<https://blog.csdn.net/u010842515/article/details/67634813>

参考博文：[http://www.cnblogs.com/handsomeye/p/5999362.html](http://www.cnblogs.com/handsomeye/p/5999362.html" \t "_blank)

一.Java多线程可以通过：

1. synchronized关键字

2. Java.util.concurrent包中的lock接口和ReentrantLock实现类

这两种方式实现加锁。

二.synchronized关键字加锁的缺陷：

如果一个代码块被synchronized修饰了，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，等待获取锁的线程释放锁，而这里获取锁的线程释放锁只会有两种情况：

　　1）获取锁的线程执行完了该代码块，然后线程释放对锁的占有；

　　2）线程执行发生异常，此时JVM会让线程自动释放锁。

　　那么如果这个获取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，其他线程便只能干巴巴地等待，试想一下，这多么影响程序执行效率。

　　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或者能够响应中断），通过Lock就可以办到。

　　再举个例子：当有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。

　　但是采用synchronized关键字来实现同步的话，就会导致一个问题：

　　如果多个线程都只是进行读操作，所以当一个线程在进行读操作时，其他线程只能等待无法进行读操作。

　　因此就需要一种机制来使得多个线程都只是进行读操作时，线程之间不会发生冲突，通过Lock就可以办到。

　　另外，通过Lock可以知道线程有没有成功获取到锁。这个是synchronized无法办到的。

**对比：**

   　1）Lock不是Java语言内置的，synchronized是Java语言的关键字，因此是内置特性。Lock是一个类，通过这个类可以实现同步访问；

　　2）Lock和synchronized有一点非常大的不同，采用synchronized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，就有可能导致出现死锁现象。

三.Lock和synchronized的选择：

　　总结来说，Lock和synchronized有以下几点不同：

　　1）Lock是一个接口，而synchronized是Java中的关键字，synchronized是内置的语言实现；

　　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synchronized却不行，使用synchronized时，等待的线程会一直等待下去，不能够响应中断；**（I/O和Synchronized都能相应中断，即不需要处理interruptionException异常）**

　　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。

**四. 锁的种类介绍：**

在前面介绍了Lock的基本使用，这一节来介绍一下与锁相关的几个概念。

　　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。

　　看下面这段代码就明白了：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | 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。但是这就会造成一个问题，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

　　可中断锁：顾名思义，就是可以相应中断的锁。

　　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

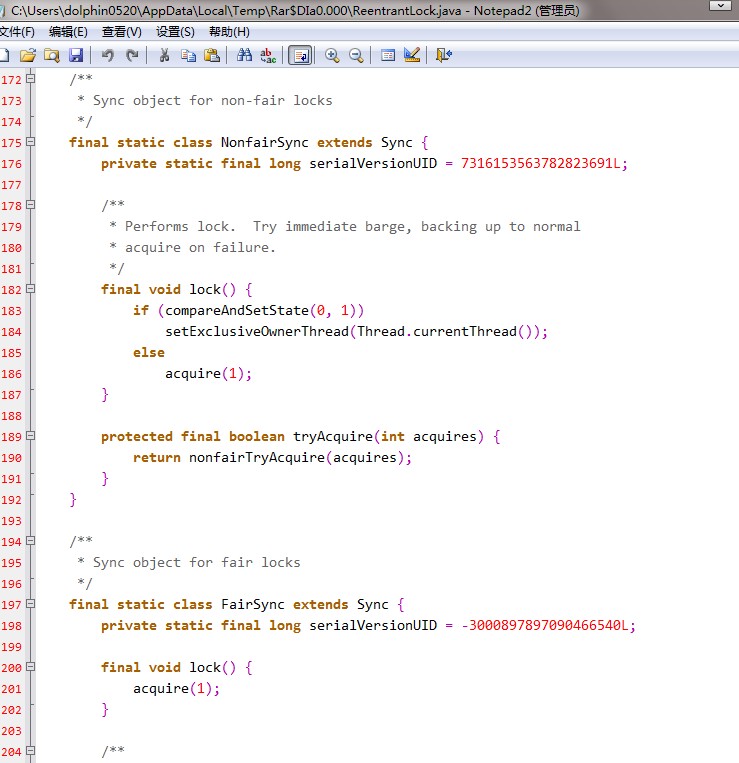
　　公平锁即尽量以请求锁的顺序来获取锁。比如同是有多个线程在等待一个锁，当这个锁被释放时，等待时间最久的线程（最先请求的线程）会获得该所，这种就是公平锁。

　　非公平锁即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。

　　看一下这2个类的源代码就清楚了：



　　在ReentrantLock中定义了2个静态内部类，一个是NotFairSync，一个是FairSync，分别用来实现非公平锁和公平锁。

　　我们可以在创建ReentrantLock对象时，通过以下方式来设置锁的公平性：

|  |  |
| --- | --- |
| 1 | ReentrantLock lock = new ReentrantLock(true); |

 　　如果参数为true表示为公平锁，为fasle为非公平锁。默认情况下，如果使用无参构造器，则是非公平锁。



　　另外在ReentrantLock类中定义了很多方法，比如：

　　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

　　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

　　读写锁将对一个资源（比如文件）的访问分成了2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。

ReadWriteLock

　　ReadWriteLock也是一个接口，在它里面只定义了两个方法：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | 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。也就是说将文件的读写操作分开，分成2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：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | public class Test {      private ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();        public static void main(String[] args)  {          final Test test = new Test();            new Thread(){              public void run() {                  test.get(Thread.currentThread());              };          }.start();            new Thread(){              public void run() {                  test.get(Thread.currentThread());              };          }.start();        }        public synchronized void get(Thread thread) {          long start = System.currentTimeMillis();          while(System.currentTimeMillis() - start <= 1) {              System.out.println(thread.getName()+"正在进行读操作");          }          System.out.println(thread.getName()+"读操作完毕");      }  } |

 　　这段程序的输出结果会是，直到thread1执行完读操作之后，才会打印thread2执行读操作的信息。

　　而改成用读写锁的话：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | public class Test {      private ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();        public static void main(String[] args)  {          final Test test = new Test();            new Thread(){              public void run() {                  test.get(Thread.currentThread());              };          }.start();            new Thread(){              public void run() {                  test.get(Thread.currentThread());              };          }.start();        }        public void get(Thread thread) {          rwl.readLock().lock();          try {              long start = System.currentTimeMillis();                while(System.currentTimeMillis() - start <= 1) {                  System.out.println(thread.getName()+"正在进行读操作");              }              System.out.println(thread.getName()+"读操作完毕");          } finally {              rwl.readLock().unlock();          }      }  } |

　　thread1和thread2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，则申请的线程会一直等待释放写锁。