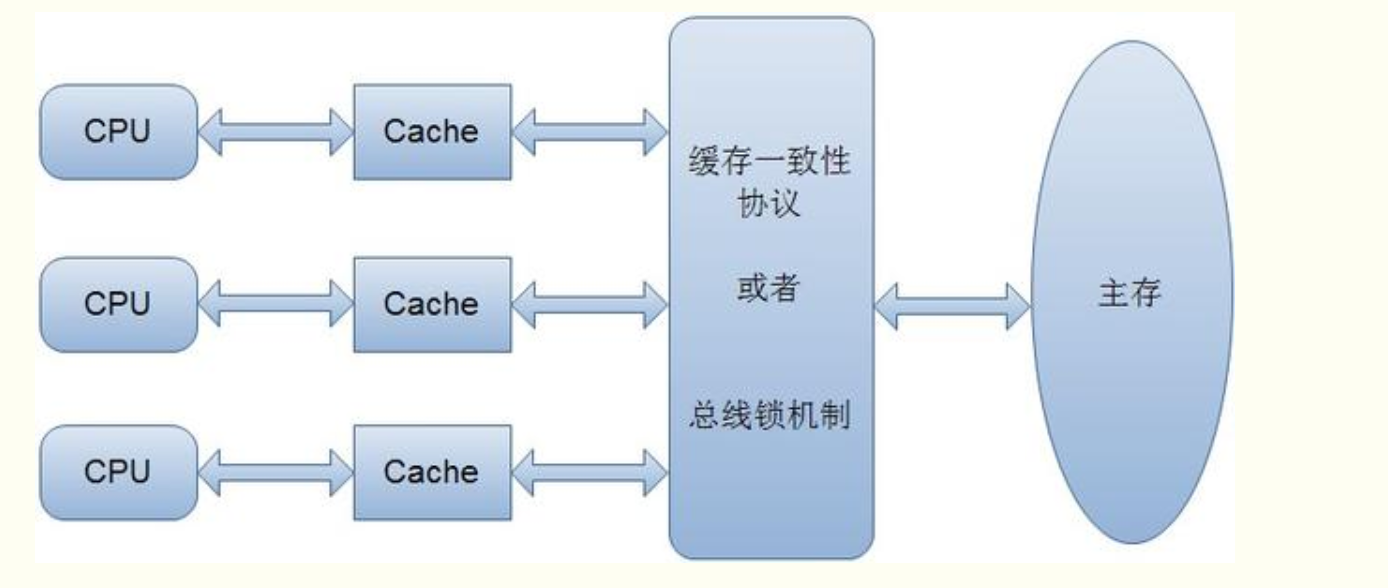
1. 为什么要使用锁？

锁-是为了解决并发操作引起的脏读、数据不一致的问题

1. 锁的实现原理？
   1. volatile

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。



结论：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**。

Volatile的内存语义是：

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立即刷新到主内存中。

当读一个volatile变量时，jmm会把该线程的对应的本地内存设置为无效，直接从主内存中读取变量的值。

当Volatile修饰的变量进行写操作时候，其对应的汇编代码中会多出一个lock操作。Lock前缀指令的作用就是：

1. 将当前处理器缓存行的数据写回到系统内存中
2. 这个写回内存的操作会导致其他cpu里缓存了该内存地址的数据无效。

Java内存模型也规定了工作内存与主内存之间交互的协议，定义了8种原子操作：

(1) lock:将主内存中的变量锁定，为一个线程所独占

(2) unclock:将lock加的锁定解除，此时其它的线程可以有机会访问此变量

(3) read:将主内存中的变量值读到工作内存当中

(4) load:将read读取的值保存到工作内存中的变量副本中。

(5) use:将值传递给线程的代码执行引擎

(6) assign:将执行引擎处理返回的值重新赋值给变量副本

(7) store:将变量副本的值存储到主内存中。

(8) write:将store存储的值写入到主内存的共享变量当中。

Volatile禁止指令重排：

多线程之间指令重新排序可能会导致线程A读取线程B的变量时，线程B的变量还未初始化。

为此，Java内存模型采取保守策略。下面是基于保守策略的JMM内存屏障插入策略：

在每个volatile写操作的前面插入一个StoreStore屏障。

在每个volatile写操作的后面插入一个StoreLoad屏障。

在每个volatile读操作的后面插入一个LoadLoad屏障。

在每个volatile读操作的后面插入一个LoadStore屏障。

