线程高级

火车售票问题

> 需求：火车站买票，一共有且仅有【100】张票，4个窗口同时贩卖
>
> 临界资源就是这个票，多个线程访问保证票有且仅有100张，
>
> 开4个线程同时买100张票
>
> 其实这个卖票案例就是多线程下同时操作一个资源(临界资源),如何保证线程操作资源安全不出现以下情况
>
> 买票不会出现重票 --> 线程1 卖出第99张票　线程2 卖出99张票
>
> 错误票 ---> 线程1 卖出第75张票. 线程2 卖出76张票 ---> 这个是对但是 线程2 卖出49张票
>
> 负数票 ---> 某个线程买完最后一张票的时候 剩余线程就不能在卖票 0 -1 -2...票
>
> 1秒卖一张 ---> 有且仅有100张 --->不要卖出400张票数

问题1：如何将票固定在100张，而不是每个线程都卖出100张票

> 如何做到所有的线程都共享这一个资源?
>
> 1.将票作为成员变量，作为成员变量之后，需要保证这个提供票的类在外界创建对象必须是唯一的，如果不唯一的，就会出现你创建票的对象每个对象都会唯一，一个成员变量，4个对象就是4个成员变量，此时相当于400张票【单例设计模式】
>
> PS：单例设计模式的原则：保证外界无论如何获取对象都是唯一的

```
//v1基础版本
package com.qfedu.Thread.SellTicket1;
//v1版本

/*
 不要让类继承与Thread，继承Thread必然要提供四个线程对象
 这样就胡出现无法控制票100张
```

```
 建议实现Runnable，以Runnable实现类方式创建线程处理
 */
public class SellTicket1 implements  Runnable{
    //提供成成员变量存储票的信息
    private int tickets = 100;

    @Override
    public void run() {
        //执行买票操作
       //提供一个for循环，这个循环执行100次
       for(int i = 0; i<100;i++){
           //提供买票的逻辑
           if(tickets > 0){//证明还票可以出售
               try {
                   //休眠1秒在买票
                   Thread.sleep(1000);
               } catch (InterruptedException e) {
                   e.printStackTrace();
               }
               //买票
               System.out.println("当前售票员
["+Thread.currentThread().getName()+"]第"
                       +"["+(tickets--)+"]票");

           }
       }
    }

    public static void main(String[] args) {
        //如果提供的是成员变量【保证存储成员变量的这个类所得到对象是
必须唯一】
        //出现400张票问题[在构建线程对象时，提供四个Runnable接口实
现类对象创建]
   /*    Thread thread1 = new Thread(new SellTicket1(),"刘
德华");
        Thread thread2 = new Thread(new SellTicket1(),"张
学友");
        Thread thread3 = new Thread(new SellTicket1(),"郭
富城");
        Thread thread4 = new Thread(new SellTicket1(),"吴
奇隆");
```

```
        thread1.start();
        thread2.start();
        thread3.start();
        thread4.start();
*/
         //在外界只创建一个Runnable接口实现类对象，保证对象是唯一的
          SellTicket1 sellTicket1 = new SellTicket1(); //只会维护一份的票对象【100】张
         //利用这个对象初始化创建4个线程对象
        Thread thread1 = new Thread(sellTicket1,"刘德华");
        Thread thread2 = new Thread(sellTicket1,"张学友");
        Thread thread3 = new Thread(sellTicket1,"郭富城");
        Thread thread4 = new Thread(sellTicket1,"吴奇隆");
        thread1.start();
        thread2.start();
        thread3.start();
        thread4.start();


    }
}
```

> 2.定义一个变量保证所有对象共享，无论外界如何创建线程对象【实现Runnable还是继承Thread类】，这里都可以保证所有兑现共享这个变量，**可以提供一个static修饰变量即静态变量**

```
package com.qfedu.Thread.SellTicket1;
//v1.1版本 ---> 提供了静态变量

public class SellTicket2 implements  Runnable{
    //提供静态成员变量存储票的信息
    private static int tickets = 100;

    @Override
    public void run() {
        //执行买票操作
       //提供一个for循环，这个循环执行100次
       for(int i = 0; i<100;i++){
           //提供买票的逻辑
```

```
            if(tickets > 0){//证明还票可以出售
                try {
                    //休眠1秒在买票
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                //买票
                System.out.println("当前售票员
["+Thread.currentThread().getName()+"]第"
                        +"["+(tickets--)+"]票");

            }
        }
    }

    public static void main(String[] args) {
      // 提供了静态变量，这个类的创建对象方式就无所谓
      Thread thread1 = new Thread(new SellTicket2(),"刘德
华");
        Thread thread2 = new Thread(new SellTicket2(),"张
学友");
        Thread thread3 = new Thread(new SellTicket2(),"郭
富城");
        Thread thread4 = new Thread(new SellTicket2(),"吴
奇隆");
        thread1.start();
        thread2.start();
        thread3.start();
        thread4.start();



    }
}
```

问题2:在买票的时候会出现、重票、错票、排序顺序的混乱的问题

> 这个问题产生就是多线程并发争抢"临界资源"出现里临界资源不安全问题

如何解决多线程并发访问临界资源，保证资源安全问题

> 此时Java为了保证多线程并发访问临界资源线程安全问题，提供了一个处理机制，可以在处理临界资源代码中添加**（同步锁/对象锁/同步代码块/同步监听器），这里四个名称都是指的是同一个操作，可以添加“锁机制”**，保证在同一个时刻的只有一个线程操作临界资源，保证临界资源安全

synchronized关键字

> synchronized翻译意思：同步
>
> 使用synchronized关键字进行同步操作，使用synchronized组成同步操作叫做【同步锁/对象锁/同步代码块】
>
> 语法：
>
> synchronized(资源锁对象){
>
> 提供操作临界资源的代码
>
> }
>
> 执行原理： 利用了同一时间内【只能有一个线程对象持有当前资源锁对象】，只要线程不释放这个所资源对象，外界线程是无法执行线程逻辑资源代码，在同步代码块中执行线程进行休眠是不会释放CPU时间片到外界给其他线程，因为sleep方法是不会释放锁资源
>
> 需要注意还需要保证【资源锁对象必须是唯一的】，不唯一锁对象数无法进行同步操作
>
> **如果保证资源锁对象是唯一的?**
>
> 个人建议：十分不推荐使用 this 作为锁资源对象，this可能锁不了，this代表的是当前对象，那么就意味着这个提供临界支援类必须是创建唯一的对象才可以，这样才可以保证this是唯一的
>
> 替代方案一：
>
> 在静态资源处理线程类中提供一个全局静态常量“锁资源”
>
> ```
> private static final Object  obj = new Object();
> synchronized(obj){
> 提供操作临界资源的代码
> }
> ```

