作是原子的，正如章节开头所讲，对long和double有一些特殊规则。八个动作中lock、unlock、read、load、use、assign、store、write对待32位的基本数据类型都是原子操作，对待long和double这两个64位的数据，java虚拟机规范对java内存模型的规定中特别定义了一条相对宽松的规则：允许虚拟机将没有被volatile修饰的64位数据的读写操作划分为两次32位的操作来进行，也就是允许虚拟机不保证对64位数据的read、load、store和write这4个动作的操作是原子的。这也就是我们常说的long和double的非原子性协定(Nonautomic Treatment of double and long Variables)。

并发内存模型的实质

Java内存模型围绕着并发过程中如何处理原子性、可见性和顺序性这三个特征来设计的。

原子性(Automicity)

由Java内存模型来直接保证原子性的变量操作包括read、load、use、assign、store、write这6个动作，虽然存在long和double的特例，但基本可以忽律不计，目前虚拟机基本都对其实现了原子性。如果需要更大范围的控制，lock和unlock也可以满足需求。lock和unlock虽然没有被虚拟机直接开给用户使用，但是提供了字节码层次的指令monitorenter和monitorexit对应这两个操作，对应到java代码就是synchronized关键字，因此在synchronized块之间的代码都具有原子性。

可见性

可见性是指一个线程修改了一个变量的值后，其他线程立即可以感知到这个值的修改。正如前面所说，volatile类型的变量在修改后会立即同步给主内存，在使用的时候会从主内存重新读取，是依赖主内存为中介来保证多线程下变量对其他线程的可见性的。 除了volatile，synchronized和final也可以实现可见性。synchronized关键字是通过unlock之前必须把变量同步回主内存来实现的，final则是在初始化后就不会更改，所以只要在初始化过程中没有把this指针传递出去也能保证对其他线程的可见性。

有序性

有序性从不同的角度来看是不同的。单纯单线程来看都是有序的，但到了多线程就会跟我们预想的不一样。可以这么说：如果在本线程内部观察，所有操作都是有序的；如果在一个线程中观察另一个线程，所有的操作都是无序的。前半句说的就是“线程内表现为串行的语义”，后半句值得是“指令重排序”现象和主内存与工作内存之间同步存在延迟的现象。 保证有序性的关键字有volatile和synchronized，volatile禁止了指令重排序，而synchronized则由“一个变量在同一时刻只能被一个线程对其进行lock操作”来保证。

总体来看，synchronized对三种特性都有支持，虽然简单，但是如果无控制的滥用对性能就会产生较大影响。

先行发生原则

如果Java内存模型中所有的有序性都要依靠volatile和synchronized来实现，那是不是非常繁琐。Java语言中有一个“先行发生原则”，是判断数据是否存在竞争、线程是否安全的主要依据。

什么是先行发生原则

先行发生原则是Java内存模型中定义的两个操作之间的偏序关系。比如说操作A先行发生于操作B，那么在B操作发生之前，A操作产生的“影响”都会被操作B感知到。这里的影响是指修改了内存中的共享变量、发送了消息、调用了方法等。个人觉得更直白一些就是有可能对操作B的结果有影响的都会被B感知到，对B操作的结果没有影响的是否感知到没有太大关系。

Java内存模型自带先行发生原则有哪些

- 程序次序原则 在一个线程内部，按照代码的顺序，书写在前面的先行发生与后边的。或者更准确的说是在控制流顺序前面的先行发生与控制流后面的，而不是代码顺序，因为会有分支、跳转、循环等。
- 管程锁定规则 一个unlock操作先行发生于后面对同一个锁的lock操作。这里必须注意的是对同一个锁，后面是指时间上的后面
- volatile变量规则 对一个volatile变量的写操作先行发生与后面对这个变量的读操作，这里的后面是指时间上的先后顺序
- 线程启动规则 Thread对象的start()方法先行发生与该线程的每个动作。当然如果你错误的使用了线程，创建线程后没有执行start方法，而是执行run方法，那此句话是不成立的，但是如果这样其实也不是线程了
- 线程终止规则 线程中的所有操作都先行发生与对此线程的终止检测，可以通过Thread.join()和Thread.isAlive()的返回值等手段检测线程是否已经终止执行
- 线程中断规则 对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生，可以通过Thread.interrupted()方法检测到是否有中断发生。

- 对象终结规则 一个对象的初始化完成先行发生于他的finalize方法的执行，也就是初始化方法先行发生于finalize方法
- 传递性 如果操作A先行发生于操作B，操作B先行发生于操作C，那么操作A先行发生于操作C。

看一个例子:

```
private int value = 0;
public void setValue(int value) {
  this.value = value;
}
public int getValue() {
  return this.value;
}
```

如果有两个线程A和B，A先调用setValue方法，然后B调用getValue方法，那么B线程执行方法返回的结果是什么？ 我们去对照先行发生原则一个一个对比。首先是**程序次序规则**，这里是多线程，不在一个线程中，不适用；然后是**管程锁定规则**，这里没有synchronized，自然不会发生lock和unlock，不适用；后面对于**线程启动规则**、**线程终止规则**、**线程中断规则**也不适用，这里与**对象终结规则**、**传递性规则**也没有关系。所以说B返回的结果是不确定的，也就是说在多线程环境下该操作不是线程安全的。 如何修改呢，一个是对get/set方法加入synchronized 关键字，可以使用**管程锁定规则**；要么对value加volatile修饰，可以使用**volatile变量规则**。 通过上面的例子可知，一个操作时间上先发生并不代表这个操作先行发生，那么一个操作先行发生是不是代表这个操作在时间上先发生？也不是，如下面的例子:

```
int i = 2;
int j = 1;
```

在同一个线程内，对i的赋值先行发生于对j赋值的操作，但是代码重排序优化，也有可能是j的赋值先发生，我们无法感知到这一变化。

所以，综上所述，时间先后顺序与先行发生原则之间基本没有太大关系。我们衡量并发安全的问题的时候不要受到时间先后顺序的干扰，一切以先行发生原则为准。