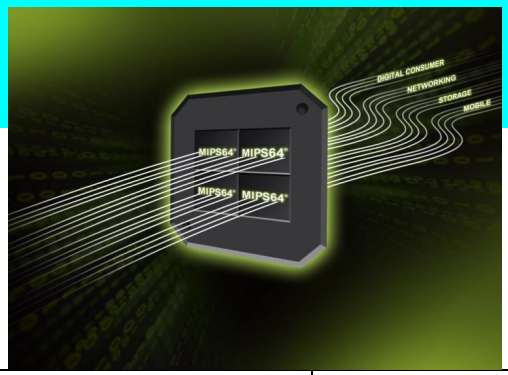
Chapter 8

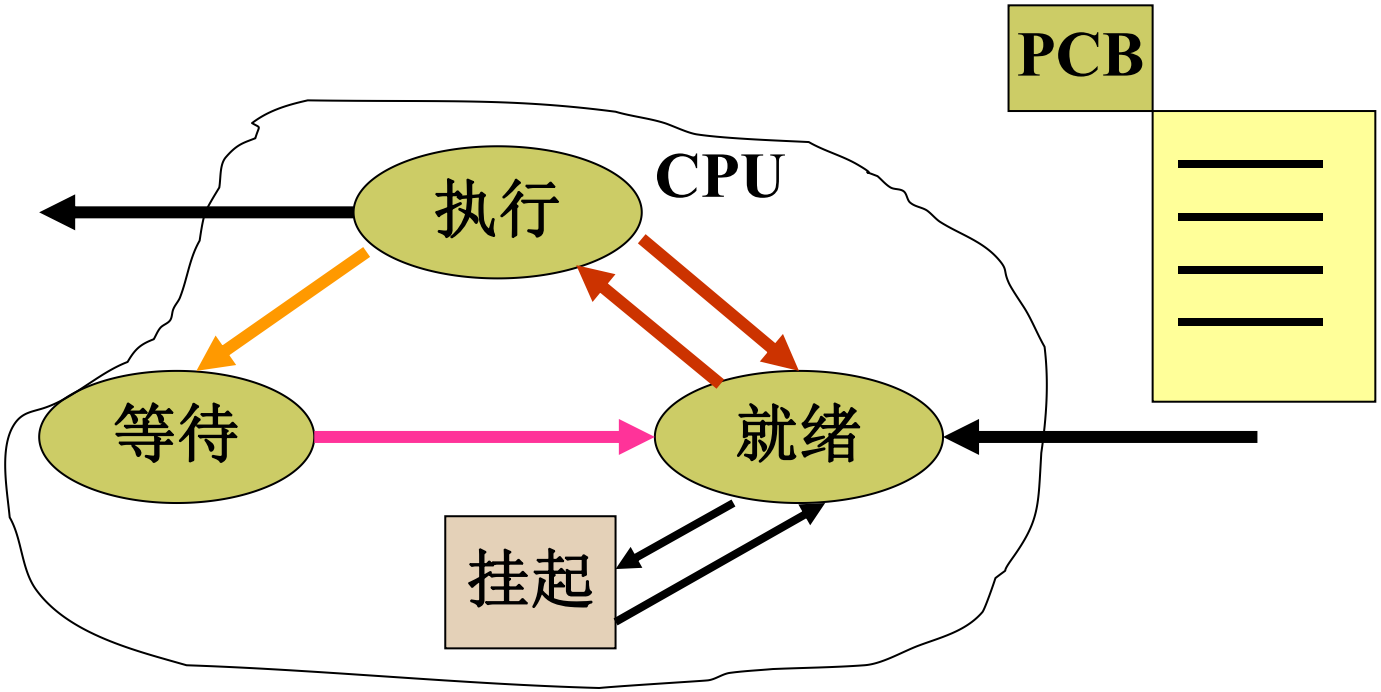
Java的多线程



1. 线程的概念
2. 线程的生命周期
3. **创建一个线程**
4. 继承Thread类与使用Runnable接口
5. 线程的同步与死锁

1. 程序—进程—线程

- 程序(Program)是为完成特定任务、用某种语言编写的一组指令的集合。指一段静态的代码。
- 进程(Process)是程序的一次执行过程，是系统进行调度和资源分配的一个独立单位。

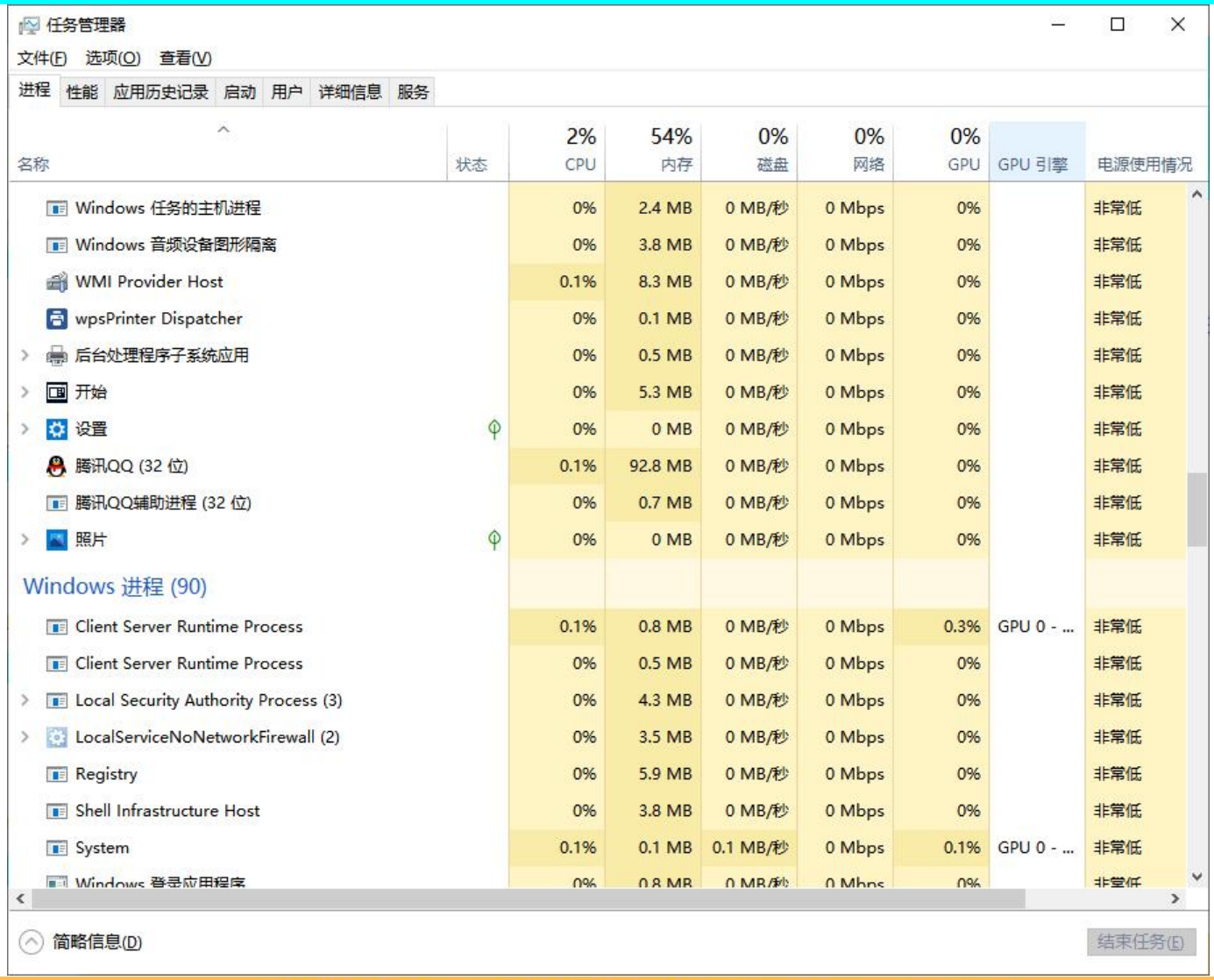


Windows 7 运行中的进程

Ctrl+Alt+Del

- 操作系统五大基本功能：
  - ✓ **处理器管理**
  - ✓ **存储器管理**
  - ✓ **设备管理**
  - ✓ **文件管理**
  - ✓ **作业管理**

Windows10任务管理器中的进程



进程

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

程序-进程-线程

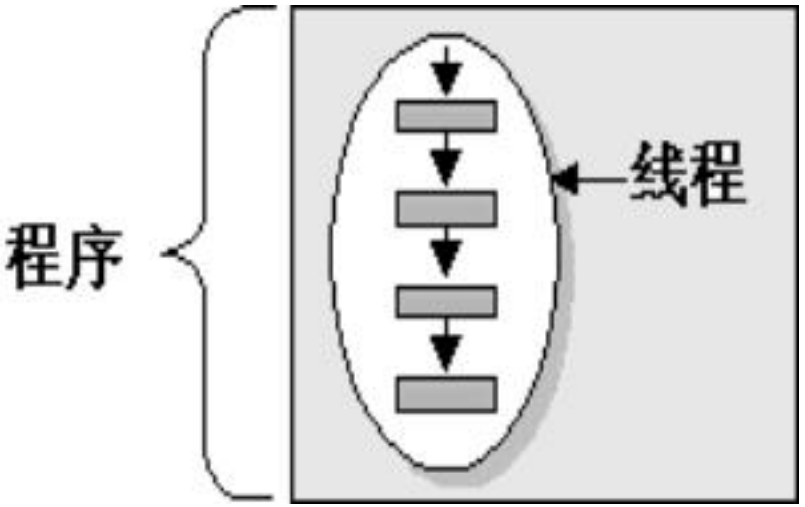
- 线程(Thread)是一种操作系统对象，它代表一个进程中代码被执行的路径。**一个进程可以包含多个线程。**

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）



建立线程例:

```
public class  MyThreadWebServer {
    public static void main(String args[]){
        try {
            ServerSocket ss=new ServerSocket(80);
            System.out.println("Web Server is OK");
            while (true) {
                Socket s=ss.accept();
                Process p=new Process(s);
                Thread  t=new Thread(p);
                t.start( );
            }
        } catch (Exception e) {System.out.println(e);}
    }
}
```

