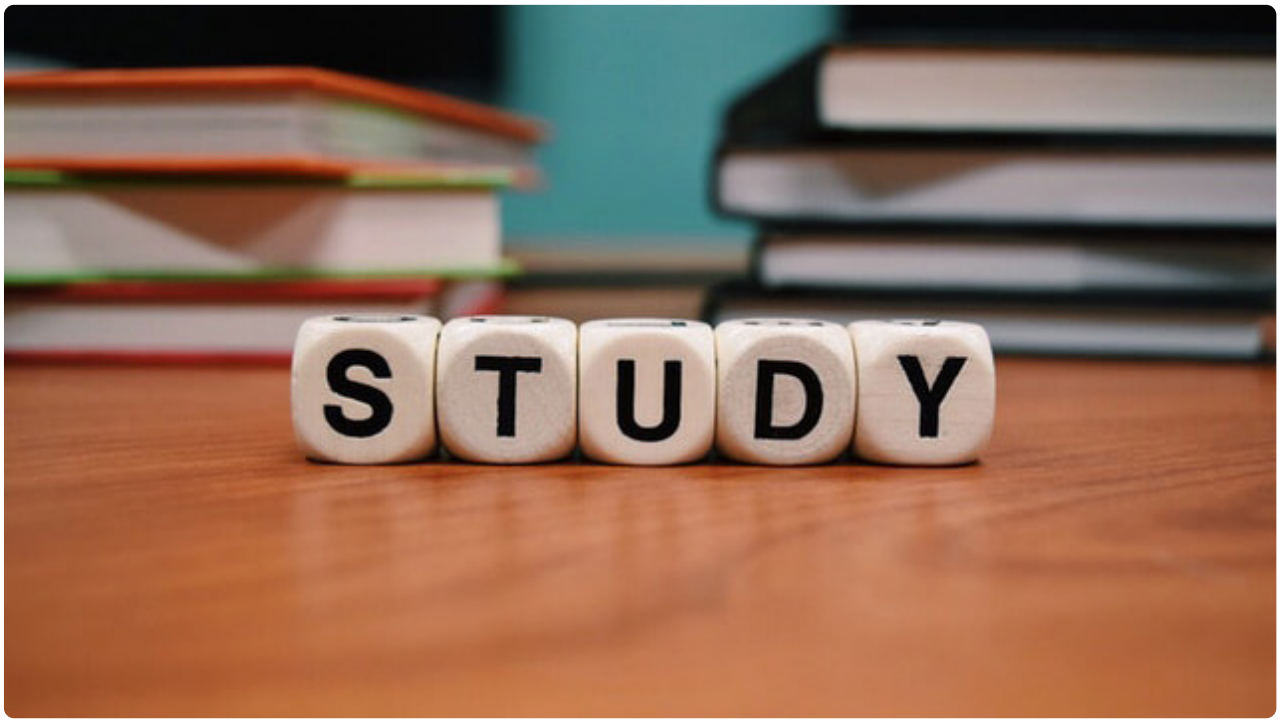
23 按需上锁—ReadWriteLock详解

更新时间：2019-11-16 17:36:26



> 合理安排时间，就等于节约时间。
>
> ——培根

前文我们分析了 java.util.concurrent.locks 包下的 Lock 接口和它的实现 ReentrantLock。ReentrantLock 是一种显式锁，提供更为高级的功能，但需要显式的上锁和解锁。ReentrantLock 也是互斥锁，有如下三种互斥情况：读/写、写/写、读/读。ReentrantLock 的加锁策略是保守的，任意操作都需要先加锁才可以.但其实在某些情况下，我们只需要控制“读/写”和“写/写”这两种互斥情况。一般情况下，绝大多数程序中读操作比例更大。其实读操作之间并不需要互斥，因为读的需求是读到最新的数据，并且在读的同时不要有其它线程修改数据即可。那我们需要的锁是“读/写”互斥，而“读/读”并不互斥。这样的好处是，读操作可以并发进行，减少了互斥的情况，能够提升程序的性能。打个比方，我们去旅游，在观景台大家可以一块欣赏风景，这就是 “读/读” 并不互斥，大家可以在观景台一块看。但是假如在观景台有个望远镜，需要使用望远镜才能看清。那么就变成了 “读/读” 互斥，人多的话显然会造成排队现象。

显然 ReentrantLock 和 synchronized 是做不到这一点的，本节我们会介绍一种新的锁—读写锁 ReadWriteLock。

1、ReadWriteLock 简介

ReadWriteLock 为我们提供了读写之间不同互斥策略的锁。因此，在某些情况下，他能够带来更好的性能。一般来说，假如你的程序有频繁的读操作，那么ReadWriteLock可能会为你带来性能的提升。但是由于读写控制的策略不一样，带来了锁内部的复杂度。所以如果你程序的读操作并没有达到一定数量，反而使用读写锁会比互斥锁性能更差。

