多线程那些事儿

现在只要出去面试，关于“Java 多线程”的问题，几乎没有一家单位不问的，可见其重要性。于是博主抽空研究了一下，确实很有意思！以下是我综合整理了网上的各种资料，和个人的一些理解，写的一篇总结博文，仅供学习、交流。

（一）多线程的概念

多线程的概念，简单理解：一个进程运行时产生了不止一个线程。

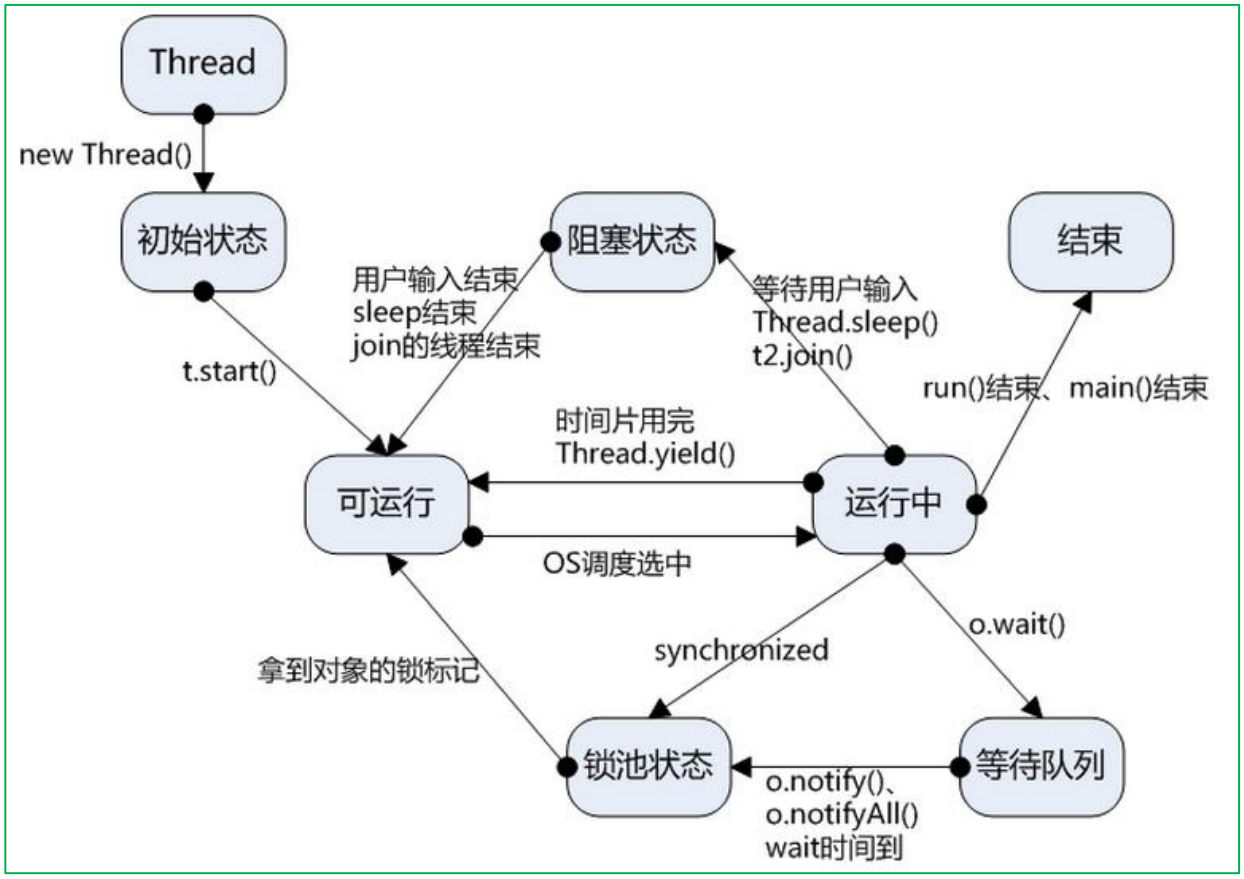
进程的概念，简单理解：正在运行的程序的实例。

两者之间的关系：进程就是线程的容器。

（二）多线程的好处

使用多线程可以提高 CPU 的利用率。在多线程程序中，一个线程必须等待的时候，CPU 可以运行其它的线程而不是等待，这样就大大提高了程序的效率。

（三）线程状态转换



1、新建状态（New）：新创建了一个线程对象。

2、就绪状态（Runnable）：线程对象创建后，其他线程调用了该对象的 start()方法。该状态的线程位于可运行线程池中，变得可运行，等待获取 CPU 的使用权。

3、运行状态（Running）：就绪状态的线程获取了 CPU，执行程序代码。

4、**阻塞状态（Blocked）**：阻塞状态是线程因为某种原因放弃 CPU 使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。阻塞的情况分三种：

①**等待阻塞**：运行的线程执行 **wait()**方法，JVM 会把该线程放入等待池中。

②**同步阻塞**：运行的线程在获取对象的**同步锁**时，若该同步锁被别的线程占用，则 JVM 会把该线程放入锁池中。

③**其他阻塞**：运行的线程执行 **sleep()**或 **join()**方法，或者发出了 **I/O 请求**时，JVM 会把该线程置为阻塞状态。当 sleep()状态超时、join()等待线程终止或者超时、或者 I/O 处理完毕时，线程重新转入就绪状态。

5、**死亡状态（Dead）**：线程执行完了或者因异常退出了 run()方法，该线程结束生命周期。

（五）创建线程的方式

创建线程的方式主要有两种：

（1）一种是**继承 Tread 类**：

```
package com.liyan.Test;
public class MyThread extends Thread {  //继承 Tread 类
    @Override
    public void run(){                  //重写 run 方法
        try{
            .......
        }catch(Exception e){
            e.printStackTrace();
        }
    }
    public static void main(String[] args) {
        MyThread myThread = new MyThread();  //产生新的线程对象
