一: 读写锁分离模式

这种锁目前在很多场景中都能遇到，尤其是在共享数据的读取并修改的时候，这时候就需要用到读写锁了，另外注 意的是其实jdk中已经有了ReadWriteLock的实现，但是这里想模拟一下这个场景。

1) 首先需要一个接口来定义锁的基本操作，也就是获取锁与释放锁:

```
package readerWriterLock;

public interface Lock {
    //获取显式锁,没有获得锁将被阻塞
    void lock() throws InterruptedException;

    //释放所获取的锁
    void unlock();
}
```

2) 然后还需要定义一个读写锁的接口，这个接口的作用就是定义读写锁的特性

```
package readerWriterLock;

public interface ReadWriteLock {

    //创建reader锁
    Lock readLock();

    //创建write锁
    Lock writeLock();

    //获取当前有多少线程正在执行写操作
    int getWritingWriters();

    //获取当前有多少线程正常等待获取写入锁
    int getWaittingWriters();

    //获取当前有多少线程正在等待获取reader锁
    int getReadingReaders();

    //工厂方法,创建ReadWriterLock
    static ReadWriteLock readWriteLock() {
        return new ReadWriteLockImpl();
    }

    //工厂方法,创建ReadWriteLock,并且传入preferWriter
    static ReadWriteLock readWriteLock(boolean preferWriter) {
        return new ReadWriteLockImpl(preferWriter);
    }
}
```

这里注意，使用的Lock其实是第一个接口Lock的内容，另外这里ReadWriteLockImpl就是对Lock的实现，同时注意这里的工厂方法的写法

3) ReadWriteLockImpl定义常见的读写锁的相关方法，如获取当前多少线程正在写入等.

```
 package readerWriterLock;

 class ReadWriteLockImpl implements  readerWriterLock.ReadWriteLock {

        //用于设置read和write的偏好
        private boolean preferWriter;

        //当前有多少个线程正在写入
        private int writingWriters = 0;

        //当前有多少个线程正在等待写入
        private int waitingWriters = 0;

        //当前有多少个线程正在read
        private int readingReaders = 0;

        //定义对象锁
        private final Object MUTEX = new Object();


        //默认情况下preferWriter为true
        public ReadWriteLockImpl() { this(true); }
```

```
        public ReadWriteLockImpl(boolean preferWriter) { this.preferWriter = preferWriter; }

        @Override
        public readerWriterLock.Lock readLock() {
            return  new ReadLock(this);
        }

        @Override
        public readerWriterLock.Lock writeLock() {
            return  new WriteLock(this);
        }

        //使写线程的数量增加
        void incrementWritingWriters() {  this.writingWriters++; }

        //使等待写入的线程数量增加
        void incrementWaittingWriters() { this.waitingWriters++; }

        //使读线程的数量增加
        void incrementReadingReaders() { this.readingReaders++;  }

        //使写线程的数量减少
        void decrementWritingWriters() { this.writingWriters--;  }

        //使等待写入的线程数量减少
        void decrementWaittingWriters() { this.waitingWriters--; }

        //使读线程的数量增加
        void decrementReadingReaders() { this.readingReaders--;  }

        //获取当前有多少个线程正在写操作
        public int getWritingWriters() { return this.writingWriters;  }

        //获取当前有多少个线程正在等待写入锁
        public int getWaittingWriters() { return this.waitingWriters; }

        //获取当前多少个线程正在进行读操作
        public int getReadingReaders() { return this.readingReaders; }

        //获取对象锁
        Object getMutex() { return this.MUTEX; }

        //获取当前是否偏向写锁
        boolean getPreferWriter() { return this.preferWriter; }

        //设置写锁偏好
        void changePrefer(boolean preferWriter) { this.preferWriter = preferWriter;}

}
```

这里可以看到通过三个参数来标识读锁和写锁的等待情况，然后还有一个偏好设置，这个偏好的作用用于在写锁释放锁之后，就设置为false,让读锁来获得锁。同理读锁释放锁之后，设置为true，让写锁快速获取锁。最终目的就是当前释放锁之后，让另外一种锁更加快速获取到锁。

4) ReadLock 读锁用于实现读锁的获取锁和释放锁的内容，实现Lock

```
package readerWriterLock;

public class ReadLock implements Lock {

    private final ReadWriteLockImpl readWriteLock;

    ReadLock(ReadWriteLockImpl readWriteLock){ this.readWriteLock = readWriteLock; }

    @Override
    public void lock() throws InterruptedException {
        //使用MUTEX作用锁
        synchronized(readWriteLock.getMutex()) {
            //若此时有线程正在进行写操作,或者有写线程在等待并且偏向写锁的标识为true时，就会无法获得读锁，只能被挂起
            while(readWriteLock.getWritingWriters() >0
                 || (readWriteLock.getPreferWriter()
                    && readWriteLock.getWaittingWriters() >0 )) {
                readWriteLock.getMutex().wait();
            }

            //成功获得锁,就会使得readingReaders的数量增加
            readWriteLock.incrementReadingReaders();
        }

    }
```

```
    @Override
    public void unlock() {
        synchronized(readWriteLock.getMutex()) {
            /**
             * 释放锁的过程就是使得当前的reading数量减少一
             * 将preferWriter设置为true,可以使得writer线程获得更多的机会
             * 通知唤醒与MUTEX关联monitor waitset中的线程
             */
            readWriteLock.decrementReadingReaders();
            readWriteLock.changePrefer(true);
            readWriteLock.getMutex().notifyAll();
        }

    }

}
```

