发件人: 方堃 fangkun119@hotmail.com
主题:
日期: 2021年3月4日 下午1:59
收件人:

CH21 多线程(C): 线程协作、生产者消费者、BlockingQueue、线程管道、死锁

2016年10月1日 星期六
16:55

目录

1. 线程协作机制（等待-通知机制）：wait(),notify(),notifyAll() 【目录】

解决的问题	等待-通知机制：线程A将自身挂起（而不是轮询忙等），条件具备时外部（例如线程B）向A发起通知，告知线程A外部已经发生变化		
机制	Java SE6	Object对象的wait(),notify(),notifyAll()方法	
	Java SE5	Condition对象的await()，signal()方法	
wait() notifyAll()	要点	与互斥锁的关系	调用wait()时，线程挂起，同时将Object的隐式锁释放（此时其他线程可以执行synchronized方法或进入临界区）
		等待时长	可选指定wait的最大时长：这样wait既可以被notify()，notifyAll()唤醒，也可以因超时而唤醒 也可以不指定最大时长：这样只能被notify(),notifyAll()唤醒
		使用限制	必须在synchronized函数或临界区中调用 wait()/notify()/notifyAll()：因为调用时隐式对象锁要被加好 类可以不实现Runnable接口 如果不在synchronized函数或临界区调用：程序可编译通过，运行时抛IlleagalMonitorStateException
		cond_flag辅助	存在多个线程同时被唤醒的情况，对wait()的调用还要包裹在while(test conf_flag){}中，参考 多个线程唤醒
	用法	对象A调用wait()	对象B通知对象A

		class A {
		...
		synchronized void func_with_wait() {
		wait();
		}
		}
			class B {
			...
			void send_notify_All(A a) {
			synchronized(a) {
			a.notifyAll()
			}
			}
			}

	替代	也可以用基于显式锁的Condition机制来替代，代码更复杂，但支持场景更细化	
例子	概述	线程A	打蜡(waxed)
		线程B	抛光(buffing)
		操作流程	打蜡-->抛光-->另一层打蜡-->抛光
	多个线程唤醒	用while(test cond_flag){}把wait()包裹起来，cond_flag要用互斥机制保护好，因为: (1) 唤醒多个线程时，其他线程可能先被唤醒，改变了这个flag的值 (2) 唤醒多个线程时，其他线程可能后被唤醒，需要检查最新状况，看是否应该修改flag的值 (3) 如果不用互斥机制将flag保护好，还会引发"错失信号的bug":wait()调用之前notify()已经结束,导致无法再等到新的notify()线程永远阻塞	
		前面的示例强调你必须用一个检查感兴趣的条件的**while**循环包围**wait()**。这很重要，因为： • 你可能有多个任务出于相同的原因在等待同一个锁，而第一个唤醒任务可能会改变这种状况（即使你没有这么做，有人也会通过继承你的类去这么做）。如果属于这种情况，那么这个任务应该被再次挂起，直至其感兴趣的条件发生变化。 • 在这个任务从其**wait()**中被唤醒的时刻，有可能会有某个其他的任务已经做出了改变，从而使得这个任务在此时不能执行，或者执行其操作已显得无关紧要。此时，应该通过再次调用**wait()**来将其重新挂起。 • 也有可能某些任务出于不同的原因在等待你的对象上的锁（在这种情况下必须使用**notifyAll()**）。在这种情况下，你需要检查是否已经由正确的原因唤醒，如果不是，就再次调用**wait()**。	

	汽车	package concurrency.waxomatic;
		import java.util.concurrent.*;
		import static net.mindview.util.Print.*;
		class Car {
		private boolean waxOn = false;
		public synchronized void waxed() {
		waxOn = true; // Ready to buff
		notifyAll();
			import java.util.concurrent.*;
			class Car {
			print boolean waxOn = false;
			//wait(),notify(),notifyAll()
			//调用notifyAll前先加好对象锁

<table>
<tr><td></td><td></td><td>

```
}
public synchronized void buffed() {   抛光
 waxOn = false; //Ready for another coat of wax
  notifyAll();
}
public synchronized void waitForWaxing()
throws InterruptedException {
  while(waxOn == false)
    wait();
}
public synchronized void waitForBuffing()
throws InterruptedException {
  while(waxOn == true)
    wait();
}
}
```

</td><td>

```
public synchronized void waxed() {
 waxOn = true; //蜡已经打好了,可以抛光了
 notifyAll(); //通知阻塞线程的同时解锁
}
...
//调用wait()前先加好对象锁.另外wait会抛异常
public synchronized void waitForWaxing()
throw InterruptedException {
 //wait包裹在while(test cond_flag){}中
 //以处理多个线程被唤醒的情况
 //也需要放在互斥机制中保护,防止并发修改
 while (waxOn == false)
   wait();
}
...
```

