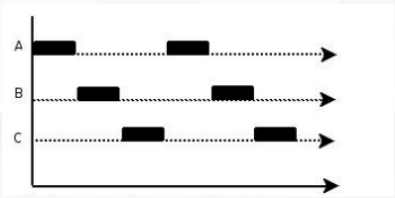
Java Thread

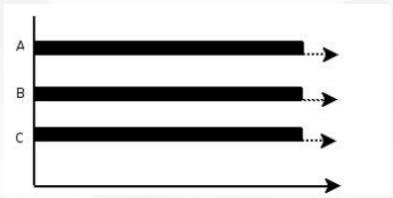
1、线程与进程

- 什么是进程？
  - 进程是指运行中的应用程序，每一个进程都有自己独立的内存空间。一个应用程序可以同时启动多个线程。进程也是程序的一次执行过程，是系统运行程序的基本单位。
- 什么是线程?
  - 线程是指进程中的一个执行流程。一个进程可以由多个线程组成。
- 进程和线程的区别？
  - 一个程序运行后至少有一个进程，一个进程中可以包含多个线程 。
  - 每个进程都需要操作系统为其分配独立的内存地址空间，而同一进程中的所有线程共享同一地址空间。

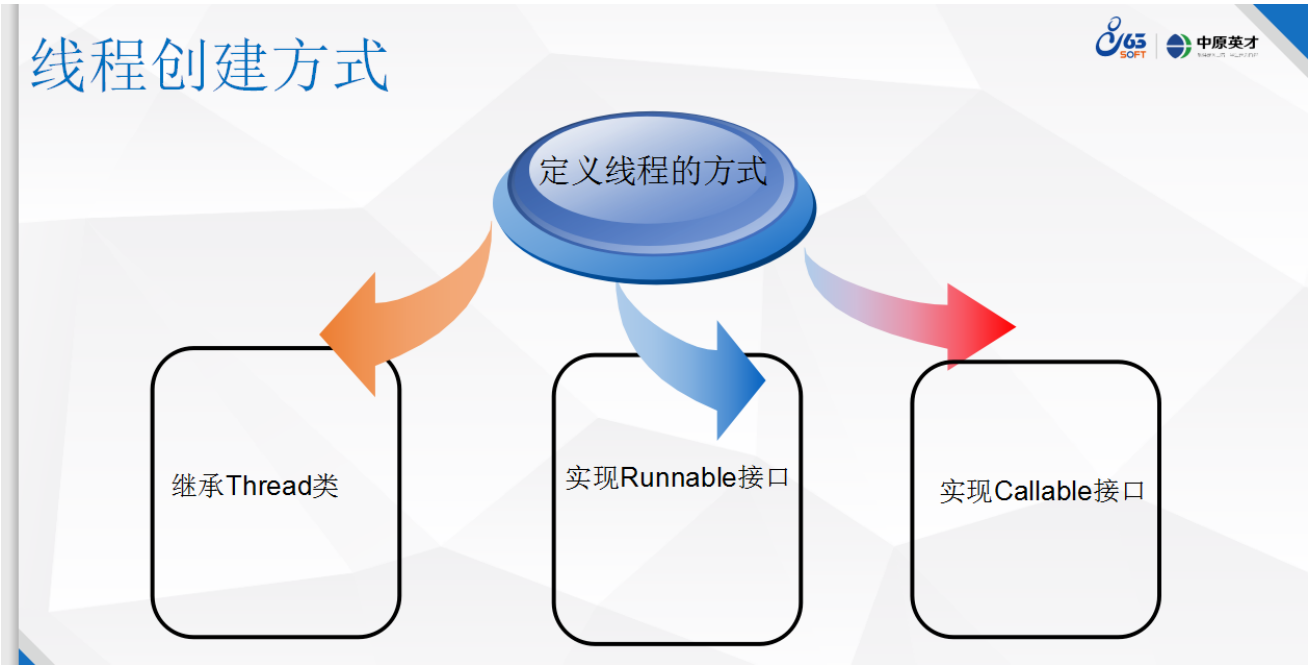
2、并发与并行

- 什么是并发？
  - 指在同一时刻只能有一条指令执行，但多个进程指令被快速的轮换执行，使得在宏观上具有多个进程同时执行的效果，但在微观上并不是同时执行的，只是把时间分成若干段，使多个进程快速交替的执行。
- 什么是并行?
  - 指在同一时刻，有多条指令在多个处理器上同时执行





