A implementação do ReentrantReadWriteLock() pode ser considerada em alguns aspectos: reentrância de reader para reader, writer para writer, writer para writer e, por fim, reader para writer. Isto porque para cada caso devemos precisamos garantir que as regras do ReadWriteLock() não sejam violadas, isto é, é preciso que readers cedam o lock a writers, podendo apenas um writer escrever por vez enquanto múltiplos readers são permitidos de maneira simultânea e essas características devem ser respeitadas pelos métodos de reentrância mencionados. Abordamos então os quatro cenários

propostos:

- Reader To Reader
Consideramos um caso em que um grupo de threads esteja lendo e possuem o
readerLock() -irão ceder o lock assim que uma requisição de writer seja
feita- consequentemente, não deve haver pedidos de escrita, assim, alguma
das threads desse grupo pode requisitar um novo acesso de leitura como
reentrância e nesse caso, como certamente não há impedimentos o acesso será
concedido novamente e a thread continuará com o lock, alterando apenas o
counter de entradas para readers.

- Writer to Writer

Agora, suponha que uma thread retenha o writerLock(), ou seja, não há outras threads acessando simultaneamente. Essa thread, antes de liberar o lock e notificar todas as outras threads em espera pode requisitar novamente o writerLock(), isso é a reentrância e, como certamente não há outras threads compartilhando o lock, o acesso será garantido e ela poderá escrever novamente sem ceder sua vez. Note que para implementar este caso é necessário que a identidade do writer que

retém o lock seja conferida, pois o acesso só será
concedido caso o requisito venha da thread que já possui o lock. Essa
necessidade não está presente nos requisitos para readerLock().

- Writer to Reader
Nesse caso temos um considerado "downgrade", em que uma thread writer deseja
o readerLock(). Portanto, como as condições iniciais são exatamente as mesmas
do cenário anterior, isto é, não há outras threads compartilhando o lock,
então a thread não possui impedimentos e irá trocar para um readerLock() e
assim que o fizer, ela fará

o respectivo unlock() alertando todas as outras threads que foram postas em espera. Assim, se não houver requisições de outros writers o grupo de readers em espera ganhará o lock junto com a thread inicial.

- Reader to Writer

Por fim, temos o cenário em que uma thread deseja um "upgrade", isto é, um reader deseja obter o writerLock() sem abrir mão de sua vez com o lock. Para que essa operação seja possível é necessário que apenas uma thread retenha o readerLock(), afinal, não pode haver outras threads lendo enquanto uma thread

adquire o writerLock(), além de não haver pedidos prévios de outras threads pelo writerLock(). Agora chegamos no verdadeiro problema, pois temos dois possíveis casos: primeiro e mais simples, ao olhar o counter de readers há apenas uma thread com o acesso de leitura & essa thread é a que pediu o writerLock(), nesse caso o acesso é concedido pois só há ela mesma; o outro caso implica que tenham múltiplas threads com o readerLock() simultaneamente e uma delas requisita a reentrância como writer, dessa vez ela será bloqueada pois não é possível a

entrada enquanto outras threads possuem o lock.

O grande problema se encontra na implementação dessa rotina, uma vez que ela
vai de encontro à abordagem que mantivemos até então. Veja bem, para garantir
que a thread que requisitou a reentrância é a única reader precisamos ser
capazes de diferenciar o caso em que o readerCounter retorna um valor maior
do que 1 e há múltiplas threads com o lock, do caso em que o readerCounter
retorna um valor maior do que 1 mas há apenas uma thread que já obteve a

reentrância com reader pelo menos uma vez antes. Note que apenas guardamos até agora a identidade da thread que obteve o writerLock(), agora, porém, seria necessário comparar as identidades das threads que tem acesso de reader e, como esse modelo que prioriza writers geralmente é implementado quando o número de readers excede consideravelmente o número de writers (na outra abordagem em que readers são priorizados isso causaria stravation de writers) seria extremamente custoso manter registro de todas as threads que detém o readerLock(), portanto a biblioteca

java.util.concurrent.locks implementada
pela Oracle escolheu não adicionar esse método a fim de poupar custos.

Uma possível implementação desse método seria com auxílio de hash maps, como
proposto em http://tutorials.jenkov.com/java-concurrency/read-write-locks.html
por Jakob Jenkov, para manter registro de todas as threads que possuem o lock.
Código comentado visível em JenkovReaderToWriter.java no mesmo diretório.