



非贷款，0元入学，不1万就业不给1分钱学费，我们已千四了！

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定义要运行的任务。  
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：

```
01. //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。
02. //通过接口的形式完成。
03. class Demo implements Runnable{
04.     public void run(){
05.         show();
06.     }
07.     public void show(){
08.         for(int x = 0; x < 20; x++){
09.             System.out.println(Thread.currentThread().getName() + "... " + x);
10.         }
11.     }
12. }
13.
14. class ThreadDemo{
15.     public static void main(String[] args){
16.         Demo d = new Demo();
17.         Thread t1 = new Thread(d);
18.         Thread t2 = new Thread(d);
19.         t1.start();
20.         t2.start();
21.     }
22. }
复制代码
```

运行结果：

```
管理员: C:\Windows\system32\cmd.exe
D:\code\day13>javac ThreadDemo.java
D:\code\day13>java ThreadDemo
Thread-0...0
Thread-1...0
Thread-0...1
Thread-1...1
Thread-0...2
Thread-1...2
Thread-0...3
Thread-1...3
Thread-0...4
Thread-1...4
Thread-0...5
Thread-1...5
Thread-0...6
Thread-1...6
Thread-0...7
Thread-1...7
Thread-0...8
Thread-1...8
Thread-0...9
Thread-1...9
Thread-0...10
Thread-1...10
Thread-0...11
Thread-1...11
Thread-0...12
Thread-1...12
Thread-0...13
Thread-1...13
Thread-0...14
Thread-1...14
Thread-0...15
Thread-1...15
Thread-0...16
Thread-1...16
Thread-0...17
Thread-1...17
Thread-0...18
Thread-1...18
Thread-0...19
Thread-1...19
D:\code\day13>
```

#### Thread类、Runnable接口内部源码关系模拟代码：

```
01. class Thread{
02.     private Runnable r ;
03.     Thread(){}
04.     Thread(Runnable r){
05.         this.r = r;
06.     }
07.
08.     public void run(){
09.         if(r !=null)
10.             r.run();
11.     }
12.     public void start(){
13.         run();
14.     }
15. }
16.
17. class ThreadImpl implements Runnable{
18.     public void run(){
19.         System.out.println("runnable run" );
20.     }
21. }
22.
23. class ThreadDemo{
24.     public static void main(String[] args){
25.         ThreadImpl i = new ThreadImpl();
26.         Thread t = new Thread(i);
27.         t.start();
28.     }
29. }
30.
31. class SubThread extends Thread{
32.     public void run(){
33.         System.out.println("hahah" );
34.     }
35. }
36.
37. class ThreadDemo5{
38.     public static void main(String[] args){
39.         SubThread s = new SubThread();
40.         s.start();
41.     }
42. }
43.
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：

```
01. class Ticket implements Runnable{
02.     private int num = 100;
03.
04.     public void run(){
05.         while(true ){
06.             if(num > 0){
07.                 try{
08.                     Thread.sleep(10);
09.                 } catch(InterruptedException e){
10.                     e.printStackTrace();
11.                 }
12.             System.out.println(Thread.currentThread().getName() +
13. "...sale..." + num--);
14.         }
15.     }
16.
17. class ThreadDemo{
18.     public static void main(String[] args){
19.         Ticket t = new Ticket();
20.         Thread t1 = new Thread(t);
21.         Thread t2 = new Thread(t);
22.         Thread t3 = new Thread(t);
23.         Thread t4 = new Thread(t);
24.
25.         t1.start();
26.         t2.start();
27.         t3.start();
28.         t4.start();
29.     }
30.
31. }
32.
33.复制代码
```

运行结果：

```
管理员: C:\Windows\system32\cmd.exe - java TicketDemo
D:\code\day13>javac TicketDemo.java
D:\code\day13>java TicketDemo
Thread-0...sale..100
Thread-1...sale..99
Thread-0...sale..99
Thread-2...sale..100
Thread-3...sale..98
Thread-1...sale..97
Thread-3...sale..96
Thread-3...sale..94
Thread-3...sale..93
.....
```

#### 原因分析：

上图线程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等情况。

