

多线程

- 一、进程和线程
 - 1.1 进程
 - 1.2 线程
 - 1.3 进程和线程区别
 - 1.4 线程组成
- 二、创建线程【重点】
 - 2.1 继承Thread类
 - 2.2 课堂案例
 - 2.3 实现Runnable接口
 - 2.4 课堂案例
- 三、线程状态
 - 3.1 线程状态（基本）
 - 3.2 常见方法
 - 3.3 常见方法演示代码
 - 3.3 线程状态
- 四、线程安全【重点】
 - 4.1 线程安全问题演示
 - 4.1.1 局部变量不能共享
 - 4.1.2 不同对象的实例变量不共享
 - 4.1.3 静态资源共享
 - 4.1.4 同一个对象的实例变量共享
 - 4.2 线程安全问题分析
 - 4.3 线程安全问题解决
 - 4.4 同步方法和同步代码块演示
 - 4.4.1 同步代码块
 - 4.4.2 同步方法
 - 4.5 单例模式线程安全问题
 - 4.5.1 饿汉式
 - 4.5.2 饿汉式
- 五、死锁
 - 5.1 什么是死锁?
 - 5.2 死锁案例
- 六、线程通信
 - 6.1 为什么要处理线程间通信?
 - 6.2 线程等待和唤醒机制
 - 6.2 生产者消费者

一、进程和线程

到目前为止，你学到的都是有关顺序编程的知识，即程序中的所有事务在任意时刻都只能执行一个步骤！这就是串行！

当然了编程中，大部分程序任务都是循序执行的，然而，对于有些问题，如果多个任务能同时执行，例如：下载功能等，这样能大大的提高效率，也显得很有必要！如果同时执行多个任务！那么这就是并行也就是所谓的并发！

Java中的多线程就是实现并发变成的技术！

tips：单核cpu的设备上，是伪并发！但是也提升程序性能！在多cpu的环境下，才能真正发挥多线程并发的能力！所以，我们后期想要提升服务器并发能力！终极奥义：《加cpu》

1.1 进程

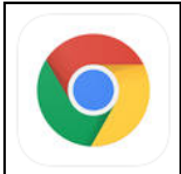


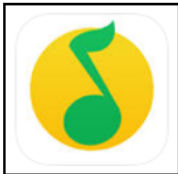
狭义定义：进程是正在运行的程序的实例！当你运行一个程序，你就启动了一个进程。显然，程序是死的(静态的)，进程是活的(动态的)。

广义定义：进程是一个程序运行的数据集合。每一个进程都有它自己的地址空间，一般情况下，包括文本区域（text region）、数据区域（data region）和堆栈（stack region）。文本区域存储处理器执行的代码；数据区域存储变量和进程执行期间使用的动态分配的内存；堆栈区域存储着活动过程调用的指令和本地变量。

特点：

- 单核CPU在任何时间点上。
- 只能运行一个进程。
- 宏观并行、微观串行。

进程



任务管理器

文件(F) 选项(O) 查看(V)

进程 性能 应用历史记录 启动 用户 详细信息 服务

名称	43% CPU	50% 内存	0% 磁盘	0% 网络
应用 (8)				
> Google Chrome (32 位)	0%	17.0 MB	0 MB/秒	0 Mbps
> iTunes	0%	189.3 MB	0 MB/秒	0 Mbps
> Java(TM) Platform SE binary	0.4%	318.6 MB	0 MB/秒	0 Mbps
> Windows 资源管理器	1.5%	31.5 MB	0 MB/秒	0 Mbps
> WPS Presentation (32 位)	6.3%	163.0 MB	0 MB/秒	0 Mbps
> WPS Writer (32 位)	0%	28.4 MB	0 MB/秒	0 Mbps
> 任务管理器	0.5%	14.8 MB	0 MB/秒	0 Mbps
> 迅雷 (32 位)	1.8%	40.1 MB	0 MB/秒	0 Mbps

1.2 线程

线程：又称轻量级进程（Light Weight Process）。

- **线程**（英语：thread）是操作系统能够进行运算调度的最小单位。
- 它被包含在进程之中，是进程中的实际运作单位。

- 一条线程指的是[进程](#)中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务。

比如：

- 迅雷是一个进程，当中的多个下载任务即是多个线程。
- Java虚拟机是一个进程，默认包含主线程（main），通过代码创建多个独立线程，与main并发执行。

1.3 进程和线程区别

- 进程是操作系统资源分配的基本单位，而线程是CPU的基本调度单位。
- 一个程序运行后至少有一个进程。
- 一个进程可以包含多个线程，但是至少需要有一个线程。
- 进程间不能共享数据段地址，但同进程的线程之间可以。

1.4 线程组成

Java中的多线程机制是抢占式的，这表示在调度机制会周期的中断线程，将上下文切换到其他线程，从而为每个线程都提供时间片，使用每个线程都会分配到数量合理的时间去执行任务！