        myThread.start();                    //开启线程
    }
}
```

（2）一种是**实现 Runnable 接口**，

```
package com.liyan.gcTest;
public class MyThread implements Runnable { //实现 Runnable 接口
    @Override
    public void run(){   //重写 run 方法
        try{
            .......
        }catch(Exception e){
            e.printStackTrace();
        }
    }
```

```
    public static void main(String[] args) {
        MyThread myThread = new MyThread();     //产生新的线程对象
        Thread thread = new Thread(myThread);  //将创建的线程作为参数传入
        thread.start();                         //开启线程
    }
}
```

（六）终止线程的方式

1. 使用退出标志终止线程

当 run 方法执行完后，线程就会退出。但有时 run 方法是永远不会结束的。如在服务端程序中使用线程进行监听客户端请求，或是其他的需要循环处理的任务。在这种情况下，一般是将这些任务放在一个循环中，如 while 循环。如果想让循环永远运行下去，可以使用 while（true）{……}来处理。但要想使 while 循环在某一特定条件下退出，最直接的方法就是设一个 boolean 类型的标志，并通过设置这个标志为 true 或 false 来控制 while 循环是否退出。下面给出了一个利用退出标志终止线程的例子。

```
package com.liyan.thread;
public class ThreadDemo extends Thread {
    public volatile boolean exit = false;
    public void run() {
        while (!exit);
    }

    public static void main(String[] args) throws Exception {
        ThreadDemo thread = new ThreadDemo();
        thread.start();
        sleep(5000); // 主线程延迟 5 秒
        thread.exit = true; // 终止线程 thread
        thread.join();
        System.out.println("线程退出!");
    }
}
```

