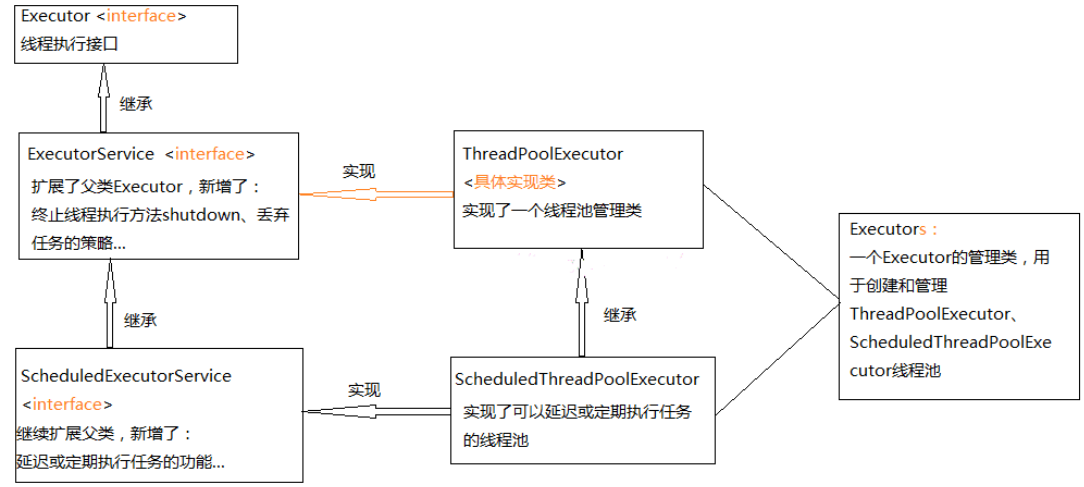
1、线程池 ThreadPool 相关

在 java.util.concurrent 包下，提供了一系列与线程池相关的类。合理的使用线程池，可以带来多个好处：

- 降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗；
- 提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行；
- 提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

线程池可以应对突然大爆发量的访问，通过有限个固定线程为大量的操作服务，减少创建和销毁线程所需的时间。

与线程执行、线程池相关类的关系如图：



我们一般通过工具类 Executors 的静态方法（如 newFixedThreadPool()）来获取 ThreadPoolExecutor 线程池或静态方法（如 newScheduledThreadPool()）来获取 ScheduleThreadPoolExecutor 线程池。如下使用：

```
ExecutorService threadpool= Executors.newFixedThreadPool(10);
```

我们指定了获取 10 个数量的固定线程池，Executors 中有很多重载的获取线程池的方法，比如可以通过自定义的 ThreadFactory 来为每个创建出来的 Thread 设置更为有意义的名称。Executors 创建线程池的方法内部也就是 new 出新的 ThreadPoolExecutor 或 ScheduleThreadPoolExecutor，给我们配置了很多默认的设置。如下：

```
public static ExecutorService newFixedThreadPool(int nThreads) {

    return new ThreadPoolExecutor(nThreads, nThreads,

                                  0L, TimeUnit.MILLISECONDS,

                                  new LinkedBlockingQueue<Runnable>());

}
```

上面通过 ThreadPoolExecutor 的构造方法，为我们创建了一个线程池，很多参数 Executors 工具类自动为我们配置好了。创建一个 ThreadPoolExecutor 线程池一般需要以下几个参数：

```
public ThreadPoolExecutor(int corePoolSize,

                          int maximumPoolSize,

                          long keepAliveTime,

                          TimeUnit unit,

                          BlockingQueue<Runnable> workQueue,

                          ThreadFactory threadFactory,

                          RejectedExecutionHandler handler)
```