> 替代方案二【推荐】：
>
> 天生对象就是唯一的，Java中这个对象无论你如何操作，就可以保证他是唯一的地址都是同一个，String字符串类，直接使用字符串对象即可---->【使用""(双引号)创建】
>
> 提供字符串空串处理 ---> 即 提供字符串空串对象

```
synchronized(""){
提供操作临界资源的代码
}
```

使用同步代码块修改买票的案例的

```
package com.qfedu.Thread.SynchronizedSellTicket1;
//v2版本 ---> 提供了静态变量 和 同步代码块执行买票操作

public class SellTicket implements  Runnable{
    //提供静态成员变量存储票的信息
    private static int tickets = 100;

    @Override
    public void run() {
        //执行买票操作
       //提供一个for循环，这个循环执行100次
       for(int i = 0; i<100;i++){
           //提供买票的逻辑
           synchronized ("") {//同步代码块
               if (tickets > 0) {//证明还票可以出售
                   try {
                       //休眠1秒在买票
                       Thread.sleep(1000);
                   } catch (InterruptedException e) {
                       e.printStackTrace();
                   }
                   //买票
                   System.out.println("当前售票员[" +
Thread.currentThread().getName() + "]第"
                           + "[" + (tickets--) + "]票");

               }
           }
```

```
        }
    }

    public static void main(String[] args) {
      // 提供了静态变量，这个类的创建对象方式就无所谓
      Thread thread1 = new Thread(new SellTicket(),"刘德
华");
        Thread thread2 = new Thread(new SellTicket(),"张学
友");
        Thread thread3 = new Thread(new SellTicket(),"郭富
城");
        Thread thread4 = new Thread(new SellTicket(),"吴奇
隆");
        thread1.start();
        thread2.start();
        thread3.start();
        thread4.start();



    }
}
```

同步方法/对象同步方法/成员同步方法

> 使用synchronized关键字进行成员方法的修饰，这个方法就会成为同步方法将操作资源代码写入到这个方法中就可以进行同步操作了
>
> 语法：
>
> 访问权限修饰符 synchronized 返回值类型 方法名(参数列表){
>
> 方法体;
>
> return;
>
> }
>
> **特别注意：使用是同步方法，所以这个方法使用锁资源对象是【this】，提供同步方法的这个类所创建对象必须是唯一的才可以锁住资源**
>
> PS：到目前位置 方法定义中其他修饰符就已经全部介绍完毕

> 类中方法可以使用：【static、final、abstract、synchronized】
>
> 接口中方法可以使用：【abstract、static、default】

修饰同步方法是该买票案例

```
package com.qfedu.Thread.SynchronizedSellTicket1;
//v2.1版本 ---> 提供了静态变量 和 同步方法进行买票

public class SellTicket2 implements  Runnable{
    //提供静态成员变量存储票的信息
    private static int tickets = 100;

    @Override
    public void run() {
        //执行买票操作
       //提供一个for循环，这个循环执行100次
       for(int i = 0; i<100;i++){
           //提供买票的逻辑 ---》调用同步方法即可
          seller();

       }
    }
     //同步方法 --> 这个所资源是this对象
    public synchronized  void  seller(){
        //提供就是线程处理临界资源操操作
        if (tickets > 0) {//证明还票可以出售
            try {
                //休眠1秒在买票
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            //买票
            System.out.println("当前售票员[" +
Thread.currentThread().getName() + "]第"
                    + "[" + (tickets--) + "]票");

        }
    }

    public static void main(String[] args) {
```

```
        // 提供了静态变量，这个类的创建对象方式就无所谓
        //同步方法提供类如果是线程类，需要保证线程类对象是唯一的
          SellTicket2 sellTicket2 = new SellTicket2();
          Thread thread1 = new Thread(sellTicket2,"刘德华");
          Thread thread2 = new Thread(sellTicket2,"张学友");
          Thread thread3 = new Thread(sellTicket2,"郭富城");
          Thread thread4 = new Thread(sellTicket2,"吴奇隆");
          thread1.start();
          thread2.start();
          thread3.start();
          thread4.start();



    }
}
```

类锁/类同步代码块/类同步方法/静态同步方法

> 如果使用是同步代码块的语法，就是将资源锁对象，从当前对象替换成类对象，通过类的方式获取对象【通反射获取】，静态同步方法就是使用static和synchronized同时修饰方法
>
> **类锁/类同步代码块**
>
> synchronized(类名.class){// 类锁对象
>
> 操作临界资源代码即可
>
> }**类同步方法/静态同步方法**
>
> 访问权限修饰符 static synchronized 返回值类型 方法名(参数列表){
>
> 方法代码
>
> return;
>
> }

```
//v2.2版本 ---> 提供了静态变量 和 静态同步操作

public class SellTicket3 implements  Runnable{
    //提供静态成员变量存储票的信息
    private static int tickets = 100;
```

```
    @Override
    public void run() {
        //执行买票操作
       //提供一个for循环，这个循环执行100次
       for(int i = 0; i<100;i++){
           //提供买票的逻辑 --->调用同步方法即可
           //seller();
           synchronized (String.class){ // 类锁
               if (tickets > 0) {//证明还票可以出售
                   try {
                       //休眠1秒在买票
                       Thread.sleep(1000);
                   } catch (InterruptedException e) {
                       e.printStackTrace();
                   }
                   //买票
                   System.out.println("当前售票员[" +
Thread.currentThread().getName() + "]第"
                           + "[" + (tickets--) + "]票");

               }
           }

       }
    }
     //静态同步方法 --> 这个锁资源就是当前类的对象【是唯一的】，通过
类获取即 类.class
    public static synchronized  void  seller(){
        //提供就是线程处理临界资源操操作
        if (tickets > 0) {//证明还票可以出售
            try {
                //休眠1秒在买票
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            //买票
            System.out.println("当前售票员[" +
Thread.currentThread().getName() + "]第"
                    + "[" + (tickets--) + "]票");
```

