1、课程名称：多线程

2、知识点

2.1、本节预计讲解的知识点

1、 了解进程和线程的区别
2、 理解 Java 线程的两种实现方法及区别
   - 可以清楚的解释出两种实现的方式、区别、联系即可
3、 了解线程的基本控制方法，尤其是线程的休眠
4、 线程的同步与死锁的区别及使用。
   - 可以清楚的讲解出为什么需要同步，以及同步的实现，和死锁问题的产生
5、 了解线程的生命周期

3、具体内容

3.1、进程与线程的区别

从操作系统的发展来看，操作系统中处理分为单进程和多进程，例如，传统的 DOS 系统有一个特点，一旦发生了病毒，则系统立刻死机，因为传统的 DOS 系统是单进程的处理，即，在同一个时间段上只能有一个程序运行，一个程序要独享 CPU、各种输入、输出设备。

但是到 windows 中即使出现了病毒，则也不会影响程序的运行。那么证明在同一个时间段上将会有多个程序同时执行，那么这种情况称为多进程处理。但是在同一个时间段上有多个程序，但是每一个时间点上只能有一个程序运行。

线程实际上是在进程基础之上的进一步划分，一个进程启动之后，里面的若干程序又可以划分成若干个线程。

一个进程如果消失掉了，则线程也将同时消失，而如果一个线程消失了，则进程不会消失。

Java 本身是少数支持多线程的操作语言，下面来观察如何实现线程的操作。

进程：一个完成独立运行的程序称为一个进程（有独立的内存空间）

线程：是进程中的一个执行路径，共享一个内存空间，线程之间可以自由切换，并发执行

一个进程最少有一个线程（单线程程序）

3.2、Java 的线程实现（理解）

在 Java 中如果要想实现多线程的操作，有两种实现方法： 一种是继承 Thread 类

另外一种是实现 Runnable 接口

3.2.1、继承 Thread 类

一个类只要继承了 Thread 类，那么此类就可以是一个多线程的操作类，Thread 类是在 java.lang 包中定义的类，所以此类可以自动导入并使用。单单只是继承 Thread 类还不能完成线程的功能，还必须覆写 Thread 类中的 run()方法。

那么，此时实现线程的格式如下：

```
class 线程类 extends Thread{          // 继承 Thread 类
    public void run(){            // 覆写 run()方法
        // 此方法编写的是线程的主体
    }
}
```

范例：按照以上的格式实现多线程

```
class MyThread extends Thread { // 继承了Thread类
    private String name; // 定义属性
    public MyThread(String name) {
        this.name = name;
    }
    public void run() { // 覆写Thread类中的run()方法
        for (int i = 0; i < 10; i++) {
            System.out.println(this.name + " --> 运行, i = " + i);
        }
    }
}
```

以上的操作，已经实现了一个线程类，那么下面在主方法之中，就要求启动此线程。多个进程是同时运行的，那么多个线程也应该是同时运行的。那么此时就需要启动线程。

```
public class ThreadDemo01 {
    public static void main(String[] args) {
        new MyThread("线程A").run() ;
        new MyThread("线程B").run() ;
    }
}
```

主方法中声明了两个线程对象，那么此时就应该是两个程序同时运行。但是，此时发现，程序并不是同时运行的，还是属于顺序执行，即：先执行完第一个对象之后，再执行第二个对象。

所以，如果要想在程序中正确的启动多线程，则必须使用 Thread 类中的 start()方法，此方法如下：

- 启动线程：public void start()，从此方法的解释上可以发现，此方法执行的时候，将由 JVM 去调用 run()
- 相当于：调用 start() → 执行 run()方法。

将以上的程序进行修改：

```
public class ThreadDemo02 {
    public static void main(String[] args) {
        new MyThread("线程A").start() ;
        new MyThread("线程B").start() ;
    }
}
```

此时，发现代码，可以完成交替的运行操作，谁抢占到了 CPU 资源，就运行那个线程。

提问？

为什么要想启动多线程必须使用 start()方法？

如果要想连接此块内容，则必须从 java 的源代码中进行分析。源代码如下：

```
public synchronized void start() {
    if (threadStatus != 0)
        throw new IllegalThreadStateException();    → 会有一个异常抛出
    group.add(this);
    start0();  → 此处调用了start0()
    if (stopBeforeStart) {
        stop0(throwableFromStop);
    }
}
private native void start0();    → 此方法为私有的，而且使用了 native 关键字声明
```

一个线程对象只能调用一次 start()方法启动，如果重复调用了，则将抛出以上的异常“IllegalThreadStateException”

native 关键字：

- 实际上从 Java 的发展来讲最早有一门技术，此技术称为 JNI（Java Native Interface），本地自然接口，此操作的主要功能是调用本机操作系统的库函数。

从操作系统来讲，线程本身需要 CPU 支持，而且要等待 CPU 进行调度，那么此操作肯定是由操作系统的低层完成的。所以一旦调用了 start()方法，就意味着调用着操作系统的低层支持，以运行多线程的程序。