**LoadLoad屏障：**对于这样的语句Load1; LoadLoad; Load2，在Load2及后续读取操作要读取的数据被访问前，保证Load1要读取的数据被读取完毕。  
**StoreStore屏障：**对于这样的语句Store1; StoreStore; Store2，在Store2及后续写入操作执行前，保证Store1的写入操作对其它处理器可见。  
**LoadStore屏障：**对于这样的语句Load1; LoadStore; Store2，在Store2及后续写入操作被刷出前，保证Load1要读取的数据被读取完毕。  
**StoreLoad屏障：**对于这样的语句Store1; StoreLoad; Load2，在Load2及后续所有读取操作执行前，保证Store1的写入对所有处理器可见。它的开销是四种屏障中最大的。        在大多数处理器的实现中，这个屏障是个万能屏障，兼具其它三种内存屏障的功能。

* 1. synchronized

synchronized通过锁机制实现同步。

先来看下利用synchronized实现同步的基础：Java中的每一个对象都可以作为锁。

具体表现为以下3种形式。

* **对于普通同步方法，锁是当前实例对象。**
* **对于静态同步方法，锁是当前类的Class对象。**
* **对于同步方法块，锁是Synchonized括号里配置的对象。**
  + 1. sychronized实现原理

synchronized是基于**Java对象头**和**Monitor**来实现同步的。

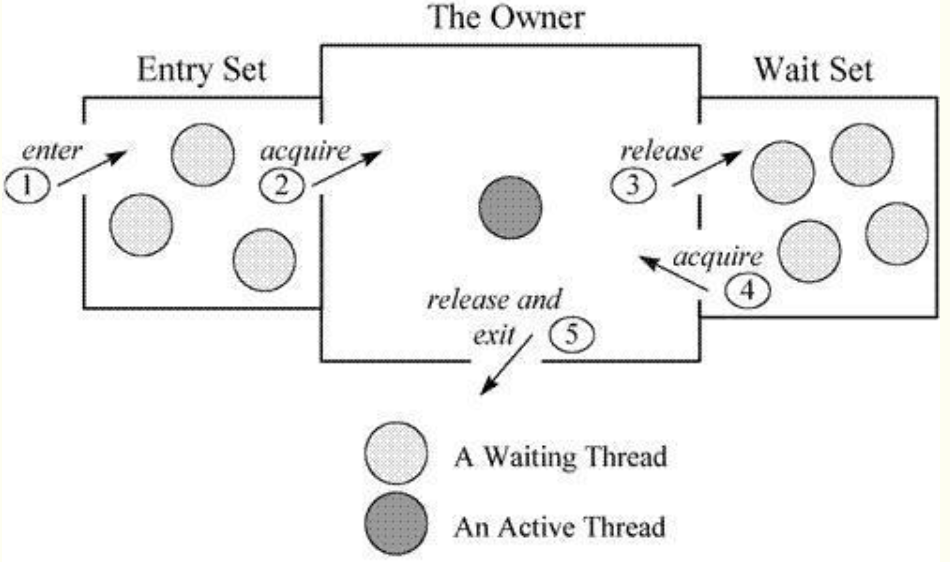
Monitor从两个方面来支持线程之间的同步：

* 互斥执行
* 协作

1、Java 使用对象锁 ( 使用 synchronized 获得对象锁 ) 保证工作在共享的数据集上的线程互斥执行。

2、使用 notify/notifyAll/wait 方法来协同不同线程之间的工作。

3、Class和Object都关联了一个Monitor。



Java Monitor 从两个方面来支持线程之间的同步，即：互斥执行与协作。 Java 使用对象锁 ( 使用 synchronized 获得对象锁 ) 保证工作在共享的数据集上的线程互斥执行 , 使用 notify/notifyAll/wait 方法来协同不同线程之间的工作。这些方法在 Object 类上被定义，会被所有的 Java 对象自动继承。