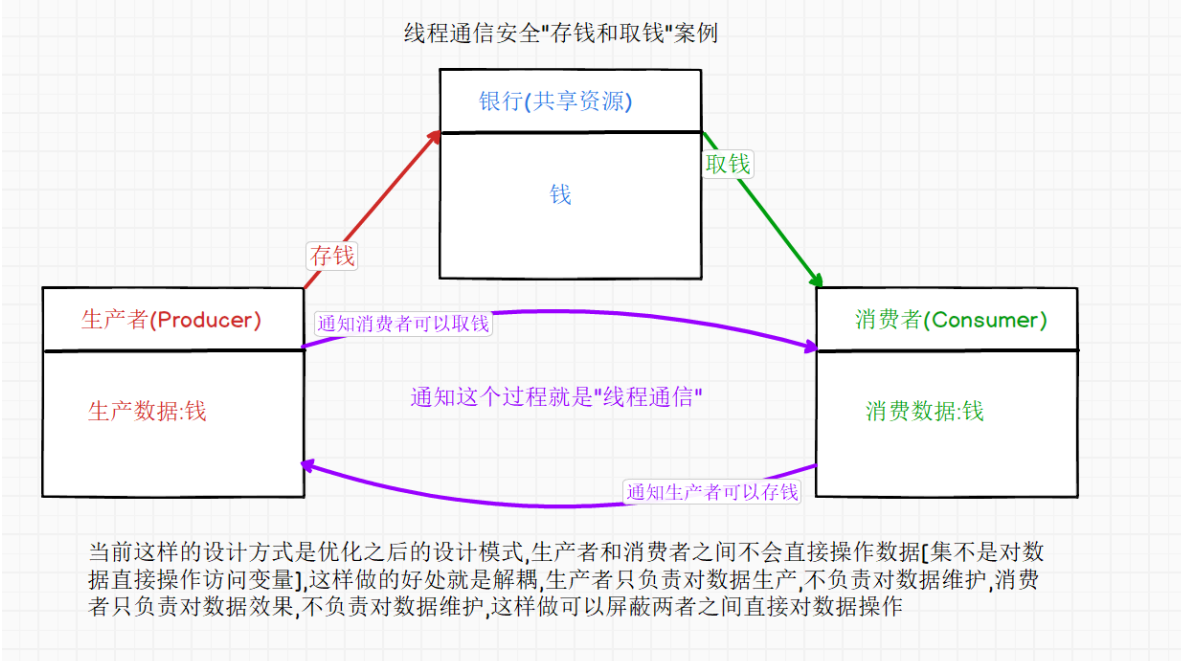
```
        }
    }

    public static void main(String[] args) {
      // 提供了静态变量，这个类的创建对象方式就无所谓
      //同步方法提供类如果是线程类，需要保证线程类对象是唯一的
        SellTicket3 sellTicket2 = new SellTicket3();
        Thread thread1 = new Thread(sellTicket2,"刘德华");
        Thread thread2 = new Thread(sellTicket2,"张学友");
        Thread thread3 = new Thread(sellTicket2,"郭富城");
        Thread thread4 = new Thread(sellTicket2,"吴奇隆");
        thread1.start();
        thread2.start();
        thread3.start();
        thread4.start();



    }
}
```

> synchronized总结：
>
> 1.在开发中常用肯定是对象锁【同步代码块】但是这个代码块的范围不要太大，不然的话会影响执行效率
>
> PS: 在什么位置对资源操作就在什么位置添加 同步代码块
>
> 2.使用synchronized关键字之后线程就会变成安全，间接就将并行线程修改为串行线程，使用锁机制将线程变成 【一个一个站排执行效果】

线程通信

> 到现目前位置线程与线程之间依旧是彼此独立，就算使用同步代码块，也是让线程执行有一定顺序，但是还是没有进行通信操作，如果需要遇到某个线程执行完毕之后需要通知另外一个线程继续执行，此时就需要使用到线程通信，可以在一个进程中开发多个线程，每个线

> 程之间可以进行通信操作，从而协调性的完成某个进程中需要执行逻辑
>
> **线程通信有一个标准的通信模型【生产者与消费者模式】**



完成生产者和消费者通信模型v1版本

Bank(银行类) ---》 提供数据维护与使用

```
//基础的第一个版本 ---》 提供临界资源并提供操作
public class Bank {
      /*
      临界资源一共有两种提供方式【提供成员变量和提供静态变量】
       */
      private double money;//存储前的临界变量
       /*
       外界无论如何创建Bank类对象，都需要是唯一的，不然的话成员变量会跟着对象多少而进行创建
       提供对象唯一创建方式【单例设计模式】
       单例设计模式可以保证外界无论如何操作得到对象都是唯一
       单例设计模式的实现：
       1. 一定要私有化构造方法    --> 防止外界调用构造方法创建对象
       2. 提供一个私有静态常量对象，这个对象就是单例类唯一创建对象
       3. 提供一个静态方法，获取这个创建好的对象，得到实例【外界只能通过这个方法获取对象】
        */
       private Bank(){}
       private static final Bank bank = new Bank();
```

```
      public static Bank getInstance(){
          return  bank;
      }

    /**
     *生产者向共享资源中存储数据的方法【生产】
     * @param m 存储钱
     */
      public void pushMoney(double m){
          money = m+money;

 System.out.println(Thread.currentThread().getName()+"存了"+m+"余额是："+money);
      }
    /**
     *消费者向共享资源中存储数据的方法【消费】
     * @param m 消费钱
     */
    public void popMoney(double m){
        money = money-m;

 System.out.println(Thread.currentThread().getName()+"取了"+m+"余额是："+money);
    }



}
```

