Java 并发编程学习总结

前言

该总结按照《Java 并发编程的艺术》的知识点顺序，对该书的重要或者一些细节进行总结，顺便对这方面进行复习。蓝色部分仍然为我认为的重要部分，里面非蓝色部分虽然不是我认为的重点，但是对于理解蓝色部分的概念起到了至关重要的作用。

里面的例子大部分都采用书上的例子（毕竟书上的例子最经典），有些括号的内容也穿插着我对该知识点的补充。后面我会再次看一遍作为复习，里面有些内容我也会进行填充，但是对于复习来说应该是足够了。

一、并发编程的挑战

1.1 上下文切换

单核处理器实现多线程的方式：CPU 通过给每个线程分配 CPU 时间片来实现这个机制。时间片是 CPU 分配给各个线程的时间，因为时间片非常短，所以 CPU 通过不停的切换线程执行，让我们感觉多个线程是同时执行的。时间片一般是几十毫秒。

CPU 通过时间片分配算法来循环执行任务，当前任务执行一个时间片后会切换到下一个任务。但是，在切换前会保存上一个任务的状态，以便下次切换回这个任务时，可以再加载这个任务的状态。所以任务从保存到再加载的过程就是一次上下文切换。

像这样的切换是会影响到效率的。保存当前任务状态就包括切换内核态堆栈、切换页表目录（当前信息是保存到内存的）、TLB 刷新等等。在高并发的情况下，这些所影响到的效率就不容忽视了。

1.1.1 多线程一定快吗？

书上代码不贴了，直接上结果：

循环次数	串行执行耗时 /ms	并发执行耗时	并发比串行快多少
1 亿	130	77	约 1 倍
1 千万	18	9	约 1 倍
1 百万	5	5	差不多
10 万	4	3	慢
1 万	0	1	慢

可以看到，在循环次数不超过 100w 时，串行速度比并发速度慢。这是因为线程有创建和上下文切换的开销。

上下文切换的开销包括直接开销和间接开销：

1）直接开销：

a. 切换页表全局目录
b. 切换内核态堆栈（现在的线程基本上是基于内核的）
c. 切换硬件上下文（线程恢复前，必须装入寄存器的数据统称为硬件上下文）
d. 刷新 TLB
e. 系统调度器的代码执行

2）间接开销：间接开销主要指的是虽然切换到一个新进程后，由于各种缓存并不热，速度运行会慢一些。如果进程始终都在一个 CPU 上调度还好一些，如果跨 CPU 的话，之前热起来的 TLB、L1、L2、L3 因为运行的进程已经变了，所以以局部性原理 cache 起来的代码、数据也都没有用了，导致新进程穿透到内存的 IO 会变多。

（参考《深入理解 Linux 内核》）

1.1.2 如何减少上下文切换

减少上下文切换次数的方法：

1）无锁并发编程：多线程竞争锁时，会引起上下文切换，所以在使用多线程处理数据时，可以采用一些策略来避免使用锁。

策略：将数据按照 id 的哈希值进行切分，不同的线程处理不同段的数据。

2）锁分离技术：分段锁 ConcurrentHashMap（这里的分段是对节点 Node）

3）CAS 算法：Java 的 Atomic 包的原子类型都使用 CAS 来更新数据

4）使用最少的线程：避免创建不需要的线程，比如任务很少，但是创建了很多线程来处理，这样会造成大量线程都出于等待状态。

5）协程：在单线程里实现多任务的调度，并在单线程里维持多个任务间的切换。

（书上写了 4 种，但是在参考了一些博客后，觉得锁分离技术也该算一种，原理和无锁并发编程的策略很像，也避免了太多线程进行上下文切换）

1.2 死锁

避免死锁的方法：

1）避免一个线程同时获得多个锁。

2）避免一个线程在锁内同时占用多个资源，尽量保证每个锁只占用一个资源。

3）尝试使用定时锁，使用 lock.tryLock(timeout)来替代使用内部锁机制。

4）对于数据库锁，加锁和解锁必须在一个数据库连接里，否则会出现解锁失败。

操作系统产生死锁条件（顺便复习）：

1）互斥条件：一个资源每次只能被一个进程使用

2）请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放

3）不剥夺条件：进程已获得的资源，在未使用完之前，不能强行剥夺

4）循环等待条件：若干进程之间形成一种头尾相接的循环等待资源关系

1.3 资源限制的挑战

资源限制是指在进行并发编程时，程序的执行速度受限于计算机硬件资源或软件资源。例如，服务器的带宽只有 2Mb/s，某个资源的下载速度是 1Mb/s 每秒，系统启动 10 个线程下载资源，下载速度不会变成 10Mb/s，所以在进行并发编程时，要考虑这些资源的限制。硬件资源限制有带宽的上传/下载速度、硬盘读写速度和 CPU 的处理速度。软件资源限制有数据库的连接数和 socket 连接数等。

在并发编程中，将代码执行速度加快的原则是将代码中串行执行的部分变成并发执行，但是如果将某段串行的代码并发执行，因为受限于资源，仍然在串行执行，这时候程序不仅不会加快执行，反而会更慢，因为增加了上下文切换和资源调度的时间。例如，之前看到一段程序使用多线程在办公网并发地下载和处理数据时，导致 CPU 利用率达到 100%，几个小时都不能运行完成任务，后来修改成单线程，一个小时就执行完成了。

对于硬件资源限制，可以考虑使用集群并行执行程序。既然单机的资源有限制，那么就让程序在多机上运行。比如使用 ODPS、Hadoop 或者自己搭建服务器集群，不同的机器处理不同的数据。可以通过“数据 ID%机器数”，计算得到一个机器编号，然后由对应编号的机器处理这笔数据。

对于软件资源限制，可以考虑使用资源池将资源复用。比如使用连接池将数据库和 Socket 连接复用，或者在调用对方 webservice 接口获取数据时，只建立一个连接。

在资源限制的情况下，根据不同的资源限制调整程序的并发度，比如下载文件程序依赖于两个资源——带宽和硬盘读写速度。有数据库操作时，涉及数据库连接数，如果 SQL 语句执行非常快，而线程的数量比数据库连接数大很多，则某些线程会被阻塞，等待数据库连接。

二、Java 并发机制的底层实现原理

Java 代码在编译后会变成 Java 字节码，字节码被类加载器加载到 JVM 里，JVM 执行字节码，最终需要转化为汇编指令在 CPU 上执行，Java 中所使用的并发机制依赖于 JVM 的实现和 CPU 的指令。

2.1 volatile 的应用

在多线程并发编程中 synchronized 和 volatile 都扮演着重要的角色，volatile 是轻量级的 synchronized，它在多处理器开发中保证了共享变量的“可见性”。可见性的意思是当一个线程修改一个共享变量时，另外一个线程能读到这个修改的值。如果 volatile 变量修饰符使用恰当的话，它比 synchronized 的使用和执行成本更低，因为它不会引起线程上下文的切换和调度。

2.1.1 volatile 的定义与实现原理

《Java 语言规范》中对 volatile 的定义：Java 编程语言允许线程访问共享变量，为了确保共享变量能被准确和一致地更新，线程应该确保通过排他锁单独获得这个变量。Java 语言提供了 volatile，在某些情况下比锁要更加方便。如果一个字段被声明成 volatile，Java 线程内存模型确保所有线程看到这个变量的值是一致的。

CPU 术语与说明：

术　　语	英文单词	术语描述
内存屏障	memory barriers	是一组处理器指令，用于实现对内存操作的顺序限制
缓冲行	cache line	缓存中可以分配的最小存储单位。处理器填写缓存线时会加载整个缓存线，需要使用多个主内存读周期
原子操作	atomic operations	不可中断的一个或一系列操作
缓存行填充	cache line fill	当处理器识别到从内存中读取操作数是可缓存的，处理器读取整个缓存行到适当的缓存（L1，L2，L3 的或所有）
缓存命中	cache hit	如果进行高速缓存行填充操作的内存位置仍然是下次处理器访问的地址时，处理器从缓存中读取操作数，而不是从内存读取
写命中	write hit	当处理器将操作数写回到一个内存缓存的区域时，它首先会检查这个缓存的内存地址是否在缓存行中，如果存在一个有效的缓存行，则处理器将这个操作数写回到缓存，而不是写回到内存，这个操作被称为写命中
写缺失	write misses the cache	一个有效的缓存行被写入到不存在的内存区域

（后面这段是自己查询 CPU 缓存原理的相关知识，参考了各种博客资料，我觉得有必要去从大环境理解上面表格概念，上面表格的术语有点难懂。具体参考十一章）

查看一段代码的汇编结果：

```
volatile Singleton instance = new Singleton();
// 汇编指令
0x01a3de1d: movb $0×0,0×1104800(%esi);
0x01a3de24: lock addl $0×0,(%esp);
```

可以看出，volatile 在进行写操作时，会变成两条汇编指令。

Lock 前缀的指令在多核处理器下会引发两件事：

1）将当前处理器缓存行的数据写回到系统内存。

2）这个写回内存的操作会使在其他 CPU 里缓存了该内存地址的数据无效。（上面两条在十一章后面做了描述，实际上就是符合 MESI 规则）

为了提高处理速度，处理器不直接和内存进行通信，而是先将系统内存的数据读到内部缓存（L1 和 L2）后再进行操作，但操作完不知道何时会写到内存。如果对声明了 volatile 的变量进行写操作，JVM 就会向处理器发送一条 Lock 前缀的指令，将这个变量所在的缓存行的数据写回到系统内存。

在多处理器下，为了保证各个处理器的缓存是一致的，就会实现缓存一致性协议，每个处理器通过嗅探在总线上传播的数据来检查自己缓存的值是不是过期了，当处理器发现自己缓存行对应的内存地址被修改，就会将当前处理器的缓存行设置成无效状态，当处理器对这个数据进行修改操作的时候，会重新从系统内存中把数据读到处理器缓存里。

（实际上多线程条件下，为了满足 MESI 协议，再其他线程或者核心要去对之前共享缓存行进行操作时，其他线程会让之前进行修改变量的线程将修改的值写回内存，然后其他线程才会从内存中读取。之所以是之前的共享缓存行，是因为在第一个线程修改缓存行时，就已经让其他线程对该共享缓存行设置为无效 I。这里是一段补充）。

因此在硬件层面上，volatile 实际上是使用 MESI 协议去保证可见性。（书上有对两条原则的具体描述，这里就不做总结了）

2.1.2 volatile 的使用优化

（这一小节在第十一章有补充）

著名的 Java 并发编程大师 Doug lea 在 JDK 7 的并发包里新增一个队列集合类 Linked- TransferQueue，它在使用 volatile 变量时，用一种追加字节的方式来优化队列出队和入队的性能。代码如下：

```
/** 队列中的头部节点 */
private transient f?inal PaddedAtomicReference<QNode> head;
/** 队列中的尾部节点 */
private transient f?inal PaddedAtomicReference<QNode> tail;
static f?inal class PaddedAtomicReference <T> extends AtomicReference T>  {
        // 使用很多 4 个字节的引用追加到 64 个字节
        Object p0, p1, p2, p3, p4, p5, p6, p7, p8, p9, pa, pb, pc, pd, pe;
        PaddedAtomicReference(T r) {
             super(r);
        }
}
public class AtomicReference <V> implements
        java.io.Serializable { private volatile V value;
        // 省略其他代码
}
```

为什么追加 64 字节能增加性能？因为对于英特尔酷睿 i7、酷睿、Atom 和 NetBurst，以及 Core Solo 和 Pentium M 处理器的 L1、L2 或 L3 缓存的高速缓存行是 64 个字节宽，不支持部分填充缓存行，这意味着，如果队列的头节点和尾节点都不足 64 字节的话，处理器会将它们都读到同一个高速缓存行中，在多处理器下每个处理器都会缓存同样的头、尾节点，当一个处理器试图修改头节点时，会将整个缓存行锁定，那么在缓存一致性机制的作用下，会导致其他处理器不能访问自己高速缓存中的尾节点，而队列的入队和出队操作则需要不停修改头节点和尾节点，所以在多处理器的情况下将会严重影响到队列的入队和出队效率。Doug lea 使用追加到 64 字节的方式来填满高速缓冲区的缓存行，避免头节点和尾节点加载到同一个缓存行，使头、尾节点在修改时不会互相锁定。

在使用 volatile 变量时，以下两种场景不应该使用追加字节的方式：

1）缓存行非 64 字节的处理器。

2）共享变量不会被频繁地写。

2.2 synchronized 的实现原理与应用

Java 中每一个对象都可以作为锁。具体表现为下面三种形式：
1）对于普通方法，锁的是当前实例对象（this）。
2）对于静态同步方法，锁的是当前类的 Class 对象。
3）对于同步代码块，锁的是 synchonized 括号里配置的对象。

在 JVM 规范中可以看到 Synchronized 在 JVM 里实现原理：JVM 基于进入和退出 Monitor 对象来实现方法同步和代码块同步。

代码块同步是使用 monitorenter 和 monitorexit 指令实现的，而方法同步是使用另外一种方式实现的，细节在 JVM 规范里并没有详细说明。但是，方法的同步同样可以使用这两个指令来实现。

monitorenter 指令是在编译后插入到同步代码块的开始位置，而 monitorexit 是插入到方法结束处和异常处（这个地方我测试了下，感觉应该是插入到同步代码块的结束处），JVM 要保证每个 monitorenter 必须有对应的 monitorexit 与之配对。任何对象都有一个 monitor 与之关联，当且一个 monitor 被持有后，它将处于锁定状态。线程执行到 monitorenter 指令时，将会尝试获取对象所对应的 monitor 的所有权，即尝试获得对象的锁。

2.2.1 Java 对象头

（这一小节在整理 JVM 里有更详细的，这里只讨论和锁有关的）

synchronized 用的锁是存在 Java 对象头里的（准确的说是锁记录位）。如果对象是数组类型，则虚拟机用 3 个字宽（Word）存储对象头，如果对象是非数组类型，则用 2 字宽存储对象头。这里一字宽等于 4 字节。

Java 对象头里的 Mark Word 里默认存储对象的 HashCode、分代年龄和锁标记位。32 位的 JVM 里的 Mark Word 如图：

锁状态	25bit	4bit	1bit 是否是偏向锁	2bit 锁标志位
无锁状态	对象的 hashCode	对象分代年龄	0	01

在运行期间，Mark Word 里存储的数据会随着锁标志位的变化而变化。Mark Word 可能变化为存储以下 4 种数据，如图：

锁状态	25bit		4bit	1bit	2bit
	23bit	2bit		是否是偏向锁	锁标志位
轻量级锁	指向栈中锁记录的指针				00
重量级锁	指向互斥量（重量级锁）的指针				10
GC 标记	空				11
偏向锁	线程 ID	Epoch	对象分代年龄	1	01

64 位的 JVM 里的 Mark Word，如图表示偏向锁：

锁状态	25bit	31bit	1bit	4bit	1bit	2bit
			cms_free	分代年龄	偏向锁	锁标志位
无锁	unused	hashCode			0	01
偏向锁	ThreadID(54bit) Epoch(2bit)				1	01

2.2.2 锁

JDK6 为了减少获得锁和释放锁带来的性能消耗，引入了“偏向锁”和“轻量级锁”，在 JDK6 中，锁一共有 4 种状态，级别从低到高依次是：无锁状态、偏向锁状态、轻量级锁状态和重量级锁状态（注意区分无锁和偏向锁所称的无锁）。

（这一小节的顺序按照《深入理解 Java 虚拟机》一书）

1. 自旋锁

互斥同步对性能最大的影响是阻塞的实现，挂起线程和恢复线程的操作都需要转入内核态中完成，这些操作给 Java 虚拟机的并发性能带来了很大的压力。同 时，虚拟机的开发团队也注意到在许多应用上，共享数据的锁定状态只会持续很短的一段时间，为了这段时间去挂起和恢复线程并不值得。现在绝大多数的个人电脑和服务器都是多路（核）处理器系 统，如果物理机器有一个以上的处理器或者处理器核心，能让两个或以上的线程同时并行执行，我们就可以让后面请求锁的那个线程“稍等一会”，但不放弃处理器的执行时间，看看持有锁的线程是否很快就会释放锁。为了让线程等待，我们只须让线程执行一个忙循环（自旋），这项技术就是所谓的自旋锁。

但是自旋等待不能代替阻塞，且先不说对处理器数量的要求，自旋等待本身虽然避免了线程切换的开销，但它是要占用处理器时间的，所以如果锁被占用的时间很短，自旋等待的效果就会非常好，反之如果锁被占用的时间很长，那么自旋的线程只会白白消耗处理器资源，而不会做任何有价值的工作，这就会带来性能的浪费。因此自旋等待的时间必须有一定的限度，如果自旋超过了限定的次数仍然没有成功获得锁，就应当使用传统的方式去挂起线程。自旋次数的默认值是十次。

2. 锁消除

锁消除是指虚拟机即时编译器在运行时，对一些代码要求同步，但是对被检测到不可能存在共享数据竞争的锁进行消除。锁消除的主要判定依据来源于逃逸分析的数据支持（逃逸分析在 JVM 知识总结时并没有做总结）。如果判断到一段代码中，在堆上的所有数据都不会逃逸出去被其他线程访问到，那就可以把它们当作栈上数据对待，认为它们是线程私有的，同步加锁自然就无须再进行。

3. 锁粗化

原则上，我们在编写代码的时候，总是推荐将同步块的作用范围限制得尽量小——只在共享数据的实际作用域中才进行同步，这样是为了使得需要同步的操作数量尽可能变少，即使存在锁竞争，等待锁的线程也能尽可能快地拿到锁。

大多数情况下，上面的原则都是正确的，但是如果一系列的连续操作都对同一个对象反复加锁和解锁，甚至加锁操作是出现在循环体之中的，那即使没有线程竞争，频繁地进行互斥同步操作也会导致不必要的性能损耗。

4. 偏向锁

大多数情况下，锁不仅不存在多线程竞争，而且总是由同一线程多次获得，为了让线程获得锁的代价更低而引入了偏向锁。当一个线程访问同步块去获取锁的时候，会在对象头和栈帧中的锁记录里存储锁偏向的线程 ID，以后该线程在进入和退出同步块时不需要进行 CAS 操作来加锁和解锁，只需要测试下对象头的 Mark Word 中是否存储着当前线程的偏向锁。如果测试成功，表示线程已经获得了锁。如果测试失败，则需要再测试一下 Mark Word 中偏向锁的标识是否设置成 1（表示当前是偏向锁）：如果没有设置，则使用 CAS 竞争锁；如果设置了，则尝试使用 CAS 将对象头的偏向锁指向当前线程。

偏向锁使用了一种等到竞争出现才释放锁的机制，所以当其他线程尝试竞争偏向锁时，持有偏向锁的线程才会释放锁。偏向锁的撤销，需要等待全局安全点（在这个时间点上没有正在执行的字节码）。它会首先暂停拥有偏向锁的线程，然后检查该线程是否活着：

1）线程不处于活动状态，则将对象头设置成无锁状态；

2）线程仍然活着，升级为轻量级锁。

（下面内容是《深入理解 Java 虚拟机》中的知识补充）

偏向锁的意思就是这个锁会偏向于第一个获得它的线程。如果在接下来的执行过程中，该锁一直没有被其他线程获取，则持有偏向锁的线程将永远不需要再进行同步。

当锁对象第一次被线程获取时，虚拟机将会把对象头（锁）中的标志位设置为“01”、把偏向模式设置为“1”，表示进入偏向模式，同时使用 CAS 操作把获取到这个锁的线程 ID 记录在对象（锁）的 Mark Word 中。如果操作成功，则持有偏向锁的线程以后每次进入这个锁的相关的代码块时，虚拟机都可以不再进行任何同步操作（这里可以解释为，当同一个线程不断重入偏向锁时，不需要进行 CAS 操作，只用判断锁记录的线程 ID 是否是该线程）。