任何一个线程都具有基本的组成部分：

- CPU时间片：操作系统（OS）会为每个线程分配执行时间。
- 运行数据：
 - 堆空间：存储线程需使用的对象，多个线程可以共享堆中的对象。
 - 栈空间：存储线程需使用的局部变量，每个线程都拥有独立的栈。
- 线程的逻辑代码。

二、创建线程【重点】

Java中创建线程主要有两种方式：

- 继承Thread类。
- 实现Runnable接口。

2.1 继承Thread类

步骤：

- 编写类、继承Thread。
- 重写run方法。
- 创建线程对象。
- 调用start方法启动线程。

案例演示：

步骤1: MyThread类:

```
public class MyThread extends Thread {  
  
    public MyThread() {  
        // TODO Auto-generated constructor stub  
    }  
}
```

```

public MyThread(String name) {
    super(name);
}

@Override
public void run() {
    for(int i=0;i<100;i++) {
        System.out.println("子线程:"+i);
    }
}
}

```

步骤2:TestMyThread类:

```

public class TestThread {
    public static void main(String[] args) {
        //1创建线程对象
        MyThread myThread=new MyThread();
        myThread.start();//myThread.run()
        //创建第二个线程对象
        MyThread myThread2=new MyThread();
        myThread2.start();
        //主线程执行
        for(int i=0;i<50;i++) {
            System.out.println("主线程====="+i);
        }
    }
}

```

获取线程名称:

- getName()。
- Thread.currentThread().getName()。

```

public void run() {
    for(int i=0;i<100;i++) {
        //this.getId获取线程Id
        //this.getName获取线程名称
        //第一种方式 this.getId和this.getName();
        //System.out.println("线程id:"+this.getId()+" 线程名
称:"+this.getName()+" 子线程....."+i);
        //第二种方式 Thread.currentThread() 获取当前线程
        System.out.println("线程id:"+Thread.currentThread().getId()+" 线程名
称:"+Thread.currentThread().getName()+" 子线程。。。。。。"+i);
    }
}

```

```

public static void main(String[] args) {
    //1创建线程对象
    MyThread myThread=new MyThread("我的子线程1");
    //2启动线程,不能使用run方法
    //修改线程名称
    //myThread.setName("我的子线程1");
}

```

```

myThread.start();//myThread.run()

//创建第二个线程对象
MyThread myThread2=new MyThread("我的子线程2");

//myThread2.setName("我的子线程2");
myThread2.start();

//主线程执行
for(int i=0;i<50;i++) {
    System.out.println("主线程===== "+i);
}
}

```

2.2 课堂案例

实现四个窗口各卖100张票。

```

/**
 * projectName: demos
 *
 * @author: 赵伟风
 * time: 2022/3/3 21:19
 * description:四个窗口各卖1000张票
 */
public class SaleTicketDemo {

    public static void main(String[] args) {

        Ticketwindow w1 = new Ticketwindow("窗口1");
        Ticketwindow w2 = new Ticketwindow("窗口2");
        Ticketwindow w3 = new Ticketwindow("窗口3");
        Ticketwindow w4 = new Ticketwindow("窗口4");

        w1.start();
        w2.start();
        w3.start();
        w4.start();
    }
}

/**
 * 售票窗口
 */
class Ticketwindow extends Thread{

    private int number = 100;

    private String windowName = null;

    public Ticketwindow(String windowName) {
        //调用父类，传入线程名
        super(windowName);
        this.windowName = windowName;
    }
}

```

```

    }

    public Ticketwindow() {

    }

    /**
     * 多线程对应工作方法
     */
    @Override
    public void run() {

        while (true) {

            if (number<=0) {
                System.out.println(windowName+":票已卖完! ");
                break;
            }
            number--;
            System.out.println(Thread.currentThread().getName()+"还剩
下: "+number+" 张票! ");

        }

    }

}

```

2.3 实现Runnable接口

步骤:

- 编写类实现Runnable接口、并实现run方法。
- 创建Runnable实现类对象。
- 创建线程对象，传递实现类对象。
- 启动线程。

案例演示:

MyRunnable类:

```

public class MyRunnable implements Runnable{

    @Override
    public void run() {
        for(int i=0;i<100;i++) {
            System.out.println(Thread.currentThread().getName()+" ....."+i);
        }
    }

}

```

TestMyRunnable类:

```

public class TestRunnable {
    public static void main(String[] args) {

```

```

//1创建MyRunnable对象,表示线程要执行的功能
MyRunnable runnable=new MyRunnable();
//2创建线程对象
Thread thread=new Thread(runnable, "我的线程1");
//3启动
thread.start();

for(int i=0;i<50;i++) {
    System.out.println("main....."+i);
}
}
}

```

2.4 课堂案例

实现四个窗口共卖100根冰棍。

Ticket类:

```

public class Ticket implements Runnable {
    private int ticket=100;//100根冰棍

    @Override
    public void run() {
        while(true) {
            if(ticket<=0) {
                break;
            }
            System.out.println(Thread.currentThread().getName()+" 卖了
第"+ticket+"根冰棍");
            ticket--;
        }
    }
}
}