3、线程的创建方式



3-1、Thread类

```
1  public class Test01_thread
2  {
3      public static void main(String[] args)
4      {
5          //常规运行
6  //        Mythread m1 = new Mythread("A程序");
7  //        Mythread m2 = new Mythread("B程序");
8  //
9  //        m1.work();
10 //        m2.work();
11
12         //使用线程
13         Thread t1 = new Mythread("A线程");
14         Thread t2 = new Mythread("B线程");
15         //run方法并不能开启线程，只是一个普通的方法
16 //        t1.run();
17 //        t2.run();
18         //开启线程，抢夺CPU资源
19         t1.start();
20         t2.start();
21
22     }
23 }
24
25 class Mythread extends Thread
26 {
27     private String name;
28     public Mythread(String name)
29     {
30         this.name = name;
31     }
```

```
    public void work()
    {
        while(true)
        {
            System.out.println(name+"执行了");
        }
    }

    //需要重写父类中的run方法

    @Override
    public void run()
    {
       work();
    }
}
```

3-2、Runnable 接口

```
public class Test01_runnable
{
    public static void main(String[] args)
    {
        //启动runnable线程
        Runnable r = new Son();

        //线程的启动都是依靠thread类
        //把我们写的runnable实现类，作为参数传递给线程thread
        //这个时候thread线程就会自动的去执行我们的run方法
        Thread t = new Thread(r);
        Thread t2 = new Thread(r);
        //我们需要线程的名字，来区分哪个线程执行了，哪个线程没有执行
        t.setName("线程1");
        t2.setName("线程2");
        //线程启动
        t.start();
        t2.start();
    }
}
class Father
{

}
//如果我们自己写的类出现了下面这种情况，这时候，就不能再去继承thread类了，因为Java是单继承
//这种情况下，我们只能去使用runnable接口
class Son extends Father implements Runnable
{

    @Override
    public void run()
    {
        int num = 0;
```

```
        while(num < 100)
        {
            //获取当前执行的线程
            String name = Thread.currentThread().getName();

            //我们可以认为的操作线程的执行
            if(name.equals("线程1") && num < 30)
            {
                System.out.println(name+":正在执行"+num);
            }
            else if (name.equals("线程2") && num > 30)
            {
                System.out.println(name+":正在执行"+num);
            }
            num++;
        }
    }
}
```

3-3、`Callable`接口

可以有返回值

```
public class Test01_callable
{
    public static void main(String[] args) throws ExecutionException, InterruptedException
    {
        MyCallable my = new MyCallable();

        FutureTask<String> ft = new FutureTask<>(my);

        Thread t = new Thread(ft);
        //启动
        t.start();
        //获取到返回值
        String result = ft.get();

        System.out.println(result);
    }
}
class MyCallable implements Callable<String>
{

    @Override
    public String call() throws Exception
    {
        String msg =Thread.currentThread().getName()+"说你好";
        return msg;
    }
}
```

3-4模拟多文件下载(简单版)

```
public class Test02_模拟文件下载
{
    public static void main(String[] args) throws Exception
    {

        long begin = System.currentTimeMillis();
        String a_url = "c:\\video\\a.mp4";
        String b_url = "c:\\video\\b.mp4";


        DownloadFile f1 = new DownloadFile(a_url);
        DownloadFile f2 = new DownloadFile(b_url);

//        f1.download();
//        f2.download();
        //给线程设置名字,setName来源父类Thread
        f1.setName("A线程");
        f2.setName("B线程");

        f1.start();
        f2.start();

        long end = System.currentTimeMillis();
        System.out.println("main主线程运行时间："+(end-begin)+"毫秒");

}



}
class DownloadFile extends Thread
{
    private String old_url="src\\download\\file.mp4";
    private String new_url;

    public DownloadFile(String new_url)
    {
        this.new_url = new_url;
    }

    public void download() throws Exception
    {
        System.out.println(getName()+"开始下载文件...");
        long begin = System.currentTimeMillis();
        //根据要操作的文件的类型（不是处理字符串的，统一使用字节流）
        File old_file = new File(old_url); //要下载的文件
        File new_file = new File(new_url);  //要存放的文件位置
        //如果有该文件，就删掉，方便我们多次测试
        if(new_file.exists())

        {
```

```
            new_file.delete();
        }
        //创建文件，一定可以创建出来
        new_file.createNewFile();

        FileInputStream fis = new FileInputStream(old_file);  //从该文件中读取数据给内存
        FileOutputStream fos = new FileOutputStream(new_file);  //向新文件输出内容

        //为了提高效率
        BufferedInputStream bufferedInputStream = new BufferedInputStream(fis);
        BufferedOutputStream bufferedOutputStream = new BufferedOutputStream(fos);

        //每次读写的大小
        byte[] bs = new byte[1024*1024*4];  //一次读取4MB的数据

        //边读边写
        int num;
        while((num = bufferedInputStream.read(bs)) != -1)
        {
            //在内存中展示数据
            System.out.println(num);
            //写入到新文件中
            bufferedOutputStream.write(bs,0,num);
        }

        bufferedInputStream.close();
        //内存中清空资源
        bufferedOutputStream.flush();
        bufferedOutputStream.close();

        long end = System.currentTimeMillis();

        System.out.println(getName()+"运行时间:"+(end - begin)+"毫秒");

    }


    @Override
    public void run()
    {
        try
        {
            download();
        } catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

3-5模拟多文件上传与下载

