JAVA 程序设计

潘微科

感谢：教材《Java大学实用教程》的作者和其他老师提供PowerPoint讲义等资料！
说明：本课程所使用的所有讲义，都是在以上资料上修改的。

Outline

- 8.1 Java中的线程
- 8.2 线程的生命周期
- 8.3 线程的优先级与调度管理
- 8.4 Thread的子类创建线程
- 8.5 Runnable接口
- 8.6 线程的常用方法
- 8.7 线程同步
- 8.8 使用wait(),notify(),notifyAll()协调同步线程
- 8.9 挂起、恢复和终止线程
- 8.10 线程联合
- 8.11 守护线程

8.1 Java中的线程

- 程序是一段静态的代码，它是应用软件执行的蓝本。

- 进程是程序的一次动态执行过程，它对应了从代码加载、执行至执行完毕的一个完整过程，这个过程也是进程本身从产生、发展至消亡（完成）的过程。

- 线程是比进程更小的执行单位。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，每条线索（即每个线程）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。

8.1 Java中的线程

- Java应用程序总是从**主类**（main class）的main()方法开始执行。

- 当JVM加载代码，发现main()方法之后，就会启动一个线程，这个线程称作“**主线程**”，该线程负责执行main()方法。
- 那么，在main()方法中再创建的线程，就称为**主线程中的线程**。

- 如果main()方法中没有创建其他线程，那么当main()方法执行完最后一条语句，即main()方法返回时，JVM就会结束我们的Java应用程序。
- 如果main()方法中又创建了其他线程，那么**JVM就要在主线程和其他线程之间轮流切换，保证每个线程都有机会使用CPU资源**，main()方法即使执行完最后的语句，JVM也不会结束我们的程序，JVM一直要等到主线程中的所有线程都结束之后，才结束我们的Java应用程序。

Outline

- 8.1 Java中的线程
- 8.2 线程的生命周期
- 8.3 线程的优先级与调度管理
- 8.4 Thread的子类创建线程
- 8.5 Runnable接口
- 8.6 线程的常用方法
- 8.7 线程同步
- 8.8 使用wait(),notify(),notifyAll()协调同步线程
- 8.9 挂起、恢复和终止线程
- 8.10 线程联合
- 8.11 守护线程

8.2 线程的生命周期

- 1.线程的4种状态
- 在Java语言中，**Thread类或其子类**创建的**对象**称作线程，新建的线程在它的一个完整的生命周期中通常要经历4种状态。

- （1）新建
- 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。此时，已经有了相应的内存空间和其他资源。

8.2 线程的生命周期

- （2）运行
- 线程创建后仅仅是占有了内存资源，在JVM管理的线程中还没有这个线程，此线程必须调用**start()**方法（是一个从父类继承的方法）通知JVM，这样JVM就会知道又有一个新的线程排队**等候**切换了。

- 当JVM将CPU使用权切换给线程时，如果线程是Thread类的子类创建的，该类中的**run()**方法就立刻执行。所以我们必须**在子类中重写（override）父类的run()方法**。Thread类中的run()方法没有具体内容，程序要在Thread类的子类中**重写**run()方法来覆盖父类的run()方法，run()方法规定了该线程的具体使命。

- **在线程没有结束run()方法之前，不要让线程再调用start()方法，否则将发生ILLegalThreadStateException异常。**

8.2 线程的生命周期

- （3）中断
- 有4种原因的中断：
- (a) **JVM**将CPU资源从当前线程切换给其他线程，使本线程让出CPU的使用权，进而处于中断状态。

- (b) 线程使用CPU资源期间，执行了**sleep(int millsecond)**方法，使当前线程进入休眠状态。
  - sleep(int millsecond)方法是Thread类中的一个**类方法/静态方法（static method）**，线程一旦执行了sleep(int millsecond)方法，就立刻让出CPU的使用权，使当前线程处于中断状态。经过参数millsecond指定的毫秒之后，该线程就重新进到线程队列中排队等待CPU资源，以便从中断处继续运行。

8.2 线程的生命周期

- (c) 线程使用CPU资源期间，执行了**wait()**方法，使得当前线程进入**中断（等待）**状态。等待状态的线程**不会主动**进到线程队列中排队等待CPU资源，必须由其他线程调用**notify()**方法通知它，使得它重新进到线程队列中排队等待CPU资源，以便从中断处继续运行。有关wait(), notify()和notifyAll()方法将在第8节详细讨论。

- (d) 线程使用CPU资源期间，执行某个操作**进入中断（阻塞）状态**，比如执行读/写操作引起阻塞。进入阻塞状态时线程不能进入排队队列，只有当引起阻塞的原因消除时，线程才重新进到线程队列中排队**等待CPU资源**，以便从原来中断处开始继续运行。

8.2 线程的生命周期

- （4）死亡
- 处于死亡状态的线程不具有继续运行的能力。线程死亡的原因：
  - 正常运行的线程**完成**了它的全部工作，即执行完run()方法中的全部语句，结束了run()方法。
  - 线程**被提前强制终止**，即强制run()方法结束。

- 所谓死亡状态就是**线程释放了实体**，即释放了分配给线程对象的**内存**。

8.2 线程的生命周期

- 【例子1】

```
class WriteWordThread extends Thread
{
    WriteWordThread(String s)
    {
        setName(s);
    }
    public void run()
    {
        for(int i=1; i<=3;i++)
            System.out.println("Thread: " + getName());
    }
}
```

```
public class Example8_1
{
    public static void main(String args[])
    {
        WriteWordThread zhang, wang;
        zhang = new WriteWordThread("Zhang"); //新建线程
        wang = new WriteWordThread("Wang"); //新建线程
        zhang.start(); //启动线程
        for(int i=1; i<=3; i++)
        {
            System.out.println("Main Thread");
        }
        wang.start(); //启动线程
    }
}
```