```

TestTicket类:

```

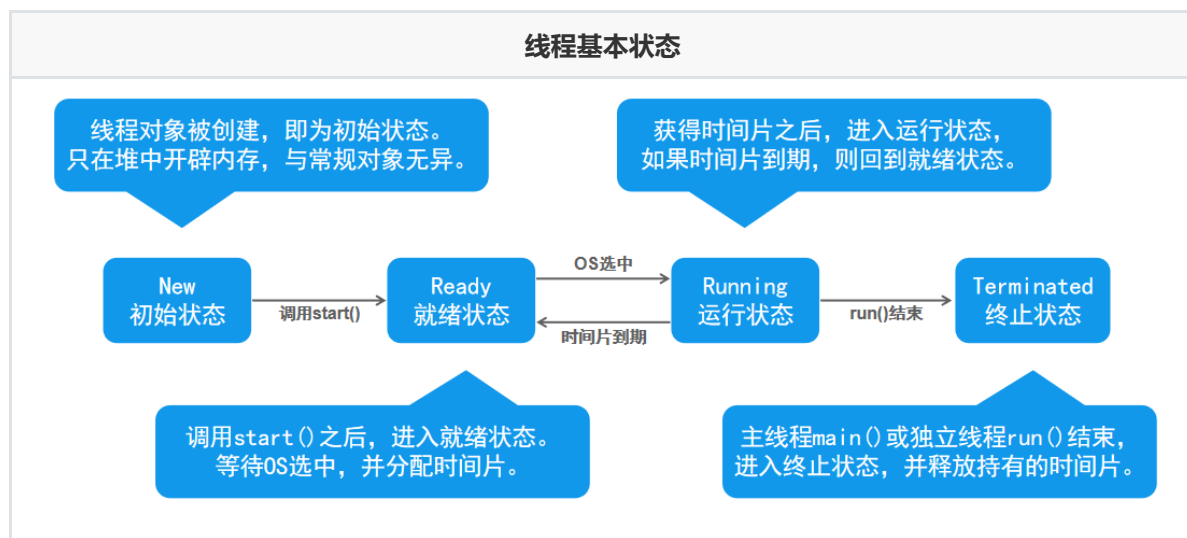
public class TestTicket {
    public static void main(String[] args) {
        //1创建买冰棍对象
        Ticket ticket=new Ticket();
        //2创建线程对象
        Thread w1=new Thread(ticket, "窗口1");
        Thread w2=new Thread(ticket, "窗口2");
        Thread w3=new Thread(ticket, "窗口3");
        Thread w4=new Thread(ticket, "窗口4");
        //3启动线程
        w1.start();
        w2.start();
        w3.start();
        w4.start();
    }
}

```

三、线程状态

3.1 线程状态（基本）

线程状态：新建、就绪、运行、终止。



3.2 常见方法

方法名	说明
public static void sleep(long millis)	当前线程主动休眠 millis 毫秒。
public static void yield()	当前线程主动放弃时间片，回到就绪状态，竞争下一次时间片。一般在当前任务执行完毕，主动释放cpu占有权！
public final void join()	当前线程插队到其他线程直到执行完毕，或者到达等待时间！
public void setPriority(int)	线程优先级为1-10，默认为5,优先级越高，表示获取CPU机会越多。注意优先级低只是获取cpu执行的频率偏低！
public void setDaemon(boolean)	线程有两类：非后台线程、后台线程！后台线程就是提供通用服务的线程，例如检查系统运行状态！不属于程序不可或缺的一部分，当所有非后台程序执行完毕以后，程序结束，程序也终止了，同时也会杀死所有后台线程！反过来说，只要有任何非后台线程还在运行，程序就没有停止！

3.3 常见方法演示代码

1. 休眠演示


```

public static void main(String[] args) {
    for (int i = 10; i>=0; i--) {
        System.out.println(i);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    System.out.println("新年快乐! ");
}

```

2. join加塞演示

```

package com.atguigu.thread.method;

/**
 * projectName: demos
 *
 * @author: 赵伟风
 * time: 2022/3/3 21:42
 * description:子线程加塞主线程
 *     主线程每隔1秒，输出1-10
 *     子线程每隔100毫秒输出 1-100
 *
 * 当主线程输出到3以后，子线程加塞，强制等待子线程输出完毕以后，主线程方可继续输出！
 *
 * 例如：
 *     子线程 1
 *     主线程 1
 *     子线程 2
 *     子线程 3
 *     子线程 4
 *     子线程 5
 *     .....
 *     主线程3
 *     子线程 .... 100
 *     主线程 .... 10
 *
 */
public class JoinThread {

    public static void main(String[] args) throws InterruptedException {

        MyThread myThread = new MyThread();

        myThread.start();

        for (int i = 0; i < 10; i++) {
            System.out.println("主线程 i = " + i);
            Thread.sleep(1000);

            if (i == 3){
                //插队，当前线程【主线程】被myThread线程插队，
                //主线程只能无限的等待下去，直到插队线程执行完毕！
                //myThread.join();
            }
        }
    }
}