WEB服务器进程中：

接受一个访问，创建一个线程！

1.线程（Threads）的概念

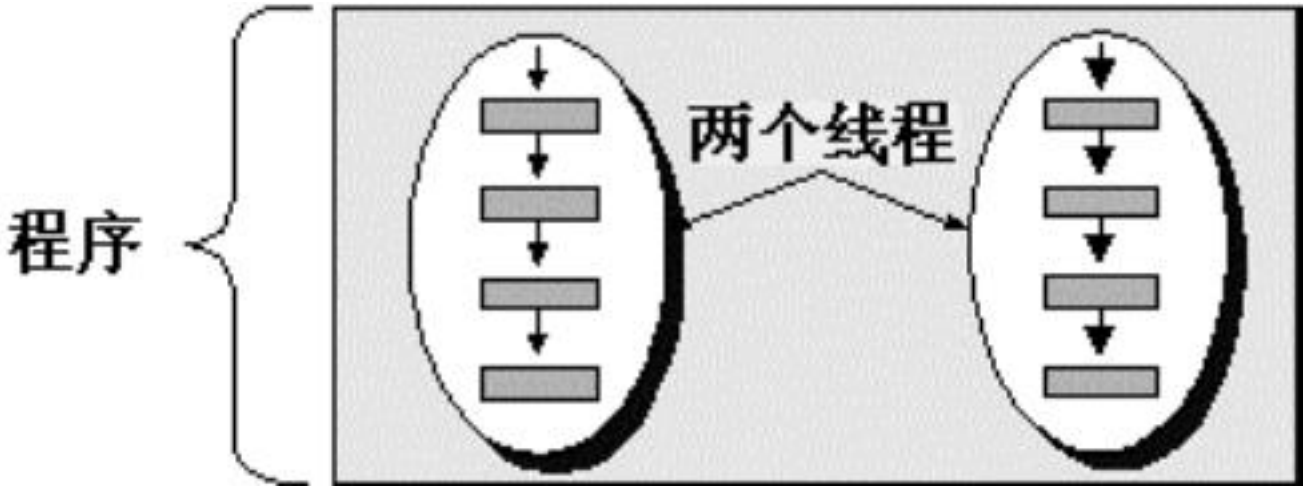
- 线程(Threads)是比进程更小一级的执行单元。
- 一个进程在其执行过程中，可以产生多个线程，形成多条执行线索。
- 每个线程也有它自身的产生、存在和消亡的过程，也是一个动态的概念。
- 一个线程有它自己的入口和出口，以及一个顺序执行的序列。
- 线程不能独立存在，必须存在于进程中，各线程间共享进程空间的数据。
- 线程—线程创建、销毁和切换的负荷远小于进程，又称为轻量级进程（lightweight process）。
  - ✓ 系统负担小，主要是CPU的分配。

基本概念之一：进程

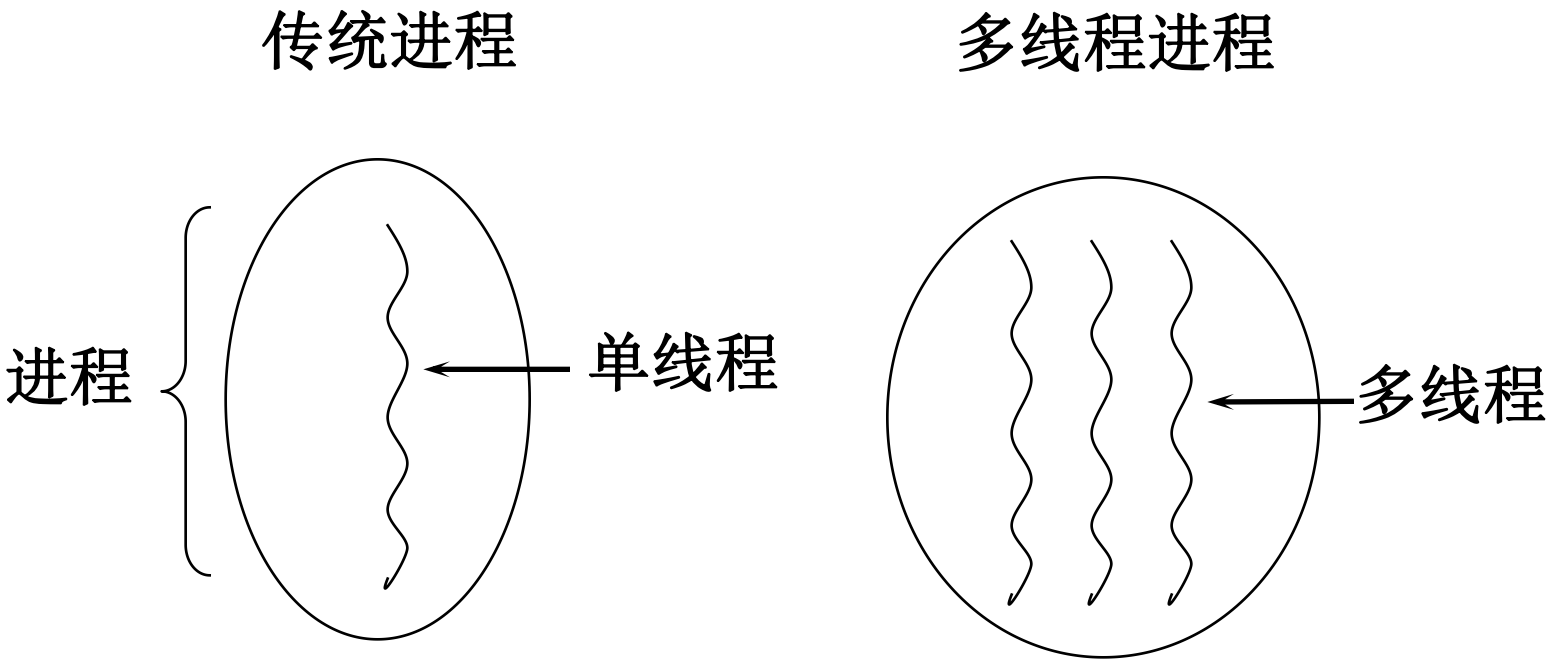
- 进程是正在运行的一个程序
- 程序：静态对象 ⟷ 进程：动态过程
- 操作系统为每个进程分配一段内存空间，
  包括：代码、数据以及堆栈等资源
- 多任务的操作系统（OS）中，进程切换对CPU资源消耗较大。

基本概念之二：多线程

- 多线程的技术则将多任务的概念的进一步扩展，它使单个程序内部也可以在同一时刻执行多个代码段，完成不同的任务，这种机制称为多线程。



进程与多线程



Concurrency—线程和进程

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。

优点

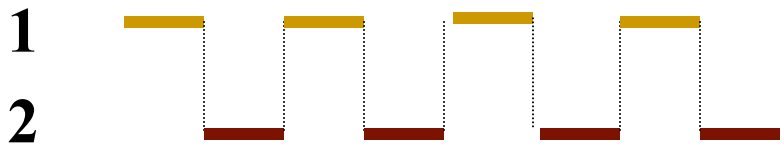
（1）线程的切换比进程切换的负担小，并发容易。

（2）由于多线程共存在于同一个内存块中通信非常容易。

tutorial/essential/concurrency/procthread.html

线程的调度

- 调度策略
  - 时间片



  - 抢占式：高优先级的线程抢占CPU
- Java的调度方法
  - 同优先级线程组成先进先出队列，使用时间片策略
  - 对高优先级，使用优先调度的抢占式策略

2. Java与多线程

- Java直接支持多线程，Java虚拟机的很多任务都依赖线程调度，而且所有的类库都是为多线程设计的。Java利用多线程实现了一个异步的执行环境。
- Java的线程是通过**Java.lang.Thread**类包来实现的。Thread类封装了对线程操作控制所必需的方法。在线程实例化对象定义了很多函数用来控制一个线程的行为。
- 每个Java程序都有一个隐含的主线程
  - ✓ application main() 方法
  - ✓ Applet小程序，主线程指挥浏览器加载并执行 Java 小程序。

JAVA实现线程的两种方式

1. extends（继承）java.lang.Thread 类

2. implements（实现） java.lang.Runnable接口

例如：

```
class PrimeThread extends Thread {
      }


class PrimeRun implements Runnable {
}
```

Defining and Starting a Thread

1、Subclass Thread.

An application can subclass Thread, providing its own implementation of run, as in the HelloThread example:

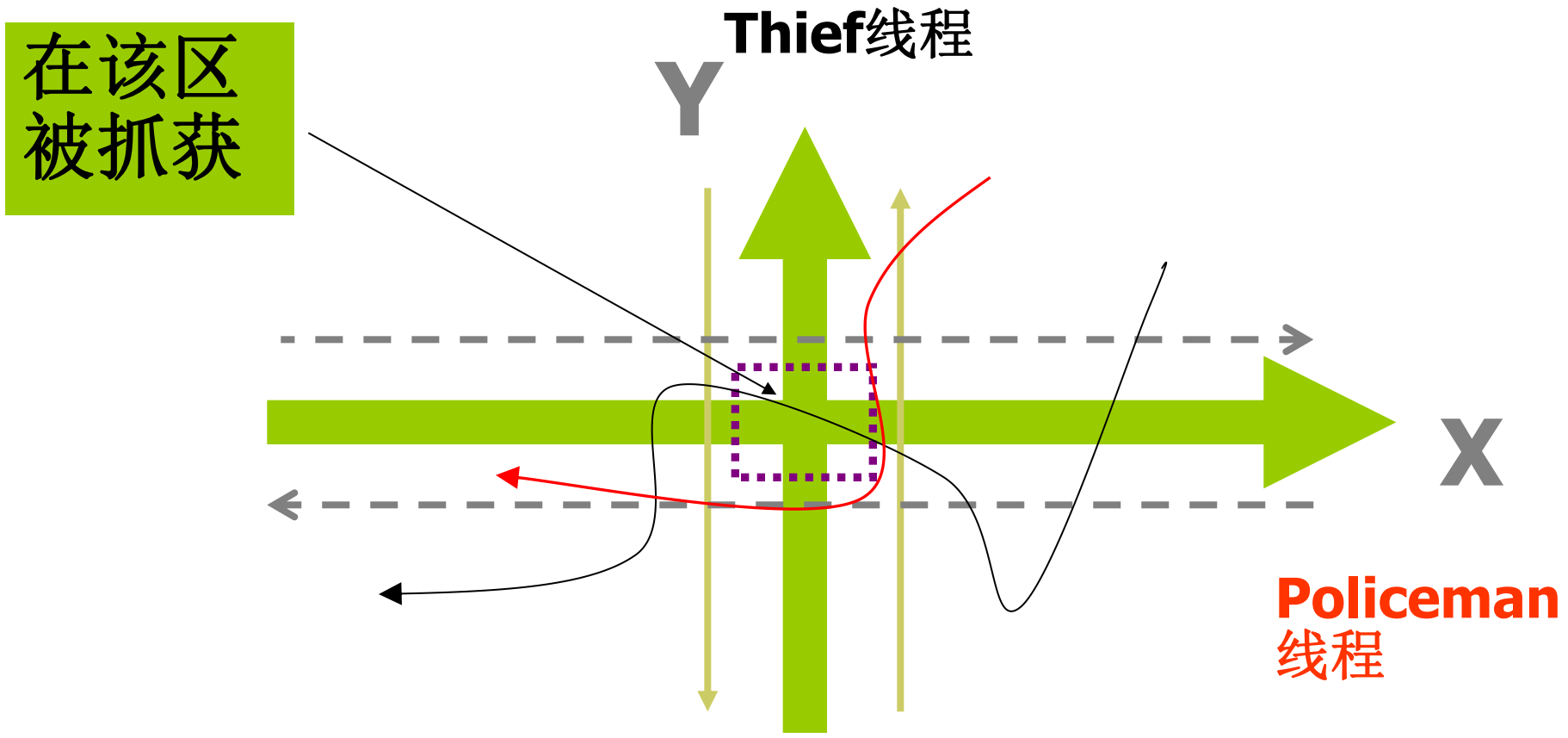
```
public class HelloThread extends Thread {
    public void run() {
        System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
        (new HelloThread()).start();
    }
}
```