计算机反复运行的结果不尽相同

8.2 线程的生命周期

- 上述程序在不同的计算机运行或在同一台计算机反复运行的结果不尽相同，输出结果依赖当前CPU资源的使用情况。

- 为了使结果尽量不依赖当前CPU资源的使用情况，我们应当**让线程主动调用sleep方法让出CPU的使用权**进入中断状态。

8.2 线程的生命周

- 【例子2】

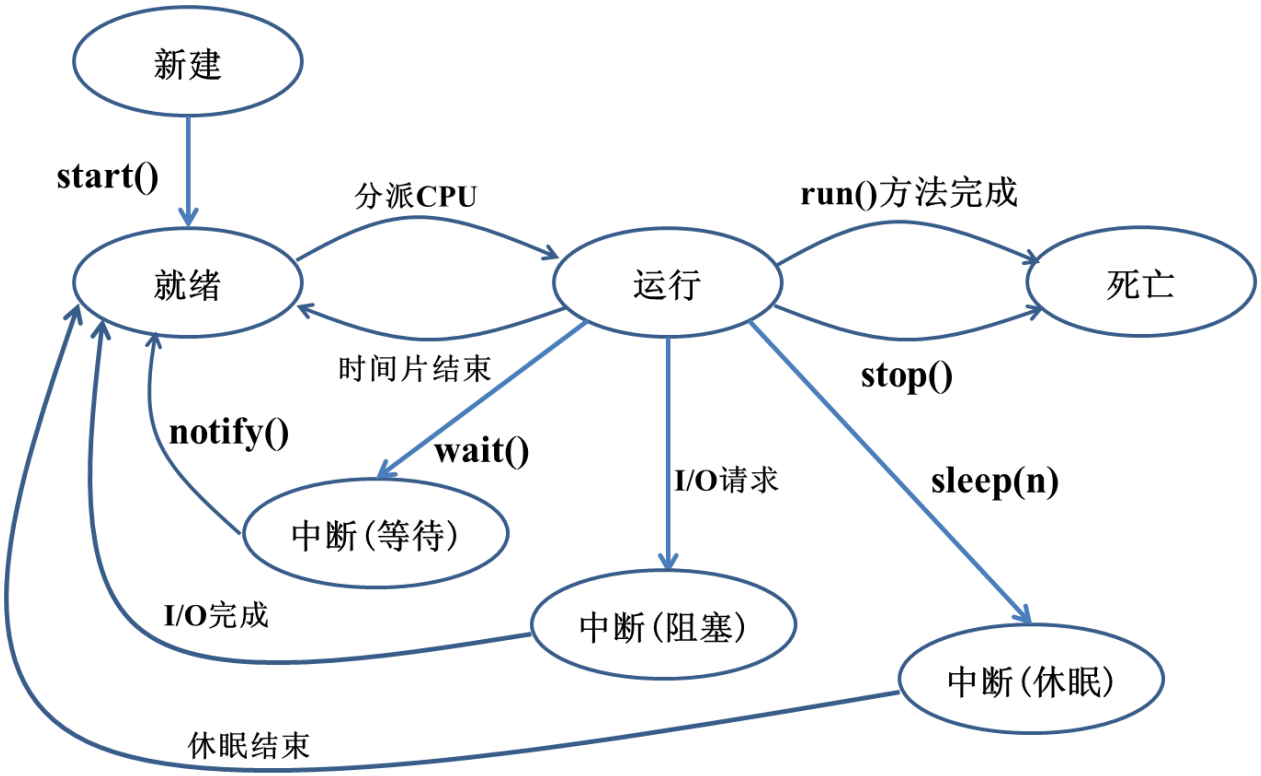
```
class WriteWordThread extends Thread
{
    int n = 0;
    WriteWordThread(String s, int n)
    {
        setName(s);
        this.n = n;
    }
    public void run()
    {
        for(int i=1; i<=3; i++)
        {
            System.out.println("Thread: " + getName());
            try
            {
                sleep(n);
            }
            catch(InterruptedException e) {}
        }
    }
}
```

```
public class Example8_2
{
    public static void main(String args[])
    {
        WriteWordThread zhang, wang;
        zhang = new WriteWordThread("Zhang", 200);
        wang = new WriteWordThread("Wang", 100);
        zhang.start();
        wang.start();
    }
}
```

计算机反复运行的结果不尽相同

8.2 线程的生命周期

- 《Java语言程序设计教程》（UOOC联盟指定参考书）



Outline

- 8.1 Java中的线程
- 8.2 线程的生命周期
- 8.3 线程的优先级与调度管理
- 8.4 Thread的子类创建线程
- 8.5 Runnable接口
- 8.6 线程的常用方法
- 8.7 线程同步
- 8.8 使用wait(),notify(),notifyAll()协调同步线程
- 8.9 挂起、恢复和终止线程
- 8.10 线程联合
- 8.11 守护线程

8.3 线程的优先级与调度管理

- JVM中的**线程调度器**负责管理线程，调度器把线程的优先级分为10个级别，分别用Thread类中的类常量表示。每个Java线程的优先级都在常数1（Thread.MIN_PRIORITY）到常数10（Thread.MAX_PRIORITY）的范围内。

- 如果没有明确设置线程的优先级别，每个线程的优先级都为常数5（Thread.NORM_PRIORITY）。

- 线程的优先级可以通过setPriority(int grade)方法调整，这一方法需要一个int类型参数。如果此参数不在1-10的范围内，那么setPriority便产生一个IllegalArgumentException异常。
- getPriority方法返回线程的优先级。需要注意是，有些操作系统只能识别3个级别：1, 5, 10。

8.3 线程的优先级与调度管理

- JVM中的线程调度器**使高优先级的线程能始终运行**。

- 如果有A,B,C,D四个线程，A和B的级别高于C和D，那么调度器首先以轮流的方式执行A和B，**一直等到A和B都执行完毕进入死亡状态**，才会在C和D之间轮流切换。

Outline

- 8.1 Java中的线程
- 8.2 线程的生命周期
- 8.3 线程的优先级与调度管理
- 8.4 Thread的子类创建线程
- 8.5 Runnable接口
- 8.6 线程的常用方法
- 8.7 线程同步
- 8.8 使用wait(),notify(),notifyAll()协调同步线程
- 8.9 挂起、恢复和终止线程
- 8.10 线程联合
- 8.11 守护线程

8.4 Thread的子类创建线程

- 在Java语言中，用Thread类或其子类创建线程对象。这一节将讲述怎样**用Thread子类创建对象**。
- 用户可以**扩展 Thread类**，但需要**重写父类的run()方法**，其目的是规定线程的具体操作，否则线程就什么也不做，因为父类的run()方法中没有任何操作语句。

- 下面例子3中除主线程外还有两个线程，这两个线程分别在命令行窗口的左侧和右侧顺序地一行一行地输出字符串。主线程负责判断输出的行数，当其中任何一个线程输出8行后，就结束**进程**。本例题中用到了System类中的类方法exit(int n)，**主线程使用该方法结束整个程序**。

8.4 Thread的子类

- 【例子3, 1/2】

```
class Left extends Thread
{
    int n = 0;
    public void run()
    {
        while(true)
        {
            n++;
            System.out.printf("\n%s","Left");
            try
            {
                sleep((int)(Math.random()*100)+100);
            }
            catch(InterruptedException e) {}
        }
    }
}
```

```
class Right extends Thread
{
    int n = 0;
    public void run()
    {
        while(true)
        {
            n++;
            System.out.printf("\n%40s","Right");
            try
            {
                sleep((int)(Math.random()*100)+100);
            }
            catch(InterruptedException e){}
        }
    }
}
```

