Java 并发开发：Lock 框架详解

摘要：

我们已经知道，synchronized 是Java的关键字，是Java的内置特性，在JVM层面实现了对临界资源的同步互斥访问，但 synchronized 粒度有些大，在处理实际问题时存在诸多局限性，比如响应中断等。Lock 提供了比 synchronized更广泛的锁操作，它能以更优雅的方式处理线程同步问题。本文以synchronized与Lock的对比为切入点，对Java中的Lock框架的枝干部分进行了详细介绍，最后给出了锁的一些相关概念。

一. synchronized 的局限性 与 Lock 的优点

如果一个代码块被synchronized关键字修饰，当一个线程获取了对应的锁，并执行该代码块时，其他线程便只能一直等待直至占有锁的线程释放锁。事实上，占有锁的线程释放锁一般会是以下三种情况之一：

- 占有锁的线程执行完了该代码块，然后释放对锁的占有；
- 占有锁线程执行发生异常，此时JVM会让线程自动释放锁；
- 占有锁线程进入 WAITING 状态从而释放锁，例如在该线程中调用wait()方法等。

synchronized 是Java语言的内置特性，可以轻松实现对临界资源的同步互斥访问。那么，为什么还会出现Lock呢？试考虑以下三种情况：

Case 1 ：

在使用synchronized关键字的情形下，假如占有锁的线程由于要等待IO或者其他原因（比如调用sleep方法）被阻塞了，但是又没有释放锁，那么其他线程就只能一直等待，别无他法。这会极大影响程序执行效率。因此，就需要有一种机制可以不让等待的线程一直无期限地等待下去（比如只等待一定的时间 (解决方案：tryLock(long time, TimeUnit unit)) 或者 能够响应中断 (解决方案：lockInterruptibly()) ），这种情况可以通过 Lock 解决。

Case 2 ：

我们知道，当多个线程读写文件时，读操作和写操作会发生冲突现象，写操作和写操作也会发生冲突现象，但是读操作和读操作不会发生冲突现象。但是如果采用synchronized关键字实现同步的话，就会导致一个问题，即当多个线程都只是进行读操作时，也只有一个线程在可以进行读操作，其他线程只能等待锁的释放而无法进行读操作。因此，需要一种机制来使得当多个线程都只是进行读操作时，线程之间不会发生冲突。同样地，Lock也可以解决这种情况 (解决方案：ReentrantReadWriteLock) 。

