**Read / Write Locks in Java**

* [Read / Write Lock Java Implementation](http://tutorials.jenkov.com/java-concurrency/read-write-locks.html#simple)
* [Read / Write Lock Reentrance](http://tutorials.jenkov.com/java-concurrency/read-write-locks.html#reentrance)
* [Read Reentrance](http://tutorials.jenkov.com/java-concurrency/read-write-locks.html#readreentrance)
* [Write Reentrance](http://tutorials.jenkov.com/java-concurrency/read-write-locks.html#writereentrance)
* [Read to Write Reentrance](http://tutorials.jenkov.com/java-concurrency/read-write-locks.html#upgrade)
* [Write to Read Reentrance](http://tutorials.jenkov.com/java-concurrency/read-write-locks.html#downgrade)
* [Fully Reentrant ReadWriteLock](http://tutorials.jenkov.com/java-concurrency/read-write-locks.html#full)
* [Calling unlock() From a finally-clause](http://tutorials.jenkov.com/java-concurrency/read-write-locks.html#finally)

读/写锁是比[**Locks in Java**](http://tutorials.jenkov.com/java-concurrency/locks.html)文中展示的Lock实现更精妙的锁。想像你一个应用读写一些资源，但是写不像读那么多。两个线程读相同资源对彼此不会引起问题，因此想要读资源的多线程被同时重叠授权访问。但是，如果一个线程想要写资源，其他线程读或写一定不能同时进行。为了解决这个允许多个读但只有一个写的问题，你需要一个读/写锁。

Java 5附带java.util.concurrent包中的读/写锁实现。即便如此，了解其实施背后的理论仍然可能是有用的。

**Read / Write Lock Java Implementation**

首先让我们总结一下获得资源的读写访问的条件：

|  |  |
| --- | --- |
| **读访问** | 如果没有线程正在写，没有线程请求写权限。 |
| **写访问** | 如果没有线程正在读或写。 |

如果一个线程想要读资源，只要没有线程正在写入，并且没有线程已经请求了资源的写访问就可以。通过提升写访问请求的优先级我们假定写请求比读请求重要得多。另外，如果读是发生最频繁的，并且我们不想提升写的优先级，则饥饿([**starvation**](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html))就会发生。请求写访问的线程会被阻塞直到全部读解锁了ReadWriteLock。如果新线程频繁授予读访问，等待写访问的线程将无限期保持阻塞，导致饥饿([**starvation**](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html))。因此只有当没有线程已经锁定ReadWriteLock来写或者已经请求锁定它来写，线程才可以授权读访问。

一个想要对资源写访问的线程能被授权如此，当没有线程正在对资源读或者写。无所谓多少线程已经请求写访问或者在什么序列，除非你想要保证请求写访问的线程之间的公平。

考虑到这些简单的规则，我们可以实现如下所示的ReadWriteLock ：

public class ReadWriteLock{

private int readers = 0;

private int writers = 0;

private int writeRequests = 0;

public synchronized void lockRead() throws InterruptedException{

while(writers > 0 || writeRequests > 0){

wait();

}

readers++;

}

public synchronized void unlockRead(){

readers--;

notifyAll();

}

public synchronized void lockWrite() throws InterruptedException{

writeRequests++;

while(readers > 0 || writers > 0){

wait();

}

writeRequests--;

writers++;

}

public synchronized void unlockWrite() throws InterruptedException{

writers--;

notifyAll();

}

}

ReadWriteLock 有两个锁方法和两个解锁方法。一个加锁和解锁方法对于读访问，一个加锁和解锁对于写访问。

读访问的规则在lockRead()方法中实现。所有线程获取读访问除非有一个线程有写访问，或者一个或更多线程已经请求写访问。

写访问的规则在lockWrite()方法中实现。想要写访问的线程起始于请求写访问(writeRequests++)。然后它会检查是否它真的可以获取写访问。一个线程可以获取写访问，如果没有线程有对资源的读访问，并且没有线程有对资源的写访问。有多少线程请求了写访问无所谓。

unlockRead()和unlockWrite()调用notifyAll()而非notify()没有任何意义。为了解释原因，想像以下场景：

在ReadWriteLock内部有线程等待读访问，有线程等待写访问。如果一个被notify()唤醒的线程是一个读访问线程，它将会被放回等待，因为有线程在等待写访问。然而，没有等待写访问的线程被唤醒，因此没有事情发生。没有线程获取读或写访问。通过调用notifyAll()，所有的等待线程被唤醒并检查是否它们可以获取期望的访问。

调用notifyAll()也有另一个优势。如果多个线程正在等待读访问，没有线程等待写访问，然后unlockWrite()被调用，全部等待读访问的线程马上被授权读访问 - 而不是一个接一个。

