# ReentrantReadWriteLock(jdk1.7.0\_67)

ReentrantReadWtiteLock实现了ReadWriteLock接口，同样与ReentrantLock一样具有可重入性。ReentrantReadWriteLock有以下特点：

1. 对于锁的获取，不会强加给读者或者写者任何的优先权，但是支持两种策略，公平锁与非公平锁，可以通过构造器参数指定使用公平锁或者非公平锁。
2. 非公平锁，通过使用默认构造器或者指定构造器参数为false创建的对象使

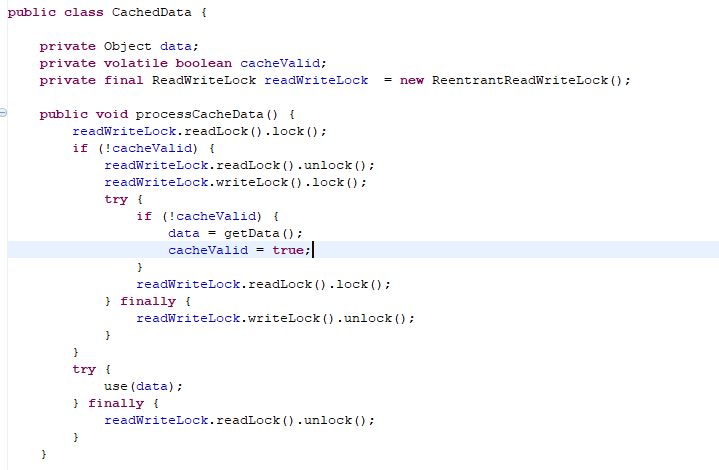
用非共平锁。非公平锁在锁争用时，可能会不定期的延期读者或写者线程，但是相较公平锁，非公平锁具有更高的性能。

b.公平锁，通过指定构造器参数为true创建的对象使用公平锁。公平锁近似遵循线程到达的顺序，当当前线程持有的锁将要释放时，最长等待的一个写线程将会获得获取写锁的资格，或者一组比写线程等待时间更长的读线程将会获得获取读锁的资格。

c.在公平锁模式下（不考虑重入），如果一个写锁已经被获取或者同步队列中存在写线程，那么企图获取读锁的线程将被阻塞，直到当前最老的等待写线程获取并释放锁为止。写锁的获取必须在读锁和写锁都空闲的状态下才可能成功否则将一直阻塞。

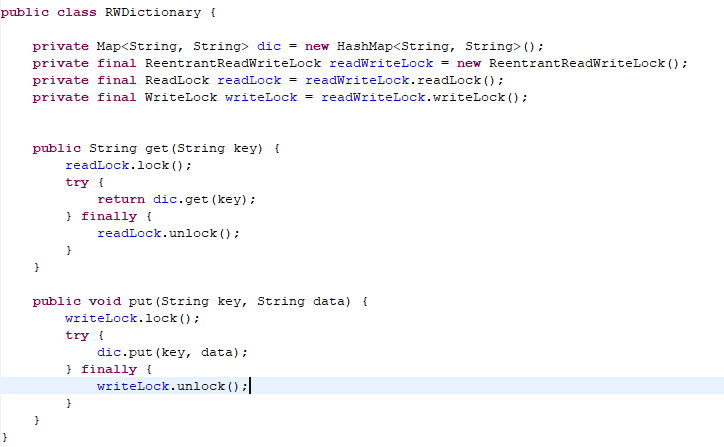
1. 可重入性，ReentrantReadWriteLock可以允许读者或者写者重复获取读锁或者写锁。另外，写者可以获取读锁，但是读者不能获取写锁。
2. 锁降级，ReentrantReadWriteLock支持从写锁降到读锁，通过先获取写锁，然后获取读锁，释放写锁的方式。但是不支持从读锁升级到写锁。