一旦出现有其他线程去尝试获取这个锁，偏向模式就结束，注意，偏向锁只有当遇到其他线程去竞争偏向锁时，持有偏向锁的线程才会释放锁。根据锁对象是否处于被锁定状态决定是否撤销偏向（这里的撤销偏向只是偏向模式设置为 0）：

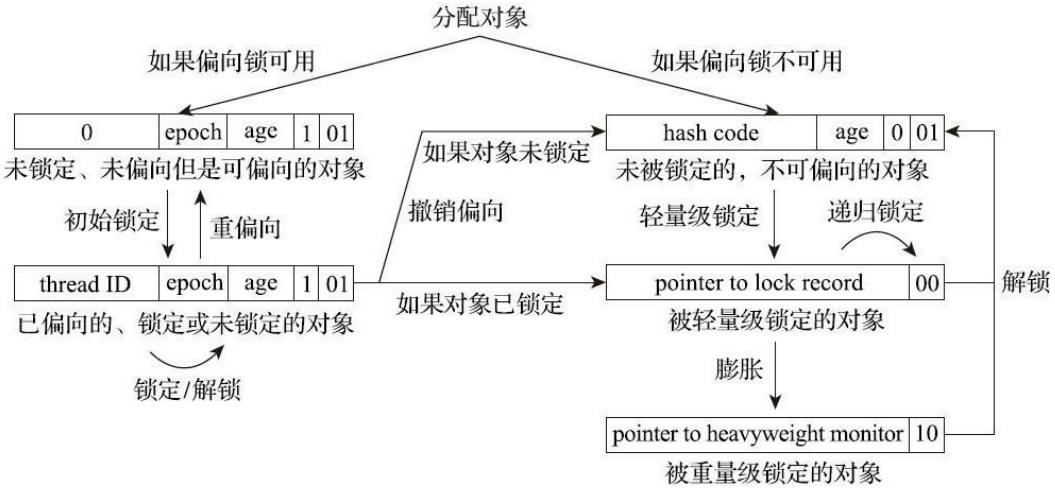
1）锁对象处于锁定状态，即标志位不为 01，不需要撤销偏向，即已经进入到轻量级锁状态，其他线程只需要自旋等待。

2）锁对象处于未锁定状态，即锁标志位为 01 的偏向锁状态（因为偏向锁不会主动释放锁，标志位为 01 就可以判断是否有锁定）。在安全点挂起该线程，检查线程是否活着。

a. 如果对象处于活动状态，升级为轻量级锁（具体看下面的轻量级锁

过程）

b. 如果对象为不活动状态，将对象头设置成无锁状态。



5. 轻量级锁

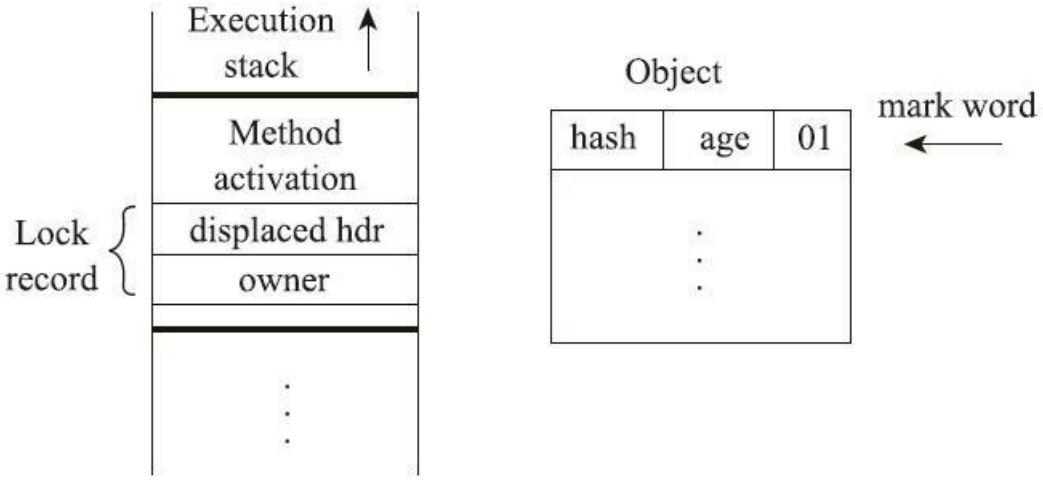
线程在执行同步块之前，JVM 会先在当前线程的栈帧中创建用于存储锁记录的空间，并将对象头中的 Mark Word 复制到锁记录中，官方称为 Displaced Mark Word。然后线程尝试使用 CAS 将对象头中的 Mark Word 替换为指向锁记录的指针。如果成功，当前线程获得锁，如果失败，表示其他线程竞争锁，当前线程便尝试使用自旋来获取锁。

轻量级解锁时，会使用原子的 CAS 操作将 Displaced Mark Word 替换回到对象头，如果成功，则表示没有竞争发生。如果失败，表示当前锁存在竞争，锁就会膨胀成重量级锁。

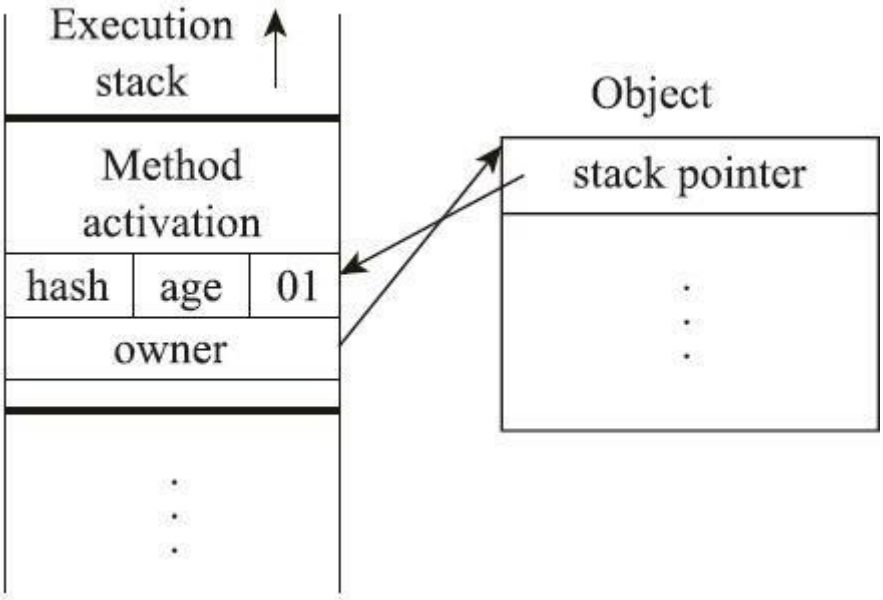
因为自旋会消耗 CPU，为了避免无用的自旋（比如获得锁的线程被阻塞住了），一旦锁升级成重量级锁，就不会再恢复到轻量级锁状态。当锁处于这个状态下，其他线程试图获取锁时，都会被阻塞住，当持有锁的线程释放锁之后会唤醒这些线程，被唤醒的线程就会进行新一轮的夺锁之争。

（下面内容是《深入理解 Java 虚拟机》中的知识补充）

在代码即将进入同步块的时候，如果锁对象没有被锁定（锁标志位为 01），虚拟机首先将在当前线程的栈帧中建立一个名为锁记录（Lock Record）的空间，用于存储锁对象目前 Mark Word 的拷贝（Displaced Mark Word）：



然后，虚拟机将用 CAS 操作尝试将对象的 Mark Word 更新为指向 Lock Word 的指针。如果更新操作成功了，就代表该线程拥有了这个对象的锁，并且对象 Mark Word 的锁标志位将转为 00，表示该对象处于轻量级锁状态。



如果这个更新操作失败了，说明至少有一条线程与当前线程竞争获取该对象的锁。虚拟机首先会检查对象的 Mark Word 是否指向当前线程的栈帧，如果是，说明当前线程已经拥有了这个对象的锁，那直接进入同步块继续执行就可以了，否则就说明这个锁对象已经被其他线程抢占了。如果出现两条以上的线程争用同一个锁的情况，那轻量级锁就不再有效，必须要膨胀为重量级锁，锁标志的状态值变为“10”，此时 Mark Word 中存储的就是指向重量级锁（互斥量）的指针，后面等待锁的线程也必须进入阻塞状态。

轻量级锁的解锁过程也同样是通过 CAS 操作来进行的，如果对象的 Mark Word 仍然指向线程的锁记录，那就用 CAS 操作把对象当前的 Mark Word 和线程中复制的 Displaced Mark Word 替换回来。假如能够成功替换，那整个同步过程就顺利完成了；如果替换失败，则说明有其他线程尝试过获取该锁，就要在释放锁的同时，唤醒被挂起的线程。

6. 锁的优缺点对比

锁	优　　点	缺　　点	适用场景
偏向锁	加锁和解锁不需要额外的消耗，和执行非同步方法相比仅存在纳秒级的差距	如果线程间存在锁竞争，会带来额外的锁撤销的消耗	适用于只有一个线程访问同步块场景
轻量级锁	竞争的线程不会阻塞，提高了程序的响应速度	如果始终得不到锁竞争的线程，使用自旋会消耗 CPU	追求响应时间 同步块执行速度非常快
重量级锁	线程竞争不使用自旋，不会消耗 CPU	线程阻塞，响应时间缓慢	追求吞吐量 同步块执行速度较长

7. 锁升级总结

锁的升级过程：

1）当线程 A 进入同步代码块之前，先检查锁对象头 Mark Word 中的线程 ID 是否与线程 A 一致，如果一致（说明还是线程 A 拿到锁），无需进行 CAS 操作加锁。

2）如果不一致，检查 Mark Word 的锁标志是否为偏向锁，如果不是偏向锁，就自旋等待锁释放（自旋默认 10 次）。

3）如果是偏向锁，判断锁对象记录的该线程是否存在（因为偏向锁不会主动释放锁，所以 Mark Word 中还是会有当前线程 ID 记录）。如果当前线程不存在，则设置 Mark Word 中线程 ID 为线程 A 的 ID。

4）如果当前线程还存在，在线程到达全局安全点时，暂停该当前线程，同时升级为轻量级锁（将锁标志位设置为 00，同时 Mark Word 复制一份进当前的栈帧中，同时设置锁对象头的 Mark Word 更新为指向当前线程栈中 Lock Record 的指针）。线程 A 自旋等待锁释放。

5）如果自旋次数（默认是 10 次）到了，当前线程还是没有释放锁，或者出现线程 B 来和 A 一起竞争这个锁对象，这时轻量级锁升级为重量级锁。阻塞住 A 和 B。

6）变成重量级锁后，如果当前线程释放锁，就唤醒所有阻塞线程，重新竞争锁。

2.3 原子操作的实现原理

原子（atomic）本意是“不能被进一步分割的最小粒子”，而原子操作（atomic operation）意为“不可被中断的一个或一系列操作”。

32 位 IA-32 处理器使用基于对缓存加锁或总线加锁的方式来实现多处理器之间的原子操作。首先处理器会自动保证基本的内存操作的原子性。处理器保证从系统内存中读取或者写入一个字节是原子的，意思是当一个处理器读取一个字节时，其他处理器不能访问这个字节的内存地址。但是复杂的内存操作处理器是不能自动保证其原子性的，比如跨总线宽度、跨多个缓存行和跨页表的访问。但是，处理器提供总线锁定和缓存锁定两个机制来保证复杂内存操作的原子性。

（1）使用总线锁保证原子性

如果多个处理器同时对共享变量进行读改写操作（比如 i++），那么共享变量就会被多个处理器同时进行操作，这样读改写操作就不是原子的，操作完的数值会和期望的不一样。

想要保证读改写共享变量的操作是原子的，就必须保证一个处理器去读改写共享变量的时候，另外的处理器不能操作缓存了该共享变量内存地址的缓存（工作内存缓存）。

总线锁就是处理器在读改写共享变量的时候发出 lock 信号（该信号作用在 2.1.1 里有说明），当该处理器在总线上输出此信号时，其他处理器的请求将会被阻塞住，就能保证该处理器独享公共内存。

（2）使用缓存锁保证原子性

在同一时刻，我们只需要保证对某个内存地址的操作是原子性的，但是总线锁的方式把 CPU 和内存之间的通信锁住了，导致开销比较大。在某些场合，使用缓存锁定更好。

在 CPU 中，频繁使用的内存会缓存在高速缓存（L1，L2，L3）中，因此原子操作就可以在处理器内部缓存中进行（一般 L3 是公有的）。如果内存区域被缓存到缓存行里，并且被锁定，那么当它执行锁操作回写到内存时，处理器不需要声明 lock 信号，只需要修改内部缓存的内存地址（这个对缓存的操作满足 MESI，修改内部缓存内存地址是因为当对缓存行数据进行操作时，其他处理器缓存行会失效 I，具体看十一章）。

但是有两种情况下处理器不会使用缓存锁定。

第一种情况是：当操作的数据不能被缓存在处理器内部，或操作的数据跨多个缓存行（cache line）时，则处理器会调用总线锁定。

第二种情况是：有些处理器不支持缓存锁定。

在 Java 中，是通过锁和循环 CAS 来实现原子操作。

在 JVM 里，CAS 操作利用了处理器提供的 CMPXCHG 指令实现。自旋 CAS 基本思路就是循环进行 CAS 操作直到成功。从 JDK 5 开始，JDK 并发包里提供了一些类来支持原子操作。

CAS 操作包含一个内存位置 V，预期值 A1 和新值 B1。在 Java 中，内存位置为 this 对象，每次用 A1 去和内存位置的值去匹配。循环 CAS 就是当匹配相同时，就将 B1 替换成内存位置的值；如果匹配不相同，就将内存位置的值 A2 替换预期值 A1，现在新的预期值就是 A2，再计算出一个新值 B2，再去匹配。

在 CAS 实现原子操作中，会出现三大问题：

（1）ABA 问题。

如果在进行操作时，有一个值原来是 A，变成了 B，又变成了 A，在 CAS 操作中发现没有变化，实际上是变化了。解决思路就是在变量前面加上版本号，每次操作时，把版本号加一，这样就变成了 A1-B2-A3。

JDK5 开始，Atomic 包里提供了一个类 AtomicStampedReference 来解决 ABA 问题。这个类的 compareAndSet 方法的作用是首先检查当前引用是否等于预期引用，并且检查当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。

（2）循环时间长，开销大。

自旋 CAS 如果长时间不成功，会给 CPU 带来非常大的执行开销。

（3）只能保证一个共享变量的原子操作。

当对一个共享变量执行操作时，我们可以使用循环 CAS 的方式来保证原子操作，但是对多个共享变量操作时，循环 CAS 就无法保证操作的原子性，这个时候就可以用锁。还有一个取巧的办法，就是把多个共享变量合并成一个共享变量来操作。比如，有两个共享变量 i＝2，j=a，合并一下 ij=2a，然后用 CAS 来操作 ij。

而锁机制保证了只有获得锁的线程才能够操作锁定的内存区域。JVM 内部实现了很多种锁机制，有偏向锁、轻量级锁和互斥锁。有意思的是除了偏向锁，JVM 实现锁的方式都用了循环 CAS（这句话是书上原话，实际上重量级锁并不是用到了 CAS，这里应该只是说的轻量级锁相关），即当一个线程想进入同步块的时候使用循环 CAS 的方式来获取锁，当它退出同步块的时候使用循环 CAS 释放锁。

三、Java 内存模型

3.1 Java 内存模型的基础

3.1.1 并发编程的两个关键问题

在并发编程中，需要处理两个关键问题：线程之间如何通信及线程之间如何同步（这里的线程是指并发执行的活动实体）。

通信是指线程之间以何种机制来交换信息。在命令式编程中，线程之间的通信机制有两种：共享内存和消息传递。

在共享内存的并发模型里，线程之间共享程序的公共状态，通过写-读内存中的公共状态进行隐式通信。在消息传递的并发模型里，线程之间没有公共状态，线程之间必须通过发送消息来显式进行通信。

同步是指程序中用于控制不同线程间操作发生相对顺序的机制。在共享内存并发模型里，同步是显式进行的。程序员必须显式指定某个方法或某段代码需要在线程之间互斥执行。

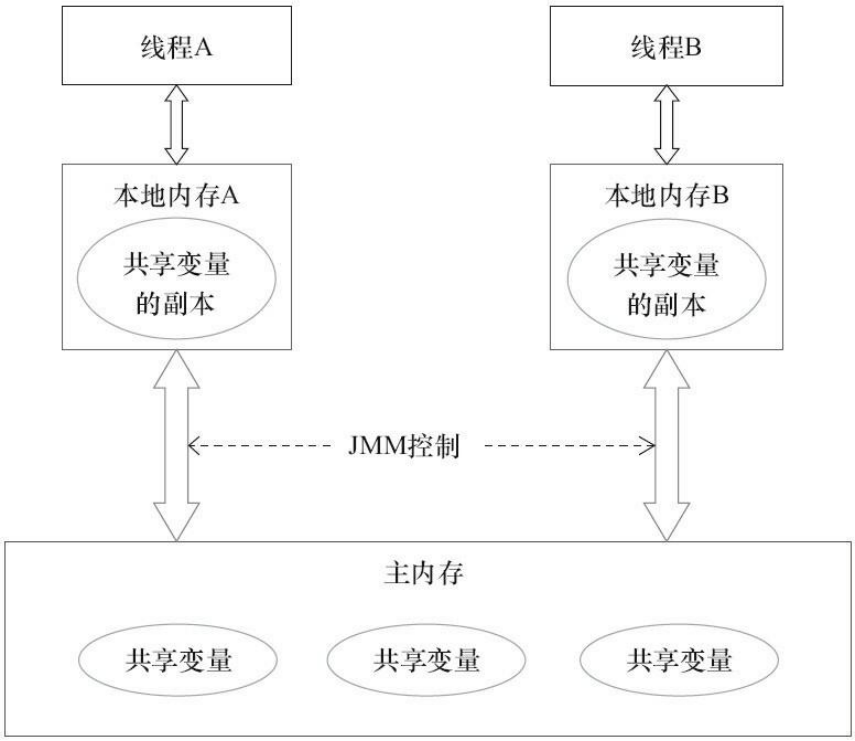
在消息传递的并发模型里，由于消息的发送必须在消息的接收之前，因此同步是隐式进行的。

Java 的并发采用的是共享内存模型，Java 线程之间的通信总是隐式进行，整个通信过程对程序员完全透明。

3.1.2 Java 内存模型的抽象结构

在 Java 中，所有实例域、静态域和数组元素都存储在堆内存中，堆内存在线程之间共享。局部变量（Local Variables），方法定义参数（Java 语言规范称之为 Formal Method Parameters）和异常处理器参数（Exception Handler Parameters）不会在线程之间共享，它们不会有内存可见性问题，也不受内存模型的影响。