因此 ReadWriteLock 是一种提升性能的手段，但不一定奏效。我们的程序可以尝试使用它来调节性能，如果发现没有效果或者更差，也可以很方便的换回互斥锁。

ReadWriteLock 顾名思义读写锁，也就是说同一个锁对读和写的上锁方式是不一样的写锁的互斥性更高。这里我们来看看锁降级和升级的概念。

锁降级

如果线程持有写锁，如果可以在不释放写锁的情况下，获取读锁，这就是锁降级。ReadWriteLock 是支持锁降级的。

锁升级

如果线程持有读锁，那么他是否可以不释放读锁，直接获取写锁。这意味着从一个低级别的锁升级到高级别的锁。其实就是变相的插队，无视其它在排队等待写锁的线程。ReadWriteLock 并不支持锁升级。

以上两种概念我们可以通过写段代码来体验下。

代码一：

```
public class Client {
    public static void main(String[] args) throws InterruptedException {
        ReadWriteLock lock = new ReentrantReadWriteLock();
        Lock readLock = lock.readLock();
        Lock writeLock = lock.writeLock();

        writeLock.lock();
        System.out.println("got the write lock");
        readLock.lock();
        System.out.println("got the read lock");
    }
}
```

输出：

```
got the write lock
got the read lock
```

代码二：

```
public class Client {
    public static void main(String[] args) throws InterruptedException {
        ReadWriteLock lock = new ReentrantReadWriteLock();
        Lock readLock = lock.readLock();
        Lock writeLock = lock.writeLock();

        readLock.lock();
        System.out.println("got the read lock");
        writeLock.lock();
        System.out.println("got the write lock");
    }
}
```

输出：

```
got the read lock
```

第一段代码中，我们可以在获取写锁后，再次成功获得读锁。而代码 2 中，我们在获取读锁后试图去获取写锁。这样会使得程序阻塞在 readLock.lock()。由于 writeLock 没有机会 unlock，就形成了死锁。

2、ReadWriteLock使用

ReadWriteLock 使用起来其实很简单，和 Lock 基本一致。我们使用它主要是为了对性能进行优化。我们通过下面的例子，一是熟悉它的使用，二来也可以测试下它对性能的优化效果。

首先我们看使用 Lock 的情况：

```
public class LockExample {
    String myName;
    ReentrantLock lock = new ReentrantLock();

    public void printMyName() {
        lock.lock();

        try {
            System.out.println(Thread.currentThread().getName() + "My name is " + myName);
            Thread.sleep(10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

    public void setMyName() {
        lock.lock();

        try {
            myName=Thread.currentThread().getName();
            System.out.println(Thread.currentThread().getName() + "set my name to " + myName);
        } finally {
            lock.unlock();
        }
    }

    public static void main(String[] args) {
        Long startTime = new Date().getTime();
        LockExample example = new LockExample();

        new Thread(()->{
            while (true){
                example.setMyName();

                try {
                    Thread.sleep(10);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }).start();

        new Thread(()->{
            while (true){

                example.setMyName();

                try {
                    Thread.sleep(new Random().nextInt(10));
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }).start();

        IntStream.range(0,100).forEach(num->{
            new Thread(()->{
                example.printMyName();
                System.out.println("NO. "+num+" reader finished. Time passed: "+(new Date().getTime()-startTime));
            }).start();
        });
    }
}
```

以上代码我们启动了两个线程不断写入，每次间隔 10ms， 以让其它线程能够获取到锁。另外有 100 个读线程 ，每次读取完成，睡眠 10ms，目的是延迟读锁的释放。由于使用了排他锁，所以读取操作间是互斥的，每次读取都要等 10ms 释放锁后，其它线程才能读取。那么 100 次读取就至少花费了 100*10 = 1000ms。再加上其它消耗，所以最终全部读取线程完成工作的时候，过去了 1182ms。输出如下：

```
NO. 96 reader finished. Time passed: 1148
Thread-99My name is Thread-0
NO. 97 reader finished. Time passed: 1160
Thread-100My name is Thread-0
NO. 98 reader finished. Time passed: 1172
Thread-101My name is Thread-0
NO. 99 reader finished. Time passed: 1182
```

接下来我们改造下程序，改为 ReadWriteLock，代码如下：

```
public class ReadWriteLockExample {
    String myName;
    ReentrantReadWriteLock lock = new ReentrantReadWriteLock();

    public void printMyName() {
        lock.readLock().lock();

        try {
            System.out.println(Thread.currentThread().getName() + "My name is " + myName);
            Thread.sleep(10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.readLock().unlock();
        }
    }

    public void setMyName() {
        lock.writeLock().lock();

        try {
            myName=Thread.currentThread().getName();
            System.out.println(Thread.currentThread().getName() + "set my name to " + myName);
        } finally {
            lock.writeLock().unlock();
        }
    }

    public static void main(String[] args) {
        Long startTime = new Date().getTime();
        ReadWriteLockExample example = new ReadWriteLockExample();

        new Thread(()->{
            while (true){
                example.setMyName();

                try {
                    Thread.sleep(10);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }).start();

        new Thread(()->{
            while (true){

                example.setMyName();

                try {
                    Thread.sleep(new Random().nextInt(10));
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }).start();

        IntStream.range(0,100).forEach(num->{
            new Thread(()->{
                example.printMyName();
                System.out.println("NO. "+num+" reader finished. Time passed: "+(new Date().getTime()-startTime));
            }).start();
        });
    }
}
```