- corePoolSize（线程池的基本大小）：当提交一个任务到线程池时，线程池会创建一个线程来执行任务，即使其他空闲的基本线程能够执行新任务也会创建线程，等到需要执行的任务数大于线程池基本大小时就不再创建。如果调用了线程池的 prestartAllCoreThreads 方法，线程池会提前创建并启动所有基本线程。
- maximumPoolSize（线程池最大大小）：线程池允许创建的最大线程数。如果队列满了，并且已创建的线程数小于最大线程数，则线程池会再创建新的线程执行任务。值得注意的是如果使用了无界的任务队列这个参数就没什么效果。

- keepAliveTime（线程活动保持时间）：线程池的工作线程空闲后，保持存活的时间。所以如果任务很多，并且每个任务执行的时间比较短，可以调大这个时间，提高线程的利用率。
- TimeUnit（线程活动保持时间的单位）：可选的单位有天（DAYS），小时（HOURS），分钟（MINUTES），毫秒(MILLISECONDS)等。
- workQueue（任务队列）：用于保存等待执行的任务的阻塞队列。 可以选择以下几个阻塞队列：ArrayBlockingQueue、LinkedBlockingQueue、SynchronousQueue、PriorityBlockingQueue
- threadFactory：用于设置创建线程的工厂，可以通过线程工厂给每个创建出来的线程设置更有意义的名字。
- handler（饱和策略）：当队列和线程池都满了，说明线程池处于饱和状态，那么必须采取一种策略处理提交的新任务。这个策略默认情况下是AbortPolicy，表示无法处理新任务时抛出异常。

我们尽量优先使用 Executors 提供的静态方法来创建线程池，如果 Executors 提供的方法无法满足要求，再自己通过 ThreadPoolExecutor 类来创建线程池。

提交任务的两种方式：

（1）通过 execute()方法，如：

```
ExecutorService threadpool= Executors.newFixedThreadPool(10);

threadpool.execute(new Runnable(){

    ...

}

);
```

这种方式提交没有返回值，也就不能判断任务是否被线程池执行成功。

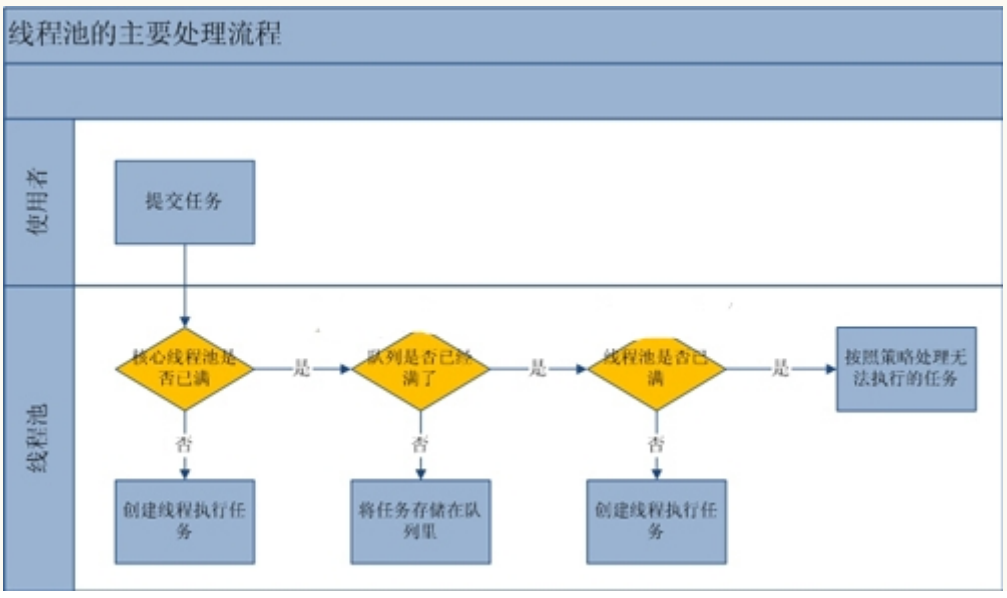
（2）通过 submit()方法，如：

```
Future<?> future = threadpool.submit(new Runnable(){...});
```

```
try {
        Object res = future.get();
    } catch (InterruptedException e) {
        // 处理中断异常
        e.printStackTrace();
    } catch (ExecutionException e) {
        // 处理无法执行任务异常
        e.printStackTrace();
    }finally{
        // 关闭线程池
        executor.shutdown();
    }
```