Case 3 ：

我们可以通过Lock得知线程有没有成功获取到锁 (解决方案：ReentrantLock) ，但这个是synchronized无法办到的。

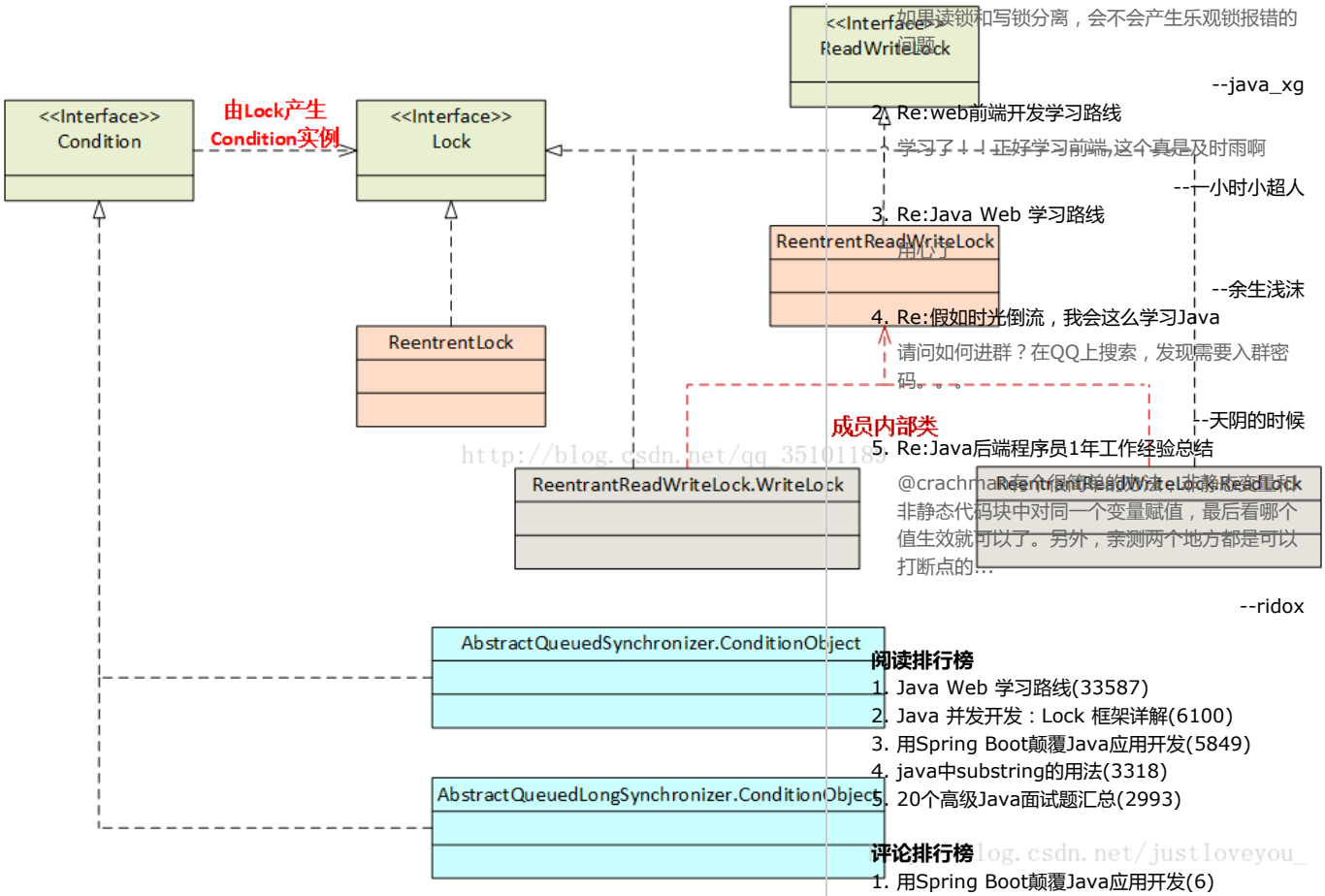
上面提到的三种情形，我们都可以通过Lock来解决，但 synchronized 关键字却无能为力。事实上，Lock 是 java.util.concurrent.locks包 下的接口，Lock 实现提供了比 synchronized 关键字 更广泛的锁操作，它能以更优雅的方式处理线程同步问题。也就是说，Lock提供了比synchronized更多的功能。但是要注意以下几点：

1）synchronized是Java的关键字，因此是Java的内置特性，是基于JVM层面实现的。而Lock是一个Java接口，是基于JDK层面实现的，通过这个接口可以实现同步访问；

2）采用synchronized方式不需要用户去手动释放锁，当synchronized方法或者synchronized代码块执行完之后，系统会自动让线程释放对锁的占用；而 Lock则必须要用户去手动释放锁，如果没有主动释放锁，就有可能导致死锁现象。

二. java.util.concurrent.locks包下常用的类与接口

以下是 java.util.concurrent.locks包下主要常用的类与接口的关系：



1、Lock

通过查看Lock的源码可知，Lock 是一个接口：

```
public interface Lock {
    void lock();
    void lockInterruptibly() throws InterruptedException;  // 可以响应中断
    boolean tryLock();
    boolean tryLock(long time, TimeUnit unit) throws InterruptedException;  //
可以响应中断
    void unlock();
    Condition newCondition();
}
```

下面来逐个分析Lock接口中每个方法。lock()、tryLock()、tryLock(long time, TimeUnit unit) 和 lockInterruptibly()都是用来获取锁的。unLock()方法是用来释放锁的。newCondition() 返回 绑定到此 Lock 的新的 Condition 实例 ，用于线程间的协作，详细内容见文章《Java 并发：线程间通信与协作》。

