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

1. 读/写锁的Java实现(Read / Write Lock Java Implementation)
2. 读/写锁的重入(Read / Write Lock Reentrance)
3. 读锁重入(Read Reentrance)
4. 写锁重入(Write Reentrance)
5. 读锁升级到写锁(Read to Write Reentrance)
6. 写锁降级到读锁(Write to Read Reentrance)
7. 可重入的ReadWriteLock的完整实现(Fully Reentrant ReadWriteLock)
8. 在finally中调用unlock() (Calling unlock() from a finally-clause)

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

```
01 public class ReadWriteLock{
02     private int readers = 0;
03     private int writers = 0;
04     private int writeRequests = 0;
05 
06     public synchronized void lockRead()
07         throws InterruptedException{
08         while(writers > 0 || writeRequests > 0){
09             wait();
10         }
11         readers++;
12     }
13 
14     public synchronized void unlockRead(){
15         readers--;
16         notifyAll();
17     }
18 
19     public synchronized void lockWrite()
20         throws InterruptedException{
21         writeRequests++;
22 
23         while(readers > 0 || writers > 0){
24             wait();
25         }
26         writeRequests--;
27         writers++;
28     }
29 
30     public synchronized void unlockWrite()
31         throws InterruptedException{
32         writers--;
33         notifyAll();
34     }
35 }
```

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：

```
01 public class ReadWriteLock{
02     private Map<Thread, Integer> readingThreads =
03         new HashMap<Thread, Integer>();
04
05     private int writers = 0;
06     private int writeRequests = 0;
07
```

```
    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;
```

```
025         return true;
026     }
027 
028 
029     public synchronized void unlockRead(){
030         Thread callingThread = Thread.currentThread();
031         if(!isReader(callingThread)){
032             throw new IllegalMonitorStateException(
033                 "Calling Thread does not" +
034                 " hold a read lock on this ReadWriteLock");
035         }
036         int accessCount = getReadAccessCount(callingThread);
037         if(accessCount == 1){
038             readingThreads.remove(callingThread);
039         } else {
040             readingThreads.put(callingThread, (accessCount -1));
041         }
042         notifyAll();
043     }
044 
045     public synchronized void lockWrite()
046         throws InterruptedException{
047         writeRequests++;
048         Thread callingThread = Thread.currentThread();
049         while(!canGrantWriteAccess(callingThread)){
050             wait();
051         }
052         writeRequests--;
053         writeAccesses++;
054         writingThread = callingThread;
055     }
056 
057     public synchronized void unlockWrite()
058         throws InterruptedException{
059         if(!isWriter(Thread.currentThread())){
060         throw new IllegalMonitorStateException(
061             "Calling Thread does not" +
062             " hold the write lock on this ReadWriteLock");
063         }
064         writeAccesses--;
065         if(writeAccesses == 0){
066             writingThread = null;
067         }
068         notifyAll();
069     }
070 
071     private boolean canGrantWriteAccess(Thread callingThread){
072         if(isOnlyReader(callingThread)) return true;
073         if(hasReaders()) return false;
074         if(writingThread == null) return true;
075         if(!isWriter(callingThread)) return false;
076         return true;
077     }
078 
079 
080     private int getReadAccessCount(Thread callingThread){
081         Integer accessCount = readingThreads.get(callingThread);
082         if(accessCount == null) return 0;
083         return accessCount.intValue();
084     }
085 
086 
087     private boolean hasReaders(){
088         return readingThreads.size() > 0;
089     }
090 
091     private boolean isReader(Thread callingThread){
092         return readingThreads.get(callingThread) != null;
093     }
094 
095     private boolean isOnlyReader(Thread callingThread){
```

```
096         return readingThreads.size() == 1 &&
097             readingThreads.get(callingThread) != null;
098     }
099 
100     private boolean hasWriter(){
101         return writingThread != null;
102     }
103 
104     private boolean isWriter(Thread callingThread){
105         return writingThread == callingThread;
106     }
107 
108     private boolean hasWriteRequests(){
109         return this.writeRequests > 0;
110     }
111 }
```

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：

```
1 lock.lockWrite();
2 try{
3     //do critical section code, which may throw exception
4 } finally {
5     lock.unlockWrite();
6 }
```

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。

chenzujie

2013/06/05 11:37下午

private boolean canGrantReadAccess(Thread callingThread){

if(writers > 0) return false;

if(isReader(callingThread) return true;

if(writeRequests > 0) return false;

return true;

}

这个方法，为什么是先判断isReader()，我觉得更应该先判断完writers>0和writeRequests>0再判断isReader()的吧

同理

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;

}

这个方法是不是也应该先判断完hasReaders()和isWriter（ ）最后再判断writingThread==null?

Fatmind
2013/07/13 2:33下午

这是基于：

1、保证写优先级更高，不会饥饿

2、实现读锁可重入

所以：isReader() 在 writeRequests 前面判断

fatmind
2013/07/15 11:53上午

否则也会导致死锁。

假设：

1、a线程读，未释放锁

2、b线程写，被阻塞

3、a要求重入读

若按照 writers>0 -> writeRequests>0 -> isReader() 导致死锁

drift_ice
2013/10/11 11:44上午

这个方法是不是也应该先判断完hasReaders()和isWriter（ ）最后再判断writingThread==null?

考虑这种情况

1.当前没有分配写锁，线程a请求写锁

2.判断isWriter(){writingThread == callingThread;} 这个时候writingThread==null.所以isWriter()返回false。所以分支

if(!isWriter(callingThread)) return false; 满足。直接返回false.