生产者【是一个线程提供数据存储操作】

```
//生产者
public class Producer implements  Runnable{
    //提供银行属性
    private Bank bank;
    public  Producer(Bank bank){
        this.bank = bank;
    }

    @Override
    public void run() {
```

```
        //提供线程存储钱的操作 存储十次 每次1000元
        for(int i = 0 ;i<10;i++){
            bank.pushMoney(1000);
        }
    }
}
```

消费者【是一个线程提供数据获取操作】

```
//消费者
public class Consumer implements  Runnable{
    //提供银行属性
    private Bank bank;
    public Consumer(Bank bank){
        this.bank = bank;
    }

    @Override
    public void run() {
        //提供线程存获取的操作 获取十次 每次1000元
        for(int i = 0 ;i<10;i++){
            bank.popMoney(1000);
        }
    }
}
```

提供一个测试类Test进行测试操作

```
package com.qfedu.Thread.waitAndnotfiny;

public class Test {
    public static void main(String[] args) {
        //通过单例获取银行对象
        Bank bank1 = Bank.getInstance();
        //2.创建线程对象并启动操作
        new Thread(new Producer(bank1),"花花").start();
        new Thread(new Consumer(bank1),"菲菲").start();

    }
}
```

问题1：现在执行速度太快了，看不到交替执行效果，让线程休眠1秒操作

问题2：虽然让线程休眠之后，出现了一个执行错乱的问题，因为多线程并发访问临界资源money，出现线程挣抢的问题，所以出现临界资源部不安全问题，添加同步代码块

Bank银行类需要修改为

```
package com.qfedu.Thread.waitAndnotfiny;
//基础的第一个版本 ---》 提供临界资源并提供操作
public class Bank {
      /*
      临界资源一共有两种提供方式【提供成员变量和提供静态变量】
       */
      private double money;//存储前的临界变量
       /*
       外界无论如何创建Bank类对象，都需要是唯一的，不然的话成员变量会跟着对象多少而进行创建
       提供对象唯一创建方式【单例设计模式】
       单例设计模式可以保证外界无论如何操作得到对象都是唯一
       单例设计模式的实现：
       1. 一定要私有化构造方法    --> 防止外界调用构造方法创建对象
       2. 提供一个私有静态常量对象，这个对象就是单例类唯一创建对象
       3. 提供一个静态方法，获取这个创建好的对象，得到实例【外界只能通过这个方法获取对象】
        */
       private Bank(){}
       private static final Bank bank = new Bank();
       public static Bank getInstance(){
           return  bank;
       }

    /**
     *生产者向共享资源中存储数据的方法【生产】
     * @param m 存储钱
     */
      public void pushMoney(double m){
          synchronized ("") {
              try {
                  Thread.sleep(1000);
                  money = m + money;
```

```
 System.out.println(Thread.currentThread().getName() + "存了" + m + "余额是: " + money);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    /**
     *消费者向共享资源中存储数据的方法【消费】
     * @param m 消费钱
     */
    public void popMoney(double m){
        synchronized ("") {
            try {
                Thread.sleep(1000);
                money = money - m;

 System.out.println(Thread.currentThread().getName() + "取了" + m + "余额是: " + money);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }



}
```

完成生产者和消费者通信模型v2版本

> 使用了同步代码块+sleep方法，虽然让程序执行有序了也保证线程中操作临界资源过程也安全了，但是还是无法保证线程可以做到交替执行的效果**【即生产者生产完数据之后通知消费者可以进行消费，消费者消费完数据之后通知生产者可以生产】**，所以如何完成这个通信操作呢?
>
> **此时就需要使用Java中Object类中提供三个方法了**

线程通信	

线程通信 线程通信 方法 方法	方法说明 方法说明
wait()	执行该方法的线程对象会释放【同步锁资源】，JVM会把当前线程放到等待池中，等待其他线程唤醒操作
notify()	执行该方法的线程对象会在等待池【随机唤醒】一个等待的线程，把线程转移到锁池中等待，再次获取资源
notifyAll()	执行该方法的线程对象会在等地池中【唤醒所有的线程】，把线程转移到锁池中等待，再次获取资源

就可以利用这三个方法进行线程通信操作了，**特别注意调用通信方法必须是锁资源对象，否则会出现异常提示信息【在哪里调用等待方法就在位置被唤醒】**

线程通信执行流程：

假设**A线程**和**B线程**共同操作一个**X锁对象**，A、B线程可以通过X锁对象调用wait和notify方法进行如下线程通信操作：

1. 当A线程执行到X锁对象时，A线程持有X锁对象，B线程是没有执行机会的，B线程在X对象的等待锁池中
2. A线程在同步代码块种执行到X.wait方法时，A线程会释放X对象所资源，A线程会进入到X对象的等待锁池中
3. B线程就会在等待锁池中得到A线程释放的X资源锁对象，B线程就开始执行操作
4. B线程在同步代码块中执行到X.notify方法时，JVM把A线程从X对象等待池中，移动到等待X对象锁资源中得到锁资源
5. B线程执行完毕之后释放锁资源，A线程就有机会获取到锁，继续执行同步方法

