JUC 笔记（一）

作者：Gao-Eason

原文链接：https://ld246.com/article/1649667297287

来源网站：链滴

许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

```
<p><img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfile
com/bing/20210803.jpg?imageView2/1/w/960/h/540/interlace/1/q/100"></p>
<h2 id="再谈多线程"><strong>再谈多线程</strong></h2>
<blockquote>
<p><strong>JUC 相对于 Java 应用层的学习难度更大，<strong><strong><strong>开篇推荐掌
的预备知识：<strong><strong><strong>JavaSE 多线程部分</strong></strong></strong>（
备）</strong></strong></strong>、操作系统、JVM</strong>****（推荐）****<strong>、计
机组成原理。掌握预备知识会让你的学习更加轻松，其中，JavaSE 多线程部分要求必须掌握，否则
法继续学习本教程！我们不会再去重复教学 JavaSE 阶段的任何知识了。</strong></p>
<p><strong>各位小伙伴一定要点击收藏按钮（收藏 = 学会）</strong></p>
</blockquote>
<p><strong>还记得我们在 JavaSE 中学习的多线程吗？让我们来回顾一下：</strong></p>
<p><strong>在我们的操作系统之上，可以同时运行很多个进程，并且每个进程之间相互隔离互不
扰。我们的 CPU 会通过时间片轮转算法，为每一个进程分配时间片，并在时间片使用结束后切换下
个进程继续执行，通过这种方式来实现宏观上的多个程序同时运行。</strong></p>
<p><strong>由于每个进程都有一个自己的内存空间，进程之间的通信就变得非常麻烦（比如要共
某些数据）而且执行不同进程会产生上下文切换，非常耗时，那么有没有一种更好地方案呢？</stron
></p>
<p><strong>后来，线程横空出世，一个进程可以有多个线程，线程是程序执行中一个单一的顺序
制流程，现在线程才是程序执行流的最小单元，各个线程之间共享程序的内存空间（也就是所在进程
内存空间），上下文切换速度也高于进程。</strong></p>
<p><strong>现在有这样一个问题：</strong></p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl"> public static void main(String[] args) {
</span></span><span class="highlight-line"><span class="highlight-cl">    &
bsp;int[] arr = new int[]{3, 1, 5, 2, 4};
</span></span><span class="highlight-line"><span class="highlight-cl">    &
bsp;//请将上面的数组按升序输出
</span></span><span class="highlight-line"><span class="highlight-cl"> }
</span></span></code></pre>
<p><strong>按照正常思维，我们肯定是这样：</strong></p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl"> public static void main(String[] args) {
</span></span><span class="highlight-line"><span class="highlight-cl">    &
bsp;int[] arr = new int[]{3, 1, 5, 2, 4};
</span></span><span class="highlight-line"><span class="highlight-cl">       //直
排序吧
</span></span><span class="highlight-line"><span class="highlight-cl">    &
bsp;Arrays.sort(arr);
</span></span><span class="highlight-line"><span class="highlight-cl">    &
bsp;for (int i : arr) {
</span></span><span class="highlight-line"><span class="highlight-cl">    &
bsp;    System.out.println(i);
</span></span><span class="highlight-line"><span class="highlight-cl">     }
</span></span><span class="highlight-line"><span class="highlight-cl"> }
</span></span></code></pre>
<p><strong>而我们学习了多线程之后，可以换个思路来实现：</strong></p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl"> public static void main(String[] args) {
</span></span><span class="highlight-line"><span class="highlight-cl">    &
bsp;int[] arr = new int[]{3, 1, 5, 2, 4};
</span></span><span class="highlight-line"><span class="highlight-cl"> 
</span></span><span class="highlight-line"><span class="highlight-cl">    &
bsp;for (int i : arr) {
```

```
</span></span><span class="highlight-line"><span class="highlight-cl">    &
bsp;    new Thread(() -&gt; {
</span></span><span class="highlight-line"><span class="highlight-cl">    &
bsp;        try {
</span></span><span class="highlight-line"><span class="highlight-cl">    &
bsp;            Thread.sleep(i * 1000);   //越小的数
眠时间越短，优先被打印
</span></span><span class="highlight-line"><span class="highlight-cl">    &
bsp;            System.out.println(i);
</span></span><span class="highlight-line"><span class="highlight-cl">    &
bsp;         } catch (InterruptedException e) {
</span></span><span class="highlight-line"><span class="highlight-cl">    &
bsp;            e.printStackTrace();
</span></span><span class="highlight-line"><span class="highlight-cl">    &
bsp;         }
</span></span><span class="highlight-line"><span class="highlight-cl">    &
bsp;   }).start();
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl"> }
</span></span></code></pre>
<p><strong>我们接触过的很多框架都在使用多线程，比如 Tomcat 服务器，所有用户的请求都是
过不同的线程来进行处理的，这样我们的网站才可以同时响应多个用户的请求，要是没有多线程，可
而知服务器的处理效率会有多低。</strong></p>
<p><strong>虽然多线程能够为我们解决很多问题，但是，如何才能正确地使用多线程，如何才能
多线程的资源合理使用，这都是我们需要关心的问题。</strong></p>
<p><strong>在 Java 5 的时候，新增了 java.util.concurrent（JUC）包，其中包括大量用于多线
编程的工具类，目的是为了更好的支持高并发任务，让开发者进行多线程编程时减少竞争条件和死锁
问题！通过使用这些工具类，我们的程序会更加合理地使用多线程。而我们这一系列视频的主角，正
</strong> <code>JUC</code><strong>。</strong></p>
<p><strong>但是我们先不着急去看这些内容，第一章，我们先来补点基础知识。</strong></p>
<hr>
<h2 id="并发与并行"><strong>并发与并行</strong></h2>
<p><strong>我们经常听到并发编程，那么这个并发代表的是什么意思呢？而与之相似的并行又是
么意思？它们之间有什么区别？</strong></p>
<p><strong>比如现在一共有三个工作需要我们去完成。</strong></p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image-20220301213510841"
data-src="https://b3logfile.com/file/2022/04/solo-fetchupload-1461083614093642917 8-7f93
f3c.jpeg?imageView2/2/interlace/1/format/jpg"></p>
<h3 id="顺序执行"><strong>顺序执行</strong></h3>
<p><strong>顺序执行其实很好理解，就是我们依次去将这些任务完成了：</strong></p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image-20220301213629649"
data-src="https://b3logfile.com/file/2022/04/solo-fetchupload-17378574013174158599-765a
82f.jpeg?imageView2/2/interlace/1/format/jpg"></p>
<p><strong>实际上就是我们同一时间只能处理一个任务，所以需要前一个任务完成之后，才能继
下一个任务，依次完成所有任务。</strong></p>
<h3 id="并发执行"><strong>并发执行</strong></h3>
<p><strong>并发执行也是我们同一时间只能处理一个任务，但是我们可以每个任务轮着做（时间
轮转）：</strong></p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image-20220301214032719"
data-src="https://b3logfile.com/file/2022/04/solo-fetchupload-3847914299463850416-d729
076.jpeg?imageView2/2/interlace/1/format/jpg"></p>
<p><strong>只要我们单次处理分配的时间足够的短，在宏观看来，就是三个任务在同时进行。</st
ong></p>
<p><strong>而我们 Java 中的线程，正是这种机制，当我们需要同时处理上百个上千个任务时，很
```

