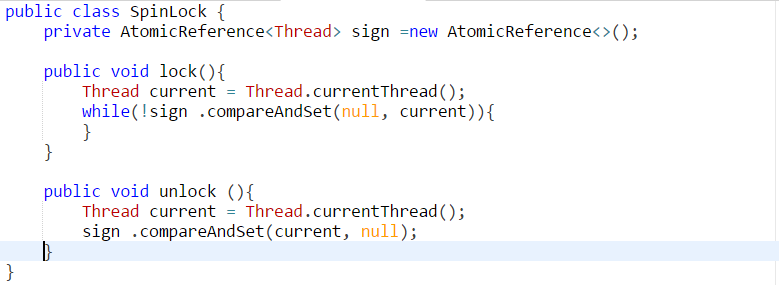
锁作为并发共享数据，保证一致性的工具，在JAVA平台有多种实现(如 synchronized 和 ReentrantLock等等 ) 。

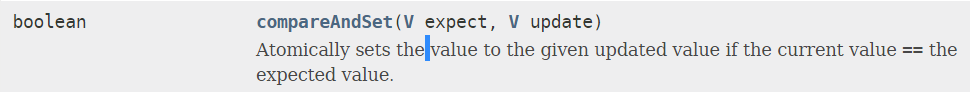
**一、自旋锁**

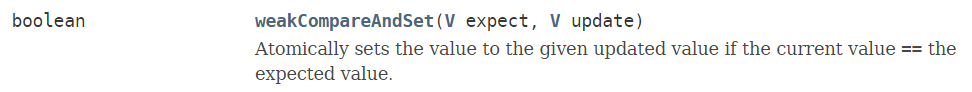
<http://ifeve.com/java_lock_see1/>

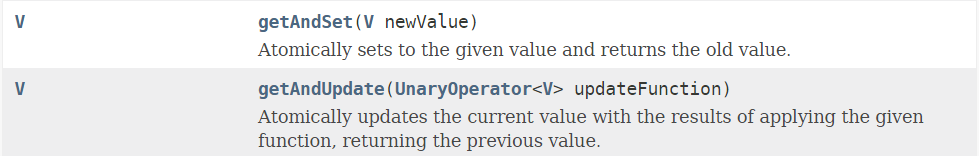
自旋锁是**采用让当前线程不停地的在循环体内执行实现的，当循环的条件被其他线程改变时才能进入临界区**。如下



AtomicReference方法：







使用了CAS原子操作，lock函数将owner设置为当前线程，并且**预测原来的值为空**。unlock函数将owner设置为null，并且**预测值为当前线程**。

当有第二个线程调用lock操作时由于owner值不为空，导致循环一直被执行，直至第一个线程调用unlock函数将owner设置为null，第二个线程才能进入**临界区**。

由于**自旋锁**只是将当前线程不停地执行循环体，**不进行线程状态的改变**，所以响应速度更快。但当线程数不停增加时，性能下降明显，因为每个线程都需要执行，占用CPU时间。如果**线程竞争不激烈，并且保持锁的时间段。适合使用自旋锁**。

