笔记总链接：http://bbs.itheima.com/thread-200600-1-1.html

5、多线程

5.1 多线程的概念

5.1.3 创建线程方式二：实现Runnable接口

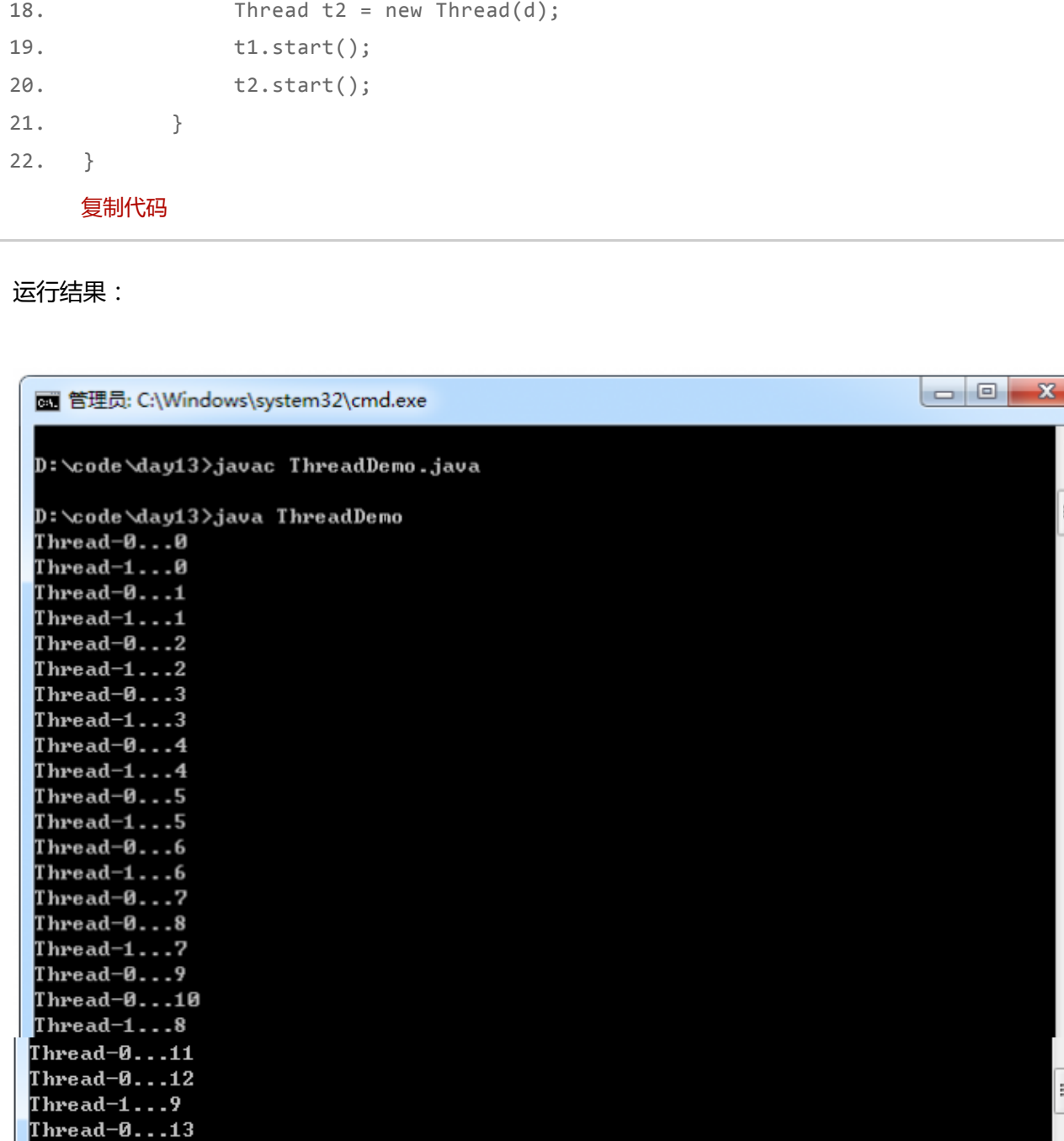
1. 定义类实现Runnable接口。
2. 覆盖接口中的run方法，将线程的任务代码封装到run方法中。
3. 通过Thread类创建线程对象，并将Runnable接口的子类对象作为Thread类的构造函数的参数进行传递。为什么？因为线程的任务都封装在Runnable接口子类对象的run方法中。所以要在线程对象创建时就必须明确要运行的任务。
4. 调用线程对象的start方法开启线程。

实现Runnable接口的好处：
1. 将线程的任务从线程的子类中分离出来，进行了单独的封装，按照面向对象的思想将任务封装成对象。
2. 避免了Java单继承的局限性。所以，创建线程的第二种方式较为常用。

示例：

```
//准备扩展Demo类的功能，让其中的内容可以作为线程的任务执行。
//通过接口的形式完成。
class Demo implements Runnable{
        public void run(){
                show();
        }
        public void show(){
                for(int x = 0; x < 20; x++){
                        System.out.println(Thread.currentThread().getName() + "..." + x);
                }
        }
}

class ThreadDemo{
        public static void main(String[] args){
                Demo d = new Demo();
                Thread t1 = new Thread(d);
                Thread t2 = new Thread(d);
                t1.start();
                t2.start();
        }
}
```
复制代码