但是，以上的操作代码是属于继承了一个类的形式完成的，那么就会有单继承的局限，所以在线程的开发中还会使用以下一种情况，通过 Runnable 接口来实现多线程。

3.2.2、实现 Runnable 接口

观察一下 Runnable 接口的定义，因为此接口也是可以实现多线程的。此接口也是在 java.lang 包中定义的。

```
public interface Runnable{
    public void run() ;
}
```

那么，此时子类只需要实现以上的接口就可以完成多线程的支持了。

范例：定义 Runnable 接口的子类

```
class MyThread implements Runnable { // 实现了Runnable接口
    private String name; // 定义属性
    public MyThread(String name) {
        this.name = name;
    }
    public void run() { // 覆写Thread类中的run()方法
        for (int i = 0; i < 10; i++) {
            System.out.println(this.name + " --> 运行, i = " + i);
        }
    }
}
```

实现之后，那么该如何启动线程呢？

之前继承 Thread 类实现多线程的时候靠的是 Thread 类中的 start()方法，但是现在实现的是 Runnable 接口，此接口中并没有 start()方法的定义，那么该如何启动呢？因为线程启动的时候必须依靠操作系统的支持，所以肯定要使用 start()方法，此时观察 Thread 类中的构造方法：

- 接收 Runnable 子类实例的构造：public Thread(Runnable target)

可以通过 Runnable 子类的对象构建 Thread 类的对象，并调用 start()方法。

```
public class RunnableDemo01 {
    public static void main(String[] args) {
        MyThread mt1 = new MyThread("线程A") ;
        MyThread mt2 = new MyThread("线程B") ;
        new Thread(mt1).start() ;   // 通过Thread类启动多线程
        new Thread(mt2).start() ;   // 通过Thread类启动多线程
    }
}
```

3.2.3、两种实现方式的区别

既然多线程的操作有两种实现方式，那么两者的区别是什么，该使用那个呢？

- 首先，从基本概念上看，使用 Runnable 接口更好，因为避免单继承局限
- 其次，使用 Runnable 接口还可以方便的实现数据的共享操作。

以一个卖票程序为例，观察这两者的实现区别。

范例：使用 Thread 类观察运行结果

```
package org.thread.demo03;
class MyThread extends Thread { // 继承了Thread类
    private int ticket = 3; // 一共是3张票
    public void run() { // 覆写Thread类中的run()方法
        for (int i = 0; i < 10; i++) {
            if (this.ticket > 0) {
                System.out.println("卖票 --> 剩余 = " + this.ticket--);
            }
        }
    }
}
public class ThreadDemo03 {
    public static void main(String[] args) {
        new MyThread().start();
        new MyThread().start();
        new MyThread().start();
    }
}
```

以上的程序一共启动了三个线程，结果卖出了 9 张票，个人卖个人的票。现在并没有实现资源的共享。

范例：现在使用 Runnable 来实现

```
package org.runnable.demo02;
class MyThread implements Runnable { // 实现了Runnable接口
    private int ticket = 3; // 一共是3张票
    public void run() { // 覆写Thread类中的run()方法
        for (int i = 0; i < 10; i++) {
            if (this.ticket > 0) {
                System.out.println("卖票 --> 剩余 = " + this.ticket--);
            }
```

```
        }
    }
}
public class RunnableDemo02 {
    public static void main(String[] args) {
        MyThread my = new MyThread() ;
        new Thread(my).start(); // 启动一个线程
        new Thread(my).start(); // 启动两个线程
        new Thread(my).start(); // 启动三个线程
    }
}
```

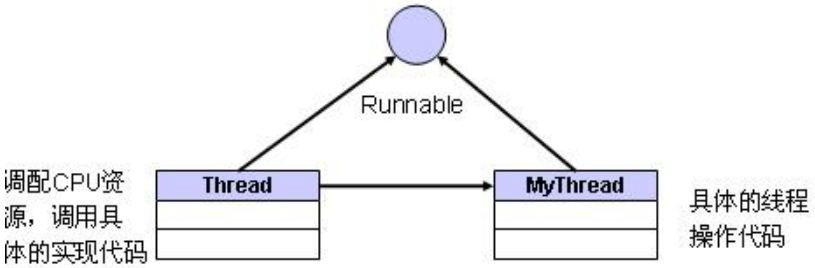
通过以上的操作代码，可以发现使用 Runnable 接口实际上可以完成资源共享的操作。

除了以上的明显标志之外，对于两种实现本身还有联系的，观察 Thread 类的定义格式：

```
public class Thread
extends Object
implements Runnable
```