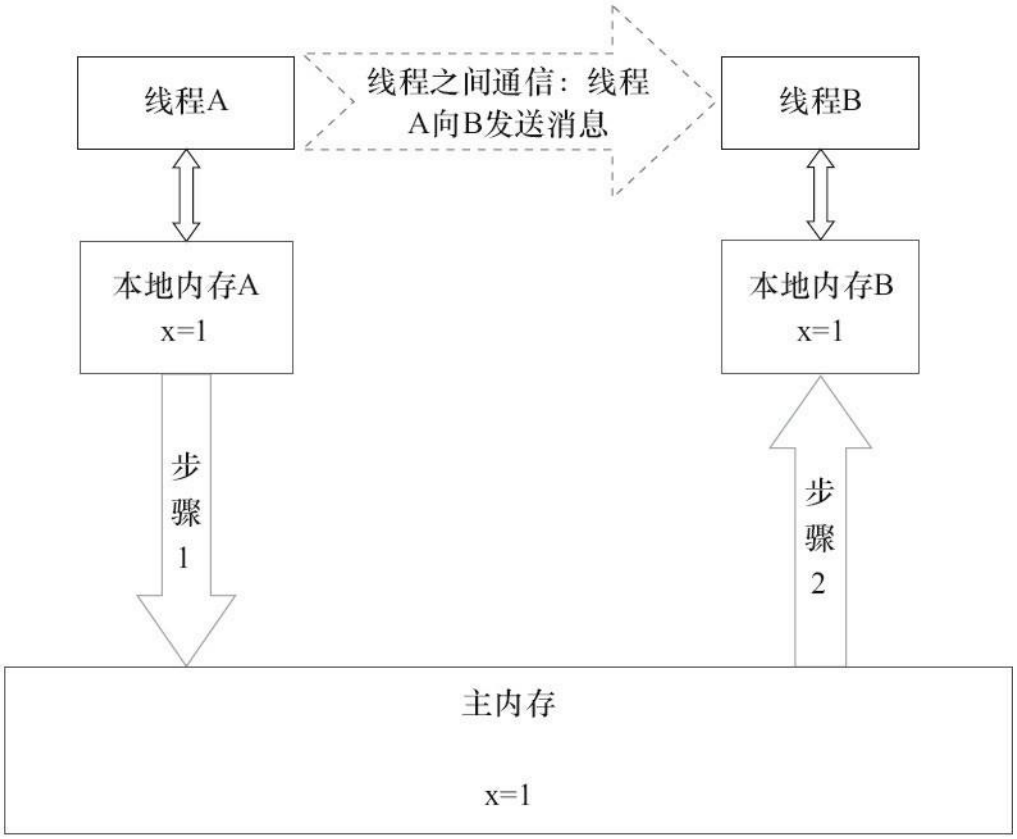
Java 线程之间的通信由 Java 内存模型控制，JMM 决定一个线程对共享变量的写入何时对另一个线程可见。从抽象的角度来看，JMM 定义了线程和主内存之间的抽象关系：线程之间的共享变量存储在主内存（Main Memory）中，每个线程都有一个私有的本地内存（Local Memory），本地内存中存储了该线程以读/写共享变量的副本。本地内存是 JMM 的一个抽象概念，并不真实存在。它涵盖了缓存、写缓冲区、寄存器以及其他的硬件和编译器优化。



从上图看，线程 A 和线程 B 要通信的话：

1）线程 A 把本地内存 A 中更新过的共享变量刷新到主内存中。

2）线程 B 到主内存去读取线程 A 更新过的共享变量。



如图所示，本地内存 A 和本地内存 B 由主内存中共享变量 x 的副本。假设初始时，这 3 个内存中的 x 值都为 0。线程 A 在执行时，把更新后的 x 值（假设值为 1）临时存放在自己的本地内存 A 中。当线程 A 和线程 B 需要通信时，线程 A 首先会把自己本地内存中修改后的 x 值刷新到主内存中，此时主内存中的 x 值变为了 1。随后，线程 B 到主内存中去读取线程 A 更新后的 x 值，此时线程 B 的本地内存的 x 值也变为了 1。

从整体来看，这两个步骤实质上是线程 A 在向线程 B 发送消息，而且这个通信过程必须要经过主内存。JMM 通过控制主内存与每个线程的本地内存之间的交互，来为 Java 程序员提供内存可见性保证。

3.1.3 从源代码到指令序列的重排序

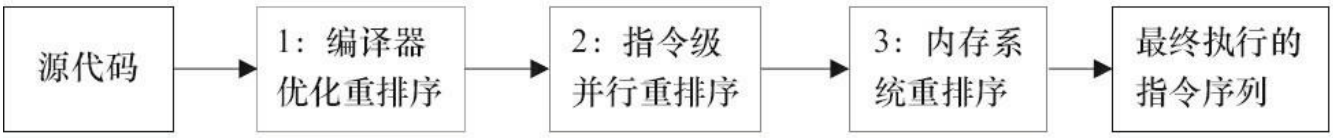
在执行程序时，为了提高性能，编译器和处理器常常会对指令做重排序。

1）编译器优化的重排序。编译器在不改变单线程程序语义的前提下，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(Instruction-Level Parallelism，ILP）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，会经历下面三种重排序：



上面的 1 属于编译器重排序，2，3 是处理器重排序。这些重排序可能会导致多线程程序出现内存可见性问题。

对于编译器，JMM 的编译器重排序规则会禁止特定类型的编译器重排序。对于处理器重排序，JMM 的处理器重排序规则会要求 Java 编译器在生成指令序列时，插入特定类型的内存屏障，通过内存屏障指令来禁止特定类型的处理器重排序。

3.1.4 并发编程模型的分类

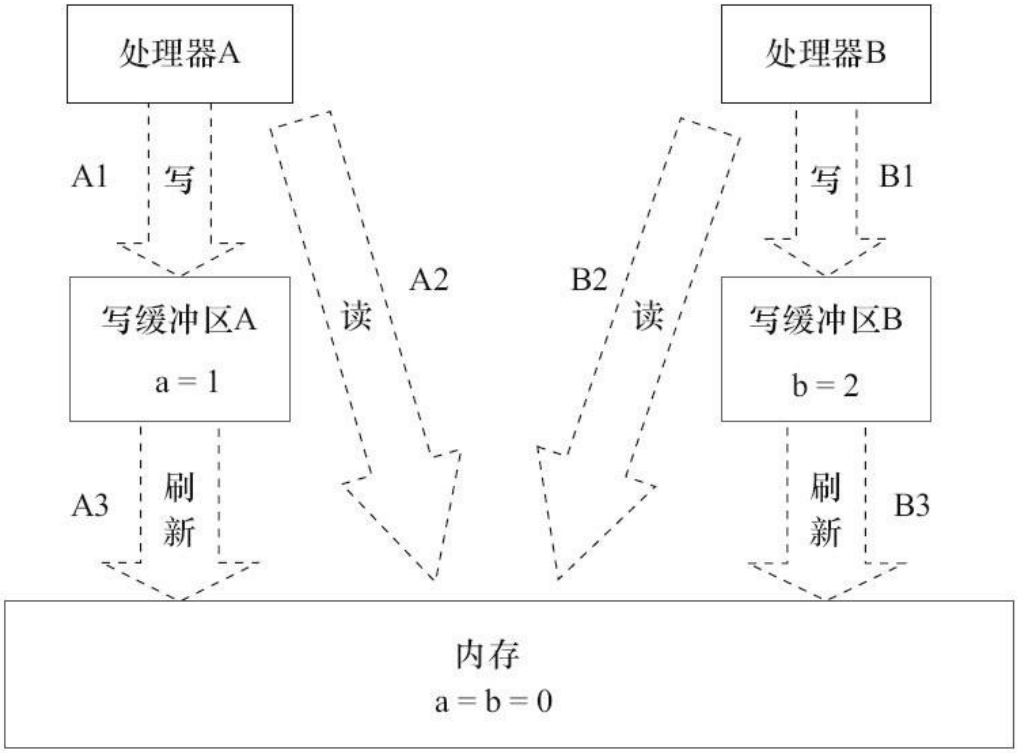
现在处理器使用的写缓冲区来临时保存向内存写入的数据。但是每个处理器的写缓冲区只对每个处理器可见。因此，处理器对内存的读写操作的执行顺序，不一定与内存实际发生的读写顺序一致。

处理器 / 示例项	Processor A	Processor B

代码	a = 1; //A1
	x = b; //A2
		b = 2; //B1
		y = a; //B2

运行结果	初始状态：a = b = 0 处理器允许执行后得到结果：x = y = 0	

上面出现该结果的原因如图：处理器 A 和 B 可以同时把共享变量写入自己的写缓冲区（A1，B1）中，然后从内存中读取另一个共享变量（A2，B2），最后才把自己写缓冲区的数据刷新到内存中。



从内存操作实际发生的顺序来看，直到处理器 A 执行 A3 来刷新自己的写缓冲区，写操作 A1 才算真正执行了。虽然处理器 A 执行内存操作的顺序为：A1→A2，但内存操作实际发生的顺序却是 A2→A1。此时，处理器 A 的内存操作顺序被重排序了。

由于写缓冲区仅对自己的处理器可见，它会导致处理器执行内存操作的顺序可能会与内存实际的操作执行顺序不一致。即，现代处理器都会允许对写读（普通）操作进行重排序。

屏障类型	指令示例	说　　明
LoadLoad Barriers	Load1; LoadLoad; Load2	确保 Load1 数据的装载先于 Load2 及所有后续装载指令的装载
StoreStore Barriers	Store1; StoreStore; Store2	确保 Store1 数据对其他处理器可见（刷新到内存）先于 Store2 及所有后续存储指令的存储
LoadStore Barriers	Load1; LoadStore; Store2	确保 Load1 数据装载先于 Store2 及所有后续的存储指令刷新到内存
StoreLoad Barriers	Store1; StoreLoad; Load2	确保 Store1 数据对其他处理器变得可见（指刷新到内存）先于 Load2 及所有后续装载指令的装载。StoreLoad Barriers 会使该屏障之前的所有内存访问指令（存储和装载指令）完成之后，才执行该屏障之后的内存访问指令

StoreLoad Barriers 是一个“全能型”的屏障，它同时具有其他 3 个屏障的效果。现代的多处理器大多支持该屏障（其他类型的屏障不一定被所有处理器支持）。执行该屏障开销会很昂贵，因为当前处理器通常要把写缓冲区中的数据全部刷新到内存中（Buffer Fully Flush）

3.1.5 happens-before 的基础

从 JDK 5 开始，Java 使用新的 JSR-133 内存模型。JSR-133 使用 happens-before 的概念来阐述操作之间的内存可见性。

如果一个操作执行的结果需要对另一个操作可见，那么这两个操作之间必须要存在 happens-before 关系。这里提到的两个操作既可以是在一个线程之内，也可以是在不同线程之间。

与程序员密切相关的 happens-before 规则如下。

1. 程序顺序规则：一个线程中的每个操作，happens-before 于该线程中的任意后续操作。

2. 监视器锁规则：对一个锁的解锁，happens-before 于随后对这个锁的加锁。

3. volatile 变量规则：对一个 volatile 域的写，happens-before 于任意后续对这个 volatile 域的读。

4. 传递性：如果 A happens-before B，且 B happens-before C，那么 A happens-before C。

两个操作之间具有 happens-before 关系，并不意味着前一个操作必须要在后一个操作之前执行！happens-before 仅仅要求前一个操作（执行的结果）对后一个操作可见，且前一个操作按顺序排在第二个操作之前。

3.2 重排序

3.2.1 数据依赖性

如果两个操作访问同一个变量，且这两个操作中有一个为写操作，此时这两个操作之间就存在数据依赖性。

名　称	代码示例	说　明
写后读	a = 1; b = a;	写一个变量之后，再读这个位置
写后写	a = 1; a = 2;	写一个变量之后，再写这个变量
读后写	a = b; b = 1;	读一个变量之后，再写这个变量

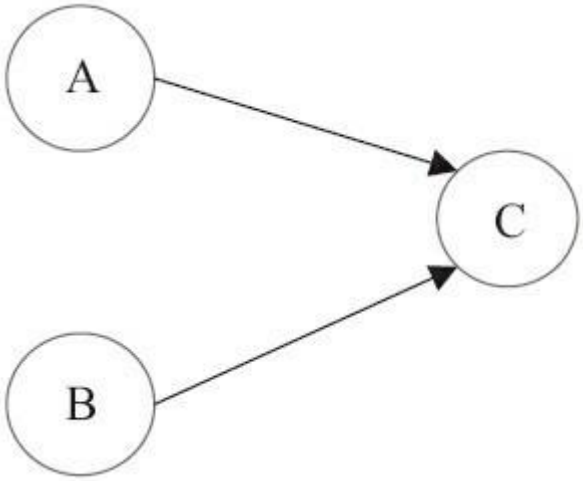
上面三种情况，只要重排序两个操作的执行顺序，程序的执行结果就会被改变。编译器和处理器可能会对操作做重排序。编译器和处理器在重排序时，会遵守数据依赖性，编译器和处理器不会改变存在数据依赖关系的两个操作的执行顺序。

这里所说的数据依赖性仅针对单个处理器中执行的指令序列和单个线程中执行的操作，不同处理器之间和不同线程之间的数据依赖性不被编译器和处理器考虑。

3.2.2 as-if-serial 语义

as-if-serial 语义的意思是：不管怎么重排序（编译器和处理器为了提高并行度），（单线程） 程序的执行结果不能被改变。编译器、runtime 和处理器都必须遵守 as-if-serial 语义。

为了遵守 as-if-serial 语义，编译器和处理器不会对存在数据依赖关系的操作做重排序，因为这种重排序会改变执行结果。但是，如果操作之间不存在数据依赖关系，这些操作就可能被编译器和处理器重排序。

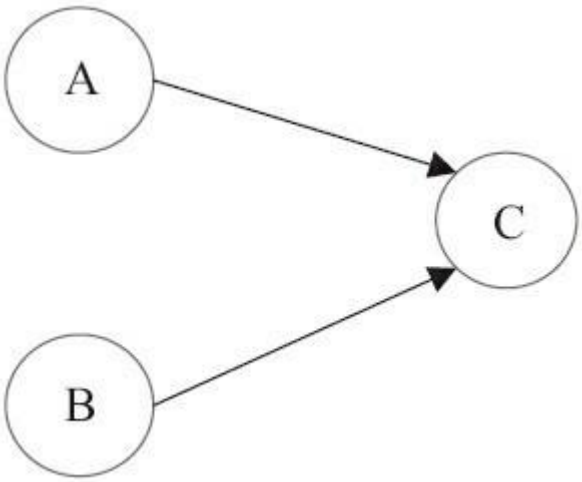


如图所示，当 A-C，B-C 之间存在数据依赖关系，A-B 之间没有依赖关系。根据 ais（as-if-serial，后面都简写，happens-before 为 hb），A-C，B-C 之间不能重排序，即 C 必须在 A 和 B 的后面。编译器可以对 A-B 做重排序，没有依赖关系，最终单线程输出的值不会改变。

as-if-serial 语义把单线程程序保护了起来，遵守 as-if-serial 语义的编译器、runtime 和处理器共同为编写单线程程序的程序员创建了一个幻觉：单线程程序是按程序的顺序来执行的。as- if-serial 语义使单线程程序员无需担心重排序会干扰他们，也无需担心内存可见性问题。

总结：单线程下因为遵守 ais，没有依赖关系间的程序无论怎么重排序都不会影响最终的结果。

3.2.3 程序顺序规则



这张图用 happens-before 的程序顺序规则。假设 A 先执行：

1）A-hp-B

2）B-hp-C

3）A-hp-C

这里 A-hp-B，但是实际执行的时候可能 B-hp-A。如果 A-hp-B，JMM 不一定要求 A 一定要在 B 之前执行。JMM 仅仅要求前一个操作（执行的结果）对后一个操作可见，且前一个操作按顺序排在第二个操作前面。这里操作 A 的执行结果不需要对操作 B 可见，而且重排序操作 A 和操作 B 后的执行结果，与操作 A 和操作 B 按 happens-before 顺序执行的结果一致。在这种情况下，JMM 认为这种重排序并不非法，允许该重排序。

书上这里写的有点绕，我仔细的看了下，可能是想说，

1）在单线程中（因为说的是程序顺序规则），假设 A 先执行，根据程序顺序规则，A-hp-B。

2）又因为 A 和 B 不存在数据依赖性（这个和 hp 无关），处理器和编译器进行重排序可能会变成 B-hp-A。

3）根据 hp 的关系概念，hp 仅仅要求前一个操作（执行的结果）对后一个操作可见，但是 A 的执行结果不需要对 B 可见。

4）虽然不满足 hp，但是 JMM 也认同了这种重排序。

总结：虽然在单线程里，写的代码顺序根据 hp 程序顺序规则来说是符合前一个操作在后续任意操作前面（见 3.1.5），但是如果两个操作的重排序结果与按照 hp 后的结果是一样的，那 JMM 允许这个重排序。用官方的话说：

两个操作之间存在 happens-before 关系，并不意味着 Java 平台的具体实现必须要按照 happens-before 关系指定的顺序来执行。如果重排序之后的执行结果，与按 happens-before 关系来执行的结果一致，那么这种重排序并不非法（也就是说，JMM 允许这种重排序）。

（这个也是 hp 的第二个规则，第一个规则是本节第一个蓝色部分）

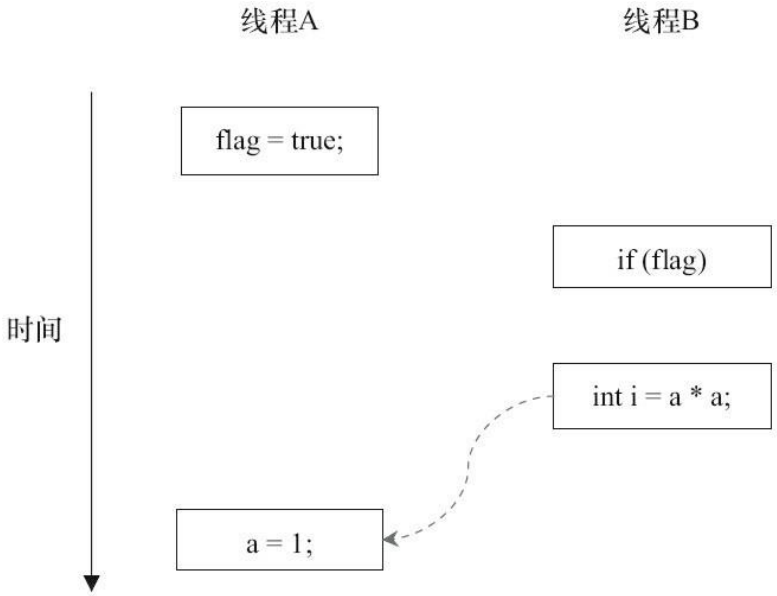
3.2.4 重排序对多线程的影响

```
class ReorderExample {
       int a = 0;
       boolean flag = false;
       public void writer() {
           a = 1; // 1
           flag = true;  // 2
       }
       public void reader() {
           if (flag) {      // 3
           int i = a * a;   // 4
           ……
           }
       }
}
```

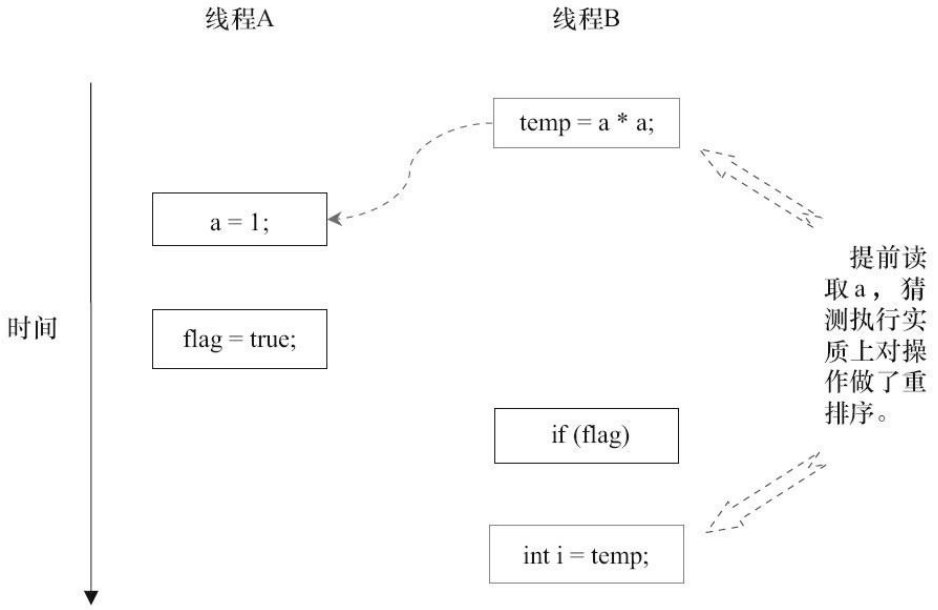
假设上面代码有两个线程 A，B.A 首先执行 writer()方法，然后 B 执行 reader()方法。线程 B 在在执行 4 的时候，不一定看到 A 在操作 1 对 a 的写入。