Defining and Starting a Thread

2、The Runnable interface

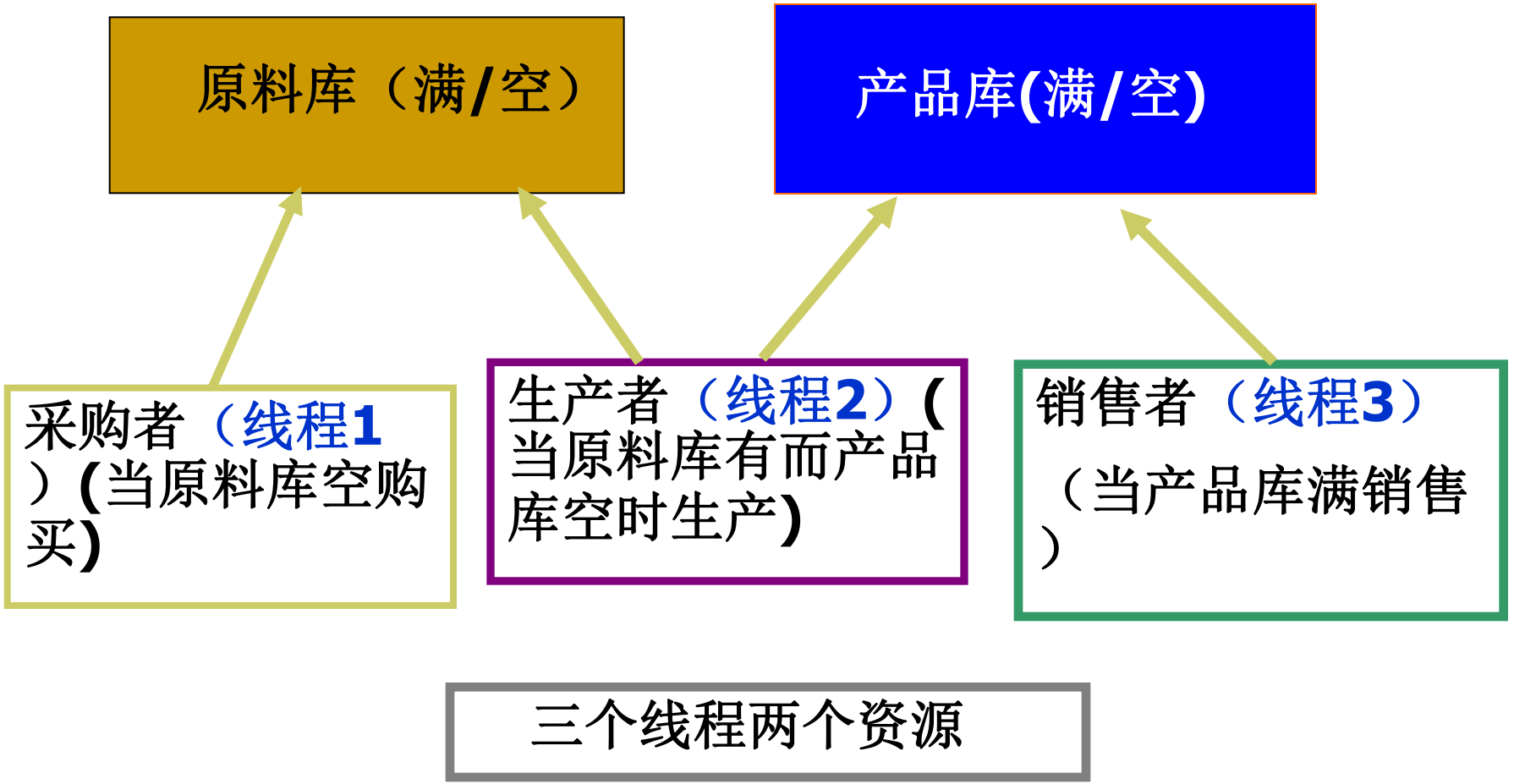
```
public class HelloRunnable implements Runnable {
    public void run() {
        System.out.println("Hello from a Runnable!");
    }
    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }
}
```

简单模拟多线程的例一



- 警察抓小偷例子

简单的模拟多线程的例二



线程的概念模型

在Java编程中，虚拟处理机封装在Thread类的一个实例里。构造线程时，定义其上下文的代码和数据是由传递给它的构造函数的对象指定的。

- 小例子：
  - SimpleThread.java
  - TwoThreadsTest.java

```
public class TwoThreadsTest {
    public static void main (String[] args) {
            new SimpleThread("Take it
myself??").start();
            new SimpleThread("Give it to
police!!").start();
    }
} //TwoThreadsTest.java
```

```
0 Take it myself??
0 Give it to police!!
1 Take it myself??
1 Give it to police!!
2 Take it myself??
2 Give it to police!!
……
8 Take it myself??
8 Give it to police!!
9 Give it to police!!
9 Take it myself??
DONE! Give it to
police!!
DONE! Take it
myself??
```

SimpleThread.java

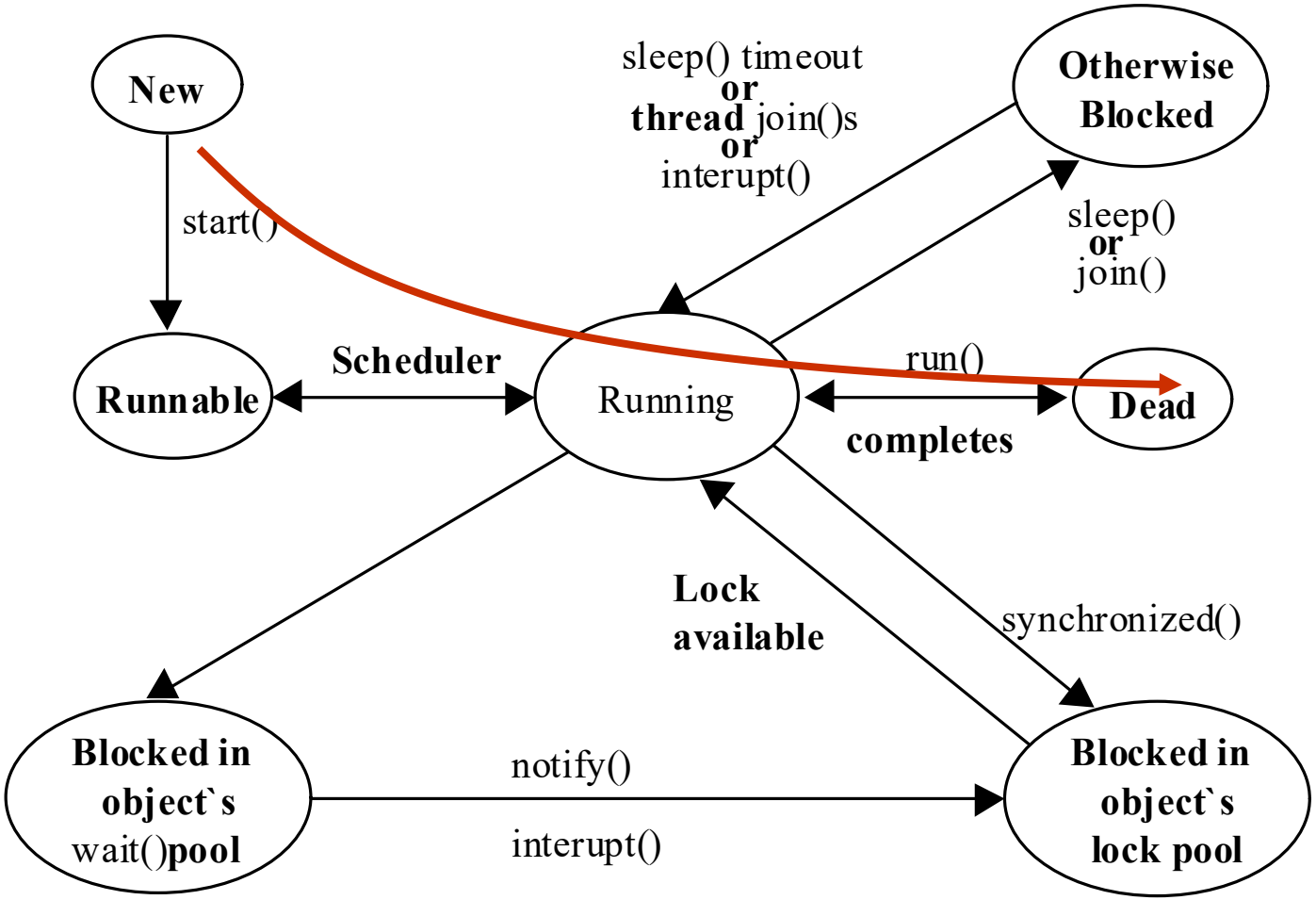
```
public class SimpleThread extends Thread {
    public SimpleThread(String str) {
        super(str); } //创建具以str为名字的线程
    public void run() {   //定义run()方法
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + getName());
            try {   //休眠一段时间
                sleep((int)(Math.random() * 1000));
            } catch (InterruptedException e) {}
        }
        System.out.println("DONE! " + getName());
    }
} // SimpleThread.java
```