</td></tr>
<tr><td></td><td>打蜡
线程</td><td>

```
class WaxOn implements Runnable {
  private Car car;
  public WaxOn(Car c) { car = c; }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        printnb("Wax On! ");
        TimeUnit.MILLISECONDS.sleep(200);
        car.waxed();
        car.waitForBuffing();
      }
    } catch(InterruptedException e) {
      print("Exiting via interrupt");
    }
    print("Ending Wax On task");
  }
}
```

</td><td>

```
//打蜡线程:打蜡->通知抛光->等待抛光完成
class WaxOn implements Runnable {
 //从构造函数传入,与抛光线程共享同一个Car对象
 private Car car;
 public WaxOn(Car c) {car = c; };
 //线程逻辑
 public void run() {
  try { //wait()会抛InterruptedException
    while(!Thread.ingterrupted()) {
      ...
      car.waxed(); //完成打蜡通知抛光线程
      car.waitForBuffring(); //等待抛光线程
    }
  } catch (InterruptedException e) {
    //中断作为线程退出机制,跳出while循环
    ...
  }
}
```

</td></tr>
<tr><td></td><td>抛光
线程</td><td>

```
class WaxOff implements Runnable {
  private Car car;
  public WaxOff(Car c) { car = c; }
  public void run() {
    try {
      while(!Thread.interrupted()) {

        car.waitForWaxing();
        printnb("Wax Off! ");
        TimeUnit.MILLISECONDS.sleep(200);
        car.buffed();
      }
    } catch(InterruptedException e) {
      print("Exiting via interrupt");
    }
    print("Ending Wax Off task");
  }
}
```

</td><td>与上面类似</td></tr>
<tr><td></td><td>main</td><td>

```
public class WaxOMatic {
  public static void main(String[] args) throws Exception {
    Car car = new Car();
    ExecutorService exec = Executors.newCachedThreadPool();
    exec.execute(new WaxOff(car));
    exec.execute(new WaxOn(car));
    TimeUnit.SECONDS.sleep(5); // Run for a while...
    exec.shutdownNow(); // Interrupt all tasks
  }
} /* Output: (95% match)
Wax On! Wax Off! Wax On! Wax Off! Wax On! Wax Off! Wax On!
Wax Off! Wax On! Wax Off! Wax On! Wax Off! Wax On! Wax Off!
Wax On! Wax Off! Wax On! Wax Off! Wax On! Wax Off! Wax On!
Wax Off! Wax On! Wax Off! Wax On! Exiting via interrupt
Ending Wax On task
Exiting via interrupt
Ending Wax Off task
*///:~
```

</td><td>

```
//两个线程共享的资源
Car car = new Car();
//启动两个线程
ExecugtorService exec = Executors.newCachedThread();
exec.execute(new WaxOff(car));
exec.execute(new Waxn(car));
//发送中断信号让两个线程退出
exec.shudownNow();
```

</td></tr>
<tr><td>错失notify的bug</td><td colspan="3">
<table>
<tr><td>错失notify</td><td colspan="2">线程不能兼容notify()过早发送的情况，调用wait()在notify之后时，错过notify，永远被阻塞</td></tr>
<tr><td>原因</td><td colspan="2">没有用conf_flag做double_check，获得conf_flag没有被对象锁保护</td></tr>
<tr><td>错误方式</td><td>

```
T1:
synchronized(sharedMonitor) {
  <setup condition for T2>
  sharedMonitor.notify();
}

T2:
while(someCondition) {
  // Point 1
  synchronized(sharedMonitor) {
    sharedMonitor.wait();
  }
}
```

</td><td>触发Bug的调度顺序:
[01] T1: synchronized(sharedMonitor) #进入临界区
[02] T2: while(cond_flag) #此时conf_flag容许T2 wait()
[03] T1: setup cond_flag #T1读不到
[04] T1: notify() #T2还没调用wait(),notify丢失
[05] T1: 离开临界区，解锁
[06] T2: synchronized(sharedMonitor) #进入临界区
[07] T2: sharedMonitor.wait() #notify已丢失,永远阻塞

根源: [02] T2检查con_flag没有被临界区保护</td></tr>
<tr><td>正确方式</td><td>用互斥把someCondition保护好
T1:保持不变

```
synchronized(sharedMonitor) {
  <setup condition for T2>
  sharedMonitor.notify();
}
```

</td><td>形象一些：cond_flag重命名为t1_done
T1先进入临界区时：
<table>
<tr><td>T1</td><td>初始值t1_done为false
耗时操作
加锁->设置t1_done为true->notify()->解锁</td></tr>
</table>
</td></tr>
</table>
</td></tr>
</table>

T2:修改一下

```
synchronized(sharedMonitor) {
  while(someCondition)
    sharedMonitor.wait();
}
```

T2	加锁->读到t1_done为true->跳过while循环->解锁

T2先进入临界区:

T1	初始值 t1_done = false 耗时操作
T2	加锁->读到t1_done为false->wait()并自动解锁
T1	加锁->设置t1_done为true->notify()并自动解锁
T2	收到notify并自动加锁->处理notify->解锁

notify()与notifyAll()区别

区别

多个任务(线程)在wait()同一个条件时

notify()	更加优化	只唤醒wait()在共享资源上的一个任务（注意饥饿现象）
notifyAll()	更加安全	会唤醒wait()在共享资源上的所有任务（不影响wait()其他资源的任务，如例子）

例子

资源

```
import java.util.concurrent.*;
import java.util.*;

class Blocker {
  synchronized void waitingCall() {
    try {
      while(!Thread.interrupted()) {
        wait();
        System.out.print(Thread.currentThread() + " ");
      }
    } catch(InterruptedException e) {
      // OK to exit this way
    }
  }
  synchronized void prod() { notify(); }
  synchronized void prodAll() { notifyAll(); }
}
```

//简单演示,没有添加conf_flag等机制

```
synchronized void waitingCall() {
  try {
    //通过中断来让线程结束运行
    while(!Thread.interrupted()) {
      wait(); //wait可被中断
    }
  } catch (InterruptedException e) {
    ...
  }
}
synchronized void notifyCall() { notify(); }
synchronized void notifyAllCall() {
  notifyAll();
}
```

任务1,2

```
class Task implements Runnable {
  static Blocker blocker = new Blocker();
  public void run() { blocker.waitingCall(); }
}

class Task2 implements Runnable {
  // A separate Blocker object.
  static Blocker blocker = new Blocker();
  public void run() { blocker.waitingCall(); }
}
```

//static Blocker blocker
所有的Task对象共享同一个blocker
所有的Task2对象共享同一个blocker
Task和Task2之间无资源共享关系

main

```
public class NotifyVsNotifyAll {
  public static void main(String[] args) throws Exception {
    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < 5; i++)
      exec.execute(new Task());
    exec.execute(new Task2());
    Timer timer = new Timer();
    timer.scheduleAtFixedRate(new TimerTask() {
      boolean prod = true;
      public void run() {
        if(prod) {
          System.out.print("\nnotify() ");          Timer先
          Task.blocker.prod();
          prod = false;
        } else {
          System.out.print("\nnotifyAll() ");
          Task.blocker.prodAll();
          prod = true;
        }
      }
    }, 400, 400); // Run every .4 second
    TimeUnit.SECONDS.sleep(5); // Run for a while...
    timer.cancel();
    System.out.println("\nTimer canceled");
    TimeUnit.MILLISECONDS.sleep(500);
    System.out.print("Task2.blocker.prodAll() ");
    Task2.blocker.prodAll();          timer线程未位，再notifyAll -
    TimeUnit.MILLISECONDS.sleep(500);
    System.out.println("\nShutting down");
    exec.shutdownNow(); // Interrupt all tasks
  }
} /* Output: (Sample)
```

```
//启动5个Task线程，1个Task2线程
ExecutorService
    exec = Executoirs.newCachedThreadPool();
for (int i = 0; i < 5; ++i)
  exec.execute(new Task());
exec.execute(new Task2());
//启动定时任务，定期执行任务，任务用匿名内部类实现
Timer timer = new Timer();
Timer,.scheduleAtFixedRate(new TimerTask() {
  boolean notifySingleThread = true;
  if (notifySingleThread) {
    Task.blocker.notifyCall(); //只唤醒一个线程
    notifySingleThread = false;//下次唤醒所有
  } else {
    Task.blocker.notifyAllCall();//唤醒所有线程
    notifySingleThread = true;//下次只唤醒一个线程
  }, 400, 400);//每4秒钟运行一次
//主线程sleep 5秒钟，让定时任务运行1次(?2次)
TimeUnit.SECONDS.seleep(5);
//停止定时任务
timer.cancel();
...
```

