博客： https://www.cnblogs.com/HOsystem/p/14116443.html

2、具体内容

在多线程的处理之中，可以利用Runnable描述多个线程操作的资源，而Thread描述每一个线程对象，于是当多个线程访问同一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

下面编写一个简单的卖票程序，将创建若干个线程对象实现卖票处理操作。

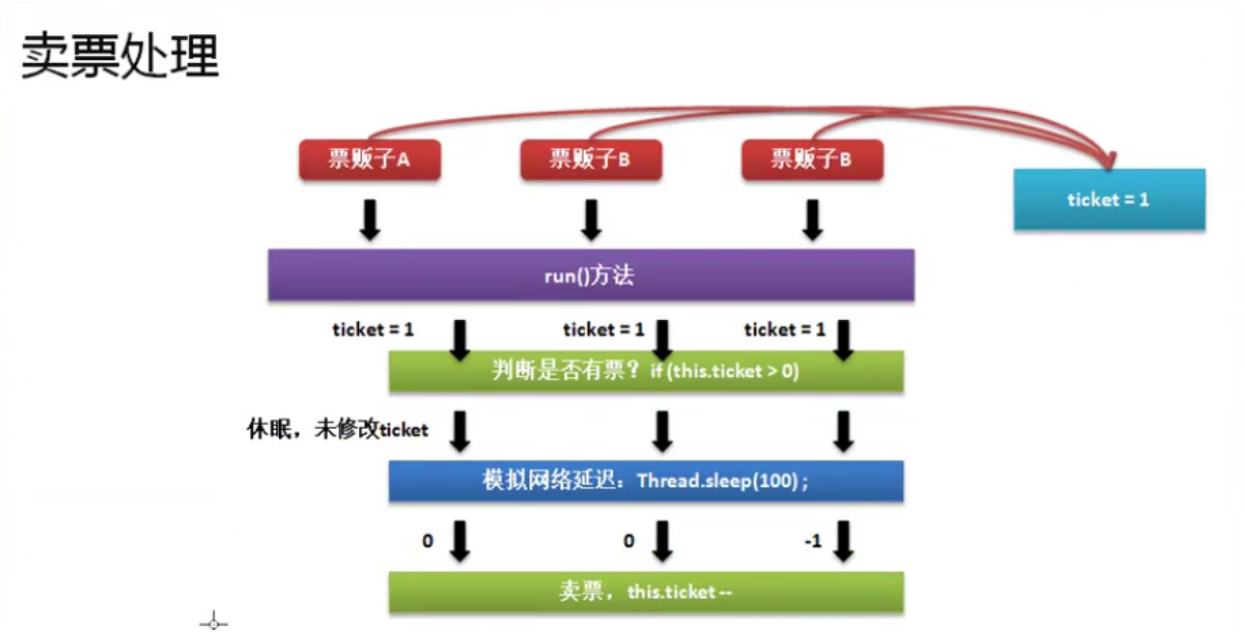
范例：实现卖票操作

```
class MyThread implements Runnable {
	private int ticket = 10 ; // 总票数为10张
	@Override
	public void run() {
		while (true) {
			if (this.ticket > 0) {
				System.out.println(Thread.currentThread().getName() + "卖票，ticket = "
+ this.ticket --);
			} else {
				System.out.println("***** 票已经卖光了 *****");
				break ;
			}
		}
	}
}
public class ThreadDemo {
	public static void main(String[] args) throws Exception {
		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();
	}
}
```

此时的程序将创建三个线程对象，并且这三个线程对象将进行10张票的出售。此时的程序在进行卖票处理的时候并没有任何的问题（假象），下面可以模拟一下卖票中的延迟操作。

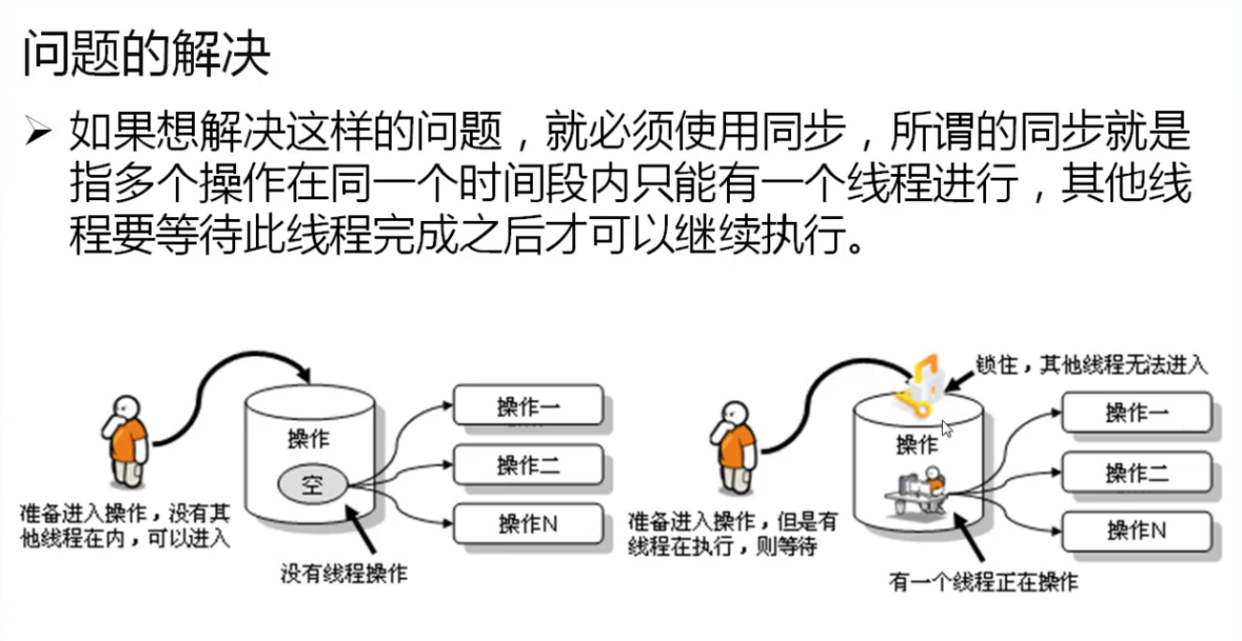
```
class MyThread implements Runnable {
	private int ticket = 10 ; // 总票数为10张
	@Override
	public void run() {
		while (true) {
			if (this.ticket > 0) {
				try {
					Thread.sleep(100);  // 模拟网络延迟
				} catch (InterruptedException e) {
					e.printStackTrace();
				}
				System.out.println(Thread.currentThread().getName() + "卖票，ticket = "
+ this.ticket --);
			} else {
				System.out.println("***** 票已经卖光了 *****");
				break ;
			}
		}
	}
}
public class ThreadDemo {
	public static void main(String[] args) throws Exception {
		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();
	}
}
```