3. 线程的生命周期

- 要想实现线程，必须在主线程中创建新的线程对象。Java语言使用Thread类及其子类的对象来表示线程，在它的一个完整的生命周期中通常要经历如下的五种状态：
  - ✓ 新建new： 当一个Thread类或其子类的对象被声明并创建时，新生的线程对象处于新建状态
  - ✓ 就绪start()：处于新建状态的线程被启动后，将进入线程队列等待CPU时间片，此时它已具备了运行的条件
  - ✓ 运行：当就绪的线程被调度并获得处理器资源时,便进入运行状态，run()方法定义了线程的操作和功能
  - ✓ 阻塞Blocked：在某种特殊情况下，被人为挂起或执行输入输出操作时，让出CPU并临时中止自己的执行，进入阻塞状态
  - ✓ 死亡：线程完成了它的全部工作或线程被提前强制性地中止 stop()或destroy()

3. 线程的生命周期

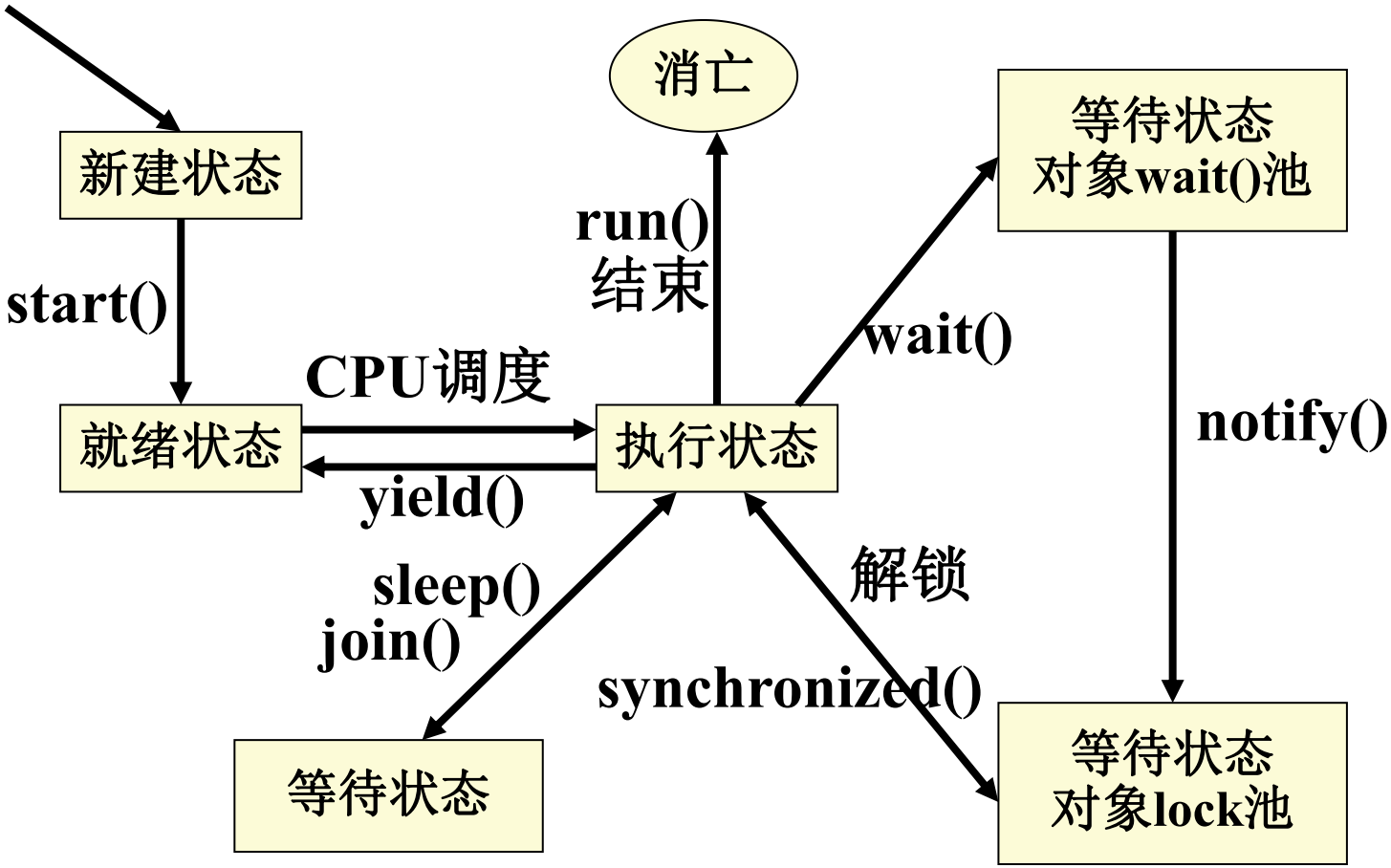
Thread states

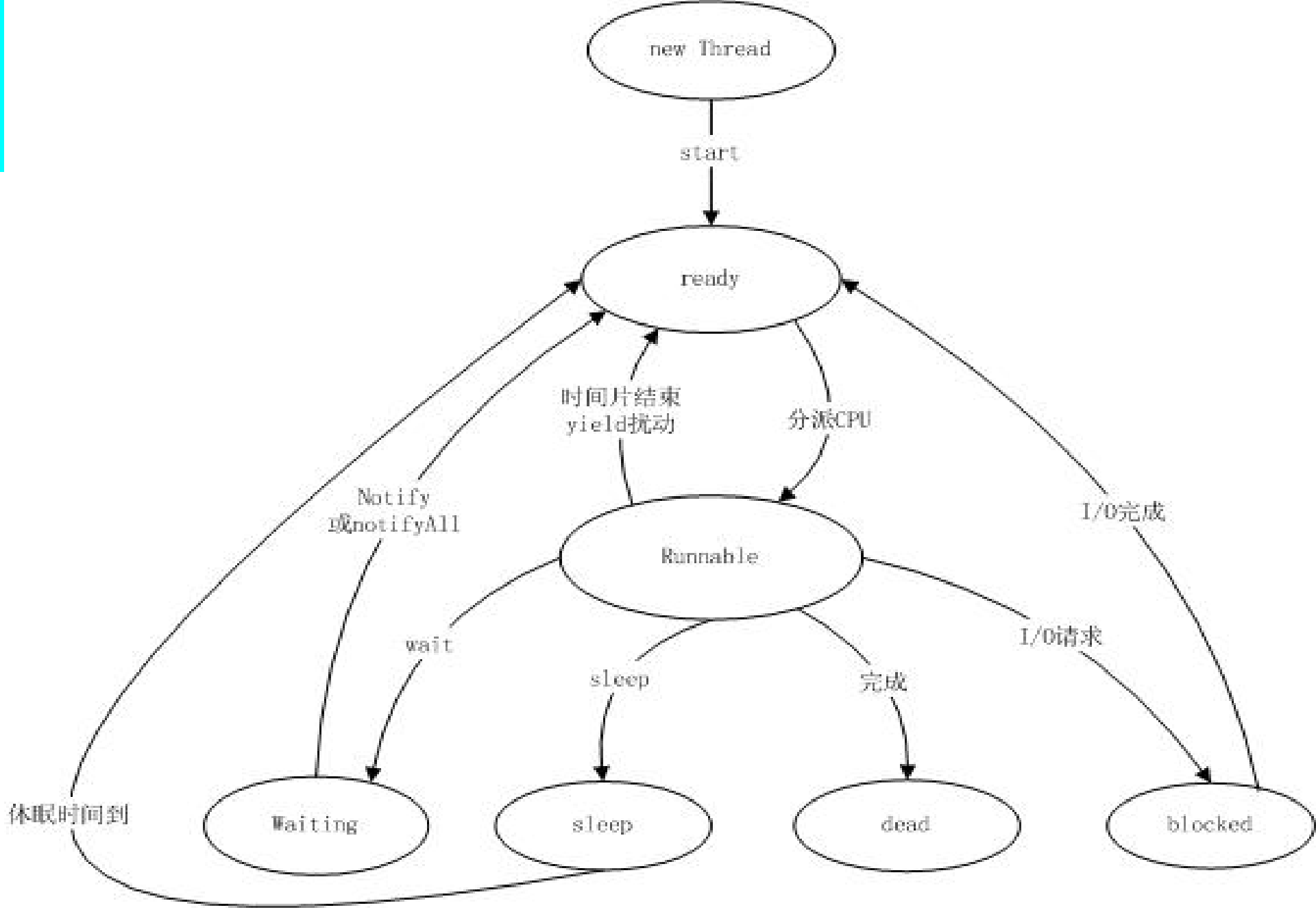


3. 线程的生命周期（续）

- New：线程已创建，但尚未执行
- Runnable：（就绪）线程已被调度，按优先级和先到先服务原则在队列中排队等待CPU时间片资源
- Runnnig：正在运行
- Blocked：（阻塞）因某事件或睡眠而被暂时性地挂起
- Dead：正常/强行中断，退出运行状态

线程状态


新建状态
start()
就绪状态
CPU调度
yield()
执行状态
run()
结束
消亡
wait()
等待状态
对象wait()池
notify()
sleep()
join()
等待状态
解锁
synchronized()
等待状态
对象lock池


new Thread
start
ready
时间片结束
yield扰动
分派CPU
Notify
或notifyAll
I/O完成
Runnable
wait
I/O请求
sleep
完成
休眠时间到
Waiting
sleep
dead
blocked