操作 1 和 2 没有数据依赖关系，因此编译器和处理器可能会对这两个操作做重排序；3,4 也没有数据依赖关系，所以可能也会做重排序：

1）当 1,2 做重排序时，A 写入 flag，然后 B 读到 flag，接着 B 将之前 a 的值用来计算结果。最后 A 才把 a 写入内存。



2）当 3,4 重排序时，在程序中，操作 3 和操作 4 存在控制依赖关系。当代码中存在控制依赖性时，会影响指令序列执行的并行度。为此，编译器和处理器会采用猜测（Speculation）执行来克服控制相关性对并行度的影响。以处理器的猜测执行为例，执行线程 B 的处理器可以提前读取并计算 a*a，然后把计算结果临时保存到一个名为重排序缓冲（Reorder Buffer，ROB）的硬件缓存中。当操作 3 的条件判断为真时，就把该计算结果写入变量 i 中。



在单线程程序中，对存在控制依赖的操作重排序，不会改变执行结果（这也是 as-if-serial 语义允许对存在控制依赖的操作做重排序的原因）；但在多线程程序中，对存在控制依赖的操作重排序，可能会改变程序的执行结果。

3.3 顺序一致性

3.3.1 数据竞争与顺序一致性

当程序未正确同步时，就可能会存在数据竞争。Java 内存模型规范对数据竞争的定义如下。在一个线程中写一个变量，在另一个线程读同一个变量，而且写和读没有通过同步来排序。

当代码中出现数据竞争时，程序的执行往往产生违反直觉的结果。如果一个多线程程序能正确同步，这个程序将是一个没有数据竞争的程序。

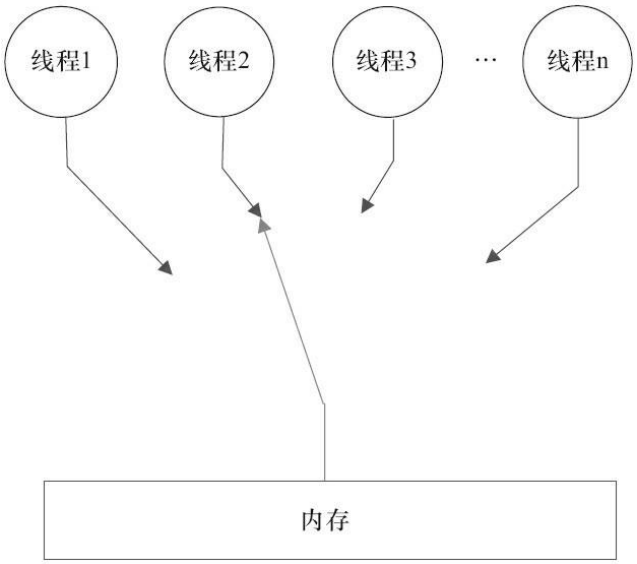
JMM 对正确同步的多线程程序的内存一致性做了如下保证：如果程序是正确同步的，程序的执行将具有顺序一致性（Sequentially Consistent）——即程序的执行结果与该程序在顺序一致性内存模型中的执行结果相同。

3.3.2 顺序一致性内存模型

顺序一致性内存模型是一个被计算机科学家理想化了的理论参考模型，它为程序员提供了极强的内存可见性保证。顺序一致性模型有两大特性：

1）一个线程中的所有操作必须按照程序的顺序来执行

2）（不管程序是否同步）所有线程都只能看到一个单一的操作执行顺序。在顺序一致性内存模型中，每个操作都必须原子执行且立刻对所有线程可见。

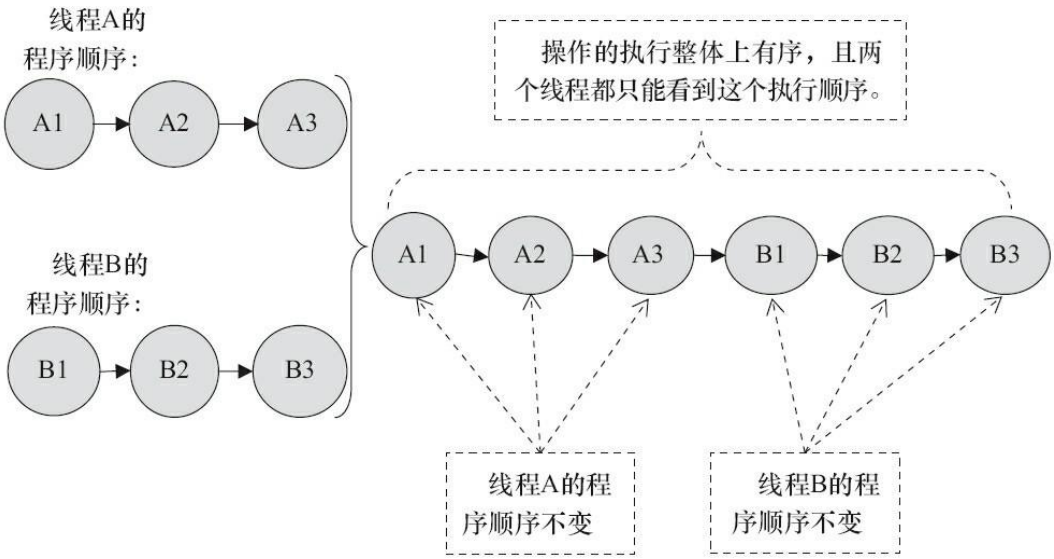


在概念上，顺序一致性模型有一个单一的全局内存，这个内存通过一个左右摆动的开关可以连接到任意一个线程，同时每一个线程必须按照程序的顺序来执行内存读/写操作。如图所示，在任意一个时间点只能有一个线程可以连接到内存。当多个线程并发执行时，图中的开关装置能把所有线程的所有内存读/写操作串行化（即在顺序一致性模型中，所有操作之间具有全序关系）。

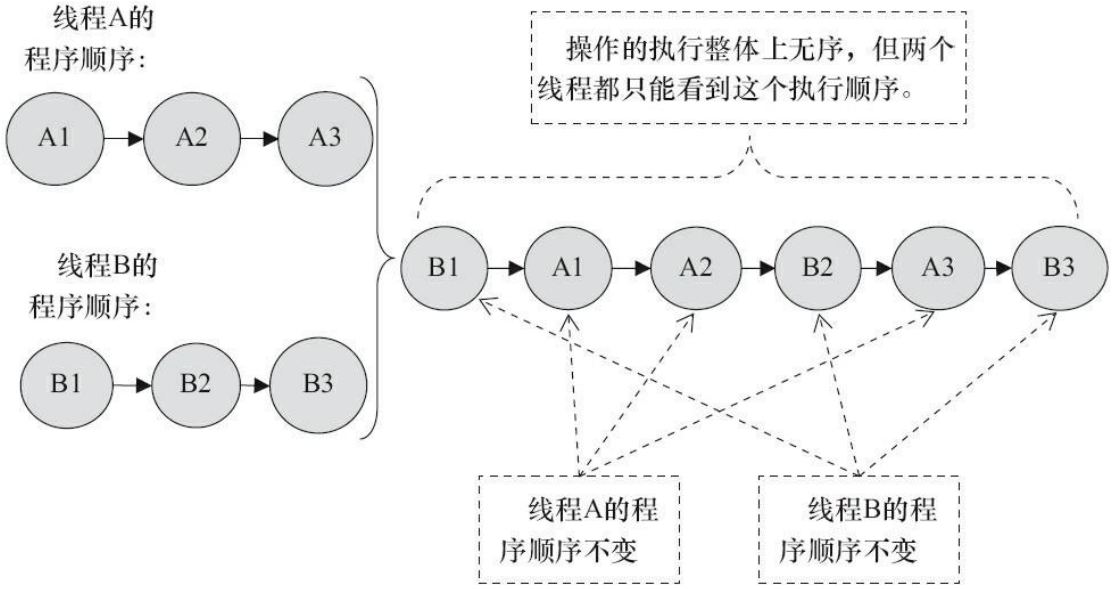
（这段不是很好理解到书上的意思，结合下面猜想书上是想说，顺序一致性模型的特点，这个特点和 JMM 不一样，JMM 要通过正确的同步操作来达到顺序一致性）

假设有两个线程 A 和 B 并发执行。其中 A 线程有 3 个操作，它们在程序里的顺序是 A1-A2-A3，B 也有 3 个操作，B1-B2-B3。

1）假设这两个线程使用监视器锁来保证正确同步：A 线程 3 个操作后释放监视器锁，虽然 B 线程获取同一个监视器锁，如图所示：



2）假设这两个线程没有同步：



未同步程序在顺序一致性模型中虽然整体执行顺序是无序的，但所有线程都只能看到一个一致的整体执行顺序。线程 A 和 B 看到的顺序都是 B1,A1,A2,B2,A3,B3。因此，在顺序一致性模型中，每个操作必须立即对任意线程可见。

但是在 JMM 中就没有这个保证。未同步的程序在 JMM 中不但整体的执行顺序是无序的，而且所有线程看到的操作执行顺序也可能不一致。比如，在当前线程把写过的数据缓存在本地内存中，在没有刷新到主内存之前，这个写操作仅对当前线程可见；从其他线程的角度来观察，会认为这个写操作根本没有被当前线程执行。只有当前线程把本地内存中写过的数据刷新到主内存之后，这个写操作才能对其他线程可见。在这种情况下，当前线程和其他线程看到的操作执行顺序将不一致。

3.3.3 同步程序的顺序一致性效果

```
class SynchronizedExample {
        int a = 0;
        boolean flag = false;
        public synchronized void writer() {  // 获取锁
                a = 1;
                flag = true;
        }  // 释放锁
        public synchronized void reader() {  // 获取锁
                if (flag) {
                    int i = a;
                    ……
                }  // 释放锁
        }
}
```

在上面示例代码中，假设 A 线程执行 writer()方法后，B 线程执行 reader()方法。这是一个正确同步的多线程程序。根据 JMM 规范，该程序的执行结果将与该程序在顺序一致性模型中的执行结果相同。

顺序一致性模型中，所有操作完全按程序的顺序串行执行。而在 JMM 中，临界区内的代码可以重排序（但 JMM 不允许临界区内的代码“逸出”到临界区之外，那样会破坏监视器的语义）。JMM 会在退出临界区和进入临界区这两个关键时间点做一些特别处理，使得线程在这两个时间点具有与顺序一致性模型相同的内存视图。虽然线程 A 在临界区内做了重排序，但由于监视器互斥执行的特性，这里的线程 B 根本无法“观察”到线程 A 在临界区内的重排序。这种重排序既提高了执行效率，又没有改变程序的执行结果。

JMM 在具体实现上的基本方针为：在不改变（正确同步的）程序执行结果的前提下，尽可能地为编译器和处理器的优化打开方便之门。

3.3.4 未同步程序的执行特性

对于未同步或未正确同步的多线程程序，JMM 只提供最小安全性：线程执行时读取到的值，要么是之前某个线程写入的值，要么是默认值（0，Null，False），JMM 保证线程读操作读取到的值不会无中生有（Out Of Thin Air）的冒出来。为了实现最小安全性，JVM 在堆上分配对象时，首先会对内存空间进行清零，然后才会在上面分配对象（JVM 内部会同步这两个操作）。因此，在已清零的内存空间（Pre-zeroed Memory）分配对象时，域的默认初始化已经完成了

JMM 不保证未同步程序的执行结果与该程序在顺序一致性模型中的执行结果一致。因为如果想要保证执行结果一致，JMM 需要禁止大量的处理器和编译器的优化，这对程序的执行性能会产生很大的影响。而且未同步程序在顺序一致性模型中执行时，整体是无序的，其执行结果往往无法预知。而且，保证未同步程序在这两个模型中的执行结果一致没什么意义。

未同步程序在两个模型中的执行特性有如下几个差异：

1）顺序一致性模型保证单线程内的操作会按程序的顺序执行，而 JMM 不保证单线程内的操作会按程序的顺序执行。

2）顺序一致性模型保证所有线程只能看到一致的操作执行顺序，而 JMM 不保证所有线程能看到一致的操作执行顺序。

3）JMM 不保证对 64 位的 long 型和 double 型变量的写操作具有原子性，而顺序一致性模型保证对所有的内存读/写操作都具有原子性。（第 3 点和处理器总线的工作机制相关，这里就不总结了，大概就是 64 位变量为了减少开销，就拆分成两个 32 位来执行，这两个 32 位可能会被分配到不同的总线事务）

3.4 volatile 的内存语义

3.4.1 volatile 的特性

对 volatile 变量的读写和对该普通变量的读写加锁同步有相同的效果：

```
class VolatileFeaturesExample {
        volatile long vl = 0L;
        public void set(long l) {
            vl = l;
        }
        public void getAndIncrement () {
            vl++;
        }
        public long get() {
            return vl;
        }
}
```

```
class VolatileFeaturesExample {
        long vl = 0L;
        public synchronized void set(long l) {
            vl = l;
        }
        public void getAndIncrement () {
            long temp = get();
            temp += 1L;
            set(temp);
        }
        public synchronized long get() {
            return vl;
        }
}
```

happens-before 的 volatile 规则意味着，对一个 volatile 变量的读，总是能看到（任意线程）对这个 volatile 变量最后的写入；而锁的特性决定了在临界区代码的执行具有原子性，但是对多个 volatile 变量的操作不具有原子性，比如 volatile++。

根据两个特性和代码总结起来，volatile 变量本身具有可见性和原子性。

3.4.2 volatile 写-读建立的 happens-before 关系

针对 volatile，更关心的应该是对线程的内存可见性的影响。

从内存语义的角度来说，volatile 的写-读与锁的释放-获取有相同的内存效果：volatile 写和锁的释放有相同的内存语；读和锁的获取有相同的内存语义。

```
class VolatileExample {
    int   a = 0;
    volatile boolean flag = false;
    public void writer() {
        a = 1;          // 1
        flag = true;    // 2
    }
    public void reader() {
       if (flag) {  // 3
           int i = a;   // 4
           ……
       }
    }
}
```

假设线程 A 执行 writer()方法后，线程 B 执行 reader()方法。根据 happens-before 规则：

1）根据程序顺序规则：1 hp 2；3 hp 4

2）根据 volatile 规则：2 hp 3

3）根据传递性规则：1 hp 4

即，A 线程写一个 volatile 变量后，B 线程读同一个 volatile 变量。A 线程在写 volaile 变量之前所有可见的共享变量（上面例子是 1），在 B 线程读同一个 volatile 变量后（3），将立即变得对 B 线程可见。

3.4.3 volatile 写-读的内存语义

volatile 写的内存语义：当写一个 volatile 变量时，JMM 会把该线程对应的本地内存中的共享变量值刷新到主内存（共享变量（实例字段、静态字段、数组对象元素））。

volatile 读的内存语义：当读一个 volatile 变量时，JMM 会把该线程对应的本地内存置为无效。线程接下来将从主内存中读取共享变量。

将两个操作综合起来看：在读线程 B 读一个 volatile 变量后，写线程 A 在写这个 volatile 变量之前所有可见的共享变量的值都将立即变得对读线程 B 可见。

总结：
线程 A 写一个 volatile 变量，实质上是线程 A 向接下来将要读这个 volatile 变量的某个线程发出了（其对共享变量所做修改的）消息。

线程 B 读一个 volatile 变量，实质上是线程 B 接收了之前某个线程发出的（在写这个 volatile 变量之前对共享变量所做修改的）消息
说明：这里的写这个 volatile 变量之前，根据 JMM 对重排序的处理，写 volatile 变量这个操作如果是第二操作，那么和其他第一操作（比如普通读写，volatile 读写）不能做重排序，但是如果写 volatile 是第一操作，可以和普通读写做重排序，详细见下一节。

线程 A 写一个 volatile 变量，随后线程 B 读这个 volatile 变量，这个过程实质上是线程 A 通过主内存向线程 B 发送消息。

3.4.4 volatile 内存语义的实现

为了实现 volatile 内存语义，JMM 会分别限制编译器和处理器重排序类型。

是否能重排序	第二个操作		
第一个操作	普通读 / 写	volatile 读	volatile 写
普通读 / 写			NO
volatile 读	NO	NO	NO
volatile 写		NO	NO

从表中可以看出：

1）当第二个操作是 volatile 写时，不管第一个操作时什么，都不能重排序。这个规则确保了 volatile 写之前的操作不会被编译器重排序到 volatile 写之后。

2）当第一个操作是 volatile 读时，不管第二个操作是什么，都不能重排序。这个规则确保了 volatile 读之后的操作不会被编译器重排序到 volatile 读之前。

3）当第一个操作时 volatile 写，第二个操作时 volatile 读时，不能重排序。

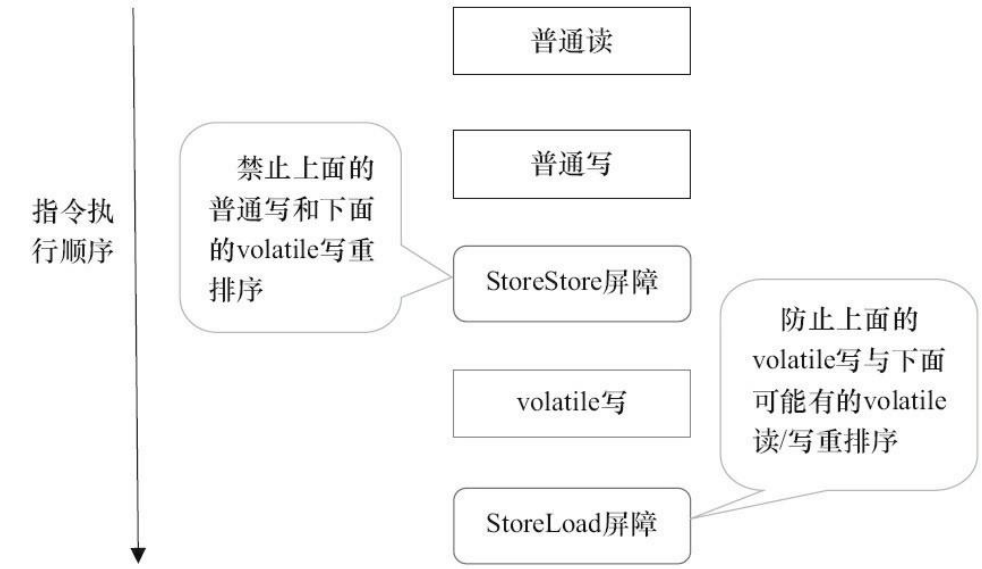
为了达到限制重排序的作用，编译器在生成字节码时，会在指令序列中插入内存屏障来禁止特定类型的处理器重排序。

a.在每个 volatile 写操作的前面插入一个 StoreStore 屏障。

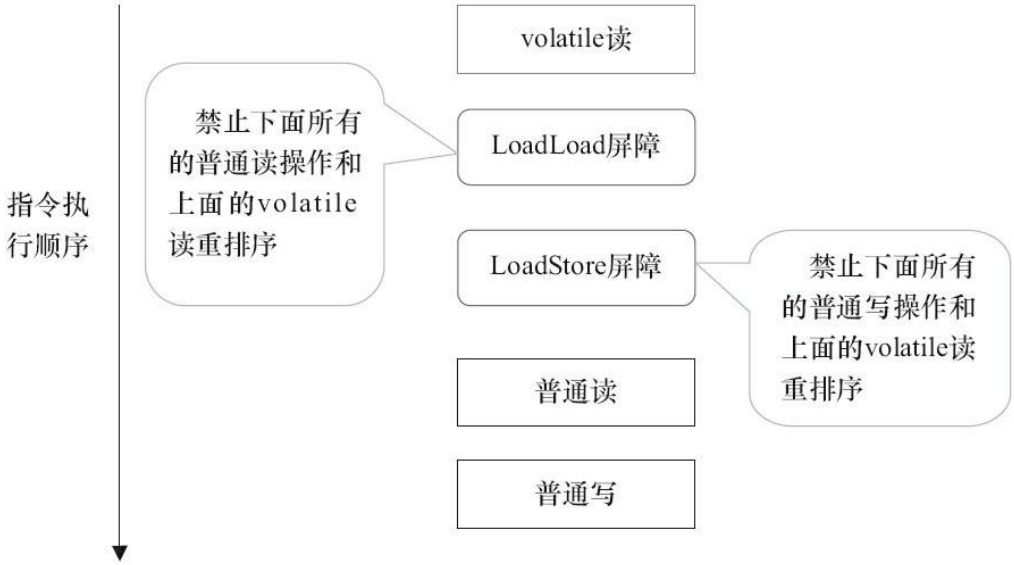
b.在每个 volatile 写操作的后面插入一个 StoreLoad 屏障。

c.在每个 volatile 读操作的后面插入一个 LoadLoad 屏障。

d.在每个 volatile 读操作的后面插入一个 LoadStore 屏障。



因为普通读写作为第一操作不能和 volatile 写做重排序，因此加上 StoreStore 屏障来保证在 volatile 写之前的写操作刷新到主内存。而 volatile 写下面只有 StoreLoad 屏障，防止和下面的 volatile 读写重排序，而 volatile 写作为第一操作是可以和普通读写做重排序。