显 CPU 的数量是不可能赶得上我们的线程数的，所以说这时就要求我们的程序有良好的并发性能，
应对同一时间大量的任务处理。学习 Java 并发编程，能够让我们在以后的实际场景中，知道该如何
对高并发的情况。</p>
<h3 id="并行执行">并行执行</h3>
<p>并行执行就突破了同一时间只能处理一个任务的限制，我们同一时间可以做多个任务
</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image-20220301214238743"
data-src="https://b3logfile.com/file/2022/04/solo-fetchupload-1756707384035823994 2-e2d
33be.jpeg?imageView2/2/interlace/1/format/jpg"></p>
<p>比如我们要进行一些排序操作，就可以用到并行计算，只需要等待所有子任务完成，
后将结果汇总即可。包括分布式计算模型 MapReduce，也是采用的并行计算思路。</p>
<hr>
<h2 id="再谈锁机制">再谈锁机制</h2>
<p>谈到锁机制，相信各位应该并不陌生了，我们在 JavaSE 阶段，通过使用 <
ode>synchronized</code> 关键字来实现锁，这样就能够很好地解决线程之间争抢资源
情况。那么，<code>synchronized</code> 底层到底是如何实现的呢？</str
ng></p>
<p>我们知道，使用 <code>synchronized</code>，一定是和某
对象相关联的，比如我们要对某一段代码加锁，那么我们就需要提供一个对象来作为锁本身：</stron
></p>
<pre><code class="highlight-chroma"><span class="highlight
cl"> public static void main(String[] args) {
   &
bsp;synchronized (Main.class) {
   &
bsp;   //这里使用的是Main类的Class对象作为锁
   }
 }
</code></pre>
<p>我们来看看，它变成字节码之后会用到哪些指令：</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image-20220302111724784"
data-src="https://b3logfile.com/file/2022/04/solo-fetchupload-1098700915240932454 0-45d
6022.jpeg?imageView2/2/interlace/1/format/jpg"></p>
<p>其中最关键的就是 <code>monitorenter</code> 指令了，可
看到之后也有 <code>monitorexit</code> 与之进行匹配（注意这里有 2 个
，<code>monitorenter</code> 和 <code>monitorexit</code
分别对应加锁和释放锁，在执行 <code>monitorenter</code>
前需要尝试获取锁，每个对象都有一个 <code>monitor</code> 监视器与之
应，而这里正是去获取对象监视器的所有权，一旦 <code>monitor</code>
有权被某个线程持有，那么其他线程将无法获得（管程模型的一种实现）。</p>
<p>在代码执行完成之后，我们可以看到，一共有两个 <code>monitorexit</
ode> 在等着我们，那么为什么这里会有两个呢，按理说 <code>monitorente
</code> 和 <code>monitorexit</code> 不应该一一对应吗，这
为什么要释放锁两次呢？</p>
<p>首先我们来看第一个，这里在释放锁之后，会马上进入到一个 goto 指令，跳转到 15
，而我们的 15 行对应的指令就是方法的返回指令，其实正常情况下只会执行第一个 <cod
>monitorexit</code> 释放锁，在释放锁之后就接着同步代码块后面的内容继续向下执
了。而第二个，其实是用来处理异常的，可以看到，它的位置是在 12 行，如果程序运行发生异常，
么就会执行第二个 <code>monitorexit</code>，并且会继续向下通过</stro
g> <code>athrow</code> 指令抛出异常，而不是直接跳转到 15 行正常运行下去。</st
ong></p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image-20220302114613847"
data-src="https://b3logfile.com/file/2022/04/solo-fetchupload-1773189842804105294 9-6bcf
2d4.jpeg?imageView2/2/interlace/1/format/jpg"></p>