修改Bank类提供线程通信操作者

```
package com.qfedu.Thread.waitAndnotfiny.V2;
//通信的版本 ---》 提供临界资源并提供操作[添加线程通信和同步代码块+sleep方法]
public class Bank {
    /*
    临界资源一共有两种提供方式【提供成员变量和提供静态变量】
     */
```

```
    private double money;//存储前的临界变量
     /*
     外界无论如何创建Bank类对象，都需要是唯一的，不然的话成员变量会跟着对象多少而进行创建
     提供对象唯一创建方式【单例设计模式】
     单例设计模式可以保证外界无论如何操作得到对象都是唯一
     单例设计模式的实现：
     1. 一定要私有化构造方法    --> 防止外界调用构造方法创建对象
     2. 提供一个私有静态常量对象，这个对象就是单例类唯一创建对象
     3. 提供一个静态方法，获取这个创建好的对象，得到实例【外界只能通过这个方法获取对象】
      */
     private Bank(){}
     private static final Bank bank = new Bank();
     public static Bank getInstance(){
         return  bank;
     }
    /*
    在外界创建生产者与消费者线程对象中都是可以获取到CPU资源
    但是问题在于，如果消费者先获取到资源，但是没有数据不能消费
    所以就提供一个变量来存储状态的，通过对这个状态的判断决定执行哪个线程
     */
    private  static  boolean isEmpty = false;
    /**
     *生产者向共享资源中存储数据的方法【生产】
     * @param m 存储钱
     */
     public void pushMoney(double m){
         synchronized ("") {
             try {
                 //添加线程通信操作
                 if(isEmpty){
                     //如果条件为true值，就证明现在有数据，无需生产

                     //生产这个就进入到等待状态
                     "".wait();
                 }
                 //因为if没有执行就证明，没有数据可以生产
                 Thread.sleep(1000);
                 money = m + money;
```

```
 System.out.println(Thread.currentThread().getName() + "存了" + m + "余额是: " + money);
                    //生产数据完毕之后需要修改状态信息
                    isEmpty = true;
                    //执行唤醒操作
                    "".notify();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    /**
     *消费者向共享资源中存储数据的方法【消费】
     * @param m 消费钱
     */
    public void popMoney(double m){
        synchronized ("") {
            try {
                // 增加线程通信操作
                //isEmpty状态值 false代表没有数据 true代表有属性
                if(!isEmpty){ //没有数据的时候不能执行消费，等待生产
                    /*
                     没有数据isEmpty值是false，取非操作 !false --》 true if就执行等待
                     有数据isEmpty值是true ，取非操作 !true --》false if就不执行等待，进行消费
                     */
                    "".wait();
                }
                //if分支语句不执行就证明有数据，进行消费操作
                Thread.sleep(1000);
                money = money - m;

 System.out.println(Thread.currentThread().getName() + "取了" + m + "余额是: " + money);
                //修改状态数据
                isEmpty = false;
                //进行唤醒操作
```

```
                "".notify();

            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }



}
```

分析执行流程

1. 现有的两个线程**生产者【花花】**和**消费者【菲菲】**默认初始isEmpty状态都是false
2. 消费者【菲菲】先获取到CPU时间片，持有所资源，判断if分支条件为!false,消费者【菲菲】就会释放锁资源并且进入到等待池中

```
//等待池
/*
 生产者【花花】
*/
```

3. 等待池中生产者【花花】就会或得到执行CPU时间片，并且获取到锁资源，判断if分支条件false，证明没有数据，生产数据并且唤醒等待池中【菲菲】修改isEmpty值true，生产者线程就执行完毕了

```
//等待池
/*
 消费者【菲菲】
*/
```

4. 此时生产者【花花】和消费者【菲菲】就会再次同时争抢CPU时间片和锁资源对象，假如生产者【花花】再次获取哦到CPU时间片和锁资源对象，执行线程逻辑，判断if分支条件为true，此时花花就会释放资源进入到等待池中

```
//等待池
/*
 生产者【花花】
*/
```

5. 消费者【菲菲】必然获取到CPU时间片，执行操作逻辑判断条件 !true 得到是false，就执行消费，并且修改状态为false，唤醒等待池中生产者，这样就完成交替执行效果

PS：以上这些操作就是典型的一对一生产者与消费者模型

多对多通信模型

> 提供线程对象创建操作， 这里从原有一对一的模式，变成多对多的模式，原来的一个生产者对应一个消费者，修改为了多个生产者对应多个消费者

```
public class Test {
    public static void main(String[] args) {
        //通过单例获取银行对象
        Bank bank1 = Bank.getInstance();
        //2.创建线程对象并启动操作
        new Thread(new Producer(bank1),"花花1").start();
        new Thread(new Consumer(bank1),"菲菲2").start();
        new Thread(new Producer(bank1),"花花3").start();
        new Thread(new Consumer(bank1),"菲菲4").start();

    }
}
```

> 当将线程对象增多之后，执行多线程同时操作就出现两个问题**“负数问题和死等待问题”**

负数问题

> 现有线程【花花1（存）、菲菲2（取）、花花3（存）、菲菲4（取）】，模拟一个执行流程
>
> 1. 菲菲2先抢到线程，不能取，所以进入到等待队列中，释放CPU时间片和释放锁资源

2. 菲菲4抢到线程资源，不能取，所以进入到等待列中，释放CPU时间片和释放锁资源

```
//等待队列  菲菲2、菲菲4
```

3. 花花1抢到线程，存储了数据，修改标记为true值，并唤醒等待队列中的线程的线程对象，现在使用唤醒机制notify方法【随机唤醒一个等待队列汇总线程对象】，假设唤醒了菲菲4

```
//等待队列中    菲菲2
//外部争抢CPU操作的时间片  花花1、花花3、菲菲4
//临界资源中存储数据是  ---》  1000
```

4. 菲菲4抢到了线程，在哪里等待就在哪里被唤醒，而且if分支语句特点，只会执行一次判断即if分支条件判断完成之后【不添加循环前提】，只会执行一次判断，菲菲4正好是在if分支语句中进行等待操作，醒来之后机会继续向后执行【它是不会在判断if分支语句】，菲菲4进行正常消费，修改标记为false，需要执行notify，唤醒等待队列菲菲2【菲菲2这个线程是通过wait进入到等待状态，它已经执行了if判断】

```
//等待队列中     空
// 外部增强CPU操作者时间片  花花1、菲菲2【从等待状态恢复而来】、花花3、菲菲4
// 临界资源中存储的数据是  ----》  0  【因为菲菲4消费了】
```

5. 菲菲2抢到了执行权，会在if分支语句继续执行操作，此时虽然标记已经修改为false，正常而言不应该消费的，但是if分支语句已经被执行过，所以不会进行判断继续执行，进行消费，修改标记，执行唤醒