1.线程进入同步方法中。

2.为了继续执行临界区代码，线程必须获取 Monitor 锁。如果获取锁成功，将成为该监视者对象的拥有者。任一时刻内，监视者对象只属于一个活动线程（The Owner）

3.拥有监视者对象的线程可以调用 wait() 进入等待集合（Wait Set），同时释放监视锁，进入等待状态。

4.其他线程调用 notify() / notifyAll() 接口唤醒等待集合中的线程，这些等待的线程需要重新获取监视锁后才能执行 wait() 之后的代码。

进入锁同步后执行monitorenter，退出锁后执行monitorexit。

1. 锁优化

**JDK1.6后对锁**的实现引入了大量的优化，如自旋锁，适应性自旋锁，锁消除，锁粗化，偏向锁，轻量级锁来减少锁的开锁。

* 1. **自旋锁**

线程的阻塞和唤醒，需要cpu从用户态转为核心态，频繁的阻塞和唤醒对cpu来所是一种负担很重的工作。对象的锁状态只会持续一段时间，为了这一段很短的时间，频繁的阻塞和唤醒线程是非常不值得的。

所谓自旋锁，就是让该线程等待一段时间不会被立刻挂起，看持有锁的线程是否很快释放锁。（怎么等待？执行一段无意义的循环）

自旋等待不能替代阻塞，它虽然避免了线程切换的开销，但是它占用了处理器的时间，如果持有锁的线程很快释放了锁，自旋的效率就会非常好，否则就会白白消耗掉处理的资源。

自旋的时间必须要有一个限度，如果超过了定义的时间没有获得锁，则应该挂起。

* + 1. **适应自旋锁**

自旋的次数不再是固定的，它是由前一次在同一个锁上的自旋按时间及锁的拥有着状态来决定。

如果自旋成功了，那么下次自旋的次数会加的更多；反之对于某个锁，很少由自旋能够成功的，减少自旋次数或者省略掉自旋，以免浪费处理器资源。

* 1. **锁消除**

JVM检测到不可能存在共享数据竞争。这时jvm会对这些同步锁进行锁消除，节约请求锁的时间。

锁消除的依据是逃逸分析数据支持。我们平时虽然没有显示的使用锁，但是在JDK的内置API时，如StringBuffer，Vector，Hashtable等，这个时候会隐性加锁。

* 1. **锁粗化**

如果是一系列加锁操作，可能会导致不必要的性能损耗，所以引入锁粗化的概念。就是将多个加锁，解锁操作连接在一起，扩展成一个**更大范围的锁。**

* 1. 锁的升级

锁主要存在四种状态：无锁状态，偏向锁状态，轻量级锁和重量级锁。随着锁的竞争，锁可以从偏向锁升级到轻量级锁，再升级的重量级锁。再jdk1.6中默认是开启偏向锁和轻量级锁的。

* + 1. 重量级锁

重量级锁是通过对象的内部监视器（Monitor）实现。操作系统之间的线程切换，需要**用户态到内核态的切换**，切换成本非常高。

* + 1. 轻量级锁

轻量级锁并不是用来代替重量级锁的，它的本意是在没有多线程竞争的前提下，减少传统对的重量级锁使用产生的性能消耗。轻量级锁适应的应用场景是**线程交替执行同步快的情况**，如果在同一时间访问同一锁的情况，就会导致轻量级锁升级为重量级锁。

轻量级锁枷锁过程：

* 1. 在代码进入同步代码快的时候，如果同步对象锁状态为无锁状态，虚拟机首先将在当前线程中建立一个名为锁记录（Lock Record）的空间，用于存储对象目前Mark Word的拷贝。
  2. Copy对象中mark word到锁记录中
  3. 虚拟机使用CAS操作尝试将对象的mark word更新为指向lock record的指针，并将lock record里的owner指针指向object mark word，如果更新成功则执行4，否则是执行5
  4. 如果这个更新操作成功了，表明这个线程拥有了该对象的锁，并且对象mark word的锁标志位设置为“00“
  5. 如果这个更新操作失败了，虚拟机会检查对象的mark word是否指向当前线程的栈帧，如果是就说明当前线程已经拥有了这个对象的锁，那么就可以进入同步代码块继续执行。否则说明多个线程竞争锁，轻量级锁就要膨胀为重量级锁，锁标志的状态值变为“10”。