<p>实际上 <code>synchronized</code> 使用的锁就是存储在 Ja
a 对象头中的，我们知道，对象是存放在堆内存中的，而每个对象内部，都有一部分空间用于存储对
头信息，而对象头信息中，则包含了 Mark Word 用于存放 <code>hashCode</code>
strong>和对象的锁信息，在不同状态下，它存储的数据结构有一些不同。</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image-20220302203846868"
data-src="https://b3logfile.com/file/2022/04/solo-fetchupload-10017402941809940505-725
5f53.jpeg?imageView2/2/interlace/1/format/jpg"></p>
<h3 id="重量级锁">重量级锁</h3>
<p>在 JDK6 之前，<code>synchronized</code> 一直被称为重
级锁，<code>monitor</code> 依赖于底层操作系统的 Lock 实现，Java 的
程是映射到操作系统的原生线程上，切换成本较高。而在 JDK6 之后，锁的实现得到了改进。我们先
最原始的重量级锁开始：</p>
<p>我们说了，每个对象都有一个 monitor 与之关联，在 Java 虚拟机（HotSpot）中，m
nitor 是由 ObjectMonitor 实现的：</p>
<pre><code class="highlight-chroma"><span class="highlight
cl"> ObjectMonitor() {
   &
bsp;_header   = NULL;
   &
bsp;_count   = 0; //记录个数
   &
bsp;_waiters   = 0,
   &
bsp;_recursions   = 0;
   &
bsp;_object   = NULL;
   &
bsp;_owner   = NULL;
   &
bsp;_WaitSet   = NULL; //处于wait状态的线程，会被加入到_WaitSet
   &
bsp;_WaitSetLock = 0 ;
   &
bsp;_Responsible = NULL ;
   &
bsp;_succ   = NULL ;
   &
bsp;_cxq   = NULL ;
   &
bsp;FreeNext   = NULL ;
   &
bsp;_EntryList   = NULL ; //处于等待锁block状态的线程，会被加入到该列表
   &
bsp;_SpinFreq   = 0 ;
   &
bsp;_SpinClock   = 0 ;
   &
bsp;OwnerIsThread = 0 ;
 }
</code></pre>
<p>每个等待锁的线程都会被封装成 ObjectWaiter 对象，进入到如下机制：<
p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="img" data-src="https://b3lo
file.com/file/2022/04/solo-fetchupload-2018309648638638305-eaf87c0b.jpeg?imageView2/2/

nterlace/1/format/jpg"></p>
<p>ObjectWaiter 首先会进入 Entry Set 等着，当线程获取到对象的 <code>
onitor</code> 后进入 The Owner 区域并把 <code>monitor</code> <str
ng>中的 <code>owner</code> 变量设置为当前线程，同时 <co
e>monitor</code> 中的计数器 <code>count</code> 加 1，若
程调用 <code>wait()</code> 方法，将释放当前持有的 <code>
onitor</code>，<code>owner</code> 变量恢复为 <
ode>null</code>，<code>count</code> 自减 1，同时该线程
入 WaitSet 集合中等待被唤醒。若当前线程执行完毕也将释放 <code>monitor</code>
并复位变量的值，以便其他线程进入获取对象的 <code>monitor</code><st
ong>。</p>
<p>虽然这样的设计思路非常合理，但是在大多数应用上，每一个线程占用同步代码块的
间并不是很长，我们完全没有必要将竞争中的线程挂起然后又唤醒，并且现代 CPU 基本都是多核心
行的，我们可以采用一种新的思路来实现锁。</p>
<p>在 JDK1.4.2 时，引入了自旋锁（JDK6 之后默认开启），它不会将处于等待状态的线
挂起，而是通过无限循环的方式，不断检测是否能够获取锁，由于单个线程占用锁的时间非常短，所
说循环次数不会太多，可能很快就能够拿到锁并运行，这就是自旋锁。当然，仅仅是在等待时间非常
的情况下，自旋锁的表现会很好，但是如果等待时间太长，由于循环是需要处理器继续运算的，所以
样只会浪费处理器资源，因此自旋锁的等待时间是有限制的，默认情况下为 10 次，如果失败，那么
进而采用重量级锁机制。</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image-20220302163246988"
data-src="https://b3logfile.com/file/2022/04/solo-fetchupload-8446509607204982432-3b5d
8f9.jpeg?imageView2/2/interlace/1/format/jpg"></p>
<p>在 JDK6 之后，自旋锁得到了一次优化，自旋的次数限制不再是固定的，而是自适应
化的，比如在同一个锁对象上，自旋等待刚刚成功获得过锁，并且持有锁的线程正在运行，那么这次
旋也是有可能成功的，所以会允许自旋更多次。当然，如果某个锁经常都自旋失败，那么有可能会不
采用自旋策略，而是直接使用重量级锁。</p>
<h3 id="轻量级锁">轻量级锁</h3>
<blockquote>
<p>从 JDK 1.6 开始，为了减少获得锁和释放锁带来的性能消耗，就引入了轻量级锁。</st
ong></p>
</blockquote>
<p>轻量级锁的目标是，在无竞争情况下，减少重量级锁产生的性能消耗（并不是为了代
重量级锁，实际上就是赌一手同一时间只有一个线程在占用资源），包括系统调用引起的内核态与用
态切换、线程阻塞造成的线程切换等。它不像是重量级锁那样，需要向操作系统申请互斥量。它的运
机制如下：</p>
<p>在即将开始执行同步代码块中的内容时，会首先检查对象的 Mark Word，查看锁对
是否被其他线程占用，如果没有任何线程占用，那么会在当前线程中所处的栈帧中建立一个名为锁记
（Lock Record）的空间，用于复制并存储对象目前的 Mark Word 信息（官方称为 Displaced Mark
Word）。</p>
<p>接着，虚拟机将使用 CAS 操作将对象的 Mark Word 更新为轻量级锁状态（数据结构
为指向 Lock Record 的指针，指向的是当前的栈帧）</p>
<blockquote>
<p>CAS（Compare And Swap）是一种无锁算法（我们之前在 Springboot 阶段已经讲
过了），它并不会为对象加锁，而是在执行的时候，看看当前数据的值是不是我们预期的那样，如果
，那就正常进行替换，如果不是，那么就替换失败。比如有两个线程都需要修改变量 <co
e>i</code> 的值，默认为 10，现在一个线程要将其修改为 20，另一个要修改为 30，如
他们都使用 CAS 算法，那么并不会加锁访问 <code>i</code>，而是直接尝
修改 <code>i</code> 的值，但是在修改时，需要确认 <code>i<
code> 是不是 10，如果是，表示其他线程还没对其进行修改，如果不是，那么说明其他
程已经将其修改，此时不能完成修改任务，修改失败。</p>
<p>在 CPU 中，CAS 操作使用的是 <code>cmpxchg</code> 指
，能够从最底层硬件层面得到效率的提升。</p>
</blockquote>