1). lock()

在Lock中声明了四个方法来获取锁，那么这四个方法有何区别呢？首先，lock()方法是平常使用得最多的一个方法，就是用来获取锁。如果锁已被其他线程获取，则进行等待。在前面已经讲到，如果采用Lock，必须主动去释放锁，并且在发生异常时，不会自动释放锁。因此，一般来说，使用Lock必须在try...catch...块中进行，并且将释放锁的操作放在finally块中进行，以保证锁一定被被释放，防止死锁的发生。通常使用Lock来进行同步的话，是以下面这种形式去使用的：

```
Lock lock = ...;
lock.lock();
try{
    //处理任务
}catch(Exception ex){

}finally{
```

```
    lock.unlock();   //释放锁
}
```

2). tryLock() & tryLock(long time, TimeUnit unit)

tryLock()方法是有返回值的，它表示用来尝试获取锁，如果获取成功，则返回true；如果获取失败（即锁已被其他线程获取），则返回false，也就是说，这个方法无论如何都会立即返回（在拿不到锁时不会一直在那等待）。

tryLock(long time, TimeUnit unit)方法和tryLock()方法是类似的，只不过区别在于这个方法在拿不到锁时会等待一定的时间，在时间期限之内如果还拿不到锁，就返回false，同时可以响应中断。如果一开始拿到锁或者在等待期间内拿到了锁，则返回true。

一般情况下，通过tryLock来获取锁时是这样使用的：

```
Lock lock = ...;
if(lock.tryLock()) {
     try{
         //处理任务
     }catch(Exception ex){

     }finally{
         lock.unlock();   //释放锁
     }
}else {
    //如果不能获取锁，则直接做其他事情
}
```

3). lockInterruptibly()

lockInterruptibly()方法比较特殊，当通过这个方法去获取锁时，如果线程 正在等待获取锁，则这个线程能够 响应中断，即中断线程的等待状态。例如，当两个线程同时通过lock.lockInterruptibly()想获取某个锁时，假若此时线程A获取到了锁，而线程B只有在等待，那么对线程B调用threadB.interrupt()方法能够中断线程B的等待过程。

由于lockInterruptibly()的声明中抛出了异常，所以lock.lockInterruptibly()必须放在try块中或者在调用lockInterruptibly()的方法外声明抛出 InterruptedException，但推荐使用后者，原因稍后阐述。因此，lockInterruptibly()一般的使用形式如下：

```
public void method() throws InterruptedException {
    lock.lockInterruptibly();
    try {
     //.....
    }
    finally {
        lock.unlock();
    }
}
```

注意，当一个线程获取了锁之后，是不会被interrupt()方法中断的。因为interrupt()方法只能中断阻塞过程中的线程而不能中断正在运行过程中的线程。因此，当通过lockInterruptibly()方法获取某个锁时，如果不能获取到，那么只有进行等待的情况下，才可以响应中断的。与 synchronized 相比，当一个线程处于等待某个锁的状态，是无法被中断的，只有一直等待下去。

2、ReentrantLock

ReentrantLock，即 可重入锁。ReentrantLock是唯一实现了Lock接口的类，并且ReentrantLock提供了更多的方法。下面通过一些实例学习如何使用 ReentrantLock。

例 1 ： Lock 的正确使用

```
public class Test {
    private ArrayList<Integer> arrayList = new ArrayList<Integer>();

    public static void main(String[] args) {
        final Test test = new Test();
```

```
        new Thread("A") {
            public void run() {
                test.insert(Thread.currentThread());
            };
        }.start();

        new Thread("B") {
            public void run() {
                test.insert(Thread.currentThread());
            };
        }.start();
    }

    public void insert(Thread thread) {
        Lock lock = new ReentrantLock();  // 注意这个地方:lock被声明为局部变量
        lock.lock();
        try {
            System.out.println("线程" + thread.getName() + "得到了锁...");
            for (int i = 0; i < 5; i++) {
                arrayList.add(i);
            }
        } catch (Exception e) {

        } finally {
            System.out.println("线程" + thread.getName() + "释放了锁...");
            lock.unlock();
        }
    }
}/* Output:
        线程A得到了锁...
        线程B得到了锁...
        线程A释放了锁...
        线程B释放了锁...
 *///:~
```