8.4 Thread的子类创建线程

- 【例子3, 2/2】

```
public class Example8_3
{
    public static void main(String args[])
    {
        Left left = new Left();
        Right right = new Right();
        left.start();
        right.start();
        while(true)
        {
            if(left.n>=8 || right.n>=8)
              System.exit(0);
        }
    }
}
```

Outline

- 8.1 Java中的线程
- 8.2 线程的生命周期
- 8.3 线程的优先级与调度管理
- 8.4 Thread的子类创建线程
- 8.5 Runnable接口
- 8.6 线程的常用方法
- 8.7 线程同步
- 8.8 使用wait(),notify(),notifyAll()协调同步线程
- 8.9 挂起、恢复和终止线程
- 8.10 线程联合
- 8.11 守护线程

8.5 Runnable接口

- 使用Thread子类创建线程的优点是：我们可以在子类中增加新的成员变量，使线程具有某种属性，也可以在子类中增加新的方法，使线程具有某种功能。但是，Java不支持多继承，Thread类的子类不能再扩展其他的类。

8.5 Runnable接口

- 1.Runnable接口与目标对象
- **创建线程的另一个途径**就是用Thread类直接创建线程对象。使用Thread类创建线程对象时，通常使用的**构造方法**是：

```
Thread(Runnable target)
```

  - 该构造方法中的参数是一个Runnable类型的接口，因此，在创建线程对象时必须向构造方法的参数传递**一个实现Runnable接口的类的实例**，该实例对象称作所创建线程的**目标对象（目标任务task）**，当线程调用start()方法后，一旦轮到它来享用CPU资源，目标对象就会自动调用接口中的run()方法（**接口回调**），这一过程是自动实现的，用户程序只需要让线程调用start()方法即可。

8.5 Runnable接口

- 下面的例子4中，两个线程：zhang和cheng，使用同一**目标对象**。两个线程**共享**目标对象的money。当money的值小于100时，线程zhang结束自己的run()方法进入死亡状态；当money的值小于60时，线程cheng结束自己的run()方法进入死亡状态。

```
class Bank implements Runnable
{
    private int money = 0; String name1,name2;
    Bank(String s1,String s2){ name1 = s1; name2 = s2; }
    public void setMoney(int mount){ money = mount; }
    public void run()
    {
        while(true)
        {
                money = money-10;
                if(Thread.currentThread().getName().equals(name1)){
                    System.out.println(name1 + ": " + money);
                    if(money<=100){
                            System.out.println(name1 + ": Finished");
                            return;
                    }
                }
                else
                  if(Thread.currentThread().getName().equals(name2)){
                            System.out.println(name2 + ": " + money);
                            if(money<=60){
                                    System.out.println(name2 + ": Finished");
                                    return;
                             }
                  }
                try{ Thread.sleep(800); }
                catch(InterruptedException e) {}
            }
        }
}
```

8.5 Runnable接口

```
public class Example8_4
{
    public static void main(String args[])
    {
        String s1="treasurer"; // 会计
        String s2="cashier"; // 出纳
        Bank bank = new Bank(s1,s2);
        bank.setMoney(120);

        Thread zhang;
        Thread cheng;
        zhang = new Thread(bank); // 目标对象bank
        cheng = new Thread(bank); // 目标对象bank
        zhang.setName(s1);
        cheng.setName(s2);
        zhang.start();
        cheng.start();
    }
}
```

```
treasurer: 110
cashier: 100
treasurer: 90
treasurer: Finished
cashier: 80
cashier: 70
cashier: 60
cashier: Finished
```

8.5 Runnable接口

- 下面的例子5中共有4个线程：threadA、threadB、threadC和threadD
  - threadA和threadB的目标对象a1
  - threadA和threadB共享a1的成员number
  - threadC和threadD的目标对象是a2
  - threadC和threadD共享a2的成员number

【例子5, 1/2】

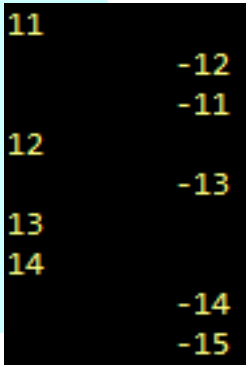
```
class TargetObject implements Runnable
{
    private int number = 0;
    public void setNumber(int n)
    {
        number = n;
    }
    public void run()
    {
        while(true)
        {
            if(Thread.currentThread().getName().equals("add"))
            {
                number++;
                System.out.printf("%d\n",number);
            }
            if(Thread.currentThread().getName().equals("sub"))
            {
                number--;
                System.out.printf("%12d\n",number);
            }
            try{ Thread.sleep(1000); }
            catch(InterruptedException e) {}
        }
    }
}
```

全局变量

【例子5, 2/2】

```
public class Example8_5
{
    public static void main(String args[])
    {
        TargetObject a1 = new TargetObject();
        a1.setNumber(10);
        TargetObject a2 = new TargetObject();
        a2.setNumber(-10);

        Thread threadA,threadB,threadC,threadD;
        threadA = new Thread(a1); // 目标对象a1
        threadB = new Thread(a1); // 目标对象a1
        threadA.setName("add");
        threadB.setName("add");
        threadC = new Thread(a2); // 目标对象a2
        threadD = new Thread(a2); // 目标对象a2
        threadC.setName("sub");
        threadD.setName("sub");
        threadA.start();
        threadB.start();
        threadC.start();
        threadD.start();
    }
}
```



8.5 Runnable接口

- 2.关于run()方法中的局部变量
- 对于具有相同**目标对象**的线程，当其中一个线程享用CPU资源时，目标对象自动调用接口中的run()方法，当轮到另一个线程享用CPU资源时，目标对象会再次调用接口中的run()方法。**不同线程的run()方法中的局部变量互不干扰**，一个线程改变了自己的run()方法中的局部变量的值不会影响其他线程的run()方法中的局部变量。

```
class Move implements Runnable
{
    String s1,s2;
    Move(String s1,String s2){this.s1=s1; this.s2=s2;}
    public void run(){
        int i=0;
        while(true){
            if(Thread.currentThread().getName().equals(s1)){
                i=i+1;
                System.out.println(s1 + ":" + i);
                if(i>=4){
                    System.out.println(s1 + "Finished");
                    return;
                }
            }
            else if(Thread.currentThread().getName().equals(s2)){
                i=i-1;
                System.out.println(s2 + ":" + i);
                if(i<=-4){
                    System.out.println(s2 + "Finished");
                    return;
                }
            }
            try{ Thread.sleep(800); }
            catch(InterruptedException e) {}
        }
    }
}
```