<p>如果 CAS 操作失败了的话，那么说明可能这时有线程已经进入这个同步代码块了，这
虚拟机会再次检查对象的 Mark Word，是否指向当前线程的栈帧，如果是，说明不是其他线程，而
当前线程已经有了这个对象的锁，直接放心大胆进同步代码块即可。如果不是，那确实是被其他线程
用了。</p>
<p>这时，轻量级锁一开始的想法就是错的（这时有对象在竞争资源，已经赌输了），所
说只能将锁膨胀为重量级锁，按照重量级锁的操作执行（注意锁的膨胀是不可逆的）</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image-20220302210830272"
data-src="https://b3logfile.com/file/2022/04/solo-fetchupload-14242394551963933448-41b
0594.jpeg?imageView2/2/interlace/1/format/jpg"></p>
<p>所以，轻量级锁 -> 失败 -> 自适应自旋锁 -> 失败 -> 重量级锁</strong
</p>
<p>解锁过程同样采用 CAS 算法，如果对象的 MarkWord 仍然指向线程的锁记录，那么
用 CAS 操作把对象的 MarkWord 和复制到栈帧中的 Displaced Mark Word 进行交换。如果替换失
，说明其他线程尝试过获取该锁，在释放锁的同时，需要唤醒被挂起的线程。</p>
<h3 id="偏向锁">偏向锁</h3>
<p>偏向锁相比轻量级锁更纯粹，干脆就把整个同步都消除掉，不需要再进行 CAS 操作了
它的出现主要是得益于人们发现某些情况下某个锁频繁地被同一个线程获取，这种情况下，我们可以
轻量级锁进一步优化。</p>
<p>偏向锁实际上就是专门为单个线程而生的，当某个线程第一次获得锁时，如果接下来
没有其他线程获取此锁，那么持有锁的线程将不再需要进行同步操作。</p>
<p>可以从之前的 MarkWord 结构中看到，偏向锁也会通过 CAS 操作记录线程的 ID，如
一直都是同一个线程获取此锁，那么完全没有必要在进行额外的 CAS 操作。当然，如果有其他线程
抢了，那么偏向锁会根据当前状态，决定是否要恢复到未锁定或是膨胀为轻量级锁。</p>
<p>如果我们需要使用偏向锁，可以添加 <code>-XX:+UseBiased</code> <
trong>参数来开启。</p>
<p>所以，最终的锁等级为：未锁定 < 偏向锁 < 轻量级锁 < 重量级锁<
p>
<p>值得注意的是，如果对象通过调用 <code>hashCode()</code> <stron
>方法计算过对象的一致性哈希值，那么它是不支持偏向锁的，会直接进入到轻量级锁状态，因为 Ha
h 是需要被保存的，而偏向锁的 Mark Word 数据结构，无法保存 Hash 值；如果对象已经是偏向锁
态，再去调用 <code>hashCode()</code> 方法，那么会直接将锁升级为重
级锁，并将哈希值存放在 <code>monitor</code>（有预留位置保存）中。<
strong></p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image-20220302214647735"
data-src="https://b3logfile.com/file/2022/04/solo-fetchupload-17798911808009255167-6c9
d638.jpeg?imageView2/2/interlace/1/format/jpg"></p>
<h3 id="锁消除和锁粗化">锁消除和锁粗化</h3>
<p>锁消除和锁粗化都是在运行时的一些优化方案，比如我们某段代码虽然加了锁，但是
运行时根本不可能出现各个线程之间资源争夺的情况，这种情况下，完全不需要任何加锁机制，所以
会被消除。锁粗化则是我们代码中频繁地出现互斥同步操作，比如在一个循环内部加锁，这样明显是
常消耗性能的，所以虚拟机一旦检测到这种操作，会将整个同步范围进行扩展。</p>
<hr>
<h2 id="JMM内存模型">JMM 内存模型</h2>
<p>注意这里提到的内存模型和我们在 JVM 中介绍的内存模型不在同一个层次，JVM 中
内存模型是虚拟机规范对整个内存区域的规划，而 Java 内存模型，是在 JVM 内存模型之上的抽象模
，具体实现依然是基于 JVM 内存模型实现的，我们会在后面介绍。</p>
<h3 id="Java内存模型">Java 内存模型</h3>
<p>我们在 <code>计算机组成原理</code> 中学习过，在我们的
PU 中，一般都会有高速缓存，而它的出现，是为了解决内存的速度跟不上处理器的处理速度的问题
所以 CPU 内部会添加一级或多级高速缓存来提高处理器的数据获取效率，但是这样也会导致一个很
显的问题，因为现在基本都是多核心处理器，每个处理器都有一个自己的高速缓存，那么又该怎么去
证每个处理器的高速缓存内容一致呢？</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image-20220303113148313"
data-src="https://b3logfile.com/file/2022/04/solo-fetchupload-4390072233351558888-ed68b

