第十章多线程

南京农业大学
谢忠红

- 1 多线程的概念
- 2 如何创建一个线程
- 3 线程的之间数据通讯
- 4 多线程之间的同步问题
- 5 线程的控制与调度
- 6 死锁
- 7 使用wait()和notify()方法实现线程间通讯
- 8 线程举例
- 9 小结

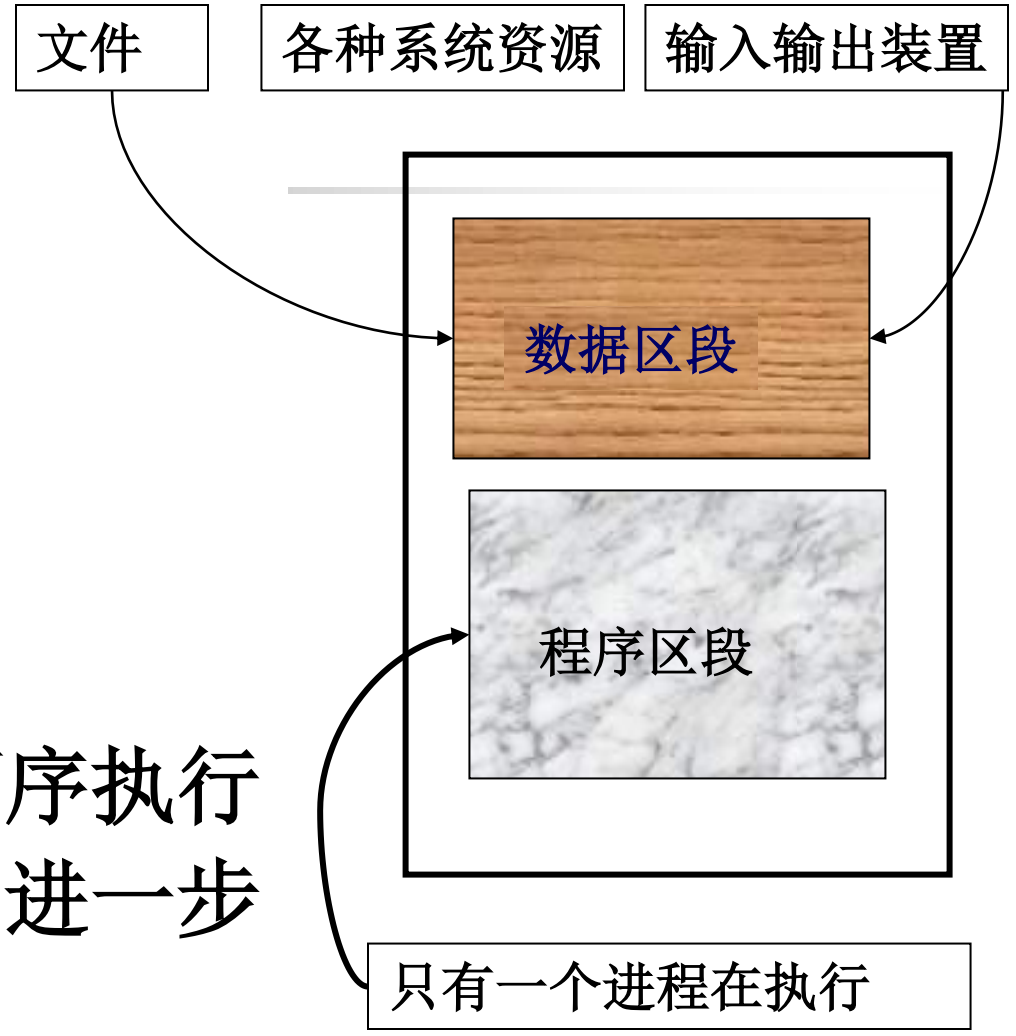
1多线程的概念

- 程序　进程　并发多任务
- 进程：程序的一次运行。
- 进程是系统运行程序的基本单位

（1）进程是一个可拥有资源的独立单位；

（2）同时又是一个可独立调度和分派的基本单位。

- 进程基本操作：
- （1）创建进程
- （2）撤销进程
- （3）进程切换
- 缺点：
- （1）进程的代码顺序执行
- （2）限制了并发的进一步发展



传统的进程

线程

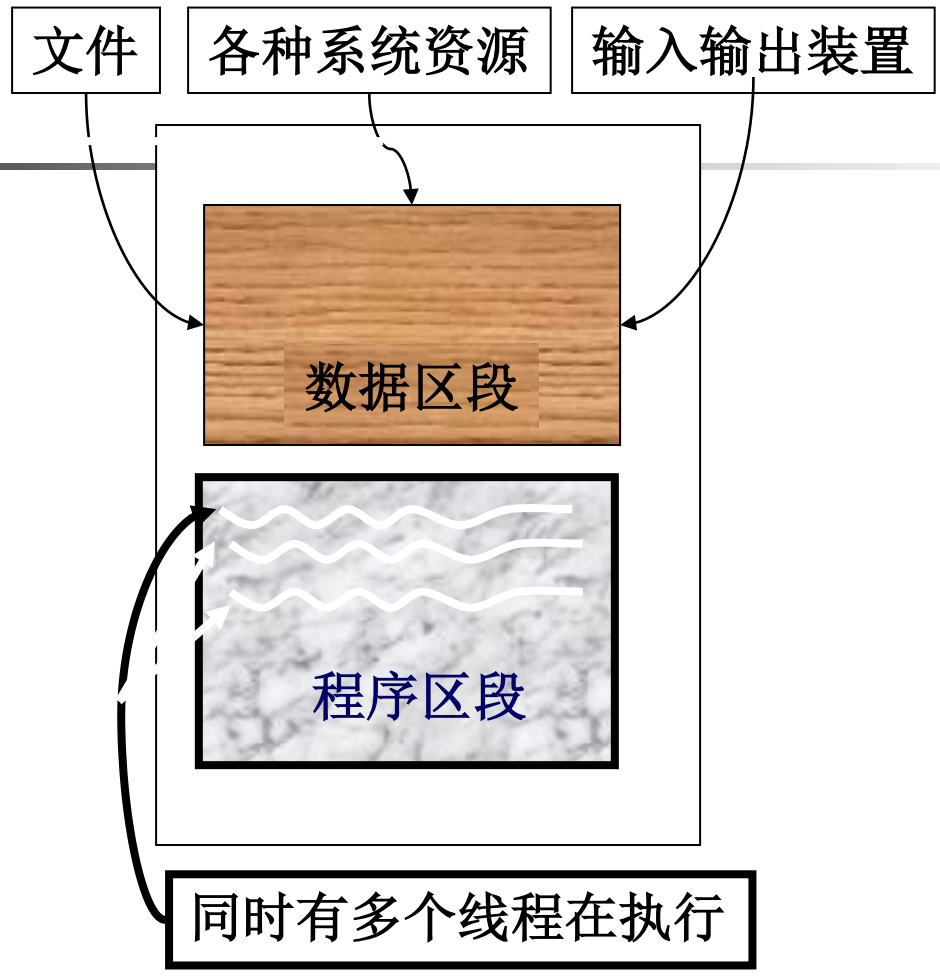
- **DEF:**比进程更小的运行单位，是程序中*单个顺序的流控制*。一个进程可以包含多个线程。