但是现在的数据是 0 在执行消费出现数据就是 -1000

为了保证线程通信可以正常执行，建议在开发的时候，不要书写if作为状态判断操作，建议使用while循环语句替代if分支，while循环语句只有条件为false，才不执行，否者就继续执行，这样就可以让等待位置线程在次被唤醒之后可以再次判断状态操作，以防出现错误数据

```
package com.qfedu.Thread.waitAndnotfiny.V3;
```

```
//通信的版本【多对多】 ---》 提供临界资源并提供操作[添加线程通信和同步代码块+sleep方法]
public class Bank {
      /*
      临界资源一共有两种提供方式【提供成员变量和提供静态变量】
       */
      private double money;//存储前的临界变量
       /*
       外界无论如何创建Bank类对象，都需要是唯一的，不然的话成员变量会跟着对象多少而进行创建
       提供对象唯一创建方式【单例设计模式】
       单例设计模式可以保证外界无论如何操作得到对象都是唯一
       单例设计模式的实现：
       1. 一定要私有化构造方法    --> 防止外界调用构造方法创建对象
       2. 提供一个私有静态常量对象，这个对象就是单例类唯一创建对象
       3. 提供一个静态方法，获取这个创建好的对象，得到实例【外界只能通过这个方法获取对象】
        */
       private Bank(){}
       private static final Bank bank = new Bank();
       public static Bank getInstance(){
           return  bank;
       }
     /*
     在外界创建生产者与消费者线程对象中都是可以获取到CPU资源
     但是问题在于，如果消费者先获取到资源，但是没有数据不能消费
     所以就提供一个变量来存储状态的，通过对这个状态的判断决定执行哪个线程
      */
     private  static  boolean isEmpty = false;
     /**
      *生产者向共享资源中存储数据的方法【生产】
      * @param m 存储钱
      */
       public void pushMoney(double m){
           synchronized ("") {
               try {
                   //添加线程通信操作
                   //只需要将if修改为while即可
                   while(isEmpty){
```

```
                        //如果条件为true值，就证明现在有数据，无需
生产
                        //生产这个就进入到等待状态
                        "".wait();
                    }
                    //因为if没有执行就证明，没有数据可以生产
                    Thread.sleep(1000);
                    money = m + money;

 System.out.println(Thread.currentThread().getName() + "存
了" + m + "余额是：" + money);
                    //生产数据完毕之后需要修改状态信息
                    isEmpty = true;
                    //执行唤醒操作
                    "".notify();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    /**
     *消费者向共享资源中存储数据的方法【消费】
     * @param m 消费钱
     */
    public void popMoney(double m){
        synchronized ("") {
            try {
                // 增加线程通信操作
                //isEmpty状态值 false代表没有数据 true代表有属
性
                //只需要将if修改为while即可
                while(!isEmpty){ //没有数据的时候不能执行消费，
等待生产
                    /*
                     没有数据isEmpty值是false，取非操作 !false
--》 true if就执行等待
                     有数据isEmpty值是true ，取非操作 !true -
-》false if就不执行等待，进行消费
                     */
                    "".wait();
                }
```

```
                    //if分支语句不执行就证明有数据，进行消费操作
                    Thread.sleep(1000);
                    money = money - m;

 System.out.println(Thread.currentThread().getName() + "取了" + m + "余额是：" + money);
                    //修改状态数据
                    isEmpty = false;
                    //进行唤醒操作
                    "".notify();

                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }



}
```

死等待问题（所有线程都进入到等待池中无人唤醒）

> **PS：这个效果是需要特殊情况下才会出现【它属于一个极端情况】**
>
> **这个原因的产生就是使用notfiy随机唤醒机制**
>
> 现有线程【花花1（存）、菲菲2（取）、花花3（存）、菲菲4（取）】，模拟一个执行流程
>
> 1. 菲菲2先抢到线程，不能取，所以进入到等待池中，释放锁和CPU时间片
> 2. 菲菲4抢到线程，不能取，所以进入到等待池中，释放锁和CPU时间片
>
> ```
> //等待队列中  菲菲2、菲菲4
> ```
>
> 3. 花花1抢到线程，存储数据，修改标记为true并进行唤醒操作【随机唤醒】
>
> ```
> //等待队列中  菲菲2
> //争抢CPU资源  花花1、花花3、菲菲4
> ```

4. 花花1在次获取线程，不能存储，所以要进入到等待队列中，释放锁和CPU时间片

```
//等待队列中  菲菲2、花花1
//争抢CPU资源  花花3、菲菲4
```

5. 花花3抢到线程，不能存储，所以要进入到等待队列中，释放锁和CPU时间片

```
//等待队列中  菲菲2、花花1、花花3
//争抢CPU资源  菲菲4
```

6. 菲菲4抢到线程，取钱消费，修改标记为false，并进入唤醒操作【随机唤醒】 --》 菲菲2唤醒

```
//等待队列汇总  花花1、花花3
//争抢CPU资源  菲菲4、菲菲2 ，现在状态时false
```