这里注意ReadWriteLockImpl作用私有变量来使用的，因为读写锁的基本功能只是获取和释放锁，其他的都是作用变量方法来使用。另外这里注意一下，获取读锁的时候，考虑好当前是否有写锁正在操作，或者偏好设置为true的，同时有写锁正在等待的情况，如果是则进行挂起，否则就能获取到锁了。释放锁的内容就相对简单一点。

5) WriteLock锁，实现Lock,定义好写锁的获取锁和释放锁的内容

```
package readerWriterLock;

public class WriteLock implements Lock {

    private final ReadWriteLockImpl readWriteLock;

    WriteLock(ReadWriteLockImpl readWriteLock){ this.readWriteLock = readWriteLock; }

    @Override
    public void lock() throws InterruptedException {
        synchronized(readWriteLock.getMutex()) {
            try {
                //使获取写入锁的数字加1
                readWriteLock.incrementWaittingWriters();
                //如果此时有其他线程正在进行读操作，或者写操作，那么当前线程将被挂起
                while(readWriteLock.getReadingReaders() >0
                     || readWriteLock.getWritingWriters() > 0) {
                    readWriteLock.getMutex().wait();
                }
            }finally {
                //成功获取到写入锁,使得等待获取写入锁的计数器减一
                this.readWriteLock.decrementWaittingWriters();
            }
            //将正在写入的线程数量加一
            readWriteLock.incrementWritingWriters();
        }

    }

    @Override
    public void unlock() {
        synchronized(readWriteLock.getMutex()) {
            //减少正在写入锁的线程计数器
            readWriteLock.decrementWritingWriters();
            //将偏好状态修改为false,可以使得读锁被快速获取
            readWriteLock.changePrefer(false);
            //通知唤醒与MUTEX关联monitor waitset中的线程
            readWriteLock.getMutex().notifyAll();
        }

    }

}
```

这里写锁的获取情况和读锁其实有点类似的，另外注意就是写锁的时候也要对相应的参数进行设置调整。

6) 读写锁的使用，前面定义完了读锁和写锁的具体获取锁与释放锁的代码，那么这里如何使用读写锁就需要代码来实现了。

```
package readerWriterLock;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import java.util.concurrent.TimeUnit;

public class ShareData {
```

```
        //定义共享数据(资源)
        private final List<Character> container = new ArrayList<>();

        //构造ReadWriteLock
        private final ReadWriteLock readWriteLock = ReadWriteLock.readWriteLock();

        //创建读取锁
        private final Lock readLock =  readWriteLock.readLock();

        //创建写入锁
        private final Lock writeLock = readWriteLock.writeLock();

        private final int length;

        public ShareData(int length) {
            this.length = length;
            for(int i =0;i< length;i++) {
                container.add(i,'c');
            }
        }

        public char[] read() throws InterruptedException{
            char[] newBuffer = new char[length];
            try {
                //首先使用读锁进行lock
                readLock.lock();
                for(int i =0 ;i< length;i++) {
                    newBuffer[i] = container.get(i);
                }
                slowly();
                //注意不要在try finally里面中的try使用return
            }finally {
                //当前所有的操作都完成之后，对读锁进行释放
                readLock.unlock();
            }
            return newBuffer;
        }

        public void write(char c) throws InterruptedException{
            try {
                //使用写锁进行lock
                writeLock.lock();
                for(int i =0;i< length;i++) {
                    this.container.add(i,c);
                }
                slowly();
            }finally {
                writeLock.unlock();
            }
        }

        //模拟操作耗时
        private void slowly() {
            try {
                Random random = new Random();
                TimeUnit.SECONDS.sleep(random.nextInt(10));
            }catch(InterruptedException e ){
                e.printStackTrace();
            }
        }

}
```

这里注意其实还是readWriteLock的方法来获取读写锁的，另外这里使用ReadWriteLock.readWriteLock()工厂方法来获取到ReadWriteLockImpl,另外这里构造方法如果有需要可以自定改造的。

7) 最后就是一个测试方法了，分别开始若干个写线程和读线程进行操作

```
  package readerWriterLock;

public class ReadWriteLockTest {

    private final static String text = "ThisIsTheExampleForReadWriteLock";

    public static void main(String[] args) {
        //定义共享数据
        final ShareData shareData = new ShareData(50);

        //创建两个线程进行写操作
```

```
        for(int i =0;i <2;i++) {
            new Thread( () ->
            {
                for(int index =0; index < text.length();index++) {
                    try {
                        char c = text.charAt(index);
                        shareData.write(c);
                        System.out.println(Thread.currentThread().getName()+ "  write  "+c);
                    }catch(InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            },"Writers_"+i).start();
        }

        //创建10个线程进行读取数据操作
        for(int i =0;i <10;i++) {
            new Thread( () ->
            {
                while(true) {
                    try {
                        char[] charStr = shareData.read();
                        System.out.println(Thread.currentThread().getName() +" read "+new String(charStr));
                    }catch(InterruptedException e) {
                        //e.printStackTrace();
                    }
                }
            },"Readers_"+i).start();;
        }

    }

}
```

这里测试的结果如下所示:

```
    Writers_0  write  T
    Readers_5 read TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
    Readers_6 read TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
    Readers_4 read TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
    Readers_8 read TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
    Readers_9 read TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
    Readers_7 read TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
    Writers_1  write  T
    Readers_0 read TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
    ....
```

那么目前来说，因为jdk版本的升级会带来一些优化，因此如果是读操作比较多的情况下，在jdk1.8下建议使用StampedLock进行操作，该锁提供了一种乐观的机制，性能相对来说比较好。但是如果是写操作比较多的情况下，那么synchronized可能会性能相对好一点，这个后面有机会进行测试一下。