这个时候追加了延迟问题就暴露出来了，而实际上这个问题一直都在。



■线程同步

经过分析之后已经可以确认同步问题所产生的主要原因了，那么下面就需要进行同步问题的解决，但是解决同步问题的关键是锁，指的是当某一个线程执行操作的时候，其它线程外面等待：



如果要想在程序之中实现这把锁的功能，就可以使用synchronized关键字来实现，利用此关键字可以定义同步方法或同步代码块，在同步代码块的操作里面的代码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){
    同步代码操作
}
```

一般要进行同步对象处理的时候可以采用当前对象this进行同步。

范例：利用同步代码块解决数据同步访问问题

```
class MyThread implements Runnable {
    private int ticket = 10 ; // 总票数为10张
    @Override
    public void run() {
        while (true) {
            synchronized(this) {        // 每一次只允许一个线程进行访问
                if (this.ticket > 0) {
                    try {
                        Thread.sleep(100);  // 模拟网络延迟
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
```

```
                        System.out.println(Thread.currentThread().getName() + "卖票，
ticket = " + this.ticket --);
                    } else {
                        System.out.println("***** 票已经卖光了 *****");
                        break ;
                    }
                }
            }
        }
}
public class ThreadDemo {
        public static void main(String[] args) throws Exception {
            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();
        }
}
```

加入同步处理之后，程序的整体的性能下降了。同步实际上会造成性能的降低。

2、利用同步方法解决：只需要在方法定义上使用synchronized关键字即可。

```
class MyThread implements Runnable {
        private int ticket = 10 ; // 总票数为10张
        public synchronized boolean sale() {
            if (this.ticket > 0) {
                try {
                    Thread.sleep(100);  // 模拟网络延迟
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(Thread.currentThread().getName() + "卖票，ticket = " +
this.ticket --);
                return true ;
            } else {
                System.out.println("***** 票已经卖光了 *****");
                return false ;
            }
        }
        @Override
        public void run() {
            while (this.sale()) {
                ;
            }
        }
}
public class ThreadDemo {
        public static void main(String[] args) throws Exception {
            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();
```

```
    }
}
```

在日后学习Java类库的时候会发现，系统中许多的类上使用的同步处理采用的都是同步方法。

注：同步会造成性能下降

■死锁

死锁是在进行多线程同步的处理之中有可能产生的一种问题，所谓的死锁指的是若干个线程彼此互相等待的状态。下面通过一个简单的代码来观察一下死锁的表现形式，但是对于此代码不作为重点。

范例：死锁的展示

```
public class DeadLock implements Runnable {
    private Jian jj = new Jian() ;
    private XiaoQiang xq = new XiaoQiang() ;
    @Override
    public void run() {
        jj.say(xq);
    }
    public DeadLock() {
        new Thread(this).start();
        xq.say(jj);
    }
    public static void main(String[] args) {
        new DeadLock() ;
    }
}
class Jian {
    public synchronized void say(XiaoQiang xq) {
        System.out.println("阿健说：此路是我开，要想从此过，留下10块钱。");
        xq.get();
    }
    public synchronized void get() {
        System.out.println("阿健说：得到了10块钱，可以买饭吃了，于是让出了路。");
    }
}
class XiaoQiang {
    public synchronized void say(Jian jj) {
        System.out.println("小强说：让我先跑，我再给你钱。");
        jj.get();
    }
    public synchronized void get() {
        System.out.println("小强说：逃过了一劫，可以继续送快餐了。");
    }
}
```

现在死锁造成的主要原因是因为彼此都在互相等待着，等待着对方先让出资源。死锁实际上是一种开发中出现的不确定的状态，有的时候代码如果处理不当则会不定期出现死锁，这是属于正常开发中的调试问题。

若干个线程访问同一资源时一定要进行同步处理，而过多的同步会造成死锁。