使用 submit 方法来提交任务，它会返回一个 Future 对象，通过 future 的 get 方法来获取返回值，get 方法会阻塞住直到任务完成，而使用 get(long timeout, TimeUnit unit)方法则会阻塞一段时间后立即返回，这时有可能任务没有执行完。

线程池工作流程分析：



从上图我们可以看出，当提交一个新任务到线程池时，线程池的处理流程如下：

- 首先线程池判断基本线程池是否已满（< corePoolSize ？）？没满，创建一个工作线程来执行任务。满了，则进入下个流程。
- 其次线程池判断工作队列是否已满？没满，则将新提交的任务存储在工作队列里。满了，则进入下个流程。
- 最后线程池判断整个线程池是否已满（< maximumPoolSize ？）？没满，则创建一个新的工作线程来执行任务，满了，则交给饱和策略来处理这个任务。

也就是说，线程池优先要创建出基本线程池大小（corePoolSize）的线程数量，没有达到这个数量时，每次提交新任务都会直接创建一个新线程，当达到了基本线程数量后，又有新任务到达，优先放入等待队列，如果队列满了，才去创建新的线程（不能超过线程池的最大数 maxmumPoolSize）。

ThreadPoolExecutor 简单实例：

```
public class BankCount {

    public synchronized void addMoney(int money){

        //存钱

        System.out.println(Thread.currentThread().getName() + ">存入：" + money);

    }

    public synchronized void getMoney(int money){

        //取钱

        System.out.println(Thread.currentThread().getName() + ">取钱：" + money);

    }

}
```

测试类：

```
public class BankTest {

    public static void main(String[] args) {

        final BankCount bankCount = new BankCount();

        ExecutorService executor = Executors.newFixedThreadPool(10);

        executor.execute(new Runnable() {

            //存钱线程

            @Override

                        public void run() {

                int i = 5;

                while(i-- > 0){

                    bankCount.addMoney(200);

                    try {

                        Thread.sleep(500);

                    }

                    catch (InterruptedException e) {

                        e.printStackTrace();

                    }

                }

            }

        }

        );

        Future<?> future = executor.submit(new Runnable() {

            //取钱线程
```

```
        @Override
                    public void run() {
            int i = 5;
            while(i-- > 0){
                try {
                    Thread.sleep(500);
                }
                catch (InterruptedException e) {
                    e.printStackTrace();
                }
                bankCount.getMoney(200);
            }
        }
    }
    );
    try {
        Object res = future.get();
        System.out.println(res);
    }
    catch (InterruptedException e) {
        // 处理中断异常
        e.printStackTrace();
    }
```

```
        catch (ExecutionException e) {
            // 处理无法执行任务异常
            e.printStackTrace();
        }
        finally{
            // 关闭线程池
            executor.shutdown();
        }
    }
}
```

打印结果如下：

```
pool-1-thread-1>存入：200
pool-1-thread-1>存入：200
pool-1-thread-2>取钱：200
pool-1-thread-1>存入：200
pool-1-thread-2>取钱：200
pool-1-thread-1>存入：200
pool-1-thread-2>取钱：200
pool-1-thread-1>存入：200
pool-1-thread-2>取钱：200
pool-1-thread-2>取钱：200
null
```

可以看到，打印出来的 future.get()获取的结果为 null，这是因为 Runnable 是没有返回值的，需要返回值要使用 Callable，这里就不再细说了。

2、生产者和消费者模型

生产者消费者模型，描述是：有一块缓冲区作为仓库，生产者可以将产品放入仓库，消费者可以从仓库中取走产品。解决消费者和生产者问题的核心在于保证同一资源被多个线程并发访问时的完整性。一般采用信号量或加锁机制解决。下面介绍 Java 中解决生产者和消费者问题主要三种仿：