- **基本特征特征:**

- **(1)**轻型实体。基本不拥有系统资源除了一些必要的寄存器和堆栈

- **(2)**共享进程资源（进程的地址空间和已打开的文件定时器等）

- **(3)**独立调度和分派的基本单位（因线程很轻所以线程切换迅速且开销小）



多线程的任务

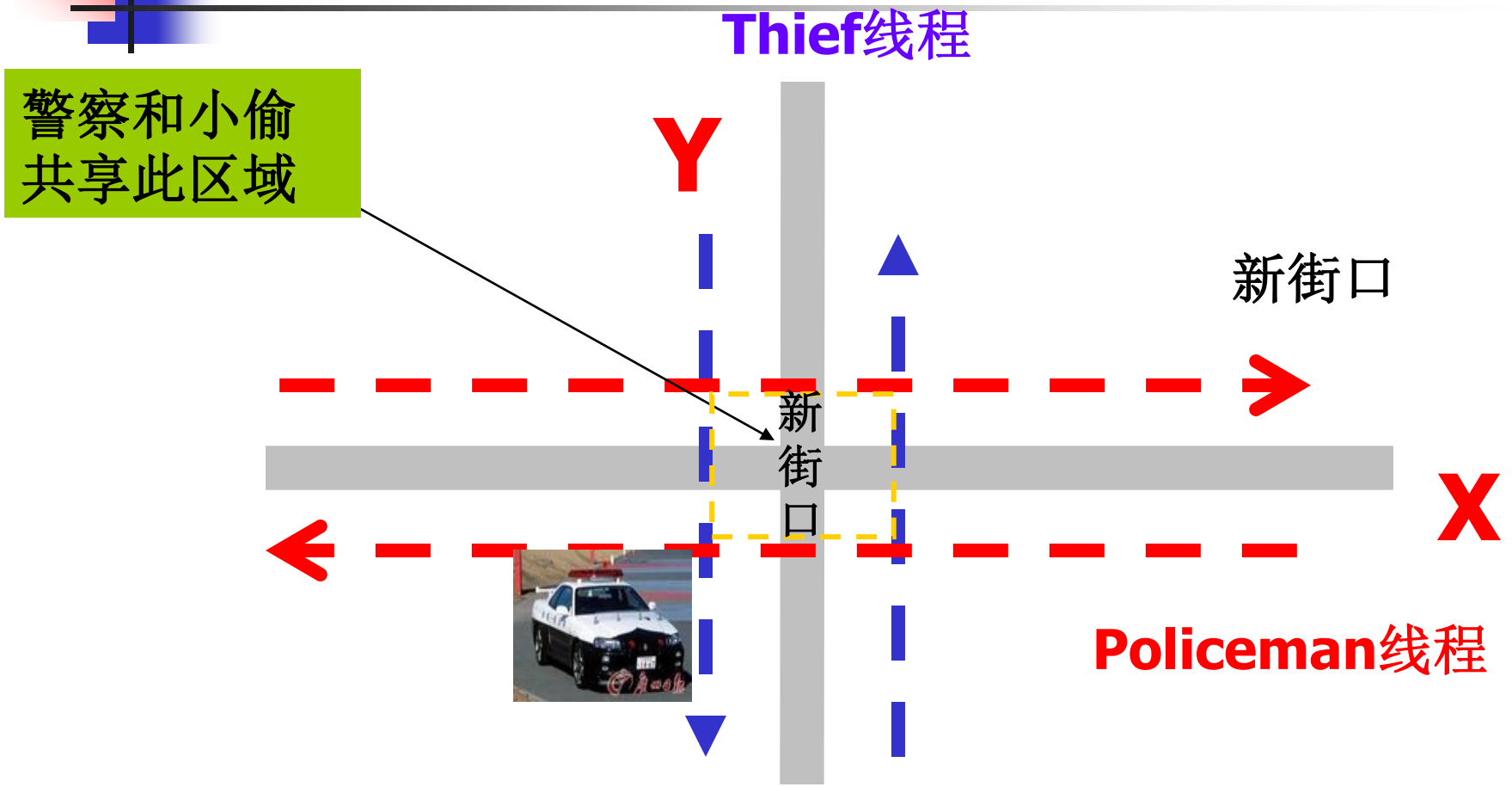
- **线程和进程的比较**
- （1）进程的内部数据和状态都是完全独立的，因而多个进程并发执行时不会相互干扰。
- （2）多个线程是共享一块内存空间和一组系统资源,因而并发执行时可能互相影响。
- **线程与进程比较所拥有的优点**
- （1）线程的切换比进程切换的负担小，并发非常容易。
- （2）由于多线程共存在于同一个内存块中通信非常容易。

- 多线程的概念

- DEF:在多线程模型中，多个线程共存于同一块内存中，且共享资源。

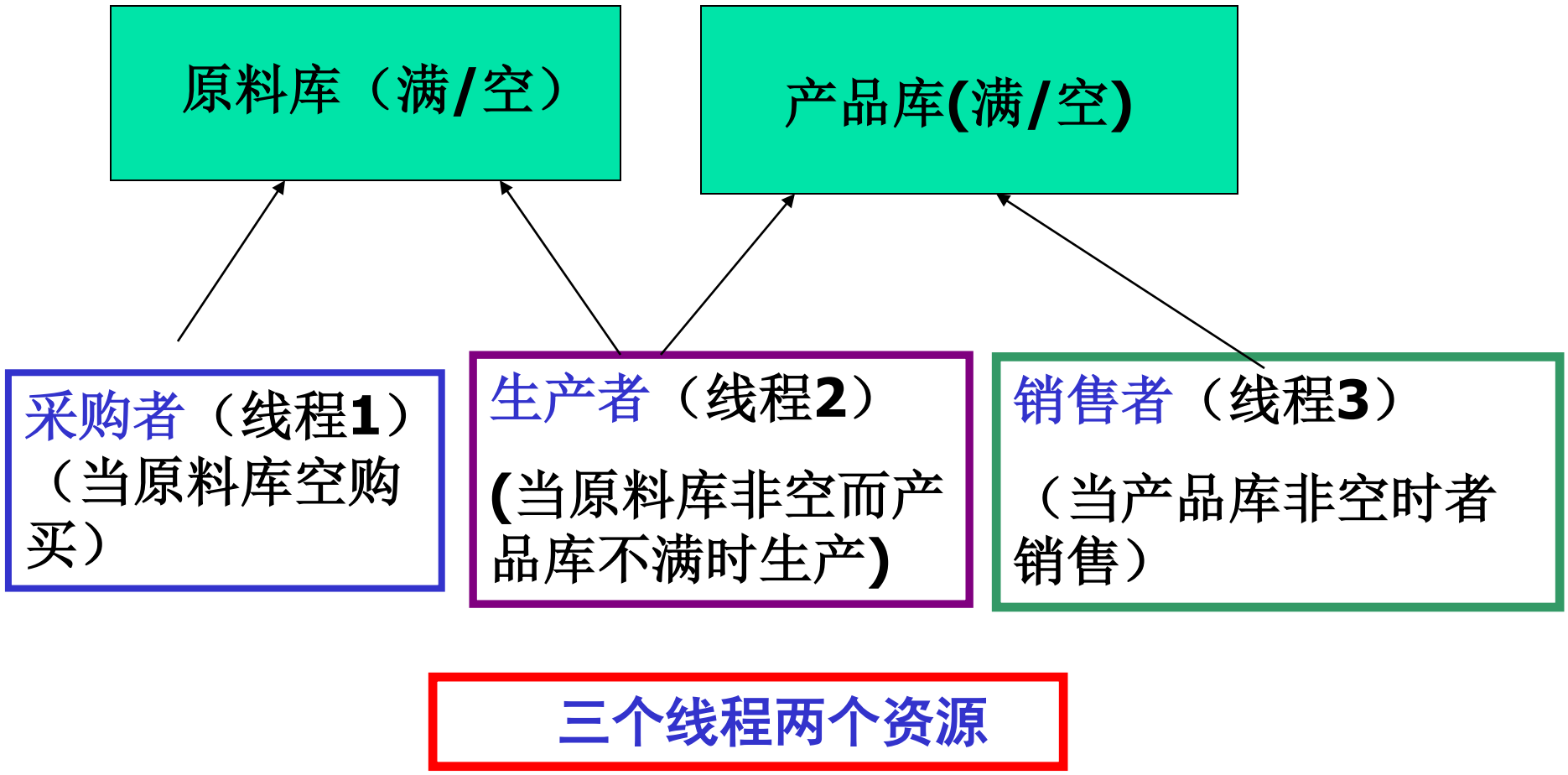
- **调度策略：**时间片轮转，先来先服务、短作业优先、优先权……

一个简单模拟多线程的例子



- 警察抓小偷例子

一个简单的模拟多线程的例子



2创建一个线程

- 方法1： 实现**Runnable**接口创建线程
- 方法2： 继承**Thread**类创建线程类，覆盖其run方法

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

Run是线程执行的起点

```
Class Thread extends Object
                implements Runnable{
    Thread(){...}
    Thread(String name){...}
    Thread(Runnable r) {...}
    Thread (Runnable r,String name){...}
    final String getName(){...}
    void Run(){...}
    void start(){...}
    ......
}
```

- 方法1实现Runnable接口创建线程
- 举例

```
class Policeman  implements  Runnable{
    Policeman(){}
    public void run(){
       while(true){
          for(int x=-100;x<=100;x++)
             System.out.println("police at: "+x);
       }
   }}
```