在上面代码中定义了一个退出标志 exit，当 exit 为 true 时，while 循环退出，exit 的默认值为 false.在定义 exit 时，使用了一个 Java 关键字 volatile，这个关键字的目的是使 exit 同步，也就是说在同一时刻只能由一个线程来修改 exit 的值。

2. 使用 stop 方法终止线程

使用 stop 方法可以强行终止正在运行或挂起的线程。我们可以使用如下的代码来终止线程：

```
thread.stop();
```

虽然使用上面的代码可以终止线程，但使用 stop 方法是很危险的，就象突然关闭计算机电源，而不是按正常程序关机一样，可能会产生不可预料的结果，因此，并不推荐使用 stop

方法来终止线程。

3. 使用 interrupt 方法终止线程

使用 interrupt 方法来终端线程可分为两种情况：

（1）线程处于阻塞状态，如使用了 sleep 方法。

（2）使用 while（！isInterrupted（））{……}来判断线程是否被中断。

在第一种情况下使用 interrupt 方法，sleep 方法将抛出一个 InterruptedException 例外，而在第二种情况下线程将直接退出。下面的代码演示了在第一种情况下使用 interrupt 方法。

```
package com.liyan.thread;
public class ThreadDemo extends Thread {
    public void run() {
        try {
            sleep(50000); // 延迟 50 秒
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }
    }

    public static void main(String[] args) throws Exception {
        Thread thread = new ThreadDemo();
        thread.start();
        System.out.println("在 50 秒之内按任意键中断线程!");
        System.in.read();
        thread.interrupt();
        thread.join();
        System.out.println("线程已经退出!");
    }
}
```

输出结果：

```
在 50 秒之内按任意键中断线程!
liyan
sleep interrupted
线程已经退出!
```

在调用 interrupt 方法后， sleep 方法抛出异常，然后输出错误信息：sleep interrupted.

注意：在 Thread 类中有两个方法可以判断线程是否通过 interrupt 方法被终止。一个是静态的方法 interrupted()，一个是非静态的方法 isInterrupted()，这两个方法的区别是 interrupted 用来判断当前线是否被中断，而 isInterrupted 可以用来判断其他线程是否被中断。因此，while（！isInterrupted（））也可以换成 while （！Thread.interrupted（））。

【参考：http://www.bitscn.com/pdb/java/200904/161228_3.html】

（七）Daemon 线程

在 Java 中有两类线程：**User Thread(用户线程)**、**Daemon Thread(守护线程)**

用个比较通俗的比如，任何一个守护线程都是整个 JVM 中所有非守护线程的保姆：

只要当前 JVM 实例中尚存在任何一个非守护线程没有结束，守护线程就全部工作；只有当最后一个非守护线程结束时，守护线程随着 JVM 一同结束工作。

Daemon 的作用是为其他线程的运行提供便利服务，它的**优先级比较低**，用于为系统中的其它对象和线程提供服务。守护线程不依赖于终端，但是依赖于系统，**与系统“同生共死”**。

守护线程最典型的应用就是 **GC (垃圾回收器)**，当我们的程序中不再有任何运行的 hread, 程序就不会再产生垃圾，垃圾回收器也就无事可做，所以当垃圾回收线程是 JVM 上仅剩的线程时，垃圾回收线程会自动离开。它始终在低级别的状态中运行，用于实时监控和管理系统中的可回收资源。

User 和 Daemon 两者几乎没有区别，唯一的不同之处就在于虚拟机的离开：如果 User Thread 已经全部退出运行了，只剩下 Daemon Thread 存在了，虚拟机也就退出了。 因为没有了被守护者，Daemon 也就没有工作可做了，也就没有继续运行程序的必要了。

```
// 设定 daemonThread 为 守护线程，default false(非守护线程)
daemonThread.setDaemon(true);

// 验证当前线程是否为守护线程，返回 true 则为守护线程
daemonThread.isDaemon();
```