```
if (num > 0) { ←
    try {
        Thread.sleep(10);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println(Thread.currentThread().getName() + "...sale..." + num--);
```

#### 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。

#### 修改后台代码：

```
01. class Ticket implements Runnable{
02.     private int num = 100;
03.
04.     public void run(){
05.         while(true ){
06.             if(num > 0){
07.                 try{
08.                     Thread.sleep(10);
09.                 } catch(InterruptedException e){
10.                     e.printStackTrace();
11.                 }
12.             System.out.println(Thread.currentThread().getName() +
13. "...sale..." + num--);
14.         }
15.     }
16.
17. class ThreadDemo{
18.     public static void main(String[] args){
19.         Ticket t = new Ticket();
20.         Thread t1 = new Thread(t);
21.         Thread t2 = new Thread(t);
22.         Thread t3 = new Thread(t);
23.         Thread t4 = new Thread(t);
24.
25.         t1.start();
26.         t2.start();
27.         t3.start();
28.         t4.start();
29.     }
30.
31. }
32.
33.复制代码
```

运行结果：

```
管理员: C:\Windows\system32\cmd.exe - java TicketDemo
D:\code\day13>javac TicketDemo.java
D:\code\day13>java TicketDemo
Thread-0...sale..100
Thread-1...sale..99
Thread-0...sale..99
Thread-2...sale..100
Thread-3...sale..98
Thread-1...sale..97
Thread-3...sale..96
Thread-3...sale..94
Thread-3...sale..93
.....
```

#### 原因分析：

上图线程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，从而无法再执行“num--”的操作了，也就不会出现0、-1、-2等情况。

```
if (num > 0) { ←
    try {
        Thread.sleep(10);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println(Thread.currentThread().getName() + "...sale..." + num--);
```

#### 线程安全问题产生的原因：

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。

#### 修改后台代码：

```
01. class Ticket implements Runnable{
02.     private int num = 100;
03.
04.     public void run(){
05.         while(true ){
06.             if(flag == true){
07.                 synchronized(this){
08.                     if(num > 0){
09.                         num--;
10.                         System.out.println(Thread.currentThread().getName() +
11. "...sale..." + num);
12.                     }
13.                 }
14.             }
15.         }
16.     }
17.
18. class ThreadDemo{
19.     public static void main(String[] args){
20.         Ticket t = new Ticket();
21.         Thread t1 = new Thread(t);
22.         Thread t2 = new Thread(t);
23.         Thread t3 = new Thread(t);
24.         Thread t4 = new Thread(t);
25.
26.         t1.start();
27.         t2.start();
28.         t3.start();
29.         t4.start();
30.     }
31.
32. }
33.
34.复制代码
```

运行结果：

```
管理员: C:\Windows\system32\cmd.exe - java TicketDemo
D:\code\day13>javac TicketDemo.java
D:\code\day13>java TicketDemo
Thread-0...sale..100
Thread-1...sale..99
Thread-0...sale..99
Thread-2...sale..100
Thread-3...sale..98
Thread-1...sale..97
Thread-3...sale..96
Thread-3...sale..94
Thread-3...sale..93
.....
```

#### 原因分析：

上图线程安全问题已被解决，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（此时num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。

```
if (flag == true){ ←
    synchronized(this){
        if (num > 0){
            num--;
            System.out.println(Thread.currentThread().getName() + "...sale..." + num);
        }
    }
}
```

#### 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。

#### 修改后台代码：

```
01. class Ticket implements Runnable{
02.     private int num = 100;
03.
04.     public void run(){
05.         while(true ){
06.             if(flag == true){
07.                 synchronized(this){
08.                     if(num > 0){
09.                         num--;
10.                         System.out.println(Thread.currentThread().getName() +
11. "...sale..." + num);
12.                     }
13.                 }
14.             }
15.         }
16.     }
17.
18. class ThreadDemo{
19.     public static void main(String[] args){
20.         Ticket t = new Ticket();
21.         Thread t1 = new Thread(t);
22.         Thread t2 = new Thread(t);
23.         Thread t3 = new Thread(t);
24.         Thread t4 = new Thread(t);
25.
26.         t1.start();
27.         t2.start();
28.         t3.start();
29.         t4.start();
30.     }
31.
32. }
33.
34.复制代码
```

运行结果：