创建线程对象

```
Policeman   p=new Policeman();
Thread   Pth=new Thread(P , "TomP");
```

- 方法2：继承Thread类创建线程，覆盖
- 其run方法

1. 定义Thread类的一个子类。
2. 定义子类中的方法run( )，覆盖父类中的方法run( )。
3. 创建该子类的一个线程对象。
4. 通过start( )方法启动线

举例

```
class Thief    extends   Thread{
    Thief(String s){  super (s);  }
    Thief(){    super();  }
    public void run(){
        while(true){
            for(int y=-100;y<=100;y++)
                System.out.println("thief at: "+y);
        }
    }
}
```

创建线程对象

```
Thread    Tth=new Thief( “hijackT”);
Tth.start();
```

- 举例：小偷和警察多线程

```
class Catch{
    public static void main(String args[]){
        Policeman   P=new Policeman();
        Thread      Pth=new Thread(P , "TomP");
        Thread     Tth=new Thief( "hijackT‟ );
        Pth.start();
        Tth.start();
      }
}
```

- **运行结果：**
  - police at: -5
  - police at: -4
  - police at: -3
  - thief at: 67
  - police at: -2
  - thief at: 68
  - police at: -1
  - thief at: 69
  - police at: 0
  - thief at: 70
- **说明：** 宏观上看两个线程同时竞争资源

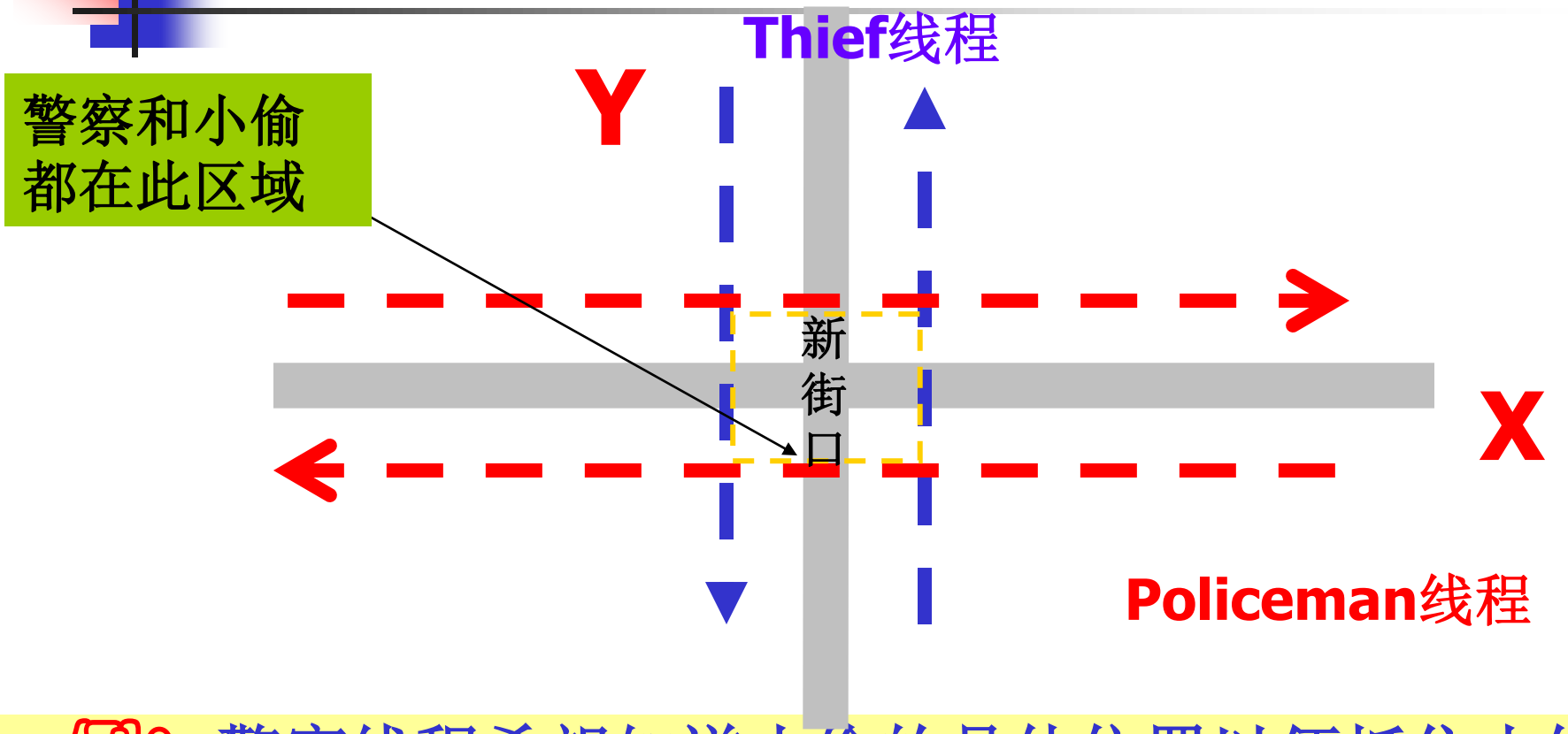
■举例2：看谁跑得快哦

```
class TimePrinter  extends Thread {
    int pauseTime;    String name;
    public TimePrinter(int pauseTime, String name) {
        this.pauseTime =pauseTime;
        this.name = name;   }
    public void run() {
        while(true) {
            try {
                System.out.println(name + ": " + new
                Date(System.currentTimeMillis()));
                Thread.sleep(  pauseTime  );
            } catch(Exception e) {System.out.println(e);}
        }
}
```

```
    public  static  void main(String args[]) {
        TimePrinter  tp1=
             new  TimePrinter( 1000, "Fast Guy");
        tp1. start() ;
        TimePrinter tp2 =
             new TimePrinter( 3000,  "Slow Guy");
        tp2. start() ;
    }
}
```

- 运行结果：
- Slow Guy: Sun Nov 15 21:03:28 CST 2009
- Fast Guy: Sun Nov 15 21:03:28 CST 2009
- Fast Guy: Sun Nov 15 21:03:29 CST 2009
- Fast Guy: Sun Nov 15 21:03:30 CST 2009
- Slow Guy: Sun Nov 15 21:03:31 CST 2009
- Fast Guy: Sun Nov 15 21:03:31 CST 2009
- Fast Guy: Sun Nov 15 21:03:32 CST 2009
- Fast Guy: Sun Nov 15 21:03:33 CST 2009
- Fast Guy: Sun Nov 15 21:03:34 CST 2009
- Slow Guy: Sun Nov 15 21:03:34 CST 2009
- Fast Guy: Sun Nov 15 21:03:35 CST 2009
- 说明：宏观上看两个线程同时竞争CPU资源

3线程间数据间的交流



- 例：警察线程希望知道小偷的具体位置以便抓住小偷，而小偷线程也希望知道警察的位置以便及时躲避。

- 如何使得不同类型的多个线程共享一些数据？

- 方法1：使用内类
- 方法2：使用构造函数传递

方法1：使用内类实现线程间数据交流

```
class testCatchThief  {
    int x ;  //policeman     int y;   //thief
    public static void main(String args[]){
        testCatchThief  x=new testCatchThief();
        x.go(); }
    public  void go(){
        Policeman  P=new Policeman();
        Thread  Pth=new Thread(P,"TomP");
        Thief  Tth=new Thief ("hijackT");
        Tth.start();       Pth.start();    }
```

```
class Policeman  implements  Runnable{
    public void run(){
        while(true){
            for(x=-100;x<=100;x++){
                System.out.println("police at: "+x);
                if(x>=-10&&x<=10&&y>=- 10&&y<=10){
                    System.out.print("police : "+x+");
                    System.out.println(" thief "+y);
                    System.out.println("thief is catched");
                    System.exit(0); }
            }
        }
    }
}
```

```
class Thief extends Thread{
    Thief(String s){super(s);}
    public void run(){
        while(true){
            for(y=-100;y<=100;y++){
                System.out.println("thief at: "+y);
                if(x>=-10&&x<=10&&y>=- 10&&y<=10)
                    System.exit(0);
            }
        }
    }
}
```

