1. **线程的同步与死锁**

在多线程的处理之中，可以利用Runnable描述多个线程的资源，而Thread描述我们每一个线程对象，而当多个线程访问同一个资源的时候，如果处理不当就会产生数据的错误操作；

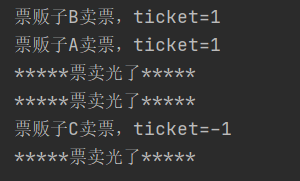
·同步问题的引出

下面我们编写一个简单的买票操作；

|  |
| --- |
| package com.company;  /\*\*  \* ticket表示票数为10  \*/  class MyThread implements Runnable{  private int ticket = 10 ;  @Override  public void run() {  while (true){  if (this.ticket > 0){  System.out.println(Thread.currentThread().getName() + "卖票，ticket=" + this.ticket);  this.ticket-- ;  }else {  System.out.println("\*\*\*\*\*票卖光了\*\*\*\*\*");  break;  }  }  }  }  public class Ticket {  public static void main(String[] args) {  MyThread mt = new MyThread();  new Thread(mt , "票贩子A").start();  new Thread(mt , "票贩子B").start();  new Thread(mt , "票贩子C").start();  }  } |

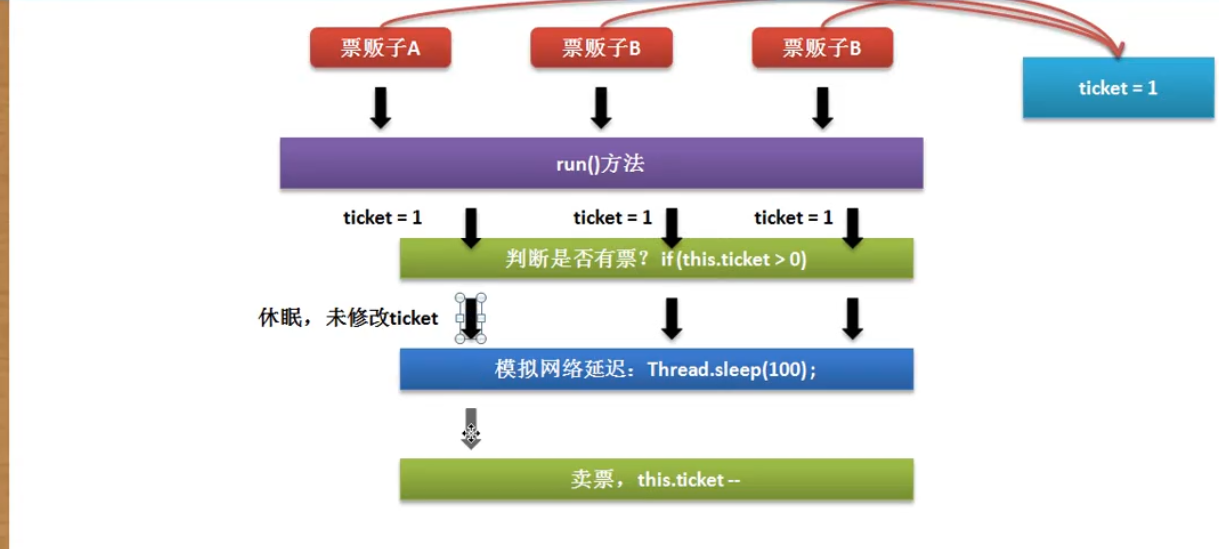
此时我们创建了三个线程，并且这三个线程访问同一个资源ticket，进行10张票的销售。此时的程序在进行我们的买票处理的时候并没有任何问题，但是这是一个假象，我们可以模拟一下延迟操作；（修改买票操作）

|  |
| --- |
| while (true){  if (this.ticket > 0){  try {  Thread.sleep(100);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println(Thread.currentThread().getName() + "卖票，ticket=" + this.ticket);  this.ticket-- ;  }else {  System.out.println("\*\*\*\*\*票卖光了\*\*\*\*\*");  break;  }  } |