运行结果：



Thread类、Runnable接口内部源码关系模拟代码：

```
class Thread{
        private Runnable r ;
        Thread(){
        }
        Thread(Runnable r){
                this.r = r;
        }

        public void run(){
                if(r !=null)
                        r.run();
        }
        public void start(){
                run();
        }
}

class ThreadImpl implements Runnable{
        public void run(){
                System.out.println("runnable run" );
        }
}

class ThreadDemo4{
        public static void main(String[] args){
                ThreadImpl i = new ThreadImpl();
                Thread t = new Thread(i);
                t.start();
        }
}

class SubThread extends Thread{
        public void run(){
                System.out.println("hahah" );
        }
}

class ThreadDemo5{
        public static void main(String[] args){
                SubThread s = new SubThread();
                s.start();
        }
}
```
复制代码

5.2 线程安全问题

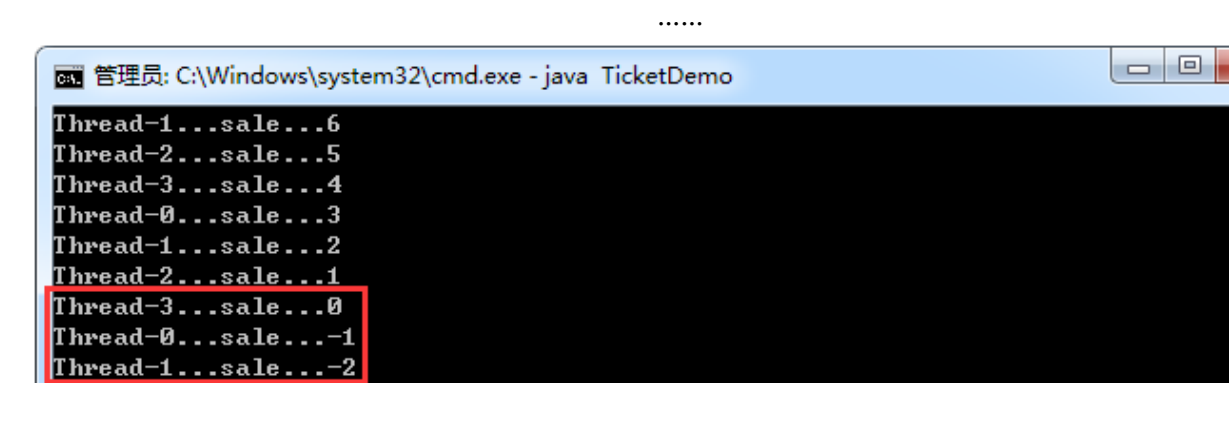
5.2.1 线程安全问题产生的原因

需求：模拟4个线程同时卖100张票。

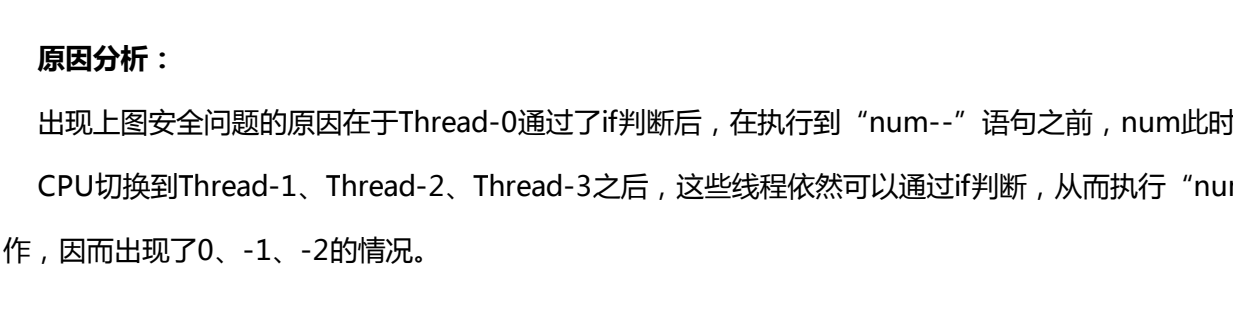
代码：

```
class Ticket implements Runnable{
        private int num = 100;

        public void run(){
                while(true ){
                        if(num > 0){
                                try{
                                        Thread. sleep(10);
                                } catch(InterruptedException e){
                                        e.printStackTrace();
                                }
                                System.out.println(Thread.currentThread().getName() +
"...sale..." + num--);
                        }
                }
        }
}

class TicketDemo{
        public static void main(String[] args){
                Ticket t = new Ticket();
                Thread t1 = new Thread(t);
                Thread t2 = new Thread(t);
                Thread t3 = new Thread(t);
                Thread t4 = new Thread(t);

                t1.start();
                t2.start();
                t3.start();
                t4.start();
        }
}
```
复制代码

运行结果：

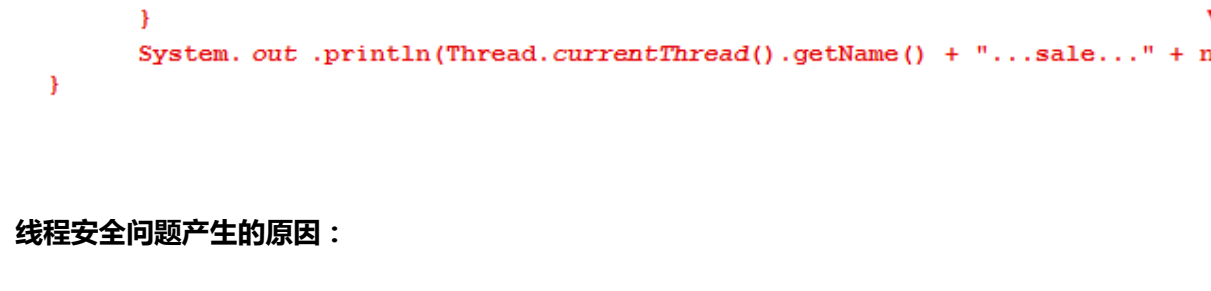


......