- **运行结果：**
- thief at: -19
- thief at: -18
- thief at: -17
- thief at: -16
- thief at: -15
- police at: -7
- police at: -6
- police at: -5
- thief at: -14
- thief at: -13
- thief at: -12
- police at: -4
- thief at: -11
- police at: -3
- thief at: -10
- police at : -3　thief at: -10thief is catched
- 说明：线程共享资源**x,y**

■方法2通过构造函数实现线程间数据交流

```
class position{
        int p_x;          int th_y;
        position(int p_x,int th_y){
             this.p_x=p_x;        this.th_y=th_y;
        }
        void show(){
           System.out.println("police at "+p_x);
           System.out.println("thief at "+th_y);
        }
       boolean isCatched(){
            if(p_x>=-10&&p_x<=10&& th_y>=-10&&th_y<=10)
                return true;
           else    return false;
       }
}
```

```
class Policeman  extends   Thread{
      position pos;
      Policeman(String s, position   pos){
          super(s);  this.pos=pos;  }
      public void run(){
          try {   this.sleep(100);  }catch (Exception e){}
          while(true){
             for(int x=-100;x<=100;x++){
                pos.p_x=x;              pos.show();
                if(pos.isCatched()){
                   System.out.println("police at  "+pos.p_x
                  +"have catched"+"the thief who is at"+pos.th_y);
                   System.exit(0);  }
         }}
}
```

```
class Thief extends Thread{
      position pos;
      Thief(String s,position pos){
          super(s);this.pos=pos;}
      public void run(){
        try {  this.sleep(1); }catch(Exception e){}
        while(true){
            for (int y=-100;y<=100;y++){
               pos.th_y=y;
               pos.show( );
               if(pos.isCatched()){   System.exit(0);  }
            }
        }
}}
```

```
class testCatchThief2 {
    public static void main(String args[]){
      testCatchThief2  x=new testCatchThief2();
      x.go(); }
    public  void go(){
        position startP=new position(20,50);
           Policeman Pth=new Policeman("TomP",startP);
        Thief  Tth=new Thief("hijackT",startP);
        Tth.start();  Pth.start();
    }
}
```

thief at 15
thief at 15
police at 90
police at 91
thief at 16
thief at 16
police at 91
police at 92
…… ……
thief at 4
police at -12
thief at 4
police at -11
thief at 4
police at -10
thief at 4
police at -10have catchedthe thief who is at4

- 4多线程同步——数据的完整性
- void show(){
- System.out.println("police at "+p_x);
- System.out.println("thief at "+th_y);
- }

- 分析上例是否是应该输出
- 一行警察位置然后肯定会
- 输出下一行小偷的位置？

thief at 15
thief at 15
police at 90
police at 91
thief at 16
thief at 16
police at 91
police at 92
…… ……
thief at 4
police at -12
thief at 4
police at -11
thief at 4
police at -10
thief at 4

- 从上面程序小偷和警察的打印位置并不是交错输出而是出现这样情形，Why?

- 原因:
- Pth和Tth两个线程共享了同一个对象startP;
- Pth线程和Tth线程可能同时在使用startP.show()方法


```
void show(){
    System.out.println("police at "+p_x);
    System.out.println("thief at "+th_y);}
```

这时Tth线程也调用对象startP.show()方法

```
void show(){
    System.out.println("police at "+p_x);
    System.out.println("thief at "+th_y);}
```

- 如何解决呢?

- 关键字**synchronized**实现同步。

当对一个对象(含方法)使用**synchronized**，这个对象便被锁定。在一个时刻只能有一个线程可以访问被锁定的对象。它访问结束时，让高优先级并处于就绪状态的线程，继续访问被锁定的对象，从而实现资源同步。

- 方法(1)锁定对象testCatchThief3.java
- 语法：
- synchronized (startP){
- startP.show(); }

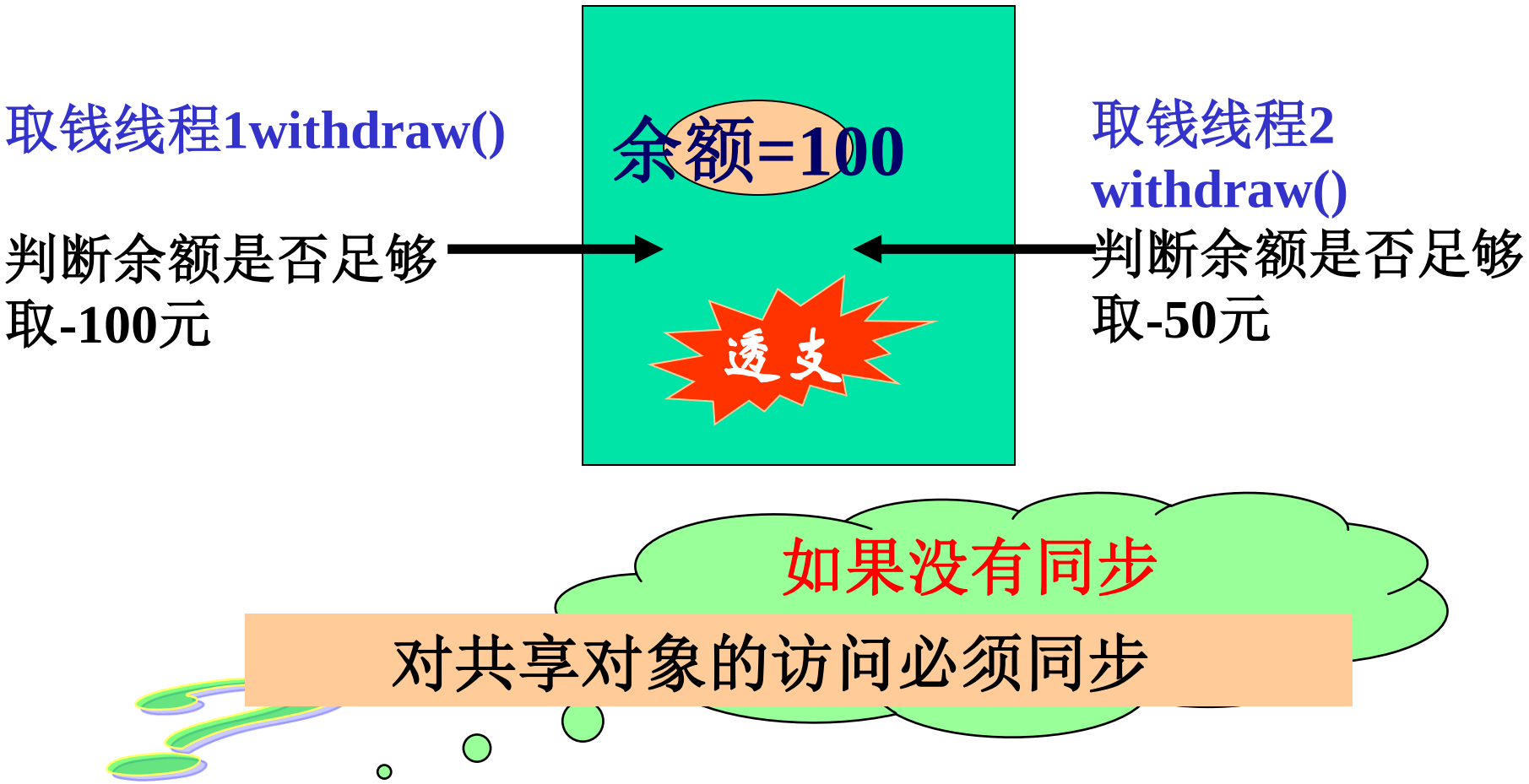
Java中每个对象都一个标志

Synchronized(startP)作用：
(1)检查startP的标志是否存在如果存在将标志取走执行{startP.show();}的内容直到执行完将标志交还给对象startP
(2)如果startP的标志不存在使用startP对象的线程会暂停等待直到其他线程将标志还给startP

- 方法（2）锁定一个方法
- 语法：
- synchronized　void show(){
- 　　System.out.println("police at "+p_x);
- 　　System.out.println("thief at "+th_y);
- }