发现，实际上 Thread 类本身也属于 Runnable 接口的子类。

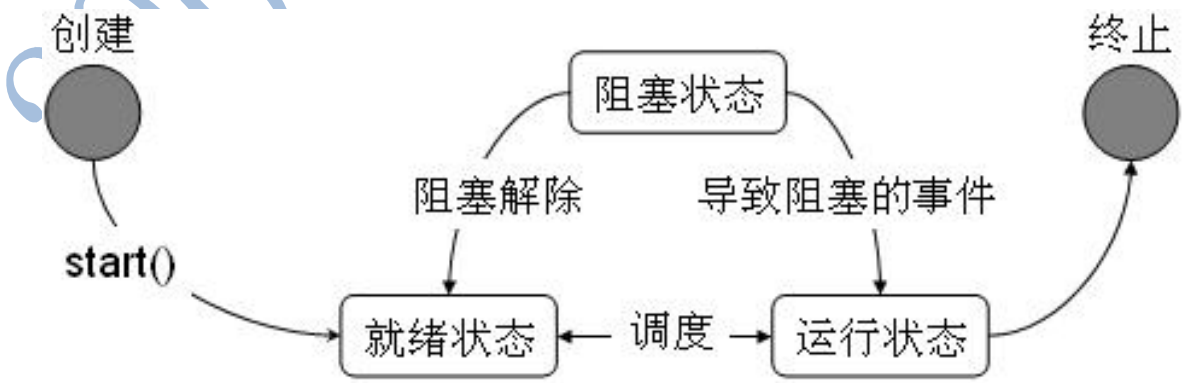
那么，之前使用 Runnable 接口实现多线程操作的时候实际上操作的效果就如下图所示。



所以，从以上的图形中可以发现，Thread 类实际上完成的只是一个代理的操作功能，而具体的功能由 MyThread（用户自定义的类）来完成，一个典型的代理设计模式应用。

3.3、线程的操作状态

之前的代码中可以发现线程有如下的操作状态：创建 → 启动 → 运行。



第一步创建一个线程的实例化对象。

第二步启动多线程，调用 start()方法，但是调用 start()方法之后并不是立刻执行多线程操作。而是跑到就绪状态。

第三步等待 CPU 进行调度，调度之后进入到运行状态。

第四步如果现在假设运行一段时间之后，被其他线程抢先了，则出现了阻塞状态，

第五步，从阻塞状态要重新回到就绪状态，等待 CPU 的下一次调度

第六步，当全部的操作执行完成之后，线程将终止执行。

虽然在代码上操作是有先有后，但是所有的线程肯定都是同时启动的，所以在线程的开发中没有顺序而言。

那个线程抢占到了 CPU 资源，那个线程就先执行。

3.4、线程的操作方法

在 Java 中提供了很多的线程操作方法，所有的方法都是在 Thread 类中定义的。也就是说，所有的操作可以直接在 java.lang.Thread 类中找到。

3.4.1、取得当前的线程、设置和取得名字

在线程中可以通过 currentThread()方法取得当前正在运行的线程对象，并且可以为每一个线程对象设置和取得名字，所有的方法如下所示：

No.	方法名称	类型	描述
1	public static Thread currentThread()	普通	返回当前正在执行的线程对象
2	public final String getName()	普通	取得线程的名字
3	public final void setName(String name)	普通	设置线程的名字
4	public Thread(Runnable target,String name)	构造	接收 Runnable 对象，并指定线程名称
5	public Thread(String name)	构造	为线程指定一个名字

但是要提醒的是，在线程的操作中最好在线程启动前设置线程的名字，而且中途不要修改名字。而且还要注意的是，不要设置同名的线程对象。

范例：取得和设置线程名称

```
package org.threadapi.namedemo01;
class MyThread implements Runnable {
    public void run() {
        for (int x = 0; x < 5; x++) { // 取得当前运行的线程名称
            System.out.println(Thread.currentThread().getName() + " --> 运行");
        }
    }
}
public class NameThreadDemo01 {
    public static void main(String[] args) {
        MyThread my = new MyThread();
        new Thread(my, "线程A").start(); // 设置了线程名称
        new Thread(my).start(); // 没设置线程名称
        new Thread(my).start(); // 没设置线程名称
        new Thread(my, "线程B").start(); // 设置了线程名称
        new Thread(my, "线程C").start(); // 设置了线程名称
        new Thread(my).start(); // 没设置线程名称
        new Thread(my).start(); // 没设置线程名称
        new Thread(my).start(); // 没设置线程名称
    }
}
```

发现，没有设置线程名字的线程对象所有的命名都采用“Thread-xxx”的形式出现。通过运行效果，可以得出，在 Thread 类中肯定存在一个 static 的属性，用于记录每次的增长结果。

了解了以上的操作之后，下面继续观察如下的代码：

```
package org.threadapi.namedemo02;

class MyThread implements Runnable {
    public void run() {
        for (int x = 0; x < 5; x++) { // 取得当前运行的线程名称
            System.out.println(Thread.currentThread().getName() + " --> 运行");
        }
    }
}

public class NameThreadDemo02 {
    public static void main(String[] args) {
        MyThread my = new MyThread();
        new Thread(my, "线程").start(); // 设置了线程名称
        my.run(); // 直接调用run()方法    → 直接调用的时候出现了main线程
    }
}
```

通过以上的返回结果，可以发现，主方法本身也是一个线程。

一直强调 Java 是多线程的操作语言，所以实际上每次 Java 命令执行的时候对于 JVM 都相当于启动了一个 Java 的进程，而主方法是在此进程上的进一步划分。