局部变量

8.5 Runnable接口

```
public class Example8_6
{
    public static void main(String args[])
    {
        String s1 = "ZHANG San";
        String s2 = "LI Si";
        Move move = new Move(s1,s2);

        Thread zhang, li;
        zhang = new Thread(move); // 目标对象move
        li = new Thread(move); // 目标对象move
        zhang.setName(s1);
        li.setName(s2);
        zhang.start();
        li.start();
    }
}
```

```
ZHANG San:1
LI Si:-1
LI Si:-2
ZHANG San:2
LI Si:-3
ZHANG San:3
LI Si:-4
ZHANG San:4
LI SiFinished
ZHANG SanFinished
```

Outline

- 8.1 Java中的线程
- 8.2 线程的生命周期
- 8.3 线程的优先级与调度管理
- 8.4 Thread的子类创建线程
- 8.5 Runnable接口
- 8.6 线程的常用方法
- 8.7 线程同步
- 8.8 使用wait(),notify(),notifyAll()协调同步线程
- 8.9 挂起、恢复和终止线程
- 8.10 线程联合
- 8.11 守护线程

8.6 线程的常用方法

- 1.start()
- 线程调用该方法将启动线程，使之从新建状态进入就绪队列排队，一旦轮到它来享用CPU资源时，就可以脱离创建它的主线程独立开始自己的生命周期了。

- 2.run()
- Thread类的run()方法与Runnable接口中的run()方法的功能和作用相同，都用来定义线程对象被调度之后所执行的操作，都是**系统自动调用而用户程序不得调用**的方法。

8.6 线程的常用方法

- 3.sleep(int millsecond)
- 优先级高的线程可以在它的run()方法中调用sleep()方法来使自己放弃CPU资源，休眠一段时间。

- 4.isAlive()
- 在线程的run()方法结束之前，即没有进入死亡状态之前，线程调用isAlive()方法返回**true**。当线程进入死亡状态后（实体内存被释放），线程仍可以调用方法isAlive()，这时返回的值是**false**。线程未调用start()方法之前，调用isAlive()方法返回**false**。

- 需要注意的是，**一个已经运行的线程在没有进入死亡状态时，不要再给线程分配实体**，由于线程只能引用最后分配的实体，先前的实体就会成为“垃圾”，并且不会被垃圾收集机收集掉（请看例子7）。

8.6 线程的常用方法

- 现在让我们看一个例子（例子7），一个线程每隔1秒钟在命令行窗口输出机器的当前时间，**在输出3秒之后，该线程又被分配了实体**，新实体又开始运行。这时，我们在命令行每秒钟能看见两行当前时间，**因为垃圾实体仍然在工作**。

8.6 线程的常用方法

- 【例子7, 1/2】

```
class A implements Runnable
{
    Thread thread;
    int n=0;
    A(){ thread=new Thread(this); }
    public void run()
    {
        while(true)
        {
            n++;
            System.out.println(new java.util.Date());
            try{ Thread.sleep(1000); }
            catch(InterruptedException e) {}
            if(n==3)
            {
                thread = new Thread(this);
                thread.start();
            }
        }
    }
}
```

8.6 线程的常用方法

- 【例子7, 2/2】

```
public class Example8_7
{
    public static void main(String args[])
    {
        A a = new A();
        a.thread.start();
    }
}
```

8.6 线程的常用方法

- 5.currentThread()
- currentThread()方法是Thread类中的**静态方法**，可以用类名调用，该方法返回当前正在使用CPU资源的线程。

- 6.interrupt()
- interrupt()方法经常用来“吵醒”休眠的线程。当一些线程调用sleep()方法处于休眠状态时，一个使用CPU资源的其它线程在执行过程中，可以**让休眠的线程分别调用interrupt()方法“吵醒”自己**，即导致休眠的线程发生InterruptedException异常，从而结束休眠，重新排队等待CPU资源。

8.6 线程的常用方法

- 在下面的例子8中，有3个线程：zhang、li和teacher
  - zhang和li准备休眠10秒钟后，再输出“Good morning”
  - teacher线程在输出3句“Let’s start…”后，“吵醒”休眠的线程

8.6 线程的常用方法

```
class ClassRoom implements Runnable{
    Thread teacher, zhang, li;
    ClassRoom(){
       teacher = new Thread(this); zhang = new Thread(this); li = new Thread(this);
       zhang.setName("Zhang"); li.setName("Li"); teacher.setName("Pan"); }

    public void run(){
       Thread thread = Thread.currentThread();
       if(thread==zhang || thread==li){
           try{  System.out.println(thread.getName() + ": Sleep for 10s");
                 Thread.sleep(10000); }
           catch(InterruptedException e){
                  System.out.println(thread.getName() + ": been wake up"); }
           System.out.println(thread.getName() + ": Good morning");
       }
       else if(thread==teacher){
           for(int i=1;i<=3;i++){
              System.out.println(thread.getName() + ": Let's start ...");
              try{  Thread.sleep(500); }
              catch(InterruptedException e) {}
           }
           zhang.interrupt();
           li.interrupt();
        }
    }
}
```

8.6 线程的常用方法

```
public class Example8_8
{
    public static void main(String args[])
    {
        ClassRoom room = new ClassRoom();
        room.zhang.start();
        room.li.start();
        room.teacher.start();
    }
}
```

```
Zhang: Sleep for 10s
Li: Sleep for 10s
Pan: Let's start ...
Pan: Let's start ...
Pan: Let's start ...
Zhang: been wake up
Zhang: Good morning
Li: been wake up
Li: Good morning
```

Outline

- 8.1 Java中的线程
- 8.2 线程的生命周期
- 8.3 线程的优先级与调度管理
- 8.4 Thread的子类创建线程
- 8.5 Runnable接口
- 8.6 线程的常用方法
- 8.7 线程同步
- 8.8 使用wait(),notify(),notifyAll()协调同步线程
- 8.9 挂起、恢复和终止线程
- 8.10 线程联合
- 8.11 守护线程

8.7 线程同步

- 线程同步是指多个线程要执行一个**synchronized修饰的方法**，如果一个线程A在占有CPU资源期间，使得synchronized方法被调用执行，那么在该synchronized方法返回之前（即synchronized方法调用执行完毕之前），**其他占有CPU资源的线程一旦调用这个synchronized方法就会引起堵塞，堵塞的线程要一直等到堵塞的原因消除（即synchronized方法返回）**，再排队等待CPU资源，以便使用这个同步方法。