原因分析：

出现上图安全问题的原因在于Thread-0通过了if判断后，在执行到"num--"语句之前，num此时仍等于1。CPU切换到Thread-1、Thread-2、Thread-3之后，这些线程依然可以通过if判断，从而执行"num--"的操作，因而出现了0、-1、-2的情况。



线程安全问题产生的原因：

1. 多个线程在操作共享的数据。
2. 操作共享数据的线程代码有多条。

当一个线程在执行操作共享数据的多条代码过程中，其他线程参与了运算，就会导致线程安全问题的产生。

5.2.2 线程安全问题的解决方案

思路：

就是将多条操作共享数据的线程代码封装起来，当有线程在执行这些代码的时候，其他线程不可以参与运算。必须要当前线程把这些代码都执行完毕后，其他线程才可以参与运算。

在java中，用同步代码块就可以解决这个问题。

同步代码块的格式：

synchronized(对象){

需要被同步的代码;

}

同步的好处：解决了线程的安全问题。

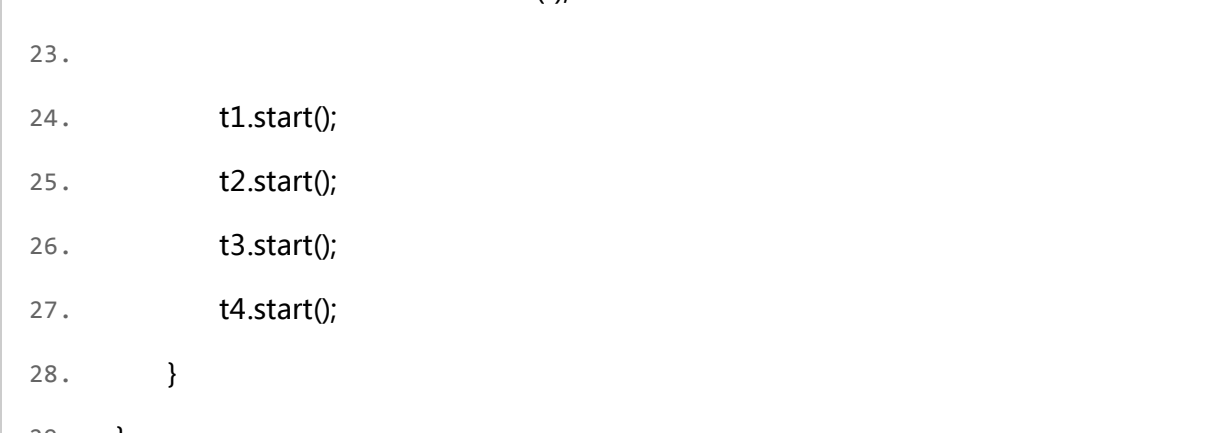
同步的弊端：当线程相当多时，因为每个线程都会去判断同步上的锁，这是很耗费资源的，无形中会降低程序的运行效率。

同步的前提：必须有多个线程并使用同一个锁。

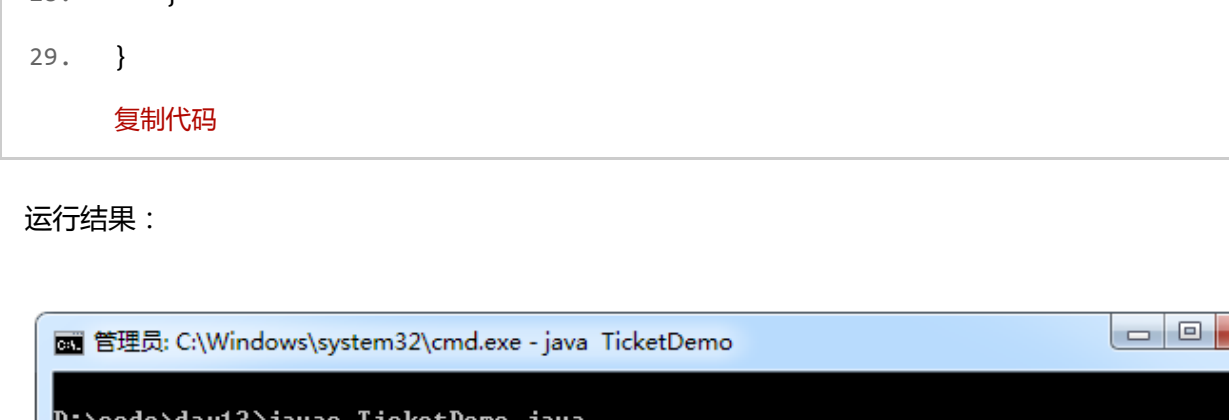
修改后代码：

```
class Ticket implements Runnable{
        private int num = 100;
        Object obj = new Object();

        public void run(){
                while(true ){
                        synchronized(obj ){
                                if(num > 0){
                                        System.out.println(Thread.currentThread().getName() + "...sale..." + num--);
                                }
                        }
                }
        }
}

class TicketDemo{
        public static void main(String[] args){
                Ticket t = new Ticket();
                Thread t1 = new Thread(t);
                Thread t2 = new Thread(t);
                Thread t3 = new Thread(t);
                Thread t4 = new Thread(t);

                t1.start();
                t2.start();
                t3.start();
                t4.start();
        }
}
```
复制代码