```

```

//还可以传入时间毫秒值，代表主线程等待这个时间，插队线程还不完事，也不等
了!

        myThread.join(3000);
    }
}

System.out.println("全部输出完毕!");

}

}

class MyThread extends Thread{

    @Override
    public void run() {

        for (int i = 0; i < 100; i++) {
            System.out.println("子线程i:"+i);
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

    }

}
}

```

3. 守护线程演示

```

public class TestThread {
    public static void main(String[] args) {
        MyDaemon m = new MyDaemon();
        m.setDaemon(true);
        m.start();

        for (int i = 1; i <= 100; i++) {
            System.out.println("main:" + i);
        }
    }
}

class MyDaemon extends Thread {
    public void run() {
        while (true) {
            System.out.println("我一直守护者你...");
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
}

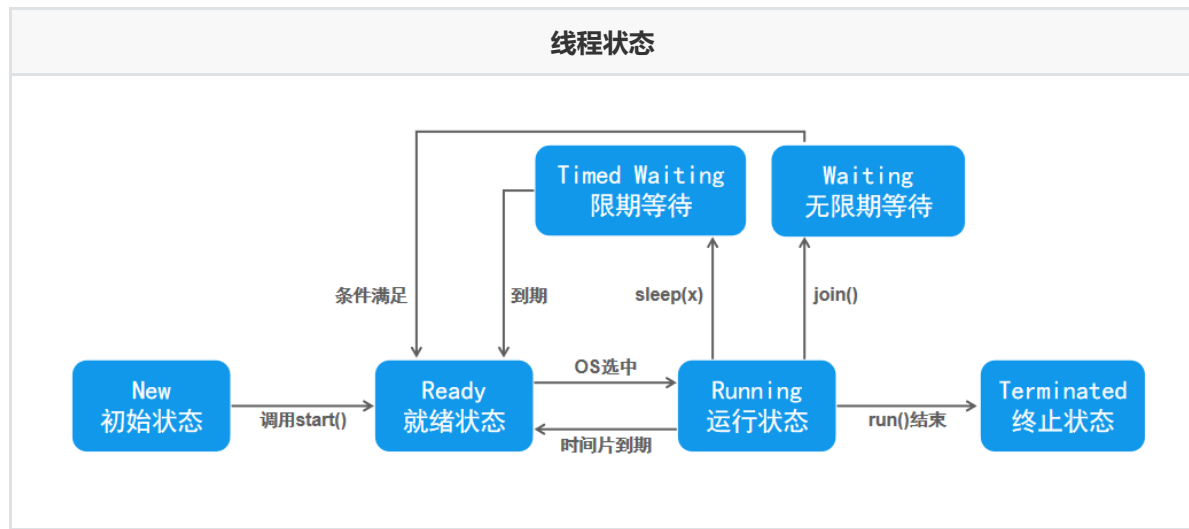
```

3.3 线程状态

线程状态：新建、就绪、运行、等待、终止。

在 `java.lang.Thread` 类内部定义了一个枚举类用来描述线程的六种状态：

```
public enum State {  
    NEW,  
    RUNNABLE,  
    BLOCKED,  
    WAITING,  
    TIMED_WAITING,  
    TERMINATED;  
}
```



四、线程安全【重点】

为什么会出现线程安全问题？

- 当我们使用多个线程访问**同一资源**（可以是同一个变量、同一个文件、同一条记录等）的时候，但是如果多个线程中对资源有读和写的操作，就会出现前后数据不一致问题，这就是线程安全问题。
- 线程不安全：
 - 当多线程并发访问临界资源时，如果破坏原子操作，可能会造成数据不一致。
 - 临界资源：共享资源（同一对象），一次仅允许一个线程使用，才可保证其正确性。
 - 原子操作：不可分割的多步操作，被视作一个整体，其顺序和步骤不可打乱或缺省。

4.1 线程安全问题演示

三个窗口售卖共计100张火车票！

4.1.1 局部变量不能共享

线程使用局部变量定义100张票

```
package com.atguigu.thread.safe;  
  
/**  
 * projectName: demos  
 */
```

```

* @author: 赵伟风
* time: 2022/3/3 22:13
* description:
*/
public class SaleTicketDemo1 {

    public static void main(String[] args) {
        window w1 = new window("窗口1");
        window w2 = new window("窗口2");
        window w3 = new window("窗口3");

        w1.start();
        w2.start();
        w3.start();
    }
}
class Window extends Thread{

    public window() {
    }

    public window(String name) {
        super(name);
    }

    @Override
    public void run(){
        int total = 100;
        while(total>0) {
            System.out.println(getName() + "卖出一张票，剩余:" + --total);
        }
    }
}