这里有几点需要**注意**：

（1）**thread.setDaemon(true)必须在 thread.start()之前设置**，否则会跑出一个 IllegalThreadStateException 异常。你不能把正在运行的常规线程设置为守护线程。

（2）**在 Daemon 线程中产生的新线程也是 Daemon 的**。

（3）**不要认为所有的应用都可以分配给 Daemon 来进行服务，比如读写操作或者计算逻辑**。 因为你不可能知道在所有的 User 完成之前，Daemon 是否已经完成了预期的服务任务。一旦 User 退出了，可能大量数据还没有来得及读入或写出，计算任务也可能多次运行结果不一样。这对程序是毁灭性的。造成这个结果理由已经说过了：一旦所有 User Thread 离开了，虚拟机也就退出运行了。

【参考：http://blog.csdn.net/shimiso/article/details/8964414】

（八）线程同步

锁提供了两种主要特性：**互斥（mutual exclusion）** 和**可见性（visibility）**。互斥即一次只允许一个线程持有某个特定的锁，因此可使用该特性实现对共享数据的协调访问协议，这样，一次就只有一个线程能够使用该共享数据。可见性要更加复杂一些，它必须确保释放锁之前对共享数据做出的更改对于随后获得该锁的另一个线程是可见的 —— 如果没有同步机制提供的这种可见性保证，线程看到的共享变量可能是修改前的值或不一致的值，这将引发许多严重问题。

在 Java 中,为了保证多线程读写数据时保证数据的一致性,可以采用两种方式：

第一种，**同步（如用 synchronized 关键字，或者使用锁对象）**。比较常见和为我们所熟悉，不再赘述；

第二种，**使用 volatile 关键字**。用一句话概括 volatile，**它能够使变量在值发生改变时能尽快地让其他线程知道**。

关于 Volatile，首先我们要先意识到有这样的现象，编译器为了加快程序运行的速度，对一些变量的写操作会先在寄存器或者是 CPU 缓存上进行，最后才写入内存。而在这个过程，变量的新值对其他线程是不可见的。而 volatile 的作用就是使它修饰的变量的读写操作都必须在内存中进行!

volatile 与 synchronized 的区别：

> **①volatile 本质是在告诉 jvm 当前变量在寄存器中的值是不确定的，需要从主存中读取；synchronized 则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住；**
>
> **②volatile 仅能使用在变量级别，synchronized 则可以使用在变量，方法；**
>
> **③volatile 仅能实现变量的修改可见性，但不具备原子特性，而 synchronized 则可以保证变量的修改可见性和原子性；**
>
> **④volatile 不会造成线程的阻塞，而 synchronized 可能会造成线程的阻塞；**
>
> **⑤volatile 标记的变量不会被编译器优化，而 synchronized 标记的变量可以被编译器优化。**

因此，在使用 volatile 关键字时要慎重，并不是只要简单类型变量使用 volatile 修饰，对这个变量的所有操作都是原来操作，**当变量的值由自身的上一个决定时，如 n=n+1、n++ 等，volatile 关键字将失效**，只有当变量的值和自身上一个值无关时对该变量的操作才是**原子级别**的，如 n = m + 1，这个就是原级别的。所以在使用 volatile 关键时一定要谨慎，如果自己没有把握，可以使用 synchronized 来代替 volatile。

总结：volatile 本质是在告诉 JVM 当前变量在寄存器中的值是不确定的，需要从主存中读取。可以实现 synchronized 的部分效果，但当 n=n+1,n++等时，volatile 关键字将失效，不能起到像 synchronized 一样的线程同步的效果。

