队列之实践篇

(转自http://blog.csdn.net/evankaka/article/details/44153709#t1)
生产消费者模式

貌似也是阻塞的问题

花了一些时间终于弄明白这个鸟东东，以前还以为是不复杂的一个东西的，以前一直以为和观察者模式差不多(其实也是差不多的，呵呵)，生产消费者模式应该是可以通过观察者模式来实现的，对于在什么环境下使用现在想的还不是特别清楚，主要是在实际中还没使用过这个。

需要使用到同步，以及线程，属于多并发行列，和观察者模式的差异也就在于此吧，所以实现起来也主要在这里的差异。

参考地址：http://blog.csdn.net/program_think/

在实际的软件开发过程中，经常会碰到如下场景：某个模块负责产生数据，这些数据由另一个模块来负责处理（此处的模块是广义的，可以是类、函数、线程、进程等）。产生数据的模块，就形象地称为生产者；而处理数据的模块，就称为消费者。

单单抽象出生产者和消费者，还够不上是生产者 / 消费者模式。该模式还需要有一个缓冲区处于生产者和消费者之间，作为一个中介。生产者把数据放入缓冲区，而消费者从缓冲区取出数据

◇解耦

假设生产者和消费者分别是两个类。如果让生产者直接调用消费者的某个方法，那么生产者对于消费者就会产生依赖（也就是耦合）。将来如果消费者的代码发生变化，可能会影响到生产者。而如果两者都依赖于某个缓冲区，两者之间不直接依赖，耦合也就相应降低了。

◇支持并发（concurrency）

生产者直接调用消费者的某个方法，还有另一个弊端。由于函数调用是同步的（或者叫阻塞的），在消费者的方法没有返回之前，生产者只好一直等在那边。万一消费者处理数据很慢，生产者就会白白糟蹋大好时光。

使用了生产者 / 消费者模式之后，生产者和消费者可以是两个独立的并发主体（常见并发类型有进程和线程两种，后面的帖子会讲两种并发类型下的应用）。生产者把制造出来的数据往缓冲区一丢，就可以再去生产下一个数据。基本上不用依赖消费者的处理速度。其实当初这个模式，主要就是用来处理并发问题的。

◇支持忙闲不均

缓冲区还有另一个好处。如果制造数据的速度时快时慢，缓冲区的好处就体现出来了。当数据制造快的时候，消费者来不及处理，未处理的数据可以暂时存在缓冲区中。等生产者的制造速度慢下来，消费者再慢慢处理掉。

用了两种方式实现了一下这个模式，主要参考了网上的一些例子才弄明白，这里对队列的实现有很多种方法，需要和具体的应用相结合吧，队列缓冲区很简单，现在已有大量的实现，缺点是在性能上面(内存分配的开销和同步/互斥的开销)，下面的实现都是这种方式；环形缓冲区(减少了内存分配的开销)，双缓冲区(减少了同步/互斥的开销)。
第一个例子是使用的信号量的东东，没有执行具体的东西，只是实现了这个例子，要做复杂的业务逻辑的话需要自己在某些方法内去具体实现

代码如下：

消费者：

Java代码

收藏代码

```
1. public class TestConsumer implements Runnable {
2.
3.     TestQueue obj;
4.
5.     public TestConsumer(TestQueue tq){
6.         this.obj=tq;
7.     }
8.
9.     public void run() {
10.         try {
11.             for(int i=0;i<10;i++){
12.                 obj.consumer();
13.             }
14.         } catch (Exception e) {
15.             e.printStackTrace();
16.         }
17.     }
18. }
```

生产者：

Java代码

收藏代码

```
1. public class TestProduct implements Runnable {
2.
3.     TestQueue obj;
4.
5.     public TestProduct(TestQueue tq){
6.         this.obj=tq;
7.     }
8.
9.     public void run() {
10.         for(int i=0;i<10;i++){
11.             try {
12.                 obj.product("test"+i);
13.             } catch (Exception e) {
14.                 e.printStackTrace();
15.             }
16.         }
17.     }
18.
19. }
```

队列(使用了信号量，采用synchronized进行同步，采用lock进行同步会出错，或许是还不知道实现的方法)：

Java代码

收藏代码

```
1. public static Object signal=new Object();
2.     boolean bFull=false;
3.     private List thingsList=new ArrayList();
```

```
    private final ReentrantLock lock = new ReentrantLock(true);
    BlockingQueue q = new ArrayBlockingQueue(10);
    /**
     * 生产
     * @param thing
     * @throws Exception
     */
    public void product(String thing) throws Exception{
        synchronized(signal){
            if(!bFull){
                bFull=true;
                  //产生一些东西，放到 thingsList 共享资源中
                System.out.println("product");
                System.out.println("仓库已满，正等待消费...");
                thingsList.add(thing);
                signal.notify(); //然后通知消费者
            }
            signal.wait(); // 然后自己进入signal待召队列

        }

    }

    /**
     * 消费
     * @return
     * @throws Exception
     */
    public String consumer()throws Exception{
        synchronized(signal){
            if(!bFull)  {
                    signal.wait(); // 进入signal待召队列，等待生产者的通知
            }
            bFull=false;
            // 读取buf 共享资源里面的东西
            System.out.println("consume");
            System.out.println("仓库已空，正等待生产...");
            signal.notify(); // 然后通知生产者
        }
        String result="";
        if(thingsList.size()>0){
            result=thingsList.get(thingsList.size()-1).toString();
            thingsList.remove(thingsList.size()-1);
        }
        return result;
    }
```