思考？

Java 每次运行的时候至少启动几个线程？

- 两个：主线程、GC 线程

3.4.2、判断线程的运行状态

一个线程对象是否正在执行，可以通过 isAlive()方法进行验证。此方法如下：

No.	方法名称	类型	描述
1	public final boolean isAlive()	普通	判断线程是否存活

以上的方法命名是：“isXxx()”，以后凡是看见以 is 开头的方法，此方法的返回值结果都是 boolean 型数据。

范例：判断线程的运行状态

```
package org.threadapi.alivedemo;
class MyThread implements Runnable {
    public void run() {
        for (int x = 0; x < 5; x++) { // 取得当前运行的线程名称
            System.out.println(Thread.currentThread().getName() + " --> 运行");
        }
    }
}
public class AliveDemo {
    public static void main(String[] args) {
        MyThread my = new MyThread();
```

```
        Thread t = new Thread(my); // 定义线线程对象
        System.out.println("1、线程启动之前：" + t.isAlive());
        t.start(); // 启动线程
        System.out.println("2、线程启动之后：" + t.isAlive());
        for (int x = 0; x < 10; x++) { // 由主线程调用
            System.out.println("主方法中的循环，x = " + x) ;
        }
        System.out.println("3、线程执行完后：" + t.isAlive());
    }
}
```

此时，如果主线程先执行完毕，则最后打印“true”，而如果主线程后执行完，则打印“false”，得出，线程每次的运行状态是不一致的。

3.4.3、线程的强制执行

之前的所有操作中，所有的线程都是同时执行的，而且在同一个时间段上会有多个线程同时执行，那么如果现在希望一个线程可以强制的执行下去的话，则可以使用 join()方法。

No.	方法名称	类型	描述
1	public final void join() throws InterruptedException	普通	强制运行
2	public final void join(long millis) throws InterruptedException	普通	规定强制运行的毫秒
3	public final void join(long millis,int nanos) throws InterruptedException	普通	规定强制运行的毫秒和纳秒

范例：验证 join 操作

```
package org.threadapi.joindemo;
class MyThread implements Runnable {
    public void run() {
        for (int x = 0; x < 10000; x++) { // 取得当前运行的线程名称
            System.out.println(Thread.currentThread().getName() + " --> 运行，x = " + x);
        }
    }
}
public class JoinDemo {
    public static void main(String[] args) {
        Thread t = new Thread(new MyThread()); // 声明线程对象
        t.start();
        int x = 0;
        while (true) {// 死循环
            System.out.println("主方法中的输出，x = " + x++);
            if (x >= 100) { // 由线程强制执行
                try {
                    t.join();// 本线程将强制运行下去
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
```

```
            }
        }
    }
}
```

此时，线程将强制执行完毕之后，再回到主方法继续执行。

3.4.4、线程的休眠

让一个线程稍微暂停一下执行，那么这样的操作称为休眠，休眠的话使用 Thread 类中的 sleep()方法完成，此方法定义如下：

No.	方法名称	类型	描述
1	public static void sleep(long millis) throws InterruptedException	普通	休眠指定的毫后继续执行
2	public static void sleep(long millis,int nanos) throws InterruptedException	普通	修面指定豪秒和纳秒之后继续执行

范例：观察线程的休眠操作

```
package org.threadapi.sleepdemo;
class MyThread implements Runnable {
    public void run() {
        for (int x = 0; x < 30; x++) { // 取得当前运行的线程名称
            try {
                Thread.sleep(1000); // 每次操作延迟1秒中
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName()
                    + " --> 运行, x = " + x);
        }
    }
}
public class SleepDemo {
    public static void main(String[] args) {
        Thread t = new Thread(new MyThread()); // 声明线程对象
        t.start();
    }
}
```

此时，代码的执行速度有所下降，证明，每次在操作的时候都进行休眠的操作。

sleep()可使优先级低的线程得到执行的机会，当然也可以让同优先级和高优先级的线程有执行的机会

3.4.5、线程的中断

在线程的操作中可以使用一个线程中断另外一个线程的运行状态，使用 interrupt 方法，此方法如下：

No.	方法名称	类型	描述
1	public void interrupt()	普通	中断线程运行

2	public boolean isInterrupted()	普通	判断线程是否被中断

范例：观察线程的中断

```
package org.threadapi.interruptdemo;
class MyThread implements Runnable {
    public void run() {
        System.out.println("1、进入run()方法，等待休眠");
        try {
            Thread.sleep(20000);
            System.out.println("2、休眠结束");
        } catch (InterruptedException e) {
            System.out.println("3、休眠被中断");
            return;// 返回被调用处
        }
        System.out.println("4、方法正常结束，休眠正常完成");
    }
}
public class InterruptDemo {
    public static void main(String[] args) {
        Thread t = new Thread(new MyThread()); // 声明线程对象
        t.start();
        try {
            Thread.sleep(2000);// 让程序先运行2秒
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        t.interrupt(); // 中断线程
    }
}
```