下面是官方给的一个锁降级的例子。



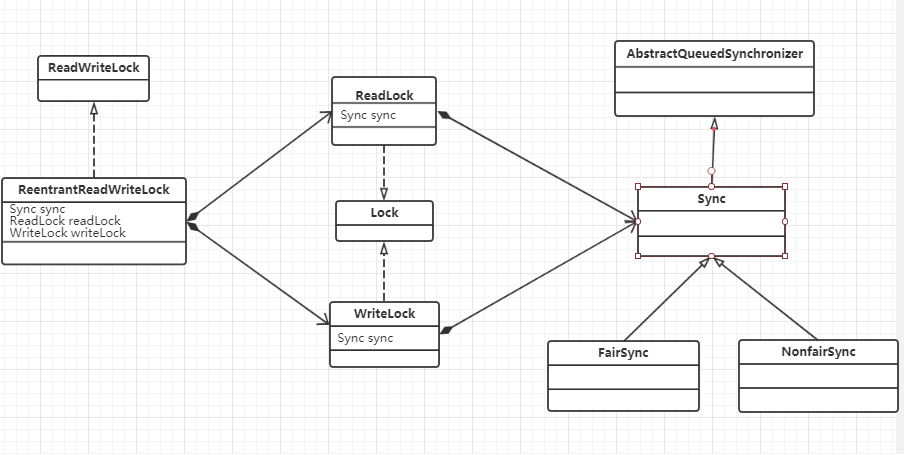
1. 写锁支持Condition，读锁不支持，在读锁上调用newCondition会抛出UnsupportedOperationException异常。

下面是一个使用ReentrantReadWriteLock读写锁的例子：



# 源码分析

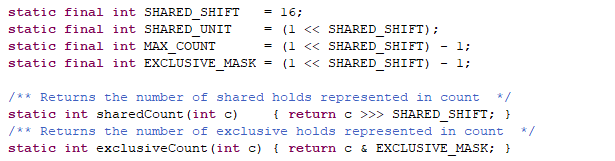
先看一下ReentrantReadWriteLock的类图。



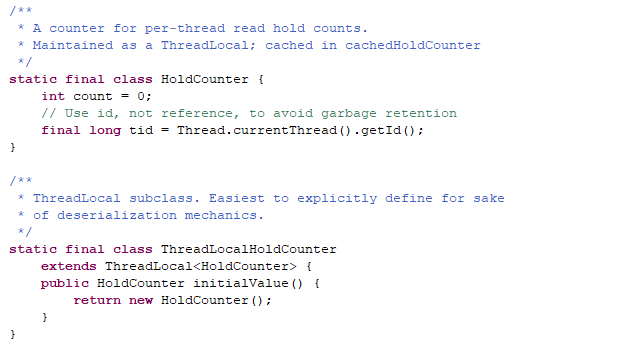
ReadLock与WriteLock是ReentrantReadWriteLock的内部类，都实现了Lock接口，ReadLock是共享锁，而WriteLock是独占锁。ReadLock和WriteLock方法的实现都委托给了Sync同步器。

## 1.1Sync

### 1.1.1属性



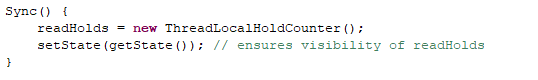
其中用状态字段的高16位表示共享线程（获取读锁的线程）的数量，低16位表示独占线程（获取写锁的线程）的数量，也就是说最多可以有65535个线程获取到锁。





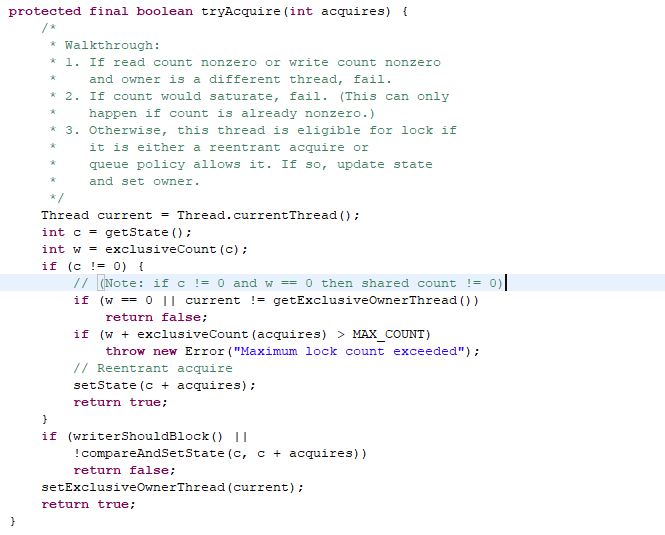
readHolds记录了每个读线程获取读锁的数量以及线程id，只有在构造函数或者readObject中被初始化，cachedHoldCounter记录了最近一次读线程获取读锁的数量，firstReader第一个获取读锁的线程，firstReaderHoldCount是firstReader获取读锁的数量。