```
1.先编写UrlHelper
2.FileService
3.main(只能完成单线程，main线程操作)
单线程测试没有问题后
4.FileUploadThread 文件上传的多线程
5.FileDownloadThread文件下载的多线程
```

```
public class UrlHelper
{
    //服务器端（就是咱们创建的项目）文件保存路径
    public static final String BASEURL = "src\\download";
}
```

```
package less4;

import java.io.*;

import static less4.UrlHelper.*;
public class FileService
{
    //用户输入的文件地址（包含了文件名）
    private String url;  //用户填写的地址
    private String cunfang_url;  //用户填写硬盘路径

    //上传：指的是从c盘或D盘将文件拷贝到项目中（BASEURL）下
   public void upload() throws IOException
   {
        baseMethod(url,BASEURL);
   }

   public void download() throws IOException
   {
        baseMethod(url,cunfang_url);
   }

   /*
   oldUrl:读取的文件
   newUrl:写入的文件
    */
   public void baseMethod(String oldUrl,String newUrl) throws IOException
   {
       //oldUlr：我们需要在内存解读的文件（FileInputStream）
       File old_file = new File(oldUrl);

       //FileOutPutStream,只提供文件夹
       //我们提供的只有文件夹的名字，没有文件名，但是我们可以通过old_file.getName()获取到用户上传的文件名
       File new_file =  new File(newUrl+File.separator+old_file.getName());


       //download文件夹  不包含文件名
```

```
        File doloadFile = new File(BASEURL);

        System.out.println("用户输入的存放路径："+newUrl);
        System.out.println("存放的完成路径："+new_file.getAbsolutePath());

        //判断用户输入的地址是否存在
        if(!old_file.exists())
        {
            System.out.println("用户输入文件路径有误，请重新输入");
            return;
        }
        //检查项目的文件夹是否存在
        if(!doloadFile.exists())
        {
            System.out.println("服务器端路径有误，请检查服务器");
            return;
        }
        //如果用户已经上传一次这个文件了，我们就将它删除
        if(new_file.exists())
        {
            new_file.delete();
        }

        //只是创建了一个空的文件
        new_file.createNewFile();

        //拷贝文件:将用户输入的文件中的内容拷贝到我们服务器上的文件中
        FileInputStream fis = new FileInputStream(old_file);
        BufferedInputStream bis = new BufferedInputStream(fis);

        FileOutputStream fos = new FileOutputStream(new_file);
        BufferedOutputStream bos = new BufferedOutputStream(fos);

        int num;
        byte[] bs = new byte[1024*1024*5];  //一次向新文件中存入5MB的内容
        while(  (num = bis.read(bs)) != -1)
        {
            bos.write(bs,0,num);
        }


        bis.close();
        bos.flush();
        bos.close();


        System.out.println(Thread.currentThread().getName()+"操作【"+new_file.getName()+"】文件完成");

    }

    // ~  get And Set

```

```
    public String getUrl()
    {
        return url;
    }

    public void setUrl(String url)
    {
        this.url = url;
    }

    public FileService(String url)
    {
        this.url = url;
    }

    public String getCunfang_url()
    {
        return cunfang_url;
    }

    public void setCunfang_url(String cunfang_url)
    {
        this.cunfang_url = cunfang_url;
    }
}
```

```
package less4;

import java.util.Scanner;

public class Test
{
    public static void main(String[] args) throws Exception
    {
        Scanner sc = new Scanner(System.in);
        while(true)
        {
            System.out.println("1.上传文件");
            System.out.println("2.下载文件");
            System.out.println("请选择：");
            int choose = sc.nextInt();
            switch (choose)
            {
                case 1:
                    System.out.println("请输入您想要上传的文件路径：");
                    String url_upload = sc.next();
                    FileService uploadSvr = new FileService(url_upload);