（1）wait() / notify()、notifyAll()

- wait 和 notify 方法是 Object 的两个方法，因此每个类都会拥有这两个方法。
- wait()方法：使当前线程处于等待状态，放弃锁，让其他线程执行。
- notify()方法：唤醒其他等待同一个锁的线程，放弃锁，自己处于等待状态。

如下例子：

```
/**
 * 仓库
 */
public class Storage {
    private static final int MAX_SIZE = 100;
    //仓库的最大容量
    private List<Object> data = new ArrayList<Object>();
    //存储载体
    /**
     * 生产操作
     */
    public synchronized void produce(int num){
```

```
        if(data.size() + num > MAX_SIZE){
            //如果生产这些产品将超出仓库的最大容量，则生产操作阻塞
            System.out.println("生产操作-->数量：" + num + "，超出仓库容量，生产阻塞！-----库存：" + data.size());
            try {
                wait();
            }
            catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        //到这里，表示可以正常生产产品
        for (int i = 0; i < num; i++){
            //生产num个产品
            data.add(new Object());
        }
        System.out.println("生产操作-->数量：" + num + "，成功入库~------库存：" + data.size());
        //生产完产品后，唤醒其他等待消费的线程
        notify();
    }
    /**
     * 消费操作
     */
```

```
    public synchronized void consume(int num){

        if(data.size() - num < 0){

            //如果产品数量不足

            System.out.println("消费操作-->数量：" + num + "，库存不足，消费阻塞！-----
-库存：" + data.size());

            try {

                wait();

            }

            catch (InterruptedException e) {

                e.printStackTrace();

            }

        }

        //到这里，表示可以正常消费

        for (int i = 0; i < num; i++){

            //消费num 个产品

            data.remove(0);

        }

        System.out.println("消费操作-->数量：" + num + "，消费成功~------库存：" + dat
a.size());

        //消费完产品后，唤醒其他等待生产的线程

        notify();

    }

}
```

生产者：

```
public class Producer implements Runnable{

    private Storage storage;

    private int num;

    //每次生产多少个

    public Producer(Storage sto,int num){

        storage = sto;

        this.num = num;

    }

    @Override

        public void run() {

        storage.produce(num);

    }

}
```

消费者：

```
public class Consumer implements Runnable{

    private Storage storage;

    private int num;

    //每次消费多少个

    public Consumer(Storage sto,int num){

        storage = sto;

        this.num = num;

    }
```

```
    @Override

        public void run() {

        storage.consume(num);

    }

}
```

测试类：

```
public class StorageTest {

    public static void main(String[] args) {

        Storage storage = new Storage();

        ExecutorService taskSubmit = Executors.newFixedThreadPool(10);

        //来使用使用上一节我们总结的线程池知识

        //给定4个消费者

        taskSubmit.submit(new Consumer(storage, 30));

        taskSubmit.submit(new Consumer(storage, 10));

        taskSubmit.submit(new Consumer(storage, 20));

        //给定6个生产者

        taskSubmit.submit(new Producer(storage, 70));

        taskSubmit.submit(new Producer(storage, 10));

        taskSubmit.submit(new Producer(storage, 20));

        taskSubmit.submit(new Producer(storage, 10));

        taskSubmit.submit(new Producer(storage, 10));

        taskSubmit.submit(new Producer(storage, 10));
```

```
        taskSubmit.shutdown();

    }

}
```

打印结果：

```
消费操作-->数量：30，库存不足，消费阻塞！------库存：0

生产操作-->数量：10，成功入库~------库存：10

生产操作-->数量：70，成功入库~------库存：80

生产操作-->数量：10，成功入库~------库存：90

生产操作-->数量：10，成功入库~------库存：100

生产操作-->数量：20，超出仓库容量，生产阻塞！------库存：100

消费操作-->数量：10，消费成功~------库存：90

生产操作-->数量：20，成功入库~------库存：110

生产操作-->数量：10，超出仓库容量，生产阻塞！------库存：110

消费操作-->数量：20，消费成功~------库存：90

消费操作-->数量：30，消费成功~------库存：60

生产操作-->数量：10，成功入库~------库存：70
```