运行结果：

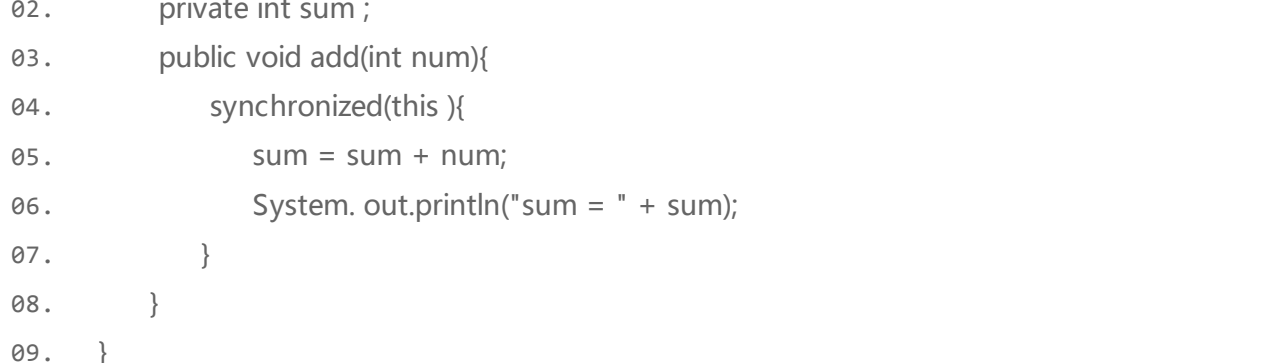


......



原因分析：

上图显示安全问题已被解决，原因在于Object对象相当于是一把锁，只有抢到锁的线程，才能进入同步代码块向下执行。因此，当num=1时，CPU切换到某个线程后，如上图的Thread-3线程，其他线程将无法通过同步代码块继而进行if判断语句，只有等到Thread-3线程执行完"num--"操作（此后num的值为0），并且跳出同步代码块后，才能抢到锁。其他线程即使抢到锁，然而，此时num值已为0，也就无法通过if语句判断，从而无法再执行"num--"的操作了，也就不会出现0、-1、-2等情况了。



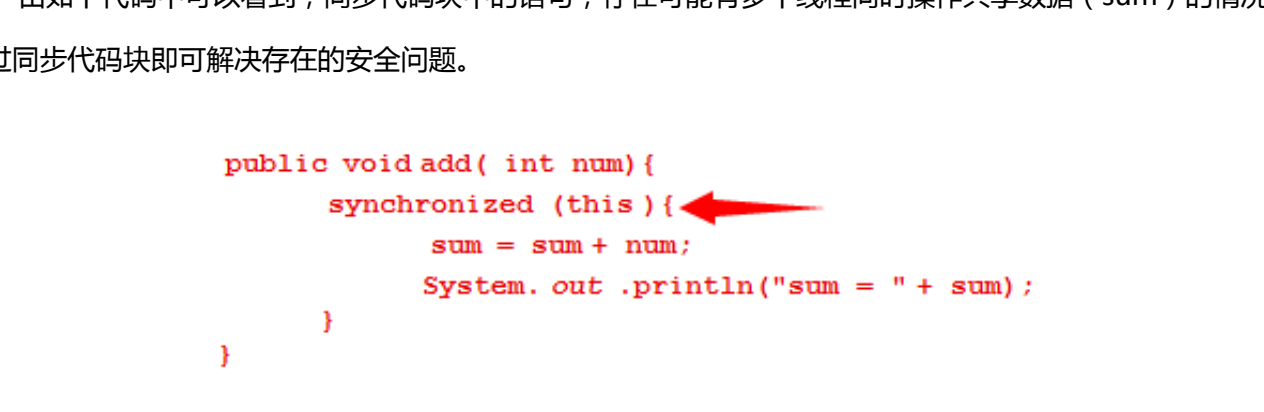
利用同步代码块解决安全问题案例：

需求：储户，两个，每个都到银行存钱，每次存100，共存三次。

代码：

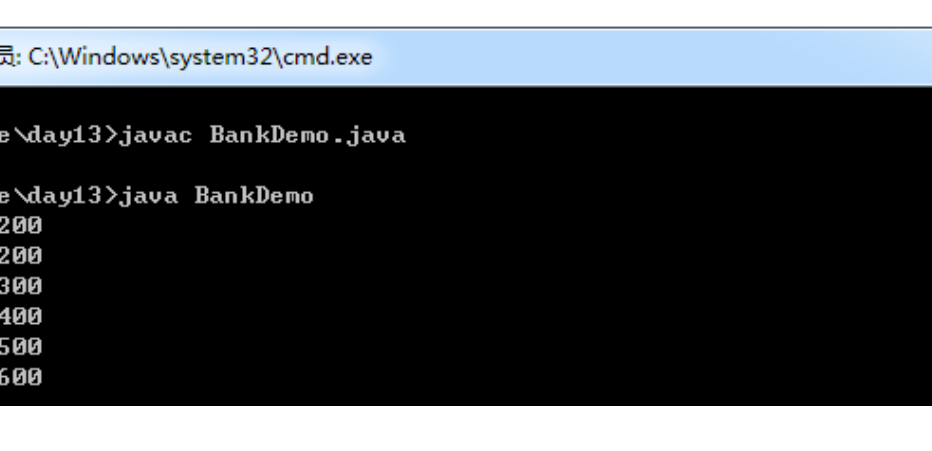
```
class Bank{
        private int sum;
        public void add(int num){
                synchronized(this ){
                        sum = sum + num;
                        System. out.println("sum = " + sum);
                }
        }
}

class Cus implements Runnable{
        private Bank b = new Bank();
        public void run(){
                for(int x = 0; x < 3; x++){
                        b.add(100);
                }
        }
}

class BankDemo{
        public static void main(String[] args){
                Cus c = new Cus();
                Thread t1 = new Thread(c);
                Thread t2 = new Thread(c);
                t1.start();
                t2.start();
        }
}
```
复制代码