结果或许让人觉得诧异。第二个线程怎么会在第一个线程释放锁之前得到了锁？原因在于，在insert方法中的lock变量是局部变量，每个线程执行该方法时都会保存一个副本，那么每个线程执行到lock.lock()处获取的是不同的锁，所以就不会对临界资源形成同步互斥访问。因此，我们只需要将lock声明为成员变量即可，如下所示。

```
public class Test {
    private ArrayList<Integer> arrayList = new ArrayList<Integer>();
    private Lock lock = new ReentrantLock();  // 注意这个地方:lock被声明为成员变
量
    ...
}/* Output:
        线程A得到了锁...
        线程A释放了锁...
        线程B得到了锁...
        线程B释放了锁...
 *///:~
```

例 2 ： tryLock() & tryLock(long time, TimeUnit unit)

```
public class Test {
    private ArrayList<Integer> arrayList = new ArrayList<Integer>();
    private Lock lock = new ReentrantLock(); // 注意这个地方：lock 被声明为成员
变量

    public static void main(String[] args) {
        final Test test = new Test();
```

```
        new Thread("A") {
            public void run() {
                test.insert(Thread.currentThread());
            };
        }.start();

        new Thread("B") {
            public void run() {
                test.insert(Thread.currentThread());
            };
        }.start();
    }

    public void insert(Thread thread) {
        if (lock.tryLock()) {     // 使用 tryLock()
            try {
                System.out.println("线程" + thread.getName() + "得到了锁...");
                for (int i = 0; i < 5; i++) {
                    arrayList.add(i);
                }
            } catch (Exception e) {

            } finally {
                System.out.println("线程" + thread.getName() + "释放了锁...");
                lock.unlock();
            }
        } else {
            System.out.println("线程" + thread.getName() + "获取锁失败...");
        }
    }
}/* Output:
        线程A得到了锁...
        线程B获取锁失败...
        线程A释放了锁...
 *///:~
```

与 tryLock() 不同的是，tryLock(long time, TimeUnit unit) 能够响应中断，即支持对获取锁的中断，但尝试获取一个内部锁的操作（进入一个 synchronized 块）是不能被中断的。如下所示：

```
public class Test {
    private Lock lock = new ReentrantLock();
    public static void main(String[] args)  {
        Test test = new Test();
        MyThread thread1 = new MyThread(test,"A");
        MyThread thread2 = new MyThread(test,"B");
        thread1.start();
        thread2.start();

        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        thread2.interrupt();
    }

    public void insert(Thread thread) throws InterruptedException{
        if(lock.tryLock(4, TimeUnit.SECONDS)){
            try {
                System.out.println("time=" + System.currentTimeMillis() + " ,线
```

```
程 " + thread.getName()+"得到了锁...");
                long now = System.currentTimeMillis();
                while (System.currentTimeMillis() - now < 5000) {
                    // 为了避免Thread.sleep()而需要捕获InterruptedException而带
来的理解上的困惑,
                    // 此处用这种方法空转3秒
                }
            }finally{
                lock.unlock();
            }
        }else {
            System.out.println("线程 " + thread.getName()+"放弃了对锁的获
取...");
        }
    }
}

class MyThread extends Thread {
    private Test test = null;

    public MyThread(Test test,String name) {
        super(name);
        this.test = test;
    }

    @Override
    public void run() {
        try {
            test.insert(Thread.currentThread());
        } catch (InterruptedException e) {
            System.out.println("time=" + System.currentTimeMillis() + " ,线程 "
+ Thread.currentThread().getName() + "被中断...");
        }
    }
}/* Output:
        time=1486693682559, 线程A 得到了锁...
        time=1486693684560, 线程B 被中断...(响应中断，时间恰好间隔2s)
 *///:~
```