由于 volatile 读作为第一操作不能和其他操作做重排序，因此做两个屏障来保证不会重排序（volatile 写前后都加了屏障，不会和 volatile 读做重排序）。

实际上，只要不改变 volatile 写-读的内存语义，编译器会省略不必要的屏障。JMM 会首先保证正确性，再追求效率。

这节总结的内存屏障实际上就是 volatile 用来防止指令重排的原理。

3.5 锁的内存语义

3.5.1 锁的释放-获取建立的 happens-before 关系

```
class MonitorExample {
        int a = 0;
        public synchronized void writer() {     // 1
           a++;                                 // 2
        }                                       // 3
        public synchronized void reader() {     // 4
           int i = a;                           // 5
           ……
        }                                       // 6
}
```

假设线程 A 执行 write()方法，随后线程 B 执行 reader()方法。根据 hp 法则：

1）根据程序次序规则，1 hp 2，2 hp 3；4 hp 5,5 hp 6

2）根据监视器锁规则，3 hp 4

3）根据传递性，2 hp 5

即，线程 A 在释放锁之前所有可见的共享变量，在线程 B 获取同一个锁之后，将立刻对 B 线程可见。

3.5.2 锁的释放-获取的内存语义

当线程释放锁时，JMM 会把该线程对应的本地内存中的共享变量刷新到主内存中。

当线程获取锁时，JMM 会把该线程对应的本地内存置为无效。使得被监视器保护的临界区代码必须从主存中读取共享变量。

对比 volatile 写-读内存语义：锁释放与 volatile 写，锁获取与 volatile 读有相同的内存语义。

总结：
线程 A 释放一个锁，实质上是线程 A 向接下来将要获取这个锁的某个线程发出了（线程 A 对共享变量所做修改的）消息。

线程 B 获取一个锁，实质上是线程 B 接收了之前某个线程发出的（在释放这个锁之前对共享变量所做修改的）消息。

线程 A 释放锁，随后线程 B 获取这个锁，这个过程实质上是线程 A 通过主内存向线程 B 发送消息。

3.5.3 锁内存语义的实现

使用书上例子：ReentrantLock

```
class ReentrantLockExample {
    int a = 0;
    ReentrantLock lock = new ReentrantLock();
    public void writer() {
        lock.lock(); // 获取锁
        try {
            a++;
        } finally {
            lock.unlock();  // 释放锁
        }
    }
    public void reader () {
        lock.lock(); // 获取锁
        try {
            int i = a;
            ……
        } finally {
            lock.unlock();  // 释放锁
        }
    }
}
```

ReentrantLock 的实现依赖于 AQS 同步队列。AQS 使用一个整型的 volatile 变量（state）来维护同步状态。

ReentrantLock 分为公平锁和非公平锁。

使用公平锁加锁时：

1）ReentrantLock:lock()

2）FairSync:lock()

3）AbstractQueuedSynchronizer:acquire(int arg)

4）ReentrantLock:tryAcquire(int acquires)

第四步开始加锁。

```
protected final boolean tryAcquire(int acquires){
    final Thread current = Thread.currentThread();
    int c = getState();
    if(c == 0){ ....
```

加锁方法先读 volatile 变量 state。

使用公平锁释放锁时：

1）ReentrantLock:unlock

2）AbstractQueuedSynchronizer:release(int arg)

3）Sync:tryRelease(int releases)

第 3 步真正开始释放锁

```
protected final boolean tryAcquire(int releases){
    int c = getState() - releases;
    ...
    if(c == 0){
    ...
    }
    setState(c)
    return free;
}
```

释放锁的最后写 volatile 变量 state。

公平锁在释放锁的最后写 volatile 变量 state，在获取锁时首先读这个 volatile 变量。根据 volatile 的 hp 规则：释放锁的过程在写 volatile 变量之前可见的共享变量，在获取锁的线程读同一个 volatile 变量后将立即变得对获取锁的线程可见。

使用非公平锁加锁时：

```
protected final boolean compareAndSetState(int expect,int update){
    return
unsafe.compareAndSwapInt(this,stateOffset,expect,update);
}
```

该方法以原子操作的方式（CAS）更新 state 变量。

CAS 防止重排序的原理这里不总结了，大概就是调用 CAS 方法会添加一个 lock 前缀。这个前缀在 intel 手册的说明里具有内存屏障的作用，和 volatile 一样。

总结：

1）公平锁和非公平锁释放时，最后都会写一个 volatile 变量 state。

2）公平锁获取锁时，首先会去读这个 volatile 变量。

3）非公平锁获取时，首先会用 CAS 更新 volatile 变量，这个操作同时具有 volatile 读和写的作用。

即：锁的内存语义包括 volatile 变量的读写或者 CAS 操作附带的 volatile 读写。

3.5.4 concurrent 包的实现

Java 线程之间的通信有下面 4 种方式：
1）A 线程写 volatile 变量，随后 B 线程读这个 volatile 变量

2）A 线程写 volatile 变量，随后 B 线程用 CAS 更新这个 volatile 变量

3）A 线程用 CAS 更新一个 volatile 变量，随后 B 线程用 CAS 更新这个 volatile 变量

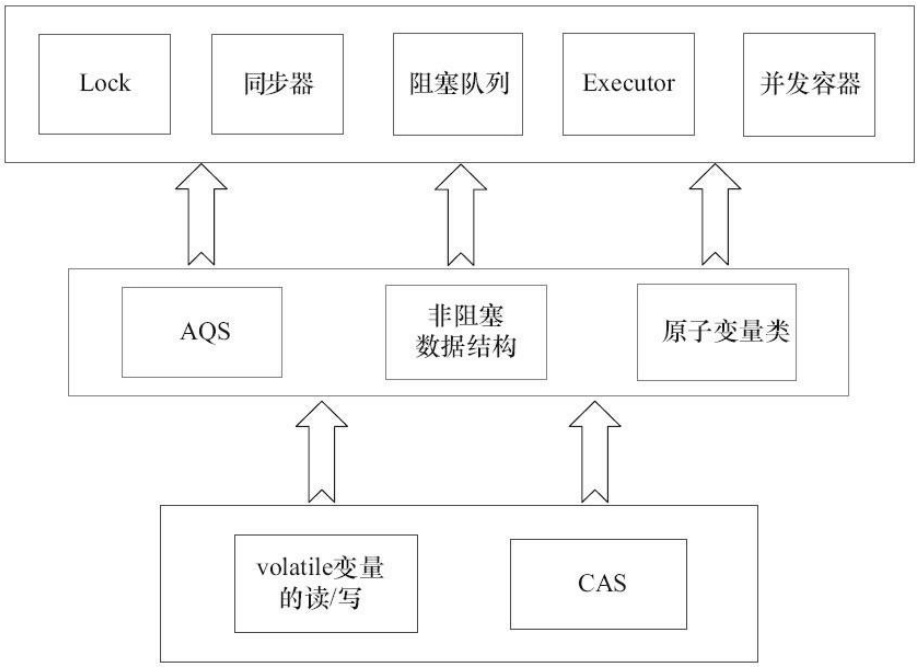
4）A 线程用 CAS 更新一个 volatile 变量，随后 B 线程读这个 volatile 变量

Java 的 CAS 会使用现代处理器上提供的高效机器级别的原子指令，这些原子指令以原子方式对内存执行读-改-写操作，这是在多处理器实现同步的本质。同时，volatile 变量的读写和 CAS 可以实现线程间的通信，这些加在一起就成了 concurrent 包的实现。

concurrent 包基本上包括：
1）声明共享变量为 volatile

2）使用 CAS 的原子条件更新来实现线程之间的同步。

3）配合以 volatile 的读/写和 CAS 所具有的 volatile 读和写的内存语义来实现线程之间的通信。



3.6 final 域的内存语义

3.6.1 final 域的重排序规则

对于 final 域，编译器和处理器要遵守两个重排序规则：
1） 在构造函数内对一个 final 域的写入，与随后把这个被构造对象的引用赋值给一个引用变量，这两个操作之间不能重排序。

2） 初次读一个包含 final 域的对象的引用，与随后初次读这个 final 域，这两个操作之间不能重排序。

3.6.2 写 final 域的重排序规则

```
public class FinalExample{
    int i;
    final int j;
    static FinalExample obj;
    public FinalExample(){
        i = 1;
        j = 2;
    }
    public static void writer(){
        obj = new FinalExample();
    }
    public static void reader(){
        FinalExample object = obj;
        int a = object.i;
        int b = object.j;
    }
}
```

假设一个线程 A 执行 writer()方法，线程 B 执行 reader()方法。

在 writer()方法里，包含构造一个 FinalExample 类型的对象，然后将该对象的引用赋值给 obj。当 B 线程去读这个引用的时候，如果违反了 final 重排序的第一规则，即先赋值一个引用，再写 final 变量，就会产生：obj 还没赋值，线程 B 读取的是初始化的 final 值 0。

即第一规则保证了，在对象引用为任意可见之前，对象的 final 域已经正确初始化了。当然，普通变量没有这个保证。很可能 B 线程读取到 i 的值是 0。

3.6.3 读 final 域的重排序规则

```
public class FinalExample{
    int i;
    final int j;
    static FinalExample obj;
    public FinalExample(){
        i = 1;
        j = 2;
    }
    public static void writer(){
        obj = new FinalExample();
    }
    public static void reader(){
        FinalExample object = obj;
        int a = object.i;
        int b = object.j;
    }
}
```

大多数处理器不会重排序初次读对象引用和初次读该对象包含的 final 域（上面例子就是 reader 里的第一行和第三行），但是还是有少量处理器会去重排序。

读 final 域的重排序规则可以确保：在读一个对象的 final 域之前，一定会先读包含这个 final 域的对象的引用。

3.6.4 final 域为引用类型

假设 final 域为引用类型，在满足基本类型的两个条件下增加了下面一个约束：

在构造函数内对一个 final 引用的对象的成员域的写入，与随后在构造函数外把这个被构造对象的引用赋值给一个引用变量，这两个操作不能重排序。

3.6.5 final 语义在处理器中的实现

写 final 域的重排序规则会要求编译器在 final 域的写之后，构造函数 return 之前插入一个 StoreStore 屏障。

读 final 域的重排序规则要求编译器在读 final 域的操作前面插入一个 LoadLoad 屏障。

由于 X86 处理器不会对写-写操作做重排序，所以在 X86 处理器中，写 final 域需要的 StoreStore 障屏会被省略掉。同样，由于 X86 处理器不会对存在间接依赖关系的操作做重排序， 所以在 X86 处理器中，读 final 域需要的 LoadLoad 屏障也会被省略掉。也就是说，在 X86 处理器中，final 域的读/写不会插入任何内存屏障。

3.7 happens-before

3.7.1 happens-before 的定义

JSR-133 使用 happens-before 的概念来指定两个操作之间的执行顺序。由于这两个操作可以在一个线程之内，也可以是在不同线程之间，因此，JMM 可以通过 happens-before 关系向程序员提供跨线程的内存可见性保证（如果 A 线程的写操作 a 与 B 线程的读操作 b 之间存在 happens- before 关系，尽管 a 操作和 b 操作在不同的线程中执行，但 JMM 向程序员保证 a 操作将对 b 操作可见）。

（这里我理解的是，hp 规则包含了单线程或者多线程间的内存可见性保证，根据规则具体描述，包括但不仅仅描述 volatile 变量的可见性关系。只要多个线程或者单线程存在具体 hp 关系，就意味着多个线程间的操作结果是“可见”的）

《JSR-133:Java Memory Model and Thread Specification》对 happens-before 关系的定义如下（具体的 3.2.3 里有描述）：

1）如果一个操作 happens-before 另一个操作，那么第一个操作的执行结果将对第二个操作可见，而且第一个操作的执行顺序排在第二个操作之前。

2）两个操作之间存在 happens-before 关系，并不意味着 Java 平台的具体实现必须要按照 happens-before 关系指定的顺序来执行。如果重排序之后的执行结果，与按 happens-before 关系来执行的结果一致，那么这种重排序并不非法（也就是说，JMM 允许这种重排序）。

上面的 1 是 JMM 对程序员的承诺。从程序员的角度来说，可以这样理解 happens-before 关系：如果 A happens-before B，那么 Java 内存模型将向程序员保证——A 操作的结果将对 B 可见，且 A 的执行顺序排在 B 之前。注意，这只是 Java 内存模型向程序员做出的保证。

上面的 2 是 JMM 对编译器和处理器重排序的约束原则。正如前面所言，JMM 其实是在遵循一个基本原则：只要不改变程序的执行结果（指的是单线程程序和正确同步的多线程程序），编译器和处理器怎么优化都行。JMM 这么做的原因是：程序员对于这两个操作是否真的被重排序并不关心，程序员关心的是程序执行时的语义不能被改变（即执行结果不能被改变）。因此，happens-before 关系本质上和 as-if-serial 语义是一回事

as-if-serial 语义保证单线程内程序的执行结果不被改变，happens-before 关系保证正确同步的多线程程序执行结果不被改变。

as-if-serial 语义制造一个幻境：单线程的程序是按程序的顺序来执行的；
happens-before 关系制造一个幻境：正确同步的多线程程序是按照 hp 指定顺序来执行的。

3.7.2 happens-before 规则

1. 程序顺序规则：一个线程中的每个操作，happens-before 于该线程中的任意后续操作。

2. 监视器锁规则：对一个锁的解锁，happens-before 于随后对这个锁的加锁。

3. volatile 变量规则：对一个 volatile 域的写，happens-before 于任意后续对这个 volatile 域的读。

4. 传递性：如果 A happens-before B，且 B happens-before C，那么 A happens-before C。

5. start()规则：如果线程 A 执行操作 ThreadB.start()（启动线程 B），那么 A 线程的 ThreadB.start()操作 happens-before 于线程 B 中的任意操作

6. join()规则：如果线程 A 执行操作 ThreadB.join()并成功返回，那么线程 B 中的任意操作 happens-before 于线程 A 从 ThreadB.join()操作成功返回

3.8 双重检查锁定与延迟初始化

3.8.1 双重检查的问题

```
public class DoubleCheckedLocking{
    private static Instance instance;
    public static Instance getInstance(){
        if(instance == null){
            synchronized(DoubleCheckedLocking.class){
                if(instance == null){
                    instance = new Instance();
                }
            }
        }
        return instance;
    }
}
```

上面代码似乎没有问题，但是在第一次进入方法去做检查 instance 时，可能 instance 引用的对象还没有初始化。

在双重检查中，创建单例对象有三个操作：
1）分配对象内存空间（分配好后就初始化为 0 值）
2）初始化对象（引用构造方法）
3）对象的引用指向内存地址
4）访问对象

问题就出在这个创建的 2,3 步骤重排序上。根据 Java 语言规范，要保证重排序不会改变单线程内的程序执行结果。在单线程中，虽然 2,3 步骤重排序了，但是结果并没有改变。

但是在多线程中，当 A 线程的 2,3 重排序后，B 线程可能看到的是一个还没初始化的对象。如果出现了这种情况，线程 B 在第一次检查 instance 时会发现该对象不为 null（因为已经指向了对象的引用地址空间，尽管该类还没有初始化，构造函数之类的都没有完成），然后由于 B 线程发现对象不为 null，于是访问的对象可能就是还没被初始化的对象。

解决：
1）不允许 2,3 重排序
2）允许重排，但是不允许其他线程看到这个重排。

3.8.2 问题的解决方案

将 instance 变量设置为 volatile 型。当设置为 volatile 类型后，上一节中的 2,3 步骤就不会重排序。这个方法是禁止重排序来保证线程安全的延迟加载。

另外一种方法。

JVM 在类的初始化阶段阶段（即在 Class 被加载后，且被线程使用之前），会执行类的初始化（即执行类构造方法<clinit>()的过程）。在执行类的初始化期间，JVM 会去获取一个锁。这个锁可以同步多个线程对同一个类的初始化。

```
public class InstanceFactory{
    private static class InstanceHolder{
        public static Instance instance = new Instance();
    }
    public static Instance getInstance(){
        return InstanceHolder.instance;
    }
}
```

假设两个线程 A,B 去执行 getInstance()方法。

假设 A 先执行，根据《Java 虚拟机规范》，在调用类的静态方法时，会立即初始化。由于此时 A 获取到一个 Class 锁，B 只有等待 A 释放 Class 锁。然后 A 释放锁，B 拿到 Class 锁进入。由于 A 已经初始化过对象（同一个类加载器下，一个类型只会被初始化一次，A 退出后，唤醒线程 B，B 不会再进入<clinit>()方法，详见《深入理解 Java 虚拟机》），B 拿到的就是已经初始化后的对象。

对比两种方法：类初始化的方法显得更加简洁；但是 volatile 有个额外优势，除了能对静态字段实现延迟初始化外，也能对实例字段实现延迟初始化。

3.9 JMM 的内存可见性保证

按程序类型，Java 程序的内存可见性保证可以分为下列 3 类：

1）单线程程序。单线程程序不会出现内存可见性问题。

2）正确同步的多线程程序。正确同步的多线程程序的执行将具有顺序一致性。JMM 通过限制编译器和处理器的重排序来为程序员提供内存可见性保证。

3）未同步/未正确同步的多线程程序。JMM 为它们提供了最小安全性保障：线程执行时读取到的值，要么是之前某个线程写入的值，要么是默认值。

四、Java 并发编程基础

4.1 线程简介

4.1.1 什么是线程

现代操作系统在运行一个程序时，会为其创建一个进程。例如，启动一个 Java 程序，操作系统就会创建一个 Java 进程。现代操作系统调度的最小单元是线程，也叫轻量级进程（Light Weight Process），（进程是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位）在一个进程里可以创建多个线程，这些线程都拥有各自的计数器、堆栈和局部变量等属性，并且能够访问共享的内存变量。处理器在这些线程上高速切换，让使用者感觉到这些线程在同时执行。

一个 Java 程序从 main()方法开始执行，然后按照既定的代码逻辑执行，看似没有其他线程参与，但实际上 Java 程序天生就是多线程程序，因为执行 main()方法的是一个名称为 main 的线程。

4.1.2 为什么要用多线程

使用多线程的原因：

1）更多的处理器核心：

随着处理器上的核心数量越来越多，以及超线程技术的广泛运用，现在大多数计算机都比以往更加擅长并行计算，而处理器性能的提升方式，也从更高的主频向更多的核心发展。如何利用好处理器上的多个核心也成了现在的主要问题。