14.jpeg?imageView2/2/interlace/1/format/jpg"></p>
<p>为了解决缓存一致性的问题，需要各个处理器访问缓存时都遵循一些协议，在读写时根据协议来进行操作，这类协议有 MSI、MESI（Illinois Protocol）、MOSI、Synapse、Firefly 及 D
agon Protocol 等。</p>
<p>而 Java 也采用了类似的模型来实现支持多线程的内存模型：</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image-20220303114228749"
data-src="https://b3logfile.com/file/2022/04/solo-fetchupload-4701156581868309698-7d01
208.jpeg?imageView2/2/interlace/1/format/jpg"></p>
<p>JMM（Java Memory Model）内存模型规定如下：</p>

所有的变量全部存储在主内存（注意这里包括下面提到的变量，指的都是会出现竞争
变量，包括成员变量、静态变量等，而局部变量这种属于线程私有，不包括在内）
每条线程有着自己的工作内存（可以类比 CPU 的高速缓存）线程对变量的所有操作，
须在工作内存中进行，不能直接操作主内存中的数据。
不同线程之间的工作内存相互隔离，如果需要在线程之间传递内容，只能通过主内存
成，无法直接访问对方的工作内存。

<p>也就是说，每一条线程如果要操作主内存中的数据，那么得先拷贝到自己的工作内存
，并对工作内存中数据的副本进行操作，操作完成之后，也需要从工作副本中将结果拷贝回主内存中
具体的操作就是 <code>Save</code>（保存）和 <code>Load</
ode>（加载）操作。</p>
<p>那么各位肯定会好奇，这个内存模型，结合之前 JVM 所讲的内容，具体是怎么实现的
？</p>

主内存：对应堆中存放对象的实例的部分。
工作内存：对应线程的虚拟机栈的部分区域，虚拟机可能会对这部分内存进行优化，
其放在 CPU 的寄存器或是高速缓存中。比如在访问数组时，由于数组是一段连续的内存空间，所以
以将一部分连续空间放入到 CPU 高速缓存中，那么之后如果我们顺序读取这个数组，那么大概率会
接缓存命中。