ThreadDemo1.java

```
public class ThreadDemo1 extends
      Thread {
private String threadName;
private long delaytime;

public ThreadDemo1(String thName,
      long time) {
    this.threadName =thName;
    this.delaytime =time;
    setDaemon(true);
  }

public void run(){
   try{
      while (true){
        System.out.println(threadName);
        sleep(delaytime);
      }
  }catch(InterruptedException e){
    e.printStackTrace() ;
  }
}
```

```
public static void main(String[]
      args){
  ThreadDemo1 thread1=new
      ThreadDemo1("Thread--1
      ",200);
  ThreadDemo1 threadDemo=new
      ThreadDemo1("Thread--2
      ",400);
  ThreadDemo1 thread3=new
      ThreadDemo1("Thread--3
      ",800);

thread1.start() ;
  threadDemo.start();
  thread3.start();
  try{
      System.in.read() ;
}catch(Exception e){
   e.printStackTrace() ;
}
  }
}
```

资料

- http://www.cnblogs.com/rollenholt/archive/2011/08/28/2156357.html (java多线程总结)
- http://www.jianshu.com/p/40d4c7aebd66 Java中的多线程你只要看这一篇就够了
- http://developer.51cto.com/art/200512/15883.htm （51CTOJava频道）

2021-5-13

4. 实现多线程的两种编程方法——继承*Thread*类与使用*Runnable*接口

提供对多线程的支持

- ✓ Thread类
  - start( ), stop( ), run( )
- ✓ Runnable 接口

实现多线程的两种编程方法

- ✓ 继承 Thread 类
- ✓ 实现 Runnable 接口

4. 实现： 继承Thread类与使用Runnable接口

- Thread类综合了Java程序中一个线程需要拥有的属性和方法
- 当生成一个Thread类的对象后，一个新的线程诞生了。
- 每个线程都是通过目标对象的方法run()来完成其操作的。
  - 方法**run()** 称为**线程体**(线程方法)。

- 提供线程体的目标对象是在初始化一个线程时指明的。
- 任何实现了Runnable接口(实现run()方法)的类实例都可以作为线程的目标对象。

Thread类

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(){…}

……

方法之一：继承Thread类

- Thread类的重要方法：run( )
  - 定义线程的具体操作
  - 系统调度此线程时自动执行
  - 初始时无具体操作内容
- 如何编程呢？

－继承Thread类，定义 run( ) 方法

例4-2：Run方法可以在Thread的子类中重构

```
public class Twothread extends Thread{
  public Twothread (String str){   super(str);    }
  public void run(){
       String [ ] Boy  = {"李玉刚","王小武","张子豪","徐国庆"};
       String [ ] Girl = {"王莉","张苹","赵白云","和平"};
       String Name = null;
       Name=this.getName();
       for (int i = 0; i < 4; i++){
           if (Name.equals("Boy: ")){
             System.out.println(Name+": "+Boy[i]+" look for a girl \n");
             try{        sleep((long)(Math.random() * 1000));
                    }catch (InterruptedException e){  }  }
            if (Name.equals("Girl: ")){
             System.out.println(Name+": "+Girl[i]+" look for a boy \n");
            try {        sleep(500);
                 }catch (InterruptedException e) {  }
    }   }   }
   public static void main(String[] args)  {
       new Twothread ("Boy: ").start();
       new Twothread ("Girl: ").start();     }
}
```

```
General Output
Boy: 线程: 李玉刚 look for a girl
Girl: 线程: 王莉 look for a boy
Girl: 线程: 张苹 look for a boy
Boy: 线程: 王小武 look for a girl
Girl: 线程: 赵云 look for a boy
Girl: 线程: 许倩 look for a boy
Boy: 线程: 张子豪 look for a girl
Boy: 线程: 徐国庆 look for a girl
```

分析

- 类Twothread是Thread类的子类，它首先调用了一个构造方法，其中参数为字符串类型，例如上面的“Girl: ”和“Boy: ”， 它的作用就是给线程取名。
- 类Twothread中的第二个方法是run()方法，它覆盖了类Thread中的run()方法。run()方法中是一个重复4次的循环，每次循环中显示一些信息,然后睡眠一个随机产生的时间间隔。系统在线程的run()函数里控制线程。一旦进入run()函数，便执行里面的任何程序。一旦run()执行完，这个线程也就结束了。
- sleep()函数只是简单地告诉线程休息多少个毫秒，其参数为休息时间。当这个时间过去后，线程即为可继续运行。当线程睡眠时sleep()并不占用系统资源。其它线程可继续工作。如欲推迟一个线程的执行，应使用sleep()函数。

Thread类

- java.lang包(**java.lang.Thread**)
- 构造函数
  - Thread();
  - Thread(String threadname); 指定线程实例名
- 线程的优先级控制
  - 三个常量：

    MAX_PRIORITY　　10;

    MIN _PRIORITY　　1;

    NORM_PRIORITY　 5;
  - getPriority()　返回线程优先值

    setPriority(int newPriority)　改变线程的优先级
  - 线程创建时继承父线程的优先级

Thread类的有关方法

- void start(): 由Newborn到Runnable
  - 启动线程
- String getName(): 返回线程的名称
- run(): 线程在被调度时执行的操作
- static void sleep(指定时间毫秒):
  - 令当前活动线程在指定时间段内放弃对CPU控制,使其他线程有机会被执行,时间到后重排队
  - 产生例外InterruptedException
  - 用try块调用sleep()，用catch块处理例外

Thread类的有关方法(续)

- suspend()：挂起线程，处于阻塞状态
- resume()：恢复挂起的线程，重新进入就绪队列排队

应用：可控制某线程的暂停与继续

方法：设一状态变量suspendStatus=false(初始)

暂停：if(!suspendStatus)
{T.suspend(); suspendStatus=true;}

继续：if(suspendStatus)
{T.resume(); suspendStatus=false;}

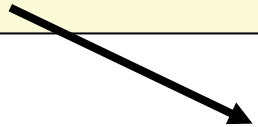
Thread类的有关方法(续)

- static void yield()：对正在执行的线程
  - ✓ 若就绪队列中有与当前线程同优先级的排队线程，则当前线程让出CPU控制权,移到队尾
  - ✓ 若队列中没有同优先级的线程，忽略此方法
- stop(): 强制线程生命期结束
- boolean isAlive()：返回boolean，表明是否线程还存在
- static currentThread(): 返回当前*线程*

生成与运行线程 –方法1

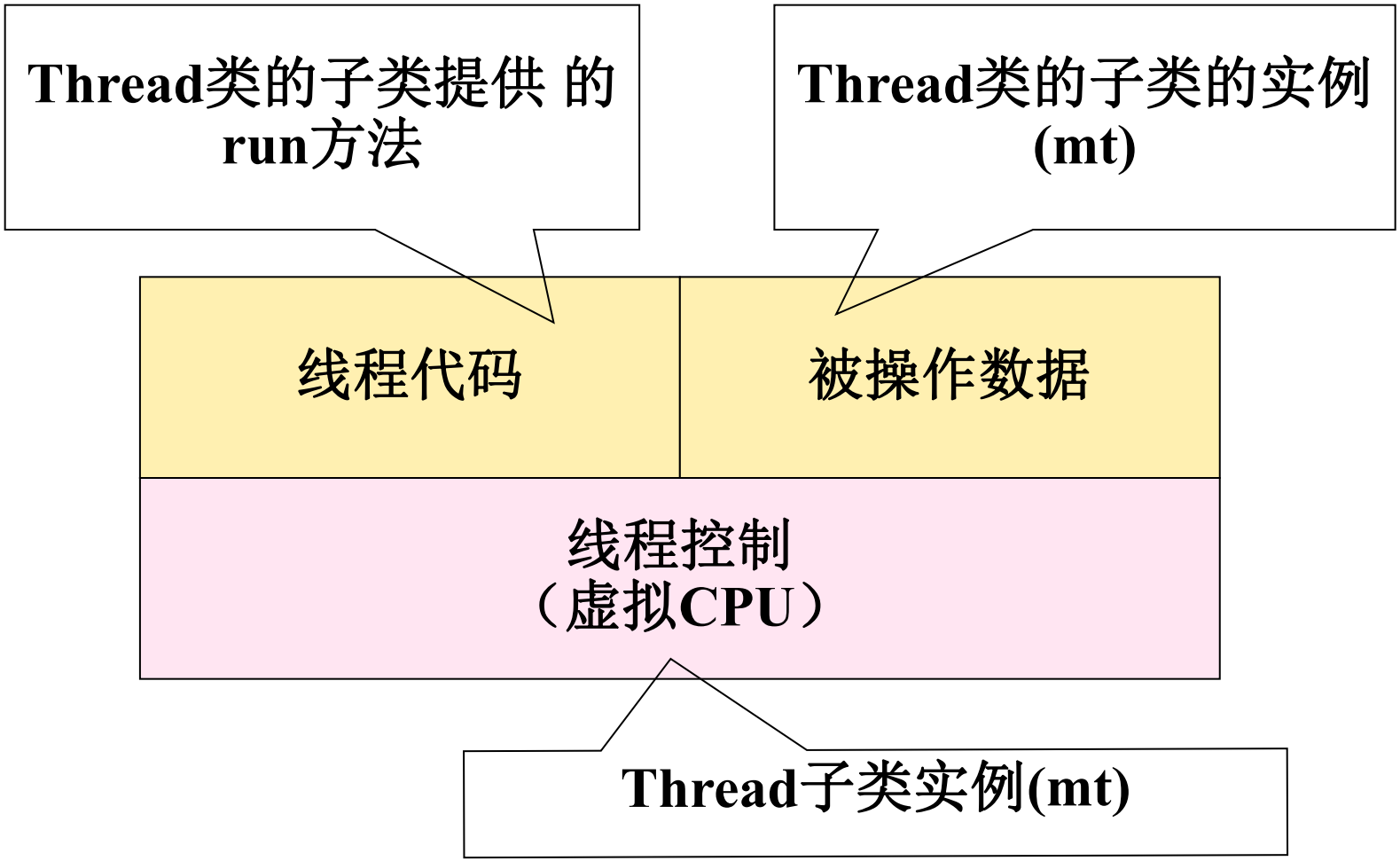
```
class MyThread extends Thread {
  public void run()  { 线程体...}
}
```

```
MyThread  mt = new MyThread();
mt.start();
```



执行run()方法

建立线程


Thread类的子类提供 的run方法
Thread类的子类的实例(mt)
线程代码
被操作数据
线程控制
（虚拟CPU）
Thread子类实例(mt)