运行结果：

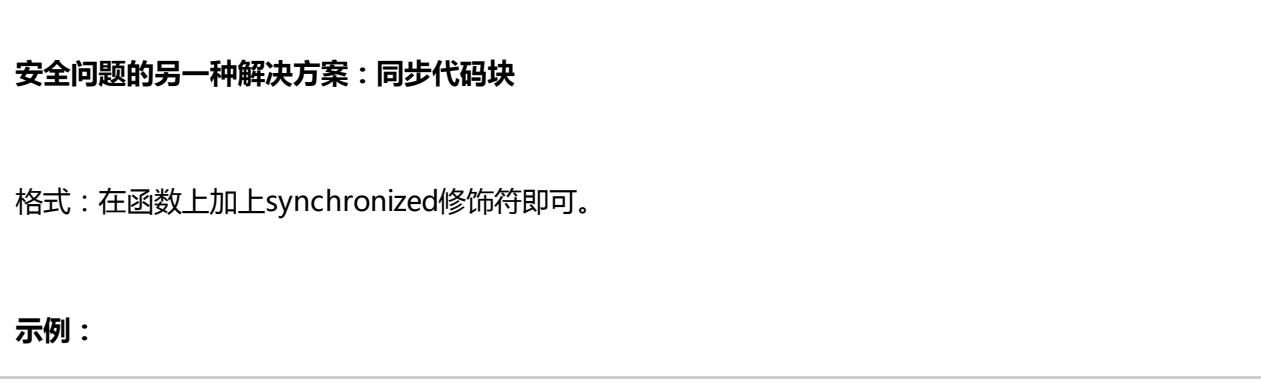


原因分析：

由如下代码中可以看到，同步代码块中的语句，存在可能有多个线程同时操作共享数据（sum）的情况，通过同步代码块即可解决存在的安全问题。



如果不设置同步代码块，出现的结果如下：



安全问题的另一种解决方案：同步代码块

格式：在函数上加上synchronized修饰符即可。

示例：

```
class Bank{
        private int sum;
        public synchronized void add(int num){ //同步函数
                sum = sum + num;
                System.out.println("sum = " + sum);
        }
}
```
复制代码

运行结果：

P.S.

同步函数和同步代码块的区别：
1. 同步函数的锁是固定的this。
2. 同步代码块的锁是任意的对象。

建议使用同步代码块。

由于同步函数的锁是固定的this，同步代码块的锁是任意的对象，那么如果同步函数和同步代码块都使用this作为锁，就可以实现同步。

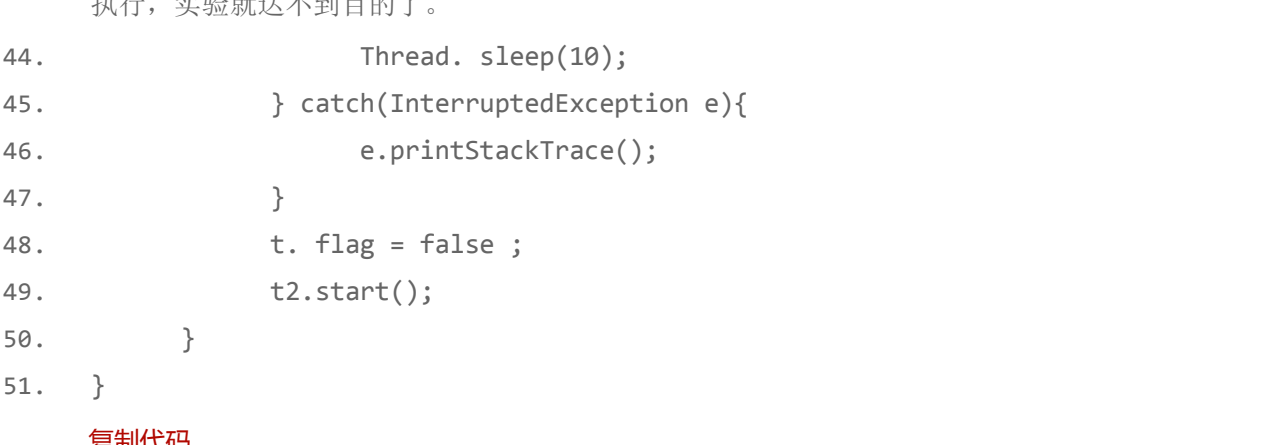
示例：

```
class Ticket implements Runnable{
        private int num = 100;
        boolean flag = true;

        public void run(){
                if(flag ){
                        while(true ){
                                synchronized(this ){
                                        if(num > 0){
                                                try{
                                                        Thread. sleep(10);
                                                } catch(InterruptedException e){
                                                        e.printStackTrace();
                                                }
                                                System.
```