<p>前面我们提到，在 CPU 中可能会遇到缓存不一致的问题，而 Java 中，也会遇到，比
下面这种情况：</p>
<pre><code class="highlight-chroma"><span class="highlight
cl"> public class Main {
   &
bsp;private static int i = 0;
   &
bsp;public static void main(String[] args) throws InterruptedException {
   &
bsp;   new Thread(() -> {
   &
bsp;   for (int j = 0; j < 100000; j++) i++;
   &
bsp;   System.out.println("线程1结束");
   &
bsp;   }).start();
   &
bsp;   new Thread(() -> {
   &
bsp;   for (int j = 0; j < 100000; j++) i++;
   &
bsp;   System.out.println("线程2结束");
<span class="highlight-line"><span class="highlight-cl">   &
bsp;   }).start();

```
</span></span><span class="highlight-line"><span class="highlight-cl">    &
bsp;    //等上面两个线程结束
</span></span><span class="highlight-line"><span class="highlight-cl">    &
bsp;    Thread.sleep(1000);
</span></span><span class="highlight-line"><span class="highlight-cl">    &
bsp;    System.out.println(i);
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl"> }
</span></span></code></pre>
<p><strong>可以看到这里是两个线程同时对变量</strong> <code>i</code> <strong>各自进
 100000 次自增操作，但是实际得到的结果并不是我们所期望的那样。</strong></p>
<p><strong>那么为什么会这样呢？在之前学习了 JVM 之后，相信各位应该已经知道，自增操作实
上并不是由一条指令完成的（注意一定不要理解为一行代码就是一个指令完成的）：</strong></p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image-20220303143131899"
data-src="https://b3logfile.com/file/2022/04/solo-fetchupload-17063584260272377287-b0d
3cd8.jpeg?imageView2/2/interlace/1/format/jpg"></p>
<p><strong>包括变量</strong> <code>i</code> <strong>的获取、修改、保存，都是被拆分
一个一个的操作完成的，那么这个时候就有可能出现在修改完保存之前，另一条线程也保存了，但是
前线程是毫不知情的。</strong></p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image-20220303144344450"
data-src="https://b3logfile.com/file/2022/04/solo-fetchupload-6744051310857330350-67ab
943.jpeg?imageView2/2/interlace/1/format/jpg"></p>
<p><strong>所以说，我们当时在 JavaSE 阶段讲解这个问题的时候，是通过</strong> <code>sy
chronized</code> <strong>关键字添加同步代码块解决的，当然，我们后面还会讲解另外的解决
案（原子类）</strong></p>
<h3 id="重排序"><strong>重排序</strong></h3>
<p><strong>在编译或执行时，为了优化程序的执行效率，编译器或处理器常常会对指令进行重排
，有以下情况：</strong></p>
<ol>
<li><strong>编译器重排序：Java 编译器通过对 Java 代码语义的理解，根据优化规则对代码指令
行重排序。</strong></li>
<li><strong>机器指令级别的重排序：现代处理器很高级，能够自主判断和变更机器指令的执行顺
。</strong></li>
</ol>
<p><strong>指令重排序能够在不改变结果（单线程）的情况下，优化程序的运行效率，比如：</st
ong></p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl"> public static void main(String[] args) {
</span></span><span class="highlight-line"><span class="highlight-cl">    &
bsp;int a = 10;
</span></span><span class="highlight-line"><span class="highlight-cl">    &
bsp;int b = 20;
</span></span><span class="highlight-line"><span class="highlight-cl">    &
bsp;System.out.println(a + b);
</span></span><span class="highlight-line"><span class="highlight-cl"> }
</span></span></code></pre>
<p><strong>我们其实可以交换第一行和第二行：</strong></p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl"> public static void main(String[] args) {
</span></span><span class="highlight-line"><span class="highlight-cl">    &
bsp;int b = 10;
</span></span><span class="highlight-line"><span class="highlight-cl">    &
bsp;int a = 20;
</span></span><span class="highlight-line"><span class="highlight-cl">    &
```

bsp;System.out.println(a + b);
 }
</code></pre>
<p>即使发生交换，但是我们程序最后的运行结果是不会变的，当然这里只通过代码的形
演示，实际上 JVM 在执行字节码指令时也会进行优化，可能两个指令并不会按照原有的顺序进行。</
trong></p>
<p>虽然单线程下指令重排确实可以起到一定程度的优化作用，但是在多线程下，似乎会
致一些问题：</p>
<pre><code class="highlight-chroma"><span class="highlight
cl">public class Main {
   private static int
a = 0;
   private static int
b = 0;
   public static vo
d main(String[] args) {
   new Thread(()
-> {
   if(b == 1) {
   if(a == 0
{
   Syst
m.out.println("A");
   }else {
   Syst
m.out.println("B");
   }
   }
   }).start();
   new Thread(()
-> {
   a = 1;
   b = 1;
   }).start();
   }
}
</code></pre>
<p>上面这段代码，在正常情况下，按照我们的正常思维，是不可能输出 <cod
>A</code> 的，因为只要 b 等于 1，那么 a 肯定也是 1 才对，因为 a 是在 b 之前完成
赋值。但是，如果进行了重排序，那么就有可能，a 和 b 的赋值发生交换，b 先被赋值为 1，而恰巧
个时候，线程 1 开始判定 b 是不是 1 了，这时 a 还没来得及被赋值为 1，可能线程 1 就已经走到打
那里去了，所以，是有可能输出 <code>A</code> 的。</p>
<h3 id="volatile关键字">volatile 关键字</h3>
<p>好久好久都没有认识新的关键字了，今天我们来看一个新的关键字 <code
volatile</code>，开始之前我们先介绍三个词语：</p>

原子性：其实之前讲过很多次了，就是要做什么事情要么做完，要么就不做，不存在
一半的情况。
可见性：指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程
够立即看得到修改的值。
有序性：即程序执行的顺序按照代码的先后顺序执行。

<p>我们之前说了，如果多线程访问同一个变量，那么这个变量会被线程拷贝到自己的工
内存中进行操作，而不是直接对主内存中的变量本体进行操作，下面这个操作看起来是一个有限循环

但是是无限的：</p>
<pre><code class="highlight-chroma"><span class="highlight
cl">public class Main {
   private static int
a = 0;
   public static vo
d main(String[] args) throws InterruptedException {
   new Thread(()
-> {
   while (a =
 0);
   System.out
println("线程结束！");
   }).start();

   Thread.sleep
1000);
   System.out.pr
ntln("正在修改a的值...");
   a = 1;   //很
显，按照我们的逻辑来说，a的值被修改那么另一个线程将不再循环
   }
}
</code></pre>
<p>实际上这就是我们之前说的，虽然我们主线程中修改了 a 的值，但是另一个线程并不
道 a 的值发生了改变，所以循环中依然是使用旧值在进行判断，因此，普通变量是不具有可见性的。<
strong></p>
<p>要解决这种问题，我们第一个想到的肯定是加锁，同一时间只能有一个线程使用，这
总行了吧，确实，这样的话肯定是可以解决问题的：</p>
<pre><code class="highlight-chroma"><span class="highlight
cl">public class Main {
   private static int
a = 0;
   public static vo
d main(String[] args) throws InterruptedException {
   new Thread(()
-> {
   while (a =
 0) {
   synchro
ized (Main.class){}
   }
   System.out
println("线程结束！");
   }).start();