```

结果：发现卖出300张票。

问题：局部变量是每次调用方法都是独立的，那么每个线程的run()的total是独立的，不是共享数据。局部变量也不存在线程安全问题！

4.1.2 不同对象的实例变量不共享

定义实例变量，包含售票数量

```

package com.atguigu.thread.safe;

/**
 * projectName: demos
 *
 * @author: 赵伟风
 * time: 2022/3/3 22:13
 * description:
 */
public class SaleTicketDemo2 {

    public static void main(String[] args) {
        window1 w1 = new window1("窗口1");
        window1 w2 = new window1("窗口2");
        window1 w3 = new window1("窗口3");
    }
}

```

```

        w1.start();
        w2.start();
        w3.start();
    }
}

class Window1 extends Thread{

    int total = 100;

    public Window1() {
    }

    public Window1(String name) {
        super(name);
    }

    @Override
    public void run(){

        while(total>0) {
            System.out.println(getName() + "卖出一张票，剩余:" + --total);
        }
    }
}

```

结果：发现卖出300张票。

问题：不同的实例对象的实例变量是独立的。也不存在线程安全问题！

4.1.3 静态资源共享

将售票属性设置为静态变量

```

package com.atguigu.safe;

public class SaleTicketDemo3 {
    public static void main(String[] args) {
        TicketThread t1 = new TicketThread();
        TicketThread t2 = new TicketThread();
        TicketThread t3 = new TicketThread();

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

class TicketThread extends Thread{
    private static int total = 10;
    public void run(){
        while(total>0) {
            try {
                Thread.sleep(10); //加入这个，使得问题暴露的更明显
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

```

```

    }
    System.out.println(getName() + "卖出一张票，剩余:" + --total);
}
}
}

```

结果：发现卖出近100张票。

问题（1）：但是有重复票或负数票问题。

原因：线程安全问题

问题（2）：如果要考虑有两场电影，各卖100张票，这场卖完就没票了，新的线程对象也没有票卖了

原因：TicketThread类的静态变量，是所有TicketThread类的对象共享。本来成员变量就是run方法共享的数据，再用static不合适。

4.1.4 同一个对象的实例变量共享

多个Thread线程使用同一个Runnable对象！

```

package com.atguigu.safe;

public class SaleTicketDemo3 {
    public static void main(String[] args) {
        TicketsSaleRunnable tr = new TicketsSaleRunnable();
        Thread t1 = new Thread(tr, "窗口一");
        Thread t2 = new Thread(tr, "窗口一");
        Thread t3 = new Thread(tr, "窗口一");

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

class TicketsSaleRunnable implements Runnable{
    private int total = 100;
    public void run(){
        while(total>0) {
            try {
                Thread.sleep(10); //加入这个，使得问题暴露的更明显
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName() + "卖出一张票，剩
余:" + --total);
        }
    }
}