**注：**该例子为非公平锁，获得锁的先后顺序，不会按照进入lock的先后顺序进行。

**二、自旋锁种类**

<http://ifeve.com/java_lock_see2/>

自旋锁中另有三种常见的锁形式：TicketLock、CLHlock 和MCSlock

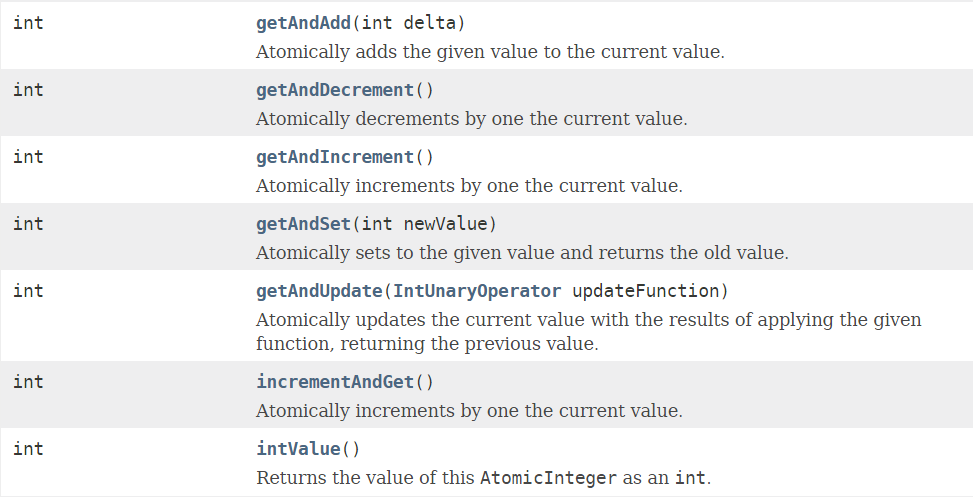
**1、TicketLock**

Ticket锁主要解决的是访问顺序的问题，主要的问题是在多核cpu上。



每次都要查询一个serviceNum 服务号，影响性能（必须要到主内存读取，并阻止其他cpu修改）。

AtomicInteger方法：



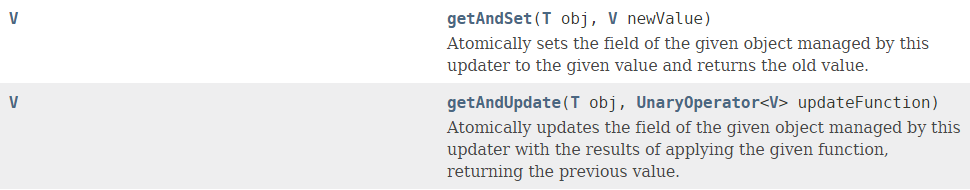
**2、CLHlock**

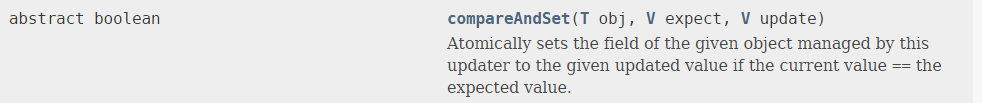
CLHLock 和MCSLock 则是两种类型相似的公平锁，采用链表的形式进行排序，