方法之二：Runnable接口

- Runnable接口
  - ✓ 自定义类实现Runnable接口
  - ✓ 使用Thread类的另一构造函数：
    - Thread(Runnable, String)

      用实现了Runnable接口的类的对象中所定义的run()方法，来覆盖新创建的线程对象的run()方法
  - ✓ 使用start()启动线程

方法之二：Runnable（续）

- 例：

```
class A implements Runnable{
    public void run(){….}
}
class B {
    public static void main(String[] arg){
      Runnable a=new A();
      Thread t=new Thread(a);
      t.start();
    }
}
```

建立线程例: MyThreadWebServer.java

Web服务器的线程体

```
class Process implements  Runnable{
     Socket s;          static int i;
    public Process (Socket s1) {          s=s1;       }
    public void run() {
        try {
            PrintStream out = new PrintStream(s.getOutputStream());
            BufferedReader in
                        = new BufferedReader(new InputStreamReader(s.getInputStream()));
                String info=in.readLine();
            System.out.println("now got "+info);   out.println("HTTP/1.0 200 OK");
            out.println("MIME_version:1.0");         out.println("Content_Type:text/html");
            i++;
            String c="<html> <head></head><body> <h1> Hi,  This is "+i+"</h1></Body>
    </html>";
            out.println("Content_Length:"+c.length( ));   out.println(""); out.println(c);
            out.close();
            s.close(); in.close( );
    } catch (IOException e) {
            System.out.println("Exception:"+e); }
}      }
```

建立线程例: MyThreadWebServer.java

```
import java.net.*;
import java.io.*;
public class  MyThreadWebServer {
    public static void main(String args[]){
        try {
            ServerSocket ss=new ServerSocket(8080);
            System.out.println("Web Server OK");
            while (true) {
                Socket s=ss.accept();
                Process p=new Process(s);
                Thread  t=new Thread(p);
                t.start( );
            }
        } catch (Exception e) {System.out.println(e);}
    }
}
```

生成与运行线程 –方法2

```
class MyRun implements Runnable {
  public void run() {线程体…}
}

MyRun mr = new MyRun();
Thread  t1 = new Thread(mr) ;
t1.start();     //Thread实例用于线程控制
```

- 适合于：定义run()方法的类必须是其他类或其他类的子类。

方法之二：Runnable（续）

- 两种方法的选择
  - ✓ 当需要从其他类，如Applet类继承时，使用**Runnable接口**
  - ✓ 当编写简单的程序时，可考虑使用继承Thread类

- 例：RandomCharacters.java (D:\JavaEx\CH8\RandomCharacters.html)
  - ✓ 具体运行结果（线程调度）与平台有关

RandomCharacters.java

```
public class RandomCharacters extends JApplet implements
        Runnable, ActionListener {
  private String alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  private Thread threads[];
  private boolean suspended[];

  public void init()
  {
    outputs = new JLabel[ SIZE ];
    checkboxes = new JCheckBox[ SIZE ];
```

比较

- 实现**Runnable**的优点
  - 从面向对象的角度来看，Thread类是一个虚拟处理机严格的封装，因此只有当处理机模型修改或扩展时，才应该继承类。一个正在运行的线程的处理机、代码和数据部分是不同的。
  - 由于Java技术只允许单一继承，所以如果你已经继承了Thread，你就不能再继承其它任何类，例如Applet。在某些情况下，这会使你只能采用实现Runnable的方法。
  - 因为有时你必须实现Runnable，所以你可能喜欢保持一致，并总是使用这种方法。
- 继承**Thread**的优点
  - 当一个run()方法体现在继承Thread类的类中，用this指向实际控制运行的Thread实例。因此，代码不再需要使用如下控制：
    - Thread.currentThread().join();
    - 而可以简单地用：join();

举例：警察和小偷多线程之——Thief.java

```
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

```
Thief    Tth=new Thief( "hijackT‟ );
Tth.start();
```

Policeman.java

```
//方法2实现Runnable接口创建线程
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");
```

举例：警察和小偷多线程

```
public 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();
    }
}
```

4. 线程间数据间的交流

- 例：警察和小偷这个例子中警察线程中希望获得小偷的具体位置以便警察判断能否抓住小偷，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 CatchTest {
    int x ; //policeman
   int y;   //thief
    public static void main(String args[]){
            CatchTest  x=new  CatchTest ();
          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.println("police : "+x+"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);
          }   }
}
}
```

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(pos.p_x=-100;pos.p_x<=100;pos.p_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(pos.th_y=-100;pos.th_y<=100;pos.th_y++)
            {  pos.show( ); }
     } }
  }
```

```
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();
} }
```

2020-12-01 周二CS

/*多线程实现龟兔赛跑，java多线程入门实例
两个线程同时开跑。
*/
//模拟跑步者的 跑步者线程
CS1912.Thread_RaceRun.java

2022-05-13 #12

5. 线程的同步与互斥

- 问题的提出
  - ✓ 多个线程执行的不确定性引起执行结果的不稳定

    如线程A: A1－A2　　线程B: B1－B2
  - ✓ 多个线程对内存、数据的共享，会造成操作的不完整性，会破坏数据。

    如push(a): i++; num[i]=a;　pop(): 取出num[i];i--;

Ø **two kinds of errors possible:**

ü ***thread interference***

ü ***memory consistency errors***

file:///I:/tutorial/essential/concurrency/sync.html

线程的同步与互斥

问题的解决

- 同步: 用**synchronized**关键字前缀给针对共享资源的操作加锁；
- 同步方法、同步块

```
synchronized void push();
 synchronized int pop();
```

- 临界区
- 实现机制：管程

synchronized的作用：

- 关键字是 Java 并发编程中线程同步的常用手段之一，其作用：
  - 确保线程互斥的访问同步锁自动释放，多个线程操作同个代码块或函数必须排队获得锁，
  - 保证共享变量的修改能够及时可见，获得锁的线程操作完毕后会将所数据刷新到共享内存区；
  - 不解决重排序，但保证有序性。
- synchronized 关键词作用在方法的前面，用来锁定方法，其实默认锁定的是 this 对象。

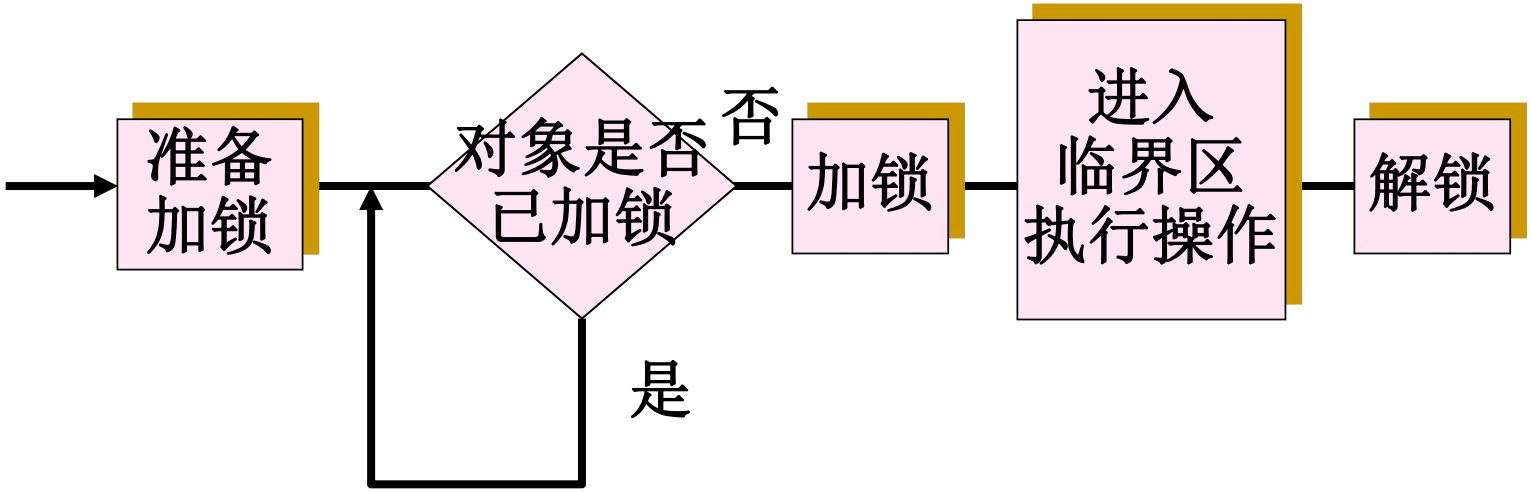

NANJING AGRICULTURAL UNIVERSITY
1914

线程的同步与互斥

- 对象互斥锁

  在Java中，每个对象有一个“互斥锁”，该锁可用来保证在同一时刻只能有一个线程访问该对象。

- 锁的使用过程（当一个线程要操作一个对象时）



线程的同步与互斥

- 加锁1 (临界区-方法）

  synchronized 方法名{ … } 进入该方法时加锁

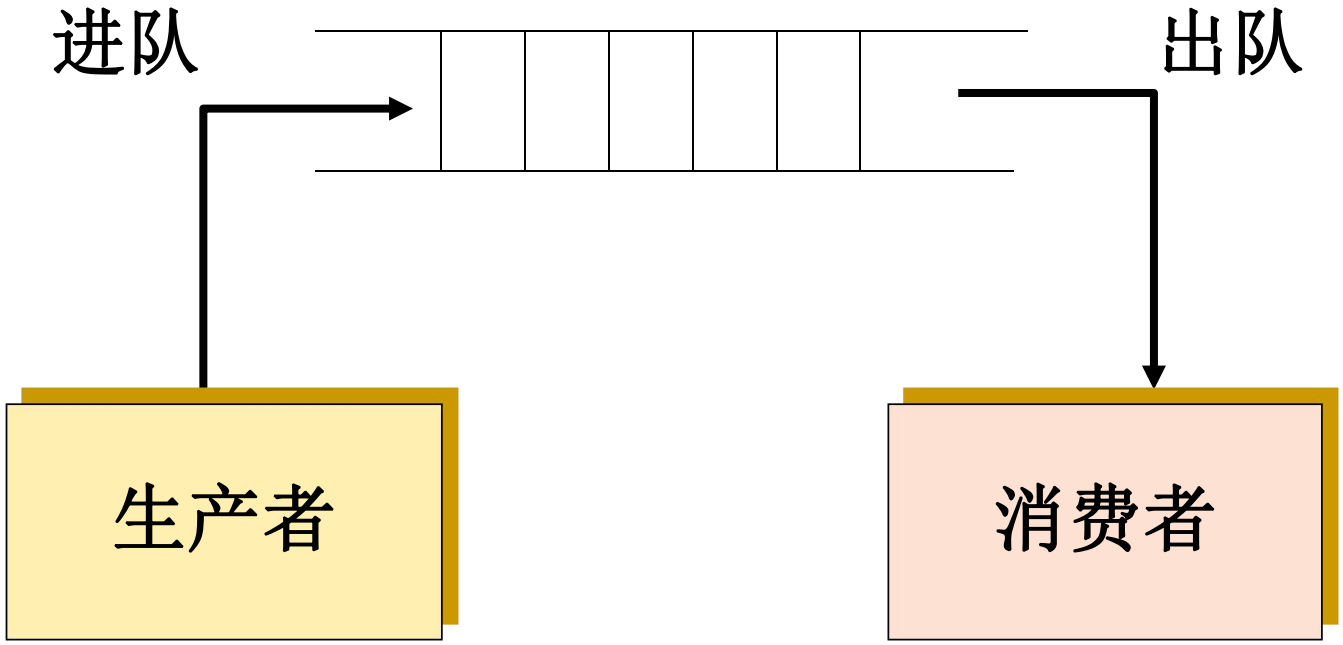
- 加锁2（临界区-代码块）