//                    uploadSvr.upload();  单线程

                    FileUploadThread thread = new FileUploadThread(uploadSvr);

                    thread.start();
```

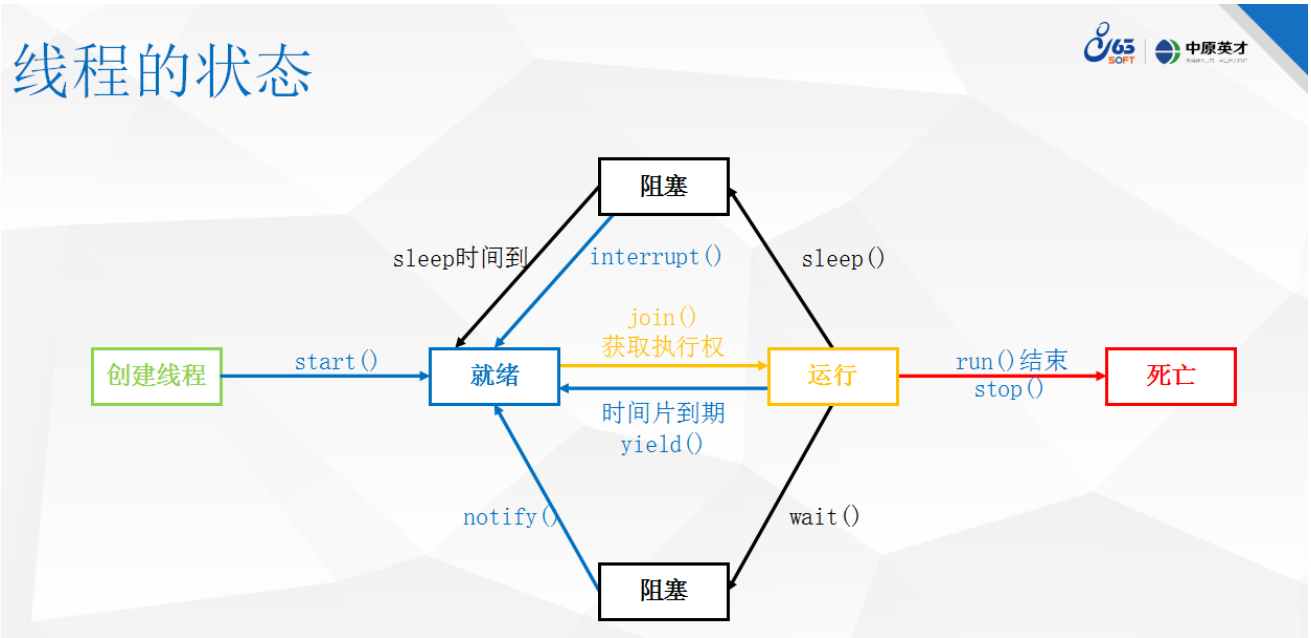
```
                    break;
                case 2:
                    System.out.println("请输入你想要下载的文件名称：");
                    String url_download = sc.next();  //src\\download\\a.mp4

                    System.out.println("请输入您想要存放的路径："); //  c:\\download
                    String url_cunfang = sc.next();

                    FileService downloadSvr = new FileService(url_download);
                    downloadSvr.setCunfang_url(url_cunfang);

                    //main下载
//                    downloadSvr.download();
                    FileDownloadThread downloadThread = new FileDownloadThread(downloadSvr);
                    downloadThread.start();
                    break;

            }
        }

    }
}
```

```
package less4;

import java.io.IOException;

public class FileUploadThread extends Thread
{
    FileService svr;
    public FileUploadThread(FileService svr)
    {
        this.svr = svr;
    }
    @Override
    public void run()
    {
        try
        {
            svr.upload();
        } catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

```
package less4;

import java.io.IOException;