可以看到只是把锁换成了 ReentrantReadWriteLock，然后 printMyName 中使用读锁，setMyName 中使用写锁。运行后输出如下：

```
NO. 92 reader finished. Time passed: 97
NO. 96 reader finished. Time passed: 97
Thread-0set my name to Thread-0
NO. 98 reader finished. Time passed: 97
Thread-1set my name to Thread-1
NO. 99 reader finished. Time passed: 97
```

最后一个 reader 完成工作，只用了 97ms，对比起使用互斥锁的 1182ms，速度提升了 10倍以上！原因就是读操作之间不会互斥，可以并发读取。从而性能大幅度得到提升。

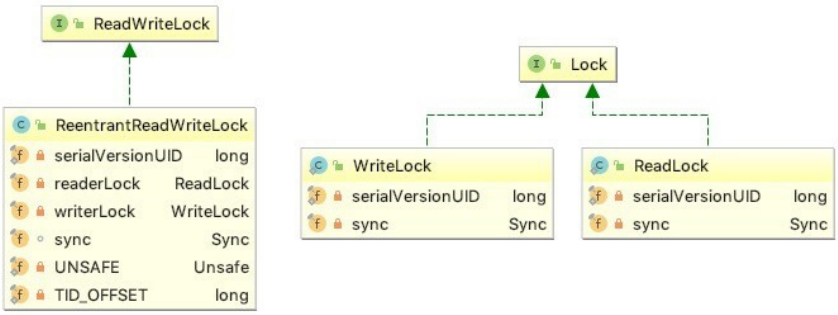
3、ReadWriteLock 使用场景

从上面例子可以看出，如果读操作远远多于写操作，使用 ReadWriteLock 可以大幅提升性能。但如果是一个写入密集型的程序，那么 ReadWriteLock 并不会带来显著性能的提升，因为即使使用 ReadWriteLock，“写/写”及“读/写”依旧是互斥的。并且由于要分开控制读写两种锁，还需要额外的开销。

如果你的并发程序存在性能问题，可以把 ReadWriteLock 作为性能调优的手段，进行尝试。究竟读和写的线程达到什么比例时，使用 ReadWriteLock 性能更好，其实并没有定论。完全和你的程序场景有关系，所以使用 ReadWriteLock 做性能调优时，一定要基于实际的测试数据，而不是一股脑的全部使用 ReadWriteLock。

4、ReadWriteLock 实现

关于 ReadWriteLock 的实现，我们先看下面的类图：



可以看到 ReentrantReadWriteLock 中持有 readerLock 和 writerLock 两把锁，而这两把锁也是 Lock 接口的实现。ReadLock 间由于是非互斥的，所以ReadLock对lock方法的实现如下：

```
public void lock() {
    sync.acquireShared(1);
}
```

而WriteLock对lock方法的实现则是如下：

```
public void lock() {
    sync.acquire(1);
}
```

可以看到 ReadLock 的 lock 方法中调用的是 acquireShared，也就是共享方式获取锁。两者都是通过 sync 来实现，两种锁的 sync 对象都是来自 ReentrantReadWriteLock 的构造函数：

```
public ReentrantReadWriteLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
    readerLock = new ReadLock(this);
    writerLock = new WriteLock(this);
}
```

而 Sync 之前我们已经讲解过，它继承自 AbstractQueuedSynchronizer，也就是 AQS。通过 AQS 提供的模版实现同步原语。而最终的实现方式则是在 AbstractQueuedSynchronizer 的子类，也就是 FairSync 和 NonfairSync 中。AQS 的原理之前已经讲过。结合 AQS 的原理，再加上之前我们对 ReentrantLock 源代码的分析，再来分析 ReentrantReadWriteLock 的源代码，并不困难，大家可以自行继续分析。

5、总结

本节我们又学习了一种比较实用的锁。ReentrantReadWriteLock 允许并发的读，如果你的程序以读取为主，那么使用 ReentrantReadWriteLock 会显著提升你的性能。但如果场景不符合，不但不会提升性能，还会因为锁的复杂度，反而降级性能。

}

22 到底哪把锁更适合你？—synchronized与ReentrantLock对比

24 经典并发容器，多线程面试必备。—深入解析ConcurrentHashMap