8.7 线程同步

- 在下面的例子9中有两个线程： treasurer和cashier，他们共同拥有一个帐本。他们都可以使用saveOrTake(int number)对帐本进行访问，treasurer使用saveOrTake方法时，向帐本上写入存钱记录； cashier使用saveOrTake方法时，向帐本写入取钱记录。因此，当treasurer正在使用saveOrTake方法时，cashier被禁止使用，反之也是这样。

```
class Bank implements Runnable{
    int money = 300;
    String treasurerName, cashierName;
    public Bank(String s1,String s2){ treasurerName=s1; cashierName=s2; }
    public void run(){ saveOrTake(30); }
    public synchronized void saveOrTake(int number){
        if(Thread.currentThread().getName().equals(treasurerName)){
            for(int i=1;i<=3;i++){
                money = money + number;
                try { Thread.sleep(1000); }
                catch(InterruptedException e) {}
                System.out.println(treasurerName + " : " + money);
            }
        }
        else if(Thread.currentThread().getName().equals(cashierName)){
            for(int i=1;i<=2;i++){
                money = money-number/2;
                try{ Thread.sleep(1000);}
                catch(InterruptedException e){}
                System.out.println(cashierName + " : " + money);
            }
        }
    }
}
```

8.7 线程同步

```
public class Example8_9
{
    public static void main(String args[])
    {
        String treasurerName = "Treaurer", cashierName = "Cashier";
        Bank bank = new Bank(treasurerName,cashierName);

        Thread treasurer, cashier;
        treasurer = new Thread(bank); // 目标对象bank
        cashier = new Thread(bank); // 目标对象bank
        treasurer.setName(treasurerName);
        cashier.setName(cashierName);
        treasurer.start();
        cashier.start();
    }
}
```

Outline

- 8.1 Java中的线程
- 8.2 线程的生命周期
- 8.3 线程的优先级与调度管理
- 8.4 Thread的子类创建线程
- 8.5 Runnable接口
- 8.6 线程的常用方法
- 8.7 线程同步
- 8.8 使用wait(),notify(),notifyAll()协调同步线程
- 8.9 挂起、恢复和终止线程
- 8.10 线程联合
- 8.11 守护线程

8.8 使用wait(),notify(),notifyAll()协调同步线程

- wait(), notify()和notifyAll()都是Object类中的final方法，是被所有的类继承、且不允许重写的方法。

- 当一个线程使用同步方法中的某个变量，而此变量又需要其它线程修改后才能符合本线程的需要，那么可以在同步方法中使用wait()方法。使用wait()方法可以中断方法的执行，使本线程等待，暂时让出CPU的使用权，并允许其它线程使用这个同步方法。其它线程如果在使用这个同步方法时不需要等待，那么它使用完这个同步方法的同时，应当用notifyAll()方法通知所有由于使用这个同步方法而处于等待的线程结束等待。

8.8 使用wait(),notify(),notifyAll()协调同步线程

- 在下面的例子10中，模拟3个人排队买票，每人买1张票。售票员只有1张五元的钱，电影票五元钱一张。
- Zhang拿1张二十元的人民币排在Sun前面买票，Sun拿1张十元的人民币排在Zhao的前面买票，Zhao拿1张五元的人民币排在最后。那么，最终的卖票次序应当是Sun、Zhao、Zhang 。

【例子10，1/3】

```
class TicketSeller
{
    int fiveNumber=1, tenNumber=0, twentyNumber=0;
    public synchronized void sellTicket(int receiveMoney)
    {
        String s=Thread.currentThread().getName();
        if(receiveMoney==5)
        {
           fiveNumber = fiveNumber+1;
           System.out.println(s + " gives $5 to seller, seller gives " + s + " a ticket");
        }
        else if(receiveMoney==10){
           while(fiveNumber<1){
               try{  System.out.println(s + " gives $10 to seller");
                     System.out.println("seller asks " + s + " to wait");
                     wait();
                     System.out.println(s + " stops waiting and starts to buy...");
               }
               catch(InterruptedException e){}
           }
           fiveNumber=fiveNumber-1;
           tenNumber=tenNumber+1;
           System.out.println(s + " gives $10 to seller, seller gives " + s + " a ticket and $5");
        }
        else if(receiveMoney==20){
           while(fiveNumber<1||tenNumber<1){
               try{  System.out.println(s + " gives $20 to seller");
                     System.out.println("seller asks " + s + " to wait");
                     wait();
                     System.out.println(s+" stops waiting and starts to buy ...");
               }
               catch(InterruptedException e){}
           }
           fiveNumber = fiveNumber-1;
           tenNumber = tenNumber-1;
           twentyNumber = twentyNumber+1;
           System.out.println(s + " gives $20 to seller, seller gives " + s + " a ticket and $15");
        }
        notifyAll();
    }
}
```

【例子10，2/3】

```
class Cinema implements Runnable
{
    TicketSeller seller;
    String name1, name2, name3;
    Cinema(String s1,String s2,String s3)
    {
       seller = new TicketSeller();
       name1 = s1;
       name2 = s2;
       name3 = s3;
    }
    public void run()
    {
       if(Thread.currentThread().getName().equals(name1))
       {
          seller.sellTicket(20);
       }
       else if(Thread.currentThread().getName().equals(name2))
       {
          seller.sellTicket(10);
       }
       else if(Thread.currentThread().getName().equals(name3))
       {
          seller.sellTicket(5);
       }
    }
}
```

8.8 使用wait(),notify(),notifyAll()协调同步线程

```
public class Example8_10
{
    public static void main(String args[])
    {
        String s1="Zhang", s2="Sun", s3="Zhao";
        Cinema cinema = new Cinema(s1,s2,s3);

        Thread zhang, sun, zhao;
        zhang = new Thread(cinema); // 目标对象cinema
        sun = new Thread(cinema); // 目标对象cinema
        zhao = new Thread(cinema); // 目标对象cinema
        zhang.setName(s1);
        sun.setName(s2);
        zhao.setName(s3);
        zhang.start();
        sun.start();
        zhao.start();
    }
}
```

```
Zhang gives $20 to seller
seller asks Zhang to wait
Sun gives $10 to seller, seller gives Sun a ticket and $5
Zhang stops waiting and starts to buy ...
Zhang gives $20 to seller
seller asks Zhang to wait
Zhao gives $5 to seller, seller gives Zhao a ticket
Zhang stops waiting and starts to buy ...
Zhang gives $20 to seller, seller gives Zhang a ticket and $15
```