输出

```
notify() Thread[pool-1-thread-1,5,main]
notifyAll() Thread[pool-1-thread-1,5,main] Thread[pool-1-
thread-5,5,main] Thread[pool-1-thread-4,5,main]
Thread[pool-1-thread-3,5,main] Thread[pool-1-thread-
2,5,main]
notify() Thread[pool-1-thread-1,5,main]
notifyAll() Thread[pool-1-thread-1,5,main] Thread[pool-1-
thread-2,5,main] Thread[pool-1-thread-3,5,main]
Thread[pool-1-thread-4,5,main] Thread[pool-1-thread-
5,5,main]
notify() Thread[pool-1-thread-1,5,main]
notifyAll() Thread[pool-1-thread-1,5,main] Thread[pool-1-
thread-5,5,main] Thread[pool-1-thread-4,5,main]
Thread[pool-1-thread-3,5,main] Thread[pool-1-thread-
2,5,main]
notify() Thread[pool-1-thread-1,5,main]
notifyAll() Thread[pool-1-thread-1,5,main] Thread[pool-1-
thread-2,5,main] Thread[pool-1-thread-3,5,main]
Thread[pool-1-thread-4,5,main] Thread[pool-1-thread-
5,5,main]
notify() Thread[pool-1-thread-1,5,main]
notifyAll() Thread[pool-1-thread-1,5,main] Thread[pool-1-
thread-5,5,main] Thread[pool-1-thread-4,5,main]
Thread[pool-1-thread-3,5,main] Thread[pool-1-thread-
2,5,main]
notify() Thread[pool-1-thread-1,5,main]
notifyAll() Thread[pool-1-thread-1,5,main] Thread[pool-1-
thread-2,5,main] Thread[pool-1-thread-3,5,main]
Thread[pool-1-thread-4,5,main] Thread[pool-1-thread-
5,5,main]
Timer canceled
Task2.blocker.prodAll() Thread[pool-1-thread-6,5,main]
```

输出内容:
notify()：Thread 1被唤醒
notifyAll()：Thread 1,5,4,3,2 被唤醒
notify()：Thread 1被唤醒
notifyAll()：Thread 1,2,3,4,5 被唤醒
notify()：Thread 1被唤醒
notifyAll()：Thread 1,5,4,3,2 被唤醒
notify()：Thread 1被唤醒
notifyAll()：Thread 1,2,3,4,5 被唤醒
notify()：Thread 1被唤醒
notifyAll()：Thread 1,5,4,3,2 被唤醒
notify()：Thread 1被唤醒

notifyAll()：Thread 1,2,3,4,5 被唤醒
另一处资源上notifyAll()：Thread 6被唤醒

从上面输出中看出，notify有引发饥饿的风险，
而notifyAll则同时唤醒多个线程再让他们争抢互斥锁

```
Shutting down
*///:~
```

2. 单生产者单消费者 【目录】

<table>
<tr><td>要点</td><td>

```
synchronized(this){
  while (!tc meetCondition)
    wait();
}
```

</td><td>(1)防止在while(...)检查meetCondition前，另一个线程改变互斥锁的状态
(2)用while(!meetCondition)避免不必要的wait()
(3)wait()只能在获得隐式对象锁的前提下调用，调用时会释放隐式对象锁</td></tr>
<tr><td>例子</td><td colspan="2">1 生产者、1消费者：厨师代表生产者、服务员代表消费者、生产者消费者共享Restaunt.meal
<table>
<tr><td>消费者</td><td>

```
class WaitPerson implements Runnable {
  private Restaurant restaurant;
  public WaitPerson(Restaurant r) { restaurant = r; }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        synchronized(this) {
          while(restaurant.meal == null)
            wait(); // ... for the chef to produce a meal
        }
        print("Waitperson got " + restaurant.meal);
        synchronized(restaurant.chef) {
          restaurant.meal = null;
          restaurant.chef.notifyAll(); // Ready for another
        }
      }
    } catch(InterruptedException e) {
      print("WaitPerson interrupted");
    }
  }
}
```

</td><td>

```
//restaurant为消费者、生产者共享，通过构造函数传给消费者、生产者
public void run() {
  try {
    //线程通过中断机制结束
    while(!Thread.interrupted()) {
      //条件变量用法:锁this->while(false==flag)->wait()
      synchronized(this) {
        while(null == restaurant.meal)
          wait();
      }
      //条件变量用法:锁peer->set(flag,false)->notifyAll()
      synchronized(other) {
        restaurant.meal = null;
        restaurant.chef.notifyAll();
      }
    }
  } catch(InterruptedException e) {
    //wait()被中断时会抛exception
    ...
  }
}
```

</td></tr>
<tr><td>生产者</td><td>

```
class Chef implements Runnable {
  private Restaurant restaurant;
  private int count = 0;
  public Chef(Restaurant r) { restaurant = r; }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        synchronized(this) {
          while(restaurant.meal != null)
            wait(); // ... for the meal to be taken
        }
        if(++count == 10) {
          print("Out of food, closing");
          restaurant.exec.shutdownNow();
        }
        printnb("Order up! ");
        synchronized(restaurant.waitPerson) {
          restaurant.meal = new Meal(count);
          restaurant.waitPerson.notifyAll();
        }
        TimeUnit.MILLISECONDS.sleep(100);
      }
    } catch(InterruptedException e) {
      print("Chef interrupted");
    }
  }
}
```

