  接近一周没更新《Java线程》专栏了，主要是这周工作上比较忙，生活上也比较忙，呵呵，进入正题，上一篇讲述了并发包下的Lock，Lock可以更好的解决线程同步问题，使之更面向对象，并且ReadWriteLock在处理同步时更强大，那么同样，线程间仅仅互斥是不够的，还需要通信，本篇的内容是基于上篇之上，使用Lock如何处理线程通信。

        那么引入本篇的主角，Condition，Condition 将 Object 监视器方法（wait、notify 和 notifyAll）分解成截然不同的对象，以便通过将这些对象与任意 Lock 实现组合使用，为每个对象提供多个等待 set （wait-set）。其中，Lock 替代了 synchronized 方法和语句的使用，Condition 替代了 Object 监视器方法的使用。下面将之前写过的一个线程通信的例子替换成用Condition实现(Java线程(三))，代码如下：

**[java]** [view plain](http://blog.csdn.net/ghsau/article/details/7481142) [copy](http://blog.csdn.net/ghsau/article/details/7481142)

[print?](http://blog.csdn.net/ghsau/article/details/7481142)

1. **public** **class** ThreadTest2 {
2. **public** **static** **void** main(String[] args) {
3. **final** Business business = **new** Business();
4. **new** Thread(**new** Runnable() {
5. @Override
6. **public** **void** run() {
7. threadExecute(business, "sub");
8. }
9. }).start();
10. threadExecute(business, "main");
11. }
12. **public** **static** **void** threadExecute(Business business, String threadType) {
13. **for**(**int** i = 0; i < 100; i++) {
14. **try** {
15. **if**("main".equals(threadType)) {
16. business.main(i);
17. } **else** {
18. business.sub(i);
19. }
20. } **catch** (InterruptedException e) {
21. e.printStackTrace();
22. }
23. }
24. }
25. }
26. **class** Business {
27. **private** **boolean** bool = **true**;
28. **private** Lock lock = **new** ReentrantLock();
29. **private** Condition condition = lock.newCondition();
30. **public** /\*synchronized\*/ **void** main(**int** loop) **throws** InterruptedException {
31. lock.lock();
32. **try** {
33. **while**(bool) {
34. condition.await();//this.wait();
35. }
36. **for**(**int** i = 0; i < 100; i++) {
37. System.out.println("main thread seq of " + i + ", loop of " + loop);
38. }
39. bool = **true**;
40. condition.signal();//this.notify();
41. } **finally** {
42. lock.unlock();
43. }
44. }
45. **public** /\*synchronized\*/ **void** sub(**int** loop) **throws** InterruptedException {
46. lock.lock();
47. **try** {
48. **while**(!bool) {
49. condition.await();//this.wait();
50. }
51. **for**(**int** i = 0; i < 10; i++) {
52. System.out.println("sub thread seq of " + i + ", loop of " + loop);
53. }
54. bool = **false**;
55. condition.signal();//this.notify();
56. } **finally** {
57. lock.unlock();
58. }
59. }
60. }

        在Condition中，用await()替换wait()，用signal()替换notify()，用signalAll()替换notifyAll()，传统线程的通信方式，Condition都可以实现，这里注意，Condition是被绑定到Lock上的，要创建一个Lock的Condition必须用newCondition()方法。

        这样看来，Condition和传统的线程通信没什么区别，Condition的强大之处在于它可以为多个线程间建立不同的Condition，下面引入API中的一段代码，加以说明。

**[java]** [view plain](http://blog.csdn.net/ghsau/article/details/7481142) [copy](http://blog.csdn.net/ghsau/article/details/7481142)

[print?](http://blog.csdn.net/ghsau/article/details/7481142)

1. **class** BoundedBuffer {
2. **final** Lock lock = **new** ReentrantLock();//锁对象
3. **final** Condition notFull  = lock.newCondition();//写线程条件
4. **final** Condition notEmpty = lock.newCondition();//读线程条件
6. **final** Object[] items = **new** Object[100];//缓存队列
7. **int** putptr/\*写索引\*/, takeptr/\*读索引\*/, count/\*队列中存在的数据个数\*/;
9. **public** **void** put(Object x) **throws** InterruptedException {
10. lock.lock();
11. **try** {
12. **while** (count == items.length)//如果队列满了
13. notFull.await();//阻塞写线程
14. items[putptr] = x;//赋值
15. **if** (++putptr == items.length) putptr = 0;//如果写索引写到队列的最后一个位置了，那么置为0
16. ++count;//个数++
17. notEmpty.signal();//唤醒读线程
18. } **finally** {
19. lock.unlock();
20. }
21. }
23. **public** Object take() **throws** InterruptedException {
24. lock.lock();
25. **try** {
26. **while** (count == 0)//如果队列为空
27. notEmpty.await();//阻塞读线程
28. Object x = items[takeptr];//取值
29. **if** (++takeptr == items.length) takeptr = 0;//如果读索引读到队列的最后一个位置了，那么置为0
30. --count;//个数--
31. notFull.signal();//唤醒写线程
32. **return** x;
33. } **finally** {
34. lock.unlock();
35. }
36. }
37. }

        这是一个处于多线程工作环境下的缓存区，缓存区提供了两个方法，put和take，put是存数据，take是取数据，内部有个缓存队列，具体变量和方法说明见代码，这个缓存区类实现的功能：有多个线程往里面存数据和从里面取数据，其缓存队列(先进先出后进后出)能缓存的最大数值是100，多个线程间是互斥的，当缓存队列中存储的值达到100时，将写线程阻塞，并唤醒读线程，当缓存队列中存储的值为0时，将读线程阻塞，并唤醒写线程，这也是ArrayBlockingQueue的内部实现。下面分析一下代码的执行过程：

        1. 一个写线程执行，调用put方法；

        2. 判断count是否为100，显然没有100；

        3. 继续执行，存入值；

        4. 判断当前写入的索引位置++后，是否和100相等，相等将写入索引值变为0，并将count+1；

        5. 仅唤醒**读线程阻塞队列**中的一个；

        6. 一个读线程执行，调用take方法；

        7. ……

        8. 仅唤醒**写线程阻塞队列**中的一个。

        这就是多个Condition的强大之处，假设缓存队列中已经存满，那么阻塞的肯定是写线程，唤醒的肯定是读线程，相反，阻塞的肯定是读线程，唤醒的肯定是写线程，那么假设只有一个Condition会有什么效果呢，缓存队列中已经存满，这个Lock不知道唤醒的是读线程还是写线程了，如果唤醒的是读线程，皆大欢喜，如果唤醒的是写线程，那么线程刚被唤醒，又被阻塞了，这时又去唤醒，这样就浪费了很多时间。

        本文来自：[高爽|Coder](http://blog.csdn.net/ghsau)，原文地址：<http://blog.csdn.net/ghsau/article/details/7481142>，转载请注明。