在Java多线程处理数据时，我们经常会遇到出现异常数据的情况，先看下面的一段程序。

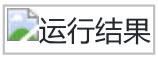
```
// 模拟三个黄牛同时进行火车票的售卖

class MyThread implements Runnable {
    // 车票总数
    private int ticket = 10;
    @Override
    public void run() {
        // 当还有余票时进行售卖
        while(this.ticket>0) {
            try {
                // 使用 sleep 模拟买票操作时的延迟
                Thread.sleep(200);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            // 售票动作
            System.out.println(Thread.currentThread().getName()+"还剩"+this.ticket--+"张票");
        }
    }

}

public class Test {
    public static void main(String[] args) {
        MyThread myThread = new MyThread();
        new Thread(myThread,"黄牛A").start();
        new Thread(myThread,"黄牛B").start();
        new Thread(myThread,"黄牛C").start();
    }
}
```

运行结果：



我们会发现，运行结果里竟然会出现 -1 这个本不可能出现的结果。

这种结果的出现我们称之为不同步操作。虽然三个线程共用一个 ticket，但是存在sleep方法模拟的网络延迟，在这段阻塞时间内，可能会同时有多个线程拿到当前的 ticket，然后进行处理。

例如，当ticket为1时，按理说只能在进入一次 while循环。这时黄牛A进入while循环，拿到了这个ticket，这时，它先要进行阻塞200毫秒，在这期间，可能会有线程也进入while循环，这时他们拿到的ticket仍然是1，因为黄牛A正在阻塞，他还没有进行ticket的改变，然后黄牛B拿着这个ticket也进行了阻塞。他们这时都处于while循环内，所以都可以进行售票，自然，结果就是一个 0，一个 -1了。

这种不同步操作又称异步处理，它具有效率高的特点，但是线程不安全，会产生不正确的数据。

虽然异步操作具有效率高的特点，但是程序设计必须是安全第一，所以我们要研究安全的操作，即同步处理。

同步处理

同步处理指的是所有线程不是一起进入到方法种执行，而是按照顺序一个一个进入。效率较低，但是线程安全，不会产生不正确数据。

1. synchronized 关键字处理同步问题

synchronized关键字进行同步处理可分为两种方式，同步代码块和同步方法

1.1 synchronized同步块

使用同步代码块必须设置一个要锁定的对象，同时只能有一个锁定的对象进入同步代码块，一般是锁定当前对象：this。

```
// 同步代码块在方法里定义，()内表示上锁对象
synchronized(this) {
    // statement
}
```

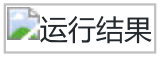
使用同步代码块改写卖票程序：

```
class MyThread implements Runnable {

    private int ticket = 10;
    @Override
    public void run() {
        while(this.ticket > 0) {
            synchronized (this) {
                if(this.ticket>0) {
                    try {
                        Thread.sleep(200);
                    } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                    System.out.println(Thread.currentThread().getName()+"还剩"+this.ticket--+"张
票");
                }
            }
        }
    }
}

public class Test {
    public static void main(String[] args) {
        MyThread myThread = new MyThread();
        new Thread(myThread,"黄牛A").start();
        new Thread(myThread,"黄牛B").start();
        new Thread(myThread,"黄牛C").start();
    }
}
```

运行结果：



1.2 synchronized同步方法

虽然同步代码块已经能够解决一些问题了，但由于代码块是位于方法内，不可避免同时有很多的线程进入方法，如果我们想让同时只有一个线程进入方法，可以使用同步方法。

同步方法定义如下：

```
// synchronized关键字位于返回值之前
public synchronized void method();
```

使用同步方法改写卖票程序。

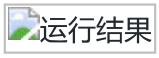
```
class MyThread implements Runnable {

    private int ticket = 10;
    @Override
    public void run() {
        while(this.ticket > 0) {
            this.sale();
        }
    }

    public synchronized void sale() {
        if(this.ticket > 0) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName()+"还剩"+this.ticket--);
        }
    }
}

public class Test {
    public static void main(String[] args) {
        MyThread myThread = new MyThread();
        new Thread(myThread,"黄牛A").start();
        new Thread(myThread,"黄牛B").start();
        new Thread(myThread,"黄牛C").start();
    }
}
```

运行结果：



同步代码块和同步方法虽然能使多线程更加安全，保持程序的完整性，但是会使程序运行速度变慢，效率变低。

2. Lock实现同步

处理同步问题，除了使用 synchronized 之外，也可以使用JDK提供的 Lock锁。

我们使用Lok来实现一下卖票程序。

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

class MyThread implements Runnable {

    private int ticket = 500;
    // 定义锁
    private Lock ticketLock = new ReentrantLock();

    @Override
    public void run() {
        // TODO Auto-generated method stub
        for(int i = 0; i < 500; i++) {
            // 打开锁
            ticketLock.lock();
            // 这里使用try-finally结构是为了 防止程序异常退出而没有关闭锁
            // 因为finally块无论如何也会执行，所以可以避免这个问题
            try {
                if(this.ticket > 0) {
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                    System.out.println(Thread.currentThread().getName()+"还剩"+this.ticket--);
                }
            } finally {
                // 无论try块中发生什么，最后一定会执行到这里
                ticketLock.unlock();
            }
        }
    }

}

public class Test {
    public static void main(String[] args) {
        MyThread myThread = new MyThread();
        new Thread(myThread,"黄牛A").start();
        new Thread(myThread,"黄牛B").start();
        new Thread(myThread,"黄牛C").start();
    }
}
```

Lock是一个对象锁。

在JDK1.5中，synchronized是性能低效的。因为这是一个重量级操作，它对性能最大的影响是阻塞的是实现，挂起线程和恢复线程的操作都需要转入内核态中完成，这些操作给系统的并发性带来了很大的压力。相比之下使用Java提供的Lock对象，性能更高一些。

到了JDK1.6，发生了变化，对synchronize加入了很多优化措施，有自适应自旋，锁消除，锁粗化，轻量级锁，偏向锁等等。导致在JDK1.6上synchronize的性能并不比Lock差。官方也表示，他们也更支持synchronize，在未来的版本中还有优化余地，所以还是提倡在synchronized能实现需求的情况下，优先考虑使用synchronized来进行同步。