【参考：http://blog.csdn.net/fanaticism1/article/details/9966163】

（九）线程安全

1.相关定义

线程安全：就是多线程访问时，采用了加锁机制，当一个线程访问该类的某个数据时，

进行保护，其他线程不能进行访问直到该线程读取完，其他线程才可使用。不会出现数据不一致或者数据污染。

线程不安全：就是不提供数据访问保护，有可能出现多个线程先后更改数据造成所得到的数据是脏数据。

堆中的对象（类变量）需要考虑线程并发安全；栈中的对象（函数变量）不需要考虑。

原子操作的描述是： 多个线程执行一个操作时，其中任何一个线程要么完全执行完此操作，要么没有执行此操作的任何步骤。

2.举例说明：

```
public Double pi() {
    int a = 22;
    int b = 7;
    return new Double(a / b);
}
```

现在在执行这个方法时，每一个线程都有自己的独立的栈区。当线程进入到方法执行断的时候，一个方法变量在方法代码段中被创建，并保存在线程的栈区(静态方法也放在这里)。不同线程执行这段代码时，会有不同的 a/b 变量。所以这里是线程安全的，因为没有数据共享。

```
    private Double pi2 = null;
    public Double pi2() {
        if (pi2 == null) {
            pi2 = new Double(22 / 7);
        }
        return pi2;
    }
```

这个地方虽然优化了，但可惜他不是线程安全的。 因为两个线程并发执行的时候同时进入到 pi==null 这个位置，这样可能会 new 出一个脏的数据. 。

```
    private static ThreadLocal<Double> pi3 = new ThreadLocal<Double>();
    public Double pi3() {
        if (pi3.get() == null) {
            pi3.set(new Double(22 / 7));
        }
        return (Double) pi3.get();
    }
```

ThreadLocal 类封装了任何类型对象，并把它绑定到当前线程。线程执行 pi()方法的时候，实例 pi 返回的是当前线程的对象。这样的调用是线程安全的。（关于 ThreadLocal 的文章请参看：http://blog.csdn.net/qq296398300/article/details/53587299）

3.常用对象分类

①提供同步机制的工具类：StringBuffer、Hashtable、Vector；

②不提供同步机制的工具类：StringBuilder、Hashmap、ArrayList；

（关于 ArrayList、Vector、Hashmap 等区别的文章，请参看如下博客：

http://blog.csdn.net/qq296398300/article/details/53082686）

③HashMap 应用于多线程场合的方法：**Collections.synchronizedMap()**、Synchronized 关键字、**ConcurrentHashMap**。

（十）线程间协作

1.wait（）、notify（）和 notifyAll（）源码

```
/**
 * Wakes up a single thread that is waiting on this object's
 * monitor. If any threads are waiting on this object, one of them
 * is chosen to be awakened. The choice is arbitrary and occurs at
 * the discretion of the implementation. A thread waits on an object's
 * monitor by calling one of the wait methods
 */
public final native void notify();

/**
 * Wakes up all threads that are waiting on this object's monitor. A
 * thread waits on an object's monitor by calling one of the
 * wait methods.
 */
public final native void notifyAll();

/**
 * Causes the current thread to wait until either another thread invokes
the
 * {@link java.lang.Object#notify()} method or the
 * {@link java.lang.Object#notifyAll()} method for this object, or a
 * specified amount of time has elapsed.
 * <p>
 * The current thread must own this object's monitor.
 */
public final native void wait(long timeout) throws InterruptedException;
```

从这三个方法的文字描述可以知道以下几点信息：

1）wait()、notify()和 notifyAll()方法是本地方法，并且为 final 方法，无法被重写；

2）调用某个对象的 wait()方法能让当前线程阻塞，并且当前线程必须拥有此对象的 monitor（即锁）；

3）调用某个对象的 notify()方法能够唤醒一个正在等待这个对象的 monitor 的线程，如果有多个线程都在等待这个对象的 monitor，则只能唤醒其中一个线程；