3.意味着，线程a拿不到写锁。

所以，很明显你说的这种顺序逻辑上是错误的

Wayne
2013/12/05 11:15上午

你指的是那块代码？

if(isOnlyReader(callingThread)) return true;

if(hasReaders()) return false;

if(writingThread == null) return true;

if(!isWriter(callingThread)) return false;

就这段逻辑来说没有问题的

ygmyth
2017/03/19 2:09下午

如果没有分配写锁， 第三个 if ， if (writingThread == null) return true;，就可以获得了

或者由读锁升级为写锁

Silence
2014/03/26 9:44下午

你好，我觉得像如下这样的代码之类的，其中的notifyAll是不是应该放在条件判断为真里面执行更好呢？重入回退到最后一层再通知。个人理解，忘解答，谢谢

```
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();

}

lida
2014/05/11 8:06下午

你好，我有些疑问

1.追根究底，锁的实现就是利用了数量标志来判断，那么肯定存在多个线程对此变量的同时读写，那么这个flag是如何保证正确的呢？

2.读写锁的实现本质上还是使用到了Synchronized.. 那么，在进入Synchronized方法之前就会有信号量判断，读写锁的效率真的会高上很多？

ygmyth
2017/03/19 2:30下午

1. 使用 Synchronized 实现的读写锁，你所说的 flags 操作都在临界区里，不会有多个线程对此变量的同时读写

2. 这个教程只是给出了读写锁的一种实现，（使用 Synchronized），java 并发包下提供的锁，是基于 AQS 的实现

Oliver1127
2014/09/02 2:14下午

private boolean canGrantReadAccess(Thread callingThread) {

if(isWriter(callingThread)) return true; // 此语句是为了让“写锁”能拥有“读锁”

/*

* 原则1：此重入锁需要保证写线程优先于读线程

* 第一步，判断是否已有“写锁”（指有线程已经拥有写锁了），有则不能分配读的权限，没有则往下走

* 第二步，判断是否当前线程是否已经持有“读锁”，如果没有则继续往下走

* 第三步，判断是否有写请求，如果有则不能分配，因为要保证原则1

*/

if (hasWriter()) return false;

if (isReader(callingThread)) return true;

if (hasWriterRequest()) return false;

return true;

}

private boolean canGrantWriteAccess(Thread callingThread) {

if (isOnlyReader(callingThread)) return true;// 为了已经获得“读锁”的线程能拥有写锁

/*

* 第一步，当前有读锁，则不能分配，否则，往下走

* 第二步，再判断当前是否有写锁，writing==null则表示没有写锁，可以分配，writingThread != null 则表示有写锁，往下走

* 第三步，判断当前线程就是持锁线程（writingThread）如果是，!isWriter(callingThread)返回false，最终表示可以分配写锁（重入）

* 如果不是持锁线程，则!isWriter(callingThread)返回true，最终表示别的线程拥有写锁，不能再分配写锁

*

*/

if (hasReaders()) return false;

if (writingThread == null) return true;

if (!isWriter(callingThread)) return false;

return true;

}

qxun_dream
2014/10/17 4:45下午

读锁升级到写锁中:

private boolean isOnlyReader(Thread thread){

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

}

readers应该是readingThreads.size()吧

阳光天鹅
2014/12/20 4:47下午

1、第一段代码是读写锁的简单实现，除了不可重入、不能锁升级、降级外，有个问题：当某线程未获取锁就去释放锁会带来什么问题？

2、使用读锁升级到写锁那段代码的情况下，若是2个已经获取到读锁的线程均去尝试升级到写锁会产生什么问题？

ygmyth
2017/03/19 2:19下午

1. lock.lockRead();

try{

//do critical section code, which may throw exception

} finally {

lock.unlockRead();

}

lock 和unlock成对使用，能 unlock说明你已经获得了锁。

2. 若有两个读锁，是不能升级为写锁的 ，if (isOnlyReader(callingThread)) return true;

xuuyin
2017/09/23 10:23上午

if(!isReader(callingThread)){

throw new IllegalMonitorStateException(“Calling Thread does not” + ” hold a read lock on this ReadWriteLock”);

}

wyc199273
2017/02/13 3:14下午

在读锁重入那里，说 重入的读锁比写锁优先级高。

这个应该是重入的读锁比写请求锁优先级高吧?

ygmyth
2017/03/19 2:10下午

是的

cenyol
2017/03/16 11:23上午

有个小优化建议：

在unlock的时候直接进行notifyAll， 如果改为先判断reader(or writer)是否小于1了，在notifyAll是否更好。

mayijiaban
2020/10/21 6:41下午

关于 读锁升级到写锁

ReadWriteLock readWriteLock = new ReadWriteLock();

readWriteLock.lockRead();

readWriteLock.lockWrite();

readWriteLock.unlockWrite();

readWriteLock.unlockRead();

如果有两个线程A、B都获取了读锁，这时A线程尝试获取写锁，会wait，因为存在多个读锁；并且新的线程获取读锁也会wait，因为A线程已经请求过写锁。这就出现了死锁的现象。

lcongwu
2021/03/04 2:45下午

读锁重入部分，获得读线程的锁数量，其中调用的方法应该是getReadAccessCount()而不是getAccessCount()。

```
19
public synchronized void unlockRead(){
20
Thread callingThread = Thread.currentThread();
21
int accessCount = getAccessCount(callingThread);
22
if(accessCount == 1) {
23
readingThreads.remove(callingThread);
24
} else {
25
readingThreads.put(callingThread, (accessCount -1));
26
}
27
notifyAll();
28
}
```