```

```java
public class FileDownloadThread extends Thread
{
    private FileService svr;
    public FileDownloadThread(FileService svr)
    {
        this.svr = svr;
    }

    @Override
    public void run()
    {
        try
        {
            svr.download();
        } catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

4、线程状态



线程的状态

1、线程刚创建时，是new状态

2、线程调用了start()方法后，进入runnable状态，此时并未真正执行，需要和其他线程竞争cpu资源

3、当该线程竞争到了CPU资源，进入running状态

4、线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态之间处于blocked状态

（1）等待阻塞：运行的线程执行wait()方法，该线程会释放占用的所有资源，JVM会把该线程放入“等待池”中，进入这个状态后，是不能自动唤醒的，必须依靠其他线程调用notify()或notifyAll()方法才能被唤醒，
（2）同步阻塞：运行的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入“锁池”中。
（3）其他阻塞：运行的线程执行sleep()或join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。
当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。

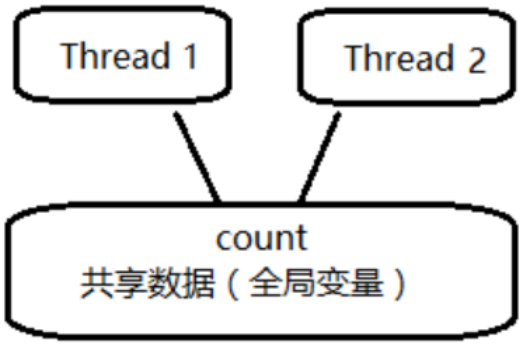
5、当线程正常执行结束会进入dead状态（一个未捕获的异常也会使线程终止）

方法	描述
getName()	获取线程名字
setName(String name)	s设置线程名字
currentThread()	获取当前线程对象
start()	开启线程
sleep(long time)	线程睡眠
join()	线程礼让
stop()	线程停止
wait()/wait(long time)	线程等待
notify()/notifyAll()	线程唤醒

5、线程同步

5-1、什么是线程同步

当多个线程同时共享同一个全局变量或静态变量，做写的操作（修改变量值）时，可能会发生数据冲突问题，也就是线程安全问题。但是做读操作时不会发生数据冲突问题。



两个线程同时修改
count变量（写操作），可能会导致数据错误

5-2、线程同步代码块

```
package less5;

public class Test01_线程同步
{
    /*
    为啥有线程同步？
    因为可能出现多个线程在抢占资源时，同时进行，对一个变量同时进行了修改，那么程序显示上就会出现不可预测的问题，我们管这个问题叫做
    线程的并发
    线程的同步就是为了解决并发问题，让关键性数据在同一时间，只能被一个线程所操作，就可以保证数据的准确性
     */
    public static void main(String[] args)
    {
        Ticket t = new Ticket();   //使用都是t对象中的number

        //构建3个线程，模拟3个窗口，每个窗口卖4张票
        Thread t1 = new Thread(t);
        Thread t2 = new Thread(t);
        Thread t3 = new Thread(t);
        t1.setName("A窗口");
        t2.setName("B窗口");
        t3.setName("C窗口");

        t1.start();
        t2.start();
        t3.start();

    }
}
class Ticket implements Runnable
{

    int number = 10;   //一共有10张票


    @Override
    public void run()
    {
        //让每个线程只循环4次，模拟每个线程卖4张票
```

```
        for(int i = 0 ; i < 4 ; i++)
        {
            //同步代码块：保证代码块中的代码都执行完毕了，再让下一个线程使用
            synchronized (this)  //锁对象：要保证访问的线程都使用同一个对象
            //this表示的是当前对象，就是谁掉用了这个方法，对象就是谁
            {
                if(number <= 0)
                {
                    System.out.println("卖完了");
                    break;
                }

                number--;  //1  - 1 = 0  A(1-1),B(0-1)
                System.out.println(Thread.currentThread().getName()+":卖了一张票，还剩"+number+"张票");
            }


            //模拟现实操作，进入等待售票售票环节
            try
            {
                Thread.sleep(1000);
            } catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }
}
```

5-3、线程同步方法

```
package less5;

public class Test02_线程同步方法
{
    public static void main(String[] args)
    {
        //保证调用对象的方法，是同一个，同步才能使用
        Ticket2 t = new Ticket2();

        Thread t1 = new Thread(t);
        Thread t2 = new Thread(t);
        Thread t3 = new Thread(t);

        t1.start();//Thread-0
        t2.start();//Thread-1
        t3.start();//Thread-2
    }
}

class Ticket2 implements Runnable
```

```
{
    int number = 10;

    public synchronized void saleTicket()
    {
        if (number <= 0)
        {
            System.out.println("票卖完了");
            return;
        }

        number--;
        System.out.println(Thread.currentThread().getName() + ":卖了一张票，还剩" + number + "张");
    }

    @Override
    public void run()
    {
        //不需要同步的代码写在run里面
        for (int i = 0; i < 4; i++)
        {
            //要求线程执行完毕后，在开放CPU的抢占
            saleTicket();
        }
    }
}
```

5-4、线程同步失效