   Thread.sleep
1000);
   System.out.pr
ntln("正在修改a的值...");
   synchronized
(Main.class){
   a = 1;
<span class="highlight-line"><span class="highlight-cl">   }

```
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<p><strong>但是，除了硬加一把锁的方案，我们也可以使用</strong> <code>volatile</code>
<strong>关键字来解决，此关键字的第一个作用，就是保证变量的可见性。当写一个</strong> <c
de>volatile</code> <strong>变量时，JMM 会把该线程本地内存中的变量强制刷新到主内存中去
并且这个写会操作会导致其他线程中的</strong> <code>volatile</code> <strong>变量缓存无
，这样，另一个线程修改了这个变时，当前线程会立即得知，并将工作内存中的变量更新为最新的版
。</strong></p>
<p><strong>那么我们就来试试看：</strong></p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">public class Main {
</span></span><span class="highlight-line"><span class="highlight-cl">    //添加volatile
键字
</span></span><span class="highlight-line"><span class="highlight-cl">    private static vo
atile int a = 0;
</span></span><span class="highlight-line"><span class="highlight-cl">    public static vo
d main(String[] args) throws InterruptedException {
</span></span><span class="highlight-line"><span class="highlight-cl">        new Thread(()
-&gt; {
</span></span><span class="highlight-line"><span class="highlight-cl">            while (a =
 0);
</span></span><span class="highlight-line"><span class="highlight-cl">            System.out
println("线程结束！");
</span></span><span class="highlight-line"><span class="highlight-cl">        }).start();
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">        Thread.sleep
1000);
</span></span><span class="highlight-line"><span class="highlight-cl">        System.out.pr
ntln("正在修改a的值...");
</span></span><span class="highlight-line"><span class="highlight-cl">        a = 1;
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<p><strong>结果还真的如我们所说的那样，当 a 发生改变时，循环立即结束。</strong></p>
<p><strong>当然，虽然说</strong> <code>volatile</code> <strong>能够保证可见性，但
不能保证原子性，要解决我们上面的</strong> <code>i++</code> <strong>的问题，以我们目
所学的知识，还是只能使用加锁来完成：</strong></p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">public class Main {
</span></span><span class="highlight-line"><span class="highlight-cl">    private static vo
atile int a = 0;
</span></span><span class="highlight-line"><span class="highlight-cl">    public static vo
d main(String[] args) throws InterruptedException {
</span></span><span class="highlight-line"><span class="highlight-cl">        Runnable r =
) -&gt; {
</span></span><span class="highlight-line"><span class="highlight-cl">            for (int i =
; i &lt; 10000; i++) a++;
</span></span><span class="highlight-line"><span class="highlight-cl">            System.out
println("任务完成！");
</span></span><span class="highlight-line"><span class="highlight-cl">        };
</span></span><span class="highlight-line"><span class="highlight-cl">        new Thread(r)
start();
</span></span><span class="highlight-line"><span class="highlight-cl">        new Thread(r)
```

```
start();
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">        //等待线程执
完成
</span></span><span class="highlight-line"><span class="highlight-cl">        Thread.sleep
1000);
</span></span><span class="highlight-line"><span class="highlight-cl">        System.out.pr
ntln(a);
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<p><strong>不对啊，</strong><code>volatile</code> <strong>不是能在改变变量的时候其
线程可见吗，那为什么还是不能保证原子性呢？还是那句话，自增操作是被瓜分为了多个步骤完成的
虽然保证了可见性，但是只要手速够快，依然会出现两个线程同时写同一个值的问题（比如线程 1 刚
将 a 的值更新为 100，这时线程 2 可能也已经执行到更新 a 的值这条指令了，已经刹不住车了，所
依然会将 a 的值再更新为一次 100）</strong></p>
<p><strong>那要是真的遇到这种情况，那么我们不可能都去写个锁吧？后面，我们会介绍原子类
专门解决这种问题。</strong></p>
<p><strong>最后一个功能就是</strong> <code>volatile</code> <strong>会禁止指令重排
也就是说，如果我们操作的是一个</strong> <code>volatile</code> <strong>变量，它将不会
现重排序的情况，也就解决了我们最上面的问题。那么它是怎么解决的重排序问题呢？若用 volatile
饰共享变量，在编译时，会在指令序列中插入</strong> <code>内存屏障</code> <strong>来禁
特定类型的处理器重排序</strong></p>
<blockquote>
<p><strong>内存屏障（Memory Barrier）又称内存栅栏，是一个 CPU 指令，它的作用有两个：<
strong></p>
<ol>
<li><strong>保证特定操作的顺序</strong></li>
<li><strong>保证某些变量的内存可见性（volatile 的内存可见性，其实就是依靠这个实现的）</str
ng></li>
</ol>
<p><strong>由于编译器和处理器都能执行指令重排的优化，如果在指令间插入一条 Memory Barri
r 则会告诉编译器和 CPU，不管什么指令都不能和这条 Memory Barrier 指令重排序。</strong></
>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image-20220303172519404"
data-src="https://b3logfile.com/file/2022/04/solo-fetchupload-9554172308687664630-743b
be8.jpeg?imageView2/2/interlace/1/format/jpg"></p>
<table>
<thead>
<tr>
<th><strong>屏障类型</strong></th>
<th><strong>指令示例</strong></th>
<th><strong>说明</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LoadLoad</strong></td>
<td><strong>Load1;LoadLoad;Load2</strong></td>
<td><strong>保证 Load1 的读取操作在 Load2 及后续读取操作之前执行</strong></td>
</tr>
<tr>
<td><strong>StoreStore</strong></td>
<td><strong>Store1;StoreStore;Store2</strong></td>
```