```
管理员: C:\Windows\system32\cmd.exe - java TicketDemo
D:\code\day13>javac TicketDemo.java
D:\code\day13>java TicketDemo
Thread-0...sale..100
Thread-1...sale..99
Thread-0...sale..99
Thread-2...sale..100
Thread-3...sale..98
Thread-1...sale..97
Thread-3...sale..96
Thread-3...sale..94
Thread-3...sale..93
.....
```

#### 原因分析：

上图线程安全问题已被解决，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（此时num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。

```
if (flag == true){ ←
    synchronized(this){
        if (num > 0){
            num--;
            System.out.println(Thread.currentThread().getName() + "...sale..." + num);
        }
    }
}
```

#### 线程安全问题产生的原因：

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。

#### 修改后台代码：

```
01. class Bank{
02.     private int sum = 100;
03.
04.     public void add(int num){
05.         synchronized(this){
06.             if(num > 0){
07.                 sum = sum + num;
08.                 System.out.println("sum = " + sum);
09.             }
10.         }
11.     }
12.
13.     class Cus implements Runnable{
14.         private Bank b = new Bank();
15.         public void run(){
16.             for(int x = 0; x < 100; x++){
17.                 b.add(100);
18.             }
19.         }
20.     }
21.
22.     class BankDemo{
23.         public static void main(String[] args){
24.             Cus c = new Cus();
25.             Thread t1 = new Thread(c);
26.             Thread t2 = new Thread(c);
27.             Thread t3 = new Thread(c);
28.             Thread t4 = new Thread(c);
29.
30.             t1.start();
31.             t2.start();
32.             t3.start();
33.             t4.start();
34.         }
35.     }
36.
37. }
38.
39.复制代码
```

运行结果：

```
管理员: C:\Windows\system32\cmd.exe - java BankDemo
D:\code\day13>javac BankDemo.java
D:\code\day13>java BankDemo
sum = 100
sum = 200
sum = 300
sum = 400
sum = 500
sum = 600
.....
```

#### 原因分析：

上图线程安全问题已被解决，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=100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“sum+=num”操作（此时sum值为600），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sum值为600，也就无法通过if语句判断，从而无法再执行“sum+=num”操作了，也就不会出现0、-1、-2等情况。

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

#### 线程安全问题产生的原因：

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。

#### 修改后台代码：

```
01. class Bank implements Runnable{
02.     private int num = 100;
03.     boolean flag = true;
04.
05.     public void run(){
06.         if(flag == true){
07.             synchronized(this){
08.                 if(num > 0){
09.                     num--;
10.                     System.out.println(Thread.currentThread().getName() + "...sale..." + num);
11.                 }
12.             }
13.         }
14.     }
15.
16.     class Cus implements Runnable{
17.         private Bank b = new Bank();
18.         public void run(){
19.             while(true ){
20.                 synchronized(this){
21.                     if(num > 0){
22.                         num--;
23.                         System.out.println(Thread.currentThread().getName() + "...sale..." + num);
24.                     }
25.                 }
26.             }
27.         }
28.     }
29.
30.     class BankDemo{
31.         public static void main(String[] args){
32.             Cus c = new Cus();
33.             Thread t1 = new Thread(c);
34.             Thread t2 = new Thread(c);
35.             Thread t3 = new Thread(c);
36.             Thread t4 = new Thread(c);
37.
38.             t1.start();
39.             t2.start();
40.             t3.start();

```

```
out.println(Thread.currentThread().getName() + "...obj..." + num--);
```

```
16.         }
17.     }
18. }
19. } else
20.     while(true)
21.         show();
22. }
23.
24. public synchronized void show(){
25.     if(num > 0){
26.         try{
27.             Thread.sleep(10);
28.         } catch(InterruptedException e){
29.             e.printStackTrace();
30.         }
31.         System.out.println(Thread.currentThread().getName() +
32.             "...function..." + num--);
33.     }
34. }
```