### 1.1.2构造器



初始化了readHolds以及同步状态。

### 1.1.3tryAcquire



获取独占锁，用来被写锁调用。

（1）同步状态不为0，说明有线程获取了读锁或者写锁。

a.如果读线程数量不为0或者写线程数量不为0并且当前线程没有持有写锁，返回false。

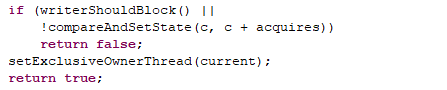


b.当前线程之前已经获取了写锁，但是再次获取的数量超过MAX\_COUNT，抛出异常。

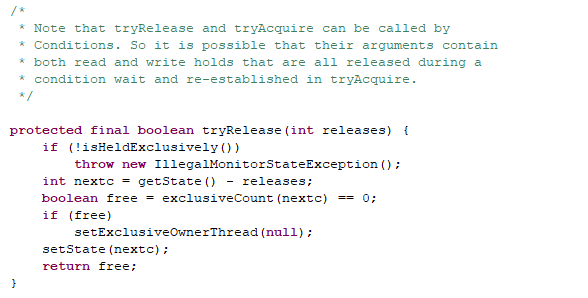


c.否则，设置同步状态。

1. 同步状态为0，说明当前既不存在读线程也不存在写线程。然后根据写者阻塞策略决定是否要设置同步状态以及更新锁的线程所有者。



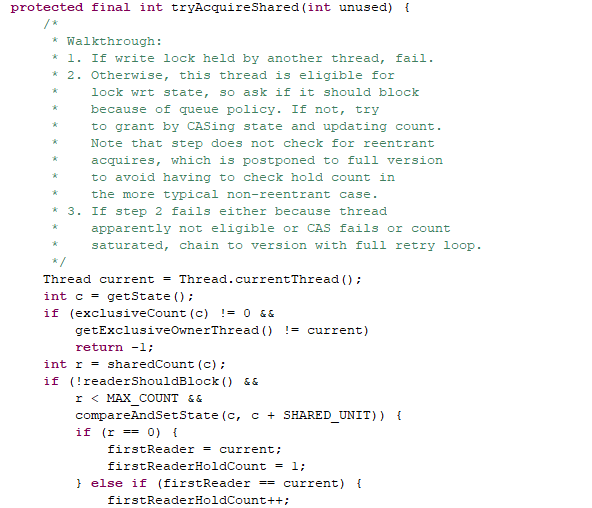
### 1.1.4tryRelease

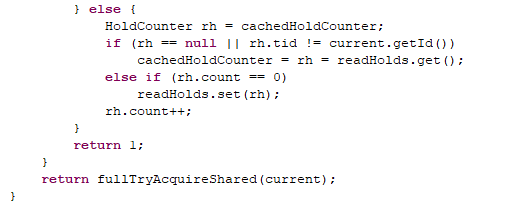


释放独占锁，用来被写锁调用。

1. 如果当前线程没有持有锁，抛出IllegalMonitorStateException异常。
2. 检查释放锁之后的状态是否为0，如果为0，说明锁已经完全释放，将锁的线程所有者设置为null。
3. 更新同步状态。