**Read / Write Lock Reentrance**

先前展示的ReadWriteLock类不是可重入的([**reentrant**](http://tutorials.jenkov.com/java-concurrency/locks.html#reentrance))。如果一个有写访问的线程再次请求它，它将会阻塞因为已经有一个写 - 它自己。除此之外，考虑这个情形：

1. 线程1获取读访问
2. 线程2请求写访问但是被阻塞，因为有一个读
3. 线程1再次请求读访问(重进锁)，但是被阻塞因为有一个写请求

在这种场景下先前的ReadWriteLock将会锁定 - 类似死锁。没有请求读或写访问的线程将会被授权。

为了使ReadWriteLock 可重入，有必要做一些改变。对于读和写的可重入将分别处理。

**Read Reentrance**

为了使ReadWriteLock对于读可重入，我们将首先建立读可重入的规则：

* 一个线程被授权读可重入，如果它可以获取读访问(没有写或者写请求)，或者如果它已经有读访问(不管写请求)

为了决定是否一个线程已经有读访问，对每个被授权读访问的线程的引用保存在一个Map中，和它已经多少次获取读锁一起。当决定是否读访问可以被授权，将检查此Map以获取调用线程的引用。这是在修改之后lockRead()和unlockRead()方法的样子：

public class ReadWriteLock{

private Map<Thread, Integer> readingThreads =

new HashMap<Thread, Integer>();

private int writers = 0;

private int writeRequests = 0;

public synchronized void lockRead() throws InterruptedException{

Thread callingThread = Thread.currentThread();

while(! canGrantReadAccess(callingThread)){

wait();

}

readingThreads.put(callingThread,

(getAccessCount(callingThread) + 1));

}

public synchronized void unlockRead(){

Thread callingThread = Thread.currentThread();

int accessCount = getAccessCount(callingThread);

if(accessCount == 1){ readingThreads.remove(callingThread); }

else { readingThreads.put(callingThread, (accessCount -1)); }

notifyAll();

}

private boolean canGrantReadAccess(Thread callingThread){

if(writers > 0) return false;

if(isReader(callingThread) return true;

if(writeRequests > 0) return false;

return true;

}

private int getReadAccessCount(Thread callingThread){

Integer accessCount = readingThreads.get(callingThread);

if(accessCount == null) return 0;

return accessCount.intValue();

}

private boolean isReader(Thread callingThread){

return readingThreads.get(callingThread) != null;

}

}

正如所见，读可重入被授权只有如果当前没有线程写资源。另外，如果调用线程已经有读访问，它的优先级高于任何写请求。

**Write Reentrance**

写可重入被授权仅当线程已经有写访问。这是lockWrite()和unlockWrite()方法在该改动后的样子：

public class ReadWriteLock{

private Map<Thread, Integer> readingThreads =

new HashMap<Thread, Integer>();

private int writeAccesses = 0;

private int writeRequests = 0;

private Thread writingThread = null;

public synchronized void lockWrite() throws InterruptedException{

writeRequests++;

Thread callingThread = Thread.currentThread();

while(! canGrantWriteAccess(callingThread)){

wait();

}

writeRequests--;

writeAccesses++;

writingThread = callingThread;

}

public synchronized void unlockWrite() throws InterruptedException{

writeAccesses--;

if(writeAccesses == 0){

writingThread = null;

}

notifyAll();

}

private boolean canGrantWriteAccess(Thread callingThread){

if(hasReaders()) return false;

if(writingThread == null) return true;

if(!isWriter(callingThread)) return false;

return true;

}

private boolean hasReaders(){

return readingThreads.size() > 0;

}

private boolean isWriter(Thread callingThread){

return writingThread == callingThread;

}

}

注意在确定调用线程是否可以获得写访问时，当前持有写锁的线程现在纳入考虑。

**Read to Write Reentrance**

有时对于一个有读访问权限的线程，也获取写权限是必要的。为了允许这一点，线程必须是唯一的读。为了实现这一点，应该修改writeLock()方法。这就是它的样子：

public class ReadWriteLock{

private Map<Thread, Integer> readingThreads =

new HashMap<Thread, Integer>();

private int writeAccesses = 0;

private int writeRequests = 0;

private Thread writingThread = null;

public synchronized void lockWrite() throws InterruptedException{

writeRequests++;

Thread callingThread = Thread.currentThread();

while(! canGrantWriteAccess(callingThread)){

wait();

}

writeRequests--;

writeAccesses++;

writingThread = callingThread;

}

public synchronized void unlockWrite() throws InterruptedException{

writeAccesses--;

if(writeAccesses == 0){

writingThread = null;

}

notifyAll();

}

private boolean canGrantWriteAccess(Thread callingThread){

**if(isOnlyReader(callingThread)) return true;**

if(hasReaders()) return false;

if(writingThread == null) return true;

if(!isWriter(callingThread)) return false;

return true;

}

private boolean hasReaders(){

return readingThreads.size() > 0;