```
35.
36. class SynFunctionLockDemo{
37.     public static void main(String[] args){
38.         Ticket t = new Ticket();
39.         Thread t1 = new Thread(t);
40.         Thread t2 = new Thread(t);
41.
42.         t1.start();
43.         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
44.             false，那么当t1执行的时候，会按照flag为false的情况执行，线程t2也按照flag为false的情况
45.             Thread.sleep(10);
46.         } catch(InterruptedException e){
47.             e.printStackTrace();
48.         }
49.         t.flag = false ;
50.         t2.start();
51.     }
52. }
```

复制代码

运行结果：

```
管理员: C:\Windows\system32\cmd.exe - java SynFunctionLockDemo
D:\code\day13>java SynFunctionLockDemo.java
D:\code\day13>java SynFunctionLockDemo
Thread-0...obj...100
Thread-0...obj...99
Thread-0...obj...98
Thread-0...obj...97
Thread-0...obj...96
Thread-0...obj...95
Thread-0...obj...94
Thread-0...obj...93
Thread-0...obj...92
Thread-0...obj...91
.....
管理员: C:\Windows\system32\cmd.exe - java SynFunctionLockDemo
D:\code\day13>java SynFunctionLockDemo
Thread-0...obj...10
Thread-0...obj...9
Thread-0...obj...8
Thread-0...obj...7
Thread-0...obj...6
Thread-0...obj...5
Thread-0...obj...4
Thread-0...obj...3
Thread-0...obj...2
Thread-0...obj...1
```

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：

```
01. class Ticket implements Runnable{
02.     private static int num = 100;
03.     Object obj = new Object();
04.     boolean flag = true;
05.
06.     public void run(){
07.         if(flag){
08.             while(true ){
09.                 synchronized(Ticket.class){//this.getClass()
10.                     if(num > 0){
11.                         try{
12.                             Thread.sleep(10);
13.                         } catch(InterruptedException e){
14.                             e.printStackTrace();
15.                         }
16.
17.                     System.out.println(Thread.currentThread().getName() + "...obj..." + num--);
18.                 }
19.             }
20.         } else
21.             while(true )
22.                 show();
23.     }
24.
25.     public static synchronized void show(){
26.         if(num > 0){
27.             try{
28.                 Thread.sleep(10);
29.             } catch(InterruptedException e){
30.                 e.printStackTrace();
31.             }
32.             System.out.println(Thread.currentThread().getName() +
33.                 "...function..." + num--);
34.         }
35.     }
36.
37. class SynFunctionLockDemo{
38.     public static void main(String[] args){
39.         Ticket t = new Ticket();
40.         Thread t1 = new Thread(t);
41.         Thread t2 = new Thread(t);
42.
43.         t1.start();
44.         try{
45.             Thread.sleep(10);
46.         } catch(InterruptedException e){
47.             e.printStackTrace();
48.         }
49.         t.flag = false ;
50.         t2.start();
51.     }
52. }
```

复制代码

运行结果：

```
管理员: C:\Windows\system32\cmd.exe - java SynFunctionLockDemo
D:\code\day13>java SynFunctionLockDemo.java
D:\code\day13>java SynFunctionLockDemo
Thread-0...obj...100
Thread-0...obj...99
Thread-0...obj...98
Thread-0...obj...97
Thread-0...obj...96
Thread-0...obj...95
Thread-0...obj...94
Thread-0...obj...93
Thread-0...obj...92
Thread-0...obj...91
Thread-0...obj...90
Thread-0...obj...89
Thread-0...obj...88
Thread-0...obj...87
Thread-0...obj...86
Thread-0...obj...85
Thread-0...obj...84
Thread-0...obj...83
Thread-0...obj...82
Thread-0...obj...81
Thread-0...obj...80
Thread-0...obj...79
.....
管理员: C:\Windows\system32\cmd.exe - java SynFunctionLockDemo
D:\code\day13>java SynFunctionLockDemo
Thread-1...function...14
Thread-1...function...13
Thread-1...function...12
Thread-1...function...11
Thread-1...function...10
Thread-1...function...9
Thread-1...function...8
Thread-1...function...7
Thread-1...function...6
Thread-1...function...5
Thread-1...function...4
Thread-1...function...3
Thread-1...function...2
Thread-1...function...1
```

5.2.3 多线程下的单例模式

饿汉式：