Outline

- 8.1 Java中的线程
- 8.2 线程的生命周期
- 8.3 线程的优先级与调度管理
- 8.4 Thread的子类创建线程
- 8.5 Runnable接口
- 8.6 线程的常用方法
- 8.7 线程同步
- 8.8 使用wait(),notify(),notifyAll()协调同步线程
- 8.9 挂起、恢复和终止线程
- 8.10 线程联合
- 8.11 守护线程

8.9 挂起、恢复和终止线程

- 在下面的例子11中，线程thread每隔一秒钟输出一个整数，输出3个整数后，该线程**挂起**；主线程负责**恢复**thread线程继续执行。

8.9 挂起、恢复和终止线程

- 【例子11，1/2】

```
class A implements Runnable{
    int number = 0;
    boolean stop = false;
    boolean getStop(){ return stop; }
    public void run(){
        while(true){
            number++;
            System.out.println(Thread.currentThread().getName() + " : " + number);
            if(number==3){
                try{  System.out.println(Thread.currentThread().getName() + " : hang up");
                stop = true;
                hangUP();
                System.out.println(Thread.currentThread().getName() + " : resumed");
                }
                catch(Exception e){}
            }
            try{ Thread.sleep(1000); }
            catch(Exception e){}
        }
    }
    public synchronized void hangUP() throws InterruptedException{ wait(); }
    public synchronized void restart(){ notifyAll(); }
}
```

8.9 挂起、恢复和终止线程

- 【例子11，2/2】

```
public class Example8_11
{
    public static void main(String args[])
    {
        A target = new A();
        Thread thread = new Thread(target);
        thread.setName("Zhang San");
        thread.start();

        while(target.getStop()==false){}

        System.out.println("Main Thread");
        target.restart();
    }
}
```

8.9 挂起、恢复和终止线程

- 在下面的例子12中，thread是用Thread的子类创建的对象，每隔一秒钟输出一个整数，输出3个整数后，该线程**挂起**。主线程负责**恢复**thread线程继续执行。

8.9 挂起、恢复和终止线程

- 【例子12，1/2】

```
class MyThread extends Thread{
    int number = 0;
    boolean stop = false;
    boolean getStop(){ return stop; }
    public void run(){
        while(true){
            number++;
            System.out.println(Thread.currentThread().getName() + " : " + number);
            if(number==3){
                try{
                    System.out.println(Thread.currentThread().getName() + " : hang up");
                    stop = true;
                    hangUP();
                    System.out.println(Thread.currentThread().getName() + " : resumed");
                }
                catch(Exception e){}
            }
            try{ Thread.sleep(1000); }
            catch(Exception e){}
        }
    }
    public synchronized void hangUP() throws InterruptedException{ wait(); }
    public synchronized void restart(){ notifyAll(); }
}
```

8.9 挂起、恢复和终止线程

- 【例子12，2/2】

```
public class Example8_12
{
    public static void main(String args[])
    {
        MyThread thread = new MyThread();
        thread.setName("Zhang San");
        thread.start();
        while(thread.getStop()==false) {}
        System.out.println("Main Thread");
        thread.restart();
    }
}
```

Outline

- 8.1 Java中的线程
- 8.2 线程的生命周期
- 8.3 线程的优先级与调度管理
- 8.4 Thread的子类创建线程
- 8.5 Runnable接口
- 8.6 线程的常用方法
- 8.7 线程同步
- 8.8 使用wait(),notify(),notifyAll()协调同步线程
- 8.9 挂起、恢复和终止线程
- 8.10 线程联合
- 8.11 守护线程

8.10 线程联合

- 一个线程A在占有CPU资源期间，可以让线程B调用join()方法和本线程联合，如：B.join(); 我们称**A在运行期间联合了B**。

- 如果线程A在占有CPU资源期间一旦联合B线程，那么A线程将**立刻中断执行**，一直等到它联合的线程B执行完毕，A线程再重新排队等待CPU资源，以便恢复执行。如果A准备联合的B线程已经结束，那么B.join()不会产生任何效果。
- 在下面的例子13中，一个线程在运行期间联合了另外一个线程。

【例子13，1/2】

```
class JoinThread implements Runnable
{
    Thread threadA, threadB;
    String content[] = {"今天晚上,", "大家不要", "回去的太早,", "还有工作", "需要大家做!"};
    JoinThread()
    {
        threadA=new Thread(this);
        threadB=new Thread(this);
        threadB.setName("经理");
    }

    public void run()
    {
        if(Thread.currentThread()==threadA){
            System.out.println("我等"+threadB.getName()+"说完再说话");
            threadB.start();
            while(threadB.isAlive()==false){}

            try{ threadB.join(); }
            catch(InterruptedException e){}
            System.out.printf("\n我开始说话:\"我明白你的意思了,谢谢\"");
        }
        else if(Thread.currentThread()==threadB){
            System.out.println(threadB.getName()+"说:");
              for(int i=0;i<content.length;i++){
                 System.out.print(content[i]);
                 try { threadB.sleep(1000); }
                 catch(InterruptedException e){}
              }
        }
    }
}
```

【例子13，2/2】

```
public class Example8_13
{
    public static void main(String args[])
    {
        JoinThread a = new JoinThread();
        a.threadA.start();
    }
}
```

```
我等经理说完再说话
经理说：
今天晚上，大家不要回去的太早，还有工作需要大家做！
我开始说话："我明白你的意思了，谢谢"
```

Outline

- 8.1 Java中的线程
- 8.2 线程的生命周期
- 8.3 线程的优先级与调度管理
- 8.4 Thread的子类创建线程
- 8.5 Runnable接口
- 8.6 线程的常用方法
- 8.7 线程同步
- 8.8 使用wait(),notify(),notifyAll()协调同步线程
- 8.9 挂起、恢复和终止线程
- 8.10 线程联合
- 8.11 守护线程

8.11 守护线程

- 一个线程调用void setDaemon(boolean on)方法可以将自己设置成一个守护（daemon）线程，例如：thread.setDaemon(true);

- 线程默认是非守护线程，非守护线程也称作用户（user）线程。

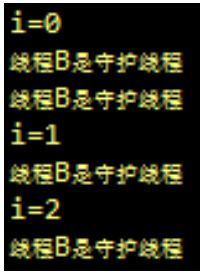
- 当程序中的所有用户线程都已结束运行时，即使守护线程的run()方法中还有需要执行的语句，守护线程也立刻结束运行。一般地，用守护线程做一些不是很严格的工作，线程的随时结束不会产生什么不良的后果。一个线程必须在运行之前设置自己是否是守护线程。

```
class Daemon implements Runnable{
    Thread A,B;
    Daemon(){
        A = new Thread(this);
        B = new Thread(this);
    }
    public void run(){
        if(Thread.currentThread()==A){
            for(int i=0;i<3;i++){
                System.out.println("i="+i) ;
                try { Thread.sleep(1000); }
                catch(InterruptedException e){}
            }
        }
        else if(Thread.currentThread()==B){
            while(true){
                System.out.println("线程B是守护线程 ");
                try{ Thread.sleep(1000); }
                catch(InterruptedException e){}
            }
        }
    }
}
```

```
public class Example8_14
{
    public static void main(String args[])
    {
        Daemon a=new Daemon ();

        a.A.start();

        a.B.setDaemon(true);
        a.B.start();
    }
}
```