3.4.6、线程的优先级

一个线程要想更多的抢占 CPU 资源，则肯定要有很高的优先级，在线程中优先级有三种级别，而且通过 setPriority() 和 getPriority()。

三种优先级实际上在 Thread 类中定义了三个常量，此三个常量的说明如下

No.	方法及常量名称	类型	描述
1	public static final int MAX_PRIORITY	常量	最高优先级
2	public static final int NORM_PRIORITY	常量	普通优先级
3	public static final int MIN_PRIORITY	常量	最低优先级
4	public final int getPriority()	普通	得到线程的优先级
5	public final void setPriority(int newPriority)	普通	设置线程的优先级

范例：演示三种优先级

```
package org.threadapi.prioritydemo;
class MyThread implements Runnable {
    public void run() {
        for (int x = 0; x < 3; x++) { // 取得当前运行的线程名称
```

```
            System.out.println(Thread.currentThread().getName()
                    + " --> 运行, x = " + x);
        }
    }
}
public class PriorityDemo {
    public static void main(String[] args) {
        Thread t1 = new Thread(new MyThread()); // 声明线程对象
        Thread t2 = new Thread(new MyThread()); // 声明线程对象
        Thread t3 = new Thread(new MyThread()); // 声明线程对象
        t1.setPriority(Thread.MIN_PRIORITY); // 最小优先级
        t2.setPriority(Thread.MAX_PRIORITY); // 最大优先级
        t3.setPriority(Thread.NORM_PRIORITY); // 普通优先级
        t1.start();
        t2.start();
        t3.start();
    }
}
```

以上的程序中并不是意味着优先级高就一定要先执行，而是说优先级高得到 CPU 先执行的机会就越大。

提问？

主方法的优先级是多少？

```
package org.threadapi.prioritydemo;
public class MainPriority {
    public static void main(String[] args) {
        System.out.println(Thread.currentThread().getPriority());
        System.out.println("MAX_PRIORITY: " + Thread.MAX_PRIORITY);
        System.out.println("NORM_PRIORITY: " + Thread.NORM_PRIORITY);
        System.out.println("MIN_PRIORITY: " + Thread.MIN_PRIORITY);
    }
}
```

主方法的优先级是普通的优先级。

3.5、线程的同步与死锁（理解）

3.5.1、线程同步的引出

在多线程的操作中，多个线程有可能同时处理同一个资源，例如：实现了 Runnable 接口之后就属于资源共享的操作。

范例：观察一下同步的问题

```
package org.synchronizeddemodemo01;
class MyThread implements Runnable { // 实现了Runnable接口
    private int ticket = 10; // 一共是10张票
    public void run() { // 覆写Thread类中的run()方法
        for (int i = 0; i < 100; i++) {
            if (this.ticket > 0) {
```

```
                try {
                    Thread.sleep(200); // 加入延迟操作
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("卖票 --> 剩余 =  " + this.ticket--);
            }
        }
    }
}
public class SynchronizedDemo01 {
    public static void main(String[] args) {
        MyThread my = new MyThread();
        Thread t1 = new Thread(my);
        Thread t2 = new Thread(my);
        Thread t3 = new Thread(my);
        t1.start();
        t2.start();
        t3.start();
    }
}
```

此时，发现程序运行的时候出现了负数，为什么会出现？

现在来分析一下程序的执行步骤：

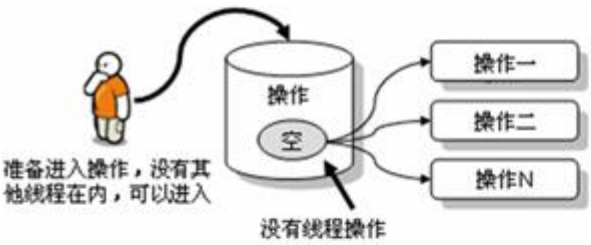
1、 判断是否还有票

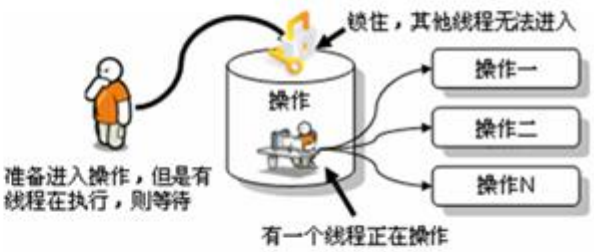
2、 修改票数

此时，所有的线程会同时进入到操作的方法之中。

问题的解决

- 如果想解决这样的问题，就必须使用同步，所谓的同步就是指多个操作在同一个时间段内只能有一个线程进行，其他线程要等待此线程完成之后才可以继续执行。





应该完成一个加锁的操作。那么这样的操作在程序中就称为同步操作，使用 synchronized 关键字完成功能。

3.5.2、线程同步

在 Java 中要想对线程进行同步，有以下两种方法：第一种使用同步代码块，第二种使用同步方法。

但是如果要想使用同步代码块的话，则必须指定要对那个对象进行同步，所以同步代码块的执行格式如下：

```
synchronized(要同步的对象){
    要同步的操作 ；
}
```