</td><td>

```
//restaurant为消费者、生产者共享，通过构造函数传给消费者、生产者
public void run() {
  try {
    //线程通过中断机制结束
    while (!Thread.interrupted()) {
      //条件变量用法:锁this->while(true==flag)->wait()
      synchronized(this) {
        while(null != restaurant.meal)
          wait();
      }
      //用ExecutorService#shutdownNow()向所有线程发中断信号
      if (++count == 10) {
        restaurant.exec.shutdownNow();
      }
      //条件变量用法:锁peer->set(flag,true)->notifyAll()
      synchronized(restaurant.waitPerson) {
        restaurant.meal = new Meal(count);
        restaurant.waitPerson.notifyAll();
      }
      TimeUnit.MILLISECONDS.sleep(100);
    }
  } catch (InterruptedException e) {
    print("Chef interrupted");
  }
}
```

</td></tr>
<tr><td>Meel</td><td>

```
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

class Meal {
  private final int orderNum;
  public Meal(int orderNum) { this.orderNum = orderNum; }
  public String toString() { return "Meal " + orderNum; }
}
```

</td><td>OrderNum订单号，唯一标识一个meal
在程序中用作cond_flag</td></tr>
<tr><td>Restaunt
main</td><td>

```
public class Restaurant {
  Meal meal;
  ExecutorService exec = Executors.newCachedThreadPool();
  WaitPerson waitPerson = new WaitPerson(this);
  Chef chef = new Chef(this);
  public Restaurant() {
    exec.execute(chef);
    exec.execute(waitPerson);
  }
  public static void main(String[] args) {
    new Restaurant();
  }
} /* Output:
```

</td><td>启动两个线程

```
Order up! Waitperson got Meal 1
Order up! Waitperson got Meal 2
Order up! Waitperson got Meal 3
Order up! Waitperson got Meal 4
Order up! Waitperson got Meal 5
Order up! Waitperson got Meal 6
Order up! Waitperson got Meal 7
Order up! Waitperson got Meal 8
Order up! Waitperson got Meal 9
Out of food, closing
WaitPerson interrupted
Order up! Chef interrupted
*///:~
```

</td></tr>
</table>
</td></tr>
</table>

3. 显示地Lock和Condition对象 【目录】

<table>
<tr><td>机制</td><td>原理</td><td>与wait()/notify()/notifyAll()相同
但是用的API不同，锁也是显式的，调用API之前要显式地加好锁（类似Linux C的条件变量）</td></tr>
<tr><td></td><td>互斥锁</td><td>
<table>
<tr><td>类</td><td>Lock lock = new ReentrantLock();</td></tr>
<tr><td>其他说明</td><td>ReentrantLock提供可被中断的互斥api</td></tr>
</table>
</td></tr>
</table>

	条件变量	类: Condition condition = lock.newCondition(); //用lock的函数生成,这样await()/signal()/signalAll()调用时才能自动解锁
		API: await() signal() signalAll()

例子

重写之前的wait()/notify()/notifyAll()一节的例子

car

```
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
import static net.mindview.util.Print.*;

class Car {
  private Lock lock = new ReentrantLock();
  private Condition condition = lock.newCondition();
  private boolean waxOn = false;
  public void waxed() {
    lock.lock();
    try {
      waxOn = true; // Ready to buff
      condition.signalAll();
    } finally {
      lock.unlock();
    }
  }
  public void buffed() {
    lock.lock();
    try {
      waxOn = false; // Ready for another coat of wax
      condition.signalAll();
    } finally {
      lock.unlock();
    }
  }
  public void waitForWaxing() throws InterruptedException {
    lock.lock();
    try {
      while(waxOn == false)
        condition.await();
    } finally {
      lock.unlock();
    }
  }
  public void waitForBuffing() throws InterruptedException{
    lock.lock();
    try {
      while(waxOn == true)
        condition.await();
    } finally {
      lock.unlock();
    }
  }
}
```

打蜡，刨光，main

```
class WaxOn implements Runnable {
  private Car car;
  public WaxOn(Car c) { car = c; }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        printnb("Wax On! ");
        TimeUnit.MILLISECONDS.sleep(200);
        car.waxed();
        car.waitForBuffing();
      }
    } catch(InterruptedException e) {
      print("Exiting via interrupt");
    }
    print("Ending Wax On task");
  }
}
public void run() {
  try {
    while(!Thread.interrupted()) {
      car.waitForWaxing();
      printnb("Wax Off! ");
      TimeUnit.MILLISECONDS.sleep(200);
      car.buffed();
    }
  } catch(InterruptedException e) {
    print("Exiting via interrupt");
  }
  print("Ending Wax Off task");
}
}
public class WaxOMatic2 {
  public static void main(String[] args) throws Exception {
    Car car = new Car();
    ExecutorService exec = Executors.newCachedThreadPool();
    exec.execute(new WaxOff(car));
    exec.execute(new WaxOn(car));
    TimeUnit.SECONDS.sleep(5);
    exec.shutdownNow();
  }
} /* Output: (90% match)
Wax On! Wax Off! Wax On! Wax Off! Wax On! Wax Off! Wax On!
Wax Off! Wax On! Wax Off! Wax On! Wax Off! Wax On! Wax Off!
Wax On! Wax Off! Wax On! Wax Off! Wax On! Wax Off! Wax On!
Wax Off! Wax On! Wax Off! Wax On! Exiting via interrupt
Ending Wax Off task
Exiting via interrupt
Ending Wax On task
*///:~
```