在仓库中，唤醒我们使用的是 notify()而没有使用 notifyAll()，是因为在这里，如果测试数据设置不当很容易造成死锁（比如一下唤醒了所有的生产进程），因为使用 wait 和 notify 有一个缺陷：

逻辑本应该要这样设计的，在 produce()操作后，只要唤醒等待同一把锁的消费者进程，在 consume()后，唤醒等待同一把锁的生产者进程，而 notify()或 notifyAll()将生产者和消费者线程都唤醒了。下面的第二种方法可以解决这个问题。

wait 和 notify 在“类消费者和生产者”问题上也很有用，比如，在 A 类的某个方法中调用了传进来的 B 对象的一个方法，A 类方法的后面代码依赖于刚刚调用

的 B 的返回值，但是 B 对象的这个方法是一个异步的操作，此时就可以在 A 方法中调用完 B 对象的方法后自我阻塞，即调用 wait()方法，而在 B 对象的那个方法中，待异步操作完成后，调用 notify()，唤醒处于等待同一锁对象的线程。如下：

A 类的某个方法中：

```
XmppManager xmppManager = notificationService.getXmppManager();

if(xmppManager != null){

    if(!xmppManager.isAuthenticated()){

        try {

            synchronized (xmppManager) {

                //等待客户端连接认证成功

                Log.d(LOGTAG, "wait for authenticated...");

                xmppManager.wait();

            }

        }

        catch (InterruptedException e) {

            e.printStackTrace();

        }

    }

    //运行到此处，说明是认证成功的，有两种可能，一是运行速度很快调用notificationService.getXmppManager()后直接返回了结果，二是B 中处理完了调用notify 方法

    Log.d(LOGTAG, "authenticated already. send SetTagsIQ now...");

    B 中处理完后：

    //客户端连接认证成功后，唤醒拥有 xmppManager 锁的对象
```

```
    synchronized (xmppManager) {

        xmppManager.notifyAll();

    }
```

（2）await() / signal()

在 JDK1.5 之引入 concurrent 包之后，新引入了 await()和 signal()方法来做同步，功能和 wait()和 notify()方法相同，可以完全取代，但 await()和 signal()需要和 Lock 机制（关于 Lock 机制前面已总结）结合使用，更加灵活。正如第一种所说，可以通过调用 Lock 的 newCondition()方法依次获取两个条件变量，一个针对仓库空的，一个针对仓库满的条件变量，通过添加变量进行同步控制。

修改仓库类 Storage：

```
/**

 * 仓库

 */

public class Storage {

    private static final int MAX_SIZE = 100;

    //仓库的最大容量

    private List<Object> data = new ArrayList<Object>();

    //存储载体

    private Lock lock = new ReentrantLock();

    //可重入锁

    private Condition full = lock.newCondition();

    //仓库满的条件变量

    private Condition empty = lock.newCondition();

    //仓库空时的条件变量
```

```
    /**
     * 生产操作
     */
    public void produce(int num){
        lock.lock();
        //加锁
        if(data.size() + num > MAX_SIZE){
            //如果生产这些产品将超出仓库的最大容量，则生产操作阻塞
            System.out.println("生产操作-->数量：" + num + "，超出仓库容量，生产阻塞！-----库存：" + data.size());
            try {
                full.await();
                //阻塞
            }
            catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        //到这里，表示可以正常生产产品
        for (int i = 0; i < num; i++){
            //生产num 个产品
            data.add(new Object());
        }
```

```
        System.out.println("生产操作-->数量：" + num + "，成功入库~------库存：" + dat
a.size());

        //生产完产品后，唤醒其他等待消费的线程

        empty.signalAll();

        lock.unlock();

        //释放锁

    }

    /**

     * 消费操作

     */

    public void consume(int num){

        lock.lock();

        //加锁

        if(data.size() - num < 0){

            //如果产品数量不足

            System.out.println("消费操作-->数量：" + num + "，库存不足，消费阻塞！-----
-库存：" + data.size());

            try {

                empty.await();

                //阻塞

            }

            catch (InterruptedException e) {

                e.printStackTrace();

            }
```

```
        }

        //到这里，表示可以正常消费

        for (int i = 0; i < num; i++){

            //消费num 个产品

            data.remove(0);

        }

        System.out.println("消费操作-->数量：" + num + "，消费成功~------库存：" + dat
a.size());

        //消费完产品后，唤醒其他等待生产的线程

        full.signalAll();

        lock.unlock();

        //释放锁

    }

}
```