例 3 ： 使用 lockInterruptibly() 响应中断

```
public class Test {
    private Lock lock = new ReentrantLock();
    public static void main(String[] args)  {
        Test test = new Test();
        MyThread thread1 = new MyThread(test,"A");
        MyThread thread2 = new MyThread(test,"B");
        thread1.start();
        thread2.start();

        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        thread2.interrupt();
    }

    public void insert(Thread thread) throws InterruptedException{
        //注意，如果需要正确中断等待锁的线程，必须将获取锁放在外面，然后将
InterruptedException 抛出
```

```
        lock.lockInterruptibly();
        try {
            System.out.println("线程 " + thread.getName()+"得到了锁...");
            long startTime = System.currentTimeMillis();
            for(    ;     ; ) {              // 耗时操作
                if(System.currentTimeMillis() - startTime >= Integer.MAX_VALUE)
                    break;
                //插入数据
            }
        }finally {
            System.out.println(Thread.currentThread().getName()+"执行
finally...");
            lock.unlock();
            System.out.println("线程 " + thread.getName()+"释放了锁");
        }
        System.out.println("over");
    }
}

class MyThread extends Thread {
    private Test test = null;

    public MyThread(Test test,String name) {
        super(name);
        this.test = test;
    }

    @Override
    public void run() {
        try {
            test.insert(Thread.currentThread());
        } catch (InterruptedException e) {
            System.out.println("线程 " + Thread.currentThread().getName() + "被
中断...");
        }
    }
}/* Output:
        线程 A得到了锁...
        线程 B被中断...
 *///:~
```

运行上述代码之后，发现 thread2 能够被正确中断，放弃对任务的执行。特别需要注意的是，如果需要正确中断等待锁的线程，必须将获取锁放在外面（try 语句块外），然后将 InterruptedException 抛出。如果不这样做，像如下代码所示：

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Test {
    private Lock lock = new ReentrantLock();

    public static void main(String[] args) {
        Test test = new Test();
        MyThread thread1 = new MyThread(test, "A");
        MyThread thread2 = new MyThread(test, "B");
        thread1.start();
        thread2.start();

        try {
            Thread.sleep(5000);
            System.out.println("线程" + Thread.currentThread().getName()
```

```
                    + " 睡醒了...");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            thread2.interrupt();
        }

        public void insert(Thread thread) {

            try {
                // 注意，如果将获取锁放在try语句块里，则必定会执行finally语句块中的
解锁操作。若线程在获取锁时被中断，则再执行解锁操作就会导致异常，因为该线程并未获
得到锁。
                lock.lockInterruptibly();
                System.out.println("线程 " + thread.getName() + "得到了锁...");
                long startTime = System.currentTimeMillis();
                for (;;) {
                    if (System.currentTimeMillis() - startTime >= Integer.MAX_VALUE)
// 耗时操作
                        break;
                    // 插入数据
                }
            } catch (Exception e) {

            } finally {
                System.out.println(Thread.currentThread().getName()
                        + "执行finally...");
                lock.unlock();
                System.out.println("线程 " + thread.getName() + "释放了锁...");
            }
        }
    }

    class MyThread extends Thread {
        private Test test = null;

        public MyThread(Test test, String name) {
            super(name);
            this.test = test;
        }

        @Override
        public void run() {

            test.insert(Thread.currentThread());
            System.out.println("线程 " + Thread.currentThread().getName() + "被中
断...");
        }
    }/* Output:
            线程A 得到了锁...
            线程main 睡醒了...
            B执行finally...
            Exception in thread "B"
                java.lang.IllegalMonitorStateException
                at java.util.concurrent.locks.ReentrantLock$Sync.tryRelease(Unknown
Source)
                at
java.util.concurrent.locks.AbstractQueuedSynchronizer.release(Unknown Source)
                at java.util.concurrent.locks.ReentrantLock.unlock(Unknown Source)
                at Test.insert(Test.java:39)
```