```

结果：发现卖出近100张票。

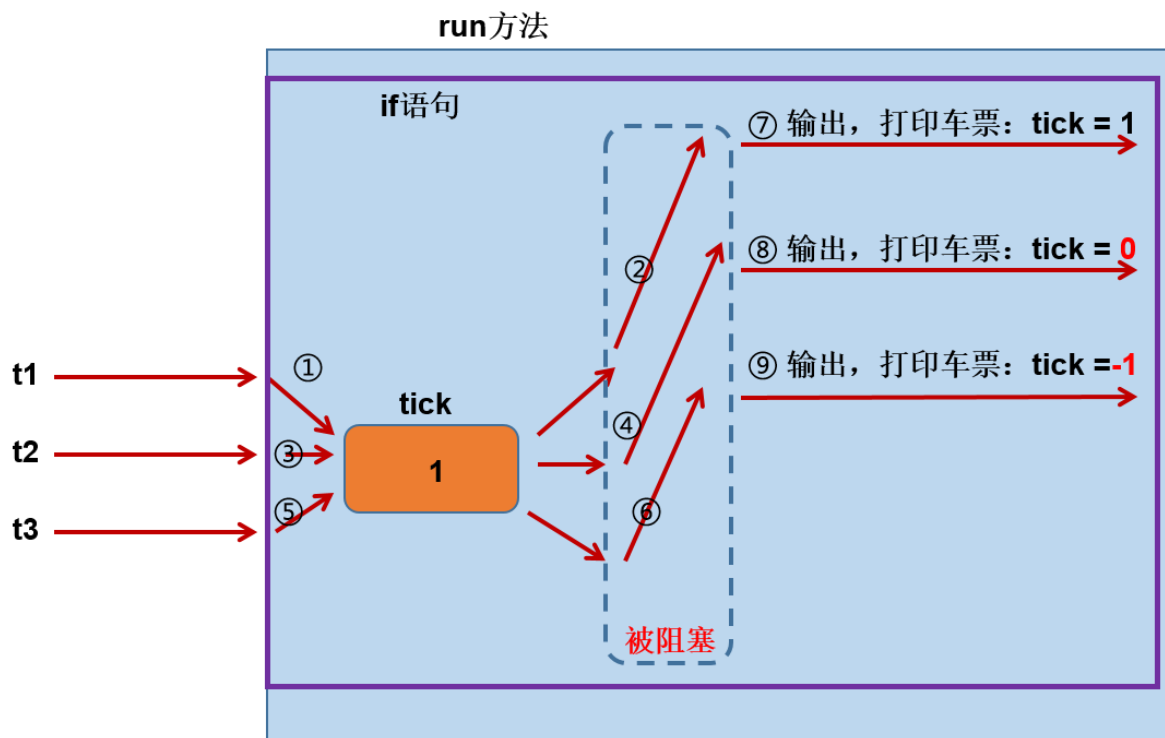
问题：但是有重复票或负数票问题。

原因：线程安全问题

4.2 线程安全问题分析

出现重复打印票和负数的问题分析

跟阻塞没关系，只是放大问题！



总结：线程安全问题的出现因为具备了以下条件

1. 多线程执行
2. 共享数据
3. 多条语句操作共享数据

4.3 线程安全问题解决

共享资源方面真的有多线程么？

线程安全问题的必备条件1和2是我们需要的，要解决只能从第三个点上想办法。要解决上述多线程并发访问一个资源的安全性问题:也就是解决重复票与不存在票问题，Java中提供了**线程同步机制**来解决。



基本上所有的并发模式解决线程冲突上，都是采用序列化的方式访问共享资源的方案！

Java中常使用关键字**synchronized** 来实现同步机制：

1. **同步方法**：synchronized 关键字直接修饰方法，表示同一时刻只有一个线程能进入这个方法，其他线程在外面等着。

```
public synchronized void method(){
    可能会产生线程安全问题的代码
}
```

2. **同步代码块**：synchronized 关键字可以用于某个区块前面，表示只对这个区块的资源实行互斥访问。

格式：

```
synchronized(同步锁){
    需要同步操作的代码
}
```

同步锁对象：

- 锁对象可以是任意类型。
- 多个线程对象 确保使用同一把锁。

4.4 同步方法和同步代码块演示

4.4.1 同步代码块

如何确保同一把锁

- 静态代码块中：使用当前类的Class对象
- 非静态代码块中：习惯上先考虑this，但是要注意是否同一个this

```
//售票线程任务类
class SaleTicket implements Runnable {
    //票数
    private int count = 100;//共享资源

    @Override
    public void run() {
        while (true) {
            //同步代码块，this为锁对象
            synchronized (this) {
                if (count > 0) {
                    System.out.println(Thread.currentThread().getName() + "--" +
count);
                    count--;
                }
            }
        }
    }
}

//测试类
public class DemoSaleTicket {
    public static void main(String[] args) {
        //创建线程任务对象
        SaleTicket st = new SaleTicket(ticket);
        //创建售票线程对象，并启动线程
        new Thread(st).start();
        new Thread(st).start();
    }
}
```



```
}  
}
```

静态代码块锁

```
package com.atguigu.thread.safe;  
  
/**  
 * projectName: demos  
 *  
 * @author: 赵伟风  
 * time: 2022/3/3 22:26  
 * description:  
 */  
public class SafeTicketDemo2 {  
  
    public static void main(String[] args) {  
        TicketThread t1 = new TicketThread();  
        TicketThread t2 = new TicketThread();  
        TicketThread t3 = new TicketThread();  
  
        t1.start();  
        t2.start();  
        t3.start();  
    }  
}  
class TicketThread extends Thread{  
    private static int total = 10;  
  
    @Override  
    public void run(){  
        while(total>0) {  
            //添加同步锁  
            synchronized (TicketThread.class) {  
                //锁里面一定再次判断，否则还是出现安全问题！  
                if (total > 0) {  
                    try {  
                        Thread.sleep(10); //加入这个，使得问题暴露的更明显  
                    } catch (InterruptedException e) {  
                        e.printStackTrace();  
                    }  
                    System.out.println(getName() + "卖出一张票，剩余:" + --total);  
                }  
            }  
        }  
    }  
}
```

4.4.2 同步方法

同步方法锁对象固定：

- 静态方法：当前类的Class对象
- 非静态方法：this

```
public class SaleTicket implements Runnable {
```

```

        private int count = 100;

        @Override
        public void run() {
            while (true) {
                //直接调用同步方法
                sell();
            }
        }
        //将售票方法改进为：同步方法，非静态的同步方法的锁对象默认为this
        private synchronized void sell() {
            if (count > 0) {
                System.out.println(Thread.currentThread().getName() + "--" + count);
                count--;
            }
        }
    }
}