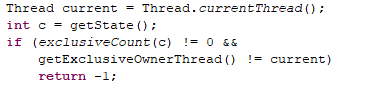
### 1.1.5tryAcquireShared



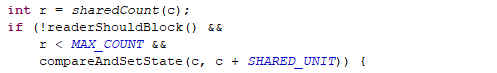


获取共享锁，用来被读锁调用。

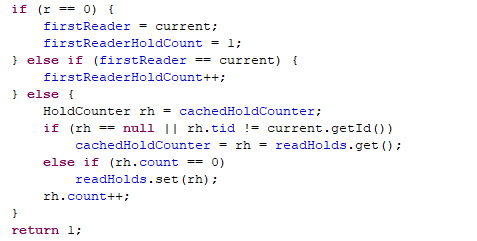
1. 如果写锁已经被其他线程获取，返回-1。



1. 根据读者阻塞策略决定是否要更新同步状态，如果读者不被阻塞并且读锁获取数量小于MAX\_COUNT，那么设置同步状态。

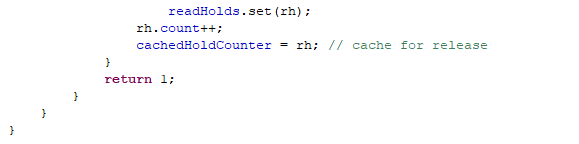
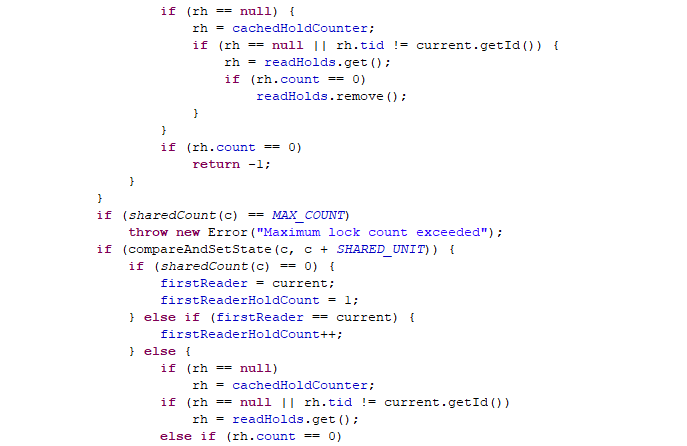
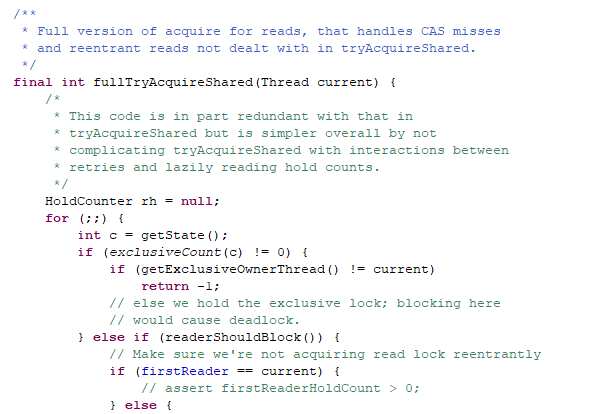


1. 同步状态设置成功后，如果线程是第一个获取读锁的线程，更新firstReader和firstReaderHoldCount，否则检查chchedHoldCounter缓存的是否是当前线程，如果不是更新chchedHoldCounter。更新当前线程读锁获取数量。返回1。

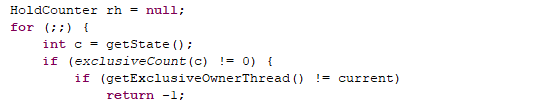


1. 如果读者被阻塞，或者读锁的获取大于等于MAX\_COUNT或者更新同步状态失败，将调用fullTryAcquireShared。

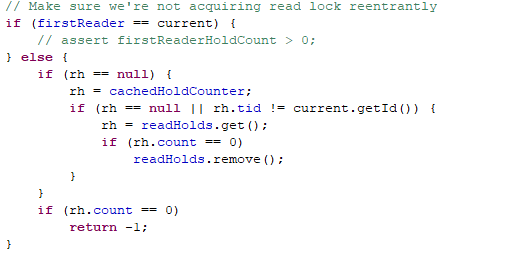
### 1.1.6fullTryAcquireShared



1. 如果写锁已经被其他线程获取，返回-1。



1. 如果写锁没有被获取并且读者应该被阻塞，检查线程是否是再次获取读锁，如果不是，返回-1。

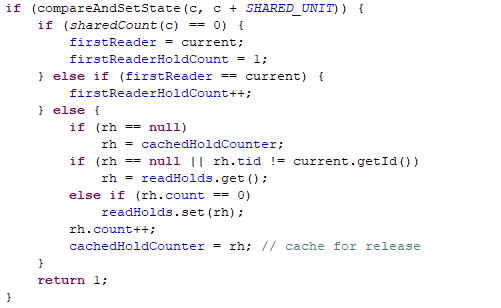


1. 如果写锁已经被当前线程获取或者当前线程再次获取读锁，

a.首先检查读锁获取数量是否超出上限。

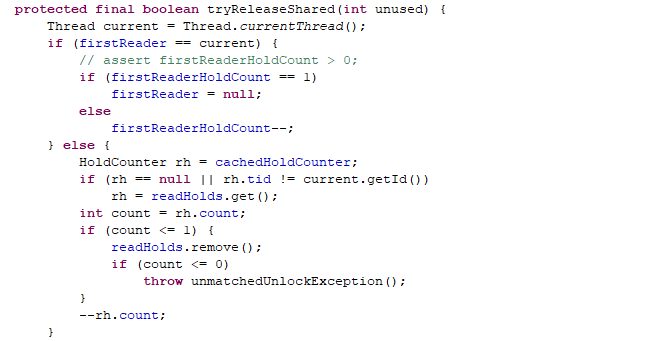


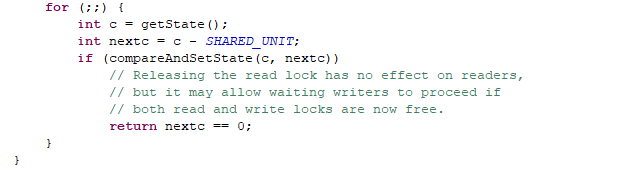
b.更新同步状态，成功之后检查是否要设置firstReader，firstReaderHoldCounter和cachedHoldCounter，更新当前线程获取读锁的数量。返回1。