范例：为以上的操作增加同步

```
package org.synchronizeddemodemo02;
class MyThread implements Runnable { // 实现了Runnable接口
    private int ticket = 10; // 一共是10张票
    public void run() { // 覆写Thread类中的run()方法
        for (int i = 0; i < 100; i++) {
            synchronized (this) {// 对当前操作的对象进行同步
                if (this.ticket > 0) {
                    try {
                        Thread.sleep(200); // 加入延迟操作
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    System.out.println("卖票 --> 剩余 = " + this.ticket--);
                }
            }
        }
    }
}
public class SynchronizedDemo02 {
    public static void main(String[] args) {
        MyThread my = new MyThread();
        Thread t1 = new Thread(my);
        Thread t2 = new Thread(my);
        Thread t3 = new Thread(my);
        t1.start();
        t2.start();
        t3.start();
    }
}
```

加入同步代码块之后，可以发现程序的执行速度有所减缓，但是最终的结果很正确的。

当然，也可以使用同步方法完成操作

```
package org.synchronizeddemodemo03;
class MyThread implements Runnable { // 实现了Runnable接口
    private int ticket = 10; // 一共是10张票
    public void run() { // 覆写Thread类中的run()方法
        for (int i = 0; i < 100; i++) {
            this.sale();
```

```
        }
    }
    public synchronized void sale() { // 售票，是同步方法
        if (this.ticket > 0) {
            try {
                Thread.sleep(200); // 加入延迟操作
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("卖票 --> 剩余 =  " + this.ticket--);
        }
    }
}
public class SynchronizedDemo03 {
    public static void main(String[] args) {
        MyThread my = new MyThread();
        Thread t1 = new Thread(my);
        Thread t2 = new Thread(my);
        Thread t3 = new Thread(my);
        t1.start();
        t2.start();
        t3.start();
    }
}
```

那么，此时，实际上就可以给出一个方法的完整定义格式了：

```
[public|protected|private] [synchronized]
[native] [static] [final]  返回值类型 | 方法名称(参数类型  参数名称,...) [throws  异常 1，异常 2]{
    [return  返回值] ;
}
```

同步准则

当编写 synchronized 块时，有几个简单的准则可以遵循，这些准则在避免死锁和性能危险的风险方面大有帮助：

1. 使代码块保持简短。Synchronized 块应该简短— 在保证相关数据操作的完整性的同时，尽量简短。把不随线程变化的预处理和后处理移出 synchronized 块。
2. 不要阻塞。不要在 synchronized 块或方法中调用可能引起阻塞的方法，如 InputStream.read()。
3. 在持有锁的时候，不要对其它对象调用方法。这听起来可能有些极端，但它消除了最常见的死锁源头。

3.5.3、线程死锁

过多的同步有可能出现死锁，死锁的操作一般是在程序运行的时候才有可能出现，下面通过一个简单的代码模拟一个死锁的操作。

```
package org.deadthread.demo01;
class BangJiaFan {
    public synchronized void say(GeGe g) {
```

```
        System.out.println("你给我５００W，我放了你妹妹！");
        g.fun() ;
    }
    public synchronized void fun() {
        System.out.println("绑架犯得到了５００W，放了妹妹");
    }
}
class GeGe {
    public synchronized void say(BangJiaFan b) {
        System.out.println("你放了我妹妹，我给你５００W！");
        b.fun() ;
    }
    public synchronized void fun() {
        System.out.println("哥哥救出了妹妹，结果损失了５００W");
    }
}
public class DeadThread implements Runnable {
    BangJiaFan b = new BangJiaFan();
    GeGe g = new GeGe();
    public DeadThread() {
        new Thread(this).start() ;
        b.say(g) ;
    }
    public void run() {
        g.say(b) ;
    }
    public static void main(String[] args) {
        new DeadThread() ;
    }
}
```

以上的操作代码中就发生了死锁，那么所有的操作都要进行等待，直到代码操作完成为止。

得出一个最重要的结论：“程序中如果要想进行资源的共享操作，则肯定需要同步，如果同步过多，则就有可能造成死锁”

3.6、线程的经典操作案例：生产者-消费者

在整个多线程的开发中有一个最经典的操作案例，就是生产者-消费者，生产者不断生产产品，消费者不断取走产品

3.6.1、基础的实现

假设现在生产者生产的是两种信息（信息名称 → 信息内容）：

- 第一种信息：张三 → Java 讲师
- 第二种信息：java → finally-m.iteye.com

那么，此时就要求生产者生产以上信息的时候，生产出一个，消费者就要取走一个。如果要想完成这样一个操作，

则肯定最好先定义出一个 Info 类，表示生产的信息。

```
package org.pcdemo.demo01;
public class Info {
    private String title = "张三" ;
    private String content = "Java讲师" ;
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }
    public String getContent() {
        return content;
    }
    public void setContent(String content) {
        this.content = content;
    }
}
```