轻量级锁释放锁的过程：

* + 1. 通过cas操作尝试把线程中复制的displace mark word对象替换当前的mark word
    2. 如果替换成功了，整个同步过程就完成了
    3. 如果替换失败，说明有其他线程尝试获取锁，那么在释放锁的同时唤醒被挂起的锁
    4. 偏向锁

引入偏向锁的目的是：为了在无多线程竞争的情况下，尽量减少不必要的轻量级锁执行路径。因为轻量级锁获取及释放依赖多次CAS原子指令，而偏向锁只需要再置换ThreadId的时候依赖一次CAS原子指令。

偏向锁获取过程：

1.访问mark word中偏向锁的标志是否设置成1，锁标志位是否为01.

2.如果为可偏向状态，则测试线程id是否指向当前线程，如果是则进入步骤5，否则进3

3.如果线程id并未指向当前线程，则通过cas操作竞争锁。如果竞争成功，则将mark word中线程id设置为当前线程id，然后执行5，失败执行4

4.如果cas获取偏向锁失败，则表示有竞争。当达到全局安全点时获得偏向锁的线程被挂起，偏向锁升级为轻量锁，然后阻塞在安全点的线程继续往下执行同步代码。

5.执行同步代码

偏向锁的释放：

偏向锁的撤销在上述第四步骤中有提到。偏向锁只有遇到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，线程不会主动去释放锁。偏向锁的撤离，需要等待全局安全点（在这个时间上没有字节码正在执行），它首先暂停拥有偏向锁的线程，判断锁对象是否处于被锁定状态，撤销偏向锁后恢复到未锁定或轻量级锁的状态。

|  |  |  |  |
| --- | --- | --- | --- |
| 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。 |

Synchronized和lock

Synchronized时java的一个关键字，是java语言内置的特性。Lock是一个类，通过这个类可以实现同步访问。Lock（java.util.concurrent.locks）需要手动释放锁

如果一个代码快被synchonized修饰了，当一个线程获取了对应的锁，并执行代码块时，其他线程便只能一直等待，等待获取锁线程释放锁。如果多线程只是读操作，线程之间不会发生冲突，通过lock就可以办到。另外lock可以知道的是否成功的获取了锁。

Lock中方法：lock（），trylock（），tryLock(long time, TimeUnit unit)和lockInterruptibly()是用来获取锁。Unlock（）释放锁。

Lock：如果锁已被其他线程获取，则进行等待。采用lock必须手动释放锁，并且在发生异常的时候，不会主动释放锁。因此lock一般在try{}catch{}中进行

Trylock(): 方法有返回值，表示尝试获得锁，如果获取成功，则返回true；如果获取失败，则返回false，这个方法无论怎样都会立刻返回，不会拿不到锁一直在等待。

tryLock(long time, TimeUnit unit)：方法和tryLock()方法是类似的，只不过区别在于这个方法在拿不到锁时会等待一定的时间，在时间期限之内如果还拿不到锁，就返回false。如果如果一开始拿到锁或者在等待期间内拿到了锁，则返回true。

LockInterruptibly（）：通过这个方法去获取锁时，如果线程正在等待获取锁，则这个线程能够响应中断，即中断线程的等待状态。（也就是说，当两个线程通过通过lockinterruptibly获取某个锁，假设线程A获取了锁，而线程B只有在等待，那么线程B调用thredB.interrupt（）方法能够中断线程B的等待过程）。该方法也要捕获异常。

Reentrantlock

“可重入锁“，它是唯一实现lock接口的类。支持重入性，表示能够对共享资源重复加锁，即当前线程获取该锁再次获取不会被阻塞。Reentranlock支持公平锁和非公平锁两种方式。

1. reentrantlock重入性的实现原理、

想支持重入性，就要解决两个问题：1.在线程获取锁的时候，如果已经获取锁的线程是当前线程则再次获取成功 2.由于锁会被获取n次，那么只有锁在释放同样n次之后，该锁才算完全释放成功。