线程是大多数操作系统调度的基本单元，一个程序作为一个进程来运行，程序运行过程中能够创建多个线程，而一个线程在一个时刻只能运行在一个处理器核心上。试想一下，一个单线程程序在运行时只能使用一个处理器核心，那么再多的处理器核心加入也无法显著提升该程序的执行效率。相反，如果该程序使用多线程技术，将计算逻辑分配到多个处理器核心上，就会显著减少程序的处理时间，并且随着更多处理器核心的加入而变得更有效率。

2）更快的响应时间

有时我们会编写一些较为复杂的代码（这里的复杂不是说复杂的算法，而是复杂的业务逻辑），例如，一笔订单的创建，它包括插入订单数据、生成订单快照、发送邮件通知卖家和记录货品销售数量等。用户从单击“订购”按钮开始，就要等待这些操作全部完成才能看到订购成功的结果。但是这么多业务操作，如何能够让其更快地完成呢？

在上面的场景中，可以使用多线程技术，即将数据一致性不强的操作派发给其他线程处理（也可以使用消息队列），如生成订单快照、发送邮件等。这样做的好处是响应用户请求的线程能够尽可能快地处理完成，缩短了响应时间，提升了用户体验。

3）更好的编程模型

Java 为多线程编程提供了良好、考究并且一致的编程模型，使开发人员能够更加专注于问题的解决，即为所遇到的问题建立合适的模型，而不是绞尽脑汁地考虑如何将其多线程化。一旦开发人员建立好了模型，稍做修改总是能够方便地映射到 Java 提供的多线程编程模型上

4.1.3 线程优先级

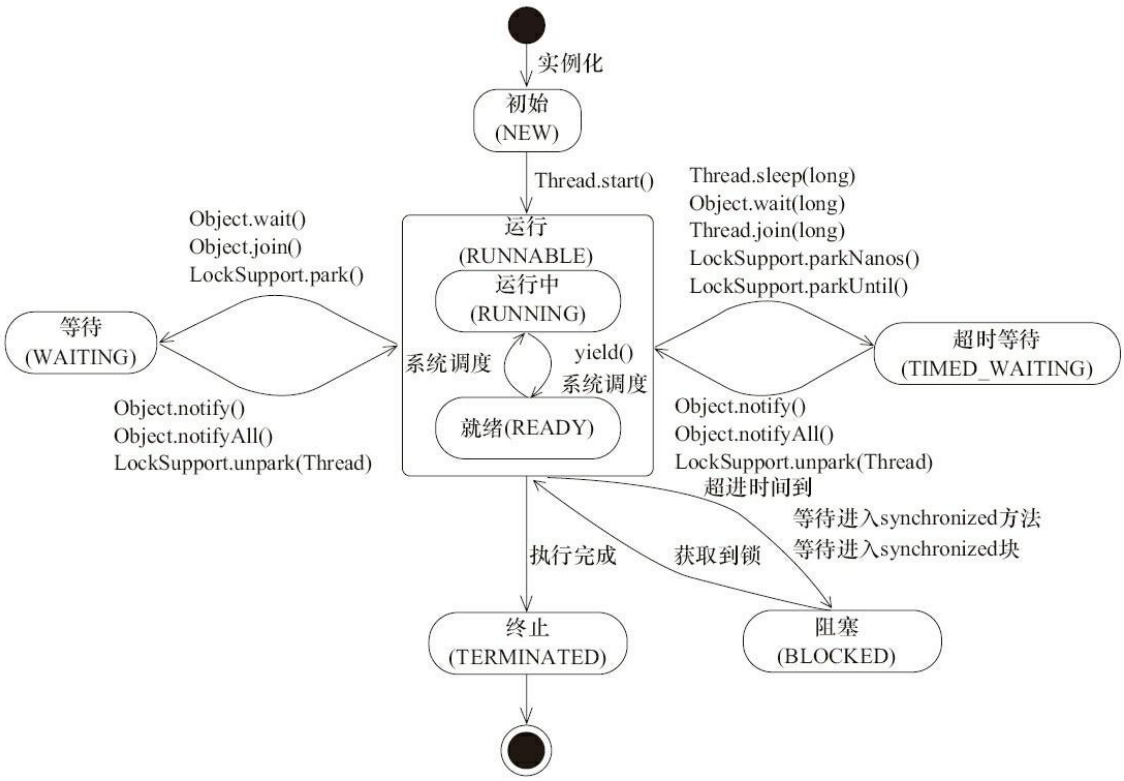
现代操作系统基本采用时分的形式调度运行的线程，操作系统会分出一个个时间片，线程会分配到若干时间片，当线程的时间片用完了就会发生线程调度，并等待着下次分配。线程分配到的时间片多少也就决定了线程使用处理器资源的多少，而线程优先级就是决定线程需要多或者少分配一些处理器资源的线程属性。

在 Java 线程中，通过一个整型成员变量 priority 来控制优先级，优先级的范围从 1~10，在线程构建的时候可以通过 setPriority(int)方法来修改优先级，默认优先级是 5，优先级高的线程分配时间片的数量要多于优先级低的线程。设置线程优先级时，针对频繁阻塞（休眠或者 I/O 操作）的线程需要设置较高优先级，而偏重计算（需要较多 CPU 时间或者偏运算）的线程则设置较低的优先级，确保处理器不会被独占。

4.1.4 线程的状态

Java 线程在运行的生命周期中可能处于表中所示的 6 种不同的状态，在给定的一个时刻，线程只能处于其中的一个状态。

状态名称	说　明
NEW	初始状态，线程被构建，但是还没有调用 start() 方法
RUNNABLE	运行状态，Java 线程将操作系统中的就绪和运行两种状态笼统地称作“运行中”
BLOCKED	阻塞状态，表示线程阻塞于锁
WAITING	等待状态，表示线程进入等待状态，进入该状态表示当前线程需要等待其他线程做出一些特定动作（通知或中断）
TIME_WAITING	超时等待状态，该状态不同于 WAITING，它是可以在指定的时间自行返回的
TERMINATED	终止状态，表示当前线程已经执行完毕



Java 将操作系统中的运行和就绪两个状态合并称为运行状态。阻塞状态是线程阻塞在进入 synchronized 关键字修饰的方法或代码块（获取锁）时的状态，但是阻塞在 java.concurrent 包中 Lock 接口的线程状态却是等待状态，因为 java.concurrent 包中 Lock 接口对于阻塞的实现均使用了 LockSupport 类中的相关方法。

4.1.5 Daemon 线程

Daemon 线程是一种支持型线程，因为它主要被用作程序中后台调度以及支持性工作。这意味着，当一个 Java 虚拟机中不存在非 Daemon 线程的时候，Java 虚拟机将会退出。可以通过调用 Thread.setDaemon(true)将线程设置为 Daemon 线程。

Daemon 线程被用作完成支持性工作，但是在 Java 虚拟机退出时 Daemon 线程中的 finally 块并不一定会执行

4.2 启动与终止线程

中断可以理解为线程的一个标识位属性，它表示一个运行中的线程是否被其他线程进行了中断操作。中断好比其他线程对该线程打了个招呼，其他线程通过调用该线程的 interrupt() 方法对其进行中断操作。

suspend()、resume()、stop()方法已经过期了。以 suspend()方法为例，在调用后，线程不会释放已经占有的资源（比如锁），而是占有着资源进入睡眠状态，这样容易引发死锁问题。同样，stop()方法在终结一个线程时不会保证线程的资源正常释放，通常是没有给予线程完成资源释放工作的机会， 因此会导致程序可能工作在不确定状态下。

中断状态是线程的一个标识位，而中断操作是一种简便的线程间交互方式，而这种交互方式最适合用来取消或停止任务。除了中断以外，还可以利用一个 boolean 变量来控制是否需要停止任务并终止该线程。

4.3 线程间通信

4.3.1 volatile 和 synchronized 关键字

关键字 volatile 可以用来修饰字段（成员变量），就是告知程序任何对该变量的访问均需要从共享内存中获取，而对它的改变必须同步刷新回共享内存，它能保证所有线程对变量访问的可见性。

关键字 synchronized 可以修饰方法或者以同步块的形式来进行使用，它主要确保多个线程在同一个时刻，只能有一个线程处于方法或者同步块中，它保证了线程对变量访问的可见性和排他性。

synchronized 同步块在字节码层面使用了 monitorenter 和 monitorexit 指令，而 synchronized 方法则是依靠方法修饰符上的 ACC_SYNCHRONIZED 来完成。两种方法的本质都是对一个对象的监视器（monitor）进行获取，而这个获取是排他的，也就是同一时刻只能有一个线程获取到由 synchronized 所保护对象的监视器。

任意一个对象都拥有自己的监视器，当这个对象由同步块或者这个对象的同步方法调用时，执行方法的线程必须先获取到该对象的监视器才能进入同步块或者同步方法，而没有获取到监视器（执行该方法）的线程将会被阻塞在同步块和同步方法的入口处，进入 BLOCKED 状态。

4.3.2 等待/通知机制

一个线程修改了一个对象的值，而另一个线程感知到了变化，然后进行相应的操作，整个过程开始于一个线程，而最终执行又是另一个线程。前者是生产者，后者就是消费者，这种模式隔离了“做什么”（what）和“怎么做”（How），在功能层面上实现了解耦，体系结构上具备了良好的伸缩性。

在 Java 中通过内置的等待/通知机制能够很好地实现上面的功能。

方法名称	描　　述
notify()	通知一个在对象上等待的线程，使其从 wait() 方法返回，而返回的前提是该线程获取到了对象的锁
notifyAll()	通知所有等待在该对象上的线程
wait()	调用该方法的线程进入 WAITING 状态，只有等待另外线程的通知或被中断才会返回，需要注意，调用 wait() 方法后，会释放对象的锁
wait(long)	超时等待一段时间，这里的参数时间是毫秒，也就是等待长达 n 毫秒，如果没有通知就超时返回
wait(long, int)	对于超时时间更细粒度的控制，可以达到纳秒

等待/通知机制，是指一个线程 A 调用了对象 O 的 wait()方法进入等待状态，而另一个线程 B 调用了对象 O 的 notify()或者 notifyAll()方法，线程 A 收到通知后从对象 O 的 wait()方法返回，进而执行后续操作。

需要注意的细节：

1）使用 wait()、notify()和 notifyAll()时需要先对调用对象加锁。

2）调用 wait()方法后，线程状态由 RUNNING 变为 WAITING，并将当前线程放置到对象的等待队列

（等待队列等待被唤醒，然后进同步队列里和其他线程争夺锁，等待队列里的线程不会去争夺锁）。

3）notify()或 notifyAll()方法调用后，等待线程依旧不会从 wait()返回，需要调用 notify()或 notifAll()的线程释放锁之后，等待线程才有机会从 wait()返回

（这里说的是有机会，在释放锁之后，会重新引起线程争夺锁，如果被其他线程拿到锁，等待线程只能等到其他线程释放锁后再去争夺）。

4）notify()方法将等待队列中的一个等待线程从等待队列中移到同步队列中，而 notifyAll() 方法则是将等待队列中所有的线程全部移到同步队列，被移动的线程状态由 WAITING 变为 BLOCKED。

5）从 wait()方法返回的前提是获得了调用对象的锁（等待线程只有获取了锁才能继续从 wait()方法往后执行）。

4.3.3 管道输入/输出流

管道输入/输出流和普通的文件输入/输出流或者网络输入/输出流不同之处在于，它主要用于线程之间的数据传输，而传输的媒介为内存。

管道输入/输出流主要包括了如下 4 种具体实现：PipedOutputStream、PipedInputStream、PipedReader 和 PipedWriter，前两种面向字节，而后两种面向字符。

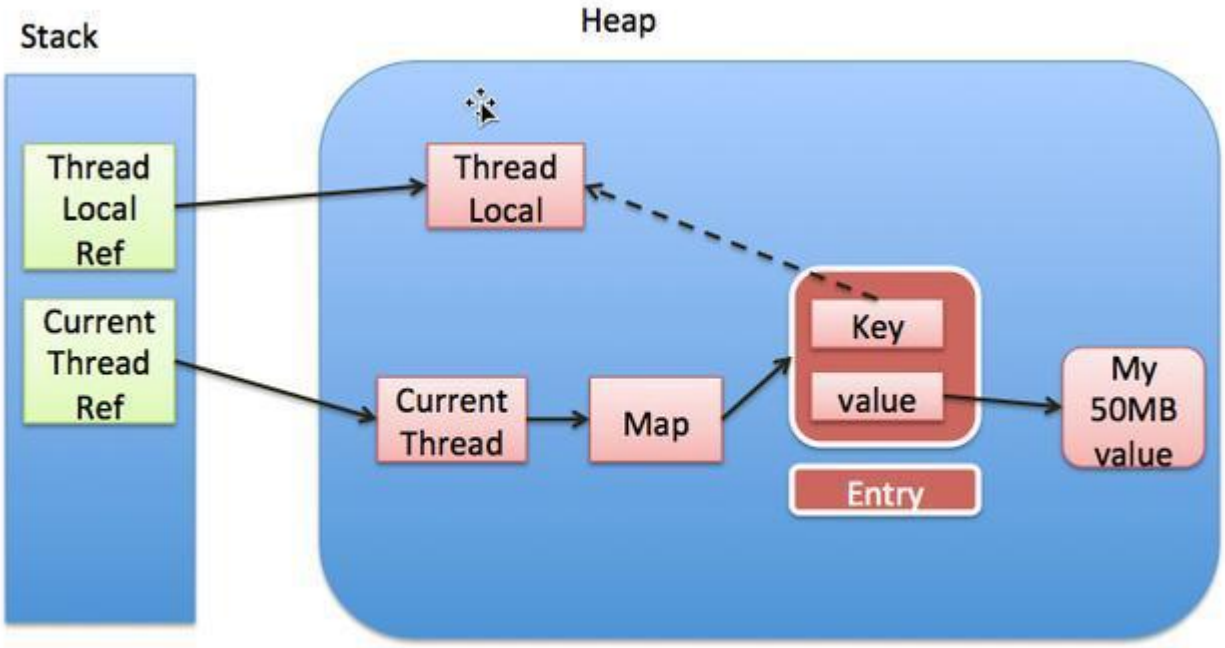
4.3.4 Thread.join()的使用

如果一个线程 A 执行了 thread.join()语句，其含义是：当前线程 A 等待 thread 线程终止之后才从 thread.join()返回。线程 Thread 除了提供 join()方法之外，还提供了 join(long millis)和 join(long millis,int nanos)两个具备超时特性的方法。这两个超时方法表示，如果线程 thread 在给定的超时时间里没有终止，那么将会从该超时方法中返回。

join 方法涉及到了等待/通知机制。

4.3.5 ThreadLocal 的使用

ThreadLocal，即线程变量，是一个以 ThreadLocal 对象为键、任意对象为值的存储结构。这个结构被附带在线程上，也就是说一个线程可以根据一个 ThreadLocal 对象查询到绑定在这个线程上的一个值。



（1）每个 Thread 维护着一个 ThreadLocalMap 的引用

（2）ThreadLocalMap 是 ThreadLocal 的内部类，用 Entry 来进行存储

（3）ThreadLocal 创建的副本是存储在自己的 threadLocals 中的，也就是自己的 ThreadLocalMap。

（4）ThreadLocalMap 的键值为 ThreadLocal 对象，而且可以有多个 threadLocal 变量，因此保存在 map 中

（5）在进行 get 之前，必须先 set，否则会报空指针异常，当然也可以初始化一个，但是必须重写 initialValue()方法。

（6）ThreadLocal 本身并不存储值，它只是作为一个 key 来让线程从 ThreadLocalMap 获取 value。

五、Java 中的锁

5.1 Lock 接口

锁是用来控制多个线程访问共享资源的方式，一般来说，一个锁能够防止多个线程同时访问共享资源（但是有些锁可以允许多个线程并发的访问共享资源，比如读写锁）。在 Lock 接口出现之前，Java 程序是靠 synchronized 关键字实现锁功能的，而 Java SE 5 之后，并发包中新增了 Lock 接口（以及相关实现类）用来实现锁功能，它提供了与 synchronized 关键字类似的同步功能，只是在使用时需要显式地获取和释放锁。虽然它缺少了（通过 synchronized 块或者方法所提供的）隐式获取释放锁的便捷性，但是却拥有了锁获取与释放的可操作性、可中断的获取锁以及超时获取锁等多种 synchronized 关键字所不具备的同步特性。

使用 synchronized 关键字将会隐式地获取锁，但是它将锁的获取和释放固化了，也就是先获取再释放。当然，这种方式简化了同步的管理，可是扩展性没有显示的锁获取和释放来的好。例如，针对一个场景，手把手进行锁获取和释放，先获得锁 A，然后再获取锁 B，当锁 B 获得后，释放锁 A 同时获取锁 C，当锁 C 获得后，再释放 B 同时获取锁 D，以此类推。这种场景下，synchronized 关键字就不那么容易实现了，而使用 Lock 却容易许多。

```
ReentrantLock lock = new ReentrantLock();
lock.lock();
try{
}finally{
   lock.unlock();
}
```

在 finally 块中释放锁，目的是保证在获取到锁之后，最终能够被释放。

不要将获取锁的过程写在 try 块中（写在里面编译器会报错），因为如果在获取锁（自定义锁的实现）时发生了异常， 异常抛出的同时，也会导致锁无故释放。

Lock 接口提供的 synchronized 不具备的特性：

特　　性	描　　述
尝试非阻塞地获取锁	当前线程尝试获取锁，如果这一时刻锁没有被其他线程获取到，则成功获取并持有锁
能被中断地获取锁	与 synchronized 不同，获取到锁的线程能够响应中断，当获取到锁的线程被中断时，中断异常将会被抛出，同时锁会被释放
超时获取锁	在指定的截止时间之前获取锁，如果截止时间到了仍旧无法获取锁，则返回

Lock 接口提供的 API:

方法名称	描　　述
void lock()	获取锁，调用该方法当前线程将会获取锁，当锁获得后，从该方法返回
void lockInterruptibly() throws InterruptedException	可中断地获取锁，和 lock() 方法的不同之处在于该方法会响应中断，即在锁的获取中可以中断当前线程
boolean tryLock()	尝试非阻塞的获取锁，调用该方法后立刻返回，如果能够获取则返回 true，否则返回 false
boolean tryLock(long time, TimeUnit unit) throws InterruptedException	超时的获取锁，当前线程在以下 3 种情况下会返回： ①当前线程在超时时间内获得了锁 ②当前线程在超时时间内被中断 ③超时时间结束，返回 false
void unlock()	释放锁
Condition newCondition()	获取等待通知组件，该组件和当前的锁绑定，当前线程只有获得了锁，才能调用该组件的 wait() 方法，而调用后，当前线程将释放锁

Lock 接口的实现基本都是通过聚合了一个同步器的子类来完成线程访问控制。

5.2 队列同步器 AQS

5.2.1 队列同步器的接口

队列同步器 AbstractQueuedSynchronizer（以下简称同步器），是用来构建锁或者其他同步组件的基础框架，它使用了一个 int 成员变量表示同步状态，通过内置的 FIFO 队列来完成资源获取线程的排队工作。

同步器的主要使用方式是继承，子类通过继承同步器并实现它的抽象方法来管理同步状态，在抽象方法的实现过程中免不了要对同步状态进行更改，这时就需要使用同步器提供的 3 个方法（getState()、setState(int newState)和compareAndSetState(int expect,int update)）来进行操作，因为它们能够保证状态的改变是安全的。

子类推荐被定义为自定义同步组件的静态内部类，同步器自身没有实现任何同步接口，它仅仅是定义了若干同步状态获取和释放的方法来供自定义同步组件使用，同步器既可以支持独占式地获取同步状态，也可以支持共享式地获取同步状态，这样就可以方便实现不同类型的同步组件（ReentrantLock、ReentrantReadWriteLock 和 CountDownLatch 等）。

