Java Concurrency Tutorial – Reentrant Locks

javacodegeeks.com Byron Kiourtzoglou September 15th, 2011 view original

Java's *synchronized* keyword is a wonderful tool – it allows us a simple and reliable way to synchronize access to critical sections and it's not too hard to understand.
But sometimes we need more control over synchronization. Either we need to control types of access (read and write) separately, or it is cumbersome to use because either there is no obvious mutex or we need to maintain multiple mutexes.

Thankfully, lock utility classes were added in Java 1.5 and make these problems easier to solve.

Java Reentrant Locks

Java has a few lock implementations in the java.util.concurrent.locks package.

The general classes of locks are nicely laid out as interfaces:

- **Lock** - the simplest case of a lock which can be acquired and released
- **ReadWriteLock** - a lock implementation that has both read and write lock types – multiple read locks can be held at a time unless the exclusive write lock is held

Java provides two implementations of these locks that we care about – both of which are reentrant (this just means a thread can reacquire the same lock multiple times without any issue).

- **ReentrantLock** - as you'd expect, a reentrant Lock implementation
- **ReentrantReadWriteLock** - a reentrant ReadWriteLock implementation

Now, let's see some examples.
An Read/Write Lock Example

So how does one use a lock? It's pretty simple: just acquire and release (and never forget to release – finally is your friend!).

Imagine we have a very simple case where we need to synchronize access to a pair of variables. One is a simple value and another is derived based on some lengthy calculation. First, this is how we would perform that with the synchronized keyword.

```
public class Calculator {
    private int calculatedValue;
    private int value;

    public synchronized void calculate(int value) {
        this.value = value;
        this.calculatedValue = doMySlowCalculation(value);
    }

    public synchronized int getCalculatedValue() {
        return calculatedValue;
    }

    public synchronized int getValue() {
        return value;
    }
}
```

Simple, but if we have a lot of contention or if we perform a lot of reads and few writes, synchronization could hurt performance. Since frequently reads occur a lot more often than writes, Using a **ReadWriteLock** helps us minimize the issue:

```
public class Calculator {
    private int calculatedValue;
    private int value;
    private ReadWriteLock lock = new
ReentrantReadWriteLock();

    public void calculate(int value) {
        lock.writeLock().lock();
        try {
            this.value = value;
            this.calculatedValue =
doMySlowCalculation(value);
        } finally {
            lock.writeLock().unlock();
        }
    }

    public int getCalculatedValue() {
        lock.readLock().lock();
        try {
            return calculatedValue;
        } finally {
            lock.readLock().unlock();
        }
    }

    public int getValue() {
        lock.readLock().lock();
        try {
            return value;
        } finally {
            lock.readLock().unlock();
        }
    }
}
```

This example actually shows one big advantage using synchronized has: it is concise and more foolproof than using explicit locks. But locks give use flexibility we wouldn't otherwise have.
In the example above, we can have hundreds of threads reading the same value at once with no issue, and we only block readers when we acquire the write lock. Remember that: many readers can acquire the read lock at the same time, but there are no readers OR writers allowed when acquiring the write lock.

A More Typical Use

Our first example may leave you confused or not totally convinced that explicit locks are useful. Aren't there other uses for them that aren't so contrived? Certainly!

We at Carfey have used explicit locks to solve many problems. One example is when you have various tasks which can run concurrently, but you don't want more than one of the same type running at the same time. One clean way to implement it is with locks. It could be done with synchronized, but locks give us the ability to fail after timing out.

As a bonus, you'll note we used a mix of synchronized and explicit locks – sometimes one is just cleaner and simpler than the other.

```
public class TaskRunner {
    private Map<Class<? extends Runnable>,  Lock> mLocks =
            new HashMap<Class<? extends Runnable>,  Lock>
();

    public void runTaskUniquely(Runnable r, int
secondsToWait) {
        Lock lock = getLock(r.getClass());
        boolean acquired = lock.tryLock(secondsToWait,
TimeUnit.SECONDS);
        if (acquired) {
            try {
                r.run();
            } finally {
                lock.unlock();
```

```
                }
            } else {
                // failure code here
            }
        }

        private synchronized Lock getLock(Class clazz) {
            Lock l = mLocks.get(clazz);
            if (l == null) {
                l = new ReentrantLock();
                mLocks.put(clazz, l);
            }
            return l;
        }
    }
```

These two examples should give you a pretty good idea of how to use both plan **Locks** and **ReadWriteLocks**. As with synchronized, don't worry about reacquiring the same lock – there will be no issue in the locks provided in the JDK since they are reentrant.
Whenever you're dealing with concurrency, there are dangers. Always remember the following:

- Release all locks in finally block. This is rule 1 for a reason.
- Beware of thread starvation! The fair setting in **ReentrantLocks** may be useful if you have many readers and occasional writers that you don't want waiting forever. It's possible a writer could wait a very long time (maybe forever) if there are constantly read locks held by other threads.
- Use synchronized where possible. You will avoid bugs and keep your code cleaner.
- Use **tryLock()** if you don't want a thread waiting indefinitely to acquire a lock – this is similar to wait lock timeouts that databases have.

That's about it! If you have questions or comments, feel free to leave them below.

Reference: Java Concurrency Part 2 – Reentrant Locks from our JCG partners at the Carfey Software blog.