4）调用 notifyAll()方法能够唤醒所有正在等待这个对象的 monitor 的线程；

有朋友可能会有疑问：为何这三个不是 Thread 类声明中的方法，而是 Object 类中声明的方法（当然由于 Thread 类继承了 Object 类，所以 Thread 也可以调用者三个方法）？其实这个问题很简单，由于每个对象都拥有 monitor（即锁），所以让当前线程等待某个对象的锁，当然应该通过这个对象来操作了。而不是用当前线程来操作，因为当前线程可能会等待多个线程的锁，如果通过线程来操作，就非常复杂了。

上面已经提到，如果调用某个对象的 wait()方法，当前线程必须拥有这个对象的 monitor（即锁），因此调用 wait()方法必须在同步块或者同步方法中进行（synchronized 块或者 synchronized 方法）。

调用某个对象的 wait()方法，相当于让当前线程交出此对象的 monitor，然后进入等待状态，等待后续再次获得此对象的锁（Thread 类中的 sleep 方法使当前线程暂停执行一段时间，从而让其他线程有机会继续执行，但它并不释放对象锁）；

notify()方法能够唤醒一个正在等待该对象的 monitor 的线程，当有多个线程都在等待该对象的 monitor 的话，则只能唤醒其中一个线程，具体唤醒哪个线程则不得而知。

同样地，调用某个对象的 notify()方法，当前线程也必须拥有这个对象的 monitor，因此调用 notify()方法必须在同步块或者同步方法中进行(synchronized 块或者 synchronized 方法）。

nofityAll()方法能够唤醒所有正在等待该对象的 monitor 的线程，这一点与 notify()方法是不同的。

这里要注意一点：notify()和 notifyAll()方法只是唤醒等待该对象的 monitor 的线程，并不决定哪个线程能够获取到 monitor。

举个简单的例子：假如有三个线程 Thread1、Thread2 和 Thread3 都在等待对象 objectA 的monitor，此时 Thread4 拥有对象 objectA 的monitor，当在 Thread4 中调用 objectA.notify()方法之后，Thread1、Thread2 和 Thread3 只有一个能被唤醒。注意，被唤醒不等于立刻就获取了 objectA 的 monitor。假若在 Thread4 中调用 objectA.notifyAll()方法，则 Thread1、

Thread2 和 Thread3 三个线程都会被唤醒，至于哪个线程接下来能够获取到 objectA 的 monitor 就具体依赖于操作系统的调度了。

上面尤其要注意一点，一个线程被唤醒不代表立即获取了对象的 monitor，只有等调用完 notify()或者 notifyAll()并退出 synchronized 块，释放对象锁后，其余线程才可获得锁执行。

下面看一个例子就明白了：

```
public class Test {
    public static Object object = new Object();
    public static void main(String[] args) {
        Thread1 thread1 = new Thread1();
        Thread2 thread2 = new Thread2();
        thread1.start();
        try {
            Thread.sleep(200);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        thread2.start();
    }

    static class Thread1 extends Thread{
        @Override
        public void run() {
            synchronized (object) {
                try {
                    object.wait();
                } catch (InterruptedException e) {
                }
                System.out.println(Thread.currentThread().getName()+"获取到了锁");
            }
        }
    }

    static class Thread2 extends Thread{
        @Override
        public void run() {
            synchronized (object) {
                object.notify();
                System.out.println(Thread.currentThread().getName()+"调用了object.notify()");
            }
```

```
            System.out.println(Thread.currentThread().getName()+"释放了锁
");
        }
    }
}
```

无论运行多少次，得到的都是：

```
线程 Thread-1 调用了 object.notify()
线程 Thread-1 释放了锁
线程 Thread-0 获取到了锁
```

2.Condition

Condition 是在 java 1.5 中才出现的，它用来替代传统的 Object 的 wait()、notify()实现线程间的协作，相比使用 Object 的 wait()、notify()，使用 Condition1 的 await()、signal()这种方式实现线程间协作更加安全和高效。因此通常来说比较推荐使用 Condition，在阻塞队列那一篇博文中就讲述到了，阻塞队列实际上是使用了 Condition 来模拟线程间协作。