Outline

- 8.1 Java中的线程
- 8.2 线程的生命周期
- 8.3 线程的优先级与调度管理
- 8.4 Thread的子类创建线程
- 8.5 Runnable接口
- 8.6 线程的常用方法
- 8.7 线程同步
- 8.8 使用wait(),notify(),notifyAll()协调同步线程
- 8.9 挂起、恢复和终止线程
- 8.10 线程联合
- 8.11 守护线程

习题

- 1、什么是线程？什么是多线程？应用程序中的多线程有什么作用？
- 2、Java为线程机制提供了什么类与接口？
- 3、编写一个线程，其任务是让一个字符串从屏幕左端向右移动，当所有的字符都消失后，字符串重新从左边出现并继续向右移动。
- 4、线程有哪5种基本状态？它们之间是如何转化的？
- 5、线程的方法sleep()与wait()有什么区别？
- 6、什么是线程调度？Java的线程调度采用什么策略？
- 7、编写程序实现如下功能：第一个线程生成一个随机数，第二个线程每隔一段时间读取第一个线程生成的随机数，并判断它是否是素数。
- 8、请编写程序模拟龟兔赛跑。要求用一个线程控制龟的运动，用另一个线程控制兔的运动。龟兔均在同一个运动场上赛跑，要求可以设置龟兔完成一圈所需要的时间，而且要求设置兔比龟跑得快。在赛跑最开始，龟兔在同一个起点出发。

补充知识: Thread Pools (1/2)

```
public class TaskThreadDemo {
  public static void main(String[] args) {
    // Create tasks
    Runnable printA = new PrintChar('a', 100);
    Runnable printB = new PrintChar('b', 100);
    Runnable print100 = new PrintNum(100);

    // Create threads
    Thread thread1 = new Thread(printA);
    Thread thread2 = new Thread(printB);
    Thread thread3 = new Thread(print100);

    // Start threads
    thread1.start();
    thread2.start();
    thread3.start();
  }
}
```

Step 2

Step 3

Step 4

This approach is convenient for a single task execution, but it is not efficient for a large number of tasks because you have to create a thread for each task. Starting a new thread for each task could limit throughput and cause poor performance.

Step 1

```
// The task for printing a character a specified number of times
class PrintChar implements Runnable {
  private char charToPrint; // The character to print
  private int times; // The number of times to repeat

  /** Construct a task with a specified character and number of
   *  times to print the character
   */
  public PrintChar(char c, int t) {
    charToPrint = c;
    times = t;
  }

  @Override /** Override the run() method to tell the system
   *  what task to perform
   */
  public void run() {
    for (int i = 0; i < times; i++) {
      System.out.print(charToPrint);
    }
  }
}

// The task class for printing numbers from 1 to n for a given n
class PrintNum implements Runnable {
  private int lastNum;

  /** Construct a task for printing 1, 2, ..., n */
  public PrintNum(int n) {
    lastNum = n;
  }

  @Override /** Tell the thread how to run */
  public void run() {
    for (int i = 1; i <= lastNum; i++) {
      System.out.print(" " + i);
    }
  }
}
```

补充知识: Thread Pools (2/2)

- Using a *thread pool* is an ideal way to manage the number of tasks executing concurrently.
- Java provides the **Executor** interface for executing tasks in a thread pool and the **ExecutorService** interface for managing and controlling tasks. ExecutorService is a subinterface of Executor.

```
import java.util.concurrent.*;

public class ExecutorDemo {
  public static void main(String[] args) {
    // Create a fixed thread pool with maximum three threads
    ExecutorService executor = Executors.newFixedThreadPool(3);

    // Submit runnable tasks to the executor
    executor.execute(new PrintChar('a', 100));
    executor.execute(new PrintChar('b', 100));
    executor.execute(new PrintNum(100));

    // Shut down the executor
    executor.shutdown();
  }
}
```

- **如果设为1**: 三个task顺序执行
- **如果Executors.newCachedThreadPool();**: 三个task并行执行

No new tasks can be accepted, but any existing tasks will continue to finish.

Tips: If you need to create a thread for just one task, use the Thread class. If you need to create threads for **multiple tasks, it is better to use a thread pool**.

补充知识: Synchronization Using Locks (1/2)

```
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

public class AccountWithSyncUsingLock {
  private static Account account = new Account();

  public static void main(String[] args) {
    ExecutorService executor = Executors.newCachedThreadPool();

    // Create and launch 100 threads
    for (int i = 0; i < 100; i++) {
      executor.execute(new AddAPennyTask());
    }

    executor.shutdown();

    // Wait until all tasks are finished
    while (!executor.isTerminated()) {
    }

    System.out.println("What is balance? " + account.getBalance());
  }

  // A thread for adding a penny to the account
  public static class AddAPennyTask implements Runnable {
    public void run() {
      account.deposit(1);
    }
  }
```

静态内部类使用场景：一般是当外部类需要使用内部类，而内部类无需外部类资源，并且内部类可以单独创建的时候。

```
// An inner class for Account
public static class Account {
  private static Lock lock = new ReentrantLock(); // Create a lock
  private int balance = 0;

  public int getBalance() {
    return balance;
  }

  public void deposit(int amount) {
    lock.lock(); // Acquire the lock

    try {
      int newBalance = balance + amount;

      // This delay is deliberately added to magnify the
      // data-corruption problem and make it easy to see.
      Thread.sleep(5);

      balance = newBalance;
    }
    catch (InterruptedException ex) {
    }
    finally {
      lock.unlock(); // Release the lock
    }
  }
}
}
```

补充知识: Synchronization Using Locks (2/2)

- A ***synchronized*** instance method **implicitly** acquires a lock on the instance before it executes the method.