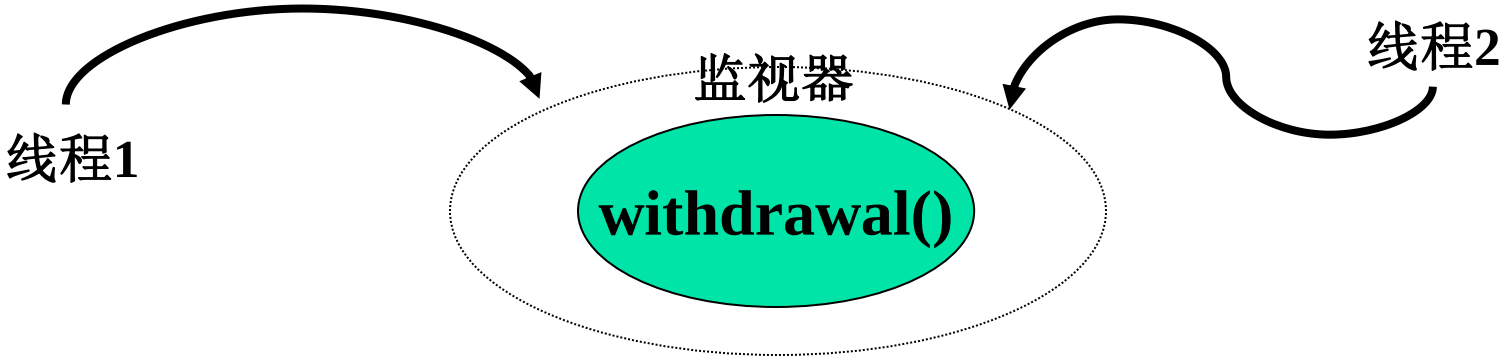
作用:任何时候只能有一个线程调用show()方法,其他线程如果试图调用必须等待

多线程同步举例


取钱线程1withdraw()
判断余额是否足够
取-100元
余额=100
透支
取钱线程2
withdraw()
判断余额是否足够
取-50元
如果没有同步
对共享对象的访问必须同步

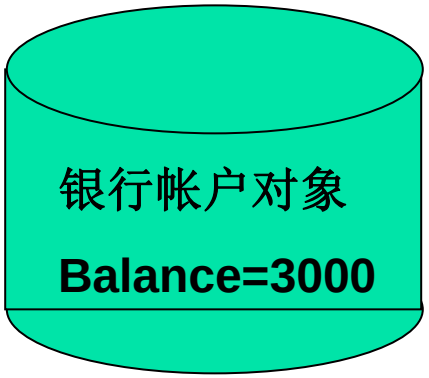
- 关键字synchronized（锁定）实现同步

- 线程1进入withdrawal方法时,线程1 withdraw()方法上(加锁);当线程1的方法执行完毕返回时,线程2方能使用withdraw方法.