```
方法名{
        ……
        synchronized(this){… } //进入该代码段时加锁
        ……
}
```

- 一个线程为某对象加锁后，便对该对象具有了监控权。

线程的同步与互斥

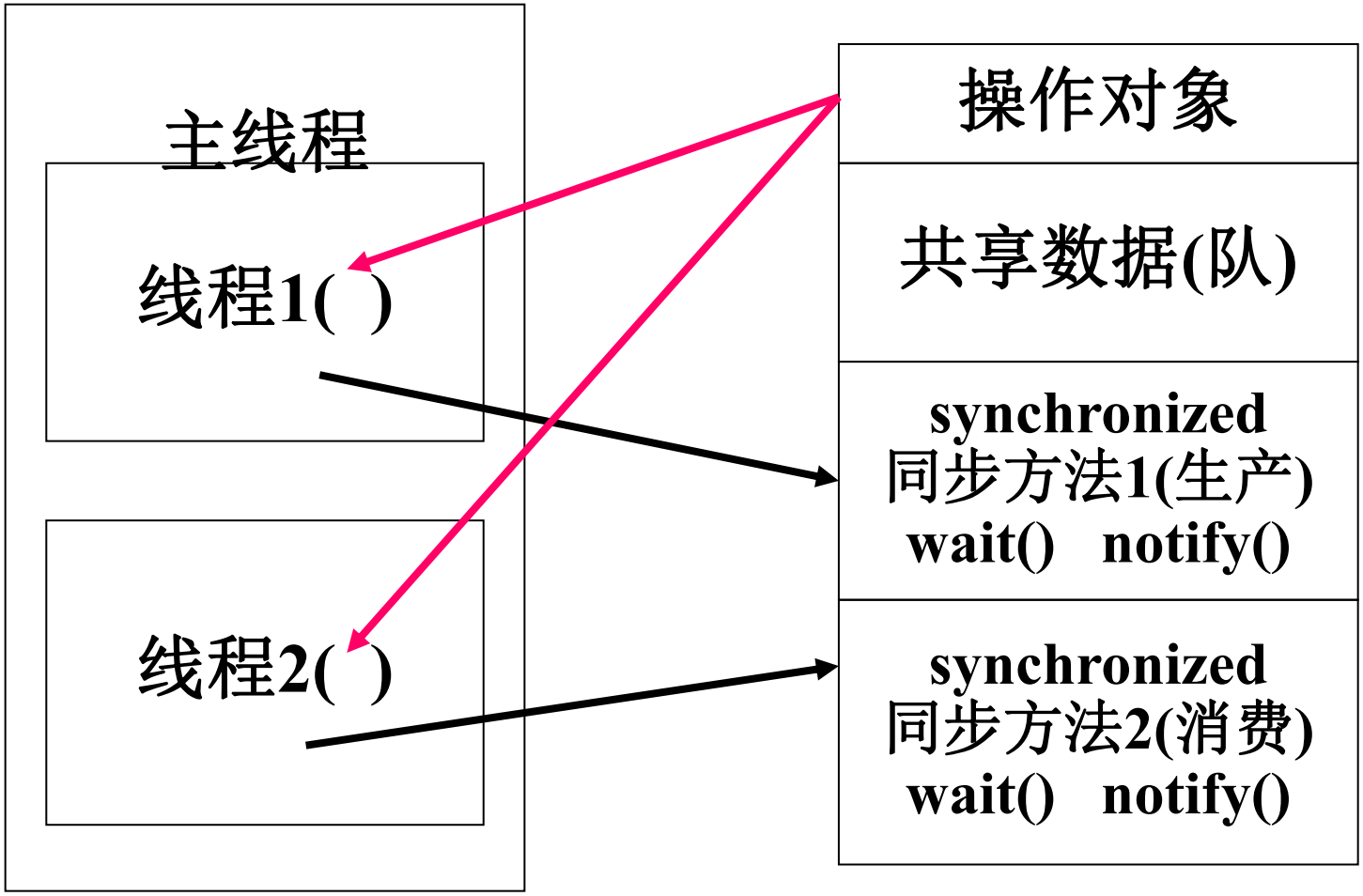
- 线程间需协调与通讯：**生产者/消费者问题**



线程的同步与互斥

- wait()与notify()
  - Object类的方法：public final void
  - wait()：令当前线程挂起并放弃管程，同步资源解锁，使别的线程可访问并修改共享资源，而当前线程排队，等候再次对资源的访问
  - notify()唤醒正在排队等待资源管程的线程中优先级最高者，使之执行并拥有资源的管程
  - wait() + notify() + 标志变量：可协调、同步不同线程的工作
- 例：CatchTest3.java

线程的同步与互斥


主线程
线程1( )
线程2( )
操作对象
共享数据(队)
synchronized
同步方法1(生产)
wait() notify()
synchronized
同步方法2(消费)
wait() notify()

线程的同步与互斥

public final void **wait**() 方法

- 在当前线程中调用方法： 对象名.wait()
- 使当前线程进入等待（某对象）状态，直到另一线程对该对象发出notify(或notifyAll)为止。
- 调用方法的必要条件：当前线程必须具有对该对象的监控权（加锁）
- 当前线程将释放对象监控权 ，然后进入等待队列（wait队列）。
- 在当前线程被notify后，要重新获得监控权，然后从断点处继续代码的执行。

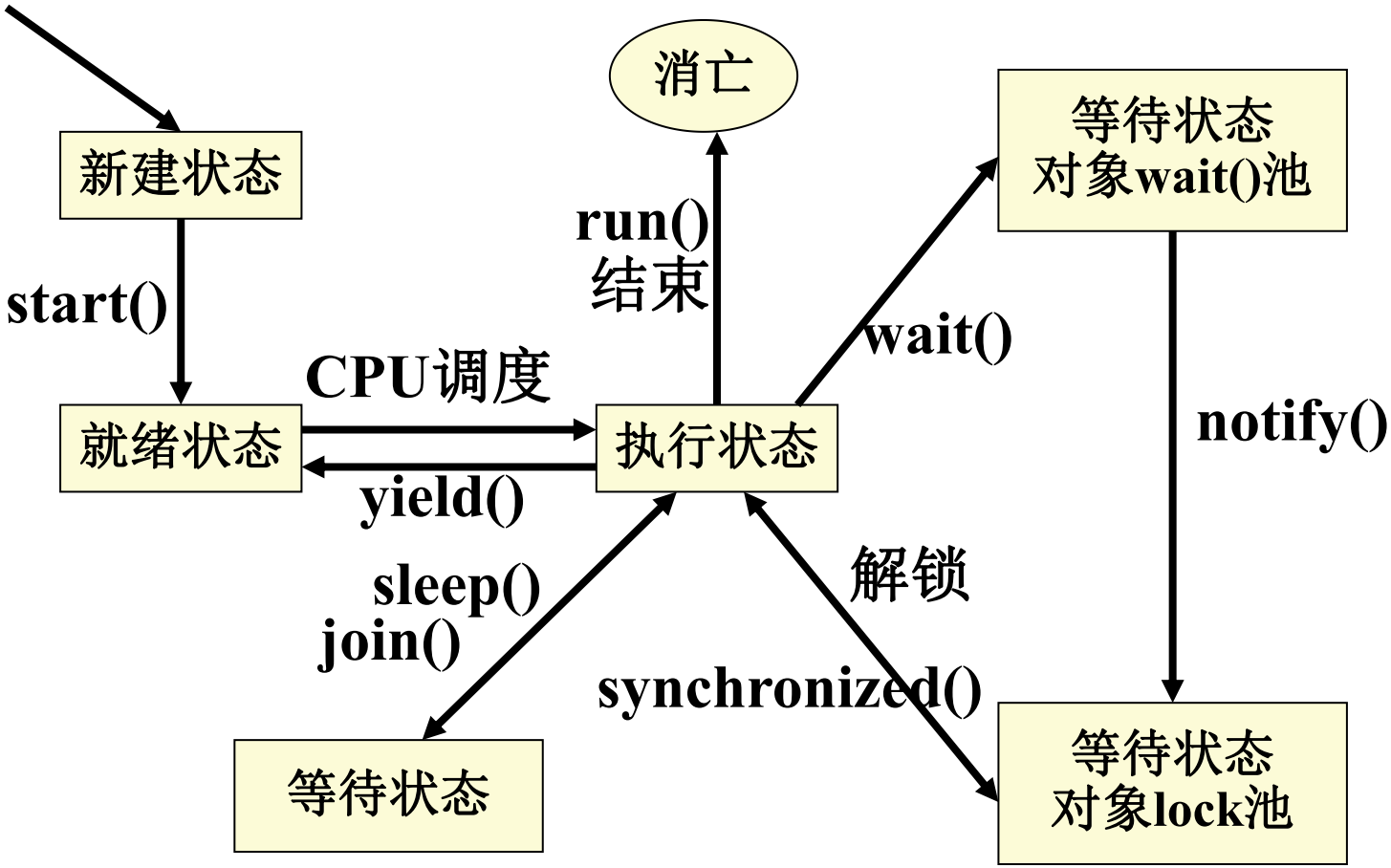
线程的同步与互斥

public final void **notify**()　方法

- 在当前线程中调用方法：　对象名.notify()
- 功能：唤醒等待该对象监控权的一个线程。
- 调用方法的必要条件：当前线程必须具有对该对象的监控权（加锁）　　notifyAll()
- 唤醒wait队列中的所有线程，并把它门移入锁申请队列。