}

private boolean isWriter(Thread callingThread){

return writingThread == callingThread;

}

**private boolean isOnlyReader(Thread thread){**

**return readers == 1 && readingThreads.get(callingThread) != null;**

**}**

}

现在ReadWriteLock类是读到写访问可重入的。

**Write to Read Reentrance**

有时一个有写访问的线程也需要读访问。一个写应当总是被授权读访问，如果请求。如果一个线程有写访问，则没有其他的线程可以有读或者写访问，因此并不危险。这是canGrantReadAccess()方法那样修改后的样子：

public class ReadWriteLock{

private boolean canGrantReadAccess(Thread callingThread){

**if(isWriter(callingThread)) return true;**

if(writingThread != null) return false;

if(isReader(callingThread) return true;

if(writeRequests > 0) return false;

return true;

}

}

**Fully Reentrant ReadWriteLock**

下面是完全可重入的ReadWriteLock实现。我已经对访问条件做了一些重构，使它们更容易阅读，从而更容易说服自己它们是正确的。

public class ReadWriteLock{

private Map<Thread, Integer> readingThreads =

new HashMap<Thread, Integer>();

private int writeAccesses = 0;

private int writeRequests = 0;

private Thread writingThread = null;

public synchronized void lockRead() throws InterruptedException{

Thread callingThread = Thread.currentThread();

while(! canGrantReadAccess(callingThread)){

wait();

}

readingThreads.put(callingThread,

(getReadAccessCount(callingThread) + 1));

}

private boolean canGrantReadAccess(Thread callingThread){

if( isWriter(callingThread) ) return true;

if( hasWriter() ) return false;

if( isReader(callingThread) ) return true;

if( hasWriteRequests() ) return false;

return true;

}

public synchronized void unlockRead(){

Thread callingThread = Thread.currentThread();

if(!isReader(callingThread)){

throw new IllegalMonitorStateException("Calling Thread does not" +

" hold a read lock on this ReadWriteLock");

}

int accessCount = getReadAccessCount(callingThread);

if(accessCount == 1){ readingThreads.remove(callingThread); }

else { readingThreads.put(callingThread, (accessCount -1)); }

notifyAll();

}

public synchronized void lockWrite() throws InterruptedException{

writeRequests++;

Thread callingThread = Thread.currentThread();

while(! canGrantWriteAccess(callingThread)){

wait();

}

writeRequests--;

writeAccesses++;

writingThread = callingThread;

}

public synchronized void unlockWrite() throws InterruptedException{

if(!isWriter(Thread.currentThread()){

throw new IllegalMonitorStateException("Calling Thread does not" +

" hold the write lock on this ReadWriteLock");

}

writeAccesses--;

if(writeAccesses == 0){

writingThread = null;

}

notifyAll();

}

private boolean canGrantWriteAccess(Thread callingThread){

if(isOnlyReader(callingThread)) return true;

if(hasReaders()) return false;

if(writingThread == null) return true;

if(!isWriter(callingThread)) return false;

return true;

}

private int getReadAccessCount(Thread callingThread){

Integer accessCount = readingThreads.get(callingThread);

if(accessCount == null) return 0;

return accessCount.intValue();

}

private boolean hasReaders(){

return readingThreads.size() > 0;

}

private boolean isReader(Thread callingThread){

return readingThreads.get(callingThread) != null;

}

private boolean isOnlyReader(Thread callingThread){

return readingThreads.size() == 1 &&

readingThreads.get(callingThread) != null;

}

private boolean hasWriter(){

return writingThread != null;

}

private boolean isWriter(Thread callingThread){

return writingThread == callingThread;

}

private boolean hasWriteRequests(){

return this.writeRequests > 0;

}

}

**Calling unlock() From a finally-clause**

当守卫一个临界区间使用一个ReadWriteLock，临界区间可能抛出异常，那么在一个finally从句内部调用readUnlock()和writeUnlock()方法很重要。这么做确保ReadWriteLock被解开以便其他的线程可以锁定它。这是个例子：

lock.lockWrite();

try{

//do critical section code, which may throw exception

} finally {

lock.unlockWrite();

}

这个小构造确保在从临界区间中的代码中抛出异常时解锁ReadWriteLock。如果unlockWrite()没有被从finally从句内部调用，异常从临界区间抛出，则ReadWriteLock将永远保持写锁定，导致全部在ReadWriteLock实例上调用lockRead()和lockWrite()的线程无限期停止。唯一能够再次解锁ReadWriteLock的是如果ReadWriteLock是可重入的，并且在将其锁定时抛出异常的线程随后成功锁定它，执行临界区间并在之后再次调用unlockWrite()。那会再次解锁ReadWriteLock。但是为什么要等待它发生，如果它一定发生？调用unlockWrite()从一个finally从句，是一个稳健得多的解决方案。