打印结果：

```
消费操作-->数量：30，库存不足，消费阻塞！------库存：0

消费操作-->数量：10，库存不足，消费阻塞！------库存：0

消费操作-->数量：20，库存不足，消费阻塞！------库存：0

生产操作-->数量：70，成功入库~------库存：70

生产操作-->数量：10，成功入库~------库存：80

生产操作-->数量：10，成功入库~------库存：90

生产操作-->数量：10，成功入库~------库存：100
```

```
生产操作-->数量：10，超出仓库容量，生产阻塞！------库存：100

消费操作-->数量：30，消费成功~------库存：70

消费操作-->数量：10，消费成功~------库存：60

消费操作-->数量：20，消费成功~------库存：40

生产操作-->数量：10，成功入库~------库存：50

生产操作-->数量：20，成功入库~------库存：70
```

使用 await 和 signal 后，加锁解锁操作就交给了 Lock，不用再使用 synchronized 同步（具体可看前面总结的同步的实现方法），在 produce 中满仓后阻塞，生产完后唤醒等待的消费线程，consume 中库存不足后阻塞，消费完后唤醒等待的生产者线程，表示可以消费了。

（3）BlockingQueue 阻塞队列方式

在上一节关于线程池的总结中，我们看到了要创建一个线程池如 ThreadPoolExecutor，需要传入一个任务队列即 BlockingQueue，BlockingQueue(接口)用于保存等待执行的任务的阻塞队列。 可以选择以下几个阻塞队列：ArrayBlockingQueue、LinkedBlockingQueue、SynchronousQueue、PriorityBlockingQueue。

ArrayBlockingQueue：是一个基于数组结构的有界阻塞队列，此队列按 FIFO（先进先出）原则对元素进行排序。
LinkedBlockingQueue：一个基于链表结构的阻塞队列，此队列按 FIFO （先进先出） 排序元素，吞吐量通常要高于 ArrayBlockingQueue。静态工厂方法 Executors.newFixedThreadPool()使用了这个队列。
SynchronousQueue：一个不存储元素的阻塞队列。每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常要高于 LinkedBlockingQueue，静态工厂方法 Executors.newCachedThreadPool 使用了这个队列。
PriorityBlockingQueue：一个具有优先级的无限阻塞队列。

BlockingQueue 的所有实现类内部都是已经实现了同步的队列，实现的方式采用的是上面介绍的第二种 await()/signal() + Lock 同步的机制。在生成阻塞队列时，可以指定队列大小。用于阻塞操作的方法主要为：