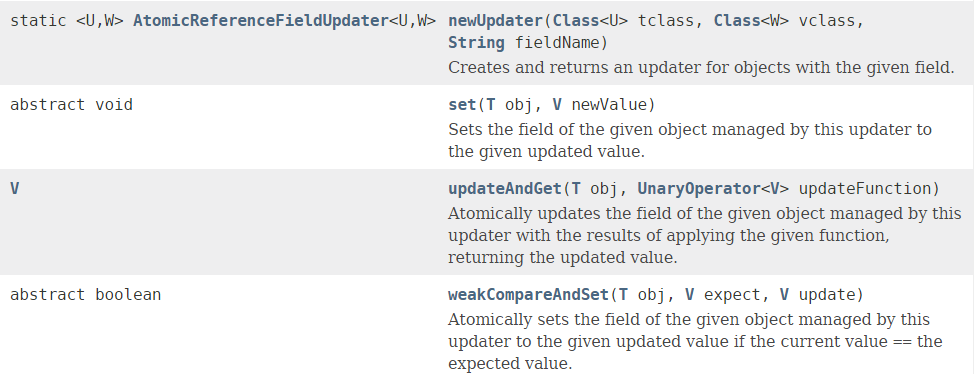


CLHlock是**不停的查询前驱变量**， 导致不适合在NUMA 架构下使用（在这种结构下，每个线程分布在不同的物理内存区域）。

AtomicReferenceFieldUpdater方法：

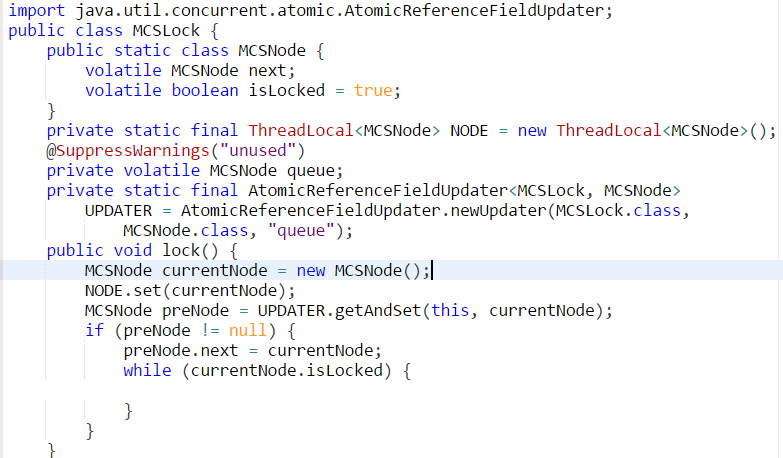


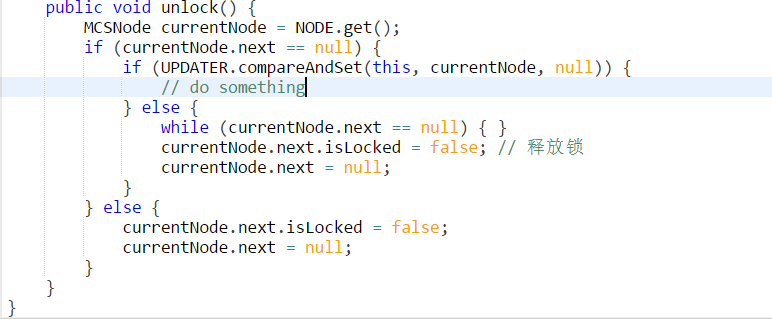




**3、MCSlock**

MCSLock则是对本地变量的节点进行循环，不存在CLHlock 的问题。





从代码上 看，CLH 要比 MCS 更简单，**CLH 的队列是隐式的队列，没有真实的后继结点属性；MCS 的队列是显式的队列，有真实的后继结点属性**。

JUC ReentrantLock 默认内部使用的锁即是 **CLH锁**（有很多改进的地方，将自旋锁换成了阻塞锁等等）。

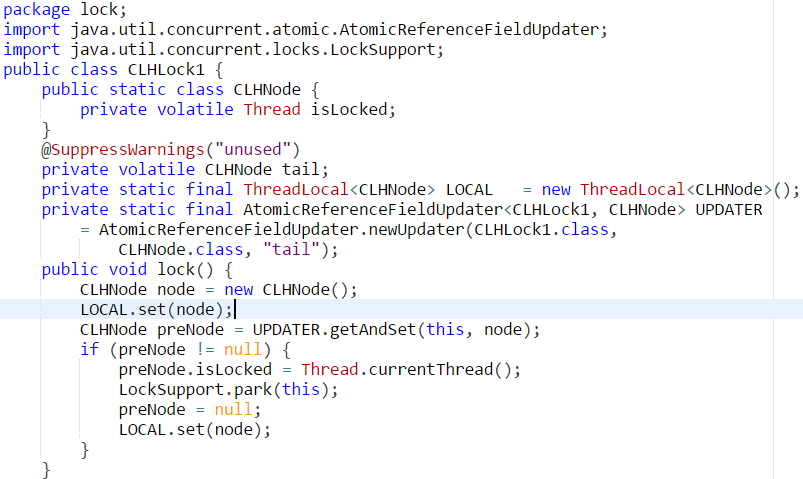
**三、阻塞锁**

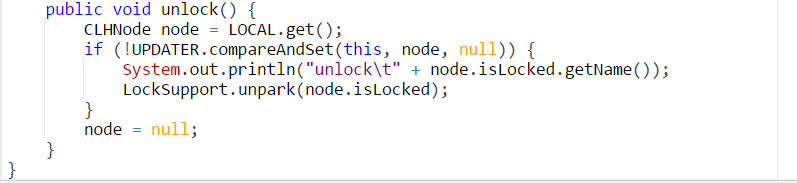
**阻塞锁，与自旋锁不同，改变了线程的运行状态**。在JAVA环境中，线程Thread有如下几个状态：新建状态、就绪状态、运行状态、阻塞状态、死亡状态

阻塞锁，可以说是让线程进入阻塞状态进行等待，当获得相应的信号（唤醒，时间） 时，才可以进入线程的准备就绪状态，准备就绪状态的所有线程，通过竞争，进入运行状态。

JAVA中，能够**进入\退出、阻塞状态或包含阻塞锁**的方法有，synchronized 关键字（其中的重量锁）、ReentrantLock、Object.wait()\notify()、**LockSupport.park()/unpart()**(j.u.c经常使用)。