```
package less5;

public class Test03_线程同步失效
{
    public static void main(String[] args)
    {
        //同步失效的原因:没有保证对象的统一
        Ticket3 ti1 = new Ticket3();//10
        Ticket3 ti2 = new Ticket3();//10
        Ticket3 ti3 = new Ticket3();//10

        Thread t1 = new Thread(ti1);
        Thread t2 = new Thread(ti2);
        Thread t3 = new Thread(ti3);

        t1.start();
        t2.start();
        t3.start();

    }
}

class Ticket3 implements Runnable
```

```
{
    int number = 10;

    public synchronized void saleTicket()
    {
        if (number <= 0)
        {
            System.out.println("票卖完了");
            return;
        }

        number--;
        System.out.println(Thread.currentThread().getName() + ":卖了一张票，还剩" + number + "张");
    }

    @Override
    public void run()
    {
        //不需要同步的代码写在run里面
        for (int i = 0; i < 4; i++)
        {
            //要求线程执行完毕后，在开放CPU的抢占
            saleTicket();
        }
    }
}
```

5-5、线程同步静态代码块

```
package less5;

public class Test04_静态同步问题
{
    public static void main(String[] args)
    {

        Ticket4 ti1 = new Ticket4();//10
        Ticket4 ti2 = new Ticket4();//10
        Ticket4 ti3 = new Ticket4();//10

        Thread t1 = new Thread(ti1);
        Thread t2 = new Thread(ti2);
        Thread t3 = new Thread(ti3);

        t1.start();
        t2.start();
        t3.start();

        //静态变量是通过类型.属性
        System.out.println(Ticket4.number);
    }
```

```
}
class Ticket4 implements  Runnable
{
    //number就不再属于对象了，而属于类，所有对象共有这个属性
    static int number = 10;




    @Override
    public void run()
    {
        for(int i = 0 ; i < 4 ;i++)
        {
            //这里锁的还是this，如果this的对象不一样，那么同步将会失效
            //如果锁的是静态属性，那么不要去锁对象，转而锁类名
            synchronized (this.getClass())  //Ticket4.class表示的类名
            {
                if (number <= 0)
                {
                    System.out.println("票卖完了");
                    return;
                }

                number--;
                System.out.println(Thread.currentThread().getName() + ":卖了一张票，还剩" +
number + "张");
            }
        }
    }
}
```

5-6、线程同步静态方法

```
package less5;

public class Test05_静态同步方法
{
    public static void main(String[] args)
    {

        Ticket5 ti1 = new Ticket5();//10
        Ticket5 ti2 = new Ticket5();//10
        Ticket5 ti3 = new Ticket5();//10

        Thread t1 = new Thread(ti1);
        Thread t2 = new Thread(ti2);
        Thread t3 = new Thread(ti3);

        t1.start();
        t2.start();
        t3.start();
```

```
19
20
21      }
22  }
23  class Ticket5 implements Runnable
24  {
25      static int number = 10;
26
27      //如果要去同步静态变量，那么就把该方法变成静态同步方法
28      public static synchronized void saleTicket()
29      {
30          if (number <= 0)
31          {
32              System.out.println("票卖完了");
33              return;
34          }
35
36          number--;
37          System.out.println(Thread.currentThread().getName() + ":卖了一张票，还剩" + number +
    "张");
38      }
39
40      @Override
41      public void run()
42      {
43          //不需要同步的代码写在run里面
44          for (int i = 0; i < 4; i++)
45          {
46              //要求线程执行完毕后，在开放CPU的抢占
47              saleTicket();
48          }
49      }
50  }
```

5-7、同步锁

```
1   package less5;
2
3   import java.util.concurrent.locks.Lock;
4   import java.util.concurrent.locks.ReentrantLock;
5
6   public class Test06_同步锁
7   {
8       public static void main(String[] args)
9       {
10          Ticket6 t = new Ticket6();
11
12          for(int i = 0 ; i < 3 ; i++)
13          {
14              Thread thread = new Thread(t);
15
16              thread.start();
```

```
        }

    }
}

class Ticket6 implements Runnable
{
    int number = 10;