4. 直接使用生产者消费者队列 【目录】

目标	wait()/notify()/noitfyAll()太底层,编写步骤比较繁琐 更高的抽象级别，使用同步队列来完成这类任务	
工具	接口	java.util.concurrent.BlockingQueue
	一些实现	LinkedBlockingQueue: 无界队列 ArrayBlockingQueue: 有固定尺寸，在被Blocking挂起之前可以放有限数量的元素 SynchronousQueue: 插入一个的时候，另一端才能取出一个，连capacity()为1都算不上因为不能peek()只有在插入时元素才在另一端可见
		队列为空时，从队列中取数据也会导致线程被挂起

例子

演示3中BlockingQueue的使用：LinkedBlockingQueue<>, ArrayBlockingQueue<>, SynchronousQueue<>

消费者

```
import java.util.concurrent.*;
import java.io.*;
import static net.mindview.util.Print.*;

class LiftOffRunner implements Runnable {
  private BlockingQueue<LiftOff> rockets;
  public LiftOffRunner(BlockingQueue<LiftOff> queue) {
    rockets = queue;
  }
  public void add(LiftOff lo) {
    try {
      rockets.put(lo);
    } catch(InterruptedException e) {
      print("Interrupted during put()");
    }
  }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        LiftOff rocket = rockets.take();
        rocket.run(); // Use this thread
      }
    } catch(InterruptedException e) {
      print("Waking from take()");
    }
    print("Exiting LiftOffRunner");
```

```
//消费者
class ConsumerRunnable implements Runnable {
 //与生产者共用一个BlockingQueue<>,由构造函数传入
 private BlockingQueue<Item> blockingQueue;
 public ConsumerRunnable(BlockingQueue<Item> queue) {
  blockingQueue = queue;
 }
 //供生产者调用的函数
 public void add(Item item) {
  //wait,notifyAll等都托管给blockingQueue
  //只要处理wait时可能会抛出的InterruptException即可
  try {
   blockingQueue.put(item);
  } catch (InterruptedException e) {
   ...
  }
 }
 //消费者线程
 public void run() {
  try {
   //中断作为线程退出机制
```

		}
		}
			while(!Thread.interrupted()) {
			//wait,notifyAll等都托管给blockingQueue
			//只要处理wait时可能会抛出的InterruptException即可
			Item item = blockingQueue.take();
			}
			} catch (InterruptedException e) {
			...
			}
			}
			}

	生产者	public class TestBlockingQueues {
		static void getkey() {
		try {
		// Compensate for Windows/Linux difference in the
		// length of the result produced by the Enter key:
		new BufferedReader(
		new InputStreamReader(System.in)).readLine();
		} catch(java.io.IOException e) {
		throw new RuntimeException(e);
		}
		}
		static void getkey(String message) {
		print(message);
		getkey();
		}
		static void
		test(String msg, BlockingQueue<LiftOff> queue) {
		print(msg);
		LiftOffRunner runner = new LiftOffRunner(queue);
		Thread t = new Thread(runner);
		t.start();
		for(int i = 0; i < 5; i++)
		runner.add(new LiftOff(5));
		getkey("Press 'Enter' (" + msg + ")");
		t.interrupt();
		print("Finished " + msg + " test");
		}
			//主线程: 兼任生产者和测试用例
			public class ProducerAndTester {
			//getKey函数只是为了和终端交互用
			static void getkey() {...};
			static void getKey(String message) {...};
			//测试用例
			static void test(
			String msg, BlockingQueue<Item> blockingQueue) {
			//启动消费者线程
			ConsumerRunnable consumerRunnable
			= new ConsumerRunnable(blockingQueue);
			Thread thread = new Thread(consumerRunnable);
			thread.start();
			//向消费者线程添加Item
			for (int i = 0; i < 5; ++i) {
			consumerRunnable.add(new Item(i));
			}
			//等待键盘敲回车，向消费者线程发中断信号，主线程退出
			...
			}
			}

	main	public static void main(String[] args) {
		test("LinkedBlockingQueue", // Unlimited size
		new LinkedBlockingQueue<LiftOff>());
		test("ArrayBlockingQueue", // Fixed size
		new ArrayBlockingQueue<LiftOff>(3));
		test("SynchronousQueue", // Size of 1
		new SynchronousQueue<LiftOff>());
		
			用3种队列做3个测试
			LinkedBlockingQueue<LiftOff>
			ArrayBlockingQueue<LiftOff>
			SynchronousQueue<LiftOff>