```
            at MyThread.run(Test.java:56)
 *///:~
```

注意，上述代码就将锁的获取操作放在try语句块里，则必定会执行finally语句块中的解锁操作。在 准备获取锁的 线程B 被中断后，再执行解锁操作就会抛出IllegalMonitorStateException，因为该线程并未获得到锁却执行了解锁操作。

3、ReadWriteLock

ReadWriteLock也是一个接口，在它里面只定义了两个方法：

```
public interface ReadWriteLock {
    /**
     * Returns the lock used for reading.
     *
     * @return the lock used for reading.
     */
    Lock readLock();

    /**
     * Returns the lock used for writing.
     *
     * @return the lock used for writing.
     */
    Lock writeLock();
}
```

一个用来获取读锁，一个用来获取写锁。也就是说，将对临界资源的读写操作分成两个锁来分配给线程，从而使得多个线程可以同时进行读操作。下面的 ReentrantReadWriteLock 实现了 ReadWriteLock 接口。

4、ReentrantReadWriteLock

ReentrantReadWriteLock 里面提供了很多丰富的方法，不过最主要的有两个方法：readLock()和writeLock()用来获取读锁和写锁。下面通过几个例子来看一下ReentrantReadWriteLock具体用法。假如有多个线程要同时进行读操作的话，先看一下synchronized达到的效果：

```
public class Test {
    public static void main(String[] args)  {
        final Test test = new Test();

        new Thread("A"){
            public void run() {
                test.get(Thread.currentThread());
            };
        }.start();

        new Thread("B"){
            public void run() {
                test.get(Thread.currentThread());
            };
        }.start();

    }

    public synchronized void get(Thread thread) {
        long start = System.currentTimeMillis();
        System.out.println("线程"+ thread.getName()+"开始读操作...");
        while(System.currentTimeMillis() - start <= 1) {
            System.out.println("线程"+ thread.getName()+"正在进行读操作...");
        }
        System.out.println("线程"+ thread.getName()+"读操作完毕...");
    }
```

```
}/* Output:
        线程A开始读操作...
        线程A正在进行读操作...
        ...
        线程A正在进行读操作...
        线程A读操作完毕...
        线程B开始读操作...
        线程B正在进行读操作...
        ...
        线程B正在进行读操作...
        线程B读操作完毕...
 *///:~
```

这段程序的输出结果会是，直到线程A执行完读操作之后，才会打印线程B执行读操作的信息。而改成使用读写锁的话：

```
public class Test {
    private ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();

    public static void main(String[] args) {
        final Test test = new Test();

        new Thread("A") {
            public void run() {
                test.get(Thread.currentThread());
            };
        }.start();

        new Thread("B") {
            public void run() {
                test.get(Thread.currentThread());
            };
        }.start();
    }

