在进入正式话题之前，我们首先要了解三个方法。

- wait()方法

```
// 暂停线程
public final void wait() throws InterruptedException {
        wait(0);
}
```

- notify()方法

```
// 唤醒线程
public final native void notify();
```

这两个方法在生产者消费者模型中应用广泛，请大家务必掌握，具体可以参考:

> Java wait(),notify()与notifyAll()方法

1. 单一生产者消费者模型

生产者消费者模型是通过一个容器来解决生产者和消费者之间的强耦合问题。生产者和消费者之间不直接通讯，而时通过阻塞队列进行通讯。

生产者生产完数据不用等待消费者处理，而是直接扔给阻塞队列，消费者不直接找生产者要数据，而是直接从阻塞队列中取。阻塞队列在这里就相当于一个缓冲区，平衡了生产者和消费者的处理能力。

这个阻塞队列就是用来给生产者和消费者解耦的。

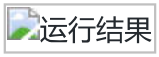
我们来看一个基础版的生产者消费者模型。

1.1 基础版

```
class Goods {
    // 产品名称
    private String goodsName;
    // 产品数量
    private int goodsNumber;

    public Goods(String goodsName, int goodsNumber) {
        super();
        this.goodsName = goodsName;
        this.goodsNumber = goodsNumber;
    }

    // 消费方法
    public synchronized void consumption() {
        this.goodsNumber--;
        System.out.println("消费 "+toString());
```

```
    }

    // 生产方法
    public synchronized void producation() {
        this.goodsNumber++;
        System.out.println("生产 "+toString());
    }

    @Override
    public String toString() {
        return "[goodsName:"+goodsName+", goodNumber: "+goodsNumber+"]";
    }

}

// 消费者类
class Consumers implements Runnable {
    private Goods goods;

    public Consumers(Goods goods) {
        super();
        this.goods = goods;
    }

    @Override
    public void run() {
        goods.consumption();
    }

}

// 生产者类
class Producers implements Runnable {
    private Goods goods;

    public Producers(Goods goods) {
        super();
        this.goods = goods;
    }

    @Override
    public void run() {
        goods.producation();
    }

}


public class Test {
    public static void main(String[] args) throws Exception {
        Goods goods = new Goods("demo", 0);
        Producers producers = new Producers(goods);
        Consumers consumers = new Consumers(goods);
        Thread producersThread = new Thread(producers);
        Thread consumersThread = new Thread(consumers);
        producersThread.start();
        Thread.sleep(2000);
        consumersThread.start();
    }
}
```

运行结果：



这里的 goods类 就相当于一个阻塞队列，生产者类和消费者类只管调用 goods类 中的生产消费方法都是同步方法，相互之间不进行之间联系。

goods类 中我们将要生产的产品名称传入，初始产品数量为0，并且生产消费方法只是进行单一生产消费，所以只会生产消费一次程序就会结束，接下来，我们把他完善一下。

```
class Goods {
    // 产品名称
    private String goodsName;
    // 产品数量
    private int goodsNumber;
    // 产品数量上限
    private int maxGoodsNumber;

    public Goods(String goodsName, int goodsNumber, int maxGoodsNumber) {
        super();
        this.goodsName = goodsName;
        this.goodsNumber = goodsNumber;
        this.maxGoodsNumber = maxGoodsNumber;
    }

    // 消费方法
    public void consumption() {
        // 循环消费
        while(true) {
            // 如果产品数量为0，停止消费，并退出线程
            if(this.goodsNumber < 1) {
                System.out.println("产品售罄");
                return;
            }
            this.goodsNumber--;
            System.out.println("消费 "+toString());
        }
    }

    // 生产方法
    public void producation() {
        // 循环生产
        while(true) {
            // 如果产品数量达到数量上限，退出生产
            if(this.goodsNumber >= this.maxGoodsNumber) {
                System.out.println("爆仓了");
                return;
            }
            this.goodsNumber++;
            System.out.println("生产 "+toString());
        }
    }

    @Override
    public String toString() {
        return "[goodsName:"+goodsName+", goodNumber: "+goodsNumber+"]";
    }
```

```
}

// 消费者类
class Consumers implements Runnable {
    private Goods goods;

    public Consumers(Goods goods) {
        super();
        this.goods = goods;
    }

    @Override
    public void run() {
        goods.consumption();
    }

}

// 生产者类
class Producers implements Runnable {
    private Goods goods;

    public Producers(Goods goods) {
        super();
        this.goods = goods;
    }

    @Override
    public void run() {
        goods.producation();
    }

}


public class Test {
    public static void main(String[] args) throws Exception {
        // 传入初始产品数量以及产品数量上限
        Goods goods = new Goods("demo", 0, 10);
        Producers producers = new Producers(goods);
        Consumers consumers = new Consumers(goods);
        Thread producersThread = new Thread(producers);
        Thread consumersThread = new Thread(consumers);
        producersThread.start();
        consumersThread.start();
    }
}
```

