Java中的读写锁

原文链接 **作者：**Jakob Jenkov **译者：**微凉 **校对：**丁一

相比Java中的锁(Locks in Java)里Lock实现，读写锁更复杂一些。假设你的程序中涉及到对一些共享资源的读和写操作，且写操作没有读操作那么频繁。在没有写操作的时候，两个线程同时读一个资源没有任何问题，所以应该允许多个线程能在同时读取共享资源。但是如果有一个线程想去写这些共享资源，就不应该再有其它线程对该资源进行读或写（*译者注：也就是说：读-读能共存，读-写不能共存，写-写不能共存*）。这就需要一个读/写锁来解决这个问题。

Java5在java.util.concurrent包中已经包含了读写锁。尽管如此，我们还是应该了解其实现背后的原理。

以下是本文的主题

读/写锁的Java实现

先让我们对读写访问资源的条件做个概述：

读取 没有线程正在做写操作，且没有线程在请求写操作。

写入 没有线程正在做读写操作。

如果某个线程想要读取资源，只要没有线程正在对该资源进行写操作且没有线程请求对该资源的写操作即可。我们假设对写操作的请求比对读操作的请求更重要，就要提升写请求的优先级。此外，如果读操作发生的比较频繁，我们又没有提升写操作的优先级，那么就会产生“饥饿”现象。请求写操作的线程会一直阻塞，直到所有的读线程都从ReadWriteLock上解锁了。如果一直保证新线程的读操作权限，那么等待写操作的线程就会一直阻塞下去，结果就是发生“饥饿”。因此，只有当没有线程正在锁住ReadWriteLock进行写操作，且没有线程请求该锁准备执行写操作时，才能保证读操作继续。

当其它线程没有对共享资源进行读操作或者写操作时，某个线程就有可能获得该共享资源的写锁，进而对共享资源进行写操作。有多少线程请求了写锁以及以何种顺序请求写锁并不重要，除非你想保证写锁请求的公平性。

按照上面的叙述，简单的实现出一个读/写锁，代码如下

```
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类中，读锁和写锁各有一个获取锁和释放锁的方法。

读锁的实现在lockRead()中,只要没有线程拥有写锁（writers==0），且没有线程在请求写锁（writeRequests ==0），所有想获得读锁的线

程都能成功获取。

写锁的实现在lockWrite()中,当一个线程想获得写锁的时候，首先会把写锁请求数加1（writeRequests++），然后再去判断是否能够真能获得写锁，当没有线程持有读锁（readers==0 ）,且没有线程持有写锁（writers==0）时就能获得写锁。有多少线程在请求写锁并无关系。

需要注意的是，在两个释放锁的方法（unlockRead，unlockWrite）中，都调用了notifyAll方法，而不是notify。要解释这个原因，我们可以想象下面一种情形：

如果有线程在等待获取读锁，同时又有线程在等待获取写锁。如果这时其中一个等待读锁的线程被notify方法唤醒，但因为此时仍有请求写锁的线程存在（writeRequests>0），所以被唤醒的线程会再次进入阻塞状态。然而，等待写锁的线程一个也没被唤醒，就像什么也没发生过一样（*译者注：信号丢失现象*）。如果用的是notifyAll方法，所有的线程都会被唤醒，然后判断能否获得其请求的锁。

用notifyAll还有一个好处。如果有多个读线程在等待读锁且没有线程在等待写锁时，调用unlockWrite()后，所有等待读锁的线程都能立马成功获取读锁 —— 而不是一次只允许一个。

读/写锁的重入

上面实现的读/写锁(ReadWriteLock) 是不可重入的，当一个已经持有写锁的线程再次请求写锁时，就会被阻塞。原因是已经有一个写线程了——就是它自己。此外，考虑下面的例子：

1. Thread 1 获得了读锁
2. Thread 2 请求写锁，但因为Thread 1 持有了读锁，所以写锁请求被阻塞。
3. Thread 1 再想请求一次读锁，但因为Thread 2处于请求写锁的状态，所以想再次获取读锁也会被阻塞。

上面这种情形使用前面的ReadWriteLock就会被锁定——一种类似于死锁的情形。不会再有线程能够成功获取读锁或写锁了。

为了让ReadWriteLock可重入，需要对它做一些改进。下面会分别处理读锁的重入和写锁的重入。

读锁重入

为了让ReadWriteLock的读锁可重入，我们要先为读锁重入建立规则：

- 要保证某个线程中的读锁可重入，要么满足获取读锁的条件（没有写或写请求），要么已经持有读锁（不管是否有写请求）。

要确定一个线程是否已经持有读锁，可以用一个map来存储已经持有读锁的线程以及对应线程获取读锁的次数，当需要判断某个线程能否获得读锁时，就利用map中存储的数据进行判断。下面是方法lockRead和unlockRead修改后的的代码：

```
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;
 }
}
```

代码中我们可以看到，只有在没有线程拥有写锁的情况下才允许读锁的重入。此外，重入的读锁比写锁优先级高。

写锁重入

仅当一个线程已经持有写锁，才允许写锁重入（再次获得写锁）。下面是方法lockWrite和unlockWrite修改后的的代码。

```
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;
 }
}
```

注意在确定当前线程是否能够获取写锁的时候，是如何处理的。

读锁升级到写锁

有时，我们希望一个拥有读锁的线程，也能获得写锁。想要允许这样的操作，要求这个线程是唯一一个拥有读锁的线程。writeLock()需要做点改动来达到这个目的：

```
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 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类就可以从读锁升级到写锁了。

写锁降级到读锁

有时拥有写锁的线程也希望得到读锁。如果一个线程拥有了写锁，那么自然其它线程是不可能拥有读锁或写锁了。所以对于一个拥有写锁的线程，再获得读锁，是不会有什么危险的。我们仅仅需要对上面canGrantReadAccess方法进行简单地修改：

```
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;
 }
}
```

可重入的ReadWriteLock的完整实现

下面是完整的ReadWriteLock实现。为了便于代码的阅读与理解，简单对上面的代码做了重构。重构后的代码如下。

```
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;
 }
}
```

在finally中调用unlock()

在利用ReadWriteLock来保护临界区时，如果临界区可能抛出异常，在finally块中调用readUnlock()和writeUnlock()就显得很重要了。这样做是为了保证ReadWriteLock能被成功解锁，然后其它线程可以请求到该锁。这里有个例子：

```
lock.lockWrite();
try{
 //do critical section code, which may throw exception
} finally {
 lock.unlockWrite();
}
```

上面这样的代码结构能够保证临界区中抛出异常时ReadWriteLock也会被释放。如果unlockWrite方法不是在finally块中调用的，当临界区抛出了异常时，ReadWriteLock 会一直保持在写锁定状态，就会导致所有调用lockRead()或lockWrite()的线程一直阻塞。唯一能够重新解锁ReadWriteLock的因素可能就是ReadWriteLock是可重入的，当抛出异常时，这个线程后续还可以成功获取这把锁，然后执行临界区以及再次调用unlockWrite()，这就会再次释放ReadWriteLock。但是如果该线程后续不再获取这把锁了呢？所以，在finally中调用unlockWrite对写出健壮代码是很重要的。