测试代码：

Java代码

收藏代码

```
public class TestMain {
    public static void main(String[] args) throws Exception{
        TestQueue tq=new TestQueue();
        TestProduct tp=new TestProduct(tq);
        TestConsumer tc=new TestConsumer(tq);
        Thread t1=new Thread(tp);
        Thread t2=new Thread(tc);
        t1.start();
        t2.start();
    }
```

```
11. }
```

运行结果：

Java代码

收藏代码

```
1. product
2. 仓库已满，正等待消费...
3. consume
4. 仓库已空，正等待生产...
5. product
6. 仓库已满，正等待消费...
7. consume
8. 仓库已空，正等待生产...
9. product
10. 仓库已满，正等待消费...
11. consume
12. 仓库已空，正等待生产...
13. product
14. 仓库已满，正等待消费...
15. consume
16. 仓库已空，正等待生产...
17. product
18. 仓库已满，正等待消费...
19. consume
20. 仓库已空，正等待生产...
21. product
22. 仓库已满，正等待消费...
23. consume
24. 仓库已空，正等待生产...
25. product
26. 仓库已满，正等待消费...
27. consume
28. 仓库已空，正等待生产...
29. product
30. 仓库已满，正等待消费...
31. consume
32. 仓库已空，正等待生产...
33. product
34. 仓库已满，正等待消费...
35. consume
36. 仓库已空，正等待生产...
37. product
38. 仓库已满，正等待消费...
39. consume
40. 仓库已空，正等待生产...
```

第二种发放使用java.util.concurrent.BlockingQueue类来重写的队列那个类，使用这个方法比较简单，并且性能上也没有什么问题。

这是jdk里面的例子

Java代码

收藏代码

```
1. * class Producer implements Runnable {
2.  *   private final BlockingQueue queue;
3.  *   Producer(BlockingQueue q) { queue = q; }
4.  *   public void run() {
5.  *     try {
```

```
6.  *       while(true) { queue.put(produce()); }
7.  *     } catch (InterruptedException ex) { ... handle ...}
8.  *   }
9.  *   Object produce() { ... }
10.  * }
11.  *
12.  * class Consumer implements Runnable {
13.  *   private final BlockingQueue queue;
14.  *   Consumer(BlockingQueue q) { queue = q; }
15.  *   public void run() {
16.  *     try {
17.  *       while(true) { consume(queue.take()); }
18.  *     } catch (InterruptedException ex) { ... handle ...}
19.  *   }
20.  *   void consume(Object x) { ... }
21.  * }
22.  *
23.  * class Setup {
24.  *   void main() {
25.  *     BlockingQueue q = new SomeQueueImplementation();
26.  *     Producer p = new Producer(q);
27.  *     Consumer c1 = new Consumer(q);
28.  *     Consumer c2 = new Consumer(q);
29.  *     new Thread(p).start();
30.  *     new Thread(c1).start();
31.  *     new Thread(c2).start();
32.  *   }
33.  * }
```

jdk1.5以上的一个实现，使用了Lock以及条件变量等东西

Java代码

收藏代码

```
1. class BoundedBuffer {
2.   final Lock lock = new ReentrantLock();
3.   final Condition notFull  = lock.newCondition();
4.   final Condition notEmpty = lock.newCondition();
5.
6.   final Object[] items = new Object[100];
7.   int putptr, takeptr, count;
8.
9.   public void put(Object x) throws InterruptedException {
10.    lock.lock();
11.    try {
12.      while (count == items.length)
13.        notFull.await();
14.      items[putptr] = x;
15.      if (++putptr == items.length) putptr = 0;
16.      ++count;
17.      notEmpty.signal();
18.    } finally {
19.      lock.unlock();
20.    }
21.  }
22.
23.  public Object take() throws InterruptedException {
24.    lock.lock();
25.    try {
26.      while (count == 0)
```

```
27.         notEmpty.await();
28.       Object x = items[takeptr];
29.       if (++takeptr == items.length) takeptr = 0;
30.       --count;
31.       notFull.signal();
32.       return x;
33.     } finally {
34.       lock.unlock();
35.     }
36.   }
37. }
```