上面的程序虽然加入了完全消费和爆仓的场景，但是在生活中，产品被消费完了但是生产者还是在生产，这时消费者处在一个排队的状态，等生产者一旦产出，再进行继续消费。

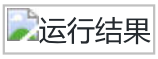
反之，商品虽然爆仓了，可是生产者可以暂停一会等待产品被消费掉然后继续生产。两者一直处在一种动态平衡的状态。如果要实现这种场景，我们就要使用到本文开头介绍的两种方法了。

1.2 wait(),notify()方法参与的生产者消费者模型

```
class Goods {
    // 产品名称
    private String goodsName;
    // 产品数量
    private int goodsNumber;
    // 产品数量上限
    private int maxGoodsNumber;

    public Goods(String goodsName, int goodsNumber, int maxGoodsNumber) {
        super();
        this.goodsName = goodsName;
        this.goodsNumber = goodsNumber;
        this.maxGoodsNumber = maxGoodsNumber;
    }

    // 消费方法
    public void consumption() throws InterruptedException {
        while(true) {
            synchronized (this) {
                if(this.goodsNumber < 1) {
                    System.out.println("产品售罄，等待生产者进行生产...");
                    wait();
                }
                this.goodsNumber--;
                Thread.sleep(500);
                System.out.println("消费 "+toString());
                notify();
            }
        }
    }

    // 生产方法
    public void producation() throws InterruptedException {
        while(true) {
            synchronized (this) {
                if(this.goodsNumber >= this.maxGoodsNumber) {
                    System.out.println("爆仓了，等待消费者消费...");
                    wait();
                }
                this.goodsNumber++;
                Thread.sleep(500);
                System.out.println("生产 "+toString());
                notify();
            }
        }
    }

    @Override
    public String toString() {
        return "[goodsName:"+goodsName+", goodNumber: "+goodsNumber+"]";
    }

}

// 消费者类
class Consumers implements Runnable {
    private Goods goods;

    public Consumers(Goods goods) {
```

```
        super();
        this.goods = goods;
    }

    @Override
    public void run() {
        try {
            goods.consumption();
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

}

// 生产者类
class Producers implements Runnable {
    private Goods goods;

    public Producers(Goods goods) {
        super();
        this.goods = goods;
    }

    @Override
    public void run() {
        try {
            goods.producation();
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

}


public class Test {
    public static void main(String[] args) throws Exception {
        Goods goods = new Goods("demo", 0, 10);
        Producers producers = new Producers(goods);
        Consumers consumers = new Consumers(goods);
        Thread producersThread = new Thread(producers);
        Thread consumersThread = new Thread(consumers);
        producersThread.start();
        consumersThread.start();
    }
}
```

运行结果(节选)：



到这里，我们就已经基本掌握了单一生产者消费者模型的基本内容了，最后，再介绍一下多对多的的生产者消费者模型。

2 多对多的的生产者消费者模型

比起单一生产者消费者模型，多对多只是将 notify()方法改成了 notifyAll()方法。

为了简化程序突出重点，本例中不进行产品数量上限的处理。

```
class Goods {
    // 产品名称
    private String goodsName;
    // 产品数量
    private int goodsNumber;

    public Goods(String goodsName, int goodsNumber) {
        super();
        this.goodsName = goodsName;
        this.goodsNumber = goodsNumber;
    }

    // 消费方法
    public void consumption() throws InterruptedException {
        while(true) {
            synchronized (this) {
                if(this.goodsNumber < 1) {
                    System.out.println("产品售罄，等待生产者进行生产...");
                    wait();
                }
                this.goodsNumber--;
                Thread.sleep(100);
                System.out.println("消费 "+toString());

            }
        }
    }

    // 生产方法
    public void producation() throws InterruptedException {
        while(true) {
            synchronized (this) {
                this.goodsNumber++;
                Thread.sleep(100);
                System.out.println("生产 "+toString());
                // 唤醒所有等待的线程
                notifyAll();
            }
        }
    }

    @Override
    public String toString() {
        return "[goodsName:"+goodsName+", goodNumber: "+goodsNumber+"]";
    }

}

// 消费者类
class Consumers implements Runnable {
    private Goods goods;
```

```
    public Consumers(Goods goods) {
        super();
        this.goods = goods;
    }

    @Override
    public void run() {
        try {
            goods.consumption();
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

}

// 生产者类
class Producers implements Runnable {
    private Goods goods;

    public Producers(Goods goods) {
        super();
        this.goods = goods;
    }

    @Override
    public void run() {
        try {
            goods.producation();
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

}


public class Test {
    public static void main(String[] args) throws Exception {
        Goods goods = new Goods("demo", 0);
        ArrayList<Thread> list = new ArrayList<>();

        // 十个生产者
        for(int i = 0; i < 10; i++) {
            Thread producersThread = new Thread(new Producers(goods));
            list.add(producersThread);
        }

        // 二十个消费者
        for(int i = 0; i < 20; i++) {
            Thread consumersThread = new Thread(new Consumers(goods));
            list.add(consumersThread);
        }

        // 启动线程
        for(Thread thread : list) {
            thread.start();
        }
```

```
    }
}
```