同步器的设计是基于模板方法模式的，也就是说，使用者需要继承同步器并重写指定的方法，随后将同步器组合在自定义同步组件的实现中，并调用同步器提供的模板方法，而这些模板方法将会调用使用者重写的方法。

重写同步器指定方法，需要使用同步器提供的下面的三个方法：

1）getState()：获取同步器状态

2）setState(int newState)：设置当前同步状态

3）compareAndSetState(int expect,int update)：使用 CAS 设置当前状态，该方法能够保证状态设置的原子性

同步器可重写的方法：

方法名称	描　　述
protected boolean tryAcquire(int arg)	独占式获取同步状态，实现该方法需要查询当前状态并判断同步状态是否符合预期，然后再进行 CAS 设置同步状态
protected boolean tryRelease(int arg)	独占式释放同步状态，等待获取同步状态的线程将有机会获取同步状态

（续）

方法名称	描　　述
protected int tryAcquireShared(int arg)	共享式获取同步状态，返回大于等于 0 的值，表示获取成功，反之，获取失败
protected boolean tryReleaseShared(int arg)	共享式释放同步状态
protected boolean isHeldExclusively()	当前同步器是否在独占模式下被线程占用，一般该方法表示是否被当前线程所独占

同步器提供的模板方法：

方法名称	描　　述
void acquire(int arg)	独占式获取同步状态，如果当前线程获取同步状态成功，则由该方法返回，否则，将会进入同步队列等待，该方法将会调用重写的 tryAcquire(int arg) 方法
void acquireInterruptibly(int arg)	与 acquire(int arg) 相同，但是该方法响应中断，当前线程未获取到同步状态而进入同步队列中，如果当前线程被中断，则该方法会抛出 InterruptedException 并返回
boolean tryAcquireNanos(int arg, long nanos)	在 acquireInterruptibly(int arg) 基础上增加了超时限制，如果当前线程在超时时间内没有获取到同步状态，那么将会返回 false，如果获取到了返回 true
void acquireShared(int arg)	共享式的获取同步状态，如果当前线程未获取到同步状态，将会进入同步队列等待，与独占式获取的主要区别是在同一时刻可以有多个线程获取到同步状态
void acquireSharedInterruptibly(int arg)	与 acquireShared(int arg) 相同，该方法响应中断
boolean tryAcquireSharedNanos(int arg, long nanos)	在 acquireSharedInterruptibly(int arg) 基础上增加了超时限制
boolean release(int arg)	独占式的释放同步状态，该方法会在释放同步状态之后，将同步队列中第一个节点包含的线程唤醒
boolean releaseShared(int arg)	共享式的释放同步状态
Collection<Thread> getQueuedThreads()	获取等待在同步队列上的线程集合

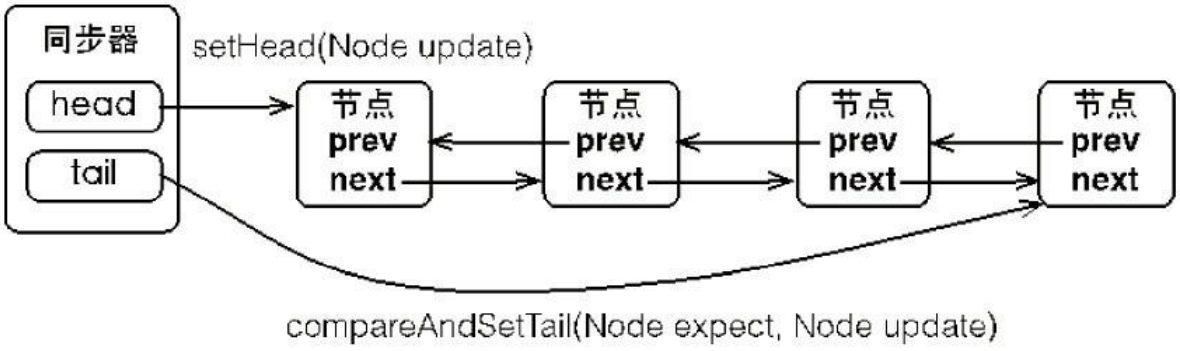
5.2.2 队列同步器的实现分析

同步器依赖内部的同步队列（一个 FIFO 双向队列）来完成同步状态的管理，当前线程获取同步状态失败时，同步器会将当前线程以及等待状态等信息构造成为一个节点（Node）并将其加入同步队列，同时会阻塞当前线程，当同步状态释放时，会把首节点中的线程唤醒，使其再次尝试获取同步状态。

同步队列中的节点（Node）用来保存获取同步状态失败的线程引用、等待状态以及前驱和后继节点：

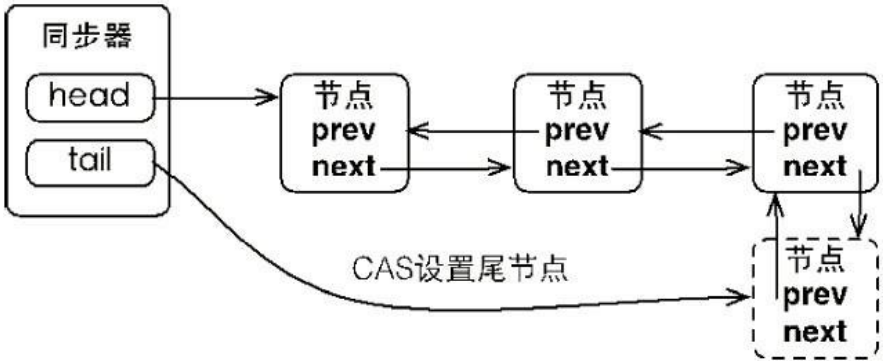
属性类型与名称	描　　述
int waitStatus	等待状态。 包含如下状态。 ① CANCELLED，值为 1，由于在同步队列中等待的线程等待超时或者被中断，需要从同步队列中取消等待，节点进入该状态将不会变化 ② SIGNAL，值为 -1，后继节点的线程处于等待状态，而当前节点的线程如果释放了同步状态或者被取消，将会通知后继节点，使后继节点的线程得以运行 ③ CONDITION，值为 -2，节点在等待队列中，节点线程等待在 Condition 上，当其他线程对 Condition 调用了 signal() 方法后，该节点将会从等待队列中转移到同步队列中，加入到对同步状态的获取中 ④ PROPAGATE，值为 -3，表示下一次共享式同步状态获取将会无条件地被传播下去 ⑤ INITIAL，值为 0，初始状态
Node prev	前驱节点，当节点加入同步队列时被设置（尾部添加）
Node next	后继节点
Node nextWaiter	等待队列中的后继节点。如果当前节点是共享的，那么这个字段将是一个 SHARED 常量，也就是说节点类型（独占和共享）和等待队列中的后继节点共用同一个字段
Thread thread	获取同步状态的线程

节点是构成同步队列的基础，同步器拥有首节点（head） 和尾节点（tail），没有成功获取同步状态的线程将会成为节点加入该队列的尾部，同步队列的基本结构如图：

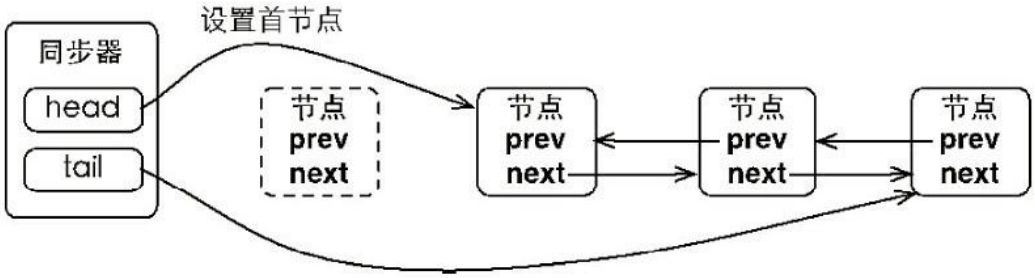


同步器提供了一个基于 CAS 的设置尾节点的方法：compareAndSetTail(Node expect,Node update)，它需要传递当前线程“认为”的尾节点和当前节点，只有设置成功后，当前节点才正式与之前的尾节点建立关联。

同步队列遵循 FIFO，首节点是获取同步状态成功的节点，首节点的线程在释放同步状态时，将会唤醒后继节点，而后继节点将会在获取同步状态成功时将自



已设置为首节点。设置首节点是通过获取同步状态成功的线程来完成的，由于只有一个线程能够成功获取到同步状态，因此设置头节点的方法并不需要使用 CAS 来保证，它只需要将首节点设置成为原首节点的后继节点并断开原首节点的 next 引用即可：



5.3 重入锁

重入锁 ReentrantLock，顾名思义，就是支持重进入的锁，它表示该锁能够支持一个线程对资源的重复加锁。除此之外，该锁的还支持获取锁时的公平和非公平性选择。

synchronized 关键字隐式的支持重进入，比如一个 synchronized 修饰的递归方法，在方法执行时，执行线程在获取了锁之后仍能连续多次地获得该锁，而不像 Mutex 由于获取了锁，而在下一次获取锁时出现阻塞自己的情况。

ReentrantLock 虽然没能像 synchronized 关键字一样支持隐式的重进入，但是在调用 lock()方法时，已经获取到锁的线程，能够再次调用 lock()方法获取锁而不被阻塞。

重进入是指任意线程在获取到锁之后能够再次获取该锁而不会被锁所阻塞，该特性的实现需要解决以下两个问题：

1）线程再次获取锁。锁需要去识别获取锁的线程是否为当前占据锁的线程，如果是，则再次成功获取。

2）锁的最终释放。线程重复了 n 次获取了锁，其他线程能够获取到该锁。锁的最终释放要求锁对于获取进行计数自增，计数表示当前锁被重复获取的次数，而锁被释放时，计数自减，当计数等于 0 时表示该锁已经成功释放。

如果一个锁是公平的，那么锁的获取顺序就应该符合请求的绝对时间顺序，也就是 FIFO。

对于非公平锁，只要 CAS 设置同步状态成功，则表示当前线程获取了锁。

对于公平锁，还要判断加入了同步队列中当前节点是否有前驱节点。

公平性锁保证了锁的获取按照 FIFO 原则，而代价是进行大量的线程切换。非公平性锁虽然可能造成线程“饥饿”，但极少的线程切换，保证了其更大的吞吐量。

5.4 读写锁

读写锁维护了一对锁，一个读锁和一个写锁，通过分离读锁和写锁，使得并发性相比一般的排他锁有了很大提升。

除了保证写操作对读操作的可见性以及并发性的提升之外，读写锁能够简化读写交互场景的编程方式。假设在程序中定义一个共享的用作缓存数据结构，它大部分时间提供读服务（例如查询和搜索），而写操作占有的时间很少，但是写操作完成之后的更新需要对后续的读服务可见。

一般情况下，读写锁的性能都会比排它锁好，因为大多数场景读是多于写的。在读多于写的情况下，读写锁能够提供比排它锁更好的并发性和吞吐量。Java 并发包提供读写锁的实现是ReentrantReadWriteLock。

特　　性	说　　明
公平性选择	支持非公平（默认）和公平的锁获取方式，吞吐量还是非公平优于公平
重进入	该锁支持重进入，以读写线程为例：读线程在获取了读锁之后，能够再次获取读锁。而写线程在获取了写锁之后能够再次获取写锁，同时也可以获取读锁
锁降级	遵循获取写锁、获取读锁再释放写锁的次序，写锁能够降级成为读锁

如果存在读锁，则写锁不能被获取，原因在于：读写锁要确保写锁的操作对读锁可见，如果允许读锁在已被获取的情况下对写锁的获取，那么正在运行的其他读线程就无法感知到当前写线程的操作。因此，只有等待其他读线程都释放了读锁，写锁才能被当前线程获取，而写锁一旦被获取，则其他读写线程的后续访问均被阻塞。

写锁的释放与 ReentrantLock 的释放过程基本类似，每次释放均减少写状态，当写状态为 0 时表示写锁已被释放，从而等待的读写线程能够继续访问读写锁，同时前次写线程的修改对后续读写线程可见。

读锁是一个支持重进入的共享锁，它能够被多个线程同时获取，在没有其他写线程访问（或者写状态为 0）时，读锁总会被成功地获取，而所做的也只是（线程安全的）增加读状态。如果当前线程已经获取了读锁，则增加读状态。如果当前线程在获取读锁时，写锁已被其他线程获取，则进入等待状态。

锁降级指的是写锁降级成为读锁。如果当前线程拥有写锁，然后将其释放，最后再获取读锁，这种分段完成的过程不能称之为锁降级。锁降级是指把持住（当前拥有的）写锁，再获取到读锁，随后释放（先前拥有的）写锁的过程。

锁降级中读锁的获取是必要的。主要是为了保证数据的可见性，如果当前线程不获取读锁而是直接释放写锁，假设此刻另一个线程（记作线程 T）获取了写锁并修改了数据，那么当前线程无法感知线程 T 的数据更新。如果当前线程获取读锁，即遵循锁降级的步骤，则线程 T 将会被阻塞，直到当前线程使用数据并释放读锁之后，线程 T 才能获取写锁进行数据更新

5.5 LockSupport 工具

当需要阻塞或唤醒一个线程的时候，都会使用 LockSupport 工具类来完成相应工作。LockSupport 定义了一组的公共静态方法，这些方法提供了最基本的线程阻塞和唤醒功能，而 LockSupport 也成为构建同步组件的基础工具。

LockSupport 定义了一组以 park 开头的方法用来阻塞当前线程，以及 unpark(Thread thread) 方法来唤醒一个被阻塞的线程。

5.6 Condition 接口

Condition 接口也提供了类似 Object 的监视器方法，与 Lock 配合可以实现等待/通知模式。

Condition 定义了等待/通知两种类型的方法，当前线程调用这些方法时，需要提前获取到 Condition 对象关联的锁。Condition 对象是由 Lock 对象（调用 Lock 对象的 newCondition()方法）创建出来的，换句话说，Condition 是依赖 Lock 对象的。

方法名称	描　　述
void await() throws InterruptedException	当前线程进入等待状态直到被通知（signal）或中断，当前线程将进入运行状态且从 await() 方法返回的情况，包括： 其他线程调用该 Condition 的 signal() 或 signalAll() 方法，而当前线程被选中唤醒 ❑ 其他线程（调用 interrupt() 方法）中断当前线程 ❑ 如果当前等待线程从 await() 方法返回，那么表明该线程已经获取了 Condition 对象所对应的锁

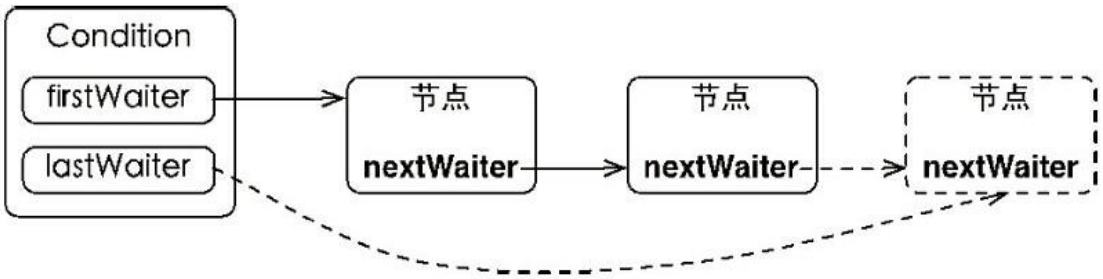
（续）

方法名称	描　　述
void awaitUninterruptibly()	当前线程进入等待状态直到被通知，从方法名称上可以看出该方法对中断不敏感
long awaitNanos(long nanosTimeout) throws InterruptedException	当前线程进入等待状态直到被通知、中断或者超时。返回值表示剩余的时间，如果在 nanosTimeout 纳秒之前被唤醒，那么返回值就是 (nanosTimeout– 实际耗时)。如果返回值是 0 或者负数，那么可以认定已经超时了
boolean awaitUntil(Date deadline) throws InterruptedException	当前线程进入等待状态直到被通知、中断或者到某个时间。如果没有到指定时间就被通知，方法返回 true，否则，表示到了指定时间，方法返回 false
void signal()	唤醒一个等待在 Condition 上的线程，该线程从等待方法返回前必须获得与 Condition 相关联的锁
void signalAll()	唤醒所有等待在 Condition 上的线程，能够从等待方法返回的线程必须获得与 Condition 相关联的锁

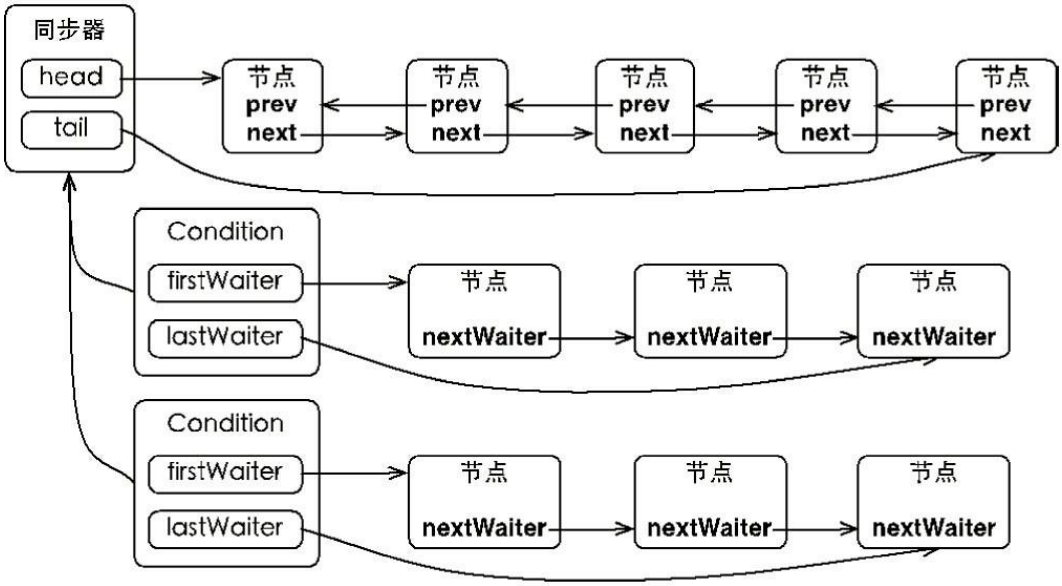
5.7 等待队列

等待队列是一个 FIFO 的队列，在队列中的每个节点都包含了一个线程引用，该线程就是在 Condition 对象上等待的线程，如果一个线程调用了 Condition.await()方法，那么该线程将会释放锁、构造成节点加入等待队列并进入等待状态。

一个 Condition 包含一个等待队列，Condition 拥有首节点（firstWaiter）和尾节点（lastWaiter）。当前线程调用 Condition.await()方法，将会以当前线程构造节点，并将节点从尾部加入等待队列，同时释放锁：