（1）Condition 是个接口，基本的方法就是 await()和 signal()方法；

（2）Condition 依赖于 Lock 接口，生成一个 Condition 的基本代码是 lock.newCondition()

（3）调用 Condition 的 await()和 signal()方法，都必须在 lock 保护之内，就是说必须在 lock.lock()和 lock.unlock 之间才可以使用；

对应关系：

Conditon 中的 await()对应 Object 的 wait()；

Condition 中的 signal()对应 Object 的 notify()；

Condition 中的 signalAll()对应 Object 的 notifyAll()。

3.生产者-消费者模型的实现

（1）使用 Object 的 wait()和 notify()实现

```
public class Test {
    private int queueSize = 10;
    private PriorityQueue<Integer> queue = new PriorityQueue<Integer>(qu
eueSize);

    public static void main(String[] args) {
        Test test = new Test();
        Producer producer = test.new Producer();
        Consumer consumer = test.new Consumer();

        producer.start();
        consumer.start();
    }
```

```
    class Consumer extends Thread {

        @Override
        public void run() {
            consume();
        }

        private void consume() {
            while (true) {
                synchronized (queue) {
                    while (queue.size() == 0) {
                        try {
                            System.out.println("队列空，等待数据");
                            queue.wait();
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                            queue.notify();
                        }
                    }
                    queue.poll(); // 每次移走队首元素
                    queue.notify();
                    System.out.println("从队列取走一个元素，队列剩余" +
queue.size() + "个元素");
                }
            }
        }
    }

    class Producer extends Thread {

        @Override
        public void run() {
            produce();
        }

        private void produce() {
            while (true) {
                synchronized (queue) {
                    while (queue.size() == queueSize) {
                        try {
                            System.out.println("队列满，等待有空余空间");
                            queue.wait();
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                            queue.notify();
                        }
```

```
                }
                queue.offer(1); // 每次插入一个元素
                queue.notify();
                System.out.println("向队列取中插入一个元素，队列剩余空间："
                        + (queueSize - queue.size()));
            }
        }
    }
}
```

（2）使用 Condition 实现

```
public class Test {
    private int queueSize = 10;
    private PriorityQueue<Integer> queue = new PriorityQueue<Integer>(qu
eueSize);
    private Lock lock = new ReentrantLock();
    private Condition notFull = lock.newCondition();
    private Condition notEmpty = lock.newCondition();

    public static void main(String[] args)  {
        Test test = new Test();
        Producer producer = test.new Producer();
        Consumer consumer = test.new Consumer();

        producer.start();
        consumer.start();
    }

    class Consumer extends Thread{

        @Override
        public void run() {
            consume();
        }

        private void consume() {
            while(true){
                lock.lock();
                try {
                    while(queue.size() == 0){
                        try {
                            System.out.println("队列空，等待数据");
                            notEmpty.await();
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
```

```
                }
                queue.poll();                //每次移走队首元素
                notFull.signal();
                System.out.println("从队列取走一个元素，队列剩余"+queue.siz
e()+"个元素");
            } finally{
                lock.unlock();
            }
        }
    }

    class Producer extends Thread{

        @Override
        public void run() {
            produce();
        }

        private void produce() {
            while(true){
                lock.lock();
                try {
                    while(queue.size() == queueSize){
                        try {
                            System.out.println("队列满，等待有空余空间");
                            notFull.await();
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }
                    queue.offer(1);        //每次插入一个元素
                    notEmpty.signal();
                    System.out.println("向队列取中插入一个元素，队列剩余空间："+(q
ueueSize-queue.size()));
                } finally{
                    lock.unlock();
                }
            }
        }
    }
}
```

（十一）java.util.concurrent(J.U.C)

又是很大的一块，可以先详见《Java 并发工具包 java.util.concurrent 用户指南》。博客地址：http://blog.csdn.net/qq296398300/article/details/53884585