7. 生产者都在等待队列中，外部都是消费这个，现在状态时false

```
//等待队列中  花花1、花花3、菲菲4、菲菲2
```

所有线程都进入到等待队列中，无人唤醒就出现死的等待问题

之所以会出现死等待原因就因为使用了notify的操作，它是随机唤醒，等待池中两个消费，外界两个生产【生产完毕，状态修改，无法唤醒】，或等待池中两个生产，外界两个消费【消费完毕，状态修改，无法唤醒】，建议使用notifyAll 替换 notfiy，进行全部唤醒，然后再进行条件判断执行操作，决定好是否存储与消费

```
//通信的版本 ---》 提供临界资源并提供操作[添加线程通信和同步代码块+sleep方法]
public class Bank {
	/*
	临界资源一共有两种提供方式【提供成员变量和提供静态变量】
	 */
	private double money;//存储前的临界变量
	 /*
	  外界无论如何创建Bank类对象，都需要是唯一的，不然的话成员变量会跟着对象多少而进行创建
	  提供对象唯一创建方式【单例设计模式】
```

```
        单例设计模式可以保证外界无论如何操作得到对象都是唯一
        单例设计模式的实现：
        1. 一定要私有化构造方法    --> 防止外界调用构造方法创建对象
        2. 提供一个私有静态常量对象，这个对象就是单例类唯一创建对象
        3. 提供一个静态方法，获取这个创建好的对象，得到实例【外界只能通过这个方法获取对象】
         */
        private Bank(){}
        private static final Bank bank = new Bank();
        public static Bank getInstance(){
            return  bank;
        }
    /*
    在外界创建生产者与消费者线程对象中都是可以获取到CPU资源
    但是问题在于，如果消费者先获取到资源，但是没有数据不能消费
    所以就提供一个变量来存储状态的，通过对这个状态的判断决定执行哪个线程
     */
    private  static  boolean isEmpty = false;
    /**
     *生产者向共享资源中存储数据的方法【生产】
     * @param m 存储钱
     */
      public void pushMoney(double m){
          synchronized ("") {
              try {
                  //添加线程通信操作
                  while(isEmpty){
                      //如果条件为true值，就证明现在有数据，无需生产

                      //生产这个就进入到等待状态
                      "".wait();
                  }
                  //因为if没有执行就证明，没有数据可以生产
                  Thread.sleep(1000);
                  money = m + money;

 System.out.println(Thread.currentThread().getName() + "存了" + m + "余额是：" + money);
                  //生产数据完毕之后需要修改状态信息
                  isEmpty = true;
```

```
                    //执行唤醒操作进行全部唤醒操作
                    "".notifyAll();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    /**
     *消费者向共享资源中存储数据的方法【消费】
     * @param m 消费钱
     */
    public void popMoney(double m){
        synchronized ("") {
            try {
                // 增加线程通信操作
                //isEmpty状态值 false代表没有数据 true代表有属性
                while(!isEmpty){ //没有数据的时候不能执行消费，等待生产
                    /*
                     没有数据isEmpty值是false，取非操作 !false --》 true if就执行等待
                     有数据isEmpty值是true ，取非操作 !true --》false if就不执行等待，进行消费
                     */
                    "".wait();
                }
                //if分支语句不执行就证明有数据，进行消费操作
                Thread.sleep(1000);
                money = money - m;

 System.out.println(Thread.currentThread().getName() + "取了" + m + "余额是: " + money);
                //修改状态数据
                isEmpty = false;
                //进行唤醒操作修改为全部唤醒
                "".notifyAll();

            } catch (InterruptedException e) {
                e.printStackTrace();
            }
```

```
            }
        }



}
```

> **总结：如果在日后的开发中出现线程通信操作，建议判断状态标签的位置使用while关键字而不是if，进行唤醒操作建议使用notfiyAll而不是notify**

扩展：Lock接口【显示锁】

> Lock接口是Java5中新加入与Synchronized比较操作，它属于显示定义，结构更加灵活，提供了一个更加便捷的操作，相对比synchronized更加简洁易用
>
> PS：这个锁资源Lock虽然语法更加简洁易用，但是它不能定义锁对象，Lock使用锁对象是【this】，使用Lock时就需要保证this的唯一性了
>
> **Lock锁如何操作?**
>
> Lock是一个接口所以需要使用到他实现类进行锁的操作
> ReentrantLock提供两常用操作

Lock锁方法	说明
lock()	添加锁(对象上锁)
unlock()	释放锁(对象所释放)

> 以为Lock是一个显示锁对象，所有不能调用Object类中提供wait、notify、notifyAll方法进行线程通信，所以系统就提供了一个和Lock锁一起使用线程通信接口Condititon

通信方法	说明
await()	线程等待释放锁资源和CPU时间片
signal()	唤醒等待池中某一个线程对象【随机唤醒】
signalAll()	唤醒等待池中所有线程对象