```
01. class Single{
02.     private static final Single s = new Single();
03.     private Single(){}
04.     public static Single getInstance(){
05.         return s ;
06.     }
07. }
```

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：

```
01. class Single{
02.     private static Single s = null;
03.     private Single(){}
04.     public static Single getInstance(){
05.         if(s ==null){
06.             synchronized(Single.class){
07.                 if(s == null)
08.                     s = new Single();
09.             }
10.         }
11.         return s ;
12.     }
13. }
```

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。

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

但若采取如下方式，则可提升效率。

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

原因在于任何一个线程在执行到第一个判断语句时，如果Single对象已经创建，则直接获取即可，而不用判断是否能获得锁，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. 如果不能够获取到锁，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：

```
01. class Ticket implements Runnable{
02.     private static int num = 100;
03.     Object obj = new Object();
04.     boolean flag = true;
05.
06.     public void run(){
07.         if(flag){
08.             while(true ){
09.                 synchronized(obj ) {
10.                     if(num > 0){
11.                         try{
12.                             Thread.sleep(10);
13.                         } catch(InterruptedException e){
14.                             e.printStackTrace();
15.                         }
16.
17.                     System.out.println(Thread.currentThread().getName() +
18.                         "...function..." + num--);
19.                 }
20.             }
21.         } else
22.             while(true )
23.                 show();
24.     }
25.
26.     public synchronized void show(){
27.         synchronized(obj ) {
28.             if(num > 0){
29.                 try{
30.                     Thread.sleep(10);
31.                     } catch(InterruptedException e){
32.                         e.printStackTrace();
33.                     }
34.                     System.out.println(Thread.currentThread().getName() +
35.                         "...function..." + num--);
36.                 }
37.             }
38.     }
39.
40.     class DeadLockDemo{
41.         public static void main(String[] args){
42.             Ticket t = new Ticket();
43.             Thread t1 = new Thread(t);
44.             Thread t2 = new Thread(t);
45.
46.             t1.start();
47.             try{
48.                 Thread.sleep(10);
49.             } catch(InterruptedException e){
50.                 e.printStackTrace();
51.             }
52.             t.flag = false ;
53.             t2.start();
54.         }
55.     }
56. }
```

复制代码

运行结果：

```
管理员: C:\Windows\system32\cmd.exe - java DeadLockDemo
D:\code\day13>javac DeadLockDemo.java
D:\code\day13>java DeadLockDemo
Thread-0...obj...100
Thread-0...obj...99
Thread-0...obj...98
Thread-0...obj...97
Thread-0...obj...96
Thread-0...obj...95
Thread-0...obj...94
Thread-0...obj...93
Thread-0...obj...92
Thread-0...obj...91
Thread-0...obj...90
Thread-0...obj...89
Thread-0...obj...88
Thread-0...obj...87
Thread-0...obj...86
Thread-0...obj...85
Thread-0...obj...84
Thread-0...obj...83
Thread-0...obj...82
Thread-0...obj...81
Thread-0...obj...80
Thread-0...obj...79
.....
管理员: C:\Windows\system32\cmd.exe - java DeadLockDemo
D:\code\day13>java DeadLockDemo
Thread-0...if lock...
Thread-1...else lock...
```

原因分析：

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执行到obj对象锁时，线程t2执行到this对象锁，线程t2无法执行，就会产生死锁。

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。

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

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--);
 }
 }
}

但若采取如下方式，则可提升效率。

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

原因在于任何一个线程在执行到第一个判断语句时，如果Single对象已经创建，则直接获取即可，而不用判断是否能获得锁，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. 如果不能够获取到锁，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2：