    Lock lock = new ReentrantLock();
    @Override
    public void run()
    {
        //让每个线程只循环4次，模拟每个线程卖4张票
        for (int i = 0; i < 4; i++)
        {
            //上锁，保证后续代码执行完毕，其他线程才可以使用
            lock.lock();
            if (number <= 0)
            {
                System.out.println("卖完了");
                break;
            }

            number--;  //1  - 1 = 0  A(1-1),B(0-1)
            System.out.println(Thread.currentThread().getName() + ":卖了一张票，还剩" + number + "张票");
            lock.unlock();//解锁，其他线程可以抢占资源了
        }


        //模拟现实操作，进入等待售票售票环节
        try
        {
            Thread.sleep(1000);
        } catch (InterruptedException e)
        {
            e.printStackTrace();
        }

    }
}
```

5-8、死锁问题

1. 使用了同步机制，资源同一时间只能被单个线程占有
2. 占有的资源不可以强行剥夺(外部无法操作资源)
3. 多线程互相等待彼此占有的资源锁释放

```
package less5;

import java.util.Date;
```

```
4
5  public class Test07_线程死锁
6  {
7      public static void main(String[] args)
8      {
9          LockA a = new LockA();
10         LockB b = new LockB();
11
12         Thread t1 = new Thread(a);
13         Thread t2 = new Thread(b);
14
15         t1.start();
16         t2.start();
17     }
18 }
19 class MyObject
20 {
21     public static Object o1 = new Object();
22     public static Object o2 = new Object();
23 }
24 class LockA implements  Runnable
25 {
26     @Override
27     public void run()
28     {
29         //2021/8/21  14:31:00
30         System.out.println(new Date().toString() +">>LockA开始执行");
31         //加长程序运行时间
32         while(true)
33         {
34             synchronized (MyObject.o1)
35             {
36                 System.out.println(new Date().toString()+">>LockA锁住了o1对象");
37
38                 try
39                 {
40                     Thread.sleep(3000);
41                 } catch (InterruptedException e)
42                 {
43                     e.printStackTrace();
44                 }
45
46                 synchronized (MyObject.o2)
47                 {
48                     System.out.println(new Date().toString()+">>LockA锁住了o2对象");
49
50                     try
51                     {
52                         Thread.sleep(3000);
53                     } catch (InterruptedException e)
54                     {
55                         e.printStackTrace();
56                     }
```

```
                }
            }
        }
    }
}
class LockB implements  Runnable
{
    @Override
    public void run()
    {
        System.out.println(new Date().toString() +">>LockB开始执行");

        while(true)
        {
            synchronized (MyObject.o2)
            {
                System.out.println(new Date().toString() +">>LockB锁住了o2对象");

                try
                {
                    Thread.sleep(3000);
                } catch (InterruptedException e)
                {
                    e.printStackTrace();
                }

                synchronized (MyObject.o1)
                {
                    System.out.println(new Date().toString() +">>LockB锁住了o1对象");

                    try
                    {
                        Thread.sleep(3000);
                    } catch (InterruptedException e)
                    {
                        e.printStackTrace();
                    }
                }
            }
        }
    }
}
```

6、线程通信

6-1、同步通信

```java
package less5;

public class Test08_同步通信
{
    public static void main(String[] args)
    {
        //如果使用匿名内部类，那么对象一定是不同的，在同步代码的时候需要按照静态同步的处理方式
        Thread t1 = new Thread(new Runnable()
        {
            @Override
            public void run()
            {
                MyTest.say();
            }
        });

        Thread t2 = new Thread(new Runnable()
        {
            @Override
            public void run()
            {
                MyTest.hello();
            }
        });

        t1.start();
        t2.start();
    }
}

class MyTest
{
    public static synchronized void say()
    {
        for(int i = 0 ; i < 3;i++)
        {
            System.out.println("say");
        }
    }

    public static synchronized  void hello()
    {
        for(int i = 0 ; i < 4;i++)
        {
            System.out.println("hello");
        }
    }
}
```

6-2、轮询通信

```java
package less5;

public class Test09_轮询通信
{
    public static void main(String[] args)
    {
        Thread t1 = new Thread(new Runnable()
        {
            @Override
            public void run()
            {
                for(int i = 60 ; i >= 0 ; i--)
                {
                    MyTest2.number++;

                    System.out.println("还剩"+i+"秒结束");

                    try
                    {
                        Thread.sleep(1000);
                    } catch (InterruptedException e)
                    {
                        e.printStackTrace();
                    }
                }
            }
        });

        Thread t2 = new Thread(new Runnable()
        {
            @Override
            public void run()
            {
                while(true)
                {
                    System.out.println(Thread.currentThread().getName()+">>在卖货物");

                    if(MyTest2.number == 60)
                    {
                        System.out.println(Thread.currentThread().getName()+">>结束");