    public void get(Thread thread) {
        rwl.readLock().lock(); // 在外面获取锁
        try {
            long start = System.currentTimeMillis();
            System.out.println("线程" + thread.getName() + "开始读操作...");
            while (System.currentTimeMillis() - start <= 1) {
                System.out.println("线程" + thread.getName() + "正在进行读操
作...");
            }
            System.out.println("线程" + thread.getName() + "读操作完毕...");
        } finally {
            rwl.readLock().unlock();
        }
    }
}/* Output:
        线程A开始读操作...
        线程B开始读操作...
        线程A正在进行读操作...
        线程A正在进行读操作...
        线程B正在进行读操作...
        ...
        线程A读操作完毕...
        线程B读操作完毕...
 *///:~
```

我们可以看到，线程A和线程B在同时进行读操作，这样就大大提升了读操作的效率。不过要注意的是，如果有一个线程已经占用了读锁，则此时其他线程如果要申请写锁，则申请写锁的线程会一直等待释放读锁。如果有一个线程已经占用了写锁，则此时其他线程如果申请写锁或者读锁，则申请的线程也会一直等待释放写锁。

5、Lock和synchronized的选择

总的来说，Lock和synchronized有以下几点不同：

- (1) Lock是一个接口，是JDK层面的实现；而synchronized是Java中的关键字，是Java的内置特性，是JVM层面的实现；
- (2) synchronized 在发生异常时，会自动释放线程占有的锁，因此不会导致死锁现象发生；而Lock在发生异常时，如果没有主动通过unLock()去释放锁，则很可能造成死锁现象，因此使用Lock时需要在finally块中释放锁；
- (3) Lock 可以让等待锁的线程响应中断，而使用synchronized时，等待的线程会一直等待下去，不能够响应中断；
- (4) 通过Lock可以知道有没有成功获取锁，而synchronized却无法办到；
- (5) Lock可以提高多个线程进行读操作的效率。

在性能上来说，如果竞争资源不激烈，两者的性能是差不多的。而当竞争资源非常激烈时（即有大量线程同时竞争），此时Lock的性能要远远优于synchronized。所以说，在具体使用时要根据适当情况选择。

三. 锁的相关概念介绍

1、可重入锁

如果锁具备可重入性，则称作为 可重入锁 。像 synchronized和ReentrantLock都是可重入锁，可重入性在我看来实际上表明了 锁的分配机制：基于线程的分配，而不是基于方法调用的分配。举个简单的例子，当一个线程执行到某个synchronized方法时，比如说method1，而在method1中会调用另外一个synchronized方法method2，此时线程不必重新去申请锁，而是可以直接执行方法method2。

```
class MyClass {
    public synchronized void method1() {
        method2();
    }

    public synchronized void method2() {

    }
}
```

上述代码中的两个方法method1和method2都用synchronized修饰了。假如某一时刻，线程A执行到了method1，此时线程A获取了这个对象的锁，而由于method2也是synchronized方法，假如synchronized不具备可重入性，此时线程A需要重新申请锁。但是，这就会造成死锁，因为线程A已经持有了该对象的锁，而又在申请获取该对象的锁，这样就会线程A一直等待永远不会获取到的锁。而由于synchronized和Lock都具备可重入性，所以不会发生上述现象。

2、可中断锁

顾名思义，可中断锁就是可以响应中断的锁。在Java中，synchronized就不是可中断锁，而Lock是可中断锁。
如果某一线程A正在执行锁中的代码，另一线程B正在等待获取该锁，可能由于等待时间过长，线程B不想等待了，想先处理其他事情，我们可以让它中断自己或者在别的线程中中断它，这种就是可中断锁。在前面演示tryLock(long time, TimeUnit unit)和lockInterruptibly()的用法时已经体现了Lock的可中断性。

3、公平锁

公平锁即 尽量 以请求锁的顺序来获取锁。比如，同是有多个线程在等待一个锁，当这个锁被释放时，等待时间最久的线程（最先请求的线程）会获得该所，这种就是公平锁。而非公平锁则无法保证锁的获取是按照请求锁的顺序进行的，这样就可能导致某个或者一些线程永远获取不到锁。

在Java中，synchronized就是非公平锁，它无法保证等待的线程获取锁的顺序。而对于ReentrantLock 和 ReentrantReadWriteLock，它默认情况下是非公平锁，但是可以设置为公平锁。

看下面两个例子：

Case：公平锁

```
public class RunFair {
    public static void main(String[] args) throws InterruptedException {
        final Service service = new Service(true);     // 公平锁，设为 true
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                System.out.println("★线程" + Thread.currentThread().getName()
                        + "运行了");
                service.serviceMethod();
            }
        };

        Thread[] threadArray = new Thread[10];
        for (int i = 0; i < 10; i++)
            threadArray[i] = new Thread(runnable);