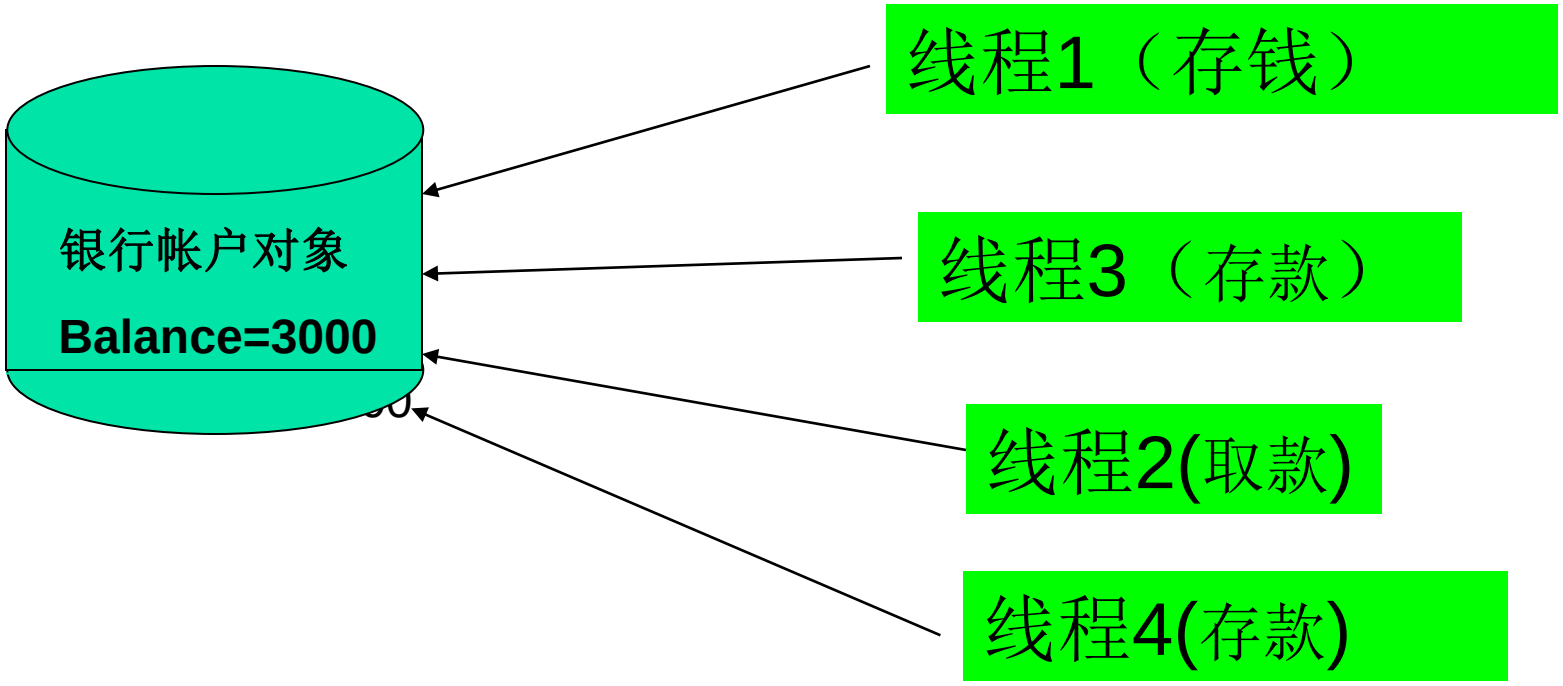
- 锁的作用：阻止两个线程同时访问同一个对象或方法。

- 实现银行帐户多线程的存款和取款
- 分析： **Accout** 类
- **People**线程类 实现取钱



线程同步的另外一个例子

- 实现银行帐户多线程的存款和取款
- 分析： **Accout** 类
- **People**线程类　能够实现取钱　和 存钱 以及查询



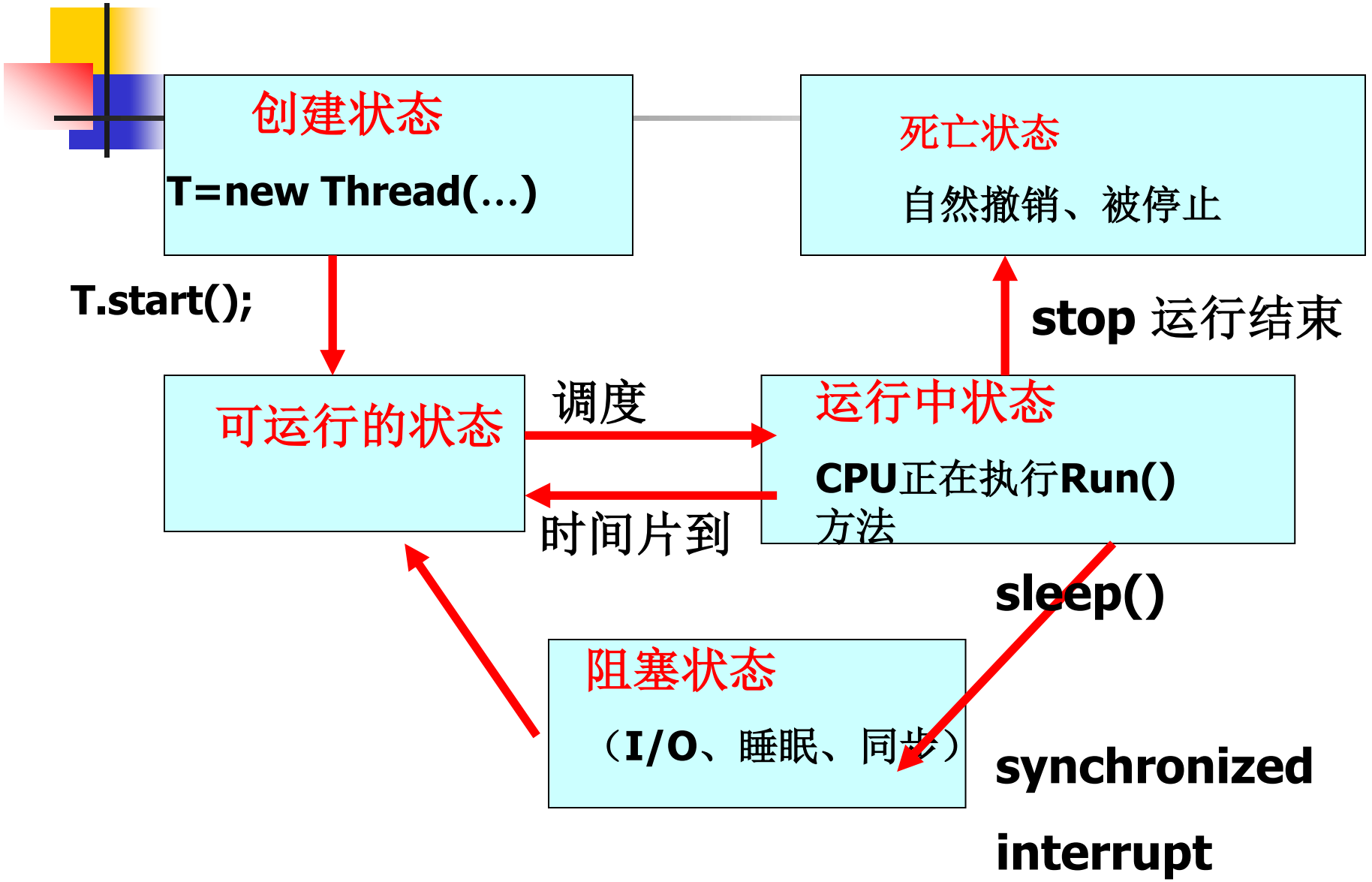
```
class Account{
    static int balance=1000;        static int expense=0;
 void withdrawl(int amount) {
            System.out.println("before withdraw");
            howmuch();
            System.out.println("you have withdraw "+amount);
           try{     Thread.sleep(200);          }catch(Exception e){}
            if (amount<=balance){
               try{        Thread.sleep(200);    }catch(Exception e){}
               balance-=amount;              expense+=amount;
               howmuch();
            }
            else {      System.out.println("it's not  enough ");}
   }
void inputMoney(int amount){
          System.out.println("before inpput");
           howmuch();
           System.out.println("you have input "+amount);
           balance+=amount;          howmuch();

}
  void howmuch() {   System.out.println("there is :"+balance); }
}
```

```
class People extends Thread{
        String name ; int money; Account  a;
        int    flag;//  1取钱   2存钱     3 查询余额
People(String name, int money, Account  a,int flag)
{
        this. name=name ;
        this.money=money;
        this.a= a; this.flag=flag;
}
public void run(){
    try{  sleep(200);    }catch(Exception e){  }
    if (flag==1)
        synchronized (a){a.withdrawl(money);}
    else if(flag==2)
        synchronized (a) {a.inputMoney(money);}
    else if (flag==3)
        synchronized (a){a.howmuch();}}

}
```

```
class testBankCard {

  public static void main(String args[]){
      Account  account1=new Account();
      People  Rose=new
              People("Rose",500,account1,1);
      People  Tom=new
                People("Tom",600,account1,2);
      Rose.run();
      Tom.run();
  }
}
```

运行结果是什么？

5线程的控制与调度


创建状态
T=new Thread(…)
死亡状态
自然撤销、被停止
T.start();
stop 运行结束
可运行的状态
调度
运行中状态
CPU正在执行Run()方法
时间片到
sleep()
阻塞状态
（I/O、睡眠、同步）
synchronized
interrupt

java.lang.Thread类

- 构造函数
- **(1) Public Thread(　)**
- **(2) Public Thread(Runnable　)**
- **(3) Public Thread(Runnable , String)**

例：

(1) Thread th1=new Thread();

(2) Policeman p=new Policeman();

Thread Pth=new Thread(P , "TomP");

- 常见的方法

start( )： 调用该方法的使线程处于可运行状态。

run( )： 该方法由**start( )**方法自动调用。

stop( )： 使线程退出可执行状态,进入死亡状态。

isAlive( )： 判断线程是否处于执行的状态，返回值**true**表示处于运行状态，**false**表示已停止。

常见方法1

static currentThread( )：返回当前运行的线程对象。

- setName(String s)：赋予线程一个名字。
- getName( )：获得调用线程的名字。
- getPriority( )：获得调用线程的优先级。
- setPriority(int p)：设置线程的优先级。

```
class testThread {
    public static void main(String args[]) {
            String name;
            int p;
            Thread curr=Thread.currentThread( );
            System.out.println("当前线程: "+curr);
            name=curr.getName( );
            p=curr.getPriority( );
            System.out.println("线程名: "+name);
            System.out.println("优先级 :"+p);
    }
}
```

```
当前线程: Thread[main,5,main]
线程名: main
优先级 :5
```

- 线程的优先级（1-10级）
- Thread类的三个静态常量
- final int MAX_PRIORITY： 最大优先级，值是10。
- final int MIN_PRIORITY： 最小优先级，值是1。
- final int NORM_PRIORITY：缺省优先级，值是5。

- 与优先级相关的两个方法：
- setPriority()
- getPriority()

- 说明：处于可运行状态的线程按优先级别排队等待

- 举例：
- public class testPriority{
- 　　public static void main(String args[]){
- 　　　　xx　ta= new　xx();
- 　　　　yy　tb= new　yy();
- 　　　　//tb.setPriority(1);
- 　　　　ta.start();
- 　　　　tb.start();
- }}

- 有tb.setPriority(1);这行的效果和没有结果相同吗?why

```
class    yy    extends Thread{
      public void run(){
      while(true){System.out.println("bbb"); }
      }
}

class    xx    extends Thread{
      public void run(){
      while(true){System.out.println("aaa");}
       }
}
```

■改变线程状态

- **Thread** 类里定义了一些用于改变线程的状态的方法。
- 休眠(Sleep)
- 静态方法：sleep(int n)

使当前运行的线程睡n个毫秒，然后继续执行。

(1)高优先级进程主动sleep让出一些CPU时间给低优先级的线程
(2)该方法也可实现延时效果

举例：上面的例子进行修改

```
class   xx    implements   Runnable{
        public void run(){
            while(true){
                System.out.println("aaa");
                try{   Thread.sleep(500);}
                catch(InterruptedException e
            }
    }
}
```

```
bbb
bbb
aaa
bbb
bbb
bbb
bbb
bbb
bbb
bbb
aaa
bbb
bbb
```

举例

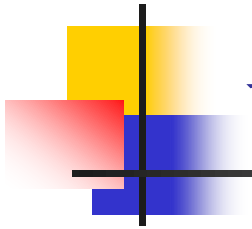
```
public  class testSleep extends Thread{
    public void run(){
      try{
         sleep(60000);//6秒
         System.out.println("sleep over");
       }catch(InterruptedException e){
         System.out.println("sleep is interrupted!");
       }
       System.out.println("end");
    }
    public static void main(String args[])throws Exception {
          testSleep  s=new testSleep();
          s.start();     Thread.sleep(10);
          s.interrupt();
  } }
```

```
sleep is interrupted!
end
```

- **Thread.yield( )静态方法**

- 作用：如果当前有相同优先级的其他线程处于就绪状态，那么**yield()**方法暂停正在运行线程（使它仍处于可执行状态），切换到其他同优先级的线程运行；若没有同优先级的可运行线程，则**yield()**方法什么都不做。