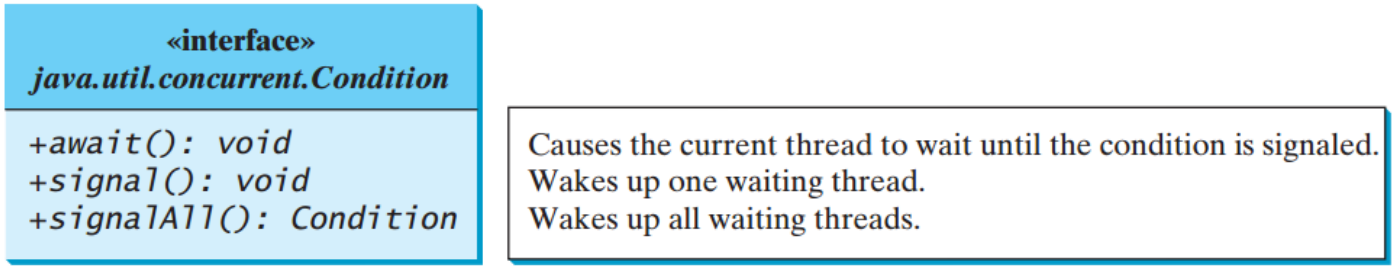
- Java enables you to **acquire locks explicitly**, which give you more control for coordinating threads. A lock is an instance of the **Lock** interface, which defines the methods for acquiring and releasing locks.
- **ReentrantLock** is a concrete implementation of **Lock** for creating mutually exclusive locks.

- In general, using **synchronized** methods or statements is **simpler** than using explicit locks for mutual exclusion. However, **using explicit locks is more intuitive and flexible** to synchronize threads with conditions.

补充知识: Cooperation among Threads (1/3)

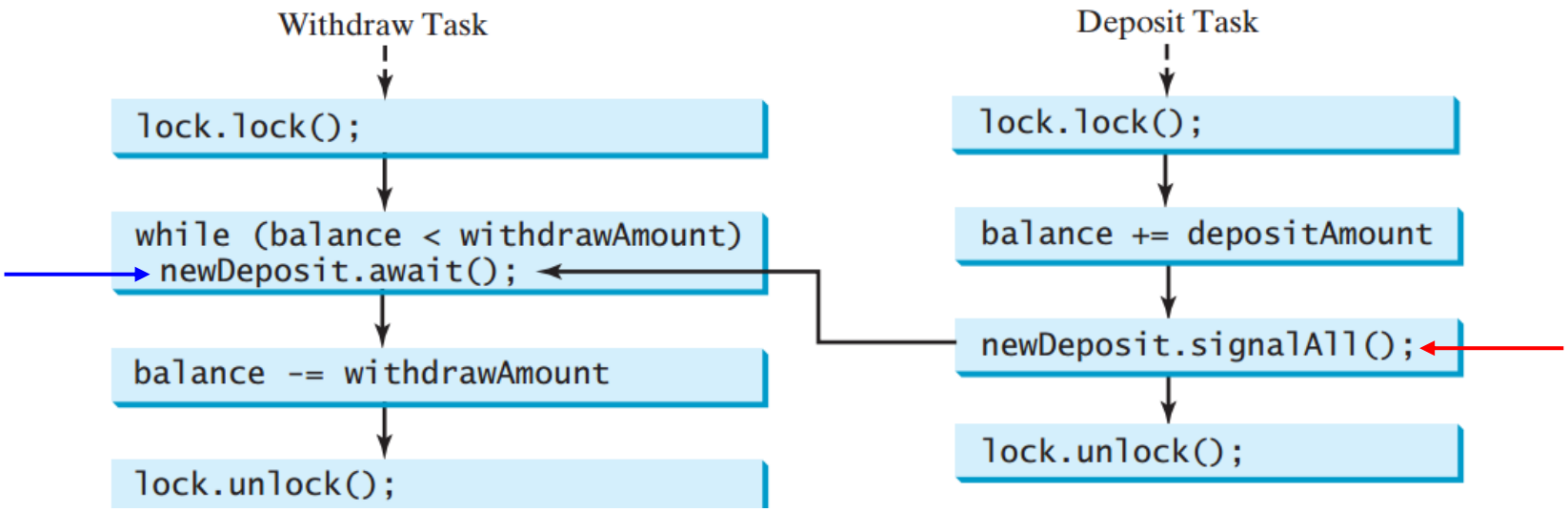
- Thread synchronization suffices to avoid race conditions by ensuring the mutual exclusion of multiple threads in the critical region, but sometimes you also need a way for threads to **cooperate**.

- Conditions can be used to facilitate communications among threads. A thread can specify what to do under a certain condition.
- Conditions are objects (对象) created by invoking the **newCondition()** method on a **Lock** object.
- Once a condition is created, you can use its await(), signal(), and signalAll() methods for thread communications



补充知识: Cooperation among Threads (2/3)

- 例: 取款(withdraw)和存款(deposit)



补充知识: Cooperation among Threads (3/3)

```
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

public class ThreadCooperation {
  private static Account account = new Account();

  public static void main(String[] args) {
    // Create a thread pool with two threads
    ExecutorService executor = Executors.newFixedThreadPool(2);
    executor.execute(new DepositTask());
    executor.execute(new WithdrawTask());
    executor.shutdown();

    System.out.println("Thread 1\t\tThread 2\t\tBalance");
  }

  public static class DepositTask implements Runnable {
    @Override // Keep adding an amount to the account
    public void run() {
      try { // Purposely delay it to let the withdraw method proceed
        while (true) {
          account.deposit((int)(Math.random() * 10) + 1);
          Thread.sleep(1000);
        }
      }
      catch (InterruptedException ex) {
        ex.printStackTrace();
      }
    }
  }

  public static class WithdrawTask implements Runnable {
    @Override // Keep subtracting an amount from the account
    public void run() {
      while (true) {
        account.withdraw((int)(Math.random() * 10) + 1);
      }
    }
  }
```

```
// An inner class for account
private static class Account {
  // Create a new lock
  private static Lock lock = new ReentrantLock();

  // Create a condition
  private static Condition newDeposit = lock.newCondition();

  private int balance = 0;

  public int getBalance() {
    return balance;
  }

  public void withdraw(int amount) {
    lock.lock(); // Acquire the lock
    try {
      while (balance < amount) {
        System.out.println("\t\t\tWait for a deposit");
        newDeposit.await();
      }

      balance -= amount;
      System.out.println("\t\t\tWithdraw " + amount +
        "\t\t" + getBalance());
    }
    catch (InterruptedException ex) {
      ex.printStackTrace();
    }
    finally {
      lock.unlock(); // Release the lock
    }
  }

  public void deposit(int amount) {
    lock.lock(); // Acquire the lock
    try {
      balance += amount;
      System.out.println("Deposit " + amount +
        "\t\t\t\t" + getBalance());

      // Signal thread waiting on the condition
      newDeposit.signalAll();
    }
    finally {
      lock.unlock(); // Release the lock
    }
  }
}
```