下面是一个JAVA 阻塞锁实例：





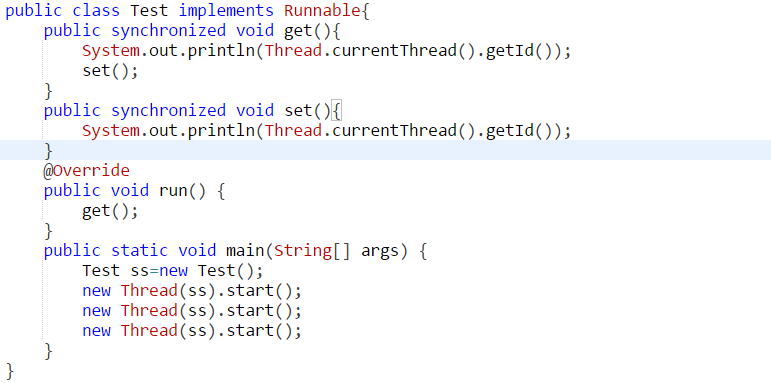
在这里我们使用了LockSupport.unpark()的阻塞锁。该例子是将CLH锁修改而成。阻塞锁的优势在于，阻塞的线程不会占用cpu时间， 不会导致 CPu占用率过高，但进入时间以及恢复时间都要比自旋锁略慢。在竞争激烈的情况下阻塞锁的性能要明显高于自旋锁。

**理想的情况则是：**在线程竞争不激烈的情况下，使用自旋锁，竞争激烈的情况下使用，阻塞锁。

**四、可重入锁**

本文里面讲的是广义上的可重入锁，而不是单指JAVA下的ReentrantLock。**可重入锁**，也叫做**递归锁**，指的是同一线程外层函数获得锁之后，内层递归函数仍然有获取该锁的代码，但不受影响。在JAVA环境下 ReentrantLock 和synchronized 都是可重入锁，可重入锁最大的作用是**避免死锁**。

实例分析：





两个例子最后的结果都是正确的，即同一个线程id被连续输出两次。

**结果如下：**

Threadid: 8

Threadid: 8

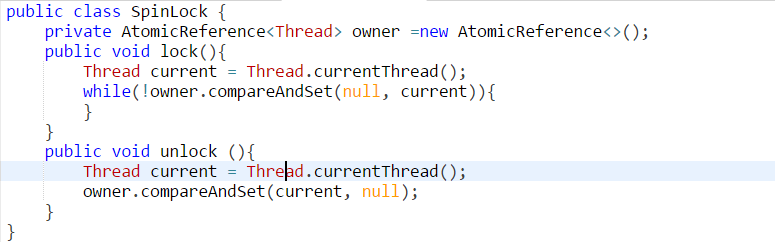
Threadid: 10

Threadid: 10

Threadid: 9

Threadid: 9

我们以自旋锁作为例子，

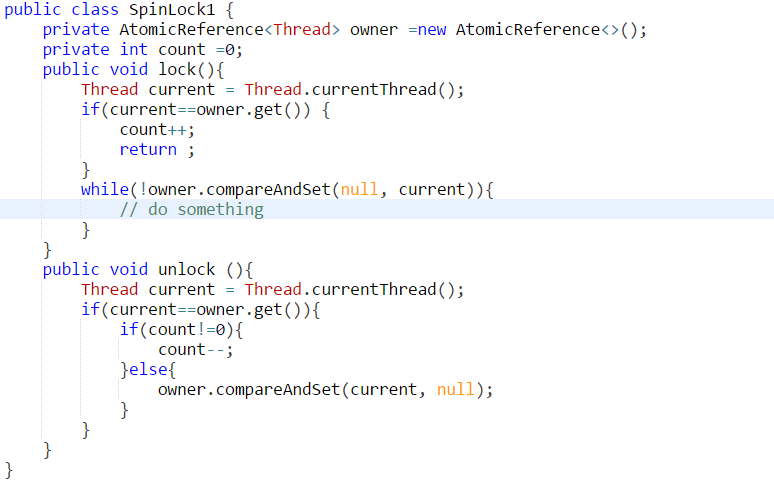


对于自旋锁来说：

1、若有同一线程两调用lock() ，会导致第二次调用lock位置进行自旋，产生了死锁，说明这个锁并不是可重入的。（在lock函数内，应验证线程是否为已经获得锁的线程）

2、若1问题已经解决，当unlock（）第一次调用时，就已经将锁释放了。实际上不应释放锁。

（采用计数次进行统计）修改之后，如下：



该自旋锁即为可重入锁。