        for (int i = 0; i < 10; i++)
            threadArray[i].start();
    }
}
class Service {
    private ReentrantLock lock;
    public Service(boolean isFair) {
        super();
        lock = new ReentrantLock(isFair);
    }
    public void serviceMethod() {
        try {
            lock.lock();
            System.out.println("ThreadName=" + Thread.currentThread().getName()
                    + "获得锁定");
        } finally {
            lock.unlock();
        }
    }
}/* Output:
        ★线程Thread-0运行了
        ★线程Thread-1运行了
        ThreadName=Thread-1获得锁定
        ThreadName=Thread-0获得锁定
        ★线程Thread-2运行了
        ThreadName=Thread-2获得锁定
        ★线程Thread-3运行了
        ★线程Thread-4运行了
        ThreadName=Thread-4获得锁定
        ★线程Thread-5运行了
        ThreadName=Thread-5获得锁定
        ThreadName=Thread-3获得锁定
        ★线程Thread-6运行了
        ★线程Thread-7运行了
        ThreadName=Thread-6获得锁定
        ★线程Thread-8运行了
        ★线程Thread-9运行了
```

```
        ThreadName=Thread-7获得锁定
        ThreadName=Thread-8获得锁定
        ThreadName=Thread-9获得锁定
*///:~
```

Case: 非公平锁

```
public class RunFair {
    public static void main(String[] args) throws InterruptedException {
        final Service service = new Service(false);  // 非公平锁，设为 false
        ...
}/* Output:
        ★线程Thread-0运行了
        ThreadName=Thread-0获得锁定
        ★线程Thread-2运行了
        ThreadName=Thread-2获得锁定
        ★线程Thread-6运行了
        ★线程Thread-1运行了
        ThreadName=Thread-6获得锁定
        ★线程Thread-3运行了
        ThreadName=Thread-3获得锁定
        ★线程Thread-7运行了
        ThreadName=Thread-7获得锁定
        ★线程Thread-4运行了
        ThreadName=Thread-4获得锁定
        ★线程Thread-5运行了
        ThreadName=Thread-5获得锁定
        ★线程Thread-8运行了
        ThreadName=Thread-8获得锁定
        ★线程Thread-9运行了
        ThreadName=Thread-9获得锁定
        ThreadName=Thread-1获得锁定
*///:~
```

根据上面代码演示结果我们可以看出（线程数越多越明显），在公平锁案例下，多个线程在等待一个锁时，一般而言，等待时间最久的线程（最先请求的线程）会获得该锁。而在非公平锁例下，则无法保证锁的获取是按照请求锁的顺序进行的。

另外， 在ReentrantLock类中定义了很多方法，举几个例子：

- isFair() //判断锁是否是公平锁
- isLocked() //判断锁是否被任何线程获取了
- isHeldByCurrentThread() //判断锁是否被当前线程获取了
- hasQueuedThreads() //判断是否有线程在等待该锁
- getHoldCount() //查询当前线程占有lock锁的次数
- getQueueLength() // 获取正在等待此锁的线程数
- getWaitQueueLength(Condition condition) // 获取正在等待此锁相关条件condition的线程数在ReentrantReadWriteLock中也有类似的方法，同样也可以设置为公平锁和非公平锁。不过要记住，ReentrantReadWriteLock并未实现Lock接口，它实现的是ReadWriteLock接口。

4.读写锁

读写锁将对临界资源的访问分成了两个锁，一个读锁和一个写锁。正因为有了读写锁，才使得多个线程之间的读操作不会发生冲突。ReadWriteLock就是读写锁，它是一个接口，ReentrantReadWriteLock实现了这个接口。可以通过readLock()获取读锁，通过writeLock()获取写锁。上一节已经演示过了读写锁的使用方法，在此不再赘述。

标签: Java, 并发, 框架

工程师-搁浅
关注 - 1
粉丝 - 439
+加关注

0 0

« 上一篇：浅谈java异常[Exception]
» 下一篇：写给想成为前端工程师的同学们

posted @ 2017-03-15 17:19 工程师-搁浅 阅读(6101) 评论(2) 编辑 收藏

评论列表

#1楼 2017-07-09 18:24 姜泮昌
ReentrantLock的例1里面，如果将Lock作为成员变量的时候，就不是可重入的了吧，输出应该是A得到A释放、B得到B释放吧

支持(0) 反对(0)

#2楼 2018-01-29 18:21 java_xg
如果读锁和写锁分离，会不会产生乐观锁报错的问题

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。