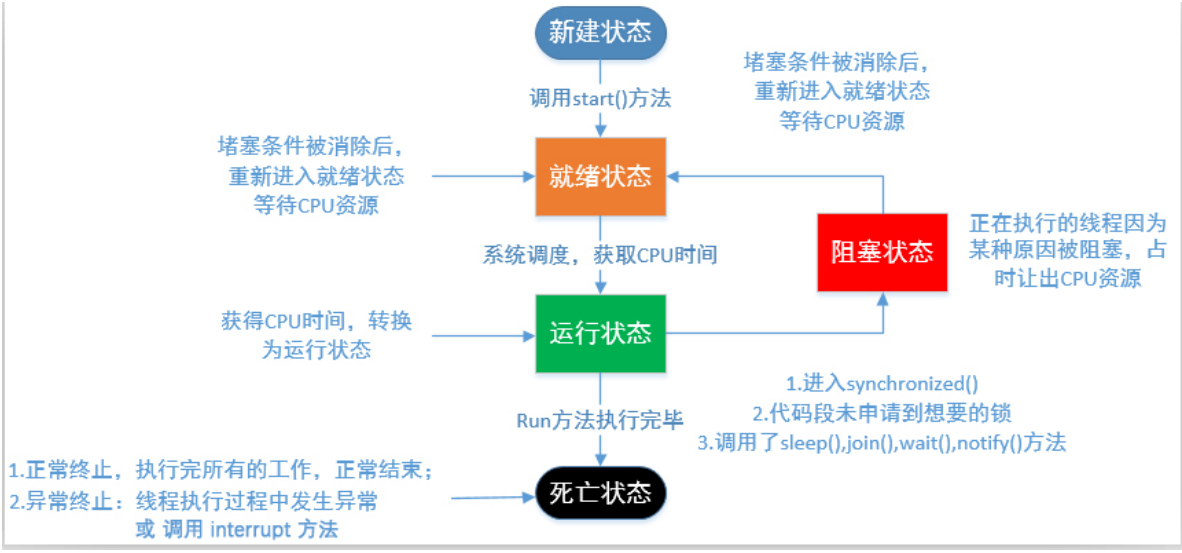
使用Lock锁替代Synchronized同步代码块进行数据操作

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

//通信的版本【多对多】 ---》 提供临界资源并提供操作[添加线程Lock锁+sleep方法+线程通信]
public class BankLock {
      /*
      临界资源一共有两种提供方式【提供成员变量和提供静态变量】
       */
      private double money;//存储前的临界变量
       /*
       外界无论如何创建Bank类对象，都需要是唯一的，不然的话成员变量会跟着对象多少而进行创建
       提供对象唯一创建方式【单例设计模式】
       单例设计模式可以保证外界无论如何操作得到对象都是唯一
       单例设计模式的实现：
       1. 一定要私有化构造方法    --> 防止外界调用构造方法创建对象
       2. 提供一个私有静态常量对象，这个对象就是单例类唯一创建对象
       3. 提供一个静态方法，获取这个创建好的对象，得到实例【外界只能通过这个方法获取对象】
        */
       private BankLock(){}
       private static final BankLock bank = new BankLock();
       public static BankLock getInstance(){
           return  bank;
       }
       //提供锁资源对象
       private  final Lock lock = new ReentrantLock();
       //提供线程通信资源对象【它是接口不能new，使用lock对象调用newCondition创建】
       private Condition condition =  lock.newCondition();
     /*
     在外界创建生产者与消费者线程对象中都是可以获取到CPU资源
     但是问题在于，如果消费者先获取到资源，但是没有数据不能消费
     所以就提供一个变量来存储状态的，通过对这个状态的判断决定执行哪个线程
      */
```

```
    private  static  boolean isEmpty = false;
    /**
     *生产者向共享资源中存储数据的方法【生产】
     * @param m 存储钱
     */
      public void pushMoney(double m){
          //synchronized ("") {//替换成lock锁资源对象
          lock.lock();
              try {
                  //添加线程通信操作
                  //只需要将if修改为while即可
                  while(isEmpty){
                      //如果条件为true值，就证明现在有数据，无需
生产
                      //生产这个就进入到等待状态
                      //  "".wait();
                      condition.await();//使用通信资源创建对
象
                  }
                  //因为if没有执行就证明，没有数据可以生产
                  Thread.sleep(1000);
                  money = m + money;

 System.out.println(Thread.currentThread().getName() + "存
了" + m + "余额是: " + money);
                  //生产数据完毕之后需要修改状态信息
                  isEmpty = true;
                  //执行唤醒操作
                  //"".notifyAll();
                   condition.signalAll();//使用通信资源创建对
象
              } catch (InterruptedException e) {
                  e.printStackTrace();
              }finally {
                  lock.unlock(); //锁资源释放
              }
          //}
      }
    /**
     *消费者向共享资源中存储数据的方法【消费】
     * @param m 消费钱
```

```
     */
    public void popMoney(double m){
        //synchronized ("") {
        lock.lock();
            try {
                // 增加线程通信操作
                //isEmpty状态值 false代表没有数据 true代表有属性
                //只需要将if修改为while即可
                while(!isEmpty){ //没有数据的时候不能执行消费，等待生产
                    /*
                     没有数据isEmpty值是false，取非操作 !false --》 true if就执行等待
                     有数据isEmpty值是true ，取非操作 !true --》false if就不执行等待，进行消费
                     */
                    //"".wait();
                    condition.await();
                }
                //if分支语句不执行就证明有数据，进行消费操作
                Thread.sleep(1000);
                money = money - m;

 System.out.println(Thread.currentThread().getName() + "取了" + m + "余额是: " + money);
                //修改状态数据
                isEmpty = false;
                //进行唤醒操作
               // "".notifyAll();
                condition.signalAll();

            } catch (InterruptedException e) {
                e.printStackTrace();
            }finally {
                lock.unlock();
            }
       // }
    }

}
```

线程声明周期



线程声明周期分为两条线:

1. 基础线程声明周期：　新建 ---》 准备就绪 ---》 运行状态 ---》 死亡状态
2. 对线程添加了一些操作造成线程阻塞现象的产生【sleep、join、wait、synchronized等等】

新建 ---》 准备就绪 ---》 运行状态 ---》【 阻塞状态 ---》 准备就绪 ---》 运行状态】 ---》 死亡状态

PS：在阻塞状态没有消失之前

　　准备就绪 ---》 运行状态 ---》阻塞状态 ---》 准备就绪 ---》 运行状态 ..... 【会一直重复执行】

1:新建状态(new):使用new创建一个线程对象,仅仅在堆中分配内存空间,在调用start方法之前.

新建状态下,线程压根就没有启动,仅仅只是存在一个线程对象而已.

Thread t = new Thread();//此时t就属于新建状态

当新建状态下的线程对象调用了start方法,此时从新建状态进入可运行状态.

线程对象的start方法只能调用一次,否则报错:IllegalThreadStateException.

2:可运行状态(runnable):分成两种状态，ready和running。分别表示就绪状态和运行状态。

就绪状态:线程对象调用start方法之后,等待JVM的调度(此时该线程并没有运行).

运行状态:线程对象获得JVM调度,如果存在多个CPU,那么允许多个线程并行运行.

3:阻塞状态(blocked):正在运行的线程因为某些原因放弃CPU,暂时停止运行,就会进入阻塞状态.

此时JVM不会给线程分配CPU,直到线程重新进入就绪状态,才有机会转到运行状态.

阻塞状态只能先进入就绪状态,不能直接进入运行状态.

阻塞状态的两种情况:

1. :当A线程处于运行过程时,试图获取同步锁时,却被B线程获取.此时JVM把当前A线程存到对象的锁池中,A线程进入阻塞状态.
2. :当线程处于运行过程时,发出了IO请求时,此时进入阻塞状态.

4:等待状态(waiting)(等待状态只能被其他线程唤醒):此时使用的无参数的wait方法,

1. :当线程处于运行过程时,调用了wait()方法,此时JVM把当前线程存在对象等待池中.

5:计时等待状态(timed waiting)(使用了带参数的wait方法或者sleep方法)

1):当线程处于运行过程时,调用了wait(long time)方法,此时JVM把当前线程存在对象等待池中.

2):当前线程执行了sleep(long time)方法.

6:终止状态(terminated):通常称为死亡状态，表示线程终止.

1):正常执行完run方法而退出(正常死亡).

2):遇到异常而退出(出现异常之后,程序就会中断)(意外死亡).

线程一旦终止,就不能再重启启动,否则报错(IllegalThreadStateException).

在Thread类中过时的方法(因为存在线程安全问题,所以弃用了【真心好用，绝对不能用】):

void suspend() :暂停当前线程

void resume() :恢复当前线程

void stop() :结束当前线程