例子	土司制作机。一台机器具有三个任务：制作土司，给土司涂抹黄油，给涂抹好黄油的土司加果酱。三个任务有前后依赖关系。		

	土司及土司队列	//: concurrency/ToastOMatic.java
		// A toaster that uses queues.
		import java.util.concurrent.*;
		import java.util.*;
		import static net.mindview.util.Print.*;
		
		class Toast {
		public enum Status { DRY, BUTTERED, JAMMED }
		private Status status = Status.DRY;
		private final int id;
		public Toast(int idn) { id = idn; }
		public void butter() { status = Status.BUTTERED; }
		public void jam() { status = Status.JAMMED; }
		public Status getStatus() { return status; }
		public int getId() { return id; }
		public String toString() {
		return "Toast " + id + ": " + status;
		}
		}
		
		class ToastQueue extends LinkedBlockingQueue<Toast> {}
			id：土司的ID
			public Toast(int id) { this.id = it; }
			public int getId() { return id; };
			
			status: 土司的制作状态
			public enum Status { DRY, BUTTERED, JAMMED }
			private Status status = Status.DRY;
			public void butter() { status = Status.BUTTERED; }
			public void jam() { status = Status.JAMMED; }
			public Status getStatus() { return status; }
			
			ToastQueue: 继承LinkedBlockingQueue<Toast>,可处理阻塞/通知
			class ToastQueue extends LinkedBlockingQueue<Toast> {};

	土司制作线程	class Toaster implements Runnable {
		private ToastQueue toastQueue;
		private int count = 0;
		private Random rand = new Random(47);
		public Toaster(ToastQueue tq) { toastQueue = tq; }
		public void run() {
		try {
		while(!Thread.interrupted()) {
		TimeUnit.MILLISECONDS.sleep(
		100 + rand.nextInt(500));
		// Make toast
		Toast t = new Toast(count++);
		print(t);
		// Insert into queue
		toastQueue.put(t);
		}
		} catch(InterruptedException e) {
		print("Toaster interrupted");
		}
		print("Toaster off");
		}
		}
			//土司制作线程
			class Toaster implements Runnable {
			//自增计数器
			private int autoIncCounter = 0;
			//构造函数传入的BlockingQueue，与黄油线程共用
			private ToastQueue dryToastQueue;
			public Toaster(ToastQueue tq) {dryToastQueue = tq; };
			//线程
			public void run() {
			try {
			while(!Thread.interrupted()) {
			TimeUnit.MILLISECONDS.sleep(100);
			dryToastQueue.put(new Toast(autoIncCounter ++));
			}
			} catch (InterruptedException e) {
			...
			}
			}
			}

	涂抹黄油线程	// Apply butter to toast:
		class Butterer implements Runnable {
		private ToastQueue dryQueue, butteredQueue;
		public Butterer(ToastQueue dry, ToastQueue buttered) {
		dryQueue = dry;
		butteredQueue = buttered;
			//黄油线程
			class Butterer implements Runnable {
			//构造函数传入的BlockingQueue
			//dryToastQueue与土司线程共用
			//butteredToastQueue与果酱线程共用

	public void run() {
	try {
	while(!Thread.interrupted()) {
	// Blocks until next piece of toast is available:
	Toast t = dryQueue.take();
	t.butter();
	print(t);
	butteredQueue.put(t);
	}
	} catch(InterruptedException e) {
	print("Butterer interrupted");
	}
	print("Butterer off");
	}
	}
		private ToastQueue dryToastQueue;
		private ToastQueue butteredToastQueue;
		public Butterer(ToastQueue dry, ToastQueue buttered) {
		dryToastQueue = dry;
		butteredcToastQueue = buttered;
		}
		//线程
		public void run() {
		try {
		while(!Thread.interrupted()) {
		Toast toast = dryQueue.take();
		toast.butter();
		butteredQueue.put(toast);
		}
		} catch (InterruptedException e) {
		...
		}
		}
		}

果酱线程	// Apply jam to buttered toast:
	class Jammer implements Runnable {
	private ToastQueue butteredQueue, finishedQueue;
	public Jammer(ToastQueue buttered, ToastQueue finished) {
	butteredQueue = buttered;
	finishedQueue = finished;
	}
	public void run() {
	try {
	while(!Thread.interrupted()) {
	// Blocks until next piece of toast is available:
	Toast t = butteredQueue.take();
	t.jam();
	print(t);
	finishedQueue.put(t);
	}
	} catch(InterruptedException e) {
	print("Jammer interrupted");
	}
	print("Jammer off");
	}
	}
		与上面类似