```
public class Drop {  // 仓库
   private String message;
   private boolean empty = true;
   public synchronized String take() {
      while (empty) {
         try {     wait();
         } catch (InterruptedException e) {}
       }
       empty = true;
       notifyAll();           //Notify producer that status has changed.
      return message;
   }
   public synchronized void put(String message) {
.       while (!empty) {        // Wait until message has been retrieved
          try {         wait();
               } catch (InterruptedException e) {}
        }
        empty = false;
        this.message = message;
        notifyAll();
     }
  }
```

线程状态


新建状态
start()
就绪状态
CPU调度
yield()
执行状态
run()
结束
消亡
wait()
等待状态
对象wait()池
notify()
sleep()
join()
等待状态
解锁
synchronized()
等待状态
对象lock池

线程调度

- public static void sleep(long millis)

  当前进程休眠指定时间

- public static void yield()

  主动让出CPU，重新排队

  正在执行的线程将CPU让给其他具有相同优先级的线程，自己进入就绪状态重新排队

- public final void join()

  等待某线程结束

```
public class Producer implements Runnable {//生产者线程类
    private Drop drop;
    public Producer(Drop drop) {
        this.drop = drop;    }
    public void run() {
        String importantInfo[] = { “Mares eat oats”,  “Does eat oats”,
                                   "Little lambs eat ivy",    "A kid will eat ivy too"      };
        for (int i = 0; i < importantInfo.length; i++) {
            drop.put(importantInfo[i]);
            System.out.format("Producer: %s%n", importantInfo[i]);
            try {
                Thread.sleep((int)Math.random()*2000);
              } catch (InterruptedException e) {}
        }
        drop.put("DONE");
    }
}
```

等待另一线程结束

```
Runnable ot = new otherThread() ;
Thread tt = new Thread(ot) ;
tt.start();
  …                              //执行自己的工作
 try {
   tt.join () ;
 } catch( interruptedException e){}
 ….                               //继续做自己的事
```

线程的同步与互斥

- Wait_Notify 程序Drop.java
- 创建用户的线程子类
  - Producer:产生数据（存数据）；
  - Consumer:消费数据（取数据）
- Drop类，共享数据区，同步方法
  - put(String message)方法
  - **String take()**方法
- 主类中创建共享数据对象，并启动两线程

```
public class Consumer implements Runnable {//消费者线程类
    private Drop drop;
    public Consumer(Drop drop) {
        this.drop = drop;
    }
    public void run() {
        for (String message = drop.take(); !
message.equals("DONE");
             message = drop.take()) {
            System.out.format("Consumer:  %s%n", message);
            try {
                Thread.sleep((int)Math.random()*5000);
            } catch (InterruptedException e) {}
        }
    }
}
```

```
class ProducerConsumerTest { //主类：测试
   public static void main(String[] args) {
       Drop drop = new Drop();  //the shared data object
         (new Thread(new Producer(drop))).start(); //启动生产者线程
         (new Thread(new Consumer(drop))).start(); //启动消费者线程
   }
} //end of class
```

程序执行结果：

生产者线程和消费者线程严格地轮流执行，获得了线程间的协调执行。

```
Producer: Mares eat oats
Consumer:  Mares eat oats
Producer: Does eat oats
Producer: Little lambs eat ivy
Consumer:  Does eat oats
Producer: A kid will eat ivy too
Consumer:  Little lambs eat ivy
Consumer:  A kid will eat ivy too
```

守护线程(dameon thread)

- 守护线程是一个为和它同时运行的其它线程或者对象提供服务的线程。任何Java线程都可以是一个守护线程。
  - Th.SetDameon()设置为true。
  - isDameon()函数可以测试一个线程是否为守护线程。

线程的死锁

- 死锁
  - 不同的线程分别占用对方需要的同步资源不放弃，都在等待对方放弃自己需要的同步资源，就形成了线程的死锁
- 解决方法
  - 专门的算法、原则
  - 尽量减少同步资源的定义

Deadlock.java

```
public class Deadlock {
    static class Friend {
        private final String name;

        public Friend(String name) {
            this.name = name;
        }
        public String getName() {
            return this.name;
        }
```

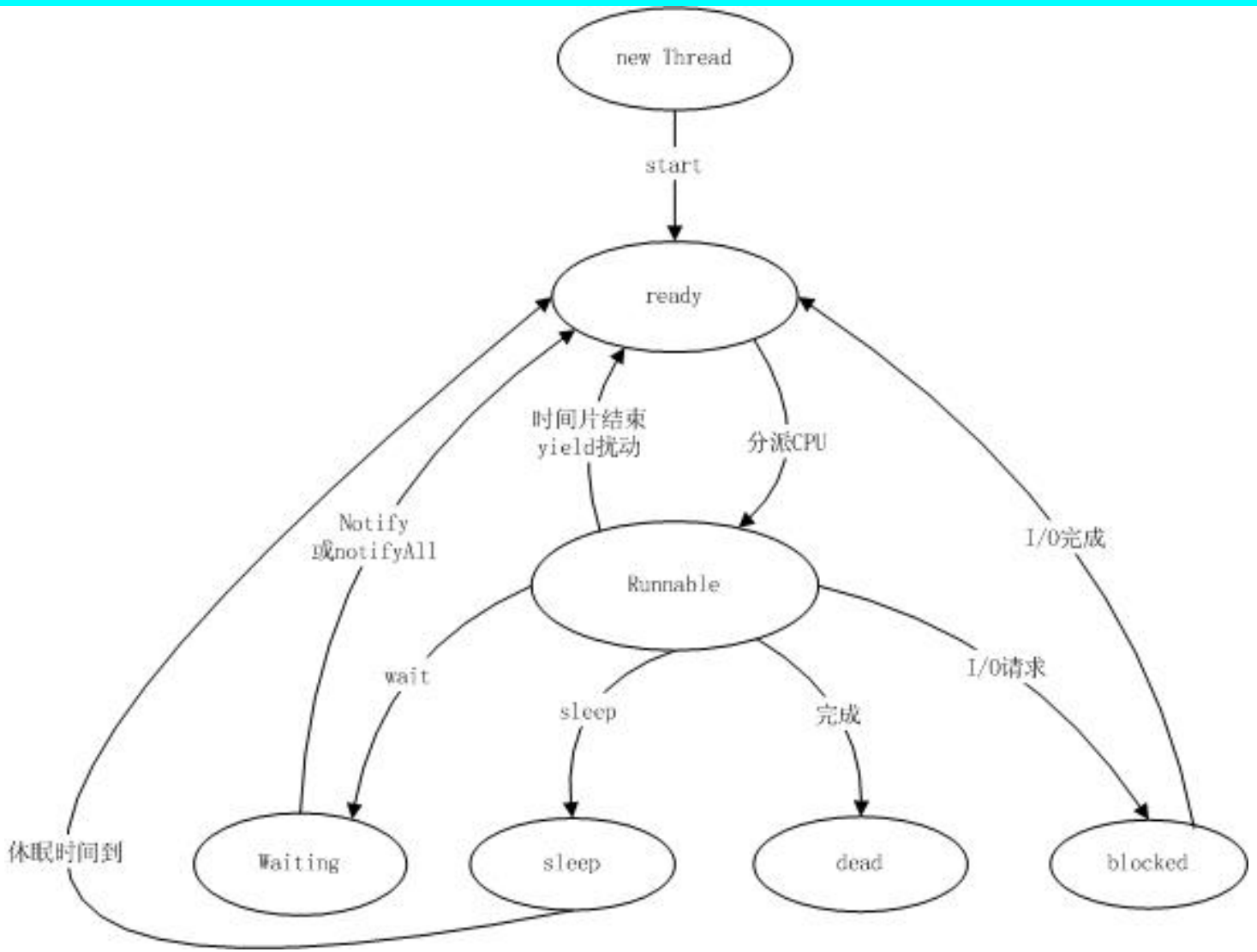
```
public synchronized void bow(Friend
bower) {
            System.out.format("%s: %s"
                + "  has bowed to me!%n",
                this.name, bower.getName());
            bower.bowBack(this);
        }
public synchronized void
bowBack(Friend bower) {
            System.out.format("%s: %s"
                + " has bowed back to
me!%n",
                this.name, bower.getName());
        }
```

什么是死锁

- 死锁就是两个或两个以上的线程被无限的阻塞，线程之间相互等待所需资源。这种情况可能发生在当两个线程尝试获取其它资源的锁，而每个线程又陷入无限等待其它资源锁的释放，除非一个用户进程被终止。就 JavaAPI 而言，线程死锁可能发生在以下情况。
  - 当两个线程相互调用 Thread.join()
  - 当两个线程使用嵌套的同步块，一个线程占用了另外一个线程必需的锁，互相等待时被阻塞就有可能出现死锁。

Starvation and Livelock

- 线程饿死，线程活锁
- 当所有线程阻塞，或者由于需要的资源无效而不能处理，无非阻塞线程资源可用。JavaAPI中线程活锁可能发生在以下情形：
  - 当所有线程在程序中执行 Object.wait (0)，参数为 0 的 wait 方法。程序将发生活锁直到在相应的对象上有线程调用 Object.notify ()或者 Object.notifyAll ()。
  - 当所有线程卡在无限循环中。


new Thread
start
ready
时间片结束
yield扰动
分派CPU
Notify
或notifyAll
I/O完成
Runnable
wait
sleep
完成
I/O请求
休眠时间到
Waiting
sleep
dead
blocked

课后作业

- 完成警察抓小偷程序
- 完成生产消费同步程序
- 线程同步实现银行帐户存取款程序（支付宝）。
  - ✓ **Accout 类**
    - **SaveMoney线程**
    - **TreadDraw线程**
- 时间服务器*（网络应用）

2021-5-20 12周