```

96777

```

//改造后，添加跳出循环
//共享资源类（将共享数据与同步方法封装到一个类中）
class Ticket {
    private int count=100;//票数

    /**同步方法，非静态同步方法的锁对象默认为this
    public synchronized void sell() {
        if (count > 0) {
            System.out.println(Thread.currentThread().getName() + "--" + count--);
        }
    }

    public int getCount() {
        return count;
    }
}

//线程任务类
class SaleTicketRunnable implements Runnable {
    //共享的资源
    private Ticket ticket;
    //通过构造器传入共享资源
    public SaleTicketRunnable(Ticket ticket) {
        this.ticket = ticket;
    }
    //线程任务
    @Override
    public void run() {
        while (true) {
            //售票
            ticket.sell();
            if(ticket.getCount()<=0)//售完跳出循环，结束线程任务
                break;
        }
    }
}
}

```

```
//测试类
public class DemoSaleTicket {
    public static void main(String[] args) {
        //创建共享资源
        Ticket ticket = new Ticket();
        //创建资源操作线程对象
        SaleTicket st = new SaleTicket(ticket);
        //创建售票线程对象，并启动线程
        new Thread(st).start();
        new Thread(st).start();
    }
}
```

锁的范围太小：不能解决安全问题，要同步所有操作共享资源的语句。

锁的范围太大：因为一旦某个线程抢到锁，其他线程就只能等待，所以范围太大，效率会降低，不能合理利用CPU资源。

4.5 单例模式线程安全问题

4.5.1 饿汉式

没有线程安全问题，因为上来就开始创建！

```
public class Singleton {
    private final static Singleton INSTANCE = new Singleton();
    private Singleton(){}
    public static Singleton getInstance(){
        return INSTANCE;
    }
}
```

4.5.2 饿汉式

```
public class Singleton {
    private static Singleton ourInstance;

    public static Singleton getInstance() {
        //一旦创建了对象，之后再次获取对象，都不会再进入同步代码块，提升效率
        if (ourInstance == null) {
            //同步锁，锁住判断语句与创建对象并赋值的语句
            synchronized (Singleton.class) {
                if (ourInstance == null) {
                    ourInstance = new Singleton();
                }
            }
        }
        return ourInstance;
    }

    private Singleton() {
    }
}
```

五、死锁

5.1 什么是死锁?

- 当第一个线程拥有A对象锁标记, 并等待B对象锁标记, 同时第二个线程拥有B对象锁标记, 并等待A对象锁标记时, 产生死锁。
- 一个线程可以同时拥有多个对象的锁标记, 当线程阻塞时, 不会释放已经拥有的锁标记, 由此可能造成死锁。

5.2 死锁案例

MyLock类:

```
public class MyLock {  
    //两个锁(两个筷子)  
    public static Object a=new Object();  
    public static Object b=new Object();  
}
```

BoyThread类:

```
public class Boy extends Thread{  
    @Override  
    public void run() {  
        synchronized (MyLock.a) {  
            System.out.println("男孩拿到了a");  
            synchronized (MyLock.b) {  
                System.out.println("男孩拿到了b");  
                System.out.println("男孩可以吃东西了...");  
            }  
        }  
    }  
}
```

GirlThread类:

```
public class Girl extends Thread {  
    @Override  
    public void run() {  
        synchronized (MyLock.b) {  
            System.out.println("女孩拿到了b");  
            synchronized (MyLock.a) {  
                System.out.println("女孩拿到了a");  
                System.out.println("女孩可以吃东西了...");  
            }  
        }  
    }  
}
```

TestDeadLock类:

```

public class TestDeadLock {
    public static void main(String[] args) {
        Boy boy=new Boy();
        Girl girl=new Girl();
        girl.start();
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        boy.start();
    }
}

```

释放锁的几种场景：

当前线程的同步方法、同步代码块执行结束。

当前线程在同步代码块、同步方法中出现了未处理的Error或Exception，导致当前线程异常结束。

当前线程在同步代码块、同步方法中执行了锁对象的wait()方法，当前线程被挂起，并释放锁。

六、线程通信

6.1 为什么要处理线程间通信？

多个线程在处理同一个资源，但是处理的动作（线程的任务）却不相同。而多个线程并发执行时，在默认情况下CPU是随机切换线程的，当我们需要多个线程来共同完成一件任务，并且我们希望他们有规律的执行，那么多线程之间需要一些通信机制，可以协调它们的工作，以此来帮我们达到多线程共同操作一份数据。

比如：线程A用来生成包子的，线程B用来吃包子的，包子可以理解为同一资源，线程A与线程B处理的动作，一个是生产，一个是消费，此时B线程必须等到A线程完成后才能执行，那么线程A与线程B之间就需要线程通信

6.2 线程等待和唤醒机制

这是多个线程间的一种**协作**机制。谈到线程我们经常想到的是线程间的**竞争 (race)**，比如去争夺锁，但这并不是故事的全部，线程间也会有协作机制。

就是在一个线程满足某个条件时，就进入等待状态（**wait()/wait(time)**），等待其他线程执行完他们的指定代码过后再将其唤醒（**notify()**）；或可以指定wait的时间，等时间到了自动唤醒；在有多线程进行等待时，如果需要，可以使用 **notifyAll()**来唤醒所有的等待线程。wait/notify 就是线程间的一种协作机制。