上述节点引用更新的过程并没有使用 CAS 保证，原因在于调用 await()方法的线程必定是获取了锁的线程，也就是说该过程是由锁来保证线程安全的。



如图就是同步队列和等待队列的关系。

六、Java 并发容器和框架

6.1 ConcurrentHashMap

6.1.1 为什么要用 ConcurrentHashMap

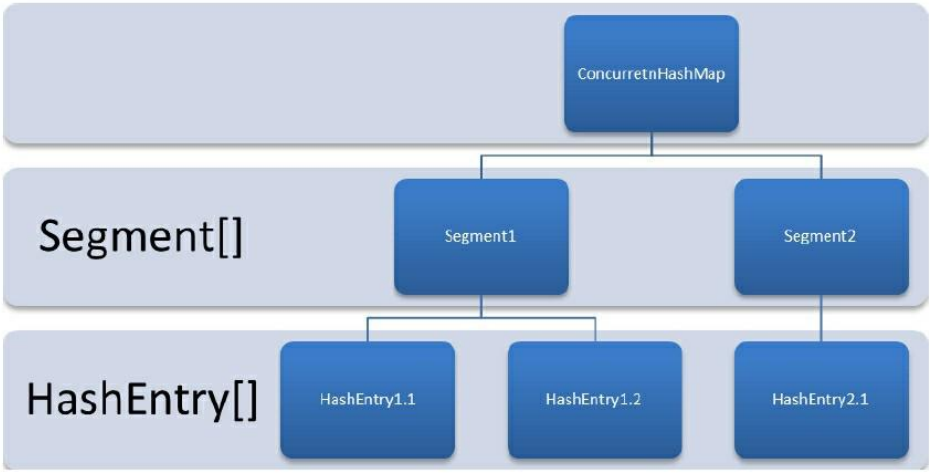
首先是 HashMap。HashMap 在多线程环境下，使用 put 操作会引起死循环。这是因为多线程导致 HashMap 的 Entry 链表形成环形数据结构，一旦形成环形数据结构，Entry 的 next 节点永远不为 null，就会产生死循环获取 Entry。（具体的会专门总结 HashMap）

HashTable 虽然是线程安全的，但是在多线程环境下，所有访问 HashTable 的线程都必须竞争同一把锁，当一个线程竞争到后，其他线程会进入阻塞或轮询状态（轻量级锁和重量级锁），总的过程效率低下。

假如容器里有多把锁，每个锁用于锁容器其中一部分数据，那么当多线程访问容器里不同的数据段时，线程间就不会存在锁竞争，从而有效提高并发访问效率。ConcurrentHashMap 所使用的就是这样的锁分段技术（JDK1.8 后已经变成对每个节点 Node 加锁）。首先将数据分成一段一段地存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。

6.1.2 ConcurrentHashMap 的结构

（书上介绍的结构都是 JDK7 中的，现在总结的是 JDK8 的结构，后面的内容将 JDK7 和 8 联系起来做对比的总结，参考 https://baijiahao.baidu.com/s?id=1617089947709260129&wfr=spider&for=pc）



JDK1.8 的实现已经摒弃了 Segment（分段）的概念（这个结构有个缺点，hash 的时间要比普通的 HashMap 长，因为定位到一个元素的过程需要两次 hash，第一次 hash 到 Segment，第二次再 hash 到链表的头部，见上图），而是直接用 Node 数组（这个数组的 value 和 next 都用 volatile 修饰）+链表+红黑树的数据结构来实现，并发控制使用 Synchronized 和 CAS 来操作，整个看起来就像是优化过且线程安全的 HashMap，虽然在 JDK1.8 中还能看到 Segment 的数据结构，但是已经简化了属性，只是为了兼容旧版本。在整体结构上，和 HashMap 很相似了，只不过 ConcurrentHashMap 通过 CAS+Synchronized 来控制并发。

和 JDK1.7 的对比：

1. 数据结构：取消了 Segment 分段锁的数据结构，取而代之的是数组+链表+红黑树的结构。

2. 保证线程安全机制：JDK1.7 采用 segment 的分段锁机制实现线程安全，其中 segment 继承自 ReentrantLock。JDK1.8 采用 CAS+Synchronized 保证线程安全。

3. 锁的粒度：原来是对需要进行数据操作的 Segment 加锁，现调整为对每个数组元素加锁（Node），粒度更细。

4. 链表转化为红黑树:定位结点的 hash 算法简化会带来弊端,Hash 冲突加剧,因此在链表节点数量大于 8 时，会将链表转化为红黑树进行存储。

5. 查询时间复杂度：从原来的遍历链表 O(n)，变成遍历红黑树 O(logN)。

（只总结了最重要的 ConcurrentHashMap，后面的如果面经问到了我继续总结）

七、Java 中的 13 个原子操作类

7.1 原子更新基本类型类

使用原子的方式更新基本类型，Atomic 包提供了以下 3 个类。

1）AtomicBoolean：原子更新布尔类型。

2）AtomicInteger：原子更新整型。

3）AtomicLong：原子更新长整型。

AtomicInteger 中的 getAndIncrement()方法使用原子操作（CAS 操作）将当前值加 1，返回自增前的值。

7.2 原子更新数组

通过原子的方式更新数组里的某个元素，Atomic 包提供了以下 3 个类。

1）AtomicIntegerArray：原子更新整型数组里的元素。

2）AtomicLongArray：原子更新长整型数组里的元素。

3）AtomicReferenceArray：原子更新引用类型数组里的元素。

7.3 原子更新引用类型

原子更新基本类型的 AtomicInteger，只能更新一个变量，如果要原子更新多个变量，就需要使用这个原子更新引用类型提供的类。

Atomic 包提供了以下 3 个类。

1）AtomicReference：原子更新引用类型。

2）AtomicReferenceFieldUpdater：原子更新引用类型里的字段。

3）AtomicMarkableReference：原子更新带有标记位的引用类型。可以原子更新一个布尔类型的标记位和引用类型。

7.4 原子更新字段类

如果需原子地更新某个类里的某个字段时，就需要使用原子更新字段类。

Atomic 包提供了以下 3 个类进行原子字段更新。

1）AtomicIntegerFieldUpdater：原子更新整型的字段的更新器。

2）AtomicLongFieldUpdater：原子更新长整型字段的更新器。

4）AtomicStampedReference：原子更新带有版本号的引用类型。该类将整数值与引用关联起来，可用于原子的更新数据和数据的版本号，可以解决使用 CAS 进行原子更新时可能出现的 ABA 问题。

八、Java 中的并发工具类

（下面的知识遇到面经上有再总结）

8.1 等待多线程完成的 CountDownLatch

8.2 同步屏障 CyclicBarrier

8.3 并发控制线程数的 Semaphore

8.4 线程间交换数据的 Exchanger

九、Java 中的线程池

在开发过程中，合理地使用线程池能够带来 3 个好处：

1）降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。

2）提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行。

3）提高线程的可管理性。线程是稀缺资源，如果无限制地创建，不仅会消耗系统资源， 还会降低系统的稳定性，使用线程池可以进行统一分配、调优和监控。

（后面实现原理以及 Executor 如果面试有再总结）

9.1 线程池的实现原理

9.2 线程池的使用

9.3 小结

十、Executor 框架

10.1 Executor 框架

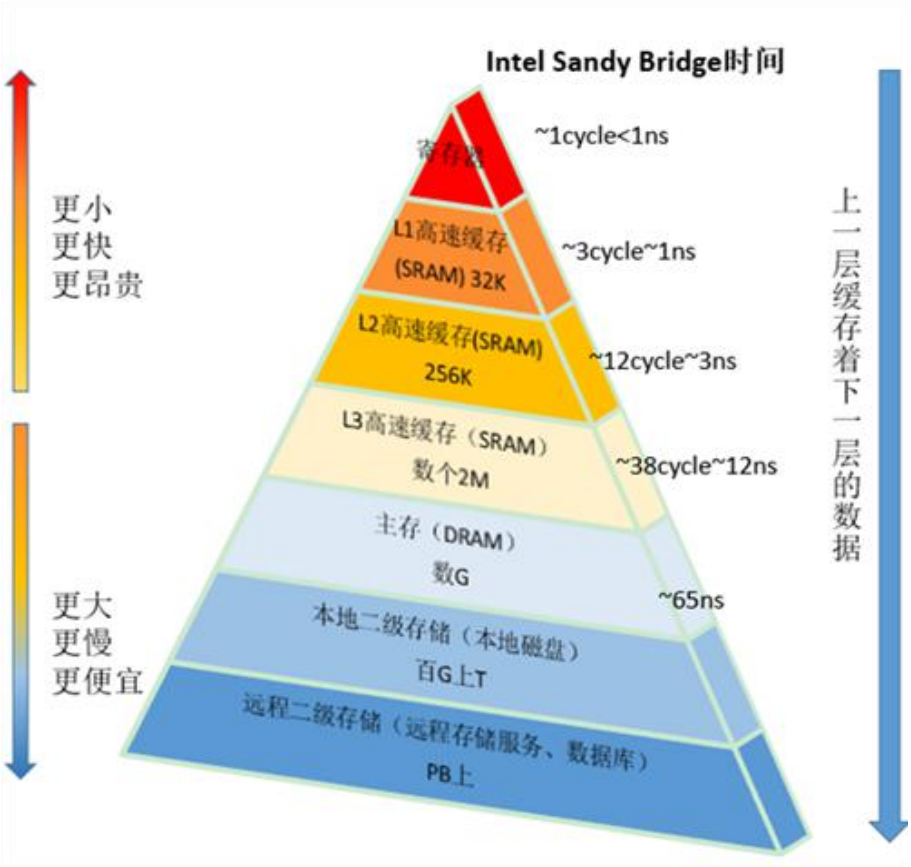
10.2 ThreadPoolExecutor 详解

10.3 ScheduledThreadExecutor 详解

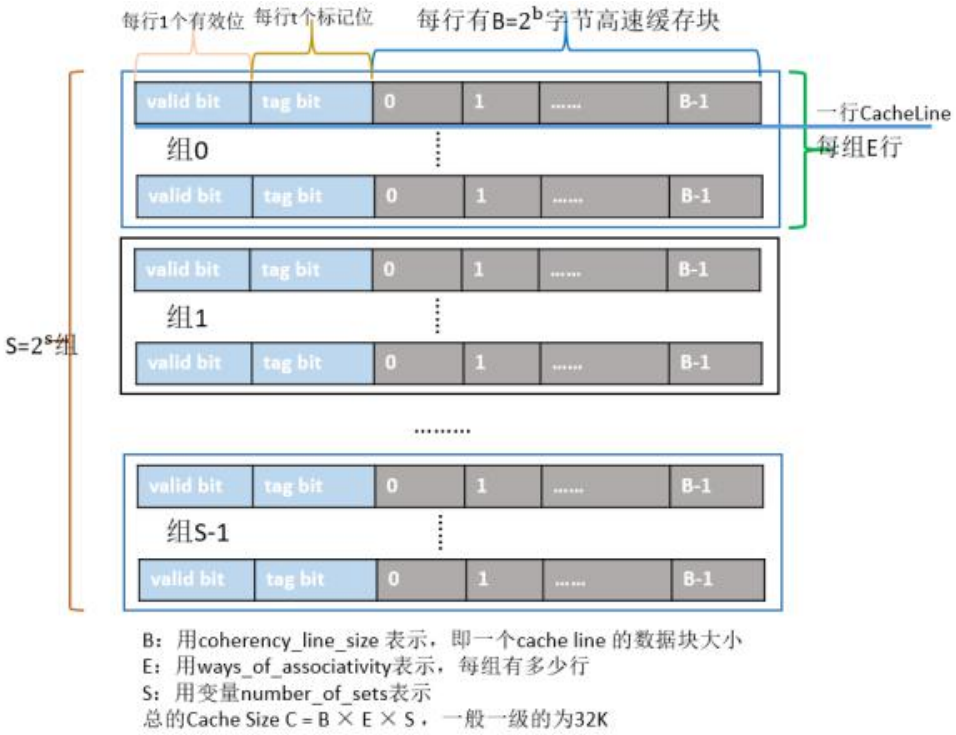
10.4 FutureTask 详解

十一、补充知识——CPU 缓存行和伪共享

随着 CPU 的频率不断提升，而内存的访问速度却没有质的突破，为了弥补访问内存的速度慢，充分发挥 CPU 的计算资源，提高 CPU 整体吞吐量，在 CPU 与内存之间引入了一级 Cache。随着热点数据体积越来越大，一级 Cache L1 已经不满足发展的要求，引入了二级 Cache L2，三级 Cache L3。注意，三级 L3 对于多核单处理器来说，是多核共享的；而 L1 和 L2 是每个核独有的。



例如 x86 芯片，Cache 的结构如图：



整个 Cache 被分为 S 个组，每个组是又由 E 行个最小的存储单元——Cache Line（后面都写缓存行）所组成。

一个缓存行中有 B（B=64，这个是常见的）个字节来存储数据，即每个缓存行能存储 64 个字节的数据，每个缓存行又额外包含一个 valid bit（有效位）、t 个 tag bit（标记位），其中有效位来表示该缓存是否有效；标记位用来协助寻址；而缓存行里的 64 个字节实际上是对应内存地址中的数据拷贝。

缓存命中（Cache hit）指的，如果进行高速缓存行填充（高速缓存位于）操作的内存位置仍然是下次处理器访问的地址时，处理器从缓存中读取操作数，而不是从内存中读取。

但是如果 CPU 访问的内存数据不在缓存中（一级、二级、三级），这就产生了 Cache Line miss（缓存行未命中）问题，此时 CPU 不得不发出新的加载指令，从内存中获取数据。这样，从内存中加载数据就会产生一个时延，为了减少这个时延的发生，就不得不提高缓存命中率，也就是充分发挥程序局部性原理。

局部性包括时间局部性，空间局部性。

时间局部性：对于同一数据可能被多次使用，自第一次加载到 Cache Line 后，后面的访问就可以多次从 Cache Line 中命中，从而提高读取速度（而不是从下层缓存读取）。

空间局部性：一个 Cache Line 有 64 字节块，我们可以充分利用一次加载 64 字节的空间，把程序后续会访问的数据，一次性全部加载进来，从而提高 Cache Line 命中率（而不是重新去寻址读取）。（其实这也可以解释为什么链表查询速

度没有数组快，因为链表的数据结构项在内存中不是相邻的，因此享受不到空间局部性带来的速度）

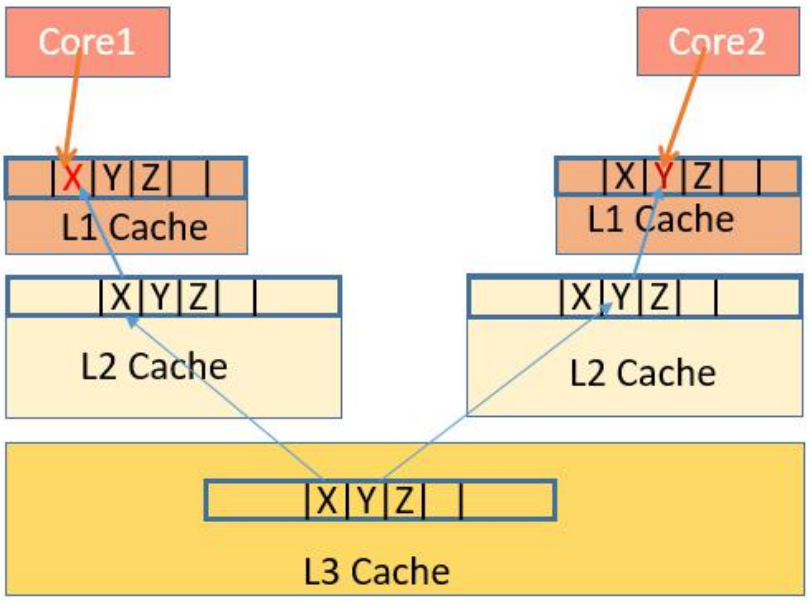
举个例子，下面代码：

```
public int run(int [] row,int[] colum){
    int sum = 0;
    for(int i = 0; i<16; i++){
        sum+=row[i]*colum[i];
    }
    return sum;
}
```

上面代码，长度为 16 的 row 和 colum 数组，在缓存行 64 字节数据块上的内存地址是连续的，能被一次加载到缓存行中，体现了空间局部性，所以在访问数组时，缓存行命中率高，性能发挥到极致。

而变量 i 则体现了时间局部性。i 作为计数器被频繁操作，一直放到寄存器中，每次都从寄存器访问，而不是从主存或者磁盘。

虽然连续紧凑的内存分配带来高性能，但并不代表它一直都能带来高性能。在多线程中，例如：



两个不同的线程，操作同一CacheLine中的不同字节，产生了不是真正共享的伪共享。

数据 X,Y,Z 被加载到同一个缓存行中。根据 MESI 法则，现在按照以下操作：

1）线程 A 和线程 B 要分别修改缓存行的 X,Y。

2）假设线程 A 先修改 X，现在 L1 中的缓存行从 S（共享）状态变为 E（修改，脏数据）状态，然后告知其他核心（Core2），其他缓存行修改为 S 状态变为 I

（无效）状态。

3）当线程 B 准备修改 Y 的时候，首先会让 Core1 将 X 写回主存（因为 Y 的缓存行已经无效了，只能重新读取），Core2 从主存重新读取该地址内容，缓存行状态由 I 变成 E（独占）。

4）当变成 E 后，线程 B 才开始修改 Y。缓存行由 E 变成 M。

注：MESI，缓存一致性协议。
本地 cache:S→E→M
触发 cache:S→E→M
其他 cache:S→I
当本地 cache 修改时，将本地 cache 修改为 E,其他 cache 修改为 I,然后再将本地 cache 为 M 状态。（其实还有很多规则，这里挑重要的）

可以看到，在高并发条件下，会频繁的将缓存行置为 I（无效）状态。虽然这些变量（上图是 X，Y，Z）没有任何关联，但是由于在一个缓存行（实际内存地址邻近），对其中变量的操作会导致频繁的缓存未命中（对其他核来说，上图是 Core2，不得不去主存拿数据），引发性能下降。

先举个例子来看看伪共享会出现的性能问题：

```
public class FalseShareTest {

    public static void main(final String[] args) throws Exception {
           VolatileLong volatileLong = new VolatileLong();
           volatileLong.value1 = 1;
           volatileLong.value2 = 1;
           long start = System.currentTimeMillis();
           Thread t1 = new Thread(() -> {
               for (int i = 0; i < 100000000; i++) {
                   volatileLong.value1++;
               }
           });
           Thread t2 = new Thread(()->{
               for (int i = 0; i < 100000000; i++) {
                   volatileLong.value2++;
               }
           });
           t1.start();
           t2.start();
           t1.join();
           t2.join();
           System.out.println(System.currentTimeMillis()-start);
```

```
    }
}
class VolatileLong {
    public volatile long value1;
    public long p1, p2, p3, p4, p5, p6,p7; // 注释此行，时间会比较
长
    public volatile long value2;
}
```

（例子来自于博客 https://blog.csdn.net/u013099854/article/details/106619796）

上面代码逻辑很简单，两个线程分别++两个不同的变量，但是在没有加上 p1-p7 这些变量时，运行会特别慢。这是因为出现了伪共享的问题，导致不停地从内存去取数据。

根据该例，解决伪共享的第一个思路：

1）padding。增大数组元素的间隔使得不同线程存取的元素位于不同的缓存行上。让修改的两个变量在不同的缓存行。

缓存行大小为 64 字节，一个 long 是 8 字节，添加 p1-p7，让它们与 value1 在一个缓存行，value1 和 value2 不在一个缓存行。这样加快了运行速度

在实际对象里，应该把一起变化的数据放在一组，与其他属性无关的属性放到一组，将不变的放入另一组。然后把每一组属性占的字节数加上前后填充的字节数凑成一个缓存行。

2）Contended 注解。在 JDK1.8 中，新增了一种注解@sun.misc.Contended，来使各个变量在 Cache line 中分隔开。如下：

```
@sun.misc.Contended
class VolatileLong {
    public volatile long value1;
    public volatile long value2;
}
```

这样就使得 value1 和 value2 在不同的缓存行。

注意：@Contended 注解会增加目标实例大小，要谨慎使用。默认情况下，除了 JDK 内部的类，JVM 会忽略该注解。要应用代码支持的话，要设置 -XX:-RestrictContended=false，它默认为 true（意味仅限 JDK 内部的类使用）。当然，也有个 –XX: EnableContented 的配置参数，来控制开启和关闭该注解的功能，默认是 true，如果改为 false，可以减少 Thread 和 ConcurrentHashMap 类的大小。

源码中，ConcurrentHashMap 中的 CounterCell 类加上了该注解。
Thread 源码里 threadLocalRandomSeed 等 long 类型变量加上了该注解。