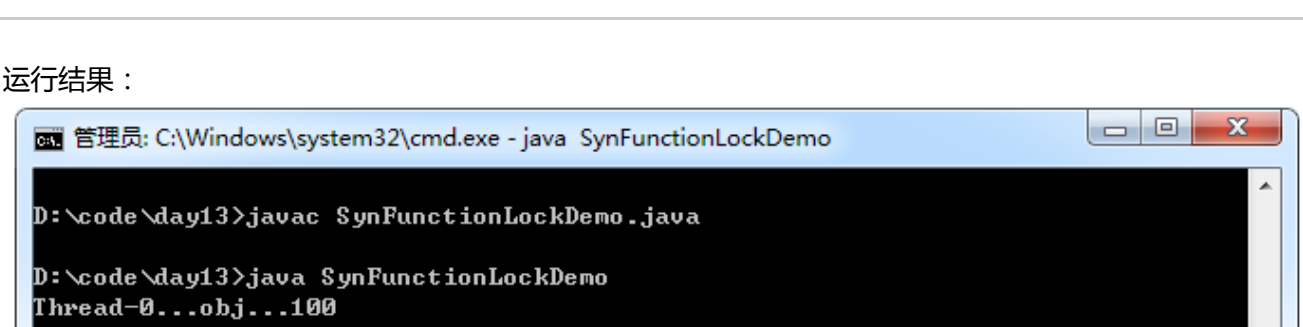
```
out.println(Thread.currentThread().getName() + "...obj..." + num--);
                                }
                         }
                  }
             } else
                    while(true )
                         show();
        }

        public synchronized void show(){
              if(num > 0){
                    try{
                         Thread. sleep(10);
                    } catch(InterruptedException e){
                         e.printStackTrace();
                    }
                    System.out.println(Thread.currentThread().getName() + "...function..." + num--);
             }
       }
}

class SynFunctionLockDemo{
       public static void main(String[] args){
             Ticket t = new Ticket();
             Thread t1 = new Thread(t);
             Thread t2 = new Thread(t);

             t1.start();
              try{//下面这条语句一定要执行。因为可能线程t1尚未真正启动，flag已经设置为false，那么当t1执行的时候，就会按照flag为false的情况执行，线程t2也按照flag为false的情况执行，实验就达不到目的了。
                   Thread. sleep(10);
             } catch(InterruptedException e){
                   e.printStackTrace();
             }
             t. flag = false ;
             t2.start();
       }
}
```

复制代码

运行结果：



......



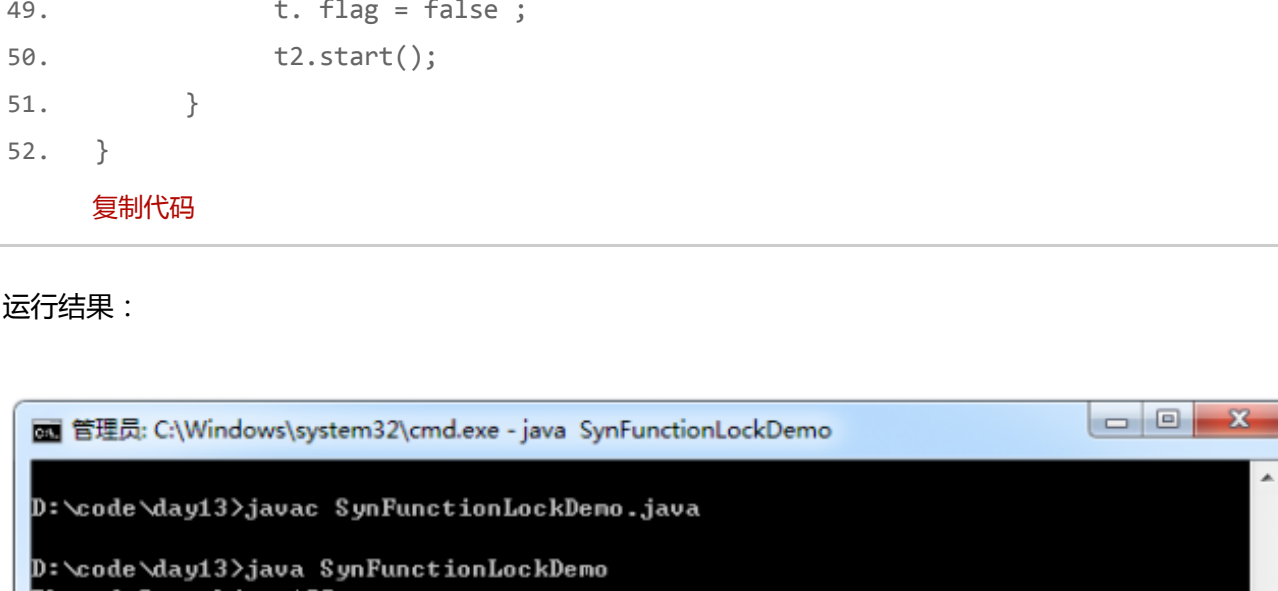
静态的同步函数使用的锁是该函数所属字节码文件对象，可以用getClass方法获取，也可以用当前类名.class表示。