<td>在 Store2 及其后的写操作执行前，保证 Store1 的写操作已刷新到主内存
/td>
</tr>
<tr>
<td>LoadStore</td>
<td>Load1;LoadStore;Store2</td>
<td>在 Store2 及其后的写操作执行前，保证 Load1 的读操作已读取结束</td

</tr>
<tr>
<td>StoreLoad</td>
<td>Store1;StoreLoad;Load2</td>
<td>保证 load1 的写操作已刷新到主内存之后，load2 及其后的读操作才能执行</strong
</td>
</tr>
</tbody>
</table>
</blockquote>
<p>所以 <code>volatile</code> 能够保证，之前的指令一定全
执行，之后的指令一定都没有执行，并且前面语句的结果对后面的语句可见。</p>
<p>最后我们来总结一下 <code>volatile</code> 关键字的三个
性：</p>

保证可见性
不保证原子性
防止指令重排

<p>在之后我们的设计模式系列视频中，还会讲解单例模式下 <code>volatile
/code> 的运用。</p>
<h3 id="happens-before原则">happens-before 原则</h3>
<p>经过我们前面的讲解，相信各位已经了解了 JMM 内存模型以及重排序等机制带来的
点和缺点，综上，JMM 提出了 <code>happens-before</code>（先行发生
原则，定义一些禁止编译优化的场景，来向各位程序员做一些保证，只要我们是按照原则进行编程，
么就能够保持并发编程的正确性。具体如下：</p>

****程序次序规则：****同一个线程中，按照程序的顺序，前面的操作 happens-befor
 后续的任何操作。

同一个线程内，代码的执行结果是有序的。其实就是，可能会发生指令重排，但是保
代码的执行结果一定是和按照顺序执行得到的一致，程序前面对某一个变量的修改一定对后续操作可
的，不可能会出现前面才把 a 修改为 1，接着读 a 居然是修改前的结果，这也是程序运行最基本的要
。

****监视器锁规则：****对一个锁的解锁操作，happens-before 后续对这个锁的加锁
作。

就是无论是在单线程环境还是多线程环境，对于同一个锁来说，一个线程对这个锁解
之后，另一个线程获取了这个锁都能看到前一个线程的操作结果。比如前一个线程将变量
code>x</code> 的值修改为了 <code>12</code> 并解锁，之后
一个线程拿到了这把锁，对之前线程的操作是可见的，可以得到 <code>x</code> <str
ng>是前一个线程修改后的结果 <code>12</code>（所以 synchronized 是有
happens-before 规则的）

****volatile 变量规则：****对一个 volatile 变量的写操作 happens-before 后续对这变量的读操作。

就是如果一个线程先去写一个 <code>volatile</code> 变量，接着另一个线程去读这个变量，那么这个写操作的结果一定对读的这个变量的线程可见。<
li>

****线程启动规则：****主线程 A 启动线程 B，线程 B 中可以看到主线程启动 B 之前操作。

在主线程 A 执行过程中，启动子线程 B，那么线程 A 在启动子线程 B 之前对共享变量修改结果对线程 B 可见。

****线程加入规则：****如果线程 A 执行操作 <code>join()</code> <str
ng>线程 B 并成功返回，那么线程 B 中的任意操作 happens-before 线程 A <code>join(
</code> 操作成功返回。
****传递性规则：****如果 A happens-before B，B happens-before C，那么 A ha
pens-before C。

<p>那么我们来从 happens-before 原则的角度，来解释一下下面的程序结果：</strong
</p>
<pre><code class="highlight-chroma"><span class="highlight
cl">public class Main {
   private static int
a = 0;
   private static i
t b = 0;
   public static vo
d main(String[] args) {
   a = 10;
   b = a + 1;
   new Thread(()
-> {
   if(b > 10)
System.out.println(a);
   }).start();
   }
}
</code></pre>
<p>首先我们定义以上出现的操作：</p>

****A：****将变量 <code>a</code> 的值修改为 <
ode>10</code>
****B：****将变量 <code>b</code> 的值修改为 <
ode>a + 1</code>
****C：主线程启动了一个新的线程，并在新的线程中获取</stro
g> <code>b</code>，进行判断，如果大于 <code>10</code> <strong
那么就打印 <code>a</code>

<p>首先我们来分析，由于是同一个线程，并且B</stron
>是一个赋值操作且读取了A</stro

g>**，那么按照程序次序规则，A h
ppens-before B，接着在 B 之后，马上执行了 C，按照线程启动规则
/strong>，在新的线程启动之前，当前线程之前的所有操作对新的线程是可见
，所以 B happens-before C，最后根据传递性规则</stro
g>，由于 A happens-before B，B happens-before C，所以 A happens-before C，
此在新的线程中会输出**<code>a</code> 修改后的结果 <code>10</code
。</p>

原文链接：JUC 笔记（一）