                        break;
                    }

                    try
                    {
                        Thread.sleep(800);
                    } catch (InterruptedException e)
                    {
                        e.printStackTrace();
                    }
                }
            }
```

```
        });

        //让多个线程去操作同一个变量，一个线程负责修改这个变量，另一个线程判断当该变量满足了一定条件，就把自己这个线程结束掉
        t1.start();
        t2.start();
    }
}
class MyTest2
{
    public static int number = 0;
}
```

6-3、等待与唤醒通信

```
package less5;

public class Test10_等待与唤醒通信
{
    public static void main(String[] args) throws InterruptedException
    {
        Thread t1 = new Thread(new Runnable()
        {
            @Override
            public void run()
            {
                synchronized (MyTest3.o)
                {
                    System.out.println("T1线程执行了");

                    try
                    {
                        System.out.println("T1线程进入等待，状态【阻塞】");
                        MyTest3.o.wait();
                    } catch (InterruptedException e)
                    {
                        e.printStackTrace();
                    }

                    System.out.println("T1线程执行完毕");
                }



            }
        });

        Thread t2 = new Thread(new Runnable()
        {
            @Override
            public void run()

            {
```

```
                synchronized (MyTest3.o)
                {
                    for(int i = 0 ; i < 10;i++)
                    {
                        MyTest3.number++;

                        System.out.println("T2线程执行了,number为"+MyTest3.number);

                        if(MyTest3.number == 5)
                        {
                            System.out.println("通知T1线程准备执行，状态【就绪】");
                           MyTest3.o.notify(); //唤醒等待的线程 t1
                            //但是有同步代码块的原因，会保证T2线程先执行完毕，再执行T1线程
                        }

                        try
                        {
                            Thread.sleep(100);
                        } catch (InterruptedException e)
                        {
                            e.printStackTrace();
                        }
                    }
                }

            }
        });

        //waite与notify
        //1、保证多个线程必须有同步代码块或者同步方法
        //2、保证多个线程锁的是同一个对象
        //3、一个被同步代码锁住的线程，可以使用waite方法主动让出cpu使用权
        //4.在其他同步线程中，可以使用notify方法唤醒等待线程，等待的线程会在该同步线程执行完毕后，再执行
        //5、总体而言符合同步要求

        t1.start();
        Thread.sleep(1000);
        t2.start();
    }


}
class MyTest3
{
    public static int number = 0;
    public static Object o = new Object();

}
```

6-4、高效率通信

```java
package less5;

public class Test11_高效率通信
{
    public static void main(String[] args) throws InterruptedException
    {
        MyTest4 thread=new MyTest4();

        thread.start();

        Thread.sleep(3000);
        //修改一个正在运行线程中的变量，正在运行的线程可以发生变化么？
        thread.flag = false;//属性 ：私有空间中，加上volatile，会去修改公共区域了
    }
}
class MyTest4 extends Thread
{
    //可以修改正在运行线程中的变量，运行中的线程会立刻就可以获取修改内容
   volatile boolean flag = true;
    @Override
    public void run()
    {
        //线程中的flag会去存放公共区域中
       while(flag)
       {
           System.out.println("执行了");

           try
           {
               Thread.sleep(1000);
           } catch (InterruptedException e)
           {
               e.printStackTrace();
           }
       }
    }
}
```

7、线程池

➢线程池：是一个容纳多个线程的容器，其中的线程可以反复使用，省去了频繁创建线程对象的操作， 无需反复创建线程而消耗过多资源，是对线程使用的一种优化。

➢使用线程池中线程对象的步骤：

1. 创建线程池对象。
2. 创建Runnable接口子类对象。
3. 提交Runnable接口子类对象。
4. 关闭线程池(一般不关闭)。

```java
package less5;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Test12_线程池
{
    public static void main(String[] args)
    {
        MyTest5 m = new MyTest5();
        //数字表示要创建几个线程
        ExecutorService es = Executors.newFixedThreadPool(3);

        for(int i = 0 ; i < 10;i++)
        {
            //只能接受runnable接口，线程池会自动创建，自动启动，自动销毁
            //我们只需告诉线程池执行哪个线程，执行几次
            es.submit(m);
        }

        //关闭线程池，原则不用关闭，系统会已最低的代价维护
        es.shutdown();
    }

}
class MyTest5 implements  Runnable
{
    @Override
    public void run()
    {
        System.out.println(Thread.currentThread().getName());
    }
}
```