之后定义生产者，生产者要占有以上类的对象引用。

```
package org.vince.pcdemo.demo01;
public class Producter implements Runnable {
    private Info info = null;
    public Producter(Info info) {
        this.info = info;
    }
    public void run() {
        for (int i = 0; i < 100; i++) { // 生产１００个
            if (i % 2 == 0) {
                info.setTitle("java");
                try {
                    Thread.sleep(300); // 加入延迟操作
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                info.setContent("finally-m.iteye.com");
            } else {
                info.setTitle("张三");
                try {
                    Thread.sleep(300); // 加入延迟操作
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                info.setContent("Java讲师");
            }
        }
    }
```

```
}
```

消费者要不断的取走信息。

```
package org.pcdemo.demo01;
public class Consumer implements Runnable {
    private Info info = null;
    public Consumer(Info info) {
        this.info = info;
    }
    public void run() {
        for (int i = 0; i < 100; i++) {
            try {
                Thread.sleep(300); // 加入延迟操作
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(this.info.getTitle() + " --> "
                    + this.info.getContent());
        }
    }
}
```

生产者将生产 100 个产品，那么消费者将取走 100 次产品。

```
package org.pcdemo.demo01;
public class TestPC {
    public static void main(String[] args) {
        Info info = new Info();
        Producter p = new Producter(info);
        Consumer c = new Consumer(info);
        new Thread(p).start(); // 启动生产者
        new Thread(c).start(); // 启动消费者
    }
}
```

以上程序中出现了两个问题：

- 第一个问题：值错误
- 第二个问题：重复取值，肯定会有重复生产

3.6.2、解决值错误的问题

之所以会出现设置的值错误，主要是因为没有加入同步的操作造成的。所以为了解决此问题，首先加入同步操作。修改 Info 类。

```
package org.pcdemo.demo02;
public class Info {
    private String title = "张三";
    private String content = "Java讲师";
    public synchronized void set(String title, String content) {
        this.setTitle(title);
```

```
        try {
            Thread.sleep(300); // 加入延迟操作
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        this.setContent(content);
    }
    public synchronized void get() {
        try {
            Thread.sleep(300); // 加入延迟操作
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(this.getTitle() + " --> " + this.getContent());
    }
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }
    public String getContent() {
        return content;
    }
    public void setContent(String content) {
        this.content = content;
    }
}
```

设置和取得的方法都是同步的操作。

生产者中没有必要再调用 setter 设置内容

```
package org.pcdemo.demo02;
public class Producter implements Runnable {
    private Info info = null;
    public Producter(Info info) {
        this.info = info;
    }
    public void run() {
        for (int i = 0; i < 100; i++) { // 生产100个
            if (i % 2 == 0) {
                info.set("java", "finally-m.iteye.com");
            } else {
                info.set("张三", "Java讲师");
            }
        }
    }
}
```

消费者，此时直接调用 get()方法取得内容

```
package org.pcdemo.demo02;
public class Consumer implements Runnable {
    private Info info = null;
    public Consumer(Info info) {
        this.info = info;
    }
    public void run() {
        for (int i = 0; i < 100; i++) {
            this.info.get() ;
        }
    }
}
```

那么，这个时候再进行测试，观察解决了那些问题。

此时，已经解决了第一个问题，起码现在设置的值没有错误的了，但是依然存在重复设置和重复读取的问题。那么该如何解决呢？

3.6.3、使用 Object 解决问题

如果要想正确的解决以上的重复设置和重复读取的问题，则需要依靠 Object 类完成，在 Object 类中提供了等待及唤醒的机制。

No.	方法名称	类型	描述
1	public final void wait() throws InterruptedException	普通	等待
2	public final void wait(long timeout) throws InterruptedException	普通	等待，并设置等待时间
3	public final void wait(long timeout,int nanos) throws InterruptedException	普通	等待，并设置等待的时间
4	public final void notify()	普通	唤醒
5	public final void notifyAll()	普通	唤醒

在以上的方法中存在两个唤醒的操作，notify()、notifyAll()，这两个操作的区别如下：

- notify()只唤醒第一个等待的线程
- notifyAll()唤醒全部等待的线程，那个优先级高，就先执行那个

当调用 wait()后，线程会释放掉它所占有的“锁标志”，从而使线程所在对象中的其它 synchronized 数据可被别的线程使用。wait()和 notify()因为会对对象的“锁标志”进行操作，所以它们必须在 synchronized 函数或 synchronized block 中进行调用。如果在 non-synchronized 函数或 non-synchronized block 中进行调用，虽然能编译通过，但在运行时会发生 IllegalMonitorStateException 的异常。