示例：

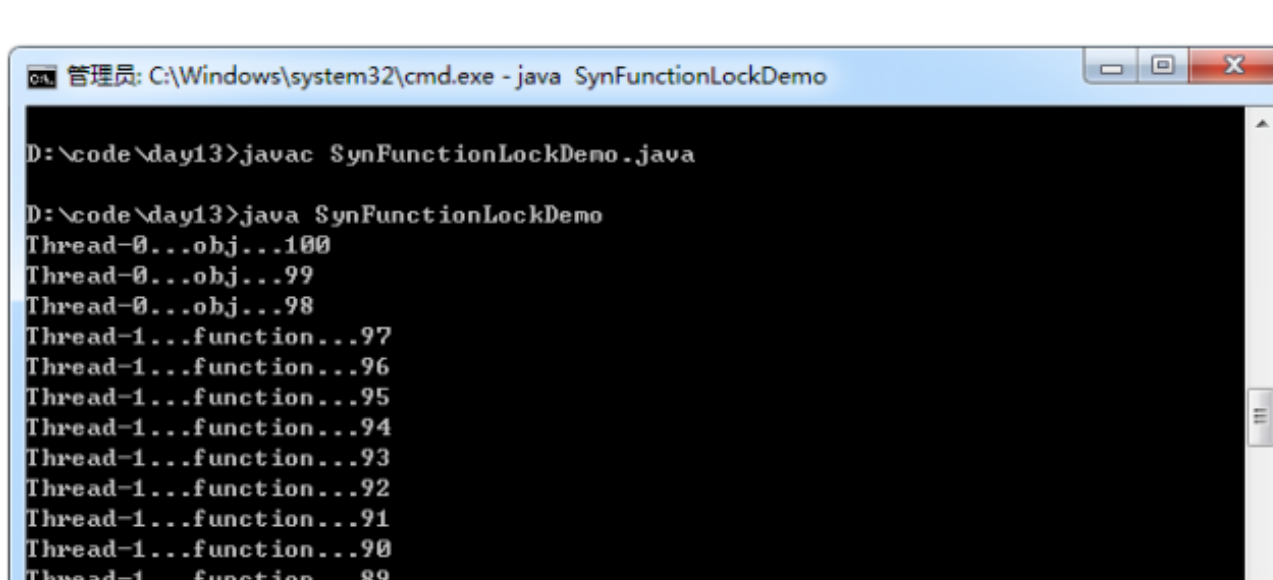
```
class Ticket implements Runnable{
        private static int num = 100;
       Object obj = new Object();
        boolean flag = true;

        public void run(){
              if(flag ){
                    while(true ){
                          synchronized(Ticket.class){//this.getClass()
                                if(num > 0){
                                      try{
                                           Thread. sleep(10);
                                      } catch(InterruptedException e){
                                           e.printStackTrace();
                                      }
System.out.println(Thread.currentThread().getName() + "...obj..." + num--);
                                }
                          }
                    }
             } else
                    while(true )
                          show();
       }

       public static synchronized void show(){
              if(num > 0){
                    try{
                         Thread. sleep(10);
                    } catch(InterruptedException e){
                         e.printStackTrace();
                    }
                    System.out.println(Thread.currentThread().getName() + "...function..." + num--);
             }
       }
}

class SynFunctionLockDemo{
        public static void main(String[] args){
             Ticket t = new Ticket();
             Thread t1 = new Thread(t);
             Thread t2 = new Thread(t);

             t1.start();
              try{
                   Thread. sleep(10);
             } catch(InterruptedException e){
                   e.printStackTrace();
             }
             t. flag = false ;
             t2.start();
       }
}
```

复制代码

运行结果：



......



5.2.3 多线程下的单例模式

饿汉式：

```
class Single{
        private static final Single s = new Single();
        private Single(){}
        public static Single getInstance(){
              return s ;
       }
}
```

复制代码

P.S.

饿汉式不存在安全问题，因为不存在多个线程共同操作数据的情况。

懒汉式：

```
class Single{
       private static Single s = null;
        private Single(){}
        public static Single getInstance(){
              if(s ==null){
                    synchronized(Single.class){
                          if(s == null)
                                s = new Single();
                   }
             }
             return s ;
       }
}
```

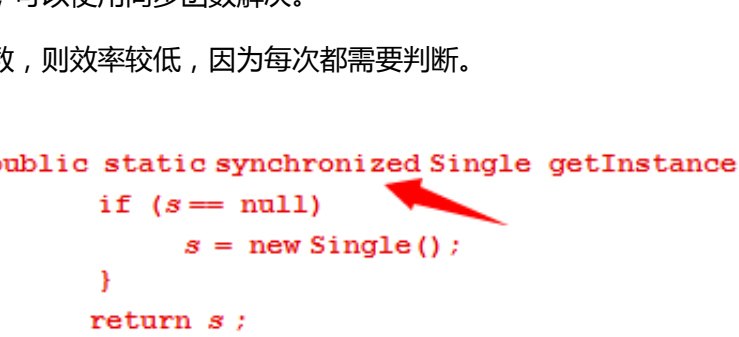
复制代码

P.S.

懒汉式存在安全问题，可以使用同步函数解决。

但若直接使用同步函数，则效率较低，因为每次都需要判断。

但若采取如下方式，即可提升效率。



原因在于任何一个线程在执行到第一个if判断语句时，如果Single对象已经创建，则直接获取即可，而不用判断是否能够获取锁，相对于上面使用同步函数的方法就提升了效率。如果当前线程发现Single对象尚未创建，则再判断是否能够获取锁。

1. 如果能够获取锁，那么就通过第二个if判断语句判断是否需要创建Single对象。因为可能当此线程获取到锁之前，已经有一个线程创建完Single对象，并且放弃了锁。此时它便没有必要再去创建，可以直接跳出同步代码块，放弃锁，获取Single对象即可。如果有必要，则再创建。