- put()方法：插入一个元素，如果超过容量则自我阻塞，等待唤醒；
- take()方法：取走一个元素，如果容量不足了，自我阻塞，等待唤醒；

put 和 take 内部自己实现了 await 和 signal、lock 的机制处理，不再需要我们做相应操作。修改 Storage 代码如下：

```
public class Storage {

    private static final int MAX_SIZE = 100;

    //仓库的最大容量

    private BlockingQueue<Object> data = new LinkedBlockingQueue<Object>(MAX_SIZE);

    //使用阻塞队列作为存储载体

    /**

     * 生产操作

     */

    public void produce(int num){

        if(data.size() == MAX_SIZE){

            //如果仓库已达最大容量

            System.out.println("生产操作-->仓库已达最大容量！");

        }

        //到这里，表示可以正常生产产品

        for (int i = 0; i < num; i++){

            //生产num 个产品

            try {

                data.put(new Object());

                //put 内部自动实现了判断，超过最大容量自动阻塞

            }

            catch (InterruptedException e) {
```

```
                e.printStackTrace();

            }

        }

        System.out.println("生产操作-->数量：" + num + "，成功入库~------库存：" + dat
a.size());

    }

    /**

     * 消费操作

     */

    public void consume(int num){

        if(data.size() ==  0){

            //如果产品数量不足

            System.out.println("消费操作--库存不足！");

        }

        //到这里，表示可以正常消费

        for (int i = 0; i < num; i++){

            //消费num 个产品

            try {

                data.take();

                //take 内部自动判断，消耗后库存是否充足，不足自我阻塞

            }

            catch (InterruptedException e) {

                e.printStackTrace();
```

```
            }

        }

        System.out.println("消费操作-->数量：" + num + "，消费成功~------库存：" + dat
a.size());

    }

}
```

打印结果：

```
消费操作--库存不足！

消费操作--库存不足！

消费操作--库存不足！

生产操作-->数量：70，成功入库~------库存：45

消费操作-->数量：30，消费成功~------库存：45

生产操作-->数量：10，成功入库~------库存：56

生产操作-->数量：20，成功入库~------库存：75

生产操作-->数量：10，成功入库~------库存：85

生产操作-->数量：10，成功入库~------库存：89

消费操作-->数量：10，消费成功~------库存：60

生产操作-->数量：10，成功入库~------库存：70

消费操作-->数量：20，消费成功~------库存：70
```

可以看到，Storage 中 produce 和 consume 方法中我们直接通过 put 和 take 方法往容器中添加或移除产品即可，没有进行逻辑控制（其实上面两个方法中 if 都可以去掉，只是为了打印效果才加上的），这是因为 BlockingQueue 内部已经实现了，不需要我们再次控制。

同时，我们看到打印的库存信息出现了不匹配，这个主要是因为我们的打印语句 Systm.out.println()没有被同步导致的，因为同步语句只是在 put 和 take 方法内部，而我们打印语句中使用了 data 这个共享变量。这里因为我们需要看效果，所以才加的打印语句，并不影响我们对 BlockingQueue 的使用。

因此，在 Java 中，使用 BlockingQueue 阻塞队列的方式可以很方便的为我们处理生产者消费则问题，推荐使用。

在我们的编程生涯中，我们自己要去写生产者和消费者问题，多是前面第一种介绍的“类似消费者生产者问题”上。

解决生产者和消费者问题还有管道的方式，即在生产者和消费者之间建立一个管道缓冲区，Java 中用 PipedInputStream / PipedOutputStream 实现，由于这种方式对于传输对象不易封装，因此实用性不高，就不具体介绍了。

3、sleep 和 wait 的区别

sleep 是 Thread 的静态方法，wait 是 Object 的方法。两个方法都会暂停当前线程

- sleep 使当前线程阻塞，让出 CPU，给其他线程执行的机会；如果当前线程拥有锁，不会释放锁，也即“睡着我也要拥有锁”。睡眠时间一到，进入就绪状态，如果当前 CPU 空闲，才会继续执行。
- wait 方法调用后，当前线程进入阻塞状态，进入到和该对象（即谁调用了 wait()方法，如 list.wait()）相关的等待池中。，让出 CPU，给其他线程执行的机会；当超时间过了或者别的线程调用了 notify()或 notifyAll()方法时才会唤醒当前等待同一把锁的线程。
- wait 方法必须要放在同步块中，如 syncbronized 或 Lock 同步中。

所以 sleep 和 wait 的主要区别是：

- sleep：保持锁，睡眠时间到进入就绪状态；
- wait：释放锁，等待其他线程的 notify 操作或超时唤醒。

下一篇将是关于 super、构造方法、transient 的理解、foreach 与正常 for 循环的效率对比等...