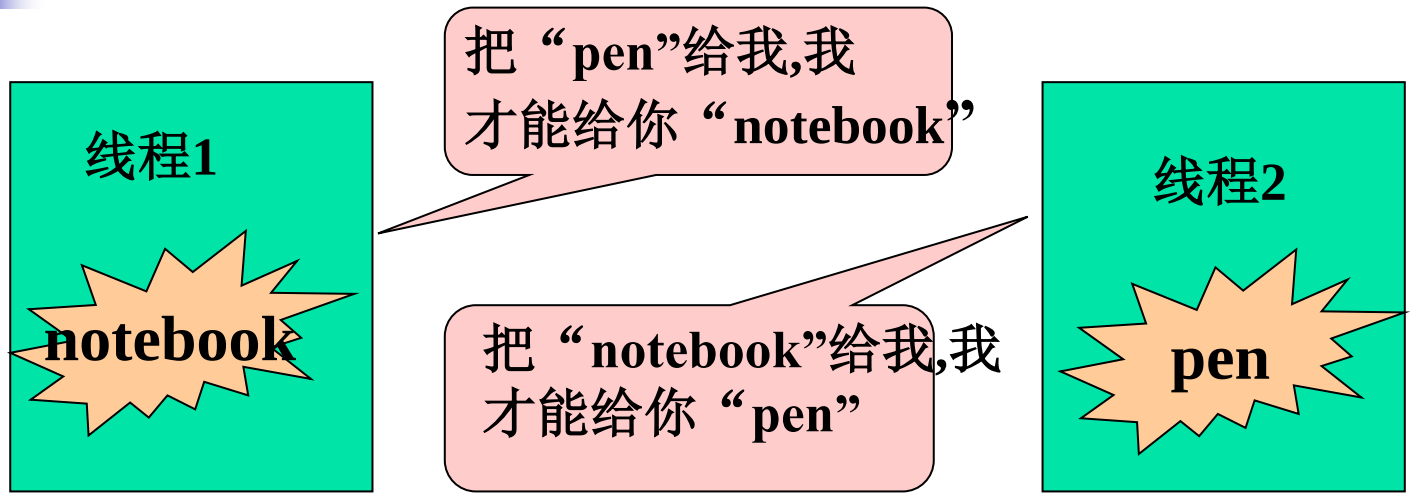
```
class MTh extends Thread{
    private static StringBuffer   str=new StringBuffer();
    private  static int count=0;
    public void run(){
      for (int a=0;a<20;a++){
          str.append(currentThread().getName() +":"+a+"  ");
         if(++count%10==0) str.append("\n");
         yield();
      }    }
    public static void main(String args[])throws Exception {
    MTh  m1=new MTh();     MTh  m2=new MTh();
    m1.setName("m1");     m2.setName("m2");
    m1.start();     m2.start();
    while(m1.isAlive()||m2.isAlive())
    {   Thread.sleep(500);  //主线程睡眠，等待m1,m2运行结束
       System.out.println(str);
    }
```



一次的运行结果

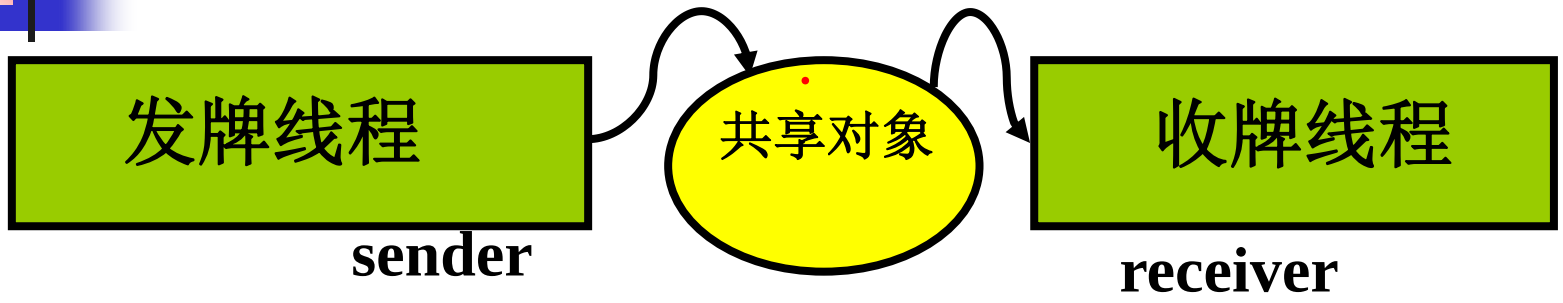
m1:0 m1:1 m1:2 m1:3 m1:4 m1:5 m1:6 m1:7 m1:8 m1:9
m1:10 m1:11 m1:12 m1:13 m2:0 m1:14 m2:1 m1:15 m2:2 m
m2:3 m1:17 m2:4 m1:18 m2:5 m1:19 m2:6 m2:7 m2:8 m2:9
m2:10 m2:11 m2:12 m2:13 m2:14 m2:15 m2:16 m2:17 m2:18
m2:19

6. 死锁问题



定义：多个处于等待状态的线程占用着系统的资源不放，又再等待无法获得的其他资源——**死锁**

7多线程问题---等待同步数据



可能出现的问题:

（1）发牌者比收牌者快时,收牌者会漏掉一些数据没有取到

（2）收牌者比发牌者快时,收牌者多次收取相同的数据.解决方法:

- notify()和wait ()方法用来协调读取的关系.

使用wait()和notify()方法实现同步数据

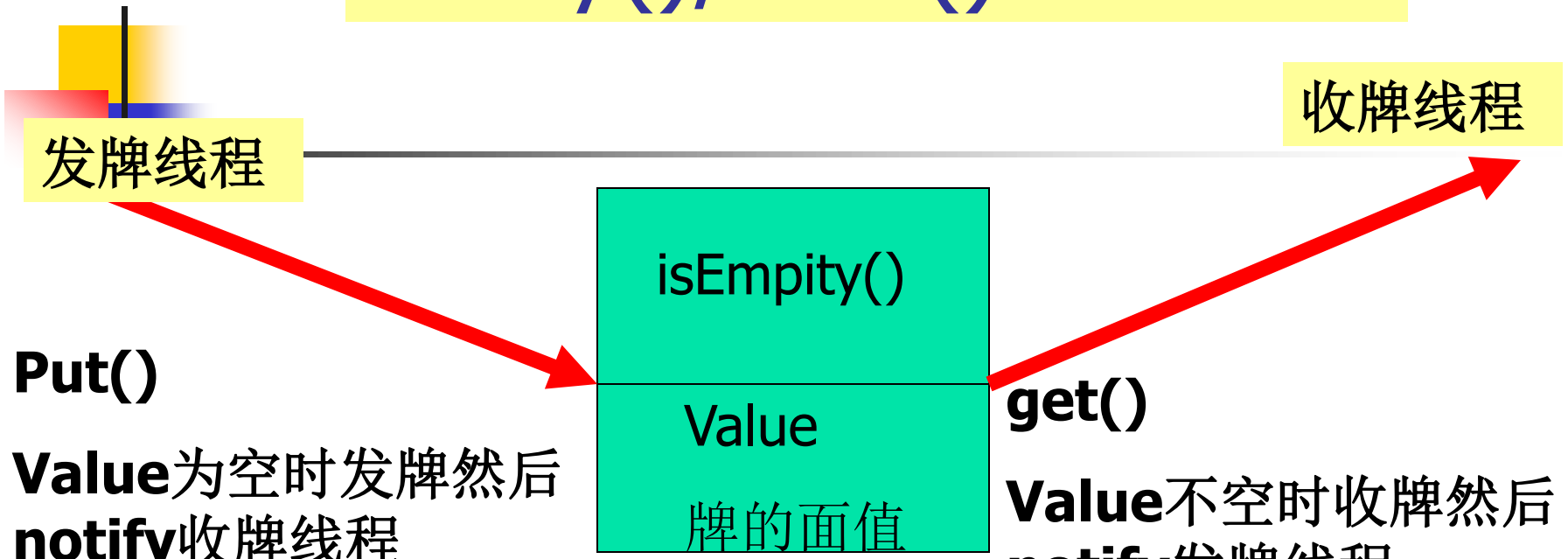
wait( )方法

作用：使当前线程进入等待状态，*主动释放*当前线程锁定的任何对象。

notify( )方法

作用：唤醒等待队列中的其它线程，使其处于可运行状态。

notify()/wait()）



Put()

Value为空时发牌然后**notify**收牌线程

为不空**wait();**

get()