2. 如果不能获取到锁，则等待，直至能够获取到锁为止，再按步骤一执行。

5.2.4 死锁示例

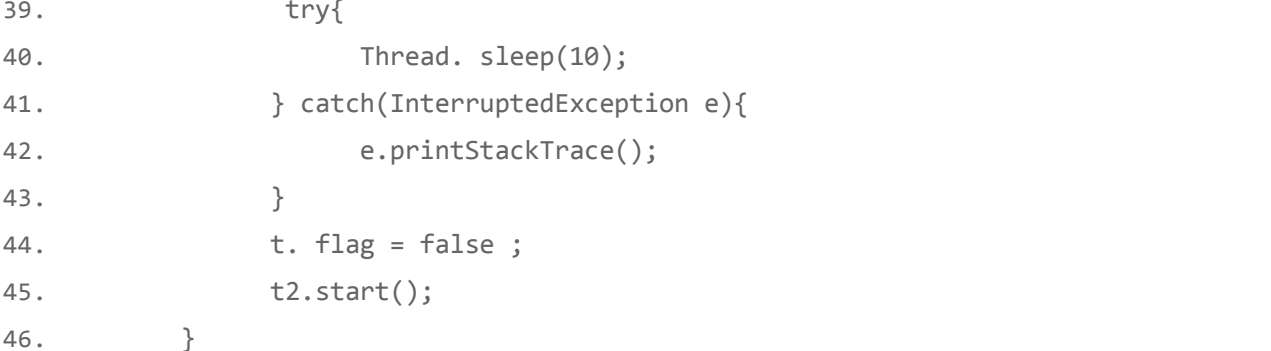
死锁常见情景之一：同步的嵌套。

示例1：

```
class Ticket implements Runnable{
        private static int num = 100;
        Object obj = new Object();
        boolean flag = true;

        public void run(){
              if(flag ){
                    while(true ){
                          synchronized(obj ){
                               show();
                         }
                   }
             } else
                    while(true )
                         show();
       }

        public synchronized void show(){
              synchronized(obj ){
                    if(num > 0){
                          try{
                               Thread. sleep(10);
                         } catch(InterruptedException e){
                               e.printStackTrace();
                         }
                         System.out.println(Thread.currentThread().getName() + "...function..." + num--);
                   }
             }
       }
}

class DeadLockDemo{
        public static void main(String[] args){
             Ticket t = new Ticket();
             Thread t1 = new Thread(t);
             Thread t2 = new Thread(t);

             t1.start();
              try{
                   Thread. sleep(10);
             } catch(InterruptedException e){
                   e.printStackTrace();
             }
             t. flag = false ;
             t2.start();
       }
}
```

复制代码

运行结果：



原因分析：

由上图可以看到程序已经被锁死，无法向下执行。

由下图代码可以看到，run方法中的同步代码块需要获取obj对象锁，才能执行代码块中的show方法。

而执行show方法则必须获取this对象锁，然后才能执行其中的同步代码块。

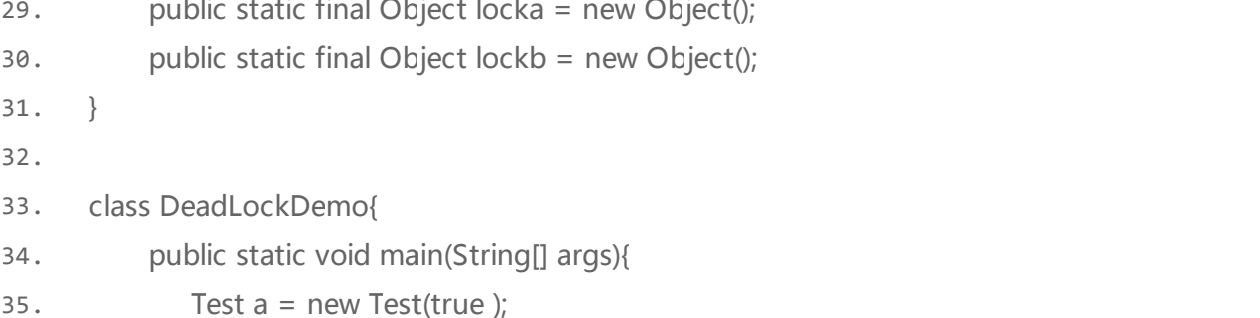
当线程t1获取到obj对象锁执行同步代码块，线程t2获取到this对象锁执行show方法。同步代码块中的show方法因无法获取到this对象锁无法执行，show方法中的同步代码块因无法获取到obj对象锁无法执行，就会产生死锁。

示例2：

```
class Test implements Runnable{
       private boolean flag ;
      Test( boolean flag){
            this.flag = flag;
      }

       public void run(){
             if(flag ){
                   while(true )
                         synchronized(MyLock.locka){
                               System.out.println(Thread.currentThread().getName() + "...if locka...");
                                synchronized(MyLock.lockb){
                                     System.out.println(Thread.currentThread().getName() + "...if lockb...");
                               }
                        }
             } else{
                   while(true )
                         synchronized(MyLock.lockb){
                               System.out.println(Thread.currentThread().getName() + "...else lockb...");
                                synchronized(MyLock.locka){
                                     System.out.println(Thread.currentThread().getName() + "...else locka...");
                        }
                   }
             }
      }
}

class MyLock{
       public static final Object locka = new Object();
       public static final Object lockb = new Object();
}

class DeadLockDemo{
       public static void main(String[] args){
            Test a = new Test(true );
            Test b = new Test(false );

            Thread t1 = new Thread(a);
            Thread t2 = new Thread(b);

            t1.start();
            t2.start();
      }
}
```

复制代码

运行结果：



~END~

~爱上海，爱黑马~