3.6.4、解决重复设置和重复取得的问题

在以上的操作中应该增加一个标志位，此标志位有两种状态，换到程序中应该使用 boolean 类型表示：

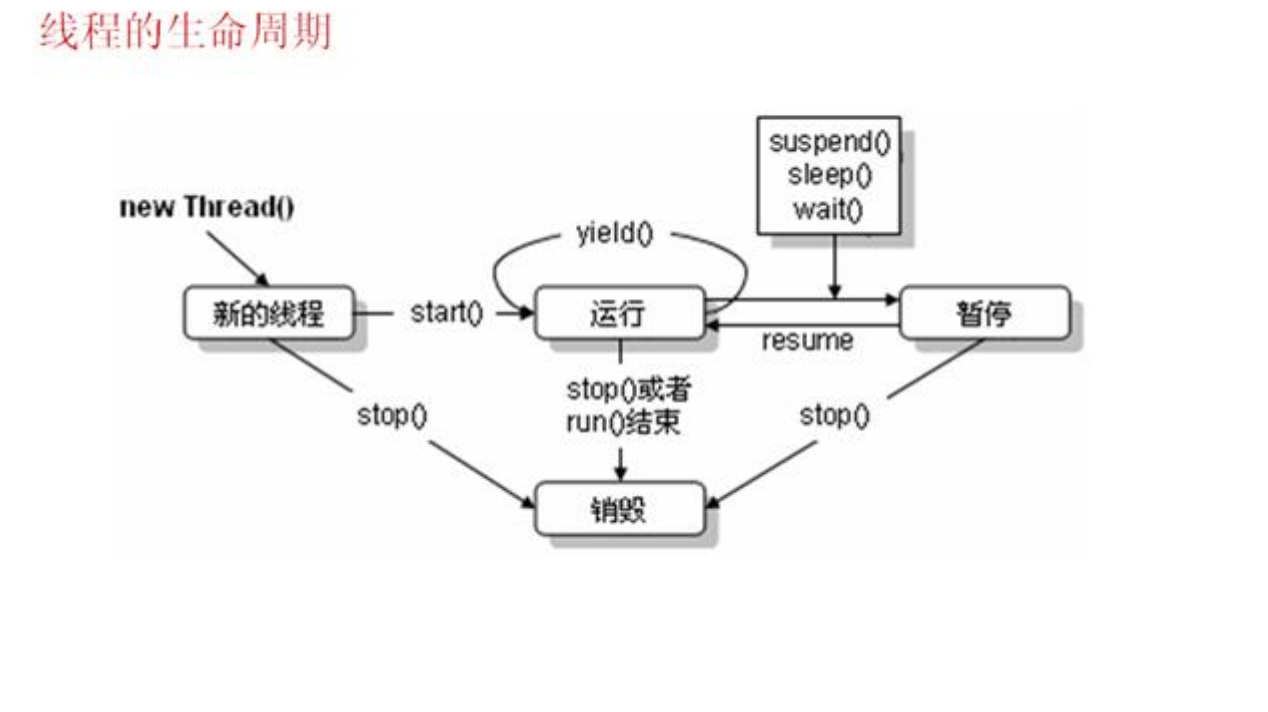
- 如果 boolean 的值是 true 的话，表示可以生产，但是不能取走
- 如果 boolean 的值是 false 的话，表示可以取走，但是不能生产

范例：修改 Info 类

```
package org.pcdemo.demo03;
public class Info {
    private String title = "张三";
    private String content = "Java讲师";
    private boolean flag = false;
    public synchronized void set(String title, String content) {
        if (flag == false) { // 不能生产，需要等待
            try {
                this.wait(); // 等待
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        this.setTitle(title);
        try {
            Thread.sleep(300); // 加入延迟操作
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        this.setContent(content);
        this.flag = false; // 表示已经生产完成，下面要取走内容
        this.notify(); // 唤醒其他线程
    }
    public synchronized void get() {
        if (this.flag == true) {
            try {
                this.wait(); // 等待
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        try {
            Thread.sleep(300); // 加入延迟操作
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(this.getTitle() + " --> " + this.getContent());
        this.flag = true; // 表示已经取走了，可以生产
        this.notify(); // 唤醒
    }
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }
```

```
    public String getContent() {
        return content;
    }
    public void setContent(String content) {
        this.content = content;
    }
}
```

3.7、线程的生命周期（了解）



以上的大部分操作都已经掌握了，但是在 Thread 类中有如下三个方法是不使用的：

1、 suspend()：表示暂时挂起线程

2、 resume()：表示恢复已挂起的线程

3、 stop()：表示停止线程运行

在 Java doc 文档中已经明确的标记出此方法都属于“Deprecated”声明，表示此操作已经不建议继续使用了。

以上的三个操作都会存在死锁的情况出现。

那么如果现在要想完成一个线程的停止操作，该如何进行呢？

最好的做法是通过标志位进行停止操作，如下代码所示：

```
package org.stopdemo;
class MyThread implements Runnable {
    private boolean flag = true;
    public void stop() {
        this.flag = false;
    }
    public void run() { // 覆写run()方法
        int x = 0;
```

```
        while (this.flag) {
            System.out.println(Thread.currentThread().getName()
                    + " --> 运行, x = " + x++);
        }
    }
}
public class StopDemo {
    public static void main(String[] args) {
        MyThread my = new MyThread();
        Thread t1 = new Thread(my);
        t1.start();
        try {
            Thread.sleep(20);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        my.stop();
    }
}
```

如果要想停止一个线程的运行，则肯定以上的操作完成。

4、总结

1、 理解线程的两种操作状态及区别和联系

2、 理解线程同步的作用，同步的关键字 synchronized 和死锁的产生

3、 了解线程的生命周期，和不建议使用的三个方法。

5、作业

1. 完成本节内容的所有示例并充分理解。