方法	说明
public final void wait()	释放锁，进入等待队列
public final void wait(long timeout)	在超过指定的时间前，释放锁，进入等待队列
public final void notify()	随机唤醒、通知一个线程
public final void notifyAll()	唤醒、通知所有线程

注意：所有的等待、通知方法必须在对加锁的同步代码块中。

1. wait：线程不再活动，不再参与调度，进入 wait set 中，因此不会浪费 CPU 资源，也不会去竞争锁了，这时的线程状态即是 WAITING或TIMED_WAITING。它还要等着别的线程执行一个**特别**的动作，也即是“**通知 (notify)**”或者等待时间到，在这个对象上等待的线程从wait set 中释放出来，重新进入到调度队列（ready queue）中
2. notify：则选取所通知对象的 wait set 中的一个线程释放；
3. notifyAll：则释放所通知对象的 wait set 上的全部线程。

调用wait和notify方法需要注意的细节

1. wait方法与notify方法必须要由同一个锁对象调用。因为：对应的锁对象可以通过notify唤醒使用同一个锁对象调用的wait方法后的线程。
2. wait方法与notify方法是属于Object类的方法的。因为：锁对象可以是任意对象，而任意对象的所属类都是继承了Object类的。
3. wait方法与notify方法必须要在同步代码块或者是同步函数中使用。因为：必须要通过锁对象调用这2个方法。
4. 被通知线程被唤醒后也不一定能立即恢复执行，因为它当初中断的地方是在同步块内，而此刻它已经不持有锁，所以她需要再次尝试去获取锁（很可能面临其它线程的竞争），成功后才能在当初调用 wait 方法之后的地方恢复执行。

6.2 生产者消费者

若干个生产者在生产产品，这些产品将提供给若干个消费者去消费，为了使生产者和消费者能并发执行，在两者之间设置一个能存储多个产品的缓冲区，生产者将生产的产品放入缓冲区中，消费者从缓冲区中取走产品进行消费，显然生产者和消费者之间必须保持同步，即不允许消费者到一个空的缓冲区中取产品，也不允许生产者向一个满的缓冲区中放入产品。

Bread类：

```
public class Bread {
    private int id;
    private String productName;
    public Bread() {
        // TODO Auto-generated constructor stub
    }
    public Bread(int id, String productName) {
        super();
        this.id = id;
        this.productName = productName;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
```

```

        this.id = id;
    }
    public String getProductName() {
        return productName;
    }
    public void setProductName(String productName) {
        this.productName = productName;
    }
    @Override
    public String toString() {
        return "Bread [id=" + id + ", productName=" + productName + "]";
    }
}

```

BreadCon类:

```

public class BreadCon {
    //存放面包的数组
    private Bread[] cons=new Bread[6];
    //存放面包的位置
    private int index=0;

    //存放面包
    public synchronized void input(Bread b) { //锁this
        //判断容器有没有满
        while(index>=6) {
            try {
                this.wait();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }

        cons[index]=b;
        System.out.println(Thread.currentThread().getName()+"生产
了"+b.getId()+"");
        index++;
        //唤醒
        this.notifyAll();
    }

    //取出面包
    public synchronized void output() { //锁this
        while(index<=0) {
            try {
                this.wait();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        index--;
        Bread b=cons[index];
        System.out.println(Thread.currentThread().getName()+"消费了"+b.getId()+"
生产者:"+b.getProductName());
        cons[index]=null;
    }
}

```

```

        //唤醒生产者
        this.notifyAll();
    }
}

```

Consume类: 消费者

```

public class Consume implements Runnable{

    private BreadCon con;

    public Consume(BreadCon con) {
        super();
        this.con = con;
    }

    @Override
    public void run() {
        for(int i=0;i<30;i++) {
            con.output();
        }
    }

}

```

Produce类: 生产者

```

public class Prodcut implements Runnable {

    private BreadCon con;

    public Prodcut(BreadCon con) {
        super();
        this.con = con;
    }

    @Override
    public void run() {
        for(int i=0;i<30;i++) {
            con.input(new Bread(i, Thread.currentThread().getName()));
        }
    }

}

```

测试类

```

public class Test {
    public static void main(String[] args) {
        //容器
        BreadCon con=new BreadCon();
        //生产和消费
        Prodcut prodcut=new Prodcut(con);
        Consume consume=new Consume(con);
        //创建线程对象
    }
}

```



```
Thread chenchen=new Thread(prodcut, "晨晨");
Thread bingbing=new Thread(consume, "消费");
Thread mingming=new Thread(prodcut, "明明");
Thread lili=new Thread(consume, "莉莉");
//启动线程
chenchen.start();
bingbing.start();
mingming.start();
lili.start();
    }
}
```