c.如果更新同步状态失败，说明有并发情况，重复循环。

### 1.1.7tryReleaseShared

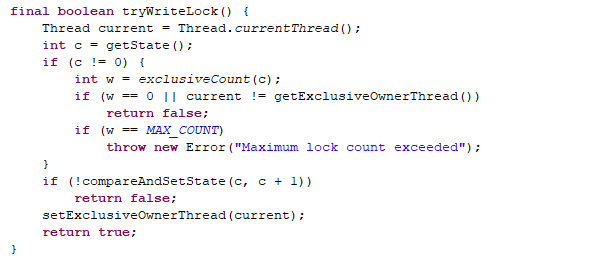




释放共享锁，该方法被读锁调用释放读锁。

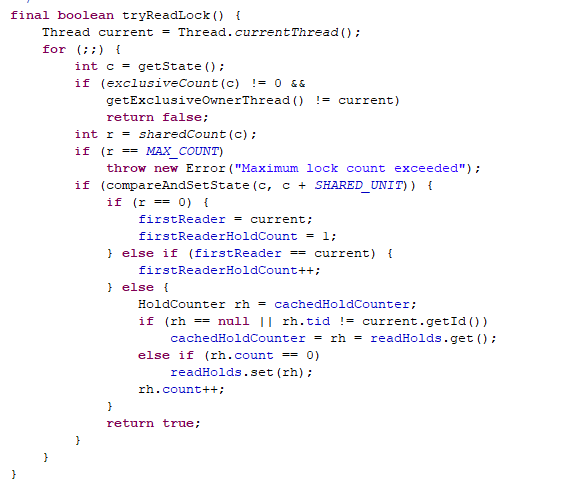
1. 如果当前线程等于firstReader，更新firstReader和firstReaderHoldCounter。否则更新当前线程获取读锁的数量。
2. 循环更新同步状态，直到更新成功为止。

### 1.1.8tryWriteLock



尝试获取写锁，如果获取失败，该方法会立即返回false。该方法不受公平策略的影响，也就是说在在公平模式下，写锁释放期间，不管同步队列中是否有线程在等待，当前线程可能立即获取到写锁。

### 1.1.9tryReadLock



tryReadLock与tryWriteLock一样，同样不受公平策略的影响。在公平锁模式下，在读锁可被获取的情境下，不管同步队列中是否有线程等待获取写锁，当前线程可能立即获取到读锁。

## 1.2NonFairSync与FairSync

NonFairSync实现了非公平锁，FairSync实现了公平锁，两者实现上的区别在writerShouldBlock和readerShouldBlock这两个方法。这两个方法决定了写者和读者在获取锁时，是否要等待同步队列中的线程先获取释放锁之后再获取锁，还是要立即获取锁。下面看一下两者不同的实现。

NonFairSync中writerShouldBlock的实现：



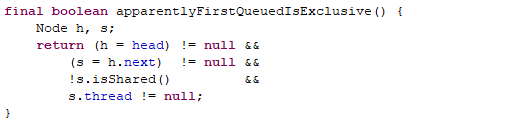
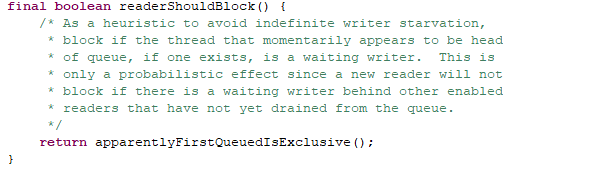
在非公平锁中，writeShouldBlock直接返回false，表示不会阻塞写者，写者可以立即获取锁。

FairSync中writerShouldBlock的实现：



在公平锁中，如果同步队列中存在等待获取锁的线程，那么将阻塞写者的获取。

NonFairSync中readerShouldBlock的实现：



如果同步队列中第一个有效的节点的线程是写者，那么读者线程将会被阻塞。这样做是为了避免写者线程尝长时间饥饿。

FairSync中readerShouldBlock的实现：



在公平锁中，如果同步队列中存在等待获取锁的线程，那么将阻塞读者的获取。

## 1.3ReadLock和WriteLock

ReadLock是读锁的实现，基于共享锁。WriteLock是写锁的实现，基于独占锁。两者的实现都委托给了Sync同步类，这里不再赘述。