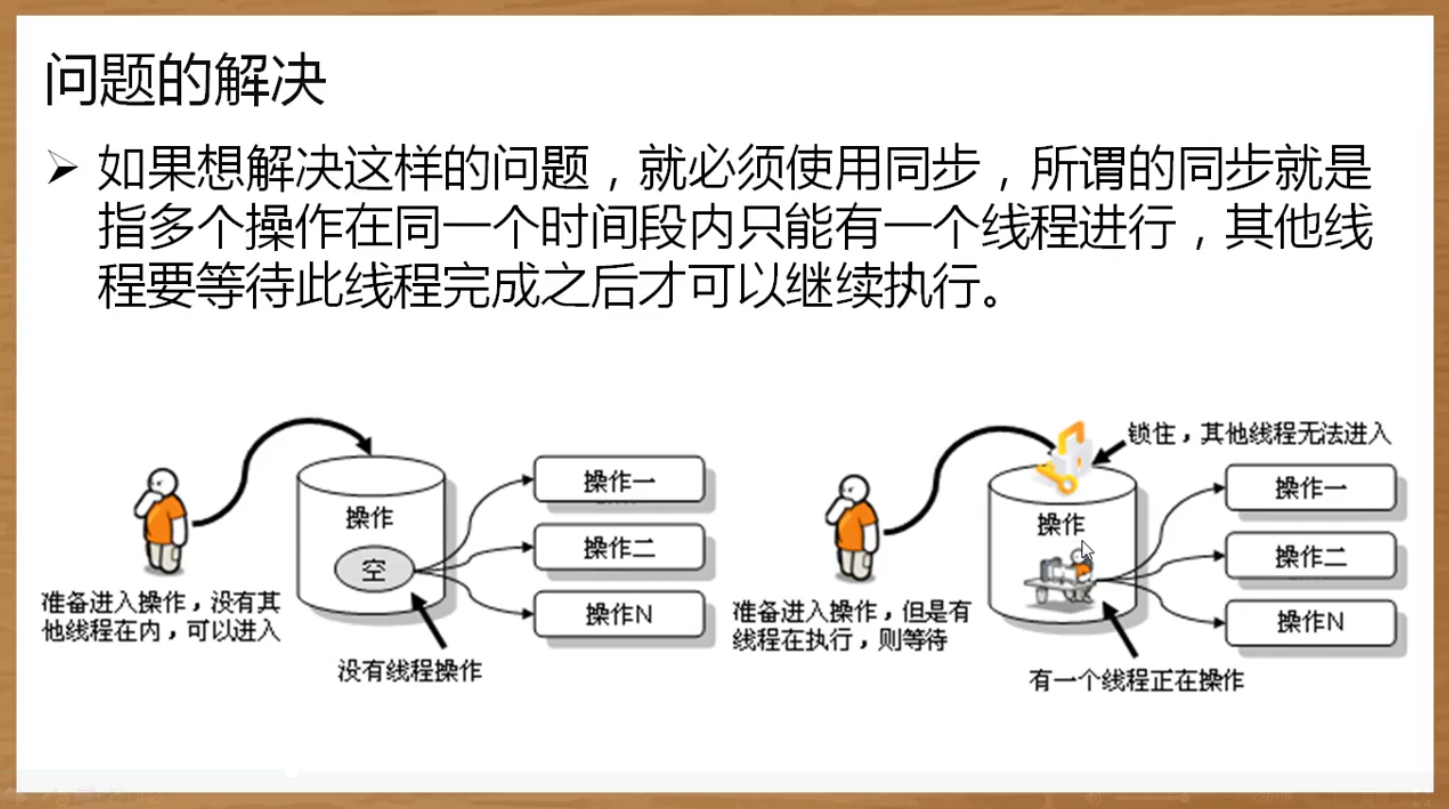
此时可以发现多个线程访问同一资源时出现了同步问题；（且可以看到出现了负数）

现在我们来进行分析，当票数只剩一张的时候，如果出现网络延迟，导致票数还没有来得及进行递减，线程B，线程C就执行了run方法，那么便会出现上述的票数为负数的情况；这就叫线程的不同步；



**·线程的同步**

经过上述的分析之后可以确认同步问题所产生的主要原因了，那么下面就要进行同步问题的解决了，而同步问题的关键时锁，指的时当某一个线程执行操作的时候，其它线程外面等待；



而如果想要在程序中实现锁的功能，则可以使用synchronized关键字定义同步方法或同步代码块；在同步代码块里面只允许一个线程执行。

1. 利用同步代码块进行处理

|  |
| --- |
| Synchronized（同步对象）{  同步代码块操作  } |

一般要进行同步对象处理时可以采用当前对象this进行同步；

范例：利用同步代码块解决数据同步访问问题；

|  |
| --- |
| package com.company;  /\*\*  \* ticket表示票数为10  \*/  class MyThread implements Runnable{  private int ticket = 10 ;  @Override  public void run() {  while (true){  synchronized(this){  if (this.ticket > 0){  try {  //模拟网络延迟  Thread.sleep(100);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println(Thread.currentThread().getName() + "卖票，ticket=" + this.ticket);  this.ticket-- ;  }else {  System.out.println("\*\*\*\*\*票卖光了\*\*\*\*\*");  break;  }  }  }  }  }  public class Ticket {  public static void main(String[] args) {  MyThread mt = new MyThread();  new Thread(mt , "票贩子A").start();  new Thread(mt , "票贩子B").start();  new Thread(mt , "票贩子C").start();  }  } |

这样会导致一个线程的执行时间太长了；

所以说加入同步处理以后，程序的整体的性能下降了。同步实际上会造成性能的降低。

1. 利用同步方法

|  |
| --- |
| package com.company;  /\*\*  \* ticket表示票数为10  \*/  class MyThread implements Runnable{  private int ticket = 10 ;  public synchronized void sale(){  while (true){  if (this.ticket > 0){  try {  //模拟网络延迟  Thread.sleep(100);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println(Thread.currentThread().getName() + "卖票，ticket=" + this.ticket);  this.ticket-- ;  }else {  System.out.println("\*\*\*\*\*票卖光了\*\*\*\*\*");  break;  }  }  }  @Override  public void run() {  this.sale();  }  }  public class Ticket {  public static void main(String[] args) {  MyThread mt = new MyThread();  new Thread(mt , "票贩子A").start();  new Thread(mt , "票贩子B").start();  new Thread(mt , "票贩子C").start();  }  } |

1. **死锁**

死锁就是若干个线程互相等待的状态；

死锁实际上时开发中的一种不确定状态，有的时候如果代码处理不当则会出现不定期死锁，这是正常调试问题；

若干个线程访问同一资源时一定要进行同步操作，而过多的同步会造成死锁。