Value不空时收牌然后**notify**发牌线程

为空时 **wait();**

发牌缓冲区类 **buffer**

发牌线程类 **sender**

取牌线程类 **receiver**

```
class Buffer {    //加互斥锁的缓冲区
    int value;    boolean isEmpty = true;
    synchronized  void  put(int i) {
        while (!isEmpty){
            try{ this.wait(); }catch(Exception e){}
        }
    value = i; isEmpty = false;notify(); //唤醒正在等待线程
    }
//取牌方法get()如何写呢？
    synchronized  int  get() {
        while (isEmpty){
            try{this.wait();}catch(Exception e) {}
        }
        isEmpty = true;  notify();   return value;
    }
}
```

```
class  Sender extends Thread {    //发牌线程
        private Buffer bf;
        public Sender(Buffer  bf) {this.bf = bf; }
        public void run( ) {
            for (int i=1;i<6;i++){
                bf.put( i );
                System.out.println("Sender  put : " + i ); }
        }
}
```

```
class  Receiver extends Thread {    //发送线程
    private Buffer bf;
    public Receiver(Buffer  bf) {this.bf = bf; }
        public void run() {
          while(true){
            System.out.println("Receiver  get : " + bf.get( ) );  }
          }
}

public class card {
    public static void main (String args[]) {
        Buffer  bf = new Buffer();
        (new Sender(bf)). start();
        (new Receiver(bf)). start();
    }
}
```

线程举例：生产者消费者程序

分析：

类Storehouse

生产者线程

消费者线程

```
class Storehouse{
  int index=0;    char[]    store;
  public Storehouse(char[] store){
    this.index=store.length;  this.store=store; }
  public synchronized  void push(char x){
    store[index]=x;      index++;  }
  public synchronized  char pop(){
    index--;    return store[index];      }
  public synchronized  void show(){
      for(int i=0;i<index;i++)
        System.out.print(store[i]+"  ");
     }
}
```

```
class Producer extends Thread{
    Storehouse s1;String name ;
    Producer(String name,Storehouse s1){
        super(name);
        this.s1=s1; }
    public void run(){
      while(true){
        char c=(char)('A');      s1.push(c);
        System.out.println(c+"<-");
        s1.show();
         /*    try{   Thread.sleep(2000);
                }catch(Exception e){}*/
       }
    }
}
```

```
class Consumer extends Thread{
    Storehouse s1;String name ;
    Consumer(String name,Storehouse s1){
        super(name);
        this.s1=s1; }
    public void run(){
      while(true){
        char c=s1.pop();
        System.out.println(c+"->");
        s1.show();
        try{   Thread.sleep(20000);
        }catch(Exception e){}
      }
}}
```

```
public class ProducerConsumer{
    public static void main(String args[]){
        char[] x={'a','b','c'};
        Storehouse s=new Storehouse(x);
        Producer  p1=new Producer("John",s);
        Consumer  c1=new  Consumer("Linda",s);
        p1.start();
        c1.start();
}}
```

产生问题：经常产生异常(堆栈的溢出)
原因：生产者生产太快或消费者消费太快造成

运行结果

- a a b->
- B B<-
- a B->
- a->
- F F<-
- F->
- Exception in thread "Linda" java.lang.ArrayIndexOutOfBoundsException: -1
- at Storehouse.pop(ProducerConsumer.java:9)
- at Consumer.run(ProducerConsumer.java:43)
- Exception in thread "John" java.lang.ArrayIndexOutOfBoundsException: -1
- at Storehouse.push(ProducerConsumer.java:7)
- at Producer.run(ProducerConsumer.java:24)

- **解决方法：**
- 在pop方法中
- 先判断一下store是否为空，如果空则正在执行pop方法的线程wait()，通知生产者线程生产
- 在push方法中
- 先判断一下store是否满如果满则正在执行push方法的线程wait()，通知消费线程消费

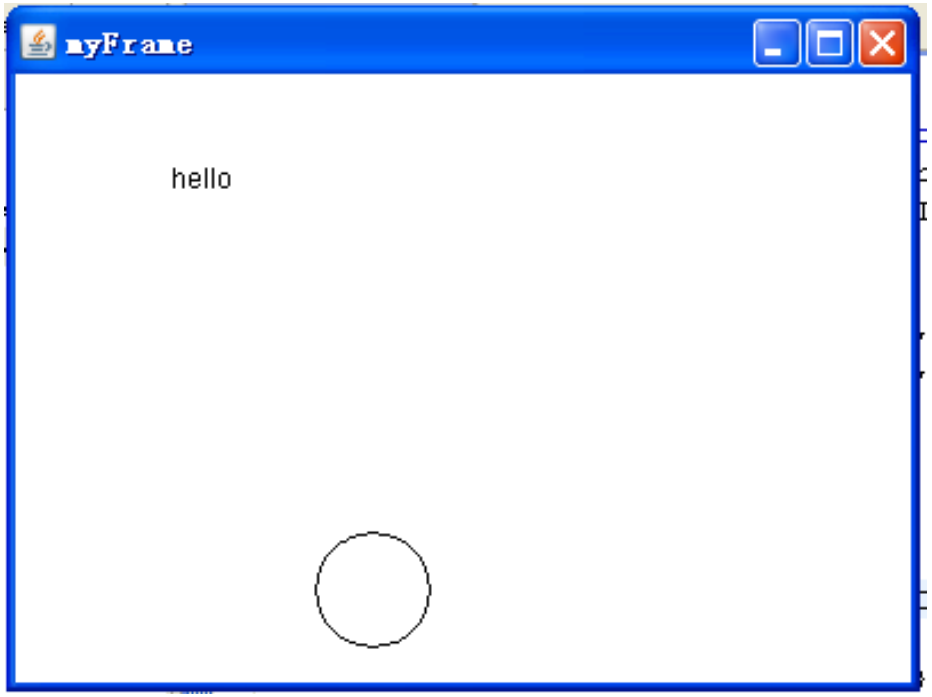
- **public synchronized void push(char x){**
-
```
while(index==store.length){
    try{this.wait();
    }catch(Exception e){}
}
 notify();
  store[index]=x;    index++;
```
- **}**
-

```
public synchronized  char pop(){
    while (index==0 ){
        try{  this.wait();
          }catch(Exception e){}
    }
    notify();
    index--;
    return store[index];
  }
```

- a b c d->
- a b c D D<-
- a b c D->
- a b c I I<-
- a b c I->
- a b c->
- a b->
- a I I<-
- a I->

a C C<-
a C->
a->
store empty
I I<-
I->
store empty
I I<-
I->
store empty
G G->
G<-
store empty
G G<-
G->
store empty
F F<-
F->
store empty
I->
I<-
store empty
D D<-
D->

- 8线程举例
- （1）在窗口上下两部分分别运行两个线程,上面的区域中显示由右向左游动的字符串**hello**,下面的区域显示从左向右滚动的圆环.



```
class ThreadMove  extends Thread{
      private int x,y,  step,  sleepTime;
      private int leftBorder,rightBorder;
      public ThreadMove(int x,int y,int step,int
            leftBorder,int rightBorder,int sleepTime) {
          this.x=x;   this.y=y;
          this.step=step; this.sleepTime=sleepTime;
          this.leftBorder=leftBorder;
          this.rightBorder=rightBorder;    }
      public void run(){
           for (;;) { x+=step;
            if (x<leftBorder) x=rightBorder;
            if (x>rightBorder) x=leftBorder;
               try{ sleep(sleepTime);}catch(Exception e){}
           }
       }
      public int getX() {   return this.x;     }
      public int getY() {   return this.y;     }
   }
```

```
public class TreadDraw extends Panel {
    public ThreadMove t1;ThreadMove t2;
    public static void main(String args[]){
        Frame f;
        TreadDraw  x=new TreadDraw();
        f=new Frame("myFrame");
        f.add(x,"Center");
        x.t1=new ThreadMove(0,50,1,0,200,100);
        x.t2=new ThreadMove(200,200,-1,0,300,50);
        x.t1.start();              x.t2.start();
        f.setSize(400,300);
        f.setVisible(true);
```

```
while (true) {
        try{ Thread.sleep(10);
             x.repaint();
        }catch(Exception e){}
}
}
public void paint(Graphics g){
        g.drawString("hello",t1.getX(),t1.getY())  ;
        g.drawOval(t2.getX(),t2.getY(),50,50);
}
}
```

Join（）方法

- 作用：暂停当前进程，转去运行另外一个进程（调用**join()**方法的进程），直到该进程运行结束或被中断才恢复上一个进程的运行

```
class testJoin{
    public static void main(String args[]){
        xx  a=new xx();
        yy b=new yy();
        a.start();
        try{ a.join();
        }catch(Exception e){}
        b.start();  }
}
```

```
aaa
aaa
aaa
aaa
aaa
aaa
……
aaa
aaa
aaa
```

说明：当前进程：**main()**

调用**join()**方法的进程: **a**

而**a**进程是个无穷循环，永远法停止，因此**main**无法继续运行**b.start()**就永远运行不到