顾客线程	// Consume the toast:
	class Eater implements Runnable {
	private ToastQueue finishedQueue;
	private int counter = 0;
	public Eater(ToastQueue finished) {
	finishedQueue = finished;
	}
	public void run() {
	try {
	while(!Thread.interrupted()) {
	// Blocks until next piece of toast is available:
	Toast t = finishedQueue.take();
	// Verify that the toast is coming in order,
	// and that all pieces are getting jammed:
	if(t.getId() != counter++ ||
	t.getStatus() != Toast.Status.JAMMED) {
	print(">>>> Error: " + t);
	System.exit(1);
	} else
	print("Chomp! " + t);
	}
	} catch(InterruptedException e) {
	print("Eater interrupted");
	}
	print("Eater off");
	}
	}
		与上面类似

main	public class ToastOMatic {
	public static void main(String[] args) throws Exception {
	ToastQueue dryQueue = new ToastQueue(),
	butteredQueue = new ToastQueue(),
	finishedQueue = new ToastQueue();
	ExecutorService exec = Executors.newCachedThreadPool();
	exec.execute(new Toaster(dryQueue));
	exec.execute(new Butterer(dryQueue, butteredQueue));
	exec.execute(new Jammer(butteredQueue, finishedQueue));
	exec.execute(new Eater(finishedQueue));
	TimeUnit.SECONDS.sleep(5);
	exec.shutdownNow();
	}
	} /* (Execute to see output) *///:~
		构造3个ToastQueue
		创建4个线程，配备合适的队列
		启动线程、sleep、然后退出

5. 线程任务间使用管道进行输入输出 【目录】

类	PipedWriter，PipedReader		
功能	为线程间的输入、输出提供支持，可以看做是BlockingQueue的变体		

例子	生产者	//: concurrency/PipedIO.java
		// Using pipes for inter-task I/O
		import java.util.concurrent.*;
		import java.io.*;
		import java.util.*;
		import static net.mindview.util.Print.*;
		
		class Sender implements Runnable {
		private Random rand = new Random(47);
		private PipedWriter out = new PipedWriter();
		public PipedWriter getPipedWriter() { return out; }
		public void run() {
		try {
		while(true)
		for(char c = 'A'; c <= 'z'; c++) {
		out.write(c);
		TimeUnit.MILLISECONDS.sleep(rand.nextInt(500));
		}
		} catch(IOException e) {
		print(e + " Sender write exception");
			Random rand： 用来随机sleep一段时间
			PipedWriter out : 管道写入端
			out.write(c)：写入管道、管道满时会阻塞
			需要捕捉的异常：
			IOException： out.write(c)会抛出
			InterruptedException: sleep()会抛出

	`} catch(InterruptedException e) {`
	`  print(e + " Sender sleep interrupted");`
	`  }`
	`}`	

消费者	`class Receiver implements Runnable {`
	`  private PipedReader in;`
	`  public Receiver(Sender sender) throws IOException {`
	`  in = new PipedReader(sender.getPipedWriter());`
	`  }`
	`  public void run() {`
	`  try {`
	`  while(true) {`
	`  // Blocks until characters are there:`
	`  printnb("Read: " + (char)in.read() + ", ");`
	`  }`
	`  } catch(IOException e) {`
	`  print(e + " Receiver read exception");`
	`  }`
	`  }`
	`}`
		PipedReader in：管道读取端，构造方式
		PipedReader(sender.getPipedWriter())保证构造出的PipedReader能与另一个线程的PipedWriter对应
		in.read()：从管道读取、管道为空时会阻塞
		需要捕捉的异常：
		IOException：in.read()会抛出

main	`public class PipedIO {`
	`  public static void main(String[] args) throws Exception {`
	`  Sender sender = new Sender();`
	`  Receiver receiver = new Receiver(sender);`
	`  ExecutorService exec = Executors.newCachedThreadPool();`
	`  exec.execute(sender);`
	`  exec.execute(receiver);`
	`  TimeUnit.SECONDS.sleep(4);`
	`  exec.shutdownNow();`
	`  }`
	`} /* Output: (65% match)`
	`Read: A, Read: B, Read: C, Read: D, Read: E, Read: F, Read: G, Read: H, Read: I, Read: J, Read: K, Read: L, Read: M,`
	`java.lang.InterruptedException: sleep interrupted Sender sleep interrupted`
	`java.io.InterruptedIOException Receiver read exception`
	`*///:~`
		• Sender：生产者线程的任务
		• Receiver：消费者线程的任务，构造方式保证讷讷个雨生产者配对
		• 启动线程，sleep，退出

6. 死锁问题 【目录】

哲学家就餐问题	略
产生死锁的四个条件	1. 存在互斥：有资源是不能被共享的 2. 至少有一个线程持有了一个资源、同时在等待另一个资源 3. 不能抢占资源，必须等占有资源的线程主动释放 4. 存在循环等待的现象
避免死锁	Java没有在语言层面上提供死锁避免机制，还需要在设计上把关