```
01. class Test implements Runnable{
02.     private boolean flag;
03.     Test(boolean flag){
04.         this.flag = flag;
05.     }
06.     public void run(){
07.         synchronized(MyLock.locka){
08.             if(flag){
09.                 while(true )
10.                     synchronized(obj ) {
11.                         if(num > 0){
12.                             try{
13.                                 Thread.sleep(10);
14.                             } catch(InterruptedException e){
15.                                 e.printStackTrace();
16.                             }
17.                             System.out.println(Thread.currentThread().getName() +
18.                                 "...locka..." );
19.                         }
20.                     }
21.             }
22.         }
23.     }
24. }
```

复制代码

运行结果：

```
管理员: C:\Windows\system32\cmd.exe - java DeadLockDemo
D:\code\day13>javac DeadLockDemo.java
D:\code\day13>java DeadLockDemo
Thread-0...if lock...
Thread-1...else lock...
```

原因分析：

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执行到obj对象锁时，线程t2执行到this对象锁，线程t2无法执行，就会产生死锁。

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。

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

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

但若采取如下方式，则可提升效率。

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

原因在于任何一个线程在执行到第一个判断语句时，如果Single对象已经创建，则直接获取即可，而不用判断是否能获得锁，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. 如果不能够获取到锁，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3：

```
01. class Ticket implements Runnable{
02.     private static int num = 100;
03.     Object obj = new Object();
04.     boolean flag = true;
05.
06.     public void run(){
07.         if(flag){
08.             while(true ){
09.                 synchronized(obj ) {
10.                     if(num > 0){
11.                         try{
12.                             Thread.sleep(10);
13.                         } catch(InterruptedException e){
14.                             e.printStackTrace();
15.                         }
16.
17.                     System.out.println(Thread.currentThread().getName() +
18.                         "...obj..." + num--);
19.                 }
20.             }
21.         } else
22.             while(true )
23.                 show();
24.     }
25.
26.     public synchronized void show(){
27.         synchronized(obj ) {
28.             if(num > 0){
29.                 try{
30.                     Thread.sleep(10);
31.                     } catch(InterruptedException e){
32.                         e.printStackTrace();
33.                     }
34.                     System.out.println(Thread.currentThread().getName() +
35.                         "...function..." + num--);
36.                 }
37.             }
38.     }
39.
40.     class DeadLockDemo{
41.         public static void main(String[] args){
42.             Ticket t = new Ticket();
43.             Thread t1 = new Thread(t);
44.             Thread t2 = new Thread(t);
45.
46.             t1.start();
47.             try{
48.                 Thread.sleep(10);
49.             } catch(InterruptedException e){
50.                 e.printStackTrace();
51.             }
52.             t.flag = false ;
53.             t2.start();
54.         }
55.     }
56. }
```

复制代码

运行结果：

```
管理员: C:\Windows\system32\cmd.exe - java DeadLockDemo
D:\code\day13>javac DeadLockDemo.java
D:\code\day13>java DeadLockDemo
Thread-0...obj...100
Thread-0...obj...99
Thread-0...obj...98
Thread-0...obj...97
Thread-0...obj...96
Thread-0...obj...95
Thread-0...obj...94
Thread-0...obj...93
Thread-0...obj...92
Thread-0...obj...91
Thread-0...obj...90
Thread-0...obj...89
Thread-0...obj...88
Thread-0...obj...87
Thread-0...obj...86
Thread-0...obj...85
Thread-0...obj...84
Thread-0...obj...83
Thread-0...obj...82
Thread-0...obj...81
Thread-0...obj...80
Thread-0...obj...79
.....
管理员: C:\Windows\system32\cmd.exe - java DeadLockDemo
D:\code\day13>java DeadLockDemo
Thread-0...if lock...
Thread-1...else lock...
```

原因分析：

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执行到obj对象锁时，线程t2执行到this对象锁，线程t2无法执行，就会产生死锁。

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。

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

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--);
 }
 }
}

但若采取如下方式，则可提升效率。

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

原因在于任何一个线程在执行到第一个判断语句时，如果Single对象已经创建，则直接获取即可，而不用判断是否能获得锁，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。